Capitolo 3

Logica proposizionale

3.1 Generalità sulla logica

La logica si occupa principalmente dei concetti di *verità* e *dimostrabilità*, che nel corso dei secoli sono stati studiati ampiamente sia dal punto di vista matematico che da quello filosofico. La generalità e l'importanza di questi concetti fa sì che la logica rivesta un ruolo centrale anche nell'informatica, ed in particolare nella progettazione dei programmi.

Per poter parlare di verità, è innanzitutto necessario disporre di un *linguaggio* per esprimere dei concetti o delle frasi. In logica vengono utilizzati linguaggi formali, in cui le frasi, o *formule*, vengono formate con regole *sintattiche* del tutto simili a quelle dei linguaggi di programmazione.

Dal punto di vista sintattico, un linguaggio può essere visto come un insieme di sequenze finite di simboli dette frasi, dove ogni simbolo appartiene ad un insieme detto *alfabeto* del linguaggio. Per definire la sintassi di un linguaggio occorre stabilire quindi:

- quali simboli appartengono al suo alfabeto;
- quali sequenze finite di elementi dell'alfabeto compongono il linguaggio.

Se la sintassi stabilisce quali sequenze di simboli siano formule logiche, non dice tuttavia niente sul significato delle stesse. Ciò è vero in tutti i linguaggi, formali o no: ad esempio, potremmo conoscere tutte le regole sintattiche per la formazione di programmi in un determinato linguaggio di programmazione, senza tuttavia essere in grado di usarlo.

Il significato di formule logiche viene definito dalla *semantica* del linguaggio, che ruota attorno al concetto di *valore di verità*: "vero" o "falso". Per stabilire il significato di una formula arbitraria, è innanzitutto necessario stabilire il significato delle formule più semplici, o *atomiche*, che la compongono; attribuire tale significato vuol dire dare un'*interpretazione* delle formule atomiche. In un sistema logico, solo fornendo un'interpretazione delle formule atomiche è possibile in generale stabilire, usando le regole del sistema, il significato di formule arbitrarie. Queste regole sono simili a quelle che permettono di valutare un'espressione algebrica a partire dalla valutazione dei suoi termini atomici. Una volta in grado di valutare formule qualsiasi, ci concentreremo sulle relazioni tra formule e interpretazioni, e sulle relazioni semantiche fra formule diverse. Definiremo quindi alcune classi particolari di formule, ad esempio quelle *valide*, ovvero che sono vere per qualsiasi interpretazione delle formule atomiche. Altre classi importanti di formule sono quelle *insoddisfacibili*, cioè false per qualsiasi interpretazione, e quelle *soddisfacibili*, cioè vere per almeno una interpretazione. Una interessante relazione semantica fra formule logiche che analizzeremo è quella dell'*implicazione logica*, che permette di stabilire quando, assumendo vera una certa formula, siamo obbligati ad accettare come conseguenza la verità di un'altra.

Inizialmente concentreremo la nostra attenzione su due sistemi logici distinti: la logica proposizionale e la logica dei predicati del prim'ordine, che è una generalizzazione della prima. Il nostro vero obiettivo è utilizzare il secondo sistema, ma inizieremo dal primo perché il suo studio ci consente di prendere familiarità con gran parte dei concetti fondamentali relativi sia alla sintassi che alla semantica. La descrizione proseguirà con altri sistemi logici che generalizzano ulteriormente i precedenti: la logica temporale e la logica dei predicati del second'ordine.

I motivi per cui la logica viene illustrata nell'ambito di questo testo sono molteplici. Innanzitutto la logica ha grande valore formativo e culturale, pari a quello di altre discipline formali come l'analisi matematica. La logica costituisce uno degli strumenti fondamentali per la specifica ed il ragionamento

formali. Uno dei contributi formativi più rilevanti che lo studio della logica fornisce è proprio la distinzione fra sintassi e semantica di un linguaggio formale. La semplicità della logica rispetto ad altri sistemi permette di affrontare entrambi questi aspetti in maniera profonda e completa.

L'altro importante motivo del nostro interesse è la rilevanza della logica nell'ambito della progettazione del software. La logica è un linguaggio universale per la descrizione formale di proprietà, e in quanto tale può essere usata per esprimere specifiche formali del software e proprietà di programmi. Nei capitoli successivi analizzeremo questo aspetto. Fra le proprietà più interessanti di un programma o sottoprogramma c'è la specifica delle condizioni che i dati di ingresso devono soddisfare affinché la sua esecuzione sia significativa. Ad esempio, una procedura per l'inserimento di un record in una lista potrebbe avere come *precondizione* il fatto che la lista sia ordinata. Un'altra proprietà interessante è la condizione che il progettista del sottoprogramma desidera che esso soddisfi dopo la sua esecuzione. Ad esempio, la procedura a cui si è fatto riferimento potrebbe avere come *postcondizione* il fatto che la lista risultante sia ancora ordinata. Le precondizioni e le postcondizioni sono degli interessanti strumenti a disposizione del progettista software, la cui rilevanza avremo occasione di apprezzare in diversi contesti, quali la specifica dei diagrammi delle classi UML (cfr. capitolo **??**).

3.2 Sintassi

La *logica proposizionale* è un sistema logico molto semplice, che si occupa della verità e della falsità di affermazioni, dette *proposizioni* o *formule proposizionali*. La semplicità del sistema dipende dal ridotto numero di modi con cui è possibile formare formule della logica proposizionale. Discuteremo di questa povertà di espressione nel capitolo 4.1, dopo aver fornito in questo paragrafo la definizione della sintassi e della semantica della logica proposizionale.

Come accennato nel paragrafo 3.1, per definire un linguaggio dobbiamo innanzitutto stabilire quali simboli appartengano al suo alfabeto.

Definizione 3.2.1 (Alfabeto della logica proposizionale) *L'alfabeto della logica proposizionale è dato da:*

- *un insieme* **LP** *di* lettere proposizionali*;*
- *i* connettivi logici $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\equiv$, *che vengono letti rispettivamente come "not", "or", "and", "implica", "equivale a";*
- *i* simboli speciali *"(" e "(".*

L'insieme **LP** di lettere proposizionali può essere scelto in molti modi senza influenzare la struttura sintattica del linguaggio. Negli esempi che seguono saranno adottate lettere minuscole dell'alfabeto italiano, ovvero assumeremo $\mathbf{LP} = \{a, b, c, \ldots, u, v, z\}$.

Dobbiamo ora specificare quali frasi, cioè sequenze finite di elementi dell'alfabeto, compongono il linguaggio.

Definizione 3.2.2 (Formule proposizionali) *Il linguaggio della* logica proposizionale *è costituito dall'insieme* **PROP** *di* formule *definito induttivamente come segue:*

- *se ϕ è una lettera proposizionale, cioè $\phi \in \mathbf{LP}$, allora ϕ è una formula;*
- *se ϕ e ψ sono formule, lo sono anche:*
  - (ϕ)
  - $\neg\phi$
  - $\phi \vee \psi$
  - $\phi \wedge \psi$
  - $\phi \rightarrow \psi$
  - $\phi \equiv \psi$

Ricordiamo che nella definizione induttiva di un insieme si specifica implicitamente che tutto ciò che non soddisfa le condizioni date non appartiene all'insieme che si sta definendo. Nel caso delle formule proposizionali quindi, la definizione è implicitamente completata da un'ulteriore clausola che asserisce che niente che non derivi dalle clausole precedenti è una formula proposizionale.

Si noti che la nozione di linguaggio che stiamo definendo è una nozione sintattica, in quanto non viene detto nulla sul significato inteso delle frasi del linguaggio. Questo aspetto, che è relativo alla semantica del linguaggio, è analizzato nel paragrafo 3.3.

Esempio 3.2.3 Le seguenti sequenze di simboli dell'alfabeto della logica proposizionale sono formule della logica proposizionale.

1. p
2. $(p \wedge q) \vee \neg s$
3. $p \wedge q \vee \neg s$
4. $p \wedge (q \vee \neg s)$
5. $s \wedge (\neg s)$
6. $(p \wedge \neg p) \rightarrow q$
7. $p \rightarrow q \equiv s$
8. $(p \rightarrow q) \equiv s$
9. $p \rightarrow (q \equiv s)$
10. $p \wedge q \wedge s$
11. $(p \wedge q) \wedge s$
12. $p \wedge (q \wedge s)$

Al contrario, le sequenze

(13) $\wedge p$

(14) $(p \wedge q \vee (\neg s)$

(15) $p \rightarrow \rightarrow q \equiv s$

(16) $p \rightarrow q) \equiv s$

non sono formule della logica proposizionale. ◦

3.3 Semantica

La semantica di un linguaggio si occupa di assegnare ad ogni frase un significato. Vedremo in questo paragrafo che la semantica della logica proposizionale assegna ad ogni formula un insieme di *interpretazioni* che la rendono "vera", dove un'interpretazione è un'assegnazione di un *valore di verità*, "vero" o "falso", ad ogni lettera proposizionale.

Come si è visto, ogni lettera proposizionale è una formula (si veda l'esempio 3.2.3 del paragrafo 3.2). La semantica della logica proposizionale parte dalla considerazione che, dal punto di vista semantico, una lettera proposizionale esprime la veridicità o falsità di un concetto atomico, assumendo uno dei due valori di verità.

Il punto fondamentale è che il valore di verità di una lettera non è stabilito all'interno della logica. In altre parole, deve esistere un meccanismo esterno alla logica che assegna un valore di verità alle lettere proposizionali.

Per comprendere meglio questa affermazione, facciamo un parallelo con le espressioni algebriche nell'ambito dei numeri interi, come $(x+y) \cdot z$. La semantica di tali espressioni parte dalla considerazione che ogni variabile può assumere un valore intero. D'altra parte, il valore dell'espressione può essere stabilito solamente nel momento in cui si stabilisce il valore delle variabili in gioco (nell'esempio, se $x = -2$, $y = 1$, $z = 4$, l'espressione vale -4).

Una volta fissato il valore di verità delle lettere proposizionali, ogni formula complessa (come le formule da 1 a 12 dell'esempio 3.2.3) assume anch'essa un valore di verità, e questo valore è determinato in funzione del valore di verità attribuito alle lettere proposizionali occorrenti in essa.

