VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO BINÁRNÍ ROZHODOVACÍ DIAGRAMY

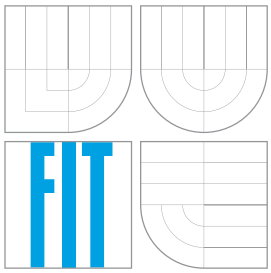
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

PETR JANKŮ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO BINÁRNÍ ROZHODOVACÍ DIAGRAMY

A LIBRARY FOR BINARY DECISION DIAGRAMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

PETR JANKŮ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. LUKÁŠ HOLÍK, Ph.D.

BRNO 2015

Abstrakt

Efektivní manipulace Booleovských funkcí je důležitou součástí mnoha počítačových návrhů. Jako datová struktura pro reprezentaci a manipulaci s Booleovskými funkcemi se běžně používají binární rozhodovací diagramy. Tyto diagramy se běžně používají v mnoha odvětvích, jako je například ověřování modelů, verifikace systému, návrh obvodů apod. V této práci jsou popsány tyto diagramy a jsou zde uvedeny i jejich modifikace. Dále jsou v této práci uvedeny a popsány techniky pro efektivní manipulaci a reprezentaci binárních rozhodovacích diagramů. Mimoto tato práce popisuje návrh a implementaci knihovny, která bude s těmito diagramy pracovat. Dále je diskutována potenciální aplikace vyvinuté knihovny v knihovně VATA pro manipulaci se stromovými automaty. Na závěr je tato knihovna porovnána s dobře známou a silně optimalizovanou knihovnou CUDD, která je volně dostupná a s knihovnou CacBDD. Výsledky experimentů ukázaly, že navrhovaná knihovna je poměrně blízká CUDD a CacBDD. (dosahuje srovnatelného a většinou i lehce lepšího výkonu)

Abstract

Efficient manipulation of Boolean functions is an important component of many computer-aided design task. As a data structure for representing and manipulating Boolean functions, Binary Decision Diagrams are commonly used. These diagrams are commonly used in many fields such as model checking, system verification, circuit design, etc. In this thesis we describe these diagrams and there are present their modifications. Furthermore, this paper present and describes techniques for effective handling and representation of binary decision diagrams. This thesis describes the design and implementation of library that will work with these diagrams. It is further discussed how the developed library can be used within the library VATA for manipulating tree automata. Finally, the library was compared with well known and heavily optimized library CUDD, which is public and with library CacBDD. The experimental results showed that the performance of the proposed library is quite close to that of CUDD a CacBDD. (has comparable and mostly even slightly better performance)

Klíčová slova

Booleovské funkce, symbolická manipulace, binární rozhodovací diagramy.

Keywords

Boolean functions, symbolic manipulation, binary decision diagrams.

Citace

Petr Janků: Knihovna pro binární rozhodovací diagramy, diplomová práce, Brno, FIT VUT v Brně, 2015

Knihovna pro binární rozhodovací diagramy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Mgr. Lukáše Holíka, Ph.D.

.....
Petr Janků
2. srpna 2015

Poděkování

Rád bych poděkoval Mgr. Lukáši Holíkovi, Ph.D. za jeho ochotu, vstřícnost, obětavost a neocenitelné rady k této práci. Také bych rád poděkoval Heleně Javorové za její morální i psychickou podporu.

Obsah

Kapitola 1

Úvod

V dnešní moderní době se lidstvo obklopuje technologiemi, kterým plně nerozumí ani jejich vlastní tvůrci. To může vést k vážným problémům. Nemůžeme plně zaručit, že elektrárna bude fungovat správně, že semafory dokážou předcházet dopravním nehodám, že letouny zůstanou ve vzduchu za všech možných podmínek a nebo že televizor skutečně reaguje na tlačítko zmáčknuté na televizním ovladači.

Je paradoxní, že právě technologie částečně zmírňují tento problém. Vytvořením abstrakce systémů, které nás obklopují, můžeme použít počítače k ověření, že funkce složitých systémů fungují řádně v souladu s jejich specifikací. Tento přístup se nazývá ověřování modelu (*model checking*). V ověřování modelu jsou systémy jako elektrárny, letadla, semafory a televize modelovány jako množiny možných stavů systému, kterých může nabývat, a množin přechodů mezi těmito stavy. Kromě toho existuje jeden nebo více počátečních stavů, které popisují stavy, ve kterých by systém mohl začínat. Stavy a přechody tvoří přechodový systém, který popisuje systémové chování. Formální logiky jako jsou CTL (což je anglická zkratka pro Computational Tree Logic) a lineární temporální logika (zkráceně LTL), mohou být pak použity k formální specifikaci vlastností jako formulí. Například vlastnost, že systém je bez uváznutí a žádný zakázaný stav nemůže být dosažen. Takto vyjádřené vlastnosti je potom možné verifikovat metodami ověřování modelu.

Jádro ověřování modelu je založeno na algoritmu dosažitelnosti, který vypočítá všechny možné dosažitelné stavy systémů na základě počátečních stavů a přechodů. Použitím tohoto algoritmu můžeme vypočítat všechny stavy s určitými vlastnostmi, které jsou dosažitelné z počátečních stavů za účelem zjištění, zda systém dodržuje CTL nebo LTL formule. Jedním z problémů ověřování modelu je velikost přechodového systému. I s malými systémy jsou nároky na výpočetní výkon a paměť k uložení všech zkoumaných stavů obrovské. Jeden způsob jak se vypořádat s tímto problémem je, že nebudeme ukládat každý stav samostatně, ale reprezentovat všechny stavy pomocí logických funkcí. To se nazývá symbolické ověřování modelu.

Booleova algebra je základním kamenem počítačové vědy a digitálního návrhu systému. Mnoho problémů v návrhu číslicových obvodů, umělé inteligenci, kombinatorice atd. může být vyjádřeno jako posloupnost operací prováděny pomocí množiny Booleovských funkcí $f : B^k \to B$, kde $B = \{0, 1\}$ a k je celé nezáporné číslo. Takové aplikace by měly značný prospěch z algoritmů, které by zvládaly jak efektivní symbolickou reprezentaci, tak efektivní manipulaci Booleovských funkcí. Bohužel některé operace nad Booleovskými výrazy, jako například splnitelnost, ekvivalence, tautologie, jsou NP-úplný nebo coNP-úplný problém. To znamená, že řešení založená na klasické reprezentaci (Karnaughovy mapy, pravdivostní tabulky a jiné) vyžadují čas, který roste v nejhorším případě exponenciálně v závislosti na

počtu proměnných.

Jednou z hlavních nevýhod většiny existujících metod pro reprezentaci Booleovských funkcí je, že neposkytují kanonickou formu, tj., daná funkce může mít více rozdílných reprezentací. Z toho vyplývá, že testování ekvivalence či splnitelnosti může být velmi náročné.

Proto byly vyvinuty binární rozhodovací diagramy, zkráceně BDD (binary decision diagram), které poskytují několik výhod oproti klasickým reprezentacím. Umožňují velice úsporně a kanonicky kódovat Booleovské funkce způsobem, který je velmi vhodný pro efektivní provádění provádění operací a testů. V mnoha praktických aplikacích umožňuje BDD kódování vyhnout se nejhorší možné exponenciální časové složitosti.

Z původní aplikační domény, ověřování modelu, se BDD rozšířila do mnoha oblastí výpočetní techniky. Můžeme jmenovat například symbolickou verifikaci, syntézu hardware, diagnostiku, rozhodovací procedury logik. V kapitole 2.2 jsou některé tyto aplikace rozebrány podrobněji.

Booleovské funkce jsou obvykle uloženy v paměti pomocí BDD, které jsou uloženy v paměti použitím hašovacích tabulek. Aby bylo možné manipulovat s Booleovskými funkcemi uloženými pomocí BDD, tak existuje několik operací, které umožňují manipulaci s BDD. Jenom čtyři operace jsou zapotřebí k výpočtu všech možných dosažitelných stavů a to výpočet $\wedge$, výpočet $\exists$, výpočet substituce a výpočet $\vee$. V běžných BDD implementacích existuje speciální algoritmus k výpočtu *relačního produktu*, který v sobě kombinuje $\wedge$ a $\exists$.

Cílem této práce je navrhnout a implementovat knihovnu pro binární rozhodovací diagramy, která by obsahovala všechny operace, které jsou zapotřebí nejen pro symbolické ověřování modelu, ale také pro další aplikace, které by využívaly manipulaci s Booleovskými funkcemi. Zároveň by tato knihovna měla být výkonnostně srovnatelná s již existujícími knihovnami, které jsou volně dostupné. Proto je podstatnou částí této práce studium existujících metod pro efektivní reprezentaci a manipulaci s BDD.

Tato práce je strukturovaná následovně. Kapitola 2.1 obsahuje úvod do binárních rozhodovacích diagramů. Jsou zde popsány základní pojmy vztahující se ke stromovým automatům. Dále je zde popsáno použití BDD a na závěr této kapitoly jsou popsány existující knihovny. Metody pro efektivní reprezentaci BDD jsou popsány v kapitole 3. V kapitole 4 je stručně popsána knihovna VATA a možné použití chystané BDD knihovny ve VATA. Kapitola 5 obsahuje návrh knihovny pro BDD. Implementace BDD knihovny je stručně rozebrána v kapitole 6. Kapitola 7 je věnována experimentům s implementovanou BDD knihovnou. Na závěr kapitola 8 obsahuje shrnutí výsledků experimentů a další možný vývoj implementované BDD knihovny.

Kapitola 2

Teoretický úvod

Tato kapitola je rozdělena na několik sekcí. V první sekci jsou teoreticky popsány BDD. Druhá sekce popisuje různé použití BDD a další sekce se zabývá již existujícími knihovnami, které taktéž implementují BDD. Poslední sekce uvádí lehký přehled do stromových automatů.

2.1 Úvod do Binárních rozhodovacích diagramů

Tato sekce představuje úvod do BDD. Jsou zde uvedeny základní definice vycházející především z [5, 10, 24, 2]. V první části nejdříve definujeme, co je to *if-then-else* normální forma. Poté jsou popsány základní definice BDD, co to je BDD, jak je reprezentováno a jak reprezentuje Booleovské funkce. Následuje popis omezení na BDD, které se nazývá *ordered BDD* neboli uspořádané BDD. Je zde také nastíněn problém přeuspořádání. Na závěr této kapitoly jsou popsány metody pro redukci redundantních uzlů pro uspořádané BDD a z toho vycházející definice pro redukované uspořádané BDD (*reduced ordered* BDD).

2.1.1 *If-then-else* normální forma

Nechť $x \to y_0, y_1$ je *if-then-else* operátor definovaný jako

$$x \to y_1, y_0 = (x \wedge y_1) \vee (\overline{x} \wedge y_0).$$

Výraz $x \to y_0, y_1$ je pravdivý, když x a y_0 jsou pravdivé, nebo jestliže x je nepravdivé a y_1 je pravdivé. Výraz x budeme nazývat *testovací výraz*. Všechny logické operace mohou být vyjádřeny použitím if-then-else operace a konstant 0 a 1. Kromě toho to lze provést takovým způsobem, aby se všechny testy provedly pouze na (ne negovaných) proměnných a proměnné se nevyskytovaly na žádném jiném místě v operaci. Proto tento operátor dává vzniknout novému druhu normální formy.

Definice 2.1. *Normální forma if-then-else* (INF) je Booleovský výraz postavený výhradně z operátoru if-then-else a konstant 0 a 1 takový, že všechny testy jsou prováděny pouze na proměnných.

Věta 2.1. Jakýkoliv Booleovský výraz je ekvivalentní výrazu v INF.

Důkaz. Budeme-li označovat $t[0/x]$ Booleovským výrazem získaným nahrazením x za 0 v t, pak není těžké vidět, že následující ekvivalence platí:

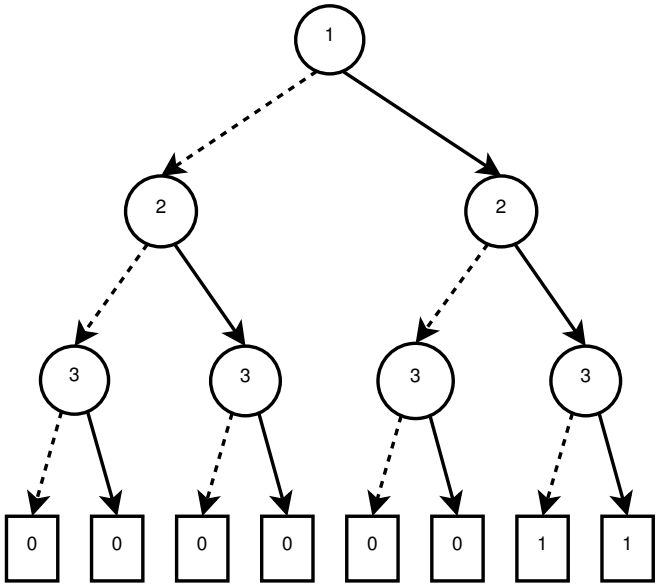
$$t = x \to t[1/x], t[0/x].$$

Toto je známé jako *Shannonův rozvoj* výrazu t s ohledem na proměnnou x. Tato jednoduchá rovnice generuje INF z jakéhokoliv výrazu t. Jestliže t neobsahuje proměnné, pak je ekvivalentní 1 nebo 0, což je INF. V opačném případě provedeme Shannonův rozvoj na t s ohledem na jednu z proměnných x ve výrazu t. Proto, když $t[0/x]$ a $t[1/x]$ oba obsahují nejméně jednu proměnnou, můžeme rekurzivně najít INF. INF pro t je tedy

$$x \to t_1, t_0.$$

2.1.2 Binární rozhodovací diagramy

Binarni rozhodovaci diagram (BDD) má podobu kořenového, orientovaného, acyklického grafu. Jeho množina vrcholů V obsahuje dva typy uzlů. Prvním typem uzlu je *neterminální* uzel v, který má jako atribut index $index(v) \in \{1, ..., n\}$, a který má dva následníky. Následníci jsou spojení s uzlem v buď přes nízkou a nebo vysokou větev. Pokud je následník spojen přes nízkou větev, potom bude takový následník označován jako $low(v)$. Podobně pokud je následník spojen přes vysokou větev, pak bude označován jako $high(v)$. Pro oba následníky musí platit $low(v), high(v) \in V$. Druhým typem uzlu je *terminální* uzel t, který má jeden atribut a to hodnotu $value(t) \in \{0, 1\}$. Příklad takového grafu je ukázán na Obrázku 2.1.



Obrázek 2.1: BDD představuje Booleovskou funkci $f(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \overline{x_3})$. Každá vrstva neterminálních uzlů představuje jeden argument z funkce $f(x_1, x_2, x_3)$. Neterminální uzly jsou reprezentovány kruhem s jejich hodnotou *index* a terminální uzly jsou reprezentovány čtvercem s hodnotou *value*. Nízké hrany jsou označeny čárkovanou čárou a vysoké hrany jsou označeny plnou čárou.

Z výše uvedeného popisu můžeme BDD formálně definovat následovně

Definice 2.2. BDD G je 7-tice $G = (N, T, index, low, high, root, value)$, kde:

- N je konečná množina neterminálních uzlů.
- T je konečná množina terminálních uzlů, $N \cap T = \emptyset$.

- $index : V \rightarrow \{1, ..., n\}$ je označení neterminálního uzlu podle argumentu funkce $f(x_1, ..., x_n)$.
- $low, high : N \rightarrow N \cup T$.
- $root \in N \cup T$ je kořenový uzel grafu G.
- $value : T \rightarrow \{0, 1\}$ označuje hodnotu terminálního uzlu.

Posloupnost hodnot argumentů $x_1, ..., x_n$ můžeme vidět jako cestu v grafu začínající od kořene. Pokud uzel v v cestě je neterminální a má $index(v) = i$ a $i > 0$, pak buď $x_i = 0$, potom cesta pokračuje přes následníka $low(v)$ nebo $x_i = 1$, potom cesta pokračuje přes následníka $high(v)$. Hodnota funkce pro tyto argumenty je rovna hodnotě terminálního uzlu na konci cesty. Cesta definovaná posloupností hodnot argumentu je vždy unikátní. Proto musí platit, že každý uzel stromu je obsažen v nejméně jedné cestě, tj. neexistuje žádný nedosažitelný uzel.

