SATIVA Project
Report

SAT solver implementing CDCL

Laurea magistrale
in
Ingegneria e scienze informatiche

Nicola Dessí VR409182

Dipartimento di Informatica
Universitá degli studi di Verona

Introduzione

Verrá presentato il progetto di SATIVA, un solutore sat (CDCL) in logica proposizionale. Il solutore risolve problemi di soddisfacibilitá/insoddisfacibilitá di formule CNF booleane, quindi riceve in input formule del tipo

$$\phi(l_1, ..., l_n) = C_1 \wedge C_2 \wedge ... \wedge C_k,\ C_i = (l_j \vee ... \vee l_g) \quad (1)$$

e ritorna soddisfacibile/insoddisfacibile con il rispettivo assegnamento/prova. Questo problema é **NP completo**, di conseguenza una ricerca esaustiva che prova tutte le possibili combinazione non é accettabile, per questo si deve ricorrere a strategie ed euristiche che permettano di fare *pruning* nello spazio di ricerca.
In particolare si é utilizzato la procedura *CDCL* che utilizza i conflitti trovati come guida per le successive scelte con l'intento di non ripetere le stesse *'scelte sbagliate'*; in questo senso per rendere la ricerca piú efficiente SATIVA utilizza varie euristiche come il ***restart***, la **1UIP** per il salto all'indietro, la **sussunzione per ottimizzazione** all'inizio, la **semplificazione** di clausole giá soddisfatte, l'euristica **VSIDS** per la decisione di variabili e la **luby activity** come scelta del numero massimo di conflitti accettabili prima di fare restart.
Il formato per specificare le clausole CNF é il formato DIMACS[1] che puó essere inoltrato al programma o via linea di comando o via file. Inoltre vi é una implementazione del problema **pigeonhole** che genera formule non soddisfacibili, il quale puó essere impiegato per verificare il funzionamento del programma se sprovvisti di formule da sottoporre.

SATIVA

SATIVA presenta 3 punti chiave: la propagazione, che in generale occupa l'80% della computazione; la procedura CDCL, che implementa il metodo principale; la procedura di risoluzione di una clausola di conflitto, che sfruttando la risoluzione in logica proposizionale trova la *prima clausola di asserzione*, fondamentale per il salto all'indietro (*backjump*).

[1] `http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`

Procedura CDCL

Algorithm 1: CDCL procedure

```
1  while true do
2      C_conflict ← propagate();
3      if C_conflict ≠ NULL then
4          if decisionLevel() = root_level then
5              return UNSAT;
6          [btLevel, C_A] ← analyze(C_conflict);
7          backtrack(btLevel);
8          record(C_A);
9      else
10         if decisionLevel()=root_level then
11             simplify();
12             decayActivity();
13         if areAllAssigned() then
14             return SAT;
15         else if #conflicts > max #conflicts
               then
16             backtrack(root_level);
17             lubyActivity();
18         else
19             assume(p_highest score);
```

I punti chiave in [**Algorithm 1**] sono la propagazione (riga 2) dove $C_{conflict}$ é la clausola di conflitto (se esiste) e C_A generata alla riga 6 che rappresenta la clausola di asserzione e definisce il prossimo '*percorso*' nello spazio di ricerca che il solutore sceglie. Questi due passi sono (computazionalmente) i piú pesanti.

Funzione di propagazione

Algorithm 2: Propagate solver function

```
1  while propQ ≠ ∅ do
2      p ← propQ.pop();
3      watchers ← watcher_of(p);
4      watcher_of(p).clear();
5      foreach x ∈ watchers do
6          if x.propagate(this, p) = false then
7              propQ.clear();
8              watcher_of(p) ← watchers_remaining;
9              return x;
10 return NULL;
```

La procedura di propagazione é quella che influenza di piú l'algoritmo poiché potenzialmente ogni cambiamento di un letterale puó andare ad influire su tutte le clausole; se a questo viene aggiunta la presenza di conflitti e quindi dover ripartire da un certo punto il numero di propagazioni cresce esponenzialmente. Di conseguenza avere una propagazione efficiente rende la ricerca della soluzione efficiente. É in questo contesto che l'utilizzo dei `watched literals`

