Modulo2--Unit Propagation

N. Galesi—Dipartimento di Informatica

1. Preliminari

Per avere un'idea più precisa di cosa succede nel solver, possiamo pensare alla ricerca di un assegnamento che soddisfa la formula come la ricerca in un albero di assegnamenti. Ogni assegnamento corrisponde ad un ramo nell'albero che il solver lentamente esplora. Noi siamo alla ricerca di un ramo che soddisfi la formula. L'esplorazione di alcuni rami però potrebbe portare a falsificare la formula pur non essendo escluso che questa sia soddisfacibile. A quel punto dobbiamo tornare indietro nell'albero e tentare un altro ramo (bakctracking). L'implementazione del solver segue l'idea precedente. In questo modulo ci occuperemo di progettare il meccanismo di esplorazione dell'albero degli assegnamenti. Nel modulo 3 ci occuperemo di "tornare indietro" e recuperare l'informazione eventualmente persa.

Come visto nella descrizione generale del progetto, una fase fondamentale del sat-solver è quella della propagazione di informazione unitaria. Supponiamo di essere alla ricerca di un assegnamento che soddisfi la formula $(x_7 \vee \neg x_{203})\ (x_2 \vee \neg x_5)\ (\neg x_7)$. Sicuramente, se esistono, tali assegnamenti avranno la caratteristica di avere $x_7 = 0$ (falso), perché la clausola $(\neg x_7)$ deve comunque essere soddisfatta. Quindi non è necessario cercare un assegnamento che soddisfi $(x_7 \vee \neg x_{203})\ (x_2 \vee \neg x_5)\ (\neg x_7)$. Possiamo limitarci a settare $x_7 = 0$, mantenere questa informazione, semplificare consistentemente tutta la formula, creare una nuova formula (nell'esempio, $(\neg x_{203}),(x_2 \vee \neg x_5)$), e cercare un assegnamento che soddisfi questa nuova clausola. Se possibile dovremo iterare questo procedimento. Se si osserva attentamente il nostro esempio, si vedrà che si è creata un'altra clausola unitaria che può quindi essere nuovamente settata su tutta la formula. Questo processo di settare clausole unitarie, semplificare la formula e eventualmente iterare il processo, prende il nome di **Propagazione Unitaria** (unit propagation -- UP). Si noti che il processo avrà termine nei seguenti casi: (1) tutte le variabili sono settate, la formula allora è soddisfacibile ed abbiamo trovato un assegnamento che la soddisfa; (2) la propagazione falsifica una clausola: in questo caso dovremo tornare indietro nella storia della costruzione della nostra propagazione per capire dove è nato il problema ed evitarlo (questo sarà fatto nel Modulo 3); (3) non esistono più clausole unitarie: a questo punto sceglieremo un nuovo letterale nella formula ed un valore a cui settarlo (sulla base di euristiche che svilupperemo nel Modulo 3), semplifichiamo la formula e torniamo a riapplicare la propagazione li dove è possibile.

Coda di clausole unitarie

Nel corso di una semplificazione della formula è possibile produrre piu di una clauola unitaria. In principio tutti i letterali cosi incontrati dovranno essere propagati (a meno che non si termini pirma nel caso (2) della UP). La gestione dei letterali "in attesa" di essere propagati verrà eseguita attraverso uno coda.

Watcher List

Un modo per eseguire efficientemente la UP è quello di salvare per ogni letterale una lista di clausole dove questo letterale occorre. Tali letterali sono detti Watched e le liste a loro collegate sono dette watcher lists. Per evitare eccessiva ridondanza, per ogni clausola C si scelgono solo i primi due letterali, diciamo p e q, e la clausola C viene salvata nella watcher list solo di ¬p e ¬q. Nel momento in cui uno di questi due letterali dovesse essere eliminato (perché settato), si procede a trovare un altro letterale nella clausola C ed aggiornare la watched list di tale letterale.

