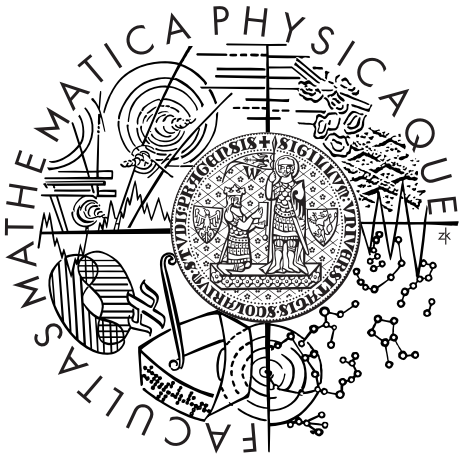
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Veronika Crkvová

Řízené křížení v genetických algoritmech

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Bojar Ondřej, Ph.D.
Studijní program: Informatika
Studijní obor: obecná informatika

Praha 2012

Poděkování především patří mému vedoucímu RNDr. Ondřeji Bojarovi, bez kterého by tato práce nikdy nemohla vzniknout, a Mgr. Viliamu Šimkovi za pomoc s Javou ve chvíli nouze.

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 3.8.2012 Podpis autora

Název práce: Řízené křížení v genetických algoritmech

Autor: Veronika Crkvová

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Bojar Ondřej, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Genetické algoritmy jsou počítačové programy, které se snaží napodobit vývoj živočišného druhu. Využívají se především k řešení problémů optimalizace a při řešení úloh, které jsou NP-obtížné nebo NP-úplné. Proces zpracování algoritmu často trvá velice dlouho, a proto jsme se v této práci začali zabývat řízeným křížením namísto náhodného křížení jedinců. Řízené křížení vychází z předpokladu, že některé rysy jsou užitečnější než jiné, a snaží se takové rysy ve stávající populaci automaticky odhalit a využít je v dalších generacích. Náš návrh a implementace řízeného křížení vedl na dvou konkrétních úlohách pouze ke slabému zlepšení výsledků ve srovnání s klasickými genetickými algoritmy a algoritmem Hill-climbingu.

Klíčová slova: genetický algoritmus, křížení, řízené křížení

Title: Directed Crossover in Genetic Algorithms

Author: Veronika Crkvová

Department: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Bojar Ondřej, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Genetic algorithms are computer programs that try to mimic the process of natural evolution. These algorithms are mostly used for solving problems of optimization, which can be NP-hard or NP-complete. The optimization using genetic algorithms is often very slow. In this thesis, we examine the idea of directed crossover instead of the standard random process. Directed crossover is based on the assumption that some features of population members are more useful than others. We thus try to identify these good features in the current population and promote them in future generations. In our implementation and experiments on two specific optimization tasks, directed crossover has lead to only a slight improvement over a standard genetic algorithm and Hill-climbing.

Keywords: genetic algorithm, crossover, directed crossover

Obsah

Úvod

Genetické algoritmy [4] (GA) jsou heuristické postupy, které simulují přírodní výběr. Využívají pravděpodobnost k tomu, aby nejlepší jedinci populace měli větší šanci předání genů na potomky. Řada optimalizačních problémů a NP-obtížných nebo NP-úplných problémů je řešena pomocí GA, protože prohledávání stavového prostoru řešení těchto úloh by trvalo příliš dlouho. GA hledají na tyto problémy optimální nebo alespoň suboptimální řešení v přijatelném čase.

Principem všech GA je postupná tvorba stále lepších generací řešení problému. Nové generace řešení problému vznikají za pomoci operátorů mutace, křížení a reprodukce (celý proces vzniku nové generace bude detailně popsán v první kapitole). Před aplikací výše vyjmenovaných operátorů je celá populace ohodnocena fitness funkcí, aby se oddělili „silní“ jedinci od „slabých“ jedinců. O tom, kteří jedinci budou kříženi, mutováni nebo reprodukováni, rozhoduje zvolená metoda selekce. Selekce je nejčastěji ovlivněna hodnotou fitness jedince. Jedinci s vyšší hodnotou fitness mají větší šanci aby byli vybráni selektorem.

My jsme se v této práci zaměříme především na binární operátory křížení, protože v nich hraje velkou roli náhoda, a to i přesto, že máme k dispozici celou předchozí generaci ohodnocených jedinců. Položili jsme si tedy otázku, zda lze křížení řídit na základě znalostí rozdílů mezi již ohodnocenými jedinci. Pokusíme se v populaci najít takový rozdíl, aby znamenal co největší přírůstek fitness a aplikovat ho na ostatní jedince. Snažíme se tak zajistit, abychom požadované řešení dostali za menší počet generací, než by potřeboval GA s klasickým náhodným křížením.

K implementaci řízeného křížení bude modifikována Java knihovna
Jenetics [10] a nedílnou součástí práce bude také porovnání výsledků řízeného křížení s klasickou metodou GA a implementací Hill-climbingu.

1. Genetické algoritmy

1.1 Obecný popis

Genetické algoritmy (GA) se řadí do třídy evolučních algoritmů [7], které mimo ně zahrnují také evoluční programování, evoluční strategii a genetické programování. Řadí se mezi stochastické metody prohledávání stavového prostoru, ale využívají také vlastností vlastních deterministických metod. Od stochastických metod převzaly stejnoměrné prohledání všech oblastí stavového prostoru („exploration“) a od deterministických metod zaměření úsilí do prohledávání nadějných oblastí („exploitation“). Jsou schopny vygenerovat relativně kvalitní řešení v přijatelně krátkém čase (vzhledem k obtížnosti problému). GA jsou založeny na mechanikách přirozeného výběru, křížení a mutace.

Goldberg ve své knize [4] formuloval základní rozdíly mezi GA a klasickou optimalizací (český překlad převzat z knihy Teoretické otázky neuronových sítí [6]):

1. GA pracuje s řetězcem zakódovaných hodnot parametrů, ne s parametry samotnými.
2. GA prohledává prostor parametrů vycházejíc z populace počátečních bodů, ne z jednoho bodu.
3. GA využívá pouze informace poskytované fitness funkcí, ne derivace nebo další doplňující znalosti.
4. GA využívá pravděpodobnostní mechanismus hledání.

1.2 Způsob použití

Nejvíce se GA používají k řešení problémů optimalizace. Problém může být formulován jako hledání minima nebo maxima funkce v závislosti na jejích parametrech. Např. problém učení neuronové sítě.

Dalším využitím je řešení NP-obtížných úloh či dokonce NP-úplných [3] (obtížně řešitelných nebo neřešitelných v polynomiálním čase).

1.3 Terminologie

Základní strukturou je *gen* (jinak je také např. nazýván *rys*). Gen je jednotkou genetické výbavy *jedince* (jedinec může být také jinak nazýván *genotyp*, *chromozom*, *struktura* či *řetězec*). Konkrétní význam genu závisí na řešeném problému a na použitém kódování (může se jednat o jeden bit, celé číslo, znak a nebo např. takzvané permutační kódování viz 1.4.3). Hodnoty, kterých gen nabývá, nazýváme *alely*. Geny jsou sdruženy do *chromozomů*. Soubor jednoho a více chromozomů je nazýván *genotyp*. Genotyp je kompletní genetická výbava jedince. Většina GA používá pouze jedince s jedním chromozomem a potom chromozom odpovídá genotypu. Genotyp obsahuje data, která jsou potřebná pro vytvoření *fenotypu*.

Fenotyp je vlastní řešení úlohy, které kromě genotypu obsahuje i *fitness funkci* (jinak také *účelovou funkci* nebo *ohodnocovací funkci*) a *hodnotu fitness* genotypu. Fitness funkce přiřadí hodnotu fitness konkrétnímu genotypu (tuto hodnotu také nazýváme *fitness value* nebo zkráceně *fitness*). Fitness jedince vyjadřuje vzdálenost tohoto řešení od optimálního řešení a určuje tak, které řešení je lepší a které horší. *Populace* je soubor genotypů a *generace* je populace, která je navíc ohodnocená fitness funkcí (je to soubor fenotypů).

1.4 Princip činnosti GA

Prvním krokem je inicializace, při které se vytvoří první populace. První populace je ohodnocena fitness funkcí. Poté následuje cyklus, který obsahuje postupně selekci, rekombinaci (křížení a mutaci), reprodukci a aplikaci fitness funkce na výslednou populaci. Cyklus se opakuje, dokud není splněna zastavovací podmínka. Jedinec s nejvyšším fitness v době zastavení je výstupem algoritmu a reprezentuje nejlepší nalezené řešení.

