# Capitolo 3
# Il problema della soddisfacibilità booleana*

Silvio Ghilardi Silvio Ranise

January 28, 2009

**Abstract**

In questo capitolo, consideriamo la logica proposizionale ed il relativo problema di soddisfacibilità proposizionale, ovvero il problema di determinare la soddisfacibilità o meno di una formula arbitraria della logica proposizionale.

Dapprima si mostra la decidibilità del problema della soddisfacibilità proposizionale, ovvero l'esistenza di un algoritmo che risponde in maniera corretta alla domanda "la formula proposizionale considerata è soddisfacibile*?*" (in un numero finito di passi). A tale scopo si utilizza il metodo delle tavole di verità e si osserva che la complessità computazionale nel caso peggiore è esponenziale.

Quindi si passa a considerare altri metodi per risolvere il problema della soddisfacibilità proposizionale, basati sulla decomposizione sintattica della formula, metodo dei tableaux, oppure su alcune proprietà semantiche della logica proposizionale, metodo di Davis-Putnam-Loveland-Loggeman (DPLL). Si considera pure il metodo di risoluzione, in particolare una sua istanza detta risoluzione unitaria che, congiuntamente all'uso di alcune tecniche di decomposizione per l'analisi dei casi (sintattiche o semantica), mette in luce i vantaggi e gli svantaggi tra il metodo dei tableaux e quello DPLL.

Infine, si considerano i Binary Decision Diagram (BDD): una struttura dati particolarmente compatta che permette di rappresentare in maniera efficente i tableaux e che permette di risolvere oltre al problema della soddisfacibilità proposizionale anche quello dell'equivalenza logica tra due formule in maniera molto semplice. Infatti, una classe particolare di BDD, detta Reduced Ordered BDD (ROBDD), ha come proprietà fondamentale quella di essere una forma normale unica e pertanto, formule logicamente equivalente, hanno identica rappresentazione come struttura dati ROBDD.

*__Draft di un capitolo per un libro sulle applicazione della logica.__

# Contents

# 1 Logica proposizionale

Si caratterizza la logica proposizionale come un sotto-insieme della logica dei predicati del primo ordine.

## 1.1 Sintassi

Il linguaggio elementare (o segnatura) della logica proposizionale è la seguente quadrupla:

$$\mathcal{L}_P \quad := \quad \langle \mathcal{P}, \emptyset, \alpha_P, \beta_P \rangle$$

dove $\alpha_P(p) = 0$ per ogni $p \in \mathcal{P}$ e $\beta_P$ è una funzione costante arbitraria poichè il suo dominio è l'insieme vuoto. In altre parole, il linguaggio elementare $\mathcal{L}_P$ contiene predicati di arietà 0, che vengono anche chiamati *lettere proposizionali*. Siccome l'insieme dei simboli di funzione è vuoto (si vesa il secondo componente della quadrupla $\mathcal{L}_P$), la definizione della funzione di arietà perde il suo interesse poichè non si potranno formare termini. Inoltre, si considera l'insieme delle variabili individuali $\mathcal{V} = \emptyset$ e quindi si utilizzeranno soltanto i connettivi proposizionali per formare le formule, in quanto i quantificatori (sia universale che esistenziale) non possono essere applicati ad alcuna variabile individuale. Riassumendo, data la segnatura $\mathcal{L}_P$ di cui sopra, l'insieme degli $\mathcal{L}_P$-termini è vuoto mentre l'insieme delle $\mathcal{L}_P$-formulae si riduce al seguente:

- se $p \in \mathcal{P}$, allora $p$ è una formula e
- se $A_1, A_2$ sono formule, allora anche $(A_1 \wedge A_2)$, $(A_1 \vee A_2)$, $(A_1 \rightarrow A_2)$, e $(\neg A_1)$ sono formule.

## 1.2 Semantica

Data la segnatura $\mathcal{L}_P$ della logica proposizionale, la nozione di $\mathcal{L}_P$-struttura si semplifica come segue. Il dominio $\mathbf{A}$ della $\mathcal{L}_P$-struttura è l'insieme contenente i due valori di verità *vero* e *falso*, che nel seguito verrano ad essere frequentemente abbreviati con $\mathtt{1}$ e $\mathtt{0}$, rispettivamente. Questo è dovuto al fatto che la funzione di interpretazione $\mathcal{I}$ $\mathcal{L}_P$-struttura deve associare ad ogni $p \in \mathcal{P}$ con $\alpha_P(p) = 0$, un sottoinsieme $\mathcal{I}(p)$ di $\mathbf{A}^0$, ovvero l'insieme $\{*\}$ che contiene un solo elemento. Ora, vi possono essere solo due sottoinsiemi di tale insieme: l'insieme vuoto $\emptyset$ e l'insieme $\{*\}$ stesso, che, per convenzione, vengono identificati con *falso* e *vero*, rispettivamente. Riassumendo, la nozione di $\mathcal{L}_P$-struttura è la seguente:

- ad ogni $p \in \mathcal{P}$, si associa uno dei due valori di verità $\mathtt{0}$ oppure $\mathtt{1}$.

Prima di considerare come si semplifica la nozione di verità per la logica proposizionale, si osservi che la parte che può variare in una $\mathcal{L}_P$-struttura è

solamente la funzione di interpretazione mentre il dominio è sempre fissato. Di conseguenza, definiamo $v \models A$, ovvero 'la $\mathcal{L}_P$-formula $A$ è vera nella $\mathcal{L}_P$-struttura con funzione di interpretazione $v$', come segue:

- $v \models p$ sse $v(p) = 1$;
- $v \models A_1 \wedge A_2$ sse ($v \models A_1$ e $v \models A_2$);
- $v \models A_1 \vee A_2$ sse ($v \models A_1$ oppure $v \models A_2$);
- $v \models \neg A$ sse $v \not\models A$;
- $v \models A_1 \rightarrow A_2$ sse ($v \not\models A_1$ oppure $v \models A_2$).

Siccome la funzione di interpretazion $v$ ha come dominio l'insieme delle lettere proposizionali e come co-dominio l'insieme dei valori di verità, viene anche chiamata (funzione di) *assegnamento proposizionale*.

I concetti di teoria, di suo modello, sua soddisfacibilità, conseguenza logica e validità possono essere adattati in maniera banale alla logica proposizionale. Tradizionalmente, la nozione di validità è chiamta *tautologia* in logica proposizionale. La differenza principale tra la logica del primo ordine e quella proposizionale consta nel fatto che il problema di determinare la soddisfacibilità di una formula è indecidibile per la prima mentre è decidibile per la seconda. Siccome i problemi di conseguenza logica e validità possono essere ridotti, per refutazione, al problema di soddisfacibilità in tempo finito, ne segue che tali problemi sono indecidibili per la logica del primo ordine mentre sono decidibili per la logica proposizionale.

# 2 Il problema di soddisfacibilità

Il problema di soddisfacibilità della logica proposizionale è definito come segue. Data una $\mathcal{L}_P$-formula $A$, esiste un assegnamento proposizionale $v$ che rende vera $A$, ovvero tale che $v \models A$*?*

## 2.1 Tavole di verità

È facile mostrare che questo problema è decidibile grazie al metodo basato sulle *tavole di verità*, che, inoltre, è facilmente implementabile su un calcolatore. Sfortunatamente, questo metodo risulta proibitivamente complesso per formule che non siano di piccole dimensioni. Illustriamo il metodo sulla seguente formula

$$A \;:=\; (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2)$$

di cui vogliamo stabilire la soddisfaicibilità. A tale scopo, notiamo che $A$ contiene due lettere proposizionali $p_1$ e $p_2$. È ovvio che a $p_i$, un certo assegnamento proposizionale $v$ potrà dare valore 1 oppure 0, per $i = 1, 2$. Di

conseguenza, dovremo considerare $2 \times 2 = 4$ possibili assegnamenti proposizionali e calcolando il valore di verità di $A$, servendoci della definizione di verità data in precedenza, per ciascuno dei possibili assegnamenti proposizionali, potremo determinare se, per almeno uno di questi, $A$ risulta vera:

1. $v_1(p_1) = \mathsf{0}$ e $v_1(p_2) = \mathsf{0}$: $v_1 \not\models (p_1 \vee p_2)$ e $v_1 \models (p_1 \vee \neg p_2)$, quindi: $v_1 \not\models A$. $A$ non risulta vera per $v_1$.

2. $v_2(p_1) = \mathsf{0}$ e $v_2(p_2) = \mathsf{1}$: $v_2 \models (p_1 \vee p_2)$ e $v_2 \not\models (p_1 \vee \neg p_2)$, quindi: $v_2 \not\models A$. $A$ non risulta vera per $v_2$.

3. $v_3(p_1) = \mathsf{1}$ e $v_3(p_2) = \mathsf{0}$: $v_3 \models (p_1 \vee p_2)$ e $v_3 \models (p_1 \vee \neg p_2)$, quindi: $v_3 \models A$. $A$ risulta vera per $v_3$.

4. $v_4(p_1) = \mathsf{1}$ e $v_4(p_2) = \mathsf{1}$: $v_4 \models (p_1 \vee p_2)$ e $v_4 \models (p_1 \vee \neg p_2)$, quindi: $v_4 \models A$. $A$ non risulta vera per $v_4$.

Concludiamo pertanto che $A$ è soddisfacibile, poichè vi sono due assegnamenti proposizionali (ne bastarebbe uno solo!) che la rendono vera. Il metodo si presta ad essere organizzato in maniera sotto forma di tabella (da cui il nome di tavola di verità) dove le colonne sono identificate dalle lettere proposizionali e della (sotto-)formule in $A$ mentre le colonne sono etichettate dai possibili assegnamenti proposizionali:

| $p_1$ | $p_2$ | $(p_1 \vee p_2)$ | $A$ | $(p_1 \vee \neg p_2)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

In generale, se una formula $A$ contiene $n$ lettere proposizionali, la corrispondente tavola di verità contiene $2^n$ righe, ovvero esistono $2^n$ possibili assegnamenti proposizionali, per cui si deve determinare il valore di verità di $A$ servendosi della definizione di verità. Chiaramente, ci sono casi in cui si puó lasciare la tavola di verità incompleta poiché dopo poche righe si è riusciti a trovare un assegnamento proposizionale che soddisfa la formula. In ogni caso, per le formule che non sono soddisfacibili (ovvero per cui non esiste alcun assegnamento proposizionale che le rende vere) resta necessario computare il valore di verità per tutti i possibili assegnamenti proposizionali. Di conseguenza, nel caso pessimo, il metodo delle tavole di verità richiede di considerare un numero esponenziale di casi. Questo tipo di dipendenza rende rapidamente intrattabile il problema: per 10 lettere proposizionali, si devono considerare 1024 assegnamenti, per il doppio, ovvero 20 lettere proposizionali, gli assegnamenti da considerare sono poco piú di un milione*!* È facile convincersi come il metodo non abbia speranze di arrivare a determinare una soluzione per formule la cui dimensione (in termini di numero di lettere proposizionali che in esse occorrono) raddopia.

