VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

APROXIMACE OBVODŮ S VYUŽITÍM ALTERNATIVNÍCH REPREZENTACÍ

APPROXIMATE CIRCUITS IN ALTERNATIVE REPRESENTATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TOMÁŠ MICHALISKO**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

154750

Ústav: Ústav počítačových systémů (UPSY)
Student: **Michalisko Tomáš, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Strojové učení
Název: **Aproximace obvodů s využitím alternativních reprezentací**
Kategorie: Umělá inteligence
Akademický rok: 2023/24

Zadání:

1. Seznamte se s datovými strukturami používanými k reprezentaci logických obvodů jako je And-Inverter-Graph, Majority-Inverter-Graph, XOR-Majority-Inverter-Graph, apod. Seznamte se s principem evoluční aproximace logických obvodů.
2. Navrhněte nástroj umožňující aproximaci logických obvodů zadaných v různých reprezentacích. Problém aproximace by mělo být možné zadat jako jedno kriteriální nebo vícekriteriální problém.
3. Navržený nástroj implementujte. Snažte se maximalizovat počet ohodnocených řešení za jednotku času, tj. o efektivní implementaci fitness funkce.
4. Vyhodnoťte parametry navrženého řešení na sadě testovacích obvodů a diskutujte dosažené výsledky.

Literatura:
- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
- Splnění bodu 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2023
Termín pro odevzdání: 17.5.2024
Datum schválení: 2.5.2024

Abstrakt

Tato diplomová práce se zabývá návrhem aproximačních obvodů s využitím alternativních reprezentací. Mezi zkoumané reprezentace patří And-inverter graf, Majority-Inverter graf a Xor-Majority graf. Pro automatizaci návrhu je použito kartézské genetické programování. Díky výpočtu aproximační chyby pomocí formálních metod je možné vytvořený systém aplikovat i na složitější obvody. V první části experimentů je vyhodnocena a optimalizována rychlost programu. Následně je hledán vhodný mutační operátor. Poté je systém otestován při aproximaci 8bitových násobiček a 16bitových sčítaček s cílem minimalizovat velikost a zpoždění. Bylo zjištěno, že sčítačky i násobičky v reprezentaci XMG dosahují lepších fitness hodnot v porovnání s evolucí na úrovni hradel. Na závěr je provedena evoluce s cílem mapování na technologii k-LUT. Zde zůstávají nejefektivnější reprezentací hradla.

Abstract

This master's thesis deals with the design of approximate circuits using alternative representations. The investigated representations include the And-inverter graph, Majority-Inverter graph, and Xor-Majority graph. Cartesian genetic programming is employed for design automation. By computing the approximation error using formal methods, the developed system can be applied to more complex circuits. In the first part of the experiments, the speed of the program is evaluated and optimized. Subsequently, a suitable mutation operator is searched for. Then, the system is tested for approximating 8-bit multipliers and 16-bit adders with the aim of minimizing size and delay. The results show that adders and multipliers in the XMG representation achieve better fitness values compared to evolution at the gate level. Finally, an evolution targeting the k-LUT technology is performed. Here, gates remain the most efficient representation.

Klíčová slova

aproximace obvodů, kartézské genetické programování, And-Inverter graf, Majority-Inverter graf, Xor-Majority graf, LUT mapování

Keywords

approximate circuits, carthesian genetic programming, And-Inverter graph, Majority-Inverter graph, Xor-Majority graph, LUT mapping

Citace

MICHALISKO, Tomáš. *Aproximace obvodů s využitím alternativních reprezentací.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Aproximace obvodů s využitím alternativních reprezentací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Zdeňka Vašíčka, Ph.D. a prof. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Tomáš Michalisko
15. května 2024

Poděkování

Děkuji doc. Ing. Zdeňku Vašíčkovi, Ph.D. a prof. Ing. Lukáši Sekaninovi, Ph.D. za ochotu a odborné vedení práce.

Obsah

1 Úvod **6**

2 Optimalizace kombinačních obvodů **7**
2.1 And-Inverter graf 7
2.2 Majority-Inverter graf 9
2.3 XOR-Majority graf 12

3 Genetické programování **14**
3.1 Reprezentace kandidátních řešení 14
3.2 Fitness funkce 15
3.3 Ukončení a výsledek evoluce 16
3.4 Statistické vyhodnocení experimentů 16

4 Kartézské genetické programování **17**
4.1 Genotyp a fenotyp 17
4.2 Ukázka zakódování obvodu 18
4.3 Algoritmy pro dekódování genotypu 19
4.4 Křížení a mutace 19
4.5 Evoluční algoritmus 20
4.6 Alternativní implementace mutace 20

5 Evoluční návrh aproximačních obvodů **22**
5.1 Hodnotící kritéria 23
5.2 Paralelní simulace obvodů 24
5.3 Aplikace formálních metod 25
5.4 Specializovaný mutační operátor 28

6 Návrh a implementace nástroje pro evoluční aproximaci obvodu **30**
6.1 Generování aritmetických obvodů 30
6.2 Převod do alternativních reprezentací a CGP 31
6.3 Modulární implementace CGP 31
6.4 Datový typ pro funkční množinu 31
6.5 Struktura pro genotyp 32
6.6 Použité mutační operátory 32
6.7 Výpočet aproximační chyby 33
6.8 Fitness funkce 33

7 Experimentální vyhodnocení **36**

7.1 Efektivní reprezentace bitového vektoru 36
7.2 Optimalizace převodu primárních výstupů na čísla 38
7.3 Porovnání mutačních operátorů 42
7.4 Výpočet hodnotících kritérií pomocí BDD 44
7.5 Minimalizace grafové reprezentace 46
7.6 Mapování na technologii LUT 49
7.7 Fitness funkce zaměřená na LUT mapování 51

8 Závěr **54**

Literatura **56**

A Další výsledky experimentů **61**

B Obsah paměťového média a návod na spuštění **64**

Seznam obrázků

2.1 V pravé části je uveden AIG pro 1bitovou poloviční sčítačku. Hrany zakončené vyplněným kolečkem jsou negované. V levé části je popis tohoto AIG v ASCII verzi formátu AIGER (pravý sloupec je pouze komentář a není součástí souboru). Rozhraní obvodu tvoří dva vstupy a dva výstupy. Například výstupu uzlu `AND 0` je přiřazen literál 6. Jeho vstupem jsou literály 13 a 15, což jsou negované verze literálů 12 a 14. Vstupům a výstupům jsou přiřazeny symboly popisující jejich význam. Komentář obsahuje název obvodu. 9

3.1 Anatomie krabicového grafu. Q_1 značí první kvartil, Q_3 třetí kvartil, svislá čára mezi Q_1 a Q_3 medián. Hodnota IQR se vypočte jako $Q_3 - Q_1$. Odlehlé hodnoty leží mimo interval $\langle Q_1 - 1,5 \times IQR; Q_3 + 1,5 \times IQR \rangle$. Vousy sahají od první neodlehlé hodnoty k poslední. [26] 16

4.1 Ukázka genotypu. Každý uzel je popsán trojicí genů – první dva jsou propojovací a třetí (podtržený) specifikuje funkci daného uzlu. Pod každou trojicí je uvedena adresa příslušného uzlu. Přerušovanou čarou jsou vyznačeny uzly, které nejsou aktivní a nepodílí se na výstupu. Poslední dva geny označují primární výstupy O_0 a 0_1. 18
4.2 Zobrazení genotypu z obr. 4.1 v CGP mřížce. Uzly jsou uspořádány do CGP mřížky, tvořené dvěma řádky a třemi sloupci. U každého uzlu je uvedena jeho adresa a funkce. Uzly a propojení vykreslené přerušovanou čarou nejsou aktivní. Uzel s adresou 7, realizuje funkci NOT. Arita této funkce je 1 a z toho důvodu je pouze první vstup označen jako aktivní. 18
4.3 Fenotyp příslušící genotypu z obr. 4.1. Tento kombinační obvod je sestaven ze 4 hradel. Primární výstupy počítají logické funkce $O_0 = \neg((I_0 \vee I_1) \wedge I_2)$ a $O_0 = (I_0 \vee I_1) \oplus I_2$. 19

5.1 Schéma základu porovnávacího obvodu používaného pro výpočet maximální absolutní chyby e_{wce}. Tento obvod počítá rozdíl výstupů porovnávaných obvodů F a $\hat{F}$, které realizují Booleovské funkce $\mathbb{B}^n \rightarrow \mathbb{B}^m$. Výsledný rozdíl ε je kódován ve dvojkovém doplňku pomocí $m + 1$ bitů. 26
5.2 Příklad aplikace deaktivační části operátoru SagTree. Červenou barvou jsou označeny uzly v deaktivovaném podgrafu S a signály které vychází z deaktivovaných uzlů. Signály (červené), které vychází z S, jsou nahrazeny náhodnými vstupy S (oranžové). Výsledkem nahrazení jsou modré signály. Převzato z [10], zjednodušeno. 29

7.1 Ukázka činnosti algoritmu pro transpozici bitové matice. 40

7.2 Výsledky porovnání jednotlivých variant převodu bitových vektorů s primárními výstupy na vektor čísel při evoluční minimalizaci sčítaček a násobiček. Počet bitů v názvech obvodů udává bitovou šířku jednotlivých operandů. Předmětem měření je celková doba běhu evolučního algoritmu ve všech třech uvažovaných reprezentacích. 41
7.3 Fitness hodnoty nejlepších nalezených řešení při evoluci 8bitových aproximačních násobiček s max. $e_{wce} = 1$ %. První parametr u `*Tree` operátorů udává maximální hloubku deaktivovaného podstromu a druhý parametr určuje pravděpodobnost, že se provede mutace místo stromové deaktivace. Parametr u `PointNoSkip` vyjadřuje počet zmutovaných genů. 43
7.4 Vývoj průměrné fitness hodnoty rodiče v průběhu evoluce 8bitových aproximačních násobiček s max. $e_{wce} = 1$ %. Počet evaluací je zobrazen na logaritmické stupnici. První parametr u `*Tree` operátorů udává maximální hloubku deaktivovaného podstromu a druhý parametr určuje pravděpodobnost, že se provede mutace místo stromové deaktivace. Parametr u `PointNoSkip` vyjadřuje počet zmutovaných genů. 45
7.5 Závislost průměrné doby výpočtu evolučního algoritmu na bitové šířce vstupních operandů. Doba výpočtu je v levém grafu zobrazena na logaritmické stupnici a v pravém grafu je na lineární stupnici. 47
7.6 Porovnání průměrné fitness hodnoty kandidátních řešení nalezených při aproximaci 16bitových sčítaček a 8bitových násobiček. 48
7.7 Mapování násobiček aproximovaných s využitím generických fitness funkcí na 4-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 50
7.8 Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 4-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 50
7.9 Vývoj průměrné fitness hodnoty v průběhu evoluce při použití fitness funkce zaměřené na 4-LUT mapování. Na vodorovné ose je použito logaritmické měřítko. Minimum resp. maximum na svislé ose odpovídá fitness hodnotě nejlepšího resp. počátečního obvodu. 52
7.10 Porovnání fitness funkcí pro reprezentace při evoluci sčítaček. Minimum resp. maximum na vodorovné ose odpovídá fitness hodnotě nejlepšího resp. nejhoršího obvodu. 52
7.11 Vývoj průměrné fitness hodnoty v průběhu evoluce při použití fitness funkce zaměřené na 4-LUT mapování. Na vodorovné ose je použito logaritmické měřítko. Minimum resp. maximum na svislé ose odpovídá fitness hodnotě nejlepšího resp. počátečního obvodu. 53
7.12 Porovnání fitness funkcí pro reprezentace při evoluci násobiček. Minimum resp. maximum na vodorovné ose odpovídá fitness nejlepšího resp. nejhoršího obvodu. 53

A.1 Mapování násobiček aproximovaných s využitím generických fitness funkcí na 5-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 61
A.2 Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 5-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 62

A.3 Mapování násobiček aproximovaných s využitím generických fitness funkcí na 6-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 62
A.4 Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 6-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení. 63

Kapitola 1

Úvod

V současné době se snažíme nalézt nové technologie, které by nám umožnily překonat limity těch stávajících. Je možné, že obvody v těchto nových technologiích budou poskládány z jiných elementárních stavebních bloků než jsme zvyklí. A proto má smysl hledat způsoby, jak tyto obvody efektivně navrhovat. Ideálně automatizovaně. Jednou z možností je využít některou z alternativních reprezentací pro popis obvodů, neboť by mohla být efektivně realizovatelná pomocí netradičních elementárních operací. Kromě toho se ukazuje, že tyto reprezentace jsou efektivní i pro optimalizace obvodů v běžně používaných technologiích.

Kromě výkonu nás zajímá také spotřeba energie nebo velikost na čipu. Díky existenci aplikací, ve kterých jsme ochotni tolerovat drobné nepřesnosti, můžeme vytvářet aritmetické obvody, jenž nám dovolují ušetřit relativně velké množství energie/plochy za cenu zanedbatelného zhoršení přesnosti. Problém spočívá v netriviálním návrhu těchto tzv. aproximačních obvodů.

Cílem této práce je navrhnout metodu, která umožní automaticky vytvářet aproximační obvody v alternativních reprezentacích. V kapitole 2 budou představeny jednotlivé reprezentace a techniky, které se používají pro jejich optimalizaci. Poté v kapitolách 3 a 4 bude představeno (kartézské) genetické programování jakožto metoda pro automatický návrh a optimalizaci obvodů. V kapitole 5 bude shrnuto, jak k problému návrhu aproximačních obvodů přistupují jiní autoři. Vytvořená implementace těchto metod bude přiblížena v kapitole 6. Experimenty v kapitole 7 se nejprve zaměřují na efektivní implementaci fitness funkce a poté zkoumají samotnou evoluci aproximačních obvodů v alternativních reprezentacích. Závěr práce je uveden v kapitole 8.

Kapitola 2

Optimalizace kombinačních obvodů

Důležitou součástí automatického návrhu je logická syntéza a optimalizace. Pojmem logická syntéza rozumíme transformaci obvodu logické funkce do podoby, která splňuje stanovená kritéria. Těmi jsou například omezení na počet uzlů, logických úrovní nebo energetické náročnosti. Uplatňuje se například na síť vzniklou zkompilováním VHDL popisu před tím, než je namapována na technologii konkrétního cílového zařízení. Další uplatnění lze nalézt například při předzpracování logických obvodů pro účely rychlého ověření funkční ekvivalence [6].

Klasické metody pro syntézu typicky sestávají z několika hlavních kroků. Mezi ně patří hledání společných logických celků, odstraňování redundantních uzlů a zjednodušování reprezentace jednotlivých uzlů. Zástupci těchto metod jsou systémy SIS [40] a MVSIS [11].

Klasické metody pro syntézu mají dle [32] hned několik nevýhod:

- Jsou založené převážně na metodě pokus-omyl a velkou roli hraje manuální ladění skriptů pro optimalizaci.
- Snaží se minimalizovat počet literálů v disjunktivní normální formě. Avšak počet literálů nemusí korelovat s velikostí obvodu po provedení mapování na cílovou technologii
- Jedná se o velmi komplexní systémy, které vyžadují dobré porozumění dané problematice a jsou časově náročné na implementaci.
- Jsou výpočetně náročné.

Ve zbytku této kapitoly budou představeny efektivnější metody logické syntézy, které těží převážně ze způsobu, jakým reprezentují obvody.

2.1 And-Inverter graf

Jednou z možných reprezentací kombinačních obvodů jsou tzv. And-Inverter grafy (AIG). Jedná se o dopředné acyklické grafy. Uzly mohou mít buď 2 nebo 0 vstupních hran. Uzel, který má 2 vstupní hrany reprezentuje dvouvstupé hradlo AND. Uzel bez vstupních hran představuje primární vstup. Některé z uzlů jsou označeny jako primární výstupy. Hrany mezi uzly jsou buď normální nebo mohou negovat příslušný signál. Libovolná logická funkce může být vyjádřena pomocí hradel AND, OR a NOT. Tato reprezentace může být převedena na AIG aplikací DeMorganových zákonů. [32]

Systém ABC

Reprezentace AIG se ukázala jako efektivní při řešení problémů z oblasti formální verifikace [27]. Jedním z nich je problém ověřování funkční ekvivalence dvou obvodů [6]. Nové poznatky z této oblasti motivovaly autory k vytvoření nástroje ABC, který slouží pro rychlou syntézu a formální verifikaci logických obvodů [8].

Hlavní výhody ABC oproti klasickým metodám ([40] a [11]) jsou dle autorů [32] následující:

- Navrhovaný systém je méně závislý na metodě pokus-omyl a ručním ladění parametrů.
- Hlavní metriky jsou počet uzlů a úrovní v AIG reprezentaci. Ty dávají přesnější odhad vlastností výsledného obvodu po provedení mapování na cílovou technologii.
- Je jednodušší a byl rychlejší na implementaci.
- Produkuje srovnatelné nebo lepší výstupy než klasické metody a je řádově mnohem rychlejší.

Nástroj ABC pracuje s AIG reprezentací a provádí rychlé lokální transformace v několika iteracích. Aby bylo možné vysvětlit základní princip činnosti ABC, bude nejprve představena používaná terminologie.

Při počáteční tvorbě AIG se využívá *strukturální hašování*. To umožňuje předcházet vytváření duplicitních kopií uzlů, které již existují. Funguje tím způsobem, že při vytváření nového AND uzlu se nejprve provede uspořádání dvojice vstupů (pro ošetření komutativity) a poté se vypočítá haš této dvojice. Pokud taková dvojice ještě neexistuje, je alokován nový uzel a vytvořen záznam v hašovací tabulce. Pokud existuje, tak se využije již existující uzel.

Řez C uzlu n je množina uzlů nazývaných *listy*, pro které platí, že existuje cesta z primárních vstupů do uzlu n procházející listem. Řez je *K-proveditelný*, pokud počet jeho listů nepřesahuje K. *Funkce řezu* je funkce, kterou vykonává uzel n. Dvě logické funkce F a G jsou *NPN-ekvivalentní*, pokud funkci F lze vytvořit z G pomocí negování (N) a permutování (P) vstupů a negování (N) výstupu.

AIG přepisování [32] je rychlý algoritmus pro minimalizaci velikosti AIG. Iterativně se snaží nahrazovat podgrafy vycházející z určitého uzlu za menší, ale funkčně ekvivalentní podgrafy. Algoritmus funguje následujícím způsobem. Prochází uzly v pořadí daném topologickým uspořádáním. Pro každý z nich zkoumá všechny 4-proveditelné řezy. Nejprve pro daný řez zjistí třídu NPN-ekvivalence. Poté zkouší podgraf daný tímto řezem nahradit za jiný NPN-ekvivalentní podgraf a to tak, aby došlo ke zmenšení celkového počtu uzlů v AIG. Ke zmenšení může dojít díky strukturálnímu hašování v případě, že nový podgraf je tvořen uzly, které se ve zkoumaném AIG již nacházejí. Tento algoritmus sice pracuje na lokální úrovni, ale díky opětovnému aplikování v několika iteracích je možné získat optimalizace na globální úrovni.

Pro optimalizaci počtu úrovní AIG se využívá vlastnosti, že AND uzly jsou asociativní. Provádí se aplikací transformace $a(bc) = (ab)c$.

Formát AIGER

Pro ukládání a přenášení obvodů v reprezentaci AIG byl vytvořen formát AIGER [5]. Knihovna pro práci s tímto formátem je veřejně dostupná. AIGER má dvě verze – binární a ASCII. ASCII se jednodušeji vytváří a je čitelný i pro člověka. Binární formát má přísnější

specifikaci, ale zabírá méně místa na disku a je určen pro programové zpracování. Příklad ASCII formátu a příslušného AIG je uveden na obrázku 2.1.

Specifikace formátu AIGER je následující. První řádek obsahuje hlavičku, která začíná řetězcem `aag` pro ASCII verzi nebo `aig` pro binární verzi. Poté následuje 5 nezáporných čísel `M`, `I`, `L`, `O`, `A`. Hodnota `M` specifikuje maximální hodnotu indexu, `I` počet vstupů, `L` počet paměťových buněk (pro sekvenční obvody), `O` počet výstupů a `A` počet uzlů AND.

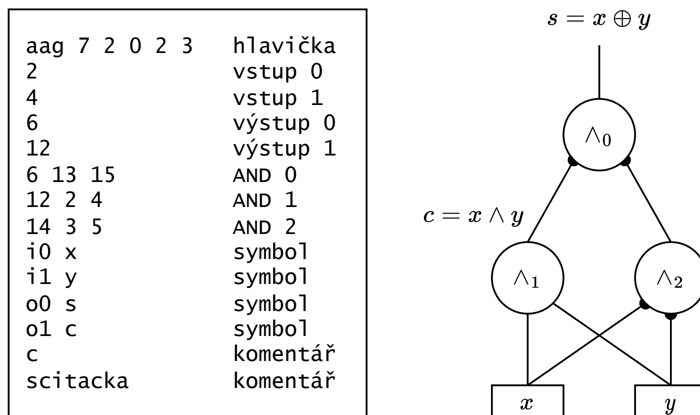
Literály mohou být konstanty nebo proměnné a jsou zapsány nezáporným číslem. Přičemž nejméně významný bit udává, zdali je daný literál negovaný nebo ne. Ostatní bity specifikují index. Literál x je negovaný pokud $x \mod 2 = 1$ a je mu přiřazen index $x/2$. Konstanta $False$ je vyjádřena literálem `0` a konstanta $True$ literálem `1`. Z čehož vyplývá že $True$ vznikne negací $False$.

Po řádku s hlavičkou následuje `I` řádků popisujících vstupy. Ty nemohou být specifikovány negovaně. Poté se pokračuje specifikací `L` paměťových buněk. Pro jejich popis se využívají dvě čísla. Dalších `O` řádků popisuje výstupy, které už mohou být negované.

V další části se nachází specifikace uzlů AND. Pro popis jednoho uzlu se využívají 3 čísla. První z nich musí být sudé a slouží pro přiřazení literálu k tomuto uzlu. Zbylé 2 čísla popisují literály, které slouží jako vstupy tohoto uzlu.

Další (nepovinnou) částí je tabulka symbolů. Symbol je libovolný tisknutelný ASCII řetězec. Mohou být přiřazeny ke vstupům (`i`), paměťovým buňkám (`l`) a výstupům (`o`). Cíl symbolu je tvořen identifikátorem (`i`, `l`, `o`) a pozicí, která říká kolikátému prvku v příslušném seznamu je tento symbol přidružen.

Poslední (nepovinnou) částí je komentář, který je uveden řádkem obsahujícím `c`.



Obrázek 2.1: V pravé části je uveden AIG pro 1bitovou poloviční sčítačku. Hrany zakončené vyplněným kolečkem jsou negované. V levé části je popis tohoto AIG v ASCII verzi formátu AIGER (pravý sloupec je pouze komentář a není součástí souboru). Rozhraní obvodu tvoří dva vstupy a dva výstupy. Například výstupu uzlu `AND 0` je přiřazen literál 6. Jeho vstupem jsou literály 13 a 15, což jsou negované verze literálů 12 a 14. Vstupům a výstupům jsou přiřazeny symboly popisující jejich význam. Komentář obsahuje název obvodu.

2.2 Majority-Inverter graf

Další posun v oblasti logické syntézy přinesly Majority-Inverter grafy (MIG) [1]. Jedná se opět o dopředné acyklické grafy. Uzly, které mají 0 vstupních hran jsou primární vstupy.

Uzly které mají 3 vstupy realizují operaci majorita definovanou jako $M(a,b,c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$. Hrany mohou být negované nebo normální.

Lze ukázat, že operace majorita se dá chápat jako zobecnění operace AND a OR. Pokud jeden ze vstupů majority je připojen na konstantu 0, tak realizuje operaci AND ($M(a,b,0) = a \wedge b$). V opačném případě, pokud je jedním ze vstupů konstanta 1, tak se chová jako OR ($M(a,b,1) = a \vee b$). Z čehož vyplývá, že libovolný AIG lze převést na MIG. Důkaz uveden v [1].

Jednou z motivací pro zkoumání obvodů reprezentovaných pomocí MIG jsou nově vznikající nanotechnologie, které poskytují velmi efektivní implementaci majority [3]. Zároveň se ukazuje, že díky odolnosti proti chybám se majorita dá efektivně uplatnit pro realizaci aproximačních obvodů [12] mapovaných na technologii kvantové tečky [36].

