VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

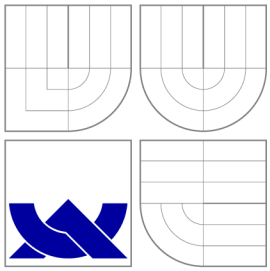
KOEVOLUCE OBRAZOVÝCH FILTRŮ A PREDIKTORŮ FITNESS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

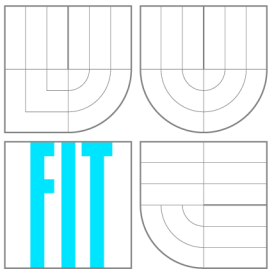
AUTOR PRÁCE
AUTHOR

Bc. JAKUB TREFILÍK

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOEVOLUCE OBRAZOVÝCH FILTRŮ A PREDIKTORŮ FITNESS

COEVOLUTION OF IMAGE FILTERS AND FITNESS PREDICTORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB TREFILÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAELA ŠIKULOVÁ

BRNO 2015

Abstrakt

Tato práce se zabývá využitím principů koevoluce pro návrh obrazových filtrů. Evoluční algoritmy se pro vývoj obrazových filtrů ukazují jako velmi výhodná metoda. Použitím koevoluce prediktorů fitness vnášíme do evolučního návrhu procesy, které vzájemným ovlivňováním populace kandidátních filtrů s populací prediktorů fitness dokáží zrychlit konvergenci řešení. Prediktor fitness je malá podmnožina množiny trénovacích vektorů a používá se k přibližnému určení fitness kandidátních filtrů. V této práci je pro evoluci prediktorů fitness využito nepřímé kódování, které reprezentuje matematický výraz, pomocí něhož jsou vybírány trénovací vektory použité pro vyhodnocení fitness kandidátních filtrů. Tento přístup byl experimentálně vyhodnocen v úloze evolučního návrhu náhodného impulzního šumu a šumu typu sůl a pepř pro různé intenzity šumu a také v úloze návrhu detektoru hran. Ukázalo se, že pomocí tohoto přístupu prediktory fitness přizpůsobují počet použitých trénovacích vektorů pro vyhodnocení kandidátního filtru souběžně s řešením úlohy a tím snižují výpočetní náročnost evolučního návrhu obrazových filtrů.

Abstract

This thesis deals with employing coevolutionary principles to the image filter design. Evolutionary algorithms are very advisable method for image filter design. Using coevolution, we can add the processes, which can accelerate the convergence by interactions of candidate filters population with population of fitness predictors. Fitness predictor is a small subset of the training set and it is used to approximate the fitness of the candidate solutions. In this thesis, indirect encoding is used for predictors evolution. This encoding represents a mathematical expression, which selects training vectors for candidate filters fitness prediction. This approach was experimentally evaluated in the task of image filters for various intensity of random impulse and salt and pepper noise design and the design of the edge detectors. It was shown, that this approach leads to adapting the number of target objective vectors for a particular task, which leads to computational complexity reduction.

Klíčová slova

Evoluční algoritmy, obrazové filtry, kartézské genetické programování, koevoluční algoritmy, predikce fitness.

Keywords

Evolutionary algorithms, image filters, cartesian genetic programming, coevolutionary algorithms, fitness prediction.

Citace

Jakub Trefilík: Koevoluce obrazových filtrů a prediktorů fitness, diplomová práce, Brno, FIT VUT v Brně, 2015

Koevoluce obrazových filtrů a prediktorů fitness

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michaely Šikulové.

.....
Jakub Trefilík
26. května 2015

Poděkování

Děkuji Michaele Šikulové za vstřícnost, cenné rady a připomínky a přátelské, ale přesto odborné vedení mé práce. Děkuji také Radkovi Hrabáčkovi a Jiřímu Hulvovi za ochotu podělit se o dřívější zkušenosti při řešení podobných problémů. Děkuji IT4Innovations za výpočetní prostředky poskytnuté v rámci projektů Centrum excelence IT4Innovations (CZ.1.05/1.1.00/02.0070).

Obsah

Kapitola 1

Úvod

Ve světě technologií, a informační technologie nevyjímaje, obecně platí pravidlo, proč vymýšlet něco, co už vymyšlené je, je to ověřeno a výborně to funguje. S touto nebo podobnou myšlenkou se mohla začít éra biologií inspirovaného počítání. Existuje mnoho variant programování neuronových sítí, můžeme vidět umělou inteligenci inspirovanou uvažováním a chováním lidí i zvířat a prostor zde nalézá také genetika. Pro tuto práci je nejdůležitější poslední zmíněná forma softwarové interpretace biologií inspirovaných procesů, konkrétně evoluční algoritmy.

Kartézské genetické programování je forma evolučních algoritmů, která se používá pro automatizovaný návrh počítačových programů. Ukazuje se jako velmi úspěšná metoda v mnoha doménách, mezi které patří i evoluční návrh obrazových filtrů. Obrazové filtry jsou softwarové nástroje, pomocí kterých lze obecně upravovat vlastnosti obrázků, např. je rekonstruovat, čili opravovat pixely nějak poškozené. Při trénování obrazového filtru se využívá dvojice – poškozený obrázek pro vstup filtru a odpovídající originál pro porovnání s výstupem, který je filtrem produkován. Na vstup filtru je například přivedeno devítiokolí filtrovaného pixelu a výstup filtru je porovnán s odpovídajícím pixelem nepoškozeného obrázku. Filtrace a porovnání je pak provedeno pro všechny pixely a jejich okolí v obrázku. Jedna taková n-tice (složená z devítiokolí pixelu a referenčního výstupu, v takovém případě n=10) se v evolučních algoritmech nazývá trénovací vektor.

Trénovacích vektorů je obvykle příliš mnoho a jejich vyhodnocení je tak časově nejnáročnější částí evolučního návrhu. Koevoluce vybírá podmnožiny trénovacích vektorů takové, které při co nejmenší velikosti dokáží co nejlépe reprezentovat celý trénovací obrázek tak, že při jejich použití je kvalita filtru co nejpřesněji vyčíslena. Podmnožinu trénovacích vektorů určenou k predikci fitness pak nazýváme prediktorem fitness. Tato práce se zaměřuje na prediktory nepřímo kódované, jejichž délka není předem určena a v průběhu návrhu se mění, v nejlepším případě podle aktuálního průběhu evoluce.

Obsah kapitoly 2 je členěn od obecnějších forem evolučních algoritmů po specializovanější, kdy v podkapitolách 2.1, 2.2 a 2.3 přejdeme od genetických algoritmů přes předchůdce kartézského genetického programování až přímo k němu. Na tento koncept naváže podkapitola 2.4 vysvětlením koevolučních algoritmů. Ukázky dříve řešených problémů ukáží míru, do které koevoluce ovlivnila svět biologií inspirovaného počítání. Podkapitolou 2.6 se začne text orientovat na řešenou problematiku tím, že vysvětlí některé dosud známé metody evolučního návrhu obrazových filtrů. Užití takto nabitých teoretických základů proběhne v kapitole 3 za účelem navrhnout vlastní řešení, které koresponduje se zadáním diplomové práce. Praktický způsob implementace se zaměřením na zajímavé prvky naleznete v kapitole 4. Statistická data vzniklá rozsáhlou experimentální činností v úloze evolučního návrhu

filtrů pro redukci impulzního šumu náhodného a typu pepř a sůl a dále pak detektoru hran, zobrazí a popíše kapitola 5, která zároveň shrne získané poznatky o koevolučním návrhu obrazových filtrů s nepřímo kódovanými prediktory fitness.

Kapitola 2

Evoluční algoritmy

Evoluční výpočetní techniky jsou nedeterministickým způsobem prohledávání založeném na aspektech Darwinovy teorie o dědičnosti a přirozeném výběru. Jednoduše řečeno, je zde vždy vybrána reprezentace problému vhodná pro zpracování počítačem pro zakódování aspektů potenciálního řešení a vložena do jedinců. Za pomocí genetikou inspirovaných operátorů rekombinace a výběrem nejlepšího jedince se s trochou štěstí povede během evoluce navrhnout řešení ve formě nejlepšího jedince [7].

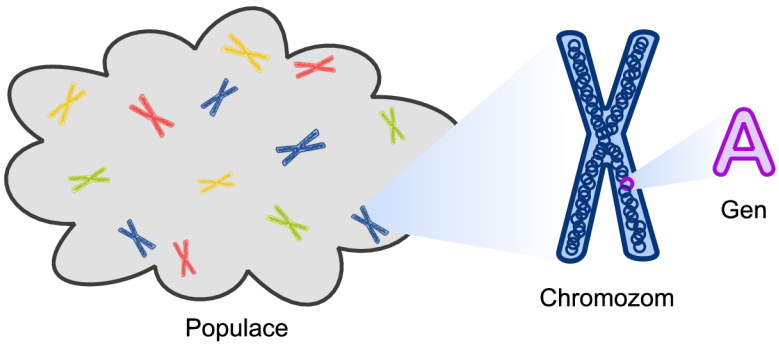
Následující text čtenáře seznámí s obecným principem *evolučních algoritmů* (*evolution algorithm, EA*) a po té v podkapitolách 2.1 resp. 2.2 se specifickými technikami evolučních algoritmů, jimiž jsou *genetické programování* (*genetic programming, GP*) a *kartézské genetické programování* (*cartesian genetic programming, CGP*) .

Mnoho výpočetních problémů lze transformovat na problém nalezení řetězce symbolů. V takovém případě můžeme použít evoluční počítání. Před tím, než si vysvětlíme princip evolučních algoritmů, budou vysvětleny určité pojmy, jejichž znalost bude další text předpokládat. Význam pojmů je definován tak, jak je vysvětlil J. Miller ve své knize [6].

Pojmy

Gen je nositel informace. Nabývá jedné hodnoty z konečné množiny nad definovanou abecedou.

Chromozom se skládá z pevného počtu genů. Kóduje jedno řešení v prohledávaném prostoru.



Obrázek 2.1: Úvodní pojmy k evolučním algoritmům.

Genotyp nebo také **jedinec** se může skládat z více chromozomů.

Populace je tvořena konečným počtem jedinců.

Fitness funkce přiřazuje jedinci hodnotu **fitness**, která udává jeho kvalitu. Díky fitness funkci je nad jedinci v populaci definována relace uspořádání.

Evaluace neboli ohodnocení znamená výpočet hodnoty fitness jedince.

Selekce je označení pro proces výběru množiny jedinců k reprodukci. Podporuje konvergenci řešení.

Křížení a mutace jsou dva nejpoužívanější *rekombinační operátory*. Křížení použije již navržená řešení a určitým, předem daným způsobem, je zkombinuje. Mutace v chromozomu zcela náhodně změní hodnotu náhodného genu. Tím přispěje k diverzitě populace a může odbourat předčasnou konvergenci a uváznutí v lokálním optimu řešení úlohy.

Princip EA

Nyní si pojďme princip výpočtu evolučního počítání vysvětlit na praktickém příkladě. Uvidíme konkrétně, jak může vypadat aplikace předchozích pojmů a taky si budeme všímat nejednoznačnosti použití jednotlivých operátorů v závislosti na právě řešeném problému. Velmi známý je *problém obchodního cestujícího* (*Travelling Salesman Problem, TSP*). Cílem TSP je nalézt nejkratší cestu mezi definovanou množinou měst tak, aby každé město bylo navštíveno právě jednou. Cesta začíná i končí ve stejném městě. Předpokládejme, že města jsou označena symboly $C_1, C_2 ... C_n$. Tyto symboly jsou v řešeném případě hodnoty, kterých mohou nabývat geny. Každý jedinec, reprezentující možné řešení, bude permutace takto definovaných genů. Jiná řešení nemá smysl uvažovat.

V prvním kroku evolučního algoritmu se vytvoří *počáteční populace*, která se skládá z náhodně generovaných jedinců. Následuje evaluace všech jedinců v populaci. V případě TSP je jako hodnota fitness funkce použita celková vzdálenost, kterou cestující na v jedinci zakódované cestě ujede. Po té jsou vybráni jedinci s nejlepší fitness – tj. jedinci kódující co nejkratší cestu mezi městy (často nazývaní *rodičové*) a jsou použiti k vytvoření jedinců *potomků* za použití křížení a mutace.

Operátory křížení a mutace nemůžeme v obecném smyslu generalizovat. V řešeném problému musí probíhat tak, aby výsledný jedinec dával smysl. V TSP musí být v programu zabezpečeno, aby jedinec byl vždy permutací všech měst. Pokud by tomu bylo jinak, nemá smysl takového jedince uvažovat a je potřeba jej opravit.

Ilustrujme si možné řešení tohoto problému. Předpokládejme, že TSP řeší problém pro 6 měst a k dispozici jsou 2 rodiče,

$$p_1 = c_3, c_1, c_4, c_6, c_5, c_2 \text{ a } p_2 = C_6, C_4, C_2, C_1, C_5, C_3.$$

Kdyby byl použit operátor křížení bez fáze oprav a vzaly by se první 3 geny od p_1 a poslední 3 od p_2, vznikl by jedinec $c_3, c_1, c_4, C_6, C_4, C_2$, který skutečně není možným řešením a nemá smysl jej uvažovat a ohodnocovat. Pro potřeby zachování permutace genů v chromozomu jedince máme k dispozici tzv. *merging crossover*, při kterém je nejdříve náhodným výběrem genů vytvořen pomocný chromozom o dvojnásobné délce, například

$$c_3, C_6, C_4, c_1, c_4, c_6, C_2, C_1, c_5, C_5, C_3, c_2$$

Tento pomocný chromozom je v následujícím kroku čten zleva doprava a jsou z něj vytvořeni 2 potomci tím způsobem, že pokud nastane kolize ve formě genu stejné hodnoty v jednom chromozomu, je tento gen umístěn ve chromozomu druhého potomka:

$$d_1 = c_3, C_6, C_4, c_1, C_2, c_5 \text{ a } d_2 = c_4, c_6, C_1 C_5, C_3, c_2.$$

Proces mutace musí být ze stejného důvodu také upraven. S použitím obecného konceptu, jak byl definován v úvodu kapitoly by z d_1 mohl vzniknout např. $d_1 = c_3, C_6, C_2, c_1, C_2, c_5$. Jednoduchý způsob úpravy tohoto genetického operátoru je náhodný výběr dvou genů v chromozomu a výměna jejich pozic:

$$d_1 = c_3, \mathbf{C_6}, C_4, c_1, C_2, \mathbf{c_5} \rightarrow \bar{d_1} = c_3, \mathbf{c_5}, C_4, c_1, C_2, \mathbf{C_6}.$$

Z vybraných rodičů a jejich potomků je vytvořena nová generace. Volitelně se v nové generaci mohou zachovat rodiče v nezměněné formě. Tento jev se nazývá *elitismus*.

Algoritmus 1: Evoluční algoritmus

Generuj počáteční populati velikosti p;
Nastav počet generací na $g = 0$;
repeat
  Vypočítej hodnotu fintess všech jedinců v populaci;
  Vyber nejkvalitnější jedince jako rodiče;
  Pomocí genetického operátoru křížení vytvoř potomky z vybraných rodičů;
  Pomocí genetického operátoru mutace uprav chromozomy některých potomků;
  Z mutovaných rodičů a potomků vytvoř novou populaci;
  [elitismus] Volitelně vložíme do takto vzniklé populace nezměněné rodiče;
  $g = g + 1$;
until $g =$ *nastavený celkový počet generací nebo hodnota fitness funkce některého z jedinců je akceptovatelná*;

2.1 Genetické programování

Genetické programování je speciální případ EA, které se orientuje na automatický návrh spustitelných struktur, tj. počítačových programů. Za zakladatele GP ve formě, jak ho známe dnes, je považován John Koza, který ho představil v roce 1992 publikací *Genetic Programming: On the Programming of Computers by Natural Selection*. Zde Koza publikoval obsáhlou práci o evoluci počítačových programů ve formě výrazů jazyka LISP[1].

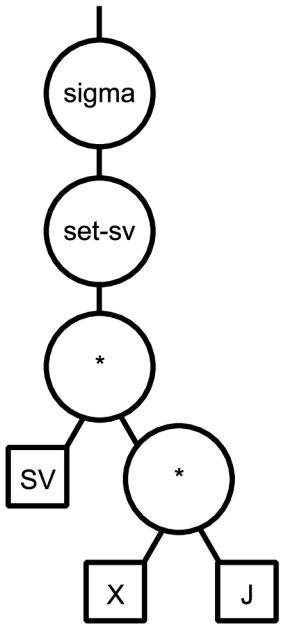
Obrázek 2.2 ukazuje stromovou reprezentaci výrazu $(SIGMA(SET\_SV(*(\%XJ))))$. Dojde k tomu, že bude navržen S-výraz takový, že bude počítat výsledek diferenciální rovnice 2.1 při nastavených počátečních podmínkách $y_{initial} = 2.718$ a $x_{initial} = 1.0$.

$$\frac{dy}{dx} - y = 0 \tag{2.1}$$

Funkce $SIGMA$ má jeden argument a provede součet s hodnotou v registru SV. Součet je definován jako inkrementace indexace proměnné J. Funkce % provádí dělení[2] svých dvou vstupních argumentů. SET_SV nastaví registr SV na hodnotu svého argumentu. SV a J

[1] LISP je v dnešní době velmi využíván v oboru Umělé inteligence. Programy v LISPu jsou reprezentovány tzv. S-výrazy seznamů symbolů uzavřených v závorkách.

[2] % je operace tzv *protection divide*, u kterého je ošetřeno dělení nulou, typicky $x\%0 = 1$.



Obrázek 2.2: Stromová reprezentace S-výrazu $(SIGMA(SET\_SV(*(\%XJ))))$.

mají stejné hodnoty. Program sčítá po sobě jdoucí argumenty x, $\frac{x^2}{2}$, $\frac{x^3}{3!}$, které aproximují řešení rovnice 2.1.

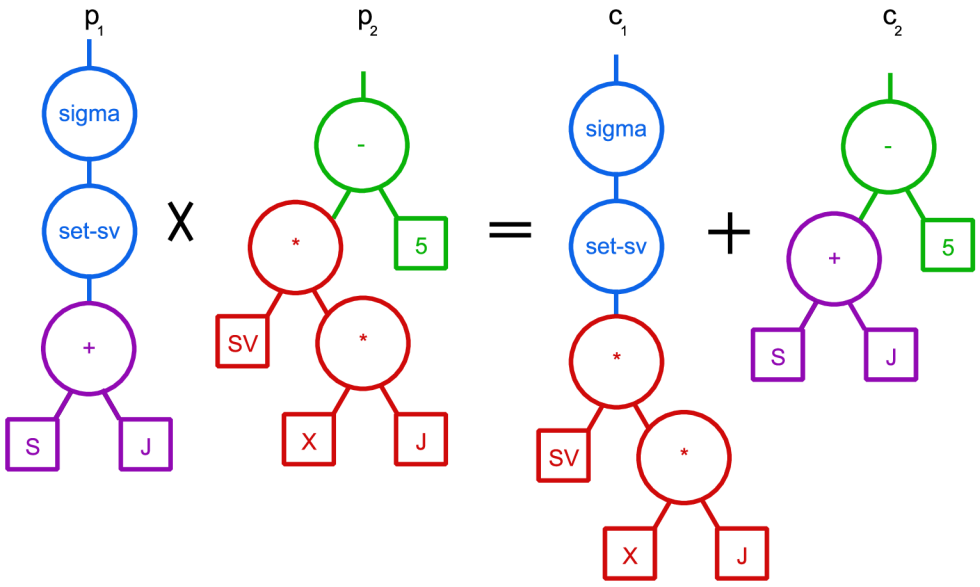
Se stromovou reprezentací spustitelné struktury pracuje GP. V prvním kroku je vygenerován počáteční jedinec (strom), který musí být sestrojen s ohledem na syntaxi programovacího jazyka. Křížení v průběhu evoluce probíhá tak, jak je ilustrováno na obrázku 2.3, tj. jsou vybrány podstromy dvou jedinců a tyto podstromy jsou zaměněny. Mutace mění zvolený podstrom za náhodně vygenerovaný.

2.2 Genetické programování využívající obecnější formy grafu

Na rozdíl od stromů, u kterých vždy mezi dvěma uzly existuje právě jedna cesta, obecné grafy umožňují vytvořit více cest mezi páry uzlů. Uvažujeme-li, že každý uzel nese nějakou funkci, reprezentace funkce ve formě grafu je mnohem kompaktnější vzhledem k tomu, že umožňuje znovupoužití dříve vypočtených podgrafů. Obecně můžeme říct, že grafová reprezentace je pro množství oborů výpočetní techniky použitelná a atraktivní. Příkladem použití obecných grafů jako výpočetních modelů mohou být i *neuronové sítě*. Alternativy GP, kterým se věnuji v této kapitole, popsal Miller v knize [6].

První, kdo v oblasti EA použil kódování založené na grafech, tzv. *Kartézskou mřížku*, byl Sushil J. Louis v roce 1990. Ten v technické zprávě použil binární genotyp, který kóduje síť logických hradel. Hradla v řešeném problému mohou v každém sloupci být připojeny k hradlům v předchozím sloupci.

Riccardo Poli, inspirovaný neuronovými sítěmi, navrhl tzv. *paralelní distribuované GP* (*parallel distributed GP, PDGP*). PDGP v principu dovoluje navrhovat standardní programy založené na stromech, logické sítě, neuronové sítě a konečné automaty. Jedná se o GP založený na grafech. PDGP nově vyžaduje definici množiny spojení (set of links), která definuje pravidla, jak mohou být uzly spojeny. Například v neuronových sítích mohou



Obrázek 2.3: Křížením chromozomu p_1 a p_2 vzniknou jedinci c_1 a c_2.

být uzly propojeny navzájem přes celou šířku grafu. Naopak, například v grafu na obrázku 2.4 je povoleno propojovat uzly s uzly v sousední vrstvě. Při implementaci PDGP je program reprezentován jako pole se stejnou topologií, jako je mřížka. Každý uzel obsahuje identifikátor funkce a horizontální posunutí uzlů v předchozí vrstvě. Funkce nebo vstupní uzly jsou spojeny se všemi uzly v mřížce, i když nejsou odkazovány – v takovém případě nejsou použity a říkáme jim introny.

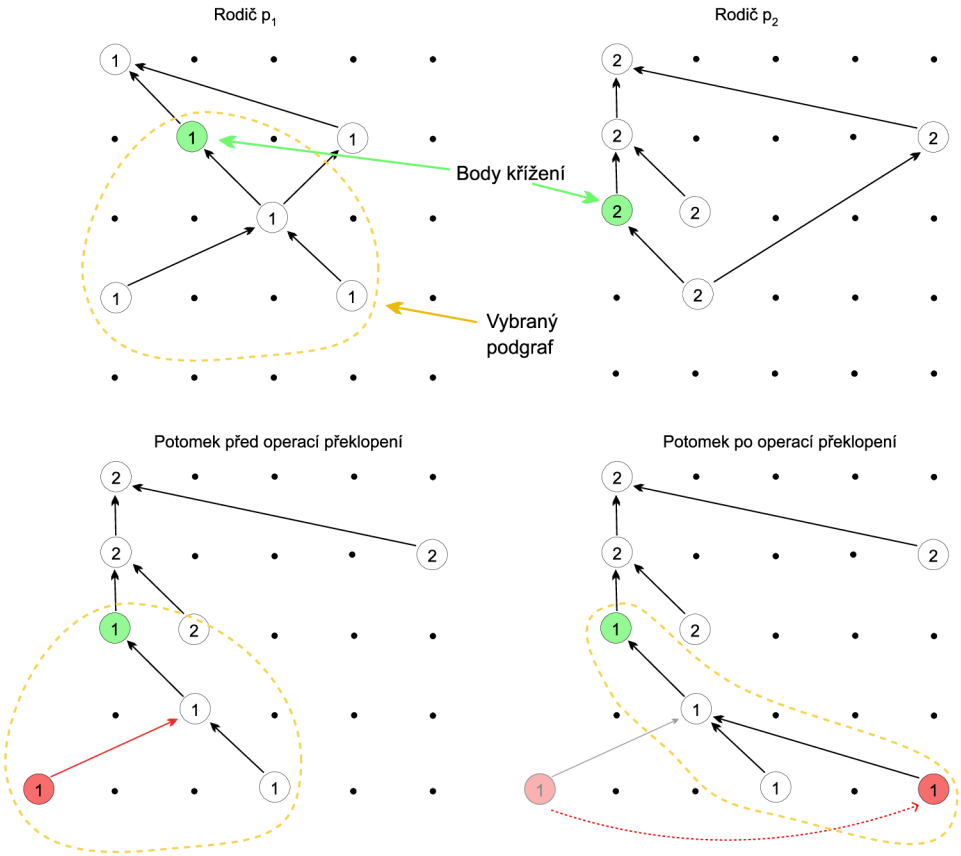
Genetický operátor křížení je v PDGP realizován jako výměna podgrafu aktivních uzlů (subgraph active-active node, SAAN). Jedná se o generalizovaný způsob křížení stromových struktur. Průběh je ilustrován na obrázku 2.4 a probíhá následujícím způsobem:

1. Vyber 2 náhodné aktivní uzly jako rodiče (bodu křížení).
2. U prvního bodu křížení vyber takový podgraf aktivních uzlů, který vede k výpočtu jeho vstupních hodnot.
3. Takto vybraný podgraf vlož do druhého bodu křížení. V případě, že některá x-ová souřadnice uzlu ve vloženém podgrafu překročila maximální povolenou šířku grafu, proveď změnu souřadnic překlopením přes mřížku grafu.

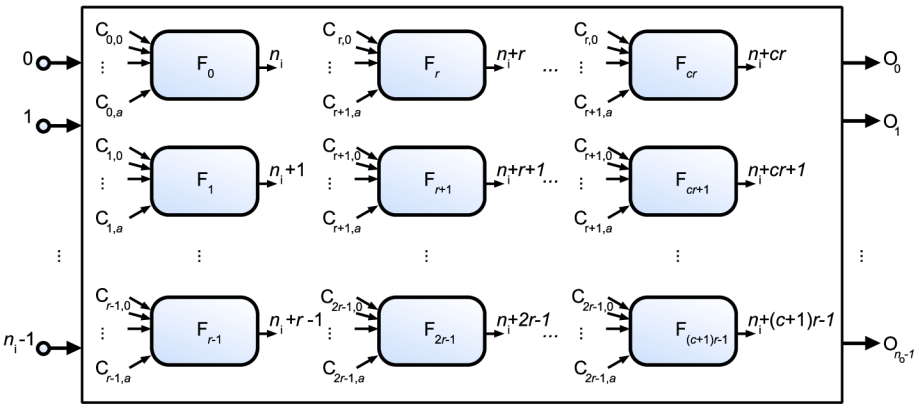
Mutaci Poli dovolil v PDGP použít dvěma způsoby. Jednalo se o tzv. *globální mutaci*, která vkládala náhodně generovaný podgraf do existujícího programu. *Mutace spojení* nejprve vybere funkční uzel, poté jeho vstupní spojení a nakonec pozmění offset související s tímto spojením.

2.3 Kartézské genetické programování

Kartézské genetické programování (*Cartesian genetic Programming, CGP*) vzniklo ze způsobu návrhu digitálních obvodů, které publikoval Julian F. Miller v roce 1997. Termín CGP byl ovšem poprvé použit roku 1999 jako speciální forma GP. Následující text však vychází z novější publikace, z knihy [5], kterou Miller vydal v roce 2011.



Obrázek 2.4: Příklad výměny podgrafu aktivních uzlů (SAAN).



Obrázek 2.5: Obecná forma CGP. Vysvětlení symbolů užitých v obrázku je v tabulce 2.1.

V CGP je program reprezentován ve formě acyklického orientovaného grafu. Graf lze vždy zobrazit dvourozměrnou mřížkou výpočetních uzlů v kartézské soustavě souřadnic – odtud plyne název metody. Geny, které tvoří genotyp CGP jsou přirozená čísla, která reprezentují:

1. odkud uzel přijímá svá vstupní data,
2. jakou operaci/funkci uzel vykonává, tzv. *funkční gen* a
3. kde se nachází výstupní data celého obvodu.

Počet genů kódující vstupy uzlu závisí na aritě operace, kterou vykonává. Funkční uzel je vždy jen jeden a představuje adresu do look-up tabulky funkcí, jejíž obsah závisí na řešeném problému.

Nekódující uzel (a jeho gen) je ten, který není použit k výpočtu výstupních dat – takovéto uzly jsou při dekódování genotypu ignorovány. Říkáme, že program dekóduje fenotyp z genotypu. Rozdíl je takový, že genotyp je pevné délky, zatímco délka fenotypu je proměnná. Pohybuje se od 0 v případě, kdy je výstup obvodu připojen přímo na vstupy CGP až po délku celého genotypu v případě, že jsou pro výpočet použity všechny uzly grafu.

Každý uzel bere pro svůj vstup buď výstup některého z předchozích uzlů, nebo z tzv. *terminálních uzlů*, neboli vstupů programu. Adresa terminálních uzlů začíná na hodnotě 0 a sekvenčně pokračuje do $n_1 - 1$, kde n_1 je počet vstupů programu. Ostatní parametry CGP shrnuje obrázek 2.5 a tabulka 2.1.

Dosud nezmíněným parametrem CGP je parametr se zaužívaným označením *l-back*. Nastavením hodnoty *l-back* uživatel ovlivní konektivitu grafu. Představuje, odkud mohou uzly v jednotlivých sloupcích přijímat svá vstupní data. Pokud má CGP nastaveno *l-back*= 1, znamená to, že může být připojen pouze na uzly umístěné bezprostředně v předchozím sloupci. Maximální hodnotou, která zajišťuje nejvyšší konektivitu je *l-back*= c, kdy mohou být uzly v každém sloupci propojeny s kterýmkoli uzlem v některém z předchozích sloupců. K terminálním uzlům mohou být uzly připojeny bez ohledu na *l-back*.

Výše bylo zmíněno, že v CGP, na rozdíl od GP je, kvůli nekódujícím uzlům, rozdíl mezi genotypem a fenotypem. Obrázek 2.6 představuje proces dekódování fenotypu z genotypu. V tomto příkladu lze vidět, že dekódování začíná od výstupního genu. Dopředným způsobem

Proměnná	Význam
n_i	Počet vstupů
n_o	Počet výstupů
r	Počet řádku kartézské mřížky
c	Počet sloupců kartézské mřížky
a	Arita operace, kterou uzel provádí
f_i	Počet funkcí ve funkční look-up tabulce
a	Arita operace, kterou uzel provádí
λ	Počet mutantů rodiče použité v ES[3]

Tabulka 2.1: Parametry CGP.

(zprava doleva) jsou zjištěny aktivní uzly. Uzly neaktivní nekódují žádnou část výpočtu, proto jsou ve výsledném fenotypu vynechány.

