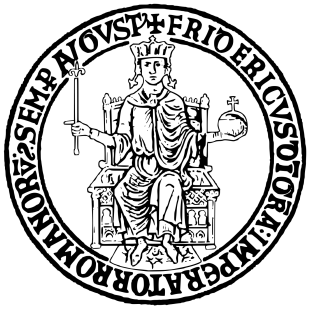
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Implementazione di una procedura di decisione per il frammento a due variabili con uguaglianza in Vampire

Relatore
Prof. Fabio Mogavero

Candidata
Michela Pollio
N86/3697

Anno Accademico 2025–2026

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 Triennale in Informatica

IMPLEMENTAZIONE DI UNA PROCEDURA DI DECISIONE PER IL FRAMMENTO A DUE VARIABILI CON UGUAGLIANZA IN VAMPIRE

Relatore

Prof. Fabio Mogavero

Candidata

Michela Pollio

N86/3697

Anno Accademico 2025–2026

Alla mia famiglia

Indice

Introduzione **5**

1 Background **7**
1.1 Logica del Primo Ordine e Dimostrazione Automatizzata 7
1.1.1 Sintassi e Semantica della Logica del Primo Ordine 7
1.1.2 Soddisfacibilità, Validità e Indecidibilità 8
1.1.3 Dimostrazione Automatica e Calcoli a Saturazione 8
1.2 Architettura del Dimostratore Vampire 9
1.2.1 Rappresentazione Interna di Formule e Clausole 9
1.2.2 Modulo di Preprocessing e Clausificazione 10
1.2.3 Kernel e Architettura Given Clause 10
1.2.4 Strategie di Saturazione: Otter, Discount e LRS 11
1.2.5 Splitting e Architettura AVATAR 11
1.3 Il Frammento a Due Variabili con Uguaglianza 12
1.3.1 Definizione Sintattica di $\mathrm{FO}^2_=$ 12
1.3.2 Proprietà del Modello Limitato e Complessità 12
1.3.3 Forma normale di Scott 13
1.3.4 Risoluzione e Problematiche legate all'Uguaglianza 13

2 Descrizione della Procedura di Decisione per $\mathrm{FO}^2_=$ **15**
2.1 Panoramica della Procedura a Due Stadi 15
2.2 La Trasformazione per l'Eliminazione dell'Uguaglianza 15
2.2.1 Lemma 1: Eliminazione delle Costanti Individuali 15
2.2.2 Lemma 2: Forma Normale di Scott 17
2.2.3 Lemma 3: Spostamento dell'Uguaglianza 17
2.2.4 Lemma 4: Risoluzione Ristretta 18
2.2.5 Lemma 5: Trasformazione dell'Uguaglianza Positiva 18
2.2.6 Lemma 6: Eliminazione dei Quantificatori di Unicità 19
2.3 La Classe S^{2+i} 19
2.4 Il Calcolo di Risoluzione Indicizzata 20
2.4.1 Ordinamento sui Termini e Funzione di Selezione Σ_2 20
2.4.2 Regole del Calcolo 21
2.5 Correttezza, Completezza e Garanzia di Terminazione 21

3 Progettazione e architettura **23**
3.1 Requisiti e vincoli progettuali 23
3.2 Architettura generale e flusso di elaborazione 24
3.3 Scomposizione in componenti 24
3.3.1 Classificazione del frammento 26
3.3.2 Eliminazione dell'uguaglianza 26
3.3.3 Preparazione e validazione delle clausole 26

3.3.4 Rappresentazione indicizzata e calcolo di risoluzione 27
3.4 Interazioni e contratti tra i componenti 27

4 Implementazione della Procedura in Vampire 29
4.1 Modulo di Preprocessing e Validazione Sintattica 30
4.1.1 Validazione dei Vincoli del Frammento $\mathcal{S}_2$ 30
4.2 Eliminazione dell'Uguaglianza: il Modulo `RemoveEquality` 35
4.2.1 Lemma 1: Eliminazione delle Costanti Individuali 36
4.2.2 Lemma 2: Normalizzazione di Scott e Rinominazione Sottoformule Complesse 44
4.2.3 Lemma 3: Gestione dell'Uguaglianza 48
4.2.4 Lemma 4: Saturazione Ristretta per Clausole di Tipo 3 53
4.2.5 Lemma 5: Decomposizione in Componenti Unarie 59
4.2.6 Lemma 6: Eliminazione dei Quantificatori d'Unicità ed Assiomi di Congruenza 62
4.3 Modulo di Risoluzione e Decision Procedure (`Resolution`) 66
4.3.1 Indicizzazione $\mathcal{S}_2{+}i$, Ordinamento $<_2$ e Selezione Σ_2 (`FO2Resolution`) 66
4.3.2 Regole di Inferenza e Semplificazione (`FO2Inferences`) 69
4.3.3 Engine di Saturazione e Risoluzione (`FO2Solver`) 73

5 Risultati sperimentali 78
5.0.1 Creazione del benchmark TPTP 78
5.0.2 Classificatore $\mathrm{FO}^2_=$ 79
5.1 Dataset sintetici generati 94
5.1.1 Dimensioni di Scaling Sintattico e Strutturale 95
5.1.2 test2Equality 95
5.1.3 test2Mix 96
5.1.4 Benchmark generati senza uguaglianza 109
5.2 Metodologia di confronto e controllo delle risorse 115
5.2.1 Parametri di Confronto Simmetrico 115
5.2.2 Limiti della simmetria 115
5.2.3 Limiti temporali e memoria 115
5.2.4 Interpretazione e riproducibilità 116

6 Conclusioni 117

A Risultati sperimentali TPTP 119

Bibliografia 133

Introduzione

La logica del primo ordine (FOL, *First-Order Logic*) costituisce un formalismo per la rappresentazione della conoscenza, la specifica formale e la verifica automatica di sistemi hardware e software. La quantificazione su domini arbitrari e l'impiego di relazioni a n-arietà generica le consentono di rappresentare proprietà e relazioni complesse.

Come stabilito dai risultati di Church e Turing, la logica del primo ordine è indecidibile: non esiste un algoritmo generale che determini la validità di una formula arbitraria e termini in ogni caso. L'insoddisfacibilità è tuttavia semidecidibile: una procedura di dimostrazione per confutazione, corretta e refutazionalmente completa, con una strategia di ricerca equa, può derivare una contraddizione in un numero finito di passi quando l'insieme di clausole è insoddisfacibile. Per gli insiemi soddisfacibili, invece, la ricerca può proseguire indefinitamente.

A fronte dell'indecidibilità della logica del primo ordine, la ricerca segue due direzioni:

1. Lo sviluppo di **dimostratori automatici di teoremi a saturazione** (ATP, *Automated Theorem Provers*), quale Vampire. Tali sistemi implementano calcoli fondati sulla superposizione e sulla risoluzione ordinata, integrando tecniche di indicizzazione dei termini ed eliminazione delle clausole ridondanti. Benché efficaci su problemi di derivazione applicata e impiegati sistematicamente nei benchmark CASC, tali motori non assicurano la terminazione a priori su formule soddisfacibili generiche.