Axiomy Booleovské algebry

Pro manipulaci MIG lze použít přístupy používané v AND/OR grafech. Avšak pro lepší využití potenciálu operace majorita byla navrhnuta Booleovská algebra $(\mathbb{B}, M, ', 0, 1)$, kde M je třívstupá majorita a $'$ je negace. Axiomy pro tuto algebru jsou definován jako

$$\mathbf{\Omega} \begin{cases} \textbf{Komutativita} - \mathbf{\Omega.C} \\ M(x,y,z) = M(y,x,z) = M(z,y,x) \\ \textbf{Majorita} - \mathbf{\Omega.M} \\ \begin{cases} M(x,y,z) = x = y & \text{pokud } x = y \\ M(x,y,z) = z & \text{pokud } x = y' \end{cases} \\ \textbf{Asociativita} - \mathbf{\Omega.A} \\ M(x,u,M(y,u,z)) = M(z,u,M(y,u,x)) \\ \textbf{Distributivita} - \mathbf{\Omega.D} \\ M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z) \\ \textbf{Propagace negace} - \mathbf{\Omega.I} \\ M'(x,y,z) = M(x',y',z') \end{cases} \tag{2.1}$$

Bylo dokázáno, že se skutečně jedná o Booleovskou algebru, a že pomocí axiomů Ω lze libovolný MIG α převést na libovolný funkčně ekvivalentní MIG β [1]. Takovýto převod by však mohl obsahovat poměrně velký počet kroků. Proto pro účely optimalizace byly zavedeny transformace

$$\mathbf{\Psi} \begin{cases} \textbf{Relevance} - \mathbf{\Omega.R} \\ M(x,y,z) = M(x,y,z_{x/y'}) \\ \textbf{Komplementární asociativita} - \mathbf{\Omega.C} \\ M(x,u,M(y,u',z)) = M(x,u,M(y,x,z)) \\ \textbf{Substituce} - \mathbf{\Omega.S} \\ M(x,y,z) = M(v,M(v',M_{v/u}(x,y,z),u),M(v',M_{v/u'}(x,y,z),u')), \end{cases} \tag{2.2}$$

kde $a_{b/c}$ značí nahrazení všech výskytů b za c v a.

Optimalizace Majority-Inverter grafů

Autoři MIG navrhli algoritmy pro jejich optimalizace [1]. Cílem prvního z nich je získat MIG, který bude obsahovat co nejméně uzlů. Druhý algoritmus optimalizuje MIG z hlediska

zpoždění a snaží transformovat MIG tak, aby obsahoval co nejméně úrovní. Oba algoritmy iterativně aplikují transformace z $\boldsymbol{\Omega}$ a $\boldsymbol{\Psi}$. Jelikož se jedná se o heuristiky, které negarantují získání ideálního řešení, tak se může stát, že optimalizace uvázne v lokálním optimu. Z toho důvodu se stanovuje limit počtu iterací.

Pseudokód algoritmu pro minimalizaci velikosti je uveden v alg. 1. Postup je možné rozdělit do dvou fází. V první z nich (řádky 2 a 5) se uplatňují takové axiomy, které vedou na redukci počtu uzlů. Například aplikace axiomu $\Omega.M$ zleva doprava ($L \rightarrow P$) může potenciálně odstranit jeden uzel ($M(x, x, y) = x$). Tato fáze se nazývá *eliminace*. Účelem druhé fáze (ř. 3 a 4) je umožnit vyváznutí z lokálního optima a nazývá se *přeuspořádání*. Využívá takové axiomy, které přesouvají proměnné nebo dokonce i přidávají uzly ($\Psi.S(\alpha)$), ve snaze vytvořit příležitost pro eliminaci. Reprezentace MIG využívá i strukturální hašování představené v sekci 2.1.

Algoritmus 1: Optimalizace velikost MIG [1]

1 **for** i = *1,2,..., i_{max}* **do**
2 $\quad \Omega.M_{L\rightarrow P}(\alpha)$; $\Omega.D_{P\rightarrow L}(\alpha)$;
3 $\quad \Omega.A(\alpha)$; $\Psi.C(\alpha)$;
4 $\quad \Psi.R(\alpha)$; $\Psi.S(\alpha)$;
5 $\quad \Omega.M_{L\rightarrow P}(\alpha)$; $\Omega.D_{P\rightarrow L}(\alpha)$;

Algoritmus 2 popisuje způsob, jakým se provádí optimalizace zpoždění. Postup lze opět rozdělit do dvou částí. První z nich se nazývá *push-up* (řádky 2 a 5) a jejím cílem je přesun uzlů do vyšší úrovně a redukce počtu úrovní. Což může způsobit, že vzroste celkový počet uzlů. Druhá fáze *přeuspořádání* (ř. 3 a 4) je stejná jako při minimalizaci počtu uzlů.

Algoritmus 2: Optimalizace zpoždění MIG [1]

1 **for** i = *1,2,..., i_{max}* **do**
2 $\quad \Omega.M_{L\rightarrow P}(\alpha)$; $\Omega.D_{L\rightarrow P}(\alpha)$; $\Omega.A(\alpha)$; $\Psi.C(\alpha)$;
3 $\quad \Omega.A(\alpha)$; $\Psi.C(\alpha)$;
4 $\quad \Psi.R(\alpha)$; $\Psi.S(\alpha)$;
5 $\quad \Omega.M_{L\rightarrow P}(\alpha)$; $\Omega.D_{L\rightarrow P}(\alpha)$; $\Omega.A(\alpha)$; $\Psi.C(\alpha)$;

Metody pro optimalizaci logických obvodů se dají rozdělit do dvou skupin [7]. První skupina obsahuje algebraické metody, které jsou relativně rychlé a aplikují se iterativně, dokud není nalezeno zlepšení. Algoritmy 1 a 2 se řadí mezi algebraické. Ve druhé skupině se nachází Booleovské metody. Ty jsou výpočetně náročnější, avšak mají větší potenciál odhalit příležitosti pro optimalizaci, které nemusí být viditelné pro algebraické metody. Systémy používané v praxi kombinují oba přístupy [8][40][11].

Autoři [2] navrhují Booleovskou optimalizační metodu pro MIG využívají jejich inherentní odolnosti vůči chybám. Logicky chybný obvod vznikne takovou modifikací původního obvodu, která způsobí, že nový obvod bude realizovat rozdílnou logickou funkci. Prvním krokem je identifikace kritických uzlů, jež jsou vhodnými kandidáty pro optimalizaci. Ta spočívá ve vytvoření třech chybných variant zvoleného uzlu a jejich následné spojení pomocí majority. Chyby zavedené do jednotlivých variant jsou vzájemně ortogonální. Dvě chyby jsou vzájemně ortogonální, pokud neexistuje žádná vstupní kombinace, při které se projeví obě zároveň. Majorita spojující tyto varianty se dá chápat jako hlasovací člen. Díky ortogonalitě jsou vždy alespoň dvě vstupní hodnoty spočítány korektně a ty „přehlasují“ třetí, potenciálně chybný, vstup. Z čehož plyne, že takto upravený MIG je funkčně ekvivalentní. Chybné varianty jsou pravděpodobně jednodušší než ty korektní a na celý obvod mohou

být poté aplikovány algebraické metody, díky čemuž je možné získat obvod, který má menší velikost nebo zpoždění.

2.3 XOR-Majority graf

Zlepšení oproti Majority-Inverter grafům přineslo rozšíření o uzly realizující XOR. Vznikla tím reprezentace nazývaná XOR-Majority graf (XMG) [18]. Na rozdíl AIG a MIG, XMG již není homogenní (uzly nereprezentují stejnou funkci). Jsou to však stále dopředné acyklické grafy, kde uzly jsou buď primární vstupy (0 vstupních hran), XOR (2 vstupní hrany) nebo majorita (3 vstupní hrany). Hrany v grafu jsou buď normální nebo negované.

Jednou z motivací pro zahrnutí funkce XOR je skutečnost, že aritmetické obvody často obsahují funkci počítající paritu ze tří vstupů. Díky funkci XOR je možné paritu realizovat mnohem kompaktněji [13].

Optimalizace XOR-Majority grafů

Technika pro optimalizaci XMG představená v [18] spočívá v rozčlenění na podgrafy, nalezení optimální reprezentace pro každý podgraf a jejich následné spojení do výsledného XMG. Tento algoritmus je aplikován opakovaně až dokud nepřestane přinášet zlepšení.

Pro rozčlenění do podgrafů autoři využívají heuristiku představenou v [28]. Ta se snaží vstupní obvod nahradit co nejmenším počtem vyhledávacích tabulek s k vstupy (k-LUT). Takto vzniklá struktura se nazývá k-LUT obal.

Po nalezení obalu se pro každý prvek, braný v pořadí daném topologickým uspořádáním, vypočítá jeho optimální XMG reprezentace. Prvním krokem je identifikace třídy NPN-ekvivalence daného prvku. Poté se vypočítá optimální XMG reprezentace příslušné třídy pomocí algoritmu založeného na [41]. Optimalizovaný XMG podgraf se uloží do databáze, aby nemusel být vytvářen opakovaně. Takto lokálně optimalizované podgrafy se následně spojí do výsledného XMG.

Autoři ukázali, že větší hodnota k vede na lépe optimalizované obvody, jelikož umožňuje vytvořit obal s menším počtem prvků a tím pádem lokálně prováděné optimalizace mají větší vliv z globálního hlediska [18].

Byla navržena i algebraická optimalizační metoda [13]. Rozšiřuje postup používaný pro minimalizaci MIG popsaný v sekci 2.2. A to takovým způsobem, že přidává pravidla pro přepis podgrafů tvořených uzly XOR a majorita.

Funkce XOR je definována pomocí

$$\mathbf{\Delta} \begin{cases} x \oplus y = 0 & \text{pokud } x = y \\ x \oplus y = 1 & \text{pokud } x = \overline{y} \\ x \oplus 0 = x \\ x \oplus 1 = \overline{x} \end{cases} \tag{2.3}$$

Základní vlastnosti jsou asociativita ($\Phi.A$), komutativita ($\Phi.C$) a propagace negace ($\Phi.I$)

$$\mathbf{\Phi}\begin{cases}\textbf{Komutativita} - \mathbf{\Phi.C}\\ x \oplus y = y \oplus x\\ \textbf{Asociativita} - \mathbf{\Phi.A}\\ (x \oplus y) \oplus z = x \oplus (y \oplus z)\\ \textbf{Propagace negace} - \mathbf{\Phi.I}\\ \overline{x \oplus y} = \overline{x} \oplus y = x \oplus \overline{y}\\ x \oplus y = \overline{x} \oplus \overline{y}\end{cases} \tag{2.4}$$

Pravidla, která pracují jak s funkcí XOR, tak i s majoritou (značenou jako $\langle xyz \rangle$), jsou následující. Aplikací rovnice 2.5 zprava doleva lze potenciálně zredukovat počet uzlů ze 4 na 2. Podmínky pro její použití jsou poměrně striktní. Avšak kombinací se vztahy z Δ a Φ vznikne několik dalších variant (viz [13]), které lze uplatnit ve speciálních případech, kdy některá z proměnných x, y nebo z je konstanta nebo je rovna proměnné u. Rovnice 2.6 umožňuje odstranit negaci a zmenšit tak počet úrovní.

$$\langle xyz \rangle \oplus u = \langle (x \oplus u)(y \oplus u)(z \oplus u) \rangle \tag{2.5}$$

$$\langle xy(\overline{y} \oplus z) \rangle = \langle xy(x \oplus z) \rangle \tag{2.6}$$

Pro optimalizaci velikosti XMG se používají pouze některé vztahy používané pro MIG. Kvůli tomu, že XMG nejsou homogenní, tak se uplatňují jenom ty vztahy, které nevyžadují provedení substituce. Algoritmus 3 popisuje optimalizaci velikosti XMG. Využívá axiomy Δ, Ω a $\Psi.C$ představené v sekci 2.2. Jedná se o heuristický algoritmus, který se může zaseknou v lokálním optimu. Z toho důvodu se stanovuje maximální počet iterací i_{max}. V každé iteraci se provádí *eliminace* a *přeuspořádání*.

Algoritmus 3: Optimalizace velikost XMG [13]

1 **for** i = *1,2,...,* i_{max} **do**
2   eliminace(α);
3   přeuspořádání(α);
4   eliminace(α);
5 **def** *eliminace(α)*
6     $\Omega.M_{L\to P}(\alpha)$; $\Omega.D_{P\to L}(\alpha)$; `// MAJ`
7   Δ; `// XOR`
8   rov. $(2.5)_{P\to L}(\alpha)$; `// MAJ-XOR`
9 **def** *přeuspořádání(α)*
10     $\Psi.C(\alpha)$; `// MAJ`
11     $\Phi.A(\alpha)$; `// XOR`
12   rov. $(2.6)_{L\to P}(\alpha)$; `// MAJ-XOR`

Implementace popsaných metod je veřejně dostupná prostřednictvím nástroje CirKit[1]. Bylo ukázáno, že nejlepších výsledků lze dosáhnout kombinací obou metod [13]. Reprezentace XMG byla úspěšně aplikována také pro návrh kvantových algoritmů [43].

[1] `github.com/msoeken/cirkit`

Kapitola 3

Genetické programování

Původní genetický algoritmus (GA) [20] slouží především pro řešení optimalizačních úloh. Kandidátní řešení daného problému je při použití GA obvykle reprezentováno pomocí posloupnosti bitů. Způsob zakódování a s tím související délka této posloupnosti je předem stanovena uživatelem. Tento způsob zakódování není vhodný pro reprezentaci struktur, u kterých nemůžeme dopředu určit jejich délku. Příkladem jsou počítačové programy nebo elektronické obvody. Tímto problémem se zabýval J. Koza, který následně vytvořil genetické programování (GP) [25]. Jedná se opět o evoluční algoritmus inspirovaný principy Darwinovy evoluční teorie. GP pracuje s množinou kandidátních řešení nazývanou populace. Na tuto populaci se iterativně aplikují genetické operátory křížení a mutace. Tímto způsobem je prohledáván prostor možných řešení s cílem nalézt řešení, které nejlépe vyhovuje zadaným kritériím.

Algoritmus GP by se dal shrnout do těchto kroků:

1. Vytvoření počáteční populace kandidátních řešení.
2. Hlavní cyklus evoluce:
   (a) Ohodnocení všech kandidátních řešení v populaci.
   (b) Vytvoření následující generace:
      i. Selekce – výběr kandidátů pro reprodukci.
      ii. Reprodukce – kopírování kandidátů bez jejich modifikace.
      iii. Křížení – vytvoření nových řešení pomocí kombinace již existujících řešení.
      iv. Mutace – náhodná modifikace genů.
3. Cyklus je ukončen splněním ukončovací podmínky.
4. Výstupem je nejlepší nalezené řešení.

Tato kapitola čerpá z [15] a [4]. Jejím cílem je seznámit čtenáře se základy genetického programování.

3.1 Reprezentace kandidátních řešení

Genetické programování reprezentuje kandidátní řešení pomocí syntaktických stromů. Tyto stromy mohou mít teoreticky nekonečnou velikost. Jedná se o přirozený způsob, jak popsat například aritmetické výrazy nebo počítačové programy. Listy těchto stromů představují terminální symboly (konstanty, názvy proměnných nebo obecně funkce s nulovou

aritou). Uzly představují neterminální symboly (sčítání, logický součin, sinus, if-then-else nebo obecně funkce s nenulovou aritou). Množiny terminálních i neterminálních symbolů volí uživatel v závislosti na řešeném problému. Je žádoucí, aby obě množiny splňovaly následující podmínky:

První podmínka říká, že funkční množina by měla být uzavřená. To znamená, že všechny funkce by měly být definované pro veškeré hodnoty vzniklé vyhodnocením terminálního symbolu nebo aplikováním funkce. Příkladem množin, které tuto podmínku nesplňují, jsou množiny obsahující konstantu π a funkci log. součin (AND), která je definovaná pro pravdivostní hodnoty. Nesplnění této podmínky může vést na nedefinované chování nebo pád programu.

Druhá podmínka vyžaduje, aby množiny terminálních a neterminálních symbolů byly tvořeny takovými funkcemi, ze kterých bude možné sestavit hledané řešení. Uvažujme příklad, kdy cílem evoluce je navrhnout strukturu klasifikační neuronové sítě. Máme dvourozměrná data pocházející ze tříd A a B. Data jsou celkem ve čtyřech shlucích, které se nepřekrývají. Třídě A přísluší shluky se středy v bodech $[0, 0]$ a $[1, 1]$. Třída B má středy v bodech $[0, 1]$ a $[1, 0]$. Pokud by funkční množina obsahovala pouze perceptron s lineární aktivační funkcí, tak by nebylo možné data správně separovat a klasifikovat, jelikož rozhodovací hranice by byla vždy přímka. Přidání nelineární aktivační funkce sigmoidy umožní vytvořit neuronovou síť, která bude schopná data správně klasifikovat. [54]

3.2 Fitness funkce

Aby bylo možné objektivně porovnat, které kandidátní řešení je lepší pro daný problém, je potřeba definovat relaci uspořádání na množině kandidátních řešení. K tomu se využívá fitness funkce $f : X \rightarrow \mathbb{R}$, kde X je množina kandidátních řešení. Fitness funkce se volí tak, aby vyjadřovala míru vhodnosti daného kandidátního řešení pro zkoumaný problém. Pokud cílem evoluce je maximalizovat (resp. minimalizovat) fitness funkci, tak řešení A je lepší než řešení B pokud $f(A) > f(B)$ (resp. $f(A) < f(B)$).

Uvažujme příklad, kdy se pomocí evoluce na úrovni hradel navrhuje kombinační obvod, který má realizovat zadanou logickou funkci. Hledaný obvod je možné specifikovat například pomocí pravdivostní tabulky, která každé vstupní kombinaci přiřadí výstupní kombinaci. Fitness funkci f_0 je poté možné stanovit jako počet výstupních bitů, ve kterých se kandidátní řešení shoduje s požadovanou funkcí, při uvážení všech možných vstupních kombinací.

Je vhodné volit fitness funkci takovým způsobem, aby lepšímu řešení byla přiřazena nižší hodnota. Pokud ideální řešení má nulovou fitness hodnotu, tak fitness funkce se dá chápat jako vzdálenost kandidátního řešení od ideálního řešení. Funkce f_0 by se upravila tak, aby vyjadřovala počet chybně vypočítaných bitů (f_{std}). Fitness hodnota ideálního obvodu je poté nula.

Dále je dobré fitness funkci normalizovat tak, aby její obor hodnot byl interval $\langle 0; 1 \rangle$. Nechť funkce z příkladu výše má 3 1bit. vstupy (2^3 kombinací) a 2 1bit. výstupy. Kandidátní řešení musí správně vypočítat oba výstupní bity pro všech 8 vstupních kombinací. Maximální hodnota fitness funkce (f_{max}) je tedy 16. Normalizovaná fitness funkce by se dala formulovat jako $f_{norm} = f_{std}/f_{max}$.

Výpočet fitness funkce je v mnohých případech časově nejnáročnější částí algoritmu. Její definice má vliv na délku evoluce a kvalitu nalezených řešení. Proto je důležité věnovat zvláštní pozornost efektivitě její implementace. V kapitole 5 budou představeny příklady z oblasti aproximace obvodů.

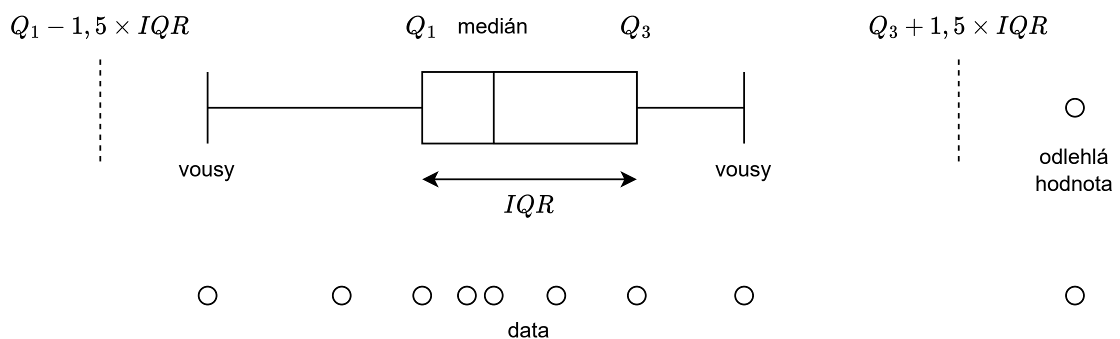
3.3 Ukončení a výsledek evoluce

Hlavní evoluční cyklus genetického programování se ukončuje po dosažení uživatelem zvolené podmínky. Problémy, u nichž známe analytické řešení, je možné ukončovat po nalezení řešení s optimální fitness hodnotou. Genetické programování se však často uplatňuje pro úlohy, kde není známa fitness hodnota ideálního řešení nebo se pracuje s omezeními, která brání dosažení ideální hodnoty fitness. V těchto případech se evoluce ukončuje po dosažení maximálního počtu generací nebo evaluací fitness funkce.

Kandidátní řešení s nejlepší fitness hodnotou nalezené v průběhu evoluce je prohlášeno za výsledek evolučního běhu.

3.4 Statistické vyhodnocení experimentů

Genetické programování je stochastický algoritmus využívající generátor pseudonáhodných čísel. Pro posouzení efektivity konkrétního nastavení algoritmu nestačí pouze jeden běh. Je vhodné provést několik (typicky minimálně 30) nezávislých běhů. Výsledky jsou poté zkoumány s využitím statistických metod. Pozoruje se primárně rychlost konvergence k ideálnímu řešení (a zdali vůbec konverguje) v závislosti na počtu generací. Neméně důležitá je kvalita výsledných řešení jednotlivých běhů. Pro grafickou reprezentaci se často využívají krabicové grafy (boxplot) popsané na obrázku 3.1.



Obrázek 3.1: Anatomie krabicového grafu. Q_1 značí první kvartil, Q_3 třetí kvartil, svislá čára mezi Q_1 a Q_3 medián. Hodnota IQR se vypočte jako $Q_3 - Q_1$. Odlehlé hodnoty leží mimo interval $\langle Q_1 - 1,5 \times IQR; Q_3 + 1,5 \times IQR \rangle$. Vousy sahají od první neodlehlé hodnoty k poslední. [26]

Kapitola 4

Kartézské genetické programování

V této kapitole bude představeno kartézské genetické programování (CGP). Jedná se o variantu genetického programování navrženou primárně pro evoluci elektrických obvodů. Své pojmenování dostal algoritmus podle dvourozměrné mřížky, pomocí níž reprezentuje kandidátní řešení. CGP bylo úspěšně aplikováno například pro návrh a optimalizaci aproximačních obvodů, obrazových filtrů a neuronových sítí [31]. Tato kapitola čerpá z [30].

4.1 Genotyp a fenotyp

Kandidátní řešení v CGP jsou orientované acyklické grafy. Uzly vykonávají výpočet a jsou organizovány do dvourozměrné mřížky. Pojmem genotyp se rozumí řetězec celých čísel nazývaných geny. Ty popisují propojení jednotlivých uzlů, jejich konkrétní funkci a také určují, které uzly jsou připojeny na výstupy obvodu. Uzly, které se nepodílí na výpočtu výstupních hodnot, mohou být ignorovány. Uzly, jež naopak jsou potřeba pro výpočet výstupu, se nazývají aktivní. Proces, v němž jsou identifikovány aktivní uzly a jednotlivým genům je přiřazena sémantika, se nazývá dekódování genotypu. Obvod (resp. program nebo funkce) vzniklá dekódováním genotypu se nazývá fenotyp. Genotyp v CGP má sice pevně stanovenou délku, avšak tato délka slouží pouze jako horní ohraničení velikosti fenotypu. Jeho skutečná velikost je dána počtem aktivních uzlů.

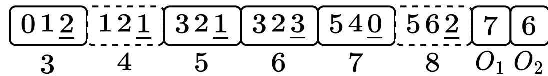
Množinu funkcí, které mohou jednotlivé uzly provádět, specifikuje uživatel. Každé funkci je přiřazen unikátní celočíselný identifikátor. První část genotypu je tvořena n-ticemi genů pro každý uzel. Pro kódování uzlů se využívají dva typy genů. První z nich se nazývá *funkční* a obsahuje identifikátor některého prvku z funkční množiny. Druhý typ genů je *propojovaní*. Ty obsahují adresy primárních vstupů (vstupy obvodu a terminální symboly) nebo uzlů, jejichž výstupy jsou připojeny jako vstupy aktuálního uzlu. Na vstup uzlu může být připojen pouze primární vstup nebo uzel, který se nachází v některém z předchozích sloupců. Počet propojovacích genů je stejný pro všechny uzly a je roven aritě funkce s největší aritou z funkční množiny. Z čehož plyne, že pro výpočet výstupu uzlu vykonávajícího funkci s menší než maximální aritou, není potřeba vyhodnocovat všechny připojené uzly. V CGP se využívá následující adresování. Primárním vstupům jsou přiřazeny adresy od 0 do $n_i - 1$, kde n_i je počet prim. vstupů. Uzly jsou organizovány do 2D mřížky a jejich výstupy jsou adresovány sekvenčně po sloupcích. První má adresu n_i a poslední $n_i + L_n - 1$, kde L_n je celková velikost mřížky. Druhou a poslední částí genotypu je n_o genů, které obsahují adresy uzlů připojených na výstupy obvodu (primární výstupy). Obrázek 4.1 obsahuje příklad kandidátního řešení.