Z příkladu na obrázku 2.6 vyplývá určitá redundance – přítomnost neaktivních uzlů, které nemají vliv na hodnotu fitness funkce. Neaktivita některých uzlů v genotypu však není jediný druh redundance, se kterým se v CGP setkáváme. Počet vstupů do každého uzlu (tzn. i počet genů kódující vstupy) podléhá funkci s největší aritou. Nastává pak situace, kdy uzel nepoužije všechny vstupy, které jsou k němu připojeny (např. unární logická funkce NOT, ke které by byly připojeny 2 vstupy použije pouze první vstup). Funkce určité skupiny uzlů může být v některých případech buďto výrazně zjednodušena, popř. vynechána úplně (např. $C = A + B - A$ můžeme nahradit $C = B$).

2.4 Koevoluční algoritmy

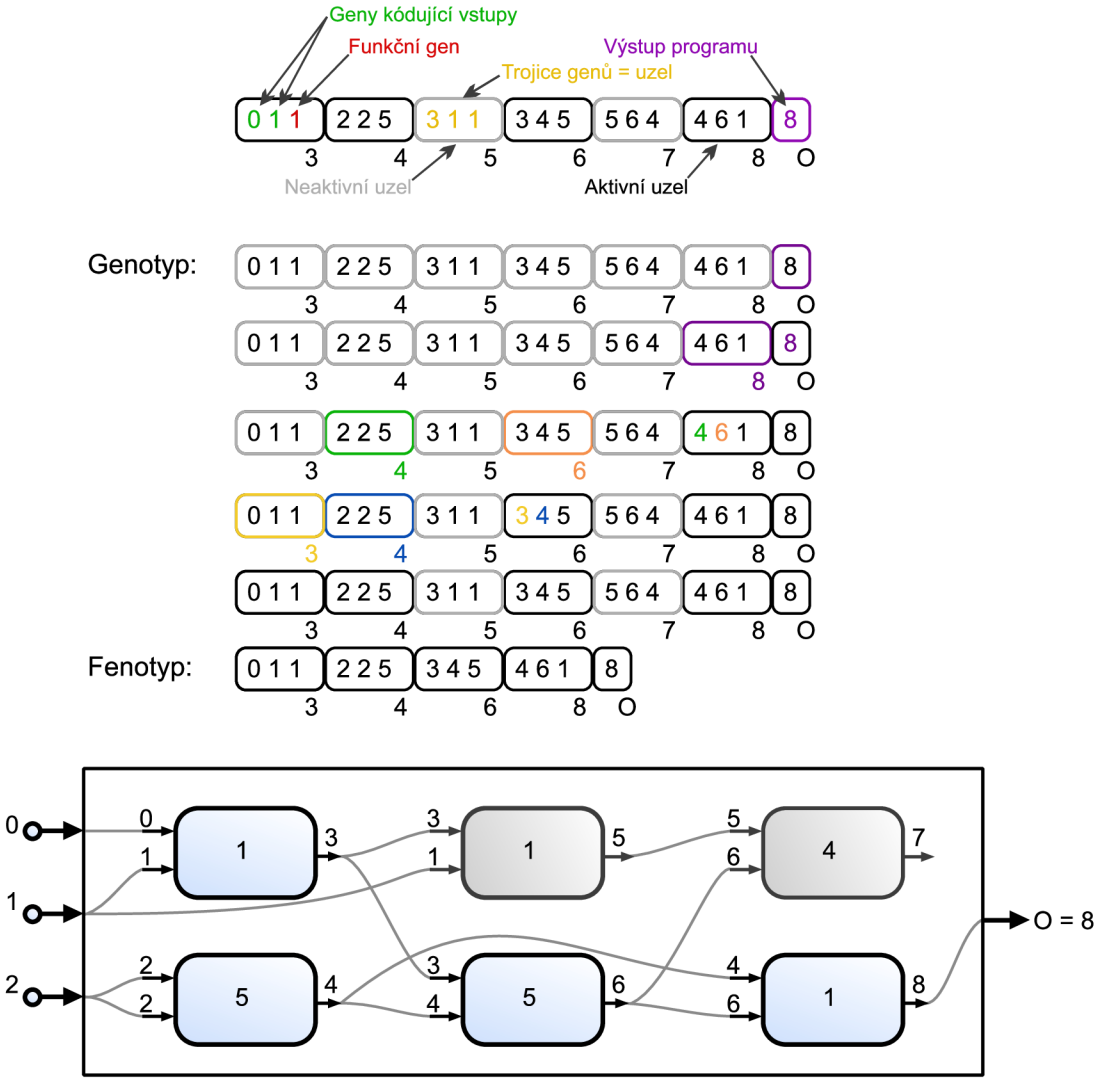
Koevoluční algoritmy (*Coevolutionary algorithms, CoEA*) vznikly, stejně jako evoluční algoritmy, na základě Darwinovy teorie o dědičnosti a přirozeném výběru. Berou ovšem navíc v úvahu existenci více souběžně vyvíjejících se druhů, které se v reálu ovlivňují a adaptují své schopnosti vzhledem k okolí. Úvod do problematiky CoEA vč. pojmů shrnují Elena Popovici a další autoři v knize Handbook of natural computing kapitolou [7].

CoEA, na rozdíl od EA, nepoužívají fitness funkci ve formě zobrazení $f : G \longrightarrow \mathbb{R}$, které přiřadí hodnotu reálného čísla každému genotypu $g \in G$. Tento typ funkce, kde pro každé dva genotypy existuje uspořádání mezi jejich hodnotami fitness funkce, se nazývá *objektivní fitness funkce*. Oproti tomu, v CoEA se setkáváme s tzv. *subjektivními fitness funkcemi*, kdy jsou dva jedinci porovnáni na základě výsledku interakce s ostatními jedinci. Rozdílný způsob evaluace jedinců vytváří základní rozdíly oproti tradičním evolučním výpočetním technikám. Nejzásadnějším rozdílem je, že ohodnocení dvou jedinců se během času mění – v jednu chvíli budeme věřit, že jedinec $g_1 \in G$ je lepší než $g_2 \in G$ a v průběhu evoluce se může změnou okolních jedinců stát, že g_2 je lepší než g_1.

Pojmy

Vzhledem k rozdílům, z nichž jeden byl nastíněn v úvodním odstavci, nastává situace, kdy určité pojmy chápeme jinak oproti klasickým EA. Také zde vznikají pojmy nové. Proto je potřeba seznámit se z následujícími pojmy v kontextu koevolučních algoritmů.

Jedinci v CoEA také představují subjekty k selekci podléhající genetikou inspirovaným



Obrázek 2.6: Proces dekódování fenotypu z genotypu s vysvětlivkou a vykreslení genotypu v grafické formě.

operátorům. Rozdíl je v tom, že jedinec vždy nepředstavuje reprezentaci potenciálního řešení problému. Vyskytuje se zde v rolích různých částí řešení nebo jako testy pomáhající zvýšit kvalitu řešení. Proto se pro jedince někdy užívá pojem *komponenta*.

Populace představuje množinu jedinců. V CoEA ovšem neexistuje jen jedna populace, jak je tomu v klasických EA. Vyskytuje se zde vždy více různorodých populací. Jedinci v jisté populaci jsou od jedinců v jiné populaci odděleni. Při představě populace jako jistého prostorového modelu můžeme říci, že se mezi jednotlivými populacemi vyskytuje bariéra. Každý typ jedince by se měl vyskytovat nejméně v jedné populaci. Častý přístup je takový, že v koevoluci existuje vždy jedna populace jedinců každého typu (populace jednotlivých komponent). Jedinci jsou sice vyvíjeni v oddělených populacích, existuje mezi nimi však částečná interakce sloužící k evaluaci jedinců v jiné populaci.

Archivy představují jiný druh množiny jedinců, jakýsi druh paměti prohledávání. Archivy umožňují oddělit průzkum od evaluace a/nebo od reprezentace řešení. V archivu se nacházejí jedinci z více generací, obvykle se zde nachází konečné nebo potenciální řešení problému. Představují určitou formu elitismu definovanou v EA. Návrat jedince do populace z archivu naopak vnese diverzitu ochraňující prohledávání před uváznutím v lokálním optimu.

Řešení či alespoň potenciální řešení můžeme nalézt buďto ve formě jednoho jedince, který může být vybrán z určité populace, z libovolné populace nebo z archivu. Řešení může být také složeno z více jedinců z jedné nebo několika populací či archivů.

Interakce může probíhat mezi jedinci stejné nebo odlišné populace. Existují různé metody interakce, lišící se v tom, zda spolu jedinci spolupracují (tzn. kooperují) nebo pracují proti sobě (tzn. kolaborují). Dále rozlišujeme interakci soustředěnou na jedince (*individual centric*) a soustředěnou na populaci (*population centric*). Interakce jedinců slouží k výpočtu hodnoty fitness jedince. Nejprve je vybrána množina interakcí a po té proběhne vyhodnocení interakcí, tj. přiřazení fitness. V případě, že jde o jednoduchou interakci, získáme hodnotu přímo.

Agregace musí být použita v případě, kdy nastává interakce mezi skupinami jedinců, tj. v případě, kdy je hodnota fitness určena na základě vyhodnocení množiny interakcí. Výsledkem interakce je množina hodnot, které musí být agregovány do jedné. Ta udává výslednou fitness. Jeden přístup je získat hodnotu prostřednictvím výpočtu z množiny hodnot. Tento způsob má tu výhodu, že můžeme k selekci rodičů použít tradiční postup prezentovaný v EA. Bohužel, různé složky zpracovávané množiny mají různou váhu pro určení kvality jedince, a vždy nemůžeme přímočaře určit, která z nich je do jaké míry důležitá. Alternativou je ponechat hodnotu fitness funkce ve formě n-tice a změnit způsob selekce. N-tice jednotlivých jedinců ve stejné roli musí být porovnatelné, to omezuje interakce, které hodnotu fitness ve formě n-tice generuje.

Komunikace slouží za účelem interakce mezi jedinci v různých populacích. Jednotlivé populace musí poskytovat svůj obsah jiným populacím. Komunikace s sebou nese volby spjaté typicky s jakýmkoli distribuovaným systémem. Koordinace (asynchronní komunikace prostřednictvím sdílené paměti nebo synchronní komunikace používající centralizovaný časovač), tok (paralelní nebo sekvenční) a frekvenci (počet cyklů evoluce mezi jednotlivými událostmi komunikace).

Třídy problémů

V následujícím textu budou uvedeny příklady problémů, na které byla koevoluce s úspěchem aplikována. Ty lze rozdělit do dvou tříd.

Koevoluční algoritmus dvou populací aplikovaný na úlohy založené na testu.
Hillis v roce 1990 [1] použil koevoluci na návrh řadicí sítě. Použil k tomu 2 populace, jedna obsahovala jedince reprezentující řadicí sítě (kandidátní řešení), zatímco druhá reprezentovala množiny neseřazených dat k testování schopností sítí řadit (testy). Jedinci z populace testů jsou ohodnocováni na základě jejich schopnosti odhalit chyby potenciálních řešení. Cílem algoritmu je navrhnout co možná nejmenší řadicí síť, která dokáže seřadit jakákoli vstupní data. Výsledkem byla síť obsahující 61 komparátorů pro 16 vstupních hodnot. To je pouze o jeden komparátor více, než má nejmenší dosud známá řadicí síť. Při použití obyčejného EA byla navržena síť o velikosti 63 komparátorů. Tento přístup se často nazývá soutěživá koevoluce a bývá přirovnáván k situaci predátor – kořist. Tyto dva druhy spolu osídlují stejné teritorium a po miliony let evoluce se jejich schopnosti lovit – chránit se navzájem ovlivňují (se zrychlením predátora dojde k vyhynutí pomalých jedinců populace kořisti, a ti již nemají možnost předávat svůj genetický materiál do nových populací).

Koevoluční vícepopulační algoritmus aplikovaný na kompozičním problému.
Části potenciálního řešení tvoří kooperující tým. Každé potenciální řešení je získáno složením těchto částí v jeden celek, typicky by se každá část měla v řešení vyskytovat nejméně jednou.

Asi nejstarší aplikace na kompoziční problém byla publikována v práci Husbanda a Milla [4] na *problému plánování procesů* (*job-shop scheduling*). Jedinci kódují potenciální půdorysy zahrnující jednotlivé části obchodu. Oddělená populace je použita pro optimalizaci plánování každé části. Další je populace arbitrů, jakýchsi agentů řešících konflikty mezi jednotlivými částmi obchodu. Fitness funkce mimo jiné zahrnuje účetnictví se sdílenými prvky, kterými jsou například čas a prostor. Výsledné plánovače se vyrovnávají s velkou nejistotou, se kterou se problém plánování procesů potýká.

2.5 Koevoluční algoritmy v CGP

Při aplikaci CGP na řešení problému symbolické regrese musí být vyhodnoceny stovky až tisíce případů fitness a to při každém výpočtu fitness funkce. Ke snížení výpočetní náročnosti evaluace slouží techniky, které dovolují fitness funkci aproximovat. Šikulová a Sekanina v článku [14] poprvé představili *koevoluci s prediktory fitness* aplikovanou na CGP, fungující na základě frameworku z článku Schmidta a Lipsona [8] navrženého pro standardní GP. Metoda kombinuje predikci fitness s koevolucí, a odstraňuje tím nevýhody, které s sebou nese *modelování fitness funkce*[4], jakými jsou potřeba naučit model fitness funkce a vliv úrovně aproximace na přesnost. Koevoluce s prediktory fitness užívá 2 populace, které se vyvíjejí souběžně a navzájem ovlivňují kvalitu jedinců druhé populace. Vlastnosti jedinců kandidátních programů mění vlastnosti jedinců v populaci prediktorů fitness a naopak. Prediktory fitness se vyvíjí na principu genetického algoritmu.

[4]Modelování fitness funkce zkracuje čas evaluece tím, že využívá různé stupně náročnosti a propracovanosti fitness funkce v různých etapách výpočtu.

Bylo ukázáno, že s využitím koevoluce v CGP v úloze symbolické regrese je průměrný čas potřebný k nalezení řešení zkrácen $2.03 - 5.45$krát oproti standardnímu CGP. Experimenty byly provedeny na pěti různých úlohách symbolické regrese. Přístup koevoluce v CGP byl ověřen i v úloze evolučního návrhu obrazových filtrů. To je pro CGP obvyklejší úloha než symbolická regrese. Evoluční návrh obrazových filtrů je složitější než symbolická regrese. U symbolické regrese evoluce pracuje se stovkami *trénovacích vektorů* (*Target Objective Vectors, TOVs*), zatímco návrh obrazových filtrů musí vždy zpracovat desítky až stovky tisíc pixelů z množiny TOVs. U obrazových filtrů se ukazuje jako vhodná velikost prediktoru v řádu tisíců trénovacích vektorů. S takovou velikostí prediktorů souvisí *problém škálovatelnosti*. Pokud je chromozom pro genetický algoritmus příliš rozsáhlý, není prohledávání efektivní[10]. Dlouhé chromozomy implikují rozsáhlé prohledávací prostory a ty je obtížné efektivně prohledávat. K řešení tohoto stavu existují různé metody, u obvodů je to např. *evoluce na úrovni funkčních bloků*, která nepracuje s hradly, ale s aplikačně specifickými komponentami různé složitosti, *inkrementální evoluce*, která buduje řešení postupně nebo jiné metody inspirované procesem ontogeneze.

Hulva ve své diplomové práci [3] navrhl nový způsob vývoje prediktorů – na principech symbolické regrese jsou zde vyvíjeny také jedinci z populace prediktorů fitness. Tímto způsobem lze dynamicky měnit velikost trénovací sady během evoluce. Evoluce může sama nalézt optimální velikost trénovací sady nebo ji může měnit po celou dobu běhu symbolické regrese. Zkoumáno bylo pět úloh symbolické regrese, z toho u tří se podařilo nalézt řešení během menšího počtu generací než u dřívější metody se statickou velikostí sady testů.

Rychlejší nalezení řešení vykazoval nově navržený způsob koevoluce pro úlohy s déle trvajícími běhy. U čtyř z pěti testovaných úloh byla pozorována adaptace na zkoumanou úlohu a to buď ve smyslu adaptace na lokální extrémy nebo výběr prediktorů rovnoměrně po celé trénovací sadě.

2.6 Evoluční návrh obrazových filtrů

Přístupy k návrhu obrazových filtrů můžeme rozdělit do dvou kategorií. Filtry jsou navrhovány buďto konvenčně, tzn. na základě nějaké matematické teorie. Příkladem konvenčního řešení může být mediánový filtr, který k výpočtu nové hodnoty každého pixelu v obrázku použije jeho lokální okolí (např. 9 resp. 25 okolí), ze kterého vypočítá medián. Mediánový filtr je použitelný do hodnoty 10 %–20 % zašumění obrázku.

Existuje mnoho přístupů evolučního návrhu obrazových filtrů, které sjednocuje rozdíl oproti konvenčnímu řešení. Funkcionalita evolučně navrženého filtru nemusí být a dokonce nebývá podložena jakýmkoli matematickým odůvodněním. Struktura filtru často nedává expertovi smysl. Blíže se problematice evolučního návrhu filtrů budu věnovat v následujících kapitolách.

2.6.1 Běžný přístup

Obrazový filtr pro odstranění výstřelového šumu typu sůl a pepř (salt-and-papper noise) pracující s devítiokolím je reprezentován logickou funkcí s devíti osmibitovými vstupy a jedním osmibitovým výstupem. Tuto funkci lze v hardware implementovat jako logický obvod. Z předchozích kapitol plyne, že pro evoluční návrh logického obvodu lze s úspěchem použít kartézské genetické programování. Že ani návrh obrazového filtru není výjimkou, to ukázal Sekanina a kolektiv v [9]. Kandidátní filtr je reprezentován mřížkou s parametry $n_i = 9, n_o = 1$ o rozměrech, typicky $n_c = 8$ a $n_r = 4$. Ostatní parametry CGP jsou

$n_a = 2$, *l-back*= 1 a $\lambda = 8$. Každý uzel reprezentuje funkci se dvěma vstupními parametry velikosti 8 bitů a produkuje jeden osmibitový výstup. Funkce, kterou uzel vykonává je jedna z tabulky 2.2.

Funkce	**Význam**	**Funkce**	**Význam**
255	Konstanta	$x \gg 1$	Posuv doprava o 1 bit
x	Identická hodnota	$x \gg 2$	Posuv doprava o 2 bity
$255 - x$	Inverze	$swap(x, y)$	Přehození nibblů
$x \vee y$	Bitová disjunkce	$x + y$	Součet
$\bar{x} \vee y$	Disjunkce s neg. x	$x +^S y$	Součet s ořezáním
$x \wedge y$	Bitová konjunkce	$(x + y) \gg 1$	Průměr
$\overline{x \wedge y}$	Negovaná konjunkce	$max(x, y)$	Maximum
$x \bigoplus y$	Exkluzivní disjunkce	$min(x, y)$	Minimum

Tabulka 2.2: Funkce, které může implementovat uzel uvnitř obrazového filtru.

Počáteční jedinec je generován náhodně. Evoluce je obvykle ukončena po předem určeném množství generací. K evaluaci kandidátního filtru je potřeba mít k dispozici originální obrázek bez šumu, který slouží k porovnání s filtrovaným obrázkem. Evoluce se snaží minimalizovat rozdíly mezi nepoškozeným a filtrovaným obrazem. Kvalita filtrování je určena numericky a mohou k tomu být použity různé přístupy z oblasti zpracování signálů. Mezi ně patří špičkový odstup signálu od šumu (PSNR), který musí být co nejvyšší a je definován rovnicí

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

kde $M \times N$ udává rozměry obrázků v (filtrovaný obrázek) a w (originální obrázek). Pro hardware, na kterém může být evoluce realizována, je ovšem výpočet hodnoty PSNR výpočetně náročná operace. Proto se nabízí alternativa ve formě střední odchylky na pixel (MDPP), kterou evoluce naopak zmenšuje

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

2.6.2 Koevoluční návrh obrazových filtrů

Podobně jako u symbolické regrese dokáže koevoluce v návrhu obrazových filtrů výrazným způsobem zkrátit dobu návrhu. Jeden běh evoluce obsahuje vytvoření 30 000 až 50 000 generací a při použití klasického CGP musí být při každé evaluaci fitness zpracovány stovky až tisíce pixelů z množiny trénovacích vektorů.

Šikulová a Sekanina v [13] ukázali, že je potřeba pouze 15 až 20 % z originálních testovacích vektorů k nalezení obrazového filtru stejné kvality, jako je schopné navrhnout standardní CGP.Rychlost návrhu se tímto způsobem zvýšila 2.99krát.

Obrázek 2.7 ilustruje princip, který autoři použili. V algoritmu 2 lze blíže poznat tento způsob. Ve dvou vláknech paralelně běží evoluce populace obrazových filtrů a evoluce trenérů, z nichž každý je reprezentován polem ukazatelů do celé množiny trénovacích vektorů.

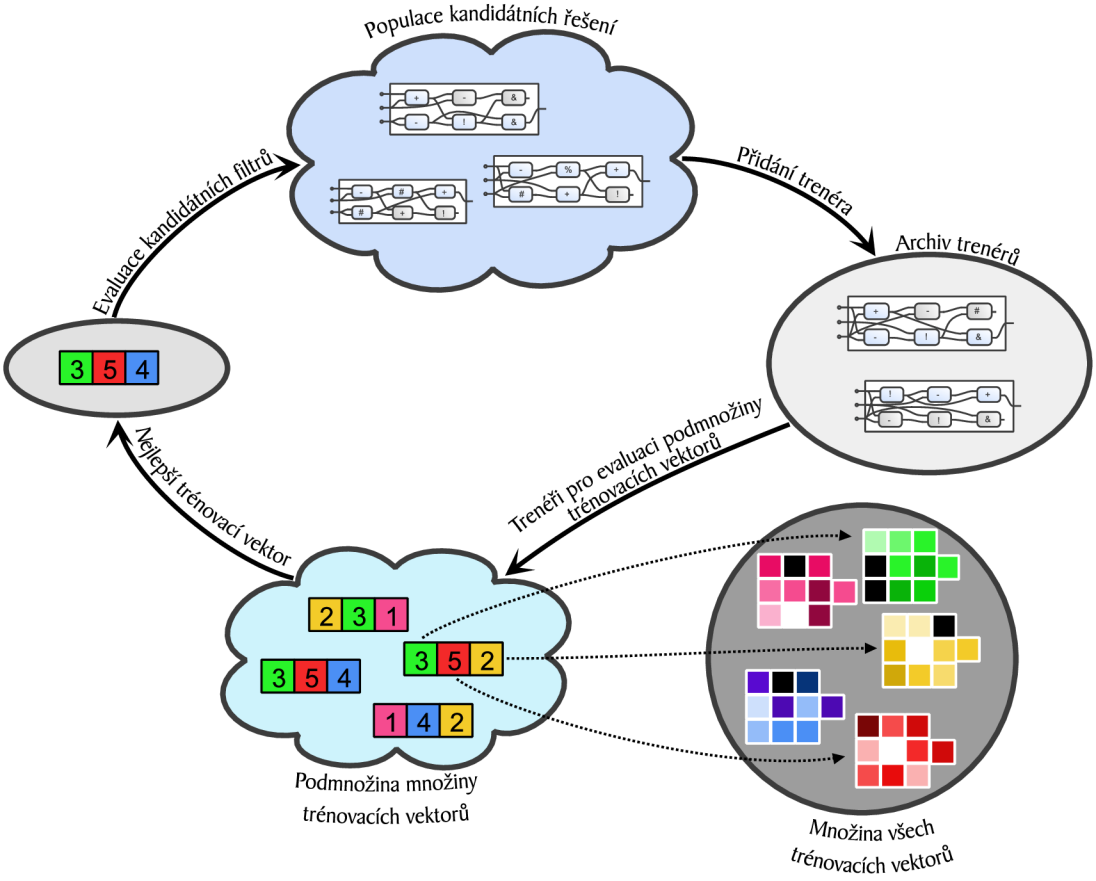
Populace spolu interagují přes dva archivy.

Algoritmus 2: Koevoluce populací kandidátních programů a testů.

```
begin Vlákno populace kandidátních filtrů
    Náhodně vygeneruj archiv trenérů;
    Náhodně vygeneruj populaci kandidátních filtrů;
    for 1 to celkový počet generací do
        Načti test ze sdílené paměti;
        Pomocí testu hodnoť filtry z populace kandidátních filtrů;
        Vyber nejlépe ohodnocený filtr jako rodiče;
        if fitness rodiče≠fitness předchozího rodiče then
            Ulož rodiče do archivu trenérů;
        Vytvoř novou generaci kandidátních filtrů s použitím rodiče;
    příznak ukončení = TRUE;
    Ohodnoť poslední nejlépe ohodnocený filtr použtitím objektivní fitness;
    return poslední nejlépe ohodnocený filtr
begin Vlákno populace testů
    Náhodně vygeneruj archiv testů;
    while TRUE do
        Načti archiv trenérů;
        [VOLITELNĚ] Ohodnoť trenéry objektivní fitness;
        Ohodnoť testy použitím trenérů;
        Vyber nejlépe ohodnocený test;
        Ulož nejlépe ohodnocený test;
        Vytvoř novou generaci testů s použitím turnajové selekce a jednobodového
        křížení;
        if příznak ukončení == TRUE then
            Ukonči evoluci populace testů;
```

Obvod sestrojený pomocí CGP může být beze změny použit pro implementaci hardwarového řešení v FPGA (*Field Programmable Gate Array*). Už od podstaty je program navrhován ve formě mřížky, jejíž programovatelné uzly mohou být přeneseny do funkcionality mřížky tvořené rekonfigurovatelnými hradly v FPGA. To, že FPGA může být výbornou platformou dokonce pro samotný koevoluční návrh obrazových filtrů publikoval Hrbáček s Šikulovou v článku [2]. Implementace koevolučního CGP v FPGA umožnila výrazné zrychlení procesu evolučního návrhu obrazových filtrů oproti standardnímu procesoru a tím umožnila provedení většího množství experimentů.

Soutěživá koevoluce není jediná cesta, kterou se můžeme při návrhu obrazových filtrů vydat. Ve formě *kompoziční koevoluce* to ukázali Šikulová, Komjáthy a Sekanina v článku [12]. Komponentami v publikovaném přístupu byly obrazový filtr a detektor šumu. Oproti případům, kdy byly evoluce těchto komponent prováděny odděleně se zlepšila kvalita filtrování o 1.3 dB v případě trénovacího obrázku a o 0.7 dB při použití série testovacích obrázků. Zrychlení doby evoluce bylo v tomto případě 1.31násobné oproti standardnímu CGP.



Obrázek 2.7: Koevoluce obrazových filtrů s trénovacími vektory.

Kapitola 3

Návrh

Tato práce navazuje na úspěšnou sérii vývoje inovací v oblasti evolučních algoritmů, které vznikaly v průběhu let na Fakultě informačních technologií VUT v Brně. Za přímého předchůdce můžeme považovat diplomovou práci Jiřího Hulvy [3], ve které byl představen nový typ prediktorů. Na základě výsledků této práce vznikl článek [11].

Nejen v případě návrhu obrazových filtrů, ale i v mnoha jiných výpočetních problémech, které jsou aplikací problémů reálného světa, je výpočet fitness kandidátního řešení velmi výpočetně náročný. V GP je fitness počítána přes množinu *případů fitness*. Případ fitness reprezentuje jednu situaci, ve které se navrhované řešení může nacházet a je potřeba chování kandidátního řešení na tuto situaci ohodnotit. Případ fitness reprezentuje jednak vstupy programu (každé devítiokolí poškozeného obrázku) a pak také výstupy, které bychom měli nalézt na výstupu ideálního řešení (hodnota prostředního pixelu originálního obrázku). Programovou strukturu, která obsahuje vstupy a očekávané výstupy od perfektně funkčního řešení nazýváme *trénovací vektor*.

Množina trénovacích vektorů je malý vzorek celého prostoru případů fitness. Zásadní rozhodnutí je právě to, z kolika případů fitness se má množina trénovacích vektorů skládat, aby byla dostatečně reprezentativní pro ohodnocení kandidátních řešení. Neméně zásadní je rozhodnutí, které případy fitness jsou pro ohodnocení kandidátního řešení vhodné.

Jeden z dřívějších přístupů ke zrychlení doby evolučního návrhu zautomatizoval druhou zmíněnou část. Evoluce prediktorů fitness probíhala na principech GA, ve kterém byl jedinec – prediktor – reprezentován jako vektor ukazatelů na trénovací data pevné délky. Zjistit délku tohoto prediktoru, aby byla co nejmenší a zároveň co nejlépe reprezentovala celý datový prostor, bylo již experimentální úlohou, která nezávisela na evoluci samotné.

V případě symbolické regrese tento přístup stále vykazoval dobré výsledky. Na pěti problémech symbolické regrese bylo zjištěno, že pouze 12 trénovacích vektorů (z celkového počtu 201) je potřeba k nalezení uspokojivého řešení. V okamžiku aplikace tohoto přístupu na návrh obrazových filtrů se ovšem experimentátor potýká s desítkami tisíc trénovacích vektorů a tedy nalezení nejvýhodnějšího počtu je daleko složitější. V této fázi tedy stojíme před otázkou, jak toto omezení znevýhodňující obrazové filtry a jim podobné výpočetní problémy reálného světa vyřešit.

3.1 Navržené řešení

Článek [11] popisuje přístup, ve kterém je hledání optimální délky prediktoru zahrnuto v samotné koevoluci. V takovém případě lze koevoluci s prediktory fitness jednodušeji apli-

kovat na složitější problémy reálného světa a ušetřit tím výpočetní čas. Ve zmíněném článku byl způsob aplikován na symbolické regresi a úkolem této diplomové práce je nový způsob s nepřímo kódovanými prediktory fitness aplikovat na problém návrhu obrazových filtrů.