2. L'individuazione di **frammenti sintattici decidibili** della logica del primo ordine. Imponendo restrizioni strutturali — sul numero di variabili distinte, sull'arietà dei simboli di predicato o sul pattern dei quantificatori — è possibile definire sottoclassi che ammettono procedure di decisione con garanzia di terminazione sia per l'insoddisfacibilità sia per la soddisfacibilità.

Tra i frammenti decidibili, il **frammento a due variabili con uguaglianza**, indicato con $\mathrm{FO}^2_=$, è costituito dalle formule del primo ordine prive di simboli di funzione (escluse le costanti) che contengono al più due variabili distinte ed estendono la sintassi con il simbolo predefinito di uguaglianza ($=$), interpretato semanticamente come la relazione di identità sul dominio. Il frammento $\mathrm{FO}^2_=$ include le formalizzazioni standard di diverse logiche modali proposizionali e logiche descrittive.

Hans de Nivelle e Ian Pratt-Hartmann hanno definito una procedura di decisione per questo frammento, basata sul paradigma della **risoluzione**. Per il frammento privo di uguaglianza, 2, sono note procedure decisionali basate su forme ristrette di risoluzione ordinata, che garantiscono la terminazione grazie a specifici vincoli sulla struttura delle clausole e sulle inferenze ammesse. L'estensione al frammento con uguaglianza presenta invece ulteriori difficoltà: le tecniche di risoluzione impiegate

per il caso privo di uguaglianza non garantiscono direttamente la terminazione, rendendo necessario un trattamento specifico del simbolo di identità. La procedura di de Nivelle e Pratt-Hartmann affronta il problema mediante una trasformazione che riduce una formula di $\mathrm{FO}^2_=$ a una formula equisoddisfacibile appartenente al frammento FO^2 privo di uguaglianza, consentendo la decisione tramite una variante specializzata di risoluzione indicizzata con ordinamento liftabile.

L'**obiettivo di questo lavoro di tesi** consiste nella progettazione, implementazione e valutazione sperimentale di tale procedura decisionale all'interno dell'ATP Vampire. L'integrazione in Vampire consente di studiare l'applicabilità di una procedura teoricamente terminante all'interno di un dimostratore automatico general-purpose, sfruttandone le strutture dati e valutandone il comportamento sperimentale.

Struttura della Tesi. Il presente lavoro si articola nei seguenti capitoli:

- Il **Capitolo 1 (Background)** introduce la sintassi e la semantica della logica del primo ordine, i principi della dimostrazione automatica a saturazione e l'architettura interna di Vampire. Definisce formalmente il frammento $\mathrm{FO}^2_=$ e illustra le criticità relative alla terminazione dei calcoli a saturazione in presenza di uguaglianza.

- Il **Capitolo 2 (Procedura di Decisione per** $\mathrm{FO}^2_=$**)** espone la procedura di de Nivelle e Pratt-Hartmann: la sequenza di riscritture per l'eliminazione dell'uguaglianza, la classe clausale bersaglio e il calcolo di risoluzione indicizzata vincolato dalla funzione di selezione e dall'ordinamento ammissibile, unitamente alla dimostrazione formale di terminazione.

- Il **Capitolo 3 (Progettazione)** definisce le specifiche funzionali e la scomposizione modulare, descrivendo l'integrazione con la pipeline di clausificazione di Vampire e l'interazione tra i moduli di classificazione, trasformazione ed elaborazione clausale.

- Il **Capitolo 4 (Implementazione in Vampire)** documenta l'integrazione della procedura nella base di codice di Vampire, analizzando gli algoritmi realizzati: la procedura di riscrittura per la rimozione dell'uguaglianza e il motore di saturazione indicizzato.

- Il **Capitolo 5 (Analisi Sperimentale e Benchmark)** illustra il protocollo sperimentale adottato, i criteri di selezione e filtraggio delle istanze dalla libreria TPTP (v8.2.0) e la generazione di suite sintetiche parametriche; analizza quantitativamente i risultati ottenuti comparando la procedura dedicata con il motore standard di Vampire.

- Il **Capitolo 6 (Conclusioni)** riassume i risultati sperimentali, definisce i limiti operativi dell'approccio implementato e delinea le possibili estensioni future del sistema.

— 1 —

Background

In questo capitolo vengono presentati i fondamenti teorici e formali su cui si articola il lavoro svolto. La prima parte definisce la sintassi e la semantica della logica del primo ordine, la nozione di soddisfacibilità e i risultati di indecidibilità, introducendo i principi della dimostrazione automatica mediante calcoli a saturazione. La seconda parte descrive l'architettura di Vampire, analizzando la rappresentazione interna delle formule, la pipeline di preprocessing, il ciclo di saturazione della *Given Clause Architecture* e il modulo di splitting Avatar. La terza parte introduce il frammento a due variabili con uguaglianza ($\mathrm{FO}^2_=$), esaminandone la complessità computazionale, la forma normale di Scott e le problematiche di terminazione della risoluzione in presenza del simbolo di uguaglianza.

1.1 Logica del Primo Ordine e Dimostrazione Automatizzata

1.1.1 Sintassi e Semantica della Logica del Primo Ordine

Un linguaggio del primo ordine (FOL, *First-Order Logic*) è specificato da una signature $\Sigma = (\mathcal{P}, \mathcal{F}, \mathcal{C})$, dove $\mathcal{P}$ denota l'insieme dei simboli di predicato, $\mathcal{F}$ l'insieme dei simboli di funzione e $\mathcal{C}$ l'insieme dei simboli di costante (formalizzabili come funzioni a zero argomenti). A ogni simbolo $p \in \mathcal{P}$ e $f \in \mathcal{F}$ è associata un'arietà $n \geq 0$.

Dato un insieme numerabile di variabili $\mathcal{X}$, l'insieme dei **termini** è definito induttivamente come segue:

1. ogni variabile $x \in \mathcal{X}$ è un termine;
2. ogni costante $c \in \mathcal{C}$ è un termine;
3. se $f \in \mathcal{F}$ ha arietà n e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine.

Le formule atomiche, o **atomi**, hanno la forma $p(t_1, \ldots, t_m)$, dove $p \in \mathcal{P}$ ha arietà m e ciascun t_i è un termine. Se la segnatura include il simbolo di uguaglianza ($=$), le formule $s = t$ costituiscono atomi di uguaglianza. Un **letterale** L è un atomo A (positivo) o la sua negazione $\neg A$ (negativo). Le **formule ben formate** sono ottenute componendo atomi mediante connettivi proposizionali ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) e quantificatori ($\forall, \exists$). Una variabile è legata se ricade nel campo d'azione di un quantificatore, altrimenti è libera. Una formula priva di variabili libere è detta enunciato (*sentence*). Un termine o una formula priva di variabili è detta *ground*.

