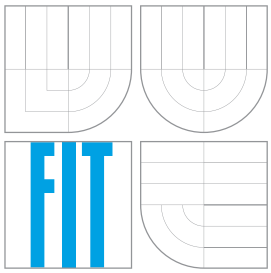
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ŘEŠENÍ PROBLÉMU SPLNITELNOSTI VÝROKOVÝCH FORMULÍ

PROJEKT DO PŘEDMĚTU EVO

AUTOR PRÁCE KAMIL DUDKA

BRNO 2008

Řešení problému splnitelnosti výrokových formulí

Zadání

Pomocí EA řešte problém splnitelnosti množiny výrokových formulí. Vstupem algoritmu je množina výrokových formulí. Výstupem ohodnocení jednotlivých booleovských poměných zaručující splnění všech výrokových formulí.

Obsah

Kapitola 1

Úvod

Úkolem bylo vytvořit program, který řeší problém *splnitelnosti množiny výrokových proměnných* pomocí *genetického algoritmu*.

Problém splnitelnosti množiny výrokových proměnných je znám v literatuře jako *SAT Problém*[1]. Jedná se o takzvaný *rozhodovací problém*, jeho přesnou definici je možné nalézt v [6]. Rozhodovací problém je zadán množinou formulí výrokové logiky. Úkolem je rozhodnout, jestli existuje takové ohodnocení výrokových proměnných, pro které jsou všechny formule splněné.

Algoritmus, který tento problém rozhoduje, spadá do třídy složitosti *NP*. Přesnou definici této třídy složitosti a odpovídající důkaz opět naleznete v [6]. Netriviální problémy spadající do této třídy složitosti není možné řešit běžnými algoritmy na současném HW v rozumném čase.

Tato práce se zabývá řešením SAT problému pomocí tzv. *genetického algoritmu*, díky kterému je možné řešit (některé) instance tohoto problému ve výrazně kratším čase, než který odpovídá teoretické složitosti. Nevýhodou tohoto přístupu je, že není zaručeno, že bude řešení nalezeno v konečném čase, pokud existuje[2].

Vytvořený program dokáže zadaný SAT problém řešit jak pomocí genetického algoritmu, tak pomocí běžného *(slepého)* algoritmu. V závěru práce jsou tyto přístupy porovnány na různých instancích SAT problému. Kompletní zdrojové kódy a programovou dokumentaci je možné stáhnout z `http://dudka.cz/fss`.

[1] Název je převzatý z angličtiny – SATisfiability problem.

[2] Experimentálně však bylo zjištěno, že při vhodně zvolených parametrech, se řešení *většinou* najde. Za určitých předpokladů lze navíc matematicky dokázat, že genetický algoritmus nalezne existující řešení v nekonečném čase.

Kapitola 2

Teoretická část

2.1 Genetický algoritmus

Citace z [2]:

> Genetický algoritmus patří mezi základní stochastické optimalizační algoritmy s výraznými evolučními rysy. V současnosti je nejčastěji používaným evolučním optimalizačním algoritmem, se širokou paletou aplikací od optimalizace vysoce multimodálních funkcí přes kombinatorické a grafově-teoretické problémy až po aplikace nazývané „umělý život“.
>
> Ukazuje se, že na základě analogie s evolučními procesy probíhajícími v biologických systémech existuje alternativní možnost, jak usměrnit náhodné generování bodů k hodnotám blízkým optimálním. Právě tato analogie se stala základem *genetického algoritmu*, který vylepšuje čistý stochastický slepý algoritmus tak, že poskytuje v reálném čase optimální řešení.
>
> Darwinova teorie evoluce se zakládá na tezi *přirozeného výběru*, podle které přežívají jen nejlépe přizpůsobení jedinci populace. *Reprodukcí* dvou jedinců s vysokým *fitness* dostáváme potomky, kteří budou s vysokou pravděpodobností dobře přizpůsobení na úspěšné přežití. Při podrobné analýze (hlavně matematické) se ukazuje, že samotné působení reprodukce není dostatečně efektivní na vznik dobře přizpůsobených jedinců s novými vlastnostmi, které významně ulehčují přežití. Do evoluce živé hmoty je nutné zapojit i tzv. *mutace*. Tyto mutace ovlivňují náhodným způsobem (kladně nebo záporně) genetický materiál populace jedinců.
>
> Biologická evoluce je progresivní změna obsahu genetické informace (*genotypu*) populace v průběhu několika generací. Zavádí se pojem *fitness*, který hraje klíčovou úlohu v úvahách o genetickém algoritmu. V biologii je fitness definovaná jako relativní schopnost přežití a reprodukce genotypu v daném prostředí. Podobně se chápe i v umělém životě, je to kladné číslo přiřazené genetické infor-

maci repreprezentující organizmus (obvykle vyjádřené pomocí bitového řetězce), které reprezentuje jeho relativní úspěšnost plnit si v daném prostředí svoje úlohy (např. sběr potravy) a vstupovat do reprodukce, t.j. tvořit nové organizmy.