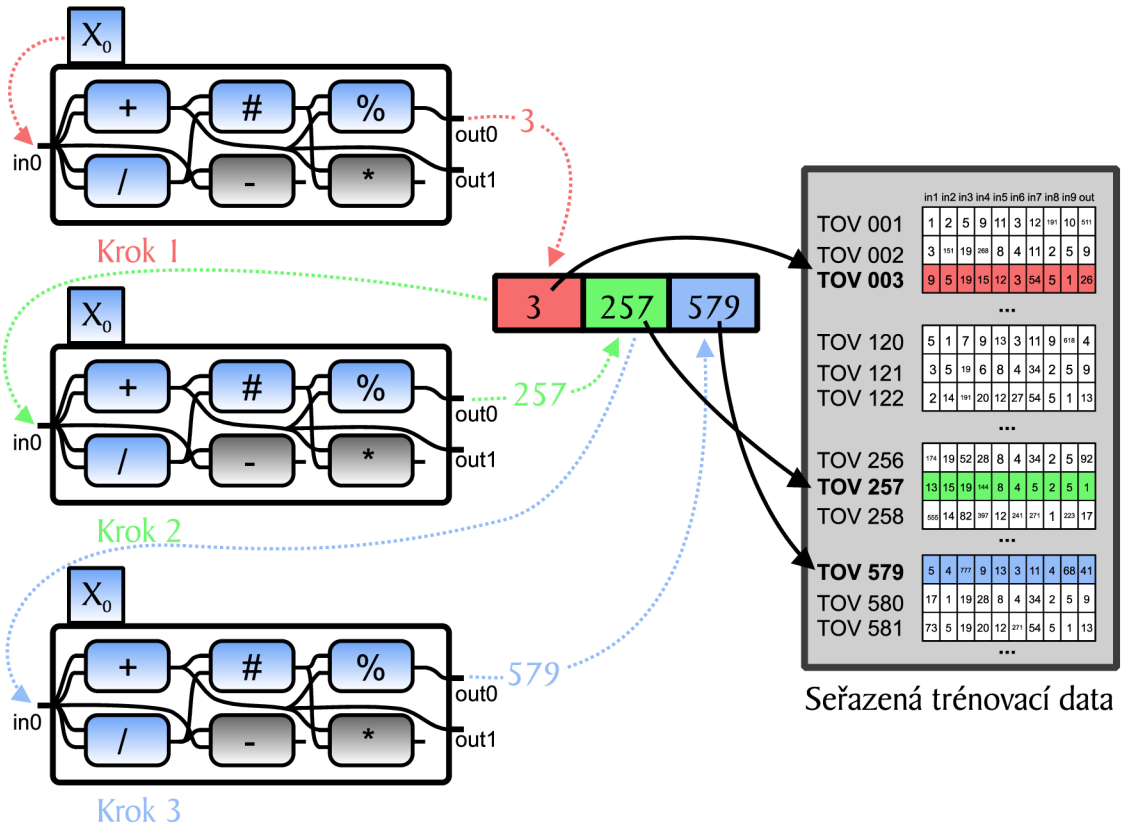
Princip navrženého řešení schematicky zobrazuje obrázek 3.4. Pojďme nyní probrat tento princip detailně a uvažujme rovnou jeho aplikaci pro návrh obrazových filtrů.

Nepřímo kódované prediktory

Nepřímé kódování prediktorů spočívá v tom, že genotypy jedinců populace jsou kódovány jako matematické výrazy, jejichž mapováním na fenotyp je získána posloupnost ukazatelů do trénovací množiny. Matematický výraz bude v populaci reprezentován kartézskou mřížkou a evoluce prediktorů bude probíhat na principech CGP. Tento přístup kódování prediktoru je schematicky zobrazen na obrázku 3.1.

Jak si lze všimnout, bloky, ze kterých se prediktory skládají, disponují dvěma vstupy a jedním výstupem. Funkce, které mohou bloky nad svými vstupy aplikovat, jsou vypsány v tabulce 3.1. Tyto funkce jsou převzaty z článku [11].

Primární vstup je jeden (in_0), zatímco výstupy prediktoru jsou dva (out_0 a out_1). Vstup přijímá hodnotu indexu trénovacího vektoru vypočítanou v předchozím kroku výpočtu. Jelikož na začátku neexistuje žádná taková hodnota, je nutné chromozom CGP rozšířit o jeden speciální gen nesoucí hodnotu pro inicializaci výpočtu, označovanou jako x_0. Hodnota genu



Obrázek 3.1: Způsob výpočtu trénovacího vektoru užitím nepřímo kódovaného prediktoru.

Funkce	Význam	Funkce	Význam
$x + y$	Součet	$x - y$	Rozdíl
$x \cdot y$	Součin	$x \div y$	Podíl
1	Konstanta	$\|x\|$	Absolutní hodnota
$min(x, y)$	Minimum	$max(x, y)$	Maximum
$\log(x)$	Logaritmus	$\sin(x)$	Sinus
$\cos(x)$	Cosinus	e^x	Mocnina Eulerova čísla
$x \mod y$	Modulo	x^y	Mocnina
$\neg x$	Negace	$\sqrt{\|x\|}$	Odmocnina absolutní hodnoty

Tabulka 3.1: Funkce implementující bloky prediktorů.

x_0 se mění v průběhu evoluce pomocí speciálního operátoru mutace – je násobena náhodně generovanou hodnotou v rozmezí zadaném uživatelem.

Primární výstup out_0 slouží pro výpočet hodnoty $j(x_i)$ ukazatele na trénovací vektor v seřazeném seznamu všech trénovacích vektorů. Index je vypočítán jako zbytek po podílu prvního primárního výstupu s počtem všech trénovacích vektorů:

$$j(x_i) = out_0(x_i) \bmod n. \tag{3.1}$$

Jak už bylo zmíněno, $out_0(x_i)$ je vždy použit v kroku $i + 1$ jako argument matematické funkce, kterou reprezentuje kartézská mřížka. Druhý výstup, označený jako out_1, slouží pro určení délky prediktoru. Proces generování nových hodnot ukazatelů j_{x_i} končí tehdy, když výstup $out_1(x_i)$ padne mimo uživatelem definovaný rozsah r_{out_1}. Existuje zde ještě jedna podmínka, při které končí generování nových ukazatelů, a to v případě, že je již predikována veškerá množina trénovacích vektorů, tzn. pro evaluaci kandidátního obrazového filtru již byl použit celý trénovací obrázek.

Trénování prediktorů

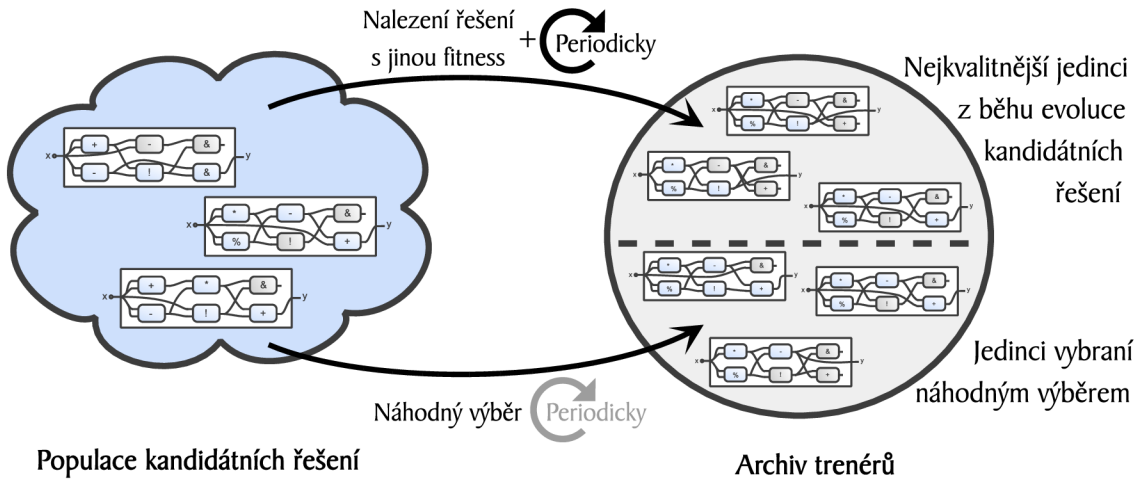
Koevoluce prediktorů fitness s evolucí kandidátních řešení je nutná z důvodu adaptace prediktorů na řešený problém v průběhu řešení. Jako trénovací data pro prediktory slouží vybraní jedinci, kteří jsou zkopírováni z populace kandidátních řešení (nazýváni *trenéři fitness*). Navíc je nutné, aby každý trenér u sebe nesl hodnotu f_{exact}, která vyjadřuje jeho skutečnou fitness, tj. PSNR vypočítanou užitím celého obrázku.

Trenéry fitness umísťujeme do struktury nazývané *archiv trenérů*. Počet trenérů v archivu je konstantní, předem definovaný. Ve fázi inicializace koevoluce jsou do archivu zkopírováni všichni jedinci z populace kandidátních řešení. V případě, že počet trenérů, kteří mají být v archivu umístěni, je vyšší než těch umístěných z první populace kandidátních řešení, chybějící je nutné vygenerovat náhodně.

Aktualizace archivu trenérů probíhá periodicky a trenér může být umístěn jednou ze dvou cest, jak je zobrazeno na obrázku 3.2 .

Umístěn zde je vždy nejlepší filtr z populace kandidátních řešení a to pouze v případě, že jeho predikovaná fitness má jinou hodnotu, než nejlepší nalezené řešení v předcházející generaci. Další trenér pro umístění do archivu je vybrán náhodně.

U nového trenéra t je vypočítána jeho f_{exact} a poté nahradí nejstaršího trenéra v archivu. Díky umísťování nejlépe ohodnoceného kandidátního řešení, udržíme v archivu reprezen-



Obrázek 3.2: Možnosti vložení kandidátního řešení do archivu: Aktuálně nejlepší nalezené řešení má rozdílnou fitness než v předcházející generaci, nebo pomocí náhodného výběru.

tativní vzorek populace řešení. Zároveň však, z důvodu umísťování náhodného jedince, diverzita fitness řešení zůstane maximální.

Fitness prediktorů

Kvalitu fitness prediktorů je potřeba hodnotit ze dvou hledisek. Jednak to je **přesnost predikce**, proti čemuž stojí **cena predikce**.

Přesnost predikce prediktoru p počítáme jako *relativní odchylku* predikované fitness $f_{predicted}$ a fitness skutečné f_{exact}:

$$prec(p) = \frac{1}{u} \sum_{j=1}^{u} u \, \frac{|f_{exact}(t_j) - f_{prediction}(t_j)|}{f_{exact}(t_j) + c}, \tag{3.2}$$

kde u je počet trenérů v archivu. Parametr c dovoluje ovlivnit rychlost nárůstu relativní chyby v případě, že se $f_{exact}(t_j)$ blíží k nule. Hodnotu převezmu z [11], tj. $c = 0.02$.

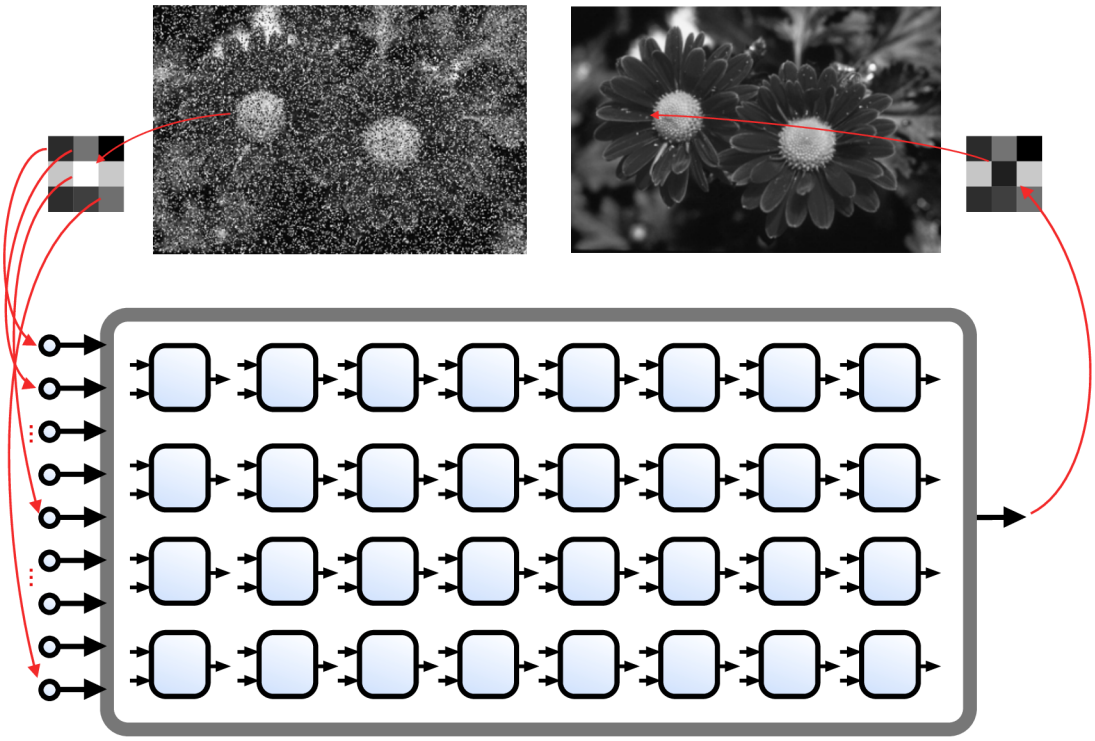
Cena predikce prediktoru p přímo úměrně závisí na počtu trénovacích vektorů, které budou použity pro výpočet predikované fitness kandidátního řešení užitím tohoto prediktoru. Počet trénovacích vektorů, na které ukazuje pole vypočítané pomocí prediktoru p budeme označovat jako $size(p)$.

Výpočet fitness prediktoru p je pak definován jako:

$$f(p) = (a \cdot prec(p))^4 + b \cdot size(p)(1 + a \cdot prec(p)^2), \tag{3.3}$$

kde a a b jsou váhy, které byly experimentálně doporučeny na hodnoty $a = 17$ a $b = 0.4$ [11]. Pro zjištění tohoto vztahu autoři použili nástroj *Eureqa*[1].

[1] Více o analyzačním nástroji Eurequa na http://www.nutonian.com/products/eureqa/.



Obrázek 3.3: Ukázka chromozomu kandidátního filtru a jeho funkce.

Kandidátní řešení

Jedinci populace kandidátních řešení jsou obrazové filtry. Pro návrh obrazových filtrů je výhodné použít kartézské genetické programování s mřížkou o rozměrech 8×4 bloky se dvěma vstupy a jedním výstupem. Počet primárních vstupů je 9 a výstup jeden, parametr l-back je 1. Ilustrativní ukázka jednoho takového chromozomu je na obrázku 3.3.

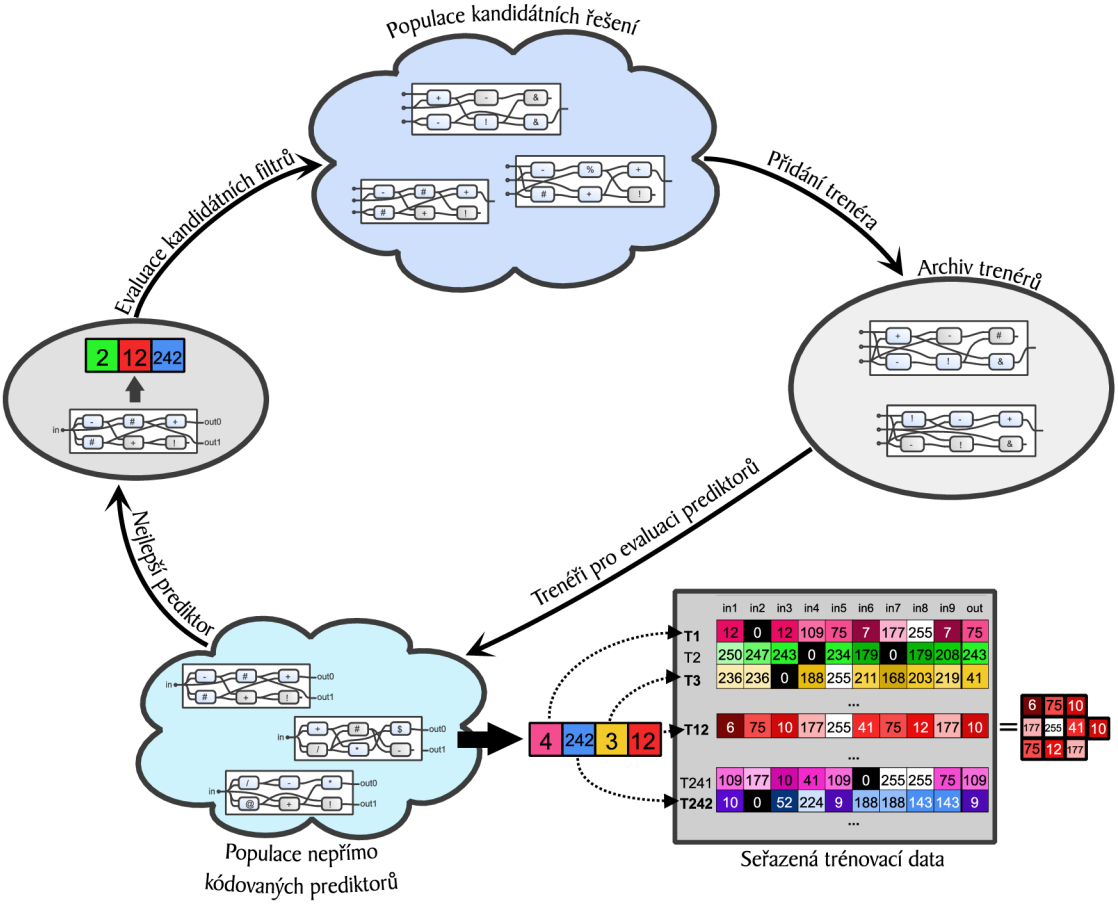
Filtr na svém vstupu přijímá devítiokolí pixelu. Díky tomu, že jsou tyto vstupy zpracovány pomocí funkčních bloků aplikující funkce z tabulky 3.2, na výstupu můžeme nalézt novou hodnotu prostředního pixelu. V ideálním případě pak získáme zcela opravený obrázek, stejně jako ilustruje 3.3.

Koevoluce obrazových filtrů a prediktorů fitness

Na obrázku 3.4 můžeme vidět schéma průběhu koevoluce. V prvním kroku jsou populace kandidátních řešení i populace nepřímo kódovaných prediktorů inicializovány náhodně.

Populace kandidátních řešení čeká na populaci prediktorů, než umístí svého nejkvalitnějšího jedince do archivu prediktorů. Evoluce kandidátního řešení probíhá na principech CGP. Trénovací vektory adresované prediktorem z archivu prediktorů jsou použity k výpočtu fitness kandidátních obrazových filtrů.

Aktuálně nejlepší prediktor je do archivu umísťován populací prediktorů periodicky, vždy po uběhnutí zadaného počtu generací. Během evoluce kandidátních řešení jsou nejlépe ohodnocené filtry s rozdílnou fitness oproti předchozí generaci zkopírovány do příslušné části archivu trenérů. V každé generaci prediktorů je jeden jedinec z druhé poloviny archivu



Obrázek 3.4: Koevoluce kandidátních řešení s prediktory fitness.

Kód	Funkce	Význam	Kód	Funkce	Význam
0	255	Konstanta	8	$x \gg 1$	Posuv doprava o 1 bit
1	x	Identická hodnota	9	$x \gg 2$	Posuv doprava o 2 bity
2	$255 - x$	Inverze	10	$swap(x, y)$	Přehození nibblů
3	$x \vee y$	Bitová disjunkce	11	$x + y$	Součet
4	$\bar{x} \vee y$	Disjunkce s neg. x	12	$x +^S y$	Součet s ořezáním
5	$x \wedge y$	Bitová konjunkce	13	$(x + y) \gg 1$	Průměr
6	$\overline{x \wedge y}$	Negovaná konjunkce	14	$max(x, y)$	Maximum
7	$x \bigoplus y$	Exkluzivní disjunkce	15	$min(x, y)$	Minimum

Tabulka 3.2: Přehled funkcí s identifikačními kódy, které mohou implementovat uzly kandidátního filtru.

trenérů nahrazen náhodně vybraným obrazovým filtrem.

Evoluce prediktorů také aplikuje principy CGP. Užitím každého prediktoru ve stávající generaci je určena predikovaná fitness a následně je určena fitness prediktoru. Nejlépe ohodnocený prediktor pak vystřídá původní prediktor v archivu prediktorů. Také je použit pro vytvoření nové generace prediktorů.

Celý proces se opakuje, dokud neproběhne zadaný počet generací.

Kapitola 4

Implementace

Pro experimentální vyhodnocení navrženého koevolučního přístupu k návrhu obrazových filtrů bylo potřeba získat spustitelnou verzi navrženého programu. Implementace probíhala postupně a směřovala k realizaci navrženého přístupu. Z úplného počátku vznikla verze běžného přístupu k návrhu obrazových filtrů, jak bylo popsáno v 2.6.1 implementovaná v programovacím jazyce Java. Přesto, že tento jazyk preferuji, neukázal se jako vhodný pro experimentální činnost. Doba evoluce byla příliš dlouhá a nedovolila by provedení potřebného počtu experimentů v zadaném čase. Tuto implementaci považuji za zdroj poučení se z chyb, kterých jsem se mohl vyvarovat při vytváření finální verze.

Bylo potřeba zvolit jazyk nižší úrovně, proto jsem přistoupil k C++. Stejně jako v programátorské praxi, tak i při vývoji softwaru pro diplomovou práci se osvědčilo pravidlo, že není nutné a ani z časového hlediska žádoucí vytvářet tentýž zdrojový kód pro různé případy použití. Jak jsem již zmínil v kapitole 3, na koevoluci s nepřímo kódovanými prediktory byla založena již práce Jiřího Hulvy, jehož aplikací byla symbolická regrese. Z programových knihoven, které vyvinul v rámci diplomové práce bylo možné použít základ – programovou kostru. Jádro programu bylo nahrazeno tak, aby spustitelná koevoluce navrhovala obrazové filtry. Muselo se přihlédnout k mnoha rozdílům, které beze sporu mezi symbolickou regresí a návrhem obrazových filtrů jsou, např. množství trénovacích vektorů. Následující odstavce se tímto zabývají blíže.

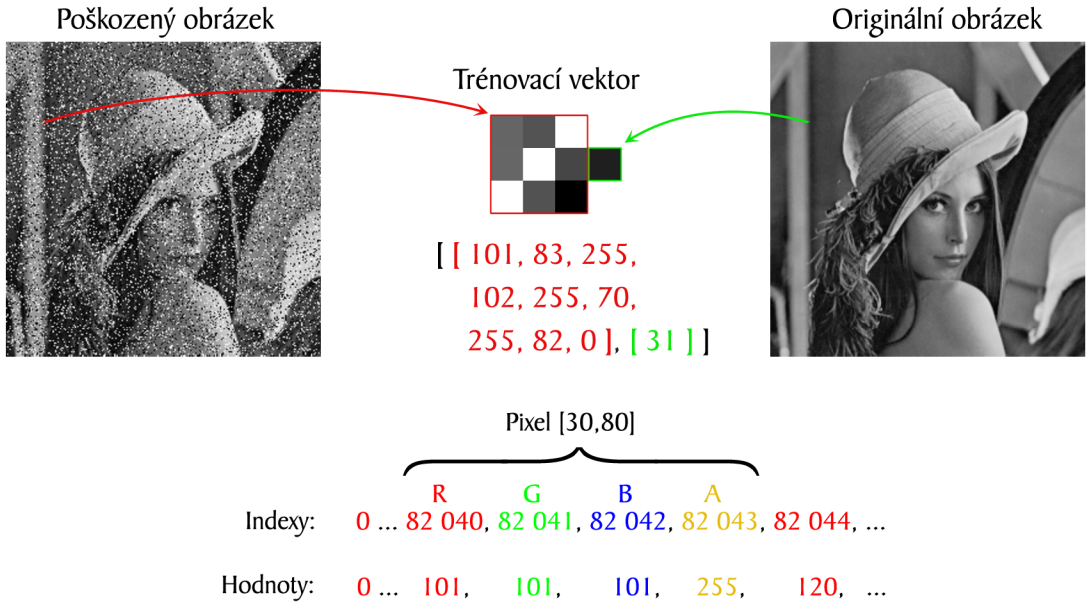
4.1 Načtení obrázku

Algoritmus pracuje s obrázky v odstínech šedé. Vyvinutý program pracuje s těmito obrázky ve formátu *PNG, (Portable Network Graphics)*. Pro práci s obrázky jsem použil knihovnu *LodePNG*[1], která nabízí komfortní způsob, jak dekódovat obrázek do vektoru s prvky typu `unsigned char`. Na webových stránkách této knihovny je k dispozici verze jak v ISO C90, tak knihovna pro C++. Pro mě bylo vhodnější zvolit druhou zmíněnou verzi.

Výstupem funkce `decode` knihovny LodePNG je vektor prvků typu `unsignet char`. Obrázek je dekódován ve formátu RGBA. Každý jeden pixel obrázku tudíž zabírá ve výstupním vektoru 4 pozice – pro červenou (R), zelenou (G) a modrou (B) složku pixelu tři pozice a na čtvrté nalezneme složku pixelu označovanou jako alfa (A). Jelikož pracujeme s obrázky v odstínech šedi, první tři složky mají stejnou hodnotu, tedy R=G=B.

V následujícím kroku program projde tímto polem a postupně naplňuje strukturu určenou pro trénovací vektor. Ta se skládá z pole o velikosti 9 pro vstupy kandidátního filtru

[1] LodePNG je knihovna volně dostupná na url http://lodev.org/lodepng/.



Obrázek 4.1: Ukázka vytvoření jednoho trénovacího vektoru a indexace prvního pixelu v poli vzniklém dekódováním poškozeného obrázku.

(tyto vstupy získáme z poškozeného trénovacího obrázku) plus jeden prvek jako originální hodnota prostředního pixelu (získaná z originálního obrázku). Vytvořený trénovací vektor je vložen do vektoru uchovávající seřazenou sekvenci všech trénovacích vektorů.

Na obrázku 4.1 je ilustrováno vytvoření jednoho trénovacího vektoru. Z poškozeného obrázku je vybráno devítiokolí pro vstup kandidátního filtru, z originálu vyčteme hodnotu prostředního pixelu, kterou by měl vypočítat ideální filtr. Ve spodní části obrázku uvádím ukázku indexace pole, které je výsledkem dekódování png obrázku. Index na hodnotu pixelu $[x, y]$ lze vypočítat jako

$$index = (y \cdot width + x) \cdot 4, \tag{4.1}$$

kde $width$ je šířka obrázku a konstanta 4 udává, že se každý pixel skládá ze čtyř složek – R, G, B a A. Výpočet indexů celého devítiokolí pak ukazuje tabulka 4.1.

4.2 Poškození obrázku

Pro určení kvality filtru metodou PSNR je nutné, aby měl program k dispozici dva kusy obrázku – jednak obrázek originální a pak také poškozený obrázek. Je nutné, aby uživatel vždy dodal originální obrázek. Z pohledu poškozeného obrázku existují 2 možnosti. Uživatel může poškozený obrázek programu dodat, implementována je ovšem i jednodušší možnost. Stačí jako parametr vložit procentuální poškození a program poškozený obrázek dopočítá. V případě, že uživatel ponechá poškození na programu a rozhodne se jej spustit s parametrem pro míru poškození, je nutné, aby vstupní originální obrázek byl ve formátu png s osmibitovou hloubkou kódování pixelů.

Poškození obrázku probíhá nad kopií pole s hodnotami pixelů originálního obrázku. Počet pixelů k poškození je vypočítán prostým násobením procent s počtem všech obra-

$(index - 4) - width \cdot 4$	$index - width \cdot 4$	$(index + 4) - width \cdot 4$
$index - 4$	$index$ (podle 4.1)	$index + 4$
$(index - 4) + width \cdot 4$	$index + width \cdot 4$	$(index + 4) + width \cdot 4$

Tabulka 4.1: Výpočet indexů pixelů devítiokolí ve vektoru po dekódování obrázku. Význam proměnných: *width* je šířka obrázku, *index* je aktuálně řešený pixel, jehož hodnota je vypočítána pomocí 4.1 a konstanta 4 znamená, že každý pixel se skládá ze čtyř složek – R, G, B a A.

zových pixelů. V cyklu probíhá poškození vypočítaného množství náhodně generovaných indexů do kopie pole originálního obrázku. Hodnota pixelu na tomto indexu je přepsána na 0 (černá) nebo 255 (bílá) pro šum typu sůl a pepř nebo náhodnou hodnotou v rozmezí od 0 do 255 (různé odstíny šedé) v případě náhodného šumu.

Procentuální zastoupení poškozených pixelů je ohlídáno použitím originálního obrázku. Indexy pro poškození obrázku jsou náhodně generovány do té doby, než je nalezen pixel, který dosud poškozený není. V cyklu vždy ověřuji, jestli pixel na náhodném indexu nese hodnotu jako pixel stejného indexu originálu a měním jej jen v tomto případě. V opačném případě (tzn. hodnoty jsou rozdílné, čili pixel v kopii je již poškozený) generuji jiný náhodný index. Mohlo by se totiž stát, že budou několikrát poškozovány stejné pixely a kvůli tomu bychom nezískali obrázek se zadanou úrovní šumu a experimentálnímu výsledku bychom tudíž nemohli přikládat takovou váhu, jaká je pro vyhodnocení metody potřeba.

V případě, že pomocí koevoluce hodláme navrhovat obrazový filtr jiného charakteru než je pepř a sůl nebo náhodný výstřelový šum (např. detektor hran), je potřeba vždy na vstup přivést jak vstupní, tak výstupní obrázek.

4.3 Paralelní procesy

Koevoluce v principu počítá s procesy navzájem ovlivňujícími se, běžícími vedle sebe. Také v přírodě, kde pro tuto metodu bereme inspiraci, musí jednotlivé entity (živočichové, rostliny apod.) existovat vedle sebe ve stejném čase, aby svůj vývoj mohly ovlivňovat v tom smyslu, jak to uvažuje koevoluce. Simulaci tohoto prostředí navzájem ovlivňujících se entit koexistujících ve stejném čase je pro výpočet vhodné provést také v softwaru.

Programová knihovna *OpenMP* umožňuje naprogramování paralelních procesů užitím tzv. direktiv pro překladač prostřednictvím API dostupného přímo pro C++ (původně API pro C a Fortran). Alternativ pro programování paralelních procesů je více, za všechny zmiňme *MPI*, jehož použití funguje na jiném principu – tzv. zasílání zpráv. Zatímco při práci s OpenMP umísťujeme direktivy do zdrojového kódu, který mohl být vytvořen i dříve, MPI a jeho zasílání a příjem zpráv používáme jako volání jednotlivých funkcí této knihovny. Daleko významnějším rozdílem, který pro koevoluci upřednostňuje OpenMP před MPI, je sdílená paměť. Archivy, prostřednictvím kterých je v koevoluci zajišťována interakce populací, jsou sdílenou pamětí obou populací. Populace prediktorů čte z archivu trenérů a zapisuje do archivu prediktorů, ze kterého naopak načítá prediktor pro ohodnocení svých jedinců populace kandidátních řešení, která má právo zapisovat do archivu trenérů. OpenMP

byla tedy jasná volba pro implementaci paralelních procesů v mém programu.

