VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

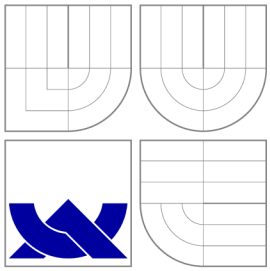
MULTIKRITERIÁLNÍ KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

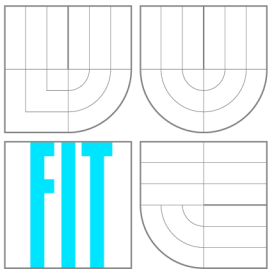
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PETRLÍK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MULTIKRITERIÁLNÍ KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ

MULTIOBJECTIVE CARTESIAN GENETIC PROGRAMMING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PETRLÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2011

Abstrakt

Cílem této diplomové práce je shrnout problematiku multikriteriálních genetických algoritmů a kartézského genetického programování. Podrobně je popsán algoritmus NSGAII a začlenění multikriteriální optimalizace do kartézského genetického programování (CGP). Navržená metoda multikriteriálního CGP byla ověřena na zvolených problémech z oblasti návrhu číslicových obvodů.

Abstract

The aim of this diploma thesis is to survey the area of multiobjective genetic algorithms and cartesian genetic programming. In detail the NSGAII algorithm and integration of multiobjective optimalization into cartesian genetic programming are described. The method of multiobjective CGP was tested on selected problems from the area of digital circuit design.

Klíčová slova

multikriteriální evoluční algoritmus, genetické programování, číslicový obvod, logická syntéza

Keywords

multiobjective evolution algorithm, genetic programming, digital circuit, logic synthesis

Citace

Jiří Petrlík: Multikriteriální kartézské genetické programování, diplomová práce, Brno, FIT VUT v Brně, 2011

Multikriteriální kartézské genetické programování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Ing. Lukáše Sekaniny, Ph.D.

.....
Jiří Petrlík
24. května 2011

Poděkování

Děkuji svému vedoucímu Doc. Ing. Lukáši Sekaninovi, Ph.D. za odborné vedení, cenné rady a podněty, které mi při řešení tohoto projektu poskytl.

Obsah

Kapitola 1

Úvod

Tato práce spadá do oblasti tzv. evolučního designu. Cílem evolučního designu je navrhovat nová inovativní řešení zadaných problémů. Je snahou do určité míry nahradit práci návrháře. Evoluční design byl již úspěšně použit v rozličných oblastech lidské činnosti. Například při návrhu programů [7], návrhu obvodů [19], optických systémů, obrazových filtrů [16], nebo také pohybujících se robotů [11] a antén [6].

Jednou z oblastí evolučního designu je oblast evoluční elektroniky, kde cílem návrhu je vytvořit funkční obvod plnící určitou funkci. Touto oblastí se zabývá tato práce a to především návrhem jednoduchých digitálních obvodů. Výsledný obvod musí mít nejen správnou funkčnost, ale musí být optimalizován i dle dalších kritérií, která mohou být vzájemně konfliktní. Typicky sledujeme plochu a zpoždění.

Cílem této práce bylo vytvořit systém pro automatizovaný návrh kombinačních obvodů. Tento systém bude umožňovat optimalizaci výsledných obvodů dle uživatelem definovaných kritérií. Dalšími cíly jsou pak otestování tohoto systému a zhodnocení efektivity navržených obvodů.

Použitá metoda pracuje na základě několika evolučních výpočetních technik. Je založena na tzv. kartézském genetickém programování [14] a algoritmu NSGAII [1] určenému pro multikriteriální optimalizaci.

Evoluční výpočetní techniky jsou předmětem výzkumu již po několik desetiletí. Snad nejznámějšími představiteli těchto metod jsou genetické algoritmy a evoluční strategie. Jejich uplatnění se nachází především v oblasti optimalizačních úloh, kde pro zadaný problém hledáme pomocí genetického algoritmu (případně evoluční strategie) hodnotu jednoho, nebo několika parametrů, s cílem maximalizovat, nebo minimalizovat hodnotu účelové funkce. Tuto oblast jsem zpracoval v druhé kapitole.

O něco složitější jsou tzv. mutlikriteriální genetické algoritmy. Tyto algoritmy se na rozdíl od běžných genetických algoritmů nesoustředí pouze na maximalizaci jednoho kritéria, ale snaží se vyhledat řešení, která jsou optimalizována podle více kritérií. Cílem v tomto případě není většinou nalezení jednoho optimálního řešení, ale několika řešení, která reprezentují výhodné kompromisy mezi zadanými kritérii.

Jedním z těchto algoritmů je i algoritmus NSGAII [1]. Jedná se o algoritmus s příznivou složitostí, který by měl poskytovat kvalitní rozprostření řešení po Pareto-optimální frontě. Principy multikriteriální optimalizace a popis algoritmu NSGAII jsou popsány ve třetí kapitole.

Jinou neméně zajímavou oblastí využití evolučního principu je genetické programování. Zde na rozdíl od klasického použití evolučních algoritmů nehledáme jen několik hodnot parametrů (například hodnot, kde účelová funkce nabývá maxima), ale snažíme se vyvinout

celý funkční program.

Tvůrcem klasického genetického programování je John Koza. Tato metoda využívá stromovou reprezentaci programu. Základní popis této problematiky je obsažen ve čtvrté kapitole.

Zvláštním přístupem v rámci genetického programování je tzv. kartézské genetické programování (zkratka CGP) [14]. Tento přístup je především vhodný pro vývoj digitálních obvodů. Jeho výstupem je orientovaný acyklický definující zapojení obvodu. Výhodou je také genotyp s pevnou délkou. Principy této metody genetického programování budou popsány v páté kapitole této práce.

Při hledání obvodové realizace algoritmu je často důležité zohledňovat více kritérií, než je pouhá správnost výsledků programu. Příkladem těchto kritérií může být potřebná plocha čipu, celkové zpoždění obvodu, spotřeba a podobně. Proto jsem se rozhodl ve své práci zkombinovat kartézské genetické programování s algoritmem NSGAII, který slouží k multikriteriální optimalizaci. Různé přístupy, které je možné použít pro multikriteriální návrh obvodů pomocí CGP, jsou popsány v šesté kapitole.

Sedmá kapitola se pak zabývá návrhem systému a osmá kapitola pak samotnou implementací programu. V rámci práce byl implementován jak systém pro automatizované generování obvodů, tak i prohlížeč umožňující zobrazení výsledných řešení uživateli. Jsou zde popsány jednak použité technologie, ale i vnitřní architektura programu.

V deváté kapitole jsou uvedeny výsledky dosažené při generování testovacích obvodů. Výsledky testů jsou porovnány s výsledky dosaženými pomocí konvenčních metod, nebo jiných metod pro evoluční návrh obvodů.

V poslední desáté kapitole jsou shrnuty vlastnosti použité metody a zhodnoceny pomocí ní dosažené výsledky.

Kapitola 2

Evoluční algoritmy

2.1 Evoluční výpočetní techniky

Evoluční algoritmy řadíme mezi stochastické algoritmy určené pro prohledávání stavového prostoru. Tyto algoritmy jsou typicky používány pro optimalizaci hodnot parametrů výsledného řešení. Lze je však také využít k prohledávání stavového prostoru s možnými řešeními při evolučním návrhu [16].

Inspirací pro evoluční algoritmy jsou biologické vývojové procesy. Mezi základní vlastnosti živých organismů patří schopnost přizpůsobit se okolnímu prostředí (adaptace), schopnost získávat z něho různé zdroje, především potravu nezbytnou pro život a rozmnožování. Znalosti, které jsou nutné pro přežití, jsou mimo jiné předávány dědičně v chromozomech příslušného jedince z rodiče na potomka. Změny informací obsažených v těchto chromozomech pak umožňují adaptaci na příslušné podmínky. Výhodnost změn je v přírodě určována schopností rozmonožovat se a přežít [12].

Při optimalizaci pomocí evoluční algoritmů se snažíme vyhledat co nejlepší řešení zadaného problému ze stavového prostoru všech možných řešení. Abychom mohli porovnávat kvalitu různých řešení, musíme být schopni ohodnotit kvalitu jednotlivých kandidátních řešení pomocí jedné hodnoty (fitness funkcí) [8].

Při samotném prohledávání stavového prostoru využíváme dvou typů operátorů a to rekombinačních operátorů a operátorů selekce. Rekombinační operátory umožňují vytvářet nová kandidátní řešení a operátory selekce umožňují upřednostňování kvalitních řešení s vyšší (resp. nižší) hodnotou fitness funkce [12].

2.2 Genetické algoritmy

Mezi tradiční evoluční výpočetní techniky patří genetické algoritmy. Algoritmus 1 uvádí kostru základního genetického algoritmu.

2.2.1 Inicializace populace a způsoby zakódování řešení

Jedním z klíčových problémů při návrhu genetických algoritmů je způsob reprezentace jedince v populaci. U klasických genetických algoritmů jsou jedinci reprezentováni pomocí chromozómů ve tvaru řetězců $S = (s_1, s_2, ..., s_L)$, kde L je konečná délka chromozomu a $s_i \in A_i$ [12].

Způsob, jakým je jedinec zakódován, budeme nazývat genotyp. V genetických algoritmech se používají rozličné způsoby zakódování. Asi nejčastějším a nejjednodušším přístu-

Algoritmus 1 Základní genetický algoritmus (převzato z [15])

t=0
Inicializuj populaci P_t
Ohodnoť populaci P_t
while (Řešení není dostatečné) **do**
  P'_t=vyber rodiče pro novou generaci z P_t
  Rekombinuj jedince v P'_t
  Mutuj jedince v P'_t
  Ohodnoť jedince v P'_t
  P_{t+1}=vyber budoucí generaci jedinců z P_t a P'_t
  t=t+1
end while

pem je takzvané binární zakódování. V něm je jedinec reprezentován jako konečná sekvence bitů, obvykle konstantní délky. Skupina několika bitů pak může reprezentovat numerickou hodnotu [15]. Pro chromozom $(\alpha_1, \alpha_2, ..., \alpha_k)$ získáme hodnotu celého čísla dle následujícího vzorce [8]:

$$int(\alpha) = \sum_{i=1}^{k} \alpha_i 2^{k-i}$$

Několik takovýchto hodnot poté může reprezentovat bod v prostoru (například v úloze hledání extrémů funkce). Tento bod bychom mohli považovat za tzv. fenotyp. Fenotyp je termín, který označuje dekódovanou formu řešení [15].

Dalším způsobem, jak provést zakódování kandidátního řešení, může být sekvence celých čísel. Toto zakódování je principiálně velmi podobné předchozímu a využívá se např. u kartézského genetického programování [16].

Další způsoby zakódování jsou již poměrně náročnější. Například se může jednat o zakódování ve formě stromu. Jeho principy stejně jako operátory křížení a mutace budou dále vysvětleny v kapitole 3 o klasickém genetickém programování [7].

Existuje ještě mnoho dalších způsobů, jak zakódovat kandidátní jedince, a to například permutační zakódování, zakódování formou tabulky, n-rozměrného pole, nebo grafu [15].

Počáteční populace je typicky vytvořena z chromozómů obsahujících zcela náhodné přípustné hodnoty. Pokud to úloha umožňuje, mohou být jedinci počáteční populace generování smysluplněji. Jedná se o jeden ze způsobů, jak využít znalost domény úlohy pro zlepšení výsledků. [12]

2.2.2 Hodnocení kandidátních řešení

Kvalitu kandidátního řešení ohodnocujeme pomocí tzv. fitness funkce, přičemž obvykle platí, že vyšší hodnota fitness funkce znamená kvalitnější řešení. Při generování nových řešení jsou preferována řešení s vyšší, resp. nižší hodnotou fitness, zároveň jsou také preferována při výběru jedinců do nové populace. [15]

Pro genetické algoritmy má podle knihy [8] tato funkce tvar $f : D \rightarrow R$, kde obvykle $D = \prod_{i=1}^{L} [a_i, b_i] = [a_1, b_1] \times [a_2, b_2] \times ... \times [a_L, b_L]$. Zde $[a_i, b_i]$ reprezentují intervaly, v kterých se mohou pohybovat hodnoty jednotlivých parametrů kandidátního řešení. Snažíme se pak nalézt globální minimum, nebo maximum této funkce [16]. Pokud hledáme globální minimum, pak hledáme x^* dané vztahem $x^* = arg\ min_{x \in D} f(x)$ [8].

Tato metoda je plně vyhovující, pokud jsme schopni kvalitu řešení ohodnotit jednou hodnotou (výsledek fitness funkce). Bohužel existují i problémy, kde je nutné jedince posuzovat dle více kritérií. V těchto případech lze stanovit pro každé kritérium zvláštní fitness funkci.

Běžné genetické algoritmy ale nejsou pro práci s více fitness funkcemi příliš použitelné. Z tohoto důvodu je nutné použít specializované algoritmy jako je například NSGAII.

2.2.3 Výběr jedinců z populace

Při výběru jedinců pro vstup do reprodukčního procesu nebo do nové populace máme několik možností jak postupovat. Výběr metody je klíčový pro správný chod genetického algoritmu.

Například tzv. výběr ruletového kola [12], označovaný též jako proporcionální výběr [15], pracuje tak, že jedince vybíráme náhodně, avšak perspektivnější jedinci (s vyšší hodnotou fitness funkce) mají větší pravděpodobnost, že budou vybráni. Tuto pravděpodobnost lze spočítat dle následujícího vzorce [15]

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j},$$

kde p_i označuje pravděpodobnost výběru jedince i, f_i fitness hodnotu jedince i a N celkový počet jedinců v populaci [15].

Další hojně využívanou metodou výběru je tzv. turnajový výběr (zkráceně turnaj). Zde nejprve zcela náhodně vybereme t jedinců, přičemž všichni jedinci mají stejnou pravděpodobnost výběru bez ohledu na svou fitness funkci. Následně porovnáme hodnoty fitness funkcí těchto t řešení a vybereme to s nejlepší hodnotou [15].

Oba způsoby zaručují, že může být vybrán i jedinec s velmi nízkou hodnotou fitness. To je většinou velice prospěšná vlastnost, protože pomáhá chránit před předčasným zkonvergováním populace [15].

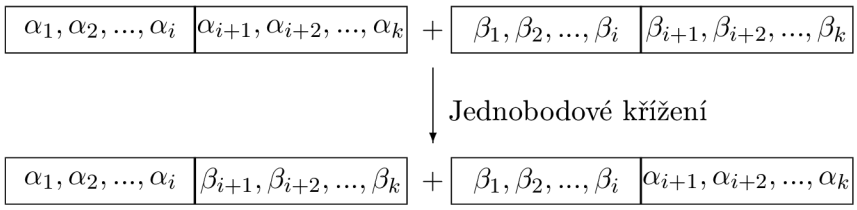
2.2.4 Operátory křížení a mutace

Aby docházelo k vývoji, musíme umožnit, aby vznikali noví jedinci v populaci. Tito jedinci vznikají pomocí tzv. modifikačních operátorů, které jsou někdy nazývány jako rekombinační [12].

Základní dva přístupy se liší v tom, kolik jedinců je nutné mít jako zdroj pro změnový operátor. Pokud potřebujeme pouze jednoho zdrojového jedince, pak tuto operaci nazýváme mutací. Jestliže je třeba více jedinců (zpravidla dva, ale i více), pak hovoříme o křížení.

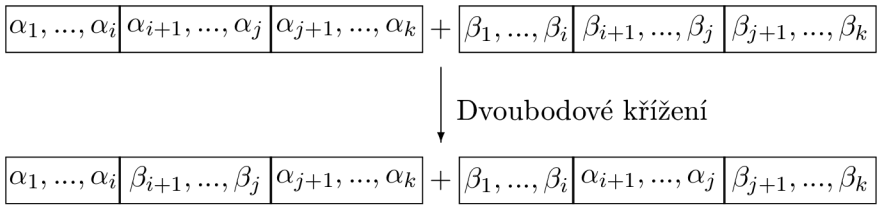
Určení způsobu křížení a mutace může být velmi důležitá znalost, kterou může návrhář vnést do genetického algoritmu, a tak i řádově zlepšit jeho výsledky. Často jsou totiž používány specializované operátory podle typu zadané úlohy [15].

Nechť $\alpha \in \{0,1\}^k$ a $\beta \in \{0,1\}^k$, pak operátor binárního křížení O_{cross} vygeneruje dva nové chromozómy α' a β' s délkou k [8]. Operátor binárního křížení lze implementovat různými způsoby. Jednobodové křížení funguje tak, že vygenerujeme náhodně číslo i, které bude určovat pozici v chromozomu, což tento chromozom rozdělí na dvě části. První výsledný jedinec pak bude obsahovat první část chromozomu z prvního jedince a druhou z druhého. Druhý jedinec bude naopak obsahovat první část chromozomu z druhého jedince a druhou část z prvního jedince [8].



Obrázek 2.1: Schéma jednobodového křížení

Druhým užívaným typem křížení je tzv. dvoubodové křížení. U tohoto typu křížení musíme náhodně určit dvě pozice v chromozomu. Postup při tomto typu křížení je ukázán na obrázku níže. [8]



Obrázek 2.2: Schéma dvoubodového křížení

Mutace je rekombinační operátor, který vyžaduje pouze jednoho předka. Tento operátor může mít více podob. Označme P_m jako pravděpodobnost mutace a délku chromozomu jako L. Pak u každého genu s pravděpodobností LP_m provedeme inverzi jednoho náhodného bitu. Druhým možným řešením je pro každý bit každého chromozómu provádět jeho inverzi s pravděpodobností P_m [12].

2.3 Evoluční strategie

V klasických evolučních strategiích je jedinec reprezentován jako dvojice $v = (\vec{x}, \vec{\sigma})$. Přičemž $\vec{x}$ reprezentuje bod prohledávaného prostoru. V základní variantě je vektor $\vec{\sigma}$ neměnný a určuje vektor standartních odchylek. Jediným změnovým operátorem je operátor mutace realizovaný vztahem: $x_{t+1} = x_t + N(0, \vec{\sigma})$. Přičemž $N(0, \vec{\sigma})$ reprezentuje vektor náhodně vygenerovaných hodnot v normálním rozložení [12].

Nejjednodušší evoluční strategie $(1 + 1)$ pracuje tak, že se u jediného rodiče provádí mutace, ze které vznikne vždy jeden potomek. Jestliže je fitness hodnota potomka lepší, než fitness hodnota rodiče, je rodič nahrazen svým potomkem. Pokud tomu tak není, potomek se nevyužije a je vygenerován další kandidátní potomek [16].

