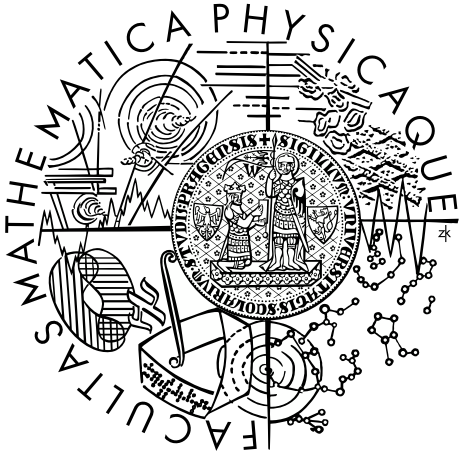
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Radek Hušek

Vlastnosti intervalových booleovských funkcí

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce:	doc. RNDr. Ondřej Čepek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Diskrétní modely a algoritmy

Praha 2014

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne ..... Podpis autora

Název práce: Vlastnosti intervalových booleovských funkcí

Autor: Radek Hušek

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: doc. RNDr. Ondřej Čepek, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Tato práce řeší problém rozpoznávání k-intervalových boolovských funkcí. Na vstup boolovské funkce můžeme nahlížet jako na binární zápis přirozeného čísla. Funkce je k-intervalová, pokud – při takto interpretovaném vstupu – nabývá hodnoty jedna právě pro vstupy z daných nejvýše k intervalů. Tento problém je pro obecné boolovské funkce zadané DNF coNP-těžký. Proto se zabýváme případem, kdy DNF náleží do dané řešitelné třídy (třída je řešitelná, pokud pro DNF z ní umíme řešit falsifikovatelnost v polynomiálním čase a je uzavřená na částečná dosazení), a ukazujeme, že v tomto případě je úloha pro pevné k řešitelná v polynomiálním čase.

Klíčová slova: booleovské funkce, intervalové funkce, polynomiální algoritmy

Title: Properties of interval Boolean functions

Author: Radek Hušek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Ondřej Čepek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Boolean function f is k-interval if – input vector viewed as n-bit number – f is true for and only for inputs from given (at most) k intervals. Recognition of k-interval fuction given its DNF representation is coNP-hard problem. This thesis shows that for DNFs from a given solvable class (class $\mathcal{C}$ of DNFs is solvable if we can for any DNF $\mathcal{F} \in \mathcal{C}$ decide $\mathcal{F} \equiv 1$ in polynomial time and $\mathcal{C}$ is closed under partial assignment) and fixed k we can decide whether $\mathcal{F}$ represents k-interval function in polynomial time.

Keywords: Boolean functions, interval functions, polynomial time algorithms

Obsah

Úvod

Tato práce se zabývá problémem rozpoznání k-intervalových boolovských funkcí. Intervalové funkce byly zavedeny v článku [1], kde byl studován následující problém: Máme zadána n-bitová čísla a a b a chceme zkonstruovat co nejkratší DNF reprezentující funkci, která je pravdivá právě pro $x \in [a, b]$. V článku [2], který je hlavní motivací pro tuto práci, je zkoumán problém opačný: Máme zadanou DNF a zkoumáme, zda (pro nějakou permutaci proměnných) reprezentuje intervalovou funkci.

Obecně je tento problém coNP-těžký: Funkce je konstantně 1, právě když je intervalová pro interval $[0, 2^n - 1]$ (pro funkci na n proměnných). Naopak dostatečným protipříkladem jsou $x < y < z$ takové vstupy, že funkce je 1 pro x a z a 0 pro y. V [2] bylo ukázáno, že pro DNF z řešitelné třídy lze rozhodnout, zda reprezentují 1-intervalovou funkci v polynomiálním čase. V navazujícím článku [3] bylo ukázáno, že dokonce v lineárním čase lze rozhodnout, zda positivní DNF je 2-intervalová.

Cílem této práce je algoritmus pro rozpoznání, zda DNF z řešitelné třídy je k-intervalová, který pracuje pro pevné k v polynomiálním čase. Protože je pojem intervalové funkce mírně nešikovný (kupř. množina k-intervalových funkcí není uzavřená vůči negaci), nahradíme jej pojmem zlomu a k-zlomových funkcí. Také oddělíme problém nalezení permutace, vůči níž má daná funkce málo zlomů (resp. intervalů), od výpočtu zlomů (tj. okrajů intervalů) samotných. Nejprve tedy definujeme zlom, poté předvedeme algoritmus pro výpočet zlomů funkce a nakonec algoritmus pro nalezení optimální permutace proměnných pro danou funkci.

Značení

Na začátek uveďme ještě několik poznámek o značení, které budeme v průběhu práce užívat. Disjunkci značíme $\vee$, konjunkci $\wedge$ (a obdobně jako značku pro násobení ji budeme běžně vynechávat) a negaci pruhem na negovanou hodnotou (např. $\overline{x}$). Množina přirozených čísel obsahuje nulu a všechny indexy začínají od 1. Množinu (resp. třídu) všech funkcí tvaru $A \to B$ budeme značit $[A \to B]$. Třídu všech objektů značíme $\mathcal{V}$.

Dále budeme pracovat s multimnožinami. Ty jsou formálně zavedeny v příloze A a odpovídají standardním definicím s výjimkou operace rozdílu multimnožin definovaného vztahem*:

$$A \setminus B := x \mapsto \begin{cases} 0 & \text{pokud } x \in B \\ A(x) & \text{jinak} \end{cases}$$

Multimnožinové sjednocení (tj. sjednocení zachovávající násobnosti) budeme značit $\uplus$. Pro úplnější přehled obecného značení vizte přílohu A. Značení týkající se boolovských funkcí bude zaváděno průběžně dle potřeby.

*Formálně pohlížíme na multimnožinu jako na funkci $A : \mathcal{V} \to \mathbb{N}$, kde $A(x)$ říká, kolikrát se prvek x v multimnožině A vyskytuje.

Členění práce

- První kapitola obsahuje obecné informace o boolovských funkcích, disjunktivní normální formě, řešitelných třídách a intervalových funkcích.
- Ve druhé kapitole nahradíme intervalové funkce pojmem k-zlomové funkce.
- Třetí kapitola podává popis datových struktur užívaných k reprezentaci DNF algoritmy popsanými dále a ukazuje, že k dosažení uvedené složitosti není třeba RAM, ale stačí Pointer Machine.
- Čtvrtá kapitola popisuje algoritmus pro výpočet množiny zlomů boolovské funkce vůči dané permutaci jejích proměnných.
- Kapitoly pět a šest postupně budují algoritmus pro rozpoznání k-zlomových funkcí a analyzují jeho složitost.
- Příloha A obsahuje shrnutí používané matematické notace, která se neváže přímo k boolovským funkcím.

1. Boolovské funkce obecně

V této kapitole nejprve podáme stručný přehled o boolovských funkcích obecně. Poté se zaměříme na části teorie boolovských funkcí důležité pro tuto práci – především disjunktní normální formu, implikanty a řešitelné třídy boolovských funkcí. Na závěr představíme výzkum, jímž je motivovaná tato práce – intervalové funkce a známé výsledky o nich.

1.1 Boolovské funkce

Boolovské funkce patří mezi nejstudovanější matematické objekty. Běžně se boolovskou funkcí myslí funkce tvaru $f : \{0,1\}^k \to \{0,1\}$, kde k je arita dané funkce. Protože ale budeme boolovské funkce reprezentovat pomocí logických formulí a budeme pracovat s pořadím proměnných, budeme na boolovské funkce nahlížet jako na funkce tvaru* $f : [\mathrm{Var}(f) \to \{0,1\}] \to \{0,1\}$, kde $\mathrm{Var}(f)$ je množina proměnných funkce f. Funkcím tvaru $\mathrm{Var}(f) \to \{0,1\}$ říkáme (úplná) ohodnocení. Částečným funkcím z $\mathrm{Var}(f)$ do $\{0,1\}$ říkáme částečná ohodnocení.

Definice 1.1: *Boolovská funkce*
Boolovská funkce je funkce tvaru

$$f : [\mathrm{Var}(f) \to \{0,1\}] \to \{0,1\}$$

Množině $\mathrm{Var}(f)$ říkáme množina proměnných funkce f a předpokládáme, že je konečná.

Definice 1.2: *Ohodnocení proměnných*
Mějme boolovskou funkci f. Funkci tvaru $\varphi : \mathrm{Var}(f) \to \{0,1\}$ říkáme (úplné) ohodnocení proměnných funkce f. Funkci $\varphi' : V \to \{0,1\}$, kde $V \subseteq \mathrm{Var}(f)$, říkáme částečné ohodnocení†. Aplikaci částečného ohodnocení φ' na funkci f značíme $f[\varphi']$.

Rovnou zaveďme značení pro konstantní boolovské funkce – funkci přiřazující všem ohodnocením hodnotu 1 značíme $\mathbf{1}_V$ a funkci přiřazující všem ohodnocením hodnotu 0 budeme značíme $\mathbf{0}_V$, kde V je v obou případech množina proměnných dané funkce. Pokud bude množina proměnných zřejmá z kontextu, budeme tento index vynechávat. Kromě konstantních funkcí máme i konstantní ohodnocení‡ – $\nvDash$ přiřazující všem proměnným hodnotu 0 a $\nVdash$ přiřazující všem 1.

*Tedy definičním oborem jsou funkce tvaru $\mathrm{Var}(f) \to \{0,1\}$. Tato definice je ekvivalentní běžné definici, protože n-složkový vektor $x \in A^n$ můžeme interpretovat jako funkci $x : [1,n] \to A$ a naopak.

†Speciálně každé úplné ohodnocení je také částečné ohodnocení.

‡U ohodnocení nezavádíme indexy pro množinu proměnných, protože máme-li ohodnocení φ množiny proměnných V a boolovskou funkci f splňující $\mathrm{Var}(f) \subseteq V$, je přirozené připustit značení $f(\varphi)$ jako zkratku za $f(\varphi \restriction \mathrm{Var}(f))$. Tedy formálně „množinou“ proměnných pro ohodnocení $\nvDash$ a $\nVdash$ je třída všech objektů $\mathcal{V}$. Značka $g \restriction A$ znamená restrikci funkce g na množinu A (předpokladem je, že A je podmnožinou definičního oboru g).

Na boolovských funkcích definujeme relaci $\leq$ přirozeným způsobem po bodech.

Definice 1.3: *Relace* $\leq$
Buď f a g boolovské funkce. Pak*

$$f \leq g := (\forall \varphi)(f(\varphi) \leq g(\varphi))$$

Pro práci s boolovskými funkcemi je potřebujeme nějak reprezentovat. Nejběžnějšími reprezentacemi boolovských funkcí jsou pravdivostní tabulky a logické formule. Pravdivostní tabulky pomineme, neboť jsou prostorově velmi neefektivní† – pro funkci o n proměnných zabírá pravdivostní tabulka 2^n bitů‡.

Opět existuje mnoho variant reprezentací pomocí logických formulí, které se vzájemně liší omezeními na tvar daných formulí. Pro nás bude významná disjunktivní normální forma§, informace o jiných reprezentacích nalezne čtenář kupříkladu v knize [4]. Disjunktivní normální forma je formule výrokové logiky, která je disjunkcí termů, term je konjunkce literálů a literál je proměnná či její negace.

Definice 1.4: *Disjunktivní normální forma (DNF)*
Literál je proměnná (positivní literál) či její negace (negativní literál). **Term** je bezesporná¶ (ne nutně neprázdná) konjunkce literálů. **Disjunktivní normální forma (DNF)** je disjunkce termů (opět nemusí být neprázdná).

Prázdný term má hodnotu 1, prázdná DNF 0. Termy lišící se pouze pořadím literálů považujeme za identické, obdobně pro DNF a pořadí termů. DNF je **triviální**, pokud neobsahuje žádný term nebo obsahuje prázdný term, netriviální jinak.

Literál je **univerzální**, pokud se vyskytuje ve všech termech dané DNF. Term je **lineární**, je-li tvořen právě jedním literálem.

Značení: DNF vetšinou značíme $\mathcal{F}$. Bude-li DNF, s níž pracujeme, zřejmá z kontextu, budeme značit n počet jejích proměnných a l počet literálů.

Důležitou úlohou je určit, zda daná DNF je falsifikovatelná – tedy existuje-li ohodnocení, které ji nesplňuje. Tento problém je NP-úplný‖.

*Povšimněme si, že ač formálně požadujeme $f(\varphi) \leq g(\varphi)$ pro každé ohodnocení z $[\mathcal{V} \to \{0,1\}]$, stačí uvažovat pouze ohodnocení tvaru $[\mathrm{Var}(f) \cup \mathrm{Var}(g) \to \{0,1\}]$, protože proměnné, které se nevyskytují v ani jedné z funkcí nemohou ovlivnit jejich hodnotu.

†Zde se myslí prostorová složitost uložení boolovských funkcí, s nimiž se člověk běžně setká – typicky ve formě nějaké formule. Z čistě teoretického hlediska je tabulka velmi rozumná reprezentace, protože existuje 2^{2^n} boolovských funkcí na n pevně daných proměnných, a tedy pro reprezentaci jedné z nich je průměrně potřeba alespoň 2^n bitů bez ohledu na použité kódování.

‡Pokud máme přirozené uspořádání na proměnných, jinak potřebujeme ještě uložit pořadí proměnných.

§Výraz disjunktivní normální forma budeme dále zkracovat jako DNF. Stejně dobře bychom mohli celou následující teorii budovat pro konjunktivní normální formu a volba DNF je spíše záležitostí zvyku než nějakého hlubšího významu.

¶Tj. neobsahuje zároveň x a $\overline{x}$ pro žádnou proměnnou x.

‖Slavná Cookova věta. Důkaz vizte třeba [5, 6].

Pro větší pohodlí (a s přihlédnutím k tomu, jak jsou DNF typicky reprezentovány v paměti počítače) budeme na DNF nahlížet jako na množiny termů a na termy jako na množiny literálů. V souladu s tím budeme značit* $|\mathcal{F}|$ počet termů a $\|\mathcal{F}\|$ počet literálů DNF $\mathcal{F}$.

Pro úplnost uveďme, že každá boolovská funkce na pevné množině proměnných je jednoznačně určena množinou svých truepointů (tj. bodů v nichž nabývá hodnoty 1), pro každý truepoint existuje term, který má hodnotu 1 pouze v tomto bodě, a tedy DNF, která je disjunkcí všech termů odpovídajících truepointům dané funkce, tuto funkci reprezentuje.

Definice 1.5: *Truepoint, falsepoint*
Mějme boolovskou funkci f. Bod (ohodnocení) $\varphi \in [\mathrm{Var}(f) \to \{0,1\}]$ nazveme truepoint (resp. falsepoint), pokud $f(\varphi) = 1$ (resp. $f(\varphi) = 0$).

Významnou třídou DNF jsou positivní (a k nim symetricky negativní) DNF.

Definice 1.6: *Positivní (a negativní) funkce*
DNF je positivní (resp. negativní), pokud je složena pouze z positivních (resp. negativních) literálů. Funkce je positivní (resp. negativní), pokud ji lze reprezentovat† positivní (resp. negativní) DNF.

1.2 Implikatny a primární implikanty

O termu t řekneme, že je implikantem funkce f, pokud $t \leq f$. Mluvíme-li o implikantu DNF, rozumíme tím implikant funkce, kterou daná DNF reprezentuje. Speciálně každý term $t \in \mathcal{F}$ je implikantem DNF $\mathcal{F}$. Dále term t nazveme primárním implikantem, pokud t je implikant a pro žádné $t' \neq t$ neplatí $t \leq t' \leq f$. Povšimněme si, že pro t a t' termy platí $t \leq t'$ právě tehdy když $t \supseteq t'$.

Definice 1.7: *Implikant, primární implikant*
Mějme boolovskou funkci f. Pak term t je implikantem f, pokud $t \leq f$. Implikant t je primární, pokud neexistuje term $t' \neq t$ takový, že $t \leq t' \leq f$.

Mějme DNF $\mathcal{F}$. Pak pro každý term $t \in \mathcal{F}$ existuje minimální (do inkluze) $t' \subseteq t$ takové, že $t' \leq \mathcal{F}$. Tento t' je primárním implikantem $\mathcal{F}$. Navíc platí‡ $\mathcal{F} \cup \{t\} \equiv \mathcal{F}$, vždy když $t \leq \mathcal{F}$, takže $\mathcal{F}'$ vzniklá z $\mathcal{F}$ zaměněním termu t za t' reprezentuje tutéž funkci. Pokud budeme tento proces opakovat, dojdeme k DNF reprezentující původní funkci, která je složena pouze z primárních implikantů. To motivuje definici primární DNF. Pokud vezmeme množinu všech primárních implikantů dané funkce f a vytvoříme z nich DNF, tak tato DNF reprezentuje funkci f, je primární a říkáme jí úplná primární DNF.

*Toto značení je opačné oproti [4], ale pro nás přirozenější kvůli interpretaci DNF jako množiny termů.

†Tato definice je mírně nestandardní, ale pro naše účely dostatečná. Běžně je zavedena nejprve positivita v proměnné (f je positivní v x, pokud $f[x := 0] \leq f[x := 1]$) a positivní funkce jsou ty, které jsou positivní ve všech proměnných. Fakt, že funkce je positivní právě, když ji lze reprezentovat positivní DNF, je ukázán jako věta.

‡Povšimněme si značení, $=$ značí rovnost DNF (pořadí v termů v DNF a literálů v termech vždy zanedbáváme), zatímco $\equiv$ je rovnost funkcí, danými DNF reprezentovaných.

Definice 1.8: *Primární DNF*
DNF $\mathcal{F}$ je primární, pokud každý term $t \in \mathcal{F}$ je jejím primárním implikantem. DNF je úplná primární*, pokud je složena právě ze všech svých primárních implikantů.

K úplné primární DNF se lze dobrat i „konstruktivnějším“ přístupem. Stačí provádět konsenzy, dokud vytvářejí nové termy, pak provést všechny možné absorbce a výsledkem je úplná primární DNF. Ještě je třeba říci, co konsenzy a absorbce jsou.

Dva termy t_1 a t_2 mají konflikt v proměnné x, pokud $x \in t_1$ a $\overline{x} \in t_2$ nebo naopak. Konsenzus je definován pro termy $t_1 = xA$ a $t_2 = \overline{x}B$ mající konflikt pouze v proměnné x vztahem $\mathrm{cons}(t_1, t_2) := AB$. Absorbcí rozumíme vypuštění termu t z DNF $\mathcal{F}$, pokud $\mathcal{F}$ obsahuje term $t' \neq t$ takový, že $t' \subseteq t$ (tj. $t \leq t'$). Absorbce ani přidání termu, který je konsenzem termů v DNF obsažených, nemění funkci, kterou DNF reprezentuje. Pro formální důkaz tvrzení, že úplnou primární DNF lze získat pomocí konsenzů a absorbcí vizte [4].

