VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH HAŠOVACÍCH FUNKCÍ POMOCÍ GENETICKÉHO PROGRAMOVÁNÍ

EVOLUTIONARY DESIGN OF HASH FUNCTIONS USING GENETIC PROGRAMMING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ MICHALISKO**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Michalisko Tomáš**
Program: Informační technologie
Název: **Návrh hašovacích funkcí pomocí genetického programování**
Hash Function Design Using Genetic Programming
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s metodami návrhu a evaluace hašovacích funkcí.
2. Zpracujte studii mapující využití evolučních algoritmů pro návrh hašovacích funkcí.
3. Navrhněte metodu pro automatizovaný návrh hašovacích funkcí pomocí genetického programování. Zaměřte se na hašování síťových toků.
4. Navrženou metodu implementujte a ověřte na zadaných datech.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Tato práce se zabývá automatizováním návrhu hašovacích funkcí. K tomu využívá kartézské genetické programování. Zvolenou metodou pro řešení kolizí je kukaččí hašování. Byly porovnány tři varianty zakódování hašovacích funkcí. Experimenty byly prováděny nad datovou sadou obsahující síťové toky. V rámci experimentů bylo nalezeno vhodné nastavení parametrů této metody včetně množiny funkcí. Nejlepší vyvinuté hašovací funkce dosahují srovnatelných výsledků jako funkce navržené odborníky. Hlavním zjištěním je, že nejlepších výsledků dosahují hašovací funkce tvořené 64bitovými operacemi.

Abstract

This thesis deals with automated design of hash functions using Cartesian genetic programming. The chosen method for collision resolution is cuckoo hashing. Three variants of hash function encodings were compared. Experiments were performed with datasets containing network flows. The most suitable parameters of CGP, including the function set, were determined. The best evolved hash functions achieved comparable results to the functions designed by experts. The main finding is that hash functions consisting of 64-bit operations achieve the best results.

Klíčová slova

Kartézské genetické programování, hašovací funkce, kukaččí hašování.

Keywords

Cartesian Genetic Programming, Hash Function, Cuckoo Hashing.

Citace

MICHALISKO, Tomáš. *Evoluční návrh hašovacích funkcí pomocí genetického programování.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Evoluční návrh hašovacích funkcí pomocí genetického programování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tomáš Michalisko
8. května 2022

Poděkování

Děkuji prof. Ing. Lukáši Sekaninovi, Ph.D. za jeho ochotu a odborné vedení práce.

Obsah

Kapitola 1

Úvod

Hašovací funkce jsou velmi důležité pro fungování mnohých aplikací. Používají se zejména tam, kde je potřeba provádět rychlé vyhledávání, což vyžadují například databáze, internetové vyhledávače nebo zařízení monitorující síťový provoz. Aby hašovací funkce byly efektivní, musí splňovat určitá kritéria. Jedno z nich říká, že musí produkovat co nejméně kolizí [10]. Kukaččí hašování je moderní metoda, která se umí s kolizemi do jisté míry vypořádat, díky tomu, že využívá několik hašovacích funkcí najednou [28].

Návrh hašovacích funkcí je poměrně složitý, což vytváří prostor pro aplikaci genetického programování. To umožňuje generovat funkce podle požadavků aplikační domény. Těmi jsou například rychlost a kvalita hašování nebo omezená instrukční sada. Další výhodou je, že dovoluje vytvářet páry hašovacích funkcí vyladěné pro použití v kukaččím hašování.

Bylo publikováno již několik studií, které se zabývají využitím genetického programování pro automatizaci návrhu hašovacích funkcí [9], [33], [21], [16], [20]. Ukázalo se, že je možné využít hned několik různých variant genetického programování a že funkce vytvořené pomocí evoluce mohou dosahovat lepších výsledků než funkce navržené klasickým způsobem. Avšak stále existuje prostor pro další zkoumání.

Cílem této práce je navrhnout, implementovat a experimentálně vyhodnotit metodu pro automatizovaný návrh hašovacích funkcí. Konkrétně se práce zaměří na kartézské genetické programování, u kterého porovná různé metody mutace. A vyzkouší, zdali je lepší vytvářet hašovací funkce použitím 16bitových, 32bitových nebo 64bitových operací.

V kapitole 2 budou představeny hašovací funkce v kontextu hašovacích tabulek, včetně moderního způsobu řešení kolizí – kukaččího hašování. Následující kapitola 3 vysvětlí základní stavení kameny genetického programování jakožto metody pro automatizovaný návrh počítačových programů. Poté v kapitole 4 bude přiblíženo kartézské genetické programování včetně dalších metod mutace. Dále v kapitole 5 budou porovnány přístupy různých autorů k návrhu hašovacích funkcí pomocí evolučních algoritmů. Kapitola 6 přiblíží volbu metod použitých v této práci a detaily jejich implementace. Předposlední kapitola 7 popíše experimentální vyhodnocení a porovnání použitých metod. Kapitola 8 shrne výsledky celé práce.

Kapitola 2

Hašování

Cílem této kapitoly je seznámit čtenáře s principem činnosti hašovacích funkcí a jejich typickým použitím v hašovacích tabulkách. Sekce 2.1 vysvětluje, co jsou to hašovací funkce a jak se využívají v hašovacích tabulkách. Následující sekce 2.2 popisuje metodu pro řešení kolizí – kukaččí hašování. V sekci 2.3 je uvedeno, jaké kritéria by měla splňovat dobrá hašovací funkce. V poslední sekci 2.4 se nachází přehled populárních hašovacích funkcí.

2.1 Hašovací funkce a tabulky

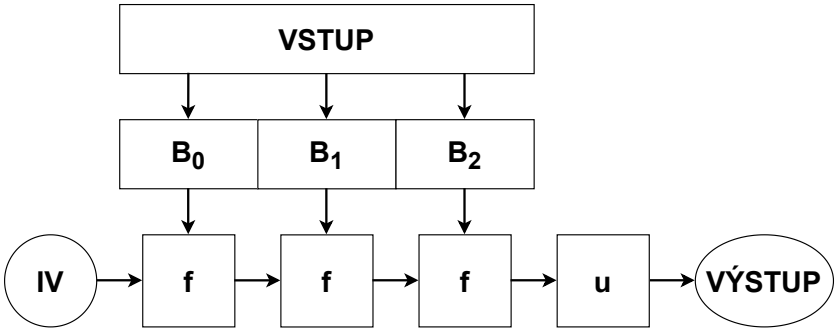
Tato sekce vychází z [15]. Slovník je abstraktní datová struktura, která slouží pro ukládání záznamů, což jsou uspořádané dvojice (k, v), kde k je jednoznačný identifikátor neboli klíč a v je příslušná hodnota. Mezi základní operace, které tato abstraktní datová struktura poskytuje, patří vkládání, vyhledávání a mazání záznamů. Slovníky se využívají především pro rychlé vyhledávání.

Jednou z nejefektivnějších implementací slovníku je hašovací tabulka. Operace prováděné nad hašovací tabulkou mají v nejhorším případě časovou složitost $O(n)$, kde n je počet záznamů v tabulce. Avšak můžeme očekávat, že typicky budou tyto operace provedeny s časovou složitostí $O(1)$.

Základem hašovací tabulky je pole T o velikosti N, do kterého se ukládají jednotlivé záznamy. Velikost tohoto pole odpovídá požadované kapacitě hašovací tabulky. Kapacita se typicky volí jako mocnina dvou nebo jako prvočíslo. Druhou důležitou částí hašovací tabulky je hašovací funkce h, pomocí které se určuje, na jakou pozici pole T se má uložit daná dvojice (k, v). Výstup hašovací funkce se nazývá haš.

Většina hašovacích funkcí pracuje na principu Merkleovy–Damgårdovy konstrukce (viz obrázek 2.1) [24][6]. Prvním krokem je inicializace vnitřního stavu (IV). Poté se vstupní klíč rozdělí do několika stejně velkých bloků $(B_0, B_1, \ldots, B_n)$, které jsou postupně kombinovány s vnitřním stavem pomocí kompresní funkce (f). Pokud je poslední blok menší, tak je doplněn na požadovanou velikost. Posledním krokem může být úprava výsledku pomocí uzavírací funkce (u).

Takto vypočítaná hodnota má typicky velikost 32, 64 nebo 128 bitů. Většinou je potřeba tuto hodnotu redukovat do rozsahu $\langle 0; N-1)$, kde N je kapacita hašovací tabulky. K tomu se nejčastěji používá operace modulo. Takže výsledný index se vypočítá jako $h(k)\ mod\ N$. Při použití některých hašovacích funkcí je dobré, aby N bylo prvočíslo [10].



Obrázek 2.1: Merkleova–Damgårdova konstrukce.

Při vkládání nových záznamů do hašovací tabulky nastane situace, kdy se dva odlišné klíče k_1 a k_2 namapují na stejnou pozici neboli $h(k_1) = h(k_2)$. Tato situace se nazývá kolize. Klíče, které mají stejnou hodnotu hašovací funkce, se nazývají synonyma. Jak často bude ke kolizím docházet závisí hlavně na kapacitě hašovací tabulky a na kvalitě použité hašovací funkce. Při implementaci hašovací tabulky je nutné s kolizemi počítat a použít některou z metod pro jejich vyřešení:

Zřetězení synonym – každá pozice pole T představuje lineární seznam, do kterého jsou ukládány příslušná synonyma. Pokud má hašovací tabulka kapacitu N a obsahuje n záznamů, tak průměrná délka těchto seznamů je n/N.

Otevřená adresace – nové záznamy jsou ukládány přímo do pole T a v případě, že se nový záznam namapuje na již obsazenou pozici, tak se postupně prochází alternativní pozice pro uložení tohoto záznamu, dokud není nalezena volná pozice. Díky tomu není potřeba pomocných datových struktur, ale komplikuje se implementace základních operací. Existuje několik způsobů, jak volit alternativní pozice.

Prvním z nich je **lineární prohledávání**. Pro záznam s klíčem k, který se mapuje na pozici p, se alternativní pozice vypočítají jako $(p+i)\ mod\ N$, pro $i = 1, 2, 3, \ldots$. Nevýhodou této metody je to, že způsobuje shlukování záznamů.

Další variantou je **kvadratické prohledávání**, kde se alternativní pozice vypočítají jako $(p + i^2)\ mod\ N$, pro $i = 1, 2, 3, \ldots$. Tato metoda nezpůsobuje shlukování jako lineární prohledávání, ale způsobuje vlastní druh shlukování, kterému se říká sekundární shlukování.

Varianta, která na rozdíl od dvou předchozích metod shlukování nezpůsobuje, se nazývá **dvojité hašování** a spočívá v tom, že alternativní pozice se vypočítají pomocí druhé hašovací funkce h' jako $(p + i \cdot h'(k))\ mod\ N$, pro $i = 1, 2, 3, \ldots$.

Po odstranění záznamu by se hašovací tabulka měla vrátit do stavu, jako by daný záznam nebyl nikdy vložen. Což by v případě otevřené adresace znamenalo přesun několika záznamů. To by však nebylo efektivní a proto se zvolí speciální hodnota, kterou se nahrazují odstraněné záznamy. Z hlediska operace pro vkládání se tato speciální hodnota považuje jako volná pozice.

Kukaččí hašování spočívá v tom, že pole T, do kterého se ukládají záznamy, se rozdělí na dvě menší pole. Každé z těchto polí využívá jinou hašovací funkci. Při vkládání se záznamy přesouvají z jednoho pole do druhého, dokud není nalezena volná pozice. Výhodou této metody je, že poskytuje operaci vyhledání s konstantní časovou složitostí, jak bude detailněji popsáno v sekci 2.2.

Kromě hašovacích tabulek mají hašovací funkce i další využití [4]. Hašovací funkce se dají uplatnit například při vyhledávání obrázků, které jsou podobné nějakému referenčnímu. Jelikož obrázky obsahují poměrně hodně dat, tak by bylo časově velmi náročné porovnávat je v originální podobě. Proto se pro zrychlení pracuje pouze s jejich haši. Další oblastí pro aplikaci hašovacích funkcí jsou počítačové sítě. Zde se používají při vyhledávání v tabulkách IP adres nebo při monitorování provozu v síti. Důležitou doménou je také kybernetická bezpečnost. Pro účely zabezpečení jsou vytvářeny speciální kryptografické hašovací funkce, které musí splňovat určitá bezpečnostní kritéria a často pracují s veřejnými a tajnými klíči. Tyto funkce slouží pro zajištění integrity a ověření původu dat nebo pro vytváření digitálních podpisů.

2.2 Kukaččí hašování

Kukaččí hašování je metoda pro řešení kolizí v hašovacích tabulkách. Tato metoda byla představena v roce 2001 a jejími autory jsou Rasmus Pagh a Flemming Friche Rodler. Název je odvozen od chování parazitických kukaček, které po vylíhnutí vytlačí ostatní vajíčka z hnízda. Tato sekce čerpá z [28].

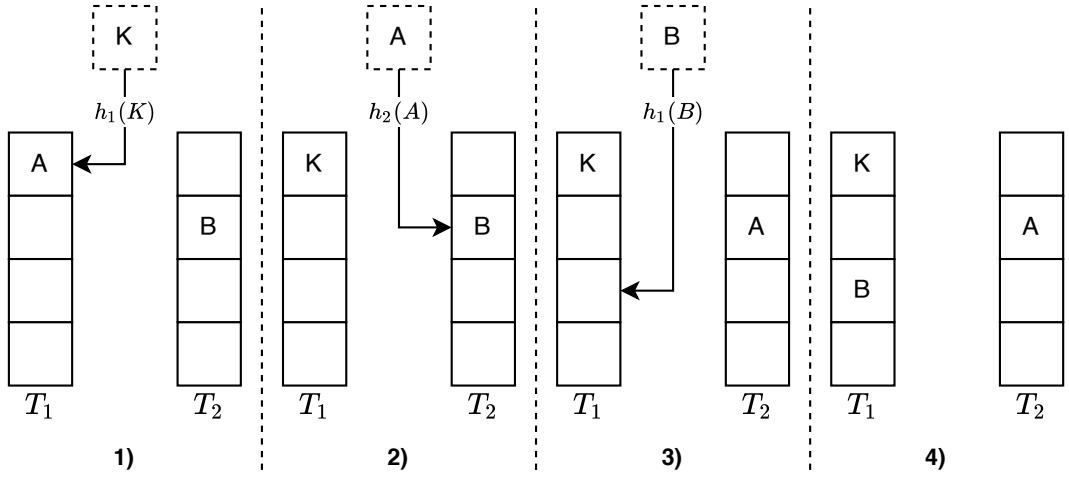
Hašovací tabulka, která využívá kukaččí hašování, se skládá ze dvou polí o stejné velikosti, T_1 a T_2, které slouží pro ukládání záznamů. Pro mapování záznamů do každého z těchto polí se používá jiná hašovací funkce, h_1 pro T_1 a h_2 pro T_2. Záznam s klíčem x může být uložen buď v T_1 na pozici $h_1(x)$, nebo v T_2 na pozici $h_2(x)$. Z toho vyplývá, že operace vyhledání záznamu má v nejhorším případě konstantní časovou složitost $O(1)$, což je hlavní výhoda kukaččího hašování.

Vkládání spočívá ve vytlačování ostatních záznamů, až dokud každý záznam nenajde své "hnízdo". Procedura pro vkládání nových záznamů je popsáná algoritmem 1. Operátor $\leftrightarrow$ značí výměnu hodnot dvou proměnných. Speciální hodnota $\perp$ reprezentuje neobsazenou pozici. Při vkládání záznamu s klíčem x se jako první zkontroluje, jestli hašovací tabulka daný klíč ještě neobsahuje (řádek 1). Pokud ne, tak je tento záznam vložen do T_1 na pozici $h_1(x)$ (ř. 3). Pokud tato pozice nebyla prázdná (ř. 4), tak původní záznam je přemístěn do T_2, kde opět vytlačí původní záznam (ř. 5). Tento postup se opakuje do té doby, než každý záznam nenajde své místo. Obrázek 2.2 demonstruje činnost algoritmu.

Je možné, že dojde k zacyklení, kdy nebude možné pomocí přemisťování uložit všechny záznamy. Proto se nastavuje konstanta $MaxLoop$, která určuje maximální počet přemístění. Pokud je tento počet překročen, tak je vkládání prohlášeno za neúspěšné a je potřeba zvolit nové hašovací funkce a přehašovat všechny záznamy, případně zdvojnásobit kapacitu hašovací tabulky (ř. 7).

