Zusammenfassung

Theorie des Logikentwurfs

Michael Gregorius

Version vom 16. September 2003

http://www.michaelgregorius.de/

Inhaltsverzeichnis

1 Boolesche Funktionen, Schaltkreise, Kostenmaße

1.1 Schaltkreise/Straight Line Programs, die Basen B_2 und U, Komplexitätsmaße von Schaltkreisen

Frage 1.1: Was ist eine Boolesche Funktion? Welche Mengen von Booleschen Funktionen kennst du?

Eine Boolesche Funktion ist im Prinzip eine Funktion, die einen Bitvektor mit n Bits auf einen Bitvektor mit m Bits abbildet, also:

$$f: \{0,1\}^n \to \{0,1\}^m$$

Durch eine solche Boolesche Funktion kann also zum Beispiel ein Schaltkreis beschrieben werden, der über n Eingänge und m Ausgänge verfügt. Mit $B_{n,m}$ bezeichnen wir die Menge aller solcher Funktionen. Ein Spezialfall ist die Menge $B_{n,1}$ die auch einfach mit B_n bezeichnet wird.

Frage 1.2: Was ist ein Straight-Line Programm (SLP)?

Ein Straight-Line Programm wird durch die folgenden Angaben beschrieben:

1. Ein Eingabealphabet I (z.B. $I = \{0,1\}$) und ein Ausgabealphabet O (für das meist $O = I$ gilt).
2. Die Eingabelänge n und die Ausgabelänge m.
3. Eine Menge E von Basisoperationen. E wird auch Basis genannt.
4. Eine Folge von Rechenschritten $G_1, \ldots, G_c$, wobei ein Rechenschritt G_i gegeben ist durch:
   - Elementaroperation $g_i \in E$
   - Eine geordnete Folge von Eingängen für g_i. Dies können Konstanten, Eingabevariablen oder schon berechnete Ergebnisse sein.
5. Beschreibung der Ausgabe.

Frage 1.3: Gebe ein Beispiel für ein SLP!

Das folgende SLP beschreibt einen Volladdierer, der später noch etwas genauer beschrieben wird. Die Eingabe besteht aus den drei Bits x_1, x_2, x_3, die Ausgabe sind die Bits x_{10}, x_5:

1. $x_4 = x_1 \oplus x_2$
2. $x_5 = x_4 \oplus x_3$
3. $x_6 = x_1 \wedge x_2$
4. $x_7 = x_1 \wedge x_3$
5. $x_8 = x_2 \wedge x_3$
6. $x_9 = x_6 \vee x_7$

7. $x_{10} = x_9 \vee x_8$

Frage 1.4: Welche Spezialfälle von SLPs kennst du?
Man kann zwei Spezialfälle von SLPs unterscheiden:

(Boolesche) Schaltkreise: Dies sind SLPs über dem Alphabet $I = O = \{0, 1\}$ mit der Basis $E = B_2$, d.h. alle Funktionen $\{0, 1\}^2 \to \{0, 1\}$.

Arithmetische Schaltkreise: SLPs mit Alphabet $I = \mathbb{Z}$, also der Menge der ganzen Zahlen, und Basis $E = \{+, -, *, /\}$, wobei / die ganzzahlige Division ohne Rest ist.

Frage 1.5: Welches sind die wichtigsten Funktionen aus B_2?
Die wichtigsten Funktionen sind $\wedge, \vee, \neg, \oplus \in B_2$, welche wie folgt definiert sind:

Konjunktion: Die Konjunktion $\wedge$ von x und y, wird genau dann 1, wenn beide Eingaben 1 sind:

$$\wedge(x, y) = 1 \;\Leftrightarrow\; x = 1 \text{ und } y = 1$$

Die Schreibweise ist entweder $x \wedge y$ oder einfach xy.

Disjunktion: Die Disjunktion $\vee$ von x und y (Schreibweise $x \vee y$) wird genau dann 1, wenn mindestens eine Eingabe 1 ist:

$$\vee(x, y) = 1 \;\Leftrightarrow\; x = 1 \text{ oder } y = 1$$

Negation: Die Negation $\neg$ einer Variablen x ist genau dann 1, wenn $x = 0$ ist:

$$\neg x = 1 \;\Leftrightarrow\; x = 0$$

Parität: Die Parität $\oplus$ (auch Exclusive-Or, XOR oder mod-2-Summe) ist genau dann 1, wenn genau eine der Eingaben 1 ist:

$$x \oplus y = 1 \;\Leftrightarrow\; (x, y) \in \{(0, 1), (1, 0)\}$$

Frage 1.6: Wie können Schaltkreise dargestellt werden?
Schaltkreise können auch in Form eines gerichteten, kreisfreien Graph (directed acyclic graph oder auch DAG) dargestellt werden. Dabei gibt es Eingabeknoten mit einem Fan-in von 0 sowie Ausgabeknoten. Die inneren Knoten sind mit der jeweiligen Elementaroperation $g_i \in E$ gekennzeichnet, wobei sich der Fan-in des Knotens aufgrund der Stelligkeit der Elementaroperation ergibt. Abbildung 1 zeigt als Beispiel den Schaltkreis eine Volladdierers, wie er auch schon etwas weiter oben kurz vorgestellt wurde.

Bei der Darstellung von Schaltkreisen in Form von gerichteten, kreisfreien Graphen wird besonders deutlich, dass es sich um ein paralleles Berechnungsmodell handelt, was aus einem SLP nicht so schnell ersichtlich wird.

Frage 1.7: Welche Komplexitätsmaße für Schaltkreise kennst du?
Es kann grob zwischen der Größe und der Tiefe eines Schaltkreises S bzw. einer Booleschen Funktion f unterschieden werden:

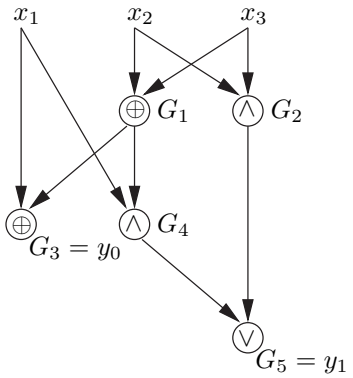


Abbildung 1: Schaltkreis eines Volladdierers

Größe: Die Größe $C(S)$ eines Schaltkreises S gibt die Anzahl der verwendeten Bausteine an, wobei die Anzahl der Bausteine dabei in etwa der Anzahl der verwendeten Elementaroperationen entspricht. Im Beispiel für das SLP würde z.B. $C(S) = 7$ gelten, während Abbildung 1 einen minimalen Schaltkreis für einen Volladdierer (Redkin, 1981) mit $C(S) = 5$ zeigt.

Es kann auch die Schaltkreiskomplexität einer Booleschen Funktion f angegeben werden. Dabei ist $C(f)$ die minimale Anzahl an Bausteinen in Schaltkreisen, die f berechnen. Mit der Größe eines Schaltkreises bzw. einer Funktion, können also zum einen die Hardware-Kosten abgeschätzt werden, die auftreten, um den Schaltkreis bzw. die Funktion zu realisieren. Gleichzeitig ist die Größe jedoch auch ein Maß für die sequentielle Rechenzeit.

Tiefe: Die Tiefe $D(S)$ ist definiert als die größte Anzahl von Bausteinen auf einem gerichteten Weg in S. Die Tiefe einer Booleschen Funktion f ist entsprechend als die minimale Tiefe aller Schaltkreise, die f berechnen, definiert.

Die Tiefe ist also ein Maß für die parallele Rechenzeit. Entsprechend der These „Speicherplatzbedarf ist Parallelzeit“ wird mit der Tiefe auch der Speicherplatz eines Schaltkreises bzw. einer Funktion gemessen.

Frage 1.8: Wann heißt ein Schaltkreis synchron?

Ein Schaltkreis heißt synchron, wenn alle gerichteten Wege zu einem Baustein die gleiche Länge haben. Der Schaltkreis in Abbildung 1 ist nicht sychron, da z.B. der Baustein G_3 einen Zeittakt warten muss, weil er das Ergebnis des Bausteins G_1 benötigt. Man kann sich einen sychronen Schaltkreis also wie einen Schaltkreis vorstellen, bei dem nur die Ergebnisse der vorherigen Ebene benutzt werden und mit diesen die Ergebnisse für die nächste Ebene berechnet werden.

Ein nicht synchroner Schaltkreis kann synchronisiert werden, indem sogenannte Delay- oder Verzögerungsbausteine eingebaut werden. Wenn wir nochmal Abbildung 1 betrachten, muss also auf allen Kanten, die eine Ebene überspringen, ein Delay eingebaut werden. Die Ausgabe eines Bausteins G_i kann z.B. um einen Zeittakt verzögert werden, indem ein neuer $\wedge$-Baustein G' eingebaut wird, dessen Eingänge beide G_i sind. Dadurch wird an G' dasselbe wie an G_i berechnet. Wir können auch die Größe eines solchen resultierenden, synchronen

Schaltkreises S' abschätzen. Wir überlegen uns zuerst, wieviele bzw. welche Komponenten eines Schaltkreises S eventuell verzögert werden müssen. Dies sind neben den n Eingabebits noch die zwei Konstanten 0 und 1 sowie alle Bausteine, wovon $C(S)$ gegeben sind. Wieviele Zeittakte müssen diese Komponenten nun höchstens verzögert werden. Auf dem längsten Weg in S kommen $D(S)$ Bausteine vor. Wenn wir am letzten Baustein sind, haben wir es jedoch schon „geschafft". Also muss jede Komponente von S höchstens um $D(S)-1$ Takte verzögert werden. Damit gilt also für die Größe und Tiefe von S':

$$C(S') \leq (C(S)+n+2)\cdot D(S) \qquad \text{und} \qquad D(S') = D(S)$$

Frage 1.9: Welche Probleme kann es bei Schaltkreisen mit beliebigem Fan-out geben? Sind diese kritisch?

In unserer bisherigen Definition konnten die Bausteinen einen beliebig hohen Fan-out aufweisen, d.h. das Ergebnis eines Bausteins konnte von beliebig vielen anderen Bausteinen genutzt werden. In der Praxis unterliegt der Fan-out eines Bausteins jedoch technologischen Schranken. Dies ist jedoch nicht kritisch, da Hoover, Klawe und Pippenger gezeigt haben, dass ein Schaltkreis S mit unbegrenztem Fan-out durch einen Schaltkreis S' mit durch 2 begrenzten Fan-out simuliert werden kann, wobei für die Größe und Tiefe von S' gilt, dass diese höchstens doppelt so groß wird wie die Größe und Tiefe von S:

$$C(S') \leq 2\cdot C(S) \qquad \text{und} \qquad D(S') \leq 2\cdot D(S)$$

Die Schaltkreise werden also nur um einen konstanten Faktor größer, weshalb man sich, von einem theoretischen Standpunkt aus betrachtet, also keine Gedanken machen muss.

1.2 Rechenregeln und Normalformen

Frage 1.10: Wieviele Boolesche Funktionen gibt es in $B_{n,m}$? Wieviele gibt es in B_n? Welche Konsequenz ergeben sich daraus?

In $B_{n,m}$ gibt es 2^{m2^n} Funktionen, d.h. $|B_{n,m}| = 2^{m2^n}$. Wie kann man sich dies nun klar machen? Es gilt:

$$B_{n,m} = \{f \mid f : \{0,1\}^n \to \{0,1\}^m\}$$

Die Frage ist also, wieviele verschiedene Abbildungen $f : \{0,1\}^n \to \{0,1\}^m$ es gibt. Es gibt 2^n Bitvektoren, für die es jeweils 2^m verschiedene Bilder gibt. Wir haben also 2^n mal die Auswahl zwischen 2^m Bildern, wobei verschiedene Bitvektoren $x, y \in \{0,1\}^n$ auch dasselbe Bild $f(x) = f(y)$ haben dürfen. Die Anzahl der möglichen Kombinationen ist dementsprechend:

$$(2^m)^{2^n} = 2^{m2^n}$$

Die Menge B_n ist nichts anderes als die Menge $B_{n,1}$. Dementsprechend gilt $|B_n| = 2^{2^n}$. Die Anzahl der Funktionen wächst für wachsendes n sehr schnell. So ist $|B_2| = 16$, $|B_3| = 256$, $|B_4| = 65536$, $|B_5| > 4\cdot 10^9$ und $|B_6| > 16\cdot 10^{18}$. Die Anzahl der Funktionen in B_6 ist also schon so groß, dass es nicht möglich ist, in „vernünftiger" Zeit alle Funktionen in B_6 zu untersuchen. Eine der Konsequenzen aus diesem Ergebnis ist die Tatsache, dass man also, wenn eine minimaler Schaltkreis zu einer Funktion f gesucht ist, nicht einfach alle Schaltkreise aufzählen und aus diesen den minimalen Schaltkreis, der f realisiert, auswählen kann. Oder anders gesagt: „Man hat nicht die Möglichkeit, alle Möglichkeiten auszuprobieren."

Frage 1.11: Welche Rechenregeln gibt es für Boolesche Ausdrücke? Wie werden sie bewiesen?

Rechnen mit Konstanten: Beim Rechnen mit Konstanten gelten die folgenden Regeln:

$$x \vee 0 = x,\ x \vee 1 = 1,\ x \wedge 0 = 0,\ x \wedge 1 = x,\ x \oplus 0 = x,\ x \oplus 1 = \overline{x}$$

Assoziativ- und Kommutativgesetze: $\wedge, \vee$ und $\oplus$ sind assoziativ und kommutativ, d.h. es gilt z.B.:

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z \qquad \text{und} \qquad x \wedge y = y \wedge x$$

Distributivgesetze: $(\wedge, \vee)$, $(\vee, \wedge)$ und $(\oplus, \wedge)$ sind distributiv, d.h.:

- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- $x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$

Vereinfachungsregeln: Es gelten die folgenden Vereinfachungsregeln:

- $x \vee x = x,\ x \vee \overline{x} = 1$
- $x \wedge x = x,\ x \wedge \overline{x} = 0$
- $x \oplus x = 0,\ x \oplus \overline{x} = 1$
- $x \vee (x \wedge y) = x,\ x \wedge (x \vee y) = x$

deMorgan-Regeln: Es gelten die deMorgan-Regeln:

$$\neg(x_1 \vee \cdots \vee x_n) = \overline{x}_1 \wedge \cdots \wedge \overline{x}_n \qquad \text{und} \qquad \neg(x_1 \wedge \cdots \wedge x_n) = \overline{x}_1 \vee \cdots \vee \overline{x}_n$$

Die ersten vier Regeln werden durch vollständige Fallunterscheidungen bewiesen. Das bedeutet, dass eine Tabelle mit allen Kombinationen von Belegungen erstellt wird, anhand der gezeigt wird, dass stets die gleichen Ergebnisse herauskommen. Die Regel von deMorgan gilt dagegen für beliebige n, wofür sich ein Beweis per vollständiger Induktion anbietet.

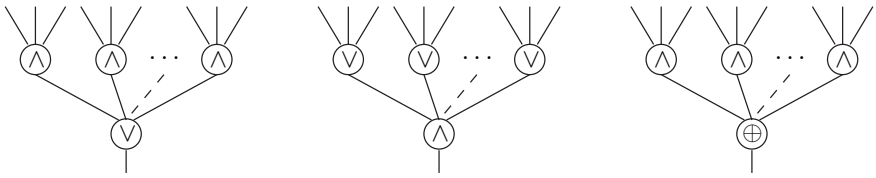
Frage 1.12: Beweise die Regel von deMorgan!

Wir beweisen nur eine der beiden Regeln, da das Vorgehen bei der anderen Regel entsprechend ist. Der Induktionsanfang besteht darin, für $n = 2$ mit einer vollständigen Fallunterscheidung zu zeigen, dass die Regel von deMorgan gilt. Die Induktionsvoraussetzung ist dann, dass die Behauptung für $n - 1$ und kleinere Werte gelte. Der Induktionsschritt wird dann für n unter Ausnutzung der Assoziativität wie folgt gezeigt:

$$\begin{aligned} \neg(x_1 \vee \cdots \vee x_n) &= \neg((x_1 \vee \cdots \vee x_{n-1}) \vee x_n) = \neg(x_1 \vee \cdots \vee x_{n-1}) \wedge \overline{x}_n \\ &= (\overline{x}_1 \wedge \cdots \wedge \overline{x}_{n-1}) \wedge \overline{x}_n = \overline{x}_1 \wedge \cdots \wedge \overline{x}_n \end{aligned}$$

Frage 1.13: Inwiefern gilt die Regel „Punkt- vor Strichrechnung“?

Frage 1.14: Was ist eine Normalform? Welche Normalformen kennst du?



(a) DNF (b) KNF (c) RSE

Abbildung 2: Skizzen der DNF, KNF und RSE einer Funktion

Eine Normalform ist die eindeutige Darstellung einer Funktion. Es werden drei Normalformen vorgestellt: die disjunktive Normalform oder DNF, die konjunktive Normalform oder KNF sowie die Ring Sum Expansion oder RSE. Wie die verschiedenen Normalformen benutzt werden, um eine Funktion f zu beschreiben, wird in Abbildung 2 schonmal skizziert und im Laufe dieses Kapitels noch im Detail besprochen.

Frage 1.15: Was ist ein Minterm? Was ist ein Maxterm? Gebe ein Beispiel an!

Unser Ziel ist es, Boolesche Funktionen eindeutig darzustellen. Dazu werden wir uns nun zuerst überlegen, wie wir Boolesche Funktionen, die nur für eine Eingabe den Wert 0 oder 1 annehmen, am besten darstellen können. Dazu betrachten wir im folgenden die Eingabe $a = (a_1, \ldots, a_n) \in \{0,1\}^n$. Diese soll entweder als einzige Eingabe auf 0 oder auf 1 abbilden. Weiter benutzen wir die Notation $x^0 = \overline{x}$ und $x^1 = x$. Es gilt dann offensichtlich:

$$x^a = 1 \;\Leftrightarrow\; x = a \text{ (wobei } a \in \{0,1\}!)$$

Dies kann durch Einsetzen verifiziert werden. Wir definieren dann den Minterm bzw. Maxterm zur Eingabe a wie folgt:

Minterm: Der zu $a = (a_1, \ldots, a_n)$ gehörige Minterm ist definiert als:

$$m_a(x) = x_1^{a_1} \wedge \cdots \wedge x_n^{a_n}$$

Maxterm: Der zu $a = (a_1, \ldots, a_n)$ gehörige Maxterm ist definiert als:

$$s_a(x) = x_1^{\neg a_1} \vee \cdots \vee x_n^{\neg a_n}$$

Minterme von $m_a(x)$ werden also genau dann 1, wenn x der Eingabe a entspricht. Maxterme $s_a(x)$ hingegen werden genau in diesem Fall 0. Oder etwas formaler ausgedrückt:

$$\begin{aligned} m_a(x) = 1 &\;\Leftrightarrow\; \forall 1 \le i \le n : \; x_i = a_i \\ s_a(x) = 0 &\;\Leftrightarrow\; \forall 1 \le i \le n : \; x_i = a_i \end{aligned}$$

Der Beweis ist relativ simpel. Der Minterm $m_a(x)$ wird genau dann 1, wenn alle $x_i^{a_i} = 1$ sind, also wenn $x_i = a_i$. Entsprechend wird der Maxterm genau dann 0, wenn alle $x_i^{\neg a_i} = 0$ sind. Dann muss jedoch gelten $x_i \neq \neg a_i$ oder anders gesagt: $x_i = \neg(\neg a_i) = a_i$.

Zum Abschluss soll noch ein kleines Beispiel behandelt werden. Gegeben sei die Eingabe $a = (1,0,0,1,1)$. Dann ergibt sich der Minterm zu a als:

$$m_a(x) = x_1^1 \wedge x_2^0 \wedge x_3^0 \wedge x_4^1 \wedge x_5^1 = x_1\overline{x}_2\overline{x}_3x_4x_5$$

Der entsprechende Maxterm ist:

$$x_1^0 \vee x_2^1 \vee x_3^1 \vee x_4^0 \vee x_5^0 = \overline{x}_1 \vee x_2 \vee x_3 \vee \overline{x}_4 \vee \overline{x}_5$$

Anhand dieser Beispiel kann man sich nun auch schnell klarmachen, dass der Minterm nur genau für $a = (1, 0, 0, 1, 1)$ den Wert 1 ergibt und der Maxterm für die Eingabe a den Wert 0.

Frage 1.16: Wie können Minterme bzw. Maxterme benutzt werden, um Funktionen darzustellen?

Wir haben schon gesehen, dass ein Minterm $m_a(x)$ nur genau dann 1 wird, wenn für die Eingabe $x = a$ gilt. Ansonsten wird der Minterm in jedem Fall 0. Wie können wir nun eine Funktion $f \in B_n$ beschreiben? Jede Eingabe für die Funktion f bildet entweder auf 0 oder auf 1 ab. Es reicht zur Beschreibung einer Funktion also aus, alle Eingaben aufzuzählen, die auf 0 abbilden. Alle anderen Eingaben müssen dann auf die 1 abbilden. Entsprechend könnten auch alle Eingaben aufgezählt werden, die auf 1 abbilden. Um weiter zu erläutern, wie eine Funktion dargestelt werden kann, nehmen wir nun an, dass die Eingaben aufgezählt werden, die auf 1 abbilden. Für jede dieser Eingaben a bilden wir nun jeweils zuerst den entsprechenden Minterm $m_a(x)$. Jeder dieser Minterme $m_a(x)$ wird nur für die Eingabe a zu 1 ausgewertet. Wir schreiben alle diese Minterme nun nebeneinander. Was könnten wir nun machen, um festzustellen, dass für eine beliebige Eingabe einer dieser Minterme zu 1 ausgewertet wird? Die Lösung ist, die Minterme durch $\vee$-Bausteine zu verknüpfen. Wenn dann mindestens einer der Minterme zu 1 ausgewertet wird, wird die ganze Funktion zu 1 ausgewertet. Analog kann eine Funktion mit Hilfe der Maxterme beschrieben werden, nur dass die Maxterme hier aus allen Eingaben gebildet werden, die auf 0 abbilden und dass die Maxterme durch $\wedge$-Verknüpfungen miteinander verbunden werden. Für eine Funktion f gilt also:

$$f(x) = \bigvee_{a \in f^{-1}(1)} m_a(x) = \bigwedge_{a \in f^{-1}(0)} s_a(x)$$

Die Darstellung der Funktion f als Disjunktion von Mintermen wird disjunktive Normalform (DNF) von f genannt. Die Darstellung als Konjunktion von Maxtermen wird konjunktive Normalform (KNF) von f genannt.

Der Beweis, dass eine Funktion f tatsächlich so dargestellt werden kann, wird nur für die DNF durchgeführt. Es $c \in \{0, 1\}^n$. Dann können zwei Fälle unterschieden werden:

Fall 1: $c \in f^{-1}(1)$, d.h. $f(c) = 1$. Dann enthält die DNF den Minterm $m_c(x)$ für den gilt: $m_c(c) = 1$. Dann gilt jedoch aufgrund der Rechenregeln mit Konstanten:

$$\bigvee_{a \in f^{-1}(1)} m_a(c) = 1$$

Fall 2: $c \in f^{-1}(0)$, d.h. $f(c) = 0$. Da die DNF nicht den Minterm $m_c(x)$ enthält, der als einziger aller Minterme für die Eingabe c zu 1 ausgewertet wird, werden also alle Minterme zu 0 ausgewertet und damit die ganze DNF.

Frage 1.17: Gebe ein Beispiel für eine Funktion als KNF bzw. DNF!

Gegeben sei die Funktion $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$. Es gilt dann:

$$\begin{aligned} f^{-1}(1) &= \{110, 001, 011, 101, 111\} \\ f^{-1}(0) &= \{000, 010, 100\} \end{aligned}$$

Die DNF, welche sich aus $f^{-1}(1)$ ergibt, sieht dann wie folgt aus:

$$f(x) = x_1x_2\overline{x}_3 \vee \overline{x}_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3 \vee x_1\overline{x}_2x_3 \vee x_1x_2x_3$$

Die KNF ergibt sich aus $f^{-1}(0)$:

$$f(x) = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x}_2 \vee x_3)(\overline{x}_1 \vee x_2 \vee x_3)$$

Frage 1.18: Was ist die duale Funktion zu einer Funktion? Gebe ein Beispiel!

Es sei eine Funktion $f(x_1, \ldots, x_n)$ gegeben. Die zu f duale Funktion f_d ist dann definiert als:

$$f_d = \neg f(\overline{x}_1, \ldots, \overline{x}_n)$$

Wie man sich recht schnell klarmachen kann, ist die zu f_d duale Funktion wieder die Funktion f. Es kann zum Beispiel gezeigt werden, dass die zu $\vee(x_1, \ldots, x_n)$ duale Funktion die Funktion $\wedge(x_1, \ldots, x_n)$ ist. Die UND-Funktion ist also dual zur ODER-Funktion. Dies ergibt sich mit Hilfe der Regel von deMorgan. Entsprechend der Definition ist die duale Funktion zu $\vee(x_1, \ldots, x_n)$ die Funktion $\neg \vee (\overline{x}_1, \ldots, \overline{x}_n)$. Wir formen dies um zu:

$$\neg \vee (\overline{x}_1, \ldots, \overline{x}_n) = \neg(\overline{x}_1 \vee \cdots \vee \overline{x}_n) = \overline{\overline{x}}_1 \wedge \cdots \wedge \overline{\overline{x}}_n = x_1 \wedge \cdots \wedge x_n = \wedge(x_1, \ldots, x_n)$$

Frage 1.19: Welche Beziehung gibt es zwischen der DNF einer Funktion und der KNF ihrer dualen Funktion?

Gegeben sei eine Funktion f in Form ihrer DNF:

$$f(x) = \bigvee_{a \in f^{-1}(1)} m_a(x) = \bigvee_{a \in f^{-1}(1)} x_1^{a_1} \ldots x_n^{a_n}$$

Dann ergibt sich die duale Funktion per Definition wie folgt:

$$f_d = \neg f(\overline{x}_1, \ldots, \overline{x}_n) = \neg \bigvee_{a \in f^{-1}(1)} \overline{x}_1^{a_1} \ldots \overline{x}_n^{a_n} = \bigwedge_{a \in f^{-1}(1)} \neg(\overline{x}_1^{a_1} \ldots \overline{x}_n^{a_n}) = \bigwedge_{a \in f^{-1}(1)} x_1^{a_1} \vee \cdots \vee x_n^{a_n}$$

Wir erhalten also die KNF der dualen Funktion f_d, indem wir in der DNF von f jeweils $\wedge$ gegen $\vee$ und ungekehrt tauschen.

Noch zeigen, dass dies die KNF von f_d ist!

Frage 1.20: Wie sieht die Darstellung einer Booleschen Funktion als Ring Sum Expansion (RSE) aus? Wie ist sie motiviert?

Unser Ziel ist nun im Folgenden eine Funktion f so wie in Abbildung 2(c) skizziert zu beschreiben, d.h. also nur mit Konjunktionen und $\oplus$-Addition. Desweiteren sollen alle Eingänge an den $\wedge$-Bausteinen nicht negiert vorliegen.

Warum kann eine solche Darstellungsweise erstrebenswert sein? Die Darstellung einer Funktion in Form ihrer KNF bzw. DNF hat den Nachteil, dass eine algebraische Struktur fehlt. Dies bedeutet, dass z.B. Gleichungen nicht eindeutig aufgelöst werden können. So lässt sich z.B. weder die Gleichung $f \wedge g = h$ noch die Gleichung $f \vee g = h$ eindeutig nach g auflösen. Ein Beispiel: Wenn $0 \vee g = 1$ gilt, folgt daraus, dass $g = 1$ sein muss. Gilt jedoch $1 \vee g = h$, dann kann über g nichts ausgesagt werden, da entweder $g = 0$ aber auch $g = 1$ gelten kann. Dagegen lässt sich $f \oplus g = h$ eindeutig zu $g = f \oplus h$ umformen, indem auf beiden Seiten eine $\oplus$-Addition von f vorgenommen wird:

$$\begin{aligned} f \oplus g = h \Leftrightarrow\ f \oplus f \oplus g &= f \oplus h \\ \Leftrightarrow\ 0 \oplus g &= f \oplus h \\ \Leftrightarrow\ g &= f \oplus h \end{aligned}$$

Oder anders gesagt: Wenn die Belegung von f und h bekannt ist, kann daraus auf die Belegung von g geschlossen werden. Dies kann man sich anhand der vier möglichen Fälle schnell klarmachen.

Frage 1.21: Beweise, dass die RSE-Darstellung für jede Boolesche Funktion existiert und dass sie eindeutig ist!

Bei der RSE-Darstellung möchten wir eine Boolesche Funktion f, so wie in Abbildung 2(c) skizziert, darstellen. Wir benutzen also Variablen, die für Elemente der Menge $\{0, 1\}$ stehen, und verknüpfen diese mit $\wedge$ und anschließend mit $\oplus$. Es kann nun leicht gezeigt werden, dass $(\{0, 1\}, \oplus, \wedge)$ isomporph zum Körper $(\mathbb{Z}_2, +, \cdot)$ ist, dass es sich bei beiden also um die gleiche Struktur handelt. Dabei übernimmt $\oplus$ die Rolle eines additiven Elements und $\wedge$ die Rolle eines multiplikativen Elements. Wenn wir die Variablen also auf die in Abbildung 2(c) beschriebene Weise verknüpfen, entstehen dabei Polynome. Denn ein Polynom ist im Prinzip eine Summe von multiplikativ verknüpften Variablen. Die RSE-Darstellung ist also die Darstellung einer Funktion als $\mathbb{Z}_2$-Polynom.

Um zu zeigen, dass für jede Boolesche Funktion f eines RSE-Darstellung existiert, können wir einen Algorithmus angeben, der aus der DNF der Funktion f die RSE-Darstellung berechnet. Dabei müssen im Prinzip zwei Probleme gelöst werden:

1. Die Ergebnisse der $\wedge$-Bausteine auf der ersten Ebene sollen nicht wie bei der DNF mit einem $\vee$-Baustein verknüpft werden, sondern mit einem $\oplus$-Baustein.

2. Die RSE-Darstellung soll keine negierten Variablen beinhalten.

Beide Probleme können im Prinzip relativ leicht gelöst werden. Die Funktion f ist als DNF gegeben, hat also die Form:

$$f(x) = \bigvee_{a \in f^{-1}(1)} m_a(x) = \bigvee_{a \in f^{-1}(1)} x_1^{a_1} \wedge \cdots \wedge x_n^{a_n}$$

Für jede Eingabe a, für die $f(a) = 1$ gilt, gibt es einen Minterm $m_a(x)$, der für die Eingabe a den Wert 1 liefert. Wir haben jedoch schon festgestellt, dass alle anderen Minterme in der DNF für die Eingabe a den Wert 0 ergeben. Für eine Eingabe ergibt also entweder ein Minterm den Wert 1 oder keiner. Dann machen wir jedoch nichts falsch, wenn wir in der DNF einfach alle $\vee$ durch $\oplus$ ersetzen, da sich die Wahrheitstabellen von $x \vee y$ und $x \oplus y$ nur für

$x = y = 1$ unterscheiden und ein solcher Fall, wie wir gerade festgestellt haben, bei der DNF nicht vorkommen kann. Es gilt also:

$$f(x) = \bigoplus_{a \in f^{-1}(1)} x_1^{a_1} \wedge \cdots \wedge x_n^{a_n} \tag{1}$$

Damit wäre das erste Problem gelöst und wir müssen und nur noch um das zweite Problem kümmern.

Es sollen im $\mathbb{Z}_2$-Polynom keine negierten Variablen vorkommen. In Formel 1 können diese jedoch noch vorkommen, nämlich wenn für $x_i^{a_i}$ gilt, dass $a_i = 0$ ist. Die Frage ist also, wie wir $\overline{x}_i$ durch x_i darstellen können, wobei nur $\wedge$, $\oplus$ oder Konstanten benutzt werden dürfen. Es zeigt sich schnell, dass folgende Ersetzung dies bewerkstelligt:

$$\overline{x}_i = x_i \oplus 1$$

Dies können wir uns schnell klarmachen, da $\overline{x}_i$ genau dann 0 wird, wenn $x_i = 1$ ist. Dann wird jedoch auch $x_1 \oplus 1$ zu 0. Ebenso werden beide für $x_i = 0$ zu 1 ausgewertet. Im zweiten Schritt werden also alle Variablen $\overline{x}_i$ durch $x_i \oplus 1$ ausgetauscht.

Nachdem die beiden ersten Schritte durchgeführt wurden, werden die Produktterme mit dem Distributivgesetz ausmultipliziert und eventuell vorhandene $\oplus$-Summanden entsprechend der Regel $t \oplus t = 0$ weggekürzt, bis keine doppelten Summanden mehr vorhanden sind.

Wie kann man nun zeigen, dass die RSE-Darstellung einer Funktion eindeutig ist? Wir wollen zeigen, dass jede Funktion eine RSE-Darstellung hat und dass jede RSE-Darstellung eindeutig eine Funktion repräsentiert. Dass jede Funktion aus B_n eine RSE-Darstellung hat, haben wir schon gesehen. Es gibt 2^{2^n} Funktionen in B_n und jede davon hat eine Darstellung als RSE. Als nächstes überlegen wir uns nun, wie viele verschiedene $\mathbb{Z}_2$-Polynome über n Variablen es gibt. Als erstes betrachten wir nur einen Summanden. Dieser kann höchstens n verschiedene Variablen enthalten, da nach der Vereinfachungs $x_i^2 = x_i \wedge x_i = x_i$ gilt. Wieviele verschiedene Summanden gibt es also. Wir können uns für jeden Summanden entscheiden, ob eine Variable in ihm vorkommen soll oder nicht. Also gibt es 2^n verschiedene Summanden. Für jedes $\mathbb{Z}_2$-Polynom können wir uns wiederum entscheiden, ob einer dieser 2^n Summanden in ihm vorkommen soll. Damit gibt es also 2^{2^n} verschiedene $\mathbb{Z}_2$-Polynome. Da es für jede der 2^{2^n} Funktionen aus B_n eine Darstellung als $\mathbb{Z}_2$-Polynom gibt und es auch genau 2^{2^n} $\mathbb{Z}_2$-Polynome gibt, muss die Abbildung also bijektiv sein und die Darstellung damit eindeutig.

2 Klassische zweistufige Logikminimierung

Frage 2.1: Wie ist die Logikminimierung motiviert?

Wir haben in Kapitel 1.2 drei Normalformen kennengelernt, mit deren Hilfe es möglich ist, Boolesche Funktionen zu beschreiben. Diese Normalformen bestehen aus zwei logischen Stufen, die auch mehr oder weniger direkt in Hardware umgesetzt werden können, sofern die Bausteine einen großen Fan-in erlauben. Jedoch kann eine solche direkte Umsetzung der Normalform sehr „teuer“ werden. So gibt es einige teilweise sehr einfach Funktionen, von denen die direkten Umsetzung exponentiell groß werden können, obwohl es sehr viel einfachere und günstigere Umsetzungen gibt. Dies liegt daran, dass die Normalformen sehr oft noch minimiert werden können.

Um uns dies klar zu machen, betrachten wir ein Beispiel. Es sei die Funktion $f \in B_n$ gegeben, für die $f(x_1, \ldots, x_n) = x_1$ gilt. Bei einer intuitiven Umsetzung in Hardware würde es offensichtlich ausreichen, den Eingang x_1 gleich als Ergebnis zu benutzen. Wird jedoch die DNF gebildet, so gibt es auf der ersten Stufe schon genau 2^{n-1} Minterme. Die Minterme werden nämlich aus allen Eingaben $(1, x_2, \ldots, x_n)$ gebildet, wovon es 2^{n-1} verschiedene gibt. Eine Implementierung entsprechend der DNF hätte für diese Funktion also exponentielle Größe, während unsere intuitive Implementierung konstante Größe hat. Es könnte nun angeführt werden, dass unsere Funktion f ja nicht essentiell von allen Eingabebits abhängt. Es gibt jedoch auch andere Funktionen, die von allen Eingabebits abhängen und eine exponentielle Größe aufweisen. Dazu gehört z.B. die Majoritätsfunktion MAJ_n, für die gilt:

$$MAJ_n = 1 \;\Leftrightarrow\; ||x|| \geq \left\lceil \frac{n}{2} \right\rceil$$

Die Majoritätsfunktion wird also genau dann 1, wenn mindestens die Hälfte aller Eingabebits 1 sind.

Frage 2.2: Was sind $\sum_2$-Schaltkreise?

$\sum_2$-Schaltkreise sind Schaltkreise, bei denen eine Disjunktion von Konjunktionen vorliegt. Der Name ergibt sich, da der Summentyp $\vee$ auf der zweiten Ebene realisiert wird. Wir werden uns in diesem Kapitel auf diese Art von Schaltkreisen konzentrieren. Es sei explizit darauf hingewiesen, dass eine DNF zwar einen $\sum_2$-Schaltkreis darstellt, jedoch nicht jeder $\sum_2$-Schaltkreis eine DNF ist!

2.1 Primimplikanten, Minimalpolynome, PLAs

Frage 2.3: Was ist ein Monom? Was ein Polynom?

Monom: Ein Monom m ist eine Konjunktion (also eine Produkt) von Literalen, wobei ein Literal eine Variable oder eine negierte Variable ist. So ist z.B. $x_1\overline{x}_3x_4$ ein Monom.

Polynom: Ein Polynom p ist eine Disjuktion (also eine Summe) von Monomen. Ein Beispiel für ein Polynom ist $x_1 \vee \overline{x}_3x_4$.

Frage 2.4: Was sind die Kosten eines Monoms bzw. eines Polynoms?

Die Kosten einem Monoms m entsprechend der Anzahl der Literale in m. Für die Kosten eines Polynoms p können zwei Maße verwendet werden:

- Die Summe der Kosten der Monome m_i in p. Die entspricht der Anzahl der Leitungen zu den $\wedge$-Bausteinen.
- Die Anzahl der Monome. Hier wird also die Anzahl der Bausteine benutzt, wobei der $\vee$-Baustein nicht mitgezählt wird.

Frage 2.5: Was ist ein Minimalpolynom?
Ein Polynom p ist ein Minimalpolynom bezüglich einer Funktion f, wenn es unter allen Polynomen, die f berechnen, minimale Kosten besitzt.

Frage 2.6: Was sind Implikanten? Was sind Primimplikanten?
Ein Polynom p, welches eine Funktion f darstellt, ist eine Summe von Monomen $m_1, \ldots, m_k$, für die gilt $f = m_1 \vee \cdots \vee m_k$. Die Funktion wird also 1 wenn mindestens eines der Monome 1 wird. Wir definieren dementsprechend:

Implikant: Ein Monom m heißt Implikant von f, wenn aus $m(a) = 1$ folgt, dass auch $f(a) = 1$ ist. Die Menge aller Implikanten einer Funktion f wird mit $I(f)$ bezeichnet.

Primimplikant: Ein Implikant m von f heißt Primimplikant, wenn keine echte Verkürzung von m ein Implikant von f ist. Die Menge aller Primimplikanten einer Funktion f wird mit $PI(f)$ bezeichnet.

Es bleibt noch die Frage, wann ein Monom m_1 nun eine Verkürzung von einem Monom m_2 ist? Ein Monom m_1 heißt Verkürzung von m_2 wenn alle Literale, die in m_1 vorkommen auch in m_2 vorkommen. Bei dieser Definition können die Monome auch genau die gleichen Literale enthalten, d.h. $m_1 = m_2$ gelten. Ist dies nicht der Fall, so ist m_1 eine echte Verkürzung von m_2.

Frage 2.7: Zeige, dass Minimalpolynome für eine Funktion nur Primimplikanten enthalten!
Wie wir schon gesehen haben, enthalten Polynome für eine Funktion f nur Implikanten von f. Wir gehen nun davon aus, dass ein Minimalpolynom $p = m_1 \vee \cdots \vee m_k$ für f gegeben ist, welches nicht nur Primimplikanten enthält. Es sei m_i mit $1 \leq i \leq k$ ein Implikant, der nicht Primimplikant ist. Es gibt dann also eine echte Verkürzung m_i' von m_i, die ebenfalls ein Implikant von f ist. Nach der Verkürzungsregel gilt $m_i' \vee m_i = m_i'$. Um dies zu zeigen, bezeichnen wir mit a alle Literale, die in m_i nicht jedoch in m_i' vorkommen. Dann gilt:

$$m_i' \vee m_i = m_i'(1 \vee a) = m_i' 1 = m_i'$$

Als Nächstes überlegen wir uns nun, dass wir nichts falsch machen, wenn wir zu einem Polynom p für eine Funktion f einen Implikanten $m \in I(f)$ hinzufügen. Denn wenn für eine Eingabe a $m(a) = 0$ gilt, so ändert dieser neue Implikant den Wert des Polynoms nicht. Gilt jedoch $m(a) = 1$, so folgt aus der Implikanteneigenschaft von m auch $f(a) = 1$ und es wird der richtige Wert für das Polynom berechnet. Unter Verwendung der eben gezeigten beiden Eigenschaften gilt dann jedoch:

$$f = m_1 \vee \cdots \vee m_i \vee \cdots \vee m_k = m_1 \vee \cdots \vee m_i \vee m_i' \vee \cdots \vee m_k = m_1 \vee \cdots \vee m_i' \vee \cdots \vee m_k$$

Ein Implikant, der nicht Primimplikant ist, kann also stets durch einen kürzeren Implikanten ausgetauscht werden. Auf diese Art und Weise könnte also zu einem Minimalpolynom, welches

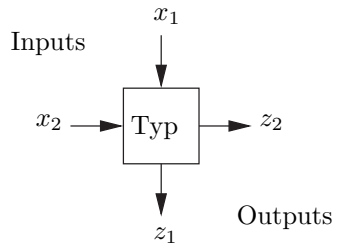


Abbildung 3: PLA-Zelle

mindestens einen Implikanten enthält, der nicht Primimplikant ist, ein günstigeres Polynom erzeugt werden, was ein Widerspruch zur Minimalitätseigenschaft des Polynoms ist. Demnach bestehen Minimalpolynome nur aus Primimplikanten.

Frage 2.8: Wie ist das Vorgehen beim Berechnen eines Minimalpolynoms?
Die Berechnung eines Minimalpolynoms geschieht in zwei Schritten:

1. Berechnung der Primimplikanten (aller Primimplikanten).
2. Auswahl der Primimplikanten treffen, die die Funktion überdeckt.

Der zweite Schritt wird durchgeführt, da ein Minimalpolynom zwar nur aus Primimplikanten besteht, jedoch nicht alle Primimplikanten enthalten muss.

Frage 2.9: Was ist ein PLA?
PLA steht für Programmable Logic Arrays. Ein PLA ist ein „programmierbarer“ Schaltkreis, d.h. es ist möglich das PLA so zu manipulieren, dass es spezielle Aufgaben erfüllt. Ein PLA besteht aus mehreren PLA-Zellen, die rechteckig angeordnet sind. Dabei ist eine PLA-Zelle ein Schaltkreis, der die Funktion $PLA \in B_{4,2}$ implementiert, welche wie folgt definiert ist:

$$PLA(x_1, x_2, y_1, y_2) = (z_1, z_2) = (x_1(\overline{y}_1 \vee y_1\overline{y}_2x_2 \vee y_1y_2\overline{x}_2),\ x_2 \vee x_1\overline{y}_1y_2)$$

Das Ergebnis, welches von einer PLA-Zelle berechnet wird, sind also die Bits z_1 und z_2. Wie schon erwähnt wurde, ist ein PLA programmierbar. Dabei wird eine einzelne PLA-Zelle über die Variablen y_1 und y_2 programmiert. Für diese zwei Variablen gibt es vier unterschiedliche Belegungen. Diese sind:

Identer: Diese Funktion wird für $(y_1, y_2) = (0, 0)$ berechnet. Es gilt $(z_1, z_2) = (x_1,\ x_2)$.

Addierer: Wird für $(y_1, y_2) = (0, 1)$ berechnet. Es gilt $(z_1, z_2) = (x_1,\ x_1 \vee x_2)$.

Multiplizierer: $(y_1, y_2) = (1, 0)$. Das Ergebnis ist $(z_1, z_2) = (x_1x_2,\ x_2)$.

Negatmultiplizierer: $(y_1, y_2) = (1, 1)$. Es wird $(z_1, z_2) = (x_1\overline{x}_2,\ x_2)$ berechnet.

Wir wollen uns nun überlegen, wie wir die PLA-Zellen programmieren können, um eine Funktion $B_{n.m}$ zu berechnen. Dabei werden wir uns eine PLA-Zelle wie in Abbildung 3 dargestellt vorstellen. Es gibt also einen „oberen“ Eingang x_1 und einen zugehörigen „unteren“ Ausgang z_1 sowie einen „linken“ Eingang x_2 und einen zugehörigen „rechten“ Ausgang z_2.

Betrachten wir die vier möglichen Typen nun genauer. Zuerst einmal gibt es einen Typ, der die Eingänge unverändert „nach unten“ und „nach rechts“ weitergibt. Dies ist der Identer.

Dann gibt es den Addierer, der ein berechnetes Ergebnis „nach rechts“ weitergibt, sowie zwei Typen, die ein berechnetes Ergebnis „nach unten“ weitergeben. Die Bedeutung der Richtungen wird klar, wenn wir nun betrachten, wie die PLA-Zellen zusammengeschaltet werden. Es soll eine Funktion $B_{n,m}$ realisiert werden. Diese Funktion besteht aus mehreren Monomen über höchstens n Variablen. Angenommen eines dieser Monome ist das Monom $m = x_2\overline{x}_3x_4$. Wie können wir den Wert dieses Monoms für eine Eingabe berechnen? Wir werden dazu n PLA-Zellen untereinander hängen, d.h. diese werden im späteren PLA, welches die komplette Funktion realisiert, eine Spalte bilden. Da wir diese Zellen untereinander hängen, sind in diesem Fall die Typen interessant, die die Ergebnisse „nach unten“ weitergeben, also der Identer, der Multiplizierer und der Negatmultiplizierer. Die Ergebnisse dieser drei Typen sind entweder nur von x_1 abhängig oder von x_1 und x_2. In letzerem Fall wird ein Produkt der beiden, d.h. entweder x_1x_2 oder $x_1\overline{x}_2$, weitergegeben. An den „linken“ Eingang x_2 werden wir nun die Eingabebits $x_1, \ldots, x_n$ anlegen (nicht mit den Eingängen x_1 und x_2 der PLA-Zellen verwechseln!). Wie müssen wir die als Spalte angeordneten PLA-Zellen nun programmieren, damit $m = x_2\overline{x}_3x_4$ realisiert wird? An den Eingang der obersten Zelle legen wir eine 1 an. Die Variable x_1 kommt in m nicht vor. Also wählen wir als erste Zelle den Identer. Bisher hierhin haben wir also nur das neutrale Element des Produkts berechnet. Der Ausgang der obersten PLA-Zelle ist mit dem Eingang der nächsten Zelle, welche für die Eingabe x_2 zuständig ist, verbunden. Wir wählen den Multiplizierer als Typ für diese Zelle. Dann haben wir also $x_1x_2 = 1x_2$ berechnet. Die nächste Zelle wird als Negatmultiplizierer gewählt. Da wir bisher $1x_2$ berechnet haben, wird dann also $1x_2\overline{x}_3 = x_2\overline{x}_3$ berechnet. Jetzt dürfte klar sein, wie die vierte Zelle für den Eingang x_4 gewählt wird, nämlich als Multiplizierer, da x_4 unnegiert in das Monom m eingehen soll. Wir haben also gezeigt, wie ein Monom mit Hilfe von PLA-Zellen realisiert werden kann. Doch wie berechnen wir eventuelle weitere Monome? Dies ist kein Problem. Denn wenn wir den Identer, Multiplizierer und Negatmultiplizierer näher betrachten, bemerken wir, dass diese alle den Eingang x_2 unverändert weitergeben. Wir können also mehrer Spalten von PLA-Zellen für die verschiedenen Monome nebeneinander anordnen, wobei sich jeweils in der i-ten Zeile um die Variable x_i gekümmert wird.

Bis hierhin haben wir nur den Identer, den Multiplizierer und den Negatmultiplizierer benutzt. Wofür ist also der Addierer gut? Wenn wir bis hierhin Monome realisieren können, liegt es nahe, dass der Addierer benutzt wird, um aus den Monomen einzelne Monome auszuwählen und zu einem Polynom zusammenzufügen. Für jede Funktion f, die von den Monomen abhängt, bilden wir dazu nun eine Zeile. Diese Zeilen liegen „unter“ den Spalten der Monome. Wenn ein Monom in Polynom zu f vorkommt, wählen wir einen Addierer, ansonsten einen Identer. Links an die Funktionen wird das neutrale Element der Addition, also eine 0, eingespeist. So lange also keines der im Polynom vorkommenden Monome eine 1 liefert, wird also eine 0 nach rechts weitergegeben.

Kommen wir nun zu einem Beispiel, an dem das oben Erläuterte wahrscheinlich um einiges klarer wird. Es sollen die Funktionen $f_1(x_1, x_2, x_3) = \overline{x}_2x_3 \vee x_1x_2x_3 \vee \overline{x}_1x_2\overline{x}_3$ und $f_2(x_1, x_2, x_3) = x_1x_3 \vee \overline{x}_2x_3 \vee \overline{x}_1x_2\overline{x}_3$ mit Hilfe eines PLAs realisiert werden. Dann werden diese wie in Abbildung 4 durch ein PLA realisiert. Es handelt sich um eine Funktion $B_{3,2}$, die durch $k = 4$ Monome realisiert wird. Also brauchen wir $(3 + 2) \cdot 4 = 20$ PLA-Zellen. Allgemein werden für eine Funktion $f \in B_{n,m}$, die durch k Monome realisiert wird, $(n+m) \cdot k$ PLA-Zellen benötigt. Wie man schnell sieht, liegen in den ersten drei Zeilen die Werte x_1, x_2 und x_3 an und in den nächsten beiden Zeilen für die Funktionen f_1 und f_2 jeweils der konstante Wert 0. Oben an jeder Spalte für die Monome liegt der konstante Wert 1 an. Die Eingaben werden von links nach rechts durchgereicht, genau wie die Auswertung der Monome

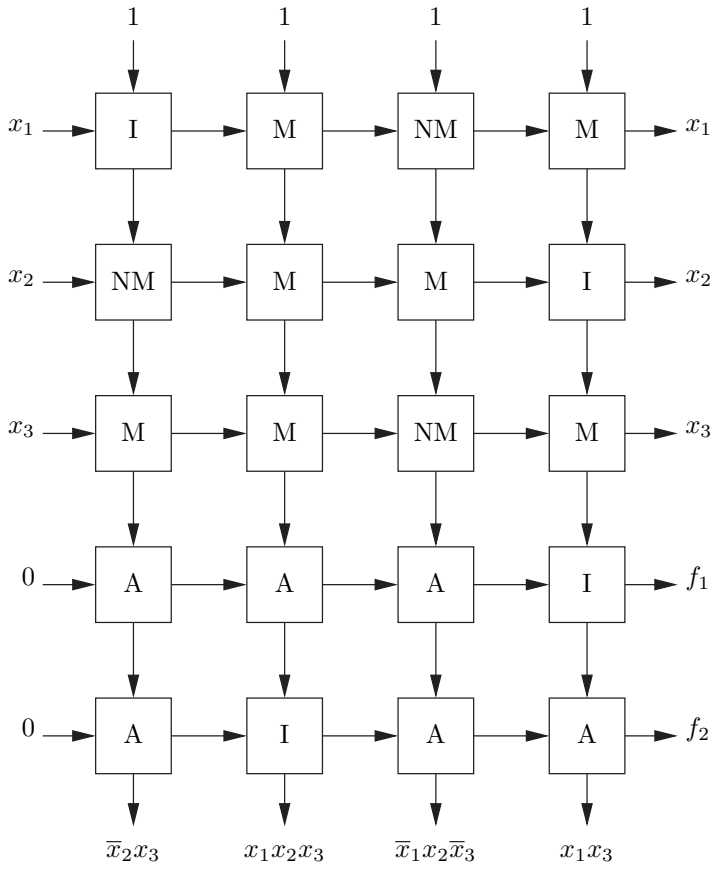


Abbildung 4: Beispiel für ein programmiertes PLA

ab der vierten Spalte stets nach unten weitergereicht werden.