é di aiuto, infatti utilizzare questi per decidere quali clausole devono essere controllate limita i controlli alle sole clausole interessate, evitando quindi di esplorare clausole non interessanti per l'assegnamento corrente. Prima di presentare l'algoritmo di propagazione delle clausole verrá presentata l'implementazione dei *watched literals* fondamentale per comprendere le operazioni effettuate:

Watched literals Per ogni clausola un vettore di letterali é allocato in memoria e sempre i primi 2 letterali sono i `watched literals`. Inoltre il solutore contiene un array dinamico in memoria di lunghezza doppia rispetto al numero di letterali con puntatore shiftato n volte dove n é proprio il numero di letterali:

$$[-n\ (-n+1)\ ...\ -1\ 0\ 1\ ...\ (n-1)\ n]$$

0 é la locazione puntata dal puntatore dell'array e non rappresenta nessuna variabile. Quindi per ogni variabile e la sua negata esiste una posizione indicizzabile nell'array sopra elencato. Questo array dinamico viene chiamato `watches`.

Ogni locazione in `watches` contiene un vettore di puntatori di clausole dove compare uno dei due *watched literals* in forma negata. In questo modo se si volesse accedere a tutte le clausole contenenti il *watched literal* l_1 basterebbe accedere alla locazione $-l_1$. Quindi basandosi sul fatto che una clausola deve essere controllata se e solo se uno dei suoi due *watched literals* é diventato falso, nella procedura di propagazione *watcher_of(p)* ritorna le clausole in cui appare $\neg p$ e quindi tutte le clausole dove p é diventato falso evitando di effettuare cicli su tutte le clausole. Una volta fatto questo il metodo delega alla clausola stessa la propagazione passando se stesso (cioé la classe solutore) come parametro e il letterale di propagazione considerato (che sará sicuramente uno dei due *watched literals*). La clausola effettua tutte le operazioni necessarie per mantenere i due *watched literals* a non falsi e ritorna *true* se ci é riuscita, false se invece no e quindi segnala la presenza di un conflitto. Successivamente verranno riaggiunte le clasuole non propagate ai loro *watchers*, ovvero alla locazione *p-esima* del vettore `watches` dove p é il letterale corrente di propagazione; ed infine verrá ritornata la clausola di conflitto se esiste, altrimenti tornerá *NULL*.
É interessante notare che in questo modo sia la ricerca di clausole di propagazione che l'operazione di backtrack sono instrinseche con la struttura *watches* e non richiedono nessuna operazione aggiuntiva di ricerca.

Algorithm 3: Propagate clause function *(solver, p)*

```
1  if deleted then
2      skip_and_destroy();
3  if literals[0] = ¬p then
4      swap(literals[0],literals[1]);
5  assert(literals[1] = ¬p);
6  if solver.value(literals[0]) = T then
7      solver.watchers[¬literals[1]]←this;
8      return true;
9  for i=2...literals_size do
10     if solver.value(literals[i]) ≠ F then
11         swap(literals[1], literals[i]);
12         solver.watchers[¬literals[1]]←this;
13         return true;
14 solver.watchers[p] ← this;
15 return solver.enqueue(this, p);
```

NB: In questo caso il metodo `solver.enqueue(·, ·)` controlla se l'assegnamento corrente é in conflitto con la clausola sotto analisi, se non lo é allora propaga il letterale che a quel punto del codice (riga 15) é di propagazione.

Generazione della clausola di asserzione

Il processo di generazione della clausola di asserzione segue tale regola:

$$\frac{C \cup \{p\}, D \cup \{\neg p\}}{C \cup D} \tag{2}$$

Questa regola viene applicata inizialmente tra la clausola di conflitto e l'ultima giustificazione dell'ultimo letterale sulla traccia che ha causato il conflitto; e ricorsivamente il risolvente viene utilizzato come nuovo membro della risoluzione. In questo modo prima o poi la risoluzione porta ad una clausola definita come *clausola di asserzione*:
Una clausola di asserzione é una clausola di conflitto tale che solo 1 dei suoi letterali é reso falso dal livello corrente.
Successivamente questá clausola di asserzione verrá imparata e *guiderá* la ricerca della soluzione.