Algoritmus 1: insert(x), převzato z [28]

```
1 if lookup(x) then return
2 for i = 1, 2, 3, ..., MaxLoop do
3 |   x ↔ T1[h1(x)]
4 |   if x == ⊥ then return
5 |   x ↔ T2[h2(x)]
6 |   if x == ⊥ then return
7 rehash()
8 insert(x)
```



Obrázek 2.2: Příklad vložení záznamu s klíčem K. V 1. kroku je záznam K vložen do T_1 na pozici $h_1(K)$, kde vytlačí z. A. Ve 2. kroku je z. A vložen do T_2 na pozici $h_2(A)$, kde vytlačí záznam B. Ve 3. kroku je z. B vložen do T_1 na pozici $h_1(B)$, která je volná. Krok 4 ukazuje stav po úspěšném vložení záznamu s klíčem K.

2.3 Požadavky na kvalitu

Kvalitní hašovací funkce, které nejsou určeny pro použití v kryptografii, by měly splňovat tato čtyři kritéria [10]:

- **Odolnost proti kolizím**. Hašovací funkce by měla produkovat co nejméně kolizí, protože kolize jsou jedním z hlavních důvodů, které vedou ke zpomalení aplikací využívající hašovací funkce. Odolnost vůči kolizím je závislá na charakteru hašovaných dat.
- **Rozložení výstupních hodnot**. Je důležité, aby výstupní hodnoty měly rovnoměrné rozložení. Každá možná výstupní hodnota by měla mít stejnou pravděpodobnost výskytu nezávisle na rozložení vstupních dat. Při nesplnění tohoto kritéria se výstupní hodnoty mohou shlukovat, což může vést k problémům. Tohle kritérium je také závislé na charakteru hašovaných dat.
- **Lavinový efekt**. Splnění tohoto kritéria by v ideálním případě znamenalo, že při změně jediného bitu ve vstupní hodnotě se každý bit ve výstupní hodnotě změní s pravděpodobností 50 %. Malá změna na vstupu způsobí velkou změnu na výstupu. Čím více se hašovací funkce přiblíží k tomuto ideálu, tím náhodnější výstupy produkuje. Tohle kritérium nezávisí na hašovaných datech, což zjednodušuje jeho ověření.
- **Rychlost**. Hašovací funkce by měly být co nejrychlejší s ohledem na dodržení výše zmíněných kritérií. Aby byly rychlé, tak by měly obsahovat malý počet instrukcí a použité instrukce by měly být efektivní z hlediska spotřeby procesorového času. Rychlost hašovacích funkcí silně závisí na architektuře procesoru, na kterém jsou vyhodnocovány.

2.4 Přehled hašovacích funkcí

Tento seznam vychází z [4] a [33].

- FNV [12]. Jde o velmi jednoduchou hašovací funkci, používanou například v systémech DNS, emailových serverech nebo anti-spamových filtrech.
- Lookup3 [19]. Jedna z nejdůležitějších hašovacích funkcí produkující 32bitové haše.
- SuperFastHash [17]. Hlavní inspirací pro tuto funkci byly Lookup3 a FNV. Mezi přednosti této funkce patří především její rychlost.
- APartow [29]. Autorem této funkce je Arash Partow, který ji publikoval ve své sbírce hašovacích funkcí.
- DJBX33A [29]. Vhodná zejména pro práci s krátkými řetězci.
- BKDR [29]. Pochází z knihy book "The C Programming Language"od Briana Kernighana and Dennise Ritchieho.
- DEK [22]. Autorem této funkce je Donald E. Knuth. Jedná se o jednu z prvních hašovacích funkcí, která je i přes svou jednoduchost stále populární.
- MurmurHash3 [1]. Populární hašovací funkce, která dosahuje perfektního lavinového efektu.
- CityHash [31]. Představená společností Google. Autoři se inspirovali především funkcí MurmurHash. Optimalizována pro rychlost na moderních procesorech.
- FarmHash [30]. Vznikla jako nástupce CityHash. Obsahuje podporu pro různé architektury procesorů.
- xxHash [5]. Extrémně rychlá hašovací funkce, jejímž limitem je rychlost paměti RAM. Používá se například v databázích MySQL a MariaDB.
- HighwayHash [35]. Velmi rychlá hašovací funkce, která adresuje bezpečnostní nedostatky funkcí CityHash a FarmHash.

Experimentální vyhodnocení a porovnání těchto hašovacích funkcí lze nalézt v [10] a [33]. Jako nejlépe hodnocené funkce z hlediska počtu kolizí, lavinového efektu a rychlosti, autoři považují MurmurHash2, SuperFastHash, lookup3 a HighwayHash. Funkce FNV, DJBX33A, BKDR a DEK jsou důležité spíše z historického hlediska a autoři doporučují raději volit jiné hašovací funkce. Pro testování kvality a rychlosti hašovacích funkcí byl vytvořen nástroj SMhasher[1], který navíc obsahuje rozsáhlou tabulku ohodnocených funkcí.

[1] https://github.com/rurban/smhasher

Kapitola 3

Genetické programování

Původní genetické algoritmy používaly ke kódování kandidátních řešení řetězce bitů, které měly předem stanovenou délku. Ukázalo se, že tento způsob není vhodný pro kódování počítačových programů, které můžou mít teoreticky neomezenou délku. Tento problém řeší J. Koza a jeho genetické programování [23]. Obdobně jako i ostatní evoluční algoritmy, genetické programování využívá základních principů Darwinovy evoluce. Začíná náhodně vygenerovanou populací kandidátních řešení a iterativním stochastickým procesem prohledává prostor všech kandidátů. Jednotlivé iterace se nazývají generace a k prohledávání využívá kombinaci dvou genetických operátorů. Prvním z nich je křížení a druhým je mutace. Tato kapitola čerpá z [2] a [11] a jejím cílem je představit genetické programování.

Algoritmus genetického programování sestává z následujících kroků, které budou detailně popsány v jednotlivých podkapitolách.

1. Vygenerování náhodné počáteční populace.
2. Evoluční cyklus:
   (a) Ohodnocení všech kandidátních řešení v populaci.
   (b) Vytvoření následující generace:
      i. selekce – výběr kandidátů pro reprodukci.
      ii. reprodukce – kopírování kandidátů bez jejich modifikace.
      iii. křížení – vytvoření nových kandidátů, pomocí výměny genetického materiálu mezi dvěma jedinci.
      iv. mutace – náhodná změna genů některých kandidátů.
3. Po splnění ukončovací podmínky je výstupem nejlepší nalezené řešení.

3.1 Reprezentace kandidátních řešení

Vhodnou reprezentací počítačových programů jsou syntaktické stromy. Jejich velikost, tvar a obsah se dá měnit aplikací genetických operátorů. Syntaktické stromy jsou pohodlně implementovatelné pomocí dynamických polí. Uzly těchto stromů představují funkční symboly (sčítání, násobení, sin, cos, AND, OR, If-Then-Else atd.) a listy reprezentují terminální symboly (proměnné a konstanty). Obě tyto množiny volí uživatel podle potřeb daného problému. Je vhodné, aby splňovaly tyto dvě podmínky:

První vyžaduje uzavřenost neboli, aby každá funkce byla definována pro každou hodnotu vzniklou evaluací terminálního symbolu nebo aplikací funkce. Například, funkční množina obsahující dělení a terminální množina obsahující nulu tuto podmínku nesplňují, protože nulou nelze dělit. Nesplnění této podmínky může mít za následek nedefinované chování nebo pád programu.

Druhá říká, že funkční a terminální množiny by měly obsahovat takové symboly, aby pomocí nich bylo možné reprezentovat správné řešení. Například pokud by cílem evoluce bylo nalézt matematickou funkci, která aproximuje nějaké body, které očividně mají charakter paraboly, tak množiny obsahující pouze funkci sčítání, symbol proměnné x a konstantu 1, zjevně nejsou vhodné, protože umožní reprezentovat pouze lineární funkce.

3.2 Inicializace populace

Prvním krokem evoluce je vygenerování náhodné počáteční populace kandidátních řešení. K tomu byly vytvořeny tři metody: **grow**, **full** a **ramped half-and-half**. Všechny z nich se řídí nastavitelným parametrem, kterým je maximální výška stromu d.

Při použití metody **grow** se stromy inicializují tak, že jako kořen stromu se vybere náhodná funkce z množiny funkčních symbolů. Jeho následníci se vygenerují z množiny funkčních a terminálních symbolů. Pro každou takto vygenerovanou funkci se tento krok rekurzivně opakuje a následníci se opět vybírají z množiny funkčních a terminálních symbolů. Ve chvíli, kdy právě generovaná větev dosáhne maximální povolené výšky d, je možné vybrat pouze terminální symbol. Nevýhodou této metody je, že distribuce vytvořené populace závisí na poměru počtů symbolů v množině funkcí a v množině terminálů a také na aritě použitých funkcí. Při velkém množství terminálních symbolů mají výsledné stromy spíše menší výšku a v opačném případě se blíží plnému zaplnění.

Metoda **full** se liší tím, že místo vybírání následníků z množiny funkčních a terminálních symbolů, vybírá pouze z funkčních symbolů. Poslední úroveň je opět tvořena pouze terminálními symboly.

Třetí způsob inicializace, **ramped half-and-half**, kombinuje předchozí dvě metody. Populace je generována po částech a to tak, že $\frac{1}{d}$ populace je vygenerována s maximální výškou 1, další $\frac{1}{d}$ s výškou 2 atd. V každé části je polovina stromů inicializována metodou **grow** a druhá polovina metodou **full**. Takto vytvořená populace má velkou diverzitu, protože obsahuje malé, velké, vyvážené i nevyvážené stromy.

3.3 Fitness funkce

K objektivnímu ohodnocení, které kandidátní řešení lépe zvládá zadaný problém, slouží fitness funkce $f : X \to \mathbb{R}$, kde X je množina kandidátních řešení. Fitness funkce se volí tak, aby vyjadřovala vzdálenost k ideálnímu řešení. Například v případě evoluce autonomního robota, který má za úkol sbírat nějaké předměty, by fitness funkce mohla vyjadřovat počet úspěšně sebraných předmětů. Není možné, aby při vyhodnocování byly vyzkoušeny všechny možné varianty rozmístění předmětů v prostoru a proto se před zahájením evoluce vybere reprezentativní množina trénovacích případů, a ty pak slouží k porovnání jednotlivých kandidátů.

Je zvykem standardizovat fitness funkci tak, aby nižší hodnoty znamenaly lepší řešení, přičemž 0 značí ideální řešení. Další úpravou je normalizace, kdy se fitness funkce upraví tak, aby její obor hodnot byl od 0 do 1.

Genetické programování tak, jak je popsáno v této kapitole, porovnává kandidáty pouze podle jednoho kritéria. V případě výše zmíněného robota může být důležitá například i rychlost a spotřeba energie. K tomuto účelu byly vytvořeny metody, které umožňují optimalizovat více parametrů, například NSGA-II [7].

3.4 Selekce

Selekce je genetický operátor, který rozhoduje, zdali daný jedinec bude vybrán k reprodukci nebo bude zahozen. V ideálním případě by lepší jedinci měli mít větší pravděpodobnost reprodukce. V rámci selekce se hovoří o selekčním tlaku. Metoda s vysokým selekčním tlakem vybírá k reprodukci pouze z malého množství těch nejlepších kandidátů. Což může vést k rychlému zlepšení fitness hodnoty za cenu zhoršené diverzity populace. Mezi základní metody selekce patří:

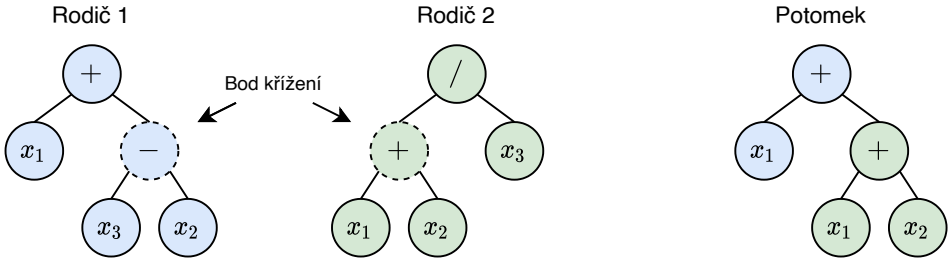
Ruleta, která vyžaduje, aby větší hodnota fitness znamenala lepší řešení, funguje tak, že každému jedinci je přiřazena pravděpodobnost výběru podle poměru jeho fitness hodnoty a součtu fitness hodnot všech jedinců v populaci. Problémy této metody se projeví ve dvou případech. Pokud mají jedinci zhruba stejné hodnoty fitness, tak je výběr prakticky náhodný a degeneruje k náhodnému prohledávání. V opačném případě pokud se v populaci nachází jedinec, který má výrazně lepší fitness než všichni ostatní, tak bude použit k vytvoření většiny potomků, což uškodí diverzitě příští generace.

Turnaj spočívá ve výběru určitého množství jedinců, kteří jsou porovnáni, a ten s nejlepší fitness hodnotou bude vybrán k reprodukci. Tento proces se poté opakuje tolikrát, kolikrát je potřeba. Tato metoda dosahuje dobrého kompromisu mezi selekčním tlakem a diverzitou populace.

Elitismus umožňuje zachovat určitý počet těch nejlepších jedinců do další generace.

3.5 Křížení

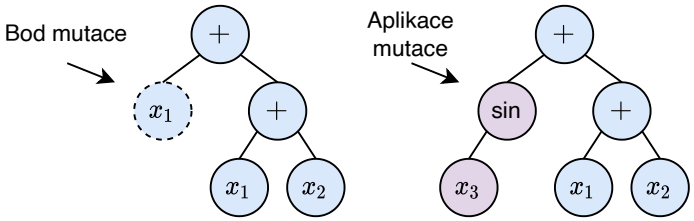
Po výběru dvou kandidátů k reprodukci je mezi nimi provedeno křížení. Tito kandidáti jsou rodiče T_1 a T_2. Standardní metoda křížení se nazývá **subtree crossover**. V každém z rodičů náhodně vybere jeden uzel, který se nazývá bod křížení. Potomek je vytvořen odstraněním bodu křížení rodiče T_1, na jehož místo je vložen bod křížení rodiče T_2 včetně jeho následníků. Je možné vytvořit i druhého potomka symetrickým způsobem. Může se stát, že takto vytvořený potomek přesáhne maximální povolenou výšku stromu. V tomto případě je zahozen a křížení se opakuje.



Obrázek 3.1: Aplikace operátoru křížení. Potomek je vytvořen odstraněním bodu křížení rodiče 1, na jehož místo je vložen bod křížení rodiče 2 včetně jeho následníků.

3.6 Mutace

Na nově vytvořené jedince je s určitou pravděpodobností (typicky velmi malou) použita mutace. Standardně funguje tak, že vybere náhodný uzel, který odstraní a místo něj vygeneruje novou větev. Mutace jsou užitečné pro případ, kdy populace konverguje k lokálnímu optimu a křížení nepřináší výrazné změny.



Obrázek 3.2: Aplikace operátoru mutace. Uzel označený jako bod mutace je nahrazen nově vygenerovanou větví.

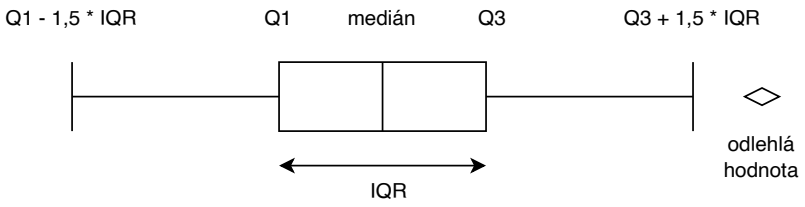
3.7 Ukončení a výsledek evoluce

Poslední částí algoritmu genetického programování je ukončovací podmínka. Při řešení problémů, které mají známé analytické řešení (např. návrh binární sčítačky), je možné ukončovat evoluci po nalezení korektního řešení. Avšak mnohé problémy nemají známé analytické řešení, a proto se evoluce ukončuje typicky po dosažení zvoleného maximálního počtu generací. Počet prozkoumaných řešení závisí na počtu generací a na velikosti populace. Proto když se hledá vhodné nastavení velikosti populace, tak se jako ukončovací podmínka nepoužívá maximální počet generací, ale maximální počet prozkoumaných řešení.

Výsledkem jednoho evolučního běhu je nalezené řešení s nejlepší fitness hodnotou.

3.8 Statistické vyhodnocení experimentů

Genetické programování je algoritmus, který je řízený generátorem náhodných čísel. Aby bylo možné zhodnotit kvalitu produkovaných řešení, je potřeba provést několik (typicky alespoň 30) nezávislých běhů. Z těch se poté vytváří statistika. Sleduje se hlavně vývoj fitness hodnoty v průběhu evoluce a fitness hodnota nejlepších řešení nalezených v daném běhu. Tyto údaje se často zobrazují pomocí grafů typu *boxplot* (viz obrázek 3.3).



