KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

EVOLUČNÍ VÝPOČETNÍ TECHNIKY

MARTIN DOSTÁL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2007

Abstrakt

Bude

Cílová skupina

Bude

Obsah

1 Genetické programování 4
1.1 Reprezentace jedinců 4
1.1.1 Jazyk reprezentace 5
1.2 Genetické programování z hlediska evoluce 6
1.2.1 Generování výchozí populace 7
1.2.2 Účelová funkce - fitness 7
1.2.3 Rekombinační operátory 8
1.3 Schéma funkce standardního algoritmu pro genetické programování 11
1.4 Hierarchická dekompozice úlohy v genetickém programování 11
1.5 Měření výpočetní náročnosti experimentů 13
1.6 Aplikace genetického programování 14
1.7 Aplikace genetického programování na hledání funkce sudé parity 15
1.7.1 Sudá parita arity 3 15
1.7.2 Sudá parita arity 4 15
1.7.3 Sudá parita arity 5 17
1.8 Aplikace genetického programování s využitím ADF na funkce sudé parity 18
1.8.1 Sudá parita arity 4 19
1.8.2 Sudá parita arity 5 20
1.8.3 Sudá parita arity 6 21
1.9 Shrnutí 21
2 Kritický pohled na účelovou funkci v genetickém programování 23
3 Seznam obrázků 28
4 Seznam tabulek 29

1 Genetické programování

Genetické programování [12], [13], [18], [23], [24], [26] (dále GP) patří mezi evolučními technikami k nejmladším, avšak nejintenzivněji zkoumaným přístupům. GP vychází z analogie genetického algoritmu (dále GA) [33], [34], [35], [25] a liší se od něj účelem použití a způsobem reprezentace jedinců.

Genetické algoritmy jsou založeny na myšlence optimalizace hodnot nezávislých proměnných pomocí adaptačního algoritmu inspirovaného evolucí v živé přírodě [1], [2], [3] a [5]. Vhodně zakódované vzorky definičního oboru problému představují kandidáty na řešení (říkáme také *jedinci*, nebo *individua*). Jedincům je přiřazována míra kvality (tzv. *fitness*), kterou můžeme obecně chápat jako „chybu“, nebo vzdálenost od hledaného optima. Výpočet kvality provádí tzv. *fitness funkce*. Kandidáti na řešení tvoří *populaci*. Schopnost adaptace pomocí GA je založena na myšlence, že kombinací vhodných (kvalitních) jedinců získáme postupně lepší a lepší jedince a budeme se blížit k řešení daného problému - žádaným hodnotám nezávislých proměnných. Noví jedinci se vytvářejí na základě kombinace stávajících jedinců pomocí tzv. *rekombinačních operátorů*. Evoluce (adaptace) probíhá v tzv. *generacích*, rekombinací jedinců získáme nové jedince, kteří tvoří novou generaci. Aplikovatelnost a efektivita GP je závislá zejména na vhodné definici fitness funkce a rekombinačních operátorech.

Genetické programování se liší od genetických algoritmů především způsobem reprezentace jedinců. GP hledá řešení problému reprezentované programem (algoritmem), takže individua ze kterých je složena populace jsou programy. V GP jsou jedinci - programy obvykle reprezentovány grafy typu strom. Tento způsob reprezentace pochází z funkcionálních programovacích jazyků. Rekombinační operátory podobně jako u GA přetvářejí jedince v jiné, lépe adaptované jedince. Měření kvality jedinců je opět založeno na fitness funkci, která je chápána analogicky jako u GA. V GP tedy evoluci podléhají jedinci v populaci, mechanismus evoluce a reprezentace jedinců je statická.

Základní pojmy:

jedinec – je program, který je kandidátem na řešení a představuje prvek prohledávacího prostoru.

populace – je množina jedinců. Počáteční populace je obvykle vytvořena náhodným vygenerováním jedinců. Při generování jedinců jsou definovány omezující podmínky jako například hloubka stromu, vyvážení stromu a podobně.

generace – je aktuální populace jedinců.

genotyp – představuje genetickou výbavu jedince. V případě GA zakódovanou do lineárního řetězce, v případě GP zakódovanou pomocí reprezentace programu jako grafu typu strom.

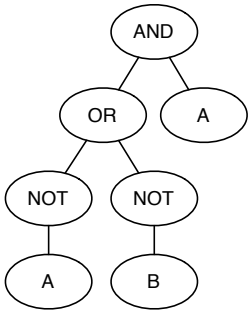
fenotyp – je interpretaci genotypu. Například v přírodě fentoyp určuje jak se konkrétní gen projeví (např. barva očí . . .), u GA fenotyp představuje dekódovaní genotypu (např. dekódování řetězce jako dekadického čísla), v GP je fentoyp reprezentován výsledkem interpretace genotypu - programu.

1.1 Reprezentace jedinců

Jedinci jsou reprezentováni pomocí grafu typu strom. Tato reprezentace je vhodná pro algoritmy z hlediska jejich snadné interpretace počítačem a zároveň je tato reprezentace díky jednoduchosti vhodná pro návrh a fungování rekombinačních operátorů. Klasické programovací jazyky, zejména jazyky imperativní, jako například C, nebo Pascal, jsou z hlediska jednoduchosti reprezentace méně vhodné, protože jejich syntax je složitější. Tento problém lze však vyřešit

pomocí převodu na stromovou strukturu kódování pomocí tzv. *Readova lineárního kódu*, viz [31]. Stromová reprezentace kódu však není v oblasti programovacích jazyků nová. Genetické programování přejalo reprezentaci z funkcionálního programovacího jazyka LISP, kde jsou programy i data reprezentovány jednotně pomocí datové struktury seznam, kterou lze chápat také jako graf typu strom. Jiné rysy funkcionálních programovacích jazyků však genetické programování nepřevzalo. Obecným rysem funkcionálních jazyků je mimořádně jednoduchá syntax založená na prefixové notaci, kde zjednodušeně řečeno, první prvek seznamu označuje funkci a zbývající prvky seznamu představují argumenty funkce, viz příklad. Převod takového zápisu na graf typu strom je jednoduchý: terminální symboly (proměnné a konstanty) představují listové (koncové) uzly stromu, a funkce[1] představují vnitřní uzly stromu.

Příklad: na obrázku 1 vidíme stromovou reprezentaci odpovídající programu `(AND (OR (NOT A) (NOT B)) A)`



Obrázek 1: Příklad stromové reprezentace

1.1.1 Jazyk reprezentace

GP reprezentuje kód jedinců pomocí *funkcí* a *terminálů*. Množina všech možných programů je tedy dána libovolnou konečnou rekurzivní kombinací funkcí z množiny $F = \{f_1, f_2 \dots f_n\}$ a terminálů z množiny $T = \{a_1, a_2 \dots a_m\}$. Každá funkce $f \in F$ má pevně danou *aritu*[2]. Množina funkcí obvykle obsahuje:

- aritmetické funkce (`+`, `-`, `*`, `/`)
- matematikcé funkce (`sin`, `cos`, `exp`, `abs`)
- boolovské logické funkce (`and`, `or`, `not`)
- podmíněné vyhodnocení (`if`)

Za předpokladu vhodné úpravy způsobu reprezentace kódu může množina funkcí také obsahovat funkce pro iteraci, popřípadě rekurzi. Terminály představují formální parametry funkcí, tedy proměnné, konstanty, nebo funkce, které nemají žádný vstupní argument. Množina funkcí se obvykle volí předem (ručně) tak, aby byla vhodná pro reprezentaci řešeného problému.

Takový způsob reprezentace kódu je jednoduchý, ale neumožňuje využít řadu dalších možností, jaké poskytují plnohodnotné (lidské) programovací jazyky, například modularitu a znovupoužití kódu. Obecné požadavky na podobu množiny funkcí a množiny terminálů v GP lze zformulovat do následujících bodů:

[1] Protože výsledkem vyhodnocení výrazu je opět výraz, mohou být na místě argumentů další výrazy představující volání funkce s argumenty.

