Sem vložte zadání Vaší práce.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

# Určování míry podobnosti logických obvodů založené na grafových algoritmech

***Bc. Janusz Piotr Wijas***

Katedra číslicového návrhu
Vedoucí práce: doc. Ing. Petr Fišer, Ph.D.

21. ledna 2021

# Poděkování

Děkuji svému vedoucímu práce doc. Ing. Petru Fišerovi, Ph.D. za pečlivé a trpělivé vedení práce a časté konzultace.

Computational resources were supplied by the project „e-Infrastruktura CZ“ (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 21. ledna 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Janusz Piotr Wijas. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Wijas, Janusz Piotr. *Určování míry podobnosti logických obvodů založené na grafových algoritmech.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

# Abstrakt

Tato práce se věnuje výpočtu podobností logických obvodů a zkoumáním závislosti „common-mode“ poruch na těchto podobnostech.

Nejprve je provedena analýza existujících řešení se zaměřením na grafové algoritmy a návrh řešení s využitím varianty existujicího algoritmu.

Výsledkem je implementace zvolené varianty algoritmu a měření závislosti získané podobnosti s množstvím odsimulovaných „common-mode“ poruch v daných obvodech.

Tyto výsledky jsou poté porovnány s řešením vedoucího práce.

**Klíčová slova** Grafová editační vzdálenost, porovnávání grafů, podobnost grafů, ATPG, logické obvody, grafové algoritmy

# Abstract

This thesis deals with calculating of logical circuits diversity and dependency of such diversity with common-mode faults.

Firstly we provide research of existing algorithms concerning graph diversity. Secondly we propose variant of existing algorithm to calculate diversity and thirdly we correlate outputs of said algorithm with simulated common-mode faults.

The results are compared with results of the supervisor.

**Keywords** Graph edit distance, graph matching, graph similarity, ATPG, logical circuits, graph algorithms

# Obsah

**Úvod** **1**
Motivace . . . 1
Cíle práce . . . 2
Členění práce . . . 2

**1 Důležité pojmy** **3**
1.1 Definice grafů . . . 3
1.1.1 Orientované grafy . . . 3
1.1.2 Neorientované grafy . . . 4
1.1.3 Cykly . . . 4
1.1.4 Bipartitní graf . . . 4
1.2 Grafová editační vzdálenost . . . 5
1.3 Logické obvody . . . 7
1.3.1 Kombinační logické obvody . . . 7
1.3.2 Sekvenční logické obvody . . . 8
1.3.3 Testování logických obvodů . . . 8
1.4 Poruchy . . . 9
1.4.1 Trvalé poruchy . . . 9
1.4.2 Přechodné poruchy . . . 10
1.4.3 Common-mode poruchy . . . 10
1.5 Algoritmy generování testu . . . 11
1.6 Korelace dat . . . 12
1.6.1 Personův korelační koeficient . . . 12

**2 Analýza dosavadních řešení** **13**
2.1 Grafová editační vzdálenost . . . 13
2.2 Přiřazovací problém a bipartitní graf . . . 14
2.3 SimRank . . . 15
2.4 Funkční ekvivalence . . . 16

2.5 Porovnávání grafů s počátečním párováním uzlů . . . . . 16

**3 Návrh** **19**
3.1 Zjištění grafové diverzity . . . . . 20
3.1.1 Logické obvody . . . . . 20
3.1.2 Atributovaný graf . . . . . 20
3.2 Simulace obvodů . . . . . 21
3.2.1 Vstupní vektory . . . . . 21
3.2.2 Common-mode poruchy . . . . . 22
3.2.3 MetaCentrum . . . . . 22
3.3 Korelace dat . . . . . 22

**4 Implementace** **23**
4.1 Algoritmus výpočtu grafové podobnosti . . . . . 23
4.1.1 Přiřazování uzlů dle indexu . . . . . 23
4.1.2 Náhodné přiřazování uzlů . . . . . 24
4.2 Algoritmus simulace obvodů . . . . . 25
4.3 Algoritmus výpočtu common-mode poruch . . . . . 25
4.4 Algoritmus výpočtu korelačního koeficientu . . . . . 26

**5 Realizace** **27**
5.1 Měření podobnosti grafů . . . . . 27
5.2 Generování vstupních vektorů . . . . . 27
5.3 Měření simulace obvodů . . . . . 27
5.4 Měření common-mode poruch . . . . . 28
5.5 Měření korelace dat . . . . . 29

**6 Analýza výsledků** **31**
6.1 Common-mode poruchy . . . . . 31
6.2 Podobnost grafů . . . . . 31
6.2.1 Náhodné přiřazování uzlů . . . . . 33
6.2.2 Přiřazování uzlů dle indexu . . . . . 33
6.3 Korelace dat . . . . . 34
6.3.1 Náhodné přiřazování uzlů . . . . . 34
6.3.2 Přiřazování uzlů dle indexu . . . . . 35
6.3.3 Relativní počet „common-mode“ poruch . . . . . 35
6.4 Porovnání výsledků . . . . . 36
6.4.1 Náhodné přiřazování uzlů . . . . . 36
6.4.2 Přiřazování uzlů dle indexu . . . . . 37
6.4.3 Relativní počet „common-mode“ poruch . . . . . 37

**Závěr** **41**

**Literatura** **43**

**A Seznam použitých zkratek** **47**

**B Skripty pro MetaCentrum** **49**

**C Obsah přiloženého USB** **53**

# Seznam obrázků


1.1 Prostý neorientovaný graf . . . . . 3
1.2 Prostý, orientovaný a acyklický graf . . . . . 4
1.3 Bipartitní graf, Hvězda . . . . . 5
1.4 Příklad editační cesty z grafu $G^p$ (nalevo) do grafu $G^q$ (napravo) . 6
1.5 Finální mapování z grafu $G^p$ do grafu $G^q$ podle 1.4 . . . . . 7
1.6 Kombinační logický obvod . . . . . 7
1.7 Sekvenční logický obvod . . . . . 8
1.8 Vizualizace pravidla deseti. . . . . 9
1.9 Pokrytí defektů poruchovým modelem. . . . . 10
1.10 Ukázkový duplexní systém . . . . . 11

2.1 Ukázka fungování SimRank. . . . . 15
2.2 Příklad mapování mezi dvěma sociálníma sítěmi. . . . . 16

3.1 Schéma procesu zjištění korelace podobnosti obvodů a „common-mode“ poruch. . . . . 19

6.1 Histogram „common-mode“ poruch mezi obvody. . . . . 32
6.2 Celkový počet „common-mode“ poruch mezi jednotlivými obvody. 32
6.3 Podobnost jednotlivých obvodů dle vedoucího práce . . . . . 33
6.4 Podobnost jednotlivých obvodů při náhodném párování. . . . . 34
6.5 Podobnost jednotlivých obvodů při metodě párování pomocí indexů. 35
6.6 Korelace podobností obvodů dle vedoucího práce a simulovaných „common-mode“ poruch. . . . . 36
6.7 Korelace podobností obvodů při náhodném přiřazování a množství „common-mode“ poruch. . . . . 37
6.8 Korelace podobností obvodů při přiřazování pomocí indexů a množství „common-mode“ poruch. . . . . 38
6.9 Korelace podobností obvodů a poměru „common-mode“ poruch v obvodu. . . . . 38

6.10 Rozdíl korelačních koeficientu mezi řešením přiřazování pomocí indexu a řešením vedoucího práce. . . . . 39
6.11 Rozdíl korelačních koeficientu mezi množstvím „common-mode“ poruch a jejich poměrem ke všem nalezeným poruchám. . . . . 39

# Seznam tabulek

5.1 Prostředky použité pro výpočet „common-mode“ poruch . . . . . 29

6.1 Výsledné korelační koeficienty. . . . . 36

# Úvod

## Motivace

V dnešní době je testování obvodů důležitou součástí výroby. Cena testování může dosahovat až 30 % celkové ceny obvodu. Vyšší cena testování se ale vyplatí, jelikož brzké odhalení poruchy zásadně redukuje náklady, které je potřeba vynaložit na odstranění daného problému [1].

Logické obvody mají v případě poruchy často na výstupu špatný výsledek. To může mít za následek špatné rozhodnutí systému, a v závislosti na činnosti systému, i kritické selhání, škodu na majetku nebo smrt osob. Z toho důvodu se používají různé typy zabezpečení. Jedním z typů je přidání druhé kopie obvodu a porovnání výsledků, takzvaný duplex. Každý vstup je v takovém případě počítán vícekrát a výsledky jsou porovnány. V případě, že se výsledky neshodují, je tato informace předána do systému.

Tento způsob dokáže zabránit velkému množství chyb, nicméně existuje druh poruch (neboli modelů fyzických poškození na logické úrovni), které tento typ ochrany nedokáže bezpečně zachytit [2]. Jedná se o takzvané „common-mode“ poruchy. Tento typ poruch se projeví ve více obvodech zároveň a způsobí, že výstup obvodů, i když je stejný, není správný. Jelikož porovnávací jednotka pouze porovnává výstupy, chybu nedokáže zachytit a označí výstup jako správný. Z toho důvodu duplex i přesto, že zvětšuje odolnost systému proti obecným poruchám, nechrání dostatečně proti „common-mode“ poruchám.

Na ochranu systému proti „common-mode“ poruchám je proto potřeba myslet již při návrhu obvodů. V případě, že obě jednotky počítající stejné vstupy si nejsou podobné, je šance „common-mode“ poruchy mnohem menší. Z toho důvodu se snažíme dokázat, že čím jsou dva logické obvody odlišnější, tím je menší množství „common-mode“ poruch.

## Cíle práce

V této práci budu zkoumat existující algoritmy pro výpočet podobnosti dvou logických obvodů (se zaměřením na grafové algoritmy). Vybraný algoritmus implementuji nebo vytvořím vlastní variantu. S jeho pomocí se budu zabývat výpočtem podobnosti obvodů a vlivu podobnosti na množství „common-mode“ poruch. Předpokládám, že čím odlišnější obvody, tím bude menší množství hledaných poruch. Výsledky porovnám mezi sebou a s výsledky vedoucího práce. Zároveň se budu snažit, aby byl algoritmus použitelný pro obvody s větším množstvím uzlů, tedy aby jeho časová náročnost nebyla příliš vysoká a tudíž byl algoritmus dobře škálovatelný.

## Členění práce

Práce je rozčleněna do osmi kapitol včetně úvodu a závěru. První kapitola obsahuje popis jednotlivých pojmů použitých v této práci.

Druhá kapitola zkoumá vybraná existující řešení porovnávání grafů a obvodů.

Ve třetí kapitole navrhuji možné řešení problému porovnávání grafů, způsob simulace obvodů a koncovou korelaci získaných dat.

Čtvrtá kapitola zkoumá implementaci jednotlivých částí porovnávání grafů.

V páté kapitole popisuji způsob spouštění jednotlivých částí navrženého programu, vzniklé problémy a jejich řešení.

V poslední kapitole na základě výsledků z předchozí kapitoly porovnávám a hodnotím získaná data.

Kapitola 1

# Důležité pojmy

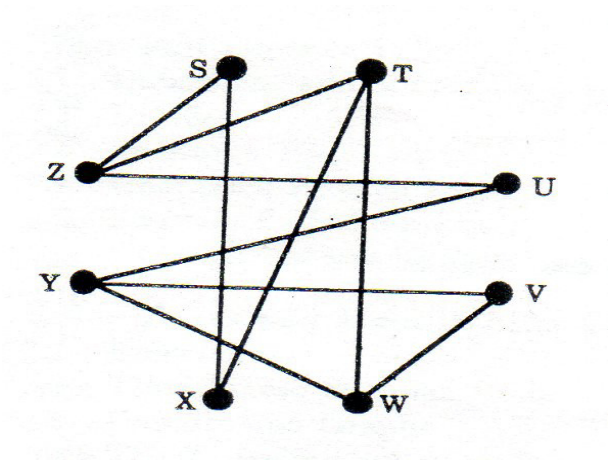
Tato kapitola představuje základní pojmy použité v této diplomové práci.

## 1.1 Definice grafů

Dle [3] je graf uspořádaná dvojice konečných množin uzlů a hran, kde hrana je spojení mezi jedním nebo dvěma uzly. Pokud není řečeno jinak, považujeme graf za prostý, tedy takový, který nemá žádné rovnoběžné hrany. Grafy můžeme dělit na orientované a neorientované.

### 1.1.1 Orientované grafy

Orientované grafy jsou takové grafy, pro které jsou hrany uspořádanou dvojicí uzlů [4]. Znamená to, že hrana *(a, b)* je hrana jdoucí z uzlu *a* do uzlu *b*.




Obrázek 1.1: Prostý neorientovaný graf.[1]

### 1.1.2 Neorientované grafy

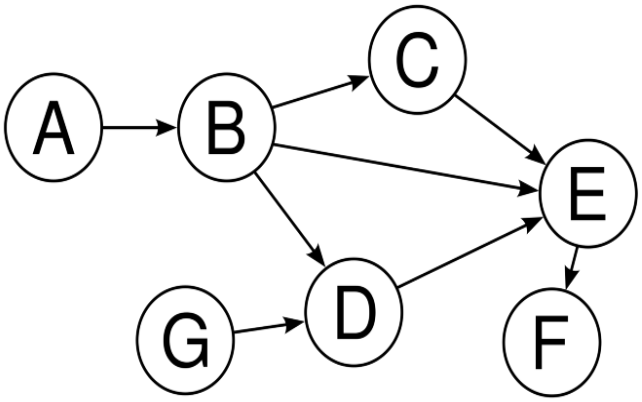
Neorientovaný graf můžeme považovat za podmnožinu orientovaného [5]. Znamená to, že každá hrana $\{a, b\}$ neorientovaného grafu odpovídá dvěma hranám orientovaného grafu a to *(a, b)* a *(b, a)*. Neboli můžeme hranu používat oběma směry stejně.

### 1.1.3 Cykly

V teorii grafů se cyklem nazývá cesta[2], jejíž první a poslední prvek je totožný. V případě orientovaného grafu nemůžeme jít proti směru hran.