Obrázek 3.3: Graf boxplot. *Q1* je první kvartil, *Q3* je třetí kvartil, svislá čára mezi *Q1* a *Q3* reprezentuje druhý kvartil (medián) a IQR je rovno $Q3 - Q1$. Odlehlá hodnota je taková hodnota, která není z intervalu $\langle Q1 - 1,5 \cdot IQR; Q3 + 1,5 \cdot IQR \rangle$. [18]

Kapitola 4

Kartézské genetické programování

Cílem této kapitoly je vysvětlil princip fungování kartézského genetického programování (CGP), jakožto metody původně vytvořené pro návrh elektrických obvodů. CGP se uplatnilo například při návrhu různých aproximačních obvodů, obrazových filtrů nebo umělých neuronových sítí [27]. Pro reprezentaci kandidátních řešení se využívá dvourozměrná mřížka, podle které dostala tato metoda své jméno. Tato kapitola vychází z [26].

4.1 Genotyp a fenotyp

Kandidátní řešení jsou tvořena acyklickými orientovanými grafy. Pro jejich reprezentaci se používá dvourozměrná mřížka tvořena výpočetními uzly. Genotyp je řetězec genů, reprezentovaných celými čísly, které určují, jaké operace jednotlivé uzly vykonávají, odkud berou svá vstupní data a co je výstupem programu. Výstup programu se získá dekódováním genotypu, při kterém jsou identifikovány aktivní uzly a jejich hodnota je vyčíslena. Aktivní uzly jsou takové uzly, které se podílí na tvorbě některého z výstupů. Program vzniklý dekódováním genotypu se nazývá fenotyp. Velikost fenotypu se po čas evoluce vyvíjí v závislosti na počtu aktivních uzlů, přičemž velikost genotypu představuje horní ohraničení počtu uzlů, které je možno využít.

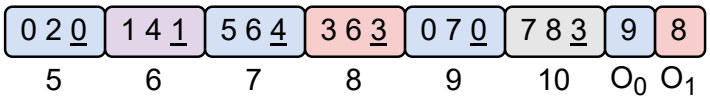
Operace, které mohou uzly vykonávat, si volí uživatel a jsou uloženy v tabulce. Jednotlivé uzly tvoří dva typy genů. Prvním je funkční gen, který obsahuje adresu některé funkce z tabulky. Každý uzel má svou adresu, kterou využívá druhý typ genů, které se nazývají propojovací a určují, odkud daný uzel bere svá data. Jsou dvě možnosti: vstup programu nebo výstup některého uzlu z předchozích sloupců. Uzel nemůže pracovat s výstupem uzlu ze stejného nebo následujícího sloupce. Vstupy programu mají adresy od 0 do $n_i - 1$, kde n_i je počet vstupů. Výstupům uzlů jsou přidělovány adresy po sloupcích od n_i do $L_n - 1$, kde L_n je celkový počet uzlů. Poslední část genotypu tvoří n_o genů, které obsahují adresy výstupů programu. Příklad adresování je uveden na obrázku 4.1.

K nastavení velikosti mřížky a propojitelnosti uzlů v CGP slouží tři parametry. Počet řádků n_r, počet sloupců n_c a konektivita l. Parametr l vyjadřuje maximální počet sloupců, napříč kterými je možné propojit dva uzly. Pokud $l = 1$, tak uzel může být propojen pouze s uzlem z předchozího sloupce nebo se vstupem programu. Maximální hodnota tohoto parametru je rovna n_c, která udává, že daný uzel může jako svůj vstup použít uzel z libovolného předchozího sloupce nebo vstup programu.

4.2 Příklad zakódováni problému

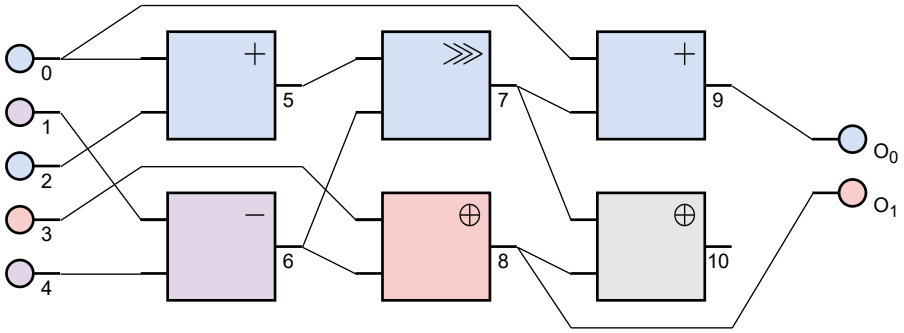
Pomocí CGP je možné řešit různé problémy, nejen navrhovat elektronické obvody. V této sekci je demonstrace použití CGP pro návrh matematické funkce.

Obrázek 4.1 obsahuje genotyp řešení nalezeného při návrhu dvojice hašovacích funkcí. Vstupem těchto funkcí je pětice 64bitových celých čísel. Výstupy O_0 a O_1 náleží jednotlivým hašovacím funkcím. Při evoluci byly parametry $n_r = 2$, $n_c = 3$ a $l = 1$. Množinu funkcí tvoří 5 operací: sčítání (0), odčítání (1), násobení (2), XOR (3) a bitová rotace vpravo (4). Každá z těchto operací používá dva operandy.



Obrázek 4.1: Příklad genotypu. Jedna trojice odpovídá jednomu uzlu, kde první dva geny obsahují adresy vstupních dat a podtržený gen reprezentuje funkci uzlu. Pod každým uzlem je uvedena jeho adresa.

Na obrázku 4.2 je vizuální reprezentace tohoto genotypu, kde jde lépe vidět propojení uzlů. Výstupu O_0 byla přiřazena modrá barva a výstupu O_1 červená. Každý uzel byl následně obarven podle toho, který výstup jej využívá. Uzel s adresou 10 je šedý, protože ho nevyužívá žádný výstup, tedy je neaktivní. Uzel označený fialově (s adresou 6) demonstruje, že CGP umožňuje používat části řešení vícekrát.



Obrázek 4.2: Reprezentace genotypu z obrázku 4.1 v mřížce uzlů.

Tomuto genotypu odpovídá fenotyp popsaný rovnicemi 4.1. Jak je vidět, že část obou funkcí je stejná a díky tomu, že tato část pochází ze sdíleného uzlu, není potřeba ji vyčíslovat dvakrát.

$$\begin{aligned} O_0 &= I_0 + ((I_0 + I_2) \ggg (I_1 - I_4)) \\ O_1 &= I_3 \oplus (I_1 - I_4) \end{aligned} \tag{4.1}$$

4.3 Identifikace aktivních uzlů a dekódování genotypu

Prvním krokem dekódování genotypu na fenotyp je identifikace aktivních uzlů, které je potřeba vyčíslit. Ostatní uzly není potřeba vyčíslovat, protože se nepodílí na tvorbě výstupních hodnot. Vytvoří se pole pravdivostních hodnot NU, které signalizují, zdali je příslušný uzel

aktivní. Všechny hodnoty se inicializují na $False$. Poté se provede průchod genotypem začínající od genů výstupů. Pozice v NU indexované hodnotami výstupních genů se nastaví na $True$. Pokračuje se průchodem všech uzlů počínaje uzlem s nejvyšší adresou. Pokud pozice tohoto uzlu v NU obsahuje $True$, tak hodnoty prvních a_i genů příslušící tomuto uzlu se použijí pro indexování dalších uzlů v NU, které se označí jako aktivní, kde a_i odpovídá aritě funkce uzlu i. Výstupem je pole NU.

Druhým krokem je samotné vyčíslení uzlů. K tomu se vytvoří pole NV. Prvních n_i prvků se inicializuje na vstupní hodnoty. Zbylých L_n prvků slouží pro výstupy uzlů a budou vypočteny v následujících krocích. Provede se průchod přes všechny uzly a ty uzly, které jsou v NU označeny jako aktivní, budou vyčísleny. Adresy vstupů uzlu se přečtou z genotypu a jejich hodnoty se získají z NV. Provede se funkce, která přísluší hodnotě funkčního genu, a výsledek se zapíše do NV na adresu odpovídající tomuto uzlu. Výstupy programu se nachází v NV na adresách, které odpovídají hodnotám výstupních genů.

4.4 Genetické operátory

Na rozdíl od genetického programování představeného v kapitole 3, CGP nepoužívá k vytváření potomků operátor křížení, ale používá pouze mutaci. V prvních verzích se sice křížení vyskytovalo, ale ukázalo se, že negativně ovlivňuje výkon evoluce [25]. V CGP se používá **bodová mutace**, která funguje tím způsobem, že vybere náhodný gen a změní jeho hodnotu na jinou validní hodnotu. Pokud je vybraný funkční gen, tak validní hodnota je adresa některé funkce z funkční množiny. Obdobně, pokud je vybrán propojovací gen, tak jeho hodnota může být nahrazena pouze adresou některého uzlu z předchozích sloupců při respektování parametru l nebo adresou vstupu programu. Validní hodnotou pro gen reprezentující výstup programu je adresa libovolného uzlu nebo vstupu programu.

Počet genů, které budou aplikací mutace změněny, definuje uživatel pomocí parametru pravděpodobnosti mutace μ_r. Jeho hodnota se vyjadřuje jako procento celkového počtu genů, které mají být modifikovány. Pokud genotyp obsahuje 100 genů, tak při $\mu_r = 5$ % bude modifikováno 5 genů.

4.5 Evoluční algoritmus a neutralita

Pro řízení evoluce se používá varianta evoluční strategie známá jako $1 + \lambda$. Parametr λ se typicky volí jako 4 [26]. Jednotlivé kroky jsou popsány algoritmem 2. Každý jedinec v populaci je ohodnocen a ten nejlepší z nich je zvolen jako rodič následující generace. V případě, že existuje několik nejlepších potomků se stejnou fitness hodnotou, ze kterých

se může stát rodič, tak se z nich vybere náhodně. Aplikací mutačního operátoru na rodiče je vytvořeno λ potomků. Poté se cyklus opakuje.

Algoritmus 2: Evoluční strategie $1 + \lambda$ [26]

```
1  Vygeneruj populaci obsahující λ + 1 náhodných jedinců
2  Označ jedince s nejlepší fitness hodnotou jako rodiče
3  while řešení není nalezeno nebo není dosaženo maxima generací do
4  |  for i = 1, 2, 3, ..., λ do
5  |  |  Aplikací mutace na rodiče vytvoř potomka x_i
6  |  Vypočítej fitness hodnotu všech jedinců v populaci
7  |  if některý z potomků má stejnou nebo lepší fitness než rodič then
8  |  |  Nejlepší potomek je zvolen jako rodič následující generace
9  |  else
10 |  |  Rodič zůstává stejný
```

V sekci 4.1 bylo řečeno, že v genotypu se můžou vyskytovat neaktivní uzly, které se nepodílí na výstupu a tím pádem nemají vliv na fitness hodnotu. Při vytváření nových potomků je možné, že budou zmutovány pouze tyto uzly. V případě, že nebude vytvořen žádný striktně lepší potomek než rodič, tak je preferované, aby se rodičem následující generace stal potomek se stejnou fitness hodnotou jako rodič. Pokud takový potomek neexistuje, rodič zůstává nezměněn. Tento mechanismus podporuje diverzitu populace [26].

Důležitost tohoto mechanismu zkoumá například [34]. Cílem evoluce bylo navrhnout tříbitovou paralelní násobičku. Bylo provedeno 100 běhů ve dvou variantách. V první se potomek stal rodičem, pokud měl stejnou nebo lepší fitness hodnotu. Druhá varianta vyžadovala, aby potomek byl striktně lepší. Při použití první varianty bylo nalezeno 27 správných řešení. Průměrná fitness hodnota byla v případě druhé varianty podstatně horší a nebylo nalezeno ani jedno korektní řešení.

4.6 Další metody mutace

Při vytváření potomků pomocí mutačního operátoru je možné, že fenotyp nově vzniklého jedince se nebude lišit od svého rodiče. Tato situace nastane, když mutace ovlivní pouze neaktivní uzly. Takto vytvořený jedinec má stejnou fitness hodnotu jako jeho rodič, a tedy opětovné volání fitness funkce je plýtvání procesorového času. Z tohoto důvodu byly navrženy metody, které tento problém řeší.

První z nich se nazývá **skip** [14]. Po vytvoření každého nového jedince jsou jeho geny porovnány s jeho rodičem. Pokud se geny aktivních uzlů neliší, tak je tomuto jedinci přiřazena fitness hodnota jeho rodiče. Geny neaktivních uzlů se mohou lišit. V opačném případě se provede ohodnocení fitness funkcí. Jelikož princip tvorby potomků zůstal nezměněný, tak průběh evoluce je totožný s klasickým CGP. Výhoda této metody je, že šetří čas a teoreticky umožňuje za stejnou dobu prozkoumat více řešení. Efektivita této metody závisí na pravděpodobnosti mutace a počtu aktivních uzlů. Při relativně vysoké pravděpodobnosti mutace nebo velkém počtu aktivních uzlů je velmi malá šance, že nově vytvořený potomek bude identický se svým rodičem a nevznikne příležitost ušetřit čas. Z tohoto důvodu je při použité této metody vhodné volit menší pravděpodobnost mutace.

Zatímco první metoda těží ze skutečnosti, že není potřeba ohodnocovat identické potomky, druhá metoda zvaná **single** zaručuje, že nově vzniklý potomek se bude lišit od svého rodiče [14]. Místo toho, aby mutace probíhala tím způsobem, že se náhodně vybere určité procento genů a ty jsou zmutovány, tak tato metoda náhodně mutuje geny a po každé mu-

taci ověřuje, zdali byl zmutován aktivní uzel, pokud ano, tak končí. Takto vzniklý jedinec se bude lišit od svého rodiče právě v jednom aktivním genu. Mohou být ovlivněny i neaktivní uzly. Při použití této metody není potřeba nastavovat pravděpodobnost mutace.

Optimalizací mutačních operátorů se zabývá například i [36], kde autoři využili metodu single pro vytvoření nového mutačního operátoru zvlášť vhodného pro návrh aproximačních obvodů.

Kapitola 5

Využití evolučních algoritmů pro návrh hašovacích funkcí

Hašovací tabulka je datová struktura pro ukládání dat, která se využívá především kvůli tomu, že poskytuje velmi rychlé vyhledávání. Aby hašovací tabulka byla efektivní, je potřeba zvolit dobré hašovací funkce. Jelikož algoritmy hašovacích funkcí obsahují na první pohled náhodné zřetězení matematických operátoru, je obtížné říct, proč některé hašovací funkce fungují lépe než ostatní, což komplikuje návrh nových funkcí. Právě tyto nejasné vztahy mezi dílčími stavebními bloky a komplikovaný analytický návrh poskytují prostor pro aplikaci evolučních algoritmů [32]. V uplynulých letech byla vyzkoušena různá odvětví evolučních algoritmů. Například stromové genetické programování [9], [33], [21], lineární genetické programování [16], kartézské genetické programování [16] nebo gramatická evoluce [20].

Některé práce se zabývají evolucí hašovacích funkcí pro použití ve specifických doménách, jiné zase navrhují funkce pro univerzální použití. Podle cílové domény se volí fitness funkce. Má-li navržená funkce být použita ve specifické doméně, je vhodnou fitness funkcí počet kolizí. Naopak, při návrhu univerzálních hašovacích funkcí se hodí použít lavinový efekt, který měří změnu výstupní hodnoty v závislosti na malé změně vstupní hodnoty, protože nezávisí na trénovacích datech.

5.1 Relevantní publikace

Návrhem univerzálních hašovacích funkcí se zabýval Estébanez et al. [9]. Jako základ autoři využili Merkleovu–Damgårdovu konstrukci a pomocí stromového genetického programování navrhli potřebnou kompresní funkci. Pro porovnání jednotlivých řešení použili lavinový efekt. Téměř všechny dosavadní práce používaly jako fitness funkci počet kolizí a autorům se povedlo ukázat, že lavinový efekt je velmi dobré kritérium pro porovnávání univerzálních hašovacích funkcí. Dále se tato práce zabývá nalezením optimální množiny funkcí a ukazuje, že „magické“ konstanty nejsou potřeba.