U složitějších evolučních strategií již není $\vec{\sigma}$ konstantní a mohou být použity operátory křížení a mutace. [12] Schopnost samočinně měnit parametr $\vec{\sigma}$ a tím i vlastnosti evoluce můžeme označit za samoadaptaci [16].

Diskrétní křížení vytváří potomka, který má jednotlivé prvky vektorů $\vec{x}$ a $\vec{\sigma}$ náhodně zvoleny od některého z rodičů. U středového křížení jsou jednotlivé prvky vektorů $\vec{x}$ a $\vec{\sigma}$

tvořeny průměry hodnot z vektorů rodičů [12].

Mutace u evolučních strategií probíhá dle vztahů, kde $\vec{\Delta\sigma}$ je uživatelem zvolený parametr [12]:

$$\vec{\sigma}' = \sigma.e^{N(0,\vec{\Delta\sigma})} \tag{2.1}$$

$$\vec{x}' = x + N(0, \vec{\sigma}') \tag{2.2}$$

Evoluční strategie umožňují využít i větší populace. Evoluční strategie $(\mu + \lambda)$ má μ rodičů. Z každého rodiče je v každém kroku evoluce generováno λ potomků. Jako rodiče do další iterace evoluce jsou vybráni nejlepší jedinci jak z rodičů, tak z vygenerovaných potomků [16].

Strategie (μ, λ) je velmi podobná strategii $(\mu + \lambda)$ s tím rozdílem, že jako rodiče pro novou iteraci evoluce mohou být vybrání pouze potomci, nikoliv současní rodiče [16].

Kapitola 3

Multikriteriální optimalizace

3.1 K čemu slouží multikriteriální optimalizace

U některých úloh se ukázalo, že optimalizace podle jednoho kritéria není zcela dostačující. Představme si situaci, kdy člověk stojí před výběrem předmětu v obchodě, například nového počítače. Jen ve velmi výjimečných situacích by se zákazník rozhodoval pouze podle jednoho parametru. Samozřejmě i tento případ může nastat, pokud bude například požadovat nejlevnější, nebo naopak nejvýkonnější model. Takovéto situace však nastávají jen zřídka.

Zákazník se bude většinou rozhodovat na základě více parametrů nového stroje. Přinejmenším bude zohledňovat dvě základní kritéria, a to cenu a výkon. Jeho rozhodování se však pravděpodobně rozdělí na posuzování velkého množství kritérií, například velikosti operační paměti, rychlosti procesoru, kvalitě grafické karty a tak dále.

Formálně můžeme říci, že se snažíme o získání n-tice takových parametrů $x = (x_1, x_2, ..., x_n)$, které optimalizují funkci $f(x) = (f_1(x), f_2(x), ..., f_m(x))$ a zároveň vyhovují předem zadaným omezením [16].

Nyní však vyvstává problém, jak sestavit fitness funkci tak, aby dokázala rozlišit, který počítač je z pohledu zákazníka výhodnější. Toto bohužel není triviální úkol. Mohli bychom například přiřadit jednotlivým kritériím váhy $w_1, ..., w_m$ a pomocí nich pak spočítat hodnotu fitness funkce [16]

$$f = \sum_{i=1}^{m} w_i f_i.$$

Při tomto řešení může být velmi obtížné stanovit hodnoty vah. Právě z tohoto důvodu vznikly evoluční algoritmy pro multikriteriální optimalizaci.

3.2 Paretova dominance

Při hodnocení jedinců u multikriteriálních problémů máme standardně k dispozici vektor hodnot $v = (f_1(x), f_2(x), ..., f_n(x))$, kde každý prvek vektoru reprezentuje jednu hodnotu fitness funkce f_i pro řešení x.

Budeme uvažovat, že cílem je minimalizovat hodnoty fitness funkcí. Abychom mohli porovnávat jednotlivé jedince, musíme pro tyto vektory definovat některé vhodné relace. Nejprve definujeme tzv. slabou Paretovu dominanci.

Definice 1 *Řekneme, že x slabě Pareto dominuje y, právě tehdy, když pro každé $i = 1, ..., n$ platí $f_i(x) \leq f_i(y)$ [15].*

Samotná Paretova dominance je pak definována v následující způsobem a označuje se symbolem $\prec$ [9].

Definice 2 *Řekneme, že $x \prec y$ (x Pareto dominuje y), právě tehdy, když jsou splněny následující dvě podmínky:*

1. *$f_i(x) \leq f_i(y)$ pro každé $i = 1, ..., n$*
2. *$f_i(x) < f_i(y)$ pro alespoň jedno i*

Dalším důležitým pojmem, který je potřebné definovat, je pojem tzv. Pareto optimálního řešení.

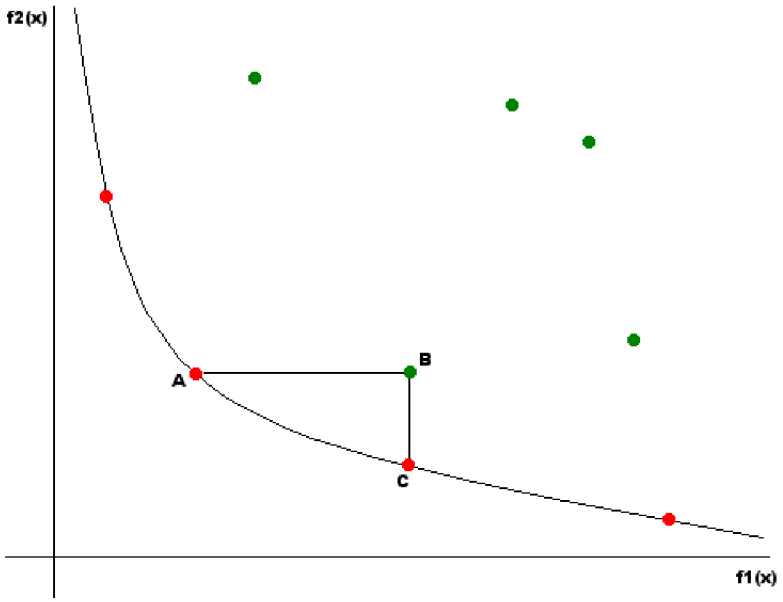
Definice 3 *Nechť P je populace s řešeními. Pak řešení $x \in P$ nazýváme Pareto optimální právě tehdy, když neexistuje žádné řešení $y \in P$ takové, že $y \prec x$ [15].*

3.3 Paretova fronta

Při řešení multikriteriálních problémů se velmi často snažíme nalézt tato Pareto optimální řešení. U většiny zadaných problémů však neexistuje pouze jedno takovéto řešení.

Definice 4 *Paretovou frontou nazýváme křivku, která prochází vektory Pareto optimálních řešení [15].*

Přičemž při multikriteriální optimalizaci se snažíme vyhledávat řešení, která buď leží na Paretově frontě, nebo v její blízkosti.



Obrázek 3.1: Paretova fronta při minimalizaci u dvoukriteriálního problému.

Na obrázku 3.1. je zobrazena Paretova fronta při minimalizaci dvoukriteriálního problému. Jednotlivé body znázorňují kandidátní řešení. Hodnoty na horizontální ose reprezentují funkční hodnoty fintness funkce f_1 a hodnoty na vertikální ose pak funkční hodnoty fitness funkce f_2.

Řešení B má stejnou hodnotu fitness funkce f_2 jako řešení A a současně má větší (u minimalizace znamená horší) hodnotu fitness funkce f_1 než řešení A. Proto je řešení B Pareto dominováno řešením A. Z toho plyne, že B není Pareto optimální a tím pádem neleží na Pareto frontě.

Řešení na Paretově frontě nejsou Pareto dominovány žádným jiným řešením. Například řešení A není Pareto dominováno řešením C, protože má nižší hodnotu fitness funkce f_1 a naopak řešení C není Pareto dominováno řešením A, protože má nižší hodnotu fitness funkce f_2.

3.4 Algoritmus NSGAII

3.4.1 Základní popis

Protože je potřeba řešit multikriteriální problémy, vzniklo množství genetických algoritmů, které jsou k tomu vhodné. V současné době je velice populární algoritmus NSGAII. Mezi hlavní přednosti tohoto algoritmu patří především jeho relativně nízká časová složitost pro nedominované řazení $O(MN^2)$, kde M je počet kritérií a N počet jedinců. Pro evoluci není třeba zadávat žádné dodatečné parametry kromě počtu jedinců v populaci. Tento algoritmus umožňuje zohlednit tzv. omezení při vývoji a v neposlední řadě by měl poskytovat řešení, která jsou rozprostřena kolem Pareto-optimální fronty [1].

K. Deb et al. vytvořili dvě nové metody, a to novou metodu nedominovaného řazení se složitostí $O(MN^2)$ a metodu pro udržování diverzity populace, která nevyžaduje žádné dodatečné parametry. Tento algoritmus je využit v praktické části diplomové práce.

3.4.2 Nedominované řazení (nondominated sorting)

Jednou z nejdůležitějších schopností multikriteriálního genetického algoritmu je schopnost seřadit jednotlivé členy populace(P) podle kvality, a to s přihlédnutím ke všem kritériím. Vhodnou relací se pro porovnání řešení ukázala dříve popsaná relace Paretovy dominance. Z tohoto důvodu je za lepší považováno to řešení, které Pareto dominuje jinému řešení. K seřazení řešení podle jejich kvality lze tedy využít již dříve definované relace Paretovy dominance.

Kvalita řešení p je označována jako p_{rank}, přičemž čím vyšší je hodnota p_{rank}, tím horší je kvalita řešení p. Postup ohodnocení kvality řešení popisuje algoritmus 2.

Nechť $P' = P$. Algoritmus nejprve najde všechna nedominovaná řešení a přiřadí jim $p_{rank} = 1$. Takto označená řešení odstraní z množiny P' a opět hledá nedominovaná řešení v P'. Těmto řešením přiřadí $p_{rank} = 2$ a také je odstraní z P'. Takto pokračuje, dokud není množina P' prázdná.

Algoritmus 2 je značně neefektivní a jeho časová složitost je $O(MN^3)$, kde M je počet kritérií [1]. Vysoká složitost je způsobena především velkým počtem porovnání. Efektivnější řešení nabízí algoritmus 3, který se snaží snížit počet porovnání a je použit v NSGAII.

V první části algoritmu 3 je nejprve nalezen počet řešení (n_p), která řešení p dominují, a zároveň množina řešení (S_p), kterým toto řešení dominuje. Pokud řešení nedominuje žádné

Algoritmus 2 Naivní nedominované řazení [1]

$i = 1$
$P' = P$ // P... aktuální populace
while $P' \neq \emptyset$ **do**
  for all $p \in P'$ **do**
    if p není v P' dominováno **then**
      $p_{rank} = i$
      $P' = P' \setminus \{p\}$
    end if
  end for
  $i = i + 1$
end while

Algoritmus 3 Rychlé nedominované řazení (převzato z [1])

// První část algoritmu - nalezne S_p a n_p pro $\forall p \in P$
for all $p \in P$ **do**
  $S_p = \emptyset$ // množina řešení, kterým p dominuje
  $n_p = 0$ // počet řešení, která dominují řešení p
  for all $q \in (P \setminus \{p\})$ **do**
    if $p \prec q$ **then**
      Přidej q do množiny S_p
    end if
    if $q \prec p$ **then**
      $n_p = n_p + 1$
    end if
  end for
  if $n_p = 0$ // řešení, která nejsou dominována žádným jiným řešením **then**
    $p_{rank} = 1$
    Přidej p do množiny F_1
  end if
end for
// Druhá část algoritmu - ohodnotí každé p hodnotu p_{rank}
$i = 1$
while $F_i \neq \emptyset$ **do**
  $Q = \emptyset$
  for all $p \in F_i$ **do**
    for all $q \in S_p$ **do**
      $n_q = n_q - 1$
      if $n_q = 0$ **then**
        $q_{rank} = i + 1$
        Přidej q do množiny Q
      end if
    end for
  end for
  $i = i + 1$
  $F_i = Q$
end while

jiné řešení, potom je tomuto řešení přiřazena hodnota $p_{rank} = 1$ a je přidáno do množiny F_1.

Ve druhé části algoritmu, při samotném ohodnocování řešení, je postupováno tak, že v každé iteraci i projdeme všechna řešení $p \in F_i$, která jsme v předchozí iteraci ohodnotili, a pro každý prvek q z jejich množiny S_p dekrementujeme n_q.

Poté zjistíme, u kterých řešení klesla hodnota n_q na nulu a ohodnotíme je číslem příslušné iterace i. Takto pokračujeme, dokud není F_i (množina řešení, která byla vybrána v minulé iteraci) prázdná.

3.4.3 Udržování diverzity populace

Snahou většiny algoritmů pro multikriteriální optimalizaci včetně NSGAII je pokud možno vyhledat řešení, která jsou rovnoměrně rozprostřena přes celou Paretovu frontu.

To znamená, že algoritmus vyhledá rozličné varianty kompromisu mezi zkoumanými kritérii. Vyhledá například obvody s nejlepším zpožděním, ale zároveň vyhledá i obvody s nejmenší cenou a podobně. Také vyhledá kompromisy mezi těmito parametry, pro něž platí, že jsou součástí Paretovy fronty.

Právě proto, aby genetický algoritmus mohl poskytovat výsledky s výše uvedenými vlastnostmi, musí neustále udržovat různorodost (diverzitu) populace. Existuje několik možných řešení tohoto problému.

Velice často používaným řešením pro udržování diverzity populace je tzv. sdílená fitness funkce. Tento přístup bohužel vyžaduje jistou znalost řešeného problému pro optimální nastavení sdílené fitness funkce.

NSGAII proto nabízí lepší řešení tohoto problému. Pro každé řešení v populaci zavádí tzv. vzdálenost pokrytí (crowding distance). Výpočet tohoto parametru je popsán algoritmem číslo 4. Z uvedeného algoritmu je zřejmé, že se budeme snažit upřednostňovat řešení s co možná nejvyšší hodnotou crowding distance.

Algoritmus 4 Přiřazení vzdálenosti pokrytí (převzato z [1])

for all $p \in P$ **do**
  $p_{distance} = 0$
end for
for all $m \in M$ (M je množina kritérií) **do**
  $sort(P, m)$ (seřaď populaci P dle kritéria m)
  $P[0]_{distance} = P[n-1]_{distance} = \infty$
  for $i = 2$ to $(n-2)$ **do**
    $P[i]_{distance} = P[i]_{distance} + \frac{P[i+1]_m - P[i-1]_m}{P[n-1]_m - P[0]_m}$
  end for
end for

3.4.4 Relace pro porovnání kvality dvou řešení

Po provedení předchozích dvou algoritmů máme již dostatek informací k tomu, aby bylo možné rozhodnout, které řešení je lepší a které je horší. Toto rozhodování se používá při výběru jedinců ke křížení a mutaci a k výběru jedinců do nové populace.

Existují dvě verze: pro úlohy s omezením a úlohy bez omezení. Omezením se myslí podmínka, při jejímž nedodržení není řešení přípustné.

Uveďmě si nejprve relaci pro úlohy, které nemají žádné omezující podmínky. Platí tedy že, $a \prec_n b$ právě tehdy, když:

- $(a_{rank} < j_{rank}) \vee$
- $((a_{rank} = j_{rank}) \wedge (a_{distance} > b_{distance}))$ [1].

Druhá verze obsahuje podporu pro omezení. Musíme zavést parametr řešení a_{cv} (constraint violation), který bude udávat míru porušení omezení. Způsob jeho výpočtu musí korespondovat s charakterem úlohy. Platí tedy, že $a \prec_n b$ právě tehdy, když:

- $(a_{cv} < b_{cv}) \vee$
- $((a_{cv} = b_{cv}) \wedge (a_{rank} < b_{rank})) \vee$
- $((a_{cv} = b_{cv}) \wedge (a_{rank} = b_{rank}) \wedge (a_{distance} > b_{distance}))$ [1].

U kartézského genetického programování je vhodné upřednostňovat řešení, která jsou novější. To bude zahrnuto do následující relace. Bude ale třeba zavést ještě jeden parametr kandidátního řešení, a to a_t (time), který reprezentuje číslo generace, ve které bylo řešení vytvořeno.

Bude tedy platit, že $a \prec_n b$ právě tehdy když:

- $(a_{cv} < b_{cv}) \vee$
- $((a_{cv} = b_{cv}) \wedge (a_{rank} < b_{rank})) \vee$
- $((a_{cv} = b_{cv}) \wedge (a_{rank} = b_{rank}) \wedge (a_{distance} > b_{distance})) \vee$
- $((a_{cv} = b_{cv}) \wedge (a_{rank} = b_{rank}) \wedge (a_{distance} = b_{distance}) \wedge (a_t > b_t))$.

3.4.5 Hlavní smyčka NSGAII

Hlavní smyčka algoritmu NSGAII vychází z obecného genetického algoritmu a začleňuje do něho výše uvedené prvky. Především je důležité říci, že při porovnávání kvality dvou řešení používá uspořádání $\prec_n$, které je definováno výše.

Toto uspořádání je použito jak při výběru jedinců při křížení a mutaci, tak při výběru jedinců do další populace. Při výběru jedinců pro křížení a mutaci je nejvhodnější použít turnajový výběr [1].

Průběh jedné iterace algoritmu NSGAII lze popsat takto: Nejprve je vytvořena množina R_t obsahující rodiče a jejich potomky. Nad nimi se provede nedominované řazení. Výsledkem je vektor $F = (F_1, F_2, ...)$, kde F_1 je tvořen prvky s $rank = 1$, F_2 prvky s $rank = 2$ a tak dále.

Následně je vytvořena nová populace P_{t+1}. Do této populace jsou přidávány celé vrstvy F_1, F_2, atd. Jakmile již není možné přidat celou vrstvu, provede se seřazení prvků této vrstvy dle relace $\prec_n$ a následně přidáme nejlepší prvky této vrstvy do nové populace.

Jakmile je vytvořena nová populace P_{t+1}, jsou vygenerování její potomci Q_{t+1}. Posune se počítadlo času t a lze pokračovat další iterací algoritmu.