[2] Počet vstupních argumentů.

uzavřenost – je stěžejním požadavkem na množinu funkcí a terminálů[3]. To znamená, že všechny použité funkce musí akceptovat jakýkoliv možný vstup tak, aby bylo zamezeno sémantickým chybám kódu. V důsledku je pak každý program vykonatelný a je považován na smysluplný. Příkladem je například tzv. *chráněné dělení* , kdy při pokusu o dělení nulou funkce vrátí výsledek 0. Výhodou tohoto přístupu je jednoduchost, nevýhodou je to, že se v populacích udržují jedinci jejichž kód je zbytečně dlouhý, prodloužený o různé části parazitního kódu, který je redundantní[4].

vyjádřitelnost – od zvolené množiny terminálů a funkcí přirozeně požadujeme, aby skládáním funkcí bylo možné dosáhnout výsledku – sestavit požadovaný program. Jiným vhodným označením tohoto požadavku by mohl být pojem *specifická úplnost*, tedy, že daná množina funkcí a terminálů je vzhledem k řešenému problému funkčně úplná.[5]

Příkladem specifické úplnosti vzhledem k boolovským funkcím je množina funkcí $F = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. Je obecně známo, že pokud původní množinu F zúžíme na $F' = \{\mathsf{AND}, \mathsf{NOT}\}$, stále máme specificky úplnou množinu vzhledem k boolovským funkcím, pokud bychom však odstranili z množiny F' funkci NOT, pak již specificky úplnou množinu nemáme.

obecnost – u genetického programování při volbě množiny funkcí zvažujeme, jaké funkce do jazyka zahrnout tak, aby byl výsledek s ohledem na použitou evoluční techniku snáze vyjádřitelný a případně čitelnější pro člověka. Uvážíme-li příklad z předchozího bodu, mohli bychom původní množinu F například rozšířit o funkce IF a NAND a dosáhnout tím rychlejší konvergence adaptačního procesu, protože takto upravená množina funkcí je z hlediska sémantiky programů řešících boolovské funkce silnější.

Poznámka: u řady aplikací genetického programování narazíme na problém, kdy není možné, anebo výhodné, zachovat požadavek na *uzavřenost* množiny funkcí a terminálů. Tento problém řeší genetické programování s typováním, které používá tzv. *typovou reprezentaci grafů*[6]. Typování je založeno na principu, kdy každému uzlu přiřadíme datový typ. Daný typ určuje, jaký typ dat vrací strom na svém výstupu, dále je nutné specifikovat počet, typ a pořadí argumentů daného uzlu. Rekombinační operátory mohou zpracovávat pouze kompatibilní uzly.

1.2 Genetické programování z hlediska evoluce

Proces postupné adaptace jedinců pomocí fitness funkce se v genetickém programování nazývá *evoluce*. Strojové pojetí evoluce v GP je volně inspirováno Darwinovým pojetím evoluce, které je, byť s různými výhradami, přijímáno jako vysvětlení vzniku a adaptace přírodních druhů v živé přírodě. Tento princip vychází z tvrzení, že lépe adaptovaní jedinci mají větší šanci v daném prostředí přežít a reprodukovat se. Genetická výbava jedinců se může odlišovat a k jejím změnám dochází převážně náhodně a v malé míře (různé pohledy na tuto problematiku poskytují například [1], [2], [3] a [5]). Nové generace v sobě akumulují genetické změny a to právě takové, které umožnily lépe se adaptovat, přežít a reprodukovat se[7].

V GP je genetická výbava jedince - *genotyp* reprezentována jeho kódem (programem). Interpretací genotypu je *fenotyp*, což je v našem případě výsledek vyhodnocení programu. Jedinci (programy) tvoří generaci a pomocí rekombinačních operátorů vytváříme jedince nové. Lépe

[3] Angl. Closure of The Function and Terminal Set.

[4] Porovnání délky a čitelnosti nalezených řešení téhož problému za použití různých systémů automatického programování je velmi zajímavé. Toto srovnání lze nalézt v dalších kapitolách.

[5] V [23], [24] se tato vlastnost označuje jako *Sufficiency*.

[6] Angl. Strongly Typed Genetic Programming.

[7] Evoluce probíhá pomalu stejně jako změny prostředí probíhají pomalu a postupně. Náhlá změna prostředí je v rozporu se schopností postupně se adaptovat, takže v takovém případě může dojít k náhlému vyhynutí druhu, podobně jako tomu bylo například u ještěrů v pravěku.

adaptovaní jedinci mají větší fitness a větší pravděpodobnost, že budou reprodukování do další generace a promítnou tak svůj genotyp do dalších potomků. Uveďme, že podstatným rozdílem proti evoluci v živé přírodě je to, že v GP je účel a cíl evoluce (adaptace) definován právě fitness funkcí, zatímco v živé přírodě evoluce probíhá podle změn prostředí a to trvale, postupně a bez cíle: *evoluce nemá žádný cílový, nebo koncový stav*. Jelikož je tato práce zaměřena zejména na hledisko reprezentace algoritmů, uvedeme rekombinační operátory s ohledem na obsažnost textu pouze přehledově. Pro podrobnější popis odkazujeme na [23] a [24].

1.2.1 Generování výchozí populace

Na počátku evoluce je třeba vytvořit výchozí populaci jedinců. Jedinci jsou vytváření pomocí náhodného generování grafu typu strom s vrcholy reprezentujícími příslušné funkce a terminály z množiny F a T. Vytvoření nového stromu lze dosáhnout mnoha způsoby, nejpoužívanější z nich uvedeme níže. Nejdelší cestu od kořene stromu k některému z koncových vrcholů budeme označovat jako *hloubku stromu*.

jednorázová metoda – tato metoda zajišťuje generování stromu tak, že délka cesty od kořene ke každému koncovému vrcholu je rovna hloubce stromu.

přírůstková metoda – generuje stromy různého tvaru. Délky cest od kořene ke koncovým vrcholům jsou různé, nejvýše však rovny hloubce stromu.

metoda „půl na půl“ – (*ramped half-and-half*) tato metoda je kombinací metod předešlých. Je založena na postupném vygenerování $\frac{100}{n-1}\%$ stromů délky x, kde $x \in < 2, n >$. U každého z těchto stromů se náhodně, se stejnou pravděpodobností, zvolí buď jednorázová nebo přírůstková metoda generování. Například, bude-li maximální hloubka $n = 5$, pak 25% stromů bude mít délku 2, 25% stromů bude mít délku 3, 25% stromů bude mít délku 4 a 25% stromů bude mít délku 5.

Podrobnější popis metod pro generování počáteční populace lze nalézt v [23].

1.2.2 Účelová funkce - fitness

Genetické programování pro měření kvality jedince používá obdobně jako genetické algoritmy tzv. *fitness funkci*. Tato funkce vyjadřuje kvalitu jedince a je stěžejním mechanismem navigace v prohledávacím prostoru směrem ke kvalitnějším jedincům. Fitness funkce je předem dánna programátorem a sama nepodléhá evoluci, narozdíl od tzv. *autokonstruktivní evoluce*, kde evoluci podléhají a samotné evoluční mechanismy. Měření kvality jedince je stěžejním principem pro zajištění konvergence algoritmu k přijatelnému výsledku. Zároveň je jedná o nejproblematičtější článek procesu adaptace v genetickém programování, protože je nutné zajistit objektivní ohodnocení jedinců, což je v případě algoritmů obtížný problém. Měření fitness lze provádět mnoha způsoby a podoba fitness funkce záleží na konkrétní aplikaci algoritmu. Téměř vždy se hodnota fitness přiřazuje každému jedinci v populaci. Základní přístupy k měření kvality jedinců jsou následující:

- sémantická analýza
- shoda na trénovací množině

Podrobněji a obecněji tuto problematiku rozebíráme v podkapitole 2 na straně 23. Základní způsoby určení a úpravy fitness v genetickém programování pocházejí z genetických algoritmů a jsou to tyto:

čistý fitness – *(raw fitness)* měření kvality jedince se obvykle provádí na *trénovací množině*, kterou si můžeme představit jako množinu tvořenou dvojicemi hodnot na vstupu a očekávané (správné) hodnoty na výstupu. Předpokládáme, že výsledkem vyhodnocení jedince je číslo, popř. pravdivostní hodnota. Výsledný fitness je pak určen následujícím vztahem:

$$r(i,t) = \sum_{j=1}^{N_e} |S(i,j) - C(j)| \tag{1.1}$$

kde t je číslo generace, $S(i,j)$ je výsledek vyhodnocení jedince i pro j-tý prvek trénovací množiny, která má N_e prvků. $C(j)$ je očekávaná hodnota výstupu pro vstup odpovídající j-tému prvku trénovací množiny. Například u boolovských funkcí odpovídá čistý fitness dosaženému počtu chyb na trénovací množině.

standardizovaný fitness – *(standardized fitness)* je přepočet čistého fitness na určitou referenční hodnotu fitness. Například, je-li čistý fitness jedince 45, maximální fitness (tedy ideální výsledek) je například 100, pak standardizovaný fitness je 100 - 45 = 55. Takže vztah pro standardizovaný fitness vypadá takto:

$$s(i,t) = r_{max} - r(i,t) \tag{1.2}$$

kde $r(i,t)$ je čistý fitness, a r_{max} je referenční hodnota fitness.

upravený fitness – *(adjusted fitness)* se používá pro převedení standardizovaného fitness na interval $< 0, 1 >$ podle tohoto vztahu:

$$a(i,t) = \frac{1}{1 + s(i,t)} \tag{1.3}$$

Hodnota 0 představuje ideálního jedince, 1 nejhoršího jedince.

normalizovaný fitness – (*normalized fitness*) se vypočte podle tohoto vztahu:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)} \tag{1.4}$$

a představuje „proporční“ přepočet fitness s těmito vlastnostmi:

1. $a(i,t) \in < 0, 1 >$
2. kvalitnější individua mají větší fitness
3. $\sum_{k=1}^{M} a(k,t) = 1$

1.2.3 Rekombinační operátory

Rekombinační operátory modifikují, resp. vytvářejí nové jedince. Jsou analogií rekombinačních operátorů používaných v genetických algoritmech. Nyní principy základních rekombinačních operátorů uvedeme, pro podrobný popis odkazujeme na [23] a [24]. Konkrétní podoba rekombinačních operátorů závisí na konkrétní řešené úloze.

reprodukce – operátor zajišťuje podle stanoveného mechanismu reprodukci vhodných jedinců do další generace. Realizace operátoru vychází z Darwinistického pojetí evoluce, kdy lépe adaptovaní jedinci mají větší schopnost se reprodukovat[8]. Jedinci jsou podle určitého kritéria vybíráni z aktuální populace a kopírování do nové populace, která vznikne reprodukcí

[8] Angl. Survival of the Fittest

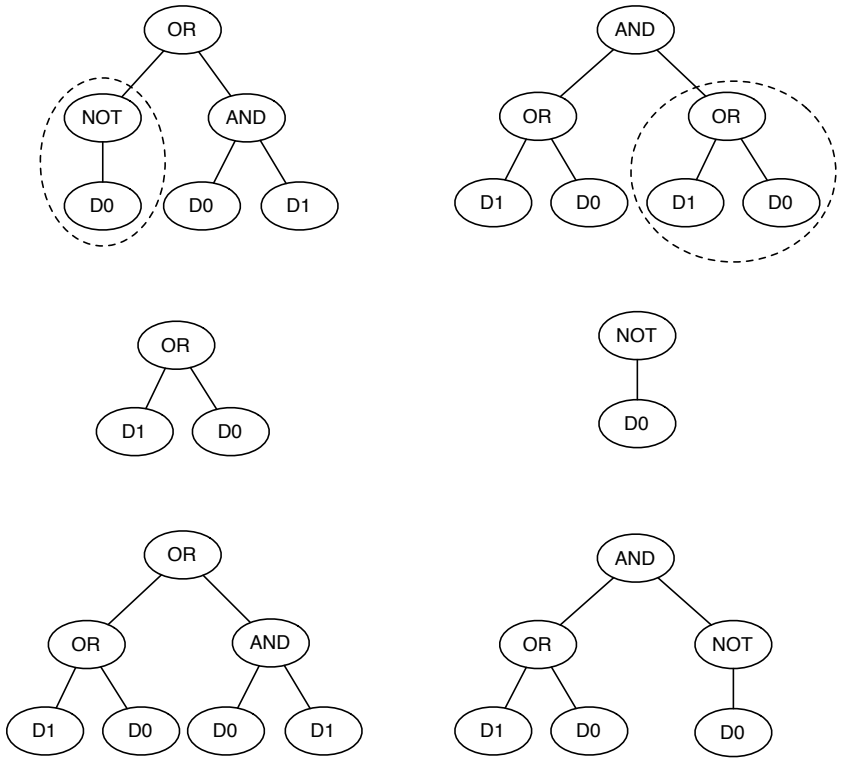
žádaného počtu jedinců[9]. Variant reprodukčních operátorů existuje velké množství, jejich základem je obvykle *proporční reprodukce*, která je určena jako pravděpodobnost p, že jedinec bude reprodukován do nové generace:

$$p = \frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t)} \tag{1.5}$$

kde $f(s_i(t))$ je normalizovaný fitness individua i v generaci t a populaci velikosti M. Platí tedy, že individuum má pravděpodobnost reprodukce odpovídající podílu jeho fitness na celkovém součtu fitness celé populace.

Druhým významným reprodukčním operátorem je tzv. *turnajová reprodukce* (*tournament selection*), která je založená na náhodném vybírání dvojic jedinců z populace, kdy je z dvojice reprodukován jedinec s vyšším fitness. Tento postup je obecně vhodnější pro zachování přiměřené diverzity v populaci. Toto je u řady úloh velmi důležité, jak známo z věty o schématech, (problematika schémat v GA a GP, věta o schématech viz [23] a [35]) počet schémat daného řádu roste exponenciálně, což může vést k předčasné konvergenci algoritmu, která se projevuje uváznutím v lokálním extrému.

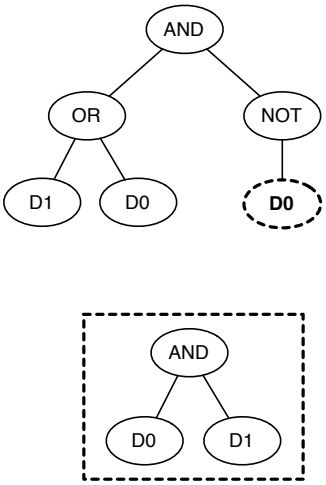
křížení – je základním rekombinačním operátorem. Křížení slouží k vytváření nových jedinců. Zkřížením dvou jedinců (rodiče) vzniknou dva noví jedinci - potomci[10]. Ve stromech reprezentující rodiče je vybrán uzel, na kterém jsou stromy odděleny a jejich části se navzájem nahradí - zkříží. Příklad křížení jedinců s kódem `(OR (NOT D0) (AND D0 D1))` a `(AND (OR D1 D0) (OR D1 D0))` ukazuje obrázek 2, jedinci vzniklí křížením jsou ve spodní části obrázku.



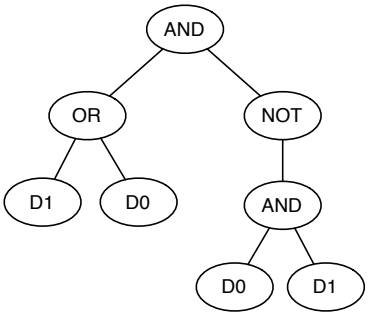
Obrázek 2: Křížení jedinců

[9] Velikost populace se obvykle během evoluce nemění.
[10] Angl. Offsprings.

mutace – tento operátor řadíme k základním rekombinačním operátorům. Operátor existuje v obrovském množství variant, běžně se setkáme s tím, že se jeho design liší problém od problému. Účelem operátoru mutace je provádět změny v individuích (podle funkce operátoru), jelikož operace křížení není schopná zavést do systému nové informace (pouze kombinace existujících individuí). Mutace je obvyklé náhodná a její vliv na populaci je poměrně malý[11]. Přesto se jedná o důležitý operátor, který zlepšuje diverzitu populace a při vhodném návrhu a nastavení snižuje riziko uváznutí evoluce v lokálním extrému. Tuto problematiku zde hlouběji nerozvádíme, podrobněji viz [33] a [35]. Příliš vysoký vliv mutace může způsobit rozkolísání a nestabilitu algoritmu. Základní princip fungování mutace je takový, že náhodně „odstřihne“ některou z větví individua a nahradí ji novou opět náhodně vygenerovanou větví, viz obrázky 3 a 4, které ukazují příklad mutace na koncovém vrcholu `D0`, kde je tento výraz nahrazen výrazem `(AND D0 D1)`.



Obrázek 3: Mutace kódu jedince



Obrázek 4: Jedinec po mutaci

permutace – provádí permutaci pořadí argumentů funkce (koncových listů stromu).

editace – slouží pro zjednodušení kódu programů pomocí transformačních pravidel, pokud kód obsahuje redukovatelné části. Například je li funkce aplikována na konstanty, je možné její výsledek uvažovat jako konstantu, v jiných případech lze některé výrazy zjednodušit pomocí logických pravidel.

[11] Přesto řádově vyšší než v živé přírodě, kde se obvykle pravděpodobnost mutace pohybuje v mezích tisícin procenta, zatímco v GP je vliv v řádu procent.

Příklad redukčních pravidel:
`(AND X X)` → `X`
`(OR X X)` → `X`
`(NOT (NOT X) X)` → `X`
`(IF T 1 5)` → `1`
`(* 2 5)` → `10`

Opakovanou aplikací pravidel se mohou některé výrazy podstatně zjednodušit:
`(NOT (NOT (NOT (NOT (OR (OR X X) X)))))` → `X`

1.3 Schéma funkce standardního algoritmu pro genetické programování

Zde uvádíme základní schéma funkce algoritmu genetického programování, pro podrobnější popis odkazujeme na [23], [24].

Algoritmus:

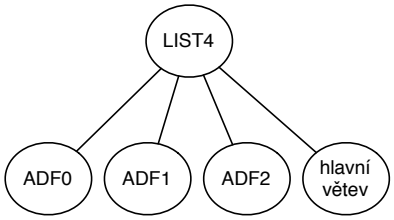
1. vygeneruj výchozí populaci o velikosti N
2. je-li splněna podmínka pro ukončení algoritmu, vrať výsledek, jinak pokračuj
3. proveď ohodnocení jedinců v populaci
4. proveď reprodukci
5. kroky 5 až 8 proveď $\frac{N}{2}$-krát:
6. vyber dva jedince z populace.
7. proveď křížení jedinců a potomky zařaď do populace.
8. proveď mutaci potomků.
9. aplikuj ostatní evoluční operátory.
10. jdi na krok 2.

