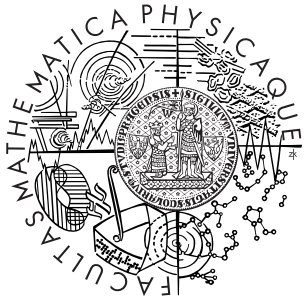
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Filip Děchtěrenko

Aplikace genetických algoritmů v automatickém dokazování vět.

Katedra teoretické informatiky a matematické logiky

prof. RNDr. Petr Štěpánek DrSc.

Studijní program: Informatika
Studijní obor: Obecná informatika

2010

Chtěl bych poděkovat všem, kteří mi zapůjčili literaturu a zejména prof. Petru Štěpánkovi za kvalitní vedení práce a ochotu při konzultacích . Dále bych chtěl poděkovat dr. Petru Pudlákovi za konzultace a Mgr Milanu Fučíkovi za přístup na fakultní server, na kterém probíhalo testování.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 18. května 2010 Filip Děchtěrenko

Obsah

Název práce: Aplikace genetických algoritmů v automatickém dokazováním vět.
Autor: Filip Děchtěrenko
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: prof. RNDr. Petr Štěpánek DrSc.
e-mail vedoucího: petr.stepanek@mff.cuni.cz

Abstrakt: V předložené práci studujeme možnosti použití genetických algoritmů v automatickém dokazování vět. Zaměříme se na dokazovač Prover9 a konkrétně na používání definic pro zrychlení prohledávání klauzulí. Na závěr zhodnotíme přínos genetických algoritmů do automatického dokazování vět.
Klíčová slova: Automatické dokazování, genetické programovaní, Prover9, definice

Title: Aplication of genetics algorithm in automated theorem proving
Author: Filip Děchtěrenko
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: prof. RNDr. Petr Štěpánek DrSc.
Supervisor's e-mail address: petr.stepanek@mff.cuni.cz

Abstract: In the present work we present possibilities of use of the genetic algorithms in the automated theorem proving. We focus on the prover Prover9 and specifically using the definitions for speeding up the search for clausulas. In the end we evaluate the benefit of the genetics algorithms in automated theorem proving.
Keywords: Automated theorem proving, genetic programming, Prover9, definitions

Kapitola 1

Úvod

Strojové dokazování vět je sice dnes již klasická problematika, ale je stále aktuální pro nové úkoly. Hledání důkazu bývá časově velmi náročné, např. pro známou Robbinsonovu domněnku[1] trvá i při dnešních výpočetní síle několik hodin. Stále častěji používané genetické algoritmy jsou velmi dobrou metodou pro hledání řešení v problémech s rozsáhlým prohledávacím prostorem.
Rozhodli jsme se zaměřit se na možnosti propojení automatického dokazování s genetickými algoritmy a zhodnotit, zda budou přínosem pro urychlení hledání důkazu. Konkrétně nás bude zajímat metoda zavádění definic, která byla použita pro zkrácení doby prohledávání důkazu u Robbinsonovy domněnky[2].
V práci budeme používat anglickou terminologii pro klíčové pojmy související s dokazovačem Prover9 v souladu s manuálem k Prover9.

[1] viz. [5]
[2] viz. [6]

Kapitola 2

Příprava

Nejprve se seznámíme se základními pojmy z automatického dokazování vět a genetických algoritmů. U následujících termínů přepodkládáme, že je s nimi čtenář seznámen: **formule**, **literál**, **klauzule**, **varianta formule**, **dokazatelnost**, **splnitelnost**, **unifikace**, **mgu**. Pojmy z logiky nalezneme v [12] a pojmy z logického programování (unifikace a mgu) jsou vysvětleny v [1]. Budeme je v textu dále používat bez dalšího vysvětlování. Formule použité v práci budeme psát v jazyce logiky prvního řádu.

2.1 Automatické dokazování vět

Automatické dokazování vět je obor na pomezí informatiky a matematiky zabývající se algoritmy, které ze souboru axiomů odvodí zadanou domněnku. Dnes je strojové dokazování využíváno např. pro verifikaci software a hardware, v bioinformatice a dalších.
Základní algoritmus pro automatické dokazování využívá následující věty z výrokové (resp. predikátové) logiky

$$T \vdash A \text{ právě když } T \cup \{\neg A\} \text{ je sporná teorie} \tag{1}$$

kde T je množina formulí a A je formule. Použitím této věty převedeme problém hledání důkazu na problém hledání spornosti teorie. Z hlediska efektivity je totiž výhodné omezit formule pouze na klauzule, protože na nich se lépe aplikují tzv. odvozovací pravidla. Odvozovací pravidlem budeme nazývat pravidlo, které z předpokladů odvodí nějaký závěr. Nejpoužívanější odvozovací pravidlo je **rezoluční pravidlo** (zkráceně rezoluce), které vy-

padá v binární verzi následovně

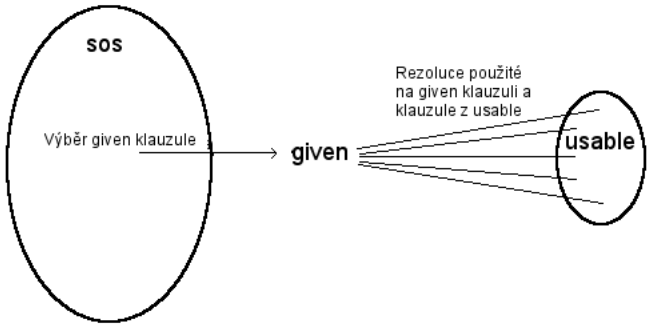
$$\frac{A \vee C \quad (B \vee \neg D)}{(A \vee B)\theta}$$

A,B jsou klauzule, C,D jsou literály a θ je mgu C a D. Formuli $A \vee B$ budeme nazývat rezolventa. Je-li rezolventa prázdná klauzule (tedy disjunkce bez argumentů) říkáme, že jsme odvodili spor. Zde je vidět, že je vhodné pracovat s klauzulemi namísto libovolných formulí, protože sestávají pouze z negací a disjunkcí a můžeme na ně snadno aplikovat rezoluční pravidlo.
Algoritmů pro hledání sporu je několik, my se však zaměříme na způsob, jakým hledá spor dokazovač Prover9.