Spuštění paralelního běhu obou generací je provedeno ve třídě `Coevolution`. Zde jsou spuštěna 2 vlákna a v každém z nich běh jedné evoluce. Evoluce kandidátního řešení jsou oddělena od evoluce prediktorů, k čemuž slouží 2 třídy – `EvoSolution` a `EvoPreditcor`. Obě třídy jsou potomky třídy `cgp`, ve které lze nalézt implementaci kartézského genetického programování. V evoluci kandidátních řešení pracuji s jedinci, kteří jsou instanciací třídy `Candidate`, která je potomkem třídy `Individual` používané pro jedince v evoluci prediktorů. `Candidate` se od `Individual` mnoho neliší, jsou ovšem nastaveny jiné funkce pro funkční bloky kartézské mřížky a jiná rozšíření zmíněná v textu dříve. Potomek ovšem může převzít společné atributy a metody, jako je například způsob dekódování fenotypu z genotypu.

Evoluce kandidátního řešení probíhá ve smyčce, která je ukončena po uběhnutí zadaného počtu generací. Ukončení evoluce je řízeno sdílenou proměnnou `running`. Okamžikem, kdy se dospěje požadovaného počtu generací (jehož hodnotu kontroluje evoluce kandidátních řešení inkrementací proměnné od nuly), je hodnota `running` změněna z `true` na `false` a tím je i evoluce prediktorů uvědoměna, že koevoluce dospěla k závěru. V každé generaci je nejprve nalezen nejkvalitnější jedinec (užitím prediktoru) a z tohoto jedince jsou mutací vytvořeni jedinci nové generace. Pokud je čas na kontrolu hodnoty prediktoru (po každém uběhnutí 40 generací), je nahlédnuto do archivu prediktorů, zda se prediktor zde uložený od poslední aktualizace, změnil. S archivy – sdílenými proměnnými, je operováno jako s kritickými sekcemi. Tedy i tato kontrola změny prediktoru je zabezpečena uzamknutím zámku typu `omp_lock_t`. Zvlášť (tzn. po jiném počtu generací – v mé implementaci po 500) je do příslušné poloviny archivu trenérů (jak bylo vysvětleno u obrázku 3.2) odeslán nový trenér. Ten je odeslán také vždy, pokud nejlepší dosud nalezené řešení má jinou predikovanou fitness oproti předchozí generaci. S odesláním prediktoru je také spojen výpočet přesné fitness f_{exact} užitím celé množiny trénovacích vektorů a uložena do archivu je až dvojice $filter, f_{exact}$.

V případě, že není vypnuta koevoluce, je spuštěna také evoluce prediktorů. První krok vývoje každé generace je shodný s toutéž částí evoluce kandidátních řešení, tj. nalezení nejkvalitnějšího jedince (užitím trenérů z archivu) a vytvoření nové populace použitím genetikou inspirovaného operátoru mutace. V další fázi této evoluce dojde ke kontrole čítače náhodného výběru trenéra. Pokud je čas na náhodný výběr, konkrétně pak každou stou generaci, touto populací náhodně vybraný trenér nahradí nejstaršího trenéra v druhé polovině archivu. Uběhne-li počet generací příslušejících počtu generací k odeslání prediktoru do archivu (nastaveno na každých 1 000 generací), je nejlepší prediktor dané generace zkopírován do archivu, kde slouží k ohodnocení kandidátních řešení.

Jak již bylo naznačeno, program bere v úvahu vypnutí koevoluce. V tomto případě, není evoluce prediktoru vůbec spuštěna a běh druhého vlákna je ihned zrušen i s vláknem samotným. Kandidátní řešení potom po celou dobu evoluce počítají s celou množinou trénovacích vektorů, tj. s celým trénovacím obrázkem.

4.4 Sběr statistických dat

Za běhu koevoluce (či evoluce, je-li aktuálním předmětem zkoumání sekvenční cgp) jsou ukládány hodnoty některých atributů, které nás zajímají pro statistické zhodnocení použitého přístupu. V průběhu běhu programu je v každé spuštěné (ko)evoluci, kterých může být spuštěno více sekvenčně za sebou, do konzole vždy vypsána formátovaná hlavička s údaji o nastavení, se kterými je koevoluce spouštěna. V řádcích pak aktuální nalezená fitness

jedince po tisícovce uběhnutých generací. Po ukončení běhu je pak vypsán procesorový čas vyčerpaný pro běh, celkový počet vyčíslení výstupu výpočetních uzlů[2], skutečná fitness nejlepšího nalezeného řešení a jeho chromozom.

V případě, že bylo spuštěno více než jeden běh (ko)evoluce, jsou z dosažených výsledků do konzole vypsány statistické údaje ve formě maximální, minimální hodnoty, mediánu a prvního a třetího kvartilu výše zmíněných sledovaných hodnot.

Pro statistické vyhodnocení metody je nutné monitorovat i samotný průběh. Proto jsou ukládány po 10 000, 30 000 generacích a vždy při ukončení poslední generace (kterých bylo pro experimenty spuštěno 100 000, jak bude zmíněno dále) tyto údaje:

- skutečná fitness (PSNR) aktuálně nejlepšího nalezeného filtru,
- procesorový čas, který byl do aktuální generace vyčerpán pro výpočet,
- počet vyčíslení výpočetních uzlů,
- chromozom nejlepšího aktuálně nalezeného kandidátního řešení a

dále je vždy po uběhnutí 500 generací uložena

- délka prediktoru umístěného v archivu prediktorů.

Tyto údaje lze nastavením příslušného parametru při spuštění uložit do souborů. Vytvořeny jsou 3 soubory ve formátu csv, které v názvu obsahují uživatelem zadanou předponu. Přípony názvů jednotlivých souborů, jejich obsah a oddělovače údajů jsou následující:

- *_boxPlot.csv* pro fitness, procesorový čas a počet vyčíslení s oddělovačem „,“ (čárka) mezi údaji,
- *_chromozoms.csv* pro chromozomy aktuálně nejlepších řešení s oddělovačem „;“ (středník) mezi údaji a
- *_prLen.csv* pro délku prediktoru umístěného v archivu prediktorů s „,“ (čárkou) mezi údaji.

4.5 Přídavná funkcionalita

Kvůli studentskému omezení přístupu na fakultní servery bylo nutné implementovat nějaká další opatření, která se přímo řešeného problému netýkala.

Skripty pro opakované spuštění programu se zadanými parametry ve skriptovacím jazyce Bash spouštěly vždy jeden běh programu v cyklu až do získání požadovaného množství běhů. Navrácené hodnoty shromažďovaly v souborech, ze kterých bylo možné vygenerovat jakési průběžné výsledky, díky kterým byl program odladěn.

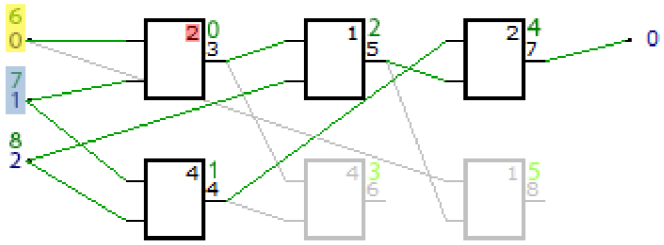
Uložení konfigurace a zotavení sekvenčního CGP bylo potřeba právě kvůli velké výpočetní náročnosti této metody. Na fakultním serveru s právy studenta nebylo možné s maximálním přiděleným procesorovým časem získat výsledky CGP z 50 000 běhů. Naštěstí, uložení konfigurace CGP nemá takovou složitost, jako uložení konfigurace koevoluce[3].

[2] Nechť `getResultCnt` je atribut každého jedince, který je z nuly inkrementován při každém vyčíslení výstupu výpočetního uzlu kartézské mřížky. Jako celkový počet vyčíslení všech uzlů uvažujeme součet `getResultCnt` všech jedinců obou populací.

[3] V koevoluci by bylo nutné uložit jak konfigurace obou populací, tak obsahy archivů.

Chromozom pro cgp viewer a filtr:

```
{3,1,3,2,2,1,5}
([3]0,1,2)([4]1,2,4)([5]3,2,1)
([6]3,4,4)([7]4,5,2)([8]0,5,1)
(7)
```



Chromozom využívající koevoluční návrh:

```
6,7,2,7,8,4,0,8,1,
0,1,4,1,2,2,6,2,1,
5
```

Obrázek 4.2: Kódování chromozomu použité při koevolučním návrhu umísťuje indexy primárních vstupů na konec chromozomu oproti kódování, které používá filtr a cgp viewer.

Během evoluce byl do zadaného souboru zapisován na první řádek počet uběhnutých generací a na navazující řádky pak hodnoty genů chromozomu nejlepšího jedince v dané generaci. Při zpětném spuštění programu bylo nejprve nahlédnuto do souboru a pokud počet generací poukazoval na rozpočítanou úlohu, program načetl chromozom jedince, kterého nastavil jako nejlepšího, ze kterého byla generována následující populace – o jedničku vyššího pořadí než ze souboru načtená hodnota.

4.6 Filtr

Mimo spustitelné verze koevoluce obrazových filtrů s nepřímo kódovanými prediktory fitness jsem vytvořil také program, který na vstupu přijme chromozom filtru navrženého koevolučním procesem, popř. sekvenčním cgp. Jako implementační jazyk pro tento program jsem zvolil Java SE 8. Tato část nebude nijak časově kritická a tak není nutné držet se nízkoúrovňového C++ a můžu bez obav využívat komfort prostředí Java Virtual Machine.

Na vstupu je přijímán chromozom v kódování zobrazeném na obrázku 4.2 nahoře. Tuto notaci zakódování chromozomu jsem převzal z aplikace *Chromozom viewer*[4]. Kódování chromozomu se liší od kódování použitém v přejatých knihovnách C++. Koevoluce předpokládá kódování jako pole čísel typu `integer` s tím, že od nuly jsou indexovány geny bloků, na něž teprve navazují indexy genů primárních vstupů. V tomto chromozomu nenalezneme informace o rozměrech mřížky, počtu primárních vstupů a výstupů, ani další parametry kartézské mřížky či označení jednotlivých výpočetních uzlů. Není to potřeba,

[4]Chromozom viewer je dostupný přes fakultní přístupové údaje na url http://www.fit.vutbr.cz/~vasicek/courses/bin_lab1/bin/viewer.zip.

protože v programu má tyto parametry evoluce zaznamenány jinde – v globálních proměnných, které v sobě nese třída `Settings`. Převod z kódování užívaného během evoluce na to, které je potřeba pro zobrazení chromozomu implementuje metoda ve třídě `Candidate`.

V implementaci obrazového filtru je z chromozomu (umístěného v souboru na zadané url) sestrojen fenotyp přesně tak, jak je zobrazeno na obrázku 4.2. Ze vstupu je načten také poškozený obrázek, který je upraven sestrojeným filtrem a nově vypočítané hodnoty pixelů jsou převedeny na obrázek formátu png. V případě, že uživatel na vstup vložil také originální obrázek, program vypočítá PSNR originálního obrázku oproti filtrovanému.

Tato implementace poslouží k výpočtu průměrné kvality (PSNR) filtru nad sérií testovacích obrázků.

Kapitola 5

Experimentální vyhodnocení

Navržená a implementovaná metoda koevolučního návrhu obrazových filtrů s nepřímo kódovanými prediktory fitness bude následně vyhodnocena na třech skupinách úloh pro obrazové filtry typických. Na níže popsaných úlohách zpracování obrazu je tendence zvyšovat kvalitu obrazových filtrů užitím biologií inspirovaných metod po řadu let. Definice těchto úloh i jejich vysvětlení je aplikací kapitoly – *Zpracování obrazu a CGP* [9]. Následovat bude vyhodnocení metody na definovaných úlohách.

5.1 Úlohy pro vyhodnocení

Zpracování obrazu zahrnuje velké množství úloh, jako například filtrování obrazu, detektor hran, vyvážení úrovní obrazu prostřednictvím histogramu, nebo úpravu kontrastu a světlosti obrazu. Při získávání následujících výsledků byly v experimentech navrhovány obrazové filtry pro různé úrovně jednotlivých druhů impulzního šumu. Zkoumanou úlohou byl také automatizovaný vývoj detektoru hran.

Impulzní šum je často vyskytujícím se typem šumu. Ve většině případů je způsoben chybnými pixely ve snímači fotoaparátu či kamery, chybami v paměťových oddílech konečného úložiště obrázku nebo z důvodu rušení během přenosu dat mezi několika nosiči. Rozlišujeme dva typy impulzního šumu: *šum typu sůl a pepř* a *náhodný výstřelový šum*. Pro obrázky poškozené šumem typu sůl a pepř platí, že poškozené pixely mají maximální nebo minimální možnou hodnotu. Název má kořeny v černobílé fotografii, kde je minimální hodnotou černá barva (v digitálním kontextu hodnota pixelu rovna 0) a hodnotou maximální barva bílá (pro 8 bitů tedy 255). Náhodný výstřelový šum je kvůli rozpětí hodnot složitější – poškozené pixely nabývají libovolné náhodné hodnoty v rozmezí od černé až k bílé, čili z intervalu $\langle 0, 255 \rangle \subset \mathbb{N}$. Impulzní šum (ať už typu sůl a pepř nebo šum náhodný) je charakterizován parametrem *intenzita šumu* p. Parametr p určuje poměr poškozených pixelů vůči celkovému množství pixelů v obrázku.

Pro další zkoumanou úlohu – návrh detektoru hran – byl jako referenční výstup zvolen nejvíce populární detektor hran, který je založený na tzv. Sobelovu operátoru. Ten funguje na principu měření spádu ve 2D prostoru obrázku. Spád je měřen pomocí matematického operátoru konvoluce. Konvoluce je provedena mezi obrázkem a dvěma jádry. Výsledkem jsou tedy dvě matice, jejichž sečtením a následným přičtením konstanty získáme výsledek – obrázek, na kterém jsou zvýrazněny detekované hrany.

Na detektor hran lze nahlížet jako na obrazový filtr, jehož vstupem je originální obrázek a na výstupu očekáváme obrázek s detekovanými hranami. Můžeme tedy experimenty

(a) Sůl a pepř, $p = 20\%$

(b) Náhodný šum, $p = 20\%$

Obrázek 5.1: Ukázka impulzního šumu typu sůl a pepř (a) a náhodného impulzního šumu (b) stejného zastoupení poškozených pixelů.

(a) Originální obrázek

(b) Sobelův operátor

Obrázek 5.2: Ukázka vstupního obrázku (a) a obrázku s detekovanými hranami (b) užitím Sobelova operátoru.

Úloha	Intenzita šumu [%]								Počet testovacích úloh (obrázků)
Impulzní šum typu sůl a pepř	10	20	30	40	50	60	70	80	9
Impulzní šum náhodný	10	20	30	40	50	60	70	80	9
Detektor hran	–								8

Tabulka 5.1: Úlohy použité pro experimentální vyhodnocení metody.

rozšířit i o návrh takového obrazového filtru. Jako trénovací obrázek bude použit obrázek, na který byl aplikován právě detektor hran používající Sobelův operátor.

5.2 Experimentální nastavení

Průběh automatizovaného návrhu řešení nějakého problému (v mém případě obrazové filtry) s využitím kartézského genetického programování lze ovlivňovat mnoha parametry. V případě, že navrhujeme koevolučně, počet parametrů ještě roste. Hodnoty jednotlivých parametrů, které jsem použil pro experimentální vyhodnocení zkoumaného přístupu představuje tato kapitola.

Evoluce kandidátních řešení a archiv trenérů

Jelikož je cílové řešení hledáno v populaci kandidátních řešení, právě jeho evoluce je evolucí řídicí, čili tou, která hlídá počet uběhnutých generací. Koevoluce je ukončena po 100 000 generacích, v rámci experimentů bude vždy spuštěno 100 běhů s jedním nastavením. Pro vyhodnocení tedy bude k dispozici 100 koevolucí vyvinutých řešení dané úlohy. Rozměry i ostatní parametry jedince této populace jsou shrnuty v tabulce 5.2. Hodnoty vychází z běžného návrhu evolučních filtrů, jak byl popsán v [9].

Pravidelné uložení aktuálně nejlepšího kandidátního řešení do archivu trenérů proběhne vždy po uběhnutí čtyř set generací. Trenér je do archivu umístěn také v případě, kdy se fitness nejlepšího jedince v populaci změní.

Evoluce prediktorů a archiv prediktorů

Vhodné hodnoty parametrů evoluce prediktorů byly experimentálně nalezeny autory článků [14] a [11]. Použité hodnoty shrnuji v tabulce 5.3.

Jedinec této populace disponuje dvěma primárními výstupy. První z nich (Out_0) je použit jako ukazatel do seřazeného pole trénovacích vektorů a pro hodnotu primárního vstupu následující iterace. Počáteční kolo výpočtu přijímá na vstupu hodnotu uloženou ve speciálním genu X_0, který rozšiřuje genotyp oproti standardnímu cgp, který publikoval Miller [5].

Out_2 ovlivňuje délku prediktoru. Pokud jeho vypočítaná hodnota padne mimo stanovený rozsah mezi -2000 a 2000, je generování predikované sady ukončeno. Velikost prediktorem určené množiny trénovacích vektorů však nikdy neklesne pod 0.2 % celkového počtu trénovacích vektorů, ani nepřesáhne 25 % obsahu této sady. V konkrétních hodnotách pro zvolený

Parametr	**Hodnota**
Počet jedinců v populaci	8
Počet řádků jedince	4
Počet sloupců jedince	8
Počet primárních vstupů a výstupů	9 vstupů, 1 výstup
Počet vstupů a výstupů funkčního bloku	2 vstupy, 1 výstup
L-back	1
Pravděpodobnost mutace (počet mutovaných genů)	5 % až 10 % (1 až 3 geny)
Funkce bloků	Podle tabulky 3.2
Počet generací do odeslání trenéra	400
Počet běhů jedné úlohy	100
Počet generací	100 000

Tabulka 5.2: Parametry určující nastavení kandidátních řešení.

Parametr	**Hodnota**
Počet jedinců v populaci	5
Počet řádků jedince	15
Počet sloupců jedince	4
Počet primárních vstupů a výstupů	1 vstup, 2 výstupy
Povolené rozmezí Out_1	$< -2000, 2000 > \subset \mathbb{R}$
Počet vstupů a výstupů funkčního bloku	2 vstupy, 1 výstup
L-back	4
Pravděpodobnost mutace (počet mutovaných genů)	1 % až 30% (1 až 18 genů)
Počet prediktorem reprezentovaných trénovacích vektorů	0.2 % až 25 %
Funkce bloků	Podle tabulky 3.1
Počet generací do odeslání prediktoru	2 000

Tabulka 5.3: Parametry určující nastavení kandidátních řešení.

trénovací obrázek v rozlišení 256×256 pixelů se jedná o vzorek trénovací sady o velikosti od 131 do 16 384 trénovacích vektorů.

Počet generací evoluce prediktorů není nijak exaktně určen. Evoluce prediktorů probíhá v samostatném vlákně a ukončena je zároveň s ukončením evoluce kandidátních řešení, tedy po uběhnutí 100 000 generací kandidátních řešení. Přesnou hodnotou je ovšem určen počet generací, který uplyne mezi jednotlivým odesláním prediktoru do archivu pro použití v evoluci kandidátních řešení. Nastaveno je 2 000 generací, které mezi dvěma odesláními pokaždé uplynou. Tato hodnota, stejně jako hodnota počtu generací mezi odesláním trenéra v případě kandidátních řešení, byla zjištěna experimentálně pro úlohu symbolické regrese jako nejlepší [11]. Jelikož se metoda nemění, je vhodné převzít tuto hodnotu při experimentálním vyhodnocení její aplikace na úlohu jinou, úlohu koevolučního návrhu obrazových filtrů s nepřímo kódovanými prediktory fitness.

5.3 Chování prediktorů

Během sta tisíc generací kandidátních řešení byla každých 500 generací pravidelně zaznamenána délka prediktoru, který je umístěn v archivu pro ohodnocení kandidátního řešení. Z každého běhu bylo tedy získáno 200 vzorků reprezentujících chování prediktorů ve smyslu konvergence jejich velikosti během koevolučního návrhu.

Impulzní šum typu sůl a pepř

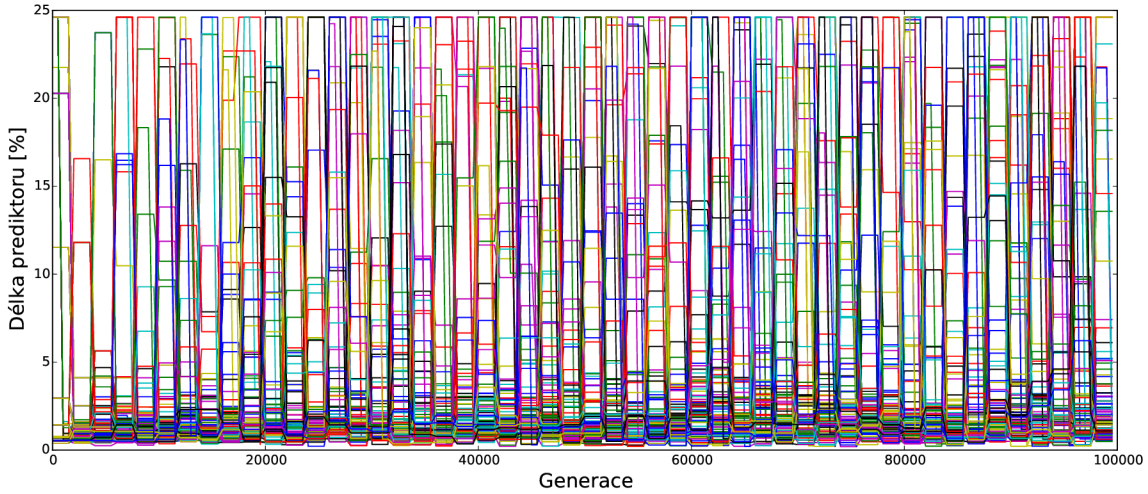
V tomto případě použití bylo sledováno, zda intenzita šumu, kterým je trénovací obrázek poškozen, ovlivní délku prediktorů použitých k ohodnocení kandidátního řešení. Pro porovnání uvažujme 3 úlohy, ve kterých využijeme koevoluci pro návrh obrazového filtru obrázků s 10 %, 50 % a 80 % pixely poškozených šumem typu sůl a pepř. Na sérii grafů 5.3, resp. 5.4 si lze jejich porovnáním všimnout jisté samovolné adaptace, kterou na koevoluci vyvíjí samotný druh úlohy.

Horizontální osa zleva doprava představuje postupující čas, nebo-li počet uplynulých generací. Na ose y je umístěno procentuální zastoupení trénovacích vektorů, které určuje aktuálně používaný prediktor pro ohodnocení kandidátních řešení. V sérii grafů 5.3 lze vidět, že při použití obrázku s deseti procentním šumem se v průběhu návrhu koevoluce uchyluje k použití většího počtu trénovacích vektorů (např. 20 % - 25 %) častěji, než pro návrh obrazového filtru pro obrázek horší kvality, s 50 % výskytem poškozených pixelů. U 80 % šumu typu sůl a pepř nepřesáhne predikovaná sada trénovacích vektorů 5 %.

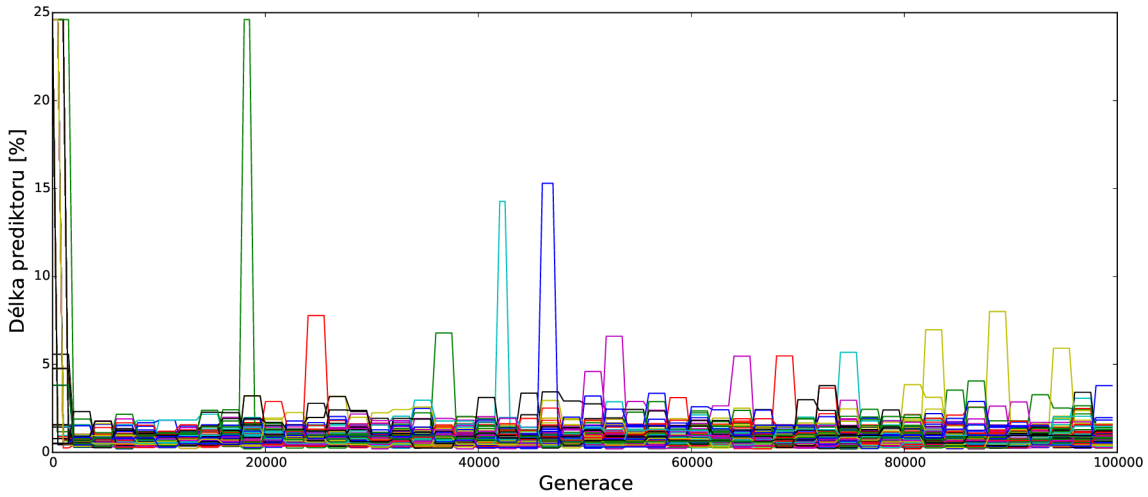
Všimněme si stejné tendence, zhodnotíme-li statistiku, kterou zobrazuje obrázek 5.4, pro tentokrát formou krabicových grafů. Rychlost konvergence roste s rostoucím poškozením trénovacího obrázku, s čímž klesá také medián počtu použitých trénovacích vektorů potřebných pro ohodnocení kandidátních řešení. Výsledky pro šum s jinou než dosud zmíněnou intenzitou lze dohledat v příloze C.1, popř. stručněji v tabulce 5.4.

Intenzita šumu	10 %	20 %	30 %	40 %	50 %	60 %	70 %	80 %
Medián	1.529	1.236	0.949	0.865	0.735	0.708	0.719	0.652
Průměr	4.160	2.353	1.263	1.055	0.973	0.892	0.941	0.8483

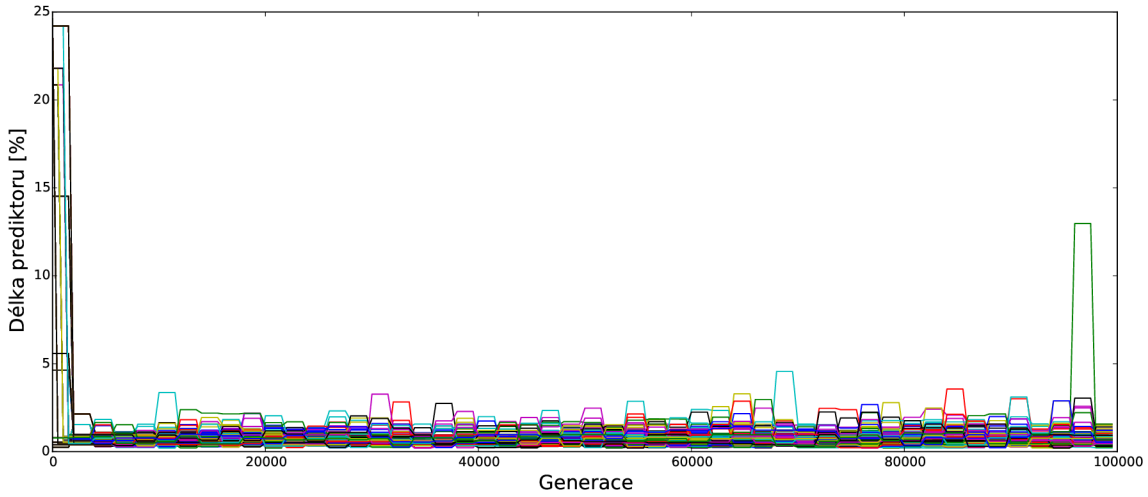
Tabulka 5.4: Závislost délky prediktoru na poškození obrázku pro šum typu sůl a pepř.



(a) 10% šum sůl a pepř

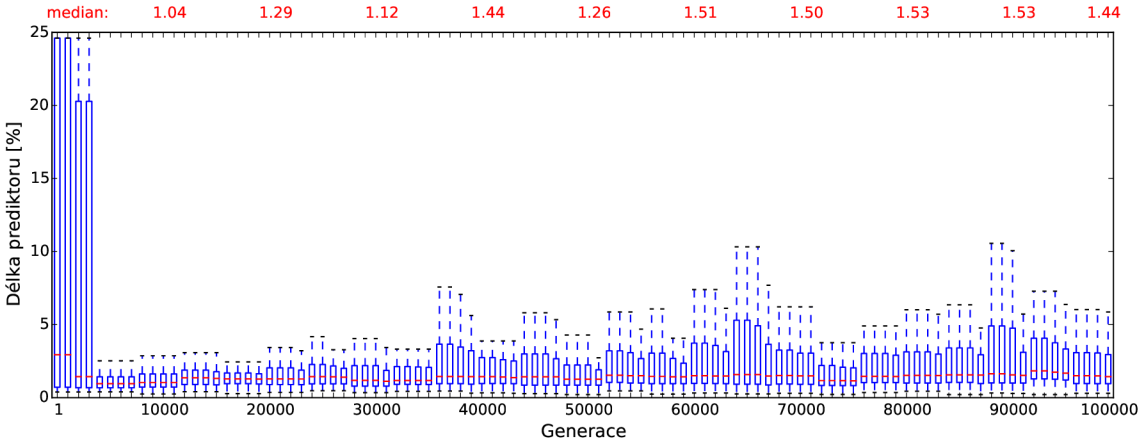


(b) 50% šum sůl a pepř

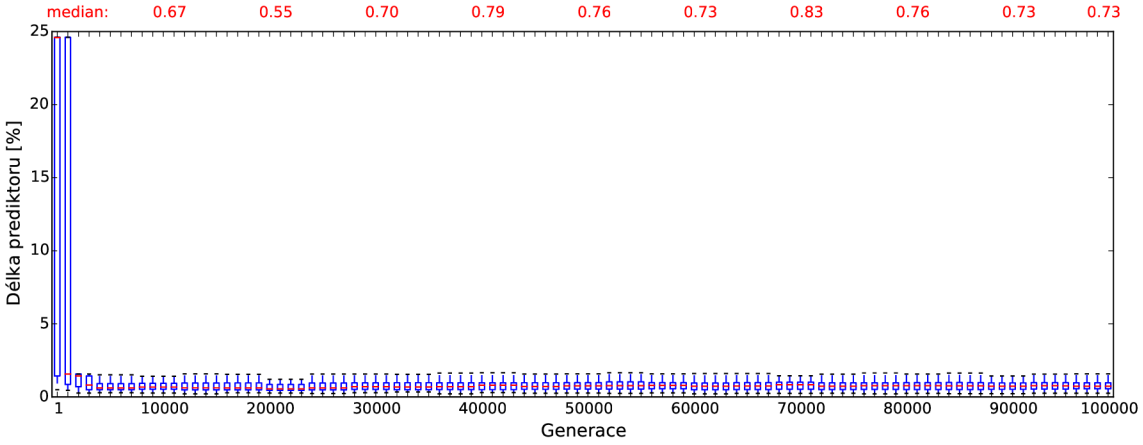


(c) 80% šum sůl a pepř

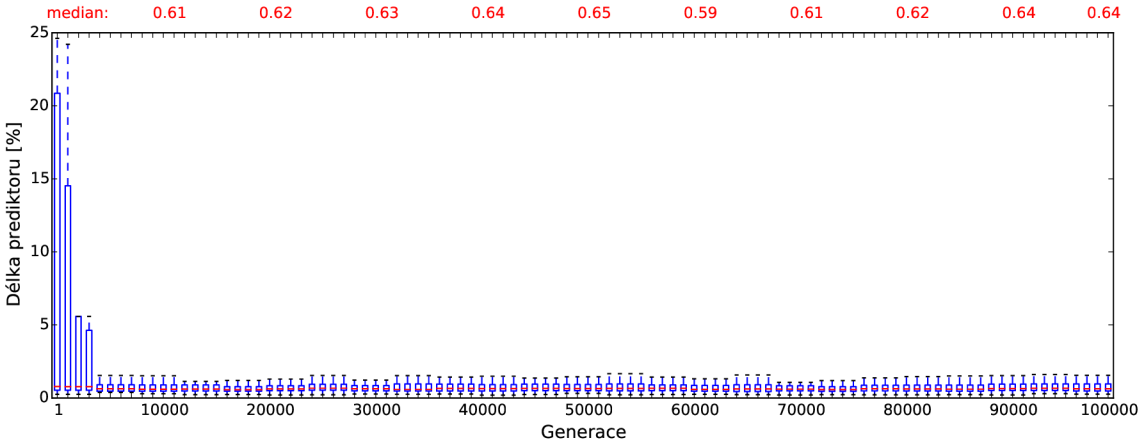
Obrázek 5.3: Vývoj délky prediktorů – průběh délky všech běhů pro šum typu sůl a pepř.