1.4 Hierarchická dekompozice úlohy v genetickém programování

Z dosavadního popisu genetického programování je patrné, že je nutné předem znát množinu funkcí a terminálů z jakých lze hledanou funkci zkonstruovat. Tato množina se během experimentu nemůže změnit, takže v základním modelu genetického programování není žádný nástroj pro rozdělení úlohy na hierarchicky vyšší celky, než jsou funkce obsažené v množině funkcí. Povaha mnoha reálných algoritmů je taková, že v nich lze identifikovat části kódu, které se v řešení dané úlohy opakují a mohly by tvořit logický podcelek hledané funkce. To by přirozeně mohlo snížit nároky pro nalezení řešení dané úlohy, protože můžeme jistou část kódu, o které lze předpokládat, že je právě takovým podcelkem, „fixovat“ pomocí funkce, která je součásti množiny funkcí. Toto genetické programování řeší pomocí tzv. *Automaticky definovaných funkcí* (ADF). Myšlenka ADF je taková, že uživatel předem množinu funkcí obohatí o pomocné funkce, jejichž struktura je předem pevně dána, a tyto funkce mohou jednotlivá individua používat (volat je) v rámci svého kódu. Během evoluce se tedy kromě kódu jedinců vytváří také kód pomocných funkcí.

Automaticky definované funkce nejsou funkcemi v tom smyslu jak je známe z programovacích jazyků. Ve skutečnosti se jedná o bloky (větve) kódu, které jsou součástí individua a tzv.

hlavní větev může obsahovat „volání“ pomocných funkcí. To znamená, že pomocné funkce jsou svázány s individuem a tak každé individuum může mít pomocné funkce definovány jinak. Struktura pomocných funkcí je však stejná, protože je definována předem a je neměnná. Strukturu individua s podporou ADF naznačuje obrázek 5. Kořen stromu vždy obsahuje výraz `LIST`n, kde n označuje počet větví výrazu individua. Poslední větev (ta nejvíce vpravo) představuje hlavní větev, takže výraz s kořenem `LIST`n obsahuje $n-1$ pomocných funkcí. Každá větev má svou množinu terminálů, pomocné funkce mají množinu funkcí stejnou s výchozí množinou, hlavní větev má množinu funkcí rozšířenou o příslušné pomocné funkce. Pomocné funkce tedy nemohou ve svém těle obsahovat volání ostatních pomocných funkcí.



Obrázek 5: Struktura jedince s ADF

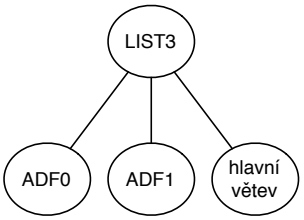
Aby došlo ke zvýšení efektivity evoluce (snížení potřebného počtu kroků) ve srovnání s případem evoluce bez použití ADF, předpokládá se, že ADF obsahují části kódu užitečné pro řešení účelové funkce a tyto jsou opakovaně použity (volány) v hlavní větvi. Nyní uvedeme příklad jedince se dvěma pomocnými funkcemi, viz obrázky 6 a 7, odpovídající kód je uveden níže. Budeme požívat dvě pomocné funkce `ADF0` a `ADF1`.

- kořen stromu je výraz `LIST3`
- `LIST3` se nevyskytuje jinde než v kořenu stromu
- levá větev představuje definici první pomocné funkce `ADF0`
- prostřední větev představuje definici druhé pomocné funkce `ADF1`
- pravá větev (hlavní funkce) představuje tělo jedince, které je interpretováno (vyhodnocováno) a může využívat pomocných funkcí `ADF0` a `ADF1`
- množina funkcí $F = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$
- množina funkcí pomocné funkce `ADF0`: $F_0 = F$
- množina terminálů `ADF0`: $T_0 = \{\mathsf{ARG0}, \mathsf{ARG1}\}$
- množina funkcí pomocné funkce `ARG1`: $F_1 = F$
- množina terminálů `ADF1`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}\}$
- množina funkcí hlavní funkce: $F_m = F \cup \{\mathsf{ADF0}, \mathsf{ADF1}\}$
- množina terminálů hlavní funkce: $T_m = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}\}$

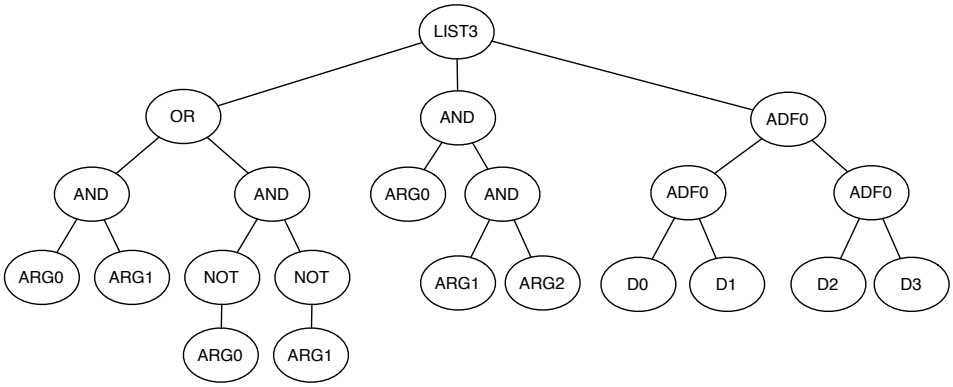
Kód odpovídající obrázku 7:

```
(LIST3 (OR (AND ARG0 ARG1) (AND (NOT ARG0) (NOT ARG1)))
       (AND ARG0 (AND (ARG1 ARG2)))
       (ADF0 (ADF0 D0 D1) (ADF0 D2 D3)))
```

Výsledky aplikace automaticky definovaných funkcí a srovnání se standardním genetickým programováním uvádíme v samostatné kapitole 1.8.



Obrázek 6: Struktura jedince se dvěma pomocnými funkcemi



Obrázek 7: Kód jedince

1.5 Měření výpočetní náročnosti experimentů

Základním ukazatelem výkonu a možností systémů pro automatické programování je průměrný počet programů, které je nutno projít než nalezneme řešení s určitou pravděpodobností. Výsledek se vyjadřuje pomocí čísla $I(M, i, z)$, které představuje průměrný počet programů (individuí), které je nutné zpracovat, abychom nalezli řešení daného problému s pravděpodobností alespoň z %:

$$I(M, i, z) = M * (i + 1) * R(z) \tag{1.6}$$

$$R(z) = \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil \tag{1.7}$$

kde M je velikost populace, i je počet generací a $R(z)$ je počet nutných opakování experimentu abychom v generaci i získali výsledek s pravděpodobností alespoň z. $P(M, i)$ je kumulativní pravděpodobnost , že bude nalezeno řešení nejvýše v generaci i. $P(M, i)$ s počtem generací roste. Tento způsob měření pochází z genetického programování [23], je však definován dostatečně obecně na to, aby byl použitelný i v systémech automatického programování založených na jiných principech než je genetické programování, včetně například takových systémů, které nepoužívají populace. Tento způsob měření výpočetní náročnosti je také použit u experimentů s jazykem Push a budeme jej používat také při měření výpočetní náročnosti experimentů v jazyce FSM.

Příklad výpočtu $I(M, i, z)$, pro $z = 0.99$ (hledáme výsledek s pravděpodobností alespoň 99 %), při velikosti populace $M = 1500$ v tabulce 1.

i	$P(M,i)$	$R(z)$	$I(M,i,z)$
25	0.08	56	2 184 000
50	0.19	22	1 683 000
100	0.44	8	1 212 000
150	0.69	4	906 000
200	0.76	4	1 206 000

Tabulka 1: Výpočet $I(M,i,z)$

1.6 Aplikace genetického programování

Genetické programování bylo úspěšně aplikováno v celé řadě oblastí. Hlavní oblasti aplikace uvádíme včetně odkazů na příslušnou literaturu.

symbolická regrese – je proces kdy pomocí adaptivního algoritmu (GP) hledáme symbolický předpis funkce podle trénovací množiny. To znamená, že hledáme takovou funkci (program), který pro zadané vstupy bude vracet žádané hodnoty na výstupu. GP bylo poměrně úspěšně aplikováno na symbolickou regresi (většinou jednoduchých) problémů jako například hledání trigonometrických funkcí, boolovských funkcí, indukce řad a podobně. Podrobnosti o experimentech a výsledcích viz [23] a [24]. Symbolická regrese je jednoznačně nejčastější aplikací GP a zřejmě obecně nejčastější aplikací systémů automatického programování.

