VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Genetické programování v platformově nezávislém jazyce

Genetic Programming Based on a Platform Independent Language

2016 Zdeněk Gold

Tuto stránku nahradíte v tištěné verzi práce oficiálním zadáním Vaší diplomové či bakalářské práce.

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

Zde vložte text dohodnutého omezení přístupu k Vaší práci, chránící například firemní know-how. Zde vložte text dohodnutého omezení přístupu k Vaší práce, chránící například firemní know-how. A zavazujete se, že

1. o práci nikomu neřeknete,
2. po obhajobě na ni zapomenete a
3. budete popírat její existenci.

A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. Konec textu dohodnutého omezení přístupu k Vaší práci.

V Ostravě 16. dubna 2009 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 16. dubna 2009 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Klíčová slova: evoluční algoritmy, genetické programování, syntaktický strom, křížení, mutace, selekce, program, jedinec

Abstract

This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract.

Keywords: evolutionary, genetic, genetic programming

Seznam použitých zkratek a symbolů

GP	–	Genetické programování
GA	–	Genetický algoritmus
SR	–	Symbolická regrese

Obsah

1 Úvod **5**

2 Předmluva **6**

3 Evoluční algoritmy **7**
3.1 Potřeba evolučních algoritmů 7
3.2 Charakteristika 7
3.3 Vybrané algoritmy 9

4 Genetické programování (GP) **11**
4.1 GA vs. GP 12
4.2 Úlohy řešené GP 12
4.3 Reprezentace jedince 12
4.4 Algoritmus GP 13
4.5 Omezení GP 13
4.6 Množina funkcí a terminálů 15
4.7 Reprezentace stromu 16
4.8 Generování počáteční populace 16
4.9 Operátory 17
4.10 Popis algoritmu 19

5 Implementace GP **21**
5.1 Evoluční procesy 21
5.2 UML model 21
5.3 Aplikace EvolTools 21

6 Paralelizace s technologií GPGPU **22**
6.1 CUDA 22
6.2 jCUDA 22
6.3 Paralelizace na GPU 23

7 Testování na vybraných problémech **25**
7.1 Vyhodnocení 25

8 Závěr **26**

9 Reference **27**

Seznam tabulek

Seznam obrázků

1 Nějaký graf 13
2 Průběh algoritmu Genetického programování 14
3 Příklad operátoru křížení 18
4 Příklad operátoru editace. Levý strom zjednodušíme a výsledkem je strom pravý 19
5 Vývojový proces přístupu CUDA přes JNI 23

Seznam výpisů zdrojového kódu

1 Úvod

Tento text je ukázkou sazby diplomové práce v LaTeXu pomocí třídy dokumentů `diploma`. Pochopitelně text není skutečnou diplomovou prací, ale jen ukázkou použití implementovaných maker v praxi. V kapitole **??** jsou ukázky použití různých maker a prostředí. V kapitole 8 bude „jako závěr“. Zároveň tato kapitola slouží jako ukázka generování křížových odkazů v LaTeXu.

2 Předmluva

Evoluční algoritmy vznikly jako řešení a jeden ze způsobů optimalizace v matematických a vědních kruzích. Celkem dlouho byl způsob optimalizace řešen dnes již klasickým matmatických aparátem, který je založen na infinitezimálním počtu, na variačních metodách aplikovaných ve funkcionálním prostoru nebo numerických metodách.

Tímto způsobem však lze nalézt optimální řešení pro problémy jednodušší charakter a pro ty složitější umožňuje nalézt pouze suboptimální řešení.

3 Evoluční algoritmy

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

3.1 Potřeba evolučních algoritmů

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

3.2 Charakteristika

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

3.2.1 Jedinec

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

3.2.2 Populace

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

Typickým rysem evolučních algoritmů je, že jsou založeny na práci s populací jedinců. Populace může být znázorněna jako matice NxM, kde sloupce představují jednotlivé jedince. Každý jedinec představuje aktuální řešení daného problému. S každým jedincem je navíc spojena hodnota účelové funkce, která říká, jak vhodný je jedinec pro další vývoj populace.

Hlavní činnosti evolučních algoritmů je cyklické vytváření nových populací, tedy náhrada starách populací novými. To vše pomocí přesně definovaných matematických pravidel.

K vytvoření populace je třeba nadefinovat tzv. vzor, podle kterého se generuje celá počáteční populace. Ve vzoru jsou pro každý parametr konkrétního jedince definovány tři konstanty, a to typ proměnné a hranice intervalu, v němž může pohybovat hodnota parametru. Volba hranice je velmi důležitý krok, protože při jejich nevhodném zvolení se může stát, že budou nalezena řešení , která nebudou možné fyzikálně realizovat nebo

nebudou mít opodstatnění. Další neméně důležtý význam hranic souvisí se samotným evolučním procesem. Může se stát, že daný problém bude reprezentován plochou, která bude nabývat lokálních extrémů stále větších hodnot se vzrůstající vzdáleností od počátku. To způsobí, že evoluce bude nacházet stále nová řešení až do nekonečna. Je to způsobeno tím, že evoluční proces směřuje do stále hlubších a vzdálenějších extrémů. Populace je na základě vzorového jedince vygenerována podle vzorce.

$$\Theta(1 + \alpha).$$

Tento vztah zajišťuje, že všchny parametry jedinců budou náhodně vygenerovány uvnitř povolených hranic prostoru možných řešení.