Euristiche

VSIDS *Variable State Indipendent Sum* setta un contatore per ogni variabile, l'idea é di fornire un modo per scegliere quali sono le variabili che vengono decise. VSIDS é definita in questo modo:

- Ogni variabile ha un contatore, inizializzato a 0.
- Quando una clausola é aggiunta all'insieme di clausola (parte del problema originale o imparata),

i contatori associati ad ogni letterale di tale clausola vengono incrementati.

- La variabile con il contatore piú alto é scelta in un passo di decisione.
- Tutti i contatori sono divisi per una costante ciclicamente.

Da questa definizione classica in SATIVA é stato fatto un cambiamento: la divisione richiede molte operazioni che vanno a diminuire le prestazioni, quindi si é preferito utilizzare la sottrazione incrementando quindi le performance.

1UIP Un *Unit Implication Point* é un assegnamento di una variabile al livello corrente, dovuto ad una implicazione, che domina tutte le implicazioni che hanno portato ad un conflitto. Con 1UIP si intende il primo *Unit Implication Point* nella traccia di esecuzione. Il processo di risoluzione che genera la clausola di asserzione trova esattamente questo 1UIP. Questo schema permette di imparare clausole che guidano la ricerca della soluzione, infatti la clausola di asserzione del 1UIP é una clausola imparata.

Sussunzione per ottimizzazione La sussunzione é definita in questo modo:

$$\frac{S \cup \{C, D\}}{S \cup \{C\}} (C \in D) \tag{3}$$

Il significato risiede nel fatto che se $C \in D$ e se C é soddisfatta, allora anche D é soddisfatta in quanto esiste un letterale anche in D (poiché C é sottoinsieme di D) che é vero e soddisfa la clausola. Di conseguenza D é ridondante e posso eliminarlo. Questo ragionamento SATIVA lo fa una solva volta quando carica la formula del problema. In aggiunta controlla se una clausola é tautologa e se lo é la elimina.

Restart Il restart prevede di ripartire dal livello 0, mantenendo gli assegnamenti dovuti a propagazione al livello 0 e togliendo gli altri. Questa tecnica é utile quando l'algoritmo cerca una soluzione in uno spazio di ricerca in cui non vi é, e la procedura continua a generare conflitti. Ripartendo da 0, i conflitti imparati guidano l'algoritmo a non ricadere nella stessa situazione. SATIVA implementa le sequenza di luby definita come:

$$t_i = \begin{cases} 2^{k-1}, \text{ if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, \text{ if } 2^{k-1} \leq 2^k - 1 \end{cases} \tag{4}$$

Questa formula non fa altro che definire una sequenza del tipo $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ...\}$, e moltiplicando per un fattore di 100, si ottengono sequenze del tipo $\{100, 100, 200, ...\}$. Questi valori danno il massimo numero di conflitti accettabili rispetto al numero di restart fatti fin'ora. Questo schema in SATIVA non é molto ottimizzato, ovvero viene semplicemente applicata questa regola allo schema di restart; tuttavia un'ottimizzazione in questo senso é necessaria, si vedrá nei risultati come la semplice applicazione di questa regola non da performance buone rispetto a solutori come **Minisat** o **Glucose**.

Semplificazione La semplificazione é un processo che SATIVA esegue solo al livello 0, e prevede l'eliminazione di clausole soddisfatte a tale livello. Questa strategia é molto buona quando vengono imparate clausole unitarie in quanto taglia molto lo spazio di ricerca eliminando tutte le clausole che contengono il letterale propagato al livello 0.

Implementazione

Il progetto é stato scritto in C++ sfruttando una progettazione modulare con il paradigma a oggetti.

Strutture Dati

- **Clausole**: Le clausole sono salvate in un vettore di puntatori di clausole. Le motivazioni di tale scelta sono 2: in primo luogo viene utilizzata meno memoria in quanto gli oggetti clausola sono istanziati una volta sola; inoltre lavorare con i puntatori rende le clausole *globali*, e quindi qualsiasi cambiamento in una clausola é globale.
- **Letterali**: I letterali sono salvati in un vettore di letterali.
- **Watchers**: I watchers, come giá accennato nella descrizione della funzione di propagazione, é un array dinamico di vettori di puntatori di clausole.
- **Clausole imparate**: Le clausole imparate sono salvate in un vettore di puntatori di clausole diverso da quelle originali del problema, la motivazione risiede nel fatto che puó essere utile poter eliminare clausole imparate non utili.