Le nozioni fondamentali relative alla semantica della logica proposizionale verranno illustrate nel seguito mediante tre passi fondamentali.

1. Si definisce la nozione di *interpretazione delle lettere proposizionali*, cioè di quel meccanismo che consente di assegnare ad ogni lettera proposizionale un valore di verità.
2. Si definisce la legge per stabilire il significato di ogni formula rispetto ad una interpretazione, cioè come viene valutata una formula per *una particolare interpretazione* delle lettere proposizionali.
3. Si stabilisce il significato di ogni formula senza riferimento a particolari interpretazioni, e si illustrano diverse nozioni interessanti legate a tale significato.

Solo al passo 3 si determina l'associazione tra una formula ed un insieme di interpretazioni che la rendono vera. Consideriamo la seguente definizione.

Definizione 3.3.1 (Interpretazione nella logica proposizionale) *Una* interpretazione *I è una funzione:*

$$I : \mathbf{LP} \longrightarrow \{\mathsf{T}, \mathsf{F}\}$$

avente come dominio l'insieme **LP** *delle lettere proposizionali e come codominio un valore di verità, cioè l'insieme* $\{\mathsf{T}, \mathsf{F}\}$ *dove* T *significa "vero" e* F *significa "falso".*

Esempio 3.3.2 Si supponga di associare alle lettere p ed e rispettivamente il valore di verità che attribuiamo ai concetti "la torre di Pisa è pendente" e "la torre Eiffel è pendente". Si ottiene una interpretazione, che chiamiamo TP, tale che $TP(p) = \mathsf{T}$ e $TP(e) = \mathsf{F}$. Questa interpretazione, che esprime il valore di verità dei due concetti, non esprime alcun'altra proprietà delle due torri. ◦

Per il passo 2, definiamo, per una data interpretazione, una funzione in grado di valutare una formula in quella interpretazione.

Definizione 3.3.3 (Valutazione di una formula proposizionale rispetto ad un'interpretazione) *Sia I una interpretazione. Definiamo, in dipendenza da I, la funzione*

$$\mathsf{eval}^I : \mathbf{PROP} \longrightarrow \{\mathsf{T}, \mathsf{F}\}.$$

dove **PROP** *denota l'insieme delle formule della logica proposizionale. La funzione è definita in modo induttivo, seguendo la struttura induttiva delle formule:*

- *se p è una lettera proposizionale allora*

$$\mathsf{eval}^I(p) = I(p)$$

- *se ϕ e ψ sono formule allora*

$$\begin{array}{lllll}
(1) & \mathsf{eval}^I(\ (\phi)\ ) & = & \mathsf{eval}^I(\phi) & \\
(2) & \mathsf{eval}^I(\neg\phi) & = & \mathsf{T} \textit{ se } \mathsf{eval}^I(\phi) = \mathsf{F} & \\
 & \mathsf{eval}^I(\neg\phi) & = & \mathsf{F} \textit{ altrimenti} & \\
(3) & \mathsf{eval}^I(\phi\vee\psi) & = & \mathsf{T} \textit{ se } \mathsf{eval}^I(\phi) = \mathsf{T} \textit{ oppure } \mathsf{eval}^I(\psi) = \mathsf{T} & \\
 & \mathsf{eval}^I(\phi\vee\psi) & = & \mathsf{F} \textit{ altrimenti} & \\
(4) & \mathsf{eval}^I(\phi\wedge\psi) & = & \mathsf{T} \textit{ se } \mathsf{eval}^I(\phi) = \mathsf{eval}^I(\psi) = \mathsf{T} & \\
 & \mathsf{eval}^I(\phi\wedge\psi) & = & \mathsf{F} \textit{ altrimenti} & \\
(5) & \mathsf{eval}^I(\phi\rightarrow\psi) & = & \mathsf{eval}^I(\neg\phi\vee\psi) & \\
(6) & \mathsf{eval}^I(\phi \equiv \psi) & = & \mathsf{eval}^I(\ (\phi\rightarrow\psi)\wedge(\psi\rightarrow\phi)\ ) &
\end{array}$$

Data una formula della logica proposizionale ϕ e un'interpretazione I:

- *se $\mathsf{eval}^I(\phi) = \mathsf{T}$, scriveremo anche $I \models \phi$,*
- *se $\mathsf{eval}^I(\phi) = \mathsf{F}$, scriveremo anche $I \not\models \phi$.*

Facciamo alcune osservazioni riguardo la definizione data.

- Riferendosi a (5) e (6), il fatto che le formule a sinistra siano riconducibili ad altre prive di $\rightarrow$ e $\equiv$, significa che questi connettivi sono ridondanti.
- In effetti anche $\wedge$ è ridondante, in quanto il suo significato in (4) potrebbe essere dato in termini dei significati di $\neg$ e di $\vee$. I connettivi $\neg$ e $\vee$ formano pertanto un insieme minimale che può servire per definire tutti gli altri. Esistono altri insiemi minimali di questo tipo, ad esempio quello formato da $\wedge$ e $\neg$.
- Dal significato dell'implicazione $\phi\rightarrow\psi$ si evince che: nel caso che ϕ sia vera, l'unico modo perché la formula complessiva sia vera è che anche ψ sia vera; nel caso che ϕ sia falsa allora la formula complessiva è comunque vera. In altre parole, l'unica possibilità di falsificare la formula $\phi\rightarrow\psi$ è che ϕ sia vera e ψ sia falsa.

ϕ	ψ	$\neg\phi$	$\phi\vee\psi$	$\phi\wedge\psi$	$\phi\rightarrow\psi$	$\phi\equiv\psi$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

Tabella 3.1 Tavola di verità per i connettivi proposizionali

La funzione eval^I può essere rappresentata dalla *tavola di verità* riportata in tabella 3.1.

Un aspetto che occorre chiarire subito è quello relativo alla potenziale *ambiguità* nella valutazione delle formule: facendo riferimento all'esempio 3.2.3, non sappiamo se la formula (3) debba essere valutata come la (2) oppure come la (4). La stessa cosa può essere detta delle formule (7), (8), (9) e (10), (11), (12).

Una maniera per rendere non ambigua la valutazione di una formula è senz'altro quella di usare le parentesi nella formula stessa. Per rendere più leggibili le formule, faremo ricorso a convenzioni sulla *precedenza* dei connettivi logici. Queste convenzioni sono del tutto simili a quelle comunemente usate, sia nei testi di matematica che nei linguaggi di programmazione, per gli operatori nelle espressioni algebriche, e che portano ad esempio ad interpretare l'espressione $-4+5*8+2$ come $((-4)+(5*8))+2$.

Allo scopo di interpretare univocamente qualsiasi formula viene adottata la seguente gerarchia tra i connettivi:

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\rightarrow$, $\equiv$

Eventuali ambiguità in una formula vengono risolte facendo riferimento a tale gerarchia. Così la formula $p\wedge q\vee\neg s$ viene interpretata come $(p\wedge q)\vee\neg s$, perché $\wedge$ ha priorità rispetto a $\vee$ nella gerarchia. Inoltre i connettivi sono *associativi a sinistra*. Ciò significa che, in casi di ambiguità in cui siano coinvolti connettivi dello stesso livello, ha priorità quello posizionato più a sinistra. Così, la formula $p\rightarrow q\equiv s$ viene interpretata come $(p\rightarrow q)\equiv s$.

Esempio 3.3.4 Consideriamo una interpretazione *Tuttofalso* che assegna F a qualsiasi lettera proposizionale $l\in\mathbf{LP}$. Se riconsideriamo l'esempio 3.2.3 si avrà

(1)	$\mathsf{eval}^{Tuttofalso}(p)$	$=\mathsf{F}$	(7)	$\mathsf{eval}^{Tuttofalso}(p\rightarrow q\equiv s)$	$=\mathsf{F}$
(2)	$\mathsf{eval}^{Tuttofalso}((p\wedge q)\vee\neg s)$	$=\mathsf{T}$	(8)	$\mathsf{eval}^{Tuttofalso}((p\rightarrow q)\equiv s)$	$=\mathsf{F}$
(3)	$\mathsf{eval}^{Tuttofalso}(p\wedge q\vee\neg s)$	$=\mathsf{T}$	(9)	$\mathsf{eval}^{Tuttofalso}(p\rightarrow(q\equiv s))$	$=\mathsf{T}$
(4)	$\mathsf{eval}^{Tuttofalso}(p\wedge(q\vee\neg s))$	$=\mathsf{F}$	(10)	$\mathsf{eval}^{Tuttofalso}(p\wedge q\wedge s)$	$=\mathsf{F}$
(5)	$\mathsf{eval}^{Tuttofalso}(s\wedge(\neg s))$	$=\mathsf{F}$	(11)	$\mathsf{eval}^{Tuttofalso}((p\wedge q)\wedge s)$	$=\mathsf{F}$
(6)	$\mathsf{eval}^{Tuttofalso}((p\wedge\neg p)\rightarrow q)$	$=\mathsf{T}$	(12)	$\mathsf{eval}^{Tuttofalso}(p\wedge(q\wedge s))$	$=\mathsf{F}$

∘

Esercizio 3.1 Si consideri l'interpretazione *Tuttovero* tale che $Tuttovero(l)=\mathsf{T}$ per qualsiasi lettera proposizionale $l\in\mathbf{LP}$. Si valuti ciascuna formula dell'esempio 3.2.3 rispetto a *Tuttovero*. ∘

Introduciamo ora alcune importanti nozioni che verranno usate in seguito.

Definizione 3.3.5 (Modello) *Una interpretazione I* soddisfa *una formula ψ se $I\models\psi$ (ovvero se $\mathsf{eval}^I(\psi)=\mathsf{T}$). Una interpretazione I che soddisfa una formula ψ è detta* modello *di ψ.*

Data una interpretazione I e una formula ψ, determinare se vale $I\models\psi$ oppure no viene chiamato il problema della *validazione di modello*, che traduce l'inglese *model checking*.

