VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

EVOLUČNÍ NÁVRH KOMBINAČNÍCH OBVODŮ
EVOLUTIONARY DESIGN OF COMBINATIONAL DIGITAL CIRCUITS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Ondřej Hojný**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Radomil Matoušek, Ph.D.**
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Ondřej Hojný**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**doc. Ing. Radomil Matoušek, Ph.D.**
Akademický rok:	2020/21

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Evoluční návrh kombinačních obvodů

Stručná charakteristika problematiky úkolu:

U kombinačních logických obvodů jsou okamžité hodnoty jeho výstupních proměnných jednoznačně určeny okamžitými kombinacemi hodnot jeho vstupních proměnných (jedná se o systémy statické – bez paměti). Evoluční návrh kombinačních obvodů reprezentuje alternativní přístup k širokému spektru známých konvenčních postupů (např. metoda Quine–McCluskey) a poskytuje mnohdy výrazně lepší řešení. Kartézské genetické programování (CGP) představuje metodu vhodnou k návrhu kombinačních obvodů.

Cíle diplomové práce:

- Popis kartézského genetického programování (CGP).
- Rešerše metod pro návrh kombinačních obvodů.
- Modelové příklady návrhu a testovací úlohy.
- Vyhodnocení dosažených výsledků.

Seznam doporučené literatury:

MILLER, J., JOB, D., VASSILEV, V.: Principles in the Evolutionary Design of Digital Circuits - Part I. Genetic Programming and Evolvable Machines. Vol. 1 No. 1, 2000, str. 8-35.

SEKANINA, L.: Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer Verlag, Berlin 2003.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2020/21

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRAKT

Tato diplomová práce se zabývá využitím kartézského genetického programování (CGP) k návrhu kombinačních obvodů. V práci je řešena problematika optimalizace vybraných logických obvodů, aritmetické sčítačky a násobičky, pomocí kartézského genetického programování. Implementace algoritmu je provedena v jazyce Python a primárně knihovnách NumPy, Numba a Pandas. Implementace CPG byla odzkoušena na zvolených příkladech a výsledky diskutovány.

ABSTRACT

This diploma thesis deals with the use of Cartesian Genetic Programming (CGP) for combinational circuits design. The work addresses the issue of optimizaion of selected logic circuts, arithmetic adders and multipliers, using Cartesian Genetic Programming. The implementation of the CPG is performed in the Python programming language with the aid of NumPy, Numba and Pandas libraries. The method was tested on selected examples and the results were discussed.

KLÍČOVÁ SLOVA

genetické programování, kartézské genetické programování, kombinační logické obvody, evoluční algoritmy, binární násobička, binární sčítačka, evoluční hardware, python, numba

KEYWORDS

genetic programming, cartesian genetic programming, combinational logic circuits, evolutionary algorithms, binary multiplier, binary adder, evolvable hardware, python, numba

ÚSTAV AUTOMATIZACE A INFORMATIKY

2021

BIBLIOGRAFICKÁ CITACE

HOJNÝ, Ondřej. *Evoluční návrh kombinačních obvodů*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky, 2021, 71 s. Diplomová práce. Vedoucí práce: doc. Ing. Radomil Matoušek, Ph.D.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato diplomová práce je mým původním dílem, vypracoval jsem ji samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků.

V Brně dne 21. 5. 2021 .....

Ondřej Hojný

PODĚKOVÁNÍ

Děkuji svému vedoucímu doc. Ing. Radomilu Matouškovi, Ph.D. za užitečné rady a odbornou pomoc při řešení tohoto projektu. Také bych rád poděkoval své rodině za nepřetržitou podporu během celého studia.

OBSAH

1 Úvod 15
2 Logické obvody a jejich návrh 17
2.1 Logické hradlo 18
2.2 Zpoždění obvodu 19
2.3 Příklady obvodů 20
2.4 Konvenční metody optimalizace 23
2.4.1 Karnaughova mapa 23
2.4.2 Quine–McCluskey algoritmus 24
3 Optimalizační algoritmy 27
3.1 Heuristické a meta-heuristické metody 27
3.2 Simulované žíhání 28
3.3 Evoluční algoritmy 28
3.3.1 Selekce 29
3.3.2 Křížení a mutace 30
3.4 Evoluční strategie 31
3.5 Genetické programování 31
4 Kartézské genetické programování 33
4.1 Princip CGP 33
4.1.1 Vlastnosti CGP 34
4.1.2 Mutace 34
4.1.3 Vyhodnocení jedince 35
4.1.4 Evoluční algoritmus pro CGP 35
4.2 Použití CGP 35
4.2.1 Logické obvody 35
4.2.2 Obrazové filtry 37
5 Vlastní implementace 39
5.1 Použité knihovny 40
5.2 Implementace 41
5.2.1 Objekt Individual 41
5.2.2 Objekt Population 43
5.2.3 Objekt Evolution_Executor 44
6 Výsledky testovacích úloh 47
6.1 Výsledky 2x2 násobičky 48
6.2 Výsledky 3x2 násobičky 51
6.3 Výsledky 3x3 násobičky 54
6.4 Výsledky 5x4 poloviční sčítačky 57
6.5 Výsledky 4x4 úplné sčítačky 60

7 Závěr 65

1 Úvod

Evoluční design je způsob návrhu složitých systémů jako programů, elektrických obvodů nebo obrazových filtrů. Systémy vytvořené tímto způsobem nestaví na konvenčních metodách návrhu konkrétních aplikací a mají proto často mnoho předností jako vyšší rychlost, optimálnější velikost apod. O návrhu takových systémů můžeme říci, že jsou inovativní.

Evoluční design stojí na teorii evolučních algoritmů spadajících do meta-heuristických optimalizačních metod. Tento způsob optimalizace hledá inspiraci v Darwinově teorii evolučního vývoje. Metody využívají populace jedinců k prohledávání prostoru řešení. Z počátku byly evoluční algoritmy používány k hledání optimálních parametrů různých úloh. S postupem času byla ale dokázána jejich efektivnost v hledání řešení i mnohem komplexnějších problémů. V kapitole 3 je problém optimalizace popsán podrobněji. [1]

Tato práce se zabývá návrhem jednoduchých logických obvodů pomocí evolučního designu. Takové obvody mohou plnit různé funkce v komplexnějším systému jako je celý počítač. Příkladem mohou být vícebitové sčítačky, násobičky nebo paritní obvody.

Požadavky na takové obvody se stále zvyšují, a proto je snaha hledat optimálnější řešení pro konkrétní aplikace. Při návrhu digitálního obvodu se snažíme optimalizovat různé parametry, zejména je to počet hradel (případně počet tranzistorů) či zpoždění obvodu. Tyto problémy jsou popsány v kapitole 2.

V této oblasti bylo provedeno mnoho experimentů s různými přístupy k problému. Příkladem může být genetické programování, které na rozdíl od klasických evolučních výpočetních technik vytváří celistvé programy (případně logické obvody). Při návrhu logických obvodů se však tento přístup potýká s omezeními jako nadměrné rozrůstání stromové struktury.

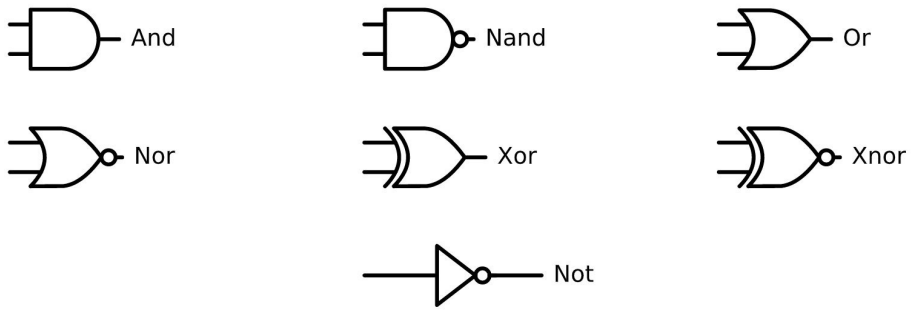
Velmi dobrých výsledků pro návrh logických obvodů bylo dosaženo zejména při použití kartézského genetického programování navrženého Julian F. Millerem. Tato varianta genetického programování reprezentuje jedince grafovou strukturou podobně jako genetické programování. Výhodu je však konstantní velikost jedince, která zamezuje nadměrnému rozrůstání složitosti obvodu. Způsob reprezentace a použití této metody jsou popsány v kapitole 4.

V této práci byla provedena implementace kartézského genetického programování v jazyce *Python*, specializovaná na vývoj logických obvodů. Vlastní implementace je popsána v kapitole 5. Program byl otestován na zvolených testovacích úlohách a výsledky popsány v kapitole 6.

2 Logické obvody a jejich návrh

Logické obvody jsou zvláštním typem elektrických obvodů, sloužících ke zpracování logických signálů. Logický signál je v praxi realizován dvouúrovňovým elektrickým signálem. Nízké napětí (obvykle 0-2 V) představuje logickou nulu a vyšší napětí (3-5 V) logickou jedničku. [2]

Číslicové obvody se nejčastěji realizují na úrovni logických hradel. Logické hradlo je elektronická součástka, realizovaná obvykle pomocí CMOS tranzistorů, která provádí určitou logickou operaci. Příkladem takové funkce může být hradlo AND, toto hradlo realizuje logický součin. Funkci každého hradla, nebo i celého obvodu, definuje pravdivostní tabulka. Značky nejpoužívanějších hradel můžeme vidět na Obr. 1. Poznamenejme, že existuje několik způsobů, jak hradla značit, nejznámější je notace dle ANSI (kulaté značky) a notace dle IEC (hranaté značky), význam je intuitivní, přičemž kolečko na výstupu znamená negaci.



Obr. 1: Schématické značky hradel podle normy ANSI/MIL

Z jednotlivých hradel se potom skládají logické obvody. Ty můžeme rozdělit na dvě základní kategorie. Kombinační obvody, kterými se zabývá tato práce, jsou obvody, jejichž výstup závisí výhradně na kombinaci vstupů. Výstup sekvenčních obvodů pak závisí i na posloupnosti předchozích hodnot. V sekvenčním obvodu tedy existují zpětné vazby.

Při optimalizaci kombinačních obvodů je jedním z nejzásadnějších parametrů počet logických hradel, provádějících algebraické operace. V Booleově algebře můžeme problém optimalizace obvodů popsat jako proces hledání ekvivalentního obvodu (tedy obvodu se stejnou pravdivostní tabulkou), který má menší počet logických funkcí. [3]

2.1 Logické hradlo

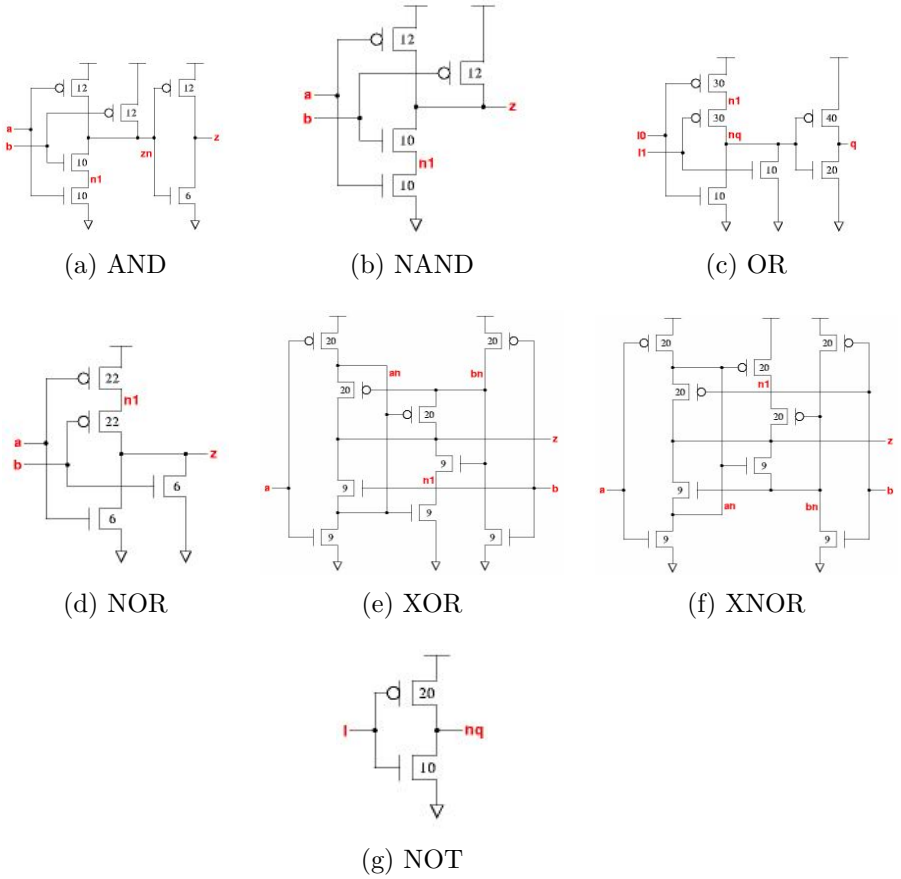
Pro provádění jednotlivých logických operací se používají různá hradla, která nemají stejné vlastnosti. Hradla provádějí různé logické operace a musí být realizována různým uspořádáním tranzistorů.

Uspořádání tranzistorů hradla má zásadní vliv na jeho spotřebu energie. CMOS obvody mají kapacitní charakter. Kvůli tomu se spotřeba energie nejvíce projevuje při přepínání logických hodnot. Tato spotřeba energie se označuje dynamický příkon a můžeme ji vyjádřit následující rovnicí [2]:

$$P_D = C_D V_{CC}^2 f, \tag{1}$$

kde f je frekvence přepínání, V_{CC} napájecí napětí a C_D kapacita obvodu při přepínání. Kapacita C_D není reálnou kapacitou, pouze popisuje chování obvodu a je obvykle udávána výrobcem.

Nejjednodušším typem logického hradla je invertor, který je složen pouze ze dvou tranzistorů. Signál vycházející z invertoru je negací signálu do něj vstupujícího. Obecně jsou invertující hradla složena z menšího počtu tranzistorů, protože invertování signálu přichází přirozeně. Hradla s více vstupy jsou obecně složena z více tranzistorů a mají tak horší vlastnosti. Větší počet tranzistorů také ovlivňuje fyzickou velikost součástky a v důsledku celkovou velikost obvodu. [2]



(a) AND (b) NAND (c) OR

(d) NOR (e) XOR (f) XNOR

(g) NOT

Obr. 2: Realizace hradel CMOS tranzistory podle [4]

2.2 Zpoždění obvodu

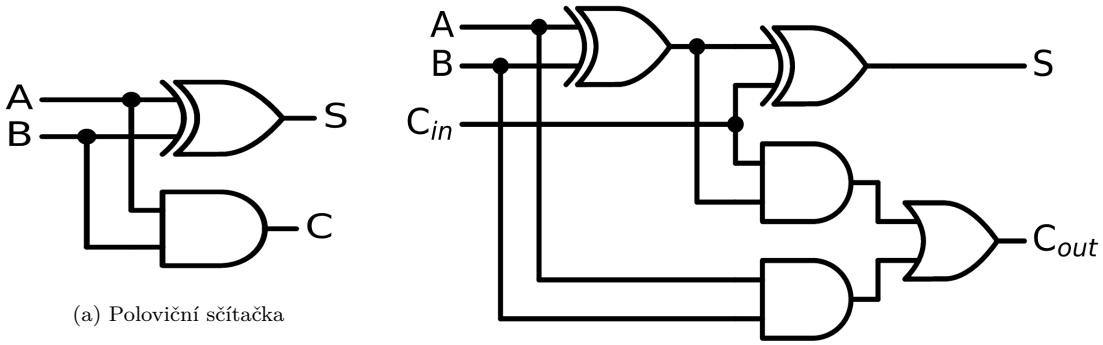
Různé typy hradel mají různé zpoždění závislé na jejich kapacitanci. Modelováním zpoždění obvodu se zabývá metoda *logical effort*.

