Semestrální práce

Hierarchický SAT solver

Vypracoval: Antonín Lejsek
Datum vypracování: 25. února 2005

1 Obecně o SAT

SAT zkracuje název „Satisfiability problem“ = problém splnitelnosti. Tím, co se má splnit je booleovská formule (funkce). SAT solvery se obvykle zaměřují na CNF-SAT = problém splnitelnosti booleovské formule v konjunktivní normální formě. Příklad takové funkce:

$$f(\boldsymbol{x}) = (x_0 \vee x_3 \vee \overline{x}_5 \vee x_6) \wedge (x_2 \vee x_5)$$

Je možné se setkat i s názvem 3SAT. Jedná se o CNF-SAT, kde všechny klauzule mají tři proměnné. Příklad takové funkce:

$$f(\boldsymbol{x}) = (x_0 \vee x_3 \vee \overline{x}_5) \wedge (x_2 \vee \overline{x}_3 \vee x_5) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$$

Všechny možnosti patří mezi NP-úplné problémy a jsou tudíž na sebe vzájemně převoditelné v polynomiálním čase(=karp redukovatelné). Polynomiální znamená v tomto případě vzhledem k délce vstupu, což nemusí obecně znamenat polynomiální vzhledem k počtu proměnných. Celkově se zdá (viz [1]), že převod do CNF má víc nevýhod než výhod. Způsobuje prodloužení zápisu funkce, ztrátu informace o struktuře a zavádění nových (redundantních) proměnných. Nabízí sice normalizovaný tvar úloh, ale tato výhoda není dostatečná.

Uvedu malou ukázku. Máme třívstupové XOR hradlo, matematicky zapsáno $x_0 = (x_1 \oplus x_2 \oplus x_3)$. Po převodu do CNF vypadá v nejlepším případě takto:

$$(x_0 \vee x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_0 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_0 \vee x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_0 \vee \overline{x}_1 \vee x_2 \vee x_3) \wedge$$

$$(\overline{x}_0 \vee x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_0 \vee \overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_0 \vee x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_0 \vee \overline{x}_1 \vee x_2 \vee \overline{x}_3)$$

Přitom stačí vytvořit samostatný blok n-vstupový XOR, jehož výpočetní náročnost bude stejná, jako obyčejné klauzule, a bude to vypadat takto:

$$(x_0, x_1, x_2, x_3)_{XOR}$$

Je sice pravda, že XOR je nejhůře převoditelná funkce (délka CNF roste exponenciálně), na druhou stranu samotný převod do CNF není triviální záležitost a délka zápisu (a počet proměnných) obvykle podstatně naroste.

2 Co se prakticky řeší

Popud k rozvoji tohoto odvětví přišel z praxe. CNF-SAT je teoretická abstrakce a většina problémů se na něj musí převést. Největší třídou je ověřování logických obvodů. Vzhledem k převoditelnosti úloh lze SAT teoreticky použít k rozvrhování, automatickému odvozování, počítačovému vidění, řízení databází, návrhu integrovaných obvodů, návrhu sítí… Prakticky jsem ale nenašel, že by někdo vystavoval kvanta instancí pocházejících z praxe.

Asi největší seznam instancí, teoreticky zahrnující i instance ze známé soutěže SAT contest [4] (prakticky to ale autoři zatím slíbili a neprovedli) je na stránkách [2]. Několik dalších třeba tady [3]. Instance jsou generované náhodně, nebo řeší teoretické problémy – faktorizace čísel, barvení grafu, kódování Hanoiských věží…

Instance jsou kódovány docela jednotně. V souboru je seznam čísel; co číslo, to proměnná, co řádek (a/nebo nula), to klauzule. Záporná čísla označují negaci. Příklad:

`13 15 -124 80 0` se převede na $\rightsquigarrow (x_{13} \vee x_{15} \vee \overline{x}_{124} \vee x_{80})$

3 Princip (hierarchického) SAT solveru

V principu není SAT solver nic jiného než algoritmus na prohledávání stavového prostoru do hloubky s ořezáváním 100%-ně jistě neperspektivních větví. Na stavový prostor se můžeme dívat jako na strom, v každém patře se přiřazuje jedna proměnná. Netriviální tvar stavového prostoru zajišťují okrajové podmínky. V klasickém pojetí jsou to klauzule, v hierarchickém bloky. Vstupem SAT solveru jsou assumptions = pevně daná ohodnocení některých proměnných. Výstupem jsou hodnoty některých dalších proměnných, případně hlášení o nesplnitelnosti.