Esempio 3.3.6 Considerando solamente le interpretazioni delle lettere proposizionali p ed f, la formula proposizionale $p\rightarrow f$ ha tre modelli, che sono le interpretazioni M_1, M_2, M_3 definite da:

- $M_1(p) = \mathsf{T}, M_1(f) = \mathsf{T}$
- $M_2(p) = \mathsf{F}, M_2(f) = \mathsf{T}$
- $M_3(p) = \mathsf{F}, M_3(f) = \mathsf{F}$

Si noti che delle quattro interpretazioni possibili, solamente quella che rende p vera e f falsa non è un modello. ◦

Non appena viene associato ad ogni lettera proposizionale il ruolo di rappresentare la verità o la falsità di un concetto relativo ad una certa realtà sotto osservazione, una singola interpretazione esprime un dato assetto di quella realtà. Se stabiliamo che il significato di una formula coincide con l'insieme dei suoi modelli, allora è possibile considerare una formula come l'espressione dell'insieme di assetti della realtà sotto osservazione associati ai modelli della formula stessa. Per questo motivo stabiliamo che *il significato di una formula coincide con l'insieme dei suoi modelli.*

Esempio 3.3.7 Se la lettera p è associata alla veridicità del concetto "la torre di Pisa è pendente" e la lettera f a quella del concetto "Pisa è famosa nel mondo", allora possiamo dire che i modelli della formula $p \rightarrow f$ rappresentano alcuni aspetti del mondo reale. Infatti, l'unica interpretazione che non è un modello è quella che corrisponde al fatto che la torre di Pisa è pendente, ma tale città non è famosa nel mondo, circostanza questa del tutto improbabile e giustamente esclusa. D'altra parte, tutti gli altri modelli sono assetti ragionevoli della realtà: se la torre è pendente, allora la città è sicuramente famosa (cfr. M_1 nell'esempio 3.3.6), mentre se non dovesse essere più pendente, la città potrebbe rimanere famosa oppure no (cfr. M_2 ed M_3). Concludiamo l'esempio notando che la formula $p \rightarrow f$ non esprime necessariamente il fatto che la torre di Pisa sia pendente. ◦

Data una formula ci si può chiedere:

- se essa non ha modelli;
- se essa ha almeno un modello;
- se ogni interpretazione sia un suo modello.

Definizione 3.3.8 (Formule soddisfacibili, insoddisfacibili e valide) *Sia ϕ una formula. ϕ è* soddisfacibile *se esiste un'interpretazione che è un modello di ϕ, ϕ è* insoddisfacibile *altrimenti. Inoltre, ϕ è* valida *se ogni interpretazione è un modello di ϕ (in questo caso si dice anche che ϕ è una* tautologia*).*

Definizione 3.3.9 (Equivalenza logica) *Siano ϕ e ψ due formule. Si dice che ϕ è* logicamente equivalente *a ψ se ϕ e ψ hanno gli stessi modelli.*

Esercizio 3.2 [Soluzione a pagina 35] Dimostrare che ogni ogni formula ϕ dell'insieme **PROP** di cui alla definizione 3.2.2 può essere riscritta in una formula ϕ' equivalente a ϕ e in cui occorrono solo i connettivi dell'insieme $\{\neg, \vee\}$ o dell'insieme $\{\neg, \wedge\}$. ◦

Esercizio 3.3 Si riconsiderino le formule mostrate negli esempi 3.2.3 e 3.3.4, e si riscriva ognuna di esse in maniera equivalente utilizzando solo i connettivi dell'insieme $\{\neg, \vee\}$ o dell'insieme $\{\neg, \wedge\}$ secondo il metodo sviluppato come soluzione per l'esercizio 3.2. ◦

Esercizio 3.4 [Soluzione a pagina 35] Si riconsiderino le formule mostrate negli esempi 3.2.3 e 3.3.4, e si dica per ognuna di esse se è soddisfacibile, insoddisfacibile o valida. ◦

Esercizio 3.5 Dimostrare che una formula proposizionale α è valida se e solo se la formula $\neg\alpha$ è insoddisfacibile. ◦

Utilizziamo ora i modelli delle formule per definire una fondamentale relazione semantica tra formule.

Definizione 3.3.10 (Implicazione logica) *Siano ϕ e ψ due formule. Si dice che ϕ* implica logicamente *ψ (scritto $\phi \models \psi$) se ogni modello di ϕ è anche un modello di ψ.*

Esempio 3.3.11 Se p e q sono lettere proposizionali, allora

è vero che	NON è vero che
$p \models p$	$p \models q$
$p \models p \vee q$	$p \vee q \models p$
$p \wedge \neg p \models q$	$p \vee \neg p \models q$
$p \models q \vee \neg q$	$p \models q \wedge \neg q$

◦

Esercizio 3.6 Siano date due formule proposizionali ϕ e ψ. Dimostrare che vale $\phi \models \psi$ se e solo se $\phi \rightarrow \psi$ è valida. Dimostrare inoltre che vale $\phi \models \psi$ se e solo se $\phi \wedge \neg\psi$ è insoddisfacibile. ◦

La definizione appena data ci consente di definire quando due formule hanno esattamente lo stesso significato, ovvero di dare la definizione 3.3.9 in maniera alternativa: siano ϕ e ψ due formule, ϕ è logicamente equivalente a ψ se $\phi \models \psi$ e $\psi \models \phi$.

Le ultime nozioni introdotte possono facilmente essere estese a insiemi di formule.

Definizione 3.3.12 (Proprietà di insiemi di formule) *Sia Σ un insieme di formule.*

- *Un* modello *di Σ è una interpretazione che è un modello di ogni formula in Σ.*
- Σ *è* soddisfacibile *se esiste un'interpretazione che è un modello di Σ, Σ è* insoddisfacibile *altrimenti.*
- *Sia ψ una formula. Si dice che Σ* implica logicamente *ψ (scritto $\Sigma \models \psi$) se ogni modello di Σ è anche un modello di ψ.*

Esempio 3.3.13 $\{p \vee q, \neg p\} \models q$ ◦

Concludiamo questo paragrafo notando che anche il problema della validazione di modello può essere ridotto ad un problema di soddisfacibilità.

Proposizione 3.3.14 *Data una interpretazione M e una formula ϕ, $M \models \phi$ vale se e solo se la formula*

$$\phi \ \wedge \bigwedge_{p \in \mathbf{LP}, \mathsf{eval}^M(p) = \mathsf{T}} p \ \wedge \bigwedge_{p \in \mathbf{LP}, \mathsf{eval}^M(p) = \mathsf{F}} \neg p \tag{3.1}$$

è soddisfacibile.

Notiamo che se la formula (3.1) è soddisfacibile, allora ha esattamente un modello, e tale modello è M.

3.4 Aspetti computazionali

In questo paragrafo ci concentriamo su alcuni aspetti computazionali delle nozioni viste. In particolare, faremo alcune considerazioni sul problema di decidere se una data formula sia soddisfacibile oppure no.

3.4.1 Complessità computazionale

Notiamo innanzitutto che, se possediamo un algoritmo per il problema della soddisfacibilità, allora siamo in grado di scriverne facilmente altri per i problemi dell'insoddisfacibilità, della validità e dell'implicazione logica (cfr. esercizi 3.5 e 3.6).

Sia ϕ una formula proposizionale e $\mathbf{L}$ l'insieme delle lettere che compaiono in ϕ. Poiché $\mathbf{L}$ è un insieme finito, e quindi il numero delle possibili interpretazioni delle sue lettere è anch'esso finito, e poiché il calcolo di $\mathsf{eval}^I(\phi)$ è immediato, possiamo definire il seguente semplice algoritmo per decidere se ϕ sia soddisfacibile oppure no:

1. calcola l'insieme $\mathbf{L}$ delle lettere che compaiono in ϕ;
2. genera tutte le possibili interpretazioni delle lettere di $\mathbf{L}$, e per ogni interpretazione I generata
   (a) valuta ϕ rispetto ad I, calcolando $\mathsf{eval}^I(\phi)$;
   (b) se $\mathsf{eval}^I(\phi) = \mathsf{T}$, allora termina, segnalando che ϕ è soddisfacibile;

3. se sono state generate tutte le interpretazioni delle lettere di **L** senza successo, allora segnala che ϕ è insoddisfacibile.

L'algoritmo appena delineato è di tipo *enumerativo*, in quanto enumera tutte le interpretazioni possibili di **L**.

È opportuno qualche commento sull'efficienza dell'algoritmo. Il numero di interpretazioni possibili che vengono considerate nel passo 2 è 2^n, dove n è la cardinalità di **L**. Come noto, ciò significa che l'algoritmo ha complessità *esponenziale* nel caso peggiore, ovvero che in pratica può essere eseguito solamente su formule ϕ di dimensione sufficientemente piccola: se n è pari a 50, allora le interpretazioni possibili sono più di un milione di miliardi; se un calcolatore è in grado di considerare cento milioni di interpretazioni al secondo, allora sono necessari più di quattro mesi per esaurirle tutte; se n è pari a 60, tale tempo aumenta di un fattore pari a 1024.

Come ultimo commento, accenniamo ad una caratteristica del problema della soddisfacibilità di una formula della logica proposizionale che gli fa rivestire un particolare interesse. Una delle questioni teoriche più affascinanti dell'informatica, che è stata formulata nel 1971 ed ancora attende risposta, consiste nella domanda: "esiste un algoritmo di costo polinomiale per il problema della soddisfacibilità?". Comunemente vengono chiamati problemi *polinomialmente trattabili* quei problemi che hanno delimitazione superiore alla complessità $O(p(n))$, dove $p(n)$ è un polinomio fissato a priori. In generale si tende a considerare i problemi polinomialmente trattabili come quelli per cui esiste un algoritmo efficiente. Per molti dei problemi comuni in informatica, ed in particolare per il problema della soddisfacibilità di una formula della logica proposizionale, non sono noti algoritmi di costo polinomiale e non è stato dimostrato che tali algoritmi non possano esistere. Altri esempi di problemi di questo tipo, che appartengono alla cosiddetta classe dei problemi *NP-hard*, sono quello della "bisaccia" e quello del "partizionamento". Il problema della validazione di modello (cf. definizione 3.3.5) è invece polinomialmente trattabile.

3.4.2 Algoritmi

In questo paragrafo accenniamo ad alcuni algoritmi per il problema della soddisfacibilità di una formula della logica proposizionale che, pur non essendo di costo polinomiale, risultano molto più efficienti dell'algoritmo di tipo enumerativo descritto nel paragrafo precedente.