2.2 Knihovna `GAlib`

`GAlib` je C++ knihovna, která obsahuje komponenty pro tvorbu genetických algoritmů. Zahrnuje nástroje pro použití genetických algoritmů k optimalizacím v různých C++ programech, které mohou využívat jakoukoliv reprezentaci a genetické operátory. `GAlib` je podporován na různých UNIX platformách (Linux, MacOSX, SGI, Sun, HP, DEC, IBM) stejně dobře jako na MacOS a DOS/Windows systémech. Knihovnu lze stáhnout z http://lancet.mit.edu/ga/.

Kapitola 3

Návrh řešení

Hlavním cílem návrhu byla znovupoužitelnost[1][3][4][5]. Většina modulů nemá žádné externí závislosti. Pokud bychom např. místo knihovny `GAlib` chtěli použít jinou knihovnu, stačilo by nadefinovat dvě nové třídy pracující se stávajícím objektovým modelem. Program je z hlediska návrhu dělen do několika částí:

- **Reprezentace problému** - modul `SatProblem`
- **Abstrakce prohledávače, observery** - modul `SatSolver`
- **Implementace prohledávače** - moduly `GaSatSolver` a `BlindSatSolver`

3.1 Vstupních data

Z matematického pohledu je instance problému definována jako množina formulí výrokové logiky. Tyto formule tedy tvoří vstupní data programu a je potřeba je programu nějak předat. K tomuto účelu byl vytvořen jednoduchý jazyk pro zápis výrokových formulí, kterému program rozumí.

Tento jazyk je podmnožinou jazyka pro zápis logických výrazů v C++. Výrokové proměnné jsou pojmenovány pomocí identifikátorů. Identifikátor je jakákoliv sekvence písmen, číslic a znaku podtržítko, přičemž musí začínat písmenem nebo podtržítkem. Dále jazyk obsahuje binární operátory, unární operátor negace a závorky. Jednotlivé formule jsou od sebe odděleny středníkem. Lexikální prvky jazyka jsou shrnuty v tabulce 3.1. Stejně jako C++ je i tento jazyk *case-sensitive*.

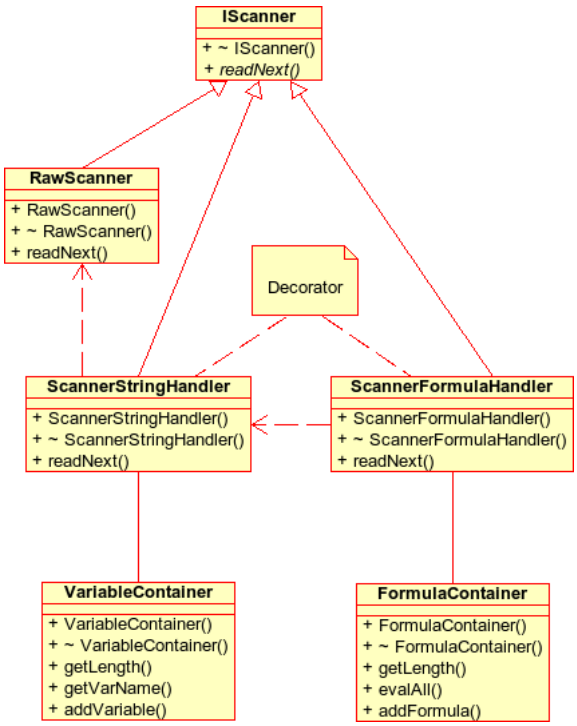
3.2 Interní reprezentace problému

Definice problému pomocí jazyka je vhodná pro uživatele, ale není příliš vhodná pro zpracování genetickým algoritmem. Navíc operace vyhodnocení jednotlivých formulí je časově kritická část výpočtu. Proto si program na základě vstupních dat vytváří interní reprezentaci problému založenou na objektech C++.