Per maggiori dettagli riguardo l'implementazione consultare il codice o la documentazione generata con doxygen lanciando lo script `generate_docs.sh`: `bash generate_docs.sh` .

Compilazione ed esecuzione

Dipendenze

Per la compilazione del codice é sufficiente avere un compilatore per c++11 (ubuntu ne é provvisto di de-

fault) e il cmake installato (lo si puó installare con il comando `sudo apt-get install cmake`.

- Come tool di sviluppo é stato utilizzato il cmake[2]. La versione minima testata é la `2.8.12`; ubuntu 14.04 dispone della versione `2.8.12` e ubuntu 16.04 la versione `3.5.1`.
- Per la generazione della prova dell'insoddisfacbilitá si é utilizzato il tool `dot` che é in grado di generare un'immagine contenente l'albero dei passi di risoluzione. Se sprovvisti basta lanciare il comando: `sudo apt-get install graphviz`.
- I file di input seguono il formato dimacs e ne vengono fornite delle istanze di esempio in `inputs/benchmark`, tuttavia é possibile scaricarle e darle in input al programma.

Compilazione

Per comoditá é presente uno script bash che compila il codice secondo le configurazioni consigliate eseguibile con il comando `bash compile.sh`; tuttavia é possibile personalizzare le impostazioni. Ecco un esempio di come fare:

- `mkdir buid && cd build`, genera la cartella contenente i file binari e i file necessari per la compilazione.
- cmake .. , crea i file necessari per il `Makefile`.
- `make inputs`, crea un link simbolico per la cartella `inputs` presente nella cartella superiore (facoltativo).
- make, compila il codice generando il file binario `sativa`.

In questa configurazione non viene generata la prova se il problema risulta UNSAT, inoltre é impostata la modalitá `VERBOSE`, dove viene mostrato il progresso della computazione o della lettura e altri passi; la modalitá `VERBOSE` impatta leggermente sulle prestazioni, quindi si puó disabilitare tale modalitá settandone il parametro: anziché eseguire `cmake ..` si deve eseguire `cmake .. -DVERBOSE=0`.
Se invece si vuole generare la prova si deve digitare `cmake .. -DP=1`.
In alternativa si puó utilizzaare lo script `compile.sh` per generare un binario in modalitá `VERBOSE` o lanciare `compile.sh proof` per generare il binario con la generazione della prova. Per ogni dubbio si puó consultare il `README` contente i dettagli di compilazione ed esecuzione.

[2] `https://cmake.org/`

Esecuzione

Per lanciare il programma il comando é il seguente:
`./sativa [-f,--file/-p,--pigeonhole] [f/n]`
dove con il parametro -f o –file si deve sottoporre un file in formato DIMACS, con -p o –pigeonhole si deve sottoporre un numero che identifica il numero che specifica la dimensione del problema **pigeonhole**. In ogni caso lanciando il programma con `-h` o `--help` ne viene indicato l'utilizzo.
Nel caso in cui venga generata la prova viene creato un file in `graphics` compatibile con il programma dot. Per semplificare la generazione dell'immagine Si puó utilizzare il makefile, lanciando il comando `make graphviz` viene generato il file `proof.png` all'interno di `graphics`. Tale file si puó aprire con
`eog graphics/proof.png`
o con qualsiasi altro visualizzatore di immagini e rappresenta l'albero di risoluzione.
Ecco un esempio di prova di insoddisfacibilitá generata con il tool appena presentato con l'insieme di clausole definito come $S = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$:

