Exploiting The Structure – Non Clausal Sat Solver

Daniele Zannella

Università degli Studi di Roma "La Sapienza"

Corso di Laurea Specialistica in Ingegneria Informatica

Tesina per il corso di Metodi Formali per l'Ing. del Software

EXPLOITING THE STRUCTURE – NON CLAUSAL SAT SOLVER

Autore:

Daniele Zannella

Docente:

Toni Mancini

Indice

Abstract

L'obiettivo di questa tesina è quello di far conoscere il panorama, per lo più ancora acerbo, dei risolutori SAT non clausali, ovvero risolutori che operano su formule booleane non Cnf, attraverso un excursus che va dallo studio di alcuni formati, come l'*EDIMACS* e *L'ISCAS*, mediante la generazione automatica di istanze di problemi visti a lezione (*ShopScheduling, Rostering e HamiltonVipCicle)*, per poi passare ad un'analisi sulle prestazioni di questi risolutori, confrontandoli anche con risolutori di tipo Cnf, come MiniSat, BerkMin, … Per concludere infine mostriamo in grandi linee, fornendo una rapida descrizione di sintassi e semantica, un nuovo formato, l'*AIGER*, nato in questi ultimi anni (2006/2007) che pian piano si sta imponendo nel mondo dei risolutori non-clausal per formule non-cnf senza perdita di struttura.

1. Introduzione

Il problema della soddisfacibilità di formule proposizionali, più comunemente conosciuto come SAT, è uno di punti cruciali, in quanto ad importanza, di diverse discipline scientifiche e tecniche; per citarne alcune, l'intelligenza artificiale, l'informatica teorica, la progettazione e verifica di hardware e software. La maggior parte delle procedure per la risoluzione di tale problema si basano su varianti dell'algoritmo di Davis-Putnamm-Logemann-Loveland (DPLL), che risolve istanze del problema in forma clausale, o CNF. Il formato CNF per molti anni è stato l'unico formato in grado di poter rappresentare questo tipo di problema e dunque la maggior parte dei risolutori oggi presenti sul mercato sono risolutori che operano su formule di questo tipo. Tuttavia la gran parte dei problemi, ha una rappresentazione in formule arbitrarie, e dunque per portarle in tale formato c'è bisogno di una trasformazione. Con il passare degli anni, alcuni teorici, tra i quali Thiffault e Bacchus, si iniziarono a chiedere il perché effettuare tale trasformazione, e se lasciare la formula nella sua forma originale potesse portare qualche beneficio, oltre al risparmio di tempo della conversione e l'aggiunta di variabili che questa comportava. Dall'analisi di questi aspetti si accorsero che decomporre la formula in CNF, in un certo senso, era come togliere un'identità alla formula stessa. Dunque proprio nella struttura originale della formula si potevano reperire informazioni utili per la risoluzione del problema, informazioni che durante la decomposizione andavano completamente perdute. Da allora iniziarono degli studi approfonditi sulla struttura intrinseca della formula, e cominciarono a nascere, come supporto a questa teoria i primi risolutori di formule non-clausali, dunque non-CNF. Quello che noi analizzeremo in questa tesina è proprio la nascita di questa teoria, alcune teorie e algoritmi a supporto di tale teoria, analizzando alcuni di questi risolutori e effettuando opportuni confronti con quelli CNF.

2 EDIMCS VS ISCAS

2.1 EDIMACS

In seguito alla sviluppo di solutori non-clausal si è iniziato a pensare ad un formato che potesse diventare una sorta di standard per formule booleane arbitrarie. Tutti questi sforzi si concentrarono sul formato Edimacs[1]. Gli sviluppatori, hanno sfruttato lo stesso principio, che ha reso famoso il formato Dimacs, il fatto che, essendo una sequenza di numeri, è facilmente leggibile e scrivibile. Dunque anche il nuovo formato Edimacs si presenta sintatticamente come una sequenza di numeri. Tuttavia presenta delle piccole restrizioni rispetto al Dimacs; restrizioni che riguardano la possibilità di poter scrivere su di una stessa riga diverse clausole. Questa riduzione di flessibilità va ricercata nel fatto che molti dei risolutori Sat non supportano tale funzione.

2.1.1 Semantica

Una formula in formato Edimacs descrive un circuito con un output e diversi input. La formula sarà *soddisfacibile* se e solo se è possibile trovare un insieme di valori, per le variabili di input, che mi fanno diventare TRUE il gate di output. Dunque il circuito consiste in una serie di interconnessioni tra gate, che rappresentano operatori, che sfociano in una *root*, che rappresenta l'output.

2.1.2 Sintassi

Andiamo adesso a vedere come è sintatticamente strutturato un file di tipo Edimcs.

2.1.2.1 Header

Nell'header troviamo le righe di commento, come accade per il precedente formato Dimacs, che iniziano con la lettera chiave "*c*" : (i.e.)

c Questo è un esempio di Commento

una riga di commento deve comparire sempre e solo all'inizio del file.

2.1.2.2 Problem Line

p noncnf VARS

come possiamo notare le differenze sono nella parola chiave "*noncnf*" invece di *cnf,* e nel fatto che non viene citato espressamente il numero delle clausole. Il parametro VARS indica il valore più grande assegnato ad una variabile, e come vedremo in seguito questo valore rappresenterà l'output del circuito.

2.1.2.3 Body

Ciascuna riga nel body descrive un gate che si trova all'interno della formula proposizionale con ulteriori informazioni; vediamo quali:

GATE #Parameters Parameter_1 .... Parameter_K IO0 .... IOn 0

analizziamo in maniera più dettagliata la formula.

- *GATE*: è un numero intero non negativo che rappresenta il tipo di operatore (*AND, OR, NOT, XOR, ...*) il formato specifica alcuni numeri riservati che si riferiscono ad operatori standard, e lascia liberi altri numeri per una futura espansione o per una possibile personalizzazione;
- *#Parameters*: questo è un numero positivo non zero, che specifica il numero di parametri che seguiranno. Se il numero indica -1, allora significa che non ci saranno parametri a seguito;
- *Parameter_i:* indica, se presente, il parametro i-esimo relativo al tipo di gate specificato; è sempre un valore intero.
- *IOi:* indica il valori di output e i valori di input rispettivamente; nel senso che IO0 -> output, mentre da IO1 in poi vengono specificati i valori di input, sempre in relazione al tipo di gate specificato. Valori negativi indicano variabili negate.
- *0 :* rappresenta come nel formato Dimacs il carattere di terminazione del Gate.

2.1.2.4 Root

Il nodo *root* rappresenta l'output del circuito ed è indicato dal numero rappresentato immediatamente dopo la *problem line,* in quanto questo valore rappresenta il massimo numero di variabili utilizzate nella codifica, e come detto su questo valore rappresenta appunto l'output del problema. Da notare che questa radice non è unica, ma in alcuni casi il circuito può presentare anche diverse uscite.

2.1.2.5 Gate Type

Per concludere forniamo una tabella in cui sono presenti tutti i tipi di gate rappresentabili con questo formato, e i rispettivi valori di parametri, input e output.

Tabella 1

GATE	Name	#_PARAMETERS	PARAMETERS	IOs	Comment
1	FALSE	-1	none	IO0 = output	always *false*
2	TRUE	-1	none	IO0 = output	always *true*
3	NOT	-1	none	IO0 = output, IO1 = input	unary negation, $IO0 = \neg IO1$
4	AND	-1	none	IO0 = output IO1...n = inputs	n-ary conjunction, $IO0 = IO1 \wedge \ldots \wedge IOn$
5	NAND	-1	none	IO0 = output IO1...n = inputs	n-ary NAND, $IO0 = \neg(IO1 \wedge \ldots \wedge IOn)$
6	OR	-1	none	IO0 = output IO1...n = inputs	n-ary disjunction, $IO0 = IO1 \vee \ldots \vee IOn$
7	NOR	-1	none	IO0 = output IO1...n = inputs	n-ary NOR, $IO0 = \neg(IO1 \vee \ldots \vee IOn)$
8	XOR	-1	none	IO0 = output IO1...n = inputs	n-ary XOR, $IO0 = IO1 \oplus \ldots \oplus IOn$
9	XNOR	-1	none	IO0 = output IO1...n = inputs	n-ary XNOR, $IO0 = \neg(IO1 \oplus \ldots \oplus IOn)$
10	IMPLIES	-1	none	IO0 = output, IO1 = input, IO2 = input	binary implication, $IO0 = \neg IO1 \vee IO2$
11	IFF	-1	none	IO0 = output IO1...n = inputs	n-ary equivalence, $IO0 = IO1 \leftrightarrow \ldots \leftrightarrow IOn$
12	IFTHENELSE	-1	none	IO0 = output, IO1..3 = input$_i$	ternary if-then-else, $IO0 = (IO1 \rightarrow IO2) \wedge (\neg IO1 \rightarrow IO3)$
13	ATLEAST	1	k	IO0 = output, IO1..n = input	at least k inputs are *true*, $IO0 = \sum_{i=1}^{n} IOi \geq k$
14	ATMOST	1	k	IO0 = output, IO1..n = input	at most k inputs are *true*, $IO0 = \sum_{i=1}^{n} IOi \leq k$
15	COUNT	1	k	IO0 = output, IO1..n = input	exactly k inputs are *true*, $IO0 = \sum_{i=1}^{n} IOi = k$
$\geq$ 10000					Gates numbered from 10000 can be used for application specifi c gates. The numbers 16–9999 are reserved for future expansion of the standard.