Navazující práce od Saez et al. řeší návrh hašovacích funkcí pro specifické domény [33]. Autoři opět využívají Markleovu-Damgardovu konstrukci a stromové genetické programování. Avšak pro porovnání kandidátních řešení při evoluci nepoužívají lavinový efekt, ale počet kolizí. Při testování používají i další metriky, například entropii nebo χ^2 statistiku výstupních hodnot hašovacích funkcí. Z experimentů na různých datových sadách se uka-

zuje, že hašovací funkce vyvinuté pro specifickou datovou sadu dosahují lepších výsledků než univerzální funkce na této sadě.

Návrhem hašovacích funkcí pro doménu IPv4 adres se zabývali Kidoň a Dobai [21]. Tato práce je zajímavá tím, že využívá kukaččí hašování jako metodu řešení kolizí a tedy cílem genetického programování je zde nalezení páru hašovacích funkcí. Další odlišností od ostatních předem zmíněných prací je, že autoři nepoužívají Merkleovu–Damgårdovu konstrukci, ale vstupem hašovacích funkcí jsou jednotlivé bajty IPv4 adres.

Další zajímavou prací je [16], kde autoři navrhují hašovací funkce pro doménu síťových toků. Autoři se nezaměřují pouze na minimalizaci počtu kolizí, ale využívají vícekriteriální optimalizaci a porovnávají funkce podle rychlosti a počtu kolizí zároveň. Zajímavé je, že navzdory relativně velkým vstupům (320 bitů) autoři nepoužívá Merkleovu–Damgårdovu konstrukci. Hlavním přínosem je vytvoření velmi rychlých hašovacích funkcí díky vícekriteriální optimalizaci. Dalším přínosem je porovnání lineárního a kartézského genetického programování, a zjištění, že kartézské dosahuje mírně lepších výsledků. Zajímavý je také způsob redukce 64bitového haše na 16bitový index hašovací tabulky, který využívá metodu XOR folding [3].

5.2 Shrnutí a zhodnocení

Bylo zveřejněno již několik publikací zabývajících se evolučním návrhem hašovacích funkcí. Většina z nich využívá nějakou variantu genetického programování. A ukazuje se, že genetické programování je vhodné pro návrh hašovacích funkcí. Evolučně vyvinuté funkce jsou schopné konkurovat „state-of-the-art“ hašovacím funkcím, které navrhli experti v tomto oboru. Nejenže jsou konkurenceschopné, ale bylo ukázáno, že genetické programování dokáže vyvinout funkce, které jsou dokonce několikanásobně rychlejší a produkují méně kolizí [9], [33], [21], [16]. Byly vyzkoušeny různé metody porovnání nalezených řešení. Pro potřeby evoluce univerzálních funkcí se vyplatí použít lavinový efekt [9] a pro případ aplikace ve specifické doméně je vhodné provádět trénování na konkrétních datech a porovnávat podle počtu kolizí [16], [33] nebo zaplnění hašovací tabulky [21]. Co se týká struktury samotných hašovacích funkcí, tak univerzální využívají Merkleovu-Damgardovu konstrukci, ale funkce pro specifické domény se bez této konstrukce obejdou. Časté je taky využití XOR foldingu pro redukci bitové šířky výsledného haše [16], [21]. Byly provedeny studie nejvhodnějších matematických operátorů pro tvorbu hašovacích funkcí pomocí experimentálního porovnání [9] nebo pomocí analýzy existujících funkcí [16]. Výše zmíněné publikace využívají především operace sčítání, násobení, XOR a bitovou rotaci.

Existuje hned několik metod mutace pro kartézské genetické programování, avšak většina řešení založených na CGP využívá základní variantu. Proto se nabízí možnost vyzkoušet tyto metody a porovnat jejich efektivitu se základní variantou. Empirická studie operátorů byla provedena pro univerzální funkce bez použití XOR foldingu. Jelikož XOR folding se dá chápat jako implicitní XOR a rotace, tak by bylo vhodné prozkoumat jeho vliv na potřebnou množinu operátorů. Řešení, která nepoužívají Merkleovu-Damgardovu konstrukci a pracují s dostatečně dlouhým vstupem, typicky navrhují pouze jeden způsob rozdělení do vstupních bloků. Bylo by možné navrhnout různé varianty a porovnat jejich vliv na výkon CGP a strukturu výsledných hašovacích funkcí.

Náměty představené v této sekci budou detailně rozebrány v kapitole 6 a experimentálně vyhodnoceny v kapitole 7.

Kapitola 6

Implementace evolučního návrhu hašovacích funkcí

V této kapitole bude popsána metoda pro automatizovaný návrh hašovacích funkcí a její implementace. Vyvíjené hašovací funkce jsou optimalizovány pro zpracovávání síťových toků. Trénovací a testovací data budou představena v sekci 6.1 včetně jejich zakódování. V následující sekci 6.2 bude popsána reprezentace kandidátních řešení pomocí kartézského genetické programování. Poté v sekci 6.3 bude vysvětlena fitness funkce použitá pro porovnávání kandidátních řešení. Sekce 6.4 přiblíží detaily implementace použitých metod a analýzy výsledků.

6.1 Trénovací a testovací data

Vyvíjené hašovací funkce budou trénovány pro hašování síťových toků. K tomu je potřeba připravit trénovací a testovací datové sady. Budou použity anonymizované data poskytnuty vedoucím této bakalářské práce. Pro trénování bude použita sada, která obsahuje 65535 síťových toků a poté pro testování bude použito 8 různých sad, které taktéž obsahují 65535 položek. Datové sady jsou uloženy v souboru, kde každý řádek představuje jeden síťový tok rozdělený do 32bitových částí oddělených středníkem. Jedná se o síťové toky IPv6, které tvoří zdrojová IPv6 adresa (4 x 32 bitů), cílová IPv6 adresa (4 x 32 b), zdrojový port (32 b) a cílový port (32 b). Nejméně významné slovo (32 b) dané čtveřice se nachází vpravo. Každý síťový tok tedy obsahuje 288 užitečných bitů (porty jsou ve skutečnosti 16bitové hodnoty).

Dále je potřeba určit v jaké formě budou data předána hašovací funkci. Vstupem hašovací funkce bude síťový tok rozdělený do několika bloků stejné velikosti. Jelikož jednotlivé toky obsahují poměrně hodně bitů, je možné navrhnout hned několik variant. První varianta rozděluje toky do pěti 64bitových bloků, přičemž u posledního bloku (I_4) je využito pouze 32 bitů a zbytek obsahuje nuly. Druhou variantou je rozdělení do devíti 32bitových bloků, tentokrát už plně využitých. Třetí varianta se vytvoří rozdělením každého 32bitového bloku do dvou 16bitových bloků, čímž vznikne osmnáct 16bitových bloků. Všechny tři varianty jsou zobrazeny na obrázku 6.1.

6.2 Evoluce pomocí CGP

Pro evoluci kandidátních řešení bude použito kartézské genetické programování. Jedno kandidátní řešení představuje dvojici hašovacích funkcí h_1 a h_2 pro potřeby kukaččího hašování.

Obrázek 6.1: Rozdělení vstupních dat do bloků. Obsahuje 64bitovou, 32bitovou i 16bitovou variantu.

Vstupy CGP odpovídají blokům z obrázku 6.1 a jejich počet závisí na použité variantě. Výstupy CGP jsou dva a reprezentují funkce h_1 a h_2. Jednotlivé uzly jsou tvořeny třemi geny, první dva jsou propojovací a ten třetí je funkční uzel. Příklad byl již uveden v sekci 4.2.

Varianta bitové šířky vstupů CGP udává bitovou šířku výstupů i jednotlivých uzlů. Takže v případě použití 64bitové varianty budou uzly provádět 64bitové operace a výstupem budou 64bitové haše.

Základní množina funkcí uzlů bude obsahovat sčítání ($+$), násobení ($\cdot$), XOR ($\oplus$) a bitovou rotaci vpravo ($\ggg$). V sekci 7.2 budou vyzkoušeny modifikované množiny.

Při evoluci bude využita strategie $1 + \lambda$, kde $\lambda = 4$. Noví kandidáti budou vytvářeni pomocí bodové mutace. Sekce 7.4 vyzkouší další metody mutace. Ostatní parametry, jako je rozměr mřížky CGP, pravděpodobnost mutace a maximální počet přemístění kukaččího hašování, budou zvoleny experimentálně v ostatních sekcích kapitoly 7.

6.3 Fitness funkce

V sekci 5.2 bylo zmíněno několik způsobů, jak porovnávat kandidátní řešení. Jelikož vyvíjené funkce budou aplikovány ve specifické doméně a pro řešení kolizí budou využívat kukaččí hašovaní, tak jako nejvhodnější varianta se jeví porovnání podle zaplnění hašovací tabulky.

Hašovací tabulka použitá ve fitness funkci využívá kukaččí hašování a je tvořena dvěma poli, kde každé má kapacitu 32768 (2^{15}) položek. Pro indexování prvního pole slouží hašovací funkce h_1 a pro indexování druhého slouží funkce h_2.

Haše, které produkují funkce h_1 a h_2, mají větší bitovou šířku než index hašovací tabulky. Aby bylo možné je použít, budou muset být redukovány na 15 bitů. Nabízí se dvě varianty, jak toho docílit. První z nich využívá operátor modulo (mod), který se použije prakticky jako bitová maska pro zachování spodních 15 bitů. Druhá varianta, inspirovaná metodou XOR folding, spočívá v rozdělení haše na několik 16bitových částí, které jsou poté zkombinovány operací XOR, čímž vznikne 16bitová hodnota, která je následně upravena operací modulo na 15 bitů. Rovnice 6.1 popisují obě varianty. První variantě odpovídá funkce r_1, druhé variantě

odpovídá r_2. Symbol $\gg$ značí bitový posuv vpravo a $\oplus$ značí XOR. Činnost funkce r_2 závisí na bitové šířce vstupního haše. Je zřejmé že pro 16bitovou variantu není mezi variantami redukce rozdíl.

$$\begin{aligned} r_1(x) &= x \bmod 32768 \\ r_{2,16bit}(x) &= x \bmod 32768 \\ r_{2,32bit}(x) &= ((x \gg 16) \oplus (x)) \bmod 32768 \\ r_{2,64bit}(x) &= ((x \gg 48) \oplus (x \gg 32) \oplus (x \gg 16) \oplus (x)) \bmod 32768 \end{aligned} \tag{6.1}$$

Ohodnocení kandidátních řešení bude probíhat následujícím způsobem. Pro každý síťový tok v trénovací sadě budou vypočítány jeho haše podle hašovacích funkcí h_1 a h_2 včetně redukce na 15 bitů. Poté se provede vložení každého toku do hašovací tabulky. Je možné, že se nepodaří přemístit položky v hašovací tabulce tak, aby bylo možné umístit tento tok. V tomto případě je vkládání neúspěšné a pokračuje se vkládáním dalšího toku. Posledním krokem je výpočet samotné fitness hodnoty, která se vypočítá podle vzorce 6.2, kde dvojice h_1, h_2 představuje jedno kandidátní řešení. $Kapacita$ je celková kapacita hašovací tabulky. $Vlozeno$ je počet úspěšně vložených síťových toků. Výstupem fitness funkce jsou hodnoty od 0 do 1, přičemž nižší hodnota je lepší. Hodnota 0,17 znamená, že 17 % hašovací tabulky zůstalo prázdné.

$$f(h_1, h_2) = \frac{Kapacita - Vlozeno}{Kapacita} \tag{6.2}$$

6.4 Implementace CGP

Před samotnou implementací bylo nejprve potřeba zvolit vhodný programovací jazyk. Vybral jsem jazyk C++ ve standardu C++20. Díky tomu, že C++ je staticky typovaný kompilovaný jazyk, je možné vytvářet programy, které jsou rychlé. Což je výhodou, protože není žádoucí, aby evoluční běhy trvaly příliš dlouho. Další výhodou je, že C++ podporuje generický datový typ. Ten se uplatní například při implementaci vyčíslování uzlů CGP. Datový typ hodnoty uzlu je určený variantou rozdělení vstupních dat do bloků (viz sekce 6.1), avšak algoritmus vyčíslování zůstává stejný. C++ poskytuje bohatou standardní knihovnu, což odstraňuje potřebu znovu implementovat běžně používané datové struktury jako je např. dynamické pole.

Konfigurace

Konfigurace CGP se nachází v hlavičkovém souboru `config.h`. A nastavuje se pomocí těchto konstant:

- `MAX_LOOP` – hodnota konstanty *MaxLoop* (viz sekce 2.2).
- `ROWS` – počet řádků CGP mřížky (viz sekce 4.1).
- `COLS` – počet sloupců CGP mřížky (viz sekce 4.1).
- `POP` – celková velikost populace (viz sekce 4.5).
- `GEN` – maximální počet generací (viz sekce 3.7).

- `MUTATE` – pravděpodobnost mutace vyjádřená jako desetinné číslo (viz sekce 4.4).

Alternativní metody činnosti se nastavují definováním těchto maker:

- `MUTATE_SINGLE` – metoda single (viz sekce 4.6).
- `DONT_COUNT_CACHED` – evoluce se ukončí po provedení `POP*GEN` evaluací (viz sekce 3.7).
- `QUICK_FITNESS` – zrychlená fitness funkce (viz sekce 7.5).
- `DONT_XOR_FOLD` – redukce haše pouze pomocí operace modulo (viz sekce 6.2).
- `LOG_ONLY_NEW_BEST` – výpis pouze generací, ve kterých se zlepšila fitness hodnota.
- `LOG_NODE_USAGE` – výpis informací o počtu aktivních uzlů.

Evoluční algoritmus

Základní kostra evolučního algoritmu se nachází v souboru `main64.cpp` (resp. `main32.cpp` a `main16.cpp`). Nejprve se provede inicializace počáteční populace a poté je zahájen evoluční cyklus. V tom se provede ohodnocení každého jedince fitness funkcí, poté jsou vypsány informace o aktuální generaci a nakonec je volána funkce pro přípravu následující generace.

Inicializace populace

Funkce potřebné pro inicializaci populace se nachází v souboru `cgp.cpp`. Jedná se o funkce `init_pop` a `random_chrom`. Genotyp jednoho kandidátního řešení je reprezentován datovým typem `std::vector<short>`.

Vyčíslení uzlů

Nejprve jsou identifikovány aktivní uzly pomocí funkce `find_active` v souboru `cgp.cpp`, která implementuje algoritmus popsaný v sekci 4.3. Poté se využije funkce `eval_nodes` ze souboru `cgp_templates.tpp` k vyčíslení hodnot aktivních uzlů. Tato funkce využívá generický datový typ pro hodnoty uzlů, což umožňuje ji využít pro všechny varianty rozdělení hašovaných síťových toků do bloků. Nachází se zde implementace všech funkcí z funkční množiny. Algoritmus této funkce je také popsán v sekci 4.3.

Fitness funkce

Po vyčíslení uzlů se provádí redukce výstupů CGP do požadovaného rozsahu pomocí redukční funkce (viz sekce 6.2). Ta je implementována ve funkci `hashes_from_outs` ze souboru `cgp.cpp`, která využívá přetížení pro odlišení jednotlivých variant. Ohodnocení kandidátních řešení se provádí pomocí fitness funkce ze souboru `fitness.cpp`. Zde se nachází implementace kukaččího hašování.

Tvorba následující generace

Po ohodnocení všech kandidátů v populaci je hledán rodič pro příští generaci. Rodič je v populaci vždy uložen na indexu 0. Díky tomu je možné rodiče pro následující generaci nalézt pomocí lineárního prohledávání. Rodičem se stane kandidát s nejvyšší fitness hodnotou. V případě shody je vybrán kandidát s vyšším indexem. Samotná tvorba následující

populace probíhá ve funkci `next_pop` ze souboru `cgp.cpp`. V této funkci jsou volány funkce pro mutaci – `mutate_basic` nebo `mutate_single`.

Výstup programu

Funkce pro tisk výsledků evoluce se nachází v souboru `logging.cpp`. Program tiskne výsledky ve formátu CSV[1] (jako oddělovač je použit středník) na standardní výstup. Na první řádek jsou vypsány hodnoty všech parametrů. Druhý řádek obsahuje hlavičku. Každý následující řádek odpovídá jedné generaci a obsahuje číslo generace, celkový počet evaluací, fitness hodnotu nejlepšího jedince a genotyp nejlepšího jedince. Na poslední řádek se tiskne celková doba běhu evoluce uvedená v sekundách.

Analýza výsledků experimentů

Analýza experimentů prováděných v kapitole 7 byla provedena v prostředí Jupyter Notebook[2] a nachází se v souboru `result_analysis.ipynb`. Byl použit jazyk Python3 ve verzi 3.8. Pro zpracování dat byla využita knihovna Pandas[3]. Grafy jsou vykresleny pomocí knihoven Matplotlib[4] a Seaborn[5].

