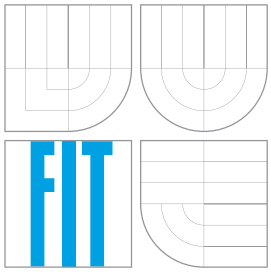
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOEVOLUCE V EVOLUČNÍM NÁVRHU OBVODŮ

COEVOLUTION IN EVOLUTIONARY CIRCUIT DESIGN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB VEŘMIŘOVSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAELA DRAHOŠOVÁ

BRNO 2016

Abstrakt

Tato práce se zabývá evolučním návrhem obvodů za pomoci kartézského genetického programování a jeho optimalizaci za pomoci koevoluce. Algoritmus koevolvuje fitness prediktory, které jsou optimalizovány pro populaci kandidátních obvodů. Práce popisuje teoretická východiska, zejména pak genetické programování, koevoluci v genetickém programování, návrh obvodů, a zabývá se návrhem využití koevoluce v evolučním návrhu kombinačních obvodů. Na základě tohoto návrhu je implementována aplikace, která umožňuje navrhovat a optimalizovat kombinační obvody. Funkčnost aplikace byla ověřena na pěti testovacích úlohách. Srovnání proběhlo mezi kartézským genetickým programováním s koevolucí a bez koevoluce. Poté řešení navržené pomocí evoluce bylo srovnáno s klasickými metodami návrhu. S použitím koevoluce se snížil počet evaluací obvodu během evoluce a v některých případech našla řešení, která mají lepší parametry (např. méně logických hradel, menší zpoždění), než řešení navržená konvenčně.

Abstract

This thesis deals with evolutionary design of the digital circuits performed by a cartesian genetic programing and optimization by a coevolution. Algorithm coevolves fitness predictors that are optimized for a population of candidate digital circuits. The thesis presents theoretical basis, especially genetic programming, coevolution in genetic programming, design of the digital circuits, and deals with possibilities of the utilization of the coevolution in the combinational circuit design. On the basis of this proposal, the application designing and optimizing logical circuits is implemented. Application functionality is verified in the five test tasks. The comparison between Cartesian genetic programming with and without coevolution is considered. Then logical circuits evolved using cartesian genetic programming with and without coevolution is compared with conventional design methods. Evolution using coevolution has reduced the number of evaluation of circuits during evolution in comparison with standard cartesian genetic programming without coevolution and in some cases is found solution with better parameters (i.e. less logical gates or less delay).

Klíčová slova

Koevoluce, kartézské genetické programování, číslicové obvody, logical effort, prediktory fitness.

Keywords

Coevolution, cartesian genetic programming, digital circuit, logical effort, fitness predictors.

Citace

Jakub Veřmiřovský: KOEVOLUCE V EVOLUČNÍM NÁVRHU OBVODŮ, diplomová práce, Brno, FIT VUT v Brně, 2016

KOEVOLUCE V EVOLUČNÍM NÁVRHU OBVODŮ

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michaely Drahošové. Další informace mi poskytl Ing. Vojtěch Mrázek. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Veřmiřovský
18. května 2016

Poděkování

Rád bych poděkoval Ing. Michaele Drahošové za odborné vedení a cenné rady a konzultace při vypracování této diplomové práce, a také Ing. Vojtěchu Mrázkovi za konzultace ohledně logical effort.

Obsah

Kapitola 1

Úvod

Ačkoliv výpočetní výkon každým rokem roste, stále existují problémy, které nelze řešit hrubou silou. Proto se stále objevují a vyvíjejí lepší heuristiky pro účelnější prohledávání stavového prostoru a rychlejší hledání řešení. Při rozvoji těchto heuristik se hledá také inspirace v přírodě, protože sama příroda byla schopna vyvinout složité systémy. Je tedy snaha formalizovat a implementovat paradigma živé přírody pro návrh nových algoritmů, postupů a metod. Jsou to například mravenčí kolonie, neuronové sítě, umělá inteligence inspirovaná uvažováním zvířat i lidí nebo také evoluční algoritmy. Evoluční algoritmy vychází z Darwinovy teorie vzniku druhů a ukazují se jako velmi výhodné pro řešení optimalizačních problémů či při vytváření elektrických obvodů [8]. Tato práce se zabývá návrhem přibližně počítajících kombinačních obvodů za použití genetického programování. Přibližné počítání je nové návrhové paradigma, které vzniklo jako reakce na zvyšující se nároky na výkon a energetickou účinnost výpočetních systémů [12]. Tento návrh využívá toho, že není vždy potřeba přesného funkčního chování nebo se použije tam, kde je rozeznání chyby obtížné. Jde o aplikace, kde neexistuje formální validace výstupu nebo lidské rozeznání není dostatečně přesné (například audio systémy) [17].

K návrhu přibližně počítajících obvodů bude použito kartézské genetické programování (CGP). To je druh genetického programování, ve kterém jsou kandidátní řešení reprezentována jako acyklické grafy. Tento přístup používá pro ohodnocení fitness funkce kandidátního řešení vyhodnocení obvodu pro každou variantu vstupní kombinace, která je uvedena v trénovací sadě a tudíž jde o výpočetně náročnou část algoritmu. Proto bude vedle populace kandidátních řešení souběžně vyvíjena i populace prediktorů fitness, která bude sloužit pro urychlení výpočtu fitness funkce každého kandidátního řešení. Cílem práce je navrhnout a implementovat program, ve kterém bude možné simulovat návrh kartézského genetického programování s koevolučním učením prediktorů (coCGP) nad zadanými úlohami a porovnat jeho přínosy oproti klasickému CGP. Budeme také zkoumat jaká je vhodná velikost pole prediktorů tak aby byl urychlen návrh obvodu. Vyhodnocení kvality získaných řešení, proběhne v porovnání s řešeními, které byly navrženy pomocí konvenčních metod návrhu.

1.1 Struktura práce

Druhá kapitola se zabývá vysvětlením základních pojmů a evolučních principů. V kapitole 2 jsou představeny evoluční algoritmy a jsou následovány genetickými algoritmy v části 2.1. Část 2.2 obsahuje genetické programování. Kartézské genetické programování se nachází v části 2.3. Část 2.4 se zabývá evolučním návrhem kombinačních obvodů, problematikou při-

bližného počítaní a logical effort. Principy koevolučních algoritmů se nachází v podkapitole 2.5 v jejichž podkapitolách je vysvětlena koevoluce prediktorů a koevoluce v kartézském genetickém programování. Návrhem systému se zabývá kapitola 3, ve které je řešen také návrh chromozomů jedinců a v části 3.1 je návrh trenérů pro koevoluci. Následující část 3.2 obsahuje návrh prediktorů fitness s podkapitolou 3.2.1 ohodnodnocení prediktoru fitness. Evoluce kandidátních řešení je navržena v části 3.3, ve které jsou v podkapitolách navrženy mutace genů 3.3.1, označení aktivních uzlů kandidátního řešení 3.3.2 a vyhodnocení fitness hodnoty kandidátních řešení v 3.3.3. Část 3.4 se zabývá koevolucí s rozdělením do podkapitol 3.4.1 evoluce prediktorů a 3.4.2 evoluce kandidátních řešení. Návrh paralelizace je v části 3.5, v 3.5.1 je návrh synchronizace vláken a struktura souboru se vstupními trénovacími daty je navržena v části 3.6. Kapitola 4 je věnována implementaci simulačního programu. V podkapitole 4.1 probrána paralelizace programu. Impelmentace množiny trenérů je vysvětlena v části 4.2. Dále v části 4.3 je vysvětlen způsob implementace fitness funkcí a následující část 4.4 se věnuje synchronizačním detailům mezi evolucemi. V části 4.5 jsou popsány funkce, které se věnují optimalizacím navrhovaného obvodu. Optimalizacím simulačního programu je věnována podkapitola 4.6 a v podkapitole 4.7 je porovnání se známými implementacemi kartézkého genetického programování. Části 4.8 a 4.9 se věnují výstupům programu resp. spuštěním programu. Experimentálním vyhodnocením se zabývá kapitola 5. V podkapitolách 5.1 resp. 5.2 je vysvětlen způsob hledání optimálních parametrů kartézkého genetického programování reps. coCGP. Podkapitoly 5.3 - 5.8 se věnují srovnáním CGP a coCGP a vyhodnocením jejich výsledků, které jsou posléze shrnuty v podkapitole 5.9.

Kapitola 2

Evoluční algoritmy

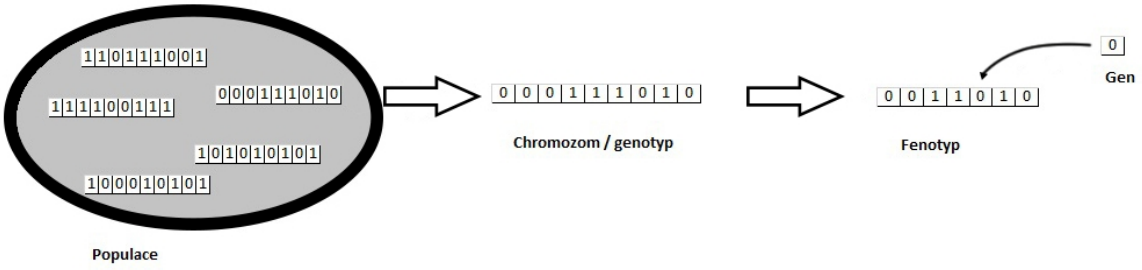
Evoluční algoritmy založené na Darwinově evoluční teorii představují nový, netradiční přístup k hledání optimálního nebo suboptimálního řešení složitých optimalizačních problémů, které nejsou řešitelné konvenčními přístupy. Pojem evoluční algoritmy vznikl až v 90. letech ačkoliv techniky, které pod evoluční algoritmy spadají, sahají historicky až k polovině 20. století [4]. Evoluční algoritmy byly z počátku využívány pouze k optimalizaci, ale časem našly své uplatnění i v návrhu systémů.

Dříve, než začneme se základním principem evolučních algoritmů, vysvětlíme významy pojmů, které jsou přebrány z biologie, ale význam nemusí být shodný [9]:

- **Gen** – nese informaci o základním stavebním bloku kandidátního řešení. Nabývá právě jedné hodnoty z konečné množiny nad danou abecedou
- **Chromozom, jedinec nebo genotyp** – reprezentace řešení ve stavovém prostoru. Řetězec je složen z genů.
- **Fenotyp** – část genotypu, kódující řešení (vztah genotyp a fenotyp zobrazen na obr. 2.1).
- **Populace** – konečná množina jedinců.
- **Fitness** – udává kvalitu kandidátního řešení.
- **Fitness funkce** – přiřazuje kandidátnímu řešení hodnotu fitness.
- **Evaluace** – výpočet hodnoty fitness jedince neboli jeho ohodnocení.
- **Selekce** – proces výběru jedinců k reprodukci.
- **Křížení, Mutace** – jsou rekombinační operátory pro produkci potomků z rodičů, které budou v práci použity.

Evoluční algoritmy mají mnoho různých implementací a použití, ale vyznačují se těmito společnými rysy [12]:

- Pracují s populací kandidátních řešení, která umožňuje paralelní přístup k prohledávání stavového prostoru.
- K vytváření nových potomků se používají biologií inspirované operátory.



Obrázek 2.1: Vztah populace ke chromozomu/genotypu, fenotypu a genu.

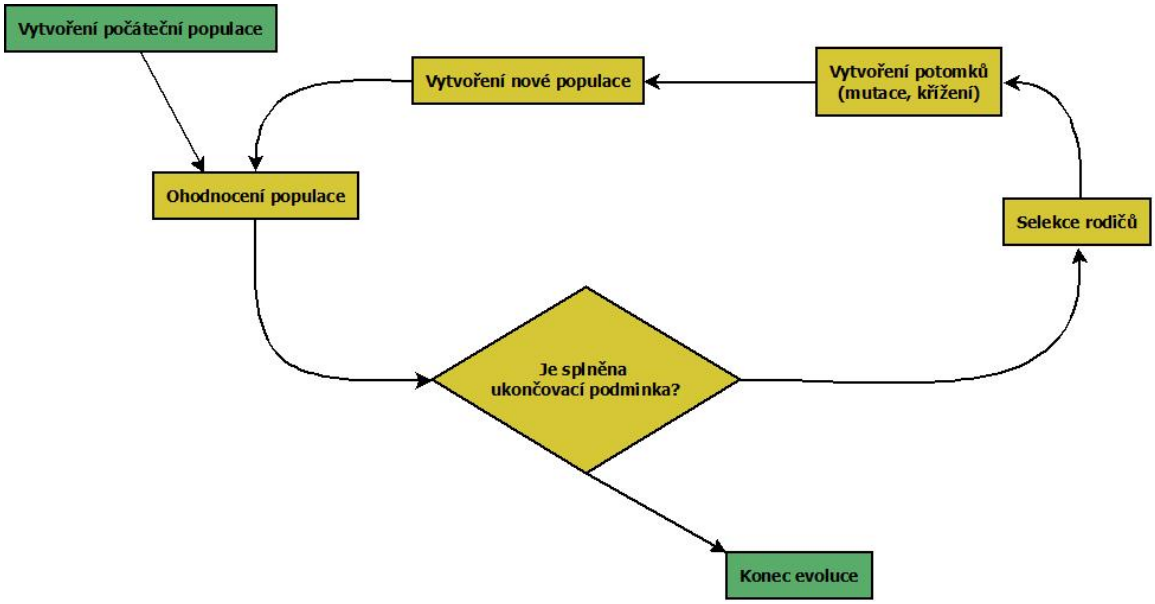
Evoluční algoritmy jsou charakteristické tím, že pracují s populací kandidátních řešení. Každé kandidátní řešení (jedinec) je zakódováno do genotypu. Jde o vektor hodnot (genů), který představuje jedno z možných řešení úlohy. Počáteční generaci je možné vytvořit buď náhodně a nebo za pomocí heuristiky. Po vytvoření počáteční populace můžeme začít prohledávat stavový prostor. To provádíme v iteracích. V každé iteraci je populace ohodnocena pomocí fitness funkce a je vybrána podmnožina jedinců, ze kterých se stávají rodiče. Z rodičů za pomocí genetických operátorů vytvoříme potomky. Z rodičů a potomků je následně sestavena nová generace. Tento proces se opakuje tak dlouho, dokud nejsou splněny ukončovací podmínky (maximální počet iterací, nalezení řešení, atd.).

Důležitou částí je takzvaná fitness, která udává kvalitu daného řešení, tzn. jak dobře řešení plní zadanou úlohu. Můžeme ji zjednodušeně definovat jako funkci:

$$f : G \to \Re, \tag{2.1}$$

kde G je množina všech genotypů (kandidátních řešení) a obor hodnot je reálné číslo, které je určeno podle chování zakódovaného daným genotypem. Pokud je cílem najít globální minimum (minimalizace argumentu) hledáme takové g* $\in$ G, pro které platí f(g*) $\leq$ f(g) pro všechna g $\in$ G. Naopak pokud hledáme globální maximum, pak hledáme argument, který splňuje f(g*) $\geq$ f(g) pro všechna g $\in$ G. Fitness funkce nemusí být spojitá, mít derivaci ani být úplně definovaná. Výpočet fitness funkce často bývá nejnáročnější částí algoritmu, tudíž je velkou výhodou pokud proběhne dostatečně rychle. Algoritmus funguje následujícím způsobem:

1. Vytvoření počáteční populace.
2. Ohodnocení populace.
3. Pokud je splněna ukončovací podmínka (nalezené řešení, maximální počet generací, atd.) ukonči algoritmus.
4. Vyber nejlépe ohodnocené rodiče a z nich vytvoř novou populaci potomků.
5. Vyber jedince do nové populace.
6. Zpět na bod 2.



Obrázek 2.2: Životní cyklus evolučního algoritmu.

2.1 Genetické algoritmy

Genetické algoritmy, které byly v sedmdesátých letech navrženy Johnem Hollandem, patří mezi základní stochastické optimalizační algoritmy s výraznými evolučními rysy. V současnosti je genetický algoritmus nejčastěji používaným evolučním optimalizačním algoritmem se širokou paletou aplikací od optimalizace vysoko multimodulárních funkcí přes kombinatorické a grafové problémy, až po aplikace zvané umělý život [8].

Základní varianta genetického algoritmu má stejné schéma jako je uvedeno výše u evolučního algoritmu. Kandidátní řešení je reprezentováno řetězcem konstantní délky. Počáteční populace je vygenerována náhodně. Celá populace je ohodnocena a z jedinců, kteří jsou vybráni selekčním algoritmem, se stanou rodiče. Jakmile jsou vybráni rodiče, můžeme z nich vytvořit novou generaci. Zde jsou genetické algoritmy taktéž inspirovány vývojem živých organismů. Nejběžněji používanými operátory pro tvorbu nových kandidátních řešení a pro práci s genotypy jsou mutace a křížení. V nové populaci se můžou vyskytovat jedinci z původní populace, ale pokud celou novou populaci utvoříme pouze z potomků, mluvíme o *překrývání populací*. *Elitismus* znamená, že nejlepší jedinec z předchozí populace se vždy dostane do nové populace.

2.1.1 Selekce

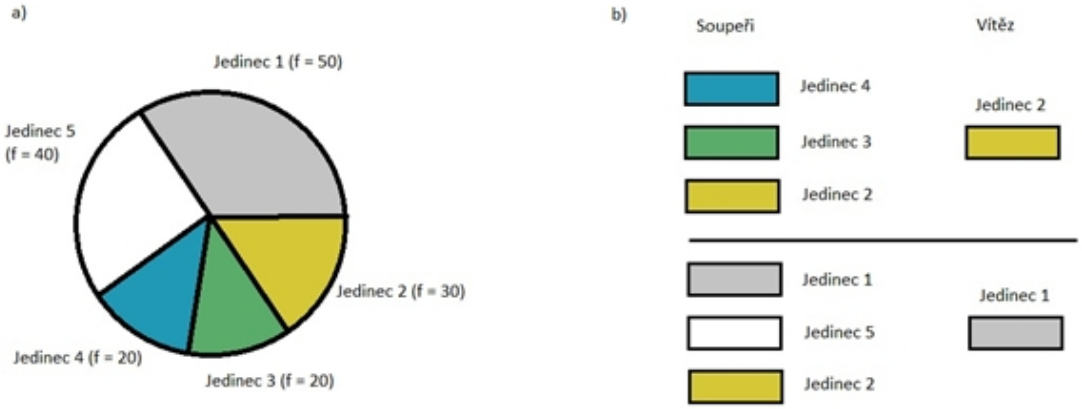
Selekce slouží k výběru jedinců k reprodukci. Tento výběr by měl být napodobením přirozeného výběru, jak je popsán v Darwinově teorii o původu druhů. To znamená, že zdatnější jedinci mají větší šanci přežít v daném prostředí a zplodit potomky.

Nejjednodušší způsob je prosté seřazení jedinců podle hodnoty fitness, kdy rodiči se stávají jedinci, kteří mají tuto hodnotu nejvyšší. Tento způsob má nevýhodu, že nezajišťuje dostatečnou různorodost populace a narůstá tak nebezpečí předčasné konvergence. Zřejmě nejrozšířenější formou implementace přirozeného výběru je použití takzvaného ruletového mechanizmu selekce obr. 2.3. Na rozdíl od klasické rulety, kde je každé číslo vybráno se stejnou pravděpodobností kvůli ekvivalentnímu obsahu přidělených výsečí, budou v tomto

případě favorizována individua s lepší hodnotou fitness funkce. Takovým jedincům je přirazena větší výseč, tudíž je zvýšena pravděpodobnost výběru, ale zároveň se nevylučuje výběr jedince s nižší hodnotou fitness funkce. Existuje několik běžně používaných způsobů určování velikosti výsečí pro jednotlivá individua. Rozšířený je mechanismus proporcionální selekce, kde má každý jedinec přiřazenou výseč, jejíž plocha je přímo úměrná ohodnocení jedince. Potom pravděpodobnost, že bude jedinec vybrán je definována vztahem:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}, i \in \{1, ..., N\}. \tag{2.2}$$

Dalším metodou selekce rodičů je tak zvaný turnajový výběr. Tato selekce funguje tak, že se do turnajového kola náhodně vyberou dva nebo více jedinců. Vítězem se stává jedinec, který má nejvyšší fitness hodnotu. Ten se poté stává rodičem. Turnaj opakujeme tolikrát, kolik rodičů potřebujeme zvolit. Kvůli náhodnému výběru jedinců z populace, se může stát, že vícero turnajů vyhraje tentýž jedinec.



Obrázek 2.3: Selekce za pomocí a) ruletového mechanismu, b) turnajové selekce.

2.1.2 Mutace

Operátor mutace pracuje pouze s jedním jedincem. Náhodně změní hodnotu jednoho, nebo více jeho genů. V případě binární reprezentace dojde k negaci určitého počtu bitů v chromozomu. V případě genů nabývajících celočíselných hodnot se používají různé strategie (například. nahrazení náhodně vygenerovaným číslem). Je nutno zařídit, aby hodnoty genu nenabývaly nelegálních (nedefinovaných) hodnot. Pravděpodobnost mutace p_m určuje s jakou pravděpodobností dojde k mutaci jednoho genu chromozomu. Tato pravděpodobnost je většinou zvolena malá (např $p_m = 0.1\%$). Při vyšších hodnotách by algoritmus nemusel konvergovat k nejlepšímu řešení, a naopak při hodnotách nižších by se mutace nemusela vůbec projevit. Tento operátor přispívá k hlubšímu prozkoumávání prohledávaného stavového prostoru, jelikož vytváří zcela nová kandidátní řešení, která nemusí při operátoru křížení vůbec vzniknout [8]. Operátor mutace můžeme tedy definovat následovně:

$$\alpha' = O_{mut}^{(i)}(\alpha), \tag{2.3}$$

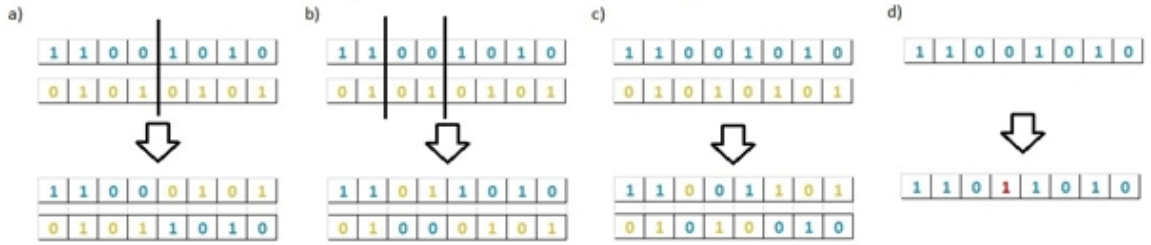
kde i je počet znaků v řetězci α, které operátor zmutuje.

2.1.3 Křížení

Během křížení dochází k výměně části genů mezi dvěma a více chromozomy. Mějme dva chromozomy α, β, operace křížení lze formálně definovat jako operátor zobrazení, který této dvojici chromozomů přiřadí dva nové chromozomy se stejnou délkou jako původní [8].

$$(\alpha', \beta') = O_{cross}(\alpha, \beta). \tag{2.4}$$

Různé způsoby realizace operátoru O_{cross} můžeme vidět na obrázku 2.4. U jednobodového křížení je náhodně vygenerován bod křížení, v němž dojde k prohození genů, které se nachází za bodem křížení. U vícebodového je princip podobný, jen je těchto bodů několik, což dovoluje promíchat genetický materiál mnohem důkladněji. V případě uniformního křížení je pro každý gen zvlášť rozhodnuto, zda dojde k výměně mezi rodiči. Křížení je považováno za základní princip, který zajišťuje, že genetický algoritmus vůbec funguje a operátor mutace je považován pouze za doplňkový [12].

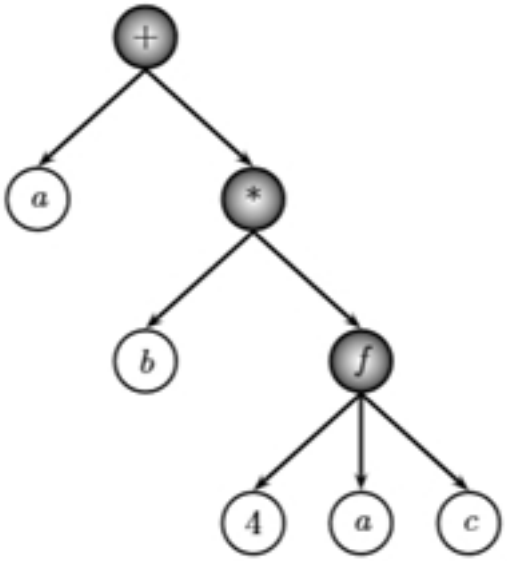


Obrázek 2.4: jednobodového (a), vícebodového (b), uniformního (c) křížení a mutace (d).

2.2 Genetické programování

Genetické programování (GP) rozšiřuje koncept evolučních algoritmů o prozkoumávání prostoru počítačových programů. To znamená, že chromozomy již nejsou řetězce pevné délky, ale hierarchicky strukturované počítačové programy, které po spuštění mohou představovat řešení daného problému. John Koza, který je považován za otce GP vytvořil metodologii, která umožňuje vytvářet počítačové programy pomocí podobných genetických operátorů, jaké jsme si uvedli v předchozí části s tím, že výsledkem evolučního procesu je program, který řeší (popřípadě přibližně řeší) konkrétní problém. Kandidátní program je nejčastěji tvořen stromovou datovou strukturou (viz obr. 2.5), kde listy jsou konstanty, vstupy programu nebo funkce bez parametrů. Uzly pak reprezentují funkce s určitým počtem vstupních parametrů. Množinu funkcí a hodnot je nutné specifikovat ještě před začátkem evolučního procesu.