Livelli di Decisione.
Quando ad una variabile viene dato un valore nel corso della UP, sarà importante conoscere il ***livello*** a cui tale decisione è stata presa. Se immaginiamo che un nodo in un ramo nelll'albero degli assegamenti si forma ogni qualvolta dobbiamo scegliere una variabile e darne un valore (caso (3) nella terminazione dell'UP), allora il livello dei decisione di una variabile nella UP corrisponde alla profondità nell'albero a cui è stata decisa l'ultima variabile che abbiamo scelto. Nel Solver per una gestione dei conflitti (clausole falsificate, Modulo 3) sarà importante mantenere traccia dei livelli di decisione delle varaibili.

2. Obiettivo
L'obiettivo del secondo modulo è quello di progettare completamente la fase di propagazione unitaria nel sat solver, immergerla in un contesto pronto per lo sviluppo completo del solver (cioè anche per il modulo 3), verificare la sua funzionalità.

3. La classe Solver
Come primo passo verso la costruzione del Modulo2, descriveremo le informazione da aggiungere nella struttura `solver`. In seguito procederemo alla modifica di due metodi `solve_search` e `solver_solve` e alla creazione di tre nuovi metodi: `solver_propagate`, `solver_enqueue` e `solver_assume`.

Struct solver
Tale struttura è la parte centrale del nostro progetto ed è quella che contiene tutte le informazioni necessarie alla ricerca dell'assegnamento:

I vari campi sono cosi suddivisi (non tutti i campi qui descritti verranno usati nel Modulo 2.):

Informazioni sulla Formula:
`int nof_var,` contiene il numero di variabili nella formula

`int nof_clauses;` contiene il numero di clausole nella formula

`vec_p clauses;` contiene le clausole della formula

`vec_p learnts;` contiene le clausole apprese durante il processo di ricerca.

Informazione per la Propagazione

`vec_p  *watches;` per ogni letterale contiene la liste della clausole watched. Si tratta di un array di dimensione 2*nof_var, con la convenzione che le info relative alla variabili xi vengono salvate in posizione 2*i se la variabile è positiva e 2*i+1 se la variabile è negativa

`vec_p  *undos;` per ogni letterale la lista delle clausole da recuperare durante il backtracking (Modulo3)

`vec_i Pcoda;` coda di propagazione dei letterali unitari.

Informazione per gli assegnamenti

`lbool *assigns;` assegnamento corrente. Si tratta di un vettore di dimensione pari al numero di variabili che memorizza un lbool (TRUE, FALSE UNDEF) a seconda che la corrispondente variabile abbia valore vero, falso o non ancora assegnato.

`lit *trail;` lista di assegnamenti in ordine cronologico. Trattato come uno stack.

`vec_i trail_lim;` Indici delle posizioni di separazione tra differenti livelli di decisioni in trail. La dimensione ci dice l'ultimo livello visto. È un vettore di dimensione pari al numero di variabili e va trattato come uno stack.

`vec_i model;` Assegnamento finale

`clause **reason;` per ogni variabile assegnata, un puntatore alla clausola che ha implicato il suo assegnamento.

`vec_i level;` per ogni variabile, il livello di decisione a cui è stata assegnata

`int root_level;` livello di decisione iniziale;

`int seed;` indice di variabile generato casualmente (serve solo per testare il Modulo2);

► Completare la struttura solver e la sua inizializzazione

Metodo `solver_solve`

```
bool solver_solve(solver *)
```

Il metodo `solver_solve` può considerasi come il main della classe solver. Riceve in input e modifica in output il solver ricevuto dal main. Restituisce in output un booleano che dovrà essere settato a TRUE se la formula è soddisfacibile e FALSE se la formula è insoddisfaciblile.

