Seminar

Perlen der Theoretischen Informatik

Branching-Programme beschränkter Breite

Ralf Berger

16.01.2002

1. Einleitung

Innerhalb der theoretischen Informatik existieren eine Reihe von verschiedenen Berechnungsmodellen. Bei allen Gemeinsamkeiten gibt es natürlich Unterschiede, die gerade die Vielzahl von Berechnungsmodellen rechtfertigen.
Beispielsweise sind Registerautomaten oder Schaltkreise für technische Anwendungen deutlich anschaulicher als etwa Turingmaschinen.
Die wesentlichsten Unterschiede zwischen den verschiedenen Berechnungsmodellen ergeben sich aber in den aus ihnen resultierenden Komplexitätsmaßen. Das heißt, dass eine Sprache in verschiedenen Berechnungsmodellen durchaus verschieden schwer sein kann, insbesondere dann, wenn man für die Berechnungsmodelle zusätzliche Constraints einführt (beschränkte Größe, Alphabet, Determinismus, ...). Zum Beispiel können Schaltkreise, im Gegensatz zu allen Maschinenmodellen, bestimmte nicht-rekursive Funktionen berechnen. Des weiteren hat die parity-Funktion in Schaltkreisen konstanter Tiefe exponentielle Komplexität, in konstant breiten Branching-Programmen jedoch nur lineare.

Im folgenden soll es um einen Beweis aus dem Jahr 1986 gehen, der für ein Berechnungsmodell (Branching-Programme konstanter Breite) eine unerwartete Mächtigkeit nachwies, was wiederum Auswirkungen auf die Komplexitätsmaße anderer Modelle hat.

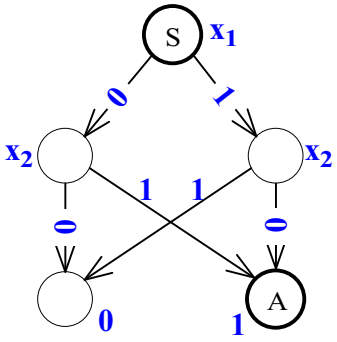
2. Branching-Programme

Wie bereits vorgestellt, handelt es sich bei Branching-Programmen um ein weiteres nicht-uniformes Berechnungsmodell. Sie werden auch als OBDD (ordered binary decision diagram) oder verallgemeinerte Entscheidungsgraphen bezeichnet und sind bis heute bezüglich ihrer Mächtigkeit und Komplexität nicht vollständig charakterisiert.
Als *nicht-uniform* bezeichnet man Berechnungsmodelle G, die nur Funktionen f_n: $B^n \rightarrow B$ fester Inputlänge n berechen können. Um die entsprechenden Komplexitätsmaße mit denen von uniformen Modellen (z.B. Turingmaschine) vergleichen zu können, betrachtet man für eine Funktion f immer ganze Familien $(G_1, ..., G_n)$, wobei G_i die Funktion f mit auf B^i eingeschränktem Definitionsbereich berechnet.

Def. 1
Ein **Branching-Programm** ist ein gerichteter azyklischer Graph. Die Knotenmenge besteht aus (inneren) Knoten mit fanout 2, die mit Variablen x_i beschriftet sind, und Knoten (Blättern) mit fanout 0, beschriftet mit booleschen Konstanten. Die beiden ausgehenden Kanten der inneren Knoten sind jeweils mit 0 und 1 beschriftet. Innerhalb der inneren Knoten wird einer als Startknoten S ausgezeichnet, ein Knoten aus der Menge der Blätter ist der akzeptierende Endknoten A.

Der Graph aus Beispiel 1 ist laut dieser Definition ein korrektes Branching-Programm.

