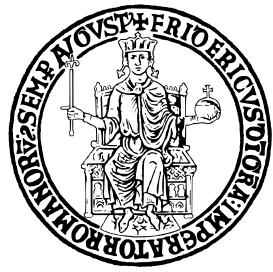


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II



SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Intelligent Web

Sviluppo di un motore inferenziale per la soddisfabilità di un concetto in \mathcal{ALC}

Docente

Prof. Luigi Sauro

Studenti
Salvatore Davide Amodio
N97000369

Monteiro Del Prete
N97000370

ANNO ACCADEMICO 2021–2022

Pagina intenzionalmente lasciata in bianco.

Indice

Utilizzo rapido	3
1 Descrizione del problema	6
2 Implementazione del metodo dei Tableaux	7
2.1 Blocking	8
2.2 Lazy Unfolding	10
3 Visualizzazione del risultato	14

Utilizzo Rapido

Di seguito è fornita una guida veloce per l'utilizzo del programma frutto dell'implementazione di un motore inferenziale per la logica \mathcal{ALC} . L'input è il seguente:

- un concetto C
- una TBox \mathcal{T}

La TBox \mathcal{T} deve essere scritta in un file serializzato in una sintassi accettata da OWL API (preferibilmente in *sintassi di Manchester*) e deve essere posizionato nella cartella `ontologies/`. Per un esempio d'uso è impostata di default la Tbox `food.man.owl`¹. Il file presenta un insieme di GCI (General Concept Inclusion) del tipo:

$$A \sqsubseteq D, A \equiv D.$$

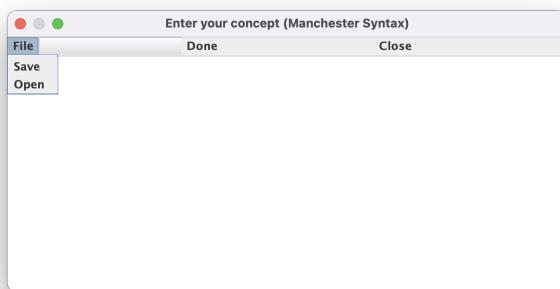


Figura 1: Caricamento concetto C da file

¹Tale TBox \mathcal{T} contiene assiomi terminologici di base sul cibo

Il concetto \mathcal{C} invece si ottiene tramite interfaccia grafica mediante due possibili modalità differenti: si può scegliere sia di importare il concetto \mathcal{C} da file tramite il tasto **Open** (figura 1), sia di scrivere il concetto manualmente nella text area (figura 2). In tal caso è importante valutare l'uso di prefissi (**t**: se l'ontologia usata è quella di default).

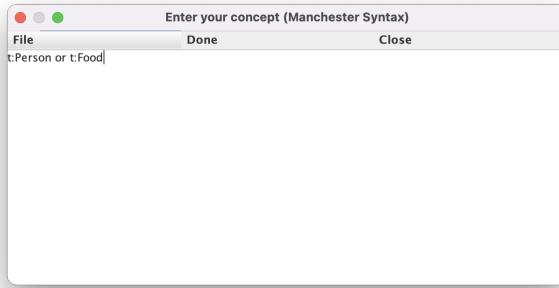


Figura 2: Caricamento concetto \mathcal{C} da text area

In generale è data all'utente anche la possibilità di poter salvare in un file il concetto costruito tramite il tasto **Save** (figura 1).

Il motore inferenziale basato sul metodo dei Tableau è utilizzato per verificare la soddisfabilità del concetto \mathcal{C} rispetto alla TBox \mathcal{T} . Per avviarlo basta premere sul tasto **Done**. Se il concetto è scritto in maniera sintatticamente corretta ed è coerente con la Tbox \mathcal{T} indicata (tutte le entità presenti in \mathcal{C} sono dichiarate in \mathcal{T}) viene mostrato un messaggio che avverte dell'elaborazione del sistema inferenziale.

