VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

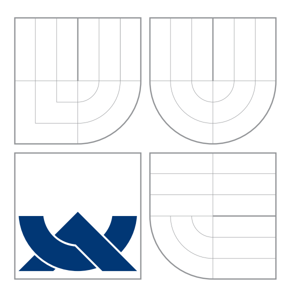
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

UŽITÍ GENETICKÉHO PROGRAMOVÁNÍ V NÁVRHU DIGITÁLNÍCH OBVODŮ

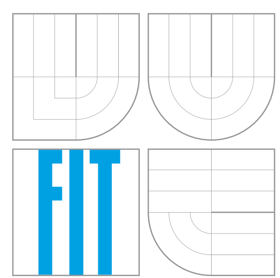
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL HEJTMÁNEK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

UŽITÍ GENETICKÉHO PROGRAMOVÁNÍ V NÁVRHU DIGITÁLNÍCH OBVODŮ

GENETIC PROGRAMMING FOR DESIGN OF DIGITAL CIRCUITS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL HEJTMÁNEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYŠEK GAJDA
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2007/2008

Zadání diplomové práce

Řešitel	**Michal Hejtmánek, Bc.**
Obor	Počítačové systémy a sítě
Téma	**Užití genetického programování v návrhu digitálních obvodů**
Kategorie	Počítačová architektura

Pokyny:

1. Prostudujte problematiku evolučního návrhu obvodů.
2. Seznamte se s technikou šíření schémat v evolučním algoritmu.
3. Navrhněte algoritmus genetického programování pro návrh obvodů.
4. Implementujte navržený algoritmus.
5. Navržený algoritmus demostrujte na netriviálních úlohách.
6. Srovnejte navrženou metodu návrhu s existujícími metodami.
7. Zhodnoťte dosažené výsledky.

Literatura:

- Kvasnička, V., Pospíchal, J., Tiňo, P: Evolučné algoritmy, STU Bratislava, 2000.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese
`http://www.fit.vutbr.cz/info/szz/`

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí	**Gajda Zbyšek, Ing.**, UPSY FIT VUT
Datum zadání	24. září 2007
Datum odevzdání	19. května 2008

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt
Cílem této práce bylo nastudování evolučních algoritmů a jejich využití pro návrh digitálních obvodů. Především jsem se zaměřil na genetické programování a jeho rozdílný způsob zacházení se stavebními bloky ve srovnání s genetickým algoritmem.
Na základě těchto dvou přístupů jsem vytvořil a odzkoušel hybridní metodu návrhu obvodů. Tato metoda využívá šíření schemat podle genetického algoritmu pro problémy řešené genetickým programováním. U složitějších obvodů dosahuje vyšší úspěšnosti návrhu i rychlejší konvergence k řešení než obecný algoritmus genetického programování.

Klíčová slova
Evoluční algoritmy, genetické programování, kartézské genetické programování, genetický algoritmus, stavební bloky, schéma teorém, evoluční návrh, evoluční optimalizace.

Abstract
The goal of this work was the study of evolutionary algorithms and utilization of them for digital circuit design. Especially, a genetic programming and its different manipulation with building blocks is mentioned in contrast to a genetic algorithm.
On the basis of this approach, I created and tested a hybrid method of electronic circuit design. This method uses spread schemes according to the genetic algorithm for the pattern problems witch are solved by the genetic programming. The method is more successful and have faster convergence to a solution in difficult electronic circuits design than a common algorithm of the genetic programming.

Keywords
Evolutionary algorithm, genetic programming, cartesian genetic programming, genetic algorithm, building blocks, theorem of schemes, evolutionary design, evolutionary optimization.

Citace
Michal Hejtmánek: Užití genetického programování v návrhu digitálních obvodů, diplomová práce, Brno, FIT VUT v Brně, 2008

Užití genetického programování v návrhu digitálních obvodů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zbyška Gajdy a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Michal Hejtmánek
14. května 2008

Poděkování

Za odborné vedení a poskytnutí přes 5 000 stran studijních materiálů v anglickém jazyce děkuji vedoucímu své práce Ing. Zbyšku Gajdovi.

© Michal Hejtmánek, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Návrh digitálních obvodů **5**
2.1 Úrovně návrhu elektronických obvodů . . . 6

3 Biologická inspirace evolučních algoritmů **8**

4 Metody moderní optimalizace **10**

5 Evoluční algoritmy **11**
5.1 Druhy evolučních algoritmů . . . 12

6 Genetický algoritmus **13**
6.1 Kódování problému . . . 13
6.2 Inicializace populace . . . 13
6.3 Ohodnocení jedince . . . 13
6.4 Selekce . . . 14
6.5 Křížení . . . 14
6.6 Mutace . . . 16
6.7 Teorie stavebních bloků . . . 16
6.8 Nahrazovací strategie . . . 17
6.9 Podmínky ukončení . . . 17
6.10 Shrnutí genetického algoritmu . . . 17

7 Genetické programování **18**
7.1 Kódování problému . . . 19
7.2 Inicializace . . . 19
7.3 Mutace . . . 19
7.4 Křížení . . . 20
7.5 Kartézské genetické programování . . . 21

8 Návrh hybridního algoritmu **23**
8.1 Vliv mutace na funkci stromu . . . 23
8.2 Stavební bloky . . . 24
8.3 Dědivost znaků podobných rodičů . . . 25
8.4 Upravené křížení . . . 26
8.5 EMOEA . . . 27
8.6 Stárnutí rodů . . . 27

9 Implementace **28**
9.1 Zapojení a výpočet výstupních hodnot hradel . . . 28
9.2 Paměťová reprezentace obvodu . . . 29
9.3 Křížení . . . 30
9.4 Simulace křížení . . . 31
9.5 Výpočet fitness . . . 32
9.6 Evoluce obvodu . . . 32
9.7 Skládání obvodu . . . 33
9.8 Validace, podmíněný překlad a watchdog . . . 33

10 Experimenty **35**
10.1 Úspěšnost optimalizace . . . 35
10.1.1 Úplná sčítačka . . . 35
10.1.2 Násobička 2x2 bity . . . 37
10.1.3 Multiplexor 4 na 1 . . . 38
10.2 Úspěšnost návrhu . . . 40
10.2.1 Násobička 3x3 bity . . . 40
10.2.2 Násobička 3x4 bity . . . 41
10.2.3 Násobička 4x4 bity . . . 42
10.3 Shrnutí výsledků úspěšnosti návrhů . . . 43

11 Závěr **44**

Literatura **45**

Slovníček pojmů **47**

A Uživatelská příručka k programům **50**
A.1 Implementace obecného a hybridního algoritmu . . . 50
A.2 Příklad konfiguračního souboru ”input.h” . . . 52
A.3 TreeViewer . . . 53
A.4 Pyramid . . . 53

B Pravdivostní tabulky **54**

Kapitola 1

Úvod

Již přes 40 let platí *Moorův zákon*, který v původním znění říká, že počet součástek na čipu se každý rok zdvojnásobí při zachování stejné ceny [25]. S rostoucí složitostí výroby a návrhu čipů bylo toto empiricky ověřené pravidlo opraveno na „každé dva roky“ (někdy se uvádí průměr 18 měsíců) a předpokládá se, že v této podobě bude platit ještě nejméně do roku 2018.

Exponenciální růst počtu tranzistorů na čipu, jež letos překročil jednu miliardu, společně s jejich rostoucí integrací a zvyšující se pracovní frekvencí dělá návrh každé další generace čipů náročnější. Kromě požadavků na výkon, velikost, spotřebu a odvod tepla z čipu je při jeho návrhu potřeba zohlednit i omezené škálování napětí, pracovní frekvence, rychlost elektronu, výtěžnost výroby a při nejvyšší integraci i kvantové jevy.

Takovýto komplexní návrh již dávno nelze celý provádět „ručně“ nebo na úrovni tranzistorů či logických členů a tak vyžaduje množství specializovaných nástrojů a probíhá současně v celé hierarchii úrovní. Úspěšně se v něm používají evoluční algoritmy mezi které patří i genetické programování, ať už pro návrh samotných komponent nebo optimalizace jejich umístění na čipu [18]. Zároveň se také ukazuje, že evoluční algoritmy jsou zatím jedinou alternativou inženýrskému návrhu, která je schopna pracovat i s reálnými vlastnostmi daného materiálu či prostředí a dosáhnout tak výrazně lepších výsledků.

V současnosti se při návrhu zapojení digitálních obvodů dává přednost kartézskému genetickému programování a původní genetické programování je opomíjeno. Přitom tento univerzální algoritmus lze použít na širší množinu problémů a jeho potenciál je vidět i v množství existujících specializovaných variant. Proto jsem se rozhodl pokračovat v jeho vývoji a pokusit se posunout možnosti jeho použití i pro návrh zapojení digitálních obvodů jako alternativě kartézskému genetickému programování.

Úvodní a druhá kapitola se věnují inženýrskému návrhu digitálních obvodů a motivaci k jeho alternativě v podobě evolučního návrhu. Třetí kapitola předkládá stručný přehled vývoje teorií evoluční biologie, kterými jsou evoluční algoritmy inspirovány. Další kapitola uzavírá obecný úvod zasazením evolučních algoritmů do metod moderní optimalizace, na kterou již navazuje kapitola rozebírající jednotlivé druhy evolučních algoritmů a jejich využití pro evoluční návrh a evoluční optimalizaci.

Šestá kapitola popisuje základní prvky a fáze genetického algoritmu a přechází v popis šíření schémat a zacházení se stavebními bloky. Další kapitola navazuje obdobou genetického algoritmu – genetickým programováním a zaměřuje se na zásadní rozdíly těchto algoritmů, především v charakteru řešených problémů a přístupu ke stavebním blokům.

V osmé kapitole je na základě zjištěných rozdílů navržena hybridní metoda práce se stavebními bloky, společně s rozšířením o „stárnutí rodů“, které lze implementovat i do obecného algoritmu. Následující kapitola popisuje reprezentaci, modifikaci a evoluci obvodů v paměti, rozšířitelnou o „simulaci křížení“.

V desáté kapitole jsou implementované metody (a jejich rozšíření) otestovány při návrhu a optimalizaci digitálních obvodů a výsledky porovnány s alternativní metodou návrhu pomocí kartézského genetického programování. Závěrečná kapitola shrnuje dosažené výsledky a naznačuje směr dalšího vývoje.

Kapitola 2

Návrh digitálních obvodů

Digitální obvody pracují s binárními hodnotami signálů a dělí se na *sekvenční* a *kombinační* [18]. Zatím co výstup *kombinačních obvodů* je dán pouze aktuálním stavem na vstupu, výstup *sekvenčních obvodů* může být závislý i na předchozím stavu obvodu. Návrh *sekvenčních obvodů* je tak výrazně složitější, protože díky vlivu předešlého stavu na funkci a na následující stav se musí testovat nejen všechny možné vstupní kombinace, ale ještě ke každé takové kombinaci všechny možné kombinace předešlé.

Kombinační obvody, jejichž návrhu se budu v této práci věnovat, realizují určitou logickou funkci přímou transformací vstupních hodnot na výstupní hodnoty pomocí základních logických členů – hradel. Hradla jsou komponenty s jedním až n jedno-bitovými vstupy (v závislosti na funkci a typu) a jedním jedno-bitovým výstupem [12]. Označují se dle svých logických funkcí AND, OR, XOR, NOT, jež odpovídají anglickým názvům operací booleovské algebry (logický součin, součet, nonekvivalence a negace). Pravdivostní tabulkou — tedy definicí odezvy na všechny kombinace vstupních hodnot — si lze vytvořit libovolnou logickou funkci nebo hradlo (viz tabulka 2.1).

Tabulka 2.1: Pravdivostní tabulky základních logických členů a jejich značení (podle standardu ANSI/IEEE Std 91-1984) [22].

hradlo	**značka**	vstupy		výstup
		a	**b**	**F**
AND		0	0	0
		0	1	0
		1	0	0
		1	1	1
OR		0	0	0
		0	1	1
		1	0	1
		1	1	1
XOR		0	0	0
		0	1	1
		1	0	1
		1	1	0
NOT		0		1
		1		0

Konvenční metody návrhu logických obvodů obvykle postupují od popisu chování obvodu (zadaného pravdivostní tabulkou) ke strukturálnímu popisu (schématu nebo algebraickému výrazu), který již přímo definuje typ a zapojení logických komponent realizujících požadovanou funkci.

Typicky se postupuje mechanickým převodem pravdivostní tabulky na výraz *booleovy algebry v úplné normální disjunktní formě* (viz tabulka 2.2) a jeho minimalizací pomocí zákonů *booleovy algebry*, *Karnaughovy mapy* nebo *Quine-McCluskeyho metody* [21]. Dále je možno výraz převést do tvaru lépe vyhovujícího zvolené množině logických komponent. Například ve zmíněné *booleově algebře* lze pomocí elementárních logických funkcí: AND, OR, NOT vyjádřit libovolnou logickou funkci, ale její zápis může být mnohem delší, než třeba v ekvivalentní Reed-Mullerově algebře s elementárními funkcemi XOR, OR, NOT, jak je vidět v tabulce 2.2.

Tabulka 2.2: Příklad převodu logické funkce zadané pravdivostní tabulkou do výrazu úplné normální disjunktní formy (ÚNDF), Reed-Mullerovy algebry a schematického zobrazení výrazu.

vstupy			výstup	výraz v	výraz v	schematické
a	**b**	**c**	**S**	ÚNDF	Reed-Mullerově algebře	zapojení
0	0	0	0	S =		
0	0	1	1	$\overline{a}\overline{b}c$		
0	1	0	1	$+\overline{a}b\overline{c}$		
0	1	1	0		$S = a \oplus b \oplus c$	
1	0	0	1	$+a\,\overline{b}\overline{c}$		
1	0	1	0			
1	1	0	0			
1	1	1	1	$+a\,bc$		

Každá z algeber používá ke strukturálnímu popisu obvodu jinou množinu elementů pokrývající *úplný soubor logických funkcí* a v každé bude zápis obvodu vypadat jinak. Všechny logické funkce lze vyjádřit i použitím samotných elementů NAND nebo NOR, ale zápisy výrazů pak bývají mnohem delší.

2.1 Úrovně návrhu elektronických obvodů

Funkci navrhovaných obvodů můžeme *simulovat* pomocí softwarových simulátorů, nebo *emulovat* v univerzálních rekonfigurovatelných obvodech a to na čtyřech základních úrovních:

1. **Molekulární úroveň** neumožňuje kvalitní simulaci, protože jevy na kvantové úrovni nejsou ještě dostatečně popsány a prozkoumány.

   Pro emulaci vznikla platforma NanoCell [18], na které již byla pomocí evolučního algoritmu vytvořena triviální zapojení některých základních logických členů. Návrh složitějších obvodů na této úrovni není zatím možný. Evoluční algoritmus se jeví jedinou možností návrhu na této úrovni, protože pracuje s reálnými vlastnostmi i vadami materiálu.

2. **Úroveň tranzistorů** potřebuje pro simulace komplikované externí nástroje (například simulátor SPICE [12]) a výsledky jsou často závislé na jejich kvalitě.

Pro emulaci existují analogové rekonfigurovatelné obvody (FPAA, FPTA), ale zatím nejsou příliš rozšířeny.

3. **Úroveň logických členů**, na kterou se v této práci zaměřím, dává názorné výsledky simulace, ověřitelné na každém počítači a pro vývoj postačí pouhý programovací jazyk. Protože složitost evolučního návrhu s každým obvodovým vstupem exponenciálně roste, lze v současné době na této úrovni (při vyhodnocování všech kombinací vstupních hodnot) provádět přímý návrh obvodů s nanejvýš osmi obvodovými vstupy a osmi výstupy [18].

   Pro emulaci je dostupná široká paleta rekonfigurovatelných obvodů (CLB, FPGA) a návrh na této úrovni pomocí kartézského genetického programování je v poslední době velmi oblíben.

4. **Úroveň funkčních bloků** sestavuje při simulaci i emulaci obvod ze složitějších logických členů s více vstupy i výstupy (sčítačky, násobičky, registry a pod) [16]. Nedosahuje sice tak efektivních výsledků (co do rychlosti a velikosti) jako úrovně předcházející, zato je schopná návrhu i složitějších obvodů.

Pro dosažení lepších výsledků lze použít i technik *inkrementální evoluce* nebo *developementu.*

Kapitola 3

Biologická inspirace evolučních algoritmů

Evoluci chápeme jako proces postupného vývoje (z latiny *evolvere* – vyvíjet) [25]. V biologii evolucí rozumíme historii vývoje života.

Roku 1809 publikoval Jean Baptiste Lamarck [25] ve svém díle *Filosofie zoologie* (Philospophie zoologique) první ucelenou evoluční teorii postupného vývoje ze společného předka. Podle jeho představ se měly do další generace přenášet adaptace vznikající za života jedinců (*lamarckismus*). Tedy děti kováře by měly zdědit více svalové hmoty po svých předcích, kteří ji získali celoživotní tvrdou prací se železem.

V roce 1859 představil Charles Darwin [2] dodnes rozšířenou teorii *O původu druhů přirozeným výběrem* (The Origin of Species by Means of Natural Selection). V ní popsal a zdůvodnil, že živočišné a rostlinné druhy vznikly a stále vznikají v přírodě postupným vývojem ze společného předka. Taktéž vysvětlil, že hybnou silou tohoto rozrůzňování druhů je přirozený výběr upřednostňující zdatnější a lépe adaptované jedince. Klíčem pro vznik přirozeného výběru je nadprodukce jedinců v prostředí, vedoucí ke vzájemné konkurenci a boji o přežití. Díky němu mohou být v populaci upřednostněni jedinci s výhodnějšími dědičnými odchylkami (adaptacemi) na úkor ostatních. Oproti Lamarckovi Darwin nepřisuzuje vznik účelných vlastností u organismů jejich snaze se přizpůsobit prostředí, ale snaze prostředí upřednostňovat lépe přizpůsobené organismy.

Na prosazení *darwinismu* měl zásluhu i otec genetiky, brněnský rodák Johann Gregor Mendel [25, 2], důkazem o dědičnosti znaků v roce 1865, který byl jedním z chybějících článků Darwinovy teorie. Dokázal totiž, že znaky nevznikají za života jako adaptace na prostředí, ale dědí se podle zákonů (*Mendelovy zákony genetiky*) do podoby vloh po rodičích, které se pak působením prostředí za života projevují. Na to, že ony dědičné znaky jsou uloženy v genech, jejichž nositelkou je DNA, se však přišlo až téměř o století později v roce 1953.

Díky tomu mohl v roce 1976 Richard Dawkins [2, 3] publikovat *Teorii sobeckého genu* (The Selfish Gene), která vysvětlila vznik a fixaci některých *altruistických modelů chování* pozorovaných ve volné přírodě. Sobeckého jedince z Darwinova světa, soupeřícího s ostatními jedinci vlastního druhu o přežití, by ohled na prospěch druhého zbytečně znevýhodňoval. Dawkinsova teorie však přenáší břemeno evoluce z jedinců na geny. Genům nejde ani tak o přežití a rozšíření jednoho nositele, jako spíše o přežití a rozšíření do co největšího počtu nositelů. Pro přežití genů je výhodné vést jedince k obětavosti pro záchranu jeho sourozenců, kteří jsou z velké části nositeli stejných genů.

Dnes se postupně prosazuje *Teorie zamrzlé plasticity* Jaroslava Flegra [2], podle níž je šíření genu ve velké populaci podmíněno především jeho schopností spolupráce s ostatními geny (*epistatické interakce*). Jinak je tomu ale v malé populaci s nižší genovou variabilitou, kde přežívají především geny zodpovědné za vyšší biologickou zdatnost jedince. K takové situaci dochází především při vzniku nového druhu, oddělením od původního (*peripatrická speciace*). Vytvoří se charakteristické znaky druhu a ty jsou již v čase expanze a růstu populace až do vymření druhu téměř neměnné (zamrzlé). K masovému rozšíření druhu dojde až po jeho dostatečném vyvinutí, čímž lze vysvětlit chybějící vývojové mezičlánky v paleontologických záznamech.

