VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODULARITA V EVOLUČNÍM NÁVRHU

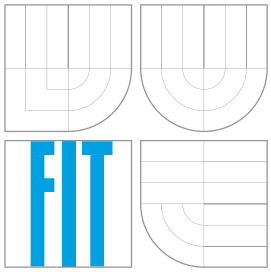
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JARMILA KLEMŠOVÁ

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

MODULARITA V EVOLUČNÍM NÁVRHU

MODULARITY IN THE EVOLUTIONARY DESIGN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JARMILA KLEMŠOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK VAŠÍČEK

BRNO 2011

Abstrakt

Tato práce popisuje vybrané evoluční algoritmy a jejich využití hlavně v oblasti návrhu číslicových obvodů. V první části se zabývá obecným principem evolučních algoritmů. Na tuto část navazuje genetickými algoritmy a genetickým programováním. Dále se zabývá popisem kartézského genetického programování a některými jeho modifikacemi jako modulární, sebemodifikující se a kartézské genetické programování s více chromozomy. Hlavní část tvoří návrh a implementace modularizační techniky pro zefektivnění evolučního návrhu. Nedílnou součástí je experimentální vyhodnocení systému na sadě benchmarkových obvodů.

Abstract

The diploma thesis deals with the evolutionary algorithms and their application in the area of digital circuit design. In the first part, general principles of evolutionary algorithms are introduced. This part includes also the introduction of genetic algorithms and genetic programming. The next chapter describes the cartesian genetic programming and its modifications like embedded, self-modifying or multi-chromosome cartessian genetic programming. Essential part of this work consists of the design and implementation of a modularization technique for evolution circuit design. The proposed approach is evaluated using a set of standard benchmark circuits.

Klíčová slova

evoluční algoritmy, kartézské genetické programování, evoluční návrh

Keywords

evolutionary algorithms, cartesian genetic programming, evolutionary design

Citace

Jarmila Klemšová: Modularita v evolučním návrhu, diplomová práce, Brno, FIT VUT v Brně, 2011

Modularita v evolučním návrhu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Ing. Zdeňka Vašíčka. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Jarmila Klemšová
22. května 2011

Poděkování

Děkuji svému vedoucímu panu Ing. Zdeňku Vašíčkovi za pomoc a rady při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Evoluční algoritmy (EA) začaly v poslední době pronikat do mnoha oblastí. Kromě optimalizačních úloh, kdy hledáme vyhovující hodnoty parametrů, se začaly uplatňovat i v oblasti návrhu. Jednou z typických úloh je návrh číslicových obvodů. Cílem EA je nalézt zapojení obvodu (tj. rozmístění a propojení komponent) tak, aby výsledný obvod splňoval specifikaci. Zatím co u konvenčního přístupu návrhář dodržuje jistá pravidla a ověřené postupy, evoluční návrh převádí úlohu na prohledávání prostoru možných řešení. Výhodou tohoto nekonvenčního přístupu je například možnost nalezení inovativního řešení, kterého bychom běžnými metodami nedosáhli. Pro návrh a optimalizaci kombinačních obvodů se jako vhodný evoluční algoritmus ukázalo kartézské genetické programování (CGP). Jinou úlohou evolučních algoritmů může být zajištění adaptace zařízení na nové okolní prostředí nebo obnova funkce poškozeného obvodu [12].

Přes mnohé výhody se evoluční algoritmy potýkají i s řadou nevýhod, zejména s problémem škálovatelnosti. Rozlišujeme problém škálovatelnosti reprezentace (kdy vznikají příliš rozsáhlé prostory, které lze jen těžko efektivně prohledat) a evaluace. Nízká škálovatelnost evaluace je velkým problémem kombinačních obvodů, protože počet evaluací roste exponenciální řadou. V práci se budeme zabývat problémem škálovatelnosti reprezentace u kartézského genetického programování. V poslední době byla navržena řada technik (modulární či sebemodifikující CGP), které se snaží tuto otázku řešit. Cílem práce je navrhnout a implementovat modularizační techniku CGP pro zefektivnění evolučního návrhu.

Text práce je členěn následovně. První část seznamuje čtenáře obecně s evolučními algoritmy a jejich základní strukturou. Dále je zde popsán, dnes již klasický zástupce těchto algoritmů, tzv. genetický algoritmus (GA). Ke konci se pak zabývám genetickým programováním (GP). Následující část popisuje kartézské genetické programování, jeho základní vlastnosti a použití. Kapitola 4 se zabývá třemi modifikacemi, které pracují se strukturou reprezentace. Jedná se o modulární CGP, sebemodifikující se CGP a CGP s více chromozomy. V následující části je popsán návrh modularizační techniky. Čtenáře seznamuje zejména se způsobem tvorby modulů a během modifikovaného evolučního algoritmu. Další kapitola se zabývá imlementací navrženého systému. Především pak reprezentací a nejdůležitějšími funkcemi pro práci s moduly. V předposlední části jsou uvedeny výsledky experimentálního ověření funkčnosti systému na vybraných kombinačních obvodech. Jedná se o sudou paritu, řadicí síť, sčítačku a násobičku. V závěrečné části (kapitola 8) jsou shrnuty výsledky práce a nastíněno možné pokračování práce.

Kapitola 2

Evoluční algoritmy

Evoluční algoritmy jsou stochastické prohledávací algoritmy, které se postupně vyvíjeli nezávisle na sobě asi od 60. let 20. století [6]. Jejich původní využití bylo především v optimalizaci. Dnes mají své místo i v oblastech návrhu [12].

EA se inspirují biologickou evolucí a procesy probíhajícími v živé přírodě [11]. Biologická evoluce má tři základní složky. Je to přirozený výběr, který zaručuje větší pravděpodobnost reprodukce pro jedince, kteří jsou lépe adaptovaní na prostředí. Dále náhodný genetický drift, který ovlivňuje populaci náhodnými událostmi jako mutace genetického materiálu či úmrtí vysoce adaptovaného jedince, ještě před jeho účastí v reprodukčním procesu. A na konec samotný reprodukční proces, kde genetická informace potomků je tvořena vzájemnou výměnou genetické informace rodičů.

2.1 Základní princip

Na počátku EA je vytvořena počáteční populace. Jedná se o předem zvolený počet kandidátních řešení. Její tvorba je buď náhodná, nebo se řídí vhodnou heuristikou, která zanáší do řešení jistou znalost o problému. Následuje ohodnocení jedinců populace, neboli výpočet tzv. fitness funkce. Její hodnota vyjadřuje kvalitu jedince. Jednotlivé kroky algoritmu nazýváme generace. V každé generaci dochází ke vzniku nové populace. Nejprve jsou vybráni z populace jedinci, ze kterých se bude tvořit populace potomků pomocí tzv. rekombinačních operátorů. Sem obvykle řadíme mutaci a křížení. Mutací rozumíme modifikaci genu určitého jedince. Křížení znamená zkombinování dvou či více jedinců. Takto vytvoření potomci jsou opět ohodnoceni. Z potomků a původní populace je vhodným způsobem vytvořena populace nová. Obecně lze říci, že na výběr má vliv fitness funkce. Jedinci s lepší hodnotou mají větší pravděpodobnost dostat se do nové populace. Jednotlivé kroky algoritmu se opakují tak dlouho, dokud není splněna ukončující podmínka. Tou může být nalezení uspokojivého řešení, nebo dovršení zadaného počtu generací.

Evoluční algoritmy se liší především v zakódování problému, realizace fitness funkce, rekombinačních operátorech, technice výběru rodičovské populace či tvorbě populace nové, což se odráží na efektivitě EA, i na době trvání evoluce. Inspirace v biologické evoluci však zavedla některé společné termíny. Zakódování jedince nazýváme chromozom nebo též genotyp. Často se jedná o binární či celočíselný řetězec. V genetickém programování se však setkáváme se složitějšími strukturami. Fenotypem pak rozumíme konečné řešení, v oblasti návrhu tedy například obvod.

Obecně lze evoluční algoritmus popsat pomocí následujícího pseudokódu:

```
begin
    t := 0;
    G(t) := inicializace();
    ohodnocení(G(t));
    while (not ukončující_podmínka) do
    begin
        S(t) := selekce_rodičů(G(t));
        P(t) := rekombinace(S(t));
        ohodnocení(P(t));
        G(t+1) := tvorba_nové_populace(G(t), P(t));
        t := t+1;
    end
end
```

2.2 Genetické algoritmy

Genetické algoritmy se staly velmi populární na konci 80. let 20. století a dnes představují klasického zástupce evolučních algoritmů [6].

Jejich základní varianta má strukturu, která byla uvedena v části 2.1. Chromozom je řetězec fixní délky, nejčasteji v binárním kódování. Počáteční populace je vygenerována náhodně. Z rodičovské generace jsou selekčními mechanismy vybráni jedinci, na které jsou s určitou pravděpodobností aplikovány rekombinační operátory. Novou populaci lze tvořit několika způsoby. O generační variantě mluvíme tehdy, pokud novou populaci tvoří pouze nově vzniklí jedinci. Při překrývání populací jsou v nové populaci zařazeni jak rodiče tak potomci. Navíc se můžeme setkat s tzv. elitismem, kdy se do populace nové dostává daný počet nejlepších jedinců z populace předešlé. Ohodnocování jedince, nebo-li fitness funkce, je závislé na řešené úloze.

2.2.1 Selekční mechanismy

Selekce je významnou součástí genetických algoritmů. Na jedné straně musí dostatečně upřednostňovat jedince s vyšší fitness, na straně druhé musí zachovávat dostatečnou různorodost populace. Mezi používané selekce patří ruleta a turnaj.

Proporciální selekce (též ruleta) je založena na náhodném výběru, avšak každý rodič má různou šanci, která je dána jeho fitness. Pravděpodobnost výběru i-tého jedince je dána vztahem:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{2.1}$$

kde f je hodnota fitness a N je počet jedinců.

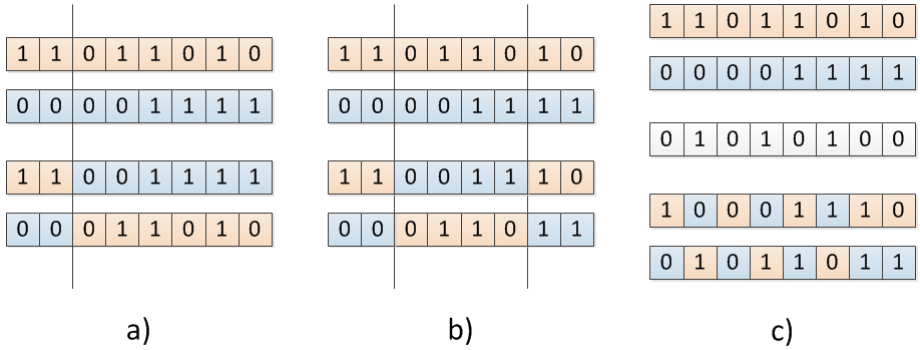
Turnajová selekce náhodně vybere dva jedince, z nichž je vybrán ten s vyšší fitness. Proces se opakuje tak dlouho, dokud neni vybrán požadovaný počet jedinců. Metoda podává velmi dobré výsledky a v reálných aplikacích se využívá téměř výhradně.

2.2.2 Rekombinační operátory

Genetické algoritmy používají pro rekombinaci mutaci a křížení. Obě operace jsou aplikovány s určitou pravděpodobností.

Mutace vytváří z vybraného jedince jednoho mutovaného jedince. Při mutaci dochází ke změně jednoho náhodně vybraného genu (např. bitu) chromozomu. Konkrétní realizace záleží na použitém zakódování. V případě binárního kódování se jedná o negaci. Pokud je reprezentace celočíselná, každý gen může nabývat hodnot z určitého intervalu. V tomto případě je vhodné zajistit, aby docházelo pouze k platným mutacím. Pravděpodobnost mutace se volí obvykle poměrně malá mezi 5 - 15 %.

Křížení je považováno za základní princip, který zajišťuje fungování genetického algoritmu. Jeho pravděpodobnost bývá v rozsahu 60 - 100 %. Během křížení dochází ke kombinaci dvou rodičovských jedinců. U jednobodového je náhodně vygenerován bod. Potomci vznikají výměnou genů od vygenerovaného bodu až po konec chromozomu. U vícebodového je generováno více bodů a genetický materiál je tak více promíchán. Posledním typem křížení je uniformní. Zde se vygeneruje maska, podle které jsou vybírány jednotlivé geny (tzn. pokud je například hodnota v masce 1, potomek získá gen z prvního rodiče, pokud 0 tak z druhého, jak ukazuje obrázek 2.1 c). Uvedené typy křížení jsou zobrazeny na následujícím obrázku 2.1.



Obrázek 2.1: Křížení - a) jednobodové, b) vícebodové, c) uniformní. Nahoře jsou vždy dva rodiče (barevně odlišeni), dole dva potomci. Svislé čáry značí body křížení. Maska pro uniformní křížení je uvedena pod rodičovskými chromozomy (bílá barva).

2.3 Genetické programování

Genetické programování je jednou z nejmladších evolučních technik [7]. Vychází z genetických algoritmů, od kterých se liší v několika základních bodech.

Zatím co genetické algoritmy se využívají převážně k optimalizaci (hledání hodnot nezávisle proměnných), genetické programování je určeno pro automatickou syntézu programů, algoritmů či rozhodovacích postupů. Další rozdíl spočívá ve způsobu ohodnocování jedinců, nebo-li fitness funkci. U GP je nutné kandidátní řešení simulovat (tzn. vykonat algoritmus, který je v kandidátním řešení zakódován) [12].