Zobrazení o tom, jak kvalitně proběhla evoluce, se provádí pomocí tzv. historie vývoje hodnoty účelové funkce ve formě jednoduchého grafu. Na něm je vykreslena závislost vývoje účelové funkce na aktuálním počtu jejich ohodnocení. Jde o sekvenci nejhorších a nejlepších řešení z jednotlivých populací. Výhodnější je však zobrazení závislosti hodnoty účelové funkce na aktuálním počtu jejich ohodnocení. To proto, že u evolučních cyklů se provádí u jednotlivých algoritmů různý počet ohodnocení účelové funkce. U prvního případu může být pomalejší konvergence hodnoty účelové funkce zobrazena jako rychlejší a naopak. Skutečná informace o kvalitě evoluce je pak zkreslená. U druhého způsobu můžeme objektivně porovnávat různé typy algoritmů bez ohledu na jejich vnitřní struktuře. Kromě vývoje nejlepšího jedince je pak vhodné zobrazovat vývoj i nejhoršího jedince z populace, a to do jednoho grafu.

Vývoj populace musí být vždy konvergentní k lepším hodnotám, což znamená, že nemůže nikdy vykazovat divergenci. V daném algoritmu funguje tzv. „elitismus“, který slouží jako jakýsi jednosměrný filtr, jenž propouští do nové populace pouze ta řešení, která jsou lepší či stejně dobrá jako ta ze staré populace.

3.2.3 Selekce

Operací selekcí v evolučních algoritmech rozumíme náhodné vybrání několika řešení z celé populace a to na základě předpokladů, které o každém řešení známe. Požadovaný je stav, aby během běhu algoritmu kulminovaly řešení v populaci k lepším, než byly na začátku. K tomu nám slouží ohodnocovací funkce. Operace selekce poté pracuje s hodnotou fitness tak, aby pozitivně zohlednila řešení s větší vhodností a znevýhodnila řešení, která mají tuto hodnotu malou.

V dalších částech této práce si představíme několik nejznámějších způsobů, jak lze z populace vybírat náhodně některá řešení a přitom zvýhodňovat lepší řešení od horších. Vždy záleží na konkrétním případě užití. Představíme si algoritmus "Ruletového kola", "Turnajového výběu"a "Výběru podle pořadí".

3.2.4 Mutace

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

3.2.5 Křížení

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

3.2.6 Ohodnocení

Každého vytvořeného reprezentativního jedince je třeba ohodnotit v rámci celé populace čítající desídky jiných jedinců tak, aby bylo možné některé jedince (řešení) upřednostňovat při výběru pro křížení před jinými (horšími) řešeními. Hodnota takto přidělená jedinci je označována za jeho fitness (neboli vhodnost).

3.2.7 Elitismus

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

3.3 Vybrané algoritmy

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

3.3.1 Genetický algoritmus (GA)

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

3.3.2 Particle Swarm

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

3.3.3 SOMA

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

4 Genetické programování (GP)

Název "genetické programování"se zrodil již počátkem 80. let, kdy byl představen jako algoritmus pro využití v problémeh jako je predikce, klasifikace, aproximace, tvorba programů. Otcem GP je standfordský informatik John Koza [1, 2]. Díky konceptu, který vytvořil, lze využít při tvorbě programů stejných evolučních operátorů, jaké obsahuje GA (křížení, mutace).

Samotné genetické programování lze definovat mnoha způsob. V různých publikacích je o něm píše různými zpsoby. Může jít o program, který umožňuje konstruovat další programy. Nebo může jít o algoritmicky založenou evoluční metodologii, inspirovananou biologickými evolučními procesy či přežití nejschopnějších, z pohledu počítačové programu, ke zjištění, který program nejlépe odpovídá řešení

Pro to, abychom pochopili smysl tohoto algoritmu, položmě si tuto otázku: "Jak se může počítač naučit řešit problémy bez toho, abychom jej k tomu přímo nenaprogramovali? Jinými slovy, jak může počítač sám dělat to co potřebujeme aby dělal, bez toho abychom mu to přesně řekli?"Touto otázkou Arthur Samuel již v roce 1959 otevřel debatu kolem umělé inteligence.

Paradigma genetického programování následuje přístup k řešení problému klasickým genetickým algoritmem s navýšením komplexnosti struktury jedince procházející adaptací. Takové jedince tvoří složitější struktury, které tvoří hirearchicky členěné počítačové programy, vyznačující se různou délkou a tvarem. Jsou to zejména problémy jako umělá inteligence, symbolická regrese, strojové učení, které lze přeformulovat na požadavek nalezení počítačového programu, který bude na výstupu generovat jiné programy jako vstup ke konkrétní úloze. Koza ve své knize tvrdí, že proces řešení těchto problémů lze formulovat jako hledání nejvhodnějšího počítačového programu v prostoru všech možných programů. Prohledávací prostor je tvořen funkcemi a terminály odpovídající doméně problému. A právě genetické programování nám poskytuje postup, jak pomocí genetického šlechtění nalézt nejlépe odpovídající program k dané úloze.