Blok pro každé (i neúplné) ohodnocení proměnných musí říct, jestli je konfliktní nebo ne. Toto hodnocení musí být monotónní, což v praxi nastává úplně přirozeně. Pro řešení SAT solverem je výhodné, aby detekce kolize nastala co nejdříve, při co nejmenším počtu ohodnocených proměnných. Jinými slovy jde nám o maximální implikativitu bloku.

Druhou vlastností bloku, kterou lze považovat za základní je schopnost odvození. Jestliže blok reprezentuje klauzuli $(a \vee \overline{b})$, lze z přiřazení a=0 odvodit b=0. Podobně blok reprezentující sčítačku může odvodit součet ze znalosti sčítanců. Odvození proměnné blokem nesmí způsobit, že ohodnocení proměnných bude pro tento blok konfliktní (opět přirozené pravidlo). Tím se nám ohodnocené proměnné dělí na dvě (tři) skupiny

1. Ty, které přiřadil SAT solver.
2. Ty, jejichž ohodnocení bylo odvozeno.
3. Assumptions – jsou tu pro úplnost, nelze je nijak měnit, jinak se neliší od bodu 1. Jedná se o ohodnocení proměnných předložená SAT solveru jako fakt.

Backtracking znamená vracení se při prohledávání stavového prostoru. Zjevně je nutné vracet se jen po proměnných ze skupiny 1(případně 3, ale v tom případě už jsme vyčerpali všechny možnosti a assumptions měnit nemůžeme, takže skončíme s tím, že úloha nemá řešení). Říká se tomu nechronologický backtracking. Příklad backtrackingu:

$$(a \vee b) \wedge (a \vee \overline{b} \vee c) \wedge (a \vee \overline{c})$$

Přiřadím a=0. Odvodím b=1, c=1. Pak odvodím c=0. To je konflikt, vracím se k nejbližší proměnné, kterou přiřadil SAT solver, což je „a“ a zmením ji na a=1. Zjevně všechny klauzule jsou splněny. Dále už by se jen přiřadilo b=0, c=0 a máme řešení. Může se zdát, že jsme neodhalili víceznačnost řešení a tím se ochudili. Ale praktické instance mají málokdy víc než jedno řešení a další řešení můžeme najít tak, že budeme simulovat konflikt a hledat dál.

Pořadí proměnných plynoucí ze zadání nemusí být pro řešení optimální. Proto se proměnná k ohodnocení většinou volí dynamicky, podle nějaké heuristiky. Například se vezme ta, která má nejvyšší aktivitu co do počtu kolizí.

Existuje možnost, jak cachovat a zobecňovat kolize a umožnit tak odvozování a detekci kolizí ještě dřív, než to udělají samotné bloky. Příklad (ta první závorka je sčítačka):

$$(a + b + c = ps) \wedge (x \vee y \vee a) \wedge (j \vee b)$$

Předpokládejme, že už máme určeno s=0, p=0, x=0, y=0, a=1, j=0, b=1. Nastane kolize (ne že by nemohla nastat dřív, ale předpokládejme, že implementace sčítačky to