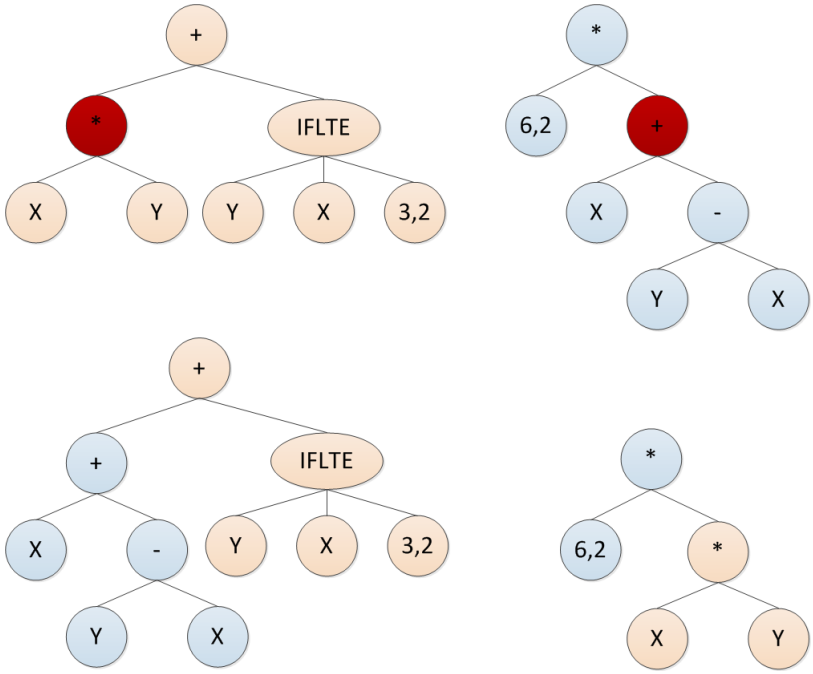
2.3.1 Reprezentace

Další odlišnost od genetických algoritmů spočívá ve způsobu reprezentace kandidátních řešení. Genetické programování pracuje se strukturami, které jsou reprezentovány stromy. Stromy jsou proměnné délky a skládají se z terminálů a nonterminálů. Terminály (tzn.

listové uzly) jsou vstupy programu, konstanty či funkce bez argumentů. Nonterminály (nelistové uzly) pak představují operace či funkce.

2.3.2 Rekombinační operátory

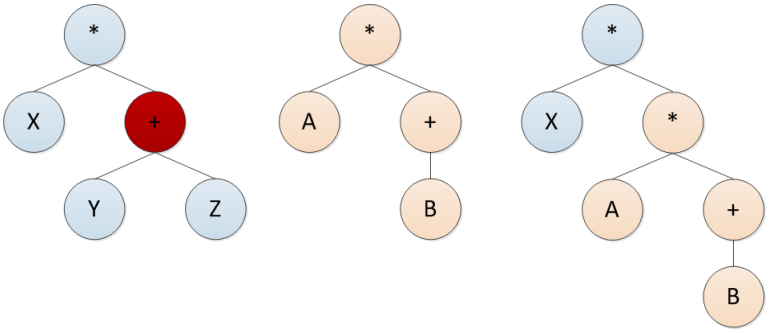
Stejně jako v genetických algoritmech i v GP hraje křížení důležitou roli. Křížení slouží k vytváření nového genetického materiálu. Do operace vstupují dva jedinci a výsledkem je jeden či dva potomci. Obvykle se postupuje tak, že v rodičovských chromozomech se náhodně vybere po jednom uzlu. Podstromy vycházející z těchto uzlů se prohodí. Aby nedocházelo k prohazování pouze listových uzlů, vnitřní uzly bývají typicky vybírány s větší pravděpodobností než listy [7]. Typická realizace křížení stromových struktur je zobrazena na následujícím obrázku 2.2.



Obrázek 2.2: Křížení stromových struktur. Nahoře rodiče, dole potomci. Vybrané uzly pro křížení označeny červeně.

Operátor mutace pracuje pouze s jedním jedincem. Nejprve se náhodně vybere uzel. Vybraný uzel, včetně jeho podstromu, je nahrazen novým podstromem, který je vygenerován stejně jako stromy počáteční populace. Typická realizace mutace stromovýsch struktur je zobrazena na obrázku 2.3.

Kromě zmíněných základních operátorů se někdy v genetickém programování používají také následující operátory: permutace (modifikuje pořadí argumentů náhodně zvoleného vnitřního uzlu), dále například editace (zjednodušování vyvýjených stromů) či zapouzdření, které bylo původně navrženo pro automatické definování stavebních bloků. Zapouzdření probíhá tak, že je náhodně vybrán vnitřní uzel a jeho podstrom je prohlášen za novou dále nedělitelnou funkci, která je přidána do množiny funkcí.



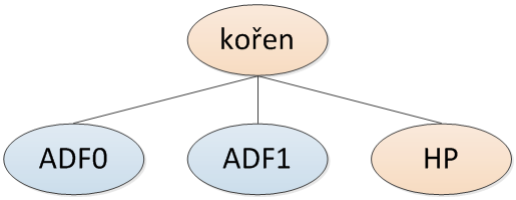
Obrázek 2.3: Mutace stromových struktur. Vlevo jedinec s červeně vyznačeným bodem mutace. Uprostřed náhodně vygenerovaný podstrom. Vpravo zmutovaný jedinec.

2.3.3 Automaticky definované funkce

Základní varianta genetického programování, která byla popsána výše, se potýká s problémem neefektivnosti generovaného kódu (kód nelze sdílet) a nízkou škálovatelností (funguje pro relativně jednoduché problémy). Řešením bylo zavedení modularizační techniky, tzv. automaticky definovaných funkcí (ADF) [5].

Program používající ADF je reprezentován několika podstromy, které se vyvíjejí současně. Pod hlavním kořenem jsou nejprve podstomy pro jednotlivé AFD a nakonec až větev reprezentující hlavní program, jak je vidět na obrázku 2.4. Jakákoliv automaticky definovaná funkce může být i několikanásobně použita v hlavním programu jako vnitřní uzel. Rozšířením konceptu ADF jsou hierarchické automaticky definované funkce, kde je povoleno přímo v těle ADF použít kromě povolených funkcí i některou z dalších ADF.

Nevýhodou koncepce automaticky definovaných funkcí je, že před započetím evoluce musí být známa architektura celého programu, což znamená, že je nutno určit kolik bude větví s ADF a počet argumentů každé větve. Možnost řešení tohoto omezení je nastíněna v [4], kde jsou zavedeny operace upravující architekturu, které řeší inicializaci či změnu počtu parametrů ADF.



Obrázek 2.4: Schéma stromu s automaticky definovanými funkcemi.

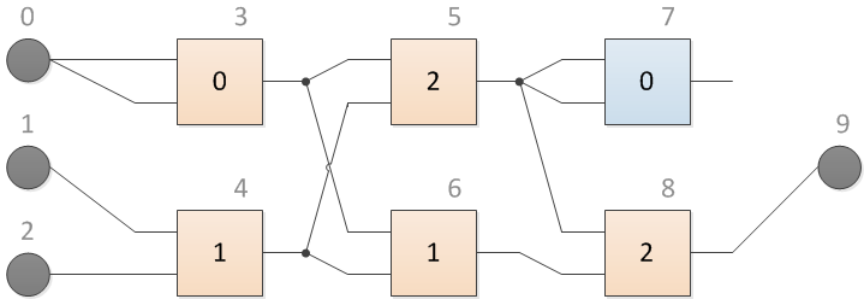
Kapitola 3

Kartézské genetické programování

Kartézské genetické programovaní bylo poprvé představeno v roce 1999 (viz [10]) jako vhodný prostředek pro evoluční návrh číslicových obvodů. Tato kapitola se zabývá popisem algoritmu CGP, používanou reprezentací, rekombinačními operátory, způsobem výpočtu fitness a typickými aplikacemi. Další informace o CGP je možné nalézt v [8], [9], [12] a [13].

3.1 Reprezentace

V této variantě genetického programování jsou kandidátní řešení reprezentována jako acyklické orientované grafy [12]. Jednotlivé elementy, neboli uzly grafu, jsou uspořádány do dvourozměrné mřížky o velikosti n_r (počet řádků) x n_c (počet sloupců). Počet těchto elementů je v základní verzi CGP (tak jak bylo definováno v roce 1999) neměnný. Dále zde máme pevný počet vstupů celého obvodu n_i a počet výstupů n_o. Každý uzel realizuje právě jednu funkci f z množiny možných funkcí F a má n_n vstupů a jeden výstup. Uzly mohou být připojeny buď přímo na primární vstupy, nebo na některý z předchozích uzlů. Připojení na uzly se řídí jistými pravidly. Žádný uzel nesmí být připojen na uzel ve stejném sloupci, pouze na uzly v sloupcích předchozích. Počet předchozích sloupců, na které lze připojovat, je dán $l-back$ parametrem. Je-li hodnota $l-back$ parametru rovna 1, uzly v k-tém sloupci se mohu připojit buď na primární vstupy obvodu nebo na uzly ve sloupci k-1.



Obrázek 3.1: CGP - příklad kandidátního obvodu. Uzel č. 7 není ve fenotypu využit.

Chromozom kódující zapojení obvodu v CGP se sestává z N celočíselných hodnot. N lze vypočítat následovně:

$$N = n_o + n_c * n_r * (n_n + 1) \tag{3.1}$$

kde n_o je počet výstupů obvodu, n_c je počet sloupců, n_r počet řádků a n_n počet vstupů funkčních bloků. Jednička se připočítává, protože každému elementu musíme přiřadit jeho funkci. Uveďme příklad chromozomu pro výše uvedený obrázek:

$$(000), (121), (342), (341), (550), (562), (8) \tag{3.2}$$

Každá závorka kóduje jeden uzel, poslední kóduje výstup, přesněji obsahuje číslo elementu, na který je výstup připojen. Každá závorka kódující konfiguraci odpovídajícího uzlu obsahuje celkem tři čísla. První reprezentuje číslo elementu, na který je připojen první vstup, druhý číslo elementu, na který je připojen druhý vstup. Třetí číslo je kód funkce elementu.

Z obrázku je patrné, že některé uzly (konkrétně uzel číslo 7) nijak neovlivňují celkový výstup, neboli fenotyp. Takovýmto uzlům se říká neaktivní. Z toho je patrné, že zatímco chromozom (též genotyp, čili soubor genů) je pevné délky, velikost fenotypu je proměnná a závisí na množství aktivních uzlů.

3.2 Použité operátory a prohledávací algoritmus

CGP narozdíl od klasického genetického programování používá pouze operátor mutace, který je realizován následovně. Z chromozomu je náhodně vybráno obecně n genů, jejichž hodnota je náhodně změněna na jinou. V každém z vybraných uzlů pak dojde ke změně připojení jednoho ze vstupů nebo změně funkce uzlu. Při změně připojení musíme dbát na to, aby nová hodnota byla v rámci přípustných hodnot.

Mutace můžeme rozlišit na neutrální a adaptivní. V případě, že mutace nemá vliv na hodnotu fitness funkce, mluvíme o mutaci neutrální. K té může dojít tím, že sice dojde ke změně fenotypu, nicméně po ohodnocení má stejnou fitness. Další možností je, že se fenotyp nezmění, protože operátor byl aplikován na neaktivní uzly. Pokud není mutace neutrální, říkáme o ní, že je adaptivní. Jedna jediná adaptivní mutace může vést k velkým změnám fenotypu, neboť může zaktivovat (nebo naopak deaktivovat) několik doposud neaktivních uzlů.

Prohledávací algoritmus odpovídá evoluční strategii $(1 + \lambda)$. To znamená, že z populace je vybrán jeden jedinec s nejlepší fitness hodnotou (rodič) a společně se svými λ mutanty (potomci) vytvoří populaci novou. V CGP bývají populace méně početné a λ se pohybuje okolo čísla 4. Je důležité říci, že v další generaci nemůže být vybrán za rodiče ten samý jedinec, který byl vybrán jako rodič v generaci předešlé.

3.3 Fitness funkce

Fitness funkce je funkce, která slouží k vyhodnocení kvality kandidátního řešení. V případě evolučního návrhu a optimalizace kombinačních obvodů je zapotřebí pravdivostní tabulka. Kandidátní řešení otestujeme pro každou možnou kombinaci vstupů a výsledek porovnáme s požadovaným výstupem. Výsledná fitness je pak rovna počtu správných bitů ve výstupním vektoru, tzn. bitů shodujících se s požadovaným výstupem.

Při naivní simulaci (postupně pro každý vstup) je nutno každý obvod projít 2^n-krát, kde n je počet vstupů. Doba simulace roste exponenciálně s rostoucím počtem vstupů, což je pro větší počet vstupů neúnosné. Proto se v implementacích běžně používá tzv. paralelní simulace. Ta využívá bitových logických operátorů, které jsou schopny provést jednu logickou operaci současně nad všemi bity operandů v jednom taktu. Při vhodném zakódování pak stačí $2^n/b$ průchodů obvodem, kde n je počet vstupů a b je počet bitů datového typu pro vstupní vektory.

3.4 Aplikace

Kartézské genetické programování se používá především pro návrh malých kombinačních číslicových obvodů, jako jsou sčítačky, násobičky či obvody řešící paritu. Dále byl návrh také úspěšný u řadicích a mediánových sítí [12]. Touto metodou lze docílit vylepšení oproti konvenčnímu návrhu dle různých kritérií, jako cena nebo zpoždění.

Standardní podobu CGP, jak byla popsána výše, lze použít pro obvody do přibližně dvaceti vstupů a do stovky hradel. S každým novým vstupem se zdvojnásobí počet vektorů, které je nutno ohodnotit fitness funkcí, čímž výrazně roste časová náročnost výpočtu. Tento problém je typický pro návrh kombinačních obvodů a nazýváme jej problém škálovatelnosti evaluace. Dalším problémem, se kterým se u CGP setkáváme, je nízká škálovatelnost reprezentace. S rostoucí složitostí obvodu roste i délka chromozomu. To však vede k velkým prostorům, které je obtížné prohledávat. Tento problém je markantní např. u evolučního návrhu násobiček, kde počet funkčních řešení tvoří zlomek všech možných řešení.

Pro řešení problému škálovatelnosti reprezentace byla navržena různá řešení. Sem patří například evoluce na úrovni funkčních bloků, kde jsou namísto hradel použity složitější komponenty jako sčítačky či násobičky a namísto prostých vodičů jsou využita vícebitová propojení [12]. Dále jsou to například modulární [14] či sebemodifikující se [3] kartézské genetické programování. Některé modifikace CGP budou představeny v následující kapitole.

Kapitola 4

Modifikace CGP

4.1 Modulární CGP

Myšlenkou modulárního kartézského genetického programování (dále ECGP z anglického Embedded CGP) je nalézt, zachovat a dále používat částečná řešení. Tedy například ze základních funkcí AND, OR a NAND za pomoci evoluce zkonstruovat funkci XOR a dále ji používat jako základní stavební blok.

Reprezentace programu je obdobná jako u základní verze CGP. Uzly jsou však typicky uspořádány v topologii jednoho řádku a s maximálním $l - back$ parametrem. Toto uspořádání umožňuje tvorbu modulů, která je popsána dále.

V části o ECGP jsem čerpala z [14], [16] a [15].

