Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

IMPLEMENTAZIONE IN JAVA DEL METODO DI RISOLUZIONE PER LA LOGICA CLASSICA, ED ESTENSIONE A LOGICHE MODALI

Relatore: Prof. Camillo FIORENTINI

Tesi di:
Nicolò IACCARINO
Matricola: 903870

Anno Accademico 2023-2024

Dediche

dedicato a . . .

Prefazione

...

Indice

Dediche **ii**

Prefazione **iii**

1 Introduzione **1**
- 1.1 Panoramica generale 1
- 1.2 Logica proposizionale 1
  - 1.2.1 Formule atomiche 1
  - 1.2.2 Formule composte 2
  - 1.2.3 Soddisfacibilità di una formula 2
  - 1.2.4 Tautologie e contraddizioni 2

2 Il metodo di risoluzione per la logica classica **3**
- 2.1 Forma normale congiuntiva 3
  - 2.1.1 Letterali 3
  - 2.1.2 Clausole 4
  - 2.1.3 Insiemi di clausole 4
- 2.2 Regola di risoluzione 5
  - 2.2.1 Clausola risolvente 5
- 2.3 Funzionamento del metodo di risoluzione 5
  - 2.3.1 Gestione delle clausole tautologiche 6
  - 2.3.2 Esempio pratico del metodo di risoluzione 6

3 Implementazione in Java del metodo di risoluzione **8**
- 3.1 Struttura del progetto Java 8
- 3.2 Strutture dati per la CNF 9
  - 3.2.1 La classe astratta Literal e le classi Atom e NegAtom 10
  - 3.2.2 La classe Clause 10
  - 3.2.3 La classe ClauseSet 11
- 3.3 La classe Resolution 11
  - 3.3.1 Il metodo “isSatisfiable” 12

3.3.2 Memorizzazione delle coppie di clausole visitate 14
3.3.3 Implementazione della regola di risoluzione 16
3.3.4 Gestione degli step 16

4 Implementazione in Java di formule generiche 19
4.1 La classe astratta Formula 19
4.2 La classe AtomicFormula 19
4.3 L'enumerazione Connective 20
4.4 La classe CompoundFormula 20
4.5 Clausificazione di una formula 21
4.5.1 Esempio di clausificazione 22
4.6 Parsing di formule 23
4.6.1 Utilizzo del parser ANTLR4 23

5 Logica modale ed estensione del metodo di risoluzione 27

6 Applicazioni pratiche del metodo di risoluzione 28
6.1 Verificare la validità di una formula 28
6.2 Dimostrazione di conseguenze logiche 28
6.3 Dimostrazione di equivalenze logiche 28

7 Testing 29
7.1 Struttura del file di test 29

8 Conclusioni 30

Ringraziamenti 32

Capitolo 1

Introduzione

1.1 Panoramica generale

Questo elaborato riguarda l'ambito della logica matematica, concentrandosi sul metodo di risoluzione per la logica classica, ovvero la logica proposizionale. Nel capitolo 5 verrà trattato il metodo di risoluzione per le logiche modali, che rappresentano un'estensione di quella proposizionale. Di seguito viene presentata una breve introduzione della logica proposizionale, utile a capire il funzionamento del metodo di risoluzione, trattato nel capitolo 2.

1.2 Logica proposizionale

La logica proposizionale è un sistema formale per la rappresentazione e l'analisi del ragionamento, essa si basa su proposizioni. La sua sintassi comprende formule atomiche e formule composte. La semantica della logica proposizionale stabilisce come valutare le formule, associando loro valori di verità in base a interpretazioni che specificano lo stato di verità di ogni proposizione atomica.

1.2.1 Formule atomiche

Le formule atomiche rappresentano il caso più semplice di formula, nelle quali non vengono usati gli operatori logici. Una formula atomica può essere scritta come una lettera proposizionale, ad esempio "p"; essa viene valutata come vera ($true$) o falsa ($false$) in base all'interpretazione considerata.

1.2.2 Formule composte

Le formule composte sono costruite mediante operatori logici (detti anche "connettivi") a partire dalle formule atomiche. I connettivi della logica proposizionale sono:

- $\neg$ è la negazione logica (**not**)
- $\wedge$ è la congiunzione logica (**and**)
- $\vee$ è la disgiunzione logica (**or**)
- $\implies$ è l'implicazione logica (**implica**)
- $\iff$ è la doppia implicazione (**se e solo se**)

Consideriamo il seguente esempio di formula composta:

$$\neg p \wedge q$$

essa è ottenuta a partire dalla atomica "p" che viene negata tramite la negazione logica ($\neg$), e successivamente messa in congiunzione logica ($\wedge$) con l'atomica "q". Si possono creare formule composte più complicate, in tal caso si usano le parentesi tonde per evitare ambiguità tra i connettivi.

L'interpretazione di una formula composta dipende dai connettivi usati e dall'interpretazione delle sue atomiche.

1.2.3 Soddisfacibilità di una formula

Una formula F si dice **soddisfacibile** se e solo se ammette almeno un'interpretazione che la rende vera, ossia esiste almeno un assegnamento di valori di verità alle atomiche che rende vera F. Se tutte le interpretazioni possibili la rendono falsa, allora F si dice **insoddisfacibile**.

1.2.4 Tautologie e contraddizioni

Una formula è una **tautologia** se e solo se ogni interpretazione della formula la rende vera. Quindi una tautologia è una formula sempre vera, che come vedremo, può essere ridondante in alcuni casi. Ad esempio, la formula "$\mathbf{p} \vee \neg\mathbf{p}$" è il caso più semplice di tautologia.

Una formula è una **contraddizione** se e solo se è insoddisfacibile, ovvero tutte le sue interpretazioni la rendono falsa. Ad esempio, la formula "$\mathbf{p} \wedge \neg\mathbf{p}$" è una contraddizione. È bene notare che se si applica la negazione logica ad una contraddizione si ottiene una tautologia, e viceversa.