Sebbene la notevole difficoltà computazionale sia insita nel problema di soddisfacibilità proposizionale, tanto da farlo individuare come prototipo di quella classe di problemi (detti $\mathcal{NP}$) che viene considerata di impossibile risoluzione sui calcolatori, vedremo che esistono metodi che, accortamente implementati, possono risolvere istanze del problema di soddisfacibilità di dimensioni ragguardevoli a dispetto dell'enorme complessità teorica del problema.

## 2.2 Forme normali

Prima di considerare metodi alternativi a quello delle tavole di verità per risolvere il problema della soddisfacibilità proposizionale, è necessario considerare alcune classi di formule che godono di una struttura particolarmente semplice, cui ogni altra formula proposizionale puó essere ridotta preservando (almeno) la soddisfacibilità della formula di partenza. Inoltre, i metodi per stabilire la soddisfacibilità proposizionale assumono come precondizione che le formule di input siano in una di queste forme normali.

### 2.2.1 Forma normale negativa

Una formula è in *forma normale negativa* (fnn) se e solo se non contiene implicazioni e contiene negazioni solo di fronte a sottoformule atomiche. Due formule $A, B$ sono *logicamente equivalenti* se e solo se $A \leftrightarrow B$ è una verità logica. Per trasformare una formula $A$ in una formula $A'$ in fnn logicamente equivalente ad $A$, è sufficiente eseguire (in ordine qualunque, ma in modo esaustivo) le seguenti trasformazioni:

$$C \rightarrow D \quad \rightsquigarrow \quad \neg C \vee D \tag{1}$$
$$\neg\neg C \quad \rightsquigarrow \quad C \tag{2}$$
$$\neg(C \vee D) \quad \rightsquigarrow \quad \neg C \wedge \neg D \tag{3}$$
$$\neg(C \wedge D) \quad \rightsquigarrow \quad \neg C \vee \neg D \tag{4}$$

Le trasformazioni vanno viste come regole di riscrittura: ossia ogniqualvolta la formula corrente $A$ contenga una sottoformula del tipo indicato a sinistra, la si rimpiazzi con la corrispondente formula del tipo indicato a destra.

Il seguente teorema è conseguenza di un lemma generale di rimpiazzamento (su cui non ci soffermiamo) e del fatto che le formule a destra e a sinistra di $\rightsquigarrow$ nelle (1)–(4) sono logicamente equivalenti fra loro.

**Theorem 2.1.** *Ogni formula è logicamente equivalente ad una formula in fnn.*

Il metodo dei tableaux considera formule in fnn.

### 2.2.2 Forma normale congiuntiva

Un *letterale* è una lettera proposizionale oppure la sua negazione. Una *clausola* è una disgiunzione di letterali. Una formula proposizionale é in *forma normale congiuntiva* (fnc) sse è una congiunzione di clausole.

È possibile trasformare una formula proposizionale arbitraria in fnc usando alcune regole di riscrittura in due fasi. La prima fase consiste nello spingere la negazione all'interno verso le sottoformule: si possono utilizzare le regole fiste precedentemente per la trasformazione in fnn, ovvero le regole (1)–(4) che preservano l'equivalenza logica. La seconda fase consiste nel distribuire la disgiunzione rispetto alla congiunzione applicando la seguente regola di riscrittura:

$$A \vee (B \wedge B') \quad \rightsquigarrow \quad (A \vee B) \wedge (A \vee B'). \tag{5}$$

Anche questa regola preserva l'equivalenza logica.

Il problema principale della seconda fase della traduzione in fnc consta nel fatto che si possono ottenere formule esponenzialmente lunghe. Per rendersi conto di questo fatto, si osservi che a destra del $\rightsquigarrow$ nella regola (5), la sottoformula $A$ si trova duplicata due volte. Applicando esaustivamente questa trasformazione, si possono duplicare sottoformule di grandi dimensioni molte volte. Per rendersi conto più concretamente di questo fenomeno, un utile esercizio è quello di provare ad applicare la regola (5) su formule con la seguente struttura:

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee \cdots \vee (p_n \wedge q_n)$$

per valori crescenti di $n \geq 1$.

Per evitare questo problema, si sono sviluppate varie tecniche che sono riconducibili tutte alla stessa intuizione, ovvero utilizzare delle lettere proposizionali aggiuntive al fine di evitare di duplicare ripetutamente sottoformule che possono essere di grandi dimensioni. Piú precisamente, la trasformazione consta di due fasi. La prima è identica alla precedente trasformazione e consiste nell'applicazione esaustiva delle regole di riscrittura (1)–(4). Per quanto riguarda la seconda fase, si traduce ogni disgiunto $A \vee B$ che non è una clausola come segue: si costruisce *ricorsivamente* una formula equisoddisfacibile $C_A^p \wedge C_B^{\neg p}$ che contiene una "nuova" lettera proposizionale $p$ (nuova nel senso che non occorre in $A \vee B$) dove $C_A^p$ e $C_B^{\neg p}$ sono formule in fnc. Precisamente, si traduce in forma clausale $A$ in modo da ottenere $C_A$ e, similmente, si traduce $B$ in forma clausale per ottenere $C_B$. A questo punto, si aggiunge $p$ come disgiunto ad ogni clausola in $C_A$ in modo da ottenere $C_A^p$ e si aggiunge $\neg p$ come disgiunto ad ogni clausola in $C_B$ in modo da ottenere $C_B^{\neg p}$.

Vediamo di capire se questa traduzione preserva la soddisfacibilità della formula risultante rispetto a quella iniziale (non è possibile ottenere l'equivalenza logica tra le due formule in quanto vengono introdotte nuove lettere proposizionali che quindi richiedono di estendere gli assegnamenti proposizionali

della formula iniziale. Vi sono due casi da considerare: $A$ oppure $B$ è soddisfacibile e sia $A$ che $B$ non sono soddisfacibili.

1. Se $A$ è soddisfacibile, allora anche $C_A$ è soddisfacibile ed anche $C_A^p$ lo è. Inoltre, possiamo considerare l'assegnamento proposizionale $v'$ che è identico a quello $v$ tale che $v \models C_A^p$ tranne che per il valore attribuito alla lettera proposizionale nuova $p$ cui $v'$ attribuisce il valore $\mathsf{0}$: notiamo che anche $v' \models C_A^p$ ed inoltre, ovviamente, $v' \models C_B^{\neg p}$. Il caso in cui sia $B$ ad essere soddisfacibile si tratta simmetricamente.

2. Consideriamo ora il caso in cui sia $A$ che $B$ non sono soddisfacibili. Dato qualsiasi assegnamento di $A$, almeno una clausola di $C_A$ non è soddisfatta e quindi, questo impone che l'assegnamento che considera anche $p$ mappi questo a $\mathsf{1}$ in modo tale da rendere vera $C_A^p$. Similmente, se $B$ non è soddisfacibile, allora almeno una delle clausole in $C_B$ non è soddisfacibile e quindi l'assegnamento che considera anche $p$ deve mappere questo a $\mathsf{0}$ per rendere vera $C_B^{\neg p}$. Pertanto, un assegnamento arbitrario che non soddisfa $A$ e non soddisfa $B$ implica che $p$ sia mappata a $\mathsf{1}$ ed a $\mathsf{0}$ contemporaneamente, cosa imposibile. Di consegueza, la formula risultante è anch'essa insoddisfacibile.

Concludiamo dunque che la traduzione preserva la soddisfacibilità ed evita di generare formule esponenzialmente piú grandi di quelle iniziali al prezzo di aggiungere lettere proposizionali rispetto a quelle della formula iniziale. Questo implica che il metodo che deve stabilire la soddisfacibilità della formula risultante dovrà cercare l'assegnamento proposizionale opportuno in uno spazio di ricerca piú grande.

**Esempio 2.1.** *Consideriamo la formula seguente:*

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2)$$

*che è un'instanza (per $n = 2$) delle formule precedentemente considerate come esempio del problema di duplicazione e della conseguente esplosione esponenziale della dimensione delle formule in fnc. Applichiamo la traduzione descritta sopra ed otteniamo:*

$$(p_1 \vee r) \wedge (q_1 \vee r) \wedge (p_2 \vee \neg r) \wedge (q_2 \vee \vee r),$$

*dove $r$ è la lettera proposizionale nuova. Se avessimo utilizzato la regola di distribuzione avremmo ottenuto la seguente formula:*

$$(p_1 \vee p_2) \wedge (q_1 \vee p_2) \wedge (p_1 \vee q_2) \wedge (q_1 \vee q_2).$$

*In questo caso, le due formule sono di lunghezza eguale. Il vantaggio, in termini di compattezza della formula risultante, della seconda sulla prima traduzione iniziano ad essere visibili per $n \geq 3$. Il lettore, per rendersi conto di questo fatto, è invitato ad applicare le due traduzioni per alcuni valori di $n \geq 3$.*

L'algoritmo di risoluzione e quello di Davis-Putnam-Loveland-Logemann (DPLL) utilizzano formule in fnc come input.

### 2.2.3 Forma normale disgiuntiva

La forma normale disgiuntiva è la duale della fnc. Un *cubo* è una congiunzione di letterali. Una formula proposizionale è in *forma normale disgiuntiva (fnd)* sse è una disgiunzione di cubi.