2.2 Der Algorithmus von Quine/McCluskey

Frage 2.10: Was macht der Algorithmus von Quine/McCluskey?
Der Algorithmus von Quine/McCluskey berechnet zu einer Booleschen Funktion f die Menge der Primimplikanten $PI(f)$. Als Eingabe bekommt er dazu die Wertetabelle der Funktion also $(a, f(a))$.

Frage 2.11: Welche Möglichkeiten kennst du, um eine Funktion zu beschreiben?
Momentan kennen wir drei Möglichkeiten, um eine Funktion f zu beschreiben:

Wertetabelle: Die Funktion f ist durch ihre Wertetabelle $(a, f(a))$ gegeben.

Polynom: Beschreibung der Funktion f durch ein Polynom p.

Strukturelle Beschreibung: Die Funktion f ist strukturell beschrieben, also z.B. in der Form: „f realisiert die Addition zweier Binärzahlen mit Übertrag“.

Wir werden in diesem Kapitel mehrere Verfahren kennenlernen, um die Primimplikanten einer Funktion f zu berechnen. Der Algorithmus von Quine/McCluskey benutzt als Eingabe

die Wertetabelle der Funktion. Andere Verfahren bekommen ein Polynom als Eingabe und berechnen aus diesem die Primimplikanten.

Frage 2.12: Welches Lemma wird im Algorithmus von Quine/McCluskey benutzt? Wie wird es benutzt?

Der Algorithmus von Quine/McCluskey benutzt das folgende Lemma. Es sei m ein Monom, in dem weder die Variable x_j noch $\overline{x}_j$ vorkommt. Dann gilt:

$$m \in I(f) \;\Leftrightarrow\; mx_j, m\overline{x}_j \in I(f)$$

Ein Monom ist also nur dann in der Menge der Implikanten, wenn seine Verlängerungen mx_j bzw. $m\overline{x}_j$ ebenfalls *beide* in der Menge der Implikanten sind. Wir zeigen beide Richtungen:

„⇒“: Wir nehmen also an, dass m ein Implikant für die Funktion f ist, dass also gilt: $m(a) = 1 \Rightarrow f(a) = 1$. Wir wollen nun zeigen, dass daraus folgt, dass auch mx_j und $m\overline{x}_j$ Implikanten sind. Es muss also $mx_j(a) = 1 \;\Rightarrow\; f(a) = 1$ gezeigt werden. Wenn nun jedoch $mx_j(a) = 1$ gilt, dann wird auch die Verkürzung $m(a)$ zu 1 ausgewertet. Da m jedoch ein Implikant ist, folgt daraus $f(a) = 1$. Wir haben also gezeigt:

$$mx_j(a) = 1 \;\Rightarrow\; m(a) = 1 \;\Rightarrow\; f(a) = 1$$

Insgesamt also die Behauptung. Der Fall für $m\overline{x}_j$ wird ebenso gezeigt.

„⇐“: Wir nehmen also an, dass die Verlängerungen von m, die die Variable x_j im Gegensatz zu m enthalten, Implikanten von f sind. Zu zeigen ist, dass daraus folgt, dass $m(a) = 1 \;\Rightarrow\; f(a) = 1$ gilt. Es sei also $m(a) = 1$. Dann ist die Eingabe a an der j-ten Stelle also entweder 0 oder 1. Egal welcher Fall vorliegt, da jeweils mx_j und $m\overline{x}_j$ Implikanten sind, folgt in beiden Fällen, dass $f(a) = 1$ ist.

Wie wird dieses Lemma nun benutzt? Der Algorithmus von Quine/McCluskey erhält als Eingabe die Wertetabelle der Funktion. Aus dieser Wertetabelle kann die DNF der Funktion berechnet werden, indem für alle Eingaben $a \in f^{-1}(1)$, also alle Eingaben a, die 1 als Ergebnis haben, der Minterm m_a berechnet wird. Wir können uns schnell klarmachen, dass diese Menge der Minterme alle Implikanten der Länge n für die Funktion enthält. Längere Implikanten kann es nicht geben und wenn ein Implikant fehlen würde, hätten wir einen Minterm vergessen. Mit Q_k wird im Folgenden die Menge aller Implikanten der Länge k für eine Funktion f bezeichnet. Zu Beginn ist also die Menge Q_n bekannt. Aus dieser werden nun sukzessive die Mengen $Q_{n-1}, \ldots, Q_0$ berechnet, wobei einige dieser Mengen auch leer bleiben können. Dabei wird die Menge Q_{i-1} für $1 \leq i \leq n$ jeweils aus der Menge Q_i berechnet. Dies ist aufgrund des eben bewiesenen Lemmas möglich. Denn für jeden Implikanten m in der Menge Q_{i-1} muss es mindestens ein Paar von Implikanten mx_j und $m\overline{x}_j$ in der Menge Q_i geben. Wir durchsuchen also die Menge Q_i und schauen, ob es Paare von Implikanten gibt, die sich als mx_j und $m\overline{x}_j$ schreiben lassen, welche sich somit nur an einer Stelle unterscheiden. Wenn wir ein solches Paar finden, tragen wir m in Q_{i-1} ein und markieren mx_j und $m\overline{x}_j$. Es wird währenddessen passieren, dass in einer Menge Q_i einige Implikanten nicht markiert werden. Von diesen gibt es also keine Verkürzung, die ebenfalls Implikant ist. Demnach stellen diese Implikanten die Primimplikanten der Länge i für die Funktion dar. Wir merken uns die Primimplikanten in den Mengen $P_i,\ 1 \leq i \leq n$. Insgesamt ergibt sich also Algorithmus 1.

Frage 2.13: Wie kann der Algorithmus von Quine/McCluskey beschleunigt werden?

Algorithmus 1 Algorithmus von Quine/McCluskey

Eingabe: Funktionstabelle $(a, f(a)),\ a \in \{0,1\}^n$
Ausgabe: Menge der Primimplikanten $PI(f)$

1. Berechne aus der Funktionstabelle die Menge Q_n, die alle Minterme m_a für $a \in f^{-1}(1)$ enthält. Setze $i = n$.
2. Solange $Q_i \neq \emptyset$:
   a) $i = i - 1$.
   b) $Q_i = \{m \mid \exists j : x_j, \overline{x}_j \notin m,\ mx_j, m\overline{x}_j \in Q_{i+1}\}$
   c) $P_{i+1} = \{m \in Q_{i+1} \mid \text{Es gibt keine Verkürzung von } m \text{ in } Q_i\}$
3. $PI(f)$ ist die Vereinigung von $P_n, \ldots, P_{i+1}$.

Im bisher beschriebenen Algorithmus werden alle Paare von Implikanten in Q_{i+1} verglichen. Dies ist jedoch nicht sehr clever. Denn wenn es ein Paar gibt, dass sich als mx_j und $m\overline{x}_j$ schreiben lässt, dann stimmen diese in m komplett überein und unterscheiden sich nur darin, dass x_j einmal negiert ist und einmal nicht. Wir brauchen also nur die Implikanten vergleichen, bei denen sich die Anzahl der negierten Variablen um 1 unterscheidet. Wir unterteilen die Menge Q_{i+1} also in die Mengen $Q_{i+1,l}$ mit $0 \leq l \leq i+1$, wobei die Menge $Q_{i+1,l}$ die Menge der Implikanten der Länge $i+1$ mit l negierten Variablen ist. Es wird dann also für jedes Monom $m \in Q_{i+1,l}$ geschaut, ob es ein Monom $m_j \in Q_{i+1,l-1}$ gibt, welches entsteht, wenn in m die negierte Variable $\overline{x}_j$ durch x_j ausgetauscht wird. Für jedes Monom werden also höchstens n andere Monome gesucht.

Wir vergleichen nun die Anzahl der untersuchten Paare beim naiven und beim verbesserten Ansatz. In jedem Schritt des Algorithmus wird eine Menge Q_{i+1} untersucht. Überlegen wir uns nun, wieviele ungeordnete Paare in dieser Menge gebildet werden können. Es gibt $|Q_{i+1}|$ Elemente in der Menge Q_{i+1}. Das erste Element können wir frei wählen, also haben wir $|Q_{i+1}|$ Möglichkeiten. Danach bleiben, da wir ein Element nicht zweimal wählen, noch $|Q_{i+1}| - 1$ Möglichkeiten. Bisher ergibt dies also $|Q_{i+1}| \cdot (|Q_{i+1}| - 1)$ Möglichkeiten. Dies ist jedoch die Anzahl der geordneten Paare. Wenn wir ein Paar (x, y) gezählt haben, soll (y, x) jedoch nicht mehr gezählt werden, da wir die beiden Elemente ja schon miteinander verglichen haben. Also fällt jedes zweite Paar weg. Insgesamt ergeben sich also $|Q_{i+1}| \cdot (|Q_{i+1}| - 1)/2$ Möglichkeiten. Alternativ kann über den Binomialkoeffizient argumentiert werden. Wie viele Möglichkeiten gibt es 2 Elemente aus $|Q_{i+1}|$ Elementen auszuwählen? Dies sind:

$$\begin{aligned}\binom{|Q_{i+1}|}{2} &= \frac{|Q_{i+1}|!}{2 \cdot (|Q_{i-1}| - 2)!} = \frac{|Q_{i+1}| \cdot (|Q_{i+1}| - 1) \cdot (|Q_{i+1}| - 2) \cdot (|Q_{i+1}| - 3) \cdots 2 \cdot 1}{2 \cdot (|Q_{i+1}| - 2) \cdot (|Q_{i+1}| - 3) \cdots 2 \cdot 1} \\ &= \frac{|Q_{i+1}|(|Q_{i+1}| - 1)}{2}\end{aligned}$$

Wie etwas weiter oben schon erwähnt wurde, werden bei der verbesserten Version für ein Monom höchstens n andere Monome gesucht. Also werden höchstens $|Q_{i+1}| \cdot n$ Paare betrachtet.

Frage 2.14: Wie können die Monome beim Algorithmus von Quine/McCluskey abgespei-

chert werden?

Beim Algorithmus von Quine/McCluskey werden Mengen $Q_{i+1,l}$ verwaltet und diese müssen irgendwie abgespeichert werden. Bei der verbesserten Variante des Algorithmus haben wir schon gesehen, dass wir in der Lage sein möchten, nachzuschauen, ob sich ein Element in einer Menge befindet. Für diese Operationen bieten sich also Dictionaries wie z.B. 2-3-Bäume an. Wir werden gleich sehen, dass dabei das Monom selbst der Schlüssel sein wird, unter dem es abgespeichert wird. Die Schlüsselmenge für ein Dictionary muss geordnet sein, damit es möglich ist, systematisch nach ihnen zu suchen. Doch wie können wir ein Monom überhaupt eindeutig beschreiben? Wir haben im Abschnitt über PLAs gesehen, wie ein Monom in einem PLA realisiert werden kann. Ist eine Variable nicht vorhanden, wählen wir an ihrer Stelle den Identer, im Buch mit dem Wert 0 (die Interpretation von $(y_1, y_2) = (0, 0)$ als Dezimalzahl) bezeichnet. Der Multiplizierer (2) wird verwendet, wenn die Variable unnegiert vorkommt, und der Negatmultiplizierer (3), wenn die Variable negiert vorkommt. Damit lässt sich ein Monom also als Wort in $\{0, 2, 3\}^n$ beschreiben und diese lassen sich lexikographisch ordnen. Es ist somit möglich, ein Monom in einem Dictionary abzuspeichern. Die Zeit zum Einfügen und Suchen ist in einem 2-3-Baum für die Menge $Q_{i+1,l}$ durch $O(\log |Q_{i+1,l}|)$ beschränkt.

Frage 2.15: Welche Laufzeit hat der Algorithmus von Quine/McCluskey?

Zuerst wollen wir uns überlegen, wie hoch die Kosten für das Einfügen bzw. Suchen eines Elements höchstens werden können. Wir benutzen eine 2-3-Baum (oder AVL-Baum) und bei diesen kann das Einfügen und Suchen für eine Menge $Q_{i+1,l}$ in $O(\log |Q_{i+1,l}|)$ bewerkstelligt werden. Die Zeit ist also abhängig von der Größe der Menge. Wie groß können die beteiligten Mengen in unserem Fall höchstens werden? Das Schlimmste was passieren kann, ist, dass eine Menge alle Monome über n Variablen enthält. Da ein Monom als ein Wort aus $\{0, 2, 3\}^n$ interpretiert werden kann, gibt es 3^n verschiedene Monome. Eine Menge kann somit höchstens 3^n Monome enthalten. Es ergibt sich also eine Zeit von höchstens $O(\log 3^n) = O(n)$ für das Einfügen und Suchen.

Wir haben auch schon festgestellt, dass für jedes Monom höchstens n andere Monome gesucht werden. Damit verursacht ein Monom höchstens Kosten von $O(n^2)$. Die letzte Frage, die wir uns nun stellen, ist, wie oft ein Monom diese Kosten verursachen kann. Oder anders gesagt: Wenn wir ein Monom behandelt haben, kann es dann im Laufe des Algorithmus wieder auftauchen? Die Antwort auf diese Frage lautet „Nein“, was wir uns schnell klarmachen können. Ein Monom, welches Kosten verursacht, liegt in einer Menge $Q_{i+1,l}$, also in der Menge der Monome mit Länge $i + 1$ und l negierten Variablen. Die Anzahl l ist dabei im Verlauf der weiteren Argumentation unwichtig. Wichtig ist, dass aus der Menge Q_{i+1} eine Menge Q_i erzeugt wird, die nur noch Monome der Länge i enthält. In dieser Menge kann das Monom jedoch nicht ebenfalls liegen. Da während des Algorithmus stets Mengen von kürzeren Monomen erzeugt werden, kann jedes Monom also nur einmal Kosten verursachen. Es gibt, wie wir schon festgestellt haben, 3^n verschiedene Monome, wobei jedes höchstens Kosten von $O(n^2)$ verursachen kann. Insgesamt ergeben sich damit Kosten von höchstens $O(n^2 \cdot 3^n)$. Wir wollen die Laufzeit nun in Abhängigkeit von der Eingabelänge $N = 2^n$ ausdrücken. Dann gilt zuerst einmal: $N = 2^n \Leftrightarrow n = \log N$. Nun müssen wir noch den Ausdruck 3^n in Abhängigkeit von N beschreiben:

$$3^n = (2^{\log_2 3})^n = (2^n)^{\log_2 3} = N^{\log 3}$$

Wenn wir die Laufzeit in Abhängigkeit von der Eingabelänge N angeben, so ergibt sich also $O(3^n n^2) = O(N^{\log 3} \log^2 N)$ und somit eine in N polynomielle Laufzeit.

Frage 2.16: Warum kann die Ausgabe beim Algorithmus von Quine/McCluskey länger werden als die Eingabe?

Der Algorithmus von Quine/McCluskey erhält als Eingabe die Wertetabelle der Funktion und erstellt aus dieser ein Polynom, indem die Minterme gebildet werden. Minterme sind Monome, in denen alle n Variablen entweder negiert oder nicht negiert vorkommen. Es ist bei Mintermen, entsprechend ihrer Definition, nicht möglich, dass eine Variable gar nicht vorkommt. Wir überlegen uns nun, wieviele Monome es gibt, die auch Minterme sind. Dies sind 2^n, da man sich für jede Variable entscheiden kann, ob sie negiert oder nicht negiert vorkommt. Bei allgemeinen Monomen, was Primimplikanten sind, hat man jedoch die zusätzliche Option, eine Variable gar nicht zu wählen. Es gibt also 3^n verschiedene Monome. Man könnte nun auf die Idee kommen, dass es vielleicht gar keine Funktion gibt, deren Menge der Primimplikanten so viele Monome enthält. Es gibt jedoch tatsächlich Funktionen, bei denen die Menge der Primimplikanten annäherend so groß wird.

Frage 2.17: Gebe eine Funktion an, bei der die Ausgabe der Primimplikanten tatsächlich länger wird als die Eingabe!

Eine Funktion, bei der die Ausgabe der Primimplikanten länger wird als die Eingabe ist die Funktion „Mittleres Drittel“ oder $MD_n \in B_n$, welche für $n = 3k$ wie folgt definiert ist:

$$MD_n(x) = 1 \;\Leftrightarrow\; k \leq ||x|| \leq 2k \quad \left(\Leftrightarrow\; \frac{1}{3}n \leq ||x|| \leq \frac{2}{3}n\right)$$

Die Funktion wird also genau dann 1, wenn mindestens ein und höchstens zwei Drittel der Eingabe aus Einsen besteht.

Wir überlegen uns nun zuerst, wie Implikanten für diese Funktion aussehen und danach welches Aussehen die Primimplikanten aufweisen. Es ist schnell einzusehen, dass die Implikanten für MD_n mindestens k positive (also unnegierte) und k negative Literale enthalten müssen. Ansonsten wäre nicht sicher gestellt, dass MD_n auch tatsächlich eine 1 berechnet. So wäre z.B. das Monom x_1 kein Primimplikant von MD_6, da es für die Eingabe $(1, 0, 0, 0, 0, 0)$ eine 1 liefern würde, MD_6 jedoch eine 0. Wie sehen nun die Primimplikanten aus? Wenn ein Implikant mindestens $k + 1$ positive Literale enthält, so können wir ein positives Literal entfernen, ohne die Implikanteneigenschaft zu verletzen. Es existiert dann also eine Verkürzung, die ebenfalls Implikant ist. Daher können Implikanten, die mehr als k positive oder negative Literale enthalten, keine Primimplikanten sein. Monome mit weniger als k positiven oder negativen Literalen sind jedoch keine Implikanten mehr, wie wir gerade beobachtet haben. Also sind die Primimplikanten von MD_n die Implikanten mit *genau* k positiven und k negativen Literalen. Wieviele solcher Monome gibt es? Wir müssen die Anzahl der Möglichkeiten herausfinden, aus den n Variablen k Variablen auszuwählen, die positive Literale sein sollen und k Stück, die negativ sein sollen. Wieviele Möglichkeiten gibt es die k positiven Literale aus n Variablen zu wählen? Dies sind:

$$\binom{n}{k}$$

Wenn wir die k positiven Variablen gewählt haben, bleiben noch die $n - k$ ungewählten Variablen übrig, um aus diesen k negative Literale auszuwählen. Dies sind

$$\binom{n-k}{k}$$

Möglichkeiten. Es ergibt sich also insgesamt die folgende Anzahl an Möglichkeiten:

$$\binom{n}{k} \cdot \binom{n-k}{k} = \frac{n!}{k!(n-k)!} \cdot \frac{(n-k)!}{k!((n-k)-k)!} = \frac{n!}{(k!)^2 \cdot (n-2k)!}$$

Wir können $(n-2k)$ zu $(3k-2k) = k$ umformen, da $n = 3k$ gilt. Also ergibt sich:

$$\frac{(3k)!}{(k!)^3}$$

Wir benutzen die Formel von Stirling, um dies weiter aufzulösen. Diese besagt:

$$n! = \Theta\left(n^{n+\frac{1}{2}} e^{-n}\right)$$

Es ergibt sich also:

$$\frac{(3k)^{3k+\frac{1}{2}} \cdot e^{-3k}}{\left(k^{k+\frac{1}{2}} \cdot e^{-k}\right)^3} = \frac{3^{3k+\frac{1}{2}} \cdot k^{3k+\frac{1}{2}} \cdot e^{-3k}}{k^{3k+\frac{3}{2}} \cdot e^{-3k}} = \frac{3^{\frac{1}{2}} \cdot 3^{3k}}{k}$$

Wir ersetzen nun noch $3k$ durch n bzw. k durch $\frac{1}{3}n$:

$$\frac{3^{\frac{1}{2}} \cdot 3^n}{\frac{1}{3} \cdot n} = 3^{\frac{3}{2}} \cdot \frac{3^n}{n} = \Theta\left(\frac{3^n}{n}\right)$$

Jeder Algorithmus der die Menge $PI(MD_n)$ berechnet, muss also $\Theta(3^n/n)$ Primimplikanten ausgeben. Jeder Primimplikant hat k positive und k negative Literale, also die Länge $2k = \frac{2}{3}n$. Damit hat die Ausgabe der Menge aller Primimplikanten die Länge $\Theta(\frac{2}{3}n \cdot 3^n/n) = \Theta(3^n)$. Bezogen auf die Eingabelänge N ergibt dies $\Theta(N^{\log 3})$, was der Laufzeit des Quine/McCluskey recht ähnlich sieht. Daraus können wir folgern, dass der Algorithmus von Quine/McCluskey im Worst Case eine gute Laufzeit aufweist.

2.3 Baummethode und Methode des iterierten Konsensus

Frage 2.18: Was berechnen die Baummethode und die Methode des iterierten Konsensus? Was unterscheidet sie vom Algorithmus von Quine/McCluskey?

Die Baummethode und die Methode des iterierten Konsensus berechnen ebenfalls die Menge der Primimplikanten zu einer Funktion f. Im Gegensatz zum Algorithmus von Quine/McCluskey, welcher als Eingabe die Wertetabelle der Funktion bekommt, bekommen diese beiden Verfahren als Eingabe ein Polynom, welches die Funktion f beschreibt. Die maximale Eingabelänge des Algorithmus von Quine/McCluskey beträgt $\Theta(2^n)$ und die maximale Anzahl an Primimplikanten, die ausgegeben werden können, beträgt $\Theta(3^n)$. Der Blow-Up an Informationen beim Algorithmus von Quine/McCluskey ist also polynomiell in der Eingabelänge. Wir werden noch sehen, dass dieser Blow-Up bei Verfahren, die Polynome als Eingabe erhalten, viel größer sein kann.

2.3.1 Die Baummethode

Frage 2.19: Was sind Pinup-Polynome? Wie können sie benutzt werden, um die Primimplikanten einer Funktion zu berechnen?

„Pinup“ steht für „Primimplikanten und Implikanten“. Entsprechend der Bedeutung der Abkürzung ist ein Pinup-Polynom für eine Funktion f ein Polynom, welches *alle* Primimplikanten und ansonsten nur Implikanten von f enthält. Es sei z.B. die Funktion $f = x_1x_3 \vee x_1x_4$ gegeben. Wir sehen schnell ein, dass x_1x_3 und x_1x_4 auch die Primimplikanten dieser Funktion sind. Eine Verlängerung der Primimplikanten ist ein Implikant. Somit ist z.B. $p = x_1x_3 \vee x_1x_4 \vee x_1x_4\overline{x}_5$ ein Pinup-Polynom für f.

Wenn nun ein Pinup-Polynom p gegeben ist, enthält es alle Primimplikanten. Wenn also die restlichen Monome, die Verkürzungen in p enthalten, gestrichen werden, erhalten wir die Menge der Primimplikanten. Das Ziel ist es also, ein Pinup-Polynom zu berechnen und dieses zu vereinfachen, um so an die Menge der Primimplikanten zu gelangen.

Frage 2.20: Was besagt das „UND“-Lemma?

Um das „UND“-Lemma zu erläutern, müssen wir zuerst die Multiplikation zweier Polynome p und q definieren. Wir definieren das Produkt $p \star q$ als:

$$p \star q = \bigvee_{m \in p,\ m' \in q \text{ mit } mm' \neq 0} mm'$$

Wir bilden also das Produkt zweier Polynome, indem wir jeden Summanden von p mit jedem Summanden von q multiplizieren, wobei wir Summanden, die 0 ergeben, auslassen. Es seien z.B. $p = x_1x_3\overline{x}_4 \vee x_1x_5$ und $q = x_2\overline{x}_4 \vee x_3x_4$ gegeben. Dann ist $p \star q$:

$$p \star q = x_1x_2x_3\overline{x}_4 \vee \underbrace{x_1x_3x_4\overline{x}_4}_{=0} \vee x_1x_2\overline{x}_4x_5 \vee x_1x_3x_4x_5$$

Nachdem die Multiplikation eingeführt wurde, können wir nun das „UND“-Lemma erläutern. Es sei p_1 ein Pinup-Polynom für die Funktion f_1 und p_2 ein Pinup-Polynom für die Funktion f_2. Dann ist $p_1 \star p_2$ ein Pinup-Polynom für die Funktion $f_1 \wedge f_2$. Um dies zu beweisen, sind zwei Dinge zu zeigen:

1. $p_1 \star p_2$ stellt die Funktion $f_1 \wedge f_2$ dar bzw. $p_1 \star p_2$ enthält nur Implikanten von $f_1 \wedge f_2$.
2. Das Polynom $p_1 \star p_2$ enthält alle Primimplikanten von $f_1 \wedge f_2$.

Beweis 1.

Wir beweisen nun zuerst Punkt 1, dass also jedes Monom in $p_1 \star p_2$ nur dann 1 wird, wenn auch $f_1 \wedge f_2$ den Wert 1 ergibt. Die Monome in $p_1 \star p_2$ haben die Form m_1m_2 mit $m_1 \in p_1$ und $m_2 \in p_2$. Wir überlegen und nun, was gilt, wenn $m_1m_2(a) = 1$ ist. Dann gilt auch $m_1(a) = 1$ und $m_2(a) = 1$. Da m_1 und m_2 jedoch Monome in Pinup-Polynomen für f_1 bzw. f_2 sind, sind sie Implikanten für f_1 bzw. f_2. Also folgt $f_1(a) = 1$ und $f_2(a) = 1$. Daraus folgt jedoch wiederum $f_1(a) \wedge f_2(a) = 1$ und die Behauptung ist gezeigt.

Beweis 2.

Im Beweis zu Punkt 2 zeigen wir nun, dass $p_1 \star p_2$ alle Primimplikanten von $f_1 \wedge f_2$ enthält. Es sei $m \in PI(f_1 \wedge f_2)$ ein beliebiger Primimplikant von $f_1 \wedge f_2$. Es gilt dann also $m(a) = 1 \Rightarrow (f_1 \wedge f_2)(a) = 1$. Die Konjunktion in dieser Folgerung kann in die Einzelfolgerungen $m(a) = 1 \Rightarrow f_1(a) = 1$ und $m(a) = 1 \Rightarrow f_2(a) = 1$ aufgeteilt werden. Dann ist $m \in I(f_1)$ und $m \in I(f_2)$, d.h. m ist ein Implikant von f_1 und f_2. Dann gibt es jedoch auch Verkürzungen $m_1 \in PI(f_1)$ und $m_2 \in PI(f_2)$, die Primimplikanten sind. Da p_1 und p_2 Pinup-Polynome sind,

kommt m_1 in p_1 und m_2 in p_2 vor. Wenn m_1 und m_2 jeweils beide Verkürzungen von m sind, ist also auch $m_1 m_2$ eine Verkürzung von m. Nach Punkt 1 ist dann $m_1 m_2$ ein Implikant von $f_1 \wedge f_2$. Da m jedoch nach Voraussetzung ein Primimplikant ist, muss also $m_1 m_2 = m$ gelten. Da wir schon gesehen haben, dass m_1 in p_1 vorkommt und m_2 in p_2, kommt $m_1 m_2 = m$ also auch im Produkt $p_1 \star p_2$ vor.

Frage 2.21: Wie sind Subfunktionen definiert?

Es sei $f \in B_n$. Dann ist die Subfunktion $f_{x_i=c}$ ist für $1 \leq i \leq n$ und $c \in \{0,1\}$ definiert als:

$$f_{x_i=c}(a) = (a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots, a_n)$$

Die Subfunktion $f_{x_i=c}$ entsteht also quasi dadurch, dass in der Funktion f die i-te Stelle der Eingabe auf die Konstante c gesetzt wird. Damit ist $f_{x_i=c}$ also nicht mehr essentiell von x_i abhängig, weshalb $f_{x_i=c}$ oft als Funktion aus B_{n-1} aufgefasst wird. Es spielen also nur noch $n-1$ Variablen ein Rolle bei der Auswertung.

Frage 2.22: Was besagt die Shannon-Zerlegung?

Die Shannon-Zerlegung besagt:

1. $f = \overline{x}_i \cdot f_{x_i=0} \vee x_i \cdot f_{x_i=1}$
2. $f = (x_i \vee f_{x_i=0}) \wedge (\overline{x}_i \vee f_{x_i=1})$

Behauptung 1 ist schnell gezeigt. Es sei $a \in \{0,1\}^n$. Wenn $a_i = 0$ gilt, selektieren wir über $\overline{x}_i$ die Funktion $f_{x_i=0}$ und berechnen also das richtige Ergebnis. Für $a_i = 1$ wird per x_i die Funktion $f_{x_i=1}$ selektiert und damit ebenfalls das richtige Ergebnis berechnet. Behauptung 2 wird in der Baummethode benutzt und ist mit Hilfe der Regeln für das Rechnen mit Konstanten und den Vereinfachungsregeln (siehe Seite 8) schnell gezeigt. Wenn $a_i = 0$ ist, gilt:

$$f = (0 \vee f_{x_i=0}) \wedge (1 \vee f_{x_i=1}) = (0 \vee f_{x_i=0}) \wedge 1 = 0 \vee f_{x_i=0} = f_{x_i=0}$$

Damit wird für $a_i = 0$ das korrekte Ergebnis berechnet. Entsprechend gilt für $a_i = 1$:

$$f = (1 \vee f_{x_i=0}) \wedge (0 \vee f_{x_i=1}) = 1 \wedge (0 \vee f_{x_i=1}) = 0 \vee f_{x_i=1} = f_{x_i=1}$$

Frage 2.23: Was besagt das „ODER“-Lemma?

Nachdem wir die Subfunktionen eingeführt haben, können wir das „ODER“-Lemma erläutern. Wir haben schon gesehen, dass eine Subfunktion $f_{x_i=c}$ nicht mehr von der Variablen x_i abhängt. Das „ODER“-Lemma besagt: Es sei p ein Pinup-Polynom für die Funktion g, die nicht von der Variablen x_i abhängt. Dann gilt:

1. $x_i \vee p$ ist ein Pinup-Polynom für $x_i \vee g$.
2. $\overline{x}_i \vee p$ ist ein Pinup-Polynom für $\overline{x}_i \vee g$.

Wir werden hier nur Fall 1 beweisen, da Fall 2 entsprechend gezeigt wird. Wir definieren zuerst $h = x_i \vee g$. Es ist trivial, dass $x_i \vee p$ nur Implikanten von h enthält, womit nur noch zu zeigen bleibt, dass $x_i \vee p$ auch alle Primimplikanten von h enthält. Es sei also ein Primimplikant $m \in PI(h)$ gegeben. Wir unterscheiden für diesen zwei Fälle: Entweder enthält m die Variable x_i oder nicht.

Fall 1: m enthält x_i. Wenn wir uns $h = x_i \vee g$ genauer anschauen, fällt auf, dass x_i selbst ein Implikant von h ist. Somit muss $m = x_i$ gelten, denn ansonsten gäbe es eine Verkürzung von m, die ebenfalls Implikant von h ist, und m hätte von vornherein kein Primimplikant sein können. Es gilt also $m = x_i$ und x_i kommt auch in $x_i \vee p$ vor. Im Prinzip haben wir gezeigt, dass x_i der einzige Primimplikant von h sein kann, in dem x_i vorkommt. $x_i \in m$

Fall 2: m enthält x_i nicht. Wir werden nun erst zeigen, dass m ein Implikant von g ist und anschließend, dass m auch ein Primimplikant von g sein muss. Dass m ein Implikant von g ist, bedeutet ja nichts anderes als: $m(a) = 1 \Rightarrow g(a) = 1$. Da m nach Annahme ein Primimplikant von h ist, gilt zumindest schonmal $m(a) = 1 \Rightarrow h(a) = 1$. Wir unterscheiden für die Eingabe a zwei Fälle. $x_i \notin m$

Im ersten Fall nehmen wir an, dass $a_i = 0$ ist. Dann gilt $h(a) = 0 \vee g(a)$. Da jedoch aus der Primimplikanteneigenschaft von m folgt, dass $h(a) = 1$ ist, folgt daraus wiederum, dass $g(a) = 1$ gelten muss, da die $\vee$-Verknüpfung ansonsten keine 1 liefern würde.

Im zweiten Fall gelte $a_i = 1$. Wir werden hier nun ausnutzen, dass g nicht von x_i abhängt. Es sei $a^* = (a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_n)$ die Eingabe, die wir aus a erhalten, wenn wir die i-te Stelle auf 0 setzen. Als nächstes wollen wir zeigen, dass wenn $m(a) = 1$ gilt auch $m(a^*) = 1$ gilt. In m kommt die Variable x_i nicht vor. Allerdings kann auch die Variable $\overline{x}_i$ nicht in m vorkommen, da $m(a) = 1$ mit $a_i = 1$ gilt. Damit können wir a an der i-ten Stelle beliebig verändern und es gilt $m(a) = m(a^*) = 1$. Da m ein Primimplikant von h ist, folgt aus $m(a^*) = 1$, dass auch $h(a^*) = 1$ ist. Dann gilt jedoch $h(a^*) = 0 \vee g(a^*) = g(a^*) = 1$. Im letzten Schritt nutzen, wir nun aus, dass g nicht von x_i abhängt. Damit folgt aus $g(a^*) = 1$, dass auch $g(a) = 1$ sein muss.

Wir haben bis jetzt also gezeigt, dass jeder Primimplikant m von h, in dem x_i vorkommt, in $x_i \vee p$ vorkommt und dass jeder andere Primimplikant m ein Implikant von g ist. Es bleibt also nur noch zu zeigen, dass m auch Primimplikant von g ist. Denn in diesem Fall kommt m in p vor, da dies ja ein Pinup-Polynom für g ist, und somit auch in $x_i \vee p$. Jeder Primimplikant m für h ist also Implikant von g. Aufgrund der $\vee$-Verknüpfung ist m dann jedoch auch Implikant von h. Stellen wir uns nun vor, es gäbe eine echte Verkürzung von m, die ebenfalls Implikant von g ist. In diesem Fall wäre die Verkürzung von m auch ein Implikant für h. Dann könnte m jedoch kein Primimplikant für h sein, woraus folgt, dass m dann auch ein Primimplikant für g sein muss.

Frage 2.24: Gebe den Algorithmus der Baummethode an! Wie gehen das „UND“- und das „ODER“-Lemma dort ein?

Mit dem „UND“-Lemma, dem „ODER“-Lemma und der Definition der Subfunktionen haben wir schon alles zusammen, um die Baummethode anzugeben. Wenn wir die Pinup-Polynome zweier Funktionen gegeben haben, können wir mit Hilfe des „UND“-Lemmas das Pinup-Polynom der Konjunktion der beiden Funktionen berechnen. Die Konjunktion trat auch im der zweiten Variante der Shannon-Zerlegung auf $(f = (x_i \vee f_{x_i=0}) \wedge (\overline{x}_i \vee f_{x_i=1}))$. Wenn wir also die Pinup-Polynome für $(x_i \vee f_{x_i=0})$ und für $(\overline{x}_i \vee f_{x_i=1})$ berechnen können, können wir aufgrund des „UND“-Lemma mit Hilfe der Multiplikation das Pinup-Polynom von f berechnen. Dank des „ODER“-Lemmas können wir auch $(x_i \vee f_{x_i=0})$ und $(\overline{x}_i \vee f_{x_i=1})$ berechnen, da die Subfunktionen $f_{x_i=0}$ und $f_{x_i=1}$ nicht mehr von x_i abhängen. Die Subfunktionen werden für ein Polynom gebildet, indem für die entsprechende Variable 0 oder 1 eingesetzt wird und anschließend das resultierende Polynom vereinfacht wird.

Algorithmus 2 Algorithmus der Baummethode

Eingabe: Ein Polynom p.
Vereinbarung: Polynome werden stets gekürzt:

- Monome, für die es Verkürzungen gibt, werden gestrichen.
- Mehrfach auftretende Monome werden nur einmal aufgeführt.
- Leere Summanden (also Terme „$\cdots \vee 0 \vee \ldots$“) werden gestrichen.
- Terme „$\cdots \vee 1 \vee \ldots$“ werden zu 1 vereinfacht.

Falls p das leere Polynom ist (also $p = 0$), liefere p zurück.
Falls p nur das Monom 1 enthält, liefere p zurück.
Ansonsten: Wähle ein i, so dass x_i oder $\overline{x}_i$ in p vorkommt und bilde die Polynome p_0 und p_1. Diese entstehen, indem in p alle Variablen x_i auf 0 bzw. 1 gesetzt werden. Die Polynome p_0 und p_1 stellen also die Subfunktionen $f_{x_i=0}$ und $f_{x_i=1}$ dar. Berechne für die Polynome p_0 und p_1 rekursiv die Pinup-Polynome p_0^* und p_1^*. Bilde:

$$p' = x_i \star p_1^* \vee \overline{x}_i \star p_0^* \vee p_0^* \star p_1^*$$

Vereinfache p' und liefere p' zurück.

Die Baummethode, welche in Algorithmus 2 beschrieben wird, berechnet also rekursiv die Pinup-Polynome der Subfunktionen und berechnet aus diesen das Pinup-Polynom der „eigentlichen“ Funktion. Die Polynome der Subfunktionen werden dabei immer kürzer, da in jedem Schritt mindestens eine Variable x_i wegfällt. Damit terminiert der Algorithmus also in jedem Fall.

Es bleibt also noch der Beweis, dass dieses Vorgehen korrekt ist. Da wir schon beobachtet haben, dass der Algorithmus in jedem Fall terminiert, bietet sich ein Induktionsbeweis an. Der Induktionsanfang wird für Funktionen gezeigt, in denen keine Literale mehr vorkommen. Dies sind die beiden konstanten Funktionen, für die in diesem Fall auch ein Pinup-Polynom zurückgeliefert wird (nämlich 0 oder 1). Im Induktionsschritt nehmen wir nun an, dass p_0^* und p_1^* Pinup-Polynome für die entsprechenden Subfunktionen $f_{x_i=0}$ bzw. $f_{x_i=1}$ sind. Wir haben im „ODER“-Lemma schon gesehen, dass dann $x_i \vee p_0^*$ bzw. $\overline{x}_i \vee p_1^*$ die Pinup-Polynome für die Funktionen $x_i \vee f_{x_i=0}$ bzw. $\overline{x}_i \vee f_{x_i=1}$ sind. Entsprechend der Shannon-Zerlegung stellen diese beiden Funktionen die ursprüngliche Funktion dar, wenn sie mit einer Konjunktion verknüpft werden. Da wir die Pinup-Polynome für die beiden Funktionen kennen, ergeben diese multipliziert ($\star$) also das Pinup-Polynom für die ursprüngliche Funktion („UND“-Lemma). Es ergibt sich dabei:

$$(x_i \vee p_0^*) \star (\overline{x}_i \vee p_1^*) = \underbrace{x_i \star \overline{x}_i}_{=0} \vee x_i \star p_1^* \vee \overline{x}_i \star p_0^* \vee p_0^* \star p_1^* = x_i \star p_1^* \vee \overline{x}_i \star p_0^* \vee p_0^* \star p_1^*$$

Genau dies berechnet der Algorithmus jedoch.

Frage 2.25: Gebe ein einfaches Beispiel für die Baummethode an!
Wir zeigen einen Fall kurz vor der Abbruchbedingung des Algorithmus. Es sei $p = x_2$. Dann ist $p_0^* = 0$ und $p_1^* = 1$. Es ergibt sich $p' = x_2 \star 1 \vee \overline{x}_2 \star 0 \vee 0 \star 1 = x_2$.

2.3.2 Methode des iterierten Konsensus

Frage 2.26: Was ist ein Konsensus?
Für zwei Monome mx_i und $m'\overline{x}_i$ mit $mm' \neq 0$ heißt mm' Konsensus von mx_i und $m'\overline{x}_i$. Wir führen noch zwei Sprechweisen ein. Gilt obige Definition, so sagen wir auch „Die Monome mx_i und $m'\overline{x}_i$ bilden den Konsensus mm'." Gibt es ein Polynom p, in dem zwei Monome einen Konsensus bilden, so sagen wir: „Es existiert der Konsensus mm' auf p."

Einen Spezialfall eines Konsensus, einen einfachen Konsensus, haben wir schon beim Algorithmus von Quine/McCluskey kennengelernt. Hier galt jeweils $m = m'$.

Frage 2.27: Zeige, dass ein Konsensus zu einem Polynom, auf dem der Konsensus existiert, hinzugefügt werden kann, ohne dass die Funktion verändert wird!
Wir wollen also zeigen, dass ein Konsensus mm', der auf einem Polynom p existiert, zu p hinzugefügt werden kann, ohne dass $f = p$ verändert wird. Zu zeigen ist also $f = p \vee mm'$. Wie zeigen wir dies nun? Der Konsensus mm' existiert auf p, d.h. die Monome mx_i und $m'\overline{x}_i$ kommen in p vor. Im Prinzip müssen wir also nur zeigen, dass $mx_i \vee m'\overline{x}_i = mx_i \vee m'\overline{x}_i \vee mm'$ gilt. Dies können wir einfach über eine Fallunterscheidung für x_i zeigen.

Fall 1: $x_i = 0$. Dann gilt: $x_i = 0$

$$\begin{aligned} m' &= m0 \vee m'1 = mx_i \vee m'\overline{x}_i \\ &= mx_i \vee m'\overline{x}_i \vee mm' = m0 \vee m'1 \vee mm' = m' \vee mm' = m'(1 \vee m) = m'1 = m' \end{aligned}$$

Fall 2: $x_i = 1$. Dann gilt: $x_i = 1$

$$\begin{aligned} m &= m1 \vee m'0 = mx_i \vee m'\overline{x}_i \\ &= mx_i \vee m'\overline{x}_i \vee mm' = m1 \vee m'0 \vee mm' = m \vee mm' = m(1 \vee m') = m1 = m \end{aligned}$$

Frage 2.28: Welche Aussage ist entscheidend für die Methode des iterierten Konsensus?
Die folgende Aussage ist entscheidend für die Methode des iterierten Konsensus: Es sei p ein Polynom, welches die Funktion f darstellt und t ein Implikant von f. Dann gilt mindestens eine der beiden Aussagen:

1. t hat eine Verkürzung in p.
2. Es gibt einen Konsensus auf p, der keine Verkürzung in p hat.

Die Aussage wird durch eine umgekehrte Induktion über die Länge von t geführt. Der Induktionsanfang wird dabei für Implikanten der Länge $l = n$ gemacht. Dies sind offensichtlich Minterme und für diese existiert eine Verkürzung in p. Denn ansonsten würde p nicht die Funktion f darstellen. Der Minterm $m_a(x)$ nimmt für die Eingabe a den Wert 1 an, d.h. $m_a(a) = 1$. Damit p die Funktion f darstellt, muss es mindestens ein Monom in p geben, welches für a auch den Wert 1 annimmt. Da Monome $\wedge$-verknüpft sind, muss das Monom also eine Verkürzung von m_a sein, damit es ebenfalls den Wert 1 liefert.

Wenn t kein Minterme ist, muss die Länge von t echt kleiner als n sein. Dann gibt es jedoch eine Variable x_i, die weder positiv noch negativ in t vorkommt. Im Beweis zum Algorithmus von Quine/McCluskey haben wir schon folgende Aussage bewiesen:

$$t \in I(f) \;\Leftrightarrow\; tx_i, t\overline{x}_i \in I(f)$$

Da die Monome tx_i und $t\overline{x}_i$ also auch Implikanten sind und eine echte Verlängerung von t darstellen, können wir auf sie also die Induktionsvoraussetzung anwenden. Wenn für mindestens eines der beiden Monome tx_i oder $t\overline{x}_i$ Bedingung 2 gilt, so sind wir fertig, da diese Aussage dann natürlich auch für t gilt. Das liegt daran, dass Aussage 2 unabhängig von t ist.

Gehen wir also davon aus, dass nach Induktionsvoraussetzung für beide Monome Bedingung 1 gilt (die ja auch als einzige Bedingung sicher für den Induktionsanfang gilt). Dann gibt es für tx_i und $t\overline{x}_i$ also Verkürzungen in p. Diese sind der Form t_1x_i und $t_2\overline{x}_i$, denn wenn x_i bzw. $\overline{x}_i$ nicht in den Verkürzungen vorkäme, so gäbe es direkt eine Verkürzung von t. Die beiden Verkürzungen t_1x_i und $t_2\overline{x}_i$ bilden somit den Konsensus t_1t_2, der eine Verkürzung von t ist, da t_1 und t_2 Verkürzungen von t sind. Für t_1t_2 gibt es nun wiederum zwei Möglichkeiten: Entweder t_1t_2 hat eine Verkürzung in p, womit dann auch t eine Verkürzung in p hat und Aussage 1 gilt. Oder aber t_1t_2 hat keine Verkürzung in p, womit dann Aussage 2 gilt.

Frage 2.29: Wie wird die entscheidende Aussage bei der Methode des iterierten Konsensus verwendet?

Wir wissen nach obigem Satz nun, dass zu jedem Implikanten einer Funktion entweder eine Verkürzung im Polynom p zu f enthalten ist oder es einen Konsensus auf p gibt, für den keine Verkürzung in p existiert. Wir werden nun zuerst zeigen, dass wenn p ein Polynom für eine Funktion f ist und es für jeden Konsensus mm' auf p eine Verkürzung in p gibt (also Aussage 2 nicht mehr gilt), dass dann p ein Pinup-Polynom für f ist.

Wir betrachten einen Primimplikanten $t \in PI(f)$. Da dieser natürlich ein Implikant ist, gilt also die obige Aussage. Nach Voraussetzung soll jedoch Aussage 2 nicht mehr gelten. Dann gibt es für t also eine Verkürzung in p. Da t jedoch Primimplikant ist, muss die Verkürzung t selbst sein. Damit ist jeder Primimplikant in p enthalten und p ein Pinup-Polynom.

Ziel der Methode des iterierten Konsensus wird es demnach sein, so lange einen Konsensus zu bilden und zu p hinzuzufügen, bis zu jedem Konsensus eine Verkürzung in p enthalten ist. Nach der obigen Erkenntnis ist p dann Pinup-Polynom.

Algorithmus 3 Algorithmus des iterierten Konsensus

Eingabe: Ein Polynom p.

Vereinfache das Polynom, d.h. streiche Duplikate und Verlängerungen.
while es existiert ein Konsensus mm' auf p, der keine Verkürzung in p hat **do**
  Füge mm' zu p hinzu.
  Vereinfache das Polynom.
end while
Liefere das Polynom zurück.

Frage 2.30: Zeige die Korrektheit der Methode des iterierten Konsensus!

Die Korrektheit des Verfahrens wurde schon zuvor gezeigt. Im Prinzip bleibt daher nur noch zu zeigen, dass der Algorithmus auch tatsächlich terminiert. In jedem Schritt wird nur ein Konsensus hinzugefügt, für den keine Verkürzung im Polynom vorhanden ist. Wenn also ein Monom m hinzugefügt wird, bleibt es bis zum Ende des Algorithmus vorhanden, es sei denn, es wird gestrichen, weil eine Verkürzung hinzugefügt wird. Dies bedeutet jedoch, dass jedes Monom m höchstens einmal zu p hinzugefügt wird. Andererseits gibt es bei n Eingabebits nur 3^n Monome und somit endlich viele. Nach endlich vielen Schritten muss der Algorithmus also terminieren.

Frage 2.31: Welches Korollar ergibt sich aus der Methode des iterierten Konsensus?
Es ergibt sich das folgende Korollar: Wenn ein Polynom p für eine Funktion f vorliegt, auf dem es keinen Konsensus gibt, dann ist p Pinup-Polynom. Denn dann muss für jeden Implikanten und somit auch für jeden Primimplikanten Fall 1 vorliegen. Das Polynom p enthält also für jeden Implikanten eine Verkürzung. Also muss es alle Primimplikanten enthalten und sonst nur Implikanten.

Wie können wir dieses Korollar nun verwenden? Es sei eine Funktion gegeben, die durch ein Polynom beschrieben wird, in dem nur positive Literale vorkommen. Aus dem Korollar folgt, dass das Polynom dann auch gleich ein Pinup-Polynom für diese Funktion sein muss. Denn da nur positive Literale vorkommen, kann es ja gar keinen Konsensus geben. Wir werden dieses Korollar auch im dem Beweis benutzen, der zeigt, dass es Funktionen gibt, auf denen die Baummethode eine exponentielle Laufzeit haben muss.

Frage 2.32: Was besagt der Satz von McMullen und Shearer?
Der Satz von McMullen und Shearer besagt, dass es Funktionen f gibt, die durch Polynome mit k Monomen dargestellt werden können und die $2^k - 1$ Primimplikanten haben. McMullen und Shearer haben jedoch auch gezeigt, dass diese Anzahl an Primimplikanten eine obere Schranke ist. Dies bedeutet, dass jede Funktion, die mit k Monomen dargestellt werden kann, höchstens $2^k - 1$ Primimplikanten haben kann. Uns interessiert hier jedoch nur der erste Ergebnis, d.h. wir werden eine konkrete Funktion betrachten, die diese maximale Anzahl an Primimplikanten aufweist. Die Funktion $f_k \in B_{2k-1}$ ist induktiv definiert. Es gilt $f_1(x_1) = x_1$ und

$$f_{k+1} = x_{2k} f_k(x_1, \ldots, x_{2k-1}) \vee \overline{x}_{2k} x_{2k+1}$$

Zuerst zeigen wir per Induktion, dass f_{k+1} mit $k+1$ Monomen darstellbar ist. Der Induktionsbeginn ist $f_1(x_1) = x_1$ und es ist offensichtlich, dass die Behauptung stimmt. Nun gehen wir davon aus, dass f_k mit k Monomen darstellbar ist. Die Funktion f_{k+1} entsteht, indem wir zu allen Monomen von f_k das Literal x_{2k} hinzu multiplizieren und indem wir $\overline{x}2kx_{2k+1}$ addieren. Bei der Multiplikation entstehen genau k neue Ausdrücke, die nicht reduzierbar sind, wenn f_k schon reduziert ist. Der Ausdruck $\overline{x}_{2k}x_{2k+1}$ ist ebenfalls nicht reduzierbar, da diese beiden Variablen in f_k überhaupt nicht vorkommen. Damit besteht die Funktion f_{k+1} also aus $k+1$ Monomen.