dřív zjistit nedokázala). Požádáme sčítačku, aby stanovila příčinu kolize a získáme a=1, p=0, což je podmnožina ohodnocených proměnných daného bloku. Příčinu kolize tedy můžeme napsat jako $(\overline{a} \vee p \vee \overline{b})$, tomuto zápisu se říká konfliktní klauzule. Díky tomu nemusíme příště až budeme mít určeno a=1, p=0 zkoušet „j“, ale odvodíme b=0. Toto bylo cachování, ale lze jít ještě dál. „a“ i „b“ jsou odvozené proměnné a lze požádat bloky, které je odvodily, o příčinu odvození. Příčina pro a=1 byla x=0 a y=0. Příčina pro b=1 je j=0. Substitucí do konfliktní klauzule získáme $(x \vee y \vee p \vee j)$. Získali jsme nový vztah, který zesílí naše odvozovací schopnosti. Navíc můžeme zlepšit backtracking. Můžeme se vrátit nejen k posledně určené proměnné, ale k posledně určené proměnné obsažené v konfliktní klauzuli. Teoreticky by šlo ptát se po příčinách ohodnocení ještě dál až do chvíle, kdy by v konfliktní klauzuli byly jen proměnné určené SAT solverem, ale toto se v praxi kvůli vysoké režii neosvědčilo. Minisat (viz. dále) dokonce nenahrazuje žádnou proměnnou určenou před poslední neodvozenou (tedy SAT solverem naposledy určenou) proměnnou.

Učení nám dává možnost použít další fintu – restartování. Na začátku vyzkoušíme nakousnout několik různých větví. Naučíme se konfliktní klauzule, které se nám můžou hodit. Nebo pokud se v některé větvi příliš zahrabeme, můžeme to zkusit jinde.

3.1 Volba ohodnocované proměnné

Jednoduchá a účinná strategie je vést každé proměnné čítač odrážející její aktivitu. Pokud se proměnná ocitne v konfliktní klauzuli, přičteme jí k čítači kladné celé číslo. Pokud hodnota čítače u některé proměnné dosáhne limitu rozsahu, všechny čítače vydělíme číslem větším než 1.

Zajímavá modifikace spočívá v náhodném zvolení proměnné v určitém malém množství případů (1%).

Změna ohodnocení proměnné je oblíbeným nástrojem ladění SAT solveru a osvědčily se i velice agresivní metody (metody s velkým stárnutím informace o aktivitě).

3.2 Vlastnosti bloku

Všechny jsme je už vyjmenovali, ale je dobré je shrnout.

1. Rozpoznání konfliktu (monotónní).
2. Odvození nového ohodnocení.
3. Stanovení příčiny kolize.
4. Stanovení příčiny odvození.

Blok by měl rozpoznat kolizi a odvození co nejdříve, ideálně při nejmenším možném počtu ohodnocených proměnných. Mít maximální odvozovací schopnost. To je ale často možné jen za cenu neúměrných paměťových nároků.

3.3 Redukce konfliktních klauzulí

I když naučené konfliktní klauzule zmenšují počet zbytečně vyzkoušených ohodnocení, mají dvě nevýhody – zabírají paměť a zdržují ověřování bezkonfliktnosti. Proto je nutné jejich počet redukovat. Strategie je obdobná jako u volby, kterou proměnnou ohodnotit.

Každá klauzule má čítač počtu konfliktů. Pokud se zúčastní konfliktu, čítač se zvýší. Časem se všechny čítače snižují. Pokud počet klauzulí přesáhne nějakou mez, jejich počet se zredukuje. Optimální limit počtu klauzulí lze těžko určit, obvykle se volí na začátku malý (srovnatelný s počtem klauzulí zadání) a při restartování se zvětšuje spolu s limitem konfliktů, po kterém nastane další restart.

Ještě zbývá zmínit, že existují klauzule, které v daném okamžiku nemůžeme odstranit. Jsou to ty, které odvodily některé ještě platné ohodnocení proměnné. Bez nich bychom nebyli schopni (v případě potřeby) stanovit příčinu odvození pro tyto proměnné.

4 Implementace

Efektivní implementace je podmínkou úspěchu. Nejvíc času spotřebuje propagace. Tedy po ohodnocení proměnné zjistit, jestli nastal konflikt nebo odvození. Výhodné je řešení, kdy každá proměnná má svůj watch list – seznam bloků, které při její změně můžou způsobit konflikt nebo odvození. A ještě lépe pokud má seznamy dva – pro případ, že se její hodnota bude true a pro případ, že bude false.