1.4.1 Inicializace

Nejdříve je potřeba vytvořit počáteční (první) generaci jedinců. Populace této generace může vzniknout několika způsoby:

1. *Inicializace populace s předem danými pravidly:* nejdříve vygenerujeme určité množství jedinců a z nich poté vybereme jedince do první generace. Výběr a tvorba jedinců jsou často omezeny sadou pravidel, která jsou definovaná danou úlohou.

2. *Náhodně:* jedince tvoříme náhodně (v rámci zachování pravidel, která říkají, jak má jedinec vypadat).

3. *První generace daná úlohou:* Jedince není třeba vytvářet, protože jsou pevně zadaní řešenou úlohou.

Každý jedinec v populaci představuje zakódované řešení zadané úlohy. Všichni jedinci populace jsou ohodnoceni fitness funkcí a tím je vytvořena první generace algoritmu.

1.4.2 Selekce

Následuje proces selekce. Selekce vytvoří 2 množiny: První množinou jsou jedinci, kteří budou reprodukováni, a druhá množina je určena k rekombinaci(to jsou jedinci ke křížení a mutaci). Selekce se snaží napodobit Darwinův přirozený výběr tak, že nevybírá pouze „nejlepší“ jedince (to jsou jedinci, kteří mají nejvyšší hodnoty fitness funkce). Každý jedinec má šanci dostat se do jedné nebo obou množin, ale jedinci s nejvyššími hodnotami fitness funkce mají největší šanci. Šance jedince dostat se do množin (a tak se dostat do další generace v podobě předaných genů) je závislá na metodě použité při selekci. Metod selekcí [4] bylo popsáno mnoho, a proto popíšeme jen nějaké (při výčtu selekcí bylo hojně čerpáno z diplomové práce Petra Pošíka [8]):

1. *Ruletová selekce (Roulette-wheel selection):* Pravděpodobnost výběru jedince je přímo úměrná jeho fitness. Selekce jedince se dá přirovnat k ruletě, kde jedinci s větší hodnotou fitness funkce mají větší zastoupení po obvodu a proto i větší šanci na výběr. Zároveň to ale znamená, že i jedinci, kteří mají hodnotu fitness funkce nižší, se mohou dostat do rekombinační dvojice.

2. *Pořadová selekce (Rank selection):* Pravděpodobnost přežití jedince u této selekce není přímo úměrná hodnotě jeho fitness. Metoda pořadové selekce odstraňuje problémy se vzorkováním u malých populací, odstraňuje problémy se škálováním (aby nedošlo k předčasné konvergenci) a zvládá maximalizační a minimalizační problémy. Stejně jako u ruletové selekce se nejdříve vypočítají hodnoty fitness jedinců v populaci, ale poté jsou podle této hodnoty seřazeni (od nejmenšího k největšímu) a nad jejich pořadím je teprve prováděna ruletová selekce. Například mějme 2 jedince v populaci, kteří mají fitness $f(A) = 1$ a $f(B) = 99$. Ruletová selekce by jim přiřadila pravděpodobnost přežití $p(A) = 0,01$ a $p(B) = 0,99$. U pořadové selekce je nejdříve seřadíme podle hodnoty fitness $por(A) = 1$ a $por(B) = 2$ a nad tímto pořadím provádíme ruletovou selekci. Pravděpodobnost výběru jedinců u této metody by tedy nakonec byla $p(A) = 0,33$ a $p(B) = 0,66$.

3. *Deterministický výběr (Deterministic sampling):* Jde o další modifikaci ruletové selekce. Nejdříve se vypočítá fitness každého jedince, z ní potom pravděpodobnost výběru jako u ruletové selekce a na základě této hodnoty se vypočítá předpokládaný počet jeho kopií (PPK). Tedy PPK je jeho pravděpodobnost výběru vynásobena velikostí populace. Výsledkem bude reálné číslo, na které se ještě aplikuje celá část (a přesně tolikrát se jedinec zkopíruje do nové populace). Poté je ještě třeba obsadit zbývající volná místa v populaci. Jejich počet se vypočítá jako součet desetinných míst před aplikací celé části již kopírovaných jedinců. Volná místa obsadíme tak, že setřídíme populaci podle desetinných částí PPK a vybereme potřebný počet jedinců z vrchu tohoto setříděného seznamu.

4. *Zbytkový stochastický výběr (Remainder stochastic sampling):* Zbytkový stochastický výběr je obměnou deterministického výběru. Začíná stejně, ale rozdílem je, že zbytek neobsazených míst v nové populaci se rozděluje na základě aplikace klasické ruletové metody.

5. *Turnajová selekce (Tournament Selection):* Podle velikosti populace se několikrát uspořádá turnaj (tolikrát, jako je velikost populace), aby se obsadil každý jedinec v nové generaci. Ze staré generace se v každém kole náhodně vybírají n-tice jedinců a do nové generace se vždy vloží nejlepší jedinci z těchto n-tic. Jedinci se v n-ticích mohou nebo nemusí opakovat (záleží na implementaci). Důležitá je volba n, protože v krajním případě, kdy by se n rovnalo velikosti populace a jedinci by se nemohli v n-tici opakovat, by nová populace byla složena z nejlepších jedinců předchozí populace a nastala by předčasná konvergence.

Z vybrané množiny pro křížení a mutaci jsou generováni noví jedinci, kteří nahradí ty staré. Generování nových jedinců probíhá pomocí operátorů křížení a mutace, které si podrobněji popíšeme:

1.4.3 Křížení

Při *křížení* (Crossover) vznikají zcela noví jedinci, kteří nebyli součástí předchozí populace. Noví jedinci vznikají pomocí náhodné výměny informací obsažených v rodičích. Vzniklá zcela nová řešení mohou GA často vymanit z partikulárního lokálního minima. Operátorů křížení je více druhů a jejich úspěšnost závisí na řešené úloze. Uvedeme si výčet pár nejznámějších operátorů křížení (čerpáno především z knihy Genetické algoritmy a genetické programování [5]):

1. *Jednobodové, dvoubodové a vícebodové křížení:* Nazývány jsou podle toho, kolik vybereme bodů křížení. Volbou n bodů se oba rodiče rozdělí na $n+1$ částí. Noví potomci poté vznikají tak, že se tyto části rozdělí mezi dva vznikající potomky. Např. se liché části z prvního rodiče a sudé části z druhého rodiče nakopírují do prvního potomka. Druhý potomek dostane části přesně opačné (tedy sudé části z prvního rodiče a liché z druhého rodiče).

2. *Uniformní řešení:* Je to zobecnění předchozích k-bodových křížení. Při vzniku prvního potomka se pro každý gen z chromozomu rozhodne s pravděpodobností $p = 0,5$, zda se převezme alelu z prvního nebo druhého rodiče. Takto se postupuje pro každý gen prvního potomka a druhý potomek dostane poté alely vždy od opačného rodiče.

Body 1. a 2. lze použít jak pro chromozomy, které mají v genech alely nabývající binární hodnoty, tak i pro chromozomy s celočíselným obsahem. Další uváděné operátory již ale budou pro chromozomy, které mají reálné hodnoty alel v genech.

3. *Aritmetický:* Nová hodnota alely vznikne aritmetickým průměrem rodičovských alel.

4. *Geometrický:* Hodnota alely nového jedince je odmocninou násobku rodičovských alel.

Další uvedené operátory jsou používány v případě permutačního kódování (každá hodnota alely se v chromozomu vyskytuje pouze jednou). Takový chromozom může reprezentovat např. určité pořadí objektů či akcí, které je nutné respektovat.

5. *Křížení s částečným přiřazením (PMX, Partially matched crossover):* Záměrem tohoto operátoru je využít uspořádané podmnožiny permutace jednoho z rodičů a přitom zachovat pořadí a pozici co nejvíce objektů podle permutace odpovídající druhému z rodičovských chromozomů. Princip tohoto operátoru si ukážeme na příkladu:

   Mějme rodiče:

   $p_1 = (1, 2, 3, |4, 5, 6, 7, |8, 9)$

   $p_2 = (9, 3, 4, |1, 8, 7, 6, |5, 2)$

   Symboly | v tomto příkladu označují body křížení. Nejdříve se vymění úsek mezi těmito body a ostatní pozice v nově vytvářených chromozomech (n1,n2) zůstanou prozatím prázdné (označíme X).