Come per la fnc, è possibile transformare una formula proposizionale arbitraria in una logicamente equivalente in fnd: a tale scopo è sufficiente trasformare la formula in una in fnn come visto in precedenza e quindi applicare esaustivamente la trasformazione duale rispetto a (5), ovvero applicare in maniera esaustiva la seguente trasformazione:

$$A \wedge (B \vee B') \quad \rightsquigarrow \quad (A \wedge B) \vee (A \wedge B').$$

Ovviamente, anche questa trasformazione preserva l'equivalenza logica. Riconsidereremo la fnd più avanti quando si discuteranno le relazioni tra i tableaux ed i BDD e vedremo come i tableaux possono essere visti come un modo alternativo per trasformare una formula in fnn in una in fnd.

## 2.3 Tableaux

Per il resto di questo capitolo, fissiamo una segnatura proposizionale $\mathcal{L}$. In questa sezione, fissiamo pure una formula $A$ in fnn di cui vogliamo determinare la soddisfacibilità o meno.

Di seguito utilizzeremo le lettere greche $\Gamma, \Delta, ...$ per indicare insiemi finiti di $\mathcal{L}$-formule proposizionali; notazioni come $\Gamma, C$ e $\Delta, C, D$ verranno usate come abbreviazioni di $\Gamma \cup \{C\}$ e $\Delta \cup \{C, D\}$, rispettivamente.

Il metodo dei tableaux costruisce un albero i cui nodi sono etichettati mediante insieme finiti di $\mathcal{L}$-formule; la radice dell'albero viene sempre etichettata dall'insieme $\{A\}$. Il metodo analizza l'assunzione che $A$ è soddisfacibile per ricavarne eventuali contraddizioni ed in questo caso si conclude l'insoddisfacibilità di $A$, altrimenti se ne deduce la soddisfacibilità. L'idea alla base del metodo è quella di analizzare la struttura di $A$ e di decomporla in modo opportuno a seconda dei connettivi logici in essa contenuti a partire dal connettivo logico principale, tenendo conto della definizione di verità data in precedenza. Più precisamente si procede nel modo seguente: se seleziona una foglia dell'albero corrente che non sia *chiusa*, ovvero che non contenga sia una formula atomatica che la sua negazione; se tutte le foglie sono chiuse allora la formula viene dichiarata insoddisfacibile e si dice che il metodo termina con una *refutazione* della soddisfacibilità di $A$. Se al nodo foglia preso in considerazione nessuna delle due regole di espansione seguenti si applica, allora la foglia viene detta *terminale* e la formula è dichiarata soddisfacibile. Altrimenti, se una delle regole di espansione è

applicabile, si aggiungono sotto la foglia uno oppure due nuovi nodi figli, etichettandoli con le formule nella conclusione della regola applicata. Le regole di espansione dell'albero sono le seguenti due, una per ogni connettivo logico binario contenuto in una formula in fnn:

$$\frac{\Gamma, B \wedge C}{\Gamma, B, C} \qquad\qquad \frac{\Gamma, B \vee C}{\Gamma, B \quad || \quad \Gamma, C}$$

La regola a sinistra si dice *regola di espansione della congiunzione* mentre quella a destra si dice *regola di espansione della disgiunzione*. Il significato della regola di espansione della congiunzione è il seguente: se abbiamo trovato un assegnamento proposizionale per cui tutte le formule in $\Gamma$ e la $B \wedge C$ sono vere, allora abbiamo trovato anche un assegnamento proposizionale che rende vere tutte le formule in $\Gamma$, la $B$ e la $C$. La regola di espansione della disgiunzione, invece, dice che se abbiamo trovato un assegnamento proposizionale in cui tutte le formule in $\Gamma$ e la $B \vee C$ sono vere, allora abbiamo anche trovato o unassegnamento che rende vere tutte le formule in $\Gamma$ e la $B$ oppure rende vere tutte le formule in $\Gamma$ e la $C$.

In linea di principio, le regole di espansione si applicano in ordine qualunque, ma scegliere un criterio di applicazione piuttosto che un altro può avere un grosso impatto in negativo o positivo sulle prestazioni del metodo. In ogni caso, è facile rendersi conto che l'applicazione esaustiva delle due regole di espansione e la verifica del fatto che le foglie siano chiuse o terminali, fornisce una procedura di decisione per il problema della soddisfacibilità proposizionale, ovvero un algoritmo capace di determinare se una qualunque formula proposizionale in fnn è in soddisfacibile o meno. Infatti, la correttezza delle due regole di espansione è una ovvia conseguenza della nozione di verità dei connettivi logici $\wedge$ e $\vee$, rispettivamente; mentre la terminazione è scontata poichè il numero massimo di volte che possono essere applicate le due regole di espansione è determinato dal numero di connettivi logici che occorrono nella formula $A$ di cui si vuole conoscere la soddisfacibilità.

Infine, notiamo come nel caso la formula $A$ sia soddisfacibile sia facile ricavare l'assegnamento proposizionale da una foglia terminale (ovviamente non chiusa): è sufficiente collezionare tutte le formule atomiche, ovvero i letterali $\ell_1, ..., \ell_n$ presenti nell'insieme di formule che etichetta la foglia e quindi costruire il corrispondente assegnamento proposizionale $v$ come segue: se $\ell_j = p$ dove $p$ è una lettera proposizionale, allora si pone $v(p) = \mathsf{1}$, se, invece, $\ell_j = \neg p$ dove $p$ è una lettera proposizionale, allora $v(p) = \mathsf{0}$, per $j = 1, ..., n$. È facile vedere che $v \models \ell_j$ per $j = 1, ..., n$, e, per la correttezza delle regole di espansione si ha pure che $v \models A$.

## 2.4 Risoluzione

Il metodo di risoluzione ha lo scopo di mostrare l'insoddisfacibilità di un insieme $S$ di clausole e consiste nell'applicazione esaustiva della seguente

regola di inferenza, detta *risoluzione proposizionale*:

$$\frac{p \vee C \qquad\qquad \neg p \vee D}{C \vee D}$$

dove $p$ è un letterale e $C, D$ sono clausole. Piú precisamente il metodo consiste nei seguenti passi: (i) selezione di due clausole $C'$ e $D'$ in $S$ che contengono una certa lettera proposizionale $p$ con polarità opposta e si dice che $p$ è il *letterale selezionato* per la risoluzione, ovvero $C'$ è della forma $p \vee C$ mentre $D'$ ha la forma $\neg p \vee D$ e si dice che $C'$ e $D'$ sono le *premesse* della regola, (ii) calcolo della *clausola risolvente* $C \vee D$ ed infine, (iii) aggiunta della clausola risolvente all'insieme delle clausole di partenza $S$, in modo da ottenere l'insieme $S'$. Il metodo ricomincia dal punto (i) fino a quando o (a) l'insieme delle clausole ha raggiunto un *punto fisso*, ovvero non si possono generare nuove clausole oppure (b) si è derivata la *clausola vuota*, denotata con $\square$. Nel caso (a), se l'insieme delle clausole che ha raggiunto il punto fisso non contiene la clausola vuota, allora si conclude che l'insieme iniziale delle clausole è soddisfacibile. Nel caso (b), si conclude che l'insieme iniziale delle clausole è insoddisfacibile.

### 2.4.1 Correttezza e terminazione

Notiamo che abbiamo assunto implicitamente che dalle clausole nell'insieme iniziale $S$ e da quelle aggiunte successivamente per applicazione della regola di risoluzione vengono eliminate eventuali occorrenze multiple dello stesso letterale. In fase di implementazione, questo puó essere realizzato adottando gli insiemi come struttura dati per memorizzare una clausola.

Inoltre, l'applicazione esaustiva della regola di risoluzione garantisce la terminazione dell'algoritmo per verificare la soddisfacibilità di un insieme di clausole: nel caso in cui venga generata la clausola vuota questo è ovvio. Altrimenti, basta osservare che il numero di lettere proposizionali che occorrono in un insieme finito $S$ di clausole è finito e pertanto si possono generare soltanto un numero finito di clausole senza doppioni (pari al numero di sottoinsiemi che si possono formare da un insieme contenente le lettere proposizionali e le loro negazioni): questo è il piǵrande insieme di clausole generabile anche per risoluzione e quindi il punto fisso viene raggiunto in un numero finito di applicazioni della regola di risoluzione.

Infine, notiamo che la regola di risoluzione è corretta poichè aggiunge all'insieme di clausole iniziale clausole che sono conseguenze logiche delle loro premesse. Per rendersi conto di questo, si noti che dato un assegnamento proposizionale $v$ tale che $v \models p \vee C$ e $v \models \neg p \vee D$, allora si ha immediatamente che $v \models C$ e $v \models D$. Per la nozione di verità proposizionale, si ha che $v \models C \vee D$, ovvero $C \vee D$ è una conseguenza logica di $\{p \vee C, \neg p \vee D\}$. In altre parole, i potenziali assegnamenti proposizionali (se ne esistono) che soddisfano le clausole in un certo insieme, vengono preservati dall'applicazione della regola

di risoluzione. Se viene derivata la clausola vuota, ovvero l'insieme vuoto di letterali, nessun assegnamento proposizionale puó rendere vera tale clausola, semplicemente perchè non è possibile che un letterale in essa sia assegnato a 1 poichè non vi sono letterali*!* Quindi è corretto concludere, nel caso (b), che l'insieme di clausole è insoddisfacibile in quanto non esistono assegnamenti proposizionali che rendono vera la clausola vuota e l'insieme dei possibili assegnamenti proposizionali viene preservato dall'applicazione della regola di risoluzione.

Siccome il metodo di risoluzione termina ed è corretto, possiamo dire che è una *procedura di decisione* per il problema di soddisfacibilità porposizionale (come lo sono le tavole di verità viste sopra), ovvero è un algoritmo che (in un numero finito di passi) fornisce una risposta al problema di determinare la soddisfacibilità di un insieme (finito) di clausole.