Každý blok se do těchto seznamů zapíše podle svého uvážení. Díky tomu třeba blok, který reprezentuje klauzuli, potřebuje jen dva zápisy bez ohledu na počet proměnných, které obsahuje. Jistě totiž nic neodvodí ani u něj nenastane konflikt dřív, než bude ohodnocena jeho předposlední proměnná.

Kvůli backtrackingu by každá proměnná měla udržovat seznam bloků, které je třeba obnovit při rušení ohodnocení této proměnné.

4.1 Implementace bloku

Velkou výhodou hierarchického SAT solveru oproti běžně používanému je možnost implementovat kusy problému kompaktně a efektivně v bloku. Implementaci každého bloku lze ušít na míru konkrétnímu problému. Existuje i několik univerzálních a přesto zajímavých řešení.

Zjevně může být blok složen z dalších bloků. Tato myšlenka vypadá v kontextu názvu (hierarchický SAT solver) přitažlivě. Problém je, že aby tohle mělo smysl, musel by se nadřazený blok umět učit, implementovat watch listy, a to vše lépe než hlavní SAT solver. Pokud bychom toho ale dosáhli, pravděpodobně budeme schopni vylepšit i samotný SAT solver na tuto úroveň. To je důvod, proč jsem tuto možnost neimplementoval.

Některé funkce (xor, klauzule...) lze efektivně implementovat tím, že budeme používat minimální nutný počet watch listů.

Jednoduchá, paměťově náročnější, ale pro menší bloky perfektní je matice všech přípustných kombinací proměnných. Jednoduchým porovnáním aktuálně ohodnocených proměnných s touto maticí lze vyvodit konflikt nebo odvození.

BDD (binární rozhodovací diagramy) jsou kompaktní formou předchozího. Průchod diagramem podle ohodnocení proměnných udává, jestli nastal konflikt nebo ne. Trochu trikově lze zařídit, že z diagramu získáme konflikt nebo odvození i při částečném přiřazení proměnných. Algoritmus je poněkud delší, zájemci ho najdou v [6].

Blok komunikuje se svým okolím funkcí, která zjišťuje konflikt/odvození. Výstupem je tedy klauzule nebo logická hodnota. Vstupem je seznam proměnných spolu s jejich seznamy, kam se může blok registrovat.

4.2 Implementace klauzulí

Klauzule jistě potřebovat budeme. Klauzule musí obsahovat seznam svých proměnných. Pro detekci kolize nebo odvození stačí, když bude zapsána ve dvou watch listech, a to dvou proměnných s opačnou logickou hodnotou, než jsou dvě proměnné v klauzuli.

Například pro klauzuli $(x_0 \vee x_3 \vee \overline{x}_5 \vee x_6)$ se zapíšeme do watch listů třeba $\overline{x}_0$ a x_5. Dojde-li pak k přiřazení $\overline{x}_0 = true$, můžou nastat dvě možnosti:

1. Některá proměnná, třeba x_3 je neurčená nebo je true. Potom klauzuli smažeme z watch listu pro $\overline{x}_0$ a přidáme ji do watch listu pro $\overline{x}_3$.

2. $x_3 = x_6 = false$. Potom buď je $\overline{x}_5 = true$, tato situace je nezajímavá. Nebo $\overline{x}_5$ není přiřazeno, pak dojde k odvození $\overline{x}_5 = true$. Nebo $\overline{x}_5 = false$, pak dojde ke konfliktu.

5 Detaily implementace

Vzhledem k popularitě (a množství už implementovaných) SAT solverů by bylo zbytečné implementovat vše od začátku, většina kódu by stejně nepřinesla nic nového. Zvolil jsem jako základ minisat od autorů Niklas Een a Niklas Sörensson, zjednodušenou verzi solveru Satzoo, vítěze ročníku 2003 výše zmiňované soutěže [4]. Oba SAT solvery lze najít na webové stránce [5]. Tento SAT solver je licencován jako LGPL.