Beispiel 1:

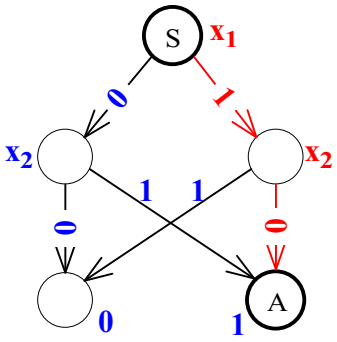


Wie und was berechnet nun aber ein solcher Graph?
Ein Branching-Programm P berechnet eine boolesche Funktion vom Typ $f \in B_n = B_{n,1} : \{0,1\}^n \to \{0,1\}$ nach folgendem Algorithmus:

- Eingabe $(a_1, ..., a_n)$
- Beginne beim Startknoten
  - Zu jedem Knoten u mit der Beschriftung x_i folge der Kante (u,v), die mit a_i beschriftet ist, d.h der neue Knoten ist v
- $f(a)$ ist dann die Beschriftung des so erreichten Blattes.

Das heißt, P berechnet f, wenn für alle $a \in \{0,1\}^n$: $f(a) = 1$ gdw. der Pfad durch P mittels a endet im akzeptierenden Endknoten.
Auf unser Beispiel 1 angewandt bedeutet das:

Eingabe a = (1, 0), d.h. $x_1 = 1$, $x_2 = 0$



$\Rightarrow f(1,0) = 1$
Wie man sofort sieht, ist $f(0,0) = 0$, $f(0,1) = 1$ und $f(1,1) = 0$.
Das Beispiel berechnet also die Funktion $f \in \{0,1\}^2 \to \{0,1\} =$ **x_1 XOR x_2** .

Betrachten wir nun Branching-Programme besonderer Form:

Def. 2
Ein Branching-Programm P = (V,E) liegt in **Schichtenform** vor, wenn sich die Knotenmenge V vollständig in disjunkte Mengen $V_0, ..., V_k$ (Schichten) separieren läßt, so dass gilt:

- $S \in V_0$
- $A \in V_k$
- $E \subseteq \{ (u,v) \mid u \in V_i,\ v \in V_{i+1},\ 0 \le i < k \}$
- (alle Knoten aus einer Schicht V_i testen die selbe Variable x_i)

Wie man sich leicht klarmacht, ist die Schichtenform keine Einschränkung. Jedes Branching-Programm kann durch Hinzufügen von „Dummy-Knoten“ in Schichtenform gebracht werden. Das Beispiel 1 liegt bereits in Schichtenform vor.

Eigenschaften von Branching-Programmen

Man bezeichnet mit der

- *Größe* von Branching-Programmen (*BP(f)*) die Anzahl seiner inneren Knoten (entspricht bei Maschinenmodellen dem Platzverbrauch),
- *Tiefe* von Branching-Programmen (*BPD(f)*) die Anzahl der Knoten im längsten Pfad (entspricht bei Maschinenmodellen dem Zeitverbrauch),
- *Breite* von (geschichteten) Branching-Programmen (*BPW(f)*) die maximale Anzahl von Knoten in einer Schicht.

BF(*f*) ist eines der wichtigsten Komplexitätsmaße für boolesche Funktionen.

Mächtigkeit von Branching-Programmen

Wie schon oben erwähnt, ist die Mächtigkeit von Branching-Programmen in allen Variationen noch nicht vollständig geklärt. Was man sicher weiß, ist zum Beispiel, dass die Klasse der durch polynomial große Branching-Programme berechenbaren Funktionen gerade (nicht-uniform) LOGSPACE ist.
Bis heute kennt man aber kein effizientes Verfahren, um zu einer gegebenen booleschen Funktion ein entsprechendes Branching-Programm zu generieren. Das zur Zeit beste Verfahren aus dem Jahr 1999 (Sauerhoff, Wegener, Werchner) erreicht für beliebige boolesche Funktionen: $BP(f) < 1{,}360 \cdot L(f)^{1{,}195}$, wobei die Anzahl der Literale L in einer beliebigen Normalform für f i.a. nichtpolynomiell in der Anzahl der Variablen ist.
Ein interessanter Satz führt uns auf eingeschränkte Branching-Programme.

Satz 1. (ohne Beweis)
Schaltkreise konstanter Tiefe lassen sich effizient durch Branching-Programme konstanter Breite simulieren. (Die Umkehrung gilt nicht.)