Als nächstes zeigen wir nun, ebenfalls per Induktion, dass f_k genau $2^k - 1$ Primimplikanten besitzt. Die Funktion $f_1(x_1) = x_1$ besitzt genau einen Primimplikanten, nämlich x_1. Nach Induktionsvoraussetzung hat f_k genau $2^k - 1$ Primimplikanten. Wir werden nun zeigen, dass dann für jeden Primimplikant $m \in PI(f_k)$ die Monome $x_{2k}m$ und $x_{2k+1}m$ Primimplikanten von f_{k+1} sind. Für jeden Primimplikanten von f_k ergeben sich also zwei neue Primimplikanten. Zusätzlich ist $\overline{x}_{2k}x_{2k+1}$ ein Primimplikant. Insgesamt ergeben sich also $2(2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1$ Primimplikanten. Zuerst zeigen wir, dass $x_{2k}m$ und $x_{2k+1}m$ Implikanten von f_{k+1} sind. Es sei also $m \in PI(f_k)$. Dann ist $x_{2k}m$ offensichtlich ein Implikant von f_{k+1}. Denn damit $x_{2k}m(a) = 1$ gilt, muss zum einen $a_{2k} = 1$ gelten und zum anderen ist m ein Implikant von f_k. Also muss auch eines der Monome in der Darstellung von f_k für a den Wert 1 ergeben und damit auch ein Summand des Produktes zwischen x_{2k} und f_k. Ebenso gilt, dass $x_{2k+1}m$ ein Implikant von f_{k+1} ist. Es gelte $x_{2k+1}m(a) = 1$. Damit muss also $a_{2k+1} = 1$ sein. Wir unterscheiden nun zwei Fälle, in Abhängigkeit von der Eingabe a_{2k}. Gilt $a_{2k} = 0$, so wird $\overline{x}_{2k}x_{2k+1}$ den Wert 1 liefern und somit $f_{k+1}(a) = 1$ gelten. Im Fall $a_{2k} = 1$ gilt ansonsten $x_{2k}m(a) = 1$ und ebenfalls $f_{k+1}(a) = 1$. Damit haben wir gezeigt, dass $x_{2k}m$ und

$x_{2k+1}m$ Implikanten sind. Es bleibt also noch der Beweis, dass sie auch Primimplikanten sind. Dazu betrachten wir ihre echten Verkürzungen. Zuerst zeigen wir, dass $m \in PI(f_k)$ kein Implikant für f_{k+1} sein kann. Denn m hängt nur von den Variablen $x_1, \ldots, x_{2k-1}$ ab. Es gibt daher Eingaben a mit $a_{2k} = 0$ und $a_{2k+1} = 0$, für die jedoch $m(a) = 1$ gilt. Wenn jedoch $a_{2k} = a_{2k+1} = 0$ gilt, muss auch $f(a) = 0$ gelten. Nun betrachten wir die echten Verkürzungen m' von m. Es kann $x_{2k}m'$ kein Primimplikant von f_{k+1} sein. Denn in diesem Fall wäre m' auch ein Implikant von f_k, was im Widerspruch dazu steht, dass die Verlängerung m von m' ein Primimplikant von f_k ist. Auch für $x_{2k+1}m'$ ergibt sich ein solcher Widerspruch. Er ist nur etwas komplizierter zu zeigen. Wir gehen also davon aus, dass $x_{2k+1}m'$ ein Implikant von f_{k+1} ist. Da m' eine (echte) Verkürzung von m ist, kann m' also höchstens über den Variablen $x_1, \ldots, x_{2k-1}$ definiert sein. Wir betrachten eine Eingabe $a' \in \{0,1\}^{2k-1}$ mit $m'(a') = 1$. Aus dieser Eingabe wollen wir nun eine Eingabe a bauen, die einen Widerspruch ergibt. Wenn wir a' entsprechend gewählt haben, haben wir noch eine Wahl bei der Belegung der Variablen x_{2k} und x_{2k+1}. Da $x_{2k+1}m'(a) = 1$ gelten soll, können wir a_{2k+1} jedoch nicht wirklich frei wählen, da $a_{2k+1} = 1$ gelten muss. Weiter wählen wir $a_{2k} = 1$. So wie wir a gewählt haben, gilt nun: $x_{2k+1}m'(a) = 1$. Da $x_{2k+1}m'$ ein Implikant für f_{k+1} sein soll, folgt daraus $f_{k+1}(a) = 1$. Der „hintere Teil“ von k_{k+1}, also $\overline{x}_{2k}x_{2k+1}$ kann jedoch nicht dafür gesorgt haben, dass f_{k+1} zu 1 ausgewertet wird, was an unserer Wahl von a_{2k} liegt. Also muss gelten $f_{k+1}(a) = 1 f_k(a) \vee 0 = f_k(a) = 1$. Da f_k nur von m' abhängig sein kann, wäre m' jedoch ein Implikant von f_k, was wieder im Widerspruch dazu steht, dass m ein Primimplikant von f_k ist.

Frage 2.33: Gebe eine Funktion an, für die die Baummethode eine exponentielle Laufzeit haben muss!

Wir wollen nun zeigen, dass es Funktionen $f \in B_n$ mit $n = m^2$ gibt, für die die Baummethode bei Eingabe aller Primimplikanten mindestens 2^m $(= 2^{\sqrt{n}})$ Schritte benötigt. Auf die Wahl der Zerlegungsvariablen wird es dabei nicht ankommen. Wir definieren:

$$f_n\left(\begin{bmatrix} x_1 & x_2 & \ldots & x_m \\ x_{m+1} & x_{m+2} & \ldots & x_{2m} \\ \vdots & \ddots & \ddots & \vdots \\ x_{(m-1)m+1} & x_{(m-1)m+2} & \ldots & x_{m^2} \end{bmatrix}\right) = 1 \Leftrightarrow \text{ Es gibt in der Matrix eine 1-Zeile}$$

Die Aufgabe der Funktion ist es also, nachzuschauen, ob in einer $m \times m$-Matrix eine der Zeilen komplett mit Einsen belegt ist. Ein Implikant dieser Funktion ist also ein Monom, welches mindestens alle Variablen einer Zeile enthält. Ein Primimplikant ist dementsprechend ein Monom, welches nur die Variablen einer Zeile enthält. Das Polynom, welches f_n beschreibt, sieht also wie folgt aus:

$$f_n(x) = \bigvee_{1 \le i \le m} x_{(i-1)m+1} \ldots x_{im}$$

Wir machen uns erst einmal klar, dass dieses Polynom genau alle Primimplikanten enthält. Es lässt sich nichts mehr wegkürzen und es wird sich um jede Zeile „gekümmert“. Desweiteren können wir das Korollar zur Methode des iterierten Konsensus benutzen. Alle Variablen sind positiv, weshalb es keinen Konsensus mehr geben kann. Also muss das Polynom alle Primimplikanten enthalten.

Wieso macht die Baummethode nun mindestens 2^m Schritte? Die Baummethode läuft in der Rekursion zurück, wenn sie auf eine konstante Subfunktion trifft. Wir überlegen uns nun,

wann dies frühestens passieren kann. In jedem Schritt wird eine Zerlegungsvariable gewählt und es werden die entsprechenden Subfunktionen gebildet. Eine Subfunktion, die sich dadurch ergibt, dass die Zerlegungsvariable auf 0 gesetzt wird, sieht genau so aus wie die ursprüngliche Funktion, nur dass das Monom, in dem die Zerlegungsvariable vorkommt, komplett wegfällt, da es zu 0 wird. Erst wenn alle Monome weggefallen sind, kann eine Subfunktion die konstante Funktion 0 sein. Da es in f_n genau m Monome gibt, müssen also mindestens m Zerlegungsschritte gemacht werden, damit dies passieren kann. Eine Subfunktion, die dadurch entsteht, dass die Zerlegungsvariable auf 1 gesetzt wird, sieht genauso aus, wie die ursprüngliche Funktion, nur dass die Zerlegungsvariable wegfällt. Wenn nach einigen Zerlegungen in einem Monom alle Variablen weggefallen sind, dann wird es zur konstanten Funktion 1 und somit die ganze Subfunktion zur konstanten Subfunktion 1. Da jedes Monom m verschiedene Variablen hat, kann dieser Fall auch frühestens nach m Zerlegungsschritten eintreten. Es sind also stets mindestens m Zerlegungen nötig. Dabei entsteht also mindestens ein vollständiger binärer Baum der Tiefe $m-1$. Auf jeder Ebene des Baumes werden 2^i Zerlegungen gemacht. Also beträgt die Gesamtzahl an Zerlegungen mindestens:

$$\sum_{i=0}^{m-1} 2^i = 2^m - 1 = \Omega(2^m)$$

Frage 2.34: Gebe eine Funktion an, für die die Methode des iterierten Konsensus eine exponentielle Laufzeit aufweisen kann!

Wir wollen nun zeigen: Es gibt Funktionen aus $k+2$ Monomen, für die die Methode des iterierten Konsensus mehr als 2^k Schritte benötigen kann, um den einzigen Primimplikanten zu finden. Wir definieren:

$$g_k = f_k \vee x_{2k} \vee \overline{x}_{2k}$$

Dabei ist f_k die induktiv definierte Funktion, die beim Satz von McMullen und Shearer eingeführt wurde. Zuerst einmal überlegen wir uns, was die Funktion g_k berechnet. Wir sehen schnell, dass $g_k \equiv 1$ ist, dass also g_k die konstante Funktion 1 darstellt. Denn für die Monome x_{2k} und $\overline{x}_{2k}$ existiert der Konsensus 1, den wir zum Polynom hinzufügen können. Dann wird jedoch die ganze Funktion zu 1. Es kann nun jedoch passieren, dass die Methode des iterierten Konsensus zuerst alle Primimplikanten für f_k berechnet und erst im letzten Schritt den Konsensus zwischen x_{2k} und $\overline{x}_{2k}$ findet. Da wir zuvor schon bewiesen haben, dass f_k genau $2^k - 1$ Primimplikanten besitzt, müssen in einem solchen Fall also mindestens 2^k Schritte gemacht werden.

2.4 Überdeckungsprobleme

Frage 2.35: Nochmal zur Wiederholung: Wie ist das Überdeckungsproblem motiviert?

In den vorherigen Kapiteln haben wir Verfahren kennengelernt, mit denen sich die Menge der Primimplikanten einer Funktion berechnen ließ. Um die jeweilige Funktion in Hardware umzusetzen (z.B. mit PLAs), wäre es rein theoretisch möglich, einfach die Disjunktion aller Primimplikanten zu realisieren. Oft werden zur Darstellung einer Funktion jedoch gar nicht alle Primimplikanten benötigt. Wenn wir Hardwarekosten sparen möchten (und das möchten wir), sollten wir also die kleinstmögliche Teilmenge von Primimplikanten finden, deren Disjunktion die Funktion beschreibt.

	0010	0011	0100	0101	0111	1010	1011	1110	1111
$\overline{w}x\overline{y}$	0	0	1	1	0	0	0	0	0
$\overline{w}xz$	0	0	0	1	1	0	0	0	0
$\overline{x}y$	1	1	0	0	0	1	1	0	0
yz	0	1	0	0	1	0	1	0	1
wy	0	0	0	0	0	1	1	1	1

Tabelle 1: Beispiel für eine PI-Tafel

Frage 2.36: Was ist ein Primimplikantentafel (PI-Tafel)?

Eine Primimplikantentafel kann immer dann verwendet werden, wenn die zu minimierende Funktion als DNF beschrieben ist. Die PI-Tafel ist eine 0-1-Matrix, in deren Zeilen die Primimplikanten der Funktion stehen. Diese bezeichnen wir mit $m_1^*, \ldots, m_k^*$. In den Spalten findet sich für jeden der Minterme der DNF $m_{a(1)}, \ldots, m_{a(s)}$ ein Eintrag. Dabei hat der Minterm $m_{a(k)}$ den Eintrag $a(k)$, also die Eingabe, für die der Minterm 1 wird. Der Matrixeintrag an der Position (i, j) ist $m_i^*(a(j))$. Dies ist der Wert, den der Primimplikant für die entsprechende Eingabe berechnet. Tabelle 1 zeigt ein Beispiel für eine PI-Tafel.

Wenn $m_i^*(a(j)) = 1$ ist, dann sagen wir, dass der Primimplikant m_i^* den Minterm $m_{a(j)}$ bzw. die Eingabe $a(j)$ überdeckt. Eine Summe $m_{i(1)}^* \vee \cdots \vee m_{i(r)}^*$ von Primimplikanten stellt eine Funktion f genau dann dar, wenn jede Eingabe $a(j)$ von mindestens einem Primimplikanten $m_{i(l)}^*$ überdeckt wird. Ein Minimalpolynom ist dann eine f darstellende Summe mit minimalen Kosten.

Frage 2.37: Welche Reduktionsregeln gibt es für PI-Tafeln? Was ist eine reduzierte PI-Tafel?

Es können drei verschiedene Reduktionsregeln für PI-Tafeln unterschieden werden, wobei die reduzierte PI-Tafel durch Anwendung dieser Reduktionsregeln entsteht.

1. Falls eine Spalte in der PI-Tafel nur eine 1 enthält, muss der zugehörige Primimplikant m_i^* auf jeden Fall im Minimalpolynom enthalten sein. Es kann dann die i-te Zeile, sowie alle Spalten, die an der i-ten Stelle eine 1 stehen haben, gestrichen werden, da diese dann ebenfalls schon durch m_i^* überdeckt werden.

2. Falls die j'-te Spalte an keiner Stelle kleiner ist als die j-te Spalte (also die j'-te Spalte an jeder Stelle größer oder gleich der j-ten Spalte ist), dann kann die j'-te Spalte gestrichen werden.

3. Falls die i-te Zeile an keiner Stelle kleiner ist als die i'-te Zeile (also an jeder Stelle größer oder gleich ist), kann die i'-te Zeile gestrichen werden.

Regel 2 mag auf den ersten Blick etwas unlogisch erscheinen. Schließlich werfen wir ja eine Spalte weg, die mehr Einsen als eine andere Spalte enthält und somit viel mehr Möglichkeiten bietet. So würden wir zum Beispiel in Tabelle 1 die Spalte für die Eingabe 0101 streichen, da diese an jeder Stelle größer oder gleich der Spalte für 0100 ist. Im Prinzip ist jedoch die größere Anzahl an Einsen genau das Argument diese Spalte zu streichen. Im Beispiel könnte es z.B. passieren, dass wir zur Überdeckung von 0101 den Primimplikanten $\overline{w}xz$ wählen und dann zur Überdeckung von 0100 nur den Primimplikanten $\overline{w}x\overline{y}$ wählen können. Es würde jedoch ausreichen, nur den Primimplikanten $\overline{w}x\overline{y}$ zu wählen, da durch diesen auch 0101 überdeckt

	0111
$\overline{w}xz$	1
yz	1

Tabelle 2: Die reduzierte PI-Tafel (nach Anwendung von Regel 1 und 2 auf Tabelle 1)

	001	010	011	100	101	110
$x\overline{y}$	0	0	0	1	1	0
$\overline{x}y$	0	1	1	0	0	0
$x\overline{z}$	0	0	0	1	0	1
$\overline{x}z$	1	0	1	0	0	0
$y\overline{z}$	0	1	0	0	0	1
$\overline{y}z$	1	0	0	0	1	0

Tabelle 3: Die nicht reduzierbare PI-Tafel für MD_3

wird. Das Streichen von Spalten mit mehr Einsen bewahrt uns also davor, unnötig viele Primimplikanten auszuwählen.

Regel 3 sorgt dafür, dass Primimplikanten, die eine Teilmenge der Eingaben überdecken, die ein anderer Primimplikant überdeckt, gestrichen werden. Denn in diesem Fall kann es nur günstiger sein, von vornherein den Primimplikanten zu wählen, der mehr Eingaben überdeckt. Es sei jedoch angemerkt, dass die durch den anderen Primimplikanten überdeckte Teilmenge der Eingaben nicht eine echte Teilmenge sein muss, d.h. die Regel kann auch für Primimplikanten angewandt werden, welche die gleichen Eingaben überdecken.

Wenn wir auf die PI-Tafel aus Tabelle 1 die ersten beiden Reduktionsregeln anwenden, ergibt sich die reduzierte PI-Tafel aus Tabelle 2. Es wird schnell ersichtlich, dass auf diese auch noch Regel 3 angewandt werden kann. In diesem Fall überdecken beide Primimplikanten gleich viele Eingaben, was bedeutet, dass wir uns aussuchen können, welchen von den beiden Primimplikanten wir streichen.

Frage 2.38: Was ist ein Kernimplikant?

Ein Kernimplikant ist ein Primimplikant der Aufgrund von Regel 1 ausgewählt wurde, der also als einziger Primimplikant eine Eingabe überdeckt. Die Kernimplikanten in Tabelle 1 sind $\overline{x}y$, $\overline{w}x\overline{y}$ und wy, womit diese also auf jeden Fall im Minimalpolynom vorkommen müssen. Wenn wir uns die Funktionen wie in [Molitor und Scholl 1999] durch n-dimensionale Würfel dargestellt vorstellen, sind die Kernimplikanten die Eingaben, die nur durch eine Kante, Fläche, etc. überdeckt werden.

Frage 2.39: Gebe ein weiteres Beispiel für die Berechnung eines Minimalpolynoms anhand der PI-Tafel!

Wir betrachten dazu die Funktion MD_3. Es gilt $n = 3 \cdot k = 3$, womit $k = 1$ folgt. Die Funktion wird also für alle Eingaben 1, in denen eine oder zwei Einsen vorkommen. In Kapitel 2.2 haben wir schon herausgefunden, dass die Primimplikanten dieser Funktion die Monome sind, die genau $k = 1$ positive und $k = 1$ negative Literale enthalten. Damit ergeben sich die Primimplikanten $x\overline{y}$, $\overline{x}y$, $x\overline{z}$, $\overline{x}z$, $y\overline{z}$ und $\overline{y}z$. Tabelle 3 stellt die PI-Tafel dar, welche auch nicht weiter reduzierbar ist.

Da jeder Primimplikant genau zwei Eingaben überdeckt, benötigen wir mindestens drei Primimplikanten, um alle sechs Eingaben zu überdecken. Drei Primimplikanten reichen jedoch

auch. Dies sind:

$$MD_3 = x\overline{y} \vee y\overline{z} \vee \overline{x}z = \overline{x}y \vee \overline{y}z \vee x\overline{z}$$

Frage 2.40: Warum ist die Berechnung eines Minimalpolynoms mit Hilfe der PI-Tafel NP-hart?

Das Problem der Berechnung eines Minimalpolynoms mit Hilfe der PI-Tafel ist NP-hart, da es auf das Problem SET COVER reduziert werden kann. Beim Problem SET COVER sind k Mengen $A_1, \ldots, A_k \subseteq \{1, \ldots, n\}$ gegeben und es soll die minimale Anzahl an Mengen herausgefunden werden, deren Vereinigung die Menge $\{1, \ldots, n\}$ ergibt.

Im Falle der Berechnung eines Minimalpolynoms mit Hilfe der PI-Tafel können wir die Menge $\{1, \ldots, n\}$ mit der Menge der Eingaben $\{a(1), \ldots, a(s)\}$ identifizieren und die Teilmengen $A_1, \ldots, A_k$ sind durch die Mengen der Eingaben gegeben, die durch den jeweiligen Primimplikanten überdeckt werden.

Frage 2.41: Welche Methode kennst du zur Berechnung eines Minimalpolynoms mit Hilfe der PI-Tafel?

Wir haben schon gezeigt, dass die Berechnung eines Minimalpolynoms mit Hilfe der PI-Tafel ein NP-hartes Problem ist. In der Vorlesung „Effiziente Algorithmen“ werden verschiedene Methoden vorgestellt, um NP-harte Probleme anzugehen, unter anderem die Branch-And-Bound Algorithmen. Branch-And-Bound Algorithmen sind heuristische Verfahren, mit deren Hilfe in jedem Fall eine optimale Lösung berechnet werden kann. In einigen Fällen müssen diese Verfahren dabei jedoch eine exponentielle Laufzeit aufweisen, da ansonsten $P = NP$ gelten würde. Wir hoffen nun, dass die Rechenzeit für viele für uns interessante Probleme in einem annehmbaren Rahmen bleibt.

Bei Branch-And-Bound Algorithmen wird jeweils eine Menge von Teilproblemen betrachtet, die zusammen das Gesamtproblem darstellen. Zu Beginn existiert nur das Gesamtproblem P, in unserem Fall die gesamte PI-Tafel. Für jedes Teilproblem P_i wird eine untere Schranke L_i und eine obere Schranke U_i für den Wert einer optimalen Lösung berechnet. Desweiteren werden eine globale untere Schranke L und eine obere Schranke U verwaltet. In jedem Schritt wird ein unzerlegtes Teilproblem in weitere Teilprobleme zerlegt, die Schranken dieser Teilprobleme berechnet und die globalen Schranken aktualisiert. Eine optimale Lösung ist gefunden, wenn $U = L$ gilt, also die Werte der globalen Schranken zusammenfallen. Es werden nun die einzelnen Module und ihre Aufgaben vorgestellt, wobei wir im Hinterkopf behalten müssen, dass das Überdeckungsproblem ein Minimierungsproblem ist.

Lower Bound: Diese Modul berechnet eine möglichst gute untere Schranke für das betrachtete Teilproblem. Bei Branch-And-Bound Algorithmen werden die Schranken, in deren Richtung wir optimieren wollen, stets über eine Relaxation berechnet, d.h. das Problem wird nicht ganz so genau betrachtet. Allerdings müssen alle Lösungen für das nicht relaxierte Problem in der Menge der Lösungen für das relaxierte Problem enthalten sein, d.h. es dürfen keine möglichen gültigen Lösungen wegfallen. Da wir ein Minimierungsproblem betrachten und daher in Richtung der unteren Schranke optimieren wollen, wird in unserem Fall also auch ein gelockerte Betrachtung vorgenommen.

Upper Bound: Die Aufgabe diese Moduls ist es, eine möglichst gute obere Schranke zu berechnen. Dies ist eine möglichst kleine obere Schranke. Dazu werden oft Greedy Algorithmen verwendet, welche effizient eine gültige Lösung berechnen. In unserem Fall wird ebenfalls ein Greedy Algorithmus verwendet werden.

Branching: Die Aufgabe des Branching Moduls ist es, ein unzerlegtes Teilproblem in weitere Teilprobleme vom selben Problemtyp wie das ursprüngliche Problem zu zerlegen.

Nachdem die einzelnen Module und ihre Aufgaben allgemein vorgestellt wurden, werden wir nun die konkreten Module für das Überdeckungsproblem vorstellen:

Lower Bound: Das Lower Bound Modul arbeitet mit einer Relaxation bzw. in unserem Fall mit einer groben Abschätzung. Es sei z_i die Anzahl der Eingaben, die der Primimplikant m_i^* überdeckt. Wir sortieren diese Anzahlen und benennen die Primimplikanten so um, dass $z_1 \geq \cdots \geq z_k$ gilt. Dann benötigt das Minimalpolynom mindestens r Primimplikanten, wenn r der kleinste Index ist, so dass gilt:

$$\sum_{i=1}^{r} z_i \geq \text{Spaltenanzahl (Anzahl aller Eingaben)}$$

Es wird also quasi optimistischerweise davon ausgegangen, dass wir die Primimplikanten mit einer Greedy Methode wählen können und die durch diese Primimplikanten überdeckten Eingaben größtenteils disjunkt sind. Es gelte z.B. $z_1 = 10,\ z_2 = 6,\ z_3 = 3$ und $z_4 = 3$. Wenn die Tabelle 17 Spalten hat, muss $r = 3$ gelten, da wir für $r = 2$ nur $10 + 6 = 16$ Spalten abdecken würden und somit noch mindestens ein weiterer Primimplikant benötigt wird.

Upper Bound: Da es sich um ein Minimierungsproblem handelt, berechnet das Upper Bound eine zulässige Lösung. Dabei wird ein einfacher Greedy Ansatz benutzt. In jedem Schritt wird der Primimplikant gewählt, der die meisten Eingaben überdeckt. Dann wird der Primimplikant, sowie alle durch ihn überdeckten Eingaben gestrichen und das Restproblem (also eine kleinere PI-Tafel) ebenfalls greedy gelöst. Dieser Ansatz liefert ein Polynom, welches aus den auf diese Art und Weise ausgewählten Primimplikanten besteht, von dem wir hoffen, dass es gut ist.

Branching: In diesem Schritt wird ein Teilproblem P_i in weitere Teilprobleme vom selben Typ aufgeteilt. Wir werden ein Teilproblem in zwei disjunkte Teilprobleme aufteilen. Dazu wählen wir einen Primimplikanten aus und bilden ein Teilproblem, in dem dieser Primimplikant auf jeden Fall gewählt wird (Inklusion), und ein Teilproblem, in dem der Primimplikant auf keinen Fall gewählt wird (Exklusion). Welchen Primimplikanten sollen wir dafür wählen? Wir wählen den Primimplikanten m_i^*, der die meisten Eingaben überdeckt. Die Inklusion wird realisiert, indem der Primimplikant, sowie alle durch ihn überdeckten Eingaben aus der PI-Tafel entfernt werden. Bei der Exklusion wird lediglich der Primimplikant entfernt.

Wie läuft nun der eigentlich Algorithmus ab? Es wird davon ausgegangen, dass in jedem Schritt die PI-Tafeln reduziert werden und die Schranken L_i' und U_i' für die reduzierten Probleme berechnet werden. Bei der Reduktion kann es passieren, dass einige Kernimplikanten auf jeden Fall gewählt werden. Also müssen wir die Kosten der Kernimplikanten sowie die Kosten der durch Inklusion gewählten Primimplikanten zu L_i' bzw. U_i' hinzu addieren, um die Kosten für L_i bzw. U_i zu bekommen.

Als Datenstruktur wird ein Branch-And-Bound Baum (BBB) verwendet, an dessen Blättern jeweils disjunkte Teilprobleme stehen, die zusammen das Gesamtproblem darstellen. Das Vorgehen ist dann wie in Algorithmus 4 beschrieben. Bei diesem Algorithmus dürften alle Schritte

Algorithmus 4 Branch-And-Bound Algorithmus für das Überdeckungsproblem

Initialisierung: Initialisiere den BBB durch einen Baum mit einem Knoten, der das Gesamtproblem darstellt.

Aktualisierung: Berechne die aktuellen Werte U und L als das Minimum aller L_i bzw. U_i an den Blättern des Baumes.

Fertig? Ist $L = U$, so ist das Polynom, welches zur oberen Schranke U gehört ein Minimalpolynom. Stop.

Branch: Ist $L < U$, dann zerlege das Problem P_i, welches zur unteren Schranke L gehört in zwei Teilprobleme, die im BBB Söhne von P_i werden. Gehe zum Schritt „Aktualisierung".

nachvollziehbar sein, bis auf die Wahl des Problems P_i, welches im Schritt „Branch" zerlegt wird. Dazu betrachten wir ein Beispiel. Es seien z.B. zwei Teilprobleme P_1 und P_2 gegeben, deren Schranken $L_1 = 17$ und $U_1 = 23$ sowie $L_2 = 19$ und $U_2 = 22$ sind. Wenn wir nun zuerst P_2 weiter zerlegen würden, dann würden dabei nur Lösungen entstehen, die im Intervall $[19, 22]$ liegen. Die untere Schranke von P_1 sagt uns jedoch, dass unter Umständen auch noch Lösungen im Bereich $[17, 19]$ möglich sind. Wenn wir zuerst P_2 betrachten, dann müssen wir also auf jeden Fall auch noch P_1 betrachten. Wird jedoch zuerst P_1 betrachtet, dann kann es passieren, dass eine Lösung im Intervall $[17, 19]$ berechnet wird (dann gilt also auch $U_j \leq 19$) und wir P_2 gar nicht mehr betrachten müssen. Wir können also nur Sparen, wenn stets das Problem mit dem kleinsten L_i-Wert weiter zerlegt wird.

Frage 2.42: Wann wird ein Monom durch ein Polynom überdeckt?
Ein Monom m wird genau dann von einem Polynom p überdeckt, wenn gilt:

$$p(a) = 1 \text{ für alle } a \in m^{-1}(1)$$

Ein Polynom überdeckt also ein Monom, wenn es für alle Eingaben, für die das Monom m eine 1 liefert, ebenfalls eine 1 liefert. Es gilt also für alle a für die $m(a) = 1$ ist, dass auch $p(a) = 1$ ist.

Frage 2.43: Wann wird eine Funktion durch eine Summe von Primimplikanten überdeckt?
Die Überdeckung einer Funktion f durch eine Summe von Primimplikanten ist ziemlich offensichtlich das Ziel des Überdeckungsproblems. Eine Funktion f wird durch eine Summe von Primimplikanten überdeckt, wenn jedes Monom in der Darstellung von f durch die Summe überdeckt wird.

Wir überlegen uns weiter, wo der Unterschied zum bisher betrachteten Überdeckungsproblem liegt. Bisher waren die zu überdeckenden Eingaben als Minterme gegeben, die nur für eine Eingabe eine 1 liefern. Daher reichte auch stets ein Primimplikant, um diese Eingabe zu überdecken. Wenn die Funktion jedoch als Disjunktion von Monomen gegeben ist, kann es sein, dass nicht mehr nur ein Primimplikant ausreicht, um ein Monom zu überdecken. Der Grund dafür liegt darin, dass ein Monom, welches kein Minterm ist, stellvertretend für mehrere Eingaben steht, die auf 1 abbilden. Es kann nun also passieren, dass jeder Primimplikant

für sich nur eine Teilmenge dieser Eingaben abdeckt und somit mehrere Primimplikanten benötigt werden.

Frage 2.44: Was sind die Überdeckungen eines Monoms? Was sind die minimalen Überdeckungen?

Es sei $p^* = m_1^* \vee \cdots \vee m_k^*$ das Polynom aller Primimplikanten von f und m sei ein Implikant von f. Dann ist die Menge $U(p^*, m)$ der Überdeckungen von m durch p^* definiert als alle Teilmengen I von $\{1, \ldots, k\}$ ($I \subseteq \{1, \ldots, k\}$) für die jeweils folgende Bedingungen gelten:

1. $m^{-1}(1) \cap (m_i^*)^{-1}(1) \neq \emptyset$ für alle $i \in I$.
2. Die Summe aller m_i^* mit $i \in I$ überdeckt m.

Die Überdeckungen sind also Mengen von Indizes der Primimplikanten, deren Summe das Monom m überdeckt und die für mindestens eine Eingabe für die m eine 1 liefert ebenfalls eine 1 liefern.

Die Menge $MU(p^*, m)$ der minimalen Überdeckungen von m durch p^* ist definiert als alle $I \in U(p^*, m)$ für die es keine echte Teilmenge in $U(p^*, m)$ gibt.

Frage 2.45: Wie entsteht eine Subfunktion für ein Monom?

Für ein Monom $m = x_{i(1)}^{a(1)} \wedge \cdots \wedge x_{i(r)}^{a(r)}$ entsteht die Subfunktion f_m aus f, indem wir im Polynom, welches f beschreibt, alle Variablen $x_{i(j)}$ durch $a(j)$ ersetzen. Im Gegensatz zur „normalen“ Definition der Subfunktion wird also nicht nur eine Variable auf einen konstanten Wert gesetzt sondern mehrere. Die Variablen werden dabei genau auf die konstanten Werte gesetzt, die dafür sorgen, dass das Monom zu 1 ausgewertet wird.

Es sei zum Beispiel die Funktion gegeben, die durch das Polynom $p = x\overline{y}$ beschrieben wird. Wir wollen nun verschiedene Subfunktionen dieses Polynoms bezüglich verschiedener Monome bilden. Zuerst betrachten wir das Monom $m_1 = \overline{y}z$. Dann ergibt sich die Subfunktion p_{m_1}, indem wir in p die Variable $y = 0$ und $z = 1$ einsetzen. Damit ergibt sich $p_{m_1} = x$. Betrachten wir nun $m_2 = x\overline{y}$, also das Monom, welches dem Polynom p entspricht. Es ist dann offensichtlich, dass p das Monom m_2 überdeckt. Wir bilden nun p_{m_2}, indem wir $x = 1$ und $y = 0$ setzen. Es ergibt sich $p_{m_2} = x\overline{y} = 11 = 1$, also die konstante Funktion 1. Damit sollte man nun schon eine Idee bekommen, wie mit Hilfe von Subfunktionen festgestellt werden kann, ob ein Monom durch ein Polynom überdeckt wird.

Frage 2.46: Wie kann mit Hilfe von Subfunktionen festgestellt werden, ob ein Monom durch ein Polynom überdeckt wird?

Ein Polynom p' überdeckt ein Monom m genau dann, wenn die Subfunktion p'_m die konstante Funktion 1 ist. Die Idee dabei ist die Folgende. Wenn ein Polynom p ein Monom m überdeckt, gilt $m(a) = 1 \;\Rightarrow\; p(a) = 1$. Das Monom sei nun $m = x_{i(1)}^{a(1)} \wedge \cdots \wedge x_{i(r)}^{a(r)}$. Es wird genau dann zu 1 ausgewertet, wenn für die Eingabe $a_{i(1)} = a(1), \ldots, a_{i(j)} = a(j)$ gilt. Ist das Monom kein Minterm, gibt es noch einige Variablen, die nicht im Monom vorkommen, bei denen es egal ist, wie wir diese belegen. Sobald aber $a_{i(1)} = a(1), \ldots, a_{i(j)} = a(j)$ gilt, wird das Monom zu 1 ausgewertet. Also muss das Polynom p für jede solche Belegung ebenfalls zu 1 ausgewertet werden. Wenn wir die entsprechenden Variablen $a(i)$ im Polynom auf die entsprechenden konstanten Werte setzen, die das Monom m zu 1 werden lassen, darf also keine Variable mehr übrig bleiben, von der das Ergebnis von p noch abhängt. Ein einfaches Beispiel: Das Monom $m = x_1x_2$ wird nicht vom Polynom $p = x_1x_2x_3$ überdeckt. Denn wenn wir $x_1 = 1$ und $x_2 = 1$

setzen, ergibt sich $p_m = 11x_3 = x_3$. Das Polynom p_m ist also noch von der Variablen x_3 abhängig. Dies bedeutet, dass z.B. für die Eingabe $a = (1, 1, 0)$ $m = 11 = 1$ und $p = 110 = 0$ gilt. Das Polynom p_m kann also nicht das Monom m überdecken.

Frage 2.47: Wie kann die minimale Überdeckung für ein Monom berechnet werden? Welcher Satz wird dabei zentral verwendet?

Gegeben sei also das Polynom $p^* = m_1^* \vee \cdots \vee m_k^*$, welches die Summe aller Primimplikanten einer Funktion f ist. Desweiteren sei ein Monom m gegeben, welches Implikant der Funktion f ist. Wir möchten nun die Menge $MU(p^*, m)$ berechnen. Dazu betrachten wir zuerst ein Beispiel, um eine Idee zu bekommen, wie die Berechnung erfolgen könnte. Die Funktion in diesem Beispiel ist MD_3, für die $p^* = x\overline{y} \vee \overline{x}y \vee x\overline{z} \vee \overline{x}z \vee y\overline{z} \vee \overline{y}z$ gilt. Das Monom, welches wir überdecken möchten, sei $m = x\overline{y}$. Im ersten Schritt bilden wir nun für jeden Primimplikanten m_i^* die Subfunktion $h_i = (m_i^*)_m$. Die Subfunktionen erhalten wir, indem wir in den Primimplikanten $x = 1$ und $y = 0$ setzen. Dabei ergeben sich die Subfunktionen $1, 0, \overline{z}, 0, 0, z$. Die Subfunktion 1 gehört zum Primimplikant $x\overline{y}$, welcher also das Monom $x\overline{y}$ überdeckt, was auch nicht wirklich verwundert. Dies ist jedoch nicht die einzige Subfunktion, die $x\overline{y}$ überdeckt. Denn es gilt ja auch $z \vee \overline{z} = 1$. Das Monom $m = x\overline{y}$ wird also auch durch das Polynom $p = x\overline{z} \vee \overline{y}z$ überdeckt. Wir erhalten also die Menge $MU(p^*, m)$, wenn wir:

1. Die Subfunktionen $h_i = (m_i^*)_m$ bilden.
2. Die nicht verkürzbaren Teilsummen aus den Monomen $h_1, \ldots, h_k$ suchen, die die konstante Funktion 1 darstellen.

Der erste Schritt ist effizient durchführbar, während der zweite Schritt im allgemeinen schwierig ist.

Es wird nun ein Verfahren vorgestellt, welches der Baummethode in der Vorgehensweise recht ähnlich ist. Die grundlegende Idee besteht darin, dass es einfacher ist, die minimalen Überdeckungen für die zwei Verlängerungen $m\overline{x}_j$ und mx_j des Monoms m zu berechnen. Dabei wird ein rekursiver Ansatz benutzt, der spätestens dann endet, wenn die minimale Überdeckung eines Minterms berechnet werden soll. Die minimalen Überdeckungen eines Minterms sind nämlich nur einelementige Mengen. Der zentrale Satz dabei ist der folgende:

1. Für $I_0 \in MU(p^*, m\overline{x}_j)$ und $I_1 \in MU(p^*, mx_j)$ existiert eine Menge $I \subseteq I_0 \cup I_1$ mit $I \in MU(p^*, m)$.
2. Für $I \in MU(p^*, m)$ gibt es Mengen $I_0 \in MU(p^*, m\overline{x}_j)$ und $I_1 \in MU(p^*, mx_j)$ mit $I = I_0 \cup I_1$.

Wir überlegen uns nun zuerst, was diese beiden Sätze bedeuten und beweisen danach die Korrektheit. Dabei fangen wir mit dem zweiten Satz an. Dieser besagt, dass es für jede minimale Überdeckung eines Monoms m mindestens eine minimale Überdeckung für $m\overline{x}_j$ und mindestens eine minimale Überdeckung für mx_j gibt, deren Vereinigung genau die minimale Überdeckung für m ergibt. Wenn wir die Mengen der minimalen Überdeckungen für $m\overline{x}_j$ und mx_j kennen, dann können wir also alle möglichen Vereinigungen der Elemente der beiden Mengen bilden, um die Überdeckungen für m zu bekommen. Dabei gibt es allerdings noch ein Problem. Satz 2 beinhaltet nur die Aussage „gibt es Mengen“. Es könnte also passieren, dass wir eine „unerlaubte“ Vereinigung durchführen und dabei eine Überdeckung konstruieren, die nicht minimal ist. Satz 1 zeigt uns jedoch, dass dies nicht schlimm ist. Er besagt, das

jede Überdeckung, die sich aus der Vereingung der Überdeckungen für $m\overline{x}_j$ und mx_j ergibt, eine Obermenge einer minimalen Überdeckung von m ist. Damit können wir also einfach alle Vereinigungen bilden und danach die Mengen streichen, für die es echte Teilmengen gibt und auf diese Art und Weise wieder alle minimalen Überdeckungen erhalten. Nun zu den Beweisen:

1. Die Summe aller $m_i^*,\ i \in I_0$ überdeckt $m\overline{x}_j$ und die Summe aller $m_i^*,\ i \in I_1$ überdeckt mx_j. Damit überdeckt die Summe aller $m_i^*,\ i \in I_0 \cup I_1$ also $m\overline{x}_j \vee mx_j = m(\overline{x}_j \vee x_j) = m1 = m$. Es gilt also mindestens $I_0 \cup I_1 \in U(p^*, m)$, da diese Überdeckung ja nicht minimal sein muss. Es gilt somit $I \subseteq I_0 \cup I_1 \in MU(p^*, m)$, egal ob die Überdeckung $I_0 \cup I_1$ minimal ist oder nicht.

2. Die Summe aller $m_i^*,\ i \in I$ überdeckt m. Damit muss sie auch $m\overline{x}_j$ und mx_j überdecken, womit $I \in U(p^*, m\overline{x}_j)$ und $I \in U(p^*, mx_j)$ gilt. Nach Definition gibt es dann auch $I_0 \subseteq I \in MU(p^*, m\overline{x}_j)$ und $I_1 \subseteq I \in MU(p^*, mx_j)$. Satz 1 besagt, dass es dann ein $I' \subseteq I_0 \cup I_1 \in MU(p^*, m)$ gibt. Es gilt also nach Voraussetzung $I \in MU(p^*, m)$ und es gibt ein weiteres $I' \in MU(p^*, m)$. Desweiteren gilt $I' \subseteq I_0 \cup I_1 \subseteq I$, d.h. $I' \subseteq I$. I' kann jedoch keine echte Teilmenge von I sein, da I ansonsten von vornherein keine minimale Überdeckung gewesen wäre. Somit gilt $I' = I_0 \cup I_2 = I$. □

Der Algorithmus funktioniert also grob gesehen so, dass so lange die minimalen Überdeckungen von Verlängerungen berechnet werden, bis ein Abbruchkriterium greift, für das die Menge der minimale Überdeckungen leicht berechnet werden kann. Aus den Ergebnismengen werden dann jeweils die Vereinigungen gebildet und Mengen, für die Teilmengen erzeugt wurden, werden gestrichen. Die entstehende Menge wird dann wiederum als Ergebnis zurückgeliefert.

Frage 2.48: Gebe ein Abbruchkriterium für den Algorithmus zur Berechnung der minimalen Überdeckungen an!

Für das Abbruchkriterium wird das folgende Lemma benutzen. Es sei $p^* = m_1^* \vee \cdots \vee m_k^*$ das Polynom aller Primimplikanten von f und m ein Implikant von f. Wir definieren die Mengen P_1 und P_2 wie folgt:

1. $P_1 = \{i \mid mm_i = 0\}$
2. $P_2 = \{i \mid m \text{ ist eine Verlängerung von } m_i\}$

Falls nun gilt, dass $P_1 \cup P_2 = \{1, \ldots, k\}$ ist, dann gilt: $MU(p^*, m) = \{\{i\} \mid i \in P_2\}$. Wir überlegen uns nun, was die beiden Mengen beschreiben.

Die Menge P_1 beschreibt alle Indizes von Primimplikanten, die auf keinen Fall in der Menge der Überdeckungen vorkommen können. Denn wenn $mm_i = 0$ ist, dann enthalten m und m_i widersprüchliche Literale. Damit ist jedoch $m^{-1}(1) \cap m_i^{-1}(1) = \emptyset$, da eine Eingabe, die m erfüllt, nicht gleichzeitig m_i erfüllen kann und umgekehrt. Damit kann $U(p^*, m)$ nach Definition keine Menge enthalten, die i enthält.

Die Menge P_2 beschreibt alle Indizes von Primimplikanten, die für sich allein in der Menge der Überdeckungen und damit auch in der Menge der minimalen Überdeckungen vorkommen. Wenn m eine Verlängerung von m_i ist, liefert m_i für alle Eingaben, für die m eine 1 liefert, ebenfalls eine 1. Also überdeckt der Primimplikant m_i das Monom m, womit $\{i\}$ in $U(p^*, m)$ vorkommt. Da es zu $\{i\}$ keine echte Teilmenge außer $\emptyset$ gibt, muss $\{i\}$ also auch in $MU(p^*, m)$ vorkommen.

Die Bedingung $P_1 \cup P_2 = \{1, \ldots, k\}$ stellt nun klar, dass für jeden Primimplikanten genau eine dieser beiden Bedingungen zutrifft. Damit ist dann für alle Primimplikanten klar, in welcher Menge sie sich befinden müssen und das Ergebnis kann direkt zurückgeliefert werden.

Frage 2.49: Beschreibe den Algorithmus zur Berechnung der minimalen Überdeckung für ein Monom!

Zur Berechnung von $MU(p^*, m)$ ergibt sich mit den bisherigen Erkenntnissen Algorithmus 5. Dieser benutzt das zuvor beschriebene Abbruchkriterium.

Algorithmus 5 Algorithmus zur Berechnung von $MU(p^*, m)$

Eingabe: Das Polynom $p^* = m_1^* \vee \cdots \vee m_k^*$ der Primimplikanten von f und $m \in I(f)$.
Ausgabe: Die Menge $MU(p^*, m)$

1. Berechne P_1 und P_2. Wenn $P_1 \cup P_2 = \{1, \ldots, k\}$, dann ist $MU(p^*, m) = \{\{i\} \mid i \in P_2\}$. Return.
2. Wähle eine Variable x_j, die nicht in m vorkommt, und berechne rekursiv $MU(p^*, m\overline{x}_j)$ und $MU(p^*, mx_j)$.
3. Bilde alle Mengen $I = I_0 \cup I_1$ mit $I_0 \in MU(p^*, m\overline{x}_j)$ und $I_1 \in MU(p^*, mx_j)$. Streiche aus der Liste der Mengen alle Mengen, für die eine echte Teilmenge in der Liste exisitert. Die verbleibenden Mengen bilden $MU(p^*, m)$.

2.5 Unvollständig spezifizierte Funktionen

Frage 2.50: Was ist eine partiell definierte Boolesche Funktion? Gebe ein Beispiel!

Eine partiell definierte Boolesche Funktion $f \in B_n^*$ ist eine Funktion $f: \{0,1\}^n \to \{0, 1, *\}$. Der Wert „$*$“ heißt „don't care“-Wert, d.h. es ist egal, welchen Wert die Funktion an dieser Stelle annimmt.

Eine solche Funktion kann z.B. beim Entwurf einer Ampelschaltung vorkommen. Tabelle 4 zeigt, wie eine solche Schaltung realisiert werden kann. Es wird je ein Bit benutzt, um zu kodieren, welches Licht der Ampel in einem der vier möglichen Zustände leuchtet. Da wir drei Bits benutzen, um die Zustände zu charakterisieren, könnten wir eigentlich acht verschiedene Zustände kodieren. Jeder Zustandsübergang wechselt in einen der vier möglichen Zustände. Aus diesem Grund kann es uns also egal sein, was die Schaltung in den anderen vier Zuständen macht, da diese eh nie erreicht werden. Da es unser Ziel ist, das billigste Minimalpolynom zu erhalten, sollten wir die „don't care“-Werte also so wählen, dass wir ein solches erhalten. Es wird sich zeigen, dass dies sehr simpel ist.

Frage 2.51: Wann stellt ein Polynom eine unvollständig definierte Funktion dar?

Ein Polynom p stellt eine Funktion $f \in B_n^*$ dar, wenn für alle $f(a) \neq *$ gilt $p(a) = f(a)$. Das Polynom muss also für die Eingaben, die nicht auf ein „don't care“ abbilden, den selben Wert liefern wie die Funktion.

Frage 2.52: Was ist die Erweiterung einer unvollständig definierten Funktion? Welche speziellen Erweiterungen kennst du?

Rot	Gelb	Grün	Rot	Gelb	Grün
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
0	0	0	*	*	*
0	1	1	*	*	*
1	0	1	*	*	*
1	1	1	*	*	*

Tabelle 4: Beispiel einer Ampelschaltung (links: Zustand, rechts: Nachfolgezustand)

Wenn wir eine Funktion $f \in B_n^*$ mit Hilfe eines Polynoms p darstellen, dann muss das Polynom für Eingaben a mit $f(a) = *$ einen Wert von 0 oder 1 liefern. Das Polynom stellt also eine „normale“ Funktion $g \in B_n$ dar. Diese Funktion $g \in B_n$ heißt dann Erweiterung von $f \in B_n^*$.

Zwei besondere Erweiterungen sind die maximale und die minimale Erweiterung einer Funktion $f \in B_n$. Die maximale Erweiterung f^1 ist die Erweiterung bei der alle „don't care“-Werte auf 1 gesetzt werden. Bei der minimalen Erweiterung f^0 werden alle Werte auf 0 gesetzt.

Frage 2.53: Was ist das Minimalpolynom für eine unvollständig definierte Funktion?

Das Minimalpolynom für eine Funktion $f \in B_n^*$ ist das billigste Polynom unter allen Polynomen, die f darstellen.

Frage 2.54: Wie werden Minimalpolynome für partiell definerte Funktionen berechnet?

Bevor wir auf das konkrete Vorgehen eingehen, zeigen wir zuerst, dass Minimalpolynome für partiell definierte Funktionen nur Primimplikanten der maximalen Erweiterung f^1 enthalten. Es sei ein Minimalpolynom $p = m_1 \vee \cdots \vee m_k$ für eine partiell definierte Funktion $f \in B_n^*$ gegeben. Dann ist p das Minimalpolynom einer Erweiterung f^e von f und enthält, da es ein Minimalpolynom ist, auch nur Primimplikanten von f^e. Es gilt also $m_i(a) = 1 \Rightarrow f^e(a) = 1$. Nach Definition gilt jedoch auch $f^e \leq f^1$, d.h. wenn die Erweiterung einer Funktion auf 1 abbildet, dann bildet die maximale Erweiterung „erst recht“ auf 1 ab. Also muss auch $f^1(a) = 1$ gelten, womit insgesamt $m_i(a) = 1 \Rightarrow f^1(a) = 1$ gefolgert werden kann. Das Monom m_i ist somit ein Implikant von f^1. Dann gibt es jedoch Verkürzungen m_i' von m_i, welche Primimplikanten für f^1 sind. Wir zeigen als nächstes, dass wir die Funktion f auch mit all diesen Verkürzungen darstellen können, dass also das Polynom $p' = m_1' \vee \cdots \vee m_k'$ ebenfalls die Funktion f beschreibt. Dazu machen wir eine Fallunterscheidung:

- $f(a) = 1$: Wenn die Funktion f für eine Eingabe a eine 1 liefert, muss auch ein Primimplikant von p eine 1 liefern. Also gilt $m_i(a) = 1$. Dann liefert die Verkürzung m_i' jedoch erst recht eine 1 und somit auch p'.
- $f(a) = 0$: Wenn die Funktion f für eine Eingabe a eine 0 liefert, dann muss auch f^1 eine 0 liefern, da f^1 ja eine maximale Erweiterung von f ist und wir daher kein „don't care“ durch eine Null ersetzen. Die Nullen kommen also aus der „Originalfunktion“ und müssen daher auch von f^1 geliefert werden. Da die m_i' Primimplikanten von f^1 sind, muss also auch für $1 \leq i \leq k$ $m_i'(a) = 0$ gelten. Damit liefert das ganze Polynom p' eine 0, d.h. es gilt $p'(a) = 0$.