Una struttura $\mathcal{M} = (D, \mathcal{I})$ è definita da un dominio non vuoto D e da una funzione di interpretazione $\mathcal{I}$ tale che:

- $\mathcal{I}(c) \in D$ per ogni $c \in \mathcal{C}$;
- $\mathcal{I}(f) : D^n \to D$ per ogni $f \in \mathcal{F}$ di arietà n;
- $\mathcal{I}(p) \subseteq D^m$ per ogni $p \in \mathcal{P}$ di arietà m.

Il simbolo di uguaglianza $=$ è interpretato canonicamente come la relazione di identità su D: $\mathcal{I}(=) = \{(d, d) \mid d \in D\}$. Dato un assegnamento di variabili $\sigma : \mathcal{X} \to D$, la relazione di soddisfacimento $\mathcal{M}, \sigma \models \phi$ è definita ricorsivamente nel modo standard.

1.1.2 Soddisfacibilità, Validità e Indecidibilità

Una formula ϕ è detta:

- **Soddisfacibile** se esiste una struttura $\mathcal{M}$ e un assegnamento σ tali per cui $\mathcal{M}, \sigma \models \phi$;
- **Valida** ($\models \phi$) se $\mathcal{M}, \sigma \models \phi$ per ogni struttura $\mathcal{M}$ e assegnamento σ;
- **Insoddisfacibile** se non ammette alcun modello.

Poiché ϕ è valida se e solo se $\neg\phi$ è insoddisfacibile, la verifica di validità può essere ricondotta alla **dimostrazione per confutazione**: data una congettura C e un insieme di assiomi $\mathcal{A}$, si valuta l'insoddisfacibilità di $\mathcal{A} \cup \{\neg C\}$.

Per i teoremi di Church e Turing, la logica del primo ordine è indecidibile: non esiste un algoritmo uniforme di decisione per la validità su formule arbitrarie. Sebbene la logica del primo ordine sia indecidibile, la validità e l'insoddisfacibilità sono semidecidibili. Una procedura di refutazione corretta e completa, eseguita secondo una strategia di ricerca equa, deriva una contraddizione in un numero finito di passi quando l'insieme di clausole è insoddisfacibile. Se l'insieme è soddisfacibile, invece, la ricerca potrebbe non terminare.

1.1.3 Dimostrazione Automatica e Calcoli a Saturazione

I dimostratori automatici di teoremi (ATP) per formule del primo ordine operano prevalentemente mediante **calcoli a saturazione** applicati a insiemi di clausole.

Una **clausola** è una disgiunzione finita di letterali:

$$C = L_1 \vee \cdots \vee L_k$$

Le variabili in una clausola sono implicitamente quantificate universalmente. Per $k = 0$, la disgiunzione individua la clausola vuota $\square$, semanticamente falsa.

Dato un insieme di clausole S e un sistema inferenziale $\mathcal{I}$, il processo di saturazione calcola la chiusura di S rispetto alle regole di derivazione ammesse. Un calcolo refutazionalmente completo garantisce che S è insoddisfacibile se e solo se è possibile derivare $\square$.

I principali formalismi inferenziali per clausole del primo ordine includono:

- **Risoluzione binaria**:
$$\frac{A \vee C_1 \qquad \neg A' \vee C_2}{(C_1 \vee C_2)\theta}$$
dove $\theta = \mathrm{mgu}(A, A')$;

- **Fattorizzazione**:
$$\frac{A \vee A' \vee C}{(A \vee C)\theta}$$
dove $\theta = \mathrm{mgu}(A, A')$;

- **Superposizione**: estende la risoluzione ordinata all'uguaglianza introducendo regole di riscrittura guidate da ordinamenti di riduzione sui termini, evitando l'esplicitazione assiomatica delle proprietà di congruenza dell'uguaglianza.

1.2 Architettura del Dimostratore Vampire

Vampire è un dimostratore automatico per la logica del primo ordine sviluppato in C++ da Andrei Voronkov e Alexandre Riazanov. Il sistema adotta un'architettura modulare suddivisa nelle componenti di **Parsing**, **Preprocessing** e **Kernel**.

1.2.1 Rappresentazione Interna di Formule e Clausole

Vampire utilizza strutture dati progettate per rappresentare e manipolare in modo efficiente termini, letterali e formule, riducendo le allocazioni e la duplicazione delle informazioni.

Termini e Letterali. I termini sono rappresentati mediante il tipo unificato `TermList`. Se il termine è una variabile, `TermList` ne contiene l'identificatore intero; se è un termine composto o una costante, contiene un puntatore a un'istanza di `Term`. La classe `Term` memorizza il simbolo funtoriale registrato nella tabella `Signature`, l'arietà e l'array degli argomenti. Attraverso una rappresentazione basata su grafi diretti aciclici (DAG) con condivisione strutturale (*perfect sharing*), occorrenze identiche di un medesimo termine fanno capo a un'unica istanza di allocazione, riducendo i confronti di identità sintattica a verifiche tra puntatori. La classe `Literal` deriva da `Term`, uniformando la gestione degli atomi a quella delle strutture funzionali.

Unit, Formula e Clause. La struttura complessiva di un problema è incapsulata nella classe `Problem`, la quale gestisce una sequenza di oggetti di tipo `Unit`:

- `Clause`: modella una clausola memorizzando un array di puntatori a letterali (`Literal**`);

- `FormulaUnit`: incapsula l'albero sintattico di una formula non ancora clausificata, rappresentata tramite derivazioni della classe astratta `Formula` (`AtomicFormula`, `QuantifiedFormula`, `NegatedFormula`, `BinaryFormula`, `JunctionFormula`).

1.2.2 Modulo di Preprocessing e Clausificazione

La fase di preprocessing prepara le formule per il successivo calcolo a saturazione. A seconda della configurazione adottata, comprende operazioni di semplificazione, normalizzazione e trasformazione in forma clausale, tra cui:

1. **Rettificazione (*Rectify*)**: ridenominazione delle variabili quantificate per garantire nomi univoci ed esplicitazione dei quantificatori sulle eventuali variabili libere;

2. **Semplificazione proposizionale (*Simplify*)**: eliminazione delle costanti booleane banali ($\top, \bot$) e riduzione delle ridondanze;

3. **Compattazione (*Flattening*)**: linearizzazione di sequenze annidate di connettivi associativi omologhi;

4. **Eliminazione dei predicati puri**: rimozione di definizioni di predicati non referenziati;

5. **Ridenominazione strutturale (*Naming*)**: introduzione di predicati definitori ausiliari per sottoformule profonde, limitando l'incremento di dimensione derivante dalla distributività;

6. **Forma normale negativa (*NNF*)**: propagazione delle negazioni a ridosso degli atomi mediante equivalenze logiche e dualità dei quantificatori;

7. **Skolemizzazione (*Skolemization*)**: sostituzione delle variabili quantificate esistenzialmente con nuove costanti o funzioni di Skolem parametrizzate sulle variabili universali dominanti;

8. **Clausificazione (*Clausification*)**: eliminazione dei quantificatori universali e distribuzione delle disgiunzioni sulle congiunzioni per produrre oggetti `Clause`.