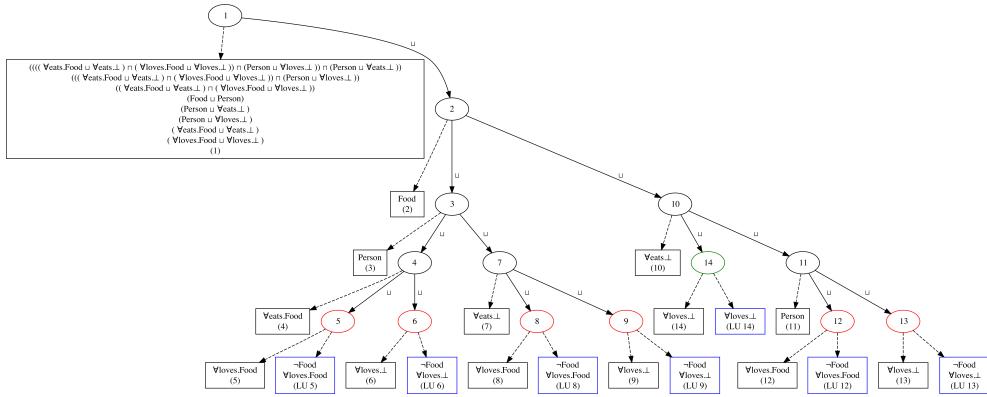


Figura 3: Rappresentazione a grafo del tableau

Alla fine dell'interrogazione il programma esporta il tableau nella cartella `result/` sia in formato `.rdf`, sia in formato `.png`. L'immagine viene mostrata alla fine della computazione (figura 3). Per arricchire il risultato del ragionamento, da terminale comparirà il concetto tradotto con OWL API, la risposta alla soddisfacibilità dello stesso e il tempo di esecuzione.

----- TABLEAUX METHOD FOR SAT IN ALC -----

INPUT CONCEPT:

```
ObjectUnionOf(<http://owl.api.tutorial#Food>
              <http://owl.api.tutorial#Person>)
```

[Time passed: 42ms] Concept satisfiability: Yes

Capitolo 1

Descrizione del problema

L'obiettivo di tale elaborato è la realizzazione di un motore inferenziale per la soddisficiabilità di un concetto per la logica \mathcal{ALC} . Nel dettaglio, il ragionatore tiene conto di una terminologia non vuota, dunque adopera la tecnica di blocking per garantire la decidibilità del metodo. Inoltre, sfrutta la tecnica di ottimizzazione del lazy unfolding per velocizzare la computazione.

Sia il concetto C che la TBox \mathcal{T} sono forniti mediante una serializzazione standard: C preso tramite interfaccia grafica come campo testo in formato Manchester e \mathcal{T} caricato tramite file con un formato supportato da OWL API. Il tableau risultato del ragionamento viene visualizzato sotto forma di grafo e salvato in un file RDF in Turtle notation con vocabolario apposito. In aggiunta viene riportato anche il tempo di computazione.

Capitolo 2

Implementazione del metodo dei Tableaux

Il seguente capitolo si concentra sull’implementazione del motore inferenziale in relazione ai principali concetti di blocking e lazy unfolding. Dunque, tratta le soluzioni scelte affiancandole ad un’analisi sommaria delle funzioni che le implementano.

Per fornire una descrizione più afferrabile è utile esporre la struttura implementativa da cui si è partiti (nell’albero della directory di lavoro sono assenti file sussidiari):

```
src/main/
└── view/
    ├── IOParser.java
    ├── Reasoner.java
    └── RDFGraphWriter.java
```

Nella directory `view` è presente l’apparato di interfaccia grafica che non verrà trattato. Il file `IOParser` gestisce la GUI necessaria per il parsing del concetto preso da campo di testo e definisce il caricamento della TBox. Il motore inferenziale è implementato nel `Reasoner`, contenete l’algoritmo DFS e funzioni di blocking e lazy unfolding. Infine `RDFGraphWriter` presenta le funzioni per la scrittura del tableau risultato del ragionamento sotto forma di grafo e file RDF. Come già accennato, le seguenti sezioni tratteranno frammenti di codice e implementazioni relative al `Reasoner`, con riferimenti alle principali classi di OWL API.

2.1 Blocking

Come richiesto, particolare attenzione è stata posta nel controllo alla base della tecnica di blocking: $\mathcal{L}(x) \subseteq \mathcal{L}(y)$ (dove y è stato generato prima di x nella costruzione del tableau). La strategia adottata viene di seguito analizzata da un punto di vista teorico, per poi essere affrontata da un punto di vista pratico.

Descrizione teorica della strategia

Al fine di rendere il controllo quanto più celere possibile, durante la creazione di un nodo quest'ultimo presenterà un puntatore al nodo padre così da poter iterare sugli antenati e controllare la condizione di blocking.