Pro řízení velikosti mřížky a propojení uzlů slouží tři parametry: počet řádků n_r, počet sloupců n_c a konektivita l. Konektivita specifikuje, z kolika předchozích sloupců může uzel získávat vstupní hodnoty. Pokud $l = 1$, tak na vstup daného uzlu může být připojen pouze uzel z předchozího sloupce nebo primární vstup. Nejméně omezující nastavení z hlediska počtu možných topologií je $n_r = 1$ a $l = n_c$.

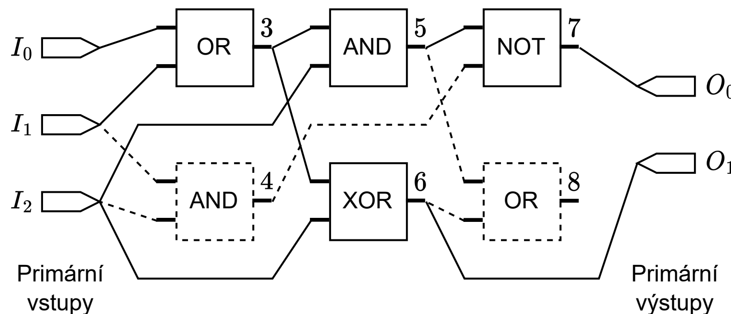
4.2 Ukázka zakódování obvodu

V této sekci je představen příklad zakódování kombinačního obvodu. Cílem evoluce je nelézt kombinační obvod, který bude realizovat zadanou logickou funkci. Obvod pracuje se třemi primárními vstupy I_0, I_1 a I_2 a produkuje dva primární výstupy O_0 a 0_1. Při evoluci byly parametry řídící velikost mřížky a konektivitu nastaveny na hodnoty: $n_r = 2$, $n_c = 3$ a $l = n_c$. Množina funkcí, které mohou uzly realizovat obsahuje hradla NOT (0), AND (1), OR (2) a XOR (3). Každé hradlo pracuje se dvěma vstupy. Jedinou výjimkou je hradlo NOT, která využívá pouze první vstup. Maximální arita je tedy 2 a proto jsou pro zakódování každého uzlu potřeba 3 geny – 2 propojovací a 1 funkční.

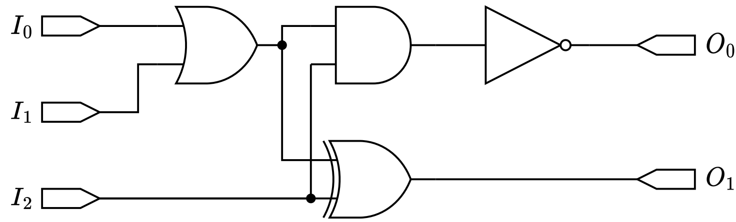
Obrázek 4.1 obsahuje genotyp zobrazený jako posloupnost číslic. Grafická reprezentace toho genotypu v CGP mřížce je uvedena na obrázku 4.2. Dekódováním se získá fenotyp (kombinační obvod) sestavený z hradel na obrázku 4.3.



Obrázek 4.1: Ukázka genotypu. Každý uzel je popsán trojicí genů – první dva jsou propojovací a třetí (podtržený) specifikuje funkci daného uzlu. Pod každou trojicí je uvedena adresa příslušného uzlu. Přerušovanou čarou jsou vyznačeny uzly, které nejsou aktivní a nepodílí se na výstupu. Poslední dva geny označují primární výstupy O_0 a 0_1.



Obrázek 4.2: Zobrazení genotypu z obr. 4.1 v CGP mřížce. Uzly jsou uspořádány do CGP mřížky, tvořené dvěma řádky a třemi sloupci. U každého uzlu je uvedena jeho adresa a funkce. Uzly a propojení vykreslené přerušovanou čarou nejsou aktivní. Uzel s adresou 7, realizuje funkci NOT. Arita této funkce je 1 a z toho důvodu je pouze první vstup označen jako aktivní.



Obrázek 4.3: Fenotyp příslušící genotypu z obr. 4.1. Tento kombinační obvod je sestaven ze 4 hradel. Primární výstupy počítají logické funkce $O_0 = \neg((I_0 \vee I_1) \wedge I_2)$ a $O_0 = (I_0 \vee I_1) \oplus I_2$.

4.3 Algoritmy pro dekódování genotypu

Dekódování genotypu sestává ze dvou částí. Úkolem prvního algoritmu je vytvořit seznam aktivních uzlů a zjistit jejich počet. Druhý algoritmus má za cíl vypočítat hodnoty primárních výstupů pro jednu konkrétní kombinaci vstupních hodnot.

Algoritmus pro tvorbu seznamu aktivních uzlů využívá pomocné pole NU, které říká, zdali je příslušný uzel aktivní. Velikost tohoto pole je rovna počtu uzlů. Prvním krokem je inicializace každého prvku na hodnotu $False$. Ve druhém kroku se provede průchod přes všechny geny primárních výstupů. Hodnoty těchto genů se použijí jako index do pole NU a indexované prvky se nastaví na hodnotu $True$. Ve třetím kroku se iteruje přes všechny uzly v sestupném pořadí počínaje uzlem s nejvyšší adresou. Pokud je hodnota pole NU příslušící tomuto uzlu nastavena na $True$, tak se vykoná průchod přes prvních a propojovacích genů daného uzlu, kde a je arita funkce uzlu. Pomocí hodnot propojovacích genů se zaindexuje pole NU a příslušné položky se nastaví na $True$. Pokud některý gen propojovací nebo primárního výstupu obsahuje adresu primárního vstupu, tak je přeskočen. V opačném případě je potřeba hodnotu použitou pro indexaci snížit o n_i (počet vstupů), jelikož uzlu s adresou n_i přísluší index 0. V posledním kroku jsou adresy všech uzlů, které jsou v poli NU označeny jako aktivní, vloženy do seznamu NP. Adresy budou seřazeny vzestupně. Výstupem algoritmu je seznam NP.

Algoritmus, který slouží pro výpočet primárních výstupů, sestává z následujících kroků. Nejprve je vytvořeno pole O. Jeho velikost je rovna $n_i + L_n$. Do prvních n_i položek jsou vloženy hodnoty primárních vstupů. Následujících L_n prvků slouží pro uložení výstupních hodnot jednotlivých uzlů. Poté je proveden průchod přes všechny aktivní uzly uložené v seznamu NP. Adresy operandů a identifikátor funkce jsou získány z genů daného uzlu. Hodnoty operandů se načtou z pole O. Výsledek aplikace funkce se uloží do pole O na adresu příslušící tomuto uzlu. Po vykonání všech aktivních uzlů se výsledek nachází na adresách specifikovaných geny primárních výstupů.

V této sekci byly algoritmy pro dekódování genotypů představeny tak, jak jsou uvedeny v [30]. V kapitole 5 budou popsány efektivnější metody.

4.4 Křížení a mutace

Hlavním genetickým operátorem, který se v CGP využívá pro prohledávání prostoru všech potenciálních kandidátních řešení, je bodová mutace. Funguje tím způsobem, že vybere náhodný gen a náhodně změní jeho hodnotu. Nová hodnota je vygenerována tak, aby byla validní. To znamená, že musí splňovat omezení představena v sekci 4.1. Může být zmutováno i více než jeden gen.

Počet genů, které mají být zmutovány, volí uživatel. Používají se dva hlavní způsoby zadávání tohoto parametru. První z nich (μ_r) je relativní a specifikuje, kolik procent genů má být podrobeno mutaci. Druhý způsob (μ_g) uvádí počet genů naopak jako absolutní hodnotu. Vztah obou parametrů se dá popsat výrazem $\mu_g = \mu_r L_g$, kde L_g je celkový počet genů v genotypu. Například, aplikací mutace na genotyp o velikosti 200 genů při nastavení $\mu_r = 5$ % bude změněno 10 genů.

V genetickém programování se uplatňuje i další genetický operátor – křížení. V CGP se však křížení typicky nepoužívá, jelikož se ukázalo, že nepřináší zlepšení a má spíše negativní efekt na evoluci [29].

4.5 Evoluční algoritmus

Evoluce v CGP je implementována evoluční strategií $1+\lambda$. Hodnota λ se většinou nastavuje na 4 [30]. Jednotlivé kroky jsou popsané algoritmem 4.

Aplikování mutačního operátoru ze sekce 4.4, může způsobit, že fenotyp potomka bude stejný jako fenotyp rodiče. Oba fenotypy budou mít tedy i stejnou fitness hodnotu. Tento jev se nazývá neutralita. Jeho využití lze pozorovat na řádku 7 algoritmu 4, který říká, že pokud neexistuje potomek, který by byl striktně lepší než rodič, tak je preferováno jako rodiče následující generace zvolit potomka, co má shodnou fitness hodnotu jako rodič.

Bylo ukázáno, že neutralita výrazně přispívá k efektivitě evolučního algoritmu [53]. Při evolučním návrhu tříbitových násobiček byla vyzkoušena alternativní evoluční strategie. V té se rodičem stal potomek pouze pokud byl striktně lepší. Výsledky experimentu ukazují, že při využití původní strategie, skončilo 27 ze 100 běhů úspěšným nalezením násobičky, přičemž ostatní běhy našly téměř korektní obvod. Kdežto v případě alternativní strategie nebyla nalezena žádná korektní násobička a průměrné fitness hodnoty byly horší.

Algoritmus 4: Evoluční strategie $1 + \lambda$ [30]

```
1  Náhodně inicializuj 1 + λ kandidátních řešení
2  Řešení s nejlepší fitness hodnotou je zvoleno jako rodič
3  while ideální řešení není nalezeno nebo není dosažen limit počtu generací do
4      for i = 1, 2, ..., λ do
5          Aplikací mutace na rodiče vytvoř potomka x_i
6          Vypočítej fitness hodnotu potomka x_i
7      if existuje potomek se stejnou nebo lepší fitness hodnotou než má rodič then
8          Nejlepší potomek je zvolen jako rodič následující generace
9      else
10         Rodič zůstává stejný
```

4.6 Alternativní implementace mutace

Aplikace bodové mutace může způsobit, že vznikne nový genotyp se stejnou fitness hodnotou jako jeho rodič. Tato situace nastane, pokud jsou zmutovány pouze geny neaktivních uzlů. Což má za následek, že dekódováním genotypu potomka vznikne fenotyp shodný s fenotypem jeho rodiče. A tedy opětovný výpočet fitness funkce je v tomto případě redundantní. Byly navrženy metody, které se snaží tento fenomén eliminovat.

Prvním možným řešením je detekování, že k této situaci došlo. Metoda skip [17] spočívá v porovnání genů rodiče a nově vzniklého potomka. Pokud se liší pouze v genech neak-

tivních uzlů, tak je potomkovi přiřazena fitness hodnota jeho rodiče. V opačném případě je potomek ohodnocený standardním způsobem. Průběh evoluce bude při použití této metody nezměněn. Přínos spočívá v ušetření procesorového času, který může být využit pro ohodnocení nových kandidátních řešení. Tato metoda se uplatní nejvíce v případech, kdy se využívá relativně nízká pravděpodobnost mutace a kandidátní řešení neobsahují velký podíl aktivních uzlů.

Druhé řešení spočívá v předcházení vzniku funkčně ekvivalentních potomků. Metoda single [17] modifikuje způsob, jakým se aplikuje bodová mutace. A to tak, že po každé aplikaci jedné bodové mutace se provedou následující kroky. Nejprve se ověří, jestli byly ovlivněny aktivní uzly. Pokud ne, tak aplikuje další bodovou mutaci. Tento cyklus se opakuje do té doby, než je ovlivněn první aktivní uzel. Takto vytvořený genotyp bude po dekódování odlišný od fenotypu jeho rodiče. Tato metoda modifikuje průběh evoluce. Její výhoda spočívá v garanci nových kandidátních řešení. Další výhodou je, že při jejím použití není nutné nastavovat pravděpodobnost mutace.

V literatuře byl představen mutační operátor navržený specificky pro evoluci aproximačních obvodů – SagTree [10].

Kapitola 5

Evoluční návrh aproximačních obvodů

Existence aplikací, které jsou inherentně odolné vůči nepřesnostem, vytváří prostor pro vznik systémů, jež umožňují ušetřit výpočetní zdroje za cenu snížené kvality výstupu.

Vhodnou příležitost pro úsporu, dle [38], nabízí aplikace, které pracují s analogovými vstupy nebo výstupy. Příkladem jsou různé obrazové a zvukové filtry, kde vstupem jsou data z reálného světa obsahující šum a výstupy jsou určeny pro interpretaci lidmi, pro které drobné nepřesnosti nejsou pozorovatelné. Dále je příhodné, pokud existuje více než jeden korektní výstup nebo se jedná o iterační algoritmus, který může být předčasně ukončen.

Odvětví, které se zabývá návrhem takovýchto systému, se nazývá aproximační výpočty. Podrobný přehled používaných technik lze nalézt v [33] a [44].

Na návrh aproximačních obvodů lze pohlížet jako na vícekriteriální optimalizační úlohu. Cílem je nalézt takový obvod, který bude poskytovat kompromis mezi přesností a ostatními hodnotícími kritérii. Typicky je snaha maximalizovat přesnost a minimalizovat velikost na čipu, zpoždění a spotřebu energie [38]. Řešením je celá množina obvodů, které mají různé vlastnosti a tvoří tzv. Pareto frontu. Její definice je následující [38]. Nechť $f_1(a), \ldots, f_k(a)$, kde $f_i(a) : \mathbb{A} \to \mathbb{R}$, k je počet kritérií a $a \in \mathbb{A}$ je kandidátní řešení z množiny kandidátních řešení $\mathbb{A}$. Cílem optimalizace je minimalizovat všechna kritéria f_i. Řešení $a \in \mathbb{A}$ dominuje $b \in \mathbb{A}$ pokud $\forall f_i : f_i(a) \leq f_i(b)$ a zároveň $\exists f_j : f_j(a) < f_j(b)$. Potom Pareto fronta je množina takových řešení z $\mathbb{A}$, která nejsou dominována žádný řešením z $\mathbb{A}$.

Jednou z možností, jak přistupovat k návrhu aproximačních obvodů, jsou evoluční algoritmy (představeny v kapitole 3). První publikací věnující se evolučnímu návrhu aproximačních obvodů na úrovni hradel je [39]. Autoři využívají kartézské genetické programování (CGP) pro návrh sčítaček. Snaží se minimalizovat chybu výsledného obvodu. Ostatní kritéria stanovují jako omezující podmínky (například maximální velikost obvodu lze specifikovat nastavením velikosti mřížky CGP). Ačkoliv bylo řečeno, že se jedná o vícekriteriální optimalizační úlohu, tak bylo ukázáno, že využití algoritmu, který se snaží přímo budovat Pareto frontu (jako je např. NSGA-II [14]), není příliš efektivní v porovnání s klasickým CGP, kde je fitness funkce zvolena jako váhovaný součet jednotlivých hodnotících funkcí (nebo kde optimalizujeme pouze jedno kritérium a ostatní tvoří omezující podmínky) [50] [51]. Je vhodné inicilizovat populaci již existujícím plně funkčním řešení, jelikož to výrazně zkrátí dobu evolučního běhu v porovnání s náhodně inicializovanou populací [51]. Rozsáhlý přehled o využití evolučních algoritmů pro návrh aproximačních obvodů lze nalézt v [38].

Ve zbytku kapitoly budou představena konkrétní hodnotící kritéria a způsoby, jak je efektivně implementovat. V závěru bude vysvětlen genetický operátor navržený speciálně pro aproximační obvody.

5.1 Hodnotící kritéria

Existuje několik různých způsobů, jak kvantifikovat rozdíl mezi výstupy korektních a aproximovaných obvodů. Ty mohou být založeny na Booleovské ekvivalenci, Hammingově vzdálenosti, absolutní vzdálenosti, kvadratické vzdálenosti nebo relativní vzdálenosti. Pro každou míru lze specifikovat variantu pro nejhorší případ a pro průměrný případ. Následující definice jsou převzaty ze studie porovnávající tyto chybové metriky implementované pomocí různých formálních metod [47].

Nejprve je potřeba definovat základní pojmy používané v této části. Nechť $f : \mathbb{B}^n \to \mathbb{B}^m$ je Booleovská funkce s n vstupy a m výstupy realizovaná korektním obvodem F. Funkce $\hat{f} : \mathbb{B}^n \to \mathbb{B}^m$ je její aproximace obvodem $\hat{\mathrm{F}}$. Dále nechť f_i je výstup i (číslováno od 0) funkce f s m výstupy.

Definice 5.1 *Nechť* $[\![P]\!] = 1$ *pokud výraz* P *je pravdivý, jinak* $[\![P]\!] = 0$. *Ekvivalence dvou funkcí je značena pomocí*

$$[\![f(x) \neq \hat{f}(x)]\!] = \bigvee_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \tag{5.1}$$

Definice 5.2 *Nechť* $nat(x)$ *je funkce* $nat : \mathbb{B}^m \to \mathbb{Z}$, *která vrací celé číslo příslušící binárnímu vektoru* x. *Její definice je* $nat(x) = \sum_{i=0}^{m-1} 2^i x_i$.

Definice 5.3 *Nechť* $x, y \in \mathbb{B}^n$ *jsou binární vektory. Vektor* x *je lexikograficky menší než* y, *značeno* $x < y$, *pokud existuje* k, $0 \leq k < n$, *takové, že* $y_k > x_k$ *a* $x_i = y_i$ *pro všechna* $i < k$. *Potom* $nat(x) < nat(y)$.

Následující metriky se považují za obecné, jelikož všem výstupním bitům dávají stejnou váhu. Lze je použít na libovolný logický obvod. Avšak pro aritmetické obvody nemusí být vhodné.

Definice 5.4 *Pravděpodobnost chyby* (e_{prob}) *je definována jako podíl vstupních kombinací, pro které dávají obvody různé výstupy.*

$$e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} [\![f(x) \neq \hat{f}(x)]\!] \tag{5.2}$$

Definice 5.5 *Hammingova vzdálenost v nejhorším případě* (e_{bf}) *udává maximální počet bitů, ve kterých se výstupy obvodů liší.*

$$e_{bf}(f, \hat{f}) = \max_{\forall x \in \mathbb{B}^n} \left(\sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \right) \tag{5.3}$$

Definice 5.6 *Průměrná Hammingova vzdálenost* (e_{mhd}) *vyjadřuje průměrný počet bitů, ve kterých se výstupy obvodů liší.*

$$e_{mhd}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \tag{5.4}$$

Pro aritmetické obvody byly zavedeny následující metriky. Vyznačují se tím, že na výstup pohlíží jako na přirozené číslo, což znamená, že jednotlivým bitům přikládají různou důležitost.

Definice 5.7 *Maximální absolutní vzdálenost (e_{wce}).*

$$e_{wce}(f, \hat{f}) = \max_{\forall x \in \mathbb{B}^n} |nat(f(x)) - nat(\hat{f}(x))| \tag{5.5}$$

Definice 5.8 *Průměrná absolutní vzdálenost (e_{mae}).*

$$e_{mae}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} |nat(f(x)) - nat(\hat{f}(x))| \tag{5.6}$$

Definice 5.9 *Průměrná kvadratická odchylka (e_{mse}).*

$$e_{mse}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \Big(nat(f(x)) - nat(\hat{f}(x))\Big)^2 \tag{5.7}$$

Definice 5.10 *Maximální relativní chyba (e_{wcre}).*

$$e_{wcre}(f, \hat{f}) = \max_{\forall x \in \mathbb{B}^n} \frac{|nat(f(x)) - nat(\hat{f}(x))|}{nat(f(x))} \tag{5.8}$$

Definice 5.11 *Průměrná relativní chyba (e_{mre}).*

$$e_{mre}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \frac{|nat(f(x)) - nat(\hat{f}(x))|}{nat(f(x))} \tag{5.9}$$

Aby bylo možné porovnávat obvody s různým počtem výstupních bitů pomocí aritmetických metrik, tak se provádí jejich normalizace

$$e_{wce\%} = \frac{1}{2^m} e_{wce}, \; e_{mae\%} = \frac{1}{2^m} e_{mae}, \; e_{mse\%} = \frac{1}{2^{2m}} e_{mse}, \tag{5.10}$$

kde 2^m odpovídá rozsahu hodnot určeného výstupním bitem m.

Bylo ukázáno, že e_{wce} je významná metrika [47]. Neboť koreluje s metrikami důležitými pro praktické použití, jako jsou e_{mae} a e_{mre}. Jedná se také o dobrý indikátor nároků na procesorový čas potřebný k výpočtu těchto metrik.

5.2 Paralelní simulace obvodů

V průběhu evolučního běhu CGP se nejvíce procesorového času spotřebuje typicky uvnitř fitness funkce na výpočtu jednoho z předešlých hodnotících kritérií. Z toho důvodu je žádané vytvořit takovou implementaci, která bude co nejrychlejší. Jednou ze základních optimalizací je tzv. paralelní simulace. Ta je vhodná zejména pro menší obvody. Hlavní myšlenka spočívá ve využití simulátoru, který umožní simulovat zkoumaný fenotyp pro několik vstupních kombinací paralelně. V literatuře lze najít implementace využívající grafické karty pro evaluaci klasifikátorů [19] nebo FPGA pro obrazové filtry [49]. Ve zbytku této sekce bude představena metoda implementovatelná na běžných procesorech.

Při provádění simulace na běžných procesorech lze paralelismus implementovat na úrovní bitů, dat, vláken nebo procesů. Porovnáním jednotlivých variant se zabývá [21], přičemž jednotlivé varianty implementuje následovně.

Hlavní myšlenka paralelismu na úrovni bitů je následující. Namísto jednotlivých vstupních kombinací jsou na vstupy obvodu přiloženy bitové vektory odpovídající příslušné vstupní proměnné (tj. sloupce pravdivostní tabulky). Uzly CGP poté pracují s celými vektory. Na 64bitovém procesoru je možné pomocí jedné instrukce zpracovat až 64 bitů. Takže je teoreticky možné dosáhnout až 64násobného zrychlení. Tento princip je dále rozšířen pomocí datového paralelismu (SIMD). Moderní procesory poskytují vektorové instrukce. Díky AVX2 je možné v jedné instrukci zpracovat až 256 bitů.

Zajímavá varianta paralelní simulace je představena v [52], kde autoři navrhují namísto interpretace zkompilovat genotyp a poté jej spouštět.

Paralelismus na úrovni vláken využívá toho, že každé kandidátní řešení v populaci je možné vytvořit a vyhodnotil nezávisle na těch ostatních. Takže je spuštěno tolik vláken, kolik je kandidátů v populaci. Každé vlákno vytvoří kopii rodiče, aplikuje na ni mutační operátor a následně provede výpočet fitness funkce. Při volbě rodiče pro další generaci je potřeba zvolit vhodnou strategii komunikace mezi vlákny.

Implementace využívající více procesorů funguje tím způsobem, že na každém procesoru je spuštěn evoluční běh a po určitém počtu generací se provede výměna nejlepších nalezených řešení. Každý procesor poté zahájí nový evoluční běh inicializovaný globálně nejlepším doposud nalezeným řešením.

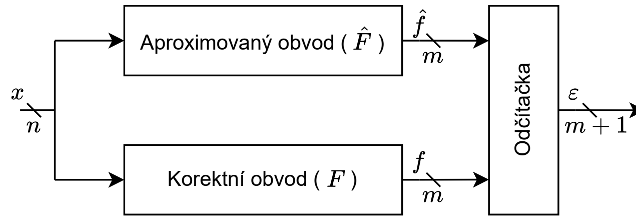
Problém všech těchto metod tkví ve škálovatelnosti [47]. Tyto techniky poskytují konstantní zrychlení v závislosti na počtu primárních vstupů. Vyžadují iterování přes všechny vstupní kombinace, jejichž počet však roste exponenciálně s počtem primárních vstupů.

5.3 Aplikace formálních metod

Alternativou k paralelní simulaci jsou metody, které využívají nástroje pro formální analýzu. Příkladem jsou SAT solvery a redukované uspořádané binární rozhodovací diagramy (ROBDD nebo jen BDD). Oba přístupy budou stručně představeny v této části. Detailní přehled a porovnání lze nalézt v [47], kde je ukázáno, že tyto postupy umožňují přesně vypočítat chybové metriky pro mnohem větší a složitější obvody.

Hlavní myšlenka spočívá v převodu výpočtu chybové metriky na problém splnitelnosti (SAT, #SAT). Dle [47] lze SAT definovat následovným způsobem. Nechť $g : \mathbb{B}^n \to \mathbb{B}$ je Booleovská funkce. Úkolem je nalézt libovolný model $a \in \mathbb{B}^n$, pro který $g(a) = 1$, nebo oznámit, že žádné takové a neexistuje. Úloha, kde cílem je zjistit počet $a \in \mathbb{B}^n$, pro které $g(a) = 1$, se nazývá #SAT. V následujícím textu bude výraz SAT(g) označovat predikát, který je pravdivý pouze pokud $\exists a \in \mathbb{B}^n : g(a) = 1$. Výrazem #SAT($g$) bude chápána procedura počítající $\sum_{\forall x \in \mathbb{B}^n} [\![g(x) = 1]\!] = \sum_{\forall x \in \mathbb{B}^n} g(x)$.