Typickým příkladem pro genetické programování je model, který se snaží nalézt logický výraz, jehož výsledkem je buď ano či ne. Můžeme si to představit na modelu banky, která eviduje své zákazníky a jejich úvěry. U každého záznamu zákazníka známe cílový stav (dostal úvěr nebo nedostal)? Nás potom ude zajímat, podle jakých kritérií bychom mohli tento model kategorizovat (čili vytvořit podmínky pro rozhodnutí, kdo úvěr dostane a kdo ne). Všechny dříve popsané algoritmy vznikly a jsou většinou užitečné v případech, kdy hledáme určitou konfiguraci pro matematický model problému tak, abychom dosáhly určitých mezních hodnot. Popusťme však ještě uzdu své fantazii a představme si situaci, kdy budeme chtít řešit problém, u kterého si nejsme jistí, jaký má být správný postup v jeho řešení. V zásadě budeme chtít vytvořit algoritmus, který dokáže generovat jiné algoritmy (či programy), který budou daný problém řešit.

Představme si situaci, kdy budeme chtít na základě nashromážděných dat vytvořit systém, který bude tyto data celkem dobře číst a popisovat je. Můžeme si to osvětlit na příkladě s bankou, která vede záznamy o úvěrech spolu s informacemi o věřitelích. Může se jednat o velikosti rodinného rozpočtu, počtu členů v rodině, počtu pracujících atd. S těmito parametry bychom potom chtěli vytvořit program, který by predikoval, zda je klient vhodným kandidátem na úvěr nebo ne.

4.1 GA vs. GP

1. Jedinec v GP tvoří řešením pro nějaký model problému, který však už není pouze konfigurace, ale celý algoritmus (program).
2. Populace v GP je tvořena stovkami až tisíci jedinci, kdy každému jedinci odpovídá nějaký program

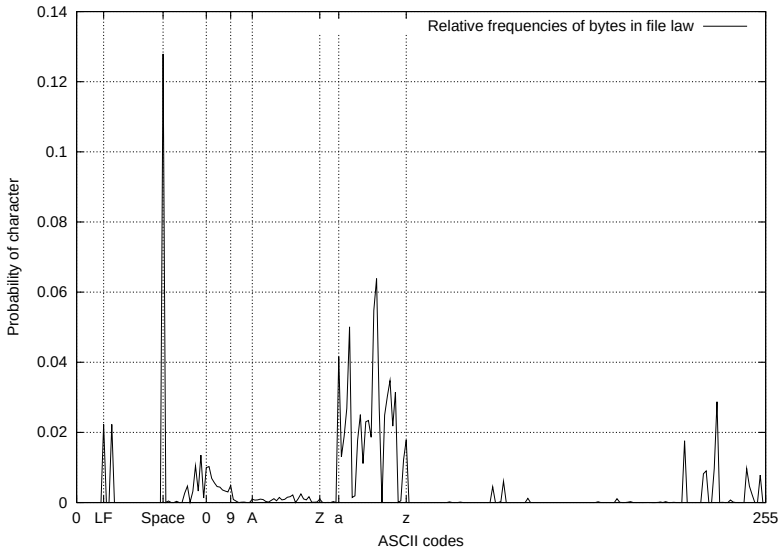
4.2 Úlohy řešené GP

1. problém umělého mravence
2. nalezení herní strategie
3. generátor náhodných čísel
4. klasifikační problémy
5. nalezení vzoru

4.3 Reprezentace jedince

První problém, který po zrození myšlenky využít GA pro tvorbu a šlechtění počítačových programů je samotná reprezentace počítačového programu. Tato reprezentace musí být dostatečně obecná pro popsání v různých programovacích jazycích a zároveň zachovávat smyslupnost, syntaktickou správnost a spustitelnost nově vyšlechtěných programů při použití evolučních operátorů mutace a křížení.

Vhodnou reprezentaci programů nalezl Koza v jazyce LISP, který reprezentuje programy jako S-výrazy, což je prakticky syntaktický strom, kterým reprezentují svoje programy překladače. Syntaktický strom je tvořeny dvěma možnými typy uzlů. Jsou to buď neterminály (tvoří funkce) a terminály (proměnné a konstanty). Při definici problémů se specifikuje množina neterminálů $\Pi = \{f_1, f_2, ..., f_n\}$ a množina terminálů $\Gamma = \{t_1, t_2, ..., t_n\}$. Příkladem syntaktického stromu, který znázorňuje výraz pro kombinaci k-té třídy z n prvků je na obrázku.



Obrázek 1: Nějaký graf

Vlastnostmi se podobá neuronovými sítěmi. Od jiných evolučních algoritmů vyčnívá svou značně velkou populací (čítající tisíce jedinců) a z toho důvodu se jeví, jako značně pomalý algoritmus. Výhodou oproti jiným algoritmům (např. genetický algoritmus), které mají většinou lineárlní strukturu, zde tvoři jedince ne-lineární chromosomy (např. stromy či grafy).

