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

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 la logica modale, che rappresenta 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**)
- $\Longrightarrow$ è l'implicazione logica (**implica**)
- $\Longleftrightarrow$ 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 un letterale semplice e un letterale negato. 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).

metodo getOpposite sovrascritto dalla classe Atom

```
@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

```
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 3.3.1)
- `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` perchè `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

```
        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 =
                       getComplementaryLiterals(c1, c2);
                    if (complemLit != null) {
                        if (index1 < index2) {
                            (visited.get(index1)).add(index2);
                        } else {
                            (visited.get(index2)).add(index1);
                        }
                        Clause newClause = resolRule(c1, c2,
                           complemLit);
                        Step step = null;
                        if (enableSteps) {
                            step = new Step(c1, c2,
                               newClause, complemLit);
                            trace.add(step);
                        }
                        if (newClause.isEmpty()) {
                            if (enableSteps) printTrace();
                            return false;
                        }
                        if (newClause.isTautology()) {
                            if (enableSteps)
                                step.setTautology();
                        } else if
                           (listCl.contains(newClause)) {
                            if (enableSteps)
                                step.setAlreadyPresent();
                        } else {
                            visited.put(newClause.getIndex(),
                               new HashSet<>());
                            listCl.add(newClause);
                        }
```

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

Una volta entrato nel secondo ciclo, il metodo esegue la regola di risoluzione (metodo `resolRule`) `c1` e `c2`

3.3.2 Memorizzazione delle coppie di clausole visitate

3.3.3 Implementazione della regola di risoluzione

3.3.4 Gestione degli step

Capitolo 4

Implementazione in Java di formule generiche

4.1 La classe astratta Formula

4.2 La classe AtomicFormula

4.3 La classe CompoundFormula

4.3.1 L'enumerazione Connective

4.4 Conversione di una formula in CNF

4.5 Parsing di formule

4.5.1 Utilizzo del parser ANTLR4

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

...