Formálně můžeme tedy definovat vztah mezi BDD a Booleovskou funkcí následovně

Definice 2.3. Uzel $v \in N \cup T$ BDD $G = (N, T, index, low, high, root, value)$ značící funkci $f_v : \{0, 1\}^n \rightarrow \{0, 1\}$, může být definován jako

- Jestliže $v \in T$, potom $f_v(x_1, ..., x_n) = value(v)$.
- Jestliže uzel $v \in N$ a $index(v) = i$, potom $f_v(x_1, ..., x_n) = x_i \cdot f_{high(v)}(x_1, ..., x_n) + \overline{x_i} \cdot f_{low(v)}(x_1, ..., x_n)$.

Jak normální forma INF popsaná v sekci 2.1.1, tak i BDD popsané v této sekci se mohou použít pro reprezentaci stejné Booleovské funkce. Proto musí existovat převod mezi INF a BDD, čehož využijeme v kapitole 3. Převod je jednoduchý a vypadá následovně. Každý podvýraz může být zobrazen jako uzel v grafu. Takový uzel je buď terminální v případě, že se jedná o konstantu 0 nebo 1, a nebo neterminální. Nízká hrana neterminálního uzlu odpovídá else části a vysoká hrana odpovídá then části.

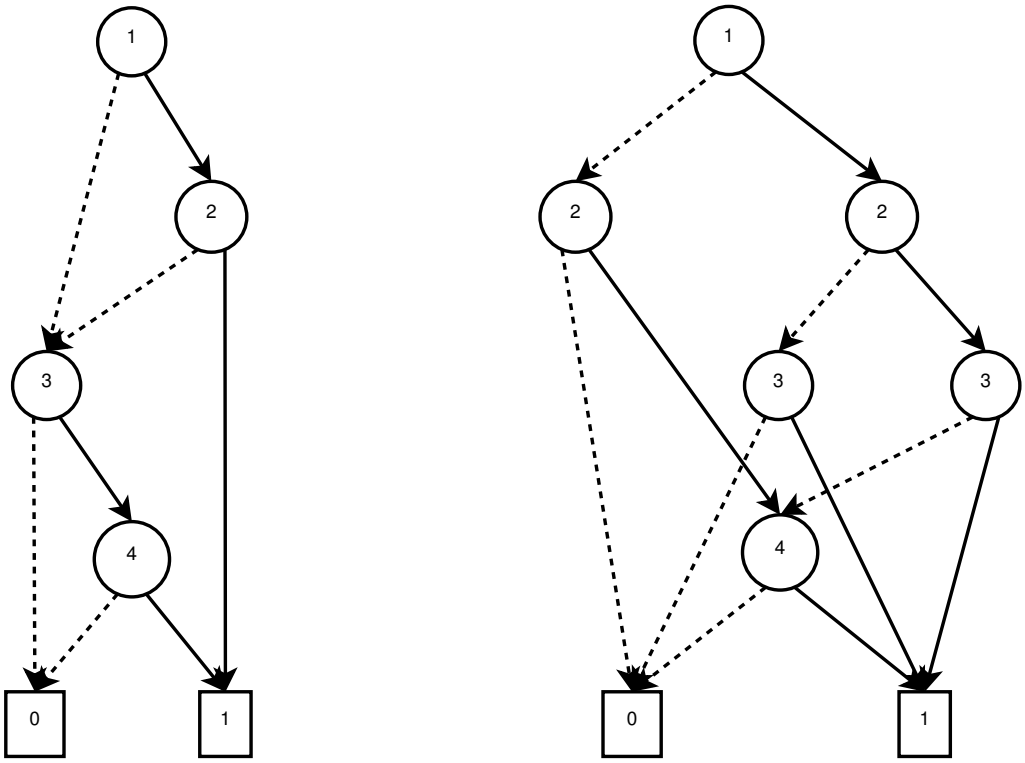
2.1.3 Uspořádané binární rozhodovací diagramy

Pokud je následník $low(v)$ neterminálního uzlu v taktéž neterminálním uzlem, potom musí platit následující omezení $index(v) < index(low(v))$. Podobné omezení
$index(v) < index(high(v))$ musí platit i v případě, pokud následník $high(v)$ je také neterminálním uzlem. Z výše uvedených omezení pro neterminální uzly vyplývá, že graf musí být acyklický, protože hodnota indexů neterminálních uzlů pro jakoukoliv cestu musí být striktně rostoucí. BDD splňující omezení, uvedená v tomto odstavci, budeme nazývat uspořádaným binárním rozhodovacím diagramem a budeme ho označovat anglickou zkratkou OBDD (ordered BDD). Ukázka OBDD je ukázána na Obrázek 2.1, kde platí následující uspořádání $x_1 < x_2 < x_3$.

Z definice 2.3 můžeme vidět, že rozklad funkce f_v může být proveden různým způsobem v závislosti na vybraných proměnných x_i. Z toho důvodu můžeme dostat různé OBDD pro stejnou Booleovskou funkci, když použijeme různé uspořádání proměnných. Následující příklad (Obrázek 2.2) ukazuje extrémní případ, jak uspořádání proměnných může působit na velikost OBDD. Mějme Booleovskou funkci

$$f(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 + x_3 \cdot x_4.$$

Taková funkce, jejichž uspořádání proměnných je $x_1 < x_2 < x_3 < x_4$, je reprezentována OBDD, které je tvořeno 6 uzly. Když ovšem změníme uspořádání na následující $x_1 < x_3 < x_2 < x_4$, tak počet uzlů vzroste na 8.



Obrázek 2.2: Ukázka dvou OBDD reprezentující stejnou funkci $f(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 + x_3 \cdot x_4$ s rozdílným uspořádáním proměnných. Vlevo OBDD s uspořádáním $x_1 < x_2 < x_3 < x_4$, vpravo OBDD s uspořádáním $x_1 < x_3 < x_2 < x_4$.

Když to zobecníme k funkcím s $2n$ argumenty, tak funkci $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$ představuje BDD s $2n + 2$ uzly, zatímco funkce $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$ představuje BDD s 2^{n+1} uzly. Z toho vyplývá, že špatně zvolené počáteční uspořádání argumentů může mít velmi nežádoucí následky. V nejhorším případě může velikost grafu růst exponenciálně v závislosti na počtu argumentů.

Bohužel nalézt vhodné uspořádání je NP-těžký problém. Proto se používají buď různé heuristiky k určení vhodného uspořádání proměnných a nebo má uživatel znalost o tom, proč pro některé Booleovské funkce je reprezentující graf obrovský a jak vybrat vstupní uspořádání tak, aby ovlivnil tuto velikost. Jednou z těchto heuristik může být vkládání poblíž sebe proměnné, které spolu úzce souvisejí. Například tak, že se hodnota jedné z těchto proměnných počítá jako výstup druhé proměnné a nebo obě proměnné jsou současně vstupem pro jiné vyhodnocení. Jiná možnost je takzvané dynamické přeuspořádání, které prezentoval Rudell [19]. Podrobněji se touto problematikou zabývá například práce [10].

Za zmínku stojí, že existují takové Booleovské funkce, které nemohou být efektivně reprezentovány BDD bez ohledu na uspořádání.

Jak již bylo zmíněno výše, tak nalézt vhodné uspořádání je NP-těžké a používané přístupy jsou výpočetně náročné a dobrý výsledek není zaručen [25]. V důsledku toho se stále častěji používá fixní uspořádání, které vychází z uživatelovi znalosti struktury dat. Knihovna CacBDD [6] neposkytuje žádnou možnost přeuspořádání a CUDD [21] má tuto

možnost standardně vypnutou.

2.1.4 Izomorfní zobrazení

Dvě BDD jsou považovány za izomorfní, jestliže se shodují jak ve struktuře grafu, tak v atributech.

Definice 2.4. Nechť BDD $G = (N_1, T_1, var_1, index_1, low_1, high_1, root_1, value_1)$ a $G' = (N_2, T_2, var_2, index_2, low_2, high_2, root_2, value_)$ jsou izomorfní, jestliže existuje zobrazení σ takové, že pro jakýkoliv uzel v platí $\sigma(v) = v'$, kde $v' \in G'$. Potom buď jsou oba uzly v a v' terminální, a musí splňovat $value(v) = value(v')$ a nebo oba uzly jsou neterminální, a splňují následující omezení $index(v) = index(v') \wedge \sigma(low(v)) = low(v') \wedge \sigma(high(v)) = high(v')$.

Formální popis 2.4 můžeme rozepsat následovně. Vzhledem k tomu, že graf obsahuje pouze jeden kořen a následník jakéhokoliv neterminálního uzlu se liší, tak izomorfní zobrazení σ mezi grafy G a G' je poměrně omezené. Kořenový uzel z G se musí zobrazit na kořenový uzel z G', nízký následník kořene z G se musí zobrazit na nízkého následníka kořene z G' a tak pokračujeme až k terminálním uzlům. Proto testování dvou grafů na izomorfismus je poměrně jednoduché (lze provést v čase lineárním k počtu uzlů grafu).

2.1.5 Redukované uspořádané binární rozhodovací diagramy

OBDD může být redukováno v rámci počtu uzlů beze změny funkčnosti, odstraněním nadbytečných vrcholů a duplicitních podgrafů. Výsledné OBDD budeme nazývat redukované OBDD (ROBDD) a bude to primární datová struktura pro reprezentaci Booleovské funkce.

Redukce OBDD je založena pouze na dvou redukčních pravidlech. Podle [10] jsou to pravidla *S-redukce* (jako Shannonové redukční pravidlo) a *merging*. Hlavní myšlenkou redukčních pravidel je odstranění redundantních uzlů z OBDD a tím zmenšení velikosti a tudíž paměťových nároků. Mimoto odstraněním nadbytečných uzlů se sníží počet testování a kontroly, zda některé Booleovské funkce jsou reprezentovány víc než jednou.

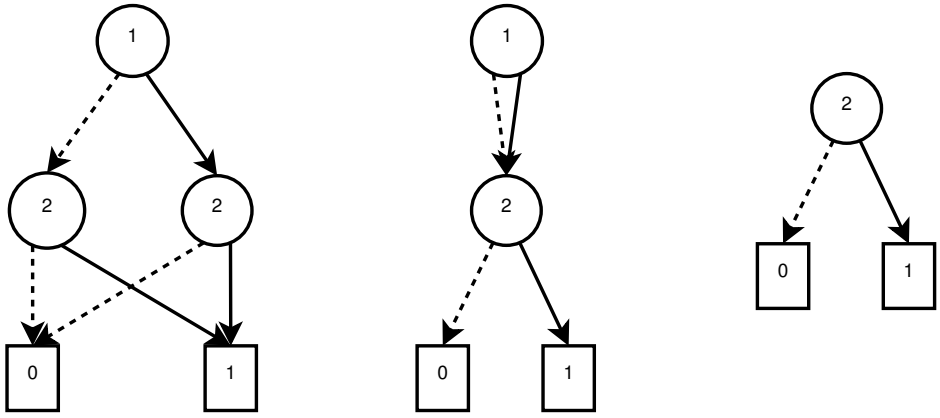
Pravidlo s-redukce může být aplikováno na uzel v, jehož obě hrany vedou do stejného uzlu w neboli $low(v) = high(v)$. Je zřejmé, že můžeme přesměrovat všechny hrany vedoucí do uzlu v, do uzlu w. Uzel v se potom stává nadbytečným, a proto ho můžeme odstranit bez toho abychom ovlivnili reprezentaci Booleovské funkce.

Pravidlo merging je aplikovatelné, jestliže existují dva uzly v a w takové, že jejich podgrafy, které mají kořeny v uzlech v a w, jsou izomorfní. Znovu je zřejmé, že můžeme přesměrovat všechny hrany vedoucí do uzlu v, do uzlu w, a že můžeme odstranit uzel v beze změny reprezentace funkce. Mimoto tato operace může být provedena i na terminálních uzlech.

Definice 2.5. OBDD G je redukované, jestliže neobsahuje žádný uzel $v \in N$ takový, že $low(v) = high(v)$ a ani neobsahuje žádné dva uzly $v, w \in N \cup T \wedge v \neq w$, jejichž podgrafy s kořeny v uzlech v a w jsou izomorfní.

Následující věta vyplývá přímo z definice 2.5.

Lemma 2.1. Nechť G je ROBDD. Pro každý uzel $v \in G$ platí, že podgraf s kořenem ve v je také ROBDD.



Obrázek 2.3: Ukázka použití redukčních pravidel. Vlevo je OBDD, které není redukované. Uprostřed se nachází OBDD po použití pravidla merging a po použití pravidla s-redukce dostaneme ROBDD, což ukazuje obrázek vpravo.

Věta 2.2. Pro jakoukoliv Booleovskou funkci f s uspořádáním proměnných π, zde existuje unikátní (až na izomorfismus) ROBDD popisující funkci f. Jakékoliv jiné ROBDD popisující funkci f obsahuje alespoň o jeden uzel navíc.

Věta 2.2 dokazuje klíčovou vlastnost ROBDD a sice, že jsou kanonickou formou pro Booleovské funkce. Kanonická forma znamená, že pro všechny Booleovské funkce a jejich uspořádání proměnných existuje jednoznačná reprezentace. Bezprostředním důsledkem je tedy následující. Terminální uzel s hodnotou 1 je jediný ROBDD, který je stále pravdivý pro jakékoliv uspořádání proměnných. Abychom tedy zkontrolovali, zda ROBDD je stále pravdivé, tak stačí pouze zkontrolovat, jestli testované ROBDD není terminální uzel s hodnotou 1. Taková operace jistě trvá konstantní čas. Podobně můžeme zjistit, zda je ROBDD stále nepravdivé, tj. musí být identické s terminálním uzlem, který má hodnotu 0. V podstatě, aby se zjistilo, jestli jsou dvě funkce stejné, tak stačí vytvořit jejich ROBDD a zkontrolovat, zda výsledné uzly jsou stejné. Důkaz věty 2.2 je představen v [5].

2.2 Reálné použití binárních rozhodovacích diagramů

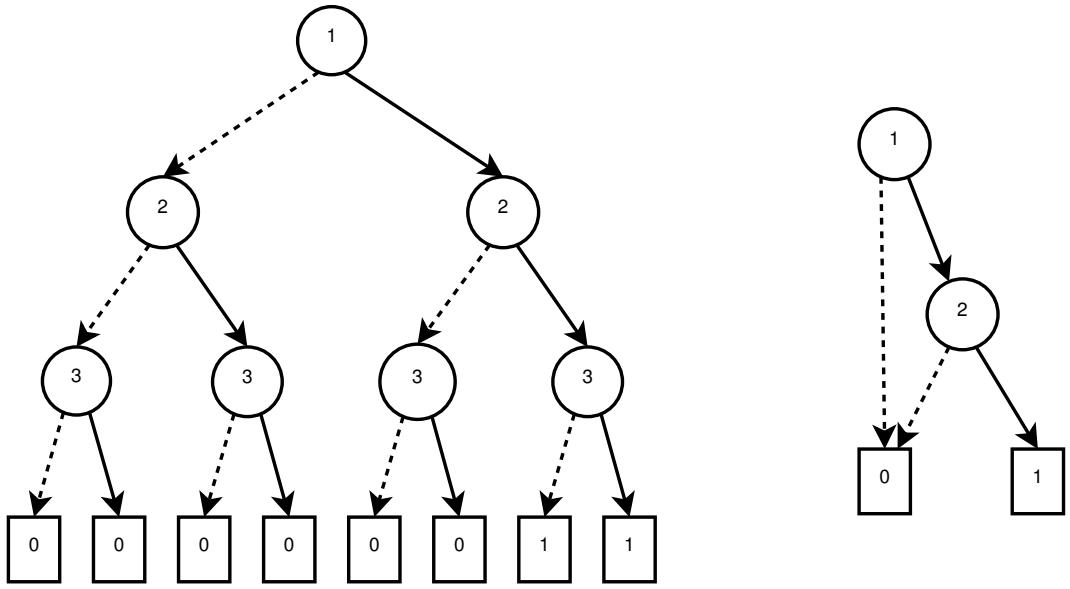
Tato sekce popisuje použití BDD v různých počítačových disciplínách.