$n_1 = (X, X, X, |1, 8, 7, 6, |X, X)$

$n_2 = (X, X, X, |4, 5, 6, 7, |X, X)$

Při této výměně je nutné si zapamatovat jednotlivé hodnoty, které se výměny účastnily. Pamatovat si je budeme formou přepisovacích pravidel: $1 \leftrightarrow 4$, $8 \leftrightarrow 5$, $7 \leftrightarrow 6$, $6 \leftrightarrow 7$

Poté se do prozatím prázdných míst chromozomů nakopírují z rodičů ty alely, kde nehrozí konflikt ve vznikajících permutacích:

$n_1 = (X, 2, 3, |1, 8, 7, 6, |X, 9)$

$n_2 = (9, 3, X, |4, 5, 6, 7, |X, 2)$

K obsazení zbylých volných míst se využijí substituční pravidla, která jsme si výše zapamatovali. Přepisovací pravidla se aplikují na hodnoty, kde je X. Např. na první pozici prvního rodiče byla 1. Proto za X do prvního potomka dosadíme 4 (podle pravidla $1 \leftrightarrow 4$).

$n_1 = (4, 2, 3, |1, 8, 7, 6, |5, 9)$

$n_2 = (9, 3, 1, |4, 5, 6, 7, |8, 2)$

Výslední potomci jsou permutace a tedy přípustná řešení problému.

1.4.4 Mutace

Mutace se aplikuje na celou množinu vybranou pro rekombinaci. Může se provádět buď před křížením nebo i po něm. Operátor mutace většinou velmi jednoduchým způsobem a s relativně malou pravděpodobností náhodně mění hodnotu alel v jednotlivých genech vybraných jedinců. Díky aplikaci mutačního operátoru se můžeme v prohledávaném prostoru dostat k řešením, která by nebyla na základě křížení dosažitelná. Při aplikaci tohoto operátoru se prochází jednotlivé geny v chromozomech a podle předem dané pravděpodobnosti mutace se může změnit jejich hodnota. Pravděpodobnost mutace se nesmí nastavit příliš velká, protože jinak se ztrácí výhody GA a prohledávání stavového prostoru je potom více náhodné. Typicky je tedy pravděpodobnost mutace nastavena na menší hodnotu (1% a méně), ale tato hodnota se může za běhu algoritmu měnit podle potřeby. Opět si uvedeme několik druhů mutačních operátorů [5]:

1. *Binární operátor mutace:* Tento druh mutace využijeme u chromozomů, které mají binární kódování. Pokud dojde k změně alely, je to v tomto případě z 0 na 1 anebo opačně.

2. *Reálné a celočíselné operátory mutace:* Operátorů mutace pro reálné a celočíselné kódování mutace je mnoho. Mezi ně patří například: *náhodná výměna* (nahrazení původní alely náhodně vygenerovanou hodnotou), *aditivní změna* (přičtení náhodné hodnoty náležící na intervalu $\langle$ -ε ; ε $\rangle$ k hodnotě alely) a *multiplikativní změna* (vynásobení původní hodnoty náhodně vygenerovaným číslem na intervalu $\langle$ $1 - \varepsilon$; $1 + \varepsilon$ $\rangle$)

3. *Výměnné operátory mutace:* Tento operátor se využije především u chromozomů, které mají permutační kódování. Pokud bychom totiž změnili hodnotu jakékoliv jedné alely, z chromozomu navíc vypadne původní číslo a mohlo

by se stát, že se nějaká hodnota bude opakovat. Operátor vezme 2 náhodné pozice genů v jednom chromozomu a vymění mezi nimi obsah. Permutace tak zůstane zachována.

4. *Posuvné operátory mutace:* Operátory posuvné mutace náhodně zvolí gen a ten je ve stejném chromozomu posunut na náhodně vygenerovanou pozici. Posunou se tedy geny od místa, kde vkládáme, až do místa kde bereme. Operátor lze využít pro mnoho druhů kódování (např. reálné, permutační, ale i binární). Pro lepší pochopení operátoru si uvedeme příklad pro binární kódování [1]:

   $(1, 1, 0, 0, 1, 1, 0, \underline{\boldsymbol{0}}, 0) \rightarrow (1, 1, 0, 0, \underline{\boldsymbol{0}}, \underline{1}, \underline{1}, \underline{0}, 0)$

5. *Inverzní operátory mutace:* Při této mutaci náhodně zvolíme 2 body v chromozomu, které vytvoří úsek. V označeném úseku poté dojde k invertování hodnot zrcadlově podle jeho středu. Operátor inverzní mutace lze opět využít pro mnoho druhů kódování a to včetně permutačního. Aplikace operátoru na permutační kódování by vypadala takto:

   $(A, \underline{\mathbf{B}}, \underline{\mathbf{C}}, \underline{\mathbf{D}}, E, F) \rightarrow (A, \underline{\mathbf{D}}, \underline{\mathbf{C}}, \underline{\mathbf{B}}, E, F)$

1.4.5 Reprodukce

Nakonec následuje proces *reprodukce* (*reproduction*), ve kterém všechny jedince z množiny určené k reprodukci kopírujeme do nové populace. Jedinci byli vybráni do množiny zvoleným selektorem. Tento operátor je umělou verzí přirozeného výběru Darwinovy teorie.

2. Řízené křížení

V této kapitole blíže popíšeme realizaci řízeného křížení a jeho variant.

2.1 Motivace

Nejznámější operátory křížení pro bitové chromozomy jsou popsané v první kapitole. Jedná se o jednobodové nebo vícebodové křížení a uniformní řešení. Oba operátory kříží jedince na základě náhodného přenosu genů z rodičů na potomky. Zeptali jsme se tedy, zda by toto křížení nemohlo být více řízené. Pokusili jsme se využít informace, které již o populaci máme. Víme totiž, kteří jedinci jsou více úspěšní (mají větší fitness hodnotu), a známe jejich podobu. Začali jsme tedy porovnávat jedince mezi sebou, snažili se najít nejvíc efektivní změnu mezi jedinci a tuto změnu aplikovat na další jedince z populace.

2.2 Popis algoritmu

Popis řízeného křížení budeme ukazovat na příkladu.

2.2.1 Vstupní data

Vstupem bude populace, která je předem ohodnocena funkcí fitness. V našem případě to bude populace o 6-ti jedincích velikosti 4 bitů. Funkci fitness označíme jako *f*.

Listing 2.1: Počáteční populace

```
0. [0000] f = 0
1. [1010] f = 2
2. [1111] f = 4
3. [1101] f = 3
4. [0100] f = 1
5. [1110] f = 3
```

2.2.2 Hledání nejefektivnější změny

Nejprve projdeme celou populaci tak, že budeme vždy spolu porovnávat každé 2 jedince. Z každého porovnání si zaznamenáme tzv. *změnu*. Změnou se rozumí rozdíl dvou porovnávaných jedinců. Prochází se postupně oba chromozomy a porovnávají se hodnoty genů na stejných pozicích v chromozomu.

Např. Budeme porovnávat první místo v obou chromozomech. Pokud má první jedinec na této pozici 0 a druhý 1, potom zaznamenáme na první místo změny 1.

Listing 2.2: Ukázka změny na první pozici chromozomu

```
1. jedinec      [0...]
2. jedinec      [1...]
Výsledná změna  [1...]
```

Pokud bude situace opačná (první jedinec má na této pozici 1 a druhý 0), zaznamená se na první pozici změny -1. Kdyby měly oba jedinci na první pozici stejnou hodnotu genu, bude na prvním místě změny hodnota 0.

Takto projdeme všechny pozice v chromozomu a vždy zaznamenáme rozdíl na příslušné místo změny.

Listing 2.3: Ukázka výpočtu celé změny

```
Příklad změny: rozdíl 2 jedinců
1. jedinec je [1010] s fitness f = 2
2. jedinec je [1111] s fitness f = 4
Vypočítejme tedy změnu z jedince 1. na 2.:

  fitness změny je 4 - 2 = +2 (z fitness 0 na fitness 2)
  změna z prvního na druhého: [0,+1,0,+1]

Výsledkem tohoto porovnání je:
  změna [0,+1,0,+1] s hodnotou +2
```

Stejně vypočítáme změnu pro všechny dvojce jedinců z populace. V našem příkladu si uděláme tabulku, která ukazuje změny fitness: viz tabulka 2.1.