Schaltkreise konstanter Tiefe sind für technische Anwendungen gerade die relevanten, da sie konstante Signallaufzeiten sichern. Daher ist die Frage nach der Mächtigkeit breitenbeschränkter Branching-Programme besonders interessant. Man weiss, dass sich alle booleschen Funktionen durch Branching-Programme der Breite 2 berechnen lassen, leider sind diese dann i.a. exponentiell groß. Betrachten wir also nur polynomial große Branching-Programme konstanter Breite k. (w-k-BP).
Bis 1986 galt die Annahme, dass z.B. die Majoritätsfunktion durch solche Branching-Programme nicht berechnet werden könne. Aufgrund dieser Annahme wurden zahlreiche „Beweise“ über untere Schranken für die Majoritätsfunktion und andere Probleme geführt.

Def. 3

$$majority(\vec{x}) = \begin{cases} 1 & \sum_{i=1}^{n} x_i \geq n/2 \\ 0 & sonst \end{cases}$$

1986 konnte M. Barrington schließlich zeigen, dass die Majoritätsfunktion eben doch durch polynomial große Branching-Programme konstanter Breite berechnet werden kann.

3. Der Barrington-Beweis

Barrington zeigte dazu mittels eines allgemeinen Konstruktionsverfahrens für die entsprechenden Branching-Programme folgende Äquivalenz:

Satz. 2

$$\textbf{w-5-BP} = \textbf{NC}^{\textbf{1}}.$$

Aufgrund dieses Satzes sind wiederum viele o.g. „Beweise“ hinfällig geworden.

Def. 4
Als $\mathbf{NC^k}$ bezeichnet man die Sprachen CirDepthSize(LOG^k, POLY).
$\Rightarrow$ $\mathbf{NC^1}$ = CirDepthSize(LOG, POLY)
NC^1 ist also die Klasse von Sprachen, die durch Schaltkreise polynomialer Größe und logarithmischer Tiefe berechnet werden können. Weiter verlangt CirDepthSize(D,S) die Konstruierbarkeit eines entsprechenden Schaltkreises in O(S·logS) Platz und O(log n) Zeit.
NC ($\bigcup_{k\in N^+} NC^k$) steht für Nick’s Class und soll an Nicholas Pippenger erinnern, der als erster diese Klasse untersuchte. Es ist die Klasse von Problemen, die schon sequentiell effizient lösbar und zusätzlich hochgradig parallelisierbar sind, d.h. bei polynomiellen Schaltkreisgrößen sublineare Zeitschranken haben. In NC^1 beispielsweise liegen Addition, Multiplikation und Maximumbildung von Binärzahlen, Merging sowie Matrixmultiplikation und die erwähnte Majoritätsfunktion.

Die Idee des Beweises von Satz 2 ist nun, für beliebige polynomial große, logarithmisch tiefe Schaltkreise die Konstruktion eines Branching-Programms konstanter Breite anzugeben, welches die selbe Funktion berechnet (und umgekehrt).

Bevor wir zum Kern des Beweises kommen, führen wir für Permutationen eine günstige Notation ein.

Def. 5
Eine **Permutation** ist eine injektive und surjektive Abbildung π: N$\rightarrow$N mit N $\subseteq \mathbb{N}$.

klassische Darstellung: $\pi = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ \pi(a_1) & \pi(a_2) & \dots & \pi(a_n) \end{pmatrix}$

Zyklenschreibweise: $(a_1, \pi(a_1), \pi(\pi(a_1)), \dots)$

Eine Permutation $(i_1, i_2, ..., i_n)$ heißt zyklisch, wenn $\pi(i_j) = \begin{cases} i_1 & j = n \\ i_{j+1} & sonst \end{cases}$

oder anschaulich, wenn sie nur einen Zyklus besitzt.
Beispiel:

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = (1,2,3) \qquad \pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 2 & 5 & 4 \end{pmatrix} = (1,3,2)(4,5)$$

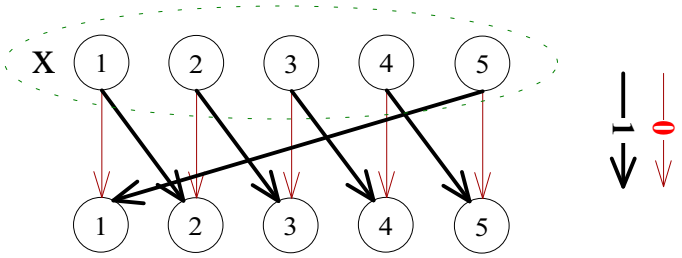
π_1 ist zyklisch, π_2 nicht.