Il suo compito è quello di eliminare informazione incrementale dalle clausole iniziali (non ve ne saranno nel nostro caso), chiamare il metodo `solve_search` mentre questo restituisce il valore indefinito (lbool UNDEF). `solver_search` va chiamato con due paramtetri double: `nof_conflitti` (numero di conflitti) e `nof_learnts` (numero di clausole apprese) inizialmente settati rispettivamente a 100 e al numero totali di clausole diviso 3. Ad ogni iterazione in cui `solve_search` restituisce UNDEF tali parametri vanno moltiplicati rispettivamente per 1.1 e dell 1.5 dei valori precedenti.

Non trattando ancora il, backtracking e il recupero delle informazioni, nel modulo 2 il valore indefinito UNDEF non sara mai restituito dal metodo `solver_search`. Per testare il risultato del modulo 2 creeremo una funzione `test_modulo2` con le seguenti caratteristiche: se solver_search restituisce TRUE allora la formula è soddisfacile e possiamo stampare l'assegnamento (che è contenuto nel campo `model` del solver). Se `solver_search` restituisce FALSE, allora stampa l'assegnamento parziale (contenuto nel campo `assign` del `solver`), la clausola di conflitto (contenuta nel campo `learnts` dl solver) e tutte le watcher-list per ogni letterale.

- **Definire il metodo `solver_solve`**
- **Definire il metodo `test_modulo2`**

Metodo `solver_search`

```
lbool solver_search(solver *,  double, double)
```

Il metodo solver_search è il metodo principale che si occupa della ricerca dell'assegnamento. In input riceve e modifica in output il solver, insime a due parametri di tipo double (numero conflitti e numero clausole apprese). In output restuisce un booleano liftato (lbool) come segue: TRUE, se ha trovato un asseganemnto che soddisfa la formula, FALSE quando il livello di decisione è quello della radice, UNDEF quando il numero di conflitti supera il limite sul numero di conflitti che ha ricevuto in input.

Questo metodo sarà completo solo nel Modulo 3. Nel Modulo 2 deve limitarsi ad eseguire quanto segue: chiamare il metodo `solver_propagate` con gli opportuni parametri, analizzare il suo output che sarà un puntatore ad clausola oppure NULL. Una clausola falsificata corrisponde ad un conflitto trovato, ovvero alla situazione in cui un assegnamento che è arrivato a falsificare una clausola. In tal caso (nel Modulo 2) tale clausola deve essere salvata nelle clausole apprese del solver (campo `learnts`), deve essere incrementato appropriatamente il numero di conflitti e il numero di clausole apprese. Per il momento (Modulo 2), la chiamata al metodo `solver_propagate` deve continuare fin quando o non si assegnano tutte le variabili (restituisce TRUE) oppure si incontra **il primo conflitto** (restituisce FALSE). Nel caso in cui non ci sia conflitto e non si siano assegnate tutte le variabili, si deve creare una funzione `solver_select` che seleziona aleatoriamente una nuova variabile ed un suo valore e lo inserisce nella coda dei letterali da propagare attraverso il metodo `solver_assume`.

► **Definire il metodo `solver_search`**
► **Definire il metodo `solver_select`**

Metodo `solver_propagate`

```
clause * solver_propagate(solver *)
```

Il metodo `solver_propagate` riceve in input e modifica in output il `solver`. In output produce un puntatore ad una clausola.
Si occupa di prelevare il prossimo letterale dalla coda delle clausole unitarie e propagarlo nella formula prelevando la watcher list (WL) del letterale p (è bene che La WL di p venga copiata in una nuova WL temporanea, perche la WL orginale di p deve essere ricostruita in questa routine).

Per ogni clausola nella WL del letterale p, si chiama il metodo `clause_propagate` per propagare correttamente p nella clausola e verificare se si crea un conflitto ed eventualmente aggiornare le informazioni della clausola (si veda il metodo `clause_propagate`). Se la clausola crea un conflitto, allora (un puntatore ad essa) viene restituita in output e viene aggiornata la WL di p eliminando quella clausola e trattenendo le rimanenti (ovvero copiandole dalla lista WL temporanea). In questo caso la coda viene cancellata.