Come si può notare dalla tabella il potere espressivo del formato è notevole; possono essere utilizzati operatori di scelta o conteggio, oltre a quelli di parità, ecc…

2.1.2.6 Valid Benchmarcks

Un benchmarcks (o istanza) scritto in formato Edimacs è valido se oltre a rispettare le regole sintattiche viste finora, rispetta anche le seguenti condizioni:

1. Nessun valore di IO deve essere specificato come output di più di un gate;
2. Il gate root, di output, non può essere utilizzato come input di qualche altro gate.

Forniamo adesso un piccolo esempio di istanza non-cnf, Edimacs, confrontandola con la rispettiva istanza Cnf, Dimacs.

2.1.3 **Exemple**

Supponiamo di dover codificare la seguente semplice formula booleana:

$(a \wedge b) \rightarrow (c \wedge d)$

La formula di cui sopra non è in forma normale congiuntiva, ma noi possiamo ricavarcela mediante la codifica Tseitine e con l'aggiunta di opportune variabili e saltando i passaggi intermedi, arriviamo ad avere al forma Cnf voluta:

(-F1 ∨ a) ∧ (-F1 ∨ b) ∧ (-a ∨ -b ∨ F1) ∧ (-F2 ∨ d) ∧ (-F2 ∨ c) ∧ (-d ∨ -c ∨ F2) (-F3 ∨ -F1 ∨ F2) ∧ (F1 ∨ F3) ∧ (-F2 ∨ F3)

come possiamo osservare la codifica in forma clausale richiede l'aggiunta di nuove ulteriori variabili, in questo caso 3 <F1, F2, F3>.

A questo punto il file Dimacs è presto trovato:

```
c  Istanza DIMACS di esempio
c --- Start Dictionary
c  a -> 1
c  b -> 2
c -> 3
c  d -> 4
c  F1 -> 5
c  F2 -> 6
c  F3 -> 7
c --- End Dictionary
p cnf 7 9
-5 1 0
-5 2 0
-1 -2 5 0
-6 4 0
-6 3 0
```

-3 -4 6 0
-5 -7 6 0
5 7 0
-6 7 0

Andiamo a vedere invece come viene codificata l'istanza Edimacs a partire dalla formula iniziale; possiamo utilizzare diverse porte (gate) per codificare il testo ne vediamo due. Il modo più diretto è utilizzare direttamente l'operatore di implicazione materiale "binaria", che riceve due input e restituisce come output l'implicazione dei due:

c Istanza EDIMACS di esempio
c a -> 1
c b -> 2
c c -> 3
c d -> 4
p noncnf 7
4 -1 5 1 2 0
4 -1 6 3 4 0
10 -1 7 5 6 0

2.2 ISCAS

Il secondo formato non-clausal che abbiamo analizzato è il formato ISCAS. È un tipo di formato non utilizzato da moltissimi risoluti, e per questo la sua documentazione è "quasi" inesistente. Tuttavia sono riuscito per grandi linee a comprenderne alcuni aspetti sintattici nonché semantici; il formato risulta molto meno potente del precedente e anche meno flessibile a causa delle limitazioni nel tipo di porte logiche utilizzabili AND, OR e NOT. Inoltre non è possibile definire dei commenti nel codice. La struttura appare abbastanza standard fatta inizialmente di un elenco in cui vengono definite tutte le variabili di input seguite dalla dichiarazione di un'unica variabile di output attraverso la parola chiave out. Per chiarir meglio è opportuno fornire un esempio, prendendo sempre la formula definita in precedenza:

// Istanza di Esempio ISCAS
INPUT(v_1)
INPUT(v_2)
INPUT(v_3)
INPUT(v_4)
OUTPUT(out)
g_1 = AND(v_1, v_2)
g_2 = NOT(g_1)
g_3 = AND(v_3, v_4)
out = OR(g_2, g_3)

Nell'ambito delle sperimentazione vengono utilizzati dei tool molto utili per la conversione di istanze dal formato Edimacs a quello Iscas (Edimacs2Iscas) e viceversa
(Iscas2Edimacs)[2]

3 Non-Clausal Sat Solver

I migliori "Sat Solver" adesso in circolazione utilizzano algoritmi di tipo DPLL che sono basati su formule Cnf, ovvero formule in formato clausale. Tuttavia la maggior parte delle formule proposizionali inizialmente sono tutte in un formato arbitrario e dunque non-cnf. Quindi per poter utilizzare questi potenti solutori bisogna trasformare queste formule arbitrarie in formato clausale. Alcuni studiosi hanno affermato che questa conversione causa la perdita di informazioni sulla struttura originale della formula, informazioni che potrebbero invece essere utilizzate per migliorare e potenziare la ricerca di soluzioni. Ed è proprio su questo principio che nascono alcuni tool molto importanti che operano su formule in formato arbitrario e che tentano di minare le basi del formato cnf e dei suoi risolutori.

4 NoClause

Il primo tool che andiamo ad analizzare in questo senso è conosciuto sotto il nome di "no-Clause" [2]. È un tool che utilizza sempre come algoritmi risolutivi, algoritmi basati su tecnica DPLL e come vedremo i risultati sono molto interessanti per due motivi: il primo è che nell'implementazione non vengono utilizzati ottimizzazioni a livello di cache, e nonostante ciò i risultati sono vicini a quelli di risolutori cnf come Zchaff; il secondo è che ci sono molte altre potenziali vie per esplorare la struttura della formula originaria oltre a quella implementata, e questo fa si che ci sono molte altre possibilità di poter rendere più efficiente la risoluzione.

Prima di procedere alla descrizione dell'algoritmo illustriamo velocemente alcuni problemi che si introducono con la conversione della formula in cnf. Oltre al problema principale visto sopra, (quello della perdita di info sulla struttura) ne abbiamo uno nuovo, che è legato all'introduzione di nuove variabili durante la conversone. Supponiamo che sia disponibile l'informazione che ci permette di distinguere tra variabili originali (input) e variabili derivate (internal signal) (oltretutto non è sempre possibile avere questo tipo di informazione). Un primo problema evidente che nasce riguarda l'aumento esponenziale dello spazio di ricerca. Questo problema da alcuni risolutori (come LSat) è risolto restringendo lo spazio di ricerca (di branching) alle sole variabili originali (input), assumendo che quelle derivate siano poi appunto "derivate" dalla propagazione dei valori di quelle originali. Questo approccio permette di lasciare costante lo spazio di ricerca; sfortunatamente però questa semplice tecnica rende la risoluzione poco robusta, difatti i risolutori basati su tale metodo raggiungono delle prestazioni rilevanti su alcune classi di problemi, ma risultati poco incoraggianti su altre. Inoltre a rendere poco attrattiva questa tecnica di branching c'è anche il fatto che un'analisi condotta anche su variabili derivate permette la costruzione di un albero di dimensione costante, rispetto al numero di variali derivate, per la refutazione della formula. A tal proposito introduciamo un teorema:

Teorema 1: "*Esiste una famiglia di circuiti booleani per la quale esiste una breve risoluzione per refutazione se e solo se è permesso fare branching sulle variabili, cosiddette, derivate".*

Dunque si può evincere che l'idea è quella di fare branching su questo tipo di variabili derivate. A questo punto occorre analizzare in maniera più approfondita quali possono essere le cause di inefficienza che questo tipo di branching può causare. Un primo aspetto che si può osservare è che non sempre una procedura di tipo DPLL mi causa un aumento dello spazio di ricerca, quando mi va ad aggiungere variabili derivate.
Esempio: supponiamo di avere una formula con tutte variabili originali *X1, ..., Xn*, mediante l'algoritmo vengono aggiunte le seguenti variabili derivate, *Y1,...,Yn*. Supponiamo che nella formula siano presenti le seguenti clausole, *(-Xi, Yi) e (-Yi, Xi);* questo rende ogni variabile derivata Yi equivalente ad ogni variabile diretta Xi. Dunque l'aggiunta, in una formula di questo tipo, non avrebbe effetti sulla dimensione dello spazio di ricerca; difatti dall'assegnazione dell'una dipende l'altra.

Un'altra sorgente di inefficienza, ben più rilevante nel branching su variabili derivate, riguarda senza dubbio il comportamento che l'algoritmo assume nei confronti di particolari tipi di variabili, definie *don't care.*
Definizione: " *Sia M un modello della formula, una variabile x si definisce don't care se in seguito ad assegnazioni effettuate a livelli precedenti porre M'=M[x=true] o porre M''=M[x=false] è la stessa cosa;, ovvero sia M' che M'' sono entrambi modelli della formula*" (*vedremo in seguito in maniera più approfondita come l'algoritmo si comporta su questo tipo di variabili*)

Fatta questa premessa iniziale sul perché gli sviluppatori e i ricercatori hanno voluto implementare un simile tool, andiamo a vedere in particolare come si sviluppa l'algoritmo di DPLL senza la conversione in forma clausale.