Každé hradlo v logickém obvodu způsobuje zpoždění signálu zapříčiněné kapacitou hradla. Z tohoto důvodu se s rostoucí zátěží zvětšuje i zpoždění hradla. Zpoždění obvodu ovlivňuje i zapojení hradel. Hradla zapojená paralelně zvyšují kapacitanci obvodu, a tedy i zpoždění signálu. V některých případech tak může přidání hradla do obvodu zlepšit jeho chování. Zpoždění jednotlivých hradel se skládá ze dvou částí. Parazitní zpoždění, které je konstantní, a zpoždění způsobené zátěží. [5]

Pro účely optimalizace se obvykle počítá pouze s nejdelší hradlovou cestou. Zpoždění obvodu vyjádříme jako $n \cdot T$, kde n je počet hradel v nejdelší hradlové cestě a T je konstantní zpoždění. [6]

2.3 Příklady obvodů

Mezi nejzákladnější logické obvody patří poloviční a úplná sčítačka. Poloviční sčítačka je obvod, který má dva vstupy a dva výstupy a provádí součet dvou binárních hodnot. Úplná sčítačka má navíc jeden vstup, který vyjadřuje přenos z předchozího řádu. Pravdivostní tabulky (Tab. 1 a 2) a realizaci těchto obvodů (Obr. 3a a 3b) můžeme vidět níže. [2]



(a) Poloviční sčítačka

(b) Úplná sčítačka

Obr. 3: Bitové sčítačky

Vlastnost úplné sčítačky přijímat bit z předchozího řádu je důležitá pro sestavení složitějších obvodů. Pokud je například naším cílem sestavit sčítačku, která bude sčítat dvě čtyřbitová čísla, můžeme zvolit kaskádové zapojení sčítaček. Problémem takového řešení je však zvětšující se zpoždění s každou přidanou sčítačkou. Další možností je vyžít *carry lookahead* sčítačku. Ta využívá polovičních sčítaček a *carry lookahead* logického obvodu, provádějícího kalkulace s přenosovými bity. Tyto sčítačky používají větší počet hradel, ale jejich zpoždění je výrazně menší. Dále existují integrované obvody jako *74283* (Obr. 4). Tento konkrétní obvod je čtyřbitová úplná sčítačka. Jeho struktura je podobná struktuře *carry lookahead* sčítačky. [2]

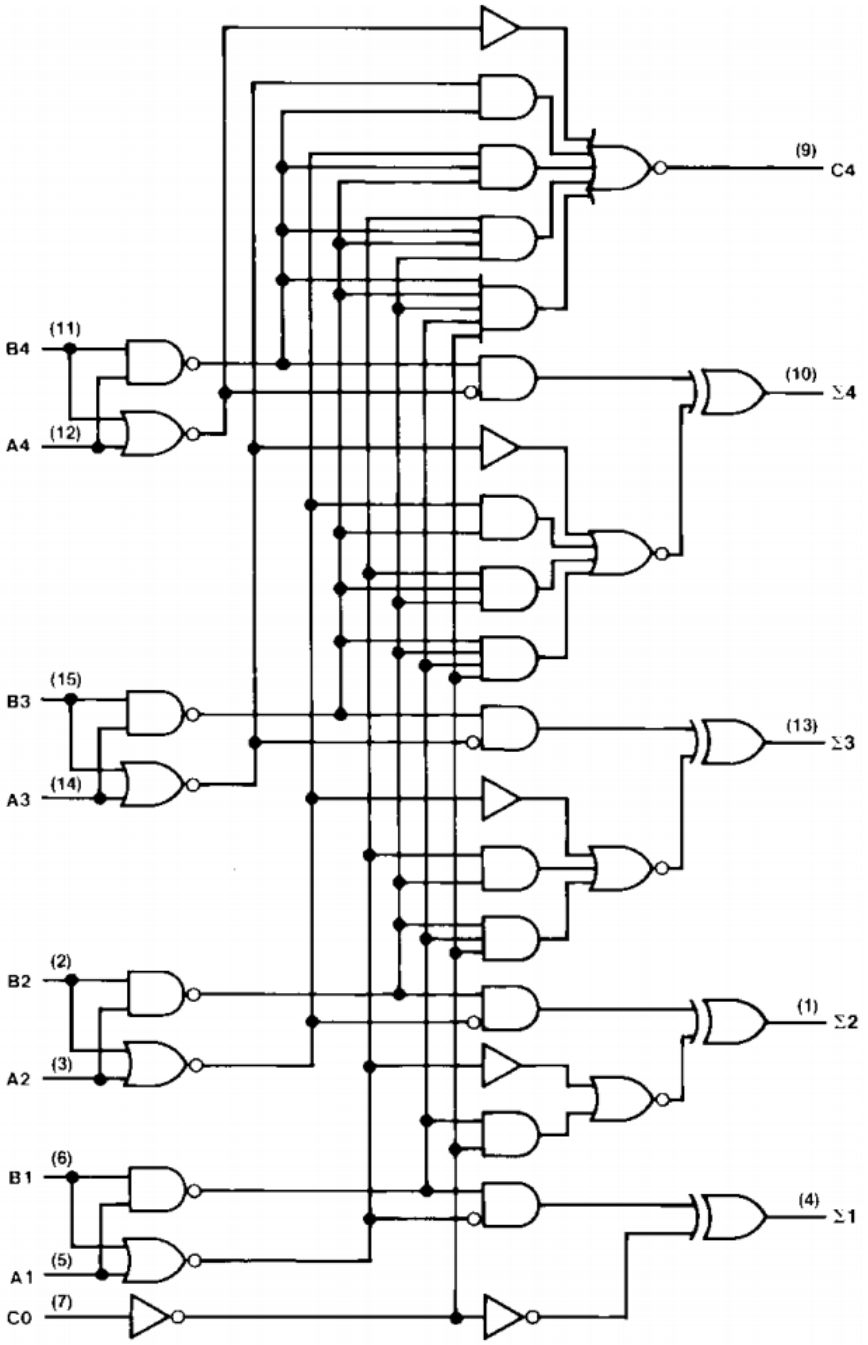
Pro návrh obvodu binární násobičky se používá podobné logiky jako při ručním násobení binárních čísel. Tato logika funguje na principu bitového posunu a bitového součtu. Tento způsob je jednoduše přeložitelný pro implementaci pomocí kombinace AND hradel, polovičních a úplných sčítaček (Obr. 5). [2, 7]

Tab. 1: Pravdivostní tabulka poloviční sčítačky

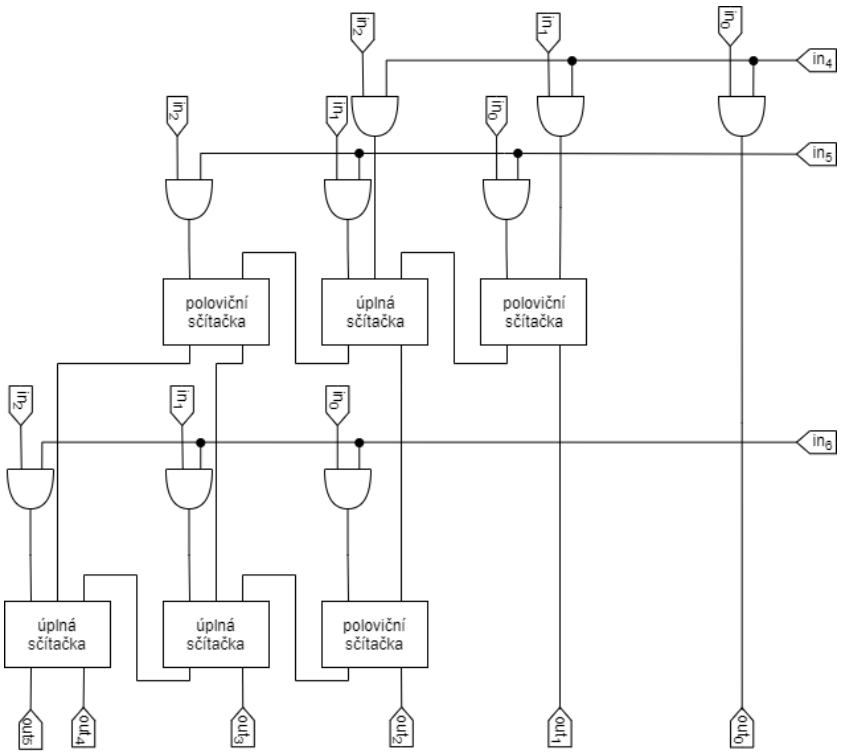
vstupy		výstupy	
A	B	S	C
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	1

Tab. 2: Pravdivostní tabulka úplné sčítačky

vstupy			výstupy	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Obr. 4: 4x4 úplná sčítačka implementovaná integrovaným obvodem *74283* [8]



Obr. 5: 3x3 násobička podle [7]

2.4 Konvenční metody optimalizace

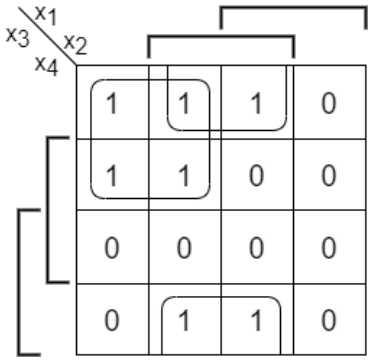
Minimalizace hradel je důležitá nejen kvůli minimalizaci ceny pro sériovou výrobu, ale také pro snížení příkonu obvodu či jeho zpoždění.

Nejzákladnějším způsobem minimalizace je algebraická minimalizace. Logický obvod si můžeme vyjádřit pomocí algebraických výrazů, které na základě pravidel Booleovy algebry můžeme zjednodušit. Tento způsob je však pracný, těžko automatizovatelný, a hlavně si nemůžeme být jisti, že je výraz minimální. Z těchto důvodů je tento způsob nevhodný pro optimalizaci složitějších obvodů. Další nevýhodou tradičních optimalizačních metod je to, že nezohledňují počet tranzistorů různých hradel. [3, 2]

2.4.1 Karnaughova mapa

Karnaughovy mapy jsou způsobem popisu logické funkce a dají se využít i k její optimalizaci. Postup při minimalizaci funkce vychází z faktu, že sousední políčka v mapě se liší v hodnotě pouze jedné proměnné. Stejné logické hodnoty pak můžeme rozdělit do skupin (dvojic, čtveřic, osmic, šestnáctic, ...), které reprezentují jednotlivé logické operace. Podaří-li se nám najít minimum skupin, jsme schopni sestavit minimalizovaný obvod.

Příklad na Obr. 6 je logickou funkcí definovanou Karnaughovou mapou. Po rozdělení do dvou čtveřic v Karnaughově mapě minimalizujeme funkci na výraz $x'_1x'_3 + x_2x'_4$.



Obr. 6: Karnaughova mapa

Opět však u této metody narážíme na omezení. Zpracovat takto můžeme pouze funkce o maximálně šesti proměnných. Metoda se tedy využívá převážně pro jednodušší funkce, a to zejména při manuálním výpočtu (pro svůj grafický charakter je nevhodná na zpracování počítačem). [3, 2]

2.4.2 Quine–McCluskey algoritmus

Tento algoritmus je pravděpodobně nejznámější metodou i díky své poměrně jednoduché aplikaci pro počítače. Algoritmus je založen na principu distributivního zákona. Jednotlivé členy logické funkce rozdělíme do skupin se stejným počtem negovaných prvků. Mezi těmito skupinami potom hledáme dvojice, pro které můžeme výraz zjednodušit. Při odlišnosti pouze v jedné proměnné můžeme tuto proměnnou odstranit a do další iterace pokračuje zjednodušený výraz (pouze shodné proměnné). Opakující se členy vypustíme a postup opakujeme, dokud členy funkce lze redukovat. Postup je znázorněn v Tab. 3 pro stejný příklad jako v kap. 2.4.1.

Tab. 3: Algoritmus získání minimálních implikantů

0:	$x'_1x'_2x'_3x'_4$	0, 1:	$x'_1x'_2x'_3$	0, 1, 4, 5:	$x'_1x'_3$
1:	$x'_1x'_2x'_3x_4$	0, 4:	$x'_1x'_3x_4$	4, 6, 12, 14:	$x_2x'_4$
4:	$x'_1x_2x'_3x'_4$	1, 5:	$x'_1x'_3x'_4$		
5:	$x'_1x_2x'_3x_4$	4, 5:	$x'_1x_2x'_3$		
6:	$x'_1x_2x_3x'_4$	4, 6:	$x'_1x_2x'_4$		
12:	$x_1x_2x'_3x'_4$	4, 12:	$x_2x'_3x'_4$		
14:	$x_1x_2x_3x'_4$	6, 14:	$x_2x_3x'_4$		
		12, 14:	$x_1x_2x'_4$		

Pro odstranění redundantních výrazů se využívá tabulka minimálních implikantů (Tab. 4). Pokud je symbol v řádku samostatně, minimální implikant nelze z konečného výrazu vypustit. V tomto případě jsou všechny výrazy potřebné a minimální výraz je $x'_1x'_3 + x_2x'_4$.

Tab. 4: Tabulka minimálních implikantů

	x_1	x_2	x_3	x_4	$x'_1x'_3$	$x_2x'_4$
0:	0	0	0	0	*	
1:	0	0	0	1	*	
4:	0	1	0	0	○	○
5:	0	1	0	1	*	
6:	0	1	1	0		*
12:	1	1	0	0		*
14:	1	1	1	0		*

Touto metodou již dokážeme zpracovat složitější funkce než pomocí Karnaughových map. Při zvětšování počtu výrazů se projevují omezení tohoto algoritmu. Problém pokrytí tabulky minimálních implikantů je totiž NP-složitý a časová složitost výpočtu tedy roste exponenciálně se zvyšujícím se počtem vstupů. [3, 9]

3 Optimalizační algoritmy

Problém optimalizace můžeme definovat jako problém hledání globálního extrému funkce $f : A \rightarrow \mathcal{R}$, kde jednotlivé prvky množiny $x \in A$ nazýváme kandidátní řešení. Hledaným extrémem $x^* \in A$ může být globální minimum, pro které platí $f(x^*) \geq f(x)$ nebo globální maximum $f(x^*) \leq f(x)$ pro všechna $x \in A$.

Funkce f nemusí mít žádné předpoklady. Může být spojitá, diskrétní, nemusí mít derivaci apod. Funkci si můžeme představit jako černou skříňku, která vrací funkční hodnotu pro zadaný argument. [1]

3.1 Heuristické a meta-heuristické metody

Existuje mnoho optimalizačních problémů, jejichž výpočetní složitost je příliš velká pro analytický, numerický nebo enumerativní přístup řešení. Z tohoto důvodu se často používají heuristické metody, či jejich vyšší forma označovaná jako meta-heuristiky. Heuristiky jsou obvykle specializované na konkrétní problémy, pro které byly navrženy. Mezi meta-heuristiky řadíme dále diskutované evoluční algoritmy, které se vyznačují dobrou schopností při řešení úloh globální optimalizace v oblasti nelineárních či komplexních optimalizačních úloh.

Tyto přístupy k hledání optimálního řešení často nachází inspiraci v přírodních nebo technických procesech. Mezi přírodou inspirované metody (*bio-inspired mothods*) můžeme zařadit velkou třídu evolučních algoritmů typu genetické algoritmy (GA), genetické programování (GP), diferenciální evoluce (DE), evoluční algoritmy (EA) aj. Dále hejnovým chováním inspirované algoritmy mravenčích kolonií (ACO) aj. Typickým příkladem inspirace ve fyzikálním procesu je algoritmus simulovaného žíhání (SA). Meta-heuristické metody jsou efektivním nástrojem pro prohledávání velkých prostorů řešení a řadí se mezi tzv. globální optimalizační metody. Poznamenejme, že jako u řady dalších optimalizačních metod, i zde obecně hrozí uváznutí v lokálním extrému. [1, 10]

Jedním z nejjednodušších heuristických přístupů je horolezecký algoritmus. V daném okolí kandidátního řešení a vytvoříme množinu nových kandidátních řešení, která následně ohodnotíme. Pokud je nejlepší nové řešení lepší než původní, použijeme ho jako nové kandidátní řešení. Tímto způsobem dokážeme poměrně efektivně najít lokální extrémy, pro hledání globálních extrémů je však potřeba použít robustnější algoritmy. [1]

3.2 Simulované žíhání

Simulované žíhání je meta-heuristický algoritmus, který se inspiruje procesem žíhání v hutnictví (postupným pomalým ochlazováním se materiál dostane na svoji energeticky nejnižší hodnotu, která v optimalizaci odpovídá globálnímu minimu). Algoritmus pracuje s pravděpodobností přijetí horšího řešení v závislosti na *„teplotě“*, tím se dosáhne úniku z lokálního extrému. Metoda funguje následovně, viz Algoritmus 1.

Algoritmus 1: Simulované žíhání [11]

```
teplota T = T_max;
stav S = S_0;
while not ukončovací podmínka do
    zvol náhodného souseda N;
    pravděpodobnost p = exp((Energie(N) − Energie(S))/T);
    if náhodné číslo u[0, 1] < p then
        S = N;
    end
    snížení teploty T;
end
```

Algoritmus nezaručuje nalezení globálního minima, ale je velmi rychlý a účinný. Pro zlepšení jeho chování lze například měnit způsob snižování teploty. [10, 11]

3.3 Evoluční algoritmy

Živé organismy nesou svou genetickou informaci v podobě DNA. Při své reprodukci se na nového jedince přenáší část informace z obou rodičů (křížení) a také dochází k náhodným mutacím. Podle teorie evoluční biologie k reprodukci nejčastěji dochází mezi nejsilnějšími jedinci.