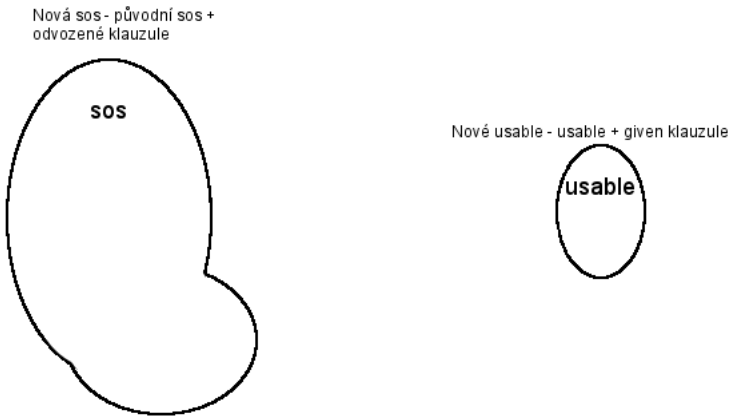
2.2 Prover9

Prover9[1] je rezoluční dokazovač pro predikátovou logiku prvního řádu s rovností, jehož autorem je William McCune. Prover9 při hledání sporu používá dvě množiny nazvané **sos** (set-of-support) a **usable**. Sos je množina dosud nezpracovaných klauzulí (reprezentovaná jako seznam). Každá klauzule v sos je označena identifikátorem, který odpovídá stáří klauzule v sos (tj. čím nižší index, tím je klauzule v sos déle), a příznakem "true" nebo "false", který Prover9 nastavuje klauzulím podle toho, zda jsou pravdivé v zadané interpretaci. Pokud není ve vstupním souboru žádná interpretace definovaná (v práci s interpretací pracovat nebudeme), nastavuje Prover9 "false" negativním klauzulím a "true" jinak. Negativní klauzulí rozumíme takovou klauzuli, která má všechny literály negativní, např. $x \neq y \vee \neg p(x, y)$, kde x,y jsou proměnné a p je binární predikátový symbol[2].
Usable je množina již zpracovaných klauzulí, které jsou používány na odvozování nových klauzulí. Celý proces hledání důkazu probíhá v krocích, přičemž jeden krok důkazu vypadá následovně:
Ze sos je vybrána klauzule (o problému vybírání se zmíníme později), kterou budeme označovat **given klauzule**. Pomocí této klauzule a ostatních klauzulí z usable jsou použitím rezolučního pravidla odvozené nové klauzule, které se uloží do sos (při vkládání do sos je jim přiřazen identifikátor a sémantická "true"/"false" hodnota). Given klauzule se přesune do usable. Názorně to je vidět na obrázcích 1 a 2.

[1] detailní popis v [4]
[2] více informací v [8]



Obrázek 1: Proces výběru klauzule.



Obrázek 2: Nové sos a usable pro další krok.

Počet nově odvozených klauzulí v každém kroku bývá velmi velký a sos se proto velmi rychle zvětšuje. Při každém tvoření nových klauzulí pomocí rezolučního pravidla se kontroluje, zda nenastal spor (v tom případě končíme prohledávání s úspěchem). Pokud již není možné vybrat klauzuli z množiny sos, skončí hledání důkazu neúspěšně.
Na začátku prohledávání je usable prázdná a sos se inicializuje (v souladu s větou (1)) axiomy zadaného problému a negací cíle problému. Přestože v automatickém dokazování pracujeme s klauzulemi, implementuje většina dokazovačů tzv. **klauzifikaci**, což je proces, který převádí formule do klauzulí. V klauzulích jsou všechny proměnné vázané všeobecnými kvantifikátory, existenčních kvantifikátorů se při klauzifikaci zbavíme skolemizací. Více o klauzifikaci a skolemizaci v [3]
Následující příklad ukazuje činnost Prover9 (pro účely názornějšího vysvětlení funkce hledání sporu vynecháme indexování klauzulí a příznaky "true"/"false"):
Mějme teorii T a domněnku A zadané následovně:

$$T = \{p(x) \vee q(x), \neg p(x)\} \text{ a } A \equiv q(x)$$

kde p,q jsou unární predikátové symboly. Na začátku tedy budeme mít:

$$sos = \{p(x) \vee q(x), \neg p(x), \neg q(c)\}, usable = \{\}$$

kde c je nová konstanta zavedená při skolemizaci. Predikát $q(c)$ jsme získali následovně: po aplikování negace na klauzuli $\neg((\forall x)q(x))$ dostaneme $(\exists x)\neg q(x)$ a po skolemizaci dostaneme klauzuli $q(c)$.
1. krok:
Za given vyber $p(x) \vee q(x)$. Protože je množina usable prázdná, nemůžeme provést žádné odvození pomocí rezoluce a pouze přidáme given klauzuli do usable
po 1. kroku: $sos = \{\neg p(x), \neg q(c)\}, usable = \{p(x) \vee q(x)\}$
2. krok:
Za given vyber $\neg p(x)$. Po použití rezoluce na $\neg p(x)$ a $p(x) \vee q(x)$ dostaneme rezolventu $q(x)$, kterou přidáme do sos.
po 2. kroku: $sos = \{q(x), \neg q(c)\}, usable = \{p(x) \vee q(x), \neg p(x)\}$
3, krok:
Za given vyber $q(x)$. Nyní nemůžeme použít rezoluční pravidlo (literál $q(x)$ nemá v klauzulích z usable unifikovatelný doplňkový literál $\neg q(x)$), proto pouze přesuneme klauzuli do usable.
po 3. kroku $sos = \{\neg q(c)\}, usable = \{p(x) \vee q(x), \neg p(x), q(x)\}$
4. krok:

Za given vyber $\neg q(c)$. Použití rezolučního pravidla na $\neg q(c)$ a $q(x)$ dostaneme spor. Více informací o smyčce. kterou využívá Prover9 v [11].
V příkladu byly klauzule vybírány nahodile, ve skutečnosti je situace složitější a budeme se jí zabývat později.
Prover9 jsme si vybrali pro naše účely především pro jeho velkou nastavitelnost pomocí parametrů (vypínání/zapínání jednotlivých odvozovacích pravidel, nastavení způsobu výběru given klauzule) a také pro kvalitní dokumentaci. Jazyk, kterým specifikujeme problémy pro dokazovač, má podobnou syntax jako Prolog. Struktura vstupního souboru vypadá následovně.

```
formulas(sos).
  % sekce pro axiomy
end_of_list.
formulas(goals).
  % sekce pro cíle
end_of_list
```

Axiomy i cíle nemusí být zapsány jako klauzule, protože Prover9 v případě potřeby před hledáním sporu provádí klauzifikaci. Prover9 používá pro zvýšení výkonu i jiná odvozovací pravidla než rezoluci, např. faktorizaci a paramodulaci.[3]
Pro usadnění práce s dokazovači byla G. Sutcliffem vytvořena databáze TPTP[4] (Thousands of Problems for Theorem Provers). TPTP obsahuje velké množství problémů z různých oblastí matematiky a informatiky, které mohou být použity pro testování. Společně s TPTP je i udržována databáze TSTP[5] (Thousands of Solutions for Theorem Provers), která obsahuje výstupy různých dokazovačů pro problémy z TPTP. Více informací v [10].

2.3 Genetické algoritmy

Nyní zaveďme pojmy zabývající se genetickými algoritmy. Genetické algoritmy jsou inspirovány teorií evoluce od Ch. Darwina a jako první se jimi zabýval John Holland. Snaží se reprezentovat řešení nějakého problému pomocí práce s populací, která sestává z **jedinců**. Každý má z nich zakódován

[3] o faktorizaci více v [11] a paramodulaci více v [9]
[4] K nalezení na adrese `http://www.cs.miami.edu/~tptp/`
[5] K nalezení na adrese `http://www.cs.miami.edu/~tptp/TSTP/`

řešení daného problému do tzv. **chromozomu**, který je posloupností **genů**. Chromozomy mívají většinou konstantní velikost. Nejčastěji jsou geny reprezentovány binárně, používají se ale i reprezentace jiné (celá čísla, řetězce, stromy,..)[6]. U každého jedince se dá určit jeho **fitness**, což je hodnota reprezentující kvalitu jedince vzhledem k danému problému. Nad těmito strukturami jsou definovány tři operace : **selekce, křížení** a **mutace**. Selekce vybírá z populace jedince, které budou použity pro základ další generace. Selekční metody mohou sledovat různé algoritmy pro výběr jedinců, nejčastěji se vybírá dle fitness. My konkrétně budeme používat nejzákladnější ruletovou selekci, která funguje následovně:
Sečti fitness všech jedinců v populaci a tuto sumu označ S, poté opakuj N-krát (kde N je velikost populace): Vygeneruj náhodné číslo R z rozsahu 0 až S. Poté postupně procházej jedince a sčítej jejich fitness. Jakmile je součet fitness větší nebo rovno než R, zastav cyklus a jedince, u něhož jsme skončili při procházení, vrať jako výstup selekce. Ruletová selekce dostala svůj název podle své podoby s roztáčením rulety. Další často používané selekce jsou například turnajová selekce nebo SUS.
Křížení je operace, která z jedinců - rodičů (zpravidla ze dvou) tvoří potomky. V naší práci budeme používat křížení podobné jednobodovému křížení. Jednobodové křížení funguje následovně:
V rodičích A a B vyber v chromozomech náhodně pozici i ($i \leq N$, kde N je velikost chromozomu a je u obou jedinců stejná), proto nám stačí vybrat jen jeden index). Poté do potomka C vlož z A část chromozomu od 0 do i včetně a z B část chromozomu od $i + 1$ do N. Existují i jiná křížení, např. vícebodové, uniformní, apod.
Mutace je operace, která náhodně pozměňuje chromozom jedince.
Základní genetický algoritmus byl vyvinut Hollandem v 70. letech a jeho hlavní smyčka vypadá následovně: Nejprve náhodně vygenerujeme jedince do počáteční populace a určíme jejich fitness. Poté opakujeme N-krát (kde N je zamýšlený počet iterací) selekci, křížení a mutaci.
Více informací o genetickém programování v [2].
Na otázku, na jaké problémy se hodí genetické algoritmy, neexistuje přesná odpověď. Pokud však pracujeme s problémem, kde je prohledávací prostor velký, kde se funkce fitness hodně mění nebo kde nepotřebujeme najít nejlepší řešení, ale stačí nám jen řešení blížící se optimu, tak potom je velká šance, že genetické algoritmy budou vhodným nástrojem pro řešení problému.