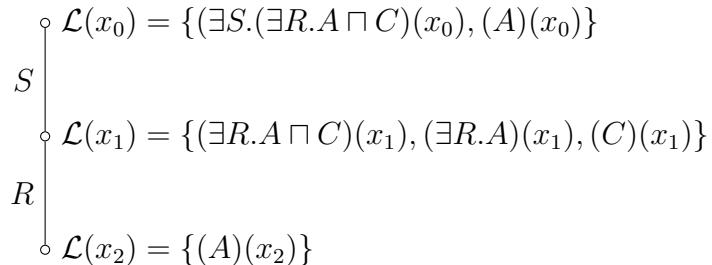


Figura 2.1: Blocking

È necessario controllare tutti gli antenati di un nodo, non basta limitarsi al controllo del padre. In figura 2.1 è rappresentato un esempio che chiarisce tale problematica. Dal primo controllo effettuato si dedurrebbe $\mathcal{L}(x_2) \not\subseteq \mathcal{L}(x_1)$ mentre risalendo all'antenato di secondo grado si avrebbe che $\mathcal{L}(x_2) \subseteq \mathcal{L}(x_0)$.

Descrizione pratica della strategia

Per comprendere al meglio l'implementazione di questa strategia, iniziamo con il descrivere gli elementi base. L'algoritmo principale di discesa in profondità utilizza la classe sussidiaria `Node`, necessaria per descrivere una struttura ad albero rappresentativa del tableau esteso. Ogni nodo ha associato un individuo (`OWLIndividual`), utilizzato per definire le *class assertion* che compongono l'insieme $\mathcal{L}(x)$, e un riferimento al nodo padre. La funzione seguente viene utilizzata per, eventualmente, impostare il nodo come bloccato.

Per semplicità di lettura alcune istruzioni sono rimosse.

```

private void setIfBlocked(Node node) {
    while(parentNode != null && !blocked) {
        for(OWLAxiom firstAxiom: structure) {
            if(firstAxiom instanceof OWLClassAssertionAxiom) {
                classAssertion = (OWLClassAssertionAxiom)
                    firstAxiom;

                ce = classAssertion.getClassExpression();
                individual = classAssertion.getIndividual();

                if(individual.equals(node.getIndividual())) {

                    if(ce instanceof OWLObjectIntersectionOf) {
                        ceFlat = ce.asConjunctSet();
                    } else if(ce instanceof OWLObjectUnionOf) {
                        ceFlat = ce.asDisjunctSet();
                    } else {
                        ceFlat = null;
                    }

                    if(ceFlat != null) {
                        // verifica se il padre contiene
                        // l'assioma appiattito
                    } else {
                        // verifica se il padre contiene
                        // l'assioma
                    }
                }
            }
        }
        parentNode = parentNode.getParent();
    }
    node.setBlocked(blocked);
}

```

A partire dal nodo si ricavano individuo e struttura ($\mathcal{L}(x)$), il resto è una verifica dell'inclusione $\mathcal{L}(x) \subseteq \mathcal{L}(y)$, dove y è un antenato di x .

Come è possibile notare, non tutte le asserzioni in $\mathcal{L}(x)$ vengono considerate: quelle che asseriscono su un individuo che non è il nodo di interesse vengono scartate, ciò permette di velocizzare il controllo. Un'ulteriore accor-

tezza viene posta nell'appiattimento di congiunzioni e disgiunzioni in modo da considerare anche i casi $(A \sqcap (B \sqcap B))$ e $((A \sqcap B) \sqcap C)$ durante l'uguaglianza degli assiomi, situazione non gestita dalla OWL API.

2.2 Lazy Unfolding

Come scelta progettuale di ottimizzazione dell'esecuzione del metodo dei tableaux è stata richiesta l'implementazione della tecnica di *Lazy Unfolding*. Lo sviluppo di tale funzionalità ha previsto l'utilizzo di due funzioni principali: `getLazyUnfoldingPartition`, delegata al partizionamento della TBox \mathcal{T} nei sottoinsiemi \mathcal{T}_u (unfoldable) e \mathcal{T}_g , e `applyLazyUnfoldingRule` delegata all'applicazione delle regole rispetto \mathcal{T}_u .

```
private Pair<List<OWLAxiom>, List<OWLAxiom>>
getLazyUnfoldingPartition ( List<OWLAxiom> tbox ) {

    for (OWLAxiom axiom : tbox){
        if (isUnfoldableAddingEquivalentClass (Tu, axiom)){
            Tu.add (axiom);
        } else if (isUnfoldableAddingSubClass (Tu, axiom)){
            axiom = transformSubClassAx (axiom);
            Tu.add (axiom);
        } else{
            Tg.add (axiom);
        }
    }
    return new Pair<List<OWLAxiom>, List<OWLAxiom>>(Tu, Tg);
}
```

L'algoritmo di partizionamento valuta se per ogni assioma presente in \mathcal{T} la sua aggiunta in \mathcal{T}_u comprometta il suo essere unfoldable. Se così fosse verrebbe aggiunto in \mathcal{T}_g . Di tutti i possibili assiomi presenti in \mathcal{T} , sono soggetti a valutazione solo gli assiomi di uguaglianza e di inclusione:

- $A \equiv C$. Principalmente viene valutato se A non sia definito due volte (non compaia due volte in LHS in \mathcal{T}_u) e se non sia definito direttamente o indirettamente in termini di se stesso (dipenda direttamente o indirettamente da se stesso).