Evoluční algoritmy jsou souborem optimalizačních metod s inspirací v evoluční biologii. Principem těchto algoritmů je prohledávání stavového prostoru pomocí populace. Populace je množinou kandidátních řešení, kterým se v tomto kontextu říká jedinci. Jedinec může být reprezentován různými způsoby (binární řetězec, list matematických operací atd.), tuto reprezentaci nazýváme chromozomem a jednotlivé prvky nazýváme geny. Nová řešení jsou pak získávána pomocí kombinací jedinců a náhodnými změnami chromozomu jedince.

V Evolučních algoritmech je průběh evoluce simulován tak, že každého jedince ohodnotíme účelovou (fitness) funkcí (která ukazuje, jak dobře splňuje naše požadavky) a vybranou selekční metodou (běžně se používá ruletový nebo turnajový výběr) vybereme jedince pro vytvoření nové generace. Základní schéma evolučního algoritmu je následující, viz Algoritmus 2.

Algoritmus 2: Evoluční algoritmus [1]

$i = 0$;
inicializace populace $P(i)$;
výpočet fitness pro populaci $fitness(P(i))$;
while *not* *ukončovací podmínka* **do**
  provedení selekce $Q(i) = selekce(P(i))$;
  vytvoření nových jedinců $Q'(i) = vytvoř\_nového\_jedince(Q(i))$
   (použití operátorů křížení a mutace);
  ohodnocení nových jedinců $fitness(Q'(i))$;
  vytvoření nové populace $P(i+1) = výběr\_nové\_populace(P(i), Q'(i))$;
  $i = i\ + 1$;
end

Vytvoření nové populace je prováděno pomocí dvou základních operátorů, a to křížení a mutace. [1, 12, 10]

3.3.1 Selekce

Selekce slouží k výběru rodičů nové populace. Snažíme se dosáhnout toho, aby novou populaci osadili co nejlepší jedinci, a zároveň zajistit různorodost populace, aby nedošlo ke stagnaci.

Pro selekci existují různé způsoby. Jednou z prvních selekčních metod byla ruletová selekce. Ruletová selekce přiřadí každému jedinci pravděpodobnost, se kterou může být vybrán pro tvorbu nových jedinců. Tato pravděpodobnost je závislá na hodnotách účelové funkce jedinců v populaci a počítá se podle rovnice (2):

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}, \tag{2}$$

kde f_i je fitness hodnota i-tého jedince a n je velikost populace. Nevýhodou ruletové selekce je velká pravděpodobnost stagnace populace. Pokud je v populaci jeden výrazně lepší jedinec, je velmi malá pravděpodobnost výběru jiných jedinců a dochází tak k degradaci populace. Podobným přístupem, který tento problém řeší, je výběr podle pořadí, kdy je populace seřazena podle hodnot fitness.

Pravděpodobnost výběru jedince je potom závislá na jeho umístění. Další variantou je selekce deterministická, kde vybíráme pouze nejlepší jedince populace.

Častěji používanou selekční metodou je turnajová selekce. Turnajová selekce využívá pouze hodnoty fitness jednotlivých jedinců. Vybereme n náhodných jedinců (pro každého je stejná pravděpodobnost výběru) a z nich vybereme toho nejlepšího. Počet jedinců n, kteří vstupují do výběru, musí být poměrně nižší než velikost populace. Tak je zajištěno, že nová populace nebude naplněna jen těmi nejlepšími jedinci a zároveň pravděpodobnost, že nejlepší jedinec nebude v nové populaci, je zanedbatelně malá. Turnajový výběr opakujeme, dokud nedosáhneme požadovaného počtu jedinců. [1, 13, 14]

3.3.2 Křížení a mutace

Křížení se provádí kombinací chromozomů dvou rodičů. Můžeme provádět jednobodové či vícebodové křížení. Jestliže chromozom vyjádříme jako řetězec genů, můžeme zvolit bod v tomto řetězci, ve kterém oba rodiče rozdělíme. Nového jedince vytvoříme kombinací rodičů ve zvoleném bodě. Vícebodové křížení pak těchto bodů definuje více a umožňuje tak důkladnější kombinování genetické informace. Další variantou křížení je uniformní křížení. Pro tento způsob má každý gen danou pravděpodobnost na prohození mezi rodiči. Principy variant křížení jsou znázorněny v Tab. 5.

Tab. 5: Příklad schématu křížení

Rodiče		Potomci
Jednobodové křížení		
(**100**|10010)	$\longrightarrow$	(**100**|**00101**)
(101|**00101**)	$\longrightarrow$	(101|10010)
Vícebodové křížení		
(**100**|100|**10**)	$\longrightarrow$	(**100**|100|**01**)
(101|**001**|01)	$\longrightarrow$	(101|**001**|10)
Uniformní křížení		
(**10**0**1**0**0**10)	$\longrightarrow$	(**10**0**0**0**1**10)
(**10**1**0**0**0**101)	$\longrightarrow$	(**10**1**1**0**0**001)

Mutaci jednoduše provedeme změnou genu v chromozomu na novou hodnotu. Při aplikaci operátoru mutace volíme parametr pravděpodobnosti mutace. Tento parametr určuje, s jakou pravděpodobností se každý gen

v chromozomu mění. Obvykle se tato pravděpodobnost volí velmi malá ($p_m = 0,1\%$), aby docházelo k postupnému prohledávání po malých krocích [1]. Tímto způsobem je zachován evoluční charakter procesu. Mutace chromozomu je znázorněna v Tab. 6.

Tab. 6: Příklad schématu mutace

Jedinec před mutací		Jedinec po mutaci
(1011**0**010)	$\longrightarrow$	(1011**1**010)

Pro některé způsoby reprezentace jedince je nutné respektovat daná pravidla pro křížení i mutaci, aby nebyl vytvořen jedinec nekompatibilní s danou metodou jeho reprezentace. [1, 12]

3.4 Evoluční strategie

Evoluční strategie je metoda, pro niž je typické, že využívá pouze operátor mutace. Pracuje na principu rozdělení populace na rodiče a potomky. Máme dva základní způsoby tvorby nové generace, a to $(\mu + \lambda)$ nebo (μ, λ), kde μ je počet rodičů a λ počet potomků.

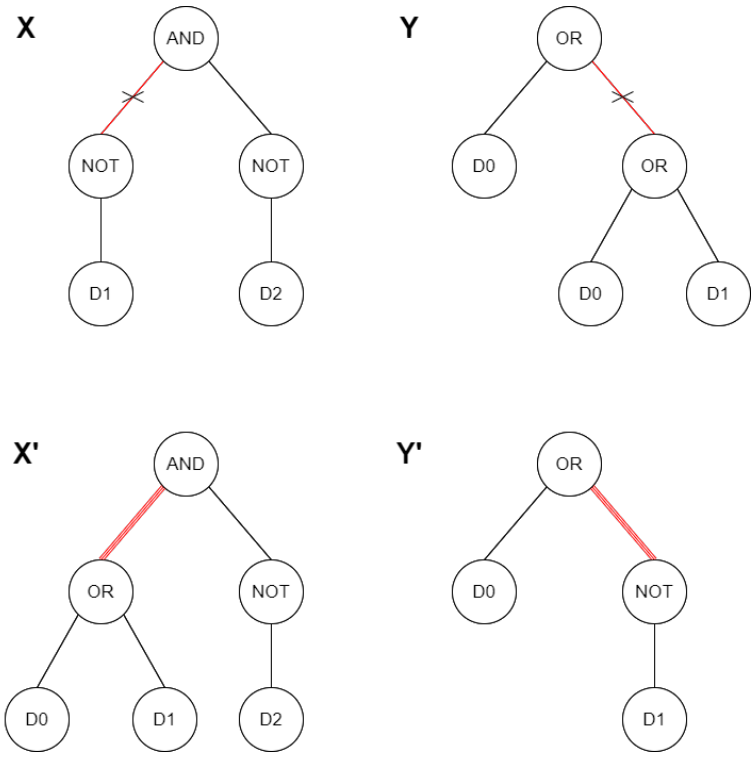
Při evoluční strategii $(\mu + \lambda)$ μ-krát opakujeme nahrazení rodiče původní populace novým (pomocí zvolené selekční metody), kde vybíráme z celé populace. Pro strategii (μ, λ) postupujeme obdobně, ale pro novou populaci se vybírá jen mezi λ potomky. Pro evoluční strategie je typická deterministická metoda selekce. [1]

3.5 Genetické programování

V evolučních algoritmech musíme jednotlivé jedince vhodně reprezentovat. Původní reprezentace jedinců byly realizovány pomocí proměnných s konstantní velikostí a evoluční algoritmus tak sloužil pouze k hledání optimálních parametrů.

Řešení předložené John R. Kozou se nazývá genetické programování [14] a slouží nejen k hledání optimálních parametrů, ale i ke generování celých programů. Výhodou tohoto způsobu je, že funkce programu je kompletně popsána jen pomocí chromozomu. Původní implementace využívaly programovacího jazyku LISP vhodného pro práci se stromovými strukturami. V takovém případě je jedinec reprezentován listem, kde jednotlivé uzly jsou buď funkce (algebraické operace) nebo terminály (konstanty nebo vstupy). [1]

Nejdůležitějším operátorem pro genetické programování je křížení (Obr. 7). Operátor křížení provede záměnu jednoho z uzlů za podstrom jiného jedince.



Obr. 7: Ilustrace křížení jedinců v genetickém programování

Operátor mutace je prováděn záměnou náhodného uzlu za nový podstrom nebo terminál a má velmi malou pravděpodobnost výskytu. Zastává tedy pouze podpůrnou funkci pro evoluční algoritmus.

Jedinec reprezentovaný touto metodu nemá statickou velikost. Pomocí genetického programování byly doposud vyvinuty pouze jednodušší programy. Při vývoji složitějších programů je problém s nadměrnou velikostí reprezentujícího grafu. [14, 1]

Genetické programování bylo krom jiného využito pro vývoj kombinačních obvodů na úrovni hradel. [15]

Variantou genetického programování je kartézské genetické programování, podrobně popsané v kapitole 4.

4 Kartézské genetické programování

Kartézské genetické programování (dále CGP) je forma genetického programování poprvé vyvinuta Julianem F. Millerem k návrhu logických obvodů [16, 17, 18]. V České republice s touto metodou extenzivně pracoval prof. Lukáš Sekanina a jeho studenti na Fakultě informačních technologií VUT v Brně [19, 20, 21]. Metoda využívá grafové reprezentace k popisu problému. Uzly grafu jsou rozmístěné ve dvourozměrné mřížce s pevně daným počtem sloupců a řad. Tyto uzly mohou reprezentovat například logická hradla nebo jiné matematické operace, hrany pak ukazují na vstupy do těchto operací. [18]

4.1 Princip CGP

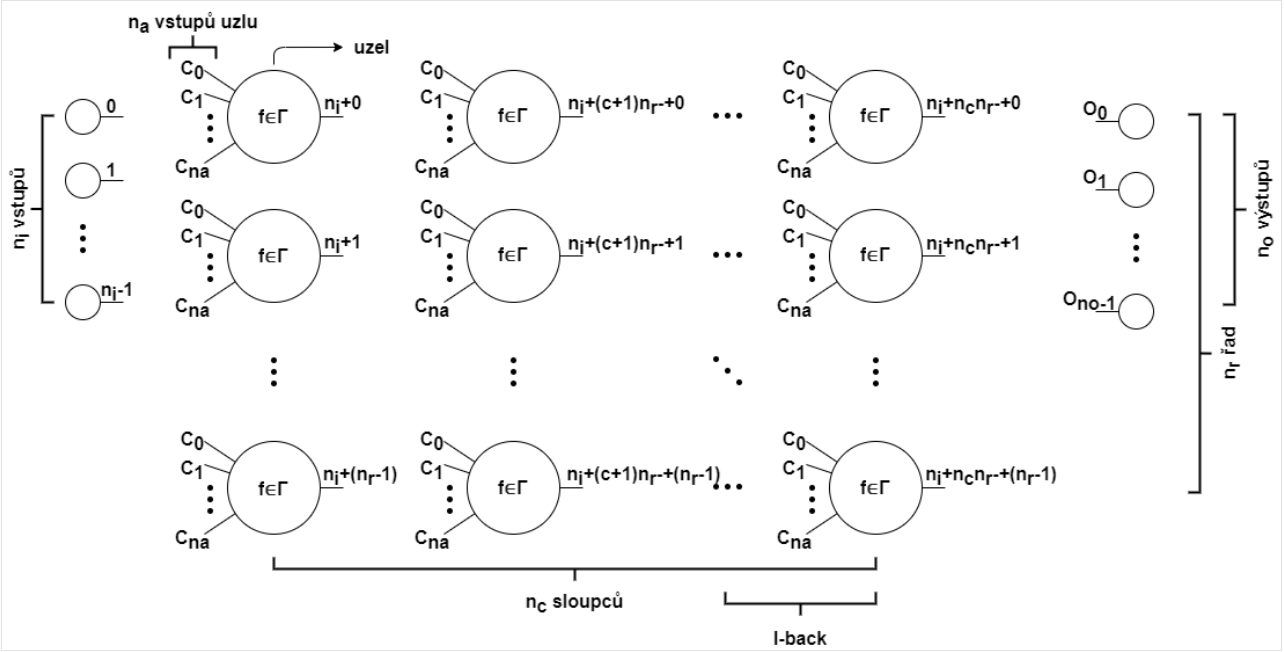
Genotyp (kódová reprezentace chromozomu, resp. jedince, tedy potenciálního řešení) je typicky pole integerů, které popisují spojení jednotlivých uzlů grafu (výhodou je statická velikost chromozomu). Jednotlivé uzly jsou indexovány počínaje vstupy. Samotná síť uzlů je indexována po sloupcích, tento způsob indexování zjednoduší práci s generováním hran grafu. Vstupní uzly mohou být pouze z předchozích sloupců, aby se zabránilo zacyklení. Dále můžeme omezit i počet sloupců, kam může uzel dosáhnout.

Jedinec je tedy definovaný následujícími konstantami:

- n_c, n_r - počet sloupců a řad
- n_i, n_o - počet vstupů a výstupů (je neměnný a závisí na daném problému)
- n_a - počet vstupů do jednotlivých uzlů
- *l-back* - definuje počet sloupců zpětného zapojení
- Γ - množina použitých funkcí (kde n_f je $|\Gamma|$)

Genotyp můžeme rozdělit na geny popisující jednotlivé uzly grafu. Každý uzel musí být popsán $n_a + 1$ integery, udávajícími typ uzlu a jeho vstupy. Geny výstupu pak odkazují na výstupní uzly a nejsou indexovány. S těmito znalostmi vidíme, že k zakódování jedince je zapotřebí $n_c \cdot n_r \cdot (n_a + 1) + n_o$ integerů. [22]

Indexování uzlů musí zohlednit počet vstupů, proto jsou vstupy indexovány jako první (0 až $n_i - 1$) a indexování uzlů grafu je v rozmezí od n_i do $n_c \cdot n_r - 1$. Na Obr. 8 je diagram CGP grafu.



Obr. 8: Reprezentace CGP [22]

4.1.1 Vlastnosti CGP

Vzhledem k charakteru CGP je na rozdíl od klasického genetického programování mnoho uzlů grafu *"neprogramujících"*. To znamená, že nemají vliv na výstup funkce. Proto může existovat více jedinců, kteří mají různé chromozomy nebo i fenotypy a zároveň jsou ohodnoceni stejnou hodnotou fitness funkce. Tato vlastnost se nazývá neutralita a je důsledkem redundance. Redundance v CGP chromozomu může být na různých úrovních. Jedná se o redundanci na úrovni uzlů (nepoužité uzly jsou redundantní), na úrovni funkcí (stejný výsledek může být realizován s využitím menšího počtu uzlů) a na úrovni vstupů (typ funkce nevyužívá všechny vstupy do uzlu). [23]

V důsledku neutrality mohou náhodné mutace vytvářet funkčně stejné jedince. Bylo ukázáno, že tato vlastnost pozitivně působí při prohledávání stavového prostoru. [24]

4.1.2 Mutace

V implementacích CGP se běžně používá jen operátor mutace. Mutaci v CGP provádíme tak, že změníme náhodný gen v chromozomu na nové náhodné číslo. Při změně genu je nutné respektovat stejná pravidla jako při tvorbě chromozomu, tzn. geny reprezentující typ funkce musí mutovat pouze na legální hodnoty a geny reprezentující vstupy uzlu rovněž nesmí přesáhnout svá omezení. Mutace může rovněž změnit gen reprezentující výstup fenotypu. [23]

4.1.3 Vyhodnocení jedince

Pro vyhodnocení výsledných hodnot na dané vstupní hodnoty potřebujeme znát použité uzly (fenotyp). Ty získáme zpětnou propagací od uzlů genů výstupu. U každého hradla zjistíme vstupy a postup opakujeme pro každý z těchto vstupů, dokud nezbudou pouze vstupy do funkce. Všechny jednotlivé uzly v tomto seznamu pak vyhodnotíme a tak zjistíme výsledky jednotlivých výstupů. [6]

Používá se také způsob, kde vyhodnotíme reakce všech uzlů v mřížce a hodnoty na výstupních uzlech použijeme k výpočtu fitness. Tím odstraníme nutnost hledání použitých uzlů. Tento způsob je ale výpočetně náročnější při velkém množství redundantních genů. [25]

4.1.4 Evoluční algoritmus pro CGP

Evoluční algoritmus používaný pro CGP je obdobou Evoluční strategie $(\mu + \lambda)$, přičemž se běžně pracuje pouze s jedním rodičem, a tak píšeme $(\mu + \lambda)$. Jelikož u CGP je část uzlů nekódujících (tzn. nemají vliv na výstup funkce), je množství provedených mutací neutrální. Podle [18] je důsledkem této vlastnosti efektivnější prohledávání stavového prostoru. Vzhledem k vysoké redundanci se proto při výběru rodiče obvykle upřednostňuje potomek (za předpokladu stejné hodnoty fitness). Ukončující podmínka evoluce může být nalezení dostatečně dobrého řešení nebo dosažení určitého počtu generací. [1]

4.2 Použití CGP

V minulosti bylo CGP použito pro různé aplikace. Zejména se jedná o návrh logických obvodů nebo o vývoj obrazových filtrů.