1.2.3 Kernel e Architettura Given Clause

Il kernel di saturazione implementa varianti della ***Given Clause Architecture*** (GCA). Lo spazio delle clausole è partizionato in due insiemi disgiunti:

- **Insieme Passivo ($\mathcal{P}$)**: contiene le clausole iniziali e i resolventi generati in attesa di selezione;

- **Insieme Attivo ($\mathcal{A}$)**: memorizza le clausole già selezionate, impiegate come premesse per le inferenze con ciascuna nuova clausola estratta.

A ogni iterazione, una clausola viene selezionata da $\mathcal{P}$ secondo una specifica euristica e designata come *given clause*. Tale clausola genera nuove inferenze combinandosi con le clausole presenti in $\mathcal{A}$. Se viene derivata la clausola vuota $\square$, il problema è dichiarato insoddisfacibile. Se invece l'insieme passivo si esaurisce senza produrre una contraddizione, il procedimento raggiunge la saturazione. Nell'ipotesi

che il calcolo sia refutazionalmente completo e che le inferenze necessarie siano state preservate, è possibile concludere che l'insieme iniziale è soddisfacibile.

Le operazioni di inferenza si distinguono in:

- **Inferenze generatrici**: produzione di nuove clausole a partire da premesse attive (es. risoluzione, superposizione);
- **Inferenze semplificatrici**: eliminazione o contrazione di clausole ridondanti mediante sussunzione diretta (*forward subsumption*) e retroattiva (*backward subsumption*).

1.2.4 Strategie di Saturazione: Otter, Discount e LRS

Vampire supporta diverse strategie di saturazione basate sul paradigma given-clause. Tra queste, Otter, Discount e Limited Resource Strategy (LRS) si distinguono principalmente per la gestione delle clausole e delle risorse disponibili.

Algoritmo Otter. La clausola selezionata viene semplificata rispetto agli insiemi $\mathcal{A}$ e $\mathcal{P}$, e a sua volta viene utilizzata per semplificare gli elementi di entrambi gli insiemi. L'inter-riduzione completa riduce la cardinalità dello spazio clausale ma introduce un sovraccarico computazionale correlato alla dimensione di $\mathcal{P}$.

Algoritmo Discount. Le semplificazioni sono limitate all'interazione tra la clausola selezionata e l'insieme attivo $\mathcal{A}$. L'insieme passivo non viene modificato retroattivamente, riducendo il costo computazionale per ciclo a fronte del mantenimento temporaneo di clausole potenzialmente ridondanti in $\mathcal{P}$.

Limited Resource Strategy (LRS). LRS adatta l'elaborazione al tempo residuo assegnato al processo: stimando la probabilità che ciascuna clausola in $\mathcal{P}$ venga estratta prima del raggiungimento del tempo limite, la strategia scarta preventivamente le clausole valutate non raggiungibili. Tale potatura introduce l'incompletezza teorica della ricerca a fronte di un tasso di successo refutazionale misurato empiricamente su vincoli temporali rigidi.

1.2.5 Splitting e Architettura AVATAR

Quando una clausola C è formata da sotto-clausole disgiunte che non condividono variabili libere ($C = C_1(x) \vee C_2(y)$ con $\mathrm{Var}(C_1) \cap \mathrm{Var}(C_2) = \emptyset$), il calcolo per superposizione tende a produrre combinazioni ridondanti.

L'architettura AVATAR (*Advanced Vampire Architecture for Theories And Resolution*) gestisce tali clausole disaccoppiando le componenti del primo ordine dalla componente booleana mediante l'integrazione di un SAT solver:

1. Ciascuna componente C_i senza variabili condivise viene associata a una variabile proposizionale $[C_i]$;
2. La clausola del primo ordine viene mappata nella clausola proposizionale corrispondente $[C_1] \vee [C_2]$ e inserita nel SAT solver;

3. Il modello proposizionale generato dal SAT solver determina quali componenti del primo ordine attivare come assunzioni temporanee nel kernel;

4. Qualora il kernel derivi una refutazione condizionata da tali assunzioni, una clausola di conflitto proposizionale viene restituita al SAT solver per escludere la configurazione.

1.3 Il Frammento a Due Variabili con Uguaglianza

1.3.1 Definizione Sintattica di $\mathrm{FO}^2_=$

Il frammento a due variabili con uguaglianza, denotato con $\mathrm{FO}^2_=$, è il sottoinsieme della logica del primo ordine soggetto ai seguenti vincoli sintattici:

1. **Segnatura priva di funzioni di arietà superiore a zero**: la segnatura ammette simboli di predicato di arietà arbitraria e costanti individuali, escludendo simboli di funzione con arietà $n \geq 1$;

2. **Restrizione sulle variabili**: le formule contengono occorrenze di al più due variabili distinte, indicate con x e y;

3. **Riuso delle variabili**: quantificatori universali ed esistenziali possono essere innestati riutilizzando le medesime variabili;

4. **Simbolo di uguaglianza**: il simbolo binario di uguaglianza ($=$) è ammesso sia in letterali positivi ($s = t$) sia negativi ($s \neq t$), con semantica di identità sul dominio.

1.3.2 Proprietà del Modello Limitato e Complessità

La proprietà del modello finito (*finite model property*) garantisce che ogni formula soddisfacibile ammetta un modello finito; tuttavia, essa non fornisce necessariamente un limite calcolabile sulla dimensione di tale modello. In assenza di questo limite, l'enumerazione delle interpretazioni su domini finiti crescenti consentirebbe di accertare la soddisfacibilità, ma non permetterebbe, da sola, di concludere l'insoddisfacibilità in un numero finito di passi.

Nel caso del frammento $\mathrm{FO}^2_=$, la decisione mediante enumerazione finita è resa possibile dalla *proprietà del modello limitato* (*bounded model property*): per ogni formula soddisfacibile è noto un limite superiore calcolabile sulla dimensione di un modello che la soddisfa.

La decidibilità del frammento privo di uguaglianza (FO^2) è stata stabilita da Scott, mentre la proprietà del modello finito per $\mathrm{FO}^2_=$ è stata dimostrata da Mortimer. Successivamente, Grädel, Kolaitis e Vardi hanno approfondito i risultati relativi alla dimensione dei modelli e alla complessità del problema di soddisfacibilità [GKV97].

In particolare, per ogni formula soddisfacibile $\phi \in \mathrm{FO}^2_=$ di lunghezza sintattica $n = |\phi|$, esiste un modello il cui dominio D soddisfa la condizione:

$$|D| \leq 2^{\mathcal{O}(n)}. \tag{1.1}$$

Questo limite permette di restringere la ricerca a strutture finite di dimensione al più esponenziale rispetto alla lunghezza della formula. Il problema della soddisfacibilità per $\mathrm{FO}^2_=$ è inoltre NEXPTIME-completo, come stabilito dai risultati di complessità riportati in [GKV97].