2.2.1 Ověřování modelu

Využití BDD v ověřování modelu vychází z práce [9]. Ověřování modelu je metoda k ověřování vlastnosti systému, protokolů, programů atd., podle manuálního upřesnění nebo automaticky generovaného abstraktního modelu systému. Tento abstraktní model může pak být použit k ověření, že jisté vlastnosti jsou dodrženy ve všech možných dosažitelných stavech systému.

Nejdříve definujeme, co je to model. Model je trojice $(X, S_{initial}, R)$, kde X je množina pojmenovaných proměnných, $S_{initial} \subset S$ je množina počátečních stavů a R je přechodová relace definující všechny možné přechody v systému.

Předpokládejme nějakou množinu $X = \{x_1, \ldots, x_n\}$ pojmenovaných proměnných. Stav $s : X \to \{0, 1\}$ je vyhodnocení všech proměnných v X. Například $s(x_1) = 1$ a $s(x_2) = 0$. Takový stav označíme jako $x_1\overline{x_2}$. Nechť S bude množina všech stavů tj. všech možných



Obrázek 2.4: Demonstrace významu zavedení ROBDD. Obě BDD představují Booleovskou funkci $f(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \overline{x_3})$. Vlevo je OBDD a vpravo je redukované OBDD.

vyhodnocení proměnných v X. Podmnožina $V \subset S$ může být označena Booleovskou funkcí $F : S \rightarrow \{0, 1\}$ indikující, které stavy jsou v množině. Například můžeme definovat $F = \{s | \neg s(x_1)\}$, což je množina všech stavů, která přiřazuje proměnné x_1 hodnotu 1. Pro jednodušší označení budeme takovou funkci značit jako $F(s) = \overline{x_1}$.

Přechodová relace je relace $R \subseteq S \times S$. Můžeme znovu použít označení pomocí Booleovské funkce $T : S \times S \rightarrow \{0, 1\}$. Například $T(s, s') = \{(s, s') | s(x_1) \wedge \neg s(x_2) \wedge s'(x_2)\}$. Tato relace definuje dva přechody: první mezi stavem $x_1\overline{x_2}$ a x_1x_2, druhý mezi $x_1\overline{x_2}$ a $\overline{x_1}x_2$. Pro snadnější zápis znovu zkrátíme $s(x_i)$ na x_i a $\neg s(x_i)$ na $\overline{x_i}$. To samé provedeme pro s'. Takže pro příklad výše dostáváme $T(s, s') = x_0x_1x'_1$.

Jak již bylo psáno v úvodu 1, tak BDD lze použít pro efektivní reprezentaci Boolovských funkcí. V předchozích odstavcích bylo ukázáno, že jak množinu stavů, tak množinu přechodů lze zakódovat do Booleovské formule. Složitější systémy mohou mít velkou množinu stavů a relací mezi nimi a proto je vhodné použít BDD pro symbolickou reprezentaci takových systémů.

2.2.2 Evoluční návrh obvodu

Využití BDD v evolučním návrhu obvodu vychází z práce [22]. Zde používají kartézské genetické programování k vytvoření návrhu obvodu. Jako u většiny genetických algoritmů bývá problém, jak správně zakódovat obvod a jak vytvořit fitness funkci. Proto zmiňovaná práce navrhuje fitness funkci transformující každý kandidátní obvod na odpovídající BDD. Navržená fitness funkce je vložena do kartézského genetického programování, což je používána metoda pro vývoj číslicových obvodů.

V evolučním algoritmu se vyžaduje fitness funkce, která má bohatý rozsah hodnot za účelem rozlišení malých hodnot mezi podobnými řešeními. Namísto použití *Sat-One* funkce (tj. nalezení takového cesty v BDD, jejíž terminální uzel má hodnotu 1) je navrženo použití *Sat-Count* funkce (funkce vrací mohutnost množiny takové, která odpovídá množině cest,

jejichž terminální uzel má hodnotu 1) na každý výstup z_i a spočítat všechny výsledky. Výsledná hodnota představuje Hammingovu vzdálenost mezi dvěma obvody.

V kontrastu k vyhodnocení odpovědi obvodu pro všechny vstupní kombinace, což představuje typický přístup v oblasti evolučního návrhu obvodu, metoda založená na ROBDD nám umožňuje efektivně porovnat podobnost dvou obvodu i v případě složitých obvodů. Podobnost mezi dvěma obvody vyjádřena jako Hammingova vzdálenost, může být spočítána v lineárním čase s ohledem na velikost ROBDD reprezentující tento obvod, což je výrazné zlepšení oproti exponenciálnímu přístupu.

Fitness funkce funguje následovně. Nejprve je potřeba zkonstruovat ke kandidátnímu obvodu C odpovídající ROBDD D^C. Takové ROBDD se vytvoří postupným provedením logických funkcí (hradla), kterými je obvod tvořen. Každá logická funkce vyžaduje dva operandy, kteří jsou interpretováni jako ukazatelé na odpovídající ROBDD uzly. V závislosti na logické funkci je jeden nebo více nových uzlů zahrnuty do D^C.

Prostřednictvím funkce *Sat-Count* je možné získat počet přiřazení b_i ze vstupů, které vyhodnotí z_i na 1. Nakonec fitness funkce f je definována jako

$$f = \sum_{i=1}^{n_o} b_i.$$

Získání $f = 0$ znamená, že bylo objeveno plně funkční řešení.

2.2.3 Symbolický automat

Teorie symbolických automatů obohacuje klasické automaty o abecedu. Činí tak tím, že nahradí explicitní abecedu abecedou popsanou implicitně Booleovou algebrou. V práci [23] se zabývají regulárními výrazy, zkráceně regex, popsanými konečnými symbolickými automaty (KSA) v souvislosti s analýzou programu.

Efektivní Booleovská algebra $\mathcal{A}$ má komponenty $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. $\mathfrak{D}$ je r.e. (rekurzivně spočetná) množina prvků domény. Ψ je r.e. množina predikátů uzavřených nad Booleovskýma spojkami (tj. může být tvořena pouze Booleovskýma spojkami) a $\bot, \top, \in \Psi$. Funkce označení $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ je r.e. a je taková, že $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, pro všechny $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, a $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. Pro $\varphi \in \Psi$, píšeme $IsSat(\varphi)$, když $[\![\varphi]\!] = \emptyset$ a říkáme, že φ je splnitelná. $\mathcal{A}$ je rozhodnutelná, jestliže $IsSat$ je rozhodnutelný.

Nyní můžeme definovat formálně, co je to KSA. *Symbolický konečný automat* M je pětice $M = (\mathcal{A}, Q, q^0, F, \Delta)$, kde $\mathcal{A}$ je efektivní Booleova algebra zvaná *abeceda*, Q je konečná množina stavů, $q^0 \in Q$ je počáteční stav, $F \subset Q$ je množina koncových stavů a $\Delta \subset Q \times \Psi_{\mathcal{A}} \times Q$ je konečná množina pohybů nebo přechodů.

Prvky $\mathfrak{D}_{\mathcal{A}}$ se nazývají znaky a konečná posloupnost znaků, prvky $\mathfrak{D}^*_{\mathcal{A}}$, se nazývají slova; ϵ značí prázdné slovo. Pohyb (přechod) $\rho = (p, \varphi, q) \in \Delta$ je také označován jako $p \xrightarrow{\varphi}_M q$ (nebo $p \xrightarrow{\varphi} q$, pokud je M jasné (tj. nemůže dojít k záměně)), kde p je *počáteční stav* značený $Src(\rho)$, q je cílový stav značený $Tgt(\rho)$ a φ je stráž nebo predikát pohybu (přechodu) značený $Grd(\rho)$. Pohyb (přechod) je *proveditelný* jestliže jeho stráž je splnitelná. Mějme znak $a \in \mathfrak{D}_{\mathcal{A}}$, *$a-$pohyb* (přechod) je pohyb (přechod) $p \xrightarrow{\varphi} q$ takový, že $a \in [\![\varphi]\!]$, také značený $p \xrightarrow{a}_M q$ (nebo $p \xrightarrow{a} q$, pokud je M jasné).

Jako abecedu pro KSA můžeme použít algebru 2^{bvk}. Doménou této algebry je konečná množina *bvk*, pro nějaké $k > 0$, složené ze všech nezáporných celých čísel menších jak 2^k, nebo rovno, všem k-bitům bitového vektoru. Predikát je reprezentován BDD s hloubkou

k. Boolovské operace korespondují přímo s operacemi nad BDD, $\perp$ je BDD reprezentací prázdné množiny. Označení $[\![\beta]\!]$ BDD β je množina všech celých čísel n takových, že binární reprezentace n odpovídá řešení β.

Zakódování regexu může vypadat následovně. Velikost abecedy je 2^{16} kvůli přijatému rozšíření UTF16 standardu znaků Unicodu. Nechť abeceda algebry je 2^{BV16}. Nechť BDD β_w^7 reprezentuje všechny ASCII slovní znaky (písmena, číslice, a podtržítko) jako množinu znaků zakódovaných {`'0'`, ..., `'9'`, `'A'`, ..., `'Z'`, `'_'`, `'a'`, ..., `'z'`}. (Píšeme `'0'` pro kód 48, `'a'` pro kód 97 apod.) Nechť také β_d^7 reprezentuje množinu všech decimálních číslic {`'0'`, ..., `'9'`} a nechť $\beta_\_$ reprezentuje podtržítko {`'_'`}. Použitím Booleovských operací, tj., $\beta_w^7 \wedge \neg(\beta_d^7 \vee \beta_\_)$ reprezentuje množinu velkých a malých ASCII písmen. Regexem to je vyjádřené jako `[\w-[\d_\x7F-\uFFF]]`.

2.3 Existující knihovny

Tato kapitola stručně popisuje knihovny CUDD [21] a CacBDD [6], které implementují BDD. Budou se zde vyskytovat pojmy, které budou podrobněji vysvětleny až v následujících kapitolách.

Kromě zmíněných dvou knihoven existují i jiné knihovny implementující BDD. Jako příklad můžeme uvést knihovnu CAL [20], což je veřejně dostupná BDD knihovna založená na prohledávání do šířky. Dále můžeme uvést knihovnu TiGeR [18], což je komerční knihovna založená stejně jako CUDD a CacBDD na prohledávání do hloubky.

2.3.1 Knihovna CUDD

Knihovna CUDD je veřejně dostupná BDD knihovna založená na prohledávání do hloubky, která je každoročně aktualizována. Poskytuje funkce pro manipulaci s BDD, s algebraickými rozhodovacími diagramy (ADD nebo také MTBDD, které jsou podrobněji popsány v kapitole 4) a nulu potlačující binární rozhodovací diagramy (ZDD). Knihovna obsahuje velkou sadu operací nad BDD, ADD, ZDD. Dále obsahuje funkce na převedení mezi jednotlivými reprezentacemi a velkou škálu metod pro přeuspořádání proměnných, které jsou standardně vypnuté.

Tato knihovna je implementována v jazyce C a používá ukazatele na uzly. CUDD používá u BDD uzlů *počitadlo referencí*, které udržuje během běhu stále aktuální. Unikátní tabulka je implementována menšími tabulkami, jejichž počet závisí na počtu proměnných. Tabulka výpočtů roste v závislosti na *hit-rate* tabulek. Knihovna používá heuristiku založenu na počítadle referencí, která zpřístupní tabulku výpočtů jenom tehdy, jestliže alespoň jeden argument má hodnotu počítadla referencí větší jak jedna. Garbage collection využívá počítadla referencí a funguje tak, že projde všechny uzly a pokud má počitadlo referencí hodnotu 0 (takový uzel se potom nazývá *mrtví*), tak je tento uzel smazán. CUDD má zajímavou strategii volání garbage collectionu. Pokud je dostatek volné paměti není volán, ale pokud dostatek volné paměti překročí určitou hranic, tak je volán častěji v závislosti na volné paměti.

2.3.2 Knihovna CacBDD

Knihovna CacBDD je veřejně dostupná BDD knihovna, která jak již bylo zmíněno výše, je založená na prohledávání do hloubky. Tato knihovna je zajímavá tím, že v práci [15] bylo ukázáno, že je rychlejší než knihovna CUDD. Poskytuje standardní operace pro manipulaci

s BDD. Je implementována v jazyce C++ a na rozdíl od knihovny CUDD používá místo ukazatelů na uzly indexy uzlů. To znamená, že BDD uzly jsou uloženy v jednom velkém poli a pro manipulaci s těmito uzly používá index uzlu v poli. Nepoužívá počitadlo referencí. Garbage collection je volán pouze v případě, když dojde volná paměť v systému. Garbage collection je implementován metodou *označ a ukliď* (mark-and-sweep), která prochází všechny BDD uzly a označí ty uzly na které již není odkazováno (tj. žádný BDD uzel na tento uzel již neukazuje a ani není kořenem). Takto označené uzly mohou být potom smazány, ale v případě CacBDD, který používá indexy, se uzly nemažou. Namísto toho jsou označené uzly znovu použity jako volné uzly (tj. uzel, který nenese žádnou informaci a je použit v případě potřeby nového uzlu), což šetří výpočetní čas. CacBDD používá jednu velkou unikátní tabulku. Dále tato knihovna používá dynamické zvětšování tabulky výpočtů (viz 3.5).

2.4 Stromové konečné automaty

V této sekci budou nejprve popsány základní pojmy týkající se konečných stromových automatů a následně stromové automaty. Tato sekce vychází z prací [12, 7]. Pojmy definované v této sekci budou dále použity v kapitole 4.

Stromové automaty zobecňují konečné automaty, místo slov akceptují množiny stromů. Stromové automaty mají řadu aplikací, od ve formální verifikace a rozhodovacích procedur logik, přes zpracování XML dokumentů, až například ke zpracovaní přirozeného jazyka. Zde se soustředíme na automaty s velkými abecedami. S těmi pracuje například rozhodovací procedura logiky WSKS [11], kde symboly abecedy jsou bitová slova délky n kódující hodnoty n proměnných, nebo ve stromovém regulárním ověřování modelu [3].

2.4.1 Termy

Ohodnocená abeceda je dvojice $(\mathcal{F}, Arity)$, kde $\mathcal{F}$ je konečná množina a $Arity$ je zobrazení z $\mathcal{F}$ do $\mathbb{N}$. Arita symbolu $f \in \mathcal{F}$ je $Arity(f)$. Množina symbolů arity p je značena jako $\mathcal{F}_p$. Prvky arity $0, 1, \ldots, p$ jsou nazývány konstanty, unární, ..., p-arní symboly. Předpokládejme, že $\mathcal{F}$ obsahuje nejméně jednu konstantu. Používáme závorky a čárky pro krátkou deklaraci symbolů s aritou. Například $f(,)$ je krátká deklarace pro binární symbol.

Nechť χ bude množina konstant zvaných proměnné. Předpokládejme, že množiny χ a $\mathcal{F}_0$ jsou disjunktní. Množina $(\mathcal{F}, \chi)$ termů nad ohodnocenou abecedou $\mathcal{F}$ a množinou proměnných χ je nejmenší množina definována jako:

- $F_0 \subset T(\mathcal{F}, \chi)$ a
- $\chi \subset T(\mathcal{F}, \chi)$ a
- jestliže $p \geq 1, f \in \mathcal{F}$ a $t_1, \ldots, t_p \in T(\mathcal{F}, \chi)$, pak $f(t_1, \ldots, t_p) \in T(\mathcal{F}, \chi)$.

Jestliže $\chi = \emptyset$, pak $T(\mathcal{F}, \chi)$ lze také zapsat jako $T(\mathcal{F})$. Termy v $T(\mathcal{F})$ jsou nazývány **uzavřené termy**. Term t v $T(\mathcal{F}, \chi)$ je lineární, jestliže každá proměnná se vyskytuje nejvýše jednou v termu t.