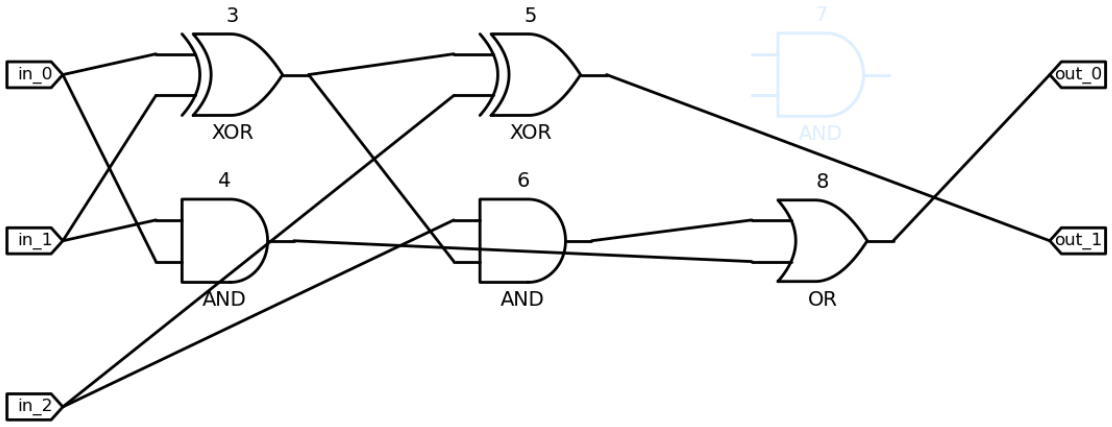
4.2.1 Logické obvody

CGP bylo použito pro vývoj nekonvenčních logických obvodů různými způsoby. Úkolem algoritmu je nalézt takový logický obvod, který splňuje funkci danou například pravdivostní tabulkou. Při řešení tohoto problému je obvykle nejdůležitějším parametrem, po nalezení správného řešení, minimalizace počtu hradel v obvodu. Byly také provedeny implementace multikriteriální evoluce, kde dalšími parametry mohou být zpoždění obvodu nebo ocenění jednotlivých hradel. Při hledání optimálních obvodů se často využívá založení populace z předem známých řešení. [18, 20]

Výhodou tohoto přístupu je nezávislost na konvenčních způsobech designu. Nalezená řešení jsou pak často zajímavější z hlediska praktických aplikací. Nalezený obvod může mít nejen lepší energetické a cenové vlastnosti, ale také okupovat menší prostor na čipu nebo mít větší spolehlivost.

Tento způsob vývoje však není bez svých nedostatků. Při zvětšujícím se počtu vstupů exponenciálně roste počet kombinací vstupních bitů, které je potřeba vyhodnotit pro výpočet fitness funkce. Krom toho se také s požadavkem na komplexnější obvody zvětšuje potřebná velikost CGP mřížky hradel reprezentovaného jedince. S každým přidaným hradlem do mřížky se pak zvětšuje i prohledávaný stavový prostor a tím pádem je nalezení řešení v něm časově náročnější. Tento problém se nazývá škálovatelnost řešení. Pro odstranění tohoto problému byly vyvinuty metody modulárního CGP, kde se část obvodu může nahradit již známým řešením. [18]

Na Obr. 9 můžeme vidět příklad jedince definovaného parametry v Tab. 7.



Obr. 9: Příklad jedince definovaného chromozomem: **2**, 0, 1; **0**, 1, 0; **2**, 3, 2; **0**, 2, 3; **0**, 2, 2; **1**, 6, 4; *8*, *5*

Tab. 7: Definující parametry jedince na Obr. 9

n_c	n_r	n_i	n_o	l-back	Γ
3	2	3	2	3	{AND, OR, XOR}

Dvoustupňová fitness funkce

Pro ohodnocení jedince při vývoji logických obvodů se obvykle využívá dvoustupňové fitness funkce (3). První stupeň závisí na počtu shodných bitů s pravdivostní tabulkou hledané funkce. Druhý stupeň pak závisí na počtu hradel ve fenotypu.

$$f = \begin{cases} b & b < n_o 2^{n_i}, \\ b + (n_c n_r - z) & \text{jinak.} \end{cases} \tag{3}$$

Kde b je počet shodných bitů, n_o počet výstupů, n_i počet vstupů a z počet hradel ve fenotypu.

Druhý stupeň tedy počítáme pouze pokud kandidátní řešení je správným (ne nutně optimálním) řešením hledaného problému. Můžeme tedy říct, že optimalizační fáze vývoje logického obvodu začne až po nalezení funkčního obvodu. Do té doby na složitosti obvodu nezáleží. [20]

Multikriteriální optimalizace

Jelikož hledání optimálního řešení reálného obvodu nezáleží pouze na počtu hradel, ale i na typech jednotlivých hradel a zpoždění obvodu, byly na tento problém aplikovány multikriteriální optimalizační algoritmy. Jako nejvhodnější se jeví algoritmus NSGAII použitý v [21, 18]. Na rozdíl od klasického CGP, kde se používá strategie $(1 + \lambda)$, se zde používá většího počtu rodičů, tedy strategie $(\mu + \lambda)$. Tento přístup podporuje diverzitu populace a dokáže blíže optimalizovat reálné obvody. V [18] byly obvody optimalizovány na počet hradel, počet tranzistorů, nejdelší hradlovou cestu a nejdelší tranzistorovou cestu. Tento přístup byl úspěšný pro návrh násobiček, sčítaček či ovladače pro sedmi-segmentový displej.

4.2.2 Obrazové filtry

CGP bylo použito také pro vývoj efektivních a kvalitních obrazových filtrů. Pro tuto aplikaci jsou uzly grafu různé bitové operace, manipulující s hodnotami pixelů jádra. Výstupem je pak hodnota pixelu po filtraci. Filtrovaný obraz je výslednou aplikací vyvinutého filtru pro každý pixel obrazu. Fitness jedince je často reprezentována jako suma absolutních hodnot rozdílů pixelu filtrovaného obrazu a obrazu bez šumu. Filtry vyvinuté touto metodou mají často lepší vlastnosti. Obrázek po jejich aplikaci není tolik rozostřen a zachovává více detailů. CGP bylo také využito k vývoji jader pro hledání hran. [18]

5 Vlastní implementace

V rámci této diplomové práce byla provedena implementace CGP (kap. 4) v jazyce *Python* pro výpočet logických obvodů. Evoluční strategie funguje na principu $(1 + \lambda)$, popsaném v kapitole 3.3. Dále je v algoritmu implementována selekce pomocí simulovaného žíhání.

Implementovaný algoritmus vidíme znázorněn pseudokódem v algoritmu 3:

Algoritmus 3: Implementovaný CGP evoluční algoritmus

```
inicializace populace 1 + λ;
inicializace teploty T = T_max;
inicializace c = (0.95, 0.999);
výpočet fitness všech jedinců populace;
rodičem R se stává nejlepší jedinec;
while not ukončovací podmínka do
    kontrola a uložení nejlepšího nalezeného jedince S do paměti;
    najdi nejlepšího potomka N;
    rozdíl fitness Δf = fitness(N) − fitness(S);
    if fitness(N) >= fitness(R) then
        nový rodič populace;
        R = N;
    else if exp(Δf/T) >= u[0, 1] then
        nový rodič populace;
        R = N;
    else
        zůstává stejný rodič populace;
    end
    vytvoření λ potomků mutací rodiče R;
    novou populaci tvoří rodič R a λ potomků;
    snížení teploty T = c · T;
    if T < T_min then
        T = T_max
    end
end
```

Výsledky jsou ukládány do souboru, který lze programem opět přečíst. Tímto způsobem můžeme pokračovat v již dokončeném výpočtu, a to i s některými změněnými parametry.

5.1 Použité knihovny

Pro realizaci programu byly využity knihovny *NumPy* a *Numba*. K paralelizaci evoluce byly použity knihovny *Pandas* a *futures3*, známější jako *concurent.futures*. Pro vizualizaci dat a obvodů knihovny *matplotlib* a *schemdraw*. Knihovna *pickle* byla využita pro ukládání výsledků.

Knihovna Schemdraw
Schemdraw je knihovna pro *Python* sloužící k vytváření diagramů elektrických obvodů a vývojových diagramů. Využívá k tomu dva základní objekty *Element* a *Drawing*. Pomocí *Element* objektů vkládáme různé prvky do objektu *Drawing*, který je následně vykreslen pomocí *matplotlib* backendu. Mezi jednotlivými *Elementy* definujeme vztahy jako vzájemné pozice či propojení součástek. [26]

Knihovna obsahuje základní elektronické součástky jako rezistory, ale i specifičtější součástky jako relé. Pro tuto práci je ale důležitý soubor logických hradel.

Knihovny NumPy, Pandas
Knihovna *NumPy* je hojně využívaná knihovna pro práci s daty. Hlavní předností této knihovny je výrazně lepší výpočetní rychlost při práci s poli. Z toho důvodu v této práci byla *NumPy* pole využita krom jiného pro reprezentaci chromozomu jedince nebo pravdivostní tabulky. [27]

Pro datovou analýzu se také běžně využívá knihovna *Pandas*. Tato knihovna je účinná pro zpracovávání velkých souborů tabulkových dat. Pro účely práce byla využita pro přehledné ukládání výsledků evoluce.

Knihovna Numba
Numba je open source JIT-compiler, který dokáže část *Python* a *NumPy* kódu přeložit do mnohonásobně rychlejšího strojového kódu. K tomu využívá knihovnu *LLVM*. Optimalizovaný kód dosahuje rychlostí blížících se rychlostem jazyků *C* nebo *FORTRAN*. Pro použití této knihovny přesto není potřeba používat oddělený překladač. *Numba* funguje nejlépe ruku v ruce s použitím *NumPy* polí.

Samotná aplikace knihovny je poměrně jednoduchá a využívá dekorátorů. Před funkci, kterou chceme zrychlit, stačí dát dekorátor a při prvním zavolání funkce *Numba* provede překlad a optimalizaci kódu. *Numba* přečte *Python* bytecode dekorované funkce a zkombinuje znalosti o typech argumentů funkce, provede analýzu a optimalizaci napsaného kódu a následně s využitím *LLVM* překladače vygeneruje strojový kód. Při dalších spuštěních je pak volána již přeložená funkce.

Nejdůležitější dekorátory jsou *@jit* a *@njit*. Dekorátor *@jit* dokáže přeložit jakoukoliv funkci, která by fungovala v jazyce *Python*. To ale neznamená, že taková

funkce bude vždy rychlejší než funkce původní. Při psaní kódu je však potřeba uvědomovat si úskalí tohoto způsobu. *Numba* si například nedokáže poradit s *Python* listy, slovníky a podobně. Pokud se v části kódu vyskytuje nějaký z těchto prvků, bude tato část běžet v *Pythonu* a přeloží se jen ty části, které jsou s *Numbou* kompatibilní. Při používání *Numby* se proto častěji využívá dekorátoru *@njit*, který nepovoluje tento *Python*-mód. [28]

Pro zrychlení výpočtu byly tyto poznatky aplikovány i v této práci. Vytvořené *Numba* funkce tak výrazně urychlily chod programu. Pro ilustraci bylo otestováno zrychlení 10 000 generací na 3x3 násobičce. S využitím *Numba* dekorátorů trvá tento výpočet průměrně 4,3 sekund. Po odstranění dekorátorů výpočet každých 10 000 generací trvá 870 sekund. Dochází tak přibližně k 200-násobnému zrychlení, což umožňuje dosažení požadovaných výsledků za výrazně kratší dobu.

5.2 Implementace

V této kapitole je podrobně popsána implementace evolučního algoritmu s kartézským genetickým programováním. Je zde popsána objektová struktura programu a vybrané metody daných objektů. Celý program hledá na základě definovaných parametrů obvod s minimálním počtem tranzistorů.

5.2.1 Objekt Individual

Objekt Individual slouží k výpočtům pro jednotlivé obvody, zejména jejich fitness funkce. Objekt nese informaci v chromozomu reprezentovaném *NumPy* polem integerů. Jelikož *Python*, jako jazyk, resp. jeho interpret, je poměrně pomalý, tak za účelem zrychlení algoritmu byly jeho časově nejdražší části napsány jako funkce pro JIT-compiler *Numba* (5.1). Tyto funkce slouží k výpočtu reakce hradel na vstup a pro výpočet fitness funkce.

___init___
Inicializace objektu vyžaduje následující vstupy:

1. **in__n** - počet vstupů hledané logické funkce
2. **out__n** - počet výstupů hledané logické funkce
3. **rows** - počet řádků pole hradel
4. **columns** - počet sloupců pole hradel
5. **fun__list** - seznam použitých typů hradel
6. **levels__back** - omezený počet sloupců, kam si každé hradlo může sáhnout pro vstup
7. **mutate__rate** - procento mutovaných genů v chromozomu

8. **truth_table_type** - typ hledaného logického obvodu (na základě této proměnné je vybrána pravdivostní tabulka)

Také je každému typu hradla přiřazeno ohodnocení v závislosti na počtu tranzistorů daného hradla. Počty tranzistorů byly získány z [4] a můžeme je vidět v Tab. 8.

Tab. 8: Počet tranzistorů použitých hradel

Typ hradla	Počet tranzistorů
NOT	2
AND, OR	6
NAND, NOR	4
XOR, XNOR	9
(not x) AND y	8

Dále je volána metoda *random_chromosome*, tím je vytvořen náhodný jedinec.

random_chromosome
Metoda *random_chromosome* náhodně vygeneruje nový chromozom na základě nastavených pravidel (počet řad, počet sloupců atd.).

mutate
Tato metoda provede náhodnou změnu n genů, kde n je závislé na proměnné *mutate_rate*. Mutace je tedy závislá i na velikosti pole hradel. Při mutaci je nutné rozlišit, který gen je mutován, a aplikovat správná omezení. Je nutné rozlišit, zda je mutován gen reprezentující typ hradla, vstup do něj nebo gen výstupu obvodu. Šance na mutaci je pro každý gen stejná.

fitness_calculation
V této metodě je realizována dvoustupňová fitness funkce prezentovaná v kapitole 4.2.1. Funkce byla upravena pro minimalizaci počtu tranzistorů místo počtu hradel. První stupeň je $b \cdot 100$ a druhý je $b + (100 - n_t/10)$, kde n_t je počet tranzistorů použitých v obvodu.

Jelikož výpočet výstupů obvodu a jeho porovnání s pravdivostní tabulkou je časově nejnáročnější částí algoritmu, jsou k tomuto účelu vytvořeny funkce pro kompilaci knihovnou *Numba*. V této metodě je tedy zavolána funkce *numba_grid_fitness* pro nalezení počtu shodných bitů a pokud bylo dosaženo správného výsledku, voláme *numba_used_nodes_rec* pro získání druhého stupně fitness funkce.

numba_grid_fitness
Tato funkce slouží jako první stupeň ohodnocení jedince. Pro každou kombinaci vstupních bitů získáme reakci všech hradel v chromozomu. Výstupní hradla jsou pak porovnána s pravdivostní tabulkou a je tak získán počet správných bitů.