4.1.1 Moduly

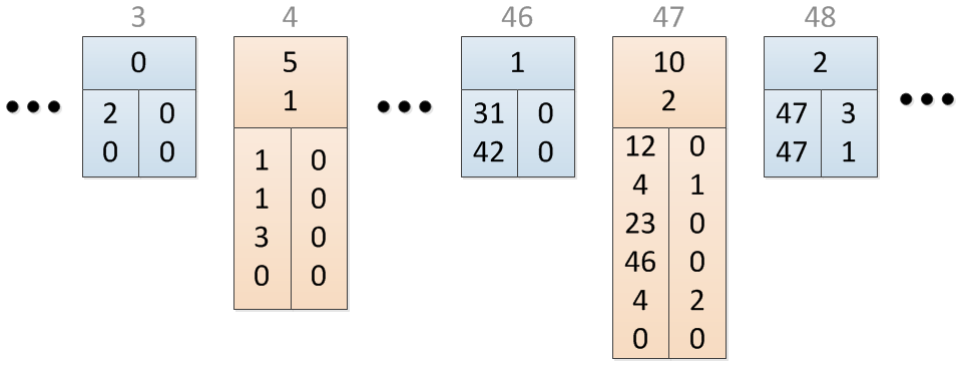
Modul je vytvořen pomocí operace *compress*. Operace vybere dva body a všechny uzly mezi nimi se zapouzdří do modulu. V genotypu je pak modul reprezentován jako jeden element. Oproti ostatním jednoduchým elementům (uzlům grafu) však může obsahovat více vstupů i výstupů. Vytvořený modul je se svou vnitřní strukturou zařazen do listu modulů. Při vzniku platí jistá pravidla a omezení. Moduly se nemohou vnořovat, tedy vybraný interval musí obsahovat pouze základní funkce. Omezen je také počet uzlů uvnitř modulu. Vytvořené moduly mohou být použity při běžné mutaci jako další možné funkce.

Moduly, které vznikly operací compress se mohou i zpětně rozpadnout. Tato operace se nazývá *expand*. Obě operace jsou aplikovány s jistou pravděpodobností. Pravděpodobnost rozpadu bývá oproti vzniku dvojnásobná.

4.1.2 Zakódování

Přítomnost modulů si vyžaduje složitější zakódování každého uzlu. Na obrázku 4.1 je schématicky znázorněno zakódování části chromozomu. V chromozomu jsou dva druhy uzlů: základní (modré) a moduly (oranžové). Oba druhy se skládají ze dvou částí, a to hlavička (nad horizontální čarou) a tělo. Hlavička u zákadního uzlu obsahuje číslo funkce, u modulu navíc typ modulu. Typ modulu je určen tím, zda byl vytvořen operací compress či prostou mutací aplikovanou na základní uzel. Tělo uzlu obsahuje čísla uzlů, na které jsou připojeny vstupy (levá část), a navíc pořadové číslo výstupu uzlu, na který je vstup připojen. Například uzel číslo 47 je modul se šesti vstupy. První vstup má připojen na uzel 12(0). Číslo v závorce říká, že první vstup uzlu 47 je připojen na nultý (respektivě první v pořadí, protože

indexujeme od nuly) výstup uzlu 12. Druhý vstup uzlu 47 je připojen na výstup uzlu 4, který je druhý v pořadí.



Obrázek 4.1: Schéma zakódování chromozomu modulárního CGP.

4.1.3 Mutace

Kromě mutace, která byla popsána u základní verze kartézského genetického programování, existuje ještě dalších pět různých mutací modulů.

První dvě jsou přidání a odebrání vstupu (*add input*, *remove input*). Po přidání vstupu je náhodně vybrán uzel uvnitř modulu, který bude k novému vstupu připojen. Odebírání je omezeno tak, že každý modul musí mít alespoň dva vstupy. Také je nutno změnit připojení uzlů, které byly spojeny s právě rušeným vstupem, a to buď k jinému vstupu modulu nebo k jinému přípustnému uzlu.

Dalšími mutacemi jsou přidání a odebrání výstupu (*add output*, *remove output*). Po přidání výstupu je náhodně vybrán uzel, který je k němu následně připojen. Odebírání je omezeno tím, že modul musí mít alespoň jeden výstup. Všechny uzly či moduly, které byly připojeny ke zrušenému výstupu musí být přepojeny na jíný výstup stejného modulu.

Posledním operátorem je běžná mutace s omezením na modul. Mutovat mohou všechny uzly uvnitř modulu, a to jak jejich vstupy, tak jejich funkce. Mutace funkce je omezena tím, že moduly se nemohou vnořovat, tedy mohou být použity pouze základní funkce.

4.1.4 Aplikace

Modulární kartézské genetické programování bylo testováno na několika typech malých obvodů jako sudá parita [14] či sčítačka [15]. Ve většině případů vykazovalo použití modulů zrychlení výpočtu ve smyslu počtu generací, které byly potřeba k dosažení řešení. Následující tabulka 4.1 shrnuje některé problémy, počet potřebných generací u CGP a ECGP a zrychlení, ke kterému došlo díky použití modulů.

Z výsledků shrnutých v tabulce 4.1 je zřejmé, že modulární CGP bylo úspěšné pro problém sudé parity, vícebitových sčítaček a násobiček. Zatímco u jednobitové sčítačky a u komparátoru došlo použitím modulů ke zpomalení.

Základní problém metody je, že modul vzniká zcela náhodně z uzlů, které leží těsně za sebou. Nezohledňuje ani to, zda jsou uzly ve výsledném kandidátním řešení aktivní či nikoli. Často tak mohou vznikat zcela neužitečné moduly. Mnohem efektivnější by mohlo být, kdyby moduly vznikaly pouze z právě aktivních uzlů.

problém	CGP	ECGP	zrychlení
sudá 4-parita	20 432	16 324	1,25
sudá 5-parita	73 393	45 480	1,61
sudá 6-parita	243 105	71 941	3,38
2-bitová násobička	35 840	35 520	1,01
3-bitová násobička	8 659 840	1 917 760	4,52
1-bitová sčítačka	26 720	35 840	0,75
2-bitová sčítačka	493 760	203 520	2,43
1-bitový komparátor	2 880	3 200	0,90
2-bitový komparátor	78 880	87 360	0,90

Tabulka 4.1: Srovnání CGP a ECGP. Uveden je vždy průměrný počet generací potřebných pro nalezení řešení a zrychlení, ke kterému došlo používáním modulů [14], [15].

4.2 Sebemodifikující se CGP

V biologii se sebemodifikací rozumí transformace genotypu na fenotyp. Proces dekódování je v buňce ovlivněn řadou faktorů. Sebemodifikující se kartézské genetická programování (dále jen SM-CGP) je pak takové rozšíření CGP, které umožňuje změnu fenotypu po ukončení evoluce. Takto lze řešit rozsáhlé problémy s více vstupy i výstupy [1]. Další informace o SM-CGP jsem čerpala z [3] a [2]

Reprezentace programu i jeho vyhodnocení má několik odlišností oproti standardnímu CGP. Tyto vlastnosti budou podrobněji popsány v následující sekcích.

4.2.1 Reprezentace

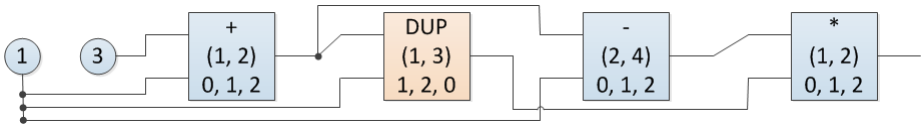
Podobně jako v modulárním CGP je i v SM-CGP uvažováno pole elementů (uzlů grafu) o jednom řádku a s maximálním $l - back$ parametrem.

Uzel grafu je kódován jako šestice. První tři čísla vyjadřují funkci a připojení vstupů. Druhá trojice jsou parametry sebemodifikující funkce. Zatímco základní varianta CGP používá absolutní adresování uzlu, SM-CGP používá relativní způsob kódováni. Má-li gen určující uzel, ke kterému je připojen vstup aktuálního uzlu, hodnotu 1, pak to znamená, že odpovídající vstup je připojen na výstup předcházejícího uzlu. Pokud je relativní adresa větší než reálný počet uzlů, na které lze element připojit, předpokládá se na tomto vstupu hodnota 0.

Schéma zakódování je zobrazeno na obrázku 4.2. Každý uzel je reprezentován svou funkcí (logickou nebo sebemodifikující). Dvojice určuje připojení vstupů (relativní adresa). Například poslední uzel má funkci sčítání a jeho vstupy jsou připojeny na první a druhý předcházející uzel. Trojice jsou parametry, které jsou potřebné pro sebemodifikující funkce. Ty budou blíže vysvětleny v následující části.

Vstupy a výstupy programu mají speciální reprezentaci pomocí uzlů. Vstupní uzel nese funkci INPUT. Pokud je funkce zavolána, uzel poskytne jednu ze vstupních hodnot programu. Při každém dalším volání poskytne další vstupní hodntu (při prvním volání první hodnotu, při druhém druhou atd.). Pokud je funkce volána vícekrát než je vstupních hodnot, uzel začne přiřazovat zase od začátku.

Výstup je reprezentován uzlem s funkcí OUTPUT. Pokud není v kandidátním řešení žádný takový uzel, jako výstupní uzly se vezme potřebný počet uzlů od konce. Pokud jich



Obrázek 4.2: Schéma zakódování sebemodifikujícího se CGP. Uzly s výpočetními funkcemi jsou vyznačeny modře, uzel se sebemodifikující funkcí oranžově. Čísla v kroužku představují vstupy s hodnotami.

je naopak více než je třeba, vezme se potřebný počet nejpravějších výstupních uzlů. V případě, že kandidátní řešení obsahuje menší počet výstupů, než je počet výstupů programu, považuje se takové řešení za špatné, což se odrazí v jeho ohodnocení.

4.2.2 Množina funkcí

Množinu používaných funkcí, můžeme rozdělit na dvě části, a to výpočetní funkce a funkce modifikující strukturu kandidátního řešení. Výpočetní funkce jsou dány řešeným problémem a může jít například o logické funkce jako AND nebo OR.

Sebemodifikujících funkcí je celá řada. Jejich množství se může lišit dle řešeného problému. Některé z nich jsou uvedeny v tabulce 4.2. Další lze nalézt v [1]. Sebemodifikující funkce pracují se čtyřmi parametry. Tři parametry (p_0, p_1 , p_2) nese zakódovaný uzel. Čtvrtý parametr (x) vyjadřuje číslo pozice uzlu. Například na obrázku 4.2, který byl uveden výše, je druhý uzel se sebemodifikující funkcí DUP (duplikace). Parametry nesou hodnoty $p_0 = 1$, $p_1 = 2$, $p_2 = 0$. Čtvrým parametrem je pořadové číslo uzlu, tzn. $x = 1$, číslováno od nuly.

Pro zacházení s adresami a parametry je několik pravidel. Adresy, které jsou mimo rozsah grafu se upraví tak, že adresy menší než 0 jsou považovány za nulové a adresy přesahující rozsah grafu jsou upraveny na délku grafu. Redundantní operace (např. kopírování nula uzlů) jsou ignorovány.

funkce	**popis**
MOV (Přesun)	Přesune uzly mezi $(p_0 + x)$ a $(p_0 + x + p_1)$ a vloží je za uzel na pozici $(p_0 + x + p_2)$.
DUP (Duplikace)	Zkopíruje uzly mezi $(p_0 + x)$ a $(p_0 + x + p_1)$ a vloží je za $(p_0 + x + p_2)$.
DEL (Smazání)	Vymaže uzly mezi $(p_0 + x)$ a $(p_0 + x + p_1)$
CHF (Změna funkce)	Změní funkci uzlu p_0 na funkci, kterou nese uzel p_1.

Tabulka 4.2: Příklad některých sebemodifikujících funkcí [2].

4.2.3 Vyhodnocení grafu

Nejdříve se vytvoří prvotní fenotyp, který je kopií genotypu. Genotyp tak zůstává po celou dobu nezměněn a všechny modifikace probíhají iterativně na fenotypu. Vyhodnocování začíná u výstupních uzlů a rekurzivně se postupuje až ke vstupům. Běžné výpočetní funkce se vyhodnocují jako u základního typu CGP.

Je-li nalezena sebemodifikující funkce a hodnota prvního vstupu je větší než hodnota druhého, sebemodifikující funkce je zařazena do *„To Do“* fronty a tato hodnota projde dále.

Pokud je větší druhá hodnota, modifikace probíhat nebude a tato hodnota pouze projde uzlem. Vyhodnocování pak probíhá v iteracích, kdy se prochází *„To Do“* fronta a postupně se aplikují jednotlivé modifikace.

4.2.4 Aplikace

Sebemodifikující se kartézské genetické programování bylo testováno na různých problémech například aproximace čísla π [3] či generování druhých mocnin [1]. Z kombinačních obvodů, které byly zmíněny u CGP je to pak sudá parita [2]. Z tabulky 4.3 je zřejmé že SM-CGP vykazovalo značné zrychlení hlavně pro obvody s více vstupy. Pro obvod se čtyřmi či pěti vstupy znamenala implementace sebemodifikace spíše zpomalení.

problém	CGP	SM-CGP	zrychlení
sudá 4-parita	81 728	308 643	0,26
sudá 5-parita	293 572	309 990	0,95
sudá 7-parita	3 499 532	313 489	11,16
sudá 8-parita	10 949 256	313 987	34,87

Tabulka 4.3: Srovnání CGP a SM-CGP. Uveden je vždy průměrný počet generací potřebných pro nalezení řešení a zrychlení, ke kterému došlo použitím SM-CGP [2].

4.3 CGP s více chromozomy

Myšlenka CGP s více chromozomy (dále jen MC-CGP) spočívá v rozložení problému s více výstupy na menší podproblémy s jedním výstupem [17]. Každý výstup je řešen samostatně, což problém zjednodušuje a řešení zrychluje. Tato varianta je proto vhodná pro obvody s více výstupy jako jsou například násobičky.

MC-CGP existuje i ve verzi s použitím modulů [17]. V tomto případě je pro všechny chromozomy používán pouze jeden list modulů. Pro zjednodušení se však dále budeme věnovat variantě bez modulů.

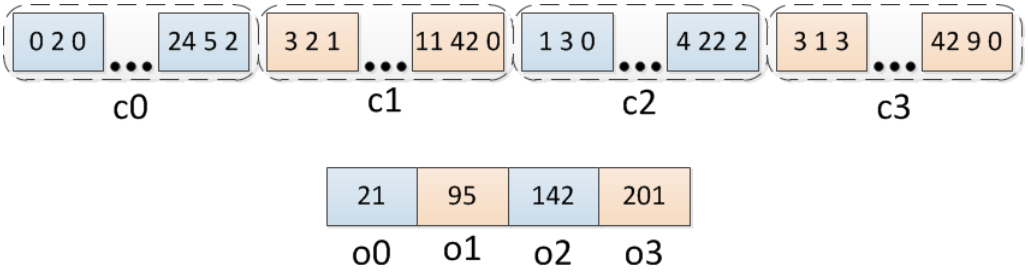
4.3.1 Reprezentace

Genotyp se skládá z několika chromozomů o stejném počtu uzlů a jedním výstupem. Vstup každého uzlu může být připojen na uzly v témže chromozomu nebo na vstupy celého programu. Pro připojení v rámci chromozomu platí stejná pravidla jako u základní varianty CGP. Na obrázku 4.3 je schématicky znázorněn genotyp pro 2-bitovou násobičku. Každý výstup je zajišťován jiným chromozomem.