1.3.3 Forma normale di Scott

L'elaborazione delle formule di $\mathrm{FO}^2_=$ fa riferimento alla *forma normale di Scott* (*Scott Normal Form*).

Ogni formula $\phi \in \mathrm{FO}^2_=$ può essere convertita, preservando l'equisoddisfacibilità, in una formula ϕ' costituita da una congiunzione di clausole strutturate secondo tre prefissi di quantificazione:

$$\phi' = \bigwedge_i \beta_i$$

dove ogni congiunto β_i rispetta uno dei seguenti schemi:

- **Schema 1**: $\exists x\ \alpha_i(x)$
- **Schema 2**: $\forall x \exists y\ \alpha_i(x, y)$
- **Schema 3**: $\forall x \forall y\ \alpha_i(x, y)$

in cui α_i rappresenta una formula priva di quantificatori in forma normale congiuntiva (CNF).

La trasformazione si basa sull'applicazione del ***naming* strutturale**: sottoformule complesse ψ quantificate esistenzialmente vengono sostituite da predicati ausiliari p_{def}, introducendo le corrispondenti definizioni universali. Questo procedimento garantisce una crescita della dimensione della formula contenuta entro limiti polinomiali.

1.3.4 Risoluzione e Problematiche legate all'Uguaglianza

Sebbene la proprietà del modello limitato garantisca la possibilità teorica di decidere la soddisfacibilità mediante la ricerca di modelli finiti, un approccio basato sull'enumerazione esaustiva delle strutture può risultare computazionalmente molto oneroso.

Per il frammento privo di uguaglianza, opportune strategie di risoluzione ordinata consentono di ottenere una procedura decisionale terminante. L'introduzione dell'uguaglianza rende più complesso il controllo delle inferenze: il trattamento dell'identità mediante le regole ordinarie non preserva necessariamente le condizioni che garantiscono la finitezza dello spazio di ricerca. Si rende quindi necessario un procedimento specifico che permetta di ricondurre la formula a un frammento privo di uguaglianza prima di applicare la risoluzione.

La decisione corretta di $\mathrm{FO}^2_=$ all'interno di motori a risoluzione richiede pertanto procedure dedicate, volte a disaccoppiare l'eliminazione dell'uguaglianza dalle fasi di saturazione vincolata. La trattazione formale della procedura a due stadi proposta da de Nivelle e Pratt-Hartmann [NP01], comprendente le riscritture di riduzione e la definizione del calcolo di risoluzione indicizzata, costituisce l'oggetto del Capitolo successivo.

— 2 —

Descrizione della Procedura di Decisione per $\mathrm{FO}^2_=$

In questo capitolo viene presentata la formulazione teorica e algoritmica della procedura di decisione per il frammento a due variabili con uguaglianza ($\mathrm{FO}^2_=$). La procedura segue l'approccio proposto da de Nivelle e Pratt-Hartmann [NP01] e si articola in due fasi successive. La prima elimina il simbolo di uguaglianza, trasformando la formula iniziale in una formula equisoddisfacibile del frammento privo di uguaglianza. La seconda applica un calcolo di risoluzione indicizzata a una classe di clausole ottenuta dalla formula trasformata.

2.1 Panoramica della Procedura a Due Stadi

Il flusso di derivazione trasforma progressivamente la formula di partenza secondo lo schema illustrato in Figura 2.1.

Ad ogni passaggio è fondamentale la conservazione dell'equisoddisfacibilità: se la formula originaria ϕ ammette un modello, ciascuna formula intermedia ϕ_k ammette a sua volta un modello ottenuto espandendo opportunamente l'interpretazione dei simboli relazionali freschi introdotti.

2.2 La Trasformazione per l'Eliminazione dell'Uguaglianza

2.2.1 Lemma 1: Eliminazione delle Costanti Individuali

L'eliminazione delle costanti individuali costituisce il primo passaggio della procedura. Le costanti vengono sostituite mediante predicati unari ausiliari, associati a vincoli di esistenza e unicità. Questa trasformazione permette di lavorare con formule prive di costanti, semplificando le successive operazioni di risoluzione ristretta.

Atom	*Replacement formula*
$p(d)$	$\exists x(p(x) \wedge p_d(x))$
$r(d,x)$	$\exists y(r(y,x) \wedge p_d(y))$
$r(d,y)$	$\exists x(r(x,y) \wedge p_d(x))$
$r(x,d)$	$\exists y(r(x,y) \wedge p_d(y))$
$r(y,d)$	$\exists x(r(y,x) \wedge p_d(x))$
$r(d,d')$	$\exists x \exists y(r(x,y) \wedge p_d(x) \wedge p_{d'}(y))$

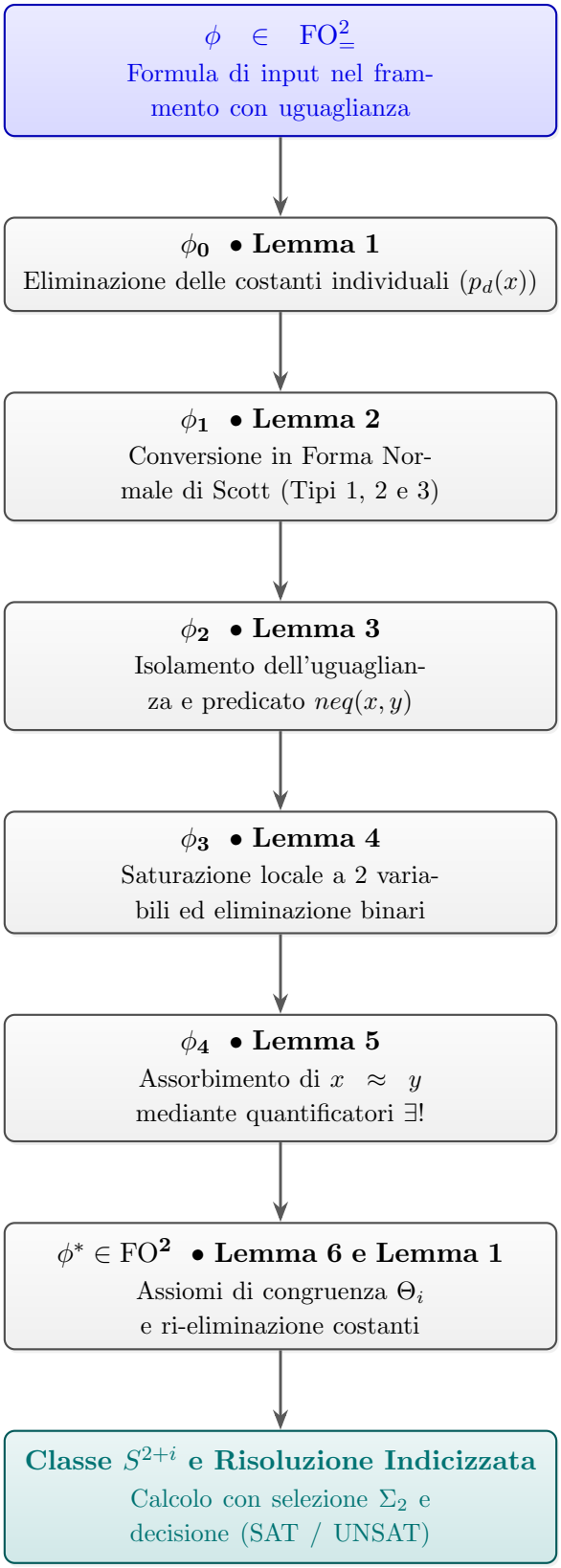


Figura 2.1: Pipeline concettuale della procedura di decisione per $FO^2_=$.