[6] tato tématika je rozebírána v [7]

Kapitola 3

Definice v ATP

Formální důkazy matematických vět bývají velmi složité a rozsáhlé. Dokazuje-li matematik nějakou větu, zavádí si v důkazu nové výrazy nebo funkční symboly, které zastupují vybrané matematické výrazy. Důkaz se potom stává přehlednějším a kratším. Naše idea je usnadnit práci dokazovačů obdobným způsobem.

3.1 Zavedení pojmu definice

Výběr given klauzule k dalšímu kroku důkazu je v automatickém dokazování vět netriviální záležitost. Vzhledem k podstatě problému dokáže výběr klauzule velmi ovlivnit výpočet. Představme si, že množiny sos a usable vypadají následovně

$$sos = \{d_1, d_2 \dots d_n, \neg c\}, usable = \{c, e_1, e_2 \dots e_m\}$$

a chceme odvodit spor (předpokládejme, že klauzule d_i jsou nezunifikovatelné pro všechna i s klauzulí c a pro žádné i a j nevede použití odvozovacích pravidel na d_i a e_j ke sporu). Budeme-li nejprve vybírat d_1 až d_n a potom až $\neg c$, bude nalezení sporu trvat déle, než kdybychom rovnou vybrali $\neg c$. Jedna z mnoha metod optimalizace prohledávání je zavedení definic.
Def.: Nechť $Func$ je množina všech funkčních symbolů použitých v daném problému. Potom *unární definicí* funkce f nazveme rovnici

$$f(x) = t(x)$$

kde f je unární funkční symbol nevyskytující se v $Func$, t je term, v němž se

vyskytují pouze funkční symboly z $Func$ a x je jediná proměnná vyskytující se v rovnici.
Obdobně se dají definovat n-ární definice, ale my budeme v práci pracovat nejvýše s binárními definicemi. Nebude-li na četnosti definice záležet, budeme používat jen termín definice.

3.2 Vliv definice na prohledávání

Vybírání given klauzule je klíčová operace v hlavní smyčce, která rozhoduje, jak dlouho bude trvat nalezení sporu. Hlavním faktorem, který ovlivňuje vybírání klauzule, je váha termu.
Def.: Nechť $Term$ je množina všech termů v axiomech daného problému. Potom funkci w, která každému termu t z množiny $Term$ přiřazuje přirozené číslo w(t) nazveme váhovou funkcí. Hodnotu w(t) nazveme váhou termu.
Vlastní hodnoty váhy se dají definovat různě, např. v Prover9 jsou definovány následovně:

- $v(t) = 1$ pokud t je konstanta nebo proměnná
- $v(f(t_1, \ldots, t_n)) = 1 + \sum_{i=1}^{n} v(t_i)$ pokud f je n-ární funkce
- $v(\neg A) = v(A)$ pro literály A
- $v(A_1 \vee A_2 \vee \ldots \vee A_n) = \sum_{i=1}^{n} v(A_i)$

kde váha literálu se spočítá jako suma váh termů obsažených v literálu. Nyní se vraťme k problému vybírání klauzule. V příkladu jsme vybírali vždy pouze jednu klauzuli, ve skutečnosti Prover9 vybírá celé skupiny klauzulí na základě poměru nastavitelných výběrových kritérií. Mezi nejpoužívanější kritéria patří stáří klauzule a sémantická hodnota klauzule (zda má příznak true nebo false). V případě stáří klauzule se vybírají nejstarší klauzule ze sos, v případě sémantické hodnoty se vybírají nejlehčí klauzule ze sos (tj. s nejmenší váhou), které mají odpovídající příznak true/false. Máme-li tedy nastavena kritéria sémantická hodnota[true]:sémantická hodnota[false]:stáří v poměru 2:3:4, tak vybereme ze sos 9 klauzulí, z toho 2 jsou nejlehčí klauzule s příznakem true, 3 nejlehčí klauzule s příznakem false a 4 nejstarší klauzule. Je vidět, že váha klauzule je v případě sémantických kritérií rozhodujícím faktorem pro výběr klauzule.

Pokud bychom přidali do problému definici $f(x) = t(x)$, kde f je nový funkční symbol, nijak výrazně by to prohledávání neovlivnilo, protože Prover9 by nadále vybíral klauzule jako v případě bez definice. Musíme tedy změnit váhu pro tento funkční symbol, aby byla formule obsahující jako podterm fukci f přednostně vybírána. Vhodné nastavení váhy je následující[1] (v syntaxi pro Prover9):

```
list(weights).
  weight(f(_))=1.
  weight(f(x))=1000.
end_of_list.
```

Tento zápis nastavuje váhu funkce f na 1, pokud je argumentem proměnná a na 1000, pokud je tam něco jiného. Tato váha tedy zvýhodňuje term, který zastupuje f, pokud je v jeho argumentu proměnná. Znevýhodnění pro argumenty jiné než proměnné je nutné. Představme si, ze přidáme definici $f(x, y) = x + y$ k teorii obsahující pouze funkční symbol "+", tedy definice f je jen jiným zápisem pro sčítání. Potom bychom úpravou váhy pouze přenastavili váhu pro sčítání. Navíc takto nastavenou váhou máme zaručeno, že na pozicích jednotlivých argumentů budeme mít pouze proměnné (a vyhneme se možnosti, že by tam byla opět funkce f). Prover9 má pro práci s definicemi speciální volbu folding, kterou je nutné nastavit pro docílení kýženého efektu (standardně není nastaven). Toho docílíme přidáním následujícího řádku do vstupního souboru.

```
assign(eq_defs,fold).
```

Pokud je folding nastaven, bude se všech klauzulích zavádět funkční symbol f, jakmile to bude možné.
Použití definic ovšem nezaručuje zrychlení. V případě, že použiji definici, která nevede k nalezení důkazu (např. přidám definici, která zastupuje term nevyskytující se v důkazu), může se doba hledání důkazu dokonce zvětšit.
Špatně použité definice mohou dokonce způsobit vyčerpání množiny sos. Máme-li například v sos klauzule $x = y + z$ a $z = (x * y) + z$, potom bychom při zavedení definic $g(x, y) = x + y$ a $h(x, y, z) = (x * y) + z$ a nastaveném foldingu dostali $x = g(y, z)$ a $z = h(x, y, z)$. Na tyto klauzule bychom už odvozovací pravidla použít nemohli, protože termy h a g nejsou unifikovatelné.