[1] https://opendata.gov.cz/standardy:csv
[2] https://jupyter.org/
[3] https://pandas.pydata.org/
[4] https://matplotlib.org/
[5] https://seaborn.pydata.org/

Kapitola 7

Experimenty

V této kapitole bude popsán průběh experimentů a jejich výsledky. Jednotlivé experimenty vychází z poznatků ze sekce 5.2. První experiment 7.1 se zabývá nastavením parametru kukaččího hašování. Poté v sekci 7.2 bude hledána optimální množina funkcí. Sekce 7.3 řeší nastavení velikosti mřížky CGP a pravděpodobnosti mutace. V následující sekci 7.4 budou vyzkoušeny další metody mutace. Dále v sekci 7.5 bude provedena evoluce pomocí zrychlené fitness funkce. Porovnání nejlepších nalezených hašovacích funkcí se state-of-the-art hašovacími funkcemi bude bude provedeno v sekci 7.6. Poslední sekce 7.7 obsahuje shrnutí výsledků experimentů.

Veškeré experimenty v této kapitole byly prováděny na osobním počítači s procesorem AMD Ryzen 5 2600, který má frekvenci 3,4 GHz. Počítač má 16 GB paměti RAM typu DDR5 s frekvencí 3200 MHz. Operační systém je Ubuntu 20.04.

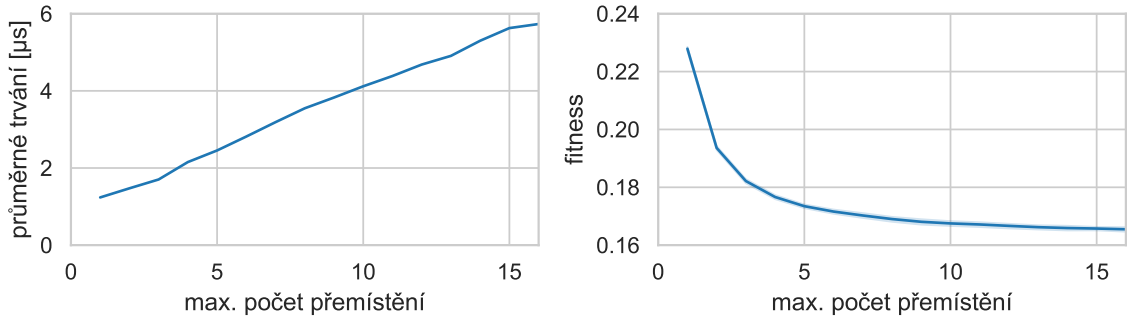
7.1 Kukaččí hašování

Před zahájením experimentů s evolučním návrhem hašovacích funkcí je potřeba nastavit parametr kukaččího hašování. Tímto parametrem je konstanta $MaxLoop$ (představená v sekci 2.2), která určuje, kolik přemístění mezi jednotlivými poli hašovací tabulky se může maximálně provést, než je vkládání prohlášeno za neúspěšné. Konstanta $MaxLoop$ ovlivňuje efektivitu řešení kolizí, ale i spotřebu procesorového času. Proto je pro potřeby následujících experimentů důležité najít vhodný kompromis.

Při měření byla jako první i druhá hašovací funkce použita MurmurHash3. Funkce se od sebe lišily počáteční hodnotou. Experiment probíhal tak, že každý síťový tok z testovací sady byl vložen do hašovací tabulky a bylo změřeno, kolik zabral výpočet času a jaké fitness hodnoty bylo dosaženo. Experiment byl zopakován na všech osmi trénovacích sadách a poté byl z naměřených hodnot vypočítán průměr.

Výsledky

Výsledky experimentu jsou vykresleny na obrázku 7.1. Graf vlevo ukazuje průměrnou dobu výpočtu a graf vpravo zobrazuje fitness hodnotu. Je vidět, že průměrná doba výpočtu roste lineárně s hodnotou $MaxLoop$, avšak závislost fitness hodnoty má charakter spíše exponenciály a od určitého bodů nedochází téměř k žádnému zlepšení. Z tohoto důvodu jsem pro všechny následující experimenty nastavil hodnotu konstanty $MaxLoop$ na 6. Při této hodnotě výpočet trval 2,8 µs a fitness hodnota byla 0,172.



Obrázek 7.1: Vliv maximálního počtu přemístění na fitness a na dobu výpočtu

7.2 Množina funkcí

První experiment prováděný s kartézským genetickým programováním se zabývá volbou funkcí, které mohou vykonávat uzly v CGP. Dle [13] je možné rozdělit funkce do skupin. Funkce, které společně tvoří jednu skupinu, se nazývají ekvivalentní a mají stejně velký vliv na efektivitu genetického programování. Cílem tohoto experimentu je nalézt co nejlepší množinu funkcí. To je taková množina, která obsahuje z každé skupiny právě jednu funkci. Zároveň bude prozkoumán vliv redukční funkce, představené v sekci 6.3, na tuto množinu.

Postup experimentu vychází z [9]. Základní množina (F0) byla vytvořena po prozkoumání existujících řešení a obsahuje sčítání ($+$), násobení ($\cdot$), XOR ($\oplus$) a bitovou rotaci vpravo ($\ggg$). Navržené úpravy této množiny jsou uvedeny v tabulce 7.1, kde & je bitový AND, | je bitový OR, $\gg$ a $\ll$ jsou bitové posuvy.

Množina	Funkce
F0	$+$, $\cdot$, $\oplus$, $\ggg$
F1	$+$, $\cdot$, $\oplus$, $\ggg$, &, |
F2	$+$, $\cdot$, $\oplus$, $\ggg$, $\gg$, $\ll$
F3	$+$, $\cdot$, $\oplus$, $\lll$
F4	$-$, $\cdot$, $\oplus$, $\ggg$
F5	$\cdot$, $\oplus$, $\ggg$
F6	$+$, $\oplus$, $\ggg$
F7	$+$, $\cdot$, $\ggg$
F8	$+$, $\cdot$, $\oplus$
F9	$\cdot$, $\ggg$
F10	$+$, $\cdot$
F11	$+$
F12	$\cdot$

Tabulka 7.1: Tabulka množin funkcí. F0 je základní množina. Symbol $\oplus$ je XOR, & je AND, | je OR, $\ggg$ a $\lll$ jsou bitové rotace, $\gg$ a $\ll$ jsou bitové posuvy.

Nastavení CGP použité při tomto experimentu je popsáno v tabulce 7.2. S každou kombinací množiny funkcí a šířky vstupních bloků bylo provedeno 50 nezávislých běhů CGP. Grafy na obrázcích 7.2 a 7.3 byly vytvořeny z nejlepších řešení nalezených v každém z těchto běhů.

Řádky	1
Sloupce	30
Vstupy	18 (16bit.), 9 (32bit.), 5 (64bit.)
Výstupy	2
Populace	4 + 1
Max. generace	2000
Mutace	5 %

Tabulka 7.2: Parametry CGP pro experiment s množinou funkcí.

Výsledky

V této sekci byl použit Mann-Whitneyův test[1] na hladině významnosti $\alpha = 0,05$.

F1 a **F2**. Podle grafů se zdá, že přidání funkcí AND a OR nebo bitových posuvů nepřináší žádné zlepšení, čemuž nasvědčuje i fakt, že tyto funkce byly ve variantě modulo využity velmi zřídka, zhruba 6krát méně než ty ostatní (viz obrázek 7.4) . V případě XOR folding byly využity zhruba 2krát méně.

F3 a **F4**. Změna směru bitové rotace ani výměna sčítání za odčítání nemá vliv na fitness hodnotu. I grafy využití těchto funkcí jsou velmi podobné.

F5, **F7** a **F9**. Odstranění sčítání (F5) nebo XOR (F7) nezpůsobí změnu fitness hodnoty ve většině variant. Výjimkou je varianta, která používá 32bitové bloky a k redukci používá modulo. V tomto případě jde o zlepšení. Podle průměrných hodnot se zdá, že sčítání je užitečnější než XOR, ale rozdíl není statisticky významný. Odstranění obou těchto operátorů (F9) už způsobí zhoršení. Avšak při použití XOR folding se nejedná o tak velké zhoršení. To je způsobeno pravděpodobně tím, že XOR folding využívá operaci XOR implicitně při redukci.

F6. Vynechání násobení způsobí v případě kombinace 64bitových bloků a modulo zřejmé zhoršení. V ostatních případech není žádný statisticky významný rozdíl.

F8. Odstranění bitové rotace způsobí u varianty modulo výrazné zhoršení (medián fitness hodnoty je asi 0,24), avšak u varianty, která využívá 16bitové bloky nebo XOR folding nenastala téměř žádná změna.

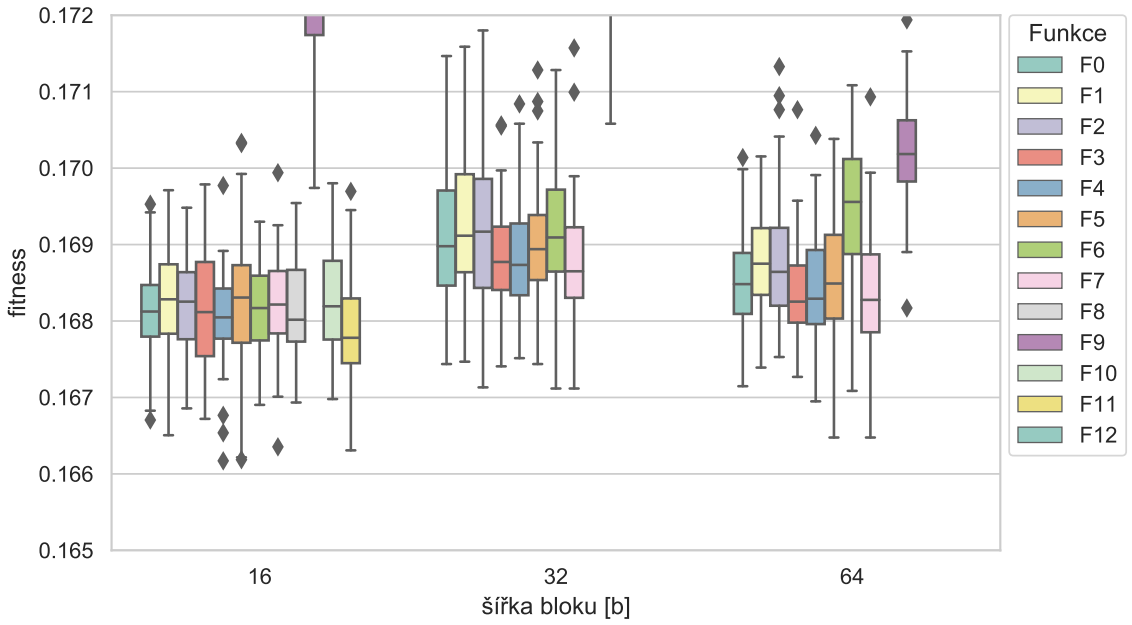
F10. Odstranění XOR a rotace. Zde je situace obdobná jako u F8. Skutečnost, že nedošlo ke zhoršení při použití XOR folding by se dalo opět vysvětlit tím, že XOR folding používá rotaci i XOR implicitně při redukci.

F11. Tato množina obsahuje pouze sčítání a je velmi překvapivé, že varianta, která využívá 16bitové nebo 32bitové bloky a XOR folding dosahuje nejlepších výsledků právě s touto množinou. U varianty modulo došlo opět k výraznému zhoršení.

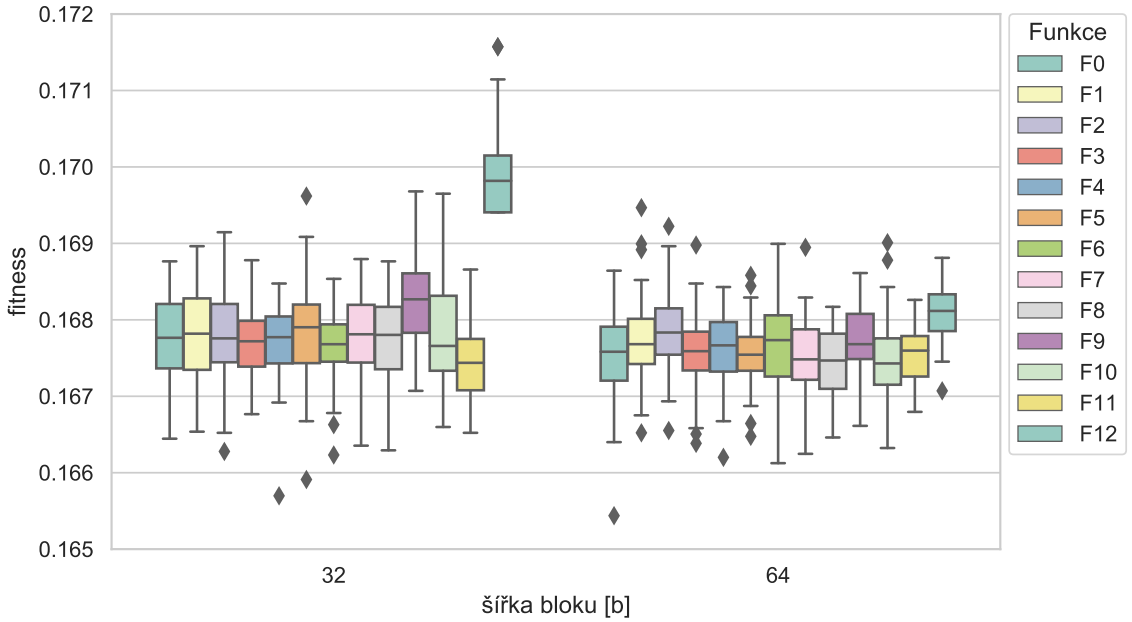
F12 obsahuje pouze násobení. U všech variant dochází k výraznému zhoršení (medián fitness hodnoty až 0,33). To by mohlo být způsobeno tím, že velké množství bloků obsahuje nulu, absorbující prvek vůči násobení. Což však bylo vyvráceno, protože žádný vstupní blok neobsahuje ani jednu nulu.

Z těchto testů je také na první pohled vidět, že varianta XOR folding dosahuje mnohem lepších výsledků v porovnání s modulo. To je pravděpodobně způsobeno tím, že pro tvorbu výsledného 15bitového výstupu využívá celou bitovou šířku výstupního uzlu, kdežto modulo využívá jenom 15 bitů a zbytek zahazuje.

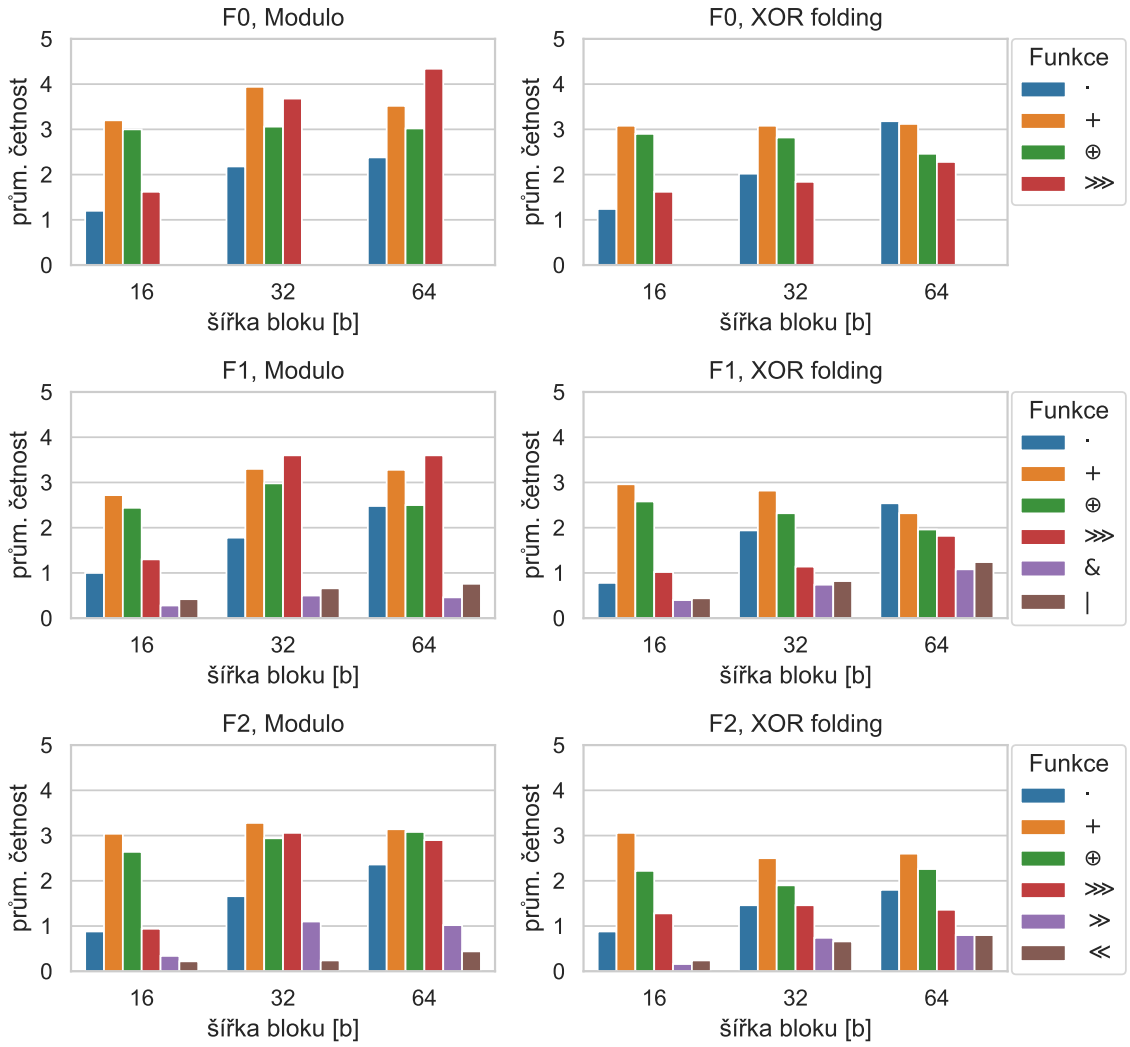
[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html



Obrázek 7.2: Vliv šířky bloků a množiny funkcí na fitness hodnotu. Varianta modulo. Množiny F8, F9, F10, F11 a F12 dosahují špatných výsledků (medián fitness hodnoty až 0,33) a proto nejsou viditelné.