La maggior parte degli algoritmi si riferiscono ad una classe di formule proposizionali strettamente contenuta nell'insieme di cui alla definizione 3.2.2: l'insieme delle formule proposizionali in CNF. Una formula proposizionale si dice in *forma normale congiuntiva* (CNF) se ha la forma:

$$c_1 \wedge \cdots \wedge c_n$$

dove n è un intero positivo o nullo e ogni c_i $(1 \leq i \leq n)$ è una *clausola*, ovvero una formula proposizionale della forma:

$$l_1 \vee \cdots \vee l_{m_i}$$

dove m_i è un intero positivo o nullo e ogni l_j $(1 \leq j \leq m_i)$ è un *letterale positivo*, ovvero una lettera proposizionale, oppure un *letterale negativo*, ovvero la negazione di una lettera proposizionale. Se m_i vale zero, si dice che la clausola è *vuota*. Se n vale zero, si dice che la formula in CNF è *vuota*.

Esempio 3.4.1 Esempi di formule in CNF che fanno riferimento all'insieme $\mathbf{LP} = \{a, b, c\}$ sono i seguenti:

- (formula f_1) $(a \vee \neg b) \wedge (b) \wedge (\neg b \vee c)$;
- (formula f_2) $(\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$;
- (formula f_3) $(a \vee b) \wedge (a \vee c) \wedge (\neg a)$.

∘

Le formule in CNF sono particolarmente interessanti perché è possibile trasformare in tempo polinomiale qualsiasi formula proposizionale ϕ in una formula in CNF ψ tale che ψ è soddisfacibile se e solo ϕ è soddisfacibile. Per questo motivo da ora in poi faremo riferimento al problema della soddisfacibilità di una formula in CNF, noto come problema "SAT". SAT è considerato il prototipo dei problemi NP-hard.

La seguente definizione estende la 3.3.1 al caso di interpretazioni *parziali*, che potrebbero non assegnare alcun valore ad una lettera proposizionale.

Definizione 3.4.2 (Assegnazione, estensione, semplificazione) *Un'*assegnazione *I è una* funzione parziale *avente come dominio l'insieme* **LP** *delle lettere proposizionali e come codominio l'insieme* $\{\mathsf{T}, \mathsf{F}\}$. *Un'assegnazione verrà rappresentata come l'insieme dei letterali che, in base all'assegnazione stessa, assumono il valore* T.

*Data un'assegnazione α ed un letterale $l \notin \alpha$, l'assegnazione che comprende anche il letterale l verrà denotata da αl. Diremo che αl è un'*estensione *di α.*

Sia ϕ una formula in CNF e α un'assegnazione. $\phi|\alpha$ denota la formula in CNF ottenuta:

- *cancellando da ϕ ogni clausola che è soddisfatta da α, e*
- *cancellando da ogni clausola di ϕ ogni letterale che è falso in α.*

$\phi|\alpha$ viene chiamata la semplificazione *di ϕ mediante α.*

Esempio 3.4.3 Con riferimento all'esempio 3.4.1, possiamo definire tre assegnazioni:

- I_1: $I_1(a) = \mathsf{T}$, $I_1(b) = \mathsf{T}$, $I_1(c) = \mathsf{T}$;
- I_2: $I_2(a) = \mathsf{F}$;
- I_3: $I_3(a) = \mathsf{F}$, $I_3(b) = \mathsf{T}$, $I_3(c) = \mathsf{T}$, $I_3(d) = \mathsf{T}$.

∘

Notiamo che alcune assegnazioni incomplete rappresentano implicitamente modelli. Nell'esempio 3.4.3, I_2 è sufficiente a determinare (secondo le regole della definizione 3.3.3) che f_2 è soddisfacibile, in quanto ogni clausola contiene un letterale ($\neg a$) che è assegnato a T. Di conseguenza possiamo completare I_2 in maniera arbitraria (ad esempio assegnando Fa tutte le variabili non assegnate), ed ognuno dei completamenti è un modello di ϕ.

Esempio 3.4.4 Con riferimento all'esempio 3.4.1 e all'assegnazione $\alpha = \{a\}$, abbiamo:

- $f_1|\alpha$: $(b) \wedge (\neg b \vee c)$;
- $f_2|\alpha$: $(b \vee c) \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$;
- $f_3|\alpha$: $()$.

Notiamo che nell'ultima formula sono state cancellate le prime due clausole, e che la terza ha perso il suo unico letterale, ovvero è divenuta vuota. ∘

3.4.2.1 Algoritmo di backtracking

Un semplice algoritmo per SAT che si basa sulla tecnica del *backtracking* (dall'inglese, percorrere all'inverso il proprio cammino) sfrutta l'idea di effettuare assegnazioni (parziali) e semplificare la formula sulla base di esse. Se la semplificazione ha eliminato tutte le clausole (`SAT`), allora non c'è bisogno di completare l'assegnazione, perché i letterali scelti sono sufficienti a determinare la soddisfacibilità della formula. Se viceversa la semplificazione ha creato una clausola vuota (`Conflict`) allora l'assegnazione va *accorciata*, in quanto contiene già scelte sbagliate. Negli altri casi (`Branch`), l'assegnazione va *allungata*, scegliendo un letterale e, in caso di fallimento, il suo negato.

Una versione ricorsiva dell'algoritmo è la seguente.

```
{SODDISFACIBILE, INSODDISFACIBILE} BT(CNF phi, Assegnazione alpha)
/* SAT */      if (phi|alpha) \‘e vuota
                  return SODDISFACIBILE
/* Conflict */ if (phi|alpha) contiene una clausola vuota
                  return INSODDISFACIBILE
/* Branch */   sia p un letterale di phi|alpha
               if BT(phi,alpha p) == SODDISFACIBILE
                  return SODDISFACIBILE
               else
                  return BT(phi,alpha NOT p)
```

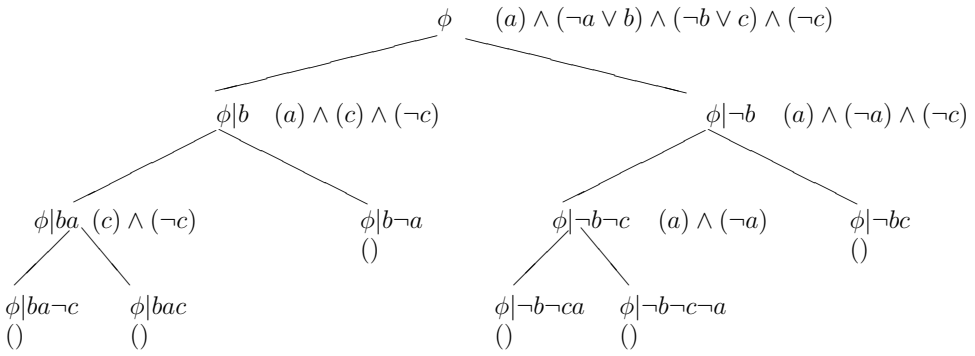

Figura 3.1 Passi dell'esecuzione dell'algoritmo `BT` per la formula $\phi \doteq (a) \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c)$

La correttezza dell'algoritmo è garantita dal fatto che ogni formula in CNF vuota è soddisfacibile per definizione, mentre ogni formula in CNF che contiene almeno una clausola vuota è insoddisfacibile per definizione. Notiamo che, se l'algoritmo restituisce `SODDISFACIBILE` per una certa formula ϕ, l'assegnazione corrente (α, αp, o $\alpha\neg p$, a seconda dei casi) rappresenta almeno un modello della formula in CNF. In particolare, se l'assegnazione è completa, allora essa è un modello di ϕ, altrimenti, possiamo completarla in maniera arbitraria, ed ognuno dei completamenti è un modello di ϕ.

Esempio 3.4.5 Consideriamo la formula in CNF $\phi \doteq (a) \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c)$. La rappresentazione (chiamata *albero del backtracking*) di una possibile esecuzione dell'algoritmo `BT` è riportata in figura 3.1. In particolare, la visita in preordine dell'albero rappresenta la sequenza dei passi dell'algoritmo. Si noti che la scelta del letterale "di branch" (variabile e segno) è arbitraria, e nello stesso livello non si sceglie necessariamante lo stesso letterale. Ad ogni nodo è associata l'assegnazione relativa e il risultato della semplificazione della formula con tale assegnazione. Il fatto che ad ogni foglia dell'albero sia associata la clausola vuota ("()") significa che ϕ è insoddisfacibile. ◦

Esercizio 3.7 Disegnare l'albero di backtracking per le seguenti formule in CNF:

- $\phi_2 \doteq (a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee \neg c)$ (soddisfacibile);
- ϕ_3, che contiene le otto clausole che si possono costruire con i tre letterali dell'insieme $\mathbf{LP} = \{a, b, c\}$ (insoddisfacibile).

◦

Con riferimento all'algoritmo enumerativo del paragrafo 3.4.1, l'algoritmo `BT` è più efficiente, in quanto evita di considerare interpretazioni complete, ma sfrutta i fallimenti delle assegnazioni incomplete. Anche `BT` ha complessità esponenziale, ma in pratica offre prestazioni notevolmente superiori.

3.4.2.2 Algoritmo DPLL

L'algoritmo DPLL (dalle iniziali del cognome dei suoi creatori Davis, Putnam, Logemann e Loveland che l'hanno ideato all'inizio degli anni '60) introduce vari miglioramenti rispetto a `BT`.

Il primo riguarda le cosiddette *clausole unitarie* (`UnitClause`), ovvero che contengono un solo letterale. Con riferimento all'esempio 3.4.1, la seconda clausola di f_1 è unitaria. È facile dimostrare che se una formula in CNF è soddisfacibile, allora ha un modello in cui tutti i letterali delle clausole unitarie sono veri. Di conseguenza, quando una formula in CNF ha una clausola unitaria, il suo letterale può essere aggiunto alla assegnazione (e propagato tramite una semplificazione).