2.4.2 Stromy

Konečný uspořádaný **strom** t nad množinou označení E je zobrazení z prefixově uzavřené množiny $\mathcal{P}os(t) \subset \mathbb{N}^*$ do E. Proto term $t \in T(\mathcal{F}, \chi)$ může být považován za konečný uspořádaný ohodnocený strom. Listy jsou označeny proměnnými nebo konstantními symboly a

vnitřní uzly jsou označeny symboly pozitivní arity s počtem výstupních hran, který je roven aritě označení. Term $t \in T(\mathcal{F}, \chi)$ může být definován jako parciální funkce $t : N^* \to \mathcal{F} \cup \chi$ s doménou $\mathcal{P}os(t)$ splňující následující podmínky:

- $\mathcal{P}os(t)$ je neprázdná a prefixově uzavřená.
- $\forall p \in \mathcal{P}os(t)$, jestliže $t(p) \in \mathcal{F}_n, n \geq 1$, pak $\{j | pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$.
- $\forall p \in \mathcal{P}os(t)$, jestliže $t(p) \in \chi \cup \mathcal{F}_0$, pak $\{j | pj \in \mathcal{P}os(t)\} = \emptyset$.

Budeme zaměňovat označení termy a stromy. To je tím, že "strom"znamená vždy konečný uspořádaný ohodnocený strom splňující předchozí podmínky.

2.4.3 Nedeterministický konečný stromový automat

Konečný stromový automat (NKSA) nad $\mathcal{F}$ je pětice $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, kde Q je množina (unárních) stavů, $Q_f \subseteq Q$ je množina koncových stavů a Δ je množina pravidel přechodů následujícího typu:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)),$$

kde $n \geq 0, f \in \mathcal{F}_n, q, q_1, \ldots, q_n \in Q, x_1, \ldots, x_n \in \chi$.

Stromové automaty pracují na uzavřených termech nad $\mathcal{F}$. Automat začíná v listech a pohybuje se směrem nahoru. Neexistuje žádný počáteční stav v NKSA, ale když $n = 0$ tedy když symbol je konstantním symbolem a, přechodové pravidlo je ve tvaru $a \to q(a)$. Z tohoto důvodu přechodové pravidla pro konstantních symboly mohou být považovány za "počáteční pravidla".

Nechť $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ bude NKSA nad $\mathcal{F}$. **Pohybová relace** $\to_{\mathcal{A}}$ je definována jako: nechť $t, t' \in T(\mathcal{F} \cup Q)$,

$$t \to_{\mathcal{A}} t' \Leftrightarrow \begin{cases} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, \ldots, u_n))]. \end{cases}$$

$\overset{*}{\to}_{\mathcal{A}}$ je reflexivní a tranzitivní uzávěr $\to_{\mathcal{A}}$. Uzavřený term $t \in T(\mathcal{F})$ je přijat konečným stromovým automatem $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ jestliže

$$t \overset{*}{\to}_{\mathcal{A}} q(t)$$

pro nějaký stav $q \in Q_f$. Množina všech uzavřených termů přijatých NKSA $\mathcal{A}$ (*Jazyk* $\mathcal{A}$) je označován jako $\mathcal{L}(\mathcal{A})$. Množina uzavřených termů $\mathcal{L}$ je *regulární*, jestliže existuje takové NKSA $\mathcal{A}$ pro které $\mathcal{L} = \mathcal{L}(\mathcal{A})$. Jestliže dva (nebo víc) NKSA přijmou stejný stromový jazyk, pak jsou *ekvivalentní*. NKSA $\mathcal{A}$ je *úplný*, jestliže existuje alespoň jedno pravidlo

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta$$

pro všechny $n \geq 0, f \in \mathcal{F}_n$, a $q_1, \ldots, q_n \in Q$. Stav $q \in Q$ je *dostupný*, jestliže existuje uzavřený term t takový, že $t \overset{*}{\to}_{\mathcal{A}} q(t)$. NKSA je *redukovaný*, právě když všechny jeho stavy jsou dostupné.

Množina přechodových pravidel může být také definována jako množina pravidel v alternativním tvaru: $f(q_1, \ldots, q_n) \to q$. Pohybová relace může být definována jako předtím až na to, že na místo zachování struktury termu, NKSA nahradí podstromy jeho stavy. Term t je pak přijat NFTA $\mathcal{A}$, pokud

$$t \overset{*}{\to}_{\mathcal{A}} q,$$

kde $q \in Q_f$.

2.4.4 Deterministický konečný stromový automat

Nyní definujeme *deterministické konečné stromové automaty* (DKSA), které jsou speciálními případy NKSA. Stejně jako v případě konečných automatů, jakýkoliv jazyk přijatý NKSA může být také přijat DKSA.

Stromový automat $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ je deterministický (DKSA), pokud neexistují dvě pravidla se stejnou levou částí (a bez ϵ-pravidel). A DKSA je *jednoznačný*, to znamená, že existuje nejvýše jeden běh pro každý uzavřený term, tedy pro každý uzavřený term t existuje nanejvýš jeden stav q takový, že $t \xrightarrow{*}_{\mathcal{A}} q$

Věta 2.3. Nechť $\mathcal{L}$ je regulární množina uzavřených termů. Pak existuje DKSA takové, že přijme $\mathcal{L}$.

Kapitola 3

Analýza

Tato kapitola se věnuje analýze tradičních technik, které jsou použity pro efektivní implementaci BDD knihovny. Budeme zde uvažovat pouze použití ROBDD a Booleovské funkce budeme zkráceně nazývat funkce. Jsou zde popsány základní techniky unikátní tabulky, tabulky výpočtů a možná vylepšení těchto tabulek. Je zde také popsána technika negace hran, která snižuje nároky na paměť. Konec této kapitoly je věnován problematice *garbage collection*.

3.1 ITE operátor

If-then-else nebo také ITE operátor je jádrem většiny BDD knihoven. ITE je ternární Boolovská funkce definována pro tři vstupy F, G, H, které jsou vyhodnoceny: Jestliže F pak G jinak H.To je ekvivalentní s:

$$ite(F, G, H) = F \cdot G + \overline{F} \cdot H.$$

Jak si můžeme všimnout, tak ITE operace může být použita k implementaci všech binárních Boolovských operací, jak je popsáno v Tabulka 7.1 (nebo také jak popisuje předcházející kapitola). Protože ITE je logická funkce vykonávající se v každém uzlu BDD, tak je také efektivním stavebním blokem pro mnoho jiných operací nad BDD (viz následující kapitola).

3.2 Negované hrany

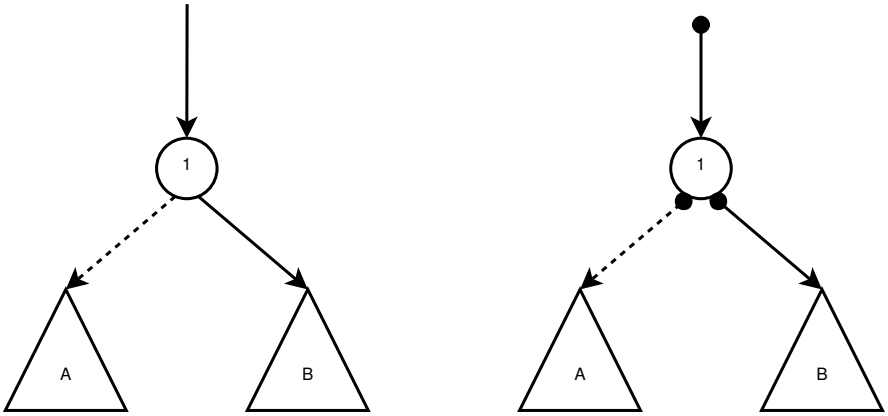
Negované hrany se používají ke snížení využití paměti a výpočetního času. Tuto techniku představil již Minato [16] a byla použita například v práci [4]. Tato práce ukázala, že použití negace hran může snížit velikost BDD až o 7% a jelikož je negace konstantní operace, tak také snižuje výrazně dobu běhu.

Uvažujme ROBDD G a ROBDD $\overline{G}$, které jsou si podobné až na to, že jejich terminální uzly 0 a 1 jsou zaměněny. Této podobnosti můžeme využít při zavedení negovaných hran. Negovaná hrana je obyčejná hrana s extra bitem (komplementární bit), který je nastavený na hodnotu 1 v případě, že funkce je interpretována jako negovaná funkce. Proto $\overline{G}$ může být reprezentováno G použitím negované hrany na uzel kořenový uzel ROBDD G se zachováním všech vnitřních uzlů.

Používáním negovaných hran ztrácíme vlastnost BDD, že jsou jedinečnou reprezentací Booleovské funkce, jak ukazuje Obrázek 3.1. Abychom zachovali kanonický tvar, musíme zavést několik omezení na používání negace hran. Minato [16] navrhl následující omezení:

Booleovský operátor	ITE
$F \wedge G$	$ite(F, G, 0)$
$F \vee G$	$ite(F, 1, G)$
$F \oplus G$	$ite(F, \overline{G}, G)$
$\neg(F \wedge G)$	$ite(F, \overline{G}, 0)$
$\neg(F \vee G)$	$ite(F, 0, \overline{G})$
$F \rightarrow G$	$ite(F, G, 1)$
$F \leftarrow G$	$ite(F, 1, \overline{G})$
$F \leftrightarrow G$	$ite(F, G, \overline{G})$
$\overline{F} \wedge G$	$ite(F, 0, G)$
$F \wedge \overline{G}$	$ite(F, \overline{G}, 0)$

Tabulka 3.1: Booleovské operace a jejich *ite* varianty.



Obrázek 3.1: ROBDD na obrázku reprezentují stejnou Booleovskou funkci $(x \wedge A) \vee (\overline{x} \wedge B) = \overline{(x \wedge \overline{A})} \vee (\overline{x} \wedge \overline{B})$, kde A a B jsou ROBDD reprezentující nějakou Booleovskou funkci.

1. Budeme používat pouze 0 jako hodnotu terminálního uzlu.
2. Nebudeme používat negaci na hranu vedoucí do nízkého následníka.

První omezení se může upravit tak, že místo používání terminálního uzlu s hodnotou 0 jako jediného terminálního uzlu budeme používat terminální uzel s hodnotou 1 a nebo nepovolíme negování hran vedoucí na terminální uzly. Alternativa pro druhé omezení je následující. Nebudeme používat negaci na hranu vedoucí do vysokého následníka.

Proto mohou být G a $\overline{G}$ reprezentovány stejným uzlem. Výhodami operace negace jsou snadná implementace a identifikace v konstantním čase.

3.3 Standardní trojice

Pro funkci a parametry $ite(F_1, F_2, F_3)$ mohou existovat parametry G_1, G_2, G_3 takové, že $ite(F_1, F_2, F_3) = ite(G_1, G_2, G_3)$, ale $F_i \neq G_i$ pro nějaké i. Definujme ekvivalenční vztah nad množinou tří funkcí F_1, F_2, F_3 založený na ekvivalenci Boolovské funkce $ITE(F_1, F_2, F_3)$.

Naším záměrem je vybrat standardní trojici z každé ekvivalenční množiny. Proto, když je volána metoda $ite(F_1, F_2, F_3)$, jsou před vyhledáním nebo vložením výsledku do tabulky výpočtů argumenty prvně nahrazeny vybranou standardní trojicí G_1, G_2, G_3. Toto vylepšení zefektivní tabulku výpočtů, jelikož zredukuje počet položek a zabrání znovu vypočítání výsledků, které jsou ekvivalentní.

Kvůli kanonické formě ROBDD a použití negovaných hran je možné rozpoznat, když dvě funkce jsou ekvivalentní nebo doplňkem v konstantním čase. Použitím pouze těchto dvou dotazů, lze snadno odhalit případ, kdy jsou dvě ekvivalentní Boolovské funkce vypočítávané. Následující příklad ukazuje ekvivalentní volání *ite* jsou pro $F + G$:

$$ite(F, F, G) = ite(F, 1, G) = ite(G, 1, F) = ite(G, G, F).$$

Vyberme standardní trojici z množiny následovně. Zaprvé pokud je to možné, aplikujeme následující zjednodušení na argumenty *ite*:

$$\begin{aligned} ite(F, F, G) &\Rightarrow ite(F, 1, G) \\ ite(F, G, F) &\Rightarrow ite(F, G, 0) \\ ite(F, G, \overline{F}) &\Rightarrow ite(F, G, 1) \\ ite(F, \overline{F}, G) &\Rightarrow ite(F, 0, G) \end{aligned}$$

Jak již bylo uvedeno kontrola, zda $F = G$ a $F = \overline{G}$ je konstantní časová operace. Dále zvažme následující ekvivalentní páry:

$$\begin{aligned} ite(F, 1, G) &= ite(G, 1, F) \\ ite(F, G, 0) &= ite(G, F, 0) \\ ite(F, G, 1) &= ite(\overline{G}, \overline{F}, 1) \\ ite(F, 0, G) &= ite(\overline{G}, 0, \overline{F}) \\ ite(F, G, \overline{G}) &= ite(G, F, \overline{F}) \end{aligned}$$

V případě zvolení unikátního prvku mezi $ite(F, 1, G)$ a $ite(G, 1, F)$ se vybere takový, jehož první argument *ite* reprezentuje menší proměnnou (viz předcházející kapitola). V případě nerozhodnutelnosti jsou formule uspořádány na základě jejich identifikátoru.

Od tohoto okamžiku zobecníme argumenty *ite* na F, G, H. Zavedením negovaných hran vede k následující ekvivalenci:

$$ite(F, G, H) = ite(\overline{F}, H, G) = \overline{ite(F, \overline{G}, \overline{H})} = \overline{ite(\overline{F}, \overline{H}, \overline{G})}.$$

Jedinečná trojice je vybrána z těchto čtyř tvarů podle následujícího pravidla: první a ani druhý argument *ite* by neměl být negovaný. Vzhledem k libovolné hodnotě F, G a H je tato podmínka splněna právě jednou pro výše uvedené tvary. V posledních dvou případech se výsledek získá negací funkce, a pak funkce bude znegována, než se vrátí.

Všimněte si, že tato pravidla efektivně detekují ekvivalenci podle DeMorganových pravidel. Například předpokládejme, že A a B nejsou negované a prvně vypočítáme $A+B$, což je $ite(A, 1, B)$. Jestliže později vypočítáme $\overline{A} \cdot \overline{B}$, což je $ite(\overline{A}, \overline{B}, 0)$, dostaneme $\overline{ite(A, 1, B)}$. Tabulka výpočtů již obsahuje výsledek, který potřebuje být pouze negován před tím, než je vrácen jako výsledek. Podobně můžeme zjistit, zda se chystá provést nadbytečný výpočet. Například $F + \overline{F} = ite(F, 1, \overline{F}) = ite(F, 1, 1) = 1$.

Kompletní množina terminálních případů je tedy: $ite(F, 1, 0) = ite(1, F, G) = ite(0, G, F) = ite(G, F, F) = F$ a $ite(F, 0, 1) = \overline{F}$.

3.4 Unikátní tabulka

Vzhledem k velkému počtu uzlů v BDD je důležité, abychom efektivně kontrolovaly tyto uzly. Místo, ve kterém jsou uloženy všechny BDD uzly, je označováno jako unikátní tabulka. Aby unikátní tabulka udržela silnou kanonickou formu (neexistují duplicitní uzly) uzlů v BDD, tak je obvykle implementována jako hašovací tabulka s kolizí vyřešenou řetězením, kde je každý uzel identifikován jedinečným identifikačním číslem. Jinými slovy, stejná Booleovská funkce může být vytvořena několika různými způsoby, ale bude zastoupena v unikátní tabulce pouze jednou. Proto, než se přidá nový uzel do BDD, tak se vyhledává v unikátní tabulce, zda uzel reprezentující Booleovskou funkci již existuje. Pokud ano, bude vrácen existující uzel. V opačném případě se nový uzel vloží do BDD a je vytvořen nový unikátní záznam tabulky. Unikátní tabulka umožňuje uložení více BDD současně.

Brace a spol. [4] navrhl, že namísto použití oddělených datových struktur pro unikátní tabulku a reprezentaci BDD, zkombinujeme unikátní tabulku a BDD do jedné datové struktury. Tudíž se každému uzlu přidá atribut, který bude reprezentovat kolizní řetězec. Jinými slovy budou uzly přímo prvky kolizního řetězce. Výhodou takové implementace je, že je jednodušší na implementování. Nevýhodou je, že hašovací tabulky vyžadují více paměťového místa, a proto může docházet k častějšímu přehašování.

V návrhu z [14] používají implementaci, která je popsána v předchozím odstavci, ale nepoužívají jednu unikátní tabulku. Místo toho používají několik menších hašovacích tabulek pro každou proměnnou, kterou uzel označuje. Každá tabulka má malou počáteční velikost (≤ 256). Když je překročen nastavený práh určité tabulky, tak se zvětší její velikost. Tímto způsobem lze ušetřit více místa a zlepšit využití.