Princip metody je podobný jako u genetických algoritmů. V prvním kroku je náhodně vygenerovaná počáteční populace, která je následně ohodnocena. Ohodnocení v GP spočívá ve spuštění programu, jehož jedinec reprezentuje se zadanými vstupy. Takto je získána odezva jedince. Fitness určujeme zpravidla tak, že máme definovanou množinu vstupních hodnot a ke každé vstupní hodnotě x_i je přiřazen očekávaný výstup y_i. Tuto dvojici nazýváme *trénovací vektor*. Fitness jedince lze pak definovat například jako odchylku mezi získanými a požadovanými hodnotami:



Obrázek 2.5: Stromová reprezentace kandidátního řešení: y = a + b * f(4, a, c).

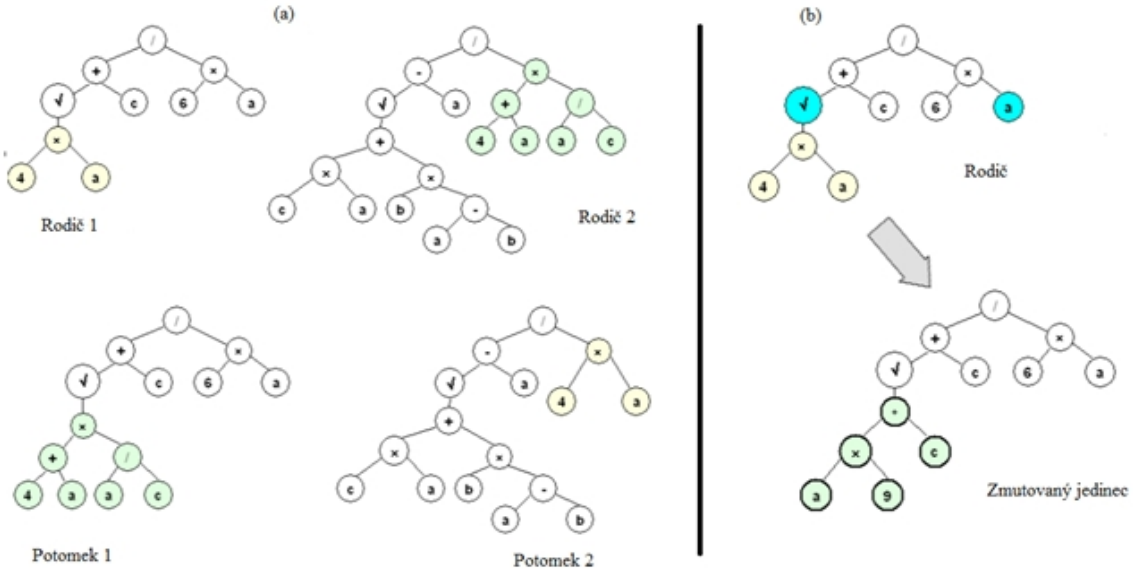
$$f = \sum_{i=1}^{N} (y_i^{GP} - Y_i)^2, \tag{2.5}$$

cílem evolučního procesu je minimalizovat tuto odchylku.Uvedený způsob výpočtu fitness je charakteristický pro tzv. symbolickou regresi, což je základní aplikace genetického programování. Cílem je najít program, který nejlépe aproximuje zadaná data v určitém intervalu. Také proces výběru a vytváření nového jedince probíhá podobně jako u genetických algoritmů. Zachovává použití operátoru křížení a mutace, které upravuje tak, aby pracovaly se stromovou reprezentací genotypu. Během křížení je náhodně vybrán jeden uzel u každého z rodičů. Tyto uzly společně s podstromy se poté prohodí, a tím vzniknou dva noví jedinci. Obdobně u operátoru mutace je náhodně vybrán uzel, a celý podstrom který na něj navazuje je nahrazen náhodně vygenerovanou strukturou (podstromem), jak je zobrazeno na obr 2.6.

2.3 Kartézské genetické programování

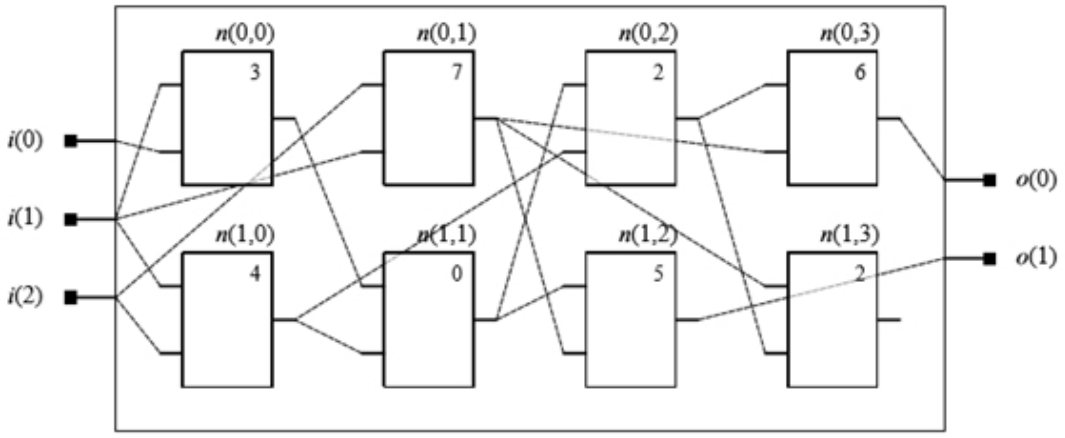
Kartézské genetické programování poprvé představil Julian Miller, který ho v roce 1999 poprvé publikoval. Jedná se o variantu genetického programování, kde jsou kandidátní řešení reprezentována pomocí obecných acyklických grafů.

Uzly jsou v něm uspořádány v pravoúhlé mřížce n_c (počet sloupců) x n_r (počet řádků) viz obrázek 2.7. Počet primárních vstupů obvodu n_in a počet primárních výstupů n_out je pevně určen na počátku evoluce. Každý uzel realizuje právě jednu funkci s n nargumenty, která je vybrána z množiny dostupných funkcí Γ. Vstupy uzlu mohou být napojeny buď na primární výstup, nebo na výstupy z předchozích L sloupců. Naopak se nesmí propojovat uzly stejného sloupce ani z následujících sloupců (zpětná vazba). Parametr L nám tedy ovlivňuje míru propojitelnosti obvodu. To znamená, že pokud zvolíme $L = 1$ můžeme



Obrázek 2.6: Operátory (a) křížení a (b) mutace jedinců.

propojovat pouze uzly sousedících sloupců a naopak pokud zvolíme $L = n_c$ pak bude možné uzly propojovat libovolně.



Obrázek 2.7: Schéma reprezentace programu v CGP [10].

Chromozom, který popisuje jedince v CGP je tvořen z A_{CGP} celočíselných hodnot kde:

$$A_{CGP} = n_r n_c (n_n + 1) + n_{out}. \tag{2.6}$$

Způsob kódování je následující: Každému primárnímu vstupu je přiřazen index z intervalu $< 0, n_{in} - 1 >$. Podobně ke všem výstupům uzlů jsou indexy přiřazeny po sloupcích s počáteční hodnotou $n_i n$ pro horní uzel, který je umístěný nejvíce nalevo. Každý uzel je kódován pomocí $n_n + 1$ celočíselných hodnot. Prvních n_n hodnot určuje indexy uzlů, které jsou připojeny na vstupy uvažovaného uzlu. Poslední hodnota určuje funkci uzlu. Na konci

chromozomu je $n_o ut$ hodnot, které určují indexy uzlů, které jsou připojeny na primární výstupy. Vlastností uvedeného kódování je, že fenotyp obvodu je variabilní a nemusí dosahovat délky chromozomu, která je konstantní. Uzlům, které nejsou obsaženy ve fenotypu, říkáme *nekódující uzly*.

Operátor křížení se ve standardním CGP nepoužívá, takže jediným operátorem je mutace. Funguje náhodným vybráním genu a následnou změnou jeho hodnoty. Parametrem operace mutace je počet genů, které má modifikovat. V genu je možné změnit jak funkci, kterou uzel vykonává, tak i změnit připojení jeho vstupů. Je vždy potřeba zajistit aby hodnota byla legální pro danou instanci CGP.

Schéma evoluce opět odpovídá obecné evoluční strategii. Počáteční populace je náhodně vygenerována. Po ohodnocení všech jedinců je vybrán nejlepší a za pomocí operátoru mutace z něj vygenerováni potomci. Nová generace se skládá z jednoho rodiče a potomků. Pokud existuje v populaci více „nejlepších" jedinců je vybrán ten, který nebyl rodičem v předchozí generaci. Tím je zajištěná genetická diverzita.

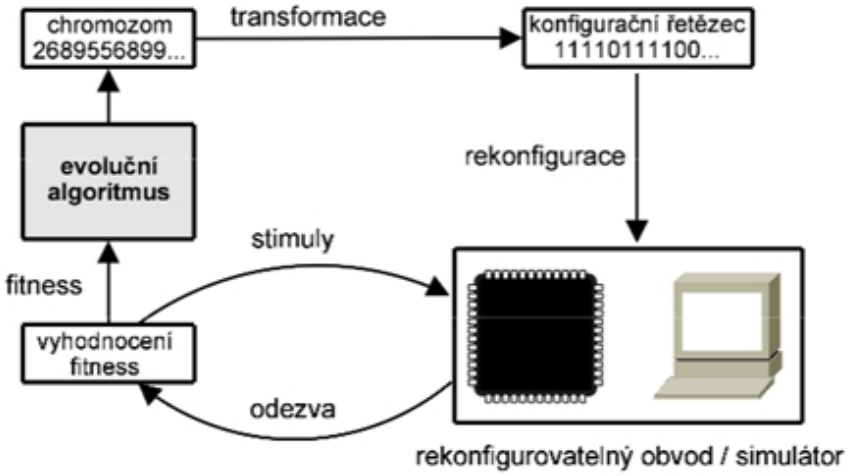
2.4 Evoluční návrh kombinačních obvodů

Evoluční algoritmy kromě hojného využití v optimalizačních úlohách našly uplatnění i v návrhu elektronických obvodů. Je to oblast, které říkáme *evolvable hardware* (*vyvíjející se obvody*). Je to technika, kterou lze využít jak ve fázi návrhu zařízení, tak při integraci přímo do cílového čipu a tak mohou vzniknout i adaptivní či sebeopravující se obvody. V prvním případě je cílem automaticky vygenerovat inovativní řešení v druhém případě je cílem zlepšit výkonnost systému pracujícího v měnícím se prostředí nebo opravit systém v případě poruchy. V kontextu evolvable hardware „inovativní řešení" znamená, že řešení vykazuje vyšší kvalitu než při konvenčním návrhu. Například bude zabírat menší plochu na čipu, bude dosahovat rychlejšího nebo přesnějšího výpočtu, mít menší spotřebu energie, atd.

Nicméně *evolvable hardware* nemůže v širokém použití konkurovat konvečnímu návrhu. Je to z důvodu škálovatelnosti evolučního návrhu [12]. Konvenční metody zvládají navrhovat i složité systémy, zatímco evoluční návrh s rostoucí složitostí zvětšuje velikost chromozomu a to implikuje rozsáhlé prohledávací prostory, které je obtížné efektivně prohledávat. Další problém se týká časové náročnosti evaluace fitness funkce. V případě evoluce kombinačních obvodů časová náročnost evaluace kandidátního řešení roste exponenciálně s počtem vstupů.

Evoluční návrh obvodů pracuje jako generátor nových řešení. Hlavní rozdíl oproti konvenčnímu návrhu spočívá v procesu „vygeneruj řešení a otestuj ho". Důležité je si uvědomit, že změna v kandidátním řešení je zcela náhodná, zatímco v konvenčním přístupu měníme obvod s určitou představou o výsledku obvykle zavedeným a dobře definovaným procesem.

Obr. 2.8 ukazuje princip evolučního návrhu obvodů s využitím rekonfigurovatelného zařízení. Chromozom je transformován na konfigurační řetězec, podle kterého je nakonfigurováno rekonfigurovatelné zařízení. Pokud EA pracuje přímo na úrovni konfiguračního řetězce, pak tento krok odpadá. V rámci evaluace kandidátního obvodu jsou generovány stimuly pro rekonfigurovatelné zařízení, měřeny jeho odezvy a vypočtena hodnota fitness. Místo rekonfigurovatelného zařízení lze použít simulátor, ale to může být o několik řádů pomalejší. V případě využití fyzického rekonfigurovatelného obvodu je nutné, aby bylo zajištěno, že náhodně vygenerovaná konfigurace nepoškodí obvod.



Obrázek 2.8: Princip vyvíjejících se obvodů [12].

2.4.1 Přibližně počítající kombinační obvody

Přibližné počítání je nové návrhové paradigma, které vzniklo jako reakce na zvyšující se nároky na výkon a energetickou účinnost výpočetních systémů [12]. Tento princip využívá toho, že není vždy potřeba přesného funkčního chování, jelikož mnoho systémů nepořebuje pracovat s absolutní přesností, nebo se používají tam, kde je rozeznání chyby obtížné. Jde o aplikace, ve kterých neexistuje formální validace výstupu, případně lidské vnímání není dostatečně přesné (například audio systémy) nebo o aplikace, ve kterých si jistou nepřesnost na úkor menší spotřeby nebo menšího zpoždění můžeme dovolit. Dále si uvedeme techniky, které se používají pro přibližný návrh obvodů.

- Over-scaling: Obvody jsou navrhovány jakožto plně fungující v normálním prostředí. Může však dojít ke snížení jejich příkonu snížením úrovně napětí, které je k obvodu přivedeno (například použitím slabšího zdroje napětí, nebo poklesu napětí, při kterém obvod může produkovat chyby). Je to podobné jako když výkon může být zvýšen přetaktováním obvodu. Časově indukované chyby mohou vznikat v případě, že na některých cestách bude obvod chybný, a indukují se pomocí zpětných vazeb.

- Funkcionální aproximace: Znamená, že obvod nebyl navržen jako plně fungující podle zadané specifikace. Nejjednodušší postup, jak se tohoto dosahuje, je snížením přesnosti výpočtu. V případě aritmetického výpočtu jsou zanedbány nejméně významné bity.

- Systematické návrhové metodologie: Vzhledem k tomu, že manuální přepracování návrhu není univerzální a efektivní metoda, pracuje se dále na vývoji systematické metody syntézy návrhu. Metoda SALSA se zabývá systematickým návrhem. Začíná na RT úrovni popisu s exaktní variantou obvodu a definovanými omezeními, které specifikují typ a množství chyb jaké může implementace obsahovat. Podrobnější popis metody je možné najít v článku [17].

- Na bázi chybové metriky: Výše uvedené metody jsou chybově orientovány v tom smyslu, že všechny logické optimalizace vedou k přibližnému řešení, které je omezeno předdefinovaným kritériem. Chybové kritérium může být vyjádřeno mnoha různými

metrikami jako například nejhorší chyba, průměrná chyba nebo pravděpodobnost chyby. Návrh je opakován dokud není dosaženo zadaného kritéria.

2.5 Logical effort

Během návrhu obvodů chceme kromě správné funkčnosti docílit také co nejlepších parametrů, jako například velké rychlosti či malého zpoždění. Nabízejí se pak následující otázky: „Která implementace jedné logické funkce má nejmenší zpoždění?“ nebo „Kolik stupňů produkuje nejmenší zpoždění?“. Odpověď není jednoznačná, jelikož někdy může přidání dalšího stupně obvod zrychlit. Těmito problémy se zabývá metoda *logical effort*. Pomocí této metody můžeme odhadnout zpoždění v *cmos* (polovodičovém) obvodu. Metoda specifikuje kolik stupňů má obvod mít nebo jaká je nejlepší velikosti tranzistoru pro logická hradla.

Metoda *logical effort* je založena na jednoduchém modelu zpoždění prostřednictvím jediného logického hradla. Model popisuje zpoždění způsobené kapacitní zátěží na logickém výstupu hradla a topologií hradla. Jednoduše řečeno, čím více výstup zatížíme, tím se zvětšuje zpoždění. Toto zpoždění závisí také na logické funkci hradla. Invertory, které jsou nejjednodušší logická hradla a mají toto parazitické zpoždění nejmenší, mají vhodné použití i jako zesilovače [13]. Tímto se lze vyrovnat s velkou zátěží výstupu [13]. Metoda *logical effort* tedy slouží pro zjednodušení analýzy zpoždění logického obvodu.

V prvním kroku modelace je nutno izolovat zpoždění dané výrobním procesem dané logické funkce. Zavedeme si jednotku zpoždění τ což bude zpoždění identického invertoru bez parazitického zpoždění. Zpoždění logického hradla d bude definováno následovně [13, 18]:

$$d_{abs} = d\tau. \tag{2.7}$$

Zpoždění vzniklé na logickém hradle má dvě části. Fixní část představuje parazitní zpoždění p, které je definováno logickou funkcí, a proporcionální část se nazývá *effort delay* (zpoždění úsilím) *f*. Celkové zpoždění v jednotkách τ tedy spočítáme:

$$d = f + p. \tag{2.8}$$

Effort delay f závisí na zatížení logického hradla a na vlastnostech logické hradla tuto zátěž řídit. Zavedeme nový symbol g, který značí vlastnosti logického hradla a elektrické úsilí které popisuje elektrické prostředí. Pak f je definováno:

$$f = g\frac{C_{out}}{C_{in}}. \tag{2.9}$$

Kde C_{out} je výstupní kapacita na logického hradla a C_{in} je vstupní kapacita logického hradla. Celkové zpoždění logického hradla v jednotkách τ je tedy definováno následovně:

$$d = g\frac{C_{out}}{C_{in}} + p. \tag{2.10}$$

V tabulce 2.1 je *logical effort* pro základní logická hradla podle [13]. Můžeme vidět, že *logical effort* má tendenci stoupat s počtem vstupů do logického hradla. Pro tato stejné logická hradla je v tabulce 2.2 parazitické zpoždění. To je zpoždění logického hradla při nulové zátěži.

Zpoždění roste kvadraticky s počtem tranzistorů n, a proto mnohdy může být výhodnější rozdělit velké logické hradlo ve dvě menší [13].

	1	2	3	4	n
Invertor	1				
NAND		4/3	5/3	6/3	(n + 2) + 2
NOR		5/3	7/3	9/3	(2n + 1) /3
XOR, XNOR		4, 4	6, 12, 6		

Tabulka 2.1: Logical effort podle počtu vstupů pro základní logická hradla [13].

	1	2	3	4	n
Invertor	1				
NAND		2	3	4	n
NOR		2	3	4	n

Tabulka 2.2: Parazitické zpoždění základních logických hradel [13].

2.6 Koevoluční algoritmy

Dosud uváděné evoluční algoritmy prováděly evoluci pouze jediné populace jedinců stejného druhu. V přírodě, ale probíhá evoluce více druhů souběžně a jedinci různých populací se navzájem ovlivňují. Jednoduchým příkladem může být dravec, který se snaží ulovit kořist. Postupně zlepšuje své lovecké vlastnosti a schopnosti. Působí tak selekčním tlakem na populaci kořisti. Kořist přirozeně také zlepšuje své dovednosti, aby přežila. Tato uzavřená smyčka tak urychluje vývoj jednotlivých druhů. Algoritmy využívající podobný souběžný vývoj nazýváme *koevoluční algoritmy*.

Koevoluční algoritmy na rozdíl od evolučních algoritmů nepoužívají fitness funkci ve formě zobrazení, jak jsme si uvedli $f : G \to \Re$, které přiřazuje hodnotu každému genotypu $g \in G$. Tento typ funkce, kdy mezi genotypy můžeme definovat relaci uspořádání, se nazývá objektivní fitness funkce. V koevolučních algoritmech se používá *subjektivní fitness funkce*, ve které jsou porovnáváni jedinci na základě interakce s ostatními jedinci ve stejné nebo oddělené populaci. Z toho plyne, že změna jednoho jedince může ovlivnit fitness hodnotu jiných jedinců. Proto je subjektivní fitness relativní a proměnlivá v čase podle složení populace. To znamená, že ohodnocení jedinců $g_1, g_2 \in G$ se v čase mění a je možné, že jedinec g_1 je lepší než jedinec g_2 v jednom okamžiku evoluce, a v průběhu evoluce se tato relace otočí.

Koevoluční algoritmy kromě populací používají ještě jeden typ kolekce jedinců, zvaný *archiv*. Do tohoto archivu se během generací postupně ukládají nejlepší jedinci, což znamená, že jej můžeme považovat za jakousi historii evolučního algoritmu. Ovšem vytvářet tuto historii není jeho hlavním úkolem. Archiv většinou ovlivňuje samotný evoluční proces. Například u některých metod se využívá při ohodnocení jedinců tak, že při ohodnocování nových generací se fitness určuje také podle toho, jak jedinci interagují s vybranými jedinci z archivu, který obsahuje pouze nejúspěšnější jedince. V úlohách, kde je použito více druhů populací, může mít každá svůj vlastní archiv. V těchto koevolučních metodách je tedy nutno zavést v algoritmu nový krok, který se postará o aktualizaci nebo o vložení nových jedinců ze současné populace do archivu, popřípadě některé z archivu odstranit.

Koevoluce je často použitá v úlohách, kde klasický evoluční přístup selhává. Jde o situace, kdy se snažíme nalézt optimální nastavení několika vzájemně působících komponent. V tomto případě by byl prohledávaný prostor kartézským součinem všech stavů všech komponent, tudíž obrovský. Další příklad využití koevoluce je v úlohách, kde nelze jednoznačně

určit fitness, jelikož je závislá na neustále se měnících okolnostech. Poslední težce řešitelnou úlohou je, když výpočet fitness jedince je výpočetně náročný z důvodů, že vstupní trénovací sada je příliš rozsáhlá. Tuto úlohu lze vyřešit, pokud se povede najít takovou podmnožinu ze vstupní trénovací sady, aby fitness šla dostatečně přesně aproximovat.

2.6.1 Koevoluce preditkorů fitness

Jedním ze způsobů, jak řešit problém velkého množství případů fitness je použití koevolučního algoritmu k *predikci fitness*. Pojmem *predikce fitness* nazýváme postup, kdy při evolučním algoritmu nepočítáme přesnou fitness jedinců, ale pouze ji přibližně odhadujeme technikami, které se samy vyvíjí během evolučního cyklu. V tomto smyslu konvenční algoritmus, který hledá optimální řešení s*, můžeme definovat takto:

$$s* = argmax_{s \in S} fitness(s), \tag{2.11}$$

kde S je množina všech možných kandidátních řešení a *fitness(s)* je fitness hodnota aktuálního řešení *s*. V koevoluci prediktorů fitness zaměníme všechny vyhodnocení fitness funkce vyhodnocením fitness prediktoru *p*. Pak optimální řešení *s** bude definováno následovně:

$$s* = argmax_{s \in S} p(s). \tag{2.12}$$

Při koevoluci fitness prediktorů v druhé populaci se vyvíjí fitness prediktory tak, aby fitness byla co nejpřesnější pro současnou generaci. Archiv budeme používat pro trénování *fitness prediktorů* a budeme jej nazývat množina fitness trenérů. Na jejich základě můžeme určit, jak se prediktory blíží požadované fitness. Optimální řešení s* pro každou populaci podle článku [11] definujeme takto:

$$s* = argmax_{s \in S} p_{best}(s), \tag{2.13}$$

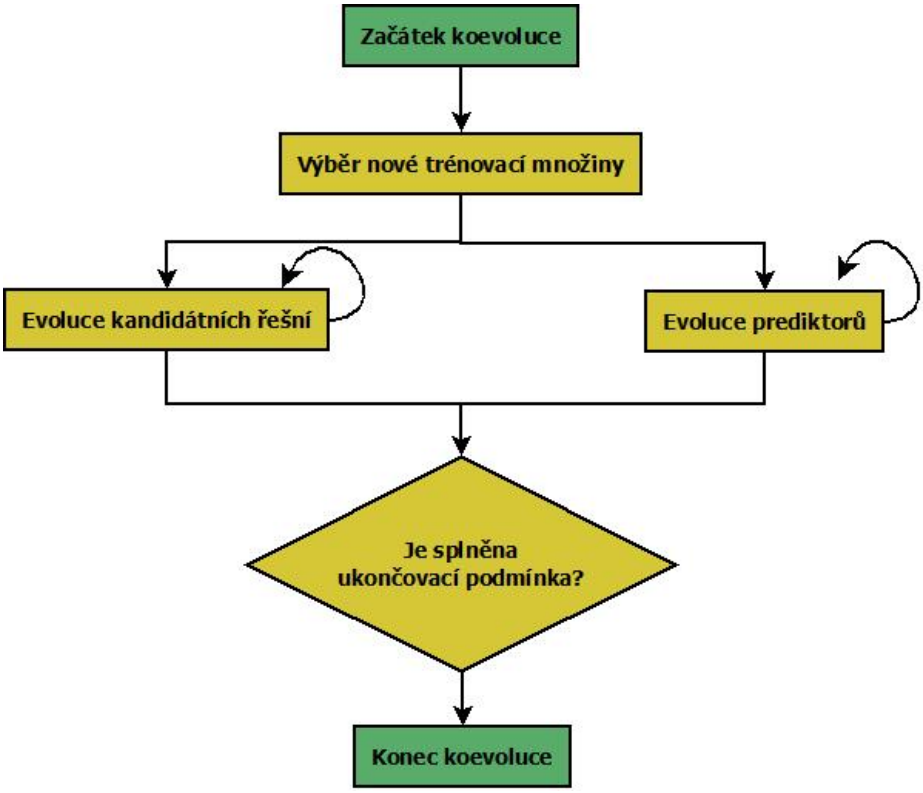
$$s* = argmax_{s \in S} \frac{1}{N} \sum_{p \in P_{cur}} (p(s) - \bar{p(s)})^2, \tag{2.14}$$

$$s* = argmin_{s \in S} \frac{1}{N} \sum_{t \in T_{cur}} |fitness(t) - p(t)|, \tag{2.15}$$

kde S je množina všech možných řešení, S_c je současná generace, P je množina všech fitness prediktorů, P_{cur} je současná generace prediktorů, T_{cur} je současná populace fitness trainers, p_{best} je nejlépe ohodnocený prediktor a p(s) je průměrná predikovaná fitness pro řešení s mezi současnými prediktory.