[1] toto nastavení bylo představeno v [6]

3.3 Statistické generování definic

Vyvstává otázka, které definice použít, aby bylo nalezení důkazu dané domněnky co nejrychlejší. Nejjednodušší je nechat danou domněnku dokázat, projít důkaz a zkoušet nahrazovat termy z důkazu definicemi. V naší práci jsme se rozhodli zvolit metodu statistického generování termů. Nejprve si zaveďme potřebné pojmy.

- T bude označovat množinu klauzulí sestávající z axiomů zadaného problému a negace cíle
- $Func$ bude značit množinu všech funkčních symbolů vyskytujících se v T
- c_f bude značit počet výskytů symbolu f v T
- c bude značit počet všech funkčních symbolů v T, tedy $c = \sum_{f \in Func} c_f$

Algoritmus tvoření termů vypadá následovně: Náhodně vygenerujeme přirozené číslo n, které bude reprezentovat počet funkčních symbolů v novém termu. Náhodně vybereme funkční symbol z $Func$, přičemž pravděpodobnost, že bude vybrán symbol f určíme ze vzorce $P(\text{vybrán f}) = \frac{c_f}{c}$. U tohoto symbolu náhodně určíme jeden z jeho argumentů a rekurzivně vygenerujeme na této pozici další symbol. Na pozici ostaních argumentů necháme náhodnou proměnnou. Toto opakujeme, dokud nevygenerujeme celkem n funkčních symbolů.
Konstanty do generování neuvažujeme - tedy všechny vygenerované symboly mají četnost aspoň 1. Názorně je vidět statistické generování termů na příkladu:

- $T = \{x + y = y + x, (x')' = 1\}$
- $Func = \{+, '\}$
- $c_+ = 2, c_{'} = 2$
- $c = 4$

Za n dostaneme náhodným generováním 4. Náhodně vybereme symbol z $Func$ a vyjde nám v tomto případě +. U binárního funkčního symbolu + náhodným generováním dostaneme druhý argument a tam pokračujeme s

generováním. Další symboly, které takto vybereme, budou +,+ a $'$ a argumenty jednotlivých funkci, kde budeme pokračovat v generování, budou po řadě 2, 2 a 1. Budeme-li požadovat unární definici, dostaneme tímto způsobem novou funkci $f(x) = x + (x + (x + (x')))$. V binárním případě dostaneme například funkci $f(x, y) = x + (y + (y + (x)')$ Takto vytvořené termy sice nemusí být všechny termy použity v důkazu, ale to nám při genetickém programování nevadí.

Kapitola 4

Aplikce genetických algoritmů

4.1 Genetické křížení definic

Dokazování mnoha domněnek bývá náročné na čas - některé důkazy trvají i několik dní. Zkusili jsme tedy použít genetické algoritmy pro urychlení nalezení sporu. Představme si nyní vstupní soubor pro Prover9 jako jedince z genetického programování, a tedy celá populace (velikosti n) sestává z n souborů. U takto definovaného jedince budeme za chromozom považovat seznam definic, které daný problém obsahuje, tedy definice budou odpovdíat jednotlivým genům. Velikosti chromozomů se mohou lišit, my budeme pracovat s malými chromozomy (do 10 definic). Mutace bude probíhat standardně, tedy vybereme náhodně pozici v chromozomu a necháme definici náhodně pozměnit. Křížení na takto definovaných chromozomech bude podobné křížení jednobodovému, musíme ale náhodně vybrat pozici v obou chromozomech, protože mohou mít různou délku (narozdíl od klasického jednobodového křížení). Mějme dva jedince A,B s chromozomy $(d1, d2, d3)$ a $(d4, d5, d6, d7)$. Potom pokud při křížení náhodně vybereme pozice 2 a 4 (číslujeme od 1), vzniknou nám dva potomci C a D s chromozomy $(d1, d7)$ a $(d4, d5, d6, d2, d3)$. Jak je vidět, může se nám po křížení změnit u potomků velikost chromozomu, což ale ničemu nevadí.
Jako metodu pro selekci zvolíme klasickou ruletovou selekci.
Nyní se podívejme, jak budeme hodnotit fitness takovýchto jedinců. Protože pracujeme se soubory s domněnkami u kterých chceme při dokazování odvodit spor co nejrychleji, bude nejlepší mít fitness závislou na době důkazu. Přímá úměra (fitness=doba důkazu v sekundách) by se ale pro minimalizaci důkazu použít nedala, v úvahu připadá tedy počítat fitness jako zápornou

dobu v sekundách. Takto zvolená fitness ale špatně zobrazuje rozdíly mezi dobrým a špatným jedincem, protože výrazně nezvýhodňuje jedince, u nichž bude nalezen spor rychleji, než kdybychom nepoužili definice . Zavedeme tedy pro každý problém horní mez pro dobu hledání důkazu, kterou pokud Prover9 překročí, bude hledání ukončeno s neúspěchem. Fitness potom budeme počítat podle vzorce