Jiná implementace unikátní tabulky je popsána v Long [13]. Používají oddělené datové struktury jak pro unikátní tabulku, tak pro BDD. Uzel je v unikátní tabulce reprezentován odkazem. Dále navrhují několik možností, jak snížit *chache miss* v unikátní tabulce. První možností je udělat unikátní tabulku větší, ale následkem je plýtvání pamětí. Druhý způsob je založen na principu stáří uzlu. Stáří uzlu určují z předpokladu, že chronologické uspořádání věku odpovídá adresovému uspořádání. Protože halda roste směrem vzhůru, tak je víc přirozené, že starší uzly mají nižší adresy. Proto když známe věk uzlu, pak během operace hledání stačí pouze najít uzel v unikátní tabulce, který má stejný hašovací klíč. Protože unikátní tabulka řeší kolize kolizními řetězci, tak stačí porovnávat věk mezi uzly v řetězci a rozhodnout, zda takový uzel existuje nebo je potřeba ho vytvořit.

3.5 Tabulka výpočtů

Tabulka výpočtů se používá k zaznamenání všech předchozích výpočtů. To vede ke snížení množství nezbytných výpočtů při rekurzivním vykonávání funkce ite. Tabulka výpočtů je realizována jako hašovací tabulka, která mapuje trojici uzlů (F, G, H) na výsledek z $ite(F, G, H)$. Vyhledávání v tabulce výpočtů se provádí před vyhledáním v unikátní tabulce, aby se zkontrolovalo, zda výsledek již existuje. Tudíž pokud funkce $ite(F, G, H)$ již byla vypočtena a jestliže výsledky jsou uvedeny v tabulce výpočtů, může být výsledek vrácen namísto přepočítání operace. Je patrné, že používáním tabulky výpočtů se zlepší výkon BDD. Tabulka výpočtů se liší od unikátní tabulky ve způsobu řešení kolize. U tabulky výpočtů novější záznamy přepíšou ty starší, zatímco unikátní tabulka používá kolizní řetězec. Proto vyžaduje méně paměti, protože není nutné propojovat prvky mezi sebou v kolisním řetězci. Nejhorší případ složitosti operace ite je exponenciální a to v nepravděpodobném případě, že výstupem hašovací funkce pro všechny klíče je stejná hodnota. Některé knihovny používají

jedinou tabulku výpočtů a jiné zase používají samostatné tabulky výpočtů.

Velikost tabulky výpočtů má významný vliv na výpočet BDD. Řízení tabulky výpočtů je velmi důležité pro výkon BDD knihovny a jak najít dobrý algoritmus pro dynamickou správu tabulky výpočtů je problém. Brace a spol. [4] uvedli, že by bylo snadné řídit kompromis mezi pamětí a během, nastavením poměru mezi počtem položek unikátní tabulky a počtem položek tabulky výpočtů. Poměr výše uvedených dvou čísel se nazývá *hit-rate*. Nicméně, metoda je stále statická a předběžná a jednoduché řízení *hit-rate* nefunguje v mnoha případech.

Možné řešení tohoto problému je v práci [15] a idea vypadá následovně. Tabulka výpočtů se používá jako cache k vylepšení manipulace s BDD a jeho velikost je omezená dostupnou pamětí. Je to problém kompromisu mezi časem a prostorem. Protože *hit-rate* tabulky výpočtů je dynamický, tak velikost tabulky by měla být upravena dynamicky také. Proto pokud je nový *hit-rate* tabulky výpočtů větší než předešlý, pak by mělo být nezbytné rozšířit velikost tabulky výpočtů. Algoritmus 1 ukazuje dynamicky řízenou cache (tabulky výpočtů).

```
1:  ccc = ccc + 1;
2:  if (ccc ≥ occ) AND (cts < limitedValue) then
3:      if (cchr ≥ ochr) OR (cts < nc * cchr) then
4:          computed_table_increase_size();
5:      end
6:      ochr = cchr;
7:      occ = 2 * ccc;
8:  end
```

Algoritmus 1: Dynamicky řízena tabulka výpočtů.

Číslo *ccc* (počáteční hodnota je 0) je aktuální počítadlo spuštění tabulky výpočtů a číslo *occ* (počáteční hodnota je stejná jako počáteční velikosti tabulky výpočtů) slouží jako hranice pro spuštění algoritmu řízení cache. Číslo *cts* je velikost tabulky výpočtů a *nc* je počet uzlů. Číslo *cchr* je aktuální *hit-rate* tabulky výpočtu a *ochr* je poslední *hit-rate* tabulky výpočtů a jsou inicializovány na 0. Dále poznamenejme, že $limitedVaue$ je maximální hodnota velikosti, kterou může dosáhnout tabulka výpočtů a tu může určit uživatel nebo BDD knihovna podle dostupné paměti.

Algoritmus správy cache nastává po překročení *occ* během spuštění operace *ite*. Kód na řádku 4 zvýší velikost cache o dvojnásobek aktuální cache. Proto na řádku 7 je *occ* přiřazeno $2 * ccc$. Je zřejmé, že změnou konstanty 2 můžeme řídit frekvenci spuštění algoritmu změny velikosti cache.

V práci [25] nabízí heuristiku, která zpřístupní tabulku výpočtů jenom tehdy, jestliže alespoň jeden argument má hodnotu počítadla referencí větší jak jedna. Tato technika je založena na skutečnosti, že jestliže všechny argumenty mají hodnotu počítadla referencí jedna, pak je nepravděpodobné, že se daný podproblém bude opakovat v této operaci. Ve skutečnosti se tento podproblém nebude opakovat v rámci téže operace. Za použití této techniky jsme schopni snížit počet vyhledávání v tabulce výpočtů až o polovinu s celkovou redukcí času až o 40%.

3.6 Garbage collection

Brace a spol. [4] popsali garbage collection následovně. Každý uzel F má počítadlo odkazů. Toto počítadlo udržuje počet odkazů jak od ostatních uzlů, tak i od formulí, které ukazují na uzel F. Toto počítadlo se průběžně udržuje a odkazy z unikátní tabulky nebo z tabulky výsledků nejsou zahrnuty. Uzel, jehož počítadlo má hodnotu 0, se nazývá *mrtvý* uzel.

Když formule je uvolněna, počítadlo odkazů odpovídajícího uzlu $F = (v, G, H)$ je sníženo. Jestliže nová hodnota počítadla F je 0, pak počítadla odkazů jeho následníků budou rekurzivně snížena. F nemůže být ihned uvolněno, protože může být na něj odkazováno stále v tabulce výpočtů.

Jestliže vyhledání v tabulce výpočtů vrátí mrtvý uzel, tak se provede operace *reklamace*. Zvýší se počítadlo odkazů uzlu a všichni následníci kteří jsou mrtví, budou rekurzivně reklamováni. To přinese mrtvý uzel a všechny jeho mrtvé následníky zpět do BDD.

Počítání referencí se používá k udržení přesného počtu mrtvých uzlů v BDD. Počet mrtvých uzlů ovlivňuje strategii paměťového managementu. Jestliže je například 10% uzlů mrtvých, pak je vykonán garbage collection. Jestliže zde není dostatek mrtvých uzlů, pak je unikátní tabulce a tabulce výpočtů zvětšena velikost a všechny prvky jsou přehašované do větších tabulek. Garbage collection se provádí za velmi nízkou cenu v průběhu této změny velikosti.

Přetečení paměti je řešeno podobnou technikou. Když využití paměti u unikátní tabulky a tabulky výpočtu přesáhne uživatelem stanovený limit, a například 10% (záleží na uživateli) uzlů je mrtvých, garbage collection uvolní dostatek paměti pro pokračování. V opačném případě se balíček *vzdá* a vrací nulový ukazatel uživateli.

V práci [14] vylepšují garbage collection následovně. Pro zlepšení znovupoužitelnosti prostoru označíme mrtvé uzly namísto jejich uvolnění. Když je alokován nový uzel, program zkontroluje, zda je prostor a může být použit v závislosti na značce. Pokud ano, může tato metoda účinně zabránit neustálé alokaci a uvolnění. Tato metoda zabrání jak používání garbage collection příliš často, tak nedostatku paměti v závislosti na příliš mnoho mrtvých uzlů. *označ a ukliď* (mark-and-sweep) zlepšuje využití prostoru a je snadno ovladatelný.

Garbage collection je problém kompromisu mezi prostorem a časem. Garbage collection je obvykle spuštěn pouze na základě procenta mrtvých uzlů. Avšak vysoká míra znovuzrození znamená, že garbage collection by měl být odložen, dokud je to možné. Podle práce [15] bychom měli spouštět garbage collection pouze za předpokladu, že dostupná fyzická paměť je téměř vyčerpána. To zajistí, že se garbage collection spustí jen v případě, kdy je to opravdu potřeba.

V dnešní době se od statického určování počtu mrtvých uzlů spíše ustupuje, což dokazují knihovny CUDD (viz. 2.3.1) a CacBDD. Takový přístup byl dříve používán, neboť velikost pamětí byla malá, a proto bylo potřeba šetřit s místem. V dnešní době je velikost pamětí poměrně dostatečná, a proto se spíše vyplatí spouštět garbage collection, až když není dostatek dostupné paměti.

Kapitola 4

VATA

V této kapitole bude popsáno, jak funguje knihovna VATA a jak by šlo využít implementovanou BDD knihovnu v knihovně VATA. Tato kapitola vychází z [12].

Knihovna VATA je optimalizovaná pro nedeterministické konečné stromové automaty. Hlavním zaměřením knihovny je použití ve formální verifikaci, ale uplatnění může najít i v jiných oblastech. Existují dva podporované přístupy kódování stromových automatů, které knihovna podporuje a to *explicitní* a *semi-symbolicky. Semi symbolické* kódování používá *multiterminální binární rozhodovací diagramy* pro ukládání tabulky přechodů automatu. Je určen k použití pro automaty s velkými abecedy, které se objevují v několika technikách formální verifikace používající stromové automaty. Knihovna podporuje následující operace nad automaty: sjednocení, průnik, komplement, odstranění zbytečných stavů, odstranění nedosažitelných stavů, test inkluze.

4.1 Multiterminální binární rozhodovací diagramy

Symbolická reprezentace přechodové funkce automatu je realizována multitermálními binárními rozhodovacími diagramy (MTBDD), které jsou zobecněným ROBDD. Myšlenka zobecnění BDD na MTBDD je přiřazení více hodnot na koncové uzly diagramu (tedy zobecnění funkce f reprezentující BDD, $f : \{0,1\}^n \to \{0,1\}$ na funkci g reprezentující MTBDD, $g : \{0,1\}^n \to \mathbb{D}$), kde $\mathbb{D}$ je jakákoliv doména taková, že obsahuje *bottom* prvek $\bot \in \mathbb{D}$). Několik předpokladů musí být splněno na doméně $\mathbb{D}$, aby bylo možné mapování na MTBDD:

- produkt $x \in \{0,1\}$ a $d \in \mathbb{D}$ je definováno jako

$$x \cdot d \Leftrightarrow \begin{cases} \bot & \text{jestliže} \quad x = 0 \\ d & \text{jestliže} \quad x = 1 \end{cases},$$

- operace sčítání nad $\mathbb{D}$ musí zajistit, aby pro $d \in \mathbb{D}$ platilo

$$d + \bot = \bot + d = d.$$

Následně definujeme zobrazení z MTBDD $g : \{0,1\}^n \to \mathbb{D}$ na Booleovskou formuli

$$\sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} (\prod_{a_i=0} \neg x_i \cdot \prod_{a_i=1} x_i \cdot g(a_1,\ldots,a_n)).$$

Jsou-li MTBDD použity pro reprezentaci tabulky přechodů, tak je každému symbolu vstupní abecedy Σ přiřazen binární řetězec. Tedy existuje kódovací funkce $enc : \Sigma \to \{0,1\}^n$, kde $n = \lceil lg|\Sigma| \rceil$. Hodnoty koncových uzlů (množina $\mathbb{D}$ z již zmíněné funkce g) je množina stavů automatu. Takové MTBDD může být použito k popisu přechodové funkce automatu pro jednotlivé stavy: vstupní symbol je zakódován funkcí enc do posloupnosti binárních číslic $(x_1, \ldots, x_n)$, kde $x_1, \ldots, x_n$ odpovídá Booleovským proměnným MTBDD. Přiřazení k proměnným označuje cestu, která má být přijata v diagramu a určuje koncový uzel (tj. další stav automatu). Takové MTBDD může buď existovat pro každý stav automatu, nebo výhodnější možnost použití sdíleného MTBDD. To je další zobecnění, které sloučí všechny diagramy do jednoho diagramu s více kořenovými uzly (každý odpovídá jinému stavu) a mění stromovou strukturu do orientovaného acyklického grafu. Takový přístup poskytuje kompaktní reprezentaci přechodové funkce dokonce i pro velké vstupní abecedy.

Sdílené MTBDD řeší problém velkých abeced u stromových automatů tak, že doména MTBDD, tj. posloupnost Booleovských proměnných $\{0,1\}^n$, reprezentuje binární kódování symbolů z množiny $\mathcal{F}$ podle již zmiňované kódovací funkce $enc : \mathcal{F} \to \{0,1\}^n$, kde $n \leq \lceil lg|\mathcal{F}| \rceil$. Použití funkce enc ke zakódování symbolů množiny $\mathcal{F}$ může, pak MTBDD reprezentovat funkci $\mathcal{F} \to \mathbb{D}$. Nejprve definujme množinu *super-stavů* $S(\Delta)$ přechodové funkce Δ jako

$$S(\Delta) = \{(q_1, \ldots, q_p) | p \geq 0, f(q_1, \ldots, q_p) \to D \in \Delta, f \in \mathcal{F}_p, D \subseteq Q, D \neq \emptyset\}$$

nebo v případě úplného automatu se sink stavem q_{sink}:

$$(\Delta) = \{(q_1, \ldots, q_p) | p \geq 0, f(q_1, \ldots, q_p) \to D \in \Delta, f \in \mathcal{F}_p, D \subseteq Q, D \neq \{q_{sink}\}\}$$

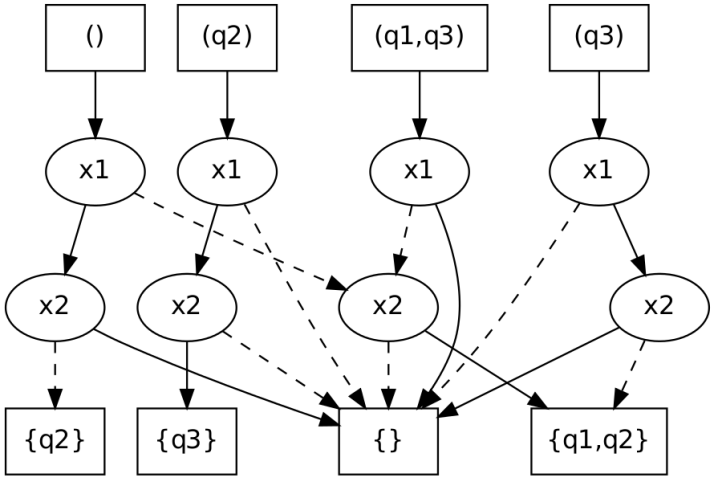
Nechť $S_n(\Delta)$ bude množina super-stavů relace Δ arity n. Všimněme si, že prázdné posloupnost $()$ reprezentuje *počáteční super-stav*, tj. super-stav, z kterého jsou možné přechody z koncového uzlu. Také jsme rozšířili definici relace náležitosti $\in$ na super-stavy následujícím způsobem

$$q \in (q_1, \ldots, q_n) \Leftrightarrow def \exists 1 \leq i \leq n : q = q_i.$$

Můžeme reprezentovat přechodovou funkci Δ stromového automatu $\mathcal{A}$ jako datovou strukturu, která spojuje každý super-stav s MTBDD, která je indexována pomocí binárního kódování symbolů $\mathcal{F}$ a má v koncových stavech podmnožinu z Q. V případě použití sdílené MTBDD je každý super-stav zobrazen na kořen dané MTBDD. Pro lepší pochopení uveďme příklad. Předpokládejme nedeterministický konečný stromový automat $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, kde $Q = \{q_1, q_2, q_3\}$, $\mathcal{F} = \{a, b_0, b_2, c_0, c_1, d_1\}$, a $\Delta = \{b_0 \to \{q_1, q_2\}, c_0 \to \{q_2\}, d_1(q_2) \to \{q_3\}, b_2(q_1, q_3) \to \{q_1, q_2\}, c_1(q_3) \to \{q_1, q_2\}\}$. ($Q_f$ není v tuhle chvíli důležité). Sdílené MTBDD odpovídající přechodové funkci Δ je ukázáno na Obrázku 4.1.