Výhodou tohoto projektu je relativně malý rozsah (přibližně 1600 řádků včetně prázdných a komentářů), nejlepší SAT solvery mají běžně >5000 řádků. Díky tomu ho lze z větší části pochopit v relativně krátkém čase. Přitom podle autorů je srovnatelný s pokročilými, složitými SAT solvery, neboť obsahuje nejdůležitější triky ze Satzoo. V neposlední řadě je na webových stránkách i popis.

Asi stojí za to si tento popis přečíst. Z vlastní zkušenosti bych řekl, že jsou zde dobře popsány obecné principy (které využívá většina SAT solverů), ale udělat si ucelený obrázek o implementaci je obtížné a tady je nutno sáhnout ke zdrojovému kódu. Jak se nakonec ukázalo, minisat umožňuje „pseudo-boolean constraints“, což není nic jiného, než bloky v našem hierarchickém SAT solveru. Pro správnou implementaci nových bloků je třeba znát samozřejmě rozhraní těchto bloků, vstupní a výstupní proměnné a podmínky. Rozhodl jsem se nakonec i pro základní popis celého minisatu. Jednak je dobré mít přehled při ladění, a jednak jsem narazil na některé drobné záludnosti.

Třída lbool - rozšířený boolean, obsahuje proměnnou value typu int. Jeho hodnoty jsou $-1 = false, 0 = unknown, 1 = true$.

Typ Var - je to proměnná. Ve skutečnosti je to integer a platí $0 = x_0, 1 = x_1, 2 = x_2 \ldots$ Speciální konstanta je $-1 = var\_Undef$.

Třída Lit - obsahuje integer x, jehož hodnota je $0 = x_0, 1 = \overline{x}_0, 2 = x_1, 3 = \overline{x}_1 \ldots$ Kromě konverzí vrací i hodnotu x ve funkci index. Speciální instancí této třídy je lit_Undef ($x = -2$) a lit_Error ($x = -1$).

Třída vec - šablonovaná třída, vektor, který může dynamicky měnit velikost. Pozor, používá realloc.

Funkce sortUnique - stabilně seřadí, vypustí duplicity.

Třída Queue - šablonovaná třída postavená nad vec<T>. Odebíráním se z ní neuvolňuje paměť, takže je ji třeba vždy vymazat. Je využita pro frontu proměnných čekající na propagaci.

Třída VarOrder - šabonovaná třída využívající dva vektory. Varpos lze indexovat proměnnou a vrátí pozici proměnné v prioritní frontě. Order vrací proměnnou, která je na daném místě v prioritní frontě. Třída se stará o udržování aktuálnosti těchto informací při změně aktivity proměnných. Last_var vrací index naposledy vybrané proměnné (třída zajišťujě i výběr). Pozice už vybraných proměnných se ve frontě už nemění.

Třída Constr - abstraktní třída, od ní budeme odvozovat naše bloky. Zahrnuje tyto funkce:

```
void remove (Solver& S, bool just_dealloc = false)
```

Žádost o uvolnění paměti alokované blokem. Proměnná dealloc označuje, že není třeba provádět odhlášení proměnných z watch listů (voláno na konci práce).

```
bool propagate (Solver& S, Lit p, bool& keep_watch)
```

Oznamuje bloku, že byla přidělena proměnná p. Toto volání zařídí solver, pokud jsme ve watch listu této proměnné. Pokud si přejeme odhlásit se z tohoto watch listu, nastavíme keep_watch na false. Pokud si přejeme přihlásit se do jiného, učiníme to sami přes proměnnou S.

```
bool simplify (Solver& S)
```

Oznamuje, že aktuální hodnota proměnných se už do konce řešení nezmění, u některých typů bloků (například klauzule, i když úspora není v tomto případě příliš významná) můžeme provést zjednodušení.

```
void undo (Solver& S, Lit p)
```

Kromě watch listů existují i undo listy, kam se může blok zapsat a při rušení ohodnocení proměnné p se tato funkce zavolá. Tyto listy jsou společné pro proměnnou a její negaci.

```
void calcReason(Solver& S, Lit p, vec<Lit>& out_reason)
```

Tato funkce zajišťuje dvě věci. Pokud p=lit_Undef, musí vrátit příčinu konfliktu. Pokud je p korektní proměnná, vrací příčinu odvození této proměnné. Obojí vrací ve formě vektoru literálů. SAT solver si sám hlídá, který blok odvodil kterou proměnnou.

```
bool constr_new(Solver& S, const vec<int>& ints, Constr*& out_clause)
```

Účelem této funkce je z vektoru ints, přečteného ze vstupního souboru, vytvořit blok. Interpretace tohoto vektoru je závislá na typu bloku.

```
const char* getType()
```

Tato funkce vrací jméno bloku. Například pro klauzule řetězec „clause“.