Il secondo riguarda i cosiddetti *letterali puri* (`PureLiteral`) in una formula in CNF f, ovvero quei letterali l tali che in f non occorre mai $\neg l$ (ovvero il letterale di segno opposto). Con riferimento all'esempio 3.4.1, $\neg a$ è puro in f_2. È facile dimostrare che se una formula in CNF è soddisfacibile, allora ha un modello in cui tutti i letterali puri sono veri. Di conseguenza, quando una formula in CNF ha un letterale puro, esso può essere aggiunto alla assegnazione (e propagato tramite una semplificazione).

Il terzo riguarda la scelta del letterale "di branch". Poiché l'algoritmo trae vantaggio dall'avere clausole unitarie, si cerca di avere tali clausole, scegliendo il letterale di branch da una clausola di lunghezza minima.

Una versione ricorsiva dell'algoritmo è la seguente.

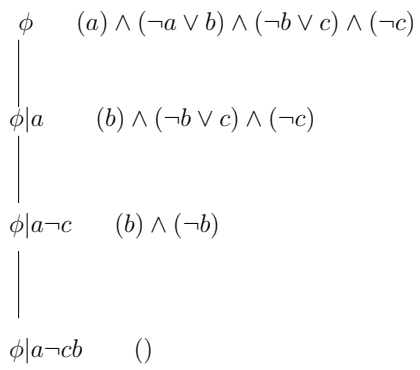

Figura 3.2 Passi dell'esecuzione dell'algoritmo DPLL per la formula $\phi \doteq (a) \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c)$

```
{SODDISFACIBILE, INSODDISFACIBILE} DPLL(CNF phi, Assegnazione alpha)
/* SAT */        if (phi|alpha) \‘e vuota return SODDISFACIBILE
/* Conflict */   if (phi|alpha) contiene una clausola vuota return INSODDISFACIBILE
/* UnitClause*/  if (phi|alpha) contiene una clausola unitaria {p}
                   return DPLL(phi,alpha p)
/* PureLiteral*/ if (phi|alpha) ha un letterale puro p
                   return DPLL(phi,alpha p)
/* Branch */     sia p un letterale da una clausola di phi|alpha di LUNGHEZZA MINIMA
                 if DPLL(phi,alpha p) == INSODDISFACIBILE
                   return SODDISFACIBILE
                 else
                   return DPLL(phi,alpha NOT p)
```

Esempio 3.4.6 Consideriamo la formula in CNF ϕ di cui all'esempio 3.4.5. La rappresentazione dell'esecuzione dell'algoritmo DPLL è riportata in figura 3.2. Si noti che non viene mai effettuato il passo Branch, ma viene sempre eseguito il passo UnitClause. Ovviamente anche in questo caso si determina che ϕ è insoddisfacibile. ◦

Esercizio 3.8 Disegnare l'albero che rappresenta l'esecuzione dell'algoritmo DPLL per le formule in CNF ϕ_2 e ϕ_3 di cui all'esercizio 3.7. ◦

Con riferimento all'algoritmo BT, DPLL è tipicamente più efficiente, in quanto sfrutta la propagazione dei letterali unitari.

3.4.2.3 Algoritmi incompleti

Le procedure BT e DPLL appartengono alla categoria degli algoritmi *corretti* e *completi*. Un algoritmo A per il problema SAT si dice corretto se, per ogni formula in CNF ϕ, quando A(ϕ) restituisce SODDISFACIBILE, allora ϕ è soddisfacibile. A si dice completo se, data una formula in CNF ϕ, quando ϕ è soddisfacibile, allora A(ϕ) restituisce SODDISFACIBILE.

Esistono anche algoritmi corretti e *incompleti* per il problema SAT. Data una formula in CNF ϕ insoddisfacibile questi algoritmi non diranno mai che ϕ è soddisfacibile, ma data una formula in CNF ψ soddisfacibile potrebbero non riuscire a dimostrarne la soddisfacibilità. Questi algoritmi possono essere efficaci quando è già noto che la formula in questione è soddisfacibile, e siamo interessati a trovarne un modello.

Un banale algoritmo corretto e incompleto è il seguente.

```
{SODDISFACIBILE, INDETERMINATA} SAT_CORRETTO_INCOMPLETO(CNF phi)
      int MAX_ITER = ...; // un numero sufficientemente grande
      for (int i = 0, i < MAX_ITER; i++) {
          Interpretazione alpha = GENERA_INTERPRETAZIONE_RANDOM();
             // assegna casualmente T o F ad ogni variabile
          if (alpha \‘e un modello di phi)
                return SODDISFACIBILE;
```

```
    }
    return INDETERMINATA;
```

Gli algoritmi implementati si basano su questo schema e spendono, per ogni interpretazione α che non è un modello, un po' di tempo cercando di "migliorarla" fino a farla diventare un modello. La tecnica di base consiste nel cambiare le assegnazioni dei singoli letterali da `T` a `F` e viceversa in modo da aumentare il numero di clausole soddisfatte. Dopo un numero prefissato di tentativi, passano ad un'altra interpretazione α. Spesso per seguire questa strategia vengono usati algoritmi di *ricerca locale*, ovvero che considerano un numero limitato di *vicini* di α e adottano euristiche come *hill climbing* o *tabu search*.

3.4.3 Programmi disponibili

Esistono decine di programmi di dominio pubblico che implementano solutori SAT, ed annualmente si svolge una competizione che premia quelli più efficienti. Molti programmi sono basati sulla procedura `DPLL` e differiscono fra loro relativamente alle euristiche per la selezione dei letterali di branch e alle strutture di dati per la rappresentazione delle clausole. Tali programmi sono in grado di determinare la soddisfacibilità di istanze di centinaia di variabili e migliaia di clausole nelle condizioni peggiori, ovvero al cosiddetto *crossover point*, che si riferisce ad una specifica generazione casuale delle formule in CNF ed è determinata sperimentalmente dal punto in cui la probabilità di una formula di essere soddisfacibile è uguale a quella di essere insoddisfacibile. Per quanto riguarda le istanze che non sono generate casualmente (le cosiddette istanze *strutturate*, che rappresentano problemi combinatori), vengono valutate formule in CNF molto grandi, che possono contenere decine di migliaia di variabili e anche un milione di clausole. Gli algoritmi incompleti (sviluppati a partire dal 1992) possono essere molto più efficienti di quelli completi.

Esiste un formato standard (chiamato DIMACS, dal nome di un centro di ricerca degli Stati Uniti) per i file che rappresentano le formule in CNF. Tale formato prevede file di testo formattati secondo queste regole:

- i letterali positivi (risp. negativi) sono rappresentati con interi positivi (risp. negativi);
- il file contiene inizialmente un numero arbitrario di righe di commento, che hanno '`c`' come carattere iniziale;
- dopo i commenti nel file è inserita una riga contenente "`p cnf `" come stringa iniziale, seguita da due interi positivi che rappresentano, rispettivamente, il numero delle variabili proposizionali distinte (ovvero la cardinalità di **LP**) e il numero di clausole;
- ogni clausola occupa una linea, che contiene la lista dei letterali corrispondenti e termina con `0`.

Ad esempio, la formula f_2 dell'esempio 3.4.1 potrebbe essere rappresentata dal seguente file.

```
c Una CNF con tre clausole
c a->1, b->2, c->3
p cnf 3 3
-1 2 3 0
-1 -2 3 0
-1 -2 -3 0
```

Vediamo ora un'implementazione in JAVA dell'algoritmo `BT`. Lo schema concettuale dei dati è rappresentato dal diagramma delle classi UML riportato in figura 3.3. Le funzioni più significative sono le seguenti:

- `FormulaCNF.semplifica(Assegnazione a)`, che implementa la nozione di semplificazione di cui alla definizione 3.4.2;
- `FormulaCNF.alfabeto()`, che restituisce l'insieme **LP** di una data formula in CNF;
- `FormulaCNF.leggiDaFileDIMACS(String)`, che costruisce un oggetto leggendo un file nel formato DIMACS;
- `Solutori.BT(FormulaCNF, Assegnazione)`, che implementa la procedura `BT`.

Il programma risultante non è molto efficiente, in parte a causa delle frequenti copie delle strutture di dati. Può essere usato per tracciare il comportamento dell'algoritmo, visualizzando lo stato della formula, le assegnazioni e il letterale scelto.