- $A \sqsubseteq C$. Anche qui viene valutato se A non compaia due volte in LHS in \mathcal{T}_u . Inoltre se l'assioma si presenta in una forma del tipo $A \sqcap B \sqsubseteq C$, prima di verificare la condizione precedente, si verifica se uno dei congiunti in LHS sia un concetto atomico; in tal caso l'assioma viene trasformato in $A \sqsubseteq \neg B \sqcup C$. In caso contrario l'assioma è scartato.

La seconda funzione invece controlla se ci siano le condizioni necessarie per poter applicare le seguenti regole di Lazy Unfolding:

<i>Regola</i>	U_1
<i>Condizione</i>	$A \in \mathcal{L}(x), (A \equiv C) \in \mathcal{T}_u$ e $C \notin \mathcal{L}(x)$
<i>Azione</i>	$\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{C\}$
<i>Regola</i>	U_2
<i>Condizione</i>	$\neg A \in \mathcal{L}(x), (A \equiv C) \in \mathcal{T}_u$ e $\neg C \notin \mathcal{L}(x)$
<i>Azione</i>	$\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{\neg C\}$
<i>Regola</i>	U_3
<i>Condizione</i>	$A \in \mathcal{L}(x), (A \sqsubseteq C) \in \mathcal{T}_u$ e $C \notin \mathcal{L}(x)$
<i>Azione</i>	$\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{C\}$

In tal modo il tableau viene espanso attraverso la tecnica di *Modus Ponens*. Dando una occhiata all'aspetto implementativo, nella seconda funzione sono presenti vari frammenti di codice delegati alla gestione delle regole. Ad ogni listato seguirà una breve spiegazione del funzionamento.

```

for (OWL Axiom axiom : Tu) {
    if (axiom instanceof OWLEquivalentClassesAxiom) {
        equivAx = (OWLEquivalentClassesAxiom) axiom;
        // fisrtClass is LHS of equivAx
        // secondClass is DHS of equivAx

        if (firstClass.equals(ce)) {
            secondClass = secondClass.getNNF();
            // create newClassAssertion as
            OWLClassAssertionAxiom(newClass , x);
            structure.add(newClassAssertion);
            break;
        }
    }
}

```

Per la gestione di U_1 , si estrapola `firstClass` e `secondClass` da ogni `axiom` (assioma di equivalenza) presente in \mathcal{T}_u . Si controlla la possibile uguaglianza tra `firstClass` e `ce` (A) e in tal caso si procede ad aggiungere `secondClass` (C) in `structure` ($\mathcal{L}(x)$).

```
for (OWLAxiom axiom: Tu) {
    if (axiom instanceof OWLEquivalentClassesAxiom) {
        equivAx = (OWLEquivalentClassesAxiom) axiom;
        // fisrtClass is LHS of equivAx
        // secondClass is DHS of equivAx

        if (firstClass.equals(ce)) {
            secondClass = secondClass.getComplementNNF();
            // create newClassAssertion as
            OWLClassAssertionAxiom(secondClass, x);
            structure.add(newClassAssertion);
            break;
        }
    }
}
```

Per l'applicazione di U_2 , dopo aver controllato che la presenza di `ce` ($\neg A$) in $(\mathcal{L}(x))$, si procede in modo analogo alla gestione della regola U_1 . La differenza sostanziale sta nella creazione di `newClassAssertion` a partire dal complemento in NNF di `secondClass`.

```
for (OWLAxiom axiom: Tu) {
    if (axiom instanceof OWLSubClassOfAxiom) {
        subClassAx = (OWLSubClassOfAxiom) axiom;

        // fisrtClass is subClass of subClassAx
        // secondClass is superClass of subClassAx

        if (firstClass.equals(ce)) {
            secondClass = secondClass.getNNF();
            // create newClassAssertion as
            OWLClassAssertionAxiom(secondClass, x);
            structure.add(newClassAssertion);
            break;
        }
    }
}
```

In ultimo viene valutata l'applicazione di U_3 . Qui si controlla la presenza di un assioma di inclusione in \mathcal{T}_u che abbia A in LHS. L'applicazione è analoga alla regola U_1 .

