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 DETEKTORŮ ŠUMU

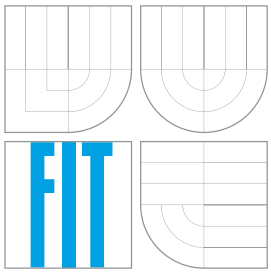
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. GERGELY KOMJÁTHY

BRNO 2014

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 DETEKTORŮ ŠUMU

COEVOLUTION OF IMAGE FILTERS AND NOISE DETECTORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. GERGELY KOMJÁTHY

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAELA ŠIKULOVÁ

BRNO 2014

Abstrakt

Táto práce se zabývá vytvořením obrazových filtrů pomocí koevolučních algoritmů. Práce obsahuje popis evolučních algoritmů, zaměřený hlavně na genetické programování, kartézské genetické programování a koevoluci. Čtenář se dále seznámí s různými typy obrazových filtrů. V dalších částech práce je popsán návrh programu pro tvorbu obrazových filtrů kombinovaných s detektory šumu pomocí kartézského genetického programování a s využitím kooperativní koevoluce, implementace a testování navrženého programu. V poslední části práce jsou filtry vytvořené pomocí koevoluce s detektory šumu porovnány s filtry s detektory šumu vytvořenými bez použití koevoluce a filtry, které nepoužívají detektory.

Abstract

This thesis deals with image filter design using coevolutionary algorithms. It contains a description of evolutionary algorithms, focusing on genetic programming, cartesian genetic programming and coevolution, the reader can learn about image filters too. The next chapters contain the design of image filters and noise detectors using cooperative coevolution, and the implementation and testing of the proposed filter. In the last chapter the proposed filter is compared to other filters created using evolutionary algorithms but without coevolution.

Klíčová slova

Evoluční algoritmus, kartézské genetické programování, koevoluční algoritmus, zpracování obrazu.

Keywords

Evolutionary algorithm, cartesian genetic programming, coevolutionary algorithm, image processing.

Citace

Gergely Komjáthy: Koevoluce obrazových filtrů a detektorů šumu, diplomová práce, Brno, FIT VUT v Brně, 2014

Koevoluce obrazových filtrů a detektorů šumu

Prohlášení

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

.....
Gergely Komjáthy
28. května 2014

Poděkování

Rád bych poděkoval vedoucí své diplomové práce Ing. Michaele Šikulové za připomínky, rady a odbornou pomoc, kterou mi během práce poskytovala.

Obsah

1 Úvod **3**

2 Evolučné algoritmy **4**
2.1 Základné pojmy v evolučných algoritmoch 4
2.2 Typy evolučných algoritmov 4
2.3 Komponenty evolučných algoritmov 5
2.3.1 Algoritmus evolučných algoritmov 7
2.4 Genetické programovanie 8
2.4.1 Algoritmus GP 9
2.4.2 Fitness 10
2.4.3 Problémy GP 11
2.5 Kartézske genetické programovanie (CGP) 11
2.5.1 Reprezentácia 11
2.5.2 Zakódovanie CGP 12
2.5.3 Algoritmus 12
2.6 Koevolučné algoritmy 12
2.6.1 Základné pojmy koevolučných algoritmov 13
2.6.2 Ohodnotenie jedincov 13
2.6.3 Rozdelenie koevolučných algoritmov 14

3 Obrazové filtre **16**
3.1 Konvenčné obrazové filtre 16
3.2 Evolučné obrazové filtre 17
3.3 Obrazové filtre a koevolúcia 18

4 Návrh obrazových filtrov s detektorom šumu **19**
4.1 Evolučný algoritmus 20
4.2 Koevolúcia 21
4.2.1 Fitness 22
4.3 Paralelizácia 23

5 Implementácia **26**
5.1 Použité štruktúry a datové typy 26
5.2 Triedy ***filter*** a ***detector*** 26
5.2.1 Generovanie (pseudo) náhodných čísel 27
5.3 Práca so súbormi 27
5.4 Funkcia main() 27
5.5 Ďalšie moduly programu 28

5.6 Vstupy a výstupy programu 28

6 Testovanie parametrov algoritmu **29**
6.1 Testovacie skripty 29
6.2 Možné parametre pre nastavenie 29
6.2.1 Experimentálne nastavenie 31
6.3 Fitness hodnota 31

7 Experimentálne vyhodnotenie **33**
7.1 Časová náročnosť 34
7.2 Veľkosť riešenia 36
7.3 Kvalita riešenia 37
7.4 Diskusia o výsledkoch 40
7.4.1 Príklad evolvovaného filtra 41

8 Záver **45**

A Filtrom opravené obrázky **48**

B Preklad a spustenie programu **60**

C Obsah CD **62**

Kapitola 1

Úvod

V dnešnej dobe vzniká stále viac fotografií, ľudia kladú stále väčší dôraz na to, aby obrázky boli čo najkvalitnejšie. S rastúcimi požiadavkami na kvalitu, sú vytvárané stále lepšie zariadenia na vytvorenie a skladovanie obrázkov. Napriek tomu, obrázky sú často poškodené nejakým šumom, čo komplikuje ich spracovanie, prípadne zhoršuje ich estetickú kvalitu.

Pre odstránenie takých chýb vznikli rôzne obrazové filtre. Obrazové filtre skúsia opraviť alebo aspoň znížiť mieru chyby v obraze, väčšinou pomocou susedných pixelov poškodeného pixela. Nevýhodou niektorých filtrov je, že zmenia hodnoty i nepoškodených pixelov. Možným riešením tohto problému je aplikovať filter len na poškodené pixely. Aby bolo možné zistiť, ktoré pixely sú poškodené, potrebujeme detektor šumu. Detektor šumu nájde poškodené pixely a na tieto pixely je aplikovaný daný filter.

Obrazové filtre a detektory šumu je možné vytvoriť rôznymi spôsobmi. Najčastejšie sa používajú filtre navrhnuté konvenčným spôsobom. Tieto filtre boli matematicky odvodené, prípadne vymyslené, ale nedávajú dobré výsledky napr. v prípade pri vysokej intenzite šumu. Aj v prípade veľmi poškodených obrázkov, potrebujeme spoľahlivé filtre, ktoré upravia obrázok aspoň do prijateľnej miery. Na úpravu takýchto obrázkov sa používajú hlavne nekonvenčné filtre, ktoré boli vytvorené napr. pomocou evolučných algoritmov.

Pomocou evolučných algoritmov je možné vyvíjať filtre „na mieru", definovaním pravidiel na daný problém. Žiaľ tieto filtre zvyčajne pracujú dobre len na tie typy problémov, na ktoré boli vytvorené.

Cieľom tejto diplomovej práce je preštudovať genetické programovanie, koevolúciu, evolučný návrh obrazových filtrov, navrhnúť program pre tvorbu obrazových filtrov kombinovaných s detektormi šumu pomocou genetického programovania a s využitím kooperatívnej koevolúcie, navrhnutý program implementovať a overiť jeho funkčnosť.

Technická správa je rozdelená do ôsmich kapitol. V kapitole 2 sa môžeme zoznámiť s evolučnými algoritmami, dôkladnejšie je rozobrané genetické programovanie (GP), kartézske genetické programovanie (CGP) a koevolučné algoritmy. Obrázkové filtre sú popísané v kapitole 3. Kapitola 4 sa zaoberá návrhom programu pre tvorbu obrazových filtrov s detektorom šumu pomocou koevolúcie. Kapitola 5 popisuje implementáciu navrhnutého programu. V kapitole 6 sú popísané jednotlivé parametre programu. Výsledky testov a porovnávanie implementovaných metód sú popísané v kapitole 7. V záverečnej kapitole (kapitola 8) sú zhrnuté výsledky práce.

Kapitola 2

Evolučné algoritmy

Evolučné algoritmy (EA) sú stochastické prehľadávacie algoritmy inšpirované Darwinovou evolučnou teóriou. Evolúciu v biológii môžeme definovať ako pozvoľnú modifikáciu druhu v čase. Varianty v rámci druhu sú overené a vylepšené prírodnou selekciou. Pozostalí jedinci prenášajú svoje znaky na ďalšie generácie. Druh sa vyvíja opakovaním tohto procesu v priebehu dlhého časového obdobia [17].

Termín evolučný algoritmus vznikol začiatkom 90-tych rokov, aj keď prvé techniky EA sa používali aj v 50-tych rokoch 20. storočia. Tieto algoritmy sa v počiatku používali výhradne pre optimalizáciu, kde cieľom bolo doladiť parametre nejakého hotového riešenia. V poslednej dobe sa často používajú v úlohách návrhu [16]. Cieľom návrhu je vytvorenie niečo nového podľa definovaných pravidiel.

2.1 Základné pojmy v evolučných algoritmoch

Výrazy použité v evolučných algoritmoch väčšinou pochádzajú z genetiky, ale ich význam je často iný. V tejto časti sú definované niektoré použité pojmy evolučných algoritmov.

- **Gén** je základným stavebným blokom chromozómu. Jeho možné hodnoty sú dopredu dané abecedou (binárne čísla, znaky, atď.).
- **Chromozóm** (jedinec) je pole génov. Chromozóm reprezentuje jedno riešenie pre danú úlohu.
- **Populácia** je konečná množina kandidátnych riešení.
- **Genotyp** je kódovaný tvar riešenia.
- **Fenotyp** je dekódovaný tvar riešenia (napríklad konkrétny odvod, obrazový filter, atď.).
- **Fitness funkcia** je funkcia, ktorá určí kvalitu jedinca.
- **Diverzita** udáva mieru rozdielov medzi jedincami populácie.

2.2 Typy evolučných algoritmov

Evolučné algoritmy delíme do štyroch základných skupín [3]:

Evolučné stratégie

Evolučné stratégie väčšinou optimalizujú vektor reálnych parametrov. Parametre vektoru sú modifikované pomocou mutácie – pričítaním (odčítaním) vygenerovaného náhodného čísla. Generátor používa Gaussovo rozloženie s nulovou strednou hodnotou a rozptylom δ. Parameter δ sa modifikuje podľa toho, či sú potomkovia úspešnejší, ako rodičia, alebo nie.

Evolučné programovanie

Evolučné programovanie je podobné evolučným algoritmom, aj keď tieto postupy boli vymyslené nezávisle na sebe. Používa sa hlavne mutácia. Z každého rodiča je vytvorený jeden potomok.

Genetické algoritmy

Genetické algoritmy sú pravdepodobne najpopulárnejšou skupinou evolučných algoritmov. Parametre riešenia sú zakódované do génov, ktoré sú väčšinou binárne. Hlavným operátorom je kríženie, mutácia je použitá pre udržaniu diverzity. Tieto algoritmy sa používajú hlavne pri optimalizačných problémoch.

Genetické programovanie

Genetické programovanie je zamerané na návrh algoritmov alebo počítačových programov. Podrobnejšie s genetickým programovaním sa zaoberá kapitola 2.4.

Spoločné rysy evolučných algoritmov

Aj keď jednotlivé typy evolučných algoritmov pracujú inak, majú niekoľko spoločných rysov [3]:

- Evolučné algoritmy väčšinou pracujú s viacerými jedincami (populáciou), nie len s jedným kandidátnym riešením.
- Výber rodičov závisí na ohodnotení jedinca: čím kvalitnejší je jedinec, tým viackrát je vybraný.
- Nové jedince sú vytvorené z existujúcich jedincov. Potomkovia sú tvorení z rodičov pomocou stochastických operátorov (napr. krížením a mutáciou).

2.3 Komponenty evolučných algoritmov

Evolučné algoritmy sa zostávajú z niekoľko komponentov, ktoré budú dôkladnejšie popísané v týchto podkapitolách [2]:

Reprezentácia

Prvým problémom, s ktorým sa stretneme, keď chceme používať nejaký evolučný algoritmus na riešenie daného problému je, že ako bude riešenie kódované do chromozómu. Čo poznáme v čase návrhu je fenotyp – ako bude riešenie vypadať v „reálnom svete". Tento fenotyp je nutné zakódovať do genotypu tak, aby s genotypom bolo možné pracovať v rámci výpočtu.

Napríklad cestu v grafe je možné zakódovať, ako postupnosť poradových čísiel uzlov. Na konci výpočtu je treba genotyp dekódovať a získať riešenie.

Fenotyp väčšinou môže byť zakódovaný na viac rôznych genotypov, ale dekódovanie genotypu je jednoznačné – každý genotyp je dekódovaný len na jeden fenotyp.

Ohodnotenie jedincov

Po vytvorení nového jedinca je nutné jeho ohodnotenie. Na ohodnotenie sa používa tzv. fitness funkcia. Výstupom fitness funkcie je číselná hodnota, vyjadrujúca kvalitu jedinca. Toto ohodnotenie je veľmi dôležité, pretože ovplyvňuje smer vývoja.

Ohodnotenie jedincov je väčšinou časovo najnáročnejšia časť výpočtu. Preto je treba optimalizovať a urýchliť výpočet fitness hodnoty. Na zrýchlenie výpočtu sú rôzne prístupy, ako napr. paralelizácia, použitie rekonfigurovateľných obvodov (FPGA) [22], použitie grafických procesorov (GPU) [5], použitie koevolúcie [19, 18], atď.

Populácia

Cieľom populácie je združovanie jedincov. Čo sa vyvíja časom, nie sú jedinci, je to populácia. Jedinci sú nezmenení, sú len časom vymenení za iných (kvalitnejších) jedincov.

Vo väčšine evolučných algoritmov populácia je daná len počtom jedincov. Existujú ale aj evolučné algoritmy, kde populácia má aj iné funkcie. Populácia môže mať priestorovú štruktúru, kde je nutné aj definovať celú štruktúru populácie [20].

Výber rodičov

Rodičia sú vybraní z ohodnotených jedincov. Pri výbere sú uprednostnení jedinci s vyššou fitness hodnotou, ale aj menej kvalitní jedinci môžu byť vybraní.
Používajú sa rôzne typy selekcie, najčastejšie sú nasledujúce:

- Ruletový výber – Každý jedinec má priradenú časť na rulete. Veľkosť tejto časti je priamo úmerná jeho hodnoteniu. Jedince s vyššou fitness hodnotou tak majú väčšiu šancu pri výbere, ale môžu byť vybrané aj jedince s nižšou fitness hodnotou. Na obrázku 2.1b a v tabuľke 2.1a je príklad rulety s pravdepodobnosťami výberu jednotlivých jedincov. Existujú rôzne vylepšenia rulety, ako napr. výber oboch rodičov v jednom kole.

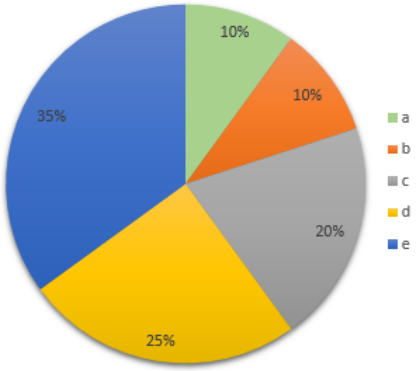
- Turnajový výber – Náhodne je vybraný daný počet jedincov. Najlepší z nich (jedinec s najvyššou fitness hodnotou) sa stáva rodičom.

- Veľmi často sa používa kombinácia ruletového a turnajového výberu – je vybraných viac jedincov pomocou rulety, najlepší z nich sa stáva rodičom.

- Výber podľa poradí – jedinci sú zoradení podľa ich fitness hodnoty. Výber rodičov potom závisí len na poradí, a nie na ich fitness hodnote. Táto metóda je vhodná, keď jedinci majú veľké rozdiely v fitness hodnotách.

Jedinec	Fitness	Pravdepodobnosť výberu
a	2	10%
b	2	10%
c	4	20%
d	5	25%
e	7	35%

(a)



(b)

Obrázok 2.1: Pravdepodobnosť výberu jedincov pomocou rulety

Variačné operátory

Cieľom variačných operátorov je vytvoriť nových jedincov na základe svojich rodičov. Potomkovia zdedia niektoré vlastnosti rodičov, preto je dôležité, aby rodičom sa stali čo najlepší jedinci. Na rekombináciu sa najčastejšie používa kríženie a mutácia.

Pre kríženie potrebujeme obvykle dvoch rodičov pričom vznikajú dvaja potomkovia. Chromozóm potomka bude tvorený z prehodených častí chromozómov rodičov. Najčastejšie sa používa jednobodové kríženie, ale môže sa použiť aj viacbodové kríženie.

Mutácia potrebuje len jedného rodiča. Úlohou mutácie je udržovať génovú variabilitu (diverzitu) v populácii. Pracuje tak, že zmení hodnotu náhodného génu (náhodných génov) na inú (legálnu) hodnotu. Väčšinou sa používa s malou pravdepodobnosťou (5%), ale sú prípady, keď jediným operátorom prehľadávania je mutácia.

Obnova populácie

Aby veľkosť populácie nerástla v každej generácii po vytvorení potomkov a po výpočte ich fitness hodnôt, nasleduje obnova populácie zjednotením rodičovskej populácie a vytvorených potomkov. Na obnovu populácie máme dva prístupy:

- **Generačný** – potomkovia nahradia celú rodičovskú populáciu,
- **Steady-state** – najlepší potomkovia nahradia časť starej populácie. Sú nahradení najhorší alebo najstarší jedinci. Často sa používa tzv. elitizmus, keď daný počet najlepšie ohodnotených jedincov postupuje do ďalšej generácie.

2.3.1 Algoritmus evolučných algoritmov

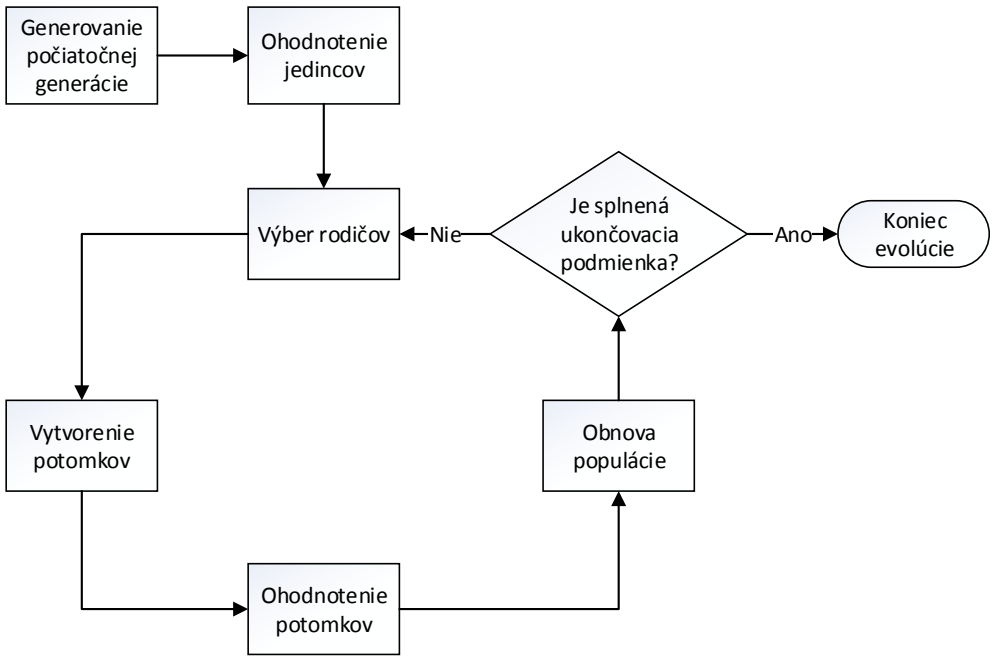
Pred samotným výpočtom je treba nastaviť jednotlivé parametre výpočtu, ako veľkosť populácie, počet generácií, mieru mutácie, atď. Tieto nastavenia ovplyvňujú kvalitu riešenia, rýchlosť výpočtu, väčšinou sú špecifické pre daný problém, preto je nutné zvoliť tieto parametre podľa špecifikácie problému.

Evolučné algoritmy sa začínajú vytvorením počiatočnej populácie. Počiatočná populácia sa vytvára náhodne, je riadená vhodnou heuristikou, ktorá vnáša znalosť o probléme do riešenia, alebo sa používa nejaké predchádzajúce riešenie. Vytvorené jedince sú potom

ohodnotené pomocou fitness funkcie. Z ohodnotených jedincov sú vybraní rodičia, z ktorých sú vytvorení pomocou rekombinácie potomkovia. Potomkovia sú potom ohodnotení a nasleduje obnova populácie.
Algoritmus sa ukončí, keď je splnená jedna z ukončovacích podmienok:

- našli sme dostatočne dobrého jedinca (výsledok),
- dosiahli sme maximálny počet generácií,
- priemerná fitness hodnota sa nezlepšila už dlhšiu dobu.