Definice 1.9: *Konflikt a konsenzus*
Termy t a t' mají konflikt v proměnné x, pokud $x \in t$ a $\overline{x} \in t'$ nebo naopak. Buď t_1 a t_2 termy, které mají konflikt v právě jedné proměnné, a tu označme x. Pak konsenzus termů t_1 a t_2 je term

$$\mathrm{cons}(t_1, t_2) := (t_1 \cup t_2) \setminus \{x, \overline{x}\}$$

Pozorování 1.10: *O konsenzu implikantů*
Mějme t_1 a t_2 implikanty boolovské funkce f. Pak $\mathrm{cons}(t_1, t_2)$, pokud je definován, je také implikant f.

Důkaz. Zjevné. Každé ohodnocení $\mathrm{Var}(f)$, které splňuje $\mathrm{cons}(t_1, t_2)$, splňuje všechny literály t_1 vyjma (BÚNO) x a všechny literály t_2 vyjma $\overline{x}$. Z literálů x a $\overline{x}$ je však vždy právě jeden splněn. ♡

Primární DNF nemusí být určena jednoznačně – např. DNF

$$\mathcal{F}_1 = a\overline{b} \vee b\overline{c} \vee c\overline{a}$$
$$\mathcal{F}_2 = \overline{a}b \vee \overline{b}c \vee \overline{c}a$$

reprezentují tutéž funkci a obě jsou primární†. Naopak úplná primární DNF je z definice jednoznačně určena. Úplné primární DNF mají ještě jednu příjemnou vlastnost – po libovolném částečném dosazení do úplné primární DNF stačí provést absorbce a výsledkem je opět úplná primární DNF.

Pozorování 1.11: *O dosazování do úplné primární DNF*
Buď $\mathcal{F}$ úplná primární DNF. Pak $\mathcal{F}'$ vzniklá z $\mathcal{F}$ dosazením za proměnnou x a provedením všech absorbcí je také úplná primární.

*V literatuře se lze setkat i s označením (Blakova) kanonická DNF dle Archiho Blaka, který ji zavedl v roce 1937.

†Protože funkce jimi reprezentovaná nemá žádný lineární implikant – vizte obrázek 1.1.

Obrázek 1.1: Graf DNF $\mathcal{F}_1$ (a $\mathcal{F}_2$) vůči libovolné permutaci proměnných

Důkaz. Z postupu konstrukce úplné primární DNF pomocí absorbcí a konsenzů víme, že DNF je úplná primární právě tehdy, když je každý konsenzus jejích dvou termů absorbován nějakým jejím termem a není možné provést žádnou absorbci. Zbývá nahlédnout, že toto platí pro $\mathcal{F}'$.

Sporem. Mějme $t'_1, t'_2 \in \mathcal{F}'$ takové, že neexistuje žádný term $t' \in F'$, který by absorboval $\text{cons}(t'_1, t'_2)$. Termy t'_1 a t'_2 vznikly z nějakých termů t_1 a t_2 DNF $\mathcal{F}$. Protože $\mathcal{F}$ je úplná primární, tak existuje term $t \in \mathcal{F}$, který absorbuje konsenzus $\text{cons}(t_1, t_2)$. Z termu t vznikl dosazením term $t' \in \mathcal{F}'$, který absorbuje konsenzus $\text{cons}(t'_1, t'_2)$ (t nemohl dosazením za x zaniknout, protože t_1 ani t_2 dosazením nezanikly). Spor. ♡

Na druhou stranu mohou být úplné primární DNF až exponenciálně dlouhé vůči jiným DNF reprezentujícím tutéž funkci. Jako příklad uveďme DNF

$$\mathcal{G} = \bigvee_{i=1}^{k} (a_i b_i \overline{a_{i+1}} \vee a_i b'_i \overline{a_{i+1}})$$

Konsenzy z této DNF můžeme získat termy tvaru $a_1 b_1^* b_2^* \dots b_k^* a_{k+1}$, v nichž každé b_i^* může být čárkované b'_i nebo nečárkované b_i. To dává 2^k možných termů tohoto tvaru. Navíc jak termy tvaru $t_1 = a_i b_i^* \overline{a_{i+1}}$, tak i $t_k = a_1 b_1^* b_2^* \dots b_k^* a_{k+1}$ jsou primární implikanty* $\mathcal{G}$.

To ukazuje, že některé termy mohou být (a speciálně v úplných DNF často bývají) nadbytečné a jejich odstraněním se reprezentovaná funkce nezmění. To vede k definici redundantního termu. DNF, která redundantní termy neobsahuje, se říká iredundantní DNF. Místo úplných primárních DNF jsou pak často studovány iredundantní primární DNF. Poznamenejme, že ani iredundantní primární DNF nemusí být jednoznačně určena – příkladem budiž opět DNF $\mathcal{F}_1$ a $\mathcal{F}_2$, které jsou také iredundantní.

Definice 1.12: *Redundantní term, iredundantní DNF*
Mějme DNF $\mathcal{F}$. Term $t \in \mathcal{F}$ je redundantní, pokud $\mathcal{F} \setminus \{t\} \equiv \mathcal{F}$. DNF je iredundantní, pokud neobsahuje žádný redundantní term.

Stejně jako v článku [2] bychom mohli pracovat s primárními DNF, ale ve skutečnosti nám bude stačit mírně slabší vlastnost – esenciální DNF.

*Nejprve si všimneme, že každý term $t \in \mathcal{G}$ obsahuje nějaké a_i a $\overline{a_j}$. Tedy $\mathcal{G}$ je pro ohodnocení přiřazující všem a_* stejnou hodnotu nepravdivá. Tedy z termů t_1 a t_k nemůžeme vypustit žádný literál obsahující a_*. Také z termů typu t_1 nemůžeme vypustit b_i, protože takovýto term by původní absorboval a $\mathcal{G}$ by po této absorbci byla na daném b_i nezávislá, což není.

S vypuštěním BÚNO b_i (postup pro b'_i je analogický) z termu typu t_k – označme si ho t – je to trochu složitější – zkonstruujeme ohodnocení φ, které splňuje příslušný term po vypuštění b_i a nesplňuje $\mathcal{G}$. Ohodnocení φ bude splňovat všechny literály v $t' = t \setminus \{b_i\}$, $\varphi(b_i) = 0$, dále $\varphi(a_j) = 1\ \forall j \leq i$ a $\varphi(a_j) = 0\ \forall j > i$. Všechny zbylé proměnné b_* a b'_* ohodnocuje 0. Zjevně φ splňuje t'. Na druhou stranu jediné termy, které může splňovat v $\mathcal{G}$ jsou $a_i b_i \overline{a_{i+1}}$ nebo $a_i b'_i \overline{a_{i+1}}$ kvůli ohodnocení a_*. Ale b_i ani b'_i se v t' nevyskytují, takže φ oběma přiřazuje 0.

Definice 1.13: *Esenciální implikant a DNF*
Implikant je esenciální, pokud se vyskytuje ve všech primárních DNF reprezentujících danou funkci. DNF je esenciální, pokud obsahuje všechny své esenciální implikanty.

Pozorování 1.14: *O positivní DNF*
Každá positivní DNF je esenciální.

Důkaz. Zjevný, v positivní DNF neexistuje žádný konsenzus. ♡

Pozorování 1.15: *O netriviální esenciální DNF*
Buď $\mathcal{F}$ esenciální DNF. Pak $\mathcal{F}$ je netriviální právě tehdy, když je nekonstantní.

Důkaz. Když je DNF triviální je zjevně konstantní. Opačná implikace. Pokud má být $\mathcal{F} \equiv 0$, tak nemůže obsahovat žádný term, a také je triviální. Mějme $\mathcal{F} \equiv 1$. Pak je prázdný term implikantem $\mathcal{F}$, a protože je to jediný primární implikant, je též esenciální. ♡

Všimněme si, že na rozdíl od primární DNF zde požadujeme pouze, aby DNF obsahovala nějaké termy, a ne aby jimi byla tvořena. Důvodem tohoto rozdílu je – kromě zpříjemnění práce, protože nemusíme DNF čistit od redundantních termů – že DNF tvořená pouze esenciálními termy vůbec nemusí existovat. Třeba výše zmíněná funkce $\mathcal{F}_1$ nemá žádný esenciální term. Uveďme ještě pozorování, které bude užitečné v následujících kapitolách.

Pozorování 1.16: *O esencialitě lineárního termu*
Buď f nekonstantní boolovská funkce a t lineární term takový, že $t \leq f$. Pak je t esenciální implikant f.

Důkaz. BÚNO $t = \{x\}$ (případ $t = \{\overline{x}\}$ je analogický). Předně t je primární implikant, protože prázdný term není implikantem f kvůli nekonstantnosti. Navíc žádný jiný primární implikant t' nemůže obsahovat x (protože $t' \leq t$) ani $\overline{x}$ (protože konsenzem t' a t bychom získali implikant t'' takový, že $t' \leq t''$). Pokud by tedy t nebyl esenciální implikant, tak existuje DNF $\mathcal{F}$ reprezentující f, ve které se proměnná x nevyskytuje. To spolu s $f[x := 1] \equiv 1$ dává spor s nekonstantností f. ♡

1.3 Řešitelné třídy

Převod DNF na primární i esenciální tvar je NP-těžký problém – pokud je DNF konstantně 1, je prázdný term jejím implikantem, a jelikož je to jediný primární implikant, je také esenciální. Tedy s pomocí esenciální DNF můžeme rozhodovat falsifikovatelnost DNF. Proto chceme-li polynomiální algoritmy, nemůžeme pracovat s obecnými DNF, ale musíme se omezit na nějakou jejich podtřídu*. Vhodným omezením jsou řešitelné třídy.

*V souladu s všeobecným přesvědčením předpokládáme P $\neq$ NP.

Definice 1.17: *Řešitelná třída DNF*
Třída DNF $\mathcal{C}$ je řešitelná, pokud

- existuje deterministický algoritmus, který řeší falsifikovatelnost pro $\mathcal{F} \in \mathcal{C}$ v polynomiálním čase
- a je uzavřená na částečná dosazení.

Zde uvedená definice řešitelné třídy je mírně slabší, než se běžně používá. Například článek [2] požaduje navíc uzavřenost na odstraňování redundantních termů a nahrazování termů jejich podtermy (v množinovém smyslu), pokud jsou tyto implikanty dané funkce. Bez těchto vlastností se obejdeme. Není totiž nutné mít iredundantní primární DNF. Stačí ji při převodu na primární tvar neprodloužit – tím jsme se zbavili podmínky na odstraňování redundantních termů. Navíc nepotřebujeme ani, aby výsledná primární DNF patřila do dané řešitelné třídy, k čemuž je nutné moci nahrazovat termy jejich primárními podtermy.

Chceme ukázat, že každou DNF $\mathcal{F}$ z řešitelné třídy $\mathcal{C}$ umíme v polynomiálním čase transformovat na primární DNF – pozor výsledná primární DNF již do třídy $\mathcal{C}$ patřit nemusí. Nejprve ukážeme, že rozhodnout $t \leq \mathcal{F}$ pro term t a $\mathcal{F} \in \mathcal{C}$ umíme v polynomiálním čase.

Pozorování 1.18: *O implikantu DNF z řešitelné třídy*
Mějme $\mathcal{C}$ řešitelnou třídu a DNF $\mathcal{F} \in \mathcal{C}$. Pak lze v polynomiálním čase ověřit, zda term t je implikantem $\mathcal{F}$.

Důkaz. Definujeme ohodnocení φ_t na všech proměnných, které se v t vyskytují, následovně:

$$\varphi_t(v) = \begin{cases} 1 & \text{pokud } v \in t \\ 0 & \text{pokud } \overline{v} \in t \end{cases}$$

Nyní vidíme, že $t \leq \mathcal{F}$ právě tehdy, když $\mathcal{F}[\varphi_t] \equiv 1$. Navíc $\mathcal{F}[\varphi_t] \in \mathcal{C}$, takže $\mathcal{F}[\varphi_t] \equiv 1$ umíme rozhodnout v polynomiálním čase. ♡

Nyní zbývá uvést algoritmus, který DNF převede na primární tvar. Algoritmus postupuje hladově. Vezme každý term a zkouší z něj vypouštět literály, přičemž kontroluje, zda výsledný term je stále implikantem původní DNF. Nakonec vezme všechny takto ořezané termy a prohlásí je za primární reprezentaci původní funkce.

Pozorování 1.19: *O algoritmu 1.2*
Algoritmus 1.2 převede vstupní DNF $\mathcal{F}$ na primární tvar v čase $\mathcal{O}(l(l + T(l)))$ – kde l je počet literálů $\mathcal{F}$ a $T(l)$ je čas nutný pro otestování falsifikovatelnosti formule o l literálech vzniklé částečným dosazením do $\mathcal{F}$ – aniž by ji při tom prodloužil.

Důkaz. Časová složitost algoritmu je zřejmá, protože vnitřní cyklus proběhne právě jednou pro každý literál vstupní DNF. Výsledná DNF také nebude delší než vstupní, protože z každého termu vstupní DNF vznikl jeden term výstupní vynecháním některých literálů. Zbývá ukázat, že výsledkem je primární DNF reprezentující původní funkci.

Input: DNF $\mathcal{F}$
Output: Primární DNF reprezentující $\mathcal{F}$

```
1  F' ← ∅
2  for t ∈ F do
3  |  t' ← t
4  |  for l ∈ t do
5  |  |  t_l ← t' \ {l}
6  |  |  if t_l ≤ F then t' ← t_l
7  |  end for
8  |  F' ← F' ∪ t'
9  end for
10 return F'
```

Algoritmus 1.2: Převod DNF do primárního tvaru

Označme $\mathcal{F}$ vstupní a $\mathcal{F}'$ výstupní DNF. Zjevně $\mathcal{F} \leq \mathcal{F}'$, protože pro každý term $t \in \mathcal{F}$ a jemu odpovídající term $t' \in \mathcal{F}'$ platí $t' \subseteq t$, což je ekvivalentní $t \leq t'$. Naopak pro každý term $t' \in \mathcal{F}$ z konstrukce platí $t' \leq \mathcal{F}$, takže $\mathcal{F}' \leq \mathcal{F}$. Tedy $\mathcal{F}$ a $\mathcal{F}'$ reprezentují tutéž funkci.

Primárnost $\mathcal{F}'$ nahlédneme sporem. Mějme term $t' \in \mathcal{F}'$ a $t'' \subset t'$ dokazující, že t' není primární implikant. Vezměme nějaký literál $l \in t' \setminus t''$. Při konstrukci termu t' jsme se někdy pokusili l odebrat z termu t, z něhož dalším odebíráním vzniklo t' (tj. $t' \subseteq t$). Protože jsme l neodebrali, tak neplatilo $t \setminus \{l\} \leq \mathcal{F}$. Ale $t'' \subseteq t \setminus \{l\}$, takže ani t'' nemůže být implikantem $\mathcal{F}$. Spor. ♡

Důsledek 1.20: *O převodu na primární tvar*
Mějme řešitelnou třídu $\mathcal{C}$. Pak existuje algoritmus, který převede každou DNF z této třídy do primárního tvaru v polynomiálním čase.

Důkaz. Stačí užít algoritmus 1.2. Dle pozorování 1.19 a vlastností řešitelné třídy poběží v polynomiálním čase. ♡

1.4 Intervalové boolovské funkce

Pojem intervalové funkce (v naší terminologii 1-intervalové) byl zaveden v článku [1] v roce 2005. Intervalová funkce $f_{[a,b]} : \{0,1\}^n \to \{0,1\}$ byla definována vztahem

$$f_{[a,b]}(x) := \begin{cases} 1 & \text{pokud } a \leq x \leq b \\ 0 & \text{jinak} \end{cases}$$

kde a a b jsou přirozená čísla mezi 0 a $2^n - 1$ a x je vektor nul a jedniček interpretovaný jako binární zápis celého čísla. Studován byl problém nalezení co nejkratší DNF (měřeno počtem termů) reprezentující danou intervalovou funkci. Motivací toho problému bylo zrychlení systémů pro automatizované testování hardware a software:

Některé z těchto systémů reprezentují požadavky na testovací vstupy pomocí boolovských formulí (v disjunktivním normálním tvaru). Intervalové funkce reprezentují celkem častý typ podmínky $a \leq x \leq b$ a čím kratší reprezentaci této podmínky pomocí DNF máme, tím je systém pracující s touto podmínkou rychlejší*.

V roce 2008 v článku [2] byl studován opačný problém – máme zadanou DNF a chceme rozhodnout, zda reprezentuje intervalovou funkci. Jelikož DNF nemá (na rozdíl od klasické formální definice boolovské funkce) nějaké implicitní přirozené uspořádání na proměnných, byl studovaný problém mírně zobecněn a byly uvažovány všechny možné permutace vstupních bitů. To vede k následující definici:

Definice 1.21: *Boolovská funkce vůči permutaci*
Mějme boolovskou funkci f a permutaci jejích proměnných π. Pak f vůči π rozumíme funkci $f_\pi : [0, 2^{|\pi|} - 1] \to \{0, 1\}$ definovanou tak, že $f_\pi(x)$, kde x je $|\pi|$-bitové číslo, je rovno hodnotě f pro ohodnocení proměnných, kde proměnné seřadíme dle permutace π a i-tá proměnná má hodnotu i-tého nejvyššího bitu x.

Definice intervalových funkcí pak byla zobecněna jak s přihlédnutím k různým permutacím proměnných, tak k více možným intervalům jedniček.

Definice 1.22: *Intervalová funkce*
Boolovská funkce f je k-intervalová, pokud existuje permutace $\pi \in \mathcal{S}_{\mathrm{Var}(f)}$ a posloupnost dvojic celých čísel† $(a_i, b_i)_{i=1}^{k}$ splňující $(\forall x : \ 0 \leq x < 2^{|\mathrm{Var}(f)|})$

$$f_\pi(x) = 1 \Leftrightarrow (\exists i)(a_i \leq x \leq b_i)$$

Rozhodnout, zda DNF $\mathcal{F}$ reprezentuje 1-intervalovou funkci je coNP-těžké. I pokud fixujeme permutaci proměnných π, tak určit, je-li $\mathcal{F}_\pi$ intervalová funkce, je coNP-úplné, protože f je tautologie (tj. vždy pravdivá funkce) právě tehdy, když $f_\pi = f_{[0,2^n-1]}$ (a zde zjevně na permutaci π nezáleží), a na vyvrácení teze, že f_π je 1-intervalová funkce, stačí $x < y < z$ takové, že $f_\pi(x) = f_\pi(z) = 1$ a $f_\pi(y) = 0$.