•	0.	1.	2.	3.	4.	5.
0.	•	+2	+4	+3	+1	+3
1.	-2	•	+2	+1	-1	+1
2.	-4	-2	•	-1	-3	-1
3.	-3	-1	+1	•	-2	0
4.	-1	+1	+3	+2	•	+2
5.	-3	-1	+1	0	-2	•

Tabulka 2.1: Tabulka změn fitness hodnot. Levý sloupec obsahuje jedince, ze kterých se vychází

K jednotlivým změnám fitness ještě vypočítáme příslušnou změnu:

Listing 2.4: Seznam všech změn příslušných k tabulce fitness hodnot

```
0. na 1. [+1,0,+1,0]     (z 1. na 0. [-1,0,-1,0])
0. na 2. [+1,+1,+1,+1]   (z 2. na 0. [-1,-1,-1,-1])
0. na 3. [+1,+1,0,+1]    (z 3. na 0. [-1,-1,0,-1])
1. na 2. [0,+1,0,+1]     (z 2. na 1. [0,-1,0,-1])
1. na 3. [0,+1,-1,+1]    (z 3. na 1. [0,-1,+1,-1])
2. na 3. [0,0,-1,0]      (z 3. na 2. [0,0,+1,0])
0. na 4. [0,+1,0,0]      (z 4. na 0. [0,-1,0,0])
0. na 5. [+1,+1,+1,0]    (z 5. na 0. [-1,-1,-1,0])
1. na 4. [-1,+1,-1,0]    (z 4. na 1. [+1,-1,+1,0])
1. na 5. [0,+1,0,0]      (z 5. na 1. [0,-1,0,0])
2. na 4. [-1,0,-1,-1]    (z 4. na 2. [+1,0,+1,+1])
2. na 5. [0,0,0,-1]      (z 5. na 2. [0,0,0,+1])
3. na 4. [-1,0,0,-1]     (z 4. na 3. [+1,0,0,+1])
3. na 5. [0,0,+1,-1]     (z 5. na 3. [0,0,-1,+1])
4. na 5. [+1,0,+1,0]     (z 5. na 4. [-1,0,-1,0])
```

Poté projdeme všechny změny a budeme hledat stejné (může nastat situace, kdy je více dvojic, které mají stejnou změnu). Pokud najdeme nějaké stejné změny, sloučíme je a sečteme jejich fitness. V našem příkladě je takovou změnou např. ta z 0. na 1. a z 4. na 5.: sloučíme tyto změny a sečteme jejich fitness.

Listing 2.5: Sloučení 2 stejných změn

```
0. na 1. [+1,0,+1,0] f=+2
4. na 5. [+1,0,+1,0] f=+2
Výslednou fitness hodnotou změny bude 2+2 = +4
```

Sloučením změn pozměníme hodnoty v tabulce 2.1. Tabulka 2.2 je již s aktuálními hodnotami.

•	0.	1.	2.	3.	4.	5.
0.	•	+4	+4	+3	+1	+3
1.	-4	•	+2	+1	-1	+1
2.	-4	-2	•	-1	-3	-1
3.	-3	-1	+1	•	-2	0
4.	-1	+1	+3	+2	•	+4
5.	-3	-1	+1	0	-4	•

Tabulka 2.2: Aktualizovaná tabulka změn fitness hodnot. Levý sloupec obsahuje jedince, ze kterých se vychází

Z této tabulky již vybereme změnu, která má nejvyšší fitness hodnotu a prohlásíme ji za "nejlepší změnu". Pokud bude takových změn více, prostě vezmeme tu první. Další zpracování nových jedinců záleží na vybrané variantě řízeného křížení. Varianty existují 2:

1. Verze se zahazováním
2. Verze bez zahazování

Obě varianty si popíšeme v dalších kapitolách.

2.2.3 Verze se zahazováním

Verze se zahazováním počítá s tím, že velikost populace se nemění. Projdeme celou vstupní populaci a na základě dané pravděpodobnosti křížení aplikujeme na vybrané jedince nejlepší změnu(viz listing 2.6). Po vytvoření nového jedince ho vždy vložíme do populace na místo jedince, ze kterého vznikl. Výstupem tedy bude populace stejné velikosti jako vstupní, která obsahuje nové jedince, kteří vznikli z některých původních.

Listing 2.6: Aplikace změny na vybraného jedince

```
Nejlepší změna budiž [+1,0,+1,0]
Původní jedinec: [0100]
Nový jedinec po aplikaci změny: [1110]
```

2.2.4 Verze bez zahazování

Tato verze žádného jedince nikdy nezapomene. Velikost populace tedy roste vkládáním nově vzniklých jedinců. Stejně jako ve verzi se zahazováním projdeme celou vstupní populaci a na základě dané pravděpodobnosti křížení aplikujeme na vybrané jedince nejlepší změnu(tím vytváříme jedince nové). Novou populaci vytvoříme tak, že do ní vložíme tu starou a přidáme nově vzniklé jedince.

3. Experimenty

Tato kapitola nejprve obecně popíše algoritmus Hill-climbing a klasickou implementaci genetického algoritmu. Poté bude srovnávat implementaci řízeného křížení s klasickou metodou genetického algoritmu a s implementací Hill-climbing.

3.1 Hill-climbing

3.1.1 Obecný popis algoritmu

Algoritmus volí metodu lokálního prohledávání. Nejprve zvolíme nějaké přípustné řešení jako počáteční a označíme ho za aktuální. V každém kroku poté pro aktuální řešení vygenerujeme všechna sousední a vypočteme pro ně jejich hodnotu účelové funkce. Na základě hodnot účelové funkce zjistíme, zda se v okolí nevyskytuje řešení lepší než naše aktuální. Pokud ano, zvolíme si toto řešení za naše aktuální (vždy máme jedno aktuální řešení, které může být nahrazeno pouze lepším). Hill-climbing skončí po předem daném počtu iterací. Hlavní nevýhodou tohoto algoritmu je, že může nalézt pouze lokálně optimální řešení.

3.1.2 Vlastní implementace Hill-climbingu

Vlastní implementace Hill-climbing, která byla vytvořena rozšířením knihovny Jenetics. Tato implementace je dále použita pro srovnání s řízeným křížením bez zahazování na jednotlivých úlohách.

Inicializace

Pro tuto implementaci Hill-climbing je obsah první populace plně nechán na implementaci řešené úlohy. První populace ale musí být veliká přesně jako velikost chromozomu jedince zvýšená o jedna (pokud je tedy např. velikost chromozomu jedince velkosti 20, potom velikost populace musí být 21). Po vytvoření první populace je po čas inicializace ještě potřeba celou populaci ohodnotit fitness funkcí.

Vytváření nových generací

Nejprve se z původní populace vybere jedinec s největším fitness. Pro tohoto jedince poté vygenerujeme všechny sousední jedince a ty spolu s ním vložíme do populace nové. Tento cyklus opakujeme do té doby, dokud není splněna ukončovací podmínka nebo po předem zadaném počtu cyklů (počet cyklů vytváření nové populace je často dán úlohou).

3.2 Základní genetický algoritmus

Všechny implementace základního genetického algoritmu jsou převzaty z knihovny Jenetics[10].

3.2.1 Inicializace

První generace vzniká tak, jak je to popsáno v kapitole 1.4.1. Po inicializaci následuje proces vytváření nových generací, který je popsán v následující kapitole.

3.2.2 Cyklus vytváření nových generací

Vždy se ze staré populace jedinců vytvoří nová. Novou populaci, která je ohodnocena fitness funkcí, nazýváme generace. Nejprve se populace pomocí operátoru selekce (viz 1.4.2) rozdělí na 2 množiny:

1. První množina se jmenuje *survivors*. Survivors jsou prvky, které se beze změny zkopírují do populace další generace (viz reprodukce v 1.4.5).

2. Druhá množina má název offsprings. Z ní se vybírají jedinci určení k mutaci a křížení. Jedinci jsou z této množiny vybráni k mutaci či křížení na základě pravděpodobnosti určené při definici úlohy (pravděpodobnost mutace a křížení se může měnit).