Při vývoji byly testovány různé způsoby včetně využití pouze programovacích hradel. Výpočet byl však pomalejší, protože bylo nutné seznam těchto hradel získat pro každého jedince, i když tento seznam není nutný znát, pokud nepožadujeme výpočet druhého stupně fitness funkce.

numba_used_nodes_rec a numba_used_nodes_rec2
Tyto funkce jsou volány pouze pokud bylo dosaženo obvodu se všemi správnými bity. Slouží k získání seznamu použitých hradel. Funkce *numba_used_nodes_rec* pro každé výstupní hradlo zavolá funkci *numba_used_nodes_rec2*, která rekurzivně volá sama sebe pro získání seznamu použitých hradel. Kvůli nekompatibilitě překladače *Numba* s *Python* listy je nutné alokovat statické pole, které je předáváno mezi funkcemi. Pokaždé je zkontrolováno, zda vstupy do právě kontrolovaného hradla jsou již obsaženy v poli použitých hradel. Pokud v něm již jsou nebo se jedná o vstupní bit, není index hradla přidán do pole a funkci *numba_used_nodes_rec2* nevoláme. Tak je zamezeno prohledávání částí grafu, které již známe. Výstupem funkce *numba_used_nodes_rec* je pak pole indexů použitých hradel a počet tranzistorů v obvodu.

5.2.2 Objekt Population

V tomto objektu je vytvořena populace jedinců (Individual), se kterými se následně provádí evoluční proces. Zde dochází k největším změnám v nastavení algoritmu v počtu potomků λ.

___init___
Inicializace objektu vyžaduje stejné parametry pro inicializaci jedince a také parametr *pop_lambda*, určující počet potomků. Poté je populace inicializována metodou *init_population*.

init_population
Tato metoda slouží k vytvoření náhodné populace. Nejdříve je vytvořen prázdný list, do kterého jsou postupně přidáni jedinci. Počet jedinců je $(1 + \lambda)$. Následně je populace seřazena, aby nejlepší jedinec byl na místě rodiče.

mutate_population
Při každém zavolání této metody je provedena kontrola, zda mezi potomky není jedinec s lepší hodnotou fitness. Pokud ano, je tento jedinec uložen zvlášť do proměnné *best_individual* mimo populaci. Dále je aplikována selekční metoda pro

výběr nového rodiče, který slouží k vytvoření nové populace. Pro rychlejší běh algoritmu je každému jedinci v populaci pouze vyměněn chromozom za chromozom rodiče. Na jedince je následně aplikován operátor mutace a je mu vypočtena hodnota fitness.

new_parent
Selekční metoda v této implementaci je inspirována metodou simulovaného žíhání (kap. 3.2). Selekční metoda je nastavena tak, aby byla aplikována pouze po nalezení správného řešení.

Z populace je vybrán nejlepší potomek. Pokud je tento jedinec správným kandidátním řešením (má všechny shodné bity), je jeho fitness porovnána s fitness nejlepšího nalezeného jedince a na základě dané teploty je rozhodnuto, zda se stane novým rodičem. V každé generaci je také provedeno snížení teploty a při poklesu pod definovanou hodnotu je teplota resetována na počáteční hodnotu.

Důsledkem této selekční metody je přijímání horších jedinců jen s určitou pravděpodobností, závislou na nejlepším nalezeném řešení a momentální teplotě. Tímto způsobem jsou důkladněji prohledávána i horší řešení a nedochází tak k uvíznutí evoluce v lokálním optimu. Inspirací pro použití této metody byl článek [20].

5.2.3 Objekt Evolution_Executor

Tento objekt slouží ke spouštění evolučního algoritmu. Je zde implementován multiprocesing tak, aby mohly jednotlivé evoluční kmeny probíhat paralelně. Také lze zajistit pravidelnou výměnu nejlepších jedinců mezi jednotlivými kmeny. Těmto kmenům také můžeme dát zvláštní nastavení. Data jsou uložena v *Pandas* tabulce. Tato tabulka obsahuje jak objekt *Population*, tak další informace o průběhu evoluce. Mezi ně patří uplynulý čas od začátku vypočtu, historie vývoje fitness hodnoty rodiče, číslo generace prvního správného řešení a prvního nejlepšího řešení. Tyto hodnoty jsou důležité pro statistická zpracování evoluce v kap. 6. Také byly implementovány metody pro uložení dat s využitím knihovny *pickle*.

Evolution
Tato metoda slouží k vykonání evolučního algoritmu. Po inicializaci populace (pokud není předem definovaná) je spuštěna *while* smyčka. V každé iteraci je provedena mutace populace a kontrola ukončující podmínky. Ukončovací podmínkou evoluce je počet generací. Při nalezení lepšího jedince je zaznamenáno číslo generace pro statistické účely. Po splnění ukončovací podmínky je evoluce ukončena a data jsou zaznamenána do *Pandas* tabulky.

Evolution_exchange a Evolution_single

Tyto metody slouží k paralelizaci evolučních běhů, jejich použitím můžeme výrazně urychlit výpočet. K paralelizaci byla využita knihovna *futures3*. Každý evoluční běh běží zvlášť na jádru procesoru. *Evolution_exchange* provádí po daném počtu generací výměnu nejlepších jedinců mezi populacemi. V praktické části byl však využit mód *Evolution_single*, kde jsou na sobě jednotlivé běhy nezávislé. Při testování bylo zjištěno, že při výměně nejlepších jedinců častěji dochází ke stagnaci populace.

6 Výsledky testovacích úloh

Implementace CGP byla otestována na testovacích úlohách zvolených na základě experimentů provedených v [20, 18]. Při běhu evoluce je sledováno, v jaké generaci bylo nalezeno první správné řešení a kdy bylo nalezeno nejlepší řešení. Pro všechny úlohy bylo zvoleno $\lambda = 5$ potomků, míra mutace 2% a $levels\_back = n_c$. Každý jedinec je ohodnocen účelovou funkcí (4).

$$f = \begin{cases} b \cdot 100 & b < n_o 2^{n_i}, \\ b \cdot 100 + (100 - n_t/10) & \text{jinak.} \end{cases} \tag{4}$$

Kde b je počet shodných bitů v pravdivostní tabulce hledaného obvodu, n_t je počet tranzistorů v obvodu, n_i a n_o jsou počty vstupů a výstupů.

Pro evoluční běhy každého testovaného problému byly zvoleny různé sady hradel (Tab. 9) a různé rozměry CGP mřížky, zvolené s přihlédnutím k očekávanému počtu hradel a zpoždění navrhovaného obvodu. Evoluce byla vždy započata z náhodně vygenerované populace a pro každé nastavení bylo provedeno 30 běhů. Ukončovací podmínka byla zvolena na základě předem provedených testovacích běhů. Pro všechny běhy byly nastaveny teploty $T_{max} = 5, T_{min} = 0,1$ a $c = 0,999$.

Tab. 9: Použité sady hradel

sada 1	Γ = {AND, NAND, OR, NOR, XOR, XNOR, (not x) AND y}
sada 2	Γ = {AND, OR, XOR}
sada 3	Γ = {NAND, NOR, XNOR}
sada 4	Γ = {AND, OR, (not x) AND y}

Hradlo *(not x) AND y* není typické sériově vyráběné hradlo, proto je v prezentovaných výsledcích počítáno jako dvě hradla (NOT, AND).

Sady hradel byly zvoleny s přihlédnutím k typům hradel používaných při konvenčním návrhu a na základě experimentů provedených v [21].

6.1 Výsledky 2x2 násobičky

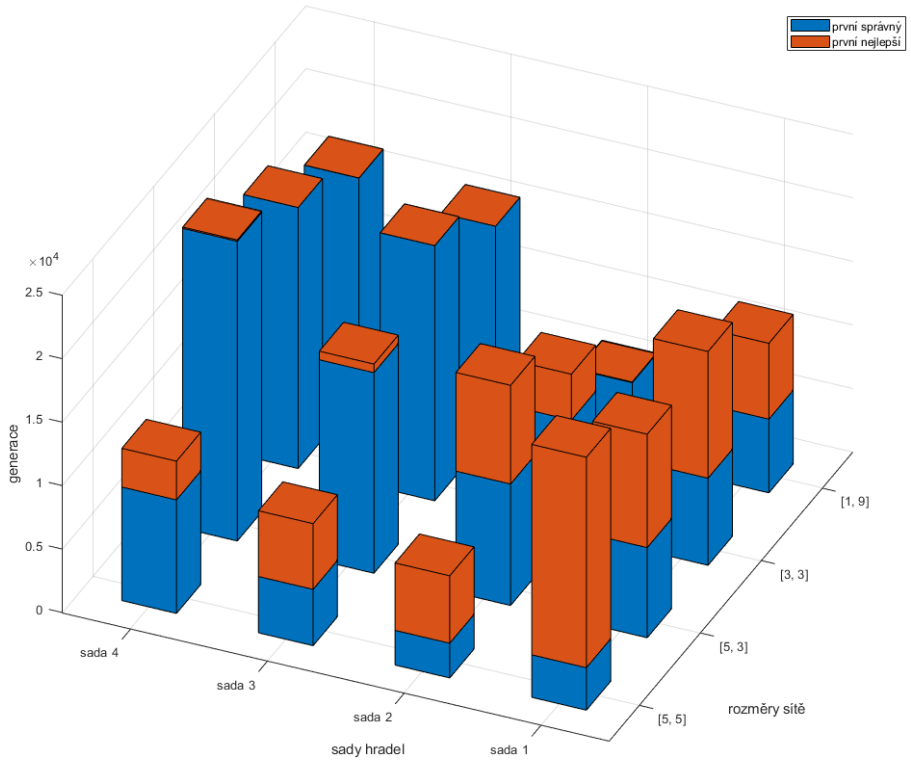
Nejjednodušším obvodem, na kterém bylo provedeno testování, je 2x2 násobička. Očekávané výsledky podle předchozích experimentů jsou obvody se 7 hradly a zpožděním 2T. Z tohoto důvodu byly zvoleny následující rozměry CGP mřížky.

1. **1x9** - $n_r = 1, n_c = 9$
2. **3x3** - $n_r = 3, n_c = 3$
3. **5x3** - $n_r = 5, n_c = 3$
4. **5x5** - $n_r = 5, n_c = 5$

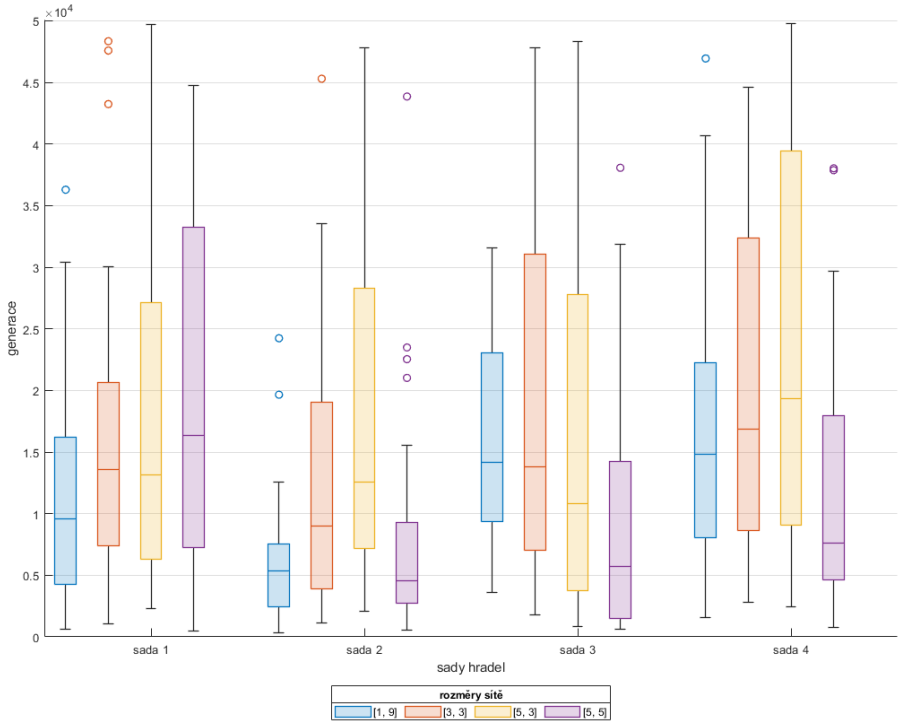
Bylo provedeno 30 evolučních běhů pro všechny sady hradel. Evoluce byla ukončena po 50 000 generacích. V Tab. 10 můžeme vidět procentuální úspěšnost dosažení správného výsledku, tedy robustnost CGP pro úlohu 2x2 násobičky. Tyto výsledky byly dále zpracovány pro grafy na Obr. 10 a 11.

Tab. 10: Úspěšnost dosažení správného výsledku 2x2 násobičky

	sada 1	sada 2	sada 3	sada 4
[1, 9]	90.0	93.3	66.7	90.0
[3, 3]	80.0	86.7	56.7	53.3
[5, 3]	100.0	100.0	86.7	90.0
[5, 5]	100.0	100.0	100.0	100.0



Obr. 10: Průměrný počet generací pro dosažení správného a nejlepšího výsledku při vývoji 2x2 násobičky



Obr. 11: Krabicový graf nejlepších nalezených řešení pro vývoj 2x2 násobičky

V grafu na Obr. 10 můžeme vidět srovnání nastavení z hlediska dosažení správných a nejlepších výsledků. Obr. 11 potom představuje krabicový graf popisující statistiku nejlepších dosažených výsledků.

Můžeme si všimnout, že při použití sady hradel 1 po nalezení správného řešení probíhá hledání lepšího řešení déle než u ostatních sad hradel, protože prohledáváme větší stavový prostor.

Dále můžeme vidět, že při použití sady 2 jsou odchylky nalezení řešení výrazně menší než u ostatních nastavení. Nejspíše je to způsobeno tím, že sada hradel je k řešení problému vhodná a hledání optima je tak efektivnější. To potvrzuje i úspěšnost nalezení správných řešení (Tab. 10).

Tabulky 11 a 12 nám potom ukazují nejlepší a průměrné výsledky.

Tab. 11: Nejlepší výsledky pro 2x2 násobičku

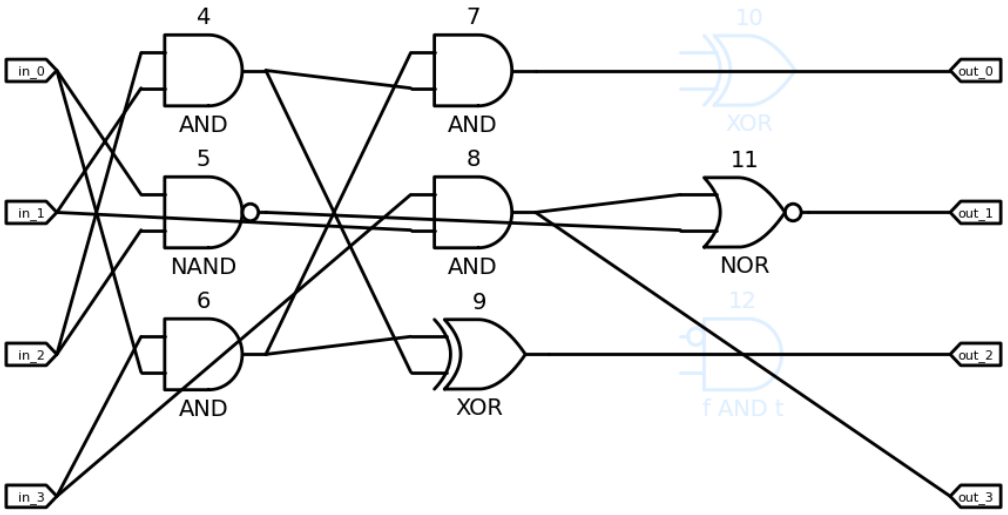
	sada 1			sada 2			sada 3			sada 4		
	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.
[1, 9]	7	37	2	7	38	3	9	57	3	10	40	3
[3, 3]	7	37	2	7	38	3	9	57	3	10	40	3
[5, 3]	7	37	2	7	38	3	9	57	3	10	40	3
[5, 5]	7	37	2	7	38	3	9	57	3	10	40	3

Tab. 12: Průměrné výsledky pro 2x2 násobičku

	sada 1			sada 2			sada 3			sada 4		
	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.
[1, 9]	8.0	40.4	2.8	7.1	38.6	3.0	9.0	57.0	3.5	10.1	40.3	3.1
[3, 3]	7.8	40.0	2.2	7.0	38.2	3.0	9.0	57.0	3.0	10.0	40.0	3.0
[5, 3]	7.9	39.7	2.5	7.0	38.0	3.0	9.1	57.7	3.0	10.4	41.5	3.0
[5, 5]	7.8	39.7	2.6	7.0	38.0	3.1	9.0	57.0	3.3	10.3	41.3	3.1

Vidíme, že sady 1 a 2 mají srovnatelné výsledky. U sad 3 a 4 jsou výsledky horší ve všech metrikách. To je způsobeno nevhodností sady hradel k vykonávání požadované operace. Pro takto jednoduchý problém nevidíme velké rozdíly mezi rozměry mřížky.