Wir haben also gezeigt, dass das Polynom p' ebenfalls die Funktion f darstellt. Wären die m'_i nun echte Verkürzungen der m_i, d.h. $m'_i \neq m_i$, so wäre das resultierende Polynom p' billiger als p, womit p kein Minimalpolynom sein könnte. Also muss $m'_i = m_i$ gelten, woraus folgt, dass Minimalpolynome nur Primimplikanten von f^1 enthalten. □

Um ein Minimalpolynom für eine unvollständig definierte Funktion $f \in B_n^*$ zu berechnen, reicht es also aus, die Primimplikanten der Erweiterung f^1 zu berechnen. Zu Beginn setzen wir somit alle „don't cares" auf 1 und berechnen die Primimplikanten der resultierenden Funktion. Damit erhalten wir die Menge aller Primimplikanten, die ein Minimalpolynom für f benutzen kann. Es bleibt somit nur noch die Aufgabe der Überdeckung zu lösen. Wir könnten dabei einfach plump alle Eingaben überdecken, für die f^1 auf 1 abbildet. Dies wäre jedoch nicht sehr clever, da wir unnötig viele Eingaben überdecken würde. Uns interessieren ja in Wirklichkeit nur die Eingaben, für die f kein $*$ liefert. Diese sind jedoch auch in f^0 enthalten, also quasi der Überdeckung, die so wenig Einsen wie möglich liefert. Das Gesamtvorgehen sieht also so aus, dass wir zuerst die Primimplikanten der Erweiterung f^1 berechnen und mit diesen auf möglichst billige Weise alle Monome eines f^0 beschreibenden Polynoms überdecken.

2.6 Funktionen mit mehreren Outputs

Frage 2.55: Wann stellt ein Polynom eine Funktion mit mehreren Output dar? Wie sind die Kosten eines solchen Polynoms definiert?

Ein Polynom $p = (p_1, \ldots, p_k)$ mit mehreren Outputs stellt eine Funktion $f = (f_1, \ldots, f_k) \in B_{n,k}$ dar, wenn p_i ein Polynom für f_i ist. Die Kosten eines solchen Polyoms ergeben sich als die Summe der Kosten aller verschiedenen Monome in $p = (p_1, \ldots, p_k)$.

Frage 2.56: Was ist ein Minimalpolynom für eine Funktion mit mehreren Outputs? Was ist dabei zu beachten?

Ein Minimalpolynom für eine Funktion $f \in B_{n,k}$ mit mehreren Outputs ist ein Polynom. welches unter allen Polynomen die f darstellen die geringsten Kosten hat. Dabei ist zu beachten, dass das Minimalpoylnom dabei oft *nicht* aus den minimalen Polynomen der einzelnen Funktionen besteht. Dies bedeutet unter anderem, dass das Minimalpolynom $p^* = (p_1^*, \ldots, p_k^*)$ für f in p_i^* Implikanten von f_i enthalten kann, die keine Primimplikanten sind.

Wir geben dazu ein Beispiel. Es gelte $f \in B_{3,2}$ mit $f = (f_1, f_2)$, wobei $f_1(x, y, z) = \overline{y}z \vee xz = p_1$ und $f_2(x, y, z) = \overline{x}y \vee yz = p_2$. Es kann schnell gezeigt werden, dass p_1 und p_2 Minimalpolynome für f_1 bzw. f_2 sind, da auf beiden kein Konsensus mehr möglich ist. Damit sind beide Pinup-Polynome, die sich nicht reduzieren lassen und von denen jeder Primimplikant nur sich selbst überdeckt. Da keiner der Primimplikanten doppelt vorkommt und jeder Kosten von 2 verursacht, betragen die Gesamtkosten, wenn wir f als $p = (p_1, p_2)$ darstellen, also 8. Allerdings ist es möglich, geringere Kosten zu erhalten. Wir betrachten dazu $p_1^* = \overline{y}z \vee xyz$ und $p_2^* = \overline{x}y \vee xyz$. Diese beiden Polynom stellen ebenfalls die Funktionen f_1 bzw. f_2 dar, jedoch sind die Kosten des Polynom $p = (p_1^*, p_2^*)$ günstiger, da das Monom xyz nur einmal gezählt wird. Es ergeben sich damit Kosten von 7. Es ist jedoch leicht einzusehen, dass das Monom xyz kein Primimplikant von f_1 oder f_2 ist. Doch wie lassen sich solche Monome charakterisieren?

Frage 2.57: Was ist ein multipler Primimplikant?

Ein Monom m heißt multipler Primimplikant für eine Funktion $f = (f_1, \ldots, f_k)$, wenn es eine nichtleere Menge $I \subseteq \{1, \ldots, k\}$ gibt, so dass m Primimplikant der Konjunktion aller $f_i,\ i \in I$ ist. Es gilt also $m \in PI(\bigwedge_{i \in I} f_i)$.

Frage 2.58: In welchem Zusammenhang stehen multiple Primimplikanten zu Funktionen mit mehreren Outputs?

Es kann gezeigt werden, dass Minimalpolynome für Funktionen f mit mehreren Outputs nur multiple Primimplikanten von f enthalten. Dazu betrachten wir ein Minimalpolynom $p = (p_1 \ldots, p_k)$, welches das Monom m mindestens einmal enthalte. Um im Folgenden eleganter argumentieren zu können, gehen wir o.B.d.A. davon aus, dass die Polynome $p_1, \ldots, p_i$ das Monom m enthalten. Dies bedeutet dann jedoch, dass m Implikant von $f_1, \ldots, f_i$ ist, d.h. für jedes $1 \leq j \leq i$ gilt $m(a) = 1 \quad \Rightarrow \quad f_j(a) = 1$. Daraus folgt wiederum, dass auch $(f_1 \wedge \cdots \wedge f_i)(a) = 1$ ist. Das Monom m ist somit auch Implikant von $f_1 \wedge \cdots \wedge f_i$. Wir zeigen nun über einen Widerspruch, dass m auch Primimplikant von $f_1 \wedge \cdots \wedge f_i$ sein muss. Angenommen, m wäre kein Primimplikant. Wenn m kein Primimplikant wäre, gäb es eine echte Verkürzung m', die Primimplikant von $f_1 \wedge \cdots \wedge f_i$ wäre. Als nächstes zeigen wir nun, dass wir dann in den Polynomen $p_1, \ldots, p_i$ jeweils m durch m' ersetzen könnten und immer noch f darstellen würden. Die Polynome, die auf diese Art und Weise entstehen, nennen wir $p'_1, \ldots, p'_i$. Auf diese Art und Weise hätte wir ein billigeres Polynom gefunden, was im Widerspruch dazu steht, dass p ein Minimalpolynom sei. Es bleibt also nur noch zu zeigen, dass für $1 \leq j \leq i$ das Polynom p'_j ebenfalls die Funktion f_j beschreibt. Wir machen eine Fallunterscheidung in Abhängigkeit davon, welchen Wert p'_j annimmt:

- $p'_j(a) = 0$: Wenn p'_j für eine Eingabe den Wert 0 liefert, muss jedes seiner Monome den Wert 0 liefern, womit $m'(a) = 0$ gelten muss. Dann liefert jedoch auch jede Verlängerung von m' für die Eingabe a den Wert 0, also gilt auch $m(a) = 0$. Die Polynome p'_j und p_j unterscheiden sich nur in einem Monom. Wenn also p'_j den Wert 0 liefert und wir m' durch m ersetzen (womit wir p_j erhalten), so liefert nach obigen Beobachtungen auch p_j für die Eingabe a den Wert 0 und es gilt $f(a) = 0$.
- $p'_j(a) = 1$: Das Monom m' ist Primimplikant von $f_1 \wedge \cdots \wedge f_i$. Damit muss es jedoch auch Implikant von f_j sein. Das Polynom p_j enthält natürlich auch nur Implikanten von f_j. Dies bleibt also auch so, wenn wir m durch m' ersetzen, wodurch p'_j entsteht. Damit impliziert $p'_j(a) = 1$ auch $f_j(a) = 1$. □

Frage 2.59: Wie kann die Menge der multiplen Primimplikanten für eine Funktion mit mehreren Outputs berechnet werden?

Es sei eine Funktion $f \in B_{n,k}$ gegeben, für die wir die Menge der multiplen Primimplikanten $MPI(f)$ berechnen möchten. Dabei können wir den folgenden Satz benutzen, der dem zentralen Satz bei der Berechnung der Menge $MU(p^*, m)$ sehr ähnlich ist. Es sei $f = f_1 \wedge \cdots \wedge f_k$ mit $f_i \in B_n$ für $1 \leq i \leq k$. Dann gilt:

1. Für $m_i \in I(f_i)$ ist $m = m_1 \wedge \cdots \wedge m_k \in I(f)$.
2. Für $m \in PI(f)$ gibt es $m_i \in PI(f_i)$, so dass $m = m_1 \wedge \cdots \wedge m_k$ ist.

Wir überlegen uns zuerst, was die beiden Sätze aussagen und beweisen sie anschließend. Dazu fangen wir mit Satz 2 an. Dieser besagt, dass sich jeder Primimplikant von f als eine Konjunktion von einzelnen Primimplikanten der Funktionen f_i, die die Funktion f bilden, beschreiben

lässt. Satz 1 zeigt uns, dass jede Summe von Implikanten für die einzelnen Funktionen f_i einen Implikanten für die Funktion f bildet. Nun zu den Beweisen:

1. Wir nehmen also an, dass $m(a) = 1$ ist und zeigen, dass dann auch $f(a) = 1$ ist. Wenn $m(a) = 1$ ist, muss für alle $1 \leq i \leq k$ auch $m_i(a) = 1$ gelten. Da m_i ein Implikant von f_i ist, ist dann auch $f_i(a) = 1$ und somit auch $f(a) = (f_1 \wedge \cdots \wedge f_k)(a) = 1$.

2. Wir nehmen also an, dass m ein Primimplikant von f ist. Wenn dann $m(a) = 1$ gilt, so gilt auch $f(a) = 1$. Damit muss für $1 \leq i \leq k$ auch $f_i(a) = 1$ gelten. Dies bedeutet jedoch, dass m auch ein Implikant für die einzelnen f_i ist, also $m \in I(f_i)$. Dann gibt es jedoch auch Verkürzungen m_i von m mit $m_i \in PI(f_i)$. Diese Verkürzungen sind also Primimplikanten. Der Satz besagt nun, dass die Summe dieser Primimplikanten m darstellt. Wenn wir die Summe $m = m_1 \wedge \cdots \wedge m_k$ bilden, dann ist diese eine Verkürzung von m, da jedes einzelne Glied dieser Summe eine Verkürzung von m ist. Nach Satz 1 ergibt diese Summe auch auch einen Implikanten von f. Dieser kann jedoch keine echte Verkürzung von m sein, da m nach Voraussetzung Primimplikant ist. □

Wir können also die multiplen Primimplikanten der Funktion $f = f_1 \wedge \cdots \wedge f_k$ berechnen, indem wir die Primimplikantenmengen der einzelnen Funktionen f_i berechnen und die Primimplikanten in diesen Mengen auf alle möglichen Arten und Weisen miteinander konjugieren. Monome, für die Verkürzungen entstehen, werden dabei gestrichen. Satz 1 sagt uns, dass wir bei diesem Vorgehen nur Implikanten erzeugen, also nichts falsch machen. Satz 2 versichert uns, dass alle Primimplikanten von f erzeugt werden, so dass Verlängerungen, die unter Umständen erzeugt werden, auf jeden Fall gestrichen werden.

In der Praxis sollten wir die Menge der Primimplikanten für $f = f_1 \wedge \cdots \wedge f_k$ wie folgt berechnen. Wir schreiben die Funktion für $i = \lfloor \frac{k}{2} \rfloor$ als:

$$f = f_A \wedge f_B \qquad \text{mit} \qquad f_A = f_1 \wedge \cdots \wedge f_i \quad \text{und} \quad f_B = f_{i+1} \wedge \cdots \wedge f_k$$

Wenn wir die Primimplikantenmengen von f_A und f_B auf dieselbe Art und Weise rekursiv berechnet haben, brauchen wir nur noch die Primimplikanten aus $PI(f_A)$ und $PI(f_B)$ entsprechend aller Möglichkeiten miteinander zu kombinieren. Bei diesem Vorgehen entsteht also ein Baum, bei dem sich mit jeder Aufteilung die Menge der beteiligten Funktionen ungefähr halbiert. An den Blättern dieses Baumes wird jeweils die Primimplikantenmenge $PI(f_i)$ einer jeden einzelnen Funktion berechnet und diese dann entsprechend des Aufrufbaums wieder „vermischt“.

2.7 Monotone und Symmetrische Funktionen

In diesem Kapitel befassen wir uns mit zwei speziellen Klassen von Funktionen. Die erste Klasse ist die Klasse der monotonen Funktionen und die zweite Klasse die Klasse der symmetrischen Funktionen, zu denen z.B. auch MD_n zählt. Die Berechnung eines Minimalpolynoms für MD_n ist mit den bisher vorgestellten Verfahren sehr aufwändig, selbst wenn n nicht sehr groß ist. Eines der Ergebnisse diese Kapitels wird sein, dass die Berechnung der Minimalpolynome für beide Funktionsklassen effizient möglich ist, womit eine nähere Beschäftigung genügend motiviert sein dürfte.

2.7.1 Monotone Funktionen

Frage 2.60: Wann heißt eine Boolesche Funktion monoton?
Eine Boolesche Funktion $f \in B_n$ heißt monoton, wenn $a \leq b$ impliziert, dass auch $f(a) \leq f(b)$ gilt. Wir schreiben dann auch $f \in M_n$. Dabei gilt $0 \leq 1$ und $a \leq b$, wenn $a_i \leq b_i$ für alle i ist. So gilt z.B. $0100 \leq 0110$.

Frage 2.61: Wie sehen die Primimplikanten für monotone Funktionen aus?
Jeder Primimplikant einer monotonen Funktion $f \in M_n$ hat nur positive Literale. Dies ist schnell gezeigt. Wir zeigen dabei, dass für jeden Implikanten $m = m'\overline{x}_j$ auch das Monom m' ein Implikant von f ist. Wir beginnen mit der Annahme, dass $m'(a) = 1$ ist. Nun bauen wir eine neue Eingabe b mit $b_j = 0$ und $b_i = a_i$ für alle anderen Eingabebits. Dann gilt $m'\overline{x}_j(a) = 1$. Wir kippen in b nun das Eingabebit an der Stelle j, so dass nun $b_j = 1$ gilt. Diese neue Eingabe ist größer als die alte Eingabe b. Da f monoton ist, muss die Funktion dann jedoch auch für diese neue Eingabe den Wert 1 liefern. Insgesamt haben wir also $m'(a) = 1 \Rightarrow f(a) = 1$ gezeigt, womit m' ein kürzerer Implikant ist. Daraus folgt, dass kein Primimplikant für eine monotone Funktion negative Literale enthalten kann.

Frage 2.62: Wie sehen Minimalpolynome für monotone Funktionen aus?
Es kann gezeigt werden, dass die Minimalpolynome für monotone Funktionen eindeutig sind und alle Primimplikanten von f enthalten. Dies ergibt sich im Prinzip aus der Eigenschaft, dass nur positive Literale in den Primimplikanten vorkommen. Es sei $m \in PI(f)$. Wir zeigen nun, dass jeder Primimplikant m ein Kernimplikant ist. Damit ist m der einzige Primimplikant, der eine Eingabe a abdeckt und muss daher im Minimalpolynom vorkommen. Wir wissen schon, dass nur positive Literale vorkommen. Um leichter argumentieren zu können, nummerieren wir die Variablen von m auch noch um. Es sei also $m = x_1 \wedge \cdots \wedge x_i$. Dann deckt m die Eingabe a ab, bei der $a_j = 1$ für alle $j \leq i$ gilt und $a_j = 0$ für alle $j > i$. Wir zeigen nun, dass kein anderer Primimplikant m' die Eingabe a abdecken kann, womit m dann ein Kernimplikant ist. Nehmen wir also an, dass es einen Primimplikanten $m' \neq m$ gibt, der a abdeckt, d.h. es gilt $m'(a) = 1$. Da alle Primimplikanten nur positive Literale enthalten, kann m' keine Variablen x_j mit $j > i$ enthalten. Es gilt jedoch auch $m' \neq m$, also müsste m' eine echte Verkürzung von m sein. Dies wäre ein Widerspruch zur Primimplikanteneigenschaft von m.

Frage 2.63: Wie kann ein Minimalpolynom für monotone Funktionen berechnet werden?
Wir wissen schon, dass es ein eindeutiges Minimalpolynom für monotone Funktionen gibt, welches alle Primimplikanten enthält. Also reicht es aus, die Primimplikanten zu berechnen, da diese ja alle im Minimalpolynom benutzt werden. Wir brauchen also kein Überdeckungsproblem mehr zu lösen. Gegeben sei also ein Polynom p, welches eine monotone Funktion beschreibt. In diesem kommen nur Implikanten vor, welche unter Umständen noch negative Literale enthalten. Wir haben schon gesehen, dass wir wieder einen Implikanten erhalten, wenn wir in einem Implikanten die negativen Literale streichen. Wir streichen also als erstes die negativen Literale in den Monomen und wissen, dass wir dabei nichts falsch machen. Dabei entsteht das Polynom p', für das wir nun als nächstes zeigen werden, dass es ebenfalls f darstellt, da $p' \geq p$ gilt. Was bedeutet das Vorkommen eines negativen Literals $\overline{x}_i$ in einem Monom m des Polynoms? Dies bedeutet, dass es Eingaben a gibt, die durch das Monom m überdeckt werden und für die die Funktion den Wert 1 liefert, wenn $a_i = 0$ gilt. Aufgrund der

Monotonieeigenschaft liefert die Funktion dann jedoch auch den Wert 1, wenn wir in denselben Eingaben $a_i = 1$ setzen. Also brauchen wir uns im Monom m im Prinzip gar nicht um die Variable x_i kümmern und können somit alle negativen Literale in allen Monomen streichen. Nun streichen wir noch alle Verlängerungen, die in p' vorkommen und erhalten so p''. Das Polynom p'' enthält nun nur noch positive Literale und nach dem Korollar für den iterierten Konsensus folgt daraus, dass p'' alle Primimplikanten von f enthält, da kein Konsensus mehr möglich ist.

2.7.2 Symmetrische Funktionen

Frage 2.64: Wie sind symmetrische Funktionen definiert?

Eine Boolesche Funktion $f \in B_n$ heißt symmetrisch, wenn für einen Wertevektor $v(f) = (v_0, \ldots, v_n) \in \{0,1\}^{n+1}$ gilt:

$$f(a) = v_{||a||} \text{ wobei } ||a|| = a_1 + \cdots + a_n$$

Bei einer symmetrischen Funktion hängt der Funktionswert also nur von der Anzahl der Einsen ab, die in der Eingabe a vorkommen. Gezählt werden die Einsen mit der Funktion $||a|| = a_1 + \cdots + a_n$. Das Ergebnis dieser Berechnung wird dann als Index in den Wertevektor $v(f)$ genommen, welcher angibt, ob die Funktion bei einer bestimmten Anzahl von Einsen den Wert 0 oder 1 liefert. Dieser hat die Länge $n + 1$, da eine Eingabe zwischen 0 und n Einsen enthalten kann. Die Menge der symmetrischen Funktionen wird auch mit S_n bezeichnet.

Frage 2.65: Wie ist die Intervallfunktion definiert? Wozu wird sie benutzt?

Die Intervallfunktion $I_{k,l}$ mit $0 \leq k \leq l \leq n$ ist eine symmetrische Funktion, für deren Wertevektor gilt, dass $v_i = 1$ für $k \leq i \leq l$ ist. Wir beschäftigen uns mit Intervallfunktionen, da diese eine Basis für die symmetrischen Funktionen darstellen. Dies bedeutet, dass sich jede symmetrische Funktion als eine Disjunktion von Intervallfunktionen darstellen lässt. So lässt sich zum Beispiel die Funktion $f \in S_{10}$ mit $v(f) = (1,0,0,1,1,1,0,1,1,0,0)$ auch darstellen als $f = I_{0,0} \vee I_{3,5} \vee I_{7,8}$.

Frage 2.66: Wie sehen die Primimplikanten einer Intervallfunktion aus?

Im Prinzip haben wir eine ähnliche Überlegung schon bei der Funktion $MD_n \in S_n$ angestellt. Die Intervallfunktion $I_{k,l}$ wird genau dann 1, wenn sie mindestens k und höchstens l Einsen enthält. Wir überlegen uns nun wieder zuerst, wie Implikanten für die Intervallfunktion aussehen. Ein Implikant benötigt mindestens k positive Literale, um sicher zu stellen, dass eine Eingabe auch mindestens k Einsen enthält. Die Eingabe darf jedoch auch nicht zuviele Einsen enthalten. Da sie höchstens l Einsen enthalten darf, brauchen wir also mindestens $n - l$ negative Literale, um sicher zu stellen, dass mindestens $n - l$ Eingabebits 0 sind. Ist dies der Fall können ja nur noch höchstens $n - (n - l) = l$ Eingabebits 1 sein. Wie sehen nun also Primimplikanten aus? Hier gilt dasselbe Argument wie bei der Funktion MD_n. Enthält ein Implikant mehr als k positive Literale, so können wir eins davon streichen und machen nichts falsch. Dasselbe gilt für die $n - l$ negativen Literale. Damit enthalten die Primimplikanten also genau k positive und $n - l$ negative Literale.

Frage 2.67: Wie sehen die Minimalpolynome für symmetrische Funktionen aus?

Das Minimalpolynom einer symmetrischen Funktion $f \in S_n$ ist die Summe von Minimalpolynomen der Intervallfunktionen in der Intervalldarstellung von f. Wir wollen also quasi

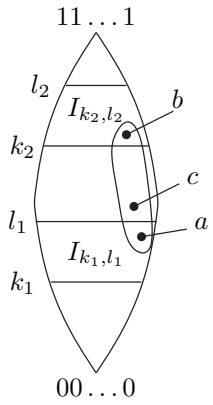


Abbildung 5: Die grobe Beweisidee, dass ein Primimplikant einer symmetrischen Funktion nur einige Eingaben *einer* Intervallfunktion überdecken kann

zeigen, dass in einem Minimalpolynom für f keine Primimplikanten vorkommen können, die aus den Minimalpolynomen mehrerer Intervallfunktionen wiederverwendet werden, so dass Kosten gespart werden könnten. Um dies zu beweisen, reicht es schon zu zeigen, dass es keinen Implikanten m für f geben kann, der Eingaben aus zwei Intervallfunktionen überdecken kann. Für Primimplikanten gilt diese Aussage dann natürlich ebenfalls. Wir können dann also die Minimalpolynome für die Intervallfunktionen berechnen und die Ergebnisse disjunktiv verknüpfen.

Wir betrachten im Folgenden also zwei Intervallfunktionen I_{k_1,l_1} und I_{k_2,l_2}, für die $l_1 < k_2 - 1$ gelten soll. Insbesondere gilt also $v_{l_1+1} = 0$. Dies bedeutet, dass es zwischen den beiden Intervallen eine echte Lücke gibt, bzw., dass die beiden Intervalle auch wirklich maximale 1-Intervalle sind und nicht verschmolzen werden können. Dieser Umstand wird im Laufe des Beweises noch ausgenutzt werden. Wir nehmen nun also an, dass ein Implikant m von f Eingaben von I_{k_1,l_1} und von I_{k_2,l_2} überdeckt. Diese Eingaben seien $k_1 \leq ||a|| \leq l_1$ und $k_2 \leq ||b|| \leq l_2$. Abbildung 5 zeigt den groben Aufbau des Beweises. Als nächstes überlegen wir uns nun, was für das Aussehen des Implikanten m folgt, wenn er sowohl a als auch b überdeckt. Da der Implikant m die Eingabe a, die $||a||$ Einsen enthält, überdecken soll, kann m höchstens $||a||$ positive Literale enthalten. Ansonsten könnte m_a nicht die konstante Funktion 1 sein, da mindestens ein positives Literal übrig bliebe. Dies ist jedoch die Voraussetzung dafür, dass die Eingabe a von m überdeckt wird. Auf dieselbe Art und Weise können wir nun bezüglich der negativen Literale in m argumentieren, wobei wir jedoch ausnutzen, dass m auch b überdecken soll. Da m auch b überdecken soll, kann m höchstens $n - ||b||$ negative Literale enthalten. Ansonsten würde m_b negative Literale enthalten und b wäre dementsprechend nicht überdeckt. Wieviele Literale enthält m also insgesamt? Der Implikant m enthält höchstens $||a||$ positive und höchstens $n - ||b||$ negative Literale und damit höchstens $||a|| + n - ||b|| = n - ||b|| + ||a|| = n - (||b|| - ||a||)$ Literale. Wir schätzen $||b|| - ||a||$ weiter ab. Da die Intervalle disjunkt sind, muss die Differenz zwischen $||b||$ und $||a||$ mindestens 2 betragen. Somit enthält m höchstens die folgende Anzahl an Literalen:

$$n - (||b|| - ||a||) \leq n - 2$$

Damit gibt es also mindestens zwei Eingabebits, die nicht in m vorkommen und die dementsprechend beliebig abgeändert werden können, wobei m immer noch eine 1 berechnet. Es kann gezeigt werden, dass es möglich ist, soviele Eingabebits abzuändern, dass eine neue Eingabe

n	3	6	9	12	15
Zahl der PI	6	90	1680	34650	756756
Zahl der PI im Min.Poly.	3	15	84	485	3003

Tabelle 5: „Zahl der PI vs. Zahl der PI in Minimalpolynomen“ für symmetrische Funktionen

c mit $||c|| = l_1 + 1$ Einsen entsteht, auf der m immer noch eine 1 berechnet. Auf Eingaben mit $l_1 + 1$ Eingabebits darf m jedoch keine 1 berechnen, da m ein Implikant für f sein soll und f für Eingaben dieser Länge eine 0 berechnet. Wir haben ja schon beobachtet, dass nach Voraussetzung $v_{l_1+1} = 0$ gelten muss. Es ergibt sich somit ein Widerspruch und es folgt, dass ein Implikant m stets nur Eingaben für eine Intervallfunktion überdecken kann.

Frage 2.68: Wieviele Primimplikanten hat die Intervallfunktion? Wieviele Primimplikanten enthält das Minimalpolynom einer Intervallfunktion?

Wir überlegen uns zuerst, wieviele Primimplikanten eine Intervallfunktion $I_{k,l}$ besitzt. Alle Primimplikanten enthalten genau k positive und $n - l$ negative Literale und kosten dementsprechend gleich viel. Wieviele Möglichkeiten haben wir nun, um diese aus n Variablen zu wählen? Wir haben schon eine ähnliche Rechnung für MD_n durchgeführt. Zuerst wählen wir aus allen n Bits die k positiven Literale aus und danach aus den verbleibenden $n - k$ Bits die $n - l$ negativen Literale. Damit ergibt sich die folgende Anzahl an Möglichkeiten:

$$\binom{n}{k} \cdot \binom{n-k}{n-l} = \frac{n!}{k! \cdot (n-k)!} \cdot \frac{(n-k)!}{(n-l)! \cdot ((n-k)-(n-l))!} = \frac{n!}{k! \cdot (n-l)! \cdot (l-k)!}$$

Damit wissen wir nun, wieviele Primimplikanten es gibt. Nun überlegen wir uns, wieviele Primimplikanten ein Minimalpolynom mindestens enthalten muss. Jeder Primimplikant für $I_{k,l}$ hat k positive und $n-l$ negative Literale. Also überdeckt jeder Primimplikant genau eine der Eingaben, die genau k Einsen enthält. Wieviele solcher Eingaben gibt es? Wir müssen aus n Bits k Bits auswählen, die 1 sein sollen, der Rest ist 0. Also ergibt dies $\binom{n}{k}$ Eingaben. Jeder Primimplikant deckt auch genau eine Eingabe ab, die $n - l$ Nullen hat (diese Eingabe hat also die „obere Schranke“ von l Einsen). Davon gibt es $\binom{n}{n-l} = \binom{n}{n-(n-l)} = \binom{n}{l}$. Damit wissen wir also, dass wir mindestens die folgende Anzahl an Primimplikanten benötigen:

$$\max\left\{\binom{n}{k}, \binom{n}{l}\right\}$$

Es kann gezeigt werden, dass diese Anzahl an Primimplikanten auch ausreicht, was wir hier jedoch nicht machen. Eines der Ergebnisse dieser Betrachtungen ist, dass für symmetrische Funktionen oft sehr viel mehr Primimplikanten existieren als im Minimalpolynom tatsächlich benötigt werden. Tabelle 5 zeigt dies anhand der Funktion MD_n für verschiedene n.

Frage 2.69: Wie funktioniert der Algorithmus zur Berechnung des Minimalpolynoms für eine Intervallfunktion? Welche Rechenzeit hat er?

Wir betrachten die Intervallfunktion $I_{k,l}$ über n Bits. Es wird also die Intervallfunktion betrachtet, die eine 1 liefert, wenn die Eingabe aus mindestens k Einsen und mindestens $n-l$ Nullen besteht. Der Algorithmus hat als Bedingung, dass $k \geq n-l$ sein muss. Die Bedingung besagt also, dass die Eingabe mehr Einsen als Nullen enthalten muss, damit der Algorithmus funktioniert, so wie er beschrieben ist. Wir überlegen uns nun, warum diese Forderung keine Einschränkung darstellt.

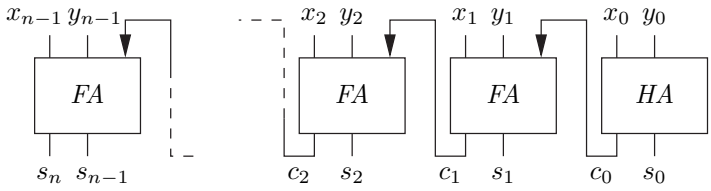
Algorithmus 6 Algorithmus zur Berechnung eine Primimplikanten von $I_{k,l}$ zur Überdeckung von a mit $||a|| = k$

Eingabe: n, k, l mit $k \geq n - l$ und $a \in \{0,1\}^n$ mit $||a|| = k$.
Ausgabe: Primimplikant p_a von $I_{k,l}$, der a überdeckt.
Datenstruktur: Ein Stack, der zu Beginn leer ist.

1. Der zu a gehörende Minterm $m_a = x_1^{a_1} \wedge \cdots \wedge x_n^{a_n}$ wird in kanonischer Reihenfolge als String von Literalen interpretiert.
2. Die x_i mit $a_i = 1$ bilden die k positiven Literale von a.
3. Der String m_a wird genau einmal durchlaufen. Negative Literale werden auf den Stack gelegt (push), während positive Literale bewirken, dass Elemente vom Stack genommen (pop) werden, sofern dieser nicht leer ist.
4. Es werden so lange Pops durchgeführt, bis der Stack leer ist.
5. Die $n - l$ negativen Literale, die zuerst gepoppt werden, bilden die $n - l$ negativen Literale in p_a.

Wenn nicht $k \geq n - l$ gilt, muss also $k < n - l$ gelten. Schreiben wir dies als $n - l > k$, sehen wir schnell, dass wir die Rollen auch vertauschen können. Betrachten wir also die Funktion $I_{n-l,n-k}$. Für diese gilt zumindest die obige Bedingung, denn es gilt $n - l \geq n - (n - k) = k$. Jedoch stellt sie nicht die Intervallfunktion dar, die uns eigentlich interessiert. Die Intervallfunktion $I_{n-l,n-k}$ berechnet eine 1, wenn die Anzahl der Einsen in der Eingabe mindestens $n - l$ und die Anzahl der Nullen mindestens $n - (n - k) = k$ beträgt. Wenn wir nun jedes der Eingabebits negieren, berechnet die Funktion also eine 1, wenn die Anzahl der Einsen mindestens k und die Anzahl der Nullen mindestens $n - l$ beträgt. Dies ist jedoch genau die Funktion, die uns interessiert. Wenn die Bedingung $k \geq n - l$ also nicht zutrifft, dann berechnen wir das Minimalpolynom von $I_{n-l,n-k}$ und tauschen in diesem x_i gegen $\overline{x}_i$ aus und umgekehrt.

Zur Berechnung des Minimalpolynoms von $I_{k,l}$ berechnen wir für jede der $\binom{n}{k}$ Eingaben a mit $||a|| = k$ mit Hilfe von Algorithmus 6 einen Primimplikanten p_a von $I_{k,l}$, der genau die Eingabe a überdeckt. Es kann gezeigt werden, dass die Summe dieser p_a die Intervallfunktion $I_{k,l}$ überdeckt, was hier jedoch nicht gemacht wird. Die k positiven Literale werden alle zum Primimplikanten hinzugefügt, während bei den $n - l$ negativen Literalen die Mischung von positiven und negativen Literalen in der kanonischen Darstellung entscheidet, welches der Literale im Primimplikanten vorkommt. In jedem Fall ist p_a ein Primimplikant von $I_{k,l}$, da es k positive und $n - l$ negative Literale enthält.



Größe: $5(n-1)+2 = 5n-3$
Tiefe: $2(n-1)+1 = 2n-1$

Abbildung 6: Die Schulmethode der Addition

3 Addition, Subtraktion, Multiplikation, Division

3.1 Die Schulmethode für die Addition

Frage 3.1: Wie ist die Addition definiert?
Die Addition ADD ist die Folge Boolescher Funktionen $ADD_n \in B_{2n,n+1}$, die aus den Binärdarstellungen $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$ die Summe $s = (s_n, \ldots, s_0)$ berechnen.

Frage 3.2: Wie funktioniert die Schulmethode für die Addition?
Die Schulmethode der Addition funktioniert bei Binärzahlen genau so, wie man es in der Schule mit Dezimalzahlen gelernt hat. Die Zahlen x_i, y_i und ein Übertrag c_{i-1} werden addiert, wodurch sich das Ergebnis an der Stelle s_i sowie ein neuer Übertrag c_i ergeben. Der Übertrag wird auch Carrybit genannt. Das Verfahren wird in Algorithmus 7 genauer beschrieben und in Abbildung 6 skizziert.

Algorithmus 7 Die Schulmethode der Addition

1. Addiere x_0 und y_0. Dabei entsteht das Summenbit s_0 und ein Übertrag c_0.
2. Für $i = 1, \ldots, n-1$: Addiere x_i, y_i und c_{i-1}. Das Ergebnis ist s_i und c_i.
3. $s_n = c_{n-1}$.

Da an der Berechnung des ersten Summenbit s_0 kein Carrybit beteiligt sein kann, reicht es aus, für diesen Schritt einen Halfadder (HA) zu benutzen. Ein Halfadder berechnet aus lediglich zwei Eingabebits ein Summenbit und ein Carrybit. Dieser kann mit zwei Bausteinen in Tiefe 1 realisiert werden. Das Summenbit s_0 ergibt sich aus x_0 und y_0 nämlich als $s_0 = x_0 \oplus y_0$, denn es wird 1, wenn nur eines der Eingabebits x_0 und y_0 auf 1 gesetzt ist. Wann wird der Übertrag c_0 nun zu 1? Dies ist der Fall, wenn beide Eingabebits x_0 und y_0 auf 1 gesetzt sind. Damit gilt also $c_0 = x_0 \wedge y_0$. Beide Werte sind unabhängig voneinander und können somit parallel mit einem $\oplus$- und einem $\wedge$-Baustein berechnet werden, womit sich eine Tiefe von 1 ergibt.

Nun überlegen wir uns, wie ein Schaltkreis für einen Fulladder aussieht. Dieser bekommt die Bits x, y, c als Eingabe und berechnet aus diesen das Summenbit $\overline{s}$ und das Carrybit $\overline{c}$. Das Summenbit $\overline{s}$ nimmt genau dann den Wert 1 an, wenn die Anzahl der Einsen in der Eingabe ungerade ist. Somit kann es wie folgt berechnet werden:

$$\overline{s} = x \oplus y \oplus c = x \oplus (y \oplus c)$$

Warum nochmal geklammert wurde, wird gleich klar. Das Carrybit $\overline{c}$ nimmt genau dann den Wert 1 an, wenn mindestens zwei Bits der Eingabe x, y, c den Wert 1 haben. Also kann $\overline{c}$ wie folgt berechnet werden:

$$\overline{c} = (x \wedge y) \vee (x \wedge c) \vee (y \wedge c) = (x \wedge (y \vee c)) \vee (y \wedge c) = (x \wedge (y \oplus c)) \vee (y \wedge c)$$

Durch die letzten Schritt, in dem $(y \vee c)$ durch $(y \oplus c)$ ersetzt wurde, ist es nun möglich, ein Zwischenergebnis der Berechnung des Summenbits $\overline{s}$ zu benutzen, d.h. wir sparen uns eine Berechnung. Die Ersetzung ist erlaubt, da sich die Ergebnisse von $(y \vee c)$ und $(y \oplus c)$ nur für $y = 1$ und $c = 1$ unterscheiden. In diesem Fall wird jedoch auch $(y \wedge c)$ zu 1 ausgewertet und das Ergebnis stimmt immer noch. Abbildung 1 auf Seite 6 zeigt, wie ein Schaltkreis für einen Fulladder aussieht. Die Größe eines Fulladders beträgt also 5 und die Größe eines Halfadders beträgt 2. Damit hat unser Additionsschaltkreis also eine Gesamtgröße von $5(n-1)+2 = 5n-3$. Bei der Tiefe müssen wir jedoch genau hinschauen. Die Tiefe eines einzelnen Fulladders beträgt 3, wie in Abbildung 1 leicht ersichtlich ist. Wir könnten nun also glauben, dass die Tiefe unseres Additionsschaltkreis somit $3(n-1)+1 = 3n-2$ beträgt. Dies ist jedoch nicht der Fall, da wir das Übertragsbit auf der dritten Ebene von Abbildung 1 quasi nach links oben in den nächsten Fulladder klappen können. Das bedeutet, dass der letzte Schritt der Berechnung des Übertrags quasi schon im nächsten Fulladder stattfinden kann, womit fast alle Fulladder in unserem Additionsschaltkreis eine Tiefe von 2 bekommen. Eine Ausnahme bildet der letzte Fulladder, dessen Übertragsbit s_n berechnet. Den Halfadder können wir uns ebenfalls in den ersten Fulladder integriert vorstellen, da er das Übertragsbit c_0 in Tiefe 1 berechnet. Es ergibt sich somit eine Tiefe von $2(n-2)+3 = 2n-4+3 = 2n-1$.

Wir überlegen uns nun, wie gut die erreichte Größe und Tiefe ist. Bezüglich der Größe konnte Redkin im Jahr 1981 zeigen, dass kein Additionsschaltkreis mit weniger als $5n-3$ Bausteinen auskommt. Die erreichte Größe entspricht damit diesem Optimum. Bezüglich der Tiefe können wir jedoch überhaupt nicht zufrieden sein, da die Addition im Prinzip eine relativ einfach Funktion ist und wir noch sehen werden, dass es viel bessere Verfahren mit geringerer Tiefe gibt. Wenn wir das Verfahren der Schulmethode genauer betrachten, so zeigt sich, dass das Warten auf die Carrybits der ausschlaggebende Punkt ist, der das Verfahren ineffizient macht.

3.2 Von Neumann Addierwerke

Frage 3.3: Was ist die Idee beim von Neumann Addierwerk?

Ein von Neumann Addierwerk ist ein Schaltwerk mit einer im Durchschnitt geringen parallelen Rechenzeit. Die Betonung liegt dabei auf „im Durchschnitt“, da die Rechenzeit im worst case linear ist, weshalb sich eine Umsetzung in Schaltkreise nicht lohnt, da diese natürlich auch in der Lage sein müssen, den worst case zu behandeln. Wir werden uns diese Art von Schaltwerken jedoch trotzdem näher anschauen.

Die grundlegende Aufgabe bei der Addition besteht darin, zu zwei Binärzahlen x und y die Summe s zu berechnen. Die Idee beim von Neumann Schaltwerk ist es nun, aus den Zahlen x und y zwei neue Zahlen x^{neu} und y^{neu} zu berechnen, deren Summe denselben Wert wie die Summe von x und y hat. Wir schreiben dies als $|x| + |y| = |x^{neu}| + |y^{neu}|$. Wenn wir die Zahlen ziellos verändern, bringt uns dies natürlich nicht weiter. Schaffen wir es jedoch, dass nach einem dieser Schritte $|y| = 0$ (also $y = (0, \ldots, 0)$) gilt, dann muss das Ergebnis der

Addition in x stehen. Denn dann gilt:

$$|x| + |y| = |x^{neu(1)}| + |y^{neu(1)}| = \cdots = |x^{neu(k)}| + |y^{neu(k)}| = |x^{neu(k)}| + 0 = |x^{neu(k)}|$$

Damit wäre die grundlegende Idee schonmal klar. Jetzt müssen wir uns nur noch ein Verfahren überlegen, mit dem wir in jedem Schritt die Werte x^{neu} und y^{neu} berechnen. Grob kann das Verfahren so beschrieben werden, dass in jedem Schritt die Summenbits berechnet werden und alle Carrybits, von denen wir in diesem Schritt wissen, dass sie auftreten. Wir überlegen uns nun zuerst, inwiefern sich dieses Vorgehen vom Schulverfahren unterscheidet. Beim Schulverfahren addieren wir jeweils von hinten beginnend die jeweiligen Binärstellen der beteiligten Zahlen und ein Carrybit. Durch diese Vorgehensweise gehen die Carrybits von rechts nach links in die Rechnung ein bzw. wenn ein Carrybit an der i-ten Stelle der Rechnung eingeht, dann wird es bei der Behandlung der i-ten Stelle auch anliegen. Beim von Neumann Addierwerk gehen wir nun anders vor. Hier werden die Carrybits in einem Schritt parallel berechnet, jedoch nur an den Stellen, an denen wir mit absoluter Sicherheit wissen, dass ein Carrybit auftritt. Dies sind die Stellen mit $x_i = 1$ und $y_i = 1$. Diese Informationen werden wir in y^{neu} speichern, wobei sich y_{i+1}^{neu} als $y_{i+1}^{neu} = x_i \wedge y_i$ berechnet, da das Carrybit ja eine Stelle weiter links in die Rechnung eingeht. Wie berechnen wir nun x^{neu}? Einige der auftretenden Carrybits haben wir ja schon berechnet. Es müssen noch Summenbits von x und y berechnet werden. Diese werden wir in x^{neu} speichern. Die Summenbits werden ganz normal als $x_i^{neu} = x_i \oplus y_i$ berechnet. Insgesamt ergibt sich also für $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$:

$$\begin{aligned} x^{neu} &= (x_n \oplus y_n,\ x_{n-1} \oplus y_{n-1}, \ldots,\ x_0 \oplus y_0) \\ y^{neu} &= (x_{n-1} \wedge y_{n-1}, \ldots, x_0 \wedge y_0, 0) \end{aligned}$$

Die Forderung $|x| + |y| = |x^{neu}| + |y^{neu}|$ ist dabei erfüllt, da die Carrybits nach links weiter gegeben werden. Wenn wir die beiden Formeln genauer betrachten, sieht es so aus, als ob x^{neu} und y^{neu} jeweils um eine Stelle wachsen. Es ist jedoch leicht einzusehen, dass die $n+1$ Bits $x_n, \ldots, x_0$ bzw. $y_n, \ldots, y_0$ ausreichen. Warum ist dies der Fall? 2^{n+1} ist die erste Zahl, für die $n+2$ Bits benötigt werden, um sie darzustellen. Die Zahlen x und y sind n Bits lang. Mit n Bits können die Zahlen $0, \ldots, 2^n - 1$ dargestellt werden. Damit gilt: $|x| + |y| \leq 2^n - 1 + 2^n - 1 = 2 \cdot 2^n - 2 = 2^{n+1} - 2 < 2^{n+1}$. Damit reichen $n+1$ Bits also sicher aus, da kein Carrybit an der $n+2$-ten Stelle (also an x_{n+1}) entstehen kann.

Es ist ebenfalls leicht zu zeigen, dass der Algorithmus terminiert. Nach dem ersten Schritt gilt $y_0 = 0$ und dieser Wert ändert sich auch während des ganzen Algorithmus nicht mehr. Gilt nun jedoch, dass für einem Paar (x_i, y_i) mindestens ein Bit den Wert 0 hat, so wird auch y_{i+1} im nächsten Schritt den Wert 0 haben, da es als $y_{i+1} = x_i \wedge y_i$ berechnet wird. Auch dieser Wert wird sich nicht mehr ändern. Die Konsequenz daraus ist, dass sich die Nullen mit jedem Schritt immer mehr von rechts nach links in y „hineinfressen“. Nach dem ersten Schritt ist also y_0 ganz sicher 0, nach dem zweiten y_1 und so also nach $n+1$ Schritten y_n und damit ganz y. Der Algorithmus terminiert somit nach spätestens $n+1$ Schritten.

Frage 3.4: Wie werden von Neumann Addierwerke in Hardware realisiert? Welche Größe haben sie? Wie groß ist die parallele Rechenzeit?

Um die Formeln für x^{neu} und y^{neu} zu berechnen, werden n Halfadder und ein zusätzlicher $\oplus$-Baustein (für $x_n \oplus y_n$) benötigt. Die Abbruchbedingung für den Algorithmus kann mit einer

einfachen Bedingung getestet werden:

$$y_n \vee \cdots \vee y_0 = 0$$

Bei Schaltwerken können Zwischenergebnisse mit Hilfe von Delay-Bausteinen zwischengespeichert und erneut als Eingabe verwendet werden. In unserem Fall sind diese Zwischenergebnisse x^{neu} und y^{neu}. Die Größe eines Schaltwerks für die von Neumann Addition ist also linear in n. Zur Berechnung werden Halfadder bzw. andere Bausteine der Tiefe 1 benutzt, weshalb die parallele Rechenzeit nur noch von der Anzahl der Schritte, die während einer Berechnung gemacht werden, abhängt.

Frage 3.5: Welche durchschnittliche parallele Rechenzeit hat ein von Neumann Addierwerk?

Wir haben schon beobachtet, dass die parallele Rechenzeit proportional zur Anzahl der verwendeten Takte für eine Eingabe ist. Als nächstes müssen wir also die möglichen Eingaben näher betrachten. Dabei charakterisieren wir eine Eingabe über die Länge ihres längsten 1-Blocks und werden zeigen, dass die Rechenzeit von dieser Größe abhängt. Ein 1-Block der Länge $l \geq 0$ liegt vor, wenn es ein Paar x_i und y_i gibt, für welches $x_i = y_i = 1$ und $x_{i+j} + y_{i+j} = 1$ für $1 \leq j \leq l$ gilt. So enthält das folgenden Beispiel einen 1-Block der Länge 3:

x_5	x_4	x_3	x_2	x_1	x_0
0	1	1	1	1	0
0	0	0	0	1	0
y_5	y_4	y_3	y_2	y_1	y_0

Mit $L(x, y)$ bezeichnen wir im Folgenden nun die Länge des längsten 1-Blocks für x und y. Dabei ist $L(x, y) = -1$ wenn x und y kein Paar mit $x_i = y_i = 1$ enthält und $L(x, y) = 0$, wenn für jedes Paar mit $x_i = y_i = 1$ gilt, dass $x_{i+1} = y_{i+1} = 0$ bzw. $x_{i+1} = y_{i+1} = 1$ ist.

Wir behaupten nun, dass ein von Neumann Addierwerk zur Addition von zwei Zahlen x und y höchstens $L(x, y) + 2$ Takte benötigt. Für den Fall, dass $L(x, y) = -1$ gilt, stimmt die Behauptung. Denn in diesem Fall hat für jedes Paar (x_i, y_i) mindestens eines der beiden Bits den Wert 0. Damit wird nach einem Takt y_{i+1} ebenfalls zu 0, da es sich ja als $x_i \wedge y_i$ berechnet. Also haben nach einem Takt alle Bits von y den Wert 0 und der Algorithmus terminiert. Es wird somit ein Takt benötigt, was auch mit der Behauptung übereinstimmt: $1 \leq L(x, y) + 2 = -1 + 2 = 1$. Als nächstes betrachten wir nun den Fall $L(x, y) \neq -1$, also $0 \leq L(x, y) \leq l$. Wir machen die folgende Beobachtung: $L(x^{neu}, y^{neu}) \leq L(x, y) - 1$, d.h. die Länge des neuen 1-Blocks wird mit jedem Takt um mindestens 1 kleiner als die Länge des ursprünglichen Blocks. Dies liegt daran, dass für ein Paar $x_i = y_i = 1$ im nächsten Schritt nicht wieder $x_i = y_i = 1$ gelten kann, da $1 \oplus 1 = 0$ ist. Hat jedoch genau eines der Bits x_{i+1} oder y_{i+1} den Wert 1, so gilt im nächsten Schritt $x_{i+1} = y_{i+1} = 1$. Ein Beispiel:

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{array} \quad \rightarrow \quad \begin{array}{cccc} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{array}$$

Aus $L(x^{neu}, y^{neu}) \leq L(x, y) - 1$ können wir folgern, dass nach spätestens $L(x, y) + 1$ Takten $L(x^{neu}, y^{neu}) = -1$ gilt. Dann wird noch ein weiterer Takt benötigt, damit der Algorithmus terminiert. Insgesamt somit höchstens $L(x, y) + 2$ Takte.

Aus diesem Ergebnis können wir auch schnell auf die worst case Rechenzeit schließen. Die Länge des längsten 1-Block kann höchstens $n - 1$ betragen. Dies ist z.B. für $x_0 = y_0 = 1$,

$x_1 = \cdots = x_{n-1} = 1$ und $y_1 = \cdots = y_{n-1} = 0$ der Fall. Dann wird das Maximum von $n+1$ Takten ausgeführt. Der best case liegt vor, wenn zu Beginn sofort $|y| = 0$ gilt. In diesem Fall werden 0 Takte benötigt.