ROBDD jsou orientované acyklické grafy používané pro reprezentaci Booleovských funkcí. Jsou tvořeny rozhodovacími a listovými uzly. Právě jeden z uzlů je označen jako kořenový. Rozhodovací uzel reprezentuje vstupní proměnnou a vychází z něj dvě hrany. Jedna pro kladné ohodnocení dané proměnné a druhá pro záporné ohodnocení. Listové uzly jsou právě dva a odpovídají hodnotám log. 0 a 1. Uspořádané BDD znamená, že pořadí proměnných na každé cestě od kořene do listových uzlů odpovídá zadanému uspořádání. Redukované BDD vyžaduje, aby všechny isomorfní podgrafy byly sloučeny. Pokud jsou dvě Booleovské funkce funkčně ekvivalentní, tak při daném uspořádání proměnných budou jejich ROBDD isomorfní. Vyřešení problému SAT pro funkci g zadanou jako ROBDD je triviální. Stačí



Obrázek 5.1: Schéma základu porovnávacího obvodu používaného pro výpočet maximální absolutní chyby e_{wce}. Tento obvod počítá rozdíl výstupů porovnávaných obvodů F a $\hat{F}$, které realizují Booleovské funkce $\mathbb{B}^n \rightarrow \mathbb{B}^m$. Výsledný rozdíl ε je kódován ve dvojkovém doplňku pomocí $m+1$ bitů.

vypočítat libovolnou cestu, která končí v log. 1 a začíná v kořenovém uzlu. Problém #SAT se liší tím, že místo jedné cesty je potřeba najít všechny.

Aby bylo možné aplikovat formální metody, tak je potřeba vytvořit tzv. porovnávací obvod (v literatuře označovaný jako „miter“ [47]). Jeho struktura se mírně liší podle počítané chybové metriky, ale hlavní myšlenka zůstává stejná. Obsahuje korektní obvod, aproximovaný obvod a obvod, který má na vstupu výstupy obou předchozích obvodů a počítá chybovou metriku. Následně se porovnávací obvod převede na ROBDD nebo do konjunktivní normální formy, kterou vyžaduje SAT solver.

Při výpočtu obecných chybových metrik se odpovídající výstupy porovnávaných obvodů spojí pomocí funkce XOR. Pokud se pro zadanou vstupní kombinaci porovnávané obvody liší v hodnotě výstupního bitu na pozici i, tak hodnota výstupního bitu na pozici i takto upraveného obvodu bude 1. Spojením výstupů všech XORů pomocí funkce OR vznikne obvod, jehož výstupem bude 1, pokud se budou výstupy porovnávaných obvodů lišit. Řešením SAT pro tento obvod se zjistí, zdali porovnávané obvody jsou funkčně ekvivalentní nebo ne. Při použití #SAT bude výstupem počet vstupních vektorů, u kterých se porovnávané obvody neshodnou na výsledku. Když se výsledek #SAT vydělí počtem všech možných vstupních vektorů, tak dostaneme pravděpodobnost chyby (e_{prob}).

Pro výpočet aproximačních chybových metrik aritmetických obvodů se uvnitř porovnávacího obvodu počítá rozdíl korektního a aproximovaného obvodu, na který je poté aplikována absolutní hodnota nebo druhá mocnina atd. V rámci experimentů v této práci bude použita maximální absolutní chyba e_{wce}. Takže nyní budou detailně představeny algoritmy pro její výpočet s využitím formálních metod.

První částí výpočtu e_{wce} je porovnávací obvod uvedený na obrázku 5.1. Ten počítá rozdíl výstupů korektního a aproximovaného obvodu. Výsledek je kódován ve dvojkovém doplňku a samozřejmě může být i záporný.

Základní algoritmus pro výpočet e_{wce} by pokračoval přidáním obvodu, který by aplikoval absolutní hodnotu na získaný rozdíl. Převod záporného čísla kódovaného ve dvojkovém doplňku na kladné číslo se provede negací všech bitů a přičtením jedničky. Negace záporného čísla se dá realizovat tak, že na všechny jeho bity se aplikuje funkce XOR se znaménkovým bitem. Výstup tohoto podmíněného negování se připojí na vstup sčítačky, která přičte znaménkový bit. Výsledkem je absolutní hodnota rozdílu. Dále by algoritmus pokračoval analýzou tohoto obvodu. Bylo ukázáno, že nejnáročnější části na analýzu jsou výpočet rozdílu a absolutní hodnoty [22]. Obvod pro absolutní hodnotu obsahuje značné množství funkcí XOR, což vede na poměrně komplexní ROBDD. Z toho důvodu byly hledány efektivnější algoritmy.

Byla představena optimalizovaná implementace výpočtu e_{wce} využívající ROBDD, jejíž hlavní myšlenka spočívá ve vynechání absolutní hodnoty a následné konstrukci větve pro kladný a pro záporný rozdíl [34]. Algoritmus 5 popisuje tuto metodu. Vstupem jsou ROBDD porovnávaných funkcí f a $\hat{f}$ a výstupem je hodnota maximální absolutní chyby (e_{wce}). Nejprve se vytvoří obvod ε, který počítá rozdíl f a $\hat{f}$. Poté na řádcích 2–7 se provede konstrukce větve pro případ, kdy je rozdíl kladný. Proměnná μ_p obsahuje obvod, který se bude postupně rozšiřovat a analyzovat. Výpočet wce je založen na binárním prohledávání. Začíná se u MSB a postupuje se k méně významným bitům. Pokud SAT($\mu_p \wedge \varepsilon_i$) uspěje, tak to znamená, že existuje nějaký vstupní vektor, který produkuje takový rozdíl, který je kladný a na pozici s váhou 2^i má jedničku. Vzhledem k tomu, že μ_p se inicializuje negovaným znaménkovým bitem $\neg\varepsilon_{sign}$, tak predikát SAT na řádku 5 uspěje pouze pro ty rozdíly, které mají nulový znaménkový bit (jsou kladné). Lze tedy říct, že se jedná o hledání maxima rozdílu. Následně se na ř. 8–13 provede analýza záporné větve, která probíhá analogicky s tou odlišností, že se pracuje s negovanými bity rozdílu. Jedná se o hledání minima rozdílu. Jelikož se pracuje ve dvojkovém doplňku, tak je potřeba ještě provést korekci přičtením jedničky k výsledku ze záporné větve. Výstupem je poté maximum z výsledků kladné a záporné větve.

Kromě maximální absolutní chyby se [34] zabývá i efektivní implementací výpočtu průměrné absolutní chyby e_{mae}.

Algoritmus 5: Výpočet maximální absolutní chyby (e_{wce}) [34]

Vstup : ROBDD funkcí f a $\hat{f}$
Výstup: Maximální absolutní chyba e_{wce}

1 $\varepsilon \leftarrow \text{subtract}(f, \hat{f})$
2 $\mu_p \leftarrow \neg\varepsilon_{sign}$
3 $wce_p \leftarrow 0$
4 **for** $i \leftarrow |\varepsilon| - 2$ **to** 0 **do**
5   **if** $SAT(\mu_p \wedge \varepsilon_i)$ **then**
6         $wce_p \leftarrow wce_p + 2^i$
7         $\mu_p \leftarrow \mu_p \wedge \varepsilon_i$
8 $\mu_n \leftarrow \varepsilon_{sign}$
9 $wce_n \leftarrow 0$
10 **for** $i \leftarrow |\varepsilon| - 2$ **to** 0 **do**
11   **if** $SAT(\mu_n \wedge \neg\varepsilon_i)$ **then**
12         $wce_n \leftarrow wce_n + 2^i$
13         $\mu_n \leftarrow \mu_n \wedge \neg\varepsilon_i$
14 **return** $\max(wce_p, wce_n + 1)$

Při evoluční aproximaci obvodů však nemusí být nutné znát přesnou hodnotu maximálního absolutního rozdílu. Pokud je chybová metrika chápána pouze jako prahová hodnota, která nemá být překročena, tak lze analýzu zjednodušit.

Základem je opět porovnávací obvod z obrázku 5.1. Výstup odčítačky se připojí na komparátor, který rozhodne, zdali je absolutní hodnota rozdílu menší nebo větší než předem specifikovaná prahová hodnota. Ani v tomto případě není potřeba vytvářet obvod pro výpočet absolutní hodnoty. Lze využít řešení, kde se kladné a záporné rozdíly zkoumají odděleně. Autoři [9] navrhují efektivní implementaci komparátoru pro porovnání s konstantní

prahovou hodnotou. Ten lze realizovat z hradel AND a OR. Obvod, který porovnává bitový vektor A s konstantní bitovým vektorem T je popsán výrazem:

$$A > T \equiv \bigvee_{0 \leq i \leq k-1 \ \wedge \ T_i = 0} \left(A_i \wedge \bigwedge_{i < j \leq k-1 \ \wedge \ T_i = 1} A_j \right). \tag{5.11}$$

Při zadané prahové hodnotě $\mathcal{T}$ platí, že $e_{wce} > \mathcal{T}$ pokud rozdíl ε je kladný a $\varepsilon > \mathcal{T}$ nebo pokud je rozdíl ε záporný a $-\varepsilon > \mathcal{T}$. Jelikož se pracuje s čísly ve dvojkovém doplňku, tak druhá podmínka je ekvivalentní s $\neg\varepsilon > (\mathcal{T} - 1)$. Ověření, zdali e_{wce} zkoumaného obvodu překračuje určenou hranici $\mathcal{T}$, lze provést jediným voláním procedury SAT pro výraz:

$$SAT\left((\neg\varepsilon_{sign} \wedge (\varepsilon > \mathcal{T})) \vee (\varepsilon_{sign} \wedge (\neg\varepsilon > (\mathcal{T} - 1)))\right). \tag{5.12}$$

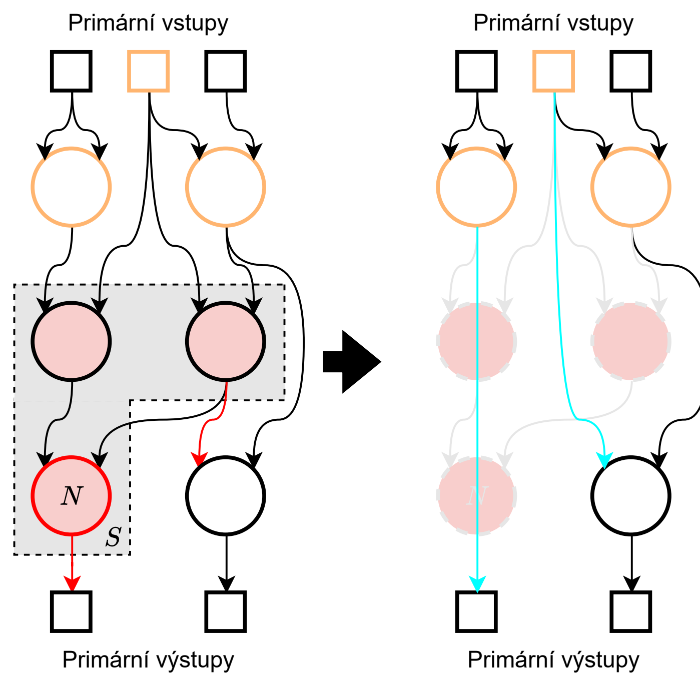
Příklad aplikace SAT solveru pro výpočet e_{wce} při evoluční aproximaci až 32bitových násobiček lze nalézt v [9]. Autoři představují evoluční strategii, která dává prioritu těm obvodům, jejichž chybovou metriku je možné ověřit ve stanoveném časovém limitu. Díky tomu jsou schopni prozkoumat podstatně víc kandidátních řešení a získat obvody s lepšími parametry.

5.4 Specializovaný mutační operátor

Byl publikován mutační operátor, který je navržen specificky pro evoluční aproximaci obvodů [10]. Autoři porovnávali existující mutační operátory, které rozdělili do dvou kategorií. První z nich jsou klasické mutační operátory (viz sekce 4.4 a 4.6) a druhou kategorii tvoří operátory, které deaktivují část obvodu. Ukázalo se, že deaktivační operátory jsou velmi efektivní při malém počtu generací, avšak poté naráží na lokální optimum a pomocí deaktivací nejsou schopné přinést další zlepšení. Na druhou stranu klasické operátory mají sice pomalejší start, ale při dostatečně vysokém počtu generací dosahují lepších výsledků.

Na základě těchto poznatků autoři navrhují sloučit nejlepší mutační a deaktivační operátor do jednoho kombinovaného operátoru. Výsledkem je operátor s názvem SagTree. Ten podle zadané pravděpodobnosti alternuje mezi mutační a deaktivační částí. Mutační část náhodně vybere jeden gen v aktivním uzlu a ten náhodně změní. Deaktivační část náhodně vybere aktivní uzel N a k němu příslušící podgraf S. Tento podgraf je tvořen z aktivních uzlů, které jsou připojeny jako vstupy uzlu N. Přidávání aktivních uzlů do S se rekurzivně opakuje a končí ve chvíli, kdy je dosažena předem stanovená maximální hloubka d. Poté je podgraf S odpojen takovým způsobem, že všechny uzly, které mají na vstupu připojený některý uzel z S, nahradí ovlivněný vstupní signál za náhodně zvolený vstup podgrafu S. Činnost deaktivační část je demonstrována na obrázku 5.2.

Autoři v experimentech nastavují pravděpodobnost mutační části na 25, 50 nebo 75 % a maximální hloubku deaktivovaného podgrafu volí náhodně z intervalu od 1 do 5.



Obrázek 5.2: Příklad aplikace deaktivační části operátoru SagTree. Červenou barvou jsou označeny uzly v deaktivovaném podgrafu S a signály které vychází z deaktivovaných uzlů. Signály (červené), které vychází z S, jsou nahrazeny náhodnými vstupy S (oranžové). Výsledkem nahrazení jsou modré signály. Převzato z [10], zjednodušeno.

Kapitola 6

Návrh a implementace nástroje pro evoluční aproximaci obvodu

V této kapitole bude popsána implementace metody pro evoluční návrh aproximačních obvodů v alternativních reprezentacích. Vytvořený systém je rozdělený na několik částí. První z nich je generátor aritmetických obvodů. Vygenerované obvody poté vstupují do programu pro převod do alternativních reprezentací. Nové reprezentace se následně použijí jako počáteční obvod pro CGP, jehož cílem je nalézt aproximovanou verzi obvodu. Kandidátní řešení získána pomocí evoluce jsou na závěr mapována na cílovou technologii (např. LUT).

Jako hlavní programovací jazyk byl vybrán jazyk C++ ve standardu C++20. Tento jazyk byl zvolen převážně z toho důvodu, že existuje řada knihoven implementovaných v C++, které budou v tomto projektu využity pro realizaci některých částí. Díky tomu, že C++ je kompilovaný jazyk, který používá statické typovaní a umožňuje manuálně spravovat alokaci paměti, tak se jedná o vhodnou volbu pro tvorbu implementace, která má být optimalizována pro rychlost. Pro tento projekt je také velmi užitečné, že C++ poskytuje poměrně rozsáhlé metaprogramování pomocí šablon. Vyhodnocení výsledků experimentů je provedeno v prostředí Jupyter Notebook [24] a část pro generování aritmetických obvodů využívá jazyk Python.

6.1 Generování aritmetických obvodů

Generování aritmetických obvodů se nachází v souboru `gen_circ.py`. Jedná se o skript v jazyce Python, který vytvoří veškeré obvody, které jsou v rámci projektu později potřeba. Samotné generování je realizováno pomocí knihovny ArithsGen [23]. Tato knihovna umožňuje jednoduše vytvářet sčítačky, násobičky, děličky, obvody „vynásob a přičti“ a další. Uživatel si můžu zvolit požadovanou bitovou šířku vstupních operandů a zdali jsou se znaménkem nebo bez. Dále knihovna poskytuje několik různých architektur pro tyto obvody. V tomto projektu je pro sčítačky použita architektura `UnsignedRippleCarryAdder` a pro násobičky `UnsignedArrayMultiplier`. Vygenerované obvody je možné exportovat do formátů jako je Verilog, Berkeley Logic Interchange Format (BLIF) a C/C++. Mimo tyto běžné formáty je možné obvod exportovat i jako genotyp pro CGP. Tímto způsobem jsou vytvořeny počáteční obvody pro standardní hradlovou reprezentaci. Pro alternativní reprezentace je potřeba ještě provést převod, a proto jsou obvody exportovány jako BLIF a dále zpracovány následujícím programem.

6.2 Převod do alternativních reprezentací a CGP

Převod do alternativních reprezentací je implementován programem, jehož zdrojový kód se nachází v souboru `main_blif2cgp.cpp`. Jádrem programu je knihovna mockturtle [42]. Ta slouží pro práci s AIG, MIG a XMG a poskytuje řadu algoritmů pro jejich optimalizace a transformace. Vstupní obvod ve formátu BLIF je načten a převeden do všech tří alternativních reprezentací pomocí funkce `mockturtle::convert_cover_to_graph`[1]. Na obvody je dále aplikován optimalizační algoritmus s cílem minimalizovat velikost. Konkrétně je použita funkce `mockturtle::cut_rewriting`[2]. Optimalizované alternativní reprezentace jsou následně exportovány jako CGP genotypy a uloženy v souboru ve formátu Comma-separated values (CSV). Při převodu do CGP bylo potřeba vyřešit, jak reprezentovat invertované hrany a jak adresovat logické konstanty. Bylo zvoleno řešení, kdy invertované hrany jsou realizovány CGP bloky, které provádí negaci. Takže první bloky ve výsledném genotypu jsou negace všech primárních vstupů. Následně jsou vkládány jednotlivé uzly a po každém uzlu je vygenerován blok, který neguje předchozí uzel. Vzniklé adresování, kdy nejméně významný bit značí znaménko, je podobné formátu AIGER (sekce 2.1). Jedná se o redundantní kódování, jelikož některé bloky nemusí být využity. Počet takovýchto neaktivních bloků bývá však velmi malý. Logické konstanty jsou reprezentovány primárními vstupy na adresách 0 a 1. Kromě genotypů obsahuje výstupní soubor i následující informace: identifikátor reprezentace, počet primárních vstupů (včetně logických konstant), počet primárních výstupů a počet CGP bloků.

6.3 Modulární implementace CGP

Pro potřeby této práce byla vytvořena implementace kartézského genetického programování (CGP). Bylo žádoucí, aby se daly měnit jednotlivé části evolučního algoritmu (například mutační operátor, funkční sada atd.) bez nutnosti zasahovat do kódu hlavní evoluční smyčky. Z toho důvodu byly při implementaci hojně využívány C++ šablony a řada funkcí implementuje návrhový vzor „Strategie“ [16]. Šablony jsou instanciovány při překladu, což umožní překladači lépe optimalizovat. CGP je implementováno jako knihovna v hlavičkových souborech s prefixem `cgp_*`. Kód hlavní evoluční smyčky se nachází v souborech `cgp_evolution_{bdd|sim}.hpp` a pro spuštění evoluce slouží funkce `RunCGPw{BDD|Sim}`. Tyto funkce implementují algoritmus popsaný v sekci 4.5 a vyžadují několik konkrétních strategií. Těmi jsou `FSet` (funkční množina), `Mutation` (mutační operátor), `ErrorMetric` (chybová metrika) a `CostF` (fitness funkce). Mezi další nastavitelné parametry patří velikost populace a maximální počet evaluací fitness funkce.

6.4 Datový typ pro funkční množinu

Pro každou funkční množinu (AIG, MIG, XMG a hradla) byla vytvořena popisná třída (viz výpis 6.1). Jejím účelem není vytvářet objekty, ale slouží pro zapouzdření důležitých konstant souvisejících s danou reprezentací a používá se při specializaci generických funkcí. Z tohoto důvodu, jsou veškeré její atributy konstantní, metody statické a konstruktory smazané. Třída obsahuje výčet operátorů, počet operátorů, maximální aritu, identifikátor reprezentace a metodu pro získáni arity daného operátoru.

[1] https://mockturtle.readthedocs.io/en/latest/algorithms/cover_to_graph.html
[2] https://mockturtle.readthedocs.io/en/latest/algorithms/cut_rewriting.html

```
class Xmg {
public:
    enum Operator : uint8_t { kWire = 0, kNot = 1, kXor = 2, kMaj = 3 };
    static const uint8_t kOperators = 4;
    static const uint8_t kMaxArity = 3;
    static inline const std::string kName = "xmg";
    static constexpr uint8_t kArity(Operator op) { ... }
    Xmg() = delete;
};
```

Výpis 6.1: Datový typ použitý pro uložení základních informací o funkční množině.

6.5 Struktura pro genotyp

Genotyp by mohl být uložen jako vektor celých čísel, avšak pro lepší čitelnost algoritmů a menší šanci na zavedení chyby byl zvolen přístup, kdy je genotyp uložen strukturovaně (viz výpis 6.2).

Pro reprezentaci uzlů byla vytvořena struktura `Block`. Jedná se o generickou strukturu, která se specializuje pomocí parametru `FSet`. Počet vstupů je roven maximální aritě a jsou uloženy v poli typu `std::array`, které vyžaduje konstantní velikost, díky čemuž umožňuje uložit elementy lokálně. Operátor může nabývat pouze hodnot daných funkční množinou.

Samotný genotyp je poté reprezentován strukturou `Genotype`, která obsahuje dynamicky alokovaný vektor bloků a vektor pro geny primárních výstupů. Je vhodné poznamenat, že velikost bloku je známá v době překladu. Vektor bloků tedy bude v paměti uložen souvisle, což je důležité pro dodržení principu lokality vyrovnávacích pamětí.

```
template<class FSet>
struct Block {
    std::array<uint16_t, FSet::kMaxArity> inputs_;
    FSet::Operator op_;
};

template<class FSet>
struct Genotype {
    std::vector<Block<FSet>> blocks_;
    std::vector<uint16_t> pos_;
};
```

Výpis 6.2: Datové typy pro reprezentaci genotypu.

6.6 Použité mutační operátory

Implementace mutačních operátorů se nachází v souboru `cgp_mutation.hpp`. Každý operátor je reprezentován třídou, která musí obsahovat statickou metodu `CreateOffspring`. Jejím výstupem je nový potomek a příznak, který oznamuje, zdali došlo k mutaci aktivního genu. Parametry těchto operátorů se nastavují pomocí C++ šablon. Byly implementovány operátory:

- `MutationPoint<K>` – mutace `K` náhodně zvolených genů. Využívá techniku skip [17] popsanou v sekci 4.6.
- `MutationPointNoSkip<K>` – mutace `K` náhodně zvolených genů. Tento operátor vždy oznamuje, že došlo k mutaci aktivního genu (simulace vynechání techniky skip).
- `MutationSingle` – mutace single [17]. Opakovaně aplikuje `MutationPoint<1>` a garantuje mutaci právě jednoho aktivního genu.
- `MutationSagTree<D,P>` – kombinovaný mutační operátor SagTree [10] popsaný v sekci 5.4. Parametr `D` označuje maximální hloubko deaktivovaného podgrafu a `P` značí pravděpodobnost volby mutační části.
- `MutationSsingleTree<D,P>` – alternativa k SagTree, která se liší tím, že v mutační části používá operátor single.

6.7 Výpočet aproximační chyby

Před samotným výpočtem aproximační chyby je potřeba analyzovat chování daného obvodu. Za tímto účelem se provede buď simulace pro všechny vstupní kombinace nebo se vytvoří BDD reprezentace.

Simulace obvodu využívá základní algoritmy popsané v sekci 4.3. Pro větší efektivitu byla implementována paralelní simulace na úrovni bitů představená v sekci 5.2. Jádrem bitového paralelismu je efektivní implementace bitového vektoru. Cílem bylo vytvořit takový kód, který využije instrukční sadu AVX2 poskytovanou použitým procesorem. Bylo vyzkoušeno několik variant. Jejich porovnání a výsledky budou detailně rozebrány v experimentu v sekci 7.1. Výstupem simulace je bitový vektor pro každý primární výstup. Funkce pro výpočet aproximační chyby však očekává vektor celých čísel (pro každou vstupní kombinaci jedno číslo). Implementace požadované transformace se ukáže jako časově velmi náročná a v sekci 7.2 budou popsány programátorské techniky použité pro její optimalizaci.

Pro práci s BBD byla využita knihovna BuDDy[3]. Tvorba BDD reprezentace je implementována v souboru `cgp_evolution_BDD.hpp`. Funkce `ConvertToBdd` postupně iteruje přes všechny aktivní uzly a vytváří jejich BDD reprezentace spojováním BDD vstupních uzlů pomocí požadovaného operátoru. Výsledkem je BDD pro každý primární výstup. Výpočet aproximačních chyb pomocí BDD je poté realizován pomocí veřejně dostupné implementace [34].