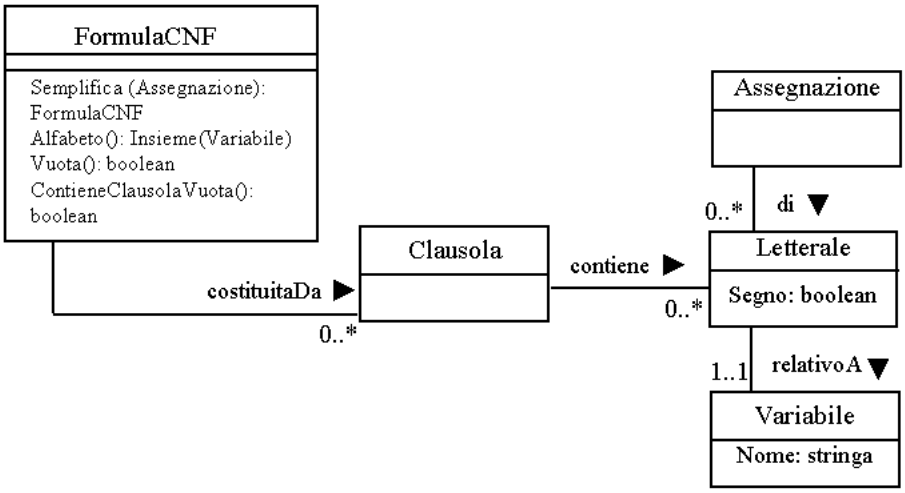

Figura 3.3 Diagramma delle classi UML per i solutori SAT

```
// File Sat/FormulaCNF.java
import java.util.*;
import java.io.*;
public class FormulaCNF implements Cloneable {
    // CAMPI DATI
    private static FileIn file;   // file di input logico
    private HashSet<Clausola> costituitaDa;
    // COSTRUTTORE
    public FormulaCNF() {
        costituitaDa = new HashSet<Clausola>();
    }
    // GESTIONE ASSOCIAZIONE
    public void inserisciLinkCostituitaDa(Clausola cla) {
        if (cla != null) costituitaDa.add(cla);
    }
    public void eliminaLinkCostituitaDa(Clausola cla) {
        if (cla != null) costituitaDa.remove(cla);
    }
    public Set<Clausola> getCostituitaDa() {
        return (HashSet<Clausola>)costituitaDa.clone();
    }
    // OPERAZIONI DELLA CLASSE
    public boolean vuota() {
        return costituitaDa.isEmpty();
    }
    public boolean contieneClausolaVuota() {
        Iterator<Clausola> itc = getCostituitaDa().iterator();
        while(itc.hasNext()) {
            Clausola elemc = itc.next();
            if (elemc.isEmpty())
                return true;
        }
        return false;
```

```
}
public Set<Variabile> alfabeto() {
    HashSet<Variabile> result = new HashSet();
    Iterator<Clausola> itc = getCostituitaDa().iterator();
    while(itc.hasNext()) {
        Clausola elemc = itc.next();
        Iterator<Letterale> itl = elemc.getContiene().iterator();
        while(itl.hasNext()) {
            Letterale eleml = itl.next();
            result.add(eleml.getVariabile());
        }
    }
    return result;
}
public FormulaCNF semplifica(Assegnazione a) {
    // ALGORITMO
    // 1. si fa una copia della FormulaCNF (e da ora in poi si
    //    modifica solo la copia)
    // 2. si elimina ogni clausola che non è soddisfatta da a
    // 3. si elimina da ogni clausola ogni letterale
    //    che è reso falso da a
    // 4. si restituisce la copia modificata
    // 1.
    FormulaCNF result = (FormulaCNF)this.clone();
    // 2.
    Iterator<Clausola> itc = result.getCostituitaDa().iterator();
    while(itc.hasNext()) {
        Clausola elemc = itc.next();
        Iterator<Letterale> itl = elemc.getContiene().iterator();
        while(itl.hasNext()) {
            Letterale eleml = itl.next();
            Iterator<Letterale> ita = a.getDi().iterator();
            while(ita.hasNext()) {
                Letterale elema = ita.next();
                if(elema.getVariabile().equals(eleml.getVariabile()) &&
                   (elema.getSegno() == eleml.getSegno()))
                    result.eliminaLinkCostituitaDa(elemc);
            }
        }
    }
    // 3.
    itc = result.getCostituitaDa().iterator();
    while(itc.hasNext()) {
        Clausola elemc = itc.next();
        Clausola copia = (Clausola)elemc.clone();
        boolean daAccorciare = false;
        Iterator<Letterale> itl = elemc.getContiene().iterator();
        while(itl.hasNext()) {
            Letterale eleml = itl.next();
            Iterator<Letterale> ita = a.getDi().iterator();
            while(ita.hasNext()) {
                Letterale elema = ita.next();
                if(elema.getVariabile().equals(eleml.getVariabile()) &&
                   (elema.getSegno() != eleml.getSegno())) {
                   daAccorciare = true;
                   copia.eliminaLinkContiene(eleml);
                }
            } // ita.hasNext()
        } // itl.hasNext()
        if (daAccorciare) {
            result.eliminaLinkCostituitaDa(elemc);
            result.inserisciLinkCostituitaDa(copia);
        }
    } // itc.hasNext()
    // 4.
    return result;
```

```
}
// FUNZIONI DI SERVIZIO E AUSILIARIE
public Object clone() {
    try {
        FormulaCNF d = (FormulaCNF)super.clone();
        d.costituitaDa = new HashSet<Clausola>();
        Iterator<Clausola> it = getCostituitaDa().iterator();
        while(it.hasNext()) {
            Clausola elem = it.next();
            Clausola copia = (Clausola)elem.clone();
            d.inserisciLinkCostituitaDa(copia);
        }
        return d;
    } catch (CloneNotSupportedException e) {
        // non può accadere, ma va comunque gestito
        throw new InternalError(e.toString());
    }
}
public String toString() {
    String result = new String("[");
    Iterator<Clausola> it = getCostituitaDa().iterator();
    while(it.hasNext()) {
        Clausola elem = it.next();
        result += elem + " ";
    }
    result += "]";
    return result;
}
public static FormulaCNF leggiDaFileDIMACS(String inputFile)
    throws FileNotFoundException {
    FormulaCNF result = new FormulaCNF();
    file = new FileIn(inputFile);
    skipComments();
    String str = file.readString(); // legge il token "cnf" e lo ignora
    int nVars = file.readInt(), nClauses = file.readInt();
    LinkedList<Variabile> listaVar = new LinkedList<Variabile>();
    for (int i = 1; i <= nVars; i++) {
        Variabile v = new Variabile(String.valueOf(i));
        listaVar.add(v);
    }
    for (int i = 1; i <= nClauses; i++) {
        Clausola c = new Clausola();
        int var;
        do {
            // CICLO INTERNO: gestisce i letterali di una clausola
            // legge fino a "0", incluso
            var = file.readInt();
            if (var != 0) {
                Letterale l;
                if (var > 0)
                    l = new Letterale(listaVar.get(var - 1),true);
                else
                    l = new Letterale(listaVar.get(-var - 1),false);
                c.inserisciLinkContiene(l);
            }
        } // fine ciclo INTERNO
        while (var != 0);
        result.inserisciLinkCostituitaDa(c);
    } // fine ciclo esterno
    return result;
}
private static void skipComments() {
    // legge fino a "p" della "linea del problema", incluso
    char ch = file.readChar();
    if (ch == 'c') {
        file.flush();
```

```
            skipComments();
        }
    }
}
```

```
// File Sat/Solutori.java
import java.util.*;

public final class Solutori {
    // CAMPI DATI
    static boolean DEBUG = false;
    // OPERAZIONI DELLA CLASSE
    public static String alfabeto2String(Set<Variabile> alfabeto) {
        String result = new String("(");
        Iterator<Variabile> it = alfabeto.iterator();
        while(it.hasNext()) {
            Variabile elem = it.next();
            result += elem.getNome() + " ";
        }
        result += ")";
        return result;
    }
    public static Assegnazione risolviConBacktracking(FormulaCNF phi) {
        // restituisce un'Assegnazione che è un modello se phi è
        // soddisfacibile, null altrimenti
        Assegnazione alpha = new Assegnazione();
        if (BT(phi,alpha))
            return alpha;
        else
            return null;
    }
    public static boolean BT(FormulaCNF phi, Assegnazione alpha) {
        // restituisce true e alpha diventa un'Assegnazione che è un modello se
        // phi è soddisfacibile, restituisce false e alpha non è significativa,
        // altrimenti
        FormulaCNF semplificazione = phi.semplifica(alpha);
        if (DEBUG)
            System.out.println("Formula corrente dopo la semplificazione = "
                               + semplificazione);
        if (semplificazione.vuota()) {
            if (DEBUG)
                System.out.println("E' vuota --> SAT");
            return true;
        }
        if (semplificazione.contieneClausolaVuota()) {
            if (DEBUG)
                System.out.println("Contiene clausola vuota --> UNSAT");
            return false;
        }
        Set alfabeto = semplificazione.alfabeto();
        if (DEBUG)
            System.out.println("Alfabeto corrente = " +
                               alfabeto2String(alfabeto));
        if (alfabeto.isEmpty()) {
        // non dovrebbe mai succedere che alfabeto.isEmpty()
            System.out.println("Alfabeto vuoto!");
            System.exit(-1);
        }
        Iterator<Set> its = alfabeto.iterator();
        Variabile var = (Variabile)its.next();
        Letterale p = new Letterale(var,true);
        alpha.inserisciLinkDi(p);
        if (DEBUG) {
            System.out.println("Letterale scelto per il tentativo = " + p);
            System.out.println("Tento assegnazione POS:" + p);
            System.out.println("Assegnazione corrente = " + alpha);
        }
```

```
        if (BT(phi,alpha))
            return true;
        else {
            // elimina p da alpha
            if (!alpha.eliminaLinkDi(p))
                System.out.println("Eliminazione di " + p + " fallita!");
            p = new Letterale(var,false);
            alpha.inserisciLinkDi(p);
            if (DEBUG) {
                System.out.println("Tento assegnazione NEG:" + p);
                System.out.println("Assegnazione corrente = " + alpha);
            }
            boolean altroTentativo = BT(phi,alpha);
            // elimina p da alpha
            alpha.eliminaLinkDi(p);
            return altroTentativo;
        }
    }
    private Solutori() { };
}
```

3.5 Esercitazione pratica

Lo scopo di questa esercitazione è la verifica di proprietà di alcune formule proposizionali utilizzando appositi programmi.

I programmi da usare sono:

- `javaSAT`, illustrato al paragrafo 3.4.3,
- `zchaff`, disponibile alla pagina `www.princeton.edu/~chaff/zchaff.html`,
- `satz`, disponibile alla pagina `www.laria.u-picardie.fr/~cli/EnglishPage.html`,
- `BG-Walksat`, disponibile alla pagina `www.cse.wustl.edu/~zhang/projects/backboneGuidedSearch/bgwalksat/`.

Tutti i programmi leggono formule CNF in formato DIMACS. Il primo, con l'opzione "`-d`" come in "`java javaSAT -d prova.cnf`", permette di ispezionare lo stato dell'assegnazione e della formula semplificata nei vari nodi dell'albero del backtracking. Il secondo e il terzo sono basati sull'algoritmo `DPLL`, mentre il quarto è incompleto e basato su tecniche di ricerca locale.

3.5.1 Prima parte

Prendere in considerazione le formule in CNF degli esempi 3.4.1 e 3.4.5 e dell'esercizio 3.7. Per ogni formula f vogliamo:

- sapere se è f soddisfacibile, e in caso affermativo vogliamo un suo modello;
- conoscere l'insieme $\{l \mid f \models l\}$ dei letterali positivi e negativi che sono logicamente implicati da f (allo scopo, si utilizzino le nozioni dell'esercizio 3.6).

Inoltre, vogliamo verificare che sia possibile risolvere il problema della validazione di modello (cfr. ad es., esempio 3.3.6) usando un solutore SAT e la proposizione 3.3.14.