Třída Clause - potomek třídy Constr, implementuje klauzuli.

Třída solver - hlavní třída. Zajímají nás hlavně (stručně uvedu v kódu dobře komentované) verbosity, model, constrs, learnts, ok, watches, undos, cla_decay, var_decay, add_unit.

`enqueue` – touto funkcí můžeme požádat SAT solver, aby ohodnotil proměnnou, v případě, že toto ohodnocení odvodíme.

`value(Var x)` – vrací ohodnocení proměnné (pokud není ohodnocena tak unknown)

`value(Lit p)` – vrací (pozor na to) hodnotu true v případě, že proměnná odpovídající literálu p je ohodnocena stejně jako p. Pokud je ohodnocena opačně, vrací false, jinak unknown.

Ostatní nás dovolím si tvrdit zajímat ani moc nebude. Sice třeba analýza konfliktu je zajímavá, ale z dostupných informací mě zlepšení nenapadá a bloky ji nijak přímo neovlivňují.

5.1 Implementované bloky

Původní SAT solver implementoval pouze řešení problému v CNF. Rozšířil jsem načítání ze souboru o další typy bloků. Tvar vstupního souboru je teď

```
c toto je
p ukázkový soubor
1 -2 3 0
xclause 1 -2 3 0
xxor 2 -4 5 0
xatmost 1 2 -3 5 0
xmatrix 3 1 2 3 1 3 3 3 1 3 2 1 1 0
```

První dva řádky jsou komentáře, začínají písmeny "p" nebo "c". Třetí a čvrtý řádek obsahuje dva rovnocenné zápisy stejné klauzule. Předposlední tři řádky jsou nové bloky.

Pro snažší přidávání dalších bloků jsem vytvořil třídu Constr_reg. Tato třída poskytuje dvě veřejné funkce.

```
bool parse_DIMACS(FILE* in, Solver& S)
```

Tato funkce zapouzdřuje veškeré parsování textu.

```
void registerConstr(Constr* c)
```

Touto funkcí je třeba zaregistrovat každý nový blok (Constraint) a třída Constr_reg pak už automaticky řeší vztahy mezi bloky, parsovaným textem a SAT solverem.

AtMost – tento blok zařídí, že nejvýše n jeho literálů je pravdivých. Například xatmost 1 2 -3 5 0 ve stupním souboru znamená, že nejvýše 1 literál z $x_2, \overline{x}_3, x_5$ je pravdivý. Toto byl ukázkový příklad v minisatu, jen jsem jej vylepšil a opravil chybu.

XorConstr – implementace n-vstupové funkce podobným stylem jako klauzule. Je třeba si uvědomit fakt, že tento blok se musí registrovat do čtyř watch listů – pro dva literály a zároveň jejich negace.

MatrixC – implementuje tabulku všech pravdivostních hodnot logické funkce. Například pro dvoubitovou sčítačku vypadá takto:

a	b	c	z	y
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tato reprezentace je náročnější na paměť. Zároveň i na výpočet, protože každý literál musí být ve dvou watch listech. A aby toho nebylo málo, při každé změně některého literálu se porovnávají všechny řádky s vektorem už ohodnocených proměnných. Z řádků, které nejsou s tímto ohodnocením v konfliktu, se udělá průnik a výsledkem jsou (někdy) nová odvozená ohodnocení proměnných. Největší výhodou tohoto řešení je maximální dokazatelnost (což je intuitivně zřejmé) a schopnost snadno postihnout jakkoli nepravidelné funkce.