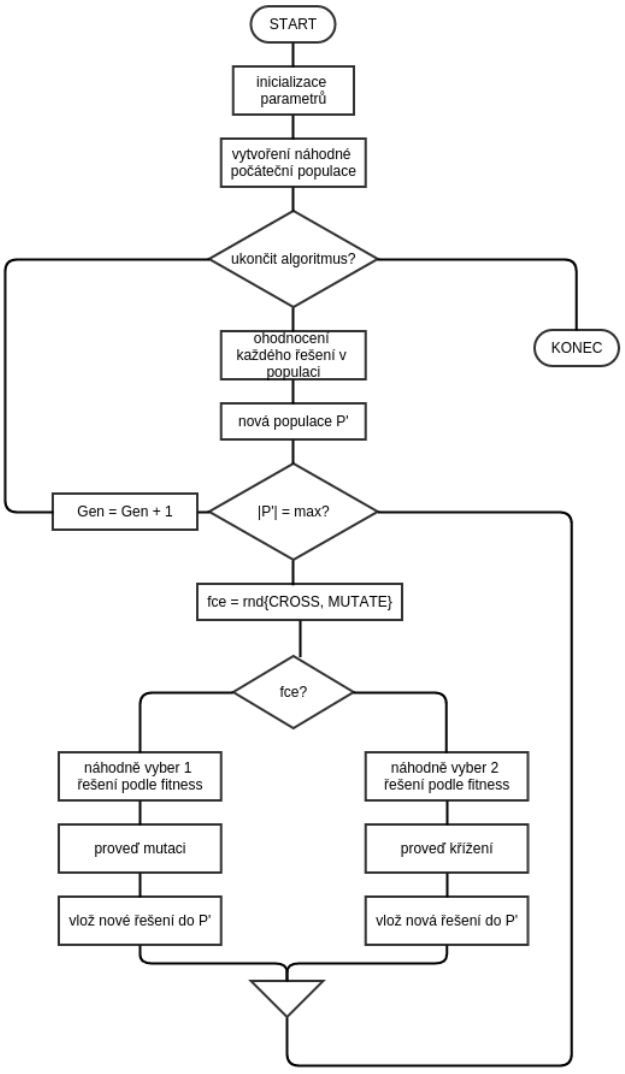
Funkce f_i z množiny Π tvořící uzly syntaktického stromu jejichž argumenty reprezentují hrany vedoucí do hloubky o jedno větší. Terminály zastupují proměnné či konstanty umístěné v listových uzlech ukončují růst stromu a samotné individuum.

4.4 Algoritmus GP

4.5 Omezení GP

Koza ve své knize definuje [1] dvojí požadavky na množinu funkcí a terminálů. Je to požadavek uzavřenosti (closure) a postačitelnosti (sufficiency) obou množin. Uzavřenost množiny je splněna tehdy, pokud může libovolná funkce příjmout jako argument libovoulnou funkci z množiny funkcí či terminál z množiny terminálních symbolů. Uzavřenost zamezí tvorbu syntakticky nesprávných programů.

Postačitelnst nám naproti tomu umožňuje, abychom byli schopní k danému problému nalézt odpovídající program (funkci), která by jej řešila. Díky tomu jsme schopni říci, že k danému problému můžeme vyjádřit řešení daného problému. Představme si problém nalezení potravy umělým mravencem v mřížkové soustavěm kterou můžeme popsat množinou funkcí tvořenou příkazy $\{KROKVPRED, OTOCVLEVO, OTOCVPRAVO\}$ a



Obrázek 2: Průběh algoritmu Genetického programování

množinou terminálů s jediným příkazem $\{ZASTAV\}$. I bez dokázaní si můžeme jasně říci, že jsme schopní díky těmto příkazům dostat mravence do jakéhokoliv místa a zastavit.

U jednoduchých příkladů, kde je množina terminálů a neterminálů tvořena pouze Booleovskými funkcemi můžeme i bez důkazů předpokládat, že takové množiny jsou uzavřené a splňují také podmínku postačitelnosti. Ovšem v reálných příkladech budeme často používat programy složené číselných proměnných a funkcích, podmínkách, rekurzích, ruzných typů v argumentech funkcí. Nebo budeme při práci s číselnými funkcemi muset ošetřit některé zakázané stavy (dělení nulou, logaritmus záporného čísla, atd.). V takových případech už podmínka uzavřenosti nemusí platit.

I v těchto případech se však můžeme odkázat na Kozu, který tvrdí, že lze jednoduchým postupem zajistit, aby byly množiny terminálů a funkcí uzavřené pro obecně jakýkoliv problém. Vezměme si případ programu pracujícího s číselnými funkcemi a terminály, kde může nastat situace, že se bude muset provést dělení nulou. Proto je třeba upravit funkce z těchto množin tak, aby umožňovaly zpracovat také nedefinovatelnou hodnotu ("undefined") v podobě nějakého zástupného symbolu a ošetřit stavy funkcí tak, aby funkce mohla vrátit kromě čísla také tuto hodnotu v případech, kdy je operae zakákána nebo v případě, že alespoň jeden z argumentů vrací tuto hodnotu. Takto upravené funkce Koza nazývá chráněné funkce (protected functions).

Vlastnost uzavřenosti množin je vhodná, ale není absolutně nutná. Jako alternativa k chráněným funkcím může sloužit jiný přístup. Pokud striktně nevyžadujeme, aby byl program syntaktického stromu validní, můžeme nesprávně generované stromy v populaci zakázat nebo je jinak penalizovat, aby nepřecházely do nových generací. V této práci se však takovým způsobům nebudeme zabývat a budeme pracovat pouze se splněnou podmínkou uzavřenosti.

Protože výsledná funkce je tvořena nejen proměnnými, ale také konstantami, je třeba si říci, jak je při tvorbě jedince reprezentovat. Protože můžeme chtít použít GP pro ekonomické modelování, měl by mít algoritmus možnost generovat vhodné konstanty v daném rozsahu a daného typu podle řešené úlohy. Koza navrhuje množinu terminálů o náhodnou konstantu C tak, že se během běhu programu při ohodnocení individuí doplní o náhodnou hodnotu z příslučné množiny. Toto číslo se potom využije jako kterýkoliv jiný terminál.