(a) 10% šum sůl a pepř



(b) 50% šum sůl a pepř



(c) 80% šum sůl a pepř

Obrázek 5.4: Vývoj délky prediktorů – statistické vyhodnocení pro šum typu sůl a pepř.

Náhodný impulzní šum

Sérii úloh pro impulzní šum s náhodnými hodnotami pixelů shrnují grafy podobným způsobem, jako tomu bylo u předcházející série.

Série grafů 5.6 ukazuje úbytek použití většího množství trénovacích vektorů (řekněme nad 15 % celé trénovací sady) při zvyšujícím se počtu poškozených pixelů. V extrémním zkoumaném případě, tj. 80 % poškozených pixelů se již nevyskytují žádné případy použití většího počtu trénovacích vektorů. Průběh délky prediktorů z návrhu filtru pro ostatní zkoumané intenzity šumu si můžete prohlédnout v příloze C.1.

Alternativní zobrazení těchto dat na obrázku 5.7 lépe osvětluje hodnoty počtu použitých trénovacích vektorů v nižších hodnotách. V případě 80 % šumu je délka prediktorů přibližně poloviční oproti prediktorům, které koevoluce používá při návrhu obrazového filtru pro 10 % impulzní šum.

Tabulkové shrnutí jednoznačně ukazuje klesající tendenci velikosti predikované sady se zvyšujícím se podílem poškozených pixelů v trénovacím obrázku. Přesto, že se jedná o složitější úlohu v porovnání s návrhem filtru pro sůl a pepř, průměr počtu trénovacích vektorů je nižší. Složitost úlohy se typicky negativně podepisuje na kvalitě navržených filtrů. Délku prediktorů ovlivňuje tato úloha spíše pozitivně, zřejmě koevoluce dokázala dospět k závěru, že ani s vyšším počtem trénovacích vektorů nedokáže dospět k lepšímu řešení úlohy.

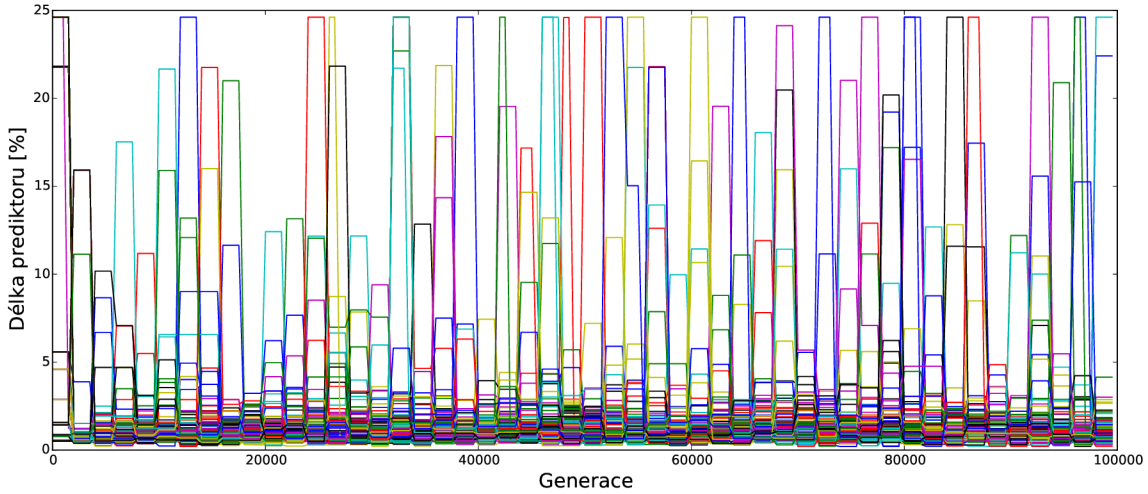
Prakticky je nižší délka prediktorů způsobena tím, že PSNR mezi originálním obrázkem a obrázkem poškozeným je vyšší i v případě vyššího počtu poškozených pixelů oproti úloze pro šum sůl a pepř. Proto také budou prediktory vyhodnoceny jako lepší z pohledu predikce trénovací sady a koevoluce může beztrestně soustředit svoji pozornost na snižování délky prediktorů (tj. kratší prediktory uvažovat jako kvalitnější). Pro ilustraci přetrvávajícího poškození i přes zvýšení hodnoty PSNR slouží obrázek 5.3.

Intenzita šumu	10 %	20 %	30 %	40 %	50 %	60 %	70 %	80 %
Medián	1.047	0.964	0.924	0.870	0.807	0.691	0.609	0.555
Průměr	1.802	1.398	1.153	1.119	1.033	0.883	0.822	0.673

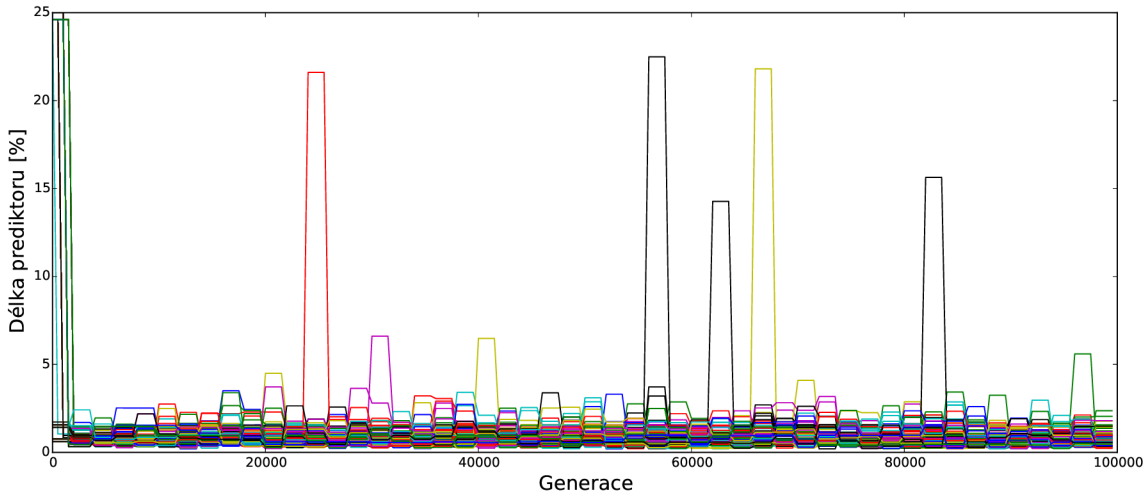
Tabulka 5.5: Závislost délky prediktoru na poškození obrázku – náhodný impulzní šum.

(a) PSNR = 16,408 dB (b) PSNR = 23,915 dB

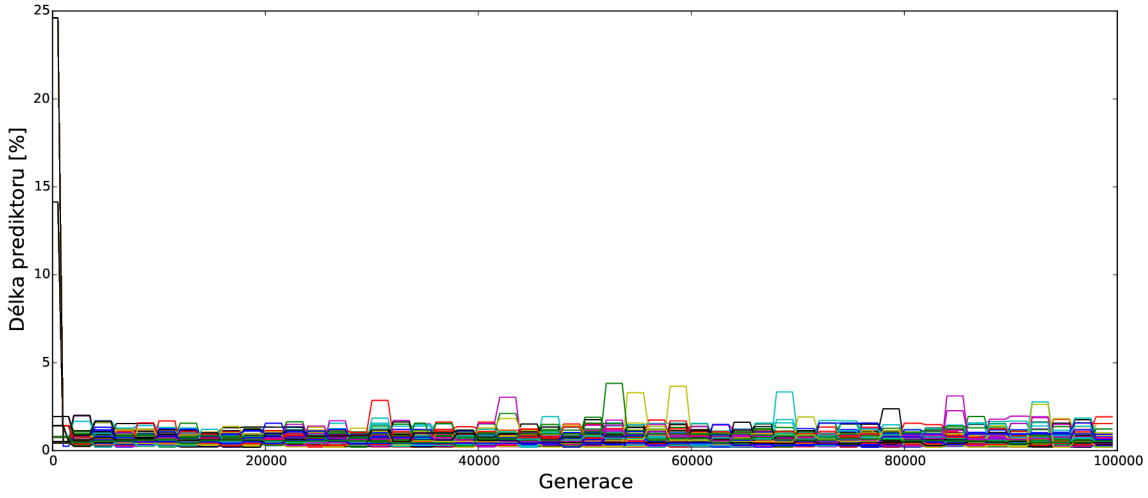
Obrázek 5.5: Poškozený obrázek (a) s 30% šumem a jeho rekonstruovaná verze (b).



(a) 10% impulzní náhodný šum

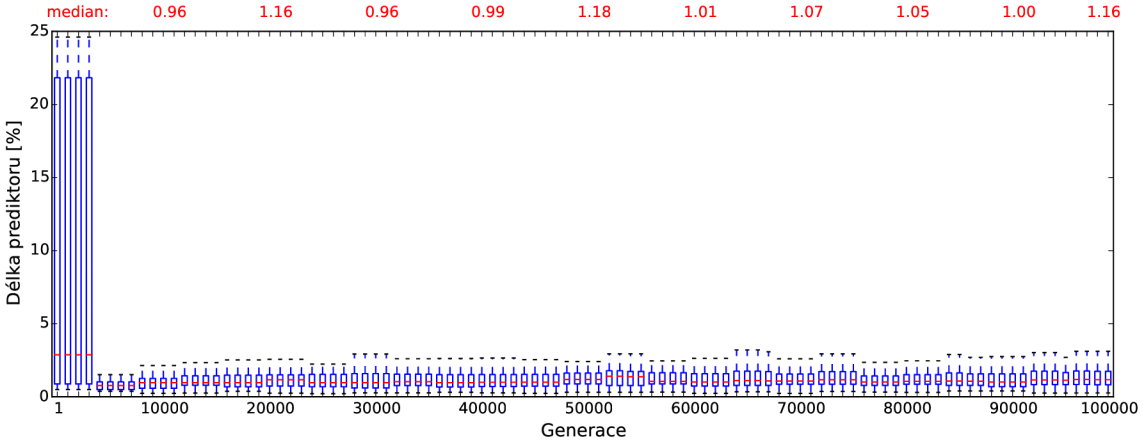


(b) 50% impulzní náhodný šum

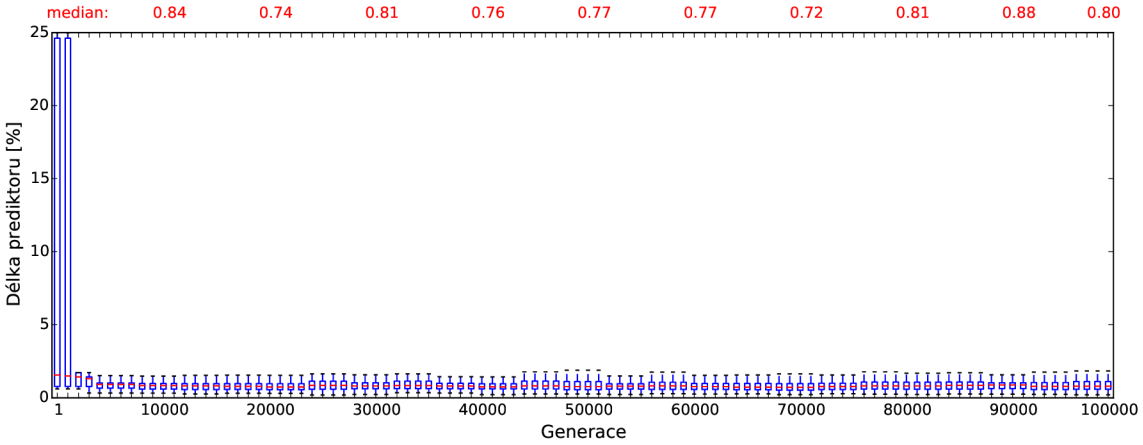


(c) 80% impulzní náhodný šum

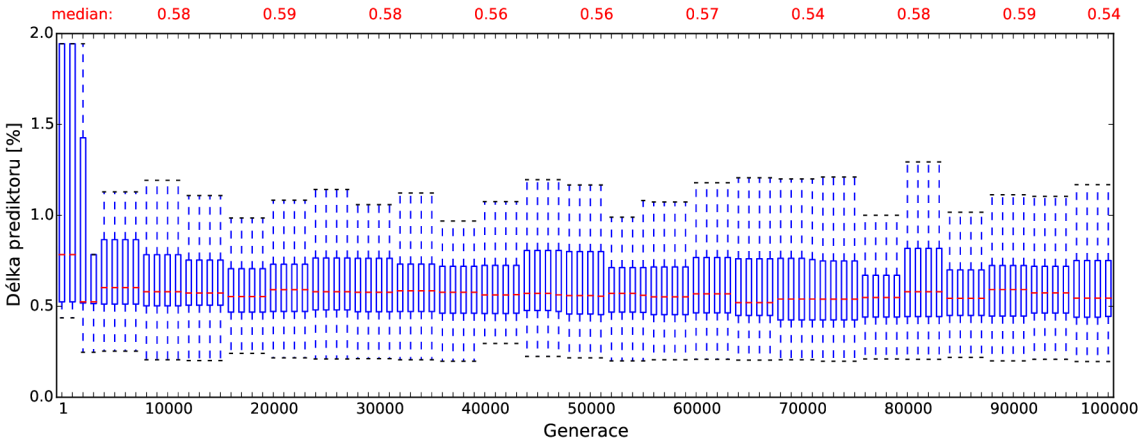
Obrázek 5.6: Vývoj délky prediktorů – průběh všech běhů pro náhodný impulzní šum.



(a) 10% impulzní náhodný šum



(b) 50% impulzní náhodný šum



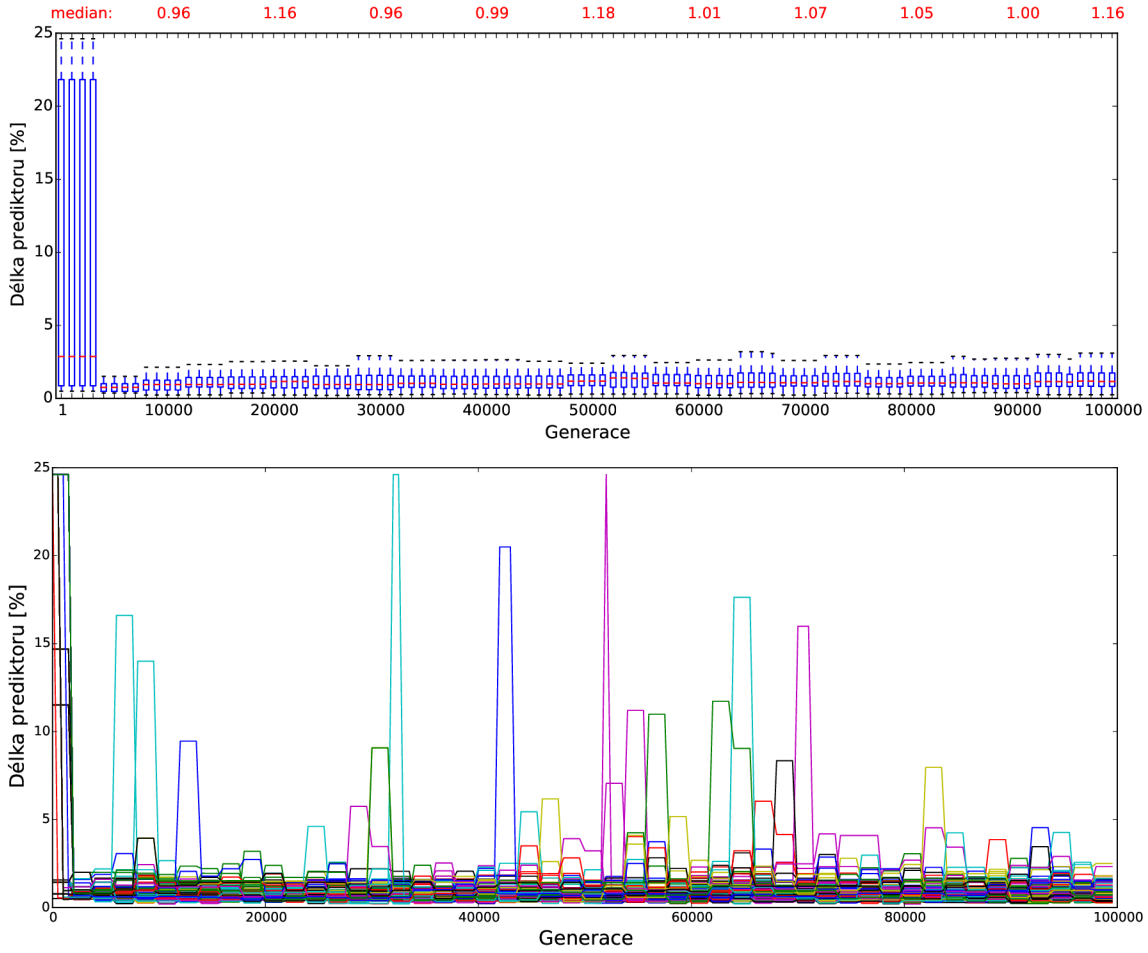
(c) 80% impulzní náhodný šum

Obrázek 5.7: Vývoj délky prediktorů – statistické vyhodnocení pro náhodný impulzní šum.

Dekektor hran

Detektor hran je odlišná úloha, přesto, nebo spíše právě proto je vhodné vyzkoušet návrh i tohoto obrazového filtru. V tomto případě nebudeme porovnávat nějakou skupinu úloh, jako tomu bylo u dvou předchozích kategorií. Jedná se o jednu úlohu, jejíž statistické výsledky zobrazují grafy na obrázku 5.8.

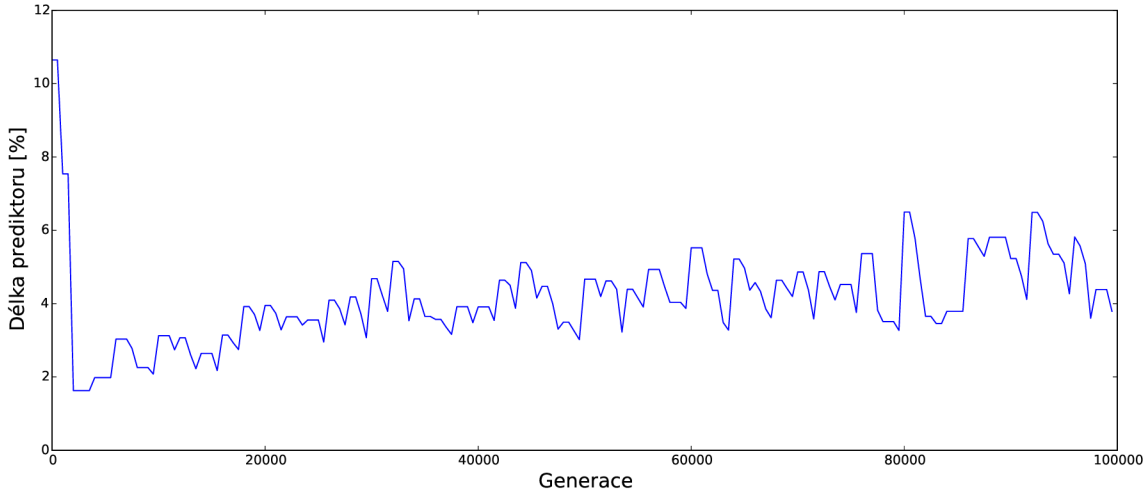
Z grafů lze vyčíst jistou adaptaci metody na nový typ úlohy. V mediánu se počet trénovacích vektorů použitých pro ohodnocení kandidátního řešení pohybuje okolo jednoho procenta, časté jsou ovšem i případy, kdy je použit prediktor délky někde v místech odlehlých hodnot, přes 5 %, 10 % nebo i 15 %.



Obrázek 5.8: Vývoj délky prediktorů pro návrh detektoru hran.

Shrnutí

V této části, kde jsme svou pozornost zaměřovali na změnu délky prediktorů během koevolučního návrhu, bylo zhodnoceno celkem 17 úloh. Výsledná statistická data ukázala, že velikost predikované testovací sady se pro rozdílné úlohy adaptuje. K adaptaci dochází bez jakékoli změny nastavení koevolučního návrhu, pouze se zasazením do daného kontextu ve smyslu předání trénovacího vstupu a požadovaného výstupu. U obou skupin úloh navrhujících filtry pro poškozené obrázky impulzním šumem se při zvyšující se intenzitě šumu



Obrázek 5.9: Průběh průměrné délky prediktorů při návrhu obrazového filtru 10% šumu typu sůl a pepř.

rychlost konvergence k optimální délce prediktoru zvyšovala. Délka prediktoru samotná se s přibývajícím šumem snižuje.

Zajímavý jev, který nelze opomenout při shrnutí této kapitoly zobrazuje graf na obrázku 5.9. Jedná se o průměrnou délku prediktoru v průběhu koevoluce. Na první pohled se může zdát, že se délka mění náhodně. Že tomu tak není, bylo ukázáno dříve. Zajímavé jsou lokální prudké nárůsty délky prediktoru, následované pozvolnějším klesáním. Koevoluce v okamžiku prodlužování prediktoru dospěla k závěru, že při zvýšení jeho délky dojde ke zvýšení kvality kandidátního řešení. Nedovolila to na dlouho, pozvolným snižováním jeho délky eliminovala trénovací vektory, které ve skutečnosti nebyly k ohodnocení kandidátního řešení potřeba.

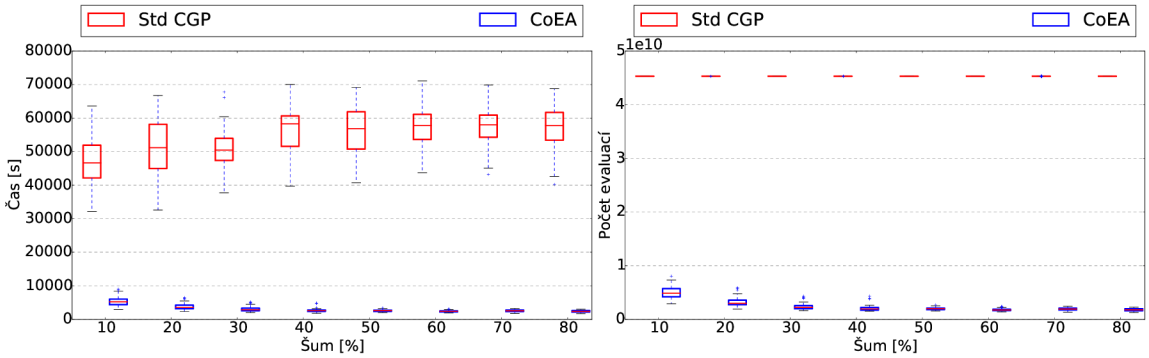
V globálním měřítku délka prediktoru jednoznačně stoupá. Nejvíce razantní vzestup lze vidět mezi generací přibližně 100 až 3 000, čili v počátečním stádiu návrhu, průměrná délka ovšem roste i dále. Na počátku koevoluce totiž k eliminaci kandidátních řešení s příliš malou fitness, čili ke zjištění jejich nízké kvality dostačuje malé množství trénovacích vektorů. V průběhu narůstajícího počtu generací stoupá také fitness kandidátních řešení a k rozeznání méně kvalitních jedinců od těch s vyšší schopností plnit danou úlohu, koevoluce potřebuje počítat fitness užitím většího počtu trénovacích vektorů. Prediktory se tedy v průměru zvyšují s počtem generací a s narůstající kvalitou kandidátních řešení.

5.4 Porovnání se standardním CGP

Podstatnou částí experimentálního vyhodnocení metody je porovnání s původním přístupem, tj. s evolučním návrhem obrazových filtrů používajícím standardní CGP. Základní otázky jsou: „Urychlí navrhovaný přístup návrh obrazových filtrů? Pokud ano, tak jakým způsobem? Nesníží se při případném zrychlení kvalita výsledného řešení?“ Tyto otázky byly položeny na třech již zmíněných sériích úloh – filtr pro šum sůl a pepř, náhodný impulzní šum a detektor hran. V následujícím textu budou analyzovány výsledné hodnoty, tj. po uběhnutí 100 000 generací. V příloze C.2 jsou však dostupné i grafy zhodnocující průběh návrhu – tzn. po 30 000 a 50 000 generacích koevolučního a evolučního návrhu.

Sůl a pepř

První série případové studie se skládá z osmi úloh pro obrazové filtry různé intenzity šumu typu sůl a pepř. Jak ukazuje graf na obrázku 5.10a, pro všechny tyto úlohy od intenzity od 10 % do 80 % poškození odstupňované po 10 % byla rychlost řešení silně redukována. Zrychlení se v průměru pohybuje v případě úlohy pro 10% sůl a pepř od devítinásobku až po 24-násobné zrychlení v případě 60 % a 80 % šumu.



(a) Uživatelský čas.

(b) Počet evaluací.

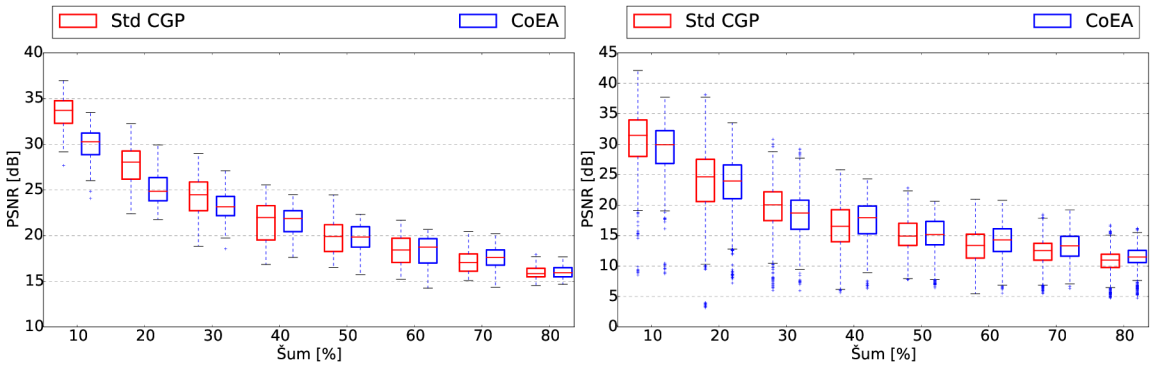
Obrázek 5.10: Srovnání doby trvání (a) a počtu evaluací (b) koevolučního návrhu vůči standardnímu CGP v úloze sůl a pepř.

Zmíněného zrychlení bylo dosaženo díky redukci počtu výpočtů výpočetních bloků tvořících jak kandidátní řešení, tak prediktory. Počet těchto výpočtů byl měřen zvlášť pro každou dílčí evoluci a při statistickém zhodnocení sečten. I tak byla redukce velká, což dokazuje graf 5.10b. Počet těchto výpočtů skutečně koresponduje s naměřeným časem. U standardního CGP se počet výpočtů pohybuje okolo 45.3×10^9, zatímco u koevolučního návrhu to je od 1.8×10^9 do 5×10^9. Snížení počtu evaluací výpočetních bloků se pohybuje mezi devíti až 25 násobkem původního počtu evaluací, obdobně jako tomu je v případě uživatelského času.

Na závěr zbývá zásadní otázka, a to, zda při takovém zrychlení návrhu lze dosáhnout stejné kvality výsledného produktu, tedy obrazového filtru. Odpověď odkrývá graf v obrázku 5.11a . U úlohy s deseti a dvaceti procentním šumem je kvalita horší o 3.5 a 2.7 dB PSNR, u šumu 30 procentního je zhoršení už jen o 1 dB. Úlohy složitější však vykazují konečná řešení stejné (50 % a 60 % šum) nebo lepší kvality (u 40 %, 70 % a 80 % šumu).

Kvalita navrženého řešení byla následně změřena na sérii testovacích obrázků dané úlohy. Výkonnost navrženého řešení oběma metodami (tj. standardní CGP a koevoluční návrh s prediktory) srovnává obrázek 5.11b. Toto srovnání vypadá ještě pozitivněji pro vyhodnocovanou metodu koevolučního návrhu. Lepší obrazové filtry nalezla koevoluce pro obrázky od 40 % šumu výše. Pro 20 % šum je koevoluční řešení v průměru pouze o 0.3 dB horší.

Tabulka 5.4 zobrazuje srovnání metod v číslech se zvýrazněným vylepšením průměrné kvality. V tabulce se také objevují hodnoty pro koevoluční přístup k návrhu obrazových filtrů s pevnou, pětiprocentní délkou prediktoru celkové trénovací sady. Jelikož se jedná o přejaté hodnoty z odlišné implementace, nemá smysl srovnávat uživatelský čas běhu koevoluce. Proto pro tuto metodu sloupec času neuvádím.



(a) Výsledná fitness.

(b) Testování kvality.