Doch wie berechnet sich nun die durchschnittliche Anzahl an Takten? Dazu verwenden wir ein Modell, bei dem die Eingabe x und y zufällig ausgewürfelt wird. Die Zufallsvariable l gibt im Folgenden die Länge des längsten auf diese Art und Weise entstehenden 1-Blocks an. Wir berechnen nun den Erwartungswert $E(l)$, d.h. die erwartete (und damit durchschnittliche) Länge des längsten 1-Blocks. Allerdings werden wir dabei nicht berücksichtigen, dass ein 1-Block mit $x_i = y_i = 1$ beginnen muss. Allerdings wollen wir $E(l)$ ja auch nach oben abschätzen. In einem 1-Block muss an einer Stelle i für die Summe der Bits x_i und y_i gelten, dass $x_i + y_i = 1$ ist. Es gibt zwei von vier Bitpaaren, bei denen dies der Fall ist. Damit beträgt die Wahrscheinlichkeit ein solches Bitpaar zu erzeugen genau $\frac{1}{2}$. Wir können uns also vorstellen, dass wir bei der zufälligen Erzeugung der Eingabe von hinten nach vorne über die beiden Bitstrings langlaufen und jedes Mal eine Münze werfen, die uns sagt, ob wir ein günstiges oder ungünstiges Bitpaar erzeugen. Wir schätzen den Erwartungswert $E(l)$ wie folgt ab:

$$
\begin{aligned}
E(l) &\leq \sum_{i=1}^{n} i \cdot \mathit{Prob}(l = i) = \sum_{i=1}^{n} \mathit{Prob}(l \geq i) && (2)\\
&= \sum_{i=1}^{\lceil \log n \rceil} \mathit{Prob}(l \geq i) + \sum_{\lceil \log n \rceil + 1}^{n} \mathit{Prob}(l \geq i) && (3)\\
&\leq \sum_{i=1}^{\lceil \log n \rceil} 1 + \sum_{\lceil \log n \rceil + 1}^{n} n \cdot 2^{-i} && (4)\\
&\leq \lceil \log n \rceil + n \cdot \sum_{\lceil \log n \rceil + 1}^{n} 2^{-i} && (5)\\
&\leq \lceil \log n \rceil + n \cdot \frac{1}{n} = \lceil \log n \rceil + 1 && (6)
\end{aligned}
$$

Bei dieser Abschätzung ergeben sich wahrscheinlich einige Fragen, warum die Umformungen so durchgeführt werden dürfen. Die Umformung in (2) können wir uns schnell klarmachen, wenn wir die Summe etwas umstellen:

$$
\overbrace{\begin{array}{ccccc}
\mathit{Prob}(l = n) & \mathit{Prob}(l = n) & \dots & \mathit{Prob}(l = n) & \mathit{Prob}(l = n) \\
 & \mathit{Prob}(l = n-1) & \dots & \mathit{Prob}(l = n-1) & \mathit{Prob}(l = n-1) \\
 & & \ddots & \ddots & \vdots \\
 & & & \mathit{Prob}(l = 2) & \mathit{Prob}(l = 2) \\
 & & & & \mathit{Prob}(l = 1) \\
\uparrow & \uparrow & \dots & \uparrow & \uparrow \\
\mathit{Prob}(l \geq n) & \mathit{Prob}(l \geq n-1) & \dots & \mathit{Prob}(l \geq 2) & \mathit{Prob}(l \geq 1)
\end{array}}^{n\text{-mal}}
$$

In (3) wird die Summe einfach nur praktisch aufgeteilt. Dann nutzen wir in (4) für die Abschätzung der linken Summe aus, dass $\mathit{Prob}(l \geq i) \leq 1$ gilt und schätzen für die rechte Summe die Wahrscheinlichkeit $\mathit{Prob}(l \geq i)$ etwas genauer ab. Die Wahrscheinlichkeit mindestens i für uns günstige Bitpaare mit $x_i + y_i = 1$ auszuwürfeln, beträgt $\frac{1}{2^i} = 2^{-i}$. Dies können

Algorithmus 8 Allgemeiner Algorithmus zur Addition zweier Zahlen

1. Berechne für $0 \leq i \leq n-1$ die Werte $u_i = x_i \wedge y_i$ und $v_i = x_i \oplus y_i$.
2. Berechne mit Hilfe der u- und v-Werte die Carrybits $c_{n-1}, \ldots, c_0$.
3. Berechne die endgültige Summe s. Dabei ist $s_n = c_{n-1}$ und $s_0 = v_0$. Für die restlichen Bits s_i mit $1 \leq i \leq n-1$ gilt $s_i = v_i \oplus c_{i-1}$.

wir uns wie folgt klarmachen. Wir werfen für jede Position eine Münze und immer wenn Kopf erscheint, bekommen wir ein günstiges Paar an der betrachteten Position. Sobald das Ereignis „i-mal Kopf“ eingetreten ist, wird der Block mindestens Länge i haben. Wir schätzen nach oben ab, dass der Block, dessen Länge mindestens i beträgt, an n Positionen beginnen kann, was er natürlich eigentlich nicht kann. Insgesamt gilt also $Prob(l \geq i) \leq n \cdot 2^{-i}$. Für die Abschätzung von (5) nach (6) müssen wir nur noch folgende Aussage beweisen:

$$\sum_{\lceil \log n \rceil + 1}^{n} 2^{-i} \leq \frac{1}{n}$$

Dies machen wir wie folgt:

$$\begin{aligned}
\sum_{\lceil \log n \rceil + 1}^{n} 2^{-i} &= \frac{1}{2^{\lceil \log n \rceil + 1}} + \frac{1}{2^{\lceil \log n \rceil + 2}} + \frac{1}{2^{\lceil \log n \rceil + 3}} + \cdots + \frac{1}{2^n} \\
&= \frac{1}{2^1} \cdot \frac{1}{2^{\lceil \log n \rceil}} + \frac{1}{2^2} \cdot \frac{1}{2^{\lceil \log n \rceil}} + \frac{1}{2^3} \cdot \frac{1}{2^{\lceil \log n \rceil}} + \cdots + \frac{1}{2^{n - \lceil \log n \rceil}} \cdot \frac{1}{2^{\lceil \log n \rceil}} \\
&\leq \frac{1}{2^1} \cdot \frac{1}{n} + \frac{1}{2^2} \cdot \frac{1}{n} + \frac{1}{2^3} \cdot \frac{1}{n} + \cdots + \frac{1}{2^{n - \lceil \log n \rceil}} \cdot \frac{1}{n} \\
&\leq \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^{n - \lceil \log n \rceil}} \right) \cdot \frac{1}{n} \\
&\leq \left(\sum_{i=1}^{\infty} \frac{1}{2^i} \right) \cdot \frac{1}{n} = 1 \cdot \frac{1}{n} = \frac{1}{n}
\end{aligned}$$

Es ergibt sich also ein Erwartungswert von $E(l) \leq \lceil \log n \rceil + 1$. Damit beträgt die durchschnittliche Rechenzeit für ein von Neumann Addierwerk höchstens $\lceil \log n \rceil + 3$.

3.3 Carry-Look-Ahead Addierer

Frage 3.6: Was ist die Hauptidee beim Carry-Look-Ahead Addierer?

Wie bei der Schultmethode der Addition recht schnell deutlich wird, liegt das Hauptproblem bei der Addition in der Berechnung der Carrybits. Algorithmus 8 zeigt dies ebenfalls nochmal auf, denn die Hauptarbeit wird bei diesem Algorithmus im zweiten Schritt gemacht. Die Schulmethode ist so ineffizient, da die Carrybits von rechts nach links (also von c_0 bis c_{n-1}) berechnet werden. Dafür benötigt diese Methode die absolut minimale Anzahl an Bausteinen.

Der Carry-Look-Ahead Addierer wird die Carrybits in einer weit geringeren Tiefe berechnen als die Schulmethode, dafür jedoch auch eine kubische Größe erreichen. Dazu benutzen wir im Folgenden die in Schritt 1 von Algorithmus 8 berechneten Informationen und klassifizieren jedes Bitpaar (x_i, y_i) anhand dieser. Wir bilden drei Klassen:

Eliminate: Ein Paar (x_i, y_i) bzw. die Stelle i heißt Eliminate oder E-Stelle, wenn $(u, v) = (0, 0)$ gilt. Beide Bits haben also den Wert 0. Eine solche Stelle wird in keinem Fall einen Übertrag erzeugen, da die Summe von $x_i = y_i = 0$ und einem Carrybit, welches unter Umständen von rechts kommt, höchstens 1 sein kann. Es gilt in diesem Fall also stets $c_i = 0$.

Propagate: Eine Stelle ist eine P-Stelle, wenn $(u, v) = (0, 1)$ gilt. Bei einer P-Stelle hat also genau eines der Bits den Wert 1. Ob diese Stelle nun einen Übertrag verursacht, hängt nur davon ab, ob ein Carrybit von rechts kommt oder nicht. Kommt kein Carrybit, ist die Summe wieder 1 und es wird kein Übertrag erzeugt. Ansonsten hat die Summe die Binärdarstellung 10 und es wird ein Übertrag erzeugt. In ersterem Fall gilt somit $c_i = 0$ und im zweiten $c_i = 1$.

Generate: Für das Generate bleibt nun nur noch ein Fall übrig. Es Stelle heißt G-Stelle, wenn $(u, v) = (1, 0)$ gilt. In diesem Fall haben also beide Bits den Wert 1 (da $u_i = x_i \wedge y_i$ ansonsten nicht 1 werden kann). Eine solche Stelle wird in jedem Fall einen Übertrag generieren, da die Binärdarstellung der Summe der drei beteiligten Bits mindestens 10 und höchstens 11 ist. Für eine G-Stelle gilt also in jedem Fall $c_i = 1$.

Das folgende Beispiel dürfte die Einteilung recht schnell klar machen:

$$\begin{array}{rcccc} x & = & 0 & 1 & 0 & 1 \\ y & = & 0 & 0 & 1 & 1 \\ & & E & P & P & G \end{array}$$

Wir sind an der Berechnung der Werte $c_{n-1}, \ldots, c_0$ interessiert, da wir in Schritt 3 von Algorithmus 8 für $1 \leq i \leq n-1$ die endgültigen Summenbits $s_i = v_i \oplus c_{i-1}$ berechnen wollen. Wenn wir uns die drei Klassifizierungen anschauen, gibt es zwei Fälle in denen c_i sofort feststeht. Im Falle einer E-Stelle gilt sofort $c_i = 0$ und im Fall einer G-Stelle $c_i = 1$. Bei einer P-Stelle hängt der Übertrag c_i von der vorhergehenden Stelle c_{i-1} ab. Erzeugt diese einen Übertrag bzw. gibt diesen weiter, so ist auch $c_i = 1$. Dies ist der Fall wenn die Stelle $i-1$ eine G-Stelle ist oder wenn sie eine P-Stelle ist, die einen Übertrag erzeugt. Wir können also zusammenfassen: Es gilt $c_i = 1$, wenn die Stelle i eine G-Stelle ist oder eine P-Stelle, die von einer G-Stelle gefolgt wird oder von einer ununterbrochenen Folge von P-Stellen, die durch eine G-Stelle abgeschlossen wird.

Als nächstes überlegen wir uns, wie wir erkennen können, ob eine Stelle eine G- oder eine P-Stelle ist. Im Falle einer G-Stelle müssen beide Bits x_i und y_i den Wert 1 haben. Also gilt für G-Stellen $u_i = x_i \wedge y_i = 1$. Bei einer P-Stelle ist genau eines der Bits 1, also gilt für eine P-Stelle $v_i = x_i \oplus y_i = 1$. Wir können also die in Schritt 1 berechneten Informationen direkt zur Klassifizierung benutzen. Wir werden die oben gemachten Beobachtungen nun verwenden, um eine Formel anzugeben, die den Wert c_i für jede Stelle berechnet:

$$c_i = \bigvee_{0 \leq j \leq i} u_j v_{j+1} \ldots v_i$$

Der Fall $j = i$ ist dabei der Fall, dass die Stelle i eine G-Stelle ist. In diesem Fall ergibt sich $u_i v_{i+1} v_i$ als Summand, was wir als u_i interpretieren, da sich ansonsten ein falsches Ergebnis ergibt. Alle anderen Fälle sind die Fälle, bei denen es an der Stelle j ein Generate gibt, welches ununterbrochen bis zur Stelle i propagiert wird. Wir testen also einfach alle Fälle durch, um zu schauen, ob an der Stelle i ein Übertrag entsteht.

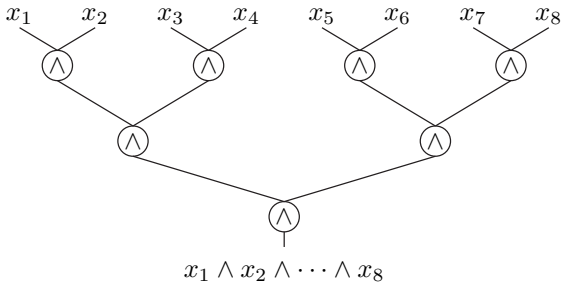


Abbildung 7: Balancierter Baum der Tiefe $\lceil \log n \rceil$ zur Berechnung von $x_1 \wedge x_2 \wedge \cdots \wedge x_8$

Wir wissen nun also, wie wir die Carry-Bits berechnen können. Durch eine Fallunterscheidung für alle drei Klassen kann schnell gezeigt werden, dass dann auch die endgültigen Summenbits $s_n, \ldots, s_0$ korrekt berechnet werden.

Frage 3.7: Wie groß und wie tief ist ein B_2-Schaltkreis zur Realisierung eines Carry-Look-Ahead Addierers?

Die Größe eines B_2-Schaltkreises, der einen Carry-Look-Ahead Addierer auf die oben beschriebene Art realisiert, ist leicht abzuschätzen. Zuerst wollen wir die Anzahl aller Summenden für alle $c_{n-1}, \ldots, c_0$ abschätzen. Diese ergibt sich als:

$$\binom{n}{2} + n = \frac{n(n+1)}{2} + n = \frac{1}{2}n^2 + \frac{1}{2}n + n = \frac{1}{2}n^2 + \frac{3}{2}n = O(n^2)$$

Der erste Summand ist die Anzahl aller möglichen Paare i und j, denn wir müssen aus den n Stellen zwei Stellen auswählen, wobei die höherwertige Stelle i ist (also die Stelle, an der der Übertrag ankommen soll) und die niederwertige j (also die Stelle, wo das Generate liegen soll). Es gibt $\binom{n}{2}$ solcher Auswahlen. Da auch $i = j$ sein kann (die Stelle i ist selbst ein Generate) müssen wir zusätzlich noch n addieren. Wie groß kann jeder Summand höchstens werden? Jeder Summand kann aus höchstens n Variablen u bzw. v bestehen, die wiederum mit einem Baustein berechnet werden können. Insgesamt ergibt sich somit eine Größe von $O(n^3)$.

Kommen wir nun zur Abschätzung der Tiefe. Gefragt ist nach einer Realisierung als B_2-Schaltkreis, d.h. jeder Baustein hat nur zwei Eingänge. Wir überlegen uns nun zuerst, wie wir einen Summanden für ein c_i berechnen können. Um $u_j v_{j+1} \ldots v_i$ zu berechnen, können wir einen balancierten Baum wie in Abbildung 7 verwenden. Ein Summand besteht aus höchstens n Variablen, wobei n jedoch keine Zweierpotenz sein muss. Damit hat ein solcher Baum die Tiefe $\lceil \log n \rceil$. Als nächstes muss für jedes c_i die Disjunktion dieser Summanden berechnet werden. Diese kann ebenfalls mit einem balancierten Baum realisiert werden. Da ein c_i höchstens n Summanden enthalten kann, reicht hier also auch die Tiefe $\lceil \log n \rceil$.

Damit können wir den zweiten Schritt aus Algorithmus 8 also in Größe $O(n^3)$ und Tiefe $2 \cdot \lceil \log n \rceil$ durchführen. Da Schritt eins und drei beide in Tiefe 1 realisiert werden können, benötigt ein Carry-Look-Ahead Addierer also insgesamt die Tiefe $2 \cdot \lceil \log n \rceil + 2$.

Frage 3.8: In welcher Tiefe kann ein Carry-Look-Ahead Addierer in U-Schaltkreisen realisiert werden?

Wir wollen nun überlegen, in welcher Tiefe wir einen Carry-Look-Ahead Addierer mit einem U-Schaltkreis realisieren können, damit wir die Addition in die AC-Hierarchie einordnen

können. Dabei werden wir sehen, dass $ADD \in AC_{0,3}$ gilt. Die Addition kann also in Tiefe 3 von einem U-Schaltkreis berechnet werden.

Die Klasse der U-Schaltkreis ist dadurch charakterisiert, dass ein unbeschränkter Fan-In möglich ist (unbounded fan-in), jedoch nur die Operationen $\wedge, \vee$ und $\neg$ zugelassen sind. Da wir bei der Addition, wie sie in Algorithmus 8 beschrieben ist, die Operation $\oplus$ benutzen, müssen wir diese durch die drei erlaubten Operationen ausdrücken. Dies ist jedoch kein Problem, denn es gelten die folgenden beiden Formel, die wir auch beide benötigen:

$$\begin{aligned} x \oplus y &= (\overline{x} \wedge y) \vee (x \wedge \overline{y}) \\ x \oplus y &= (\overline{x} \vee \overline{y}) \wedge (x \vee y) \end{aligned}$$

Als nächstes überlegen wir uns nun, welche Tiefe eine naive Umsetzung in einen U-Schaltkreis hätte. Der erste Schritt würde die Tiefe 2 benötigen, da wir für die $\oplus$-Operation auf der ersten Ebene zwei $\wedge$-Bausteine benötigen und auf der zweiten Ebene einen $\vee$-Baustein. Der zweite Schritt benötigt ebenfalls die Tiefe 2. Zuerst werden die Konjunktionen $u_j v_{j+1} \dots v_i$ berechnet und anschließend auf der nächsten Ebene deren Konjunktionen. An dieser Stelle sparen wir im U-Schaltkreis also am meisten gegenüber dem B_2-Schaltkreis, da keine balancierten Bäume mehr nötig sind. Die teuerste Operation im dritten Schritt ist wieder das $\oplus$, womit wir diese Stufe ebenfalls in Tiefe 2 schaffen. Insgesamt hat eine naive Umsetzung also die Tiefe 6.

Wir können jedoch auch die Tiefe 3 erreichen, indem wir die Stufen „ineinander schieben“. Wir werden dabei nun „das Feld von hinten aufrollen“, d.h. wir betrachten zuerst die Berechnung des Summenbits $s_i = v_i \oplus c_{i-1}$. Dieses berechnet sich mit $\wedge$- und $\vee$-Bausteinen ja wie folgt:

$$s_i = v_i \oplus c_{i-1} = (\overline{v}_i \wedge c_{i-1}) \vee (v_i \wedge \overline{c}_{i-1})$$

Wir möchten diese Darstellung auch gerne für s_n und s_0 benutzen, die ja extra berechnet werden. Es gilt $s_n = c_{n-1}$, also muss in der obigen Formel $v_n = x_n \oplus y_n$ zu 0 werden. Dies erreichen wir, indem wir $x_n = y_n = 0$ setzen. Damit $s_0 = v_0$ gilt muss c_{i-1}, also in diesem Fall $c_{-1} = 0$ gelten. Wir führen also zwei weitere Variablen x_{-1} und y_{-1} ein und setzen diese auf 0. Dann gilt auch $c_{-1} = 0$, da an dieser Stelle kein Übertrag generiert werden kann, der in die Summe von x_0 und y_0 eingeht.

Nun betrachen wir den ersten Summanden $(\overline{v}_i \wedge c_{i-1})$ im obigen Ausdruck etwas genauer, damit wir eine Idee bekommen, wie die Berechnungen „ineinander geschoben“ werden. Mit Hilfe des Distributivgesetzes lässt er sich wie folgt umformen:

$$\overline{v}_i \wedge c_{i-1} = \overline{v}_i \wedge \bigvee_{0 \le j \le i-1} u_j v_{j+1} \dots v_{i-1} = \bigvee_{0 \le j \le i-1} u_j v_{j+1} \dots v_{i-1} \overline{v}_i$$

Damit wir denselben „Trick“ auch mit $\overline{c}_i$ (bzw. $\overline{c}_{i-1}$) durchführen können, überlegen wir uns, wie wir $\overline{c}_i$ so ähnlich wie c_i darstellen können. Die Variable $\overline{c}_i$ drückt aus, dass an der Stelle i kein Übertrag auftritt kann. Dies ist z.B. der Fall, wenn an der Stelle i ein Eliminate vorliegt. Es liegt jedoch auch kein Übertrag vor, wenn die Stelle i eine P-Stelle ist, gefolgt von einer ununterbrochenen (eventuell auch leeren) Folge von P-Stellen, die durch ein Eliminate abgeschlossen werden. Wir können also genauso argumentieren, wie bei c_i, nur dass ein Eliminate auch schon an der Stelle -1 entstehen kann bzw. sogar ganz sicher entsteht. Für das Eliminate führen wir also eine neue Variable w_i ein, die den Wert 1 liefert, wenn die i-te Stelle ein Eliminate ist. Es gilt also $w_i = \overline{x}_i \wedge \overline{y}_i$. Es gilt dann also:

$$\overline{c}_i = \bigvee_{-1 \le j \le i} w_j v_{j+1} \dots v_i$$

Wir benutzen diese Darstellung für $\overline{c}_{i-1}$ und erhalten insgesamt für s_i:

$$s_i = (\overline{v}_i \wedge c_{i-1}) \vee (v_i \wedge \overline{c}_{i-1}) = \bigvee_{0 \leq j \leq i-1} u_j v_{j+1} \dots v_{i-1} \overline{v}_i \quad \vee \bigvee_{-1 \leq j \leq i-1} w_j v_{j+1} \dots v_{i-1} v_i$$

Die dritte Ebene unseres U-Schaltkreises besteht also aus $\vee$-Bausteinen und berechnet die Disjunktion der $u_j v_{j+1} \dots v_{i-1} \overline{v}_i$ bzw. $w_j v_{j+1} \dots v_{i-1} v_i$. Da wir einen Schaltkreis der Tiefe 3 realisieren wollen, müssen wir diese Summanden also in Tiefe 2 berechnen können. Die Summanden bestehen aus den Werten u_i, w_i, v_i sowie $\overline{v}_i$. Unser Ziel wird es nun sein, diese Werte so zu berechnen, dass sich auf der ersten Ebene lauter $\vee$-Bausteine befinden. Auf der zweiten Ebene sollen nur $\wedge$-Bausteine benutzt werden, damit wir diese Berechnungen in die Berechnung der Summanden $u_j v_{j+1} \dots v_{i-1} \overline{v}_i$ bzw. $w_j v_{j+1} \dots v_{i-1} v_i$ „einbetten“ können. Tatsächlich können wir alle vier Werte auf diese Art berechnen, wobei wir in der Formel für $\overline{v}_i$ zweimal die Regel von deMorgan anwenden:

$$\begin{aligned}
u_i &= x_i \wedge y_i \\
w_i &= \overline{x}_i \wedge \overline{y}_i \\
v_i &= x_i \oplus y_i = (\overline{x}_i \vee \overline{y}_i) \wedge (x_i \vee y_i) \\
\overline{v}_i &= \neg(x_i \oplus y_i) = \neg((\overline{x}_i \wedge y_i) \vee (x_i \wedge \overline{y}_i)) = \neg(\overline{x}_i \wedge y_i) \wedge \neg(x_i \wedge \overline{y}_i) = (x_i \vee \overline{y}_i) \wedge (\overline{x}_i \vee y_i)
\end{aligned}$$

Auf der ersten Ebene wird also $(x_i \vee y_i)$, $(\overline{x}_i \vee \overline{y}_i)$, $(x_i \vee \overline{y}_i)$ und $(\overline{x}_i \vee y_i)$ berechnet. Anschließend werden auf der zweiten Ebene die endgültigen Werte für u_i, w_i, v_i und $\overline{v}_i$ und damit gleichzeitig die Summanden $u_j v_{j+1} \dots v_{i-1} \overline{v}_i$ bzw. $w_j v_{j+1} \dots v_{i-1} v_i$ berechnet. Dabei benötigen wir auf der ersten Ebene $4n$ $\vee$-Bausteine, auf der zweiten Ebene $O(n^2)$ $\wedge$-Bausteine und auf der dritten Ebene $n+1$ $\vee$-Bausteine. Damit haben wir also eine $\vee$-$\wedge$-$\vee$-Schaltkreis der Tiefe 3 konstruiert, womit $ADD \in AC_{0,3}$ gilt.

3.4 Conditional Sum Addierer

Frage 3.9: Wie ist die Select-Funktion definiert und was macht sie?

Die Funktion $sel \in B_3$ ist wie folgt definiert:

$$sel(x, y, z) = \begin{cases} y & \text{falls } x = 0 \\ z & \text{fall } x = 1 \end{cases}$$

Die Funktion sel wählt also in Abhängigkeit vom Bit x eines der beiden Bits y oder z aus. Sie kann wie folgt implementiert werden: $sel(x, y, z) = \overline{x}y \vee xz$. Die Funktion sel hat also die Größe 3 und die Tiefe 2.

Frage 3.10: Was ist die grundlegende Idee beim Conditional Sum Addierer?

Die Aufgabe besteht immer noch darin, zwei Zahlen $x = (x_{n-1}, \dots, x_0)$ und $y = (y_{n-1}, \dots, y_0)$ zu addieren. Im Folgenden gehen wir jedoch der Einfachheit halber davon aus, dass n eine Zweierpotenz ist, dass also $n = 2^k$ mit $k \geq 0$ gilt. Warum wir dies machen, wird gleich ersichtlich. Als nächstes überlegen wir uns nun, was wir beachten müssen, wenn wir die Summe von x und y wie folgt aufgeteilt berechnen:

$$\begin{array}{cccc|ccccc}
x_{n-1} & x_{n-2} & \dots & x_{\frac{n}{2}} & | & x_{\frac{n}{2}-1} & \dots & x_1 & x_0 \\
y_{n-1} & y_{n-2} & \dots & y_{\frac{n}{2}} & | & y_{\frac{n}{2}-1} & \dots & y_1 & y_0
\end{array}$$

Im Folgenden gelte $x_{hi} = (x_{n-1}, x_{n-2}, \ldots, x_{\frac{n}{2}})$ und $x_{low} = (x_{\frac{n}{2}-1}, \ldots, x_1, x_0)$. Entsprechend sei y_{hi} der Teil der $\frac{n}{2}$ höherwertigen Bits von y und y_{low} der Teil der $\frac{n}{2}$ niederwertigen Bits. Wir wollen also $x_{hi} + y_{hi}$ und $x_{low} + y_{low}$ getrennt berechnen und daraus das Gesamtergebnis berechnen. Wenn wir $x_{low} + y_{low}$ rekursiv korrekt berechnen, bekommen wir damit schon die korrekten Summenbits $s_{\frac{n}{2}-1}, \ldots, s_0$ sowie das Übertragsbit $c_{\frac{n}{2}-1}$, welches sich auf die Addition von $x_{\frac{n}{2}}$ und $y_{\frac{n}{2}}$ bzw. auf die ganze Berechnung von x_{hi} und y_{hi} auswirken kann. In einem Fall ist das Übertragsbit 0 und im anderen 1. Weitere Fälle müssen auf dieser Stufe nicht unterschieden werden. Die Idee des Conditional Sum Addierers ist es nun, rekursiv die Summe von x_{hi} und y_{hi} zu berechnen, wobei in einem Fall angenommen wird, dass ein Übertragsbit von rechts reinkommt (also im obigen Beispiel $c_{\frac{n}{2}-1} = 1$), und im anderen Fall angenommen wird, dass keines reinkommt ($c_{\frac{n}{2}-1} = 0$). Wenn wir diese beiden Summen korrekt berechnet haben, können wir dann über das Übertragsbit $c_{\frac{n}{2}-1}$ entscheiden, welche Summe wir für das Endergebnis auswählen. Dazu benutzen wir Select-Bausteine, die die Funktion *sel* realisieren.

Die beiden Teilsummen sollen rekursiv berechnet werden, also müssen wir uns überlegen, auf welcher Stufe wir abbrechen können. Es liegt nahe, dazu die Teilaufgaben der Größe $n = 1$ zu wählen. Für diese können wir das Ergebnis ähnlich wie in Schritt 1 von Algorithmus 8 berechnen. Wir müssen also die Summe der Bits x_i und y_i einmal unter der Bedingung berechnen, dass der Übertrag $c_{i-1} = 0$ ist und einmal unter der Bedingung, dass $c_{i-1} = 1$ ist. Im Fall $c_{i-1} = 0$ berechnen wir:

$$c_i = x_i \wedge y_i \qquad \text{und} \qquad s_i = x_i \oplus y_i$$

Entsprechend berechnen wir im Fall $c_{i-1} = 1$:

$$c_i = x_i \vee y_i \qquad \text{und} \qquad s_i = x_i \oplus y_i \oplus 1$$

Es sieht nun so aus, als ob wir auf dieser Stufe 5 Bausteine bräuchten. Wenn wir etwas genauer hinschauen, sehen wir jedoch, dass wir mit 4 B_2-Bausteinen auskommen. Wir überlegen uns, wann $s_i = x_i \oplus y_i \oplus 1$ den Wert 1 annimmt. Dies ist der Fall, wenn x_i die gleiche Belegung wie y_i hat, also $x_i \equiv y_i$ gilt. Diese Funktion ist jedoch auch eine der grundlegenden Funktionen aus B_2, d.h. es gibt einen B_2-Baustein, der die Funktion $\equiv$ realisiert. Somit kommen wir im Fall $n = 1$ mit Tiefe $D(1) = 1$ und Größe $C(1) = 4$ aus.

Frage 3.11: Welche Größe und welche Tiefe hat der Conditional Sum Addierer?

Wir erreichen Stufe 0 für alle n Eingabebits, daher benötigen wir für diese Stufe insgesamt $4n$ Bausteine. Stufe 0 ist auch die einzige Stufe, auf der „wirkliche“ Additionsarbeit geleistet wird. Die restlichen (Teil-)Summen ergeben sich nur noch über die entsprechenden Selektionen mit parallel geschalteten Select-Bausteinen. Daher hängt die restliche Größe nur noch von der Anzahl der Select-Bausteine ab. Wir überlegen uns nun, wieviele Select-Bausteine wir pro Ebene benötigen und addieren diese Anzahlen. Betrachten wir dazu zuerst exemplarisch die Blöcke von (x_3, y_3) und (x_2, y_2) auf Stufe 1. Mit der beiden Select-Eingaben des Blocks von (x_2, y_2) müssen die richtigen zwei Bits des benachbarten Blocks (x_3, y_3) selektiert werden. Insgesamt gibt es $\frac{n}{2} = \frac{n}{2^1} = 2^{k-1}$ solcher Blöcke auf Stufe 1. Da sich die Anzahl der Blöcke in jedem Schritt halbiert, halbiert sich auch mit jeder Stufe die Anzahl der Blöcke, die das richtige Ergebnis aus ihrem Nachbarblock auswählen. Dafür verdoppelt sich grob gesehen die Anzahl der auszuwählenden Bits. Allgemein gibt es auf Stufe i also 2^{k-i} Blöcke, die das richtige Ergebnis auswählen, wobei jeder Block zwei Select-Eingaben hat. Die Länge des auszuwählenden Ergebnis ist dabei $2^{i-1} + 1$ (Summenbits plus Übertrag). Damit gibt es auf

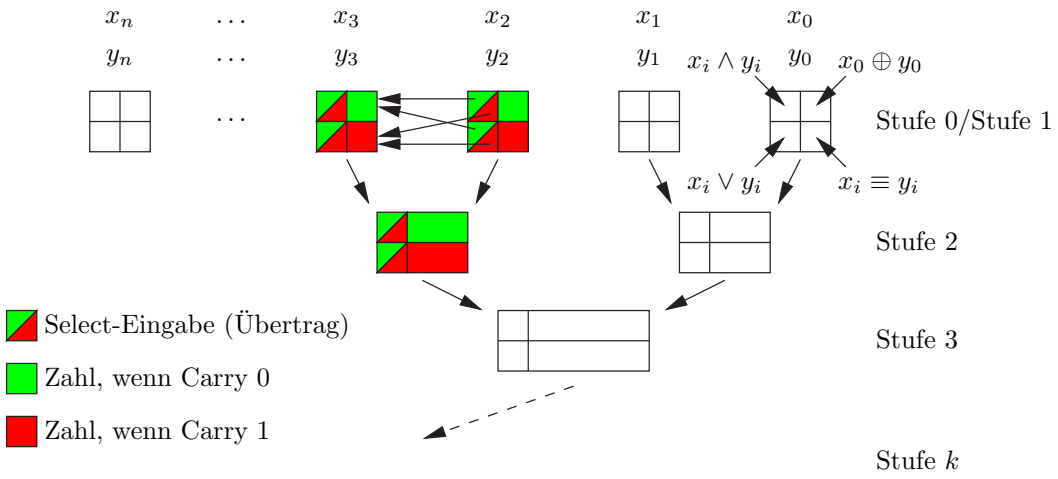


Abbildung 8: Rekursionsbaum des Conditional Sum Addierers

Ebene i also $2 \cdot 2^{k-i} \cdot (2^{i-1}+1)$ Select-Bausteine. Da der Rekursionsbaum die Tiefe k hat, ergibt sich die folgenden Gesamtanzahl an Select-Bausteinen:

$$\begin{aligned}
\sum_{i=1}^{k}(2^{i-1}+1)\cdot 2\cdot 2^{k-i} &= \sum_{i=1}^{k}(2^{i-1}+1)\cdot 2^{k-i+1} = \sum_{i=1}^{k}\left(2^{k-i+1+i-1}+2^{k-i+1}\right)\\
&= \sum_{i=1}^{k}2^k + \sum_{i=1}^{k}2^{k-i+1} = k\cdot 2^k + \sum_{i=1}^{k}2^i = k\cdot 2^k + \sum_{i=0}^{k-1}2^{i+1}\\
&= k\cdot 2^k + 2\cdot\sum_{i=0}^{k-1}2^i = k\cdot 2^k + 2\cdot(2^k-1)\\
&= k\cdot 2^k + 2\cdot 2^k - 2
\end{aligned}$$

Es gilt $n = 2^k$ und somit $k = \log n$. Wenn wir dies oben einsetzen ergibt sich also die folgende Anzahl an Select-Bausteinen in Abhängigkeit von n:

$$n \cdot \log n + 2n - 2$$

Die Größe von Stufe 0 haben wir schon mit $4n$ berechnet. Da ein Select-Baustein die Größe 3 hat, ergibt sich folgende Gesamtgröße:

$$3 \cdot (n \cdot \log n + 2n - 2) + 4n = 3n\log n + 6n - 6 + 4n = 3n\log n + 10n - 6$$

Die Tiefe lässt sich nun schnell abschätzen. Stufe 0 kann in Tiefe 1 realisiert werden. Wir haben auch schon gesehen, dass der Rekursionsbaum k Ebenen von parallelen Select-Bausteinen hat, wobei ein Select-Baustein die Tiefe 2 hat. Insgesamt ergibt sich somit folgende Tiefe:

$$1 + 2\log n$$

3.5 Optimale Präfixberechnung

Frage 3.12: Worin besteht das Präfixproblem?

Beim Präfixproblem ist eine Menge M gegeben, auf der eine assoziative, binäre Verknüpfung $\circ$ definiert ist. Gesucht ist nun der Wert aller p_i für $0 \leq i \leq n-1$ mit:

$$p_i = x_i \circ \cdots \circ x_0$$

Wir werden das Präfixproblem beim Ladner/Fisher Addierer benutzen, um effizient zu addieren. Wir können die weitere Betrachtung hier jedoch auch durch folgendes Problem motivieren. Gegeben ist ein Bitverktor $x = (x_{n-1}, \ldots, x_0)$. Wir wollen nun für jede Position i wissen, ob die Anzahl der Einsen im von Position 0 bis i gerade oder ungerade ist. Dies kann berechnet werden, indem wir $x_i \oplus \cdots \oplus x_0$ berechnen. Ist dieser Wert 1, so ist die Anzahl ungerade, ansonsten ist sie gerade. Die Verknüpfung $\oplus$ ist binär und assoziativ. Wenn wir also das Präfixproblem effizient lösen können, so können wir auch dieses Problem effizient lösen.

Frage 3.13: Gebe einen naiven Algorithmus zur Lösung des Präfixproblems an!

Ein naiver Algorithmus würde mit der Lösung $p_0 = x_0$ beginnen und dann sukzessive p_i als $p_i = x_i \circ p_{i-1}$ berechnen. Es ist offensichtlich, dass dieser Algorithmus linere Größe und Tiefe hat. Es werden insgesamt $n-1$ $\circ$-Bausteine benötigt, die hintereinander geschaltet werden.

Frage 3.14: Welche drei effizienten Algorithmen zur Lösung des Präfixproblems kennst du?

Es werden nun zwei Algorithmen vorgestellt, die das Präfixproblem effizient lösen. Danach wird eine Kombination dieser beiden Algorithmen vorgestellt, die von den drei Lösungen am effizientesten ist.

Algorithmus 1: Die Idee dieses Algorithmus wird in Abbildung 9 dargestellt. Der Algorithmus löst das Problem, indem er die Eingabe, so lange dies möglich ist, rekursiv in zwei Teile zerteilt und für diese Teilprobleme das Präfixproblem löst. Die Lösungen dieser beiden Teilprobleme werden dann kombiniert, um die Lösung für das Gesamtproblem zu bekommen. Konkret bedeutet dies, dass eine Eingabe $x_{n-1}, \ldots, x_0$ in die zwei Teile $x_{n-1}, \ldots, x_{\frac{n}{2}}$ und $x_{\frac{n}{2}-1}, \ldots, x_0$ zerlegt wird und für diese beiden Teile das Präfixproblem gelöst wird. Wir überlegen uns nun, wie wir aus der korrekten Lösung für diese beiden Teilprobleme die korrekte Lösung für das Gesamtproblem bekommen. Für $x_{\frac{n}{2}-1}, \ldots, x_0$ liegen nach diesem Schritt sofort die korrekten Ergebnisse $p_{\frac{n}{2}-1}, \ldots, p_0$ vor, lediglich die Ergebnisse für $x_{n-1}, \ldots, x_{\frac{n}{2}}$ müssen noch mit Hilfe des Ergebnis für $x_{\frac{n}{2}-1}, \ldots, x_0$ berechnet werden. Die p_i der vorderen $\frac{n}{2}$ Eingabebits ergeben sich dann als $p_i \circ p_{\frac{n}{2}-1}$.

Wir kommen nun zur Abschätzung der Größe und der Tiefe. Die Größe des Schaltkreises ergibt sich durch die folgende rekursive Formel: $C(n) = 2 \cdot C(\frac{n}{2}) + \frac{n}{2}$ mit $C(1) = 0$. Wir lösen diese Formel mit $n = 2^k$ auf und erhalten:

$$\begin{aligned} C(2^k) &= 2 \cdot C(2^{k-1}) + 2^{k-1} = 2 \cdot (2 \cdot C(2^{k-2}) + 2^{k-2}) + 2^{k-1} \\ &= 2^2 \cdot C(2^{k-2}) + 2 \cdot 2^{k-1} = 2^2 \cdot (2 \cdot C(2^{k-3}) + 2^{k-3}) + 2 \cdot 2^{k-1} \\ &= 2^3 \cdot C(2^{k-3}) + 3 \cdot 2^{k-1} = 2^i \cdot C(2^{k-i}) + i \cdot 2^{k-1} \end{aligned}$$

Wir setzen $i = k$ damit wir $C(2^{k-i}) = C(2^{k-k}) = C(1) = 0$ nutzen können und erhalten:

$$C(2^k) = 2^k \cdot C(1) + k \cdot 2^{k-1} = k \cdot 2^{k-1}$$

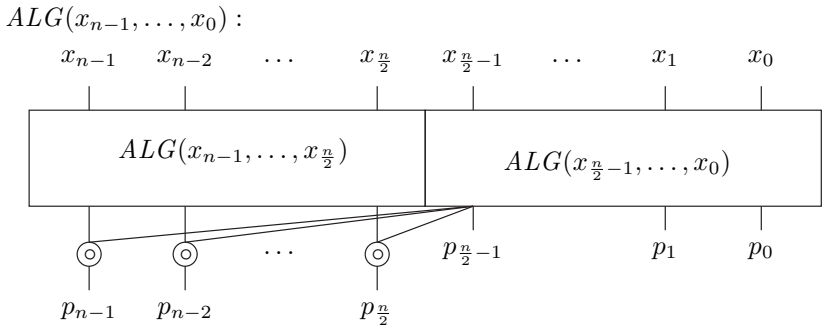


Abbildung 9: Algorithmus 1 zur Lösung des Präfixproblems

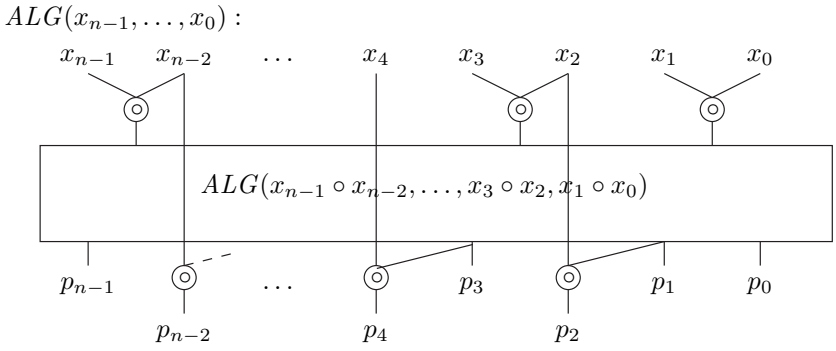


Abbildung 10: Algorithmus 2 zur Lösung des Präfixproblems

Es gilt somit $C(n) = \log n \cdot \frac{n}{2} = \frac{1}{2} n \log n$. Für die Tiefe gilt die Rekursionsformel $D(n) = D(\frac{n}{2}) + 1$ mit $D(1) = 0$. Wir lösen diese Formel wie folgt auf:

$$D(2^k) = D(2^{k-1}) + 1 = D(2^{k-2}) + 1 + 1 = D(2^{k-2}) + 2 \cdot 1 = \cdots = D(2^{k-i}) + i \cdot 1$$

Auch hier setzen wir $i = k$ und erhalten:

$$D(2^k) = D(2^{k-k}) + k \cdot 1 = D(1) + k = k$$

Damit gilt $D(n) = \log n$.

Algorithmus 2: Algorithmus 2 ist in Abbildung 10 dargestellt. Auf der ersten Stufe werden parallel die $\frac{n}{2}$ Paare $x_{n-1} \circ x_{n-2}, \ldots, x_1 \circ x_0$ berechnet. Die $\frac{n}{2}$ Ergebnis dieser Berechnung werden als Eingabe einer rekursiven Berechnung mit diesem Algorithmus benutzt. Auf diese Art und Weise werden alle p_i für die ungerade i berechnet. Wenn wir diese Ergebnisse für die ungeraden p_i haben, können wir aus diesen auf einfach Art und Weise die Ergebnisse für die geraden p_i berechnen. Es gilt $p_{i+1} = x_{i+1} \circ p_i$, mit $i = 2k - 1$ mit $1 \leq k \leq \frac{n}{2} - 1$.

Wir kommen nun zur Abschätzung der Größe $C(n)$. Die Berechnung für n Elemente benötigt $\frac{n}{2}$ Bausteine für die Berechnungen, die vor dem rekursiven Aufruf gemacht werden, und $\frac{n}{2} - 1$ Bausteine für die Berechnungen nach dem rekursiven Aufruf (der Baustein für das gerade p_0 fehlt, da wir p_0 wegen $p_0 = x_0$ „geschenkt“ bekommen). Der rekursive Aufruf benötigt $C(\frac{n}{2})$ Bausteine. Das Ergebnis für 2 Elemente können wir mit

einem Baustein berechnen, d.h. es gilt $C(2) = 1$. Insgesamt gilt also:

$$C(n) = C\left(\frac{n}{2}\right) + \frac{n}{2} + \frac{n}{2} - 1 = C\left(\frac{n}{2}\right) + n - 1$$

Wir lösen diese Formel wieder für $n = 2^k$ auf. Im übrigen gilt auch $C(1) = 0$.

$$\begin{aligned} C(2^k) &= C(2^{k-1}) + 2^k - 1 = C(2^{k-2}) + 2^{k-1} - 1 + 2^k - 1 \\ &= C(2^{k-2}) + 2^{k-1} + 2^k - 2 = C(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k - 3 \\ &= \ldots = C(2^{k-i}) + \sum_{j=0}^{i-1} 2^{k-j} - i \end{aligned}$$

Wir setzen $i = k$, um $C(1) = 0$ ausnutzen zu können und erhalten:

$$\begin{aligned} C(2^k) &= C(2^{k-k}) + \sum_{j=0}^{k-1} 2^{k-j} - k = C(1) + \sum_{j=1}^{k} 2^j - k = 2 \cdot \sum_{j=0}^{k-1} 2^j - k \\ &= 2 \cdot (2^k - 1) - k = 2 \cdot 2^k - k - 2 \end{aligned}$$

Damit ist also $C(n) = 2n - \log n - 2$. Im Vergleich zu Algorithmus 1 hat Algorithmus 2 also eine geringere Größe. Wir kommen nun zur Abschätzung der Tiefe. Vor und nach der Rekursion haben wir jeweils eine Ebene von Bausteinen. Dazu kommt noch die Anzahl der Bausteine, die wir zur Lösung von Problemen mit $\frac{n}{2}$ Eingaben benötigen. Die Rekursionsformel für die Tiefe lautet damit:

$$D(n) = D\left(\frac{n}{2}\right) + 2$$

Zur Lösungen eines Problems der Größe 2 benötigen wir nur einen $\circ$-Baustein. Damit gilt $D(2) = 1$. Wir lösen $D(n)$ mit $n = 2^k$ auf:

$$D(2^k) = D(2^{k-1}) + 2 = D(2^{k-2}) + 2 + 2 = D(2^{k-3}) + 3 \cdot 2 = D(2^{k-i}) + i \cdot 2$$

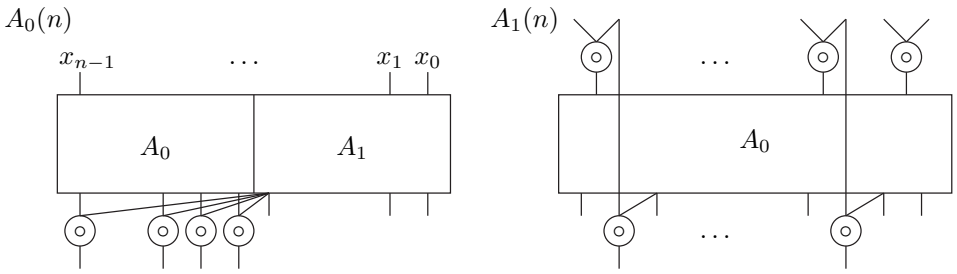
Wir setzen $i = k - 1$, um $D(2) = 1$ ausnutzen zu können:

$$D(2^k) = D(2^{k-(k-1)}) + (k-1) \cdot 2 = D(2) + 2 \cdot (k-1) = 1 + 2k - 2 = 2k - 1$$

Es gilt somit $D(n) = 2 \log n - 1$. Die Tiefe dieses Algorithmus ist damit leider ungefähr doppelt so groß wie die Tiefe von Algorithmus 1.

Algorithmus 3: Wir haben gerade gesehen, dass Algorithmus 1 eine gute Tiefe hat, während Algorithmus 2 eine gute Größe aufweist. Es wäre nun schön, einen Algorithmus zu haben, der beide positiven Eigenschaften vereint. Ladner und Fisher haben dies erreicht, indem sie Algorithmus 1 und Algorithmus 2 „gemischt“ haben. Dies geschieht wie folgt:

Algorithmus 0: Wende parallel $A_1\left(\frac{n}{2}\right)$ auf die Elemente $x_{\frac{n}{2}-1}, \ldots, x_0$ und $A_0\left(\frac{n}{2}\right)$ auf die Elemente $x_{n-1}, \ldots, x_{\frac{n}{2}}$ an. Die fehlenden Präfixe werden in einem Schritt berechnet (wie etwas weiter oben in Algorithmus 1). Abbildung 11(a) skizziert die Idee.



(a) $A_0(n)$ (b) $A_1(n)$

Abbildung 11: Algorithmus 3 zur Lösung des Präfixproblems

Algorithmus 1: Berechne parallel die $\frac{n}{2}$ Paare $x_{n-1} \circ x_{n-2}, \ldots, x_1 \circ x_0$. Wende auf diese $\frac{n}{2}$ Paare Algorithmus 0 an. Die fehlenden Präfixe werden in einem Schritt berechnet (wie etwas weiter oben in Algorithmus 2). Das Vorgehen dieses Algorithmus zeigt Abbildung 11(b).

Der „Haupttrick“ besteht bei diesem Algorithmus darin, dass das vorderste Bit, welches A_1 berechnet, in einer Tiefe berechnet wird, die um eins kleiner ist als die Gesamttiefe von A_1. Damit kann das vorderste Bit also relativ früh als Eingabe für die Ergebnisse von A_0 verwendet werden kann. Wir werden sehen, dass dieser eine Takt „früh genug“ ist und wir durch diesen Trick auf logarithmische Tiefe bei linearer Größe kommen.

Wir kommen nun zur Abschätzung der Größe und der Tiefe. Mit C_0 bezeichnen wir im Folgenden die Größe des Schaltkreises für A_0 und mit C_1 die Größe von A_1. Es gilt $C_0(1) = C_1(1) = 0$ und $C_0(2) = C_1(2) = 1$. Für $n \geq 4$ können wir die folgenden Rekursionsformeln angeben:

$$\begin{aligned} C_0(n) &= C_0\left(\frac{n}{2}\right) + C_1\left(\frac{n}{2}\right) + \frac{n}{2} \\ C_1(n) &= C_0\left(\frac{n}{2}\right) + n - 1 \end{aligned}$$

Da wir an $C_0(n)$ interessiert sind (damit starten wir ja), ersetzen wir in der Formel für $C_0(n)$ den Term $C_1(\frac{n}{2})$ durch den entsprechenden Wert:

$$C_0(n) = C_0\left(\frac{n}{2}\right) + C_0\left(\frac{n}{4}\right) + \frac{n}{2} - 1 + \frac{n}{2} = C_0\left(\frac{n}{2}\right) + C_0\left(\frac{n}{4}\right) + n - 1$$

Wir zeigen nun per Induktion, dass $C_0(n) < 4n$ gilt. Für $C_0(1) = 0 < 4$ bzw. $C_0(2) = 1 < 8$ gilt die Behauptung. Nach Induktionannahme gelte die Behauptung für alle $C(k)$ mit $k < n$. Dann schätzen wir ab:

$$\begin{aligned} C_0(n) &= C_0\left(\frac{n}{2}\right) + C_0\left(\frac{n}{4}\right) + n - 1 < 4 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + n - 1 \\ &< 2n + n + n - 1 = 4n - 1 < 4n \end{aligned}$$

Damit erreichen wir mit Algorithmus 3 also lineare Größe. Die Abschätzung der Tiefe ist etwas komplizierter und wird hier auch nicht komplett durchgeführt. Wir sind an

$D_0(n)$ interessiert und es kann gezeigt werden, dass folgende Gleichung gilt:

$$D_0(n) = \max\left\{D_0\left(\frac{n}{4}\right) + 2, D_0\left(\frac{n}{2}\right) + 1\right\}$$

Da $D_0(1) = 0$ und $D_0(2) = 1$ gilt, kann leicht gezeigt werden, dass $D_0(n) = \log n$ ist. Die Behauptung gilt offensichtlich für den Induktionsanfang $n = 0$. Wir nehmen nun an, dass sie für $k < n$ ebenfalls gelten. Dann gilt für $D_0(n)$:

$$\begin{aligned} D_0(n) &= \max\left\{D_0\left(\frac{n}{4}\right) + 2, D_0\left(\frac{n}{2}\right) + 1\right\} = \max\left\{\log\left(\frac{n}{4}\right) + 2, \log\left(\frac{n}{2}\right) + 1\right\} \\ &= \max\{\log n - \log 4 + 2, \log n - \log 2 + 1\} \\ &= \max\{\log n - 2 + 2, \log n - 1 + 1\} \\ &= \max\{\log n, \log n\} \\ &= \log n \end{aligned}$$

Somit löst Algorithmus 3 das Präfixproblem also in linearer Größe und logarithmischer Tiefe.