4.1 DPLL Without Conversion

L'algoritmo prende come input una formula proposizionale arbitraria e la trasforma in un "operator tree" ovvero un albero in cui ciascun nodo interno rappresenta un operatore logico (and, or, not,…), e in cui i nodi foglia rappresentano invece le variabili di input. Una volta effettuata questa trasformazione, l'albero viene compresso e tradotto in grafo diretto senza cicli (DAG), in cui per efficienza le clausole doppie vengono fuse insieme. Per ciascun nodo del grafo viene memorizzato:

- un identificatore univoco;
- una lista di nodi padre (in quanto in un DAG un nodo può avere diversi nodi padre);
- il tipo del nodo (variabile proposizionale, and gate, or gate,…);
- un valore di verità;
- il livello decisionale a cui il valore di verità del nodo cambia da "unknown" (per unknow si intende quando il nodo non ha ancora assegnato alcun valore di verità);
- la ragione per cui il valore di verità del nodo cambia.

Dato ciò il nostro obbiettivo diventa assegnare un valore di verità ai nodi e controllare che esista un assegnamento di verità tale che renda vero il nodo radice, che corrisponde al valore della formula o dimostrare che tale assegnamento non esiste.
Questa ricerca avviene naturalmente mediante un algoritmo di tipo backtracking (i.e. il DPLL), che sceglie tra le variabili che non sono ancora state etichettate una da assegnare e gli assegna: TRUE o FALSE. Effettuato questo assegnamento l'algoritmo lo propaga ricorsivamente in maniera consistente sugli altri nodi del DAG. Dove per "consistente" si intende che rispetti la logica del tipo di nodo. Spieghiamo questa ultima cosa mediante un esempio che illustra quattro regole di propagazione consistenti per l'operatore and;

1. Se un nodo and diventa TRUE, allora tutti i suoi figli devono necessariamente essere etichettati a TRUE;
2. Se un figlio di un nodo and diventa FALSE, allora immediatamente il FALSE dovrà essere propagato al nodo and padre;
3. Se tutti i figli di un nodo and divengono TRUE, allora bisogna propagare TRUE al nodo and padre;

4. Se il nodo and diventa FALSE, e i tutti figli tranne uno sono false, allora si deve propagare il FALSE anche all'unico nodo non ancora etichettato.ì

Naturalmente in una situazione di questo tipo si arriva ad una condizione di "conflitto" quando si tenta di associare ad un nodo due valori di verità diversi. Quando ciò accade allora l'algoritmo fa backtracking e cambia l'assegnazione alla variabile precedente. Rispetto a solutori cnf, questo algoritmo tiene traccia del set di nodi che hanno causato un conflitto, questo affinché se in futuro sta per accadere una situazione simile, in base all'esperienza sul passato questa viene evitate e ci si guadagna in termini di efficienza. Dunque ogni qual volta si verifica un conflitto, le cause del conflitto, dunque il conflict set , vengono salvate all'interno di un database clausale.
È sicuramente interessante spendere due parole su come avvenga la propagazione in maniera efficiente all'interno del DAG. Ci sono diversi modi per rendere più veloce questa propagazione, e adesso andiamo a vedere quali, facendo riferimento alle regole di propagazione citate sopra rispetto all'operatore and (per gli altri funziona in maniera per lo più simile). Per quanto riguarda le prime due regole la tecnica che viene utilizzata per aumentare l'efficienza è quella di dotare tutti i nodi di una lista di nodi padre divisa per tipi, ovvero: una lista di genitori or, una per i genitori and e così via… in maniera tale che quando un nodo diventa false (i.e.) è più facile individuare i nodi and, scorrere la lista per propagare immediatamente il risultato. Per quanto riguarda le altre due regole quello che si fa è utilizzare i cosiddetti "watches literal" che sono una sorta di nodi sentinella che hanno lo scopo di allertare il nodo padre quando avviene qualche cambiamento tra i suoi figli o viceversa quando avvengono dei cambiamenti nel nodo padre verso gli altri nodi figlio.

4.2 Exploiting the Structure

4.2.1 Don't care propagation

La propagazione di variabili definite "dont care" è particolarmente interessante, soprattutto in termini di efficienza spieghiamo di cosa si tratta mediante un semplice esempio.
Supponiamo di avere una formula così composta : PHPn ∨ (p ∧ q), dove PHPn il famoso Pigeon Hole Problem con n-pigeon che risulta insoddisfacibile e la sua dimostrazione per refutazione richiede uno spazio di ricerca esponenziale, mentre p e q sono due banali lettere proposizionali. Osserviamo quali sono gli ipotetici scenari di un normale risolutore Sat clausale: a seguito di una codifica, i.e. Tseiten, che mi associa nuove variabili a quelle della formula affinché questa diventi Cnf, abbiamo che la formula diventa: B3≡ (B1 v B2) , dove B1≡(PHPn) e B2≡(p∧ q), oltre naturalmente alle variabili che verrebbero fuori dalla trasformazione del problema PHPn. A questo punto supponiamo che il risolutore setta la variabile B3=TRUE e scelga come variabile di branching B2=TRUE; questa scelta fa si che immediatamente le due variabili proposizionali p e q vengano settate a TRUE. A questo punto la formula risulta soddisfatta, tuttavia il lavoro del risolutore non è ancora finito in quanto ci sono ancora le variabili generate dal problema PHPn da assegnare, ma avendo già trovato una soluzione che mi rende la formula TRUE, tutto ciò che il solutore deve fare è forzare l'uscita delle suddette variabili a FALSE, ed il lavoro è finito. Adesso supponiamo, come è lecito, che il solutore faccia la scelta sbagliata sul letterale derivato di branch ovvero

scelga per primo di porre B1=TRUE. Questa scelta per le prestazioni si rivelerà ben presto molto dispendiosa in quanto il risolutore come prima cosa cercherà di trovare una possibile assegnazioni di variabili per il problema del PHPn inutilmente in quanto tale problema è insoddisfacibile, e richiede per la dimostrazione per refutazione uno spazio di ricerca esponenziale. Effettuata la dimostrazione il solutore farà backtracking, setterà a FALSE B1 e conseguentemente mi andrà a mettere a TRUE p e q. Dunque ciò che abbiamo potuto notare è che un solutore clausale proprio perché deve decidere su variabili derivate può perder tempo a trovare l'assegnazione di variabili don't care, come vengono chiamate quelle del PHPn, invece che concentrarsi sulle altre. Questo problema purtroppo per i risolutori clausali anche quelli di ultima generazione non è superabile, in quanto non c'è informazione che possa orientare il solutore su una variabile o su un'altra. Ed è qui che si inserisce appunto la novità dei risolutori non-clausal, risolutori che mantengono informazioni sulla struttura, e che proprio grazie a questa sono in grado di scegliere di non esaminare le variabili don't care, e dunque evitare di perder tempo dietro a problemi che non c'è bisogno di risolvere al fine della soddifacibilità o non della formula finale. Ci sono due possibili tecniche per sfruttare la struttura del circuito a tale scopo:

- la prima, nominata "Gupta at al.", assegna alle variabili dei tag che indicano il fan-in e il fan-out della stessa rispetto al circuito. Tale informazione viene sfruttata per individuare quale clausola si riferisce a parti del circuito che non influenzano più il gate di output. Queste clausole vengono etichettate come "inattive" e dunque non vengono considerate nel computo finale. Può accadere che una clausola inattiva diventi attiva solo in seguito a backtrack dell'algoritmo:
- la seconda ("Safarpour et al.") invece è una tecnica, cosiddetta lazy, che può anche essere sfruttata da solutori cnf, in quanto effettua uno scan dell'intero circuito e utilizza questa scansione per indicare dinamicamente le variabili che divengono via via don't care. Queste variabili poi vengono tolte dell'insieme delle variabili di branch su cui può operare un solutore cnf.

La tecnica utilizzata in questo tool è principalmente la seconda, anche se con qualche miglioria, dovuta al fatto che qui non vengono mantenute entrambe le codifiche come avviene di sopra, ovvero quella del circuito originale e quella Cnf, inoltre per aumentare l'efficienza vengono utilizzati degli appositi nodi sentinella detti don't care whatches. Per capire come agiscono questi nodi sentinella forniamo di seguito un piccolo esempio.
Supponiamo che ci sia un nodo figlio unlabeled (non assegnato) di un nodo and che è stato indicato come FALSE. A questo punto rispetto al nodo and padre il valore del nodo figlio diventa irrilevante, in quanto qualsiasi valore esso assuma il nodo padre non cambia. Attenzione però, il suo valore non diventa don't care immediatamente, in quanto nella sua lista di nodi padre potrebbe essercene qualcun'altro che invece risente dei cambiamenti di valore del nodo. Così un nodo figlio diventa don't care solo quando il suo valore è irrilevante per tutti i nodi padre nella sua lista o se il nodo padre e divenuto da solo irrilevante ai fini del risultato finale.