Inak pokračujeme s výberom nových rodičov. Celý algoritmus je zobrazený na obrázku 2.2.



Obrázok 2.2: Algoritmus EA

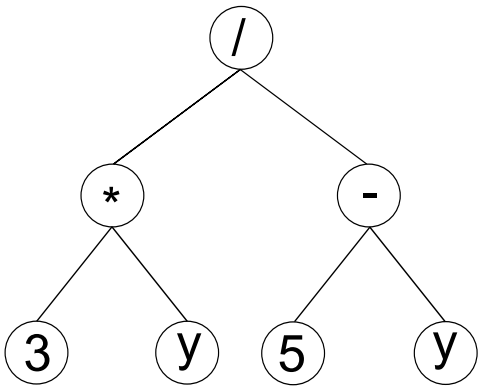
2.4 Genetické programovanie

Genetické programovanie, ktoré je jednou z najmladších evolučných techník, bolo predstavené v 80. rokoch 20. st. Johnom Kozom. GP bolo vyvinuté pre generovania programov hlavne v rámci symbolickej regresie [16].

GP pracuje so spustiteľnými štruktúrami. Jedinci majú tvar stromových štruktúr. Na reprezentáciu kandidátnych programov sa najčastejšie používajú:

- **stromy** – stromová reprezentácia je ukázaná na obrázku 2.3. Pre vykonanie programu je nutné prejsť strom,
- **lineárna reprezentácia** – lineárna reprezentácia je vlastne postupnosť inštrukcií,

- **obecný graf** – obecný graf je použitý napr. v kartézskom genetickom programovaní. Na obrázku 2.5 (v nasledujúcej časti) je uvedená reprezentácia formou orientovaného acyklického grafu.



Obrázok 2.3: Stromová reprezentácia výrazu $\frac{3y}{5-y}$.

Uzly stromu môžu byť terminály alebo neterminály. Terminály sú vstupy programu, funkcie bez argumentu alebo konštanty, neterminály predstavujú funkcie alebo operácie. Pred riešením úlohy pomocou GP je treba definovať:

- množinu terminálov,
- množinu neterminálov,
- spôsob výpočtu fitness hodnoty,
- parametre GP,
- spôsob určenia výsledku a ukončenie evolúcie.

2.4.1 Algoritmus GP

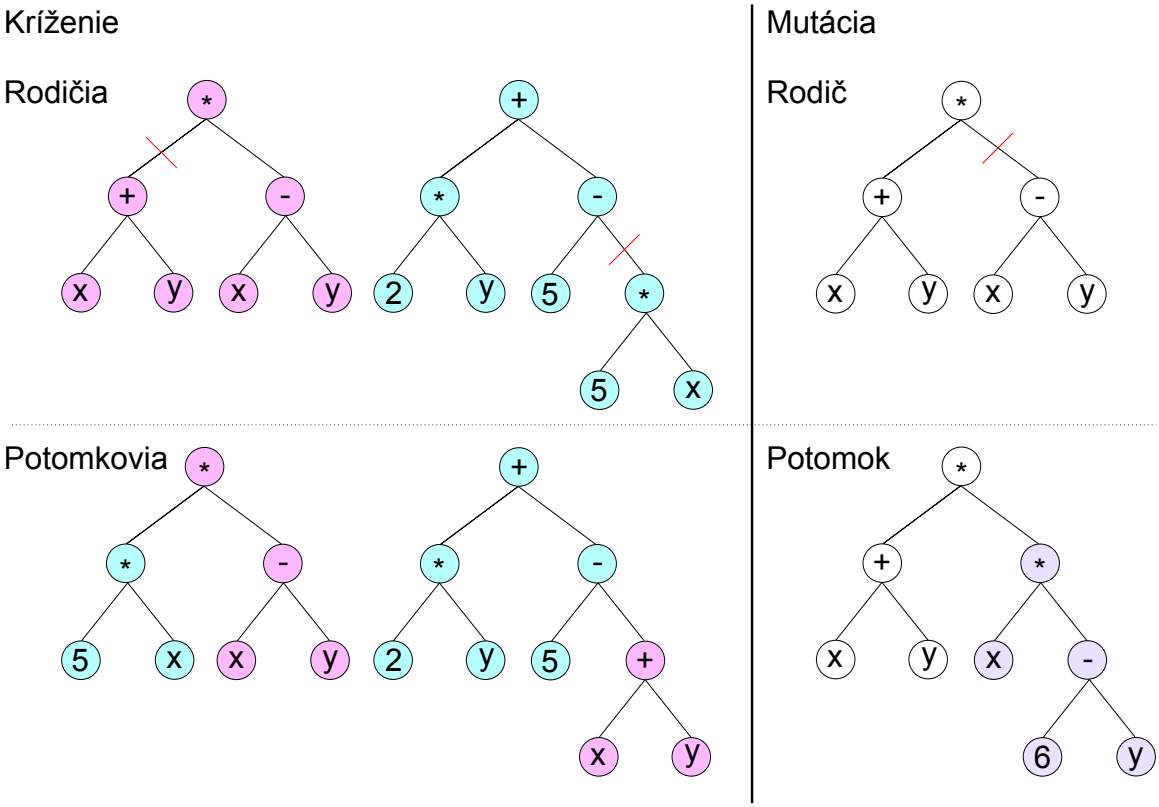
Prvým krokom algoritmu je vytvorenie počiatočnej populácie, na čo sa najčastejšie používajú tri metódy [10]:

- **Grow** – uzly stromu sú náhodne terminály alebo neterminály, vznikajú tak nepravidelné stromy,
- **Full** – do danej hĺbky sú uzly vybrané z množiny neterminálu, potom sa vyberá z terminálov,
- **Ramped Half-and-Half** – Táto metóda kombinuje Grow a Full metódy. Populácia je rozdelená na n-1 časti, kde n je maximálna hĺbka stromu. Každá skupina bude mať inú maximálnu hĺbku (2..n). Polovica každej skupiny je potom vygenerovaná pomocou metódy Grow, druhá polovica pomocou metódy Full.

Pri rekombinácii sa používajú operácie kríženia a mutácie. Pri krížení sa náhodne vyberá jeden podstrom každého rodiča a tieto podstromy sú vymenené. Tak vznikajú dvaja

potomkovia. Mutácia funguje tak, že je vybraný jeden podstrom, a ten podstrom je nahradený jedným náhodne vygenerovaným stromom. Priebeh mutácia a kríženia je zobrazený na obrázku 2.4.

Okrem kríženia a mutácie sa používajú aj iné operátory, ktoré môžu mať rôzne funkcie, ako napr. generovanie podprogramov.



Obrázok 2.4: Genetické operátory – kríženie a mutácia.

2.4.2 Fitness

Pre ohodnotenie jedinca je potrebné spustiť program, ktorý jedinec reprezentuje. Tento program je väčšinou testovaný pomocou trénovacej množiny. Jednotlivé výstupy programu sú porovnané s referenčnými výstupmi a následne podľa rozdielov referenčných a výstupných hodnôt je vypočítaná fitness hodnota.

Rozlišujeme štyri typy fitness funkcií [10]:

- **hrubý fitness** – fitness môže mať akúkoľvek hodnotu. Nie je definovaný, či je lepšia nižšia alebo vyššia hodnota. Hodnota je vlastne závislá na probléme.
- **štandardizovaný fitness** – hodnota fitnessu je vždy väčšia alebo rovná nule. Najkvalitnejší jedinec má fitness hodnotu nulu. Čím väčšia je hodnota, tým menej kvalitný je jedinec.
- **prispôsobený fitness** – je najpoužívanejším typom fitnessu. Štandardizovaný fitness je prepočítaná do intervalu $\langle 0,1 \rangle$. Najkvalitnejší jedinec má fitness hodnotu 1.

- **normalizovaný fitness** – fitness hodnota jedincov je v intervale ⟨0,1⟩ rovnako ako u prispôsobenom fitnesse, súčet fitness hodnôt všetkých jedincov je ale vždy 1.

2.4.3 Problémy GP

Ako všetky metódy, aj genetické programovanie má svoje nedostatky.

Prvým nedostatkom je rast dĺžky chromozómu počas evolúcie bez lepšenia fitness hodnoty. Tento efekt sa volá *bloat* [12]. Zväčšenie dĺžky chromozómov má negatívny efekt na výkon – vyhodnotenie jedincov zaberá viac času, väčšie chromozómy potrebujú väčšiu pamäť. Riešením tohto problému je napr. penalizácia dlhých chromozómov v fitness funkcii. Ďalším problémom je, že GP nie je vhodné na vytvorenie zložitých programov.

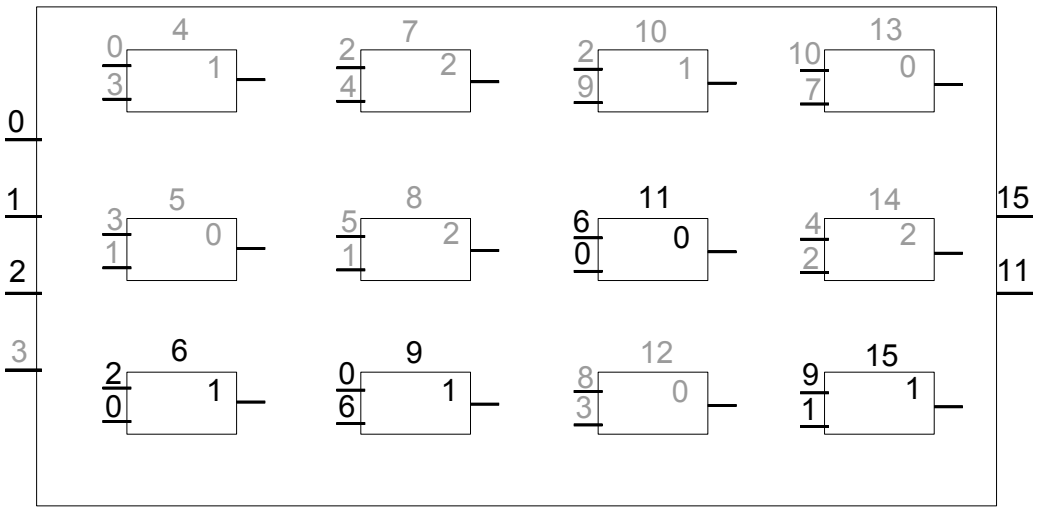
Často sa stane, že vyevolvovaný program funguje dobre len na trénovacie dáta, ale na testovacie nie. Tento jav sa nazýva overfitting a jeho riešením môže byť vhodne vybraná trénovacia množina.

2.5 Kartézske genetické programovanie (CGP)

Kartézske genetické programovanie [11] je variantom genetického programovania a bolo predstavené v roku 1999 J. Millerom. V CGP kandidátne riešenie je reprezentované orientovaným acyklickým grafom a je zakódované pomocou polí celých čísel.

2.5.1 Reprezentácia

Jedince v CGP sú reprezentované orientovaným acyklickým grafom. Uzly grafu sú usporiadané do dvojdimenzionálnej mriežky, kde počet stĺpcov (n_c), riadkov (n_r), vstupu (n_i) a výstupu (n_o) sa nemení počas výpočtu. Každý uzol realizuje jednu funkciu (f) z danej množiny funkcií (F), a má daný počet vstupov (n_n) a jeden výstup. Uzly môžu byť pripojené buď na vstup, alebo na výstup uzlu z predchádzajúcich stĺpcov. To, že koľko predchádzajúcich stĺpcov je možné používať, udáva parameter *L-back* (l). L-back môže mať hodnoty 1 až n_c(len predchádzajúci stĺpec až všetky stĺpce). Výstupy je možné pripojiť na hociktorý uzol, alebo priamo na vstup. Na obrázku 2.5 je príklad kandidátneho riešenia.



Obrázok 2.5: Reprezentácia kandidátneho riešenia CGP.

2.5.2 Zakódovanie CGP

Chromozóm je zakódovaný pomocou N celých čísel. N sa dá vypočítať pomocou nasledujúcej rovnice:

$$N = n_c \cdot n_r \cdot (n_n + 1) + n_o. \tag{2.1}$$

Primárne vstupy majú priradené hodnoty od 0 do n_i - 1, uzly sú indexované od n_i. Každý uzol je zakódovaný pomocou $n_n + 1$ čísel – n_n čísel pre vstupy uzlov a jedno číslo pre funkciu bloku. Funkcie sú tiež očíslované od 0. Posledných n_o čísla chromozómu označuje, že ktoré uzly sú prepojené na výstup.

Chromozóm uvedený na obrázku 2.5 má 4 vstupy (0-3), 12 uzlov (4-15) a 2 výstupy. Uzly majú 2 vstupy (ľavá strana) a jeden výstup. Číslo funkcie uzlu je číselná hodnota v každom uzle.
Zakódovanie chromozómu je nasledujúce:
0 3 1, 3 1 0, 2 0 1, 2 4 2, 5 1 2, 0 6 1, 2 9 1, 6 0 0, 8 3 0, 10 7 0, 4 2 2, 9 1 1, 15, 11
Jednotlivé uzly a výstupy sú oddelené čiarkou, nepoužité uzly sú označené šedou farbou.

Na rozdiel od GP, veľkosť genotypu je konštantná, ale veľkosť fenotypu sa môže líšiť u jednotlivých riešení. Pre grafy použitých v CGP platí [16]:

- niektoré uzly grafu môžu byť použité viackrát, iné uzly nemusia byť použité,
- niektoré vstupy uzlu nemusia byť použité,
- niektoré vstupy grafu nemusia byť použité.

2.5.3 Algoritmus

Na začiatku algoritmu je vygenerovaná počiatočná populácia. Používa sa stratégia $(1 + \lambda)$, čo znamená, že populácia je tvorená jedným rodičom a λ potomkami. CGP pracuje s malým počtom jedincov, ktoré sú vyvinuté po mnoho generácií. Pre výpočet fitness hodnoty jedincov, výstupy z kandidátneho programu sú porovnané s referenčnými výstupmi. Najlepšie ohodnotený jedinec sa stáva rodičom, alebo keď v predchádzajúcej generácii bol rodičom ten istý jedinec a je ešte iný jedinec s rovnakou fitness hodnotou, vyberie sa ten, ktorý ešte nebol rodičom. Nová populácia je vytvorená z najlepšieho jedinca pomocou mutácie, kríženie sa nepoužíva. Pri mutácii sa zmení náhodne vybraný gén – vstup uzlu, funkcia daného uzlu, alebo uzol, na ktorý je pripojený výstup. Nová hodnota zmutovaného génu musí byť validná hodnota pre daný gén.

Rozlišujeme mutáciu adaptívnu a neutrálnu. V prípade adaptívnej mutácie fitness hodnota jedinca sa zmení. Môžu sa aktivovať/deaktivovať zatiaľ neaktívne/aktívne uzly. Opakom je neutrálna mutácia. Keď mutácia je neutrálna, fitness hodnota sa nemení. Väčšinou zmutovaný gén kóduje neaktívny uzol, ale môže sa stať, že dôjde k zmene fenotypu, ale fitness hodnota bude rovnaká.

2.6 Koevolučné algoritmy

Zatiaľ zmienené algoritmy predpokladajú len jednu populáciu, prípadne to, že medzi populáciami neexistuje vzťah, ktorý by mal vplyv na iné populácie. V prírode je to práve naopak.

Jednotlivé populácie (druhy zvierat, rastlín atď.) ovplyvňujú smer vývoju ostatných. Tento jav sa volá koevolúcia. Podľa [9] koevolúcia znamená evolučnú zmenu vlastností jedincov jednej populácie v reakcii na vlastností jedincov druhej populácie a následne evolučnú zmenu vlastností jedincov druhej populácie v reakcii na vlastnosti prvej populácie.

Aj keď slovo koevolúcia naznačuje, že v koevolúcii sa zúčastňuje viac populácií, nemusí to byť pravda. Existujú aj koevolučné algoritmy s jednou populáciou. V tomto prípade jedinci interagujú s inými jedincami populácie podobne, ako v tradičných koevolučných algoritmov interagujú s jedincami inej populácie.

Koevolučné algoritmy sú najčastejšie použité v interaktívnych doménach, kde kvalita kandidátneho riešenia sa väčšinou nedá algoritmicky vypočítať. Fitness hodnota jedincov je daná výsledkom interakcií s inými jedincami [13].

2.6.1 Základné pojmy koevolučných algoritmov

Jedinec

V evolučných algoritmoch jedinci reprezentujú kandidátne riešenie na daný problém. V koevolučných algoritmoch jedinci môžu reprezentovať len časť riešenia, ale v riešení sa vôbec nemusia zúčastniť – napr. jedinci reprezentujúce testy pre zvýšenie kvality riešenia.

Populácia

Populácia, podobne ako v evolučných algoritmoch je množina jedincov, ktorí sa vyvíjajú v čase. Hlavný rozdiel je to, že v koevolučných algoritmoch obvykle pracujeme s viacerými populáciami, ale sú aj koevolučné algoritmy pracujúce s jednou populáciou. V prípade, že pracujeme s jednou populáciou, daní jedinci musia byť rozdelení od seba nejakou vhodnou bariérou.

Archívy

V koevolučných algoritmoch sa často používajú archívy. Archívy slúžia pre uchovanie niektorých jedincov, ktoré môžu byť potom použité napr. pre ohodnotenie iných jedincov.

Riešenie problému

Ako už tu bolo zmienené, riešením evolučného algoritmu je najlepšie ohodnotený jedinec na konci evolúcie. V koevolučných algoritmoch riešením môže byť jeden jedinec z niektorej populácie alebo z archívu, prípadne viac jedincov, väčšinou z viacerých populácií alebo archívov.

2.6.2 Ohodnotenie jedincov

Algoritmus koevolučných algoritmov je veľmi podobný evolučným algoritmom – vytvárajú sa počiatočné populácie, pre rekombináciu sa používa kríženie a mutácia, jedinci sa vyvíjajú mnoho generácií. Čo sa najviac líši od evolučných algoritmov, je ohodnotenie jedincov. V koevolučných algoritmoch fitness hodnota jedincov je vypočítaná na základe interakcií jedinca s inými jedincami.

Najjednoduchším spôsobom je interakcia jedinca so všetkými ostatnými jedincami. Nevýhodou tohto prístupu je veľká časová a výpočtová náročnosť. Práve preto počet interakcií je nutné minimalizovať. Na minimalizáciu interakcií sú dva prístupy. Prvým prístupom je,

že interakcie sú vybrané pre každého jedinca zvlášť. Druhým prístupom je vybrať interakcie pre celú populáciu. Po interakciách je vypočítaná fitness hodnota pre jednotlivé jedince. Výsledky predchádzajúcich krokov evolúcie sú často použité na výber interakcií nasledujúcich krokov.

Výpočet fitness hodnoty môže byť ťažký, keď je vypočítaný z viac interakcií. V tomto prípade je možné vypočítať z hodnôt získaných z interakcií jednu fitness hodnotu – napr. vybrať najlepší/najhorší fitness, alebo vypočítať priemer týchto hodnôt. Druhou metódou je pracovať s fitness hodnotou, ako n-ticou čísel a z n-tice vybrať najlepší fitness, napr. Paretovské optimum.

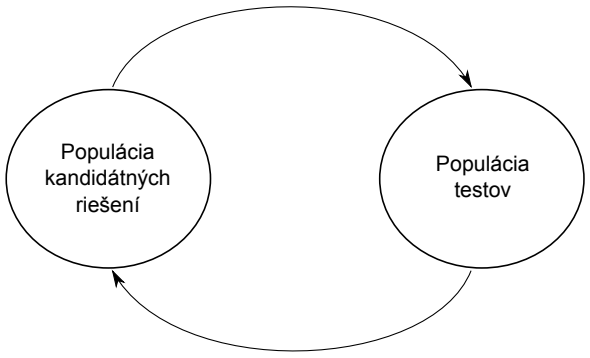
Pre interakcie jedincov je potrebné zaistiť komunikáciu medzi populáciami. Komunikácia môže byť synchrónna alebo asynchrónna. V prípade asynchrónnej komunikácie, populácie (jedince) komunikujú pomocou zdieľanej pamäti. Nová informácia je napísaná a získaná zo zdieľanej pamäti. Pri synchrónnej komunikácii k výmene informácii dochádza v daných časových intervaloch.