3.5.2 Seconda parte

Il famoso *problema delle N regine* è il seguente: dato un intero positivo N si determini un posizionamento di N regine in una scacchiera $N \times N$ tale che nessuna regina minacci qualche altra regina, considerando che una regina minaccia un altro pezzo quando questo pezzo è collocato sulla stessa riga, sulla stessa colonna o su una stessa diagonale della regina stessa. La disposizione mostrata nella figura 3.4 è una soluzione dell'istanza 4.

È possibile modellare il problema delle N regine mediante un'istanza del problema SAT. In particolare, per l'istanza N progettiamo una formula in CNF $queens_N$ con le seguenti caratteristiche:

- ha un numero di variabili proposizionali distinte pari a N^2; denoteremo $\mathbf{LP}$ con $\{q_{i,j} \mid 1 \leq i, j \leq N\}$;

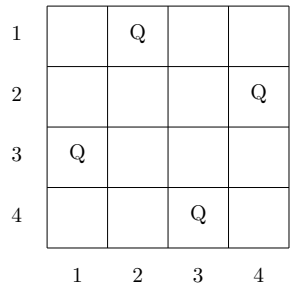

Figura 3.4 Una soluzione del problema delle N regine per l'istanza con dimensione della scacchiera pari a 4

- il significato intuitivo della lettera proposizionale $q_{i,j}$ è: c'è una regina nella riga i e nella colonna j della scacchiera;
- ha un insieme di clausole che esprimono i vincoli del problema; in particolare, ci saranno clausole che esprimono:
  - la presenza di almeno una regina per riga; con riferimento all'istanza $N = 4$ ad esempio, poiché ci deve essere almeno una regina nella prima riga, ci sarà la clausola

    $$q_{1,1} \vee q_{1,2} \vee q_{1,3} \vee q_{1,4},$$

    insieme ad altre tre clausole analoghe, ognuna per ogni altra riga;
  - la presenza di al più una regina per riga; con riferimento all'istanza $N = 4$ ad esempio, poiché ci deve essere al più una regina nella prima riga, ci saranno le clausole:

    $$(\neg q_{1,1} \vee \neg q_{1,2}),\ (\neg q_{1,1} \vee \neg q_{1,3}),\ (\neg q_{1,1} \vee \neg q_{1,4}),$$
    $$(\neg q_{1,2} \vee \neg q_{1,3}),\ (\neg q_{1,2} \vee \neg q_{1,4}),$$
    $$(\neg q_{1,3} \vee \neg q_{1,4});$$

    la prima di queste clausole (che può anche essere scritta come $q_{1,1} \rightarrow \neg q_{1,2}$) afferma che se c'è una regina nella prima riga e nelle prima colonna, allora non può esserci una regina nella prima riga e nella seconda colonna; per ogni riga abbiamo quindi bisogno di sei clausole;
  - la presenza di al più una regina per colonna; con riferimento all'istanza $N = 4$ ad esempio, poiché ci deve essere al più una regina nella prima colonna, ci saranno le clausole:

    $$(\neg q_{1,1} \vee \neg q_{2,1}),\ (\neg q_{1,1} \vee \neg q_{3,1}),\ (\neg q_{1,1} \vee \neg q_{4,1}),$$
    $$(\neg q_{2,1} \vee \neg q_{3,1}),\ (\neg q_{2,1} \vee \neg q_{4,1}),$$
    $$(\neg q_{3,1} \vee \neg q_{4,1});$$

    anche in questo caso, abbiamo bisogno di replicare le clausole per la seconda, la terza e la quarta colonna;
  - la presenza di al più una regina per linea parallela alle diagonali; con riferimento all'istanza $N = 4$ ad esempio, poiché ci deve essere al più una regina nelle celle attraversate dalla linea tratteggiata in figura 3.5, ci saranno le clausole:

    $$(\neg q_{4,2} \vee \neg q_{3,3}),\ (\neg q_{4,2} \vee \neg q_{2,4}),$$
    $$(\neg q_{3,3} \vee \neg q_{2,4});$$

    abbiamo bisogno di replicare queste clausole per ogni linea parallela alle diagonali.

La formula CNF relativa all'istanza $N = 3$, generata automaticamente nel formato DIMACS dal programma NP-SPEC che sarà illustrato nel Capitolo **??**, è la seguente:

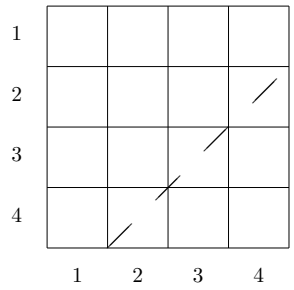

Figura 3.5 Vincoli relativi alle linee diagonali

```
c Generated by spec2sat V.1.1 May 2005
c generation time: Mon Mar 27 17:55:04 2006
c command line: spec2sat -o queens3.cnf queens.nps
c in 0.047 seconds (Translation CPU time)
c DICTIONARY:
c spec2sat dic queens(1,1) 1
c spec2sat dic queens(1,2) 2
c spec2sat dic queens(1,3) 4
c spec2sat dic queens(2,1) 3
c spec2sat dic queens(2,2) 6
c spec2sat dic queens(2,3) 7
c spec2sat dic queens(3,1) 5
c spec2sat dic queens(3,2) 8
c spec2sat dic queens(3,3) 9
c spec2sat: dictionary end
p cnf 9 31
-1 -2 0
-1 -3 0
-1 -4 0
-1 -5 0
-2 -4 0
-3 -5 0
1 2 4 0
-3 -6 0
-2 -6 0
-3 -7 0
-2 -8 0
-6 -7 0
-6 -8 0
3 6 7 0
-5 -8 0
-4 -7 0
-5 -9 0
-4 -9 0
-8 -9 0
-7 -9 0
5 8 9 0
-3 -2 0
-6 -1 0
-6 -4 0
-7 -2 0
-5 -4 0
-5 -6 0
-8 -3 0
-8 -7 0
-9 -1 0
-9 -6 0
```

Esercizio 3.9

2	3	1	4
3	4	2	1
4	1	3	2
1	2	4	3

Figura 3.6 Un quadrato latino di ordine 4

2			4
	4		1
4		3	
1			

Figura 3.7 Esiste un completamento che costituisce un quadrato latino

1. Verificare mediante un solutore per SAT che la formula in CNF appena mostrata è insoddisfacibile (è noto che il problema ammette soluzione se e solo se $N > 3$).

2. Scrivere la formula in CNF relativa all'istanza $N = 4$ nel formato DIMACS e verificare mediante un solutore per SAT che è soddisfacibile. Utilizzare il modello generato per ottenere una disposizione delle regine sulla scacchiera.

○

3.5.3 Terza parte

La metodologia mostrata nel paragrafo precedente può essere facilmente adattata ad altri problemi combinatori. Un semplice esempio è quello del *quadrato latino* (Eulero 1783): un quadrato latino di ordine n è una matrice $n \times n$ di n simboli tale che ogni simbolo occorre esattamente una volta in ogni riga e in ogni colonna della matrice. La disposizione mostrata nella figura 3.6 è una soluzione per $n = 4$.

Esercizio 3.10

1. Dato un intero n, progettare una formula in CNF i cui modelli rappresentino soluzioni del problema del quadrato latino.

2. Scrivere la formula in CNF per $n = 3$ in formato DIMACS, e trovarne un modello utilizzando un solutore per SAT.

3. Considerare il problema del *completamento del quadrato latino*, che consiste nell'avere anche una assegnazione parziale dei simboli alle celle della scacchiera, e chiedere se esista un completamento dell'assegnazione che formi un quadrato latino (cfr. figura 3.7 e 3.8). Risolvere i punti 1 e 2 di questo esercizio per il problema del completamento del quadrato latino.

○

3			4
			1
		2	
	2		

Figura 3.8 Non esiste un completamento che costituisce un quadrato latino

3.6 Nota bibliografica

L'algoritmo DPLL è descritto in [15, 14]. Alcune euristiche per la scelta del letterale di branching sono descritte in [24]. Per una rassegna aggiornata sullo stato dell'arte dei solutori SAT si rimanda a [31]. Il *crossover point* delle formule CNF è descritto in [39]. Un algoritmo incompleto (WALKSAT) basato su ricerca locale randomizzata è descritto in [38]. I siti `satlib.org` e `satlive.org` contengono vari riferimenti a solutori, istanze e articoli. Per la rilevanza del problema della soddisfacibilità di una formula proposizionale nell'ambito dell'informatica, si rimanda al testo [19]. Il sito `www.csplib.org` contiene la descrizione del problema del righello di Golomb (cfr. esercizio 3.31) e di molti altri problemi combinatori.

3.7 Esercizi proposti

Esercizio 3.11 Inserire le parentesi nelle seguenti formule in maniera da evidenziarne la struttura.

- $\neg p \equiv q \rightarrow r \vee s$
- $p \vee q \wedge s \wedge \neg p \equiv q$
- $q \rightarrow r \vee s \wedge t$
- $p \vee q \wedge s \wedge \neg p \equiv q \rightarrow r \vee s \wedge t$

Esercizio 3.12 Si consideri la seguente formula proposizionale α: $(\neg a \vee b) \wedge \neg(b \wedge \neg c)$. Dire se la formula α è soddisfacibile e, in caso positivo, fornirne un modello.

Esercizio 3.13 Dire se la formula α dell'esercizio precedente implica logicamente la formula $(\neg a \vee c)$.

Esercizio 3.14 Scrivere una formula proposizionale α tale che sia α che $\neg\alpha$ sono soddisfacibili. Mostrare almeno un modello per entrambe le formule.

Esercizio 3.15 Per ognuna delle formule seguenti, dire se è soddisfacibile, insoddisfacibile, valida.

(1)	$a \vee b$	(6)	$(a \rightarrow b) \equiv (\neg a \vee b)$
(2)	$(a \wedge c) \vee (\neg c)$	(7)	$a \equiv \neg a$
(3)	$a \wedge b \rightarrow \neg c$	(8)	$a \equiv \neg\neg a$
(4)	$c \equiv c$	(9)	$a \rightarrow \neg a$
(5)	$c \rightarrow c$	(10)	$(a \rightarrow b \rightarrow c) \equiv (a \wedge b \rightarrow c)$

Esercizio 3.16 Sia dato un insieme finito di formule proposizionali $\Sigma = \{\phi_1, \ldots, \phi_n\}$. Dimostrare che l'insieme dei modelli di Σ coincide con l'insieme dei modelli della formula proposizionale $\phi_1 \wedge \cdots \wedge \phi_n$.