Algoritmus 5 Průběh jedné iterace NSGAII (převzato z [1])

$R_t = P_t \cup Q_t$ /* sjednocení předků P_t a potomků Q_t */
F =nedominované_řazení(R_t) /* kde $F = (F_1, F_2, ...)$ */
$i = 1$
$P_{t+1} = \emptyset$ /* nová populace */
/* dokud je možné umístit celou vrstvu do nové populace */
while $|P_{t+1}| + |F_i| \leq N$ **do**
  spočítej vzdálenosti pokrytí pro prvky F_i
  $P_{t+1} = P_{t+1} \cup F_i$ /* přidá celou vrstvu do příští populace */
  $i = i + 1$
end while
/* Doplň P_{t+1} o část vrstvy F_i */
Seřaď F_i pomocí relace $\prec_n$
$P_{t+1} = P_{t+1} \cup F_i[1 : (N - P_{t+1})]$
/* Vytvoří potomky */
Q_{t+1} = vytvoř_potomky(P_{t+1})
$t = t + 1$

3.4.6 Složitost algoritmu NSGAII

Složitost jednotlivých částí algoritmu je:

- nedomonované řazení - $O(MN^2)$,
- crowding distance assignment - $O(MNlog(N))$ a
- provedení jedné iterace - $O(MN^2)$,

kde M je počet kritérií a N počet prvků populace [1].

Kapitola 4

Genetické programování

4.1 Co řeší genetické programování

Velkou výzvou pro výzkum je v současné době tzv. genetické programování. Tradičně se genetické algoritmy využívají k optimalizaci několika parametrů určitého problému. Například mohou hledat extrémy funkcí.

Naproti tomu u genetického programování se snažíme vytvořit zcela nový postup, jak vyřešit zadaný problém. Například můžeme hledat matematickou funkci nebo zapojení obvodu [16].

Takto zadaný problém však často vyžaduje prohledávání extrémně velkých stavových prostorů. To poněkud komplikuje tvorbu složitých programů, případně obvodů.

Ve většině případů je žádoucí, aby vytvořený program, případně obvod, byl nejen správný, ve smyslu bezchybného generování výstupů na základě vstupních hodnot, ale splňoval i některé další požadavky kladené na například rychlost programu, počet používaných operací a podobně.

Právě zde nachází uplatnění multikriteriální optimalizace a tím pádem multikriteriální evoluční algoritmy. Je také otázkou, zda-li bude možné použít běžné multikriteriální evoluční algoritmy pro genetické programování beze změny, nebo je bude třeba vhodně upravit, případně použít algoritmy nové, speciálně vyvinuté pro tyto problémy.

Principy klasického genetického programování jsou rozpracoval a podrobně popsal John Koza ve své knize [7]. Pokud chceme řešit problém pomocí genetického programování, musíme nejprve stanovit:

1. množinu terminálů,
2. množinu neterminálů,
3. způsob výpočtu fitness funkce,
4. parametry evoluce,
5. zastavovací podmínku,
6. architekturu výsledného programu [9].

4.2 Způsob zakódování programu

Klasická metoda genetického programování používá pro zakódování programu stromovou strukturu (bod 6). Listy tohoto stromu jsou vybírány z množiny terminálů T a představují proměnné nebo konstanty (bod 1). Nelistové uzly jsou vybírány z množiny neterminálů F a představují funkce (bod 2) [13].

Základním požadavkem na stromovou strukturu je v případě genetického programování uzavřenost množiny funkcí a množiny terminálů, což v podstatě znamená, že výstup libovolného terminálu, nebo neterminálu může být použit jako vstup jakéhokoliv jiného uzlu.

Pokud tato podmínka není splněna, například při dělení nelze používat jako dělitel nulu, musí být navrženy tzv. chráněné operace. V příkladu s dělením by mohla být taková operace definována jako: Jestliže $jmenovatel = 0$, pak vrať 1, jinak vrať podíl čítatele a jmenovatele [13].

4.3 Generování počáteční populace

Existují dva základní přístupy pro generování kandidátních řešení. Oba požadují, aby byla zadána maximální hloubka generovaného stromu D. Růstové generování funguje tak, že nejprve je vybrán kořen stromu z množiny F. Na něj jsou v dalších patrech připojovány uzly z množiny $T \cup F$. Když je dosaženo hloubky $D - 1$ omezí se výběr připojovaných uzlů pouze na prvky z množiny T [13].

Úplné generování postupuje tak, že je opět nejprve vybrán kořen stromu z množiny F. Do dalších pater jsou však vybírány uzly pouze z množiny F. V patře $D - 1$ jsou pak opět připojovány uzly z množiny T [13].

V literatuře je doporučováno generovat populaci vždy půl na půl pomocí obou výše zmíněných metod [13].

4.4 Výpočet fitness funkce

Definice fitness funkce (bod 3) silně závisí na typu řešené úlohy. Její výpočet je většinou založen na spuštění programu, kdy mu jsou předávány na vstup hodnoty testovacích vektorů a výstupy jsou pak porovnávány se správnými výsledky.

Vektory trénovací množiny mohou být buď při každém ohodnocení stejné, nebo mohou být generovány náhodně. Za lepší je obecně považována fitness funkce s jemnou zrnitostí, tedy taková, která i při malém zlepšení kandidátního řešení zvýší svoji hodnotu [9].

4.5 Parametry evoluce a zastavovací podmínka

Parametry evoluce (bod 4) ovlivňují jak běh evolučního algoritmu, tak podobu kandidátních řešení. Parametr velikost populace udává počet vyhodnocovaných jedinců v jedné generaci [9]. Parametr D pak maximální hloubku stromů, které jsou generovány jako kandidátní řešení [13].

Zastavovací podmínka (bod 5) je většinou definována tak, že evoluce končí v momentě nalezení funkčního řešení, nebo po dosažení určitého počtu generací. Často bývá lepší spustit více běhů téhož experimentu, než zvětšovat počet generací experimentu [9].

4.6 Úloha symbolické regrese

Aby bylo možné na příkladech demonstrovat operátory křížení a mutace, je zde popsána jedna z úloh řešitelných pomocí genetického programování. Jedná se o úlohu symbolické regrese. U symbolické regrese se snažíme nalézt funkci, která by dobře reprezentovala naměřené hodnoty.

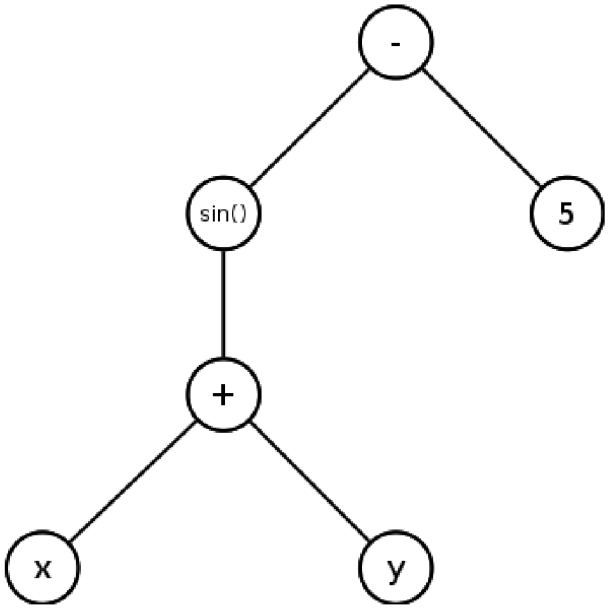
Vyhodnocovat kvalitu řešení můžeme například podle vzorce 4.1 [16], kde y_j^{GP} je hodnota, kterou vytvořil evolučně získaný program, y_j je správná hodnota z trénovací množiny a trénovací množina má N prvků.

$$f = \sum_{j=1}^{N} (y_j^{GP} - y_j)^2 \tag{4.1}$$

Pro konstrukci kandidátních stromů (programů) je nutné definovat dvě množiny, a to množinu terminálů a množinu funkcí. V našem případě symbolické regrese budou množinu funkcí (neterminálu F) tvořit různé matematické operace jako sčítání, odčítání, dělení, odmocňování, atd., přičemž je nutné zabezpečit, aby výstup předchozí funkce bylo vždy možné použít jako vstup funkce následující, což například u operace dělení není možné.

Druhou potřebnou množinou, kterou musíme definovat, je množina terminálů T. Hodnoty těchto terminálů budou umístěny v listech stromu a v našem případě se bude jednat o konstanty.

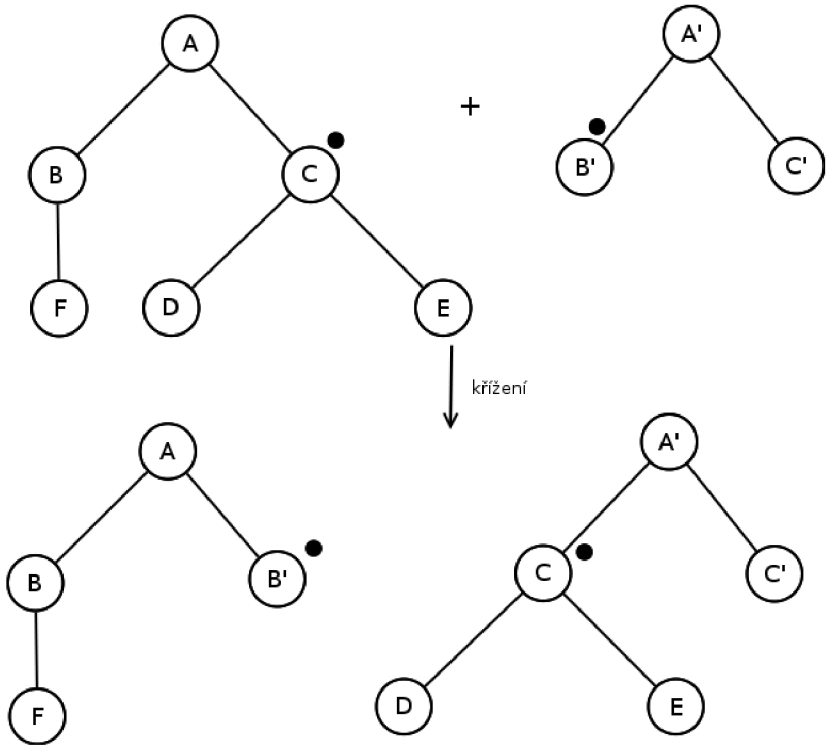
Jako příklad, jak může vypadat jedinec u takto definovaného zakódování, je uveden obrázek znázorňující funkci $sin(x + y) - 5$.



Obrázek 4.1: Stromová reprezentace jedince $sin(x + y) - 5$

4.7 Operace křížení a mutace

Na obrázku 4.2. je znázorněn jeden z možných principů křížení. Křížení dvou jedinců probíhá tak, že nejprve u každého jedince vybereme náhodně jeden uzel. Tyto dva uzly pak opět definují dva podstromy (u každého jedince jeden), jejichž vzájemnou záměnou dostaneme dva nové jedince [16].



Obrázek 4.2: Ukázka křížení stromové struktury v genetickém programování

Mutace může probíhat například tak, že náhodně vybereme uzel stromu kandidátního řešení. Následně náhodně vygenerujeme nový strom pomocí dříve uvedených postupů. Tento strom poté zapojíme na pozici náhodně vybraného uzlu kandidátního řešení [16].

Oba uvedené operátory mají tendenci zvětšovat hloubku jednotlivých stromů. To může být nežádoucí, proto často omezujeme maximální hloubku stromu pomocí parametru D_{max} [13].

Permutace je operátor mutace, který náhodně zvolí libovolný neterminální uzel a provede náhodnou změnu v pořadí jeho argumentů (tedy změnu pořadí navěšených podstromů) [13].

Kapitola 5

Kartézské genetické programování

Další metodou pro genetické programování je tzv. kartézské genetické programování, anglicky cartesian genetic programming (zkráceně CGP) [14]. U klasického genetického programování je program reprezentován jako strom. Naproti tomu v kartézském genetickém programování je program reprezentován jako orientovaný acyklický graf. Přičemž platí, že graf je obecnější struktura než strom [14].

Tento způsob genetického programování byl prvně popsán v Millerově a Thomsonově publikaci [14] a je také podrobně rozebrán v další literatuře [16, 19]. Přičemž se postupně ukázal jako velmi vhodný pro evoluční design logických obvodů [19, 20, 23, 5].

5.1 Princip CGP

Základem této metody je dvourozměrné pole uzlů, nebo také můžeme hovořit o funkčních jednotkách. Každý z těchto uzlů může reprezentovat například jednovstupé nebo dvouvstupé hradlo [19], sčítačku, bitový posuv [20] a podobně. Záleží na typu řešené úlohy.

Počet sloupců 2D pole je označován jako n_c a počet řádků pole označíme jako n_r. U většiny úloh je k dispozici několik druhů uzlů. Množina typů uzlů se označuje jako Γ a počet typů uzlů jako n_f (platí tedy vztah $n_f = |\Gamma|$).

Každý typ uzlu z Γ má předem daný konečný počet vstupů. Maximální počet vstupů jednoho uzlu je označován n_a.

Dále je vždy při vývoji nezbytné vědět, kolik vstupů má celý obvod, toto číslo se značí jako n_i a počet výstupů celkového obvodu jako n_0. Posledním důležitým parametrem je tzv. L-back (značený jako L), který souvisí s možnostmi propojování jednotlivých uzlů [16].

5.1.1 Zakódování

Chromozom má u kartézského genetického programování podobu vektoru nezáporných celých čísel [14]. Primární vstupy obvodu jsou očíslovány od 0 do $n_i - 1$. Výstupy jednotlivých uzlů CGP jsou pak očíslovány od n_i do $n_i + n_c n_r - 1$, a to po sloupcích z levého horního rohu mřížky [16].

Nejprve je v chromozomu zakódován popis jednotlivých uzlů CGP [19]. Pro zakódování informace o jednom vstupu uzlu je třeba právě jedné alely (tedy jednoho celého čísla). Pro zakódování funkce uzlu je zapotřebí také jedné alely. Nejprve jsou kódovány informace o vstupech uzlu a poté o typu uzlu. K zakódování jednoho uzlu je tedy třeba $n_a + 1$ alel [16].

Vstupy jednotlivých uzlů lze připojovat na primární vstupy obvodu, nebo na uzly v předchozích sloupcích CGP. Není povoleno připojovat vstupy na uzly z téže vrstvy, ani

na uzly z následujících vrstev. Tím se zabrání vzniku nežádoucí zpětné vazby [19]. Počet předchozích vrstev, na které může být připojen vstup uzlu, udává parametr L-back [14]. Například pokud by byl $L = 1$, pak je možné připojit vstup uzlu pouze na výstup uzlu předchozí vrstvy, nebo na primární vstup. Naopak, pokud by byl $L = n_c$, pak může být vstup uzlu napojen na libovolný uzel předchozích vrstev, nebo na primární vstup.

Hodnotu $L = 1$ je výhodné použít v případě, pokud chceme obvod používat jako pipeline [19]. Hodnota genu c_{kj}, který kóduje připojení vstupu uzlu, kde k je číslo vstupu a j je číslo vrstvy, tedy nabývá hodnot z intervalu $0 \leq c_{kj} < n_i$ (vstup je napojen na primární vstup), nebo hodnoty z níže definovaného intervalu (vstup je připojen na výstup některého z předchozích uzlů) [14]:

$$e_{min} = n_i + (j - L)n_r \tag{5.1}$$
$$e_{max} = n_i + jn_r \tag{5.2}$$
$$c_{kj} < e_{max}, j < L \tag{5.3}$$
$$e_{min} \leq c_{kj} < e_{max}, j \geq L \tag{5.4}$$

Gen c'_k, který kóduje typ funkce k-tého uzlu, musí patřit do intervalu [14]:

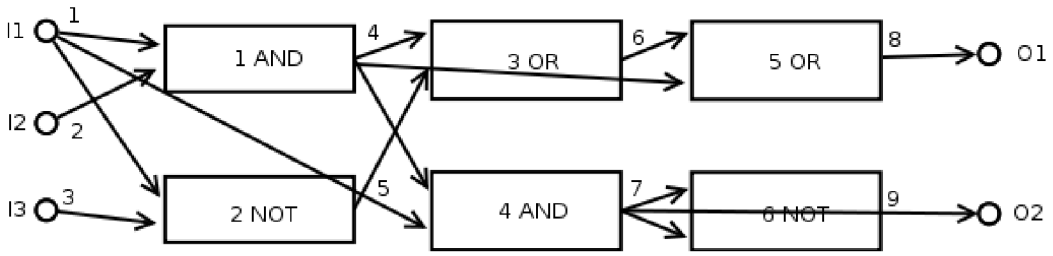
$$0 \leq c'_k < n_f \tag{5.5}$$

Po alelách kódujících jednotlivé uzly pole následuje n_0 alel, které určují čísla uzlů, na které jsou připojeny primární výstupy obvodu. Uzly jsou číslovány od nuly po sloupcích z levého horního rohu mřížky [16]. Pokud označíme hodnotu genu kódujícího zapojení k-tého primárního výstupu jako c^0_k, potom platí:

$$n_i \leq c^0_k < n_c n_r + n_i \tag{5.6}$$

Z výše uvedeného popisu vyplývá, že celková velikost chromozomu je dána vztahem $n_c n_r(n_a + 1) + n_0$ [14]. Obrázek 5.1. názorně ilustruje příklad jednoduchého chromozomu a jeho reprezentace:

Chromozom: (1,2,1, 1,3,3, 4,5,2, 4,1,1, 6,4,2, 7,7,3, 5,4)



Obrázek 5.1: Příklad CGP

Pro lepší názornost jsou také odděleny hodnoty, které se týkají jednotlivých uzlů mezerami. Čísla jednotlivých uzlů jsou uvedeny uvnitř před typem uzlu.

Parametr	Hodnota
n_c	2
n_r	3
Γ	{and, or, not}
n_f	3
n_a	2
n_i	3
n_0	2
L	2

Tabulka 5.1: Parametry příkladu CGP

Prvnímu uzlu (vlevo nahoře) tedy odpovídá trojice 1, 2, 0. To znamená, že první vstup tohoto hradla je připojen na prvek číslo jedna, kterým je první vstup celého obvodu I1. Druhý vstup je pak připojen na prvek číslo 2, kterým je globální vstup číslo dva. Třetí hodnota, tedy 1, reprezentuje typ uzlu. V takovémto příkladě je používáno následující číslování 1 – *and*, 2 – *or*, 3 – *not*.

Dalším popisovaným uzlem je uzel číslo 5. Jeho první vstup je napojen na prvek číslo 6, kterým je uzel číslo 3. Druhý vstup je napojen na prvek číslo 4, kterým je uzel číslo 1. Zde je vidět, že parametr L-back musí být větší než jedna. Poslední hodnota 2 udává, že uzel je typu *or*.