Znaková reprezentace	Alternativní reprezentace	Sémantika
&	AND	konjunkce (binární operátor)
|	OR	disjunkce (binární operátor)
^	XOR	exclusive OR (binární operátor)
~	NOT	negace (unární operátor)
( )		závorky
0	FALSE	nepravda
1	TRUE	pravda
`[a-zA-Z_][a-zA-Z0-9_]`		identifikátor
;		středník - oddělovač formulí

Tabulka 3.1: Lexikální prvky jazyka pro zápis výrokových formulí

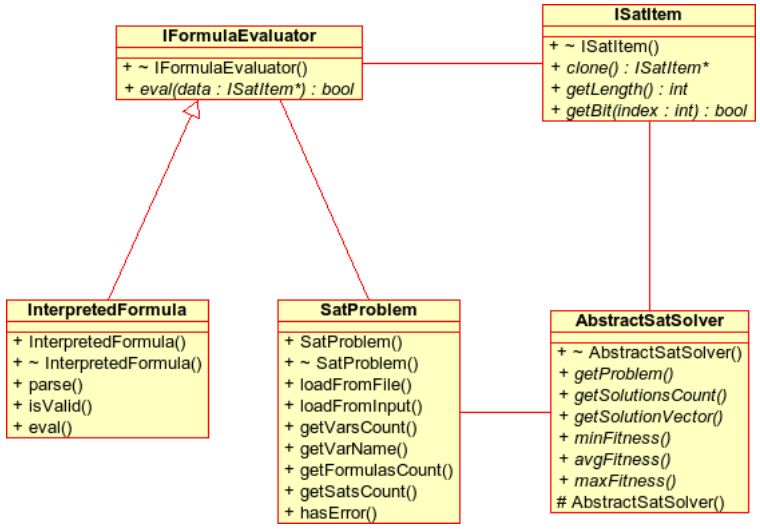
Diagram tříd, které zajišťují převod vstupních dat na interní reprezentaci, je zachycen na obr. 3.1. Na tomto diagramu je znázorněno využití návrhového vzoru *decorator*. Výsledkem zpracování vstupu je množina načtených názvů proměnných a množina načtených výrokových formulí. Tato data jsou uložena v kontejnerech `VariableContainer` a `FormulaContainer`.



Obrázek 3.1: Zpracování vstupních dat - objektový model

Načítání problému ze vstupního souboru je považováno pouze za jednu z variant, jak problém definovat. Proto je interní reprezentace dat zcela oddělena od algoritmů, které

s daty pracují. Toho je docíleno pomocí tzv. *rozhraní*[1]. Rozhraní `IFormulaEvaluator` představuje vyhodnotitelnou výrokovou formuli a rozhraní `ISatItem` její ohodnocení. Definici problému jako celek zpřístupňuje objekt *SatProblem* (návrhový vzor *Facade*). Situace je znázorněna na diagramu 3.2.



Obrázek 3.2: Interní reprezentace SAT problému - objektový model

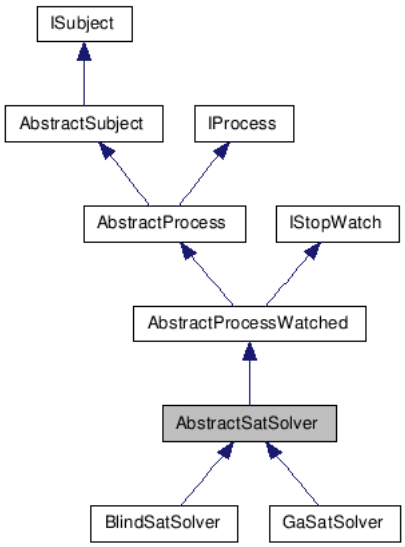
3.3 Abstrakce prohledávače

Jak bylo naznačeno v úvodu, program umožňuje řešit daný SAT problém dvěma způsoby – pomocí genetického algoritmu a pomocí slepého prohledávače. Abstraktní třída `AbstractSatSolver` definuje společné rozhraní pro oba prohledávače. Tuto třídu lze také použít jako bázovou třídu pro definici jakéhokoliv prohledávače, např. pro využití jiné knihovny než je `GAlib`.

Definice společného rozhraní je důležitá pro spolupráci s *observery*. Observery jsou objekty, které mohou sledovat aktuální stav prohledávače a nějak reagovat na změny tohoto stavu – např. tím, že vypisují informace o průběhu prohledávání, nebo vyhodnocují podmínky pro zastavení prohledávače.