3.6 Ladner/Fisher Addierer

Frage 3.15: Was ist die Hauptidee beim Ladner/Fisher Addierer?

In Kapitel 3.5 haben wir Algorithmen kennengelernt, mit denen sich das Präfixproblem effizient lösen lässt. Wir werden gleich sehen, dass sich das Problem der Berechnung der Carrybits ebenfalls als Präfixproblem modellieren lässt. Wenn die Bausteine zur Berechnung der verwendeten assoziativen, binären Verknüpfung nicht zu groß oder zu tief werden, können wir die Carrybits dann mit Hilfe von Algorithmus 3 aus Kapitel 3.5 effizient berechnen.

Wie können wir die Berechnung der Carrybits als Präfixproblem beschreiben? Wir betrachten noch einmal die drei verschiedenen Charakterisierungen der Eingabebits x_i und y_i. In Kapitel 3.3 haben wir die verschiedenen Bitpaare in die Klassen Eliminate, Propagate und Generate eingeteilt. Wir werden diese Klassen nun zuerst einmal als Funktionen auffassen, die aus dem Carrybit der vorherigen Stelle das Carrybit der aktuell betrachteten Stelle berechnen. Wenn c das Carrybit der vorherigen Stelle ist, so gilt für das Carrybit der betrachteten Stelle:

Eliminate: $E(c) = 0$. Das Carry der vorherigen Stelle wird eliminiert.

Propagate: $P(c) = c$. Das Carry der vorherigen Stelle wird weitergegeben.

Generate: $G(c) = 1$. Ein Carry wird unabhängig von der vorherigen Stelle erzeugt.

Es sei nun $A_i \in \{E, P, G\}$ die Charakterisierung der Bits an der Stelle i und c_i das Carrybit an der Stelle i. Dann ergibt sich c_i wie folgt, wobei $c_{-1} = 0$ gilt:

$$c_i = A_i(c_{i-1}) = A_i(A_{i-1}(c_{i-2})) = A_i \circ A_{i-1}(c_{i-2}) = A_i \circ \cdots \circ A_0(0)$$

Dabei ist $\circ$ die Hintereinanderausführung von Funktionen, welche stets assoziativ ist. Die Menge $\{E, P, G\}$ bildet mit $\circ$ als Operation eine Halbgruppe, deren Verknüpfungstabelle in Tabelle 6 dargestellt ist. Die Halbgruppeneigenschaft bedeutet, dass $\{E, P, G\}$ bezüglich $\circ$

$\circ$	E_{i-1}	P_{i-1}	G_{i-1}
E_i	E	E	E
P_i	E	P	G
G_i	G	G	G

$\Rightarrow$

$\circ$	$(0,0)$	$(0,1)$	$(1,0)$
$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
$(0,1)$	$(0,0)$	$(0,1)$	$(1,0)$
$(1,0)$	$(1,0)$	$(1,0)$	$(1,0)$

Tabelle 6: Verknüpfungstabelle von $(\{E,P,G\},\circ)$ und ihre Kodierung

assoziativ ist und dass jede $\circ$-Verknüpfung von zwei Elementen aus $\{E,P,G\}$ wieder ein Element aus $\{E,P,G\}$ ergibt. Dabei bedeutet z.B. $E_i \circ P_{i-1} = E$, dass anstatt der Operation $E_i(P_{i-1}(x)) = E_i \circ P_{i-1}(x)$ auch gleich die Funktion $E(x)$ berechnet werden kann. Es kann also allgemein für einige mit $\circ$ verknüpften Funktionen berechnet werden, welche Funktion wir direkt benutzen können, um dasselbe Ergebnis zu bekommen. Die drei Zeilen der Tabelle machen recht schnell deutlich, dass E die Rolle einer Linksnull spielt, während P die Identität ist und G eine Linkseins darstellt. Wenn wir für $A_{n-1},\ldots,A_0$ und $\circ$ das Präfixproblem lösen, dann ergibt sich c_i also als $c_i = B_i(0)$, wenn $B_i = A_i \circ \cdots \circ A_0$ gilt.

Nachdem die Grundidee bzw. das Ziel nun klar sein sollte, müssen wir uns nun überlegen, wie wir E, P und G kodieren und wie wir die $\circ$-Verknüpfung mit diesen Kodierungen realisieren. Die rechte Seite von Tabelle 6 zeigt die entsprechenden Kodierungen von E,P und G anhand ihrer u- und v-Werte. Wir wollen uns anhand dieser Kodierungen nun überlegen, wie sich $(u,v) = (u_i,v_i) \circ (u_{i-1},v_{i-1})$ ergibt. Dabei nutzen wir aus, dass $u = 1$ bzw. $v = 1$ nur jeweils für eine Zeile bzw. Spalte gilt. Wann nimmt u nun in Tabelle 6 den Wert 1 an? Dies in der zweiten Zeile in der letzten Spalte der Fall sowie in der kompletten letzten Zeile. Damit gilt also:

$$u = v_i u_{i-1} \vee u_i$$

Der Wert von v nimmt nur für $P_i \circ P_{i-1}$, also für $(0,1)\circ(0,1)$, den Wert 1 an. Dann gilt also:

$$v = v_i v_{i-1}$$

Wenn die u- und v-Werte berechnet sind, kann die $\circ$-Operation also mit 3 Bausteinen in Tiefe 2 realisiert werden.

Es sei nun A'_i die an Stelle i berechnete Funktion und (u'_i,v'_i) ihre Kodierung. Wir sind dann am Ergebnis von $A'_i(0)$ interessiert. Es gilt $E(0) = P(0) = 0$ sowie $G(0) = 1$. Damit entspricht der Wert des Carrybit an Stelle i jedoch dem Wert u'_i. Denn es gilt $u'_i = 0$, wenn A'_i die Funktion E oder P ist, und $u'_i = 1$, wenn A'_i die Funktion G ist.

Frage 3.16: Welche Größe bzw. Tiefe hat der Ladner/Fisher-Addierer?

Das Vorgehen des Ladner/Fisher Addierers ist das in Algorithmus 8 (siehe Seite 56) beschriebene. Wir überlegen uns zuerst, wieviele Bausteine und welche Tiefe wir für den ersten und dritten Schritt benötigen. Im ersten Schritt berechnen wir für n Paare von Eingabebits die Werte $u_i = x_i \wedge v_i$ und $v_i = x_i \oplus y_i$. Wir benötigen für diesen Schritt also $2n$ Bausteine und Tiefe 1. Im dritten Schritt berechnen wir $n-1$ Stellen als $s_i = v_i \oplus c_{i-1}$. In diesem Schritt benötigen wir somit $n-1$ Bausteine und ebenfalls Tiefe 1. Insgesamt benötigen Schritt 1 und 3 also $3n-1$ Bausteine und Tiefe 2.

Um die Carrybits mit Hilfe des Präfixalgorithmus zu berechnen, benötigen wir $C_0(n)$ $\circ$-Bausteine, welche in Tiefe $\log n$ das Präfixproblem lösen. Unsere Implementierung des $\circ$-Bausteins hat Größe 3 und Tiefe 2. Es ergibt sich also eine Größe von $3 \cdot C_0(n) < 3 \cdot 4n = 12n$

Algorithmus 9 Die Schulmethode der Multiplikation zweier Zahlen $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$

1. Berechne $z_i = (x_{n-1} \wedge y_i, \ldots, x_0 \wedge y_i)$ für $0 \leq i \leq n-1$.
2. Addiere die Zahlen $|z_i|2^i$ für $0 \leq i \leq n-1$.

und eine Tiefe von $2\log n$. Die Gesamtgröße eines Ladner/Fisher Addierers beträgt somit $C(n) \leq 12n + 3n - 1 = 15n - 1$ und die Gesamttiefe $2\log n + 2$.

3.7 Schulmethode für die Multiplikation

Frage 3.17: Wie ist die Multiplikation definiert?

Die Multiplikation ist die Folge der Booleschen Funktionen $MUL_n \in B_{2n,2n}$, die aus den beiden Zahlen $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$ die Binärdarstellung des Produkts von x und y berechnen.

Frage 3.18: Wie funktioniert die Schulmethode der Multiplikation?

Die Schulmethode der Multiplikation zweier Binärzahlen funktioniert im Prinzip genauso wie die Schulmethode zur Multiplikation zweier Dezimalzahlen und wird in Algorithmus 9 beschrieben. Dabei entspricht Schritt 1 der Multiplikation einer jeden Stelle von y mit x, wobei hier jedoch nur mit 0 oder 1 multipliziert wird. Das Ergebnis einer jeden solchen Multiplikation ist also entweder die Zahl x selbst oder 0. Damit wird die Zahl z_i also korrekt berechnet. In Schritt 2 wird dann die Summe dieser Ergebnisse gebildet, wobei die Zahl z_i vorher mit 2^i multipliziert wird. Dies entspricht einem Shift der Binärdarstellung von z_i um i Stellen nach links. Bei der Schulmethode für Dezimalzahlen multiplizieren wir die Ergebnisse nicht direkt mit 10^i, schreiben aber jedes Ergebnis einer Einzelmultiplikation um eine Stelle nach links verschoben unter das vorhergehende Ergebnis, wodurch wir denselben Effekt erzielen.

Zur schnellen Addition der n Zahlen $|z_i|2^i$ können wir Ladner/Fisher Addierer benutzen, die baumartig angeordnet sind. Es werden also in einem Schritt parallel mehrere (verschiedene) Paare von Zahlen addiert und mit den Ergebnissen dasselbe bemacht. Diese Struktur sieht also so ähnlich aus wie in Abbildung 7 auf Seite 58, nur dass statt den $\wedge$-Bausteinen Ladner/Fisher Addierer benutzt werden und statt den Eingaben x_i die Zahlen $|z_i|2^i$ benutzt werden. Damit brauchen wir also höchstens $\lceil \log n \rceil$ Ebenen von Ladner/Fisher Addierern.

Frage 3.19: Welche Größe und Tiefe hat der Schulmethode der Multiplikation?

Schritt 1 von Algorithmus 9 kann in Tiefe 1 bewerkstelligt werden. Die Addition der in diesem Schritt berechneten Zwischenergebnisse benötigt $n-1$ Ladner/Fisher Addierer. Denn jede Addition von zwei Zwischenergebnissen reduziert die Anzahl der Zwischenergebnisse um eins. Nach $n-1$ Additionen bleibt also nur noch ein Ergebnis über. Ein Ladner/Fisher Addierer hat eine lineare Größe von $O(n)$. Wenn wir $n-1$ solche Addierer benutzen, hat der resultierende Schaltkreis somit eine Größe von $O(n^2)$.

Kommen wir nun zur Abschätzung der Tiefe. Wenn wir die Ladner/Fisher Addierer baumartig anordnen, benötigen wir höchsten $\lceil \log n \rceil$ Ebenen. Ein Ladner/Fisher Addierer hat die Tiefe $2\log n + 2 = O(\log n)$. Somit beträgt die Gesamttiefe also $O(\log n \cdot \log n) = O(\log^2 n)$.

Frage 3.20: Wie kann die Schulmethode der Multiplikation in geringerer Tiefe durchgeführt werden?

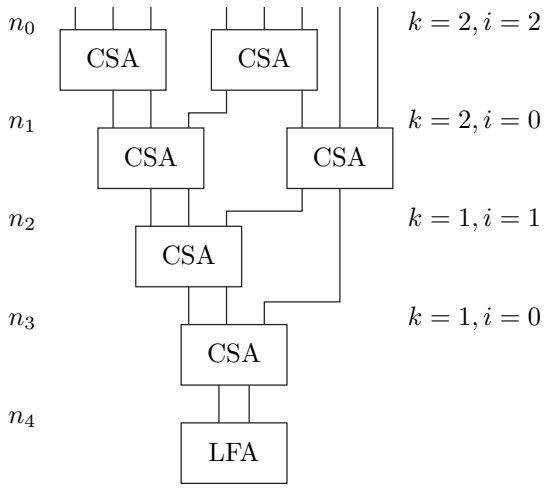


Abbildung 12: Multiplikation mit einem Wallace-Tree von CSAs und einem LFA

Bei der bisher vorgestellten Schulmethode der Multiplikation wird die Anzahl der Summanden reduziert, indem jeweils zwei Zahlen addiert werden, um eine neue Zahl zu bekommen. Jede dieser Additionen hat dabei eine Tiefe von $2 \cdot \log(2n) + 2 = O(\log n)$, was bei einer baumartigen Anordnung der Addierer zu einer Tiefe von $O(\log^2 n)$ führt. Die im Folgenden vorgestellte Methode reduziert die Anzahl der Summanden in einer geringeren Tiefe, ohne dabei das Ergebnis zu verfälschen. Dabei wird ein ähnlicher „Trick“ wie beim von Neumann Addiererwerk verwendet. Statt zwei Zahlen zu addieren, nehmen wir nun drei Zahlen x, y und z und erzeugen aus diesen zwei neue Zahlen u und v, deren Summe mit der von x, y und z übereinstimmt. Es gilt also $|x| + |y| + |z| = |u| + |v|$. Der Baustein, der dies realisiert, ist ein sogenannter Carry-Save-Adder (CSA). Ein Carry-Save-Adder, der drei l Bit lange Zahlen zu zwei $l + 1$ Bit lange Zahlen addiert besteht aus l parallel geschalteten Fulladdern. Jeder Fulladder berechnet für die Eingabebits x_i, y_i und z_i das Carrybit u_i und das Summenbit v_i. Wenn wir diese beiden Werte für alle Tripel (x_i, y_i, z_i) berechnen, dann haben die Zahlen $u = (u_{l-1}, \ldots, u_0, 0)$ und $v = (v_{l-1}, \ldots, v_0)$ die gewünschte Eigenschaft. Ein solcher Carry-Save-Adder besteht also aus $5l$ Bausteinen und hat die Tiefe 3.

Um die n Zwischenergebnisse der Multiplikation zu addieren, ordnen wir die Carry-Save-Adder also wieder baumartig an, wobei mit jedem Carry-Save-Adder ein Summand reduziert wird. Dies machen wir so lange, bis nur noch zwei Summanden übrig sind, welche zum Schluss mit einem Ladner/Fisher Addierer summiert werden, um das Endergebnis zu bekommen. Die bei diesem Vorgehen entstehende Baumstruktur wird auch Wallace-Tree genannt und ist in Abbildung 12 dargestellt.

Frage 3.21: Welche Tiefe hat ein Wallace-Tree?

Wir wollen nun die Tiefe eines Wallace-Trees ermitteln. Dazu überlegen wir uns zuerst, wieviele Summanden auf einer Stufe im Wallace-Tree höchstens noch vorhanden sein können. Anhand dieser Formel ermitteln wir dann, wieviele Stufen mindestens benötigt werden, bis nur noch zwei Summanden vorhanden sind, die dann mit einem Ladner/Fisher Addierer addiert werden. Die Anzahl der Summanden auf der j-ten Stufe bezeichen wir mit n_j, wobei n_0 die allererste Stufe darstellt, also die Stufe auf der $n_0 = n$ gilt.

Zuerst überlegen wir uns nun, in welchem Maß die Anzahl der Summanden abnimmt. Dazu

gehen wir davon aus, dass auf der $j-1$-ten Stufe $n_{j-1} = 3k + i$ Summanden vorhanden sind, wobei $k \geq 0$ und $0 \leq i \leq 2$ gelte. Aus den „ersten“ $3k$ Summanden können wir mit k Carry-Save-Addern $2k$ neue Summanden erzeugen. Die restlichen $0 \leq i \leq 2$ Summanden reichen nicht mehr aus, um einen weiteren Carry-Save-Adder komplett zu füllen. Wir machen somit aus $n_{j-1} = 3k + i$ Summanden $n_j = 2k + i$ Summanden auf Stufe j. Wir wollen nun die Anzahl der Summanden auf der j-ten Stufe in Abhängigkeit von $n_0 = n$ ausdrücken. Dazu drücken wir n_j erst in Abhängigkeit von n_{j-1} aus und „hangeln uns dann langsam nach oben“. Es gelte also $n_j = 2k + i$ mit $0 \leq i \leq 2$. Dann formen wir dies wie folgt um:

$$n_j = 2k + i = \underbrace{\frac{2}{3} \cdot 3k}_{2k} + \underbrace{\frac{2}{3} \cdot i + \frac{1}{3} \cdot i}_{i} = \frac{2}{3} \cdot (3k + i) + \frac{1}{3} \cdot i = \frac{2}{3} \cdot n_{j-1} + \frac{1}{3} \cdot i \leq \frac{2}{3} \cdot n_{j-1} + \frac{2}{3}$$

Letztere Abschätzung gilt wegen $0 \leq i \leq 2$, womit $0 \leq \frac{1}{3} \cdot i \leq \frac{2}{3}$ folgt. Wir schätzen nun n_{j-1} genauso ab und erhalten:

$$n_j \leq \frac{2}{3} \cdot n_{j-1} + \frac{2}{3} \leq \frac{2}{3} \cdot \left(\frac{2}{3} \cdot n_{j-2} + \frac{2}{3} \right) + \frac{2}{3} = \left(\frac{2}{3} \right)^2 \cdot n_{j-2} + \left(\frac{2}{3} \right)^2 + \left(\frac{2}{3} \right)$$

Wenn wir uns i Ebenen hoch hangeln, gilt also anscheinend:

$$n_j \leq \left(\frac{2}{3} \right)^i \cdot n_{j-i} + \left(\frac{2}{3} \right)^i + \cdots + \left(\frac{2}{3} \right)^2 + \left(\frac{2}{3} \right) = \left(\frac{2}{3} \right)^i \cdot n_{j-i} + \sum_{k=1}^{i} \left(\frac{2}{3} \right)^k$$

Da wir den Wert $n_0 = n_{j-j} = n$ erreichen wollen, setzen wir $i = j$ und erhalten:

$$n_j \leq \left(\frac{2}{3} \right)^j \cdot n + \sum_{k=1}^{j} \left(\frac{2}{3} \right)^k$$

Wir schätzen die Summe mit Hilfe der geometrischen Reihe ab:

$$\sum_{k=1}^{j} \left(\frac{2}{3} \right)^k \leq \sum_{k=1}^{\infty} \left(\frac{2}{3} \right)^k = \sum_{k=0}^{\infty} \left(\frac{2}{3} \right)^k - 1 = \frac{1}{1 - \frac{2}{3}} - 1 = 3 - 1 = 2$$

Insgesamt gilt somit:

$$n_j \leq \left(\frac{2}{3} \right)^j \cdot n + 2$$

Unser Ziel ist es, zu ermitteln, auf welcher Stufe nur noch höchstens zwei Summanden übrig sind. Es muss also $n_j \leq 2$ gelten. Damit wir gleich beim Umformen jedoch keine Probleme nach der Subtraktion der 2 bekommen, fragen wir uns, ab welcher Stufe j die Bedingung $n_j < 3$ gilt.

$$\left(\frac{2}{3} \right)^j \cdot n + 2 < 3 \quad \Leftrightarrow \quad \left(\frac{2}{3} \right)^j \cdot n < 1 \quad \Leftrightarrow \quad n < \left(\frac{3}{2} \right)^j \quad \Leftrightarrow \quad j > \lceil \log_{\frac{3}{2}} n \rceil$$

Wir benötigen somit höchstens $\lceil \log_{\frac{3}{2}} n \rceil$ Stufen, bis wir den Ladner/Fisher Addierer benutzen können. Jede diese Stufen besteht aus parallel geschalteten Carry-Save-Addern mit Tiefe 3. Der CSA-Teil des Wallace-Trees hat somit Tiefe $3 \cdot \lceil \log_{\frac{3}{2}} n \rceil$. Der Ladner/Fisher Addierer

muss anschließend zwei Zahlen der Länge $2n$ addieren, was in Tiefe $2\log(2n)+2 = 2\cdot(\log 2 + \log n)+2 = 2\cdot(1+\log n)+2 = 2\log n + 4$ möglich ist. Der erste Schritt der Schulmethode benötigt Tiefe 1, womit sich folgende Gesamttiefe ergibt:

$$\begin{aligned} 3\cdot\lceil\log_{\frac{3}{2}} n\rceil + 2\log n + 4 + 1 &= 3\cdot\left\lceil\frac{\log n}{\log\frac{3}{2}}\right\rceil + 2\log n + 5 \approx 3\cdot\lceil 1.71\log n\rceil + 2\log n + 5 \\ &\approx 5.13\log n + 2\log n + 5 = 7.13\log n + 5 \end{aligned}$$

3.8 Multiplikation mit einem Divide-and-Conquer Verfahren

Frage 3.22: Was ist die grundlegende Idee beim Verfahren von Karatsuba und Ofman?
Das Multiplikationsverfahren von Karatsuba und Ofman ist ein Divide-and-Conquer Algorithmus. Konkret bedeutet dies, dass in einem Schritt nicht zwei große Zahlen multipliziert werden, sondern mehrere kleinere Zahlen. Aus diesen Ergebnissen wird dann das Ergebnis der Multiplikation der beiden großen Zahlen berechnet. Wir überlegen uns nun, wie wir zwei „große" Zahlen $x = (x_{n-1},\ldots,x_0)$ und $y = (y_{n-1},\ldots,y_0)$ zerlegen können, wenn wir annehmen, dass $n = 2^k$ gilt. Dann können wir die Zahl x auch schreiben als:

$$|(x_{n-1},\ldots,x_0)| = |\underbrace{(x_{n-1},\ldots,x_{\frac{n}{2}})}_{x'}|\cdot 2^{\frac{n}{2}} + |\underbrace{(x_{\frac{n}{2}-1},\ldots,x_0)}_{x''}|$$

Mit x' bezeichnen wir also die vorderen $\frac{n}{2}$ Bits von x und mit x'' die hinteren $\frac{n}{2}$ Bits. Entsprechendes gelte für y. Dann können wir das Produkt von x und y auch wie folgt ausdrücken:

$$\begin{aligned} |p| &= |x|\cdot|y| = (|x'|2^{\frac{n}{2}} + |x''|)\cdot(|y'|2^{\frac{n}{2}} + |y''|) \\ &= |x'||y'|\cdot 2^{\frac{n}{2}}\cdot 2^{\frac{n}{2}} + |x'||y''|2^{\frac{n}{2}} + |x''||y'|2^{\frac{n}{2}} + |x''||y''| \\ &= |x'||y'|\cdot 2^n + (|x'||y''| + |x''||y'|)2^{\frac{n}{2}} + |x''||y''| \end{aligned}$$

Wir haben ein Multiplikationsproblem somit in vier Multiplikationsteilprobleme zerlegt. Die Multiplikationen mit Zweierpotenzen sind kostenlos, da nur einige Nullen angehängt werden müssen. Bisher haben wir jedoch noch nichts gewonnen, da die Größe bei einer direkten Anwendung dieses Ansatzes immer noch quadratisch werden würde. Dies ist schnell gezeigt. Mit $C(l)$ bezeichnen wir die Größe des Schaltkreises wenn zwei Zahlen der Länge l multipliziert werden sollen. Es gilt $C(1) = 1$, da die Multiplikation von zwei Zahlen der Länge 1 nur einen $\wedge$-Baustein erfordert. Wenn wir also nur die Multiplikationen zählen, gilt schon $C(n) = 4\cdot C(\frac{n}{2})$. Wir lösen die für $n = 2^k$ weiter auf:

$$C(2^k) = 4\cdot C(2^{k-1}) = 4\cdot 4\cdot C(2^{k-2}) = \cdots = 4^i\cdot C(2^{k-i})$$

Wir setzen $i = k$, damit wir $C(1) = 1$ ausnutzen können, und erhalten:

$$C(n) = C(2^k) = 4^k\cdot C(1) = 4^k = \left(2^2\right)^k = \left(2^k\right)^2 = n^2 = \Theta(n^2)$$

Der „Trick" von Karatsuba und Ofman besteht nun darin, die Anzahl der Teilprobleme von vier auf drei zu reduzieren. Dazu schreiben wir den mittleren Summanden $|x'||y''| + |x''||y'|$ etwas um:

$$|x'||y''| + |x''||y'| = (|x'| + |x''|)\cdot(|y'| + |y''|) - (|x'||y'| + |x''||y''|)$$

Wieso sparen wir hier? Die einzige Multiplikation, die hier durchgeführt wird, ist $(|x'| + |x''|) \cdot (|y'| + |y''|)$. Im Ergebnis kommen auch $|x'||y''|$ und $|x''||y'|$ vor, jedoch auch $|x'||y'|$ und $|x''||y''|$ welche wir also wieder subtrahieren müssen. Interessanterweise müssen wir $|x'||y'|$ und $|x''||y''|$ jedoch sowieso berechnen, so dass hier keine zusätzlichen Multiplikationen anfallen und wir uns also insgesamt eine Multiplikation sparen. Wir können x und y somit wie folgt multiplizieren, woraus sich auch direkt ein Algorithmus ergibt:

$$|p| = |x| \cdot |y| = |x'||y'|2^n + \left[(|x'| + |x''|) \cdot (|y'| + |y''|) - (|x'||y'| + |x''||y''|)\right] 2^{\frac{n}{2}} + |x''||y''|$$

Frage 3.23: Beschreibe den Algorithmus zur Multiplikation von Karatsuba und Ofman!

Der Algorithmus ergibt sich im Prinzip direkt aus der oben gewonnenen Formel und wird hier auch nicht vollständig im Detail beschrieben. Im ersten Schritt werden die Summen $(|x'| + |x''|)$ und $(|y'| + |y''|)$ berechnet. Dabei entstehen Zahlen der Länge $\frac{n}{2} + 1$, da zwei Zahlen der Länge $\frac{n}{2}$ addiert werden. Im Anschluss werden die drei Multiplikationen parallel durchgeführt. Sind die drei Ergebnisse bekannt, müssen nur noch einige Additionen und eine Subtraktion durchgeführt werden. Bei der Addition von $|x'||y'|2^n + |x''||y''|$ fallen keine Kosten an, da die beiden Zahlen einfach konkateniert werden müssen. Das Ergebnis von $|x''||y''|$ ist nämlich n Stellen lang und $|x'||y'|2^n$ hat am Ende genau n Nullen. Wir müssen somit nur diese Nullen durch die Bits von $|x''||y''|$ ersetzen. Auf die Additionen und die Subtraktion wird hier nicht weiter eingegangen, da die Größe dieser später bei der Anschätzung der Größe des gesamten Schaltkreises durch $O(n)$ abgeschätzt wird.

Frage 3.24: Welche Größe und Tiefe haben Karatsuba-Ofman Multiplikationsschaltkreise?

Wir schätzen nun die Anzahl der Bausteine ab, die wir auf der Ebene benötigen, auf der zwei Zahlen der Länge n multipliziert werden. Die Anzahl bezeichnen wir mit $M(n)$. Die Größe der Additionen und der Subtraktion auf dieser Ebene schätzen wir mit $c \cdot n = O(n)$ ab, da wir konstant viele Additionen benötigen, welche in linearer Größe realisiert werden können. Dann gilt:

$$M(n) \leq 2 \cdot M\left(\frac{n}{2}\right) + M\left(\frac{n}{2} + 1\right) + c \cdot n$$

Der Ausdruck $M(\frac{n}{2} + 1)$ in dieser Formel ist unschön. Wir ersetzen ihn deshalb, wobei wir denselben Trick wie schon etwas zuvor benutzen. Eine Zahl x mit $\frac{n}{2} + 1$ Stellen können wir nämlich auch wie folgt schreiben:

$$|x| = |(x_{\frac{n}{2}}, \ldots, x_1)| \cdot 2 + |x_0|$$

Wenn wir $x' = (x_{\frac{n}{2}}, \ldots, x_1)$ sowie $x'' = x_0$ setzen und dieselbe Notation für eine Zahl y benutzen, dann gilt für das Produkt:

$$|p| = |x| \cdot |y| = |x'||y'| \cdot 2^2 + (|x'||y''| + |x''||y'|) \cdot 2 + |x''||y''|$$

Für die Berechnung von $|x'||y'|$ benötigen wir $M(\frac{n}{2})$ viele Bausteine, da x' und y' je $\frac{n}{2}$ Bits lang sind. Bei der Berechnung von $|x'||y''|$ bzw. $|x''||y'|$ wird jeweils eine $\frac{n}{2}$ Bits lange Zahl mit einer ein Bit langen Zahl multipliziert. Dafür reichen jeweils $\frac{n}{2}$ $\wedge$-Bausteine. Für die Multiplikation von x'' mit y'' reicht ein $\wedge$-Baustein, da beide Zahlen nur ein Bit lang sind. Die Addition schätzen wir wieder mit $d \cdot n = O(n)$ ab. Insgesamt erhalten wir somit:

$$M\left(\frac{n}{2} + 1\right) \leq M\left(\frac{n}{2}\right) + c \cdot n$$

Wenn wir dieses Ergebnis in die Formel von $M(n)$ einsetzen, erhalten wir:

$$M(n) \leq 3 \cdot M\left(\frac{n}{2}\right) + c \cdot n$$

Wieder setzen wir $n = 2^k$ und lösen auf:

$$\begin{aligned}
M(2^k) &\leq 3 \cdot M(2^{k-1}) + c \cdot 2^k \leq 3 \cdot \left(3 \cdot M(2^{k-2}) + c \cdot 2^{k-1}\right) + c \cdot 2^k \\
&= 3^2 \cdot M(2^{k-2}) + \left(\frac{3}{2}\right) c2^k + c2^k \leq 3^2 \cdot \left(3 \cdot M(2^{k-3}) + c \cdot 2^{k-2}\right) + \left(\frac{3}{2}\right) c2^k + c2^k \\
&= 3^3 \cdot M(2^{k-3}) + \left(\frac{3}{2}\right)^2 c2^k + \left(\frac{3}{2}\right) c2^k + \left(\frac{3}{2}\right)^0 c2^k \\
&\leq 3^i \cdot M(2^{k-i}) + \left(\frac{3}{2}\right)^i c2^k + \left(\frac{3}{2}\right)^2 c2^k + \left(\frac{3}{2}\right) c2^k + \left(\frac{3}{2}\right)^0 c2^k \\
&= 3^i \cdot M(2^{k-i}) + c2^k \sum_{j=0}^{i-1} \left(\frac{3}{2}\right)^j
\end{aligned}$$

Damit wir $M(1) = 1$ nutzen können, setzen wir $i = k$:

$$M(n) = M(2^k) = 3^k \cdot M(1) + c2^k \sum_{j=0}^{k-1} \left(\frac{3}{2}\right)^j$$

Es gilt:

$$\sum_{j=0}^{k} a^j = \frac{a^{k+1} - 1}{a - 1}$$

In unserem Fall gilt also:

$$\sum_{j=0}^{k-1} \left(\frac{3}{2}\right)^j = \frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = \frac{\left(\frac{3}{2}\right)^k - 1}{\frac{1}{2}} = 2 \cdot \left(\left(\frac{3}{2}\right)^k - 1\right) = 2 \cdot \left(\frac{3}{2}\right)^k - 2 \leq 2 \cdot \left(\frac{3}{2}\right)^k$$

Dieses Ergebnis setzen wir in die Formel für $M(n)$ ein:

$$M(n) = 3^k + c2^k \cdot 2 \cdot \left(\frac{3}{2}\right)^k = 3^k + c2^k \cdot 2 \cdot \frac{3^k}{2^k} = 3^k + 2c \cdot 3^k = (1 + 2c) \cdot 3^k$$

Wir wollen dieses Ergebnis in Abhängigkeit von n ausdrücken und formen 3^k wie folgt um:

$$3^k = \left(2^{\log 3}\right)^k = \left(2^k\right)^{\log 3} = n^{\log 3}$$

Es gilt somit $M(n) \leq (1+2c) \cdot n^{\log 3} = \Theta(n^{\log 3})$, d.h. Karatsuba-Ofman Multiplikationsschaltkreise haben eine Größe von $\Theta(n^{\log 3})$.

Kommen wir nun zur Abschätzung der Tiefe. Um zwei Zahlen der Größe n zu multiplizieren, müssen parallel drei Zahlen der Größe $\frac{n}{2}$ multipliziert werden und eine konstante Anzahl an Additionen durchgeführt werden. Die Subtraktion kostet in etwa soviel wie eine Addition. Wenn wir für die Additionen Ladner/Fisher Addierer benutzen, können wir eine Addition in

$O(\log n)$ bewerkstelligen. Wir schätzen die Tiefe der Additionen daher mit $d \cdot \log n$ ab und bekommen die folgende Rekursionsformel:

$$D(n) = D\left(\frac{n}{2}\right) + d \cdot \log n \qquad \text{mit} \qquad D(1) = 1$$

Für $n = 2^k$ erhalten wir dann:

$$\begin{aligned} D(2^k) &= D(2^{k-1}) + d \cdot k = D(2^{k-2}) + d \cdot (k-1) + d \cdot k = \ldots \\ &= D(2^{k-i}) + d \cdot (k-i+1) + \cdots + d \cdot (k-1) + d \cdot k \end{aligned}$$

Wir setzen $i = k$:

$$D(n) = D(2^k) = D(1) + d \cdot \sum_{j=1}^{k} j = 1 + d \cdot \frac{k \cdot (k+1)}{2} = O(k^2) = O(\log^2 n)$$

Wir wollen die Tiefe im Folgenden noch auf $O(\log n)$ drücken. Wenn wir uns die Rekursionsformel und ihre Entwicklung anschauen, fällt auf, dass die Additionen in jeweils logarithmischer Tiefe dafür sorgen, dass die bisherige Version eine Laufzeit von $O(\log^2 n)$ bekommt. Wenn wir in der Lage wären, in konstanter Tiefe zu addieren, würden wir eine Laufzeit von $O(\log n)$ erreichen. Im nächsten Kapitel wird die Radix-4 Darstellung eingeführt, mit deren Hilfe dies möglich ist. Eine Frage, die sich nun stellt, ist, warum wir Binärzahlen dann nicht immer in Radix-4 Darstellung addieren. Dies liegt daran, dass die Umwandlung einer Binärzahl in ihre Radix-4 Darstellung zwar in Tiefe $O(1)$ möglich ist, die Rückumwandlung des Ergebnisses aus einer Radix-4 Darstellung in eine Binärzahl jedoch eine Tiefe von $O(\log n)$ benötigt. Ein solcher Algorithmus hätte aufgrund der Umwandlungen also ebenfalls eine Tiefe von $O(\log n)$, genau wie die effizientesten der bisher vorgestellten Verfahren. Die Verwendung der Radix-4 Darstellung macht das Verfahren von Karatsuba-Ofman nur deshalb effizient, da alle Additionen in Radix-4 Darstellung vorgenommen werden und nur eine Rückumwandlung am Ende stattfindet, d.h. die Extratiefe von $\log n$ wird nur einmal zusätzlich addiert, weshalb die Tiefe bei $O(\log n)$ bleibt.

3.9 Die Radix-4 Darstellung

Frage 3.25: Zur Motivation: Warum beschäftigen wir uns mit der Radix-4 Darstellung?
Die Radix-4 Darstellung ermöglicht es, die Addition zweier Zahlen sowie die Multiplikation mit Zahlen 2^i in Größe $O(n)$ und Tiefe $O(1)$ durchzuführen. Die Umwandlung einer Binärzahl in eine Radix-4 Darstellung geht ebenfalls in $O(1)$. Das Verfahren von Karatsuba-Ofman wird also dadurch optimiert, dass die Rekursion bis zur Multiplikation einstelliger Zahlen durchlaufen wird, die einstelligen Zahlen multipliziert werden und anschließend die Ergebnisse in $O(1)$ in die Radix-4 Darstellung umgewandelt werden. Danach werden erst einmal jede Menge Additionen und Multiplikationen mit Zahlen der Form 2^i durchgeführt (also Shifts gemacht). Da dies nun in $O(1)$ möglich sind, hängt die Laufzeit somit nur noch von der Tiefe des Rekursionsbaums ab, welche $O(\log n)$ beträgt. Im Anschluss muss das Ergebnis, welches in Radix-4 Darstellung vorliegt, wieder zurück in eine Binärdarstellung umgewandelt werden, was in Zeit $O(\log n)$ möglich ist. Damit hat der gesamte Algorithmus eine Laufzeit von $O(\log n + \log n) = O(2 \log n) = O(\log n)$.

Frage 3.26: Was ist ein Fermat-Ring?

$\mathbb{Z}_m$ heißt Fermat-Ring, wenn $m = 4^n + 1$ für eine ganzzahlige Zahl n ist. Beispiele für Fermat-Ringe sind also $\mathbb{Z}_2, \mathbb{Z}_5, \mathbb{Z}_{17}, \mathbb{Z}_{65}, \mathbb{Z}_{257}$, usw.

Frage 3.27: Wie sieht die Radix-4 Darstellung für ein Element eines Fermat-Rings aus?

Es sei $a \in \mathbb{Z}_m$, wobei $\mathbb{Z}_m$ ein Fermat-Ring sei. Dann ist der Vektor $x = (x_n, \ldots, x_0)$ mit $-3 \leq x_i \leq 3$ eine Radix-4 Darstellung für a, wenn gilt:

$$|x|_4 = x_n \cdot 4^n + x_{n-1} \cdot 4^{n-1} + \cdots + x_0 \equiv a \bmod m$$

Diese Darstellung ist durch zwei Umstände redundant. Zum einen weil negative x_i erlaubt sind (was z.B. bei der Binärdarstellung von Zahlen nicht „erlaubt“ ist) und zum anderen weil modulo m gerechnet wird. So gibt es z.B. für das Element $3 \in \mathbb{Z}_5$ die direkte Darstellung als $(0, 3)$, die Darstellung $(1, -1)$, welche aufgrund der negativen x_i möglich ist, und die Darstellung $(2, 0)$, die aufgrund der Rechnungen modulo m möglich ist.

Die Kodierung eines jeden x_i in einem Vektor x kann durch drei Bits geschehen. Dabei wird in einem Bit das Vorzeichen kodiert und in den anderen beiden der Betrag der Zahl. So könnte $x_i = -3$ also als $(1, 1, 1)$ kodiert werden.

Frage 3.28: Wie kann die Binärdarstellung einer Zahl in die Radix-4 umgewandelt werden?

Die Umwandlung einer Binärdarstellung in eine Radix-4 Darstellung ist in Zeit $O(1)$ möglich, da die Binärdarstellung sofort als Radix-4 Darstellung aufgefasst werden kann. Unser Ziel ist es, die Radix-4 Darstellung $y = (y_n, \ldots, y_0)$ einer Binärzahl $x = (x_{2n}, \ldots, x_0)$ zu gewinnen. Zuerst einmal überlegen wir uns, warum die Binärzahl aus $2n+1$ Stellen besteht, da wir gleich aus Konsistenzgründen $x_{2n+1} = 0$ setzen werden. Wir rechnen in $\mathbb{Z}_m$ mit $m = 4^n + 1$. Da wir modulo m rechnen, liegen alle Zahlen also zwischen 0 und $m - 1 = 4^n = 2^{2n}$. Mit $2n$ Bits können wir höchstens die Zahl $2^{2n} - 1$ darstellen. Also benötigen wir für die Binärdarstellung der Zahl 2^{2n} mindestens $2n + 1$ Bits und genau so lang ist die gegebene Binärzahl x.

Wir werden gleich sehen, dass die Bits x_{2i+1} und x_{2i} zur Radix-4 Komponente y_i korrespondieren. Wir setzen daher $x_{2n+1} = 0$, damit wir diese Sichtweise für x und y benutzen können. Es gilt dann $y_i = 2x_{2i+1} + x_{2i}$. Wir bekommen die Komponente y_i des Vektors y also, indem wir die Bits $x_{2i+1}x_{2i}$ als Binärzahl interpretieren, deren Dezimaldarstellung dem Wert von y_i entspricht. Die Kodierung von y_i ergibt sich entsprechend als $(0, x_{2i+1}, x_{2i})$. Warum gilt diese Korrespondenz? Die Stelle x_{2i+1} wird mit 2^{2i+1} gewichtet und die Stelle x_{2i} mit 2^{2i}. Entsprechend wird die Stelle y_i mit 4^i gewichtet. Wir müssen also nur zeigen, dass $x_{2i+1}2^{2i+1} + x_{2i}2^{2i} = (2x_{2i+1} + x_{2i})4^i$ gilt. Hier könnten wir nun eine Fallunterscheidung machen, in der wir dies für alle vier möglichen Fälle nachweisen. Stattdessen können wir jedoch einfach den ersten Ausdruck umformen:

$$x_{2i+1}2^{2i+1} + x_{2i}2^{2i} = 2x_{2i+1}2^{2i} + x_{2i}2^{2i} = (2x_{2i+1} + x_{2i})2^{2i} = (2x_{2i+1} + x_{2i})4^i$$

Frage 3.29: Wie kann die Radix-4 Darstellung einer Zahl in die Binärdarstellung umgewandelt werden?

Die Umwandlung einer Radix-4 Darstellung y in die Binärdarstellung x ist leider nicht so einfach wie die Umwandlung in die andere Richtung. Dies liegt daran, dass sowohl positive als auch negative y_i in der Radix-4 Darstellung y gemischt vorkommen können. Wir haben

schon gesehen, dass die Umwandlung einer positiven Binärzahl in eine Radix-4 Darstellung einfach ist. In der resultierenden Radix-4 Darstellung kommen nur positive y_i vor, da die Binärstellen x_{2i+1} und x_{2i} als positive Binärzahl $x_{2i+1}x_{2i}$ interpretiert werden und so den Wert von y_i bestimmen. Wenn alle y_i positiv sind, ist jedoch auch die Rückumwandlung einfach, da sich die Stelle $x_{2i+1}x_{2i}$ einfach als Binärdarstellung von y_i ergibt. Sind alle Stellen negativ (oder Null) ist die Umwandlung ebenfalls einfach. Wir negieren alle y_i, berechnen die Binärdarstellung und geben dieser ein negatives Vorzeichen.

Was machen wir jedoch, wenn wir eine gemischte Radix-4 Darstellung vorliegen haben? Wir werden uns das Vorgehen in diesem Fall nun an einem Beispiel klarmachen. Wir setzen $n = 4$ und befinden uns somit im Fermat-Ring $\mathbb{Z}_{257}$. Als Element aus $\mathbb{Z}_{257}$ wählen wir die Zahl 216, die unter anderem die Radix-4 Darstellung $(1, -1, 2, -2, 0)$ hat. Es gilt:

$$|(1, -1, 2, -2, 0)|_4 = 1 \cdot 4^4 - 1 \cdot 4^3 + 2 \cdot 4^2 - 1 \cdot 4 + 0 \bmod 257$$

Wir ordnen die hintere Summe nun um, wobei wir den Ausdruck „mod m“ der Übersicht halber weglassen:

$$1 \cdot 4^4 - 1 \cdot 4^3 + 2 \cdot 4^2 - 1 \cdot 4 = 4 \cdot 4^4 + 2 \cdot 4^2 - 1 \cdot 4^3 - 2 \cdot 4 = 4 \cdot 4^4 + 2 \cdot 4^2 - (1 \cdot 4^3 + 2 \cdot 4)$$

Dieses Ergebnis können wir wieder umschreiben zu:

$$4 \cdot 4^4 + 2 \cdot 4^2 - (1 \cdot 4^3 + 2 \cdot 4) \bmod m = |\underbrace{(1, 0, 2, 0, 0)}_{y^+}|_4 - |\underbrace{(0, 1, 0, 2, 0)}_{y^-}|_4$$

Die Vektoren $y^+ = (1, 0, 2, 0, 0)$ und $y^- = (0, 1, 0, 2, 0)$ enthalten beide nur positive y_i, daher können wir leicht die Binärdarstellung von y^+ und y^- berechnen. Wenn wir diese entsprechend voneinander subtrahieren, bekommen wir somit die Binärdarstellung von y. Der Vektor y^+ enthält also nur alle positiven Werte von y. Stellen, die in y negativ oder 0 sind, werden in y^+ alle auf 0 gesetzt. Entsprechend enthält y^- alle negativen Stellen von y, wobei diese jedoch zusätzlich negiert werden. Alle Stellen die positiv oder 0 sind, werden in y^- auf 0 gesetzt. Es gilt also für einen Vektor y, dass sich der Vektor y^+ als $y^+ = (y_n^+, \ldots, y_0^+)$ und y^- als $y^- = (y_n^-, \ldots, y_0^-)$ bildet, wobei gilt:

$$y_i^+ = \begin{cases} y_i & \text{falls } y_i > 0 \\ 0 & \text{sonst} \end{cases} \qquad \text{und} \qquad y_i^- = \begin{cases} -y_i & \text{falls } y_i < 0 \\ 0 & \text{sonst} \end{cases}$$

Zur Selektion, ob y_i oder 0 bzw. $-y_i$ oder 0 gewählt wird, können wir Select-Bausteine (siehe Kapitel 3.4) benutzen, wobei das Vorzeichenbit der Kodierung von y_i angibt, ob y_i oder 0 gewählt wird.

Bisher haben wir bei diesem Vorgehen jedoch noch nicht beachtet, dass wir modulo m rechnen wollen, d.h. dass das Ergebnis im Bereich $0, \ldots, m-1 = 4^n$ liegen soll. Es kann nun sein, dass wir sofort in diesem Bereich „landen“, jedoch können wir ihn auch nach oben oder nach unten überschreiten. Wir überlegen uns nun zuerst, wie weit wir diesen Bereich höchstens überschreiten können. Die maximale Überschreitung in die positive Richtung liegt vor, wenn $y^+ = (3, 3, \ldots, 3)$ und $y^- = (0, 0, \ldots, 0)$ gilt. Die maximale negative Überschreitung liegt vor, wenn y^- nur Dreien enthält und y^+ Null ist. Die maximale Überschreitung ist also $+|(3, \ldots, 3)|_4$ bzw. $-|(3, \ldots, 3)|_4$, wobei wir nun nicht modulo m rechnen, da die Binärzahl,

	x_n	$\dots$	x_i	$\dots$	x_2	x_1	x_0
	y_n	$\dots$	y_i	$\dots$	y_2	y_1	y_0
0	v_n	$\dots$	v_i	$\dots$	v_2	v_1	v_0
c_n	c_{n-1}	$\dots$	c_{i_1}	$\dots$	c_1	c_0	0
c_n	s_n	$\dots$	s_i	$\dots$	s_2	s_1	s_0

Tabelle 7: Die Schritte der Addition zweier Radix-4 Zahlen

die wir bei diesem Vorgehen rausbekommen, ja ebenfalls nicht modulo gerechnet wird. Es gilt:

$$|(3,\dots,3)|_4 = 3\cdot 4^n + \dots + 3\cdot 4^1 + 3 = 3\cdot\sum_{i=0}^{n} 4^i = 3\cdot\frac{4^{n+1}-1}{4-1} = 4^{n+1}-1$$

Es kann nun gezeigt werden, dass von der Zahl $4^{n+1}-1$ genau dreimal die Zahl 4^n+1 abgezogen werden muss, damit der Bereich $0,\dots,m-1=4^n$ erreicht wird. Ebenso muss zur Zahl $-(4^{n+1}-1)$ genau vier mal die Zahl 4^n+1 addiert werden, damit wir im selben Bereich landen. Dabei wurden hier jedoch die Grenzfälle betrachtet, d.h. einige Ergebnisse liegen unter Umständen direkt im Bereich $0,\dots,4^n$ und zu wieder anderen Ergebnissen muss nicht mehr so oft 4^n+1 addiert werden, damit sie in diesem Bereich liegen. Insgesamt gibt es also nur acht Fälle zu unterscheiden. Ist b die Binärdarstellung von $|y^+|_4 - |y^-|_4$, dann berechnen wir also parallel für $-3 \le i \le 4$ die Werte $b + i\cdot m = b + i\cdot(4^n+1)$. Einer dieser acht Werte muss dann im Bereich $0,\dots,m-1$ liegen und genau diesen Wert wählen wir aus. Im Prinzip werden alle Ergebnisse weggeschmissen, die echt kleiner als 0 sind oder die echt größer als 4^n sind. Ersteren Fall erkennen wir direkt am Vorzeichenbit des Ergebnisses. Den zweiten Fall erkennen wir dadurch, dass im Ergebnis das Bit $x_{2n}=1$ ist und noch mindestens ein anderes Bit gesetzt ist oder dass direkt eines der höherwertigen Bits vor x_{2n} gesetzt ist.

Frage 3.30: Wie werden Radix-4 Zahlen addiert und subtrahiert? Welche Größe und Tiefe hat der Schaltkreis für diese Operationen?

Die Addition bzw. Subtraktion zweier Radix-4 Zahlen ist in Größe $O(n)$ und Tiefe $O(1)$ möglich. Zuerst überlegen wir uns, dass es ausreicht, die Addition zu betrachten. Der Grund dafür liegt darin, dass folgende Gleichung gilt:

$$|x|_4 - |y|_4 = |(x_n,\dots,x_0)|_4 - |(y_n,\dots,y_0)|_4 = |(x_n,\dots,x_0)|_4 + |(-y_n,\dots,-y_0)|_4$$

Wir wollen nun also die Zahlen $x=(x_n,\dots,x_0)$ und $y=(y_n,\dots,y_0)$ addieren. Dabei werden wir ähnlich wie beim von Neumann Addierwerk vorgehen. Das heißt, dass wir aus den Zahlen x und y zwei neue Radix-4 Zahlen v und c berechnen werden, deren Summe den gleichen Wert hat, wie die Summe von x und y. Auch hier werden wir wieder ein Summenbit v_i und ein Carrybit c_i berechnen, wobei die „Bits“ in diesem Fall jedoch zwischen -3 und 3 liegen. Das Ganze sieht nach dem ersten Schritt dann wie im Mittelteil von Tabelle 7 dargestellt aus.

Wenn wir genau wie beim von Neumann Addierwerk vorgehen würden, hätten wir nichts gewonnen. Aufgrund der Redundanz der Darstellung haben wir jedoch etwas Spielraum, wie wir die Summe x_i+y_i auf die Summenbits v_i und die Carrybits c_i verteilen. Die Carrybits werden nach der Formel $x_i+y_i = 4\cdot c_i + v_i$ verteilt. Wenn die Summe z.B. 3 ergibt, würden wir im „Normalfall“ $v_i=3$ und $c_i=0$ setzen, d.h. die 3 würde komplett in das Summenbit eingehen und es würde kein Carry erzeugt werden. Wir haben jedoch schon gesehen, dass Radix-4 Zahlen eine redundante Darstellung besitzen. Dies werden wir hier nun speziell im

$x_i + y_i$	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
v_i	-2	-1	0	1	-2	-1	0	1	2	-1	0	1	2
c_i	-1	-1	-1	-1	0	0	0	0	0	1	1	1	1

Tabelle 8: Die Wahl der Werte v_i und c_i für $x_i + y_i$

Fall von $x_i + y_i = 3$ und $x_i + y_i = -3$ ausnutzen. Alternativ können wir im Fall $x_i + y_i = 3$ nämlich auch ein Carrybit erzeugen und dies korrigieren, indem wir im Summenbit wieder 1 abziehen, d.h. wir setzen $c_i = 1$ und $v_i = -1$. Tabelle 8 zeigt, wie wir die Summenbits und Carrybits wählen. Dabei fällt auf, dass $|v_i| \leq 2$ und $|c_i| \leq 1$ gilt. Die Summe s von $v = (0, v_n, \ldots, v_0)$ und $c = (c_n, \ldots, c_0, 0)$ ergibt also dasselbe Ergebnis wie die Summe von x und y. Wenn wir v und c jetzt wieder „normal“ addieren, gilt jedoch aufgrund von $|v_i| \leq 2$ und $|c_i| \leq 1$, dass auch $|v_i + c_i| \leq 3$ gilt. Es entsteht also kein neues Carrybit, wenn wir v und c addieren. Die Summe s ergibt sich bis hierhin als $s_i = v_i + c_{i-1}$ für $1 \leq i \leq n$. Weiter setzen wir $s_{n+1} = c_n$ und $s_0 = v_0$.