Capitolo 2

Il metodo di risoluzione per la logica classica

Il metodo di risoluzione è un sistema di calcolo logico per inferire la soddisfacibilità di una formula, esso ha avuto un impatto significativo in vari settori della matematica, dell'informatica e dell'ingegneria. È stato utilizzato per dimostrare teoremi importanti, risolvere problemi pratici e sviluppare algoritmi per l'intelligenza artificiale, la verifica formale e la progettazione dei circuiti.

Questo metodo consiste nell'applicare più volte una sola regola: la regola di risoluzione. Il difetto è che essa opera soltanto su formule espresse in **Forma normale congiuntiva (CNF)**, ovvero come congiunzione di clausole. Prima di vedere il metodo di risoluzione è bene capire in che cosa consiste la *CNF*.

2.1 Forma normale congiuntiva

2.1.1 Letterali

Un letterale è una formula atomica, oppure la sua negazione. Ad esempio "a" e "$\neg b$" sono dei letterali, essi rappresentano l'elemento fondamentale delle clausole.

Opposto di un letterale

Dato un letterale L, il suo *opposto* $\overline{L}$ è la sua negazione, ad esempio: se $L = p$ allora $\overline{L} = \neg p$ e se $L = \neg p$ allora $\overline{L} = p$.

2.1.2 Clausole

Una clausola è una disgiunzione di letterali. Consideriamo il seguente esempio:

$$a \vee \neg b \vee \neg c \vee d$$

questa è una clausola formata dai letterali "a", "$\neg b$", "$\neg c$", "d".

Una clausola può anche essere rappresentata in notazione insiemistica, in questo modo diventa un insieme di letterali. La clausola dell'esempio precedente diventa:

$$\{a, \neg b, \neg c, d\}$$

che come si può notare, il simbolo di disgiunzione logica non è più presente, ma è sottinteso. D'ora in avanti useremo sempre la notazione insiemistica per le clausole.

L'interpretazione delle clausole è semplice: una clausola è vera se e solo se almeno un letterale appartenente ad essa è vero in una data interpretazione (a causa della disgiunzione).

Clausola tautologica

Una clausola è una tautologia se e solo se contiene un letterale ed il suo opposto. Ad esempio: "$\{a, b, c, \neg a\}$" è una tautologia, perché contiene il letterale "a" ed il suo opposto. Infatti questo tipo di clausola risulta essere sempre vera, qualunque sia l'interpretazione dei suoi letterali (si veda la sezione 1.2.4).

Clausola vuota

è importante notare che una clausola può non contenere alcun letterale, in tal caso si parla di **clausola vuota**. La clausola vuota rappresenta la **contraddizione** (si veda la sezione 1.2.4) e si può indicare con "$\{\}$".

2.1.3 Insiemi di clausole

La *CNF* consiste in una congiunzione di clausole. Seguendo lo stesso approccio per le clausole, la *CNF* può essere rappresentata anch'essa in notazione insiemistica come **insieme di clausole**, quindi ad esempio la seguente *CNF*:

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee c)$$

diventa

$$\{ \{a, b\}, \{\neg a, c\}, \{\neg b, c\} \}$$

anche in questo caso la congiunzione è sottintesa. Possiamo quindi considerare la *CNF* e gli insiemi di clausole come lo stesso oggetto semantico.

Soddisfacibilità di un insieme di clausole

Un insieme di clausole S è **soddisfacibile** se e solo se esiste almeno un'interpretazione che rende vere **tutte** le clausole appartenenti a S (a causa della congiunzione).

2.2 Regola di risoluzione

La regola di risoluzione è una regola di inferenza che, a partire da due clausole di premessa, genera una clausola conclusione (detta **risolvente**). Per poter applicare questa regola su due clausole $C1$ e $C2$, è necessario che esista un letterale $L \in C1$ ed il suo opposto $\overline{L} \in C2$. Se questo non dovesse capitare, allora la regola di risoluzione non è applicabile su $C1$ e $C2$.

2.2.1 Clausola risolvente

Per poter ottenere la clausola risolvente si rimuove il letterale L dalla clausola $C1$ ed il suo opposto $\overline{L}$ dalla clausola $C2$, infine si uniscono le due clausole (con l'operazione di **unione** insiemistica). Vediamo un esempio:

$$C1 = \{a, b\}$$

$$C2 = \{\neg b, c, d\}$$

la risolvente R è:

$$R = \{a, c, d\}$$

ottenuta cancellando il letterale "b" (da $C1$) ed il suo opposto "$\neg b$" (da $C2$), e facendo l'unione.

2.3 Funzionamento del metodo di risoluzione

Il metodo di risoluzione opera su un insieme S di clausole, e applica (dove possibile) la regola di risoluzione su tutte le coppie di clausole appartenenti a S con lo scopo di trovare la contraddizione (clausola vuota). Ogni volta che il metodo applica la regola di risoluzione, la risolvente viene aggiunta a S, la quale potrà poi essere considerata come clausola di premessa per una successiva applicazione della regola. Se si riesce a trovare la clausola vuota, allora il metodo prova che S è **insoddisfacibile**; se invece, dopo aver applicato la regola su tutte le possibili coppie di clausole non trova la contraddizione, allora il metodo prova che S è **soddisfacibile**.

2.3.1 Gestione delle clausole tautologiche

Nella *CNF* le clausole tautologiche rappresentano una ridondanza, perché essendo la *CNF* una congiunzione di clausole, esse non indicano alcun valore informativo. Questo si traduce col fatto che nel metodo di risoluzione si possono ignorare questo tipo di clausole, rendendo più semplice l'esecuzione del metodo anche da un punto di vista computazionale (come vedremo nel capitolo 3).