Capitolo 3

Visualizzazione del risultato

Una volta fissata una terminologia e fornito in ingresso un concetto, il risultato del ragionamento sarà una risposta booleana arricchita da una descrizione visiva ed una più formale. Entrambe vengono definite durante l'estensione stessa del tableau tramite metodi appositi contenuti nella classe `RDFGraphWriter.java`. La prima riguarda la rappresentazione a grafo del tableau, in cui ogni arco è etichettato a seconda se la regola applicata è quella di disgiunzione o l'esistenziale. Inoltre ogni nodo ha un identificativo che richiama l'ordine di visita da parte dell'algoritmo.

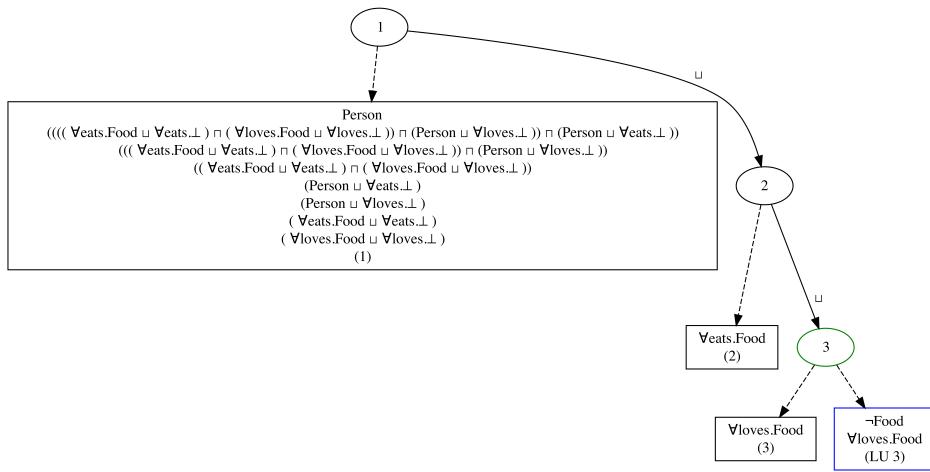


Figura 3.1: Rappresentazione a grafo di un tableau risultato

Dalla figura 3.1, ogni nodo in aggiunta all'arco della regola applicata ha un arco segmentato che va ad indicare l'insieme $\mathcal{L}(x)$ ad esso associato. La

colorazione in blu indica che l'insieme di concetti è derivato dall'applicazione delle regole di lazy unfolding.

La rappresentazione a grafo ha un corrispettivo nella rappresentazione in formato RDF. Per questa è stato ideato un vocabolario i cui QNames sono indicativi di archi della \sqcup -rule (**ex:orEdge**), archi della \exists -rule (**ex:exEdge**) ed insieme delle asserzioni $\mathcal{L}(x)$ (**ex:labels**). La serializzazione scelta per il modello RDF è la Turtle notation per una maggiore leggibilità.

```
@prefix ex: <http://example.org/> .

<http://example.org/node#2>
    ex:labels  "⊸eats.Food" ;
    ex:orEdge  <http://example.org/node#3> .

<http://example.org/node#1>
    ex:labels  "Person , ((( (⊸eats.Food ⊓ ⊸eats.⊥) ⊓ ( ⊸
        loves.Food ⊓ ⊸loves.⊥ )) ⊓ (Person ⊓ ⊸loves.⊥)) ⊓
        (Person ⊓ ⊸eats.⊥)), ((( (⊸eats.Food ⊓ ⊸eats.⊥) ⊓
        ( ⊸loves.Food ⊓ ⊸loves.⊥)) ⊓ (Person ⊓ ⊸loves.⊥
        )), (( (⊸eats.Food ⊓ ⊸eats.⊥) ⊓ ( ⊸loves.Food ⊓ ⊸
        loves.⊥)), (Person ⊓ ⊸eats.⊥), (Person ⊓ ⊸loves.⊥
        ), ( ⊸eats.Food ⊓ ⊸eats.⊥), ( ⊸loves.Food ⊓ ⊸
        loves.⊥)" ;
    ex:orEdge  <http://example.org/node#2> .

<http://example.org/node#3>
    ex:labels  "¬Food , ⊸loves.Food" .
```