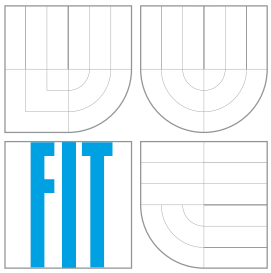
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SOUBĚŽNÉ UČENÍ V KOEVOLUČNÍCH ALGORITMECH

COLEARNING IN COEVOLUTIONARY ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL WIGLASZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAELA ŠIKULOVÁ

BRNO 2015

Zadání diplomové práce/17108/2014/xwigla00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2014/2015

Zadání diplomové práce

Řešitel: **Wiglasz Michal, Bc.**
Obor: Bioinformatika a biocomputing
Téma: **Souběžné učení v koevolučních algoritmech**
Colearning in Coevolutionary Algorithms
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou evolučních algoritmů spojených s učením, genetického programování, koevolučních algoritmů a evolučního návrhu obrazových filtrů.
2. Navrhněte program umožňující provádět evoluční návrh obrazových filtrů s využitím koevoluce prediktorů fitness. Velikost prediktorů fitness adaptujte v průběhu koevoluce jako reakci na konvergenci fitness při vývoji kandidátních filtrů.
3. Program z bodu 2 implementujte.
4. Ověřte funkčnost programu na zadaných úlohách.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucí práce.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2 zadání.

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

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

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

Vedoucí: **Šikulová Michaela, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2014
Datum odevzdání: 27. května 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Kartézské genetické programování je druh genetického programování, ve kterém jsou kandidátní programy reprezentovány jako orientované acyklické grafy. Bylo ukázáno, že je možné evoluci kartézských programů urychlit použitím koevoluce, kde se ve druhé populaci vyvíjí prediktory fitness. Prediktory fitness slouží k přibližnému určení kvality kandidátních řešení. Nevýhodou koevolučního přístupu je nutnost provést mnoho časově náročných experimentů pro určení nejvýhodnější velikosti prediktoru pro daný problém. V této práci je představena nová reprezentace prediktorů fitness s plastickým fenotypem, založená na principech souběžného učení v evolučních algoritmech. Plasticita fenotypu umožňuje odvodit různé fenotypy ze stejného genotypu. Díky tomu je možné adaptovat velikost prediktoru na současný průběh evoluce a obtížnost řešeného problému. Navržený algoritmus byl implementován v jazyce C a optimalizován pomocí vektorových instrukcí SSE2 a AVX2. Z experimentů vyplývá, že použitím plastického fenotypu lze dosáhnout srovnatelné kvalitních obrazových filtrů jako u standardního CGP při kratší době běhu programu (průměrné zrychlení je 8,6násobné) a zároveň odpadá nutnost hledání nejvýhodnější velikosti prediktoru jako u koevoluce s prediktory s fixní velikostí.

Abstract

Cartesian genetic programming (CGP) is a form of genetic programming where candidate programs are represented in the form of directed acyclic graphs. It was shown that CGP can be accelerated using coevolution with a population of fitness predictors which are used to estimate the quality of candidate solutions. The major disadvantage of the coevolutionary approach is the necessity of performing many time-consuming experiments to determine the best size of the fitness predictor for the particular task. This project introduces a new fitness predictor representation with phenotype plasticity, based on the principles of colearning in evolutionary algorithms. Phenotype plasticity allows to derive various phenotypes from the same genotype. This allows to adapt the size of the predictors to the current state of the evolution and difficulty of the solved problem. The proposed algorithm was implemented in the C language and optimized using SSE2 and AVX2 vector instructions. The experimental results show that the resulting image filters are comparable with standard CGP in terms of filtering quality. The average speedup is 8.6 compared to standard CGP. The speed is comparable to standard coevolutionary CGP but it is not necessary to experimentally determine the best size of the fitness predictor while applying coevolution to a new, unknown task.

Klíčová slova

Koevoluční algoritmus, kartézské genetické programování, evoluční algoritmus, Baldwinův efekt, plasticita fitness, predikce fitness, zpracování obrazu.

Keywords

Coevolutionary alghorithm, cartesian genetic programming, evolutionary algorithm, Baldwin effect, fitness plasticity, fitness predictor, image processing.

Citace

Michal Wiglasz: Souběžné učení v koevolučních algoritmech, diplomová práce, Brno, FIT VUT v Brně, 2015

Souběžné učení v koevolučních algoritmech

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michaely Šikulové. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Wiglasz
25. května 2015

Poděkování

Děkuji Ing. Michaele Šikulové nejen za cenné rady, náměty a nápady, ale především za její trpělivost, která je zřejmě nekonečná, a za veškerý čas, který mi během roku věnovala.

Dále děkuji Ing. Radku Hrbáčkovi, který mi ušetřil spoustu času tím, že připravené experimenty spouštěl na superpočítači Anselm.

Děkuji všem, kteří mě během studia podporovali, všem, kteří jen tiše našlapovali okolo, když jsem potřeboval klid, všem, kteří mi v těžkých chvílích pomohli dobrou radou, pošťouchnutím správným směrem, hřejivým slovem nebo i teplou večeří. Rád bych vás zde všechny vyjmenoval, bohužel tato stránka je příliš malá. Rozhodně ale nemohu nezmínit své rodiče a svého staršího bratra.

Také děkuji všem kapelám, zpěvákům, zpěvačkám a muzikantům, kteří nahráli a vydali spoustu úžasných desek, díky kterým mi práce lépe ubíhala.

Děkuji Českým drahám, že na trasu Brno–Bohumín konečně zařazují i vagony se stoly a zásuvkami, ve kterých se dá poměrně pohodlně pracovat.

Děkuji IT4Innovations a MetaCentru za výpočetní prostředky poskytnuté v rámci projektů Centrum excelence IT4Innovations (CZ.1.05/1.1.00/02.0070) a Velká infrastruktura CESNET (LM2010005).

Obsah

Kapitola 1

Úvod

Ačkoli se objevují nové a efektivnější algoritmy, stále existují problémy, které neumíme řešit běžným „inženýrským“ přístupem, protože zatím nebyl objeven vhodný teoretický popis. I pokud takový popis existuje, zahrnuje pouze podmnožinu všech možných reprezentací řešení. Prohledáváním v prostoru všech možných kombinací dostupných výpočetních prvků pak můžeme nalézt řešení, kterých nelze dosáhnout konvenčními metodami, ale mohou mít některé výhodné vlastnosti. I když je dnes běžně dostupný dříve nepředstavitelný výpočetní výkon, je mnohdy nemožné prohledat celý stavový prostor a vybrat nejlepší řešení. Proto se používají metody, které se snaží usměrňovat prohledávání správným směrem a zároveň prohledávaný prostor co nejvíce zmenšit, to vše aniž by utrpěla kvalita nalezeného řešení.

Mnohé metody umělé inteligence se inspirují v přírodě, například neuronové sítě, mravenčí kolonie, výpočty simulující rojení hmyzu nebo evoluční algoritmy, které mají svůj základ v Darwinově teorii o vzniku druhů. Evoluční algoritmy se ukazují jako výhodné pro různé optimalizační problémy, ale také lze pomocí genetického programování automaticky tvořit počítačové programy, elektrické obvody, optické systémy a dokonce i umělecká díla.

Jednou z úspěšných aplikací genetického programování je automatizovaná tvorba obrazových filtrů. Ty umožňují zrekonstruovat obrázky poškozené při přenosu dat nebo kvůli nefunkčním bodům ve snímači fotoaparátu či kamery. Často jsou součástí předzpracování dat u úloh zpracovávajících obraz a na jejich kvalitě závisí úspěšnost celého algoritmu. Také jsou významné v kosmonautice, kde je obtížné zamezit chybám při přenosu fotografií na Zemi nebo dokonce opravit či vyměnit kameru umístěnou na vesmírné sondě.

Tato práce se zabývá návrhem programu, který bude schopen tvořit obrazové filtry pomocí koevolučního algoritmu. Vedle obrazových filtrů se vyvíjí i populace tzv. prediktorů fitness, které slouží k urychlení výpočtu kvality každého vytvořeného filtru. Protože výběr vhodné velikosti prediktorů není triviální a někdy je nutné ke stanovení správné hodnoty provést velké množství experimentů, je nově do algoritmu zahrnuto souběžné učení, jehož cílem je adaptovat velikost prediktorů na aktuální průběh evoluce filtrů.

Tato diplomová práce je strukturována následovně: Kapitola 2 se zabývá evolučními algoritmy. Nejprve jsou představeny obecné principy evolučních algoritmů (v části 2.1), genetický algoritmus 2.2 a kartézské genetické programování 2.3. Následuje popis evolučního návrhu obrazových filtrů 2.4. Předposlední část 2.5 je věnována koevolučním algoritmům, především ve spojitosti s kartézským genetickým programováním, a závěrečná část 2.6 se zabývá souběžným učením, Baldwinovým efektem a plasticitou fitness. Kapitola 3 se zabývá návrhem řešení tvorby obrazových filtrů s použitím koevolučního algoritmu se souběžným učením. Jeho implementací se zabývá kapitola 4. Závěrečná kapitola 5 popisuje provedené experimenty a srovnání navrženého algoritmu se standardním a koevolučním CGP.

Kapitola 2

Evoluční algoritmy

První zmínky o evolučních algoritmech se v literatuře objevují od 50. let 20. století. Postupně a nezávisle na sobě vzniklo několik podobných metod (například evoluční strategie, evoluční programování nebo genetické algoritmy), které se v různých obměnách používají i dnes. Až s rozšířením spolupráce mezi těmito nezávislými výzkumnými týmy v 90. letech vznikl pojem „evoluční algoritmus“ [18].

2.1 Základní principy evolučních algoritmů

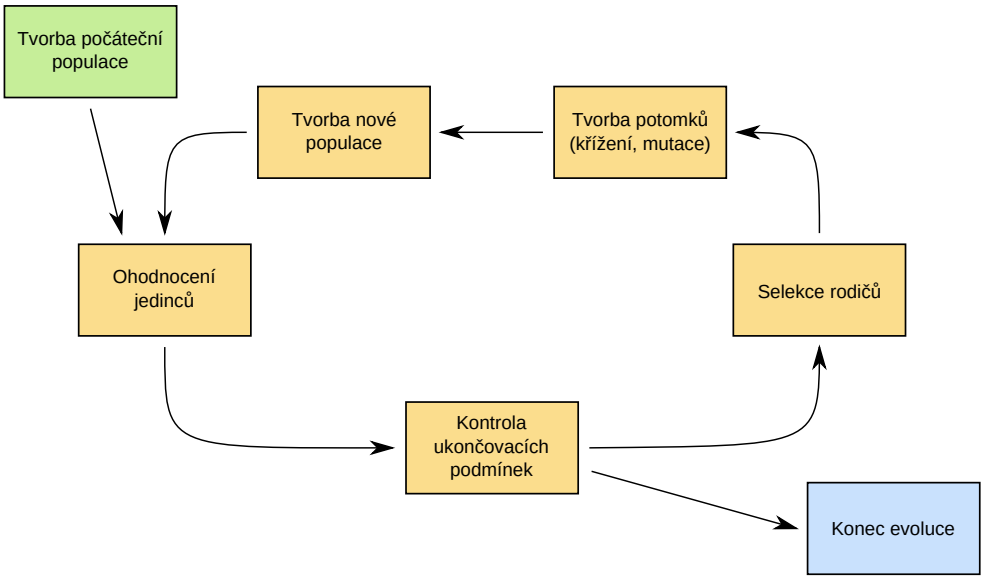
Evoluční algoritmy jsou stoachastické optimalizační metody inspirované přírodou, konkrétně Darwinovou teorií evoluce. Ačkoliv pojmy používané v evolučních algoritmech vycházejí z biologie, jejich význam není vždy přesně stejný. Mezi základní pojmy patří:

- *Gen* je základním stavebním blokem kandidátního řešení, je omezen předem danou abecedou (binární čísla, písmena apod.).
- *Alela* je konkrétní varianta genu.
- *Chromozom* nebo *jedinec* reprezentuje jedno řešení ve stavovém prostoru.
- *Genotyp* je posloupnost genů kódující řešení.
- *Fenotyp* je kandidátní řešení odvozené z genotypu.
- *Populace* je konečná množina kandidátních řešení.
- *Fitness* udává „kvalitu“ jedince s ohledem na řešený problém.
- *Fitness funkce* přiřazuje každému jedinci právě jedno reálné číslo – hodnotu fitness.
- *Diverzita* udává rozmanitost populace, tj. nakolik se od sebe chromozomy v populaci navzájem liší.

Typickým rysem evolučních algoritmů je to, že pracují s populací několika kandidátních řešení, od kterých odvozují nová řešení pomocí speciálních biologií inspirovaných operátorů. Jednoduchý evoluční algoritmus znázorňuje schéma na obrázku 2.1 a lze zapsat takto:

1. Náhodně vygeneruj populaci o dané velikosti.

2. Opakuj, dokud není splněna ukončovací podmínka:
   (a) urči fitness jedinců v populaci,
   (b) z populace vyber rodiče a vytvoř potomky,
   (c) některé jedince v populaci nahraď potomky.
3. Výsledkem je jedinec s nejvyšší fitness (nejlepší nalezené řešení problému).



Obrázek 2.1. Schéma evolučního algoritmu

Nevýhodou evolučních algoritmů je velké množství parametrů a podproblémů. Před samotným výpočtem pomocí evolučního algoritmu je třeba si položit například tyto otázky:

- Jak se budou kódovat kandidátní řešení do genotypu?
- Kolik jedinců má být v populaci?
- Jak se budou vybírat rodiče nové populace?
- Jak se budou tvořit potomci?
- Kolik jedinců v populaci má být nahrazeno potomky?

Je třeba brát v potaz i stochastickou povahu evolučních algoritmů, z čehož plyne, že pro porovnání různých algoritmů (nebo různých hodnot parametrů) je potřeba statisticky vyhodnotit větší množství běhů [9, 18].

2.2 Genetický algoritmus

Genetický algoritmus byl vytvořen Johnem Hollandem v sedmdesátých letech. Samotný termín „genetický algoritmus“ se začal běžně používat až po publikaci dizertační práce

Kena De Jonga v roce 1975. Do širšího povědomí se dostává v půlce osmdesátých let, kdy se ukázalo, že jej lze použít k řešení řady obtížných úloh.

Schéma genetického algoritmu odpovídá obecnému evolučnímu algoritmu. Počáteční populace je vytvořena náhodně. V každé iteraci se populace ohodnotí (určí se fitness jedinců) a podle zvoleného selekčního algoritmu se vyberou rodiče, ze kterých jsou pomocí operátorů křížení a mutace vytvořeni potomci. Kromě potomků je možné v nové populaci zachovat i několik jedinců z původní populace (s případnou mutací). Pokud se používá *elitismus*, je nejlepší jedinec vždy součástí nové populace [18, 29].

2.2.1 Reprezentace řešení

Genotyp v genetickém algoritmu je často jednoduchá datová struktura, jako je vektor bitů nebo čísel konstantní délky. V případně binárního chromozomu může být vhodné použít Grayovo kódování, ve kterém se dvě po sobě jdoucí hodnoty liší vždy pouze jedním bitem. To pomáhá překonat tzv. *Hammingovu bariéru*, kdy malá změna v genotypu způsobí velkou změnu ve fenotypu a naopak. Například ve fenotypu je mezi čísly 15 a 16 malý rozdíl, ale v binárním genotypu se liší o 5 bitů [29].

2.2.2 Selekce rodičů

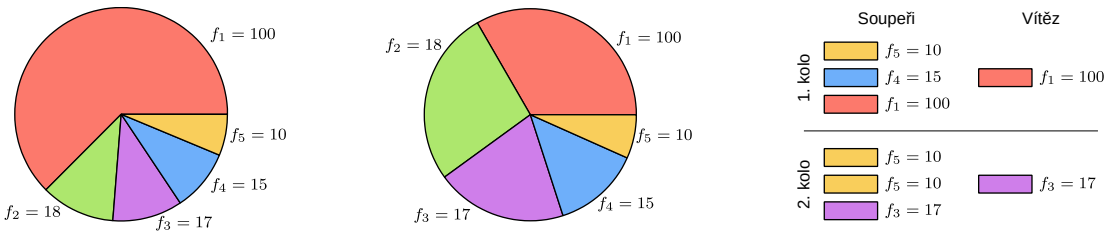
Volbu K rodičů z populace lze provést několika způsoby. Nejjednodušší je prosté seřazení jedinců podle jejich fitness, rodiči se pak stává K jedinců s nejvyšší fitness. O trochu složitější je ruletová a turnajová selekce, které znázorňuje obrázek 2.2.

U *ruletové selekce* je pravděpodobnost, že se konkrétní jedinec stane rodičem, přímo úměrná jeho fitness:

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

Lze si to představit jako ruletu, kde každá výseč odpovídá jednomu jedinci a čím vyšší má jedinec fitness, tím je příslušná výseč větší. Nevýhodou je, že pokud má jeden nebo více jedinců výrazně vyšší fitness než zbytek, stávají se tito jedinci téměř vždy rodiči a zmenšuje se diverzita populace. Jedním ze způsobů, jak lze tento problém řešit je, že se jedinci seřadí podle velikosti fitness a pravděpodobnost výběru je úměrná pořadí jedince a ne jeho fitness.

Jiným přístupem k výběru rodičů je *turnajová selekce*. Do turnajového kola jsou náhodně vybráni dva nebo více jedinců. Vítězem kola se stává jedinec s nejvyšší fitness a ten se stává rodičem. Turnaj se opakuje tolikrát, kolik rodičů je potřeba zvolit. Může se stát, že mezi rodiči se bude některý jedinec opakovat [18].

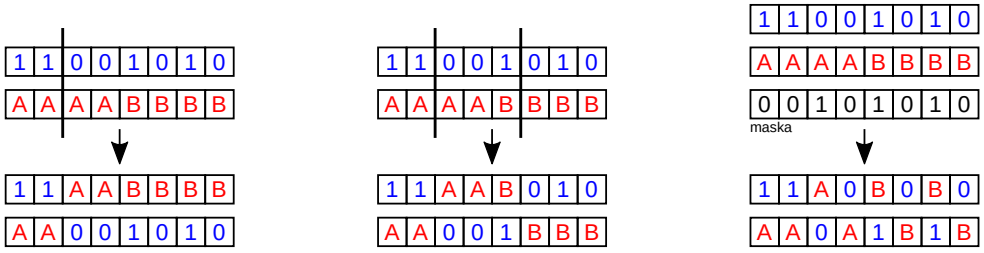


(a) Ruleta podle fitness (b) Ruleta podle pořadí (c) Turnaj

Obrázek 2.2. Mechanismy selekce v genetickém algoritmu

2.2.3 Křížení

Křížení je považováno za hlavní operátor genetického algoritmu. Během něj je vytvořen nový jedinec jako kombinace genů dvou nebo více chromozomů. Obrázek 2.3 znázorňuje tři základní mechanismy: jednobodové, vícebodové a uniformní křížení. U *jednobodového křížení* je náhodně zvolen bod, ve kterém se chromozomy rozdělí a geny za tímto bodem si mezi sebou vymění. U *vícebodového křížení* je těchto bodů několik. V případě *uniformního křížení* se rozhoduje, zda dojde k prohození nebo ne, u každého genu zvlášť [18].



(a) Jednobodové (b) Vícebodové (c) Uniformní

Obrázek 2.3. Tři základní mechanismy křížení v genetickém algoritmu

2.2.4 Mutace

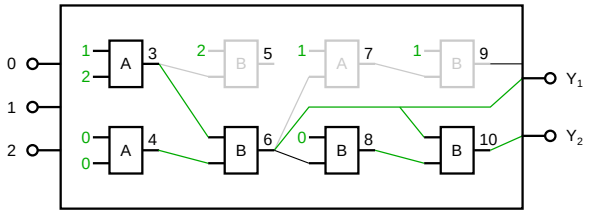
Mutace je malá změna genotypu, která se s malou pravděpodobností provádí u potomků vzniklých křížením. V případě binárně kódovaného chromozomu mutace spočívá v překlopení několika náhodně zvolených bitů. U celočíselného kódování je hodnota genu nahrazena náhodným číslem. Existují také speciální mutační operátory pro reálná čísla nebo pro permutačně kódované chromozomy. Pokud je z povahy problému hodnota genu omezená výčtem nebo intervalem, je vhodné, aby výsledkem mutace nebyla nepřípustná hodnota, aby nevznikala neplatná řešení [18].

2.3 Kartézské genetické programování

Další variantu evolučního algoritmu, genetické programování, představil koncem 80. let 20. století John Koza. Umožňuje automatizovanou tvorbu celých programů, kdy neřeší jak má program pracovat, pouze co má být jeho výstupem. John Koza pracoval s jazykem LISP, pro který je vhodná reprezentace programu pomocí stromů. Existují ale i jiné způsoby kódování programů do chromozomu, například ve formě kartézských programů. Kartézské genetické programování (CGP) představil Julian Miller koncem devadesátých let [13].

V CGP se programy kódují jako orientované acyklické grafy, reprezentované dvourozměrnou kartézskou mřížkou výpočetních uzlů (funkčních bloků) o předem daných rozměrech. Příklad kartézského programu je zobrazen na obrázku 2.4. Počet primárních vstupů a výstupů je fixní. Každý výpočetní uzel vykonává nějakou funkci z předem daného seznamu. Ten je vhodné přizpůsobit řešené úloze, například pro tvorbu logických obvodů to mohou být dostupná logická hradla, pro symbolickou regresi aritmetické operace, ale jsou i práce, které se zabývají návrhem obvodů na úrovni tranzistorů pomocí CGP [14]. Vstupy funkčních bloků jsou napojeny buď na některý z primárních vstupů, nebo na výstup uzlu umístěného v některém sloupci nalevo. O kolik sloupců doleva je možné uzly propojovat

určuje parametr l-back. Pokud je roven jedné, je možné propojovat pouze uzly ze sousedních sloupců, pokud je l-back roven počtu sloupců, lze uzly propojovat libovolně. Zpětné vazby a cykly nejsou v CGP povoleny.



Chromozom: {1, 2, A; 0, 0, A; 2, 3, B; 3, 4, B; 1, 6, B; 0, 6, A; 1, 7, B; 6, 8, B; 6, 10}

Obrázek 2.4. Program v kartézském genetickém programování. Uzly 5, 7 a 9 jsou neaktivní

Chromozom je posloupnost celých čísel konstantní délky, jednotlivé geny určují funkci výpočetních uzlů a jejich propojení. Na konci chromozomu je pro každý primární výstup jeden gen obsahující číslo uzlu, jehož výstup má být použit. Velikost fenotypu je variabilní, ale nikdy nepřesáhne velikost genotypu. Je to proto, že ne všechny funkční bloky jsou přímo či nepřímo napojeny na primární výstupy programu – hovoříme o *neaktivních blocích*.

Výhodou kartézského genetického programování je omezený stavový prostor, nehrozí, že by délka programu během evoluce rostla nad únosnou mez, tak jako u stromové reprezentace. Prohledávací prostor lze dále omezovat parametrem l-back a omezením množiny dostupných funkcí [12, 18, 27].

2.3.1 Průběh evoluce

Schéma evoluce opět odpovídá obecnému evolučnímu algoritmu. Počáteční populace je vygenerována náhodně. Oproti genetickému algoritmu se u kartézského genetického programování většinou nepoužívá křížení, protože jeho přínos pro konvergenci řešení není velký. Po ohodnocení všech jedinců je nejlepší z nich určen rodičem. Ostatní jedinci jsou nahrazeni potomky – náhodnými mutacemi rodiče. Tento způsob tvorby nových generací se označuje jako evoluční strategie $(1 + \lambda)$ [18].

2.3.2 Mutace

Mutace mění náhodně hodnotu některých genů jedince. Jejich počet se běžně volí jako procento z celkového počtu genů, označované jako *míra mutace*. Geny nelze nahrazovat libovolnými hodnotami, je třeba brát ohled na parametr l-back, počet primárních vstupů a počet dostupných funkcí pro výpočetní uzly:

- U genu kódujících funkci, je náhodně vygenerován index funkce ze seznamu dostupných funkcí.
- U genu kódujících vstup funkčního bloku jsou povolenými hodnotami:
  - číslo primárního vstupu programu,
  - číslo některého uzlu ze sloupců nalevo, s ohledem na parametr l-back.
- U genu kódujících primární výstup je náhodně vybráno číslo některého výpočetního uzlu nebo primárního vstupu.

Ne každá mutace vede na změnu fenotypu – například může dojít ke změně funkce neaktivního bloku nebo propojení mezi neaktivními bloky. Ukazuje se ale, že i tyto *neutrální mutace* jsou prospěšné pro konvergenci algoritmu. Proto pokud je v populaci několik nejlepších jedinců se stejnou hodnotou fitness, je vhodné vybírat jako rodiče toho, který nebyl rodičem v předchozí generaci, tj. chromozom, který vznikl neutrální mutací. V opačném případě, kdy se rodičem může stát pouze jedinec s vyšší hodnotou fitness, evoluce nachází méně kvalitní programy.

Ačkoliv mutace mění pouze malý počet genů a v genetických algoritmech slouží spíše jako doplňkový operátor, v případě CGP může malá mutace vést na velkou změnu fenotypu. Například po změně pouze jednoho propojení funkčních bloků se může funkce programu zásadně změnit, protože se najednou stane aktivními větší množství funkčních bloků. Mutace tak v CGP slouží i jako extenzivní operátor [12, 18].

2.3.3 Výpočet fitness

Genetické programování umožňuje automatizovaně vytvářet programy s požadovaným chováním. Chování – požadované odezvy programu na zadané vstupy – je definováno v *trénovací množině*. Každý prvek této množiny reprezentuje jeden *případ fitness*. Například pokud je cílem nalézt program aproximující polynom $x^4 + x^3 + x^2 + x$ nad množinou přirozených čísel menších než deset, je každé z těchto čísel jedním případem fitness. V případě programu $x^2 + 1$ je hodnota $2^2 + 1 = 5$ odezva programu pro případ fitness 2. Fitness kandidátního řešení je pak určena jako součet odchylek požadovaného a skutečného výstupu programu pro všechny případy fitness z trénovací množiny.

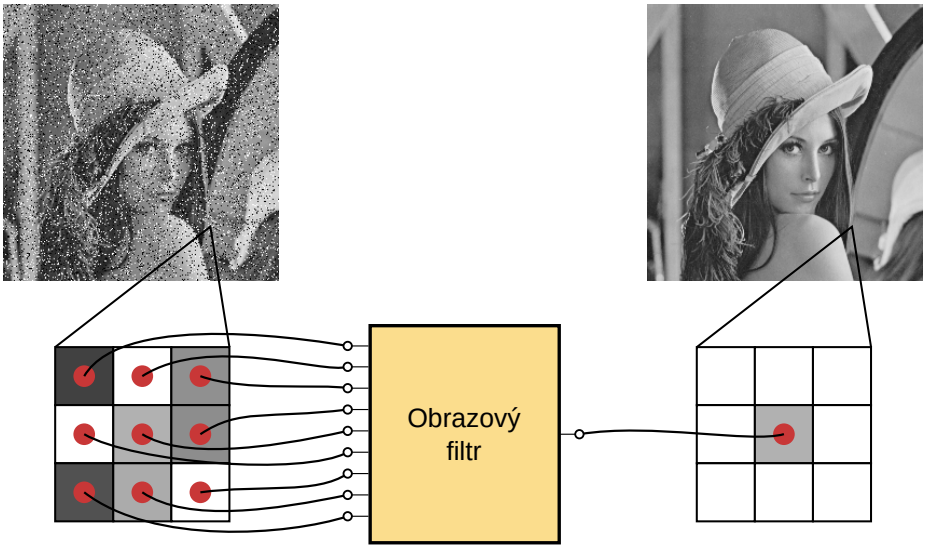
U složitějších problémů může být takových případů fitness velmi mnoho. Protože výpočet fitness je časově nejnáročnější částí evoluce, je vhodné do trénovací množiny zahrnout pouze část případů fitness a urychlit tak výpočet. Na druhou stranu je třeba velikost množiny a používané případy fitness zvolit tak, aby i pro kombinace vstupů neobsažené v trénovací množině byly výstupy programu v souladu s požadovaným chováním.

Samotný výpočet fitness lze v CGP provádět „zleva doprava“, kdy jsou postupně vypočítávány výstupy všech funkčních bloků, bez ohledu na to, zda jsou použity ve fenotypu nebo ne. Druhou možností, kdy se pracuje pouze s aktivními bloky, je postupovat rekurzivním sestupem od výstupů programu ke vstupům. Nevýhodou je, že některé uzly se mohou vyhodnotit vícekrát. Jako ideální se jeví nejprve určit aktivní bloky a poté směrem od vstupů programu vypočítat výstupy aktivních bloků a tím i celého programu. Aktivní bloky lze určit i bez použití rekurze, průchodem po jednotlivých uzlech od konce programu. Jako aktivní se nejprve označí uzly připojené na primární výstupy. Při průchodu pak platí, že pokud je uzel aktivní, označí se za aktivní i uzly připojené na jeho vstupy [18, 27].