Populace řešení se vyvíjí podle nejlepšího fitness prediktoru p_{best}. Do množiny fitness trenérů jsou vybírána nejlepší řešení z populace řešení. Děje se tak vždy při objevení nového nejlepšího řešení, nebo periodicky. Populace prediktorů je vyvíjena tak, aby minimalizovala rozdíl mezi přesnou fitness jedince a predikovanou fitness.

Algoritmus je zobrazen na obr. 2.9. Na začátku se náhodně vygenerují populace řešení i prediktorů. Populace řešení je vyhodnocena přesnou fitness a nejlépe ohodnocení jedinci jsou vložení do množiny trainers. Pomocí této množiny je již řešena evaluace populace prediktorů. Nyní již populace řešení k ohodnocení používá nejlépe ohodnocený prediktor p_{best}. Obě populace jsou tedy vyvíjeny současně a algoritmus končí splněním ukončovací podmínky.



Obrázek 2.9: Základní životní cyklus koevolučního algoritmu.

2.6.2 Koevoluce v kartézském genetickém programování

Evaluace fitness funkce je v kartézském genetickém programování většinou nejnáročnější částí. Z důvodu nutnosti testování všech možných vstupních kombinací je vhodné v CGP tuto část algoritmu optimalizovat pro větší použitelnost. V článku [6] bylo ukázáno, že s využitím koevoluce v CGP lze čas potřebný k nalezení řešení zkrátit 2.02-5.45krát oproti standardnímu CGP v úloze symbolické regrese. Základní princip koevoluce v CGP je znázorněn na obr. 2.10.

Dříve než vysvětlíme průběh algoritmu, vysvětlíme pojmy, které se na obr. 2.10 vyskytují:

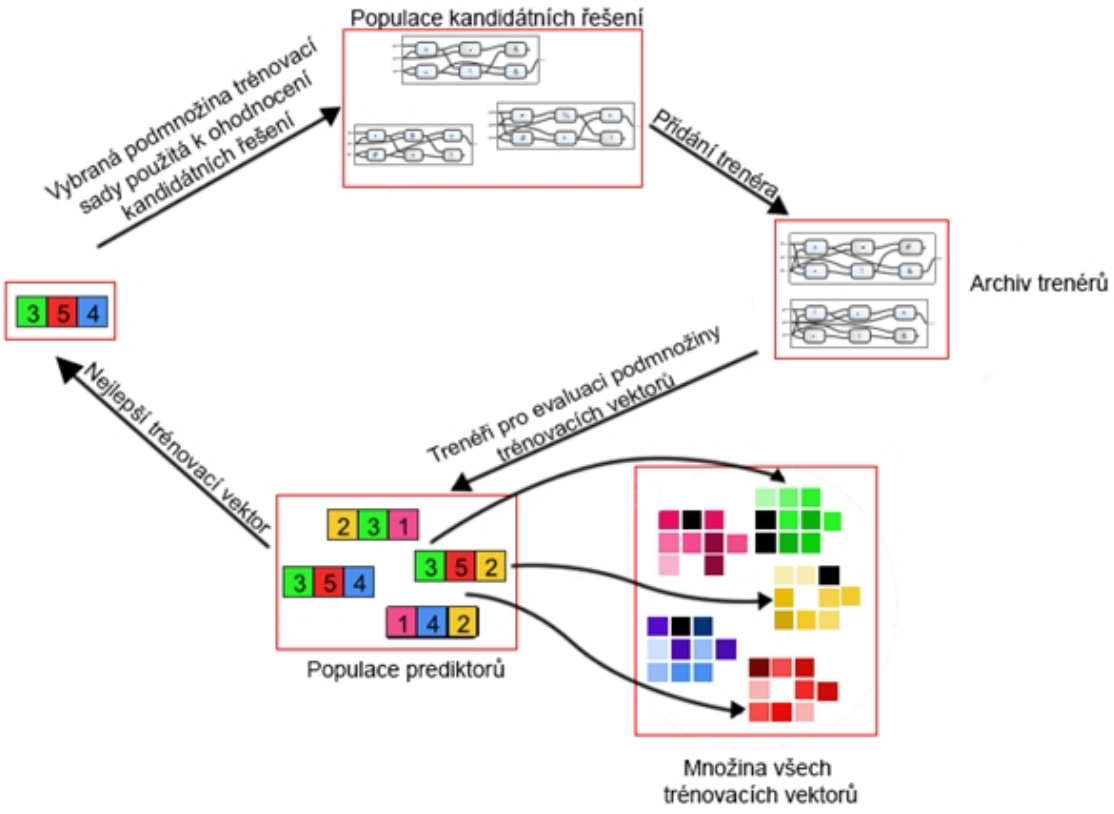
- Populace kandidátních řešení: Zde se hledá řešení zadané úlohy. V našem případě se řešení hledá pomocí kartézského genetického programování. K ohodnocení jedinců se nepoužívá celá trénovací sada, ale pomocí prediktorů je vybrána pouze malá reprezentativní část, čímž se dosahuje rychlejšího ohodnocení fitness.

- Populace prediktorů fitness: Tato populace slouží k výběru malé reprezentativní podmnožiny z celé trénovací sady. Tato část by měla být dostatečně spolehlivá pro odhad skutečné fitness. Fenotyp prediktorů fitness se tedy sestává pouze z ukazatelů na trénovací vektory z celé trénovací sady. Evoluce se provádí pomocí genetického algoritmu. Ohodnocení kvality prediktorů probíhá na základě trenérů uložených v archivu. Pro každý prediktor fitness je fitness funkce vyhodnocena jako odchylka mezi skutečnou fitness a fitness odhadnutou pomocí prediktoru.

- Archiv trenérů: Z populace kandidátních řešení je periodicky vybírán nejlépe ohodno-

cený jedinec, který je vložen do archivu. Jedince z tohoto archivu nazýváme trenéty, přičemž tito trenéři slouží k ohodnocení prediktorů fitness. Archiv je početně omezená množina, kde maximální počet trenérů je definován jako parametr evoluce. Proto se při vkládání nového jedince odstraní nejstarší jedinec, který se v archivu nachází. To znamená, že archiv funguje podobně jako kruhový seznam. Pro zajištění různorodosti se kromě nejlepších kandidátních řešení dané generace, vkládají i náhodně vygenerovaní jedinci. Tím je zajištěna diverzita archivu.

Evoluce kandidátních řešení a evoluce prediktorů běží souběžně vedle sebe. Na začátku algoritmu se náhodně vygenerují populace prediktorů a kandidátních řešení, dále je náhodně vygenerován archiv trenérů. Všechny populace ohodnotíme. Jako počáteční prediktor pro evaluaci kandidátních řešení se vybere nejlépe ohodnocený prediktor z populace prediktorů a spustí se evoluční process pro obě populace. Vzájemná rychlost evaluací by měla být nastavena tak, aby populace kandidátních řešení byla řádově rychlejší. Přesněji řečeno na jednu generaci prediktorů by měly být vyhodnoceny desítky až stovky generací kandidátních řešení, protože příliš časté změny prediktorů by vedly spíše k náhodnému prohledávání stavového prostoru. Pokud je objeveno nové lepší kandidátní řešení, je uloženo do archivu trenérů odkud je používá populace prediktorů ke svému trénovaní. Během evaluací kandidátních řešení se kontroluje, zda predikovaná fitness dosáhla požadované úrovně a další evaluace probíhá na základě celé trénovací sady, což znamená, že již bude počítaná přesná fitness. Při nalezení řešení, pro které přesná fitness spadá do intervalu povolené odchylky, je algoritmus ukončen s úspěchem nalezení řešení. Pokud je dosaženo jiné ukončovací podmínky, algoritmus končí neúspěchem.



Obrázek 2.10: Schéma koevolučního algoritmu s využitím CGP.

Kapitola 3

Návrh

Tato kapitola se zabývá návrhem systému, který bude umožňovat řešení návrhu kombinačních obvodů pomocí kartézského genetického programování a koevoluce. Použití koevoluce jak bylo popsáno v podkapitole 2.6.2, může v porovnání s klasickým kartézským genetickým programováním výrazně urychlit nalezení řešení.

V CGP je výpočet fitness výpočetně náročný a počítá se přes množinu případů fitness. Tento případ fitness reprezentuje jednu situaci, ve které se navrhované řešení může nacházet, a pro tuto situaci chceme kandidátní řešení ohodnotit. Případ fitness je tedy reprezentován primárními vstupy programu a definuje k těmto vstupům primární výstupy očekáváné od příslušného řešení. Strukturu, která obsahuje vstupy a k nim očekávané výstupy od funkčního řešení, nazýváme trénovací sada. Zrychlení se dosahuje tím, že není vyhodnocována celá tato trénovací sada, ale pouze její podmnožina. Zásadní je právě rozhodnutí jak velká tato podmnožina má být, aby byla dostatečně reprezentativní pro ohodnocení kandidátních řešení a také které případy podmnožina bude obsahovat. Jak bylo popsáno v podkapitole 2.5.1. druhý problém byl zautomatizován a řeší jej evoluce prediktorů.

Evoluce prediktoru fitness probíhá na principu genetického algoritmu. Prediktor je reprezentován jako vektor ukazatelů na případy fitness v trénovací sadě. Zjistit vhodnou délku prediktoru, aby byl co nejmenší a zároveň aby co nejlépe reprezentoval celý datový prostor je již experimentální úlohou, která nezávisí na evoluci samotné, ale je vázána ke každé úloze.

V navrhovaném systému budou probíhat dvě různé evoluce, a to za použití různých evolučních algoritmů, tudíž je nutné navrhnout reprezentaci jak pro evoluci prediktorů fitness, která probíhá pomocí genetického algoritmu, tak i pro evoluci kandidátních řešení, ve kterých se používá kartézské genetické programování.

Genotyp prediktorů fitness je složen pouze z celočíselných hodnot, které slouží jako ukazatele do množiny trénovacích dat. Délka genotypu je známa při spuštění programu a během evoluce je neměnná.

Genotyp jedince kartézského genetického programování je také vektor, který je složen z celočíselných hodnot kódujících acyklický orientovaný graf. Skutečnost, že rozměry mřížky uzlů jsou také předem známy, usnadňuje zakódování struktury genů do jejich genotypu. Všechny uzly mají přiděleny indexy, které korespondují s jejich pozicí v mřížce. V genotypu jsou tyto uzly uspořádány podle pořadí postupně za sebou a každý obsahuje $n_{in} + 1$ genů. Každý z uzlů obsahuje tři hodnoty:

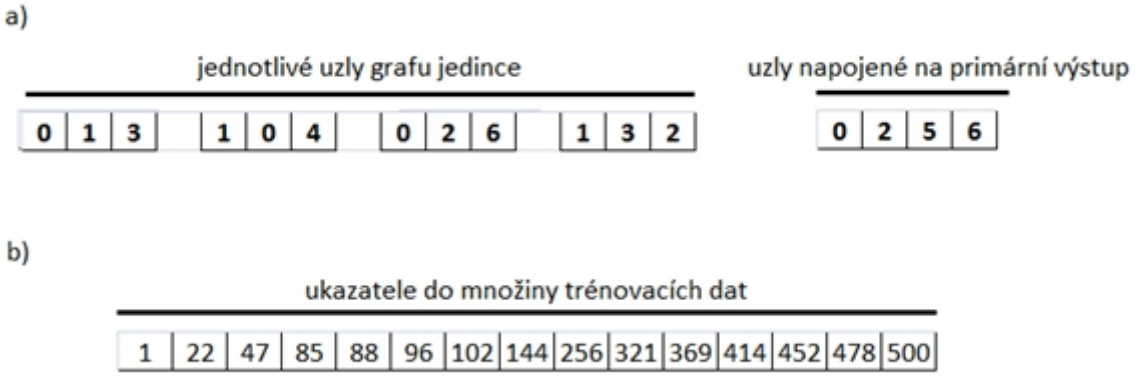
- index uzlu připojeného na první vstup uzlu,
- index uzlu připojeného na druhý vstup uzlu,

- číselné označení logické funkce, kterou uzel nad vstupy vyčísluje.

Poslední uzel obsahuje n_{out} geny, které jsou složeny z indexů, jež značí uzly, které jsou vyvedeny na primární výstupy.

Jelikož fenotyp řešení neobsahuje všechny uzly genotypu, je vhodné si ke každému uzlu přiřadit informaci o tom, zda je aktivní. Pomocí této informace jsme schopni urychlit výpočet fitness, neboť se nemusí počítat funkční hodnota každého uzlu, ale pouze těch, které jsou obsaženy ve fenotypu jedince.

Struktury obou chromozomů jsou zobrazeny na obrázku 3.1.



Obrázek 3.1: Zakódování chromozomů kandidátního řešení (a), prediktorů (b).

3.1 Trenéři

Trenéři jsou množina, která obsahuje kandidátní řešení ohodnocené skutečnou fitness hodnotou. Trenéři se používají pro ohodnocení fitness prediktorů. Množina trenérů musí obsahovat kandidátní řešení s různou fitness, aby bylo možné prediktory co nejpřesněji ohodnotit. Proto je množina trenérů rozdělena na dvě části. První část se plní při evoluci kandidátních řešení tak, že při evoluci se vybírají vždy nejlepší, ale vždy různá, kandidátní řešení. V druhé části se kvůli zajištění rozmanitosti množiny trenérů obměňují náhodně vygenerovaná kandidátní řešení. Množina trenérů je sdílená evolucí kandidátních řešení s evolucí prediktorů fitness.

3.2 Prediktory fitness

Evoluce prediktorů je nutná z důvodů adaptace prediktorů na řešený problém v průběhu řešení. Prediktor fitness je podmnožina všech trénovacích vektorů z trénovací množiny. Chromozom prediktoru je zobrazen na obrázku 3.1 a jak bylo zmíněno v kapitole 3 obsahuje pouze předem definovaný počet ukazatelů do trénovací množiny a zároveň žádné dva geny nemají stejnou hodnotu. Důležitou částí evoluce prediktorů je vyhodnocení jejich fitness, kterým se zabývá podkapitola 3.3.1.

3.2.1 Ohodnocení prediktoru

Ohodnocení fitness prediktoru lze vypočítat dvěma způsoby. V prvním případě lze spočítat absolutní odchylku řešení ohodnocených skutečnou fitness a řešení ohodnocených predikto-

#	Funkce	Popis
1	x_1	NOT
2	$x_1 \wedge x_2$	AND
3	$x_1 \bar{\wedge} x_2$	NAND
4	$x_1 \vee x_2$	OR
3	$x_1 \bar{\vee} x_2$	NOR
3	$x_1 \oplus x_2$	XOR
3	$x_1 \bar{\oplus} x_2$	NXOR

Tabulka 3.1: Seznam všech logických funkcí.

rem. Takto dostaneme vztah definovaný následovně:

$$f(p) = \frac{1}{u} \sum_{u=1}^{u} |f_{exact}(s(i)) - f_{predicted}(s(i))|, \tag{3.1}$$

kde *f(p)* je fitness prediktoru *p*, *u* je počet trenérů, f_{exact} je skutečná fitness řešení *s(i)*, a $f_{predicted}$ je fitness řešení *s(i)* predikovaná prediktorem *p*.

Dalším způsobem jak určit fitness prediktoru je pomocí přidělování bodů za výsledek predikované fitness, když odpovídá skutečné fitness kandidátního řešení. Fitness funkci pak definujeme takto:

$$f(p) = \sum_{i=1}^{o} g(s(i)), \tag{3.2}$$

$$g(s(i)) = \begin{cases} 0 & pro \quad |f_{exact}(s(i)) - f_{predicted}(s(i))| \geq \sigma \\ 1 & pro \quad |f_{exact}(s(i)) - f_{predicted}(s(i))| \leq \sigma \\ , \end{cases} \tag{3.3}$$

kde σ je povolená odchylka predikce fitness (parametr evoluce) a *o* je počet trenérů.

Prediktor, který dosáhne nejlepšího ohodnocení je používán pro predikci fitness při evoluci kandidátních řešení. To znamená, že evoluce prediktorů při každé generaci vybírá nového nejlepšího jedince a přepisuje jej, zatímco evoluce kandidátních řešení jej pouze používá, ale nemění.

3.3 Kandidátní řešení

Jedinci populace kandidátních řešení jsou kombinační obvody. To znamená, že jednotlivé uzly kartézské mřížky budou představovat logická hradla. Velikost mřížky v kartézském genetickém programování nelze určit nebo doporučit, dokud neznáme problém, který řešíme. Budeme uvažovat uzly (logická hradla), které mají pouze dva vstupy a jeden výstup, nebo u invertoru pouze jeden vstup a jeden výstup. Počet vstupů i výstupů je konfigurovatelný podle potřeb zadané úlohy. Logické funkce, kterých můžou uzly kandidátního řešení nabývat, můžeme vidět v tabulce 3.1.

3.3.1 Mutace genů

V kartézském genetickém programování vzniká nová generace jedinců pouze pomocí operátoru mutace uplatněného na genotypu rodiče. Během mutace je z definovaného rozsahu

(vstupní parametr) zvoleno náhodné číslo, udávající počet genů, které se mají zmutovat. Poté se náhodně vygenerují indexy genů, v nichž dojde k mutaci. Jelikož geny v genotypu nemají stejnou funkci a můžeme je rozdělit podle typu a liší se i způsob mutace těchto genů. Rozlišujeme tyto tři druhy mutace:

- Geny pro vstup do uzlu: Při mutaci genů, které kódují vstupy do jednotlivých uzlů, je potřeba nejprve stanovit povolené hodnoty, kterých gen pro daný uzel může nabývat (tzn. vypočítat indexy uzlů na které je možné daný vstup připojit). To je závislé na hodnotě parametru L-back a také umístění mutovaného uzlu v mřížce. Ve výběru je nutno uvažovat také primární vstupy. Následně je náhodně vygenerována hodnota z tohoto rozsahu a gen je změněn na tuto hodnotu.
- Geny funkce uzlu: Geny, jež kódují index logické funkce uzlu, kterou uzel vykonává, je náhodně vygenerována nová hodnota indexu funkce.
- Primární výstupy: Tyto geny nesou informaci, který uzel je napojen na primární výstup. Mutací se gen mění na náhodnou hodnotu indexu libovolného uzlu.

3.3.2 Označení aktivních uzlů

Při vyčíslování primárních výstupů jedince pro různé kombinace vstupních parametrů v kartézském genetickém programování se postupuje způsobem postupného vyčíslování uzlů v mřížce. Jelikož mřížka obsahuje mnoho neaktivních uzlů (nejsou obsaženy ve fenotypu), je možné zkrátit čas potřebný k výpočtu, přeskočením vyčíslovaní těchto neaktivních uzlů. Z tohoto důvodu se před první kombinaci vstupních parametrů označí uzly, které se podílí na výstupu. Algoritmus, který se stará o tuto filtraci aktivních uzlů funguje tak, že postupuje zpětně obvodem – od výstupů ke vstupům. Při inicializaci se do množiny aktivních uzlů vloží pouze ty uzly, které jsou přímo napojeny na primární výstup obvodu. Poté jsou uzly z množiny postupně vybírány, označeny jako aktivní a do této množiny vloženy ty uzly, které jsou napojeny na vstup tohoto aktivního uzlu. Pokud je vstupem uzlu primární vstup, algoritmus pokračuje dále bez vložení vstupu do množiny aktivních uzlů. Algoritmus končí ve chvíli, kdy se v množině aktivních uzlů nenachází žádný uzel.Ukázku rozdílného fenotypu od genotypu můžeme vidět na obrázku 3.2.

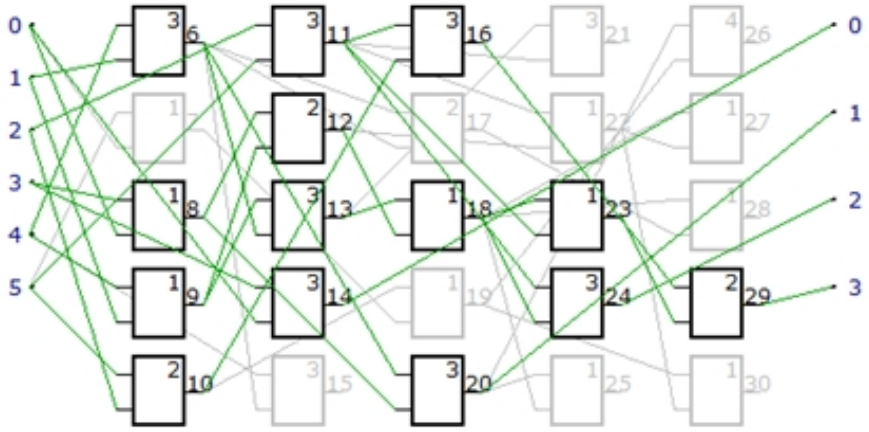
3.3.3 Vyhodnocení fitness kandidátních řešení

Během posuzování kvality kandidátního řešení je nejdůležitějším faktorem, jestli jeho odezva na zadané vstupy odpovídá referenčním výstupům k těmto vstupům. Jedna z možností počítání fitness je jako skóre počtu správného vyčíslení pro zadané vstupní kombinace. Výpočet skóre pak definujeme jako:

$$f(p) = \sum_{i=1}^{n} h(g(x_i)), \quad kde \tag{3.4}$$

$$h(g(x_i)) = \begin{cases} 0 & pro \quad |y_i - g(x_i)| \geq \epsilon \\ 1 & pro \quad |y_i - g(x_i)| \leq \epsilon, \end{cases} \tag{3.5}$$

kde ϵ je uživatelem definovaná maximální povolená odchylka. Fitness hodnota tedy udává počet správných hodnot podle referenčního souboru.



Obrázek 3.2: Genotyp vs. fenotyp (bez zašedlých uzlů) kandidátního řešení CGP.

Další možností výpočtu fitness je jako aritmetická odchylka. Takto definovaná fitness může být vhodná pro operace, které mají pro některé výstupy větší váhu než výstupy jiné. Jako bit s největší váhou je brán LSB (left significant bit) a naopak hodnotu nejnižší má RSB (right significant bit). Pro lepší představu, můžeme uvažovat, že binární číslo převedeme na desítkové a počítáme stejně, jako je uvedeno ve vzorci 3.5.