Stručným výtahem z „evoluce“ evoluční biologie jsem se snažil naznačit, že žádná z prezentovaných teorií není absolutně správná, ale ani absolutně špatná. V každé jsou patrny mechanismy předešlé, jen mají obvykle jiné místo a význam. Výsledkem tohoto vývoje je stále věrnější představa přírodních procesů.

Kapitola 4

Metody moderní optimalizace

Optimalizace je matematická disciplína, ve které hledáme takové řešení x z množiny všech stavů prostoru M (dále jen stavového prostoru), pro které účelová funkce $f(x)$, definující kvalitu tohoto řešení, dosáhne minimální (resp. maximální) hodnoty [26].

$$\min_{x \in M} f(x) \qquad (resp. \max_{x \in M} f(x)) \tag{4.1}$$

Vždy, když se algoritmicky snažíme najít řešení nějakého problému, jedná se o optimalizační problém, který můžeme obvykle řešit hned několika optimalizačními metodami [11, 8]. Ty můžeme rozdělit podle přístupu na tři základní typy:

Deterministické metody se zaměřují na hledání ve slibných oblastech (*exploitation*). Slušná řešení sice nacházejí velmi rychle, ale bohužel mohou *předčasně konvergovat* do *lokálního extrému* a uváznout tak v *klamném optimu*, ze kterého se ke *globálnímu* již nedostanou. Nalezení nejlepšího řešení tedy nezaručují. Patří sem například *metoda největšího spádu*, *metoda sdružených gradientů* nebo *kvazi-Newtonova metoda*.

Stochastické metody řeší problém *deterministických metod* přidáním náhodného prvku do hledání. Jejich snahou je naopak zaručit nalezení nejlepšího řešení prohledáním co největší části stavového prostoru (*exploration*). To ale ve velkém stavovém prostoru může trvat neúnosně dlouhou dobu. Patří sem *metoda náhodného prohledávání*, *simulovaného žíhání* nebo *horolezecký algoritmus*.

Evoluční algoritmy (EA) jsou univerzálním kompromisem obou těchto přístupů. Řadí se sice do *stochastických metod* [18], ale často bývají uvedeny odděleně, protože jsou inspirovány biologií a na rozdíl od ostatních *stochastických metod* nerozvíjí jen jedno řešení, ale pracují hned s několika současně. Podrobnějšímu popisu těchto algoritmů se budu věnovat v následujících kapitolách.

Každá metoda je vhodnější pro řešení jiného typu problému[1]. EA však můžeme použít jako univerzální algoritmus. Míru zaměření na slibné oblasti (*exploration vs. exploitation*), můžeme nastavit přímo pro daný problém velikostí populace, která určuje počet současně rozvíjených řešení. Malá znamená riziko uváznutí v *lokálním extrému* (viz *deterministické metody*), velká zbytečně dlouhou dobu hledání (viz *stochastické metody*).

[1] *No Free Lunch Theorems* [28] – Čím efektivněji metoda prohledává určitý charakter *stavového prostoru*, tím hůře bude prohledávat jiný, protože průměrný výkon každé metody je stejný.

Kapitola 5

Evoluční algoritmy

Evoluční algoritmy se po vzoru biologické evoluce snaží působením *selekčního tlaku* vyvinout (*evolvovat*) řešení zadaného problému. Začínají s populací (*množinou*) zpravidla náhodných řešení a vybírají z ní *kandidátní řešení*, která nejvíce vyhovují zadání. Z těch potom různými úpravami vytváří novou generaci, která pak nahradí méně vyhovující jedince. Tím v populaci dochází každou generací k množení lepších řešení na úkor horších, neboli uměle prováděný *Darwinův přirozený výběr*. Tento vývoj může být ukončen splněním *ukončovací podmínky*, kterou může být nalezení optimálního řešení, uplynutí nastaveného počtu generací, stagnace růstu kvality řešení v populaci apod. V zásadě se evoluční algoritmy využívají ke dvěma odlišným účelům [4]:

Evoluční optimalizace se aplikuje na doladění parametrů již hotového řešení. EA hledá konfiguraci takových hodnot těchto parametrů, při kterých bude celý systém pracovat co možná nejefektivněji. Známým příkladem je optimalizace spotřeby motorů pro Boeing 777 [15]. Lepším nastavením hodnot parametrů došlo ke zlepšení o 2,5% oproti původnímu inženýrskému návrhu. Výsledkem byla roční úspora na provozu v řádech milionů dolarů.

Evoluční návrh se oproti tomu pokouší vyvinout zcela vlastní nové řešení. Má k dispozici jen komponenty, pravidla jejich zapojování a výslednou funkci požadovaného systému. Definujeme tedy jen *co a s čím* má systém dělat a EA již navrhne *jakým způsobem.*

Škálovatelnost návrhu pomocí EA, tj. schopnost navrhovat funkční řešení podle různých požadavků, je obrovskou výhodou evolučního návrhu. Při návrhu elektronických obvodů tak mohou vznikat velmi komplikované obvody ze zcela libovolných typů součástek, jen těžko navrhnutelné člověkem.

Konvenční inženýrské metody návrhu digitálních obvodů takovéto různorodosti požadavků nevyhovují, protože jsou obvykle založeny na matematických modelech, které striktně vymezují typy a způsob použití hradel (viz kapitola 2). Například metody využívající *booleovské* logiky jsou k realizaci požadované funkce schopny využít jen hradel AND, OR, NOT nebo u *Reed-Mullerovy logiky* zase jen XOR, OR, NOT [18]. Evoluční návrh tedy pracuje pružněji s širší množinou řešení a je tak schopný inženýrská občas překonat.

Známým příkladem evolučního návrhu je anténa vyvinutá pro NASA [23]. Svými parametry dalece překonala používaná konvenční řešení a získala tak uznání *Human Competitive Result* v soutěži Humies 2004 [24].

5.1 Druhy evolučních algoritmů

Nezávisle na sobě vznikly čtyři druhy EA s rozdílnou aplikační oblastí. Charakteristické jsou zejména způsobem vytváření nových jedinců a strukturou, která tyto jedince reprezentuje [4]:

Evoluční strategie (ES) vytváří nové jedince kopírováním a modifikací (*mutací*) nejúspěšnějších jedinců v populaci. Strategiemi $(1 + 1)$, $(1 + \lambda)$, $(\mu + \lambda)$ nebo (μ, λ), označuje počty rodičů, potomků a způsob jejich výběru do další generace. Kromě samotného řešení v sobě jedinci mohou kódovat v podobě reálných čísel i tzv. strategické parametry (míra mutace apod.), které se adaptují (vyvíjejí) společně s řešením [18].

Evoluční programování (EP) kóduje jedincem graf a jako v předešlém případě, nový jedinec vzniká mutovanou kopií starého. Bylo navrženo pro automatizaci návrhu konečných automatů.

Genetické algoritmy (GA) jsou v současnosti nejpoužívanějšími evolučními optimalizačními algoritmy. Každý jedinec je tvořen řetězcem pevné délky, obvykle binárních znaků, které kódují parametry účelové funkce a GA hledají nejlepší kombinaci jejich hodnot (viz optimalizace parametrů motorů pro Boeing 777 uvedená výše).

Pro GA je klíčový operátor křížení, kterým vytváří nové jedince kombinováním znaků úspěšných jedinců. Jako jediné z EA používají křížení tak, že potomek je podobný oběma rodičům a mutaci používají jen pro udržení *diverzity* jedinců v populaci. GA se hodí na problémy s menším počtem kratších vazeb mezi blízkými znaky v řetězci a budou podrobněji popsány v kapitole 6.

Genetické programování (GP) pracuje obvykle se syntaktickými stromy libovolné velikosti, kterými kóduje proměnné i funkce. Používá se na složité problémy s větším počtem delších vazeb mezi vzdálenějšími znaky, kde křížením dochází k jejich častému znehodnocení (přetrhání).

GP se křížením ani tak nesnaží cíleně kombinovat znaky zdatných jedinců jako GA, ale používá ho spíše k udržení *genové diverzity* populace. Navíc je křížení definováno tak, že v potomku nedodržuje umístění ani funkci stavebních bloků druhého rodiče a tak jsou potomci zpravidla podobní jen jednomu rodiči.

GP se dělí na dvě základní skupiny: V *tree-based genetic programming* chromozom představuje přímo stromovou strukturu a *linear genetic programming* (LGP), kde je tato struktura v chromozomu kódována nepřímo řetězcem znaků, například pomocí *Readova lineárního kódu* [1]. V této práci, počínaje kapitolou 7, se ale zaměřím především na TGP a jeho reprezentaci digitálních obvodů.

Speciálním druhem GP je pak **kartézské genetické programování** (CGP), které obvykle používá pouze mutaci a je popsáno v podkapitole 7.5.

Kapitola 6

Genetický algoritmus

V této kapitole jsou vysvětleny základní pojmy, operátory a fáze genetického algoritmu, které v odpovídajícím pořadí popisují jednotlivé podkapitoly. Dále je zmíněna teorie stavebních bloků, na kterou budou navazovat kapitoly příští.

6.1 Kódování problému

Chromozomy jedinců jsou v GA řetězce pevné délky, obvykle binárních znaků, kódujících parametry účelové (hodnotící) funkce. Jsou vytvořeny po vzoru *haploidních sad chromozomů* nepohlavně se rozmnožujících organismů. Existují i varianty s *diploidními sadami chromozomů* [8], které se vyznačují především dobrou adaptivitou na měnící se podmínky. V případě evolučního návrhu jsou ale tyto podmínky neměnné a tak budu dále uvažovat jen chromozomy haploidní.

Genotyp/fenotyp jedince označuje jedincem představované řešení v zakódovaném/dekódovaném tvaru. *Znak fenotypu* je kódován jedním *genem* a bývá (spolu) zodpovědný za některou *fenotypovou vlastnost* jedince. Konkrétní hodnota určitého genu se nazývá *alela*.

6.2 Inicializace populace

Počáteční populace jedinců (*kandidátních řešení*) je vytvořena pomocí generátoru náhodných čísel tak, aby co nejrovnoměrněji pokryla prohledávaný stavový prostor. Chceme-li rychleji najít vyhovující řešení, o kterém víme, v jaké cílové oblasti se nachází, můžeme do ní zaměřit hledání algoritmu jejím hustším navzorkováním [15].

6.3 Ohodnocení jedince

Každý nový jedinec v populaci je ohodnocen *fitness funkcí*, jejíž hodnota udává jeho zdatnost. Tato hodnota určuje na kolik splňuje jím představované řešení zadání a tedy jak slibné bude hledat jemu podobné optimální řešení v jeho okolí. Rozlišujeme několik druhů *fitness funkcí* [15, 1]:

Hrubá (raw) má hodnoty závislé na problému. Menší hodnota může znamenat menší odchylku od požadavků a představovat lepší řešení, ale také nemusí.

Standardizovaná (standardized) hodnotí optimální řešení nulovou hodnotou. Vždy platí: čím vyšší hodnota, tím horší řešení.

Přizpůsobená (adjusted) hodnota fitness se pohybuje v intervalu $< 0, 1 >$, kde 0 znamená opět optimální řešení.

Normalizovaná (normalized) hodnota fitness se také pohybuje v intervalu $< 0, 1 >$, ale odpovídá poměrné zdatnosti jedince vůči ostatním v populaci. Součet všech normalizovaných fitness hodnot v populaci je vždy roven jedné.

6.4 Selekce

Provádí kvazináhodný výběr vhodných jedinců dle jejich fitness pro proces reprodukce (rekombinace) [1]. Počet takto vybíraných rodičů je jedním z nastavitelných parametrů GA a v průběhu evoluce se nemění. Používá se několik typů selekce (selection) [18, 10, 15]:

Ruletová (roulette-wheel) nebo také proporcionální selekce, vybírá jedince s pravděpodobností odpovídající jejich fitness. Tato selekce není moc vhodná v případě velkých rozdílů fitness hodnot mezi jedinci, protože by mohla způsobit *saturaci* populace do potomků jednoho jedince.

Turnajová (tournament) náhodně vybírá skupiny kandidátů a z každé vybere jednoho s největší fitness. Hodí se i pro velké rozdíly fitness hodnot jedinců v populaci, protože dává šanci na množení širšímu spektru jedinců.

Pořadová (rank) seřadí jedince podle fitness hodnot a vybírá je s pravděpodobností úměrnou jejich pozici. Také je vhodná při velkých rozdílech fitness hodnot mezi jedinci.

Volba vhodného typu selekce je závislá na tvaru *účelové funkce* (viz kapitola 4) a má vliv na rychlost konvergence algoritmu.

6.5 Křížení

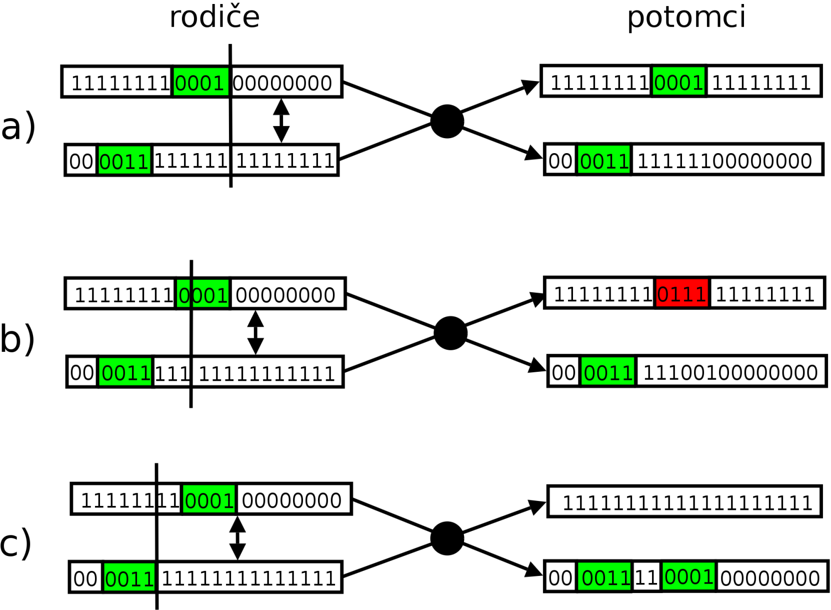
Jak již bylo zmíněno v kapitole 5.1, GA se snaží křížením nerovnoměrně rozdělit do potomků silné sekvence genů z obou nadprůměrných rodičů. Protože dopředu nevíme, které sekvence genů v chromozomech rodičů jsou za jejich nadprůměrnost zodpovědné, rozdělí se různě do obou potomků a ten zdatnější bude mít větší šanci v šíření genů pokračovat.

Křížením tak může dojít buď k rovnoměrnému rozdělení silných sekvencí genů do obou potomků, rozbití některých silných sekvencí na nefunkční části nebo k akumulaci více silných sekvencí do jednoho z potomků na úkor druhého (viz obrázek 6.1). Který z případů nastane záleží na délce silných sekvencí, jejich rozmístění v chromozomu a pozici náhodně zvoleného bodu (nebo bodů) křížení, od kterého se prohodí zbývající části chromozomu rodičů.

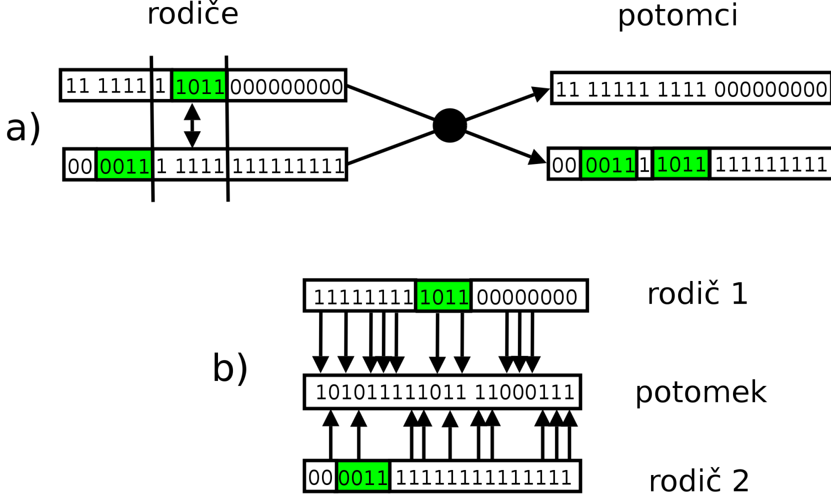
Aby byla šance na zachování vazeb co nejlepší, vybírá se podle počtu genů a vzdálenosti jejich vazeb vhodný způsob křížení [17]:

Jednobodové (one-point) je vhodné i pro větší počty genů s blízkými vazbami, nejlépe mezi sousedními geny. Takové jsou zvýrazněny na obrázcích 6.1 a 6.2.

Vícebodové (multi-point) je lepší pro vzdálenější vazby malého počtu genů. Čím více vzdálenějších vazeb, tím více bodů křížení.



Obrázek 6.1: Příklad a) rozdělení, b) rozbití, c) akumulace silných sekvenci genů při jednobodovém křížení.



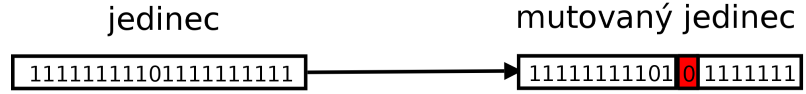
Obrázek 6.2: Příklad a) dvoubodového křížení a b) uniformního křížení.

Uniformní (uniform) má význam v případě velkého počtu vazeb mezi jednotlivými vzdálenými geny, náhodně rozmístěnými v chromozomu.

I zde je volba *problémově závislá* a vyžaduje vložení do algoritmu určité znalosti povahy problému, pro jeho efektivnější řešení.

6.6 Mutace

Úkolem mutace je udržování *genové variability* (diversity) v populaci, prováděním náhodných změn genů v chromozomu [18]. Obvykle se provádí jako inverze náhodného genu, jak znázorňuje obrázek 6.3. Mutace následuje hned po křížení, ale používá se jen s malou pravděpodobností (kolem 5%) [1]. Velká pravděpodobnost mutace bude narušovat funkci křížení, ale malá znamená delší dobu strávenou v lokálních extrémech a pomalý vznik nových stavebních bloků. Vyladění tohoto parametru pro daný charakter prohledávaného prostoru má pozitivní vliv na efektivitu algoritmu.

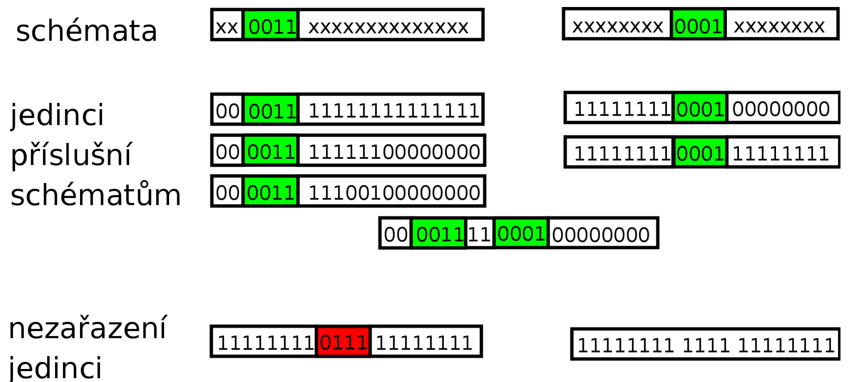


Obrázek 6.3: Příklad mutace jedince inverzí náhodného genu jeho chromozomu.

6.7 Teorie stavebních bloků

Silné sekvence vzájemně závislých genů nazýváme *stavební bloky*. Jejich skládáním, rozbíjením a důkazem *konvergence* GA se zabývá *teorie stavebních bloků* (building block hypothesis) [25]. Pro jejich sledování v populaci slouží *schémata* nebo též šablony. Jsou tvořeny pevnou a proměnnou částí, přičemž pevná část je společná všem jedincům stejného schématu. Na obrázku 6.4 je rozděleno osm jedinců z obrázku 6.1 podle schémat, jejichž pevná část odpovídá zvýrazněné silné sekvenci genů.