Poslední dvě hodnoty chromozomu určují, na které uzly jsou nasměrovány výstupy celého obvodu. V našem případě se jedná o uzly s čísly 5 a 4.

Tabulka 5.1. udává pro úplnost základní parametry CGP pro tento příklad.

5.1.2 Genotyp a fenotyp

„Genotypem rozumíme kombinaci alel, které nese ve svých buňkách konkrétní jedinec. Fenotypem potom označujeme celého jedince, všechny jeho znaky, vlastnosti a projevy“ [16].

U kartézského genetického programování tvoří genotyp chromozom pevné délky, který obsahuje celá čísla. Fenotypem je orientovaný acyklický graf reprezentující samotný obvod, který může mít proměnnou, avšak konečnou velikost v závislosti na tom, kolik funkčních jednotek je v poli aktivních [14].

5.1.3 Neutralita a redundance

U kartézského genetického programování je časté, že několik různých genotypů vytváří stejný fenotyp. To zvyšuje četnost neutrálních mutací [14]. Neutrální mutace je mutace, po jejímž provedení se nezmění hodnota fitness funkce kandidátního řešení. Opakem je mutace adaptivní, po jejímž aplikování dojde ke změně hodnoty fitness funkce [16]. Literatura o CGP často uvádí, že velké množství neutrálních mutací umožňuje lepší procházení stavového prostoru úlohy [14].

V kartézském genetickém programování se vyskytuje velké množství redundance na různých úrovních:

1. **Na úrovni uzlů**
   Velké množství uzlů nemusí být součástí fenotypu. Geny, které kódují vlastnosti těchto uzlů, jsou redundantní [14].

Četnost mutace v %	Průměrný počet generací	Úspěšnost evoluce v %
0,2	7 588 148	41
0,4	8 500 657	49
0,8	14 661 452	54
1,6	46 242 803	19
proměnná 0,2–1,6	**5 321 766**	**58**

Tabulka 5.2: Porovnání konstantní četnosti mutace s proměnnou rychlostí mutace u násobičky 3×3 bity. Převzato z [23].

2. **Funkční redundance**
   Nastává, pokud je určitá funkce uvnitř CGP vytvářena více uzly než je potřeba [14].

3. **Vstupní redundance**
   Mohou existovat primární vstupy, které nejsou u kandidátního řešení zapojeny [14].

5.1.4 Získávání nových kandidátních řešení

Pro získávání nových kandidátních řešení používáme pouze mutaci. Tento operátor pracuje tak, že předem stanovený počet náhodně vybraných genů zamění za nově náhodně vygenerované přípustné hodnoty [19]. Počet mutovaných genů může být zadán buď absolutně, nebo relativně, a to vzhledem k celkovému počtu genů v chromozomu.

Wang a Lee popisují metodu, jakým způsobem zakódovat četnost mutace do chromozomu. Tato metoda umožňuje, aby se četnost mutace vhodně měnila v průběhu evoluce. Pro každý sloupec CGP může četnost mutace nabývat jedné ze čtyř hodnot (0,2 %, 0,4 %, 0,8 % a 1,6 %). Metoda byla testována na CGP poli o šesti sloupcích a dvanácti řádcích. Jako testovací obvod byla vybrána binární násobička s 3b vstupy. Z výsledků tabulky 5.2. vyplývá, že bylo dosaženo zlepšení, a to jak v průměrném počtu generací potřebných pro vývoj, tak v úspěšnosti evoluce [23].

5.1.5 Průběh evoluce

K ohodnocení kvality kandidátního obvodu slouží tzv. fitness funkce. Její výpočet se liší dle úlohy, pro kterou je CGP použito. Způsoby těchto výpočtů jsou podrobně popsány v části 5.2., samotná evoluce pak probíhá pomocí následujícího algoritmu.

Algoritmus 6 Evoluce CGP (převzato z [16])

Inicializuj populaci $1 + \lambda$ jedinců.
Spočítej hodnotu fitness funkce u všech jedinců.
Vyber nejlepšího jedince, pokud máme několik nejlepších jedinců, použij toho, který v předchozí generaci nebyl rodičem.
Vytvoř λ mutantů vybraného jedince.
Vybraného jedince a jeho λ potomků použij jako další populaci.
Jestliže nebyla splněna ukončovací podmínka, vrať se na druhý krok algoritmu.

5.2 Použití CGP

Kartézské genetické programování má mnoho použití, zejména v oblasti evolučního návrhu kombinačních obvodů a symbolické regresi. V této práci se zaměříme na evoluční návrh binárních násobiček a násobiček s vícenásobnými konstantními koeficienty.

5.2.1 Binární násobičky

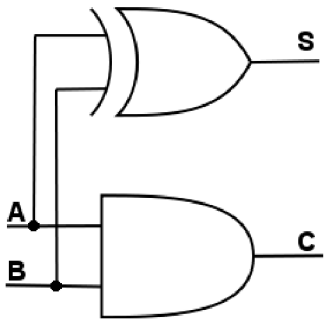
Typickou testovací úlohou pro testování kartézského genetického programování je evoluční návrh kombinačních binárních násobiček [18, 23, 5, 19]. Binární kombinační násobička je obvod, který má $n+m$ binárních vstupů a $n+m$ binárních výstupů, kde prvních n vstupů reprezentuje operand a a dalších m vstupů operand b, výstupy obvodu reprezentují výsledek c, přičemž platí $c = ab$ [2].

Konvenční řešení se skládá z poloviční, úplných sčítaček a hradel *and* [2]. Poloviční sčítačka je obvod, který má dva binární vstupy a dva binární výstupy. Funkčnost poloviční sčítačky je dána následující Tabulkou 5.3.

A	B	S	C_{out}
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Tabulka 5.3: Pravdivostní tabulka pro poloviční sčítačku

Obyčejně je poloviční sčítačka realizována pomocí hradel *and* a *xor*. Schéma zapojení je znázorněno na obrázku 5.2.



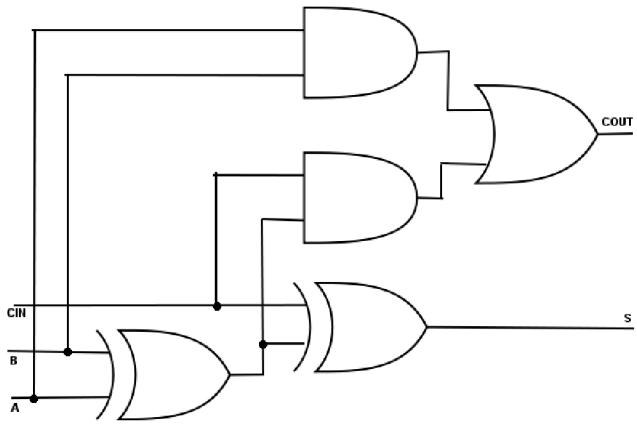
Obrázek 5.2: Schéma zapojení poloviční sčítačky

Úplná sčítačka je obvod se třemi binárními vstupy C_i, A, B a dvěma binárními výstupy C_{out} a S [2]. Její funkčnost je definována následující tabulkou (Tabulka 5.4).

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

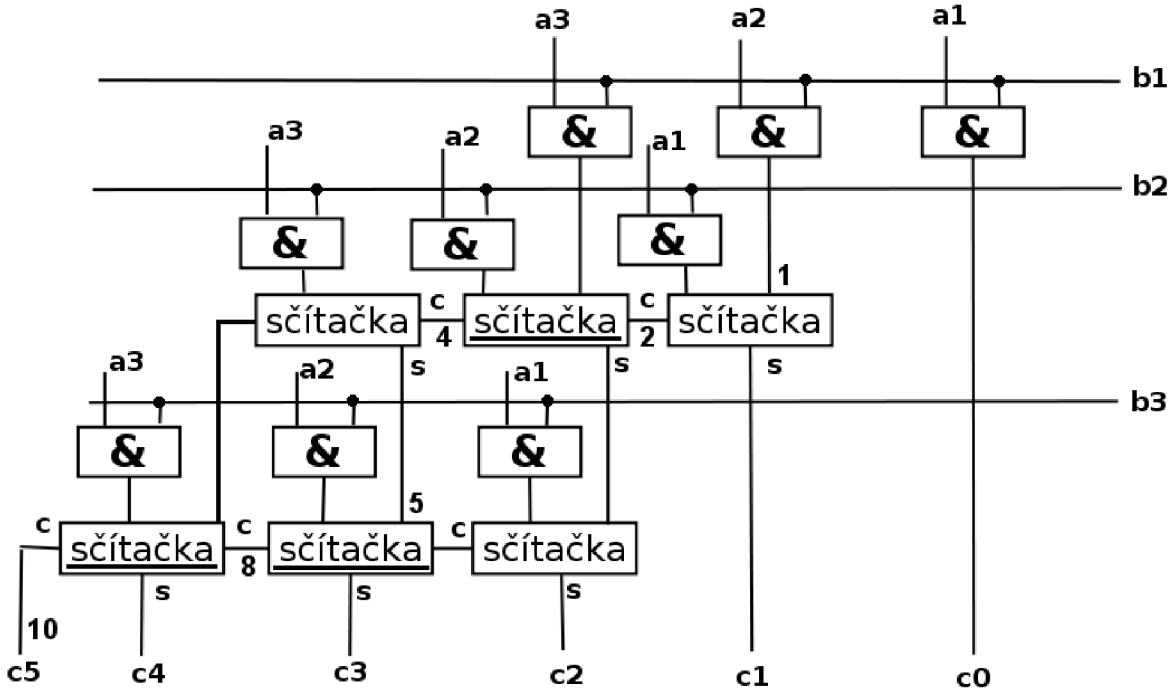
Tabulka 5.4: Pravdivostní tabulka pro úplnou sčítačku.

Tradičně je tento obvod úplné sčítačky implementován dle schématu na obrázku 5.3.



Obrázek 5.3: Schéma zapojení úplné sčítačky

Konvenční násobička pro 3 bitové operandy je pak zapojena dle obrázku 5.4. [2], kde & symbolizuje hradlo *and*, jednotka sčítačka bez podtržení poloviční sčítačku a sčítačka s podtržením úplnou sčítačku. Nejdelší cesta obvodem je pak vyznačena pomocí hodnot, které udávají zpoždění jednotlivých jejích částí. Důležité je upozornit, že konvenčí binární násobička se skládá pouze z hradel *and*, *or* a *xor*. Uvedené řešení není zdaleka optimální, zpoždění lze redukovat zavedením například takzvaného uchování přenosu.



Obrázek 5.4: Schéma zapojení binární násobičky dle [2]

Při vytváření kombinační binární násobičky pracuje CGP na úrovni hradel. Je možné použít i jiné typy hradel než *and*, *or* a *xor* použitých v konvenčním řešení. Pokud CGP pracuje na úrovni hradel, je nutné ověřit 2^n kombinací vstupů obvodu, kde n je počet binárních vstupů [19]. Není možné vynechat žádnou kombinaci vstupních hodnot. Příčinou je neschopnost CGP v tomto případě generalizovat [16].

Kombinace vstupů se na primární vstupy přiřazují postupně dle hodnot jednotlivých řádků pravdivostní tabulky. Následně jsou vypočítány výstupy a porovnány s požadovanými výstupy v příslušném řádku pravdivostní tabulky. Hodnota fitness funkce udává počet shodných výstupních hodnot s požadovanými výstupními hodnotami pro všechny kombinace vstupů z pravdivostní tabulky [19]. Při vývoji je cílem maximalizovat hodnotu této fitness funkce. Při evolučním návrhu na úrovni hradel prudce vzrůstá počet potřebných generací s rostoucím počtem vstupů a výstupů obvodu [18, 16].

5.2.2 Násobičky s vícenásobnými konstantními koeficienty

Násobičky s vícenásobnými konstantními koeficienty mají jeden vstup a několik výstupů, přičemž platí, že na vstup přiložíme jednu hodnotu a na jednotlivých výstupech se objeví tato hodnota vynásobená požadovanými, předem danými konstantami [20].

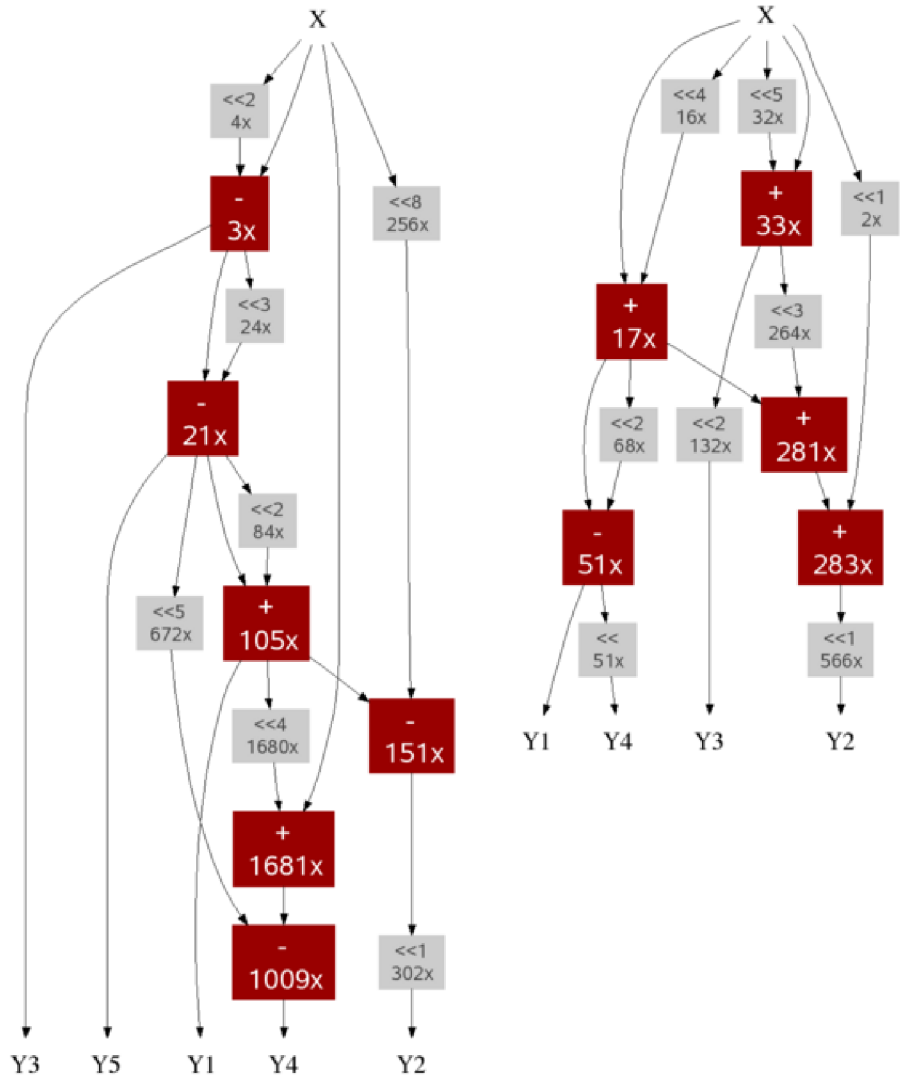
Tento druh násobiček je často používán při zpracování digitálních signálů [16], například při návrhu FIR filtrů [20]. Jejich využití může být jak na hardwarové úrovni (vytvořením příslušného obvodu dle schématu), tak na softwarové úrovni implementací schématu výpočtu pomocí procesoru. Zvláště vhodné je použití softwarového přístupu u jednoduchých procesorů, které nemají jednotku násobení, ale podporují operace sčítání a bitového posuvu [21].

Návrh těchto násobiček představuje NP úplný problém [21]. Za konvenční řešení lze

považovat heuristický přístup popsaný v publikcaci Voronenko et al. [21]. Násobičky lze podle tohoto řešení generovat za pomocí appletu, který je bezplatně dostupný na Internetu [25].

V literatuře najdeme práci o evolučním návrhu těchto násobiček pomocí CGP [21]. Evoluce zde probíhá na úrovni funkčních bloků, což umožňuje navrhovat složitější obvody, než pokud pracujeme na úrovni běžných hradel. Na úrovni hradel je dnes možné navrhovat pomocí CGP obvody s asi 10 až 12 vstupy [16].

Jako funkční bloky jsou použity operace sčítání, odčítání a bitový posuv [20]. Na obrázku 5.5 jsou uvedeny příklady násobiček složených z těchto bloků. Použití těchto bloků zaručuje, že bude výsledný obvod lineární a tudíž stačí pro vyhodnocení správnosti obvodu ověřit funkčnost pouze pro jednu nenulovou hodnotu vstupu. Typicky se používá hodnota 1, což je velice výhodné, protože můžeme rychle ohodnotit kvalitu kandidátních řešení [20, 16].



Obrázek 5.5: Příklady MCM násobiček - vygenerováno pomocí appletu [25].

Hodnota fitness funkce se vypočítá jako součet odchylek výstupů obvodu od hodnot

požadovaných [16]. Přičemž se snažíme o minimalizaci hodnoty fitness funkce. Obvod lze považovat za korektní tehdy, pokud hodnota fitness funkce dosáhne nuly.

Dalším parametrem, na který je možné obvod optimalizovat, je počet uzlů [16]. Operace bitového posuvu je typicky realizována na mnohem menší ploše než operace sčítání a odčítání. Obvod je z tohoto důvodu vhodné optimalizovat na počet operací sčítání a odčítání. Posledním kritériem je zpoždění obvodu.

5.2.3 Další aplikace

Mezi další případná využití kartézského genetického programování, která by se mohla stát potenciálně vhodnými testovacími úlohami, patří:

- návrh obvodů na úrovni tranzistorů,
- klasifikace pomocí CGP,
- symbolická regrese a
- tvorba obrazových filtrů [16].

Kapitola 6

Mutlikriteriální optimalizace v CGP

Jako jediné kritérium pro ohodnocení kvality kandidátních obvodů slouží v klasickém CGP správnost obvodu. To bohužel u praktických aplikací nestačí. Mohlo by se totiž stát, že obvod, který vyvineme je sice z hlediska správné funkce korektní, ale obsahuje například příliš mnoho jednotek, má neúnosné zpoždění a podobně [17].

Z těchto důvodů je nutné zohledňovat při vývoji další kritéria. Problémem je, jakým způsobem v takovém případě sestavit fitness funkci pro ohodnocení kandidátního řešení. Tato kapitola představuje několik možných řešení tohoto problému.