Samotné aproximační chybové metriky jsou implementovány v hlavičkovém souboru s názvem `cgp_approximation_error.hpp`. Opět se jedná o třídy, které mají smazaný konstruktor a obsahují pouze metodu `Calculate`. Byly implementovány chybové metriky maximální a průměrná absolutní vzdálenost (e_{wce} a e_{mae}) ve variantách pro simulaci a BDD. Jsou reprezentované třídami `Error{WorstCase|Mean}Absolute{Sim|BDD}`.

6.8 Fitness funkce

Pokud aproximační chyba kandidátního řešení nepřesahuje stanovenou hranici, tak je mu přiřazena fitness hodnota podle zvolené cenové funkce. V opačném případě je jeho fitness nastavena na nejhorší hodnotu dané cenové funkce. Implementace podle tohoto popisu by

[3] https://github.com/SSoelvsten/buddy

však nebyla příliš efektivní. Jelikož je výpočet aproximační chyby typicky mnohem náročnější než výpočet cenové funkce, tak je lepší nejprve porovnat cenu kandidátního řešení s cenou nejlepšího doposud nalezeného řešení a aproximační chybu kontrolovat pouze v případě, kdy kandidátní řešení nemá horší cenu a tedy je možné, že se stane rodičem příští generace.

V rámci této práce bylo implementováno několik různých fitness funkcí, které se nachází v souboru `cgp_cost.hpp`. Každá fitness funkce je implementována jako třída, která musí mít statickou metodu `Cost`. Parametrem šablony této metody je třída s funkční množinou. Fitness hodnoty 16bitových sčítaček a 8bitových násboček jsou uvedeny v tabulkách 6.2 a 6.3. Ve zdrojovém souboru se nachází třídy:

- `CostActiveBlockCountArea` – počet aktivních bloků.
- `CostUniformExcInv{Area|Delay}` – přiděluje každému uzlu v grafu uniformní velikost/zpoždění. Zpoždění se vypočte jako součet zpoždění uzlů na nejdelší cestě od primárních vstupů k výstupům. Jedná se o heuristiku, kterou používají např. autoři MIG nebo XMG ve svých optimalizačních algoritmech [1][18].
- `CostUniformIncInv{Area|Delay}` – přiděluje i invertorům jednotkovou velikost/zpoždění. Tato fitness funkce je motivována vznikajícími technologiemi, které poskytují efektivní implementaci majority [3].
- `Cost45nmMajToAndOr{Area|Delay}` – jednoduché mapování do 45nm technologie. Každému operátoru je přidělena velikost podle 6.1 (převzato z [10]). Majorita má stejnou velikost jako 3 hradla AND a 2 hradla OR. Ale pokud je jedním ze vstupů konstantní signál, tak majoritě je přidělena velikost AND nebo OR. Zpoždění všech operátorů kromě majority je jednotkové. Majorita má zpoždění 2, protože první úroveň obsahuje 3 hradla AND a druhá úroveň může teoreticky být tvořena jedním hradlem OR se třemi vstupy.
- `CostAreaDelayProduct<AreaF,DelayF>` – kombinuje dvě fitness funkce a počítá jejich součin. V experimentech bude používáno označení ADP.
- `CostAreaDelayProductLut<K>` – využívá knihovnu mockturtle [42] pro mapování na K-LUT pomocí funkce `mockturtle::lut_map`[4]. Tato knihovní funkce provádí mapování, které se snaží minimalizovat buď velikost nebo zpoždění. Jelikož jsou alternativní reprezentace počátečních obvodů optimalizovány s ohledem na velikost, tak i tato fitness funkce provádí mapování se zaměřením na velikost. Výsledná fitness hodnota je poté součin velikosti a zpoždění.

Operátor	INV	AND	OR	XOR	NAND	NOR	XNOR	MAJ
Relativní cena	1,40	2,34	2,34	4,69	1,87	2,34	4,69	11,7

Tabulka 6.1: Relativní cena operátorů při použití fitness funkce `Cost45nmMajToAndOr`.

[4] https://mockturtle.readthedocs.io/en/latest/algorithms/lut_mapping.html

Repr.	IncInv	ExInv	45nm	4-LUT	5-LUT	6-LUT
aig	310×65	139×32	565×65	31×15	24×8	24×8
mig	171×18	109×17	482×32	31×15	24×8	24×8
xmg	111×16	79×16	405×31	31×15	24×8	24×8
hradla	77×31	77×31	253×31	31×15	24×8	24×8

Tabulka 6.2: Velikost × zpoždění 16bitových sčítaček (arch. `UsignedRippleCarryAdder`).

Repr.	IncInv	ExInv	45nm	4-LUT	5-LUT	6-LUT
aig	1055×106	528×53	1973×106	125×20	106×19	96×17
mig	640×50	432×38	1751×62	129×20	106×19	96×17
xmg	224×26	224×26	1217×39	128×20	110×19	100×17
hradla	320×40	320×40	993×40	125×20	108×19	93×17

Tabulka 6.3: Velikost × zpoždění 8bitových násobiček (arch. `UsignedArrayMultiplier`).

Kapitola 7

Experimentální vyhodnocení

V této kapitole budou experimentálně ověřeny algoritmy pro evoluční návrh aproximačních obvodů v alternativních reprezentacích. Veškeré experimenty budou spouštěny na počítači s procesorem AMD Ryzen 2600 v prostředí Windows Subsystem for Linux 2 s Ubuntu 22.04. Pro překlad byl použit překladač g++ verze 11.4.0. Při překladu byly použity přepínače `-std=c++20 -O3 -march=znver1`.

Nejprve v sekci 7.1 bude hledán vhodný datový typ pro reprezentaci bitových vektorů potřebných pro paralelní simulátor obvodů. Poté v sekci 7.2 bude změřena časová náročnost jednotlivých částí evolučního algoritmu a kritické části budou optimalizovány. Následně v sekci 7.3 budou porovnány populární mutační operátory. Porovnání paralelního simulátoru s přístupem založeným na BDD bude provedeno v sekci 7.4. Samotné porovnání výsledků evoluce v alternativních reprezentacích při použití jednoduchých fitness funkcí bude v sekci 7.5. Návazná sekce 7.6 se bude zabývat mapováním získaných výsledků na k-LUT. Fitness funkce zaměřená na k-LUT mapování bude vyzkoušena v sekci 7.7.

7.1 Efektivní reprezentace bitového vektoru

V sekci 5.2 bylo řečeno, že je vhodné věnovat snahu na efektivní implementaci simulátoru obvodů. V případě realizace simulátoru založeného na bitově paralelní simulaci jsou jeho jádrem operace nad bitovými vektory. Tato sekce je věnována experimentu, jehož cílem je nalézt vhodný datový typ, který poskytne efektivní implementaci logických operátorů nad bitovými vektory.

Předmětem experimentu je simulace vyhodnocení CGP bloku realizujícího třívstupou majoritu. Průběh experimentu je následovný. Program nejprve načte argumenty příkazové řádky, které určují velikost bitového vektoru a počáteční hodnotu generátoru náhodných čísel. Poté jsou alokovány bitové vektory A, B, C a R. Vektory A, B a C jsou inicializovány náhodně pomocí generátoru náhodných čísel. Vektor R slouží pro uložení výsledku, který se vypočítá jako $R_i = (A_i \wedge B_i) \vee (A_i \wedge C_i) \vee (B_i \wedge C_i)$. Doba výpočtu vektoru R je změřena pomocí funkcí z knihovny `std::chrono`. Parametry experimentu, naměřený čas a počet bitů nastavených na log. 1 v R jsou vypsány na standardní výstup.

Byly porovnány různé varianty implementace bitového vektoru. Výsledky porovnání jsou uvedeny v tabulce 7.1. Ta obsahuje několik sloupců, pro různé velikosti vstupních vektorů. Pro každou variantu bylo provedeno 30 běhů, z nichž byl vypočítán aritmetický průměr.

První zkoumanou variantou je `std::vector<bool>`. Dokumentace specifikuje, že tento datový typ může být implementovaný tak, že pro jeden prvek se využije pouze jeden bit [45].

Bohužel však neposkytuje implementaci pro logické operátory, které by pracovaly s dvojicí těchto vektorů. Takže v rámci experimentu bylo potřeba iterovat po jednotlivých bitech a aplikovat požadované logické operátory. Z výsledků je patrné, že tohle není efektivní implementace a je zde uvedena spíše jako odstrašující případ.

O něco lepší je implementace `boost::dynamic_bitset`[1] z knihovny boost. Ta už poskytuje implementaci logických operátorů. Má však také několik nedostatků. Při zkoumání výstupu překladače bylo zjištěno, že pro vygenerování kódu logických operátorů nevyužil instrukční sadu AVX2, která je k dispozici. Dalším problémem jsou mezivýsledky. Jelikož operandy jsou vektory, tak i mezivýsledky se ukládají jako vektory, což znamená, že se pro ně musí alokovat značné množství paměti. Jedná se o zbytečné ukládání a pozdější načítání z hlavní paměti. Při porovnání s některým z lepších řešení je vidět, že relativní doba výpočtu se skokově zvětšuje. To se dá vysvětlit tím, že menší vektory se vejdou do nižších úrovní vyrovnávací paměti, avšak větší vektory už se tam nevejdou.

Nejlepších výsledků dosahují implementace pomocí `std::vector<uint64_t>` nebo prostých polí. Při výpočtu majority se iteruje přes všechny vstupní vektory zároveň, takže mezivýsledky se mohou ukládat do registrů a do hlavní paměti se přistupuje až při zápisu výsledné hodnoty. Mezi implementací pomocí prostých polí a `std::vector` není rozdíl. Překladači se podařilo obě verze automaticky vektorizovat a vygeneroval pro ně stejný kód. Ten je uveden na výpisu 7.1.

Dále bylo vyzkoušeno, jestli pomůže alokovat paměť se zarovnáním na 32 bajtů (šířka registru AVX2). Vygenerovaný kód uvedený na výpisu 7.2 se od původní varianty liší pouze v instrukci pro načítání dat, která využívá zarovnanou variantu. Z naměřených výsledků je vidět, že tato změna nepřinesla zlepšení.

Varianta	Doba výpočtu [μs]						
	2^8 b	2^{10} b	2^{12} b	2^{14} b	2^{16} b	2^{18} b	2^{20} b
`std::vector<bool>`	3.97	6.41	24.61	98.87	376.08	1488.03	6253.10
`boost::dynamic_bitset`	0.33	0.36	0.53	1.29	3.22	23.21	128.61
`uint8_t[]`	0.04	0.06	0.08	0.14	0.44	1.62	6.48
`uint16_t[]`	0.05	0.07	0.10	0.17	0.44	1.62	6.59
`uint32_t[]`	0.04	0.06	0.08	0.16	0.44	2.01	6.63
`uint64_t[]`	0.05	0.06	0.08	0.16	0.45	1.61	6.64
`uint8_t[] (simd aligned)`	0.04	0.08	0.09	0.13	0.44	1.62	6.76
`uint16_t[] (simd aligned)`	0.04	0.06	0.10	0.16	0.45	1.61	7.11
`uint32_t[] (simd aligned)`	0.04	0.06	0.08	0.16	0.44	1.60	6.80
`uint64_t[] (simd aligned)`	0.05	0.05	0.08	0.16	0.45	1.59	6.81
`std::vector<uint8_t>`	0.07	0.09	0.11	0.17	0.45	1.62	6.40
`std::vector<uint16_t>`	0.04	0.05	0.08	0.15	0.43	1.61	6.42
`std::vector<uint32_t>`	0.04	0.06	0.09	0.16	0.43	1.63	6.37
`std::vector<uint64_t>`	0.04	0.06	0.07	0.14	0.43	1.60	6.36

Tabulka 7.1: Průměrná doba výpočtu majority pro různé implementace bitového vektoru.

```
.L251:
    vmovdqu xmm2, XMMWORD PTR [r14+rdx]
    vpor xmm0, xmm2, XMMWORD PTR [rbp+0+rdx]
```

[1] https://www.boost.org/doc/libs/release/libs/dynamic_bitset/dynamic_bitset.html

```
    vpand xmm1, xmm2, XMMWORD PTR [rbp+0+rdx]
    vpand xmm0, xmm0, XMMWORD PTR [r12+rdx]
    vpor xmm0, xmm0, xmm1
    vmovdqu XMMWORD PTR [r13+0+rdx], xmm0
    add rdx, 16
    cmp rcx, rdx
    jne .L251
```

Výpis 7.1: Vygenerovaný kód pro varinaty s prostým polem a `std::vector`.

```
.L202:
    vmovdqa xmm2, XMMWORD PTR [r15+rdx]
    vpor xmm0, xmm2, XMMWORD PTR [rbx+rdx]
    vpand xmm1, xmm2, XMMWORD PTR [rbx+rdx]
    vpand xmm0, xmm0, XMMWORD PTR [r14+rdx]
    vpor xmm0, xmm0, xmm1
    vmovdqa XMMWORD PTR [rbp+0+rdx], xmm0
    add rdx, 16
    cmp rdx, rsi
    jne .L202
```

Výpis 7.2: Vygenerovaný kód pro varianty se zarovnáním na 32 bajtů.

7.2 Optimalizace převodu primárních výstupů na čísla

Po implementování základní varianty CGP využívající simulátor založený na bitově paralelní simulaci bylo nejprve potřeba zjistit, která část algoritmu spotřebuje nejvíce procesorového času. Poté byla provedena řada experimentů s cílem zlepšit efektivitu implementovaného algoritmu. Tato sekce se věnuje profilování základní varianty CGP a hledání optimalizací funkce pro převod bitových vektorů s primárními výstupy na vektor čísel.

Experiment probíhal následovně. Bylo spuštěno CGP, jehož cílem bylo minimalizovat počet aktivních bloků v zadaném obvodu. Přičemž kandidátní obvod byl uvažován pouze v případě, pokud jeho maximální absolutní vzdálenost (e_{wce}) od referenčního obvodu nepřesáhla 1 %. Evoluce byla ukončena po provedení 100 generací. V každé generaci byla vygenerována 4 kandidátní řešení pomocí mutačního operátoru single, který garantuje odlišnost od rodiče. Zadanými obvody byly sčítačky a násobičky v AIG, MIG a XMG reprezentacích. Bitová šířka jednotlivých vstupních operandů byla postupně zvyšována ze 4 na 10 bitů. Bylo vyzkoušeno celkově 9 různých implementací převodu bitových vektorů s primárními výstupy na vektor čísel, které budou postupně představeny a porovnány. Pro každou variantu bylo provedeno 10 různých běhů. Stejný průběh evoluce napříč variantami byl zajištěn totožnou inicializací generátoru náhodných čísel.

Před samotnými optimalizacemi byl však nejprve vytvořen profil programu. Na základě něho byla zvolena část programu, kterou by bylo vhodné optimalizovat. K tomu byl využit nástroj `perf`[2]. Výsledky měření jsou uvedeny v tabulce 7.2. Byly uvedeny pouze ty časově nejnáročnější části. Z výsledků je patrné, že u testovaných obvodů je nejnáročnější funkce, která transformuje bitové vektory primárních výstupů na vektor čísel, který je potřeba pro výpočet chybové metriky. Z toho důvodu je tato sekce věnována optimalizaci této

[2] https://perf.wiki.kernel.org

funkce. Dále je možné pozorovat, že relativní doba výpočtu této funkce roste se zvětšující se bitovou šířkou primárních výstupů. V tabulce je uvedena i velikost počátečních obvodů v AIG, MIG a XMG reprezentaci. Zdá se, že velikost sčítaček roste mnohem pomaleji než velikost násobiček při zvětšující se bitové šířce vstupních operandů, což by vysvětlovalo prudký nárůst času spotřebovaného na simulaci bloků CGP. Optimalizací simulace bloků se již částečně zabývala sekce 7.1. V tabulce nejsou uvedeny funkce provádějící mutaci a výpočet velikosti obvodu (fitness funkce), jelikož doba jejich výpočtu je zanedbatelná.

Obvod	Velikost	Převod p. v.	Nalezení akt. b.	Simulace b.	e_{wce}
Sčítačka 4b	70, 39, 27	44,9 %	9,5 %	8,8 %	4,1 %
Sčítačka 10b	190, 105, 69	76,6 %	0,1 %	9,4 %	7,97 %
Násobička 4b	207, 128, 48	46,9 %	19,8 %	9,1 %	3,2 %
Násobička 10b	1719, 1040, 360	52,2 %	0,1 %	42,2 %	2,9 %

Tabulka 7.2: Výsledky profilování implementovaného CGP. Procentuální hodnoty vyjadřují, kolik času bylo relativně spotřebováno na jednotlivých částech programu při evoluci zadaného obvodu. Sloupec *velikost* udává počet aktivních bloků referenčního obvodu v reprezentaci AIG, MIG a XMG. Celé názvy ostatních sloupců jsou *převod primárních výstupů*, *nalezení aktivních bloků* a *simulace bloků* .

Počáteční varianta funkce pro převod primárních výstupů na čísla je `PIs2UnsignedV0`. Tato varianta byla použita při profilování implementace CGP. Její kód je uveden na výpisu 7.3. Vstupem funkce jsou: `pos` – indexy bloků reprezentujících primární výstupy obvodu, `memory` – výstupy všech bloku a `circuit` – parametry zadaného obvodu. Cykly na řádcích 8 a 9 postupně iterují přes výstupy pro všechny vstupní kombinace. Řádek 10 obsahuje cyklus, který prochází jednotlivé primární výstupy. Na řádku 12 se provede načtení části bitového vektoru, ze kterého se vymaskuje bit odpovídající požadovanému primárnímu výstupu, který se posune na správnou pozici a poté se tento bit nastaví v čísle reprezentující výsledek pro danou vstupní kombinaci. Výstupem funkce je vektor výsledků pro každou vstupní kombinaci.

```
1  std::vector<uint64_t> Convert(const std::vector<uint16_t> &pos,
2                                const std::vector<std::vector<uint64_t>> &memory,
3                                const Circuit &circuit)
4  {
5    auto result = std::vector<uint64_t>(1ull << (circuit.pis_ - 2));
6    const size_t chunks = memory[0].size();
7
8    for (size_t chunk = 0; chunk < chunks; ++chunk)
9      for (size_t bit = 0; bit < 64; ++bit)
10       for (size_t po = 0; po < pos.size(); ++po)
11         result[chunk * 64 + bit]
12             |= ((memory[pos[po]][chunk] & (1ull << bit)) >> bit) << po;
13
14   return result;
15 }
```

Výpis 7.3: Zdrojový kód funkce pro převod primárních výstupů na čísla. Jedná se o neoptimalizovanou variantu `PIs2UnsignedV0`.

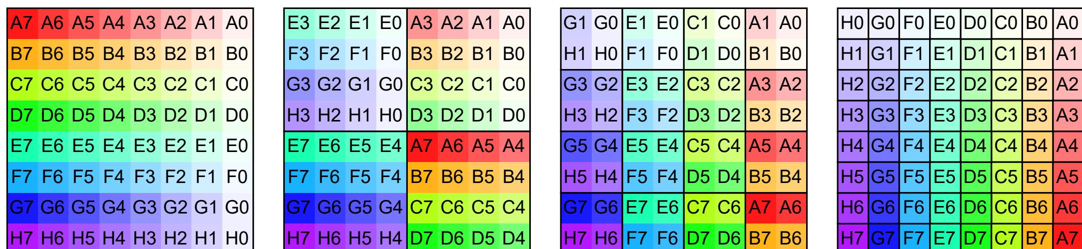
První modifikací, která přinesla zlepšení, byla změna pořadí smyček a je implementována ve variantě `PIs2UnsignedV1`. Jeden z problémů původní implementace je ten, že v každé iteraci nejvnitřnější smyčky se načítá z jiné adresy. Takže byly prohozeny smyčky na řádcích 8 a 10. Tím byl získán lepší přístupový vzor do paměti. Vnější smyčka iteruje přes bitové vektory, jejichž lokace může být prakticky náhodná. Následující smyčka iteruje přes části bitových vektorů, které jsou uložené na sousedících adresách. Nakonec nejvnitřnější smyčka iteruje přes bity jednoho konkrétního 64bitového čísla.

Další změnou, které vedla ke zrychlení, bylo rozgenerování nejvnitřnější smyčky (ve výpisu 7.3 se jednalo o smyčku přes bity na řádku 9). Při analýze kódu vygenerovaného překladačem pro variantu `PIs2UnsignedV1` bylo zjištěno, že značné množství času se spotřebuje na režii nejvnitřnější smyčky. Z toho důvodu bylo vytvořeno několik implementací, označených jako `PIs2UnsignedV{2-7}`, které se liší počtem rozgenerovaných iterací. Varianta `PIs2UnsignedV2` vznikla zkopírováním řádků 11 a 12 a přičtením 1 k hodnotě proměnné `bit` v kopii a zdvojnásobením délky kroku proměnné `bit`. Každá další varianta vznikla obdobným způsobem s tím, že délka kroku se vždy zdvojnásobila. Takže ve variantě `PIs2UnsignedV7` je smyčka rozgenerována kompletně.

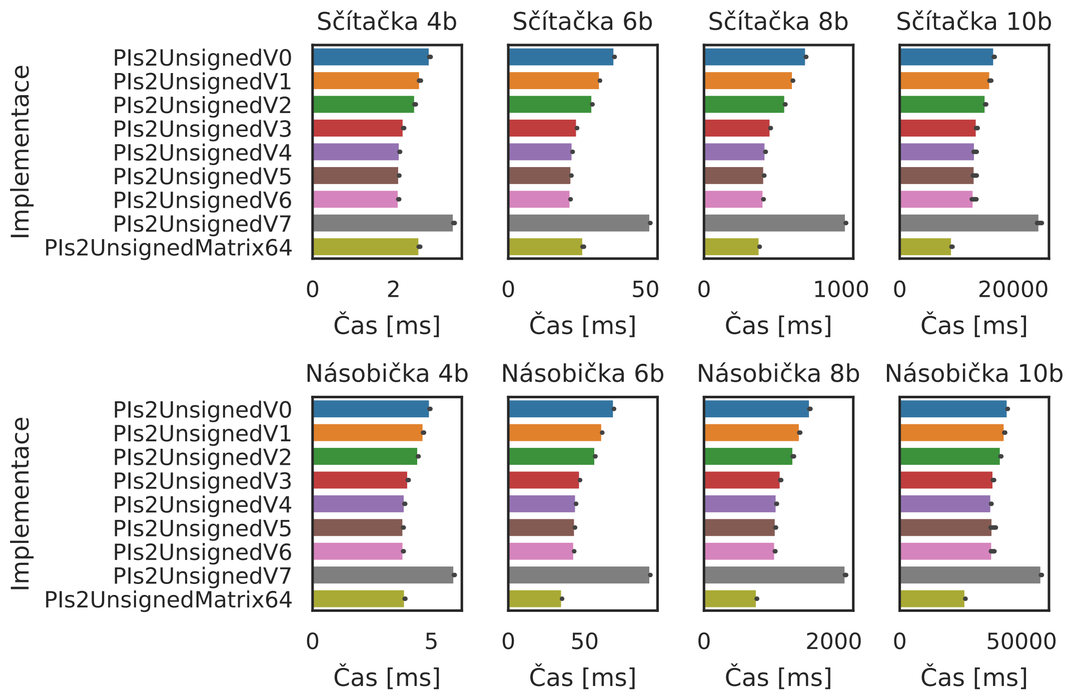
Dalším problémem je asymptotická časová složitost použitého algoritmu. Úloha, kterou se zabývá tahle sekce, by se dala chápat jako transpozice bitové matice. Algoritmus z výpisu 7.3 provádí transpozici vstupní matice bit po bitu. Takže pokud je vstupem čtvercová matice velikosti $n \times n$, tak její transpozice pomocí zmíněného algoritmu má časovou složitost $O(n^2)$. Existuje však algoritmus s lepší časovou složitostí.

Efektivnější algoritmus pro transpozici bitové matice spočívá v paralelismu na úrovni bitů [46]. Místo toho, aby maskoval a posouval bity jeden po druhém, tak využívá skutečnosti, že na 64bitových procesorech lze pracovat s 64bity najednou. Činnost algoritmu je ukázána na obrázku 7.1. V prvním kroku je vstupní matice rozdělena na 4 menší čtvercové bloky. Následně je provedena transpozice těchto bloků. Poté se tento postup aplikuje rekurzivně na každý blok. Rekurze je ukončena v momentě, kdy velikost bloku je 1×1. Pro vybrání správných bitů v první iteraci se použije maska `0b11110000`, ve druhé iteraci `0b11001100` a ve třetí `0b10101010`. Obdobným způsobem se dá algoritmus rozšířit na matice 16×16, 32×32 atd. Mimochodem by se dalo ukázat, že ve skutečnosti nezáleží na pořadí masek. Tento algoritmus má časovou složitost $O(n \cdot \log n)$ za předpokladu, že je prováděn procesorem, který má n-bitové registry a poskytuje logické operátory nad n-bitovými operandy. S běžně dostupnými procesory jsme omezeni na matice velikosti 64×64. Existují však i podobné algoritmy, které využívají vektorovou instrukční sadu a dovolují efektivně transponovat větší matice[37].