Toto ohodnocení, ale rozšíříme o ohodnocení podle zpoždění, které je uvedeno v podkapitole 2.5.2. V tomto rozšíření budeme postupovat tak, že jedince ohodnocujeme pouze podle skóre v první části a pokud toto ohodnocení dosáhne zvolené odchylky, poté ohodnocení upřesňujeme pomocí zpoždění. Samozřejmě jde o to, aby zpoždění bylo co nejmenší. Takto definovaná fitness pak vypadá následovně:

$$f(g) = (\sum_{i=1}^{n} h(g(x_i))) + \frac{1}{\sum_{i \in out} f_{logicaleffort}(i)}. \tag{3.6}$$

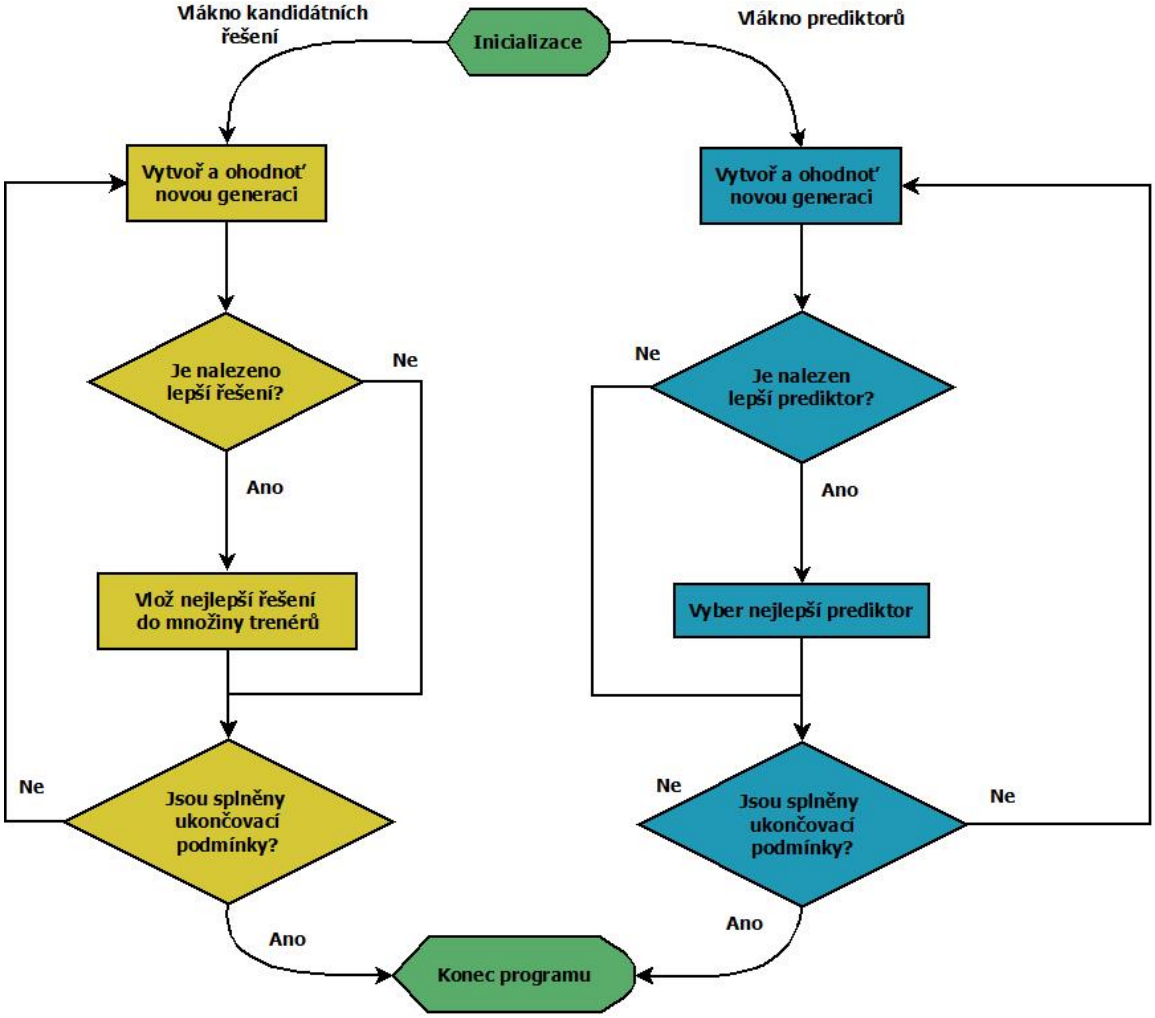
3.4 Koevoluce

Koevoluce se skládá ze dvou dílčích evolucí. Evoluce populace kandidátních řešení probíhá pomocí kartézského genetického programování v jednom vlákně, zatímco v druhém vlákně je vyvíjena evoluce populace prediktorů. Koevoluce je tedy vývoj účelných vlastností na základě změny vlastností jiné populace tak, že si populace mezi sebou vyměňují jedince k ohodnocení. Jedinci sloužící k ohodnocení druhé populace se předávají pomocí proměnných, které se sdílejí mezi vlákny. Noví trenéři se obměňují tak, jak je vysvětleno v části 3.1.

Na počátku koevoluce je potřeba nejprve inicializovat všechny populace, množiny a proměnné, které se jakkoliv podílejí na koevoluci. Podle vstupních parametrů koevoluce se náhodně vygenerují trenéři a populace kandidátních řešení. Poté se náhodně vygeneruje populace prediktorů generováním náhodných čísel v rozsahu referenčních dat a ohodnotí se pomocí trenérů. Nejlepší prediktor je pak vybrán pro ohodnocení kandidátních řešení.

Je potřeba zajistit vyloučení vzájemného přístupu ke sdíleným zdrojům mezi vlákny, aby nedošlo k nekonzistenci v populaci. K té může dojít například tak, že během ohodnocování kandidátních řešení bude prediktor vyměněn a tudíž můžou být jedinci ohodnoceni a porovnáváni různou fitness funkci v jedné generaci. Taková situace by mohla vést k tomu,

že řešení které je horší než ostatní může nabýt vyšší fitness hodnoty než ostatní, a být tak vybráno jako nejlepší.



Obrázek 3.3: Vývojový diagram koevolučního návrhu obvodů.

3.4.1 Evoluce prediktorů

Na počátku evolučního cyklu evoluce prediktorů, jsou ohodnoceni všichni trenéři pomocí skutečné fitness. Poté dochází k ohodnocení prediktorů takovým způsobem, že pro každý prediktor se vyhodnotí fitness každého jedince z množiny trenérů. Na základě odchylky predikované fitness a skutečné fitness jedinců z množiny trenérů je vypočtena fitness hodnota prediktoru jak je uvedeno v podkapitole 3.2.1.

V generaci při vytváření nové populace je ponechán vždy pouze nejlepší prediktor z předchozí generace. Výpočet fitness hodnoty prediktorů je vždy vyhodnocen pro celou populaci, tudíž i pro prediktor, pro který máme vyčíslenou fitness hodnotu z předchozí generace, a to z důvodu, že při změně obsahu množiny trenérů se fitness hodnota prediktoru může změnit a pak nemusí být nejlepším kandidátem pro predikci fitness kandidátních řešení.

Prediktor, který v dané generaci dosáhl nejlepší fitness je sdílen pro využití v evoluci kandidátních řešení.

Novou generaci prediktorů vytváříme pomocí operátoru křížení. Pro výběr rodičů je použita turnajová selekce. Ze současné populace jsou náhodně vybráni dva jedinci a lepší z nich se stává prvním rodičem potomků. Druhý rodič je vybrán stejně jako první, pouze je zajištěno, aby nebyl vybrán tentýž jako rodič první. Potomci jsou pak vytvoření pomocí jednobodového křížení. U takto vzniklého potomka je nutno provést kontrolu, zda se indexy do množiny referenčních dat neopakují. Pokud tomu tak je, je provedena mutace na tomto kolizním genu. Tuto kontrolu je nutno provést proto, aby se nesnižoval počet ukazatelů do množiny referenčních dat.

Evoluce je ukončena ve chvíli, kdy evoluce kandidátních řešení splní některou z ukončovacích podmínek.

3.4.2 Evoluce kandidátních řešení

Evoluce kandidátních řešení před počátkem spuštění evoluce čeká, než je vybrán nejlépe ohodnocený prediktor, který bude použit pro ohodnocení kandidátních řešení. Ke změně prediktoru dochází periodicky s vyloučením nekonzistence popsané v části 3.4.

Následně je ohodnocena populace pomocí prediktoru. Pokud je nalezen jedinec, který má lepší predikovanou fitness je zařazen do množiny trenérů. Výběr rodiče je opět řízen tak, že se vybere jedinec, který nabývá nejlepší hodnoty fitness a poté je z něj pomocí operátoru mutace vytvořena nová populace. Pravidlem je, že pokud dva jedinci v populaci dosahují stejné nejvyšší fitness hodnoty, pak je jako rodič vybrán jedinec, který nebyl rodičem současné generaci.

Počet zmutovaných genů je řízen náhodnou hodnotou z intervalu $< 1, mut_{max} >$. Během hledání vhodného nastavení evolučních parametrů je nutné se zaměřit i na parametr mut_{max}.

Pokud výpočet predikované fitness spadá do intervalu, který povoluje zadaná odchylka, je řešení ohodnoceno skutečnou fitness. Je-li i v tomto povoleném intervalu i hodnota skutečné fitness, můžou nastat dvě situace. Provádíme evoluci s ohledem na optimalizaci na zpoždění nebo počet použitých funkčních bloků, evoluce pokračuje a hledá optimálnější řešení. V druhém případě nám jde pouze o nalezení řešení, které spadá do zadané povolené odchylky a evoluce je ukončena. Pokud hodnota skutečné fitness nespadá do povoleného intervalu, koevoluce pokračuje dále.

3.5 Návrh paralelizace

V článku [7] je uveden příklad paralelizace pomocí dvou vláken. První vlákno se stará o evoluci kandidátních řešení, zatímco druhé vlákno o evoluci fitness prediktorů, nahrazování trenérů a ohodnocení nových trenérů skutečnou fitness. Tudíž je potřeba řešit tři různé problémy:

- evoluci kandidátních řešení
- evoluci prediktorů
- vytváření nových trenérů a jejich ohodnocení

Tyto tři problémy můžeme provádět sériově (sekvenčně), částečně paralelně a nebo paralelně, což záleží na počtu použitých vláken. Pokud se rozhodneme použít sekvenční přístup probíhá evoluce kandidátních řešení a periodicky je střídána s evolucí prediktorů.

Ke střídání může docházet, jak při každé iteraci, tak jednou za několik generací evoluce kandidátních řešení.

Naopak plně paralelní přístup vyžaduje tři vlákna pro oddělení všech tří zmiňovaných problémů. Vše je opět řízeno v evoluci kandidátních řešení, která řídí kdy má dojít k iteraci množiny trenérů a signalizuje vlákno, ve kterém běží evoluce prediktorů fitness.

V této práci podle článku [7] byl zvolen částečně paralelní přístup, který pro koevoluci využívá dvou vláken. V prvním vlákně opět běží evoluce kandidátních řešení a podle svých výsledků vybírá nové trenéry do množiny trenérů z vlastní populace. Jelikož bylo navrženo, že část množiny trenérů je generována náhodně, kvůli zajištění různorodosti množiny, první vlákno periodicky zasílá signál množině trenérů, na základě kterého dojde k vygenerování nových náhodných trenérů. V druhém vlákně je prováděna evoluce fitness prediktorů. Je potřeba vlákna synchronizovat a zajistit vzájemné vyloučení, aby nedošlo k nekonzistenci výpočtu. Problémem synchronizace vláken se dále zabývá podkapitola 3.5.1.

3.5.1 Synchronizace vláken

Běh vláken je nutno aspoň částečně synchronizovat. Je to nutné i z toho důvodu, že vlákna nemusí mít rovnoměrně přidělena výpočetní zdroje a pak evoluce, kterou simuluje určité vlákno, poběží rychleji, než evoluce běžící v druhém vlákně. Nutnost synchronizovat nastává i v případě, že obě vlákna dostanou stejné výpočetní zdroje. Důvodem takových případů je, že evoluce kandidátních řešení je většinou výpočetně náročnější než evoluce prediktorů.

V případě rychlejší evoluce prediktorů by nebyla zajištěna diverzita populace a kvůli neměnným podmínkám by hrozilo přeučení populace. Tento problém by nenastával kdyby pro evoluci prediktorů bylo místo genetického programování použito kartézského genetického programování. To proto, že se jako rodič vybírá pouze jeden jedinec z populace a nepoužívá se operátor křížení jako je tomu u genetického programování.

Jistým řešením tohoto problému je nechat zpožďovat evoluci prediktorů, oproti evoluci kandidátních řešení. Jenže to by přinášelo problém výchozího nastavení, protože pro různé problémy trvá evaluace řešení různě dlouho. Proto je nejlepším řešením z vlákna kandidátních řešení signalizovat vlákno evoluce prediktorů fitness. Půjde lehce řídit, o kolik pomaleji má evoluce prediktorů běžet a nebude docházet k plýtvání výpočetních zdrojů, pokud ovšem nebude použito aktivního čekání ve vlákně evoluce prediktorů fitness. I přesto bude potřeba pomocí testování najít vhodné nastavení zpoždění evolucí mezi sebou, aby nedocházelo k přeučení, a také aby konvergence hledání řešení byla co nejrychlejší, protože při přílišném zpomalení by nedocházelo k výměně prediktorů podle aktuálního stavu.

3.6 Návrh struktury souboru s trénovacími daty

Struktura souboru, který obsahuje vstupní trénovací data, je navržena tak, aby neobsahovala redundantní informace a byla co nejkompaktnější a nejúčelnější. První řádek v souboru obsahuje informace o počtu primárních vstupů a výstupů výsledného obvodu. Zbylé řádky obsahují trénovací data, která nemusí obsahovat všechny kombinace vstupů. Pokud nějaká vstupní kombinace v souboru chybí, tak pro výsledný obvod není definovaná a ve výpočtu fitness funkce ji nebere v potaz, protože může nabývat libovolné hodnoty, z čehož plyne, že jakýkoliv výstup by byl správný. Příklad struktury souboru pro obvod, který má 4 primární vstupy a 2 primární výstupy může vypadat následovně:

5, 4	Definice počtu sloupců a řádků
0000 : 00	První trénovací vektor
0101 : 01	Druhý trénovací vektor
1010 : 10	Třetí trénovací vektor
1111 : 11	Čtvrtý trénovací vektor

Tabulka 3.2: Struktura vstupního souboru.

Na příkladě souboru, můžeme vidět, že soubor začíná definicí kolik řádků a sloupců má být použito pro evoluci CGP. Poté každý trénovací vektor je na samostatném řádku. Struktura trénovacích vektorů je taková, že nejprve jsou definovány vstupy obvodu a za dvojtečkou jsou uvedeny výstupy jaké má obvod mít na tyto vstupy. Kombinace vstupů, které nejsou uvedeny v souboru nejsou kontrolovány v evaluaci fitnes funkce.

Kapitola 4

Implementace

Pro ověření teoretických přínosů koevolučního návrhu kombinačních obvodů bylo nutné z navrženého přístupu implementovat spustitelnou verzi a experimentálně vyhodnotit, zda opravdu implementované řešení dosahuje teoretických výhod zmíněných v podkapitole 2.6.1. V návrhu byl program dekomponován do různých podčástí, které byly postupně implementovány a testovány jak během implementace samostatně, tak i jako celek. Jako programovací jazyk byla zvolena Java, a to i přes obavy, že budou výpočty trvat dlouho kvůli nemožnosti nízkoúrovňové optimalizace, jako je možná například v jazyce C. Javu jsem zvolil proto, že ji dobře znám a mám zkušenosti s její optimalizací. Právě díky tomu počáteční obavy nebyly naplněny a doba evoluce nebyla příliš dlouhá, naopak byla doba potřebná k implementaci a odladění chyb mnohem kratší proto, že se nejedná o nízkoúrovňový jazyk. Porovnání se známou implementací v jazyce C, můžeme vidět v části 4.7. Java byla použita ve verzi 1.8, ale jelikož nebyly použity žádné vlastnosti, které tato verze přináší, je možné aplikaci přeložit a spustit i s verzí 1.7.

V této kapitole se budu věnovat implementačním detailům jednotlivých podčástí programu. V podkapitole 4.1 se věnuji implementaci paralelizace programu a v její podkapitole detaily implementace evoluce populací CGP i prediktorů. Implementací množiny trenérů se věnuji v podkapitole 4.2, následovanou implementací vyhodnocení fitness funkcí v podkapitole 4.3. V části 4.4 se zabývá implementačními detaily synchroznizací evolucí a část 4.5 optimalizačními funkcemi. Podkapitola 4.6 je zaměřena na optimalizaci jednotlivých dlouho trvajících částí programu. V následujících dvou podkapitolách se věnuji výstupům programu a jeho spuštěním.

4.1 Paralelizace

Jak bylo navrhnuto v části 3.5, budeme koevoluci implementovat s využitím dvou vláken, kdy každá z evolucí poběží ve svém vlastním vlákně. Pro implementaci vláken byly použity prostředky, které poskytuje Java v JDK, bez nutnosti použití dalších knihoven. Jedná se o třídu Thread, která vytváří nové vlákno, jemuž je možné definovat prioritu. Java Virtual Machine (JVM) se automaticky stará o přepínání kontextu, je-li tomu potřeba (například při nedostatku procesorových jader, která by umožňovala čistě paralelní běh). Pokud tuto prioritu nedefinujeme, nové vlákno poběží se stejnou prioritou jako vlákno, ze kterého bylo vytvořeno. Jelikož řízení vzájemné rychlosti běhu vláken chceme řídit sami přesně podle počtu generací jednotlivých evolucí, a ne v závislosti na přidělených výpočetních zdrojích, prioritu necháme nastavenou na výchozí hodnotu, čili obě vlákna mají stejnou prioritu.

4.1.1 Implementace paralelní evoluce obou populací

V implementaci není potřeba řešit žádné uživatelské rozhraní (GUI), a proto je možné hlavní vlákno využít pro výpočet. Jako hlavní vlákno tedy bylo zvoleno to, ve kterém probíhá evoluce kartézského genetického programování. V něm proběhne rovněž inicializace před počátkem evoluce, a dále se bude starat o ukončení evolucí po splnění nějaké z ukončujích podmínek. Během inicializace je vytvořeno nové vlákno, které reprezentuje evoluci prediktorů za pomoci genetického programování. Během inicializace obou evolucí jsou náhodně vygenerovány počáteční populace. Je nutné, aby z hlediska správného vyhodnocení byla nejprve inicializována evoluce prediktorů, aby byl zajištěn výběr nejlepšího prediktoru pro ohodnocení generace kartézského genetického programování. Pokud je zvolena varianta bez koevoluce, vlákno prediktorů je okamžitě ukončeno s tím, že vytvoří pro generaci kartézského genetického programování prediktor, který obsahuje ukazatele na celou trénovací množinu. Po inicializaci obou vláken jsou obě evoluce spuštěny. Běhy evolucí se provádí v metodě **run()**, která je implementována jak v třídě CGP, reprezentující kartézské genetické programování, tak i v třídě GAPred, která simuluje evoluci prediktorů. Evoluce prediktorů beží v nekonečném cyklu a čeká na příkaz k ukončení evoluce, který dostane od vlákna, v němž beží CGP. Evoluce CGP běží v počítaném cyklu, jelikož jedna z ukončujících podmínek je dosažení maximálního počtu generací. O ohodnocení populací se starají metody **evaluatePopulation()** implementované pro každou evoluci zvlášť. Mezi oběma vlákny probíhá výměna jedinců. CGP přidává do množiny trenérů a naopak prediktory vybírají svůj nejlepší prediktor pro ohodnocení CGP. Kvůli konzistenci výpočtů musí tyto výměny probíhat jako kritické sekce proto, aby část jedinců v generaci nebyla ohodnocena jiným prediktorem respektive jinou množinou trenérů. Implementací množiny trenérů se zabývám v následující kapitole.

4.2 Implementace množiny trenérů

Při implementaci množiny trenérů bylo potřeba zvážit fakt, že k ní musí mít nezávislý přístup vlákno evoluce kartézského genetického programování a také vlákno evoluce prediktorů. Vlastnosti, které musí množina trenérů splňovat, se podobají návrhovému vzoru *jedináček*. Proto jsem ji tímto způsobem implementoval. Implementace se nachází ve třídě **TrainersSet**. Jelikož množinu trenérů potřebuje ke svému chodu evoluce prediktorů, její řízení je umístěno právě v této evoluci. Že tím bude více zatíženo vlákno evoluce prediktorů nám nevadí, jelikož běží pomaleji než vlákno evoluce kartézského genetického programování, a tudíž je méně zatíženo.

Množina trenérů implementovaná jako návrhový vzor jedináček umožňuje evoluci kartézského genetického programování vložit nového trenéra tak, aby nebyl ovlivněn výpočet evoluce prediktorů. Nahrazen je vždy nejstarší jedinec, který se v množině nachází. Stejně tak je implementováno přegenerování náhodného jedince. Po předem definovaném počtu cyklů evoluce prediktorů je nejstarší jedinec z náhodné části množiny trenérů nahrazen nově náhodně vygenerovaným.

4.3 Vyhodnocení fitness funkcí

Evaluace fitness funkce je výpočetně velice náročnou částí. Byly implementovány dvě varianty přístupu k fitness funkci. V první se porovnávají jednotlivé bity, jestli jsou ekvivaletní s požadovaným výstupem, a ve druhém přístupu je výpočet prováděn jako aritmetický

Logické hradlo	$C_{in}PIN_A$	$C_{in}PIN_B$	C_{out}
AND	0.0129077	0.0125298	0.116541
NAND	0.0125	0.0129035	0.196269
OR	0.0150616	0.0144258	0.16074
NOR	0.0144193	0.0150643	0.127895
XOR	0.0296528	0.0342661	0.150727
NXOR	0.0296656	0.0342715	0.150698
NOT	0.00932456	–	0.1398

Tabulka 4.1: Seznam kapacit pro logické hradla.

rozdíl. To znamená, že jsou jednotlivým bitům přiřazeny váhy (viz podkapitola 3.3.3). Každá z těchto implementací implementuje interface **ComparingFunction**, který deklaruje třídám, že musí implementovat metodu **evaluate()**. Ta má jako vstupní parametry žádanou odezvu a odezvu vypočtenou daným jedincem. V této metodě se pak nachází implementace Bitového rozdílu (třída **BitXORComparison**) resp. aritmetického rozdílu (třída **ArithmeticComparison**).

4.4 Synchronizace evoluce

V návrhu řešení bylo předpokládáno, že evoluce prediktorů poběží pomaleji než evoluce CGP. Zpomalení nespočívá v delším výpočtu evoluce prediktorů, ale v dosažení toho, aby nedocházelo k častým změnám prediktorů. Cílem tedy je, aby jedna generace prediktorů proběhla za zvolený počet generací CGP. Je proto nutné, aby CGP hlídalo, kdy má proběhnout další iterace v evoluci prediktorů. Vlákno CGP tedy počítá své generace a po dosažení příslušného počtu generací odešle signál vláknu prediktorů za pomocí atomické operace. Té dosahuji pomocí objektu **AtomicBoolean**, který reprezentuje boolean hodnotu, s níž je manipulace atomická. Tento objekt je součástí JDK.

4.5 Optimalizační funkce navrhovaných obvodů

V programu byly implementovány optimalizační funkce, které optimalizují obvod na počet bloků a na zpoždění, které je vypočteno pomocí zjednodušeného modelu zpoždění, nebo na zpoždění podle logical effort. Typ optimalizace je zvolen před spuštěním programu pomocí vstupních parametrů. Optimalizace jsou aplikovány poté, co je nalezeno řešení, které je za pomocí těchto optimalizačních funkcí dále vylepšováno.

Pro logical effort byly použity hodnoty kapacit z knihovny osu180 nm [3], a jsou vypsány v tabulce 4.1.

4.6 Optimalizace programu

V této kapitole se budu věnovat částem implementace, které byly výpočetně náročné, a bylo potřeba se více zaměřit na optimalizaci. Pro lokalizaci těchto částí a také pro testování zlepšení při optimalizacích byl použit nástroj jvisualvm. Pomocí tohoto nástroje jsem profiloval aplikaci jak na paměťovou náročnost tak také na výpočetní náročnost. Ukázku výstupu profilovaní aplikace na výkon (CPU) z tohoto programu je možné vidět na obrázku 4.1.

První výpočetně náročnou částí je evaluace jedince CGP, jelikož je pro jeho ohodnocení potřeba vyhodnotit obvod, který jedinec specifikuje pro vícero vstupů. Stejný výpočet probíhá také v množině trenérů, kde jsou generováni náhodní jedinci. Jak už bylo zmíněno v návrhu, první optimalizace a zrychlení bylo dosaženo tak, že nejsou vyhodnocovány všechny uzly, ale pouze ty, které se podílí na výstupu. Další prostor pro optimalizaci vznikl v reprezentaci výsledku a následného porovnání s žádaným výstupem. Možností byla reprezentace jako pole boolean a implementace vlastního objektu, který toto pole obalí a následně implementovat porovnávací metody, které jsou potřeba. Jelikož bylo jasné, že s optimalizací takového kódu to nebude lehké, hledal jsem jiné řešení. V článku [1] jsem nalezl třídu **BitSet**. Ta implemetuje všechny mnou potřebné metody a je vysoce optimalizovaná jak na paměťové nároky, tak na rychlost. Z této třídy jsem použil metodu **xor()**, která počítá exlusivní disjunkci. Jelikož tato metoda operuje s dvěma BitSety, požadované výstupy k jednotlivým vstupům jsou při inicializaci také načteny jako reprezentace pomocí BitSet. Pomocí exlusivní disjunkce počítám ve fitness evaluaci bitový rozdíl. Poté metodou **cardinality()** zjistím, kolik se ve výsledku nachází jedniček, tzn. rozdílu od požadované hodnoty. Pro aritmetickou fitness třída BitSet implementuje metodu **toLongArray()**, která vrátí long hodnotu, kterou bitové číslo představuje, a dále je určen rozdíl od požadované hodnoty.

Z důvodů vysoké parametrizace programu je také kód velice členěný. Jenže každé rozhodování, do jaké větve jít (každý **if**), představuje další příkaz pro procesor. Na to jsem při implementaci myslel, a proto místa, kde bylo navrhováno vícero možností evaluace (například pro vyhodnocení fitness funkce), nejsou implementováno jako větvení v evaluaci jedinců, ale pouze volaní metody na interface **ComparingFunction**. Do něj je během inicializace dosazena správná implementace podle zvoleného parametru. Díky tomuto je možné jednoduše implementovat další implementace fitness funkcí, bez větších zásahů do kódu.