Jako nejlepší 2x2 násobička byl vybrán jedinec z evolučního běhu se sadou 1 v mřížce o rozměrech 3x3 (Obr. 12).



Obr. 12: Nejlepší nalezená 2x2 násobička (7 hradel, 37 tranzistorů, zpoždění 2T)

6.2 Výsledky 3x2 násobičky

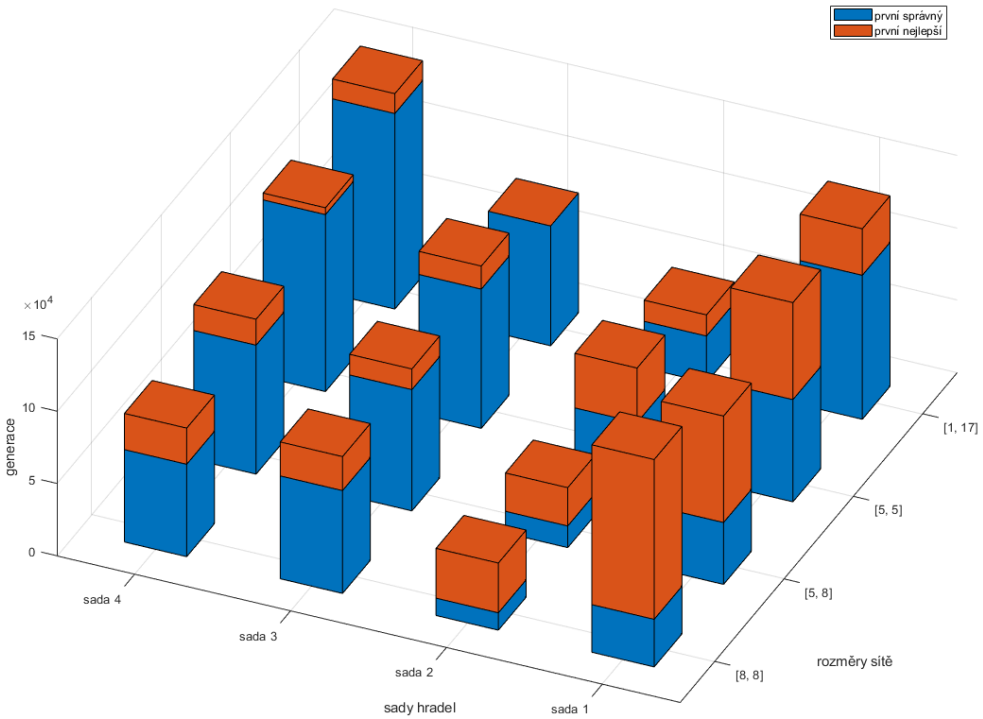
3x2 násobička provádí násobení dvou binárních čísel o třech a dvou bitech. Pro tento problém je očekávaný počet hradel 13 a zpoždění 3T, proto byly zvoleny následující rozměry.

1. **1x17** - $n_r = 1, n_c = 17$
2. **5x5** - $n_r = 5, n_c = 5$
3. **5x8** - $n_r = 5, n_c = 8$
4. **8x8** - $n_r = 8, n_c = 8$

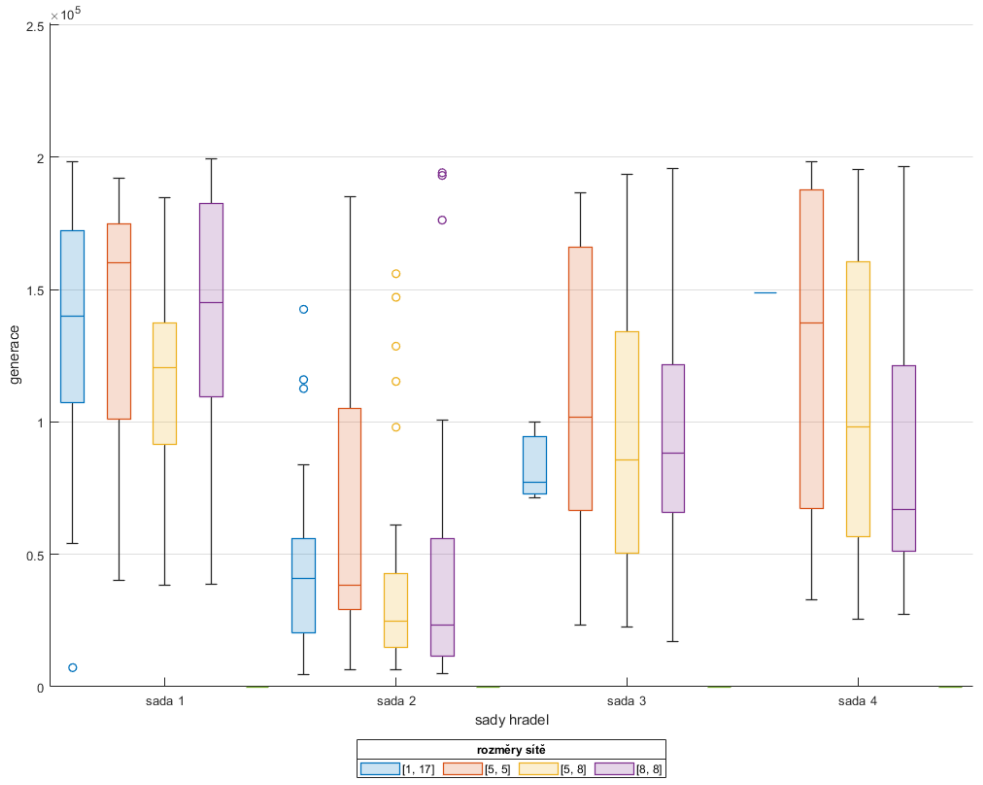
Protože tento problém není příliš časově náročný, byly využity všechny sady hradel. Všechny běhy byly zastaveny po 250 000 generacích.

Tab. 13: Úspěšnost dosažení správného výsledku 3x2 násobičky

	sada 1	sada 2	sada 3	sada 4
[1, 17]	76.7	100.0	10.0	3.3
[5, 5]	83.3	100.0	56.7	40.0
[5, 8]	100.0	100.0	86.7	80.0
[8, 8]	100.0	100.0	100.0	96.7



Obr. 13: Průměrný počet generací pro dosažení správného a nejlepšího výsledku při vývoji 3x2 násobičky



Obr. 14: Krabicový graf nejlepších nalezených řešení pro vývoj 3x2 násobičky

V grafu na Obr. 13 vidíme evidentní trend vyšší obtížnosti hledání řešení pro menší rozměry mřížky. Obecně pro větší mřížky bylo hledání správného výsledku méně efektivní. Důvodem je menší počet redundantních genů, a tedy méně efektivní prohledávání prostoru řešení.

V Tab. 13 můžeme vidět, že pro sady 3 a 4 bylo obtížné nalézt správné řešení. Pro sadu 4 a rozměry mřížky 1x17 bylo nalezeno pouze jedno správné řešení. Sada 2 měla nejlepší výsledky z hlediska rychlosti nalezení řešení. Zároveň vidíme, že pro sadu 1 bylo hledání výsledků složitější. To je způsobeno ztížením hledání ve větším stavovém prostoru (více druhů hradel).

Tabulky 14 a 15 opět ukazují nejlepší a průměrné výsledky.

Tab. 14: Nejlepší výsledky pro 3x2 násobičku

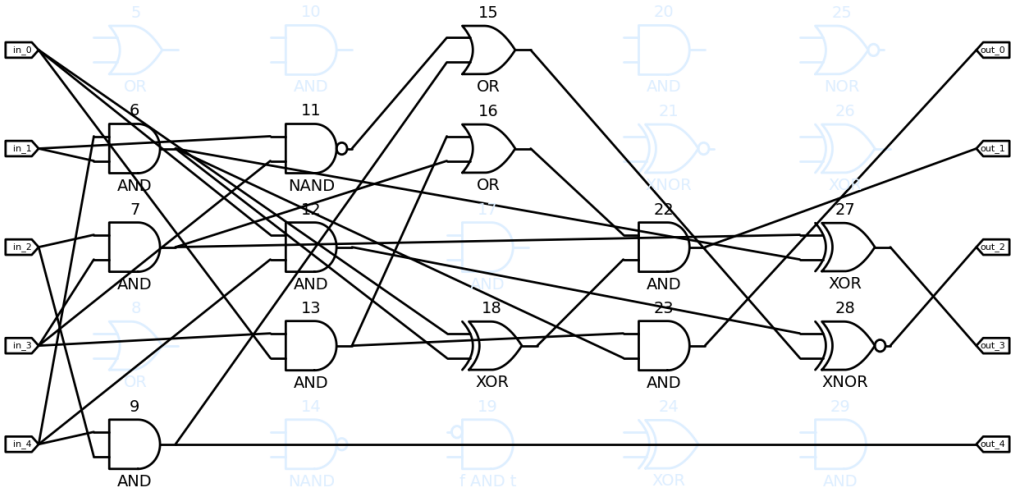
	sada 1			sada 2			sada 3			sada 4		
	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.
[1, 17]	13	71	3	13	72	3	15	99	4	20	80	5
[5, 5]	13	69	3	13	72	3	15	99	4	20	80	4
[5, 8]	13	69	3	13	72	4	15	99	4	20	80	4
[8, 8]	13	68	3	13	72	3	15	99	4	20	80	4

Tab. 15: Průměrné výsledky pro 3x2 násobičku

	sada 1			sada 2			sada 3			sada 4		
	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.
[1, 17]	15.0	80.4	4.2	13.3	73.5	4.4	15.7	105.0	4.3	20.0	80.0	5.0
[5, 5]	13.6	73.9	3.6	13.2	72.8	3.9	15.7	104.5	4.2	20.4	81.7	4.3
[5, 8]	14.2	77.2	4.0	13.5	74.3	4.2	16.1	106.6	4.4	20.9	83.7	4.6
[8, 8]	13.9	75.5	4.0	13.6	74.7	4.2	16.2	107.5	4.7	20.9	83.7	4.7

Z průměrných výsledků vidíme, že sada 2 nachází nejlepší výsledky ve všech nastaveních rozměrů mřížky. Sada 1 je však vhodnější pro minimalizaci tranzistorů, protože obsahuje „levnější“ hradla. Z výsledků dále vidíme, že sady 3 a 4 mají opět horší výsledky a nejsou tedy tak vhodné pro vývoj násobiček.

Nejlepší výsledek (Obr. 15) napříč nastaveními byl nalezen se sadou 1 v mřížce o rozměrech 5x5.



Obr. 15: Nejlepší nalezená 3x2 násobička (13 hradel, 69 tranzistorů, zpoždění 3T)

6.3 Výsledky 3x3 násobičky

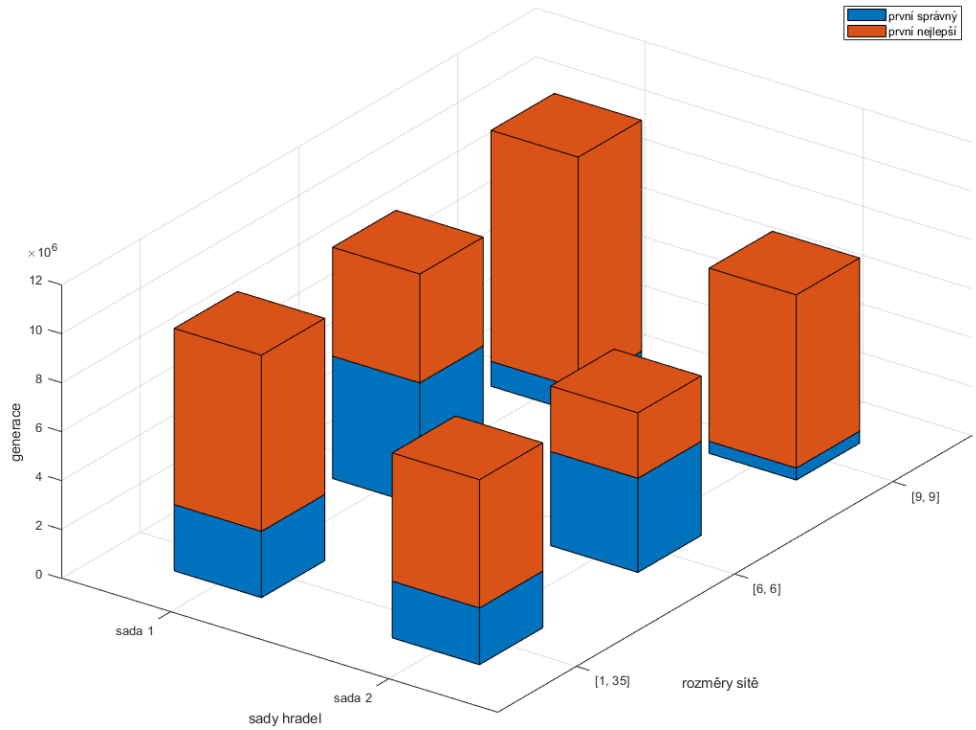
3x3 násobička mezi sebou násobí dvě tříbitová čísla a je výrazně složitějším obvodem v porovnání s 3x2 násobičkou. Z předchozích experimentů můžeme očekávat obvod o 27 hradlech a zpoždění 6T, případně lepší. Rozměry mřížky byly zvoleny s přihlédnutím k těmto očekáváním.

1. **1x35** - $n_r = 1, n_c = 35$
2. **6x6** - $n_r = 6, n_c = 6$
3. **9x9** - $n_r = 9, n_c = 9$

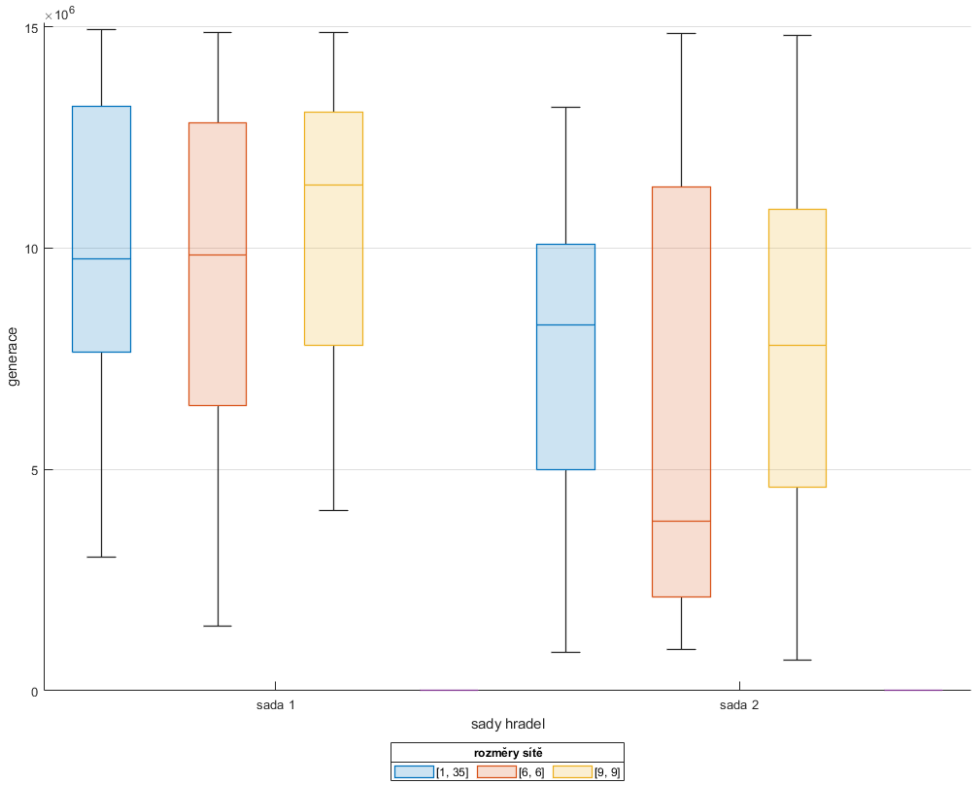
Na základě výsledků 3x2 a 2x2 násobiček byly zvoleny sady hradel s nejlepšími výsledky, a to sady 1 a 2. Vzhledem ke složitosti problému byl navýšen počet generací evolučního běhu na 15 000 000.