#### 1.1.3.1 Orientovaný acyklický graf

Acyklický graf je takový graf, který neobsahuje žádné cykly, viz obrázek 1.2.




Obrázek 1.2: Prostý, orientovaný a acyklický graf.[3]

### 1.1.4 Bipartitní graf

Bipartitní graf je takový graf, jehož uzly můžeme rozdělit do dvou neprázdných podmnožin $U$ a $V$ tak, že všechny hrany daného grafu spojují podmnožiny $U$ a $V$ [8].

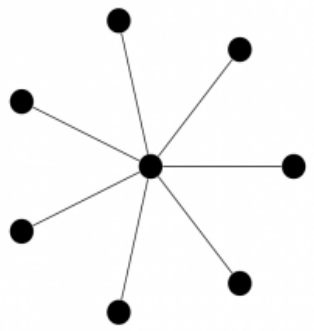
#### 1.1.4.1 Hvězda

Hvězdou nazýváme takový bipartitní graf, jehož velikost jedné z množin je rovna jedné [4], jak je vidět na obrázku 1.3.

[1] Převzato a upraveno z [3].

[2] Cestou nazýváme uspořádanou n-tici uzlů, ve které každý soused je také sousedem v grafu [6].

[3] Převzato a upraveno z [7].




Obrázek 1.3: Bipartitní graf, hvězda.[4]

## 1.2 Grafová editační vzdálenost

Grafová editační vzdálenost – GED (**G**raph **E**dit **D**istance) je flexibilní, aproximační mechanismus, který nám pomáhá určit, jaká je vzdálenost mezi dvěma grafy (tedy jak moc se tyto grafy liší). Je to nejčastěji používaná metoda pro aproximační porovnávání grafů [9]. Tento mechanismus je vyvíjený už od 80. let (viz [10]) a jsou tedy dostupné různé verze jeho implementace.

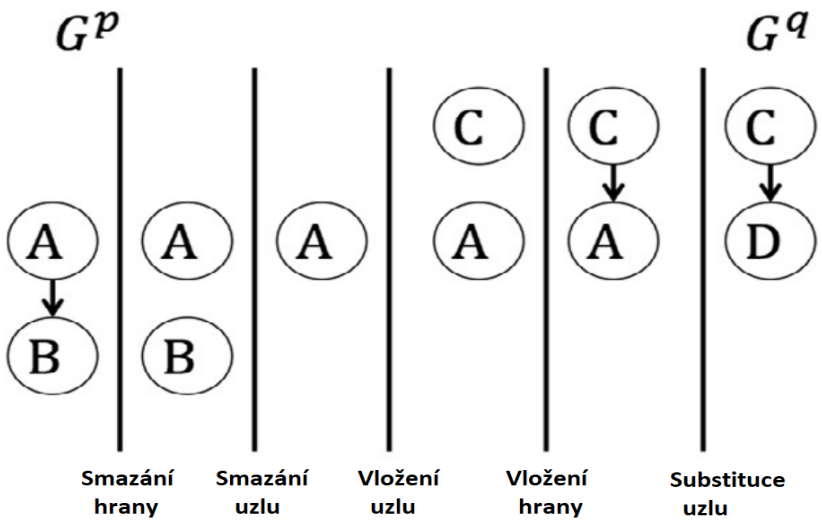
Graf můžeme přetransformovat z jednoho na druhý, za pomoci sekvence konečného množství změn. Tyto změny a jejich cena mohou být v různých algoritmech definovány různým způsobem, ale GED hledá sekvenci takovou, aby její cena byla nejnižší. Příklad můžeme vidět na obrázku 1.4. Konečnou cenu této transformace můžeme spočítat jako součet cen jednotlivých operací. Je zřejmé, že hlavním problémem těchto algoritmů je určení podobnosti jednotlivých komponent v grafech a cen přechodů [10].

Základními operacemi pro různé algoritmy jsou:

- Vložení hrany
- Vložení uzlu
- Smazání hrany
- Smazání uzlu
- Substituce hrany
- Substituce uzlu

Samozřejmě v závislosti na algoritmu a grafech, pro který je algoritmus vyvíjen, nejsou některé operace použité. Jako příklad si můžeme představit

graf v němž jsou všechny uzly stejné. V takovém případě je zřejmé, že není nutné uzly substituovat.




Obrázek 1.4: Příklad editační cesty z grafu $G^p$ (nalevo) do grafu $G^q$ (napravo).[5]

Na obrázku 1.4 vidíme jednoduché použití operací pro transformaci grafu $G^p$ do grafu $G^q$ za použití postupně operací:

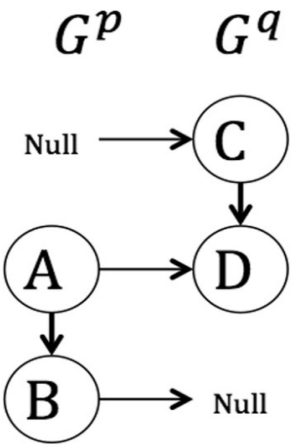
- Smazání hrany (A, B)
- Smazání uzlu B
- Vložení uzlu C
- Vložení hrany (C, A)
- Substituce uzlu A na uzel D

Pokud by cena všech operací byla nastavena na 10, výslednou cenu bychom mohli určit sečtením cen operací a v tomto případě by to vyšlo celkem 50. Výsledné mapování lze vidět na obrázku 1.5.

Nicméně [11] podotýká, že přesné porovnávání grafů je NP-problém. Proto existuje mnoho heuristických metod pro nalezení porovnání se sníženou výpočetní cenou (dle téhož článku).

Dle [10] se k implementaci heuristických algoritmů nejčastěji používají SOM (self organizing map, popis algoritmu např. v [12]) nebo Dijkstrův algoritmus (popis algoritmu např. v [13]). Nicméně [14], ze kterého vycházíme v této práci, používá jako základní ukázkový algoritmus $A^*$.

[5] Převzato a upraveno z [9].
[6] Převzato a upraveno z [9].



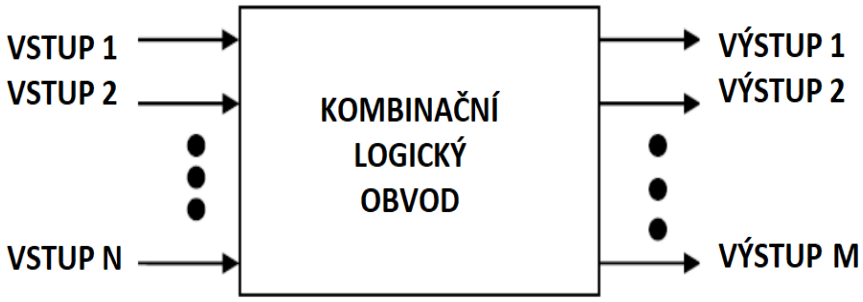

Obrázek 1.5: Finální mapování z grafu $G^p$ do grafu $G^q$ podle obrázku 1.4.[6]

## 1.3 Logické obvody

Logické obvody dělíme nejčastěji na sekvenční a kombinační.

### 1.3.1 Kombinační logické obvody

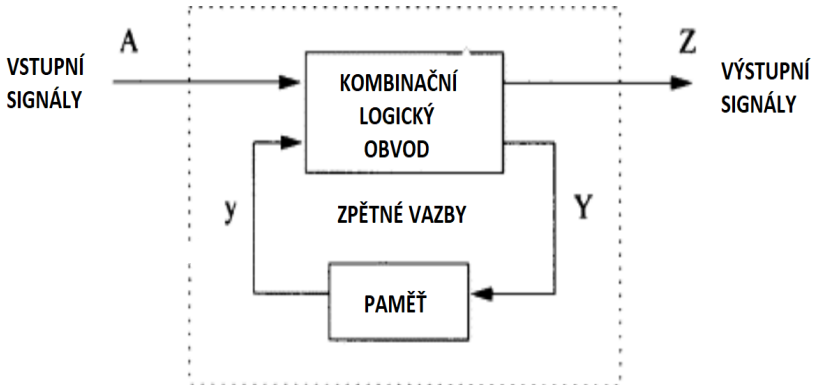
Dle [15] jsou kombinační logické obvody implementace funkce přijímající $n$ externích vstupů a poskytující $m$ logických výstupů (viz obrázek 1.6). Dle [16] jsou hodnoty výstupních proměnných závislé pouze na hodnotách vstupních proměnných ve stejném časovém okamžiku. Znamená to, že stejný vstup v jakémkoli časovém okamžiku by vždy měl zaručit stejný výstup. Tyto obvody lze tedy popsat logickou funkcí. Dle [17] je můžeme taky zobrazit jako orientovaný acyklický graf (viz kapitola 1.1.3.1).




Obrázek 1.6: Kombinační logický obvod.[7]

### 1.3.2 Sekvenční logické obvody

U sekvenčních logických obvodů závisí dle [18] výstupy na vstupech jako u kombinačních obvodů, ale zároveň také na vstupech předchozích. V každém kroku je v obvodu ukládán stav, který poté zpětně ovlivňuje chování obvodu v dalších krocích, jak je vidět na obrázku 1.7. Tato zpětná vazba je realizovaná zpětnovazebnými registry, viz [19]. Matematicky lze tyto obvody popsat konečnými automaty (**F**inite **S**tate **M**achine) viz [16].




Obrázek 1.7: Sekvenční logický obvod.[8]

### 1.3.3 Testování logických obvodů

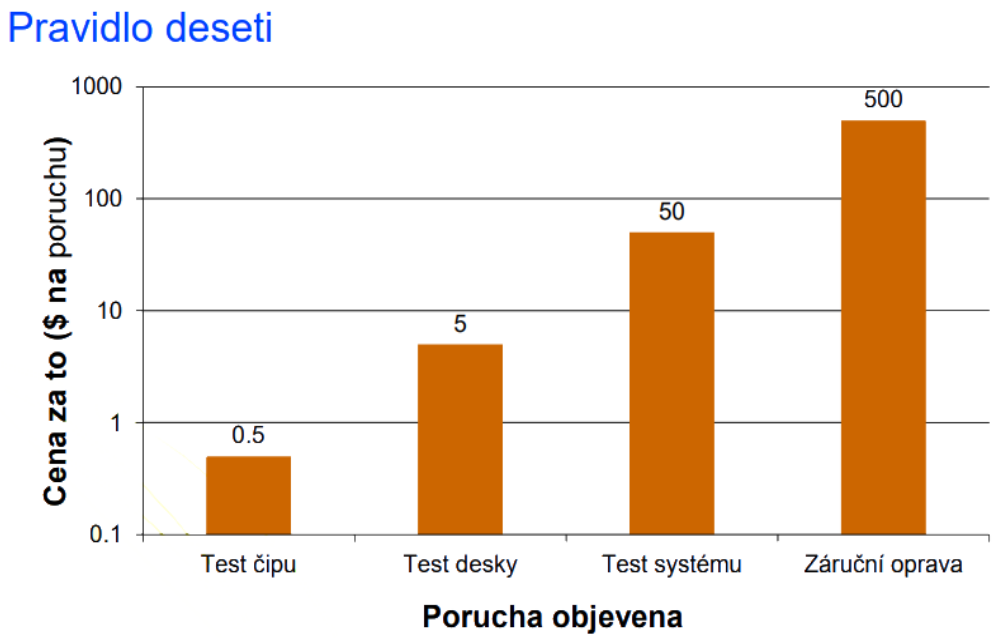
Cena testování v současné době může tvořit kolem 30 % ceny čipů, nicméně je to proces, který v konečném důsledku šetří peníze [20]. Podle pravidla deseti [21], čím později objevíme chybu ve výrobním procesu, tím nás bude stát řádově více. Tato závislost je dobře vyobrazena na obrázku 1.8. Pozorujeme, že otestování čipu a nalezení poruchy může ušetřit až tisícinásobně více než objevení chyby u zákazníka.

Dle [22] je testem číslicového obvodu množina dvojic. Prvním prvkem páru z množiny je vstup a druhým výstup. Tento pár pak můžeme nazvat krokem testu.

Defekty (*defects*), jsou poškození na fyzické úrovni, tedy například špatně zapojený konektor. Poruchy (*faults*) definujeme jako modelované defekty na úrovni logické [20]. Na obrázku 1.9 je znázorněna vazba mezi defekty a poruchami.

V této práci se budu zabývat testováním na úrovni logické, tedy testováním poruch.

[7] Převzato a upraveno z [15].
[8] Převzato a upraveno z [18].
[9] Převzato a upraveno z [20].
[10] Převzato a upraveno z [20].




Obrázek 1.8: Vizualizace pravidla deseti.[9]

## 1.4 Poruchy

Poruchy lze dělit na mnoho typů, mezi nimiž jsou:

- Trvalé poruchy
- Přechodné poruchy
- Občasné poruchy
- „Common-mode“ poruchy

### 1.4.1 Trvalé poruchy

Trvalé poruchy jsou nejčastěji používané pro testování. Jedná se o poruchy u kterých jsou změny permanentní. Dělí se na několik typů:

- Trvalá 0 nebo 1
- Zkrat
- Zpoždění

*Trvalá 0 nebo 1* znamená, že na vodiči je permanentně nastavená hodnota logické jedničky nebo nuly. Tento typ poruch pokrývá většinu případů, které testujeme. Existují metody, díky kterým je možné redukovat počet testovaných vodičů, např. dominancí nebo ekvivalencí poruch, viz [1].

*Zkrat* představuje propojení dvou nebo více vodičů ve chvíli, kdy propojené být nemají.

*Zpoždění* znamená, že doba šíření signálu trvá déle než je očekáváno. Může to být způsobeno např. větším odporem nebo kapacitou vodiče.