2.6.3 Rozdelenie koevolučných algoritmov

Historicky koevolučné algoritmy boli rozdelené na kooperatívne a súťaživé. Toto rozdelenie pochádza z teórie hier. Podľa [13] toto rozdelenie nie je vhodné pre klasifikáciu problémov a algoritmov, ktoré majú väčší vplyv na chovanie koevolučných algoritmov, ako interaktívna doména. Preto je lepšie rozdeliť koevolučné algoritmy podľa toho, čo predstavuje lepšie riešenie, na kompozičné úlohy a na úlohy založené na teste.

Úlohy založené na teste

V úlohách založených na teste sú rozdelené populácie kandidátnych riešení a populácie testov. Cieľom populácií kandidátnych riešení je čím lepšie splniť testy, kým cieľom populácií testov je vytvoriť také testy, ktoré sú ťažšie splniteľné kandidátnym riešením. Schéma úlohy založenej na teste je uvedená na obrázku 2.6.

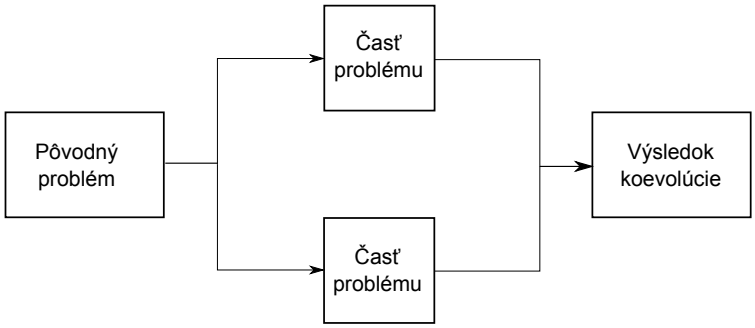


Obrázok 2.6: Úlohy založené na teste.

Kompozičné úlohy

Koevolúcia môže byť veľmi dobrá voľba, keď chceme riešiť nejakú komplexnú úlohu, ktorú je možné rozdeliť na podúlohy. Každú podúlohu potom môže riešiť iná populácia a riešenie problému bude zložené z jedincov všetkých populácií. Tieto úlohy sú kooperatívne, pretože

jednotlivé populácie pracujú spolu, aby výsledok bol čo najkvalitnejší. Schéma kompozičnej úlohy je zobrazená na obrázku 2.7.



Obrázok 2.7: Kompozičné úlohy.

Viac publikácií sa zaoberá tematikou kompozičných koevolučných algoritmov. Jedna z najstarších je práca Husbanda a Milla [6] o rozvrhovaní dielenských zákaziek. V tejto práci sú využité populácie kódujúce plány procesu jednotlivých komponentov, iné populácie skúsia vytvoriť plán pre najlepšiu spoluprácu pri výrobe komponentov. Fitness funkcia tých jedincov zahrňuje použitie zdieľaných zdrojov, čo znamená, že bez potreby ďalšieho plánovania výsledok evolúcie bude efektívne plánovaný. Ešte jeden typ populácií je použitý: populácia *Arbitrator* (rozhodcovia). *Arbitrator*-y riešia konflikty medzi ostatnými populáciami. Fitness hodnota jedincov závisela na tom, ako dobre vyriešili problémy.

Kapitola 3

Obrazové filtre

Účelom obrazových filtrov je potlačiť alebo zvýrazniť jednotlivé štruktúry obrazu, ako potlačenie šumu, odhalenie hrán, atď. Táto kapitola sa zaoberá s potlačením/odstránením šumu.

Šum (poškodenie obrázku) môže vzniknúť pri vytvorení alebo posielaní obrázkov. Existuje mnoho typov šumu [8], najčastejšie sa stretneme so šumami typov:

- impulzný šum: hodnota niektorých pixelov je zmenená na minimálnu alebo maximálnu hodnotu,
- gaussový šum: každý pixel obrazu je poškodený. Miera poškodenia pixelov má normálne rozloženie, ktoré je možné popísať s rovnicou [4]:

$$p(z) = \frac{1}{\delta\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\delta^2}}, \tag{3.1}$$

  kde μ je stredná hodnota šumu a σ^2 je jeho rozptyl.

Na opravu takých poškodených obrázkov vznikli rôzne filtre. Tieto filtre vlastne aplikujú rôzne funkcie na daný obrázok (pixely obrázku), ktorým výstupom je filtrovaný obrázok (filtrované pixely). Tieto funkcie môžu byť lineárne alebo nelineárne. Pre lineárne funkcie platí princíp superpozície:

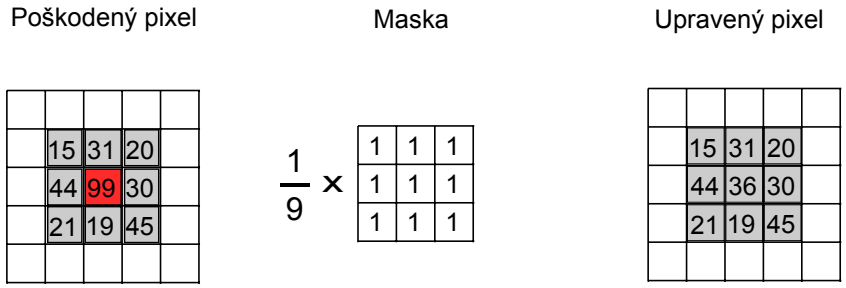
$$f(o_1) + f(o_2) = f(o_1 + o_2). \tag{3.2}$$

Okrem lineárnych filtrov existujú aj filtre nelineárne, ktoré sú tiež často použité v spracovaní obrazov.

3.1 Konvenčné obrazové filtre

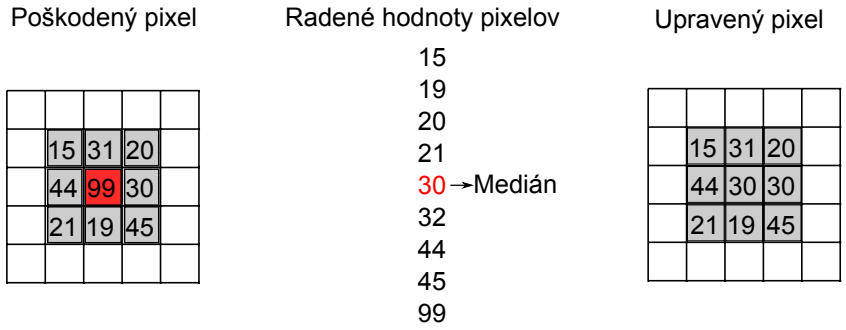
Najjednoduchšie filtre, ktoré sa používajú na odstránenie šumu sú filtre lineárne. Jednotlivé hodnoty pixelov sú vypočítané konvolúciou susedných pixelov a použitím konvolučnej masky. Väčšinou sú použité pixely z okolí 3x3, 5x5 alebo 7x7. Výsledná hodnota pixelu bude vážená priemerná hodnota daných pixelov. Na obrázku 3.1 je uvedený príklad použití konvolučného filtra. Nevýhodou lineárnych filtrov je, že často rozmažú výsledný obrázok, čo je nežiadúce.

Často sa používajú aj filtre nelineárne. Asi najpopulárnejším nelineárnym filtrom je filter medián. Mediánový filter pracuje tak, že hodnota pixelu je vypočítaná ako medián



Obrázok 3.1: Konvolučný filter.

susedných pixelov. Je vhodný hlavne pre odstránenie výstrelového šumu. Na obrázku 3.2 je príklad použitia mediánového filtra na jeden pixel.



Obrázok 3.2: Mediánový filter.

Mediánový filter nedáva dobré výsledky, keď veľká časť obrazu je poškodená (viac ako 40% pixelov). Preto vznikli rôzne modifikácie mediánového filtra, ako napr. adaptívny mediánový filter [7], vážený mediánový filter [1], atď., ktoré riešia tento problém.

3.2 Evolučné obrazové filtre

Evolučné algoritmy sa používajú pre riešenie problémov vo viacerých oblastiach. Jeden z nich je spracovanie obrazu.

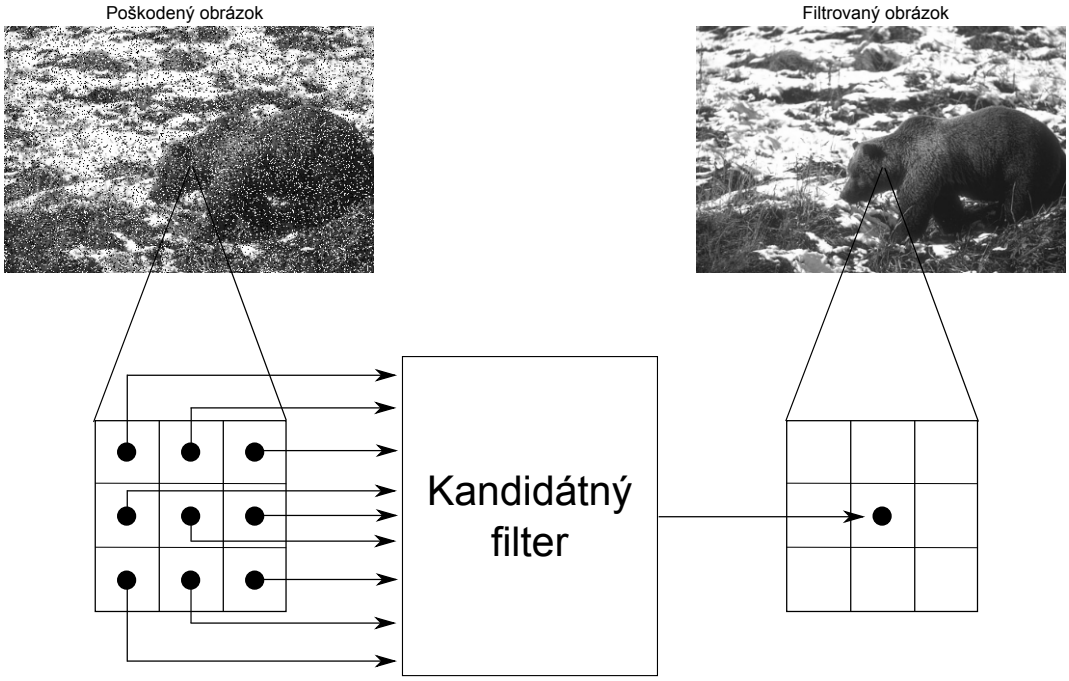
Konvenčné filtre, ktoré sú používané pre odstránenie šumu v obraze väčšinou nefungujú dobre, keď je poškodená veľká časť obrázku. Pomocou evolučných algoritmov je možno vytvoriť také filtre, ktoré dávajú dobré výsledky aj pre takto poškodené obrázky.

Pre vytvorenie obrazových filtrov sa najčastejšie používa kartézske genetické programovanie. Vstupy CGP pre filter sú susedné pixely práve sledovaného pixelu, výstupom je jeho filtrovaná hodnota. Na obrázku 3.3 je zobrazená schéma filtra, ktorá bola vyvinutá pomocou CGP. Príklad množiny funkcií použitých pre návrh obrazových filtrov je zobrazený v tabuľke 3.1.

Pre vytvorenia filtra pomocou evolučného algoritmu, potrebujeme trénovaciu množinu poškodených obrázkov a ich nepoškodených verzií. Fitness hodnota filtrov je vypočítaná z rozdielov medzi originálnym obrazom a filtrovaným obrazom. Cieľom programu je minimalizovať tento rozdiel. Po skončení evolúcie vyvinuté filtre sú testované na testovacích obrázkoch.

Kód	Funkcia	Kód	Funkcia
0	255	8	$x \gg 1$
1	x	9	$x \gg 2$
2	255 - x	10	$swap(x,y)$
3	$x \vee y$	11	$x + y$
4	$\neg x \vee y$	12	$x +^{S} y$
5	$x \wedge y$	13	$(x + y) \gg 1$
6	$\neg(x \wedge y)$	14	$max(x,y)$
7	$x \oplus y$	15	$min(x,y)$

Tabuľka 3.1: Príklad použitých funkcií pri návrhu obrazových filtrov [15].



Obrázok 3.3: Návrh obrazových filtrov pomocou CGP.

3.3 Obrazové filtre a koevolúcia

Pomocou CGP je možné vytvoriť kvalitné obrazové filtre. Nevýhodou CGP však je, že je veľmi časovo a výpočtovo náročný. Na zrýchlenie výpočtu existujú rôzne prístupy, ako napr. použitie rekonfigurovateľných zariadení (FPGA), alebo využitie grafických jednotiek (GPU) v počítačoch.

Pomocou koevolúcie je možné urýchliť výpočet. V článku [18] bola ukázaná metóda, kde zrýchlenie bolo dosiahnuté znížením počtu využitých pixelov pre výpočet fitness jednotlivých jedincov.

Boli používané dve populácie: jedna pre vývoj filtra pomocou CGP a druhé na výber podmnožiny pixelov pre výpočet fitness hodnoty pomocou genetického algoritmu.

Kapitola 4

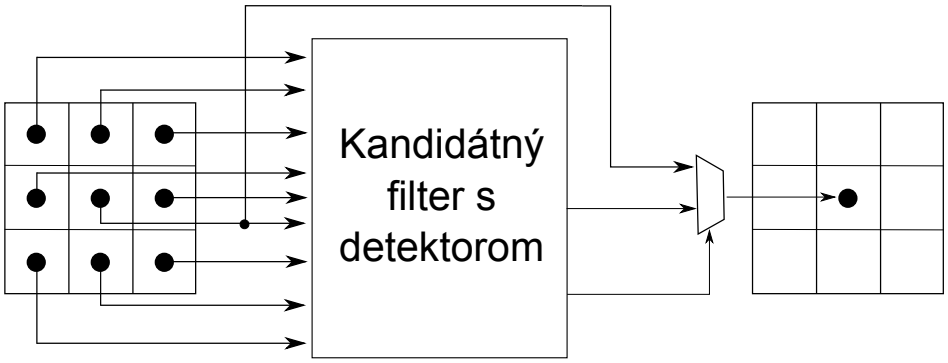
Návrh obrazových filtrov s detektorom šumu

V tejto kapitole je popísaný návrh obrazových filtrov s detektorom šumu. Prvým krokom návrhu je vybrať metódy a techniky, ktoré budú použité v programe – vybrať typ evolučného algoritmu, typ koevolúcie, ako bude program paralelizovaný, atď., potom tieto komponenty spojiť, rozšíriť, kým nemáme navrhnutý celý program. V tejto kapitole sú podrobnejšie popísané jednotlivé použité techniky.

Jeden z problémov obrazových filtrov pre odstránení šumu je to, že opravia (zmenia hodnotu) i nepoškodených pixelov. Aby k tomu nedošlo, často sa používajú detektory šumu, ktoré najprv zistia, ktoré pixely sú poškodené, a potom len tieto poškodené pixely budú filtrom opravené.

V článku [21] bola predstavená metóda pre vývin filtrov kombinovaných s detektorom šumu pre odstránení robustného impulzného šumu.

Filter a detektor sú zakódované do jedného chromozómu. CGP má dva výstupy: filtrovaná hodnota pixelu, a pravdivostná hodnota, či je pixel poškodený alebo nie. Na základe tejto pravdivostnej hodnoty bude nastavená hodnota pixelu na originálnu (nefiltrovanú), alebo na filtrovanú hodnotu. Schéma filtra je zobrazená na obrázku 4.1.



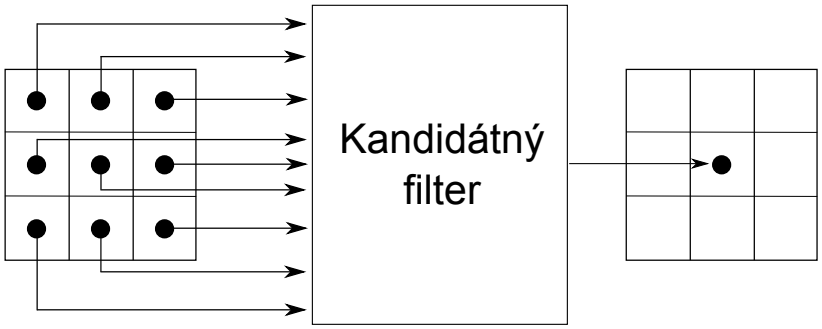
Obrázok 4.1: Schéma filtra kombinovaná s detektorom šumu z článku [21].

Táto práca sa tiež zaoberá návrhom obrazových filtrov a detektorov šumu, ale pomocou koevolúcie. To znamená, že filter a detektor nebudú zakódované do jedného chromozómu, ale práve naopak filter i detektor budú zakódované do iných chromozómov, a budú tvoriť odlišné populácie. Tieto populácie počas výpočtu budú interagovať, aby výsledná dvojica

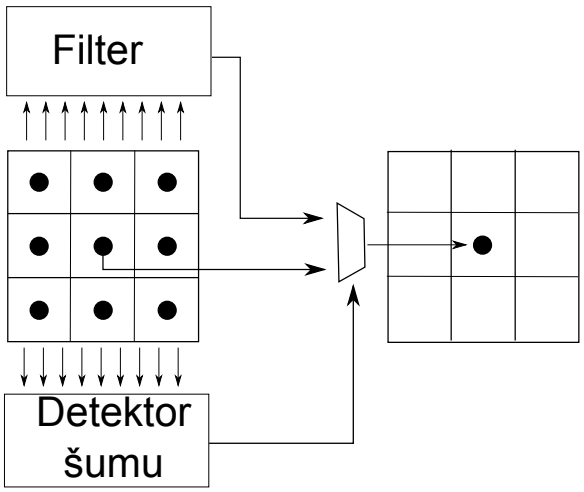
detektor - filter bola čo najkvalitnejšia.

Aby bolo možné zistiť, či pomocou koevolúcie obrazových filtrov a detektorov šumu môžeme získať kvalitnejšie riešenie ako bez koevolúcie, je nutné ich porovnať s verziami bez koevolúcie. V tejto práci budú implementované tri verzie:

- s použitím len filtra, bez detektora šumu a koevolúcie (obrázok 4.2),
- s použitím filtra a detektora šumu vyvinutých bez koevolúcie (obrázok 4.3),
- s použitím filtra a detektora šumu vyvinutých pomocou koevolúcie (obrázok 4.3).



Obrázok 4.2: Schéma filtra.



Obrázok 4.3: Schéma filtra kombinovaná s detektorom šumu.

Jednotlivé verzie budú podrobnejšie popísané v ďalších kapitolách.

4.1 Evolučný algoritmus

Existuje viac typov evolučných algoritmov, ktoré sú aplikovateľné na rôzne problémy. Na vytvorenie obrazových filtrov sa často používa kartézske genetické programovanie na princípoch predstavených v kapitole 3.2. Hodnoty okolitých pixelov práve skúmaného pixelu slúžia ako primárne vstupy obvodu CGP, výstupom je opravená hodnota pixelu.

Pretože vstupy CGP sú okolité pixely skúmaného pixela, krajné pixely obrazu, ktoré nemajú všetky potrebné susedné pixely budú ignorované počas výpočtu.

Detektor šumu pracuje podobne ako filter, len jeho výstupom je pravdivostná hodnota – či skúmaný pixel je poškodený alebo nie. Kvôli týmto podobnostiam medzi detektorom a filtrom som vybral takú istú metódu pre ich implementáciu – aj filter aj detektor bude vyvinutý pomocou CGP.

Priebeh evolúcie

Evolučný algoritmus sa začína s generovaním počiatočnej generácie. Počiatočná generácia je generovaná náhodne, ale pretože výpočet môže trvať aj dlhšiu dobu, pridal som možnosť pokračovania výpočtu. V prípade pokračovania jeden jedinec je načítaný zo súboru (pre každú populáciu) a ostatné jedince populácie budú jeho mutanti.

V každej generácii je vybraný jeden rodič (z každej populácie). Rodičom sa stáva jedinec z najlepšie ohodnotených dvojíc/jedincov (kapitola 4.2.1). Keď viac dvojíc/jedincov má rovnaké ohodnotenie, rodičom sa stáva jedinec (z dvojice), ktorý ešte rodičom nebol.