V článku [2] bylo ukázáno, že rozhodnout, zda reprezentuje 1-intervalovou funkci, lze pro positivní DNF v lineárním čase a pro DNF z řešitelné třídy v polynomiálním čase. Dále v článku [3] byl předveden algoritmus pro rozpoznání 2-intervalových funkcí zadaných positivní DNF v lineárním čase.

*Poznamenejme, že původní problém byl komplikován tím, že na výslednou DNF byl často kladen požadavek ortogonality – tj. aby každé ohodnocení splňovalo nejvýše jeden term. Ortogonalita byla důležitá proto, aby bylo možné testovací vstupy generovat náhodně s uniformním rozdělením.

†Speciálně může být $b < a$, což značí prázdný interval. Tedy k-intervalové funkce jsou podmnožinou $(k + 1)$-intervalových.

2. Optimální permutace

V předchozí kapitole jsme definovali intervalové boolovské funkce a ukázali, proč je zajímavé je studovat. Také jsme představili algoritmy pro rozpoznání obecných 1-intervalových a positivních 2-intervalových funkcí. Ty bychom nyní rádi zobecnili i pro víceintervalové funkce.

Nejprve nahradíme pojem intervalu pojmem zlom, který je z technického hlediska šikovnější – kupř. je na rozdíl od intervalu „lokální“ a také se struktura zlomů nemění negací funkce či jejích proměnných. Poté pomocí pojmu rekurzivně optimální permutace provedeme první (neúspěšný) pokus o zobecnění výše zmíněných algoritmů.

2.1 Zlomy a optimální permutace

V souladu s definicí k-intervalových funkcí budeme na boolovskou funkci f spolu s permutací jejích proměnných π nahlížet jako na funkci z počátečního úseku přirozených čísel do množiny $\{0, 1\}$ a budeme ji značit f_π.

Zlom je pak místo, kde f mění svou hodnotu – tj. pokud $f_\pi(x) \neq f_\pi(x+1)$, pak je mezi x a $x+1$ zlom. Z technických důvodů je nevhodné považovat za zlom místo mezi dvěma čísly, a proto formálně definujeme zlom jako hodnotu argumentu funkce za zlomem (tj. $x+1$ v předchozím příkladě).

> **Definice 2.1:** *Zlom*
> Fixujme boolovskou funkci f a π permutaci jejích proměnných*. Pak $x \in [1, 2^{|\pi|} - 1]$ je zlom, pokud $f_\pi(x-1) \neq f_\pi(x)$. Dále $\mathrm{Br}(f, \pi)$ značí počet zlomů funkce f vůči permutaci π.

Začneme-li uvažovat o počtu zlomů boolovské funkce, všimneme si, že bez ohledu na permutaci proměnných jsou první a poslední hodnota funkce f (tj. $f_\pi(0)$ a $f_\pi(2^{|\mathrm{Var}(f)|} - 1)$) stále stejné, a to $f(⊬)$ resp. $f(⊮)$. Tedy počet zlomů dané boolovské funkce musí zachovávat paritu. Počet zlomů se nezmění, přejdeme-li od funkce f k negované funkci $\overline{f}$ či znegujeme-li všechny literály, protože tento přechod odpovídá převrácení grafu funkce f dle vodorovné resp. svislé osy.

Poznamenejme ještě, že počet zlomů může být exponenciálně velký. Kupř. parita je funkce symetrická ve všech proměnných, takže všechny její permutace mají stejně zlomů a na n proměnných má alespoň 2^{n-1} zlomů†.

Pomocí počtu zlomů můžeme přirozeně definovat **optimální permutaci** funkce f jako takovou permutaci jejích proměnných, která minimalizuje počet zlomů. Zjevně optimální permutace nemusí být pouze jedna – kupř. pro konstantní funkci mají všechny její permutace nula zlomů, a tedy jsou všechny optimální.

*Dále budeme z „estetických“ důvodů užívat výrazu permutace funkce f, čímž vždy myslíme permutaci proměnných funkce f. Obdobně budeme říkat, že permutace π má zlom, jako zkratku za funkce f má vůči permutaci π zlom.

†Zlom je vždy alespoň mezi čísly $2x$ a $2x+1$, jelikož tato se liší pouze posledním bitem.

Definice 2.2: *Optimální permutace*
Fixujme boolovskou funkci f. Pak permutaci jejích proměnných π nazveme **optimální**, pokud funkce f vůči této permutaci má minimální možný počet zlomů (tj. $\mathrm{Br}(f,\pi) \leq \mathrm{Br}(f,\sigma) \ \ \forall\sigma \in \mathcal{S}_{\mathrm{Var}(f)}$).

Dále o funkci říkáme, že je k**-zlomová**, pokud má její optimální permutace nejvýše k zlomů. Povšimněme si úzkého vztahu mezi k-zlomovými a l-intervalovými funkcemi – každá l-intervalová funkce je $2l$, $(2l-1)$ nebo $(2l-2)$-zlomová podle toho, zda se žádný z jejích intervalů jedniček nedotýká okraje, resp. jeden interval, resp. oba.

Definice 2.3: *k-zlomová funkce*
Boolovská funkce je k**-zlomová**, má-li její optimální permutace nejvýše k zlomů.

Značení:

- $\mathrm{Perm}(f,k) := \left\{\pi \in \mathcal{S}_{\mathrm{Var}(f)} : \ \mathrm{Br}(f,\pi) \leq k\right\}$ – množina všech permutací s nejvýše k zlomy
- $\mathrm{Br}^*(f) := \min\left\{\mathrm{Br}(f,\pi) : \ \pi \in \mathcal{S}_{\mathrm{Var}(f)}\right\}$ – počet zlomů optimální permutace*
- $\mathrm{Perm}^*(f) := \mathrm{Perm}(f, \mathrm{Br}^*(f))$ – množina optimálních permutací

2.2 Rekurzivně optimální permutace

Pokusíme-li se zobecnit algoritmus pro rozpoznávání 1-intervalových funkcí z článku [2], první problém, na který narazíme, je paralelní rekurze, kde jednotlivé instance mezi sebou sdílejí informace†. Jednou z možností, jak se paralelní rekurze zbavit, je pokusit se ji nahradit sekvenční rekurzí – místo paralelního počítání jedné optimální permutace sekvenčně spočíst vhodné množiny (optimálních) permutací a proniknout je.

Z argumentace v článku [2] plyne, že pro rozpoznávání 2-zlomových funkcí skutečně stačí v průběhu rekurze spočíst množiny optimálních permutací pro levou a pravou podfunkci a zkontrolovat, že mají neprázdný průnik. Platí toto i pro funkce s více zlomy? Platí-li tato hypotéza pro funkci f a $\pi = \sigma.\tau$ její optimální permutaci, tak pro každý prefix σ platí, že je-li ψ libovolné ohodnocení proměnných ze σ, tak τ je optimální permutací pro $f[\psi]$ – permutaci π pak říkáme rekurzivně optimální.

Definice 2.4: *Rekurzivně optimální permutace*
Fixujme boolovskou funkci f. Pak permutaci jejích proměnných π nazveme **rekurzivně optimální**, pokud $\forall\sigma,\tau$ takové, že $\pi = \sigma.\tau$ a $\forall\psi \in [\sigma \to \{0,1\}]$ částečné ohodnocení platí, že τ je optimální permutací pro $f[\psi]$.

*Speciálně je-li f konstanta (nezaměňovat s konstantní funkcí) – tj. $\mathrm{Var}(f) = \emptyset$ – tak existuje právě jedna permutace jejích proměnných – a to prázdná permutace – takže $\mathrm{Br}^*(f) = 0$ a ne nekonečno.

†Viz článek [2], algoritmus 2, test na řádku 23.

Prozatím pomineme výpočetní složitost úlohy reprezentovat množinu všech optimálních permutací a zaměříme se pouze na zkoumání, které boolovské funkce mají rekurzivně optimální permutace. Zpočátku vypadá situace nadějně, protože – jak ukazuje následující pozorování – je každá permutace o nejvýše dvou zlomech optimální, a tedy i rekurzivně optimální. Intuice je, že dva zlomy jsou příliš málo, aby graf funkce vůči této permutaci mohl obsahovat neoptimální úsek.

> **Pozorování 2.5:** *O 2-zlomových optimálních permutacích*
> Buď f boolovská funkce a π permutace jejích proměnných. Pak je-li $\mathrm{Br}(f, \pi) \leq 2$, je π rekurzivně optimální pro f.

Důkaz. Je-li $\mathrm{Br}(f, \pi) = 0$, je f konstantní a všechny její permutace jsou rekurzivně optimální. Pro ostatní případy nahlédneme rekurzivní optimalitu sporem. Není-li π rekurzivně optimální, existuje σ a τ takové, že $\pi = \sigma.\tau$ a částečné ohodnocení $\psi : \sigma \to \{0, 1\}$ takové, že τ není optimální pro $f[\psi]$. Funkce $f[\psi]$ není konstantní, protože jinak by byly všechny její permutace optimální. Tedy $\mathrm{Br}(f[\psi], \tau) \geq \mathrm{Br}^*(f[\psi]) + 2 \geq 3$ (+2 a ne jen +1 díky zachování parity). Spor s $\mathrm{Br}(f, \pi) \leq 2$. ♡

Každá 3-zlomová permutace sice nemusí být optimální, ale ukážeme, že optimální 3-zlomové permutace jsou rekurzivně optimální. Tím bohužel platnost hypotézy, že každá boolovská funkce má rekurzivně optimální permutaci, (téměř) končí. Pro 5 zlomů totiž existuje protipříklad – funkce $ab \vee cd$.

> **Pozorování 2.6:** *O 3-zlomových optimálních permutacích*
> Je-li f 3-zlomová, jsou všechny její optimální permutace rekurzivně optimální.

Důkaz. Má-li optimální permutace f méně než 3 zlomy, tvrzení plyne z předchozího. Sporem. Mějme f a π její optimální permutaci takovou, že $\mathrm{Br}(f, \pi) = 3$ a σ prefix π (tj. $\pi = \sigma.\tau$) a částečné ohodnocení $\psi : \sigma \to \{0, 1\}$ dokazující, že π není rekurzivně optimální. Platí $\mathrm{Br}(f[\psi], \tau) = 3$, jinak by τ byla optimální dle pozorování 2.5.

Buď τ' optimální permutace $f[\psi]$. Všimneme si, že $f[\chi]$ je konstantní funkce pro všechna $\chi : \sigma \to \{0, 1\}$ různá od ψ. Tedy $\mathrm{Br}(f, \sigma.\rho) = \mathrm{Br}(f[\psi], \rho)$ pro všechna* ρ.

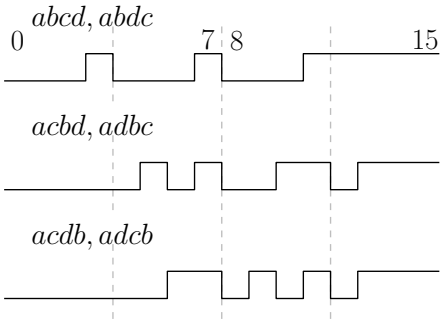
$$\mathrm{Br}(f, \pi) = \mathrm{Br}(f[\psi], \tau) > \mathrm{Br}(f[\psi], \tau') = \mathrm{Br}(f, \sigma.\tau')$$

Spor s optimalitou π. ♡

> **Pozorování 2.7:** *O funkcích bez rekurzivně optimální permutace*
> Existují funkce bez rekurzivně optimální permutace.

Důkaz. Příkladem budiž $ab \vee cd$. Uvažme optimální permutaci. BÚNO začíná a. Máme tedy podfunkce $L = f[a := 0] = cd$ a $R = f[a := 1] = b \vee cd$. Všechny optimální permutace R (bcd a bdc) začínají b, ale pro L jsou optimální pouze permutace cdb a dcb. ♡

*Zatím toto tvrzení podáváme k uvěření intuitivně. Rámec nutný pro formální důkaz bude formulován v kapitole 5.



Obrázek 2.1: Grafy všech permutací funkce $ab \vee cd$ začínajících a

Toto zjištění nám ničí naději na algoritmus pro rozpoznání k-zlomových funkcí, který by místo jedné optimální permutace vracel množinu všech optimálních permutací a vyhnul se tak synchronizaci rekurzivních volání, protože pouze optimální permutace podfunkcí ke konstrukci optimální permutace původní funkce zjevně nestačí.

Zajímavé je, že pro 4-zlomové funkce neumíme ukázat, že mají rekurzivně optimální permutace, ani neznáme protipříklad. Pokud však protipříklad existuje, víme jaké vlastnosti musí splňovat. Protipříkladem nemůže být positivní (ani negativní) funkce, protože ty mají vždy lichý počet zlomů. Dále buď f protipříklad a π jeho libovolná optimální permutace. Pak existují posloupnosti σ, τ a částečné ohodnocení $\psi : \sigma \to \{0, 1\}$ splňující $\pi = \sigma.\tau$, $\mathrm{Br}(f[\psi], \tau) = 3$ a $\mathrm{Br}^*(f[\psi]) = 1$. Tedy předběhneme-li s terminologií*, tak minimální (co do počtu proměnných) protipříklad má prázdný maximální blok a vůči optimální permutaci nemá zlom uprostřed.

*Viz sekce 4.2.

3. Reprezentace DNF

Cílem této kapitoly je vybudovat datové struktury, které budeme využívat v algoritmech uvedených dále. Pokud čtenáře nezajímá přesná složitost dále uvedených algoritmů, ale pouze jejich polynomialita (pro pevný počet zlomů), může tuto kapitolu přeskočit.

Tyto datové struktury jsou z velké části rozšířením standardních struktur užívaných při řešení Horn-SATu [7, 8, 9]. Protože pro implementaci těchto struktur nebudeme potřebovat výpočetní sílu RAMu, ale postačí nám Pointer Machine, uvedeme – s ohledem na neexistenci jedné všeobecně užívané definice* – nejprve krátký přehled výpočetních modelů.

Tímto mírně zesilujeme výsledky publikované v [2], protože tento článek implicitně předpokládá jako výpočetní model RAM, zatímco zde uvedené struktury ukazují, že PM je dostatečný. PM je dostatečně silný i pro algoritmus pro rozpoznání positivních 2-intervalových DNF publikovaný v [3], ale Rozšířená reprezentace by musela být ještě mírně poupravena.

3.1 Výpočetní modely

Dnes nejběžněji užívané výpočetní modely jsou Turingův stroj, Pointer Machine (dále PM) a Random Access Machine (dále RAM). Turingův stroj je na rozdíl od zbylých dvou velmi nepodobný dnešním počítačům, a tedy nepříliš užívaný pro studium rychlosti praktických algoritmů. Proto ho pomineme†.

Pointer Machine

Nejprve představíme PM a různé varianty tohoto modelu. Tyto varianty jsou výpočetně různě silné‡. Navíc všechny jsou slabší než RAM, který uvádíme především pro úplnost, jelikož je nejběžněji užívaným modelem. Začneme základní variantou PM. Ta je ze všech zde uvedených modelů nejslabší, ale stále je dostatečně silná pro všechny algoritmy uváděné v této práci.

Paměť PM se skládá z registrů a buněk. Buňky mohou vznikat v průběhu výpočtu a jejich počet není omezen. Všechny buňky mají stejnou strukturu§ a obsahují pevně daný počet písmen z konečné abecedy a pevně daný počet ukazatelů na jiné buňky. Registrů je opět pevně daný počet a obsahují písmena z konečné abecedy nebo ukazatele. Paměťová složitost algoritmu je počet užitých buněk.

*Zde uvedené definice výpočetních modelů vycházejí z přednášek Martina Mareše. Pro podrobnější informace vizte kapitolu 2 jeho disertace [10] či kapitolu 7 (v prvním vydání) skript z Grafových algoritmů [11].

†Pro úplnost poznamenejme, že libovolný Turingův stroj lze simulovat na Pointer Machine s pouze konstantním zpožděním – stačí pásky Turingova stoje nahradit spojovým seznamem.

‡Výpočetní silou myslíme, jak (asymptoticky) rychle jsou v daných modelech řešitelné problémy. Z hlediska vyčíslitelnosti jsou samozřejmě jsou všechny varianty PM i RAM stejně silné jako Turingův stroj.

§Z hlediska výpočetní síly ekvivalentní, formálně komplikovanější, ale pro praktické užití vhodnější je představovat si, že máme konečný počet různých typů buněk.

Instrukce PM jsou několika typů:

- „aritmetické“ instrukce, které dokáží provést libovolnou funkci nad písmeny v registrech a její výsledek opět uložit do registru,
- instrukce pro načtení a uložení písmene / ukazatele z registru do buňky jejíž adresa je uložena v registru,
- instrukce podmíněného (dle hodnot v písmenkových registrech či (ne)rovnosti ukazatelů v registrech) a nepodmíněného skoku,
- instrukce pro vytvoření nové buňky,
- a instrukce pro zastavení výpočtu.

Časová složitost je počet provedených instrukcí. Zbývá vyřešit vstup a výstup. Klasickým řešením je přidání vstupní a výstupní pásky nad konečnou abecedou. Toto kódování vstupu je „neefektivní“*, kupř. prohledání grafu do hloubky nedokážeme provést v čase† $\mathcal{O}(n+m)$, kde n je počet vrcholů a m počet hran. Proto budeme vstup a výstup reprezentovat také pomocí buněk, přičemž ukazatel na vstup a výstup bude předán pomocí určeného registru.

Ještě uveďme dvě možná rozšíření PM. První spočívá v přidání lineárního uspořádání na buňkách paměti – typicky dle času vytvoření buňky. Díky tomu můžeme čísla reprezentovat ukazatelem do spojového seznamu takového, že uspořádání na buňkách tohoto seznamu odpovídá uspořádání čísel, která tyto buňky reprezentují, a například zjistit, který ukazatel představuje větší číslo v konstantním čase. Druhé rozšíření, silnější‡ než přidání uspořádání, je povolení přirozených čísel délky W bitů, kde W je parametr a platí, že 2^W je větší než velikost vstupu i největší číslo ve vstupu se vyskytující, a běžných C-like§ operací nad nimi.

Random Access Machine

RAM je nejsilnější dnes užívaný výpočetní model. Zde popsaná verze se nazývá Word-RAM. Jeho paměť je tvořena polem buněk. Buňky jsou indexovány přirozenými čísly a každá obsahuje jedno číslo – typicky¶ přirozené o délce nejvýše W bitů, kde W je – obdobně jako u PM s čísly – parametr tak velký, aby byl možné adresovat celý vstup a pracovat se všemi čísly na vstupu.