dove ogni p_d è un simbolo predicativo unario fresco. Allora ϕ è equisoddisfacibile con la formula:

$$\phi_0 := \phi' \wedge \bigwedge_{d \in \mathcal{C}} \exists! x \; p_d(x) \tag{2.1}$$

L'operatore $\exists! x \; \psi(x)$ abbrevia formalmente l'asserzione di esistenza e unicità:

$$\exists x \; (\psi(x) \wedge \forall y \; (\psi(y) \rightarrow x = y))$$

2.2.2 Lemma 2: Forma Normale di Scott

Per disciplinare la struttura delle alternanze dei quantificatori, la formula ϕ_0 viene ricondotta alla forma normale definita da Scott, che isola la complessità sintattica entro tre schemi canonici di prefisso.

Forma Normale di Scott: Sia ϕ una formula di $\mathrm{FO}^2_=$ priva di costanti. Esiste una formula equisoddisfacibile ϕ_1 avente la struttura:

$$\phi_1 = \beta_1 \wedge \cdots \wedge \beta_n \tag{2.2}$$

in cui ciascun congiunto β_i appartiene rigorosamente a una delle seguenti tre classi:

- **Tipo 1**: $\exists x \; \alpha_i(x)$
- **Tipo 2**: $\forall x \exists y \; \alpha_i(x, y)$
- **Tipo 3**: $\forall x \forall y \; \alpha_i(x, y)$

dove ogni α_i è una formula proposizionale priva di quantificatori espressa in forma normale congiuntiva (CNF).

La riduzione si basa sul *naming* ricorsivo: ogni sottoformula quantificata complessa non atomica $\psi(\bar{z})$ (con $\bar{z} \subseteq \{x, y\}$) posta sotto uno scope esistenziale viene rimpiazzata da un atomo fresco $P_\psi(\bar{z})$, aggiungendo alla formula complessiva l'assioma di definizione:

$$\forall \bar{z} \; (P_\psi(\bar{z}) \leftrightarrow \psi(\bar{z}))$$

2.2.3 Lemma 3: Spostamento dell'Uguaglianza

Una volta ottenuta la forma normale di Scott, le occorrenze atomiche di identità vengono isolate all'esterno dei corpi congiuntivi α_i.

Isolamento dell'Uguaglianza: Sia ϕ_1 in forma normale di Scott. È possibile derivare una formula equisoddisfacibile $\phi_2 = \beta'_1 \wedge \cdots \wedge \beta'_n$ tale che ciascun blocco assuma la forma:

- **Tipo 1**: $\exists x \; \alpha_i(x)$
- **Tipo 2**: $\forall x \exists y \; (x \not\approx y \wedge \alpha_i(x, y))$
- **Tipo 3**: $\forall x \forall y \; (x \approx y \vee \alpha_i(x, y))$

in cui ciascuna matrice α_i è priva di simboli di uguaglianza. Inoltre, tutte le formule di Tipo 3 possono essere fuse in un'unica espressione universale congiunta:

$$\forall x \forall y \ (x \approx y \vee \alpha_t(x, y)) \tag{2.3}$$

Le occorrenze negative dell'uguaglianza $x \neq y$ nei corpi clausali vengono rimosse introducendo un nuovo predicato binario $neq(x, y)$ corredato dall'assioma di irriflessività $\forall x \ (\neg neq(x, x))$.

2.2.4 Lemma 4: Risoluzione Ristretta

Il passaggio centrale della rimozione dell'uguaglianza consiste nella semplificazione della formula congiunta di Tipo 3 mediante una saturazione parziale.

Risoluzione Ristretta a Due Variabili: Siano $\beta(x, y) \vee C_1$ e $\neg\beta(x, y) \vee C_2$ due disgiunzioni appartenenti alle matrici di ϕ_2, dove β è un atomo binario non identitario contenente entrambe le variabili. Il risolvente ristretto è definito come la disgiunzione:

$$(C_1 \vee C_2)\sigma$$

dove σ ammette unicamente lo scambio di variabili $(x \leftrightarrow y)$ ma vieta istanziazioni con costanti o funzioni. L'inferenza è consentita:

1. tra coppie di disgiunzioni interne alla formula di Tipo 3 (aggiungendo il risultato alla matrice di Tipo 3);

2. tra una disgiunzione di Tipo 2 e una di Tipo 3 (aggiungendo il risultato alla formula di Tipo 2 coinvolta).

Eliminazione dei Letterali Binari Sia ϕ_2 la formula di partenza e sia ϕ_3 la sua chiusura finita sotto la regola di risoluzione ristretta. Sia ϕ_4 la formula ottenuta espungendo dalla matrice universale di Tipo 3 tutte le disgiunzioni che contengono ancora predicati a due variabili. Allora:

$$\phi_3 \text{ ammette un modello} \iff \phi_4 \text{ ammette un modello}$$

La dimostrazione semantica poggia sulla possibilità di ridefinire, per ogni coppia di elementi distinti (b_1, b_2) nel dominio di un modello di ϕ_4, l'interpretazione dei predicati binari (*pattern*) in modo da rendere vere le clausole rimosse senza invalidare i vincoli di Tipo 2, proprietà assicurata dalla chiusura rispetto ai risolventi misti Tipo 2 / Tipo 3.

2.2.5 Lemma 5: Trasformazione dell'Uguaglianza Positiva

A seguito del Lemma 4, la formula universale di Tipo 3 contiene disgiunzioni costituite unicamente da atomi unari e dall'uguaglianza positiva.

Decomposizione delle Disgiunzioni Universali: Siano $\gamma(x)$ e $\delta(y)$ formule a una sola variabile libera. La formula universale:

$$\forall x \forall y \ (\gamma(x) \vee \delta(y) \vee x \approx y) \tag{2.4}$$

è logicamente equivalente a:

$$\forall x\ \gamma(x) \vee \forall x\ \delta(x) \vee (\exists! x\ \neg\gamma(x) \wedge \forall x\ (\gamma(x) \leftrightarrow \delta(x))) \tag{2.5}$$

La validità dell'equivalenza scaturisce dall'analisi dei modelli: affinché la disgiunzione $(\gamma(x) \vee \delta(y) \vee x \approx y)$ sia soddisfatta per ogni scelta di x e y, se esistono elementi che falsificano γ e δ, tali elementi non possono essere più di uno e devono coincidere, in questo modo, la condizione espressa dall'uguaglianza viene ricondotta a un vincolo di unicità.