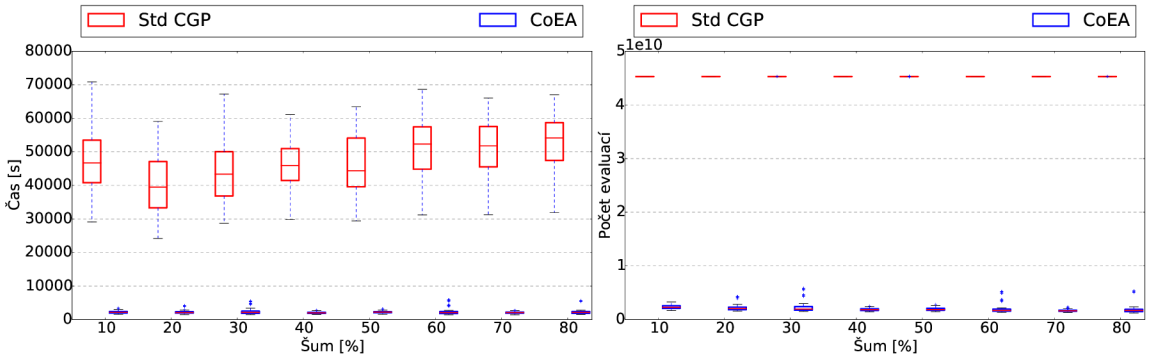
Obrázek 5.11: Srovnání fitness výsledného řešení (a) a kvality ze sady testovacích obrázků (b) v koevolučním návrhu a standardním CGP v úloze sůl a pepř.

		PSNR [dB]			Evaluace [$\times 10^9$]			Čas [min]	
Intenzita šumu		CGP	CoEA 5 %	ACoEA	CGP	CoEA 5 %	ACoEA	CGP	ACoEA
10 %	Min	27.7	22.7	24.1	45.3	19.3	2.9	534.9	48.5
	Průměr	33.4	30.1	29.9	45.3	22.6	5.0	793.5	88.4
	Max	36.9	35.1	33.5	45.3	29.9	8.1	1060.1	149.9
20 %	Min	22.4	17.8	21.8	45.3	19.9	1.9	543.4	39.2
	Průměr	27.8	24.7	25.1	45.3	23.9	3.2	853.4	62.9
	Max	32.3	31.5	30.0	45.3	34.0	5.9	1112.1	58.6
30 %	Min	18.8	16.8	18.5	45.3	19.2	1.6	629.0	33.9
	Průměr	24.2	20.6	23.1	45.3	25.6	2.4	847.1	50.6
	Max	28.9	25.9	27.1	45.3	37.8	4.3	1129.4	47.5
40 %	Min	16.8	14.8	17.6	45.3	19.7	1.5	661.3	29.9
	Průměr	**21.5**	19.2	**21.6**	45.3	27.3	2.1	934.1	44.7
	Max	21.9	23.7	24.5	45.3	38.9	4.3	1167.7	81.0
50 %	Min	16.5	15.2	15.7	45.3	20.8	1.5	678.3	33.1
	Průměr	**19.8**	24.6	**19.8**	45.3	27.6	1.9	940.7	42.8
	Max	24.4	18.0	22.3	45.3	47.7	2.7	1151.9	56.3
60 %	Min	15.2	14.7	14.2	45.3	16.1	1.4	728.5	31.8
	Průměr	**18.4**	17.3	**18.4**	45.3	29.7	1.8	955.5	39.5
	Max	21.7	21.1	20.7	45.3	65.0	2.4	1185.3	51.6
70 %	Min	15.1	14.9	14.3	45.3	19.8	1.3	719.9	29.2
	Průměr	**17.1**	16.4	**17.5**	45.3	29.5	1.9	955.9	42.5
	Max	20.4	18.9	20.4	45.3	58.0	2.4	1164.2	53.7
80 %	Min	14.1	14.9	14.7	45.3	20.7	1.3	670.0	28.2
	Průměr	**15.9**	15.7	**16.0**	45.3	29.3	1.8	955.2	39.8
	Max	17.9	17.0	17.7	45.3	63.2	2.2	1146.2	49.9

Tabulka 5.6: Srovnání statistických údajů v úloze sůl a pepř pro různé intenzity šumu (CGP – Standardní CGP, CoEA 5 % – Koevoluční algoritmus s pevnou 5% délkou prediktorů, ACoEA – Koevoluce s adaptivní délkou nepřímo kódovaných prediktorů).

Náhodný impulzní šum

Dobu návrhu v kontextu s počtem evaluací u osmičlenné série úloh výstřelového šumu zobrazují grafy na obrázku 5.12. Zrychlení je opět patrné pouhým pohledem na hodnoty zobrazené formou krabicových grafů. Porovnáním uživatelských časů spotřebovaných na výpočet při návrhu bylo zjištěno, že se zrychlení v průměru pohybuje od 17.3 násobku (30% šum) až po 24.8 násobek (70% šum). Z pohledu počtu vyhodnocení se u stejných úloh jedná o zrychlení mezi 18.9-26.6 násobkem oproti původnímu řešení implementovanému na principech standardního CGP.



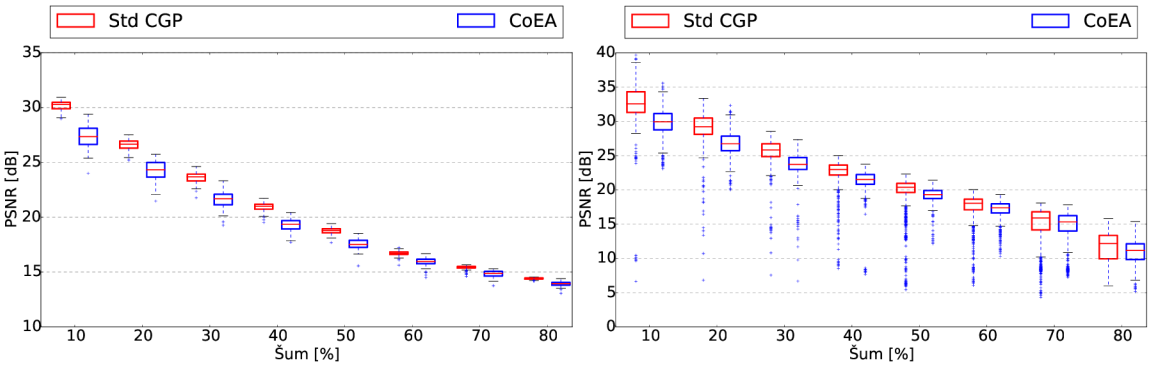
(a) Uživatelský čas.

(b) Počet evaluací.

Obrázek 5.12: Srovnání doby trvání (a) a počtu evaluací (b) koevolučního návrhu vůči standardnímu CGP v úloze výstřelového šumu.

U tohoto typu úloh měla koevoluce (resp. populace prediktorů) tendenci velmi snižovat počet prediktorů. Je to dáno tím, že pokud mají poškozené pixely hodnotu v celém rozmezí definičního oboru, PSNR udává vyšší hodnoty oproti šumu sůl a pepř při stejné intenzitě šumu (jak bylo demonstrováno obrázkem 5.3). Také se mohlo stát, že určité množství náhodných hodnot poškozených pixelů trénovacího obrázku bylo příliš blízko originálním hodnotám. Kvůli některému z těchto faktorů, popř. jejich kombinaci pak evoluce prediktorů ohodnocovala i příliš krátké prediktory jako kvalitní, což ve výsledku nedovolilo evoluci kandidátních řešení dosáhnout obrazového filtru se stejnou kvalitou jako standardní CGP.

Graf 5.13a zobrazuje výsledek porovnání standardního CGP oproti koevolučnímu návrhu, který více přibližuje tabulka 5.4. Nejzajímavější výsledky nalezneme ve vysokých hodnotách šumu (80 %), kde výsledek koevoluce zaostává oproti standardnímu CGP v průměru o pouhých 0.4 dB. Na obrázku 5.13b je graf porovnávající výsledek získaný z aplikace vyvinutých filtrů na sérii devíti testovacích obrázků každé úlohy impulzního šumu. Rozdíl se u testovacích obrázků ještě snížil, u poškození nad 50 % nepřesáhne rozdíl mezi srovnávanými řešeními v průměru 0.5 dB (70 % šumu je rozdíl u trénovacích obrázků v průměru pouhých 0.1 dB a u 50 % 0.2 dB).



(a) Výsledná fitness.

(b) Testování kvality.

Obrázek 5.13: Srovnání fitness konečného řešení (a) a kvality přes testovací obrázky (b) koevolučního návrhu a standardního CGP v úloze výstřelového šumu.

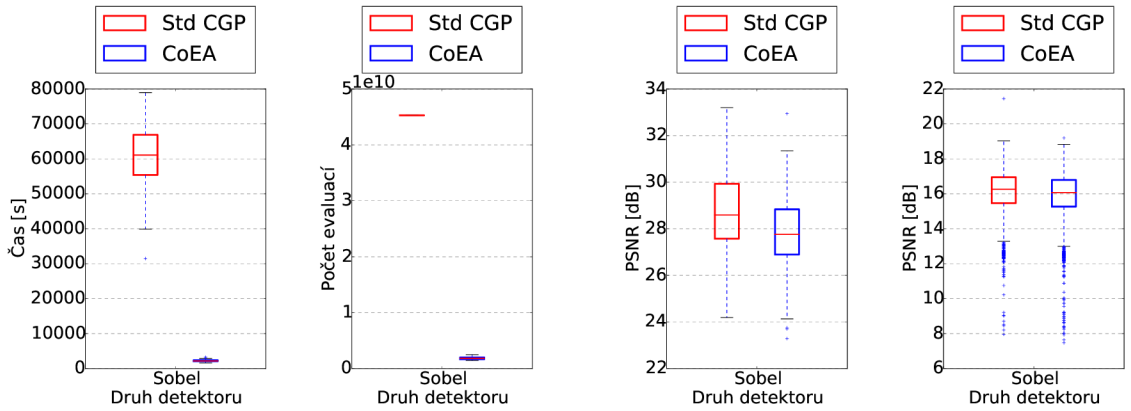
		PSNR [dB]			Evaluace [$\times 10^9$]			Čas [min]	
Intenzita šumu		CGP	CoEA 5 %	ACoEA	CGP	CoEA 5 %	ACoEA	CGP	ACoEA
10 %	Min	28.9	23.8	24.0	45.3	3.8	1.7	485.3	26.0
	Průměr	30.2	28.9	27.3	45.3	34.3	2.3	795.2	37.4
	Max	30.9	30.3	29.4	45.3	13.9	3.3	1181.1	58.5
20 %	Min	25.2	20.9	21.5	45.3	5.3	1.6	403.6	25.1
	Průměr	26.6	25.7	24.2	45.3	23.5	2.2	686.5	37.3
	Max	27.5	27.1	25.7	45.3	36.5	4.3	984.5	70.5
30 %	Min	21.7	19.1	19.2	45.3	5.2	1.5	478.1	24.4
	Průměr	23.6	23.2	21.6	45.3	19.7	2.4	728.7	42.2
	Max	24.6	24.9	23.3	45.3	41.5	5.9	1121.1	93.1
40 %	Min	19.5	18.6	17.7	45.3	10.1	1.4	496.9	25.6
	Průměr	20.9	20.7	19.3	45.3	24.0	1.9	767.2	33.4
	Max	21.7	22.5	20.4	45.3	30.9	2.5	1018.0	47.2
50 %	Min	17.6	17.1	15.5	45.3	8.9	1.5	490.1	26.3
	Průměr	18.7	18.6	17.6	45.3	34.5	1.9	774.2	36.9
	Max	19.4	20.1	18.5	45.3	47.3	2.8	1058.5	54.8
60 %	Min	15.6	15.5	14.5	45.3	10.1	1.4	520.1	24.3
	Průměr	16.7	16.9	15.9	45.3	37.0	2.1	853.3	40.9
	Max	17.2	17.6	16.7	45.3	57.1	5.2	1144.2	98.8
70 %	Min	14.6	15.2	13.8	45.3	6.9	1.3	520.1	23.8
	Průměr	15.4	15.8	14.8	45.3	20.1	1.7	853.4	34.4
	Max	15.7	16.4	15.3	45.3	48.9	2.4	1101.2	46.4
80 %	Min	14.2	14.6	13.1	45.3	9.5	1.2	531.9	25.5
	Průměr	14.4	15.2	14.0	45.3	38.7	1.9	878.9	39.6
	Max	14.5	14.9	14.4	45.3	78.3	5.4	1117.0	95.4

Tabulka 5.7: Srovnání statistických údajů v úloze náhodný impulzní šum pro různé intenzity (CGP – Standardní CGP, CoEA 5 % – Koevoluční algoritmus s pevnou 5% délkou prediktorů, ACoEA – Koevoluce s adaptivní délkou nepřímo kódovaných prediktorů).

Detektor hran

Poslední úlohou, pro kterou byla nashromážděna statistická data, je návrh detektoru hran. Tato úloha nemá žádné další členění a tak jsou výsledky experimentů shrnuty na obrázku 5.14. Návrh na principech standardního CGP vykázalo 27.1 násobnou dobu trvání oproti koevolučnímu návrhu.

Kvalita je v této úloze také určena metodou PSNR, přičemž význam není takový, že bychom posuzovali čistotu obrázku, ale porovnáváme výstup s výstupem očekávaným, čili s obrázkem s detekovanými hranami užitím sobelova operátoru. Z koevoluce vyšel v průměru o 0.9 dB horší detektor hran oproti pomalejší verzi standardního CGP. Pro obecné zhodnocení byla použita sada osmi testovacích obrázků. V této fázi srovnání obě série vykazovaly naprosto srovnatelnou kvalitu produkovaného obrázku, což dokazuje graf 5.14d.



(a) Uživatelský čas. (b) Počet evaluací. (c) Výsledná fitness. (d) Kvalita testování.

Obrázek 5.14: Srovnání doby trvání (a), počtu evaluací (b), fitness konečného řešení (c) a kvality přes testovací obrázky (d) koevolučního návrhu vůči standardnímu CGP v úloze detektoru hran.

	PSNR [dB]		Evaluace [$\times 10^9$]		Čas [min]	
Detektor hran	CGP	ACoEA	CGP	ACoEA	CGP	ACoEA
Min	24.2	23.3	45.3	1.4	524.9	26.8
Průměr	28.6	27.7	45.3	1.8	1009.5	37.2
Max	33.1	32.9	45.3	2.5	1316.3	54.7

Tabulka 5.8: Srovnání statistických údajů v úloze návrhu detektoru hran (CGP – Standardní CGP, ACoEA – Koevoluce s adaptivní délkou nepřímo kódovaných prediktorů).

Kapitola 6

Závěr

Diplomová práce se zabývala automatizovaným návrhem spustitelných struktur, konkrétně obrazových filtrů, dosud neprozkoumaným způsobem. Nový přístup, na který byla zaměřena pozornost, byl postaven na principech koevoluce, ve které populace prediktorů fitness přejímala prvky symbolické regrese. Prediktory byly kódovány nepřímo, jako matematické funkce, prostřednictvím kterých je až posléze vypočítáno pole ukazatelů do trénovacích dat pro ohodnocení kandidátního řešení.

Na základě principiálního pochopení navržené metody mohlo být jasné, že proces návrhu by měl být zkrácen oproti předcházejícím, v textu zmíněným přístupům. Po části nastudování problematiky automatizovaného návrhu spustitelných struktur (programů) následoval návrh, jehož spustitelná verze byla následně vytvořena a otestována na 17 úlohách. K vyhodnocení metody sloužila data posbíraná z velkého množství experimentů (17 úloh se 100 běhy pro 2 přístupy, tj. celkem 3 400 experimentů) s koevolučním návrhem a evolučním návrhem (CGP) obrazových filtrů a následného výpočtu statistik pro testovací sadu obrázků (9 obrázků pro 100 navržených obrazových filtrů každé intenzity šumu a detektor hran pro oba přístupy po 10, 30 a 100 tisících generací, tj. celkem 91 800 vzorů pro statistické vyhodnocení).

Z vyhodnocení vyplynula velká časová redukce u všech zkoumaných úloh. Ve většině případů se srovnatelným výsledkem, v jiných s mírně horším výsledným produktem a v některých dokonce kvalitou převyšující standardní CGP. Za všechny můžeme zmínit nalezení obrazového filtru pro šum typu sůl a pepř o intenzitě 70 % se zlepšením 0.4 dB oproti standardnímu CGP (ze 17.1 na 17.5 dB) při zrychlení času návrhu průměrně 22.5 krát. Když nazveme úlohy, u kterých vykazovalo řešení na testovací sadě obrázků v průměru až o 1 dB lepší nebo horší kvalitu ve srovnání se standardním CGP jako úlohy úspěšné, pak takovýchto případů můžeme napočítat 14 ze 17 zkoumaných se současným šestnácti až sedmadvaceti násobným zrychlením automatického návrhu. Porovnání je však prováděno po určitém počtu generací evoluce kandidátních filtrů shodném pro evoluci i koevoluci. Jelikož použitím koevoluce bylo dosaženo významného zrychlení, lze například zdvojnásobit počet generací, po který koevoluce pracuje, a tím umožnit algoritmu delší dobu prohledávat. Koevoluci bychom tak umožnili nalézt kvalitnější řešení a to stále s významnou časovou úsporou oproti přístupu bez využití koevoluce.

Porovnání bylo provedeno také pro metodu koevoluce s pevnou délkou prediktoru, konkrétně s prediktorem velikosti 5 % trénovací sady. Tento přístup produkoval výsledky vyšší kvality při návrhu obrazového filtru pro impulzní šum s náhodnými hodnotami poškozených pixelů, ovšem za cenu menšího zrychlení oproti CGP než vykazovala zkoumaná metoda. V každém případě toto porovnání poukazuje na fakt, že v budoucnu bude vhodné upravit

průběh evoluce tak, aby ještě méně upřednostňovala prediktory malé délky před delšími, které ovšem povedou k lepším výsledkům, třeba i za cenu ne tak rapidního zrychlení návrhu. Experimenty by bylo vhodné rozšířit pro úlohy jiného typy, např. evolučního návrhu kombinačních obvodů. CGP je vhodně navrženo pro akceleraci v FPGA, a jelikož se obě populace vyvíjí na principech CGP, bylo by možné tento přístup akcelerovat jeho implementací do FPGA.

V této práci jsem si zopakoval a prohloubil znalosti z evolučních algoritmů, konkrétně jsem svoji pozornost směřoval ke zdokonalení znalosti evolučního návrhu obrazových filtrů aplikací kartézského genetického programování a jeho urychlení užitím koevoluce. Při realizaci programové implementace jsem nabyl nových znalostí a schopností při práci s technologiemi C++, OpenMP, Java, Bash a jeho adaptací pro potřeby spuštění experimentů na superpočítači Anselm a statistického zpracování velkého množství výstupů experimentů automatizovaně skripty v jazyku Python.

Literatura

[1] Hillis, W. D.: Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, ročník 42, č. 1, 1990: ISSN 0167-2789, s. 228–234.

[2] Hrbáček, R.; Šikulová, M.: Coevolutionary Cartesian Genetic Programming in FPGA. In *Advances in Artificial Life, ECAL 2013, Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems*, MIT Press, 2013, ISBN 978-0-262-31709-2, s. 431–438.

[3] Hulva, J.: Koevoluční algoritmus pro úlohy založené na testu. 2014, diplomová práce FIT VUT.

[4] Husbands, P.; Mill, F.: Simulated Co-Evolution as The Mechanism for Emergent Planning and Scheduling. In *Proceedings of the 4th International Conference on Genetic Algorithms*, 1991, ISBN 1-55860-208-9, s. 264–270.

[5] Miller, J.: Cartesian Genetic Programming. Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7, s. 17–34.

[6] Miller, J.: Introduction to Evolutionary Computation and Genetic Programming. In *Cartesian Genetic Programming*, editace J. F. Miller, Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7, s. 1–16.

[7] Popovici, E.; Bucci, A.; Wiegand, R. P.; aj.: Coevolutionary principles. In *Handbook of Natural Computing*, Springer, 2012, ISBN 978-3-540-92909-3, s. 987–1033.

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

[9] Sekanina, L.; Harding, S.; Banzhaf, W.; aj.: Image Processing and CGP. In *Cartesian Genetic Programming*, editace J. F. Miller, Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7, s. 181–215.

[10] Sekanina, L.; Vašíček, Z.; Růžička, R.; aj.: *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Edice Gerstner, Academia, 2009, ISBN 978-80-200-1729-1, 328 s.

[11] Šikulová, M.; Hulva, J.; Sekanina, L.: Indirectly Encoded Fitness Predictors Coevolved with Cartesian Programs. In *Genetic Programming*, LNCS 9025, Springer International Publishing, 2015, ISBN 978-3-319-16500-4, s. 113–125.

[12] Šikulová, M.; Komjáthy, G.; Sekanina, L.: Towards Compositional Coevolution in Evolutionary Circuit Design. In *2014 IEEE International Conference on Evolvable Systems Proceedings*, Institute of Electrical and Electronics Engineers, 2014, ISBN 978-1-4799-4479-8, s. 157–164.

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

[14] Šikulová, M.; Sekanina, L.: Coevolution in Cartesian Genetic Programming. In *Genetic Programming*, *Lecture Notes in Computer Science*, ročník 2012, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-29138-8, s. 182–193.

Příloha A

Obsah CD

Strukturu souborů a adresářů, které naleznete na přiloženém CD popisuje následující seznam.

- `Aplikace/`
  - `Koevoluční algoritmus/` ... Přeložená implementace koevolučního algoritmu, ukázkové skripty a obrázky pro možnost okamžitého spuštění.
  - `Obrazový filtr/` ... Přeložená implementace obrazového filtru, ukázkové skripty, chromozomy a obrázky pro možnost okamžitého spuštění.
- `Technická zpráva/`
  - `tex/` ... Zdrojové soubory technické zprávy.
  - `Technická zpráva.pdf` ... Technická zpráva diplomové práce ve formátu pdf.
- `Zdrojové kódy/`
  - `Koevoluční algoritmus/` ... Zdrojové kódy v jazyku C++ implementace koevolučního algoritmu, Makefile, skripty a adresář s obrázky pro spuštění.
  - `Obrazový filtr/` ... Zdrojové kódy v jazyku Java implementace obrazového filtru, skripty a adresář s obrázky a chromozomy pro spuštění.
- `ReadMe.pdf` ... Podrobný popis obsahu CD a uživatelská příručka programů.

Příloha B

Manuál

B.1 Koevoluční algoritmus

Implementace koevolučního algoritmu byla použita k provádění experimentů, jejichž výsledky vedly ke statistickému vyhodnocení metody.

Překlad programu

Přiložen je soubor `Makefile`, jehož použitím lze program přeložit. Překlad je nutno provést s gcc verze 4.3 a vyšší.

Ovládání

Následující seznam shrnuje přepínače programu, kterými lze měnit parametry evolučního a koevolučního návrhu.

- `-to FILENAME` ... Určuje cestu k originálnímu obrázku.
- `-tc FILENAME` ... Určuje cestu k poškozenému obrázku.
- `-o FILENAME` ... Cesta k souboru, do kterého může uživatel uložit výstupní průběžné informace programu (bez zadání parametru budou vypsány pouze do konzole).
- `-d NOISE` ... V případě, že uživatel chce, aby program poškozený obrázek dopočítal, tímto parametrem zadá míru poškození.
- `-sp` ... Používá se pouze v kombinaci s přepínačem `-d` pro destrukci obrázku šumem typu sůl a pepř.
- `-rn` ... Používá se pouze v kombinaci s přepínačem `-d` pro destrukci obrázku náhodným impulzním šumem.
- `-stats CORE` ... Kořen názvu souboru pro ukládání statistických dat.
- `-rounds NUMBER` ... Počet kol, který má být v rámci jednoho spuštění programu proveden.
- `-gnrts NUMBER` ... Počet generací (ko)evoluce.

- `-nocoea` ... Slouží k vypnutí koevoluce, program pak bude spouštět standardní sekvenční CGP.
- `-chpoint FILENAME` ... Cesta k souboru, kam ukládat, popř. odkud číst data průběžné konfigurace standardního sekvenčního CGP pro případ zotavení se z předčasného ukončení programu.
- `-csv FILENAME` ... Cesta k místu, kam mají být uloženy statistické údaje ve formě tří souborů formátu csv.

Pokud chce uživatel například spustit 50 kol koevolučního návrhu s dvojicí testovacích obrázků `sp10.png` a `orig.png` po dobu 50 000 generací a ukládat statistická data s kořenem názvu `sp10`, zadá:

```
./coea -tc sp10.png -to orig.png -gnrts 50000 -rounds 50 -csv sp10.
```

Pro provedení 30 kol sekvenčního návrhu s 50 000 generacemi při použití originálního obrázku `orig.png` s tím, že uživatel požaduje automatické poškození náhodným impulzním šumem intenzity 40 % a ukládání průběžné konfigurace do souboru `checkpoint`, ať program spustí takto:

```
./coea -to orig.png -d 40 -rn -rounds 30 -gnrts 50000 -chpoint checkpoint.
```

B.2 Filtr

Implementace obrazového filtru sloužila k ohodnocení metody na sérii testovacích obrázků.

Překlad programu

Program je přiložen i s metadaty pro IDE NetBeans, pro komfortní prohlížení kódu a překlad je tedy možno otevřít projekt v tomto IDE. V příkazovém řádku s nainstalovanou knihovnou Appache Ant lze provést překlad užitím příkazu `ant jar`.

Ovládání

Program filtr lze spustit s následujícími parametry.

- `-ch FILENAME` ... Cesta k souboru s chromozomem kódujícím obrazový filtr.
- `-oi FILENAME` ... Cesta k originálnímu obrázku.
- `-ci FINENAME` ... Cesta k poškozenému obrázku.
- `-outf FILENAME` ... Cesta k souboru pro zápis výstupních dat.
- `-outi FILENAME` ... Umístění pro uložení filtrovaného obrázku.

Například, pokud uživatel hodlá filtrovat obrázek `sp80.png` filtrem zakódovaným v chromozomu `chromozom.chr` a na závěr kvalitu filtru zhodnotit užitím obrázku `original.png` a výsledek tohoto porovnání uložit do souboru `result`, může program spustit následovně:

```
java -jar FilterIt.jar -ci sp80.png -oi original.png -outf result.
```

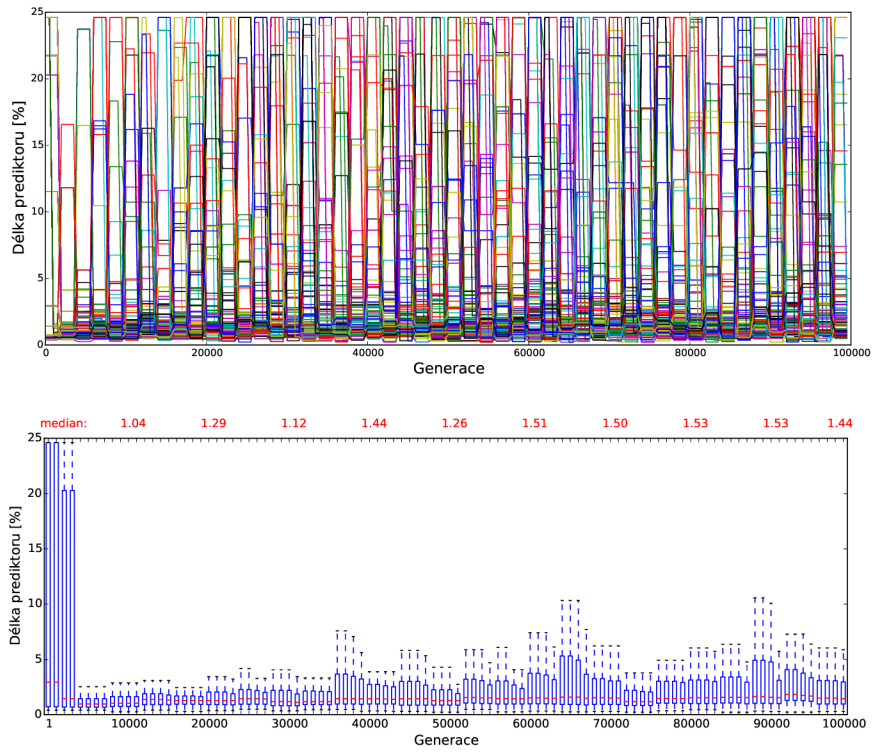
Příloha C

Grafy výsledků experimentů

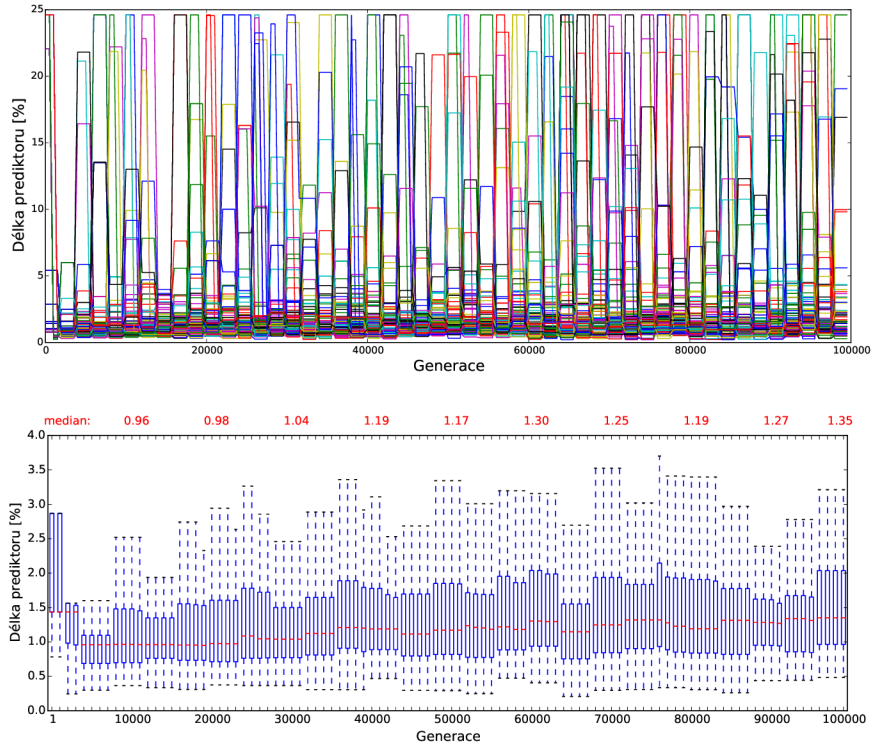
V textu technické zprávy byly vybrány výstupy jen několika provedených experimentů. Následující stránky obsahují soubor všech výsledků, jichž bylo v rámci experimentálního vyhodnocení metody dosaženo.