*Problémem je, že výpočetní model dokáže pracovat s nekonstantním množstvím informace v konstantním čase (každý ukazatel obsahuje nekonstantní informaci) a pokud totéž neumožňuje i formát vstupu a výstupu, tak výpočetní model zbytečně „brzdí“.

†Pro zakódování grafu potřebujeme typicky více než $\mathcal{O}(m+n)$ bitů: Uvažme (orientovaný) graf s n vrcholy z nichž každý má výstupní stupeň 1. Takovýchto grafů existuje $(n-1)^n$, takže na jejich zakódování je potřeba $\Omega(\log((n-1)^n)) \approx \Omega(n \log n)$ bitů.

‡Omezíme-li se na algoritmy užívající $2^{\mathcal{O}(W)}$ paměti. Speciálně algoritmy běžící v polynomiálním čase toto splňují.

§Tedy aritmetické operace ($+$, $-$, $\cdot$, / a modulo), bitové posuny, logické operace (konjunkce, disjunkce a negace), bitové operace (and, or, xor a not po bitech) a porovnání.

¶Jinou možností je neomezovat velikost čísel na W bitů, ale vzdát se konstantní složitosti operací s nimi (což většinou vede ke zpomalení algoritmů o faktor $\log(n)$). Obdobně existují varianty modelu, které povolují jiné operace s čísly – místo C-like třeba AC^0 nebo průnik těchto dvou tříd.

Instrukce jsou tvořeny operacemi s čísly (C-like), podmíněným* skokem a instrukcí zastavení výpočtu. Všechny instrukce pracující s čísly mohou mít své operandy (vstupní i výstupní) určené třemi způsoby:

- konstantou – v kódu je uvedena hodnota daného operandu (nelze použít pro výstupní operand),
- přímá adresace – v kódu je uvedena adresa buňky, v níž je operand, nebo
- nepřímá adresace – v kódu je adresa buňky, v níž je adresa buňky s operandem.

Časová složitost je počet provedených instrukcí a prostorová je počet užitých buněk. Vstup a výstup jsou předávány na určeném místě paměti.

Dodejme, že takto definovaný RAM není (z hlediska vyčíslitelnosti) ekvivalentní Turingovu stroji, protože nedokáže pracovat s více než 2^W buňkami paměti, a tedy na něm není možné spouštět algoritmy vyžadující exponenciální paměť.

Toto omezení lze napravit kupř. povolením libovolně velkých čísel a časovou složitostí instrukce odpovídající délce jejich operandů†. Pro účely analýzy efektivních algoritmů toto nevadí, protože za efektivní typicky považujeme pouze algoritmy pracující v polynomiálním čase. Navíc variabilní časová složitost instrukcí komplikuje analýzu.

3.2 Reprezentace DNF v paměti PM

Datovou strukturu pro reprezentaci DNF budeme budovat na Pointer Machine. Pro větší přehlednost budeme při popisu struktur užívat čísla (typicky pro různá počítadla) a poté předvedeme, jak je možné se bez nich obejít (většinou nahrazením spojovým seznamem).

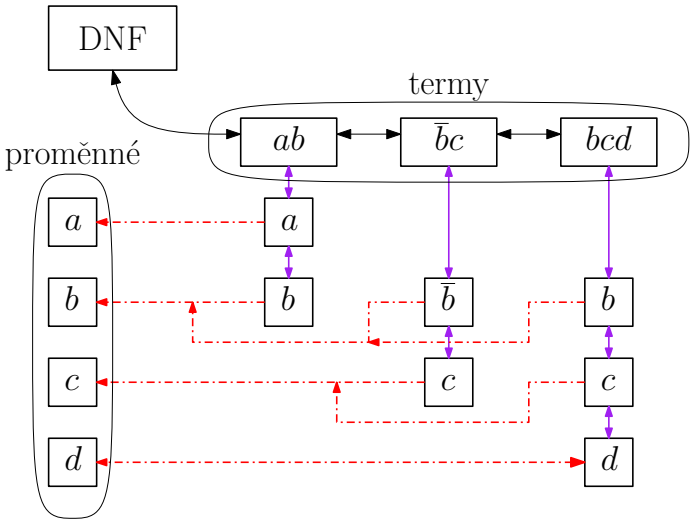
Naivní reprezentace

Přímočarou možností je reprezentovat DNF spojovým seznamem termů a každý term spojovým seznamem literálů. Každý literál je tvořen indikátorovou proměnnou, která určuje, zda je positivní či negativní, a reprezentací dané proměnné z DNF. Na RAMu jsou proměnné většinou reprezentované celými čísly, ale protože chceme užívání čísel omezit, budeme každou proměnnou reprezentovat buňkou paměti a každý literál bude obsahovat ukazatel na buňku reprezentující příslušnou proměnnou. Této reprezentaci budeme říkat **naivní reprezentace**, příklad viz obrázek 3.1.

Tato reprezentace je však pro některé operace příliš pomalá – speciálně postupné dosazování za všechny proměnné trvá s naivní reprezentací $\mathcal{O}(nl)$ místo

*Skok se provede, pokud hodnota operandu není 0. Speciální instrukci nepodmíněného nepotřebujme, protože podmíněný skok s konstantním operandem se chová jako nepodmíněný.

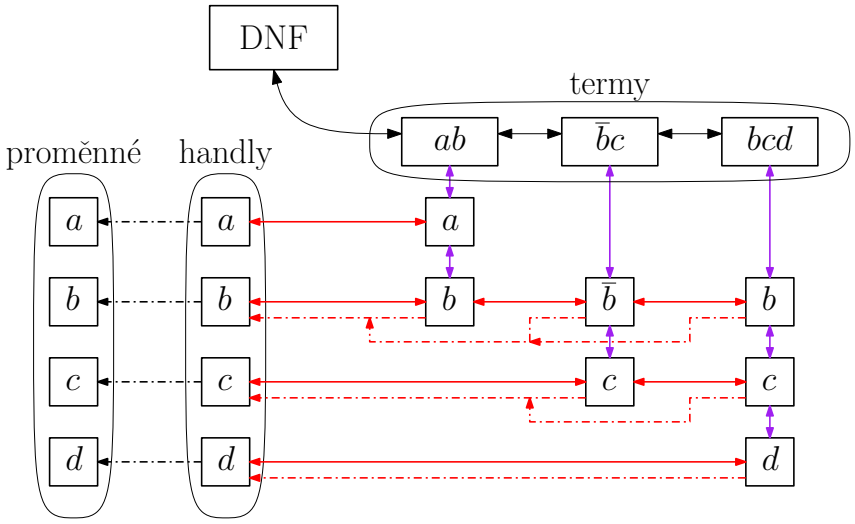
†Formálně je samozřejmě možné zavést operace s libovolně velkými čísly v čase $\mathcal{O}(1)$, ale takovýto model je pro praktické užití příliš silný.



Obrázek 3.1: Naivní reprezentace $ab \vee \bar{b}c \vee bcd$. Plné čáry značí obousměrný spojový seznam, čerchované pouze jednosměrné ukazatele.

optimálního $\mathcal{O}(n + l)$. Proto ke každé proměnné chceme přidat seznam literálů, v nichž se vyskytuje. Jednu proměnnou může obsahovat více DNF, a proto nemůžeme struktury představující proměnné modifikovat.

Tento problém vyřešíme „přejmenováním“ proměnných* – každá instance DNF si vytvoří vlastní kopie proměnných, které pro ní budou privátní a kterým budeme říkat handly. Do těchto kopií již můžeme přidávat potřebné informace a navíc si pamatují, z jakých proměnných vznikly. Takto upravenou reprezentaci vidíme na obrázku 3.2.



Obrázek 3.2: Naivní reprezentace $ab \vee \bar{b}c \vee bcd$ s handly a seznamem literálů

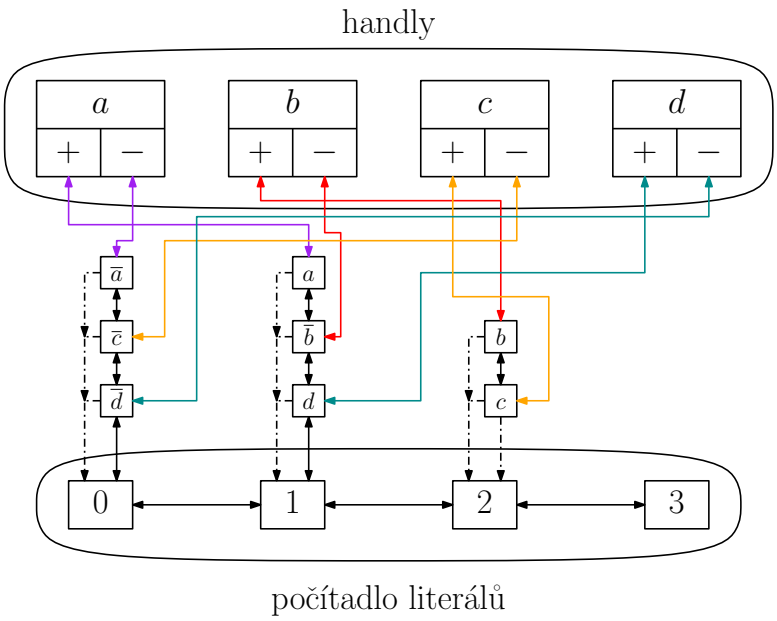
*Přejmenování můžeme provést, protože algoritmům, v nichž tyto struktury budeme užívat je jedno, zda pracují s původními proměnnými či nějak přejmenovanými. Pouze na závěr musíme přejmenovat proměnné zpět. To stihneme v čase $\mathcal{O}(1)$ na proměnou, a tedy tím nemůžeme zhoršit asymptotickou časovou složitost.

Rozšířená reprezentace

Kromě rychlého dosazování ještě budeme potřebovat najít všechny lineární termy a univerzální literály. Proto je budeme udržovat ve spojových seznamech – celkem budeme mít 4 – pro positivní / negativní univerzální literály / lineární termy. Abychom věděli, kdy do těchto seznamů potřebujeme přidat prvek, opatříme každou proměnnou počítadly kladných a negativních výskytů*. Ukazatele na tato počítadla budeme udržovat v seznamech setříděné dle jejich hodnoty. Navíc do každého literálu přidáme ukazatel na term do nějž náleží.

Protože nemáme čísla, musíme počítadla výskytů proměnných implementovat pomocí spojových seznamů. K tomuto účelu bude mít každá DNF alokován spojový seznam délky odpovídající počtu termů a počítadla budou ukazatele na prvky tohoto seznamu. Protože stejně potřebujeme zpětné ukazatele, každý prvek seznamu tvořícího počítadlo obsahuje spojový seznam všech proměnných, které na něj ukazují (spolu s příznakem zda jde o positivní či negativní počítadlo). Příklad vizte obrázek 3.3.

Dále budeme často zkoumat hodnotu DNF v bodech $\nvdash$ a $\nVdash$. V nich je DNF pravdivá, právě když obsahuje negativní resp. positivní term†. Proto přidáme počítadla positivních a negativních termů – každé z nich bude (jednosměrný) spojový seznam obsahující tolik prvků, kolik je hodnota příslušného počítadla. Teto reprezentaci budeme říkat **rozšířená reprezentace**.



Obrázek 3.3: Počítadla výskytu literálů pro $ab \vee \overline{b}c \vee bcd$

Pozorování 3.1: *O konstrukci Rozšířené reprezentace*
Rozšířenou reprezentaci zkonstruujeme z naivní reprezentace v čase $O(n + l)$.

*Do termů přidávat počítadlo literálů nemusíme, protože zjistit, zda spojový seznam má délku 1 lze v konstantním čase.

†Term je positivní, je-li složen pouze z positivních literálů. Analogicky pro negativní term.

Důkaz. Obtížná část je nahrazení ukazatelů na proměnné ukazateli na handly. K tomuto účelu si musíme poznamenat ukazatel na handle do každé proměnné (a z vlastností modelu struktura popisující proměnnou vždy obsahuje místo na alespoň jeden ukazatel). Toto můžeme provést, pouze po přeznačení musíme vrátit proměnné do původního stavu. Konstrukci všech počítadel pak provedeme jedním projitím DNF. ♡

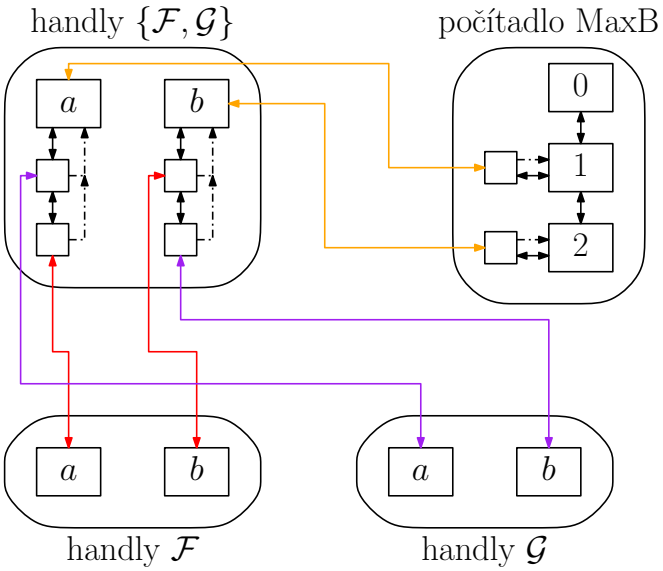
> **Pozorování 3.2:** *O částečném dosazování do Rozšířené reprezentace*
> Dosazení do rozšířené reprezentace za handle x trvá $\mathcal{O}(1 + \text{počet výskytů } x$ v dané DNF).

Důkaz. Zjevné. Stačí projít seznam výskytů x a odebrat x nebo příslušný term, což stihneme v čase $\mathcal{O}(1)$ na výskyt. ♡

Reprezentace množiny DNF

V kapitole 6 budeme pracovat s množinami DNF nad stejnými proměnnými, proto na závěr této kapitoly popíšeme reprezentaci množin DNF. Prvky množiny budeme mít uložené pomocí rozšířené reprezentace. Navíc množina bude mít vlastní handly (tj. privátní kopie proměnných), které obsahují seznam jim odpovídajících handlů DNF z dané množiny. Díky tomu budeme moci stále rychle provádět částečná dosazení.

Navíc budeme potřebovat seznam obsahující všechny proměnné (handly) maximálního bloku*. To zařídíme pomocí spojového seznamu sloužícího jako počítadlo spolu se zpětnými ukazateli, stejně jako při udržování seznamu univerzálních literálů v rámci rozšířené reprezentace. Příklad vizte obrázek 3.4.



Obrázek 3.4: Množina DNF $\{\mathcal{F}, \mathcal{G}\}$. Provázání handlů množiny s handly jednotlivých DNF v ní a počítadlo výskytů v maximálním bloku.

*Viz definice 4.5.

4. Výpočet množiny zlomů

Na rozdíl od algoritmů v článcích [2] a [3] bude náš algoritmus pro rozpoznávání k-zlomových boolovských funkcí vracet pouze optimální permutaci (a implicitně její počet zlomů) a nikoli i zlomy samotné*. Pro úplnost tedy nejprve uvedeme algoritmus, který k zadané boolovské funkci a její permutaci vypočte příslušnou množinu zlomů.

Jelikož chceme polynomiální algoritmus a rozhodnout, zda existuje nějaký zlom pro zadanou DNF $\mathcal{F}$ a permutaci je NP-úplné†, budeme dále předpokládat, že vstupní DNF náleží do pevně dané řešitelné třídy. Také velikost výstupu může být exponenciální vzhledem k velikosti vstupu‡, takže ji budeme muset zahrnout do odhadu složitosti.

4.1 Rozpoznávání triviálních podfunkcí

Algoritmus pro výpočet množiny zlomů bude rekurzivně prohledávat rozhodovací strom příslušný zkoumané boolovské funkci (zadaná permutace určuje pořadí rozhodování). Samozřejmě nechceme tento strom prohledávat celý (což má časovou složitost řádově 2^n), ale pouze větve, které vedou k nějakým zlomům. Potřebujeme tedy umět rozpoznat konstantní podstromy.

Máme-li (nekonstantní) funkci f a proměnnou $x \in \mathrm{Var}(f)$, za kterou chceme dosadit, existují čtyři případy, které potřebujeme rozpoznávat – funkce f může být konstantní v levé či pravé polovině§ a může zde nabývat hodnoty 0 či 1. Dále v našem zkoumání budeme uvažovat pouze boolovské funkce zadané pomocí DNF, ale všechny výsledky by bylo možné analogicky formulovat i pro CNF. První pozorování říká, že funkce f je na své levé (resp. pravé) polovině vůči proměnné x identicky rovna nule právě tehdy, je-li x (resp. $\overline{x}$) univerzální literál v každé její DNF.

Pozorování 4.1: *O univerzálním literálu*
Buď $\mathcal{F}$ DNF. Pak:

$$\mathcal{F}[x := 0] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(x \in t)$$

$$\mathcal{F}[x := 1] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(\overline{x} \in t)$$

*Všechny prezentované algoritmy je možné upravit, aby kromě optimální permutace počítaly i jí příslušnou množinu zlomů. Zbytečně by to ale komplikovalo jak algoritmy samotné, tak jejich analýzu, a asymptotickou složitost oproti spuštění Algoritmu 4.1 na výslednou optimální permutaci by to nijak nezlepšilo. Proto budeme tyto problémy řešit odděleně.

†Problém je ve třídě NP, protože otestovat, je-li x je zlom, je triviální. Naopak bez ohledu na zadanou permutaci je odpověď ANO právě tehdy, pokud $\mathcal{F}$ není konstantní, čímž máme převod falsifikovatelnosti na tento problém.

‡Uvažme funkci $f(x, y_1, \ldots, y_n) = x$ a permutaci $\pi = y_1 \ldots y_n x$. Zde pro každé $x \in [1, 2^{n+1} - 1]$ platí $f_\pi(x) \neq f_\pi(x-1)$, a tedy $\mathrm{Br}(f, \pi) = 2^{n+1} - 1$.

§Levou polovinou funkce f vůči proměnné x se rozumí funkce $f[x := 0]$ (a analogicky pro pravou polovinu).

Důkaz. Implikace zprava doleva jsou zřejmé. Opačné implikace. Ukážeme případ $x := 0$, $x := 1$ je analogický. Sporem. Mějme $t \in \mathcal{F}$ takové, že $x \notin t$. Z předpokladů lze t splnit nějakým ohodnocením ψ, a protože $x \notin t$, tak ψ' vzniklá ze ψ přiřazením $x := 0$, stále splňuje t, tedy $\mathcal{F}(\psi') = 1$. Spor s $\mathcal{F}[x := 0] \equiv 0$. ♡

Pro úplnost následuje triviální pozorování, že obsahuje-li $\mathcal{F}$ positivní lineární term x, tak je $\mathcal{F}[x := 1] \equiv 1$ (a analogicky pro negativní lineární term), i když více nás bude zajímat opačná implikace.