4.2.2 Conflict Clause Reduction

Un ulteriore tecnica implementata dall'algoritmo che si fonda sulla conoscenza della struttura del circuito , è quella della riduzione delle clausole in conflitto. Per clausola in questo caso si intende una sottoformula consistente di nodi interni (i.e. and node, not node,etc…) e di nodi esterni (leaf node, ovvero gli input del circuito). Questa tecnica risulta abbastanza innovativa nel panorama dei sat solver ed anche abbastanza semplice ed intuitiva. Ogni volta che viene individuata una di queste clausole in conflitto ne vengono esaminate l'insieme di etichette assegnate alla variabili, e si nota se c'è qualche etichetta "ridondante" rispetto alla struttura del circuito; se la risposta è positiva queste etichette vengono semplicemente rimosse. Noi diremo che un'etichetta l fa diventare una etichetta l' ridondante, se c'è una regola di propagazione all'interno del DAG che genera l' da l.
Esempio: Supponiamo che all'interno del nostro DAG ci sia un and-node interno *n*, con uno dei suoi figli *n'*. Adesso supponiamo che si abbia una clausola del tipo (*n,n',x,...);* noi potremmo ridurre questa clausola in questo modo: *(n,x,...),* se durante l'esecuzione dell'algoritmo ad un certo punto si assegna n=FALSE. Questo tipo di assegnazione, difatti, rende immediatamente l'assegnazione del nodo *n' ridondante,* in quanto se *n'* come detto precedentemente è uno dei figli di *n*, allora si avrà che *n'=False =>n=False,* dando origine alla clausola [*–(n'=flase) V n=false],* quindi *(n'=true, n=false)*. Che se confrontata con quella originale corrisponde proprio alla riduzione effettuata.
Dunque quello che segue è la rimozione dalla clausola in conflitto del letterale n', in quanto non porta valore aggiunto.

4.3 Risultati empirici

Il tool come input prende naturalmente una formula preposizionale arbitraria in formato ISCAS, utilizzando la struttura del circuito per ridurre tutte le clausole analizzate ad 100 o meno. Questo formato permette di rappresentare una formula in formato non-Cnf grazie all'utilizzo di una sintassi molto flessibile. Il formato non è per nulla ben documentato, e nel corso della tesina ne forniremo una breve trattazione. Come solutore Cnf di paragone per questi confronti viene utilizzato zchaff benché esso non sia il miglior Sat solver in circolazione, avendo il codice open source è stato possibile da parte degli implementatori replicare le stesse euristiche di braching le stesse tecniche di gestione del db clausale in maniera tale da minimizzare le differenze e enfatizzare il fatto che uno (NoClause) si basa sulla struttura del circuito e l'altro (zchaff) no, dunque o benefici nel lasciare la formula in formato non-cnf.
Un'altra piccola restrizione è dovuta al fatto che come detto l'input del tool e un file di tipo ISCAS, questa codifica oltre ad essere di difficile implementazione soffre anche di certa una perdita di struttura in quanto può contenere solo porte logiche di tipo and, or e not. (come noteremo in seguito questa è una limitazione di cui soffre anche l'altro nostro solutore non-cnf SatMate).
Fatte queste premesse andiamo a vedere i risultati utilizzando un processore da 2.4 GHz e 3 GB di Ram.

- Tabella 1

	ZCHAFF				NOCLAUSE			
Benchmark	**Time**	**Decisions**	**Impl/s**	**Size**	**Time**	**Decisions**	**Impl/s**	**Size**
sss-sat-1.0 (100)	128	2,970,794	728,144	70	225	1,532,843	616,705	39
vliw-sat-1.1 (100)	3,284	154,742,779	302,302	82	1,033	4,455,378	260,779	55
fvp-unsat-1.0 (4)	245	3,620,014	322,587	326	172	554,100	402,621	100
fvp-unsat-2.0 (22)	20,903	26,113,810	327,590	651	4,104	5,537,711	267,858	240

Nella Tabella_1 viene mostrato il comportamento dei due solutori a confronto su quattro suites (insieme di problemi) codificati sia in Cnf che in ISCAS, rispetto ai sec della cpu necessari per risolvere l'intera suite (time), al numero di branching nell'albero di ricerca, il tasso di propagazione per sec e infine la dimensione media di una clausola di conflitto. Da tutte queste informazioni osserviamo che fatta eccezione per il primo insieme di problemi (sss-sat-1.0) il solutore non-cnf presenta delle interessanti performance sia in intermini di Cpu/sec, sia di numero di branching, sia in termini di dimensione della conflict clause. Come possiamo notare il comportamento di NoClause è molto più efficiente per problemi più difficili (vedi fvp-unsat2.0(22) rispetto all'altro); anche per questo le prestazioni per la prima suite di problemi, relativamente più semplice, risulta inferiore rispetto a zchaff.

Tabella 2

	NOCLAUSE					NOCLAUSE without DON'T CARES			
Benchmark	**Time**	**Decisions**	**Step**	**Impl/s**	**DC/s**	**Time**	**Decisions**	**Step**	**Impl/s**
sss-sat-1.0 (100)	225	1,532,843	4.20	411,760	204,945	272	3,095,245	6.75	652,927
vliw-sat-1.1 (100)	1,033	4,455,378	6.32	175,995	84,784	2,120	13,208,363	10.86	381,188
fvp-unsat-1.0 (4)	172	554,100	3.95	212,012	190,609	494	3,442,123	12.42	295,179
fvp-unsat-2.0 (22)	4,104	5,537,711	3.03	186,603	81,255	30,934	20,382,047	3.18	335,242

Di seguito andiamo a fare un confronto tra due solutori NoClause mettendo in evidenza le caratteristiche sopra illustrate e mostrando come queste possano incidere in maniera pesante sulle prestazioni del sistema. Vediamo questi peggioramenti prestando attenzione alle colonne time (come su) e step, la quale si riferisce al numero di livelli, all'interno del DAG, che bisogna riattraversare in una situazione di backtrack dell'algoritmo, presenti nella Tabella_2.
Subito colpisce il peggioramento di NoClause privo di don't care in termini di tempo (sempre Cpu/sec), non solo rispetto al NoClause con don't care, ma anche rispetto a zchaff. Difatti le prestazioni in tre suite su quattro scendono al disotto dello zchaff. Questo è spiegato dal fatto che con l'utilizzo dei don't care il solutore evita di effettuare alcuni passaggi e dunque visita un numero inferiore di livelli in backtrack.

Tabella 3

	NoClause						NoClause without reductions			
Benchmark	**Time**	**Decisions**	**Impl/s**	**Size**	**Exam**	**Rem**	**Time**	**Decisions**	**Impl/s**	**Size**
sss-sat-1.0	225	1,532,843	616,705	39	90%	12%	228	1,624,312	628,953	52
vliw-sat-1.1	1,033	4,455,378	260,779	55	88%	11%	984	4,322,679	281,017	90
fvp-unsat-1.0	172	554,100	402,621	100	73%	11%	402	820,582	311,127	119
fvp-unsat-2.0	4,104	5,537,711	267,858	240	33%	16%	5,675	7,614,898	246,498	418

Per concludere andiamo a vedere questa ultima tabella, la Tabella_3, che ci mostra l'impatto sulle prestazioni che ha la tecnica di riduzione delle conflict caluse. Nella tabella abbiamo due nuove colonne, Exam, che si riferisce alla percentuale di conflict clause analizzate, e Rem, che ci da la percentuale di letterali eliminati per conflict clause. Notiamo che per le prime due suite di problemi, la riduzione non ha molto effetto, questo è spiegato dal fatto che i problemi sono più semplici e dunque le clausole sono già molto piccole; il miglioramento notevole si nota nelle suite fvp-unsat-1.0 e fvp-unsat-2.0, dove specialmente in quest'ultimo l'utilizzo della tecnica è in grado quasi di dimezzare la dimensione delle conflict clause. Un ultimo aspetto interessante da osservare nella tabella è che all'aumentare della dimensione delle conflict clause diminuisce la percentuale di conflict clause esaminate; questo però non deve trarre in inganno, in quanto anche riducendo solo alcune clausole l'impatto sulle altre può essere molto significativo (i.e. su fvp-unsat-2.0 un'analisi del 30% porta ad una riduzione di quasi il 50% della dimensione media).

Tali risultati sono tratti dall'articolo di Walsh e Bacchus[2].
~ Altri risultati empirici su semplici problemi saranno visualizzati in seguito.

4.4 Conclusioni

Dunque per terminare possiamo dire che alla luce di quanto visto la conversione Cnf non è necessaria in quanto un solutore come questo basato su un algoritmo DPLL non-cnf, riesce ad ottenere le stesse performance dei normali risolutori Cnf, anzi su alcune istanze di problemi risulta anche più veloce grazie alle tecniche innovative descritte sopra, prime fra tutti: 1) l'utilizzo dei watches per migliorare la propagazione attraverso l'albero di ricerca e 2) la tecnica di riduzione delle clausole. Infine questo solutore è il primo basato su struttura del circuito, che fa uso dei don't care, e della loro tecnica di propagazione.