Z rodičov sú vytvorení potomkovia. Jediný operátor, ktorý sa používa na rekombináciu je mutácia. Mutácia zmení náhodne vybrané gény na náhodné (ale platné) hodnoty. Aby nebolo treba pri každej mutácii vypočítať platné hodnoty chromozómu, tieto hodnoty sú vypočítané a uložené na začiatku výpočtu. Pri generovaní počiatočnej generácie sú tiež použité tieto hodnoty.

Výpočet fitness funkcie je popísaný v kapitole 4.2.1. Evolúcia končí po daných počtoch generácií, prípadne keď bola dosiahnutá maximálna fitness hodnota.

4.2 Koevolúcia

Použitie koevolúcie má veľa výhod, ale má aj isté nevýhody. Použitie koevolučného algoritmu môže urýchliť výpočet a dávať lepšie výsledky ako štandardné evolučné algoritmy, ale je treba implementovať niekoľko dodatočných modulov, aby sme dostali funkčný koevolučný algoritmus.

Prvou úlohou je rozdeliť úlohy. Rozdelenie úloh je v tomto prípade jednoduché: jedna populácia pre detektory a druhá pre filtre. Koevolúcia má kompozičný charakter, najlepšia kombinácia jednotlivých jedincov oboch populácií bude tvoriť riešenie.

Interakcia medzi jedincami populácií je ďalším problémom. Musíme sa rozhodnúť, koľko jedincov bude z prvej populácie interagovať s koľkými jedincami z druhej. Na to máme niekoľko možností – každý jedinec s každým, jeden jedinec s viacerými jedincami, len najlepší jedinec z oboch populácií atď.

V programe bude každý jedinec prvej populácie interagovať s každým jedincom druhej populácie. Napriek tomu, že časová náročnosť tejto verzie je najvyššia, vybral som túto metódu, pretože s väčšou pravdepodobnosťou nájdeme dvojice jedincov, ktoré dobre spolupracujú.

V mnohých prípadoch koevolučných algoritmov koevolúcia nie je použitá v každej generácii – jednotlivé populácie sa vyvíjajú sami, a s ostatnými populáciami sa interagujú len po niekoľkých generáciách.

V koevolučných algoritmoch sú často použité aj archívy, ktoré môžu mať rôzne funkcie (kapitola 2.6.1). V prípade, že populácie neinteragujú v každej generácii, a fitness hodnota jedincov je vypočítaná inak, môže sa stať, že fitness nových jedincov bude menší, ako ich

predchodcov. Preto bol pridaný jednoduchý archív. Do archívu je uložená zatiaľ nájdená najlepšia dvojica detektor – filter.

4.2.1 Fitness

V prípade detektoru šumu fitness jedinca závisí na správne klasifikovaných pixeloch – či je pixel poškodený alebo nie. Čím viac dobre klasifikovaných pixelov máme, tým väčšia je fitness hodnota.

Klasifikácia je na základe najvýznamnejšieho bitu (MSB) výstupom detektoru. Keď najvýznamnejší bit je 1 (výstup detektoru je väčší ako 127), pixel je detekovaný ako poškodený.

V prípade obrazových filtrov fitness hodnota väčšinou závisí na rozdieloch medzi originálnymi a opravenými hodnotami pixelov obrázku. Často sa používa špičkový pomer signálu k šumu (PSNR – peak signal-to-noise ratio) alebo stredná odchýlka pixelov (MDPP – Mean Difference Per Pixel) [15]:

$$PSNR = 10 \cdot log_{10} \frac{255^2}{(1/MN) \sum_{i,j} (v(i,j) - w(i,j))^2}, \tag{4.1}$$

$$MDPP = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} \mid v(i,j) - w(i,j) \mid, \tag{4.2}$$

kde M a N sú rozmery obrázku, v a w sú filtrovaný resp. originálny obrázok.

V prípade PSNR väčšia hodnota znamená kvalitnejšieho jedinca. U MDPP je to práve naopak. Ak používame MDPP, našim cieľom je minimalizovať fitness hodnotu. V projekte som sa rozhodol implementovať obidve metódy a zistiť, ktorá z nich je vhodnejšia na danú úlohu.

Aj keď vo všetkých implementovaných metódach bude použitá na výpočet fitness hodnoty filtra jedna z týchto dvoch funkcií, v jednotlivých metódach sa líši sada pixelov, ktoré sú použité pre výpočet.

V prípade, keď vytvoríme len filter, na výpočet sú použité všetky pixely. Keď máme filter a detektor bez koevolúcie, môžeme vypočítať fitness použitím všetkých pixelov, alebo len pomocou poškodených. Keď používame len poškodené pixely pre trénovanie, môžeme získať kvalitnejší filter (pri predpoklade, že detektor detekuje poškodené pixely spoľahlivo), a i časová náročnosť vývoja filtra klesá.

V koevolučných algoritmoch ohodnotenie jedincov závisí na interakcii s inými jedincami. V navrhnutom koevolučnom algoritme fitness hodnota nie je zvlášť vypočítaná ani pre jednu populáciu – pracujeme s dvojicami, detektor a filter. V prvom kroku detektor zistí, ktoré pixely sú poškodené, a na tieto pixely je potom použitý filter. Ostatné (nedetekované) pixely nie sú filtrované, ale sú použité vo výpočte fitness hodnoty.

V generáciách, keď filtre a detektory neinteragujú, je fitness hodnota vypočítaná podobne ako vo verzii s detektorom a filtrom bez koevolúcie.

Zrýchlenie výpočtu fitness hodnoty

Výpočet fitness funkcie je časovo najnáročnejším krokom evolučných algoritmov. V prípade kartézskeho genetického programovania pre výpočet fitness hodnoty je treba vypočítať hodnoty jednotlivých uzlov, aby mohli byť použité ako vstupy ďalších uzlov.

Pre výpočet fitness hodnoty jedinca je treba vypočítať pomocou kandidátneho filtra filtrovanú hodnotu každého pixelu, k tomu je potrebné v každom prípade vypočítať hodnoty uzlov filtra.

Väčšinou nie všetky uzly sú použité v jedincovi. Hodnoty neaktívnych uzlov nepotrebujeme, takže ich výpočet je zbytočný. Aby sme zistili, ktoré uzly sú použité a ktoré nie, je nutné prejsť celý graf: na začiatku sú všetky uzly neaktívne. Najprv označíme uzol, ktorý je pripojený na výstup, ako aktívny, potom označíme rekurzívne všetky neaktívne uzly, ktoré sú použité aktívnymi uzlami.

4.3 Paralelizácia

Cieľom tejto práce nie je zrýchlenie alebo optimalizácie výpočtu, ale pre získanie dobrých výsledkov a pre rôzne experimenty je treba spustiť program niekoľkokrát s rôznymi nastaveniami. Bez optimalizácie a paralelizácie jeden beh môže trvať hodiny, keď pracujeme s viacerými väčšími obrázkami, tak aj viac ako deň, čo je nežiadúce. Preto je treba výpočet urýchliť. Čo sa týka paralelizácie, máme niekoľko možností. Najjednoduchšie je celý algoritmus implementovať sekvenčne. Nevýhodou tohto postupu je veľká časová náročnosť výpočtu. Preto je vhodné použiť paralelný algoritmus.

Keďže pracujeme s dvoma populáciami, môžeme použiť dve vlákna, jedno pre každú populáciu. V tomto prípade, keď používame koevolúciu, je nutné koordinovať komunikáciu medzi populáciami. V prípade synchrónnej komunikácie je treba zvoliť vhodný časový interval pre synchronizáciu. Častá synchronizácia veľmi spomalí výpočet, v prípade príliš zriedkavej synchronizácie nie sú zdieľaní niektorí jedinci, čo môže spomaliť evolúciu.

Použitie asynchrónnej komunikácie má výhodu v tom, že nedôjde k zbytočnému synchronizovaniu (napr. keď nedošlo k zmene chromozómu) – najlepší jedinci sú uložení do zdieľanej pamäti a v prípade zmeny sa použije už nový jedinec.

Tiež je možné paralelizovať len časť algoritmu. Časovo najnáročnejšia časť evolučných algoritmov je výpočet fitness funkcie. V našom prípade fitness funkcia vypočíta novú hodnotu každého pixelu (kapitola 4.2.1). Tento výpočet je dobre paralelizovateľný, pretože pre každý pixel je aplikovaná tá istá funkcia, prípadne je možné vypočítať fitness hodnoty jedincov paralelne.

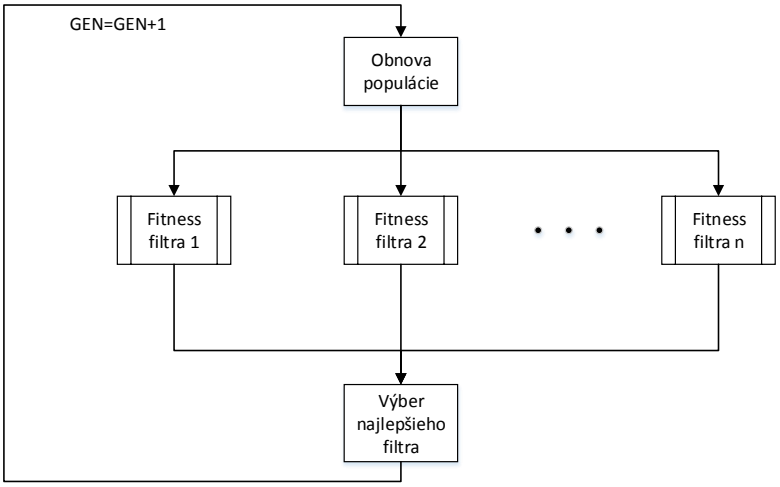
V práci bola vybraná nasledujúca metóda paralelizácie:

v prípade filtra sú fitness hodnoty jednotlivých jedincov vypočítané paralelne (obrázok 4.4).

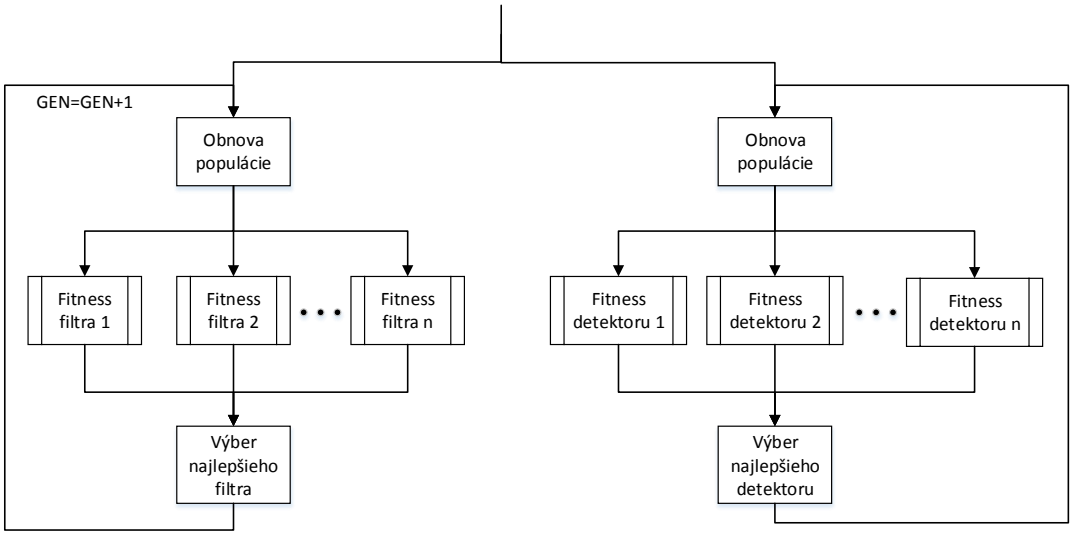
Keď pracujeme s filtrom a detektorom šumu bez koevolúcie, program pracuje na dvoch vláknach – jeden pre detektor a jeden pre filter. Fitness hodnoty jednotlivých jedincov sú ale vypočítané tiež paralelne, podobne, ako v prípade použitia len filtrov (obrázok 4.5).

Keď použijeme aj koevolúciu, máme dve možnosti – keď použijeme koevolúciu v každej generácii, pre každý filter je vypočítaný kombinovaný fitness na inom vlákne (obrázok 4.6).

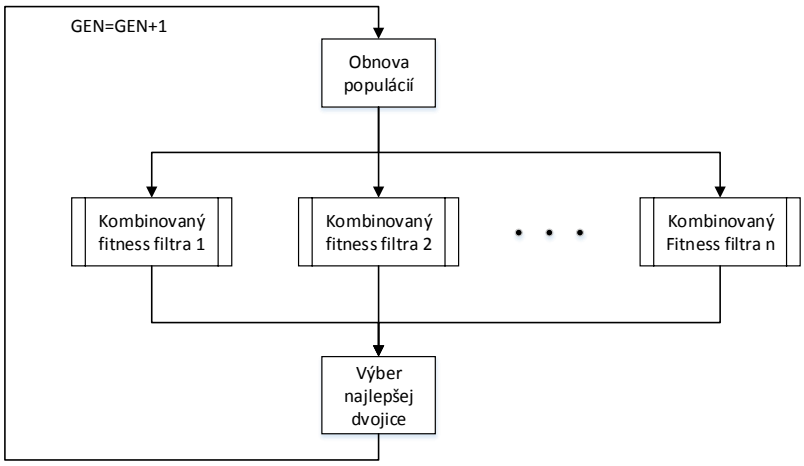
V prípade, keď interakcia populácií je len po niekoľko generácií, je používaná kombinácia metód použitých pri verzii s detektorom a filtrom bez koevolúcie a z verzie, keď v každej generácii je použitá koevolúcia (obrázok 4.7).



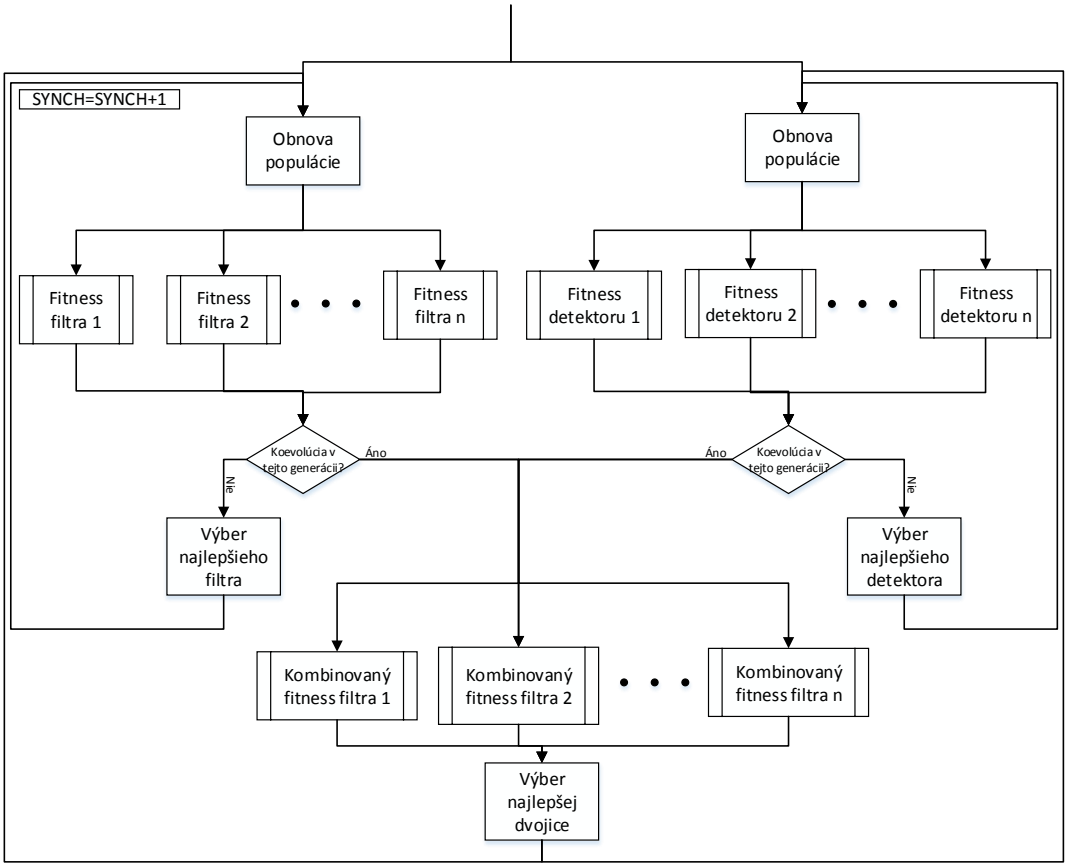
Obrázok 4.4: Paralelný výpočet fitness hodnoty filtrov.



Obrázok 4.5: Paralelný výpočet fitness hodnoty filtrov a detektorov bez koevolúcie.



Obrázok 4.6: Paralelný výpočet fitness hodnoty filtrov a detektorov s koevolúciou.



Obrázok 4.7: Paralelný výpočet fitness hodnoty filtrov a detektorov s koevolúciou.

Kapitola 5

Implementácia

Program som implementoval v jazyku C++. Okrem štandardných C++ knižniciach používam OpenCV pre načítanie a zápis obrázkov, a OpenM pre paralelizáciu.

Program sa skladá z niekoľkých modulov, a je rozdelený do niekoľko zdrojových súborov. Jednotlivé moduly budú podrobnejšie popísané v nasledujúcich kapitolách.

Návod na preklad a spustenie programu je popísané v prílohe B.

5.1 Použité štruktúry a datové typy

V programe vytvorené a použité štruktúry sú definované v hlavičkovom súboru *types.h*. V programe som definoval tri štruktúry:

- chromozóm je reprezentovaný dátovým typom **chromosome**, čo je vlastne pole typu **int**. Chromozóm obsahuje vstupy a funkcie jednotlivých blokov a výstup,
- ďalšia štruktúra je použitá pre uloženie obrazov (**image**), obsahuje tri položky: počet stĺpcov a riadkov obrazu (**int**), a dvojdimenzionálne pole typu **int** pre hodnoty pixelov,
- poslednou štruktúrou je **validValues**, táto štruktúra je použitá pri vytvorení počiatočnej generácie a pri mutácii. Obsahuje dve položky: počet možných hodnôt (**int**), a pole (**int***) hodnôt. Na začiatku programu je vytvorené pole typu **validValues** veľkostí dĺžky chromozómu, a pre každý gén sú nastavené jeho možné hodnoty.

5.2 Triedy *filter* a *detector*

Pre filter a detektor boli vytvorené triedy, ktoré reprezentujú jedného jedinca v populácii. Triedy obsahujú všetky potrebné metódy pre evolúciu. Pretože filter a detektor majú veľa spoločných rysov, tieto triedy sú podobné. Jednotlivé metódy budú dôkladnejšie popísané.

Konštruktor oboch tried inicializuje niektoré potrebné premenné. Obe triedy obsahujú metódy pre nastavený obrázok, ktorými budú jedince pracovať (**initImg()**), pre generovanie jedinca (**generate()**), pre výpočet fitness hodnoty (**fitness()**), pričom používa ďalšiu metódu pri zistení, ktoré bloky sú použité (**getUsedBlocks()**), metóda pre získanie počtu použitých blokov (**numUsedBlocks()**), pre mutáciu (**mutate()**), pre nastavenie chromozómu (**setChromosome()**) – hlavne použitý pre mutáciu a pre vrátenie chromozómu (**getChromosome()**).

Detektor okrem týchto metód obsahuje metódu pre vrátenie detekovaných pixelov (**getDetectedPixels()**), a filter podobne metódu pre nastavenie tých pixelov (**setDetectedPixels()**). Filter obsahuje ešte jednu metódu pre nastavenie výstupného obrazu (**outImg()**).

5.2.1 Generovanie (pseudo) náhodných čísel

V evolučných algoritmoch používanie náhodných čísel je veľmi dôležité – používa sa na generovanie počiatočnej generácie a aj počas evolúcie (napr. v mutácií).

Aby v každom behu sme mohli získať iné výsledky, musíme náhodný generátor inicializovať inou hodnotou. Na inicializáciu sa často používa funkcia **time()**.