Tab. 16: Úspěšnost dosažení správného výsledku 3x3 násobičky

	sada 1	sada 2
[1, 35]	93.3	93.3
[6, 6]	76.7	86.7
[9, 9]	100.0	100.0



Obr. 16: Průměrný počet generací pro dosažení správného a nejlepšího výsledku při vývoji 3x3 násobičky



Obr. 17: Krabicový graf nejlepších nalezených řešení pro vývoj 3x3 násobičky

Graf na Obr. 16 ukazuje, že nejefektivnější nastavení v hledání správného řešení jsou opět mřížky větších rozměrů. Evoluční běhy používající sadu hradel 1 opět vykazují známky složitějšího prohledávání prostoru řešení. Pro síť 6x6 je patrná menší úspěšnost v nalezení správného řešení. Důvodem může být omezení zpoždění dané počtem sloupců CGP mřížky.

Tab. 17: Nejlepší výsledky pro 3x3 násobičku

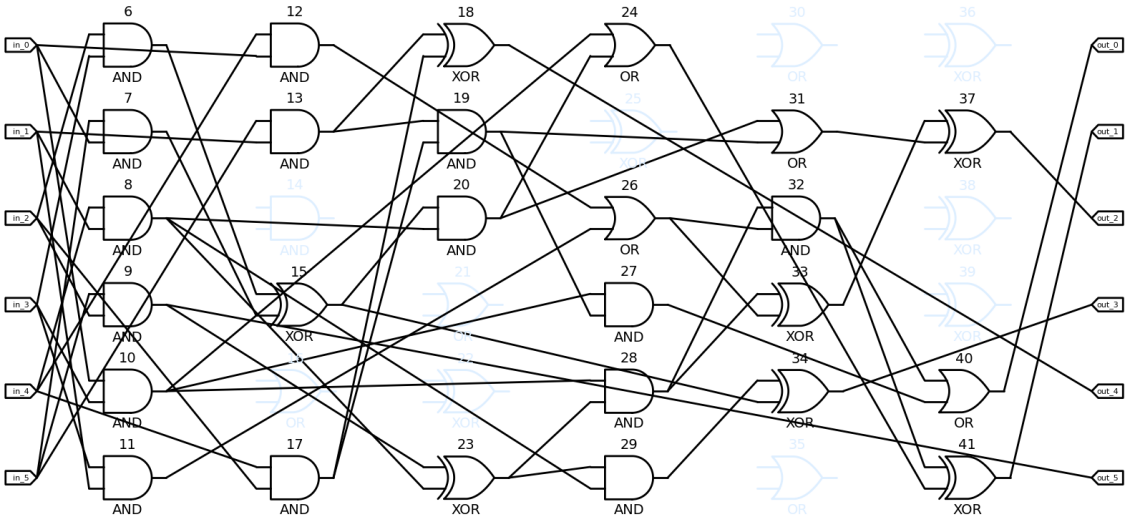
	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	24	135	6	25	138	6
[6, 6]	26	144	5	26	139	5
[9, 9]	26	136	6	26	139	5

Tab. 18: Průměrné výsledky pro 3x3 násobičku

	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	27.2	151.6	7.8	27.1	149.5	7.7
[6, 6]	28.4	155.1	5.8	27.4	148.8	5.8
[9, 9]	27.8	151.9	6.6	27.2	147.3	6.2

Z výsledků v tabulkách 17 a 18 vidíme, že sada 2 má lepší průměrné výsledky z hlediska počtu hradel i počtu tranzistorů. To je způsobeno větším stavovým prostorem a tedy složitějším prohledáváním.

Pomocí mřížky 1x35 byl nalezen jedinec s nejnižším počtem hradel. Tento obvod má však poměrně vysoké zpoždění 8T. Jako nejlepší obvod byl proto zvolen jedinec zobrazený na Obr. 18, nalezený v mřížce 6x6 se sadou hradel 2. Tento jedinec nemá sice nejnižší počet hradel, má přesto pouze o čtyři tranzistory více a výrazně lepší zpoždění.



Obr. 18: Nejlepší nalezená 3x3 násobička (26 hradel, 139 tranzistorů, zpoždění 5T)

6.4 Výsledky 5x4 poloviční sčítačky

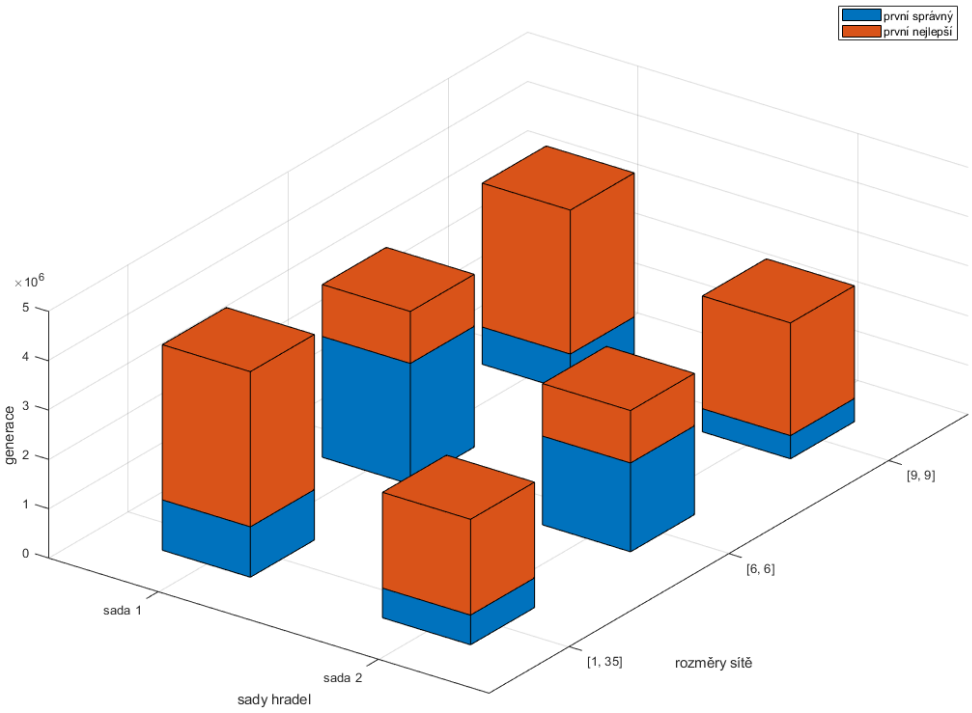
5x4 poloviční sčítačka provádí součet pětibitového a čtyřbitového čísla. Obvod má tedy 9 vstupů a 6 výstupů. Rozměry testovaných velikostí mřížky byly zvoleny na základě předem provedených experimentů.

1. **1x35** - $n_r = 1, n_c = 35$
2. **6x6** - $n_r = 6, n_c = 6$
3. **9x9** - $n_r = 9, n_c = 9$

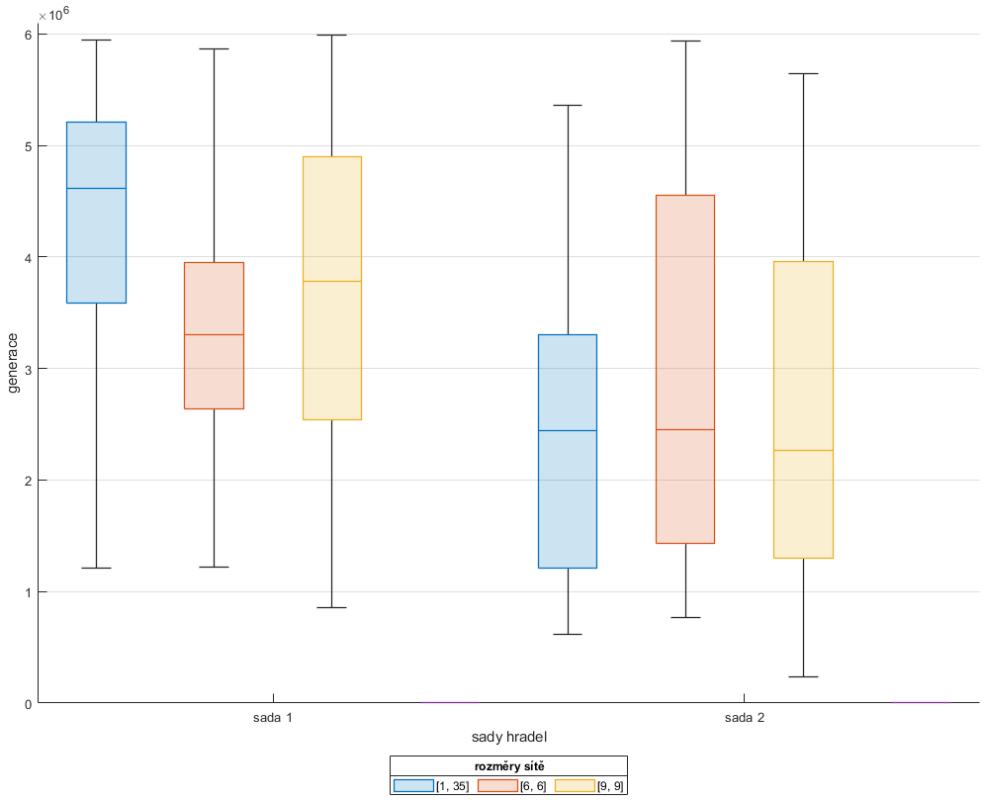
Hledání tohoto obvodu pro CGP není tolik složité jako 3x3 násobička, proto byl maximální počet generací stanoven na 6 000 000. Byly opět použity sady hradel 1 a 2.

Tab. 19: Úspěšnost dosažení správného výsledku 5x4 poloviční sčítačky

	sada 1	sada 2
[1, 35]	96.7	100.0
[6, 6]	60.0	66.7
[9, 9]	100.0	100.0



Obr. 19: Průměrný počet generací pro dosažení správného a nejlepšího výsledku při vývoji 5x4 poloviční sčítačky



Obr. 20: Krabicový graf nejlepších nalezených řešení pro vývoj 5x4 poloviční sčítačky

V grafu na Obr. 19 se opět potvrzuje větší efektivnost nalezení správného řešení pro mřížky větších rozměrů. Výjimkou je síť 6x6, která výrazně omezuje zpoždění obvodu. Z toho důvodu bylo prohledávání složitější a průměrné výsledky (z hlediska počtu hradel a tranzistorů) horší. Důsledkem omezení zpoždění je tedy větší počet hradel v obvodu.

Tab. 20: Nejlepší výsledky pro 5x4 poloviční sčítačku

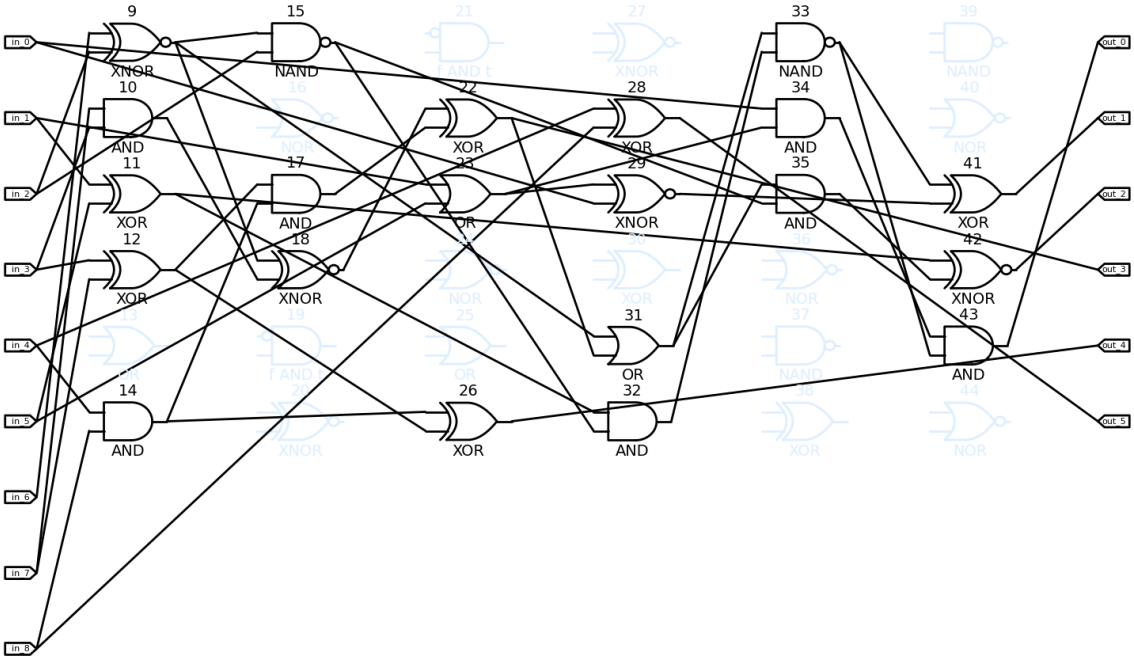
	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	19	120	8	19	116	8
[6, 6]	21	136	6	23	132	6
[9, 9]	20	122	6	19	116	7

Tab. 21: Průměrné výsledky pro 5x4 poloviční sčítačku

	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	19.9	132.9	9.3	19.2	118.5	8.5
[6, 6]	23.8	151.6	6.0	24.4	142.4	6.0
[9, 9]	21.8	142.3	7.2	20.9	126.0	7.3

Tabulky 20 a 21 ukazují, že nejlepší výsledky z hlediska počtu hradel byly nalezeny v mřížce 1x35. V mřížce 6x6 měly nalezené obvody větší počet hradel ale omezené zpoždění. Můžeme tedy říci, že minimalizace hradel byla provedena na úkor zpoždění.

Obvod s nejnižším nalezeným počtem hradel má 19 hradel a zpoždění 9T. Z důvodu velkého zpoždění není nejvhodnějším kandidátem. Vybrán byl proto obvod na Obr. 21, nalezený pomocí mřížky 6x6 a sady hradel 1.



Obr. 21: Nejlepší nalezená 5x4 poloviční sčítačka (21 hradel, 138 tranzistorů, zpoždění 6T)

6.5 Výsledky 4x4 úplné sčítačky

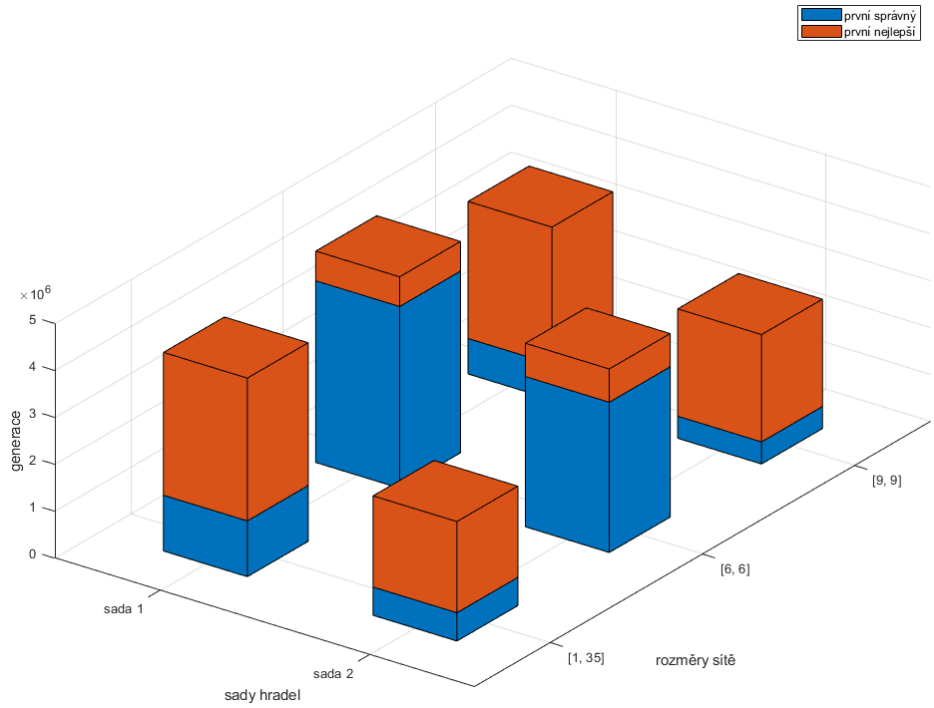
4x4 sčítačka provádí součet dvou čtyřbitových čísel s jedním *carry-in* bitem. Obvod má tedy 9 vstupů a 5 výstupů. Jako referenci pro očekávané výsledky můžeme použít integrovaný obvod *74283*, který má 36 hradel a zpoždění 4T. Je nutné si ale uvědomit, že tento obvod využívá vícevstupová hradla a proto srovnání není úplně ideální. Byly tedy provedeny experimenty, na základě kterých byly stanoveny následující rozměry mřížky:

1. **1x35** - $n_r = 1, n_c = 35$
2. **6x6** - $n_r = 6, n_c = 6$
3. **8x8** - $n_r = 8, n_c = 8$

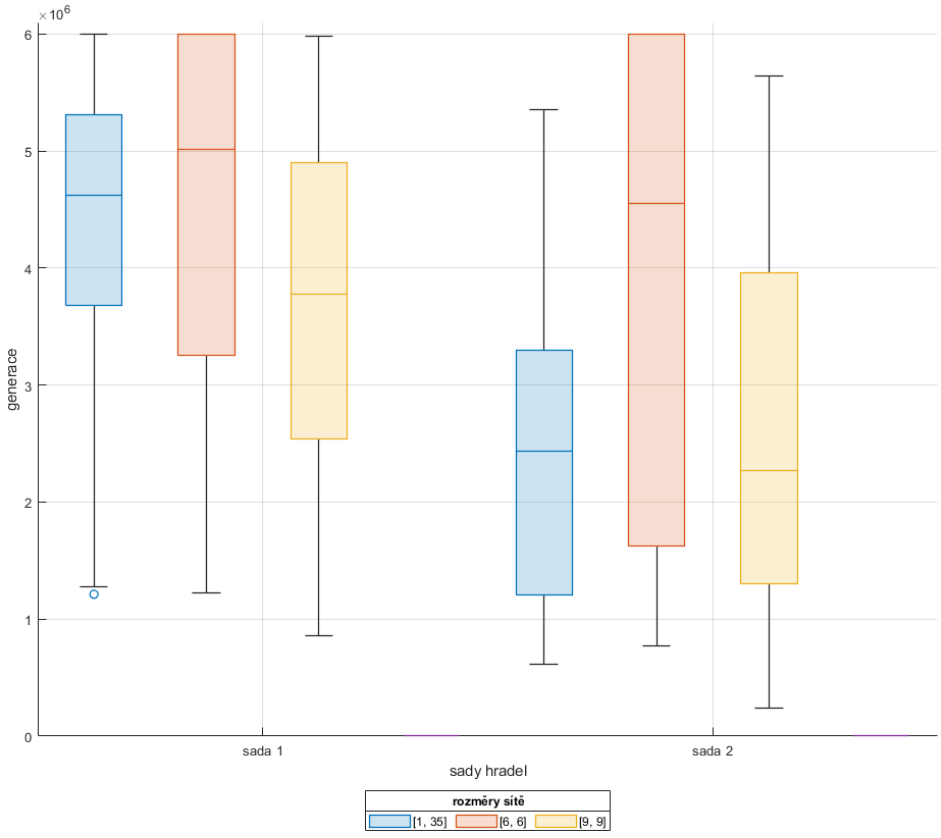
Jako maximální počet generací pro tento výpočet byl stanoven na 5 000 000 a opět byly použity sady hradel 1 a 2.