Un altro aspetto da tenere in considerazione è che applicando la regola di risoluzione è possibile che la risolvente sia una tautologia, in tal caso la risolvente viene scartata (non viene aggiunta a S). Vediamo un esempio di clausole $C1$ e $C2$ che generano una risolvente R tautologica:

$$C1 = \{a, \neg b\}$$

$$C2 = \{\neg a, b\}$$

$$R = \{\neg b, b\}$$

per ottenere R è stato rimosso il letterale a ed il suo opposto. Si può notare che in questo caso si poteva anche considerare il letterale $\neg b \in C1$ ed il suo opposto $b \in C2$ ed ottenere $R^{'} = \{\neg a, a\}$, ma dal punto di vista del metodo di risoluzione non cambia nulla perché R e $R^{'}$ sono entrambe tautologie, e quindi scartate.

2.3.2 Esempio pratico del metodo di risoluzione

Esempio su un insieme di clausole insoddisfacibile

Consideriamo il seguente insieme di clausole S:

$$S = \{\ \{a\}, \{\neg a, b\}, \{\neg b\}\ \}$$

in questo caso la contraddizione si ricava applicando due volte la regola di risoluzione:

Step 1:

$$C1 = \{a\}$$

$$C2 = \{\neg a, b\}$$

$$R = \{b\}$$

Step 2:

$$C1 = \{b\}$$

$$C2 = \{\neg b\}$$

$$R = \{\}$$

nello *Step 2* la regola di risoluzione ha trovato la clausola vuota, questo dimostra che S è **insoddisfacibile**. Si noti che la clausola $C1$ dello *Step 2* è la risolvente R dello *Step 1*.

Esempio su un insieme di clausole soddisfacibile

Consideriamo ora il seguente insieme di clausole $S^{'}$:

$$S^{'} = \{ \ \{a\}, \{\neg a, b\}, \{\neg c\} \ \}$$

in questo caso il metodo di risoluzione applica una sola volta la regola:

$$C1 = \{a\}$$

$$C2 = \{\neg a, b\}$$

$$R = \{b\}$$

stavolta, il metodo non riesce più ad andare avanti, perché la regola non è più applicabile in nessun'altra coppia di clausole presenti in $S^{'}$, anche tenendo conto della clausola R appena generata. Questo significa che la contraddizione non può essere ricavata, e quindi $S^{'}$ è **soddisfacibile**.

Capitolo 3

Implementazione in Java del metodo di risoluzione

Questo capitolo si concentra sull'implementazione pratica del metodo di risoluzione per la logica classica in linguaggio Java. Esploreremo come tradurre i concetti teorici esaminati nei capitoli precedenti in codice eseguibile, analizzando le classi, i metodi e le strutture dati necessari per realizzare efficacemente il metodo di risoluzione. Partiremo con una panoramica generale dell'architettura del progetto Java, identificando le principali classi per rappresentare le strutture dati necessarie per il metodo di risoluzione. Successivamente, affronteremo la classe che implementa l'algoritmo di risoluzione. Nel capitolo 4 vedremo l'implementazione delle formule della logica proposizionale e la loro conversione in *CNF*.

3.1 Struttura del progetto Java

Di seguito viene mostrato l'elenco dei package contenuti nella directory *src* del progetto, e i file java contenuti in ognuno di essi:

- **literal**
  - *Literal.java*
  - *Atom.java*
  - *NegAtom.java*
- **cnf**
  - *Clause.java*
  - *ClauseSet.java*

- **resolution**
  - *Resolution.java*
  - *Step.java*
- **formula**
  - *Formula.java*
  - *AtomicFormula.java*
  - *CompoundFormula.java*
- **connective**
  - *Connective.java*
- **antlr4**
  - *FormulaExpression.g4*
  - *FormulaExpressionListener.java*
  - *FormulaExpressionBaseListener.java*
  - *FormulaExpressionLexer.java*
  - *FormulaExpressionParser.java*
  - *FormulaListenerImplementation.java*
  - *ParseFormula.java*
- **test**
  - *ResolutionTest.java*
  - (file *txt* per il test)
- *App.java*

In questo capitolo ci concentriamo sui primi tre package dell'elenco, che sono i principali per l'implementazione del metodo di risoluzione. Gli altri package contengono le classi per rappresentare le formule, eseguire il parsing delle formule, effettuare il testing, ed infine è presente file *App.java* che contiene il metodo *main* (si noti che questa classe non è contenuta in alcun package).

3.2 Strutture dati per la CNF

I package "**literal**" e "**cnf**" contengono le classi per rappresentare la CNF, fondamentali per il metodo di risoluzione.

3.2.1 La classe astratta Literal e le classi Atom e NegAtom

La classe `Literal` permette di rappresentare i letterali. Essa è una classe astratta che fornisce un'interfaccia comune per le classi `Atom` e `NegAtom`, che la estendono per rappresentare rispettivamente una variabile atomica e la negazione di essa. Questo approccio consente una gestione modulare dei letterali, facilitando l'estensione e il mantenimento del codice. La classe `Literal` contiene una stringa come campo privato che identifica il nome del letterale, e il corrispettivo metodo `getName()` che lo restituisce; inoltre, ha un metodo astratto `getOpposite()` sovrascritto dalle due classi che la estendono, che permette di restituire l'opposto del letterale sul quale viene chiamato (se chiamato su un'istanza di `Atom` restituisce l'istanza corrispondente di `NegAtom`, e viceversa). Il metodo `toString` della classe `NegAtom` usa il carattere "$\sim$" per rappresentare testualmente la negazione logica in un atomo negato (ad esempio "$\sim p$").

metodo getOpposite sovrascritto dalla classe Atom

```java
    @Override
    public Literal getOpposite() {
        return new NegAtom(this.getName());
    }
```

3.2.2 La classe Clause

Questa classe permette di rappresentare le clausole della *CNF*, tenendo conto della loro notazione insiemistica. La classe contiene il campo `literals` di tipo `Set<Literal>` che consiste nell'insieme di letterali, inoltre contiene anche il campo `index` che rappresenta un indice numerico che identifica la clausola istanziata.