V paralelnom spracovaní (pomocou OpenMP) sa môžeme stretnúť s problémom, že v iných vláknach sú generované tie isté čísla. Možným riešením je inicializovať náhodný generátor v každej vlákne iným číslom.

Pretože náhodné čísla sú použité len počas evolúcie, a sú použité len metódami tried **filter** a **detector**, rozhodol som sa pridať vlastnú metódu pre generovania pseudonáhodných čísel **lcg_rnd()** do tých tried. Ide o jednoduchý lineárny kongruentný generátor, ktorý vráti pseudonáhodné číslo vypočítané z predchádzajúceho pseudonáhodného čísla:

$$X_{n+1} = (a \cdot X_n + c) \mod m, \tag{5.1}$$

kde m=2^{32}, a=1664525 a c=1013904223 [14].

Inicializácia pseudonáhodných generátorov prebieha ešte v sekvenčnej časti programu, aby každý generátor bol inicializovaný iným číslom.

5.3 Práca so súbormi

Program pracuje s dvoma typmi súborov: s obrázkami a s textovým súborom. Pre oba typy súborov sú implementované funkcie pre čítanie a zápis. V prípade obrázkov je použitá funkcia **imread()**, pomocou ktorej je možné jednoducho načítať obrázky uložené v rôznych formátoch. Pre zápis obrázkov je tiež použitá funkcia z knižnice OpenCV: **imwrite()**.

Do textových súborov sú uložené jednotlivé chromozómy. Pre načítanie a zápis chromozómov je použitá trieda **ifstream**/**ofstream**.

5.4 Funkcia main()

Jadro programu tvorí funkcia **main()**, ktorá najprv inicializuje potrebné premenné. Ďalšou jej úlohou je spracovanie parametrov, a podľa parametrov zavolať zodpovedajúce funkcie.

Funkcia **main()** tiež riadi evolúciu, a paralelné spracovanie. Používanie knižnice OpenMP zjednodušuje paralelné spracovanie. V programe sú použité dve techniky paralelizácie:

- použitie sekcií – *#pragma omp section*, ktorý je použitý pre rozdelenie na evolúciu filtrov a detektorov v generáciách, keď detektor a filter spolu neinteragujú,
- použitie paralelného spracovania cyklu for – *#pragma omp parallel for*, ktorý sa používa pre paralelný výpočet fitness hodnoty jedincov v jednej generácii.

Komunikácia medzi vláknami/sekciami je nutná len v prípade koevolúcie, keď populácie neinteragujú v každej generácii. Táto komunikácia je realizovaná pomocou globálnej zdieľanej premennej.

Filter je vyvinutý za daný počet generácií pred koevolučnou časťou (použitím cyklu **for**), kým v prípade detektoru je použitý cyklus **while**. Cyklus skončí, keď je nastavená globálna premenná filtrom po skončení **for** cyklu.

5.5 Ďalšie moduly programu

Pre trénovanie filtrov a detektorov potrebujeme originálny aj poškodený obrázok. Pre poškodenie obrázkov bola implementovaná funkcia, ktorá poškodí obrázok šumom typu *salt and pepper* - nastaví zadané percento náhodných pixelov na hodnoty 0 alebo 255 náhodne.

Ďalšou implementovanou funkciou je funkcia pre filtrovanie filtrom typu medián. Tento filter bol implementovaný pre porovnávanie s filtrami vytvorenými pomocou evolučných algoritmov.

5.6 Vstupy a výstupy programu

Program má niekoľko vstupov a výstupov. V prípade trénovania filtra/detektoru vstupom programu sú originálny a poškodený obrázok, v prípade, že pokračujeme výpočtom, je potrebné ešte uložené chromozómy načítať. Pri testovaní je tiež potrebný zadať originálny a poškodený obrázok, a v tomto prípade sú nutné i chromozómy.

Výstupy v prípade trénovania sú opravený obrázok a najlepší chromozóm (všetkých populácií), v prípade koevolúcie ešte chromozómy z archívu. Na štandardný výstup je ešte vypísaný fitness najlepšieho jedinca, fitness jedinca z archívu (v prípade koevolúcie) a čas behu v sekundách. Keď je logovanie zapnuté, v zadaných intervaloch je vypísaný fitness najlepšieho jedinca.

Kapitola 6

Testovanie parametrov algoritmu

Testy a experimenty boli prevedené na počítači s procesorom Intel Core i7-3632QM a operačným systémom Linux (Ubuntu). Procesor má štyri jadrá a umožňuje súčasný beh ôsmych vláken, čo je postačujúce na paralelný beh programu.

6.1 Testovacie skripty

Pretože v evolučných algoritmoch pracujeme s náhodnými číslami, kvalita výsledku sa väčšinou líši pre každý beh. Pre zistenie kvality vybraných nastavení a použitých metód je preto potrebné spustiť program viackrát s rovnakými nastaveniami, a z výstupov týchto behov vypočítať štatistiky.

Aby nebolo treba ručne spustiť program viackrát, boli vytvorené jednoduché testovacie skripty, ktoré spustia program viackrát s vhodnými nastaveniami.

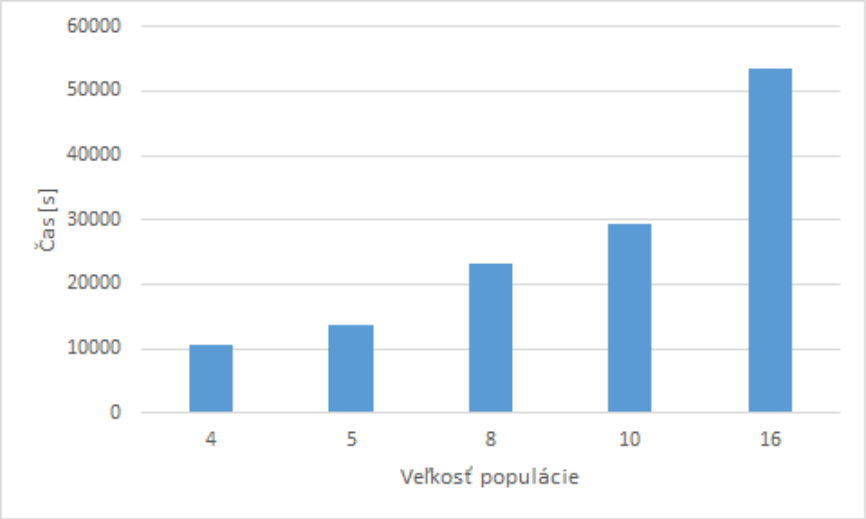
6.2 Možné parametre pre nastavenie

Priebeh evolúcie a kvalita výsledkov sú závislé na parametroch evolučného algoritmu, preto je dôležité, aby tieto parametre boli vybrané tak, aby čo najviac vyhovovali na riešenie daného problému. Pre zistenie čo najvhodnejších nastavení boli prevedené testy s rôznymi konfiguráciami.

- veľkosť populácie – pre zistenie najvhodnejšej veľkosti populácie boli prevedené testy s rôznym počtom jedincov – 4, 5, 8, 10, 16. Väčší počet jedincov znamená rýchlejšiu konvergenciu programu, ale výrazne spomalí výpočet. Pre výsledné experimenty bol vybraný menší počet jedincov (štyri) pre všetky populácie, aby bolo možné paralelné spracovanie, a aby bol výpočet rýchlejší – v prípade koevolúcie, keď fitness vypočítame pre každú dvojicu, časová náročnosť zvyšuje kvadraticky (počet jedincov v populácii detektorov krát počet jedincov v populácii filtrov).

  Na obrázku 6.1 je zobrazená časová náročnosť vývinu filtra (bez detektoru).

- veľkosť chromozómu – veľkosť chromozómu závisí na počte blokov (počte riadkov a stĺpcov kartézského programu), počte vstupov a funkcií jednotlivých blokov a výstupov. Jednotlivé bloky majú dve vstupy a jednu funkciu, CGP má jeden výstup, takže experimenty boli prevedené s rôznym počtom stĺpcov a riadkov. Najlepšie výsledky (najrýchlejšia konvergencia) boli získané s nastavením piatich riadkov a osem stĺpcov, čo dáva veľkosť chromozómu 121.



Obrázok 6.1: Časová náročnosť programu v závislosti na počte jedincov populácie.

- L-back – parameter L-back udáva, že z koľkých predchádzajúcich stĺpcov je možné výstupy blokov spojiť do vstupu blokov. Boli vyskúšané rôzne nastavenia, a najlepšie výsledky dosiahol maximálny L-back – bloky je možné pripojiť na všetky bloky predchádzajúcich stĺpcov.
- počet mutácií – počet mutácií v každej generácii a u každého jedinca je náhodné číslo. Počas experimentov bol pokus o zistenie najvhodnejšieho maximálneho počtu mutácií. Keď počet maximálnych mutácií bol menší ako 8 alebo väčší ako 16, konvergencia filtrov bola menšia. V tomto intervale (8-16) konvergencia filtrov bola veľmi podobná. Maximálny počet mutácií bol určený na 12.
- počet generácií – maximálny počet generácií bol určený hlavne kvôli časovej náročnosti algoritmu. V prípade filtrov maximálna fitness hodnota nebola dosiahnutá ani v jednom prípade, v prípade detektorov (bez koevolúcie) spoľahlivé detektory boli vyvinuté za niekoľko 10 000 generácií. Maximálny počet generácií bol stanovený na 100 000, hlavne kvôli časovej náročnosti programu.
- interakcia jedincov – v prípade koevolúcie, keď populácie neinteragujú v každej generácii, je treba zistiť, ako často je potrebná interakcia jedincov. Príliš častá interakcia spomaľuje výpočet, a zvyšuje uviaznutie v lokálnych extrémoch. Príliš zriedkavá interakcia môže spôsobiť práve opak, že aj keď fitness najlepších jedincov oboch populácií sa zvyšuje, ale kombinovaná fitness hodnota sa znižuje. V programe bolo vybrané, že interakcia populácií bude každých 100 generácií.
- výpočet fitness – výpočet fitness funkcie je možné pomocou MDPP alebo PSNR. Experimenty ukázali, že filtre vytvorené použitím MDPP pre výpočet fitness hodnoty boli menej kvalitné (po vypočítaní PSNR hodnoty výsledného obrázku).

V prvej verzii programu bolo možné jednotlivé parametre evolučného algoritmu nastaviť/zadať pomocou parametrov programu. Vo výslednej verzii to bolo ale zmenené, pretože zadanie všetkých tých parametrov nebolo užívateľsky prívetivé. Jednotlivé parametre sú nastaviteľné v súboru *params.h*.

6.2.1 Experimentálne nastavenie

V tabuľke 6.1 sú uvedené vybrané nastavenia evolučného algoritmu, ktoré boli použité v experimentoch:

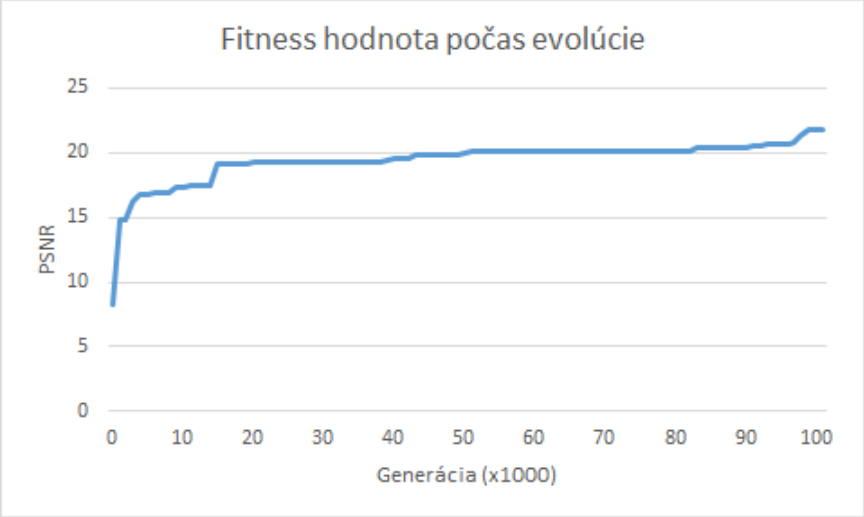
Parameter	Hodnota
Veľkosť populácie filtrov	4
Veľkosť populácie detektorov	4
Počet generácií	100 000
Maximálný počet mutácií	12
Interakcia jedincov po .. generácií	100
Počet riadkov	5
Počet stĺpcov	8
L-back	8
Počet vstupov	9

Tabuľka 6.1: Nastavenie parametrov.

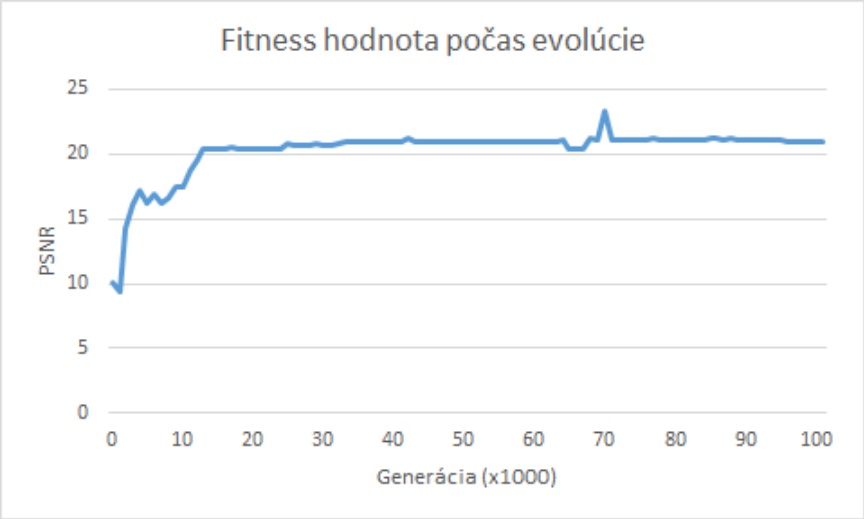
6.3 Fitness hodnota

V prípade koevolúcie, keď populácie neinteragujú v každej generácii, fitness hodnota nie je rastúca na celom intervale (obrázok 6.2b). To je dané tým, že filter i detektor boli vyvinuté samostatne, a pre ich ohodnotenie bola použitá iná fitness funkcia – ich ohodnotenie nebolo závislé na druhej populácii. Môže sa stať, že fitness klesá práve pred ukončením výpočtu, a kvalita výsledku bude nižšia. Toto je dôvod použitia archívu v tomto prípade – aj keď fitness klesla, máme uloženú zatiaľ nájdenú najlepšiu dvojicu detektor-filter.

Tento „problém" v iných prípadoch (filter, koevolúcia v každej generácii, vývin filtra a detektoru nezávisle) nenastáva, preto archív ani nie je u nich použitý (obrázok 6.2a).



(a) Bez použitia koevolúcie.



(b) Použitím koevolúcie.

Obrázok 6.2: Vývoj fitness hodnoty jedinca.

Kapitola 7

Experimentálne vyhodnotenie

V tejto kapitole sú porovnané jednotlivé implementované metódy filtrovania obrazu na základe časovej náročnosti, veľkosti riešení a kvality výsledkov. Tieto metódy sú:

- použitie len filtra,
- použitie detektoru a filtra bez koevolúcie, filter je trénovaný len na poškodených pixeloch,
- použitie detektoru a filtra bez koevolúcie, filter je trénovaný na všetkých pixeloch,
- použitie detektoru a filtra s koevolúciou, filter je trénovaný na všetkých pixeloch, interakcia populácií každých 100 generácií,
- použitie detektoru a filtra s koevolúciou, filter je trénovaný na poškodených pixeloch, interakcia populácií každých 100 generácií.

Nastavenie programu pre experimenty je uvedené v tabuľke 6.1. Pre každú metódu a pre intenzity šumu 10, 20, 30, 40 a 50% bolo prevedených 30 behov programu. Na trénovanie bol použitý obrázok lena_gray_256.tif s rozlíšením 256x256 (poškodený 10% šumom – obrázok A.1a, 20% šumom – obrázok A.8a, 30% šumom – obrázok A.15a, 40% šumom – obrázok A.22a, 50% šumom – obrázok A.29a). Na testovanie bola použitá sada obrázkov:

- obrázok použitý na trénovanie (lena_gray_256.tif), ale s iným náhodným šumom (10% šumom – obrázok A.2a, 20% šumom – obrázok A.9a, 30% šumom – obrázok A.16a, 40% šumom – obrázok A.23a, 50% šumom – obrázok A.30a),
- obrázok lena_gray_512.tif s rozlíšením 512x512 (poškodený 10% šumom – obrázok A.3a, 20% šumom – obrázok A.10a, 30% šumom – obrázok A.17a, 40% šumom – obrázok A.24a, 50% šumom – obrázok A.31a),
- obrázok livingroom.tif s rozlíšením 512x512 (poškodený 10% šumom – obrázok A.4a, 20% šumom – obrázok A.11a, 30% šumom – obrázok A.18a, 40% šumom – obrázok A.25a, 50% šumom – obrázok A.32a),
- obrázok mandril_gray.tif s rozlíšením 512x512 (poškodený 10% šumom – obrázok A.5a, 20% šumom – obrázok A.12a, 30% šumom – obrázok A.19a, 40% šumom – obrázok A.26a, 50% šumom – obrázok A.33a),

- obrázok woman_blonde.tif s rozlíšením 512x512 (poškodený 10% šumom – obrázok A.6a, 20% šumom – obrázok A.13a, 30% šumom – obrázok A.20a, 40% šumom – obrázok A.27a, 50% šumom – obrázok A.34a),
- obrázok woman_darkhair.tif s rozlíšením 512x512 (poškodený 10% šumom – obrázok A.7a, 20% šumom – obrázok A.14a, 30% šumom – obrázok A.21a, 40% šumom – obrázok A.28a, 50% šumom – obrázok A.35a).

7.1 Časová náročnosť

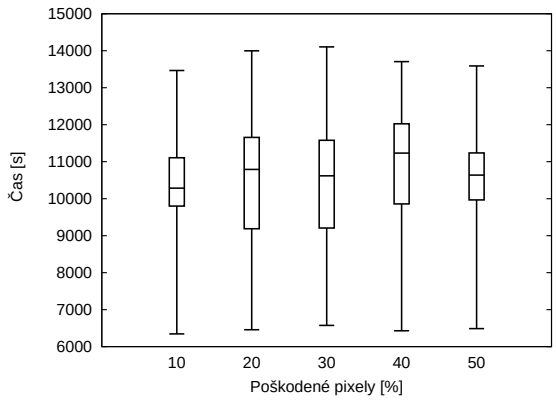
Aj keď kvalitný filter je treba vytvoriť len raz, časová náročnosť vývinu je dôležitá, pretože nie každý vyevolvovaný filter je kvalitný. Je treba viackrát opakovať evolúciu, kým nedostaneme kvalitný filter. Ďalším problémom je, že vyevolvovaný filter funguje spoľahlivo len na typy šumu (na typoch obrázkov, atď.), na ktorý bol trénovaný. Pre rôzne problémy je treba vytvoriť iné filtre, väčšinou s inými nastaveniami evolúcie.

Prvým cieľom vtedy je nájsť čím rýchlejšiu metódu vývoja, aby sa ostatné vlastnosti výsledných filtrov (hlavne kvalita) nezhoršili.

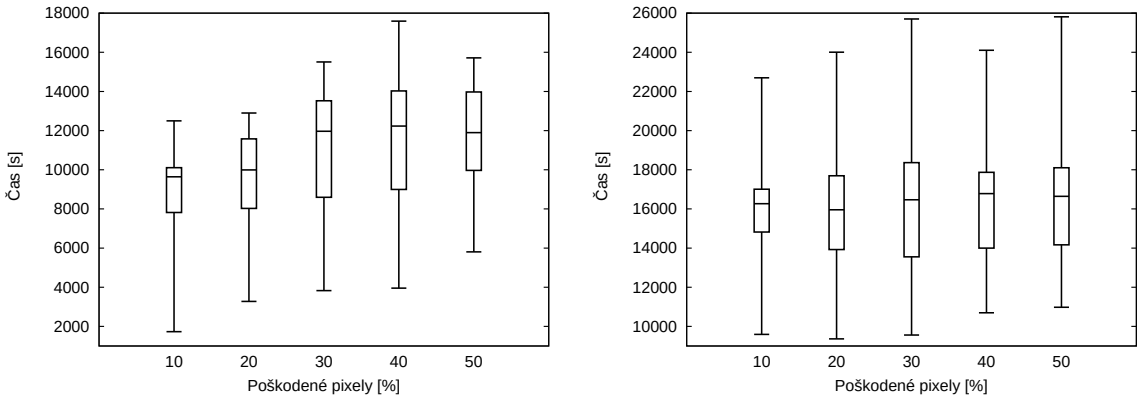
Časovú náročnosť vývinu filtrov ovplyvňuje niekoľko parametrov, hlavne:

- počet jedincov populácie,
- počet použitých blokov,
- počet pixelov použitých pre výpočet fitness hodnoty,
- počet generácií.