Po rozdělení populace se aplikují operátory mutace a křížení a následně vznikne nová populace slitím množin survivors a offsprings. Při vzniku nové populace se kontroluje stáří jednotlivých fenotypů (počet generací po které se nezměnil). Maximální stáří je předem dáno a pokud je překročeno, je tento fenotyp nahrazen novým náhodně vygenerovaným. Nakonec je nová populace ohodnocena funkcí fitness (tím je proces vytváření nové generace hotový).

3.3 Počítání jedniček

3.3.1 Obecný popis úlohy

Tato úloha také bývá nazývána Ones counting. Je to jedna z nejjednodušších úloh, která se často používá pro srovnání různých implementací algoritmů. Úkolem je maximalizovat počet jedniček v binárním chromozomu.

3.3.2 Fitness funkce

Úkolem fitness funkce je spočítat počet jedniček v binárním chromozomu. Počet bitů nastavených na hodnotu 1 je přímo hodnotou fitness. Tedy např. chromozom [1101|10001111|00000111] má fitness rovno 11. Fitness funkce bude pro všechny implementace stejná.

3.3.3 Inicializace

Počáteční populace bude pro všechny implementace stejná. Má 50 chromozomů délky 20. Každý chromozom z této populace má všechny bity nastaveny na 0. Jedná se tedy o 50 kopií tohoto chromozomu [0000|00000000|00000000]. Tato populace je poté ohodnocena fitness funkcí. Hodnota fitness bude v počáteční populaci pro všechny chromozomy hodnoty 0, protože všechny bity začaly na nulové pozici.

3.3.4 Implementace klasického genetického algoritmu

Tato úloha je přímo v implementovaná v knihovně Jenetics. Nejprve proběhne inicializace počáteční populace(viz kapitola 3.3.3).

Po inicializaci následuje vytváření nových generací, které je stejné jako je popsané v části 3.2.2. Populace je tedy rozdělena na survivors a offsprings. Rozdělení probíhá pomocí ruletového selektoru (viz 1.4.2). Množina offsprings je měněna pomocí binárního operátoru mutace s pravděpodobností 0.55 a operátoru jednobodového křížení s pravděpodobností 0.06. Nakonec je nová populace ohodnocena fitness funkcí a jsou vyměněny příliš staré fenotypy.

3.3.5 Implementace pomocí řízeného křížení

Tato implementace je blíže popsána v kapitole 2. Budeme testovat jak variantu se zahazováním (viz 2.2.3), tak bez zahazování (viz 2.2.4). Počáteční populace a fitness funkce bude stejná jako u ostatních uváděných implementací této úlohy.

3.3.6 Implementace Hill-climbing

Jedná se o vlastní implementaci Hill-climbing algoritmu, jak byla popsána v kapitole 3.1.2, která byla upravena aby odpovídala jejím potřebám. Byla tedy upravena velikost populace a chromozomů při inicializaci.

3.4 Binární batoh

3.4.1 Obecný popis úlohy

Na začátku úlohy jsou dány předměty, které mají každý svou hmotnost a hodnotu (cena předmětu). Úkolem je vybrat ze zadaných předmětů množinu tak, aby se vešla do batohu nosnosti N a zároveň byl součet jejich hodnot co největší. Seznam předmětů, které jsou obsažené v množině, je reprezentován pomocí binárního chromozomu. Hodnota genu je jednoduše nastavena na 1, pokud je konkrétní předmět obsažen v množině. Problém binárního batohu se často nazývá 0/1 Knapsack.

3.4.2 Fitness funkce

Fitness jedince je v této úloze součet hodnot vybraných předmětů. Pokud by ale hmotnost vybraných předmětů překročila nosnost batohu N, potom je fitness příslušného jedince rovná 0. Tím se zajistí, aby jedinci s hmotností překračující N nebyli vybíráni.

3.4.3 Inicializace

Počáteční populace má 100 jedinců délky 15 bitů. Všichni jedinci mají přesně jeden chromozom, který má v počáteční populaci všechny bity nastavené na 0. Nakonec inicializace je ještě počáteční populace ohodnocena funkcí fitness a jelikož tato populace neobsahuje žádný gen, který by měl hodnotu 1, mají všichni jedinci fitness 0.

3.4.4 Implementace klasického genetického algoritmu

Popis implementace v knihovně Jenetics. Inicializace proběhne podle kapitoly 3.4.3. Poté následuje vytváření nových generací, které probíhá podle algoritmu popsaného v sekci 3.2.2. Populace je rozdělena na survivors a offsprings pomocí ruletového selektoru (viz 1.4.2). Množina offsprings je změněna operátorem binární mutace s pravděpodobností 0.115 a jednobodového křížení s pravděpodobností 0.16. Nakonec jsou příliš staří jedinci nahrazeni nově vygenerovanými a celá populace je ohodnocena fitness funkcí.

3.4.5 Implementace pomocí řízeného křížení

Implementace se zahazováním i bez zahazování (viz 2) s počáteční generací a fitness funkcí stejnou jako pro všechny implementace.

3.4.6 Implementace Hill-climbing

Opět se jedná o vlastní implementaci Hill-climbing algoritmu jak byla popsána v kapitole 3.1.2. Pro každý test je třeba přizpůsobit velikost populace a chromozomů, aby odpovídala omezovacím podmínkám (konkrétní hodnoty při jednotlivých budou uvedeny v tabulkách).

3.5 Srovnání implementací

Srovnávali jsme hodnoty fitness na jednotlivých úlohách.

Tabulka 3.1: Počítání jedniček: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,1, počet generací = 10

•	MIN f	MAX f	Průměr f
Hill-climbing	10	10	10
Klasická implementace	29	36	32,43333333
Řízené křížení bez zahazování	36	46	40,83333333
Řízené křížení se zahazováním	37	44	40,66666667

Tabulka 3.2: Počítání jedniček: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,1, počet generací = 20

•	MIN	MAX	Průměr
Hill-climbing	20	20	20
Klasická implementace	32	38	33,96666667
Řízené křížení bez zahazování	42	49	45,05
Řízené křížení se zahazováním	40	50	45,86666667

V tabulkách 3.1 a 3.2 je výsledek porovnání jednotlivých implementací na úloze počítání jedniček(viz 3.3). Nejprve jsme pustili implementace pro počet generací 10 a poté 20 a porovnali jsme průměrné hodnoty fitness. Všechny implementace byly puštěny se stejnou pravděpodobností mutace a křížení. Maximální hodnota fitness funkce těchto příkladů počítání jedniček je 50.

Tabulka 3.3: Binární batoh: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,2, počet generací = 10

•	MIN	MAX	Průměr
Hill-climbing	145,4444037	183,5841081	165,1498153
Klasická implementace	125,3009758	185,0752377	156,7729073
Řízené křížení bez zahazování	127,6496782	193,8896548	160,3377982
Řízené křížení se zahazováním	125,0968462	180,1796982	155,0405564

Tabulka 3.4: Binární batoh: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,2, počet generací = 20

•	MIN	MAX	Průměr
Hill-climbing	155,6313364	191,2670737	172,3350826
Klasická implementace	125,5995053	183,3427399	157,5980993
Řízené křížení bez zahazování	135,8946448	205,2108895	167,9989682
Řízené křížení se zahazováním	127,337926	176,6679779	159,5077561

Tabulky 3.3 a 3.4 jsou obdobným srovnáním jako 3.1 a 3.2. Opět jsme porovnávali implementace na běhu 10-ti a 20-ti generací. Byly pouze změněny testovací pravděpodobnosti mutace a křížení. Uvedené průměrné hodnoty fitness jsou průměrem z 40 běhů programu.

Je vidět, že Hill-climbing u úlohy počítání jedniček nemá žádný rozptyl fitness hodnot. To je způsobeno tím, že počáteční populace je naplněná samými nulami a tak počítá vždy stejně, protože Hill-climbing se v každé iteraci posune maximálně na sousední hodnotu. U úlohy binárního batohu už rozptyl hodnot pozorujeme. To je proto, že se předměty v úloze binárního batohu generují náhodně (ale s určitými omezeními daných implementací). V úloze počítání jedniček dopadl Hill-climbing z uvedených implementací výrazně nejhůře, protože při posunu na sousedního jednice se můžeme hodnotu fitness zvednout maximálně o 1, a to je nejmenší možná. Naopak u úlohy binárního batohu se Hill-climbing osvědčil jako nejjednodušší a nejlepší řešení. V této úloze sousední jedinci znamenají, že se počet přemětů v batohu buď o 1 sníží nebo zvýší.