5 Satmate

Molti solutori, anche non-cnf come quello visto, sono sviluppati sulla base di due principali strategie: Davis-Putnam-Logemann-Loveland (DPLL) o la local heuristic serch. Delle due la seconda non offre garanzie di completezza e dunque molti tra i più conosciuti Sat Solver come GRASP, SATO, zchaff, MINISAT, ecc… fanno uso della prima.
La proposta di Satmate[4] è quella di un solutore su formule arbitrarie non-cnf, dunque simile a NoClause, però a differenza del primo l'algoritmo utilizzato non è il famoso DPLL. L'idea sviluppata si basa sulla tecnica del "General Mathings" che di seguito andiamo ad esporre. La formula di input viene presa e tradotta senza perdita di struttura in una formula bidimensionale che va sotto il nome di "vhpform" ovvero "vertical-horizontal path form"; in questa forma le "congiunzioni" (∧) vengono indicate nei percorsi verticali, i cosidetti *vertical path,* mentre le "disgiunzioni" (∨) vengono indicate nei percorsi orizzontali, i cosiddetti *horizontal path*. Una formula è soddisfacibile se e solo se esiste un vertical path in cui non sono presenti due letterali opposti, ovv. l e –l. Una prima differenza che possiamo notare con l'algoritmo DPLL è che nel General Matings lo spazio di ricerca è l'insieme dei possibili vertical path; mentre nel DPLL lo spazio di ricerca è dato dall'insieme di tutti i possibili assegnamenti delle variabili. Un'ultima cosa da citare prima di addentrarci nella descrizione e riferita proprio all'assegnazione, riguarda il fatto che l'algoritmo può produrre delle assegnazioni "parziali", quando possibile; cosa che vedremo meglio in dettaglio in seguito.

5.1 Preliminaries

Prima di fornire una descrizione dell'algoritmo di Genaral Mathings dobbiamo introdurre alcune nozioni che serviranno in seguito nella trattazione. Per prima cosa introduciamo il concetto di negation normal form (nnf) affermando che una formula si dice nnf se e solo se contiene i connettivi logici ∧, ∨ e ¬, e tale che ogni occorrenza del not sia riferita ad una sola variabile proposizionale. C'è un teorema che dice: "ogni formula proposizionale è equivalente ad una formula scritta in nnf". Questo teorema viene sfruttato per determinare la rappresentazione di una formula nel nostro solutore, che è dunque in nnf. Naturalmente come detto precedentemente l'algoritmo utilizza un formato proprietario, il vhpform, che alla luce del teorema rappresenterà la combinazione di due formule nnf, una verticale e una orizzontale. Per chiarire meglio la questione facciamo un breve esempio. Supponiamo di avere la formula F = ((p ∨ q) ∧ -r ∧ -q) ∨ (-p ∧ (r ∨ –s) ∧q), allora diremo con riferimento alla figura 1:

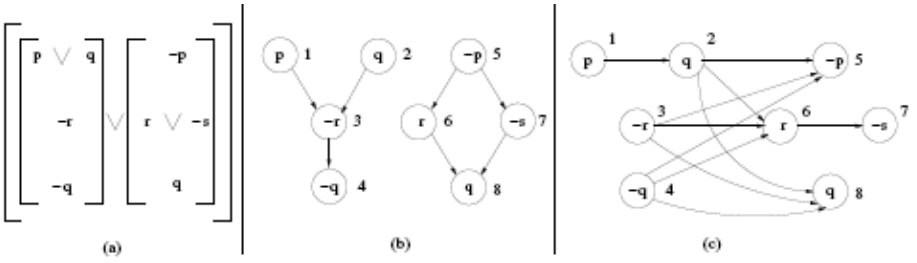

Fig. 1. We show the negation of a variable by a − sign. (a) vhpform for the formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ (b) the corresponding vpgraph (c) the corresponding hpgraph.

Vertical Path: è una sequenza di letterali singoli della formula iniziale, che si trovano a destra e a sinistra del raggio di applicazione di *ciascun* operatore di disgiunzione (∨). Nel nostro esempio avremo l'insieme di vertical path formati da {<p, -r, -q>, <q, -r, -q>, <-p, r , q>, <-p, -s, q>}. (*figura 1.a*)

Horizontal Path: viceversa è una sequenza di letterali singoli della formula iniziale, che si trovano a destra e a sinistra dell'operatore di congiunzione (∧). Nel nostro esempio avremo: {{<p, q, -p>, <p, q, r, -s>, <-p, q , q>, <-r, -p>, <-r, q>, <-q, -p>, <-q, r, -s>, <-q, q>}. (*figura 1.b*)

Detto ciò abbiamo citiamo di seguito due importanti teoremi che sono alla base della nostra procedura. Sia F la nostra formula, e sia σ un possibile assegnamento (totale o parziale):

Teorema1: diremo che σ soddisfa F se e solo se esiste un vertical path nella codifica vhpform di F tale che σ, appunto, soddisfa tutti i letterali del cammino.

Teorema 2: σ falsifica invece F, se esiste un horizontal path nella codifica vhpform di F, tale che σ falsifica tutti i letterali del cammino.

Dunque come emerge dal primo teorema, l'algoritmo verifica la soddisfacibilità mediante l'analisi dei percorsi verticali, mentre utilizza il secondo per individuare possibili conflitti (Figura 1 (b) e (c)).

5.2 Graph representation

Per sfruttare al meglio la forma vhpform, il solutore si crea in memoria una struttura a grafo che codifica esattamente la formula. Più che un grafo, se ne crea due, uno per i percorsi verticali e uno per i percorsi orizzontali, andiamo a vedere come.
VpGraph: descrive l'insieme di tutti i percorsi verticali della nnf "F"; viene indicato attraverso una tupla del tipo: Gv(F):=<V, R, L, E, Lit>, dove V rappresenta l'insieme dei nodi che è in corrispondenza biunivoca con l'insieme dei letterali della formula, R è l'insieme dei nodi root, ovvero i nodi iniziali, L è l'insieme ei nodi foglia, E è l'insieme degli archi, una sorta di transazioni, nei vertical path e infine la funzione Lit che associa ad ogni nodo del grafo un letterale. (Figura 1.b) Si dimostra in maniera induttiva, che qui non mostreremo che è possibile costruire il grafo anche induttivamente a partire dalla formula iniziale senza passare per la rappresentazione vhpform.
HpGraph: descrive tutti i percorsi orizzontali presenti nell'insieme hpform, ed è composto in maniera del tutto simile al "grafo verticale". (Gh(F):=<V, R, L, E, Lit>) (figura 1.c). Diamo qualche nozione di complessità computazionale: la costruzione di entrambi i grafi ha un costo di O(k), nel caso peggiore, dove il k indica la dimensione della formula. Andiamo a fornire una serie di definizioni interessanti che ci serviranno in seguito:

- r-path (rooted path): un percorso π in G(F), si dice r-path, se e solo inizia con un nodo che appartiene all'insieme R;
- rl-path(root-leaf path): un percorso invece si dice rl-path se e solo inizia con un nodo radice e termina con un nodo qualsiasi che appartiene all'insieme L;
- conflict-nodes: due nodi n1, n2 ∈ V si dicono in conflitto se e solo se

Lit(n1) = - Lit(n2), ovvero due nodi che corrispondono al medesimo letterale ma sono di segni opposti;

Dunque giunti a questo punto dobbiamo riformulare la nozione di soddisfacibilità per la rappresentazione a grafo della nostra iniziale formula arbitraria F . per far questo ci serviamo di due corollari che derivano direttamente dai due teoremi precedenti; data una formula F e un assegnamento σ di variabili, anche parziale, ho:

Corollario 1: un assegnamento σ "soddisfa" F se e solo se esiste almeno un rl-path π appartenente a Gv(F) tale che σ soddisfa tutti i letterali di π.

Corollario 2: una assegnamento σ "falsifica" F se e solo se esiste un rl-path π appartenente a Gh(F), tale che σ falsifica tutti i letterali di π.

A questi ne aggiungiamo altri due:

Corollario 3: Una formula F si dice soddisfacibile se e solo se esiste un rl-path π in Gv(F) tale che π è soddisfacibile.

Corollario 4: un percorso π di G(F) è soddisfacibile se e solo se non contiene due nodi in conflitto.