► **Definire il nuovo metodo `solver_propagate`**

Metodo `solver_enqueue`

```
bool solver_enqueue(solver *, lit, clause *)
```

È il metodo che si occupa di memorizzare tutte le informazioni che seguono all'assegnamento di una variabile. Riceve in input il solver, un letterale (variabile +valore) ed un puntatore ad una clausola. Restituisce in output un booleano: TRUE se l'inserimento elle informazioni va a buon fine o se essite gai un assegnamento per quella variabile allo stesso valore di verità. FALSO se il letterale è correntemente gia assegnato ad un valore differente (si veda il campo assign del solver). L'inserimento dell'informazione nella struct `solver` è la seguente:

- aggiornare il vettore `assigns` usando p;
- aggiornare il vettore `level`, assegnando la dimensione corrente di `trail_lim` (implementare una funzione `dlevel` che restituisce tale valore e puo essere chiamata da altre parti).
- pushare in `trail` l'indice della variabile assegnata (si veda la sezione Code e Stack)
- salvare in `reason`, nella posizione pari all'indice della variabile la il puntatore alla clausola ricevuto in input.
- inserire l'indice del letterale nella coda `Pcoda` (si veda Classe Code e Stack) dei letterali da propagare per l'inserimento

► Definire il metodo solver_enqueue

Metodo `solver_assume`

```
void solver_assume (solver *, lit)
```

Riceve in input il solver e un letterale (pensato come variabile + valore) e restituisce un solver aggiornato con le informazioni relative all'inserimento del letterale.

Il suo compito è quindi solo quello di chiamare `solver_enqueue`. Si noti però che poiché `solver_assume`, è chiamato da `solver_search` solo nel momento in cui si seleziona una nuova variabile (e non durante la propagazione) in questo caso dobbiamo aggiornare anche le informazioni negli stack `trail`. In particolare nello stack `trail_lim` dobbiamo salvare la dimensione attuale di `trail`. (si usi la classe code e stack)

► Definire il nuovo metodo `solver_assume`

4. Classe Clausola

Nella classe clauola in questo modulo ci occuperemo di gestire correttamente il metodo dei watched literals (si vedano i Preliminari per una speigazione). Il lavro da fare in questo modulo si limita a quanto segue. Dovremo pirma di tutto modificare il metodo new_clause, iniziato nel Modulo 1. In particolare per questo modulo dovremo prevedere una prima fase di semplificazione delle clausole lette dal file e poi dovremo aggiungere ogni clauosola nella corrett WL. Il metodo non sarà concora completo e verrà ancora modificato nel Modulo 3 quando gestiremo completamente anche le clauole apprese.

Un altro meotdo da definire nel Modulo 2 è il metodo `clause_propagate`. Tale metodo (uno dei piu importanti anche se non centrale) si occupa propgare correttamente nela formula (cioe nelle WL) l'informazione che proviene dall'aver dato un nuovo valore ad una variabile.

Metodo `new_clause`

Per prima cosa per il metodo vanno aggiunti nuovi parametri e tolti alcuni che erano ridonadanti nel Modulo1 (l'informazione size si può recuperare da v->size)

```
bool  new_clause(solver *, bool, float, vec_i, clause *)
```

I parametri sono gli stessi del Modulo1 eccetto per il solver che questa volta viene passato e modificato. In output restituisce un booleano che sarà FALSO solo se la clausola da aggiungere ha dimensione 0.

Prima di salvare in `clause`, `new_clause` deve "semplificare" la clausola: ovvero eliminare letterali ripetuti, restituire TRUE se la clauola contiene un letterale e la sua negazione. Semplificata la clausola, ne va controllata la dimensione. Se la clausola contiene un solo letterale, tale letterale va allora già posto nella coda dei letterali da

propagare (metodo `solver_enqueue`). Se la clauola non è unitaria, allora la clausola va aggiunta nella WL solo della negazione dei suoi due primi letterali (anche le WL possono essere tratttate come stack, si veda la sezione Classe Code e Stack). In questo modulo non tratteremo ancora il caso di clausole apprese e per tanto il parametro `learnt` si suppone sempre FALSO.