C.1 Vývoj délky prediktorů

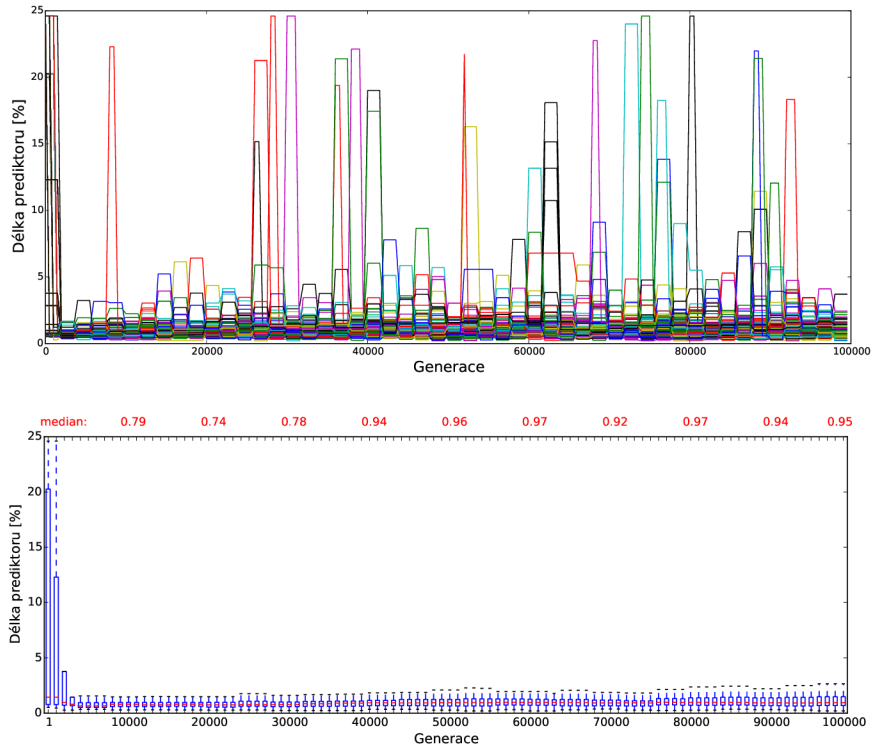
Úloha filtru pro šum typu sůl a pepř



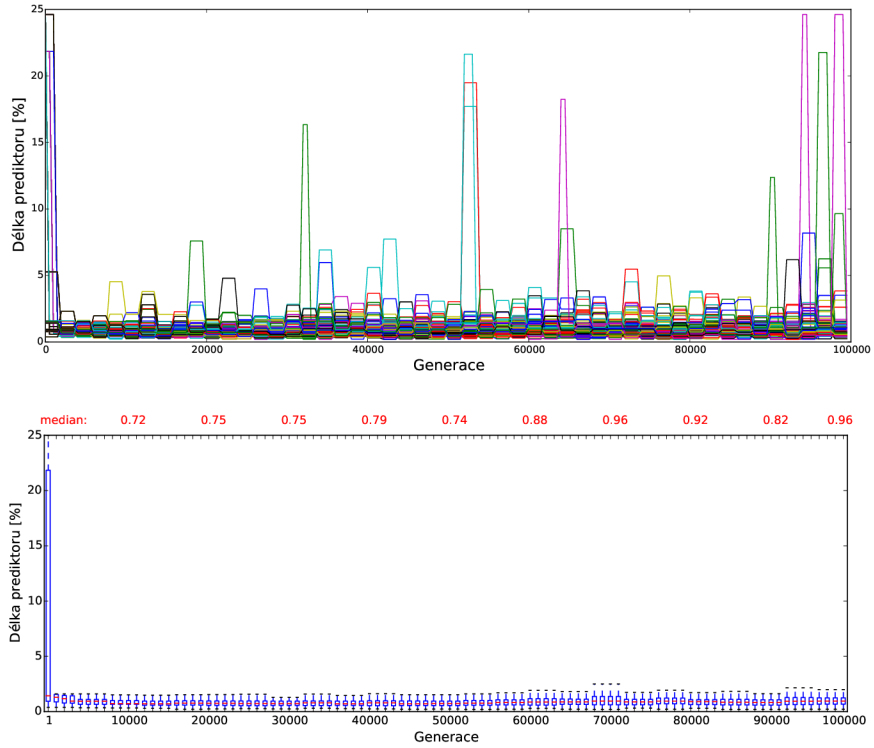
Obrázek C.1: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 10 %.



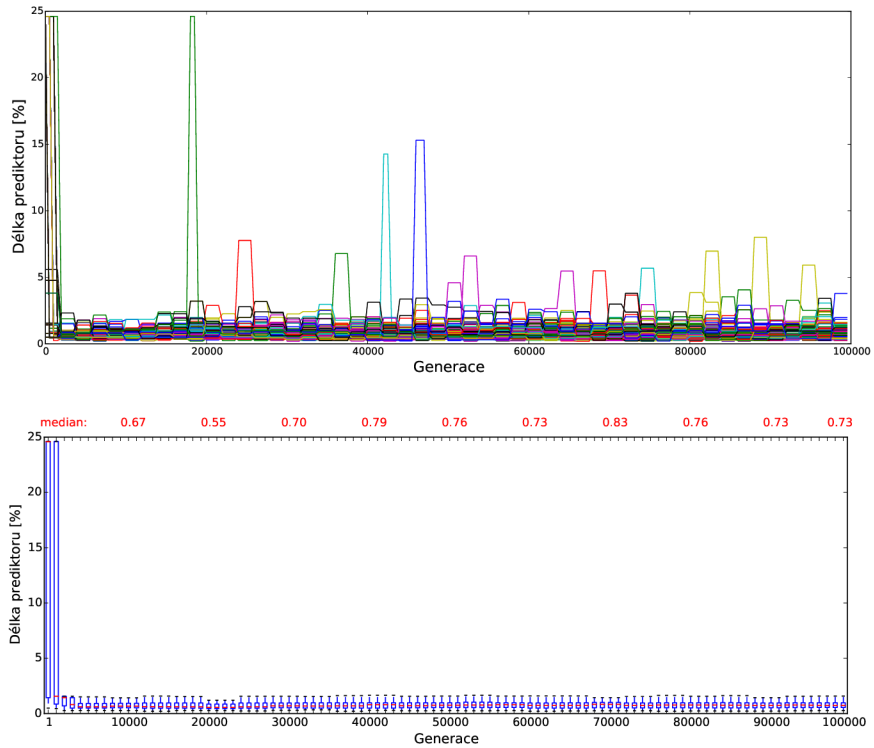
Obrázek C.2: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 20 %.



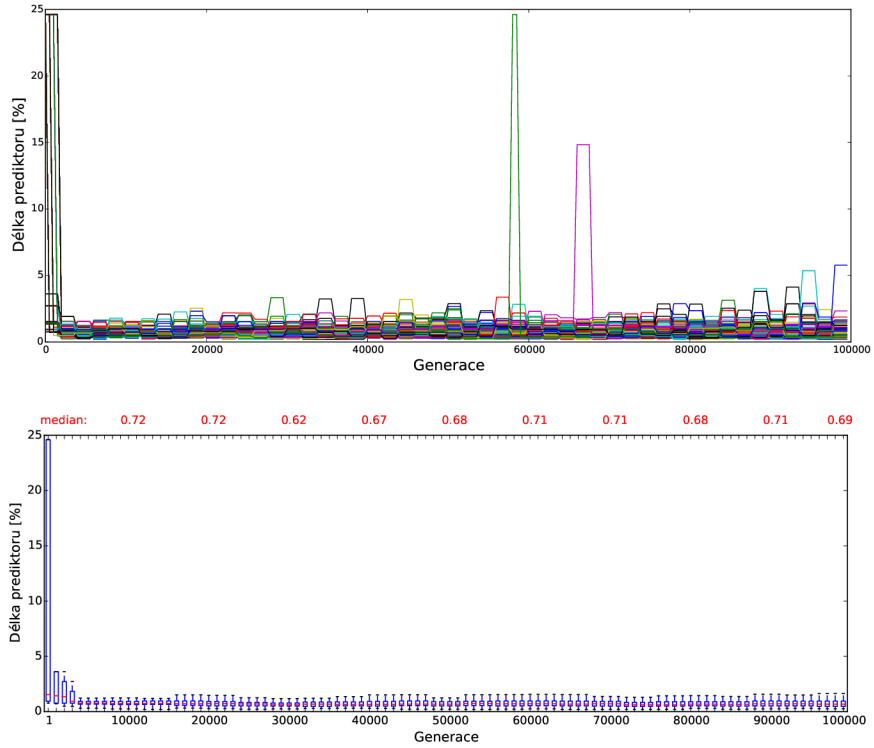
Obrázek C.3: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 30 %.



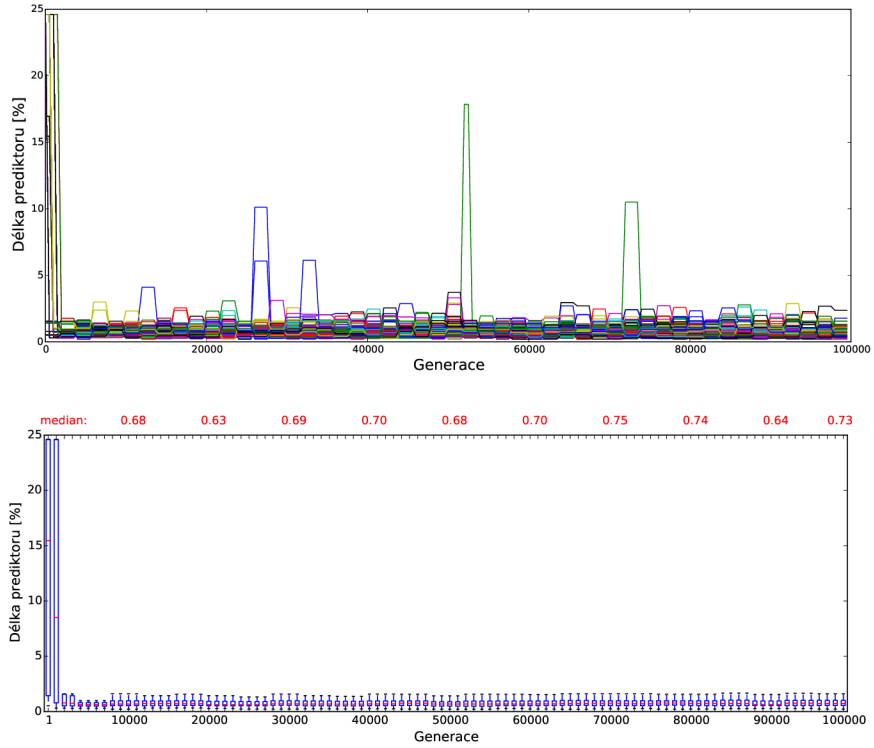
Obrázek C.4: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 40 %.



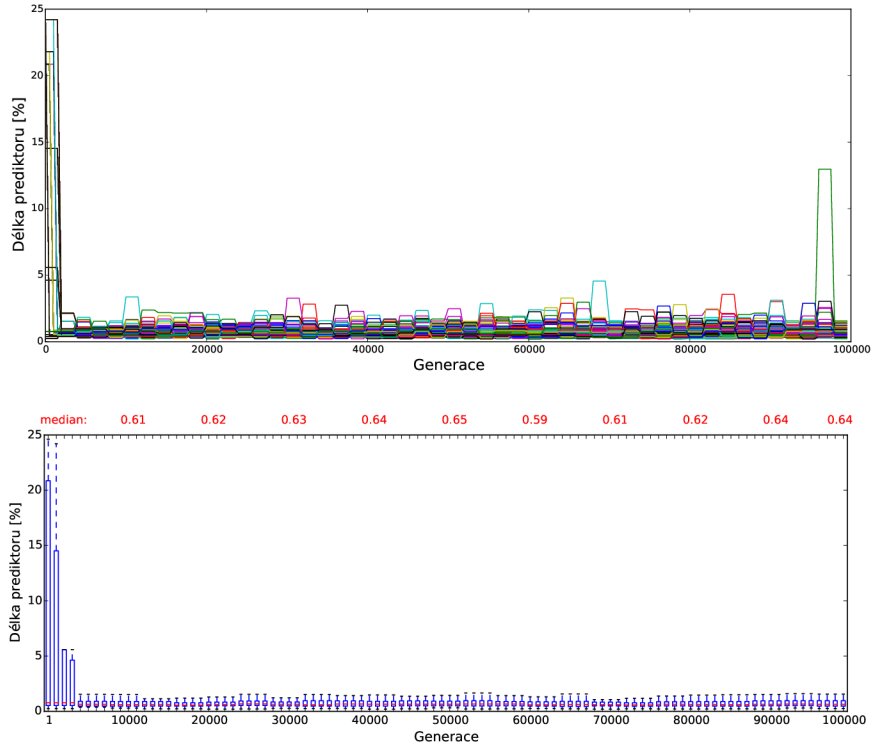
Obrázek C.5: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 50 %.



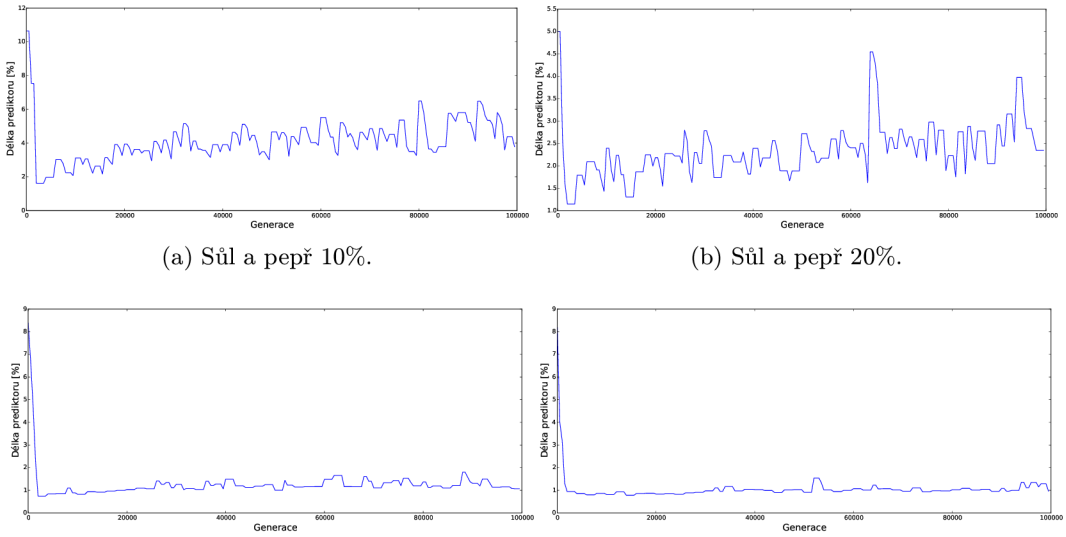
Obrázek C.6: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 60 %.



Obrázek C.7: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 70 %.



Obrázek C.8: Délka prediktorů v jednotlivých bězích - šum sůl a pepř intenzity 80 %.



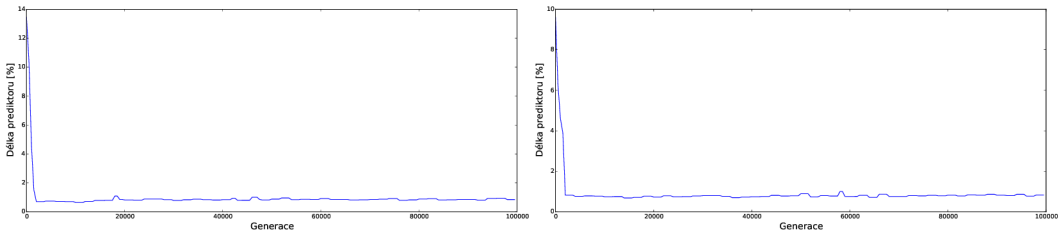
(a) Sůl a pepř 10%.

(b) Sůl a pepř 20%.

(c) Sůl a pepř 30%.

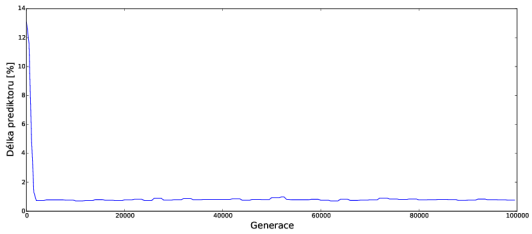
(d) Sůl a pepř 40%.

Obrázek C.9: Průměrná délka prediktorů u 10%, 20%, 30% a 40% šumu typu sůl a pepř.

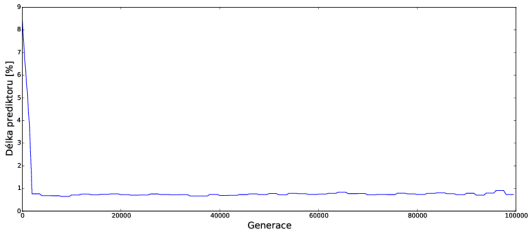


(a) Sůl a pepř 50%.

(b) Sůl a pepř 60%.



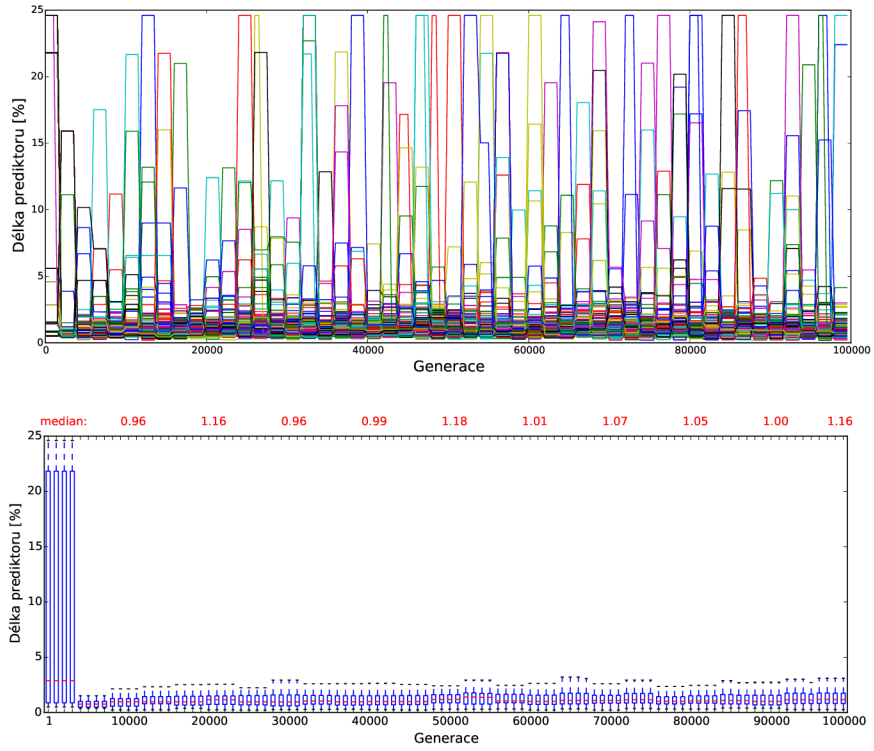
(c) Sůl a pepř 70%.



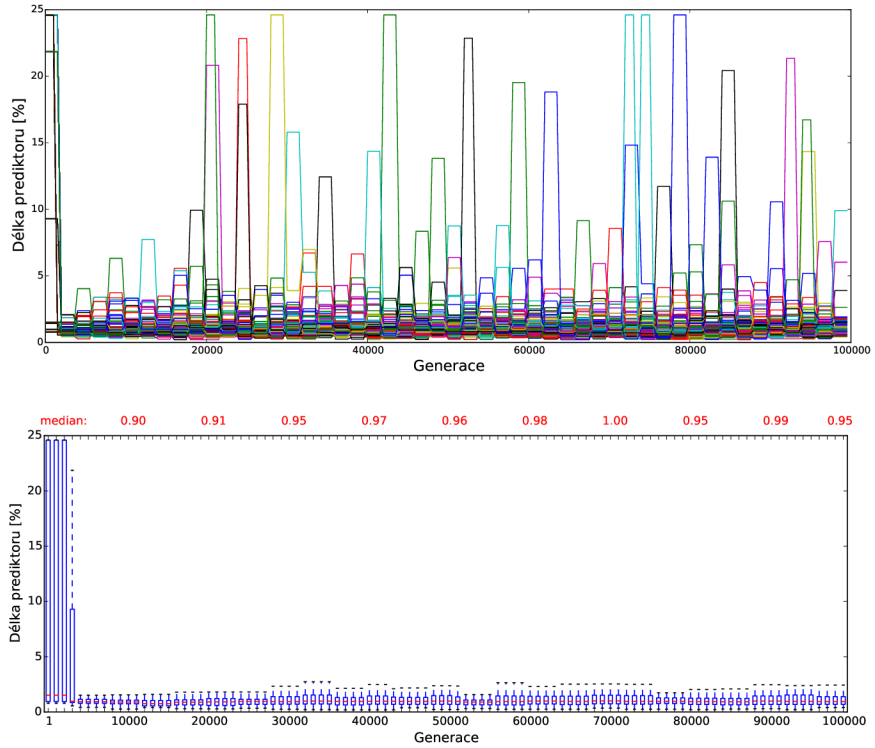
(d) Sůl a pepř 80%.

Obrázek C.10: Průměrná délka prediktorů u 50%, 60%, 70% a 80% šumu typu sůl a pepř.

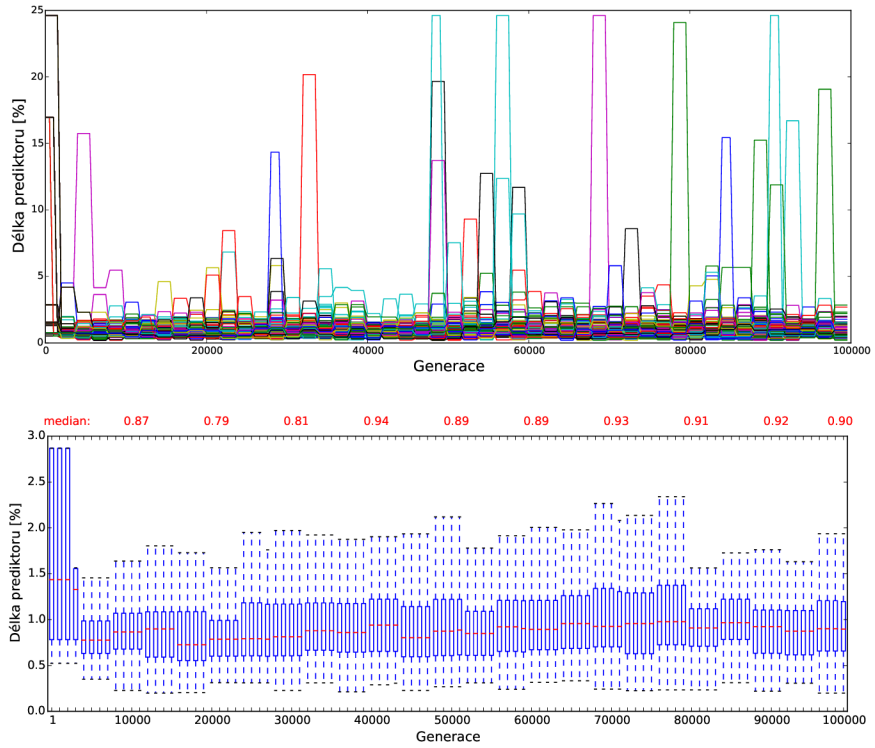
Úloha filtru pro impulzní náhodný šum



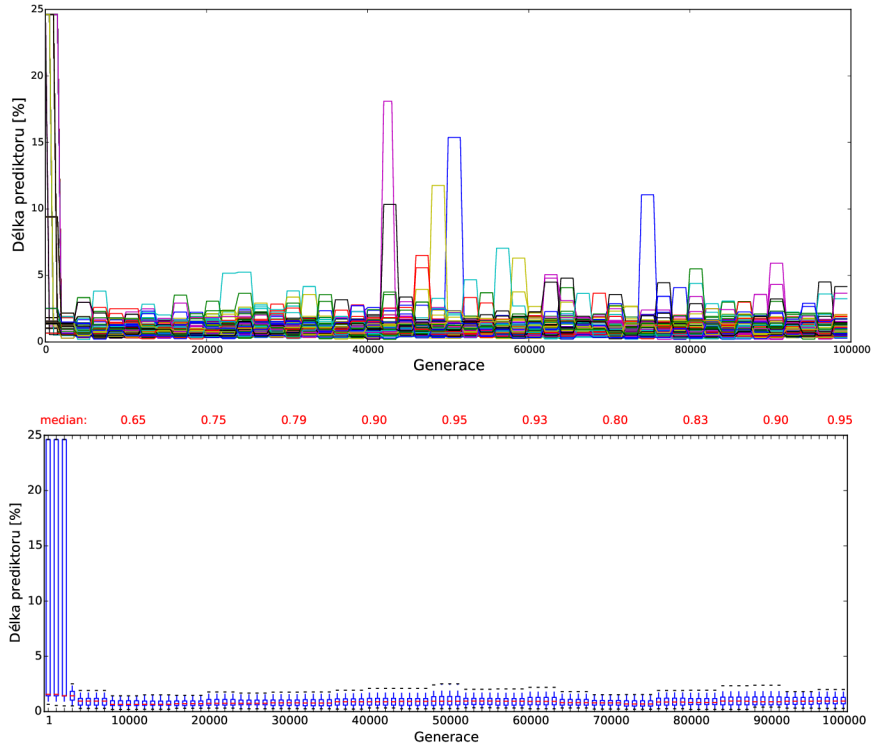
Obrázek C.11: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 10 %.



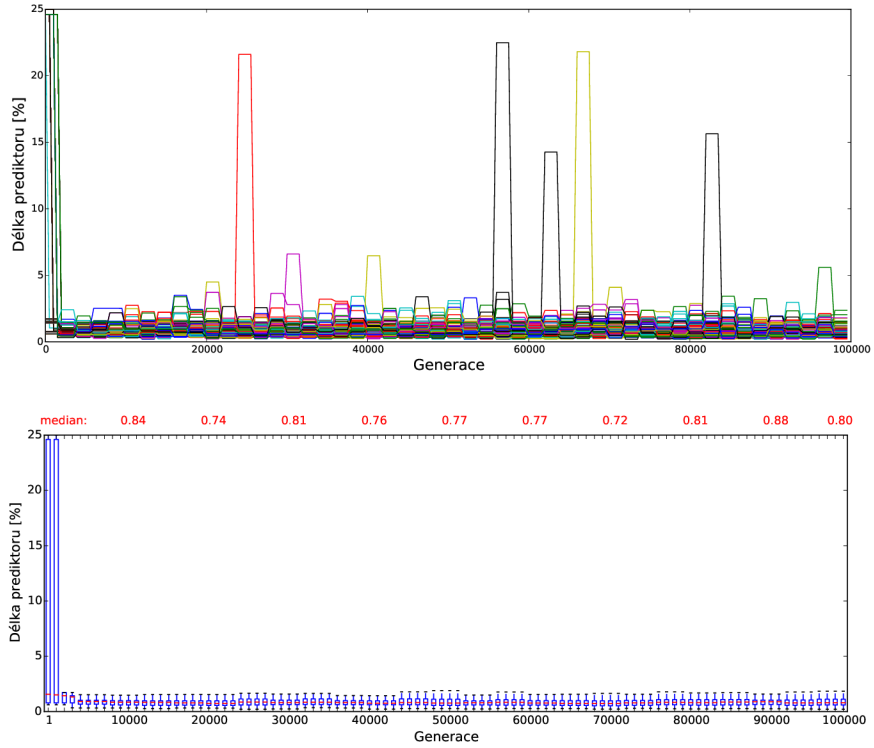
Obrázek C.12: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 20 %.



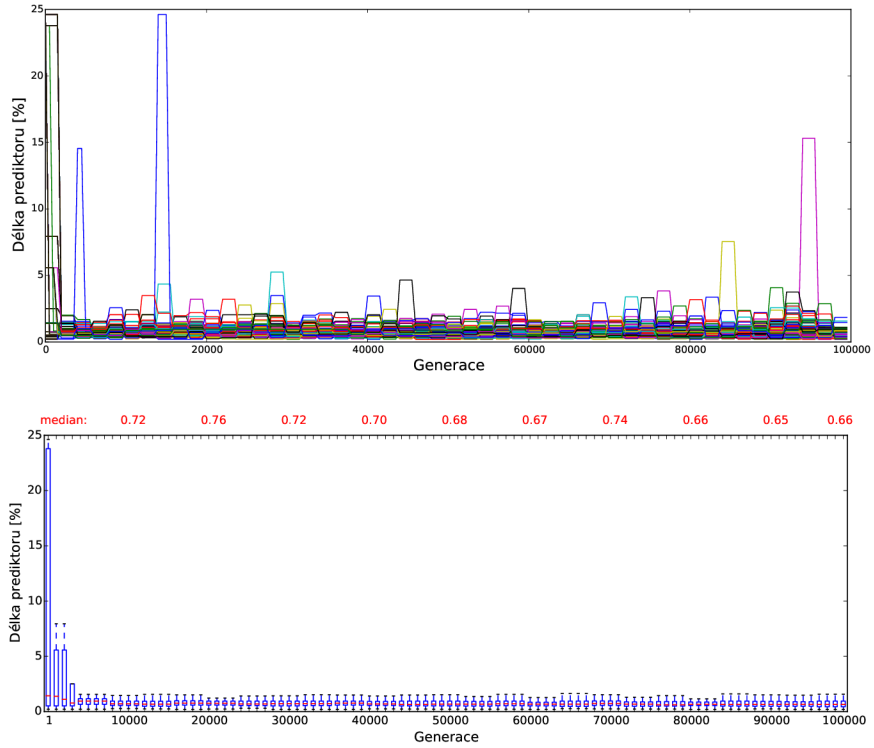
Obrázek C.13: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 30 %.



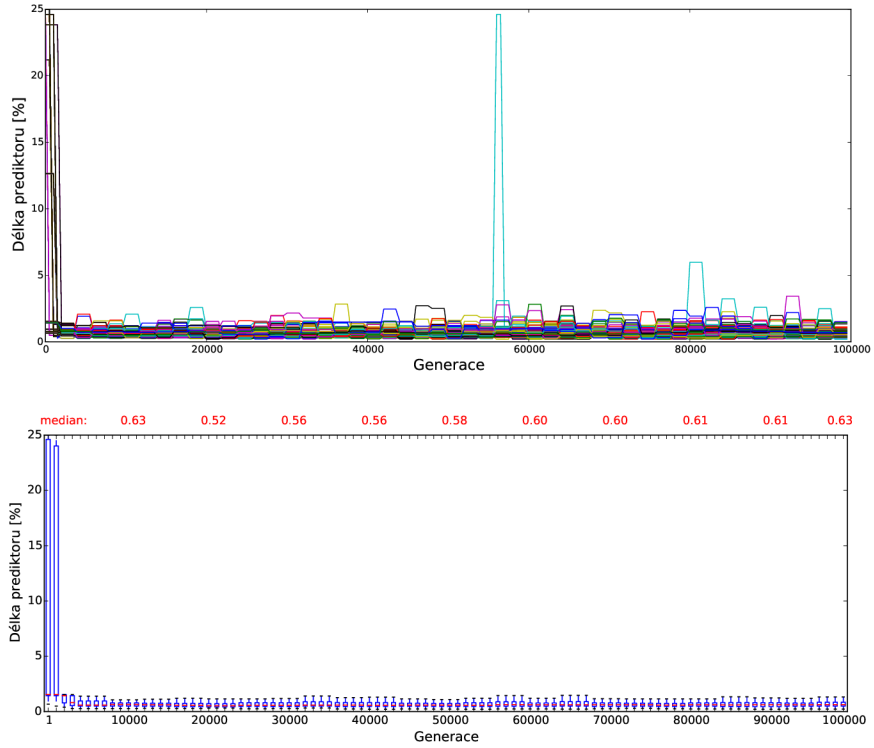
Obrázek C.14: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 40 %.



