Implementazione della procedura DPLL attraverso il linguaggio di programmazione JAVA

Alberto Righetti

INTRODUZIONE

La procedura DPLL è considerata il metodo computazionalmente più efficiente per valutare se una formula proposizionale S sia soddisfacibile o meno; l'istanza di ingresso della procedura è rappresentata da un insieme di clausole ground (ovvero un insieme dove non appaiono variabili ma solo letterali) in forma normale congiunta (CNF), mentre l'output consiste nel decidere se l'insieme sia soddisfacibile o meno, nel caso in cui lo sia l'algoritmo restituisce un assegnamento che soddisfa l'insieme di clausole.
Quindi, a partire da una formula S in CNF si cerca di costruire un assegnamento che verifichi la formula, tale assegnamento viene generato usando un meccanismo di backtracking. La procedura può essere vista come un albero di possibili assegnamenti, nel quale ciascun nodo rappresenta un insieme di clausole e per ogni nodo ad un letterale viene assegnato un valore di verità; tale valore viene poi propagato per ridurre il numero di assegnamenti futuri. Un ramo dell'albero non viene più espanso se l'insieme di clausole è vuoto (S=$\{\}$ perciò S è soddisfacibile) oppure se è stata generata la clausola vuota $\square$ (in questo caso S è non soddisfacibile). Gli assegnamenti vengono effettuati applicando alcune regole che preservano la soddisfacibilità; vediamole in dettaglio:

- **Tautology Elimination:** dall'insieme di clausole vengono eliminate le istanze ground che sono tautologie

$$S = \{(\neg P \vee Q \vee P \vee \neg R) \wedge Q \wedge R\}$$

$$S' = \{Q \wedge R\}$$

- **One-Literal:** se esiste una clausola unitaria $L \in S$, otteniamo S' da S eliminando quelle clausole ground in S che contengono L. Se S=$\{\}$ allora S è soddisfacibile altrimenti otteniamo un insieme S'' da S' eliminando $\neg L$ da tutte le clausole; S è non soddisfacibile se e solo se S'' è non soddisfacibile. Quando applichiamo questa regola fissiamo L=$\top$ nell'assegnamento parziale.

$$S = \{P \vee Q \vee \neg R, P \vee \neg Q, \neg P, R, U\}$$

  Applichiamo One-Literal su $L = \neg P$ e otteniamo

$$S' = \{Q \vee \neg R, \neg Q, R, U\}$$

- **Pure Literal:** $L \in S$ è un letterale puro se $\neg L \notin S$, se esiste tale letterale in S allora otteniamo S' rimuovendo tutte le clausole dove appare L (quando

questa regola viene applicata, fissiamo $L{=}\top$ nell'assegnamento parziale)

$$S = \{P \vee Q, P \vee \neg Q, R \vee Q, R \vee \neg Q\}$$

applichiamo Pure-Literal su $L = P$ e otteniamo

$$S' = \{R \vee Q, R \vee \neg Q\}$$

- **Splitting Rule:** se possiamo scrivere S nella seguente forma

  $$S = (C_1 \vee L) \wedge \cdots \wedge (C_m \vee L) \wedge (D_1 \vee \neg L) \wedge \cdots \wedge (D_m \vee \neg L) \wedge S_r$$

  dove C_i e D_i sono clausole in cui L e $\neg L$ non appaiono, e dove S_r è un insieme di clausole dove L e $\neg L$ non appaiono. Allora possiamo ottenere due insiemi $S'{=}C_1 \wedge \cdots C_m \wedge S_r$ e $S''{=}D_1 \wedge \cdots D_m \wedge S_r$. L'insieme S è non soddisfacibile se e solo se S' e S'' sono non soddisfacibili. Quando questa regola viene applicata, viene eseguito uno split dell'albero degli assegnamenti e fissiamo $L{=}\top$ per il ramo di S'' e $L{=}\bot$ per il ramo di S'.

  $$S = \{P \vee \neg Q \vee R, \neg P \vee Q, Q \vee \neg R, \neg Q \vee \neg R\}$$

  Applichiamo lo split su P e generiamo due rami:

  $$S' = \{\neg Q \vee R, Q \vee \neg R, \neg Q \vee \neg R\} per P = \bot$$

  $$S'' = \{Q, Q \vee \neg R, \neg Q \vee \neg R\} per P = \top$$

Il linguaggio di programmazione scelto per l'implementazione del progetto è JAVA, versione 1.6.0.