Bisher haben wir jedoch einen wichtigen Umstand nicht beachtet. Das Carrybit c_n wird ebenfalls modulo m gerechnet und könnte uns bei der Argumentation, dass keine neuen Carries bei der Addition von v und c erzeugt werden, „dazwischenfunken“. Indem wir die Summe $s = (c_n, s_n, \ldots, s_0)$ nun als die Summe zweier anderer Zahlen s' und s'' darstellen, werden wir zeigen, dass trotz des c_n nichts Schlimmes passiert. Und zwar setzen wir $s' = (0, 0, s_{n-1}, \ldots, s_0)$ und $s'' = (c_n, s_n, 0, \ldots, 0)$. Es gilt offensichtlich $s' + s'' = s$. Als nächstes suchen wir nun eine alternative Darstellung für s'', wobei wir ausnutzen werden, dass die Radix-4 Darstellung aufgrund der Rechnungen modulo m redundant ist. Die Stelle c_n geht mit der Wertigkeit 4^{n+1} ein. Wir formen dies nun um, wobei wir modulo $m = 4^n + 1$ rechnen:

$$c_n \cdot 4^{n+1} \equiv c_n \cdot 4 \cdot 4^n \equiv 4c_n \cdot ((4^n + 1) - 1) \equiv 4c_n \cdot (4^n + 1) - 4c_n \equiv 0 - 4c_n \equiv -c_n \cdot 4 \bmod m$$

Damit können wir s'' also auch als $s'' = (0, s_n, 0, \ldots, -c_n, 0)$ darstellen. Mit dem Wert s_n werden wir nun ähnlich verfahren:

$$s_n \cdot 4^n \equiv s_n \cdot ((4^n + 1) - 1) \equiv -s_n \equiv -s_n \cdot 4^0 \bmod m$$

Insgesamt können wir s'' somit als $s'' = (0, 0, \ldots, -c_n, -s_n)$ darstellen. Wenn wir nun in $s' = (0, 0, s_{n-1}, \ldots, s_0)$ und $s'' = (0, 0, \ldots, -c_n, -s_n)$ jeweils die führende 0 streichen, bekommen wir wieder zwei Radix-4 Zahlen mit $n + 1$ Stellen. Beide Zahlen haben eine 0 an der (neuen) führenden Stelle. Wenn wir diese beiden Zahlen also nach der oben beschriebenen Methode addieren, gilt somit $c_n = 0$ und wir können wie oben beschrieben argumentieren.

Frage 3.31: Wie kann eine Zahl in Radix-4 Darstellung mit einer Zweierpotenz multipliziert werden?

Wir wollen uns nun überlegen, wie eine Zahl $y = (y_n, \ldots, y_0)$ in Radix-4 Darstellung mit einer Zweierpotenz 2^k mit $k \leq 2n$ multipliziert werden kann. Es wird sich herausstellen, dass diese Operation auf die Addition zurückgeführt werden kann, womit sie in Größe $O(n)$ und Tiefe $O(1)$ realisiert werden kann.

Zuerst machen wir nun eine Fallunterscheidung, ob der Exponent k gerade oder ungerade ist. Wir schreiben k nun also als $k = 2l + i$, wobei $i \in \{0, 1\}$ gelte. Wenn $i = 1$ und damit k ungerade ist, schreiben wir die Multiplikation wie folgt:

$$|y|_4 \cdot 2^k = |y|_4 \cdot 2^{2l+1} = |y|_4 \cdot 2^{2l} \cdot 2 = |y|_4 \cdot 2^{2l} + |y|_4 \cdot 2^{2l}$$

Die Multiplikation mit einer Zweierpotenz, die einen ungeraden Exponenten hat, kann somit auf die Addition zweier Zahlen zurückgeführt werden, die durch Multiplikation mit einer Zweierpotenz mit geradem Exponenten entstehen. Wir müssen uns somit nur noch überlegen, wie die Multiplikation mit 2^k funktioniert, wenn k gerade ist. In diesem Fall schreiben wir $|y|_4 \cdot 2^k$ wie folgt um:

$$\begin{aligned}
|y|_4 \cdot 2^k &= |y|_4 \cdot 2^{2l} = |y|_4 \cdot 4^l \\
&= \left(y_n \cdot 4^n + y_{n-1} \cdot 4^{n-1} + \cdots + y_{n-l+1} \cdot 4^{n-l+1} + y_{n-l} \cdot 4^{n-l} + \cdots + y_0\right) \cdot 4^l \\
&= y_n \cdot 4^{n+l} + y_{n-1} \cdot 4^{n+l-1} + \cdots + y_{n-l+1} \cdot 4^{n+1+l-l} + y_{n-l} \cdot 4^{n+l-l} + \cdots + y_0 \cdot 4^l \\
&= y_n \cdot 4^{n+l} + y_{n-1} \cdot 4^{n+l-1} + \cdots + y_{n-l+1} \cdot 4^{n+1} + y_{n-l} \cdot 4^n + \cdots + y_0 \cdot 4^l
\end{aligned}$$

Es gilt nun mit $m = 4^n + 1$:

$$4^n \equiv 4^n + 1 - 1 \equiv (4^n + 1) - 1 \equiv -1 \bmod m$$

Wir setzen dies nun wie folgt in die obige Formel ein:

$$\begin{aligned}
|y|_4 \cdot 2^k &= -y_n \cdot 4^l - y_{n-1} \cdot 4^{l-1} - \cdots - y_{n-l+1} \cdot 4^1 + y_{n-l} \cdot 4^n + \cdots + y_0 \cdot 4^l \\
&= |(0, \ldots, 0, -y_n, -y_{n-1}, \ldots, -y_{n-l+1}, 0)|_4 + |(y_{n-l}, y_{n-l-1}, \ldots, y_0, \ldots, 0)|_4
\end{aligned}$$

Die Multiplikation mit einer Zweierpotenz 2^{2l} kann somit als Addition der Radix-4 Zahlen $(0, \ldots, 0, -y_n, -y_{n-1}, \ldots, -y_{n-l+1}, 0)$ und $(y_{n-l}, y_{n-l-1}, \ldots, y_0, \ldots, 0)$ realisiert werden und ist somit in Größe $O(n)$ und Tiefe $O(1)$ möglich.

3.10 Schulmethode für die Division

Frage 3.32: Welche Darstellungen für rationale Zahlen kennst du? Welche wird in Rechnern benutzt?

Man kann zwei Darstellungen von rationalen Zahlen unterscheiden. Beide Darstellungen haben jedoch Nachteile, aufgrund derer sie in der Praxis nicht verwendet werden. Die erste Darstellung ist die in Form eines Bruchs, d.h. als ein Paar von Zähler und Nenner. Ein Nachteil dieser Darstellung ist, dass wir zur Addition zweier Brüche eine Multiplikation benötigen, um den Hauptnenner zu berechnen. Also kennen wir schon für die Addition zweier Brüche keinen Schaltkreis linearer Größe. Wenn wir nicht mit unnötig großen Zahlen rechnen wollen, müssen wir desweiteren in der Lage sein, zu kürzen, wofür keine effizienten Schaltkreise bekannt sind. Die zweite Darstellung ist die in Form einer unter Umständen periodischen Binär- bzw. Dezimaldarstellung. Auch für diese sind keine linearen Additionsschaltkreise bekannt. Aufgrund der erwähnten Nachteile wird in der Praxis eine endliche Binärdarstellung verwendet, bei der es zu Rundungsfehlern kommen kann.

Frage 3.33: Wie ist die endliche Binärdarstellung für rationale Zahlen definiert?

Die endliche Binärdarstellung für rationale Zahlen ist wie folgt definiert. Gegeben sei ein Vektor $x = (x_{n-1}, \ldots, x_0, x_{-1}, \ldots, x_{-m}) \in \{0,1\}^{n+m}$. Dann ist $|x|_{-m}$ definiert als:

$$|x|_{-m} = x_{n-1}2^{n-1} + \cdots + x_0 2^0 + x_{-1}2^{-1} + \cdots + x_{-m}2^{-m}$$

Die normale Binärdarstellung ist nach dieser Definition also als $|x|_0$ gegeben. Der Index $-m$ kann als die Genauigkeit betrachtet werden, mit der wir rechnen, wenn wir uns m Nachkommastellen gönnen. Die Genauigkeit beträgt dann 2^{-m}, d.h. die Differenz zwischen der Zahl, die wir eigentlich darstellen wollen und ihrer endlichen Binärdarstellung beträgt höchstens 2^{-m}. Dies kann man sich wie folgt klarmachen. Uns fehlt der Betrag der Stellen von $x_{-(m+1)}$ bis $x_{-\infty}$. Wenn dieses alle 1 sind, fehlt uns also:

$$\begin{aligned}\sum_{i=m+1}^{\infty}\left(\frac{1}{2}\right)^i &= \sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i - \sum_{i=0}^{m}\left(\frac{1}{2}\right)^i = \frac{1}{1-\frac{1}{2}} - \frac{\left(\frac{1}{2}\right)^{m+1}-1}{\frac{1}{2}-1} = 2 + 2\cdot\left(\left(\frac{1}{2}\right)^{m+1} - 1\right)\\ &= 2\cdot\left(\frac{1}{2}\right)^{m+1} = \left(\frac{1}{2}\right)^m = 2^{-m}\end{aligned}$$

So ergibt sich bei der Darstellung als $|x|_0$ also eine Genauigkeit von 1. Dies kann man sich z.B. klarmachen, indem wir $1.\overline{9}$ als 1 darstellen. Der Fehler ist höchstens 1.

Frage 3.34: Wie ist die Division definiert?

Die Division DIV ist definiert als die Folge partiell definierter Funktionen $DIV_n \in B^*_{2n,n}$, die aus $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$ mit $|y| \neq 0$ die Binärdarstellung $q = (q_{n-1}, \ldots, q_0)$ des ganzzahligen Quotienten $\lfloor |x|/|y| \rfloor$ berechnen.

Frage 3.35: Warum genügt es, sich auf die ganzzahlige Division zu beschränken?

In der obigen Definition der Division DIV_n wird der ganzzahlige Quotient $\lfloor |x|/|y| \rfloor$ berechnet. Dies reicht auch aus. Denn zur Erhöhung der Genauigkeit brauchen wir nur die Eingabe x mit 2^k zu multiplizieren (also um k Stellen nach links shiften) und anschließend im Ergebnis das Komma um k Stellen zu verschieben. Wir verdeutlichen dies an einem kleinen Beispiel. Wenn wir sofort den Wert von $\frac{26}{35}$ berechnen, erhalten wir als Ergebnis der ganzzahligen Division den Wert 0, da das Ergebnis nach unten gerundet wird. Wenn wir den Zähler jedoch zuerst mit 10000 multiplizieren und das Ganze durch eine Division durch 10000 wieder korrigieren (Komma verschieben), so erhalten wir:

$$\frac{26}{35} = \frac{1}{10000}\cdot\frac{26\cdot 10000}{35} = \frac{1}{10000}\cdot\frac{260000}{35} = \frac{1}{10000}\cdot 7428 = 0.7428$$

Mit der im Anschluss vorgestellten Schulmethode der Division kann man sich dieses Vorgehen so klarmachen, dass der Algorithmus durch die Vergrößerung von x länger läuft und dadurch genauer wird.

Frage 3.36: Wie funktioniert die Schulmethode für die Division zweier Binärzahlen? Welche Größe und Tiefe hat sie?

Die Schulmethode für die Division zweier Binärzahlen funktioniert so ähnlich wie die Schulmethode für die Division zweier Dezimalzahlen und wird in Algorithmus 10 beschrieben: Da das Ergebnis in jedem Schritt des Algorithmus vom Ergebnis des vorherigen Schritt abhängt, kann er nicht sehr gut parallelisiert werden. Lediglich die Vergleiche und Subtraktionen in jedem Schritt können parallelisiert werden. Jede Subtraktion hat eine Tiefe von $O(\log n)$ und da wir n Schritte machen, beträgt die Gesamttiefe $O(n \log n)$. Die Größe beträgt $O(n^2)$, denn jede Subtraktion hat mindestens Größe $O(n)$.

Frage 3.37: Wie lösen die anderen Verfahren das Problem der Division?

Algorithmus 10 Die Schulmethode für die Division zweier Binärzahlen

Führe folgende Schritte für $i = n-1, \ldots, 0$ durch.

1. Vergleiche die Binärzahlen $x' = (x_{n-1}, \ldots, x_i)$ und $y = (y_{n-1}, \ldots, y_0)$.
2. Falls $|x'| < |y|$, setze $q_i = 0$.
3. Falls $|x'| \geq |y|$, setze $q_i = 1$ und ersetze $(x_{n-1}, \ldots, x_i)$ durch die Differenz von x' und y.

Die beiden anderen Verfahren lösen das Problem der Division durch die Berechnung des Inversen des Divisors. Denn wenn wir x durch z dividieren wollen, ist dies dasselbe als wenn wir x mit z^{-1} multiplizieren.

Frage 3.38: Wie ist die Funktion INV definiert?

INV ist die Folge Boolescher Funktionen $INV_n \in B_{n-1,n}$, die aus $z = (1, z_{-2}, \ldots, z_{-n})$ eine von $|z|_{-n}^{-1}$ um höchstens 2^{-n+1} abweichende Approximation $y = (1, y_{-1}, \ldots, y_{-n})$ berechnen. Da bei z die Stelle $z_{-1} = 1$ gesetzt ist und bei y die Stelle $y_0 = 1$ gesetzt ist, gilt also $\frac{1}{2} \leq z < 1$ und $1 \leq y < 2$.

Frage 3.39: Wie können wir dafür sorgen, dass eine Eingabe eine gültige Eingabe für INV ist?

Damit $z = (z_{-1}, \ldots, z_{-n})$ eine gültige Eingabe für *INV* ist, müssen wir dafür sorgen, dass $z_{-1} = 1$ ist. Multiplikationen mit Zweierpotenzen sind kostenlos, da es sich nur um das Anhängen von Nullen bzw. das Shiften der Eingabe handelt. Wir müssen jedoch erst einmal herausbekommen, mit welcher Zweierpotenz wir z multiplizieren müssen, damit die Bedingung $z_{-1} = 1$ erfüllt ist. Dazu suchen wir quasi von links nach rechts die erste Eins in z bzw. die letzte Stelle bis zu der nur Nullen gelesen werden. Gilt zu Beginn $z_{-1} = 1$ müssen wir gar nicht bzw. mit 2^0 multiplizieren. Ist die Stelle z_{-i} die letzte Stelle für die von links gesehen $z_{-i} = 0$ gilt, müssen wir z mit 2^i multiplizieren, damit die Bedingung $z_{-1} = 1$ gegeben ist. Wir setzen $z_0 = 0$ damit diese Ansichtsweise allgemein gilt.

Damit eine Stelle i die letzte Stelle ist, bis zu der nur Nullen gelesen werden, müssen alle Stellen $0 \leq j \leq i$ nur Nullen enthalten und die nächste Stelle $i+1$ eine 1. Mit u_i wollen wir nun die Funktion bezeichnen, welche angibt, ob bis zur i-ten Stelle nur Nullen in z vorhanden sind. In diesem Fall soll $u_i = 1$ gelten. Dann ist u_i also gegeben als $u_i = \overline{z}_0 \wedge \cdots \wedge \overline{z}_{-i}$. Dabei setzen wir $z_0 = 0$. Nun wollen wir eine weitere Funktion v definieren, die nur dann 1 wird, wenn die i-te Stelle die letzte Null ist. In diesem Fall müssen wir die Eingabe um i Bits nach rechts shiften bzw. mit 2^i multiplizieren. Es gilt $v_i = u_i \wedge z_{-(i+1)}$, denn alle Stellen bis einschließlich i dürfen nur Nullen enthalten und die Stelle rechts von i muss eine 1 enthalten. Gilt für eine Stelle $v_i = 1$, so gilt aufgrund der Definition für alle anderen Stellen $v_j = 0$ mit $j \neq i$. Wir sind also an allen Werten v_i interessiert. Da die Verknüpfung $\wedge$ eine assoziative, binäre Verknüpfung ist, handelt es sich bei der Berechnung aller u_i um das Präfixproblem, welches wir mit Algorithmus 3 aus Kapitel 3.5 in Größe $O(n)$ und Tiefe $O(\log n)$ lösen können. Im Anschluss berechnen wir die v_i, indem wir jedes Ergebnis u_i der Präfixberechnung mit $z_{-(i+1)}$ verknüpfen. Aus den berechneten v_i bauen wir die Zahl $(v_{n-1}, \ldots, v_0)$ mit der wir z multiplizieren, um eine neue Zahl z' zu bekommen. Für diese Zahl z' gilt dann $\frac{1}{2} \leq |z'|_{-n} < 1$. Für diese Zahl z' berechnen

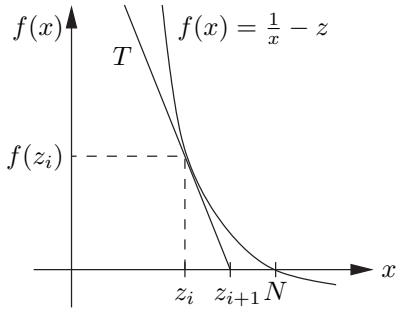


Abbildung 13: Die Funktion $f(x) = \frac{1}{x} - z$

wir nun das Inverse und multiplizieren dies erneut mit $(v_{n-1}, \ldots, v_0)$ um eine Approximation für $|z|_{-n}^{-1}$ zu erhalten.

3.11 Newtonmethode für die Division

Frage 3.40: Wie funktioniert die Newtonmethode für die Division?

Das Newtonverfahren ist ein Verfahren zur Berechnung von Nullstellen. Damit das Verfahren funktioniert, muss die Funktion konvex, monoton fallend und differenzierbar sein. Doch was hat die Berechnung von Nullstellen überhaupt mit der Division bzw. mit der Berechnung des Inversen zu tun? Wir wollen im Folgenden wieder da Inverse z^{-1} zu einer Zahl z berechnen, damit wir die Rechnung $\frac{x}{z}$ auf die Multiplikation xz^{-1} zurückführen können. Wir wollen nun erreichen, dass z^{-1} die Nullstelle einer konvexen, monoton fallenden und differenzierbaren Funktion $f(x)$ ist. Denn in diesem Fall können wir das Newtonverfahren benutzen, um die Nullstelle zu berechnen und so z^{-1} zu bekommen. Die Funktion, die genau diese Eigenschaften hat lautet:

$$f(x) = \frac{1}{x} - z$$

Es ist leicht einzusehen, dass $f(z^{-1}) = 0$ gilt: $f(z^{-1}) = \frac{1}{z^{-1}} - z = z - z = 0$. Da die Funktion konvex und monoton fallend ist, ist z^{-1} auch die einzige Nullstelle dieser Funktion. Im Folgenden bezeichnen wir mit N den Wert der Nullstelle von f. Wenn wir nun davon ausgehen, dass ein Approximation der Nullstelle mit $z_i \leq N$ gegeben ist, dann können wir den Wert $f(z_i)$ berechnen und an diesem Punkt die Tangente mit der Steigung $f'(z_i)$ anlegen. Da die Funktion f konvex und monoton fallend ist, liegt die Tangente unterhalb von f und die Nullstelle z_{i+1} der Tangente im Intervall $]z_i, N]$. Damit ist z_{i+1} eine bessere Approximation der Nullstelle N. Abbildung 13 verdeutlicht noch einmal diese Idee.

Wir überlegen uns nun, wie wir die Tangentengleichung $T(x)$ bekommen und wie sich die Nullstelle von $T(x)$ berechnet. Wir wissen, dass $T(z_i) = f(z_i)$ und dass $T'(x) = f'(z_i)$ ist. Damit bekommen wir die Geradengleichung von $T(x)$ als:

$$T(x) = (x - z_i)f'(z_i) + f(z_i) = x \cdot f'(z_i) - z_i \cdot f'(z_i) + f(z_i)$$

Wir sind an der Nullstelle z_{i+1} interssiert, also muss $T(x) = 0$ gelten. Dies ist der Fall für:

$$x = \frac{z_i \cdot f'(z_i) - f(z_i)}{f'(z_i)} = z_i - \frac{f(z_i)}{f'(z_i)}$$

Es gilt somit:

$$z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)}$$

Zu zeigen bleibt, dass die Folge der z_i mit wachsendem i gegen die Nullstelle von f konvergiert. Wir wissen, dass die Folge der z_i monoton wachsend und durch N beschränkt ist. Damit konvergiert die Folge z_i. Wir zeigen nun, dass sie auch tatsächlich gegen die Nullstelle N von f konvergiert. Dass die Folge der z_i konvergiert, bedeutet, dass der Abstand zwischen benachbarten Folgegliedern jeweils ab einem bestimmten i beliebig klein wird. Damit konvergiert also $z_i - z_{i+1}$ gegen 0. Wir stellen obige Formel nach $z_i - z_{i+1}$ um und erhalten:

$$z_i - z_{i+1} = \frac{f(z_i)}{f'(z_i)}$$

Dann konvergiert jedoch auch die Folge der $f(z_i)$ gegen 0 und somit z_i gegen die Nullstelle N.

Mit der obigen Formel für z_{i+1} wissen wir also, wie sich ein Folgeglied aus seinem Vorgänger berechnet. Dummerweise kommt in dieser Formel jedoch eine Division vor, also das Problem, welches wir mit diesem Verfahren ja gerade lösen wollen. Glücklicherweise fällt diese Division jedoch weg, wenn wir $f(z_i)$ und $f'(z_i)$ durch die entsprechenden Formeln ersetzen. Es gilt ja:

$$f(z_i) = \frac{1}{z_i} - z \qquad \text{und} \qquad f'(z_i) = -\frac{1}{z_i^2}$$

Setzen wir dies oben ein, so ergibt sich:

$$z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)} = z_i - \frac{\frac{1}{z_i} - z}{-\frac{1}{z_i^2}} = z_i + z_i^2 \cdot \left(\frac{1}{z_i} - z\right) = z_i + z_i - z z_i^2 = 2z_i - z z_i^2$$

Somit können wir z_{i+1} also aus z und z_i mit Hilfe von einer kostenlosen Multiplikation mit 2, zwei Multiplikationen zur Berechnung von $z z_i^2$ und einer Subtraktion berechnen. Für die Subtraktion spendieren wir uns einen Ladner/Fisher Addierer. In diesem Fall wird die Komplexität eines Approximationsschrittes nur noch von den Kosten der Multiplikation bestimmt.

Frage 3.41: Welche Konvergenzgeschwindigkeit hat das Newtonverfahren zur Berechnung des Inversen einer Zahl?

Wir wollen nun herausfinden, wie schnell sich die Folge z_i der tatsächlichen Nullstelle N annähert. Aus dieser Geschwindigkeit können wir dann folgern, nach wievielen Schritten wir die gewünschte Genauigkeit für die Approximation von z^{-1} erreichen und das Verfahren abbrechen können. Mit ε_i bezeichnen wir den Fehler, den wir mit z_i machen, d.h. die Größe der Abweichung zwischen z^{-1} und z_i. Dementsprechend gilt also $\varepsilon_i = z^{-1} - z_i$ bzw. nach z_i umgestellt: $z_i = z^{-1} - \varepsilon_i$. Desweiteren werden wir $z_{i+1} = 2z_i - z z_i^2$ benötigen. Das Ziel ist es, ε_{i+1} in Abhängigkeit von ε_i auszudrücken. Es gilt:

$$\begin{aligned}
\varepsilon_{i+1} &= z^{-1} - z_{i+1} = z^{-1} - \left(2z_i - z z_i^2\right) = z^{-1} - 2z_i + z z_i^2 \\
&= z^{-1} - 2 \cdot \left(z^{-1} - \varepsilon_i\right) + z \cdot \left(z^{-1} - \varepsilon_i\right)^2 = z^{-1} - 2z^{-1} + 2\varepsilon_i + z \cdot \left(z^{-2} - 2z^{-1}\varepsilon_i + \varepsilon_i^2\right) \\
&= -z^{-1} + 2\varepsilon_i + z^{-1} - 2\varepsilon_i + z\varepsilon_i^2 = z\varepsilon_i^2 < \varepsilon_i^2
\end{aligned}$$

Die letzte Abschätzung gilt aufgrund der Voraussetzung $z < 1$. Das Newtonverfahren konvergiert also quadratisch gegen die Nullstelle N und damit gegen den Kehrwert z^{-1}.

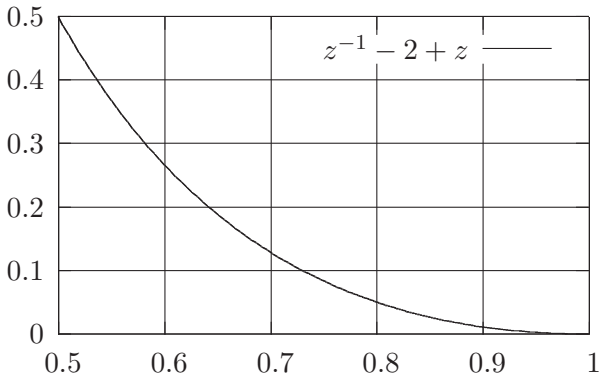


Abbildung 14: Plot der Funktion $z^{-1} - 2 + z$ mit Maximalwert für $z = \frac{1}{2}$

Wieviele Schritte müssen wir nun machen, damit die Genauigkeit 2^{-n+1} beträgt? Es gilt $\frac{1}{2} \leq z < 1$. Für den „echten“ Kehrwert z^{-1} gilt dann also $1 < z^{-1} \leq 2$. Wenn wir nun mit $z_0 = 1$ beginnen, sind wir somit schon „recht nah“ am richtigen Ergebnis. Wie groß ist der Fehler ε_0 dann höchstens? Es gilt $\varepsilon_0 = z^{-1} - z_0 \leq 2 - 1 = 1$. Zu Beginn beträgt der Fehler höchstens 1. Für den zweiten Wert z_1 müssen wir uns noch „von Hand“ überlegen, wie groß der Fehler ε_1 höchstens sein kann, danach können wir das Ergebnis $\varepsilon_{i+1} < \varepsilon_i^2$ benutzen. Es gilt $\varepsilon_1 = z^{-1} - z_1$ und, da sich z_1 als $z_1 = 2z_0 - zz_0^2$ berechnet, $\varepsilon_1 = z^{-1} - (2z_0 - zz_0^2) = z^{-1} - 2z_0 + zz_0^2$. Wir wissen, dass $z_0 = 1$ ist, womit sich $\varepsilon_1 = z^{-1} - 2 + z$ ergibt. Für welches $\frac{1}{2} \leq z < 1$ wird ε_1 maximal? Aus dem Plot in Abbildung 14 wird schnell ersichtlich, dass dies für $z = \frac{1}{2}$ bzw. $z^{-1} = 2$ der Fall ist. Somit gilt $\varepsilon_1 \leq 2 - 2 + \frac{1}{2} = \frac{1}{2}$. Für die nächsten Fehlerwerte gilt dann:

$$\varepsilon_2 < \left(\frac{1}{2}\right)^2 = \frac{1}{2^2}, \quad \varepsilon_3 < \left(\frac{1}{2^2}\right)^2 = \frac{1}{2^{2\cdot 2}} = \frac{1}{2^{2^2}}, \quad \ldots, \quad \varepsilon_i < \frac{1}{2^{2^{i-1}}} = 2^{-2^{i-1}}$$

Wir wollen das Ergebnis nach der n-ten Stelle abschneiden, daher wollen wir nun erst eine Genauigkeit von 2^{-n} erreichen. Für welches k ist der Fehler also kleiner als 2^{-n}?

$$2^{-2^{k-1}} \leq 2^{-n} \quad \Leftrightarrow \quad -2^{k-1} \leq -n \quad \Leftrightarrow \quad 2^{k-1} \geq n \quad \Leftrightarrow \quad k \geq \log n + 1$$

Für $k = \lceil \log n \rceil + 1$ erreichen wir also eine Genauigkeit von 2^{-n}. Wenn wir das Ergebnis nun nach der n-ten Stelle abschneiden, kommt ein weiterer Fehler von höchstens 2^{-n} hinzu. Insgesamt also ein Fehler von $2^{-n} + 2^{-n} = 2 \cdot 2^{-n} = 2^{-n+1}$. Dies ist die Genauigkeit, die in der Definition von *INV* gefordert wird.

Frage 3.42: Welchen Nachteil hat das „unbehandelte“ Newtonverfahren zur Berechnung des Inversen einer Zahl? Was kann dagegen unternommen werden?

Der Nachteil des bisher beschriebenen Newtonverfahrens liegt darin, dass in jedem Schritt bei der Berechnung eines z_i unter anderem drei Zahlen multipliziert werden. Die Multiplikation ist eine Funktion aus $B_{2n,2n}$, d.h. sie bekommt zwei Zahlen der Länge n und erzeugt daraus eine neue Zahl der Länge $2n$. Bei der Berechnung von zz_i^2 kommt also eine Zahl heraus, die mindestens doppelt so groß ist wie die ursprünglichen Zahlen. Wenn wir mit $L(i)$ die Länge der Zahlen im i-ten Schritt bezeichnen, gilt somit $L(0) \leq n = 2^0 \cdot n$ und allgemein $L(i) \leq 2^i \cdot n$.

Nach $\lceil \log n \rceil + 1$ Schritten gilt somit:

$$L(\lceil \log n \rceil + 1) \leq 2^{\lceil \log n \rceil + 1} \cdot n \leq 2^{(\log n + 1) + 1} \cdot n = 2^{\log n + 2} \cdot n = 2^2 \cdot n \cdot n = 4n^2 = O(n^2)$$

Am Schluss rechnen wir also ungefähr mit quadratisch großen Zahlen, was das Verfahren ineffizient bezüglich der Größe macht. Es kann jedoch gezeigt werden, dass es ausreicht, stets mit Zahlen der Länge $2n$ weiter zu rechnen. Das bedeutet, dass von jedem Zwischenergebnis nur die ersten $2n$ Stellen benutzt werden und der Rest abgeschnitten wird. Damit ergibt sich ein Fehler von höchstens 2^{-2n}. Das Verfahren benötigt dann doppelt so viele Schritte wie das ursprüngliche Verfahren, also immer noch $O(\log n)$ Schritte.

Frage 3.43: Welche Größe und Tiefe ergibt sich nach obigen Ergebnissen für einen Divisionsschaltkreis? Gibt es Verbesserungsmöglichkeiten?

Wir haben schon beobachtet, dass die Größe und Tiefe eines Divisionsschaltkreises im wesentlichen durch die Größe und Tiefe des verwendeten Multiplikationsschaltkreises bestimmt wird. Ist $M(n)$ die Größe und $D(n)$ die Tiefe des verwendeten Multiplikationsschaltkreises, so hat der Divisionsschaltkreis eine Größe von $O(\log n \cdot M(n))$ und eine Tiefe von $O(\log n \cdot D(n))$, da wir $O(\log n)$ Schritte benötigen.

Die Größe lässt sich jedoch noch verbessern, wobei hier nur kurz die Idee und die Voraussetzungen angesprochen werden. Die Voraussetzung, die ein Multiplikationsschaltkreis erfüllen muss, damit er sich für die Verbesserung der Größe eignet, ist die folgende:

$$M\left(\frac{n}{m}\right) = O\left(\frac{M(n)}{m}\right)$$

Was bedeutet dies umgangssprachlich? Wann erfüllt ein Multiplikationsverfahren diese Forderung? Gegeben sei ein Multiplikationsschaltkreis für Eingaben der Länge n. Dann hat dieser die Größe $M(n)$. Wenn wir nun einen Schaltkreis für Eingaben bauen, deren Länge nur $\frac{1}{m}$ der Länge n ist, dann muss die Größe des Schaltkreises auch ungefähr $\frac{1}{m}$ der Größe betragen, die ein Schaltkreis für Eingaben der Länge n hat. Das Multiplikationsverfahren muss also skalierbar sein. Wir zeigen nun, dass z.B. die Schulmethode der Multiplikation diese Eigenschaft besitzt. Die Größe der Schulmethode beträgt $O(n^2)$. Für Eingaben der Länge $\frac{n}{m}$ beträgt die Größe also $O(n^2/m^2)$. Die Frage ist nun, ob folgende Aussage gilt:

$$\frac{n^2}{m^2} = O\left(\frac{n^2}{m}\right)$$

Für $m \geq 1$ gilt die Aussage, denn wir können $\frac{n^2}{m^2}$ auch schreiben als $\frac{1}{m} \cdot \frac{n^2}{m}$. Damit ist $\frac{n^2}{m^2}$ also immer durch $\frac{n^2}{m}$ nach oben beschränkt. Für die Methode von Karatsuba-Ofman kann ebenfalls gezeigt werden, dass sie diese Voraussetzung erfüllt.

Die Idee besteht nun darin, zu Beginn des Verfahrens mit Zahlen kurzer Länge zu rechnen, da die die Fehler, die zu Beginn des Verfahrens gemacht werden, eh groß sind. Zu Beginn des Verfahrens rechnen wir also etwas ungenau bzw. suchen uns quasi grob den Bereich, in dem das Inverse ungefähr liegen muss. Erst gegen Ende der Rechnung wird mit genauen Werten gerechnet, da für die Größe des Schaltkreises im wesentlichen die letzten Stufen entscheidend sind. Auf diese Art und Weise ist es möglich, Divisionsschaltkreise mit Größe $O(M(n))$ und Tiefe $O(\log n \cdot D(n))$ zu entwerfen.

3.12 Die IBM-Methode

Im Folgenden wird nur die Idee der IBM-Methode vorgestellt, da die Beweise der Korrektheit bzw. die Überlegungen, wie genau wir rechnen müssen, relativ umständlich und sehr mathematisch sind und daher auch in einer Prüfung hoffentlich nicht vorkommen.

Frage 3.44: Welche Idee benutzt die IBM-Methode zur Berechnung des Inversen einer Zahl?

Die IBM-Methode ist ein weiteres Verfahren zur Berechnung des Inversen z^{-1} einer Zahl z. Auch hier sei die Forderung $\frac{1}{2} \leq z < 1$ erfüllt. Die Idee der IBM-Methode besteht nun darin, $z^{-1} = \frac{1}{z}$ so zu erweitern, dass der Nenner des Bruchs sehr nah bei 1 liegt. Im Zähler muss dann eine Approximation des Wertes des Inversen selbst stehen (wegen $z^{-1} = \frac{z^{-1}}{1}$). Dazu berechnen wir zuerst den Wert $x = 1 - z$, d.h. es gilt $0 < x \leq \frac{1}{2}$. Dann können wir z auch schreiben als $z = 1 - x$ und das Inverse von z als:

$$z^{-1} = \frac{1}{z} = \frac{1}{1-x}$$

Wie müssen wir diesen Bruch nun erweitern, so dass der Nenner nahe bei 1 liegt? Wir werden dazu den dritten binomischen Satz ausnutzen, der besagt, dass $(a-b)(a+b) = a^2 - b^2$ ist. Wenn wir den Bruch also mit $(1+x)/(1+x)$ erweitern, dann steht anschließend $(1-x)(1+x) = 1^2 - x^2 = 1 - x^2$ im Nenner. Was müssen wir nun machen, damit der Nenner sich gegen 1 entwickelt? Wir sollten dafür sorgen, dass das, was von der Eins abgezogen wird, möglichst schnell klein wird. Also erweitern wir im nächsten Schritt mit $(1+x^2)/(1+x^2)$ und erhalten so $(1-x^2)(1+x^2) = 1-x^4$ im Nenner. Allgemein erweitern wir nach dem folgenden Schema, wobei $x(j) = x^{2^j}$ gelte:

$$z^{-1} = \frac{1}{1-x} \cdot \frac{1+x(0)}{1+x(0)} \cdot \frac{1+x(1)}{1+x(1)} \cdot \frac{1+x(2)}{1+x(2)} \cdot \cdots \cdot \frac{1+x(k)}{1+x(k)} = \frac{P_k(x)}{1-x(k+1)} \approx P_k(x)$$

Der Nenner $1 - x(k+1)$ ergibt sich durch fortgesetzte Anwendung des dritten binomischen Satzes. Da $0 < x \leq \frac{1}{2}$ gilt, strebt der Ausdruck $x(k+1)$ für wachsendes k somit gegen 0 und der gesamte Nenner gegen 1. Interessiert sind wir also an $P_k(x)$, welches sich durch den Ausdruck im Zähler ergibt:

$$P_k(x) = (1+x(0)) \cdot (1+x(1)) \cdot (1+x(2)) \cdot \cdots \cdot (1+x(k)) = \prod_{i=0}^{k} (1+x(i))$$

Aus dieser Formel ergibt sich ziemlich direkt der Algorithmus zur Berechnung von $P_k(x)$. Es sei dabei angemerkt, dass $x(j+1) = x(j)^2$ gilt. Wir können den Wert $x(j+1)$ also berechnen, indem wir den Wert $x(j)$ mit sich selbst multiplizieren. Da wir den Startwert $x(0) = x$ kennen, können wir also auf diese Art und Weise alle restlichen Werte berechnen.

Algorithmus 11 beschreibt die Berechnung von $P_k(x)$ algorithmisch. Wie schnell ersichtlich ist, werden in jedem Schritt zwei Multiplikationen benötigt. Mit der ersten wird $x^*(j)$ aus $x^*(j-1)$ berechnet und mit der zweiten $P^*_{j-1}(x)$ mit $1 + x(j)$ multipliziert. Zu Beginn der Rechnung wird zusätzlich eine Subtraktion benötigt, um $x(0) = x = 1 - z$ zu berechnen.

Frage 3.45: Welche Größe und Tiefe hat die IBM-Methode zu Berechnung des Inversen?

Die Parameter k und s in Algorithmus 11 geben an, wieviele Iterationen (und damit Multiplikationen) gemacht werden und wie groß die Zahlen sind, mit denen gerechnet wird. Mit

Algorithmus 11 Die IBM-Methode zur Berechnung des Inversen

Der Parameter $k \in \mathbb{N}$, der die Anzahl der Iterationen angibt und der Parameter $s \in \mathbb{N}$, welcher die Genauigkeit bestimmt, seien gegeben. Abrunden bedeutet, die Zahl nach der s-ten Nachkommastelle zu runden.

1. Es sei $x^*(0) = x(0) = x$. Für $1 \leq j \leq k$ berechnen wir $x^*(j)$ indem wir $i^*(j-1)$ quadrieren und das Ergebnis abrunden.
2. Es gilt $x^*(0) \leq \frac{1}{2}$, womit $x^*(j) \leq \frac{1}{2}$ für alle $1 \leq j \leq k$ gilt. Damit verursacht die Rechnung $1 + x^*(j)$ keine Kosten, da nur $x_0 = 1$ gesetzt werden muss.
3. Es ist $P_0^*(x) = 1 + x^*(0)$. Für $1 \leq j \leq k$ wird $P_j^*(x)$ berechnet, indem $P_{j-1}^*(x)$ mit $1 + x^*(j)$ multipliziert und das Ergebnis abgerundet wird.

einigen Rechnungen kann gezeigt werden, dass $k = \lceil \log(n+2) \rceil - 1$ Iterationen benötigt werden, die auf Zahlen der Länge $s = n + 4 + \lceil \log k \rceil = O(n)$ arbeiten. Ist $M(n)$ die Größe des Multiplikationsschaltkreis und $D(n)$ die Tiefe, so hat die IBM-Methode also eine Größe von $O(\log n \cdot M(n))$ und eine Tiefe von $O(\log n \cdot D(n))$.

4 Ordered Binary Decision Diagrams - Eine Datenstruktur für Boolesche Funktionen

4.1 Operationen auf Booleschen Funktionen

Frage 4.1: Welche Eigenschaften sollte eine Datenstruktur für Boolesche Funktionen haben? Welche Operationen sollte sie unterstützen?

Eine Datenstruktur für Boolesche Funktionen sollte für möglichst viele (bzw. für uns interessante) Boolesche Funktionen eine kompakte Darstellung haben. Desweiteren sollte die Manipulation der Booleschen Funktionen effizient möglich sein. Die folgende Liste zeigt wünschenswerte Operationen auf der Datenstruktur:

Auswertung: Gegeben sei eine Darstellung G für die Funktion $f \in B_n$. Wir wollen in der Lage sein, die Funktion auszuwerten, d.h. für eine Eingabe $a \in \{0,1\}^n$ den Funktionswert $f(a)$ zu berechnen.

Erfüllbarkeit: Gegeben sei eine Darstellung G für $f \in B_n$. Gibt es eine Eingabe $a \in \{0,1\}^n$ mit $f(a) = 1$? Oder anders gesagt: Gilt $|f^{-1}(1)| \geq 1$?

Erfüllbarkeit-Anzahl: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne $|f^{-1}(1)|$.

Erfüllbarkeit-Alle: Gegeben sei eine Darstellung G für $f \in B_n$. Gebe alle $a \in \{0,1\}^n$ aus, für die $f(a) = 1$ gilt.

Reduktion/Minimierung: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne eine Darstellung G' für f, deren Größe minimal ist. Falls die minimale Darstellung G' für jede Funktion f definiert ist, heißt diese Operation Reduktion.

Äquivalenztest: Gegeben seien die Darstellungen G_f und G_g für $f, g \in B_n$. Teste, ob $f = g$.

Synthese: Gegeben seien die Darstellungen G_f und G_g für $f, g \in B_n$ und eine Operation $\otimes \in B_2$. Berechne eine Darstellung für $f \otimes g$.

Ersetzung durch Konstanten: Gegeben sei eine Darstellung G für $f \in B_n$, sowie eine Variable x_i und eine Konstante c. Berechne $f_{|x_i=c}$.

Ersetzung durch Funktionen: Gegeben seien die Darstellungen G_f und G_g für $f, g \in B_n$ und eine Variable x_i. Berechne eine Darstellung für

$$f_{|x_i=g} = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$$

Quantifizierung: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i. Berechne eine Darstellung für $(\forall x_i : f) := f_{|x_i=0} \wedge f_{|x_i=1}$ (bzw. für $(\exists x_i : f) := f_{|x_i=0} \vee f_{|x_i=1}$).

Redundanztest: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i. Teste, ob die Variable x_i redundant ist, d.h. ob $f_{|x_i=0} \equiv f_{|x_i=1}$ gilt.

Frage 4.2: Motiviere die Operation „Äquivalenztest“!

Wenn wir einen Schaltkreis für eine bestimmte Funktion entworfen haben, möchten wir gerne wissen, ob er die gewünschte Funktion auch tatsächlich implementiert. Die gewünschte

Funktion liegt dabei oft in Form einer Spezifikation oder eines anderen Schaltkreises vor. Um also zu testen, ob die gewünschte und realisierte Funktion gleich sind, müssen wir beide in die jeweilige Darstellung umwandleln und einen Gleichheitstest durchführen.

Frage 4.3: Motiviere die Operation „Synthese“!
Wenn eine Datenstruktur für Boolesche Funktionen die Operation „Synthese“ unterstützt, kann diese benutzt werden, um einen B_2-Schaltkreis in die entsprechende Darstellung der Datenstruktur umzuwandeln. Dazu wird der Schaltkreis in topologischer Reihenfolge durchlaufen, d.h. jeder Knoten bzw. Baustein wird erst bearbeitet, nachdem alle seine Vorgänger abgearbeitet wurden. Zu Beginn berechnen wir die Darstellung für die Funktionen $x_1, \ldots, x_n$. Im Anschluss werden die Bausteine im topologischer Reihenfolge bearbeitet. Wenn wir dabei einen Baustein C erreichen, an dem die Funktionen f_1 und f_2 mit dem binären Operator $\otimes$ verknüpft werden, müssen wir also die Darstellungen der Funktionen f_1 und f_2, die wir ja vorher berechnet haben, mit dem Operator $\otimes$ verknüpft werden. Genau dies leistet die Synthese.

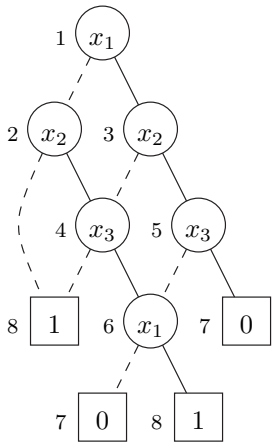
Frage 4.4: Motiviere die Operation „Ersetzung durch Funktionen“!
Die Operation „Ersetzung durch Funktionen“ kann ebenfalls bei der Umwandlung eines Schaltkreises in die Darstellung der entsprechenden Datenstruktur verwendet werden. Der Grund dafür ist der, dass Schaltkreise oft modular aufgebaut sind, d.h. aus Modulen, welche gewisse Funktionen berechnen und deren Ergebnis wieder Eingabe für ein anderes Modul ist. Wenn also z.B. ein Modul eine Funktion $g \in B_n$ berechnet, deren Ergebnis eines der Bits der Eingabe für eine Funktion $f \in B_n$ ist, dann berechnen wir also die Darstellung G_g für die Funktion g und im Anschluss $f_{|x_i=g}$.

Frage 4.5: Motiviere die Operation „Erfüllbarkeit-Alle“!
Wenn ein Schaltkreis nicht die gewünschte Funktion g implementiert sondern eine fehlerhafte Funktion h, dann ist es interessant zu wissen, auf wievielen und auf welchen Eingaben der Schaltkreis etwas Falsches berechnet. Dies bekommen wir heraus, indem wir $(g \oplus h)^{-1}(1)$ berechnen, da $g \oplus h$ nur den Wert 1 liefert, wenn g und h verschiedene Ergebnisse liefern.

Frage 4.6: Welche Darstellungen für Boolesche Funktionen kennst du unter anderem?
Im Prinzip haben wir im Laufe dieses Textes schon einige Darstellungen kennengelernt. Zum einen gibt es die Darstellung als Wertetabelle, welche jedoch für jede Funktion eine exponentielle Größe aufweist. Alternative Darstellungen sind die in Form von Schaltkreisen oder in Form von Polynomen. Diese Darstellungen sind oft sehr kompakt, jedoch ist der Äquivalenztest dieser Darstellungen co-NP-vollständig und das Erfüllbarkeitsproblem NP-vollständig. OBDDs werden sich als ein guter Kompromiss herausstellen, wobei wir hoffen müssen, dass die Umwandlungen in die Darstellung als OBDD für viele für uns interessante Funktionen relativ effizient möglich sind.

4.2 Einführung in OBDDs

Frage 4.7: Was ist ein BDD?
Ein BDD (Binary Decision Diagram) bzw. Branchingprogramm ist ein gerichteter azyklischer Graph bestehend aus einer Quelle (also einem Knoten ohne eingehende Kanten), inneren Knoten mit einem Ausgangsgrad von 2 und Senken mit einem Ausgangsgrad von 0. Die



1: if x_1 then goto 3 else goto 2;
2: if x_2 then goto 4 else goto 8;
3: if x_2 then goto 5 else goto 4;
4: if x_3 then goto 6 else goto 8;
5: if x_3 then goto 7 else goto 6;
6: if x_1 then goto 8 else goto 7;
7: Ausgabe 0;
8: Ausgabe 1;

Abbildung 15: Beispiel für ein BDD

inneren Knoten, zu denen wir auch die Quelle zählen, sind mit einer Variablen markiert. Jeder innere Knoten hat genau zwei ausgehende Kanten, von denen eine mit 0 und die andere mit 1 markiert ist. Die Senken sind ebenfalls mit 0 oder mit 1 markiert.

Abbildung 15 gibt ein Beispiel für ein BDD. In Zeichnungen werden die 0-Kanten in Form von gestrichelten und die 1-Kanten in Form von durchgezogenen Linien gezeichnet. Desweiteren werden die Pfeile an den Kanten weggelassen, da wir davon ausgehen, dass die Kanten von oben nach unten gerichtet sind.

Wir können BDDs auch als eine spezielle Form von Programmen interpretieren, was ebenfalls in Abbildung 15 verdeutlicht wird. Diese Programme haben nur Befehle der Form „if x_i goto y else goto z“ oder „Ausgabe 0(1)“. Die goto-Anweisungen dürfen dabei jedoch nur zu späteren Anweisungen springen, da der resultierende Graph ansonsten nicht azyklisch wäre.

Frage 4.8: Wie wird ein BDD ausgewertet?
Gegeben sei eine Eingabe $a = (a_1, \ldots, a_n) \in \{0,1\}^n$, für die wir wissen möchten, welchen Wert $f(a)$ die durch das BDD dargestellte Funktion f liefert. Die Auswertung beginnt an der Quelle. Sind wir an einem Knoten, der mit der Variablen x_i markiert ist, folgen wir entsprechend der Belegung der Variablen a_i seiner 0-Kante ($a_i = 0$) oder seiner 1-Kante ($a_i = 1$). Dieses Vorgehen wird so lange iteriert, bis eine Senke erreicht wird. Der Funktionswert $f(a)$ entspricht dann der Markierung der Senke.

Frage 4.9: Was ist das wichtigste Komplexitätsmaß für BDDs?
Das wichtigste Komplexitätsmaß für BDDs ist die Anzahl der Knoten. Da jeder Knoten genau zwei ausgehende Kanten hat, ist dies zugleich ein Maß für den Speicherbedarf.

Frage 4.10: Was ist ein OBDD?
Ein OBDD (Ordered Binary Decision Diagram) ist ein BDD bei dem alle Variablen auf jedem Pfad von der Quelle zu einer Senke höchstens einmal und entsprechend einer festgelegten Reihenfolge getestet werden. Es gibt somit eine Permutation $\pi \in S_n$ und für zwei mit x_i und x_j markierte Knoten, zwischen denen eine Kante verläuft, gilt $\pi(i) < \pi(j)$. Durch die Permutation π entsteht also eine Ordnung auf den Variablen. Wir werden noch sehen, dass sich die Variablenordnung in sehr starken Maße auf die Größe eines OBDDs auswirken kann.