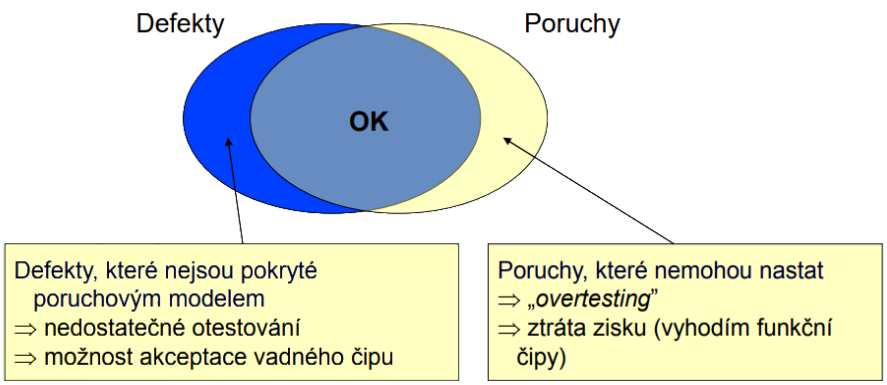
### 1.4.2 Přechodné poruchy

Tento typ poruch je nutné testovat pouze online, tedy za běhu systému. Je to způsobeno tím, že tyto poruchy se mohou objevit a následně zmizet. Jsou způsobeny vnějšími vlivy, jako například kosmickým zářením, výkyvy napětí nebo teploty.

V některých případech se může jednat i o zásah útočníka, takzvaný útok zaváděním chyb (Fault injection attack) [23].

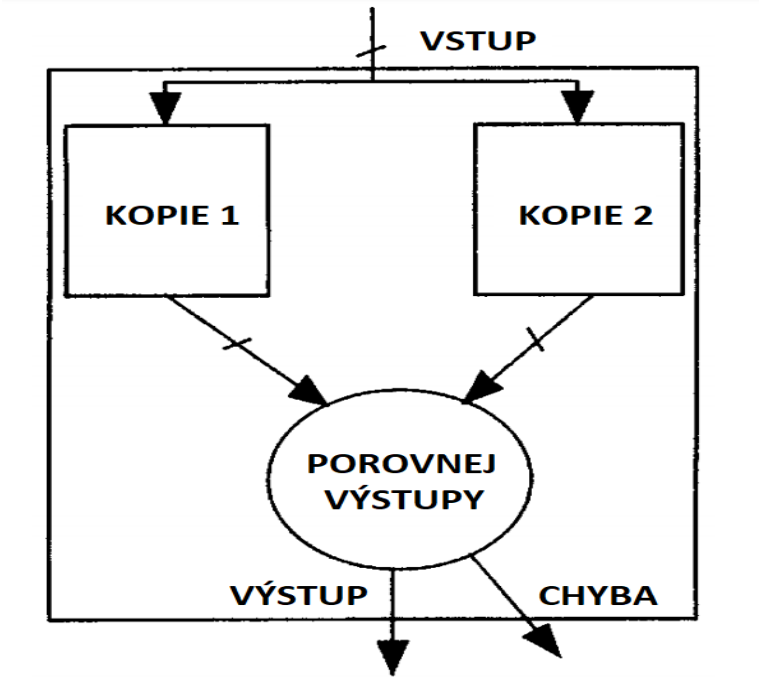
### 1.4.3 Common-mode poruchy

Tento typ poruch se projeví tak, že se ve dvou nezávislých obvodech objeví stejná porucha. Tyto poruchy mohou, ale nemusí, vzniknout ze stejného důvodu. Příkladem „common-mode“ poruchy je duplexní obvod, viz obrázek 1.10. Předpokládejme, že pravděpodobnost poruchy jedné kopie je $10^{-6}$. V takovém případě pravděpodobnost poruchy systému bude $10^{-12}$. Nicméně pokud pro „common-mode“ poruchu v tomto systému je pravděpodobnost $10^{-7}$, pak pravděpodobnost selhání celého systému není $10^{-12}$, pouze ale $10^{-7}$.




Obrázek 1.9: Pokrytí defektů poruchovým modelem.[10]

Toto znamená, že přidání duplexní jednotky nemusí zvýšit odolnost systému proti poruchám, jak bychom očekávali [24].




Obrázek 1.10: Ukázkový duplexní systém.[11]

## 1.5 Algoritmy generování testu

ATPG (**A**utomatic **T**est **P**atterns **G**eneration) jsou skupinou metod, které generují vstupy pro obvod takovým způsobem, že testovací aparatura podle výstupů může poznat zda, případně kde se nachází porucha v logickém obvodu.

Nalezení poruchy není nicméně zaručeno ze dvou důvodů:

- Porucha není detekovatelná
- Porucha je detekovatelná, ale ATPG jí nenajde

V případě, že porucha není detekovatelná, to může znamenat, že porucha samotná nedokáže změnit výstup testu. Takovým poruchám pak říkáme redundantní.

Problémem v případě, že porucha je detekovatelná, ale pouze nenalezená je fakt, že nalezení poruch je NPC-problém, a ATPG vzdá hledání než jí má šanci najít.

[11] Převzato a upraveno z [24].

## 1.6 Korelace dat

V obecném smyslu slova „korelace“ označuje míru asociace dvou proměnných. Ve statistice tento pojem říká, jestli mají dvě veličiny lineární vztah. Tento vztah můžeme spočítat pomocí korelačního koeficientu, který nabývá hodnot mezi -1 a 1. Hodnota 1 znamená přímou závislost mezi veličinami a hodnota -1 označuje nepřímou závislost.

Tuto hodnotu můžeme spočítat například pomocí pomocí Personova korelačního koeficientu.

### 1.6.1 Personův korelační koeficient

Personův koeficient spočteme vydělením kovariance množin jejími směrodatnými odchylkami. Dle [25, 26] vzorec 1.1 znázorňuje výpočet korelačního koeficientu mezi množinou $x$ a $y$. Vzorec 1.3 popisuje směrodatnou odchylku a vzorec 1.2 popisuje kovarianci. Vzorec 1.4 dále popisuje střední hodnotu množiny.

$$\rho_{x,y} = \frac{cov(X,Y)}{\sigma_x \sigma_y} \tag{1.1}$$

$$cov(X,Y) = E[(X - E[X])(Y - E[Y])] \tag{1.2}$$

$$\sigma = \sqrt{E((X - E(X))^2)} \tag{1.3}$$

$$E = \sum_I s_i p_i, \text{ kde } P[X = s_i] = p_i \text{ pro } i \in I \tag{1.4}$$

Kapitola 2

# Analýza dosavadních řešení

V této kapitole popíšu existující způsoby řešení porovnávání dvou logických obvodů. Jelikož jak víme z [17], lze popsat logické obvody grafem, zaměříme se na porovnávání dvou grafů.

Existuje celá řada algoritmů, které se zabývají porovnáváním dvou grafů. Mnoho z těchto algoritmů se z důvodu zvýšení efektivity zaměřuje na specifické typy grafů [14]. Příkladem mohou být algoritmy zaměřující se na planární grafy s lineární asymptotickou složitostí [27] (bohužel s vysokou konstantou) nebo algoritmy s kvadratickou složitostí zaměřující se na grafy s unikátními hodnotami jednotlivých vrcholů [28].

Níže popsané algoritmy jsem vybral pro ukázku grafové editační vzdálenosti (GED), jelikož se jedná o algoritmy, které byly základem pro mé řešení.

## 2.1 Grafová editační vzdálenost

První ze základních algoritmů výpočtu GED (dle [14]) na vstupu očekává dva neprázdné grafy $g_1 = (V_1, E_1, u_1, v_1)$ a $g_2 = (V_2, E_2, u_2, v_2)$, kde $V_1$ a $V_2$ jsou množiny uzlů, $E_1$ a $E_2$ jsou množiny hran grafů, $u_1$ a $u_2$ jsou množiny ohodnocení uzlů a $v_1$ a $v_2$ jsou množiny ohodnocení hran. Tedy součástí každého uzlu z $V_1$ nebo $V_2$ je jeho ohodnocení z $u_1$ nebo $u_2$. A obdobně ke každé hraně z $E_1$ nebo z $E_2$ patří ohodnocení z $v_1$ nebo z $v_2$. Výstupem tohoto algoritmu

je nejlevnější cesta (tj. editační vzdálenost) z $g_1$ do $g_2$.

**Algoritmus 1:** Grafová editační vzdálenost

```
Vstup: g1 = (V1, E1, u1, v1) a g2 = (V2, E2, u2, v2), kde
  V1 = {u1, ..., u|V1|} a V2 = {u1, ..., u|V2|}
Výstup: Grafová editační cesta s nejmenší cenou z g1 do g2
inicializuj OPEN jako prázdný set { }
pro každý uzel w ∈ V2, vlož substituci {u1 → w} do OPEN
vlož smazání uzlu {u1 → ϵ} do OPEN
while true do
    vyndej nejlevnější cestu v OPEN;
    if cesta je celá z g1 do g2 then
        cesta je řešením;
    else
        if délka cesty je menší než počet uzlů g2 then
            vytvoř kopie cesty, rozšiř každou mapováním dalšího uzlu
             z g1 na jeden nenamapovaný uzel g2;
            tyto kopie vlož do OPEN;
        else
            vlož do OPEN cestu rozšířenou o smazání všech
             zbývajících uzlů g1;
        end
    end
end
```

Jak je vidět z popisu algoritmu 1, tento algoritmus neřeší explicitně hrany grafu. S jejich editací se počítá v rámci úprav uzlů.

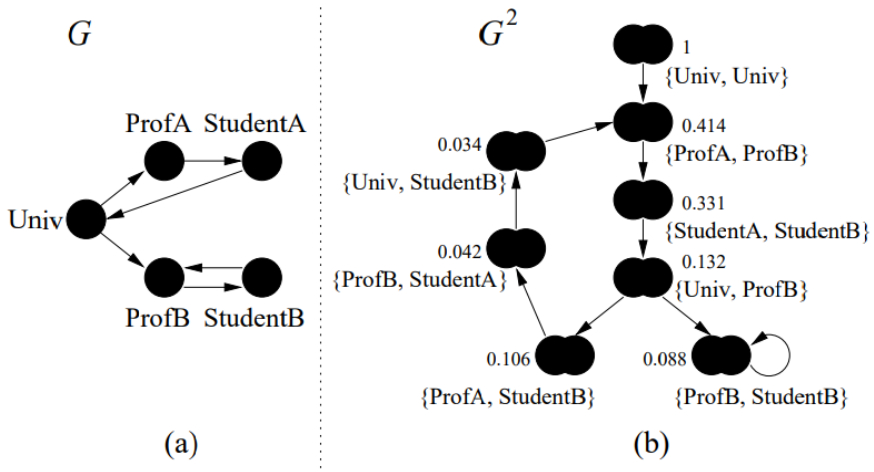
## 2.2 Přiřazovací problém a bipartitní graf

Algoritmus vytvořený v [14] pracuje na základě přiřazovacího problému na bipartitním grafu.

Za předpokladu, že každá operace má přiřazenou nějakou cenu, můžeme vytvořit matici, která znázorňuje cenu přechodu z každého uzlu jednoho grafu do každého uzlu z druhého. V takovém případě můžeme optimální transformaci z jednoho grafu do druhého najít pomocí toku v úplném bipartitním grafu [14].

Dle autorů je složitost tohoto algoritmu $O(n^3)$. Jak ale podotýkají, je tato složitost lepší než metoda hrubou silou se složitostí $O(n!)$.

Pro tento problém byl vytvořen algoritmus zvaný Maďarská metoda [29] v roce 1955, který je popsaný jako algoritmus přiřazující pro každého z $n$ lidí jednu z $n$ prací, tak aby jejich výkon byl co nejlepší.




Obrázek 2.1: Ukázka fungování algoritmu SimRank.[12]

## 2.3 SimRank

Glen Jeh a Jennifer Widom v [30] popsali algoritmus nazvaný SimRank pro grafy, jejichž úkolem je určit podobnost dvou objektů. Principem použitým v algoritmu je: „Podobnost dvou objektů je závislá na podobností objektů se kterými jsou v relaci." Neboli, pokud porovnáváme dva objekty, které mají podobné sousedy, pak tyto objekty budou mnohem více podobné než takové objekty u kterých sousedi jsou různí.

Tento algoritmus se neukázal jako použitelný pro můj problém. Je totiž zaměřený na porovnávání dvou objektů v rámci stejného grafu. Teoreticky by bylo možné oba porovnávané grafy propojit do jednoho, například propojením vstupních a výstupních uzlů. Nicméně i v takovém případě by bylo nutné po získání výsledných podobností mezi jednotlivými uzly nějakým způsobem vytvořit metriku, která by tyto data vzala a určila jak jsou grafy podobné.

Předpokládám, že by se takovou metriku dalo vytvořit a nějakým způsobem využít, nicméně vzhledem k tomu, že existují algoritmy, které toto řeší s menší komplexitou, rozhodl jsem se pro zkoumání jiných způsobů.

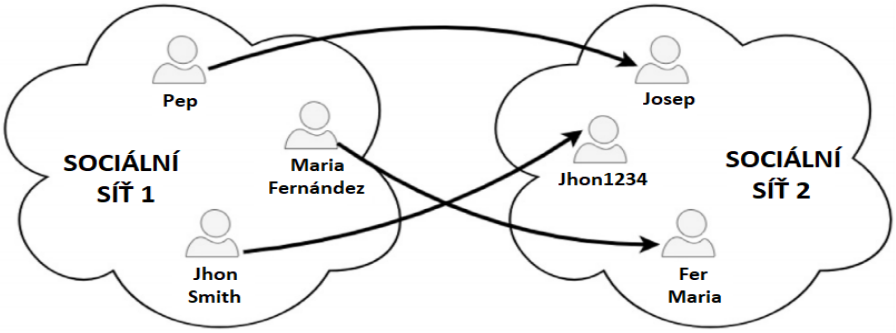
[12] Převzato a upraveno z [30].

## 2.4 Funkční ekvivalence

Vedoucí práce, doc. Ing. Petr Fišer, Ph.D., vypracoval metodu porovnávání dvou logických obvodů založenou na funkční ekvivalenci signálů.

Je založena na porovnávání každého signálu v prvním obvodu s každým signálem ve druhém obvodu a určování jejich funkční ekvivalence. Tato metoda bohužel má vysokou časovou složitost a není tudíž vhodná pro složitější obvody.