4.3.2 Fitness funkce a strategie

Fitness funkce se počítá obdobně jako u základní varianty CGP, ale s malou změnou. Pro každý chromozom genotypu se určí zvlášť jako hammingova vzdálenost skutečného a požadovaného výstupu. Výsledkem fitness genotypu je tedy n hodnot, kde n je počet výstupů.

Evoluční strategie je opět $(1+\lambda)$ jako u základní varianty CGP. Změna je však ve výběru nejlepšího jedince. Nejlepší jedinec je vytvořen tak, že pro každý vstup je vybrán chromozom



Obrázek 4.3: Schéma genotypu CGP s více chromozomy se čtyřmi výstupy. Každý chromozom ($c0$ - $c3$) je pripojen na jeden výstup ($o0$ - $o3$).

s nejlepší fitness a tyto chromozomy jsou spojeny v nový genotyp. Nově vytvořený jedinec se stává rodičem a je z něj vytvořeno λ potomků.

4.3.3 Aplikace

CGP s více chromozomy bylo testováno na obvodech jako sčítačka či násobička [17]. Rozložení na podproblémy vykazovalo zrychlení a to jak u varianty bez modulů tak i s nimi. Příklady některých problémů jsou opět shrnuty v tabulce.

problém	**CGP**	**MC-CGP**	**zrychlení**
2-bitová násobička	52 000	11 200	4,64
3-bitová násobička	18 509 600	873 600	21,19
2-bitová sčítačka	469 200	140 800	3,33
3-bitová sčítačka	8 190 400	1 286 000	6,36

Tabulka 4.4: Srovnání CGP a MC-CGP. Pro každý problém je uveden průměrny počet generací, potřebných pro nalezení řešení a zrychlení, ke kterému došlo použitím CGP s více chromozomy [17].

Kapitola 5

Návrh řešení

V prvním kroku návrhu řešení jsem se zaměřila na analýzu základní verze CGP. K analýze jsem použila sadu nástrojů pro kartézské genetické programování [18], která byla vyvinuta na fakultě FIT VUT v Brně. Zaměřila jsem se na problém sudé 4-bitové parity. Pomocí programu cgpviewer jsem pozorovala postupný vývoj obvodu. Žádné zajímavé strukturní závislosti se mně však nepodařilo zjistit.

Pro řešení se nabízelo vylepšení ECGP tak, že modul by se vybíral pouze z právě aktivních uzlů. Nakonec jsem se však rozhodla pro vlastní metodu. Ta využívá moduly s pevně danými vstupy a výstupy, ale s proměnlivou vnitřní strukturou.

5.1 Reprezentace

Při způsobu reprezentace jsem vycházela ze základní verze CGP. Uzly grafu jsem rozmístila do jednoho řádku s možností připojení na primární vstupy nebo jakýkoli uzel před (maximální $l - back$ parametr). Uzly mohou být dvojího typu. Základní logické funkční bloky, které mají dva vstupy, jeden výstup a logickou funkci, nebo moduly. Modul může mít libovolný počet vstupů (alespoň však dva) i výstupů (alespoň jeden). Zakódovaný uzel musí nést číslo funkce či modulu a pro každý vstup jeho napojení a pořadové číslo výstupu, na který je připojen.

5.2 Modul

Modul je dán počtem vstupů, výstupů a základních funkčních bloků, které obsahuje. Moduly uvnitř modulů nejsou povoleny. Svou vnitřní strukturou odpovídá menšímu kombinačnímu obvodu.

Uzly uvnitř modulu jsou opět v jednom řádku s maximálním $l - back$ prametrem. Jediným omezením je, že výstup nelze připojit přímo na primární vstup. V takovém případě by vznikla propojka, což není žádoucí.

Pro vývoj modulu je důležitá jeho populace, která se tvoří obvyklým způsobem. Po úvodní inicializaci je každý jedinec ohodnocen a vybrán rodič. Rodič a jeho potomci vzniklí mutací tvoří další generaci. V populaci modulu je zapotřebí znát jedince, s nímž má vybraný chromozom reprezentující celý obvod nejlepší fitness a její hodnotu.

5.3 Algoritmus

Běh celého programu se řídí základním principem evolučních algoritmů, jak byl popsán v 2.1. Nejprve se nainicializuje modul a jeho populace. Na počátku se označí první jedinec populace jako nejlepší. S tímto modulem se začne vývoj chromozomu. Až nastane vývoj modulu, použije se tento jedinec jako rodič. Pokud bude používáno více modulů, nainicializuje se populace pro každý modul zvlášť. Po té se nainicializuje počáteční populace chromozomů. Jako funkce elementů lze použít základní logické funkce i moduly. Celá evoluce probíhá ve smyčce dokud není dosažen zadaný počet generací nebo není dosažena maximální fitness. Hlavní smyčka je rozdělěna na dvě části. Jsou to *evoluce chromozomu* a *evoluce modulu*.

Evoluce chromozomu trvá určený počet generací, kdy probíhají mutace elementů v chromozomu. Mění se jejich připojení nebo funkce. Jako funkce se zde používají, stejně jako v inicializaci, základní logické funkce i moduly. V případě změny elementu na modul, či naopak, je potřeba pohlídat počet vstupů a výstupů. Při změně počtu vstupů, je třeba mít na paměti, že každý musí být připojen. Pokud se mění počet výstupů (nový počet bude menší), je potřeba překontrolovat zbylé elementy, jestli nejsou připojeny na zaniklý výstup a bude třeba je přepojit na jiný výstup téhož elementu. Po vyčerpání zadaného počtu generací pro chromozom, začíná vývoj modulů.

Evoluce modulu probíhá následovně. Z části, kdy se vyvíjel chromozom se vezme chromozom, který měl dosud nejlepší ohodnocení. Na tomto chromozomu se pak vyvíjí modul. Populace modulu se vyvíjí běžným způsobem, jako u kartézského genetického programování. Při mutaci elementů v modulu se opět mění připojení nebo funkce. Každý element může být připojen na primární vstup modulu nebo na nějaký element před ním. Výstup může být připojen pouze na jeden z vnitřních uzlů. Vybraný chromozom se ohodnotí pro každého jedince z populace modulu zvlášť. Jedinec s nímž měl chromozom lepší nebo stejnou fitness jako s rodičovským modulem, se stává rodičem do další generace. Tímto způsobem se modul vyvíjí po předem určený počet generací. Pokud je modulů více, po ukončení vývoje prvního modulu se začne vyvíjet modul další. Při ohodnocování se do chromozomu dosazuje vždy ten jedinec z populace modulu, který skončil jako nejlepší (pokud se tento modul zrovna nevyvíjí), nebo postupně všichni jedinci (pokud se zrovna vyvíjí). Po ukončení evoluce všech modulů nastává opět evoluce chromozomu. V chromozomu se pak používají ti jedinci z populací modulů, kteří skončili jako nejlepší.

Při evoluci modulů se nabízela možnost vyvíjet všechny moduly současně. Pro m modulů, každý s populací o velikosti p, by však bylo potřeba p^m ohodnoceni v jedné generaci. Předpokládám, že konečné řešení by se dosáhlo za menší počet generací, ovšem časově by bylo řešení mnohem náročnější. Proto jsem se nakonec rozhodla vyvíjet moduly postupně, kdy v jedné generaci ohodnotím p jedinců, stejně jako při vývoji chromozomu.

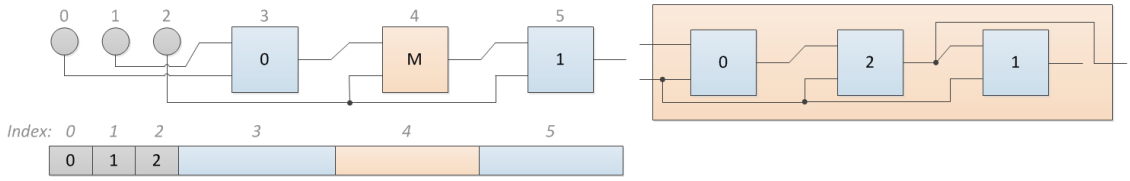
5.4 Dekomprese modulů

Fitness funkce vyhodnocuje základní funkční bloky. Před samotným ohodnocením je proto třeba každý chromozom přepsat do základních hradel. K tomuto účelu slouží dekomprese modulů, při které se modul rozepíše do jednoduchých hradel.

Největším problémem při dekompresi modulů je napojení na ostatní hradla. Při dekompresi dochází k prodlužování chromozomu, protože místo jednoho modulu se zařadí několik jednoduchých elementů. Připojení vstupů dále v chromozomu, které byly původně připojeny na modul, je potřeba přečíslovat. Tento problém jsem se rozhodla vyřešit pomocným polem, které vzniká průběžně a obsahuje nová posunutá čísla bloků. Výhodou tohoto

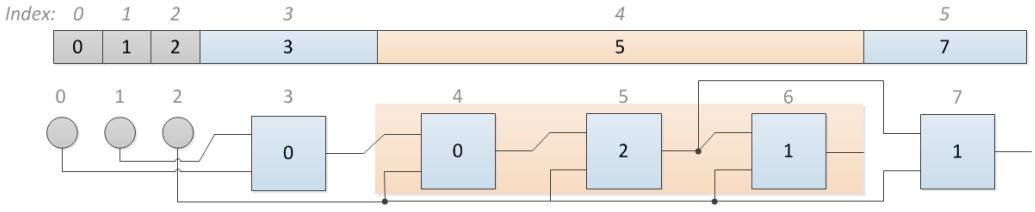
přístupu je, že všechny moduly jsou rozbaleny během jediného průchodu chromozomem.

Na obrázku 5.1 je znázorněna část kombinačního obvodu s třemi primárními vstupy (šedé kruhy), jedním modulem (oranžově), pomocné pole a vnitřní struktura modulu. Na počátku dekomprese se do pomocného pole na indexy primárních vstupů vloží tyto indexy.



Obrázek 5.1: Schéma opravy napojení vstupů během dekomprese modulů. Nalevo část kombinačního obvodu s třemi vstupy (šedě) a jedním modulem (oranžově). Pod obvodem je pomocné pole s počáteční inicializací. Vpravo vnitřní struktura modulu.

Při dekompresi se postupně prochází každý element v chromozomu. Prvním elementem na obrázku 5.1 je funkční blok. V takovémto případě se přečte napojení vstupů (čísla 1 a 0). Tato čísla slouží jako indexy do pomocného pole, které obsahuje nová čísla, na které se vstupy elementu přepojí. V případě připojení na primární vstupy k žádnému posunu nedochází. Nakonec se do pomocného pole na index právě zpracovávaného elementu vloží nové posunuté pořadové číslo tohoto elementu. Protože ještě nenastal posun, vloží se číslo 3. Dalším elementem je modul. Modul obsahuje celkem tři funkční bloky, které se vloží do chromozomu. Bloky napojené na primární vstupy modulu se napojí dle připojení elementu reprezentující modul. Výstup modulu je napojen na druhý blok (poslední blok není použit). Po vložení do chromozomu bude mít tento výstupní blok pořadové číslo 5. První blok získá pořadové číslo zpracovávaného elementu. Do pomocného pole se tedy na index elementu vloží číslo 5. Všechny další elementy, které byly v původním chromozomu připojeny na blok číslo 4 naleznou v pomocném poli na indexu 4 nové připojení, číslo 5. Tímto je modul zpracován a přejde se k dalšímu elementu. V našem příkladě se jedná o jednoduchý funkční blok. Pomocí pomocného pole se přepojí vstupy. Nové pořadové číslo tohoto bloku je 7. Do pomocného pole se na index 5 (index zpracovávaného elementu před dekompresí) vloží nové pořadové číslo elementu. Výsledek dekomprese uvedeného příkladu i s vyplněným polem je uveden na obrázku 5.2.



Obrázek 5.2: Schéma opravy napojení vstupů během dekomprese modulů. Nahoře vyplněné pomocné pole. Dole chromozom s rozpojeným modulem. Oranžově jsou podbarveny bloky, které byly vloženy místo modulu.

Kapitola 6

Implementace

Program jsem implementovala v jazyce C/C++ s použitím knihovny STL. Celý program je rozdělen do několika zdrojových souborů. Soubor main.cpp s funkcí `main()`, která obsahuje počáteční inicializaci a hlavní smyčku, která řídí evoluci. Soubor cgp.cpp obsahuje funkce, které jsou potřeba pro běh základní verze CGP od inicializace populace, přes mutaci a ohodnocení až po funkci, která vypisuje výsledný chromozom ve formátu pro nástroj cgpviewer. Poslední zdrojový soubor mod.cpp obsahuje funkce pro práci s moduly. Tyto funkce jsou blíže popsány v části 6.3.

6.1 Konstanty

Všechny konstanty jsou definovány v souboru modcgp.h. V tabulce 6.1 je uveden jejich výčet a popis.

konstanta	**popis**
POP_MAX	počet chromozomů v populaci
MUT_MAX	maximální počet genů v chromozomu pro mutaci
GENERATIONS	maximální počet generací
FUNCTIONS	počet funkci použitých pro základní elementy
PARAM	počet základních elementů v chromozomu
MOD_POP_MAX	počet modulů v populaci
MOD_MUT_MAX	maximální počet genů pro mutaci elementů v modulu
MOD_FUNCTIONS	počet funkcí pro elementy v modulu
MOD_PLUS_FUNC	posun číslování funkcí v modulu
CHR_GEN	počet generací pro vývoj chromozomu (jeden cyklus)
MOD_GEN	počet generací pro vývoj modulu (jeden cyklus)

Tabulka 6.1: Seznam použitých konstant a jejich popis.

V programu uvažuji i možnost, že elementy modulu mohou nabývat jiných funkcí než elementy v chromozomu. Proto jsem zařadila dvě konstanty, které určují počet funkcí v chromozomu (FUNCTIONS) a počet funkcí v modulu (MOD_FUNCTIONS). Samotné funkce jsou pak uvedeny ve funkci `fitness()`, která se nachází v souboru cgp.cpp. Zatímco funkce v chromozomu jsou číslovány od nuly, identifikátory funkcí v modulu jsou posunuty o konstantu MOD_PLUS_FUNC.