4.6 Množina funkcí a terminálů

Množina funkcí může obsahovat:

1. aritmetické operace (+, -, *, /, atd.)
2. matematické funkce (AND, OR, NOT, NAND, atd.)

3. podmínkové operátory (IF-THEN-ELSE)
4. iterační konstrukce (DO, FOREACH, WHILE)
5. rekurzivní funkce

Aby byl algoritmus GP skutečně úspěšný a efektivní při hledání vhodného řešení, musíme velmi pečlivě definovat množinu použitých funkcí a také terminálních symbolů. Chybí-li v množině funkcí nějaký klíčový krok, můžeme se nám tvorba správného programu zesložitit či v extrémním případě nemusíme dojít k řešení vůbec. Naproti tomu, pokud budeme mít množinu funkcí příliš bohatou, rozšíří se prostor možných řešení a tedy i časová složitost pro prohledání celého prostoru. Může se stát, že i přesto, že je výsledná funkce jednoduchá, bude přesto časově zdlouhavé, abychom toto řešení nalezli. Je to jako hledat jehlu v kupce sena. Vhodné je omezit funkce na co nejmenší počet.

4.7 Reprezentace stromu

Implementací syntaktického stromu je několik. Jednou z možností jak strom v počítačích reprezentovat je s použitím uzlů obsahující sezname potomků. Jde o přirozené vnímání stromové struktury, která je však při samotné implementaci dosti obtížně manipulovatelná.

Druhou možností je seznam, který představuje strom v prefixovém tvaru zápisu. Pořadí prvků v seznamu odpovídá pořadí navštívení každého uzlu ve stromě, při procházení stromu do hloubky. Zajištění zpětné rekonstrukce stromu nám umožňuje informace o tom, zda prvek patří do množiny terminálních symbolů nebo naopak do množiny funkcí a arita každé funkce. Příkladem takovéto stromové reprezentace je na obrázku níže.

4.8 Generování počáteční populace

Při prvotním generování populace se budeme zabývat, jakým způsobem lze náhodně vygenerovat několik individuí. Existují dvě metody řídící růst stromu. Úplná (full method), kdy všechny listové uzly stromu mají stejnou hloubku, která je rovna maximální velikosti stromu. Metoda růstová (grow method) naproti tomu dovoluje tvorbu rozmanitějších stromových struktůr. Pro generování následného potomka se náhodně rozhoduje, zda je použitý termínál nebo uzel. I zde je limitní maximální hloubka stromu.

Koza ve své knize [1] doporučuje kombinaci obou metod, což v praxi znamená, že pro polovinu populace se použije růstová metoda a pro druhou polovinu úplná. Zároveň rovnoměrným rozložením hloubky stromů mezi všechny jedince, snížíme pravděpodobnost výskytu stejnývh řešení, která jsou v populaci nevyhovující.

Vše si můžeme ukázat na příkladě populace s $N = 500$ jedinců, kde je stanovena maximální hloubka $h_max = 6$. Při generování základní populace budeme postupovat tak, že 100 jedinců bude generováno s $h_max = 2$ (50 s růstovou metodou a 50 s úplnou metodou),následně pro dalších 100 řešení generujeme obdobné stromy hloubký o 1 vyšší $h_max = 3$ a tak dále, až do hloubky $h_max = 6$.

4.9 Operátory

4.9.1 Křížení

V GP budeme používat stejné operace jako u GA. Operátor křížení pracuje s dvojícemi stromů. U každého stromu z dvojíce zvolí náhodně uzel pro křížení a v dalším kroku zamění podstromy pod zvolenými uzly mezi sebou. Více osvětlíme příkladem na obrázku. Oproti GA se zde potkáváme s problémem postupného růstu hloubky reprezentujících stromů. V první generaci si obstaráme stromy, který mají definouvanou maximální hloubku. Při aplikaci křížení však snad nastává případ, kdy je například listový uzel jednoho stromu nahrazen téměř celým podstromem druhého jedince. V podstatě jen v případech, kdy se budou zaměňovat podstromy z uzlů, které jsou ve stejné hloubce se žádný z obou stromů nerozšiřuje. Ve všech ostatních případech bude jeden nebo druhý strom rozšířený.

Koza doporučuje, abychom definovali i maximální hloubku stromů vzniklých křížením. Doporučuje se trojnásobek hloubky počáteční populace. Pokud bychom při křížení překročili tuto maximální povolenou hranici, takto vzniklí potomci budou odmítnuti a nahrazeni jedním z rodičovských jedinců.