2.4 Návrh obrazových filtrů evolučními algoritmy

Jedna z úloh, kterou lze řešit pomocí evolučních algoritmů je návrh obrazových filtrů [19]. Ty se používají jako jeden z prvních kroků u úloh zpracování obrazových dat. Čím lépe filtr dokáže obnovit poškozené části obrazu, tím lepší výsledky lze získat v dalších krocích algoritmu, jako je například segmentace nebo klasifikace.

Většina hardwarových i softwarových implementací obrazových filtrů pracuje s lokálním okolím pixelů, nejčastěji se používá okolí 9 nebo 25 pixelů. Novou hodnotu pixelu určuje



Obrázek 2.5. Princip filtrace obrazu

funkce, na jejímž vstupu jsou hodnoty všech pixelů ve zvoleném okolí. Tato funkce se postupně aplikuje na celý obrázek. Tento princip znázorňuje obrázek 2.5.

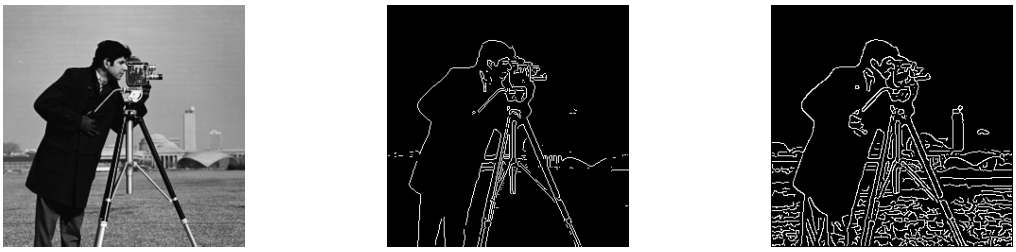
Obrazové filtry lze rozdělit na lineární a nelineární. U lineárních platí princip superpozice a lze je charakterizovat pomocí impulzní odezvy. Výsledný obraz je dán konvolucí vstupního obrazu a impulzní odezvy – na konvoluci lze nahlížet jako na vážený součet zvoleného okolí pixelu. U nelineárních filtrů není výsledný obraz dán konvolucí, ale nějakou nelineární funkcí, například mediánem hodnot okolních pixelů.

Existují různé typy šumů, pro které jsou vhodné různé typy obrazových filtrů. Poměrně častý je impulzní šum, který vzniká kvůli nefunkčním pixelům v kameře, chybným paměťovým buňkám nebo chybám při přenosu dat. Poškozené body buď mají vždy minimální nebo maximální možnou hodnotu (šum typu sůl a pepř) nebo mají náhodnou hodnotu. Pro impulzní šum je vhodný nelineární mediánový filtr, ale při intenzivnějším šumu je kvalita nedostatečná a také dochází ke ztrátě detailů, jak je vidět na obrázcích 2.6.

(a) Vstupní obrázek (b) Mediánový filtr s oknem 3×3 (c) Mediánový filtr s oknem 5×5

Obrázek 2.6. Výstup mediánového filtru pro obrázek s 25% šumem typu sůl a pepř

Obrazové filtry se uplatňují i v jiných úlohách než filtrování šumu. Jedním z kroků některých algoritmů zpracování obrazu bývá detekce hran. Obrazový filtr má nyní za úkol označit pixely, které jsou ve vstupním obrázku součástí nějaké hrany. Mezi běžně používané detektory hran patří Sobelův nebo Cannyho detektor.



(a) Vstupní obrázek (b) Sobelův detektor (c) Cannyho detektor

Obrázek 2.7. Výstup některých detektorů hran

Evoluční algoritmy lze při tvorbě obrazových filtrů využít různými způsoby. Jednou možností je použít již existující obrazový filtr a u něj pomocí genetického algoritmu optimalizovat nastavení jeho parametrů tak, aby se dosáhlo co nejlepších výsledků v zamýšleném prostředí. Evolučními algoritmy ale lze také přímo navrhovat nové obrazové filtry. Zvláště výhodné je to u nelineárních filtrů, pro jejichž analýzu a návrh schází vhodný matematický aparát a jejich tvorba je tak obtížnější než v případě lineárních filtrů.

Jako vhodné pro tuto úlohu se ukazuje kartézské genetické programování [19]. Na vstup kartézského programu je přivedeno zvolené okolí pixelu, výstupem je filtrovaný pixel. Jako fitness funkce se nejčastěji používá špičková hodnota poměru signál/šum v decibelech (Peak Signal to Noise Ratio, PSNR) nebo střední odchylka pixelů (Mean Difference Per Pixel, MDPP), která je vhodnější pro hardwarovou implementaci:

$$PSNR = 10 \log_{10} \frac{255^2}{\frac{1}{MN} \sum_{i,j} \left(v\left(i,j\right) - w\left(i,j\right) \right)^2} \tag{2.2}$$

$$MDPP = \frac{1}{MN} \sum_{i}^{M} \sum_{j}^{N} \left| v\left(i,j\right) - w\left(i,j\right) \right| \tag{2.3}$$

kde M a N označují rozměry obrázku, v filtrovaný a w původní obrázek.

Pro výpočet fitness kandidátního filtru musíme zpracovat celý obrázek, přičemž každý pixel můžeme považovat za jeden případ fitness. Protože i pro poměrně malé obrázky jich je několik tisíc, je výpočet fitness časově velmi náročný [18, 19].

2.5 Koevoluce v kartézském genetickém programování

Jedním ze způsobů jak řešit problém velkého množství případů fitness (zmíněný v části 2.3.3) je použití koevolučního algoritmu [15]. Oproti dosud zmíněným evolučním algoritmům v obecných koevolučních algoritmech existuje několik populací, které na sebe navzájem působí a ovlivňují svůj vývoj. Fitness jedince pak nezávisí pouze na jeho genetických předpokladech, ale určuje se podle toho, jak „dobrý“ je při interakci s jedinci z jiných populací.

Koevoluce s populacemi různého druhu se používá pro úlohy založené na testu. Populace mezi sebou interagují prostřednictvím fitness funkce, jejíž výsledek závisí i na jedincích okolních populací. V jedné populaci jsou vyvíjena kandidátní řešení problému, druhá populace obsahuje *testy*, což jsou podmnožiny množiny případů fitness. Typicky jsou jedinci jedné populace ohodnocování pomocí nejlepšího nebo několika nejlepších jedinců z jiné populace. Příkladem tohoto typu koevoluce je *soutěživá koevoluce*, ve které mezi sebou soutěží jedinci

ze dvou populací. Cílem populace kandidátních řešení je správně vyřešit všechny případy fitness v nejlepším testu, naopak cílem populace testů je najít ty případy fitness, ve kterých nejlepší kandidátní řešení selhává. Soutěživou koevoluci poprvé představil Daniel Hillis na úloze návrhu řadicích sítí [3]. Řadicí síť je algoritmus, který řadí posloupnosti dané délky pomocí posloupnosti komparátorů, které mají dva vstupy a výstupy. Komparátor porovnává prvky na vstupy mezi sebou, a pokud nejsou v požadovaném pořadí, na výstupu je zamění. V soutěživé koevoluci tvoří řadicí sítě populaci kandidátních řešení, druhou populaci testů tvoří neseřazené posloupnosti. Na kandidátní sítě lze nahlížet jako na „hostitele“ a na neseřazené posloupnosti jako na „parazity“. Cílem evoluce řadicích sítí je správně seřadit posloupnost v nejlepším testu, cílem evoluce testů je nalézt takovou neseřazenou posloupnost, která na výstupu sítě není ve správném pořadí.

Jiným příkladem koevoluce je *kompoziční koevoluce*, kterou lze použít pro řešení některých složitějších problémů, které lze dekomponovat na několik jednodušších. Řešení jednotlivých podproblémů mohou být reprezentovány různými chromozomy. Jedinci pak spolu spolupracují na řešení úlohy a jejich kvalita závisí i na kvalitě ostatních jedinců.

Další variantou je typ koevoluce, kdy jsou populace stejného druhu. Takové jsou navzájem oddělené bariérou a vyvíjejí se nezávisle na sobě až na občasné migrace jedinců mezi podpopulacemi (například pokud některá z nich uvízne v lokálním extrému).

Koevoluční algoritmy využívají kromě populací i jiný typ množiny jedinců – tzv. *archivy*, do kterých jsou umisťování nejlepší nalezení jedinci.

Z implementačního hlediska je možné vyhradit pro každou z populací samostatné vlákno nebo proces, které mezi sebou komunikují zasíláním zpráv nebo přes sdílenou paměť. Není to ale nutné, koevoluci lze modelovat také sekvenčně tak, že v každé iteraci algoritmu proběhne několik generací v první populaci a poté několik generací ve druhé populaci.

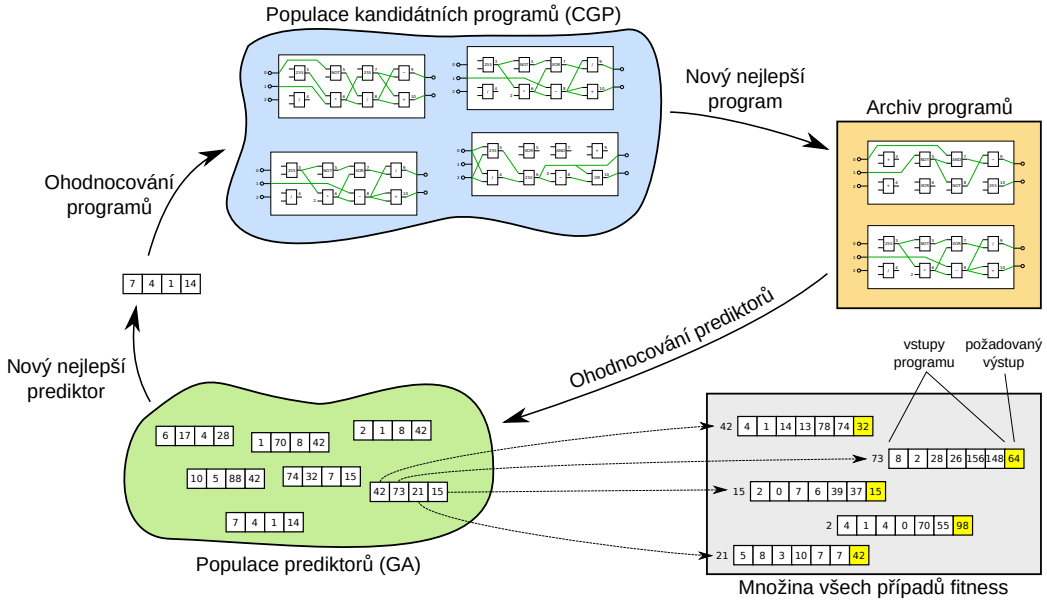
Použitím koevoluce CGP a genetického algoritmu lze řešit problém velkého množství případů fitness tak, že druhá populace vybírá vhodnou podmnožinu případů fitness, která slouží k přibližnému určení fitness kandidátních programů [21].

2.5.1 Koevoluční řešení symbolické regrese

Koevoluční výpočet za účelem snížení výpočetní náročnosti CGP byl poprvé úspěšně ukázán na úloze symbolické regrese [23]. Zde jsou použity dvě populace: populace kartézských programů (kandidátních řešení symbolické regrese) a populace *prediktorů fitness*. Prediktory jsou malou podmnožinou množiny případů fitness. Určují, které případy fitness mají být použity k ohodnocení kandidátních programů. Kromě těchto populací je součástí řešení archiv sdílený oběma populacemi, který obsahuje několik kartézských programů, které slouží pro ohodnocení prediktorů. Schéma populací a interakcí mezi nimi znázorňuje obrázek 2.8.

Evoluce kandidátních programů probíhá pomocí kartézského genetického programování. V případě koevoluce se pro ohodnocení kandidátních řešení používají dvě různé fitness funkce: f_{exact}, ve které se používá celá trénovací množina, a $f_{predicted}$, omezená pouze na některé případy fitness.

Pokud je během evoluce nalezen jedinec s lepší predikovanou fitness (vypočtenou funkcí $f_{predicted}$) než nejlepší jedinec v předchozí generaci, je umístěn do archivu sdíleného s populací prediktorů. Ten je rozdělen na dvě části – první z nich obsahuje nejlepší nalezená řešení, do druhé části jsou pravidelně umisťovány náhodně vygenerované programy, čímž je zajištěna větší diverzita jedinců v archivu. Každý jedinec umístěný do archivu je ohodnocen pomocí celé trénovací množiny (funkcí f_{exact}).



Obrázek 2.8. Schéma koevoluce CGP a prediktorů fitness [23]

Evoluce prediktorů fitness je řízena genetickým algoritmem. Jejich chromozomy jsou vektory ukazatelů do trénovací množiny o konstantní délce. Potomci jsou tvořeni pomocí jednobodového křížení a mutace, navíc je nejhorší jedinec v populaci nahrazen náhodně vygenerovaným chromozomem. Fitness prediktoru je určena jako střední absolutní odchylka skutečné a predikované fitness všech programů v archivu:

$$f(p) = \frac{1}{u} \sum_{i=1}^{u} |f_{exact}(s\ \ (i)) - f_{predicted}(s\ \ (i))| \tag{2.4}$$

kde p označuje prediktor a s archiv, který obsahuje u jedinců. Prediktor s nejnižší odchylkou je pak použit pro ohodnocování řešení z populace kartézských programů.

V článku [23] byl tento koevoluční algoritmus porovnán s běžným CGP na pěti různých úlohách symbolické regrese, přičemž trénovací množina pro každou z nich obsahovala 200 případů fitness. Ukázalo se, že koevolucí lze nalézt přijatelné řešení s použití mnohem menšího počtu evaluací kandidátních programů než u standardního CGP. Také se ukázalo, že zatímco standardní CGP nenalezlo přijatelné řešení v 23,6 % běhů, koevoluční algoritmus byl úspěšný ve všech případech. Co se výpočetní náročnosti týče, byl koevoluční přístup v jednotlivých úlohách symbolické regrese přibližně dvakrát až pětkrát rychlejší.

2.5.2 Koevoluční návrh obrazových filtrů

Podobně jako symbolickou regresi lze pomocí koevoluce akcelerovat i evoluční návrh obrazových filtrů [21]. Opět existují dvě populace, kandidátních filtrů v CGP a podmnožin případů fitness, také je použit sdílený archiv obrazových filtrů. Jednotlivé případy fitness jsou složeny z devítiokolí poškozeného pixelu a hodnoty téhož pixelu v původním obrázku (což je očekávaný výstup filtru).

V článku [21] jsou zmíněny dva různé koevoluční přístupy. V prvním případě šlo o koevoluci s prediktory fitness (CFP, coevolution of fitness predictors), podobně jako u řešení

symbolické regrese popsané výše. Fitness filtrů byla určena pomocí funkce MDPP (viz rovnice 2.3), fitness prediktorů pak jako:

$$f_{CFP}(p) = \frac{1}{T}\sum_{i=1}^{T} |MDPP_{exact}(s\ (i)) - MDPP_{partial}(s\ (i))| \tag{2.5}$$

kde T označuje počet položek v archivu filtrů a $MDPP_{partial}$ je střední odchylka podmnožiny pixelů určených prediktorem, která je dána jako:

$$MDPP_{partial} = \frac{1}{K}\sum_{l}^{K} |v(l) - w(l)| \tag{2.6}$$

kde K označuje počet případů fitness, v filtrovaný a w původní obrázek. Cílem evoluce prediktorů je minimalizace hodnoty fitness.

Ve druhém případě je použita soutěživá koevoluce (CC, competitive coevolution). Podmnožiny případů fitness zde slouží jako jako testy, které reprezentují případy fitness, které kandidátní řešení v aktuálním stavu koevoluce neumí správně řešit. Cílem filtrů je správně vyřešit všechny případy fitness obsažené v testu, cílem populace testů je pak najít takové devítiokolí pixelů, které nalezené filtry nejsou schopny vyřešit správně – tedy nalézt podmnožinu pixelů s maximalní střední odchylkou filtrované a původní hodnoty. Fitness funkci testů lze odvodit z rovnice 2.6 následovně:

$$f_{CC} = \frac{1}{T}\sum_{i=1}^{T}\frac{1}{K}\sum_{l}^{K} |v(l) - w(l)| \tag{2.7}$$

Podobně jako u prediktorů je nejlepší nalezený test použit pro ohodnocení populace filtrů. V obou variantách jsou nejlepší nalezené filtry umisťovány do archivu, který pak slouží k ohodnocování prediktorů či testů.

V porovnání se standardním CGP bylo dosaženo srovnatelné kvality filtrů při použití podmnožiny případů fitness o velikosti pouze 15 % z celkového počtu pixelů v obrázku. V tomto případě byl koevoluční výpočet přibližně třikrát rychlejší než standardní CGP. Oba zmíněné koevoluční přístupy (prediktory fitness i soutěživá koevoluce) vedou na srovnatelně kvalitní filtry [21].

2.5.3 Akcelerace koevolučního návrhu v hardware

Koevoluční CGP bylo také úspěšně implementováno v hardware na rekonfigurovatelném obvodu FPGA. Na úloze návrhu obrazových filtrů, kdy byl měřen čas potřebný na dosažení 10 000 generací CGP, byla hardwarová implementace 18krát až 58krát rychlejší než optimalizovaná[1] softwarová implementace, v závislosti na počtu případů fitness v prediktorech [5].

2.5.4 Návrh obrazových filtrů pomocí kompoziční koevoluce

Jiným přístupem ke koevolučnímu návrhu je použití kompoziční koevoluce. K samotnému filtru je možné přidat detektor šumu, který rozhoduje, zda je právě zpracovávaný pixel poškozený a má být opraven. Cílem je pak nalézt nejlepší kombinaci filtru a detektoru šumu. Experimentálně bylo ověřeno, že koevoluční algoritmus vede na kvalitnější filtry oproti oddělené evoluci filtrů a detektorů šumu bez vzájemné interakce [20].

[1] Pomocí OpenMP a vektorových (SIMD) instrukcí z instrukční sady SSE 4.1.

2.5.5 Otevřené problémy koevoluce

Jak bylo ukázáno, použitím koevoluce lze zkrátit dobu potřebnou pro nalezení přijatelného řešení, v některých případech až pětinásobně. Koevoluční algoritmy dokonce poměrně spolehlivě nachází řešení v případech, kdy standardní CGP selhává. Na druhou stranu je zapotřebí poměrně velké množství běhů k nalezení ideálního nastavení. Kromě velikosti populací nebo počtu kandidátních řešení v archivu jde zejména o délku chromozomu prediktorů fitness. Ukazuje se, že pro různé úlohy je vhodná jiná hodnota. Při suboptimální konfiguraci pak nemusí být dosaženo takového urychlení výpočtu, jako by bylo možné, anebo kvalita nalezených řešení nemusí být přijatelná. Například v případě úlohy symbolické regrese bylo potřeba provést více než sto tisíc nezávislých běhů, než bylo nalezeno nejvhodnější nastavení [23].

Pokud jsou v genetickém algoritmu použity dlouhé chromozomy čítající tisíce genů, vyvstává problém škálovatelnosti. U evolučních algoritmů se pod tímto pojmem rozumí situace, kdy evoluce nenachází přijatelná řešení pro rozsáhlejší úlohy, ačkoliv v menším měřítku funguje dobře [20]. Také použití genetických operátorů na příliš dlouhé chromozomy může být neefektivní.

2.5.6 Nepřímo kódované prediktory fitness

Jedním ze způsobů, jak obejít nutnost najít nejvhodnější délku prediktoru pro konkrétní úlohu, je jejich nepřímé kódování, které evoluci umožní tvořit prediktory různé délky a adaptovat je na konkrétní trénovací data. Prediktory nejsou reprezentovány jako vektor ukazatelů do trénovací množiny, ale jako funkce generující posloupnost ukazatelů, které se mají použít při výpočtu fitness.

Pro hledání vhodné funkce je možné použít kartézské genetické programování [22]. Funkce je reprezentována jako program s jedním vstupem a dvěma výstupy. Součástí chromozomu je oproti standardnímu CGP ještě hodnota x_0 udávající první vstup programu, pomocí které je získán první prvek posloupnosti. Jeden z výstupů slouží jako další prvek posloupnosti (a zároveň jako nový vstup programu), druhý pak udává, zda má být posloupnost ukončena. Fitness generátoru závisí kromě odchylky predikované a skutečné fitness také na délce generované posloupnosti – delší posloupnost vede na větší výpočetní náročnost.

Během experimentů na úloze symbolické regrese se ukázalo, že počet použitých případů fitness odpovídá počtu zjištěnému experimentálně s přímo kódovanými prediktory. Je tedy možné použít koevoluci s prediktory fitness na nové úlohy bez časově náročného hledání optimálního nastavení.

2.6 Vztah učení a evoluce

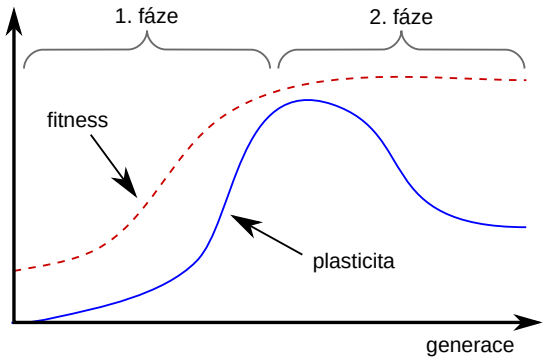
Vztahem mezi učením a evolucí se zabýval už James Mark Baldwin na konci 19. století. Ve svém článku [1] se zabýval vývojem komplexního instinktivního chování. *Baldwinův efekt* popisuje, jakým způsobem lze postupným vývojem po menších krůčcích dosáhnout komplexních instinktů zakódovaných v genotypu. Jedinec, jehož instinkt daný genotypem není dokonalý, jej může učením během života zdokonalit a zvýšit tak svou fitness a pravděpodobnost, že i nedokonalý instinkt se přenese na další generaci. Učení má ovšem i své nevýhody, například v přírodě hrozí, že se během experimentování jedinec zraní nebo zemře, proto jsou upřednostňováni jedinci, kteří mají v genotypu zakódované dokonalejší instinkty, čímž se postupně vytváří komplexní instinkt [25].

2.6.1 Plasticita fitness

Schopnost jedince přizpůsobit se prostředí se označuje jako *plasticita fitness* nebo *plasticita fenotypu*. Fenotyp plastického jedince nezáleží jen na genotypu, ale také na okolním prostředí. Jinými slovy, stejný genotyp může tvořit různé fenotypy. V různých fázích evoluce jsou preferováni jedinci s různou plasticitou. Po radikální změně prostředí jsou ve výhodě plastičtější jedinci, kteří se změně dokáží lépe přizpůsobit, protože mají větší schopnosti učení. Později, když se prostředí opět ustálí, jsou upřednostňování jedinci, kteří mají nově potřebné vlastnosti přímo zakódované do genotypu. Tento průběh znázorňuje obrázek 2.9.

Různou míru plasticity lze pozorovat nejen mezi jedinci v populaci, ale také se mění během života jedince. V průběhu života existují *senzitivní období*, během kterých mají vnější stimuly zvlášť velký význam pro rozvoj jedince – v tomto období má vyšší plasticitu. Například pokud bylo kotěti sešito jedno oko, bylo v dospělosti na něj slepé i po jeho otevření, protože absence vizuálních podnětů v senzitivním období znemožnila jeho zdravý vývoj. Z tohoto plyne, že učící aparát je udržován pouze po určitou dobu, nezbytnou pro osvojení příslušné schopnosti.

Přítomnost nebo nepřítomnost schopnosti učení závisí i na dynamice okolního prostředí. Podle četnosti změn v prostředí lze odlišit několik různých situací. V případě velmi stabilního prostředí schopnost učení není tolik potřebná a vlastnosti jedince mohou být přímo součástí genotypu. V opačném případě, kdy ke změnám dochází velmi často je také učení zbytečné, protože neexistují dlouhodobě platná pravidla, které by se jedinec mohl naučit. Mezi těmito extrémy lze pozorovat různou míru plasticity jedinců. Pokud je četnost změn nižší, spíše se projevuje Baldwinův efekt a naučené chování se v dalších generacích stává součástí genotypu. Se zvyšující se četností změn se objevují senzitivní období na začátku života jedince, od určitého okamžiku pak je schopnost učit se přítomna neustále [2].

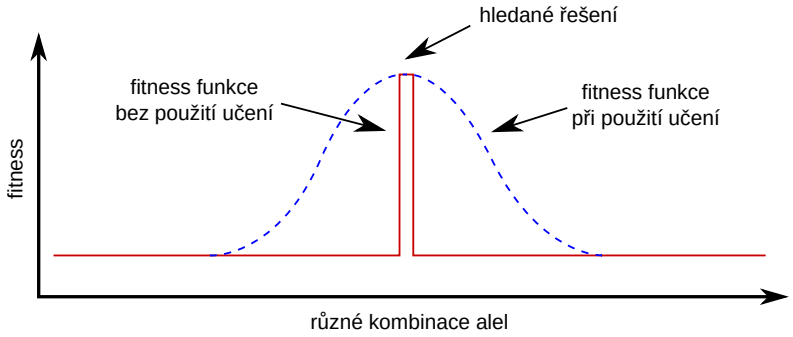


Obrázek 2.9. Baldwinův efekt – nejprve jsou upřednostňování plastičtější jedinci, později převáží cena učení a plasticita klesá [2]

2.6.2 Baldwinův efekt v evolučních algoritmech

První výpočetní model Baldwinova efektu představili Geoffrey Hinton a Steven Nowlan [4]. Pracovali s populací jedinců o 20 genech, které mohly nabývat hodnoty **0**, **1** nebo **?**. Genotyp je interpretován jako nastavení 20 spínačů. Alely **1** a **0** znamenají, že je příslušný spínač zapnut nebo vypnut, pokud gen obsahuje hodnotu **?**, může jedinec se spínačem experimentovat. Cílem bylo sepnout všechny spínače. V případě, že některý gen byl nastaven na

0, měl jedinec minimální fitness ($f = 1$), pokud byly všechny geny nastaveny na **1**, získal jedinec maximální fitness ($f = 20$). Jedinci s „otazníkovými“ alelami měli 1 000 pokusů na nalezení správného řešení a jejich fitness byla úměrná počtu pokusů, které vyčerpali ($f = 1 + 19\frac{1000-i}{1000}$, i je počet využitých pokusů). Po 50 generacích se ukázalo, že jedinci mají v průměru 11 genů nastavených na **1** a 9 genů na **?** a průměrnou fitness $f = 11,6$. „Nulové“ geny byly poměrně rychle eliminovány. Bez užití učení (a „otazníkových“ alel) je očekávaná fitness $f = 1$, protože pravděpodobnost vzniku jedince se všemi geny nastavenými na **1** je velmi malá a fitness funkce nedokáže odlišit, který jedinec je lepší a který horší (každý má buď maximální nebo minimální fitness), jak znázorňuje obrázek 2.10. Přidáním učení se fitness funkce „vyhladí“.



Obrázek 2.10. Fitness funkce z experimentu Hintona a Nowlana [4]

Byly představeny také modifikace kartézského genetického programování využívající plastických jedinců. V jedné z nich chromozom obsahoval oproti standardnímu CGP jeden primární výstup navíc, který nebyl přímou součástí kandidátního řešení. U nově vzniklých potomků je nejprve vytvořen fenotyp, který je posléze modifikován, dokud tento výstup není roven předem stanovené hodnotě. Fitness jedince je pak určena až pomocí modifikovaného fenotypu. V experimentech na klasifikačních úlohách se ukázalo, že oproti standardnímu CGP lze dosáhnout menší chybovosti [26].

V jiné variantě plastického kartézského genetického programování byly z chromozomu odstraněny geny kódující propojení primárních výstupů na funkční bloky. Nově vzniklý jedinec pak pro každý primární výstup hledá nejvhodnější funkční blok, tak aby dosáhl co nejvyšší fitness. Na úloze návrhu úplné sčítačky se ukázalo, že algoritmus končí úspěšně bez ohledu na velikost kartézské mřížky, zatímco u standardního CGP úspěšnost s větší mřížkou klesala [10].

Kapitola 3

Návrh obrazových filtrů pomocí souběžného učení

Cílem této práce je navrhnout systém pro tvorbu obrazových filtrů založený na principech evolučních algoritmů, koevoluce a souběžného učení a tento systém implementovat a experimentálně vyhodnotit.

Jedním z nedostatků koevoluce s prediktory fitness, popsané v podkapitole 2.5.2, je nutnost zvolit vhodnou velikost podmnožiny případů fitness, kterou vybírají prediktory. Tato velikost by měla být co nejmenší, aby byl počet nutných vyhodnocení co nejnižší (a výpočet co nejrychlejší), ale zároveň taková, aby odchylka mezi predikovanou a skutečnou fitness kandidátních řešení byla stále přijatelně nízká. Velikost vhodná pro jednu úlohu nemusí být vhodná pro jinou. Například v případě symbolické regrese mohou u jednoduchých rovnic stačit pouze jednotky trénovacích vektorů, u obrazových filtrů to mohou být i tisíce. Někdy je pro stanovení optimální délky nutné provést velké množství experimentů s různým nastavením; mohou to být i tisíce nezávislých běhů.