automatizovaný návrh a optimalizace elektronických obvodů – zde bylo GP aplikováno na elektronické obvody navržené člověkem, pomocí GP se podařilo dosáhnout lepších elektrických vlastností zařízení po optimalizaci obvodu, viz [19].

robotika – GP bylo úspěšně použito například při navigování robota se subsumpční architekturou pro pohyb podél zdi [16], [17] a [23]. Jiným příkladem je experiment s hledáním programu s emergentním účinkem v kolonii umělých robotů, viz [20], [22].

optimalizace ve strojírenství – zde jsou například známy optimalizace návrhu mechanické konstrukce motorů.

ekonometrie – genetické programování bylo použito ve třech základních oblastech dotýkajících se ekonomie. První oblastí je tzv. ověřování hypotéz pomocí empirických dat (law rediscovery). Zde uvažujeme hypotézu, která představuje funkční předpis. Pomocí symbolické regrese se snažíme najít takovou funkci, která odpovídá empirickým trénovacím datům a tím dostatečně-krát opakovaným experimentem hypotézu ověříme, nebo vyvrátíme. V [23] je uveden experiment založený na ověření Ohmova zákona, nebo třetího Keplerova zákona. Článek [14] uvádí experiment, kde je ověřena makroekonomická rovnice směny kvantitativní teorie peněz (podrobněji viz [4]). Jako empirická data jsou uvažovány makroekonomické údaje ekonomiky USA v rozsahu 30-ti let.

Chen v [6] a [7] uvádí aplikaci genetického programování na ověření hypotézy efektivního trhu. Také je známa řada experimentů s umělými ekonomikami aplikovanými v multiagentových systémech. Genetické programování bylo také použito pro modelování ekonomických procesů, například spekulací na trhu [8], nebo modelování chování firem na trhu [10].

Několik publikací je věnováno aplikacím genetického programování pro účely predikce makroekonomických veličin v čase, například [14] uvádí predikci deflátoru HDP, nebo Kaboudan [11] použil genetické programování pro predikci poptávky po plynu v USA, Chen v [9] uvádí aplikaci genetického programování na predikci vývoje inflace.

generátory náhodných čísel – principem optimalizace algoritmem GP bylo nalézt vhodný funkční předpis pro generátor náhodných čísel, který na vstup kontinuálně dostává hodnoty od 0 do 16384. Generátor náhodných čísel založený na genetickém programování

dosáhl výsledků srovnatelných se specializovanými algoritmy pro generování náhodných čísel (například Park-Miller, IBM-RANDU, TI-random, SHUFFLE), podrobný popis realizace úlohy viz [15].

1.7 Aplikace genetického programování na hledání funkce sudé parity

Nejtypičtější aplikaci genetického programování v oblasti symbolické regrese je hledání funkce pro výpočet sudé parity a různém počtu argumentů. Výsledky a parametry algoritmu genetického programování přebíráme z [23] a uvádíme je zejména ze srovnávacích důvodů .

- množina funkcí $F = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NAND}, \mathsf{NOR}\}$. Je výpočetně úplná a tedy použitelná pro libovolnou úlohu symbolické regrese.
- velikost populace $M = 4000$
- maximální počet generací je 51
- testují se všechny kombinace vstupů, testovacích případů je tedy 2^k, kde k je arita funkce sudé parity
- fitness je počet špatně vyhodnocených vstupů
- ukončovací podmínka je $fitness = 0$

1.7.1 Sudá parita arity 3

Množina terminálů pro sudou paritu 3 argumentů: $T_3 = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}\}$.

Příklad nalezeného řešení:

```
(AND (OR (OR D0 (NOR D2 D1))
         (AND (NAND
               (NOR (NOR D0 D2) (AND (AND D1 D1) D1))
               (NAND (OR (AND D0 D1) D2) D0))
              (OR
               (NAND
                (AND D0 D2)
                (OR (NOR (OR D2 D0)) D1))
               (NAND (NAND D1 (NAND D0 D1)) D2)))))
```

U experimentů symbolické regrese je obvyklé, že udáváme počet jedinců, které je potřeba projít tak, abychom nalezli řešení problému s pravděpodobností alespoň 99% (viz 1.5), tedy $I(M, i, 0.99)$. Po 66-krát opakovaém experimentu jsme nalezli řešení v 9. generaci v 91 % případů. Pro přepočet na pravděpodobnost 99 % je použito $R(z)$, kde $z = 0.99$ a $P(M, i) = 0.91$ (viz 1.5). $R(0.99)$ vyjde 2. Takže $I(M, i, 0.99) = 4000 * (9 + 1) * 2 = 80000$.

1.7.2 Sudá parita arity 4

Pro hledání sudé parity 4 argumentů je nutné množinu terminálů rozšířit o jeden vstupní argument, takže $T_4 = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}\}$, jinak jsou podmínky experimentu stejné jako v předchozím experimentu.

Příklad nalezeného řešení:

```
(AND (OR (OR (OR (NOR D0 (NOR D2 D1))
                 (NAND
                  (OR
                   (NOR (AND D3 D0) D2)
                   (NAND
                    D0
                    (NOR D2 (AND D1 (OR D3 D2)))))
                  D3))
             (AND (AND D1 D2) D0))
         (NAND
          (NAND
           (NAND
            D3
            (OR (NOR D0 (NOR (OR D3 D2) D2))
                (NAND
                 (AND (AND (AND D3 D2) D3) D2)
                 D3)))
           (NAND
            (OR (NAND (OR D0 (OR D0 D1)) (NAND D0 D1))
                D3)
            (NAND D1 D3)))
          D3))
     (OR (OR (NOR
              (NOR
               (AND
                (OR
                 (NOR D3 D0)
                 (NOR
                  (NOR
                   D3
                   (NAND (OR (NAND D2 D2) D2) D2))
                  (AND D3 D2)))
                D1)
               (AND D3 D0))
              (NOR D3 (OR D0 D2)))
             (NOR
              D1
              (AND
               (OR
                (NOR (AND D3 D3) D2)
                (NAND D0 (NOR D2 (AND D1 D0))))
               (OR
                (OR D0 D3)
                (NOR
                 D0
                 (NAND (DR (NAND D2 D2) D2) D2))))))
         (AND (AND
               D2
               (NAND
                D1
                (NAND
                 (AND D3 (NAND D1 D3))
```

```
                (AND D1 D1))))
            (OR D3 (OR D0 (OR D0 D1))))))
```

Po 60-krát opakovaném experimentu bylo nalezeno řešení po 28 generacích s pravděpodobností 35 % a v generaci 50 s pravděpodobností 45 %. Pak $R(0.99) = 11$ a $I(M, i, 0.99) = 4000 * (28 + 1) * 11 = 1\,276\,000$.

1.7.3 Sudá parita arity 5

Pro funkci sudé parity 5-ti argumentů jsem množinu terminálů rozšířili o další terminál, takže $T_5 = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}, \mathsf{D4}\}$. Protože pro velikost populace $M = 4000$ nebylo během 20-krát opakovaného experimentu nalezeno žádné řešení, byla velikost populace zdvojnásobena na $M = 8000$. Při tomto nastavení se $I(M, i, 0.99) = 7\,840\,000$.

Příklad nalezeného řešení:

```
(NAND
 (NAND
  (OR (AND (AND (AND (OR D2 D3) (OR D4 D2))
                (AND (OR D4 D0) (AND D1 D1)))
           (NOR
            (NOR D1 D4)
            (OR (NOR D3 D4) (NOR D0 D2))))
      (NOR
       (NOR
        (NAND (AND (OR D1 D0) (OR D3 D2)) (OR D1 D3))
        (NOR D1 D2))
       (NOR
        (AND (NAND D4 D3) (NAND D4 D0))
        (NAND (OR D2 D4) (OR D2 D1)))))
  (NOR
   (NOR
    (OR (NOR (AND D2 D0) (NOR D1 D4))
        (AND (NOR
              (NOR
               (OR
                (NOR (AND D2 D0) (NOR D1 D4))
                (AND (NOR D0 D2) (NAND D4 D0)))
               (AND
                (AND (NAND D4 D3) (OR D3 D0))
                (OR D4 D3)))
              (NOR
               (NOR
                (AND (AND D1 D1) (AND D4 D2))
                (NAND (NAND D0 D2) (NAND D4 D0)))
               (NAND
                (AND D0 D4)
                (NAND (NOR D1 D4) (OR D1 D0)))))
             (NAND (AND D4 D1) (OR D2 D0))))
    (AND (AND (NAND D4 D3) (OR D3 D0)) (OR D2 D1)))
   (NOR
    (NOR
     (AND (AND D1 D1)
          (NOR
```

```
           (NAND (NAND D4 D2) (NAND D4 D4))
           (OR (AND D2 D0) (AND D4 D1))))
    (NAND
     (OR (AND (OR
                (AND D3 D0)
                (OR D4 (NAND (OR (NOR D1 2) D3) D4)))
               (NAND (NAND D0 D1) (NAND D2 D2)))
         (NAND
          (AND (NOR D0 D1) (OR D3 D4))
          (OR (NAND D3 D4) (AND D3 D1))))
     (NAND D4 (AND (OR D1 D0) (OR D3 D2)))))
   (NAND (OR (NAND D1 D0) (NOR D2 D0)) (NAND D3 D2)))))
(NAND
 (AND (NAND
       (OR (NOR
            (NAND (OR D1 D3) (AND D0 D4))
            (NOR (AND D1 D4) (NOR D2 D2)))
           (AND (OR
                 (NOR D1 D3)
                 (NOR
                  (AND
                   (NOR D2 D2)
                   (NOR (NOR D2 D2) (AND D2 D1)))
                  (AND
                   (NAND
                    (NAND D4 D0)
                    (NAND
                     (NOR
                      (NAND (NOR D4 D4) (NOR D0 D4))
                      (OR (AND D2 D3) (AND D4 D1)))
                     (AND
                      (OR (NOR D3 D4) D1)
                      (AND (OR D1 D0) (OR D3 D2)))))
                   (OR D2 D3))))
                (OR (OR D2 D3) (NAND D3 D0))))
       (NAND
        (AND (NOR (AND D0 D2) (OR D4 D0))
             (AND D4 D1))
        (OR (AND D1 D4)
            (NAND (NAND D1 D3) (OR D3 D1)))))
      (OR (OR D2 D3) (NAND D3 D0)))
 (AND (OR (NOR D3 D4) D3)
      (AND (OR D1 D0) (OR D3 D2)))))
```