2.2.6 Lemma 6: Eliminazione dei Quantificatori di Unicità

Le occorrenze residue di identità sono ora interamente confinate all'interno di formule quantificate con $\exists! x\ \zeta_i(x)$.

Eliminazione di $\exists!$ Mediante Congruenza: Sia η una congiunzione di formule chiuse di FO^2 e siano $\zeta_1, \ldots, \zeta_m$ sottoformule univariete prive di quantificatori e costanti. La formula:

$$\psi := \eta \wedge \bigwedge_{1 \le i \le m} \exists! x\ \zeta_i(x) \tag{2.6}$$

è equisoddisfacibile con:

$$\psi^* := \eta \wedge \bigwedge_{1 \le i \le m} \Theta_i \tag{2.7}$$

dove ogni Θ_i è definito introducendo una nuova costante individuale e_i e ponendo:

$$\begin{aligned} \Theta_i := \zeta_i(e_i) &\wedge \bigwedge_{p \in \mathcal{P}} \forall x\ (\zeta_i(x) \rightarrow (p(x) \leftrightarrow p(e_i))) \\ &\wedge \bigwedge_{q \in \mathcal{R}} \forall x \forall y\ (\zeta_i(x) \rightarrow (q(x,y) \leftrightarrow q(e_i,y))) \\ &\wedge \bigwedge_{q \in \mathcal{R}} \forall x \forall y\ (\zeta_i(x) \rightarrow (q(y,x) \leftrightarrow q(y,e_i))) \end{aligned}$$

con p che varia sui predicati unari e q sui predicati binari del linguaggio.

In assenza del simbolo di uguaglianza, gli assiomi di congruenza Θ_i impongono che ogni elemento che soddisfa ζ_i sia equivalente alla costante $\rceil_i$ rispetto a tutti i predicati unari e binari della formula. L'applicazione finale del Lemma 1 rimuove le costanti e_i appena introdotte, restituendo una formula finale $\phi^* \in \mathrm{FO}^2$ rigorosamente priva di identità.

2.3 La Classe S^{2+i}

A seguito della Skolemizzazione e della clausificazione della formula ϕ^*, l'insieme di clausole ottenuto appartiene a una classe sintattica ristretta denominata S^2, estesa mediante un sistema di indicizzazione numerica.

Classe S^2: Una clausola C appartiene alla classe S^2 se soddisfa le seguenti proprietà:

1. C contiene al massimo due variabili distinte e non presenta simboli funzionali innestati (termini *shallow*);
2. Se C contiene letterali ground, allora C è interamente ground;
3. Ogni termine funzionale non ground in C contiene tutte le variabili che occorrono nella clausola;
4. Esiste almeno un letterale in C che contiene congiuntamente tutte le variabili della clausola.

Per guidare la selezione senza violare la stabilità rispetto alle sostituzioni, de Nivelle e Pratt-Hartmann introducono un'indicizzazione binaria sui letterali, formalizzando la classe S^{2+i}.

Classe Indicizzata S^{2+i}: Un letterale indicizzato è una coppia $L : a$ con $a \in \{0, 1\}$. Una clausola indicizzata appartiene alla classe S^{2+i} se rispetta i vincoli di S^2 unitamente alle condizioni:

1. Tutti i simboli di funzione presenti sono unari ($f(x)$);
2. Se la clausola contiene due variabili distinte, i letterali che contengono entrambe le variabili hanno indice 1, mentre tutti gli altri hanno indice 0.

2.4 Il Calcolo di Risoluzione Indicizzata

2.4.1 Ordinamento sui Termini e Funzione di Selezione Σ_2

L'adozione di indici numerici consente di superare l'impossibilità teorica di definire un ordinamento liftabile che separi atomi aventi le stesse variabili (quali $p(x, y)$ e $p(x, x)$), i quali al livello ground ammettono istanze comuni come $p(a, a)$.

Ordinamento Ground $<_2$: Siano $A : a$ e $B : b$ due letterali indicizzati ground. L'ordinamento $<_2$ è definito come segue:

1. Se la profondità sintattica del termine di A è strettamente minore di quella di B, allora $A : a <_2 B : b$;
2. Se A e B hanno identica profondità sintattica e $a < b$, allora $A : a <_2 B : b$.

L'ordinamento $<_2$ induce per sollevamento (*lifting*) la funzione di selezione ammissibile Σ_2 operante sulle clausole arbitrarie con variabili:

Funzione di Selezione Σ_2 Data una clausola indicizzata $C \in S^{2+i}$, un letterale $A : a \in C$ è selezionato ($A : a \in \Sigma_2(C)$) a meno che non sia verificata almeno una delle seguenti condizioni inibitorie:

1. A è privo di termini funzionali, possiede indice $a = 0$, ed esiste nella clausola un letterale $B : b$ avente indice $b = 1$;
2. $A : a$ è privo di termini funzionali, ed esistono nella clausola altri letterali contenenti termini funzionali non ground.

2.4.2 Regole del Calcolo

Il calcolo di saturazione opera mediante le seguenti regole:

Risoluzione indicizzata. Siano $C_1 \vee A_1 : a_1$ e $C_2 \vee \neg A_2 : a_2$ due clausole indicizzate, nelle quali i letterali complementari sono selezionati dalla funzione Σ_2. Se gli atomi A_1 e A_2 sono unificabili, sia $\sigma = \mathrm{mgu}(A_1, A_2)$. Il risolvente è:

$$\frac{C_1 \vee A_1 : a_1 \qquad C_2 \vee \neg A_2 : a_2}{(C_1 \vee C_2)\sigma}$$

I letterali residui conservano i rispettivi indici, mentre la sostituzione σ agisce sui termini.

Fattorizzazione indicizzata. Sia $C \vee A_1 : a_1 \vee A_2 : a_2$ una clausola indicizzata, in cui $A_1 : a_1$ è selezionato da Σ_2. Se gli atomi A_1 e A_2 sono unificabili, sia $\sigma = \mathrm{mgu}(A_1, A_2)$. Il fattore è:

$$\frac{C \vee A_1 : a_1 \vee A_2 : a_2}{(C \vee A_2 : a_2)\sigma}$$

Anche in questo caso, la sostituzione agisce sui termini e gli indici vengono conservati.

Sussunzione Indicizzata. Una clausola C_1 sussume C_2 ($C_1 \sqsubseteq C_2$) se esiste una sostituzione θ tale che:

$$(L : a)\theta \in C_2 \quad \text{per ogni } L : a \in C_1$$

La corrispondenza esplicita degli indici impedisce eliminazioni non conformi alle restrizioni di selezione.