Další slabinou koevolučního návrhu je i problém škálovatelnosti – u složitějších úloh, jako je návrh obrazových filtrů, může prediktor čítat tisíce genů. Práce s takto velkými genotypy je pak poměrně neefektivní.

Tyto nedostatky lze obejít použitím jiného kódování prediktorů, ve kterém prediktor neobsahuje přímo ukazatele do množiny všech případů fitness, ale pouze návod, podle kterého se mají případy fitness vybírat. V části 2.5.6 je popsáno kódování prediktorů jako programy tvořené dle principů kartézského genetického programování, které generují posloupnost ukazatelů do množiny případů fitness, která může nabývat různé délky.

Tato práce se však zabývá novým typem přímého kódování prediktorů, jehož cílem je zmírnit současné nedostatky koevoluce pomocí souběžného učení. Pokud umožníme tvorbu různě velkých prediktorů z jednoho genotypu, mohou jedinci získanou plasticitu využít pro adaptaci na složitost právě řešené úlohy i na aktuální průběh evoluce.

Tato kapitola se nejprve v části 3.1 zabývá návrhem kódování reprezentace obrazových filtrů, následující část 3.2 pak samotným průběhem evoluce bez použití koevoluce. Kapitola 3.3 popisuje koevoluci CGP a prediktorů fitness. Poslední část 3.4 uvádí návrh nového kódování prediktorů fitness s plastickým fenotypem a možnostmi, jak lze získanou plasticitu využít pro adaptaci prediktorů na průběh evoluce.

3.1 Kandidátní obrazové filtry

Obrazové filtry jsou vyvíjeny pomocí kartézského genetického programování, popsaného v kapitole 2.3. Chromozom tvoří vektor celých čísel kódující acyklický orientovaný graf. Jednotlivé funkční bloky jsou tvořeny trojicí genů – první z nich označuje prováděnou funkci, druhý a třetí gen pak určují, kam jsou připojeny jeho vstupy. Seznam funkcí, které mohou výpočetní uzly realizovat je v tabulce 3.1. Programy mají devět primárních vstupů a jeden primární výstup. Na vstupy jsou přiváděny hodnoty pixelů z devítiokolí právě zpracovávaného pixelu, výstup udává novou hodnotu pixelu, jak je popsáno v kapitole 2.4. V devítiokolích pixelů na okraji obrázku se na pozicích mimo obrázek používá hodnota nejbližšího okrajového pixelu.

Tabulka 3.1. Seznam funkcí výpočetních uzlů. Převzato z článku [21]

#	funkce	popis	#	funkce	popis
0	255	konstanta	8	$i_1 \gg 1$	posun vpravo o 1 bit
1	i_1	identita	9	$i_1 \gg 2$	posun vpravo o 2 bity
2	$255 - i_1$	inverze	10	$\text{swap}(i_1, i_2)$	prohození 4 bitů
3	$i_1 \vee i_2$	OR	11	$i_1 + i_2$	sčítání
4	$\neg i_1 \vee i_2$	OR s negovaným vstupem	12	$i_1 +^S i_2$	saturované sčítání
5	$i_1 \wedge i_2$	AND	13	$(i_1 + i_2) \gg 1$	průměr
6	$\neg(i_1 \wedge i_2)$	NAND	14	$\max(i_1, i_2)$	maximum
7	$i_1 \oplus i_2$	XOR	15	$\min(i_1, i_2)$	minimum

Protože součástí fenotypu nejsou všechny výpočetní uzly, je u každého funkčního bloku v chromozomu uložena i informace, zda je aktivní. Pomocí těchto údajů lze urychlit výpočet fitness, protože není nutné počítat výstup neaktivních bloků. V případě, že jedinec vznikl pouze mutací neaktivních uzlů, není třeba fitness počítat vůbec. Jako fitness funkce je použita špičková hodnota poměru signál/šum (Peak Signal to Noise Ratio):

$$f\left(cgp\right) = 10 \log_{10} \frac{255^2}{\frac{1}{N} \sum_{i}^{N} \left(v\left(i\right) - w\left(i\right)\right)^2} \tag{3.1}$$

kde N označuje počet použitých případů fitness, $v(i)$ a $w(i)$ pixel filtrovaného (resp. původního) obrázku označený v množině případů fitness indexem i.

3.2 Evoluce obrazových filtrů bez koevoluce

Vstupem algoritmu jsou dva obrázky: jeden slouží jako vstup filtru (např. šumem poškozený obrázek), druhý jako referenční výstup (např. nepoškozený obrázek). Každé devítiokolí pixelů vstupního obrázku a odpovídající pixel referenčního obrázku tvoří jeden případ fitness.

Na začátku je náhodně vytvořena počáteční populace a vypočtena fitness všech jedinců. Jedinec s nejlepší fitness se stává rodičem následující generace. Ostatní jedinci jsou nahrazeni jeho kopiemi a několik jejich genů je pozměněno mutací – používá se tedy evoluční strategie $(1 + \lambda)$. Mutace dodržuje pravidla zmíněná v části 2.3.1 s tím rozdílem, že není

povoleno na primární výstup připojit primární vstup. Takto vytvořená populace je opět ohodnocena fitness funkcí a algoritmus pokračuje výběrem rodiče a tvorbou další generace.

Evoluce je ukončena po dosažení předem specifikovaného počtu generací. Volitelně je možné i určit cílovou hodnotu PSNR, po jejímž dosažení bude výpočet ukončen.

3.3 Koevoluce obrazových filtrů a prediktorů fitness

Koevoluce obrazových filtrů a prediktorů pracuje dle principů popsaných v části 2.5.2. Součástí algoritmu je kromě populace obrazových filtrů také populace prediktorů fitness, která se vyvíjí pomocí genetického algoritmu (viz část 2.2). Obě populace interagují prostřednictvím dvou archivů, které obsahují jednak kandidátní filtry a jednak prediktor, který se používá pro ohodnocení filtrů.

3.3.1 Fitness kandidátních filtrů

U populace obrazových filtrů jsou použity dvě různé fitness funkce. *Skutečná fitness* f_{exact} je určena pomocí celé množiny případů fitness a slouží pro výpočet fitness prediktorů a také pro posuzování průběhu evoluce při modifikaci proměnné *UsedGenes*. Oproti tomu *predikovaná fitness* $f_{predicted}$ používá při výpočtu pouze případy fitness určené prediktorem. Používá se pro ohodnocení jak obrazových filtrů, tak prediktorů. V obou případech je jako fitness použita hodnota PSNR dle rovnice 3.1.

3.3.2 Prediktory fitness pevné délky

Chromozom prediktorů je reprezentován jako celočíselný vektor, kdy každý gen je ukazatelem do množiny případů fitness o konstantní délce, přičemž žádné dva geny nemají stejnou hodnotu. Cílem evoluce je nalézt takové prediktory, u kterých je odchylka mezi skutečnou fitness f_{exact} a predikovanou fitness $f_{predicted}$ u všech kandidátních programů v archivu co nejmenší. Fitness funkci lze zapsat následovně:

$$f(p) = \frac{1}{u} \sum_{i=1}^{u} |f_{exact}(s(i)) - f_{predicted}(s(i))| \tag{3.2}$$

kde $f(p)$ je fitness prediktoru, u počet filtrů uložených v archivu s, které se používají k ohodnocení prediktoru, f_{exact} označuje skutečnou fitness filtru $s(i)$ a $f_{predicted}$ predikovanou fitness filtru $s(i)$.

3.3.3 Archivy

Archiv kandidátních filtrů je reprezentován polem filtrů o pevné délce. Pokud je zaplněn, jsou nejstarší položky přepisovány novými (kruhový buffer). Každý záznam obsahuje chromozom kandidátního filtru a jeho skutečnou fitness. Filtry obsažené v archivu slouží k výpočtu fitness prediktorů dle rovnice 3.2.

Archiv prediktorů je tvořen jedinou položkou – nejlepším nalezeným prediktorem. Ten slouží k výpočtu predikované fitness při evoluci obrazových filtrů.

3.3.4 Průběh evoluce

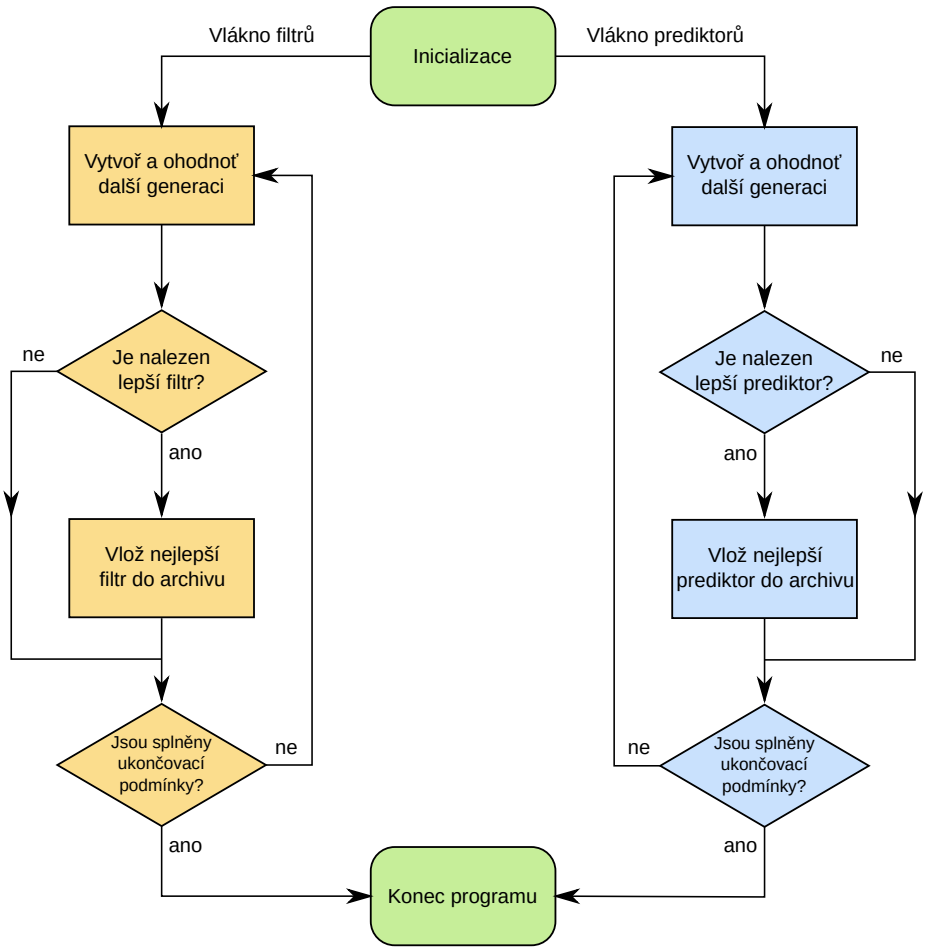
Na začátku běhu programu jsou náhodně vytvořeny počáteční populace filtrů a prediktorů fitness. Poté je vypočtena skutečná fitness filtrů a nejlepší jedinec je umístěn do archivu.

Pomocí něj jsou ohodnoceny prediktory fitness a je určen nejlepší ten, který bude zpočátku sloužit pro výpočet predikované fitness. Po této inicializaci archivů pokračuje běh programu ve dvou oddělených vláknech, kde každé obsluhuje jednu populaci. Běh programu znázorňuje diagram na obrázku 3.1.

Evoluce kandidátních filtrů probíhá stejně jako v případě bez koevoluce (viz část 3.2). Rozdíl je v tom, že pokud je predikovaná fitness nejlepšího potomka vyšší než fitness jeho rodiče (a stává se tak novým rodičem), je umístěn do sdíleného archivu.

V populaci prediktorů fitness je nová populace tvořena třemi druhy potomků. Jednu čtvrtinu celkového počtu jedinců tvoří nejlepší jedinci (elitismus). Druhá čtvrtina potomků je vygenerována zcela náhodně, aby se zachovala diverzita populace. Zbývající dvě čtvrtiny potomků jsou tvořeni pomocí jednobodového křížení a mutace, která nahrazuje malé množství genů náhodnými hodnotami. Jejich rodiče jsou voleni pomocí turnaje, ve kterém se každého kola účastní dva náhodně zvolení jedinci. V případě, že má některý potomek vyšší fitness než má prediktor v archivu, nahradí jej.

Evoluce je stejně jako v případě bez koevoluce ukončena po dosažení stanoveného počtu generací nebo požadované hodnoty PSNR.



Obrázek 3.1. Vývojový diagram koevoluce obrazových filtrů a prediktorů fitness

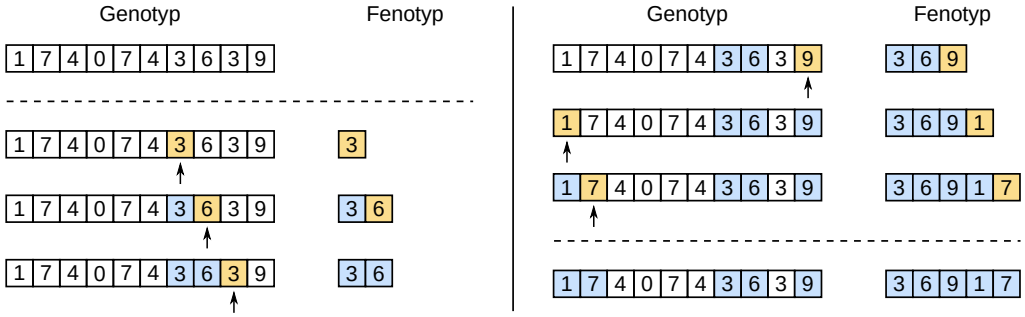
3.4 Souběžné učení v koevoluci filtrů a prediktorů fitness

V případě souběžného učení v koevolučním CGP se používá mírně odlišné kódování prediktorů, které se vyznačuje plastickým fenotypem, což znamená, že lze z jednoho genotypu odvodit různé fenotypy. V této části je popsáno, jak toto kódování vypadá a jakým způsobem lze získanou plasticitu využít pro adaptaci prediktorů na složitost řešeného problému v průběhu evoluce.

3.4.1 Plastické prediktory fitness

Chromozom je stejně jako v případě prediktorů pevné délky reprezentován jako vektor číselných ukazatelů do množiny případů fitness. Rozdíl je v počtu genů, který u plastického kódování odpovídá celkovému počtu případů fitness. Ačkoliv ani zde není žádoucí, aby prediktor obsahoval některé ukazatele vícekrát, jsou v genotypu povoleny duplicitní hodnoty. Zamezit by jim šlo použitím například permutačního kódování, ale to je pro chromozomy o tisících genech, jenž se dají u návrhu obrazových filtrů očekávat, už poměrně neefektivní.

Plasticita je dosažena tím, že fenotyp netvoří všechny ukazatele obsažené v genotypu, ale pouze jejich část. Fenotyp je z genotypu tvořen postupným čtením genů od zvolené pozice (*offsetu*). Pokud hodnota právě čteného genu není obsažena ve fenotypu, je do něj vložena, v opačném případě se gen ignoruje. Poté je přečten další gen a proces se opakuje. Během tvorby fenotypu nejsou čteny úplně všechny geny – jejich počet je určen hodnotou *UsedGenes*, která je součástí prostředí. Pokud je při čtení dosaženo konce genotypu, ale nebyl ještě přečten potřebný počet genů, pokračuje se od jeho začátku. Příklad tvorby fenotypu při použití šesti genů z deseti ukazuje obrázek 3.2.



Obrázek 3.2. Postup konstrukce fenotypu prediktoru. Šipka označuje právě přečtený gen

S plasticitou jedinců lze pracovat různými způsoby. Především lze velikost prediktorů adaptovat na průběh evoluce kandidátních filtrů. Další možností je simulovat u prediktoru učení a adaptaci na prostředí tak, že nově vytvořený prediktor se snaží najít nejvýhodnější offset několika náhodnými pokusy. Také lze v situaci, kdy evoluce filtrů stagnuje, skokově změnit offset všech prediktorů v populaci, což může pomoci v posunu z lokálního optima. Všechny zmíněné možnosti lze kombinovat.

3.4.2 Adaptace velikosti prediktoru na průběh evoluce

Adaptace velikosti prediktoru probíhá na základě sledování rychlosti vývoje skutečné fitness kandidátních řešení. Podobně, jako lze pozorovat senzitivní období v životě jedince (viz kapitola 2.6.1), lze očekávat, že i populace prochází obdobími, kdy je celková schopnost

adaptace vyšší (a fitness populace stoupá) a obdobími, kdy se naopak adaptovat příliš nedokáže (a fitness se nemění). Důležitý je i směr změny fitness, protože je pro ohodnocení jedinců v CGP používána predikovaná fitness a skutečná fitness nejlepšího jedince tak může i klesat. Rychlost evoluce je možné například určit jako podíl změny fitness nejlepšího jedince v populaci a počtu generací, které uběhly od minulé změny:

$$v = \frac{\Delta f_{exact}}{\Delta G} \tag{3.3}$$

Lze však pracovat i s delším úsekem historie, například jako celkovou rychlost v použít medián či (vážený) průměr rychlostí za několik posledních změn. Také lze ručně připravit trénovací data sestávající z několika na sebe navazujících změn a odvodit z nich vzorec pro výpočet celkové rychlost evoluce, například pomocí symbolické regrese.

U příliš krátkých prediktorů (v extrémním případě to může být i jen jeden případ fitness) hrozí, že se velmi dobře adaptují na kandidátní řešení uložená v archivu a predikce u jiných filtrů bude velmi nepřesná. Proto se sleduje i nepřesnost predikce a pokud překročí stanovenou mez $I_{threshold}$, prediktory se prodlouží. Nepřesnost prediktoru lze určit jako poměr mezi predikovanou a skutečnou hodnotou fitness nejlepšího filtru v populaci:

$$I = \frac{f_{predicted}}{f_{exact}} \tag{3.4}$$

Velikost prediktoru se upravuje vždy při změně rodiče v populaci filtrů, což v CGP znamená, že některý z potomků má vyšší nebo stejnou predikovanou fitness jako jeho rodič. Také lze volitelně nastavit, že změna velikosti musí nastat přinejhorším po stanoveném počtu generací CGP.

Vždy, když má dojít k úpravě velikosti prediktoru, je zvolen koeficient, pomocí kterého se upraví současná hodnota proměnné *UsedGenes*, čímž se změní velikost fenotypů prediktorů. Na výběr jsou dva způsoby. U prvního z nich se nová hodnota *UsedGenes* získá prostým vynásobením současné hodnoty koeficientem. Ve druhém případě je koeficientem vynásoben celkový počet případů fitness a výsledná hodnota je přičtena k současné hodnotě *UsedGenes*. Hodnoty koeficientů jsou určeny experimentálně, nicméně základní předpoklady jsou:

Koeficient c_I: Pokud je nepřesnost predikce příliš vysoká, prediktory se prodlouží.

Koeficient c_0: Pokud se fitness nemění ($v \approx 0$), evoluce pravděpodobně uvázla v lokálním optimu, prediktory se zkrátí, čímž mohou pomoci se z tohoto optima posunout dále.

Koeficient c_{-1}: Pokud fitness klesá ($v < 0$), evoluce pravděpodobně opouští lokální optimum a mírné zkrácení prediktoru může pomoci postup urychlit.

Koeficienty c_{+1} a c_{+2}: Pokud fitness roste ($v > 0$), je vhodně prediktory prodloužit, čímž se zpřesní predikce. Rozlišuje se mezi *pomalým* a *rychlým* růstem.

Hraniční rychlost, při které je rychlost evoluce ještě považována za nulovou určuje parametr v_{zero}. V případě kladné rychlosti (rostoucí fitness) se mezi *pomalým* a *rychlým* růstem rozhoduje podle hodnoty parametru v_{slow}.

Všechna pravidla použitá v algoritmu shrnuje tabulka 3.2. Pokud je splněna podmínka některého z pravidel, další pravidla v pořadí se již nevyhodnocují.

Tabulka 3.2. Zvolená pravidla pro změnu velikosti prediktoru

(a) Relativně k současné velikosti prediktoru

#	podmínka	nová hodnota *UsedGenes*	popis
1.	$I > I_{threshold}$	$UsedGenes \cdot c_I$	nepřesná predikce
2.	$\|v\| \leq v_{zero}$	$UsedGenes \cdot c_0$	fitness se nemění
3.	$v < 0$	$UsedGenes \cdot c_{-1}$	fitness klesá
4.	$0 < v \ \leq v_{slow}$	$UsedGenes \cdot c_{+1}$	fitness pomalu stoupá
5.	$v > v_{slow}$	$UsedGenes \cdot c_{+2}$	fitness rychle stoupá

(b) Podle celkového počtu případů fitness N

#	podmínka	nová hodnota *UsedGenes*	popis
1.	$I > I_{threshold}$	$UsedGenes \cdot c_I$	nepřesná predikce
2.	$\|v\| \leq v_{zero}$	$UsedGenes + (N \cdot c_0)$	fitness se nemění
3.	$v < 0$	$UsedGenes + (N \cdot c_{-1})$	fitness klesá
4.	$0 < v \ \leq v_{slow}$	$UsedGenes + (N \cdot c_{+1})$	fitness pomalu stoupá
5.	$v > v_{slow}$	$UsedGenes + (N \cdot c_{+2})$	fitness rychle stoupá

3.4.3 Paměť průběhu evoluce

Aby bylo možné vypočítat rychlost evoluce kandidátních filtrů, je potřeba udržovat historii fitness rodičů delší, než jen mezi aktuální a novou generací. Protože není potřeba udržovat celou historii, ale zajímá nás jen poslední vývoj, jsou nejstarší položky přepisovány novějšími – z implementačního pohledu jde o kruhový buffer. Ukládá se zejména:

- Δf_{exact} – rozdíl skutečné fitness filtrů oproti poslednímu záznamu,
- ΔG – počet uběhlých generací,
- $\frac{\Delta f_{exact}}{\Delta G}$ – „rychlost“ evoluce (průměrná změna fitness na jednu generaci).

Nová položka je do paměti vložena vždy, když je predikovaná fitness nejlepšího potomka vyšší, než byla u jeho rodiče. Volitelně mohou být záznamy tvořeny v pravidelných intervalech, například každých tisíc generací. V tomto případě je možné, že fitness bude delší dobu stejná a kruhový buffer již nebude obsahovat informaci o směru poslední změny. Proto paměť obsahuje zvláštní pozici pro poslední záznam s nenulovým rozdílem fitness, která není součástí kruhového bufferu.

3.4.4 Průběh evoluce

V případě souběžného učení probíhá evoluce stejně, jako u koevolučního CGP. Hlavní rozdíl je v prováděných akcích po nalezení filtru s lepší predikovanou fitness, než měl jeho rodič. V takovém případě je navíc do paměti průběhu evoluce vložen nový záznam a upraví se hodnota *UsedGenes*, čímž se změní velikost fenotypu prediktorů. K úpravě hodnoty *UsedGenes* může dojít také po stanoveném počtu generací CGP, během kterých nebyl nalezen lepší obrazový filtr.

3.5 Paralelizace programu

V navrženém algoritmu lze některé činnosti provádět paralelně. Jde například o souběžnou evoluci populace filtrů a populace prediktorů, výpočet fitness jedinců a tvorbu potomků (filtrů i prediktorů), jednotlivá kola turnajové selekce rodičů v populaci prediktorů nebo výpočet výstupů kandidátního filtru pro různé případy fitness. Také výpočet skutečné fitness programů uložených do archivu může probíhat v samostatném vlákně.

V případě koevoluce (a souběžného učení) se program po inicializaci rozdělí do dvou vláken, v prvním z nich probíhá evoluce obrazových filtrů, ve druhém evoluce prediktorů fitness. Jejich běh není synchronizován, evoluce na sebe „nečekají“. Vlákna mezi sebou komunikují prostřednictvím sdílené paměti, ve které jsou uloženy především archivy s kandidátními programy a s nejlepším nalezeným prediktorem. Pokud se používá souběžné učení, obsahuje sdílená paměť navíc historii změn fitness nejlepšího programu (viz část 3.4.3).

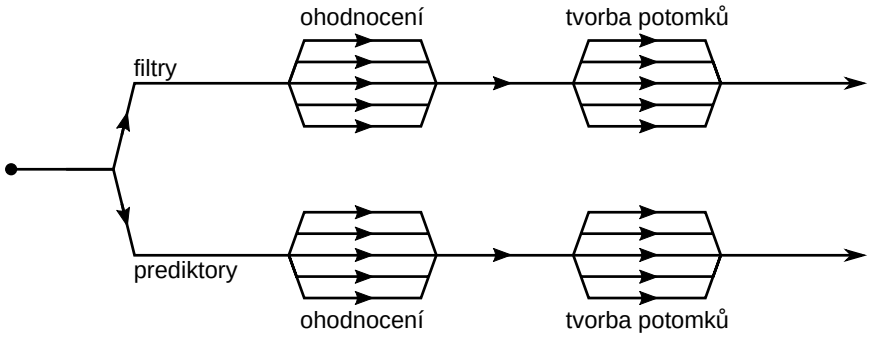
Přístup ke sdíleným proměnným musí být ošetřen vhodným synchronizačním mechanismem, aby nedocházelo k nežádoucím chybám (*race-conditions*). V případě, že je do archivů kopírován nový kandidátní filtr nebo prediktor, nemůže druhé vlákno ohodnocovat potomky. Při použití souběžného učení navíc nelze během změny velikosti prediktorů a následném přepočtu jejich fitness s novým fenotypem ohodnocovat obrazové filtry ani vkládat nové kandidátní filtry do archivu.

Koevoluci lze implementovat také pseudoparalelně, kdy se obě populace periodicky střídají – po uplynutí stanoveného počtu generací populace filtrů se provede několik generací populace prediktorů. Sice odpadá nutnost ošetřování přístupu ke sdíleným proměnným, program by ale byl pomalejší.

Rozhodl jsem se výpočet skutečné fitness nevyčleňovat do samostatného vlákna, probíhá tedy v rámci vlákna zpracovávající evoluci filtrů ihned po uložení programu do archivu.

Kromě těchto dvou vláken se běh programu dělí na další vlákna během ohodnocování jedinců a tvorbě potomků. Schéma paralelizace při použití koevoluce je na obrázku 3.3. U standardního CGP není potřeba na začátku běhu programu vytvářet vlákna pro evoluci populací. Výpočet fitness a tvorba potomků probíhá paralelně stejně jako u koevoluce.

Ukončovací podmínky se kontrolují ve vlákně evoluce filtrů. Druhému vláknu je konec běhu signalizován prostřednictvím sdílené proměnné `finished`.



Obrázek 3.3. Rozdělení běhu programu na vlákna při použití koevoluce. U standardního CGP se program na začátku nedělí do dvou paralelních sekcí

Kapitola 4

Implementace navrženého algoritmu

Kvůli velkému množství plánovaných experimentů byl kladen důraz především na co nejrychlejší běh programu. Implementován byl proto v jazyce C, ve kterém lze snadno pracovat i s assemblerem a rozšiřujícími instrukčními sadami procesoru. V implementaci se využívá některých vlastností normy C11 (například klíčová slova pro zarovnání dat v paměti), ale většina využitých vlastností je definována i ve standardu POSIX, i když někdy trochu jiným způsobem. Při překladu se pak pomocí maker preprocesoru vybere dostupná implementace.

Tato kapitola se nejprve v části 4.1 zabývá paralelizací evoluce na úrovni populací a jedinců, která byla zmíněna i v části 3.5. Část 4.2 popisuje optimalizací výpočtu výstupů filtrů pomocí vektorových (SIMD) instrukcí a úpravy, které jsou potřeba pro efektivní vektorizaci. Následující část 4.3 se zabývá práci s obrázky a poslední část 4.4 programovou podporou pro experimentální vyhodnocení.

4.1 Paralelizace evoluce

Navržený program lze v některých částech výpočetu paralelizovat, jak je uvedeno v části 3.5. Na výběr je několik možností, jak paralelní zpracování implementovat. Lze využít prostředků operačního systému a běh programu rozdělit na vlákna nebo podprocesy, ale je to poměrně pracné. Existují knihovny, pomocí kterých lze pracovat na vyšší úrovni abstrakce. Zejména jde o standardy *Message Passing Interface* (MPI)[1] a *OpenMP*[2]. Zatímco MPI je založeno na zasílání zpráv mezi výpočetními uzly (např. v počítačovém clusteru), OpenMP je zaměřeno na paralelizaci se sdílenou pamětí. Lze je i mezi sebou kombinovat, například MPI může sloužit pro rozdělení výpočtu mezi jednotlivé uzly a OpenMP pro paralelizaci v rámci jednoho uzlu [16].

4.1.1 Implementace pomocí OpenMP

Standard OpenMP definuje speciální direktivy překladače pro označení paralelních částí programu v jazyce FORTRAN, C nebo C++. Také poskytuje několik knihovních funkcí a proměnných prostředí, pomocí kterých lze zjišťovat nebo upravovat nastavení běhu, například maximální počet používaných vláken. Při použití OpenMP se programátor vůbec

[1] Implementovaný například v knihovně OpenMPI (http://www.open-mpi.org/).
[2] http://openmp.org/

nemusí zabývat samotnou implementací paralelizace na nejnižší úrovni. Také se nemusí zabývat případnými rozdíly v implementaci vláken na různých platformách. Veškerý nízkoúrovňový kód vygeneruje překladač. Také lze ze stejného zdrojového kódu vytvořit sekvenční verzi programu prostým ignorováním direktiv OpenMP [16].