Für den Beweis von Satz 2 führte Barrington eine neue Form von Branching-Programmen ein, die er Permutations-Branching-Programm nannte.

Def. 6
Ein **Permutations-Branching-Programm (PBP)** ist ein Branching-Programm mit folgenden Einschränkungen:

- ein PBP liegt in Schichtenform vor
- jede Schicht hat genau 5 Knoten, welche jeweils von 1-5 durchnummeriert sind und alle die selbe Variable testen
- die 0- und 1-Kantenverbindungen zwischen 2 Schichten realisieren jeweils Permutationen auf der Menge {1,2,3,4,5}
- auf einen einzelnen expliziten Start- und Endknoten wird i.a. verzichtet
- die Beschriftung der Blätter entfällt

Beispiel 2:



Dieses Beispiel besteht aus genau 2 Schichten, d.h nur die erste Schicht testet eine Variable x. Die 0-Kanten zwischen der ersten und der zweiten Schicht realisieren die Permutation $\begin{pmatrix}1&2&3&4&5\\1&2&3&4&5\end{pmatrix}$ = id. Die 1-Kanten entsprechen der Permutation $\begin{pmatrix}1&2&3&4&5\\2&3&4&5&1\end{pmatrix}$ = (1,2,3,4,5).

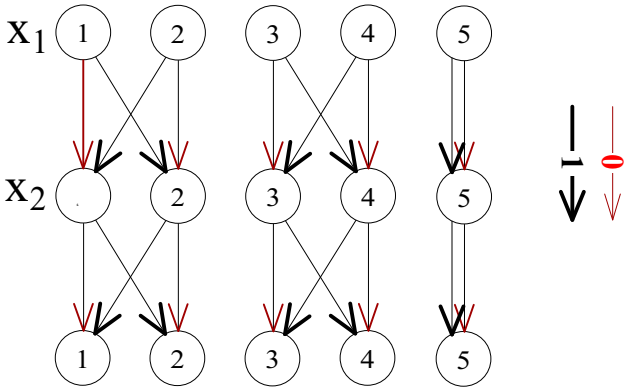
Auch die Berechnungsvorschrift ist nun natürlich eine andere:
Ein PBP **π-berechnet** eine boolesche Funktion f (computes via π) wenn:

- π ist eine Permutation, $\pi \neq id$
- $\forall\, a=(a_1,\ldots,a_n) \in \{0,1\}^n$: mit $f(a)=1$ gilt:
  die Verbindungen zwischen den 5 Knoten der ersten und der letzten Schicht bei Eingabe $a=(a_1,\ldots,a_n)$ realisieren gerade die Permutation π.
- $\forall\, a=(a_1,\ldots,a_n) \in \{0,1\}^n$: mit $f(a)=0$ gilt:
  die Verbindungen zwischen den 5 Knoten der ersten und der letzten Schicht bei Eingabe $a=(a_1,\ldots,a_n)$ realisieren gerade die Identität id.

Auf das Beispiel 2 angewendet, bedeutet das: Bei Eingabe a = 0 ergibt sich für die Verbindungen zwischen der ersten und der letzten Schicht gerade die Identität. Bei Eingabe a = 1 permutieren die Knoten mit (1,2,3,4,5).

Das bedeutet, wir haben mit Bsp.2 bereits ein PBP gefunden, welches die Funktion $f(x) = x$ π-berechnet mit π = (1,2,3,4,5).

Betrachten wir ein weiteres (3-schichtiges) Beispiel
Beispiel 3:



Die 0-Kanten zwischen der ersten und der zweiten Schicht realisieren die Permutation $\begin{pmatrix}1&2&3&4&5\\1&2&3&4&5\end{pmatrix}$ = id. Die 1-Kanten entsprechen der Permutation $\begin{pmatrix}1&2&3&4&5\\2&1&4&3&5\end{pmatrix}$ = (1,2)(3,4)(5). Zwischen der zweiten und dritten Schicht ergeben sich hier nocheinmal die selben Permutationen.