SCELTE IMPLEMENTATIVE

La classe che implementa la procedura DPLL inizialmente deve leggere l'insieme di clausole da un file costruito seguendo un formato standard così definito: il file inizia con un'intestazione rappresentata dalla seguente linea

p cnf num_lett num_cla

dove num_lett indica il numero di lettere proposizionali che occorrono nella formula e num_cla indica il numero delle clausole presenti nella formula; ogni linea che inizia con la lettera c viene considerata un commento.
Gli atomi sono espressi con numeri da 1 a num_lett; la negazione di un atomo è rappresentata da un numero negativo. Una clausola è invece rappresentata da una riga di letterali separati tra loro da uno spazio e terminante con uno 0. Per esempio la formula $C = \{p1 \ ? \ p2, p1 \ ? \ \neg p2, \neg p1 \ ? \ p2, \neg p1 \ ? \ \neg p2\}$, sara' descritta dal seguente file:

```
c *** insoddisfacibile ***
p cnf 2 4
 1 2 0
 1 -2 0
 -1 2 0
 -1 -2 0
```

Inizialmente, quindi, viene aperto uno stream in lettura di un file avente la forma appena menzionata, ogni riga di tale stream viene convertita in stringa e vengono eseguite due operazioni:

- attraverso il metodo num_lit() viene fatto uno scorrimento delle stringhe corrispondenti a ciascuna riga fino a che il primo carattere letto di una riga corrisponda al carattere "p", in questo caso vengono ricavati tre interi corrispondenti al numero totale di atomi presenti nella formula, al numero di lettere proposizionali che occorrono e al numero di clausole presenti;

- attraverso il metodo create_clause() viene fatto uno scorrimento dello stream fino a che il primo carattere letto di una riga sia diverso dai caratteri "c" e "p", ciò significa che la sequenza di valori numerici corrispondenti ad atomi della formula ha inizio e quindi ogni intero incontrato viene inserito in un array.

L'operazione successiva consiste nel creare, a partire dall'array statico appena definito, un array di liste sul quale poi verranno eseguite le quattro regole viste in precedenza; il costrutto utilizzato è ArrayList() contenuto nel pacchetto java.util. L'idea, quindi, è quella di avere un array, avente dimensione statica pari al numero di clausole, di liste, le quali possono essere modificate in modo dinamico. L'informazione relativa all'insieme di clausole iniziali comunque non viene persa, visto che rimane memorizzata nell'array statico definito dal metodo create_clause() e che non viene mai modificato durante l'esecuzione.