Střídání evoluce chromozomu a modulů se řídí konstantami CHR_GEN a MOD_GEN. Ty určují po kolika generacích vystřídá evoluci chromozomu evoluce modulu a naopak. V případě více modulů platí MOD_GEN pro každý modul zvlášť. Pro dva moduly bude cyklus vypadat následovně. Začne se vývojem chromozomu po CHR_GEN generací. Po té jej vystřídá vývoj prvního modulu po MOD_GEN generací. Následuje vývoj druhého modulu také po MOD_GEN generací. Zde cyklus končí a opět se začne vyvíjet chromozom.

6.2 Struktury a datové typy

V hlavičkovém souboru struktury.h jsou uloženy všechny použité struktury. Ty slouží pro reprezentaci elementu, modulu a informace o populaci modulu. Jsou v něm uloženy i používané datové typy definované pomocí klíčového slova typedef.

Element chromozomu je reprezentován strukturou element.

```
typedef struct{
    int id;
    bool mod;
    vector<int> in;
    vector<int> out;
}element;
```

Význam položek:

`id`	Číslo logické funkce, v případě modulu se jedná o číslo modulu (index do listu modulů). Výstupní element nese hodnotu -1.
`mod`	Indikátor, zda se jedná o modul. Pokud ano, nese hodnotu 1, jinak 0.
`in`	Napojení vstupů elementu. Index ve vektoru odpovídá pořadovému číslu vstupu.
`out`	Obsahuje čísla výstupů, na něž jsou připojeny vstupy. Index ve vektoru odpovídá pořadovému číslu vstupu.

Modul je reprezentován strukturou modul.

```
typedef struct{
    int in_num;
    int out_num;
    int el_num;
    vector<element> el;
}modul;
```

Význam položek:

`in_num`	Počet vstupů.
`out_num`	Počet výstupů.
`el_num`	Počet funkčních bloků.
`el`	Elementy modulu.

Potřebné informace o populaci modulu jsou zaznamenány ve struktuře info.

```
typedef struct{
    int bestidx;
    int bestfit;
}info;
```

Význam položek:

`bestidx` Index nejlepšího modulu v populaci.

`bestfit` Fitness chromozomu s nejlepším modulem v populaci.

Datový typ chromozomu je definován jako vektor struktur element.

```
typedef vector<element> chromozom;
```

Posledním datovým typem je populace modulu. Ta je definována jako vektor struktur modul.

```
typedef vector<modul> modul_pop;
```

List modulů je pak vytvořen ve funkci `main()` a jedná se o vektor populací modulů. Informace o populacích (struktury info) jsou také uloženy ve vektoru. Indexy obou vektorů spolu souvisejí. Nultý modul má svoji populaci na nultém indexu v listu a informace o jeho populaci se opět nacházejí na nultém indexu.

6.3 Funkce pro práci s moduly

Funkce pro práci s moduly jsou obsaženy v souboru modcgp.cpp. Popis nejdůležitějších z nich je uveden níže.

```
modul_pop mod_init(int el, int in, int out);
```

Tato funkce zajišťuje vytvoření modulu včetně jeho populace. Jako parametry přijímá počet funkčních bloků, počet vstupů a počet výstupů modulu.

```
void mod_mutace(modul_pop &mdp, int parent);
```

Další významnou funkcí je mutace, kterou vzniká nová populace. Funkce přijímá jako parametry původní populaci a index rodičovského jedince v ní.

```
chromozom decompress(chromozom &chr,
                     vector<modul_pop> &modul_list,
                     vector<info> &best_mod,
                     int idm,
                     int var);
```

Funkce zajišťuje dekompresi modulů v chromozomu, kterou je třeba udělat před ohodnocením či výpisu chromozomu. Jako parametry přijímá chromozom, který bude ohodnocován, list modulů a vektor s informacemi o nejlepších jedincích v populacích modulů. Poslední dva parametry souvisejí s vyvíjením populace modulu. Pokud se vyvíjí některý modul v parametru *idm* se předá index populace modulu, která se vyvíjí, a v parametru *var* index jedince z populace, který se má právě dosadit a ohodnotit. Pokud se zrovna nevyvíjí žádný modul, oba parametry nesou hodnoty -1 a do chromozomu se vloží moduly, které jsou zapsány ve vektoru *best_mod*.

```
vector<int> mod2el(element el,
                   chromozom &p_chr,
                   int idx,
                   modul &m,
                   vector<vector<int> > &shift);
```

Tato funkce je volána při dekompresi, pokud se v chromozomu nalezne modul, aby jej rozložila na jednotlivá hradla. Jako parametry přijímá element z chromozomu, který obsahuje připojení na ostatní elementy. Dále chromozom bez modulů, který postupně vzniká ve funkci `decomperss()`, index elementu představující modul v chromozomu, modul, který je právě rozkládán a nakonec pomocný posuvný vektor na přečíslování připojení vstupů. Funkce vrací vektor, který je připojen do pomocného posuvného vektoru na index rozdělávaného elementu.

6.4 Vizualizace chromozomu

Pro vizualizaci výsledného řešení jsem se rozhodla využít nástroj cgpviewer, který byl vyvinut na fakutě FIT VUT v Brně. V práci jsem proto implementovala dvě funkce, které jsou uloženy v souboru cgp.cpp.

```
string cgp2str(chromotom &chr, int uzito);
```

Tato funkce slouží pro přepis chromozomu s rozdělanými moduly do řetězce pro cgpviewer. Jako parametry přijímá dekomprimovaný chromozom a počet bloků, které jsou ve výsledném řešení použity. Vrací požadovaný řetězec.

```
void log4viewer(string str);
```

Funkce zapíše výsledné řešení do souboru result.chr, který je možno načíst přímo do nástroje cgpviewer. Přijímá jediný parametr, a to zapisovaný řetězec.

Kapitola 7

Experimentální vyhodnocení

V této kapitole jsou uvedeny výsledky experimentů implementovaného systému na vybraných typech kombinačních obvodů. Zvolené obvody jsou typickými zástupci problémů řešených pomocí CGP. Jedná se o následující benchmarkové obvody: sudá parita, řadicí síť, sčítačka a násobička. U všech experimentů jsem provedla celkem sto běhů evoluce. Ve výsledcích je uveden průměrný počet generací, za který došlo k vývoji vyhovujícího řešení, tj. byla dosáhnuta maximální hodnota fitness. V tabulce 7.1 je uvedeno základní nastavení hlavních parametrů pro všechny experimenty.

parametr	**hodnota**
POP_MAX	5
MUT_MAX	6
GENERATIONS	15 000 000
PARAM	100
MOD_POP_MAX	5
MOD_MUT_MAX	2
MOD_PLUS_FUNC	7
CHR_GEN	100
MOD_GEN	100

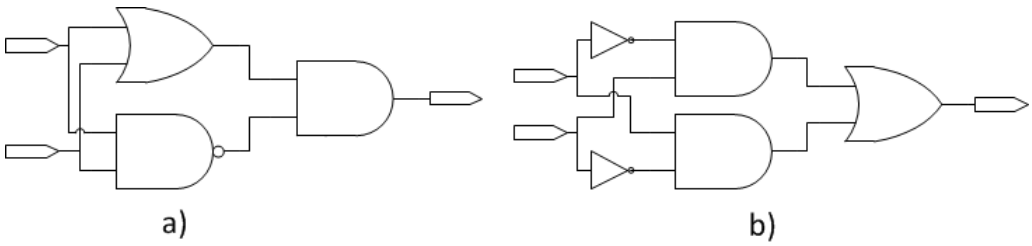
Tabulka 7.1: Nastavení hlavních parametrů evoluce.

Při určování hodnot parametrů jsem vycházela z obvyklého nastavení CGP. Počet generací jsem volila záměrně větší tak, aby byl umožněn vývoj všech řešení. Počet uzlů jsem zvolila taktéž s dostatečnou rezervou pro všechny typy obvodů. S nastavením parametrů CHR_GEN a MOD_GEN jsem experimentovala u obvodu sudá parita (část 7.1.1). Různé hodnoty parametrů však neměly výrazný vliv na běh evoluce, proto jsem se rozhodla všechny základní experimenty provést se shodným nastavením. Počet mutací v chromozomu jsem nastavila dle modulárního CGP na 2 % (viz [15]). Počet mutací u modulu jsem odvodila z obvyklého nastavení CGP, tj. 5 %. Pro moduly s pěti až patnácti elementy by však vycházela žádná až jedna mutace, což není optimální. Nakonec jsem zvolila dva geny.

7.1 Sudá parita

Prvním vybraným problémem byla sudá parita. V experimentu jsem vycházela z optimálního řešení, které se skládá pouze z funkčních bloků XOR. Myšlenkou bylo postupně vyvinout modul s funkcí XOR a dál jej využít pro sestavení obvodu.

Experiment jsem prováděla s dvěma funkčními sadami. První sadou byly funkce AND, OR a NAND pro celý chromozom i modul. XOR z těchto funkcí zabírá celkem tři elementy, má dva vstupy a jeden výstup. Modul byl tedy inicializován s dvěma vstupy, jedním výstupem a pěti elementy uvnitř. Druhou sadou byly funkce AND, OR a NOT, taktéž pro celý chromozom i modul. XOR z těchto funkcí lze sestavit různými způsoby. Já si vybrala takový, který zabírá pět elementů. Modul jsem tedy inicializovala s dvěma vstupy, jedním výstupem a s deseti elementy uvnitř. Počet elementů v modulu jsem volila vždy v nadbytku tak, aby bylo možno více variant zapojení. Schémata hradel XOR pro obě funkční sady jsou uvedeny na obrázku 7.1. U obou sad jsem pro srovnání provedla běh bez modulů (základní verze CGP) a s pevným modulem reprezentující XOR, který není předmětem evoluce, ale je použit jako další možná funkce.



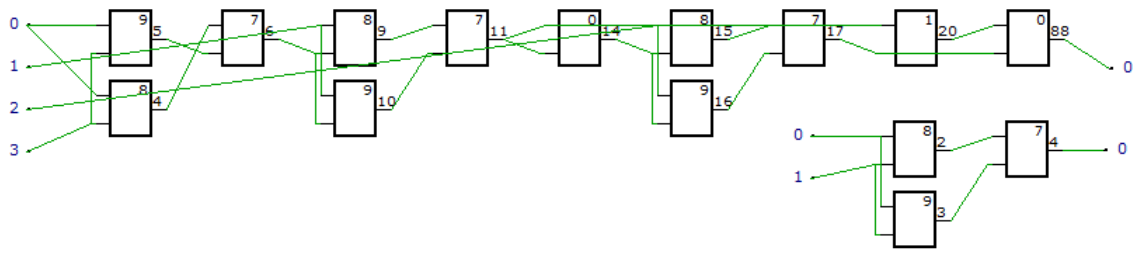
Obrázek 7.1: Schéma funkcí XOR: a) funkční sada AND, OR, NAND b) funkční sada AND, OR, NOT

Výsledky experimentu s první funkční sadou (AND, OR, NAND) jsou uvedeny v tabulce 7.2. Příklad výsledného řešení s příslušným modulem je zobrazen na obrázku 7.2. Výsledky experimentu pro druhou funkční sadu (AND, OR, NOT) jsou uvedeny v tabulce 7.3. Příklad výsledného obvodu s nalezeným modulem je zobrazen na obrázku 7.3. Poslední sloupce tabulek uvádějí zrychlení, ke kterému došlo používáním vyvíjejících se modulů vzhledem k variantě bez modulů (CGP). Obrázky byly vytvořeny pomocí programu cgpviewer. Pro větší přehlednost jsem zvolila typ zobrazení, kde pokud je to možné nejsou vstupy uzlů připojeny přímo na daný uzel či primární vstup, ale na jiný vstup uzlu před ním, který má stejné připojení.

Ze získaných výsledků je zřejmé, že vyvíjející modul má výrazný vliv na počet generací potřebných pro vývoj funkčního obvodu. U první funkční sady (AND, OR, XOR) se průměrný počet generací pohyboval okolo 10 000, a to i pro rozsáhlejší obvody s více vstupy. Pro paritu 9 a 10 bitů to znamená více jak stonásobné zrychlení oproti základní verzi CGP, tj. verzi bez modulů. Druhá funkční sada si vedla s obdobnými výsledky. Obvod s druhou funkční sadou je sice náročnější než s první, zrychlení u jednotlivých obvodů je ale srovnatelné.

problém	průměrný počet generací			zrychlení
	CGP	pevný modul	modul	CGP/modul
4-bit parita	11 506	421	9 823	1,17
5-bit parita	41 609	775	12 938	3,22
6-bit parita	117 796	1 272	9 476	12,43
7-bit parita	316 479	1 648	9 362	33,80
8-bit parita	745 413	2 517	8 206	90,83
9-bit parita	1 536 466	4 391	9 985	153,88
10-bit parita	1 992 127	4 916	14 947	133,28

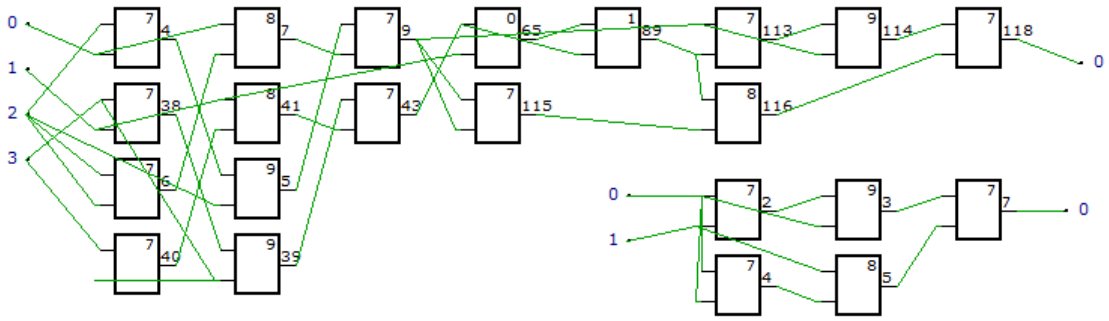
Tabulka 7.2: Shrnutí výsledků evolučního návrhu sudé parity s funkční sadou F={AND, OR, NAND}.