### 2.4.2 Implementazione

Il metodo di risoluzione descritto in precedenza richiede che si eseguano tutte le possibili applicazioni della regola di inferenza. Per implementare questo metodo, bisogna dunque ricordare quali sono quelle applicazioni della regola che sono già state eseguite. L'algoritmo utilizzato in molti dimostratori automatici di teoremi basati su risoluzione consiste nel suddividere l'insieme $S$ di clausole in input in due insiemi: $Wo$ in cui tutte le possibili applicazioni della regola di risoluzione sono state eseguite tra le clausole appartenti questo insieme e $Us$ che è l'insieme di clausole che devono essere ancora considerate per partecipare all'applicazione della regola di risoluzione. All'inzio, chiaramente avremo che $Wo$ è vuoto mentre $Us$ è l'intero insieme $S$. Iterativamente, quindi, si procede all'estrazion di un clausola in $Us$, si aggiunge a $Wo$ e si eseguono tutte le applicazioni della regola di risoluzione tra le clausole di questo insieme. Quindi si aggiungono tutte le clausole ottenute a $Us$. Questa procedura puó essere riassunta in pseudocodice come in Figura 1. Osserviamo che nell'insieme `New` delle clausole risolventi vi possono essere delle tautologie, ovvero delle clausole sempre valide. Ad esempio, se si applica risoluzione sulle clausole $p \vee \neg q$ e $\neg p \vee q$, si ottiene $p \vee \neg p$ oppure $q \vee \neg q$ a seconda del letterale selezionato per la risoluzione.

La selezione della clausola `C` influisce sulle prestazioni dell'algoritmo. Consideriamo, ad esempio, questo insieme di clausole, etichettate con un identificatore numerico per comodità:

$$\{1 : p_1 \quad 2 : p_2 \quad 3 : p_3 \quad 4 : p_4 \quad 5 : p_5 \quad 6 : p_6 \quad 7 : p_7 \quad 8 : p_8 \quad 9 : p_9 \quad 10 : \neg p_1\}.$$

Ora, se il criteria di selezione della clausola segue l'ordine sugli identificatori, è chiaro che la clausola vuota sarà trovato solo alla diecima iterazione del ciclo. Se, invece, il criterio fosse tale da permettirci di selezionare la clausola 1 e subito dopo la 10, riusciremmo a derivare la clausola vuota alla seconda iterazione. Ovviamente trovare delle buone euristiche per la selezione di

```
Wo := ∅
Us := S
WHILE (Us≠ ∅ AND □ ∉ Us) DO
  Selezione ed estrazione di una clausola C da Us
  Wo  := Wo ∪ { C }
  New := clausole risolventi tra C e quelle in Ws
  Us  := Us ∪ New
END
IF (Us= ∅) THEN RETURN   soddisfacibile
IF (□ ∈Us) THEN RETURN insoddisfacibile
```

Figure 1: Meccanizzazione del metodo di risoluzione proposizionale

una clausola è un problema molto importante per migliorare, in pratica, le prestazioni dell'algoritmo.

Un altro problema consiste nel calcolo delle clausole risolventi tra quella selezionata e quelle presenti in $Ws$: se la clausola selezionata contiene piú di un letterale, allora bisogna tenere presente che uno qualsiasi di tali letterali puó essere quello selezionato per la risoluzione. Questo ulteriore grado di non determinismo aggiunge ulteriori possibilità di inefficienza se l'euristica adottata non è opportuna. Al fine di evitare alcuni di questi problemi, si sono sviluppate alcune varianti della regola di risoluzione. Qui di seguito, in preparazione al metodo DPLL, vedremo la risoluzione unitaria.

### 2.4.3 Risoluzione unitaria

Una clausola $C$ è *unitaria* quando contiene un solo letterale. La regola di *risoluzione unitaria* è un'instanza della regola di risoluzione dove una delle due premesse è una clausola unitaria, ovvero:

$$\frac{\ell \qquad \overline{\ell} \vee D}{D}$$

dove $\ell$ è un letterale e $\overline{\ell}$ è la sua negazione, ovvero se $\ell = p$ allora $\overline{\ell} = \neg p$, oppure se $\ell = \neg p$ allora $\overline{\ell} = p$, dove $p$ è una lettera proposizionale.

Se adottassimo la regola di risoluzione unitaria nell'algoritmo di Figura 1, alcune delle problematiche discusse prima si semplificherebbero notevolmente poichè, ad esempio, si dovrebbe selezionare sempre una clausola unitaria e, di conseguenza, il letterale su cui fare risoluzione sarebbe automaticamente determinato. Sfortunatamente questa regola, da sola, rende la procedura *incompleta*, ovvero potrebbe non derivare la clausola vuota quando l'insieme delle clausole iniziali è insoddisfacibile. Per rendersi conto

di questo, si consideri il seguente insieme di clausole:

$$\widehat{S} \;:=\; \{1 : p_1 \vee p_2, \quad 2 : \neg p_1 \vee p_2, \quad 3 : p_1 \vee \neg p_2, \quad 4 : \neg p_1 \vee \neg p_2\}.$$

Utilizzando la regola di risoluzione introdotta in precedenza è facile derivare la clausola vuota (si provi a farlo). Se, invece, si usa la regola di risoluzione unitaria questo non è possibile poichè tutte le clausole contengono due letterali e non ve ne è alcuna che ne contiene soltanto uno*!*

Fortunatamente è possibile rimediare al problema ricordando la definizione di verità per la disgiunzione, ovvero $A_1 \vee A_2$ è soddisfacibile se esiste un assegnamento proposizionale $v$ tale che $v \models A_1 \vee A_2$ ovvero

$$v \models A_1 \text{ oppure } v \models A_2.$$

Questo implica che, se consideriamo una clausola qualsiasi, è sufficiente che uno solo dei suoi letterali sia vero affinchè tutta la clausola sia soddisfacibile. In questo modo, si possono creare tante clausole unitarie quanti sono i letterali contenuti in una certa clausola e quindi considerare la soddisfacibilità di ciascuno di essi con l'insieme delle clausole rimanenti: se almeno uno di tali insiemi risulta soddisfacibile allora possiamo concludere la soddisfacibilità dell'insieme di partenza, altrimenti, se nessuno degli insiemi ottenuti è soddisfacibile, concludiamo l'insoddisfacibilità dell'insieme di partenza. Il punto chiave è la formazione delle clausole unitarie che permettono l'applicazione della regola di risoluzione unitaria.

Nel caso dell'insieme di clausole $\widehat{S}$, utilizzando la prima clausola, otteniamo i seguenti due insiemi di clausole:

$$\widehat{S}_1 := \{p_1\} \cup \widehat{S}_0 \quad e \quad \widehat{S}_2 := \{p_2\} \cup \widehat{S}_0,$$

dove $\widehat{S}_0 := \{2 : \neg p_1 \vee p_2, 3 : p_1 \vee \neg p_2, 4 : \neg p_1 \vee \neg p_2\}$. È facile vedere che l'applicazione della regola di risoluzione unitaria a $\widehat{S}_1$ ed a $\widehat{S}_2$, permette di derivare in entrambi i casi la clausola vuota e ci permette di concludere l'insoddisfacibilità di $\widehat{S}$.

Riassumendo, oltre alla regola di risoluzione unitaria, bisogna adottare la seguente regola di *decomposizione* (splitting), che data una clausola $C$ contenente $n$ letterali ritorna un insieme di $n$ clausole unitarie contenenti ciascuna gli $n$ letterali della clausola $C$. In generale a differenza dell'esempio sopra riportato, è possibile che dopo l'applicazione esaustiva della regola di risoluzione unitaria, si possa ancora applicare la regola di decomposizione. Dal punto di vista implementativo, è possibile utilizzare la ricorsione per realizzare il meccanismo di ricerca esaustiva tra i possibili insiemi di clausole generati dall'applicazione della regola di risoluzione unitaria e da quella di decomposizione come descritto in Figura 2. Si noti l'utilizzo dell funzione `UR` che prende in input un insieme $S$ di clausole e restituisce l'insieme di clausole ottenuto dall'applicazione esustiva della regola di risoluzione unitaria ad $S$. In altre parole, il corpo della funzione `UR` non è altro che

```
FUNCTION URSplit(S : insieme di clausole)
  S := UR(S);
  IF (□ ∈ S) THEN RETURN insoddisfacibile
  {ℓ_1, ..., ℓ_n} := selezione e decomposizione di una clausola in S
  IF n = 1 THEN RETURN soddisfacibile
  FOREACH j ∈ {1, ..., n} DO
    IF URSplit(S∪ {ℓ_j}) = soddisfacibile
    THEN RETURN soddisfacibile
  END
RETURN insoddisfacibile
```

Figure 2: Meccanizzazione del metodo di risoluzione unitaria

l'algoritmo di figura 1 specializzato all'applicazione della risoluzione unitaria (con le semplificazioni discusse prima) e dove non si eseguono i due test finali sull'insieme `Us`. Non è difficile dimostrare che anche la funzione `URSplit` termina ed è corretta ed è quindi una procedura di decisione per il problema della soddisfacibilità proposizionale. Similmente a prima, è chiaro che diversi criteri per selezionare e la clausola da decomporre possono far variare in maniera importante le prestazioni dell'algoritmo.

### 2.4.4 Confronto tra i metodi

Fino ad ora abbiamo visto quattro procedure di decisione per la risoluzione del problema di soddisfacibilità proposizionale: tavole di verità, tableaux, risoluzione e risoluzione unitaria (con decomposizione).

È noto che i metodi della tavole di verità e di risoluzione non sono adatti a risolvere problemi di soddisfacibilità proposizionale se non di piccole dimensioni. Il metodo dei tableaux e di risoluzione unitaria sono migliori dei precedenti in generale ma soffrono pure loro di qualche problema, insito nella regola di decomposizione. Infatti, sia i tableaux che il metodo di risoluzione unitaria possono comportarsi peggio delle tavole di verite per alcune classi di formule. Infatti, mentre le tavole di verità enumerano tutti i possibili assegnamenti proposizionali in maniera mutuamente esclusiva, questo puó non avvenire utilizzando la regola di decomposizione vista in precedenza.