Třída `AbstractSatSolver` proto implementuje několik rozhraní, které mohou observery využívat při práci s prohledávačem. Nejdůležitější z nich je rozhraní `ISubject` na samém vrcholu hierarchie dědičnosti, které umožňuje jednotlivé observery k prohledávači připojovat. Hierarchie dědičnosti je znázorněna na obr. 3.3.

[1] Rozhraní (*interface*) je z pohledu překladače C++ čistě abstraktní třída (*pure virtual*).



Obrázek 3.3: Hierarchie dědičnosti třídy `AbstractSatSolver`

3.4 Observery

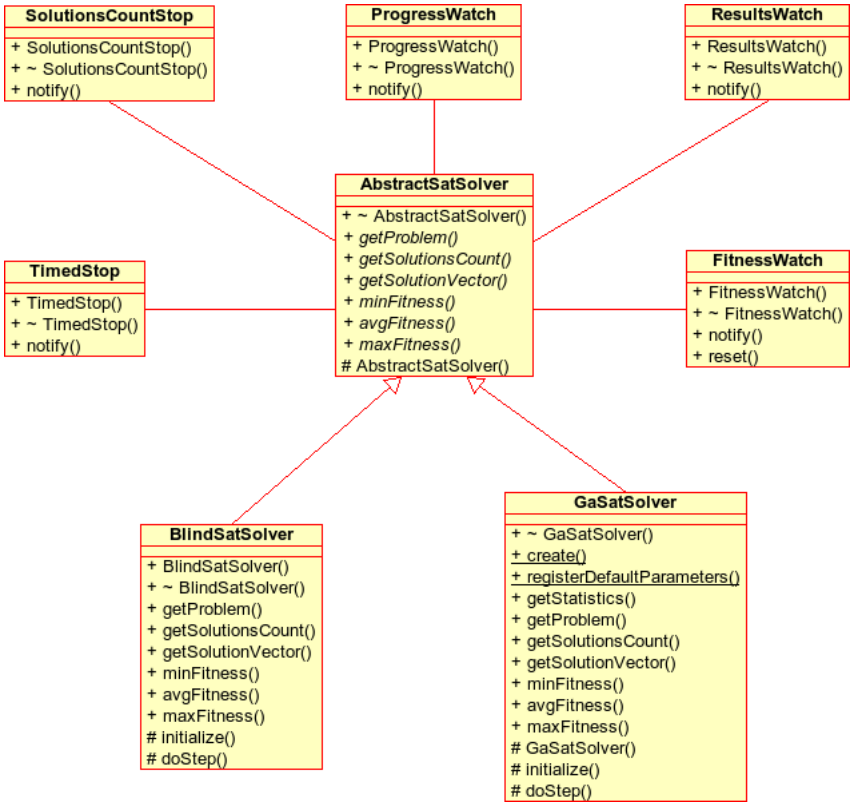
Observery jsou navrženy tak, aby byly pokud možno nezávislé na konkrétním typu prohledávače. Observery implementované v současné verzi aplikace se starají pouze o výpis potřebných informací do konzole a případné ukončení prohledávání při splnění uživatelem definovaných podmínek. Spolupráci prohledávače s observery znázorňuje znázorňuje diagram tříd na obr. 3.4. Seznam jednotlivých observerů je uveden v tabulce 3.2.

Třída	Chování
`TimedStop`	Zastaví prohledávání po uplynutí předem daného času.
`SolutionsCountStop`	Zastaví prohledávání, pokud je nalezen požadovaný počet řešení.
`ProgressWatch`	Průběžně vypisuje informace o průběhu prohledávání.
`ResultsWatch`	Průběžně vypisuje nalezená řešení.
`FitnessWatch`	Zobrazuje informace, pokud bylo nalezeno ohodnocení s vyšší hodnotou *fitness*.