Řízené křížení jsme testovali v obou verzích (se zahazováním i bez). Je důležité říci, že varianta bez zahazování trvá ze všech uvedených implementací nejdéle, protože velikost populace stále narůstá (a s ní i počet dvojic, které jsou určené k porovnání). Doba počítání genetického algoritmu s řízeným křížením bez zahazování prudce narůstá, pokud zvedáme počet generací nebo pravděpodobnost křížení. Jak v úloze počítání jedniček, tak v úloze binárního batohu, se můžeme

přesvědčit, že rozdíl mezi fitness hodnotou verze se zahazováním a bez zahazování není tak velká. Proto doporučujeme spíše použití verze se zahazováním.

Řízené křížení dopadlo ve většině případů v uvedených testech o trochu lépe, než klasická varianta genetického algoritmu. V úloze počítání jedniček se ukázalo toto křížení dokonce o dost lepší, ale v úloze binárního batohu začne být o něco lepším až po 20 generacích.

Úlohu binárního batohu jsme dosud pouštěli na náhodně vygenerovaných předmětech. Proto jsme se rozhodli zkusit provést experimenty ještě na jedné vybrané startovní množině předmětů. Tabulky 3.5 a 3.6 ukazují výsledek tohoto experimentu. Výsledek je velice podobný pouštění s náhodně vygenerovanými předměty (to je proto, že předměty jsou náhodně generované pouze v určitém rozsahu hodnot).

Tabulka 3.5: Binární batoh: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,2, počet generací = 10, stejné startovní předměty

•	MIN	MAX	Průměr
Hill-climbing	157,056	157,056	157,056
Klasická implementace	135,809	166,761	146,299
Řízené křížení bez zahazování	139,4298032	178,3588068	156,6860766
Řízené křížení se zahazováním	1142,259	169,636	151,4786236

Tabulka 3.6: Binární batoh: Pravděpodobnost mutace = 0,05, pravděpodobnost křížení=0,2, počet generací = 20, stejné startovní předměty

•	MIN	MAX	Průměr
Hill-climbing	169,865	169,865	169,865
Klasická implementace	151,046	167,8	160,9199375
Řízené křížení bez zahazování	145,132	177,33773	162,8816142
Řízené křížení se zahazováním	148,541	168,0499	159,2615354

4. Srovnání s podobnými technikami z literatury

V této kapitole nejdříve obecně popíšeme techniku řízené mutace, která je popsána v článku Directed mutation in genetic algorithms od D. Bhandari [2], a poté ji stručně porovnáme s naší navrženou metodou řízeného křížení.

4.1 Řízená mutace

Jedná se o techniku řízené mutace, která byla navržena tak, aby urychlila genetický algoritmus řešící optimalizaci komplexní funkce. Podobně jako řízené křížení využívá tato metoda informace z předchozí generace.

4.1.1 Obecný popis genetického algoritmu využívající řízenou mutaci

Řeší se optimalizace komplexní funkce $f(x)$, která je zároveň i fitness funkcí popisovaného genetického algoritmu. Funkce $f(x)$ má n parametrů (x=)$x_1, x_2, ...x_n$. Každý parametr je zakódován jako sekvence nul a jedniček (kódování používané v článku není pro porovnání s řízeným křížením podstatné, a proto se mu nebudeme blíže věnovat). Jedincem rozumíme soubor zakódovaných parametrů pro jedno řešení fitness funkce f. Tedy například x je sada parametrů $x = x_1, x_2, ...x_n$, kde x_1 může například být 10000101.

Řízená mutace

Celá aktuální generace se nejprve dekóduje na skutečné hodnoty. Nový jedinec x^* je vytvořen z nejlepšího jedince z aktuální generace a nejlepšího jedince z předcházející generace podle vzorce $x^* = x^*_\tau + \alpha g(x^*_t, x^*_\tau)$. Kde x^*_τ je nejlepší jedinec z předchozí populace, x^*_t je nejlepší jedinec z aktuální populace, α je předem daná konstanta a $g()$ je akcelerační funkce. Funkce $g(x^*_t, x^*_\tau)$ může být gradient funkce $f(x)$ a nebo například může být lineární extrapolace. Počáteční populace je generována náhodně.

Vytváření nové generace s využitím řízené mutace

První krok je dekódování celé populace a její ohodnocení pomocí fitness funkce f. Poté je aplikována řízená mutace a nakonec jsou jedinci opět zakódováni. Dále proběhne selekce, křížení i mutace klasickým způsobem. Křížení je formou vícebodového křížení a selekce pouze vybírá jedince určené k reprodukci. Pravděpodobnost mutace je nastavena na velmi malé číslo a je to klasická mutace, která podle dané pravděpodobnosti změní hodnotu určitých genů z 0 na 1 nebo naopak. Na konec cyklu je do populace ještě přidán jedinec, který vznikl za pomoci řízené mutace.

4.2 Porovnání řízeného křížení a řízené mutace

Obě techniky využívají informace, které vyčetly z generací algoritmu. Řízená mutace najde nejlepší dva jedince ze dvou generací a vytvoří právě jednoho. Řízené křížení naopak vezme celou generaci a snaží se najít nejlepší změnu mezi dvěma jedinci, kterou aplikuje na větší počet jedinců v populaci.

Obě implementace využívají klasickou metodu mutace, ale implementace řízené mutace ještě navíc nakonec aplikuje vícebodové křížení.

Závěr

V práci jsme nejprve v první kapitole shrnuli známá fakta o GA, abychom mohli lépe nahlédnout do problematiky. V druhé kapitole jsme již navrhli konkrétní řešení řízeného křížení. Zavedli jsme řízené křížení, které dodržuje zadanou velikost populace (verze se zahazováním), a také řízené křížení, které populaci vždy rozšíří o nové jedince a nikdy žádného nezahodí (verze bez zahazování).

Ve třetí kapitole jsme blíže popsali implementaci klasického GA a Hill-climbingu. Popsané implementace jsme v téže kapitole využili k experimentům na úlohách počítání jedniček a binárního batohu. Tyto úlohy jsme zkoušeli pouštět na popsaných implementacích a implementaci řízeného křížení s různým počtem vytvořených generací, abychom zjistili efektivnost řízeného křížení oproti klasickému GA a Hill-climbingu. Ukázalo se, že řízené křížení sice zlepší výsledek oproti bodovému křížení, ale při stejném počtu generací pouze o málo a za cenu delšího času počítání křížení. V experimentech s binárním batohem se ukázal nejvíce efektivní algoritmus Hill-climbing, který svojí průměrnou nejlepší fitness výrazně převýšil jak bodové křížení, tak řízené. Všechny zdrojové texty experimentů a s nimi i uživatelskou a programátorskou dokumentaci lze najít na přiloženém CD.

Na závěr práce jsme ve čtvrté kapitole porovnali řízené křížení s podobnou nalezenou technikou.

Nápady na další vylepšení řízeného křížení

Ještě dodáme pár návrhů na možné vylepšení řízeného křížení:

- Křížení může porovnat množinu pár nejlepších jedinců se stejně velkou množinou nejhorších jedinců a možná tak najít změnu, která by byla efektivnější, než ta, kterou jsme počítali my.
- Vypočítanou nadějnou změnu zkoušet schválně aplikovat na nejlepší nebo naopak nejhorší jedince v populaci.
- Po řízeném křížení můžeme ještě zkusit pustit na populaci nějaký klasický operátor křížení (např. vícebodové křížení).
- U varianty řízeného křížení se zahazováním vymyslet jinou strategii vybírání jedinců, kteří mají být v populaci nahrazováni.

Seznam použité literatury

[1] Genetické algoritmy. http://www.sledujplanuj.cz/Clanek/10/Geneticke-algoritmy, April 2011.

[2] Bhandari, D.; Pal, N. R.; Pal, S. K.: Directed Mutation in Genetic Algorithms. *Information sciences 79 (3-4)*, 1994: s. 251–270.

[3] Garey, M. R.; Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[4] Goldberg, D. E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[5] Hynek, J.: *Genetické algoritmy a genetické programování*. Grada Publishing a.s., 2008.