$$fitness = \frac{4 * TLim}{TPro}$$

kde $TLim$ je horní mez pro důkaz a $TPro$ je doba hledání důkazu. Konstantu 4 jsme zvolili experimentálně, aby přiměřeně zvýhodnila úspěšné jedince. V takto nastavené fitness budou mít jedinci, u kterých prohledáváni skončilo neúspěchem, fitness rovnou 1. Na první pohled se zdá, že jsme si moc nepomohli, protože rozdíly mezi neúspěšným jedincem a úspěšným jedincem opět nejsou velké. Je ale třeba si uvědomit, že limitní čas pro důkaz bývá nastaven mnohem větší, než jakýkoli úspěšný důkaz, takže potom bude fitness opravdu dobře škálovatelná.
Občas se nám může stát, že Prover9 skončí, protože množina sos bude prázdná. Tato neúspěšná situace zpravidla vzniká, pokud byly na začátku zvoleny špatné definice. V tomto případě (prázdná sos) budeme nastavovat jedincům fitness 0, protože takový jedinec je naprosto nevhodný. To, že prohledávání skončilo neúspěšně kvůli dosažení časového limitu, ještě neznamená, že jde o špatného jedince. Některá z definic v chromozomu mohla prohledávání urychlit, ale ne natolik, aby skončilo do časového limitu. Proto jim dáváme fitness 1, zatímco jedincům, kteří skončili z důvodu vyčerpání sos, nastavujeme fitness 0.
Vyvstává ještě otázka, jak na začátku zvolit definice do jedinců. Jedna možnost by byla projít si důkaz dané domněnky a použít termy vyskytující se tam. My zkusíme ale využít statistického generovaní termů, které nám sice vytvoří spoustu definic sémanticky nesmyslných, ale to nám nevadí, protože takoví jedinci budou mít malou fitness.
Na takto definované populaci poté můžeme spustit samotný proces evoluce.

4.2 Implementace

Rozhodli jsme se implementovat výše popsané genetické algoritmy v jazyce C# na platformě .NET. Tento jazyk byl zvolen pro svou snadnou přenositelnost mezi operačními systémy. Pro práci s genetickými algoritmy

jsme se rozhodli použít knihovnu AForge[1], která má pro naše potřeby dostačující možnosti. Implementace byla pojmenována GenPro9. V hlavní části programu nejprve spustíme Prover9 na vstupní soubor, aby nám převedl formule z problému do klauzulí. Prover9 nemá přímo mód pro převádění formulí do klauzulí, proto si budeme muset pomoci malým trikem - přidáme do vstupního souboru volbu s nastavením maximálního počtu given klauzulí na 0 a Prover9 skončí hned po provedení klauzifikace.
Po získání klauzulí ze vstupního souboru na ně spustíme předzpracování. Prover9 má velmi volnou syntax pro zápis problémů (umožňuje zapisovat funkční symboly i v infixové formě, není striktní na závorky apod.), a proto je třeba klauzule převést do jazyka, kterému bude rozumět GenPro9. Stejně jako Prover9 máme dvě možnosti zápisu problému - standardní styl pro Prover9 a prologovský styl. Při standardním stylu musely být některé možnosti jazyka LADR omezeny, protože jejich implementace by byla obtížná s ohledem k zisku z jejich použití. Konkrétně se jedná o možnost použít písmeno "v" jako funkční symbol stejně jako proměnnou. Tedy klauzule

```
(v v v) v (v v v) = v.
```

je syntakticky naprosto v pořádku. Rozhodli jsme se proto zakázat použití symbolu v jakožto funkčního symbolu. Toto nás nijak neomezuje, protože pokud bychom pracovali s teorií, která používá v jako funkční symbol, stačilo by ve vstupním souboru změnit výskyt v za jiný nepoužitý symbol. Při použití prologovského styly zápisu podobný problém nenastal.
Po předzpracování spočítáme počty výskytů jednotlivých funkčních symbolů v klauzulích a vytvoříme počáteční generaci pro genetické algoritmy. Velikost populace, maximální velikost chromozmu a počet iterací lze nastavit před spuštěním aplikace. Jednotliví jedinci jsou vytvořeni přidáním vygenerovaných definic do vstupního souboru. Pracovali jsme s unárními a binárními defincemi.
Poté je spuštěna klasická smyčka genetických algoritmů a po stanoveném počtu iterací program skončí. Po každém kroku iterace jsou do souboru specifikovaným parametrem logFile uloženy informace o průměrné fitness populace a 3 nejlepší jedinci jsou vypsáni jako vstupní soubory pro Prover9 do stejného adresáře pro pozdější snadnou práci s nejlepšími řešeními. Název souboru s jedincem je tvořen podle šablony *název problému*_best_g*pořadí generace*_*pořadí jedince v populaci*, např pro problém GRP236-1, bude mít první jedinec šesté populace tvar GRP236-1_best_g6_1.

[1] `http://code.google.com/p/aforge/`

Kapitola 5

Porovnání výsledků

5.1 Výběr problémů na testování

Výběr třídy problémů, na kterých bychom mohli otestovat účinnost metody, nebyl jednoduchý. Nejprve jsme použili z několika vzorových problémů ze stránek Prover9 problém dpt_bw, na kterém se nám podařilo dosáhnout velkého zrychlení. Poté jsme testovali několik problémů z kvazigrup z databáze qptp od D. Stanovského[1]. Problémy z qptp se ale pro účely GenPro9 obecně nehodily, protože si je třeba uvědomit, ze proces dokazování spouštíme mnohokrát. Trvá-li například dokazování daného problému 20s, populace má 50 jedinců a chceme 50 iterací, potřebovali bychom na doběhnutí celé smyčky genetického algoritmu přibližně 13.88h (pokud bychom nastavili jako horní mez pro dokazovač 20s). Musíme tedy vybírat jen problémy, jejichž důkaz trvá maximálně několik minut. Nakonec jsme vybrali z databáze TPTP několik problémů, u kterých Prover9 standardně skončí v rozmezí 30-150s. Informace o době běhu Prover9 na problémech z TPTP jsme získali z TSTP.