6.1 Dvoustupňová fitness funkce

Tato metoda umožňuje kromě optimalizace správné funkčnosti obvodu zohledňovat při optimalizaci ještě jedno další kritérium. Vyhodnocování fitness funkce probíhá ve dvou stupních, a to v návrhovém stupni (design stage) a v optimalizačním stupni (optimization stage).

V návrhovém stupni je vyhodnocována pouze správnost obvodu. Ta je ohodnocena pomocí fitness funkce f_1. Pokud kandidátní řešení není zcela funkční, další stupeň není dále vyhodnocován a výsledná fitness funkce je dána následujícím vzorcem

$$f = f_1.$$

Pokud je obvod v první fázi vyhodnocen jako zcela funkční, provede se druhý stupeň vyhodnocování fitness funkce. V něm je obvod ohodnocen pomocí předem daného kritéria. Hodnota fitness funkce se při tomto ohodnocování označuje jako f_2. Výslednou hodnotu fitness funkce lze poté spočítat dle vztahu [23]:

$$f = f_1 + f_2.$$

U obou fitness funkcí f_1 a f_2 usilujeme o maximalizaci hodnot. Toto řešení bylo úspěšně použito v několika případech [19, 23, 20]. Jeho hlavní nevýhodou je to, že umožňuje optimalizaci pouze na jedno další kritérium, což je v praxi ne vždy dostačující.

Alternativně je možné optimalizovat kritéria $f_2,\ldots,f_n$, která však musíme sloučit pomocí vhodně zvolených vah. O nevýhodách tohoto přístupu jsme hovořili v kapitole 3.

6.2 Omezení velikosti pole CGP

Tato metoda byla použita v při hledání binárních násobiček o velikosti 4×3 bity [19]. Pomocí dvoustupňové fitness funkce byl minimalizován počet hradel a pro dosažení nízkého zpoždění byl omezen počet sloupců.

Nevýhodou tohoto přístupu je, že při zmenšení pole v podstatě zpřísníme požadavky na evoluci. Z toho plyne, že klesne pravděpodobnost úspěšného nalezení řešení [19].

Omezením obou rozměrů CGP pole můžeme dosáhnout takových řešení, které bude mít nízký počet hradel. Cenou za tento přístup je však opět menší pravděpodobnost úspěšného nalezení řešení.

6.3 Přístupy založené na algoritmu NSGAII

V mnoha případech se i optimalizace na jedno další kritérium ukazuje jako nedostatečná a je třeba provést optimalizaci dle mnoha kritérií [5, 4, 22]. K tomuto účelu je vhodné použít algoritmus NSGAII [1].

Tento algoritmus je vhodný zejména proto, že podporuje diverzitu populace. To znamená, že výsledná řešení jsou pokud možno rozprostřena přes celou Paretovu frontu [5].

Metoda pro použití NSGAII při vývoji obvodů pomocí kartézského genetického programování popsaná autory Hilder et al., počítá fitness funkci ve dvou stupních [5]. Prvním stupněm je obdobně jako u metody popsané v sekci 6.1. ohodnocení správné funkčnosti obvodu.

Teprve tehdy, jakmile je obvod po stránce funkcionality správný, je proveden výpočet dalších požadovaných fitness funkcí. Pomocí hodnot těchto fitness funkcí jsou kandidátní řešení seřazena postupem používaném algoritmem NSGAII [5].

Na rozdíl od klasického kartézského genetického programování, kde se při vývoji používá strategie $(1 + \lambda)$, se u tohoto postupu používá strategie $(\mu + \lambda)$ [5]. Ta pracuje tak, že zpočátku je náhodně vygenerováno μ jedinců počáteční populace. Z každého jedince je pomocí operátoru mutace vygenerováno λ potomků. Tito potomci jsou ohodnoceni pomocí dříve popsaného postupu. Do další generace se dostane μ nejlepších jedinců z rodičů a jejich potomků. Pokud jsou rodič a potomek stejně kvalitní, do další generace je použit potomek.

Tato metoda využití NSGAII při kartézském genetickém programování byla úspěšně použita pro návrh dvou a tříbitových kombinačních binárních násobiček, dvou a tříbitových sčítaček a ovladače pro sedmisegmentový displej. Uvedené obvody byly optimalizovány na počet hradel, počet tranzistorů, nejdelší cestu na úrovni hradel a nejdelší cestu na úrovni tranzistorů [5].

Uvedená metoda pak byla použita i ke generování obvodů v technologii CMOS. Pro simulaci takto vytvořených obvodů byl použit simulátor SPICE. Navrhována byla jednoduchá hradla *nand*, *and*, *nor*, *or*, *xnor*, *xor*, dále pak *D flip-flop* [4, 22].

Kapitola 7

Navržený systém

7.1 Specifikace

V rámci této diplomové práce byl implementován program umožňující hledání jednoduchých číslicových obvodů pomocí kartézského genetického programování a jejich optimalizaci na několik kritérií. Výstup tohoto programu je ukládán jako XML soubor a obsahuje chromozómy s řešeními a hodnocení kvality těchto řešení. Zároveň je vytvořen textový soubor s informacemi o průběhu evoluce, tj. například vývojem jednotlivých fitness funkcí v průběhu času.

K automatizovanému spouštění skriptů slouží skript *benchmarkmanager.py*, který dovoluje spustit předepsaný počet běhů programu a navíc se dokáže vypořádat s časovými omezeními pro běh programů nastavených na některých výpočetních serverech.

Pro analýzu výsledků experimentů slouží skript *parser.py*, který slouží k automatizovanému vyhodnocování souborů s průběhem evoluce a poskytuje souhrnné informace o automatizovaných testech.

Výsledná řešení je možné zobrazovat pomocí programu *ChromosomeViewer*, který dovolí graficky zobrazit vygenerovaný obvod. Při prohlížení lze simulovat výstupy jednotlivých uzlů obvodu podle zadaných vstupů. Výsledek je také možné uložit do souboru jako obrázek.

7.2 Algoritmus

Program je založen na algoritmu NSGAII a vývoj obvodu provádí pomocí kartézského genetického programování. Při vývoji požívá dvoustupňovou fitness funkci, kde první stupeň tvoří hodnocení správné funkčnosti obvodu, tedy správnost výstupů.

Přičemž jako lepší je vyhodnocen obvod s menší odchylkou od požadovaných výstupů. Obvody, které jsou z funkčního hlediska zcela správné, jsou vybrány a jsou ohodnoceny podle dalších kritérií. Dále je nad nimi provedeno nedominované řazení (nondominated sorting) algoritmu NSGAII a je jim přiřazena hodnota vzdálenost pokrytí.

Obě tyto operace jsou popsány v části o algoritmu NSGAII. Poté je možné jednotlivá řešení porovnat pomocí operátoru $\prec_n$. Správnost obvodu je nyní uvažována jako omezení x_{cv}.

Při vývoji je použita strategie $(\mu + \lambda)$ a počáteční generace je zvolena náhodně. Vývoj je zastaven po dosažení určitého uživatelem zadaného počtu generací.

Algoritmus 7 Multikriteriální CGP

Náhodně vygeneruj μ rodičů P_0
$t = 0$
while $(t < max)$ **do**
  Pomocí mutace vygeneruj pro každého rodiče λ potomků (Q_t - množina potomků).
  $R_t = P_t \cup Q_t$
  for all $r \in R_t$ **do**
    Vypočítej r_{cv} (míra porušení omezení, pokud $r_{cv} = 0$ obvod má správnou funkčnost).
  end for
  $R_{t,s}$ je množina prvků R_t, které mají $r_{cv} = 0$.
  Nad prvky $R_{t,s}$ proveď nedominované řazení z algoritmu NSGAII (tím je pro každé řešení z $R_{t,s}$ spočítána hodnota $rank$).
  Množiny $R_{t,s1}$, $R_{t,s2}$,... obsahují řešení z $R_{t,s}$ s $rank = 1, 2, ...$
  Nad každou množinou $R_{t,si}$ proveď algoritmus přiřazení vzdálenosti pokrytí z NSGAII (tím je pro každé řešení z těchto množin vypočtena hodnota $distance$).
  Seřaď$(R_t, \prec_n)$ (seřaď množinu R_t dle relace $\prec_n$ - verze této relace pro CGP byla definovaná v kapitole 3.4.4).
  Do P_{t+1} vyber prvních μ jedinců ze seřazeného R_t.
  $t = t + 1$
end while

7.3 Testovací úlohy

Program umožňuje evoluční design binárních kombinačních násobiček a násobiček s vícenásobnými konstantními koeficienty (MCM). Obě tyto úlohy jsou podrobně popsány v kapitole 5.3.2 o kartézském genetickém programování.

Kapitola 8

Implementace

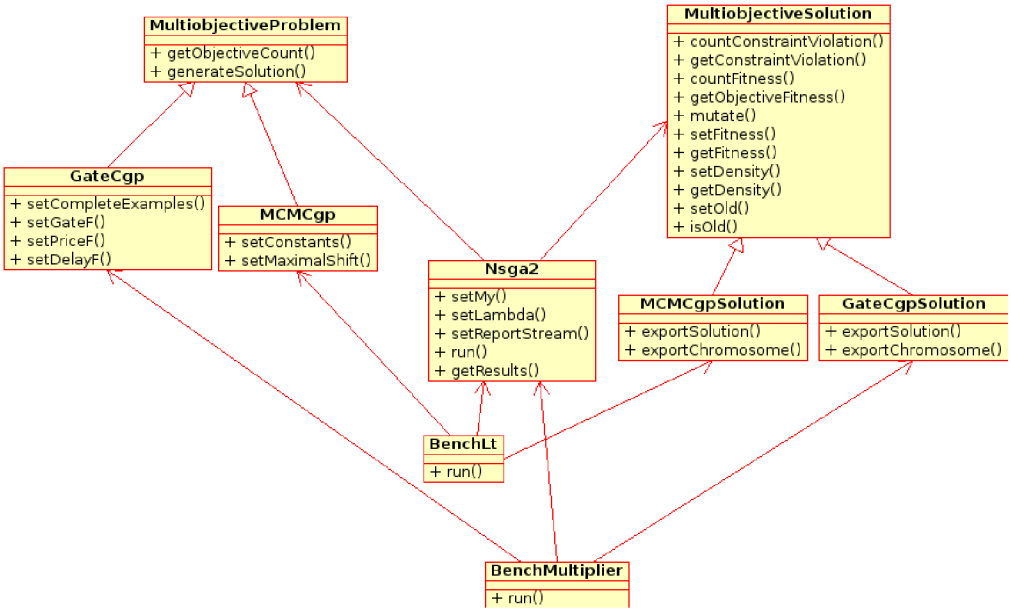
8.1 Použité technologie

Program je implementován v objektově orientovaném jazyce C++ s použitím pouze standardních knihoven. To umožňuje vysokou přenositelnost mezi různými platformami. Dále pak jazyk C++ umožňuje psaní programů na poměrně nízké úrovni, což umožňuje psát efektivní programy [10].

Program je plně funkční na školních serverech Edesign1 a Edesign2, na kterých bylo prováděno veškeré testování.

8.2 Vnitřní architektura

Program je naprogramován s využitím objektově orientovaného přístupu, což umožňuje jeho snadnou rozšiřitelnost o další testovací obvody. Diagram tříd je uveden na obrázku 7.1.



Obrázek 8.1: Diagram tříd zachycující architekturu programu.

8.2.1 Třída MultiobjectiveProblem

Třída MultiobjectiveProblem je abstraktní třída pro reprezentaci problému, který je třeba optimalizovat na více kritérií. Tuto třídu dědí třídy GateCgp a MCMCgp. Objekty těchto tříd jsou předávány konstruktoru třídy Nsga2, která obsahuje algoritmus pro multikriteriální optimalizaci. Do této třídy patří metody:

- **getObjectiveCount**
  Vrací počet kritérií, na které je třeba řešení problému optimalizovat.
- **generateSolution**
  Vygeneruje náhodného jedince.

8.2.2 Třída GateCgp

Třída GateCgp reprezentuje problém řešitelný pomocí CGP pracujícího na úrovni hradel. Objekty této třídy jsou použity ke hledání binárních násobiček. Třída je potomkem třídy MultiobjectiveProblem. Do této třídy patří metody:

- **Konstruktor GateCgp**
  Vytvoří objekt typu GateCgp. Jako parametry je nutné zadat počet primárních vstupů a výstupů obvodu, rozměry CGP mřížky, parametr L a velikost mutace.
- **setCompleteExamples**
  Slouží pro nastavení pravdivostní tabulky vyvíjeného obvodu.
- **setGateF**
  Nastavuje, zda při vývoji bude obvod optimalizován na počet hradel. A jejím parametrem je logická hodnota.
- **setPriceF**
  Nastavuje, zda při vývoji bude obvod optimalizován na celkovou plochu. A jejím parametrem je logická hodnota.
- **setDelayF**
  Nastavuje, zda při vývoji bude optimalizován na zpoždění. A jejím parametrem je logická hodnota.

8.2.3 Třída MCMCgp

Tato třída reprezentuje problém řešitelný pomocí CGP pracujícího na úrovni operací popsaných v podkapitole 5.2.2 o MCM násobičkách. Třída je potomkem třídy MultiobjectiveProblem. Do této třídy patří metody:

- **Konstruktor MCMCgp**
  Vytvoří objekt typu MCMCgp. Jako parametry je třeba zadat počet primárních vstupů, výstupů obvodu, rozměry mřížky, parametr L a velikost mutace.
- **setConstants**
  Nastaví konstanty pro násobení MCM pro výsledný obvod.
- **setMaximalShift**
  Nataví maximální velikost binárního posuvu.

8.2.4 Třída MultiobjectiveSolution

Tato abstraktní třída slouží pro reprezentaci řešení problému, který je třeba optimalizovat na více kritérií. Tuto třídu dědí třídy GateCgpSolution a MCMCgpSolution. Do této třídy patří metody:

- **countConstraintViolation**
  Spočítá hodnotu omezení, která je pak uložena v objektu a je jí možno načíst pomocí metody getConstraintViolation. Tato hodnota v případě CGP vystihuje správnou funkci obvodu.
- **getConstraintViolation**
  Vrací spočítanou hodnotu omezení.
- **countFitness**
  Spočítá hodnoty všech fitness funkcí. Tyto hodnoty jsou uloženy v objektu a je možné se na ně dotazovat pomocí metody getObjectiveFitness.
- **getObjectiveFitness**
  Vrací spočítanou hodnotu fitness. Parametrem je číslo fitness funkce.
- **mutate**
  Vytvoří nové řešení pomocí mutace.
- **setFitness**
  Nastaví rank řešení (rank viz kapitola o NSGAII).
- **getFitness**
  Vrací rank řešení.
- **setDensity**
  Nastaví hodnotu vzdálenost pokrytí.
- **getDensity**
  Vrací hodnotu vzdálenosti pokrytí.
- **setOld**
  Nastaví příznak, že řešení již bylo v předchozí generaci.
- **isOld**
  Vrací true, pokud bylo řešení obsaženo už v předchozí generaci.

8.2.5 Třída GateCgpSolution

Objekty této třídy reprezentují řešení pro CGP pracující na úrovni hradel. Třída je potomkem MultiobjectiveSolution. Do třídy patří metody:

- **exportSolution**
  Zapíše informace o řešení do výstupního streamu.
- **exportChromosome**
  Vrací chromozóm jako řetězec.

8.2.6 Třída MCMCgpSolution

Objekty této třídy reprezentují řešení pro CGP pracující na úrovni operací popsaných v podkapitole 5.2.2. Třída je potomkem MultiobjectiveSolution. Do třídy patří metody:

- **exportSolution**
  Zapíše informace o řešení do výstupního streamu.
- **exportChromosome**
  Vrací chromozóm jako řetězec.

8.2.7 Třída Nsga2

NsgaII je třída, v které je implementován algoritmus NSGAII. Nejprve je jí v konstruktoru nutné předat instanci třídy MultiobjectiveProblem. Následně se nastaví parametry evoluce (μ, λ a počet generací).

Samotná evoluce se spouští pomocí metody run a výsledné obvody je možné získat pomocí metody getResults. Do této třídy patří metody:

- **Konstruktor NSGAII**
  V konstruktoru je nutné zadat objekt třídy MultiobjectiveProblem.
- **setMy**
  Nastaví parametr μ.
- **setLambda**
  Nastaví parametr λ.
- **setReportStream**
  Nastaví stream, do kterého jsou průběžně posílány zprávy o stavu evoluce.
- **run**
  Spustí výpočet.
- **getResults**
  Vrací vygenerovaná řešení.

8.2.8 Třída BenchMultiplier

Objekty této třídy umožňují vývoj binárních kombinačních násobiček. Do této třídy patří metody:

- **Konstruktor BenchMultiplier**
  V konstruktoru je nutné zadat počet bitů prvního a druhého operandu násobičky, rozměr matice CGP a parametr LBack.
- **run**
  Spustí evoluci násobičky. Jako parametr je nutné zadat μ a λ, výstupní soubor a maximální počet generací evoluce. Výsledek je uložen do výstupního souboru.

8.2.9 BenchLT

Objekty této třídy umožňují vývoj násobiček s vícenásobnými konstantními koeficienty. Do této třídy patří metody:

- **Konstruktor BenchLt**
  V konstruktoru je nutné zadat konstanty násobičky, rozměr matice CGP a parametr LBack.

- **run**
  Spustí evoluci násobičky. Jako parametr je nutné zadat μ a λ, výstupní soubor a maximální počet generací evoluce. Výsledek je uložen do výstupního souboru.

8.3 Skript pro automatizované spouštění testů

Při testování stochastických algoritmů je nutné spustit několik běhů daného algoritmu. Typicky se jedná o dvacet a více běhů. Provádět toto ručně by bylo zbytěčně náročné. Pro automatizaci spouštění testů je navržen skript *benchmarkmanager.py*.

Skript je napsán v jazyce Python. Tento jazyk se slabým typováním umožňuje efektivní psaní jednoduchých programů [3] a je tedy pro podobné účely ideální.

Dalším problémem, s kterým se skript umožňuje vypořádat, jsou omezení doby pro běh programů u některých výpočetních serverů. Skript umožňuje uložení dočasných výsledků a znovuspuštění programu z místa, kde byl výpočet přerušen.

8.4 Skript pro sumarizaci výsledků testů

Tento skript je opět napsán v jazyce Python [3] a slouží k sumarizaci výsledků sady testů, která byla vytvořena pomocí skriptu *benchmarkmanager.py*.