Obrázek 7.2: Příklad výsledného řešení 4-bitové parity včetně nalezeného modulu (vpravo dole). Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - NAND. 7 - 9 značí funkce hradel nacházejících se uvnitř modulu. Vykresleno v nástroji cgpviewer.

problém	průměrný počet generací			zrychlení
	CGP	pevný modul	modul	CGP/modul
4-bit parita	38 758	615	18 802	2,06
5-bit parita	117 704	1 148	37 817	3,11
6-bit parita	373 162	2 379	22 104	16,88
7-bit parita	1 289 040	3 088	19 372	66,54
8-bit parita	2 128 962	5 269	31 043	68,58
9-bit parita	5 117 422	6 702	37 496	136,48
10-bit parita	9 292 097	18 776	100 133	92,80

Tabulka 7.3: Shrnutí výsledků evolučního návrhu sudé parity s funkční sadou F={AND, OR, NOT}.



Obrázek 7.3: Příklad výsledného řešení 4-bitové parity včetně nalezeného modulu (vpravo dole). Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - NOT. 7 - 9 značí funkce hradel nacházejících se uvnitř modulu. Vykresleno v nástroji cgpviewer.

7.1.1 Vyhodnocení vlivu parametrů na úspěšnost evoluce

Na první funkční sadě (AND, OR, NAND) jsem provedla několik dalších experimentů, které testovaly vliv nastavení parametrů evoluce či nastavení inicializace modulu.

Nejprve jsem se zaměřila na nastavení parametrů CHR_GEN a MOD_GEN. Otestovala jsem několik variant. V prvním experimentu jsem nastavila oba parametry na 500, aby chromozom i modul měli dostatečný čas pro evoluci. V druhém jsem snížila nastavení obou parametrů na 100. V dalším experimentu jsem ještě snížila na 50. Cílem posledního experimentu bylo ověřit chování při dvojnásobném počtu generací pro vývoj modulu. Parametr CHR_GEN byl nastaven na 100, MOD_GEN na 200. U každé varianty jsem se navíc soustředila na vliv na délku výsledného chromozomu po dekompresi a na počet použitých bloků ve výsledném řešení. Výsledky této sady experimentů jsou shrnuty v tabulce 7.4. Pro každý problém je uveden průměrný počet generací potřebný pro nalezení funkčního řešení obvodu a v závorce průměrný počet funkčních bloků v chromozomu po dekompresi a průměrný počet použitých bloků ve výsledném řešení.

problém	průměrný počet generací a počet uzlů			
	v1: 500/500	**v2: 100/100**	**v3: 50/50**	**v4: 100/200**
4-bit parita	14 181 (199, 26)	9 823 (199, 26)	9 400 (202, 26)	11 594 (202, 26)
5-bit parita	18 574 (201, 30)	12 553 (206, 32)	11 207 (201, 29)	10 806 (208, 30)
6-bit parita	29 922 (205, 33)	9 476 (205, 33)	7 804 (208, 33)	13 496 (210, 33)
7-bit parita	41 626 (211, 39)	9 362 (212, 36)	7 159 (210, 36)	8 677 (212, 36)
8-bit parita	20 622 (215, 43)	8 206 (215, 40)	11 685 (216, 38)	18 501 (218, 40)
9-bit parita	13 762 (219, 43)	9 985 (219, 44)	9 051 (220, 43)	15 188 (220, 44)
10-bit parita	17 942 (222, 48)	13 293 (224, 46)	14 123 (223, 45)	21 984 (221, 45)

Tabulka 7.4: Shrnutí výsledků vlivu nastavení parametrů CHR_GEN a MOD_GEN na evoluci a délku chromozomu.

Z výsledků je zřejmé, že nastavení parametrů CHR_GEN a MOD_GEN má vliv na počet generací potřebných k vývoji obvodu. Ze zkoušených variant nejlépe vycházela varianta s nastavením 100 pro oba parametry. Ve variantě s hodnotou 50 jsou výsledky v některých případech lepší, ale někdy i horší. Výsledky těchto dvou variant nejsou příliš odlišné, proto

jsem se nakonec rozhodla ponechat pro oba parametry hodnotu 100. Z tabulky lze také vyčíst, že nastavení těchto parametrů nemá vliv na počet funkčních bloků ve výsledném chromozomu po dekompresi, ani na počet použitých bloků v řešení. Pro všechny varianty jsou tato čísla srovnatelná.

Další sadou experimentů jsem zjišťovala chování evoluce při rozdílných funkčních sadách pro modul a pro chromozom. Modul měl v tomto případě povoleny funkce AND, OR a NAND. Chromozom měl k dispozici propojku nebo modul. Pro srovnání jsem experiment provedla jak pro pevný modul, který není předmětem evoluce, tak pro vyvíjející se modul. Výsledky jsou shrnuty v tabulce 7.5.

problém	**průměrný počet generací**	
	propojka - pevný	**propojka - modul**
4-bit parita	110	8 263
5-bit parita	360	16 387
6-bit parita	1 197	105 042
7-bit parita	4 304	108 696
8-bit parita	23 211	268 144
9-bit parita	108 611	142 313
10-bit parita	521 795	387 542

Tabulka 7.5: Shrnutí výsledků evolučního návrhu sudé parity s různými funkčními sadami pro modul (AND, OR, NAND) a pro chomozom (propojka).

Oproti základní verzi CGP (tabulka 7.2) došlo ke snížení počtu generací potřebných pro nalezení funkčního řešení. Také lze pozorovat, že počet generací roste s počtem vstupů obvodu. Srovnáme-li pak výsledky s verzí, kde byla použita jedna funkční sada pro chromozom i modul (tabulka 7.2) lze vyvodit závěr, že není nutný vývoj modulu reprezentující funkci XOR. Tato domněnka se potvrdila i pozorováním výsledků v programu cgpviewer. V mnoha případech se XOR vůbec nevyvinul.

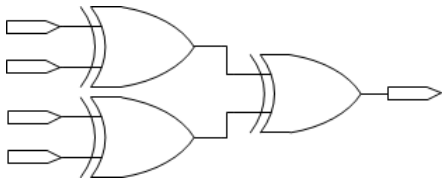
V dalším experimentu jsem ověřovala vliv nastavení inicializace modulu. Nejprve jsem se zaměřila na počet elementů v modulu. Modul představuje menší kombinační obvod. Předpokladem bylo, že počet elementů v modulu nebude mít výrazný vliv na počet generací. Tento parametr nemá výrazný vliv ani na počet elementů v základní verzi CGP. Měření jsem provedla na obou funkčních sadách v základním nastavení, tj. stejné funkce pro modul i chromozom, CHR_GEN i MOD_GEN nastaveny na 100. U první funkční sady (AND, OR, NAND) jsem použila dvě nastavení, a to pět a deset elementů uvnitř modulu. U druhé funkční sady (AND, OR, NOT) jsem použila deset a patnáct elementů uvnitř modulu. Výsledky experimentu jsem shrnula v tabulce 7.6.

Z naměřených výsledků je zřejmé, že počet elementů, které může evoluce použít uvnitř modulu, nemá žádný výrazný vliv na evoluci. Proto jsem volila počty elementů spíše menší, ale vždy s dostatečnou rezervou pro více možných řešení.

Poslední dva testy jsem provedla s pevně daným modulem v základním nastavení na funkční sadě AND, OR, NAND pro modul i chromozom. Pro experiment jsem vybrala paritu s osmi vstupy. Nejprve jsem vytvořila pevný modul se čtyřmi vstupy a jedním výstupem. Vnitřní struktura tohoto modulu odpovídala třem XOR funkcím. Schéma zapojení je na obrázku 7.4. Cílem bylo ověřit zda je modularita přínosem. Předpokladem bylo, že s větším modulem obsahující tři hradla XOR bude evoluce rychlejší než s menším modulem

problém	průměrný počet generací			
	AND, OR, NAND		AND, OR, NOT	
	5 el	10 el	10 el	15 el
4-bit parita	9 823	7 779	18 802	17 789
5-bit parita	12 553	7 422	37 817	29 194
6-bit parita	9 476	8 791	22 104	21 522
7-bit parita	9 362	7 965	19 372	22 749
8-bit parita	8 206	9 485	31 043	23 029
9-bit parita	9 985	11 484	37 496	36 118
10-bit parita	13 293	14 516	100 133	98 931

Tabulka 7.6: Shrnutí výsledků vlivu počtu elementů v modulu u evolučního návrhu sudé parity.



Obrázek 7.4: Schéma zapojení funkcí XOR ve větším modulu. V implementaci byl XOR sestaven z hradel AND, OR a NAND.

reprezentující jeden XOR. Pro poslední experiment jsem použila pevně daný modul reprezentující funkci XOR, avšak místo jednoho výstupu jsem přidala výstup, který byl připojen na jeden ze vstupů modulu. Cílem bylo zjistit, zda má na vývoj vliv počet výstupů modulu. Výsledky obou experimentů jsou shrnuty v tabulce 7.7.

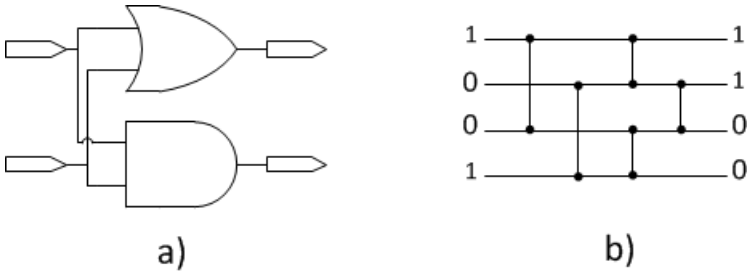
problém	průměrný počet generací		
	pevný xor	3xor	xor 2 výstupy
8-bit parita	2 517	1 594	2 223

Tabulka 7.7: Shrnutí výsledků experimentů s modulem reprezentující tři hradla XOR a s modulem s více výstupy.

Výsledky experimentu potvrdily domněnku. Při vývoji sudé parity lze použít modularizačních technik. Vývoj s větším modulem, tj. s třemi funkcemi XOR, urychlil vývoj obvodu oproti verzi s menším modulem s jednou funkcí XOR, a to 1,58-krát. Experiment také ukázal, že počet výstupů modulu nemá na úspěšnost evoluce porovnatelný vliv. Výsledky s pevným modulem XOR s dvěma výstupy byly srovnatelné s výsledky s jednovýstupovým pevným modulem.

7.2 Řadicí síť

Řadicí síť je kombinační obvod složen z komparátorů, který slouží k seřazení posloupnosti bitů. Komparátor má dva vstupy a dva výstupy, přičemž horní výstup vrací větší a spodní menší vstup. Uvnitř je složen z jednoho hradla AND a jednoho hradla OR. Schéma komparátoru a řadicí sítě je uveden na obrázku 7.5.



Obrázek 7.5: Řadicí síť: a) schéma komparátoru b) schéma 4-bitové řadicí sítě. Svislé čáry značí připojení komparátorů.

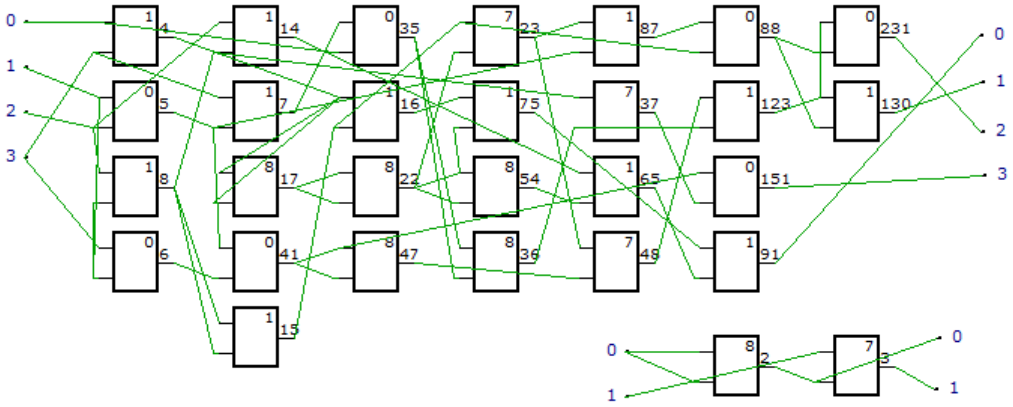
V experimentu jsem vycházela ze struktury komparátoru. Snahou bylo během evoluce vyvinout a udržet modul, který funguje jako komparátor. Modul jsem na počátku inicializovala se dvěma vstupy, dvěma výstupy a pěti elementy uvnitř. Experimenty jsem prováděla se dvěmi funkčními sadami. První sadou byly funkce AND a OR, a to jak v chromozomu, tak v modulu. Druhou sadou bylo AND, OR a propojka opět pro chromozom i modul. V obou případech jsem pro srovnání provedla experiment bez modulů (základní verze CGP) a s pevně inicializovaným komparátorem, který není předmětem evoluce.

Výsledky experimentů jsou uvedeny v tabulkách 7.8 a 7.9. Poslední sloupce tabulek uvádějí zrychlení, ke kterému došlo používáním vyvíjejících se modulů vzhledem k variantě bez modulů (CGP). Příklad výsledného obvodu pro funkční sadu AND, OR včetně nalezeného modulu je na obrázku 7.6. Příklad nalezeného obvodu pro funkční sadu AND, OR, propojka včetně příslušného modulu je na obrázku 7.7.

problém	**průměrný počet generací**			**zrychlení**
	CGP	**pevný modul**	**modul**	**CGP/modul**
4-bit síť	1 637	1 186	3 448	0,47
5-bit síť	7 251	6 413	14 580	0,50
6-bit síť	27 630	23 779	53 502	0,52
7-bit síť	110 708	63 975	135 633	0,82

Tabulka 7.8: Shrnutí výsledků evolučního návrhu řadicí sítě s funkční sadou F={AND, OR}.