Tabulka 3.2: Seznam observerů a jejich chování

Pro jednodušší práci s observery byla vytvořena C++ šablona, která observer vytvoří s požadovanými parametry a zároveň jej připojí ke sledovanému objektu:

```
// Create observer OBSERVER and attach it to subject
template <
    class OBSERVER,
    class SUBJECT,
    class ARG>
inline OBSERVER* createAttached(SUBJECT *subject, ARG &arg) {
```



Obrázek 3.4: Spolupráce prohledávače s observery

```
  OBSERVER *observer= new OBSERVER(subject, arg);
  subject->addObserver(observer);
  return observer;
}
```

Samotný zápis kódu přidávající jednotlivé observery pak vypadá mnohem přehledněji, přičemž není nijak omezeno původní rozhraní observeru. Lze tedy kombinovat oba dva způsoby připojení observeru:

```
// attach progress indicator
const int progressBits = satProblem->getVarsCount()-stepWidth;
if (progressBits > 0) {
  progressWatch = new ProgressWatch(satSolver, 1<<progressBits, std::cout);
  satSolver->addObserver(progressWatch);
}

// Display message if maxFitness is increased
fitnessWatch = createAttached<FitnessWatch>(satSolver, std::cout);
```

3.5 Jednotlivé implementace prohledávače

Jak bylo zmíněno v úvodu, program poskytuje dvě implementace prohledávače, mezi kterými lze přepínat – slepý prohledávač a prohledávač využívající genetický algoritmus. V tabulce 3.3 jsou shrnuty výhody a nevýhody obou implementací. Třídy implementující tyto prohledávače se jmenují `BlindSatSolver` a `GaSatSolver`.

Slepý prohledávač	GA prohledávač
+ dokáže nalézt všechna existující řešení	- není zaručeno, že najde všechna řešení
+ čas potřebný k prohledávání lze spočítat předem	- nelze obecně definovat podmínky zastavení
- vhodný pouze pro triviální problémy	+ dokáže řešit složité problémy na běžné arch.
- asymptotická časová složitost je exponenciální	+ většinou nalezne řešení velmi rychle, pokud existuje
- počet proměnných je omezen architekturou počítače	- pro určité třídy problémů je potřeba upravit řídící parametry

Tabulka 3.3: Srovnání výhod a nevýhod obou implementací prohledávačů

Implementace slepého prohledávače je velmi jednoduchá. Ohodnocení jednotlivých proměnných se ukládá jako celé číslo. Tím je zaručeno efektivní zpracování pomocí aritmetických a bitových operací nad tímto typem. Prohledávací smyčka provádí pouze inkrementaci tohoto čísla a vyhodnocení výrokových formulí. Počet výrokových proměnných je v tomto případě omezen přibližně na `sizeof(long)` dané architektury. To však v praxi příliš nevadí, protože pro složitější problémy je slepý prohledávač stejně nepoužitelný.

Třída `GaSatSolver` funguje jako propojení knihovny `GAlib` se zbytkem programu. Na obr. 3.5 je znázorněn graf spolupráce této třídy. Tato třída pouze implementuje rozhraní prohledávače pomocí tříd knihovny `GAlib`. Používá genetický algoritmus definovaný třídou `GASimpleGA`. Během experimentů byly porovnávány i jiné algoritmy - viz. kapitola 5. Pro použití jiného algoritmu stačí (díky kvalitnímu objektovému modelu knihovny `GAlib`) změnit jediný řádek v programu[2].

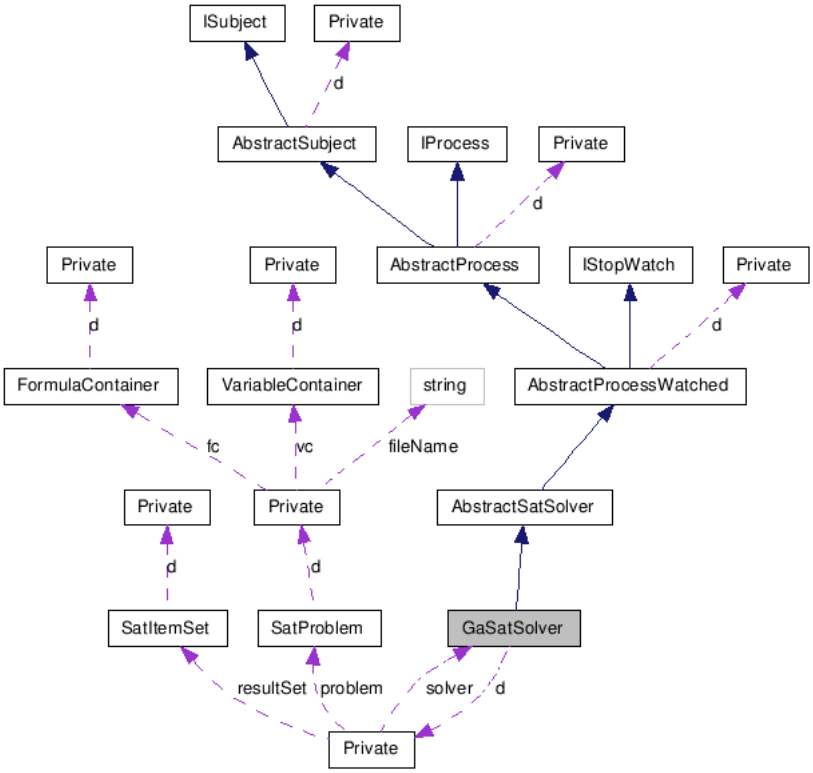
Pro kódování chromozómu byla použita třída `GA1DBinaryStringGenome`, která kóduje jedince jako jednorozměrné bitové řetězce. K zjištění hodnoty *fitness* jedince je potřeba provést nad ním vyhodnocení všech formulí – je tedy potřeba tyto objekty *obalit*, aby implementovali dříve definované rozhraní `ISatItem`. K tomu slouží třídy `GaSatItem` a `SatItemGalibAdatper`, přičemž druhá z nich pracuje pouze s odkazem na objekty knihovny `GAlib` (z důvodu optimalizace). Plnohodnotná implementace je zase potřeba pro tvoření kontejnerů (množina, vektor, ...).