Poskytuje informace o nejlepších a průměrně dosažených hodnotách fitness funkcí, průměrném a nejmenším počtu generací potřebném pro vývoji. Dále pak o průměrném a nejvyšším počtu bodů v Paretově frontě.

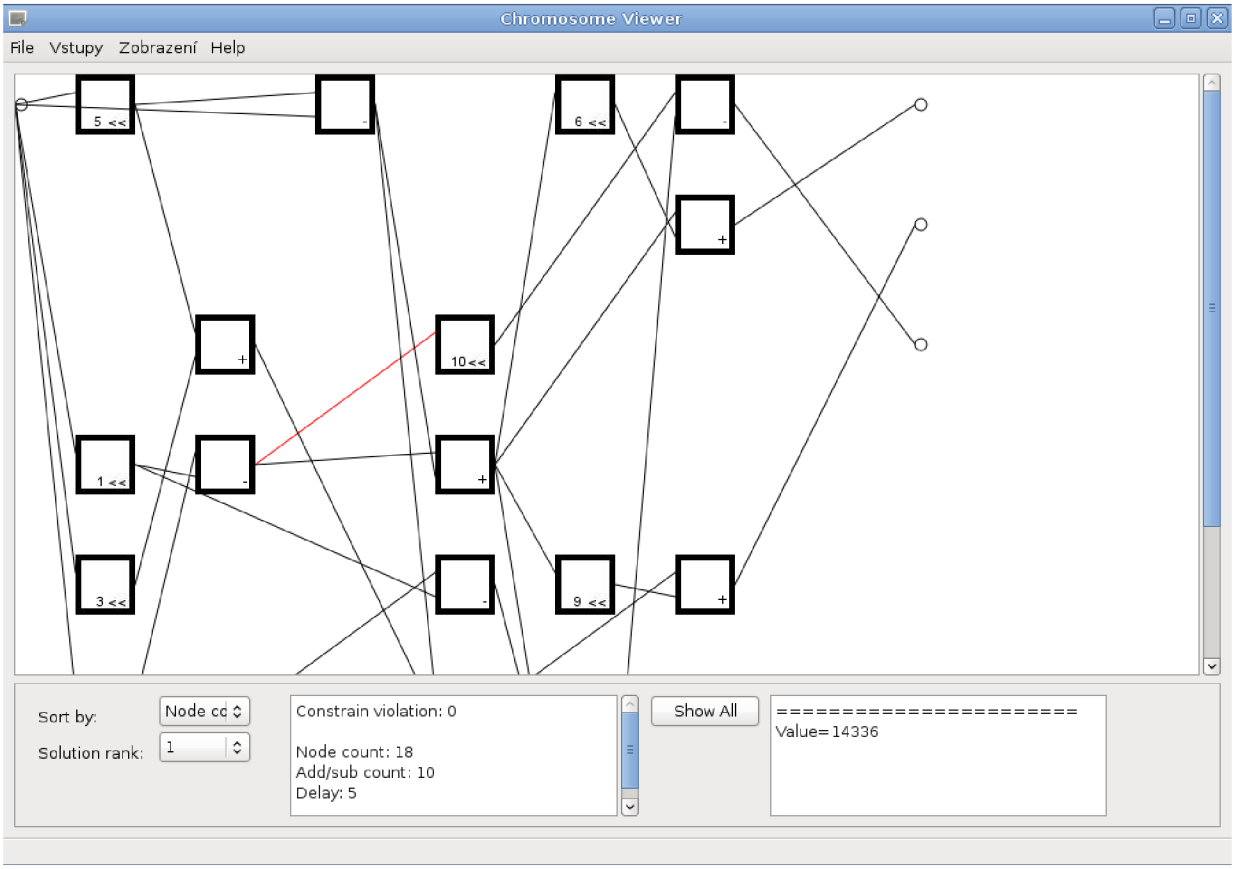
8.5 Program pro zobrazování výsledků

Aby bylo možné prohlédnout si jednotlivé vygenerované obvody, byl v rámci práce naprogramován prohlížeč *ChromosomeViewer*.

Vstupem programu je xml soubor vygenerovaný pomocí dříve popsaného programu pro generování číslicových obvodů. Tento soubor obsahuje několik možných zapojení obvodu. Ty je možné seřadit dle jednotlivých kritérií a zobrazovat.

Při zobrazování je možné zadat hodnoty vstupů obvodu a sledovat hodnoty na výstupech a na jednotlivých uzlech obvodu. Uzlem obvodu je myšlen prvek mřížky CGP například hradlo, sčítačka a podobně, záleží na charakteru úlohy.

Další vlastností programu je schopnost ukládat schémata vygenerovaných obvodů do souboru. Podoporovaným formátem pro export schématu je obrázkový formát GIF.



Obrázek 8.2: Ukázka programu pro zobrazování vygenerovaných obvodů s načtenou MCM násobičkou

8.5.1 Použité technologie

Program je naprogramován v jazyce C++ [10] s použitím knihovny Qt. Jedná se o multiplatformní knihovnu především pro tvorbu grafických uživatelských rozhranní. Navíc umožňuje práci s XML soubory, přenos dat pomocí TCP/IP a další možnosti [24].

Kapitola 9

Výsledky

Systém byl otestován na úlohách evoluce binárních násobiček a násobiček s vícenásobnými konstantními koeficienty. Principy těchto úloh jsou popsány v kapitole 5.2. Byl sledován vliv velikosti parametrů četnosti mutace, μ a λ na kvalitu výsledných řešení a počet řešení na Paretově frontě. Kvalita výsledných řešení byla porovnána s výsledky prací [18, 20] a řešení generovaných pomocí appletu [25].

9.1 Evoluce binárních kombinačních násobiček

9.1.1 Vliv velikosti mutace na kvalitu výsledných řešení

Pro otestování vlivu velikosti mutace na výsledná řešení byla zvolena jako testovací úloha evoluce násobičky se dvěma tříbitovými operandy. V obvodu mohla být použita hradla *and*, *or*, *xor*, *not*, *nand*, *nor* a *xnor*. Velikost CGP pole byla zvolena 8×8. Evoluce násobičky byla testována pro mutace o rozsahu 0,2%, 0,4%, 0,6%, 0,8%, 1%, 2%, 5%, 10%. Rozsah mutace určuje relativně počet mutovaných genů v chromozomu CGP při jedné operaci mutace. Obvod byl optimalizován na počet hradel, počet tranzistorů a zpoždění. Počty tranzistorů, které jsou třeba k realizaci jednotlivých hradel jsou uvedeny v tabulce 9.1. Soubory s výsledky jsou nahrány na přiloženém CD.

Pro každou hodnotu velikosti mutace bylo provedeno dvacet běhů programu. Při hodnocení nastavení byly z každého běhu vybrány nejlepší dosažené hodnoty fitness funkcí. Z těchto hodnot byl vytvořen průměr. Výsledky pro jednotlivá nastavení zachycuje tabulka 9.2. Aby jednotlivá kritéria měla při hledání nejlepšího řešení stejnou váhu, byly jejich hodnoty normalizovány do intervalu $< 0, 1 >$, tyto normalizované hodnoty jsou uvedeny v závorkách. Odečtením průměru z takto normalizovaných hodnot od jedničky získáme kvalitu nastavení.

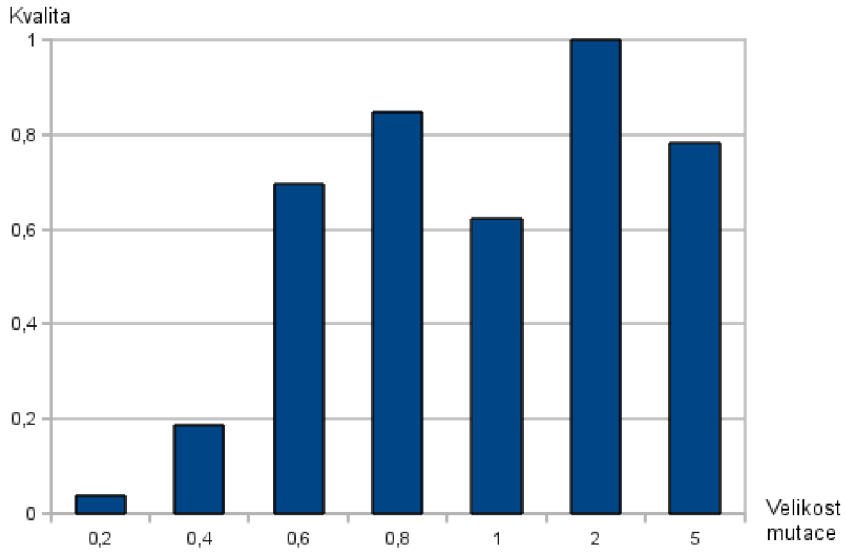
Typ hradla	**Počet tranzistorů**
not	2
nand, *nor*	4
and, *or*	6
xnor	9

Tabulka 9.1: Počty tranzistorů jednotlivých typů hradel dle [5].

Velikost mutace	Počet hradel	Cena	Zpoždění	Kvalita nastavení
0,2	32,55 (0,9)	190,35 (0,99)	6,3 (1)	**0,036**
0,4	33 (1)	190,7 (1)	5,9 (0,444)	**0,185**
0,6	30,45 (0,437)	168,6 (0,38)	5,65 (0,097)	**0,695**
0,8	28,8 (0,073)	162,55 (0,21)	5,7 (0,167)	**0,85**
1	30,55 (0,459)	170,6 (0,436)	5,75 (0,236)	**0,623**
2	28,47 (0)	155,05 (0)	5,58 (0)	**1**
5	29,5 (0,227)	165,83 (0,302)	5,67 (0,125)	**0,782**
10	-	-	-	**-**

Tabulka 9.2: Vliv mutace na kvalitu výsledných řešení.

Kvalita pro jednotlivá nastavení byla pro přehlednost zakreslena do grafu. Zde je jasně patrné, že nejlepších výsledků je dosahováno při velikosti mutace 2%. Při velikosti mutace 10% nebylo nalezeno funkční řešení ani v jednom z běhů.



Obrázek 9.1: Kvalita výsledných řešení při evolučním návrhu násobičky 3×3 bity v závislosti na velikosti mutace.

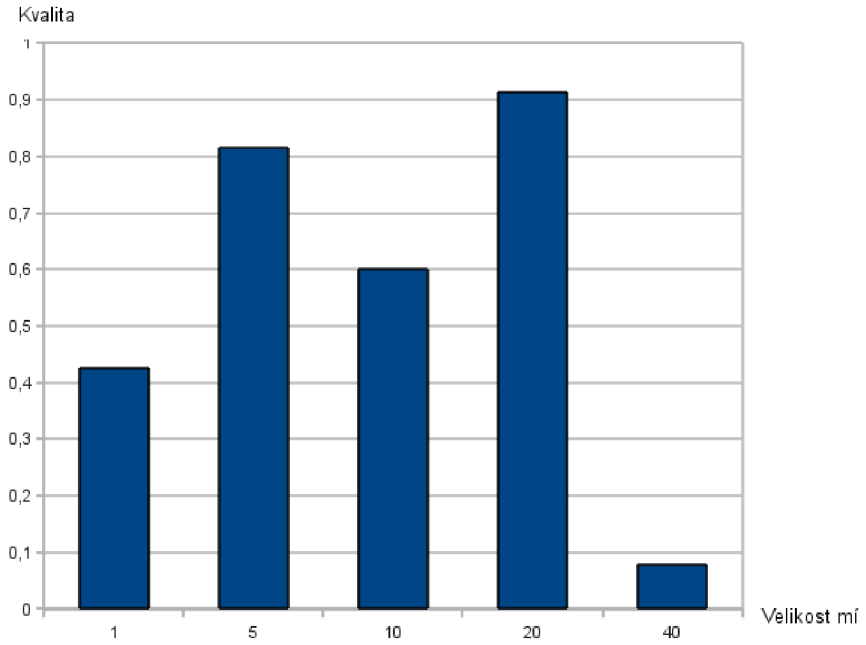
9.1.2 Vliv velikosti parametru μ na kvalitu výsledných řešení

K otestování vlivu velikosti parametru μ na kvalitu řešení byla použita stejná testovací úloha jako v sekci 9.1.1. Testovány byly hodnoty μ =1, 5, 10, 20, 40. Testy pro jednotlivé hodnoty byly spouštěny tak, aby počet ohodnocených kandidátních řešení byl vždy stejný. V každé iteraci evolučního algoritmu je třeba nově ohodnotit $\mu\lambda$ jedinců. Rodiče, jejichž počet je μ, není třeba znova ohodnocovat. Při evoluci byla použita 2% velikost mutace, která se ukázala jako nejvýhodnější v experimentu 9.1.1. Vyhodnocení kvality probíhalo stejným způsobem jako v 9.1.1. Parametr λ byl nastaven na hodnotu 4. Výsledky jsou uvedeny v tabulce 9.3. a soubory s evolučně navržené obvody jsou přiloženy na CD.

μ	Generací	Počet hradel	Cena	Zpoždění	Kvalita
1	4.10^8	28,55 (0,337)	157,5 (0,389)	5,75 (1)	**0,425**
5	8.10^7	28,2 (0)	157,75 (0,428)	5,6 (0,128)	**0,815**
10	4.10^7	28,9 (0,673)	157,55 (0,397)	5,6 (0,128)	**0,601**
20	2.10^7	28,47 (0,26)	155,05 (0)	5,58 (0)	**0,913**
40	1.10^7	29,24 (1)	161,35 (1)	5,71 (0,767)	**0,078**

Tabulka 9.3: Vliv velikosti parametru μ na kvalitu výsledných řešení. V každém běhu bylo ohodnoceno $1,6.10^9$ jedinců.

Kvalita výsledků byla zanesena do grafu 9.2. Jako nejlepší se tedy ukázala velikost μ=20.



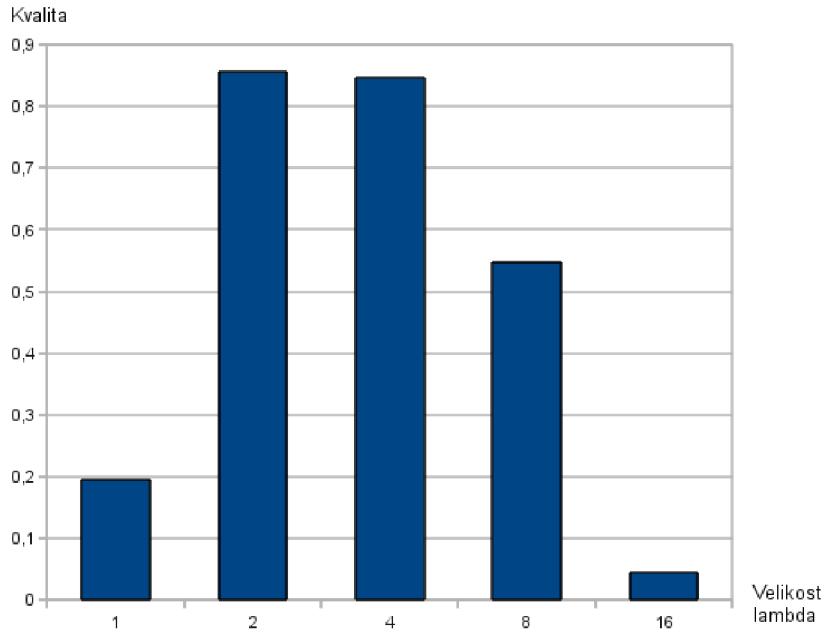
Obrázek 9.2: Kvalita výsledných řešení při evolučním návrhu násobičky 3×3 bity v závislosti na velikosti μ.

9.1.3 Vliv velikosti parametru λ na kvalitu výsledných řešení

Vliv velikosti parametru λ na kvalitu výsledných řešení byl opět testován na úloze popsané v 9.1.1. Byly testovány hodnoty λ =1, 2, 4, 8, 16. Při experimentech byla použita velikost mutace 2% a parametr μ 20. Počet generací pro jednotlivá nastavení byl volen tak, aby byl ohodnocen stejný počet kandidátních řešení. Výsledky jsou shrnuty v tabulce 9.4. a soubory s evolučně navrženými obvody jsou přiloženy na CD. Pro názornost byl opět vytvořen graf 9.3., z kterého plyne, že jako nejvhodnější nastavení se ukázalo pro $\lambda = 2$.

λ	Generací	Počet hradel	Cena	Zpoždění	Kvalita
1	8.10^7	29,7 (1)	165,6 (0,928)	5,65 (0,489)	**0,194**
2	4.10^7	28,3 (0)	155,6 (0,048)	5,6 (0,383)	**0,856**
4	2.10^7	28,47 (0,121)	155,05 (0)	5,58 (0,34)	**0,846**
8	1.10^7	29,26 (0,686)	162,68 (0,671)	5,42 (0)	**0,548**
16	5.10^6	29,52 (0,871)	166,42 (1)	5,89 (1)	**0,078**

Tabulka 9.4: Vliv velikosti parametru λ na kvalitu výsledných řešení. V každém běhu bylo ohodnoceno $1,6.10^9$ jedinců.



Obrázek 9.3: Kvalita výsledných řešení při evolučním návrhu násobičky 3×3 bity v závislosti na velikosti parametru λ.

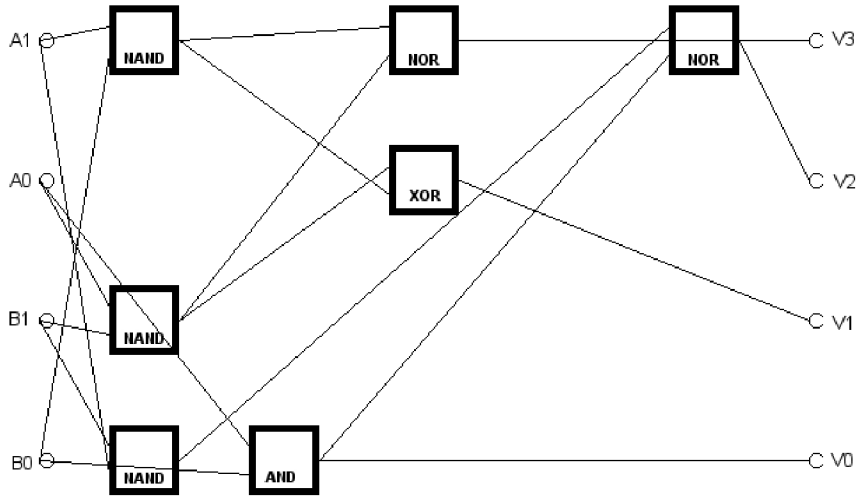
9.1.4 Vývoj násobičky 2×2

Nejmenší násobičkou, která byla v rámci testování vyvíjena, je násobička o dvou dvoubitových operandech. Velikost pole CGP byla zvolena 5×5. Velikost mutace byla nastavena na 2%. Testování proběhlo dvakrát pro dvě různé sady hradel a to nejprve pro sadu schodnou se sadou použitou v [5] tedy *and*, *or*, *xor*, *not*, *nand*, *nor* a *xnor* (označme ji jako I.). U této sady byly obvody optimalizovány na počet hradel, počet tranzistorů a zpoždění. V druhém případě pro sadu použitou v [18], tedy *and*, *xor* a *and* s jedním negovaným vstupem (označme ji jako II.). V tomto případě byly obvody optimalizovány na počet hradel a zpoždění. Pro obě testovací sady bylo provedeno dvacet běhů evolučního algoritmu o $\mu = 20$, $\lambda = 2$ a jednom miliónu generací.