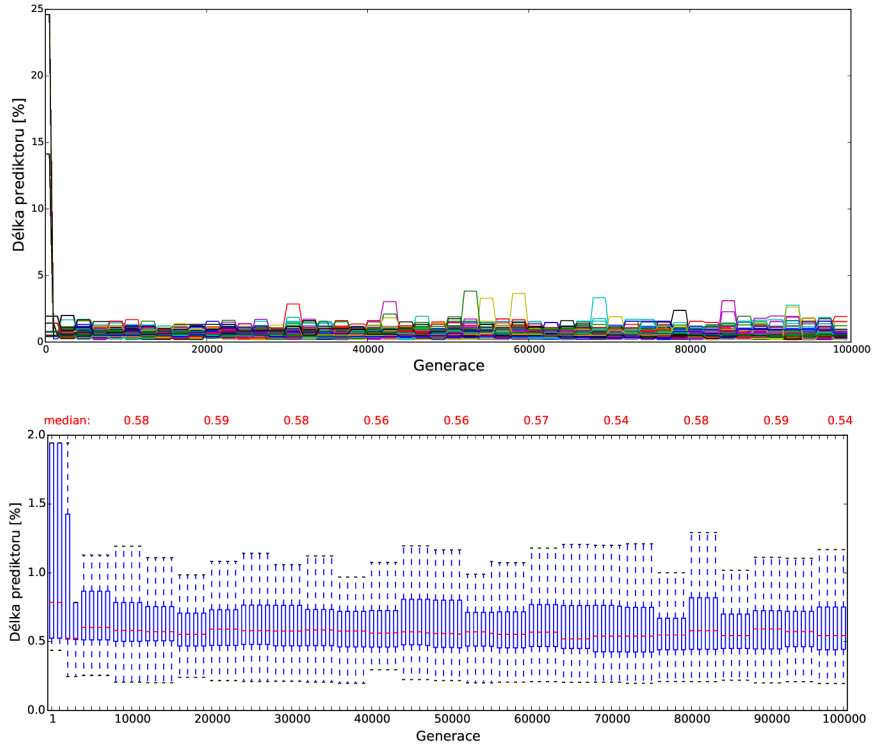
Obrázek C.15: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 50 %.



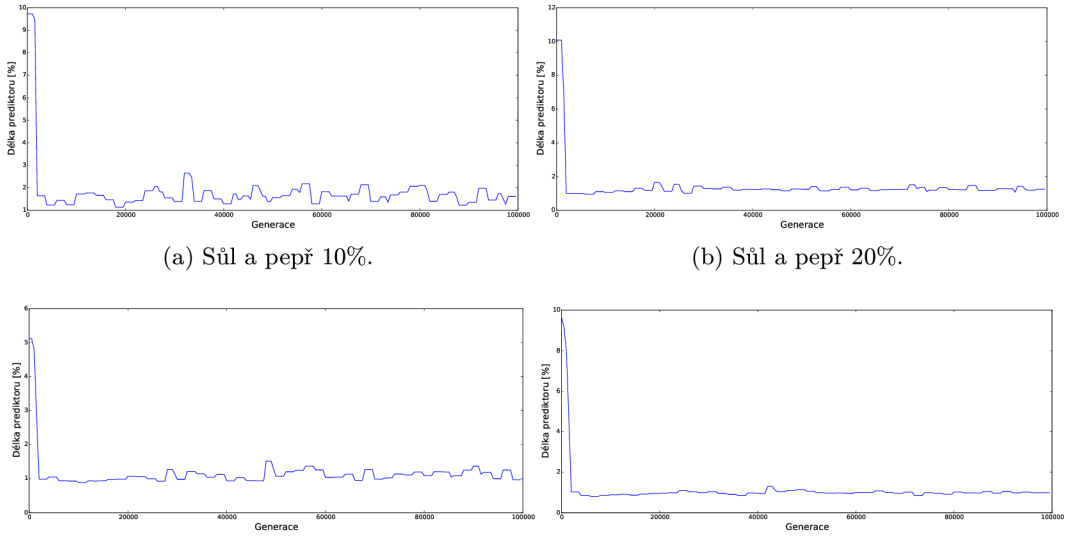
Obrázek C.16: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 60 %.



Obrázek C.17: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 70 %.



Obrázek C.18: Délka prediktorů v jednotlivých bězích - výstřelový šum intenzity 80 %.



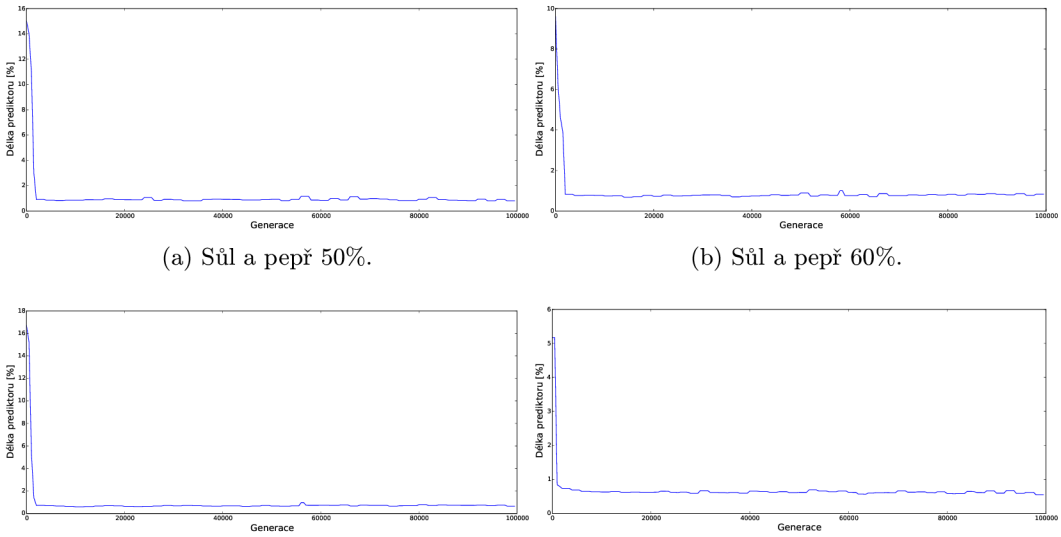
(a) Sůl a pepř 10%.

(b) Sůl a pepř 20%.

(c) Sůl a pepř 30%.

(d) Sůl a pepř 40%.

Obrázek C.19: Průměrná délka prediktorů u 10%, 20%, 30% a 40% impulzní náhodný šum.



(a) Sůl a pepř 50%.

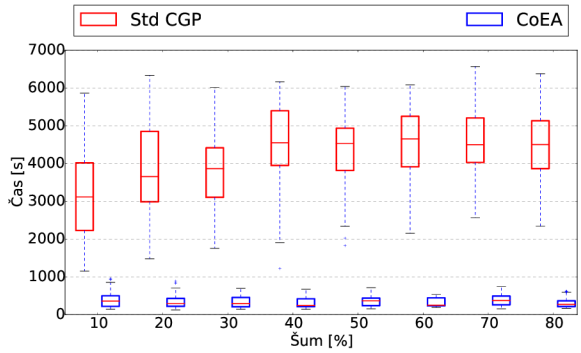
(b) Sůl a pepř 60%.

(c) Sůl a pepř 70%.

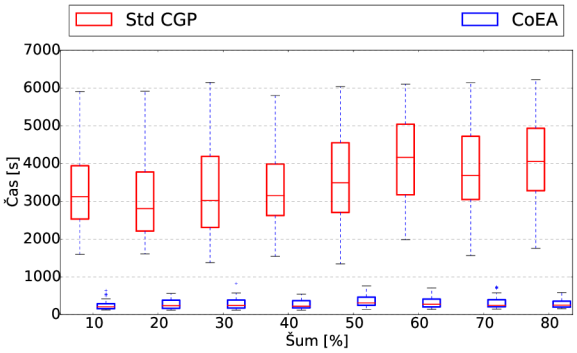
(d) Sůl a pepř 80%.

Obrázek C.20: Průměrná délka prediktorů u 50%, 60%, 70% a 80% impulzní náhodný šum.

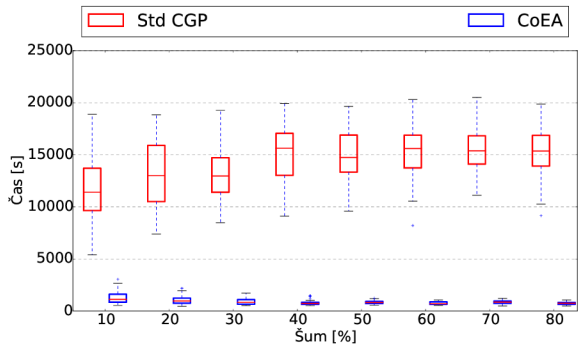
C.2 Porovnání se standardním CGP



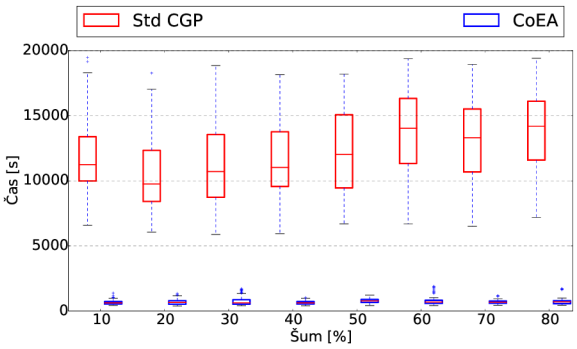
(a) Sůl a pepř 10 000 generací.



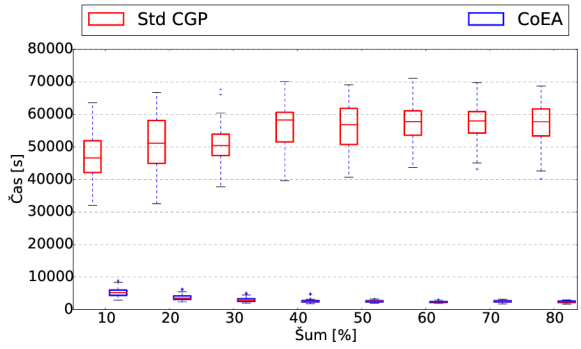
(b) Výstřelový 10 000 generací.



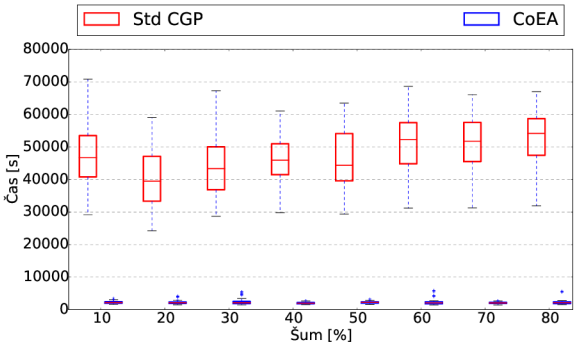
(c) Sůl a pepř 30 000 generací.



(d) Výstřelový 30 000 generací.

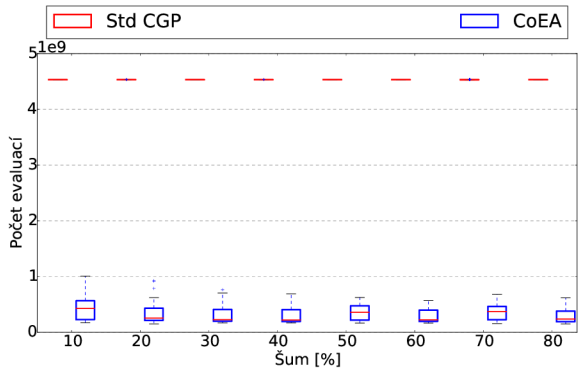


(e) Sůl a pepř 100 000.

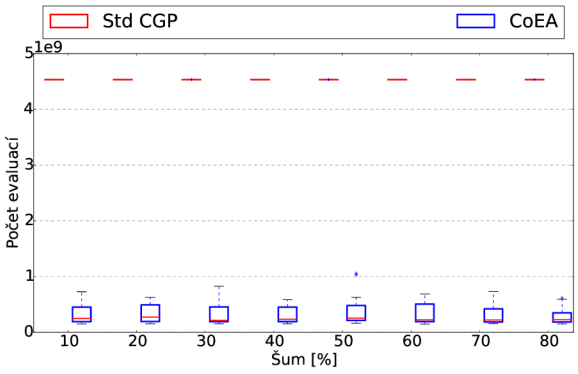


(f) Výstřelový 100 000 generací.

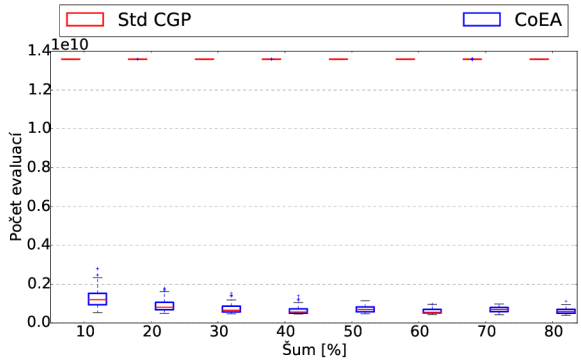
Obrázek C.21: Srovnání doby trvání návrhu řešení.



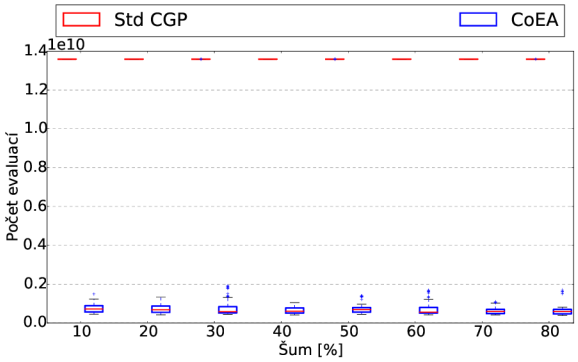
(a) Sůl a pepř 10 000 generací.



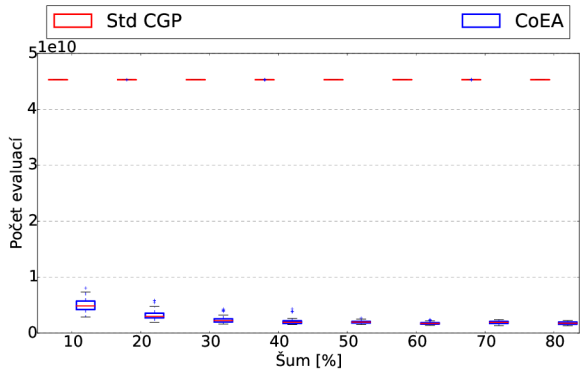
(b) Výstřelový 10 000 generací.



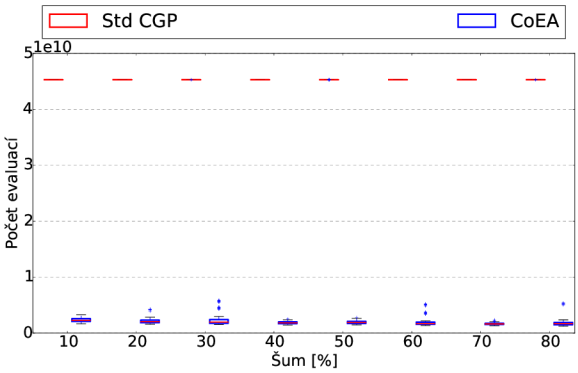
(c) Sůl a pepř 30 000 generací.



(d) Výstřelový 30 000 generací.

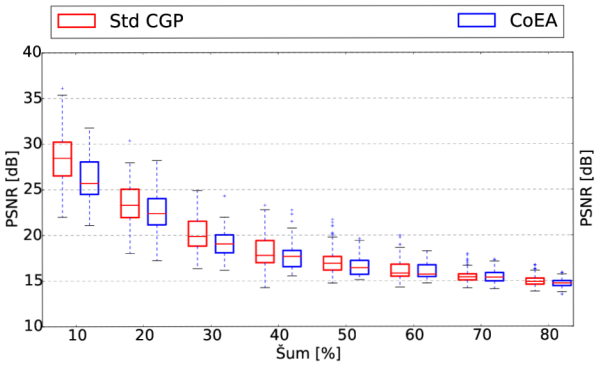


(e) Sůl a pepř 100 000.

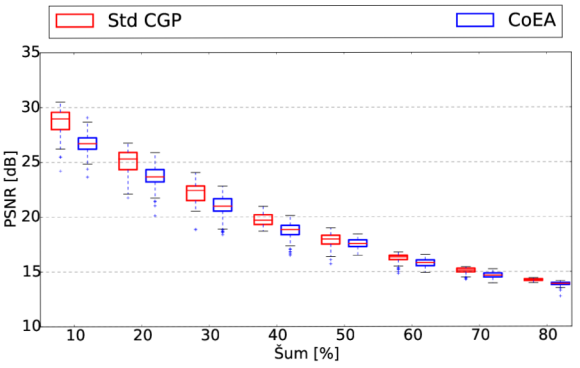


(f) Výstřelový 100 000 generací.

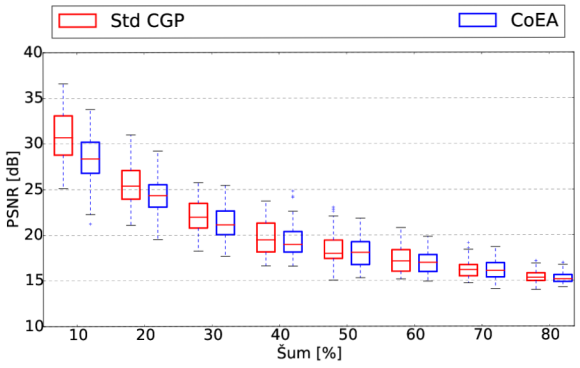
Obrázek C.22: Srovnání počet výpočtů výpočetních bloků.



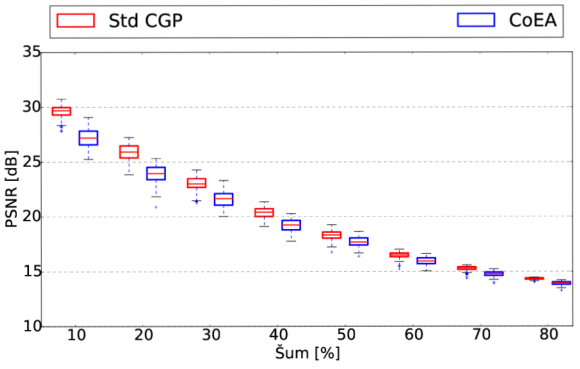
(a) Sůl a pepř 10 000 generací.



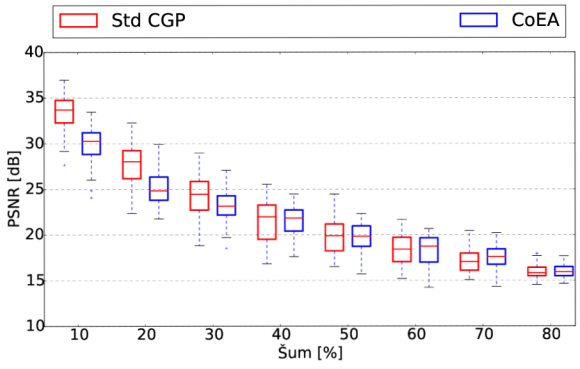
(b) Výstřelový 10 000 generací.



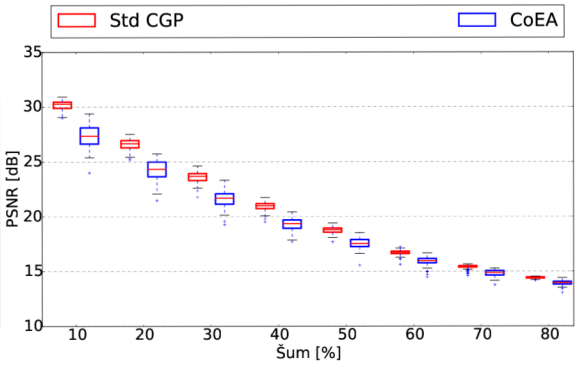
(c) Sůl a pepř 30 000 generací.



(d) Výstřelový 30 000 generací.

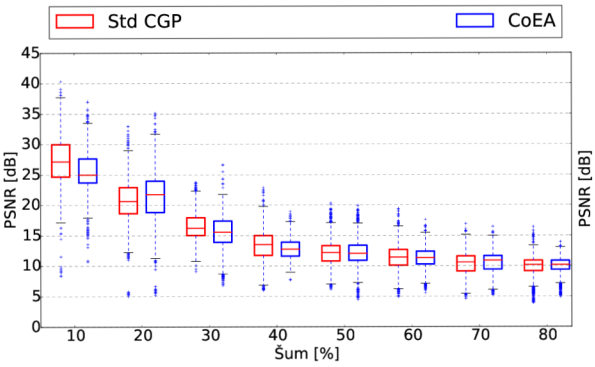


(e) Sůl a pepř 100 000.

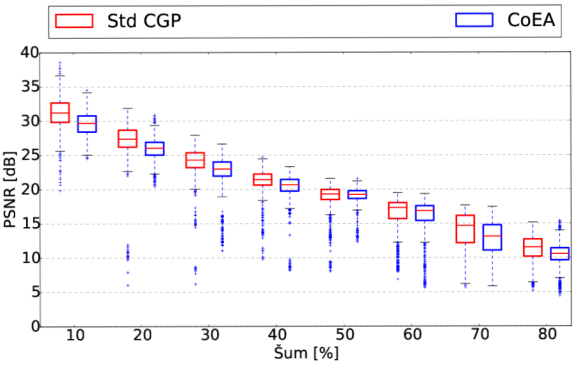


(f) Výstřelový 100 000 generací.

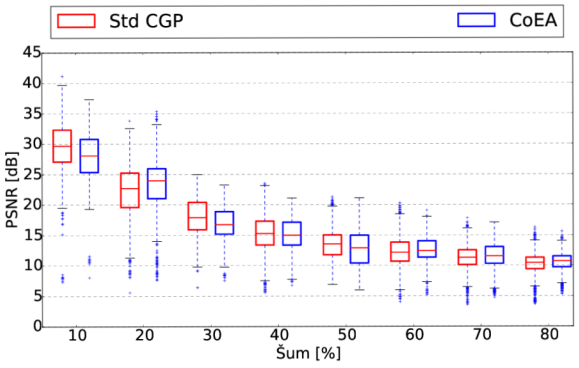
Obrázek C.23: Srovnání fitness výsledného produktu.



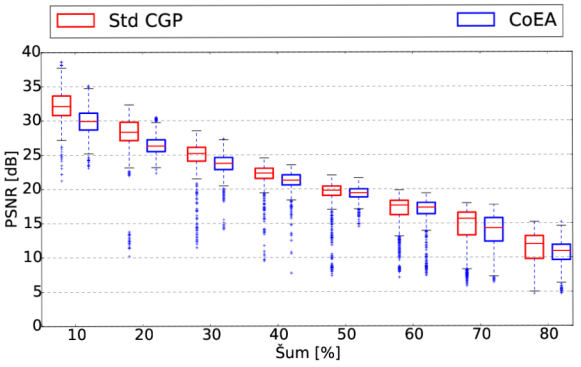
(a) Sůl a pepř 10 000 generací.



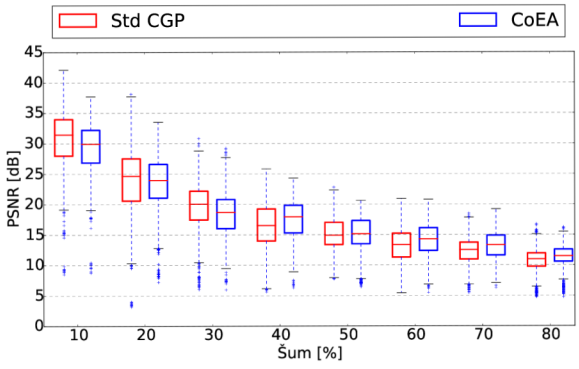
(b) Výstřelový 10 000 generací.



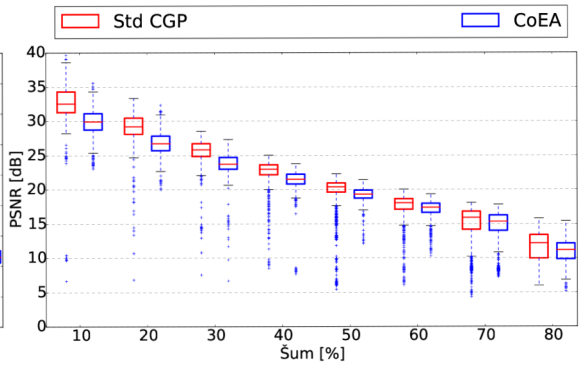
(c) Sůl a pepř 30 000 generací.



(d) Výstřelový 30 000 generací.

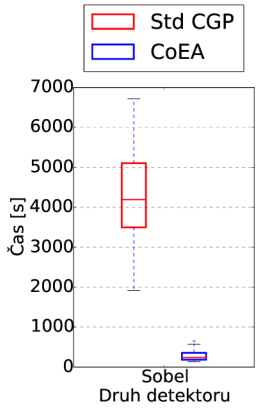


(e) Sůl a pepř 100 000.

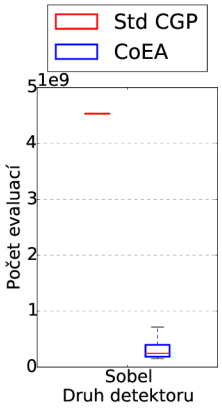


(f) Výstřelový 100 000 generací.

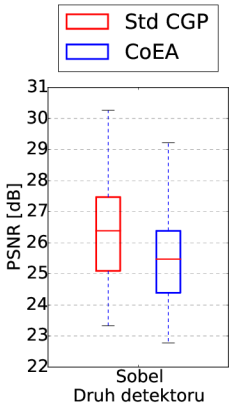
Obrázek C.24: Srovnání kvality výsledného filtru přes sadu testovacích obrázků.



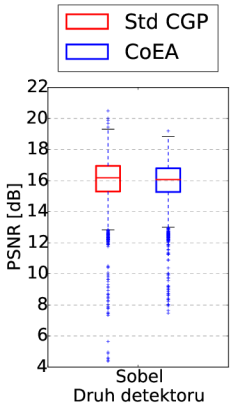
(a) Uživatelský čas po 10 000 generací.



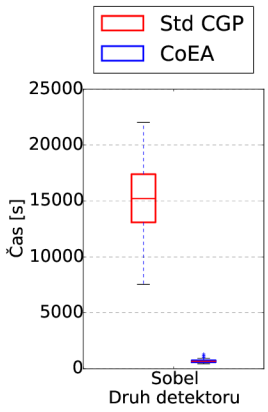
(b) Počet evaluací po 10 000 generací.



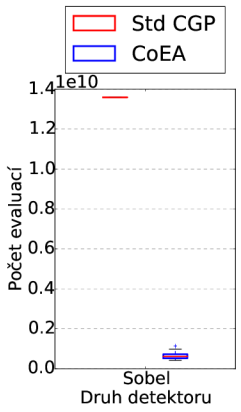
(c) Výsledná fitness po 10 000 generací.



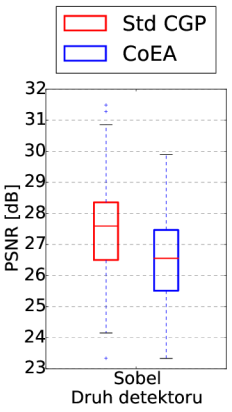
(d) Kvalita testování filtru po 10 000 generací.



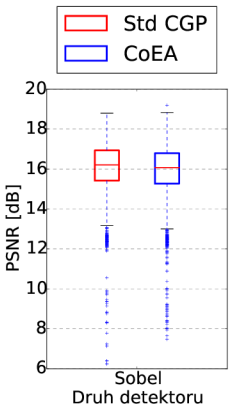
(e) Uživatelský čas po 30 000 generací.



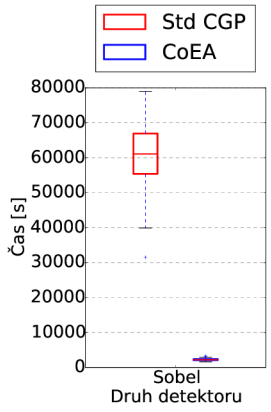
(f) Počet evaluací po 30 000 generací.



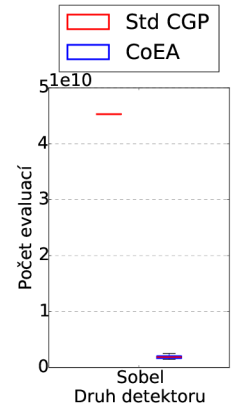
(g) Výsledná fitness po 30 000 generací.



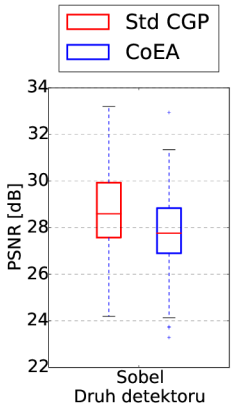
(h) Kvalita testování filtru po 30 000 generací.



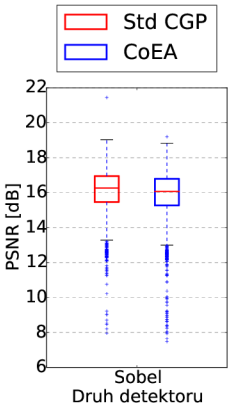
(i) Uživatelský čas po 100 000 generací.



(j) Počet evaluací po 100 000 generací.



(k) Výsledná fitness po 100 000 generací.



(l) Kvalita testování filtru po 100 000 generací.

Obrázek C.25: Srovnání doby trvání, počtu evaluací, fitness konečného řešení a kvality přes testovací obrázky v úloze detektoru hran.