All'interno della classe *Clause* sono presenti i classici metodi per gli insiemi (`add`, `remove`, `contains`, ecc.), in aggiunta al metodo `union` che permette di eseguire l'unione insiemistica con un'altra clausola specificata come parametro. Un altro importante metodo è `isTautology` che restituisce *true* se e solo se la clausola è una tautologia (si veda il codice 3.1). La classe *Clause* segue il design pattern "iterator", che permette di iterare facilmente sui letterali di una clausola tramite il ciclo *for-each* di Java.

Codice 3.1: Metodo isTautology della classe Clause

```java
    public boolean isTautology() {
        for (Literal l1 : this.literals) {
            for (Literal l2 : this.literals) {
                if (l1.equals(l2.getOpposite())) return true;
            }
        }
```

```
        return false;
    }
```

Questo codice mostrato esegue un doppio loop sulla clausola (utilizzando il campo `literals`) per verificare la presenza di un letterale ed il suo opposto all'interno di essa. Se questo dovesse capitare, allora la clausola è una tautologia e il metodo restituisce *true*; altrimenti restituisce *false* dopo aver terminato il doppio loop.

3.2.3 La classe ClauseSet

Questa classe rappresenta la *CNF* in notazione insiemistica, ovvero gli insiemi di clausole. Nella classe è presente il campo `clauses` di tipo `Set<Clause>`, che contiene le clausole dell'istanza di *ClauseSet*. Anche in questo caso ci sono i metodi per gestire gli elementi dell'insieme come nella classe *Clause*, ed il metodo `union` per fare l'unione insiemistica dell'oggetto con un'altra istanza di *ClauseSet*. Importante è il metodo `removeTautologies` che rimuove le clausole tautologiche dall'oggetto, in questo modo si tolgono le ridondanze, rendendo più semplice la *CNF* (si veda il codice 3.2). Anche la classe *ClauseSet* segue il design pattern "iterator".

Codice 3.2: Metodo "removeTautologies" della classe ClauseSet

```
    public void removeTautologies() {
        List<Clause> tautologies = new ArrayList<>();
        for (Clause c : this.clauses) {
            if (c.isTautology()) {
                tautologies.add(c);
            }
        }
        for (Clause taut : tautologies) {
            this.clauses.remove(taut);
        }
    }
```

Questo codice esegue la rimozione delle clausole tautologiche: prepara una lista `tautologies` vuota, esegue un loop sul campo `clauses` per aggiungere alla lista le tautologie, ed infine esegue un loop sulla lista `tautologies` per rimuovere le tautologie contenute in `clauses`.

3.3 La classe Resolution

La classe *Resolution* è una classe senza costruttori che contiene alcuni campi statici e metodi statici per l'implementazione del metodo di risoluzione. I campi della classe sono tre:

- `visited`: è di tipo `Map<Integer, Set<Integer>>` e consiste in una mappa che associa una clausola ad un insieme di clausole (utilizzando i loro indici). Essa memorizza tutte le clausole alle quali è stata applicata la regola di risoluzione con la clausola rappresentata dalla chiave della mappa (si veda la sottosezione 3.3.2).
- `enableSteps`: è un campo booleano che, se impostato a `true`, permette di abilitare i passaggi del metodo di risoluzione quando viene eseguito (si veda la sottosezione 3.3.4).
- `trace`: è una lista di `Step`, che memorizza tutti i passaggi del metodo di risoluzione se il campo `enableSteps` è impostato a `true` (si veda la sottosezione 3.3.4).

I metodi statici della classe sono:

- `isSatisfiable` (spiegato nella sottosezione 3.3.1)
- `getComplementaryLiteral` (spiegato nella sottosezione 3.3.1)
- `alreadyVisited` (spiegato nella sottosezione 3.3.2)
- `resolRule` (spiegato nella sottosezione 3.3.3)
- `setEnableSteps` (spiegato nella sottosezione 3.3.4)
- `printTrace` (spiegato nella sottosezione 3.3.4)

3.3.1 Il metodo "isSatisfiable"

Questo è il metodo più importante della classe *Resolution*. Esso ha come parametro in input un oggetto ClauseSet `s` e restituisce `true` se `s` è soddisfacibile, `false` altrimenti. Di seguito viene mostrato il codice. Il metodo *isSatisfiable* dopo aver controllato che `s` non sia `null` o un insieme vuoto, elimina tutte le clausole tautologiche appartenenti a `s` richiamando il metodo `removeTautologies` sull'oggetto `s`. Successivamente controlla che sia vuoto (in tal caso viene restituito `true` perché `s` è una tautologia), e in caso negativo continua l'esecuzione inizializzando i campi `visited` e `trace`; inoltre, le clausole in `s` vengono inserite nella lista di clausole `listCl` per poter consentire la modifica della lista durante il suo scorrimento (eseguito dal ciclo *for*).

A questo punto vengono eseguiti due cicli for innestati su `listCl`; il ciclo esterno itera la lista utilizzando la clausola `c1`, il ciclo interno utilizza la clausola `c2`. Il codice 3.3 mostra l'esecuzione dei due cicli.