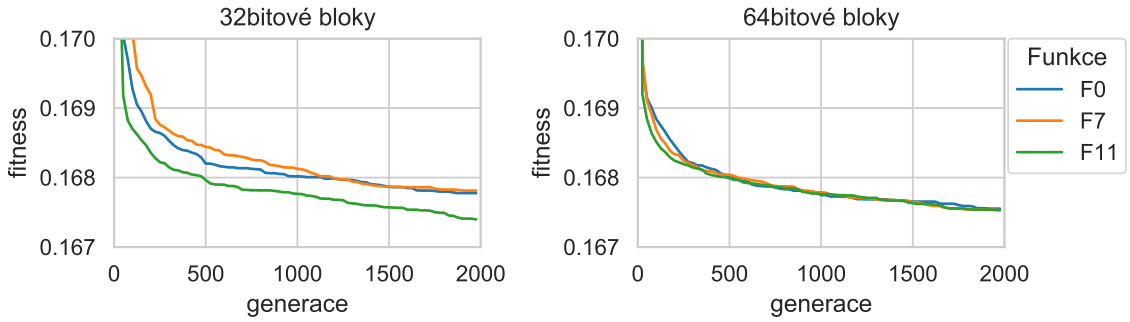
Obrázek 7.3: Vliv šířky bloků a množiny funkcí na fitness hodnotu. Varianta XOR folding.



Obrázek 7.4: Průměrný počet jednotlivých funkcí využitých nejlepšími kandidáty.

Na grafech četností jednotlivých funkcí na obrázku 7.4 je možno pozorovat, že užitečnost násobení a rotace se zvyšuje v závislosti na šířce bloků. Rotace se jeví jako obzvlášť důležitá pro variantu modulo. Užitečnost XOR v porovnání s AND a OR pravděpodobně spočívá v tom, že XOR je reverzibilní operátor, ale AND ani OR nejsou. Nechť $a \circ C = b$, kde C je konstanta. Operátor $(\circ)$ je reverzibilní pokud hodnota proměnné a se dá jednoznačně odvodit od hodnoty b. Bitové posuvy jsou pravděpodobně špatné z toho důvodu, že vstupní bloky obsahují vysoká čísla (větší než 64), takže ve většině případů je výsledkem této operace nula.

Nejlepších výsledků dosahují množiny F0 $(+, \cdot, \oplus, \ggg)$, F7 $(+, \cdot, \ggg)$ a F11 $(+)$. Množiny F0 a F7 produkují dobré výsledky napříč všemi variantami. F11 není tak univerzální, avšak při použití 16bitových bloků nebo varianty XOR folding může dosahovat lepších výsledků než F0 a F7. Průměrná fitness hodnota v průběhu evoluce s využití těchto třech množin a varianty XOR folding je na obrázku 7.5, kde jde vidět, že F11 je v případě 32bitových bloků efektivnější, ale u 64bitových bloků není téměř žádný rozdíl. Jelikož se 64bitová varianta jeví jako nejlepší, tak v následujících experimentech bude použita množina F0.



Obrázek 7.5: Průměrná fitness hodnota vybraných množin funkcí v průběhu evoluce. Varianta XOR folding.

7.3 Rozměry mřížky a pravděpodobnost mutace

Cílem tohoto experimentu je nalézt dobré nastavení parametrů kartézského genetického programování. Jedná se o počet řádků a sloupců a pravděpodobnost mutace. Počet řádků a sloupců byl měl být dostatečně velký, aby umožnil reprezentaci dobrého kandidátního řešení. Je však žádoucí vytvořit genotyp mnohem větší než je třeba, aby mohl obsahovat velké množství neaktivních uzlů, což, jak bylo řečeno v sekci 4.5, přispívá k efektivitě evoluce. Pravděpodobnost mutace by měla být dostatečně vysoká, aby umožnila rychlé nalezení dobrého řešení, ale neměla by být příliš vysoká, protože by působila spíše destruktivně.

Zvolil jsem, že celkový počet uzlů bude 10, 50 a 100 a každou z těchto kategorií jsem dále rozdělil na variantu s různým poměrem řádků a sloupců. Vznikly tyto varianty: 1×10, 2×5, 5×2, 1×50, 2×25, 5×10, 1×100, 2×50 a 5×20. Mutace je implementována tím způsobem, aby vyjadřovala počet genů v chromozomu, které budou zmutovány. Byly uvažovány následující hodnoty: 1 %, 10 %, 30 %, 50 % a 80 %. Počet generací byl zvýšen, v porovnání s předchozím experimentem, z 2000 na 10000. Nastavení ostatních parametrů je popsáno v tabulce 7.3. S každou kombinací pravděpodobnosti mutace a rozměrů mřížky bylo provedeno 30 běhů.

Množina funkcí	$+$, $\cdot$, $\oplus$, $\ggg$
Způsob redukce	XOR folding
Řádky × Sloupce	1×10, 2×5, 5×2, 1×50, 2×25, 5×10, 1×100, 2×50, 5×20
Vstupy	18 (16bit.), 9 (32bit.), 5 (64bit.)
Výstupy	2
Populace	$4 + 1$
Max. generace	10000
Mutace	1 %, 10 %, 30 %, 50 %, 80 %

Tabulka 7.3: Parametry CGP pro experiment s mřížkou CGP a pravděpodobností mutace.

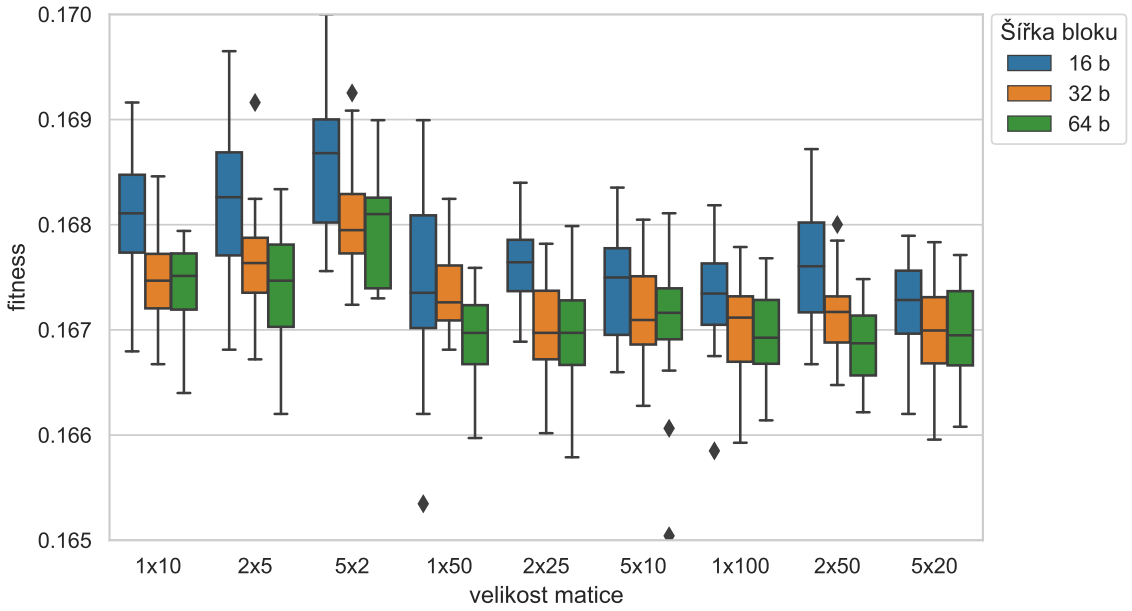
Výsledky

Jako první byla prozkoumána pravděpodobnost mutace. Z porovnání na obrázku 7.7 je jasně vidět, že pro všechny rozměry matice i vstupních bloků se jako nejlepší pravděpodobnost jeví 10 %. Hodnota 1 % se zdá příliš nízká a hodnoty větší než 10 % jsou příliš destruktivní

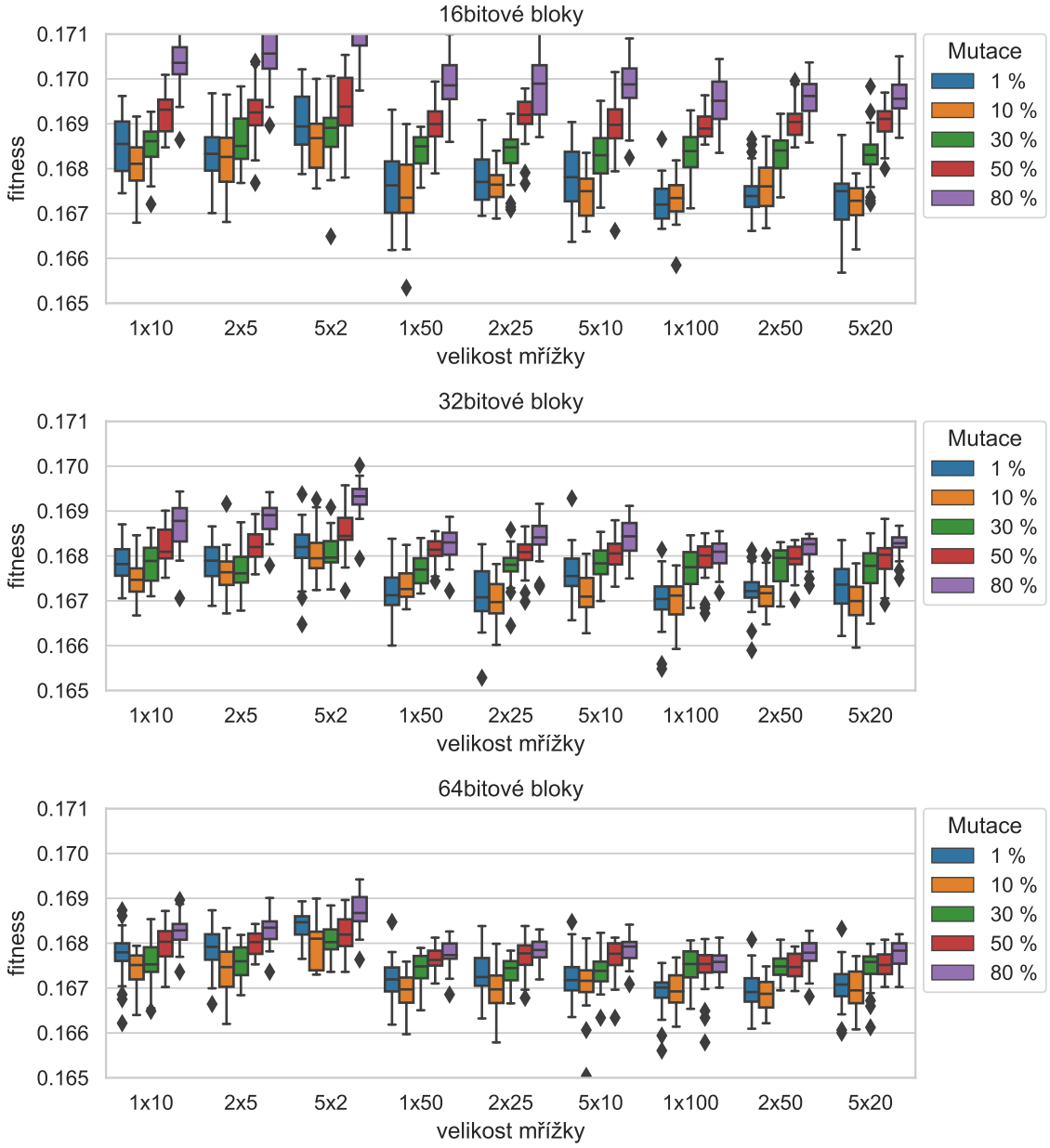
a postupně se přibližují efektivitě náhodného prohledávání. Další z poznatků je ten, že se zvětšující se šířkou vstupních bloků klesá vliv pravděpodobnosti mutace na výsledky. To je pravděpodobně způsobeno tím, že širší bloky znamenají menší počet vstupů a tím pádem menší prohledávací prostor, což daný problém zjednodušuje.

V grafu na obrázku 7.6 je porovnání jednotlivých variant šířky vstupních bloků, při pravděpodobnosti mutace 10 %. Téměř na všech rozměrech mřížky se jako nejlepší šířka jeví 64 bitů, ale i 32bitová varianta dosahuje porovnatelných výsledků, konkrétně při rozměrech 2×25 a 5×20. U všech variant je vidět, že větší počet uzlů vede na lepší výsledky s tím, že je preferovaný spíše menší počet řádků, který je méně restriktivní z hlediska tvaru možných řešení.

Z experimentu v této sekci bylo zjištěno, že nejvhodnější nastavení pravděpodobnosti mutace je 10 % a že nejlepších výsledků dosahuje varianta, která používá 64bitové vstupní bloky, 2 řádky a 50 sloupců. Průměrná fitness hodnota této varianty je 0,1669.



Obrázek 7.6: Fitness v závislosti na velikosti CGP mřížky a šířce vstupních bloků. Pravděpodobnost mutace je 10 %.



Obrázek 7.7: Fitness v závislosti na velikosti CGP mřížky, šířce vstupních bloků a pravděpodobnosti mutace.

7.4 Další metody mutace

Prostor pro zefektivnění evolučního běhu nabízí fakt, že v případě, kdy nedojde k mutaci aktivního uzlu, není potřeba ohodnocovat dané kandidátní řešení, protože má stejnou fitness hodnotu jako jeho rodič. Je zřejmé, že při nižší pravděpodobnosti mutace se zmenšuje šance na mutaci aktivního uzlu, a tedy roste užitečnost metody, která tuto skutečnost odhalí. V sekci 4.6 byly představeny metody **skip** a **single**. Skip tento problém řeší tak, že znovu neohodnocuje jedince, kteří mají stejné aktivní uzly jako jejich rodič. Single zaručuje, že nově vzniklý potomek se bude lišit alespoň v jednom aktivním uzlu.

Před zahájením experimentu bylo potřeba zjistit, jak nízká musí být hodnota pravděpodobnosti mutace, aby mělo smysl použít metodu skip. Provedl jsem 10 běhů s nastavením uvedeném v tabulce 7.4. Zaměřil jsem se na variantu, která dosahovala v předchozích experimentech nejlepších výsledků.

Množina funkcí	$+$, $\cdot$, $\oplus$, $\ggg$
Způsob redukce	XOR folding
Řádky × Sloupce	1×100
Vstupy	5 (64bit.)
Výstupy	2
Populace	$4 + 1$
Max. generace	1000 (bude modifikováno)
Mutace	Single, skip: 1 %, 5 %, 10 %

Tabulka 7.4: Parametry CGP pro experiment s dalšími metodami mutace.