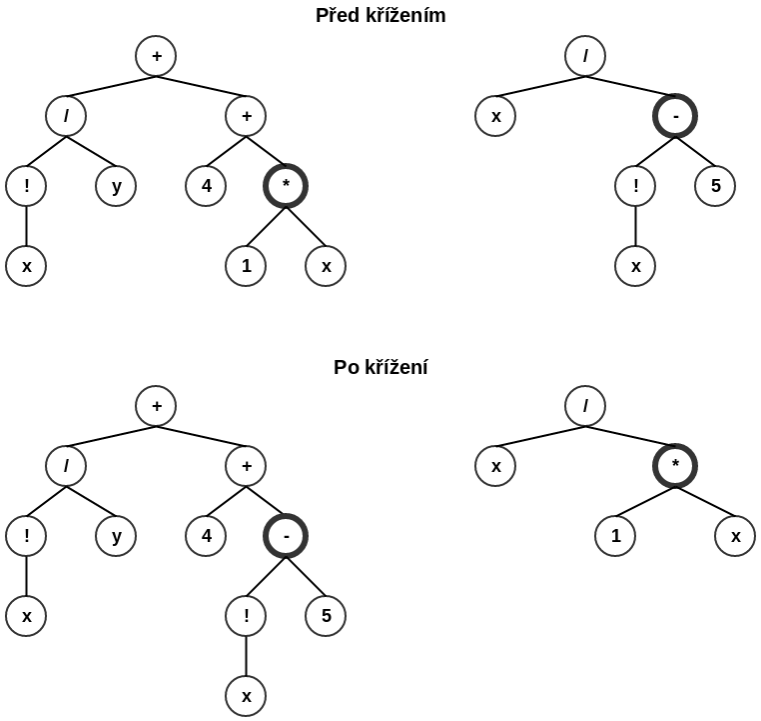
Speciálním případem při křížení je tzv. "incestní křížení", které nastane tehdy, pokud do křížení vstoupí dva identiční jedinci. V případě GA je takový případ nežádoucí, protože vznikají identiční potomci. V případe GP k tomu nastává pouze tehdy, když by byl v obou stromech vybrán tentýž uzel pro křížení.

4.9.2 Mutace

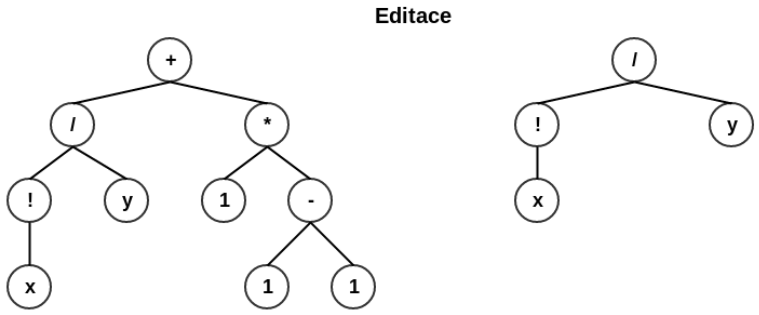
Operátor mutace také náhodně vybere uzel stromu a celý jeho podstrom odstraní. Na tomto místě se poté náhodně vygeneruje chybějící část stromu do maximální hloubky předepsané parametrem tak, aby vyýsledný strom nebyl příliš velký.

Variant operace mutace je několik. Mezi základní patří tyto:

1. uzlová mutace (point mutation) - neterminální uzel nahradí neterminálem se stejnou aritou a terminální uzel jiným terminálem
2. vyzvedávající mutace (boist mutation) nahrazuje celý syntaktický strom některým z jeho podstromů



Obrázek 3: Příklad operátoru křížení



Obrázek 4: Příklad operátoru editace. Levý strom zjednodušíme a výsledkem je strom pravý

3. smršťující mutace (shrink mutation) nahrazuje náhodně zvolený podstrom jediným terminálem

4.9.3 Další operátory

To však není celý výčet možných operátorů. Existují i operátor, které se v GP nevyužívají zcela běžně, ale stojí za to se o nich zmínit. Mezi méně časté operátory řadíme například operátor permutace, zapouzdření, editace či decimace.

Operátor permutace je příkladem asexuálního operátora, který využívá jen jednoho rodiče a náhodně zvolí neterminál s n operátory, jejichž pořadí náhodně prohodí. Jde tedy o náhodné prohození pořadí větví takovéhoto podstromu.

Operátor editace se snaží poskytnou nedestruktivní redukci hloubky stromu tak, aby výsledný strom představoval tentýž program, ale bez přebytečných uzlů. Nejlépe, když si takový případ ukážeme na obrázku.

4.10 Popis algoritmu

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

Příklad 4.1
Postup evolučního algoritmu

1. Vymezení parametrů evoluce – jako je stanovení kritéria ukončení (počet cyklu, vhodnost řešení,...), stanovení účelové funkce, případně tzv. vhodnosti. Účelovou funkcí se rozumí obvykle matematický model, jehož minimalizace/maximalizace vede k řešení

2. Generování prvopočáteční populace (obecně matice M x N, kde M je počet parametrů jedince a M je počet jedinců v populaci). Jedincem se rozumí vektor čísel s takovým počtem složek, kolik je optimalizovaných parametrů. Složky jsou nastaveny nahodile a každý jedinec představuje jedno možné řešení
3. Všichni jedinci se ohodnotí přes definovanou účelovou funkci a každému z nich se přiřadí: a) buď přímá hodnota vrácená účelovou funkcí, nebo b) vhodnost, což je upravená hodnota účelové funkce
4. Nastává výběr rodičů podle jejich kvality
5. Křížením rodičů se tvoří potomci. Proces křížení je u každého algoritmu odlišný.
6. Každý potomek je zmutován
7. Každý jedinec se ohodnotí stejně jako v kroku 3.
8. Vyberou se nejlepší jedinci
9. Vybraní jedinci zaplní novou populaci
10. Stará populace je zapomenuta a na její místo nastupuje populace nová. Dále se pokračuje krokem 4.