## 2.5 Porovnávání grafů s počátečním párováním uzlů




Obrázek 2.2: Příklad mapování mezi dvěma sociálníma sítěmi. [13]

P. Santacruz a F. Serratosa v [11] navrhují variantu GED algoritmu popsaného v [14]. V této variantě zjednodušují celý proces pomocí počáteční množiny dvojic o kterých víme, že je lze na sebe převést. Tato množina je využita jako počáteční seed a celý algoritmus se zjednodušuje na hledání podobnosti na úrovní hvězd, neboli jediného uzlu z každého grafu a jejich sousedů. Samozřejmě přesnost porovnávání grafů se tím zmenšuje, nicméně časová složitost je drasticky menší (algoritmus měl složitost $O(n^3)$) a dle [11] byla tato metoda první, která se dala použít na grafy o 5000 uzlech. Dle autorů je složitost rovná $O(s.\sqrt{d}.n)$, kde $s$ je cena přiřazování základní struktury na které pracuje algoritmus (v případě [11] se jedná o hvězdu), $d$ je množství hran každého z uzlů a $n$ celkové množství uzlů.

Na obrázku 2.2 můžeme vidět ukázku dvou sociálních sítí a tří mapování nazvanými Seed.

Algoritmus 2.1 na vstupu očekává dva grafy a počáteční mapování (řádek 1). Z každé dvojice uzlů lze vytvořit jejich hvězdy (uzel jako prostředek a jeho

[13] Převzato a upraveno z [11].

sousedí jako opozice). Tyto dvojice hvězd se vloží do fronty a spočítají cenu jaká je potřebná na přetvoření hvězdy prvního uzlu na hvězdu druhého (řádky 3-4). Hlavní smyčkou algoritmu je vybrání té dvojice hvězd s nejnižší cenou přetvoření (6,7), přidání všech hvězd sousedících (9-12) a vymazání z fronty jiných variant přetvoření vybraných hvězd (8).

Zdrojový kód 2.1: Pseudokód algoritmu pro určení diverzity dle [11].

```
1. load G1, G2, Seeds
2. Computed = Seeds, Pending = [], Matching = []
3. for seed1, seed2 in Seeds:
4.    Pending.append = MatchStar(seed1, seed2)
5. while Pending.size > 0:
6.    min_pair = min(Pending)
7.    Matching.append(min_pair)
8.    remove any pairs from Pending that has any of stars from
   min_pair
9.    for mapping in min_pair:
10.       if mapping not in Computed
11.           Computed.append(mapping)
12.           Pending.append(mapping + MatchStar(g1[mapping],
   g2[mapping])
13. for every node that was not mapped from g1:
14.    Computed.append(node, NULL)
15. for every node that was not mapped from g2:
16.    Computed.append(NULL, node)
```

Kapitola 3

# Návrh

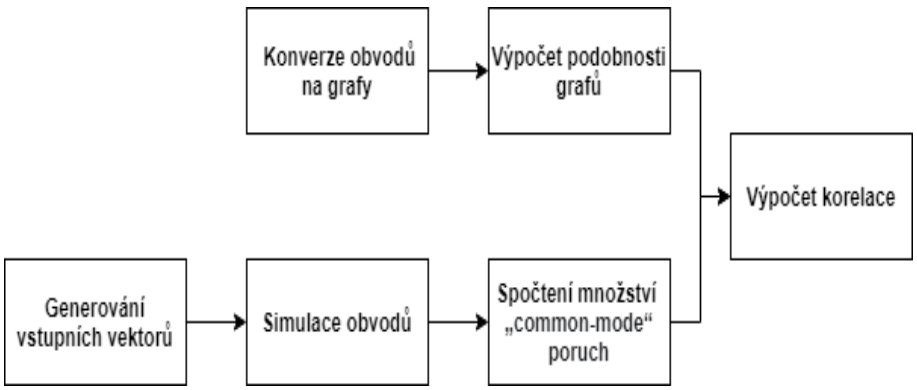
Návrhovou část můžeme rozdělit na několik podkategorií, kterými jsem se zabýval.

První z nich je výpočet diverzity logických obvodů pro zjištění závislosti „common-mode“ poruch na této diverzitě. Toto provedeme konvertováním obvodů na grafy dle 1.3.1.

Druhou podkategorií je odhadnutí množství „common-mode“ poruch (viz kapitola 1.4.3), které tyto grafy obsahují.

Poslední podkategorií je porovnání výstupů výpočtu diverzity a výpočtu množství „common-mode“ poruch. Mým očekáváním je, že čím větší diverzita dvou grafů, tím je množství „common-mode“ poruch menší, alternativně bude menší poměr „common-mode“ poruch ke všem odhaleným poruchám.

Na schématu 3.1 znázorňuji navrhovaný postup pro výpočet korelace podobnosti grafů a množství „common-mode“ poruch.




Obrázek 3.1: Schéma procesu zjištění korelace podobnosti obvodů a „common-mode“ poruch.

Od svého vedoucího práce jsem obdržel sadu 100 logických obvodů spolu s maticí diverzit mezi jednotlivými obvody. Tato matice byla vypočtena meto-

dou funkční ekvivalence signálů 2.4. Tyto grafy jsem měl k dispozici pro testování své implementace. Tyto obvody jsou funkčně ekvivalentní, neboli pro stejné vstupy poskytují stejné výstupy. Nicméně jejich struktura je odlišná. Všechny obvody jsou implementovány pomocí hradel AND a negací signálu. Každý obvod obsahuje 23 vstupů a 2 výstupy. Celkově tedy existuje přes osm miliónu možných vstupních vektorů.

## 3.1 Zjištění grafové diverzity

Při návrhu implementace zjištění diverzity grafů byla důležitá velikost grafů určených pro výpočty a tudíž i časová efektivita algoritmu. Vzhledem k tomu, že většina logických obvodů, které jsem počítal byly v rozsahu 500 až 1000 uzlů, rozhodl jsem se pro použití algoritmu „Belief propagation graph matching“ 2.5 dle [11]. Tento algoritmus byl dle autorů schopen porovnávat dva grafy o velikosti 5000 uzlů, tedy měl by být časově dostačující pro výpočet 100 grafů každý s každým o 500 až 1000 uzlech.

### 3.1.1 Logické obvody

Prvním krokem při výpočtu diverzity je konverze logických obvodů na grafy. Všechny obdržené obvody jsou složeny pouze z hradel typu AND. Z tohoto důvodu je konverze velmi jednoduchá, každé hradlo odpovídá jednomu uzlu v grafu. Jednotlivé hrany grafu budou znázorňovat propojení signálu mezi hradly. Na doporučení vedoucího práce jsem se rozhodl ignorovat negace signálu, jelikož jejich existence by neměla mít vliv na přesnost výpočtu.

### 3.1.2 Atributovaný graf

Problémem na který jsem narazil bylo, že v tomto algoritmu, a i v mnoha dalších [31], se počítá s atributovaným grafem, díky čemuž se při přiřazování uzlů berou v potaz hodnoty v uzlech a porovnávají se mezi sebou. Grafy, které jsem vytvořil z logických obvodů bych mohl také „atributovat“, nicméně dlouho otázkou bylo jak.

Jednou z možností bylo přiřadit každému uzlu hodnotu dle typu hradla, které reprezentuje. Je možné, že v obecném případě by toto bylo dobrou atributizací, nicméně v mém případě byly grafy složeny pouze z hradel typu AND. Tedy ohodnocení všech uzlů by bylo stejné a jejich přiřazování by bylo náhodné. Je to tedy varianta pro mně v důsledku stejná jako poslední, ale s nutností většího množství výpočtů a tedy jsem ji zavrhl pro mé specifické zadání už při návrhu.

Jinou možností nad kterou jsem přemýšlel, bylo přiřazení hodnoty na základě topologického uspořádání. Nicméně vyžadovalo by to výpočet topologických stromů pro všechny grafy a nepředpokládal jsem žádný významný vliv na přesnost výpočtu. Tuto možnost jsem tedy zavrhl už při návrhu.

Třetí možností je použít jako atributy indexy jednotlivých uzlů v matici sousednosti. Pseudokód této možnosti můžeme vidět v 3.1. Tento přístup také pomůže zajistit, že pokud dva grafy budou naprosto totožné, pak algoritmus označí jejich diverzitu jako nulovou.

Zdrojový kód 3.1: Pseudkód MatchStar

```
0. input: graph1, graph2; output: list of transformations
1. for each node in graph1:
2.     if node index in graph2:
3.         LoT.append(index in g1, index in g2)
4. for any node that is not paired from g1 and g2:
5.     make paires by pairing first with the first and so on
6.     if there is not enough nodes in both graphs, pair nodes with
    empty nodes
```

Poslední možností je použít náhodné přiřazování a to tak, že v rámci porovnávání hvězd přiřadíme k sobě jednotlivé uzly náhodně. U této možnosti se nicméně obávám, jestli výsledky nebudou příliš náhodné.

Po prozkoumání těchto možností jsem se rozhodl implementovat poslední dvě možnosti, tedy přiřazování uzlů podle jejich indexů a přiřazování uzlů náhodně.

## 3.2 Simulace obvodů

Druhou částí mé práce je zjistit množství „common-mode“ poruch v jednotlivých dvojicích obvodů. Od doc. Ing. Petr Fišera, Ph.D. jsem obdržel ATPG. Pomocí tohoto programu odsimuluji vstupní vektory a poruchy ve všech obvodech. Následně pomocí programu Matlab najdu množství „common-mode“ poruch mezi jednotlivými grafy na základě obdržených výstupů a odhalených chyb.

### 3.2.1 Vstupní vektory

Obvody, které jsem obdržel mají 23 vstupů. Znamená to tedy, že celkem existuje 8 388 608 různých vstupních vektorů. Je zřejmé, že pro takové množství vektorů by mohl výpočet trvat velmi dlouho a není v rozumném čase možné otestovat všechny.

Nicméně pro co nejpřesnější výsledek mého měření je nutné vygenerovat a otestovat velké množství binárních vektorů. Tento problém lze jednoduše vyřešit pomocí skriptu v programovacím jazyce Python. Samotná generace vektorů není nijak výpočetně náročná, ale budu náhodně generovat pouze omezené množství vzhledem k předpokládané výpočetní náročnosti ostatních částí výpočtů.

Pro odstraňování duplicit lze použít jednoduchý program *uniq* ve skriptovacím jazyce Bash.

### 3.2.2 Common-mode poruchy

Pomocí ATPG dokážu odsimulovat vstupní vektory pro jednotlivé obvody. Nicméně je nutné následně výstupní data zpracovat. Pro tento účel navrhuji použít program Matlab určený pro práci s velkým množstvím dat. Abych spočítal všechny odhalené „common-mode“ poruchy pro jednotlivé vektory, je nutné vzít odpovědi dvou obvodů pro jeden vektor a zjistit jestli daný vektor by pomohl odhalit nějakou poruchu u obou obvodů. Pokud ano, je nutné zjistit jestli daná porucha způsobuje stejný výstup na obou obvodech. V případě, že tomu tak je, jedná se o „common-mode“ poruchu a mohu inkrementovat množství těchto poruch, které nejsou detekovatelné mezi jednotlivými obvody.

Tímto způsobem musím prozkoumat všechny dvojice obvodů pro všechny odsimulované vstupní vektory.

### 3.2.3 MetaCentrum

Z důvodu předpokládané výpočetní náročností jsem požádal o přístup k Národní Gridové Infrastruktuře MetaCentrum spravované oddělením organizace CESNET.

Na této platformě je možné provádět velké množství dlouhodobých i krátkodobých výpočtů hlavně pro vědecké účely. Dle oficiálních stránek [32] je k systému připojeno více než 23 tisíc procesorů z celé České republiky. Do programu je zapojena řada výzkumných organizací mezi nimž je botanický ústav AV ČR nebo technická univerzita Liberec.

## 3.3 Korelace dat

Posledním výpočetním krokem v mé práci je porovnání množství „common-mode“ poruch nacházejících se v obvodech s vypočítanou diverzitou grafů jednotlivých obvodů.

Očekávám, že množství „common-mode“ poruch, nacházejících se ve dvou obvodech, bude tím menší, čím větší je diverzita, neboli čím míň jsou si tyto obvody podobné. Předpokládám tedy, že mezi výslednými daty obou výpočtů bude vysoká kladná korelace.

Tuto korelaci vypočítáme pomocí programu Matlab porovnáním matic s výstupními daty.

Při konečném návrhu jsem se rozhodl pro změnu diverzity na similaritu z důvodu lepší čitelnosti grafů. Toho docílím následovně: $similarita = 1 - diverzita$.

Tímto způsobem dostanu všechny grafy v kladné ose, ale výsledek se nezmění.

Kapitola 4

# Implementace

V této kapitole se zabývám způsobem implementace algoritmů pro výpočet podobnosti mezi jednotlivými grafy a skriptů pro výpočet „common-mode“ poruch. V poslední sekci řeším skripty pro výpočet korelace mezi oběma částmi.

## 4.1 Algoritmus výpočtu grafové podobnosti

Z důvodu jednoduchosti vytvoření skriptu a poměrně nízké časové náročnosti jsem se rozhodl pro vytvoření této části v jazyce Python. Kód tohoto programu se nachází na přiloženém médiu, zde bych chtěl pouze ukázat hlavní funkci MatchStar, přesněji dvě verze její implementace: přiřazování dle indexu, viz 4.1 a náhodné přiřazování, viz 4.2. Tato funkce na vstupu očekává dvě hvězdy, neboli dva binární vektory znázorňující pomocí jedniček sousedy daného uzlu.

### 4.1.1 Přiřazování uzlů dle indexu

Jak je vidět z obdržených vektorů dostaneme všechny indexy sousedů uzlů. Pomocí těchto indexů poté přiřazujeme jednotlivé uzly k sobě. V druhé části funkce přiřadíme postupně všechny uzly, jejichž index se nenachází ve druhém grafu. Případné uzly které přebývají, přidáme pokud jsou z grafu $g1$ nebo naopak odebereme, pokud jsou z grafu $g2$. Toto udělám tak, že uzly spáruji s hodnotou $-1$, která označuje prázdný uzel.