Na obrázkoch 7.1, 7.2a, 7.2b, 7.3a a 7.3b, sú uvedené jednotlivé časy merané počas experimentov. Pri meraní času bola použitá sekvenčná verzia programu okrem koevolúcie (v prípade, že jedinci neinteragovali v každej generácii), kde boli použité dve vlákna, jeden pre detektor a jeden pre filter. V tomto prípade bol použitý procesorový čas (čas behu programu x 2).

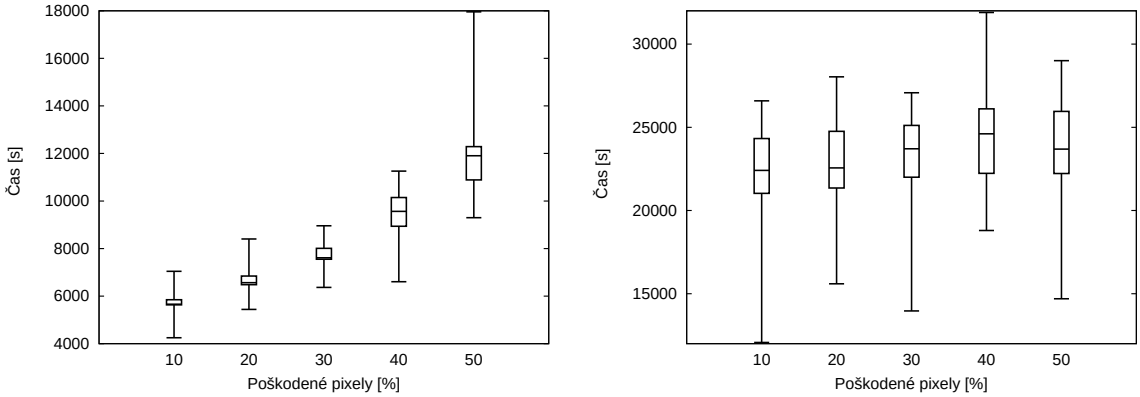


Obrázok 7.1: Časová náročnosť vývinu filtra.



(a) Filtre trénované len na poškodených pixeloch. (b) Filtre trénované na všetkých pixeloch.

Obrázok 7.2: Časová náročnosť vývinu filtra a detektoru bez koevolúcie.



(a) Filtre trénované len na poškodených pixeloch. (b) Filtre trénované na všetkých pixeloch.

Obrázok 7.3: Časová náročnosť vývinu filtra a detektoru pomocou koevolúcie.

Ako vidíme z obrázkov, keď filter bol trénovaný na všetkých pixeloch, časová náročnosť vývoja filtrov je podobná pre rôzne intenzity šumu (obrázky 7.1, 7.2b a 7.3b).

Použitím len poškodených pixelov (obrázky 7.2a a 7.3a) je možné časovú náročnosť výpočtu výrazne znížiť. Toto zrýchlenie sa dá najlepšie vidieť v prípade koevolúcie (obrázky 7.3b a 7.3a), kde zrýchlenie je až štvornásobné v prípade 10 percentného šumu. Aj keby sme mohli očakávať v tomto prípade skoro desaťnásobné zrýchlenie, to nie je možné dosiahnuť, pretože koevolučná časť, ktorá je časovo veľmi náročná, sa nezrýchľuje.

V prípade filtra a detektoru bez koevolúcie (obrázky 7.2b a 7.2a) zrýchlenie nie je tak výrazné, pretože detektor je takisto trénovaný na všetkých pixeloch, vtedy zrýchľuje len výpočet fitness hodnoty filtra.

Zo všetkých implementovaných metód najrýchlejší bol vývoj pomocou koevolúcie (interakcia jedincov každých 100 generácií) (obrázok 7.3a), keď filtre boli trénované len na poškodených pixeloch. Pomalšie boli metódy, kedy sme pre výpočet fitness hodnoty filtra

používali všetky pixely, najpomalšou z nich bola metóda používajúca koevolúciu (obrázok 7.3b).

7.2 Veľkosť riešenia

Veľkosť riešenia v našom prípade znamená počet použitých blokov. V prípade kartézskeho genetického programovania sa veľmi často využíva len časť blokov.

Počet použitých blokov má vplyv na rýchlosť vývinu filtrov a detektorov (pri nepoužitých blokoch nie je treba vypočítať ich hodnotu), ale aj na rýchlosť výsledného filtra/detektoru – čím viac operácií je treba na detekciu/filtrovanie pixelu, tým dlhšie to bude trvať.

Ďalším dôvodom pre minimalizáciu počtu blokov je, keď chceme daný filter implementovať na hardware, kde máme limitovanú veľkosť čipu.

V nasledujúcich tabuľkách sú uvedené minimálne, priemerné a maximálne veľkosti vyevolvovaných filtrov.

Poškodené pixely [%]	10	20	30	40	50
Minimum	0	6	5	8	8
Priemer	12,93333	14,23333	15,125	15,06452	15,67742
Maximum	20	19	23	22	20

Tabuľka 7.1: Počet použitých blokov filtra (vývin len filtra).

Poškodené pixely [%]	10	20	30	40	50
Minimum	7	8	9	5	8
Priemer	13,86667	13,93333	13,92308	13,27273	13,90909
Maximum	19	21	19	20	19

Tabuľka 7.2: Počet použitých blokov filtra (vývin filtra a detektoru bez koevolúcie).

Poškodené pixely [%]	10	20	30	40	50
Minimum	2	2	3	2	2
Priemer	5,133333	5,5	6,333333	7,076923	6,307692
Maximum	9	10	9	11	10

Tabuľka 7.3: Počet použitých blokov detektoru (vývin filtra a detektoru bez koevolúcie).

Poškodené pixely [%]	10	20	30	40	50
Minimum	8	8	6	6	8
Priemer	14,36667	14,333333	14,43333	14,5	14,46667
Maximum	22	21	21	23	22

Tabuľka 7.4: Počet použitých blokov filtra (vývin filtra a detektoru s koevolúciou).

Poškodené pixely [%]	10	20	30	40	50
Minimum	0	1	1	2	3
Priemer	4,6	5,4	5,8	6,2	6,53333
Maximum	10	10	11	11	10

Tabuľka 7.5: Počet použitých blokov detektoru (vývin filtra a detektoru s koevolúciou).

Experimenty ukázali, že v prípade detektorov veľkosť riešenia závisí na intenzite šumu (tabuľky 7.5 a 7.3). Vyššia intenzita šumu vyžaduje komplexnejšie detektory. Medzi veľkosťami detektorov vyevolvovaných pomocou kovolúcie a bez koevolúcie sú menšie rozdiely – detektory vytvorené použitím koevolúcie používajú v priemere menší počet blokov.

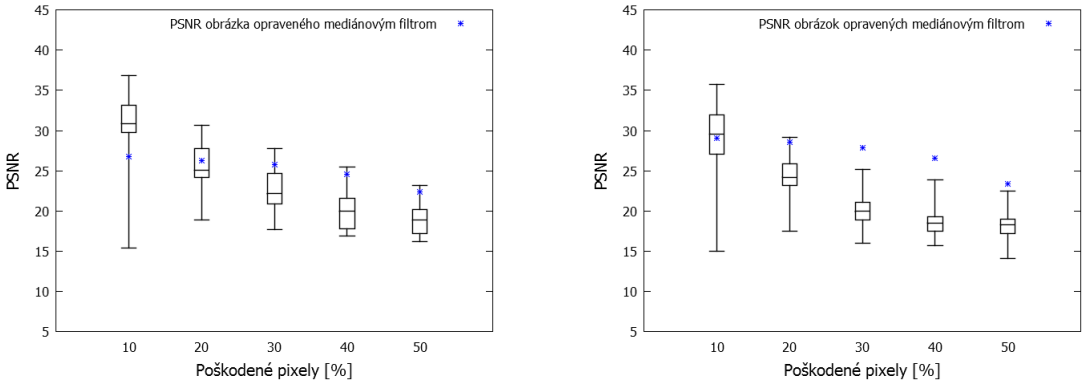
V prípade filtrov najviac blokov používali filtre trénované na každom pixeli, kde je možné tiež vidieť rastúci počet blokov potrebných pre filtrovanie obrazov s vyššiu intenzitou šumu (tabuľka 7.1).

V prípade filtrov trénovaných len na poškodených pixeloch počet použitých blokov neukazuje rastúcu sa tendenciu so zvýšením intenzity šumu (tabuľky 7.4 a 7.4).

7.3 Kvalita riešenia

Najdôležitejšou vlastnosťou filtra je, že ako dobre opraví poškodený obrázok. V tejto podkapitole bude porovnaná efektívnosť jednotlivých metód.

Počas evolúcie pre výpočet fitness hodnoty jedinca boli použité MDPP a PSNR, ale pre porovnávanie bude použitá hodnota PSNR, aby výsledky boli lepšie porovnateľné. Pretože použijeme PSNR pre výpočet fitness hodnoty, väčšia hodnota znamená kvalitnejšieho jedinca. Keď počas vývinu bolo použité MDPP, hodnota PSNR bola vypočítaná z výsledného a originálneho obrázku.



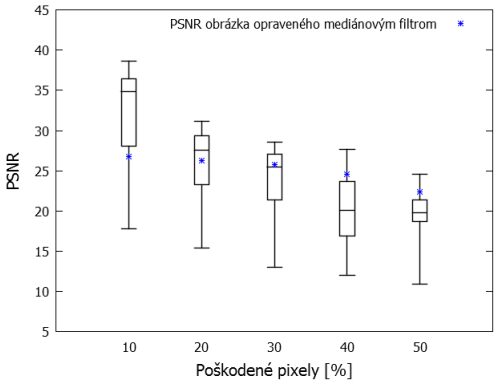
(a) Výsledky trénovania.

(b) Výsledky testov

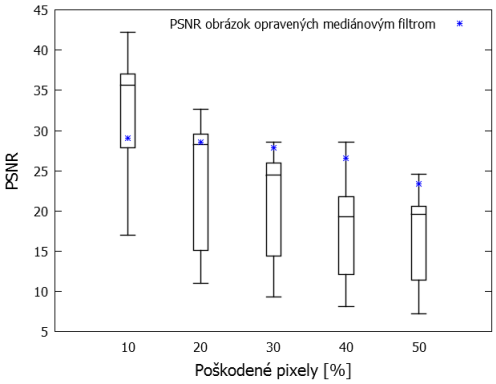
Obrázok 7.4: Štatistiky filtra bez detektoru.

Experimenty ukázali, že najhoršie výsledky z implementovaných metód dávali filtre bez detektoru. Aj keď v prípade trénovacieho obrázku boli nájdené filtre, ktoré boli kvalitnejšie, ako mediánový filter (obrázok 7.4a), na testovacích obrázkoch vyevolvované filtre boli menej kvalitné (obrázok 7.4b).

Použitie detektorov zlepšuje kvalitu filtrovaného obrázku. Na trénovacom obrázku najkvalitnejší filter/detektor bol nájdený bez použitia koevolúcie, a v prípade, keď filtre boli

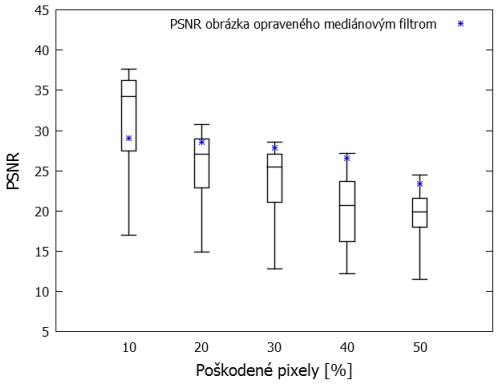


(a) Výsledky trénovania.

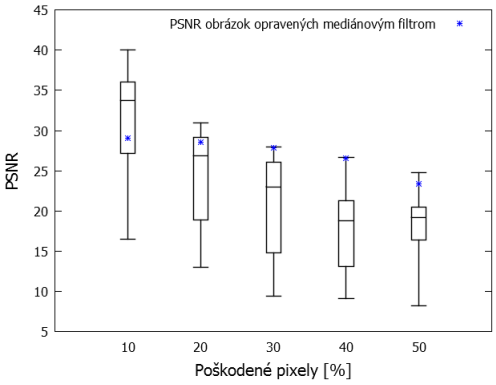


(b) Výsledky testov

Obrázok 7.5: Štatistiky filtra a detektora bez koevolúcie, filtre trénované na poškodených pixeloch.



(a) Výsledky trénovania.



(b) Výsledky testov

Obrázok 7.6: Štatistiky filtra a detektora bez koevolúcie, filtre trénované na všetkých pixeloch.

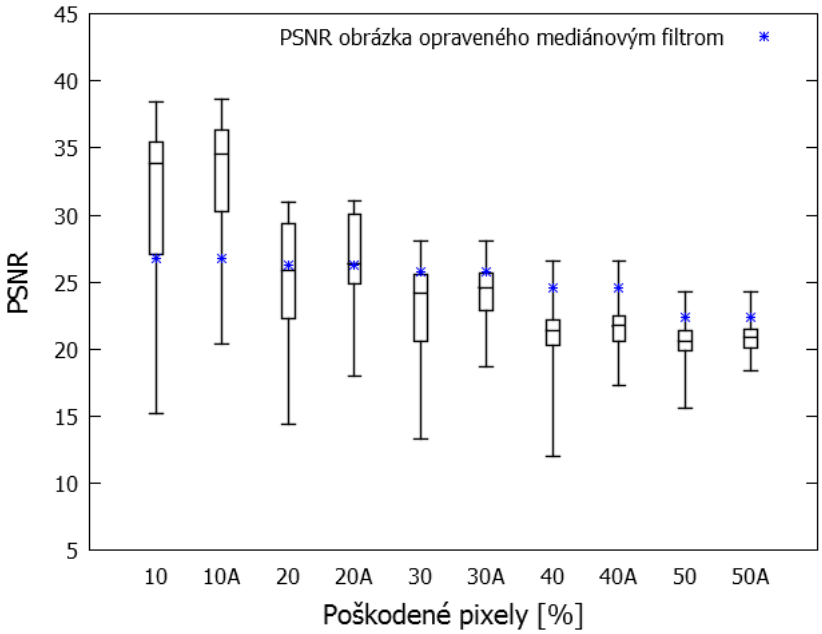
trénované len na poškodených pixeloch (obrázok 7.5a). Je to dané tým, že na trénovacom obrázku vyevolvované detektory väčšinou mali úspešnosť v detekovaní poškodených pixelov až 100%, kým v prípade koevolúcie pomer takých detektorov bol oveľa menší.

Filtre, ktoré boli trénované na všetkých pixeloch mali tiež nižšiu kvalitu na trénovacom obrázku, tiež kvôli kvalitným detektorom (obrázky 7.6a a 7.8a).

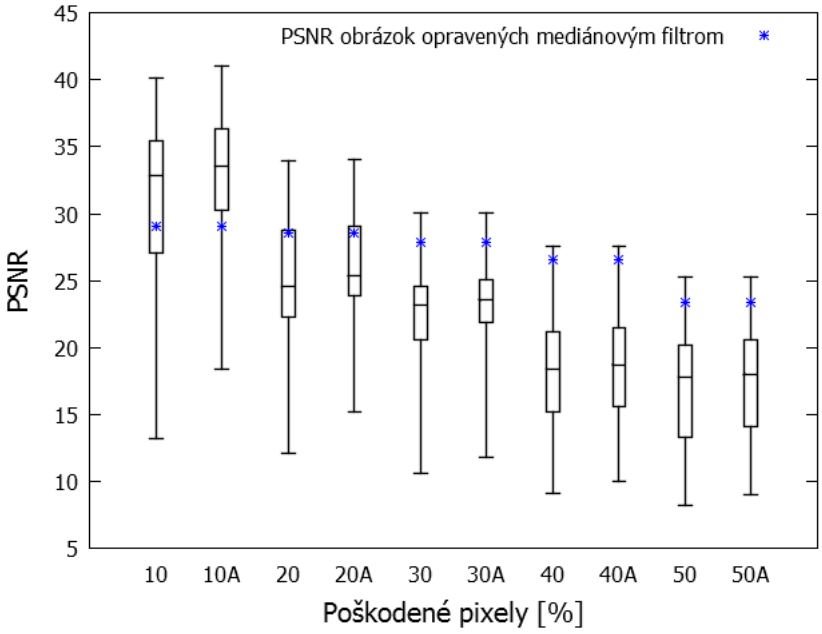
Aj keď najlepšie filtre/detektory boli vytvorené bez koevolúcie, v prípade použitia koevolúcie (obrázok 7.7a) rozptyl kvalít filtrov/detektorov je oveľa menší (hlavne v prípade veľkých počtov poškodených pixelov a použitím archívu).

V prípade testov filtre väčšinou dávali horšie výsledky, ako na trénovacom obrázku, ale nie v každom prípade. Na obrázkoch, kde detektory fungovali spoľahlivo, filtre trénované len na poškodených pixeloch dávali lepšie výsledky (približne polovica najlepších výsledkov pre rôzne obrazy a rôzne intenzity šumu boli dosiahnuté pomocou koevolúcie (obrázok 7.7b), druhá polovica s použitím filtra a detektoru bez koevolúcie (obrázok 7.5b)).

V prípade filtrov trénovaných na všetkých pixeloch, filtre a detektory, ktoré boli trénované použitím koevolúcie (obrázok 7.8b), dávali lepšie výsledky na obrázkoch s intenzitou šumu aspoň 20%, ako filtre a detektory trénované bez použitia koevolúcie (obrázok 7.6b).



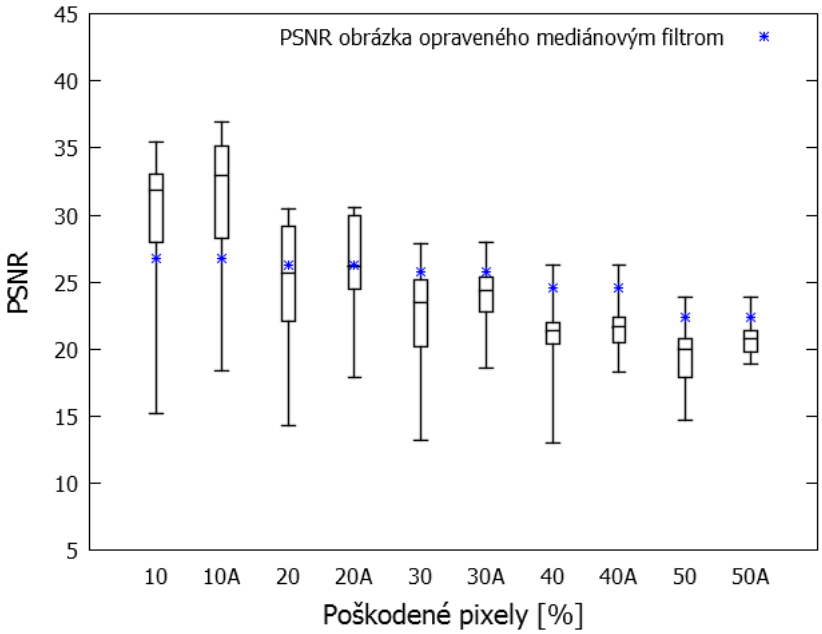
(a) Výsledky trénovania.