Varianta `PIs2UnsignedMatrix64` využívá algoritmus založený na transpozici čtvercových matic dostupný z [46]. Vstupem funkce je matice o velikosti $m \times n$, kde m je počet



Obrázek 7.1: Ukázka činnosti algoritmu pro transpozici bitové matice.



Obrázek 7.2: Výsledky porovnání jednotlivých variant převodu bitových vektorů s primárními výstupy na vektor čísel při evoluční minimalizaci sčítaček a násobiček. Počet bitů v názvech obvodů udává bitovou šířku jednotlivých operandů. Předmětem měření je celková doba běhu evolučního algoritmu ve všech třech uvažovaných reprezentacích.

primárních výstupů a n je počet vstupních kombinací. Tato matice je pokryta čtvercovými maticemi o velikosti 64×64. Každá z těchto matic je transponována a tím je získán výsledek. Dalo by se očekávat, že tato varianta začne být efektivní až pro obvody s větším počtem primárních výstupů. Kvůli tomu, že při malém počtu primárních výstupů bude většina čtvercové matice obsahovat nuly, které budou zbytečně prohazovány. Původní algoritmus pracoval pouze s platnými bity.

Porovnání všech zmíněných variant převodu bitových vektorů s primárními výstupy na vektor čísel je uvedeno na obrázku 7.2. Z naměřených výsledků je možné vyčíst, že změna pořadí smyček (`V1`) měla pozitivní efekt na dobu běhu CGP pro všechny testované obvody. Rozgenerování nejvnitřnější smyčky (`V2` až `V7`) se také projevilo pozitivně. Je vidět, že zvyšování počtu iterací, které jsou rozgenerovány, přinášelo další zlepšení (`V2` – 2 iterace, `V3` – 4 it., `V4` – 8 it.). Další zvýšení počtu rozbalených iterací již nepřineslo zlepšení (`V5` až `V7`). Při úplném rozbalení dokonce došlo k výraznému zhoršení (`V7`). Při zkoumání maticového algoritmu (`Matrix64`) se potvrdil předpoklad, že jeho efektivita se projeví až při obvodech s větším počtem primárních výstupů. První zlepšení přináší u obvodu „Sčítačka 8b“, což je obvod s 9 výstupy. U násobiček se ukázalo to stejné. Vzhledem k výsledkům u 4bit. a 6bit. násobiček, by se dalo očekávat, že maticový algoritmus začne být efektivní u 5bit. násobiček, což by byl obvod s 10 výstupy. Takže na základě těchto měření se jako nejvhodnější varianta pro menší obvody (8 a méně výstupů) jeví libovolná z `V4`, `V5` a `V6`. Pro obvody, které mají 9 a více vstupů se vyplatí použít maticový algoritmus.

Tato sekce se doposud zabývala pouze obvody, které pracují s nezápornými čísly. Výše představené algoritmy je potřeba doplnit, aby fungovaly správně i pro obvody, které podporují záporná čísla. Nejvýznamnější výstupní bit u těchto obvodů představuje znaménko. Předchozí algoritmus se dá použít pro transpozici, avšak poté je potřeba zkontrolovat, zdali číslo je záporné a pokud ano, tak se musí provést znaménkové rozšíření.

Výpis B.1 obsahuje kód pro znaménkové rozšíření čísla `num`, které je zakódované v dvojkovém doplňku na `circuit.pos_` bitech. Jsou zde uvedeny dvě varianty. První z nich je sice intuitivní, ale má jednu podstatnou nevýhodu. Vzhledem k tomu, že obvody jsou upravovány pomocí náhodných mutací, tak nelze předpokládat, že posloupnost znamének výstupních čísel bude nějakým způsobem pravidelná. Což je velmi nepříjemné pro prediktor skoku a spekulativní načítání instrukcí. Takže tato podmínka způsobí, že značná část spekulativního výpočtu bude muset být zahozena a provedena znovu ve správné větvi. Varianta 2 využívá techniku pro odstranění podmíněného skoku. Ta spočívá ve vynásobení masky pro znaménkové rozšíření výsledkem testu na záporné znaménko. Výsledkem testu je buď 1 nebo 0. Takže buď je aplikována původní maska nebo maska plná nul, což nemá žádný efekt.

```
1 uint64_t sign_mask = 1ull << (circuit.pos_ - 1);
2 uint64_t ext_mask = std::numeric_limits<uint64_t>::max() << circuit.pos_;
3
4 // varianta 1
5 if (num & sign_mask) { num |= ext_mask; }
6
7 // varianta 2
8 num |= ext_mask * ((num & sign_mask) == sign_mask);
```

Výpis 7.4: Úryvek kódu realizující znaménkové rozšíření.

Výsledky porovnání obou variant se nachází v tabulce 7.3. Jedná se o celkovou dobu běhu CGP. Z naměřených hodnot je patrné, že druhá varianta je ve všech případech rychlejší. U sčítaček a obecně menších obvodů jsou rozdíly větší, jelikož v této části kódu je spotřebováno více času. U větších obvodů se více času spotřebuje na simulaci obvodu a tedy rozdíl mezi porovnávanými variantami je potom méně podstatný.

Obvod	Varianta 1 [ms]	Varianta 2 [ms]	Rozdíl
Sčítačka 4b	3.268	3.024	7.466 %
Sčítačka 10b	10800	10225	5.323 %
Násobička 4b	4.595	4.298	6.466 %
Násobička 10b	29236	28353	3.020 %

Tabulka 7.3: Výsledky porovnání variant pro znaménkové rozšíření z výpisu B.1. Počet bitů v názvech obvodů udává bitovou šířku jednotlivých operandů. Předmětem měření je celková doba běhu evolučního algoritmu ve všech třech uvažovaných reprezentacích.

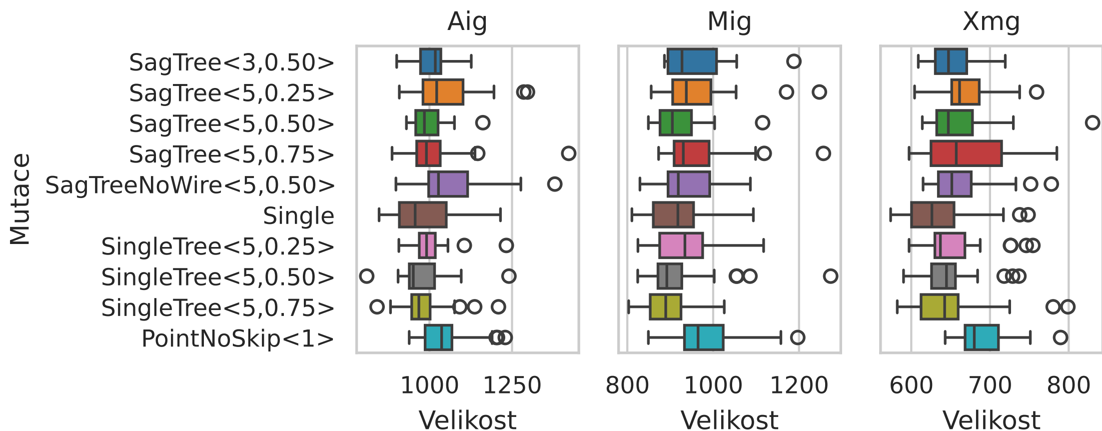
7.3 Porovnání mutačních operátorů

Vytváření nový kandidátních řešení se v CGP provádí pomocí mutačních operátorů. Volba mutačního operátoru má přímý vliv na efektivitu evoluce. Z toho důvodu je tato sekce vě-

nována experimentálnímu vyhodnocení vhodnosti různých mutačních operátorů pro návrh aproximačních obvodů v alternativních reprezentacích.

V tomto experimentu bylo cílem CGP aproximovat násobičky, které pracují s 8bitovými operandy bez znaménka. Jako fitness funkce byl použit odhad velikosti obvodu (`Cost45nmMajToAndOrArea`). Kandidátní řešení bylo uvažováno pouze v případě, pokud jeho e_{wce} nepřesáhlo 1 % (což v případě obvodů s 16bitovými výstupy znamená, že rozdíl mezi aproximovaným a referenčním výsledkem je maximálně 655). Byla použita evoluční strategie $1 + 4$. Evoluce byla ukončena po 500000 evaluacích fitness funkce. Použité mutační operátory a jejich výsledky budou rozebrány v následujících odstavcích. Pro každý z nich bylo provedeno 24 nezávislých běhů.

Vývoj fitness hodnoty v průběhu evoluce je uveden na obrázku 7.4 a porovnání nejlepších nalezených řešení je na obrázku 7.3.



Obrázek 7.3: Fitness hodnoty nejlepších nalezených řešení při evoluci 8bitových aproximačních násobiček s max. $e_{wce} = 1$ %. První parametr u `*Tree` operátorů udává maximální hloubku deaktivovaného podstromu a druhý parametr určuje pravděpodobnost, že se provede mutace místo stromové deaktivace. Parametr u `PointNoSkip` vyjadřuje počet zmutovaných genů.

Základní zkoumanou variantou mutačního operátoru je bodová mutace s označením `PointNoSkip<1>`. Parametr v ostrých závorkách určuje, kolik genů je podrobeno mutaci. Tento operátor byl implementován jednoduchým způsobem a neprovádí detekci, zdali došlo k mutaci aktivního genu. Jelikož značná část genů je neaktivních, tak je velká šance, že bude zmutován některý z nich. To má za následek, že fitness funkce bude vyhodnocována zbytečně. Tento nedostatek je pravděpodobně důvodem, proč `PointNoSkip<1>` výrazně zaostává za všemi ostatními operátory z hlediska vývoje fitness hodnoty.

Dalším mutačním operátorem je `Single`, který opakovaně mutuje náhodné geny a skončí ve chvíli, kdy zmutuje jeden aktivní gen [17]. Tento operátor je jedním z možných řešení, jak se vyhnout ohodnocování kandidátních řešení totožných s jejich rodičem. Z hlediska vývoje fitness funkce po většinu času zaostává za operátory z rodiny `SagTree`, avšak při porovnání nejlepších řešení na konci evoluce se dá říct, že `Single` není horší než operátory `SagTree`. Výhoda tohoto operátoru je ta, že nepotřebuje nastavovat parametry.

Prvním kombinovaným mutačním operátorem je `SagTree<D,P>`, který aplikuje buď mutaci jednoho aktivního genu nebo deaktivuje náhodný podstrom s náhodně zvolenou hloubkou, která je maximálně `D`. Pravděpodobnost mutace je $P_m =$ `P` a pravděpodobnost deak-

tivace je $P_d = 1 - \mathtt{P}$. Autoři toho operátoru ve svých experimentech nastavili maximální hloubku na 5 [10]. Vzhledem tomu, že prováděli experimenty na mnohem větších obvodech než jsou použity zde, byla vyzkoušena i varianta s menší maximální hloubkou. Konkrétně byla uvažována i maximální hloubka 3. Myšlenka byla taková, že příliš velká hloubka bude mít příliš destruktivní účinky. Z výsledků je však vidět, že varianta s hloubkou 5, která má shodnou pravděpodobnost mutace dosahuje lepších výsledků. Dále byly vyzkoušeny 3 varianty s různou pravděpodobností mutace (25 %, 50 % a 75 %). Potvrdily se poznatky autorů tohoto operátoru. Varianta s menší pravděpodobností mutace dosahuje lepších výsledků na nízkém počtu evaluací. Avšak při dostatečně velkém počtu evaluací začnou být efektivnější varianty s větší pravděpodobností mutace. Nejlepší variantou se na zvoleném počtu evaluací jeví varianta s pravděpodobností mutace 50 %.

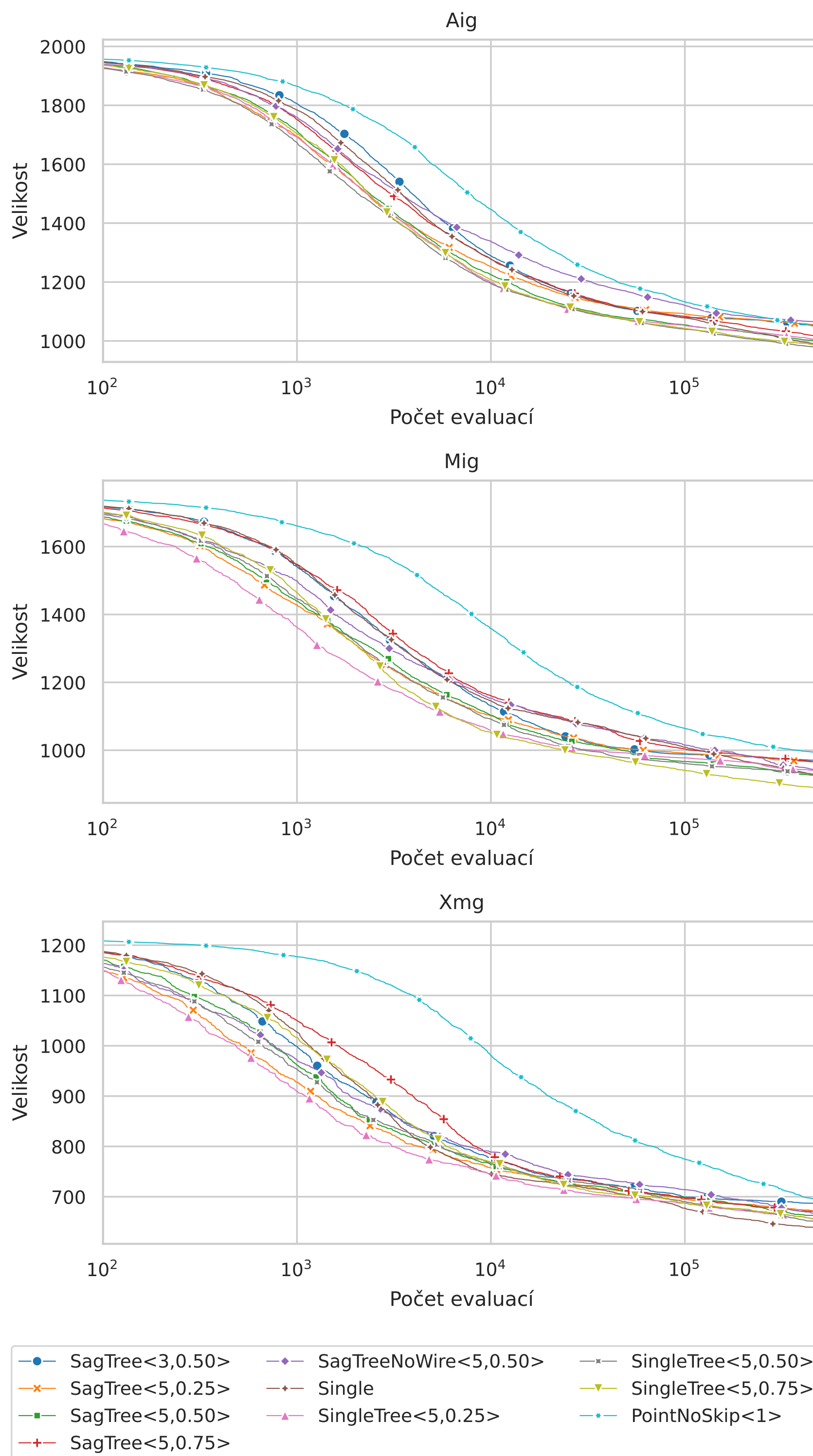
Byly vyzkoušeny i alternativní funkční množiny. Kromě funkcí uzlů příslušné reprezentace a invertoru, obsahují funkční množiny i přímé propojení. Motivace za inkluzí přímého propojení je taková, že umožní prostým mutačním operátorům (tj. `Single` a `PointNoSkip`) snadno deaktivovat jeden blok. Na druhou stranu operátory, které explicitně provádí stromovou deaktivaci, by mohly benefitovat z odstranění prostého propojení, což by vedlo na zmenšení prostoru kandidátních řešení. Tato úprava byla vyzkoušena v kombinaci s operátorem `SagTree<5,0.50>` a je označena sufixem `NoWire`. Z výsledků je možné vyčíst, že takto upravená funkční množina nepřinesla zlepšení pro tento mutační operátor.

Posledním testovaným mutačním operátorem je `SingleTree<D,P>`. Sémantika parametrů `D` a `P` je stejná jako u `SagTree`. Tento kombinovaný operátor byl vytvořen v návaznosti na výsledky předchozích experimentů. V nichž se ukázalo, že `Single` má sice pomalý start, ale i přesto je velmi efektivní. Na druhou stranu `SagTree` má rychlý start, ale v závěru evoluce, kdy je kladen důraz převážně na mutační část tohoto operátoru, tak již není tolik efektivní. Z toho důvodu bylo otestováno, zdali změna mutační části `SagTree` na `Single` přinese zlepšení. Při porovnání `SingleTree` a `SagTree` při stejném nastavení jejich parametrů lze vyvodit následující závěry. Změny parametru `P` nají stejné následky jako u `SagTree` – operátor bude efektivnější buď na začátku evoluce nebo až ke konci. Z hlediska průběhu evoluce lze pozorovat, že u všech variant parametrů a pro všechny tři reprezentace obvodů došlo ke zlepšení, které je patrné při porovnání na libovolném počtu evaluací.

7.4 Výpočet hodnotících kritérií pomocí BDD

Jednou z možností, jak vypočítat hodnotící kritéria pro aproximační obvody, je pomocí BDD. Bylo ukázáno, že pro větší sčítačky je tento přístup mnohem efektivnější než simulace obvodu [35]. Tato sekce se zabývá porovnáním přístupů založených na simulaci a na BDD při použití alternativních reprezentací obvodů.

Cílem CGP v tomto experimentu bylo aproximovat sčítačky, které pracují se 4bitovými až 24bitovými operandy bez znaménka. Byly uvažovány pouze ty obvody, které mají e_{wce} menší než 1 %. Jako fitness funkce byla použita `Cost45nmMajToAndOrArea`. Fitness hodnoty kandidátních řešení byly vypočítány pomocí simulátoru obvodů včetně optimalizací ze sekce 7.2 nebo pomocí BDD. Implementace BDD varianty byla získána z [34]. Byla použita evoluční strategie $1 + 4$. Evoluce byla ukončena po 100000 evaluacích fitness funkce. Vzhledem k poznatkům ze sekce 7.3 byl jako mutační operátor zvolen `SingleTree<5,0.5>`. Pro každou variantu nastavení bylo provedeno 24 nezávislých běhů, které byly spouštěny sekvenčně. Generátor náhodných čísel byl inicializován takovým způsobem, aby průběh evoluce využívající porovnávané implementace byl totožný.



Obrázek 7.4: Vývoj průměrné fitness hodnoty rodiče v průběhu evoluce 8bitových aproximačních násobiček s max. $e_{wce} = 1$ %. Počet evaluací je zobrazen na logaritmické stupnici. První parametr u `*Tree` operátorů udává maximální hloubku deaktivovaného podstromu a druhý parametr určuje pravděpodobnost, že se provede mutace místo stromové deaktivace. Parametr u `PointNoSkip` vyjadřuje počet zmutovaných genů.

Závislost doby výpočtu na bitové šířce vstupních operandů je uvedena na obrázku 7.5. Z výsledků je možné vyčíst, že varianta využívající BDD škáluje mnohem lépe než simulace obvodu, avšak asymptotická časová složitost zůstává exponenciální. Pro sčítačky pracující s 6bitovými a menšími operandy se vyplatí použít simulaci, ale pro větší obvody už je lepší použít BDD.

Při porovnání jednotlivých reprezentací obvodů byly pozorovány následující vlastnosti. U varianty využívající simulaci byla nejrychlejší evoluce využívající XMG, druhá nejrychlejší je MIG a nejpomalejší je AIG. Ve všech případech relativní rozdíl činí necelých 10 %. Příčinou je pravděpodobně to, že AIG typicky obsahuje nejvíce aktivních bloků a XMG nejméně. Implementace pomocí BDD vykazuje podobný trend. AIG je nejpomalejší a na menších obvodech je XMG nejrychlejší. Avšak rozdíl mezi MIG a XMG se postupně zmenšuje a u obvodů, které mají větší než 10bitové operandy, začne být MIG rychlejší než XMG. U větších obvodů jsou rozdíly mezi reprezentacemi mnohem výraznější. Například evoluce v AIG trvá přibližně dvakrát tak dlouho jako evoluce v MIG nebo XMG.

Doba výpočtu reprezentací využívajících majority byla následně ještě zkrácena. Zavedená optimalizace využívá faktu, že majorita se třemi vstupy se dá chápat jako programovatelné AND/OR hradlo. Takže při převodu CGP genotypu na BDD lze detekovat, zdali blok realizující majoritu je připojen na konstantní 0 resp. 1 a pokud ano, tak se vytvoří AND resp. OR zbylých dvou vstupů. Tímto se zmenší počet operací nad BDD. V případě varianty MIG bylo dosaženo zrychlení o 3 až 7 %.

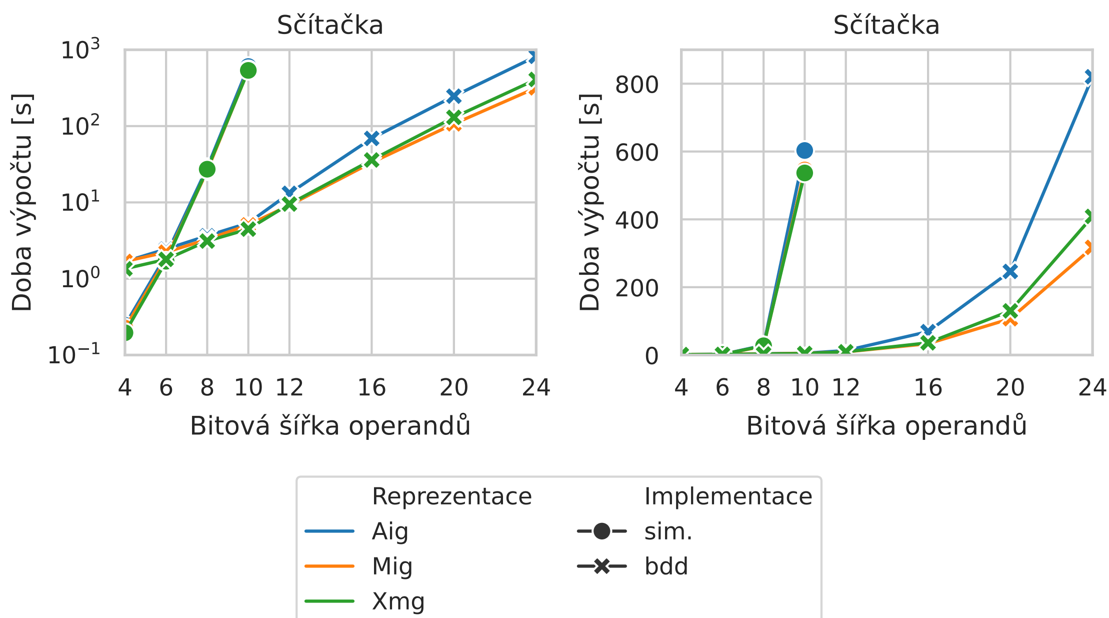
Bylo ukázáno, že BDD dokáže výrazně zrychlit výpočet hodnotících kritérií aproximovaných sčítaček v alternativních reprezentacích. Nevýhodou BDD je velká spotřeba paměti. U testovaných obvodů se pohybovala řádově v jednotkách GB. Další drobnou nevýhodou z hlediska statistického vyhodnocení evoluce je skutečnost, že implementace knihovny BuDDy neumožňuje spouštět více evolučních běhů paralelně z různých vláken. Jedná se však spíše o implementační detail, který se dá obejít použitím paralelismu na úrovni procesů.

7.5 Minimalizace grafové reprezentace

Cílem této sekce je prozkoumat, jakých fitness hodnot lze dosáhnout při evoluci aproximačních obvodů v alternativních reprezentacích v porovnání s evolucí na úrovni hradel.

Pro porovnání byla provedena evoluční aproximace násobiček s 8bitovými operandy a sčítaček se 16bitovými operandy. Oba obvody předpokládají vstupy bez znaménka. Experiment byl proveden pro různé hodnoty maximální absolutní chyby e_{wce}. Byly použity 3 různé fitness funkce počítající součin velikosti a zpoždění (ADP). Ve všech případech je cílem minimalizace fitness hodnoty. CGP využívá evoluční strategii 1 + 4 a k ukončení běhu došlo po prozkoumání 500000 násobiček nebo 100000 sčítaček. Jako mutační operátor byl na základě předchozích experimentů zvolen `SingleTree<5,0.5>`. V případě násobiček bylo pro každou hodnotu e_{wce} provedeno 12 nezávislých běhů a pro sčítačky jich bylo provedeno 24. Výsledky experimentu jsou uvedeny na obrázku 7.6.