[6] Šíma, J.; Neruda, R.: *Teoretické otázky neuronových sítí*. MATFYZPRESS, 1996.

[7] Mitchell, T. M.: *Machine Learning*. McGraw-Hill, 1997.

[8] Pošík, P.: *Paralelní genetické algoritmy*. Diplomová práce, České vysoké učení technické, 2001.

[9] Wilhelmstotter, F.: Jenetics: Java Genetic Algorithm Library Manual. http://sourceforge.net/projects/jenetics/.

[10] Wilhelmstotter, F.: Jenetics - Java Genetic Algorithm. http://jenetics.sourceforge.net/, 2012.

Seznam tabulek

Seznam použitých zkratek

GA - genetický algoritmus

Přílohy

A. Obsah přiloženého CD

Součástí této práce je CD obsahující tyto položky:

- Zdrojové soubory implementovaných programů
- Spustitelné soubory implementovaných programů
- Tato bakalářská práce ve formátu PDF (včetně programátorské a uživatelské dokumentace)

B. Programátorská dokumentace

Tato část práce nejprve popíše vlastní struktury použité v kódu a poté se věnuje konkrétní implementaci obou verzí řízeného křížení (verzi se zahazováním najdeme v **RizeneKrizeniSZ.java** a bez zahazování v **RizeneKrizeniBZ.java**) a Hill-climbingu (**HCAlterer.java**). Potřebné prerekvizity pro překlad programů z této dokumentace jsou: JDK 1.7 a vyšší a knihovna Jenetics [10].

B.1 Vlastní struktury

Nejprve si popíšeme struktury použité v našich zdrojových kódech, které neobsahovala knihovna Jenetics[10].

B.1.1 Alteration

Třída Alteration je použita jako struktura pro zaznamenání změny mezi jedinci. Stručně řečeno, Alteration zachycuje, které bity z genu změna neovlivňuje a které naopak nastavuje na hodnotu 1 nebo 0.

Proměnné Alteration

Struktura obsahuje:

Listing B.1: Obsah struktury Alteration

```
int[] _zmena;
List<int[]> _indexyNemennych;
double _fitness;
```

- **int[] zmena**: Je to pole znázorňující změnu mezi jedinci. Vytváření změny bylo obecně popsáno v kapitole 2.2.2 a předvedeno na příkladu v listingu 2.3.
- **List<int[]> _indexyNemennych**: Je arraylist dvojic, které mají mezi sebou změnu **zmena**.
- **double _fitness**: Fitness hodnota příslušné změny **zmena**. To je rozdíl fitness zúčastněných dvojic.

Metody Alteration

Následující funkce pouze vrací nebo přímo přepisují hodnoty proměnných Alteration (viz B.1.1):

- **getZmena()/setZmena()**: Vrátí/přepíše proměnnou **zmena**.
- **getIndexyNemennych()/setIndexyNemennych(List<int[]>)**: Vrátí/přepíše arraylist **_indexyNemennych**.
- **public double getFitness/setFitness(double)**: Vrátí/přepíše proměnnou **_fitness**.

- **insertToIndexyNemennych(int,int)**: Vezme přijatou dvojici intů a vloží ji do arraylistu **_indexyNemennych**.
- **containsInIndexyNemennych(int,int)**: Vyhledá, zda je vstupní dvojice již obsažena v **_indexyNemennych**.

B.2 Implementace řízeného křížení

Je implementována jak varianta se zahazováním (**RizeneKrizeniSZ.java**), tak bez zahazování (**RizeneKrizeniBZ.java**). Obě popíšeme zároveň, protože se liší jen v detailu, který bude dále popsán. Pokud se v textu mluví o **změně**, odkazujeme se na změnu popsanou v kapitole 2.2.2. Obě varianty řízeného křížení jsou definované **interfacem Alterer** (viz listingu B.2).

Listing B.2: Interface Alterer definovaný v knihovně jenetics

```
public interface Alterer<G extends Gene<?, G>> {
 public <C extends Comparable<? super C>> int alter(
  final Population<G, C> population,
  final int generation
        );
}
```

Rozhraní Alterer deklaruje metodu **alter()**, která se volá při vytváření nové generace. V této funkci nejdříve zavoláme **eachWithEach(population)** (viz dole), která porovná všechny jedince a vybere nejlepší změnu. Poté projde celou původní populaci a na základě pravděpodobnosti křížení vybírá jedince, na které aplikuje vybranou nejlepší změnuf voláním metody **applyChange()**.

O tom, jak se naloží s nově vzniklým jedincem ve funkci **applyChange()**, se rozhoduje podle verze řízeného křížení. Ve verzi bez zahazování se noví jedinci přidávají do původní populace (tj. populace, která byla předána jako parametr **alter()**) a ve verzi se zahazováním se nově vzniklý jedinec vymění v populaci za toho, ze kterého vznikl. Funkce **alter()** je definována v interface Alterer tak, aby vracela počet změněných jedinců (tuto hodnotu reprezentuje v naší implementaci proměnná int alterations).

B.2.1 Funkce eachWithEach(population)

eachWithEach(population) projde postupně všechny dvojice původní populace a pro každou vypočítá změnu. Změna je vypočítána za pomocí funkce **diff(jedinec,jedinec)**. Vypočítaná změna je spolu s údaji, o které jedince se jedná, a hodnotou rozdílu jejich fitness zapsána do instance třídy Alteration. Tyto nové instance jsou vkládány do hashovací mapy **zmeny**, odkud je nakonec vybrána změna s maximálním rozdílem fitness (pomocí **Collections.max()**). Důležitou součástí je využití faktu, že ke každé změně existuje změna přesně opačná (viz příklad B.3). Stačí nám tedy při vypočtení jedné změny vložit do HashMap **zmeny** i její opačnou (tím snížíme počet iterací při porovnávání jedinců).

Listing B.3: Ukázka opačné změny

```
Mějme 2 jedince:
1. jedinec je [1010] s fitness f = 2
2. jedinec je [1101] s fitness f = 3
Vypočítaná změna je [0,+1,−1,+1] s hodnotou +1.
Opačná změna vypadá takto:
Počítáme změnu z jedince 2. na 1. a výsledkem je:
[0,−1,+1,−1] s hodnotou −1
```

B.2.2 Funkce vytvářející testovací populaci

Součástí obou implementací řízeného křížení je i funkce **pretvorPopulaci(populace)**. Ta slouží pouze pro ladění a přepíše příchozí populaci na pevně danou. Pro její spuštění stačí odkomentovat první řádek funkce **alter()**.

B.3 Implementace Hill-climbing

Hill-climbing je implementovaný v **HCAlterer.java**. Algoritmus zachovává pravidla, která jsme si stanovili v kapitole 3.1.1. Pro provedení Hill-climbingu je vždy zavolána jeho vnitřní funkce **alter()**. Stejně jako u řízeného křížení, je tato třída definována **interfacem Alterer** (viz listing B.2).

Implementace funkce **alter()**: Začneme tím, že z celé vstupní populace vybereme v cyklu jedince, který má největší fitness. Potom vybraného jedince vezmeme a vygenerujeme pro něj sousední jedince tak, že vždy změníme jeden jeho bit. Ukázka generování sousedních jedinců je v listingu B.4 a konkrétní generování nového jedince popisuje listing B.5.

Listing B.4: Příklad vytvoření sousedních jedinců

```
Mějme jedince [0000]
Všichni jeho sousední jedinci:
[1000]
[0100]
[0010]
[0001]
```

Listing B.5: Konkrétní vytvoření nového jedince

```
MSeq<Chromosome<G>> chromosomes = chromosomesMax.copy();
//nová kopie chromozomů vybraného nejlepšího jedince

MSeq<G> genes = genesMax.copy();
//kopie sekvence genů nejlepšího jedince

genes.set(i, genes.get(i).newInstance());
//změní gen na vybrané pozici na opačný

chromosomes.set(0, chromosomes.get(0).newInstance(
```

```
genes.toISeq()));
//sekvence genů je zabalena do chromozomu

population.set(i, population.get(i).newInstance(
Genotype.valueOf(chromosomes.toISeq()), generation));
//na původní místo v populaci je vložen jedinec nový
```

Typ MSeq<> a Iseq<> v listingu B.5 je vlastní typ knihovny Jenetics a jeho popis se dá nalézt v její příručce[9]. Nová populace vznikne tak, že tu starou naplníme nejlepším jedincem a všemi jeho sousedy (což je v souladu popisu Hill-climbingu v této práci).