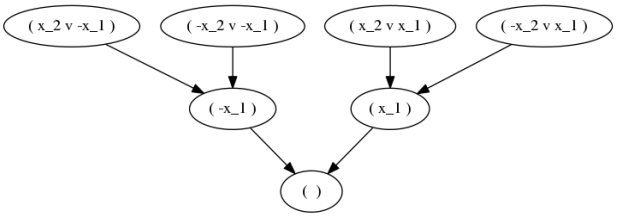

Figure 1: Prova dell'insoddisfacibilitá di S

Conclusioni

Per testare la validitá del programma sono state prese 1000 formule soddisfacibili e 1000 formula insoddisfacibili di dimensione ridotta e si é verificato che la risposta del programma fosse corretta in tutte le 2000 formule. Una volta accertato questo fatto si é passati all'analisi delle performance e delle euristiche.
La macchina utilizzata per fare tutti i test ha le seguenti caratteristiche:

- Sistema operativo: Ubuntu 14.04 LTS
- Processore: Intel Core i5-3470 CPU @ 3.20 GHz x 4

Pigeonhole

Dati n buchi ed $n+1$ oggetti, ponendo il vincolo che ogni buco possiede al piú un oggetto e chiedendo se tutti gli oggetti hanno un buco il problema risulta insoddisfacibile. Nella figura [2] viene evidenziato come l'euristica **restart con luby sequence** non

nome	literals	clauses	CPU TIME Sativa	Glucose 3.0	Minisat 2.0	learnts	decision	propagation	deleted	mem used
aes_32_1_keyfind_1.cnf	300	1016	0,10	0,03	0,02	2105	2506	148412	212	13
aim-100-1_6-no-1.cnf	100	160	0,00	0,00	0,00	21	164	399	12	12
bf0432-007.cnf	1040	3668	0,02	0,01	0,01	271	995	42059	2505	13
C168_FW_UT_518.cnf	1909	7511	0,00	0,01	0,01	1	383	2139	5256	13
CBS_k3_n100_m418_b90_185.cnf	100	418	0,03	0,00	0,00	1204	1484	40531	60	12
uum8.smt2-stp212.cnf	1006	3359	61,67	12,92	0,00	84241	95974	44657172	1091	121
velev-sss-1.0-cl.cnf	1453	12531	11,61	0,27	10,80	70379	168224	10631149	5730	53
vars-250-10.cnf	250	1063	0,24	0,09	0,76	5617	8300	325946	0	13
vars-250-2.cnf	250	1063	914,04	13,27	2,80	594392	713437	33820799	217	150
vars-250-3.cnf	250	1063	663,14	6,31	0,52	541632	670327	30775399	50	139
vars-250-4.cnf	250	1063	14,27	5,78	1,61	77406	107706	4558608	0	30
pigeonhole 8	72	297	0,24	1,16	3,33	1922	1952	48386	87	13
pigeonhole 9	90	415	0,71	18,66	23,56	4438	4514	123514	143	14
pigeonhole 10	110	561	2,63	165,67	777,45	10219	10333	319226	155	16
pigeonhole 11	132	738	10,4508	1339	timeout	23255	23435	770255	214	20
pigeonhole 12	156	949	51,9856	timeout	timeout	52541	52851	1848427	238	37
pigeonhole 13	182	1197	275,877	timeout	timeout	131871	132780	4705515	338	75

Table 1: Tabella dei risultati con un timeout di 1000 secondi

dia i migliori risultati per tale problema, mentre sia piú conveniente non utilizzare i restart ma lasciare che SATIVA si addentri il piú possibile nello spazio di ricerca. La **restart luby sequence** é una tecnica molto usata da solutori come **minisat2.0** o **glucose3.0**, inoltre la tecnica del restart é piú ottimizzata in questi solutori; tuttavia, come si é giá visto, per il problema pigeonhole il restart non é la soluzione migliore; infatti nella figura [3] si sottolinea maggiormente come il restart impatti negativamente sul problema.