V chromozomu délky k, jehož geny mohou nabývat n hodnot (*alel*), můžeme najít $(n+1)^k$ schémat[1]. Proto nás budou zajímat jen ta zvlášť úspěšná, jež budou představovat silné stavební bloky kandidátních řešení. Úspěšnost schématu v populaci je dána počtem nositelů nebo lépe průměrem jejich fitness hodnot.



Obrázek 6.4: Klasifikace jedinců z obrázku 6.1 podle schémat silných sekvencí genů.

[1] n symbolů řetězce délky k tvoří n^k pevných kombinací. Proměnlivou část schématu v řetězci definujeme novým symbolem abecedy například x. Odtud tedy $(n+1)^k$ možných schémat.

Schéma teorém [17, 10, 1]**:** Počet krátkých, nadprůměrných schémat nízkého řádu se s přibývajícími generacemi genetického algoritmu v populaci exponenciálně zvyšuje.

Schématy nízkého řádu jsou myšlena schémata s malou pevnou částí, představující malý počet vzájemně vázaných genů. Krátké schéma zase znamená, že vázané geny nejsou na chromozomu daleko od sebe. Jinými slovy schéma teorém říká že: GA efektivně rozpoznává a kombinuje nadprůměrné stavební bloky, tvořené **malým počtem blízce vázaných genů**.

6.8 Nahrazovací strategie

Selekcí, křížením a případně mutací se ke staré populaci kandidátních řešení vytvoří nová populace potomků. Abychom udrželi populaci na stále stejném počtu, musí za tvorbou nových jedinců následovat likvidace starých. Sjednocení rodičovské populace a jejich potomků se nazývá *nahrazovací strategie* (replacement strategies) a rozlišuje dva základní přístupy s jedním volitelným [15, 18]:

Generační (generational) nahrazuje celou starou populaci potomky.

Steady-state vytvoří novou populaci nahrazením nejhorších, případně nejstarších jedinců staré populace, stanoveným počtem potomků (obvykle jen těch nejlepších). Počet nahrazovaných jedinců opět závisí na problému. Někdy se uvádí 40–60% nahrazení, jinde jen 10–25%.

Elitismus doplňuje některý z výše uvedených přístupů o mechanismus zajišťující přežití nejlepších jedinců jejich kopírováním do nové populace (doporučuje se cca 10% [1]).

6.9 Podmínky ukončení

V této fázi se vyhodnotí dosažené výsledky a rozhodne o dalším pokračování vývoje [14]. Pokud jsme s výsledky spokojeni, můžeme evoluci ukončit. Stejně tak pokud zjistíme, že vývoj již probíhá příliš dlouho (nastavený počet generací), populace saturovala do potomků jednoho jedince nebo že vývoj uvázl v lokálním extrému a několik posledních generací se průměrná fitness populace nezlepšuje. Nejčastěji se ale rozhodneme ve vývoji pokračovat selekcí a tvorbou další generace.

6.10 Shrnutí genetického algoritmu

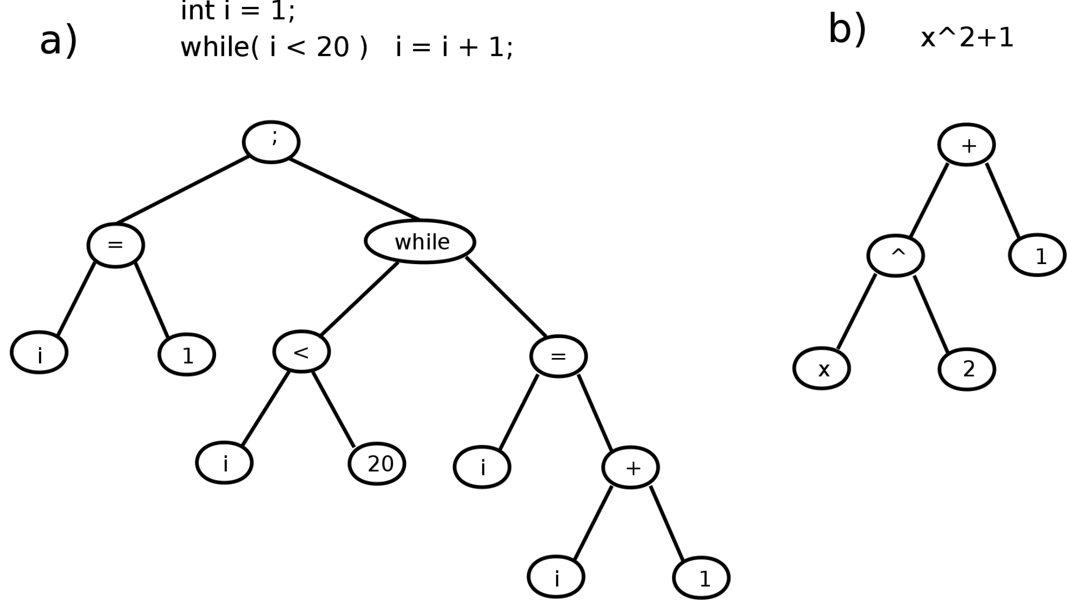
Operátor selekce vybírá nadprůměrné jedince, křížení se snaží kombinovat jejich silné stavební bloky a operátor mutace zodpovídá za tvorbu nových bloků ke kombinování. Tím každou generací roste koncentrace silných stavebních bloků v chromozomech části potomků, kteří postupně nahrazují ty méně zdatné. GA je efektivní v nacházení a kombinování malých stavebních bloků z genů na chromozomu blízko sebe. S jejich rostoucí velikostí a koncentrací v rodičích však dochází křížením a mutací k jejich častějšímu rozbíjení. To brzdí vývoj a snižuje efektivitu prohledávání stavového prostoru, což způsobuje nelineární nárůst času potřebného k jeho prohledání.

Kapitola 7

Genetické programování

Genetické programování je obdobou genetického algoritmu, ale pracuje s chromozomy proměnné délky pomocí upravených operátorů [18]. Způsoby selekce a ohodnocení však zůstávají stejné. Chromozom má obvykle *stromovou* nebo *grafovou* genotypovou podobu shodnou se strukturou fenotypu, proto GP na rozdíl od většiny EA mezi genotypem a fenotypem nerozlišuje.

GP navrhl J. Koza počátkem devadesátých let zejména pro automatický návrh programů a *symbolickou regresi*. Tyto aplikace jsou typické většími stavebními bloky (podstromy) ze vzdálenějších vazeb, se kterými GA nejsou schopny efektivně pracovat. Přitom nejsilnější stavební bloky bývají nahuštěny v kořenech stromů (více v kapitole 8.1), což znemožňuje jejich akumulaci do jednoho chromozomu. A aby tomu nebylo málo, tak samotné stavební bloky mezi sebou často vytváří komplikované vazby (*epistatické interakce*), které stírají jejich hranice a vytváří jeden stavební superblok. GP se proto ani nesnaží o cílenou kombinaci silných stavebních bloků rodičů jako GA, ale vytváří potomky kopií superbloku rodičů s jednou náhodně vybranou zaměněnou částí.



Obrázek 7.1: Příklad a) syntaktického stromu programu a b) funkce.

7.1 Kódování problému

Chromozom v *tree-based genetic programming* (TGP) [25] představuje *syntaktický strom* (parser tree, syntax tree), znázorněný na obrázku 7.1, který se skládá ze tří typů uzlů: Kořenové uzly mohou být *terminály* i *funkce* a jejich výstupy tvoří výstupy celého stromu. Uzly ve větvích stromu reprezentují funkce, zpracovávající výstupy uzlů pod nimi, pro uzly v hierarchii stromu výše. Listové uzly, jsou *terminální* vstupní hodnoty, zakončující každou větev stromu.

V *linear genetic programming* (LGP) je tento strom v chromozomu kódován řetězcem znaků, například pomocí Readova lineárního kódu [1].

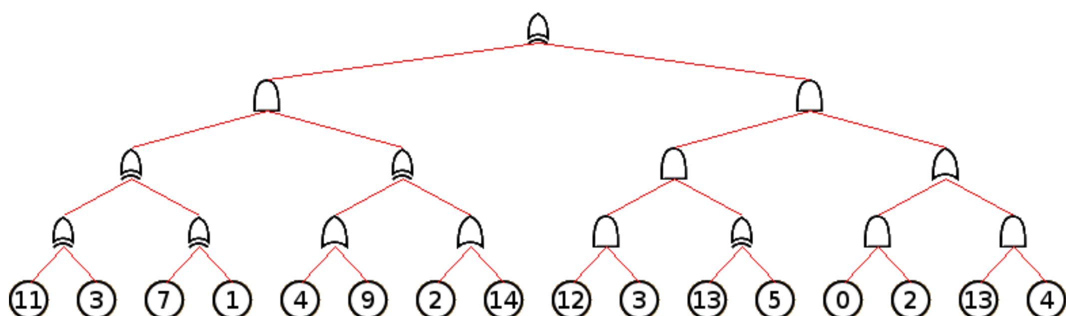
7.2 Inicializace

Pro náhodnou inicializaci počáteční populace se nejčastěji používá jedna ze tří metod, lišících se pokrytím stavového prostoru [15]:

Grow náhodně generuje každý uzel stromu z množiny funkcí a terminálů. Pro každou zvolenou funkci generuje i uzly pod ní tak, aby každá funkce byla zakončena některým terminálem. Při dosažení nastavené maximální hloubky stromu je větev ukončena terminálem automaticky. Výsledkem jsou jedinci se stromy nepravidelného tvaru.

Full vybírá pouze z množiny funkcí, dokud nedosáhne maximální hloubky, kde každou větev ukončí terminálem. Tvoří jedince se stromy pravidelného tvaru (viz obrázek 7.2).

Ramped Half-and-Half vytváří rovnoměrné skupiny stromů s různě nastavenou hloubkou od minimální až po maximální. Navíc polovinu stromů vytvoří metodou *Grow* a druhou metodou *Full*. Na rozdíl od předešlých dvou generuje rozmanitější stromy s lepším pokrytím stavového prostoru.



Obrázek 7.2: Metodou *Full* náhodně vygenerovaný pětipatrový binární strom.

7.3 Mutace

Použití mutace je v GP volitelné. Na rozdíl od GA, kde se aplikovala po křížení, se v GP obvykle aplikuje místo křížení s pravděpodobností cca 10% [9]. Vzhledem k tomu, že slouží jen jako doplněk pomáhající operátoru křížení zachovávat diverzitu velké populace, není její četnost pro rychlost konvergence až tak zásadní. Obvykle se implementuje jako nahrazení

náhodně vybraného podstromu novým náhodně vytvořeným podstromem, ale lze definovat i jiný způsob (*inzerce, delece, bodová mutace a pod.*) [13].

7.4 Křížení

V GP probíhá křížení podobným způsobem jako v GA, jen na místo skládání podřetězců rodičů, skládá jejich podstromy (viz obrázek 7.3).

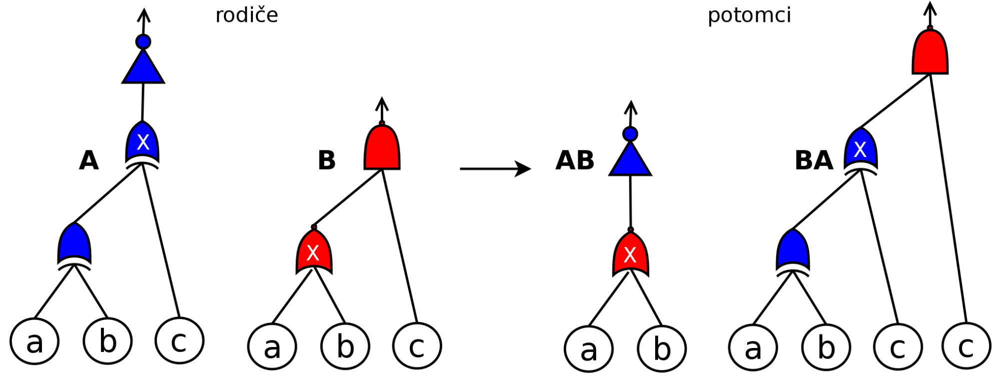
Důležitým rozdílem však je, že každý z potomků je podobný pouze na jednoho svého rodiče. Od druhého rodiče má sice jeho náhodně vybraný podstrom, ale na náhodném (jiném) místě. V potomkovi tedy zpravidla plní zcela jinou funkci, než v rodiči.

Křížení v GP nemá kombinovat stromy rodičů, ale pouze zachovávat diverzitu populace, což má v GA obvykle na starost mutace [12]. Nevýhodou této „mutace křížením“ je nutnost udržovat značnou populaci (v řádu několika tisíc) jedinců pro zajištění dostatečně pestrého počtu vzájemně vyměnitelných podstromů [18]. Křížení lze také doplnit „obyčejnou“ mutací a v CGP (popsaném v kapitole 7.5) jej dokonce zcela nahrazuje.

Bohužel, pravděpodobnost nalezení lépe fungujícího podstromu (respektive podgrafu), ať už náhodným výběrem (GP) nebo náhodným vytvořením (CGP), se postupem vývoje neustále zmenšuje. Čím lepší je obvod, tím je menší i množina jeho pozitivních změn, což také bývá častou příčinou uváznutí vývoje v *lokálním extrému*.

Například rozšíří-li se kvalitní obvod v populaci příliš, stane se tak na úkor konkurence, ze které by se časem mohl vyvinout obvod lepší. Navíc pokles *genové diverzity* populace sníží pestrost křížením vyměnitelných podstromů, což prakticky omezí vývoj lepšího obvodu na pouhé kombinace již existujícího. V takovém případě je velmi nepravděpodobné, že se z něj skokově vyvine obvod s lepším potenciálem pro další vývoj, ihned konkurenceschopný rozšířenému obvodu, jež se do současné podoby dlouho vyvíjel.

Obrázek 7.3 znázorňuje situaci, kdy se snažíme vyvinout obvod detekující přenos v jednobitové sčítačce. Podařilo se nám najít dva kandidáty, splňující většinu pravdivostní tabulky 7.1 a nyní se pokoušíme jejich křížením (kombinací) vyvinout obvod lepší.



Obrázek 7.3: Křížení stromů představujících kombinační obvody.

Jak vidíme v tabulce 7.1, *dědivost znaků* (vyjadřující míru v jaké se daný znak dědí z rodičů na potomky [2]) je v pravdivostních tabulkách různých rodičů velmi malá. Potomci mají téměř 40% znaků zcela nepodobných ani jednomu z rodičů a podle *standardizované fitness* v řádku „rozdílů proti **F**“, jsou i výrazně horší.

Tabulka 7.1: Pravdivostní tabulka a hodnoty fitness obvodů z obrázku 7.3. Sloupec **F** znázorňuje hledanou funkci, sloupce **A** a **B** funkce nalezených kandidátů a sloupce **AB** a **BA** funkce jejich potomků.

abc	**F**	**A**	**B**	**AB**	**BA**
000	0	1	0	1	0
001	0	0	0	1	1
010	0	0	0	0	0
011	1	1	1	0	0
100	0	0	0	0	0
101	1	1	1	0	0
110	1	1	0	0	0
111	1	0	1	0	1
rozdílů proti **F**:	0	2	1	6	4

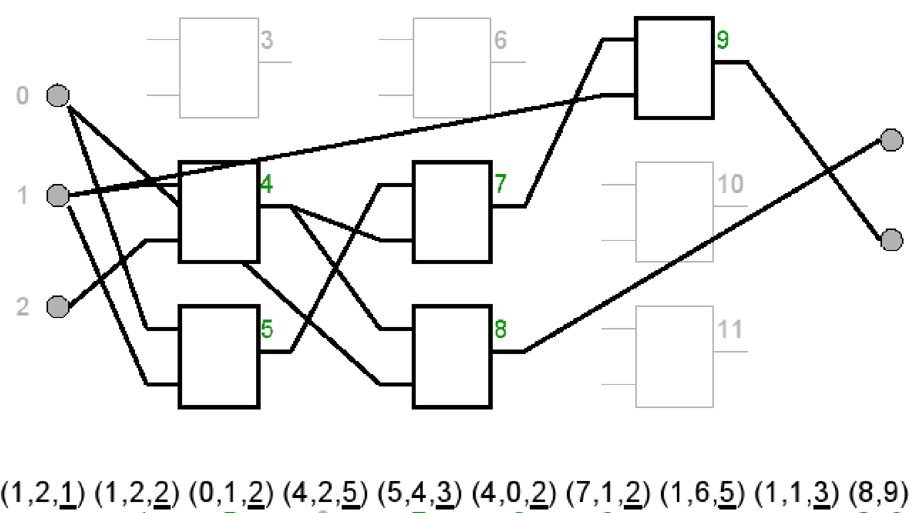
7.5 Kartézské genetické programování

CGP představili J. Miller a Y. Tompson v roce 2000, jako variantu GP zaměřenou na návrh digitálních obvodů na úrovni hradel (ve struktuře podobnější rekonfigurovatelným obvodům FPGA) [18]. CGP používá evoluční strategii $(1+L)$, tedy L potomků je vytvořeno kopií a mutací nejlepšího jedince. Na rozdíl od GP používá malou populaci (typicky v jednotkách jedinců), kterou vyvíjí po mnoho generací. Obvod v CGP reprezentuje propojení uzlů (hradel) mřížky $n \times m$ zakódované řetězcem celých čísel uložené v chromozomu pevné délky. CGP tak na rozdíl od GP rozlišuje mezi *fenotypem* a *genotypem*. Rozměry mřížky vymezující prohledávaný stavový prostor jsou zadány pevně a musí být vhodně zvoleny s ohledem na hledané řešení. Obecné GP žádné omezení stavového prostoru nemá.

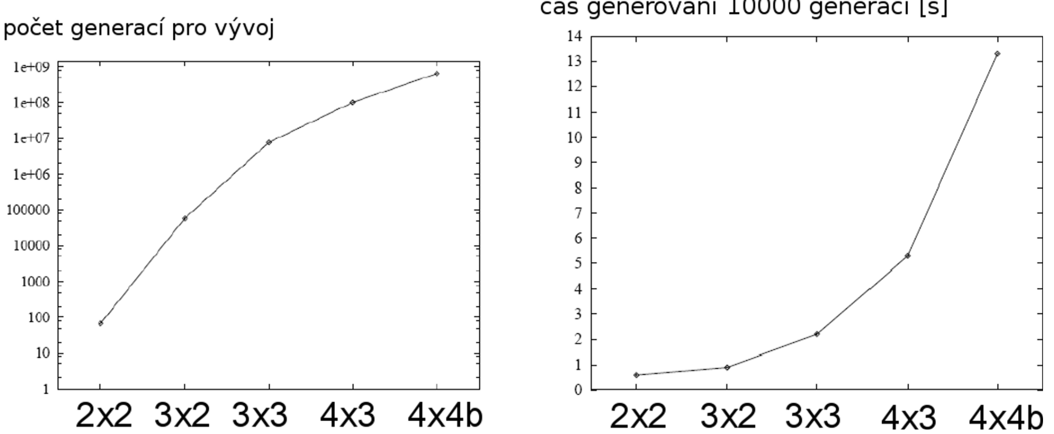
Aby propojováním hradel vznikaly pouze kombinační obvody (nikoli sekvenční), mají omezenu *konektivitu* pouze na předešlá hradla a to do vzdálenosti definované `l-back` parametrem [18]. Hodnota `l-back` parametru, mimo to že dále omezuje stavový prostor řešení a tím počet kombinací které může algoritmus projít, také zásadně ovlivňuje vlastnosti výsledného zapojení. Velká hodnota může být sice přínosná pro nízký počet hradel nebo malé zpoždění nalezeného řešení, ale zároveň zhoršuje zřetězitelnost funkce tohoto obvodu (*pipelining*).