4.2 Možné využití implementované knihovny pro práci s konečnými automaty

Knihovna VATA vychází z knihovny MONA [17]. MONA je knihovna, která pracuje s deterministickými konečnými stromovými automaty. Kódování použité knihovnou MONA odpovídá kódování popsaném v sekci 4.1, ale díky determinismu automatu může provést z jednoho super-stavu pod jedním symbolem f přechod do maximálně jednoho stavu (na



Obrázek 4.1: Reprezentace Δ sdíleným MTBDD. Kódování symbolů $\mathcal{F}$: $a : 00, b : 01, c : 10, d : 11$. Čárkované čáry reprezentují hodnotu 0 dané proměnné a plné čáry reprezentují hodnotu 1. Obrázek převzat z [12]

rozdíl od nedeterministického automatu, který může přejít do jednoho z *množiny* možných stavů). Doména terminálů MTBDD pak může být tvořena jednoduše stavy automatu. VATA dosahuje zobecnění na nedeterministické automaty nejjednodušší možnou cestou, tedy nahrazením jednotlivých stavů množinami stavů (viz. sekce 4.1). Výhodou tohoto konceptuálně jednoduchého zobecnění je, že umožňuje přímočarou implementaci většiny algoritmů pro práci se stromovými automaty. Existuje však potenciálně efektivnější alternativa. Konkrétně, množiny stavů, které jsou ve VATě hodnotami terminálů MTBDD, by mohly být samy reprezentovány pomocí BDD (BDD by obsahovalo proměnnou pro každý stav indikující, zda je stav přítomen v reprezentované množině stavů). Takovým kódováním terminálních hodnot MTBDD, množin stavů automatu, pomocí BDD, bychom převedli MTBDD na klasické BDD. Výhodami tohoto přístupu je 1) na místo MTBDD použití BDD, která jsou jednodušší a je možné je lépe optimalizovat, 2) možnost pracovat se množinami stavů reprezentovanými kompaktně pomocí BDD. Nevýhodou by pak byla větší složitost automatových algoritmů, které by musely pracovat s takovou symbolickou reprezentací množin stavů. BDD knihovnu vyvíjenou v této práci můžeme proto využít pro kódování stromových automatů v knihovně VATA několika způsoby.

Prvním způsobem využití vyvíjené knihovny je rozšířit jí o MTBDD, což by znamenalo několik změn. Nejdříve by se musela vytvořit nová datová struktura pro koncové uzly. Dále by bylo zapotřebí implementovat další operace, které by pracovaly s MTBDD, protože metody pracující s BDD by nešly použít. Jednalo by se především o operaci *Apply* [5], která by měla na vstupu dva odkazy na uzly MTBDD a logickou operaci, která by se měla provést nad těmito uzly. Další nutnou operací je operace, která se nazývá *Projekce*. Rovněž by nebylo možné použít některé techniky zmiňované v kapitole 3 jako například negace hran, operátor *if-then-else* apod.

Druhou možností je použití BDD pro symbolickou reprezentaci nedeterministických stromových automatů. Kódovali bychom množiny následníků super-stavů pomocí BDD a mohli bychom použít přímo knihovnu vyvíjenou v této práci. Potom po provedení relace přechodu bychom na rozdíl od kódování založeného na MTBDD nedostali množinu následných stavů automatu, ale BDD reprezentující následující množinu stavů automatu. Naivním

způsobem použití takového kódování by bylo implementovat operaci materializace množiny stavů kódované pomocí BDD (tato operace by vrátila výčet stavů kódovaným BDD). Za použití materializace by bylo možné všechny automatové algoritmy implementované knihovnou VATA použít v téměř nezměněné podobě. Nevýhodou je, že kromě úspory prostoru pro kódování automatu by takto byly ztraceny všechny výhody symbolické reprezentace množin stavů. Zajímavější avšak náročnější cestou by tedy bylo vyvinout varianty algoritmů knihovny VATA, které materializaci množin stavů nepoužívají a pracují přímo s jejich symbolickou reprezentací. Takové modifikace se zdají být netriviální a vyžadují další výzkum.

V návaznosti na tuto práci plánujeme všechny tři zmíněné možnosti kódování stromových automatů (MTBDD, BDD s materializací, BDD bez materializace) implementovat a porovnat. Doufáme, že vývoj varianty používající BDD bez materializace přinese zvýšení efektivity automatových algoritmů.

Kapitola 5

Návrh

V této části práce budou navrženy a popsány algoritmy a datové struktury, které budou implementovány v knihovně. Tato kapitola začíná popisem reprezentace BDD, jejíž analýzu jsme provedli v kapitole 3. Následuje popis algoritmů pro operace nad BDD, které používají tuto reprezentaci. Dále v kapitole používáme zkratku BDD pro označení ROBDD, který byl definován v části 2.1.5. Tato kapitola vychází z prací [5, 10, 2, 25].

5.1 Binární rozhodovací diagramy

V této části popíšeme návrh implementace základních datových struktur, které se následně budou používat v další části této kapitoly. Bude zde také popsán návrh základních operací nad těmito strukturami.

V navrhované knihovně budeme používat techniku negovaných hran. Budeme uvažovat, že vysoký následník nemůže být negován a je pouze jeden terminální uzel s hodnotou 1. Negaci bude signalizovat nejvíc významný bit adresy uzlu. Pokud tento bit bude mít hodnotu jedna, pak je hrana negovaná. V opačném případě, tedy pokud bude mít bit hodnotu nula, hrana není negovaná.

5.1.1 Unikátní tabulka

Jak již bylo zmíněno v kapitole 3, tak unikátní tabulka slouží k zachování silné kanonické formy. Unikátní tabulka bývá nejčastěji implementována jako hašovací tabulka. Různé BDD knihovny se liší v ukládání BDD uzlů do unikátní tabulky. Existují dva přístupy jak ukládat uzel v unikátní tabulce. První přístup je založen na ukládání dynamických ukazatelů namísto uzlů. Tento přístup je ovšem velmi náročný, co se týče spravování těchto dynamických ukazatelů. Druhý přístup je založen na indexování uzlů. To znamená, že pracujeme s indexem uzlu, což je umístění v poli, ve kterém se BDD uzly nacházejí. Datová struktura BDD uzlu, se kterou pracuje unikátní tabulka, je znázorněna na Kód 5.1.

```
struct UTNode
{
    int Var;
    int Then;
    int Else;
    int Next;
```

```
}
```

Kód 5.1: Ukázka základní datové struktury unikátní tabulky.

Můžeme si všimnout, že na rozdíl od běžných BDD knihoven, neobsahuje datová struktura BDD uzlu počítadlo referencí. Proč tomu tak je, bude vysvětleno níže 5.1.2.

Nejdůležitější metodou nad unikátní tabulkou, kterou používá většina operací manipulující s BDD, je vyhledání nebo vložení BDD uzlu. Jak vyhledání tak vložení BDD uzlu je u většiny BDD knihoven spojené do jedné metody. Popis této metody je na Algoritmu 2. Vstupem algoritmu jsou odkazy na nízkého následníka, `e`, a vysokého následníka, `t`, hledaného uzlu. Dále je potřeba zadat ještě index proměnné `v`, kterou uzel reprezentuje. Podmínka na prvním řádku udává, kdy se má zvětšit unikátní tabulka. Tedy v případě po-

BDD `FindOrAddUniqueTable`(**Index** v, **BDD** e, **BDD** t)

```
1:  if load factor greater than 4 then
2:      ResizeTable();
3:  pos = Hash(v, t, e);
4:  r = uniqueTable[pos];
5:  while r! = 0 do
6:      if r > t and r > e then
7:          if find the node successfully then
8:              return r;
9:          r =get next node from collision chain;
10: end
11: UTNode = get the empty space;
12: set properties of UTNode;
13: uniqueTable[pos] = UTNode;
14: return UTNode;
```

Algoritmus 2: Metoda `FindOrAddUniqueTable`

kud je velikost tabulky $4x$ menší než počet uzlů uložených v tabulce. Algoritmus v případě úspěšného vyhledání, vrací nalezený BDD uzel. V opačném případě vrací index na nově vytvořený BDD uzel. Tato metoda zahrnuje techniku založenou na stáří BDD uzlu, která byla popsána v sekci 3.4.

5.1.2 Tabulka výpočtů

BDD operace jsou typicky založeny na použití dynamického programování. Stěžejní myšlenkou je rozklad problému na podproblémy, které jsou řešeny zvlášť. Aby se zabránilo vícenásobnému výpočtu podproblému, tak je výsledek výpočtu ukládán do paměti (*cache*).

V našem případě hraje tabulka výpočtů roli cache a bude stejně jako unikátní tabulka také implementována jako hašovací tabulka. Každá položka tabulky výpočtů se skládá ze čtyř složek, a to z operátoru, dvou operandů a výsledku.

```
struct CTNode
{
    int Operator;
    int Op1;
    int Op2;
    int Result;
```

}

Kód 5.2: Ukázka položky tabulky výpočtů.

Ovšem v navrhované BDD knihovně se nachází i operace, které mají tři argumenty. V takovém případě je potom místo operátoru použit index (případně ukazatel) na první argument.

Důležitými operacemi nad tabulkou výpočtů jsou vkládání položky a vyhledání položky. Bohužel není možné tyto dvě operace spojit do jedné jako v případě unikátní tabulky. Naštěstí jsou tyto operace triviální, jak ukazuje Algoritmus 3 a Algoritmus 4. Vstupy obou algoritmů jsou následující: `op` je index prováděné operace, `f` a `g` jsou operandy operace, a `res` je výsledek operace.

Void `Insert`(**Index** *op*, **BDD** *f*, **BDD** *g*, **BDD** *res*)

1: *pos* = `Hash`(*op*, *t*, *e*);
2: set properties of new *node*;
3: *computedTable*[*pos*] = *node*;

Algoritmus 3: Operace `Insert` tabulky výpočtů

Bool `Lookup`(**Index** *op*, **BDD** *f*, **BDD** *g*, **BDD** *res*)

1: *pos* = `Hash`(*op*, *t*, *e*);
2: *r* = *computedTable*[*pos*];
3: **if** find the node successfully **then**
4: *res* = *r*;
5: **return** true;
6: **else**
7: **return** false;

Algoritmus 4: Operace `Lookup` tabulky výpočtů

5.1.3 Garbage collection

Jak již bylo zmíněno v sekci 3.6, garbage collection je výpočetně náročný a je potřeba ho provádět co nejméněkrát. Proto jsem se rozhodl provádět garbage collection až v případě, kdy není dostatek paměti. To umožňuje navrhnout jednoduchý garbage collection založený na *mark-and-sweep*. Dále nás to zbavuje potřeby zaznamenávat počet referencí na jednotlivé uzly, čímž se sníží paměťové nároky. Na druhou stranu, v případě častého volání garbage collection, by se mohl zvýšit výpočetní čas. Návrh algoritmu je ukázán v Algoritmus 5.

V `mark phase` se označí všechny živé uzly a provede se inicializace polí. Pole `marked` obsahuje příznaky, zda jsou BDD uzly živé, `mem` je pole obsahující BDD uzly a `currentAddress` je pole obsahující aktuální adresy BDD uzlů.

5.1.4 Hašovací funkce

Hašování je zásadní součástí BDD konstrukce, jelikož jak unikátní tabulka, tak tabulka výpočtů jsou založeny na hašovacích tabulkách. Současně není stále objevená jakákoliv teoreticky dobrá hašovací funkce pro manipulaci s více hašovacími klíči. V práci [25] empiricky zjistili, že hašovací funkce

$$H(k_1, k_2) = ((k_1 \cdot p_1 + k_2) \cdot p_2)/2^{w-n} \tag{5.1}$$

Void `GarbageCollection`()

```
1:   mark phase;
2:   numberOfDeadNodes = 0;
3:   i = 0;
4:   for i to number of nodes do
5:       if ) then marked(i
6:           mem[i − numberOfDeadNodes] = mem[i];
7:           currentAddress[i] = i − numberOfDeadNodes;
8:       else
9:           numberOfDeadNodes = numberOfDeadNodes + 1;
10:  end
11:  update phase;
12:  refresh unique table;
13:  refresh computed table;
```

Algoritmus 5: Operace `GarbageCollection`

dobře distribuuje BDD uzly, kde k_i jsou hašovací klíče, p_i jsou dostatečně velká prvočísla, w je počet bitů datového typu *integer* a 2^n je velikost hašovací tabulky.

Základní myšlenkou této hašovací funkce je distribuce a kombinace bitů v hašovacích klíčích vyšších řádů bitů pomocí celočíselného násobení a následně extrahování výsledku z vyšších řádů bitů. Velikost hašovací tabulky je 2^n z důvodů vyhnutí se více náročné operaci modulo.

V CUDD [21] a CacBDD [15] používají upravenou verzi této hašovací funkce, která vypadá následovně

$$H(k_1, k_2, k_3) = (((k_1 + k_2) \cdot p_1 + k_2) \cdot p_2)/2^{w-n}. \tag{5.2}$$

V mém návrhu budu využívat hašovací funkci zmíněnou výše 5.2. Je to z důvodu, že na rozdíl od hašovací funkce 5.1 bude potřeba hašovací funkce, která má tři argumenty.

5.2 Operace nad binárními rozhodovacími diagramy

V této sekci budou popsány základní operace pro manipulaci s BDD. Tyto operace spoléhají, že na vstupu bude pouze ROBDD.

5.2.1 Ite

Kompletní algoritmus `ite` je prezentován jako Algoritmus 6. Návrh algoritmu vychází z podkapitoly 3.1. Pro rekapitulaci `ite` je jádrem většiny BDD knihoven. Je to ternární Boolovská funkce definována pro tři vstupy F, G, H, které jsou vyhodnoceny: Jestliže F pak G jinak H. Argumenty algoritmu F, G, H jsou odkazy na BDD uzly. Na řádku 6 je zmíněn pojem *top proměnná*. Tento výraz znamená, že je nalezena proměnná pro uzly F, G a H, která má nejnižší index. Tedy pokud uzly reprezentují proměnné x_1, x_2, x_3 a je definováno pořadí proměnných $x_1 < x_2 < x_3$, tak výsledkem operace je 1. Metoda `ite` vrací odkaz na výsledný BDD uzel.

BDD `ite`(**BDD** f, **BDD** g, **BDD** h)

```
1:  if terminal case then
2:      return result;
3:  if computedTable.Lookup(f, g, h, r) then
4:      return result;
5:  else
6:      let v be the top variable of {f, g, h};
7:      t = ite(f|v=1, g|v=1, h|v=1);
8:      e = ite(f|v=0, g|v=0, h|v=0);
9:      if t == e then
10:         return t;
11:     end
12:     r = uniqueTable.FindOrAddUniqueTable(v, t, e);
13:     computedTable.Insert(f, g, h, r);
14:     return r;
```

Algoritmus 6: Operace *ite*

5.2.2 And a Xor

Toto jsou dvě důležité operace v BDD knihovnách. Pomocí operace *And* a negace jsme schopni sestrojit zbývající logické operace. Navíc operace *And* a Xor jsou často používané operace jak v symbolickém ověřování modelu, tak v návrhu obvodů. Specializovanou implementací těchto operací můžeme dosáhnout větší efektivity než používáním operace *ite*. Návrh je podobný jak pro operaci *ite*. Liší se pouze v terminálních případech.