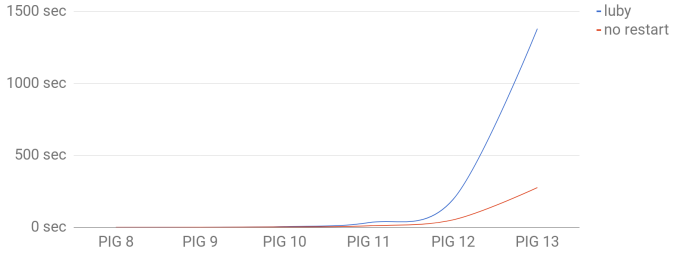

Figure 2: Pigeonhole luby sequence and no restart

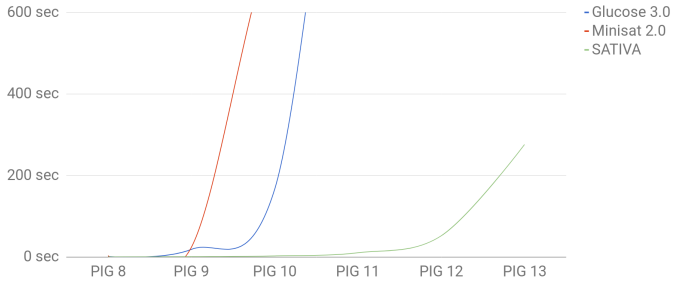

Figure 3: Pigeonhole different sat solvers, timout 600 sec

Dalla tabella [1] si nota chiaramente che quanto detto fin'ora per i pigeonhole non vale. Se testiamo SATIVA con altri solutori sat rispetto ad altri problemi notiamo una differenza notevole: l'ottimizzazione riguardo i restart presumibilmente impatta molto sulle qualitá dell'algoritmo, tanto che SATIVA che non implementa un'ottimizzazione nei restart raffinata quanto **glucose 3.0** oppure **minisat 2.0** computa addirittura 100 volte peggio.

Benchmarks

Sono stati presi d'esempio diverse tipologie di problemi: **C168_FW_UT_518** descrive un problema di validazione di prodotti del settore automobilistico; **aim_*** sono problemi costruiti artificialmente[3]; **velev-sss-1.0-cl.cnf** é un problema di verifica formale di un microprocessore[4];**ex*** sono esempi visti a lezione; **zebra*** é un famoso problema di *logic puzzle*[5]; **aes_*** sono problemi di crittografia trovati online; tutti gli altri problemi sono problemi trovati nelle varie competizioni scaricabili da qui `http://www.satcompetition.org/`

Considerazioni

SATIVA implementa la procedura CDCL, si avvale dei watched literals e utilizza diverse euristiche quali VSIDS, tecnica del restart, 1UIP e la sussunzione per evitare una ricerca esaustiva. Una progettazione di questo tipo permette di realizzare un solutore in grado di competere con i piú avanzati solutori disponibili online, come sottolineato dai risultati; tuttavia ottenere tempistiche molto buone é un compito assai arduo che da una parte dipende dal problema da cui deriva la formula, come si é visto analizzando il problema pigeonhole; e dall'altra da quali euristiche si utilizza per la ricerca. Qualsiasi operazioni potenzialmente va a peggiorare/migliorare le prestazioni, in quanto una semplice divisione al posto di una sottrazione puó rendere fino a 100 volte peggiore la computazione. Quindi l'attenzione nei dettagli nel progettare un solutore sat é fondamentale, d'altronde si risolve problemi NP-completi.

[3] `https://people.sc.fsu.edu/~jburkardt/data/cnf`
[4] `http://www.miroslav-velev.com/sat_benchmarks.html`
[5] `https://en.wikipedia.org/wiki/Logic_puzzle`

Bibliografia

[1] Niklas Eén, Niklas Sörenson, Chalmers University of Technology, Sweden, An extensible SAT-solver[extended version 1.2] http://minisat.se/downloads/MiniSat.pdf

[2] Niklas Eén, Niklas Sörenson, Chalmers University of Technology, Sweden, Effective Preprocessing in SAT through Variable and Clause Elimination http://minisat.se/downloads/SatELite.pdf

[3] João P. Marques Silva, Ines Lynce, Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marjin Heule, Hans Van Maaren and To by Walsh (Eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, Vol. 185, pages 131 153, IOS Press, 2009. http://www3.cs.stonybrook.edu/~cram/cse505/Fall16/Resources/cdcl.pdf

[4] Inês Lynce, João Marques-Silva, Efficient data structures for backtrack search SAT solvers, https://link.springer.com/article/10.1007/s10472-005-0425-5

[5] Lintao Zhang, Sharad Malik. The quest for efficient Boolean satisfiability solvers. In Proceedings of the 18th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 2392, 295313, Springer, July 2002 .