Hlavním přínosem CGP je snadnější evoluce nejen simulací obvodu v počítači, ale přímo i emulací obvodu v rekonfigurovatelných obvodech, což přináší 20–50 násobné urychlení vývoje zapojení obvodu. Bohužel složitost návrhu obvodu s počtem komponent roste velmi nelineárně, takže ani několikanásobné urychlení návrhu nám ke složitějším obvodům příliš nepomůže. Zatím co pro návrh násobičky 2×2 bity (4 vstupy, 4 výstupy) je potřeba v průměru 100 generací, pro návrh násobičky 4×4 bity (8 vstupů, 8 výstupů) je to už miliarda generací [18] a složitější návrh je v současnosti jen těžko možný. Jediným řešením je zřejmě urychlení konvergence prohledávacího algoritmu.

Pokud neomezíme vývoj obvodu jen na proces návrhu a implementujeme evoluční algoritmus na stejný čip společně s rekonfigurovatelnou logikou, můžeme tak vytvořit i částečně samoopravující se, případně vysoce odolné obvody. Ty již ale spadají do oblasti vyvíjejícího se (*evolvable*) hardware a my se budeme nadále zabývat čistě evolučním návrhem.



Obrázek 7.4: Příklad zapojení úplné jednobitové sčítačky v CGP a chromozom ve kterém je uloženo. Podtržené číslo určuje logickou funkci hradla (NAND=0, NOR=1, XOR=2, AND=3, OR=4, NOT=5), zbývající čísla zapojení vstupů [18].



Obrázek 7.5: Příklad nelineárního nárůstu složitosti při návrhu násobiček $n \times m$ bitů v CGP [18].

Kapitola 8

Návrh hybridního algoritmu

V této kapitole jsem nejprve zjišťoval vliv jednotlivých prvků na funkci obvodu vedoucí k nerovnoměrnému rozložení stavebních bloků v chromozomu GP oproti GA, podle kterého jsem pak navrhl hybridní algoritmus.

8.1 Vliv mutace na funkci stromu

Na konferenci GECCO 2000 byl prezentován algoritmus, zjednodušující binární rozhodovací diagramy (*binary decision diagrams – BDDs*), které reprezentují booleovskými funkcemi popsaný obvod [6]. Algoritmus odhadoval dopad aproximace funkcí uzlů konstantami a vybíral vhodné kandidáty pro nahrazení. Autoři dosáhli zajímavých výsledků při zmenšení rozhodovacího stromu s minimálním vlivem na jeho funkci. Řešili sice jen stromy s logickými prvky AND a OR, ale předvedli, že lze odhadnout dopad změny funkce prvku na funkci celého stromu.

Vliv libovolného logického prvku na výslednou funkci záleží sice na konkrétním zapojení obvodu, ale průměrný vliv změny prvku lze odhadnout i z jeho pozice. Konkrétně vezmu-li pravdivostní tabulky dvouvstupových hradel AND, OR, XOR, liší se mezi sebou v průměru polovinou bitů. Změna jedné logické funkce kořenového hradla na některou jinou z těchto tří tedy přinese v průměru 50% změnu funkce stromu. Podle dalších výpočtů již ale obdobná změna o patro níže má průměrně jen 33% dopad a s každým dalším patrem dopad dále klesá.

Pro ověření této domněnky jsem sestavil program `pyramid` (viz příloha A.4) a nechal ho metodou *Full* generovat náhodné binární stromy z hradel AND, OR, XOR různých velikostí. V každém nelistovém patře každého stromu jsem dočasně změnil funkci nejlevějšího hradla[1] a porovnal pravdivostní tabulku změněného stromu s původní.

Výsledky opakovaně prokázaly předpokládaný přibližně 33% pokles průměrného počtu vyvolaných změn s každým nižším patrem, pro stromy všech testovaných velikostí (viz tabulka 8.1). Obdobné výsledky přinesly ternární, kvaternární, smíšené stromy i stromy s různou kardinalitou množiny listů. Jednotlivé typy stromů se mezi sebou liší jen hod-

[1] Na každém patře stačí změnit jen jedno hradlo, protože díky komutativitě použitých operací je průměrný vliv ostatních pozic stejného patra ekvivalentní. Pokud by operace byly i distributivní (což nejsou), byl by ekvivalentní i vliv jednotlivých pater.

Například výrazy [(a AND b) XOR (c OR d)] i [(c OR d) XOR (b AND a)] jsou si funkčně rovnocenné, na rozdíl od výrazu [a AND (b XOR c) OR d].

notou průměrného poklesu, která je opět pro všechna patra všech velikostí daného typu stromu stejná.

Tabulka 8.1: Vliv pozice hradla na funkci binárního stromu (průměr ze 100 pokusů).

	průměrný počet změněných bitů z 65536						
výška stromu / patro	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2.	32276						
3.	30617	21401					
4.	32485	19577	14193				
5.	33317	22810	15801	8822			
6.	33116	21062	14615	8868	6389		
7.	32748	21416	12757	7943	6536	3381	
8.	34106	19588	13311	7836	4614	2937	1949
zjištěný vliv:	47-52%	30-35%	19-24%	12-14%	7-10%	4-5%	3%
předpokládaný vliv:	50%	33%	22%	15%	10%	7%	4%

Podíl vlivu kořenového a nižších hradel na funkci obvodu je dobře patrný i z tabulky 8.2. Kořenové hradlo určené prvním znakem zápisu má největší podíl na umístění v tabulce. Proto se obvody s kořenovým hradlem AND, jehož funkce dává v 75% případů nulu, se vyznačují malým počtem jedniček a shlukují se v levé části tabulky, zatím co obvody s kořenovým hradlem OR, jehož funkce dává naopak v 75% jedničku, mají jedniček výrazně více a tak se shlukují vpravo. Hradla XOR s funkcí vracející v polovině případů nuly a v druhé jedničky jsou pak uprostřed.

Zajímavá situace nastane, když kořenové hradlo utlumuje jedničky a obě hradla pod ním je naopak podporují (například AOO) nebo opačná situace s OAA. Jen v takovém případě překoná společný vliv obou nižších hradel opačný vliv hradla nadřazeného.

Tabulka 8.2: Všech 3^3 obvodů tvaru (a Y b) X (c Z d) zařazených podle počtu jedniček v jejich pravdivostních tabulkách, kde X, Y, Z jsou hradla AND, OR nebo XOR a znaky a, b, c, d obvodové vstupy. Každý obvod je zapsán trojicí písmen, kde kořenové hradlo X označuje počáteční písmeno (A – AND, O – OR, X – XOR) a další dvě písmena označují zbývající hradla Y a Z.

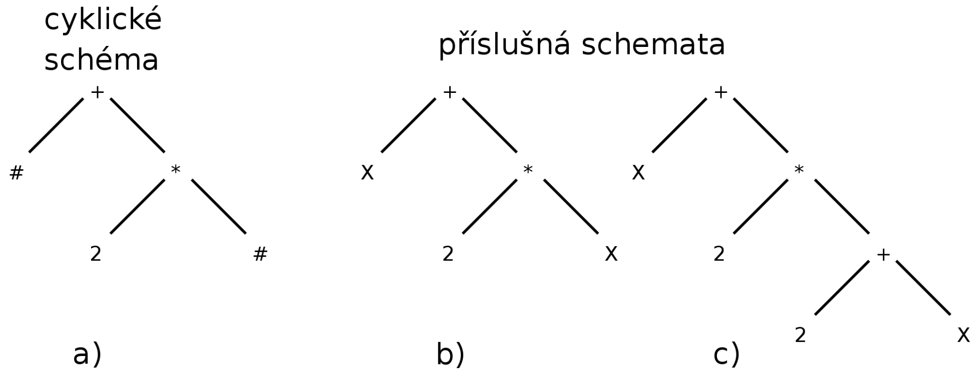
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AAA	AXA	AOA	AXX		AOX	OAA	XXX	AOO	OXA		OXX	OOA	OOX	OOO
	AAX	AAO			AXO		XXA		OAX			OAO	OXO	
					XAA		XXO		XOA					
					XOO		XOX		XAO					
							XAX							

8.2 Stavební bloky

Z experimentů plyne nerovnoměrné rozložení vlivu celých skupin prvků v chromozomu na výslednou funkci stromu. Čím hlouběji se podstrom ve stromu nachází, tím menší má

vliv na jeho funkci. Silné stavební bloky nejvíce zodpovědné za dobrou funkci stromu tak zřejmě vznikají především v kořenových částech, což znemožňuje jejich efektivní kombinování v jednom chromozomu a tím i rychlejší šíření v populaci širším počtem zdatných nositelů. Proto se s klasickými lineárními schématy sledujícími šíření stavebních bloků v GP nesetkáváme tak často jako v GA.

Kvůli nízké využitelnosti stejných stavebních bloků v různých stromech jich vzniká velké množství. To se snaží kompenzovat *cyklická schémata* [5], která představují celou skupinu řešení, založených na různém počtu iterací stejného cyklického stavebního bloku (viz obrázek 8.1). Ty se však využívají spíše pro symbolickou regresi a tak je zde zmiňuji jen okrajově.



Obrázek 8.1: Příklad a) cyklického schéma a dvou jedinců b) a c), které reprezentuje [5].

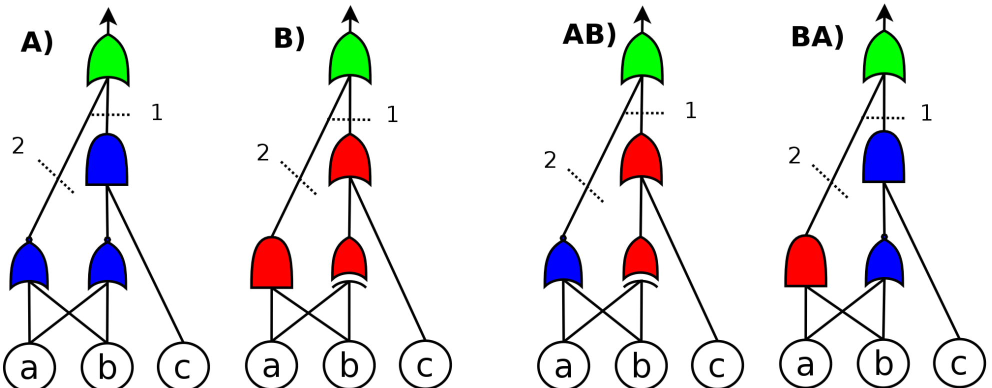
8.3 Dědivost znaků podobných rodičů

Křížení na obrázku 8.2 je obdobou příkladu 7.4. Opět hledáme funkci **F** pomocí křížení kandidátních řešení A a B, jenže tentokrát mají rodiče stejnou kořenovou strukturu. V pravdivostní tabulce 8.3 těchto obvodů je vidět, že potomek AB je svým fenotypem velmi podobný rodiči B ze kterého zdědil největší stavební blok a funkcí dokonce zcela shodný. Naopak potomek BA je fenotypem podobný rodiči A a „pouhá“ čtvrtina jeho znaků v pravdivostní tabulce se nepodobá ani jednomu z rodičů.

Potomci podobných rodičů sice dosahují vyšší dědivosti znaků, než potomci různých rodičů, ale přesto je část jejich znaků dána vazbami mezi stavebními bloky a ty bude i nadále těžké křížením zachovat. Výhodou křížení podobných rodičů jsou nejen průměrně zdatnější potomci oproti potomkům různých rodičů, ale také potomci podobnější na své rodiče, což je dělá předpověditelnějšími.

Nevýhodou je nižší diverzita genů v populaci, kdy při křížení velmi podobných jedinců může dojít k opakovanému vzniku již vyzkoušených *fenotypů*. Podobnost jedinců na obrázku 8.2 je natolik velká, že výběrem kteréhokoli ze dvou bodů křížení vzniknou stejní potomci AB a BA a jejich opětovným křížením vzniknou potomci identičtí se svými prarodiči A a B.

Párováním rodičů podle pravdivostních tabulek a vhodným nastavením minimální nutné mohutnosti množiny *přípustných bodů* křížení (na zmíněných pět) by mělo podíl tohoto negativního příbuzenského křížení významně omezit.



Obrázek 8.2: Příklad křížení obvodů se společnou nadřazenou strukturou. Dvě varianty přípustných bodů křížení 1,2 podle kapitoly 8.4 jsou označeny tečkovanou čarou a číslem. Ve skutečnosti jsou si ale stromy až příliš podobné a jejich křížení by nebylo povoleno.

Tabulka 8.3: Pravdivostní tabulka obvodů z obrázku 8.2. Sloupec **F** znázorňuje hledanou funkci, sloupce **A** a **B** funkce nalezených kandidátů a sloupce **AB** a **BA** funkce jejich potomků.

abc	**F**	**A**	**B**	**AB**	**BA**
000	0	0	0	0	0
001	0	1	1	1	0
010	0	1	1	1	0
011	1	1	1	1	1
100	0	0	1	1	0
101	1	1	1	1	1
110	1	1	1	1	1
111	1	1	1	1	1
rozdílů proti **F**:	0	2	3	3	0

8.4 Upravené křížení

Velké množství zmetků s nízkou dědivostí znaků vznikajících při křížení **různých** jedinců (popsané v kapitole 7.4) a klesající vliv stavebních bloků s jejich vzdáleností od kořene (z kapitoly 8.2), mne vedly k návrhu algoritmu s křížením **podobných** jedinců preferujícím zachování nejsilnějších stavebních bloků v kořenech stromů. Tento hybridní algoritmus zachovává pozice cíleně kombinovaných stavebních bloků jako GA a přitom respektuje jejich kořenovou orientaci s vazbami typickými pro GP. Křížení vychází z obecného GP, ale snaží se vyhýbat dříve popsaným neefektivním situacím následujícími omezeními:

1. Kříží se mezi sebou jen jedinci se shodnou kořenovou strukturou, jejichž pravdivostní tabulky se nejlépe doplňují. Právě u takovýchto rodičů je největší šance na vytvoření lepšího potomka vhodnou kombinací jejich podstromů.

2. Body křížení se pro oba rodiče vybírají párově tak, aby se jimi obvody lišily a přitom měly oba stejnou nadřazenou strukturu (tedy shodnou cestu od bodu křížení až ke ko-

řeni stromu). Vyměněné stavební bloky pak mají v potomcích podobnou funkci jako v rodičích, což zvyšuje *dědivost znaků* rodičů.

3. Čím jsou si jedinci podobnější, tím jsou jejich body křížení od kořene vzdálenější a tím méně se mohou potomci funkčně lišit (viz kapitola 8.1). Takovéto urychlení konvergence značně zvyšuje citlivost metody k *uváznutí* v důsledku ztráty *genové diverzity* populace. Proto je zavedena mutace a *stárnutí rodů* (popsané v kapitole 8.6).

4. Poklesu genové diverzity populace, vedoucí ke zbytečnému zkoušení stále stejných kombinací stavebních bloků (viz také příklad 8.3), se brání také tím, že se kříží jen jedinci, kteří mají na výběr alespoň mezi pěti body křížení. Jedinci rodičovské populace, kteří si nenašli dostatečně odlišného partnera pro křížení, jsou mutováni, čímž se genová diverzita populace obnovuje.

Tyto omezující podmínky snižují pravděpodobnost rozbití stavebních bloků rodičů, případně jejich neuchycení v potomcích z důvodu odlišné nadřazené struktury. Zachováním umístění a funkce jednotlivých stavebních bloků v dalších generacích je podporován jejich vývoj a specializace pro funkci na dané pozici.

Omezení křížení jen na povolené kombinace rodičů ale také snižuje jejich šanci na nalezení vhodného partnera v rodičovské populaci. Proto se zbývající nepároví jedinci rodičovské populace budou klonovat s mutací. Míra mutace tak bude nepřímo úměrná *genové diverzitě populace*, čímž se sama stabilizuje na potřebné hodnotě *zápornou zpětnou vazbou*. Mutace bude spočívat ve vybrání podstromu a jeho nahrazení náhodně vygenerovaným stromem, jako v obecném GP.

8.5 EMOEA

Na konferenci NASA/ESA (2006) [7] byla představena metoda vývoje obvodu EMOEA (*Efficient Multi-Objective Evolutionary Algorithm*) s podobným konceptem. Autoři také uvažovali o zvýšení efektivity obecného algoritmu vhodným výběrem bodů křížení. Tyto body ale vybírali podle jejich vlivu na fitness hodnotu obvodu zjištěném při jejich poslední změně. Tato metoda je podobná zpětnému zjišťování podílu vah neuronů na odchylce výstupní vrstvy neuronové sítě metody *zpětného šíření chyby* (backpropagation)[20].

8.6 Stárnutí rodů

Při návrzích některých obvodů, typicky u bitových násobiček, dochází k častému uváznutí v lokálním extrému popsaném v kapitole 7.4. Proto jsem do návrhu hybridního algoritmu zavedl i jednoduchý *fenotypový watchdog* s penalizací hojně se množících a přesto se nelepšících obvodů.

Rodovou linii tvoří obvody, které od svého předka zdědily kořenovou (tedy nejvlivnější) část *fenotypu*. Každý obvod si počítá kolikrát se křížil a každý potomek jeho rodové linie v tomto počtu pokračuje. Jen obvody lepší než jejich rodič začínají počítat zase od nuly. U ostatních po překročení nastavené meze dojde k penalizaci hodnoty jejich fitness.

Vymírání rodových linií dlouhodobě se nelepšících obvodů umožňuje vyvinout se i zatím méně konkurenceschopným obvodům a snižuje riziko uváznutí ve slepé uličce vývoje tohoto *fenotypu*. Kratší doba vývoje neperspektivních obvodů také urychluje konvergenci algoritmu ke hledanému řešení.

Kapitola 9

Implementace

V programovacím jazyce C++ jsem implementoval algoritmus obecného a hybridního GP s volitelnými rozšířeními o „stárnutí rodů“ a „simulaci křížení“ (popsaných v kapitolách 8.6 a 9.4). Jedná se o konzolové multiplatformní programy navrhující libovolné *sekvenční digitální obvody* na *úrovni logických hradel* (viz kapitola 2.1). Funkce, použití a ovládání programů jsou podrobně popsány v uživatelské příručce v příloze A.

9.1 Zapojení a výpočet výstupních hodnot hradel

Logické hradlo je v programu definováno zcela obecně, jako uzel (krabička) určitého typu, který definuje počet jeho vstupů a logickou funkci nad nimi. Tyto uzly jsou obousměrně zapojeny (v počáteční populaci náhodně) do stromových struktur reprezentujících digitální obvody. Jednoduchým nastavením na vyhrazené pozici zdrojového kódu si lze určit ze kterých typů hradel se mají vyvíjené obvody skládat, případně bez nutnosti změny algoritmu samotného definovat typ nový s libovolným počtem vstupů.

Každý uzel stromu zpracovává hodnoty uzlů připojených bezprostředně pod ním a výsledky zase poskytuje těm přímo připojeným nad ním. Listové prvky stromu (v hierarchii nejnižší) jsou typu Ix, nemají již žádný vstup a jejich funkcí je vracet konstanty reprezentující všechny kombinace vstupních hodnot obvodu (viz příklad 9.1). Hodnotou kořenového prvku (v hierarchii nejvyššího) je odezva obvodu na všechny kombinace hodnot obvodových vstupů. Tato hodnota je pak porovnávána se zadanou pravdivostní tabulkou požadovaného obvodu, která je definována pro obvodový výstup, reprezentovaný tímto kořenovým prvkem.