Výsledky evoluce jsou uvedeny v tabulce 9.5. Všechny běhy byly úspěšné a nalezly nejlepší řešení. Výsledky jsou stejné, jako výsledky uvedené v [5]. Paretova fronta obsahovala vždy jedno řešení. Příklad evolučně získané násobičky je na obrázku 9.4.

Experiment	Prům. hradel	Prům. tranzistorů	Prům. zpoždění
Sada hradel I.	7	35	2
Výsledky z [5]	7	35	2
Sada hradel II.	7	-	2

Tabulka 9.5: Výsledky evoluce násobičky 2×2.



Obrázek 9.4: Příklad evolučně navržené násobičky 2×2.

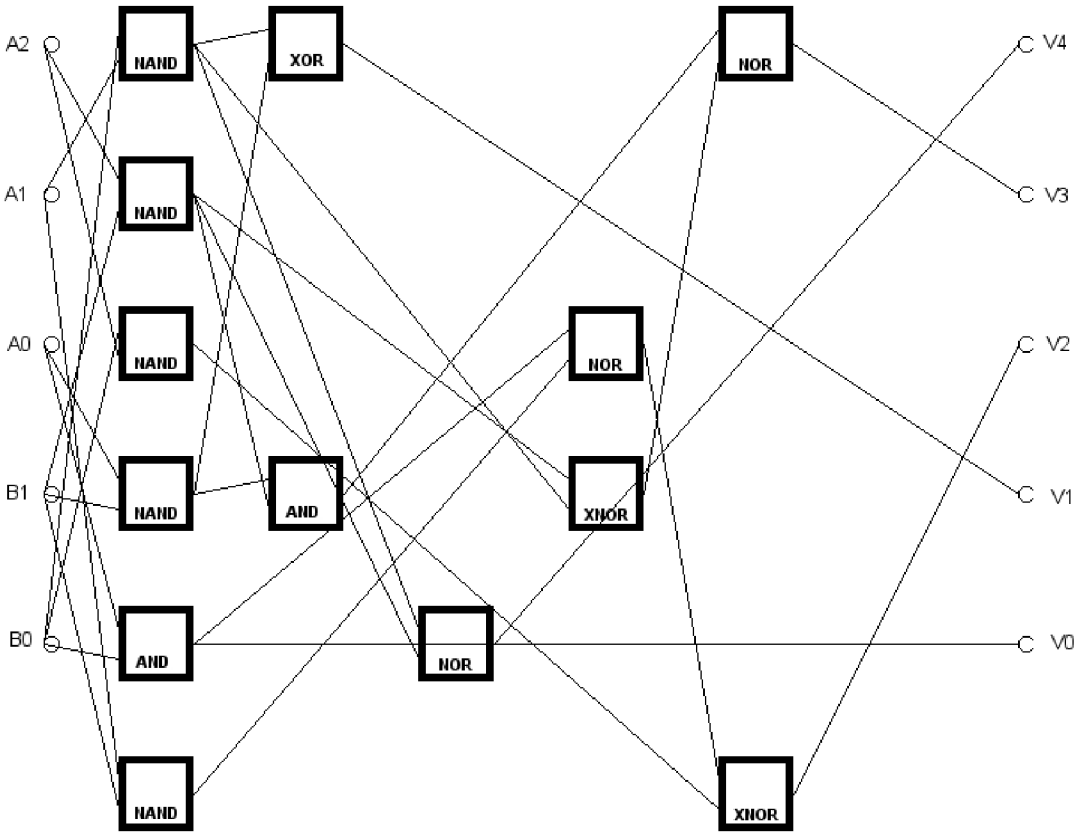
9.1.5 Vývoj násobičky 3×2

Násobička o jednom tříbitovém a jednom dvoubitovém operandu byla vyvíjena s pomocí dvou sad hradel uvedených u příkladu 9.1.4 a byla optimalizována na stejné parametry. Evoluce probíhala na poli 6×6 a parametrem $L = 5$. Velikost mutace byla nastavena na 2%. Pro obě testovací sady bylo provedeno dvacet běhů evolučního algoritmu o $\mu = 20$, $\lambda = 2$ a 2.10^7 generací.

Výsledky evoluce jsou uvedeny v tabulce 9.6. Příklad evolučně získané násobičky je uveden na obrázku 9.5. U sady hradel I. byl průměrný počet výsledků na Pareto frontě 1,6 a nejvyšší počet 3. U sady hrade II. byl v Pareto frontě vždy pouze jeden bod. Nejlepší evolučně navržená násobička byla o 4 hradla menší, než konvenční a měla o jedna nižší zpoždění, než násobička vytvořená v [18].

Experiment	Průměrný výsledek			Nejlepší výsledek		
	Hradel	**Tranzistorů**	**Zpoždění**	**Hradel**	**Tranzistorů**	**Zpoždění**
Sada hradel I.	13,15	68,4	3,25	13	66	3
Sada hradel II.	13	-	3,15	13	-	3
Konv. řešení dle [18]	-	-	-	17	-	-
Výsledek z [18]	-	-	-	13	-	4

Tabulka 9.6: Výsledky evoluce násobičky 3×2.



Obrázek 9.5: Příklad evolučně navržené násobičky 3×2.

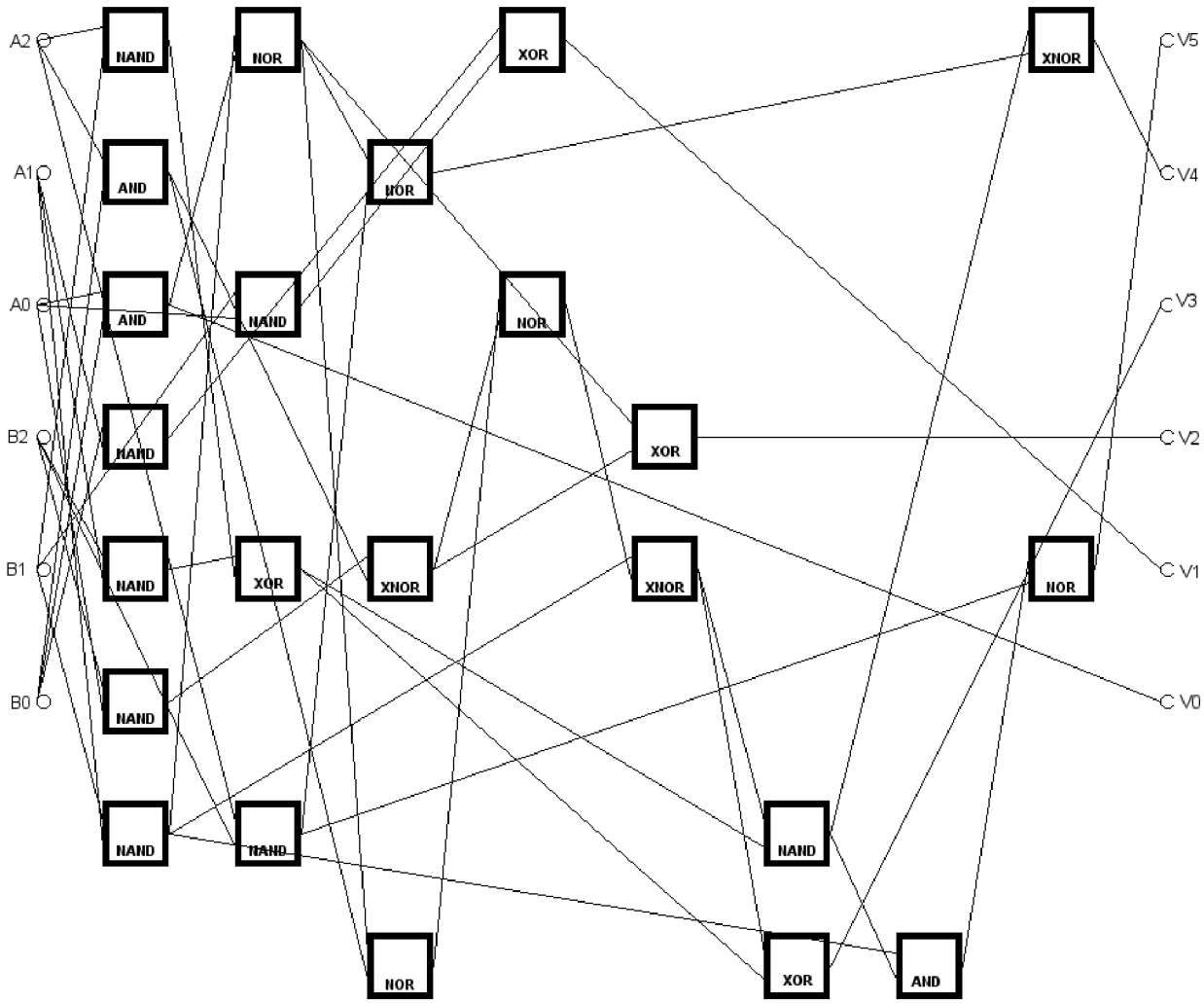
9.1.6 Vývoj násobičky 3×3

Násobička o dvou tříbitových operandech byla vyvíjena pomocí dvou sad hradel uvedených v 9.1.4. a byla optimalizována na stejné parametry. Evoluce probíhala na poli 8×8 a s parametrem $L = 7$. Velikost mutace byla nastavena na 2%. Pro obě testovací sady bylo provedeno dvacet běhů evolučního algoritmu o $\mu = 20$ a $\lambda = 2$ a 4.10^7 generací.

Experiment	Průměrný výsledek			Nejlepší výsledek		
	Hradel	**Tranzistorů**	**Zpoždění**	**Hradel**	**Tranzistorů**	**Zpoždění**
Sada hradel I.	28,3	155,6	5,6	23	133	5
Výsledek z [5]	31,64	177,3	5,86	28	148	5
Sada hradel II.	27,8	-	5,5	25	-	5
Konv. řešení dle [18]	-	-	-	30	-	-
Výsledek z [18]	-	-	-	23	-	7

Tabulka 9.7: Výsledky evoluce násobičky 3×3.

Výsledky evoluce jsou zachyceny v tabulce 9.7. Příklad evolučně získané násobičky je na obrázku 9.6. U sady I. byl průměrný počet výsledků na Pareto frontě 1,85 a nejvyšší počet 5. U sady hradel II. byl průměrný počet výsledků na Pareto frontě 1,3 a nejvyšší počet 2. U sady hradel I. se povedlo nalézt lepší řešení, než je uvedeno v [5] a naopak u sady hradel II se nepovedlo nalézt tak kvalitní řešení, jako je uvedeno v [18].



Obrázek 9.6: Příklad evolučně navržené násobičky 3×3.

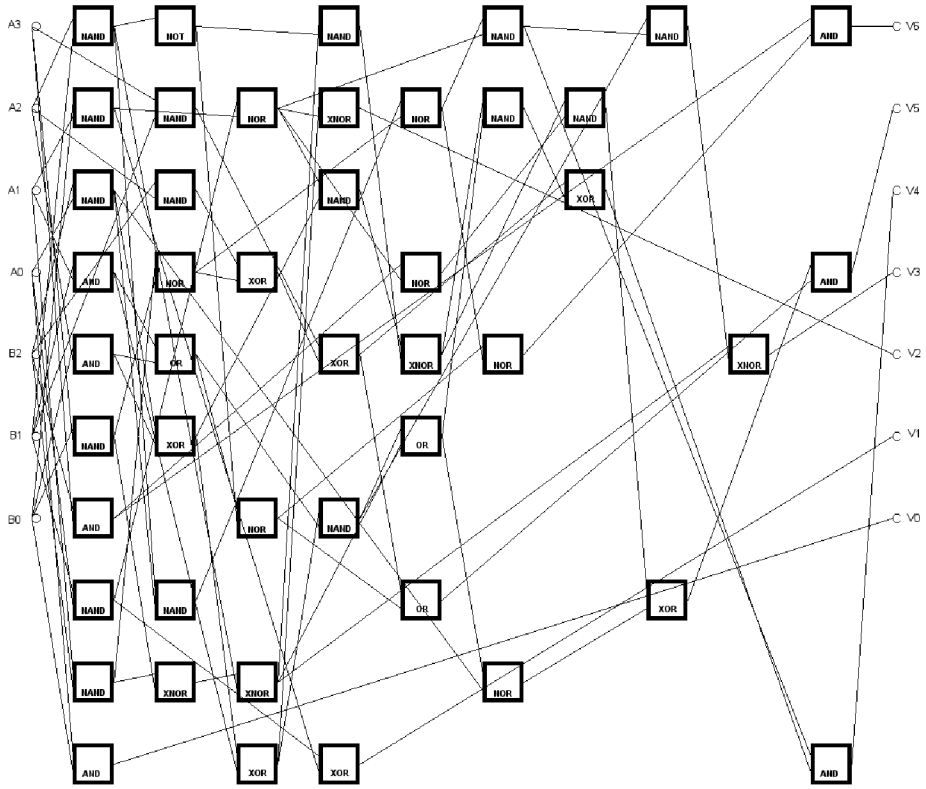
9.1.7 Vývoj násobičky 4×3

Násobička o jednom čtyřbitovém a jednom tříbitovém operandu byla vyvíjena s pomocí sady hradel I. uvedené v experimentu 9.1.4. byla optimalizována na počet hradel, počet tranzistorů a zpoždění. Evoluce probíhala na poli 10×10 a s parametrem $L = 9$. Velikost mutace byla nastavena na $0,5\%$. Bylo provedeno dvacet běhů evolučního algoritmu o $\mu = 20$, $\lambda = 2$ a 6.10^7 generacích.

Výsledky evoluce jsou zachyceny v tabulce 9.8. Příklad evolučně získané násobičky je na obrázku 9.7. Průměrný počet výsledků na Pareto frontě byl 1,68 a nejvyšší počet 4. Testovaná metoda sice dokázala překonat konvenční návrh a navrhnout obvod s nižším počtem hradel. Ten ale zdaleka nedosahuje kvality evolučně získaného řešení z [18]. Na vině mohl být nedostatečný počet generací.

Experiment	Průměrný výsledek			Nejlepší výsledek		
	Hradel	**Tranzistorů**	**Zpoždění**	**Hradel**	**Tranzistorů**	**Zpoždění**
Sada hradel I.	53,79	320,37	8,11	46	262	7
Konv. řešení dle [18]	-	-	-	47	-	-
Výsledek z [18]	-	-	-	37	-	10

Tabulka 9.8: Výsledky evoluce násobičky 4×3.



Obrázek 9.7: Příklad evolučně navržené násobičky 4×3.

9.2 Evoluce násobiček s vícenásobnými konstantními koeficienty

9.2.1 Vliv rychlosti mutace na kvalitu výsledných řešení

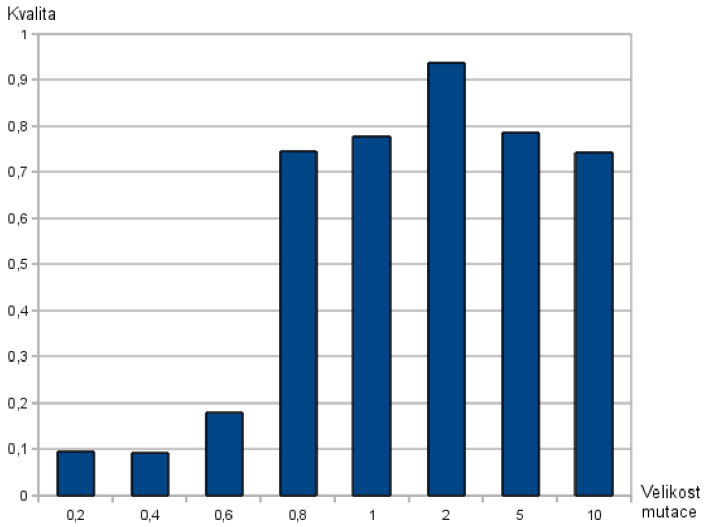
Pro otestování vlivu velikosti mutace na výsledná řešení byla zvolena úloha evolučního vývoje násobičky s třemi konstantními koeficienty 2925, 23111 a 13781. Velikost CGP pole byla zvolena 7×7 s parametrem $L = 6$. Evoluce byla prováděna pro mutace o velikostech $0,2\%$, $0,4\%$, $0,6\%$, $0,8\%$, 1%, 2%, 5% a 10%. Obvod byl optimalizován na počet jednotek, počet jednotek sčítání a odečítání a zpoždění. Soubory s výsledky jsou nahrány na přiloženém CD.

Pro každou velikost mutace bylo provedeno dvacet běhů programu. Pro hodnocení kvality jednotlivých nastavení byla požita stejná metoda jako při experimentu 9.1.1. Kvalita jednotlivých nastavení byla zakreslena do grafu 9.8. Jako nejvýhodnější se ukázala velikost

Velikost mutace	Počet jednotek	Počet sčítaček	Zpoždění	Kvalita
0,2	23,8 (0,99)	13,75 (0,892)	5,85 (0,833)	**0,094**
0,4	23,42 (0,899)	13,57 (0,829)	5,95 (1)	**0,091**
0,6	23,84 (1)	14,05 (1)	5,63 (0,467)	**0,178**
8	20,45 (0,183)	11,75 (0,179)	5,59 (0,4)	**0,746**
1	19,69 (0)	11,25 (0)	5,75 (0,667)	**0,778**
2	20,3 (0,147)	11,35 (0,036)	5,35 (0)	**0,939**
5	20,57 (0,212)	11,89 (0,229)	5,47 (0,2)	**0,786**
10	20,75 (0,255)	12,31 (0,379)	5,43 (0,133)	**0,744**

Tabulka 9.9: Vliv mutace na kvalitu výsledných řešení.

mutace 2%.