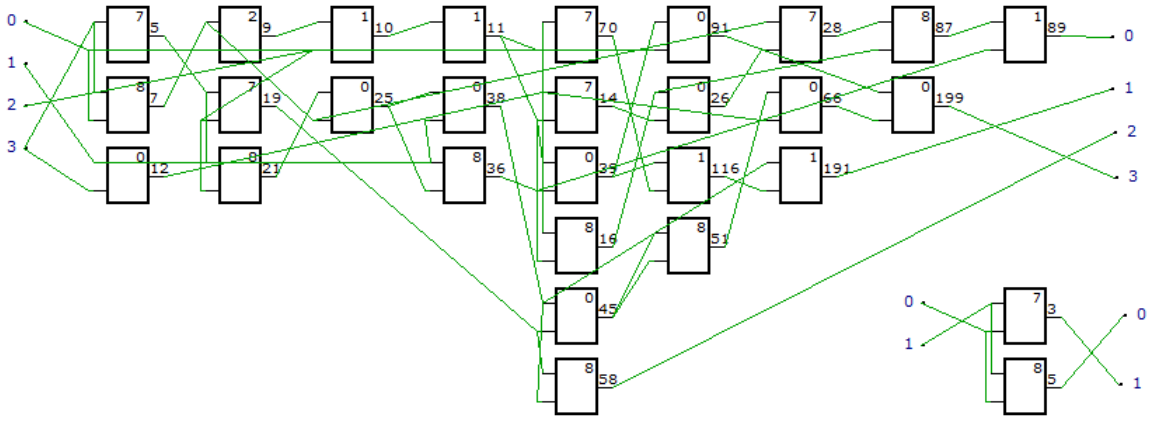
Z experimentálních výsledků je zřejmé, že ani u jedné funkční sady nedošlo používáním vyvíjejících se modulů k redukci počtu potřebných generací pro nalezení řešení. Ve většině případů došlo ke zpomalení, a to asi o polovinu. Z obrázků výsledných řešení ja také patrné, že často se z modulu použil pouze jeden výstup. Po důkladném prohlédnutí celého chromozomu jsem sice našla zapojené oba výstupy, ovšem elementy na ně připojené nebyly



Obrázek 7.6: Příklad výsledného řešení 4-bitové řadicí sítě včetně nalezeného modulu (vpravo dole). Funkční sada: 0, 7 - AND, 1, 8 - OR. 7 - 8 značí funkce hradel nacházejících se uvnitř modulu. Vykresleno v nástroji cgpviewer.

problém	průměrný počet generací			zrychlení
	CGP	pevný modul	modul	CGP/modul
4-bit síť	1 998	1 575	4 252	0,49
5-bit síť	10 214	7 341	17 573	0,58
6-bit síť	35 387	22 339	58 025	0,61
7-bit síť	123 356	66 716	139 299	0,89

Tabulka 7.9: Shrnutí výsledků evolučního návrhu řadicí sítě s funkční sadou F={AND, OR, propojka}.



Obrázek 7.7: Příklad výsledného řešení 4-bitové řadicí sítě včetně nalezeného modulu (vpravo dole). Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - propojka. 7 - 9 značí funkce hradel nacházejících se uvnitř modulu. Vykresleno v nástroji cgpviewer.

použity ve výsledném řešení. U pevně daných nevyvíjejících se komparátorů ovšem došlo k mírnému zrychlení. To dalo impulz k dalšímu experimentu.

Další sadou experimentů jsem se snažila ověřit, zda lze použít modularitu u obvodu řadicí síť. Pro chromozom jsem povolila pouze funkci propojka. Modul obsahoval funkce AND a OR. Myšlenkou bylo vyvinout modul s komparátorem a z něj sestavit celou síť. Pro srovnání jsem opět provedla experiment s pevně inicializovaným modulem, který není předmětem evoluce. Experiment s vyvíjejícím se modulem jsem provedla ve dvou verzích. První byla verze s nastavením konstant CHR_GEN i MOD_GEN na 100, jak bylo uvedeno v úvodu. Druhá verze měla obě konstanty sníženy o polovinu, nesly tedy hodnotu 50. Druhou variantu jsem zavedla po úvaze, že chromozom, který obsahuje pouze propojku nebo modul, nepotřebuje tolik generací pro vývoj. Navíc bude docházet pouze k prodlužování chromozomu, díky zabudovávání modulů a celý program bude prováděn pro delší dobu, protože dekomprese modulů je časově náročná operace. Výsledky popsaných experimentů jsou uvedeny v tabulce 7.10.

problém	**průměrný počet generací**		
	pevný modul	**modul v1**	**modul v2**
4-bit síť	1 823	4 610	4 778
5-bit síť	8 911	18 156	17 446
6-bit síť	25 854	44 871	53 706
7-bit síť	50 226	112 209	113 490

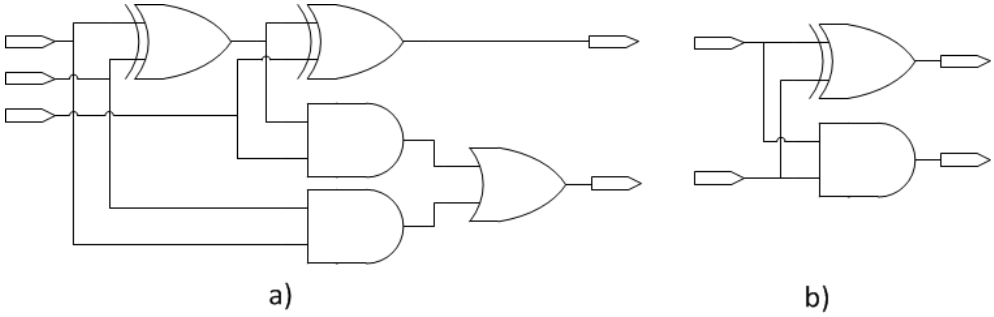
Tabulka 7.10: Shrnutí výsledků evolučního návrhu řadicí sítě s funkcí propojka pro chromozom a funkcemi AND a OR pro modul. V1 značí verzi s nastavením CHR_GEN a MOD_GEN na 100. V2 je verze s nastavením CHR_GEN a MOD_GEN na 50.

Z výsledků experimentu je zřejmé, že nastavení konstant CHR_GEN a MOD_GEN nemá na výsledné řešení vliv, jak se potvrdilo již u parity. Ani v této sadě experimentů se nepovedlo dojít k výsledku za kratší počet generací, než ve verzi bez modulu. Proto jsem experiment uzavřela s tím, že u kombinačního obvodu řadicí síť používáním modulů s pevným počtem vstupů i výstupů a vyvíjející se vnitřní strukturou nedojde k urychlení vývoje.

7.3 Sčítačka

Dalším vybraným problémem byla sčítačka. Snahou bylo vyvinout vícebitovou sčítačku s postupným přenosem složenou z úplných jednobitových sčitaček. Testovací data byla sestavena tak, že pro dvoubitovou sčítačku (tj. dva plus dva bity) měla čtyři bity pro vstupy. Pokud bych chtěla obvod složit pouze z modulů reprezentující úplnou sčítačku, nebylo by to možné. Úplná sčítačka obsahuje přenos, který ovšem v trénovacích datech není použit. Problém lze vyřešit dvěma způsoby. Buď přidat další bit k trénovacím datům představující přenos a nesoucí hodnotu 0. V tomto případě by pro vývoj posačoval jeden modul, protože celý obvod lze seskládat z jednoho typu menšího obvodu, podobně jako u předchozích experimentů. Nebo zavést dva moduly, a to úplnou a poloviční sčítačku. Schémata úplné a poloviční sčítačky jsou na obrázku 7.8.

Experiment jsem provedla ve dvou variantách. První byla pouze s jedním modulem. Zde byla snaha o vývoj úplné sčítačky. Poloviční sčítačka se tedy musela vyvinout samostatně v chromozomu. Druhá varianta obsahovala dva moduly. Jeden pro úplnou a jeden pro po-



Obrázek 7.8: Schémata sčítaček z funkcí AND, OR a XOR: a) úplná b)poloviční

loviční sčítačku. Teoreticky by tato varianta měla být ještě rychlejší. Modul pro úplnou sčítačku jsem inicializovala s třemi vstupy, dvěma výstupy a deseti elementy uvnitř. Modul pro poloviční sčítačku s dvěma vstupy, dvěma výstupy a pěti elementy uvnitř. Pro experiment jsem zvolila funkční sadu AND, OR a XOR pro chromozom i pro modul. Měření jsem pro srovnání provedla opět i v CGP bez modulů a s pevným modulem, který není předmětem evoluce.

Výsledky experimentů jsem shrnula v tabulce 7.11. Zrychlení verze s vyvíjejícími se moduly oproti verze bez modulů (CGP) je v tabulce 7.12. Příklad nalezeného obvodu s jedním modulem je na obrázku 7.9. Příklad obvodu s dvěma moduly je na obrázku 7.10, detail modulů na obrázku 7.11.

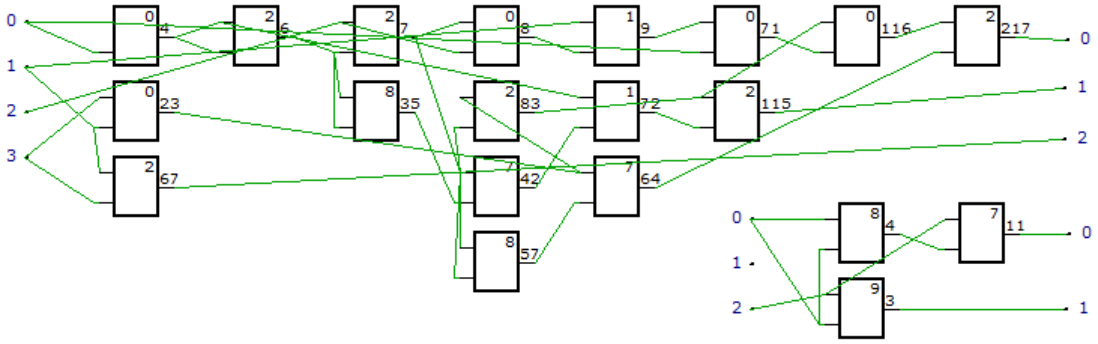
problém	**průměrný počet generací**				
	CGP	**pevný FA**	**1 modul**	**pevný FA, HA**	**2 moduly**
2-bit sčítačka	3 303	2 162	7 268	2 472	9 076
3-bit sčítačka	23 509	31 526	46 801	24 728	70 838
4-bit sčítačka	111 149	135 593	185 517	92 395	358 236
5-bit sčítačka	480 601	380 678	319 786	386 127	472 558

Tabulka 7.11: Shrnutí výsledků evolučního návrhu sčítačky s funkční sadou F={AND, OR, XOR}. FA značí úplnou sčítačku, HA poloviční sčítačku.

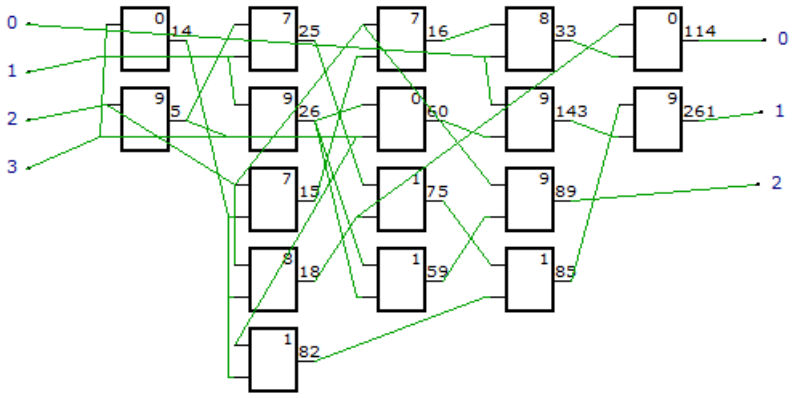
problém	**zrychlení CGP/modul**	
	FA	**FA, HA**
2-bit sčítačka	0,45	0,36
3-bit sčítačka	0,50	0,33
4-bit sčítačka	0,60	0,31
5-bit sčítačka	1,50	1,02

Tabulka 7.12: Zrychlení evoluce sčítačky při používání vyvíjejících se modulů oproti verzi bez modulů.

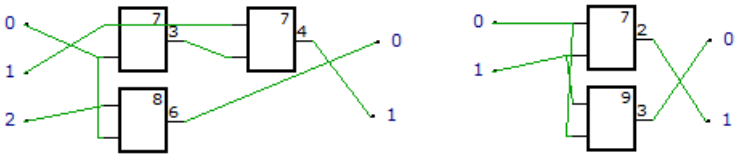
Z tabulky je zřejmé, že použitím vyvíjejících se modulů nedošlo k žádnému významnému zrychlení co do počtu generací. Ve většině případů došlo naopak ke zpomalení. Ani



Obrázek 7.9: Příklad výsledného řešení 2-bitové sčítačky včetně nalezeného modulu (vpravo dole). Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - XOR. 7 - 9 značí funkce hradel nacházejících se uvnitř modulu. Vykresleno v nástroji cgpviewer.



Obrázek 7.10: Příklad výsledného řešení 2-bitové sčítačky vykresleného v nástoji cgpviewer. Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - XOR. 7 - 9 značí funkce hradel nacházejících se uvnitř modulu.



Obrázek 7.11: Detail nalezených modulů pro obvod na obrázku 7.10. Vykresleno v nástoji cgpviewer.

domněnka, že dva moduly povedou k lepšímu výsledku než jeden modul, se nepotvrdila.

V dalším experimentu jsem použila pro chromozom jedinou funkci, a to propojku. Pro moduly byla povolena původní funkční sada AND, OR, XOR. V této variantě bylo nutno použít dva moduly, protože chromozom nemá funkce, ze kterých by sestavil poloviční sčítačku. Nastavení modulů bylo stejné jako v první sadě experimentů. Opět sem provedla dvě varianty, a to s pevným modulem, který není předmětem evoluce a s vyvíjejícím se modulem. Výsledky tohoto experimentu jsou shrnuty v tabulce 7.13.

problém	průměrný počet generací	
	propojka - pevný	propojka - modul
2-bit sčítačka	1 348	16 304
3-bit sčítačka	20 332	183 792
4-bit sčítačka	126 762	389 736
5-bit sčítačka	258 458	342 491

Tabulka 7.13: Shrnutí výsledků evolučního návrhu sčítačky s různými funkčními sadami pro modul (AND, OR, NAND) a pro chomozom (propojka).

Ani tento experiment nevedl k žádnému výraznému zlepšení, kromě pětibitové sčítačky. U té došlo opět k menšímu zrychlení a to 1,4-krát. I zde nastal problém jako u řadící sítě a z některých modulů byl využíván pouze jeden výstup. Ikdyž v paritě se nepotvrdilo, že více výstupů má vliv na evoluci, ověřila jsem to ještě posledním experimentem.

V posledním experimentu jsem se snažila ověřit, zda na vývoj nemá vliv více výstupů u modulu. Rozdělila jsem tedy úplnou sčítačku na dvě části tak, aby každý výstup měl svůj modul. Měření jsem provedla na vybraném problému tříbitové sčítačky. Modul i chromozom měli stejnou funkční sadu, a to AND, OR a XOR. Experiment jsem opět provedla jak s pevně inicializovanými moduly, tak s vyvíjejícími moduly. Výsledky jsou shrnuty v tabulce 7.14.

problém	průměrný počet generací	
	pevný	modul
3-bit sčítačka	18 117	101 560

Tabulka 7.14: Shrnutí výsledků evolučního návrhu sčítačky s jednovýstupovými moduly.

V tomto experimentu sice u varianty s pevně inicializovanými moduly došlo k mírnému zrychlení. Původní CGP se vyvinulo za 23 509, avšak jedná se o průměry ze sta běhů. Rozdíl není výrazný a varianta s vyvíjejícími moduly potvrdila, že více výstupů problémem není. Experiment jsem tedy uzavřela s tím, že mnou navržená modularizační technika není pro sčítačku přínosem.