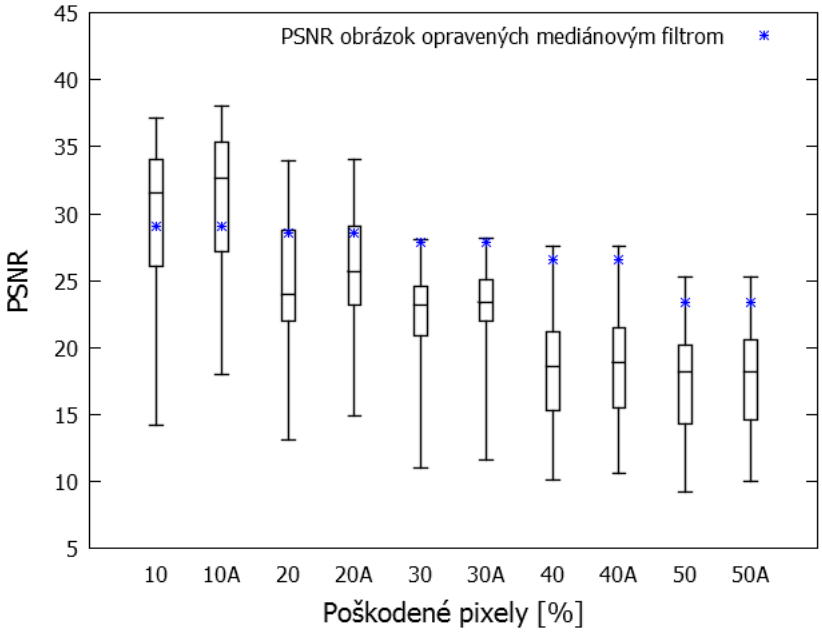
(b) Výsledky testov

Obrázok 7.7: Štatistiky filtra a detektora s koevolúciou, filtre trénované na poškodených pixeloch. Písmeno „A" u poškodených pixelov značí kvalitu archivovaných filtrov a detektorov.

Výsledky trénovania sú zhrnuté v tabuľke 7.6a a výsledky testov v tabuľke 7.6b.



(a) Výsledky trénovania.



(b) Výsledky testov

Obrázok 7.8: Štatistiky filtra a detektora s koevolúciou, filtre trénované na všetkých pixeloch. Písmeno „A" u poškodených pixelov značí kvalitu archivovaných filtrov a detektorov.

7.4 Diskusia o výsledkoch

V predchádzajúcej časti boli jednotlivé implementované metódy porovnávané s ohľadom na časovú náročnosť vývinu filtra a detektoru a na veľkosti a kvalite riešenia.

Z experimentov sa ukázalo, že keď máme k dispozícii kvalitný detektor, nevyplatí sa trénovať filtre na všetkých pixeloch – časová náročnosť vývinu je oveľa väčšia – až štvorná-

sobná v prípade malej intenzity šumu, veľkosť filtra je väčšia, a aj kvalita filtra je nižšia.

Filtre a detektory vytvorené pomocou koevolúcie mali trošku horšiu kvalitu na trénovacom obrázku ako filtre a detektory vytvorené bez koevolúcie, napriek tomu na testovacích obrázkoch boli približne rovnako kvalitné. Filtre a detektory vyvinuté pomocou koevolúcie dali lepšie výsledky na obrázkoch, kde detektor nefungoval spoľahlivo. Časová náročnosť návrhu pomocou koevolúcie bola menšia, ale veľkosť týchto riešení je trochu väčšia ako veľkosť filtrov a detektorov vytvorených bez koevolúcie.

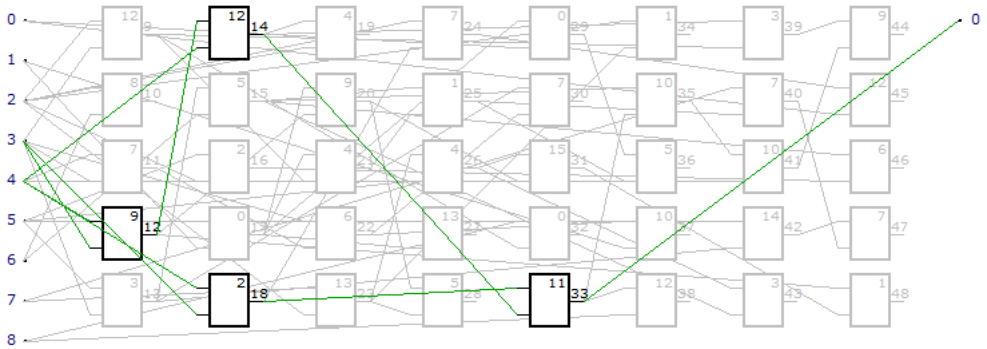
Použiť koevolúciu sa oplatí hlavne vtedy, keď nie je možné vytvoriť kvalitné detektory.

7.4.1 Príklad evolvovaného filtra

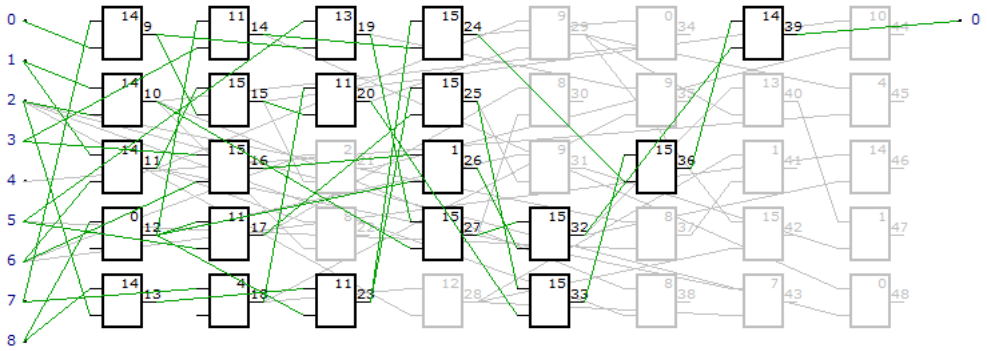
Na obrázkoch 7.9b a 7.9a je uvedený najlepší nájdený filter a detektor pre šum typu *salt and pepper* s intenzitou 30%. Výsledok filtrovania pomocou tohto filtra (obrázok 7.11d) je porovnaný s mediánovým filtrom (3x3 – obrázok 7.11c a 5x5 – obrázok 7.11b). Ako vidíme na obrázku, použitie väčšieho okolia pixelu pre filtrovanie spôsobuje rozmazanie obrazu, použitie menšieho okolia pixelu nie je postačujúce pre potlačenie šumu.

Evolvovaný filter nerozmaže obrázok, používa len menšie okolie pixelu (3x3) a potlačuje šum oveľa lepšie, ako mediánový filter s rovnakou veľkosťou okna.

Pretože kartézska mriežka filtra obsahuje neaktívne uzly, a veľkosť detektoru je malá vo väčšine prípadov, je možné detektor a filter spojiť do jednej kartézskej mriežky, ktorá bude mať dva výstupy. Spojenie detektoru a filtra má výhody hlavne v prípade hardwarovej implementácie, kde veľkosť čipu môže byť limitovaná. Príkladom spojenia detektora a filtra (z obrázkov 7.9a a 7.9b) je ukázaná na obrázku 7.10.

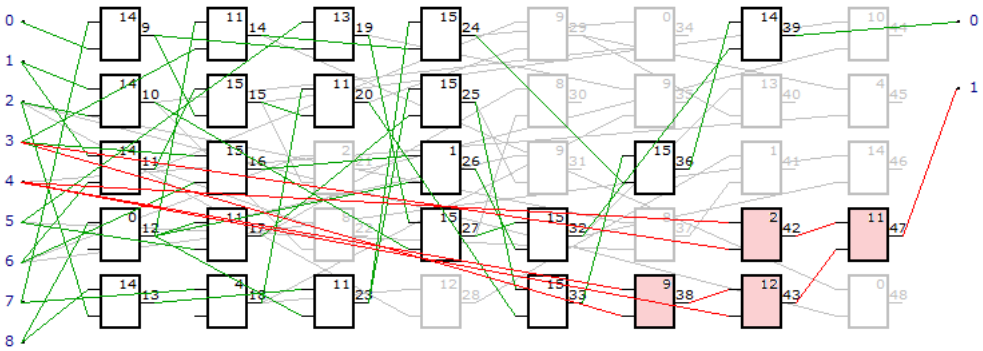


(a) Zapojenie detektoru.



(b) Zapojenie filtra.

Obrázok 7.9: Najlepší nájdený detektor a filter pre intenzitu šumu 30%.



Obrázok 7.10: Najlepší nájdený detektor a filter pre intenzitu šumu 30% po spojení. Bloky detektoru sú označené ružovou farbou.

Šum		Typ filtra						
		1	2	3	4	5	6	7
10%	Minimum	15.4	17.8	17.0	15.2	**20.4**	15.2	18.4
	Priemer	31.2	33.0	32.8	32.9	**33.4**	31.4	32.1
	Maximum	36.9	**38.7**	37.6	38.5	38.6	35.5	37.0
20%	Minimum	**18.9**	15.4	14.9	14.4	18.0	14.3	17.9
	Priemer	25.4	26.2	25.9	25.7	**26.4**	25.6	26.2
	Maximum	30.7	**31.2**	30.8	31.0	31.1	30.5	30.6
30%	Minimum	17.7	13.1	12.9	13.3	**18.7**	13.2	18.6
	Priemer	22.3	**24.9**	24.7	24.7	**24.9**	23.7	24.3
	Maximum	27.8	**28.6**	28.5	28.1	28.1	27.9	28.0
40%	Minimum	16.9	12.0	12.2	12.0	17.3	13.0	**18.3**
	Priemer	20.0	20.1	20.6	21.0	21.4	21.1	**21.7**
	Maximum	25.5	**27.7**	27.2	26.6	26.6	26.3	26.3
50%	Minimum	16.2	11.0	11.5	15.6	18.4	14.7	**18.9**
	Priemer	19.1	19.7	19.7	20.5	**20.7**	19.9	**20.7**
	Maximum	23.1	**24.6**	24.5	24.3	24.3	23.9	23.9

(a) Na trénovacom obrázku.

Šum		Typ filtra						
		1	2	3	4	5	6	7
10%	Minimum	15.0	17.0	16.5	13.2	**18.4**	14.2	18.0
	Priemer	29.3	32.6	32.1	32.4	**33.1**	29.9	30.2
	Maximum	35.8	**42.2**	40.0	40.1	41.0	37.1	38.0
20%	Minimum	**17.5**	11.0	13.0	12.1	15.2	13.1	14.9
	Priemer	24.3	25.1	24.7	24.6	**25.4**	24.5	25.0
	Maximum	29.1	32.6	30.9	34.0	**34.2**	34.0	34.1
30%	Minimum	**16.0**	9.3	9.5	10.6	11.8	11.0	11.6
	Priemer	20.1	21.6	21.1	22.5	22.9	22.8	**23.0**
	Maximum	25.2	28.5	28.0	**30.1**	**30.1**	28.1	28.2
40%	Minimum	**15.7**	8.1	9.1	9.1	10.0	10.1	10.6
	Priemer	18.5	17.3	17.2	18.4	**18.7**	18.4	18.6
	Maximum	23.9	**28.6**	26.6	27.6	27.6	27.5	27.7
50%	Minimum	**14.1**	7.3	8.3	8.2	9.0	9.2	10.0
	Priemer	17.9	16.7	17.3	17.8	**18.1**	17.9	18.0
	Maximum	22.5	24.6	24.7	25.3	25.3	25.3	**25.4**

(b) Na testovacích obrázkoch.

Tabuľka 7.6: Minimálne, priemerné a maximálne fitness hodnoty vytvorených filtrov. Číslo u typy filtra značí: 1: Filtre (bez detektoru). 2: Filtre a detektory vytvorené bez koevolúcie, filtre trénované len na poškodených pixeloch. 3: Filter a detektory vytvorené bez koevolúcie, filtre trénované na všetkých pixeloch. 4: Filter a detektory vytvorené použitím koevolúcie, filtre trénované len na poškodených pixeloch. 5: Archivované filtre a detektory vytvorené použitím koevolúcie, filtre trénované len na poškodených pixeloch. 6: Filter a detektory vytvorené použitím koevolúcie, filtre trénované na všetkých pixeloch. 7: Archivované filtre a detektory vytvorené použitím koevolúcie, filtre trénované na všetkých pixeloch.

(a) Poškodený obrázok. (b) Medián (5x5).

(c) Medián (3x3). (d) Evolvovaný filter.

Obrázok 7.11: Filtrovania obrázku pomocou filtra.

Kapitola 8

Záver

Cieľom tejto diplomovej práce bolo preštudovať genetické programovanie, koevolučné algoritmy, evolučný návrh obrazových filtrov, navrhnúť a implementovať program na tvorbu obrazových filtrov kombinovaných s detektormi šumu pomocou genetického programovania a s využitím kooperatívnej koevolúcie, overiť funkčnosť programu a zhodnotiť dosiahnuté výsledky.

Navrhnutý a implementovaný program je schopný vytvoriť obrazové filtre, ktoré môžu byť vyvinuté samostatne, s detektorom alebo s detektorom pomocou koevolúcie. V rámci práce filtre boli vytvorené pre odstránenie šumu typu *salt and pepper* s rôznymi intenzitami. Jednotlivé vytvorené filtre boli porovnané na základe časovej náročnosti vývinu, veľkosti riešenia a kvalitu filtrov.

Použitie detektoru zlepšuje kvalitu výsledkov, a môže i znížiť časovú náročnosť vývoja – keď sú použité i detektory, filtre stačí trénovať len na poškodených pixeloch. Najmenšiu časovú náročnosť vývoja mali takto trénované filtre a detektory vytvorené pomocou koevolúcie (až do 40% šumu).

Okrem menšej časovej náročnosti vývinu filtra a detektoru, ďalšou výhodou filtrov vytvorených pomocou koevolúcie je menší rozptyl kvality – pomocou koevolúcie nevzniká veľký počet nekvalitných filtrov, ako bez použitia koevolúcie.

Aj keď na trénovacom obrázku najkvalitnejšie filtre boli tie, ktoré boli vytvorené bez koevolúcie, na testovacích obrázkoch taký rozdiel v kvalite sa neprejavil – filtre a detektory vytvorené pomocou koevolúcie boli až tak kvalitné, ako filtre a detektory vytvorené bez koevolúcie.

Podľa experimentov na vytvorenia filtrov a detektorov použitie koevolúcie sa oplatí v prípade, keď nie je možné vytvoriť spoľahlivé detektory, alebo v prípade, keď testovacie obrázky majú iný charakter, ako trénovací obrázok.

V pokračovaní projektu by bolo možné program optimalizovať vzhľadom k časovej náročnosti, prípadne vzhľadom k veľkosti riešenia. Tiež by bolo vhodné vyskúšať vytvorenie filtrov na potlačení iných typov šumu.

Vypracovanie diplomovej práce bolo pre mňa veľmi prínosné. Zopakoval som si evolučné algoritmy a obrazové filtre, bližšie som spoznal kartézske genetické programovanie, a zoznámil som sa s koevolučnými algoritmami. Tiež som získal praktické skúsenosti s paralelným programovaním použitím knižnice OpenMP.

Literatúra

[1] Brownrigg, D. R. K.: The Weighted Median Filter. *Commun. ACM*, ročník 27, č. 8, 1984: s. 807–818, ISSN 0001-0782.

[2] Eiben, A. E.; Smith, J. E.: *Introduction to Evolutionary Computing*. SpringerVerlag, 2003, 299 s., ISBN 3540401849.

[3] Freitas, A.: Basic Concepts of Evolutionary Algorithms. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*, Natural Computing Series, Springer Berlin Heidelberg, 2002, s. 79–106, ISBN 978-3-642-07763-0.

[4] Gonzalez, R. C.; Woods, R. E.: *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006, 954 s., ISBN 013168728X

[5] Harding, S.; Banzhaf, W.: Fast Genetic Programming on GPUs. *Genetic Programming*, *Lecture Notes in Computer Science*, ročník 4445, Springer Berlin Heidelberg, 2007, s. 90–101, ISSN 0302-9743.

[6] Husbands, P.; Mill, F.: Simulated Co-Evolution as the Mechanism for Emergent Planning and Scheduling. *Proc. of the Fourth International Conference on Genetic Algorithms*, San Diego, CA, 1991, s. 264–270, ISBN 1-55860-208-9.

[7] Hwang, H.; Haddad, R.: Adaptive median filters: new algorithms and results. *IEEE Transactions on Image Processing*, ročník 4, č. 4, 1995: s. 499–502, ISSN 1057-7149.

[8] Jain, R.; Kasturi, R.; Schunck, B. G.: *Machine Vision*. McGraw-Hill, 1995, ISBN 0-07-032018-7.

[9] Janzen, D. H.: When is it co-evolution? *Evolution*, ročník 34, č. 3, 1980: s. 611–612, ISSN 0014-3820.

[10] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992, 609 s., ISBN 0-262-11170-5.

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

[12] Poli, R.; Langdon, W. B.; McPhee, N. F.: *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008, 252 s., ISBN 978-1-4092-0073-4.

[13] Popovici, E.; Bucci, A.; Wiegand, R.; aj.: Coevolutionary Principles. *Handbook of Natural Computing*, Springer Berlin Heidelberg, 2012, s. 987–1033, ISBN 978-3-540-92909-3.

[14] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; aj.: *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992, ISBN 0-521-43108-5.

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

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

[17] Snustad, D.; Simmons, M.; Matalová, A.: *Genetika*. Masarykova univerzita, 2009, ISBN 9788021048522.

[18] Šikulová, M.; Sekanina, L.: Acceleration of Evolutionary Image Filter Design Using Coevolution in Cartesian GP. Springer, 2012, s. 163–172, ISSN 0302-9743.

[19] Šikulová, M.; Sekanina, L.: Coevolution in Cartesian Genetic Programming. *Proceedings of the 15th European Conference on Genetic Programming*, EuroGP'12, Berlin, Heidelberg: Springer-Verlag, 2012, s. 182–193, ISBN 978-3-642-29138-8.

[20] Tomassini, M.: *Spatially structured evolutionary algorithms: artificial evolution in space and time*. Dordrecht: Springer, 2005, ISBN 3540241930.

[21] Vašíček, Z.; Bidlo, M.; Sekanina, L.; aj.: Evolutionary design of efficient and robust switching image filters. *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2011, s. 192–199, ISBN 978-1-4577-0598-4.

[22] Vašíček, Z.; Sekanina, L.: Hardware Accelerators for Cartesian Genetic Programming. *Genetic Programming*, *Lecture Notes in Computer Science*, ročník 4971, Springer Berlin Heidelberg, 2008, s. 230–241, ISSN 0302-9743.

Príloha A

Filtrom opravené obrázky

V nasledujúcej časti sú uvedené trénovacie a testovacie obrázky opravené pomocou vytvorených filtrov.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.1: Trénovací obrázok pre šum s intenzitou 10%.

(a) Poškodený obrázok.

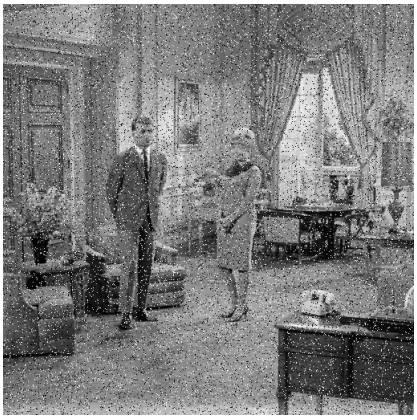
(b) Filtrovaný obrázok.

Obrázok A.2: Testovací obrázok lena_gray_256.tif pre šum s intenzitou 10%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

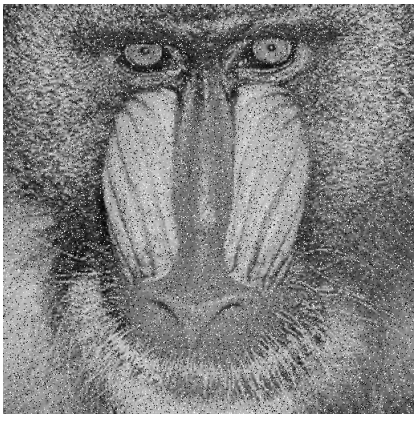
Obrázok A.3: Testovací obrázok lena_gray_512.tif pre šum s intenzitou 10%.



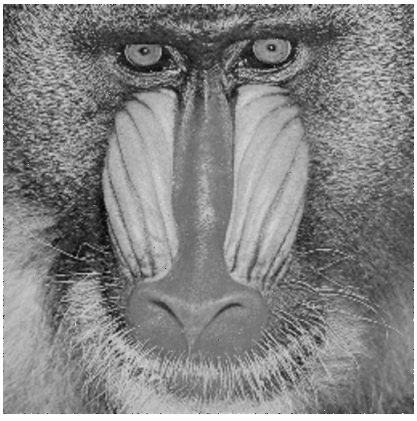
(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.4: Testovací obrázok livingroom.tif pre šum s intenzitou 10%.