První z použitých fitness funkcí (ADP unif. bez inv.) přiděluje každému uzlu v grafu uniformní velikost i zpoždění. Výsledná fitness hodnota se poté vypočte jako součin celkové velikosti všech uzlů a zpoždění nejdelší cesty od primárních vstupů k výstupům. Invertované hrany nejsou počítány. Z výsledků je patrné, že poměr fitness hodnot pro různé reprezentace obvodů zůstává přibližně stejný nezávisle na maximální povolené hodnotě e_{wce}. U násobiček i sčítaček dosahuje nejlepších hodnot XMG a nejhorších AIG. Násobičky v AIG mají $1,67\times$ větší (horší) fitness než MIG, které mají $3\times$ větší fitness než XMG. U sčítaček mají AIG $2,2\times$ větší fitness než MIG, které mají $1,7\times$ větší fitness než XMG. Tento výsledek se dal

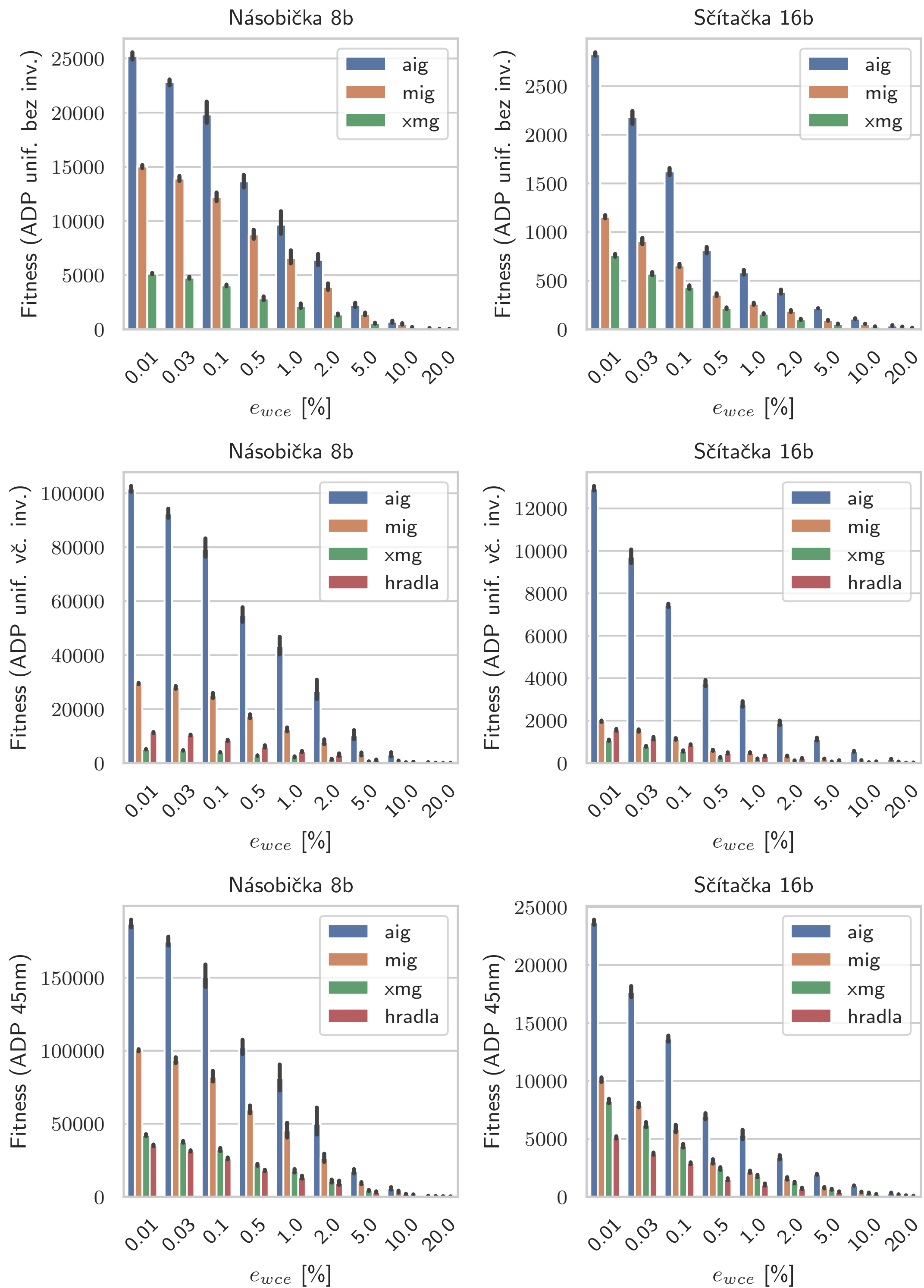


Obrázek 7.5: Závislost průměrné doby výpočtu evolučního algoritmu na bitové šířce vstupních operandů. Doba výpočtu je v levém grafu zobrazena na logaritmické stupnici a v pravém grafu je na lineární stupnici.

očekávat, jelikož AIG mají nejmenší vyjadřovací schopnost a XMG největší. Z porovnání MIG a XMG je evidentní, že přidání XOR do funkční množiny je pro násobičky přínosnější než pro sčítačky.

Druhá fitness funkce (ADP unif. vč. inv.) se liší tím, že i invertorům přiděluje jednotkovou velikost a zpoždění. Pro porovnání byla provedena evoluce využívající dvouvstupé hradla. Z výsledků je možné vyčíst, že pořadí alternativních reprezentací je stejné jako u předchozí fitness funkce, avšak relativní rozdíly mezi reprezentacemi jsou zde větší. Z porovnání nejlepší alternativní reprezentace XMG a hradel bylo zjištěno, že sčítačky i násobičky v XMG mají lepší fitness hodnotu. Zároveň bylo pozorováno, že u násobiček při zvyšování prahové hodnoty e_{wce} se zmenšuje relativní rozdíl mezi XMG a hradly z $2,2\times$ na $1,5\times$. Naopak u sčítaček se vyskytuje opačný trend, kdy relativní rozdíl roste se zvětšující se chybou. Při minimální hodnotě e_{wce} mají hradla $1,5\times$ větší fitness a při maximální chybě mají $2,1\times$ větší fitness.

Třetí fitness funkcí je jednoduché mapování do 45nm technologie (ADP 45nm). Výsledné fitness hodnoty jednotlivých alternativních reprezentací mají podobné relativní rozdíly jako v případě první fitness funkce. Když se porovnají hradla a XMG, tak v tomto případě dosahují hradla lepších fitness hodnot. U násobiček jsou XMG průměrně $1,2\times$ horší a u sčítaček jsou XMG $1,6\times$ horší. Navzdory tomu, že se jedná o poměrně primitivní mapování, tak XMG nemají zase o tolik horší fitness hodnoty než hradla. Z toho důvodu by mohlo být zajímavé zkusit aplikovat sofistikovanější mapovací algoritmus na výsledná kandidátní řešení.



Obrázek 7.6: Porovnání průměrné fitness hodnoty kandidátních řešení nalezených při aproximaci 16bitových sčítaček a 8bitových násobiček.

7.6 Mapování na technologii LUT

Příkladem reálné technologie, která se používá pro realizaci obvodů jsou programovatelná hradlová pole (FPGA). Základním stavebním kamenem FPGA jsou vyhledávací tabulky (LUT). LUT má k vstupů, jeden výstup a provádí zadanou logickou funkci. LUT jsou vzájemně propojené a můžou reprezentovat libovolný logický obvod. Existují algoritmy, které provádí mapování AIG, MIG a XMG na LUT [42]. Cílem této sekce je prozkoumat vhodnost alternativních reprezentací pro evoluční aproximaci obvodů mapovaných do LUT.

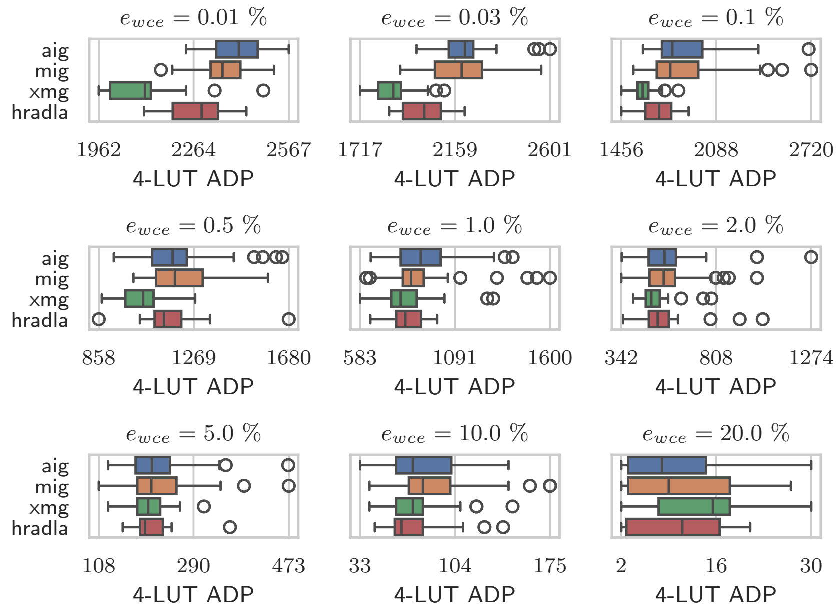
Pro tento experiment byly využity obvody získané v sekci 7.5 (Minimalizace grafové reprezentace). Jedná se o násobičky s 8bitovými vstupy a sčítačky s 16bitovými vstupy. Každé kandidátní řešení bylo namapováno na technologii k-LUT pomocí `mockturtle::lut_map`[3]. V rámci tohoto experimentu byly uvažovány LUT s 4, 5 a 6 vstupy. Jelikož je možné nastavit, jestli se má mapování zaměřit na minimalizaci počtu LUT nebo počtu úrovní LUT, tak pro každé kandidátní řešení vznikly dvě alternativy.

Prvním zkoumaným kritériem je součin počtu LUT a počtu úrovní LUT (LUT ADP). Jelikož pro každé kandidátní řešení existují dvě různá mapování, tak pro tohle porovnání byla uvažována ta varianta, která dosáhla lepší hodnoty LUT ADP.

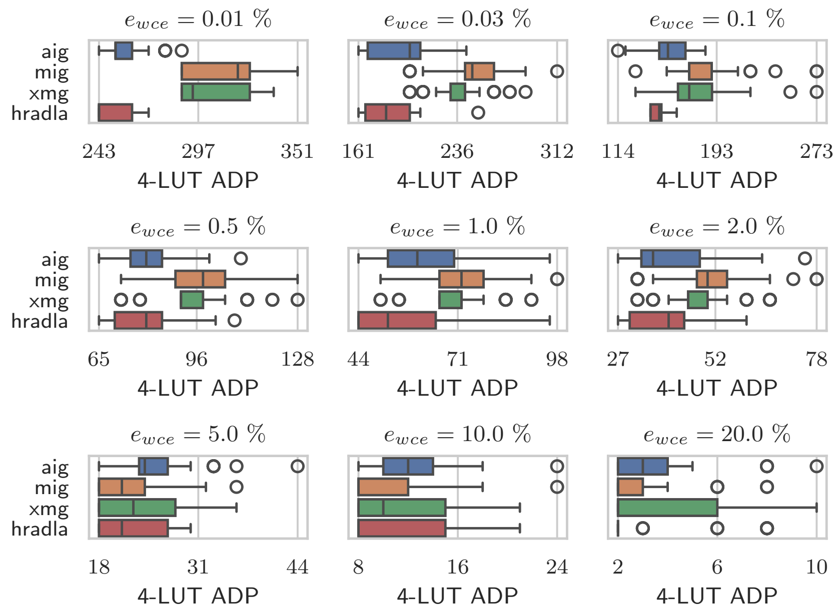
Na obrázku 7.7 jsou uvedeny výsledky mapování na 4-LUT pro 8bitovou násobičku. Při nižších hodnotách e_{wce} jsou výrazné rozdíly mezi reprezentacemi, ale když se e_{wce} zvyšuje, tak mezi nimi přestává být rozdíl. Poslední hodnotou e_{wce}, při které některá z reprezentací vyčnívá mezi ostatními, je 1 %. Nejlepší výsledky byly získány při evoluci v XMG. Naopak nejslabší řešení produkuje AIG. Evoluce na úrovni hradel je druhá nejlepší. Výsledky pro 5-LUT a 6-LUT jsou v příloze A. V případě mapování na 5-LUT jsou výsledky hradel a XMG srovnatelné. U 6-LUT jsou opět pozorovatelné rozdíly mezi hradly a XMG, přičemž XMG produkuje lepší řešení. Jako nejslabší varianta pro 5-LUT a 6-LUT se ukazuje MIG.

Výsledky mapování 16bitových sčítaček na 4-LUT jsou vizualizovány na obrázku 7.8. I u sčítaček platí, že čím větší e_{wce}, tím menší jsou rozdíly mezi jednotlivými reprezentacemi. Pro sčítačky není XMG tak vhodné jako pro násobičky. Nejlepších výsledků bylo totiž dosaženo pomocí evoluce na úrovni hradel. Druhou nejlepší reprezentací je AIG. Hradla se jeví jako nejlepší volba i pro mapování na 5-LUT a 6-LUT. U těchto variant je pozoruhodné, že při dostatečně velkém e_{wce} začne AIG zaostávat i v porovnání s MIG nebo XMG.

[3] https://mockturtle.readthedocs.io/en/latest/algorithms/lut_mapping.html



Obrázek 7.7: Mapování násobiček aproximovaných s využitím generických fitness funkcí na 4-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.



Obrázek 7.8: Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 4-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.

7.7 Fitness funkce zaměřená na LUT mapování

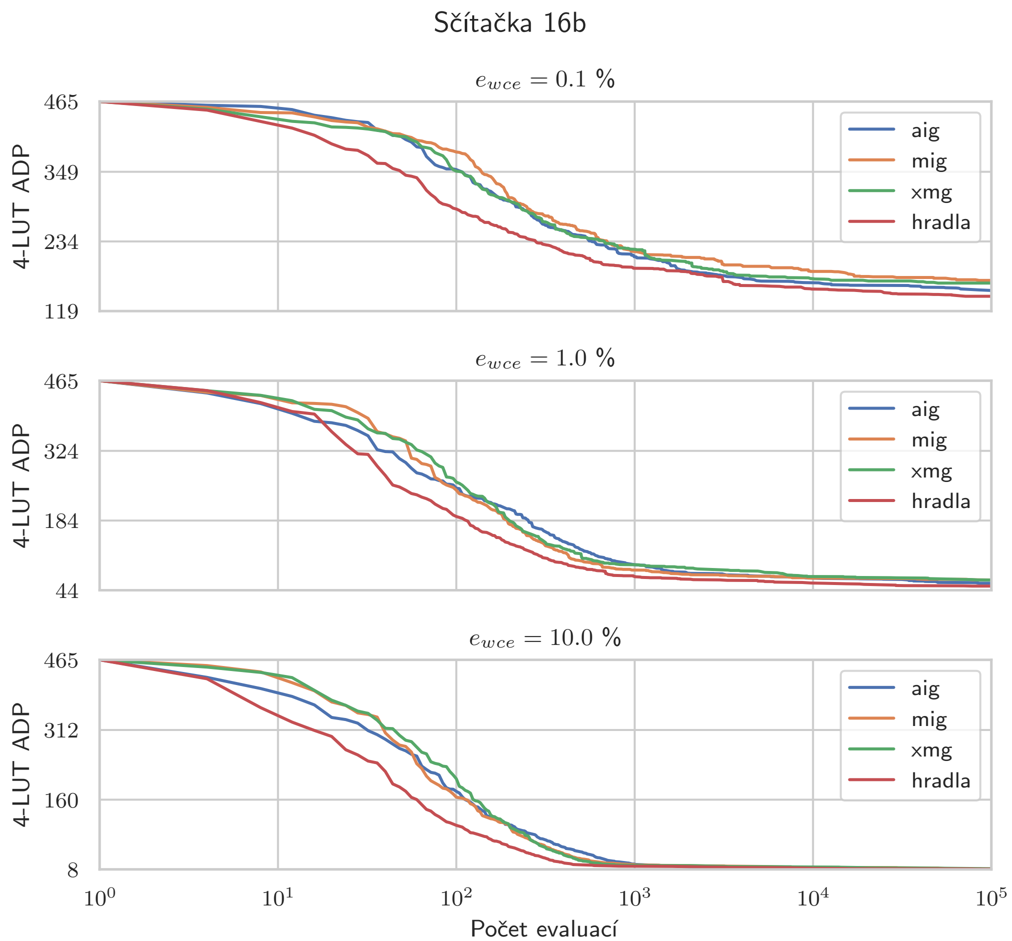
Předchozí experimenty, které se zabývaly mapováním na LUT, využívaly obvody získané pomocí evoluce na generických fitness funkcích. Pokud se však použije fitness funkce zaměřující se na technologii LUT, tak lze získat lepší výsledky [48]. Což je motivace pro následující experiment, ve kterém je cílem porovnat evoluci na úrovni hradel s evolucí v alternativních reprezentacích při použití fitness funkce zaměřené na LUT mapování.

V rámci experimentu byly pomocí CGP hledány aproximační 8bitové násobičky a 16bitové sčítačky. Použitá fitness funkce provádí mapování na 4-LUT se zaměřením na minimalizaci počtu LUT. Fitness hodnota přiřazená každému kandidátnímu řešení je poté vypočtena jako součin počtu LUT a počtu úrovní LUT (4-LUT ADP). CGP využívá evoluční strategii 1 + 4. Evoluce byla ukončena po prozkoumání 100000 sčítaček nebo 500000 násobiček. Použitým mutačním operátorem byl `SingleTree<5,0.5>`. V případě násobiček bylo pro každou hodnotu e_{wce} provedeno 12 nezávislých běhů a pro sčítačky jich bylo provedeno 24.

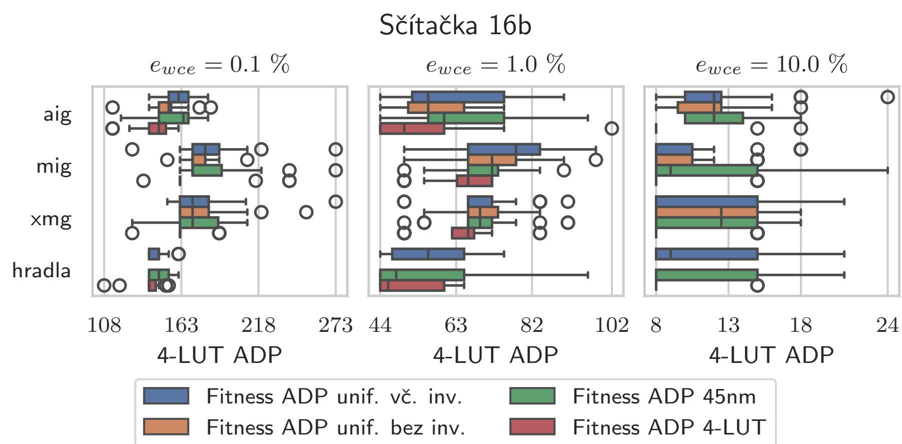
Na obrázku 7.9 jsou uvedeny výsledky získané při evoluci 16bitových sčítaček. Grafy zobrazují vývoj průměrné hodnoty fitness funkce v průběhu evoluce. Při porovnání jednotlivých reprezentací je vidět, že pro všechny uvažované hodnoty e_{wce} se jako nejlepší jeví evoluce na úrovni hradel. Reprezentace pomocí hradel dosahuje nejlepších průměrných výsledků jak na konci evoluce, tak i v libovolně zvoleném okamžiku v průběhu evoluce. Mezi alternativními reprezentacemi nejsou pozorovány výrazné rozdíly. Z těchto průběhů je také vidět, že evoluce, které cílí na nižší aproximační chybu, konvergují pomaleji než evoluce s relativně velkou chybou.

Průběh evoluce násobiček je vykreslen na obrázku 7.11, na kterém jsou zobrazeny průměrné hodnoty fitness funkce v průběhu evoluce. I v tomto případě se jako nejlepší ukazuje evoluce na úrovni hradel. Během počátečních generacích se XMG zdá být nadějné, neboť dosahuje stejných nebo lepších hodnot než hradla. Avšak v pozdější fázi evoluce hradla překonávají XMG a stávají se nejlepší reprezentací. Z hlediska finální průměrné fitness hodnoty je výrazný rozdíl mezi hradly a alternativními reprezentacemi.

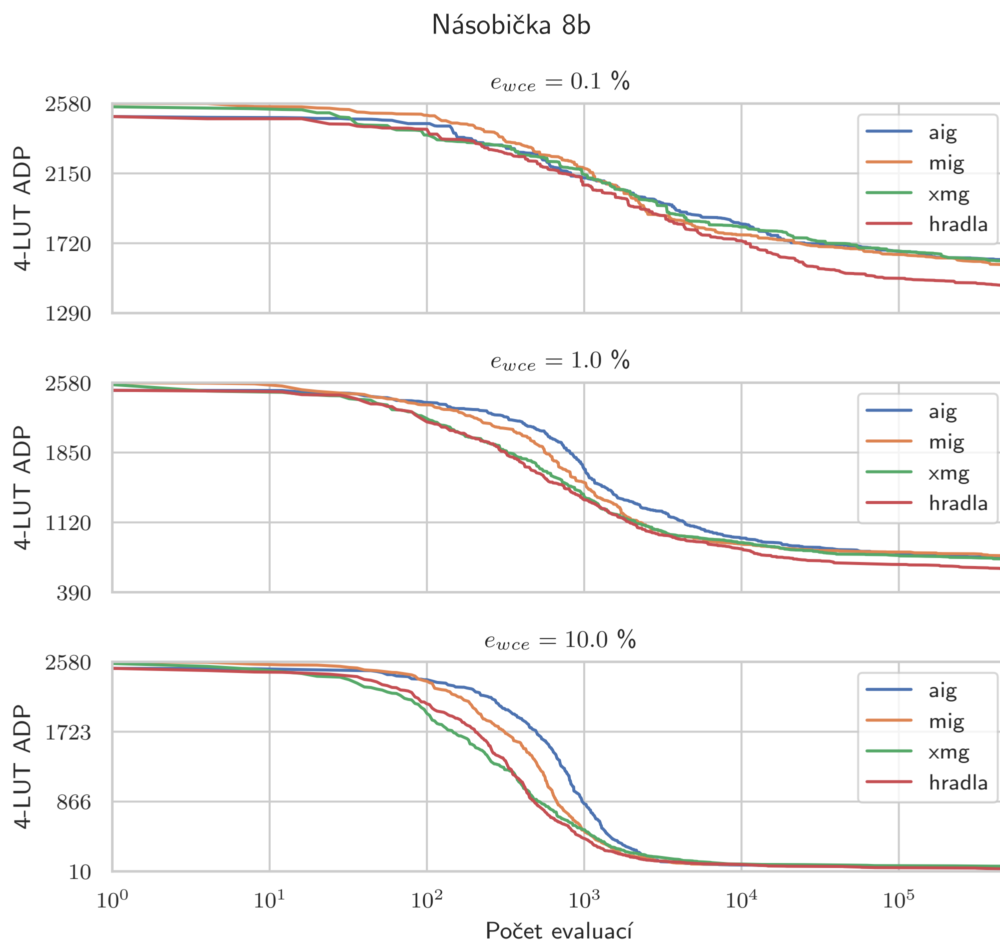
Porovnání s obvody získanými pomocí evoluce využívající generické fitness funkce jsou uvedeny na obrázcích 7.10 a 7.12. Z hlediska fitness funkcí je vidět, že pro sčítačky a násobičky aproximované i v alternativních reprezentacích se vyplatí použít fitness funkci zaměřenou na LUT mapování. U sčítaček se objevuje stejný trend jako při experimentu ze sekce 7.6, kdy MIG a XMG přináší podstatně horší výsledky než AIG nebo hradla. Při použití fitness funkce zaměřené na LUT mapování pro aproximaci násobiček se mezi alternativními reprezentacemi nevyskytují výrazné rozdíly. Nejlepší reprezentací pro násobičky však zůstávají hradla, které poskytují konzistentně lepší výsledky.