Verfolgen wir nun die Pfade für die verschiedenen Eingaben a:
Sei a = (1,0), d.h. $x_1 = 1$ und, $x_2 = 0$. Vom Knoten 1 in der ersten Schicht kommt man so über Knoten 2 in der zweiten zu Knoten 2 in der letzten. Von Knoten 2 gelangt man über Knoten 1 zu Knoten 1 usw. .
Als Permutation zwischen der ersten und der letzten Schicht bei a = (1,0) erhält man also $\begin{pmatrix}1&2&3&4&5\\2&1&4&3&5\end{pmatrix}\bullet\begin{pmatrix}1&2&3&4&5\\1&2&3&4&5\end{pmatrix}$ = (1,2)(3,4)(5) · id = (1,2)(3,4)(5).
Analog ergibt sich für a = (0,0) und a = (1,1) die Identität id und für a = (0,1) wieder die Permutation (1,2)(3,4)(5).

Wir haben also mit Beispiel 3 ein PBP, welches die Funktion x_1 XOR x_2 π-berechnet, mit π = (1,2)(3,4)(5).

Lemma 1
Jedes Permutations-Branching-Programm, welches eine Funktion *f* π-berechnet, kann durch ein normales Branching-Programm derselben Größe und Tiefe simuliert werden.
Beweis:

Wegen π ≠ id existiert ein Knoten i in der ersten Schicht mit π(i) ≠ i. Sei nun i in der ersten Schicht der Startknoten und der Knoten π(i) in der letzten Schicht der akzeptierende Endknoten. Dieses Branching-Programm berechnet *f* im üblichen Sinne.

Es reicht also, die Äquivalenz zu NC^1 auf Basis von Permutations-Branching-Programmen zu zeigen.

Lemma 2

Sei P ein PBP, das f π-berechnet, wobei π eine zyklische Permutation ist. Sei weiter auch σ eine beliebige zyklische Permutation. Dann gibt es ein PBP P' der selben Größe wie P, welches f σ-berechnet.

Beweis:

Seien: $\pi = (i_1, i_2, i_3, i_4, i_5)$ und $\sigma = (j_1, j_2, j_3, j_4, j_5)$.

Weiter sei τ eine neue Permutation $\tau = \begin{pmatrix} i_1 & i_2 & i_3 & i_4 & i_5 \\ j_1 & j_2 & j_3 & j_4 & j_5 \end{pmatrix}$

und dementsprechend $\tau^{-1} = \begin{pmatrix} j_1 & j_2 & j_3 & j_4 & j_5 \\ i_1 & i_2 & i_3 & i_4 & i_5 \end{pmatrix}$.

Dann gilt: $\sigma = \tau^{-1} \circ \pi \circ \tau$, da

$$\begin{aligned} \tau^{-1} \circ \pi \circ \tau &= \tau\left(\pi\left(\tau^{-1}(j_k)\right)\right) \\ &= \tau\left(\pi(i_k)\right) \\ &= \tau\begin{pmatrix} i_1 & k=5 \\ i_{k+1} & sonst \end{pmatrix} \\ &= \begin{cases} j_1 & k=n \\ j_{k+1} & sonst \end{cases} \\ &= \sigma(j_k) \end{aligned}$$

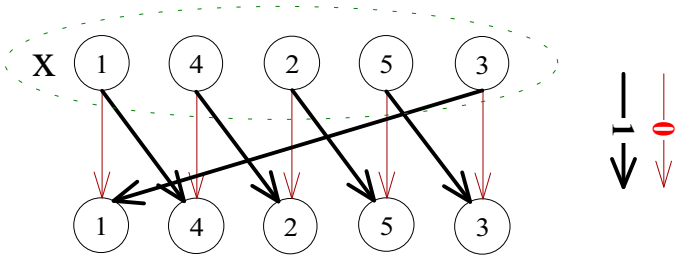
Weiter gilt natürlich $\tau^{-1} \circ id \circ \tau = \tau^{-1} \circ \tau = id$.