Pozorování 4.2: *Triviální o lineárním termu*
Mějme DNF $\mathcal{F}$. Pak:

$$\{x\} \in \mathcal{F} \Rightarrow \mathcal{F}[x := 1] \equiv 1$$

$$\{\overline{x}\} \in \mathcal{F} \Rightarrow \mathcal{F}[x := 0] \equiv 1$$

Důkaz. Zřejmý. ♡

Kýžená opačná implikace obecně neplatí, protipříklad budiž $\mathcal{F} = xy \vee x\overline{y} \vee z$, kde $\mathcal{F}[x := 1] \equiv 1$. Je-li funkce na své pravé* polovině vůči proměnné x konstantně 1, je x její implikant. Navíc je-li $\mathcal{F}$ nekonstatní, tak prázdný term není jejím implikantem a x je tedy primární implikant. To svádí k použití úplné primární DNF. Ta ale může být – oproti jiným DNF reprezentujícím tutéž funkci – exponenciálně dlouhá. Naštěstí je x nejen primární, ale dokonce esenciální implikant, takže se vyskytuje v každé DNF reprezentující danou funkci složené z primárních implikantů.

Pozorování 4.3: *O lineárním termu*
Buď $\mathcal{F}$ netriviální esenciální DNF. Pak:

$$\mathcal{F}[x := 1] \equiv 1 \Rightarrow \{x\} \in \mathcal{F}$$

$$\mathcal{F}[x := 0] \equiv 1 \Rightarrow \{\overline{x}\} \in \mathcal{F}$$

Důkaz. Zjevně je x (resp. $\overline{x}$ v druhé implikaci) implikantem $\mathcal{F}$. Dle pozorování 1.16 je esenciální. ♡

Důsledek 4.4: *O esenciální DNF*
Mějme netriviální esenciální DNF $\mathcal{F}$ a $x \in \mathrm{Var}(\mathcal{F})$. Pak:

- $\mathcal{F}[x := 0] \equiv 1 \Leftrightarrow \{\overline{x}\} \in \mathcal{F}$
- $\mathcal{F}[x := 1] \equiv 1 \Leftrightarrow \{x\} \in \mathcal{F}$
- $\mathcal{F}[x := 0] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(x \in t)$
- $\mathcal{F}[x := 1] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(\overline{x} \in t)$

*Analogicky pro levou polovinu a $\overline{x}$.

4.2 Maximální blok

Důsledek 4.4 shrnuje všechny zajímavé případy, které chceme v esenciální DNF rozpoznávat. Jelikož je nepraktické je všechny vždy znovu vyjmenovávat, shrneme je do pojmu maximálního bloku. Blok je podmnožina proměnných boolovské funkce taková, že za každou proměnnou z bloku lze dosadit tak, aby výsledkem byla konstantní funkce.

Definice 4.5: *Maximální blok*
Mějme boolovskou funkci f. **Blok proměnných*** je množina $B \subseteq \mathrm{Var}(f)$ taková, že

$$(\forall x \in B)(\exists a, b \in \{0,1\})(f[x := a] \equiv b)$$

Blok je maximální, pokud je maximální vzhledem k inkluzi. Maximální blok funkce f budeme značit $\mathrm{MaxB}(f)$.

Přímo z definice plyne, že sjednocení bloků je opět blok, a tedy maximální blok je jednoznačně určen. Máme-li funkci f reprezentovanou netriviální esenciální DNF $\mathcal{F}$ tak – dle důsledku 4.4 – proměnná x náleží do maximálního bloku právě tehdy, pokud buď $\mathcal{F}$ obsahuje lineární term x či $\overline{x}$ nebo existuje univerzální literál, který ji obsahuje (tj. x či $\overline{x}$).

Důležitost pojmu blok se plně ukáže až sekci 6.1, kde s jeho pomocí vytvoříme efektivní algoritmus pro rozpoznávání k-zlomových funkcí. Již zde ho definujeme pouze kvůli následujícímu pozorování, které nám umožní provést lepší horní odhad časové složitosti algoritmu 4.1. Pozorování říká, že máme-li esenciální DNF a dosadíme do ní za proměnnou z bloku, je výsledkem opět esenciální DNF.

Pozorování 4.6: *O dosazovaní za proměnnou bloku*
Buď $\mathcal{F}$ esenciální DNF, $x \in \mathrm{MaxB}(\mathcal{F})$. Pak $\mathcal{F}[x := a]$ je esenciální DNF $(\forall a)$.

Důkaz. Pokud je $\mathcal{F}[x := a]$ triviální, je zjevně i esenciální. Mějme $\mathcal{F}[x := a]$ (a tedy i $\mathcal{F}$) netriviální. Rozlišme případy, kdy je x (nebo $\overline{x}$) lineární term, a kdy je součástí univerzálního literálu.

Nejprve případ, kdy x je součástí univerzálního literálu. BÚNO je tento literál x (a ne $\overline{x}$) a $a = 1$. Pak z libovolné primární DNF reprezentující $\mathcal{F}$ vyrobíme primární DNF reprezentující $\mathcal{F}[x := 1]$ vypuštěním x ze všech termů† a naopak. Z toho plyne, že esenciální implikanty $\mathcal{F}[x := 1]$ jsou právě esenciální implikanty $\mathcal{F}$ po vypuštění literálu x. Proto i DNF $\mathcal{F}[x := 1]$ je esenciální.

Zbývá případ, kdy x tvoří lineární term, BÚNO x (a ne $\overline{x}$) a $a = 0$. Jelikož je x implikant $\mathcal{F}$, tak žádný jiný primární implikant $\mathcal{F}$ nemůže obsahovat x (protože by ho x absorbovalo) ani $\overline{x}$ (konsenzem s x bychom získali kratší implikant, který by ten původní absorboval). Z toho plyne, že mezi primárními DNF reprezentujícími $\mathcal{F}$ a $\mathcal{F}[x := 0]$ lze přecházet vypuštěním (resp. přidáním) termu x. Tedy esenciální implikanty $\mathcal{F}[x := 0]$ jsou právě esenciální implikanty $\mathcal{F}$ vyjma x. Tedy $\mathcal{F}[x := 0]$ je esenciální DNF. ♡

*Slovo „proměnných“ budeme dále vynechávat.

†Díky pozorování 4.1 víme, že x je univerzální literál v každé DNF reprezentující tutéž funkci jako $\mathcal{F}$.

4.3 Algoritmus pro výpočet množiny zlomů

Jak bylo zmíněno výše algoritmus pro výpočet množiny zlomů bude rekurzivně prohledávat rozhodovací strom funkce f. Ve chvíli, kdy narazí na konstantní podstrom, vrátí prázdnou množinu zlomů* (řádek 2).

Není-li funkce konstantní, označí x další proměnnou v pořadí a σ zbytek permutace. Zkontroluje, zda je mezi podstromy určenými proměnnou x zlom, a pokud ano, přidá ho do pomocné množiny M (řádek 6). Rekurzivně spočte L a R – množiny zlomů v levém a pravém podstromu – a vrátí jejich sjednocení spolu s M, pouze R předtím musí posunout o velikost levého podstromu (tj. $2^{|\sigma|}$).

> **Značení:** Buď $X \subset \mathbb{Z}$ a $a \in \mathbb{Z}$, pak $X + a := \{x + a : \ x \in X\}$.

```
1  Function BrSet(F, π)
       Input: DNF F, permutace π ∈ S_Var(F)
       Output: X množina zlomů F vůči π
2      if Br(F, π) = 0 then return ∅
3      M ← ∅
4      x ← π[1]
5      σ ← π[2..]
6      if F[x := 0](⊮) ≠ F[x := 1](⊩) then M ← {2^|σ|}
7      L ← BrSet(F[x := 0], σ)
8      R ← BrSet(F[x := 1], σ)
9      return M ∪ L ∪ (R + 2^|σ|)
10 end
```

Algoritmus 4.1: Výpočet množiny zlomů

Korektnost

S každým rekurzivním voláním délka permutace o 1 klesne. Tedy rekurze má hloubku nejvýše n a algoritmus zastaví†.

Zbývá ukázat, že $\texttt{BrSet}(\mathcal{F}, \pi)$ skutečně vrátí množinu zlomů funkce f vůči permutaci π. Budeme postupovat indukcí dle délky permutace π. Případ $|\pi| = 0$ je triviální. Indukční krok: Označme $\pi = x.\sigma$. Zlomy f vůči π můžeme rozdělit na zlomy v levé části (tj. $\mathcal{F}[x := 0]$), v pravé části ($\mathcal{F}[x := 1]$) a případný zlom uprostřed‡. Algoritmus tedy rekurzivně spustíme na levou a pravou část (zde nám z indukce vrátí správné výsledky) a vrátíme sjednocení výsledků levé části, zlomu uprostřed, který je na pozici $2^{|\sigma|}$, a pravé části posunuté o $2^{|\sigma|}$.

*Speciálně každý list stromu tvoří konstantní podstrom, takže rekurze zastaví.

†Podmínka $\mathrm{Br}(\mathcal{F}, \pi) = 0$ je triviálně splněna v případě, že $\mathcal{F}$ je konstanta – tj. $\mathrm{Var}(f) = \emptyset$.

‡Ač byl zlom formálně zaveden jako hodnota argumentu, před níž se mění funkční hodnota, zde je opět přirozenější dívat se na něj jako na místo mezi.

Výpočetní model a reprezentace dat

Složitost algoritmu budeme počítat pro implementaci na Pointer machine, přičemž nám bude stačit nejslabší verze PM bez čísel a uspořádání na ukazatelích. Za povšimnutí stojí, že ač by nám v obecné analýze použití silnější verze PM nebo RAMu nijak nepomohlo, při analýze implementace pro konkrétní řešitelnou třídu boolovských funkcí může být jejich použití vhodné, neboť složitost funkce pro převod DNF do tvaru obsahujícího esenciální implikanty se v těchto modelech může lišit.

Vstupní DNF budeme mít zadanou pomocí naivní reprezentace popsané v předchozí kapitole. Abychom docílili polynomiální časové složitosti budeme navíc předpokládat, že vstupní DNF patří do pevně určené řešitelné třídy a každou funkci z této třídy umíme transformovat na tvar obsahující všechny esenciální implikanty* v čase $T(\|\mathcal{F}\|)$, kde $\|\mathcal{F}\|$ je počet literálů transformované formule.

Množina zlomů bude opět (neuspořádaný) spojový seznam a každý zlom – jsa $|\pi|$-bitové číslo – spojový seznam $|\pi|$ bitů uspořádaný od nejvýznamnějšího bitu. Permutace π je spojový seznam ukazatelů na proměnné.

Složitost – jednoduchá analýza

Analýzu složitosti rozdělíme na dvě části. V první předvedeme přímočarou analýzu, která povede na složitost $\mathcal{O}(|X|\, n(l + T(l)))$. V druhé části uvedeme jemnější analýzu, která povede na složitost $\mathcal{O}(|X|\, (n + l + T(l)))$, což speciálně pro positivní DNF a konstantní počet zlomů dává lineární algoritmus.

Označme n počet proměnných a l počet literálů $\mathcal{F}$. Jedno volání funkce `BrSet` bez rekurze a posouvání množiny R stihneme v čase $T(l) + l$. Posouvání budeme účtovat jednotlivým prvkům množiny R a všimneme si, že každý prvek může být posunut nejvýše n-krát†. Posunutí jednoho prvku nás stojí $\mathcal{O}(1)$‡, a tedy všechno posouvání dohromady nejvýše $\mathcal{O}(|X|\, n)$.

Zbývá spočítat počet volání funkce `BrSet`. Strom rekurze je binární a každý vrchol, jehož oba synové jsou listy, přispívá do množiny X jedním zlomem§, takže těchto vrcholů je nejvýše $|X|$. Navíc (neostře) pod každým vrcholem stromu rekurze, který není list, je takovýto vrchol, takže všech vrcholů ve stromu rekurze je $\mathcal{O}(n\,|X|)$¶.

Celková časová složitost algoritmu 4.1 je $\mathcal{O}(n\,|X| + n\,|X|\,(l + T(l))) = \mathcal{O}(n\,|X|\,(l + T(l)))$.

*Převod na primární iredundantní formu je příkladem takové transformace, ale občas se může hodit, že na požadovaný tvar nejsme tak přísní – kupř. je-li vstupem positivní DNF, tak z ní získáme primární iredundantní tvar absorbcemi, ale vždy obsahuje všechny své esenciální implikanty, takže absorbce dělat nemusíme.

†Protože n je maximální hloubka rekurze.

‡Číslo je realizováno jako spojový seznam číslic svého binárního zápisu, přičítáme vždy mocniny 2, a to postupně od nejméně významného bitu.

§Tento vrchol provedl rekurzi, takže funkce na jeho vstupu nebyla konstantní, ale vstupy obou jeho synů byly konstantní.

¶Ač tento odhad vypadá celkem hrubě, je pro $|X| = 2^{o(n)}$ asymptoticky těsný. Stačí uvážit výpočet, který větví než, dosáhne $|X|$ větví (na což dle předpokladu stačí $o(n)$ hladin), a pak všechny větve pokračují bez větvení až na n-tou hladinu.

Složitost – jemnější analýza

Jediné, co nám v předcházející analýze zabraňuje dosáhnout lepší složitosti $\mathcal{O}(|X|\,(n + l + T(l)))$, je čas $\mathcal{O}(l + T(l))$ na jedno volání funkce `BrSet`. K jeho zlepšení musíme změnit vnitřní reprezentaci DNF z naivní na rozšířenou*. Nejprve se zbavíme části $T(l)$.

Dle pozorování 4.6 je DNF po dosazení proměnné z bloku opět esenciální, takže na ni nemusíme volat proceduru pro převod do esenciálního tvaru. Zbývá odhadnou počet dosazení za proměnné, které nenáleží do maximálního bloku. Dosazeními za takovou proměnnou vždy získáme dvě nekonstantní boolovské funkce, a tedy jich nemůže být více než celkový počet zlomů (tj. $|X|$). Celkový čas strávený převodem DNF do esenciálního tvaru je $\mathcal{O}(|X|\,T(l))$.

Zbývá spočítat ostatní práci v rámci volání funkce `BrSet`. Test, zda je $\mathcal{F}$ konstantní, zvládneme s esenciální DNF za $\mathcal{O}(1)$. Taktéž v konstantním čase zvládneme inicializaci proměnných M, x a σ a sjednocení množin při returnu. Zbývá otestování zlomu uprostřed, případná výroba číselné reprezentace tohoto zlomu a dosazení do $\mathcal{F}$ za x. Výrobu reprezentace můžeme účtovat vytvořenému zlomu, což dává $\mathcal{O}(|X|\,n)$ za celý běh algoritmu.

Test $g(\not\vdash) = 1$ odpovídá tomu, je-li v DNF g positivní term (a obdobně pro $g(\not\vdash) = 1$), což lze z rozšířené reprezentace zjistit v $\mathcal{O}(1)$. Pokud tedy do $\mathcal{F}$ nejprve dosadíme, máme test na řádku 6 v konstantním čase. Zbývá vyřešit pouze dosazování za x. Zde rozlišíme, zda $x \in \mathrm{MaxB}(\mathcal{F})$ či nikoli. Jak bylo řečeno výše, dosazení $x \notin \mathrm{MaxB}(\mathcal{F})$ není více než $|X|$, takže na každé z nich můžeme obětovat lineární čas.

Pokud x náleží do maximálního bloku, tak jeden z výsledků dosazení je konstantní funkce. Který to je, zjistíme v $\mathcal{O}(1)$ z rozšířené reprezentace, takže toto dosazení nemusíme provádět a jen vrátíme příslušnou konstantní funkci. Druhé dosazení provedeme, ale těchto dosazení do jedné DNF uděláme nejvýše n a dle vlastností rozšířené reprezentace všech n dosazení dohromady trvá $\mathcal{O}(n+l)$. Dosazováním tedy strávíme $\mathcal{O}(|X|\,(n + l))$.

Celkem trávíme čas $\mathcal{O}(|X|\,n)$ posouváním množin R, $\mathcal{O}(|X|\,T(l))$ převody do esenciálního tvaru a $\mathcal{O}(|X|\,(n + l))$ ostatní prací.

Důsledek 4.7: *O algoritmu pro výpočet množiny zlomů (4.1)*
Algoritmus 4.1 je korektní a jeho časová složitost je† $\mathcal{O}(|X|\,(n + l + T(l)))$.

*Vstupní DNF pořád zůstává v naivním tvaru, ale to nevadí. Na počátku algoritmu můžeme obětovat čas $\mathcal{O}(n + l)$ na její převod. Na druhou stranu při rekurzivních voláních chceme vždy předávat rozšířenou reprezentaci. Navíc si u ní chceme pamatovat, zda je esenciální, abychom zbytečně nevolali transformační proceduru.

†Povšimněme si, že dolní odhad na složitost je $\Omega(|X|\,n)$, protože tolik je velikost výstupu.

5. Optimální permutace množin funkcí

V kapitole 2 jsme nahlédli, že není zřejmé, jak efektivně optimální permutace hledat. Hlavním problémem byla „paralelní rekurze“, kdy mezi sebou jednotlivá rekurzivní volání musí komunikovat, aby našla stejnou permutaci. Tomuto problému se pokusíme vyhnout zobecněním zkoumané úlohy.

Místo permutace optimální pro jednu boolovskou funkci budeme hledat permutaci optimální pro množinu (vlastně multimnožinu, protože opakování je důležité) boolovských funkcí nad stejnou množinou proměnných. Díky tomu se při rekurzi množina funkcí sice zvětší, ale stále to bude jedna množina, a tedy i jedno rekurzivní volání.

Způsobů jak zavést optimální permutace nad množinami funkcí je jistě mnoho, ale pro naše účely je vhodná součtová metrika – tj. počet zlomů množiny je součet počtu zlomů jednotlivých funkcí (zde vidíme, proč musíme pracovat s multimnožinami) a optimální permutace jsou ty, které tuto hodnotu minimalizují.

Definice 5.1: *Optimální permutace množiny funkcí*
Buď X (multi)množina* boolovských funkcí na množině proměnných $\mathrm{Var}(X)$. Pak počet zlomů množiny X vůči permutaci π je

$$\mathrm{Br}(X, \pi) := \sum_{f \in X} \mathrm{Br}(f, \pi)$$

a π je optimální permutací X, pokud $\mathrm{Br}(X, \pi) \leq \mathrm{Br}(X, \varphi)\ \forall \varphi \in \mathcal{S}_{\mathrm{Var}(X)}$.