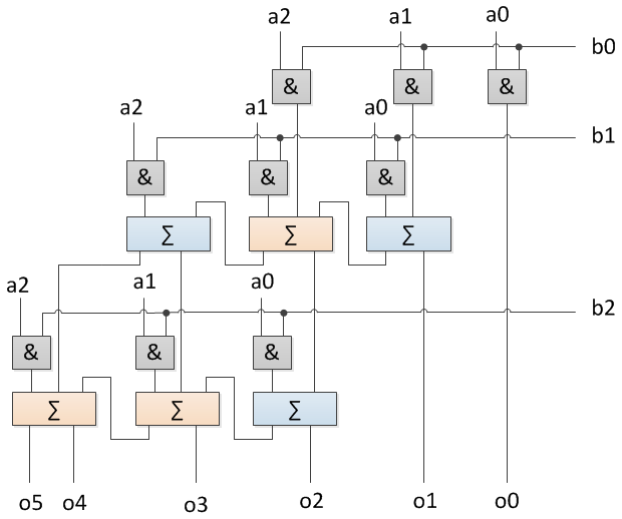
7.4 Násobička

Posledním zkoušeným obvodem byla násobička. Ta se skládá z úplných a poloviční sčítaček a z řady hradel AND. Schéma násobičky se nachází na obrázku 7.12. Jedná se o velice složitý obvod, proto jsem testovala pouze na dvou variantách, a to dvou a tříbitová. Vícebitové násobičky jsou již tak rozsáhlé, že je nebylo možno v přijatelném čase otestovat.

Jelikož se obvod skládá převážně ze sčítaček, opět jsem použila sadu funkcí AND, OR a XOR pro modul i celý chromozom. Očekávané moduly byly úplná a poloviční sčítačka. Inicialializovány byli se třemi vstupy, dvěma výstupy a deseti elementy uvnitř pro úplnou a dvěma vstupy, dvěma výstupy a pěti elementy uvnitř pro poloviční sčítačku. Experiment jsem provedla ve třech variantách, a to základní verze CGP (bez modulů), pevně inicializované moduly, které nejsou předmětem evoluce a vyvíjející se moduly.

Výsledky jsem shrnula do tabulky 7.15. V posledním sloupci je uvedeno zrychlení, ke kterému došlo používáním vyvíjejících se modulů oproti verzi bez modulů. Příklad naleze-

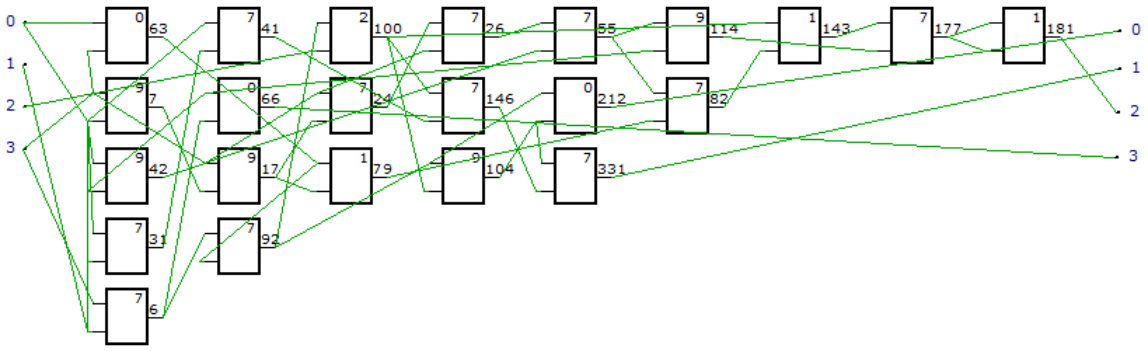
ného obvodu je na obrázku 7.13. Detail nalezených modulů na obrázku 7.14.



Obrázek 7.12: Schéma tříbitové násobičky.

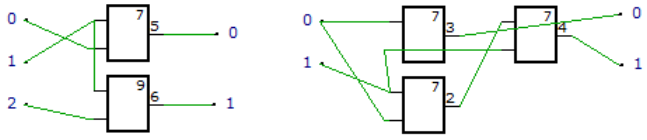
problém	průměrný počet generací			zrychlení
	CGP	pevný FA, HA	2 moduly	CGP/modul
2-bit násobička	3 118	2 299	6 467	0,48
3-bit násobička	283 861	386 513	506 212	0,56

Tabulka 7.15: Shrnutí výsledků evolučního návrhu násobičky s funkční sadou F={AND, OR, XOR}. FA značí úplnou sčítačku, HA poloviční sčítačku.



Obrázek 7.13: Příklad výsledného řešení 2-bitové násobičky vykresleného v nástoji cgp-viewer. Funkční sada: 0, 7 - AND, 1, 8 - OR, 2, 9 - XOR. 7 - 9 značí funkce hradel nacházejících se uvnitř modulů.

Ani v případě násobičky, se nepovedlo navrženou modularitou zlepšit vývoj obvodu co do počtu generací. Výsledek byl v tomto případě očekávaný, protože systém nebyl úspěšný ani u sčítaček, kde byly použity stejné moduly i funkce.



Obrázek 7.14: Detail nalezených modulů pro obvod na obrázku 7.10 v nástoji cgpviewer.

Pro úplnost jsem ještě provedla poslední experiment s různými funkčními sety pro chromozom a pro modul. V tomto případě nebylo možné v chromozomu použít pouze propojku, protože kromě úplné a poloviční sčítačky je zapotřebí funkce AND. Funkční set pro chromozom obsahoval tedy propojku a AND. Pro moduly byly povolené funkce AND, OR a XOR. Experiment jsem provedla jak s pevnými moduly tak s vyvíjejícími. Výsledky jsou shrnuty v tabulce 7.16.

problém	**průměrný počet generací**	
	propojka, AND - pevný	**propojka, AND - modul**
2-bit násobička	2 601	7 280
3-bit násobička	396 323	454 481

Tabulka 7.16: Shrnutí výsledků evolučního návrhu násobičky s různými funkčními sadami pro modul a celý chromozom.

Ani u násobičky nebyl navržený systém úspěšný. Výsledky korespondují s očekáváním, jelikož systém nebyl úspěšný ani u jednodušších sčítaček. Proto jsem experiment uzavřela s tím, že navržený systém není pro násobičku přínosem.

Kapitola 8

Závěr

V rámci práce jsem se seznámila s obecnou strukturou evolučních algoritmů. Také jsem se zabývala hlavními zástupci těchto algoritmů, jako jsou genetické algoritmy a genetické programování. Protože téma práce je zaměřeno na evoluční návrh číslicových obvodů, významnou část tvořilo seznámení se s kartézským genetickým programováním, které je pro návrh v dané oblasti nejvhodnější. Na to jsem navázala modifikacemi CGP, které se snaží řešit některé jeho nedostatky. K těmto modifikacím patří modulární CGP, sebemodifikující se CGP či CGP s více chromozomy. Hlavní náplní diplomové práce byl návrh a implementace modularizační techniky pro zefektivnění evolučního návrhu. Součástí bylo ověření účinnosti techniky na vybraných kombinačních obvodech.

Pro práci jsem navrhla systém, který modifikuje základní verzi kartézského genetického programování následujícím způsobem. Do systému jsem zavedla moduly. Ty představují menší kombinační obvody. Mají pevný a předem daný počet vstupů, výstupů i funkčních bloků. Vnitřní zapojení těchto bloků je však měnitelné a postupem času se vyvíjí. V chromozomu, který reprezentuje celý obvod, vystupuje modul jako jediný element. Součástí chromozomu se modul stává buď při inicializaci nebo mutací. Při těchto operacích modul představuje jednu z funkcí. Celá evoluce je navržena tak, že se střídá vývoj chromozomu jako celku s vývojem samostatného modulu. Systém samozřejmě může obsahovat modulů více.

Nedílnou součástí bylo experimentální ověření celého sytému na vybraných kombinačních obvodech. Zvolila jsem sudou paritu, řadicí sít, sčítačku a násobičku. V experimentech jsem se zaměřila i na zkoumání vlivu nastavení různých parametrů. U některých obvodů se metoda ukázala jako nevhodná. Tyto obvody měly společné znaky modulů. Všechny moduly měly více výstupů. Z prvu jsem se domnívala, že neúspěšnost metody souvisí s nezapojenými výstupy modulů. Další experimenty tuto domněnku ale vyvrátily.

Úspěšnost navrženého systému je závislá na řešeném problému. Pro evoluční návrh sudé parity byl poměrně úspěšný a došlo ke snížení počtu generací potřebných pro nalezení funkčního obvodu. Pro řadicí síť, sčítačku a násobičku se ukázala tato cesta jako nevhodná. To ovšem neznamená, že nelze nalézt jinou modularizační techniku. Pro další práci se nabízí ověřit řešení, o kterém jsem z počátku uvařovala, a to obměna ECGP, kde by se moduly tvořily dynamicky, ale pouze z právě aktivních uzlů.

Literatura

[1] Harding, S.; Miller, J. F.; Banzhaf, W.: Developments in Cartesian Genetic Programming: Self-modifying CGP. http://www.evolutioninmaterio.com/preprints/smcgpjournal.pdf.

[2] Harding, S.; Miller, J. F.; Banzhaf, W.: Self Modifying Cartesian genetic Programming: Parity. http://www.cs.mun.ca/~banzhaf/papers/smcgp_cec1.pdf.

[3] Harding, S.; Miller, J. F.; Banzhaf, W.: Self-Modifying Cartesian genetic Programming. http://www.cs.mun.ca/~banzhaf/papers/smcgp.pdf, 2007.

[4] Koza, J. R.: *Genetic Programming II: Automatic Discovery of Reusable Pragrams*. Bradford Book, 1994, iSBN 0-262-11189-6.

[5] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Bradford Book, 1994, iSBN 0-262-11170-5.

[6] Mařík, V.; Štěpánková, O.; Lažanský, J.; aj.: *Umělá inteligence (3)*. Academia, 2001, iSBN 80-200-0472-6.

[7] Mařík, V.; Štěpánková, O.; Lažanský, J.; aj.: *Umělá inteligence (4)*. Academia, 2003, iSBN 80-200-1044-0.

[8] Miller, J. F.; Harding, S. L.: GECCO 2010 Tutorial: Cartesian Genetic Programming. http://portal.acm.org/citation.cfm?id=1830924, 2010.

[9] Miller, J. F.; Jacob, D.; k. Vassilev, V.: Principles in the Evolutionary Design of Digital Circuits - Part 1. 2000.

[10] Miller, J. F.; Thomson, P.: Cartesian Genetic Programming. In *Proceedings of the 3rd European Conference on Genetic Programming*, Springer, 2000, s. 121–132.

[11] Schwarz, J.; Sekanina, L.: Aplikované evoluční algoritmy, 2006, studijní opora.

[12] Sekanina, L.; kol.: *Evoluční hardware*. Academia, 2009, iSBN 978-80-200-1729-1.

[13] Vašíček, Z.; Sekanina, L.: Evoluční návrh kombinačních obvodů. *Elektrorevue - www.elektrorevue.cz*, 2004, http://www.stud.fit.vutbr.cz/~xvasic11/doc/cl2004.pdf.

[14] Walker, J. A.; Miller, J. F.: Evolution and Acquision of Modules in Cartesian genetic Programming. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.9942&rep=rep1&type=pdf.

[15] Walker, J. A.; Miller, J. F.: Investigating the Performance of Module Acquisition in Cartesian Genetic Programming.
http://www.cartesiangp.co.uk/papers/2005/wmgecco2005.pdf, 2005.

[16] Walker, J. A.; Miller, J. F.: Embedded Cartesian Genetic Programming and the Lawnmower and Hierarchical - if - and - only - if Problems.
http://www.cartesiangp.co.uk/papers/2006/wmgecco2006.pdf, 2006.

[17] Walker, J. A.; Miller, J. F.; Cavill, R.: A Multi-chromosome Approach to Standard and Embedded Cartesian genetic Programming. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.9637&rep=rep1&type=pdf, 2006.

[18] WWW stránky: Nástroje pro kartézské genetické programování Tools4CGP [online].
http://www.fit.vutbr.cz/~vasicek/cgp/, 2008 [cit. 2011-05-07].

Seznam příloh

Dodatek A

Obsah přiloženého CD

- Diplomová práce v elektronické podobě
- Zdrojové texty diplomové práce pro LaTeX(adresář tex)
- Implementovaný systém (adresář prog)
  - Zdrojové soubory systému včetně Makefile a trénovacích dat (adresář src)
  - Spustitelný soubor pro systém Linux (adresář dist)
  - Programová dokumentace (adresář dok)
  - Soubor readme.txt

Dodatek B

Překlad a použití programu

Překlad

Program byl úspěšně přeložen pod systémy Linux (školní server merlin) a Windows (Home XP 32-bit). Pro překlad je nutný překladač g++.

V adresáři prog/src je k dispozici soubor Makefile. Příkazem `make` se do téže složky vytvoří spustitelný soubor modcgp. Příkazem `make clean` lze odstranit všechny soubory vytvořené překladem.

Použití

Program se spouští příkazem `./modcgp`. Při běhu programu je při změně fitness funkce na standardní výstup vypisována hodnota fitness a generace, ve které ke změně došlo. Po ukončení evoluce je na standardní výstup vypsán výsledný chromozom a příslušné moduly v zakódovaní pro program cgpviewer. Současně program vytvoří soubor result.chr s výsledným chromozomem, který lze použít pro nahrání do programu cgpviewer.

V adresáři prog/dist je uložen spustitelný soubor pro systém Linux (přeloženo na školním serveru merlin). Soubor spustí experiment 4-bitová sudá parita s funkční sadou AND, OR, NAND. Pro změnu typu experimentu či jeho nastavení je potřeba udělat následující kroky:

- Pro změnu typu obvodu je nutno v souboru modcgp.h změnit vstupní soubor. Vstupní soubory, se kterýma jsem prováděla experimenty, jsou uloženy v adresáři prog/src/test.
- Funkční sadu lze definovat na vyznačeném místě v souboru cgp.cpp ve funkci `fitness()`. Počty funkcí pro chromozom a modul jsou definovány v souboru modcgp.h. Počty funkcí pro chromozom a modul musí souhlasit s definovanou funkční sadou.
- Počet a nastavení modulů lze měnit na vyznačeném místě při inicializaci ve funkci `main()`.

Po jakékoli změně je třeba zdrojové soubory znova přeložit.