Obrázek 9.8: Kvalita výsledných řešení při evolučním návrhu násobičky se třemi konstantními koeficienty v závislosti na velikosti mutace.

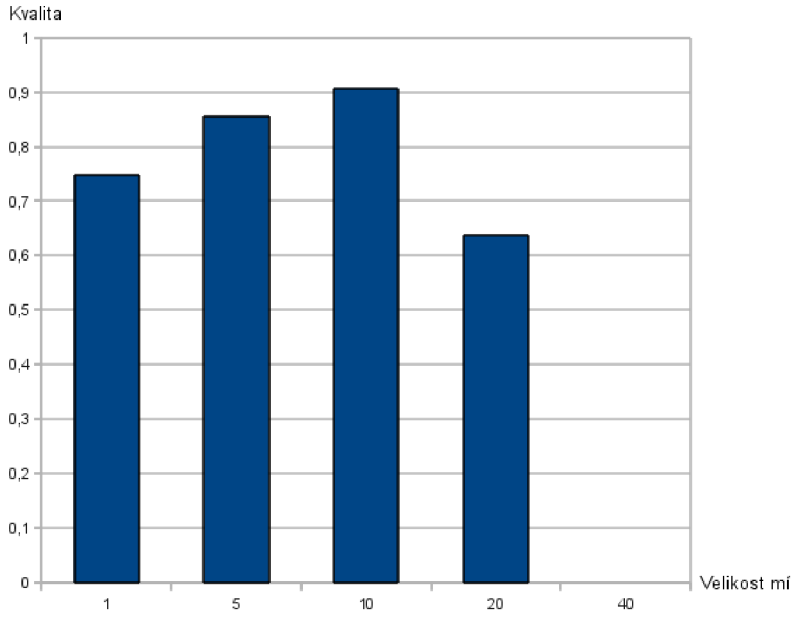
9.2.2 Vliv parametru μ na kvalitu výsledných řešení

K otestování vlivu velikosti parametru μ na kvalitu řešení byla použita stejná testovací úloha jako v sekci 9.2.1. Testovány byly hodnoty μ =1, 5, 10, 20, 40. Testy pro jednotlivé hodnoty byly spouštěny tak, aby počet ohodnocených kandidátních řešení byl vždy stejný. Při evoluci byla použita 2% velikost mutace, která se ukázala jako nejvýhodnější v experimentu 9.2.1. Parametr λ byl nastaven na hodnotu 4.

Výsledky jsou uvedeny v tabulce 9.10. a soubory s evolučně navrženými obvody jsou přiloženy na CD.

μ	Generací	Počet jednotek	Počet sčítaček	Zpoždění	Kvalita
1	4.10^8	18,8 (0)	10,8 (0)	5,6 (0,758)	**0,747**
5	8.10^7	18,9 (0,043)	11,1 (0,24)	5,4 (0,152)	**0,855**
10	4.10^7	19,35 (0,238)	10,85 (0,04)	5,35 (0)	**0,907**
20	2.10^7	20,3 (0,649)	11,35 (0,44)	5,35 (0)	**0,637**
40	1.10^7	21,11 (1)	12,05 (1)	5,68 (1)	**0**

Tabulka 9.10: Vliv mutace na kvalitu výsledných řešení. V každém běhu bylo ohodnoceno $1,6.10^9$ jedinců.



Obrázek 9.9: Kvalita výsledných řešení při evolučním návrhu násobičky se třemi konstantními koeficienty v závislosti na velikosti μ.

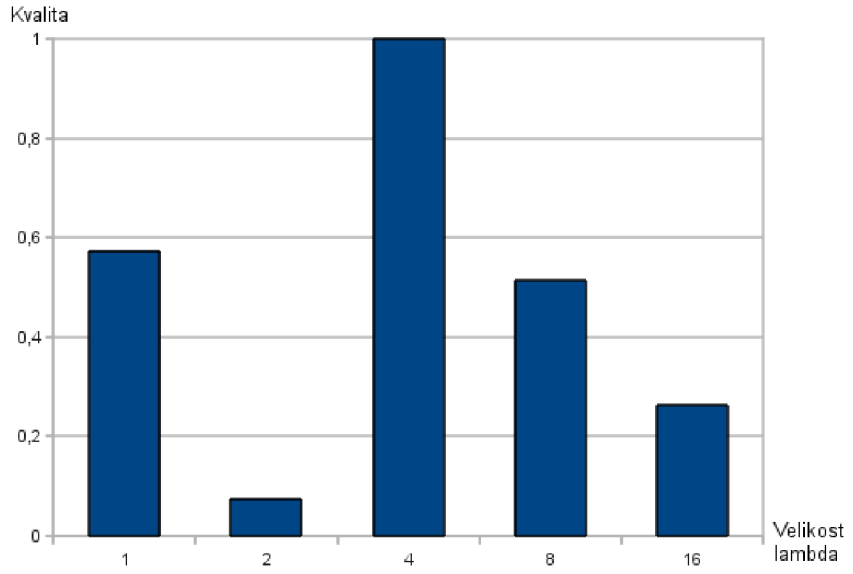
9.2.3 Vliv parametru λ na kvalitu výsledných řešení

Vliv velikosti parametru λ na kvalitu výsledný řešení byl testován na úloze z 9.2.1. Byly testovány hodnoty λ=1, 2, 4, 8, 16. Při experimentech byla použita mutace 2% a parametr $\mu = 10$. Tyto hodnoty parametrů byly zjištěny jako nejvhodnější v experimentech z 9.2.1. a 9.2.2. Počet generací pro jednotlivá nastavení byl volen tak, aby byl ohodnocen stejný počet kandidátních řešení.

Výsledky jsou shrnuty v tabulce 9.11. a soubory s evolučně navrženými obvody jsou na přiloženém CD. Pro názornost byl vytvořen graf 9.10. Z výsledků vyplývá, že nejvhodnější je volit $\lambda = 4$.

λ	Generací	Počet jednotek	Počet sčítaček	Zpoždění	Kvalita
1	8.10^7	19,35 (0)	11,05 (0,444)	5,6 (0,833)	**0,574**
2	4.10^7	20,15 (1)	11,2 (0,778)	5,65 (1)	**0,074**
4	2.10^7	19,35 (0)	10,85 (0)	5,35 (0)	**1**
8	1.10^7	19,84 (0,613)	11,05 (0,444)	5,47 (0,4)	**0,514**
16	5.10^6	20,05 (0,875)	11,3 (1)	5,45 (0,333)	**0,264**

Tabulka 9.11: Vliv velikosti parametru λ na kvalitu výsledných řešení. V každém běhu bylo ohodnoceno $1,6.10^9$ jedinců.



Obrázek 9.10: Kvalita výsledných řešení při evolučním návrhu násobičky se třemi konstantními koeficienty v závislosti na velikosti λ.

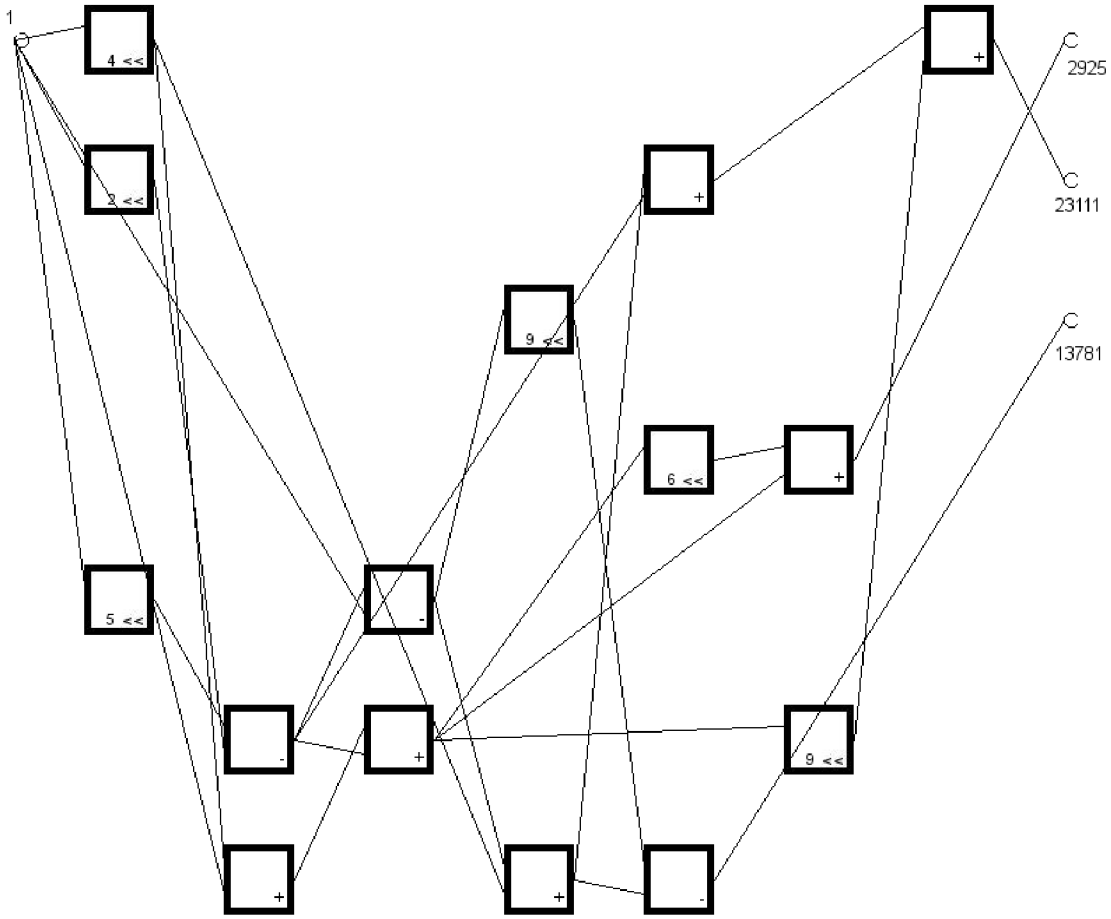
9.2.4 Vývoj vybraných typů MCM násobiček

Při testování byly vyvíjeny násobičky s vícenásobnými konstantními koeficienty o 3, 5, 10 a 20 koeficientech. Pro každý typ násobičky bylo provedeno 20 běhů algoritmu s parametry $\mu = 10$, $\lambda = 4$ a bylo vždy vytvořeno 40 000 000 generací.

Výsledky testování jsou uvedeny v tabulce 9.12. a výsledné obvody jsou obsaženy na přiloženém CD. Metodou byla vytvářena lepší průměrné výsledky obvodů oproti [20]. Oproti metodě [21] pak poskytuje lepší zpoždění výsledných obvodů.

Nastavení		Průměrný výsledek			Nejlepší výsledek		
sloupců x řádků	maximálně generací	zpoždění	uzly +-	uzly	zpoždění	uzly +-	uzly
3 konstanty: 2925, 23111, 13781							
Heuristiky [21]					8	8	16
5x6 dle [20]	20 M	-	14	-	5	9	17
6x6 dle [20]	20 M	-	14	-	6	8	16
7x4 dle [20]	40 M	-	13	-	7	8	14
7x7	**40 M**	**5,45**	**11,15**	**19,45**	**5**	**9**	**15**
5 konstant: 83, 221, 71, 387, 13							
Heuristiky [21]					5	6	12
4x6 dle [20]	20 M	-	10	-	4	7	13
5x6 dle [20]	20 M	-	11	-	5	6	12
6x6 dle [20]	20 M	-	11	-	6	6	11
6x6	**40 M**	**4**	**7,2**	**12,95**	**4**	**6**	**11**
10 konstant: 117, 1123, 743, 221, 1069, 7605, 987, 16689, 3033, 29							
Heuristiky [21]					8	14	27
10x4 dle [20]	40 M	-	23	-	7	15	27
7x6 dle [20]	20 M	-	23	-	6	17	28
9x4 dle [20]	40 M	-	22	-	9	17	26
10x6	**40 M**	**5,25**	**19,75**	**31,5**	**5**	**17**	**29**
20 konstant: 1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71							
Heuristiky [21]					4	19	27
4x10 dle [20]	40 M	-	23	-	4	19	23
5x10 dle [20]	40 M	-	23	-	4	19	23
6x5 dle [20]	40 M	-	21	-	5	19	22
6x10	**40 M**	**3,95**	**20,4**	**24,7**	**3**	**20**	**24**

Tabulka 9.12: Výsledky testů generování MCM násobiček ve srovnání s jinými metodami. Výsledky metody, která byla představena v této práci, jsou označeny tučně.



Obrázek 9.11: Příklad evolučně navržené násobičky s třemi konstantními koeficienty (2925, 23111, 13781).

Kapitola 10

Závěr

V teoretické části práce byly popsány základy evolučních algoritmů, evolučních strategií, multikriteriální optimalizace, principy klasického genetického programování a kartézského genetického programování.

Hlavním cílem práce bylo začlenit multikriteriální optimalizaci do kartézského genetického programování. Použitá metoda je podrobně popsána v kapitole 7. Tato metoda byla implementována a otestována na návrhu kombinačních číslicových obvodů. Byly provedeny testy jak pro evoluci na úrovni hradel při návrhu binárních kombinačních násobiček, tak na úrovni funkčních bloků při návrhu násobiček s vícenásobnými konstantními koeficienty.

Při testování byly nalezeny vhodné parametry evoluce pro obě úlohy a poté byly vyvinuty obvody pro srovnání s jinými typy metod návrhu. U binárních kombinačních násobiček bylo dosaženo podobných výsledků, které dosahovaly ostatní metody. Výsledky byly horší u násobičky s jedním čtyřbitovým a jedním tříbitovým operandem. Na vině je pravděpodobně nízký počet generací.

U násobiček s vícenásobnými konstantními koeficienty se podařilo dosáhnout lepších výsledků především u zpoždění obvodu. Systém je naprogramován s využitím objektového přístupu. To umožňuje snadné přidávání dalších testovacích problémů.

Literatura

[1] Deb K., Pratap A., Agarwal A.,Meyarivan T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transtactions on Evolutionary Computation. vol. 6, no. 2, pp. 182-197, 2002.

[2] Drábek V.: Výstavba počítačů. skriptum VUT Brno, 1995.

[3] Harms D., McDonald K.: *Začínáme programovat v jazyce Python.* Computer Press, 2008.

[4] Hilder J., A., Walker J., A., Tyrrell A., M.: Optimising Variability Tolerant Standard Cell Libraries. Evolutionary Computation, 2009. CEC '09. IEEE Congress on, pp. 2273-2780, 2009.

[5] Hilder J., Walker J. A., Tyrrell A.: Use of Multi-Objective Fitness Function to Improve Cartesian Genetic Programming Circuits. NASA/ESA Conference on Adaptive Hardware and Systems, pp. 179-185, 2010.

[6] Hornby G., Globus A., Linden D.: Automated Antenna Design with Evolutionary Algorithms. In Proc. 2006 AIAA Space Conference, Sun Jose, pp. 8, 2006.

[7] Koza J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, 1992.

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

[9] Langdon V. B.: *Genetic Programming and Data Structures.* Kluwer Academic Publishers, 2000.

[10] Liberty J.: *Naučte se C++ za 21 dní.* Computer Press, 2007.

[11] Lipson H., Pollack J.: Automatic design and manufacture of robotic lifeforms. Nature vol. 406, pp. 974-978, 2000.

[12] Mařík V, kol.: *Umělá inteligence (3).* ACADEMIA Praha, 2001.

[13] Mařík V, kol.: *Umělá inteligence (4).* ACADEMIA Praha, 2003.

[14] Miller J. F., Thomson P.: Cartesian Genetic Programming. In Proceedings of the Third European Conference on Genetic Programming (EuroGP2000). LNCS 1802, Springer Verlag, pp. 121-132, 2000.

[15] Schwarz J., Sekanina L.: Aplikované evoluční algoritmy - studijní opora. FIT VUT v Brně, 2006.

[16] Sekanina L., Vašíček Z., Růžička R. , Bidlo M., Jaroš J., Švenda P.: *Evoluční hardware - Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům.* ACADEMIA, 2009.

[17] Sekanina L., Walker J., Kaufmann P., Platzner M.: Cartesian Genetic Programming (ed. J. F. Miller). Springer Verlag, 2011, 58 s. (v tisku), 2011.

[18] Vassilev V. K., Job D., Miller J. F.: Towards the Automatic Design of More Efficient Digital Circuits. Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on, pp. 151-160, 2000.

[19] Vašíček Z., Sekanina L.: Evoluční návrh kombinačních obvodů. Elektrorevue, č. 43, s. 1-6, 2004.

[20] Vašíček Z., Žádník M., Sekanina L., Tobola J.: On Evolutionary Synthesis of Linear Transforms in FPGA. In Proc. of the Conference on Evolvable Systems: From Biology to Hardware, LNCS 5216, Springer Verlag, pp. 141-152, 2008.

[21] Voronenko Y., Püschel M.: Multipliers Multiple Constant Multiplication. ACM Transactions on Algorithms, vol. 3, no. 2, pp. 1-28, 2007.

[22] Walker J. A., Hilder J A., Tyrrell A. M.: Towards Evolving Industry-feasible Intrinsic Variability Tolerant CMOS Designs. IEEE Congress on Evolutionary Computation, pp. 1591-1598, 2009.

[23] Wang J., Lee C. H.: Evolutionary design of combinational logic circuits using VRA processor. IEICE Electronics Express, vol. 6, no. 3, pp. 141-147, 2009.

[24] www stránky: Qt Reference Documentation. [cit. 2011-04-05].
URL http://doc.qt.nokia.com/4.6/

[25] www stránky: Spiral project. [cit. 2011-04-05].
URL http://www.spiral.net

Dodatek A

Seznam zkratek

- **CGP** - Cartesian Genetic Programming
- **ES** - evoluční strategie
- **GA** - genetický algoritmus
- **GP** - genetické programování
- **NSGAII** - Fast and Elitist Multiobjective Genetic Algorithm

Dodatek B

Seznam příloh

1. Přiložené CD
    - Program pro evoluční vývoj digitálních obvodů.
    - Prohlížeč vyvinutých obvodů.
    - Uživatelská příručka.