Tab. 22: Úspěšnost dosažení správného výsledku 4x4 úplné sčítačky

	sada 1	sada 2
[1, 35]	100.0	96.7
[6, 6]	20.0	66.7
[8, 8]	93.3	100.0



Obr. 22: Průměrný počet generací pro dosažení správného a nejlepšího výsledku při vývoji 4x4 úplné sčítačky



Obr. 23: Krabicový graf nejlepších nalezených řešení pro vývoj 4x4 úplné sčítačky

V grafu na Obr. 22 můžeme vidět podobné výsledky jako pro obvod v kap. 6.4. Omezení zpoždění opět komplikuje hledání správného řešení a menší stavový prostor sady 2 opět způsobuje efektivnější hledání řešení.

Tab. 23: Nejlepší výsledky pro 4x4 úplnou sčítačku

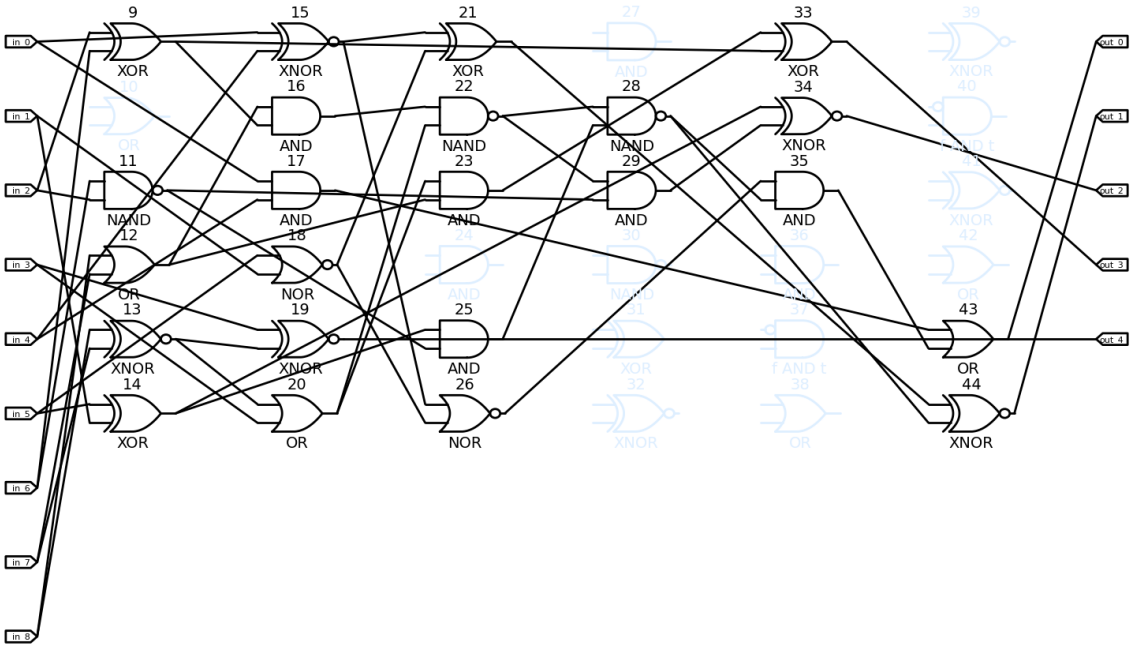
	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	20	120	8	20	120	8
[6, 6]	23	147	6	25	140	6
[8, 8]	21	136	6	22	128	7

Tab. 24: Průměrné výsledky pro 4x4 úplnou sčítačku

	sada 1			sada 2		
	hr.	tr.	zp.	hr.	tr.	zp.
[1, 35]	21.0	142.3	10.4	20.3	129.7	9.3
[6, 6]	25.3	158.3	6.0	26.0	146.2	6.0
[8, 8]	24.1	162.5	7.4	23.6	141.9	7.5

V tabulkách 23 a 24 můžeme vidět, že nejlepší průměrné výsledky vykazuje sada hradel 2. S použitím sady hradel 1 bylo však nalezeno několik obvodů s menším počtem hradel.

Jako nejlepší obvod byl zvolen jedinec nalezený sadou 1 v mřížce 6x6. Ve srovnání s integrovaný obvodem *74283* má tento obvod výrazně menší počet hradel, ale větší zpoždění.



Obr. 24: Nejlepší nalezená 4x4 úplná sčítačka (23 hradel, 147 tranzistorů, zpoždění 6T)

7 Závěr

V teoretické části práce byl proveden popis problému návrhu digitálních kombinačních obvodů. Byly popsány některé tradiční metody návrhu kombinačních obvodů a omezení tradičních způsobů jejich optimalizace. Dále byly stručně popsány heuristické a meta-heuristické optimalizační metody a jejich vhodné použití při hledání globálního optima analyticky složitě popsatelných problémů. Zvláštní pozornost byla věnována kartézskému genetickému programování jako vhodnému způsobu pro optimalizaci kombinačních obvodů. Tento způsob genetického programování byl použit pro praktickou část této práce.

Kartézské genetické programování bylo implementováno v jazyce *Python* pro vývoj kombinačních obvodů. Výsledky na testovacích příkladech byly prezentovány v kapitole 6.

Implementace evolučního algoritmu se od přístupů popsaných v [23] liší zejména v přístupu k selekci rodiče, kde byl zvolen přístup inspirovaný simulovaným žíháním. Tato selekce měla porovnatelné výsledky s výsledky z článku [20], kde byla popsána selekční metoda podobného smyslu.

Dalším rozdílem od tradičních způsobů optimalizace kombinačních obvodů pomocí CGP je volba minimalizace počtu tranzistorů místo počtu hradel. Bylo ukázáno, že obvody získané tímto způsobem jsou v počtu hradel ekvivalentní obvodům vyvinutým v [20]. Výhodou je vyšší míra optimalizace po nalezení nízkého počtu hradel. V průběhu evoluce byly nalezeny obvody s ekvivalentním počtem hradel, ale různým počtem tranzistorů.

Testovací úlohy byly provedeny pro různé kombinace sad hradel a rozměrů CGP mřížky. Tyto parametry mají zásadní vliv na strukturu nalezených obvodů.

Bylo ukázáno, že některé sady hradel nejsou vhodné po vývoj konkrétních logických obvodů. Nejlepší výsledky měla často sada obsahující všechny druhy hradel. Nevýhodou však je větší prohledávaný prostor řešení, a tedy zvyšující se počet generací potřebných pro dosažení dobrých výsledků.

Dále bylo ukázáno, že rozměry mřížky mají zásadní vliv zejména na zpoždění nalezených obvodů. Pokud je tedy naším cílem najít obvod s konkrétním maximálním zpožděním, je vhodné zvolit počet sloupů CGP mřížky ekvivalentní požadovanému zpoždění.

Srovnání výsledků pro dané úlohy vzhledem k jejich parametrizaci mřížkou a sadou je informativní, přesto zřejmé. Medián řešení v box grafech ukazuje dominanci sady 2 a konkrétních konfigurací mřížky, což je dobrým vodítkem pro další úvahy a rozšířeni. Rigorózní srovnání statistickým testem nebylo vzhledem k malému statistickému souboru a předpokládané nenormalitě rozdělení provedeno, což je další možnost pokračovaní této práce.

Jednodušší nalezené testovací obvody jsou porovnatelné s výsledky v [20] a [21]. Pro 3x3 násobičku se výsledky zvoleného přístupu od těchto experimentů liší. Jelikož primární optimalizace v této práci sloužila k minimalizaci tranzistorů, obvody nalezené tímto přístupem mohou mít více hradel při srovnatelném počtu tranzistorů. Srovnání výsledků z hlediska zpoždění a počtu hradel vidíme v Tab. 25.

Tab. 25: Srovnání dosažených výsledků s [20, 21]

	2x2 násobička			3x2 násobička			3x3 násobička		
	hr.	tr.	zp.	hr.	tr.	zp.	hr.	tr.	zp.
nejlepší zpoždění	7	37	2	13	69	3	26	139	5
nejlepší počet hradel	7	37	2	13	68	4	24	135	8
Petrlík, J., 2011 [21]	7	35	2	13	66	3	23	133	5
Sekanina, L., 2010 [20]	7	-	-	13	-	-	23	-	-

Pro nalezené obvody sčítaček (kap. 6.4 a 6.5) není vhodného srovnání kromě obvodu *74283* (Obr. 4) pro 4x4 úplnou sčítačku. Nalezený obvod má výrazně menší počet hradel i tranzistorů, ale delší nejdelší hradlovou cestu. Obvod *74283* však používá vícevstupových hradel a srovnání je tedy složitější. Nejdelší hradlová cesta v tomto obvodu prochází čtyřmi hradly, ale zpoždění těchto vícevstupových hradel je vyšší než u hradel dvouvstupových.

Seznam použité literatury

[1] Sekanina, L.: *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Praha : Academia, 2009, ISBN 978-80-200-1729-1.

[2] Wakerly, J. F.: *Digital Design: principals practices*. Prentice Hall, 2000, ISBN 0-13-769191-2.

[3] Švarc, I.; Matoušek, R.; Šeda, M.; aj.: *Automatické řízení*. Brno: Akademické nakladatelství CERM, 2011, ISBN 978-80-214-4398-3.

[4] Petley, G.: *VLSI and ASIC Technology Standard Cell Library Design*. [online]. URL `http://www.vlsitechnology.org/index.html`

[5] Sutherland, I.; Sproul, B.; Harris, D.: *Logical Effort: Designing Fast CMOS Circuits*. Academic Press, 1999, ISBN 978-00-805-1043-9.

[6] Vašíček, Z.; Sekanina, L.: *Evoluční návrh kombinačních obvodů*. *Elektrorevue - www.elektrorevue.cz*, ročník 2004, č. 43, 2004: s. 1–6, ISSN 1213-1539.
URL `https://www.fit.vut.cz/research/publication/7539`

[7] Drábek, V.: *Výstavba počítačů*. PC-DIR, 1995, ISBN 80-214-0691-7.

[8] semiconductor, F.: *DM74LS283 4-Bit Binary Adder with Fast Carry*. 2000.
URL `http://eeshop.unl.edu/pdf/DM74LS283.pdf`

[9] Niedoba, P.: *Minimalizace Booleových funkcí pomocí Quineovy-McCluskeyovy metody*. 2010, vedoucí bakalářské práce prof. RNDR. Ladislav Skula, DrSc.

[10] Salhi, S.: *Heuristic Search: The Emerging Science of Problem Solving*. Springer, 2017, ISBN 978-3-319-49354-1.

[11] Dowsland, K. A.; Thompson, J. M.: *Simulated Annealing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-540-92910-9, s. 1623–1655, doi: 10.1007/978-3-540-92910-9_49.
URL `https://doi.org/10.1007/978-3-540-92910-9_49`

[12] Hynek, J.: *Genetické algoritmy a genetické programování*. Praha: Grada Publishing, 2008, ISBN 978-80-247-2695-3.

[13] Holland, J. H.: *Adaptation in Natural and Artificial Systems*. MIT Press, 1992, ISBN 978-0262581110.

[14] Koza, J. R.: *Genetic Programming: On the programming of computers by means of natural selection*. The MIT Press, 1992, ISBN 978-0-262-11170-6.

[15] Karakatic, S.; Podgorelec, V.; Hericko, M.: *Optimization of Combinational Logic Circuits with Genetic Programming. Elektronika ir Elektrotechnika*, ročník 19, č. 7, Sep. 2013: s. 86–89, doi:10.5755/j01.eee.19.7.5169.
URL `https://eejournal.ktu.lt/index.php/elt/article/view/5169`

[16] Miller, J. F.; Job, D.; Vassiliev, V. K.: *Principles in the Evolutionary Design of Digital Circuits—Part I. Genetic Programming and Evolvable Machines*, ročník 1, 04 2000: s. 7 – 35, doi:https://doi.org/10.1023/A:1010016313373.

[17] Kalganova, T.; Miller, J.: *Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness.* In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, 1999, s. 54–63, doi:10.1109/EH.1999.785435.

[18] Miller, J.: *Cartesian genetic programming.* New York: Springer, 2011, ISBN 978-364-2173-103.

[19] Sekanina, L.; Gajda, Z.: *On the Practical Limits of the Evolutionary Digital Filter Design at the Gate Level.* In *Applications of Evolutionary Computing*, editace F. Rothlauf; J. Branke; S. Cagnoni; E. Costa; C. Cotta; R. Drechsler; E. Lutton; P. Machado; J. H. Moore; J. Romero; G. D. Smith; G. Squillero; H. Takagi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-33238-1, s. 344–355.

[20] Sekanina, L.; Gajda, Z.: *An Efficient Selection Strategy for Digital Circuit Evolution.* In *Evolvable Systems: From Biology to Hardware*, editace G. Tempesti; A. M. Tyrrell; J. F. Miller, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-15323-5, s. 13–24.

[21] Petrlík, J.: *Multikriteriální kartézské genetické programování.* 2011, vedoucí bakalářské práce Doc. Ing. Lukáš Sekanina, Ph.D.

[22] Miller, J.: *PPSN 2014 Tutorial: Cartesian Genetic Programming.* [online], 2014.
URL `http://ppsn2014.ijs.si/files/slides/ppsn2014-tutorial3-miller.pdf`

[23] Sekanina, L.: *Evolvable components: From theory to hardware implementations.* Springer, 2004, ISBN 3-540-40377-9.

[24] Miller, J.; Smith, S.: *Redundancy and computational efficiency in Cartesian genetic programming. Evolutionary Computation, IEEE Transactions on*, ročník 10, 05 2006: s. 167 – 174, doi:10.1109/TEVC.2006.871253.

[25] Vašíček, Z.: *Nástroje pro kartézské genetické programování*. [online].
URL http://www.fit.vutbr.cz/~vasicek/cgp/tools/

[26] Delker, C. J.: *Schemdraw documentation*. 2021.
URL https://schemdraw.readthedocs.io/en/latest/index.html

[27] Harris, C. R.; Millman, K. J.; van der Walt, S. J.; aj.: *Array programming with NumPy*. *Nature*, ročník 585, č. 7825, Září 2020: s. 357–362, doi:10.1038/s41586-020-2649-2.
URL https://doi.org/10.1038/s41586-020-2649-2

[28] *LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, New York, NY, USA: Association for Computing Machinery, 2015, ISBN 9781450340052.

Přílohy

- hlavní program
- skripty pro generování pravdivostních tabulek
- skripty spouštění evolučních běhů
- soubory evolučních běhů