1.8 Aplikace genetického programování s využitím ADF na funkce sudé parity

Automaticky definované funkce (ADF) mohou přinést efektivnější řešení některých problémů symbolické regrese pomocí definování funkčních bloků. Struktura takovýchto bloků musí být předem definována. Nyní uvedeme příklad takovéto aplikace na funkce sudé parity pro 4 až 6 vstupních argumentů a výsledky porovnáme s předchozími experimenty.

Nejprve uvedeme obecné parametry algoritmu:

- velikost populace $M = 4000$

- maximální počet generací 51
- testovací případy: všechny kombinace vstupů
- $fitness = 2^k - x$, kde k je počet vstupních argumentů a x je počet správně vyhodnocených vstupů
- ukončovací podmínka: $fitness = 2^k$
- množina funkcí $F = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NAND}, \mathsf{NOR}\}$

1.8.1 Sudá parita arity 4

Pro hledání řešení funkce sudé parity 4 argumentů budeme využívat dvě pomocné funkce: `ADF0` a `ADF1`, první bude funkce dvou argumentů, druhá bude funkce tří argumentů. Pro aplikaci ADF je nutné předdefinovat strukturu (a omezující podmínky) výrazu obsahujícího definice pomocných funkcí a strukturu pomocných funkcí (viz 1.4). Podmínky jsou následující:

- kořen stromu je výraz `LIST3`
- `LIST3` se nevyskytuje jinde než v kořenu stromu
- levá větev představuje definici první pomocné funkce `ADF0`
- prostřední větev představuje definici druhé pomocné funkce `ADF1`
- pravá větev (hlavní funkce) představuje tělo jedince, které je interpretováno (vyhodnocováno) a může využívat pomocných funkcí `ADF0` a `ADF1`
- množina funkcí pomocné funkce `ADF0`: $F_0 = F$
- množina terminálů `ADF0`: $T_0 = \{\mathsf{ARG0}, \mathsf{ARG1}\}$
- množina funkcí pomocné funkce `ADF1`: $F_1 = F$
- množina terminálů `ADF1`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}\}$
- množina funkcí hlavní funkce: $F_m = F \cup \{\mathsf{ADF0}, \mathsf{ADF1}\}$
- množina terminálů hlavní funkce: $T_m = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}\}$

Při 168-krát opakovaném experimentu, bylo nalezeno řešení sudé parity 4 argumentů v 93 % případů nejpozději v 9. generaci a v 99 % případů nejpozději v 50. generaci. Pak $I(M, i, z) = 4000 * (9 + 1) * 2 = 80\,000$.

Příklad nalezeného řešení:

```
(LIST3
 (OR (AND ARGO ARG1) (NOR ARG1 ARGO))
 (NOR (AND ARG2 ARGO) (NOR (OR ARGO ARGO) ARG2))
 (ADF1
  (ADFO D1 DO)
  (AND (OR (ADF1 D1 D2 DO) (OR D2 D1))
       (OR (AND DO DO) (AND D3 DO)))
  (ADF1 D3 DO D2)))
```

1.8.2 Sudá parita arity 5

Funkce sudé parity 5-ti argumentů bude využívat tři pomocné funkce. Podmínky pro strukturu výrazu reprezentujícího kód jedince jsou tyto:

- kořen stromu je výraz `LIST4`
- `LIST4` se nevyskytuje jinde než v kořenu stromu
- levá větev představuje definici první pomocné funkce `ADF0`
- druhá větev zleva představuje definici druhé pomocné funkce `ADF1`
- třetí větev zleva představuje definici druhé pomocné funkce `ADF2`
- pravá větev (hlavní funkce) představuje tělo jedince, které je interpretováno (vyhodnocováno) a může využívat pomocných funkcí `ADF0`, `ADF1`, `ADF2`
- množina funkcí pomocné funkce `ADF0`: $F_0 = F$
- množina terminálů `ADF0`: $T_0 = \{\mathsf{ARG0}, \mathsf{ARG1}\}$
- množina funkcí pomocné funkce `ADF1`: $F_1 = F$
- množina terminálů `ADF1`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}\}$
- množina funkcí pomocné funkce `ADF2`: $F_1 = F$
- množina terminálů `ADF2`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}, \mathsf{ARG3}\}$
- množina funkcí hlavní funkce: $F_m = F \cup \{\mathsf{ADF0}, \mathsf{ADF1}, \mathsf{ADF2}\}$
- množina terminálů hlavní funkce: $T_m = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}, \mathsf{D4}\}$

Pro funkcí sudé parity 5-ti argumentů bylo při 7-krát opakovaném experimentu nalezeno řešení ve 100 % případů v generaci 37. Pak $I(M, i, z) = 4000 * (37 + 1) * 1 = 152\,000$.

Příklad nalezeného řešení:

```
(LIST4
 (NAND
  (OR (OR (NAND ARGO ARG1) (NOR ARGO ARG1))
      (AND (NOR ARGO ARG1) (AND ARGO ARG1)))
  (NAND
   (AND (NAND ARG1 ARGO) (NOR ARG1 ARG1))
   (NOR (OR ARGO ARGO) (OR ARG1 ARGO))))
 (AND (NAND ARG2 ARGO) (OR ARGO ARG2))
 (OR (NAND
      (NAND ARGO ARG2)
      (OR ARG2 ARG1)
      (AND (AND ARG1 ARGO) (NAND ARG2 ARGO))
      (ADFO
       (NAND (ADF1 D1 D4 D4) (ADF1 D1 D4 D4))
       (ADF1 (ADFO D2 DO) (NOR D2 DO) (AND D3 D3))))))
```

1.8.3 Sudá parita arity 6

Analogický postup jako u předchozích funkcí je zvolen pro sudou paritu 6-ti vstupních argumentů: pomocné funkce budou nyní čtyři a podmínky pro strukturu výrazu jsou tyto:

- kořen stromu je výraz `LIST5`
- `LIST5` se nevyskytuje jinde než v kořenu stromu
- první (levá) větev představuje definici první pomocné funkce `ADF0`
- druhá větev zleva představuje definici druhé pomocné funkce `ADF1`
- třetí větev zleva představuje definici druhé pomocné funkce `ADF2`
- čtvrtá větev zleva představuje definici druhé pomocné funkce `ADF3`
- pravá větev (hlavní funkce) představuje tělo jedince, které je interpretováno (vyhodnocováno) a může využívat pomocných funkcí `ADF0`, `ADF1`, `ADF2` a `ADF3`
- množina funkcí pomocné funkce `ADF0`: $F_0 = F$
- množina terminálů `ADF0`: $T_0 = \{\mathsf{ARG0}, \mathsf{ARG1}\}$
- množina funkcí pomocné funkce `ADF1`: $F_1 = F$
- množina terminálů `ADF1`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}\}$
- množina funkcí pomocné funkce `ADF2`: $F_1 = F$
- množina terminálů `ADF2`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}, \mathsf{ARG3}\}$
- množina funkcí pomocné funkce `ADF3`: $F_1 = F$
- množina terminálů `ADF3`: $T_1 = \{\mathsf{ARG0}, \mathsf{ARG1}, \mathsf{ARG2}, \mathsf{ARG3}, \mathsf{ARG4}\}$
- množina funkcí hlavní funkce: $F_m = F \cup \{\mathsf{ADF0}, \mathsf{ADF1}, \mathsf{ADF2}, \mathsf{ADF3}\}$
- množina terminálů hlavní funkce: $T_m = \{\mathsf{D0}, \mathsf{D1}, \mathsf{D2}, \mathsf{D3}, \mathsf{D4}, \mathsf{D5}\}$