Výsledky

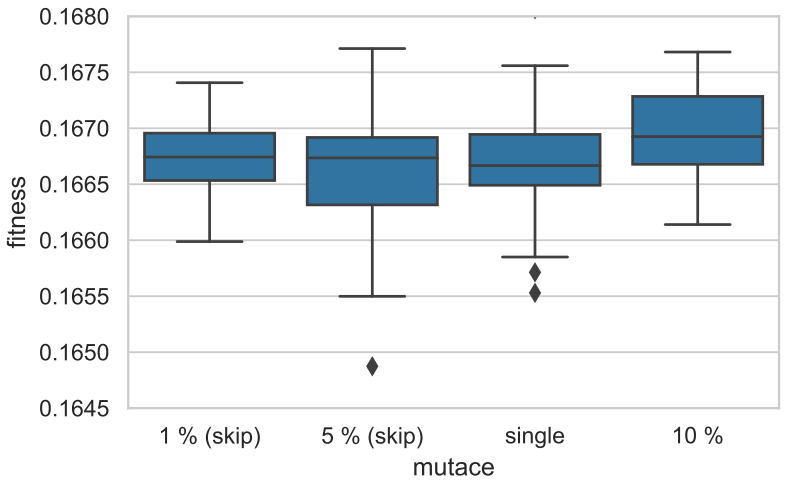
V jednom běhu s nastavením z tabulky 7.4 se prozkoumá 5000 jedinců, respektive 4001, protože zbylých 999 jsou rodiče dané generace, jejichž fitness hodnoty jsou známé z předchozí generace. Pouze v první generaci je potřeba ohodnotit všech 5 kandidátů. Tabulka 7.5 ukazuje, kolik kandidátních řešení není potřeba ohodnocovat, díky metodě skip. A jak je vidět, tak při pravděpodobnosti větší než 10 % nemá smysl používat skip, protože téměř všechna kandidátní řešení se liší od svého rodiče. Na druhou stranu, při pravděpodobnosti 1 % jde o značnou úsporu, protože průměrně více než polovinu kandidátů není potřeba ohodnocovat.

Mutace	Průměr	Odchylka	Min.	Max.
10 %	15	26	0	79
5 %	217	148	47	504
1 %	2242	367	1718	2701

Tabulka 7.5: Počet kandidátních řešení, které není potřeba ohodnocovat. Rodiče generací nejsou započítáni. Výsledky jsou zaokrouhleny na celá čísla.

Všechny experimenty doposud používaly jako ukončovací podmínku evoluce dosažení maximálního počtu generací. Po implementaci metod, které neprovádějí zbytečné evaluace, dává větší smysl jako ukončovací podmínku použít počet evaluací. Z tohoto důvodu jsem provedl 30 běhů s nastavením uvedeném v tabulce 7.4, ale jednotlivé běhy jsem ukončoval až po dosažení 50000 evaluací.

Výsledky jsem porovnal s nejlepších nastavením nalezeném v experimentu ze sekce 7.3 (10 %). Obrázek 7.8 zobrazuje dosažené výsledky, ze kterých je vidět, že metody skip i single dosahují, při stejném počtu evaluací, lepších fitness hodnot než doposud nejlepší nalezené nastavení. Jelikož drtivou většinu času (asi 90 %) zabere ohodnocení jedince, tak čas potřebný na provedení těchto metod je zanedbatelný.



Obrázek 7.8: Porovnání metod mutace.

7.5 Zrychlená fitness funkce

Jeden ze způsobu zrychlení výpočtu spočívá v modifikaci fitness funkce. Funkce, kterou jsem doposud používal, funguje tak, že nejprve vypočítá haš všech klíčů v testovací sadě pomocí obou hašovacích funkcí a poté vkládá všechny tyto klíče do hašovací tabulky a provádí přemisťování podle kukaččího hašování.

Idea je taková, že v případě špatného řešení, které vyprodukuje velké množství kolizí už během vkládání prvních klíčů, je možné vkládání přerušit a dané řešení zahodit. Takže původní fitness funkci jsem upravil do podoby kterou používali Dobai a Kořenek [8]. Jakmile nastane první selhání, je vkládání ukončeno a výsledná hodnota fitness funkce udává počet úspěšně vloženích klíčů, než nastalo první selhání.

Cílem tohoto experimentu je zjistit, jak dlouho bude trvat evoluce na zrychlené fitness funkci a zdali vůbec jsou nalezená řešení dostatečně dobrá, aby mohla konkurovat řešením nalezeným pomocí původní fitness funkce.

Pro tento experiment jsem použil parametry uvedené v tabulce 7.6. S každou kombinací šířky bloků a velikostí mřížky bylo provedeno 30 běhů.

Množina funkcí	$+$, $\cdot$, $\oplus$, $\ggg$
Způsob redukce	XOR folding
Řádky × Sloupce	1×100, 2×50, 5×20
Vstupy	18 (16bit.), 9 (32bit.), 5 (64bit.)
Výstupy	2
Populace	$4 + 1$
Max. generace	10000
Mutace	10 %

Tabulka 7.6: Parametry CGP pro experiment se zrychlenou fitness funkcí.

Výsledky

Nejprve byl prozkoumán vliv na dobu evoluce. Při této části byl počet generací snížen na 1000 a byla použita varianta $1+100$. Výsledky se nachází v tabulce 7.7. Jedná se o průměrné hodnoty.

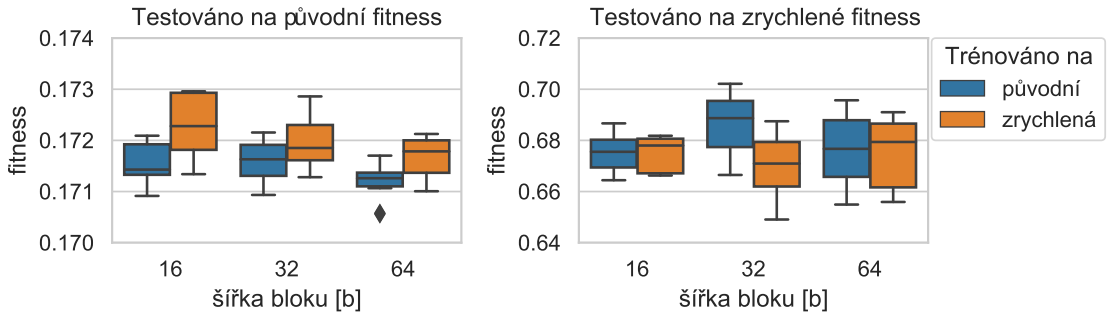
Šířka bloku	Doba běhu (původní)	Doba běhu (zrychlená)	Zrychlení
16 bitů	10,8 s	3,7 s	2,9×
32 bitů	13,4 s	5,8 s	2,3×
64 bitů	16,3 s	9,4 s	1,7×

Tabulka 7.7: Průměrná doba jednoho evolučního běhu.

Poté jsem nastavil parametry tak, jak jsou uvedeny v tabulce 7.6, a provedl jsem trénování na původní i zrychlené fitness funkci. Celkově bylo pro každou šířku bloků provedeno 90 běhů. Z těchto 90 běhů jsem vzal 10 těch nejlepších a ty jsem otestoval na 8 testovacích sadách, představených v sekci 6.1, s použitím původní i zrychlené fitness funkce. Pro každé kandidátní řešení jsem vypočítal průměr všech testovacích sad. Z těchto průměrů byly vykresleny grafy na obrázku 7.9.

Jak je z výsledků vidět, tak při testování na zrychlené fitness funkci dosahují nejlepších výsledků řešení, která používala tuto funkci i při trénování. S výjimkou 32bitových bloků však jde o velmi malé rozdíly.

Pokud jsou nalezená řešení testována podle původní fitness funkce, tak ve všech případech jde o zhoršení. Bylo sice dosaženo zhruba dvojnásobné zrychlení evoluce, ale za cenu kvality výsledných řešení, což je z praktického hlediska důležitější.



Obrázek 7.9: Porovnání fitness funkcí. Pro každé kandidátní řešení byl vypočítán průměr fitness hodnot na všech testovacích sadách. Z těchto průměrů byly vykresleny tyto grafy.

7.6 Porovnání se state-of-the-art hašovacími funkcemi

V této sekci budou detailněji prozkoumána nejlepší doposud nalezená řešení. Nejprve budou zmíněny základní informace o nejlepších kandidátech a hašovaných datech. Poté bude provedeno porovnání variant šířky vstupních bloků a metod redukce. A nakonec budou nejlepší kandidáti porovnáni s existujícími hašovacími funkcemi, které navrhli experti v tomto oboru.

Pokud by nalezená řešení byla seřazena podle fitness hodnoty na trénovací sadě, tak mezi prvními 30 by bylo 14 64bitových, 13 32bitových a 3 16bitové dvojice hašovacích funkcí. Aby každá varianta měla rovnoměrné zastoupení, tak bylo od každé varianty použito 10 nejlepších kandidátů. Tímto způsobem bylo vybráno 30 řešení, které provádí redukci pomocí XOR folding a 30 řešení, které používají pouze modulo.

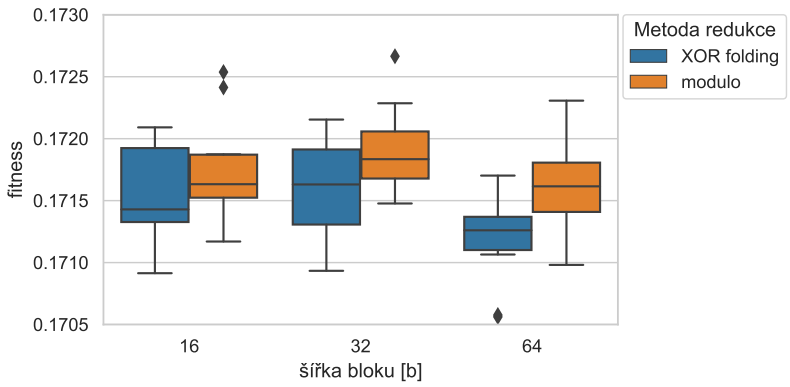
Veškeré testováni v této sekci je prováděno pomocí testovacích sad (viz sekce 6.1).

Výsledky

Zajímalo mě, jaké vstupní bloky jsou nejvíce využívané a jestli existuje nějaká korelace mezi využitím jednotlivých vstupních bloků a počtem různých hodnot v každém z bloků. Na obrázku 7.11 jsou vykresleny tyto statistiky. Byla použita pouze varianta XOR folding.

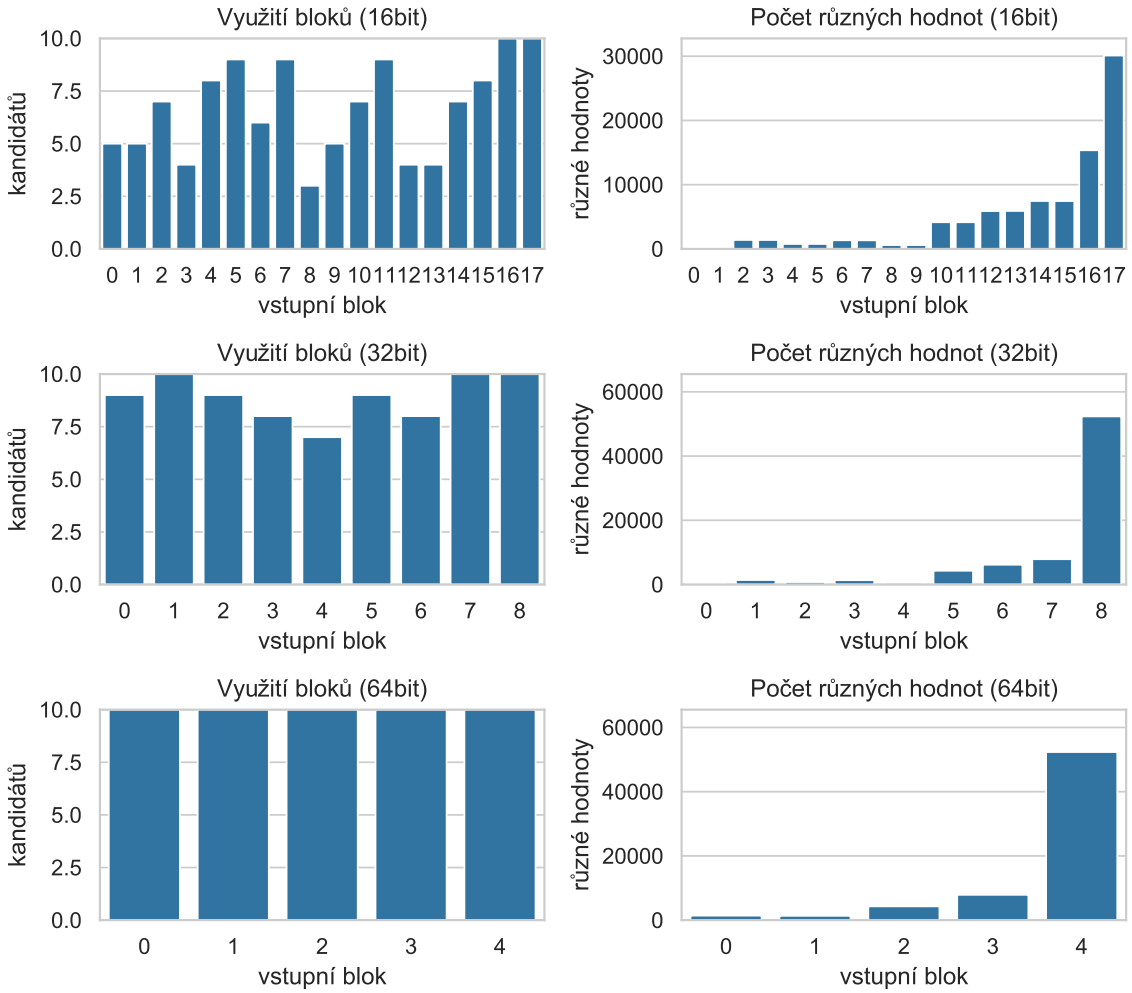
Nejvíce různých hodnot se nachází v blocích, které obsahují zdrojový a cílový port. Tyto bloky jsou využity všemi kandidáty u všech šířek bloků. Pouze 64bitová varianta využívá všechny vstupní bloky, u ostatních variant se zvyšuje celkový počet bloků a klesá jejich průměrné využití. Při analýze struktury kandidátních řešení bylo zjištěno, že všechny varianty jsou průměrně tvořeny asi 20 aktivními uzly (což je 20 % maximální velikosti při rozměrech 1×100). Maximálně bylo aktivováno 33 uzlů. Je možné, že 16bitová varianta dosahuje horších výsledků právě kvůli tomu, že nedokáže využít všechny vstupní bloky, což může být způsobeno příliš malými rozměry mřížky CGP.

Výsledky porovnání varianty XOR folding a modulo lze vidět na obrázku 7.10. Byla vypočítána fitness hodnota každého kandidátního řešení na každé z osmi testovacích sad. Pro každé kandidátní řešení byl spočítán průměr všech testovacích sad. Z těchto průměrů je vykreslen tento graf, ze kterého lze vyčíst, že se potvrdila hypotéza, že nejlepších výsledků dosahuje varianta, která používá 64bitové bloky a k redukci na 15 bitů používá XOR folding.

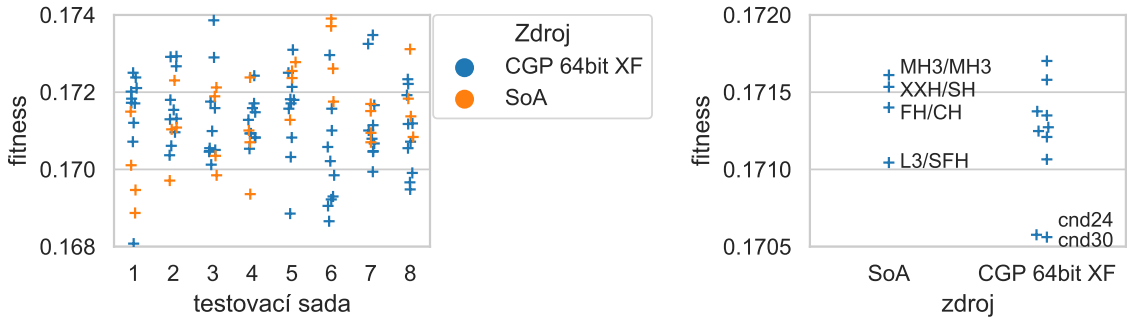


Obrázek 7.10: Porovnání metod redukce. Průměr výsledků ze všech testovacích sad.

Z existujících hašovacích funkcí jsem použil MurmurHash3, FarmHash, CityHash, Lookup3, SuperFastHash, xxHash a SpookyHash. Z těchto funkcí jsem pro potřeby kukaččího hašování vytvořil dvojice. Snažil jsem se párovat moderní s moderními a starší se staršími. Provedl jsem ohodnocení na testovacích sadách. Výsledky jsou na obrázku 7.12, ze který vyplývá, že dvojice hašovacích funkcí vytvořené pomocí CGP dosahují srovnatelných výsledků jako funkce navržené experty v tomto oboru. Byla dokonce nalezena dvě kandidátní řešení, která mají lepší průměrné výsledky.