Při spuštění programu lze parametry příkazové řádky určit, zda se má použít koevoluce nebo pouze standardní CGP. V prvním případě je po inicializaci program rozdělen do dvou paralelních sekcí (direktivou `#pragma omp parallel sections`). V každé z nich se pak vyvíjí jedna populace. Přístupy ke sdílené paměti, která obsahuje archivy s kandidátními programy a s nejlepším nalezeným prediktorem, jsou uzavřeny do kritických sekcí (`#pragma omp critical`). Jinak není běh obou populací nijak synchronizován. Rozdělení jedinců mezi vlákna při výpočtu fitness u koevolučního i standardního CGP zajišťuje direktiva `#pragma omp parallel for`. Po určení rodičů je stejným způsobem paralelizována i tvorba potomků a nové populace.

V případě souběžného učení se nová délka prediktoru počítá ve vlákně populace CGP, ačkoliv samotná změna velikosti se provádí ve vlákně prediktorů (ihned po vytvoření nové generace prediktorů). Je to z toho důvodu, že během jedné generace prediktorů může proběhnout i několik generací CGP, a v případě, že by se nová velikost počítala ve vlákně prediktorů, nebyly by zahrnuty některé změny nejlepší fitness filtrů.

V případě, že na cílové platformě není dostupný překladač s podporou OpenMP, nelze spustit koevoluci. Toto by bylo možné obejít pseudoparalelní implementací, touto variantou jsem se ale nezabýval. Program tak lze používat jen se standardním CGP, ale ohodnocování jedinců a tvorba potomků bude probíhat sekvenčně.

4.2 Vektorizace výpočtu CGP

Největší důraz byl kladen na optimalizaci rychlosti výpočtu výstupních hodnot kartézských programů, což je časově nejnáročnější část výpočtu. Jako výhodné se ukázalo použití instrukcí typu SIMD (Single Instruction, Multiple Data), pomocí kterých lze počítat s vektory několika desetinných či celých čísel najednou.

4.2.1 Instrukční sada Streaming SIMD Extensions (SSE)

V procesorech Intel Pentium III se v roce 1999 poprvé objevila rozšiřující instrukční sada *Streaming SIMD Extensions* (SSE). Navazuje na starší instrukční sadu *MultiMedia Extensions* (MMX), ve které bylo možné pracovat s celočíselnými vektory o celkové velikosti 64 bitů. K tomu se ale využívaly registry matematického koprocesoru (FPU), nebylo tak možné prolínat vektorové instrukce s instrukcemi pro FPU [11].

Při použití první verze SSE měl programátor k dispozici osm 128bitových vyhrazených registrů a instrukce pro výpočty s vektory čtyř 32bitových čísel s plovoucí desetinnou čárkou. Nejde tedy přísně vzato o nástupce MMX. Za toho lze považovat až následující verzi SSE2, která přinesla instrukce pro vektory dvou 64bitových desetinných čísel o dvojnásobné přesnosti a pro vektory celých čísel o šířce od osmi do 32 bitů. Později bylo SSE dále rozšiřováno, poslední verze je SSE4.2. Časem se také zvýšil počet registrů vyhrazených pro SSE z osmi na šestnáct [7, 24].

4.2.2 Instrukční sada Advanced Vector Extensions (AVX)

Instrukční sada *Advanced Vector Extensions* (AVX) je zatím nejnovější rozšíření pro výpočty s číselnými vektory. AVX byla poprvé obsažena v procesorech Intel architektury Sandy Bridge uvedených na trh v roce 2011. Nově jsou k dispozici 256bitové registry a odpovídající instrukce pro vektorové operace. Dle požadované přesnosti tak lze pracovat s vektorem šestnácti nebo osmi čísel s plovoucí desetinnou čárkou. Mezi novými instrukcemi ale chybí podpora pro 256bitové vektory celých čísel.

Později bylo uvedeno rozšíření nazvané AVX2, jehož podpora se poprvé objevila v procesorech Intel architektury Haswell v roce 2013. Zde už je podpora nových 256bitových registrů rozšířena na všechny instrukce původně pocházející z instrukčních sad SSE, včetně instrukcí pro výpočty nad vektory celých čísel. V případě osmibitových celých čísel tak lze pracovat s 32 hodnotami najednou [6].

Intel již představil novou generaci AVX, instrukční sadu AVX-512, ve které se zdvojnásobí počet registrů AVX na 32 a zároveň se jejich velikost zvětší na 512 bitů. První procesory s tímto rozšířením by měly být dostupné v roce 2015 [17].

4.2.3 Zarovnání dat v paměti

Při kopírování dat z paměti do registrů SSE či AVX je vhodné mít data v paměti zarovnaná na násobek 16, resp. 32 bajtů. Sice existují i instrukce pro přesun nezarovnaných dat, jejich použití ale může vést k nižší rychlosti programu. Standard C11 pro tyto účely zavádí klíčové slovo `alignas(N)` pro deklarace proměnných a datových typů a funkci `aligned_alloc` pro dynamickou alokaci paměti. Pro starší verze překladačů lze využít funkci `posix_memalign` definovanou v normě POSIX a pro deklarace proměnných a datových typů použít klíčová slova, která podporuje použitý překladač, například pro GCC to je zápis `__attribute__ ((aligned (N)))`.

4.2.4 Výpočet CGP pomocí SIMD instrukcí

Výpočet výstupů CGP byl implementován sériově i vektorově pomocí instrukcí SSE2 a AVX2. Starší a v současnosti více rozšířenou instrukční sadu AVX bohužel nelze použít, protože neobsahuje instrukce pro práci s 8bitovými celými čísly. V základní sériové implementaci se každý výstupní pixel počítá zvlášť a pro zpracování celého obrázku o rozměrech 256×256 pixelů je třeba 65 536 průchodů filtrem. U vektorové implementace pomocí SSE2 lze spočítat výstup pro 16 pixelů zároveň a v případě AVX2 dokonce 32 pixelů. Pro zpracování zmíněného obrázku s použitím instrukcí AVX2 pak stačí jen 2 048 průchodů filtrem.

Vektorová implementace využívá tzv. *intrinsic* funkcí, jejichž volání překladač přeloží jako jednu instrukci (například volání intrinsic funkce `_mm_srli_epi16` se přeloží jako instrukce `PSRLW`). Protože v plánovaných experimentech měla mřížka CGP vždy 8 sloupců a 4 řádky, bylo možné kód ručně optimalizovat tak, aby se co nejlépe využívalo registrů SSE (resp. AVX) a přístupů do paměti bylo co nejméně. Počítá se po sloupcích, přičemž 4 registry vždy obsahují výstupní hodnoty sloupců předchozího sloupce a do dalších 4 registrů se ukládají hodnoty právě používaného sloupce. Další registry slouží pro různé pomocné hodnoty. K tomu bylo v deklaracích příslušných proměnných použito klíčové slovo `register`.

Přeložený program obsahuje všechny tři zmíněné implementace výpočtu CGP. Která implementace se použije při výpočtu, závisí na tom, jaké instrukční sady podporuje počítač a operační systém, na kterém je program spuštěn. To se zjišťuje pomocí instrukce `CPUID`. Přednost má implementace s instrukcemi AVX2, kdy je naměřené zrychlení oproti sekvenční

implementaci u standardního CGP přibližně šestnáctinásobné. Další v pořadí je implementace pomocí SSE2, u níž je zrychlení zhruba desetinásobné. Pokud ani tuto instrukční sadu procesor nebo operační systém nepodporuje, použije se základní sekvenční implementace.

4.2.5 Další možnosti akcelerace

V budoucnu je možné do progamu přidat další efektivnější implementace výpočtu CGP. Nabízí se například využití zmíněné připravované instrukční sady AVX-512, pomocí které by bylo možné zpracovávat 64 pixelů zároveň.

Výpočet by bylo možné dále urychlit překladem kandidátních programů do nativního kódu. Při výpočtu se pro každý případ fitness a pro každý funkční blok v kartézském programu opakovaně prochází blokem `switch`, jehož každá větev odpovídá jedné funkci, kterou může blok vykonávat. Toto vede na poměrně velké množství provedených skoků. Program přeložený do nativního kódu tyto skoky neobsahuje. Překlad lze provést pomocí externího překladače, ale ukázalo se, že je efektivnější použít jednoduchý překladač zabudovaný do vlastního programu [28].

4.3 Načítání a ukládání obrázků

Protože jazyk C ani standardní knihovny neposkytují podporu pro práci s obrazovými soubory, je třeba použít buď vlastní implementaci nebo sáhnout po nějaké volně dostupné knihovně. Pro načítání vstupních obrázků jsem se rozhodl použít knihovnu `stb_image.h` od Seana T. Barretta. Její výhodou je přímočará integrace do vlastních programů (jeden soubor s jednoduchým API) a podpora načítání všech běžně používaných formátů (zejména BMP, JPG, PNG, GIF). Od stejného autora je i knihovna `stb_image_write.h`, která obsahuje funkce pro ukládání obrázků ve formátech BMP, PNG nebo TGA. Obě knihovny jsou dostupné k volnému využití jako součást tzv. *public domain*[3].

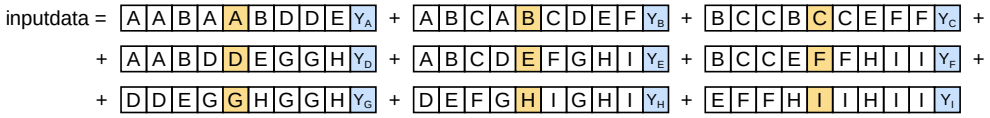
Načtený obrázek, který se bude používat na vstupech filtru, se ihned rozdělí na devítiokolí. U pixelů na okraji obrázku jsou na pozicích devítiokolí, které jsou už mimo obrázek, použity hodnoty nejbližších pixelů.

Pro sériové zpracování je výhodnější kvůli lepší lokalitě dat trénovací data uchovávat jako pole struktur, skládající se z devítiokolí a požadovaného výstupu. To je ale nevýhodné pro zpracování SIMD instrukcemi, kdy je potřeba do jednoho registru načíst několik pixelů ze stejné pozice v devítiokolí, které ale v paměti nejsou uloženy za sebou. Proto jsou devítiokolí v paměti uložena i jako struktura polí, kde jsou pixely z jedné pozice uloženy v samostatném poli a jedno pole obsahuje odpovídající požadované výstupní hodnoty. Oba způsoby uložení do paměti znázorňuje obrázek 4.1.

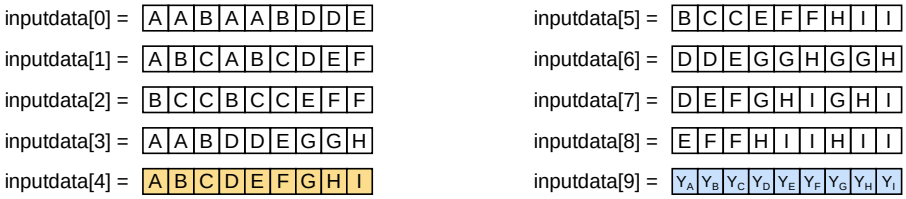
O něco složitější je výpočet přibližné fitness pomocí prediktoru. Protože vybírá pouze podmnožinu všech pixelů, jsou jejich hodnoty náhodně rozprostřeny v paměti. Pokud by se do registrů SIMD kopírovaly po jedné, byl by výpočet zbytečně pomalý. Řešení je podobné, jako u uložení dat celého obrázku. Každý prediktor obsahuje navíc pomocnou strukturu polí, která obsahuje hodnoty devítiokolí a požadované výstupní hodnoty filtru těch pixelů, které jsou obsaženy ve fenotypu prediktoru. Tato struktura se naplní daty ihned po vzniku prediktoru a také po každé změně jeho fenotypu (tj. velikosti).

[3] Lze je stáhnout na adrese https://github.com/nothings/stb.

(a) Zdrojový (zašuměný) a cílový (originální) obrázek



(b) Struktura dat pro sekvenční zpracování



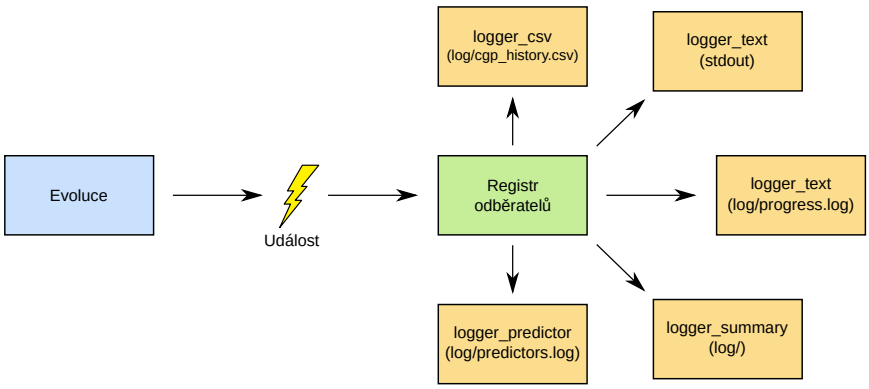
(c) Struktura dat pro vektorové zpracování

Obrázek 4.1. Uspořádání vstupních dat v paměti pro sekvenční a vektorové zpracování. Žlutě je označen střed devítiokolí, modře požadovaný výstup filtru

4.4 Sběr statistických dat

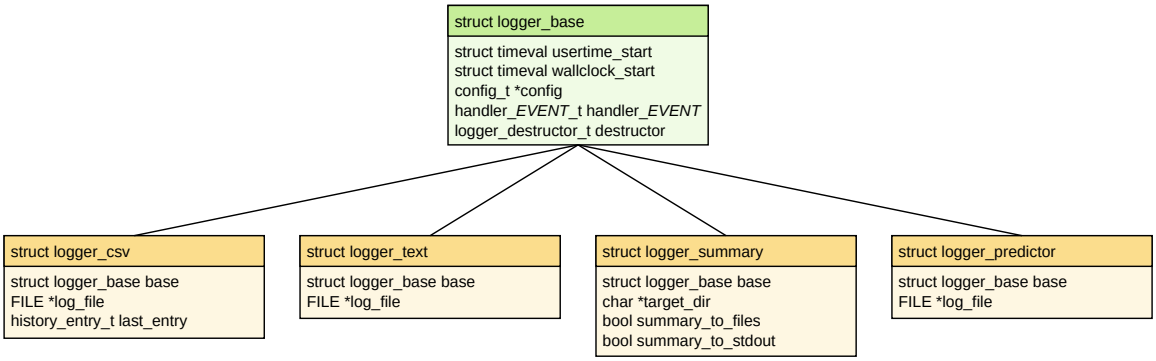
Běh evoluce a její výsledky jsou pro pozdější statistické zpracování zaznamenávány do souborů. Každý typ výstupu obsluhuje samostatný modul, které se přihlašují k odběru událostí – jde o návrhový vzor *pozorovatel* (*observer*). Události vznikají v hlavní smyčce evoluce a jsou následně předávány všem odběratelům, jak znázorňuje obrázek 4.2.

Implementace jednotlivých obslužných modulů událostí je inspirovaná objektovým programováním. Základním datovým typem je struktura `struct logger_base` („abstraktní třída“), která obsahuje položky pro ukazatele na obslužné funkce, konfiguraci programu a čas počátku evoluce. Prvním parametrem každé obslužné funkce je ukazatel na proměnnou



Obrázek 4.2. Způsob obsluhy událostí vzniklých během evoluce (např. nalezení lepšího kandidátního řešení) za účelem sběru statistických dat

tohoto typu („instanci třídy“), v jejímž kontextu se má obsluha události provést. Pokud některý modul pro svůj běh potřebuje uchovávat další údaje (například jméno cílového souboru), může definovat vlastní datový typ („odvozenou třídu“), přičemž jeho první položka musí být typu `struct logger_base`. Explicitním přetypováním lze s takovou proměnnou pak pracovat stejně, jako by byla přímo typu `struct logger_base`. Tímto se částečně simuluje dědičnost tříd a polymorfismus – při předávání události odběrateli není třeba znát jeho konkrétní datový typ. Všechny implementované datové typy jsou uvedeny na obrázku 4.3.



Obrázek 4.3. Datové typy modulů obsluhující události a jejich hierarchie

Ve výchozím nastavení se k obsluze událostí zaregistruje pouze jedna „instance“ typu `logger_text`, který zapisuje na standardní výstup vybrané události v čitelné podobě. Pokud se v příkazové řádce uvede jméno cílové složky pro záznam evoluce, zaregistruje se druhá „instance“, která bude totéž zapisovat do souboru. Také se zaregistrují „objekty“ typů `logger_csv` a `logger_summary`. První z nich zapisuje průběh evoluce ve formátu CSV, který je pro další strojové zpracování vhodnější než textový výpis. Druhý zapíše do jednotlivých souborů nejlepší nalezený filtr a příslušný výstupní obrázek, celou konfiguraci programu a také soubor s krátkým slovním shrnutím výsledků. Nejlepší nalezený filtr je uložen do cílové složky v jednoduchém textovém formátu kompatibilním s programem CGP Viewer od Zdeňka Vašíčka, jako kód v jazyce C a také v grafické podobě (*ASCII Art*) snadno srozumitelné pro uživatele. Dalším parametrem příkazové řádky lze zapnout ukládání všech použitých prediktorů do cílové složky, což zajišťuje „instance“ typu `logger_predictor`.

V budoucnu je možné poměrně snadno přidat další implementace obsluhy událostí. Nabízí se například vytvořit modul, který by informace o průběhu evoluce zasílal grafickému uživatelskému rozhraní.

4.5 Generování náhodných čísel

Protože genetické algoritmy jsou ze své povahy stochastické, je potřeba v implementaci použít nějaký zdroj náhodných nebo pseudonáhodných čísel. Nejjednodušší je použít funkci `rand` obsaženou ve standardní knihovně. Protože se experimenty budou spouštět dávkově, není vhodné pro inicializaci pseudonáhodné posloupnosti použít funkci `time`, která vrací datum a čas s přesností na sekundy. Všechny běhy spuštěné ve stejné sekundě by pak byly inicializovány stejnou hodnotou a pracovaly se stejnou posloupností čísel.

Vhodnější je pro inicializaci použít funkci `gettimeofday`, která vrací aktuální čas od půlnoci s přesností na mikrosekundy. Pravděpodobnost, že dva běhy použijí pro inicializaci totožnou hodnotu je pak velmi malá, přestože se čísla mohou každých 24 hodin opakovat.

4.6 Výpočet fitness

Ať už se jedná o predikovanou nebo skutečnou fitness, její hodnota je určena jako špičková hodnota poměru signál/šum (Peak Signal to Noise Ratio) dle rovnice 3.1:

$$f\left(cgp\right) = 10\log_{10}\frac{255^2}{\frac{1}{N}\sum_{i}^{N}\left(v\left(i\right)-w\left(i\right)\right)^2}$$

Výpočet logaritmu je ale poměrně náročná operace. Jeho odstraněním je možné program urychlit, přičemž se nezmění ani monotónnost funkce ani pořadí kandidátních řešení v populaci. V implementaci se proto používá upravený vzorec:

$$f\left(cgp\right) = \frac{255^2}{\frac{1}{MN}\sum_{i,j}\left(v\left(i,j\right)-w\left(i,j\right)\right)^2} \tag{4.1}$$

Kapitola 5

Experimentální vyhodnocení

Tato kapitola popisuje provedené experimenty a parametry testovacího prostředí a také srovnání výsledků získaných navrženým algoritmem s adaptivní velikostí prediktorů fitness, koevolucí s prediktory pevné délky a také se standardním CGP bez koevoluce.

Všechny výpočty byly prováděny na superpočítači Anselm, který provozuje Národní superpočítačové centrum IT4Innovations. Skládá se celkem z 209 výpočetních uzlů, z toho 180 obsahuje vždy dva osmijádrové procesory Intel Xeon E5-2665 (2,4 až 3,1 GHz s technologií Turbo Boost) a 64 GB operační paměti. Zbývající uzly mají po dvou procesorech Intel Xeon E5-2470 (2,3 až 3,1 GHz s technologií Turbo Boost) a alespoň 96 GB operační paměti [8]. Některé pomocné výpočty probíhaly i na počítačích zapojených do projektu Národní Gridové Infrastruktury MetaCentrum.

5.1 Testovací problémy

Pro srovnání navrženého algoritmu se standardním (ko)evolučním CGP byly vybráno několik úloh z oblasti zpracování obrazu. V prvním případě je cílem nalézt vhodný obrazový filtr pro rekonstrukci obrázků poškozených šumem typu sůl a pepř. U tohoto šumu jsou poškozené pixely buď bílé nebo černé – mají minimální nebo maximální možnou hodnotu. Příčinou mohou být vadné body ve snímači v kameře, nefunkční paměťové buňky nebo chyby během přenosu dat. Z konvenčních filtrů je možné použít mediánový, během filtrování ale dochází ke ztrátě detailů a výsledky u vyšší intenzity šumu nejsou uspokojivé.

Druhým typem šumu použitém během experimentování je náhodný impulzní šum, kdy mají poškozené pixely zcela náhodnou hodnotu. U tohoto typu šumu lze také použít mediánový filtr, se stejnými omezeními. V obou případech je šum charakterizován svou intenzitou – poměrným počtem poškozených pixelů. Pro evoluci filtrů byl použit obrázek 5.1(a).

Poslední testovací úloha spočívá v nalezení filtru, který provádí detekci hran v obraze. Výstupem detektoru je jednobitový černobílý obrázek, ve kterém jsou hrany označeny bílou barvou. Mezi konvenční filtry patří Sobelův nebo Cannyho detektor hran. V tomto případě byly filtry trénovány na obrázku 5.1(b).

5.2 Parametry evoluce

Parametry CGP a koevoluce vychází z článku [21]. Kartézské genetické programování pracuje s mřížkou o 8 sloupcích a 4 řádcích, parametr l-back je roven jedné. Každý program

(a) Šum typu sůl a pepř a náhodný impulzní šum

(b) Detekce hran

Obrázek 5.1. Obrázky používané pro evoluci filtrů

má 9 primárních vstupů, na které jsou přiváděny devítiokolí pixelů trénovacího obrázku a jeden výstup, který určuje filtrovanou hodnotu pixelu uprostřed devítiokolí. V populaci je celkem 8 jedinců, přičemž jeden z nich se stává rodičem populace a potomci jsou od něj odvozeni mutací 1 až 5 genů. Průběh evoluce je podrobněji popsán v kapitole 2.3.

Populace prediktorů fitness obsahuje 32 jedinců, kteří se vyvíjejí genetickým algoritmem popsaným v kapitole 2.2. Po ohodnocení jedinců je vytvořena nová populace. Jedna čtvrtina nové populace vznikne prostým zkopírováním nejlepších 8 jedinců z předešlé generace. Druhá čtvrtina je vytvořena zcela náhodně, což vede na vyšší diverzitu jedinců a brání to degradaci populace. Zbylých 16 jedinců vznikne křížením a mutací. Rodiče těchto jedinců určuje turnaj, v němž mezi sebou soupeří vždy dva náhodně zvolení jedinci. Jejich potomek pak vznikne jednobodovým křížením a následnou mutací až 5 % genů.

5.3 Hledání parametrů souběžného učení

V této práci je představen modifikovaný algoritmus koevoluce s prediktory fitness, kdy je navíc možné průběžně adaptovat velikost prediktorů na průběh evoluce fitness kandidátních kartézských programů. Tato nová část algoritmu vyžaduje několik nových parametrů, jejichž optimální hodnotu je třeba určit experimentálně. Tato část popisuje, jakým způsobem různá nastavení těchto parametrů ovlivňují kvalitu nacházených obrazových filtrů a celkovou rychlost programu. Není-li uvedeno jinak, jsou použity pravidla změny velikosti prediktoru z tabulky 3.2(a), tedy nová velikost je relativní k současné velikosti prediktoru.

5.3.1 Nepřesnost predikce

Ukázalo se, že pokud má prediktor dostatečnou délku, bývá hodnota predikované fitness velmi blízko hodnotě skutečné fitness. Avšak v případě, že je prediktor již příliš krátký, může být predikovaná fitness i několikanásobně vyšší. Jako vhodná mez, kdy nepřesnost predikce již negativně ovlivňuje průběh evoluce obrazových filtrů, se ukázala hodnota $I_{threshold} = 1,2$. Po jejím překročení jsou prediktory skokově prodlouženy na dvojnásobek ($c_I = 2$).

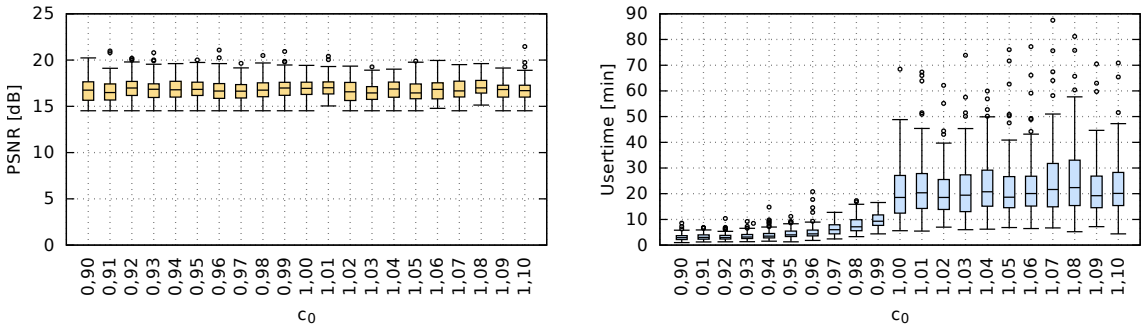
5.3.2 Změna velikosti prediktorů

Pro každý z parametrů byly testovány hodnoty z intervalu $0,9$ až $1,1$ (s krokem $0,01$), přičemž ostatní parametry měly fixní hodnoty. Počáteční velikost prediktoru byla nastavena na 50 % všech případů fitness. Evoluce byla ukončena po dosažení 30 tisíc generací CGP.

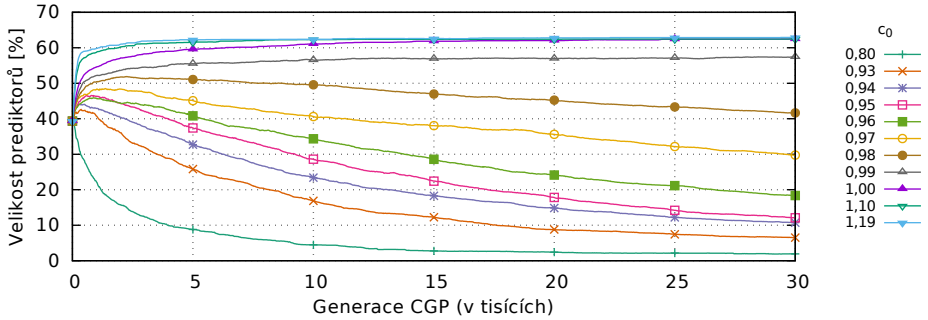
Při neměnné fitness filtrů

Za stagnující vývoj je považován stav, kdy je absolutní hodnota rychlosti evoluce $|v|$ nižší než hodnota parametru v_{zero}, který byl ve všech případech nastaven na $0,001$. Velikost prediktorů se v tomto případě upravuje pomocí koeficientu c_0. Pro každou testovanou hodnotu bylo spuštěno celkem 100 nezávislých běhů.

Jak je vidět na grafech 5.2, hodnota parametru c_0 neovlivňuje kvalitu nalezených filtrů. Ukazuje se ale, že tento parametr určuje trend směru změny velikosti prediktorů a tím i dobu výpočtu. U nižších hodnot velikost prediktoru v průběhu evoluce klesá. S rostoucí hodnotou má křivka průběhu velikosti prediktorů stále menší sklon a od hodnoty 0,99 velikost prediktorů už spíše roste. Pokud se hodnota parametru c_0 dále zvyšuje, tak se pouze urychluje konvergence k maximální možné velikosti prediktoru.