Evoluční algoritmy nejsou populární jen proto, že jsou moderní a odlišné od klasických, ale hlavně pro fakt, že v případě vhodného aplikování jsou schopny nahradit člověka ■

5 Implementace GP

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

5.1 Evoluční procesy

Evoluční procesy, do kterých řadíme "přirozený výběr", "mutaci", "křížení"a další techniky, které jsou inspirované zákonitostmi v přírodě jsou základním kamenem pro jakýkoliv program napodobující evoluční chování za pomocí počítačů. Proto je teřba dbát o co nejvhodnější a nejpreciznější implementaci tak, aby byl výsledný algoritmus dobře použitelný i pro nejrůznější problémy. Představme si nyní některá úskalí, která je třeba dobře postihnout a zefektivnít výsledný kód.

5.1.1 Výběrová funkce

V evolučních technikách se využívá vždy a ovlivňuje populační vývoj, protože předchází evoluční operátory (křížení, mutaci). Tato funkce se má stát přirozeným výběrem, který v přírodě funguje již od pradána. Tedy právo silnějšího (odolnějšího, přizpůsobivějšího) jedince na tvorbu nové generace.

Několik pravidel pro efektivnější výběrovou funkci:

1. Zamezení výběru dvou totožných jedinců pro křížení
2. Možnost vícenásobného výběru stejného jedince (posílení vlivu lepších řešení do dalších populací)

5.2 UML model

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

5.3 Aplikace EvolTools

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

6 Paralelizace s technologií GPGPU

Nedávným trendem dnešních běžných stolních systémů je využít univerzálních grafických procesorových jednotek (GPGPU) s cílem získat o řád vyšší výpočetní výkon než u CPU jednotek. Tradiční CPU je současně schopno spouštět pouze několik málo vláken. Naopak GPU umožňuje provádět stovky až tisíce vláken. Jako příklad lze uvést NVIDIA Compute Unified Device Architecture (CUDA) se ukázal jako populární programovací model pro GPGPUs s použitím C/C++.

V tomto článku představíme programovací rozhraní s názvem JCUDA, které mohou být použity Java programátory k vyvolání jádra CUDA. Pomocí tohoto rozhraní mohou programátoři píší Java kódy, které přímo volají jádra CUDA, aniž by se museli starat o podrobnostech přemostění běhového prostředí Java a CUDA runtime.

6.1 CUDA

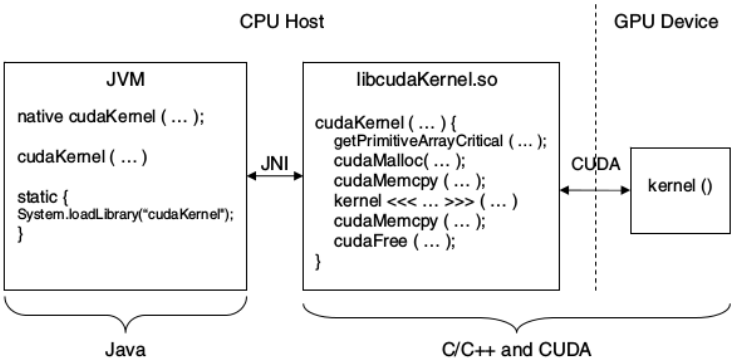
Poháněn nenasytnou poptávkou po real-time 3D hrách a multimediálním zážitku, programovatelné GPU (Graphics Processing Unit) se vyvinul do vysoce paralelní, vícevláknových, manycore procesoru. Současné GPU mají desítky či stovky fragmentů procesorů a vyšší paměťové šířky pásma než běžné procesory. Například, NVIDIA GeForce GTX 295 grafická karta je dodáván se dvěma GPU, každý s 240 procesorovými jádry a pamětí s 233,8 Gb/s šířky pásma, což je asi 10-krát rychlejší než u současných procesorů 1,8 GB. Stejný GPU je součástí NVIDIA Tesla C1060 procesor počítače, který je GPU procesor použitý v našich výkonnostních hodnoceních.

Programovací model CUDA je rozšíření jazyka C. Programátoři píší aplikace se dvěma částmi kód - funkce, které mají být provedeny na počítači a funkce procesoru, které mají být provedeny na GPU zařízení. Vstupní funkce kódu součástky jsou označeny globální klíčové slovo, a jsou označovány jako jádra. Kernel provádí paralelně v sadě paralelních vláken v jediné instrukce Multiple Thread (SIMT) Model

6.2 jCUDA

Smyslem projektu jCUDA (Java pro CUDA®) je nabídnout pro jazyk Java vazbu na NVidia CUDA framework spolu s jejími knihovnami, které umožňují plně využít výpočetní výkon grafických karet podporujícími různé verze CUDA, na různých platformách, bez nutnosti měnit svůj kód. Knihovna je nabízená jak pro platformu Windows (32, 64-bit), tak Linux(32, 64-bit).

Chcete-li využit knihoven JCUDA, je potřeba mít grafickou kartu, podporující CUDA spolu s ovladači. Informace o podpoře CUDA u grafických karet NVidia jsou zveřejněny na stránkách NVidie. Požaduje se minimální nároky na CUDA verzi nejméně 1.1 a Java Runtime 1.5 a vyšší.