Pro sudou paritu 6-ti vstupních argumentů není v [23] $I(M, i, z)$ uvedeno. Příklad řešení funkce sudé parity pro 6 argumentů z důvodu rozsáhlosti neuvádíme, lze ho nalézt v [23]. Tabulka 2 ukazuje, že použítm ADF došlo k výraznému snížení výpočetní náročnosti.

systém	parita 3	parita 4	parita 5
GP	80 000	1 276 00	7 840 000
GP(ADF)		80 000	152 000

Tabulka 2: Srovnání výsledků sudé parity

1.9 Shrnutí

Genetické programování je nejpoužívanějším systémem automatického programování se silným zastoupením ve výzkumu. Výzkum v této oblasti je převážně orientován na problematiku designu rekombinačních operátorů s důrazem na rychlou konvergenci algoritmu a zachování diverzity v populacích. Své zastoupení má také výzkum autoadaptivních evolučních mechanismů. Jisté úsilí je věnováno propojení genetického programování s formálními gramatikami, tzv. *gramatická evoluce*, viz [27], [28], [29] a [30]. Zkoumána je také problematika znovupoužitelnosti kódu pomocí iterace, nebo rekurze, viz [21], [23]. Problematika reprezentace jedinců

nepatří k výrazněji zkoumaným oblastem genetického programování a běžně je přejímán základní přístup, který jsme uvedli v podkapitole 1.1 a podrobněji je rozveden v monografii [23]. Jak je patrné z podkapitol 1.7 a 1.8 podařilo se nalézt řešení poměrně jednoduchých funkcí s přiměřenými nároky. Nevýhodou dosažených výsledků - nalezených programů - je délka kódu jejich řešení a díky tomu také nízká „čitelnost“ pro člověka. Řešení složitějších funkcí pomocí genetického programování (bez dodatečných rozšíření) nebylo zatím s přijatelnými výpočetními nároky dosaženo.

2 Kritický pohled na účelovou funkci v genetickém programování

Účelové funkce v optimalizačních algoritmech slouží k ohodnocení kvality potenciálních řešení. Správný návrh účelové funkce je zásadní pro dosažení dobrých výsledků při optimalizaci.

Základní, dnes převažující přístup v systémech automatického programování, spočívá v ohodnocení míry shody očekávaného a dosaženého výstupu na vhodně zvolené trénovací množině. Tento přístup pochází z genetických algoritmů [33], [34] a je analogicky použit také u genetického programování [23]. Řada dalších systémů automatického programování tento přístup převzala a případně modifikovala.

Většina evolučních optimalizačních algoritmů vychází z inspirace Darwinovským pojetím evoluce, tj. že *lépe adaptovaní jednici mají větší šanci přežít a reprodukovat se*. Tento princip je realizován pomocí ohodnocovací (též účelové, nebo fitness) funkce. Setkáme se s ním mimo jiné v genetických algoritmech [25], [33], [34], genetickém programování [23], nebo v systémech založených na evolučním programování [31].

Fitness funkce slouží k ohodnocení jedinců evolučního algoritmu. V případě optimalizace parametru (např. genetickým algoritmem) je situace jasná: obvykle víme jakým způsobem zjistit „vzdálenost“ nalezené hodnoty od žádané hodnoty parametru. V případě evolvování *algoritmů* je situace podstatně složitější. Výchozím problémem je, jakým způsobem zjistit, že jeden algoritmus je objektivně „lepší“ než jiný, nebo jinak řečeno, že je částečným řešením dané úlohy, nebo se k řešení této úlohy nějak blíží ? Obvyklým způsobem měření kvality programů je míra shody získaného výstupu individua s trénovací (testovací) množinou. Problémem tohoto přístupu je to, že měření kvality podle výstupu funkce snadno umožní prohlásit za kvalitnější funkce i takové funkce, které ve skutečnosti ze sémantického hlediska mají s řešeným problémem jen minimum společného, a naopak. Ideálním způsobem ohodnocení by byla sémantická analýza kódu jedince, ta však není dostatečně obecně strojově možná. K podrobnému zpracování problematiky sémantiky v počítačových systémech odkazujeme na [32]. Dosavadní výsledky systémů automatického programování naznačují, že výstupní hodnota funkce (testování na trénovací množině) není dostatečně objektivním měřítkem pro přiřazení míry kvality algoritmu, toto měřítko je leckdy dokonce zavádějící. Tuto tezi nyní ilustrujeme na příkladech.

Příklad 1: uvažme následující jednoduchý příklad. Mějme funkci logické rovnosti dvou argumentů `EQ`, viz tabulka 3.

A	B	EQ
NIL	NIL	T
NIL	T	NIL
T	NIL	NIL
T	T	T

Tabulka 3: Tabulka pravdivostních hodnot funkce EQ

Není těžké nahlédnout, že správným řešením funkce `EQ` je například funkce:

```
(LAMBDA (A B) (IF A B (NOT B)))
```

Uvažme nyní jedinou změnu v uvedené funkci, a to prohození výrazu `B` a `(NOT B)`. Získáme tedy funkci:

```
(LAMBDA (A B) (IF A (NOT B) B))
```

Této funkci odpovídá tabulka 4:

Její fitness[12] je tedy 0, protože v žádném z testovacích případů není funkce vyhodnocena

[12] Budeme-li fitness chápat jako poměr správně vyhodnocených vstupů vůči počtu testovacích případů. Nejvyšší (nejlepší) možný fitness je 0.

A	B	F
NIL	NIL	NIL
NIL	T	T
T	NIL	T
T	T	NIL

Tabulka 4: Tabulka pravdivostních hodnot

správně, ačkoliv její kód je velmi blízký správnému řešení (správných řešení je samozřejmě více). Naopak, uvážíme-li například tuto konstantní funkci:

```
(LAMBDA (A B) T)
```

A	B	F
NIL	NIL	T
NIL	T	T
T	NIL	T
T	T	T

Tabulka 5: Tabulka pravdivostních hodnot konstantní funkce `T`

Tato funkce vyhoví pro dva ze čtyř možných případů (fitness 0.5 z 1), viz tabulka 5. Podle klasické metody ohodnocení na základě shody na trénovací množině by tato funkce byla pokládána za „kvalitnější“ než předchozí funkce, ačkoliv je zřejmé[13], že předchozí funkce `(IF A (NOT B) B)` má ke správnému řešení mnohem blíže než poslední uvedená funkce.

Příklad 2: uvažme nyní složitější funkci a to funkci pro „součet“ seznamů, přirozené číslo n bude reprezentováno seznamem délky n. Testovací případy mohou být například takové, jak uvádíme v tabulce 6.

A	B	PLUS (žádaný výstup)
(NIL NIL)	(NIL)	(NIL NIL NIL)
NIL	(NIL NIL NIL)	(NIL NIL NIL)
(NIL)	(NIL)	(NIL NIL)
(NIL NIL NIL NIL)	(NIL NIL NIL)	(NIL NIL NIL NIL NIL NIL NIL)

Tabulka 6: Tabulka pravdivostních hodnot funkce PLUS

Předpokládejme, že v průběhu experimentu získáme například funkce `E-198` a `E-212`:

```
(LABEL E-198 (LAMBDA (A B) (IF A (CONS A NIL) B)))
```

Výstup funkce `E-198` bude až na poslední testovací případ správný, viz tabulka 7.