Každý obvod z obrázku 8.2 se třemi vstupy (označenými a, b, c) má 2^3 možných kombinací vstupních hodnot. Pravdivostní tabulka 8.3 příslušná jednobitovému obvodovému výstupu, definuje jeho hodnotu v každé z těchto kombinací a tak má osm jednobitových hodnot. Čím kvalitnější je obvod v tím více kombinacích bude jeho výstupní hodnota odpovídat požadované hodnotě v pravdivostní tabulce a tím lepší ohodnocení fitness se mu dostane. Funkce obvodu A z obrázku 8.2 odpovídá výrazu:

```
((a OR b) OR ((a OR b) AND c))
```

Vyhodnocením tohoto výrazu pro každé z osmi možných ohodnocení jeho boolovských proměnných lze vypočítat pravdivostní tabulku obvodu, uvedenou ve sloupci **A** tabulky 8.3. Pro počítačové zpracování je však mnohem výhodnější, pokud jsou boolovské proměnné nahrazeny konstantami podle tabulky 9.1 [18]. Bitové pozice těchto konstant rovněž pokrývají všechny kombinace hodnot vstupů a jsou na sobě také nezávislé, ale přitom je lze vyhodnotit najednou – jako jeden bajt.

Tabulka 9.1: Konstanty pokrývající všechny kombinace vstupních hodnot třívstupových obvodů.

vstup	**konstanta**
a	10101010
b	11001100
c	11110000

Tabulku 9.1 lze rozšířit pro čtyřvstupé obvody zdvojením počtu bitů stávajících konstant a přidáním řádku $d = 1111111100000000$. Takto lze indukcí vytvořit tabulku pro obvod s libovolným počtem vstupů. Konstanty obvodů se sedmi a více vstupy, přesahující velikost použitého datového typu `unsigned long long int` se vyhodnocují postupně. Tento datový typ jsem zvolil kvůli dobré přenositelnosti mezi 32 a 64-bitovými architekturami i operačními systémy. V celém programu je však použita pouze jeho přetypovaná varianta, takže jedinou změnou v hlavičkovém souboru lze použít datový typ jiný (například uint64_t) a při správném nastavení konstant třeba některý 32 či 128-bitový.

Paralelním výpočtem po celých 64-bitových segmentech pravdivostní tabulky, který je pro počítač 64-bitové architektury stejně náročný jako výpočet jejího jediného bitu, dochází k významnému urychlení vyhodnocení funkcí obvodů. Výstupní hodnoty hradel na kterých závisí funkce obvodu se u každého obvodu počítají jen jednou a jsou zapamatovány. Tyto hodnoty jsou od jedincova vzniku až po jeho zánik neměnné, protože jakékoli změny zapojení se provádí jen na jeho potomcích. Vzhledem k tomu, že potomci vznikají mutací případně křížením rodičů a protože tyto operace mění zapojení potomka proti rodiči jen velmi málo a předvídatelně, nepočítá se takto vzniklý obvod zcela znovu, ale pouze od bodu změny (bodu křížení) přes závislé uzly až ke kořenovému hradlu. Tím se rovněž urychluje vyhodnocení nového obvodu, které je zvláště patrné u nízkých a širokých stromů, vznikajících při optimalizaci na minimální zpoždění obvodu.

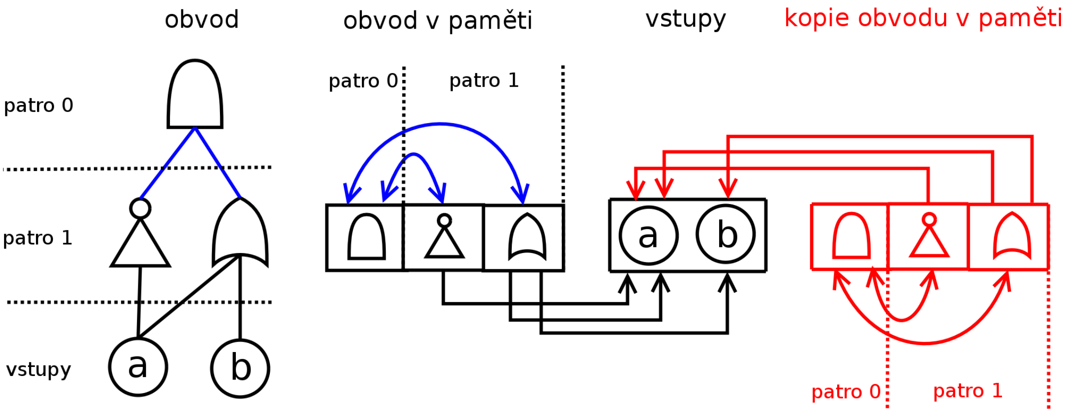
9.2 Paměťová reprezentace obvodu

Stromová struktura hradel tvořících obvod je v programu uložena v lineárním poli prvků, které jsou spolu provázány ukazateli. Tyto ukazatele ukazují pouze na prvky v rámci pole. Proto z počáteční adresy pole a hodnoty ukazatele lze spočítat index prvku na který ukazuje, index přenést do nového pole a v něm snadno duplikovat původní zapojení, jak znázorňuje obrázek 9.1.

K „naklonování“ obvodu tak postačí jediná dynamická alokace pole prvků obvodu, jejich hromadná inicializace a odpovídající propojení ukazatelů. Jak je z obrázku 9.1 také patrné, ukazatele na listové prvky představující obvodové vstupy jsou společné všem obvodům v paměti, zapojují se pouze jednosměrně a při „klonování“ se jen zkopírují.

Prvky obvodu jsou v paměťovém poli poskládány od kořenového s indexem 0, nasledovaném nejlevějším prvkem dalšího patra, až po nejpravější prvek nejnižšího patra. Patro ve kterém se prvek nachází souvisí s jeho indexem v poli a má klíčový význam pro zapojení dalších prvků nebo určení pořadí vyhodnocení při výpočtu výstupních hodnot prvků (viz zapojení kombinačních obvodů a význam `l_back` parametru v kapitole 7.5).

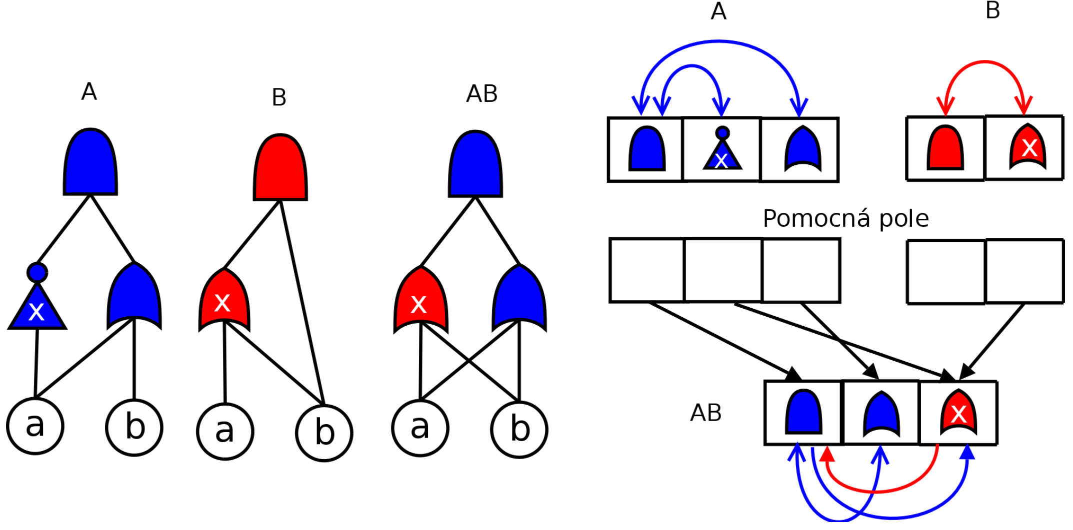
Protože zapojení obvodu je zohledněno již pořadím prvků v poli, lze všechny výstupní hodnoty hradel obvodu ve správném pořadí přepočítat od posledního prvku pole až po první (kořenový).



Obrázek 9.1: příklad reprezentace obvodu v paměti a vytvoření jeho kopie.

9.3 Křížení

Proces křížení kombinující stromy rodičů A a B v jeden strom AB probíhá od kořenového prvku rodiče A až po jeho prvek v bodu křížení stejně jako „klonování“ z minulé kapitoly. Prvek v bodu křížení rodiče A již není do potomka překopírován, stejně jako všechny další prvky rodiče A, které nemají vazbu mimo podstrom daný bodem křížení. Jak ukazuje obrázek 9.2, do potomka AB jsou naopak navíc překopírovány všechny prvky podstromu X rodiče B, takže počty ani indexy prvků kombinovaného zapojení AB neodpovídají originálům A a B. Proto jsou vytvořena dvě pomocná pole ukazatelů odpovídající polím prvků rodičů a postupně každý kopírovaný prvek si v nich na svoji pozici (odpovídající jeho pozici ve starém poli) uloží vlastní adresu v novém poli. Také se adresa odpovídající neexistující kopii prvku X rodiče A nastaví na pozici odpovídající kopii prvku X rodiče B, aby všechny zkopírované prvky původně odkazující na prvek X ve stromu A nově ukazovaly již na kopii prvku X ze stromu B.



Obrázek 9.2: příklad vytvoření potomka AB kombinací rodičů A a B s body křížení X.

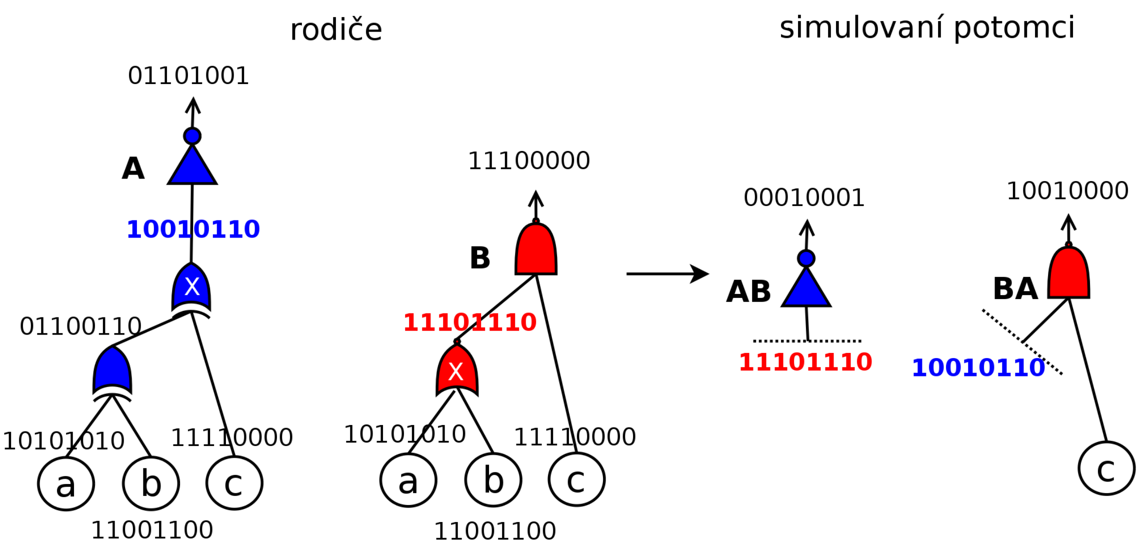
Kopírování prvků probíhá postupně po jednotlivých patrech z obou rodičů, takže každý kopírovaný prvek si může z pomocného pole zjistit adresy kopií svých původních rodičovských prvků, které byly překopírovány již dříve a rovnou s nimi obnovit obousměrnou vazbu. Jednosměrné vazby na listové prvky s neměnnou adresou se kopírují přímo.

Nejvytíženější funkce v programu, jako například křížení, jsem se snažil navrhnout jednoprůchodově s lineární složitostí. Kvůli tomu často používám pomocnou značku („flag“), kterou si označím prvky, se kterými jsem již při tomto volání funkce pracoval. Pokud bych pro tuto značku použil boolovskou proměnnou, musel bych ji vždy na konci (nebo začátku) funkce nastavit všem prvkům zpět na hodnotu false. Místo toho jsem použil celočíselnou proměnnou, kterou při použití prvku nastavuji na aktuální hodnotu globální proměnné. Po provedení funkce pak stačí pouze inkrementovat hodnotu globální proměnné a všechny hodnoty značek jsou odlišné od globální proměnné, tedy nastaveny zpět jako „nepoužity“.

Porovnání dvou celočíselných hodnot je přibližně stejně náročné, jako porovnání dvou booleovských hodnot a funkci ušetří jeden zbytečný průchod pole prvků.

9.4 Simulace křížení

Díky paměti výstupních hodnot každého hradla lze využít již existující zapojení obvodu v obou rodičích a pouze provést simulaci křížení dočasnou záměnou těchto výstupních hodnot mezi hradly v bodech křížení s přepočítáním vyvolaných změn (viz obrázek 9.3). Pro výpočet výstupní hodnoty obvodů je zcela nepodstatné, zda jsou mezi jedinci vzájemně prohozeny vybrané podstromy, spočítána a do zbytků stromů propagována jejich výstupní hodnota nebo zda se přepojování přeskočí a jedinci si rovnou vymění jejich dávno známé výstupní hodnoty.



Obrázek 9.3: Příklad simulace křížení záměnou výstupních hodnot hradel rodičů v bodech křížení (obdoba obrázku 7.3).

Tak se lze zcela vyhnout časově i paměťově nejnáročnější části vývoje, tj. vytvoření a zapojení nového obvodu (o následné likvidaci případného zmetka, kterých vzniká naprostá většina, ani nemluvě).

Vyhodnocení simulací je výrazně rychlejší než skutečné vytvoření, zapojení a vyhodnocení nového obvodu. Celkové urychlení algoritmu ale bude stejně záviset na nastaveném prahu odmítaných a přijímaných jedinců podle výsledků simulace, protože ty přijaté bude zapotřebí stejně vytvořit a zapojit.

Bude-li práh nastaven příliš přísně, může to mít neblahé důsledky na *genovou diverzitu* populace i *rychlost konvergence* algoritmu. Naopak časté vytváření křížených potomků degraduje potřebu simulace, která se tak stává nadbytečnou.

Nevýhodnou této simulace křížení je, že pro svoji jednoduchost není schopna dopředu odhadnout počet použitých hradel obvodu, podle kterého je ve fázi optimalizace obvod hodnocen. To lze samozřejmě napravit obsáhlejší heuristikou, ale ta by zatížila náročnost simulace, což by bylo proti jejím smyslu. Jednodušší bude omezit použití simulace jen na fázi návrhu a při optimalizaci již simulaci křížení nepoužívat.

Naopak výhodou této simulace je přirozený výběr kvalitních potomků, kteří pak budou nahrazovat podprůměrně kvalitní rodiče. Tímto selekčním tlakem i na populaci potomků by se mohla (při správném vyvážení) rychleji zvyšovat průměrná fitness v populaci.

Do obecného i hybridního algoritmu jsem tedy implementoval simulaci křížení jako volitelné rozšíření, které vymezuje kvalitu jedinců populace potomků pouze na jedince zdatnější (lépe splňující požadavky) než rodič po kterém zdědili kořenové hradlo.

9.5 Výpočet fitness

Nesplňuje-li hodnocený obvod pravdivostní tabulku, odpovídá jeho hodnota penalizaci za neúplnost plus počet nesplněných bitů pravdivostní tabulky. Jinak hodnota odpovídá počtu hradel nebo celkovému zpoždění obvodu, podle zvoleného optimalizačního kritéria.

Asi nejčastější volbou bude v první řadě zpoždění (počet pater) obvodu a pak teprve jeho velikost (počet hradel). Tuto volbu označuje hodnota `MINIMAL_BOTH` v konfiguračním souboru popsaném v příloze A a vyvážení těchto protichůdných požadavků je 100 hradel na jedno patro. Hodnota fitness funkčního obvodu je tedy počet jeho hradel plus 100× počet pater. Implicitní hodnota penalizace za neúplnost je nastavena na 10 000.

$$fitness = \begin{cases} 10000 + \text{počet nesplněných bitů}, & \text{není-li pravdivostní tabulka splněna} \\ \text{optimalizovaná hodnota}, & \text{jinak} \end{cases} \tag{9.1}$$

$$\text{optim. hodnota} = \begin{cases} \text{počet hradel obvodu}, & \text{při nastavení MINIMAL_BOXS} \\ \text{počet pater obvodu}, & \text{při nastavení MINIMAL_LEVELS} \\ 100 * \text{patra} + \text{hradla}, & \text{při nastavení MINIMAL_BOTH} \\ 0, & \text{při nastavení NONE} \end{cases} \tag{9.2}$$

9.6 Evoluce obvodu

Obvod s více výstupy vzniká odděleným *pseudoparalelním* vývojem jednotlivých obvodových výstupů, které jsou pak složeny dohromady. Ne jinak je tomu i v metodě EMOEA (zmíněné v kapitole 8.5), kde je složitost návrhu obvodu I^O také rozdělena na $I \times O$ (kde I a O jsou počty obvodových vstupů a výstupů).

Oproti Millerovu algoritmu [7], který vyvíjí celý obvod se všemi obvodovými výstupy přímo, je tato varianta zřejmě efektivnější, ale také implementačně náročnější. Milleruv

algoritmus nepotřebuje zvláštní heuristiky pro dodatečné složení obvodu, ale změna jednoho hradla v něm může mít vliv na více obvodových výstupů (pravdivostních tabulek), což je při vývojem postupně zmenšované množině pozitivních změn nepříjemná vlastnost.

Každý obvodový výstup se vyvíjí po nastavený počet generací (například tisíc) nebo dokud nesplňuje zadání. Pak se výsledky uloží a vyvíjí se další obvodový výstup. Pokud nebylo řešení některého z nich nalezeno, dostane šanci v dalším cyklu. Bylo-li nalezeno, zapamatuje se a vývoj tohoto obvodového výstupu je až do fáze optimalizace pozastaven. Jsou-li známa řešení všech obvodových výstupů, je tento vývojový cyklus zakončen jejich složením pomocí *heuristiky* popsané v kapitole 9.7. Po té se vývoj buď ukončí nebo pokračuje dalším cyklem optimalizací na velikost obvodu, rychlost, případně obojí podle zadání (více v kapitole 9.5).

Pro inicializaci počáteční populace každého obvodového výstupu jsem použil metodu *Ramped Half-and-Half* a pro selekci podle *standardizované hodnoty fitness turnajovou selekci* (viz kapitoly 7.2, 6.3 a 6.4). Jako nahrazovací strategii jsem použil *steady-state* s *elitismem*, jež jsou popsány v kapitole 6.8 a kromě genetického operátoru křížení používám i mutaci.

9.7 Skládání obvodu

Nejlepší nalezená úplná (unikátní) řešení jednotlivých obvodových výstupů jsou průběžně ukládána do paměťových kontejnerů. Pokud známe alespoň jedno řešení od každého výstupu, proběhne na konci každého vývojového cyklu (popsaného v podkapitole 9.6) jejich složení dohromady. Zkouší se všechny kombinace všech řešení obvodových výstupů a nejlepší se zapíše do souboru s výsledky.

Zvolená kombinace obvodových výstupů se skládá s vynecháváním duplicitních hradel se stejnou výstupní hodnotou. Začíná se od řešení s nejvyšším stromem (obvodový výstup s největším zpožděním) a na něj se postupně přepojují vazby duplicitních hradel. Ostatní hradla se postupně přiřazují, až výsledné zapojení obsahuje řešení všech obvodových výstupů.

Tímto způsobem je také provedena dodatečná optimalizace, protože duplicitní výpočty jsou často přítomny i v řešení jednotlivých obvodových výstupů.

9.8 Validace, podmíněný překlad a watchdog

Pro urychlení fáze ladění programu a pozdější údržby obsahuje zdrojový kód kontrolní testy povoleného rozsahu hodnot klíčových proměnných. Tyto testy jsou však zabaleny do maker preprocesoru, takže díky jejich podmíněnému překladu běh výsledného programu nezpomalují.

Například jednoduché makro `RANGE(0,x,10)` čitelnost zdrojového kódu příliš nezatěžuje a přitom v pracovní verzi programu testuje zda se hodnota proměnné `x` pohybuje v intervalu $< 0, 10 >$. Pokud zjistí že ne, tak na standardní chybový výstup a do *logovacího souboru* vypíše jméno proměnné, soubor, funkci a řádek na kterém test se nachází, společně s konkrétní hodnotou a rozsahem který překročila.