Per rendersi conto di questo, consideriamo di volere verificare la soddisfacibilità dell'insieme di clausole $S' := S \cup \{p_1 \vee p_2\}$ e di aver provato che $S \cup \{p_1\}$ è insoddisfacibile. Per concludere sulla soddisfaicibilità di $S'$, dobbiamo verificre se $S \cup \{p_2\}$ è soddisfacibile o meno. A tale scopo, siamo liberi di assumere che $p_1$ sia assegnato a 0 o, equivalentemente, possiamo considerare la soddisfacibilità dell'insieme $S \cup \{p_2, \neg p_1\}$. Questo è corretto in quanto abbiamo mostrato che $S \cup \{p_1\}$ è insoddisfacibile e quindi ogni insieme che

lo contiene (come nel caso in cui assumessimo che $S \cup \{p_2, p_1\}$) è necessariamente insoddisfacibile. Equivalentemente, possiamo dire che tutti gli assegnamenti tali che $p_1$ viene assegnato a $\mathsf{1}$ sono insoddisfacibili quando abbiamo considerato il caso $S \cup \{p_1\}$ e quindi considerarli nuovamente quando si vuole verificare la soddisfacibilità di $S \cup \{p_2\}$ è ridondante, ovvero una perdita di tempo. È facile costruire delle formule in fnc che mettono in luce questo comportamento indesiderabile della regola di decomposizione considerata fino a questo momento. In particolare, vi sono formule per cui il metodo dei tableaux o di risoluzione unitaria considerano un numero di casi pari a $k!$ dove $k$ è il numero di variabili proposizionali nella formula mentre il metodo delle tavole di verità ne considera soltanto $2^k$.

Per evitare questo problema, vi sono due tecniche fondamentali. La prima consiste nel modificare la regola di espansione per la disgiunzione dei tableaux affinchè si eviti di riconsiderare più volte assegnamenti proposizionali che, sicuramente, danno orgine all'insoddisfacibilità. Tale tecnica è detta della generazione dei lemmi e, in combinazione con opportune strutture dati per la rappresentazione degli alberi generati dal metodo dei tableux porta all definizione dei BDD: strutture dati particolarmente compatte capaci di memorizzare tutti i possibili assegnamenti proposizionali che rendono vera una certa formula. I BDD sono utilizzati per il model checking di sistemi a stati finiti come vedremo nel Capitolo XX. La seconda tecnica, invece, consiste nell'adottare una diversa regola di decomposizione (o splitting), che ha una motivazione "semantica" rispetto a quella presentata in precedenza, che è piú legata al connettivo di disgiunzione e quindi alla sintassi. Questa regola di decomposizione semantica, combinata a tecniche di visita dinamica dello spazio di ricerca di tutti i possibili assegnamenti proposizionali ci porta alla definizione del metodo DPLL: una tecnica che è alla base di tutte le implementazioni moderne delle procedure di decisione per il problema della soddisfacibilità proposizionale e chiamati comunemente SAT solver.

## 2.5 Metodo DPLL

La nuova regola di *decomposizione* (o splitting) *semantico* si prefigge di evitare il problema di considerare più volte gli assegnamenti proposizionali che assegnano un valore di verità ad una certa lettera proposizionale e, in particolare, si basa sulla seguente (ovvia) osservazione: in un assegnamento proposizionale, una lettera proposizionale $p$ può assumere il valore di verità $\mathsf{1}$ oppure quello $\mathsf{0}$. Dato un insieme di clausole $S$ ed una lettera proposizionale $p$ che occorre in $S$, in qualsiasi momento siamo liberi di considerare tutti gli assegnamenti proposizionali che assegnano $p$ a $\mathsf{1}$ e tutti quegli assegnamenti proposizionali che assegnano $p$ a $\mathsf{0}$. Se siamo in grado di stabilire che non esistono assegnamenti proposizionali con $p$ assegnato a $\mathsf{1}$ che rendono vero l'insieme di clausole $S$ e, similmente, che non esistono assegnamenti proposizionali con $p$ assegnato a $\mathsf{0}$ che rendono vero l'insieme di clausole

$S$, allora possiamo concludere che non esistono assegnamenti proposizionali, che assegnano $p$ a `1` oppure `0` (non importa), che rendono vero l'insieme di clausole $S$.

Ora, è possibile forzare gli assegnamenti proposizionali ad assegnare il valore di verità `1`, semplicemente aggiungendo all'insieme di clausole $S$, la clausola unitaria $p$ (ovvero considerando l'insieme di clausole $S \cup \{p\}$), e, dualmente, assegnare il valore di verità a `0`, assigungendo all'insieme di clausole $S$, la clausola unitaria $\neg p$ (ovvero considerando l'insieme di clausole $S \cup \{\neg p\}$). In altre parole, il precedente schema di ragionamento che ci permette di considerare separatamente gli assegnamenti proposizionali che assegnano $p$ a `1` e $p$ a `0`, è riassumibile nella regola di *decomposizione semantica*: dato un insieme di clasuole $S$ ed una lettera proposizionale $p$ che occorre in $S$, si consideri la soddisfacibilità dell'insieme di clausole $S \cup \{p\}$ e dell'insieme $S \cup \{\neg p\}$. Se, uno dei due insiemi risulta soddisfacibile, allora possiamo concludere che anche l'insieme $S$ iniziale è soddisfacibile. Se, invece, entrambi gli insiemi risultano insoddisfacibili, allora possiamo concludere che l'insieme $S$ è insoddisfacibile. Chiaramente, al fine di stabilire la soddisfacibilità di $S \cup \{p\}$ e $S \cup \{\neg p\}$, possiamo utilizzare la regola di risoluzione unitaria (esattamente come fatto in precedenza con la regola di decomposizione sintaticca) ed eventualmente riapplicando la regola di decomposizione semantica se quella di risoluzione unitaria è impossibile da applicare.

Non dovrebbe essere difficile convincersi che l'applicazione esaustiva delle due regole di risoluzione unitaria e di decomposizione semantica fornisce una procedura di decisione per il problema della soddisfacibilità proposizionale: la correttezza deriva dall'osservazione all'inizio di questa sezione e che fornisce l'intuizione alla base di questa regole mentre la terminazione è ovvia in quanto vi è un numero finito di lettere proposizionali che occorrono in $S$ e sulle quali è possibile applicare la regola di decomposizione semantica.

È possibile implementare l'applicazione esaustiva della regola di risoluzione unitaria e di decomposizione semantica in maniera molto simile a quanto fatto in precedenza con la funzione `URSplit`, ottenendo in questo modo una versione ricorsiva dell'algoritmo di Davis-Putnam-Loveland-Logeman (DPLL): si veda la figura 3 (dove `UR` è la stessa funzione utilizzata in precedenza per `URSplit`). Una possibile interpretazione del funzionamento della funzione `DPLL-rec` è, come abbiamo detto in precedenza, l'applicazione esaustiva delle regole di risoluzione unitaria e di decomposizione semantica. È istruttivo comunque considerare un'altra interpretazione (semantica) della funzione come esplorazione dello spazio degli assegnamenti proposizionali possibili all'insieme di clausole di input. Lo spazio dei potenziali assegnamenti proposizionali è organizzato ad albero ed è costruito durante l'esplorazione come segue. Anzitutto un ordine arbitrario viene assegnato alle lettere proposizionali (ed è insito nel criterio di selezione su cui applicare la decomposizione semantica). La radice dell'albero è etichettata dalla

```
FUNCTION DPLL-rec(S : insieme di clausole)
  S := UR(S);
  IF (□ ∈ S) THEN RETURN insoddisfacibile
  p := selezione di una lettera proposizionale in S
  IF (non vi sono lettere selezionabili) THEN RETURN soddisfacibile
  IF (DPLL-rec(S∪{p})=insoddisfacibile) THEN RETURN DPLL-rec(S∪{¬p})
  ELSE RETURN soddisfacibile
```

Figure 3: Meccanizzazione del metodo DPLL (ricorsivo)

lettera proposizionale secondo l'ordine precedentemente stabilito. In generale, un nodo $n_i$ a profondità $i$ nell'albero è etichettato dall'$i$-esima lettera proposizionale $p_i$ secondo l'ordine stabilito. Ogni nodo $n$ ha due sottoalberi, identificati con $c_n^{\mathtt{1}}$ e $c_n^{\mathtt{0}}$. L'arco tra il nodo $n_i$ a profondità $i$ etichettato dalla lettera proposizionale $p_i$ e la radice del sottoalbero $c_{n_i}^{\mathtt{1}}$ rappresenta l'assegnamento proposizionale che assegna $p_i$ a 1; similmente, l'arco tra $n_i$ e la radice del sottoalbero $c_{n_i}^{\mathtt{0}}$ rappresenta l'assegnamento proposizionale che assegna $p_i$ a 0. In questo modo, ogni percoso nell'albero dalla radice ad una sua foglia rappresenta un assegnamento proposizionale "totale."[1] Pertanto, l'abero definito sopra rappresenta lo spazio di tutti i possibili assegnamenti di un insieme di clausole. La funzione `DPLL-rec` esegue una ricerca esaustiva (grazie al meccanismo di ricorsione) in tale albero di quell'assegnamento proposizionale totale che soddisfa l'insieme delle clausole di input estendendo incrementalmente assegnamenti parziali mano a mano che esplora in maggiore profondità l'albero. Si noti inoltre come la funzione `UR` svolga un ruolo di semplificazione dell'insieme di clausole ogni volta che si è deciso che una lettera proposizionale deve assumere un certo valore di verità (secondo la regola di decomposizione semantica). Pertanto, anche per questo metodo, le euristiche per la scelta della lettera proposizionale su cui applicare la regola di decomposizione semantica possono dar luogo a variazioni importanti nelle prestazioni.

Il metodo DPLL è riconosciuto dare origine alle migliori implementazioni della procedura di decisione per il problema di soddisfacibilità proposizionale. Tali implementazioni sono conosciute con il nome di *SAT solver* ed hanno conosciuto un grande sviluppo negli ultimi anni. Il loro successo è testimoniato dalla loro adozione in ambito industriale principalmente per la verifica di circuiti sequenziali da colossi come l'Intel. Nel seguito, consideri-

[1] Ovvero un assegnamento proposizionale che assegna ad ogni lettera proposizionale che occorre nell'insieme di clausole in input un valore di verità. A volte diremo che un assegnamento proposizionale è parziale quando questo assegna valori di verità ad un sottoinsieme stretto dell'insieme di lettere proposizionali occorrenti in un certo insieme di clausole.

amo brevemente alcuni delle tecniche implementative principali che hanno determinato il successo dei SAT solver.