Obrázek 7.11: Využití vstupních bloků deseti nejlepšími kandidáty. Včetně počtu různých hodnot v každém z bloků.



Obrázek 7.12: Porovnání state-of-the-art (SoA) hašovacích funkcí s evolučně navrženými. Levý graf zobrazuje výsledky na jednotlivých testovacích sadách. Graf vpravo zobrazuje průměr všech sad. Použité zkratky: MH3 – MurmurHash3, FH – FarmHash, CH – CityHash, L3 – Lookup3, SFH – SuperFastHash, XXH – xxHash a SH – SpookyHash.

```
#include <bit>

uint64_t cnd30_h1(uint64_t in[]) {
    return (((rotr(in[1], in[3]) + (rotr(in[1], in[3]) ^ in[4])) *
       (rotr(in[4], in[0]) ^ in[2])) * rotr((in[0] + in[4]), in[3] *
       in[2]));
}

uint64_t cnd30_h2(uint64_t in[]) {
    return (rotr(in[1], in[3]) * (rotr(in[4], in[0]) + rotr(((rotr(in[1],
       in[3]) + (rotr(in[1], in[3]) ^ in[4])) ^ (rotr((in[3] * in[2]),
       rotr(in[4], in[0])) + in[4])), (in[3] ^ (rotr(in[4], in[0]) ^
       in[2])))));
}
```

Výpis 7.1: Kandidátní řešení `cnd30` implementované v jazyce C++.

Po porovnání nejlepších kandidátních řešení a state-of-the-art funkcí byla dvě nejlepší řešení implementována v jazyce C++ (viz výpis 7.1, který obsahuje jedno z nich). U těchto implementací byla změřena doba potřebná pro vložení všech síťových toků z datové sady do hašovací tabulky. Tabulka 7.8 obsahuje průměrnou dobu vložení všech testovacích sad. K měření byla použita knihovna `std::chrono`. I přestože evolučně navržené funkce nebyly optimalizovány pro rychlost, tak dosahují srovnatelné rychlosti jako state-of-the-art funkce.

Hašovací funkce	Fitness hodnota	Doba výpočtu[µs]
cnd30	0.1706	1213
cnd24	0.1706	1470
L3/SFH	0.1710	1327
cnd26	0.1711	—
cnd23	0.1712	—
cnd25	0.1712	—
cnd22	0.1713	—
cnd27	0.1713	—
cnd28	0.1714	—
FH/CH	0.1714	859
XXH/SH	0.1715	1248
cnd29	0.1716	—
MH3/MH3	0.1716	1356
cnd21	0.1717	—

Tabulka 7.8: Výsledky nejlepších řešení (cndX) a state-of-the-art funkcí na testovacích sadách. Jedná se o průměrné hodnoty. Doba výpočtu je doba potřebná pro vložení všech toků z jedné testovací sady do hašovací tabulky. Použité zkratky: MH3 – MurmurHash3, FH – FarmHash, CH – CityHash, L3 – Lookup3, SFH – SuperFastHash, XXH – xxHash a SH – SpookyHash.

7.7 Shrnutí výsledků experimentů

V této kapitole bylo experimentálně hledáno vhodné nastavení parametrů CGP.

Sekce 7.1 se zabývala vlivem hodnoty konstanty *MaxLoop* na fitness hodnotu a na dobu výpočtu. Bylo zjištěno, že doba výpočtu roste lineárně v závislosti na *MaxLoop*, ale přírůstky fitness hodnoty se postupně zmenšují.

Následně v sekci 7.2 byla hledána vhodná množina funkcí. Jako nejvíce univerzální se jeví množina obsahující právě sčítání, násobení, XOR a bitovou rotaci. Ukázalo se, že kandidátní řešení, které používají XOR folding jsou odolné vůči změnám této množiny. V tomto případě stačí, aby množina funkcí obsahovala pouze sčítání. Přidání bitových operací AND a OR ani bitových posuvů nepřineslo žádné zlepšení.

Poté v sekci 7.3 byla zkoumána velikost mřížky CGP a pravděpodobnost mutace. Výsledky ukazují, že pro vytvoření dobrých řešení stačí, aby mřížka CGP obsahovala 50 uzlů. Nejlepších výsledků bylo dosaženo při nastavení pravděpodobnosti mutace na 10 %.

Sekce 7.4 porovnávala další metody mutace. Bylo zjištěno, že při nízkých hodnotách pravděpodobnosti mutace (menší než 10 %) se značná část kandidátních řešení neliší od svých rodičů. Díky zavedení metody skip je možné tyto kandidáty odhalit a ušetřit procesorový čas. Zároveň bylo ukázáno, že vhodnou alternativou může být metoda single, která nevyžaduje nastavení pravděpodobnosti mutace.

Dále v sekci 7.5 byla využita alternativní fitness funkce. Tato fitness funkce značně urychlila evoluci za cenu mírného zhoršení kvality výsledných řešení.

Poslední sekce 7.6 porovnala nejlepší nalezená řešení s hašovacími funkcemi, které navrhli experti v tomto oboru. Ukázalo se, že evolučně vytvořené hašovací funkce mohou konkurovat existujícím funkcím. Dokonce byly nalezeny i funkce, které dosahovaly lepších výsledků. Zároveň se v této sekci ukázalo, že nejlepších výsledků dosahuje varianta, která používá 64bitové vstupní bloky i operace a k redukci na požadovaných 15 bitů používá XOR folding.

Přehledné shrnutí nastavení všech parametrů CGP je vidět v tabulce 7.9.

Množina funkcí	$+, \cdot, \oplus, \ggg$
Způsob redukce	XOR folding
Řádky × Sloupce	1×100
Vstupy	18 (16bit.)
Výstupy	2
Populace	$4 + 1$
Metoda mutace	Skip
Pravděpodobnost mutace	5 %

Tabulka 7.9: Nejvhodnější nastavení parametrů CGP nalezené v této kapitole.

Kapitola 8

Závěr

Cílem práce bylo navrhnout, implementovat a experimentálně vyhodnotit metodu pro automatizovaný návrh hašovacích funkcí.

Nejprve byly představeny hašovací funkce v kontextu hašovacích tabulek včetně několika metod, které umožňují vypořádat se s kolizemi, a kritérií, které by dobrá hašovací funkce měla splňovat. Poté bylo vysvětleno genetické programování obecně a následně byly detailně popsány kroky kartézského genetického programování a také různé metody mutace, konkrétně skip a single. V rámci práce byla zpracována studie, která shrnuje přístup jiných autorů k návrhu hašovacích funkcí pomocí evolučních algoritmů a vyzdvihuje, čím jsou jednotlivé publikace zajímavé a přínosné.

Pro evoluční návrh bylo využito kartézské genetické programování a jako metoda řešení kolizí bylo zvoleno kukaččí hašování. Tyto metody byly implementovány v jazyce C++.

Vytvořené hašovací funkce byly porovnávány na datové sadě obsahující IPv6 síťové toky. V rámci experimentů bylo zjišťováno, jestli lepších výsledků dosáhnou hašovací funkce využívající 16bitové, 32bitové nebo 64bitové operace a vstupní bloky. A také byly uvažovány dvě metody redukce výsledných hašů, konkrétně modulo a XOR folding. Bylo zkoumáno, jak tyto varianty ovlivní množinu potřebných funkcí a další parametry CGP.

Ukázalo se, že varianta XOR folding dosahuje lepších výsledků než modulo a že není tak citlivá na volbu množiny funkcí. Pro vytvoření dobrých řešení stačí, aby množina funkcí obsahovala pouze sčítání. Avšak modulo potřebuje, kromě sčítání, také bitovou rotaci, násobení a případně i XOR. Co se týká bitové šířky těchto funkcí, tak nejlepších výsledků dosahuje 64bitová varianta. Metody single i skip se ukázaly být efektivnější než klasická bodová mutace. Nejlepší řešení nalezená pomocí CGP byla porovnána s existujícími hašovacími funkcemi, které navrhli experti v tomto oboru. Ukázalo se, že CGP dokáže najít konkurenceschopné hašovací funkce z hlediska počtu kolizí i rychlosti hašování.

V práci by se dalo pokračovat zavedením vícekriteriální optimalizace a kromě počtu kolizí optimalizovat i rychlost hašování. Dalo by se zkoumat, jak výše zmíněné varianty ovlivní výsledky.

Literatura

[1] Appleby, A. *MurmurHash3* [online]. 2010 [cit. 2022-24-02]. Dostupné z: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[2] Bäck, T. a Kok, J. N. *Handbook of Natural Computing*. 1. vyd. Springer Berlin Heidelberg, 2012. ISBN 978-3-540-92911-6.

[3] Cao, Z. a Wang, Z. Flow identification for supporting per-flow queueing. In: *Proceedings Ninth International Conference on Computer Communications and Networks (Cat.No.00EX440)*. 2000, s. 88–93. ISBN 0-7803-6494-5.

[4] Chi, L. a Zhu, X. Hashing Techniques: A Survey and Taxonomy. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. duben 2017, sv. 50, č. 1. ISSN 0360-0300.

[5] Collet, Y. *XxHash* [online]. 2015 [cit. 2022-24-02]. Dostupné z: http://cyan4973.github.io/xxHash/.

[6] Damgård, I. B. A Design Principle for Hash Functions. In: Brassard, G., ed. *Advances in Cryptology — CRYPTO' 89 Proceedings*. New York, NY: Springer New York, 1990, s. 416–427. ISBN 978-0-387-34805-6.

[7] Deb, K., Pratap, A., Agarwal, S. a Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*. IEEE. 2002, sv. 6, č. 2, s. 182–197.

[8] Dobai, R. a Kořenek, J. Evolution of Non-Cryptographic Hash Function Pairs for FPGA-Based Network Applications. In: *2015 IEEE Symposium Series on Computational Intelligence*. Institute of Electrical and Electronics Engineers, 2015, s. 1214–1219. ISBN 978-1-4799-7560-0.

[9] Estébanez, C., Saez, Y., Recio, G. a Isasi, P. Automatic design of noncryptographic hash functions using genetic programming. *Comput. Intell.* USA: Blackwell Publishers, Inc. nov 2014, sv. 30, č. 4, s. 798–831. ISSN 0824-7935.

[10] Estébanez, C., Saez, Y., Recio, G. a Isasi, P. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*. 2014, sv. 44, č. 6, s. 681–698.

[11] Floreano, D. a Mattiussi, C. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008. ISBN 978-0-262-06271-8.

[12] Fowler, G., Vo, P. a Noll, L. C. *FNV Hash* [online]. 1991 [cit. 2022-24-02]. Dostupné z: http://isthe.com/chongo/tech/comp/fnv/.

[13] Gang, W. a Soule, T. How to Choose Appropriate Function Sets for GP. In: Keijzer, M., O'Reilly, U.-M., Lucas, S. M., Costa, E. a Soule, T., ed. *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*. Coimbra, Portugal: Springer-Verlag, červenec 2004, sv. 3003, s. 198–207. LNCS. ISBN 3-540-21346-5.

[14] Goldman, B. W. a Punch, W. F. Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms. *IEEE Transactions on Evolutionary Computation*. 2015, sv. 19, č. 3, s. 359–373.

[15] Goodrich, M. T. a Tamassia, R. *Algorithm Design: Foundations, Analysis and Internet Examples*. 2. vyd. USA: John Wiley & Sons, Inc., 2009. ISBN 0470088540.

[16] Grochol, D. a Sekanina, L. Evolutionary Design of Hash Functions for IPv6 Network Flow Hashing. In: *IEEE Congress on Evolutionary Computation*. IEEE Computational Intelligence Society, 2020, s. 1–8. ISBN 978-1-7281-6929-3.

[17] Hsieh, P. *Hash Functions* [online]. 2004 [cit. 2022-24-02]. Dostupné z: http://www.azillionmonkeys.com/qed/hash.html.

[18] Hunter, J., Dale, D., Firing, E. a Droettboom, M. *Matplotlib.pyplot.boxplot* [online]. 2022 [cit. 2022-30-04]. Dostupné z: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html.

[19] Jenkins, B. *Lookup3* [online]. 2006 [cit. 2022-24-02]. Dostupné z: http://burtleburtle.net/bob/c/lookup3.c.

[20] Karásek, J., Burget, R. a Morský, O. Towards an automatic design of non-cryptographic hash function. In: *2011 34th International Conference on Telecommunications and Signal Processing (TSP)*. 2011, s. 19–23. ISBN 978-1-4577-1411-5.

[21] Kidoň, M. a Dobai, R. Evolutionary design of hash functions for IP address hashing using genetic programming. In: *IEEE Congress on Evolutionary Computation (CEC)*. 2017, s. 1720–1727. ISBN 978-1-5090-4601-0.

[22] Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2. vyd. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN 0201896850.

[23] Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*. Jun 1994, sv. 4, č. 2, s. 87–112. ISSN 1573-1375.

[24] Merkle, R. C. One Way Hash Functions and DES. In: Brassard, G., ed. *Advances in Cryptology — CRYPTO' 89 Proceedings*. New York, NY: Springer New York, 1990, s. 428–446. ISBN 978-0-387-34805-6.

[25] Miller, J. F. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, s. 1135–1142. GECCO'99. ISBN 1558606114.

[26] Miller, J. F. Cartesian Genetic Programming. In: Miller, J. F., ed. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 17–34. ISBN 978-3-642-17310-3.

[27] Miller, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*. Springer. 2020, sv. 21, č. 1, s. 129–168.

[28] Pagh, R. a Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*. Elsevier. 2004, sv. 51, č. 2, s. 122–144.

[29] Partow, A. *General Purpose Hash Function Algorithms* [online]. 2010 [cit. 2022-24-02]. Dostupné z: http://www.partow.net/programming/hashfunctions/.

[30] Pike, G. *Introducing FarmHash* [online]. 2014 [cit. 2022-24-02]. Dostupné z: https://opensource.googleblog.com/2014/03/introducing-farmhash.html.

[31] Pike, G. a Alakuijala, J. *Introducing CityHash* [online]. 2011 [cit. 2022-24-02]. Dostupné z: https://opensource.googleblog.com/2011/04/introducing-cityhash.html.

[32] Poli, R., Langdon, W. a Mcphee, N. *A Field Guide to Genetic Programming*. Leden 2008. ISBN 978-1-4092-0073-4.

[33] Saez, Y., Estébanez, C., Quintana, D. a Isasi, P. Evolutionary hash functions for specific domains. *Applied Soft Computing*. 2019, sv. 78, s. 58–69. ISSN 1568-4946.

[34] Vassilev, V. K. a Miller, J. F. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Miller, J., Thompson, A., Thomson, P. a Fogarty, T. C., ed. *Evolvable Systems: From Biology to Hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, s. 252–263. ISBN 978-3-540-46406-8.

[35] Wassenberg, J. a Alakuijala, J. *HighwayHash* [online]. 2017 [cit. 2022-24-02]. Dostupné z: https://github.com/google/highwayhash/blob/master/README.md.

[36] Češka, M., Matyáš, J., Mrázek, V., Sekanina, L., Vašíček, Z. et al. SagTree: Towards efficient mutation in evolutionary circuit approximation. *Swarm and Evolutionary Computation*. 2022, sv. 69, s. 100986. ISSN 2210-6502.

Příloha A

Obsah paměťového média a spuštění programu

Přiložené paměťové médium obsahuje:

- `src/cgp/` – zdrojové soubory CGP
- `src/datasets/` – trénovací a testovací datové sady
- `src/experiments/` – skripty pro vygenerování konfiguračních souborů a spuštění experimentů
- `src/result_analysis/` – výsledky experimentů a skript pro vygenerování grafů
- `src/other/` – zdrojové soubory state-of-the-art hašovacích funkcí
- `latex/` – zdrojové soubory tohoto textu
- `text.pdf` – tento dokument
- `text-tisk.pdf` – tento dokument, verze pro tisk
- `README.txt` – návod na překlad a spuštění

Pro překlad je potřeba program CMake[1] (testováno na verzi 3.16.3). Konfigurace CGP se provádí pomocí souboru `config.h` (viz sekce 6.4). Příklad spuštění programu na počítači s operačním systémem Ubuntu:

```
$ cd src
src$ cmake -DCMAKE_BUILD_TYPE=Release -S . -B build
src$ cmake --build build --target prg64
src$ cd build
src/build$ ./prg64
```

[1] https://cmake.org/