Poi viene invocato il metodo taut_check() che si occupa di verificare se, nell'insieme di clausole, sono presenti delle tautologie: in sostanza, per ciascuna clausola attraverso due cicli **for** viene fatto un matching incrociato tra ogni letterale e i successivi e viene controllato se, per ciascun letterale ovvero per ciascun numero, sia presente il suo corrispondente negato; in caso affermativo, attraverso il metodo clear() fornito dalla classe ArrayList() viene eliminata la clausola, ovvero la lista, che contiene la tautologia. Il passo seguente è quello di identificare se esiste un letterale unitario: il metodo one_literal() va a scorrere l'array di liste, quando ne trova una avente dimensione pari a due viene invocato, come in precedenza per il caso della tautologia, il metodo clear() che va a rimuovere la lista; se invece non ne trova, restituisce 0.
Qualora un letterale unitario venisse trovato, ciò significa che nell'insieme di clausole è presente una clausola unitaria perciò la lista corrispondente contiene due elementi, uno dei quali è lo 0 che è il carattere utilizzato come delimitatore della stringa. Oltre a rimuovere la lista, il metodo one_literal() ritorna anche il il letterale individuato perché quest'informazione è necessaria per l'operazione successiva: infatti, se esso è diverso da 0, viene invocato il metodo unit_clause() che si occupa di cercare, nelle altre liste, se è presente il letterale opposto a quello unitario; se viene trovato, attraverso il metodo remove(i) fornito dalla classe ArrayList() è possibile eliminare l'i-esimo carattere della lista, il quale rappresenta il letterale unitario negato. Viene inoltre fatto un controllo sulla dimensione della lista corrispondente del letterale appena eliminato: se tale dimensione è uguale a 1, significa che è presente solo lo 0 e che è stata generata la clausola vuota (è il caso in cui ci siano due clausole come ad esempio P and $\neg P$), in caso contrario l'esecuzione procede. Attraverso un ciclo **while**, quindi, vengono eseguiti i passi appena descritti fino a che vengono trovati letterali unitari.

Al termine di questi passi viene applicata la regola Pure Literal: il metodo pure_literal() va a scorrere l'insieme di clausole e, per ciascun letterale, viene invocato il metodo isPure(); tale funzione controlla se nelle altre liste è presente o meno il corrispondente letterale negato, in caso affermativo la procedura prosegue il controllo sugli altri letterali, in caso negativo allora significa che è presente un letterale puro perciò viene rimossa la lista che contiene tale letterale. Viene quindi eseguito un ciclo che prosegue fino a che un letterale puro viene trovato.

Al termine dell'esecuzione delle tre regole (Tautology Elimination, One Literal, Pure Literal) viene eseguito un controllo sulla dimensione delle liste per capire se ci sono ancora letterali presenti: se tutte le liste sono vuote oppure se presentano un solo valore (cioè lo 0), allora siamo riusciti a determinare se la formula è soddisfacibile o meno, altrimenti viene invocato il metodo split().
In questo metodo innanzitutto viene creata una lista che è la copia di quella (passata come parametro) che contiene tutte le clausole della formula, questo perché nel caso in cui lo split non andasse a buon fine con il letterale scelto (ovvero la formula risultasse non soddisfacibile), la procedura andà ripetuta con il letterale opposto sullo stesso array di liste. Poi viene scelto il letterale su cui effettuare lo split in modo abbastanza semplice: infatti viene considerata la prima clausola (ovvero la prima lista) in cui sia presente più di un letterale e quello prescelto è il primo della lista.
L'operazione successiva consiste nell'invocazione del metodo check_sat(): questo metodo va ad eseguire una procedura ricorsiva che verifica se il ramo intrapreso scegliendo il letterale per lo split porterà ad una soluzione o meno per l'insieme di clausole; nel caso in cui la procedura ritornasse il risultato di non soddisfacibilità , essa andrà ripetuta considerando come variabile di split il letterale opposto a quello scelto nell'iterazione precedente.
Il metodo check_sat() prima di tutto va a scorrere l'array di liste per rimuovere le clausole (liste) che contengono il letterale su cui è stato deciso di effettuare lo split, ed inoltre va a rimuovere i letterali che corrispondono all'opposto dell'array su cui si sta facendo lo split; dopo di che vengono nuovamente invocati in sequenza i metodi relativi alle regole One Literal e Pure Literal. Se ci sono ancora letterali presenti nelle clausole, significa che si rende necessario un ulteriore split e quindi viene invocato nuovamente il metodo split(). Perciò, viene eseguita una procedura ricorsiva fino a che si arriva alla conclusione che l'insieme di clausole sia soddisfacibile o meno.