Zdrojový kód 4.1: Funkce MatchStar

```
def edit_dist(g1, g2):
    g1_indexes = np.nonzero(g1)[0]
    g2_indexes = np.nonzero(g2)[0]
    actions = []
    if g1_indexes[0] != 0
        for i, g1_ind in enumerate(g1_indexes):
            if g1_ind in g2_indexes:
                actions.append((g1_ind, g1_ind))
```

```
9.                g1_indexes = np.delete(g1_indexes,
    np.nonzero(g1_indexes == g1_ind)[0][0])
10.               g2_indexes = np.delete(g2_indexes,
    np.nonzero(g2_indexes == g1_ind)[0][0])
11.   g1_num_of_neighbours = len(g1_indexes)
12.   g2_num_of_neighbours = len(g2_indexes)
13.   i = 0
14.   if g2_num_of_neighbours > g1_num_of_neighbours:
15.       for i, g1_ind in enumerate(g1_indexes):
16.           actions.append((g1_ind, g2_indexes[i]))
17.       i += 1
18.       while i < len(g2_indexes):
19.           actions.append((-1, g2_indexes[i]))
20.           i += 1
21.   else:
22.       for i, g2_ind in enumerate(g2_indexes):
23.           actions.append((g1_indexes[i], g2_ind))
24.       i += 1
25.       while i < len(g1_indexes):
26.           actions.append((g1_indexes[i], -1))
27.           i += 1
28.   return actions
```

### 4.1.2 Náhodné přiřazování uzlů

V případě náhodného přiřazování uzlů ve hvězdě, nepřiřazuji uzly z prvního grafu k druhému na základě indexů. Místo toho randomizuji pořadí uzlů v první hvězdě a následně přiřazují tyto uzly popořadě k uzlům grafu druhého. V případě, že uzly přebývají, odebírám nebo přidávám tyto uzly spárováním uzlů s hodnotou $-1$, která označuje prázdný uzel.

Toho docílím tak, že v kódu zaměním řádky *5-10* za randomizování pořadí indexů, viz 4.2. Předpokládám ale, že tato varianta bude mnohem méně přesná.

Zdrojový kód 4.2: Funkce MatchStar pro randomizované přiřazování

```
1. def edit_dist(g1, g2):
2.     g1_indexes = np.nonzero(g1)[0]
3.     g2_indexes = np.nonzero(g2)[0]
4.     actions = []
5.     random.shuffle(g1_indexes)
6.     g1_num_of_neighbours = len(g1_indexes)
7.     g2_num_of_neighbours = len(g2_indexes)
8.    ...
```

## 4.2 Algoritmus simulace obvodů

Simulaci obvodů nebylo nutné implementovat, jelikož jsem spolu se zadáním obdržel od vedoucího práce i ATPG.

Jedinou nutností bylo vygenerovat náhodné vstupní vektory pro simulaci. Toto jsem provedl napsáním jednoduchého skriptu v jazyce Python.

Zdrojový kód 4.3: Skript pro generaci vstupních vektorů

```
import random

with open("vectors.pat", "w") as file:
    for i in range(10000):
        vec = random.getrandbits(23)
        vec = bin(vec)
        vec = vec[2:]
        while len((vec)) < 25:
            vec = '0' + vec
        file.write((vec)[2:])
        file.write("\n")
```

## 4.3 Algoritmus výpočtu common-mode poruch

Implementace výpočtu množství „common-mode“ poruch byla provedena v programu Matlab. Zdrojové kódy jsou k nalezení na přiloženém médiu. Hlavní smyčka programu je paralelizovaná tak, že jednotlivé porovnávání souboru tvoří jednu úlohu. Paralelizace je docíleno pomocí doplňku *Parallel Computing Toolbox* v Matlabu.

V ukázce kódu je vidět smyčku prováděnou pro každý vstupní vektor dvou obvodů. Tyto dva cykly procházejí skrz všechny nalezené poruchy v obou obvodech a porovnávají výstupy obvodů při poruchách. Provádím to, jelikož „common-mode“ poruchy mají na výstupů obou obvodů stejné hodnoty.

Jedná se o jednoduché porovnávání dvou výstupů obvodů pro daný vstup a všechny poruchy v obvodech.

Všechny matice jsou přiložené na fyzickém úložišti, dodaném s touto diplomovou prací.

Zdrojový kód 4.4: Výpočet common-mode poruch

```
for index1 = g1Indexes
   for index2 = g2Indexes
       different = different + 1;
      if g1Outputs{index1} == g2Outputs{index2}
          index1 + "; " + index2 + newline + g1Outputs(index1) + "; "
              + g2Outputs(index2);
          matrixOut(index1,index2) = matrixOut(index1,index2) + 1;
          same = same + 1;
```

```
    end
  end
end
```

## 4.4 Algoritmus výpočtu korelačního koeficientu

Výpočet korelačního koeficientu je opětovně proveden v programu Matlab. Jedná se o jednoduché porovnání dvou matic. První maticí je matice podobnosti jednotlivých grafů a druhou maticí je počet „common-mode“ poruch mezi jednotlivými obvody. Korelaci počítám pro každý řádek matic zvlášť, neboli pro každý obvod. Využil jsem funkce *corrcoef*, která používá Pearsonův korelační koeficient. Implementace celého výpočtu je k nalezení na přiloženém médiu.

Ukázka hlavní smyčky pro výpočet korelačního koeficientu se nachází v 4.5. Rozhodl jsme se pro iteraci skrz obě matice, podobnosti a množství poruch, ručně, jelikož nemá smysl počítat korelaci mezi podobností prvního obvodů s množstvím poruch v obvodu posledním.

Zdrojový kód 4.5: Výpočet korelačního koeficientu pro obvody

```
for i=1:100
    tmp = corrcoef(totalCount(i,:),my(i,:));
    corrTable(i) = tmp(1,2);
end
```

Kapitola 5

# Realizace

Implementace se skládala ze čtyř částí a pro jejich výpočet jsem musel použít kromě svého osobního počítače i cluster MetaCentrum. Jedná se o výpočet podobnosti grafů znázorňujících obvody, simulace jednotlivých obvodů, výpočet „common-mode“ poruch na základě simulace a následný výpočet korelace ze získaných výsledků.

## 5.1 Měření podobnosti grafů

Vzhledem k tomu, že výpočty podobnosti stačilo udělat pouze jednou pro každou ze dvou metod přiřazování uzlů a tento algoritmus nemá vysokou složitost viz [11], dovolil jsem si tento algoritmus pustit na svém přístroji a ten zvládl spočítat vše v rozumném čase.

## 5.2 Generování vstupních vektorů

Pro generování vstupních vektorů jsem naprogramoval jednoduchý skript v Pythonu a z toho důvodu jsem ho spustil na vlastním stroji. Doba generování 10000 vektorů trvala méně než 1 sekundu.

## 5.3 Měření simulace obvodů

Tento výpočet jsem se také pokusil udělat na svém osobním počítači, nicméně už spočítání 500 vstupních vektorů trvalo přibližně pět hodin. Nicméně 500 vektorů je velmi malá část vstupů, kterých je přes osm miliónů. Proto jsem se rozhodl pro registraci v MetaCentru, které poskytuje výpočetní prostředky pro akademický výzkum. Mohl jsem proto pustit výpočet paralelně, a to pro každý obvod jeden procesor. Toto mi dovolilo odsimulovat deset tisíc vektorů na všech obvodech za pouhé dvě hodiny. Nicméně i to je pouze zlomek všech

možných vstupů, který ale dovoluje zjistit, kterým směrem bude výsledek přibližně ubírat.

Skript 5.1 byl vytvořen pro zadání jedné úlohy do čekací fronty pro každý obvod. Velmi důležitým parametrem je *-v param=$I*. Tento parametr předá do skriptu *job.sh* číslo obvodu, který bude zpracovávat. Díky tomu, skript *job.sh* se dozví, který obvod je nutné překopírovat na výpočetní uzel. Toho je docíleno čtením v skriptu *job.sh* proměnné *param*.

Zdrojový kód 5.1: Spouštěcí skript pro simulaci obvodů

```
#!/bin/bash

for I in {0..9}
do
    qsub -l select=1:ncpus=2:mem=1gb:scratch_local=1gb:os=debian10 -l
        walltime=2:00:00 -v param=$I -N ATPG$I job.sh
done

for I in {10..99}
do
    qsub -l select=1:ncpus=2:mem=1gb:scratch_local=1gb:os=debian10 -l
        walltime=2:00:00 -v param=$I -N ATPG$I job.sh
done
```

V kódu 5.2 je ukázaný způsob volání ATPG ve skriptu *job.sh*. Důležitým argumentem je argument *-pattype 2*, který způsobí vypsání odpovědí pro všechny poruchy v obvodu pro všechny vstupní vektory. Samozřejmě před provedením samotného výpočtu je nutné přenést všechna potřebná data na výpočetní uzel a po skončení výpočtu vrátit výsledky zpět.

Zdrojový kód 5.2: Volání ATPG ze skriptu *job.sh*

```
./atpg -sim -pr vectors.pat -pw ext-pat/cordic-$param-ext.pat
    -pattype 2 blif/cordic-$param.blif
```

## 5.4 Měření common-mode poruch

Také tento výpočet vzhledem k množství matic nutných k výpočtu probíhal paralelně na několika procesorech v MetaCentru. Vzhledem k tomu, že výpočet probíhal v programu Matlab, bylo nutné použití doplňku *Parallel Computing Toolbox*, který dovoluje používat paralelní for cykly. Díky tomu výpočet této části pro deset tisíc vstupů, trval pouze patnáct hodin, během toho času výpočet použil skoro 161 hodin procesorového času. Všechny použité zdroje jsou ukázany v tabulce 5.1

Ve výseku ze skriptu 5.3 jsou vidět dvě důležité části skriptu pro spouštění výpočtu v programu Matlab na MetaCentrum. Jak ukazuje první řádka je

Tabulka 5.1: Prostředky použité pro výpočet „common-mode“ poruch

| Zdroje použité pro výpočet | | |
|---|---|---|
| RAM | 64% | 29GB / 48GB |
| CPU | 74% | 160:44:07 / (14*15:31:51) |
| čas běhu | 65% | 15:31:51 / 24:00:00 |

pro správnou funkcionalitu nutné do prostředí přidat modul Matlab. Zároveň je nutné oznámit programu, že jeho spuštění je vyžadováno bez grafického prostředí. Toho docílíme předáním tří argumentů *-nodisplay -nodesktop -nosplash* jak je vídět na poslední řádce. Parametrem *-r "atpgMatrix"* vybereme, která funkce se má spustit.

Zdrojový kód 5.3: Výsek ze spouštěcího skriptu pro výpočet množství „common-mode poruch“

```
module add matlab
...
matlab -nodisplay -nodesktop -nosplash -r "atpgMatrix"
```

## 5.5 Měření korelace dat

Díky předchozím výpočtům byl výpočet korelace množství „common-mode“ poruch a similarity obvodů, jednoduchým korelováním matic o velikosti deset tisíc prvků a proto na mém osobním stroji trval 2 sekundy.

Kapitola 6

# Analýza výsledků

Tato kapitola se věnuje analýze výsledků. Celkové množství simulovaných vektorů bylo deset tisíc pro sto obvodů.

Tyto obvody jsem obdržel od vedoucího této práce. Všechny obvody jsou vzájemně funkčně ekvivalentní (viz 2.4), ale mají rozdílné struktury. Obvody mají přibližně 1000 až 1600 poruch. Tyto obvody čísluji od 1 do 100 podle jejich pořadového čísla.

## 6.1 Common-mode poruchy

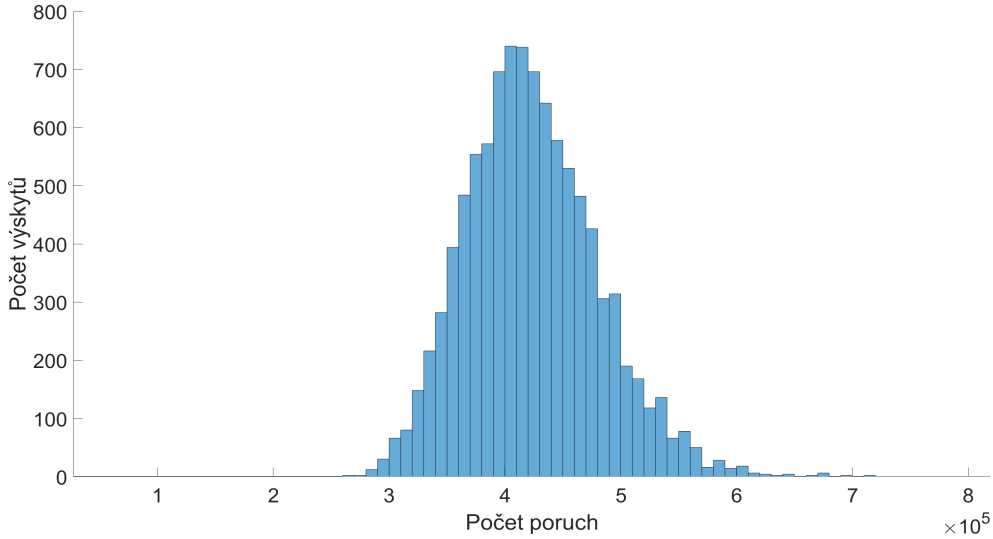
Graf 6.2 znázorňuje množství „common-mode“ poruch. Největší množství poruch je na diagonále, neboli při porovnávání stejných obvodů, znázorněno bílou barvou. Nicméně je zřejmé, že množství „common-mode“ poruch mezi dvěma stejnými grafy bude řádově větší než mezi grafy odlišnými. Proto je na diagonále (mezi stejnými obvody) množství „common-mode“ poruch řádově v milionech s největším množstvím na úrovni 3,1 miliónu a nejmenším množstvím „common-mode“ poruch je 1,4 milionu.