K těmto testům jsem přidal i důkladný *test integrity* obvodu uloženého v paměti, rovněž podmínečně vkládaný za každou operaci měnící zapojení obvodu. Testuji například, zda v zapojení hradel nevznikl cyklus, který je u kombinačních obvodů nepřípustný, jestli jsou všechna hradla opět korektně obousměrně propojena nebo zda mají správnou výstupní hodnotu.

Pro snadnější ladění a provoz programu jsem vložil do evolučního cyklu *watchdog* výpis. Ten za každých pět minut normálního vývoje jednoho obvodového výstupu, po kterých nedojde k přepnutí na další, vypíše tečku. Vývoj každého obvodového výstupu se přepíná na další (implicitně) po 1000 generacích, které by neměly přesáhnout tuto dobu. Nicméně protože tato doba je závislá na velikosti populace, složitosti navrhovaného obvodu a výkonnosti cílového počítače, je lépe mít kontrolu, jestli se program někde definitivně „zacyklil“ a nebo jestli vývoj zdárně pokračuje (jen pomaleji než se předpokládalo). Stejně je ošetřena i fáze skládání obvodu, která by neměla přesáhnout 15 minut.

Kapitola 10

Experimenty

V této kapitole jsou nejprve implementované metody porovnány při návrhu a optimalizaci řešení jednodušších obvodů jako: úplná sčítačka, násobička 2x2 bity a multiplexor 4 na 1. Další experimenty jsou pak zaměřeny na porovnání úspěšnosti návrhu násobiček 3x3, 3x4 a 4x4 bity jednotlivými variantami implementovaných metod a CGP (implementovaného Ing. Vašíčkem) [19] již bez optimalizace nalezených řešení.

10.1 Úspěšnost optimalizace

Tato podkapitola popisuje návrh obvodů a hledání jejich optimálního řešení obecným i hybridním algoritmem. Složitost navrhovaných obvodů byla volena pro co nejvyšší úspěšnost hledání řešení oběma testovanými algoritmy a porovnání především rychlosti algoritmů pri optimalizaci nalezených řešení na zpoždění i velikost obvodu (viz kapitola 9.5).

Každá metoda byla otestována při velikosti populace 5 000 jedinců pro každý obvodový výstup a nebude-li řečeno jinak s množinou dvouvstupových hradel AND, OR, XOR (u násobičky 2x2 bity a multiplexoru 4 na 1 i s jednovstupovým NOT). Vývoj obvodu byl ukončen dosažením limitu 50 000 generací.

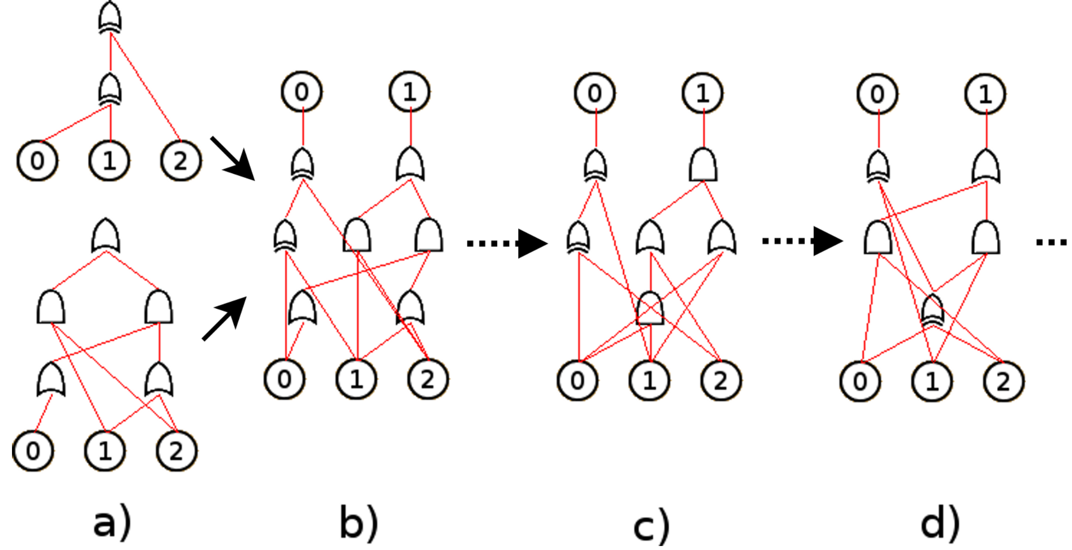
10.1.1 Úplná sčítačka

První test obecného a hybridního algoritmu jsem provedl na obvodu úplné bitové sčítačky (full adder). Funkční zapojení je oběma algoritmy vyvinuto zpravidla v jediné generaci během zlomku vteřiny. Zapojení obvodových výstupů, znázorněných v obrázku 10.1.a, bývá u takto triviálního obvodu často přítomno již v počáteční náhodné populaci.

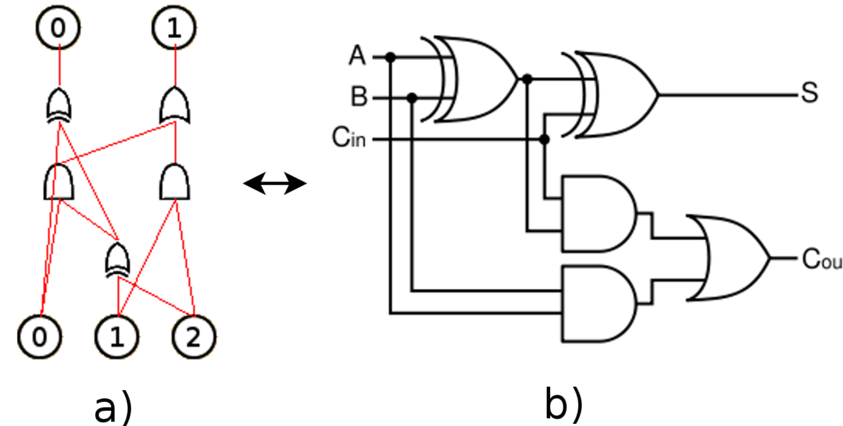
Nalezená řešení však bývají natolik odlišná, že heuristika většinou nenajde žádná společná (redundantní) hradla a tak jejich složením, na obrázku 10.1.b, nedojde k žádné optimalizaci. Nalezení lepšího řešení na obrázku 10.1.c, které má o hradlo méně pak trvá přibližně 3 000 generací (asi minutu). Optimální řešení z pěti hradel na obrázku 10.1.d, vznikne vhodným složením obvodových výstupů, kterých bývá pro úspěšné kombinování dostatečný výběr až po dalších 6 000 generacích.

Pokračováním ve vývoji vznikne další ekvivalentní řešení přibližně každých 3000 generací. Maximální počet v testech nalezených (třípatrových pětihradlových) řešení úplné sčítačky je 12. Jedno z nalezených optimálních řešení odpovídá i vzorovému zapojení ze stránek Wikipedie [27] (viz obrázek 10.2).

Pouhé tři obvodové vstupy a dva výstupy s řešením v jednotkách hradel, tvoří velmi malý stavový prostor přímo učebnicové úlohy. Vzhledem k hustotě pokrytí (navzorkování) tohoto



Obrázek 10.1: Příklad postupného vývoje úplné sčítačky v krocích a) b) c) a d).

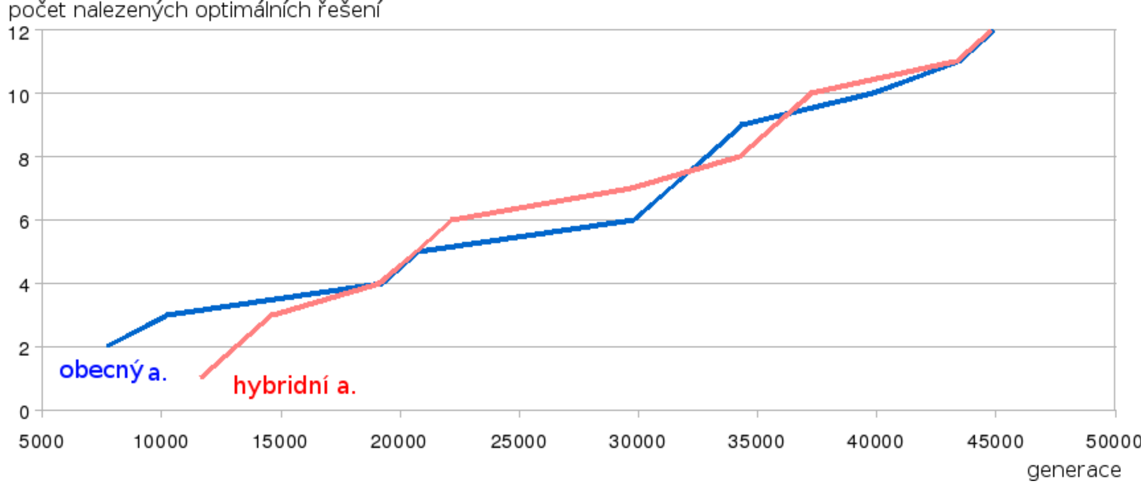


Obrázek 10.2: Jedno z nalezených zapojení a) odpovídá obvodu ze stránek Wikipedie b).

stavového prostoru se náhodně vytvořené počáteční populace a tím i průběhy vývoje obou algoritmů v grafu 10.3 prakticky neliší. Další algoritmy jsem proto zkoušel až na složitějších obvodech.

Všechny testy jsou prováděny s následujícími experimentálně stanovenými parametry[1], při kterých dosahovaly metody nejlepších výsledků:
Do turnajové selekce se vybírají 3 jedinci, rodičovská populace čítá 2000 jedinců, elitizmus je nastaven na 500 jedinců a podíl mutace na úkor křížení je 30%. Limit penalizace pro *stárnutí rodů* je u obecného algoritmu 50 křížení a u hybridního 100. U hybridního algoritmu je pro křížení obvodů požadována minimální množina pěti společných *bodů křížení.*

[1] Záznamy provedených experimentů, které jsou shrnuty v tabulkách 10.1 až 10.5, jsou uloženy na přiloženém CD (v adresáři „experimenty“) a jejich význam je podrobně popsán v příloze A. Pravdivostní tabulky testovaných obvodů jsou uvedeny v příloze B a jejich výpočet zaznamenán v adresáři „pravdivostni_tabulky_obvodu“ ve formátech ”ods” a ”xls”.



Obrázek 10.3: Průběh nacházení optimálních řešení úplné sčítačky obecným a hybridním algoritmem.

10.1.2 Násobička 2x2 bity

Násobička 2x2 bity patří stále k velmi jednoduchým obvodům, jak je vidět z obrázku 10.4. Proto jsem provedl jen pět běhů pro každou variantu implementovaných algoritmů[2] a výsledky zprůměroval do tabulky 10.1. Procentuální úspěšnost algoritmu v tabulce udává celkový počet nalezených optimálních řešení v pěti bězích.

Tabulka 10.1: Průběh návrhu a optimalizace násobičky 2x2 bity pomocí obecného i hybridního algoritmu a jejich variant.

algoritmus	úspěšnost	prům. počet generací vývoje	
		prvního zapojení	dvanácti optimálních z.
obecný	100%	17	19 723
obecný se stárnutím	100%	23	19 930
obecný se simulací	100%	20	18 504
obecný se simulací i stárnutím	100%	20	19 928
hybridní	100%	61	20 470
hybridní se stárnutím	93%	44	23 495
hybridní se simulací	100%	65	20 313
hybridní se simulací i stárnutím	97%	44	26 563

Nejrychlejší návrhy prvního zapojení násobičky má obecný algoritmus. Jeho rozšíření o stárnutí se v pouhých desítkách generací do prvního zapojení ještě nestihne projevit a při hledání optimálních variant nevykazuje zlepšení. Simulace křížení je při dalším hledání lepších řešení bez užitku, protože je aplikována pouze do nalezení prvotního řešení což u tohoto obvodu nastane prakticky ihned.

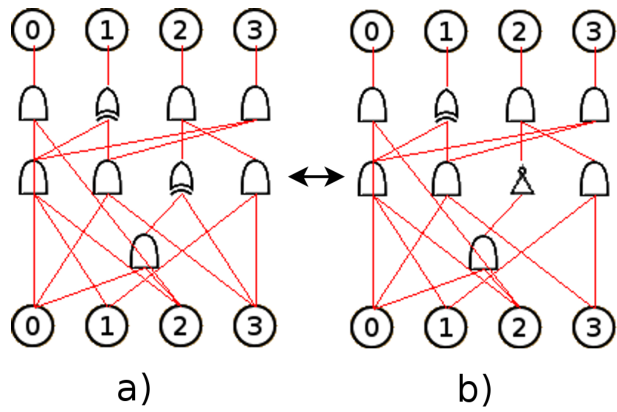
Všechny varianty hybridního algoritmu jsou výrazně méně efektivní, protože obvody příslušné jednotlivým obvodovým výstupům jsou tvořeny pouze několika hradly. V tak

[2] Varianty stárnutí a simulace k obecnému i hybridnímu algoritmu jsou popsány v kapitolách 8.6 a 9.4.

malých řešeních lze těžko najít dvojici s pěti různými hradly se stejnou nadřazenou strukturou, což je minimální požadovaný počet bodů křížení mezi rodiči a proto celý vývoj probíhá pouze zásluhou mutace.

Bohužel u hybridního algoritmu stárnutí způsobilo pokles úspěšnosti. Ve dvou případech bylo v limitu 50 000 generací nalezeno místo obvyklých 12 optimálních řešení pouze 10 (tj. 93% úspěšnost).

Zajímavé ale je, že oproti prvotnímu návrhu jsou na tom v průměrném počtu generací pro nalezení všech optimálních řešení varianty obecného i hybridního algoritmu dost podobně. Předpokládám že obyčejné křížení je zdaleka nejefektivnější právě v rané části vývoje a s klesající množinou pozitivních změn v obvodu stoupá i počet křížením vznikajících zmetků a proto je pak výhodnější hybridní křížení nebo samotná mutace.



Obrázek 10.4: Ukázka dvou možných zapojení a) a b) ze dvanácti nalezených optimálních řešení násobičky 2x2 bity.

10.1.3 Multiplexor 4 na 1

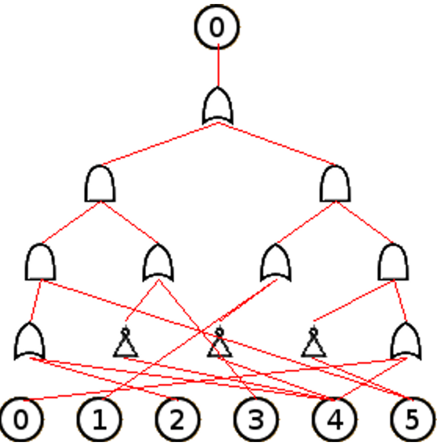
Pro návrh multiplexoru jsem zvolil deset běhů každého testu a výsledky opět shrnul do tabulky 10.2. Varianty algoritmů používající simulaci jsem z tohoto experimentu vynechal, protože jak je patrno z minulého testu, simulace nemá význam pro takto jednoduché obvody.

Tabulka 10.2: Úspěšnost optimalizace multiplexoru 4 na 1 pomocí obecného a hybridního algoritmu.

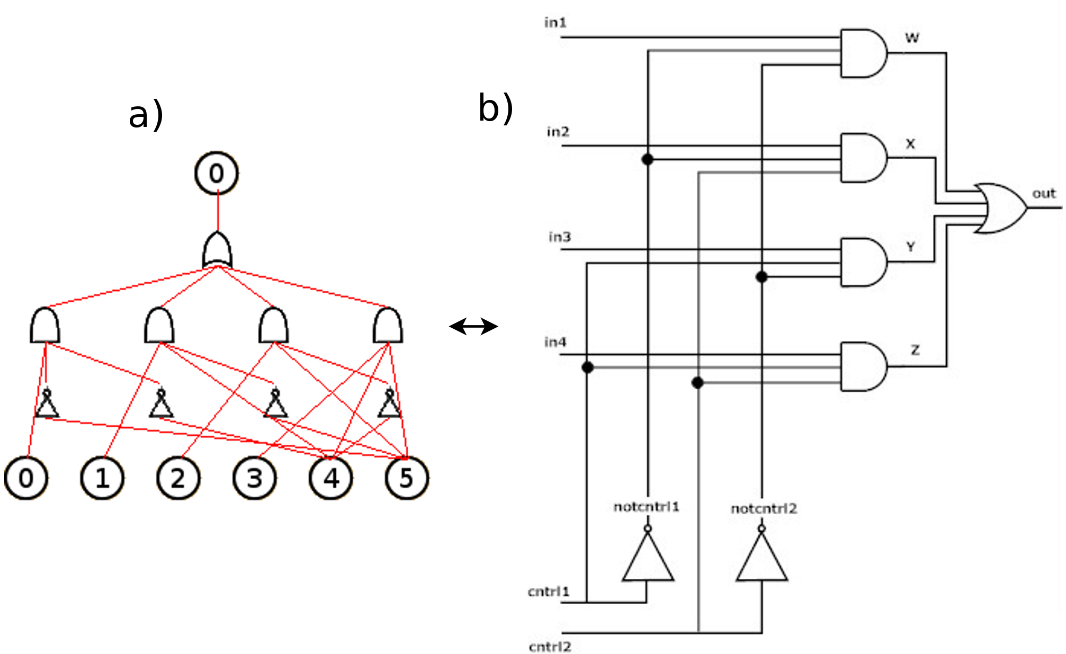
algoritmus	dosažená fitness				průměrná hodnota fitness
	411	412	413	horší	
obecný	20%	10%	40%	30%	432,3
obecný se stárnutím	10%	20%	40%	30%	442,3
hybridní	10%	20%	20%	50%	462,2
hybridní se stárnutím	0%	80%	20%	0%	412,2

Optimálního zapojení jedenácti hradel ve čtyřech patrech (hodnota fitness 411) dosáhl nejčastěji obecný algoritmus. Jeho varianta se stárnutím již byla o poznání horší a hybridní

algoritmus byl při vývoji jednoduchých řešení opět nejhorší. Hybridní algoritmus ve variantě se stárnutím sice nenašel v žádném z pokusů optimální řešení, zato ale dosáhl zdaleka nejlepších průměrných výsledků. Efekt stárnutí mu tedy přinesl rovnoměrnější prohledání stavového prostoru, ale bohužel na úkor jeho hloubky.



Obrázek 10.5: Jedno z nalezených řešení multiplexoru z nanejvýš dvouvstupových hradel.



Obrázek 10.6: Evoluční řešení multiplexoru z nanejvýš čtyřvstupových hradel a) odpovídající inženýrskému přístupu b).

Na obrázcích 10.5 a 10.6 jsou vidět výsledky evolučních návrhů s různými množinami použitých hradel. Vyvinutý multiplexor se tří a čtyřvstupými hradly je dokonce ekvivalentní konvenčnímu inženýrskému řešení. Není sice optimální (má dvě hradla NOT navíc) jako inženýrské, protože heuristika odstraňující duplicitní hradla se používá až při skládání více obvodových výstupů, ale to lze v další verzi programu napravit.

10.2 Úspěšnost návrhu

Pro porovnání úspěšnosti návrhu obvodů mnou implementovanými metodami a CGP jsem zvolil násobičky 3x3, 3x4 a 4x4 bity.

Každá metoda GP byla otestována stejně jako v minulých experimentech při velikosti populace 5 000 jedinců (2 000 nových v každé generaci) s množinou dvouvstupových hradel AND, OR, XOR. Vzniklí potomci větší než 100 hradel byli automaticky zahozeni a řešení obvodového výstupu přesahující 20 pater nebyla akceptována. Vývoj obvodu byl ukončen nalezením požadovaného zapojení nebo dosažením limitu 20 000 generací.