Mittels τ^{-1} und τ haben wir jetzt eine Transformation von π nach σ gefunden. Diese Transformation erreicht man einfach durch Umbenennung aller Knoten i der ersten und letzten Schicht in τ(i). Die Umbenennung in der ersten Schicht wirkt auf die Permutation π gerade wie $\tau^{-1} \circ \pi$, da $\tau^{-1}(j) = i$ gdw. $j = \tau(i)$.

Betrachten wir das Beispiel 2:

Das PBP berechnet die Funktion $f(x) = x$ via $\pi=(1,2,3,4,5)$. Sei nun beispielsweise $\sigma=(1,4,2,5,3)$. $\Rightarrow \tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 5 & 3 \end{pmatrix}$

Wir ändern die Beschriftung der Knoten in der ersten und letzten Schicht des Beispiels gemäß τ.



Wie man sich schnell klarmacht, berechnet dieses PBP ebenfalls $f(x) = x$, allerdings via $\sigma=(1,4,2,5,3)$.

Als Konsequenz von Lemma 2 sind wir, wann immer wir ein PBP für eine Funktion f haben, in der Wahl der berechnenden Permutation π völlig frei; können diese also in Zukunft frei wählen.

Lemma 3

Sei P ein PBP, das f π-berechnet, wobei π eine zyklische Permutation ist. Sei weiter auch σ eine beliebige zyklische Permutation. Dann gibt es ein PBP P' der selben Größe wie P, welches $\neg f$ σ-berechnet.

Beweis:

Wir benennen alle Knoten i der letzten Schicht von P in $\pi^{-1}(i)$ um.

Fall1: $f(a) = 1$

$P(i_1, ..., i_5) = \pi(i_1), ..., \pi(i_5) = \pi$

$\Rightarrow \pi^{-1}(P(i_1, ..., i_5)) = \pi^{-1}(\pi(i_1), ..., \pi(i_5)) = id$

$\Rightarrow f'(a) = 0$

Fall1: $f(a) = 0$

$P(i_1, ..., i_5) = id$

$\Rightarrow \pi^{-1}(P(i_1, ..., i_5)) = \pi^{-1}(id) = \pi^{-1}$

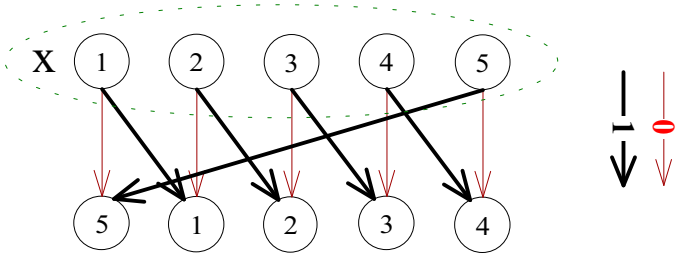
$\Rightarrow f'(a) = 1$

$\Rightarrow f'(a) = \neg f(a)$

Wegen Lemma 2 sind wir in der Wahl von σ frei.

Betrachten wir das Beispiel 2:

Wir ändern die Beschriftung der Knoten in der letzten Schicht des Beispiels in π^{-1}.



Wie man sich schnell klarmacht, berechnet dieses PBP ebenfalls $\neg f(x) = x$ via $\sigma = (1,5,4,3,2)$.

Wir haben schon Permutations-Branching-Programme für x und die NOT-Funktion gefunden. Zur Simulation beliebiger Schaltkreise benötigen wir noch eine Möglichkeit z.B. die AND-Funktion als PBP darzustellen.

Lemma 4

Sei P ein PBP, das f π-berechnet, wobei π eine zyklische Permutation ist. Sei weiter Q ein PBP, das g σ-berechnet, wobei auch σ eine zyklische Permutation ist. Dann gibt es ein PBP R der Größe $2\cdot\|P\| + 2\cdot\|Q\|$, welches $f \wedge g$ φ-berechnet.