### 2.5.1 SAT solver

La modifica più rispetto alla funzione `DPLL-rec` è l'assenza della ricorsione che gestice implicitamente uno stack di assegnamenti proposizionali e che implicitamente attraversa l'albero di ricerca. Nelle implementazioni moderne, lo stack è mantenuto esplicitamente e, in generale, si evita di visitare in maniera esaustiva l'albero di ricerca grazie a tecniche dette di *learning*. Ma vediamo brevemente le più importanti fra queste tecniche:

- euristiche per stabilire l'ordine con cui vengono considerate le lettere proposizionali. Siccome è possibile mostrare l'insoddisfacibilità di molti insiemi di clausole considerando soltanto un sottoinsieme delle lettere proposizionali che vi occorrono (ovvero considerando quegli assegnamenti parziali che assegnano un valore di verità a tali lettere), sarebbe vantaggioso considerare queste lettere come priorietarie nell'ordinamento. Sfortunatamente, è possibile dimostrare che identificare tali insiemi di lettere proposizionali è altrettanto difficile che mostrare la soddisfacibilità dell'insieme di clausole in cui occorrono. Pertanto, il meglio cui possiamo aspirare è trovare delle euristiche che suggeriscano ordinamenti di lettere proposizionali sub-ottimali. Una classe comune nei SAT solver moderni di queste euristiche è detta MOM (Maximum Occurrences in Minimum length clauses), la cui intuizione è quella di scegliere la lettera proposizionale su cui fare la decomposizione sulla lettera più "vincolata" e di misurare il "grado di vincolo" delle lettere proposizionali in base al numero di occorrenze di queste in clasuole contenenti pochi letterali: l'idea è che più una lettera proposizionale è vincolata, più ampio sarà l'effetto di semplificazione dovuto al suo assegnamento ad un valore di verità;

- ottimizzazione delle strutture dati per la rappresetazione dell'insieme di clausole per eseguire la loro semplificazione (ovvero l'implementazione efficace di `UR`). Se una lettera proposizionale $p$ è stata assegnata a `1`, la semplificazione di cui si parla qui consiste nel marcare come *risolte* le clausole che contengono $p$ come letterale e nel riscrivere una clausola della forma $\neg p \vee D$ a $D$; dualmente, quando $p$ è assegnata a `0`. Inoltre, se tutte le clausole dell'insieme di input sono marcate come risolte, allora si conclude la soddisfacibilità dell'insieme. È stato osservato che questo processo di semplificazione occupa la maggior parte del tempo di esecuzione di un moderno SAT solver. Vi sono due colli di bottiglia in questa attività. Il primo è relativo alla semplificazione delle clausole che può richiedere la scansione delle lettere proposizionali in ogni

clausola contenente la negazione dell'ultima lettera proposizionale assegnata. Il motivo di questa scansione è di determinare se la clausola è diventata unitaria oppure vuota. Il secondo collo di bottiglia è relativo al backtracking durante il quale lo stato di ogni clausola deve essere ricomputato. Per risolvere questi problemi è stato proposto il metodo detto del *two literal watching* che consiste nel tenere traccia (watch) di due letterali in una clausola in modo tale che le lettere proposizionali in essei non vengano assegnate il più a lungo possibile. In questo modo, si ottiene che una clausola è unitaria quando uno solo dei due letterali in questione non è assegnato mentre è vuota quando entrambi sono assegnati: lo status della clausola è valutabile in tempo costante. Di conseguenza, il backtracking può essere effettuato senza alcuna rivalutazione della clausola e la semplificazione delle clausole difficilmente richiede una scansione delle stesse.

- le già citate tecniche di learning. Quando la funzione `DPLL-rec` stabilisce che un assegnamento non rende vero l'insieme delle clausole, essa esegue un backtracking detto *cronologico* ovvero considera l'assegnamento proposizionale che differisce dal precedente nell'assegnare in maniera diversa l'ultima (rispetto all'ordinamento considerato) lettera proposizionale. Con il learning, invece, l'algoritmo DPLL prima determina un sottoinsieme delle lettere proposizionali nel dominio dell'assegnamento tale da rendere insoddisfacibile l'insieme di clausole. Quando si tratta di fare il backtracking, l'algoritmo utilizza questa informazione per evitare di considerare quegli assegnamenti proposizionali che assegnano le lettere proposizionali a quei valori che, sicuramente, conducono all'insoddisfacibilità. In questo modo, l'algoritmo impara (da cui il nome di learning) quali sono quelle regioni dello spazio di ricerca che deve evitare perchè sicuramente conducono a considerare assegnamenti proposizionali che non rendono vero l'insieme delle clausole.

Per riassumere la situazione, in figura 4, si riporta lo pseudo-codice relativo ad un implementazione iterativa dell'algoritmo di DPLL che è alla base dei moderni SAT solver. Anzitutto, notiamo che questa volta l'assegnamento proposizionale è un parametro della procedura (a differenza di prima che era implicitamente rappresentato dallo stack delle chiamate ricorsive). Inoltre, si noti che l'assegnamento proposizionale `v` è inizialmente vuoto ed è organizzato come uno stack. (Si assume che per `DPLL` come per le altre funzioni e procedure utilizzate in seguito, i parametri vengano passati per riferimento.) La procedura `decide-next-branch(S,v)` seleziona una lettera proposizionale che occorre in `S` e non è ancora assegnata in `v` secondo una certa euristica (ad esempio MOM, si veda sopra). Questa operazione viene chiamata *decisione* ed il numero di lettere proposizionali assegnati in `v` dopo questa operazione è detto *livello di decisione*: all'inzio dell'esecuzione di `DPLL`, il livello di decisione è quindi 0.

```
FUNCTION DPLL(S : insieme di clausole, v : assegnamento proposizionale)
  WHILE (TRUE) DO
    decide-next-branch(S,v);
    WHILE (TRUE) DO
      status = deduce(S,v);
      IF (status = soddisfacibile) THEN RETURN soddisfacibile
      ELSE IF (status = conflitto) THEN
              blevel = analyze-conflict(S,v)
              IF (blevel = 0) THEN RETURN insoddisfacibile
              ELSE backtrack(blevel,S,v)
      ELSE BREAK
    END
  END
```

Figure 4: Meccanizzazione del metodo DPLL (iterativo)

La funzione `deduce(S,v)` ha il compito di implementare in maniera efficace la funzione `UR` utilizzata per `DPLL-rec`, magari utilizzando il metodo del two watched literal discusso sopra. Essa ritorna (1) `soddisfacibile` quando l'assegnamento `v` soddisfa `S`, (2) `conflitto` quando `v` non soddisfa `S` e (3) `insoddisfacibile` quando non si possono fare altre semplificazioni di `S` (in letteratura, si dice anche the `deduce` implementa la Boolean Constraint Propagation o BCP).

Nel primo caso (ovvero quando `deduce` ritorna `soddisfacibile`), pure `DPLL` ritorna `soddisfacibile`. Nel secondo caso (ovvero quando `deduce` ritorna `conflitto`), si richiama la procedura `analyze-conflict(S,v)` che implementa la tecnica di learning, ovvero determina un sotto-assegnamento parziale `w` di `v` sufficiente a determinare l'insoddisfacibilità di `S` e calcola il livello di decisione `blevel` cui fare backtracking: per fare questo è sufficiente andare a determinare il livello di decisione minore in `v` delle lettere proposizionali nel dominio di `w`. Se `blevel` è 0, allora il conflitto esiste senza prendere alcuna decisione e quindi `DPLL` ritorna `insoddisfacibile`. Altrimenti, `backtrack(blevel,S,v)` aggiunge la "negazione" del contenuto del sotto-assegnamento parziale calcolato precedentemente da `analyze-conflict` ad `S` (fase di learning) e quindi esegue un backtracking al livello di decision `blevel` (fase di backjumping) aggiornando opportunamente sia `S` che `v`: si noti quindi che il backtracking non è più cronologico come nel caso della versione ricorsiva del metodo DPLL e permette quindi di evitare di esplorare in maniera esaustiva lo spazio di tutti i possibili assegnamenti. Nel terzo caso (ovvero quando `deduce` ritorna `insoddisfacibile`), `DPLL` esce dal ciclo più interno (`break`) e considera una nuova lettera proposizionale per la decomposizione.

Ma vediamo più precisamente cosa significa "aggiunge la "negazione" del contenuto del sotto-assegnamento `w` a `S`," ovvero come si implementa la fase di learning. Anzitutto, si osservi che un assegnamento proposizionale può essere espresso come una fnc in cui tutte le clausole sono unitarie: se $p$ è assegnata a `1`, allora si considera la clausola unitaria $p$, altrimenti, se $p$ è assegnata a `0`, allora si considera la clausola unitaria $\neg p$. Sia $w$ la fnc corrispondente all'assegnamento `w` ottenuta come spiegato. A questo punto, "negare `w`" è equivalente a negare la formula $w$. Siccome $w$ è la congiunzione di clausole unitarie, $\neg w$ non sarà altro che una clausola: se $\ell_1 \wedge \cdots \wedge \ell_n$ è $w$, allora $\neg w$ è $\neg(\ell_1 \wedge \cdots \wedge \ell_n)$ che è equivalente a $\overline{\ell_1} \wedge \cdots \wedge \overline{\ell_n}$, dopo alcune semplici manipolazioni logiche, dove $\overline{\ell}$ è la negazione del letterale $\ell$, ovvero se $\ell = p$ allora $\overline{\ell} = \neg p$, oppure se $\ell = \neg p$ allora $\overline{\ell} = p$ e $p$ è una lettera proposizionale. A questo punto, "aggiungere la negazione di `w`" consiste semplicemente nell'agiungere la clausola corrispondente a $\neg w$. Si noti come $\neg w$ sia una conseguenza logica dell'insieme `S`[2] e pertanto la sua aggiunta a `S` non cambia l'insieme degli assegnamenti proposizionali per `S`.