Codice 3.3: Metodo "isSatisfiable" della classe Resolution

```java
        for (int i = 0; i < listCl.size(); i++) {
            Clause c1 = listCl.get(i);
            int index1 = c1.getIndex();
            for (int j = 0; j < listCl.size(); j++) {
                Clause c2 = listCl.get(j);
                int index2 = c2.getIndex();
                if ((i != j) && !alreadyVisited(c1, c2)) {
                    Literal complemLit =
                        getComplementaryLiteral(c1, c2);
                    if (complemLit != null) {
                        if (index1 < index2) {
                            (visited.get(index1)).add(index2);
                        } else {
                            (visited.get(index2)).add(index1);
                        }
                        Clause resolvent = resolRule(c1, c2,
                            complemLit);
                        Step step = null;
                        if (enableSteps) {
                            step = new Step(c1, c2,
                                resolvent, complemLit);
                            trace.add(step);
                        }
                        if (resolvent.isEmpty()) {
                            if (enableSteps) printTrace();
                            return false;
                        }
                        if (resolvent.isTautology()) {
                            if (enableSteps)
                                step.setTautology();
                        } else if
                            (listCl.contains(resolvent)) {
                            if (enableSteps)
                                step.setAlreadyPresent();
                        } else {
                            visited.put(resolvent.getIndex(),
                                new HashSet<>());
                            listCl.add(resolvent);
                        }
```

```
                    }
                }
            }
        }
        if (enableSteps) printTrace();
        return true;
    }
```

Una volta entrato nel secondo ciclo, il metodo esegue la regola di risoluzione (metodo `resolRule`) su `c1` e `c2`, se e solo se le due clausole non sono uguali, non sono già state visitate in precedenza, e contengono almeno un letterale complementare in comune (metodo `getComplementaryLiteral`). Una volta eseguita la regola sulle due clausole, viene controllato se la clausola `resolvent` è vuota (metodo `isEmpty`); in tal caso il metodo restituisce `false` e termina la sua esecuzione, altrimenti la continua e aggiunge la risolvente a `listCl` se e solo se `resolvent` non è tautologica e non è già presente nella lista. A questo punto si esegue la successiva iterazione del `for` interno.

Il metodo "getComplementaryLiteral"

Questo metodo controlla se nella clausola `c1` è presente un letterale `l1`, ed il suo opposto `l2` nella clausola `c2`. Se lo trova lo restituisce, altrimenti restituisce `null`. Il codice 3.4 mostra questo metodo.

Codice 3.4: Metodo "getComplementaryLiteral" della classe Resolution

```
    private static Literal getComplementaryLiteral(Clause c1, Clause c2) {
        for (Literal l1 : c1) {
            for (Literal l2 : c2) {
                if (l1.equals(l2.getOpposite())) return l1;
            }
        }
        return null;
    }
```

3.3.2 Memorizzazione delle coppie di clausole visitate

Per evitare che si esegua la regola di risoluzione più di una volta su una stessa coppia di clausole, è opportuno memorizzare la coppia sulla mappa `visited`. Essa utilizza gli indici delle clausole, ed ha come chiave un `Integer`, e come valore un insieme di interi (`Set<Integer>`). Ogni volta che viene eseguita la regola di risoluzione su `c1` e `c2`, viene aggiunto l'indice più grande alla mappa ottenuta come valore a partire dalla chiave corrispondente all'indice più piccolo. In questo modo viene gestita più semplicemente la simmetria, infatti se `c1` e `c2` sono già state visitate, allora vale la stessa cosa anche per `c2` e `c1`. Nel codice 3.3 L'inserimento degli indici nella mappa viene eseguito tra la riga 10 e la riga 14.

Esempio di struttura della mappa

Consideriamo il seguente insieme di clausole con associati gli indici ad ognuna di essa:

Clausole	$\{a, \neg b, c\}$	$\{\neg a, d\}$	$\{\neg c\}$
Indici	0	1	2

In questo caso, il metodo `isSatisfiable` richiama il metodo `resolRule` sulle clausole **0** e **1**, e crea la risolvente con indice **3**. Nella mappa viene aggiunto **1** nell'insieme corrispondente alla chiave **0** (perché $0 < 1$). Poi va avanti eseguendo altre volte la regola; le clausole che vengono aggiunte all'insieme di partenza sono le seguenti:

Clausole	$\{\neg b, c, d\}$	$\{a, \neg b\}$	$\{d, \neg b\}$
Indici	3	4	5

Nella situazione finale la mappa visited avrà la seguente struttura:

Chiavi	**Valori**
0	$\{1, 2\}$
1	$\{4\}$
2	$\{3\}$

La prima riga dice che la clausola **0** è stata visitata con le clausole **1** e **2** (infatti le clausole **3** e **4** sono state ottenute dalle coppie **0 - 1** e **0 - 2**). La seconda riga dice che la clausola **1** è stata visitata con la clausola **4** (per ottenere la clausola **5**); infine, la terza riga dice che la clausola **2** è stata visitata con la clausola **3** (in quest'ultimo caso viene generata la clausola **6** che è uguale alla clausola **5**, quindi viene scartata).

Il metodo "alreadyVisited"

Questo metodo utilizza la mappa `visited` per verificare se una coppia di clausole è gia stata visitata in precedenza, ossia la regola di risoluzione è già stata applicata su di essa. Per farlo controlla se nell'insieme ottenuto dall'indice più piccolo è presente l'indice più grande (utilizzando il metodo `contains`). Questo garantisce che l'algoritmo possa terminare. Il codice 3.5 mostra questo metodo.

Codice 3.5: Metodo "alreadyVisited" della classe Resolution

```
    private static boolean alreadyVisited(Clause c1, Clause c2) {
        int i1 = c1.getIndex();
        int i2 = c2.getIndex();
        if (i1 < i2) {
            return (visited.get(i1)).contains(i2);
        }
        return (visited.get(i2)).contains(i1);
    }
```

3.3.3 Implementazione della regola di risoluzione

La regola di risoluzione viene implementata dal metodo `resolRule`; esso prende in input le clausole `c1` e `c2`, insieme al letterale `lit`. Il metodo esegue l'unione delle due clausole (tramite il metodo `union` della classe `Clause`), e successivamente rimuove il letterale `lit` ed il suo opposto dalla clausola risultante; infine restituisce il risultato. Il codice 3.6 mostra questo metodo.