Beweis:

Aufgrund von Lemma 2 können wir π und σ frei wählen und setzen $\pi = (1,2,3,4,5)$ und $\sigma = (1,3,5,4,2)$. Mit dem selben Argument finden wir 2 PBP P' und Q' gleicher Größen, die die Funktion f via $\pi^{-1} = (5,4,3,2,1)$ bzw. die Funktion g via $\sigma^{-1}=(2,4,5,3,1)$-berechnen. Dann berechnet das PBP $R = P \circ Q \circ P' \circ Q'$ (Hintereinanderreihung der 4 PBPs) die Funktion $f \wedge g$ via φ mit $\varphi = (1,3,2,5,4)$.

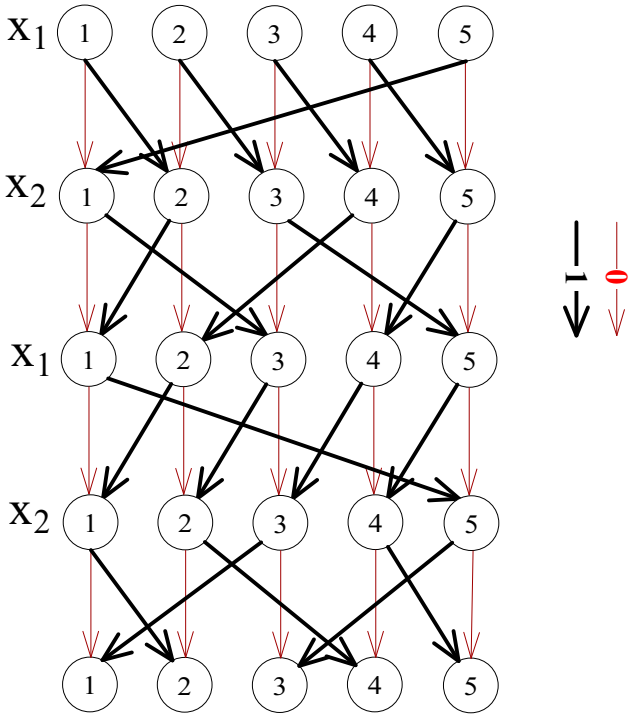
Fall 1: $f(a) = g(a) = 0 \Rightarrow f \wedge g = 0$
$R(a) = id \circ id \circ id \circ id = id$

Fall 2: $f(a) = 0 \;\; g(a) = 1 \quad \Rightarrow f \wedge g = 0$
$R(a) = id \circ \sigma \circ id \circ \sigma^{-1} = \sigma \circ \sigma^{-1} = id$

Fall 3: $f(a) = 1 \;\; g(a) = 0 \quad \Rightarrow f \wedge g = 0$
$R(a) = \pi \circ id \circ \pi^{-1} \circ id = \pi \circ \pi^{-1} = id$

Fall 4: $f(a) = g(a) = 1 \Rightarrow f \wedge g = 1$
$R(a) = \pi \circ \sigma \circ \pi^{-1} \circ \sigma^{-1} = (1,2,3,4,5) \circ (1,3,5,4,2) \circ (5,4,3,2,1) \circ (2,4,5,3,1)$
$R(a) = (1,3,2,5,4) = \varphi.$

Beipiel 4: $(x_1 \wedge x_2)$



Man überzeugt sich leicht, dass dieses PBP die Funktion $x_1 \wedge x_2$ φ-berechnet mit $\varphi = (1,3,2,5,4)$.

Wir haben nun Permutations-Branching-Programm-Entsprechungen für die Funktionen x, NOT und AND gefunden und können somit den ersten Teil des Beweises führen.

NC1 ⊆ w-5-BP:

Beweis durch Induktion über die Schaltkreistiefe d.
Sei $f \in B_n$ eine boolesche Funktion, die durch einen NC^1-Schaltkreis S der Tiefe $d = O(\log n)$ berechnet wird. O.B.d.A. enthalte S nur NOT / AND-Gatter (U_2-Schaltkreis).
Beh.: Dann existiert ein (Permutations)-Branching-Programm der Breite 5 und polynimialer Größe, welches *f* berechnet.

I.A.: $d = 0 \rightarrow$ trivial

I.S.: $d = d' + 1$

Fall 1:

- $S = NOT(S')$; wir haben ein PBP P', welches S' berechnet
- Mittels Lemma 3 finden wir ein PBP P der Breite 5 und der Tiefe von P', welches S berechnet.