Tyto data potvrzují, to co jsme věděli (viz 1.4.3) a tedy, že použití duplexu, kde oba obvody jsou kopie stejného obvodu, rozhodně nechrání před „common-mode“ poruchami.

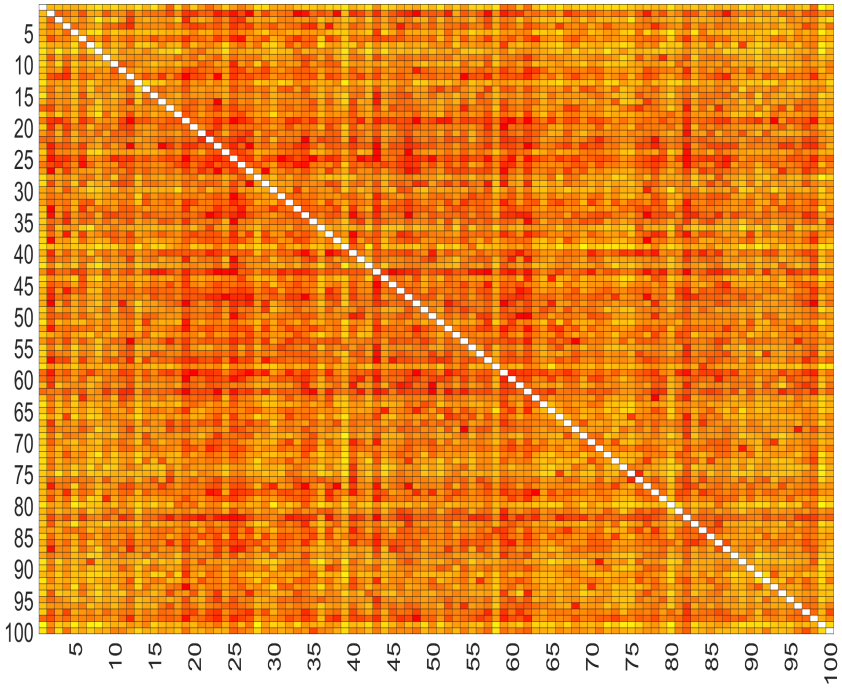
Pokud vynecháme diagonálu, zbylé hodnoty „common-mode“ poruch se nacházejí mezi 260 tisíci a 720 tisíci, jak je vidět na histogramu 6.1. Pro deset tisíc vstupních vektorů je tedy možné průměrně neodhalit pomocí duplexu 26 až 72 „common-mode“ poruch pro každý vstup.

## 6.2 Podobnost grafů

Podobnost grafů jsem počítal dvěma způsoby a to pomocí přiřazování ve vnitřních strukturách podle indexu uzlu nebo náhodně. Pro porovnání jsem vytvořil teplotní mapu z dat od vedoucího práce 6.3. Je na ní vidět, že několik




Obrázek 6.1: Histogram „common-mode“ poruch mezi obvody.




Obrázek 6.2: Celkový počet „common-mode“ poruch mezi jednotlivými obvody.

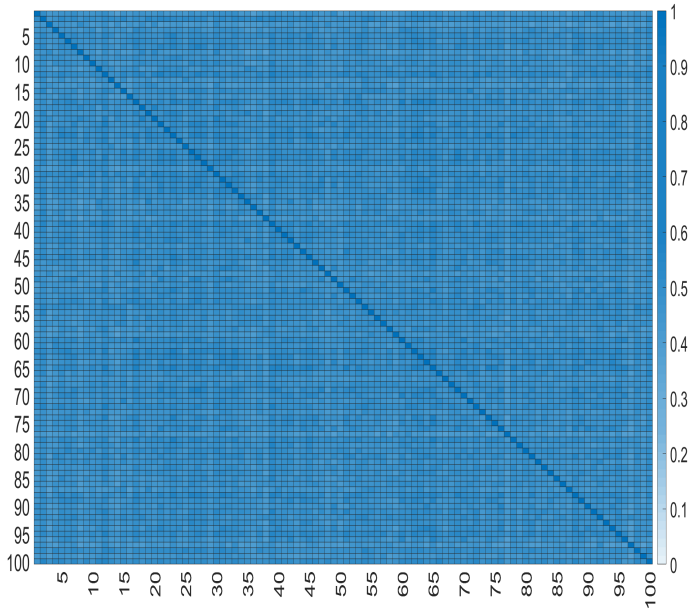
grafů je rozhodně odlišných od zbytku, nicméně střední hodnota podobnosti

je přibližně 0,48.

### 6.2.1 Náhodné přiřazování uzlů

Výsledky podobnosti pro náhodné přiřazování dle předpokladu chybně ukazují na nízkou podobnost. Zřetelně to můžeme pozorovat na diagonále, která zobrazuje porovnání dvou stejných grafů, viz obrázek 6.4.

Náhodné přiřazování označí stejné grafy ve většině případů za podobnější než dva různé. Nicméně není to pravidlem. Velmi dobře tuto skutečnost odráží střední hodnota podobnosti na úrovni 0,14.




Obrázek 6.3: Podobnost jednotlivých obvodů dle vedoucího práce

### 6.2.2 Přiřazování uzlů dle indexu

U metody přiřazování uzlů dle indexu je vidět, že v případě obdržení stejného grafu je za stejné označí.

Výsledky aplikace metody přiřazování uzlů dle indexu ukazují na vyšší podobnost mezi porovnávanými grafy. Z diagonály na obrázku 6.5, která znázorňuje porovnání grafu se sebou samým, je vidět, že v případě dvou stejných grafů je metoda správně označí za stejné.

Na obrázku 6.5 je také vidět, že některé množiny obvodů jsou si více nebo méně podobné oproti ostatním obvodům (např. grafy č. 45 až 50).

Průměrná hodnota podobnosti je 0,32. Není tedy tak nízká jako v případě náhodného přiřazování, ale ani podobná výsledkům vedoucího práce. Nicméně je důležité si uvědomit, že se jedná o zjednodušenou aproximační metodu.

## 6.3 Korelace dat

Posledním krokem je porovnat výsledky vypočtené podobnosti grafů a množství „common-mode“ poruch. To je provedeno korelací jednotlivých řádků matic, tedy získané výsledky jsou pro jednotlivé obvody. Na ose $x$ se nachází číslo obvodu a na ose $y$ je hodnota korelačního koeficientu.

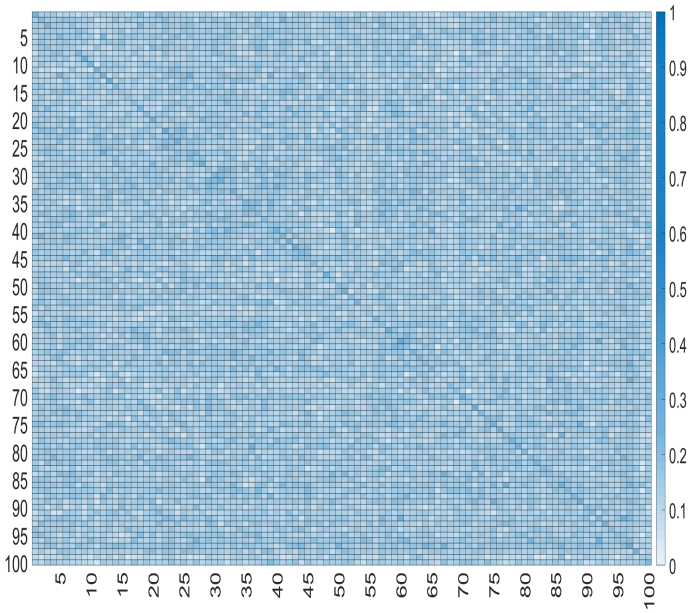
Jako referenční výsledek ukazuji data z mé simulace obvodů korelovaná s maticí podobností vedoucího práce. Výsledek ilustruje graf 6.6. Průměrný korelační koeficient pro všechny korelace spočítáme pomocí vzorce 6.1 dle [33]:

$$s_i = \frac{1}{n} \sum_{j=1}^{n} M_j, \tag{6.1}$$

kde $n$ je počet korelačních koeficientů a $corr_j$ značí $j$-tý korelační koeficient Průměrný korelační koeficient je v tomto případě 0,84.

### 6.3.1 Náhodné přiřazování uzlů

Na grafu 6.7 je vidět, že korelační koeficient při náhodném přiřazování uzlů ve hvězdách je velmi malý a pro každý obvod naprosto někde jinde. Zároveň dvě různá měření mohou dostat výsledek naprosto odlišný. Lze tedy usuzovat, že tento způsob není dostatečně přesný. Průměrný korelační koeficient má hodnotu 0,23. Toto naznačuje, že nějakou závislost mezi daty najdeme, ale není nijak významná.




Obrázek 6.4: Podobnost jednotlivých obvodů při náhodném párování.

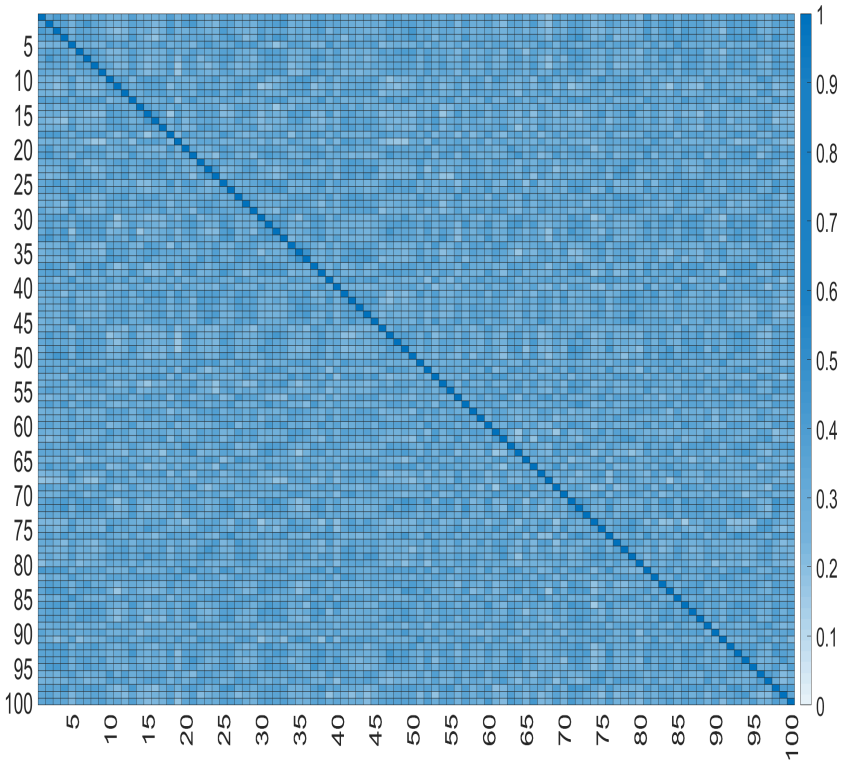
### 6.3.2 Přiřazování uzlů dle indexu

Z grafu korelačních koeficientů 6.8 je zřejmé, že mezi podobností grafu a počtem „common-mode“ poruch je poměrně vysoká závislost.

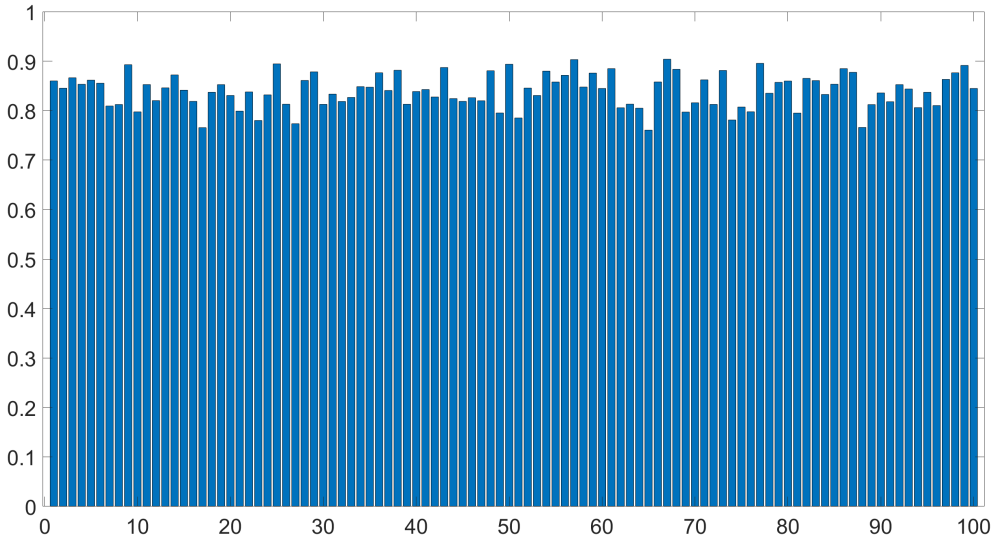
Průměr korelačního koeficientu pro tuto metodu a grafy je 0,77. Tato metoda tedy dle obdržených výsledků funguje s poměrně dobrou přesností.

### 6.3.3 Relativní počet „common-mode“ poruch

Vzhledem k faktu, že všechny výše uvedené grafy pracují s absolutními hodnotami, rozhodl jsem se také zjistit, jestli existuje závislost v případě, že budu korelovat diverzitu oproti poměru „common-mode“ poruch vůči všem nalezeným poruchám na daných obvodech. Výsledné korelační koeficienty je možné vidět na obrázku 6.9. Je vidět, že i když existuje přímá závislost, není příliš silná. Průměrný korelační koeficient v tomto případě je 0,4.




Obrázek 6.5: Podobnost jednotlivých obvodů při metodě párování pomocí indexů.




Obrázek 6.6: Korelace podobností obvodů dle vedoucího práce a simulovaných „common-mode“ poruch.