Codice 3.6: Metodo "resolRule" della classe Resolution

```
    private static Clause resolRule(Clause c1, Clause c2, Literal lit) {
        Clause result = c1.union(c2);
        result.remove(lit);
        result.remove(lit.getOpposite());
        return result;
    }
```

3.3.4 Gestione degli step

Per verificare la correttezza dell'implementazione del metodo di risoluzione è possibile tenere traccia di tutti gli step che vengono eseguiti dal metodo `isSatisfiable`, ovvero di tutte le applicazioni della regola di risoluzione. Per farlo è stata scritta la classe `Step`, le cui istanze memorizzano il numero di step attuale, le due clausole di premessa, la clausola risolvente, il letterale da considerare per la regola, ed infine le informazioni che dicono che la risolvente viene scartata perchè è tautologica oppure già presente nella lista di clausole.

Nella classe `Resolution` è presente il metodo `setEnableSteps` che prende in input un valore booleano che viene impostato sul campo statico `enableSteps`. Se questo campo è impostato a `true`, verrà stampata su Standard Output (*Stdout*) la lista degli step quando viene chiamato il metodo `isSatisfiable`.

Gli step vengono memorizzati nel campo statico `trace` (di tipo `List<Step>`). Nel codice 3.3, nelle righe 16-19 viene creata una nuova istanza di `Step` e aggiunta a `trace` soltanto nel caso in cui il campo `enableSteps` è `true`. Nelle righe 25-30 viene controllato se la clausola risolvente è tautologica oppure è già presente in `listCl`, e vengono chiamati i metodi `setTautology` o `setAlreadyPresent` sullo step, per indicare che in quello step la risolvente viene scartata, specificandone il suo motivo. Nella riga 22 e nella riga 39 viene richiamato il metodo `printTrace`; esso itera sulla lista `trace`, stampando tutti gli step su *Stdout*.

Esempio di funzionamento degli step

Supponendo che il campo `enableSteps` sia impostato a `true`, consideriamo l'esecuzione del metodo `isSatisfiable` con il seguente insieme di clausole `s` in input:

$$\mathtt{s} = \{\ \{a\}; \{\neg a, b\}; \{\neg b\}\ \}$$

La lista degli step che verrà stampata su *Stdout* è la seguente:

```
STEP NUMBER 1:
First premise: {a}
Second premise: {~a, b}
Conclusion: {b}
obtained by removing the literal "a" and its opposite.
----------------------------------------------------------

STEP NUMBER 2:
First premise: {~a, b}
Second premise: {~b}
Conclusion: {~a}
obtained by removing the literal "b" and its opposite.
----------------------------------------------------------

STEP NUMBER 3:
First premise: {~b}
Second premise: {b}
Conclusion: {}
obtained by removing the literal "~b" and its opposite.
----------------------------------------------------------
```

Come si può notare, nello *step 3* viene trovata la clausola vuota, quindi in questo caso il metodo restituisce `false` (`s` è *insoddisfacibile*).

Capitolo 4

Implementazione in Java di formule generiche

Finora ci siamo concentrati soltanto su formule espresse in *CNF* rappresentate mediante la classe `ClauseSet`. Se volessimo implementare una formula qualsiasi della logica proposizionale sono necessari altri elementi. I package *formula* e *connective* del progetto java visto nella sezione 3.1 contengono i file necessari per rappresentare le formule della logica proposizionale. Se si volesse verificare la soddisfacibilità di una formula, è necessario convertire prima la formula in *CNF* (ottenendo così l'istanza di `ClauseSet` associata), e poi richiamare il metodo `isSatisfiable` della classe `Resolution` dando in input quella istanza. Questo aspetto viene affrontato nella sezione 4.5.

4.1 La classe astratta Formula

Come visto nel capitolo 1, una formula può essere *atomica* o *composta*, perciò nell'implementazione in Java è stata scritta la classe astratta `Formula`, che viene estesa dalle classi concrete `AtomicFormula` e `CompoundFormula`.

La classe `Formula` contiene soltanto il metodo astratto `toCnf` che permette di convertire la formula in *CNF*, restituendo l'istanza di `ClauseSet` corrispondente (si veda la sezione 4.5).

4.2 La classe AtomicFormula

Questa classe estende `Formula` ed istanzia una formula atomica. Essa contiene un campo `atm` di tipo `Atom` che lo identifica, e alcuni semplici metodi per la sua gestione:

il metodo per ottenere il nome (`getName`), il metodo per ottenere il letterale associato (`toLiteral`), e i classici metodi `equals` e `toString`.

4.3 L'enumerazione Connective

Prima di considerare la classe `CompoundFormula` è bene vedere prima l'enumerazione `Connective`. Essa si trova nel package *connective* e definisce delle costanti enumerative che descrivono i connettivi della logica proposizionale usati dalle formule composte. La seguente tabella mostra le costanti associate alla loro rappresentazione testuale (ottenute dal metodo `toString`):

Costante enumerativa	**Rappresentazione testuale**
NOT	$\sim$
AND	&
OR	$\|$
IMPLIES	$->$
IFF	$<->$