Sia ψ un'altra formula proposizionale. Dimostrare che vale $\Sigma \models \psi$ se e solo se vale $\phi_1 \wedge \cdots \wedge \phi_n \models \psi$, se e solo se $\phi_1 \wedge \cdots \wedge \phi_n \wedge \neg\psi$ è insoddisfacibile.

Esercizio 3.17 [Soluzione a pagina 36] Si considerino le seguenti formule della logica proposizionale:

$\gamma_1 = (a_1 \wedge a_2 \wedge a_3) \rightarrow a_4$

$\gamma_2 = \neg a_4$

$\gamma_3 = a_5 \rightarrow a_2$

$\gamma_4 = \quad a_5$

$\gamma_5 = \quad a_1 \vee \neg a_2$

Si dimostri che l'insieme di formule $\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ non implica logicamente γ_5.

Esercizio 3.18 [Soluzione a pagina 36] Data una formula ψ contenente n lettere proposizionali distinte $l_1, \ldots, l_n$, la *relazione verofunzionale* espressa da ψ è la funzione del tipo

$$f_\psi : \{\mathsf{T}, \mathsf{F}\}^n \longrightarrow \{\mathsf{T}, \mathsf{F}\},$$

definita nel seguente modo: per ogni interpretazione I delle lettere $l_1, \ldots, l_n$,

$$f_\psi(I(l_1), \ldots, I(l_n)) = \mathsf{eval}^I(\psi).$$

Ad esempio, la relazione verofunzionale f espressa da $a \wedge b$ è definita come: $f(\mathsf{F}, \mathsf{F}) = \mathsf{F}$, $f(\mathsf{F}, \mathsf{T}) = \mathsf{F}$, $f(\mathsf{T}, \mathsf{F}) = \mathsf{F}$, e $f(\mathsf{T}, \mathsf{T}) = \mathsf{T}$.

Si definisca la relazione verofunzionale della formula (2) dell'esempio 3.3.4.

Esercizio 3.19 Dimostrare che per ogni funzione

$$H : \{\mathsf{T}, \mathsf{F}\}^n \longrightarrow \{\mathsf{T}, \mathsf{F}\} \quad (n \geq 1)$$

esiste una formula ψ nella quale occorrono n lettere proposizionali distinte che rappresenta la relazione verofunzionale espressa da H.

Esercizio 3.20 Sia data la funzione $H : \{\mathsf{T}, \mathsf{F}\}^3 \longrightarrow \{\mathsf{T}, \mathsf{F}\}$ rappresentata dalla seguente tabella.

p	q	s	$H(p,q,s)$	p	q	s	$H(p,q,s)$
F	F	F	T	T	F	F	F
F	F	T	F	T	F	T	F
F	T	F	T	T	T	F	F
F	T	T	T	T	T	T	T

Si scriva una formula proposizionale con tre lettere proposizionali che rappresenta la relazione verofunzionale espressa da H.

Esercizio 3.21 Sia data una formula proposizionale α in cui occorrono n lettere proposizionali distinte. Quante sono le interpretazioni di α?

Esercizio 3.22 Scrivere un algoritmo che, data in ingresso una stringa contenente solamente i caratteri '`(`', '`)`', '`&`', '`|`', '`!`', '`a`', ..., '`z`', determini se la formula rappresentata dalla stringa sia soddisfacibile oppure no, (i caratteri '`&`', '`|`', '`!`' stanno per $\wedge$, $\vee$, $\neg$, rispettivamente; si assuma che la stringa è ben formata secondo le regole del paragrafo 3.2).

Esercizio 3.23 Con riferimento all'esempio 3.4.4 e all'assegnazione β definita come $\alpha\neg b$ (estensione di α), calcolare $f_1|\beta$, $f_2|\beta$ e $f_3|\beta$.

Esercizio 3.24 Progettare una versione non ricorsiva dell'algoritmo `BT` visto nel paragrafo 3.4.2.1.

Esercizio 3.25 Dimostrare che è possibile trasformare in tempo polinomiale qualsiasi formula proposizionale ϕ in una formula in CNF ψ tale che ψ è soddisfacibile se e solo ϕ è soddisfacibile (cfr. paragrafo 3.4.2).

Esercizio 3.26 Dimostrare che se una formula in CNF è soddisfacibile, allora ha un modello in cui tutti i letterali delle clausole unitarie sono veri (cfr. paragrafo 3.4.2.2).

Esercizio 3.27 Dimostrare che se una formula in CNF è soddisfacibile, allora ha un modello in cui tutti i letterali puri sono veri (cfr. paragrafo 3.4.2.2).

Esercizio 3.28 Progettare una versione non ricorsiva dell'algoritmo `DPLL` visto nel paragrafo 3.4.2.2.

Esercizio 3.29 Realizzare in JAVA l'algoritmo `DPLL`.

Esercizio 3.30 Un algoritmo `A` per il problema SAT che, per ogni formula in CNF ϕ, quando `A(`ϕ`)` restituisce `INSODDISFACIBILE`, allora ϕ è insoddisfacibile si dice corretto, completo, entrambi, o nessuno dei due?

Esercizio 3.31 Il problema del *righello di Golomb* consiste, dato un intero positivo m, nel trovare un insieme di m interi $0 = a_1 < a_2 < \cdots < a_m$ tali che le $m(m-1)/2$ differenze $a_i - a_j$ $(1 \leq i < j \leq m)$ sono distinte. I numeri $a_1, \ldots a_m$ sono detti *marche* e a_m è detta la *lunghezza* del righello.

Utilizzare la metodologia illustrata al Paragrafo 3.5.2 per risolvere il problema del righello di Golomb per m piccolo (ad es. 3 o 4). La seguente tabella riporta alcuni valori minimi di a_m, in funzione di m, tali che il problema ha soluzione. Per completezza, notiamo che il problema del righello di Golomb può essere anche formulato come problema di ottimizzazione, ma che questa formulazione non si presta ad una traduzione in un'istanza del problema SAT.

m	a_m
2	1
3	3
4	6
5	11
6	17
7	25

Esercizio 3.32 Scrivere una funzione JAVA che, data in ingresso una stringa contenente solamente i caratteri '`(`', '`)`', '`&`', '`|`', '`!`', '`a`', ..., '`z`', restituisca `1` se la stringa rappresenta una formula proposizionale ben formata secondo le regole del paragrafo 3.2, restituisca `0` altrimenti (i caratteri '`&`', '`|`', '`!`' stanno per $\wedge$, $\vee$, $\neg$, rispettivamente).

3.8 Soluzione di alcuni esercizi

Soluzione esercizio 3.2. I commenti immediatamente successivi alla definizione 3.3.3 chiariscono che $\rightarrow$ e $\equiv$ sono ridondanti.

Sia ϕ una formula del tipo $\alpha \vee \beta$, dove α e β sono formule. Consideriamo la formula ϕ' definita come $\neg(\neg\alpha \wedge \neg\beta)$. Abbiamo che $\mathsf{eval}^I(\phi') = \mathsf{T}$ se $\mathsf{eval}^I(\alpha) = \mathsf{T}$ oppure $\mathsf{eval}^I(\beta) = \mathsf{T}$, e $\mathsf{eval}^I(\phi') = \mathsf{F}$ altrimenti. In altre parole, $\mathsf{eval}^I(\phi') = \mathsf{eval}^I(\phi)$. Seguendo la struttura induttiva delle formule α e β potremmo provare che ϕ e ϕ' sono equivalenti. Questa considerazione chiarisce che $\vee$ è ridondante.

La ridondanza di $\wedge$ può essere chiarita prendendo in considerazione una formula ϕ del tipo $\alpha \wedge \beta$, dove α e β sono formule e considerando la formula ϕ' definita come $\neg(\neg\alpha \vee \neg\beta)$. Abbiamo che $\mathsf{eval}^I(\phi') = \mathsf{T}$ se $\mathsf{eval}^I(\alpha) = \mathsf{eval}^I(\beta) = \mathsf{T}$, e $\mathsf{eval}^I(\phi') = \mathsf{F}$ altrimenti. In altre parole, $\mathsf{eval}^I(\phi') = \mathsf{eval}^I(\phi)$. ∘

Soluzione esercizio 3.4.

(1)	p	è sodd.	(7)	$p \rightarrow q \equiv s$	è sodd.
(2)	$(p \wedge q) \vee \neg s$	è sodd.	(8)	$(p \rightarrow q) \equiv s$	è sodd.
(3)	$p \wedge q \vee \neg s$	è sodd.	(9)	$p \rightarrow (q \equiv s)$	è sodd.
(4)	$p \wedge (q \vee \neg s)$	è sodd.	(10)	$p \wedge q \wedge s$	è sodd.
(5)	$s \wedge (\neg s)$	è insodd.	(11)	$(p \wedge q) \wedge s$	è sodd.
(6)	$(p \wedge \neg p) \rightarrow q$	è valida	(12)	$p \wedge (q \wedge s)$	è sodd.

La formula (6), in quanto valida, è anche soddisfacibile. Nessuna altra delle formule (1)-(12) è valida. ∘

Soluzione esercizio 3.17. Si può dimostrare considerando la seguente interpretazione I delle variabili proposizionali a_1, a_2, a_3, a_4, a_5:

- $I(a_1) = \mathsf{F}$
- $I(a_2) = \mathsf{T}$
- $I(a_3) =$ qualsiasi (T o F)
- $I(a_4) = \mathsf{F}$
- $I(a_5) = \mathsf{T}$

L'interpretazione I rende vere le formule γ_1, γ_2, γ_3, γ_4 ma non rende vera la formula γ_5. ∘

Soluzione esercizio 3.18. La relazione verofunzionale cercata è una funzione del tipo

$$\{\mathsf{T},\mathsf{F}\}^3 \longrightarrow \{\mathsf{T},\mathsf{F}\}$$

definita dalla seguente tabella

p	q	s	$f(p,q,s)$	p	q	s	$f(p,q,s)$
F	F	F	T	T	F	F	T
F	F	T	F	T	F	T	F
F	T	F	F	T	T	F	F
F	T	T	T	T	T	T	T

∘