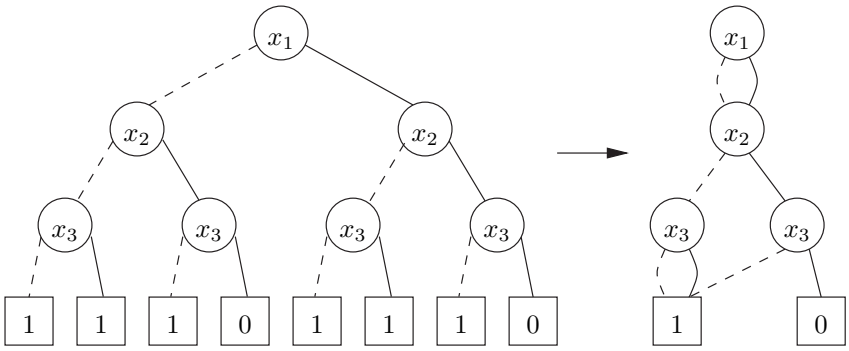


Abbildung 16: Entscheidungsbaum und OBDD für $\overline{x}_2 \vee \overline{x}_3$

Frage 4.11: Zeige, dass jede Boolesche Funktion durch ein OBDD dargestellt werden kann!

Gegeben sei eine Funktion $f(x_1, \ldots, x_n) \in B_n$. Diese wollen wir nun durch ein OBDD darstellen, wobei wir die Variablenordnung $\pi = \text{id} = (x_1, \ldots, x_n)$ wählen. Wir zerlegen f nun mit Hilfe der Shannonzerlegung in $f = \overline{x}_1 f_{|x_1=0} \vee x_1 f_{|x_1=1}$ und erzeugen einen mit x_1 markierten Knoten v. Am 0-Nachfolger von v müssen wir nun die Funktion $f_{|x_1=0}$ darstellen und am 1-Nachfolger die Funktion $f_{|x_1=1}$. Also berechnen wir das OBDD für diese Funktionen rekursiv und lassen den 0-Nachfolger von v auf die Quelle des OBDDs für $f_{|x_1=0}$ zeigen und den 1-Nachfolger entsprechend auf die Quelle des OBDDs für $f_{|x_1=1}$. Dabei sei noch angemerkt, dass $f_{|x_1=0}$ und $f_{|x_1=1}$ nicht mehr von der Variablen x_1 abhängen und wir diese Subfunktionen deshalb als $f(x_2, \ldots, x_n) \in B_{n-1}$ auffassen. Somit tragen die Nachfolgerknoten von v beide die Markierung x_2.

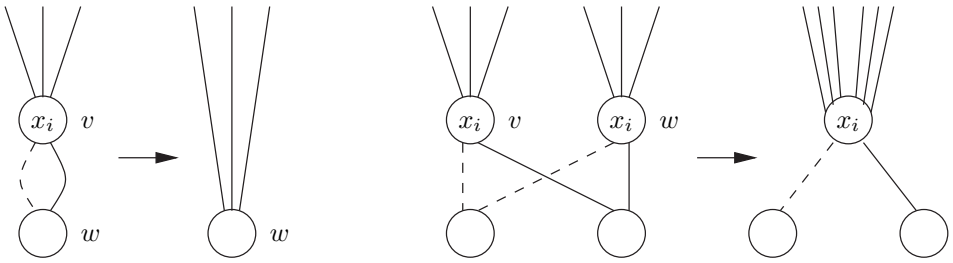
Auf diese Art und Weise entsteht ein sogenannter Entscheidungsbaum, in dem alle Knoten außer der Quelle einen Eingangsgrad von 1 haben. Abbildung 16 zeigt den Entscheidungsbaum, den wir mit diesem Verfahren für die Funktion $f = \overline{x}_2 \vee \overline{x}_3$ erhalten.

Frage 4.12: Welche Reduktionsregeln gibt es für OBDDs?

Wenn wir den Entscheidungsbaum in Abbildung 16 noch einmal betrachten, so fällt auf, dass die beiden Teilbäume, die einen Knoten mit der Markierung x_2 als Wurzel haben, isomorph sind. Dies bedeutet jedoch, dass der weitere Durchlauf durch den OBDD davon unabhängig ist, ob wir die Variable $a_1 = 0$ oder $a_1 = 1$ setzen. Es ist also sinnvoll, isomorphe Teilgraphen zu verschmelzen, d.h. zum Beispiel die beiden Ausgänge von x_1 beide auf einen Teilgraph zeigen zu lassen. Statt zwei gleich aussehende Teilgraphen zu speichern, speichern wir also nur einen Teilgraphen. Zum Verschmelzen von Teilgraphen gibt es zwei sogenannte Reduktionsregeln, welche in Abbildung 17 graphisch dargestellt sind.

Deletion Rule: Diese Reduktionsregel kann benutzt werden, wenn für einen Knoten v mit der Knotenmarkierung x_i der 0-Nachfolger und der 1-Nachfolger auf denselben Knoten w zeigen. In diesem Fall ist das weitere Vorgehen offensichtlich von der Belegung der Variablen x_i unabhängig, so dass wir alle Kanten, die auf den Knoten v zeigen, direkt auf den Nachfolger w richten können. Abbildung 17(a) verdeutlicht die Idee.

Merging Rule: Die Merging Rule kann auf Knoten v und w angewandt werden, die dieselbe Knotenmarkierung x_i haben und deren 0- und 1-Nachfolger übereinstimmen. Da wir denselben Funktionswert erhalten, unabhängig davon, ob wir in v oder w starten, können



(a) Deletion Rule (b) Merging Rule

Abbildung 17: Die beiden Reduktionsregeln für BDDs

wir die Knoten miteinander verschmelzen. Das heißt, wir lassen alle Kanten, die auf v zeigen, auf w zeigen und löschen anschließend den Knoten v. Dieses Vorgehen ist in Abbildung 17(b) festgehalten.

Frage 4.13: Wie kann gezeigt werden, dass für jede Funktion und jede Variablenordnung ein eindeutiges minimales OBDD existiert?

Wir wollen nun zeigen, dass für jede Funktion und Variablenordnung ein eindeutiges minimales OBDD existiert. Ist dies gezeigt, macht es also Sinn, von *dem* minimalen OBDD für eine Funktion und Variablenordnung zu sprechen. Dazu gehen wir jetzt in mehreren Schritten vor. Zuerst machen wir uns Gedanken darüber, was an den einzelnen Knoten eines OBDDs dargestellt wird. Es wird sich zeigen, dass dies spezielle Subfunktionen der an der Quelle des OBDDs repräsentierten Funktion f sind. Diese Subfunktionen können wir dann benutzen, um ein OBDD zu konstruieren, welches jede dieser Subfunktionen an einem Knoten berechnet. Für dieses konstruierte OBDD können wir dann wiederum beweisen, dass es minimal sein muss, das heißt, dass es kein OBDD mit weniger Knoten geben kann. Zu guter Letzt benötigen wir noch ein Kriterium, an dem wir festmachen können, ob ein gegebenes OBDD minimal ist. Es wird sich hier zeigen, dass dies der Fall ist, wenn keine der Reduktionsregeln mehr anwendbar ist.

Frage 4.14: Was wird an einem Knoten in einem OBDD berechnet?

Wir betrachten im Folgenden stets OBDDs mit der Variablenordnung $\pi = \mathrm{id} = (x_1, \ldots, x_n)$, da wir die folgenden Argumente stets durch eine Umbenennung der Variablen in die Argumentation für andere Variablenordnungen überführen können.

Gegeben sei also ein OBDD für eine Funktion f mit π. Wir betrachten nun einen Knoten v, der mit x_i markiert ist. Auf dem Weg von der Quelle zu diesem Knoten dürfen entsprechend der Variablenordnung nur die Variablen $x_1, \ldots, x_{i-1}$ getestet werden. Wenn wir davon ausgehen, dass der Knoten v von der Quelle aus erreichbar ist, dann erreichen wir ihn also, indem wir die Variablen $x_1, \ldots, x_{i-1}$ passend belegen. Dabei kann es mehrere solcher Belegungen geben, die zum Knoten v führen, z.B. $x_1 = c_1, \ \ldots, \ x_{i-1} = c_{i-1}$ oder $x_1 = d_1, \ \ldots, \ x_{i-1} = d_{i-1}$. Da an der Quelle die Funktion f berechnet wird und wir uns auf diesem Weg von der Quelle zum Knoten v jeweils entscheiden, wie wir eine Variable belegen, wird im Knoten v somit die Subfunktion $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ bzw. $f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$ berechnet. Eine wichtige Beobachtung ist nun, dass in jedem Knoten nur eine Subfunktion berechnet werden kann. Denn wir können

den Knoten v einfach als Quelle eines neuen OBDDs betrachten, dessen Funktion nur noch von der Belegung der Variablen $x_i, \ldots, x_n$ abhängt. Wenn nun die Darstellung einer Funktion in einem OBDD nicht eindeutig wäre, könnten wir gleich alle weiteren Überlegungen vergessen, da OBDDs in diesem Fall offensichtlich als Datenstrukturen für Boolesche Funktionen nutzlos wären. Es gilt also $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} \equiv f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$ bzw. jede Belegung der Variablen $x_1, \ldots, x_{i-1}$, die zum Knoten v führt, ergibt dieselbe Subfunktion.

Wieviele verschiedene Knoten mit der Markierung x_i kann es also geben? Mit S_i bezeichnen wir die Menge der Subfunktionen, welche sich ergeben, wenn wir die Variablen $x_1, \ldots, x_{i-1}$ auf alle möglichen Weisen konstant setzen. Dabei zählen wir jedoch nur die Subfunktionen g, die essentiell von x_i abhängen. Für alle anderen Subfunktionen könnte im OBDD die Deletion Rule auf den entstehenden Knoten angewandt werden. Denn dann würde $g_{|x_i=0} \equiv g_{|x_i=1}$ gelten und die 0- und die 1-Kante würden direkt auf denselben Knoten zeigen oder auf zwei Knoten die gemergt werden könnten. Nach einem Merge würden dann beide Kanten auf denselben Knoten zeigen und die Deletion Rule wäre anwendbar. Es gibt somit $|S_i|$ mit x_i markierte Knoten. Wir werden gleich zeigen, dass ein OBDD, welches für jede Subfunktion aus S_i einen Knoten x_i enthält, minimal ist.

Frage 4.15: Wie wird ein OBDD mit Hilfe der oben definierten Subfunktionen konstruiert? Beweise, dass es die darzustellende Funktion auch tatsächlich darstellt!

Gegeben sei also eine Funktion f. Für diese wollen wir nun ein OBDD mit der Variablenordnung $\pi = \mathrm{id} = (x_1, \ldots, x_n)$ konstruieren, welches sich als minimal herausstellen wird. Dazu berechnen wir nun für $i = 1, \ldots, n$ die Mengen S_i. Es sind dann zwei Fälle zu unterscheiden.

Fall 1: Die Funktion f ist eine konstante Funktion. In diesem Fall gilt für $1 \leq i \leq n$, dass $S_i = \emptyset$ ist. Das OBDD besteht also nur aus einer Senke.

Fall 2: Die Funktion f ist keine konstante Funktion und hängt damit von mindestens einer Variablen essentiell ab. Wir konstruieren nun für jede Funktion $g \in S_i$ einen Knoten mit der Markierung x_i (dies machen wir für $1 \leq i \leq n$) sowie eine 0- und eine 1-Senke. Jetzt müssen wir die Knoten nur noch über 0- und 1-Kanten miteinander verbinden. Dabei gehen wir wie folgt vor, wobei v ein Knoten sei, der für die Funktion $g \in S_i$ erzeugt wurde. Wir betrachten nun die Subfunktionen $g_{|x_i=0}$ und $g_{|x_i=1}$. Die Funktion $g_{|x_i=0}$ ist entweder eine konstante Funktion oder eine Funktion aus einer Menge S_j mit $j > i$. In beiden Fällen wurde für $g_{|x_i=0}$ ein Knoten v_0 erzeugt, so dass wir eine 0-Kante von v zu v_0 ziehen können. Entsprechend wird für $g_{|x_i=1}$ argumentiert.

Wir wollen nun zeigen, dass das so konstruierte OBDD auch tatsächlich die Funktion f darstellt. Dazu werden wir zeigen, dass jeder Knoten die Subfunktion berechnet, für die er erzeugt wurde. Da die Quelle mit der Markierung x_1 für die Funktion $f \in S_1$ erzeugt wurde, ist dann gezeigt, dass das OBDD die Funktion f darstellt. Wir führen nun eine Induktion über eine „umgekehrte topologische Ordnung“ durch, d.h. jeder Knoten wird erst dann behandelt, wenn seine Nachfolger behandelt wurden.

Der Induktionsanfang wird für die Senken gemacht. Für diese gilt die Aussage, da an den Senken die konstanten Funktionen 0 und 1 dargestellt werden. Der Induktionsschritt wird nun für einen Knoten v mit der Markierung x_i gemacht, an dem die Funktion g berechnet werden soll. Die 0-Kante führt zu einem Knoten v_0, an dem somit $g_{|x_i=0}$ berechnet werden muss und die 1-Kante zu einem Knoten v_1, an dem $g_{|x_i=1}$ berechnet werden muss. Die Induktionsannahme besteht nun darin, dass $g_{|x_i=0}$ und $g_{|x_i=1}$ korrekt berechnet werden. Dann wird am Knoten

v jedoch die Funktion $\overline{x}_i g_{|x_i=0} \vee x_i g_{|x_i=1}$ berechnet. Entsprechend der Shannon-Zerlegung ist dies die Funktion g.

Frage 4.16: Beweise, dass das konstruierte OBDD minimal ist!

Wir nehmen nun an, dass das so konstruierte OBDD G nicht minimal ist, und zeigen, dass sich ein Widerspruch ergibt. Wenn G nicht minimal ist, muss es also ein OBDD G' geben, welches weniger Knoten als G enthält. Für beide OBDDs gehen wir wieder von der Variablenordnung $\pi = \mathrm{id} = (x_1, \ldots, x_n)$ aus. Das bedeutet jedoch, dass es ein i geben muss, so dass es auf der i-ten Ebene des OBDDs weniger als $|S_i|$ Knoten mit der Markierung x_i gibt. Das OBDD G hat ja auf jeder Ebene i genau $|S_i|$ Knoten. Wenn G' also insgesamt weniger Knoten haben soll, muss es mindestens eine solche Ebene geben.

Es können zwei Fälle unterschieden werden, warum G' weniger als $|S_i|$ Knoten auf der Ebene i hat. Im ersten Fall laufen zwei verschiedene Subfunktionen $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ und $f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$ in einen Knoten. Dies ist ein Widerspruch, da jeder Knoten nur eine Subfunktion darstellen kann. Im anderen Fall müsste für mindestens eine Subfunktion aus S_i die Ebene i übersprungen werden. Wir durchlaufen alle Belegungen $x_1, \ldots, x_{i-1}$. Die Menge S_i ergibt sich ja unabhängig vom betrachteten OBDD. Daher müssen auf diese Weise immer noch $|S_i|$ Subfunktionen erzeugt werden, die von x_i essentiell abhängen. Da es nun aber weniger als $|S_i|$ Knoten mit der Markierung x_i gibt, muss es eine Belegung von $x_1, \ldots, x_{i-1}$ geben, für die ein x_j-Knoten mit $j > i$ erreicht wird. An einem solchen Knoten kann jedoch keine von x_i essentiell abhängige Funktion berechnet werden, da x_i an diesem Knoten entweder schon belegt ist oder die Belegung von x_i an diesem Knoten unter Umständen sogar egal ist. Es ergibt sich also ein Widerspruch und das wie oben beschrieben konstruierte OBDD mit jeweils $|S_i|$ x_i-Knoten muss minimal sein.

Frage 4.17: Zeige, dass das minimale OBDD eindeutig ist!

Wir haben gesehen, dass wir ein minimales OBDD erzeugen können, indem wir die Mengen S_i berechnen und das OBDD wie oben beschrieben konstruieren. Wenn wir die Konstruktion genauer betrachten, so fällt auf, dass wir zu keinem Zeitpunkt eine Wahlmöglichkeit haben. Die Anzahl der x_i-Knoten ergibt sich aus $|S_i|$ und für jede Subfunktion gibt es nur einen Knoten, so dass wir für die Zielknoten der 0- und 1-Kanten keine Wahl haben. Das OBDD muss somit bis auf Isomorphie eindeutig sein.

Frage 4.18: Wann ist ein gegebenes OBDD minimal?

Wir werden nun zeigen, dass ein OBDD G für eine Funktion f und eine Variablenordnung π (in diesem Fall wieder $\pi = (x_1, \ldots, x_n)$) genau dann *dem* minimalen OBDD entspricht, wenn keine Reduktionsregel auf G anwendbar ist. Da es sich um einen „genau dann“-Beweis handelt, sind beide Richtungen zu zeigen.

Das OBDD G sei also das minimale OBDD für f und π. Es folgt aus dieser Eigenschaft direkt, dass keine Reduktionsregel anwendbar sein kann, da G ansonsten nicht minimal sein könnte. Denn jede Reduktion erniedrigt die Anzahl der Knoten um 1. „⇒“

Für die Rückrichtung nehmen wir nun an, dass auf G keine Reduktionsregel mehr anwendbar ist und zeigen, dass daraus folgt, dass G minimal ist. Wir zeigen dazu, dass wenn G nicht minimal ist, eine Reduktionsregel anwendbar ist. Im Beweis zuvor haben wir schon gesehen, dass ein minimales OBDD für eine Funktion f auf der Ebene i genau $|S_i|$ Knoten mit der Markierung x_i hat. Wenn ein OBDD, welches f darstellt, nicht minimal ist, muss es in diesem OBDD also mindestens eine Ebene i geben, welche mehr als $|S_i|$ Knoten enthält. Es sei i^* nun „⇐“

das größte i, für welches dies gilt. Dann werden alle Subfunktionen S_j mit $j > i^*$, also die in den Ebenen darunter, an genau einem Knoten dargestellt bzw. das OBDD ist unterhalb von i^* minimal. Wir können nun zwei Gründe unterscheiden, warum die Ebene i^* mehr als $|S_i|$ x_i-Knoten enthält.

1. Es gibt eine Funktion $g \in S_{i^*}$, die an mehreren x_{i^*}-Knoten dargestellt wird. Wir überlegen uns nun, dass dann auch die 0- und 1-Nachfolger dieser Knoten gleich sein müssen. Am 0-Nachfolger der Knoten wird die Subfunktion $g_{|x_{i^*}=0}$ berechnet. Diese wird also in einer Ebene $j > i^*$ berechnet, die sich unterhalb der Ebene i^* befindet. Unterhalb der Ebene i^* ist das OBDD jedoch, so wie wir i^* gewählt haben, minimal und jede Subfunktion $g \in S_j$ für $j > i^*$ wird in einem Knoten berechnet. Somit müssen die 0-Kanten alle auf denselben Knoten zeigen. Entsprechendes gilt für die 1-Kanten. Somit können wir auf einige der x_{i^*}-Knoten die Merging Rule anwenden.

2. Es gibt einen x_{i^*}-Knoten u, der eine Funktion g berechnet, die nicht essentiell von x_i abhängt. Dann gilt jedoch $g = g_{|x_{i^*}=0} = g_{|x_{i^*}=1}$. Für die Subfunktionen $g_{|x_{i^*}=0}$ bzw. $g_{|x_{i^*}=1}$ gibt es wieder nur einen Knoten, an dem diese Subfunktionen berechnet werden. Also zeigen die 0- und die 1-Kante des Knotens u auf denselben Nachfolger, womit die Deletion Rule anwendbar wäre.

Wir erhalten somit ein minimales OBDD, indem wir die Reduktionsregeln so lange anwenden, bis dies nicht mehr möglich ist.

4.3 OBDD-Darstellungen von ausgewählten Funktionen

Frage 4.19: Wie sehen OBDDs für symmetrische Funktionen aus?

Zur Erinnerung: Bei einer symmetrischen Funktion hängt das Ergebnis der Funktion nur von der Anzahl der Einsen in der Eingabe ab (siehe Kapitel 2.7.2). Ein OBDD für eine symmetrische Funktion zu bilden, ist simpel. Für jeden Eintrag v_i des Wertevektors bilden wir eine Senke s_i, die die Markierung v_i trägt. Eine Senke s_i darf nur erreicht werden, wenn genau i Einsen gelesen werden. Dies ist der Fall, wenn wir das OBDD so wie in Abbildung 18 aufbauen. Die Senke s_0 wird nur erreicht, wenn wir nur nach links weiter gehen und somit nur Nullen lesen. Die Senke s_1 mit der Markierung v_1 wird genau dann erreicht, wenn wir nur eine Eins lesen und ansonsten nur Nullen. Wann wir diese Eins lesen (also für welche Variable) ist egal, sie muss nur genau einmal gelesen werden. Entsprechend wird s_4 nur erreicht, wenn alle vier Variablen auf Eins gesetzt werden.

Ein OBDD, welches wie in Abbildung 18 gebildet wird, kann unter Umständen noch reduziert werden. Ein größeres OBDD wird jedoch nie entstehen. Wie groß kann das OBDD also höchstens werden? Das OBDD für eine symmetrische Funktion über n Variablen $x_1, \ldots, x_n$ hat offensichtlich die folgende Anzahl an Knoten, wobei wir die Senken mitzählen:

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} = O(n^2)$$

Wir werden später noch Funktionen kennenlernen, deren OBDDs unter Umständen exponentielle Größe haben. Bei symmetrischen Funktionen kann dies nicht passieren. Es gibt auch symmetrische Funktionen, die eine lineare Größe aufweisen, nachdem das OBDD reduziert

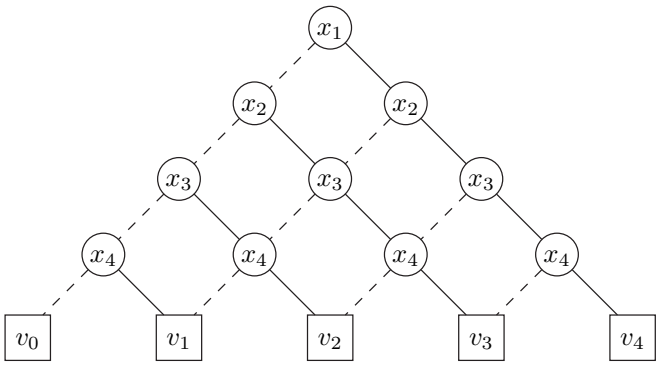


Abbildung 18: OBDD für eine symmetrische Funktion mit dem Wertevektor $(v_0, v_1, v_2, v_3, v_4)$

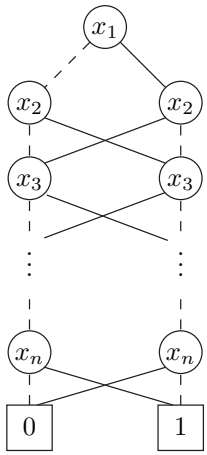


Abbildung 19: OBDD für die Parity-Funktion

wurde. Dazu zählt z.B. die Paritätsfunktion, welche angibt, ob die Anzahl der Einsen in der Eingabe gerade oder ungerade ist. Die Parity-Funktion $f = x_1 \oplus \cdots \oplus x_n$ wird 1, wenn die Anzahl der Einsen in der Eingabe ungerade ist und ansonsten 0. Sie hat also einen Wertevektor, der wie folgt aussieht: $(0, 1, 0, 1, \ldots, 0)$ (wenn n gerade ist) bzw. $(0, 1, 0, 1, \ldots, 1)$ (n gerade). Das reduzierte OBDD für die Parity-Funktion ist in Abbildung 19 skizziert.

Frage 4.20: Welche Funktion kennst du, deren Größe je nach Variablenordnung linear oder exponentiell werden kann?

Die Funktion DQF_n kann je nach Variablenordnung entweder eine lineare oder eine exponentielle Größe aufweisen. Sie ist wie folgt definiert:

$$DQF_n(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = x_1x_2 \vee x_3x_4 \vee \cdots \vee x_{2n-1}x_{2n}$$

Die Funktion setzt sich also aus n Paaren x_ix_{i+1} (wobei i immer ungerade ist) zusammen, die mit einer Disjunktion verknüpft werden. Die Funktion nimmt somit genau dann den Wert 1 an, wenn für mindestens eines dieser Paare x_ix_{i+} gilt, dass $x_i = x_{i+1} = 1$ ist.

Wir betrachten zuerst die gutartige Variablenordnung $\pi = \mathrm{id} = (x_1, x_2, \ldots, x_{2n-1}, x_{2n})$. Diese wird zu einem OBDD linearer Größe führen. Um eine Idee dafür zu bekommen, wie das OBDD aussieht, überlegen wir uns für die Quelle v, welche Subfunktionen an ihrem 0-

Nachfolger v_0 bzw. 1-Nachfolger v_1 dargestellt werden. Die Subfunktion f^0 sei die Subfunktion des 0-Nachfolgers von v und f^1 die des 1-Nachfolgers. Dann gilt:

$$\begin{aligned} f^0 &= f_{|x_1=0} = x_3x_4 \vee \cdots \vee x_{2n-1}x_{2n} \\ f^1 &= f_{|x_1=1} = x_2 \vee x_3x_4 \vee \cdots \vee x_{2n-1}x_{2n} \end{aligned}$$

Die Subfunktion f^0, die an v_0 berechnet wird, hängt also nicht mehr von x_2 ab und trägt daher entsprechend der Variablenordnung die Markierung x_3. Wir betrachten die Nachfolger von v_1 ebenfalls. Der Knoten v_1 hat die Markierung x_2. An seinen Nachfolgern wird $f^1_{|x_2=0} = x_3x_4 \vee \cdots \vee x_{2n-1}x_{2n}$ und $f^1_{|x_2=1} = 1$ berechnet. Die 1-Kante von v_1 führt somit direkt zur 1-Senke und die 0-Kante zum Knoten v_0. An v_0 wird die Funktion $DQF_{n-1}(x_3, x_4, \ldots, x_{2n-1}, x_{2n})$ berechnet, für die wir analog argumentieren können. Für die Größe $S(n)$ des OBDDs für DQF_n gilt also $S(n) = S(n-1) + 2$. Es gilt $S(1) = 4$ und somit insgesamt $S(n) = 2n + 2$.

Die ungünstige Variablenordnung betrachtet zuerst alle Variablen mit ungeraden Indizes. Es gilt also $\pi = (x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})$. Dadurch können wir für eine ungerade Variable x_i, die auf 1 gesetzt wird, erst nach mindestens n weiteren gelesenen Variablen endgültig entscheiden, ob der Ausdruck x_ix_{i+1} den Wert 0 oder 1 annimmt. Wir wollen nun nicht die genaue Anzahl der Knoten abschätzen, sondern nur zeigen, dass die Anzahl der Knoten mit der Markierung x_2 exponentiell in n ist. Wieviele Knoten mit der Markierung x_2 gibt es? Dazu müssen wir die Menge der Subfunktionen betrachten, die dadurch entstehen, dass wir die Variablen $x_1, x_3, \ldots, x_{2n-1}$ auf alle möglichen Arten und Weisen belegen, und die essentiell von x_2 abhängen. Es kommt nun auf die Belegung einer ungeraden Variablen x_i an, ob x_{i+1} in der Subfunktion vorhanden ist oder nicht. Gilt $x_i = 1$, so ist die Variable x_{i+1} in der Subfunktion enthalten, ansonsten nicht. Die entstehenden Subfunktionen haben also die Form:

$$\bigvee_{j \in M \subseteq \{2,4,\ldots,2n\}} x_j$$

Wieviele Teilmengen $M \subseteq \{2, 4, \ldots, 2n\}$ gibt es? Gesucht ist also die Größe der Potenzmenge von $\{2, 4, \ldots, 2n\}$. Da $|\{2, 4, \ldots, 2n\}| = n$ gilt, gibt es also 2^n Teilmengen und somit auch 2^n Subfunktionen, da alle Subfunktionen voneinander verschieden sind. Allerdings sind wir nur an den Subfunktionen interessiert, die essentiell von x_2 abhängen. Wir müssen demnach alle Subfunktionen wegwerfen, die nicht essentiell von x_2 abhängen. Diese sind von der folgenden Form:

$$\bigvee_{j \in M \subseteq \{4,\ldots,2n\}} x_j$$

Es gibt 2^{n-1} solcher Subfunktionen. Also gibt es $2^n - 2^{n-1} = 2 \cdot 2^{n-1} - 2^{n-1} = 2^{n-1}$ verschiedene Subfunktionen, die essentiell von x_2 abhängen. Die genaue Anzahl der Knoten für die ungünstige Variablenordnung kann ebenfalls berechnet werden und beträgt 2^{n+1}. Die Anzahl der x_2-Knoten beträgt also ein Viertel der Gesamtanzahl und demnach ist unsere Abschätzung von mindestens 2^{n-1} Knoten also gar nicht mal so schlecht.

Frage 4.21: Welche Funktionen, die für jede Variablenordnung exponentielle Größe haben, kennst du?

Mit der Funktion DQF_n haben wir eine Funktion kennengelernt, deren OBDD je nach Variablenordnung lineare oder exponentielle Größe haben kann. Es gibt jedoch auch Funktionen, die für jede Variablenordnung eine exponentielle OBDD-Größe aufweisen. Dazu zählt z.B. die

Funktion MUL_n, die das n-te Bit bei der Multiplikation zweier n Bit langer Zahlen berechnet. Da der Beweis dieser Eigenschaft für MUL_n jedoch recht technisch ist, zeigen wir für eine andere Funktion, dass sie bei jeder Variablenordnung exponentielle OBDDs hat. Die Achillesfersenfunktion $ACH_n \in B_{2n+k}$ ist für $n = 2^k$ wie folgt definiert, wobei $x = (x_0, \ldots, x_{n-1})$, $y = (y_0, \ldots, y_{n-1})$ und $s = (s_0, \ldots, s_{k-1})$ gilt:

$$ACH_n(x, y, s) = x_0 y_{|s|} \vee x_1 y_{(|s|+1) \bmod n} \vee \cdots \vee x_{n-1} y_{(|s|+n-1) \bmod n}$$

Wir überlegen uns nun, was diese Funktion genau macht. Im Prinzip ist die Funktion ACH_n eine spezielle Variante der Funktion DQF_n. Im Gegensatz zu DQF_n wird hier jedoch über die Variable s festgelegt, welche Paare der x- und y-Variablen miteinander verknüpft werden. Dazu wird s als Binärzahl interpretiert und die y-Variablen um $|s|$ Stellen nach links rotiert. Dies kann man sich wie folgt vorstellen. Zu Beginn stehen die Strings $x_0, \ldots, x_{n-1}$ und $y_0, \ldots, y_{n-1}$ übereinander. Das s gibt nun an, um wieviele Stellen der y-String nach links rotiert wird. Jedes Element aus y wird also um $|s|$ Stellen nach links verschoben. Wenn dabei ein Element links „herausfällt“, kommt es auf der rechten Seite wieder rein. Die Elemente die nach einer solchen Rotation untereinander stehen, werden miteinander verknüpft.

Unser Ziel ist es nun, zu zeigen, dass ein OBDD für ACH_n für jede Variablenordnung exponentielle Größe haben muss. Da das OBDD auch auf alle Belegungen von s reagieren muss, werden wir nun zeigen, dass es für jede Variablenordnung eine Belegung von s gibt, so dass ein OBDD, welches sich nur um dieses s „kümmert“ schon exponentielle Größe hat. Es sei also eine Variablenordnung π gegeben. In dieser kommen in irgendeiner Reihenfolge die Variablen $x_0, \ldots, x_{n-1}$, $y_0, \ldots, y_{n-1}$ und $s_0, \ldots, s_{k-1}$ vor. Wir filtern aus π nun alle s-Variablen aus und betrachten nur noch die x- und y-Variablen in der Reihenfolge, wie sie in der Variablenordnung auftreten. Es ergibt sich also ein $2n$ langer String aus x- und y-Variablen. Mit L werden wir nun die vorderen n Variablen in diesem String bezeichnen und mit R die hinteren n Variablen. Desweiteren nehmen wir o.B.d.A. an, dass sich mindestens $\frac{n}{2}$ der x-Variablen in L befinden und entsprechend mindestens $\frac{n}{2}$ der y-Variablen in R. Es kann nicht vorkommen, dass in L und R weniger als $\frac{n}{2}$ x-Variablen vorkommen, da ansonsten die Summe der x-Variablen kleiner als n sein müsste und wir somit x-Variablen „vergessen“ hätten. Entsprechend kann für die y-Variablen argumentiert werden. Mit L' wollen wir nun die mindestens $\frac{n}{2}$ x-Variablen in L bezeichnen und mit R' die mindestens $\frac{n}{2}$ y-Variablen in R. Für welchen Wert von $|s|$ werden nun eine x-Variable aus L' und eine y-Variable aus R' miteinander verknüpft? Wenn $|s| = t$ ist, dann muss $x_i \in L'$ und $y_{(i+t) \bmod n} \in R'$ gelten. Mit $J(t)$ bezeichnen wir nun die Menge aller Paare für die dies gilt. Da $t \in \{0, \ldots, n-1\}$ gilt, gibt es also die Mengen $J(0), J(1), \ldots, J(n-1)$. Wir überlegen uns nun zuerst, wieviele solcher Paare es insgesamt geben muss bzw. wie groß die Vereinigung $J(0) \cup J(1) \cup \cdots \cup J(n-1)$ mindestens sein muss. Es gibt mindestens $\frac{n}{2}$ x-Variablen in L', die mit mindestens $\frac{n}{2}$ y-Variablen in R' verknüpft werden können. Somit gibt es mindestens $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$ solcher Paare. Für jedes dieser Paare kann auch ein t gefunden werden, so dass es in die Menge $J(t)$ eingeordnet wird. Im nächsten Schritt überlegen wir uns nun, dass auch mindestens eine der Mengen $J(0), \ldots, J(n-1)$ mindestens $\frac{n}{4}$ solcher Paare enthalten muss. Wären alle Mengen kleiner, so würde die Summe der Mengen nicht $\frac{n^2}{4}$ ergeben, denn es würde gelten $|J(0) \cup \cdots \cup J(n-1)| < n \cdot \frac{n}{4} = \frac{n^2}{4}$. Es gibt also ein t, so dass $|J(t)| \geq \frac{n}{4}$ ist. Wir suchen uns nun genau $\frac{n}{4}$ dieser Paare aus $J(t)$ aus, wodurch sich die folgende Menge ergibt:

$$\left\{ x_{i(1)} y_{j(1)}, \ldots, x_{i\left(\frac{n}{4}\right)} y_{j\left(\frac{n}{4}\right)} \right\} \subseteq J(t)$$

Dabei stehen alle $x_{i(\cdot)}$ vor allen $y_{j(\cdot)}$ in der Variablenordnung. Wir bilden nun die Subfunktion, die sich ergibt, wenn wir s so setzen, dass $|s| = t$ gilt, und alle x- und y-Variablen, die nicht in unserer Auswahl vorkommen, auf 0. Es ergibt sich dann die Subfunktion:

$$x_{i(1)}y_{j(1)} \vee x_{i(2)}y_{j(2)} \vee \cdots \vee x_{i(\frac{n}{4})}y_{j(\frac{n}{4})}$$

Wir haben schon gesehen, dass alle x-Variablen dieser Subfunktion vor allen y-Variablen in der Variablenordnung stehen. Aus der Analyse von DQF_n wissen wir auch schon, dass das OBDD für eine solche Variablenordnung die Größe $2^{\frac{n}{4}+1}$ hat. Wieso hat dann nun auch das OBDD für ACH_n exponentielle Größe? Denn wir könnten ja nun argumentieren, dass wir die Variablen in keiner systematischen Reihenfolge konstant gesetzt haben und deshalb nichts über eine Subfunktion aussagen können. Wenn wir z.B. nur die ersten k Variablen in der Variablenordnung konstant setzen würden, bekämen wir nach dem Struktursatz eine Subfunktion, die ganz sicher an einem der Knoten berechnet werden würde. Wir haben jedoch irgendwelche Variablen in der Variablenordnung konstant gesetzt. Wir müssen also ein wenig anders argumentieren. Eine der in Kapitel 4.1 gewünschten Operationen war die Operation „Ersetzung durch Konstanten“. In Kapitel 4.5 werden wir sehen, wie diese Operation für OBDDs realisiert wird. Es wird sich dabei zeigen, dass ein OBDD bei dieser Operation nie größer sondern eher kleiner wird. Wenn wir also das OBDD für die ganze Funktion ACH_n nach dem oben beschriebenen System nach und nach konstant setzen, wird sich das OBDD also höchstens verkleinern. Sind alle Variablen entsprechend konstant gesetzt, bekommen wir ein OBDD für eine Funktion, die exponentiell große OBDDs benötigt. Also muss auch schon das ursprüngliche OBDD für ACH_n exponentielle Größe haben, da es sich ja höchstens verkleinern kann.

4.4 Das Variablenordnungsproblem

4.5 Algorithmen auf OBDDs

Frage 4.22: Wie wird die Operation „Auswertung“ auf OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Gegeben sei ein OBDD G für eine Funktion $f \in B_n$. Die Operation „Auswertung“ funktioniert bei einem OBDD genau so wie die in Kapitel 4.2 beschriebene Auswertung von BDDs. Wir beginnen bei der Quelle und folgen den Kanten entsprechend der Belegung von $a \in \{0,1\}^n$ bis wir eine Senke erreichen. Da wir höchstens n Variablen betrachten müssen, beträgt die Laufzeit also $O(n)$. Es wird nur der Platz zum Speichern des OBDDs selbst benötigt. Dieser beträgt $O(|G|)$.

Frage 4.23: Wie wird die Operation „Erfüllbarkeit“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Die Operation „Erfüllbarkeit“ können wir auch als die Frage ansehen, ob es sich um die Nullfunktion handelt oder nicht. Denn nur die Nullfunktion nimmt für alle Eingaben den Wert 0 an. Wenn das OBDD also reduziert ist, reicht es, zu überprüfen, ob die Quelle direkt die Nullsenke ist. Ansonsten müssen wir einen DFS-Durchlauf durch das OBDD machen und schauen, ob wir währenddessen eine 1-Senke erreichen. Der Zeit und Platzbedarf ist also $O(|G|)$.

Frage 4.24: Wie wird die Operation „Erfüllbarkeit-Anzahl“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Gegeben sei ein OBDD G für eine Funktion $f \in B_n$. Wir sind also an $|f^{-1}(1)|$ interessiert, d.h. wir wollen wissen, für wieviele Eingaben wir eine 1-Senke in G erreichen. Dazu wollen wir nun für jeden Knoten bzw. für jede Kante berechnen, für wieviele Eingaben der Knoten bzw. die Kante durchlaufen wird. Die Quelle wird offensichtlich für 2^n Eingaben durchlaufen. Nun wollen wir uns überlegen, für wieviele Eingaben ein Knoten in Abhängigkeit von seinen Vorgängerknoten durchlaufen wird. Wir betrachten eine Knoten v mit der Markierung x_i. O.B.d.A. sei Variablenordnung $\pi = \mathrm{id}$. Wenn der Knoten v für eine Belegung $a = (a_1, \ldots, a_n)$ erreicht wird, dann wird er auch für die Belegung $a' = (a_1, \ldots, a_{i-1}, \overline{a}_i, a_{i+1}, \ldots, a_n)$ erreicht. In einem Fall folgen wir also der 0-Kante und im anderen Fall der 1-Kante, d.h. die Eingaben, für die v erreicht wird, werden gleichmäßig auf seine beiden Nachfolger verteilt. Um „Erfüllbarkeit-Anzahl“ zu berechnen, markieren wir die Quelle also mit 2^n und alle übrigen Knoten mit 0. Im Anschluss gehen wir schichtenweise durch das OBDD und betrachten jeden Knoten. Wenn wir einen Knoten v mit der Markierung c erreichen, addieren wir zu seinen beiden Nachfolgern jeweils den Wert $\frac{c}{2}$. Das Ergebnis ist die Summe der Markierungen der 1-Senken.

Frage 4.25: Wie wird die Operation „Erfüllbarkeit-Alle“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Wir erhalten die Menge Erfüllbarkeit-Alle, indem wir alle möglichen Pfade, die von der Quelle zu einer 1-Senke laufen, betrachten. Für einen Pfad erhalten wir die Belegungen, die diesen Pfad durchlaufen, indem wir die auf dem Pfad getesteten Variablen entsprechend setzen und die Variablen, die nicht auf dem Pfad getestet werden, auf alle möglichen Weisen belegen. Dazu können wir einen rekursiven Algorithmus benutzen, der von der Quelle startend aufgerufen wird. Jeder Knoten, für den der Algorithmus aufgerufen wird, ruft den Algorithmus zuerst rekursiv für seinen 0-Nachfolger auf und anschließend rekursiv für seinen 1-Nachfolger. Wir dabei eine 1-Senke erreicht, wird der entsprechende Pfad rekonstruiert und alle Belegungen, die diesen Pfad durchlaufen, ausgegeben. Der Platzbedarf beträgt $O(|G|)$ und der Speicherbedarf $O(|G| + n|f^{-1}(1)|)$.

Frage 4.26: Wie wird die Operation „Äquivalenztest“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Gegeben seien zwei OBDDs G_f bzw. G_g für die Funktionen $f, g \in B_n$, die jeweils dieselbe Variablenordnung benutzen. Da das minimierte OBDD für eine Funktion eindeutig ist, reicht es aus, G_f und G_g zu reduzieren, wobei die OBDDs G_f^* und G_g^* entstehen. Wenn $f \equiv g$ ist, müssen die OBDDs G_f^* und G_g^* also isomorph zueinander sein. Wir könnten nun erst auf die Idee kommen, dass dies nicht effizient möglich ist, da es sich ja um das NP-vollständige Problem handelt, zu entscheiden, ob zwei Graphen isomorph zueinander sind. Unser Problem ist jedoch etwas spezieller. Zum einen gibt es für beide OBDDs eine klar ausgezeichnete Quelle und zum anderen hat jeder Knoten genau zwei Kanten, die ebenfalls eindeutig ausgezeichnet sind. Der Test ob zwei OBDDs dieselbe Funktion darstellen, besteht also darin, auf den beiden reduzierten OBDDs G_f und G_g einen parallelen DFS-Durchlauf auszuführen und in jedem Schritt zu schauen, ob die erreichten Knoten dieselbe Knotenmarkierung haben.

Frage 4.27: Wie wird die Operation „Synthese“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

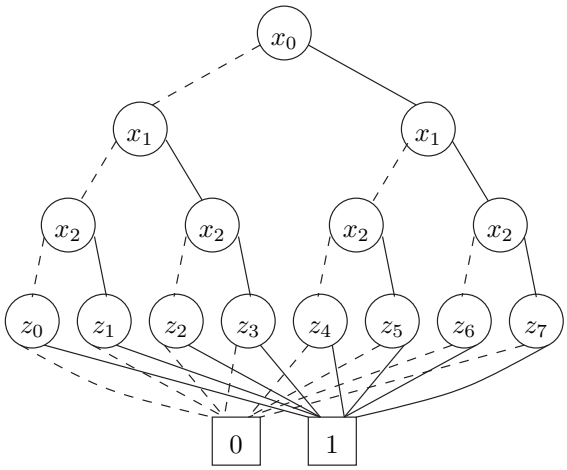


Abbildung 20: Beispiel-OBDD für MUX mit $n = 2^3 = 8$

Frage 4.28: Welche Funktion kann benutzt werden, um zu zeigen, dass es bei der Synthese tatsächlich zu quadratischem Blowup kommen kann?

Wir wollen nun zeigen, dass es bei der Synthese tatsächlich zu einem quadratischen Blowup kommen kann, d.h. dass die Schranke von $O(|G_f| \cdot |G_g|)$ auch tatsächlich erreicht werden kann. Dazu benutzen wir die Funktion MUX, die für $n = 2^k$ wie folgt definiert ist:

$$MUX(x_0, \ldots, x_{k-1}, z_0, \ldots, z_{n-1}) = z_{|x|}$$

Die Funktion MUX bekommt als Eingabe also eine Adresse $x = (x_0, \ldots, x_{k-1})$ sowie die Belegung von n Speicherzellen $z_0, \ldots, z_{n-1}$ gegeben und gibt den Wert der durch x adressierten Speicherzelle aus. Wir können uns schnell klarmachen, dass wir mit den x-Variablen, von denen es $k = \log n$ gibt, alle $2^k = n$ Speicherzellen adressieren können.

Als nächstes überlegen wir uns nun, wie wir MUX mit einem OBDD darstellen können. Als Variablenordnung wählen wir $\pi = (x_0, \ldots, x_{k-1}, z_0, \ldots, z_{n-1})$. Im OBDD bilden die x-Variablen einen vollständigen Entscheidungsbaum mit $n = 2^k$ Blättern. An jedem dieser Blätter hängt eine Speichervariable z_i, wobei diese von „links nach rechts“ in der Reihenfolge $z_0, z_1, \ldots, z_{n-1}$ angeordnet sind. Welche Größe hat ein solches OBDD? Für die x-Variablen benötigen wir $2^k - 1 = n - 1$ Knoten und für die z-Variablen n Knoten. Zuzüglich der beiden Senken ergibt dies insgesamt $2n+1 = O(n)$ Knoten. Die Funktion MUX, welche über $n+\log n$ Variablen definiert ist, hat also eine lineare Größe in der OBDD-Darstellung. Abbildung 20 zeigt ein Beispiel für $n = 2^3$.

Kommen wir nun zu dem, was wir eigentlich zeigen wollen. Wir definieren uns zwei MUX-Funktionen:

$$\begin{aligned} f &= MUX(x_0, \ldots, x_{k-1}, z_0, \ldots, z_{n-1}) = z_{|x|} \\ g &= MUX(y_0, \ldots, y_{k-1}, z_0, \ldots, z_{n-1}) = z_{|y|} \end{aligned}$$

Wir wollen nun die Funktion $f \vee g$ synthetisieren, wobei wir die Variablenordnung $\pi = (x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}, z_0, \ldots, z_{n-1})$ wählen. Um zu zeigen, dass der Blowup tatsächlich quadratisch wird, zeigen wir, dass die Anzahl der z-Knoten schon mindestens quadratisch ist.

Dazu überlegen wir uns zuerst, wieviele z_i-Knoten es gibt. Nach dem Struktursatz müssen wir uns dazu überlegen, wieviele essentiell von z_i abhängige Subfunktionen entstehen, wenn wir $(x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}, z_0, \ldots, z_{i-1})$ auf alle möglichen Arten konstant setzen. Da wir eine untere Schranke zeigen wollen, beschränken wir uns auf bestimmte Konstantsetzungen. Wir setzen die x-Variablen so, dass wir z_i auswählen, d.h. $(x_0, \ldots, x_{k-1})$ ist die Binärdarstellung von i. Die y-Variablen setzen wir nun auf alle möglichen Arten, so dass ein z_j mit $j \geq i$ ausgewählt wird. Die z-Variablen werden alle auf 0 gesetzt. Bei diesen Konstantsetzungen entstehen also Funktionen $z_i \vee z_j$ mit $j \geq i$. Diese Funktionen sind von z_i essentiell abhängig und es gibt $n - i$ verschiedene solcher Funktionen für ein z_i. Damit beträgt die Anzahl aller z-Knoten also mindestens:

$$\sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Omega(n^2)$$

Frage 4.29: Wie wird die Operation „m-äre Synthese“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Frage 4.30: Wie wird die Operation „Ersetzung durch Konstanten“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Bei dieser Operation ist ein OBDD G für eine Funktion $f \in B_n$ sowie eine Variable x_i und eine Konstante c gegeben. Wir möchten nun das OBDD für die Funktion $f_{|x_i=c}$ berechnen. Dazu müssen wir unterscheiden, ob die Quelle oder ein innerer Knoten die Markierung x_i trägt. Wenn die Quelle die Markierung x_i trägt, müssen wir einfach der Kante mit der Markierung c folgen. Der erreichte Knoten ist die Quelle des Ergebnis-OBDDs. Ansonsten durchlaufen wir den Graph mit einer DFS-Suche. Für jeden Knoten w, den wir dabei erreichen, schauen wir nun, ob eine seiner Kanten auf einen Knoten mit der Markierung x_i zeigt. Ist dies für eine Kante der Fall, ist der neue Knoten, auf den diese Kante zeigt, der c-Nachfolger des Knotens mit der Markierung x_i. Durch dieses Vorgehen überspringen wir im Prinzip alle Knoten mit der Markierung x_i und gehen vom Vorgängerknoten sofort zum entsprechenden c-Nachfolger. Wir löschen alle x_i-Knoten, wobei es nun vorkommen kann, dass nicht mehr alle Knoten erreichbar sind. Knoten, die nicht mehr erreichbar sind, werden ebenfalls gelöscht.

Das auf diese Art und Weise entstehende OBDD muss nicht minimal sein, selbst wenn das ursprüngliche OBDD minimal war. Wir müssen den Reduktionsalgorithmus jedoch auch nicht mehr auf alle Ebenen anwenden. Da der Reduktionsalgorithmus bottom-up vorgeht und das OBDD somit ebenenweise minimiert, müssen wir nur die Ebenen erneut minimieren, in denen sich auch was geändert hat. In den Ebenen unterhalb von i wurden nur Knoten gelöscht, aber keine Zeiger umgesetzt. Dies kann höchstens in den Ebenen oberhalb von i passiert sein, so dass wir erst ab Ebene i beginnen müssen.

Frage 4.31: Wie wird die Operation „Ersetzung durch Funktionen“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Gegeben sind zwei OBDDs G_f und G_g für die Funktionen $f, g \in B_n$. Desweiteren ist eine Variable x_i gegeben. Wir wollen nun das OBDD der Funktion $f_{|x_i=g}$ berechnen, die wie folgt definiert ist:

$$f_{|x_i=g} = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$$

Das i-te Eingabebit der Funktion f wird also für jede Eingabe $a \in \{0,1\}^n$ stets durch den Wert ersetzt, den g auf der Eingabe a berechnet. Liefert g den Wert 0, so wollen wir also die Funktion $f_{|x_i=0}$ berechnen und ansonsten $f_{|x_i=1}$. Damit kann die Funktion $f_{|x_i=g}$ also entsprechend der Shannon-Zerlegung wie folgt geschrieben werden:

$$f_{|x_i=g} = \overline{g} f_{|x_i=0} \vee g f_{|x_i=1}$$

Frage 4.32: Wie wird die Operation „Quantifizierung“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Frage 4.33: Wie wird die Operation „Redundanztest“ bei OBDDs realisiert? Welchen Zeit- und Platzbedarf hat sie?

Gegeben ist ein OBDD G für eine Funktion f, sowie eine Variable x_i. Wir wollen nun wissen, ob die Funktion essentiell von x_i abhängt. Wenn f nicht essentiell von x_i abhängt, gilt $f_{|x_i=0} \equiv f_{|x_i=1}$.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Literatur

[Molitor und Scholl 1999] Molitor, Paul und C. Scholl (1999). *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. B.G. Teubner Stuttgart, Erste Aufl.

[Wegener 1989] Wegener, Ingo (1989). *Effiziente Algorithmen für grundlegende Funktionen*. B.G. Teubner Stuttgart, Erste Aufl.

Index