Resta ancora da descrivere come si possa determinare la causa del conflitto, ovvero, il sotto-assegnamento `w` che permette il learning e quindi il backjumping. L'analisi di un conflitto implementata nella procedura `analyze-conflit` comincia con la costruzione di un *grafo di implicazione*. Tale grafo è costruito ripercorrendo in senso inverso le semplificazioni che sono state fatte da `deduce` a partire dalla clausola vuota. I vertici del grafo sono etichettati con gli assegnamenti delle lettere proposizionali come clausole unitarie o, equivalentemente letterali similmente a quanto visto in precedenza. Ogni volta che un letterale è stato dedotto in quanto unico letterale non ancora assegnato in una certa clausola, un arco viene aggiunto al grafo a partire dalla negazione degli altri letterali nella clausola al fine di dedurre il letterale. Ad esempio, considerando l'assegnamento $v(p_1) = \mathtt{0}$ e $v(p_2) = \mathtt{1}$, la clausola $p_1 \vee \neg p_2 \vee p_3$ produrrebbe il grafo di implicazione seguente: i vertici sono etichettati da $\neg p_1, p_2, p_3$ e gli archi sono $\langle \neg p_1, p_3 \rangle$ e $\langle p_2, p_3 \rangle$. Gli archi rappresentanti le implicazioni vengono quindi percorsi in senso inverso alla loro orientazione dai letterali contenuti nella clausola vuota fino ad arrivare ai letterali di decisione. Siccome i letterali di decisione non sono stati dedotti, essi non possono essere destinazione di alcun arco ma solo sorgenti.

A questo punto, l'analisi del conflitto passa alla fase successiva di calcolo di un *taglio* (cut) nel grafo di implicazione che è un insieme di archi che separa le negazioni dei letterali di decisione dal punto di conflitto. Ad esempio, si può prendere come taglio l'insieme di tutti gli archi nel grafo di implicazione che hanno come sorgente dei nodi etichettati con la negazione di letterali di decisione. Vi sono molti altri possibili modi per determinare i

[2]Siccome `w` non soddisfa `S`, possiamo dire che $\mathtt{S} \wedge w$ è insoddisfacibile, ovvero, equivalentemente $\mathtt{S} \wedge w \models \square$ che, per refutazione è equivalente a $\mathtt{S} \models \neg w$, ovvero $\neg w$ è conseguenza logica di `S`.

tagli di un grafo di implicazione che hanno diverso impatto sulle prestazione di `DPLL`. Una volta che si è determinato il taglio, è sufficiente considerare le etichette di tutti i nodi sorgente come sotto-assegnamento proposizionale causa dell'insoddisfacibilità.

Concludiamo con un esempio di esecuzione della funzione `DPLL` questo capitolo al fine di illustrare più concretamente le varie tecniche discusse sopra.

[[[TO DO!]]]

### 2.5.2 Formato di input dei SAT solver

L'enorme sviluppo ed il successo dei moderni SAT solver si deve anche ad uno sforzo della comunità degli sviluppatori di tali strumenti per rendere disponibile un formato di input comune che facilità lo scambio dei vari problemi di soddisfacibilità ed il confronto delle prestazioni dei vari solver. Tale formato si chiama formato DIMACS e permette di rappresentare problemi di soddisfacibilità per formule in fnc come file in formato ASCII. Precisamente, un file in formato DIMACS è suddiviso in due parti: il *preambolo* e le *clausole*.

Il preambolo contiene informazioni sull'istanza del problema di soddisfacibilità proposizionale contenuto nel file. Tali informazioni sono suddivise in linee del file ed il primo carattere di ogni linea specifica il tipo di informazione in essa contenuto:

- commenti: le linee di commento contengono informazioni per spiegare il cotenuto del file ad uso degli esseri umani e vengono ignorate dai solver. Le linee di commento devono apparire all'inizio del preambolo ed ogni linea deve iniziare con il carattere minuscolo `c`;

- linea del problema: vi è una sola linea di questo tipo in ogni file in formato DIMCAS. La linea del problema deve avere la seguente struttura:

  `p` *FORMATO VARIABILI CLAUSOLE*

  dove il carattere minuscolo `p` identifica univocamente il fatto che si tratta di una linea di problema. Il campo *FORMATO* permette ai solver di determinare il formato in cui è espresso il problema: nel nostro caso, il contenuto di tale campo sarà sempre `cnf` (conjunctive normal form, ovvero forma normale congiuntiva in inglese) poichè i problemi che noi considereremo saranno sempre espressi come formule in fnc. Il campo *VARIABILI* contiene un intero positivo che specifica il numero di lettere proposizionali (o, equivalentemente, variabili Booleane) che sono contenute nel problema di soddisfacibilità contenuto nel file. Infine, il campo *CLAUSOLE* contiene un intero positivo che specifica il numero di clausole nel problema di soddisfacibilità contenuto nel file. La linea di problema deve essere l'ultima del preambolo.

Le clausole appaiono immediatamente dopo la linea del problema. Le lettere proposizionali (o variabili booleane) sono identificate dai numeri interi da 1 al numero specificato nel campo *VARIABILI* nella linea del problema. Non è necessario che ogni lettera appaia nel problema di soddisfacibilità contenuto nel file. Ogni clausola viene ad essere rappresentata da una sequenza di interi, ogni intero separato da uno spazio, un carattere tab, oppure un carattere di a a capo. Un letterale positivo (ovvero una lettera proposizionale non negata) cui corrisponde l'intero $i$ viene rappresentato da $i$ mentre un letterale negativo (ovvero una lettera proposizionale negata) cui corrisponde l'intero $i$ viene rappresentato da $-i$. Ogni clausola è terminata dall'intero 0.

Consideriamo come esprimere il seguente problema di soddisfacibilità proposizionale in formato DIMCAS: la seguente fnc

$$(p_1 \vee p_3 \vee p_4) \wedge (p_4) \wedge (p_2 \vee \neg p_3)$$

è soddisfacibile*?* Un possibile file in formato DIMCAS corrispondente a tale problema è il seguente:

```
c Esempio di problema in formato fnc
c
p cnf 4 3
1 3 -4 0
4 0
2 -3 0
```

dove si è deciso di rappresentare $p_i$ con $i$, vi sono 4 lettere proposizionali ($p_1, p_2, p_3, p_4$ rappresentate rispettivamente da $1, 2, 3, 4$) e 3 clausole (($p_1 \vee p_3 \vee p_4$), ($p_4$) e ($p_2 \vee \neg p_3$) rappresentate da `1 3 -4 0`, `4 0` e `2 -3 0`, rispettivamente). Notare come questa informazione sia condificata nella linea di problema `p cnf 4 3`.

## 2.6 Dai Tableaux ai BDD

Come abbiamo già visto precedentemente, la regola di decomposizione (splitting) può dar luogo a riconsiderare più volte l'assegnamento di una lettera proposizionale ad un certo valore di verità che implica l'insoddisfacibilità di un insieme di clausole. Un fenomeno simile, questa volta per le sottoformule della formula di cui si vuole verificare la soddisfacibilità, si ha nei tableaux con la regola di espansione della disgiunzione. È possibile modificare questa regola di espansione includendo quelli che sono chiamati *lemmi* che hanno il compito di impedire di riconsiderare lo stesso valore di verità per una sottoformula, valore di verità che sappiamo portare ad una refutazione.

Vi sono due modi per giustificare i lemmi: il primo utilizza le tavole di verità mentre il secondo è di natura più operazionale. Il primo, che

generalizza a sottoformule quanto detto in precedenza per i letterali quando si considerava la regola di decomposizione, può essere illustrato considerando le tavola di verità per la formula $B \vee C$ e per $B \vee (\neg \wedge C)$, rispettivamente a sinistra ed a destra qui di seguito:

| $B$ | $C$ | $B \vee C$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $B$ | $C$ | $B \vee (\neg B \wedge C)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Analizzando la tavola di verità, si può notare come le due formule siano logicamente equivalenti, pertanto la seconda si può sostituire alla prima senza che questo comporti un cambiamento dell'insieme degli assegnamenti proposizionali che soddisfano un insieme di formule in cui la prima occorre. Possiamo quindi immaginare che, ogni volta che troviamo una formula del tipo $B \vee C$, la si sostituisca con $B \vee (\neg B \wedge C)$ e quindi si applichi le due regole di espansione dei tableaux all'insieme di formule risultanti. Ma vediamo qual è l'effetto di applicare le regole di espansione dei tableaux alla formula $B \vee (\neg B \wedge C)$:

$$\frac{B \vee (\neg B \wedge C)}{B \quad || \quad \dfrac{\neg B \wedge C}{\neg B, C}}$$

Come si può vedere, il risultato di applicare, nell'ordine, la regola di espansione della disgiunzione e quella della congiunzione ci porta a considerare due casi: il primo in qui si cerca un assegnamento proposizionale che renda vera $B$ ed il secondo in cui si cerca un assegnamento proposizionale che renda vere sia $\neg B$ che $C$. Notiamo come l'assegnamento proposizionale che rende vera $B$ nel primo caso, sicurmente non renderà vere le due formule che vengono ad essere considerate nel secondo caso, ovvero $\neg B$ e $C$: l'informazione presente nei due casi considerati è più precisa di quella considerata precedentemente applicando la regola di espansione della disgiunzione sulla formula $B \vee C$. Questo ci permette di proporre la seguente *regola di espansione della disgiunzione con lemmi* che riassume in maniera pi'čompatta quanto detto sopra:

$$\frac{\Gamma, B \vee (\neg B \wedge C)}{\Gamma, B \quad || \quad \Gamma, \neg B, C}$$

Veniamo ora alla visione operazionale di questa nuova regola di espansione della disgiunzione. Il lemma di cui si parla è $\neg B$ nel ramo destro dell'albero: tale lemma viene generato dalla refutazione di $\Gamma, B$ nel ramo sinistro dell'albero e può essere visto come l'abbreviazione della refutazione

di $B$, che potrebbe essere riscoperta ripetutamente anche nel ramo destro (si pensi al caso in cui $\Gamma$ contenga $B$). Siccome tali refutazioni possono essere arbitrariamente complesse, i lemmi possono ridurre la lunghezza delle refutazioni in maniera considerevole.