5.1 Naivní algoritmus

Výhodou práce s množinami funkcí je, že všechny optimální permutace jsou v jistém smyslu rekurzivně optimální – začneme-li s množinou $\{f\}$, která má optimální permutaci $x.y.\pi$, tak $y.\pi$ je optimální† pro množinu $\{f[x := 0], f[x := 1]\}$, π je optimální pro $\{f[x := 0, y := 0], f[x := 0, y := 1], f[x := 1, y := 0], f[x := 1, y := 1]\}$ atd. Nejprve formalizujme uvedené množiny a uveďme jejich základní vlastnosti.

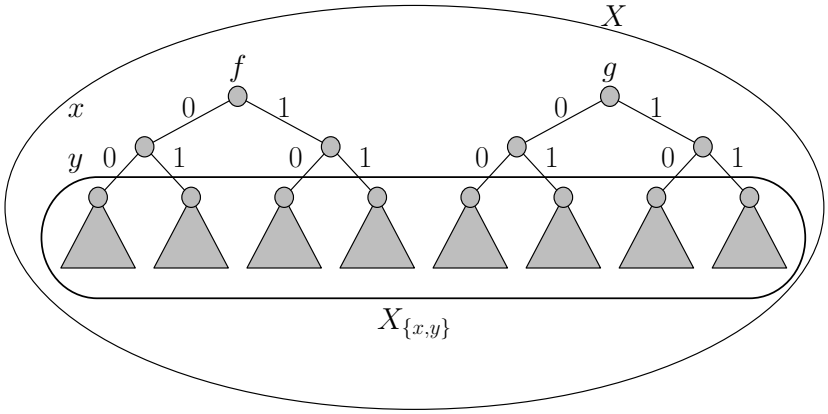
Definice 5.2: *Množina podfunkcí*
Buď X množina boolovských funkcí na proměnných $\mathrm{Var}(X)$ a $V \subseteq \mathrm{Var}(X)$. Pak definujeme množinu podfunkcí množiny X dle V:

$$X_V := \{f[\psi] \mid f \in X,\ \psi \in [V \to \{0, 1\}]\}$$

*Předponu multi- budeme dále vynechávat.

†Optimalita pro množinu $\{f[x := 0], f[x := 1]\}$ ovšem neznamená, že permutace $y.\pi$ je optimální pro kterýkoli z jejích prvků.

Tedy množina X_V obsahuje všechny funkce $f[\psi]$, kde $f \in X$ a ψ je ohodnocení proměnných z V, spolu se všemi násobnostmi. Graficky si můžeme X představovat jako množinu rozhodovacích stromů, v nichž se rozhodujeme nejprve dle proměnných z V a pak dle ostatních. Potom je X_V množina všech podstromů vzniklých otrháním X na $|V|$-té úrovni – vizte obrázek 5.1.



Obrázek 5.1: Znázornění množiny podfunkcí $X_{\{x,y\}}$ pro $X = \{f, g\}$

Uveďme ještě několik zjevných a užitečných vlastností množin podfunkcí:

- $X_\emptyset = X$
- $X_{\{x\}} = \biguplus_{f \in X} \{f[x := 0], f[x := 1]\}$
- Pro A, B disjunktní: $X_{A \cup B} = (X_A)_B$

Snadno nahlédneme, že pro každou boolovskou funkci f platí $\mathrm{Br}(f, x.\sigma) = \mathrm{Br}(f[x := 0], \sigma) + \mathrm{Br}(f[x := 1], \sigma) + [\![f[x := 0](\not\models) \neq f[x := 1](\not\models)]\!]$. Méně formálně, počet zlomů f je počet zlomů v levé plus v pravé polovině plus případný zlom uprostřed. Toto pozorování můžeme zobecnit na množinu boolovských funkcí.

Pozorování 5.3: *O počtu zlomů*
Buď X množina boolovských funkcí a $x.\sigma \in \mathcal{S}_{\mathrm{Var}(X)}$:

$$\mathrm{Br}(X, x.\sigma) = \mathrm{Br}(X_{\{x\}}, \sigma) + \sum_{f \in X} [\![f[x := 0](\not\models) \neq f[x := 1](\not\models)]\!]$$

Důkaz. Zjevný. ♡

Jelikož rozdíl $\mathrm{Br}(X, x.\sigma) - \mathrm{Br}(X_{\{x\}}, \sigma)$ nezávisí na σ, můžeme zavést následující definici:

Definice 5.4: *Počet zlomů vůči proměnné*
Počet zlomů množiny X vůči proměnné x značíme $\mathrm{Br}_{\{x\}}(X)$ a je definován vztahem:

$$\mathrm{Br}_{\{x\}}(X) := \mathrm{Br}(X, x.\sigma) - \mathrm{Br}(X_{\{x\}}, \sigma)$$

Dle pozorování 5.3 platí $\mathrm{Br}_{\{x\}}(X) = \sum_{f \in X} [\![f[x := 0](\not\models) \neq f[x := 1](\not\models)]\!]$. Tedy počet zlomů funkce f vůči proměnné x je jedna či nula podle toho, zda je či není mezi levou a pravou polovinou funkce f zlom (přičemž proměnná x

určuje, co je levá a pravá polovina). Aplikujeme-li uvedenou rovnost rekurzivně, získáme následující pozorování. Dobrou intuicí je, představit si boolovskou funkci dle permutace jako rozhodovací strom a pozorování říká, že každý zlom se nachází mezi nějakými stejně velkými sousedními podstromy.

Pozorování 5.5: *O počtu zlomů mezi podstromy*

$$\mathrm{Br}(X, \pi) = \sum_{i=1}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]})$$

Důkaz. Indukcí dle počtu proměnných. Pro $|\pi| = 0$ triviální. Indukční krok. Mějme $\pi = x.\sigma$. Z indukce plyne (druhá rovnost je pouhé přeznačení):

$$\begin{aligned}\mathrm{Br}(X_{\{x\}}, \sigma) &= \sum_{i=1}^{|\sigma|} \mathrm{Br}_{\{\sigma[i]\}}((X_{\{x\}})_{\sigma[1\,..\,i-1]}) \\ &= \sum_{i=2}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]})\end{aligned}$$

Užijeme pozorování 5.3:

$$\begin{aligned}\mathrm{Br}(X, \pi) &= \mathrm{Br}_{\{x\}}(X) + \mathrm{Br}(X_{\{x\}}, \sigma) \\ &= \mathrm{Br}_{\{x\}}(X) + \sum_{i=2}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]}) \\ &= \mathrm{Br}_{\{\pi[1]\}}(X_\emptyset) + \sum_{i=2}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]}) \\ &= \sum_{i=1}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]}) \qquad \heartsuit\end{aligned}$$

Z pozorování 5.3 vytěžíme ještě jeden výsledek, který je do značné míry protipólem zlomu funkce vůči proměnné. Definice zlomu funkce vůči proměnné a pozorování 5.3 hovoří o počtu zlomů daných prefixem permutace a jejich nezávislosti na jejím zbytku. Naopak následující pozorování říká, že rozdíl počtu zlomů vůči dvěma permutacím nezávisí na jejich společném prefixu. To nám bude velmi užitečné při důkazu korektnosti algoritmu pro výpočet optimální permutace představenému dále v této kapitole.

Pozorování 5.6: *O rozdílu počtu zlomů*
Mějme množinu boolovských funkcí X a π, ρ, σ takové, že $\pi.\rho, \pi.\sigma \in \mathcal{S}_{\mathrm{Var}(X)}$. Pak:

$$\mathrm{Br}(X, \pi.\rho) - \mathrm{Br}(X, \pi.\sigma) = \mathrm{Br}(X_\pi, \rho) - \mathrm{Br}(X_\pi, \sigma)$$

Důkaz. Indukcí dle délky společného prefixu π. Pro $|\pi| = 0$ triviální. Označme $x = \pi[1]$ a $\pi' = \pi[2\,..\,]$. Pro $|\pi| = 1$ dle pozorování 5.3:

$$\begin{aligned}\mathrm{Br}(X, x.\sigma) - \mathrm{Br}(X, x.\rho) &= \mathrm{Br}(X_{\{x\}}, \sigma) + \mathrm{Br}_{\{x\}}(X) - (\mathrm{Br}(X_{\{x\}}, \rho) + \mathrm{Br}_{\{x\}}(X)) \\ &= \mathrm{Br}(X_{\{x\}}, \sigma) - \mathrm{Br}(X_{\{x\}}, \rho)\end{aligned}$$

Indukční krok:

$$\begin{aligned}\mathrm{Br}(X, \pi.\rho) - \mathrm{Br}(X, \pi.\sigma) &= \mathrm{Br}(X, x.\pi'.\rho) - \mathrm{Br}(X, x.\pi'.\sigma) \\ &= \mathrm{Br}(X_{\{x\}}, \pi'.\rho) - \mathrm{Br}(X_{\{x\}}, \pi'.\sigma) \\ &= \mathrm{Br}((X_{\{x\}})_{\pi'}, \rho) - \mathrm{Br}((X_{\{x\}})_{\pi'}, \sigma) \\ &= \mathrm{Br}(X_\pi, \rho) - \mathrm{Br}(X_\pi, \sigma)\end{aligned}$$

Kde první a poslední rovnost jsou triviální a prostřední plynou z indukce. ♡

Nyní máme dostatek prostředků pro formulaci a dokázání výše zmíněné rekurzivní vlastnosti optimálních permutací.

Pozorování 5.7: *O rekurzivní optimalitě permutace množiny funkcí*
Mějme množinu boolovských funkcí X a její optimální permutaci $\pi = \rho.\sigma$. Pak σ je optimální permutací pro X_ρ.

Důkaz. Sporem. Mějme σ' takové, že $\mathrm{Br}(X_\rho, \sigma) > \mathrm{Br}(X_\rho, \sigma')$. Pak $\mathrm{Br}(X, \pi) > \mathrm{Br}(X, \rho.\sigma')$ dle pozorování 5.6. Spor. ♡

Povšimněme si kontrastu s rekurzivně optimálními permutacemi – definici rekurzivně optimální permutace lze přeformulovat tak, že π je rekurzivně optimální pro f pokud:

$$(\forall \sigma, \tau)((\pi = \sigma.\tau) \Rightarrow (\forall f' \in \{f\}_\sigma)(\tau \in \mathrm{Perm}^*(f')))$$

Tedy zatímco pro „běžnou“ optimální permutaci platí globální vlastnost – τ je optimální pro $\{f\}_\sigma$, tak rekurzivně optimální permutace je optimální lokálně tj. τ je optimální pro každý prvek $\{f\}_\sigma$.

S pomocí předešlého pozorovaní můžeme sestrojit algoritmus 5.2 pro výpočet optimální permutace. Tento algoritmus je velmi jednoduchý, ale extrémně pomalý (řádově n^n), proto ukážeme pouze jeho korektnost a časovou složitostí se nebudeme zabývat. Poté ho s pomocí dalších poznatků zrychlíme na přijatelnou úroveň*.

Před samotným důkazem korektnosti algoritmu 5.2 zformulujeme ještě pomocné lemma, které říká, že množina P permutací σ' zkonstruovaných cyklem 4 je dominantní† vůči množině optimálních permutací.

Pozorování 5.8: *O dominantní množině*
Mějme množinu boolovských funkcí X a $P \subseteq \mathcal{S}_{\mathrm{Var}(X)}$, pak:

$$(\forall x \in \mathrm{Var}(X))(\exists \sigma \in \mathrm{Perm}^*(X_{\{x\}}))(x.\sigma \in P) \Rightarrow P \cap \mathrm{Perm}^*(X) \neq \emptyset$$

Důkaz. Uvažme π optimální permutaci pro X. Označme $x := \pi[1]$. Dle předpokladů $\exists \sigma \in \mathrm{Perm}^*(X_{\{x\}})$ takové, že $x.\sigma \in P$. Protože je σ optimální pro $X_{\{x\}}$, tak $\mathrm{Br}(X_{\{x\}}, \sigma) \leq \mathrm{Br}(X_{\{x\}}, \pi[2..])$, a tedy dle pozorování 5.6 $\mathrm{Br}(X, x.\sigma) \leq \mathrm{Br}(X, \pi)$, takže $x.\sigma$ je také optimální permutace pro X. ♡

*Tedy polynomiální časovou složitost pro pevný počet zlomů a řešitelnou třídu DNF.
†Množina X je dominantní vůči množině Y, pokud $X \cap Y \neq \emptyset$.

Značení: Pro pohodlí definujeme novou konstantu Null s následujícími vlastnostmi: $\mathrm{Br}(f, \mathrm{Null}) := \infty$ a $\pi.\mathrm{Null} := \mathrm{Null}$.

```
1 Function NaiveOptPerm(X)
      Input: Multimnožina boolovských funkcí X
      Output: π optimální permutace
2     if Var(X) = ∅ then return ∅
3     σ ← Null
4     for x ∈ Var(X) do
5         σ' ← x.NaiveOptPerm(X_{x})
6         if Br(X, σ') < Br(X, σ) then σ ← σ'
7     end for
8     return σ
9 end
```

Algoritmus 5.2: Naivní výpočet optimální permutace množiny boolovských funkcí

Pozorování 5.9: *O korektnosti Naivního algoritmu (5.2)*
Algoritmus 5.2 najde optimální permutaci.

Důkaz. Algoritmus skončí, protože každé rekurzivní volání probíhá na množině s menším množstvím proměnných, než měl vstup, a cyklus 4 je proveden nejvýše $|\mathrm{Var}(X)|$-krát.

Korektnost ukážeme indukcí dle hloubky rekurze. Případ $\mathrm{Var}(X) = \emptyset$ je triviální. Indukční krok plyne přímo z předešlého pozorování, protože cyklus 4 konstruuje množinu $P := \left\{x.\texttt{NaiveOptPerm}(X_{\{x\}}) : \ x \in \mathrm{Var}(X)\right\}$ vyhovující jeho předpokladům a vrací její nejlepší prvek. ♡

5.2 Naivní algoritmus s ořezáváním

Problémem algoritmu 5.2 je časová složitost, protože provádí rekurzi hloubky n s větvícím faktorem až n. Zde nám pomůže maximální blok množiny funkcí definovaný v následující kapitole, který umožní, pokud je neprázdný, vybrat vhodnou proměnnou bez backtrackování.

To samo nestačí, a proto přidáme ještě parametr k určující maximální počet zlomů v nalezené permutaci, díky kterému budeme moci ořezávat neperspektivní větve výpočtu. Pro snazší dokazovaní nejprve ukážeme variantu naivního algoritmu s ořezáváním a až v následující kapitole do něj zapracujeme užívání proměnných z bloku.

Značení: Buď X množina boolovských funkcí, pak $\|X\| := |X \setminus \{\mathbf{0}, \mathbf{1}\}|$.

Jediným rozdílem oproti algoritmu 5.2 je přidání ořezávací podmínky na řádku 2. Všimněme si, že $\|X\|$ je vždy dolní odhad pro $\mathrm{Br}^*(X)$, a tedy ořezávání touto podmínkou je korektní.

```
1  Function NaiveOptPerm2(X, k)
       Input: Multimnožina boolovských funkcí X, maximální počet
              zlomů k
       Output: Optimální permutace s nejvýše k zlomy, nebo Null
               pokud taková neexistuje
2      if ‖X‖ > k then return Null
3      if Var(X) = ∅ then return ∅
4      σ ← Null
5      for x ∈ Var(X) do
6          σ' ← x.NaiveOptPerm2(X_{x}, k − Br_{x}(X))
7          if Br(X, σ') < Br(X, σ) then σ ← σ'
8      end for
9      return σ
10 end
```

Algoritmus 5.3: Naivní výpočet optimální permutace množiny boolosvkých funkcí s ořezáváním

Pozorování 5.10: *O korektnosti Naivního algoritmu s ořezáváním (5.3)* Algoritmus 5.3 skončí a vydá $\pi \in \mathrm{Perm}^*(X)$ takové, že $\mathrm{Br}(X, \pi) \leq k$, pokud takové existuje, a Null jinak.

Důkaz. Z důkazu korektnosti algoritmu 5.2 plyne, že algoritmus 5.3 skončí, a vrátí-li permutaci π, je tato optimální pro X.

Pokud algoritmus vrátí permutaci π, tak na dně rekurze úspěšné větve výpočtu* platí $k \geq 0$. V průběhu této větve výpočtu bylo od parametru k odečteno $\sum_{i=1}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]})$, což je dle pozorování 5.5 rovno $\mathrm{Br}(X, \pi)$. Z toho plyne $k \geq \mathrm{Br}(X, \pi)$.

Naopak, pokud algoritmus vrátí Null, tak to znamená, že existuje množina prefixů S, která pokrývá† celou $\mathcal{S}_{\mathrm{Var}(X)}$ a $\forall \pi \in S$:

$$\|X_\pi\| + \sum_{i=1}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]}) > k$$

Zde si stačí všimnout, že $\|X_\pi\| + \sum_{i=1}^{|\pi|} \mathrm{Br}_{\{\pi[i]\}}(X_{\pi[1\,..\,i-1]})$ je dolní odhad na počet zlomů $\mathrm{Br}(X, \pi.\sigma)$ pro libovolné σ. ♡

*Tj. té, která zkonstruovala permutaci π.

†Množina S pokrývá množinu slov Y, pokud pro každé slovo $y \in Y$ existuje jeho prefix s (klidně i prázdný) takový, že $s \in S$.

6. Polynomiální algoritmus

V předchozí kapitole jsme rozšířili pojem počtu zlomů na množiny boolovských funkcí a představili jsme jednoduchý ale velmi pomalý algoritmus pro hledání optimálních permutací množin funkcí. Nyní připomene pojem maximálního bloku definovaný v kapitole 4, rozšíříme ho na množiny funkcí a s jeho pomocí zkonstruujeme lepší algoritmus pro hledání optimálních permutací.

6.1 Blokové permutace

Maximální blok jsme v kapitole 4 (definice 4.5) definovali jako množinu těch proměnných funkce f, vůči nimž je (alespoň) jedna polovina této funkce konstantní. Také jsme uvedli pozorovaní o dosazování za proměnnou z bloku do esenciální DNF. Na druhou stranu jsme zcela ignorovali význam maximálního bloku z hlediska počtu zlomů funkce. To nyní napravíme.

Nejprve rozšíříme pojem maximálního bloku na množiny funkcí. Maximální blok množiny funkcí je průnik bloků jednotlivých funkcí, tedy $x \in \mathrm{MaxB}(X)$, pokud $x \in \mathrm{MaxB}(f)$ pro každou $f \in X$.