Jelikož jsem během návrhu uvažoval, že CGP nebude implementovat pouze kombinační obvody, ale například i výpočet matematických výrazů, stejný přístup jako je zvolen u fitness funkce by nestačil, protože obě aplikace mají různý vnitřní stav a proto je potřeba aby evaluace umožnovala různé návratové typy. Proto jsem využil další vlastnost javy – generických typů. Princip evaluace blokové funkce je tedy stejný jako u evaluace fitness funkcí s tím rozdílem, že do interface byl přidán generický typ, který je použit pro návratovou hodnotu jako výsledek evaluace. I když jsem od tohoto rozšíření nakonec upustil, v implementaci je kód ponechán pro snadné rozšíření funkčnosti programu. Blokové funkce tedy implentuje třída **BlockFunction**, která má návratou hodnotu evaluační metody **Boolean**. Bohužel to se ukázalo jako veliké zpomalení běhu programu, protože generika nepracují s primitivními typy a je nutné při každé evaluaci vykonávat autoboxing a unboxing. Jelikož se jedná o nejvíce vytíženou metodu programu, zpomalení dosahuje kolem 30% celkového času (je možné vidět na obrázku 4.1). Rozhodl jsem se i přesto tuto implementaci ponechat, protože přispívá čitelnosti kódu a jednoduchosti jeho rozšiřitelnosti.

Výrazného zrychlení bylo dosaženo v množině trenérů, i když na úkor paměťových nároků. To v našem případě nevadí, jelikož výpočet je paměťově nenáročný. Princip této optimalizace je takový, že máme pro každého trénéra uloženou jeho odezvu pro daný vstup. To znamená, že odezvu trenérů nemusíme vyhodnocovat pro každé ohodnocení prediktorů, ale vypočteme ji pouze jednou a uložíme. Pro další ohodnocení se již používají vypočtené hodnoty.

Poslední částí optimalizace bylo využití vlastností množin implementováné ve třídě **HashSet**. Pomocí této vlastnosti jde jednoduše a přitom velice efektivně s malými výpočetními nároky implementovat například označení aktivních uzlů. To je implementováno tak, že obvod prochází od výstupů ke vstupům. Uzly, které jsou napojené na výstup jsou přidány do

množiny uzlů určených k expanzi. Poté jsou v cyklu uzly postupně vyjmuty z této množiny a přesunuty do množiny expandovaných uzlů. Do množiny uzlů k expandování jsou vloženy uzly, které jsou napojené na uzel, který jsme přesunuli do množiny expandovaných uzlů, ale jen pokud se už nevyskytují v množině expandovaných uzlů.

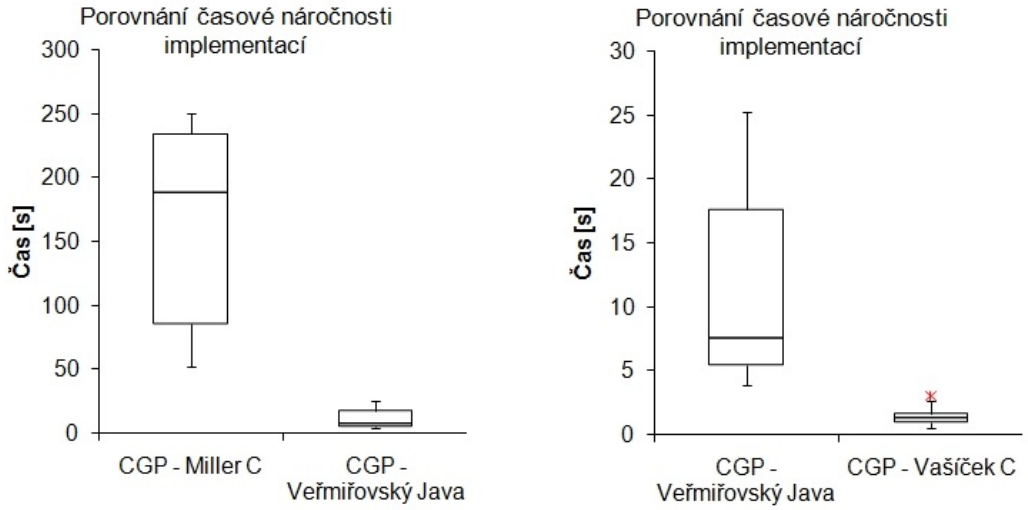
Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
cz.vutbr.fit.xvermi00.cgp.core.functions.BlockFunction.**evaluate** (Object, Ob		4 060 ms (35,9%)	5 524 ms	69 196 032
cz.vutbr.fit.xvermi00.cgp.util.Util.**evaluateCircuit** (cz.vutbr.fit.xvermi00.cgp.		3 417 ms (30,2%)	9 138 ms	3 700 544
cz.vutbr.fit.xvermi00.cgp.core.functions.BlockFunction.**evaluate** (Boolean, B		1 464 ms (13%)	1 464 ms	69 196 032
cz.vutbr.fit.xvermi00.cgp.util.Util.**fitness** (cz.vutbr.fit.xvermi00.cgp.core.Gen		601 ms (5,3%)	10 485 ms	3 698 688
cz.vutbr.fit.xvermi00.cgp.util.Util.**evaluateCircuit** (cz.vutbr.fit.xvermi00.cgp.		358 ms (3,2%)	9 596 ms	3 698 688
cz.vutbr.fit.xvermi00.cgp.core.CGP.**evaluatePopulation** ()		301 ms (2,7%)	11 113 ms	9 632
cz.vutbr.fit.xvermi00.cgp.util.Util.**setGridInputs** (cz.vutbr.fit.xvermi00.cgp.c		287 ms (2,5%)	287 ms	3 698 688
cz.vutbr.fit.xvermi00.cgp.util.Util.**setOutputs** (cz.vutbr.fit.xvermi00.cgp.core		196 ms (1,7%)	196 ms	3 700 544
cz.vutbr.fit.xvermi00.cgp.util.Util.**getOnlyUsedGens** (cz.vutbr.fit.xvermi00.c		133 ms (1,2%)	205 ms	57 821
cz.vutbr.fit.xvermi00.cgp.core.functions.BitXORComparison.**evaluate** (java.l		112 ms (1%)	112 ms	3 716 927
cz.vutbr.fit.xvermi00.cgp.core.CGP.**mutation** ()		94,6 ms (0,8%)	133 ms	9 631
cz.vutbr.fit.xvermi00.cgp.core.Gene.**compareTo** (Object)		88,6 ms (0,8%)	111 ms	1 543 056
cz.vutbr.fit.xvermi00.cgp.core.CGP.**getOnlyUsedGens** (cz.vutbr.fit.xvermi00		75,2 ms (0,7%)	114 ms	9 631
cz.vutbr.fit.xvermi00.cgp.core.Gene.**clone** (cz.vutbr.fit.xvermi00.cgp.core.Ge		39,1 ms (0,3%)	39,1 ms	1 685 425
cz.vutbr.fit.xvermi00.cgp.core.Gene.**compareTo** (cz.vutbr.fit.xvermi00.cgp.c		22,8 ms (0,2%)	22,8 ms	1 543 056
cz.vutbr.fit.xvermi00.cgp.core.CGP.**runCGP** ()		14,8 ms (0,1%)	11 257 ms	1
cz.vutbr.fit.xvermi00.cgp.util.Constants.**lateInit** ()		8,52 ms (0,1%)	8,54 ms	1
cz.vutbr.fit.xvermi00.cgp.core.GA.**createRandomPredictor** ()		4,69 ms (0%)	4,73 ms	32
cz.vutbr.fit.xvermi00.cgp.Main.**parseFile** ()		3,85 ms (0%)	6,43 ms	1
cz.vutbr.fit.xvermi00.cgp.core.GA.**evaluatePredictors** ()		3,12 ms (0%)	5,5 ms	1
cz.vutbr.fit.xvermi00.cgp.util.Constants.**<clinit>**		2,28 ms (0%)	2,30 ms	1
cz.vutbr.fit.xvermi00.cgp.core.Trainer.**<init>** ()		1,14 ms (0%)	1,23 ms	29
cz.vutbr.fit.xvermi00.cgp.core.TrainersSet.**<clinit>**		0,927 ms (0%)	0,927 ms	1
cz.vutbr.fit.xvermi00.cgp.core.TrainersSet.**evaluateTrainer** (cz.vutbr.fit.xve		0,903 ms (0%)	12,4 ms	29

Obrázek 4.1: Náhled použití nástroje jvisualvm pro profilování aplikace.

4.7 Porovnání implementace v jazyce C

Během porovnání rychlosti implementace jsem vzal implementaci knihovny CGP [14], která je napsaná v jazyce C a autorem je Andrew James Turner z výskumné skupiny Juliana Millera (autora CGP), který knihovnu implemtoval pod dozorem autora CGP. Tuto implementaci jsem spustil pro stejné úlohy jako vlastní implementaci. Časy z tohoto porovnání můžeme vidět v grafu na obr. 5.37. Vidíme, že implementace knihovny CGP byla více než 10x pomalejší. Z tohoto důvodu je optimalizace implementace považována za velice úspěšnou, jelikož se nenaplnila obava, že implementace v jazyce Java bude mnohem pomalejší.

Porovnání bylo provedeno také s vysoce optimalizovanou implementací v jazyce C [16], která dosahuje vyšší rychlosti než výše uvedená knihovna v jazyce C a také než implementace popsaná v této práci a to přibližně dvacetinásobně. Přesto nebylo z důvodů uvedených v kapitole 4 upuštěno od implementace v jazyce Java. Převzít a upravit tuto rychlejší implementaci by nebylo jednoduché kvůli tomu, že kvůli velké míře optimalizace není dostatečně univerzální a rozšiřitelná a tudíž by bylo potřeba dělat velké zásahy do implementace při použití v jiném typu úloh.



Obrázek 4.2: Časová náročnost známých implementací.

4.8 Výstupy programu

Výsledný program má více možností výstupu. První možností je, že nás zajímá pouze výsledek (tzn. pouze výsledný obvod). V takovém případě jsou výstupem pouze nejdůležitější informace: počet generací potřebný k nalezení řešení, celkové zpoždění obvodu (pokud je obvod optimalizován na zpoždění), čas doby běhu evoluce, a především výsledný jedinec ve formátu jako je na obr. 3.1.

Program byl implementován především pro experimentování, proto je implementováno vypisování údajů jako například jak se mění fitness v průběhu evoluce. Je tedy možné vidět, ve které generaci bylo nalezeno vyhovující řešení, nebo jak se dařilo obvod optimalizovat. Formát výstupu je ve formátu csv. Příklad těchto dat je možné nalézt v příloze B.

Je možné také využít implementace CGPVieweru [2]. Byla implementována podpora formátu, který tento zobrazovací program používá. Pro tento výstup je implementován přepínač, který nahradí standardní výpis jedince.

4.9 Spuštění programu

Pro spuštění implementovaného programu je potřeba mít nainstalovanou alespoň javu verze 1.7. Protože evoluce mají mnoho parametrů a nelze je odvodit univerzálně, je také implementovaný program velice parametrizovatelný, aby byla zajištěna možnost širokého použití. Právě z důvodu velkého množství parametrů nejsou žádné povinné, a pokud nejsou na vstupu specifikovány, použije se jejich výchozí hodnota. Tyto výchozí hodnoty byly vybírány pomocí experimentů tak, aby měly co nejširší použití. Program je možno spustit na serverech Merlin, Eva a v prostředí FIT VUT v Brně a všude, kde je nainstalovaná Java minimálně ve verzi 1.7. Příklad spuštění programu je možné nalézt v příloze A.

Kapitola 5

Experimentální vyhodnocení

V této kapitole si popíšeme hledání vhodných parametrů, experimentování a zhodnocení vlastností navrženého a implementovaného programu koevolučního návrhu obvodů. Experimenty budou prováděny na sadě úloh, které jsou srhnuty v tabulce 5.1.

Úlohy jsou seřazeny vzestupně, podle jejich složitosti. Složitost je určena podle velikosti trénovací sady pro daný obvod. Úlohy reprezentují základní kombinační obvody, jako je kodér, sčítačka, násobička a parita. Tyto byly vybrány proto, že je jejich princip jednoduchý a obecně známý, a proto na nich půjde dobře prezentovat přínos CGP. Také bude zajímavé porovnat řešení navržené implementovaným programem s řešením, které bylo navrženo pomocí konvenčních metod návrhu obvodů. Pravdivostní tabulky těchto úloh je možné najít v příloze D.

Všechny experimenty byly prováděny na počítači s konfigurací: Intel core i7-4720HQ (2,6 GHz až 3,5 GHz s technologií Turbo Boost) a 12 GB operační paměti. Každý experiment byl spuštěn pro získání statistických dat 50krát.

Označení	Název úlohy	Počet vstupů/výstupů	Počet řádků hodnot v trénovací sadě
F1	Kodér	10/4	10
F2	Parita 5 bit	5/1	32
F3	3 bitová sčítačka	6/4	64
F4	4 bitová násobička	8/5	256
F5	5 bitová sčítačka	14/8	1024

Tabulka 5.1: Seznam všech úloh pro experimenty.

5.1 Nastavení evoluce kartézského genetického programování

Tato část se zabývá hledáním optimálních parametrů pro CGP, které jsou použity pro běh algoritmu v zadaných experimentech. Toto nastavení CGP bude použito ve všech experimentech prováděných v této práci, pokud nebude uvedeno jinak. Jelikož každý experiment má různou složitost, nastavení budou pro každý z experimentů různá.

Hledání parametrů jsem rozdělil do podúloh, které byly seskupeny tak, aby dávaly smysl a postupně na sebe navazovaly. To znamená, že pro každou další podúlohu byly použity výsledky z predešlé podúlohy. Postup byl následující:

- Hledání optimálních rozměrů mřížky (parametry: počet řádků, počet sloupců, l-back).
- Hledání optimální velikosti populace.

- Hledání optimálního počtu genů, které jsou určeny k mutaci a výběr typu mutace (zda mutovat pouze aktivní geny, nebo jakékoliv).
- Určení ukončující podmínky (parametr maximální počet generací).

5.1.1 Určení optimální velikosti mřížky

Během hledání optimálních parametrů, jsem postupoval tak, že jsem pustil evoluce pro všechny kombinace parametrů: počet řádků CGP mřížky, počet sloupců CGP mřížky, L-back. Poté jsem z výběru vyřadil ty kombinace, které nebyly schopny nalézt řešení. Jelikož stále zůstavalo mnoho kombinací, počet jsem zredukoval vyloučením kombinací, které v nějakém z testovaných opakování nebyly schopny řešení najít. Dále jsem vybral nejlepší tak, že jsem porovnal, kolik generací bylo potřeba k nalezení řešení, jakou měly časovou náročnost, a jak se řešení dařilo optimalizovat. Ukázku porovnání pro úlohu F3 je možné vidět na obrázku 5.1. Z tohoto grafu můžeme vidět, že pro úlohu F3 je nejlepší velikost mřížky 6 řádků, 5 sloupců a parametr L-back nastaven na 2. Při jiných testovaných velikostech potřebovalo CGP k vyřešení úlohy více generací.

5.1.2 Určení optimální velikosti populace

Pro určení optimální velikosti populace byl spuštěn test, který s použitím předchozích parametrů testuje, jak bude CGP reagovat na změnu velikosti populace. Optimální velikost je určena podle počtu generací nutných k vyřešení a podle zvýšení výpočetní náročnosti (tzn. nebude vybrán parametr, který sice sníží počet generací potřebných k nalezení řešení, ale neúměrně zvýší čas potřebný k nalezení řešení), viz obrázky 5.2 a 5.3, na kterém jsou zobrazeny výsledky pro úlohu F3. Počet jedinců v populaci sice zvyšoval úspěšnost, ale taky razantně zvyšoval časovou náročnost, z tohoto důvodu bylo pro úlohu F3 zvoleno 6 jedinců v populaci.

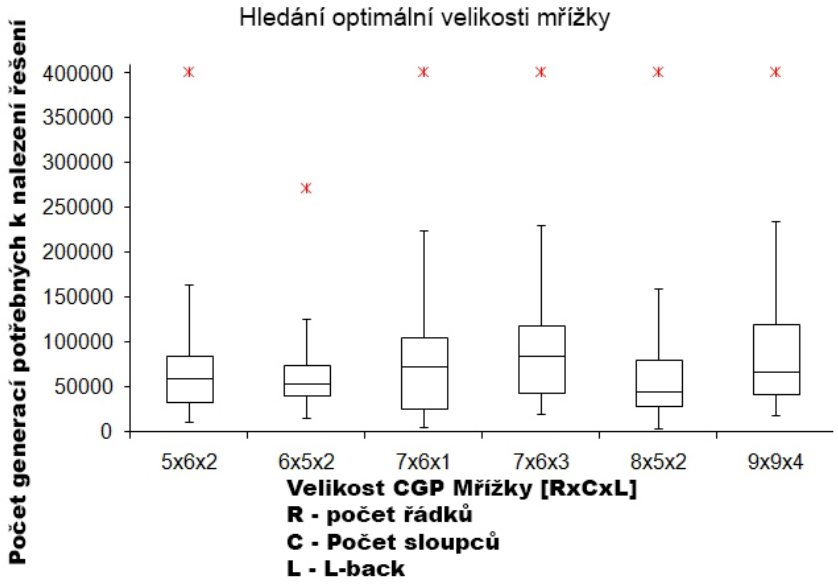
5.1.3 Určení optimální mutace

Hledání optimálního počtu genů k mutaci bylo obdobné jako v předchozích úlohách. Byly spuštěny dva na sobě nezávislé testy, jeden s mutací pouze aktivních genů, druhý s klasickou mutací. V obou testech jsem zvyšoval počet genů, které mohou zmutovat, až do 40% velikosti mřížky, viz obr. 5.4, na němž jsou zobrazeny nejlepší hodnoty pro úlohu F3.

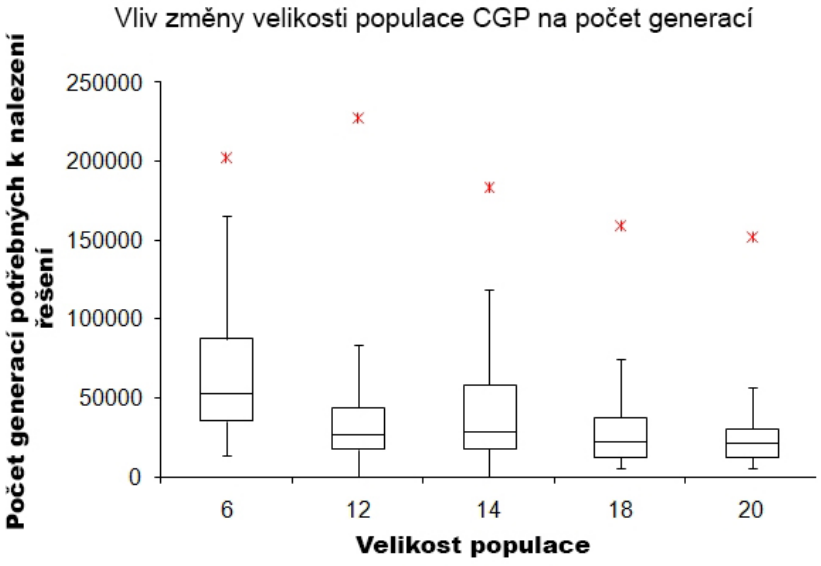
Test mutace nedopadl přesně podle očekávání. Očekával jsem, že varianta pouze aktivních genů bude dosahovat lepších výsledků než klasická mutace. Výsledek zobrazený na obrázcích 5.4 a 5.5, kde jsou zobrazeny nejlepší hodnoty pro úlohu F3, je ale takový, že při klasické mutaci CGP potřebovala skoro 2x méně generací k nalezení řešení, a také dosahovala větší úspěšnosti nalezení řešení než při mutaci pouze aktivních genů. Byl proto vybrán tento typ mutace. Počet genů určených k mutaci byl 10, protože dosáhl největší úspěšnosti, a navíc k nalezení řešení potřeboval jen nízký počet generací.

5.1.4 Určení optimálního počtu generací

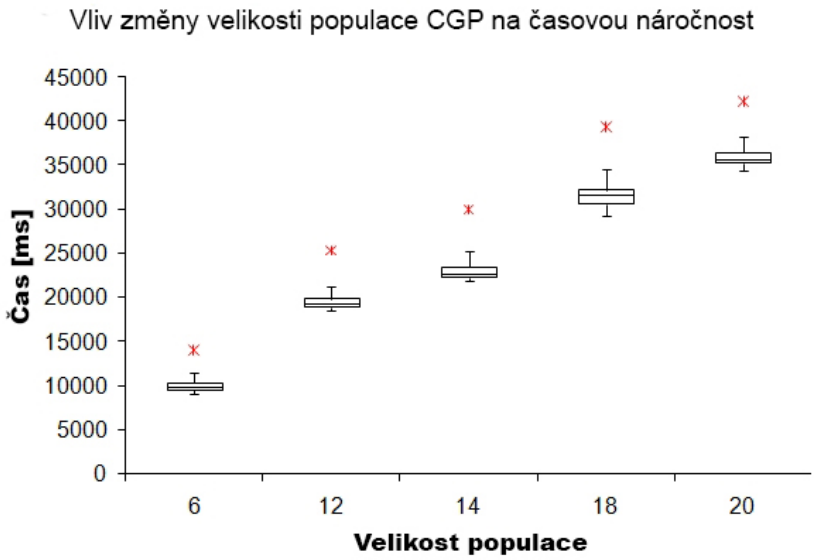
Optimální počet generací jsem určil tak, že jsem vybral počet generací potřebných k vyřešení úlohy pro nejhorší běh pro dané optimální rozměry mřížky v dané úloze a přičetl jsem k této hodnotě alespoň 10%. Proto byla pro úlohu F3, která byla zvolena jako příklad i v předchozích úlohách hledání optimálních parametrů, zvolena hodnota maximálního počtu generací na 250 000.



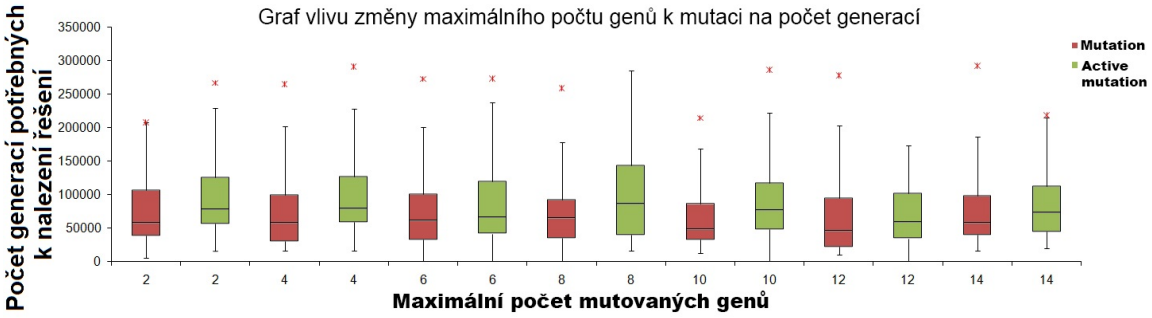
Obrázek 5.1: Graf hledání optimální velikosti mřížky pro CGP.



Obrázek 5.2: Graf vlivu změny velikosti populace CGP na počet generací.



Obrázek 5.3: Graf vlivu změny velikosti populace CGP na časovou náročnost.



Obrázek 5.4: Graf vlivu změny maximálního počtu genů k mutaci na počet generací.

Obrázek 5.5: Graf úspěšnosti nalezení řešení na změnu maximálního počtu genů k mutaci.

5.1.5 Shrnutí

Výsledky hledání optimálních parametrů jsou přehledně zobrazeny v tabulce 5.2.

Označení	F1	F2	F3	F4	F5
Velikost populace	6	12	6	5	6
Počet řádků	3	3	6	10	10
Počet sloupců	5	6	5	20	9
L-back	3	3	2	4	3
Počet genů určených k mutaci	6 klasická	12 aktivní	10 klasická	5 klasická	5 klasická
Maximální počet generací	100 000	20 000	250 000	50 000 000	700 000

Tabulka 5.2: Seznam optimálních parametrů pro CGP.

5.2 Nastavení koevolučních parametrů

Po nalezení optimálních parametrů pro CGP bylo potřeba hledat parametry pro koevoluci. Při hledání byly použity nejvýhodnější experimentálně nalezené hodnoty pro CGP, nalezené v předchozí kapitole. Nalezené parametry shrnuje tabulka 5.2.

Stejně jako v hledání optimálních parametrů pro CGP jsem i zde rozdělil hledání optimálních parametrů pro koevoluci do podúloh:

- Hledání optimální velikosti pole prediktoru.
- Hledání optimální velikosti populace.
- Hledání optimálního poměru zpomalení vůči evoluci CGP.
- Určení velikosti množiny trenérů a poměru jejího složení.

5.2.1 Určení optimální velikosti pole prediktorů

Velikost pole prediktorů určuji procentuálně, ve vztahu s velikostí trénovací sady. Tato velikost byla testována v rozsahu od 15% do 90%. Testovat v menším rozsahu by nemělo smysl, a větší by nepřinesl už žádné výrazné vylepšení oproti samostatnému CGP. V testech byla velikost pole inkrementována po 5% (pokud takový inkrement dával smysl – například u F1 bylo pole inkrementováno po 10%, jelikož menší inkrement není možný). Dále byly porovnávány změny v potřebném počtu generací pro nalezení řešení a změny v časové složitosti, viz obrázky 5.6 a 5.7, které zobrazují grafy hledání velikosti pro úlohu F3. Pro tuto úlohu byla jako kompromis mezi počtem potřebných generací a dobrou úspěšností zvolena velikost 50% velikosti trénovací sady.