Input: vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ and hpgraph $G_h(\phi) = (V', R', L', E', Lit')$
Output: If $G_v(\phi)$ has a satisfiable rl-path return SAT, else return UNSAT
Algorithm:

```
1: st ← R                                  //push all roots in G_v(φ) on stack st
2: σ ← ∅                                   //initial truth assignment is empty
3: ∀n ∈ V : mrk(n) ← false                 //all nodes are un-marked to start with
4: while (st ≠ ∅)                          //stack st is not empty
5:    m ← st.top()                         // top element of stack st
6:    if (mrk(m) == false)                 //can we extend current r-path CRP with m
7:        if (conflict == prune())         //check if taking m causes conflict
8:            learn()                      //compute reason for conflict and learn
9:            backtrack ()                 //non-chronological backtracking
10:           continue                     //goto while loop (line 4)
11:       end if
12:       mrk(m) ← true                    //extend current satisfiable r-path with m
13:       σ ← σ ∪ {Lit(m)}                 //add Lit(m) to current assignment
14:       if (m ∈ L)                       //node m is a leaf
15:           return SAT                   //we found a satisfiable rl-path in G_v(φ)
16:       else
17:           push all children of m on st //try extending current r-path CRP⟨m⟩ to reach a leaf
18:       end if
19:   else                                 //backtracking mode
20:       backtrack ()                     //non-chronological backtracking
21:   end if
22: end while
23: return UNSAT                           //no satisfiable rl-path exists in G_v(φ)
```

Fig. 2. Searching a vpgraph for a satisfiable rl-path.

5.3 L'Algoritmo

Abbiamo dunque tutte le informazioni per analizzare il cuore del tool, l'algoritmo di risoluzione (Fig. 2). Il compito dell'algoritmo è quello di trovare all'interno del grafo G(F) un possibile rl-path, che sia soddisfacibile e in tal caso restituire SAT ; o d'altro canto verificare che tale cammino non esista e dunque restituire UNSAT. L'algoritmo è di tipo enumerativo, e come tale non dovrebbe essere efficiente, in quanto è esponenziale nella dimensione dell'input; però è stato provato che per formule del tipo dnf, ovvero una formula che rappresenta tutti e soli i cammini verticali della vhpform, il numero di rl-path risulta piccolo e polinomiale nella dimensione della formula. Inoltre per rendere più efficiente la ricerca vengono utilizzate tecniche di search space pruning, conflict driver learning, non-chronological backtracking.
Come possiamo osservare, l'algoritmo riceve come input un grafo G(F), e utilizza le seguenti strutture dati:

- st: rappresenta uno stack, all'interno del quale vengono memorizzati i nodi appartenenti a Gv(F); l'insieme viene inizializzato con i nodi dell'insieme root;
- σ rappresenta la corrente assegnazione; ciascun nodo di σ deve essere soddisfatto dall'assegnamento e la consistenza dell'insieme e data dal fatto che al suo interno non saranno mai presenti due letterali di segno opposto.
- mrk(): è una struttura che mi rappresenta il corrente r-path in Gv(F) (Fig. 3); nell'algoritmo si fa riferimento al root-path con l'abbreviazione CRP, e dunque ogni nodo n di st appartenente a CRP avrà mrk(n) =true.

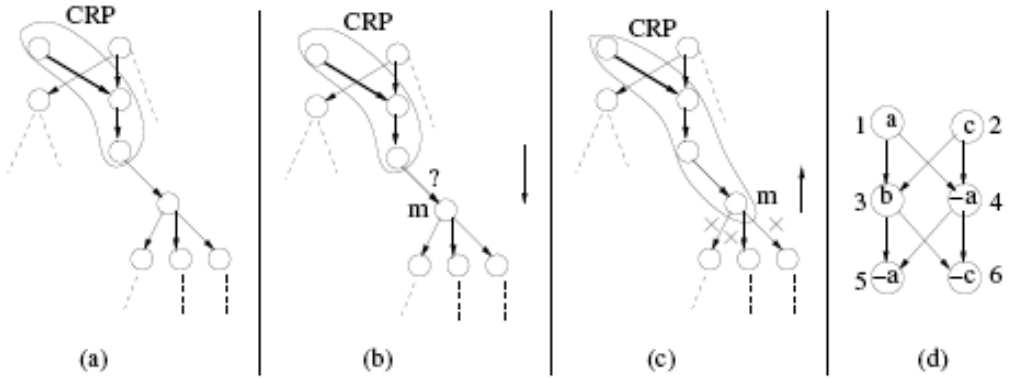

Fig. 3. (a) Current r-path or CRP in a vpgraph (b) Can CRP be extended by node m? (c) Backtracking from node m. (d) vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$

Il cuore dell'algoritmo è costituito dal ciclo while che termina o quando l'insieme st risulta vuoto (UNSAT) o quando trova un rl-path (SAT).
Da notare le tre routine si prune(), learn() e backtrack(), che di seguito andiamo a vedere più in dettaglio.

5.3.1 **prune()** -> Search Space Pruning

La prima funzione che andiamo ad analizzare è la funzione prune(), che ha lo scopo di controllare se il CRP può essere esteso (CRP<m>) con il nodo m, oppure no. Per essere esteso si intende, se passando per il nodo m è possibile arrivare d un percorso rl-path con prefisso CRP<m> che sia soddisfacibile. In caso contrario viene restituito Conflict. Quando la routine viene invocata la funzione di assegnazione σ soddisfa il cammino corrente, dunque ci sono tre possibili casi in cui si genera una situazione di conflitto:

- Caso1: è il caso standard, ovvero quando CRP<m> non è soddisfacibile; a questo significa che come l'algoritmo tenta di aggiungere il nodo m al root-path si verifica una situazione di conflitto tra il nodo m e un nodo appartenente a CRP, ovvero Lit(m)= -Lit(n) per qualche n di CRP;
- Caso 2 (Global Conflict): avviene quando l'assegnazione σ mi falsifica la formula. Tanto basta per poter affermare che non esiste alcun rl-path π in Gv(F), con prefisso CRP<m> che sia soddisfacibile. La dimostrazione avviene per assurdo: supponiamo che esista un rl-path, con prefisso CRP<m>, che sia soddisfatto dall'assegnazione σ'. Allora avremo sicuramente che se σ' soddisfa il cammino a maggior ragione soddisferà il vecchio CRP<m>, e l'insieme dei letterali di σ è sicuramente contenuto nell'insieme dei letterali di σ , t.c. (Lit(n) | n∈CRP<m>) ⊆ (Lit(n') | n'∈ π). Questo fa si che σ ⊆ σ'; ma se σ per definizione falsifica la formula allora non è possibile che σ', la soddisfi; dunque una contraddizione!
- Caso 3 (Local Conflict): avviene un conflitto locale in una situazione più interessante; quando in seguito all'estensione del CRP con m (CRP<m>) si determina che non sia possibile trovare nessun percorso dalla radice ad una foglia che sia soddisfacibile con questo prefisso. Dunque formalmente significa che ∀ π (rl-path) con prefisso CRP<m>, ∃ sempre un letterale l∈ π t.c. questo letterale sia in conflitto con un letterale –l ∈ CRP<m>, questo fatto determina che la formula non sia soddisfacibile.

Osservazione: da quanto visto, si evince che ogni conflitto Globale sarà anche un conflitto Locale, il viceversa non è sempre vero, in quanto il secondo come vincolo è più stingente.

5.3.2 **learn()** -> Learning

Questa funzione risulta molto interessante; essa ha lo scopo di memorizzare in un DB le "cause" del conflitto, quando avviene. Iniziamo con il dare la definizione di clausola come una disgiunzione di letterali. Allora una clausola sarà conflict se e solo se tutti i suoi letterali sotto la corrente assegnazione sono falsificati; altrimenti viene detta consistent. Ci sono due tipi di learning:

Global learning: una globaly learned clause (glc) di questo tipo deve essere mantenuta consistente indipendentemente dallo stato corrente, dunque del CRP. Una glc viene generata da una conflict-clause C, la quale può apparire in due modi:

- Ogni qual volta si genera un Global Conflict. In questa situazione difatti abbiamo una assegnazione che mi falsifica un percorso π rl-path nel grafo dei percorsi

orizzontali. L'insieme dei letterali dati dalla funzione Lit() e appartenenti al percorso costituiscono la clausola C = OR (Lit(n)), $\forall$ n $\in \pi$.

- Un seconda caso accade quando la glc è gia memorizzata, e a causa dell'assegnazione corrente, questa diventa falsa

Local Learning: una local learning clause (llc) viene generata ogni qual volta si scatena un Local Conflict ripetto ad una data variabile n. La clausola in questo caso è rappresentata dalla disgiunzione di tutti i letterali che mi creano il conflitto locale, ed essa associata proprio al nodo n in cui viene riscontrata. Inoltre la consistenza della clausola deve essere mantenuta soltanto qunado n fa parte del cammino corrente (CRP). Qui la differenza rispetto al caso di prima quando invece la consistenze delle glc i indipendente dal percorso corrente.

5.3.3 **backtrack()** ->Non Chronological Backtracking

La routine di backtrack() che appare all'interno dell'algoritmo varia a seconda del tipo di conflitto che la invoca.

- Non Chronological Backtracking on Global Conflict: quando si verifica un conflitto globale i letterali corrispondenti ai nodi del CRP<m> mi falsificano un rl-path di Gh(F), e da qui viene generata la global conflict clause C come visto sopra. Di questa clausola C solo uno dei letterali, ed in aprticlare l'ultimo cioè m, appartiene al correntre livello decisionale dell'albero di ricerca, mentre tutti gli altri nodi appartengono a livelli superiori. Il solutore identifica il cosidetto maxd(C), il livello decisionale più elevato ripetto ai nodi presenti nella clausola, e fa backtrack sul nodo presente a questo livello, e riprende la ricerca da qui.
- Non Chronological Backtracking on a Local Conflict: qunado si verica un Local Conflict, il solutore analizza la struttura del vpgraph, le cause del conflitto e prende una decisione coerente con le scoperte.