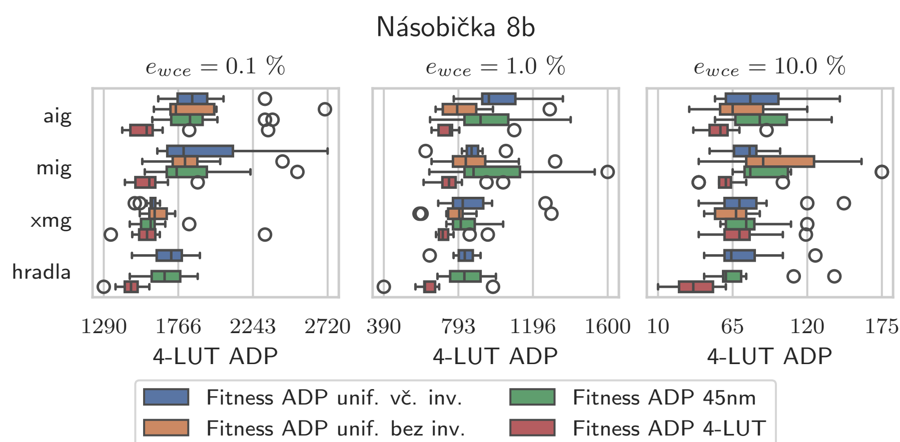
Obrázek 7.9: Vývoj průměrné fitness hodnoty v průběhu evoluce při použití fitness funkce zaměřené na 4-LUT mapování. Na vodorovné ose je použito logaritmické měřítko. Minimum resp. maximum na svislé ose odpovídá fitness hodnotě nejlepšího resp. počátečního obvodu.



Obrázek 7.10: Porovnání fitness funkcí pro reprezentace při evoluci sčítaček. Minimum resp. maximum na vodorovné ose odpovídá fitness hodnotě nejlepšího resp. nejhoršího obvodu.



Obrázek 7.11: Vývoj průměrné fitness hodnoty v průběhu evoluce při použití fitness funkce zaměřené na 4-LUT mapování. Na vodorovné ose je použito logaritmické měřítko. Minimum resp. maximum na svislé ose odpovídá fitness hodnotě nejlepšího resp. počátečního obvodu.



Obrázek 7.12: Porovnání fitness funkcí pro reprezentace při evoluci násobiček. Minimum resp. maximum na vodorovné ose odpovídá fitness nejlepšího resp. nejhoršího obvodu.

Kapitola 8

Závěr

Cílem této práce bylo navrhnout, implementovat a experimentálně vyhodnotit metodu pro evoluční aproximaci obvodů v alternativních reprezentacích.

Nejprve byly představeny reprezentace AIG, MIG a XMG, které se používají pro logickou syntézu. Byly vyzdviženy motivace pro jejich vznik a popsány algoritmy pro práci s nimi. Poté bylo ukázáno kartézské genetické programování jakožto metoda pro automatizovaný návrh obvodů. Následně bylo shrnuto, jak k návrhu aproximačních obvodů přistupují jiní autoři. Zde byla uvedena kritéria používaná pro hodnocení nejen aritmetických aproximačních obvodů, způsoby pro jejich efektivní výpočet založené na paralelní simulaci nebo metod formální analýzy a mutační operátor navržený specificky pro aproximaci obvodů.

Na základě nastudovaných poznatků byla vytvořena implementace systému pro evoluční aproximaci obvodů v alternativních reprezentacích. Ta je realizována v jazyce C++ a zakomponovává existující nástroje pro logickou syntézu a generování aritmetických obvodů.

V experimentech byly využity násobičky, které pracují s 8bitovými vstupy bez znaménka a sčítačky s 16bitovými vstupy bez znaménka. Cílem evoluce bylo ve většině experimentů minimalizovat velikosti obvodu a maximální zpoždění. Obvody však nesměly překročit stanovenou hranici aproximační chyby (maximální absolutní vzdálenost).

Experimenty se nejprve zabývaly profilováním implementace a aplikacemi optimalizací. Byly porovnány různé varianty bitového vektoru (včetně populárních knihovních implementací), který je potřeba pro paralelní simulátor. Žádná z knihovních implementací nebyla vhodná pro výpočet majority a jako nejlepší se ukázala vlastní implementace pomocí `std::vector`, která dokáže dobře využít AVX2 a vyrovnávací paměti.

Při profilování se jako časově nejnáročnější ukázala funkce pro převod výstupu paralelního simulátoru na vektor čísel požadovaný pro výpočet aproximační chyby. Použité optimalizace spočívají v záměně pořadí vnořených smyček, rozbalení nejvnitřnější smyčky a eliminaci větvení. Pro obvody s větším počtem primárních výstupů se vyplatilo využit algoritmus pro efektivní transpozici bitové matice. V případě sčítaček se vyplatilo paralelní simulaci nahradit za přístup založený na BDD.

Z hlediska mutačních operátorů se jako nejefektivnější ukázal operátor single. Ten byl zkombinován s operátorem pro deaktivaci podgrafu. Tato úprava vedla na zlepšení průběhu fitness funkce v průběhu evoluce.

Evoluční aproximace aritmetických obvodů byla nejprve provedena s využitím jednoduchých fitness funkcí, které měří velikost a zpoždění z hlediska počtu uzlů v grafové reprezentaci. Alternativní reprezentace byly porovnány s evolucí na úrovni hradel. Bylo ukázáno, že sčítačky i násobičky v XMG mají menší součin velikosti a zpoždění v porovnání s hradly. AIG a MIG jsou naopak horší než hradla.

Získané aproximační obvody byly namapovány na technologii k-LUT. Ukázalo se, že pro násobičky vede evoluce v XMG na lepší výsledky než evoluce v hradlech. U sčítaček se naopak nejlepší řešení podařilo získat pomocí hradel nebo AIG. Se zvětšující se mírou aproximační chyby se rozdíly mezi reprezentacemi zmenšují.

V závěru byla vyzkoušena fitness funkce, která přímo provádí mapování na k-LUT. Bylo ukázáno, že pro sčítačky i násobičky je nejlepší použít hradla, protože konvergují rychleji než evoluce v alternativních reprezentacích. Při zvětšující se aproximační chybě se zmenšuje doba konvergence u všech reprezentací. Tato fitness funkce dosahuje konzistentně dobrých výsledků v porovnání s jednoduchými fitness funkcemi. Její nevýhodou je však větší výpočetní náročnost.

Literatura

[1] Amarú, L.; Gaillardon, P.-E. a De Micheli, G. Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2014, s. 1–6.

[2] Amarú, L.; Gaillardon, P.-E. a De Micheli, G. Boolean logic optimization in Majority-Inverter Graphs. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, s. 1–6.

[3] Amarú, L.; Gaillardon, P.-E.; Mitra, S. a De Micheli, G. New Logic Synthesis as Nanotechnology Enabler. *Proceedings of the IEEE*, 2015, sv. 103, č. 11, s. 2168–2195.

[4] Bäck, T. a Kok, J. N. *Handbook of Natural Computing*. 1. vyd. Springer Berlin Heidelberg, 2012. ISBN 978-3-540-92911-6.

[5] Biere, A. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[6] Bjesse, P. a Boralv, A. DAG-aware circuit compression for formal verification. In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004*. 2004, s. 42–49.

[7] Brayton, R.; Hachtel, G. a Sangiovanni Vincentelli, A. Multilevel logic synthesis. *Proceedings of the IEEE*, 1990, sv. 78, č. 2, s. 264–300.

[8] Brayton, R. a Mishchenko, A. ABC: An Academic Industrial-Strength Verification Tool. In: Touili, T.; Cook, B. a Jackson, P., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 24–40. ISBN 978-3-642-14295-6.

[9] Češka, M.; Matyáš, J.; Mrázek, V.; Sekanina, L.; Vašíček, Z. et al. Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-bit Multipliers Accomplished. In: *Proceedings of 36th IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Institute of Electrical and Electronics Engineers, 2017, s. 416–423. ISBN 978-1-5386-3093-8.

[10] Češka, M.; Matyáš, J.; Mrázek, V.; Sekanina, L.; Vašíček, Z. et al. SagTree: Towards efficient mutation in evolutionary circuit approximation. *Swarm and Evolutionary Computation*, 2022, sv. 69, s. 100986. ISSN 2210-6502.

[11] Chai, D.; Jiang, J.-H.; Jiang, Y.; Li, Y.; Mishchenko, A. et al. MVSIS 2.0 user's manual, 2003.

[12] Chu, Z.; Shang, C.; Zhang, T.; Xia, Y.; Wang, L. et al. Efficient Design of Majority-Logic-Based Approximate Arithmetic Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2022, sv. 30, č. 12, s. 1827–1839.

[13] Chu, Z.; Soeken, M.; Xia, Y.; Wang, L. a De Micheli, G. Structural rewriting in XOR-majority graphs. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2019, s. 663–668. ASPDAC '19. ISBN 9781450360074.

[14] Deb, K.; Pratap, A.; Agarwal, S. a Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 2002, sv. 6, č. 2, s. 182–197.

[15] Floreano, D. a Mattiussi, C. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008. ISBN 978-0-262-06271-8.

[16] Gamma, E.; Helm, R.; Johnson, R. a Vlissides, J. M. Design Patterns: Elements of Reusable Object-Oriented Software. In:. 1. vyd. Addison-Wesley Professional, 1994, kap. Strategy, s. 315–323. ISBN 0201633612.

[17] Goldman, B. W. a Punch, W. F. Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms. *IEEE Transactions on Evolutionary Computation*, 2015, sv. 19, č. 3, s. 359–373.

[18] Haaswijk, W.; Soeken, M.; Amarù, L.; Gaillardon, P.-E. a De Micheli, G. A novel basis for logic rewriting. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, s. 151–156.

[19] Harding, S. a Banzhaf, W. Implementing cartesian genetic programming classifiers on graphics processing units using GPU.NET. In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2011, s. 463–470. GECCO '11. ISBN 9781450306904.

[20] Holland, J. H. Genetic Algorithms. *Scientific American*. Scientific American, a division of Nature America, Inc., 1992, sv. 267, č. 1, s. 66–73. ISSN 00368733, 19467087.

[21] Hrbacek, R. a Sekanina, L. Towards highly optimized cartesian genetic programming: from sequential via SIMD and thread to massive parallel implementation. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2014, s. 1015–1022. GECCO '14. ISBN 9781450326629. Dostupné z: https://doi.org/10.1145/2576768.2598343.

[22] Keszocze, O. BDD-Based Error Metric Analysis, Computation and Optimization. *IEEE Access*, 2022, sv. 10, s. 14013–14028.

[23] Klhufek, J. a Mrazek, V. ArithsGen: Arithmetics Circuit Generator for HW Accelerators. In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '22)*. 2022, s. 4.

[24] Kluyver, T.; Ragan Kelley, B.; Pérez, F.; Granger, B.; Bussonnier, M. et al. Jupyter Notebooks ? a publishing format for reproducible computational workflows. In: Loizides, F. a Scmidt, B., ed. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, s. 87–90.

[25] Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, Jun 1994, sv. 4, č. 2, s. 87–112. ISSN 1573-1375.

[26] Krzywinski, M. a Altman, N. Visualizing samples with box plots. *Nature Methods*, Feb 2014, sv. 11, č. 2, s. 119–120. ISSN 1548-7105.

[27] Kuehlmann, A.; Paruthi, V.; Krohm, F. a Ganai, M. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2002, sv. 21, č. 12, s. 1377–1394.

[28] Manohararajah, V.; Brown, S. a Vranesic, Z. Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006, sv. 25, č. 11, s. 2331–2340.

[29] Miller, J. F. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, s. 1135–1142. GECCO'99. ISBN 1558606114.

[30] Miller, J. F. Cartesian Genetic Programming. In: Miller, J. F., ed. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 17–34. ISBN 978-3-642-17310-3.

[31] Miller, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*. Springer, 2020, sv. 21, č. 1, s. 129–168.

[32] Mishchenko, A.; Chatterjee, S. a Brayton, R. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In: *Proceedings of the 43rd Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2006, s. 532–535. DAC '06. ISBN 1595933816.

[33] Mittal, S. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery, mar 2016, sv. 48, č. 4. ISSN 0360-0300.

[34] Mrazek, V. Optimization of BDD-based Approximation Error Metrics Calculations. In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, červenec 2022.

[35] Mrazek, V. a Vasicek, Z. Evolutionary design of large approximate adders optimized for various error criteria. In: *Proceedings of the Genetic and Evolutionary*

Computation Conference Companion. New York, NY, USA: Association for Computing Machinery, 2018, s. 294–295. GECCO '18. ISBN 9781450357647.

[36] Porod, W. Quantum-dot devices and Quantum-dot Cellular Automata. *Journal of the Franklin Institute*, 1997, sv. 334, č. 5, s. 1147–1175. ISSN 0016-0032. Visions of Nonlinear Mechanics in the 21st Century.

[37] Sandberg, M. What is SSE !@# good for? Transposing a bit matrix. *Coding on the edges* online. 24. července 2011. Dostupné z: https://mischasan.wordpress.com/2011/07/24/what-is-sse-good-for-transposing-a-bit-matrix/. [cit. 2024-05-04].

[38] Sekanina, L. Evolutionary Algorithms in Approximate Computing: A Survey. *Journal of Integrated Circuits and Systems*, 2021, sv. 16, č. 2, s. 1.

[39] Sekanina, L. a Vasicek, Z. Approximate Circuit Design by Means of Evolvable Hardware. In: *2013 IEEE International Conference on Evolvable Systems (ICES)*. Singapur: IEEE Computer Society, 2013, s. 21–28. Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). ISBN 978-1-4673-5847-7.

[40] Sentovich, E.; Singh, K.; Lavagno, L.; Moon, C.; Murgai, R. et al. SIS: A System for Sequential Circuit Synthesis, 1998.

[41] Soeken, M.; Amarù, L. G.; Gaillardon, P.-E. a Micheli, G. de. Optimizing Majority-Inverter Graphs with functional hashing. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2016, s. 1030–1035.

[42] Soeken, M.; Riener, H.; Haaswijk, W.; Testa, E.; Schmitt, B. et al. *The EPFL logic synthesis libraries*. červen 2022. ArXiv:1805.05121v3.

[43] Soeken, M.; Roetteler, M.; Wiebe, N. a De Micheli, G. Design automation and design space exploration for quantum computers. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, s. 470–475.

[44] Stanley Marbell, P.; Alaghi, A.; Carbin, M.; Darulova, E.; Dolecek, L. et al. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery, jun 2020, sv. 53, č. 3. ISSN 0360-0300.

[45] std::vector<bool>. *Cppreference.com* online. 29. dubna 2023. Dostupné z: https://en.cppreference.com/w/cpp/container/vector_bool. [cit. 2024-05-03].

[46] Tsai, J. Bit-Matrix Transpose. *Github.com* online. 31. srpna 2015. Dostupné z: https://github.com/dsnet/matrix-transpose. [cit. 2024-05-04].

[47] Vasicek, Z. Formal Methods for Exact Analysis of Approximate Circuits. *IEEE Access*, 2019, sv. 7, s. 177309–177331.

[48] Vasicek, Z. Synthesis of approximate circuits for LUT-based FPGAs. In: *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2021, s. 17–22.

[49] Vasicek, Z. a Sekanina, L. Hardware Accelerators for Cartesian Genetic Programming. In: O'Neill, M.; Vanneschi, L.; Gustafson, S.; Esparcia Alcázar, A. I.; De Falco, I. et al., ed. *Genetic Programming.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 230–241. ISBN 978-3-540-78671-9.

[50] Vasicek, Z. a Sekanina, L. Circuit Approximation Using Single- and Multi-objective Cartesian GP. In:. Duben 2015, sv. 9025, s. 217–229. ISBN 978-3-319-16500-4.

[51] Vasicek, Z. a Sekanina, L. Evolutionary Approach to Approximate Digital Circuits Design. *IEEE Transactions on Evolutionary Computation*, 2015, sv. 19, č. 3, s. 432–444. ISSN 1089-778X.

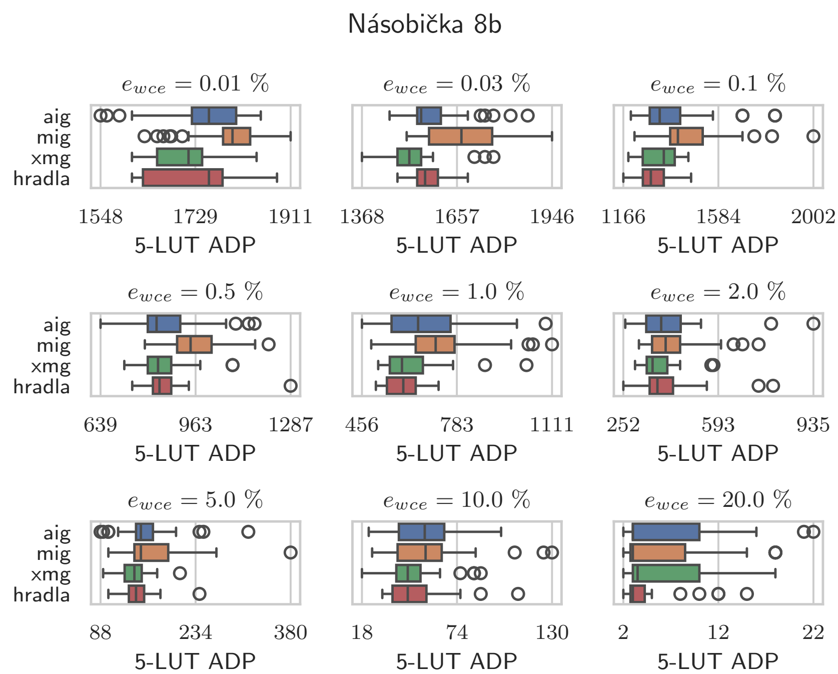
[52] Vasicek, Z. a Slany, K. Efficient Phenotype Evaluation in Cartesian Genetic Programming. In: Moraglio, A.; Silva, S.; Krawiec, K.; Machado, P. a Cotta, C., ed. *Genetic Programming.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 266–278. ISBN 978-3-642-29139-5.

[53] Vassilev, V. K. a Miller, J. F. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Miller, J.; Thompson, A.; Thomson, P. a Fogarty, T. C., ed. *Evolvable Systems: From Biology to Hardware.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, s. 252–263. ISBN 978-3-540-46406-8.

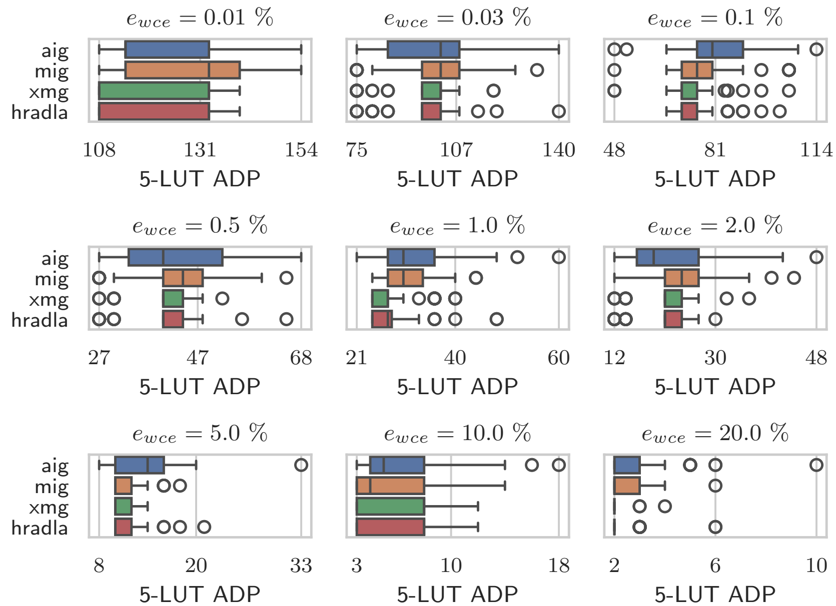
[54] Yanling, Z.; Bimin, D. a Zhanrong, W. Analysis and study of perceptron to solve XOR problem. In: *The 2nd International Workshop on Autonomous Decentralized System, 2002.* 2002, s. 168–173.

Příloha A

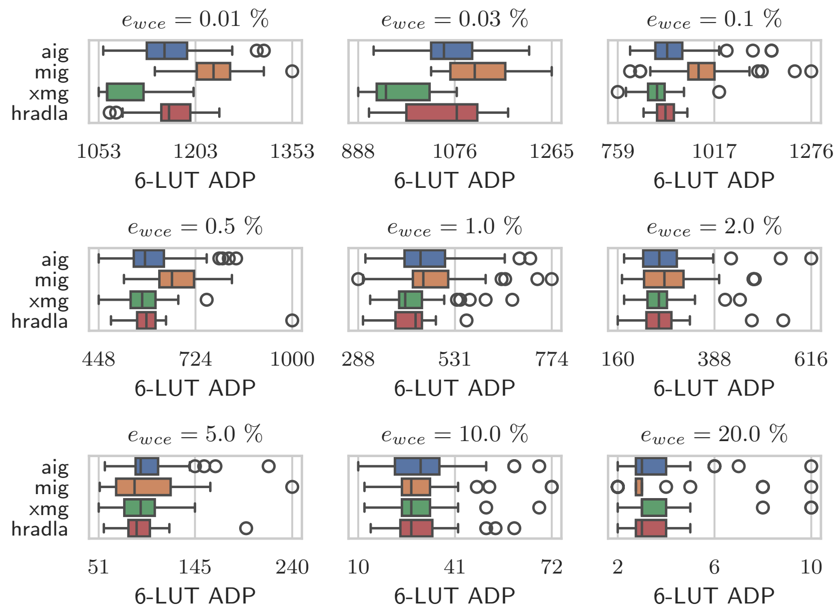
Další výsledky experimentů



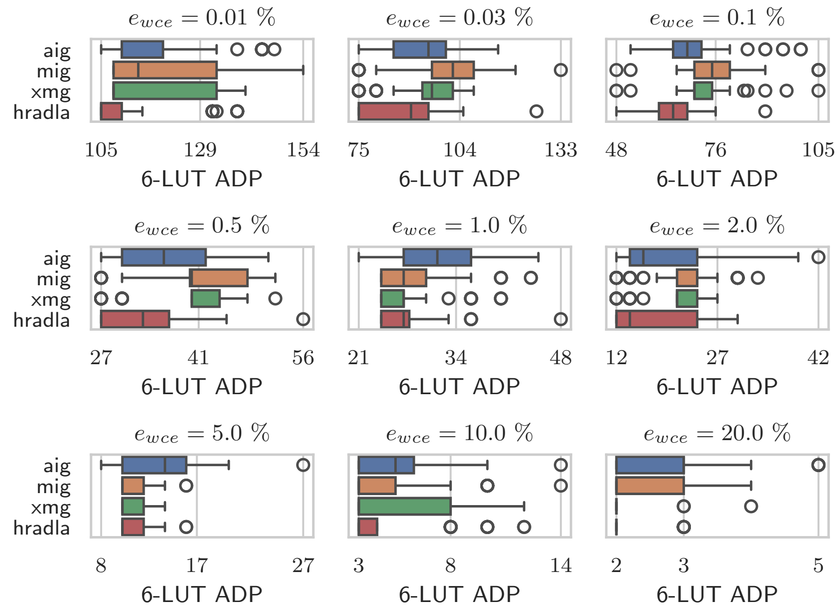
Obrázek A.1: Mapování násobiček aproximovaných s využitím generických fitness funkcí na 5-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.



Obrázek A.2: Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 5-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.



Obrázek A.3: Mapování násobiček aproximovaných s využitím generických fitness funkcí na 6-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.



Obrázek A.4: Mapování sčítaček aproximovaných s využitím generických fitness funkcí na 6-LUT. Minimum resp. maximum na vodorovné ose odpovídá součinu počtu LUT a počtu úrovní LUT pro nejlepší resp. nejhorší řešení.

Příloha B

Obsah paměťového média a návod na spuštění

Přiložené paměťové médium obsahuje:

- `arihs_gen/` – knihovna ArithsGen [23]
- `figures/` – vygenerované grafy
- `latex/` – zdrojové soubory této technické zprávy
- `lib/` – potřebné knihovny
- `logs/` – naměřené výsledky
- `results_analysed/` – mapování nejlepších řešení na cílovou technologii
- `seeds/` – počáteční obvody
- `source/cgp_*.hpp` – vytvořená implementace CGP
- `source/ex_*.cpp` – spouštění experimentů
- `source/main_*.cpp` – utility pro převod do CGP, analýzu počátečních obvodů a syntézu nejlepších řešení do LUT
- `ex_*.ipynb` – analýza výsledků a generování grafů
- `gen_circ.py` – generování počátečních obvodů
- `text.pdf` – tento dokument

Pro překlad je potřeba program CMake (testováno na verzi 3.25). Ukázka použití vytvořené knihovny je uvedena v souboru `source/main_example.cpp`. Samotný překlad a spuštění se na počítači s operačním systémem Ubuntu provede pomocí příkazů:

```
$ cmake -DCMAKE_BUILD_TYPE=Release -S . -B build
$ cmake --build build --target cgp_example
$ ./build/source/cgp_example
$ cat example_details.csv # vypis prubehu evoluce
$ cat example_result.csv # vypis nejlepsiho nalezeneho reseni
```

Výpis B.1: Ukázka překladu a spuštění.