5.2.2 Určení optimální velikosti populace prediktorů

Optimální velikost populace byla testována po určení velikosti pole prediktorů, a to tak, že byly experimentálně vyzkoušeny hodnoty, které se pohybují kolem hodnoty 5, jež bývá (podle [7]) často používanou velikostí populace. Výsledky testů jsou shrnuty v tabulce 5.2. V grafu na obr. 5.8 najdeme příklad nejlepších výsledků pro úlohu F3, pro kterou byla vybrána hodnota 5 jedinců v populaci.

5.2.3 Určení složení populace prediktorů

V této podúloze jsem zjišťoval, jaký vliv mají změny vytváření nové populace prediktorů na nalezení řešení. Pro úlohu F3 nebylo vybráno nejlepší nalezené řešení, protože nejlepší řešení neobsahovalo žádné náhodně vygenerované prediktory, což by mohlo způsobit, že by v některých případech řešení uvázlo v lokálním maximu (viz obrázek 5.9). Proto bylo vybráno řešení s 50% nejlepších jedinců, 25% jedinců vzniklých pomocí operace křížení a 25% jedinců náhodně vygenerovaných. V případech kdy zadané procentuální rozdělení není možné dodržet (nejsou možná desetinná čísla), jsou hodnoty zaokrouhleny směrem dolů a celý zbytek je pak přirazen k náhodně vygenerovaným jedincům.

5.2.4 Určení optimálního poměru zpomalení vůči evoluci CGP

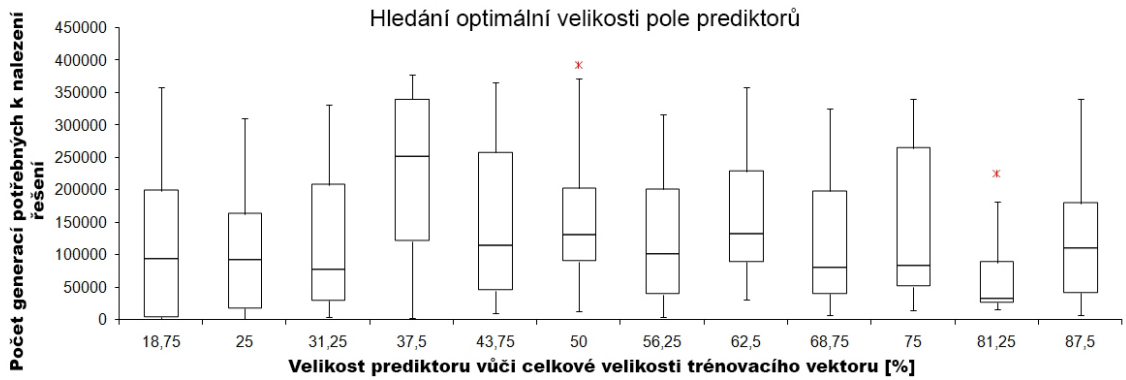
V tomto testu bylo úkolem nalézt takové zpomalení, které by bylo co největší, a zároveň neovlivňovalo schopnost nalezení řešení. K tomu by mohlo dojít při velkém zpomalení, kdy by populace prediktorů nedokázala dostatečně rychle reagovat na změny v populaci CGP. Naopak při velké rychlosti by se populace prediktorů mohla měnit tak rychle, že by evoluce kandidátních řešení na změny nestihla reagovat a nebo by zbytečně zatěžovala výpočetní zdroje. Jako příklad je opět zobrazen graf na obr. 5.12 pro úlohu F3. V něm můžeme vidět, že příliš rychlé změny prediktoru způsobují klesání úspěšnosti řešení. Proto bylo v této úloze rozhodnuto, že k synchronizaci dojde jednou za 36 000 generací, přičemž bylo dosaženo 100% úspěnosti nalezení řešení.

5.2.5 Určení velikosti archivu trenérů a jeho složení

Hledání optimální velikosti archivu trenérů probíhalo tak, že velikost byla měnena v rozmezí od velikosti populace CGP do jejího dvojnásobku. Nejvhodnější velikostí byla ta, která nejlépe ovlivňovala prediktory, aby počet generací potřebných k nalezení řešení byl co nejmenší. Do tohoto testu byly zařazeny také změny poměru složení množiny trenéru náhodně vygenerováných trenéru k nejlepším trenérům. Vybrána byla kombinace, která dosahovala nejlepšího výsledku. Příklad řešení pro úlohu F3 můžeme vidět na obr. 5.13.

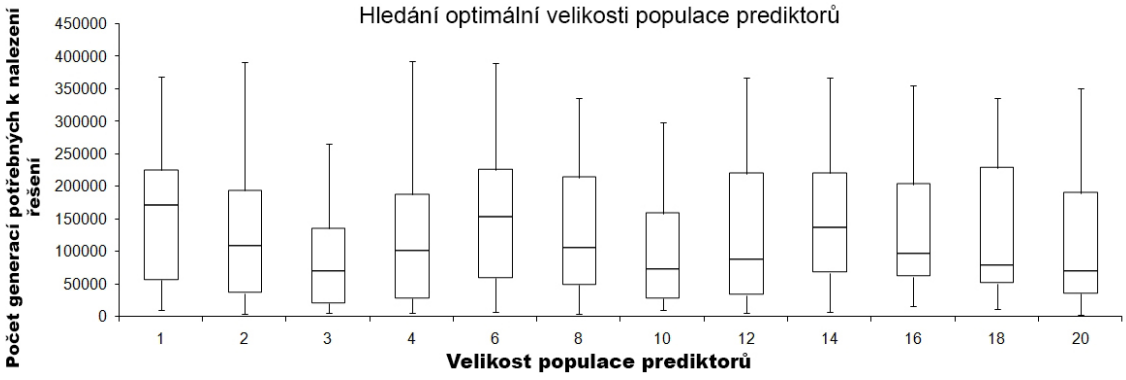
5.2.6 Povolená odchylka

Během experimentování s koevolucí jsem zjistil, že pokud zvolím povolenou odchylku pro koevoluci, řešení je nalezeno s větší úspěšností a rychleji, než pokud žádná tolerance není zvolena. Proto byl proveden test, který zjišťuje, jak velká odchylka pro koevoluci je nejvhodnější. Hodnoty byly testovány od 0 do 25% po změnách 5%. Větší hodnoty rozhodně smysl nemají, jelikož už při 25% může celá čtvrtina výstupů, na které prediktor ukazuje, být špatná. V kombinaci grafů na obrázcích 5.10 a 5.11 můžeme vidět, že nejvhodnější hodnota pro úlohu F3 byla 10%.

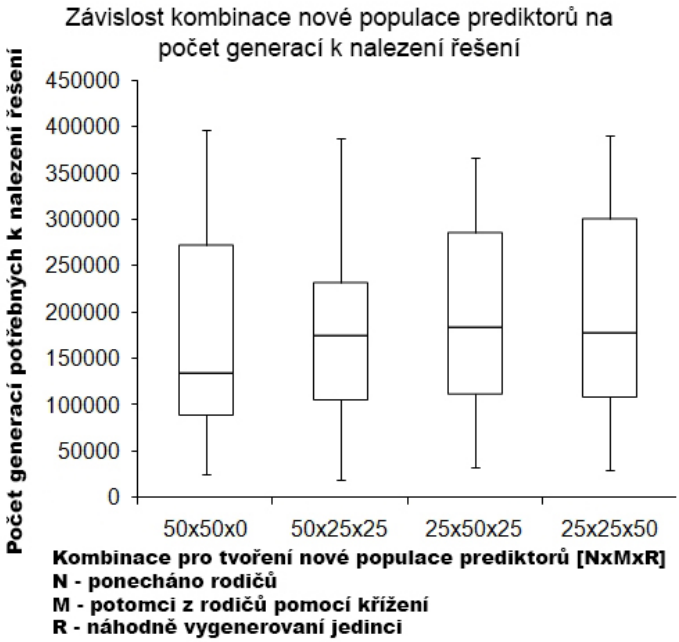


Obrázek 5.6: Graf hledání optimální velikosti prediktoru v závisloti na počtu generací.

Obrázek 5.7: Graf hledání optimální velikosti prediktoru v závisloti na úspěšnosti nalezení řešení.

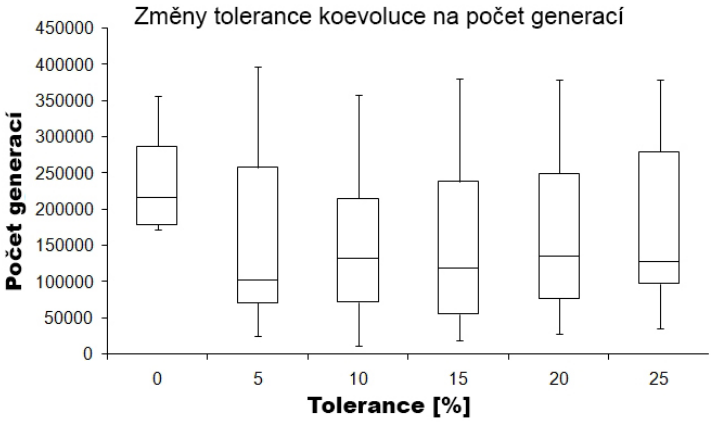


Obrázek 5.8: Graf hledání optimální velikosti populace prediktorů na počtu generací k nalezení řešení.



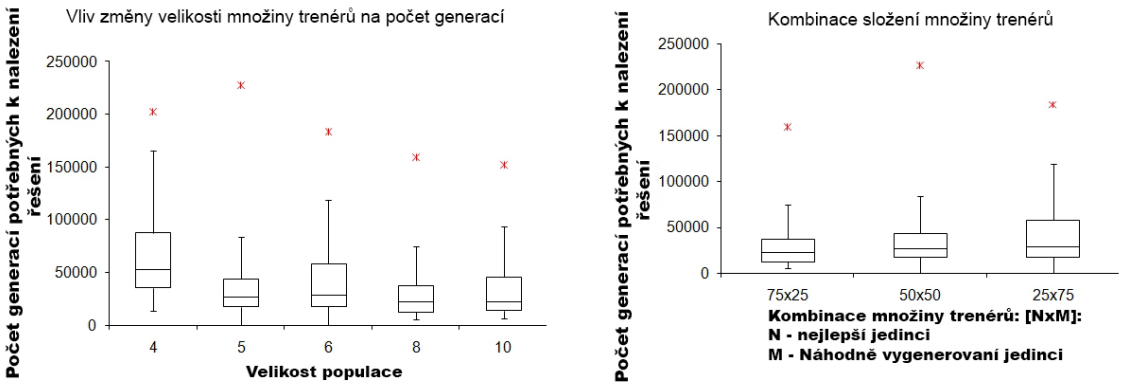
Obrázek 5.9: Graf hledání optimální velikosti kombinace nové generace prediktorů na počtu generací k nalezení řešení.

Obrázek 5.10: Graf vlivu změny velikosti tolerance koevoluce na úspěšnost nalezení řešení.



Obrázek 5.11: Graf vlivu změny velikosti tolerance koevoluce na počet generací potřebných k nalezení řešení.

Obrázek 5.12: Graf hledání optimální doby synchronizace populace prediktorů vůči CGP.



Obrázek 5.13: Grafy hledání optimální velikosti množiny trenérů a jejího složení.

5.2.7 Shrnutí

Optimální hodnoty pro koevoluci jsou shrnuty přehledně v tabulce 5.3.

Označení	F1	F2	F3	F4	F5
Velikost populace prediktorů	3	7	5	32	–
Kombinace nové populace prediktorů	50x0x50	50x25x25	50x25x25	50x25x25	–
Velikost pole prediktorů	40%	31.25%	50%	128	–
Četnost iterací koevoluce k CGP	2 600	800	36 000	5 000 000	–
Velikost množiny trenérů	6	6	8	10	–
Poměr nejlepších trenérů k náhodným	2:1	2:1	3:1	3:1	–
Povolená odchylka	10	20	10	15	–

Tabulka 5.3: Seznam optimálních parametrů pro koevoluci.

5.3 Vyhodnocení řešených úloh

V této kapitole jsou shrnuty výsledky, kterých bylo dosaženo při experimentování s úlohami (tabulka 5.1) a jejich nastavením, jenž jsou popsány v předešlých kapitolách. Pro větší přehlednost budou výsledky promítnuty do přehledných grafů.

5.4 Úloha F1 – kodér 10 na 4

Tato část se věnuje srovnání mezi CGP s koevolucí a bez koevoluce. Poté řešení navržené pomocí evoluce bude srovnáno s klasickými metodami návrhu.

5.4.1 Srovnání CGP s koevolucí v CGP

První úloha, pro kterou byl porovnáván přínos koevoluce oproti klasickému CGP, byl kodér 10 na 4. Úloha z testovaných úloh má nejmenší počet trénovacích vektorů, a tudíž je přínos koevoluce zanedbatelný. Na obr. 5.14 vidíme, že úspěšnost neklesla a počet generací (obr. 5.15) byl snížen zanedbatelně (cca o 2%). Časová náročnost však stoupla proto, že pro takto malou trénovací množinu byla režie spojená s koevolucí větší než přínos koevoluce.

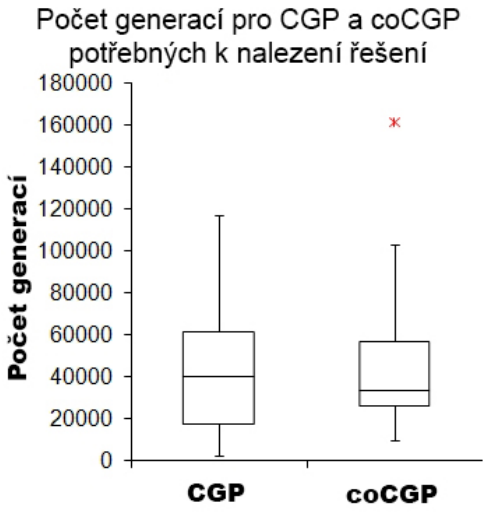
Při běhu zaměřeném na optimalizaci výsledného obvodu jsou již časy velice podobné, což je způsobeno tím, že optimalizace je ukončena až ukončující podmínkou maximálního počtu generací CGP. Tato optimalizace tudíž potřebuje ke svému běhu více generací, než je nutné pouze pro nalezení řešení. A jelikož pro hledání optimalizovaného řešení již koevoluce nepřináší žádné zrychlení, je tento časový rozdíl velice malý.

Na obr. 5.18 je vidět, že počet evaluací obvodu pro vyhodnocení fitness funkce se zmenšil podstatně i pro malý trénovací vektor, jaký je použit v úloze F1.

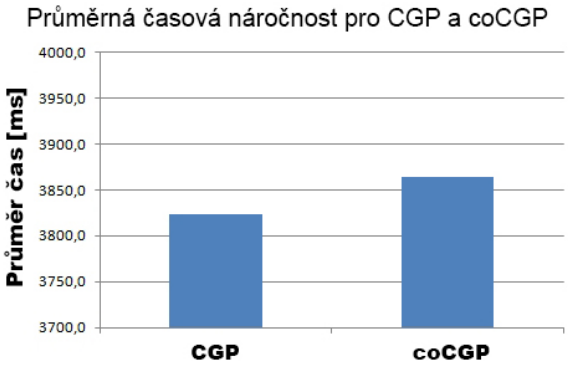
5.4.2 Nejlepší nalezené řešení úlohy

Nejlepší nalezené řešení pro úlohu F1 a optimalizaci na počet bloků můžeme vidět na obrázku 5.19. Řešení obsahuje 10 funkčních bloků. Nejlepší řešení, které bylo optimalizováno na zpoždění pomocí logical effort, je zobrazeno na obrázku 5.20. Toto řešení obsahuje 10 logických hradlel stejně jako při řešení pro optimalizaci na počet hradel, ale logické funkce hradel jsou rozdílné. Zpoždění podle logical effort má $6{,}47745\tau$.

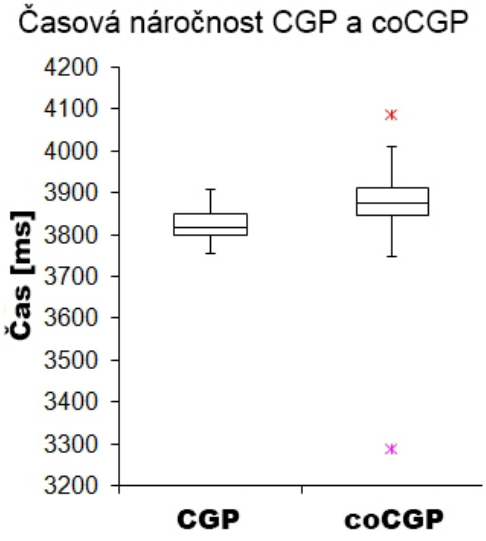
Obrázek 5.14: Graf úspěšnosti nalezení řešení CGP a coCGP pro F1.



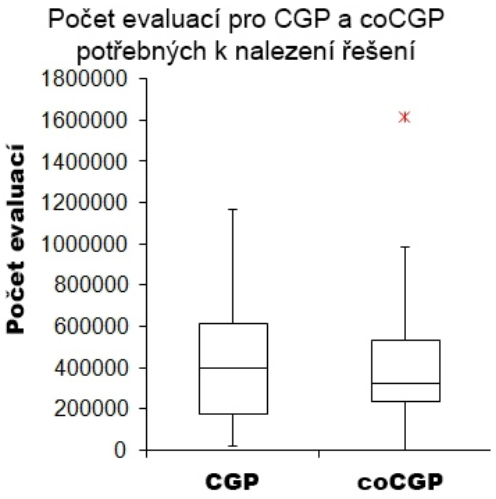
Obrázek 5.15: Graf potřebného počtu generací k nalezení řešení v CGP a coCGP pro F1.



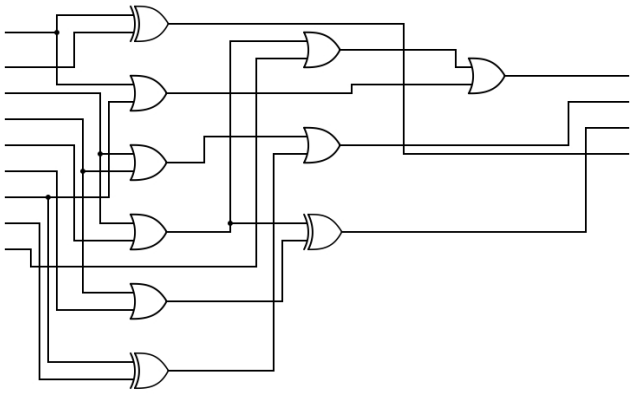
Obrázek 5.16: Graf časové náročnosti CGP a coCGP pro F1.



Obrázek 5.17: Graf časové náročnosti CGP a coCGP pro F1.



Obrázek 5.18: Graf počtu evaluací CGP a coCGP pro F1.



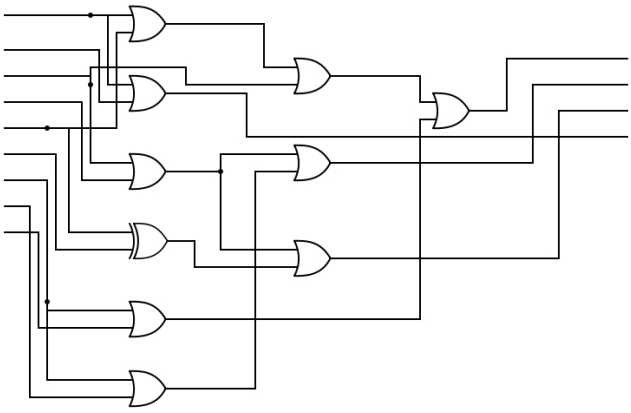
Obrázek 5.19: Nejlepší nalezené řešení kodéru 10 na 4 pomocí coCGP optimalizované na počet log. hradel.

5.4.3 Srovnání nalezeného řešení s konvenčními metodami návrhu

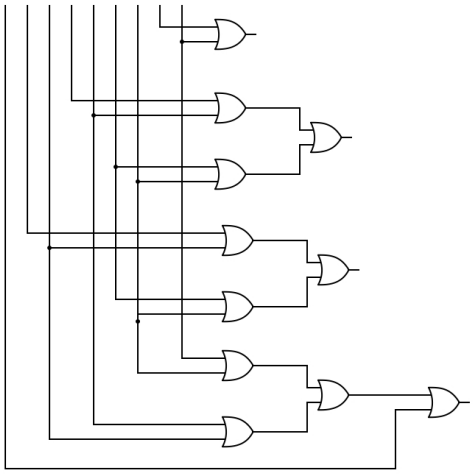
Na obrázku 5.21 vidíme kodér navržený pomocí konvenčních metod návrhu. Tento obvod má o jedno logické hradlo více, než nejlepší navžené řešení pomocí CGP. Zpoždění pomocí logical effort je u konvenčního řešení stejné jako u nejlépe navrženého řešení pomocí CGP optimalizovaného na zpoždění. Shrnutí je uvedeno v tabulce 5.4.

Návrh řešení pomocí:	počet bloků	zpoždění logical effort
Konvenční metoda návrhu	11	6,47745τ
coCGP	10	6,47745τ

Tabulka 5.4: Tabulka srovnání parametrů obvodu mezi CGP a konvenční metodou návrhu pro úlohu F1.



Obrázek 5.20: Nejlepší nalezené řešení kodéru 10 na 4 pomocí coCGP optimalizované na zpoždění.



Obrázek 5.21: Kodér 10 na 4 navržený pomocí konvenčních metod.

5.4.4 Přibližné počítání

Jak je uvedeno v podkapitole 2.4.1, je možné dosáhnout snížení zpoždění nebo menšího počtu hradel pomocí snížení přesnosti výpočtu. Výsledky, kterých jsem dosáhl v úloze F1 při snížení přesnosti výpočtu, jsou shrnuty v tabulce 5.5, a to jak pro optimalizace na počet hradel, tak pro optimalizaci na zpoždění.

Snížení přesnosti o:	bloků / logical effort
10%	8 / 4,249246τ
15%	8 / 4,249246τ
20%	6 / 4,238725τ
25%	6 / 4,238725τ
30%	4 / 2τ

Tabulka 5.5: Tabulka zlepšení parametrů obvodu při snížení přesnosti výpočtu pro úlohu F1.

5.5 Úloha F2 – parita 5 bit

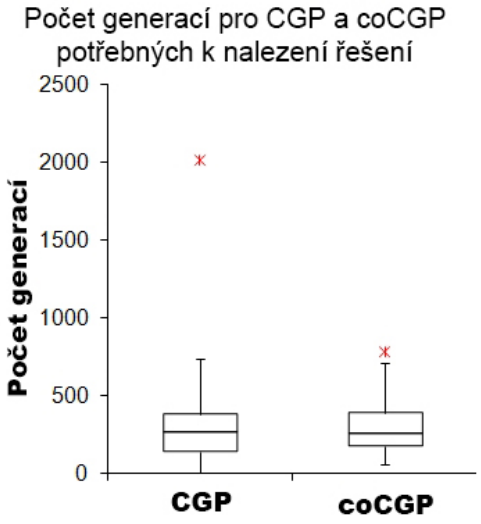
Tato část se věnuje srovnání mezi CGP s koevolucí a bez koevoluce. Poté řešení navržené pomocí evoluce bude srovnáno s klasickými metodami návrhu.

5.5.1 Srovnání CGP s koevolucí v CGP

V úloze 5 bitové parity se již nachází více trénovacích vektorů, což však stále pro přínos koevoluce nestačí. Úspěšnost nalezení řešení je pro oba přístupy stejná a to 100% (viz obr. 5.22). Na obr. 5.23 vidíme, že počet generací je v podstatě také totožný jak v CGP, tak v coCGP, ale v grafu na obr. 5.24 je zobrazena průměrná časová náročnost a na obr. 5.25 , která je již v coCGP vyšší. To je opět způsobeno větší režií evolucí v coCGP. Úspěšnost nalezení řešení byla opět pro obě varianty 100%. Na obr. 5.26 je vidět stejně jako v předchozí úloze, že koevoluce podstatně zmenšila počet vyhodnocení obvodu při výpočtu fitness.

Stejně jako v předchozí úloze, při běhu zaměřeném na optimalizaci výsledného obvodu jsou již časy velice podobné. To je opět způsobené tím, že optimalizace je ukončena až ukončující podmínkou maximálního počtu generací CGP. Tato optimalizace tudíž potřebuje ke svému běhu více generací, než je nutné pouze pro nalezení řešení. A jelikož pro hledání optimalizovaného řešení již koevoluce nepřináší žádné zrychlení, je tento časový rozdíl velice malý.

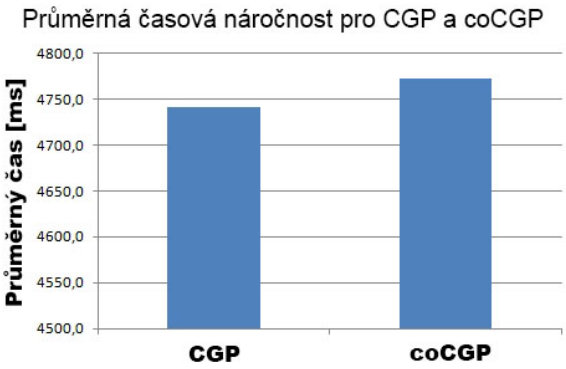
Obrázek 5.22: Graf úspěšnosti nalezení řešení CGP a coCGP pro F2.