Při testování jsme zvažovali distribuovat výpočet mezi více jader a tím výrazně urychlit hledání řešení, nakonec jsme ale sílu více procesorů využili pouze k testování několika problémů naráz. Použít více procesorů pro jeden problém se ukázalo nevhodné proto, že pro každý proces se spuštěným Prover9 potřebujeme stejné výpočetní kapacity, aby se daly běhy procesů porovnávat. Máme-li například 2 procesory a systémové činnosti zabírají přibližně 20% procesoru, mají poté instance Prover9 spuštěné na obou procesorech nestejné podmínky. Proto jsme spouštěli tolik instancí GenPro9,

[1] k dispozici na adrese `http://www.karlin.mff.cuni.cz/~stanovsk/qptp/`

kolik má aktuální stroj procesorů. Celé testování probíhalo na osmijádrovém serveru octopi v budově MFF UK.

5.2 Účinnost implementace

Program GenPro9 jsme testovali na 8 problémech. U každého problému jsme si nejprve z TSTP zjistili, jak dlouho dokazuje Prover9 problém bez použití definic. Tuto dobu si označme $Time$. Dále jsme na problém spustili GenPro9 s parametry nastavenými následovně:

- Velikost populace - 50 (velikost populace je konstantní v celém algoritmu)
- Počet iterací - 20
- Maximální četnost nových definic - 2 (používali jsme binární definice)
- Maximální počet funkčních symbolů použitých v jedné definici - 10 (tedy kdyby v teorii byli pouze symboly +, mohli bychom nejvýše dostat definici obsahující 10x +
- Časový limit pro Prover9 - $Time * 1,2$

Maximální velikost chromozomu (tedy počet definic ve vstupním souboru) jsme nastavili na 10. U každého jedince se tedy pohybuje počet definic v rozmezí 1 až 10. Připomeňme, že definice byly generovány pomocí metody statistického generování definic. Stejné nastavení jsme použili pro všechny testovací problémy.
V následujících tabulkách shrnujeme výsledky. Pro každý problém vypisujeme v každé iteraci průměrnou a maximální fitness v populaci. Iterací rozumíme pořadové číslo generace v genetickém algoritmu, průměrná fitness je aritmetický průměr fitness všech jedinců v generaci. Maximální fitness je nejvyšší fitness vyskytující se v populaci (tedy fitness nejlepšího jedince). Problém AKP06_ je z databáze QPTP, problém dpt_bw je ze stránek pro Prover9, ostatní jsou z databáze TPTP. Více informací o problémech (odkaz na stažení, stručný popis) je k dispozici na přiloženém disku.

AKP06_1		
Iter	Prům. fit	Max fit
1	1	1
2	1	1
3	5.70	17.77
4	13.18	17.77
5	15.84	32
6	18.84	32
7	20.81	53.33
8	24.91	53.33
9	34.46	80
10	44.89	80
11	57.33	80
12	70.78	80
13	76.26	80
14	72.10	80
15	72.18	80
16	72.53	80
17	72	80
18	72	80
19	70.4	80
20	67.73	80

GRP748-5		
Iter	Prům. fit	Max fit
1	3.23	10.16
2	12.59	50
3	21.84	50
4	37.37	54.54
5	43.81	54.54
6	50.28	54.54
7	49.88	54.54
8	49.91	54.54
9	49.72	54.54
10	50.13	54.54
11	50.39	54.54
12	51.60	54.54
13	51.83	54.54
14	51.69	54.54
15	51.23	54.54
16	51.10	54.54
17	52.45	54.54
18	52.31	54.54
19	52.41	54.54
20	51.13	54.54

Na problémech AKPO06_1 a GRP748-5 byly genetické algoritmy nejúspěšnější. U obou se postupně zlepšovala průměrná a maximální fitness. U problému AKP06_1 dokonce první dvě generace skončily dosažením časového limitu a až poté se podařilo vykřížit dobrého jedince. Vzhledem k velikosti populace a tedy velkému počátečnímu množství různých definic, bylo optimum nalezeno během deseti generací (u GRP748-5 dokonce po třech generacích). V dalších generacích už nebyl křížením ani mutací nalezen lepší jedinec.

COL036-1		
Iter	Prům. fit	Max fit
1	21.00	40
2	25.30	40
3	26.22	40
4	27.82	40
5	36.48	80
6	35.04	80
7	43.80	80
8	46.40	80
9	56.8	80
10	59.52	80
11	65.6	80
12	64.52	80
13	68.26	80
14	68	80
15	64.8	80
16	66.4	80
17	64.8	80
18	67.2	80
19	66.66	80
20	70.4	80

GRP236-1		
Iter	Prům. fit	Max fit
1	44.84	56
2	46.61	56
3	45.17	56
4	44.13	46.66
5	43.92	56
6	45.52	56
7	46.02	56
8	46.8	56
9	46.24	56
10	46.66	56
11	46.85	56
12	47.04	56
13	46.90	56
14	47.76	56
15	45.92	56
16	46.34	56
17	46.74	56
18	46.66	56
19	46.34	56
20	47.54	56