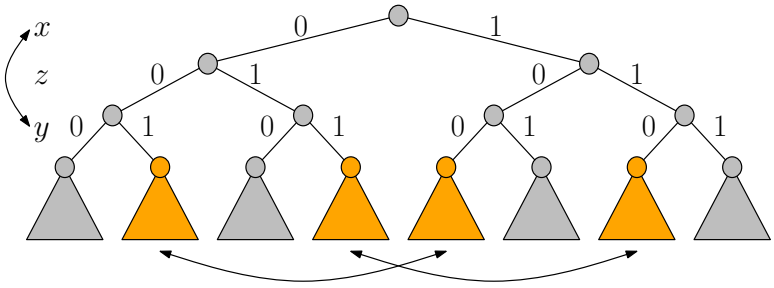
Definice 6.1: *Maximální blok množiny funkcí*
Maximální blok množiny boolovských funkcí X je:

$$\mathrm{MaxB}(X) := \bigcap_{f \in X} \mathrm{MaxB}(f)$$

Mějme permutaci $y.\sigma.x.\tau$ funkce f a $x \in \mathrm{MaxB}(f)$. První pozorování říká, můžeme x vyměnit s y a počet zlomů tím nezvýšíme.

Pozorování 6.2: *O proměnné z bloku*
Buď $\pi = x.\sigma.y.\tau$ permutace boolovské funkce f takové, že $x \in \mathrm{MaxB}(f)$. Pak $\mathrm{Br}(f, x.\sigma.y.\tau) \leq \mathrm{Br}(f, y.\sigma.x.\tau)$.

Důkaz. BÚNO buď x takové, že $f[x := 0] \equiv 1$. Pak levá část grafu této f vůči permutaci π je konstantní 1. Transpozice x a y znamená prohození částí grafu mezi levou a pravou polovinou (resp. prohození podstromů v rozhodovacím stromě). Uvažme tedy podstrom A z levé poloviny a podstrom B z pravé poloviny, které si prohozením x a y vymění pozice.



Obrázek 6.1: Příklad k důkazu pozorování 6.2 pro $\sigma = z$

Zachováme-li vhodné pořadí prohazování (zleva je-li levá polovina konstantní), tak je každý podstrom A těsně před prohozením z obou stran obklopen hodnotami 1 vyjma degenerovaného případu $|\sigma| = 0$, ten ale vyřešíme zvlášť na závěr. Pokud $B(\nvdash) = B(\not\nvdash) = 1$, tak se počet zlomů nezmění. Pokud $B(\nvdash) = 0$, tak v levé části přibude zlom na počátku B a v pravé na počátku A přibude či ubude, celkově tedy počet zlomů nezmenší. Obdobně pro $B(\not\nvdash)$.

Pro degenerovaný případ $y.x.\tau \to x.y.\tau$ máme ve hře 3 potenciální zlomy. Zlom na levé straně A není, takže má-li prohození počet zlomů snížit, musí být zlomy jak mezi A a B, tak i napravo od B. (Protože počet zlomů zachovává paritu.) Navíc po prohození nesmí nalevo od B vzniknout zlom, tedy $B(\nvdash) = 1$. Byl-li na počátku zlom mezi A a B, tak $B(\nvdash) = 0$. Spor. ♡

Aplikujeme-li toto pozorování na optimální permutaci a množinu funkcí X, zjistíme, že $\forall x \in \mathrm{MaxB}(X)$ existuje optimální permutace množiny funkcí X začínající proměnnou x.

Důsledek 6.3: *O proměnné z bloku množiny funkcí*
Mějme množinu funkcí X a $\pi = y.\sigma.x.\tau$ její optimální permutaci taková, že $x \in \mathrm{MaxB}(X)$. Pak i permutace $x.\sigma.y.\tau$ je optimální.

Všechny prezentované algoritmy pro výpočet optimální permutace konstruují permutace od začátku. Proto je přirozené pokusit se užít tento důsledek rekurzivně. To vede k definici blokové permutace. Elegantní rekurzivní definice blokové permutace je následující: Prázdná permutace je bloková. Permutace $x.\sigma$ je bloková pro množinu X, pokud $x \in \mathrm{MaxB}(X)$ nebo $\mathrm{MaxB}(X) = \emptyset$ a σ je bloková permutace pro $X_{\{x\}}$. Ekvivalentní nerekurzivní definice je uvedena níže.

Definice 6.4: *Bloková permutace*
Mějme množinu boolovských funkcí X. Pak permutaci $\pi \in \mathcal{S}_{\mathrm{Var}(X)}$ říkáme **bloková**, pokud $\forall \sigma, x$, kde $\sigma.x$ je prefix π, platí $\mathrm{MaxB}(X_\sigma) \neq \emptyset \Rightarrow x \in \mathrm{MaxB}(X_\sigma)$. Dále permutace π je bloková pro boolovskou funkci f, pokud π je bloková pro $\{f\}$.

Rekurzivní aplikací předchozího důsledku zjistíme, že každá funkce má optimální blokovou permutaci. Alternativní důkaz tohoto pozorování plyne z korektnosti algoritmu 6.2, který konstruuje optimální blokové permutace.

Pozorování 6.5: *O existenci blokové optimální permutace*
Každá množina boolovských funkcí má optimální blokovou permutaci.

Důkaz. Uvážíme nějakou optimální permutaci a rekurzivní aplikací důsledku 6.3 z ní uděláme blokovou permutaci, aniž bychom porušili optimalitu. ♡

Poznamenejme ještě, že není pravda, že každá bloková permutace je optimální a ani že každá optimální permutace je bloková. Příkladem prvního případu budiž funkce $g = abcd \vee ab\bar{c}\bar{d}$. Tato funkce má právě dva truepointy, jedním z nich je $\not\nvdash$ a vůči žádné permutaci tyto truepointy nejsou vedle sebe. Proto má g vůči každé permutaci právě 3 zlomy. Na druhou stranu $\mathrm{MaxB}(g) = \{a, b\}$, takže např. permutace *dcba* je optimální, ale není bloková.

Druhý případ můžeme ilustrovat na funkci $h = ab \vee cd$, která byla protipříkladem hypotézy o existenci rekurzivně optimální permutace v kapitole 2. Uvažme permutaci $\pi = acbd$. S pomocí obrázku 2.1 jsme nahlédli, že π není optimální permutace pro h, protože má 7 zlomů, zatímco h je 5-zlomová. Zbývá nahlédnout, že π je bloková.

Sama h má prázdný blok, takže a blokovou podmínku neporušuje. Zkusme druhou proměnnou – $\{h\}_{\{a\}} \setminus \{\mathbf{0}, \mathbf{1}\} = \{b \vee cd, cd\}$, $\mathrm{MaxB}(b \vee cd) = \{b\}$ a $\mathrm{MaxB}(cd) = \{c, d\}$, takže $\mathrm{MaxB}(\{h\}_{\{a\}}) = \mathrm{MaxB}(b \vee cd) \cap \mathrm{MaxB}(cd) = \emptyset$, a ani druhá proměnná blokovou podmínku neporušuje.

Zbývá ověřit třetí proměnnou (protože všechny permutace délky 1 jsou blokové, poslední proměnnou kontrolovat nemusíme). Máme množinou podfunkcí $\{h\}_{\{a,c\}} \setminus \{\mathbf{0}, \mathbf{1}\} = \{b, b \vee d, d\}$, její maximální blok je opět prázdný, takže ani třetí proměnná blokovou podmínku neporušuje, a $acbd$ je bloková permutace.

6.2 Rekurzivní verze

Nyní představíme mírnou modifikaci Naivního algoritmu s ořezáváním, která již poběží v čase řádově n^k místo n^n. Jediným rozdílem je užití proměnné z bloku, pokud taková existuje. Díky tomu se zbavíme velké části backtrackování. U tohoto algoritmu opět složitost dokazovat nebudeme, protože po něm uvedeme jeho iterativní verzi, která bude pro analýzu lepší.

```
1  Function OptPermRec(X, k)
       Input: Multimnožina boolovských funkcí X, maximální počet
              zlomů k
       Output: Optimální permutace s nejvýše k zlomy, nebo Null
               pokud taková neexistuje
2      if ‖X‖ > k then return Null
3      if Var(X) = ∅ then return ∅
4      if ∃x ∈ MaxB(X) then
5          return x.OptPermRec(X_{x}, k − Br_{x}(X))
6      end if
7      σ ← Null
8      for x ∈ Var(X) do
9          σ′ ← x.OptPermRec(X_{x}, k − Br_{x}(X))
10         if Br(X, σ′) < Br(X, σ) then σ ← σ′
11     end for
12     return σ
13 end
```

Algoritmus 6.2: Výpočet optimální blokové permutace množiny boolovských funkcí – rekurzivní verze

Pozorování 6.6: *O korektnosti Rekurzivního algoritmu (6.2)*
Algoritmus 6.2 zastaví a vydá permutaci $\pi \in \mathrm{Perm}^*(X)$ takovou, že $\mathrm{Br}(X, \pi) \leq k$, pokud taková existuje, a Null jinak.

Důkaz. Jediným rozdílem oproti algoritmu 5.3 je přidání podmínky 4, která zvolí proměnnou z maximálního bloku, pokud nějaká taková existuje. Pokud algoritmus 5.3 vrací Null, tak zjevně Null vrací i tento algoritmus.

Naopak pokud algoritmus 5.3 vrátí permutaci: Pokud podmínka 4 neplatí, tak se algoritmus chová stejně jako algoritmus 5.3. Pokud platí, tak dle pozorování 6.3, lze x doplnit na optimální permutaci, takže opět najdeme a vrátíme optimální permutaci. ♡

6.3 Iterativní verze

Kvůli analýze složitosti budeme v iterativní verzi pracovat s množinou DNF $\mathcal{X}$ místo množiny funkcí X. Kromě toho se iterativní verze od rekurzivní liší nahrazením rekurze v případě $x \in \mathrm{MaxB}(X)$ cyklem, explicitním výpočtem $\mathcal{X}_{\{x\}}$ a $\mathrm{Br}_{\{x\}}(\mathcal{X})$ (dle algoritmu 6.3) a průběžným čištěním množiny $\mathcal{X}$ od konstantních DNF. Tyto úpravy děláme především pro snazší analýzu. Speciálně díky převedení rekurze v případě proměnné z maximálního bloku na cyklus máme jen jedno místo, kde v algoritmu probíhá rekurze.

Input: Množina DNF $\mathcal{X}$, maximální počet zlomů k, proměnná $x \in \mathrm{Var}(\mathcal{X})$
Output: Množina $\mathcal{X}' = \mathcal{X}_{\{x\}} \setminus \{\mathbf{0}, \mathbf{1}\}$ a $k' = k - \mathrm{Br}_{\{x\}}(\mathcal{X})$
1 $k' \leftarrow k$
2 $X' \leftarrow \emptyset$
3 **for** $\mathcal{F} \in \mathcal{X}$ **do**
4 $\quad \mathcal{X}' \leftarrow \mathcal{X}' \uplus (\{\mathcal{F}[x := 0], \mathcal{F}[x := 1]\} \setminus \{\mathbf{0}, \mathbf{1}\})$
5 $\quad$ **if** $\mathcal{F}[x := 0](\not\vdash) \neq \mathcal{F}[x := 1](\vdash)$ **then** $k' \leftarrow k' - 1$
6 **end for**

Algoritmus 6.3: Explicitní výpočet $\mathcal{X}_{\{x\}}$ a $\mathrm{Br}_{\{x\}}(\mathcal{X})$

Věta 6.7: *O korektnosti Iterativního algoritmu (6.4)*
Algoritmus 6.4 zastaví a vrátí správný výsledek.

Důkaz. Porovnáme běh s algoritmem `OptPermRec`. První rozdíl je, že množinu $\mathcal{X}$ udržujeme bez konstantních funkcí, takže můžeme používat $|.|$ místo $\|.\|$. Dále rekurzivní volání s $x \in \mathrm{MaxB}(\mathcal{X})$ (řádek 5 v algoritmu 6.2) byla převedena na cyklus, což si vyžádalo přidání ukončující podmínky 17. Poslední změnou je přidání podmínky pro neúspěšné ukončení na řádku 18. Ta je pouhou rychlostní optimalizací, aby se neprováděl následující cyklus, pokud nemá šanci uspět*. ♡

*Protože $|\mathcal{X}| = k \Rightarrow |\mathcal{X}'| > k'$, jak dokážeme v 6.11, a každé rekurzivní volání by okamžitě selhalo.

```
1  Function OptPerm(𝒳, k)
       Input: Multimnožina DNF 𝒳, maximální počet zlomů k
       Output: Optimální permutace s nejvýše k zlomy, nebo Null
               pokud taková neexistuje
2      return OptPerm'(𝒳 \ {0, 1}, k)
3  end

4  Function OptPerm'(𝒳, k)
5      if |𝒳| > k then return Null
6      π ← ∅
7      while ∃x ∈ MaxB(𝒳) do
8          π ← π.x
9          𝒳' ← ∅
10         for ℱ ∈ 𝒳 do
11             𝒳' ← 𝒳' ⊎ ({ℱ[x := 0], ℱ[x := 1]} \ {0, 1})
12             if ℱ[x := 0](⊮) ≠ ℱ[x := 1](⊩) then k ← k − 1
13         end for
14         𝒳 ← 𝒳'
15         if |𝒳| > k then return Null
16     end while
17     if Var(𝒳) = ∅ then return π
18     if k = |𝒳| then return Null
19     σ ← Null
20     for x ∈ Var(𝒳) do
21         k' ← k
22         𝒳' ← ∅
23         for ℱ ∈ 𝒳 do
24             𝒳' ← 𝒳' ⊎ ({ℱ[x := 0], ℱ[x := 1]} \ {0, 1})
25             if ℱ[x := 0](⊮) ≠ ℱ[x := 1](⊩) then k' ← k' − 1
26         end for
27         σ' ← x.OptPerm'(𝒳', k')
28         if Br(𝒳, σ') < Br(𝒳, σ) then σ ← σ'
29     end for
30     return π.σ
31 end
```

Algoritmus 6.4: Výpočet optimální blokové permutace množiny DNF – iterativní verze

6.4 Časová složitost

Nyní spočteme časovou složitost Iterativní verze algoritmu pro nalezení optimální permutace. Algoritmus bude implementovat na Pointer Machine pomocí datových struktur popsaných v kapitole 3. Na vstupu bude DNF v naivní

reprezentaci a výstupem bude permutace reprezentovaná jako spojový seznam ukazatelů na proměnné. Uvnitř algoritmu budeme pro DNF užívat rozšířenou reprezentaci a s proměnnými pracovat pomocí handlů.

Opět předpokládáme, že všechny DNF na vstupu patří do nějaké pevně dané řešitelné třídy $\mathcal{C}$ a $T(\|\mathcal{F}\|)$ značí čas nutný pro převod DNF $\mathcal{F} \in \mathcal{C}$ na esenciální tvar.

Značení:

- n – počet proměnných
- l – počet literálů nejdelší DNF z množiny $\mathcal{X}$
- $T(\|\mathcal{F}\|)$ – čas nutný pro převod $\mathcal{F} \in \mathcal{C}$ na esenciální tvar

Volání `OptPerm` bez volání `OptPerm'` převede všechny $\mathcal{F} \in \mathcal{X}$ na esenciální tvar, změní jejich reprezentaci z naivní na rozšířenou, a poté odstraní konstantní funkce. Toto vše stihneme v čase $\mathcal{O}(|\mathcal{X}|\,(n + l + T(l)))$.

Pozorování 6.8: *O volání `OptPerm`*
Volání funkce `OptPerm` bez rekurze trvá $\mathcal{O}(|\mathcal{X}|\,(n + l + T(l)))$.

Důkaz. Viz výše. ♡

Hloubka rekurze

Nyní zanalyzujeme volání `OptPerm'`. Nejprve odhadneme maximální hloubku rekurze, což spolu s maximálním větvícím faktorem n dá odhad na maximální počet volání funkce `OptPerm'`. Poté upočítáme složitost jednotlivých částí funkce `OptPerm'` a dopočteme celkovou časovou složitost.

Hloubku rekurze odhadneme pomocí rozdílu $k - |\mathcal{X}|$. Ukážeme, že tento neroste v rámci jednoho volání `OptPerm'` a ostře klesá při rekurzi. K tomu využijeme následující pozorování.

Pozorování 6.9: *O odhadovaní počtu zlomů*
Buď X množina boolovských funkcí a $x \in \mathrm{Var}(X)$.

$$\|X\| \leq \mathrm{Br}_{\{x\}}(X) + \left\|X_{\{x\}}\right\|$$

Navíc pokud $x \notin \mathrm{MaxB}(X)$:

$$\|X\| < \mathrm{Br}_{\{x\}}(X) + \left\|X_{\{x\}}\right\|$$

Důkaz. Neostrá nerovnost: Stačí dokázat pro jednoprvkovou $X = \{f\}$, kde f je nekonstantní funkce. Tedy $\|X\| = 1$, takže potřebujeme ukázat, že $\mathrm{Br}_{\{x\}}(X)$ a $\left\|X_{\{x\}}\right\|$ nemohou být zároveň nula. Sporem. Snadno nahlédneme* $\|Y\| = 0 \Rightarrow \mathrm{Br}(Y, \pi) = 0$ pro libovolné π.

$$1 = \|X\| \leq \mathrm{Br}(X, \pi) = \mathrm{Br}_{\{x\}}(X) + \mathrm{Br}(X_{\{x\}}, \pi) = 0$$

*Protože konstantní funkce nemají žádné zlomy.

Spor.

Ostrá varianta: Stačí si povšimnout, že pro $x \notin \mathrm{MaxB}(f)$ je $\|\{f\}\| = 1$ a $\left\|\{f\}_{\{x\}}\right\| = 2$, a posčítat přes všechny $f \in X$ (pro alespoň jedno f platí $x \notin \mathrm{MaxB}(f)$ a pro ostatní platí neostrá varianta). ♡

> **Pozorování 6.10:** *O monotonii $k - |\mathcal{X}|$*
> V rámci jednoho volání funkce `OptPerm'` v algoritmu 6.4 hodnota $k - |\mathcal{X}|$ nestoupá.