## 6.4 Porovnání výsledků

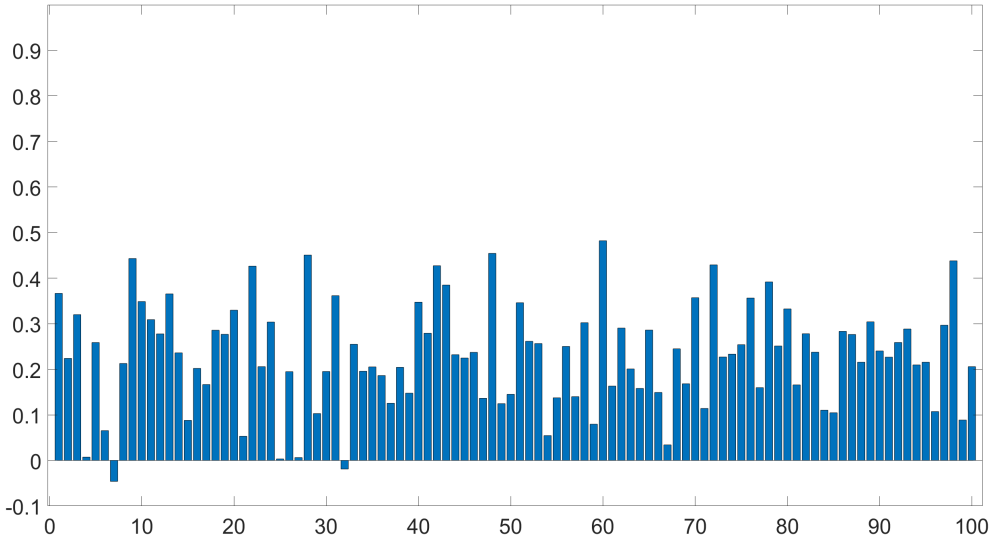
V tabulce 6.1 se nacházejí všechny průměrné korelační koeficienty pro jednotlivé varianty řešení.

Tabulka 6.1: Výsledné korelační koeficienty.

| Výpočet podobnosti obvodů | Průměrný korelační koeficient |
|---|---|
| Náhodné přiřazování | 0,2300 |
| Přiřazování dle indexu | 0,7656 |
| Dle vedoucího práce | 0,8400 |
| Přiřazování dle indexu s relativním množstvím poruch | 0,3972 |

### 6.4.1 Náhodné přiřazování uzlů

Ze získaných výsledků je jednoznačně vidět, že přístup pomocí náhodného přiřazování je nejméně dokonalý. Předpokládám, že tento systém by byl mnohém přesnější, jestliže by byla použitá heuristika a tedy změnila by se tato metoda na pseudonáhodnou.




Obrázek 6.7: Korelace podobností obvodů při náhodném přiřazování a množství „common-mode“ poruch.

### 6.4.2 Přiřazování uzlů dle indexu

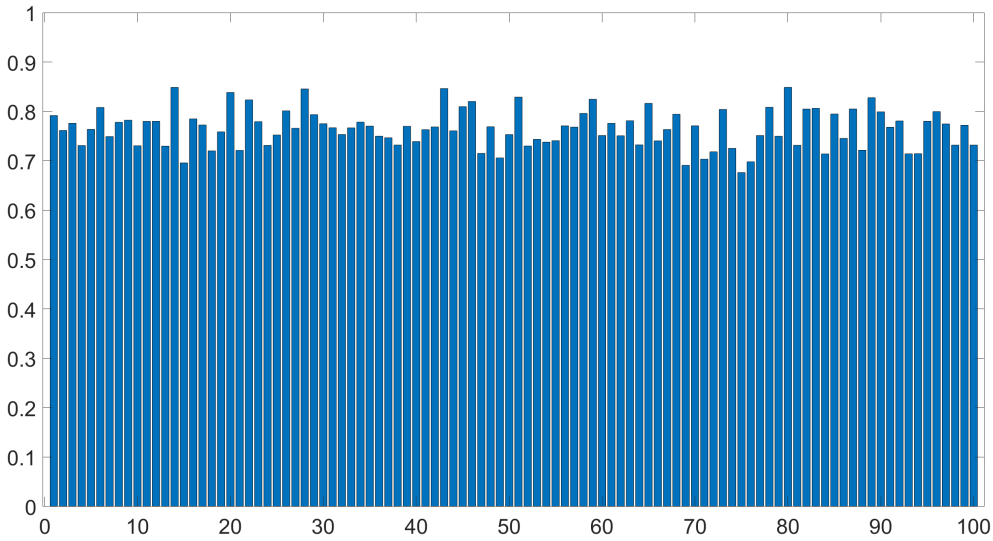
Výsledné korelační koeficienty mého přiřazování založeného na použití indexu vypadají velmi dobře. Při porovnání těchto výsledků z výsledky vedoucího práce dostaneme graf 6.10. Je z něj dobře vidět, že výsledky vedoucího práce 6.6 přisuzují obvodům, až na pár výjimek, vyšší korelaci než mé řešení 6.8. Je to pravděpodobně způsobeno přesnějšímu způsobu výpočtu podobnosti jednotlivých obvodů.

Problémem, který může nastat u mé metody je počítání podobnosti dvou grafů, které jsou stejné, ale mají jiné indexy jednotlivých uzlů.

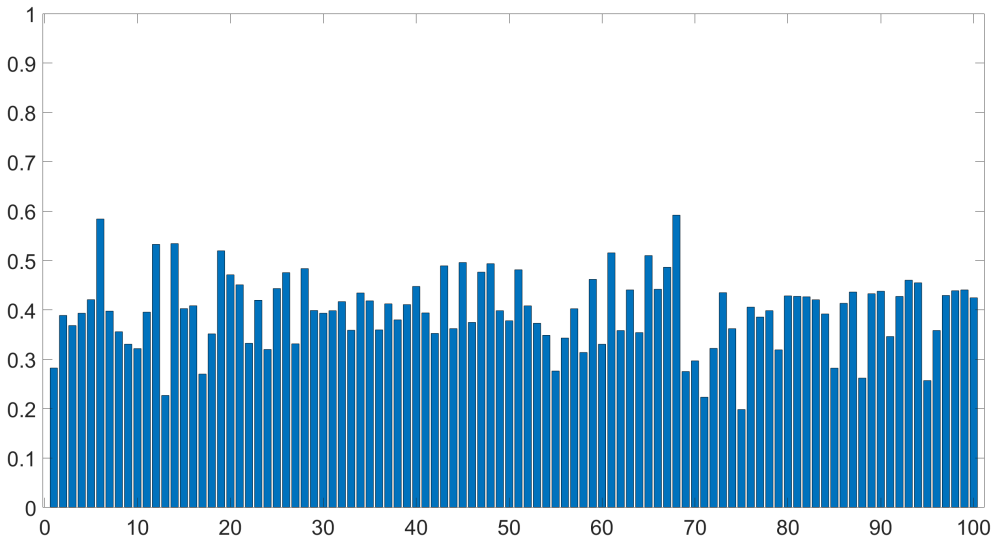
### 6.4.3 Relativní počet „common-mode“ poruch

Při této korelaci 6.9 jsem předpokládal lepší výsledky. Korelační koeficienty na úrovni 0,4 se dají považovat za střední závislost, nicméně koeficient u mnoha obvodů je pod úrovní 0,3.

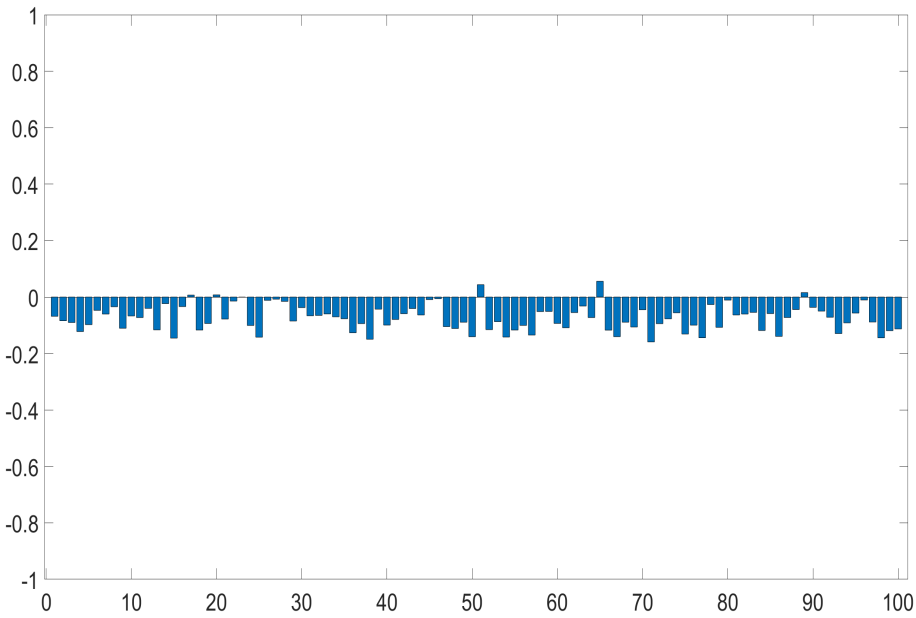
Rozdíl v hodnotě korelačních koeficientů je vidět na grafu 6.11. Můžeme pozorovat, že řešení s množstvím „common-mode“ poruch koreluje mnohem silněji.



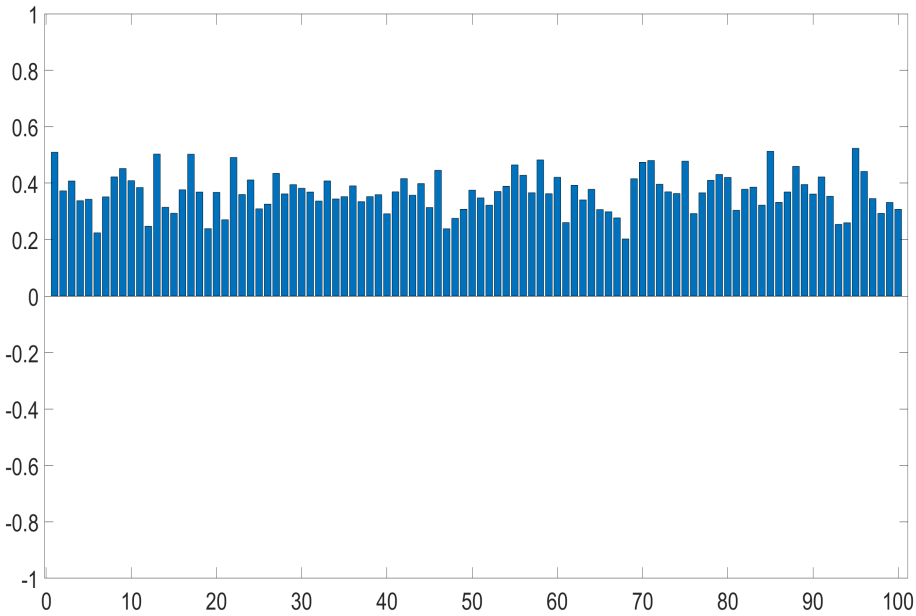

Obrázek 6.8: Korelace podobností obvodů při přiřazování pomocí indexů a množství „common-mode“ poruch.




Obrázek 6.9: Korelace podobností obvodů a poměru „common-mode“ poruch v obvodu.




Obrázek 6.10: Rozdíl korelačních koeficientu mezi řešením přiřazování pomocí indexu a řešením vedoucího práce.




Obrázek 6.11: Rozdíl korelačních koeficientu mezi množstvím „common-mode“ poruch a jejich poměrem ke všem nalezeným poruchám.

# Závěr

V této práci jsem prozkoumal dosavadní řešení výpočtu podobnosti dvou obvodů, nicméně zaměřil jsem se hlavně na grafové algoritmy. Implementoval jsem variantu algoritmu „Belief propagation graph matching“, jelikož se jednalo o algoritmus s nízkou časovou složitostí, což zlepšilo škálovatelnost. Vypočetl jsem podobnost obdržených obvodů a porovnal se zjištěným množstvím „common-mode“ poruch. Výsledek dle předpokladu ukázal, že existuje vysoká závislost mezi podobností dvou obvodů a množstvím nalezených „common-mode“ poruch.

V první kapitole práce jsem zadefinoval základní použité pojmy, jako například grafovou editační vzdálenost (GED).

Ve druhé kapitole jsem prozkoumal několik existujících řešení porovnávání dvou grafů, mezi nimiž je SimRank nebo funkční ekvivalence.

Třetí kapitola se zabývá návrhem řešení. V této kapitole prezentuji vlastní variantu počítání podobnosti grafů, založenou na variantě algoritmu GED v [11]. Dále navrhuji také způsob vypočtení množství „common-mode“ poruch mezi jednotlivými obvody a výslednou korelaci získaných dat.

Všechny implementace popisuji v kapitole číslo čtyři. Implementaci výpočtu podobnosti všech grafů a generaci vstupních vektorů jsem se rozhodl naprogramovat z důvodu jednoduchosti v jazyce Python. Pro implementaci vypočtení množství „common-mode“ poruch a finální korelaci získaných dat jsem se použil program Matlab, který je optimální pro práci s velkým množstvím dat.

Kapitola pátá popisuje běh výpočtů, použité prostředky a skripty potřebné pro běh programů ve výpočetním centru MetaCentrum. Díky použití MetaCentra jsem zvládl odsimulovat obvody pro deset tisíc vstupních vektorů v poměrně nízkém čase. Na této platformě jsem také z výstupů obvodů spočítal množství „common-mode“ poruch mezi jednotlivými obvody.

Poslední kapitola se zabývá analýzou získaných výsledků. Navržený algoritmus se ukázal jako funkční a jeho výsledky souhlasily s referenčními výsledky vedoucího práce.

# Literatura


[1] Novák, O.; Gramatová, E.; Raimund, U.: *Handbook of testing electronic systems.* Czech Technical University Publishing House, 2005, ISBN 80-01-03318-X.

[2] Taher, F. N.; Joslin, M.; Balachandran, A.; aj.: Common-Mode Failure Mitigation: Increasing Diversity through High-Level Synthesis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, s. 1563–1566, doi:10.23919/DATE.2019.8714816.