(a) Kvalita nalezených filtrů a doba běhu programu (součet všech vláken)



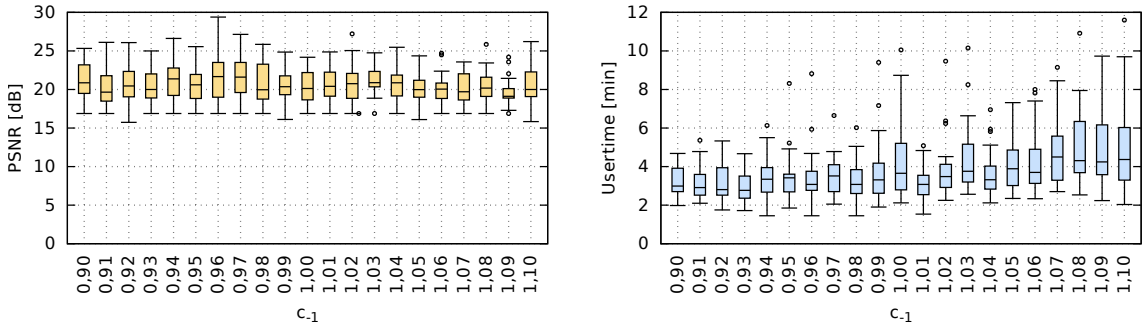
(b) Průběh velikosti prediktorů (průměr ze 100 běhů)

Obrázek 5.2. Vliv parametru c_0 (pravidlo $|v| \leq v_{zero} : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} \cdot c_0$)

Při klesající fitness filtrů

Druhé z pravidel se aplikuje v případě, že skutečná fitness nejlepšího jedince v populaci filtrů klesá ($v < -v_{zero}$). Velikost prediktoru se pak upraví dle parametru c_{-1}. Dosažené výsledky z 24 nezávislých běhů znázorňují grafy na obrázku 5.3. Tento parametr má větší vliv na kvalitu filtrů než parametr c_0, i když není úplně zřejmá vazba mezi hodnotou c_{-1} a dosaženou kvalitou. Jako optimální se z tohoto pohledu jeví hodnoty $0,96$ a $0,97$. Také lze pozorovat, že s vyšší hodnotou roste i počet evaluací CGP a tím i doba běhu programu, i když ne tak razantně, jako v případě parametru c_0. Zajímavá je hodnota $c_{-1} = 1$, kdy se velikost prediktoru nijak nemění. Zde je algoritmus výrazně pomalejší než v případě jiných

okolních hodnot. Zřejmě je pro evoluci výhodnější změna velikosti prediktoru jakýmkoliv směrem (a tím i změna predikované fitness filtrů), než pokud k žádné úpravě nedojde.



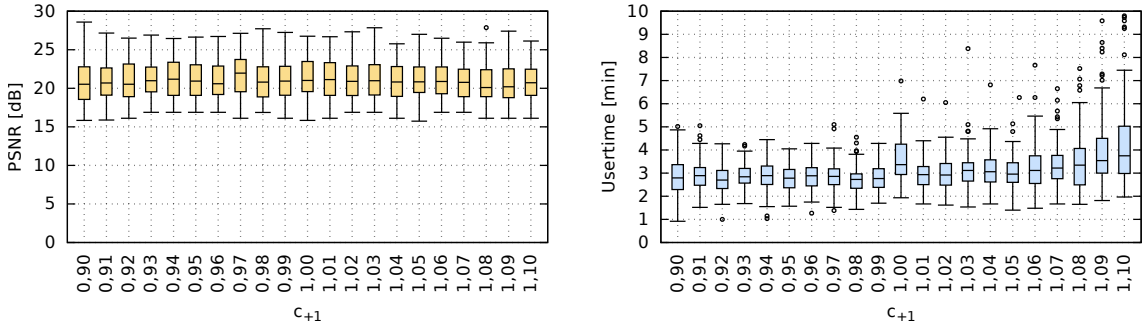
Obrázek 5.3. Vliv parametru c_{-1} (pravidlo $v < 0 : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} \cdot c_{-1}$)

Při rostoucí fitness filtrů

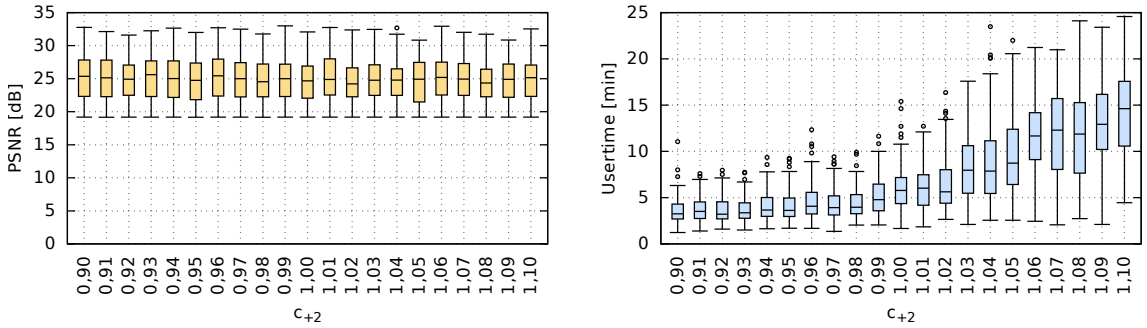
Pokud je rychlost v vyšší než nula, rozlišuje se „pomalý“ a „rychlý“ nárůst fitness, čemuž odpovídají koeficienty c_{+1} a c_{+2}. Jako práh byla zvolena hodnota $v_{slow} = 0,1$. Nejvhodnější hodnota obou parametrů byla hledána zvlášť, pro každé nastavení bylo spuštěno 100 běhů.

V případě, že fitness filtrů roste pomaleji, uplatní se parametr c_{+1}. Jak je vidět na grafech na obrázku 5.4, nemá jeho hodnota znatelný vliv na kvalitu filtrů. Podobně jako u jiných parametrů ovlivňující práci s velikostí prediktorů, má vyšší hodnota za následek pomalejší běh programu, ale nárůst není tolik výrazný. Také zde je při nastavení c_{+1} potřebný čas vyšší než u okolních hodnot.

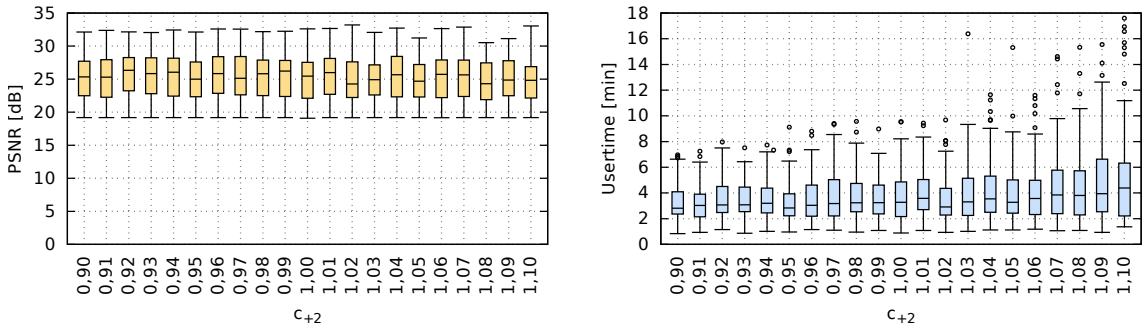
Také v případě parametru c_{+2} se ukazuje, že kvalita filtrů se příliš nemění, viz grafy na obrázku 5.5. Při pohledu na čas běhu programu se zdá, že je výhodnější i v tomto případě prediktory zkracovat. To je ale v rozporu s předpokladem (uvedeným v části 3.4.2), že pokud fitness filtrů roste, tak je vhodné prodlužovat prediktory a zpřesňovat predikovanou fitness. Ukazuje se, že je to způsobeno počáteční velikostí prediktorů (50 %), protože ta v průběhu evoluce konverguje k podstatně nižší hodnotě (jednotky procent), a proto je ve všech případech výhodnější zkracování. Pokud se počáteční velikost nastaví na 3 %, odpovídá chování předpokladům a vliv parametru c_{+2} na čas běhu programu není tak výrazný, jak je vidět na grafu 5.5(b).



Obrázek 5.4. Vliv parametru c_{+1} (pravidlo $0 < v \leq v_{slow} : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} \cdot c_{+1}$)



(a) Při počáteční velikosti prediktoru 50 %

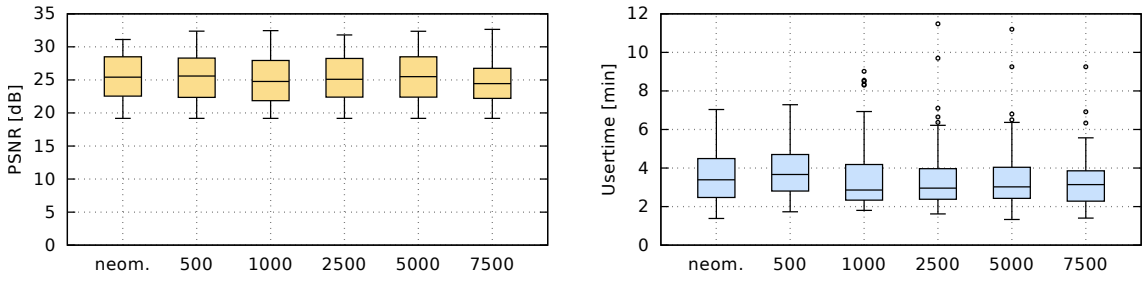


(b) Při počáteční velikosti prediktoru 3 %

Obrázek 5.5. Vliv parametru c_{+2} (pravidlo $v > v_{slow} : UsedGenes \leftarrow UsedGenes \cdot c_{+2}$)

5.3.3 Četnost změny velikosti prediktoru

Jak bylo zmíněno v kapitole 3.4.2, ke změně velikosti prediktoru dochází při každé výměně rodiče v populaci filtrů. Volitelným parametrem lze ale nastavit, aby ke změně velikosti došlo nejpozději po uběhnutí určitém počtu generací CGP od poslední změny. Testovány byly hodnoty 500, 1 000, 2 500, 5 000 a 7 500 generací, přičemž evoluce byla ukončena po 30 tisících generacích. Výsledky získané ze sta nezávislých běhů jsou na grafech 5.6. Ukazuje se, že tento parametr nemá příliš vliv ani na rychlost programu, ani na kvalitu filtrů.



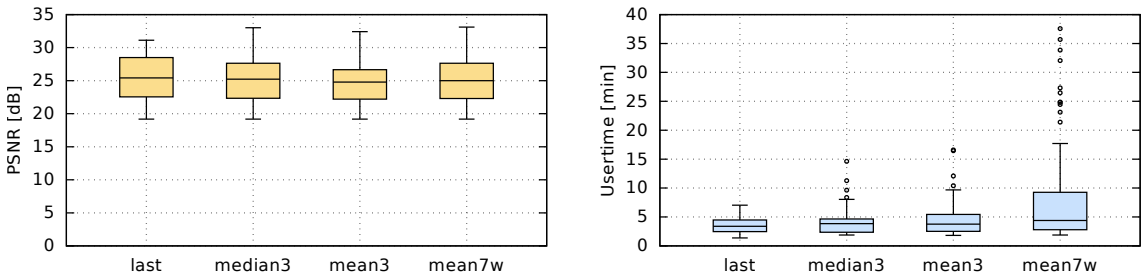
Obrázek 5.6. Vliv nastavení maximálního možného počtu generací mezi dvěma úpravami velikosti prediktoru

5.3.4 Způsob určení rychlosti evoluce

Rychlost evoluce v, podle které se určuje, jakým způsobem se upraví velikost prediktoru, lze určit několika způsoby. V předchozích experimentech záleželo jen na poslední změně fitness nejlepšího kandidátního filtru. Pro srovnání, zda lze jiným přístupem urychlit evoluci či zlepšit její výsledky, byly vybrány tyto varianty:

- podle poslední změny (výchozí nastavení, *last*),
- medián posledních 3 změn (*median3*),
- aritmetický průměr posledních 3 změn (*mean3*),
- vážený průměr posledních 7 změn (*mean7w*).

Dosažené výsledky po 30 tisících generacích a 100 nezávislých běhů jsou na grafech 5.7. Ukazuje se, že u všech zkoumaných variant je kvalita filtrů srovnatelná. Také potřebný čas běhu je podobný, s výjimkou varianty *mean7w*, u které je evoluce v průměru 2,2krát pomalejší než u *last*.

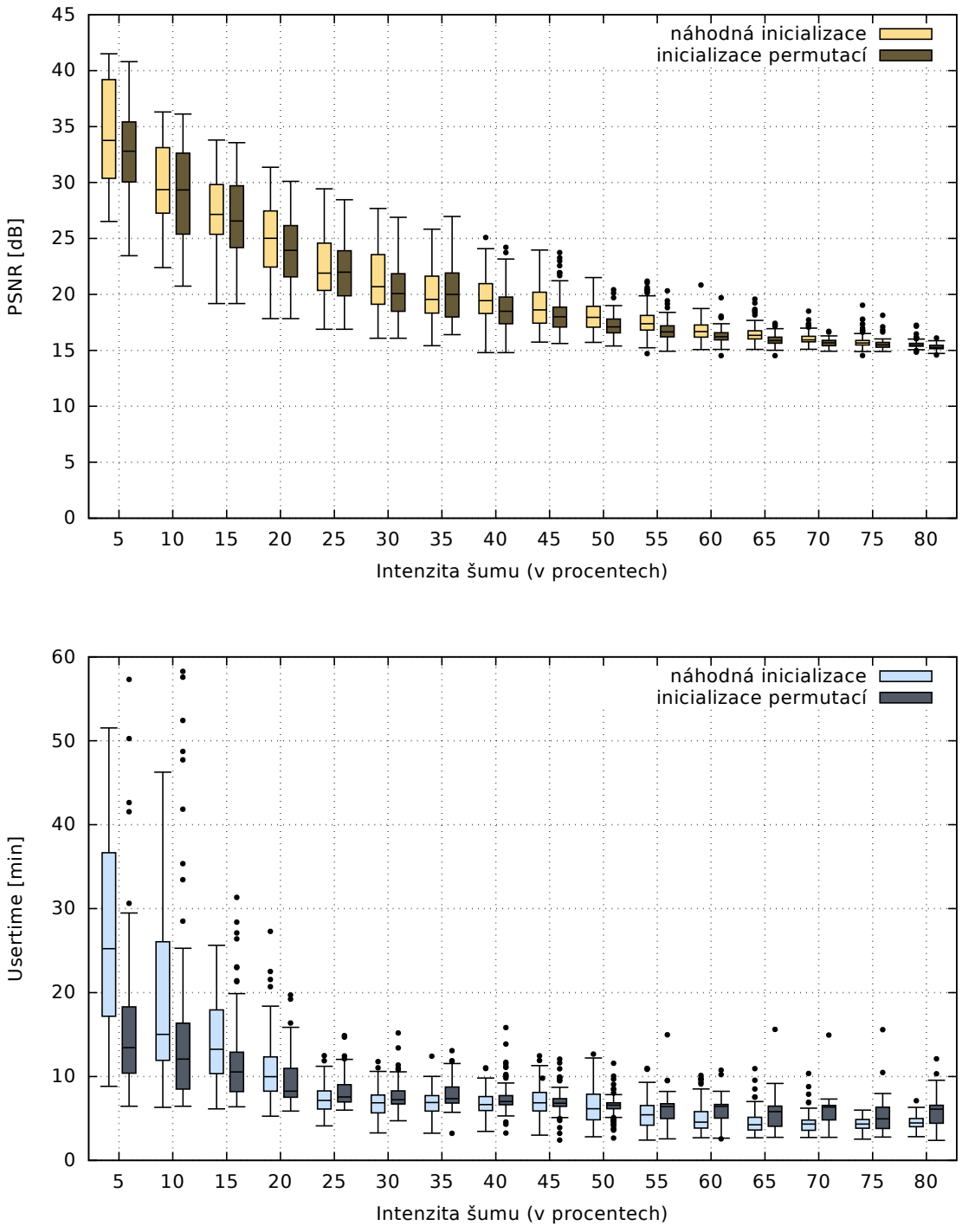


Obrázek 5.7. Vliv různých způsobů výpočtu rychlosti evoluce v

5.3.5 Způsob inicializace prediktorů

Při tvorbě nových prediktorů fitness je nejjednodušší vytvořit jejich genom zcela náhodně. Takto ale může vzniknout relativně velké množství duplicitních genů, což znamená kratší fenotyp a horší schopnost predikce fitness. Genotyp lze ale také inicializovat pomocí permutace všech možných hodnot, kde žádné duplicity nevznikají. Objevit se mohou až v dalším vývoji prediktoru buď křížením nebo mutacemi. Obě varianty byly testovány u filtrů pro šum typu sůl a pepř o intenzitě od 5 do 80 procent (s krokem po pěti procentech). Evoluce byla ukončena po 100 tisících generacích. Na obrázku 5.8 jsou grafy kvality získaných filtrů a doby běhu evoluce.

Ukazuje se, že při použití permutace ($init_{perm}$) je evoluce o něco rychlejší než náhodná inicializace ($init_{rand}$), ale výsledné filtry jsou obecně méně kvalitní. S rostoucí intenzitou šumu časový rozdíl klesá a od intenzity 55% je už $init_{perm}$ pomalejší. V případě $init_{perm}$ je u procesorového času také více odlehlých hodnot. Některé běhy byly i několikanásobně pomalejší, než byl průměr (např. u 5% šumu nejpomalejší běh spotřeboval 141 minut, přičemž průměr je 18,5 minut). Zdá se tedy, že náhodná inicializace prediktorů je výhodnější z pohledu kvality filtrů, i když evoluce trvá o něco déle.



Obrázek 5.8. Vliv způsobu inicializace prediktorů fitness pro různé intenzity šumu typu sůl a pepř

5.3.6 Změna velikosti prediktoru podle celkového počtu případů fitness

Experimentoval jsem také s druhou sadou pravidel, uvedených v tabulce 3.2(b), kdy je změna velikosti prediktoru vypočtena z celkového počtu případů fitness. Byly testovány hodnoty od -0.10 po $+0.10$ s krokem po 0.01 pro koeficienty c_0, c_{-1}, c_{+1} a c_{+2}. Grafy kvality nalezených filtrů a potřebného času pro různé hodnoty uvedených koeficientů jsou na obrázku 5.9. Ve všech případech platí, že různé hodnoty nemají vliv na kvalitu filtrů, ale jen na rychlost programu. Obecně platí, že při každé změně se prediktor zvětší či zmenší o větší počet případů fitness než v případě dosud popisované relativní změny a lze zde proto pozorovat výraznější zvýšení potřebného času u všech koeficientů. Potřebný čas stoupá až od určité hodnoty koeficientu, konkrétně:

- u c_0 od hodnoty 0,
- u c_{-1} od $-0,01$,
- u c_{+1} od $+0,03$,
- u c_{+2} od $+0,02$,

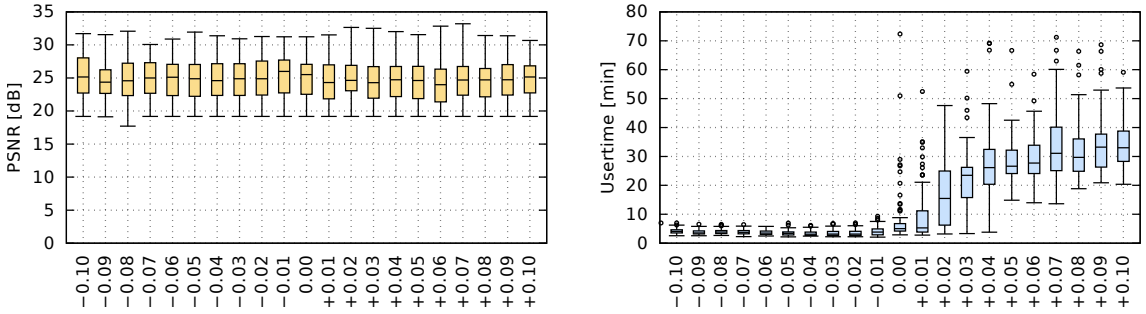
U nižších hodnot se rychlost algoritmu neliší tak výrazně. U parametru c_{-1} lze pozorovat mírné zpomalení programu i pro hodnoty nižší než $-0,06$. Navíc pro nulovou hodnotu je program znatelně pomalejší než u okolních hodnot, což bylo pozorováno i v experimentech s relativní změnou u koeficientů c_{-1} a c_{+1}.

5.4 Adaptace velikosti prediktoru na úlohu

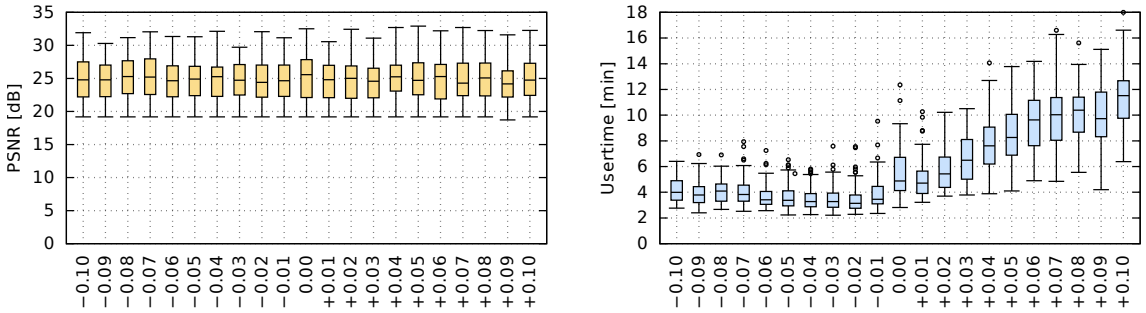
Pro ověření schopnosti adaptace velikosti prediktoru na řešený problém byla provedena řada výpočtů s různou počáteční velikostí prediktoru. Jako testovací problém byl zvolen šum typu sůl a pepř o intenzitě 5, 10, 15, 25 a 50 procent a detektor hran. Počáteční délka prediktoru byla 3 % a od 5 do 100 % případů fitness s krokem po pěti procentech. Tato velikost určuje počet genů použitých pro tvorbu fenotypu prediktoru, fenotyp pak je o něco kratší kvůli duplicitám v genomu. Ve všech případech byla sledována průměrná velikost ze 100 nezávislých běhů, evoluce byla ukončena po 100 tisících generacích.

Ukazuje se, že pokaždé velikost prediktoru konverguje ke stejné hodnotě, jak je vidět na grafech na obrázku 5.10. Ta závisí na intenzitě šumu, při nejnižší intezitě 5 % je konečná velikost okolo 25 % celkového počtu pixelů, při nejvyšším 50% šumu velikost konverguje ke 3 procentům. Také fitness nacházených filtrů je vždy přibližně stejná. Rychlost konvergence velikosti se liší podle typu šumu. U 5 % šumu už zhruba po šesti tisících generacích má prediktor přibližně 20 % případů fitness, zatímco v případě 50% šumu je potřeba asi 30 až 40 tisíc generací než velikost dosáhne zmíněná 3 procenta. Velikost prediktoru má dále rostoucí tendenci – to odpovídá navrženým pravidlům, podle kterých se mají prediktory při rostoucí fitness prodlužovat, aby se zpřesnila predikce.

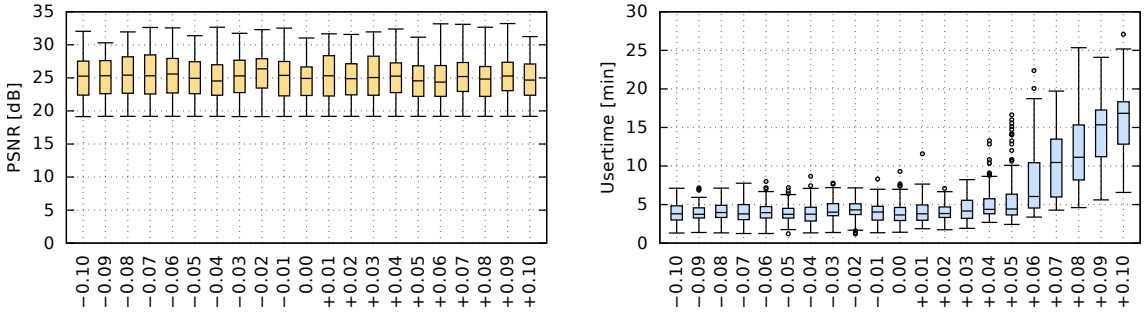
Protože délka prediktoru má zásadní vliv na rychlost běhu programu, je výhodnější začínat s menšími prediktory – jako ideální se z testovaných hodnot jeví počáteční velikost 3 % celkového počtu případů fitness.



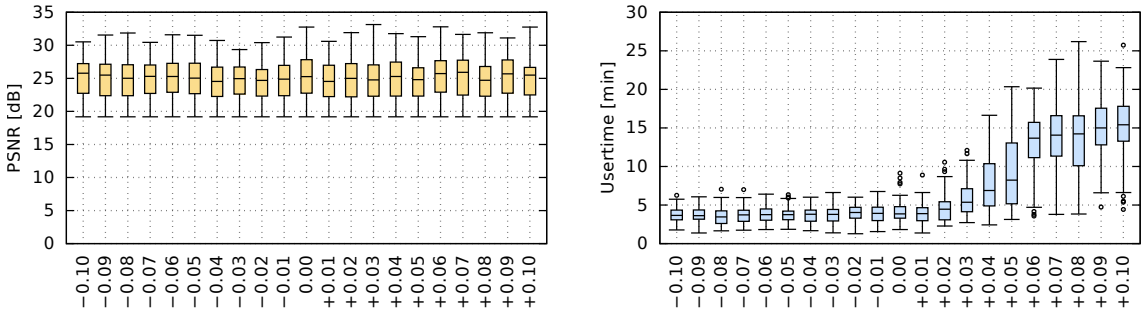
(a) Vliv parametru c_0 (pravidlo $|v| \leq v_{zero} : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} + N \cdot c_0$)



(b) Vliv parametru c_{-1} (pravidlo $v < 0 : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} + N \cdot c_{-1}$)

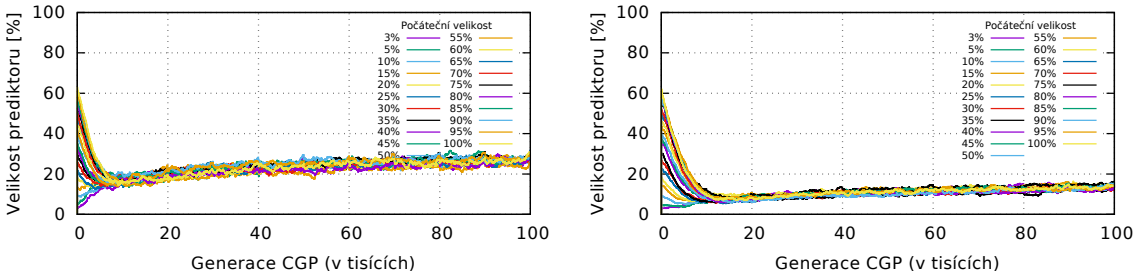


(c) Vliv parametru c_{+1} (pravidlo $0 < v \leq v_{slow} : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} + N \cdot c_{+1}$)



(d) Vliv parametru c_{+2} (pravidlo $v > v_{slow} : \mathit{UsedGenes} \leftarrow \mathit{UsedGenes} + N \cdot c_{+2}$)

Obrázek 5.9. Vliv koeficientů změny velikosti prediktorů v případě, že je velikost změny určena podle celkového počtu případů fitness



(a) 5% šum typu sůl a pepř

(b) 15% šum typu sůl a pepř

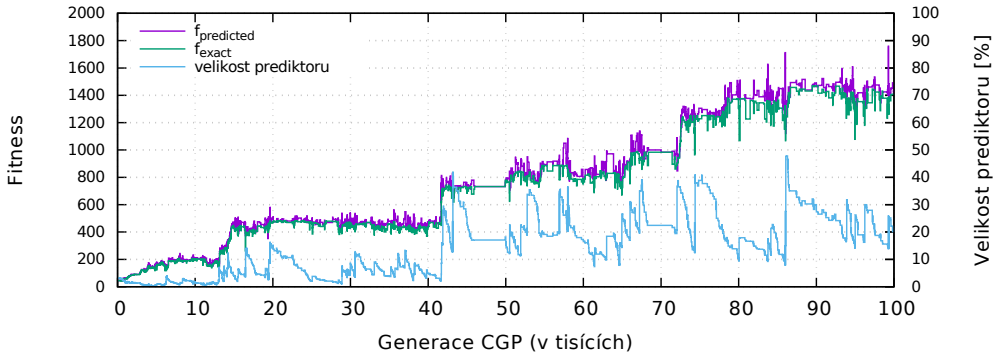
(c) 50% šum typu sůl a pepř

(d) Detektor hran

Obrázek 5.10. Vývoj velikosti prediktorů fitness při různé počáteční velikosti

5.4.1 Vztah fitness a velikosti prediktoru

Průběh vývoje fitness a velikosti prediktoru jednoho konkrétního běhu pro 15% šum typu sůl a pepř je na obrázku 5.11. Zde je na svislou osu namísto hodnoty PSNR vynesena čistá hodnota fitness, která se v programu používá, jak je popsáno v části 4.6. Počáteční velikost prediktoru zde byla 3 % případů fitness. Zpočátku se až na drobné výkyvy nemění, až při rychlejších změnách skutečné fitness velikost skokově stoupá. Po těchto skocích fitness stagnuje nebo mírně klesá a velikost prediktoru se pozvolna snižuje. Při dalším razantnějšího zvýšení fitness se velikost prediktorů opět zvýší. Velikost prediktoru se po některých změnách již nemusí vrátit k původním nižším hodnotám – v tomto případě přibližně po 42 tisících generacích velikost prediktoru skokově stoupne a drží si vyšší průměrnou hodnotu a až na ojedinělé výjimky neklesne pod 10 %.