Hodnota *fitness* se počítá podle následujícího vztahu:

$$fitness = \frac{satisfied\ formula\ count}{total\ formula\ count}$$

Hodnota tedy leží v intervalu $\langle 0, 1 \rangle$. 0 znamená, že není splněna ani jedna formule, 1 znamená, že jsou splněny všechny formule (nalezeno řešení).

[2]Nebyl by problém změnit program tak, aby bylo možné jednotlivé genetické algoritmy přepínat za běhu. Experimenty však ukázaly, že to samo o sobě nepřinese velký užitek.



Obrázek 3.5: Graf spolupráce třídy `GaSatSolver`

Nalezená řešení jsou ukládána vně knihovny `GAlib` pomocí kontejneru *množina* – třídy `SatItemSet`. Tento kontejner zajistí, aby každé ohodnocení proměnných (nalezené řešení) bylo v kontejneru právě jednou. Tím lze sledovat celkový počet nalezených (navzájem různých) řešení a nakonec je všechny vypsat.

Kapitola 4

Implementace

Program je implementovaný v jazyce C++, většinu jeho modulů lze kompilovat bez externích závislostí. Pouze moduly `GaSatSolver` a `fss` (modul obsahující funkci `main`) jsou závislé na knihovně `GAlib`. Spustitelný soubor se jmenuje `fss`. Přehled parametrů je uveden v tabulce 4.1. Jednotlivé parametry se zadávají vždy jako pár název—hodnota. Pokud zvolíte GA prohledávač, je možné zadat i další parametry, které zpracovává `GAlib` – tyto parametry nejsou uvedeny v tabulce. Podrobnosti naleznete v dokumentaci ke knihovně `GAlib` a její třídě `GASimpleGA`.

Název parametru	Zkrácený název	Sémantika
`input_file`	`input`	Vstupní soubor obsahující definici SAT problému. Znak „-“ představuje standardní vstup.
`color_output`	`color`	`1/0` zapíná/vypíná barevný konzolový výstup.
`verbose_mode`	`verbose`	`1/0` zapíná/vypíná kontrolní výstup knihovny GAlib.
`blind_solver`	`blind`	Přepíná mezi slepým a GA prohledávačem. 1 zapíná slepý prohledávač. 0 zapíná GA prohledávač (výchozí).
`step_width`	`stepw`	*(jenom pro slepý prohledávač)* Granularita oznamování a řízení prohledávače. Výchozí hodnota je 16.
`min_count_of_solutions`	`minslns`	Minimální požadovaný počet řešení.
`max_count_of_solutions`	`maxslns`	Maximální počet hledaných řešení.
`max_count_of_runs`	`maxruns`	Maximální počet opakování běhu GA, pokud není nalezen požadovaný počet řešení.
`max_time_per_run`	`maxtime`	Běh je bezpodmínečně přeruše pokud je překročen daný čas (v milisekundách).
`term_upon_convergence`	`convterm`	`0 ->` Běh GA je přerušen po `ngen` generacích. `1 ->` Běh GA je přerušen podle kritéria konvergence.