Podobná situace jako u předchozích dvou je i u problému COL036-1. Tady je zřetelně vidět, jak se v populaci postupně zvětšoval počet kvalitních jedinců (což je typická vlastnost u genetických algoritmů). U problému GRP236-1 je situace odlišná. U tohoto problému bylo velké množství definic urychlujících nalezení sporu, a protože pracujeme s velkou populací (a tedy velkým množstvím definic), byla průměrná fitness vysoká už v první generaci.

dpt_bw		
Iter	Prům. fit	Max fit
1	5.50	8.75
2	5.59	8.75
3	6.00	8.75
4	5.82	8.75
5	6.32	8.75
6	6.75	8.75
7	7.23	12.72
8	7.57	14
9	7.64	14
10	7.72	14
11	7.59	14
12	8.04	14
13	7.99	12.72
14	7.59	8.75
15	7.80	9.33
16	7.92	10
17	7.94	10
18	8.13	10
19	8.25	10
20	8.34	10

Na problému dpt_bw je vidět, že někdy mohou genetické algoritmy kvůli vlivům mutace a selekce přijít o nejlepší řešení. Proto jsme si vypisovali v průběhu algoritmu nejlepší jedince do souboru.
U problémů RNG029-5, GRP183-4 a robbinsonovy domněnky se zlepšující řešení nalézt nepodařilo a všichni jedinci skončili dosažením časového limitu. U větších problémů bývají důkazy rozsáhlé a potřebovali bychom mnohem větší výpočetní kapacitu, abychom mohli najít zlepšující definice.
U všech problémů (u kterých bylo zlepšení nalezeno) převažovali v pozdějších generacích jedinci s malým počtem definic (1-3). Podrobnější výsledky testovacích problémů jsou k dispozici na přiloženém disku.

Porovnejme ještě pro jednotlivé problémy doby hledání důkazů bez definic a doby hledání důkazu s nejlepšími definicemi.

Problém	Bez definic	Nejlepší jedinec	
AKP06_1	6,69s	2,18s	
dpt_bw	29.96s	10.52s	(jedinec je z 10. generace)
GRP183-4	84,78s	timeout	
GRP236-1	24,66s	5,24s	
GRP748-5	60,02s	11,19s	
RNG	43.05s	timeout	
robbins	4h	timeout	

Kapitola 6

Závěr

Seznámili jsme se s genetickými algoritmy, automatickým dokazováním a dokazovačem Prover9. Naimplementovali jsme program využívající genetické křížení definic, na kterém jsme testovali problémy z databáze TPTP. Pro některé problémy (nejvíce pro malé problémy) se nám podařilo najít zlepšující definice, některé urychlily důkaz až 5x. Metoda se ukázala nevhodná pro velké problémy, protože jen pro problém, u nějž nalezení důkazu bez definic trvalo desítky minut, by celý test trval týden. Zdá se, že by bylo výhodné otestovat hledání definic na širší škále problémů používajících stejnou axiomatizaci (např. některou z algebraických teorií) a ověřit, zda pro danou axiomatizaci nexistuje obecná skupina zlepšujících definic výhodná pro většinu důkazu z daných axiomů. Dalším zajímavým rozšířením by mohlo být použití genetických algoritmů i jiným způsobem, například křížením pomocných lemat. Ve všech problémech, kde genetické algoritmy našly řešení, převažovali v posledních generacích jedinci s malým počtem definic. Zajímavým rozšířením by mohlo být sledování průměrného počtu definic v jedincích, aby byla ověřena klesající tendence na počet definic v jedincích.

Literatura

[1] Apt R.K.: *From Logc Programming to Prolog*, Prentice Hall, Británie, 1997.

[2] Koza J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, USA, 1992.

[3] Mařík V., Štěpánková O., Lažanský J.: *Umělá inteligence, Svazek 1*, 3. kapitola, Academia, Praha, 1994.

[4] McCune W. Prover9, WWW stránky, `http://www.cs.unm.edu/~mccune/mace4/`, 2005-2010.

[5] McCune W.: *Solution of the Robbins Problem*, str. 1-2, preprint, 1997.

[6] McCune W.: Robbins Redux, WWW stránky, `http://www.cs.unm.edu/~mccune/ADAM-2007/robbins/`, 2007.

[7] Mitchell M.: *An Introduction to Genetic Algorithms*,str. 156-158, The Mit Press, Británie, 1998.

[8] Kelemen J. J.: *Fundamentals of Artificial Intelligence Research*, str. 151, Springer, Berlín, 1991.

[9] Robinson A., Vornokov A.: *Handbook of Automated Reasoning Volume I*, str. 373, The Mit press, Holandsko, 2001.

[10] Sutcliffe G.: *Journal of automated Reasoning Volume 43, number 4*, str. 337-362, Springer, Nizozemí, 2009.

[11] Štěpánek P., Urban J., Vyskočil J.: *Automatické uvažování*, zatím nevydáno.

[12] Švejdar V.: *Logika: neúplnost, složitost a nutnost*, Academia, Praha, 2002.

Dodatek A

Nápověda pro GenPro9

```
GenPro9 usage:  GenPro9.exe [OPTIONS] --file FILE
-------------------------------------------------
FILE: path to input file which will be optimized
         with definitions
OPTIONS:
--help,-h: shows this help
--proverPath PATH: path to binary of Prover9
--logFile LOG: path to log file. All logs will be stored
         at same dir
--verbose: prints more output informations
--popSize SIZE: size of the population
--iter ITER: number of iterations of genetic algorithm
--maxSec SEC: time limit for Prover9
--maxArity ARITY: maximum arity of new definitions
--maxFuncSymbols FUNC: maximum function symbols
         in one definition
--prologStyle: variables and constants in input files are
         written as in Prolog
```