Fall 2:

- $S = AND(S', S'')$; wir haben PBPs P'und P'', welche S' und S'' berechnen
- Mittels Lemma 4 finden wir ein PBP P der Breite 5 und der Tiefe $2 \cdot d(P') + 2 \cdot d(P'')$.

Somit haben wir ein (Permutations-)Branching-Programm P erzeugt mit Breite 5 und der Tiefe $\leq 4^d$.
Es gilt: $d = O(\log n)$, d.h. $d = c \cdot \log n$.
$\Rightarrow BP(P) \leq 5 \cdot 4^d = 5 \cdot 4^{c \cdot \log n} = 5 \cdot (2^{\log n})^c \cdot (2^{\log n})^c = 5 \cdot n^{2c} \in POLY$.
D.h. unser Branching-Programm hat polynomiale Größe.

□

Damit haben wir bereits gezeigt, dass majotity $\in$ w-5-BP.

NC1 $\supseteq$ w-5-BP:

Beweis durch Induktion über die Branching-Programm-Tiefe d.

Sei $f \in B_n$ eine boolesche Funktion, die durch ein Branching-Programm P konstanter Breite k und polynomialer Größe berechnet wird. O.B.d.A. sei die Tiefe von P eine Zweierpotenz, d.h. $d(P) = 2^m$.
Nehmen wir weiter an, wir hätten einen Schaltkreis C konstanter Größe / Tiefe, der die Konkatenation zweier Funktionen, die durch beliebige Branching-Programme konstanter Breite erzeugt werden, berechnen könnte.

Beh.: Dann existiert ein NC^1-Schaltkreis S logarithmischer Tiefe der f berechnet.

I.A.: $d(P) = 2^0 = 1 \rightarrow$ trivial ($d(S) = 1$)

I.S.: $d(P) = 2^{k+1}$

- Teile P in der Mitte in die Branching-Programme P' und P'' der Tiefe 2^k, für die bereits Schaltkreise S' und S'' existieren.
- Erzeuge einen Schaltkreis $S = C(S', S'')$, welcher P simuliert.

$\Rightarrow d(S) = d(C) + \max(d(S') + d(S'')) = c \cdot \log n = O(\log n)$

Für polynomial große Branching-Programme hat ein so konstruierter Schaltkreis logarithmische Tiefe.

Wir müssen nun noch die Existenz des benutzten Schaltkreises c zeigen.:

> Sei Q ein zusammenhängender Subgraph von P. Q hat höchstens die Breite k. Die gesamte Funktionalität von Q wird durch sein Ein-/Ausgabeverhalten f_x^Q bestimmt.
> $f_x^Q(i) = j$, mit $i,j \in \{1,...,k\}$ bedeutet, beginnend bei Knoten i gelangt man bei Eingabe x zu Knoten j. Offensichtlich gibt es höchstens k^k = konstant viele verschiedene solcher Funktionen (Variation von k aus k Ausgabeknoten mit Wiederholung). Diese verschiedenen Funktionen kann man durchnummerieren und mit $k \cdot \log k$ vielen bits kodieren. Das bedeutet, um die Konkatenation zweier Branching-Programme konstanter Breite k zu realisieren, kann man **einen** Schaltkreis konstanter Größe konstruieren mit fan-in $2 \cdot k \cdot \log k$ und fan-out $k \cdot \log k$, der so kodiert das Verhalten der Branching-Programme beschreibt.

Nach der letzten Stufe des so konstruierten Schaltkreises S benötigt man lediglich noch einen Entscheider, der die $k \cdot \log k$-Kodierung in eine true / false - Entscheidung transformiert.

□

4. Folgerungen

Offensichtlich gilt:

$$\textbf{w-k-BP} \subseteq \textbf{w-(k+1)-BP} .$$

Aus dem Barrington-Satz folgt direkt:

$$\textbf{w-5-BP} = \textbf{w-(5+x)-BP} . \quad x \geq 0$$

Da die majority-Funktion in w-2-BP exponentielle Komplexität hat, weiß man:

$$\textbf{w-2-BP} \subset \textbf{w-5-BP} .$$

Ob die Hierarchie dazwischen echt ist, ist bisher noch offen.