V případě CGP byla velikost populace ponechána na přednastavených pěti jedincích (čtyři nově vzniklí v každé generaci) a funkce bloků byla nastavena na AND, OR, XOR a propojka. Velikost mřížky jsem nastavil na 14x14 hradel pro co nejpodobnější rozsah prohledávaného stavového prostoru s experimenty GP. Limit počtu generací jsem adekvátně přepočítal podle počtu vytvářených jedinců na 100 000 generací. Také jsem drobně pozměnil implementaci, aby byl vývoj stejně jako v GP ukončen nalezením řešení a vynecháním fáze optimalizace se ušetřil čas. Kvalita navrhovaných obvodů je díky zcela odlišnému přístupu nesrovnatelná a není tak předmětem testu, protože má implementace GP skládá řešení jednotlivých výstupů v jeden obvod, zatímco CGP vyvíjí celý obvod najednou, díky čemuž dosahuje menšího počtu, ale zpravidla lepších výsledků.

Všechny experimenty byly provedeny na notebooku TOSHIBA Satellite 14F s 1GB RAM DDR2 a procesorem Intel Pentium Dual-Core při taktovací frekvencí 1.46 GHz. Experimenty s metodami GP byly provedeny na linuxovém operačním systému KUbuntu 7.10 s jádrem 2.6.22-14-generic a CGP na operačním systému windows XP. Pro orientační porovnání časové náročnosti algoritmů byly k počtům generací úspěšných a neúspěšných návrhů zaznamenány i naměřené časy. Výpisy o průběhu experimentů ze kterých jsou spočítány následující statistiky jsou opět uloženy na přiloženém CD v adresáři `experimenty`.

10.2.1 Násobička 3x3 bity

Násobička 3x3 bity je již dostatečně složitý obvod (cca 40 hradel) i pro navržený hybridní algoritmus. Nechal jsem provést 100 běhů každého algoritmu a výsledky uvedl do následující tabulky.

Tabulka 10.3: Úspěšnost a rychlost návrhu násobičky 3x3 bity.

algoritmus	úspěšnost	prům. počet generací	prům. počet ohodnocení	čas 1000 generací
obecný	65%	4 636	9 272 000	123,8s
obecný se stárnutím	64%	3 295	6 590 000	119,5s
obecný se simulací	66%	3 800	7 600 000	93,9s
obecný se simulací i stárnutím	76%	4 149	8 298 000	99,5s
hybridní	94%	1 695	3 390 000	205,3s
hybridní se stárnutím	94%	1 585	3 170 000	191,4s
hybridní se simulací	97%	1 726	3 452 000	150,1s
hybridní s oběma vylepšeními	98%	1 636	3 272 000	180,8s
CGP	98%	1 068 308	4 273 232	0,071s

Podle statistiky v tabulce 10.3 vychází z GP metod celkově nejlépe hybridní algorit-

mus s vylepšením o stárnutí a simulaci. Vliv stárnutí je vidět na variantách hybridního i obecného algoritmu. V obou případech vedl k významnému snížení průměrného počtu generací na návrh, ale bez zvýšení úspěšnosti. Simulace vedla k výrazné časové úspoře, až 40% v čase potřebném pro vyhodnocení 1 000 generací, ale jen k drobnému zvýšení úspěšnosti a v případě obecného algoritmu i drobnému snížení průměrného počtu potřebných generací pro úspěšný návrh. Společná kombinace stárnutí i simulace pak vedla k nejvyšší úspěšnosti a průměrnému času trvání 1 000 generací mezi samostatnými variantami stárnutí a simulace.

Generace hybridního algoritmu oproti obecnému trvá kvůli hledání vhodných partnerů a bodů křížení přibližně o polovinu déle, ale díky vyšší efektivitě jich je pro úspěšný návrh potřeba pouze třetinový počet. Podrobnější porovnání časové složitosti variant testovaných algoritmů a CGP na různých násobičkách je uvedeno v podkapitole 10.3.

Podle očekávání se experimenty s návrhy obvodů algoritmem GP a CGP těžko porovnávají. CGP potřebuje značný počet generací pro úspěšný návrh, ale díky minimální velikosti populace a efektivnosti maticové (oproti stromové) reprezentaci obvodu je vyhodnocuje extrémně rychle. Proto lze porovnat pouze úspěšnost návrhu a průměrný počet ohodnocených obvodů potřebných pro úspěšný návrh. Hybridní algoritmus s oběma vylepšeními dosahuje při návrhu tohoto obvodu stejné úspěšnosti jako CGP a v průměrném počtu vyhodnocení obvodů je dokonce ještě lepší. A to i přes to, že CGP vyhodnocuje celý obvod najednou, zatím co GP do této hodnoty počítá vyhodnocení každého obvodového výstupu zvlášť.

10.2.2 Násobička 3x4 bity

Násobička 3x4 má proti násobičce 3x3 pro každý obvodový výstup pravdivostní tabulku dvojnásobné velikosti. Pro výpočet těchto 128 bitových tabulek jsou již potřeba dva segmenty a tak i ohodnocení obvodů by mělo teoreticky trvat déle. Prakticky ale dochází k paralelizaci jejich výpočtu přímo v procesoru při určování pořadí prováděných instrukcí (technika Out-of-Order) [29], protože výpočet obou segmentů je nezávislý. Vyšší časovou náročnost experimentu jsem kompenzoval snížením počtu běhů každého algoritmu na 50.

Tabulka 10.4: Úspěšnost a rychlost návrhu násobičky 3x4 bity.

algoritmus	úspěšnost	prům. počet generací	prům. počet ohodnocení	čas 1000 generací
obecný	20%	12 186	24 372 000	154,2s
obecný se stárnutím	8%	7 294	14 588 000	146,6s
obecný se stárnutím2	12%	12 734	25 468 000	146,2s
obecný se simulací	26%	10 146	20 292 000	103,0s
obecný se simulací i stárnutím	14%	9 483	18 966 000	107,2s
hybridní	50%	7 582	15 164 000	346,7s
hybridní se stárnutím	50%	8 143	16 286 000	297,3s
hybridní se simulací	52%	8 249	16 498 000	277,1s
hybridní se simulací i stárnutím	66%	6 221	12 442 000	265,3s
CGP	94%	2 842 620	11 370 480	0,141s

Statistika z tabulky 10.4 v podstatě pokračuje ve stejném trendu jako tabulka 10.3. Hybridní algoritmus se stárnutím a simulací opět dosáhl z GP nejlepší úspěšnosti i nejmenšího

průměrného počtu generací potřebných pro návrh a simulace také výrazně urychlila běh algoritmu i lehce zvýšila jeho úspěšnost.

V případě obecného algoritmu vedlo stárnutí opět k významnému snížení průměrného počtu generací na návrh, ale tentokrát i drastickému snížení úspěšnosti. Příčinou poklesu úspěšnosti je zřejmě hranice stárnutí nezohledňující složitější návrh násobičky 3x4, na který je potřeba v průměru přibližně trojnásobný počet generací vývoje proti násobičce 3x3. Vyzkoušel jsem tedy i variantu obecného algoritmu s limitem stárnutí nastaveným z 50 křížení na 130 (v tabulce označenou jako obecný se stárnutím2). Zmírnění limitu sice úspěšnost zvýšilo o třetinu, ale průměrný počet generací vývoje návrhu se vrátil zpět na úroveň obecného algoritmu. Kombinace stárnutí i simulace (s původním limitem 50 křížení) přinesla průměrné výsledky obou, tedy i snížené úspěšnosti.

V případě hybridního algoritmu již samotné stárnutí ani simulace nedosahují výrazných rozdílů a tak jsem posun limitu stárnutí u hybridního algoritmu nezkoušel.

CGP ale dosahuje na složitějších obvodech vyšší úspěšnosti a co se týče průměrného počtu ohodnocení, tak srovnatelné výsledky již dává jen hybridní algoritmus s oběma vylepšeními.

10.2.3 Násobička 4x4 bity

Násobička 4x4 bity je v této práci nejsložitějším navrhovaným obvodem. Je definován osmi 256-bitovými pravdivostními tabulkami, které GP vyhodnocovalo po čtyř 64-bitových segmentech a CGP dokonce po osmi 32-bitových. Kvůli náročnosti tohoto návrhu jsem musel rozšířit prohledávaný stavový prostor posunem limitu uvažovaných řešení (každého obvodového výstupu) na 200 hradel pro algoritmus GP a adekvátně i mřížku CGP na 15x20 hradel. Také jsem desetkrát zvýšil limit počtu generací na 200 000 pro GP a odpovídajících 10 000 000 pro CGP. Rovněž vzhledem k extrémní časové náročnosti tohoto experimentu (v řádu stovek hodin) jsem si mohl dovolit plně otestovat 10 běhů jen dvou metod. Zvolil jsem proto nejlepší variantu GP z minulých experimentů a CGP. U ostatních metod jsem změřil jen čas potřebný pro vyhodnocení 1000 generací.

Tabulka 10.5: Úspěšnost a rychlost návrhu násobičky 4x4 bity.

algoritmus	úspěšnost	prům. počet generací	prům. počet ohodnocení	čas 1000 generací
obecný	–	–	–	256s
obecný se stárnutím	–	–	–	236s
obecný se simulací	–	–	–	161s
obecný se simulací i stárnutím	–	–	–	159s
hybridní	–	–	–	614s
hybridní se stárnutím	–	–	–	561s
hybridní se simulací	–	–	–	479s
hybridní se simulací i stárnutím	50%	87 318	174 636 000	519s
CGP	100%	52 800 056	211 200 224	0,382s

Jak je vidět z výsledků v tabulce 10.5 s rozšířeným stavovým prostorem dosáhl algoritmus CGP při návrhu násobičky 4x4 bity vyšší úspěšnosti než při návrhu násobičky 3x4 nebo 3x3 bity. V průměrném počtu ohodnocených obvodů během úspěšného návrhu je na tom GP sice o poznání lépe než CGP, ale v úspěšnosti se mu již nemůže rovnat.

10.3 Shrnutí výsledků úspěšnosti návrhů

Pro lepší přehlednost celkových výsledků vybraných metod jsem udělal celkový souhrn v následující tabulce. Chybí zde pouze test návrhu násobičky 4x4 bity obecným GP, který jsem z časových důvodů nemohl provést.

Tabulka 10.6: Souhrn experimentů s návrhem bitových násobiček.

návrh obvodu	I / O	metoda	úspěšnost	prům. počet ohodnocení
násobička 2x2 bity	4 / 4	obecný algotitmus	100%	34 000
násobička 3x3 bity	6 / 6		65%	9 272 000
násobička 3x4 bity	7 / 7		20%	24 372 000
násobička 4x4 bity	8 / 8		–	–
násobička 2x2 bity	4 / 4	hybridní algotitmus se stárnutím i simulací	100%	88 000
násobička 3x3 bity	6 / 6		98%	3 272 000
násobička 3x4 bity	7 / 7		66%	12 442 000
násobička 4x4 bity	8 / 8		50%	174 636 000
násobička 2x2 bity	4 / 4	CGP	100%	29 964
násobička 3x3 bity	6 / 6		98%	4 273 232
násobička 3x4 bity	7 / 7		94%	11 370 480
násobička 4x4 bity	8 / 8		100%	211 200 224

Pokud vezmeme hodnotu průměrného počtu ohodnocení potřebných pro daný návrh z tabulky 10.6 jako měřítko náročnosti tohoto návrhu, vidíme, že při zhruba trojnásobném zvýšení nároků násobičky 3x4 bity proti násobičce 3x3 bity poklesla úspěšnost obecného algoritmu přibližně třikrát. Proto předpokládám že při dalším více než desetinásobném zvýšení nároků při přechodu na násobičku 4x4 bity (zjištěném na hybridním i CGP algoritmu) poklesne úspěšnost obecného algoritmu natolik, že tento návrh již prakticky nebude obecným algoritmem možný.

Oproti tomu hybridní algoritmus GP rozšířený o stárnutí a simulaci dosáhl na návrhu násobičky 3x3 bity o 33% vyšší úspěšnosti než obecný algoritmus, na násobičce 3x4 bity již o 46% a násobičku 4x4 bity dokonce zvládl navrhnout také, čímž prokázal výrazné zlepšení proti obecnému algoritmu. Až na příliš jednoduché obvody (jako je násobička 2x2 bity) dosáhl proti obecnému algoritmu i polovičního až třetinového počtu potřebných ohodnocení na návrh.

Úspěšnost CGP je sice stále ještě nejvyšší, hybridní algoritmus se mu už ale vyrovná alespoň v průměrném počtu vyhodnocených obvodů na úspěšný návrh.

Kapitola 11

Závěr

Nastudoval jsem problematiku evolučního návrhu obvodu a seznámil se s technikou šíření schémat v evolučním algoritmu. Pokusil jsem se podat ucelený přehled různých typů evolučních algoritmů, zvláště pak GA, GP a CGP. Na základě zjištěných charakteristických vlastností stavebních bloků a jejich odlišného zpracování jsem navrhl, implementoval a otestoval hybridní algoritmus GA a GP, společně se dvěma dalšími vylepšeními použitelnými i pro obecný algoritmus GP.

Na provedených experimentech s bitovými násobičkami jsem prokázal pozitivní vliv implementovaných vylepšení obecného i hybridního algoritmu GP. Hybridní algoritmus navíc proti obecnému dosáhl ve všech zkoušených variantách na dostatečně složitých obvodech výrazně lepších výsledků.

Experimenty s násobičkami jsem zopakoval i s alternativní metodou návrhu CGP, která se pro evoluční návrh logických obvodů používá nejčastěji. Pro její běh je sice potřeba vhodné nastavení mřížky na daný problém, pak ale obvykle dosahuje lepších výsledků než genetické programování.

Navržený algoritmus GP lze samozřejmě vyzkoušet i na symbolickou regresi nebo dále vylepšit pro návrh obvodů zohledněním odhadu dopadu změny hradla při výběru bodů křížení, ale lepší výsledky by mohlo přinést zapracování odzkoušených vylepšení GP do CGP. Například nepřepočítávat vždy celou matici, ale jen změněnou a navazující část nebo odvozovat míru mutace jednotlivých hradel podle statistiky jejich vlivu na funkci obvodu. Také lze uvažovat o paměti nalezených řešení a v případě uváznutí se k nim vracet.

Literatura

[1] Kvasnička, V., Pospíchal, J., Tiňo, P.: *Evolučné algoritmy.* STU Bratislava, 2000.

[2] Flegr, J.: *Zamrzlá evoluce, aneb je to jinak pane Darwin.* Praha, Nakladatelství Academia, 2006.

[3] Jelínek J., Zicháček V.: *Biologie pro gymnázia.* Nakladatelství Olomouc, 1999.

[4] Bidlo, M.: *Biologií inspirovaný vývin jako technika evolučního návrhu*, In: Kognice a umělý život VII, Opava, CZ, SLU, 2007, s. 43-53.

[5] Poli, R.: *Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory*, In: Genetic Programming, Proceedings of EuroGP'2000, Edinburgh, EN, 2000, s. 163–180. Dokument dostupný na URL citeseer.ist.psu.edu/poli00hyperschema.html (3.2008).

[6] Drechsler, R., Gunther, W.: *Evolutionary Synthesis of Multiplexor Circuits under Hardware Constraints*, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Las Vegas, Nevada, USA, 2000, s. 513–518. Dokument dostupný na URL citeseer.ist.psu.edu/drechsler00evolutionary.html (3.2008).

[7] Liu, R., aj.: *An Efficient Multi-Objective Evolutionary Algorithm for Combinational Circuit Design*, In: First NASA/ESA conference on Adaptive hardware and Systems (AHS-2006), Istanbul, Turkey, 2006, s. 215–221.

[8] Ošmera, P.: *Genetické algoritmy a jejich apliakce*, [Habilitační práce], Vysoké učení technické v Brně, 2001.

[9] Hendler, J.: *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*, [Disertační práce], University of Maryland, 2000.

[10] Pošík, P.: *Paralelní genetické algoritmy*, [Diplomová práce], České vysoké učení technické Praha, 2001.

[11] Štefka, D.: *Alternativy k evolučním optimalizačním algoritmům*, [Diplomová práce], České vysoké učení technické Praha, 2005.

[12] Žaloudek, L.: *Evoluční návrh obvodů na úrovni tranzistorů*, [Diplomová práce], Vysoké učení technické v Brně, 2007.

[13] Zelenka, J.: *Využití genetického programování pro konstruktivní uvažování*, 2007. Dokument dostupný na URL http://hilbert.chtf.stuba.sk/KUZVII/abstracts/Zelenka.pdf (11.2007).

[14] Váňová, I.: *Úvod do genetických algoritmů*, 2007. Dokument dostupný na URL http://vanova.org/texty/geneticke_algoritmy.pdf (11.2007).

[15] Faridani, S.: *Using Genetic Algorithm for Product Family Optimization*, 2007. Dokument dostupný na URL http://eml.ou.edu/siamak/ga/forwebsite/report.htm (11.2007).

[16] Sekanina, L., Drábek, V.: *Evolvable hardware - evoluce na čipu*, 1999. Dokument dostupný na URL http://www.elektrorevue.cz/clanky/99005/index.html (11.2007).

[17] Schwarz, J.: *Aplikované evoluční algoritmy*, Vysoké učení technické v Brně, 2006. Dokument dostupný na URL http://www.fit.vutbr.cz/study/courses/EVO (11.2007).

[18] Sekanina, L.: *Biologií inspirované počítače*, Vysoké učení technické v Brně, 2007. Dokument dostupný na URL http://www.fit.vutbr.cz/study/courses/BIN (11.2007).

[19] Vašíček, Z.: *Biologií inspirované počítače - laboratorní cvičení 1*, Vysoké učení technické v Brně, 2008. Dokument dostupný na URL http://www.fit.vutbr.cz/ vasicek/courses/bin_lab1/ (04.2008).

[20] Zbořil, F.: *Soft Computing*, Vysoké učení technické v Brně, 2007. Dokument dostupný na URL http://www.fit.vutbr.cz/study/courses/SFC (04.2008).

[21] Fučík, O.: *Návrh číslicových systémů*, Vysoké učení technické v Brně, 2007. Dokument dostupný na URL http://www.fit.vutbr.cz/study/courses/INC (04.2008).

[22] Wakerly, J., F.: *IEEE standard symbols apendix A*, 1999. Dokument dostupný na URL http://www.ddpp.com/DDPP3_pdf/IEEEsyms.pdf (04.2008).

[23] NASA - Computational Sciences Division*Automated Antenna Design*, 2008. Dokument dostupný na URL http://ti.arc.nasa.gov/projects/esg/research/antenna.htme (04.2008).

[24] *THE 2006 "HUMIES" AWARDS*, 2008. Dokument dostupný na URL http://www.genetic-programming.org/hc2006/cfe2006.html (04.2008).

[25] Wikipedie: *Otevřená encyklopedie*, 2007. Dokument dostupný na URL http://wikipedia.org/ (11.2007).

[26] Wikipedie: *Optimalizace*, 2008. Dokument dostupný na URL http://cs.wikipedia.org/wiki/Optimalizace (04.2008).

[27] Wikipedie: *Adder (electronics)*, 2008. Dokument dostupný na URL http://en.wikipedia.org/wiki/Adder_(electronics) (04.2008).

[28] *No Free Lunch Theorems*, 2008. Dokument dostupný na URL http://www.no-free-lunch.org (04.2008).

[29] Svět hardware *Intel Core - pohled na architekturu I - Out-of-Order zpracování*, 2006. Dokument dostupný na URL http://www.svethardware.cz/art_doc-4366B9A529A3DB85C125716000369FCB.html (04.2008).

Slovníček pojmů

Altruistické chování je opakem sobeckého, jedinec koná pro dobro ostatních i když ho to znevýhodňuje.

CLB (Configurable Logic Block) je pole libovolně propojitelných logických buněk s programovatelnou funkcí.

Development je evoluční technika vývoje chromozomu obsahujícího instrukce pro vývoj fenotypu.