Ripetto a quest'ultimo per chiarire meglio facciamo un esempio.
Dalla Figura 4(a) possiamo osservare uno Gv(F); supponiamo di trovarci in una situazione in cui il nostro CRP è <1,2,3,5> e l'algoritmo tenta di estendere tale cammino con il nodo 7. A tal punto si verifica però un local conflict al nodo 7 che mi genera una conflict-clause del tipo (–a v –b v –c). In una situazione di questo tipo, l'algoritmo fa un backtrack per così dire "intelligente", ovvero esaminando la struttura del grafo si rende conto che tale conflitto può essere risolto solo facendo un backtrack al nodo 2, in quanto tutti gli altri cammini con prefisso (1, 2) passando per 7 soffrono dello stesso problema. Il backtrack al nodo 2 tenterà poi di estendere il cammino di prefisso (1, 2) in un nodo ancora non visitato che sia diverso da 3 o 4 ma non trovandolo fa backtrack su 1 e ritorna UNSAT. Come possiamo notare tale tecnica ci fa risparmiare il passaggio, peraltro inutile, sui cammini <1,2,3,6,7,…>, <1,2,4,6,7,…> e <1,2,34,5,7,…>.

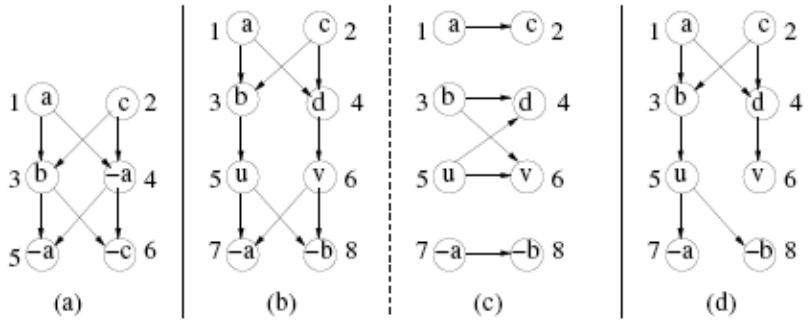

Fig. 4. (a) vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$. (b,c) vpgraph and hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$, respectively (d) vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$

Fig. 4

5.4 Empirical Results

Prima di addentrarci sulla spiegazione, bisogna dire alcune cose: il solutore accetta input in formato EDIMACS o ISCAS, con la restrizione di poter utilizzare solo porte and, or e not; gli esperimenti sono condotti su una macchiane di 1,5 GHz con 3 GB di Ram, su sistema operativo Linux. Il solutore non-clausale Satmate, viene confrontato con quattro solutori cnf MiniSat, BerkMin, Sage e Zchaff. Andiamo a vedere il comportamento su 3 benchmarks differenti.

QG6 benchmarks: questo banchmarks constite di circa 256 formule non clausali di media difficoltà. La versione non-cnf consiste in media di circa 300 variabili e 7500 gate; di contro la versione cnf, ottenuta in maniera diretta, implementando il problema di "classiyfing quasigroup" in formato Cnf ha 1700 variabili e 7500 clausole.

QG6* benchmarks: è una variante del problema precedente, dove si ottiene la formula Cnf, non in maniera diretta come prima ma effettuando la traduzione dalla formula non-clausale mediante l'aggiunta di nuove variabili.La nuova formula Cnf presenta 7500 variabili per 30000 clausole. Quella non-clausal resta la stessa.

Pigeon benchmarks: implementa il principio delle tane e dei piccioni,
date n e n+1 piccioni.

Le prime due suites contengono sia problemi soddifacibili che non, gli ultimi due hanno tutti problemi insoddisfacibili.

Osserviamo il comportamento nella Tabella 1:

Tabella 1

Bench -mark	#Probs	SatMate		MiniSat		BerkMin		Siege		zChaff	
		Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved
QG6	256	23266	235	49386	179	46625	184	46525	184	47321	180
QG6*	256	23266	235	37562	211	15975	239	30254	225	45557	186
Mboard	19	4316	12	4331	12	4947	11	4505	12	5029	11
Pigeon	19	5110	11	6114	9	5459	10	6174	9	5483	11

Nella tabella 1 vengono indicati due parametri: il tempo massimo (Time) nella risoluzione di ciascuna suite (si è stimato circa 10 minuti di timeout per problema), in termini di secondi di Cpu, includendo il tempo speso per la risoluzione dei problemi effettivamente risolti più il tempo speso per la risoluzione dei problemi irrisolti fino al timeout; il numero di problemi (Solved) prima del timeout.
Dai dati possiamo notare come per la prima suite di problemi (QG6), SatMate sia veloce quasi il doppio in termini di tempo di risoluzione degli altri solutori clausali e risolve quasi 50 problemi in più degli stessi solutori; per la seconda suite il comportamento è simile fatta eccezione per il Sat-Solver Cnf Berkmin che ottiene delle prestazioni maggiori. Questa differenza va trovata nella differenza di codifica del problema, come abbiamo visto su. Per gli altri due proemi le prestazioni di SatMate sono apparentemente migliori degli altri.
E interessante inoltre osservare il risultato che traspare dalla Tabella 2 dove i solutori sono paragonati su singoli problemi e a differenza di prima nella tabella troviamo delle nuove colonne: Local Conflict e Global Conflict, naturalmente legate a SatMate. In questo esperimento si è considerato un timeout (TO) di 1 ora per problema. Si nota che quando il numero dei Local Conflict scoperti si avvicina a quello dei Global Conflic, le prestazioni del tool sono migliori di quelle dei solutori cnf; quando il gap tra i due valori è rilevante, soprattutto in quando vengono evidenziati pochi local conflict, le prestazioni scendono di molto fino ad arrivare in alcuni casi al timeout.

Anche in questo caso altri risultati empirici saranno forniti in seguito. (vd. Appendice A)

6 AIGER: And-inverter-Graph

Prima di concludere la nostra trattazione ho pensato utile fare un cenno particolare ad formato uscito di recente (2007) che si offre come primo candidato a rimpiazzare i formati descritti in questa tesina.
Citiamo alcune date importanti:

- nel 2005 Ahiem Bacchus e Toby Walsh introdussero un formato nuovo per cercare di rappresentare formule arbitrarie, non-clausali, il formato eDimacs[2]. Tuttavia questo formato non raccolse molti consensi, difatti è anche pessimamente documentato.
- Nel 2006 Armine Bire introdusse un nuovo modo di scrivere formule non-clausali preservando la struttura, il cosiddetto AIGER format[5] che sfrutta la l'Iverter graph (AIG).
- Nel 2007 nasce la prima HardWare Model Checking Competition basata anche su AIGER format.

Andiamo a vedere in grandi linee in che cosa consiste questo formato.

6.1 AIGER: And Inverter Graph

AIGER è un nuovo tipo di formato molto potente che permette di modellare circuiti combinatori, di esprimere formule booleane senza perdita di struttura dunque di descrivere problemi per il Model Checking (SAT), modellare circuiti sequenziali, e molte altre cose, inoltre include una serie di librerie molto utili per la conversione da a formule cnf in aig format (cnf2aig) e viceversa (JAigerToCnf), oppure utility che mi permettono di ottenere da un file di tipo smv un file in formato AIGER (smvtoaig). Anche in questo formato le variabili vengono rappresentate mediante dei valori interi senza segno (unsigned).
Il formato può essere di due tipi: ASCII o binario. I formati sono molti simili tra di loro soprattutto nella sintassi, differiscono nel fatto che il primo è più permissivo nella sintassi e dunque risulta più leggibile da un utente umano, mentre il secondo è più restrittivo, meno comprensibile però più compatto e dunque a parità di modelli è di dimensioni molto inferiori rispetto al primo.
Facciamo anche un piccolo cenno sulla sintassi, del formato ASCII e sulla struttura del formato.
Il file inizia con una parola chiave, aag (nel formato binario la parola è aig), che sta per AIGER ASCII, e una sequenza di 5 interi non negativi rispettivamente rappresentati:

- M: il numero massimo di variabili utilizzabili;
- I: il numero di input;
- L: il numero di latch (questo per rappresentare appunto circuiti sequenziali)
- O: il numero di output;
- A: il numero di AND gates;

una cosa che prima non abbiamo detto è che nel formato sono indicate solo porte logiche AND (dopo forniremo un piccolo esempio su come ottenere altre porte).
Le righe che seguono rappresentano rispettivamente in numero e tipo i precedenti letterali.
Vediamo alcuni esempi banali per capire meglio.