► Definire il nuovo metodo `new_clause`

Il metodo `clause_propagate`

```
bool clause_propagate(solver *, lit, clause *)
```

Il metodo clause_propagate riceve il solver, una clasuola ed un letterale e restituisce un boleano. L'ouptut sarà TRUE se la clausola è già soddisfatta oppure se tutto è andato a buon fine.

Il metodo per prima cosa sposta se necessario il letterale in input nella seconda posizione della clausola. Quindi usando il campo `assigns` del solver, verifica che il letterale non abbia valore TRUE. In questo caso reintroduce la clausola nella WL e restituisce TRUE.

Se il letterale non è soddifatto, cerca il primo letterale (escluso il primo) nella clausola che non è stato assegnato (UNDEF) o che ha valore TRUE e lo intercambia con il letterale in input (che si trova in prima posizione) e salva la clauola nella WL di questo nuovo letterale.

Infine se assegendo il letterale la clausola diventa unitaria, la clauola ve reiniserita nella WL del lettereale, il letterale risultante va salvato nella coda delle clausole unitarie e la clausola deve essere inserita tra le sue ragioni (si veda campo `reasons` del solver e metodo `solver_enqueue`)

► Definire il nuovo metodo `clause_propagate`

5. Classe Code e Stack

Si deve progettare completamente una nuova classe CodeStack (per il momento può essere fatto nella classe `vettori`) che si occupa di trattare (1) la coda dei letterali da propagare (salvata nel campo `Pcoda` del `solver`). Si devono implementare le seguenti funzioni:

- `insert`: che inserisce un nuovo elemento nella coda
- `dequeue`: che preleva un elemento dalla coda
- `clear`: che cancella il contenuto della coda
- `size`: che restituisce la dimensione della coda

(2) uno stack che consente di trattare i vettorI `trail e trail_lim`. Le operazioni da implementare sono le solite:

- `push`
- `pop`

- `size`
- `top`
- `clear`

► Progettare la classe Code e Stack

6. Input, Output e Testing del Modulo2.

Output

L'output del programma è fondamentalmente gestito dal metodo `test_modulo2`. A scopo chiarificatore elencheremo qui cosa deve produrre in output e come avverrà il suo test. La funzione `test_modulo2` si attiva nel momento che `solver_search` restituisce TRUE o FALSE (si ricordare che UNDEF non verra mai restituito nel Modulo2). Se TRUE allora l'assegnamento soddisfa la formula e viene copiato nel campo `model` del `solver`. Va quindi stampato in output. Se `solver_search` restituisce FALSE allora vuol dire che la ricerca si è fermata al primo conflitto incontrato. `test_modulo2` deve restituire una fotografia dello stato del solver in questo momento, ovvero:

- clausola di conflitto (che in questo caso sarà solo la clausola iniziale falsificata dall'assegnamento),
- assegnamento che conduce al conflitto,
- contenuto delle WL di tutti i letterali.

Input del programma

Il modulo è una buona occasione per fare in modo che il programma riceva i suoi input dalla linea di comando (parametri del main). Per il modulo2, a parte l'ovvio nome del file, deve essere fornito un intero che dovrà essere salvato nel campo `seed` del solver e deve essere usato per generare aleatoriamente un indice di variabile nel metodo `solver_select`.

Test del programma

Il programma deve essere testato, rieseguendo il main diverse volte dando un seed differente ogni volta, in modo da assicurarsi una diversa selezione di variabili. Sarà importante testare il programma su esempi di CNF soddisfacibili e di piccole dimensione che daranno un'idea più precisa del funzionamento del Modulo 2.