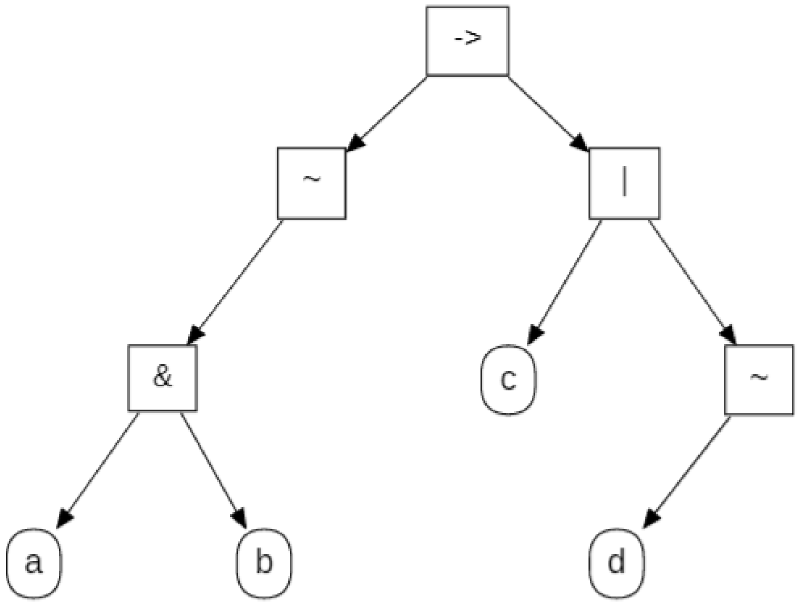
4.4 La classe CompoundFormula

Questa classe estende `Formula` ed istanzia una formula composta, utilizzando le costanti dell'enumerazione `Connective`. Per la rappresentazione interna di una formula composta si usa un albero binario, che definisce la sua struttura ricorsiva. La radice dell'albero contiene il connettivo principale della formula, i nodi interni contengono i connettivi delle sue sottoformule, ed infine le foglie contengono le sue formule atomiche. I nodi che contengono i connettivi binari hanno due figli, mentre i nodi che contengono i connettivi unari hanno un solo figlio (il figlio sinistro).

Consideriamo il seguente esempio di formula:

$$\neg(a \wedge b) \implies (c \vee \neg d)$$

La sua rappresentazione ad albero è la seguente:



In questo albero, la radice è il nodo "$->$", perché è il connettivo principale della formula dell'esempio. I nodi "$\sim$" hanno solo il figlio sinistro perché rappresentano la negazione, e le foglie sono le atomiche *a, b, c, d.*

La classe `CompoundFormula` ha quindi come attributi il campo `mainConnective` che definisce il connettivo principale, e il campo `subFormulas` che consiste in un array di formule che contiene i nodi figli. La classe contiene dei metodi getter per ottenere il connettivo, il figlio destro ed il figlio sinistro, e sovrascrive i metodi `toString` e `toCnf`.

4.5 Clausificazione di una formula

Come spiegato all'inizio di questo capitolo, se volessimo verificare la soddisfacibilità di una formula F tramite il metodo di risoluzione, è fondamentale effettuare prima la *clausificazione* di F, ovvero la sua conversione in *CNF*, in modo tale da ottenere l'insieme di clausole necessario per il metodo di risoluzione. Per fare ciò bisogna applicare alcune regole di inferenza:

1. **Eliminazione della doppia implicazione**:

$$(p \iff q) \quad \equiv \quad ((p \implies q) \land (q \implies p))$$

2. **Eliminazione dell'implicazione**:

$$(p \implies q) \quad \equiv \quad (\neg p \vee q)$$

3. **Proprietà distributiva di or su and**:

$$(p \vee (q \wedge r)) \quad \equiv \quad ((p \vee q) \wedge (p \vee r))$$

4. **Doppia negazione**:

$$\neg\neg p \quad \equiv \quad p$$

5. **Leggi di De Morgan**:

$$\neg(p \wedge q) \quad \equiv \quad (\neg p \vee \neg q)$$

$$\neg(p \vee q) \quad \equiv \quad (\neg p \wedge \neg q)$$

Queste regole vengono applicate nel metodo astratto `toCnf` della classe `Formula`. Questo metodo ha quindi due diverse implementazioni: una per le formule atomiche e una per le formule composte.

L'implementazione del metodo `toCnf` nella classe `AtomicFormula` è molto semplice: esso restituisce un `ClauseSet` contenente una sola clausola, tale clausola contiene soltanto il letterale associato a quella formula atomica.

L'implementazione del metodo nella classe `CompoundFormula` invece applica sulla formula le regole di inferenza sopra citate, seguendo un approccio ricorsivo

4.5.1 Esempio di clausificazione

Consideriamo la seguente formula:

$$(a \vee b) \implies c$$

la clausificazione viene fatta in tre passaggi:

- Eliminazione implicazione:
$$\neg(a \vee b) \vee c$$
- De Morgan:
$$(\neg a \wedge \neg b) \vee c$$
- Proprietà distributiva:
$$(\neg a \vee c) \wedge (\neg b \vee c)$$

La formula ricavata nell'ultimo passaggio è in *CNF*, ed essa corrisponde all'insieme di clausole:

$$\{ \{\neg a, c\}; \{\neg b, c\} \}$$

4.6 Parsing di formule

Se volessimo utilizzare un programma che legge da un flusso di input una formula in formato testuale (ad esempio da *Stdin* o da un file di testo), è necessario interpretare la stringa di testo in modo tale da ottenere la formula. La trasformazione di questa rappresentazione testuale in strutture dati utilizzabili dal software è un processo noto come parsing. Nel nostro sistema, implementiamo un parser per convertire le formule da stringhe di testo in oggetti di tipo `Formula`, che possono essere utilizzati dal programma. Per eseguire il parsing, bisogna definire prima una grammatica che riconosce il linguaggio per le formule. A partire dalla grammatica vengono generati il *lexer*, il *parser* e il *listener*.

4.6.1 Utilizzo del parser ANTLR4

Il parser utilizzato nel nostro sistema è *ANTLR4*. Nella struttura del progetto presente nella sezione 3.1, nel package *antlr4* è presente il file *FormulaExpression.g4* che contiene la grammatica. Inoltre, sono presenti quattro file che sono stati generati automaticamente da *ANTLR4*:

- **FormulaExpressionLexer.java**: esso consiste nel lexer, ovvero l'analizzatore lessicale; il suo scopo è quello di analizzare la stringa in input ed ottenere da esso un flusso di *token*. In questo caso i token sono i connettivi, le variabili atomiche, e i caratteri di *white space*.