6.2 Examples

Esempio 1: un circuito vuoto senza input ne output:

```
aag 0 0 0 0 0        // header
```

Esempio 2: la costante riservata FALSE viene indicata così:

```
aag 0 0 0 1 0        // header
0                    // output
```

Esempio 3: la costante riservata TRUE viene indicata così:

```
aag 0 0 0 1 0        // header
1                    // output
```

Esempio 4: vediamo adesso un circuito che realizza un AND:

```
aag 3 2 0 1 1        // header
2                    // input 0
4                    // input 1
6                    // output 0
6  2  4              // AND gate 0  --→ (1 & 2)
```

nell'header si afferma che ci saranno 3 variabili totali, 2 di input e 1 di output e inoltre che sarà presente una porta di AND; le variabili sono <1, 2, 3>. Attenzione alla differenza che viene fatta tra letterali e variabili; i letterali sono i valori che compaiono in input, in output, nella definizione dei latch e negli AND, e praticamente sono ottenuti dalle variabili moltiplicandole per 2. Ad esempio: il letterale 2 corrisponde alla variabile 1, in quanto 2*1 = 2; stessa cosa per il letterale 4=2*2 e 6=3*2. Dunque per ottenere le variabili dai letterali bisogna dividere per 2, viceversa per ottenere il letterali bisogna moltiplicare per 2. C'è un'altra da dire quando c'è la necessita di utilizzare delle variabili "negate", dato che come abbiamo dette gli unici valori che possono comparire sono letterali unsigned, quello che si aggiungere 1 al letterale stesso. Esempio: se io volessi ottenere la variabile -3, dorò utilizzare il letterale 7 = (2*3)+1. Per vedere se un letterale è positivo o negativo, basta prendere il letterale stesso farne il mod 2 e vedere se il risultato è 1 -> negativo, 0-> positivo.

Esempio 5: circuito che realizza un OR:

```
aag 3 2 0 1 1        // header
2                    // input 0
4                    // input 1
7                    // output 0 -→  ! ( !1 & !2 ) che mi realizza l'OR
6  3  5              // AND gate 0  -→ ( !1 & !2 )
```

Detto ciò ci sarebbe molto da aggiungere su questo formato, che in questi anni si sta imponendo su molti solutori SAT ma lo scopo di questa tesina era offrirne una panoramica generale sulla sintassi e sulla semantica.
(Potrebbe essere interessante un utilizzo di questo formato anche in combinazione con tool come NuSMV su modelli sequenziali.)

7 Conclusioni

Dall'analisi portata sui risolutori non-cnf mostrati in questa tesina abbiamo potuto notare come sia possibile risolvere istanze di problemi SAT, più o meno difficili, facendo a meno di modificare le formule proposizionali arbitrarie, per portarle in forma clausale. Abbiamo osservato che per molti problemi oltre ad essere possibile, la risoluzione, presenta anche delle prestazioni maggiori rispetto a molti solutori SAT Cnf presenti sul mercato, grazie all'utilizzo di informazioni presenti all'interno della struttura della formula. Il campo dei risolutori non-cnf e dei formati per rappresentare formule non-cnf, è un campo ancora molto inesplorato, e per questo anche poco documentato; tuttavia in questi ultimi tempi, grazie al fatto che si è scoperto che molte possono essere le tecniche di poter sfruttare la struttura di una formula arbitraria per migliorare le prestazioni di un risolutore, qualcosa in tal senso si sta muovendo. Tuttavia ancora c'è molto e molto da fare.
Per concludere all'interno dell'Appendice A sono presenti ulteriori test portati su questi risolutori, che vengono utilizzati per valutare istanze di problemi SAT quali, ShopScheduling, Rostering e HamiltonVipCicle, opportunamente create, sia in formato Edimacs che Iscas. Tali risultati si possono confrontare tra gli stessi risolutori e con risolutori Cnf (nel caso particolare Zchaff).

BIBLIOGRAFIA

1. Edimacs format, www.satcompetition.org/2005/edimacs.pdf
2. NoClause Sat Solver, www.cs.toronto.edu/~fbacchus/sat.html
3. Tool Converter, http://users.rsise.anu.edu.au/~ssiddiqi/
4. Satmate Sat Solver, www.cs.cmu.edu/~modelcheck/satmate/
5. Aiger Format, http://fmv.jku.at/aiger/FORMAT.aiger

APPENDICE A

I test sono stati svolti su una macchina 1.92 GHz e con 512 MB di Ram, su Sistema Linux. Il tempo indicato nelle colonne va inteso come, tempo della Cpu in secondi per cercare di risolvere il problema.

1° Test: "*Hamilton Vip Cicle*"

I test sono stati eseguiti mantenendo fissi tutti i parametri e studiando le variazioni nella creazione e nella risoluzione di istanze Cnf e Non, al variare della percentuale di BusLine, unico parametro considerato variabile. Il timeout viene impostato sulla risoluzione delle istanze, in quanto è il processo più inefficiente.

c Number of Destinations = 200
c Number of vip destinations = 40 (40% density)
c First destination ever home = 1
c N=BusLine (**Variabile**)

Risoluzione

BUS LINE	SATMATE	NO CLAUSE	ZCHAFF	MINISAT	Esito
80%	Kill	14.170	9.235		SAT
75%	Kill	15.160	10.848		SAT
70%	Kill	17.480	10.576		SAT
65%	Kill	17.170	11.708		SAT
60%	Kill	17.360	11.624		SAT
55%	Kill	20.190	12.820		SAT
50%	Kill	19.860	12.016		SAT
40%	Kill	18.600	TO		SAT
35%	Kill	20.260	TO		SAT
30%	Kill	18.320	TO		SAT

Creazione Istanze (timeout 1h)

BUS LINE	NCNF	CNF
80%	2m / 2.872 s	21m / 54.540 s
75%	2m / 41.958 s	17m / 57.867 s
70%	2m / 50.809 s	20m / 6.058 s
65%	3m / 4.825 s	21m / 46.567 s
60%	3m / 14.627 s	21m / 51.370 s
55%	3m / 28.548 s	24m / 49.674 s
50%	3m / 32.648 s	51m / 6.354 s
40%	3m / 55.151 s	TO
35%	4m / 8.025 s	TO
30%	4m / 22.354 s	TO

2° Test: "*Rostering*"

I test sono stati eseguiti mantenendo fissi tutti i parametri e studiando le variazioni nella creazione e nella risoluzione di istanze Cnf e Non, al variare del parametro che mi fissa il numero minimo di impiagati che devono lavorare per ogni turno, unico parametro considerato variabile. Il timeout viene impostato sulla risoluzione.

c Number of employees: 14
c Number of slots in each day: 3
c Number of night slots: 1
c Min number of employees that must be at work in each slot: **Variabile**
c Min number of break slots: 2
c Min number of working slots: 2

Risoluzione (timeout 1h:30m)

MinEmp AtWork	SATMATE	NO CLAUSE	ZCHAFF	MINISAT	Esito
2	1.6401	0.050	0.0041		SAT
3	12.2688	20.050	1.006		SAT
4	TO	TO	TO		TO
5	TO	TO	TO		TO
6	TO	6367.620	TO		UNSAT
7	KILL	982.620	1628.298		UNSAT
8	KILL	1379.300	1172.710		UNSAT
9	3216.672	440.730	104.887		UNSAT
10	437.927	211.380	40.137		UNSAT

Creazione Istanze

MinEmp AtWork	NCNF	CNF
2	0m / 0.314 s	0m / 1.295 s
3	0m / 0.660 s	0m / 3.049 s
4	0m / 1.845 s	0m / 8.188 s
5	0m / 5.716 s	0m / 18.433 s
6	0m / 7.056 s	0m / 31.480 s
7	0m / 10.614 s	0m / 40.674 s
8	0m / 8.900 s	0m/ 39.683 s
9	0m / 5.404 s	0m/ 29.947 s
10	0m / 3.066 s	0m/ 17.434 s

3° Test: "*Shop Scheduling*"

c Number of jobs=10
c Number of machines=10
c The Deadline = **Variabile**

Risoluzione(timeout 1h 30m)

Deadline	SATMATE	NO CLAUSE	ZCHAFF	MINISAT	Esito
150	TO	96.220	2751.071		SAT
145	TO	126.890	1054.345		SAT
140	TO	184.240	785.449		SAT
135	TO	207.810	1584.031		UNSAT
130	TO	89.560	5105.033		UNSAT
125	TO	1169.540	TO		UNSAT
120	TO	TO	TO		TO
115	TO	TO	TO		TO
110		TO	TO		TO
105	TO	TO	888.764		UNSAT
100	TO	698.400	64.800		UNSAT

Creazione Istanze

Deadline	NCNF	CNF
150	0m / 31.842 s	1m / 2.946 s
145	0m / 29.439 s	0m / 59.103 s
140	0m / 27.805 s	0m / 54.546 s
135	0m / 25.598 s	0m / 53.209 s
130	0m / 23.723 s	0m / 49.514 s
125	0m / 22.431 s	0m / 46.514 s
120	0m / 20.383 s	0m / 43.716 s
115	0m / 19.011 s	0m / 40.834 s
110	0m / 17.527 s	0m / 37.265 s
105	0m / 16.181 s	0m / 34.528 s
100	0m / 24.043 s	0m / 33.972 s