5.2.3 Podpora

Podpora je operace, která vrací proměnné, které se vyskytují v nějakém BDD f. Tato metoda rekurzivně prohledá BDD f a jako výsledek vrátí všechny proměnné ve formě ROBDD, který vzniká operací *And* všech proměnných a dále bude takový produkt nazýván *and-produkt*. Tuto operaci ilustruje obrázek 5.1. Operace podpora je lépe vysvětlena v Algoritmu 8. Funkce `var()` vrací adresu proměnné. Pole `vars` obsahuje příznaky, které

`SupportRecur`(**BDD** f, **Array** vars)

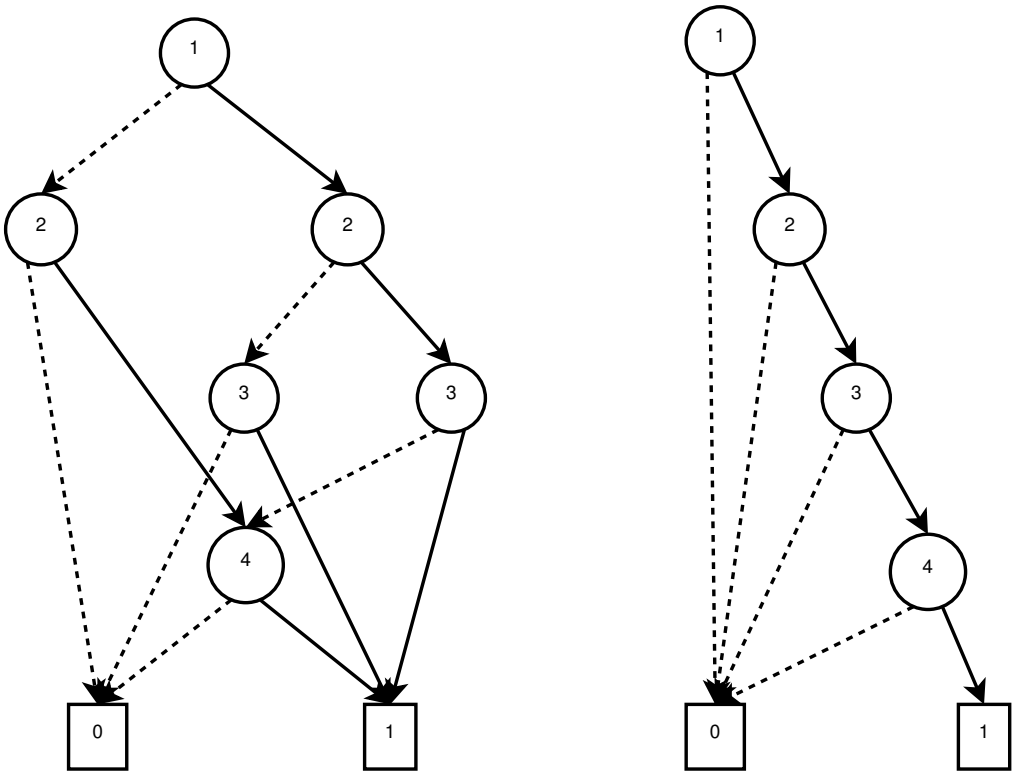
```
1:  if terminal case then
2:      return result;
3:  let v be the top variable of f;
4:  vars[var(v)] = true;
5:  Void SupportRecur(f|v=1, vars);
6:  SupportRecur(f|v=0, vars);
```

Algoritmus 7: Operace *SupportRecur*

indikují, zda se proměnná v BDD f nachází.

5.2.4 Restrikce

Další operace je restrikce. Tento algoritmus se snaží zjednodušit BDD tím, že se pokouší odstranit uzly. Přesněji máme dané BDD f a BDD c a operace `Restrict` nalezne jiné



Obrázek 5.1: Ukázka operace `Support`. Vlevo je BDD, které představuje and produkt z BDD na pravé straně.

BDD f', které je typicky menší nebo rovno f. Například $f = (x_2 \wedge (x_1 \Leftrightarrow (x_3 \Rightarrow x_4))) \vee \overline{(x_2 \vee ((x_4 \Rightarrow x_1) \wedge (x_1 \vee x_3)))}$ a omezení $c = (\overline{x_2} \wedge x_3 \wedge x_4) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$ potom $Restrict(f, c) = x_1 \wedge x_2$. Algoritmus 9 obsahuje pseudokód operace `restrikce` a vychází z práce [8].

5.2.5 Kompozice

Kompozice je ROBDD operace, která provádí ekvivalent substituce na Booleovských výrazech. Často se značí $f[f'/g]$ a používá se k vyjádření výsledku substituce všech volných výskytů g ve f. Provedením této substituce na BDD získáme následující rovnici

$$f|_{x_i=g} = g \wedge f|_{x_i=1} \vee \overline{g} \wedge f|_{x_i=0},$$

kde $f|_{x_i=g}$ znamená, že v BDD f se má nahradit každý uzel, který reprezentuje proměnnou x_i, kořenovým uzlem BDD g. Metoda kompozice je ukázána v Algoritmu 10.

5.2.6 Existenční a univerzální kvantifikátor

Při používání BDD se často používá také existenční kvantifikátor. Existenční kvantifikátor je Booleovská operace $\exists x.t$. Definice existenčního kvantifikátoru Booleovské proměnné je dán následující rovnicí:

$$\exists x.t = t|_{x=0} \vee t|_{x=1}.$$

Nejjednodušším návrhem této operace by bylo použít nejdříve operaci kompozice 5.2.5 a poté $\texttt{OR}(t|_{x=0}, t|_{x=1})$. Takový návrh by ovšem nebyl moc efektivní. Proto například CUDD

BDD `Support`(**BDD** f)

```
1:  initialization vars to false;
2:  r = true;
3:  i = number of variables;
4:  SupportRecur(f, vars);
5:  for i to number of nodes do
6:      if ) then vars(i
7:          r = And(r, var(i));
8:  end
9:  return (r)
```

Algoritmus 8: Operace `Support`

BDD `Restrict`(**BDD** f, **BDD** c)

```
1:  if terminal case then
2:      return result;
3:  if computedTable.Lookup(Restrict, f, c) then
4:      return result;
5:  let v_f be the top variable of f;
6:  let v_c be the top variable of c;
7:  if v_f > v_c then
8:      r = Restrict(f, OR(c|_{v_c=1}, c|_{v_c=0}));
9:  else if v_f < v_c then
10:     r = ite(v_f, Restrict(f|_{v_f=1}, c), Restrict(f|_{v_f=0}, c);
11: else
12:     if c|_{v_c=0} == false then
13:         r = Restrict(f|_{v_f=1}, c|_{v_c=1});
14:     else if c|_{v_c=1} == false then
15:         r = Restrict(f|_{v_f=0}, c|_{v_c=0});
16:     else
17:         r = ite(v_f, Restrict(f|_{v_f=1}, c|_{v_c=1}), Restrict(f|_{v_f=0}, c|_{v_c=0});
18: computedTable.Insert(Restrict, f, c, r);
19: return r;
```

Algoritmus 9: Operace `Restrict`

BDD `Compose`(**Index** v, **BDD** f, **BDD** g)

```
1:  let τ_f be the top variable of f;
2:  let τ_g be the top variable of g;
3:  if τ_f > v then
4:      return f;
5:  if computedTable.Lookup(Compose, f, g) then
6:      return result;
7:  if τ_f == v then
8:      r = ite(g, f|_{τ_f=1}, f|_{τ_f=0});
9:  else
10:     let τ be the top variable of {f, g};
11:     t = Compose(v, f|_{τ=1}, g|_{τ=1});
12:     e = Compose(v, f|_{τ=0}, g|_{τ=0});
13:     r = ite(τ, t, e);
14: computedTable.Insert(Compose, f, g, r);
15: return r;
```

Algoritmus 10: Operace `Compose`

[21] nebo CacBDD [15] používají zvlášť implementace existenčního kvantifikátoru, která je znázorněna v Algoritmus 11. Vstupem nemusí být jen jedna proměnná, ale i množina proměnných. Musí být zkonstruován jako and-produkt, který je ukázán na Obrázku 5.1.

Operace *Universal* představuje univerzální kvantifikátor. Ten sestrojíme jednoduše pomocí existenčního kvantifikátoru, protože platí vztah $\forall x.t = \overline{\exists x.\bar{t}}$. Tato operace bude přibližně stejně výpočetně náročná jako operace *Exist*, protože operace negace je pouze bitová operace.

5.2.7 Relační produkt

Existují dva oblíbené algoritmy založené na BDD pro výpočet následných stavů systému, což, jak již bylo zmíněno v úvodu, se používá v symbolickém ověřování modelu. Jedním z těchto algoritmů je relační produkt (*relation product*) známý také jako *AndExist*, který se aplikuje na relaci přechodů a množinu stavů.

Relační produkt počítá $\exists v.f \wedge g$ a je použit k výpočtu množin stavů po provedení dopředných nebo zpětných přechodů. Bylo dokázáno, že je to NP-těžký problém. BDD Algoritmus 12 znázorňuje výpočet relačního produktu. Tento algoritmus je strukturálně velmi podobný BDD algoritmu pro Booleovskou operaci `AND`. Hlavní rozdíl oproti `AND` je, že když top proměnná potřebuje být kvantifikována, tak je vytvořeno nové BDD (používá se operace `OR`(r_0, r_1)). Díky této dodatečné rekurzi je nejhorší případ složitosti tohoto algoritmu exponenciální v závislosti na velikosti grafů vstupních argumentů.

Stejně jako u existenčního kvantifikátoru, tak i tady nemusí být vstupem jediná proměnná, ale i and-produkt.

BDD `Exists`(**BDD** f, **BDD** c)

```
1:  if terminal case then
2:      return result;
3:  if computedTable.Lookup(Exists, f, c) then
4:      return result;
5:  let v_f be the top variable of f;
6:  let v_c be the top variable of c;
7:  if v_f == v_c then
8:      if f|_{v_f=1} == true or f|_{v_f=0} == true or f|_{v_f=1} == ¬f|_{v_f=0} then
9:          return true;
10:     r_1 = Exists(f|_{v_f=1}, c|_{v_c=1});
11:     if r_1 == true then
12:         computedTable.Insert(Exists, f, c, r);
13:         return true;
14:     r_2 = Exists(f|_{v_f=0}, c|_{v_c=1});
15:     r = Or(r_1, r_2);
16: else
17:     r = ite(v_f, Exists(f|_{v_f=1}, c, Exists(f|_{v_f=0}, c);
18: computedTable.Insert(Exists, f, c, r);
19: return r;
```

Algoritmus 11: Operace `Exists`

BDD `AndExists`(**Index** v, **BDD** f, **BDD** g)

```
1:  if terminal case then
2:      return result;
3:  end
4:  if computedTable.Lookup(AndExists, f, g) then
5:      return result;
6:  let τ be the top variable of {f, g};
7:  r_0 = AndExists(v, f|_{τ=0}, g|_{τ=0});
8:  if τ ∈ v then
9:      if r_0 == true or (r_0 == f|_{τ=1}) or (r_0 == g|_{τ=1}) then
10:         r = r_0;
11:     else if r_0 == ¬f|_{τ=1} then
12:         r = Exists(v, g|_{τ=1});
13:     else if r_0 == ¬g|_{τ=1} then
14:         r = Exists(v, f|_{τ=1});
15:     else
16:         r_1 = AndExists(v, f|_{τ=1}, g|_{τ=1});
17:         r = OR(r_0, r_1);
18: else
19:     r_1 = AndExists(v, f|_{τ=1}, g|_{τ=1});
20:     r = uniqueTable.FindOrAddUniqueTable(τ, r_0, r_1);
21: computedTable.Insert(AndExists, f, g, r);
22: return r;
```

Algoritmus 12: Operace `AndExists`

Kapitola 6

Implementace

Tato kapitola obsahuje popis tříd, které byly použity k implementaci knihovny, jejíž návrh je popsán v kapitole 5. Jako implementační jazyk byl vybrán C++ kvůli jeho efektivnosti, dobré podpoře a rozsáhle standardní knihovně.

Implementace unikátní tabulky, jejíž návrh je popsán v 5.1.1, se nachází ve třídě `UniqueTable`. Byl implementován přístup založený na indexování uzlu. Tento přístup byl vybrán především z důvodu jeho lehčí implementace. Tato třída obsahuje kromě metody pro vyhledání nebo vložení prvku `FindOrAddUniqueTable` také metodu pro zotavení se z garbage collection `Refresh` a metodu pro zvětšení tabulky `Resize`. Důležitou součástí této třídy je odkaz na třídu `Heap`, která obsahuje pole BDD uzlů a která se stará o správu těchto uzlů. Třída `Heap` obsahuje metodu `GetFreeNode`, která vrací index na nově vytvořený uzel. Pokud uživatel nezadá jinak, tak počet BDD uzlů, vytvořených při inicializaci objektu `Heap`, je roven velikosti 100MB. Takové velké množství uzlů se inicializuje pro zefektivnění celé knihovny. Třída `UTNode` reprezentuje BDD uzel a s objekty této třídy pracuje třída `Heap`.

Třída `ComputedTable` implementuje tabulku výpočtů. Tato třída pracuje s instancemi třídy `CTNode`, která reprezentuje položku v tabulce výpočtů. Třída `ComputedTable`, kromě metod pro vkládání `Insert` a vyhledání `LookUp` položek, obsahuje stejně jako třída `UniqueTable` metodu pro zvětšení `Resize` a metodu pro zotavení se z garbage collection `Refresh`.

`BDDManager` je třída, která je jádrem celé BDD knihovny. Obsahuje všechny operace, které manipulují s BDD, popsané v kapitole 5 jako například `Ite`, `Restrict` apod. Tato třída obsahuje mimo jiné implementaci metody zmíněné v sekci 3.5, implementaci metody `GarbageCollection` a také obsahuje instance tříd unikátní tabulky, tabulky výpočtů a haldy. Uchovává také seznam proměnných a odkazy na BDD uzly ze třídy `BDD`, která je popsána o odstavec níž. Dědí od virtuální třídy `Manager`, která poskytuje metody pro uživatelskou manipulaci. Mezi tyto metody patří:

`GetZero` tato metoda vrací objekt třídy `BDD`, který reprezentuje false.

`GetOne` tato metoda vrací objekt třídy `BDD`, který reprezentuje true.

`GetVariable` tato metoda vrací objekt třídy `BDD`, který reprezentuje poslední proměnnou. Zároveň tato metoda přidává proměnou do seznamu proměnných.

Třída `BDD` abstrahuje složitou manipulaci se třídou `BDDManager` a poskytuje uživateli jednoduchou manipulaci s knihovnou. Tato třída obsahuje přetížené operátory:

+ pro Booleovskou funkci *or*.

* pro Booleovskou funkci *and*.

^ pro Booleovskou funkci *xor*.

% pro Booleovskou funkci *Xnor*.

&& pro Booleovskou funkci *Nand*.

|| pro Booleovskou funkci *Nor*.

! pro negaci.

Obsahuje taktéž metodu `GetVariable`, která vrací reprezentovanou proměnnou.

Kapitola 7

Experimenty

Tato kapitola se zabývá vyhodnocením provedených experimentů nad implementací BDD knihovny popsané v kapitole 6. Budeme testovat výkon knihoven z hlediska času s různým počátečním nastavením velikosti tabulek ke zkoumání vlivu počátečního nastavení velikosti tabulek na výkon jednotlivých knihoven. Všechny experimenty byly prováděny na počítači s CPU Intel Core i7 (2.20GHz) s pamětí 4GB na operačním systém ubuntu 14.04. Použitý překladač je g++ verze 4.8.2. Naměřené časy jsou CPU a každý test byl proveden pětkrát. Výsledky byly potom zprůměrovány. Uspořádání proměnných odpovídá uspořádání v testovaných souborech.

K testování implementované knihovny jsem použil dvě sady převzatých benchmarků z [1]. První sada benchmarků se nazývá *smv-bdd-traces98*(*abp11*, *dartes*, *motors-stuck*, *valves-gates*, *abp4*, *dme1*, *dme2*, *gigamax*, *guidance*) a reprezentuje symbolické ověřování modelu Symbolic Model Verifier dále jen SMV) z Carnegie Mellon University. Tyto benchmarky byly získány zaznamenáním volání BDD funkcí během provádění SMV. Byly zaznamenány pouze klíčové Booleovské operace. Druhá sada benchmarků reprezentuje kombinační obvody a tato sada benchmarků bývá označována jako *ISCAS85*(*c2670*, *c3540*, *c6288-10*, *c6288-11*, *c6288-12*, *c6288-13*). Tato sada benchmarků je možná jednou z nejoblíbenějších sad benchmarků pro vyhodnocení výkonu pro BDD [25]. Tyto benchmarky se používají k měření výkonu i v jiných pracích jako například [15, 25]. Obě sady benchmarků jsou ve stejném formátu. Za účelem testování je použita aplikace *bdd-trace-driver-0.9* [1], která je napsána v jazyce C a která poskytuje rozhraní mezi použity benchmarky a BDD knihovnami.