Obrázek 5.11. Průběh predikované a skutečné fitness a velikosti prediktoru pro jeden konkrétní běh s 15% šumem typu sůl a pepř

5.4.2 Počet použití pravidel souběžného učení

V experimentech jsem se zajímal i o to, jak často se které pravidlo změny velikosti prediktoru uplatní. Bylo provedeno 100 nezávislých běhů s počáteční velikostí 3 % a 50 % pro 15% šum typu sůl a pepř. V prvním případě v průměru proběhlo 907 úprav velikosti prediktoru, což v průměru odpovídá jedné úpravě každých 110 generací. Úpravy velikosti lze rozdělit podle situace v populaci filtrů takto:

- 350× skutečná fitness klesala (koeficient c_{-1}),
- 270× skutečná fitness rychle rostla (koeficient c_{+2}),
- 145× skutečná fitness pomalu rostla (koeficient c_{+1}),
- 87× se skutečná fitness nezměnila (koeficient c_0),
- 22× byla predikce fitness příliš nepřesná (koeficient c_I).

Toto pořadí platí v celém průběhu evoluce. Pokud se použije počáteční velikost prediktoru 50 %, je celkový počet úprav nižší – v průměru jich bylo za celou dobu evoluce 600. Pořadí pravidel je stejné, s výjimkou prvních 10 tisíc generací, kdy se nejvíce používají koeficienty c_{+2} a c_{+1}.

5.5 Četnost výskytu pixelů v prediktorech

Zabýval jsem se také tím, jaká devítiokolí jsou obsažená v aktivních prediktorech a používají se při výpočtu predikované fitness – zda se prediktory zaměřují více na poškozené či nepoškozené pixely, v případě detektoru hran zda jsou častěji obsažené pixely v blízkosti hran než ty, které jsou uprostřed ploch.

K testování byly použity tři obrázky poškozené šumem typu sůl a pepř, dva byly poškozené po celé ploše šumem s intenzitou 15 a 50 procent, třetí obrázek byl částečně poškozen 25% šumem. Pro úlohu detekce hran byl zvolen obrázek „kameraman“. Pro každý obrázek bylo spuštěno 100 nezávislých běhů.

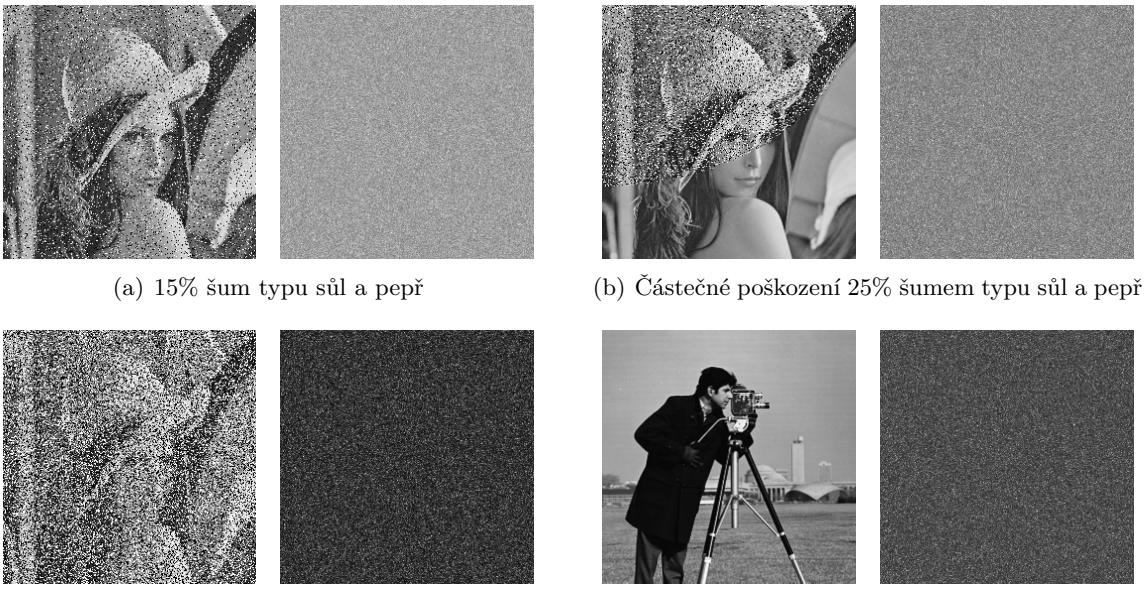
Na obrázku 5.12 jsou nalevo použité testovací obrázky a napravo 2D histogramy výskytu devítiokolí v prediktorech. Čím je bod v histogramu světlejší, tím častěji bylo pro výpočet predikované fitness použito devítiokolí, jehož střed je pixel odpovídající pozici v histogramu. Pokud je bod zcela bílý, bylo odpovídající devítiokolí používané po celou dobu evoluce, pokud je černý, nebylo použito vůbec.

Ukazuje se, že u intenzity 15 % je výskyt jednotlivých devítiokolí v prediktorech poměrně vyrovnaný. Nejvíce používané devítiokolí bylo použito 2,6krát častěji než nejméně používané. Algoritmus nemá tendenci se více zaměřovat na poškozené pixely. Výsledný filtr totiž musí nejen co nejlépe opravit poškozené pixely, ale zároveň i co nejméně poškodit pixely nezasažené šumem.

V histogramu u 50 % šumu je rozdíl mezi nejsvětlejšími a nejtmavšími body výraznější. Prediktory tady nevybírají devítiokolí tolik rovnoměrně jako u nižší intenzity šumu. Nejméně používané devítiokolí se vyskytovalo v aktivních prediktorech přibližně 46krát méně než to nejvíce používané. Také je větší počet devítiokolí, která nebyla použita vůbec – v průměru jich v jednom běhu bylo 26 446, což odpovídá přibližně 40 % všech případů fitness. V případě 15% šumu bylo nepoužitých devítiokolí průměrně 8 285.

U částečně poškozeného obrázku se nejvýrazněji projevuje, že jsou vybírány všechna devítiokolí rovnoměrně bez ohledu na umístění poškození obrázku. V histogramu není žádný rozdíl mezi poškozenou a nepoškozenou části obrázku. Podobně i v úloze detekce hran jsou devítiokolí vybírána rovnoměrně, jejich umístění na hraně či v ploše nemá na výběr vliv.

Rovnoměrný výběr devítiokolí v prediktorech je způsoben především tím, jak je navržena fitness funkce prediktorů. Cílem populace prediktorů je, aby predikovaná fitness byla ideálně stejná, jako kdyby se používaly všechny případy fitness. Jiné by to bylo v případě soutěživé koevoluce, kde je cílem prediktorů vybírat ty případy fitness, na kterých kandidátní filtry selhávají. Tam bývá v prediktorech obsaženo více poškozených devítiokolí [21].



(a) 15% šum typu sůl a pepř

(b) Částečné poškození 25% šumem typu sůl a pepř

(c) 50% šum typu sůl a pepř

(d) Detektor hran

Obrázek 5.12. Četnost výskytu pixelů v aktivních prediktorech fitness při návrhu různých obrazových filtrů. Nalevo je obrázek, který byl na vstupu filtru, napravo 2D histogram výskytu pixelů v prediktoru ve 100 bězích. Čím je pixel v histogramu světlejší, tím častěji byl použit v aktivních prediktorech

5.6 Srovnání souběžného učení, koevoluce a CGP

Navržený algoritmus se souběžným učením a adaptivními prediktory fitness byl porovnán s koevolučním CGP s prediktory fitness pevné délky a se standardním CGP bez koevoluce. Experimentovalo se se dvěma variantami souběžného učení. Jednotlivé algoritmy jsou v textu dále označovány takto:

- CGP_{STD}: standardní nekoevoluční CGP,
- FP_{FIX}: koevoluce s prediktory fitness pevné délky,
- FP_{REL}: souběžné učení se změnou velikosti podle současné velikosti,
- FP_{ABS}: souběžné učení se změnou velikosti podle celkového počtu případů fitness.

Konkrétní hodnoty parametrů pro souběžné učení jsou uvedeny v tabulce 5.1. Ve všech případech byla použita stejná implementace CGP i koevoluce. Stejně jako v předchozích experimentech byla porovnávána kvalita nalezených obrazových filtrů a délku běhu evoluce. Experimenty byly provedeny na úloze návrhu obrazových filtrů pro obrázky poškozené šumy typu sůl a pepř a náhodným impulzním šumem o intenzitách 5 až 80 % s krokem po 5 % a na úloze návrhu detektoru hran v obrazu.

Evoluce byla ukončena po 100 tisících generací. Kvalita filtrů byla porovnávána pomocí sady 12 obrázků uvedených na obrázku 5.13, které jsou běžně používané komunitou zabývající se zpracováním obrazu. Jako kritérium kvality filtrů byla použita funkce PSNR.

Tabulka 5.1. Použité hodnoty parametrů souběžného učení

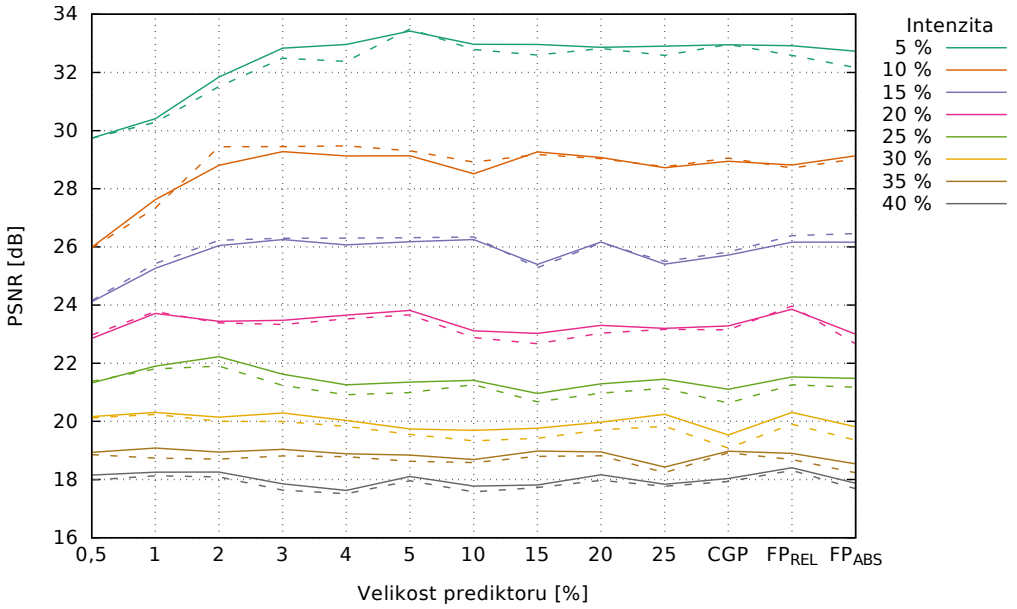
(a) FP_{REL}: změna podle současné velikosti prediktoru

parametr	hodnota	parametr	hodnota
$I_{threshold}$	1,2	v_{zero}	0,001
c_I	2	v_{slow}	0,1
c_0	0,90	počáteční velikost prediktorů	3 %
c_{-1}	0,96	inicializace prediktorů	náhodně
c_{+1}	1,07	četnosti změny velikosti	jen při zvýšení $f_{predicted}$
c_{+2}	1	výpočet rychlosti v	podle poslední změny (*last*)

(b) FP_{ABS}: změna podle celkového počtu případů fitness

parametr	hodnota	parametr	hodnota
$I_{threshold}$	1,2	v_{zero}	0,001
c_I	2	v_{slow}	0,1
c_0	-0,01	počáteční velikost prediktorů	3 %
c_{-1}	-0,07	inicializace prediktorů	náhodně
c_{+1}	0,01	četnosti změny velikosti	max. po 1 000 generacích
c_{+2}	0	výpočet rychlosti v	podle poslední změny (*last*)

Obrázek 5.13. Obrázky používané pro testování kvality filtrů



Obrázek 5.14. Hodnoty PSNR získané pomocí 12 testovacích obrázků pro FP_{FIX} s různou velikostí prediktoru, CGP_{STD}, FP_{REL} a FP_{ABS} pro různé intenzity šumu typu sůl a pepř. Plnou čarou je vyznačen průměr ze 100 nezávislých běhů, čárkovanou medián

Na obrázku 5.14 je vyznačen průměr a medián PSNR vypočteného nad 12 testovacími obrázky pro filtry získané vybranými evolučními algoritmy. Pro přehlednost jsou uvedeny jen výsledky pro šum typu sůl a pepř do intenzity 40 %. U 5–15% šumu je vidět, že menší prediktory již nevedou na dostatečně kvalitní filtry, jinak je kvalita srovnatelná pro všechny velikosti prediktoru. Toto platí i pro ostatní intenzity šumu a pro náhodný impulzní šum. Kvalita výsledných filtrů navrženého algoritmu se souběžným učením je srovnatelná s CGP_{STD} i FP_{FIX}. Podrobné grafy kvality filtrů a procesorového času pro všechny testované úlohy jsou uvedeny v příloze B.

Co se týče rychlosti, spotřebovává FP_{REL} a FP_{ABS} zhruba stejně procesorového času jako FP_{FIX} s 5% prediktorem. Neplatí to u šumu typu sůl a pepř o intenzitě 5 a 10 procent, kdy rychlost se souběžným učením odpovídá spíše FP_{FIX} s 10–15% prediktorem. Stále je ale rychlejší než standardní CGP a přitom není nutné experimentovat s nejvýhodnější velikostí prediktoru pro danou úlohu jako v případě FP_{FIX}.

Průměrný čas potřebný pro jeden běh CGP_{STD}, FP_{REL} a FP_{ABS} pro všechny intenzity šumu je uveden v tabulce 5.2. V průměru je zrychlení oproti CGP_{STD} 8,6násobné v případě FP_{REL} a 6,4násobné v případě FP_{ABS} při srovnatelné kvalitě výsledných filtrů.

5.7 Nalezené filtry

Výstup a zapojení nejlepšího evolvovaného filtru pro 25% šum typu sůl a pepř je na obrázku 5.15. Pro srovnání je zobrazen i výstup mediánového filtru s oknem 3×3. Oproti mediánovému filtru si evolvovaný filtr lépe poradil se shluky poškozených pixelů a zachovalo se více detailů. Také hodnota PSNR je vyšší, pro evolvovaný filtr činí 29,46 dB, kdežto u mediánového filtru to je pouze 24,61 dB. Ve filtru je použita pouze část možných funkcí bloků, nejčastěji se vyskytuje sčítání. V několika případech jde o součet primárního vstupu

Tabulka 5.2. Srovnání rychlosti CGP_{STD}, FP_{REL} a FP_{ABS}. Uvedené hodnoty jsou průměr procesorového času (součtu všech vláken) v minutách ze 100 nezávislých běhů

(a) Šum typu sůl a pepř

Intenzita	5 %	10 %	15 %	20 %	25 %	30 %	35 %	40 %
CGP_{STD}	65,15	64,98	64,91	65,22	65,11	64,83	64,76	64,54
FP_{REL}	26,89	18,54	11,85	10,59	7,29	6,87	6,84	6,73
Zrychlení	2,42×	3,50×	5,48×	6,16×	8,94×	9,44×	9,46×	9,59×
FP_{ABS}	23,05	18,11	13,64	10,72	9,58	9,84	9,80	9,77
Zrychlení	2,83×	3,59×	4,76×	6,09×	6,80×	6,59×	6,61×	6,61×

Intenzita	45 %	50 %	55 %	60 %	65 %	70 %	75 %	80 %
CGP_{STD}	64,85	64,98	64,65	64,53	64,80	64,73	64,68	64,90
FP_{REL}	7,01	6,39	5,53	5,10	4,56	4,39	4,35	4,51
Zrychlení	9,26×	10,17×	11,69×	12,64×	14,20×	14,75×	14,86×	14,40×
FP_{ABS}	9,51	9,49	9,94	9,48	9,78	9,56	9,24	8,96
Zrychlení	6,82×	6,85×	6,50×	6,80×	6,62×	6,77×	7,00×	7,24×

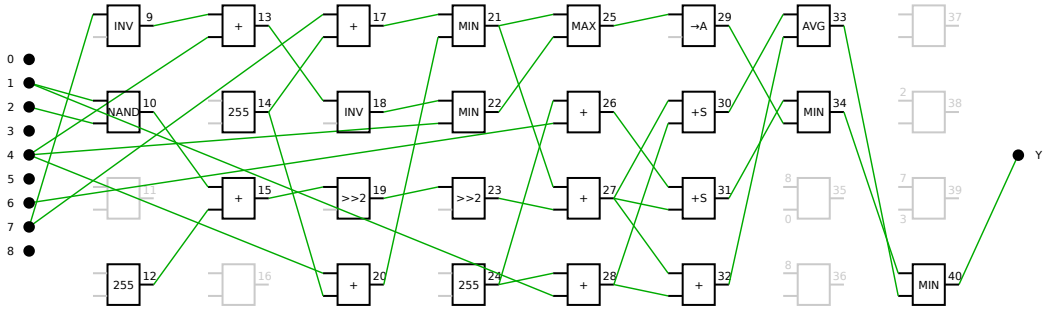
(b) Náhodný impulzní šum

Intenzita	5 %	10 %	15 %	20 %	25 %	30 %	35 %	40 %
CGP_{STD}	65,15	64,98	64,91	65,22	65,11	64,83	64,76	64,54
FP_{REL}	26,89	18,54	11,85	10,59	7,29	6,87	6,84	6,73
Zrychlení	2,42×	3,50×	5,48×	6,16×	8,94×	9,44×	9,46×	9,59×
FP_{ABS}	23,05	18,11	13,64	10,72	9,58	9,84	9,80	9,77
Zrychlení	2,83×	3,59×	4,76×	6,09×	6,80×	6,59×	6,61×	6,61×

Intenzita	45 %	50 %	55 %	60 %	65 %	70 %	75 %	80 %
CGP_{STD}	64,85	64,98	64,65	64,53	64,80	64,73	64,68	64,90
FP_{REL}	7,01	6,39	5,53	5,10	4,56	4,39	4,35	4,51
Zrychlení	9,26×	10,17×	11,69×	12,64×	14,20×	14,75×	14,86×	14,40×
FP_{ABS}	9,51	9,49	9,94	9,48	9,78	9,56	9,24	8,96
Zrychlení	6,82×	6,85×	6,50×	6,80×	6,62×	6,77×	7,00×	7,24×

a konstanty 255, což v případě 8bitových číslech bez znaménka odpovídá odečtení jedničky. Tato operace se na různých místech filtru provádí pro každý použitý primární vstup. Sníží se tak dynamický rozsah šumu – v případě černého pixelu (s hodnotou 0) je výsledkem hodnota 255, v případě bílého (255) je výsledkem 254.

(a) Poškozený obrázek (b) Mediánový filtr (c) Evolvovaný filtr



(d) Zapojení evolvovaného filtru

Obrázek 5.15. Srovnání mediánového a evolvovaného filtru pro 25% šum typu sůl a pepř

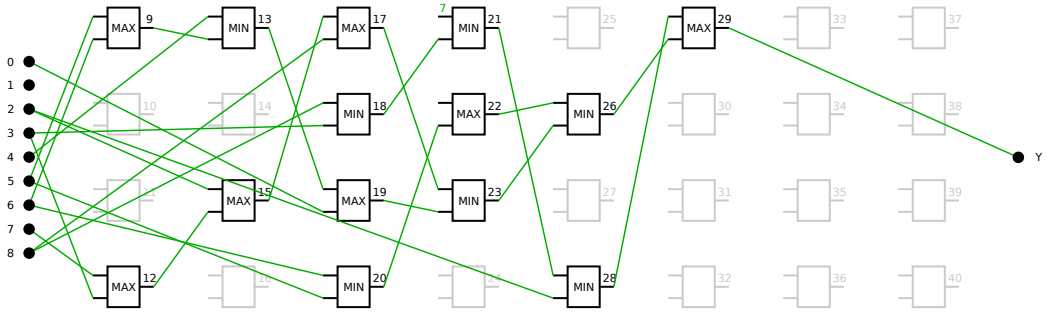
Na obrázku 5.16 je obdobné srovnání pro nejlepší nalezený filtr pro 10% náhodný šum. U tohoto šumu mají poškozené pixely náhodnou hodnotu a je proto obtížnější jej odstranit. Zatímco u šumu sůl a pepř jsou pixely, které nemají čistě bílou nebo černou barvu jednoznačně poškozené, u náhodného impulzního šumu to neplatí. V tomto případě je PSNR evolvovaného filtru 29,91 dB, což je jen o trochu více než u mediánového filtru, kde to je 28,96 dB. Nicméně opět platí, že i když evolvovaný filtr neodstraní veškerý šum, lépe zachovává detaily. Oproti filtru pro šum typu sůl a pepř je zcela jiná skladba použitých funkcí, používají se pouze funkce minimum a maximum. Také jsou zde použity téměř všechny primární vstupy, tj. body z devítiokolí zpracovávaného pixelu.

Schéma a výstup nejlepšího nalezeného filtru pro detekci hran je na obrázku 5.17. Pro srovnání je uveden i výstup Sobelova detektoru, který byl použit při evoluci jako referenční obrázek. Evolvovaný filtr dobře detekuje hrany kromě věže, kterou ale nedetekuje ani Sobelův detektor. Je možné, že při použití jiného referenčního obrázku by ji evolvovaný filtr byl schopen detekovat. Výstup evolvovaného filtru je oproti Sobelova filtru osmibitový a hrany proto nejsou tak ostré. Toto by bylo možné vylepšit přidáním prahování, které by výstup filtru převádělo na čistě bílou nebo černou barvu. Množina použitých funkcí bloků je opět odlišná, nejvíce se používá sčítání se saturací a jeden blok realizuje funkci OR s negovaným vstupem, která se u uvedených filtrů pro odstranění šumu nevyskytovala vůbec.

(a) Poškozený obrázek

(b) Mediánový filtr

(c) Evolvovaný filtr



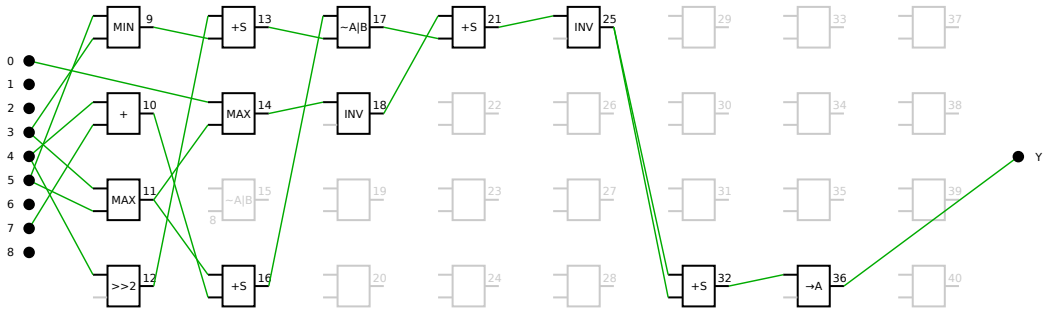
(d) Zapojení evolvovaného filtru

Obrázek 5.16. Srovnání mediánového a evolvovaného filtru pro 10% náhodný šum

(a) Vstupní obrázek

(b) Sobelův detektor

(c) Evolvovaný detektor



(d) Zapojení evolvovaného detektoru

Obrázek 5.17. Srovnání Sobelova a evolvovaného detektoru hran

Kapitola 6

Závěr

Tato diplomová práce se zabývá problematikou evolučních algoritmů, evolučního návrhu obrazových filtrů a vztahem mezi učením a evolucí. Jejím hlavním cílem bylo navrhnout a experimentálně otestovat systém pro tvorbu obrazových filtrů využívající souběžné učení v koevoluci s prediktory fitness. Pomocí souběžného učení lze překonat hlavní nevýhodu koevoluce, což je nutnost experimentálně hledat nejvýhodnější velikost prediktoru pro řešenou úlohu. Při použití učení je sledována fitness nejlepšího jedince v populaci kandidátních řešení a podle velikosti a směru jejích změn se pomocí jednoduchých pravidel upravuje velikost prediktoru. To je možné díky plastickému kódování, kdy lze ze stejného genotypu prediktoru odvodit různé fenotypy.

Navržený systém byl implementován v jazyce C s důrazem na co nejvyšší rychlost, proto byla použita paralelizace pomocí OpenMP a vektorizace pomocí instrukcí SSE2 a AVX2. Díky rychlé implementaci bylo možné provést větší množství experimentů zaměřených na studium vlastností a chování souběžného učení a na porovnání navrženého algoritmu se standardním a s koevolučním CGP. Porovnání proběhlo na 33 úlohách: na návrhu obrazových filtrů pro šum typu sůl a pepř a náhodný impulzní šum a na návrhu detektoru hran.

Z dosažených výsledků vyplývá, že souběžné učení dokáže vhodně adaptovat velikost prediktoru na řešenou úlohu. Zároveň tato velikost není náhodná, protože bez ohledu na počáteční nastavení pro stejnou úlohu algoritmus konverguje vždy ke stejně velkým prediktorům. Například pro 5% šum typu sůl a pepř je konečná velikost okolo 25 % celkového počtu pixelů, pro 50% šumu jsou to přibližně 3 procenta.

Ve srovnávacích experimentech se ukázalo, že kvalita výsledných obrazových filtrů je srovnatelná s filtry získanými pomocí standardního CGP i pomocí koevolučního CGP s vhodnou experimentálně určenou fixní velikostí prediktorů. Navržený algoritmus je oproti standardnímu CGP v průměru 8,6krát rychlejší a srovnatelně rychlý s koevolučním CGP s pevnou velikostí prediktorů, přičemž není třeba experimentovat s vhodnou velikostí pro právě řešenou úlohu. Je tedy umožněno aplikovat koevoluci prediktorů fitness k řešení nové, dříve neznámé, úlohy bez nutnosti experimentálního hledání vhodné velikosti prediktoru.

Při práci na tomto tématu jsem si zopakoval a upevnil principy evolučních algoritmů a obrazových filtrů. Při implementaci jsem si poprvé vyzkoušel práci s pokročilými instrukčními sadami, paralelizaci pomocí OpenMP a simulaci polymorfismu a dědičnosti v jazyce C. Také jsem si zkusil práci se superpočítačem Anselm a s MetaCentrem. Nové pro mě bylo i statistické zpracování poměrně velkého množství výstupních dat (jen záznamy průběhu evoluce měly asi 29 GB), kde jsem využil jak existující nástroje, tak vlastní skripty.

Tato práce byla publikována na studentské konferenci Excel@FIT 2015, kde byla jako jedna z 12 nejzajímavějších vybrána k ústní prezentaci.

Literatura

[1] BALDWIN, J. Mark. A New Factor in Evolution. *The American Naturalist*. 1896, roč. 30, č. 354, s. 441–451. ISSN 00030147.

[2] ELLEFSEN, Kai Olav. Balancing the Costs and Benefits of Learning Ability. In: *Advances in Artificial Life, ECAL 2013*. Cambridge (USA): MIT Press, 2013, s. 292–299. ISBN 978-0-262-31709-2.

[3] HILLIS, W. Daniel. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*. 1990, roč. 42, č. 1–3, s. 228–234. ISSN 0167-2789.

[4] HINTON, Geoffrey E.; NOWLAN, Steven J. How learning can guide evolution. *Complex systems*. 1987, roč. 1, č. 3, s. 495–502. ISSN 0891-2513.

[5] HRBÁČEK, Radek; ŠIKULOVÁ, Michaela. Coevolutionary Cartesian Genetic Programming in FPGA. In: *Advances in Artificial Life, ECAL 2013*. Cambridge (USA): MIT Press, 2013, s. 431–438. ISBN 978-0-262-31709-2.

[6] INTEL CORPORATION. *Intel® Advanced Vector Extensions Programming Reference* [online]. 2011 [cit. 2015-04-22]. Dostupný z WWW: ⟨https://software.intel.com/file/36945⟩.

[7] INTEL CORPORATION. Processors – Define SSE2, SSE3 and SSE4. [online]. 2015, [cit. 2015-04-08]. Dostupný z WWW: ⟨http://www.intel.com/support/processors/sb/CS-030123.htm?wapkw=sse2⟩.

[8] IT4INNOVATIONS. *Anselm Cluster Documentation: Hardware Overview* [online]. [cit. 2015-03-18]. Dostupný z WWW: ⟨https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview⟩.

[9] JONG, Kenneth De. Generalized Evolutionary Algorithms. In: *Handbook of Natural Computing*. 2011, s. 626–635. ISBN 978-3-540-92909-3.

[10] KHATIR, Mehrdad; JAHANGIR, Amir Hossein; BEIGY, Hamid. Investigating the Baldwin effect on Cartesian Genetic Programming efficiency. In: *2008 IEEE Congress on Evolutionary Computation*. 2008, s. 2360–2364. ISBN 978-1-4244-1822-0.

[11] LEMPEL, O.; PELEG, A.; WEISER, U. Intel's MMX™ Technology - A New Instruction Set Extension. In: *Compcon '97. Proceedings, IEEE*. 1997, s. 255–259. ISSN 1063-6390.

[12] MILLER, Julian F. Cartesian Genetic Programming. In: *Cartesian Genetic Programming*. 2011, s. 17–34. ISBN 978-3-642-17309-7.

[13] MILLER, Julian F.; THOMSON, Peter. Cartesian Genetic Programming. In: *Genetic Programming*. 2000, s. 121–132. Lecture Notes in Computer Science č. 1802. ISBN 978-3-540-67339-2.