Obrázek 5: Vývojový proces přístupu CUDA přes JNI

jCUDA je nabízená s volnou licencí pro osobní požití, vědeckou či výzkumnou činnost a komerční využití. Ovšem zde není poskytnuta žádná podpora ani záruka.

Java Native Interface (JNI) [6], rozhraní Javy pro provádění funkcí nativního jazyka C, hrálo také významnou roli v projektech JGF, jako je umožnění Message Passing Interface (MPI) pro Javu. V JNI, programátor deklaruje vnější nativní metody vybraných funkcí C, které lze vyvolat Java programem. Po kompilaci zdrojové soubory Java je použit nástroj javah ke generování hlavičkové soubory C, které obsahuje rozhraní nativního kódu. JNI podporuje také bohatou paletu funkcí zpětného volání a umožní nativnímu kódu přístup k Java objektům a služebám.

Obrázek 5 shrnuje proces třístupňový, že programátor Java potřebuje řídit přístup k CUDA přes JNI dnes. To zahrnuje psaní kódu Java a JNI pahýl kód v jazyce C pro provedení na CPU, stejně jako CUDA kód pro provedení na GPU zařízení. Otočný kód musí také zvládnout alokaci a uvolnění dat v paměti přístroje a datové přenosy mezi hostitelem a zařízení. Je jasné, že tento proces je zdlouhavý a náchylné k chybám, a že by bylo více produktivní použít kompilátor nebo programovací nástroj, který automaticky generuje pahýl kód a přenosu dat volání.

6.3 Paralelizace na GPU

Pro využití grafického procesoru k výpočtům algoritmu genetického programování je třeba daný proces správně paralelizovat, aby byl výpočet, pokud možno co nejefektivnější a nejjednodušší.

6.3.1 Paralelizace genetického programování

V následující části bude popsáno, jakým způsobem bude program komunikovat s grafickým procesorem a jak bude algoritmus paralelizován přímo na grafickém procesoru.

Pro dosažení maximálního výkonu při běhu algoritmu, přesuneme celý algoritmus přímo na GPU. Pro tento účel bude třeba vytvořit kernel gp.cu, který se bude řídit provádění evolučního cyklu. Kernel bude volán s parametry: gen - počet generací populace, popSize - velikost populace, pop - kolekci řešení v poli, funs - funkce, oper - operátory.

Je třeba si uvědomit, že takýto přístup k paralelizaci je individuální a je závislý na daném problému, který genetický algoritmus řeší. Proto mimo tento kernel bude existovat funkce art ant.cu, která se bude starat o vyhodnocování jednotlivých řešení.

6.3.2 Návrh paměťového modelu

Aby byla paralelizace úspěšná, je důležité si dobře zanalyzovat umístění a zarovnání dat pro výpočet v paměti. Lze bez nadsázky říci, že tento krok je při dosažení správných výsledků a také výkonu nejdůležitější.

7 Testování na vybraných problémech

Po představení pestré, ale zdaleka né kompletní, škály algoritmů je třeba si představit a vyhodnotit jejich sílu na problémech, které se objevují v mnoha publikacích jako vhodní adepti. V následujících pasážích si představíme ty nejběžnější avšak netriviální problémy. U každé definice problému je popsáno, jakými evolučními algoritmy je vhodné tyto problémy řešit. Spolu s definicí problému si popíšeme i výsledky, které byly během testování zaznamenány a konfigurace s jakými byly algoritmy spouštěny. Podtržením toho všeho bude srovnání s výsledky, které byly zveřejněny v publikacích jako optimální, přičemž nemáme za cíl dosáhnout co nejlepšího řešení, ale orientačně vyhodnotit použitelnost na problémech s přihlédnutím k tomu, že výsledným produktem této práce bude aplikace sloužící jako opora při úvodu do problematiky a výuky evolučních algoritmů a hlavně genetického programování.

7.0.3 Sudoku

7.0.4 Aproximace modelu

7.1 Vyhodnocení

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

8 Závěr

Tak doufám, že Vám tato ukázka k něčemu byla. Další informace najdete v publikacích [3, 4].

Zdeněk Gold

9 Reference

[1] Koza, J. R., *Genetic programming. On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

[2] Koza, J. R., *Genetic programming II. Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press, 1994.

[3] Goossens, Michel, *The LaTeX companion*, New York: Addison, 1994.

[4] Lamport, Leslie, *A document preparation system: user's guide and reference manual*, New York: Addison-Wesley Pub. Co., 2015.

[5] Koza J.R, *Genetic Programming*, MIT Press, ISBN 0-262-11189-6, 1998

[6] Koza J.R, Bennet F.H., Andre D., Keane M., *Genetic Programming III*, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999

[7] Lampinen Jouni, Zelinka, Ivan, *New Ideas in Optimization and Mechanical Engineering Design Optimization by Differential Evolution. Volume 1*, London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5

[8] Yan Y., Grossman M., Sarkar V., *Euro-Par 2009 Parallel Processing*, Department of Computer Science, Rice University, 2009. 899 p. ISBN 978-3-642-03868-6

[9] CUDA GPUs. *NVidia Corporation: Accelerated Computing [online] 2016*, [cit. 2016-02-15]. Dostupné z: https://developer.nvidia.com/cuda-gpus

[10] JCUDA. *Company for Advanced Supercomputing Solutions LTD: jCUDA [online] 2008-2015*, [cit. 2016-02-15]. Dostupné z: http://www.cass-hpc.com/solutions/legacy/jcuda/