Důkaz. Jediné místo, které modifikuje $\mathcal{X}$ a k, je cyklus 7. Označíme-li $\mathcal{X}'$ a k' hodnoty $\mathcal{X}$ a k po jednom průběhu tohoto cyklu, tak platí $\mathcal{X}' = \mathcal{X}_{\{x\}} \setminus \{\mathbf{0}, \mathbf{1}\}$ a $k' = k - \mathrm{Br}_{\{x\}}(\mathcal{X})$.

$$k - |\mathcal{X}| = (k' + \mathrm{Br}_{\{x\}}(\mathcal{X})) - \|\mathcal{X}\| \geq k' - \|\mathcal{X}'\| = k' - |\mathcal{X}'|$$

Protože dle předchozího pozorování $\mathrm{Br}_{\{x\}}(\mathcal{X}) - \|\mathcal{X}\| \geq -\left\|\mathcal{X}_{\{x\}}\right\|$. ♡

> **Pozorování 6.11:** *O poklesu $k - |\mathcal{X}|$*
> Při rekurzivním volání na řádku 27 funkce `OptPerm'` v algoritmu 6.4 platí $k' - |\mathcal{X}'| < k - |\mathcal{X}|$.

Důkaz. Analogicky s předchozím důkazem, jen užijeme ostrou variantu nerovnosti z pozorování 6.9. ♡

> **Důsledek 6.12:** *O hloubce rekurze*
> Hloubka rekurze funkce `OptPerm'` v algoritmu 6.4 je nejvýše $k - 1$.

Důkaz. Na počátku výpočtu je $k - |\mathcal{X}| \leq k - 1$ a rekurze se zastaví ve chvíli, kdy $k - |\mathcal{X}| \leq 0$. Dle předchozích pozorování toto nastane po nejvýše $k - 1$ úrovních rekurze. ♡

Složitost jednoho průběhu funkce `OptPerm'`

Nyní víme, že hloubka rekurze je $k - 1$ a větvící faktor je nejvýše n, takže celkem volání `OptPerm'` je $\mathcal{O}(n^{k-1})$. Zbývá spočítat, jak dlouho trvá jedno volání. Jeden průběh funkce `OptPerm'` rozdělíme na několik částí. Vše kromě cyklů 7 a 20 stihneme v konstantním čase*.

Cyklus 7, kde zpracováváme proměnné z maximálního bloku, potřebuje při jednom průběhu konstantní čas na inicializaci proměnných, přiřazení $\mathcal{X} \leftarrow \mathcal{X}'$ a test $|\mathcal{X}| > k$. Vnořený cyklus: V částečném dosazení na řádku 11 je alespoň jedna z výsledných funkcí konstantní (protože dosazujeme za proměnnou z bloku), takže stačí dosadit pouze hodnotu dávající netriviální výsledek† a DNF nemusíme kopírovat. Navíc víme‡, že za dobu života jedné DNF strávíme těmito až n dosazeními dohromady čas $\mathcal{O}(n + l)$.

*Pro inicializace a test $\mathrm{Var}(\mathcal{X}) = \emptyset$ je to zřejmé, pro testy na velikost množiny $\mathcal{X}$ si u ní musíme pamatovat, zda je velká k či větší. To ale zjistíme již při její konstrukci.

†Která to je, poznáme v $\mathcal{O}(1)$ z rozšířené reprezentace.

‡Viz vlastnosti rozšířené reprezentace v závěru kapitoly 3.

Čas na dosazení tedy budeme účtovat jednotlivým DNF. Test na zlom uprostřed, který stihneme v $\mathcal{O}(1)$, taktéž. Dohromady tedy v cyklu 7 strávíme čas $\mathcal{O}(1)$ plus čas za dosazování do DNF. Každá DNF do, které dosazujeme, musela někdy vzniknout a při svém vzniku byla transformována do esenciálního tvaru, což trvá čas $\mathcal{O}(n+l)$, takže dosazování můžeme naúčtovat této transformaci.

Pozorování 6.13: *O složitosti deterministické části*
Jedno volání funkce `OptPerm'` bez cyklu 20 a s cyklem 10 účtovaným výrobě esenciálních DNF trvá $\mathcal{O}(1)$.

Důkaz. Viz výše. ♡

Zbývá rozebrat cyklus 20. Ten proběhne až n-krát a vyjma vnořeného cyklu trvá jeden průběh $\mathcal{O}(k)$*. Vnořený cyklus může proběhnout až k-krát a čas na jeden jeho průběh je nejvýše $\mathcal{O}(n+l+T(l))$. Celkem tedy v cyklu 20 (bez rekurze) strávíme čas $\mathcal{O}(nk(n+l+T(l)))$.

Pozorování 6.14: *O složitosti spekulativního cyklu*
Vykonání cyklu 20 ve funkci `OptPerm'` trvá bez započtení rekurzivních volání trvá $\mathcal{O}(nk(n+l+T(l)))$.

Důkaz. Viz výše. ♡

Celková složitost

Nyní zbývá poskládat výše uvedená pozorování dohromady. Především si všimneme, že cyklus 20 neprobíhá v listech rekurze†. Všech vrcholů stromu rekurze je $\mathcal{O}(n^{k-1})$, ale vrcholů, které nejsou listy, je nevýše $\mathcal{O}(n^{k-2})$. To dává čas $\mathcal{O}(n^{k-1}k(n+l+T(l)))$ na cykly 20 a $\mathcal{O}(n^{k-1})$ na zbytek volání `OptPerm'`. Připočteme-li čas $\mathcal{O}(|\mathcal{X}|\,(n+l+T(l)))$ nutný na volání `OptPerm` dostáváme časovou složitost algoritmu 6.4:

Věta 6.15: *O složitosti iterativního algoritmu*
Časová složitost algoritmu 6.4 je

$$\mathcal{O}((|\mathcal{X}|+kn^{k-1})(n+l+T(l)))$$

Důkaz. Viz výše. ♡

Na závěr ještě zanalyzujme složitost algoritmu 6.4 v případě, že vstupem jsou DNF z třídy positivních funkcí. Positivní DNF vždy obsahují všechny své esenciální termy, takže je nijak transformovat nemusíme a $T(l)=\mathcal{O}(1)$. Trochu méně patrným faktem je, že máme-li nekonstantní positivní funkci f a x její proměnnou, která nenáleží do maximálního bloku, tak nejen že f má vůči x zlom jak v levé tak v pravé polovině (což plyne z $x \notin \mathrm{MaxB}(f)$), ale má i zlom uprostřed.

*Vše, krom porovnání počtu zlomů permutací stihneme v konstantním čase, ale protože nemáme čísla, tak i když si počet zlomů u každé permutace pamatujeme, trvá jejich porovnání $\mathcal{O}(k)$, což je horní odhad na počet zlomů.

†Pokud by proběhl, tak by provedl rekurzi, a nebyly by to listy.

Pozorování 6.16: *O zlomu uprostřed positivní funkce*
Buď f nekonstantní positivní funkce a $x \in \mathrm{Var}(f) \setminus \mathrm{MaxB}(f)$ její proměnná. Pak $Br_{\{x\}}(f) = 1$.

Důkaz. Protože f je positivní, tak i $f[x := 0]$ a $f[x := 1]$ jsou positivní funkce. Z předpokladů jsou obě nekonstantní. Pro každou nekonstantní positivní funkci g platí $g(\nvdash) = 0$ a $g(\nVdash) = 1$. ♡

Z tohoto pozorování plyne, že pro positivní DNF při rekurzi klesá rozdíl $k - |\mathcal{X}|$ ne jen o jedna, ale vždy alespoň o 2. Tedy hloubka rekurze je nejvýše* $\mathcal{O}(n^{\frac{k-1}{2}})$. To pro positivní DNF zlepšuje odhad celkové složitosti na $\mathcal{O}((|\mathcal{X}| + kn^{\frac{k-1}{2}})(n + l))$.

Věta 6.17: *O složitosti iterativního algoritmu pro positivní DNF*
Časová složitost algoritmu 6.4 pro positivní DNF je

$$\mathcal{O}((|\mathcal{X}| + kn^{\frac{k-1}{2}})(n + l))$$

Důkaz. Viz výše. ♡

Pokud bychom zapojili ještě lemma 2.4 z článku [3] tvrdící, že každá primární positivní právě 3-zlomová† DNF má tvar

$$xy \vee x\mathcal{F}_x \vee y\mathcal{F}_y$$

kde $\mathcal{F}_x$ a $\mathcal{F}_y$ jsou positivní DNF neobsahující proměnné x a y, a následující lemmata, která říkají, že existuje optimální permutace začínající libovolným takovýmto párem x, y, můžeme docílit rozpoznání 3-zlomových positivních funkcí v čase $\mathcal{O}(n + l)$ a celý algoritmus poběží v čase‡ $\mathcal{O}((|\mathcal{X}| + kn^{\frac{k-3}{2}})(n + l))$ (pro $k \geq 3$).

*Všimněme si, že netriviální positivní funkce má vždy lichý počet zlomů, takže algoritmus pro positivní DNF má smysl spouštět pouze s lichým k. Tedy zlomek $\frac{k-1}{2}$ je vždy celé číslo.

†Tj. taková, která není 2-zlomová.

‡Samozřejmě bychom museli upravit Rozšířenou reprezentaci, aby dokázala zjistit, že term DNF má popsaný tvar. To lze zajistit přidáním dalších ukazatelů do počítadla výskytů literálů – tyto nové ukazatele budou počítat na tomtéž spojovém seznamu ale odzadu. Pak stačí projít všechny kvadratické termy a u každého se podívat, zda počítadlo „popředu“ jednoho literálu a „pozadu“ druhého ukazují o 1 vedle sebe (správným směrem). Detaily ponecháváme k rozmyšlení čtenáři.

Závěr

V této práci jsme se zabývali rozpoznáváním DNF, které reprezentují l-intervalové funkce. Protože je tento problém pro obecné DNF coNP-těžký, omezili jsme naše zkoumání na řešitelné třídy DNF. Pojem l-intervalové funkce jsme nahradili technicky šikovnějším pojmem k-zlomové funkce. Poté jsme ukázali, že lze snadno oddělit problém nalezení optimální permutace proměnných dané DNF od nalezení jejích zlomů vůči této permutaci.

Pro hledání zlomů pro DNF z řešitelné třídy a danou permutaci jejích proměnných jsme ukázali algoritmus, který pracuje v polynomiálním čase vůči součtu velikostí vstupu a výstupu (zlomů může být exponenciálně mnoho vůči délce DNF). Pro hlavní problém – nalezení optimální permutace – jsme předvedli algoritmus pracující v čase $\mathcal{O}(kn^{k-1}(n+l+T(l)))$, kde $T(l)$ je čas na transformaci DNF délky l na esenciální tvar a k je maximální počet zlomů. Tedy pro pevný počet zlomů je tento algoritmus polynomiální, což je hlavní výsledek.

Navíc jsme ukázali, že tento algoritmus běží rychleji, jsou-li vstupní DNF positivní – místo členu řádově n^k pouze $n^{\frac{k}{2}}$. Kromě toho jsme ukázali, že výsledky prezentované v článku [2] nepotřebují výpočetní sílu RAMu, ale stačí Pointer Machine. Totéž jsme naznačili pro algoritmus pro rozpoznání 2-intervalových positivních DNF z článku [3].

Seznam použité literatury

[1] Schieber, Baruch, Geist, Daniel and Zaks, Ayal. Computing the minimum DNF representation of boolean functions defined by intervals. *Discrete Applied Mathematics*, 149:154–173, 2005.

[2] Čepek, Ondřej, Kronus, David and Kučera, Petr. Recognition of interval Boolean functions. *Annals of Mathematics and Artificial Intelligence*, 52(1):1–24, 2008.

[3] Kronus, David and Čepek, Ondřej. Recognition of positive 2-interval Boolean functions. In *Proceedings of 11th Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty*, 115–122. 2008.

[4] Crama, Yves and Hammer, Peter L. *Boolean Functions: Theory, Algorithms, and Applications*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011.

[5] Cook, Stephen A. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium*, 151–158. New York: ACM, 1971.

[6] Garey, Michael R. and Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[7] Dowling, William F. and Gallier, Jean H. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[8] Itai, A. and Makowsky, J.A. Unification as a complexity measure for logic programming. *The Journal of Logic Programming*, 4(2):105–117, 1987.

[9] Minoux, M. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Inf. Process. Lett.*, 29(1):1–12, 1988.

[10] Mareš, Martin. *Saga of Minimum Spanning Tree*. Disertační práce, Matematicko-fyzikální fakulta Univerzity Karlovy, 2008.

[11] Mareš, Martin. *Krajinou grafových algoritmů*, 2007. Skripta k přednášce Grafové algoritmy.

[12] Syropoulos, Apostolos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, 347–358. London, UK: Springer-Verlag, 2001.

[13] Singh, D., Ibrahim, A. M., Yohanna, T. and Singh, J. N. An overview of the applications of multisets. *Novi Sad J. Math*, 37(2):73–92, 2007.

[14] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. *Introduction To Algorithms*. MIT Press, 2001.

Seznam obrázků a algoritmů

Seznam definic

Příloha A – Notace

Obecné značení

- **Pravdivostní hodnota výroku.** Mějme φ predikátový výraz (třeba $a = b$ či $a < c$), pak
$$[\![\varphi]\!] \begin{cases} 1 & \text{pokud je } \varphi \text{ pravdivý} \\ 0 & \text{jinak} \end{cases}$$
- **Funkce.** Definiční obor funkce f značíme $\mathrm{Dom}(f)$, obor hodnot $\mathrm{Rng}(f)$ (pozor platí $\mathrm{Rng}(f) \supset f[\mathrm{Dom}(f)]$, rovnost obecně neplatí).
- **Restrikce funkce.** Restrikci funkce f na množinu $A \subseteq \mathrm{Dom}(f)$ značíme $f \restriction A$.
- **Množina všech funkcí.** Množinu (či třídu) všech funkcí z A do B značíme $[A \to B]$.
- **Třída všech objektů.** Třídu všech objektů značíme $\mathcal{V}$.
- **Množina přirozených čísel.** $\mathbb{N} := \{0, 1, 2, \dots\}$.
- **Interval celých čísel.** Pro $a, b \in \mathbb{Z}$:
$$[a, b] := \{a, a + 1, a + 2, \dots, b\}$$
- **Řetězce.** Řetězce typicky značíme π, σ, τ. Písmena abecedy značíme x, y, z. Konkatenaci řetězců značíme $\pi.\tau$. Písmena abecedy, nad níž jsou příslušné řetězce konstruovány, ztotožňujeme s řetězci délky 1. Je-li to nutné, interpretujeme řetězec jako multimnožinu jeho písmen. Tedy $|\pi|$ značí délku řetězce. Prázdný řetězec značíme $\emptyset$.
- **Indexování řetezců.** Písmeno na pozici i značíme $\pi[i]$ přičemž indexujeme od 1. Podřetězec od pozice i (včetně) do pozice j (také včetně) značíme $\pi[i\,..\,j]$. Dále $\pi[i\,..\,]$ je zkratka za $\pi[i\,..\,|\pi|]$ (tj. podřetězec od i do konce).
- **Permutace.** Permutace považujeme za speciální případ řetězců. Inverzi k permutaci značíme π^{-1} a je to funkce typu $\pi \to [1, |\pi|]$ (tj. z písmen do přirozených čísel) – pozor tato inverze **není** permutace, pokud π není permutace čísel 1 až $|\pi|$. Množinu všech permutací na množině X značíme $\mathcal{S}_X$.

Multimnožiny

Multimnožiny budeme chápat jako funkce z třídy všech objektů $\mathcal{V}$ do přirozených čísel, kde hodnota funkce je počet výskytů daného prvku v multimnožině. Máme-li multimnožiny $A, B : \mathcal{V} \to \mathbb{N}$, tak běžné operace jsou definovány následovně:

- $|A| := \sum_{x \in \mathcal{V}} A(x)$

- $x \in A := A(x) > 0$
- $A \subseteq B := (\forall x)(A(x) \leq B(x))$
- $A \cup B := x \mapsto \max\{A(x), B(x)\}$
- $A \uplus B := x \mapsto A(x) + B(x)$
- $A \cap B := x \mapsto \min\{A(x), B(x)\}$
- $A \setminus B := x \mapsto \begin{cases} 0 & \text{pokud } x \in B \\ A(x) & \text{jinak} \end{cases}$

Zde uvedená definice rozdílu je mírně nestandardní (běžnější je* $A \setminus B := x \mapsto \max\{0, A(x) - B(x)\}$), ale pro naše účely bude vhodnější.

Množiny můžeme chápat jako multimnožiny, pro něž platí $(\forall x)(A(x) \in \{0, 1\})$. Povšimněme si, že pro všechny uvedené operace vyjma $\uplus$, je výsledek množina vždy, když jsou oba parametry množiny, a na množinách se chovají standardním způsobem.

Asymptotická notace

Pro formální zavedení asymptotické notace v kontextu výpočetní složitosti vizte kupř. [14]. Definice jednotlivých asymptotických tříd zde uvádíme pouze pro úplnost†. Dodejme, že ač jsou symboly $\mathcal{O}()$, $o()$, $\Theta()$, $\Omega()$ a $\omega()$ zavedeny jako třídy funkcí, je ustálené užívaní rovnítka místo symbolu $\in$ (a však příslušná třída v tomto případě musí být vždy na pravé straně). Obdobně se užívá značení typu $2^{\mathcal{O}(n)}$ jako zkratka za $\left\{2^f \mid f \in \mathcal{O}(n)\right\}$.

$$\begin{aligned}
\mathcal{O}(f) &:= \{g \mid (\exists N, c > 0)(\forall n > N)(|g(n)| \leq c \cdot f(n))\} \\
o(f) &:= \{g \mid (\exists N > 0)(\forall c > 0)(\forall n > N)(|g(n)| \leq c \cdot f(n))\} \\
\Omega(f) &:= \{g \mid (\exists N, c > 0)(\forall n > N)(|g(n)| \geq c \cdot f(n))\} \\
\omega(f) &:= \{g \mid (\exists N > 0)(\forall c > 0)(\forall n > N)(|g(n)| \geq c \cdot f(n))\} \\
\Theta(f) &:= \{g \mid (\exists N, c_1, c_2 > 0)(\forall n > N)(c_1 \cdot f(n) \leq |g(n)| \leq c_2 \cdot f(n))\}
\end{aligned}$$

Třídu $\Theta(f)$ lze ekvivalentně zavést jako $\mathcal{O}(f) \cap \Omega(f)$.

*S definicí $A \setminus B := x \mapsto \max\{0, A(x) - B(x)\}$ totiž platí identita $(A \uplus B) \setminus B = A$ – tedy rozdíl je inverzí k multimnožinovému sjednocení. Více o multimnožinách vizte [12, 13].

†Uvedené definice úmyslně vynechávají určení nad jakými třídami funkcí pracují, protože diskuze technických podrobností přesahuje rámec této přílohy. Typicky stačí uvažovat funkce $\mathbb{N} \to \mathbb{N}$.