[14] MRÁZEK, Vojtěch; VAŠÍČEK, Zdeněk. Evolutionary Design of Transistor Level Digital Circuits Using Discrete Simulation. In: *Genetic Programming*. 2015, s. 66–77. Lecture Notes in Computer Science č. 9025. ISBN 978-3-319-16500-4.

[15] POPOVICI, Elena; BUCCI, Anthony; WIEGAND, R. Paul et al. Coevolutionary Principles. In: *Handbook of Natural Computing*. 2011, s. 988–1028. ISBN 978-3-540-92909-3.

[16] QUINN., Michael J. *Parallel programming in C with MPI and openMP*. Boston: McGraw-Hill, 2004. ISBN 007-282256-2.

[17] REINDERS, James. AVX-512 instructions. *Intel Software Blogs* [online]. 2013, [cit. 2015-04-08]. Dostupný z WWW: ⟨https://software.intel.com/en-us/blogs/2013/avx-512-instructions⟩.

[18] SEKANINA, Lukáš; VAŠÍČEK, Zdeněk; RŮŽIČKA, Richard et al. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Praha: Academia, 2009. Edice Gerstner, sv. 4. ISBN 978-80-200-1729-1.

[19] SEKANINA, Lukáš; HARDING, Simon L.; BANZHAF, Wolfgang et al. Image Processing and CGP. In: *Cartesian Genetic Programming*. 2011, s. 181–214. ISBN 978-3-642-17309-7.

[20] ŠIKULOVÁ, Michaela; KOMJÁTHY, Gergely; SEKANINA, Lukáš. Towards Compositional Coevolution in Evolutionary Circuit Design. In: *2014 IEEE International Conference on Evolvable Systems Proceedings*. Piscataway (USA): Institute of Electrical a Electronics Engineers, 2014, s. 157–164. ISBN 978-1-4799-4479-8.

[21] ŠIKULOVÁ, Michaela; SEKANINA, Lukáš. Acceleration of evolutionary image filter design using coevolution in cartesian GP. In: *Parallel Problem Solving from Nature-PPSN XII*. 2012, s. 163–172. Lecture Notes in Computer Science č. 7491. ISBN 978-3-642-32937-1 ISSN 0302-9743.

[22] ŠIKULOVÁ, Michaela; HULVA, Jiří; SEKANINA, Lukáš. Indirectly Encoded Fitness Predictors Coevolved with Cartesian Programs. In: *Genetic Programming*. 2015, s. 113–125. Lecture Notes in Computer Science č. 9025. ISBN 978-3-319-16500-4.

[23] ŠIKULOVÁ, Michaela; SEKANINA, Lukáš. Coevolution in cartesian genetic programming. In: *Genetic Programming*. 2012, s. 182–193. Lecture Notes in Computer Science č. 7244. ISBN 978-3-642-29138-8.

[24] THAKKUR, S.; HUFF, T. Internet Streaming SIMD Extensions. *Computer*. 1999, roč. 32, č. 12, s. 26–34. ISSN 0018-9162.

[25] TURNEY, Peter. How to shift bias: Lessons from the Baldwin effect. *Evolutionary Computation*. 1996, roč. 4, č. 3, s. 271–295. ISSN 1063-6560.

[26] ULLAH, Fahad; KHAN, Gul Muhammad; MAHMUD, Sahibzada A. Exploiting developmental plasticity in Cartesian Genetic Programming. In: *2012 IEEE Symposium on Computers and Informatics (ISCI)*. 2012, s. 180–184. ISBN 978-1-4673-1685-9.

[27] VANNESCHI, Leonardo; POLI, Riccardo. Genetic Programming – Introduction, Applications, Theory and Open Issues. In: *Handbook of Natural Computing*. 2011, s. 710–734. ISBN 978-3-540-92909-3.

[28] VAŠÍČEK, Zdeněk; SLANÝ, Karel. Efficient Phenotype Evaluation in Cartesian Genetic Programming. In: *Genetic Programming*. 2012, s. 266–278. Lecture Notes in Computer Science č. 7244. ISBN 978-3-642-29138-8.

[29] WHITLEY, Darrell; SUTTON, Andrew M. Genetic Algorithms – A Survey of Models and Methods. In: *Handbook of Natural Computing*. 2011, s. 638–669. ISBN 978-3-540-92909-3.

Příloha A

Uživatelská příručka

A.1 Překlad programu

Program je rozdělen do několika modulů. Jejich překlad a sestavení výsledného programu je definován v Makefile a spouští se příkazem `make`. Je potřeba překladač GCC ve verzi alespoň 4.8, lépe ve verzi 4.9, která má vylepšenou podporu SIMD instrukcí, zejména umí využít všechny dostupné registry. Kvůli některým použitým funkcím (např. pro získání spotřebovaného procesorového času) musí operační systém implementovat standard POSIX. Program byl testován na systémech Ubuntu 12.04 a na superpočítači Anselm, který používá systém bullx Linux odvozený od Red Hat Enterprise Linux.

Pokud překlad končí chybou a na výstupu jsou takovéto zprávy:

```
/tmp/ccQ9hvQS.s: Assembler messages:
/tmp/ccQ9hvQS.s:33: Error: suffix or operands invalid for ‘vpcmpeqd’
/tmp/ccQ9hvQS.s:85: Error: suffix or operands invalid for ‘vpcmpeqd’
/tmp/ccQ9hvQS.s:92: Error: suffix or operands invalid for ‘vpcmpeqd’
```

je třeba z Makefile odstranit z proměnné `CFLAGS` část `-DAVX2`, čímž se vypne kompilace kódu akcelerovaného pomocí instrukcí AVX2. Chyby jsou s největší pravděpodobností způsobeny tím, že použitý překladač je novější verze a generuje kód, kterému použitý assembler nerozumí[1]. Tento problém se vyskytl na Anselmu, který AVX2 nepodporuje.

Jednotlivé moduly programu se překládají samostatně do složky `build`. Při změnách ve zdrojovém kódu tak není nutné opakovaně překládat moduly, do kterých se nezasahovalo. Výstupem překladu je několik spustitelných souborů:

- `coco`[2] – hlavní program, samotný evoluční návrh obrazových filtrů,
- `coco_apply` – aplikace filtru uloženého v souboru `*.chr` na libovolný obrázek,
- `coco_predvis` – vizualizace použitých prediktorů fitness,
- `coco_predhist` – tvorba histogramu pixelů používaných v prediktorech fitness.

[1] Zdroj: http://code.compeng.uni-frankfurt.de/issues/718

[2] Název `coco` je zkratka pro COlearning in COevolution.

A.2 Formáty souborů

Obrázky

Načítání trénovacích obrázků je řešeno pomocí knihovny `stb_image.h` (viz část 4.3). Jsou tedy podporovány formáty implementované v této knihovně, s většími či menšími omezeními. Konkrétně to jsou formáty JPEG, PNG, TGA, BMP, PSD, GIF, HRD a PIC. Všechny obrázky musí být uloženy ve stupních šedi (8 bitů na pixel). Testovány byly pouze obrázky ve formátech BMP a PNG. Výstupní obrázky (např. výstup nejlepšího filtru) jsou ukládány vždy ve formátu PNG.

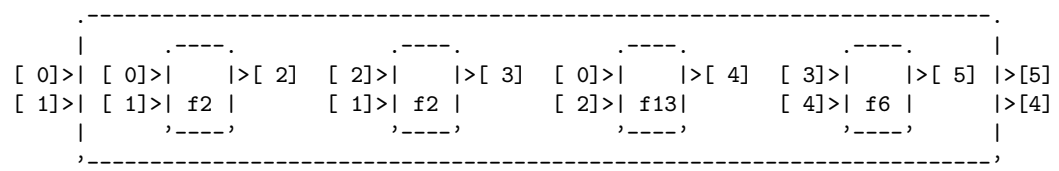
Obrazové filtry

Chromozomy obrazových filtrů jsou ukládány do textového formátu, který je totožný s formátem, který používá program CGP Viewer od Zdeňka Vašíčka a který vypadá takto:

```
{2, 2, 4, 1, 2, 1, 16}                                          Konfigurace CGP
([3] 0, 1, 2)([4] 2, 1, 2)([5] 0, 2, 13)([6] 3, 4, 6)           Vstupy a funkce bloků
(5, 4)                                                          Primární výstupy
```

Na začátku je ve složených závorkách uvedena konfigurace mřížky a funkčních bloků: počet primárních vstupů, primárních výstupů, šířka a výška mřížky, počet vstupů a výstupů funkčních bloků a počet možných funkcí, které může blok vykonávat. Dále jsou pro každý blok v kulatých závorkách uvedeny jeho číslo (v hranatých závorkách) a jeho geny: čísla bloků, na které jsou připojené jeho vstupy a číslo funkce, kterou vykonává. Na závěr jsou v kulatých závorkách uvedeny čísla bloků, kam jsou připojeny primární výstupy programu.

Filtry jsou ukládány také jako kód v jazyce C a v textovém formátu (jako ASCII Art) srozumitelném pro uživatele, který pro stejný program vypadá takto:



A.3 Spuštění evoluce obrazových filtrů

Povinné parametry jsou pouze dva, a sice:

- `--noisy FILE` nebo `-n FILE`,
- `--original FILE` nebo `-i FILE`.

Pomocí nich jsou předána jména souborů se zdrojovým (poškozeným) a cílovým (nepoškozeným) obrázkem. V případě detekce hran se u parametru `--noisy` uvede zdrojový obrázek a u `--original` obrázek s vyznačenými hranami. Ostatní parametry jsou volitelné, pokud nejsou uvedeny, použije se výchozí hodnota. Mezi standardním CGP, koevolucí a koevolucí se souběžným učením se přepíná pomocí parametru `--algorithm` nebo `-a`.

Evoluce se ukončí po dosažení stanoveného počtu generací nebo požadované kvality filtru. Při přijetí signálu SIGINT, SIGTERM nebo SIGXCPU se program ukončí korektně a vytvoří se i všechny výstupní soubory.

Příklady spuštění

`./coco -n noisy.png -i original.png -a cgp -g 10000`
Spustí standardní CGP, evoluce je ukončena po 10 000 generacích.

`./coco -n noisy.png -i original.png -a coev -S 50 -g 10000`
Spustí koevoluci s 50% prediktory fitness. Evoluce je ukončena po 10 000 generacích.

`./coco -n noisy.png -i original.png -a colearn -S 100 -I 3 -g 10000`
Spustí koevoluci se souběžným učením. Na počátku mají prediktory velikost 3 %. Evoluce je ukončena po 10 000 generacích.

Parametry evoluce

`--algorithm ALG, -a ALG` (výchozí: `coev`)
Algoritmus evoluce:

- `cgp` – standardní CGP,
- `coev` – koevoluce s pevnými prediktory fitness,
- `colearn` – koevoluce se souběžným učením.

`--random-seed N, -r N` (výchozí: dle `gettimeofday()`)
Inicializační hodnota generátoru pseudonáhodných čísel.

`--max-generations N, -g N` (výchozí: 50 000)
Evoluce se ukončí po dosažení zvoleného počtu generací CGP.

`--target-psnr N, -t N`
Evoluce se ukončí po dosažení zvolené hodnoty PSNR.

`--target-fitness N, -f N`
Evoluce se ukončí po dosažení zvolené fitness. Jde o hodnotu PSNR před logaritmováním, jak je uvedeno v části 4.6.

Parametry sběru dat

`--log-dir DIR, -l DIR`
Složka, do které se na konci evoluce uloží záznamy o běhu (viz část A.4). Pokud neexistuje, bude vytvořena.

`--log-interval N, -k N`
Kromě záznamů při změně fitness filtru, zapíše do logu také každou N-tou generaci.

`--log-pred-file FILE`
Do souboru `FILE` zapíše obsah všech prediktorů použitých při ohodnocování fitness. Soubor lze dále zpracovat nástroji `coco_predhist` a `coco_predvis`.

Parametry populace obrazových filtrů

`--cgp-mutate N, -m N` (výchozí: 5)
Maximální počet pozměněných genů filtru při mutaci.

`--cgp-population-size N, -p N` (výchozí: 8)
Počet jedinců v populaci filtrů.

`--cgp-archive-size N, -s N` (výchozí: 8)
Velikost archivu filtrů pro ohodnocování prediktorů.

Parametry populace prediktorů fitness

`--pred-size N, -S N` (výchozí: 0,25)
Velikost genotypu prediktorů v procentech (jako desetinné číslo). V případě souběžného učení je vhodné uvést hodnotu 1.

`--pred-mutate N, -M N` (výchozí: 5)
Maximální procento pozměněných genů prediktoru při mutaci.

`--pred-population-size N, -P N` (výchozí: 32)
Počet jedinců v populaci prediktorů.

`--pred-type TYPE, -T TYPE` (výchozí: `permuted` nebo `repeated` dle `--algorithm`)
Typ prediktoru:

- `permuted` – genotyp bez duplicitních genů, výchozí pro koevoluci s fixními prediktory (`-a coev`), nelze použít při souběžném učení,
- `repeated` – s duplicitními geny, výchozí pro souběžné učení (`-a colearn`),
- `repeated-circular` – s duplicitními geny, offset tvorby fenotypu se určí jako nejlepší z 5 náhodných možností.

Parametry souběžného učení

`--bw-pred-initial-size N, -I N` (výchozí: hodnota `--pred-size`)
Počáteční velikost prediktoru (hodnota *UsedGenes)* v procentech. Nesmí být vyšší než celková velikost prediktoru (daná parametrem `--pred-size`).

`--bw-pred-min N, -N N` (výchozí: 0)
Minimální velikost prediktoru (hodnota *UsedGenes*) v procentech.

`--baldwin-interval NUM, -b NUM` (výchozí: 0)
Maximální počet generací CGP mezi dvěma změnami velikost prediktoru. Hodnota 0 znamená, že se velikost upravuje jen při změně rodiče v populaci filtrů.

`--bw-inac-tol N` (výchozí: 1,2)
Parametr $I_{threshold}$

`--bw-inac-coef N` (výchozí: 2)
Parametr c_I

`--bw-zero-eps N` (výchozí: 0,001)
Parametr v_{zero}

`--bw-slow-thr N` (výchozí: 0,1)
Parametr v_{slow}

`--bw-by-max-length`
Změna velikost prediktoru je vypočtena z celkového počtu případů fitness a ne relativně k současné velikosti. Při použití této volby se namísto parametrů `--bw-XXXX-coef` používají parametry `--bw-XXXX-inc`

`--bw-zero-coef N` (výchozí: 0,9)
Parametr c_0 (pokud není použito `--bw-by-max-length`)

`--bw-decr-coef N` (výchozí: 0,96)
Parametr c_{-1} (pokud není použito `--bw-by-max-length`)

`--bw-slow-coef N` (výchozí: 1,07)
Parametr c_{+1} (pokud není použito `--bw-by-max-length`)

`--bw-fast-coef N` (výchozí: 1)
Parametr c_{+2} (pokud není použito `--bw-by-max-length`)

`--bw-zero-inc N` (výchozí: -0,01)
Parametr c_0 (při použití `--bw-by-max-length`)

`--bw-decr-inc N` (výchozí: -0,07)
Parametr c_{-1} (při použití `--bw-by-max-length`)

`--bw-slow-inc N` (výchozí: 0,01)
Parametr c_{+1} (při použití `--bw-by-max-length`)

`--bw-fast-inc N` (výchozí: 0)
Parametr c_{+2} (při použití `--bw-by-max-length`)

A.4 Výstupy programu

Pokud se uvede parametr `--log-dir DIR`, je vytvořena složka `DIR`, ve které se po ukončení evoluce nacházejí tyto soubory:

`best_circuit.c`
Nejlepší nalezený filtr jako zdrojový kód v jazyce C.

`best_circuit.chr`
Nejlepší nalezený filtr ve formátu pro CGP Viewer.

`best_circuit.txt`
Textový soubor s nejlepším nalezeným filtrem (mj. obsahuje filtr ve formátu CGP Vieweru a ASCII Artu).

`cgp_history.csv`
Záznam průběhu evoluce vhodný pro strojové zpracování ve formátu CSV. Nový řádek se zapíše při každé změně rodiče v populaci filtrů a v intervalech dle parametru `--log-interval`. Soubor obsahuje tyto sloupce:

- *generation*: číslo generace,

- *predicted_fitness*: predikovaná fitness nejlepšího filtru v populaci ($f_{predicted}$),
- *real_fitness*: skutečná fitness nejlepšího filtru v populaci (f_{exact}),
- *inaccuracy (pred/real)*: nepřesnost predikce I (viz část 3.4.2),
- *best_fitness_ever*: fitness dosud nejlepšího nalezeného filtru,
- *active_predictor_fitness*: fitness aktivního prediktoru,
- *pred_length*: počet použitých genů prediktoru (hodnota *UsedGenes*),
- *pred_used_length*: velikost fenotypu prediktoru,
- *cgp_evals*: počet CGP evaluací,
- *velocity*: rychlost evoluce v,
- *delta_generation*: počet generací od poslední změny fitness nejlepšího filtru,
- *delta_fitness*: velikost změny fitness nejlepšího filtru,
- *delta_velocity*: rozdíl současné a předchozí rychlosti evoluce ($v - v_{prev}$)
- *wallclock*: uběhlý čas (v minutách),
- *usertime*: spotřebovaný procesorový čas (v minutách),
- *pred_generation*: číslo generace prediktorů.

`config.log`
Kompletní konfigurace programu.

`img_best.png`
Výstup nejlepšího nalezeného filtru.

`img_noisy.png`
Použitý zdrojový (zašuměný) obrázek.

`img_original.png`
Použitý cílový (původní) obrázek.

`progress.log`
Záznam průběhu evoluce v textové podobě. Stejný záznam se vypisuje na standardní výstup programu.

`summary.log`
Krátké textové shrnutí výsledků evoluce. Vypadá například takto:

```
Final summary:

Generation: 30000
Best fitness: 49.30966296
PSNR: 16.93
CGP evaluations: 1270038112

Time in user mode: 3m8.050000s
Wall clock: 3m26.836038s
```

Použije-li se parametr `--log-pred-file FILE`, bude soubor `FILE` na každém řádku obsahovat číslo generace CGP, kdy se daný prediktor začal používat, jeho délku a indexy použitých případů fitness. Příklad:

```
Generation 0: Predictor phenotype length 4 [ 57341 16110 55958 61163 ]
Generation 1: Predictor phenotype length 5 [ 25693 2860 6516 4163 40272 ]
Generation 155: Predictor phenotype length 12 [ 41093 61202 40966 39558
    46992 7314 9857 19569 1033 63920 48820 46560 ]
```

A.5 Nástroj coco_apply

Tento nástroj slouží pro aplikaci evolvovaného filtru na libovolný obrázek, volitelně i pro výpočet PSNR a vizualizace zapojení filtru. Příklad použití:

```
./coco_apply --chromosome filter.chr --input lena.png > filtered.png
```

Povinné parametry

`--chromosome FILE, -c FILE`
: Soubor s uloženým chromozomem filtru ve formátu pro CGP Viewer.

`--input FILE, -i FILE`
: Vstupní obrázek filtru. Pokud není uveden nebo se uvede `-`, použije se standardní vstup.

`--output FILE, -o FILE`
: Výstupní obrázek filtru. Pokud není uveden nebo se uvede `-`, použije se standardní výstup.

Volitelné parametry

`--calc-psnr REFIMG, -p REFIMG`
: Na chybový výstup vypíše PSNR výstupního obrázku a referenčního obrázku.

`--print-ascii, -a`
: Na chybový výstup vypíše zapojení filtru ve formě ASCII Artu.

`--output FILE, -o FILE`
: Výstupní obrázek filtru.

A.6 Nástroj coco_predhist

Tento nástroj slouží pro tvorbu histogramů vybíraných prediktorů. Na standardní výstup vypíše počet použití každého devítiokolí ve formátu CSV. Volitelně vytvoří i 2D histogram jako obrázek. Příklad použití:

```
./coco_predhist --width 256 --height 256 --generations 100000 LOGFILE
```

Povinné parametry

`--width X, -x X`
: Šířka obrázku používaného při evoluci.

`--height Y, -y Y`
: Výška obrázku používaného při evoluci.

`--generations N, -g N`
: Počet generací, pro které se vytvoří histogram.

`LOGFILE`
: Za přepínači se uvedou jména jednoho či více souborů s výpisem použitých prediktorů, který se vygeneruje při evoluci při uvedení parametru `--log-pred-file`.

Volitelné parametry

`--output-image FILE, -o FILE`
: Vytvoří 2D histogram a uloží jej jako obrázek PNG do souboru `FILE`.

A.7 Nástroj coco_predvis

Tento nástroj slouží pro tvorbu vizualizaci všech použitých prediktorů. Na standardní výstup vypíše číslo generace a počet obsažených devítiokolí pro každý použitý prediktor. Volitelně vizualizuje použité pixely v každém prediktoru. Takto byly vygenerovány histogramy na obrázku 5.12. Příklad použití:

```
./coco_predvis --log predictors.log --image lena.png --outdir predvis
```

Povinné parametry

`--log FILE, -l FILE`
: Soubor s výpisem použitých prediktorů, který se vygeneruje při evoluci při uvedení parametru `--log-pred-file`.

Povinné parametry pro vizualizaci

`--image FILE, -i FILE`
: Podkladový obrázek pro vizualizaci prediktorů. Typicky jde o obrázek používaný při evoluci.

`--outdir FILE, -o FILE`
: Složka, do které se uloží vizualizované prediktory. Soubory jsou pojmenovávány podle čísla generace. Pokud složka neexistuje, vytvoří se.

Volitelné parametry pro vizualizaci

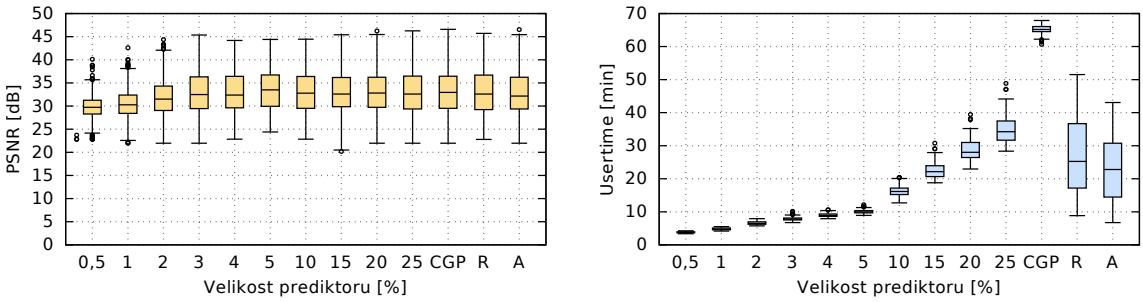
`--color RRGGBB, -o RRGGBB` (výchozí: červená – `FF0000`)
: Barva bodů, které při vizualizaci označují použitá devítiokolí. Zadává se v hexadecimálním „HTML“ formátu.

Příloha B

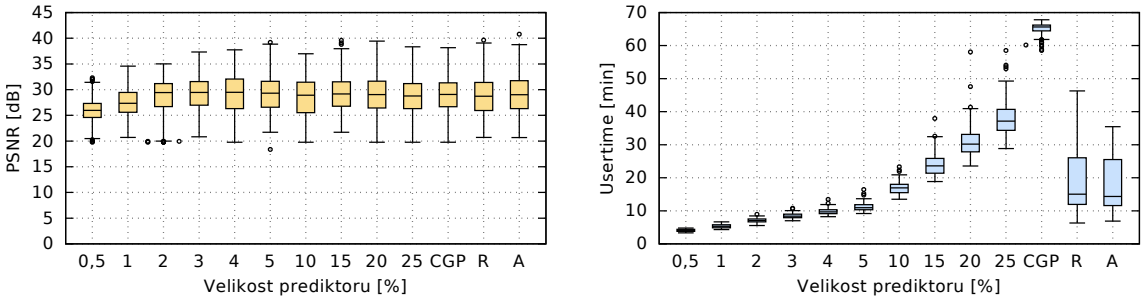
Výsledky experimentů

Následující grafy zobrazují kvalitu získaných filtrů pomocí standardního CGP, koevoluce prediktorů fitness různé velikosti a koevoluce se souběžným učením. Podrobnější nastavení je uvedeno v části 5.6. Sloupec „R“ označuje FP_{REL} a sloupec „A“ označuje FP_{ABS}). Hodnota PSNR je průměr pro 12 testovacích obrázků (viz obrázek 5.13). Čas běhu programu je určen jako součet procesorového času všech vláken. Pro každou intenzitu šumu (resp. pro detektor hran) a pro každý algoritmus a nastavení bylo provedeno 100 nezávislých běhů.