L'utilizzo dei lemmi all'interno dei tableaux sembra essere molto semplice; sembra infatti sufficiente adottare la nuova regola di espansione della disgiunzione. In realà l'adozione di tale regola implica due problemi. Il primo consiste nel fatto che la rappresentazione ad albero del tableaux non è molto buona poichè duplica le occorrenze delle sottoformule che diventano lemmi: dovremmo rappresentare un sottoformula ed il relativo lemma separatamente. Il secondo e, forse, più importante problema consiste nel fatto che non possiamo più limitarci a considerare formule in fnn: la negazione di una sottoformula in fnn, in generale, non è a sua volta in fnn. Per risolvere questi problemi è opportuno adottare una rappresentazione del tableaux diversa da quella di albero. Precisamente, introduciamo la nozione di *espressione condizionale* detta anche espressione *ite* dall'inglese *if-then-else*, come segue. L'insieme ITE delle espressioni condizionali è il più piccolo insieme tale che

- $\bot$ e $\top$ appartengono a ITE,
- se $p$ è una lettera proposizionale, $B_\bot$ e $B_\top$ sono in ITE, allora l'espressione $ite(p, B_\bot, B_\top)$ è in ITE.

La semantica di $\bot$ e $\top$ è definita come segue: per ogni assegnamento proposizionale $v$ sia che $v(\bot) = \mathtt{0}$ e $v(\top) = \mathtt{1}$. Inoltre, la semantica di $ite(p, B_\bot, B_\top)$ è definita come quella della formula seguente: $(p \wedge B_\bot) \vee (\neg p \wedge B_\top)$. La rappresentazione grafica delle espressioni ITE coincide con i *Binary Decision Diagrams (BDD)*. I BDD non sono altro che degli alberi binari i cui nodi sono etichettati da lettere proposizionali (se non sono foglie) oppure dalle espressioni $\bot$ e $\top$ (se sono foglie). In particolare, le espressioni ITE possono essere trasformate in BDD ricorsivamente come segue: l'espressione $ite(p, B_\bot, B_\top)$ viene rappresentata dall'albero binario che ha come radice un nodo etichettato dalla lettera $p$ e che ha come figlio sinistro l'albero binario corrispondente all'espressione $B_\bot$ e come figlio destro l'albero binario corrispondente all'espressione $B_\top$. La base della ricorsione si ha chiaramente con $\bot$ e $\top$ che vengono rappresentate come le foglie dell'albero.

La motivazione per utilizzare le espressioni ITE o, equivalentemente, i BDD è quella di una migliore gestione dei lemmi rispetto all'adozione della regola di espansione della disgiunzione con i lemmi prima mostrata: infatti è possibile verificare che i percorsi all'interno di un BDD dalla sua radice ad una foglia qualsiasi etichettata da $\top$ corrispondono alla rappresentazione dei rami di un tableaux in cui si è utilizzata la regola di espansione della disgiunzione con i lemmi. Per comprendere questo punto bisogna considerare

il significato delle foglie terminali in un tableaux (senza lemmi) ed i percorsi (dalla radice ad una foglia etichettata co 1) in un BDD.

- Foglie terminali in un tableaux. Si assuma di avere un tableaux per una certa formula $F$ in fnn in cui le foglie siano tutte chiuse o terminali (stiamo considerando la regola di espansione per la disgiunzione senza lemmi). Come abbiamo visto in precedenza, possiamo interpretare le foglie terminali come congiunzioni di letterali corrispondenti agli assegnamenti proposizionali che soddisfano $F$. Se consideriamo la disgiunzione di tutte le foglie terminali (considerate come congiunzioni di letterali), otteniamo l'insieme di tutti gli assegnamenti proposizionali che soddisfano $F$. Osserviamo ora che la disgiunzione di tutte le foglie terminali è in fnd e, siccome, questa ha gli stessi assegnamenti proposizionali che soddisfano $F$, possiamo considerare questa come la fnd di $F$. In altre parole, abbiamo appena visto come il metodo dei tableaux non fa altro che trasformare una formula in fnn in una formula in fnd.

- Percorsi in un BDD. Un percorso può essere visto come una sequenza di letterali: consideriamo l'espressione $ite(p, B_\bot, B_\top)$. Se dal nodo etichettato con $p$ viene presa la parte relativa al *then*, allora il letterale è positivo ovvero si aggiunge $p$ al percorso che si sta costruendo; altrimenti, se viene presa la parte relativa all'*else*, il lettrale è negativo ovvero si aggiunge $\neg p$ al percorso. Analogamente a quanto fatto in precedenza per i tableaux, si possono considerare questi percorsi come congiunzioni di letterali. La differenza con i tableaux è che vi sono due tipi di percorsi: quelli che conducono ad una foglia etichettata con $\top$ e quelli che conducono ad una etichettata con $\bot$. I percorsi che conducono alla foglia etichettata con $\top$ giocano lo stesso ruolo delle foglie terminali nei tableaux ovvero costituiscono i disgiunti della corrispondente fnd della formula di partenza. I percorsi che conducono a foglie etichettate con $\bot$, invece, costituiscono dualmente la fnd della negazione della formula iniziale.

  Riassumendo, un tableaux per una formula $F$ rappresenta gli assegnamenti proposizionali che soddisfano $F$, mentre un BDD per $F$ rappresenta sia gli assegnamenti proposizionali che soddisfa $F$ sia quelli che soddisfano $\neg F$.

Rimane ora da verificare che i BDD offrono una rappresentazione compatta dei tableaux con i lemmi. Illustriamo l'idea sulla seguente formula: $p_1 \vee (p_2 \wedge p_3)$. Nel tableaux con lemmi si hanno due foglie terminali: la prima etichettata con $p_1$ mentre la seconda è etichettata con $\neg p_1, p_2, p_3$. Ora consideriamo l'espressione ite corrispondente alla formula di cui sopra: $ite(p_1, ite(p_2, \bot, ite(p_3, \bot, top)), \top)$. Vi sono due percorsi alle due foglie etichettate con $\top$: $p_1$, che corrisponde alla prima foglia terminale del tableaux con lemmi, e $\neg p_1, p_2, p_3$, che corrisponde alla seconda foglia terminale del

tableaux con lemmi. Non è difficile (anche se la prova è tecnicamente un po' lunga) generalizzare questa osservazione per formule proposizionali arbitrarie. Pertanto concludiamo che i percorsi dalla radice alla foglia etichettata con $\top$ in un BDD corrispondono alle foglie terminali di un tableaux.

Esiste una particolare sottoclasse di BDD di particolare interesse: i Reduced Ordered BDD (ROBDD). I BDD appartenenti a tale classe sono costituiti da una particolare struttura dati a grafo orientato aciclico tale che:

1. ogni percorso rispetta un dato ordine sulle lettere proposizionali (ordered),

2. nessun percorso contiene delle occorrenze multiple dello stesso letterale (reduced),

3. nessun sottografo occorre più di una volta in un BDD (maximal sharing).

Queste proprietà sono tali da garantire che i ROBDD formao una forma normale *unica* (in constrasto, sia la forma normale negativa, che quella congiuntiva o disgiuntiva non sono uniche) per le formule proposizionali. In altre parole, se una data formula è logicamente equivalente al vero oppure è insoddisfacibile, la sua rappresentazione come ROBDD sarà $\top$ e $\bot$, rispettivamente. Al fine di garantire le tre proprietà dei ROBDD che permettono di considerarli cone una forma normale unica per la logica proposizionale, alcune tecniche implementative ormani divenute standard vengono utilizzate. Ad esempio, per garantire il maximal sharing, viene utilizzata un struttura dati di tipo *cache*, che ha il compito di memorizzare i puntatori ai grafi che rappresentano certe formule: prima di computare il grafo per una formula, si andrà a guardare nella cache per vedere se è già stata computata e si utilizzerà un puntatore alla struttura già esistente per garantire che nessun sottografo occorra più di una volta nel ROBDD. Queste tecniche esulano dal contenuto di questo libro e non verranno qui descritte.

Ritorniamo ai vari modi in cui si posono utilizzare i ROBDD, sfruttando la loro proprietà di essere una forma normale unica per le formule proposizionali. Se due formule $A$ e $B$ sono logicamente equivalenti, la loro rappresentazione come ROBDD sarà identica: in questo modo, implementando tramite puntatori la struttura dati a grafo sottostante ai ROBDD, il test di equivalenza logica tra due formule può essere fatto in tempo costante. Questo suggerisce la seguente procedura di decisione per il problema di soddisfacibilità della logica proposizionale: data una formula $A$, si trovi il suo BDD, se questo è $\bot$ allora $A$ è insoddisfacibile, altrimenti (ovvero se è diverso da $\bot$) $A$ è sodisfacibile. In pratica, questa procedura di decisione per la soddisfacibilità proposizionale basata sui BDD non viene utilizzata e si preferisce quella basata su DPLL. Il motivo è semplice: come abbiamo visto, i BDD rappresentano in memoria tutti gli assegnamenti proposizionali

che soddisfano una formula (e pure quelli che ne soddisfano la negazione): per fare questo, una grande quantità di memoria, nel caso peggiore, esponenziale nel numero di lettere proposizionali che occorrono nella formula deve essere allocata per il BDD. Per una data formula proposizionale, la quantità di memoria allocata varia a seconda dell'ordinamento scelto sulle lettere proposizionali: sfortunamente determinare l'ordinamento migliore ha la stessa complessità computazionale che risolvere il problema di soddisfacibilità proposizionale.

La procedura di decisione per la soddisfacibilità proposizionale basata sui BDD, quindi, soffre del problema di esplosione della memoria. Al contrario, siccome l'algoritmo DPLL non rappresenta mai lo spazio dei possibili assegnamenti proposizionali in memoria ma lo esplora dinamicamente ha la possibilità di gestire formule di dimensione molto più elevata. L'utilizzo dei BDD è particolarmente indicato, invece, quando si deve rappresentare l'insieme degli stati di un certo sistema, ad esempio un circuito hardware sequenziale, e si vuole determinare l'equivalenza funzionale con un altro. Ad esempio, si vuole verificare che un circuito ottimizzato per l'addizione di due registri a 64 bit sia equivalente al circuito standard (ovvero non ottimizzato) che esegue la stessa operazione. Per fare questo, si rappresentano i due circuiti come due formule proposizionali e quindi si costruiscono i due ROBDD corrispondenti: se risultano essere identici, allora anche l'insieme dei loro assegnamenti proposizionali è lo stesso e si può concludere che i due circuiti sono equivalenti.

[[[TO DO: ESEMPIO]]]

# 3 Riferimenti bibliografici

[[[TO DO]]]