Tabulka 4.1: Přehled parametrů programu `fss`

4.1 Sestavení a spuštění

Pro úspěšné sestavení celé aplikace je potřeba mít nainstalovanou knihovnu `GAlib`. Program používá automatický *build-systém* `CMake`, který nalezne potřebnou knihovnu v systému, detekuje závislosti mezi moduly apod. Sestavení aplikace ze zdrojových kódů je velmi jednoduché:

```
$ cmake .
$ make
```

Nedílnou součástí dokumentace je dokumentace API, kterou lze vygenerovat pomocí programu `Doxygen` – vše potřebné je v adresáři `doc/api`. Tuto dokumentaci lze také, stejně jako zdrojové kódy, stáhnout z webu projektu http://dudka.cz/fss. Na obrázku 4.1 je ukázka použití programu.

```
$ ./fss input input.txt color 1 minslns 6 maxslns 6 popsize 100
--- Formulas count: 20
--- Variables count: 20
--- Variables: a8, a0, a19, a12, a17, a9, a1, a3, a18, a5, a7, a11, a2, a14, a4, a15, a10,
>>> Using GAlib solver
>>> Run   1 of   8
--- satisfaction: 85.0%(avg: 85.0, min: 40.0), generation     1, time elapsed:     0.01 s
--- satisfaction: 90.0%(avg: 85.0, min: 35.8), generation     6, time elapsed:     0.07 s
--- satisfaction: 95.0%(avg: 89.0, min: 37.4), generation    31, time elapsed:     0.32 s
--- satisfaction:100.0%(avg: 90.6, min: 39.3), generation    42, time elapsed:     0.44 s
--- 1. solution found in     0.44 s
--- 2. solution found in     0.51 s
--- 3. solution found in     0.70 s
--- 4. solution found in     2.48 s
<<< Found    4 solutions in   2.56 s
<<< Total    4 solutions in   2.56 s

>>> Run   2 of   8
--- satisfaction: 85.0%(avg: 85.0, min: 30.0), generation     1, time elapsed:     0.01 s
--- satisfaction: 90.0%(avg: 85.0, min: 30.0), generation     1, time elapsed:     0.01 s
--- satisfaction: 95.0%(avg: 89.7, min: 33.7), generation    15, time elapsed:     0.16 s
--- satisfaction:100.0%(avg: 92.4, min: 36.9), generation    31, time elapsed:     0.32 s
--- 5. solution found in     0.93 s
--- 6. solution found in     2.33 s
<<< Found    2 solutions in   2.33 s
<<< Total    6 solutions in   4.89 s

    1. a8=0, a0=1, a19=1, a12=0, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=0, a11=0, a2=1,
    2. a8=0, a0=1, a19=1, a12=0, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=1, a11=0, a2=1,
    3. a8=0, a0=1, a19=1, a12=1, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=0, a11=0, a2=0,
    4. a8=0, a0=1, a19=1, a12=1, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=0, a11=0, a2=1,
    5. a8=0, a0=1, a19=1, a12=1, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=1, a11=0, a2=0,
    6. a8=0, a0=1, a19=1, a12=1, a17=1, a9=1, a1=1, a3=1, a18=0, a5=0, a7=1, a11=0, a2=1,
```

Obrázek 4.1: Ukázka použití programu

Kapitola 5

Výsledky

Hlavní překážkou při testování programu byl nedostatek testovacích dat. Jinými slovy nepodařilo se mi sehnat vhodné zadání SAT problému, na kterém bych prohledávač otestoval. Vytvořil jsem si proto generátor náhodných SAT problémů – program `fss-satgen`. Tento program je také součástí archivu a je velmi jednoduchý. Jako parametry se mu zadají počet výrokových proměnných a počet formulí, které z nich má sestavit. Výstupem je množina náhodně vygenerovaných výrokových formulí zapsaných v jazyku, kterému rozumí prohledávač.

Nevýhodou takového přístupu je skutečnost, že u takto vygenerovaného problému nemáme předem informaci o tom, jestli má nějaké řešení, popř. kolik má řešení. Pro jednodušší problémy (obvykle max. 20 – 30 výrokových proměnných) je možné na problém pustit nejprve slepý prohledávač. Výstupem slepého prohledávače je přesná informace o počtu řešení. Stejný problém potom můžeme předhodit GA prohledávači a porovnat čas, který potřebuje k jeho vyřešení. Tohle byl hlavní směr, kterým se testování ubíralo.