- **FormulaExpressionParser.java**: esso consiste nel parser, che esegue l'analisi sintattica della stringa in input generando un **albero sintattico** (*AST*), fondamentale per la costruzione della formula.

- **FormulaExpressionListener.java**: è un interfaccia che definisce i metodi che vengono richiamati durante l'attraversamento dell'albero sintattico.

- **FormulaExpressionBaseListener.java**: è una classe che implementa l'interfaccia `FormulaExpressionListener`, dove ogni metodo sovrascritto non esegue alcuna operazione.

Oltre a questi file, ne sono presenti altri due:

- **FormulaListenerImplementation.java**: è una classe che estende il Base Listener. Utilizza uno *stack* come campo per la creazione delle sottoformule, e sovrascrive alcuni metodi per la creazione della formula durante l'attraversamento dell'*AST*.
- **ParseFormula.java**: è una classe che contiene soltanto il metodo `parse`. Esso è il metodo principale che prende in input una stringa, e mette insieme tutte le componenti viste precedentemente, creando tutto l'occorrente per il parsing. Infine restituisce la formula ottenuta a partire dalla stringa in input.

La grammatica utilizzata

Nel codice 4.1 è presente il contenuto del file *FormulaExpression.g4* che descrive la grammatica.

Codice 4.1: file FormulaExpression.g4

```
grammar FormulaExpression;

start : formula EOF;

formula : atomic_formula
        | '(' formula ')'
        | unary_connective formula
        | formula binary_connective formula
        ;

atomic_formula : LITERAL ;

unary_connective : NOT ;

binary_connective : AND | OR | IMPLIES | IFF;

//token
AND : '&' ;
OR : '|' ;
IMPLIES : '->' ;
NOT : '~' ;
IFF : '<->';

LITERAL : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

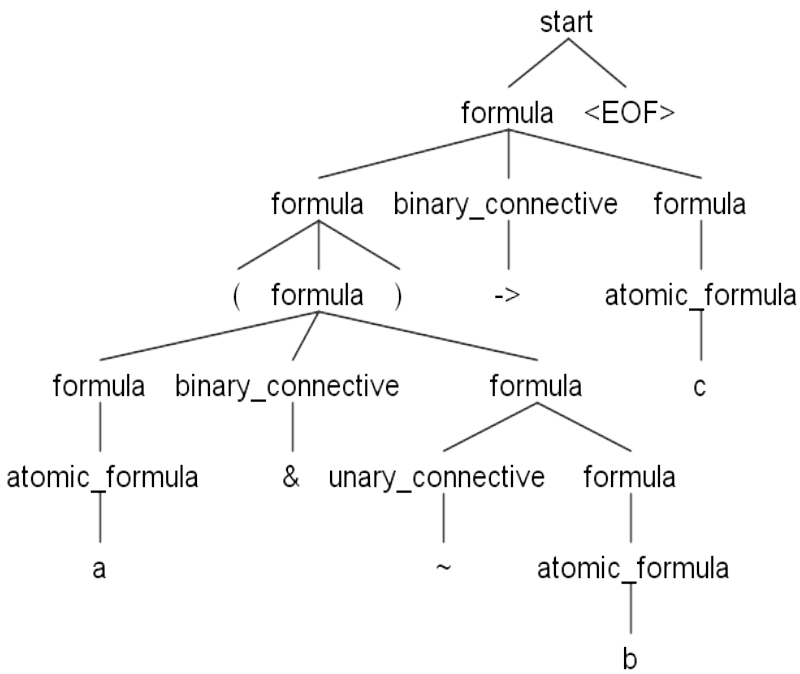
In questa grammatica ci sono le regole che definiscono le formule generiche, le formule atomiche e composte, e le regole che descrivono i connettivi unari e binari; le sottoformule sono distinguibili grazie alle parentesi tonde, che specificano anche la precedenza tra i vari connettivi della formula. in fondo al file è presente la definizione dei token. È bene notare che la rappresentazione testuale dei connettivi in questa grammatica è la medesima utilizzata dall'enumerazione `Connective`.

Esempio di *AST*

Consideriamo la seguente stringa:

$$(a \;\&\; \sim b) \quad -> \quad c$$

se eseguiamo il parsing di questa stringa otteniamo il seguente AST:



durante l'attraversamento di questo albero vengono eseguiti i metodi `ExitAtomic_formula` e `exitFormula`, che vengono implementati nella classe `FormulaListenerImplementation` e vengono richiamati rispettivamente quando si esce da un nodo *atomic_formula* e da un nodo *formula*. Nel primo caso si crea una formula atomica e la si aggiunge allo *stack* (tramite l'operazione di *push*), nel secondo caso si controlla la presenza di un figlio che contiene un connettivo unario o binario, e si crea una formula composta applicando quel connettivo alla formula in cima allo stack (nel caso di connettivo unario) oppure alle prime due formule in cima allo stack (nel caso di connettivo binario). Per ottenere la formula in cima allo stack viene usata l'operazione di *pop* su di esso.

Alla fine della visita dell'AST, in cima allo stack ci sarà la formula associata alla stringa di partenza, che viene ottenuta dal metodo `getFormula` e a sua volta restituita dal metodo `parse`; nell'esempio visto in precedenza la formula ottenuta è:

$$(a \land \neg b) \implies c$$

Capitolo 5

Logica modale ed estensione del metodo di risoluzione

Capitolo 6

Applicazioni pratiche del metodo di risoluzione

6.1 Verificare la validità di una formula

6.2 Dimostrazione di conseguenze logiche

6.3 Dimostrazione di equivalenze logiche

Capitolo 7

Testing

7.1 Struttura del file di test

Capitolo 8

Conclusioni

Bibliografia

[1] ...

Ringraziamenti

...