[3] Samanta, P.: Introduction to Graph Theory. 03 2011, doi:10.13140/RG.2.2.25721.88166.

[4] Bollobas, B.: *Modern Graph Theory.* Springer Science & Business Media, 2013-12-01, ISBN 978-1-4612-0619-4.

[5] Vrba, A.: *Grafy.* Státní pedagogické nakladatelství, n. p., v Praze, první vydání, 1989.

[6] Bondy, J.: Large cycles in graphs. *Discrete Mathematics*, ročník 1, č. 2, 1971: s. 121 – 132, ISSN 0012-365X, doi:https://doi.org/10.1016/0012-365X(71)90019-7. Dostupné z: `http://www.sciencedirect.com/science/article/pii/0012365X71900197`

[7] Team, K.: Guide: What is Directed Acyclic Graph? [online], 2019, [cit. 2020-12-28]. Dostupné z: `https://medium.com/kriptapp/guide-what-is-directed-acyclic-graph-364c04662609`

[8] bipartite graph. doi:10.1093/oi/authority.20110803095507587. Dostupné z: `https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095507587`

[9] Serratosa, F.: Fast computation of Bipartite graph matching. *Pattern Recognition Letters*, ročník 45, 2014: s. 244 – 250, ISSN 0167-8655,

doi:https://doi.org/10.1016/j.patrec.2014.04.015. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167865514001330

[10] Gao, X.; Xiao, B.; Tao, D.; aj.: A survey of graph edit distance. *Pattern Anal. Appl.*, ročník 13, 02 2010: s. 113–129, doi:10.1007/s10044-008-0141-y.

[11] Santacruz, P.; Serratosa, F.: Error-tolerant graph matching in linear computational cost using an initial small partial matching. *Pattern Recognition Letters*, ročník 134, 2020: s. 10 – 19, ISSN 0167-8655, doi:https://doi.org/10.1016/j.patrec.2018.04.003, applications of Graph-based Techniques to Pattern Recognition. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167865518301235

[12] Kohonen, T.: Self organizing maps. *Springer*, 1995.

[13] Dijkstra, E.: A note on two problems in connexion with graphs. *Numer Math*, 1959: s. 269-271.

[14] Riesen, K.; Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, ročník 27, č. 7, 2009: s. 950 – 959, ISSN 0262-8856, doi:https://doi.org/10.1016/j.imavis.2008.04.004, 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007). Dostupné z: http://www.sciencedirect.com/science/article/pii/S026288560800084X

[15] Roy, S.; Bhunia, C.: On Synthesis of Combinational Logic Circuits. *International Journal of Computer Applications*, ročník 127, 10 2015: s. 21–26, doi:10.5120/ijca2015906311.

[16] Kubátová, H.: Struktura a architektura počítačů: Kombinační obvody, logická syntéza. [online], 2020, [cit. 2020-12-28]. Dostupné z: https://courses.fit.cvut.cz/BI-SAP/media/lectures/02/sap-3-seq.pdf

[17] Entrena, L.; Cheng, K. .: Sequential logic optimization by redundancy addition and removal. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, s. 310–315, doi:10.1109/ICCAD.1993.580074.

[18] Crowe, J.; Hayes-Gill, B.: 5 - Asynchronous sequential logic. In *Introduction to Digital Electronics*, editace J. Crowe; B. Hayes-Gill, Oxford: Newnes, 1998, ISBN 978-0-340-64570-3, s. 125 – 149, doi:https://doi.org/10.1016/B978-034064570-3/50007-1. Dostupné z: http://www.sciencedirect.com/science/article/pii/B9780340645703500071

[19] Grout, I.: CHAPTER 5 - Introduction to Digital Logic Design. In *Digital Systems Design with FPGAs and CPLDs*, editace I. Grout, Burlington: Newnes, 2008, ISBN 978-0-7506-8397-5, s. 217 – 331, doi:https://doi.org/10.1016/B978-0-7506-8397-5.00005-2. Dostupné z: `http://www.sciencedirect.com/science/article/pii/B9780750683975000052`

[20] Fišer, P.: Úvod Terminologie, typy defektů, poruch. [online], 2019, [cit. 2020-12-28]. Dostupné z: `https://moodle-vyuka.cvut.cz/pluginfile.php/168072/mod_page/content/7/Lecture_1-intro.pdf`

[21] Wang, L.-T.; Wu, C.-W.; Wen, X.: *VLSI Test Principles and Architectures: Design for Testability.* Elsevier, 2006, ISBN 978-0-08-047479-3.

[22] Hlavička, J.: *Diagnostika a spolehlivost.* Praha: ČVUT, 1998, ISBN 80-01-01846-6.

[23] Rothbart, K.; Neffe, U.; Steger, C.; aj.: High level fault injection for attack simulation in smart cards. In *13th Asian Test Symposium*, 2004, s. 118–121, doi:10.1109/ATS.2004.48.

[24] Mitra, S.; Saxena, N. R.; McCluskey, E. J.: Common-mode failures in redundant VLSI systems: a survey. *IEEE Transactions on Reliability*, ročník 49, č. 3, 2000: s. 285–295, doi:10.1109/24.914545.

[25] HENDL, J.: *Přehled statistických metod zpracování dat: analýza a metaanalýza dat.* Portál, 2006, ISBN ISBN 80-7367-123-9. Dostupné z: `https://dnnt.mzk.cz/uuid/uuid:59e57080-9cc4-11e5-8b2a-5ef3fc9bb22f`

[26] Hrabák, P.; Vašata, D.: MI-SPI a MI-VSM – Statistika pro informatiku a Vybrané statistické metody Přednáška 3: Náhodné vektory. [online], 2020, [cit. 2021-01-20]. Dostupné z: `https://courses.fit.cvut.cz/MI-SPI/lectures/files/mi-spi-lec-03-handout.pdf`

[27] Hopcroft, J. E.; Wong, J. K.: Linear Time Algorithm for Isomorphism of Planar Graphs (Preliminary Report). In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, New York, NY, USA: Association for Computing Machinery, 1974, ISBN 9781450374231, str. 172–184, doi:10.1145/800119.803896. Dostupné z: `https://doi.org/10.1145/800119.803896`

[28] Dickinson, P. J.; Bunke, H.; Dadej, A.; aj.: On Graphs with Unique Node Labels. In *Graph Based Representations in Pattern Recognition*, editace E. Hancock; M. Vento, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-45028-3, s. 13–23.

[29] Kuhn, H. W.; Yaw, B.: The Hungarian method for the assignment problem. *Naval Res. Logist. Quart*, 1955: s. 83–97.

[30] Jeh, G.; Widom, J.: SimRank: A Measure of Structural-Context Similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, New York, NY, USA: Association for Computing Machinery, 2002, ISBN 158113567X, str. 538–543, doi:10.1145/775047.775126. Dostupné z: https://doi.org/10.1145/775047.775126

[31] Gori, M.; Maggini, M.; Sarti, L.: Exact and approximate graph matching using random walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ročník 27, č. 7, 2005: s. 1100–1111, doi:10.1109/TPAMI.2005.138.

[32] MetaCentrum. [online], 2020, [cit. 2021-01-20]. Dostupné z: https://www.metacentrum.cz/en/

[33] *Arithmetic Mean*. New York, NY: Springer New York, 2008, ISBN 978-0-387-32833-1, s. 15–18, doi:10.1007/978-0-387-32833-1_12. Dostupné z: https://doi.org/10.1007/978-0-387-32833-1_12

Příloha **A**

# Seznam použitých zkratek

**ATPG** Automatic test patterns generation

**GED** Graph edit distance

**FSM** Finite state machine

**SOM** Self organizing map

Příloha B

# Skripty pro MetaCentrum

Zdrojový kód B.1: runner.sh spouštěcí skript pro simulaci obvodů

```
#!/bin/bash

for I in {0..9}
do
    qsub -l select=1:ncpus=2:mem=1gb:scratch_local=1gb:os=debian10 -l
        walltime=2:00:00 -v param=$I -N ATPG$I job.sh
done

for I in {10..99}
do
    qsub -l select=1:ncpus=2:mem=1gb:scratch_local=1gb:os=debian10 -l
        walltime=2:00:00 -v param=$I -N ATPG$I job.sh
done
```

Zdrojový kód B.2: job.sh skipt pro simulaci obvodů

```
#!/bin/bash

# The 4 lines above are options for scheduling system: job will run 1
    hour at maximum, 1 machine with 4 processors + 4gb RAM memory +
    10gb scratch memory are requested, email notification will be
    sent when the job aborts (a) or ends (e)

# define a DATADIR variable: directory where the input files are
    taken from and where output will be copied to
HOMEDIR="/storage/praha1/home/wijasjan"
GRAPHDIR="/storage/praha1/home/wijasjan/graphs" # substitute username
    and path to to your real username and path
ATPGDIR="/storage/praha1/home/wijasjan/atpg"
OUTPUTDIR="/storage/praha1/home/wijasjan/output"
# append a line to a file "jobs_info.txt" containing the ID of the
```

```
    job, the hostname of node it is run on and the path to a scratch
    directory
# this information helps to find a scratch directory in case the job
    fails and you need to remove the scratch directory manually
echo "$PBS_JOBID is running on node `hostname -f` in a scratch
    directory $SCRATCHDIR" >> $OUTPUTDIR/jobs_info.txt

#loads the Gaussian's application modules, version 03
#module add g03

# test if scratch directory is set
# if scratch directory is not set, issue error message and exit
test -n "$SCRATCHDIR" || { echo >&2 "Variable SCRATCHDIR is not
    set!"; exit 1; }
trap 'clean_scratch' TERM EXIT
mkdir $SCRATCHDIR/blif/ || { echo >&2 "First mkdir failed"; exit 1; }

# copy input file to scratch directory
# if the copy operation fails, issue error message and exit
cp $ATPGDIR/atpg $SCRATCHDIR || { echo >&2 "Error while copying
    atpg!"; exit 2; }
cp $ATPGDIR/minisat $SCRATCHDIR || { echo >&2 "Error while copying
    minisat!"; exit 2; }
cp $GRAPHDIR/* $SCRATCHDIR/blif/. || { echo >&2 "Error while copying
    graphs!"; exit 2; }
cp $ATPGDIR/vectors.pat $SCRATCHDIR || { echo >&2 "Error while
    copying atpg!"; exit 2; }

# move into scratch directory
cd $SCRATCHDIR
mkdir $SCRATCHDIR/ext-pat/ || { echo >&2 "Second mkdir failed"; exit
    1; }
# run Gaussian 03 with h2o.com as input and save the results into
    h2o.out file
# if the calculation ends with an error, issue error message an exit
./atpg -sim -pr vectors.pat -pw ext-pat/cordic-$param-ext.pat
    -pattype 2 blif/cordic-$param.blif|| { echo >&2 "Calculation
    ended up erroneously (with a code $?) !!"; exit 3; }

# move the output to user's DATADIR or exit in case of failure
cp ext-pat/* $OUTPUTDIR/. || { echo >&2 "Result file(s) copying
    failed (with a code $?) !!"; exit 4; }

# clean the SCRATCH directory
clean_scratch
```

Zdrojový kód B.3: matlab.sh skript spouštějící výpočet common-mode poruch

```
#!/bin/bash
```

```
# define a DATADIR variable: directory where the input files are
    taken from and where output will be copied to
HOMEDIR="/storage/praha1/home/wijasjan"
EXTPATDIR="/storage/praha1/home/wijasjan/ext-pat"
MATLABDIR="/storage/praha1/home/wijasjan/matlab"
OUTPUTDIR="/storage/praha1/home/wijasjan/outputMatrixes"
# append a line to a file "jobs_info.txt" containing the ID of the
    job, the hostname of node it is run on and the path to a scratch
    directory
# this information helps to find a scratch directory in case the job
    fails and you need to remove the scratch directory manually
echo "$PBS_JOBID is running on node `hostname -f` in a scratch
    directory $SCRATCHDIR" >> $OUTPUTDIR/jobs_info.txt

#loads the Gaussian's application modules, version 03
module add matlab

# test if scratch directory is set
# if scratch directory is not set, issue error message and exit
test -n "$SCRATCHDIR" || { echo >&2 "Variable SCRATCHDIR is not
    set!"; exit 1; }
trap 'clean_scratch' TERM EXIT
mkdir $SCRATCHDIR/ext-pat || { echo >&2 "error while creating
    ext-pat!"; exit 1; }
mkdir $SCRATCHDIR/outputMatrixes || { echo >&2 "error while creating
    outputMatrixes!"; exit 1; }
# copy input file to scratch directory
# if the copy operation fails, issue error message and exit
cp $EXTPATDIR/* $SCRATCHDIR/ext-pat/ || { echo >&2 "Error while
    copying data!"; exit 2; }
cp $MATLABDIR/* $SCRATCHDIR || { echo >&2 "Error while copying
    matlab!"; exit 2; }
# move into scratch directory
cd $SCRATCHDIR

# run Gaussian 03 with h2o.com as input and save the results into
    h2o.out file
# if the calculation ends with an error, issue error message an exit
matlab -nodisplay -nodesktop -nosplash -r "atpgMatrix"

# move the output to user's DATADIR or exit in case of failure
cp -r $SCRATCHDIR/outputMatrixes $OUTPUTDIR/. || { echo >&2 "Result
    file(s) copying failed (with a code $?) !!"; exit 4; }

# clean the SCRATCH directory
clean_scratch
```

Příloha C

# Obsah přiloženého USB

```
readme.txt ............................ stručný popis obsahu USB
data ..................... adresář se spustitelnou formou implementace
  ext-pat ................................ odsimulované obvody
  outputMatrixes ............................ výstupní matice
  vectors.pat ............................... vstupní vektory
src ...................................... adresář se zdrojovými kódy
  scripts ............................ zdrojové kódy implementace
  thesis ..................... zdrojová forma práce ve formátu LaTeX
text ........................................................ text práce
  thesis.pdf ........................... text práce ve formátu PDF
```