B.4 Pozměněné části knihovny Jenetics

B.4.1 BitGene.java

Byla pozměněna metoda **newInstance()**: Původně vracela kopii genu náhodné hodnoty (0 nebo 1) a teď vrací vždy opačnou hodnotu, než jakou má instance na které se **.newInstance()** zavolá. Funkce byla přetvořena kvůli vytváření nových jedinců používaném v řízeném křížení.

B.4.2 RKBZGeneticAlgorithm.java

Je to kopie GeneticAlgorithm.java, která ale po každé aplikaci **alter()** změní velikost populace podle toho, jak se změnila po křížení. Zatím má využití pouze v kombinaci s **RizeneKrizeniBZ.java**, která vrací populaci větší, než tu, kterou dostala parametrem.

B.4.3 MySelector.java

Selektor, který pouze vrátí tu populaci, kterou dostane na vstup. Selektor je využitý pro Hill-climbing, kde není potřeba vybírat jedince určené ke křížení.

B.5 Popis implementace příkladů

Všechny příklady vznikly rozšířením knihovny Jenetics. Pro více možností při tvorbě a úpravě příkladů doporučuji nahlédnout do příručky pro tuto knihovnu [9]. V této sekci popíšeme pouze zvláštnosti pro jednotlivé implementace. Některé základní příkazy potřebné k vytváření úloh jsou vypsány v listingu B.6

Listing B.6: Základní příkazy při vytváření úloh

```
Factory<Genotype<BitGene>> gtf = Genotype.valueOf(
  new BitChromosome(20, 0.15)
 );
//Takto zadáme jak mají vypadat chromozomy jedinců

Function<Genotype<BitGene>, Integer> ff
                                = new OneCounter();
```

```
//Přiřadíme fitness funkci

GeneticAlgorithm<BitGene, Integer> ga =
new GeneticAlgorithm<>(
        gtf, ff, Optimize.MAXIMUM
 );
//Vytvoříme instanci třídy, která obsluhuje běh GA

ga.setStatisticsCalculator(
  new NumberStatistics.Calculator<BitGene, Integer>()
 );
//Nastavení statistik

ga.setPopulationSize(50);
//Velikost populace

ga.setOffspringFraction(1.00); //tento řádek zajistí,
//aby nebyli žádní jedinci reprodukováni

ga.setSelectors(
  new RouletteWheelSelector<BitGene, Integer>()
 );
//Nastavení selektoru

ga.setAlterers(
  new Mutator<BitGene>(0.05),
  new SinglePointCrossover<BitGene>(0.10)
);
//Nastavení typu mutace a křížení spolu
//s jejich pravděpodobnostmi

ga.setup();
//Povinný řádek, který se stará o inicializaci algoritmu

ga.evolve(20);
//Do závorky se uvede počet generací,
//po které má algoritmus běžet

System.out.println(ga.getBestStatistics());
//Vytisknutí nejlepších statistik
```

B.5.1 Hill-climbing úlohy

Při tvorbě Hill-climbing úlohy musíme zajistit, aby délka chromozomů byla o 1 menší než délka populace (jak je uváděno v práci výše). Selektor nastavíme jako MySelektor (viz B.4.3), mutace není žádná a křížení je pomocí HCAlterer (B.3).

Listing B.7: Hill-climbing

```
Factory<Genotype<BitGene>> genotype = Genotype.valueOf(
  new BitChromosome(15, 0.5)
 );
...
ga.setPopulationSize(16); //velikost chromozomu + 1
...
ga.setOffspringSelector(
  new MySelector<BitGene, Float64>()
 );
ga.setSurvivorSelector(
  new RouletteWheelSelector<BitGene, Float64>()
 );

ga.setOffspringFraction(1.00);
...
```

B.5.2 Úlohy s řízeným křížením se zahazováním

Všechny změny oproti klasické úloze implementované v Jenetics jsou v listingu B.8. Selektor pro reprodukující jedince není důležitý, protože žádní nebudou.

Listing B.8: Úloha používající řízené křížení se zahazováním

```
...
ga.setOffspringSelector(
  new MySelector<BitGene, Integer>()
 );
ga.setSurvivorSelector(
  new RouletteWheelSelector<BitGene, Integer>()
 );

ga.setAlterers(
  new Mutator<BitGene>(0.05),
  new RizeneKrizeniSZ<BitGene>(0.1)
 );

ga.setOffspringFraction(1.00);
...
```

B.5.3 Úlohy s řízeným křížením bez zahazování

Úloha se vytvoří úplně stejně jako verze řízeného křížení se zahazováním (B.5.2) s jedinou změnou: místo klasického GeneticAlgorithm.java je zvolen upravený RKBZGeneticAlgorithm.java (viz B.4.2).

Listing B.9: Úloha používající řízené křížení bez zahazování

```
...
RKBZGeneticAlgorithm<BitGene, Integer> ga =
```

```
  new RKBZGeneticAlgorithm<>(
  gtf, ff, Optimize.MAXIMUM
 );
...
```

C. Uživatelská dokumentace

V této kapitole jsou popsány spustitelné soubory, které byly vytvořeny pro experimenty této bakalářské práce. Všechny najdete ve složce /**executable**. Pro spuštění našich programů potřebujete mít naistalovanou Javu 1.7 a novější.

C.1 Hill-climbing experimenty

Příklady na Hill-climbing implementaci jsou dva:

- **HCMyKnapsack.java**
- **HCMyOnesCounting.java**

Pouštějí se bez parametru, protože Hill-climbing nemá ani pravděpodobnost mutace ani křížení (příklad puštění je uveden v listingu C.1). Uvedené výpisy jsou v režii Jenetics a nás z nich zajímá pouze best fitness.

Listing C.1: Spuštění problému počítání jedniček pomocí Hill-climbing

```
$ java -jar HCMyOnesCounting.java

+-------------------------------------------------------------+
|  Population Statistics                                      |
+-------------------------------------------------------------+
|                    Age mean: 0,00000000000                  |
|                Age variance: 0,00000000000                  |
|                     Samples: 51                             |
|                Best fitness: 30                             |
|               Worst fitness: 28                             |
+-------------------------------------------------------------+
+-------------------------------------------------------------+
|  Fitness Statistics                                         |
+-------------------------------------------------------------+
|                Fitness mean: 28,84313725490                 |
|            Fitness variance: 0,97490196078                  |
|       Fitness error of mean: 4,03884706589                  |
+-------------------------------------------------------------+
```

C.2 Ostatní experimenty

Ostatní příklady už se mohou pouštět se dvěma parametry. Ten první je pravděpodobnost křížení a druhý je pravděpodobnost mutace (obě hodnoty musí být v rozsahu čísla s desetinnou čárkou od 0.00 až do 1.00). Pokud zadáme jeden parametr, bude jím pravděpodobnost křížení. Když není zadán parametr žádný, použijí se defaultní hodnoty v programu. Výčet spustitelných souborů:

- **MyKnapsack.java**: Klasická implementace binárního batohu

- **MyOnesCounting.java**: Klasická implementace počítání jedniček
- **RKBZMyKnapsack.java**: Implementace binárního batohu pomocí řízeného křížení bez zahazování
- **RKBZMyOnesCounting.java**: Implementace počítání jedniček pomocí řízeného křížení bez zahazování
- **RKSZMyKnapsack.java**: Implementace binárního batohu pomocí řízeného křížení se zahazováním
- **RKSZMyOnesCounting.java**: Implementace počítání jedniček pomocí řízeného křížení se zahazováním

Listing C.2: Ukázka spuštění implementace počítání jedniček pomocí řízeného křížení se zahazováním

```
$ java -jar RKSZMyOnesCounting.java
+-------------------------------------------------------+
|  Population Statistics                                |
+-------------------------------------------------------+
|                    Age mean: 0,26000000000            |
|                Age variance: 0,27795918367            |
|                     Samples: 50                       |
|                Best fitness: 20                       |
|               Worst fitness: 0                        |
+-------------------------------------------------------+

+-------------------------------------------------------+
|  Fitness Statistics                                   |
+-------------------------------------------------------+
|                Fitness mean: 11,14000000000           |
|            Fitness variance: 38,81673469388           |
|       Fitness error of mean: 1,57543390848            |
+-------------------------------------------------------+
```