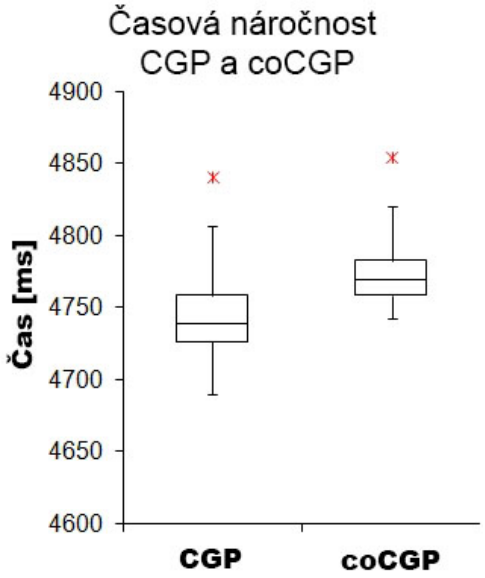
Obrázek 5.23: Graf potřebného počtu generací k nalezení řešení v CGP a coCGP pro F2.

5.5.2 Nejlepší nalezené řešení úlohy

Nejlepší nalezené řešení pro úlohu F2 a optimalizaci na počet bloků můžeme vidět na obrázku 5.27. Stejné řešení bylo také nalezeno při optimalizaci na zpoždění. Řešení obsahuje 4 logická hradla a zpoždění podle logical effort má 13,573855τ.



Obrázek 5.24: Graf časové náročnosti CGP a coCGP pro F2.



Obrázek 5.25: Graf časové náročnosti CGP a coCGP pro F2.

5.5.3 Srovnání nalezeného řešení s konvenčními metodami návrhu

V úloze F2 je nejlepší nalezené řešení na počet bloků totožné s řešením, které bylo navrženo konvenčním způsobem (viz obrazek 5.28). Jedinou změnou je propojení uzlů, ale jelikož je celý obvod složen pouze z jediného typu logické funkce, výsledná funkce je totožná a na pořadí propojení vstupů nezáleží. Toto propojení se pozitvně projevilo na zpoždění. Konvečně navržené řešení má zpoždění pomocí logical effort 18.728τ. Z toho vyplývá, že nalzené řešení pomocí koevoluce má zpoždění menší o 5.15τ. Takového výsledku by docílil návrhář také, pokud by mu šlo o snížení zpoždění obvodu, stejnou změnou jakou provedlo CGP. Úprava je jednoduchá, stačí přesunout jeden ze XORů tak, aby nebyly všechny XORy v sérii, ale byly dva na stejné úrovni. Shrnutí je uvedeno v tabulce 5.6.

Návrh řešení pomocí:	počet bloků	zpoždění logical effort
Konvenční metoda návrhu	4	18.728τ
coCGP	4	$13{,}573855\tau$

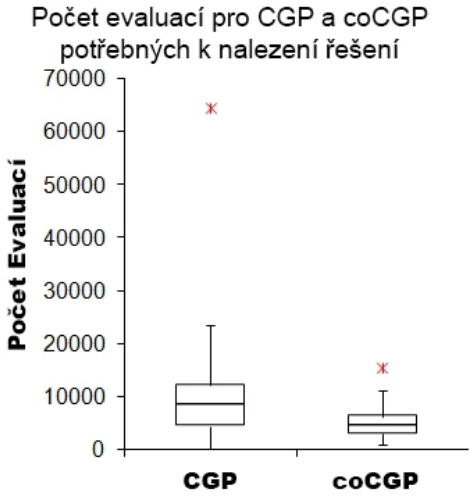
Tabulka 5.6: Tabulka srovnání parametrů obvodu mezi CGP a konvenční metodou návrhu pro úlohu F2.

5.5.4 Přibližné počítání

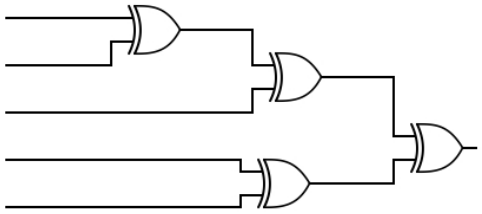
Stejně jako v podkapitole 5.4.4, ve které jsou shrnuty výsledky zlepšení parametrů obvodu při snížení přesnosti výpočtu, jsou v tabulce 5.7. shrnuty výsledky pro úlohu F2.

Snížení přesnosti o:	bloků / logical effort
10%	4 / 13,57385τ
15%	4 / 11,27764τ
20%	4 / 10,98647τ
25%	4 / 10,98647τ
30%	4 / 10,98647τ

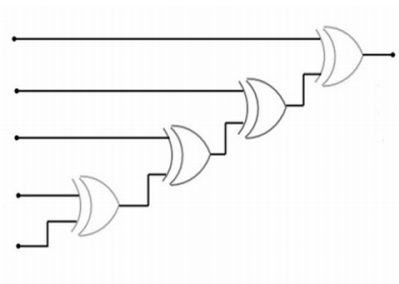
Tabulka 5.7: Tabulka zlepšení parametrů obvodu při snížení přesnosti výpočtu pro úlohu F2.



Obrázek 5.26: Graf počtu evaluací CGP a coCGP pro F2.



Obrázek 5.27: Nejlepší řešení 5 bit parity nalezené pomocí coCGP.



Obrázek 5.28: Parita 5 bit navržena pomocí konvenčních metod.

5.6 Úloha F3 – 3–bitová sčítačka

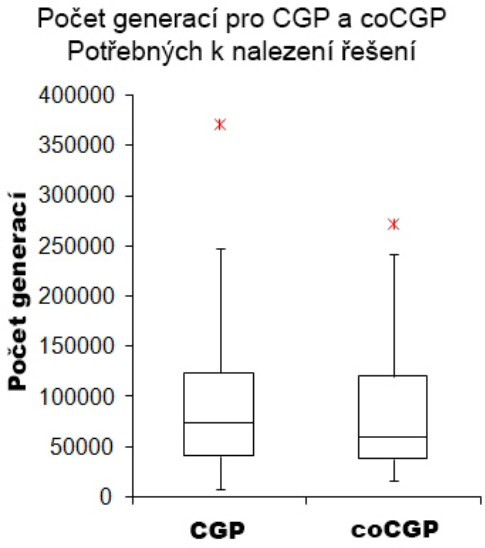
Tato část se věnuje srovnání mezi CGP s koevolucí a bez koevoluce. Poté řešení navržené pomocí evoluce bude srovnáno s klasickými metodami návrhu.

5.6.1 Srovnání CGP s koevolucí v CGP

V této úloze jde již naplno vidět přínos koevoluce oproti klasickému CGP v úloze 3–bitové sčítačky. Jak můžeme vidět na obrázcích 5.29 a 5.30, koevoluce a CGP mají stejnou úspěšnost a přibližně stejný počet generací potřebných k nalezení řešení, ale průměrná časová náročnost klesla o 20% (viz obr. 5.31). Na obr. 5.32 vidíme, že poklesly i hraniční hodnoty časové náročnosti. Stejně jako v předchozích úlohách koevoluce pro úlohu F3 podstatně snížila počet evaluací obvodu. Porovnaní počtu evaluací je zobrazeno na obrázku 5.33.

Při běhu zaměřeném na optimalizaci výsledného obvodu, je závěr stejný jako v předchozích dvou úlohách.

Obrázek 5.29: Graf úspěšnosti nalezení řešení CGP a coCGP pro F3.



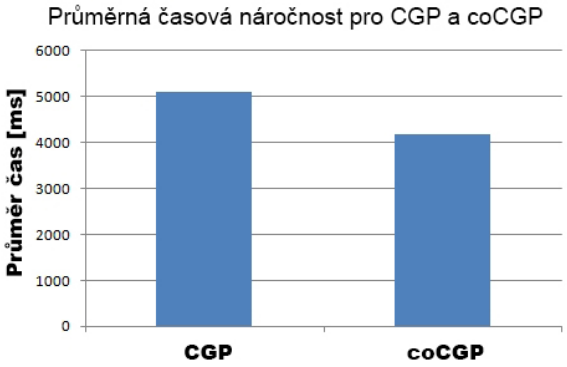
Obrázek 5.30: Graf potřebného počtu generací k nalezení řešení v CGP a coCGP pro F3.

5.6.2 Nejlepší nalezené řešení úlohy

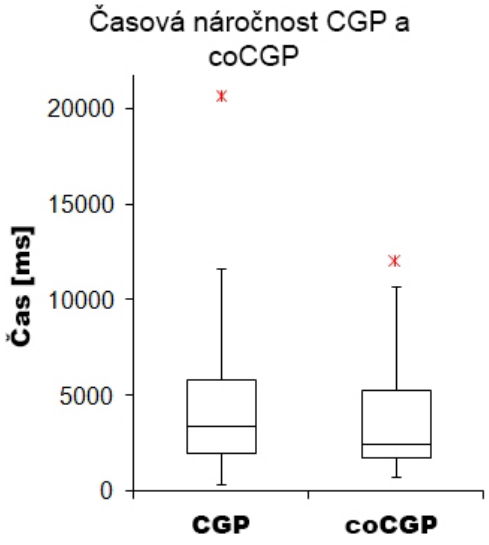
Nejlepší nalezené řešení pro úlohu F3 a optimalizaci na počet bloků můžeme vidět na obrázku 5.34. Řešení obsahuje 13 funkčních bloků. Nejlepší řešení, které bylo optimalizováno na zpoždění pomocí logical effort, je zobrazeno na obrázku 5.35. Toto řešení obsahuje 20 log. hradel a zpoždění má 11,02963τ.

5.6.3 Srovnání nalezeného řešení s konvenčními metodami návrhu

Nejlepší nalezené řešení pomocí coCGP v porovnání s řešením navrženém pomocí konvenčních metod zobrazeném na obrázku 5.36, má pouze o 1 logické hradlo navíc, což není



Obrázek 5.31: Graf časové náročnosti CGP a coCGP pro F3.



Obrázek 5.32: Graf časové náročnosti CGP a coCGP pro F3.

o mnoho horší. Zpoždění pro konvenční řešení je 14.35792τ. To je o $3{,}32829\tau$ větší než v řešení nalezeném pomocí coCGP.

Zajímavostí také je, že obvody se na sebe celkem podobají (optimalizovaná verze na počet hradel). Kromě třetí vrstvy, kde je o jeden XOR více, je v každé vrstvě stejný počet logických hradel. Poslední tři vrstvy jsou dokonce složeny ze stejných typů logických hradel.

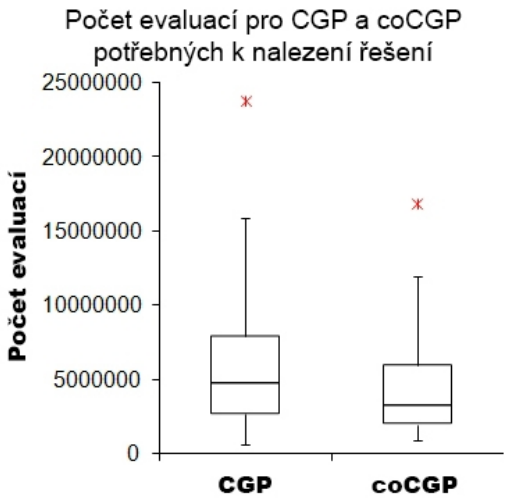
Můžeme si také všimnout, že rozšířit sčítačku navrženou pomocí koevoluce by nebylo jednoduché, zatímco ze struktury konvenčně navržené sčítačky jsme schopni odhadnout, jak funguje, a rozšíření na vícebitovou sčítačku je triviální.

V případě optimalizace na zpoždění pomocí coCGP, které je zobrazeno na obrázku 5.36, je pochopení funčnosti ještě obtížnější. CGP zde navrhlo strukturu, která se již vůbec nepodobá na řešení navržené pomocí konvenčních metod a také takto navrženou sčítačku nelze rozšířit na více bitů jednoduchými úpravami obvodu.

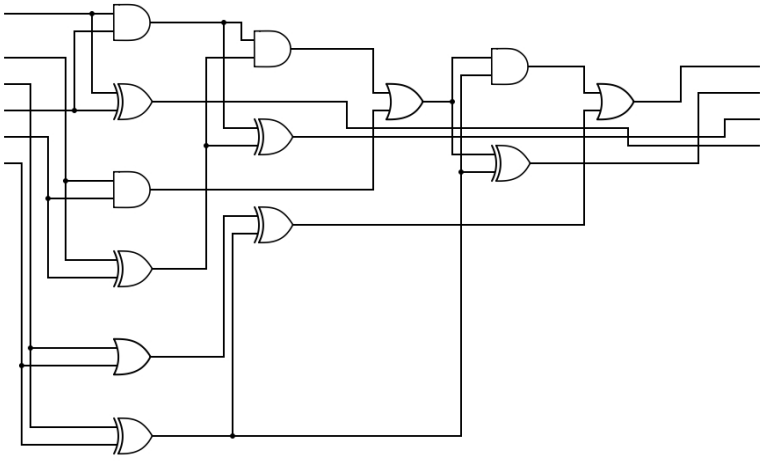
Shrnutí parametrů jednotlivých návrhu obvodů je uvedeno v tabulce 5.8.

Návrh řešení pomocí:	počet bloků	zpoždění logical effort
Konvenční metoda návrhu	12	14.35792τ
coCGP	13	$11{,}02963\tau$

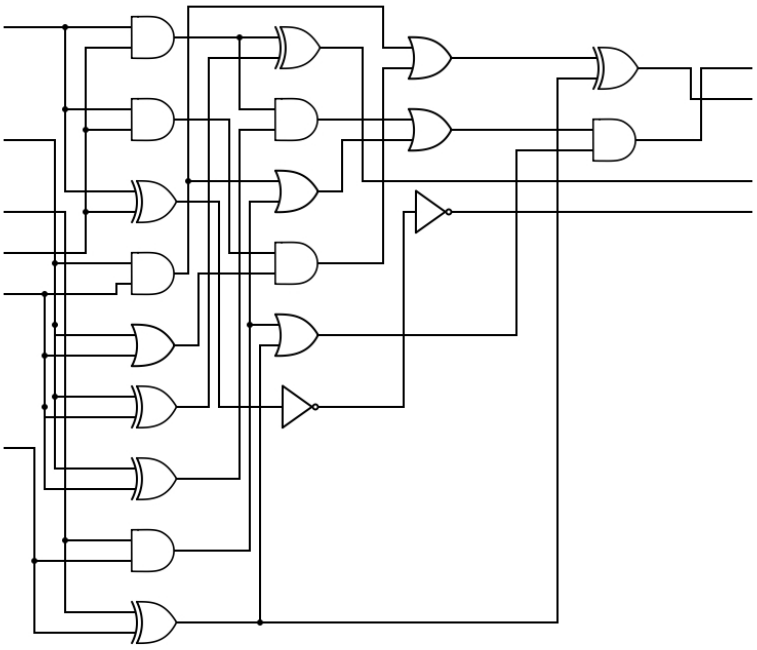
Tabulka 5.8: Tabulka srovnání parametrů obvodu mezi CGP a konvenční metodou návrhu pro úlohu F3.



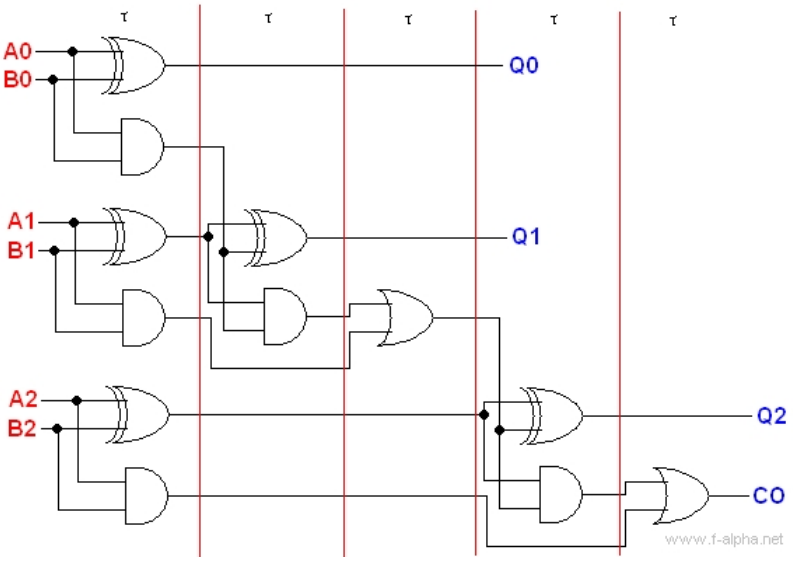
Obrázek 5.33: Graf počtu evaluací CGP a coCGP pro F3.



Obrázek 5.34: Nejlepší řešení 3 bit sčítačky nalezené pomocí coCGP optimalizované na počet log. hradel.



Obrázek 5.35: Nejlepší řešení 3 bit sčítačky nalezené pomocí coCGP optimalizované na zpoždění.



Obrázek 5.36: Sčítačka navržená pomocí konvenčních metod.

5.6.4 Přibližné počítání

V tabulce 5.9 jsou srhnuty výsledky zlepšení parametrů obvodu (zpoždění, počet hradel) při snížení přesnosti výpočtu.

Snížení přesnosti o:	bloků / logical effort
10%	11 / 8,86032τ
15%	10 / 8,786927τ
20%	9 / 8,786927τ
25%	8 / 6,3828325τ
30%	7 / 6,3828325τ

Tabulka 5.9: Tabulka zlepšení parametrů obvodu při snížení přesnosti výpočtu pro úlohu F3.

5.7 Úloha F4 – 4 bitová násobička

Tato část se věnuje srovnání mezi CGP s koevolucí a bez koevoluce. Poté řešení navržené pomocí evoluce bude srovnáno s klasickými metodami návrhu.

5.7.1 Srovnání CGP s koevolucí v CGP

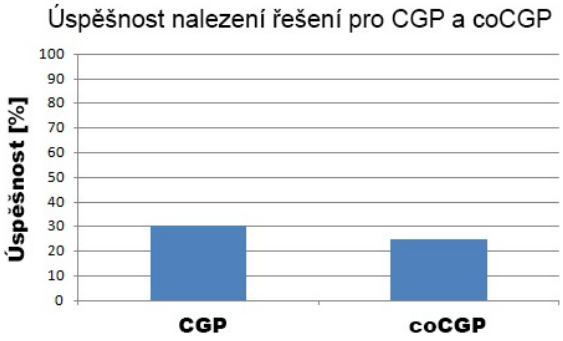
Úloha F4 je již velice složitá, a proto byl přínos koevoluce očekáván větší, než v úloze F3. Bohužel, jak můžeme vidět v grafu na obrázku 5.37 oproti klasickému CGP, zde klesla úspěšnost nalezení řešení. To je nejspíše zapříčiněno tím, že nemusely být nalezeny ideální nastavení parametrů koevoluce, z důvodu dlouho trvajících výpočtů. Obrázek 5.38 zobrazující graf počtu generací potřebných pro nalezení řešení ukazuje, že pro koevoluci je počet generací menší, než pro klasické CGP. Přínos koevoluce je i v této úloze ve snížení počtu evaluací. Na obrázku 5.41 můžeme vidět, že počet evaluací obovdu klesnul přibližně o 30% což v tomto množství je podstatné zlepšení. Z tohoto důvodu je také menší časová náročnost koevoluce (viz. obrázek 5.40) a také průměrná časová náročnost (viz. obrázek 5.39).

5.7.2 Nejlepší nalezené řešení úlohy

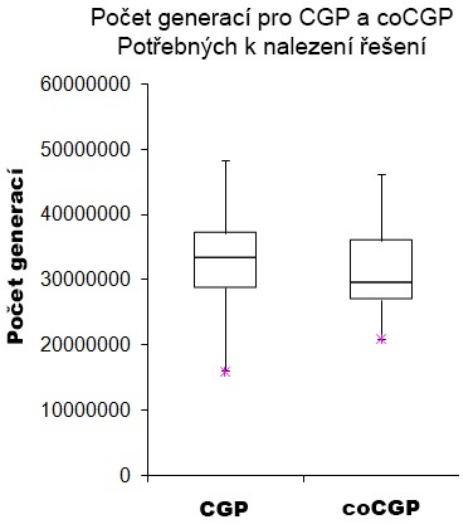
Nejlepší nalazené řešení pro úlohu F4 optimalizovanou na počet hradel mělo 104 logických hradel a je zobrazeno na obrázku C.1 v příloze C. Řešení optimalizované na zpoždění pomocí logical effort je zobrazeno na obrázku C.2 v příloze C a mělo zpoždění 44,64269τ.

5.7.3 Srovnání nalezeného řešení s konvenčními metodami návrhu

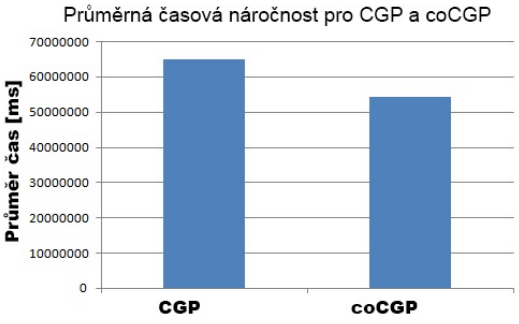
Řešení navržené pomocí konvenčních metod je zobrazeno na obrázku 5.42. Tento obrázek je zjednodušen o doplnění bloků pro poloviční sčítačky *HA* (každá obsahuje 2 logické hradla) a úplné sčítačky *FA* (každá obsahuje 5 logických hradel) z důvodu velkého množství logických hradel. Toto řešení obsahuje 58 logických hradel a má zpoždění 58,68472τ.



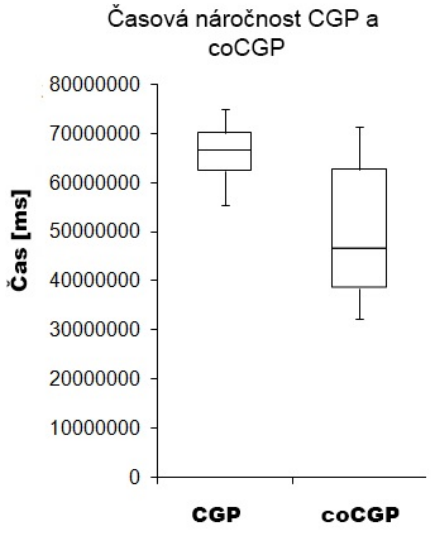
Obrázek 5.37: Graf úspěšnosti nalezení řešení CGP a coCGP pro F4.



Obrázek 5.38: Graf potřebného počtu generací k nalezení řešení v CGP a coCGP pro F4.



Obrázek 5.39: Graf časové náročnosti CGP a coCGP pro F4.



Obrázek 5.40: Graf časové náročnosti CGP a coCGP pro F4.

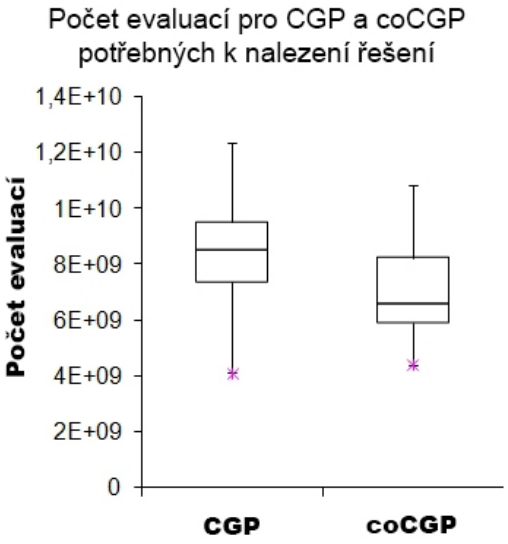
Z výsledků můžeme vidět, že co se týče optimalizace na počet logických hradel, konvenční návrh byl mnohem lepší – obsahoval skoro dvakrát méně hradel, než řešení nalezené pomocí coCGP. Může to být způsobeno krátkou dobou evoluce, jelikož pouze najít řešení trvalo většinu času evoluce. V praxi by bylo vhodnější, kdyby evoluce začínala z funkčního řešení (například navrženého pomocí konvenčních metod) a pouze se jej snažila optimalizovat.

Zato za úspěšnou optimalizaci můžeme prohlásit optimalizaci na zpoždění, kde řešení nalezené pomocí coCGP mělo o 14,04203τ menší zpoždění, než řešení navržené pomocí konvenčních metod. I zde lze předpokládat, že řešení by bylo lepší, kdyby evoluce začínala z funkčního řešení.

Shrnutí parametrů jednotlivých návrhu obvodů je uvedeno v tabulce 5.10.

Návrh řešení pomocí:	počet bloků	zpoždění logical effort
Konvenční metoda návrhu	58	58,68472τ
coCGP	104	44,64269τ

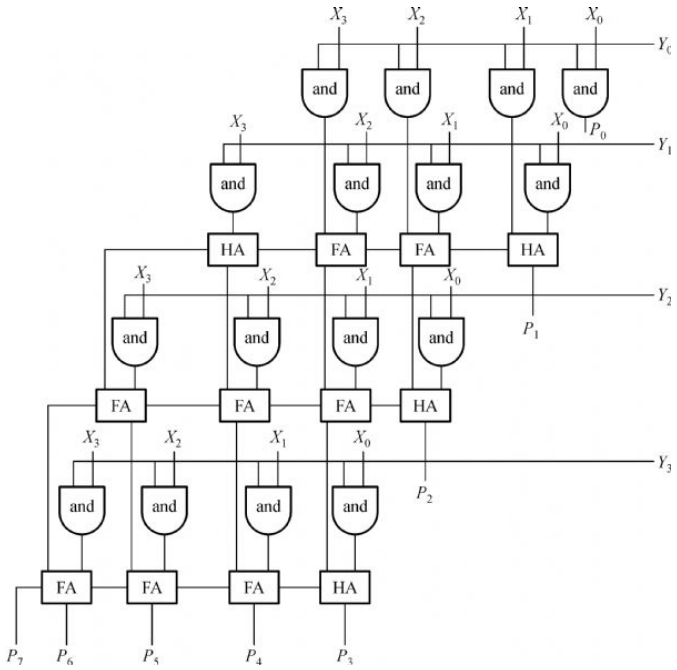
Tabulka 5.10: Tabulka srovnání parametrů obvodu mezi CGP a konvenční metodou návrhu pro úlohu F4.