(a) Poškodený obrázok.



(b) Filtrovaný obrázok.

Obrázok A.5: Testovací obrázok mandril.tif pre šum s intenzitou 10%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.6: Testovací obrázok woman_blonde.tif pre šum s intenzitou 10%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.7: Testovací obrázok woman_darkhair.tif pre šum s intenzitou 10%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

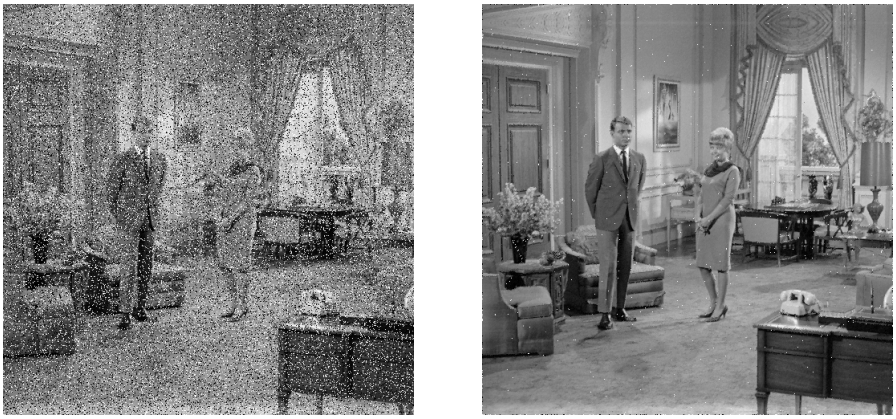
Obrázok A.8: Trénovací obrázok pre šum s intenzitou 20%.

(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.9: Testovací obrázok lena_gray_256.tif pre šum s intenzitou 20%.

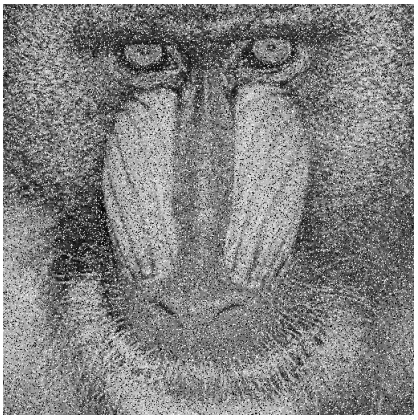
(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.10: Testovací obrázok lena_gray_512.tif pre šum s intenzitou 20%.

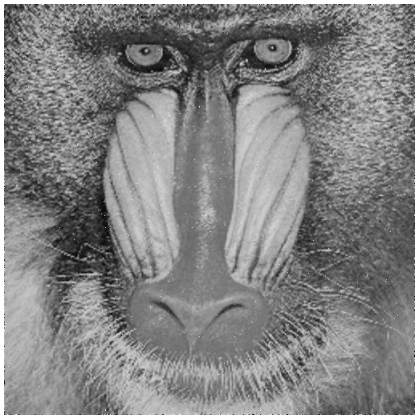


(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.11: Testovací obrázok livingroom.tif pre šum s intenzitou 20%.



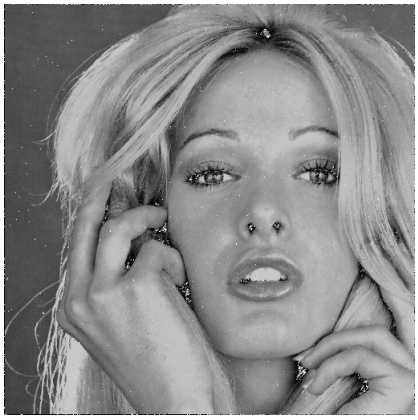
(a) Poškodený obrázok.



(b) Filtrovaný obrázok.

Obrázok A.12: Testovací obrázok mandril.tif pre šum s intenzitou 20%.

(a) Poškodený obrázok.



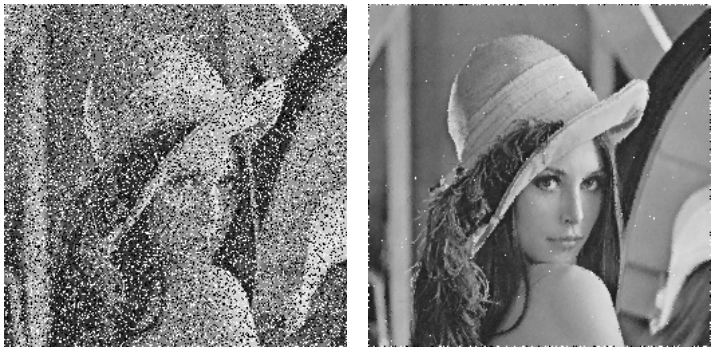
(b) Filtrovaný obrázok.

Obrázok A.13: Testovací obrázok woman_blonde.tif pre šum s intenzitou 20%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.14: Testovací obrázok woman_darkhair.tif pre šum s intenzitou 20%.

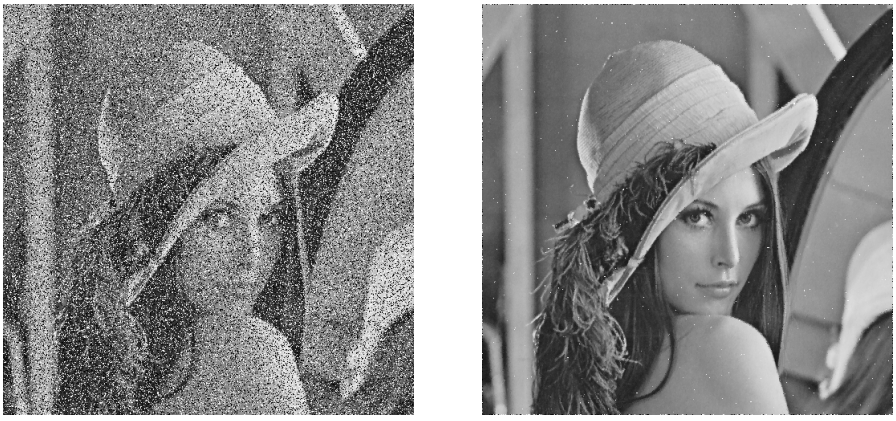


(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.15: Trénovací obrázok pre šum s intenzitou 30%.

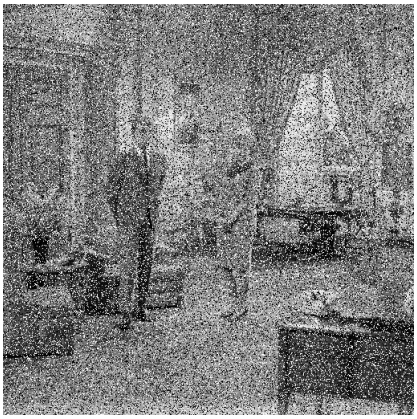
(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.16: Testovací obrázok lena_gray_256.tif pre šum s intenzitou 30%.



(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.17: Testovací obrázok lena_gray_512.tif pre šum s intenzitou 30%.

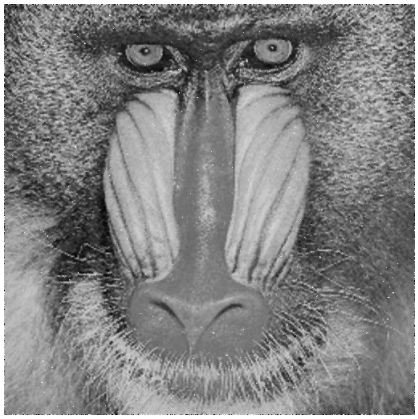


(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.18: Testovací obrázok livingroom.tif pre šum s intenzitou 30%.

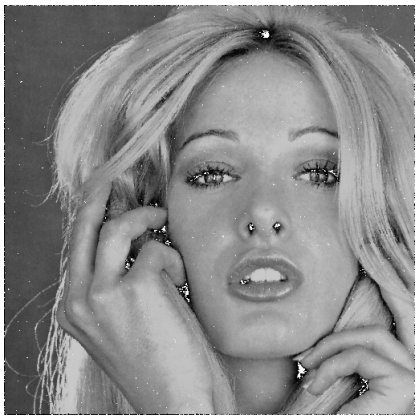
(a) Poškodený obrázok.



(b) Filtrovaný obrázok.

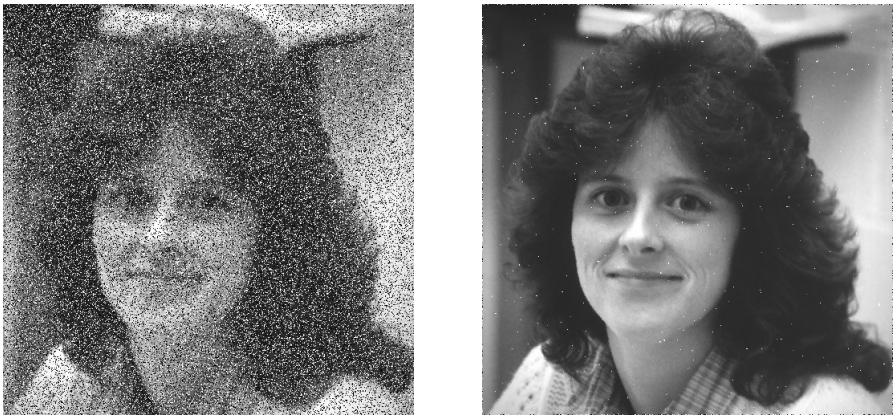
Obrázok A.19: Testovací obrázok mandril.tif pre šum s intenzitou 30%.

(a) Poškodený obrázok.



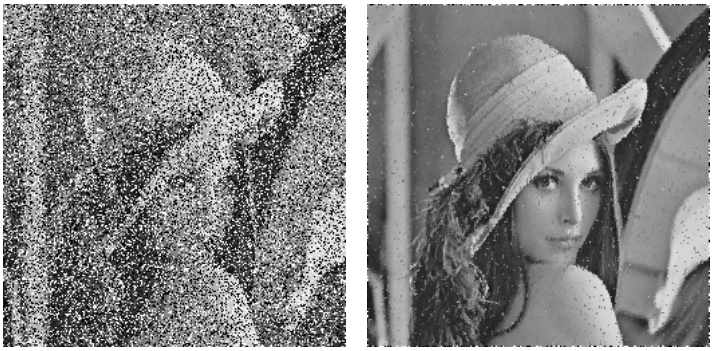
(b) Filtrovaný obrázok.

Obrázok A.20: Testovací obrázok woman_blonde.tif pre šum s intenzitou 30%.



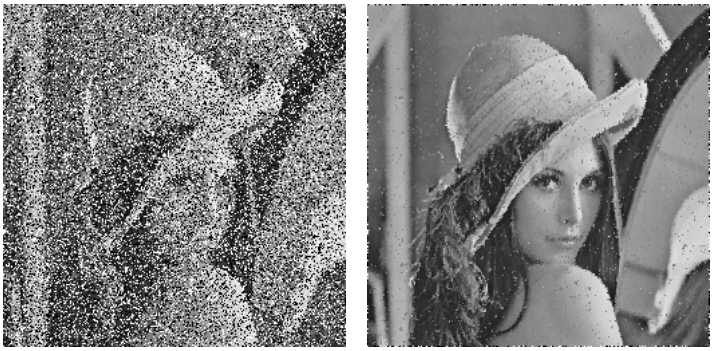
(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.21: Testovací obrázok woman_darkhair.tif pre šum s intenzitou 30%.



(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.22: Trénovací obrázok pre šum s intenzitou 40%.



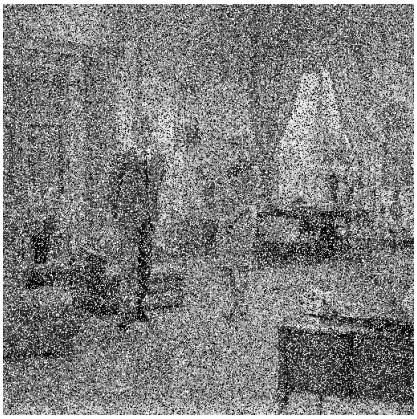
(a) Poškodený obrázok. (b) Filtrovaný obrázok.

Obrázok A.23: Testovací obrázok lena_gray_256.tif pre šum s intenzitou 40%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

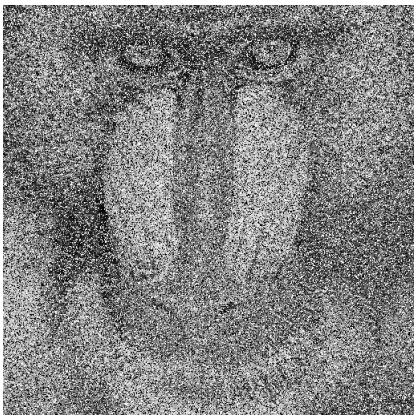
Obrázok A.24: Testovací obrázok lena_gray_512.tif pre šum s intenzitou 40%.



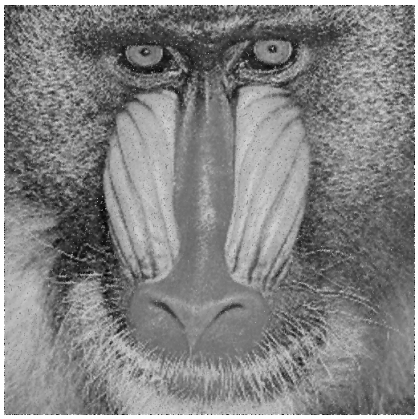
(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.25: Testovací obrázok livingroom.tif pre šum s intenzitou 40%.



(a) Poškodený obrázok.



(b) Filtrovaný obrázok.

Obrázok A.26: Testovací obrázok mandril.tif pre šum s intenzitou 40%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.27: Testovací obrázok woman_blonde.tif pre šum s intenzitou 40%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.28: Testovací obrázok woman_darkhair.tif pre šum s intenzitou 40%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.29: Trénovací obrázok pre šum s intenzitou 50%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.30: Testovací obrázok lena_gray_256.tif pre šum s intenzitou 50%.

(a) Poškodený obrázok.

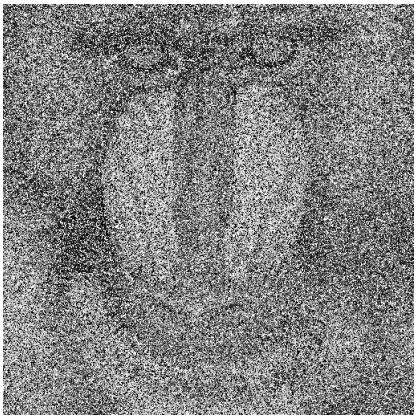
(b) Filtrovaný obrázok.

Obrázok A.31: Testovací obrázok lena_gray_512.tif pre šum s intenzitou 50%.

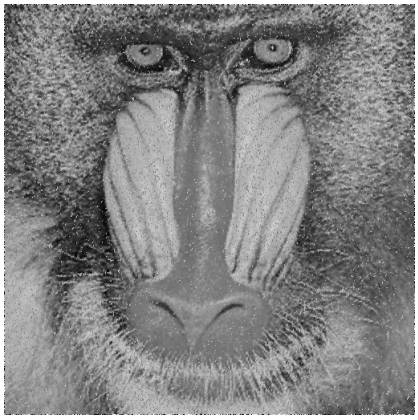
(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.32: Testovací obrázok livingroom.tif pre šum s intenzitou 50%.



(a) Poškodený obrázok.



(b) Filtrovaný obrázok.

Obrázok A.33: Testovací obrázok mandril.tif pre šum s intenzitou 50%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.34: Testovací obrázok woman_blonde.tif pre šum s intenzitou 50%.

(a) Poškodený obrázok.

(b) Filtrovaný obrázok.

Obrázok A.35: Testovací obrázok woman_darkhair.tif pre šum s intenzitou 50%.

Príloha B

Preklad a spustenie programu

Program je preložiteľný na systémoch Linux. Pre preklad je potrebný prekladač **g++**. Okrem štandardných C++ knižniciach je potrebná knižnica **OpenCV**, a pri paralelnom výpočte knižnica **OpenMP**. Program je preložiteľný pomocou *Makefile*-u, ktorý je k dispozícii spolu so zdrojovými súbormi v adresári *src*. Príkazom **make** sa vytvorí spustiteľný súbor coev. Táto verzia používa paralelizáciu. Sekvenčná verzia je preložiteľná pomocou príkazu **make ser**. Prekladom vytvorené súbory je možné odstrániť príkazom **make clean**.

Program sa dá spustiť pomocou príkazu **./coev**. Program potrebuje niekoľko súborov, ktorými bude pracovať. Cestu k tým súborom je potrebné zadať ako parametre programu. Tiež pomocou parametrov sa dá nastaviť, ako sa bude chovať program. Možné parametre programu a ich významy sú nasledujúce:

- **-coev** < originálny obrázok >< poškodený obrázok >< výstupný chromozóm detektoru >< výstupný chromozóm filtra >< výstupný chromozóm archivovaného detektoru >< výstupný chromozóm archivovaného filtra >< výstupný obrázok >
  Filter a detektor budú vyvinuté pomocou koevolúcie. Filter bude trénovaný len na poškodených pixeloch.

- **-coevall** < originálny obrázok >< poškodený obrázok >< výstupný chromozóm detektoru >< výstupný chromozóm filtra >< výstupný chromozóm archivovaného detektoru >< výstupný chromozóm archivovaného filtra >< výstupný obrázok >
  Filter a detektor budú vyvinuté pomocou koevolúcie. Filter bude trénovaný na všetkých pixeloch.

- **-fd** < originálny obrázok >< poškodený obrázok >< výstupný chromozóm detektoru >< výstupný chromozóm filtra >< výstupný obrázok >
  Filter a detektor budú vyvinuté bez koevolúcie. Filter bude trénovaný na poškodených pixeloch.

- **-fdall** < originálny obrázok >< poškodený obrázok >< výstupný chromozóm detektoru >< výstupný chromozóm filtra >< výstupný obrázok >
  Filter a detektor budú vyvinuté bez koevolúcie. Filter bude trénovaný na všetkých pixeloch.

- **-fo** < originálny obrázok >< poškodený obrázok >< výstupný chromozóm filtra > < výstupný obrázok >
  Vývin len filtra. Filter bude trénovaný na všetkých pixeloch.

- **-testc** < originálny obrázok >< poškodený obrázok >< chromozóm detektoru > < chromozóm filtra >< výstupný obrázok >
  Testovanie filtra a detektoru.

- **-testf** < originálny obrázok >< poškodený obrázok >< chromozóm >< výstupný obrázok >
  Testovanie filtra.

- **-median** < originálny obrázok >< poškodený obrázok >< výstupný obrázok >
  Filtrovanie pomocou mediánového filtra. Originálny obrázok je treba k výpočtu fitness (PSNR/MDPP) hodnoty.

- **-noise** < originálny obrázok >< výstupný poškodený obrázok >< pomer šumu >
  Vytvorí šum na obrázku, poškodí zadané percento pixelov.

- **-printch** < vstupný chromozóm >< výstupný chromozóm >
  Konvertuje chromozóm do formátu vhodného pre zobrazení pomocou **cgpviewer.exe**.

- **-fit** < obrázok 1 >< obrázok 2 >
  Vypočíta fitness (MDPP/PSNR) obrazu.

V prípade pokračovania výpočtu je treba zadať uložený chromozóm filtra (uložené chromozómy detektoru a filtra) ako posledný parameter (posledné dva parametre).

Príklad spustenia programu:
./coev -coev lena.png lena_30.png chrom30det.txt chrom30fil.txt chrom30arch_det.txt chrom30arch_fil.txt out_30.png

Príloha C

Obsah CD

Priložený CD obsahuje:

- Zdrojové súbory programu (adresár **src**).
- Spustiteľný súbor (pre systém Linux) (adresár **prog**).
- Programovú dokumentáciu (adresár **doc**).
- Technickú správu vo formátu PDF (adresár **report**).
- Zdrojové súbory technickej správy pre LaTeX (adresár **latex**).
- Sadu testovacích obrázkov (adresár **img**).
- Súbor readme.txt.