A	B	E-198
(NIL NIL)	(NIL)	(NIL NIL NIL)
NIL	(NIL NIL NIL)	(NIL NIL NIL)
(NIL)	(NIL)	(NIL NIL)
(NIL NIL NIL NIL)	(NIL NIL NIL)	*(NIL NIL NIL NIL NIL)*

Tabulka 7: Tabulka pravdivostních hodnot funkce E-198

Funkce `E-212` má následující kód:

[13] Ovšem člověku, stroji nikoliv . . .

```
(LABEL
 E-212
 (LAMBDA (A B)
   (IF A (E-212 (CDR A) (CONS (CAR A) B)) NIL)))
```

tato funkce se v žádném z testovacích případů nevyhodnotí správně:

A	B	E-212
(NIL NIL)	(NIL)	NIL
NIL	(NIL NIL NIL)	NIL
(NIL)	(NIL)	NIL
(NIL NIL NIL NIL)	(NIL NIL NIL)	NIL

Tabulka 8: Tabulka pravdivostních hodnot funkce E-212

Podívejme se na kód funkcí `E-198` a `E-212` a položme si otázku: která z funkcí je blíže správnému řešení ? Na první pohled vidíme, že druhá (podle tradičního přístupu k fitness mnohem méně kvalitní) funkce obsahuje rekurzi, včetně restrikce prvního argumentu pomocí `(CDR A)`, takže ze sémantického hlediska je „lepší“ funkce druhá. Není obtížné nahlédnout, že pouhou záměnou posledního atomu `NIL` za `B` ve funkci `E-212` získáme, správnou funkci `PLUS`:

```
(LABEL
 E-212-MODIFIED
 (LAMBDA (A B)
   (IF A
       (E-212-MODIFIED (CDR A) (CONS (CAR A) B))
     B)))
```

A	B	E-212-MODIFIED
(NIL NIL)	(NIL)	(NIL NIL NIL)
NIL	(NIL NIL NIL)	(NIL NIL NIL)
(NIL)	(NIL)	(NIL NIL)
(NIL NIL NIL NIL)	(NIL NIL NIL)	(NIL NIL NIL NIL NIL NIL NIL)

Tabulka 9: Tabulka pravdivostních hodnot funkce E-212-MODIFIED

Tyto dva uvedené příklady funkcí `E-198` a `E-212` jsou sice pouze ilustrativní, ale dobře ukazují úskalí tradičního pojetí účelové funkce. Lze říci, že s mírou složitosti řešení hledané funkce je toto tradiční měřítko pro kvalitu algortimů méně a méně objektivní. Představíme-li si funkci pro třídění čísel, byl by počet správně setříděných čísel objektivním měřítkem? Je zřejmé, že nikoliv.

Reference

[1] Behe, Michael J. Darwinova Ä‰ernåÏ skĹÅå›Ľ>ka. Praha : NåÏvrat domĹÑ, 2001.

[2] Dawkins, R. Sobecká˝ gen. Praha : MladåÏ fronta, 1998.

[3] Dawkins, R. Slepá˝ hodinåÏĺĹÅ : zåÏzrak Ĺživota oÄ‰ima evoluÄ‰nå› biologie. Paseka, 2002.

[4] Frait, J. Makroekonomie. Ostrava : VysokåÏ ĹÏkola båÏĺ>skåÏ-TechnickåÏ univerzita, 1996.

[5] Flegr, J. Mechanismy mikroevoluce. Praha : Karolinum, 1998.

[6] Chen S.H, Yeh C.H. Genetic programming and the efficient market hypothesis. In Koza J, Goldberg D, Fogel D, Riolo R (Eds), Genetic programming 1996: proceedings of the first annual conference. MIT Press, Cambridge, 1996. 45-53.

[7] Chen S.-H, Yeh C.H. Towards a computable approach to the efficient market hypothesis: An application of genetic programming. Journal of Economic Dynamics and Control 21, 1997. 1043-1064.

[8] Chen S.H, Yeh C.H. Modeling speculators with genetic programming. In Angeline P. Reynolds R. McDonnell J. Eberhart R. (Eds.) Evolutionary programming VI, Springer-Verlag, Berlin, 1997. 137-147.

[9] Chen S.H, Yeh C.H. Modeling the expectations of inflation in the OLG model with genetic programming. Soft Computing 3, 1997. 53-62.

[10] Chen S.H, Yeh C.H. Simulating economic transition processes by genetic programming. Annals of Operation Research, 2000. 265-286.

[11] Kaboudan, M., Liu, Q. Forecasting quarterly US demand for natural gas. ITEM (Inf. Technol. Econ. Manag.), Vol. 2, 2004.

[12] Koza, John R. Hierarchical genetic algorithms operating on populations of computer programs . In Proceedings of the 11th International Joint Conference on Artificial Intelligence. San Mateo, Morgan Kaufmann. Volume I, 1989. 768-774.

[13] Koza, John R. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In: Proceedings of the Second International Conference on Tools for AI. Los Alamitos, IEEE Computer Society Press, 1990. 819-827.

[14] Koza, John R. A genetic approach to econometric modeling. pĹÅednåÏĺÏka ze Sixth World Congress of the Econometric Society, 1990.

[15] Koza, John R. Evolving a computer program to generate random numbers using the genetic programming paradigm. In Belew, Rik, Booker, Lashon (eds.). Proceedings of the Fourth International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers Inc., 1991. 37-44.

[16] Koza, John R. Evolution of subsumption using genetic programming. In Varela, Francisco J., Bourgine, Paul (eds). Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life. Cambridge, MA: The MIT Press, 1992. 110-119.

[17] Koza, John R., Rice, J. P. Automatic programming of robots using genetic programming. In Proceedings of Tenth National Conference on Artificial Intelligence. Menlo Park, CA: AAAI Press / The MIT Press, 1992. 194-201.

[18] Koza, John R. Discovery of a main program and reusable subroutines using genetic programming. Proceedings of the Fifth Workshop on Neural Networks: An International Conference on Computational Intelligence: Neural Networks, Fuzzy Systems, Evolutionary Programming, and Virtual Reality. San Diego, CA: The Society for Computer Simulation, 1993. 109-118.

[19] Keane, Martin A., Koza, John R., Rice, James P. 1993. Finding an impulse response function using genetic programming. In Proceedings of the 1993 American Control Conference. Evanston, IL: American Automatic Control Council. Volume III, 1993. 2345-2350.

[20] Koza, John R. Evolution of emergent cooperative behavior using genetic programming. In Paton, Ray (ed.). Computing with Biological Metaphors. London: Chapman and Hall, 1994. 280-297.

[21] Koza, John R., Andre, D. Evolution of iteration in genetic programming. In Fogel, Lawrence J., Angeline, Peter J., Baeck, T. Evolutionary Programming In: Proceedings of the Fifth Annual Conference on Evolutionary Programming. Cambridge, MA: The MIT Press, 1996. 469-478.

[22] Koza, John R. Using biology to solve a problem in automated machine learning. In Wynne, Clive and Staddon, John (eds.). Models of Action: Mechanisms for Adaptive Behavior. Hillsdale: Lawrence Erlbaum Associates, 1998.

[23] Koza, John R. Genetic Programming: On the Programming of Computers by Means of Natural Selection, The MIT Press, 1992.

[24] Koza, John R. Genetic Programming II: Automatic Discovery of Reusable Programs, The MIT Press, 1994.

[25] MaĹÅå›k, V. a kol. UmÄÊlåÏ inteligence III. Praha : Academia, 2002.

[26] MaĹÅå›k, V. a kol. UmÄÊlåÏ inteligence IV . Praha : Academia, 2003.

[27] O'Neill M., Ryan C. Grammatical Evolution: A Steady State approach. In Late Breaking Papers at the Genetic Programming 1998 Conference, University of Wisconsin, Madison, WI: Omni Press, 1998. 419-423.

[28] O'Neill M., Ryan C. Automatic Generation of High Level Functions using Evolutionary Algorithms. In Proceedings of SCASE 1999, Soft Computing and Software Engineering Workshop, University of Limerick, Ireland, 1999. 21-29.

[29] O'Neill M. Automatic Programming with Grammatical Evolution. In Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program, Orlando, Florida USA. San Francisco, Morgan Kaufmann, 1999. 390-391.

[30] O'Neill M., Ryan C. Grammar based function definition in Grammatical Evolution. In Proceedings of GECCO 2000, the Genetic and Evolutionary Computation Conference, 2000. 485-490.

[31] Pospå›chal, J., KvasniÄ‰ka V., TiĹ>o P. EvoluÄ‰nåŠ algoritmy, STU Bratislava, 2000.

[32] Searle, John R. Mysl, mozek a vÄÊda. Praha : MladåÏ fronta, 1994.

[33] VolnåÏ, E. NeuronovåŠ så›tÄÊ a genetickåŠ algoritmy. Ostrava : OstravskåÏ univerzita, 1998.

[34] VondråÏk, I. UmÄÊlåÏ inteligence a neuronovåŠ så›tÄÊ. Ostrava : VysokåÏ ĹÏkola båÏĹ>skåÏ-TechnickåÏ univerzita, 1994.

[35] Zelinka, I. UmÄÊlåÏ inteligence I : neuronovåŠ så›tÄÊ a genetickåŠ algoritmy. Brno : VUTIUM, 1998.

3 Seznam obrázků

1 Příklad stromové reprezentace 5
2 Křížení jedinců 9
3 Mutace kódu jedince 10
4 Jedinec po mutaci 10
5 Struktura jedince s ADF 12
6 Struktura jedince se dvěma pomocnými funkcemi 13
7 Kód jedince 13

4 Seznam tabulek

1 Výpočet $I(M, i, z)$ 14
2 Srovnání výsledků sudé parity 21
3 Tabulka pravdivostních hodnot funkce EQ 23
4 Tabulka pravdivostních hodnot 24
5 Tabulka pravdivostních hodnot konstantní funkce `T` 24
6 Tabulka pravdivostních hodnot funkce PLUS 24
7 Tabulka pravdivostních hodnot funkce E-198 24
8 Tabulka pravdivostních hodnot funkce E-212 25
9 Tabulka pravdivostních hodnot funkce E-212-MODIFIED 25