TEST E ANALISI

Il tool sviluppato è stato testato su un insieme di istanze presenti online; l'analisi dei risultati è stata fatta tenendo conto del tempo di esecuzione impiegato: grazie al metodo currentTimeMillis() fornito dalla classe System sono stati misurati il tempo (in millisecondi) in cui ha inizio l'esecuzione e il tempo in cui ha fine, in modo così da poter avere una misura (che verrà fornita in secondi) del tempo complessivo di esecuzione.
L'insieme di clausole su cui è stata effettuata un'analisi è il seguente:

- *sat 20 91* raggruppa formule tutte soddisfacibili in cui il numero di lettere proposizionali è 20 ed il numero di clausole è 91; sono state testate le seguenti formule, per ciascuna delle quali è riportato il corrispondente

tempo di esecuzione:

$ISTANZA$	$T_{esecuzione}$
uf20-01.cnf	0.9 sec
uf20-02.cnf	0.4 sec
uf20-03.cnf	1.2 sec
uf20-04.cnf	1.4 sec
uf20-05.cnf	0.7 sec
uf20-06.cnf	1.0 sec
uf20-07.cnf	1.1 sec
uf20-08.cnf	0.3 sec

Tabella 1: *sat 20 91*

- *sat 50 218* raggruppa formule tutte soddisfacibili in cui il numero di lettere proposizionali è 50 ed il numero di clausole è 218:

$ISTANZA$	$T_{esecuzione}$
uf50-01.cnf	10 sec
uf50-02.cnf	35 sec
uf50-03.cnf	1.5 sec
uf50-04.cnf	29 sec
uf50-05.cnf	3 sec

Tabella 2: *sat 50 218*

- *unsat 50 218* raggruppa formule tutte non soddisfacibili in cui il numero di lettere proposizionali è 50 ed il numero di clausole è 218:

$ISTANZA$	$T_{esecuzione}$
uuf50-01.cnf	60 sec
uuf50-02.cnf	22 sec
uuf50-03.cnf	74 sec
uuf50-04.cnf	23 sec
uuf50-05.cnf	38 sec

Tabella 3: *unsat 50 218*

- *sat 75 325* raggruppa formule tutte soddisfacibili in cui il numero di lettere proposizionali è 75 ed il numero di clausole è 325:

$ISTANZA$	$T_{esecuzione}$
uf75-01.cnf	40 sec
uf75-02.cnf	84 sec
uf75-03.cnf	96 sec
uf75-04.cnf	304 sec

Tabella 4: *sat 75 325*

- *unsat 75 325* raggruppa formule tutte non soddisfacibili in cui il numero di lettere proposizionali è 75 ed il numero di clausole è 325:

$ISTANZA$	$T_{esecuzione}$
uuf75-01.cnf	223 sec
uuf75-02.cnf	242 sec
uuf75-03.cnf	664 sec
uuf75-04.cnf	311 sec

Tabella 5: *unsat 75 325*

- *sat 100 430* raggruppa formule tutte soddisfacibili in cui il numero di lettere proposizionali è 100 ed il numero di clausole è 430:

$ISTANZA$	$T_{esecuzione}$
uf100-01.cnf	1190 sec
uf100-02.cnf	2479 sec
uf100-03.cnf	1961 sec

Tabella 6: *sat 100 430*

- *unsat 100 430* raggruppa formule tutte non soddisfacibili in cui il numero di lettere proposizionali è 100 ed il numero di clausole è 430:

$ISTANZA$	$T_{esecuzione}$
uuf100-01.cnf	5252 sec

Tabella 7: *unsat 100 430*

CONCLUSIONE

Come è possibile evincere dai risultati dei test, il tempo di computazione aumenta con la complessità delle formule che vengono di volta in volta testate; ciò dipende dalla quantità maggiore di split che risultano necessari e che vanno ad influire sulla complessità di computazione.
Questo aspetto potrebbe essere migliorato in una futura revisione del tool andando a valutare, di volta in volta, su quale variabile effettuare lo split considerando il numero di volte in cui essa appare nella formula; così facendo, il numero complessivo di split diminuirebbe.
Infine, un ulteriore miglioria che si potrebbe apportare può essere il fatto di riuscire a far ritornare il modello che soddisfa la formula, nel caso ovviamente in cui essa risulti soddisfacibile.