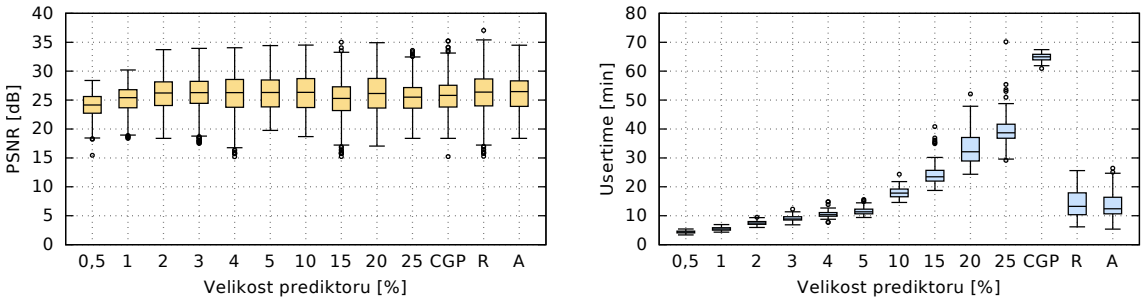
B.1 Šum typu sůl a pepř



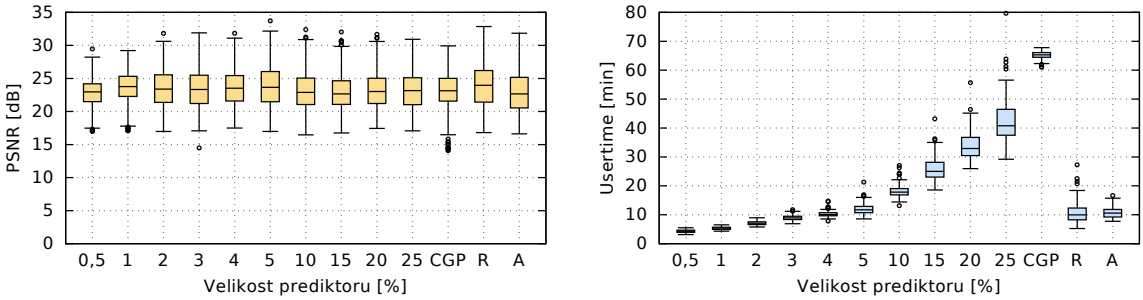
Obrázek B.1. 5% šum typu sůl a pepř



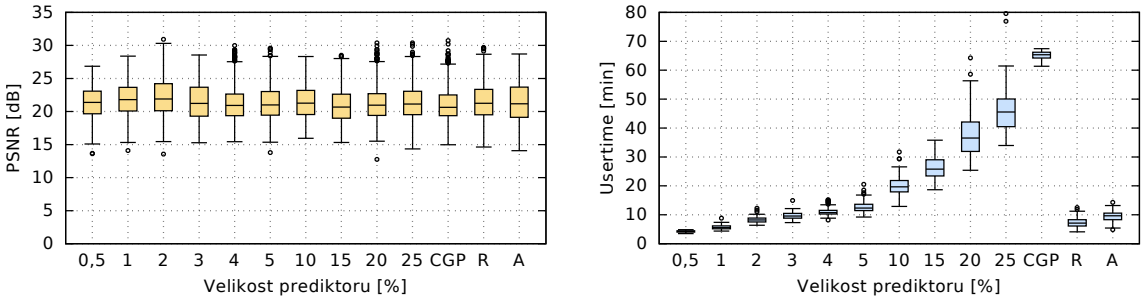
Obrázek B.2. 10% šum typu sůl a pepř



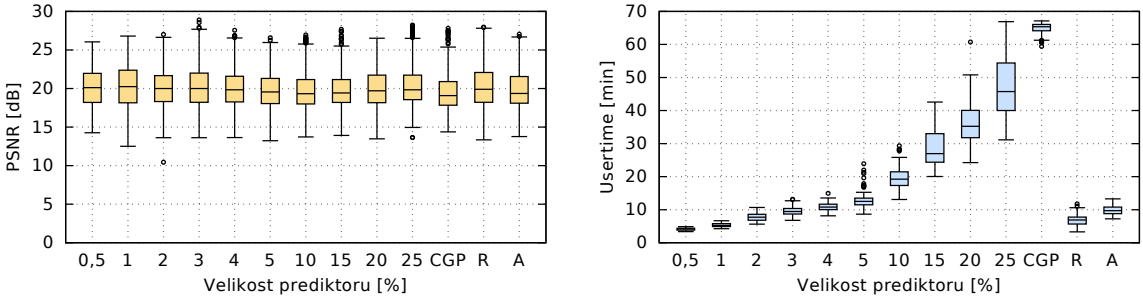
Obrázek B.3. 15% šum typu sůl a pepř



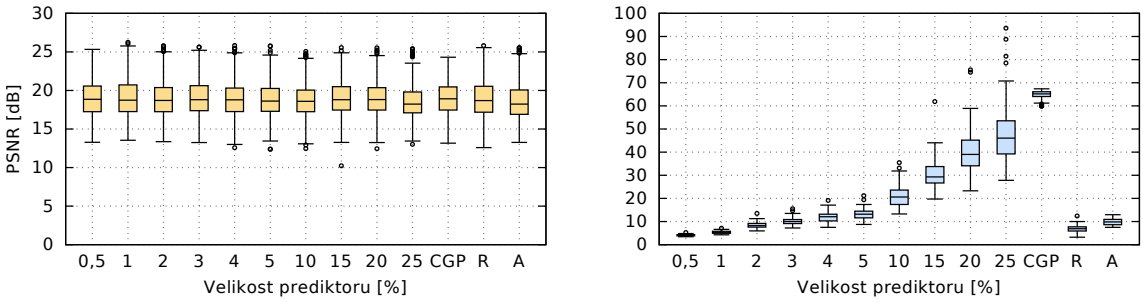
Obrázek B.4. 20% šum typu sůl a pepř



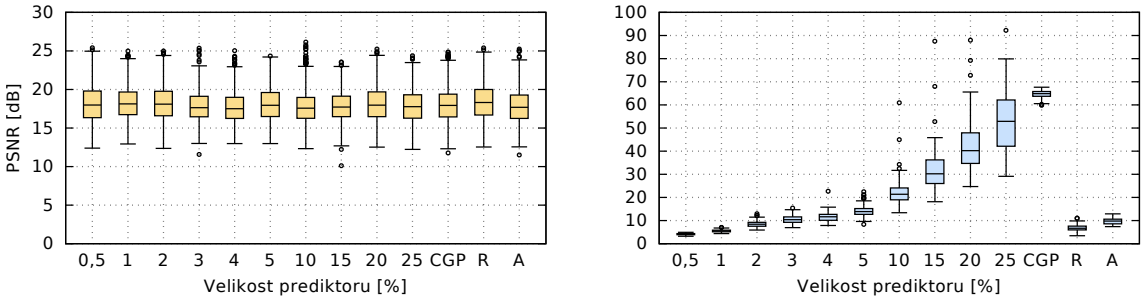
Obrázek B.5. 25% šum typu sůl a pepř



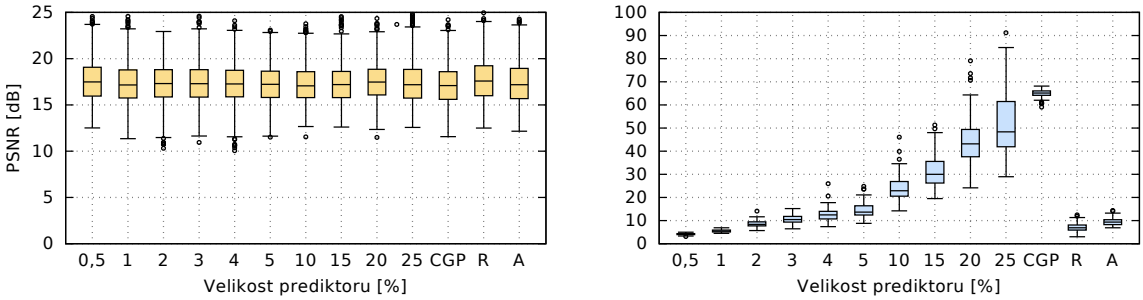
Obrázek B.6. 30% šum typu sůl a pepř



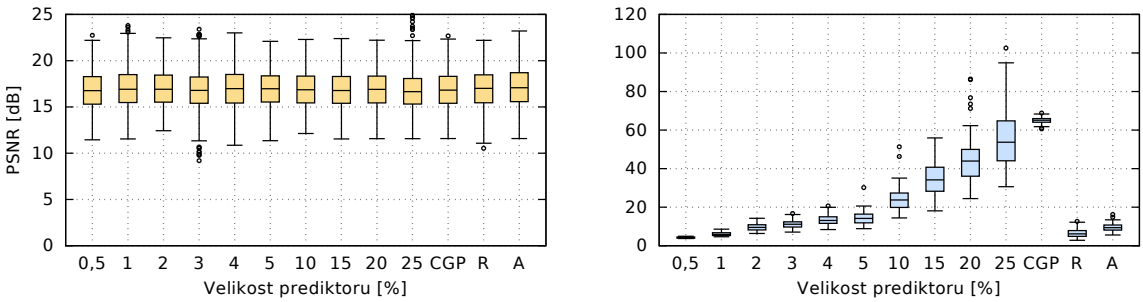
Obrázek B.7. 35% šum typu sůl a pepř



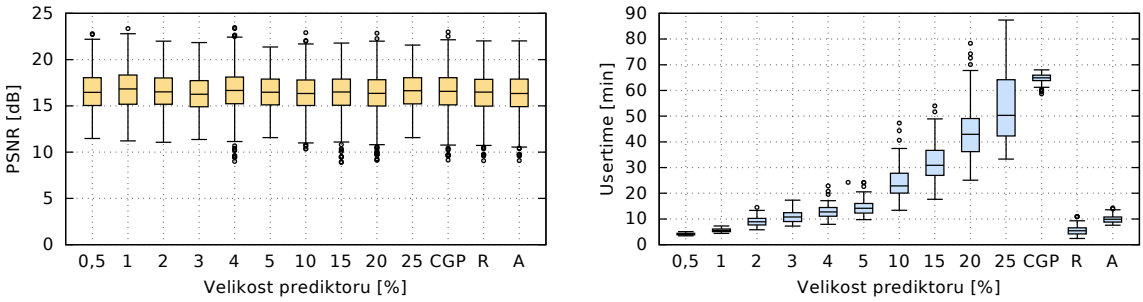
Obrázek B.8. 40% šum typu sůl a pepř



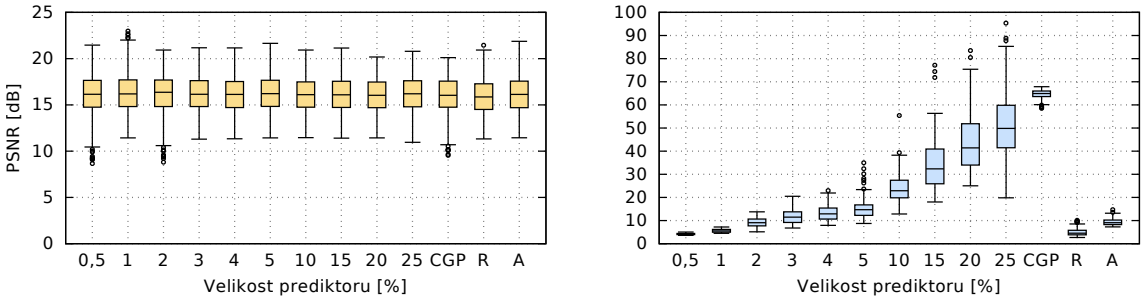
Obrázek B.9. 45% šum typu sůl a pepř



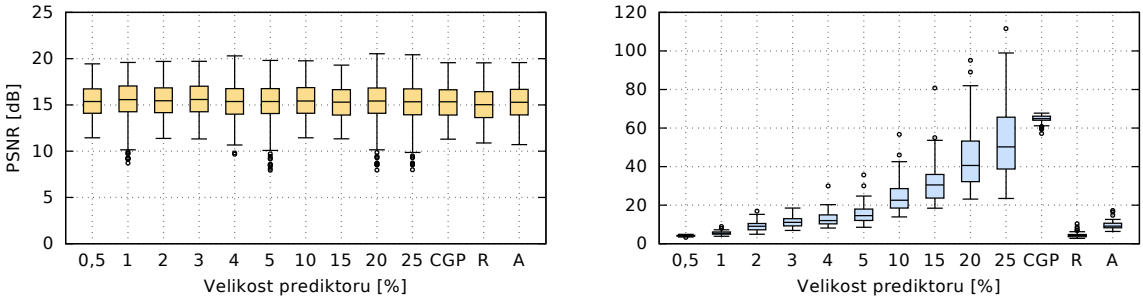
Obrázek B.10. 50% šum typu sůl a pepř



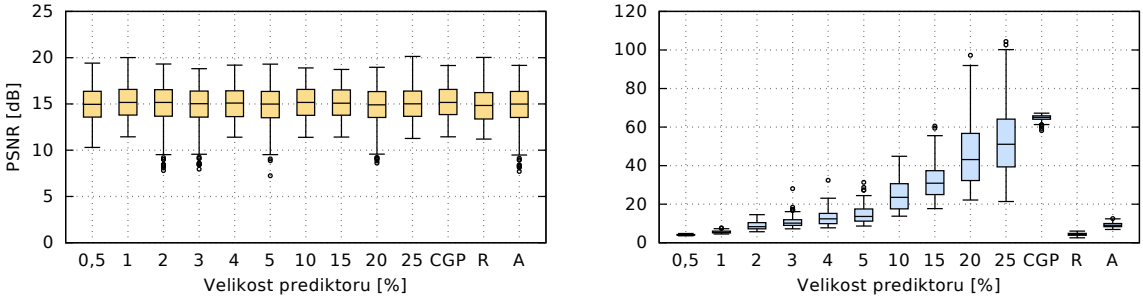
Obrázek B.11. 55% šum typu sůl a pepř



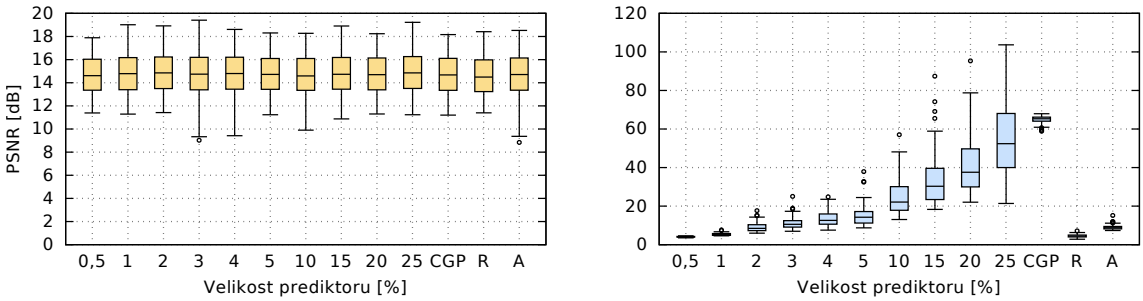
Obrázek B.12. 60% šum typu sůl a pepř



Obrázek B.13. 70% šum typu sůl a pepř

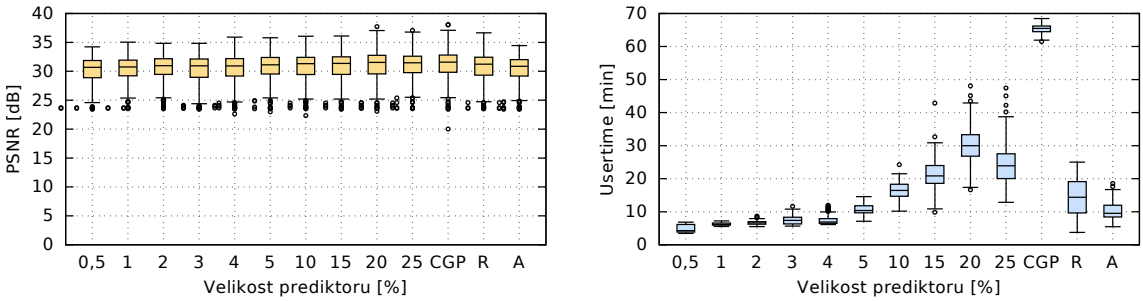


Obrázek B.14. 75% šum typu sůl a pepř

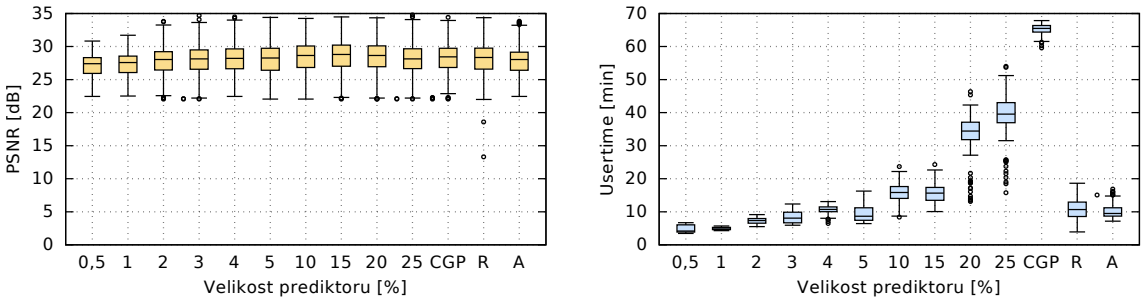


Obrázek B.15. 80% šum typu sůl a pepř

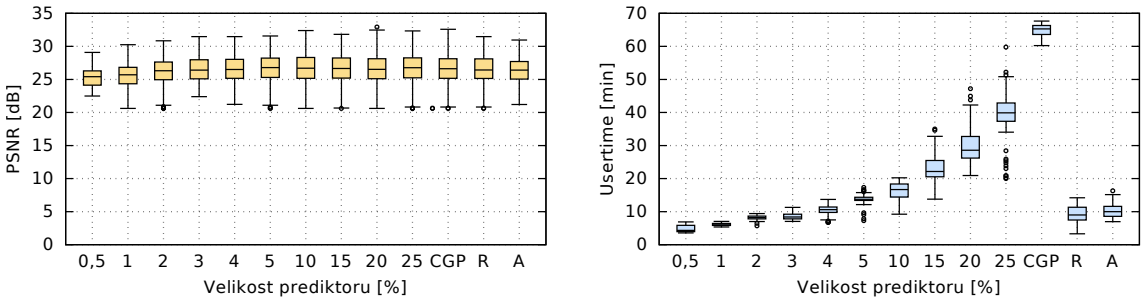
B.2 Výstřelový šum



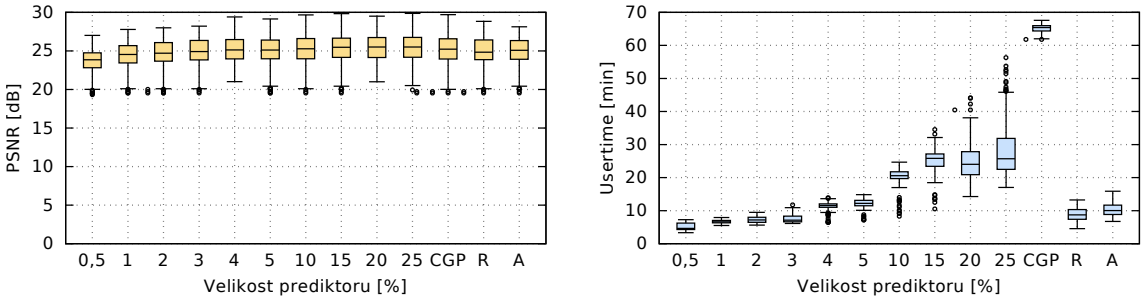
Obrázek B.16. 5% výstřelový (náhodný) šum



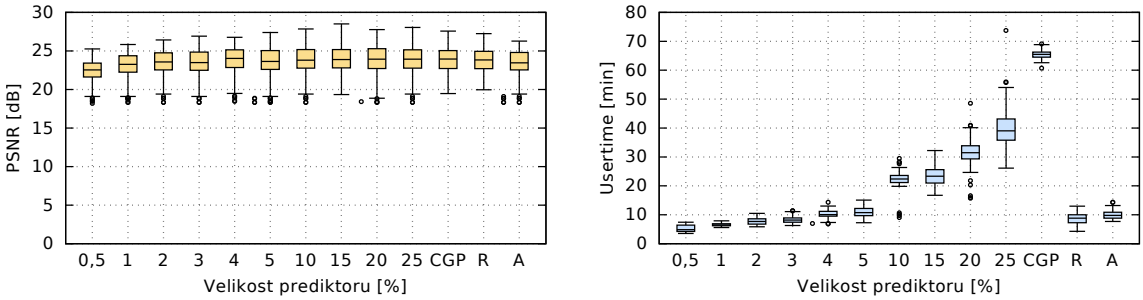
Obrázek B.17. 10% výstřelový (náhodný) šum



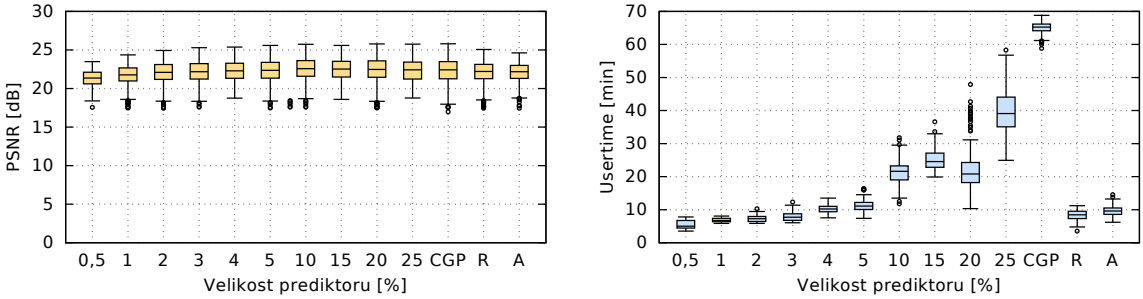
Obrázek B.18. 15% výstřelový (náhodný) šum



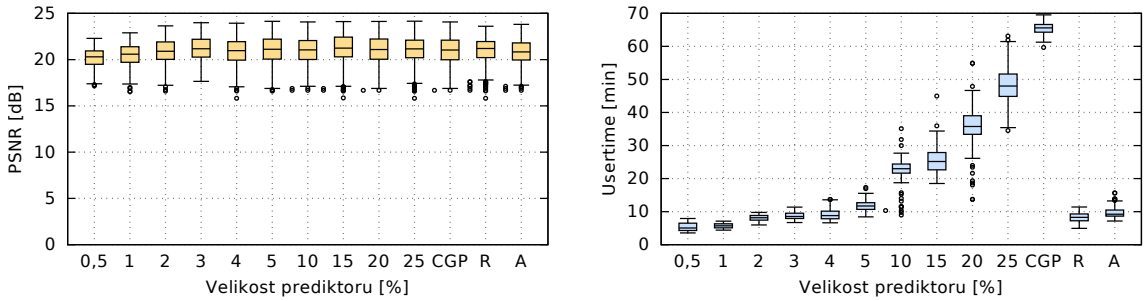
Obrázek B.19. 20% výstřelový (náhodný) šum



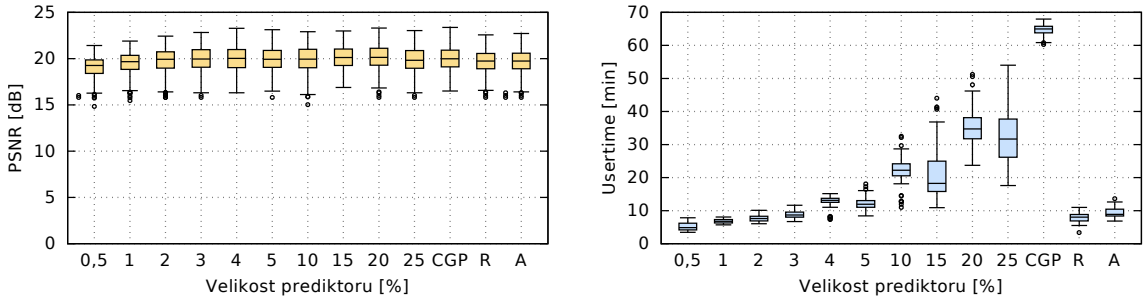
Obrázek B.20. 25% výstřelový (náhodný) šum



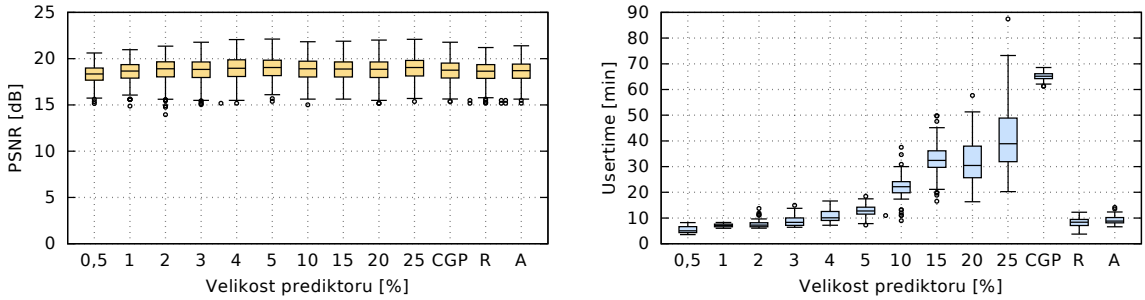
Obrázek B.21. 30% výstřelový (náhodný) šum



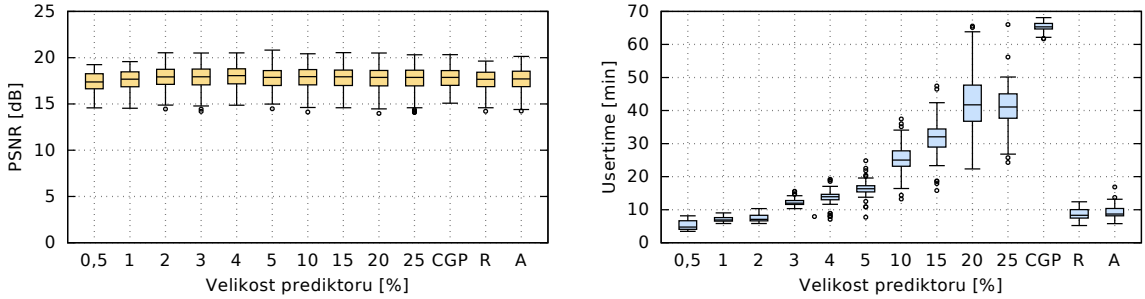
Obrázek B.22. 35% výstřelový (náhodný) šum



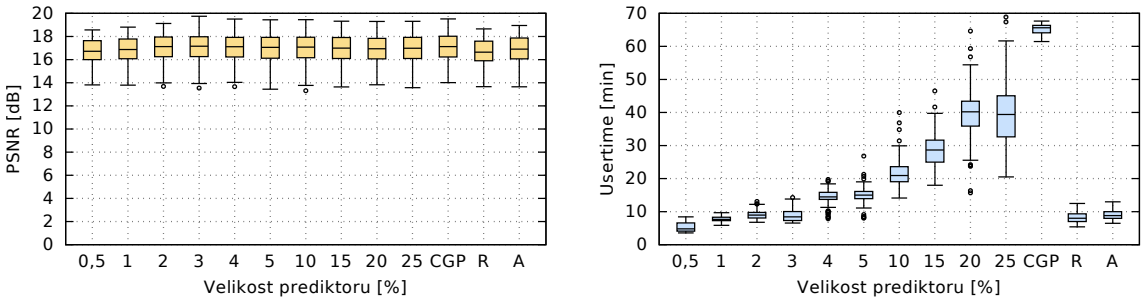
Obrázek B.23. 40% výstřelový (náhodný) šum



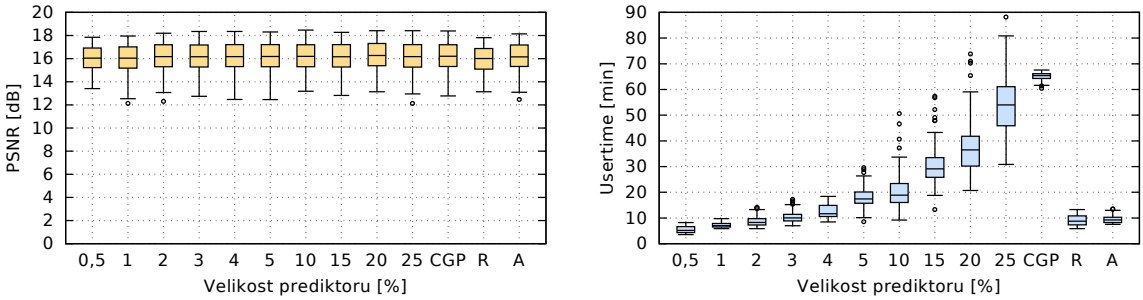
Obrázek B.24. 45% výstřelový (náhodný) šum



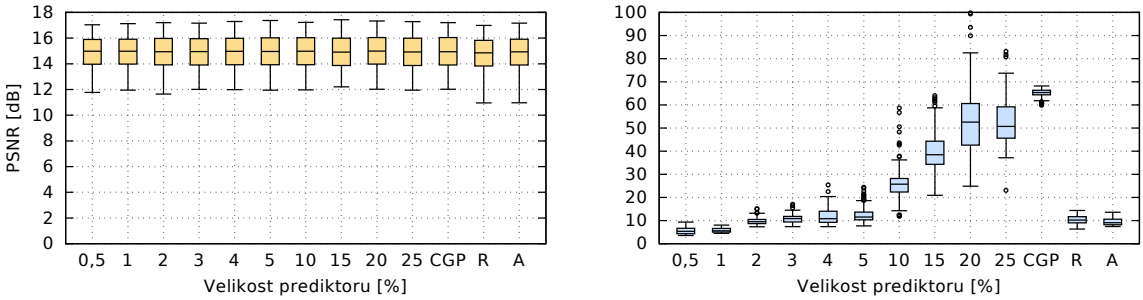
Obrázek B.25. 50% výstřelový (náhodný) šum



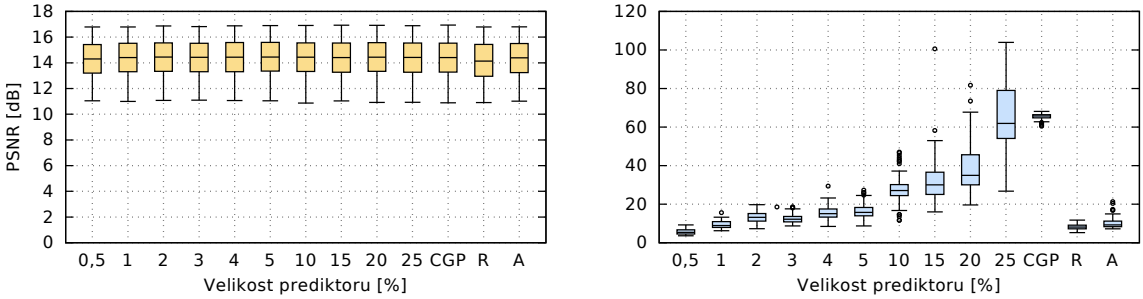
Obrázek B.26. 55% výstřelový (náhodný) šum



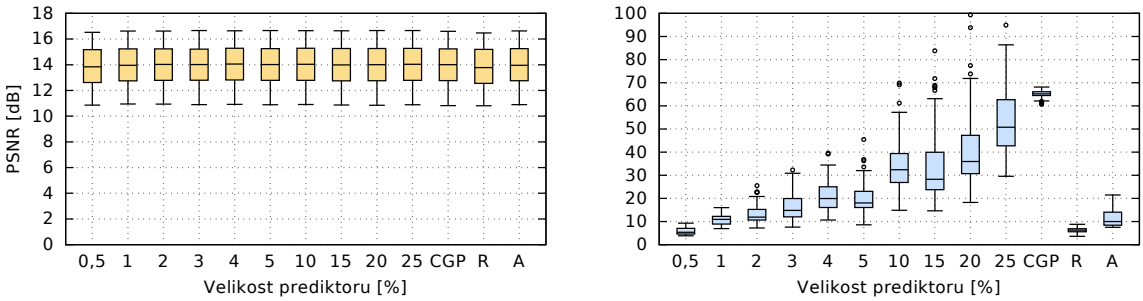
Obrázek B.27. 60% výstřelový (náhodný) šum



Obrázek B.28. 70% výstřelový (náhodný) šum

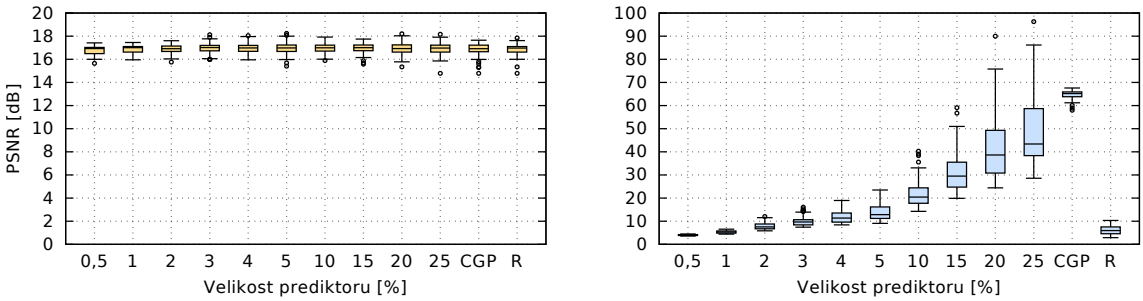


Obrázek B.29. 75% výstřelový (náhodný) šum



Obrázek B.30. 80% výstřelový (náhodný) šum

B.3 Detektor hran



Obrázek B.31. Detektor hran