Obrázek 5.41: Graf počtu evaluací CGP a coCGP pro F4.

5.7.4 Přibližné počítání

V tabulce 5.11 jsou srhnuty výsledky zlepšení parametrů obvodu (zpoždění, počet hradel) při snížení přesnosti výpočtu. V této úloze jsou už změny podstatné a nezanedbatelné, což je dáno složitostí úlohy. Obzvlášť za zmínku stojí snížení přesnosti o 30%, kdy parametry obvodu (počet hradel obvodu, zpoždění) jsou zlepšeny skoro na 1/3 hodnot při plné přesnosti.



Obrázek 5.42: 4 bitová násobička navržená pomocí konvenčních metod.

Snížení přesnosti o:	bloků / logical effort
10%	81 / 27,61675τ
15%	74 / 26,26913τ
20%	63 / 20,91769τ
25%	53 / 18,84941τ
30%	38 / 13,85549τ

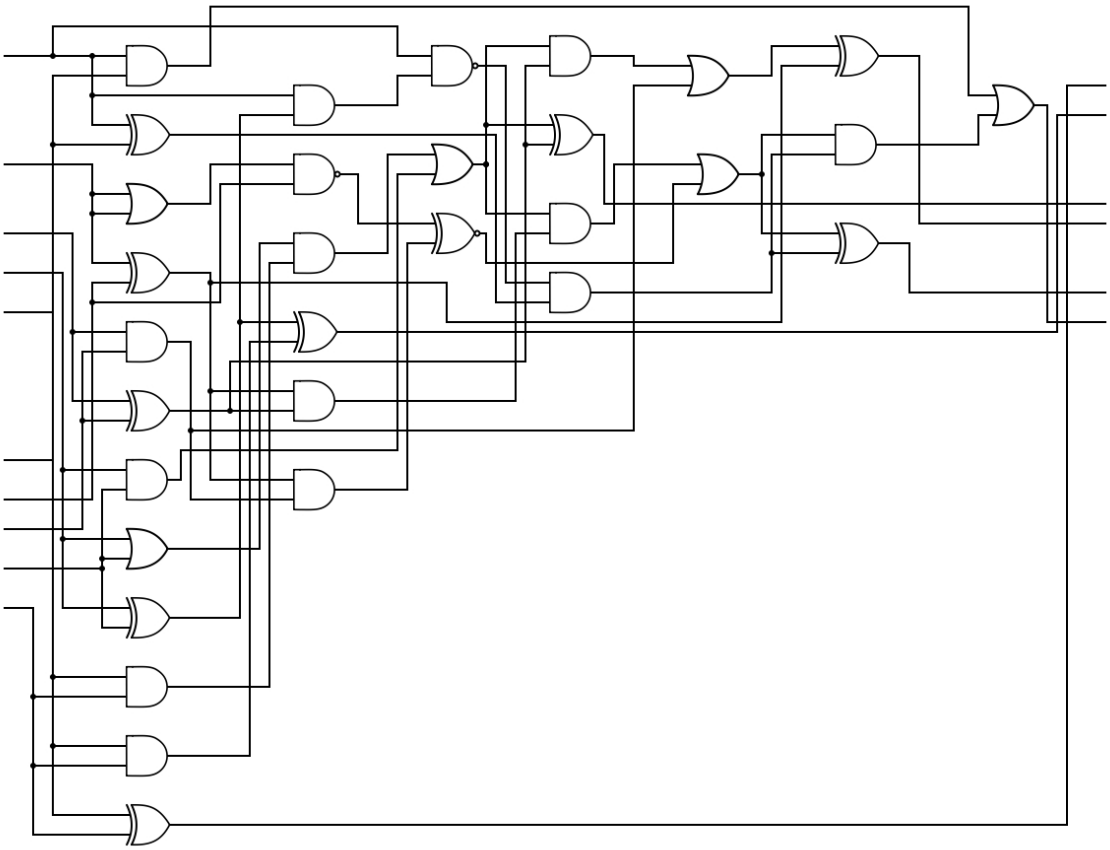
Tabulka 5.11: Tabulka zlepšení parametrů obvodu při snížení přesnosti výpočtu pro úlohu F4.

5.8 Úloha F5 – 5 bitová sčítačka

Úloha F5 již byla na tolik složitá, že CGP již nebylo schopno nalézt řešení v čase, který umožňuje vypracování této práce. Proto byla naiplementována možnost, začít z předloženého řešení. V této úloze tedy bylo předloženo funční řešení navrženo pomocí konvečních metod. Z tohoto důvodu zde nebude porovnání mezi koevolucí a klasickým CGP (není co testovat, jelikož plně funkční řešení je předloženo při začátku evoluce), ale pouze hledání optimálnějších řešení než je počáteční.

5.8.1 Nejlepší nalezené řešení úlohy

Při optimalizaci na počet použitých logických hradel se nepovedlo nalézt řešení, které by obsahovalo méně logických hradel, než má konvenční řešení zobrazené na obrázku 5.44, čili řešení, které bylo předloženo na začátku evoluce. Během optimalizace na zpoždění se povedlo vždy nalézt řešení, které mělo zpoždění menší. Nejlepší řešení mělo zpoždění 16,77611τ a 31 logických hradel. Toto řešení je zobrazeno na obrázku 5.43.



Obrázek 5.43: Nejlepší řešení 5 bit sčítačky nalezené pomocí coCGP optimalizované na zpoždění pomocí logical effort.

5.8.2 Srovnání nalezeného řešení s konvenčními metodami návrhu

Řešení navržené pomocí konvečních metod zobrazené na obrázku 5.44 má zpoždění 24,4469τ a obsahuje 22 logických hradel. Zde můžeme vidět, že CGP sice zvýšilo počet hradel o skoro 50%, ale zpoždění je o 7,670τ menší.

Shrnutí parametrů jednotlivých návrhu obvodů je uvedeno v tabulce 5.12.

Návrh řešení pomocí:	počet bloků	zpoždění logical effort
Konvenční metoda návrhu	22	24,4469τ
coCGP	31	16,77611τ

Tabulka 5.12: Tabulka srovnání parametrů obvodu mezi CGP a konvenční metodou návrhu pro úlohu F5.

5.8.3 Přibližné počítání

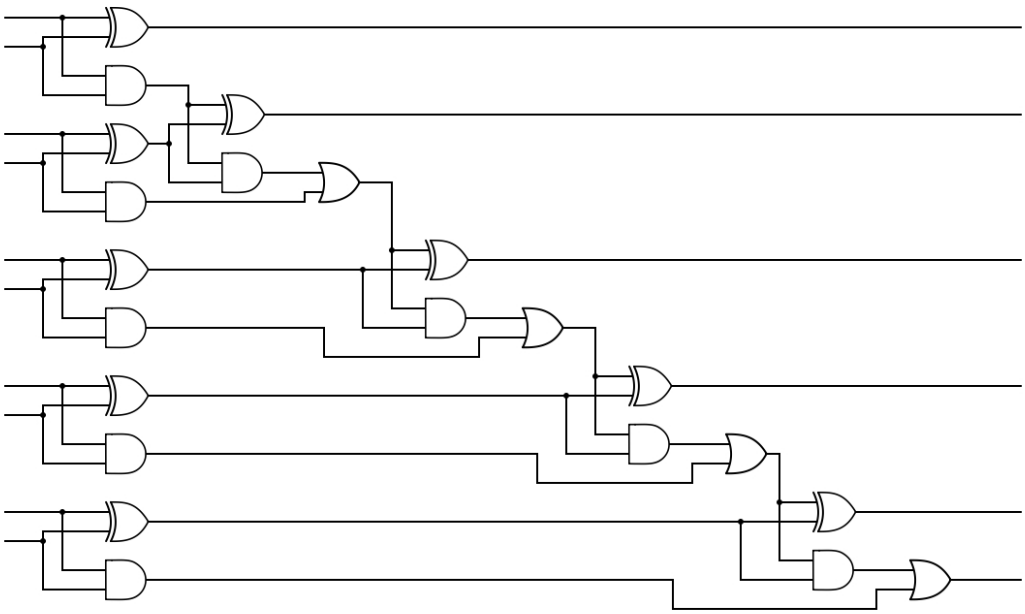
V tabulce 5.13 jsou srhnuty výsledky zlepšení parametrů obvodu (zpoždění, počet hradel) při snížení přesnosti výpočtu.

Snížení přesnosti o:	bloků / logical effort
10%	21 / 8,909355τ
15%	20 / 6,399705τ
20%	19 / 4,301126τ
25%	18 / 4,288414τ
30%	17 / 4,288414τ

Tabulka 5.13: Tabulka zlepšení parametrů obvodu při snížení přesnosti výpočtu pro úlohu F5.

5.9 Shrnutí

V úlohách F1 a F2 koevoluce nepřináší žádnou výhodu oproti klasickému CGP, protože režie s koevolucí spojená si žádá více výpočetních zdrojů, než ušetří v takto jednoduchých úlohách. Při složitějších úlohách F3 a F4, které jsou již dostatečně složité, dosahuje koevoluce teoretických výhod a počet evaluací obvodu sníží až o 30% a časovou náročnost přibližně o 20%. V úloze F5 koevoluce se neprojeví, jelikož v této úloze bylo vycházeno z plně funkčního obvodu, tudíž evoluce začíná rovnou s plnou trénovací sadou. Ale i v této úloze si CGP našlo své místo a to v optimalizaci obvodu a také hledání řešení, kterému byla snížena přesnost výpočtu.



Obrázek 5.44: 5 bitová sčítačka navržená pomocí konvenčních metod.

Kapitola 6

Závěr

Tato diplomová práce se zabývala problematikou genetického programování, koevoluce prediktorů fitness a evolučního návrhu číslicových obvodů. Na tomto základě jsem navrhnul funkční program, který umožňuje provádět evoluční návrh číslicových obvodů pomocí kartézského genetického programování s využitím koevoluce prediktorů fitness. Program kromě návrhu obvodu umožňuje výsledný obvod optimalizovat na počet hradel nebo na zpoždění, za pomocí metody logical effort.

Čas potřebný k nalezení řešení je závislý na nastavení parametrů kartézského genetického programování a koevoluce. Jelikož tyto parametry jsou pro každou úlohu různé, tak pro jejich hledání byla provedena sada experimentů, která hledá co nejvýhodnější nastavení těchto parametrů. Při porovnání pak byla použita stejná nastavení.

V jednoduchých úlohách (F1 a F2) koevoluce nepřinesla žádné výrazné zlepšení, jelikož režie u těchto jednoduchých úloh byla větší, než čas ušetřený snížením počtu evaluací. Zatímco ve složitějších úlohách (F3 a F4) se podařila snížit celková doba potřebná k výpočtu oproti řešení bez koevoluce přibližně o 20% a počet evaluací obvodu byl snížen o 30%, což je podstatná změna. Pro úlohu F5 koevoluce nehraje žádnou roli, jelikož bylo potřeba začít z plně funkčního řešení a docházelo pouze k optimalizace plně funčního řešení.

Při řešení složitějších úloh, bylo potřeba doimplementovat programu možnost definovat na vstup řešení, od kterého má evoluce započít. To z důvodu, že prohledávaný prostor byl tak velký, že nebylo možné v rozumném čase najít řešení z náhodného obvodu.

Aplikace také umožňuje výsledný obvod optimalizovat buďto na počet hradel nebo na zpoždění. V porovnání s konvenčními metodami návrhu, kartézské genetické programování lehce zaostávalo v optimalizaci na počet hradel oproti konvenčnímu návrhu. Kromě úlohy F1, kde se počet hradel povedlo snížít a úlohy F2, ve které byl počet hradel totožný, se CGP nepovedlo najít řešení, které by obsahovalo méně logických hradel. V optimalizaci na zpoždění je situace zcela opačná. CGP bylo schopno vždy najít řešení, které mělo nižší zpoždění, než řešení navržené konvenčními metodami návrhu. A to i v případě úlohy F5, kde CGP vycházelo z funčního řešení. CGP mnohdy nalezlo zcela jedinečné řešení, které obsahovalo netradiční vazby. CGP tedy může být použito i pro vylepšení obvodů, které byly navrženy konvenčními metodami návrhu.

Při snížení přesnosti výpočtů bylo poukázáno, že teoretických přínosů (snížení počtu logických hradel, zmenšení zpoždění obovdu) bylo dosaženo i v praktických úlohách. Již při snížení přesnosti výpočtu o 10% byly vlastnosti vylepšeny ve složitých úlohách F4 resp. F5 o 20% resp. 10%. Nejmarkatnější rozdíl lze pozorovat v úloze F4, jelikož je složena z největšího počtu logických hradel, kde při snížení přesnosti o 30% byly parametry obvodu (počet logických hradel, zpoždění obvodu) zlepšeny o 2/3 z hodnot, kterých obvod dosahoval

při plné přesnosti. Vlastnosti (počet logických hradel, zpoždění obvodu) byly vylepšeny jak pro případy, kdy evoluce hledala (konstruovala) nové řešení, tak i pokud již začínala z plně funkčního obvodu. V této aplikaci CGP může také nalézt uplatnění, jelikož obvody, které byly takto navrženy by konvenčními metodami nebyly objeveny a mohou dosáhnout parametrů.

Během zpracování tohoto tématu jsem si nastudoval přístupy evolučně inspirovaných metod pro návrh číslicových obvodů, možnostmi optimalizací číslicových obvodů a při implementaci jsem si práci s těmito metodami i prakticky vyzkoušel.

Literatura

[1] Java Performance Tuning Guide - Java performance tuning guide - various tips on improving performance of your Java code. [Online; navštíveno 25.3.2016].
URL `<http://java-performance.info/>`

[2] WWW stránky: Nástroje pro kartézské genetické programování. [Online; navštíveno 25.3.2016].
URL `<http://www.fit.vutbr.cz/~vasicek/cgp/tools/>`

[3] WWW stránky: VLSI Computer Architecture Research Group. [Online; navštíveno 25.3.2016].
URL `<http://vlsiarch.ecen.okstate.edu/>`

[4] De Jong, K.: Generalized Evolutionary Algorithms. In *Handbook of Natural Computing*, Springer Berlin Heidelberg, 2012, ISBN 978-35-409-2909-3, s. 625–635.

[5] Hynek, J.: *Genetické algoritmy a genetické programování*. Praha: Grada, 2008, ISBN 978-80-247-2695-3, 200 s.

[6] Šikulová, M.; Sekanina, L.: Acceleration of Evolutionary Image Filter Design Using Coevolution in Cartesian GP. *Lecture Notes in Computer Science*, ročník 2012, č. 7491, 2012: s. 163–172, ISSN 0302-9743.

[7] Šikulová, M.; Sekanina, L.: Coevolution in Cartesian Genetic Programming. *Lecture Notes in Computer Science*, ročník 2012, č. 7244, 2012: s. 182–193, ISSN 0302-9743.

[8] Kvasnička, J.: *Evolučné algoritmy*. Bratislava: STU, 2000, ISBN 802-27-1377-5, 223 s.

[9] Miller, J.: *Cartesian genetic programming*. Springer, 2011, ISBN 978-36-421-7310-3, 351 s.

[10] Popovici, E.; Bucci, A.; Wiegand, R.; aj.: Coevolutionary Principles. In *Handbook of Natural Computing*, Springer Berlin Heidelberg, 2012, ISBN 978-35-409-2909-3, s. 987–1033.

[11] Schmidt, M. D.; Lipson, H.: Coevolution of Fitness Predictors. *IEEE Trans. on Evolutionary Computation*, ročník 12, č. 6, 2008: s. 736–749, ISSN 1089-778X.

[12] Sekanina, L.: *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Praha: Academia, 2009, ISBN 978-80-200-1729-1, 328 s.

[13] Sutherland, I.: *Logical effort : designing fast CMOS circuits*. San Francisco, Calif: Morgan Kaufmann Publishers, 1999, ISBN 155-86-0557-6, 26 s.

[14] Turner, A. J.; Miller, J. F.: WWW stránky: Introducing A Cross Platform Open Source Cartesian Genetic Programming Library. [Online; navštíveno 25.3.2016].
URL <http://www.cgplibrary.co.uk/>

[15] Vanneschi, L.; Poli, R.: Genetic Programming – Introduction, Applications, Theory and Open Issues. In *Handbook of Natural Computing*, Springer Berlin Heidelberg, 2012, ISBN 978-35-409-2909-3, s. 709–739.

[16] Vasicek, Z.: WWW stránky: Cartesian Genetic Programming. [Online; navštíveno 25.3.2016].
URL <http://www.fit.vutbr.cz/~vasicek/cgp/>

[17] Vasicek, Z.; Sekanina, L.: Evolutionary Approach to Approximate Digital Circuits Design. *IEEE Trans. on Evolutionary Computation*, ročník 19, č. 3, 2015: s. 432–444, ISSN 1089-778X.

[18] Weste, N.: *CMOS VLSI design : a circuits and systems perspective*. Boston: Addison Wesley, 2011, ISBN 032-15-4774-8, 864 s.

[19] Whitley, D.; Sutton, A.: Genetic Algorithms – A Survey of Models and Methods. In *Handbook of Natural Computing*, Springer Berlin Heidelberg, 2012, ISBN 978-35-409-2909-3, s. 637–671.

Příloha A

Spuštění programu

```
cgp.jar [arguments] [heuristic]
Arguments:
-h   print help
-v   verbose (print statistics data)
-f [fileName]   defines input file name
Parameters for cgp:
-r [n]     defines number n of rows for cgp. Default is: 6
-c [n]     defines number n of cols for cgp. Default is: 5
-l [n]     defines number n of l_back for cgp. Default is: 3
-g [n]     defines generation size for cgp. Default is: 250000
-p [n]     defines population size for cgp. Default is: 6
-ms [n]    defines maximum number n of genes whitch can be mutated. Default is: 5
-eps [n]   defines fitness toleration for cgp. Default is: 1.0
-am [n]    active mutation
Parameters for coevolution:
-gs [n]    defines size of predictors array. Default is: 50
-gp [n]    defines size of predictors population. Default is: 32
-gt [n]    defines size of trainers set. Default is: 8
-gce [n]   defines computation effort (how many cgp cycles to one predictors
           evolution cycle). Default is: 36000
-gct [n]   defines cycles for add new trainers (how many predictors evolution
           cycles to add new trainers). Default is: 100
-geps [n]  defines fitness toleration for coevolution. Default is: 0.9

Atributes for creating new population of size [P] for coevolution. Only parents and
children can be defined. Rest is used for ramdom members.
-gmp [n]   defines number of best parent for next predictor population: [P] / [n].
           Default is: 4
-gmc [n]   defines number of children for next predictor population: [P] / [n].
           Default is: 2

Others:
-arthm        defines arithmetic weight durring evaluation. Default is bit XOR.
-start-chrom  defines start chromozome for evolution.
-dop          disable optimalization of circuit.
```

```
-dco        disable coevolution (run only cgp).
-simpled    optimalize circuit to delay (simple). Default is optimize to number
            of blocks.
-logeff     optimalize circuit to delay (logical effort). Default is optimize
            to number of blocks.
-pcgpv      print result chromosome in cgp viewer format.
```

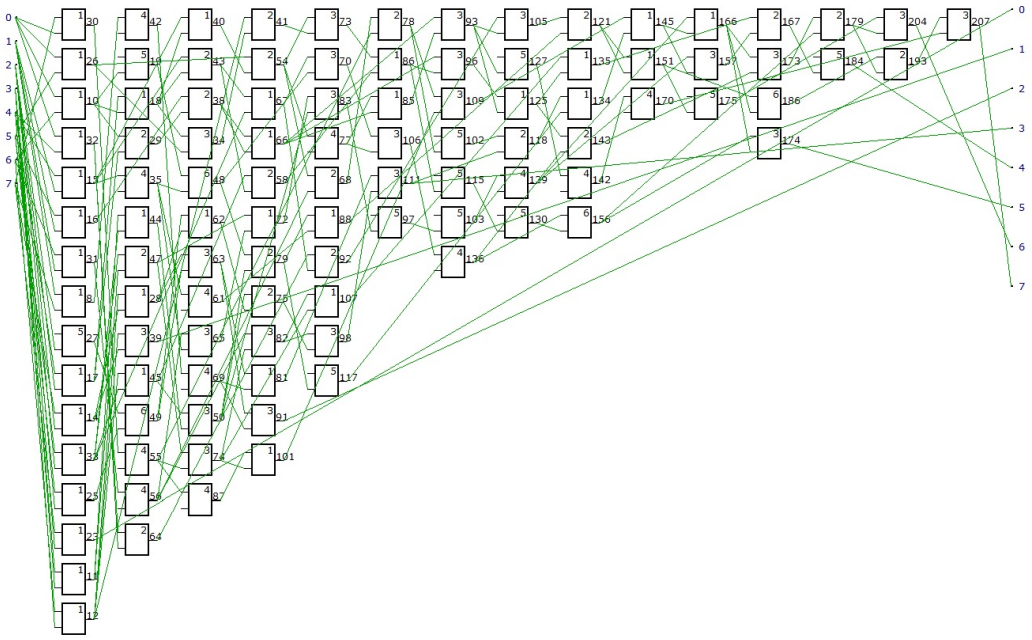
Příloha B

Příklad výstupu statistických dat

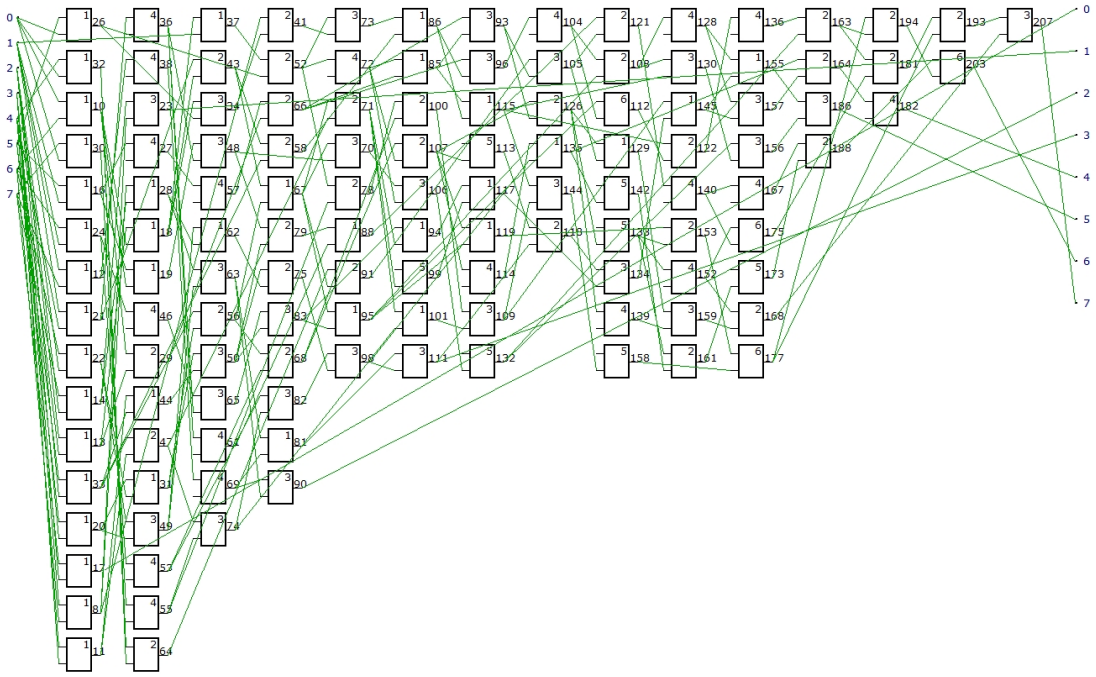
```
fitness: 50.0 cycle: 2500
fitness: 52.0 cycle: 4971
fitness: 54.0 cycle: 12571
fitness: 56.0 cycle: 13902
fitness: 58.0 cycle: 20616
Full_evaluation: 521463 first_best: 1024.0409049664052 first_number_of_blocks:
28 1024.0409049664052 0 Optimalization_from: 625455 Best: 1024.0475157447113
best_number_of_blocks: 40 generation: 5000000 Time: 708368
```

Příloha C

Nejlepší řešení úlohy F4



Obrázek C.1: Nejlepší řešení 4 bitové násobičky nalezené pomocí coCGP optimalizované na počet log. hradel.



Obrázek C.2: Nejlepší řešení 4 bitové násobičky nalezené pomocí coCGP optimalizované na zpoždění pomocí logical effort.

Příloha D

Obsah CD

- Tento dokument v elektronické podobě
- Zkompilovaná aplikace
- Zdrojové kódy aplikace
- Testovací úlohy (pravdivostní tabulky)