7.1 Experiment 1

Zde provedu experiment, který porovná implementovanou knihovnu se dvěma existujícími knihovnami. Tento experiment bude sloužit jako referenční pro další experimenty. Porovnávané knihovny jsou CUDD a CacBDD, které již byly popsány v podkapitole 2.3. Pro rekapitulaci, knihovna CUDD, napsaná v jazyce C, používá místo jedné velké unikátní tabulky několik menších unikátních tabulek pro každou proměnnou. Nepoužívá techniku dynamické tabulky a používá počítadlo referencí pro omezení vyhledávání a vkládání do tabulky výpočtů. Knihovna CacBDD, napsaná v jazyce C++, používá jednu velkou unikátní tabulku a používá techniku dynamické tabulky výpočtů. Testované knihovny mají dynamickou unikátní tabulku, tj. při překročení určitého počtu položek tabulky (v případě mé implementované knihovny, když je počet položek unikátní tabulky $4x$ větší než

je kapacita tabulky) dojde k zvětšení unikátní tabulky. Většinou dojde ke zdvojnásobení velikosti.

Aby byl test co možná nejspravedlivější, byly nastaveny výchozí hodnoty tabulek (unikátní a výpočtů) na stejnou velikost a to na 2^{18} položek. Tato hodnota byla zvolena, neboť je dostatečně vysoká a v knihovně CacBDD je to výchozí hodnota tabulek. V Tabulce 7.1 jsou ukázány naměřené hodnoty pro jednotlivé soubory z benchmarku. První tři sloupce tabulky jsou časy jednotlivých knihoven a další sloupce označené *Mem* jsou paměť. Každý řádek tabulky představuje výsledek knihoven pro uvedený benchmark. Naměřené časy představují dobu potřebnou ke zpracování jednoho benchmarku a použitá časová jednotka v tabulce jsou sekundy. Dále byla měřena i spotřebovaná paměť na konci výpočtu u jednotlivých knihoven. V tabulce je naměřená hodnota vyjádřena v MB.

V tomto experimentu vyšla časově lépe moje implementovaná knihovna (FITBDD) než CUDD a CacBDD. V paměťových nárocích vyšla líp knihovna CacBDD.

Instance	CUDD	FITBDD	CacBDD	CUDDMem	FITBDDMem	CacBDDMem
c2670	2.63	2.67	2.66	856,669	171.748	171.748
c3540	10.21	5.07	5.15	895,017	557.371	477.371
c6288-10	0.34	0.16	0.16	298,701	23.5998	23.5998
c6288-11	0.95	0.62	0.64	358,382	79.3724	59.3724
c6288-12	3.57	2.41	2.44	419,323	198.281	198.281
c6288-13	12.42	8.27	8.40	767,192	675.286	515.286
abp11	7.93	6.65	11.45	539,721	354.071	74.0712
dartes	9.06	9.11	7.01	835.178	631.711	471.711
motors-stuck	7.57	6.13	9.67	643,283	550.761	310.761
valves-gates	7.42	7.78	9.48	648,466	612.489	372.489
abp4	0.40	0.11	0.09	380,723	12.3676	12.3676
dme1	0.43	0.19	0.19	475,461	37.0789	27.0789
dme2	0.28	0.11	0.11	483,599	21.8265	21.8265
gigamax	0.19	0.06	0.05	420,319	9.0295	9.0295
guidance	1.13	0.75	0.68	627,070	86.807	66.809

Tabulka 7.1: Porovnání implementované knihovny s CUDD a CacBDD. Naměřený čas je v sekundách a použitá paměť je v MB.

7.2 Experiment 2

Tento test se zaměřuje na vyhodnocení vlivu počáteční velikosti unikátní tabulky na celkový čas a paměťové nároky potřebné ke zpracování benchmarků. Výchozí nastavení tabulky výpočtů je 2^{18} položek a výchozí nastavení unikátní tabulky je 2^8 položek. Hodnota 2^8 byla vybrána záměrně, neboť je dostatečně malá, aby byl znát vliv tabulky na výpočet. Výsledek experimentu je vidět v Tabulce 7.2.

Výsledek experimentu ukazuje, že naměřený čas knihoven se příliš neliší od těch referenčních z Tabulky 7.1. Z toho můžeme dojít k závěru, že počáteční velikost unikátní

tabulky příliš neovlivňuje výkon implementované knihovny. Ani paměťové nároky nebyly nějak výrazně ovlivněny.

Knihovna FITBDD je lehce rychlejší než zbylé knihovny. V paměťových nárocích je ovšem horší než CaCBDD, ale v případě CUDD je na tom se spotřebovanou pamětí lépe.

Instance	CUDD	FITBDD	CacBDD	CUDDMem	FITBDDMem	CacBDDMem
c2670	4.37	2.71	2.69	275,781	171.748	171.748
c3540	11.77	5.38	5.47	891,026	557.371	477.371
c6288-10	0.37	0.17	0.18	38,6101	23.5998	23.5998
c6288-11	1.48	0.66	0.67	114,281	79.3724	59.3724
c6288-12	4.57	2.49	2.51	317,031	198.281	198.281
c6288-13	14.62	8.24	8.84	813,846	675.286	515.286
abp11	10.31	6.97	11.91	184,846	354.071	74.0712
dartes	10.06	9.86	7.04	789.178	631.711	471.711
motors-stuck	8.67	5.95	9.72	322,658	550.761	310.761
valves-gates	8.47	7.75	9.98	327,841	612.489	372.489
abp4	0.29	0.11	0.09	10,9498	11.8676	11.8676
dme1	0.34	0.20	0.19	35,0049	37.0789	27.0789
dme2	0.18	0.11	0.12	14,0700	21.8265	21.8265
gigamax	0.19	0.06	0.07	420,319	8.2795	8.27996
guidance	1.13	0.79	0.68	627,070	86.807	66.809

Tabulka 7.2: Porovnání implementované knihovny s CUDD a CacBDD. Naměřený čas je v sekundách a použitá paměť je v MB.

7.3 Experiment 3

Experiment se zaměřuje na pozorování vlivu při nastavení výchozí velikosti tabulky výpočtů na celkový čas a paměťové nároky potřebné k výpočtu benchmarků. Počáteční hodnota unikátní tabulky je výchozí, tj. 2^{18} položek, a počáteční hodnota položek tabulky výpočtů je 2^8. Výsledek měření je uveden v tabulce 7.3.

Vyhodnocení experimentu ukazuje, že počáteční velikost tabulky výpočtů ovlivňuje výkon knihoven FITBDD a CacBDD. Stejně tak ovlivňuje i paměťové nároky obou knihoven. Obě knihovny mají implementovanou již zmíněnou techniku dynamické tabulky výpočtů. Zatímco rychlost je ovlivněna negativně, tak paměťové nároky knihoven byly sníženy až na CUDD. Všimněme si, že k největšímu zhoršení výkonu dochází u benchmarků, které představují ověřování modelu (model checking). To odpovídá i závěrům z práce [25], kde tvrdí, že při ověřování modelu dochází k velkému množství opakování podproblému na rozdíl od kombinačních obvodů.

Implementovaná knihovna v tomto experimentu je výrazně pomalejší než CUDD a je také pomalejší než CacBDD. Na druhou stranu má knihovna FITDD paměťové nároky výrazně nižší než CUDD a srovnatelné s CacBDD.

Instance	CUDD	FITBDD	CacBDD	CUDDMem	FITBDDMem	CacBDDMem
c2670	2.61	29.99	29.60	856,669	92.9983	92.9983
c3540	10.08	7.29	7.15	895,017	397.684	397.684
c6288-10	0.33	0.14	0.15	298,701	13.6194	13.6096
c6288-11	0.99	0.57	0.56	358,382	39.4114	39.4114
c6288-12	3.86	2.42	2.28	419,323	118.437	118.359
c6288-13	12.29	8.10	7.90	767,192	355.598	355.598
abp11	8.35	15.37	15.52	539,721	36.5712	35.3212
dartes	9.06	305.2	146.84	835.178	321.711	316.711
motors-stuck	7.81	50.17	48.58	643,283	233.261	233.261
valves-gates	7.56	39.22	38.64	648,466	294.989	293.739
abp4	0.41	0.61	0.62	380,723	12.4456	11.8676
dme1	0.42	2.50	3.20	475,461	17.3914	17.3914
dme2	0.27	0.34	0.33	483,599	11.8655	11.8655
gigamax	0.17	0.12	0.13	420,319	4.04903	4.04903
guidance	1.12	4.00	3.79	627,070	47.1195	47.1215

Tabulka 7.3: Porovnání implementované knihovny s CUDD a CacBDD. Naměřený čas je v sekundách a použitá paměť je v MB.

7.4 Diskuze

V této kapitole byly provedeny experimenty nad knihovnou FITBDD. Z experimentů celkově vyšlo, že FITBDD celkem obstála v konkurenci knihoven CUDD a CacBDD. Dále z experimentů bylo zjištěno, že výkon a paměťové nároky knihovny FITBDD ovlivňuje počáteční velikost tabulky výpočtů a to zejména v případě ověřování modelu. V případě nízkého počtu položek tabulky výpočtů bylo dosaženo zpomalení, ale na druhou stranu byly sníženy paměťové nároky. V případě vysokého počtu položek tabulky výpočtů nastal pravý opak. V případě zkoumání vlivu počátečního nastavení velikosti unikátní tabulky nebylo dosaženo výrazného zhoršení ani zlepšení jak výkonu tak paměťových nároků. Knihovna FITBDD oproti knihovně CacBDD vykazovala zvýšené nároky na paměť.

Kapitola 8

Závěr

Hlavním cílem této práce bylo vytvořit BDD knihovnu, která má být efektivní a která by obsahovala základní operace nad BDD. Dále tato knihovna by měla být výkonnostně podobná s existujícími dostupnými BDD knihovnami.

V první části této práce bylo popsáno, co to je BDD, k čemu slouží, jak vzniká a jeho další rozšíření. Byl zde nastíněn i problém uspořádání proměnných, které se z důvodu vysoké výpočetní náročnosti a ne vždy dobrých výsledků [25] dále neuvažuje. Dále práce obsahuje popis existujících knihoven a stejně tak i možné aplikace BDD v praxi. V další části práce jsou popsány klasické techniky používané v BDD knihovnách a jejich různé modifikace. Následně je popsána knihovna VATA a je diskutována možnost propojení implementované knihovny (FITBDD) s knihovnou VATA. V závěru práce je popsán návrh a implementace BDD knihovny.

Knihovna FITBDD implementuje jednu velkou unikátní tabulku. Tabulka výpočtu používá techniku dynamické tabulky výpočtu a garbage collection je volán pouze v případě, když dochází paměť systému. Tento přístup implementují i některé další knihovny jako například CUDD a CacBDD. FITBDD obsahuje také techniku negaci hran pro snížení nároku na paměť.

V kapitole 7 byly provedeny experimenty nad knihovnou FITBDD. Z experimentů vyšlo, že naše implementovaná knihovna je rychlejší než CUDD v případě dostatečné počáteční velikosti tabulky výpočtů a je také lehce rychlejší než CacBDD. Knihovna FITBDD má také menší paměťové nároky než CUDD, ale zase větší než CacBDD. Z provedených experimentů také vyšlo, že počáteční velikost tabulky výpočtů má vliv na výkon knihovny FITBDD.

Zde stále existuje plno možností pro budoucí práci. Jedna z nich byla nastíněna v kapitole 4. Jedná se o propojení knihovny FITBDD s knihovnou VATA. Bylo by zajímavé porovnat existující verzi knihovny VATA, která používá MTBDD s verzí, která by používala zde popisovanou knihovnu. Dále by bylo zajímavé změnit návrh knihovny a místo jedné velké unikátní tabulky použít stejně jako CUDD několik menších v závislosti na počtu proměnných. Další vylepšení, které poskytuje knihovna CUDD a které je zmiňované v podkapitole 3.5 je omezení hledání a vkládání do tabulky výpočtu. Vhodné by také bylo prošetřit zvýšené paměťové nároky oproti knihovně CacBDD.

Literatura

[1] The BDD benchmark. `http://www.cs.cmu.edu/~bwolen/software/`.

[2] Andersen, H. R.: An introduction to binary decision diagrams. *Course Notes on the WWW*, 1997.

[3] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; aj.: Abstract regular (tree) model checking. ročník 14, 2012: s. 167–191.
URL http://dx.doi.org/10.1007/s10009-011-0205-y

[4] Brace, K. S.; Rudell, R. L.; Bryant, R. E.: Efficient Implementation of a BDD Package. In *DAC*, 1990, s. 40–45.

[5] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, ročník 35, č. 8, 1986: s. 677–691.

[6] CacBDD: Web pages of CacBDD. `http://kailesu.net/CacBDD/`.

[7] Comon, H.; Dauchet, M.; Gilleron, R.; aj.: Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.

[8] Coudert, O.; Madre, J. C.: A Unified Framework for the Formal Verification of Sequential Circuits. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1990, Santa Clara, CA, USA, November 11-15, 1990. Digest of Technical Papers*, IEEE Computer Society, 1990, s. 126–129.

[9] van Dijk, T.: The parallelization of binary decision diagram operations for model checking. 2012.
URL http://essay.utwente.nl/61650/

[10] Drechsler, R.; Sieling, D.: Binary decision diagrams in theory and practice. *STTT*, ročník 3, č. 2, 2001: s. 112–136.

[11] Elgaard, J.; Klarlund, N.; Møller, A.: MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification, CAV '98*, *LNCS*, ročník 1427, Springer-Verlag, s. 516–520.

[12] Lengál, O.: *An Efficient Finite Tree Automata Library: The Design of BDD-based Semi-symbolic Algorithms for Nondeterministic Finite Tree Automata*. Lambert Academic Publishing, 2012, ISBN 978-3-659-27069-7, 64 s.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10206

[13] Long, D. E.: The design of a cache-friendly BDD library. In *ICCAD*, 1998, s. 639–645.

[14] Lv, G.; Chen, Y.; Feng, Y.; aj.: A Succinct and Efficient Implementation of a 2^32 BDD Package. In *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, IEEE Computer Society, 2012, s. 241–244.

[15] Lv, G.; Su, K.; Xu, Y.: CacBDD: A BDD Package with Dynamic Cache Management. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, *Lecture Notes in Computer Science*, ročník 8044, Springer, 2013, s. 229–234.

[16] Minato, S.; Ishiura, N.; Yajima, S.: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In *DAC*, 1990, s. 52–57.

[17] MONA: Web pages of MONA. `http://www.brics.dk/mona/`.

[18] O. Coudert, H. T., J. C. Madre: *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.

[19] Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, IEEE Computer Society, 1993, s. 42–47.

[20] Sanghavi, J. V.; Ranjan, R. K.; Brayton, R. K.; aj.: High Performance BDD Package By Exploiting Memory Hiercharchy. In *DAC*, 1996, s. 635–640.

[21] Somenzi, F.: CUDD: CU Decision Diagram Package Release. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[22] Vasícek, Z.; Sekanina, L.: How to evolve complex combinational circuits from scratch? In *2014 IEEE International Conference on Evolvable Systems, ICES 2014, Orlando, FL, USA, December 9-12, 2014*, IEEE, 2014, s. 133–140.

[23] Veanes, M.: Applications of Symbolic Finite Automata. In *CIAA'13*, *LNCS*, ročník 7982, Springer, 2013, str. 16?23.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=196314

[24] Vojnar, T.: *Formální analýza a verifikace*. FIT VUT v Brně, 2014.

[25] Yang, B.; Bryant, R. E.; O'Hallaron, D. R.; aj.: A Performance Study of BDD-Based Model Checking. In *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*, *Lecture Notes in Computer Science*, ročník 1522, Springer, 1998, s. 255–289.