Digitální obvod pracuje s digitálním signálem. Tedy signálem, který je vzorkovaný a zároveň kvantovaný, tj. je tvořen posloupností vzorků, které mohou nabývat pouze omezeného počtu hodnot.

Diploidní buňky mají dvě sady párových chromozomů. Například člověk má v každé buňce (mimo pohlavní) 23 párů chromozomů, jeden z párů od otce a druhý od matky.

Diverzita populace nebo genů znamená rozmanitost jedinců případně variant genů v populaci.

DNA (Deoxyribonucleic acid) je nositelkou genetické informace všech živých organismů.

ESA (European Space Agency) je mezivládní organizace pro využití vesmíru, která má v současnosti 17 členských států.

EMOEA (Efficient Multi-Objective Evolutionary Algorithm) je algoritmus vybírající body křížení podle jejich vlivu na fitness hodnotu obvodu.

Emulace napodobení činnosti jednoho zařízení pomocí jiného

FPAA (Field Programmable Analog Arrays) je rekonfiguravatelný obvod s volitelným zapojením pole operačních zesilovačů nebo tranzistorů.

FPGA (Field Programmable Gate Arrays) je rekonfiguravatelný obvod na bázi hradlových polí, které lze libovolně propojovat.

FPTA (Field Programmable Transistor Arrays) je rekonfiguravatelný obvod s volitelným zapojením pole tranzistorů (podmnožina FPAA).

Globální extrém funkce $f(x)$ je bod, ve kterém je funkční hodnota nejvyšší, resp. nejnižší pro celý definiční obor funkce.

GUI (Grafic user interface) je grafické uživatelské rozhraní, neboli grafická podoba ovládání programu.

Haploidní buňky mají jen jednu sadu chromozomů. Vyskytují se u primitivních rostlin nebo v pohlavních buňkách živočichů.

Heuristika v informatice je postup, dávající orientační informace o řešení daného problému, které pak mohou vést k jeho nalezení.

Inkrementální evoluce rozděluje systém na jednodušší subsystémy, které se navrhují samostatně a potom spojí.

Kardinalita množiny (také mohutnost množiny) je pojmem teorie množin, vyjadřující velikost konečných ale i nekonečných množin.

Klamné optimum (viz lokální extrém)

Konvergence je pojem označující sbíhavost, sbližování, popř. vývoj, který vede ke sblížení.

Kvantové jevy jsou interakce elementárních částic. Například vzájemné působení dvou protichůdných elektronů v sousedních vodičích, které negativně ovlivňuje rychlost šíření těchto částic.

Kvazináhodný výběr je náhodný výběr s nerovnoměrně rozloženou pravděpodobností.

Lokální extrém funkce $f(x)$ je bod, ve kterém je funkční hodnota vyšší (lokální maximum) či nižší (lokální minimum) než funkční hodnota v okolních bodech.

Mohutnost množiny (viz. kardinalita množiny)

NASA (National Aeronautics and Space Administration) je americká vládní agentura zodpovědná za americký kosmický program a všeobecný výzkum v oblasti letectví.

Předčasná konvergence znamená ukvapené označení nejlepšího řešení, které je přitom nejlepší pouze v dané oblasti.

Pseudoparalelismus je paralelizmus simulovaný rychlým přepínáním sériově běžících procesů.

Saturace populace jedním jedincem má význam nasycení, kdy daný jedinec dosáhl maximálního počtu kopií v populaci.

Selekční tlak je tlak, kterým působí prostředí nebo člověk na určitou populaci tím, že z ní odstraňuje nositele určitého znaku.

Simulace je napodobování dějů a procesů obvykle v rámci zjednodušeného modelu.

SPICE (Simulation Program with Integrated Circuit Emphasis) je simulátor používaný pro návrh obvodů.

Symbolická regrese je hledání funkce aproximující množinu zadaných bodů (matematická obdoba úlohy z IQ-testu: doplnění chybějícího čísla v číselné řadě).

Škálovatelnost znamená proměnlivost, volitelnost nebo přizpůsobitelnost.

ÚNDF (Úplná Normální Disjunktní Forma) je výraz ve tvaru součtu součinů booleovských proměnných.

Watchdog (z angličtiny – „hlídací pes“) je původně počítačová periferie, která resetuje systém při jeho zaseknutí. U softwaru označuje hlídaný časový limit programové smyčky nebo změny hlídané proměnné.

Záporná zpětná vazba nastává pokud zvýšení hodnoty, přiváděné z výstupu na vstup, způsobí snížení hodnoty na výstupu. Používá se ke stabilizaci systému.

Dodatek A

Uživatelská příručka k programům

A.1 Implementace obecného a hybridního algoritmu

Oba programy jsou C++ konzolové multiplatformní aplikace algoritmu genetického programování s volitelnými rozšířeními o stárnutí rodů a simulaci křížení.

Vstupem obou programů je hlavičkový soubor s parametry evoluce a pravdivostní tabulkou pro každý výstup požadovaného obvodu (příklad v části A.2). Proto je nutná rekompilace po každé změně tohoto zadání. Výhodou je optimalizace spouštěného programu pro každé zadání, snadná přenositelnost mezi počítači i operačními systémy a jednoduchost. Nepotřebuje žádné GUI ani ošetřování chyb při načítání zadání ze souboru.

Výstupem programu jsou průběžné informační výpisy o průběhu evoluce. Každý řádek výstupu odpovídá jednomu cyklu vývoje, ve kterém se postupně vyvíjely všechny obvodové výstupy:

```
0.  .96%  17b  301s,   |      1000g     301s | -
1.   1    16b  103s,   |      1043g     404s | 516  79% 1
2.   1    14b  206s,   |      2559g     602s | 414  99% 1
3.   2    14b  202s,   |      3973g     804s | 414  99% 2
```

Řádek začíná pořadovým číslem cyklu (`0.`), následovaným trojicí hodnot pro každý obvodový výstup:

- Pokud řešení obvodového výstupu doposud nebylo nalezeno, tak vypíše z kolika procent splňuje nejlepší doposud nalezené řešení zadání (`96%`), průměrný počet hradel jedince v populaci (`17b`) a čas vývoje v tomto cyklu (`301s`). Překročí-li doba vývoje tisíce generací mezi výpisy pět minut, je vypsána kontrolní tečka signalizující zdárné pokračování ve vývoji tohoto obvodového výstupu (watchdog) a zmíněná trojice hodnot charakterizující jeho vývoj až po dokončení tisíce generací.

- Pokud bylo řešení nalezeno, tak je vypsán počet nalezených (nejlepších originálních) řešení (`1`), jejich velikost (`16b`) a čas vývoje v tomto cyklu (`1s`).

Na konci každého cyklu za značku '`|`' program vypíše celkový počet proběhlých generací (dohromady ve všech populacích obvodových výstupů) a celkový čas od spuštění programu (`1000g   601s`). Nejsou-li známa řešení všech obvodových výstupů, vypíše `| -` a pokračuje dalším cyklem. Jinak provede ještě jejich složení v jeden obvod a vypíše hodnotu

fitness složeného obvodu (`516`), z kolika procent odpovídá (nastavené `BEST_FITNESS`) nejlepší dosažitelné hodnotě (`79%`) a počet nalezených řešení (`1`). Z fitness složeného obvodu lze (při nastavení optimalizace na `MINIMAL_BOTH`) vyčíst i počet pater a hradel obvodu. Například hodnota 516 udává 5 pater a 16 hradel.

Zápis výsledků se provádí v textové podobě do zadaného výstupního souboru (implicitně `output.txt`). Zapojení obvodu se všemi obvodovými výstupy je zaznamenáno na jeden řádek a pro lepší přehlednost je doplněno i řádky se zapojením každého obvodového výstupu zvlášť. Celý záznam je ukončen prázdným řádkem.

Záznam jednoho obvodového výstupu na řádku vypadá například takto:

```
(0x8063fac, 0, 0x8057c90, 0x8057c20)|(0x8057c20, 4)(0x8057c90, 6)
```

Závorky oddělují jednotlivé prvky a značky '`|`' patra. Nejprve je vypsán kořenový prvek a poslední jsou listové prvky představující vstupy obvodu. Každý prvek má nejprve svůj identifikátor, číslo s typem prvku a seznam identifikátorů prvků na které je zapojen.

Pro grafické zobrazení těchto výsledků evoluce jsem napsal program „TreeViewer“, popsaný v příloze A.3.

Pro překlad zvolené varianty, spuštění a záznam statistiky (implicitně) 100 testů jsou připraveny pro oba algoritmy čtyři *bash skripty* (`spustit_*.sh`). Lze je spustit po překopírování zdrojových souborů z přílohy na školní server merlin, nebo jiný unixový systém.

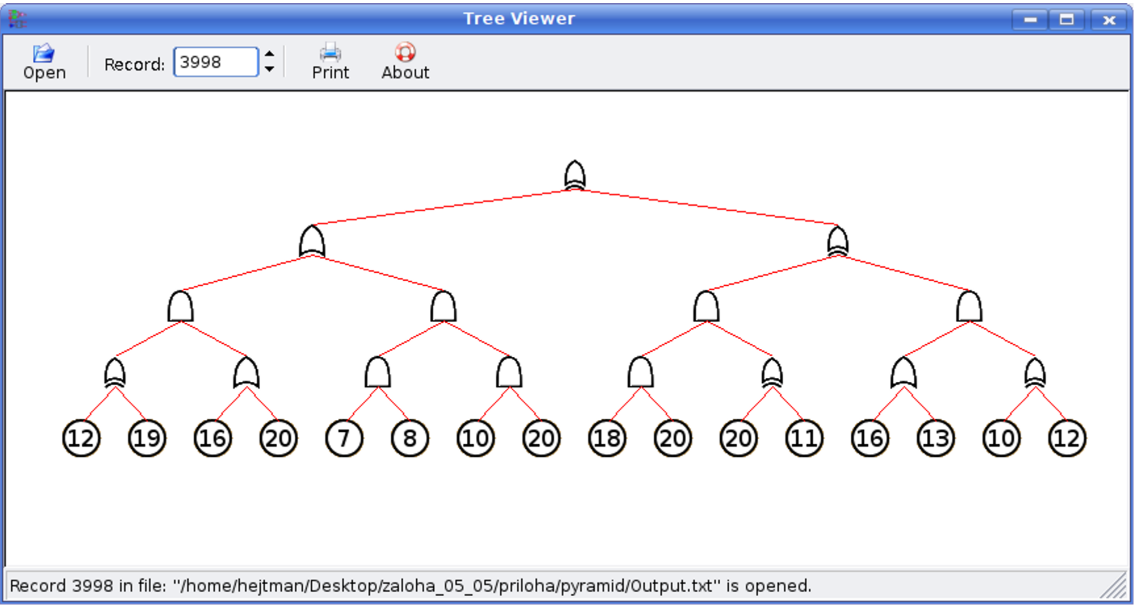
A.2 Příklad konfiguračního souboru "input.h"

```
#ifndef INPUT_H
#define INPUT_H


//-------------------------------------------------------------------------------------------------------------------//
// Nastavitelne parametry genetickeho algoritmu:                                                                    //
//-------------------------------------------------------------------------------------------------------------------//
namespace GEN
{
    const uint GENERATION    = 50000;                                             // Generacni omezeni evoluce (0 vypnuto)
    const uint SWITCH_STEP   = 1000;                                              // Pocet gen. vyvoje kazdeho nevyvinuteho
                                                                                  // vystupu.
    const uint POPULATION    = 5000;                                              // Velikost populace pro kazdy vystup
    const uint CROSS_SIZE    = 2000;                                              // Pocet jedincu ke~krizeni (sudy pocet!)
    const uint TURNAJ_SIZE   = 3;                                                 // Pocet jedincu ucastnicich se~turnaje
                                                                                  // (minimum 2)
    const uint ELITISM       = 500;                                               // Pocet preferovanych nejlepsich jedincu
    const uint MUTATION      = 300;                                               // Pravdepodobnost mutace v~promile
}                                                                                 // stejnym poctem prvku.

enum EOptimize{   MINIMAL_BOXS, MINIMAL_LEVELS, MINIMAL_BOTH, NONE   };


//-------------------------------------------------------------------------------------------------------------------//
// Nastavitelne parametry pozadovaneho obvodu:                                                                      //
//-------------------------------------------------------------------------------------------------------------------//
namespace SET                                                                     //  PARAMETRY POZADOVANEHO OBVODU
{
    typedef unsigned long long int ull;                                           //  Definice datoveho typu segmentu
                                                                                  //  Lze pouzit i uint64_t z <sys/types.h>
    const char NAME[] = "nasobicka 4x4 bity";                                     //! Nazev obvodu

    const unsigned int INPUTS   = 8;                                              //! pocet vstupu obvodu
    const unsigned int OUTPUTS  = 8;                                              //! pocet vystupu obvodu
    const unsigned int L_BACK   = 10;                                             //! uroven konektivity (minimum 1)
                                                                                  //  Pocet vstupu pokryvanych jednim
    const unsigned int SEG_COVER = 6;                                             //  segmentem (log(8*sizeof(ull))/log(2))

    const unsigned int SEGMENTS = (INPUTS <= SEG_COVER) ? 1                       //  pocet segmentu nutnych na~pokryti
                                          : 1 << (INPUTS - SEG_COVER);//  vsech vstupu pravdivostni tabulky

    const unsigned int MAXIMUM_BOXS       = 300;                                  //! Horsi reseni obv. vystupu nepripusti
    const unsigned int BEST_FITNESS       = 1000;                                 //! Nejlepsi dosazena fitness
    const unsigned int ACCEPTABLE_FITNESS = 2100;                                 //! Horsi reseni obvodoveho vyst.nezapise
    const EOptimize    OPTIMIZE           = NONE;                                 //! Neoptimalizuje (ukoci vyvoj nalezenim)


    const ull TRUTH_TABLE[OUTPUTS][SEGMENTS] = {                                  //! Pravdivostni tabulka pro kazdy vystup
     { 0xAAAA0000AAAA0000ull, 0xAAAA0000AAAA0000ull, 0xAAAA0000AAAA0000ull, 0xAAAA0000AAAA0000ull }, //!
     { 0x6666AAAACCCC0000ull, 0x6666AAAACCCC0000ull, 0x6666AAAACCCC0000ull, 0x6666AAAACCCC0000ull }, //!
     { 0x1E1E66665A5AAAAAull, 0xB4B4CCCCF0F00000ull, 0x1E1E66665A5AAAAAull, 0xB4B4CCCCF0F00000ull }, //!
     { 0x01FE1E1E39C36666ull, 0x6B925A5A55AAAAAAull, 0xAB54B4B4936CCCCCull, 0xC738F0F0FF000000ull }, //!
     { 0x5554AB545294B4B4ull, 0x4924936C66CCCCCCull, 0x3398C7381C70F0F0ull, 0x07C0FF0000000000ull }, //!
     { 0x999833986318C738ull, 0x8E381C7078F0F0F0ull, 0xC3E007C01F80FF00ull, 0xF800000000000000ull }, //!
     { 0xE1E0C3E083E007C0ull, 0x0FC01F807F00FF00ull, 0xFC00F800E0000000ull, 0x0000000000000000ull }, //!
     { 0xFE00FC00FC00F800ull, 0xF000E00080000000ull, 0x0000000000000000ull, 0x0000000000000000ull }, //!
    };
}


#endif // INPUT_H
```

A.3 TreeViewer

Program zobrazující v grafické podobě textové výsledky evoluce z výstupních souborů obecného a hybridního algoritmu. GUI programu je postaveno na krosplatformním toolkitu wxWidgets a program je opět přeložitelný i na serveru merlin. Pro vzdálené spuštění s grafickým výstupem z tohoto serveru je potřeba se přihlásit například přes X-terminál:

```
ssh -X xlogin@merlin.fit.vutbr.cz xterm
```



Obrázek A.1: Screenshot programu TreeViewer zobrazující náhodný binární pětipatrový strom vytvořený metodou Full v programu Pyramid.

Bohužel tento program neumí rozpoznávat a měnit množiny použitých hradel a tak je pro zobrazení záznamů s jinou množinou potřeba také zásah do kódu a rekompilace. Proto je na přiloženém CD skompilován ve třech variantách s různými množinami hradel.

A.4 Pyramid

Program testující závislost změny funkce obvodu na pozici měněného hradla. Jedná se o triviální C++ konzolovou aplikaci řízenou pomocí konstant v souboru `main.cpp`. Výstupem programu je níže popsaná statistika a soubor s náhodnými stromy zpracovanými během experimentu, které jsou zobrazitelné programem TreeViewer.

Metodou Full generuje binární náhodné stromy zadané hloubky z hradel AND, OR, XOR (viz obrázek A.1). V každém patře každého stromu změní hradlo, přepočítá funkci stromu a porovná ji s původní funkcí. Provede (implicitně) tisíc pokusů a zobrazí průměrný počet vyvolaných změn pro každé patro:

```
prumer zmen v patre 0:   32422
prumer zmen v patre 1:   20931
prumer zmen v patre 2:   14213
prumer zmen v patre 3:   9468
```

Dodatek B

Pravdivostní tabulky

Obsáhlejší výpočet následujících pravdivostních tabulek je uveden na přiloženém CD ve formátech "ods" a "xls". Jejich funkce jsou uvedeny ve zkráceném hexadecimálním tvaru, kde každý řádek definuje chování jednoho obvodového výstupu ve všech kombinacích vstupních hodnot (viz. tabulka 9.1).

1. **Úplná jednobitová sčítačka (FullAdder)** – 3 vstupy, 2 výstupy, pro vyhodnocení pravdivostní tabulky je potřeba jeden 64-bitový segment:

```
0x96
0xE8
```

2. **Multiplexor 4 na 1** – 6 vstupů, 1 výstup, potřebuje jeden 64-bitový segment:

```
0xFF00F0F0CCCCAAAA
```

3. **Násobička 2x2 bity** – 4 vstupy, 4 výstupy, potřebuje jeden 64-bitový segment:

```
0xA0A0
0x6AC0
0x4C00
0x8000
```

4. **Násobička 3x3 bity** – 6 vstupů, 6 výstupů, potřebuje jeden 64-bitový segment:

```
0xAA00AA00AA00AA00
0x66AACC0066AACC00
0x1E6655AAB7CCF000
0x54B46CCC38F00000
0x983870F0C0000000
0xE0C0800000000000
```

5. **Násobička 3x4 bity** – 7 vstupů, 7 výstupů, potřebuje dva 64-bitové segmenty:

```
0xAAAA0000AAAA0000 AAAA0000AAAA0000
0x6666AAAACCCC0000 6666AAAACCCC0000
0x1E1E66665A5AAAAA B4B4CCCCF0F00000
0xAB54B4B4936CCCCC C738F0F0FF000000
0x3398C7381C70F0F0 07C0FF0000000000
0xC3E007C01F80FF00 F800000000000000
0xFC00F800E0000000 0000000000000000
```

6. **Násobička 4x4 bity** – 8 vstupů, 8 výstupů, potřebuje čtyři 64-bitové segmenty:

```
0xAAAA0000AAAA0000 AAAA0000AAAA0000 AAAA0000AAAA0000 AAAA0000AAAA0000
0x6666AAAACCCC0000 6666AAAACCCC0000 6666AAAACCCC0000 6666AAAACCCC0000
0x1E1E66665A5AAAAA B4B4CCCCF0F00000 1E1E66665A5AAAAA B4B4CCCCF0F00000
0x01FE1E1E39C36666 6B925A5A55AAAAAA AB54B4B4936CCCCC C738F0F0FF000000
0x5554AB545294B4B4 4924936C66CCCCCC 3398C7381C70F0F0 07C0FF0000000000
0x999833986318C738 8E381C7078F0F0F0 C3E007C01F80FF00 F800000000000000
0xE1E0C3E083E007C0 0FC01F807F00FF00 FC00F800E0000000 0000000000000000
0xFE00FC00FC00F800 F000E00080000000 0000000000000000 0000000000000000
```