Ve vstupním souboru je jako první číslo uloženo množství proměnných. Za ním následují jména proměnných a nakonec je tabulka. Logické hodnoty jsou uloženy takto: 1=false, 2=don't care, 3=true. Sčítačka nad proměnnými $x_1, \overline{x}_2, x_3, x_4, x_5$ bude zapsána takto:

```
xmatrix 5
1 -2 3 4 5
1 1 1 1 1
1 1 3 3 1
1 3 1 3 1
1 3 3 1 3
3 1 1 3 1
3 1 3 1 3
3 3 1 1 3
3 3 3 3 3
0
```

5.2 Spouštění programu

Program používá konzoli, syntaxe vstupního řádku je

`<jméno programu> <vstup> [-v <level>]`

<jméno programu> – standardně je to minisath.exe

<vstup> – jméno souboru, ze kterého je načteno zadání úlohy.

<level> – číslo, určuje množství informací, které program poskytuje. Při level=0 (defaultní hodnota) vypisuje jen základní souhrnné informace, při level=2 vypisuje i veškeré detaily postupu při řešení.

6 Výsledky

Program jsem psal a ladil pod vývojovým prostředím MS Visual C++ 7 (v licenci pro školní použití), takže zde je možnost využít přímo projekt. V ostatních překladačích je třeba poradit si podle jeho specifických konvencí, například v unixu bude v mnoha případech fungovat g++ -ggdb -D DEBUG -O3 Main.C Solver.C Constraints.C VarOrder.C Parsing.C -o minisath, případně nějaká variace s vyšší optimalizací.

Program je konzolový a používá minimum vnějších knihoven (cstdlib, cstring, apod., viz. global.h). Překladač sice vygeneruje množství warningů, ale těmi se nemusíte vzrušovat, jsou způsobeny snahou o efektivní kód (autoři minisatu to ověřovali měřením, tyto drobné prohřešky opravdu nejsou samoúčelné).

Co se týká testovacích dat, v projektu jsou obsaženy problémy v CNF, od malých až po rozsáhlé, a několik drobných testů pro specifické případy. Těmi lze ověřit činnost všech implementovaných bloků a sledovat způsob, jakým SAT solver postupuje při řešení. Rozsáhlejší úlohy neklauzálního tvaru se mi objevit nepodařilo. Podle informací, které jsem objevil, ti, kdo je potřebovali, si je generovali sami. Tento přístup jsem zavrhnul, protože bych musel nastudovat převod problému do CNF, což pokud má být provedeno kvalitně, znamená značný problém samo o sobě.

Nejlépe by bylo objevit v okolí někoho, kdo tento typ úloh potřebuje prakticky řešit a vývoj dalších bloků odlaďovat přímo pro ten typ úloh. V CNF už nejsou žádné informace, které by šly zužitkovat a vše se redukuje na řešení soustavy klauzulí. Do té doby, než uděláme zásadnější zásahy do samotného SAT solveru, je efektivita přesně rovna efektivitě výchozího minisatu.

7 Závěr

Moderní SAT solvery jsou už velice sofistikované algoritmy, tomu bohužel odpovídá i implementační náročnost. Základní SAT solver minisat byl úspěšně rozšířen o další typy bloků a možnost jednoduššího dalšího rozšiřování tímto směrem. Přínos nového zlepšení ale nebyl z důvodu nedostatku potřebných úloh otestován v plné míře.

Reference

[1] *Toby Walsch: Solving non-clausal formulas*
http://www.cs.toronto.edu/ horst/cogrobo/presentations/NCDPLL.ppt

[2] *Knihovna SAT instancí*
http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html

[3] *Několik dalších instancí*
http://www.is.titech.ac.jp/ watanabe/gensat/

[4] *Soutěž SAT solverů*
http://www.lri.fr/ simon/contest/

[5] *Niklas Een, Niklas Sörensson: Satzoo, minisat*
http://www.cs.chalmers.se/ een/Satzoo/

[6] *Yakov Novikov, Raik Brinkmann: Foundations of Hierarchical SAT-Solving*