Kromě klasického genetického algoritmu (tak jak jej implementuje `GAlib`) bylo implementováno automatické restartování běhu GA v případě, že byly splněny podmínky pro ukončení běhu a zároveň nebyl nalezen požadovaný počet řešení. Tohle nepatrné vylepšení se ukázalo jako velmi přínosné pro snížení času prohledávání.

Bylo zjištěno, že program nalezne požadovaný počet řešení mnohem rychleji, pokud má předem informaci o počtu řešení. Takového chování dosáhneme specifikací parametrů `minslns` a `maxslns` (viz. obr. 4.1). Horní ohraničení umožňuje zastavit GA při dosažení daného počtu řešení. Zatímco dolní ohraničení umožňuje restartovat běh, pokud nevede k cíli. Tyto parametry se obecně mohou lišit.

Co se týká klasických parametrů GA, výraznější zlepšení bylo dosaženo upravením velikosti populace pro některé problémy. U problémů, které měly hodně řešení, byla řešení nalezena rychleji při nastavení menší velikosti populace. U jednoho testovaného problému, který měl více než 12000 řešení, se jako optimální hodnota ukázala velikost populace dokonce jenom 30 jedinců.

Asi nejdůležitějším parametrem byl počet generací jako ukončovací kritérium běhu.

Kromě ukončení po daném počtu generací je možné ukončit běh podle konvergenčního kritéria tak, jak jej definuje `GAlib`. Nejzajímavější z pohledu uživatele je asi ukončení běhu po uplynutí předem daného času. Tohle ukončení lze kombinovat s ostatními kritérii a zaručuje ukončení výpočtu v uživatelem stanoveném čase bez ohledu na počet nalezených řešení.

Program byl testován na různých SAT problémech různé složitosti – jak počet proměnných, tak počet formulí dosahovali řádu stovek. V tabulce 5.1 je nastíněna závislost času prohledávání na použitých parametrech. Pro vytvoření této tabulky byl jako vstup použit jednoduchý SAT problém s 20ti proměnnými a 20ti formulemi – díky tomu bylo možné použít slepý i GA prohledávač.

Parametry prohledávání	Počet nalezených řešení	Doba prohledávání
bez parametru	3 – 6	1.29 s
`ngen 500`	5 – 6	2.58 s
`minslns 6`	6	1.29 – 5.19 s
`minslns 6 maxslns 6`	6	0.60 – 5.19 s
`minslns 6 maxslns 6 time 1000 maxruns 1`	0 – 6	0.60 – 1.01 s
`blind 1`	6	116.07 s
`blind 1 maxslns 6`	6	92.03 s
`blind 1 maxtime 60000 stepw 10`	4	60.12 s

Tabulka 5.1: Závislost doby prohledávání na parametrech

Kapitola 6

Závěr

Cílem návrhu bylo vytvořit znovupoužitelné komponenty pro řešení SAT problému. Tyto komponenty byly propojeny s již hotovou a odladěnou knihovnou `GAlib` a tím bylo dosaženo vysoké efektivity řešení SAT problému pomocí genetického algoritmu. Za nejvýznamnější výsledky práce považuji srovnání běžného postupu řešení s GA z hlediska časové náročnosti a použitelnosti pro daný problém.

Ukázalo se, že pomocí GA je možné řešit problémy, které by jinak nebyly vůbec řešitelné na současném HW. Kromě toho vznikla plnohodnotná aplikace, která umožňuje SAT problémy řešit. Nezanedbatelný je i můj osobní přínos – seznámení se s knihovnou `GAlib`, která má mnohem širší obor využití, než kterým se zabývá tato práce.

Literatura

[1] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1997, ISBN 0-201-63361-2.

[2] Kvasnička, V.; Pospíchal, J.; Tiňo, P.: *Evolučné algoritmy*. 2000, ISBN 80-227-1377-5.

[3] Pecinovský, R.: *Návrhové vzory*. 2007, ISBN 978-80-251-1582-4.

[4] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley, special edition vydání, 1997, ISBN 0-201-88954-4.

[5] Sutter, H.; Alexandrescu, A.: *C++ 101 programovacích technik*. Zoner Press, 2005, ISBN 80-86815-28-5.

[6] Češka, M.; Vojnar, T.; Smrčka, A.: Teoretická informatika - Studijní opora. https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/oporaTIN.pdf, 2007.