Splitting. La regola di splitting si applica a una clausola che può essere suddivisa in due componenti non vuote prive di variabili condivise.

Data una clausola della forma

$$C = R_1(x) \vee R_2(y),$$

con $\mathrm{Var}(R_1) \cap \mathrm{Var}(R_2) = \emptyset$, la ricerca può essere suddivisa in due rami indipendenti, uno per ciascuna componente. la soddisfacibilità dell'insieme iniziale dipende dalla gestione corretta dei rami della ricerca.

2.5 Correttezza, Completezza e Garanzia di Terminazione

La validità della procedura come algoritmo decisionale è stabilita dal seguente risultato formale.

Decidibilità di $\mathrm{FO}^2_=$: Sia $\phi \in \mathrm{FO}^2_=$.

1. La trasformazione da ϕ a ϕ^* preserva l'equisoddisfacibilità;
2. La chiusura dell'insieme clausale S^{2+i} generato da ϕ^* sotto il calcolo di risoluzione indicizzata termina sempre in tempo finito;

3. ϕ è insoddisfacibile se e solo se la saturazione deriva la clausola vuota $\Box$.

Dimostrazione (Terminazione)

Le proprietà precedenti garantiscono che, per una segnatura finita, il numero di clausole distinte generabili nella classe S^{2+i}, a meno di rinominazione delle variabili, è finito.

Una procedura di saturazione che evita la rielaborazione indefinita di clausole già considerate può quindi raggiungere la saturazione in un numero finito di passi.

La completezza refutazionale del calcolo, insieme alla preservazione dell'equisoddisfacibilità nelle trasformazioni precedenti, permette di decidere la soddisfacibilità della formula iniziale.

Queste garanzie riguardano la procedura teorica descritta da de Nivelle e Pratt-Hartmann [NP01].

— 3 —

Progettazione e architettura

Il capitolo descrive le scelte architetturali adottate per integrare in Vampire la procedura specializzata per il frammento a due variabili con uguaglianza, $\mathrm{FO}^2_=$. La procedura teorica, presentata nel Capitolo 2, richiede di trasformare le formule con uguaglianza e di applicare successivamente un calcolo di risoluzione indicizzata. Dal punto di vista software, queste operazioni devono essere coordinate con la rappresentazione dei problemi e con le fasi di elaborazione già disponibili nel dimostratore.

La progettazione è organizzata attorno a quattro componenti principali: classificazione del problema, eliminazione dell'uguaglianza, preparazione delle clausole e risoluzione specializzata. La separazione delle responsabilità consente di distinguere i controlli di ammissibilità dalle trasformazioni logiche e dalle operazioni di ricerca. In questo capitolo sono descritti i compiti dei componenti, i dati scambiati e i vincoli che regolano le loro interazioni. Le scelte implementative e gli algoritmi delle singole operazioni sono trattati nel capitolo successivo.

3.1 Requisiti e vincoli progettuali

L'estensione è stata progettata per lavorare su problemi riconducibili a $\mathrm{FO}^2_=$ e per utilizzare un percorso di elaborazione distinto dalla saturazione generale di Vampire. I requisiti principali riguardano il riconoscimento del frammento, la preservazione delle proprietà logiche durante le trasformazioni e il rispetto delle condizioni richieste dal calcolo specializzato.

Classificazione dell'input. Prima dell'applicazione della procedura, il sistema deve accertare che il problema soddisfi i vincoli sintattici del frammento: uso di al più due variabili distinte e assenza di simboli di funzione di arità positiva nella formula iniziale. Deve inoltre rilevare la presenza dell'uguaglianza, per stabilire se sia necessaria la fase dedicata alla sua eliminazione.

Preservazione dell'equisoddisfacibilità. Le trasformazioni devono preservare la soddisfacibilità della formula di partenza. Il sistema può introdurre simboli ausiliari e modificare la rappresentazione delle formule, purché il problema trasformato sia equisoddisfacibile a quello iniziale.

Compatibilità con Vampire. I componenti devono interagire con le rappresentazioni di formule, clausole e problemi messe a disposizione dal dimostratore, limitando la necessità di conversioni tra formati intermedi.

Preparazione del calcolo specializzato. Prima della saturazione, le clausole devono rispettare i vincoli strutturali della classe S^{2+i} descritta nel capitolo pre-

cedente. L'indicizzazione dei letterali e la funzione di selezione Σ_2 costituiscono parte integrante del contratto tra la preparazione delle clausole e il motore inferenziale.

Separazione della ricerca specializzata. Le inferenze del decisore devono essere gestite da un sottosistema dedicato, distinto dalle strategie generali di superposizione. Questa scelta consente di applicare le restrizioni proprie del calcolo per S^{2+i} senza attribuirle al motore generale.

Gestione dell'esito. L'interfaccia del decisore deve distinguere gli esiti `SATISFIABLE` e `UNSATISFIABLE` dai casi in cui non è possibile fornire un risultato conclusivo, rappresentati da `UNKNOWN`. Quest'ultimo esito riguarda le condizioni operative dell'estensione e non modifica le proprietà teoriche della procedura.

Questi requisiti definiscono le responsabilità dei componenti e le condizioni necessarie per il passaggio da una fase alla successiva. In particolare, la validazione dell'appartenenza al frammento e la validazione delle clausole svolgono ruoli diversi: la prima riguarda la sintassi del problema iniziale, la seconda la forma richiesta dal calcolo di risoluzione.

3.2 Architettura generale e flusso di elaborazione

L'architettura segue una pipeline nella quale ogni fase riceve il risultato della precedente e prepara i dati necessari alla successiva. Il problema viene acquisito tramite l'infrastruttura di parsing di Vampire e sottoposto alla classificazione. In presenza di uguaglianza si attiva il componente che applica le trasformazioni previste dalla procedura di de Nivelle e Pratt-Hartmann; in sua assenza, questa fase non è necessaria. Il problema viene quindi clausificato e verificato rispetto ai vincoli della classe clausale di destinazione, prima di essere affidato al motore di saturazione specializzato.

La Figura 3.1 rappresenta i quattro componenti logici dell'estensione e il loro rapporto con l'infrastruttura di Vampire. Le frecce indicano il passaggio del problema tra le fasi, non una dipendenza da specifiche classi o funzioni del codice sorgente.

Lo schema rappresenta il flusso degli input ammessi dalla procedura. L'eventuale trattamento di un input non riconosciuto come appartenente al frammento è distinto dalla pipeline di decisione e dipende dalla modalità con cui l'estensione è richiamata. Questa distinzione evita di attribuire al motore specializzato la responsabilità di risolvere problemi al di fuori del suo dominio.

3.3 Scomposizione in componenti

La suddivisione in componenti segue le principali responsabilità della procedura. Ogni componente espone alla fase successiva una rappresentazione del problema con proprietà più specifiche rispetto a quella ricevuta in ingresso. La separazione ha inoltre lo scopo di mantenere le trasformazioni dell'uguaglianza indipendenti dalla gestione della saturazione.

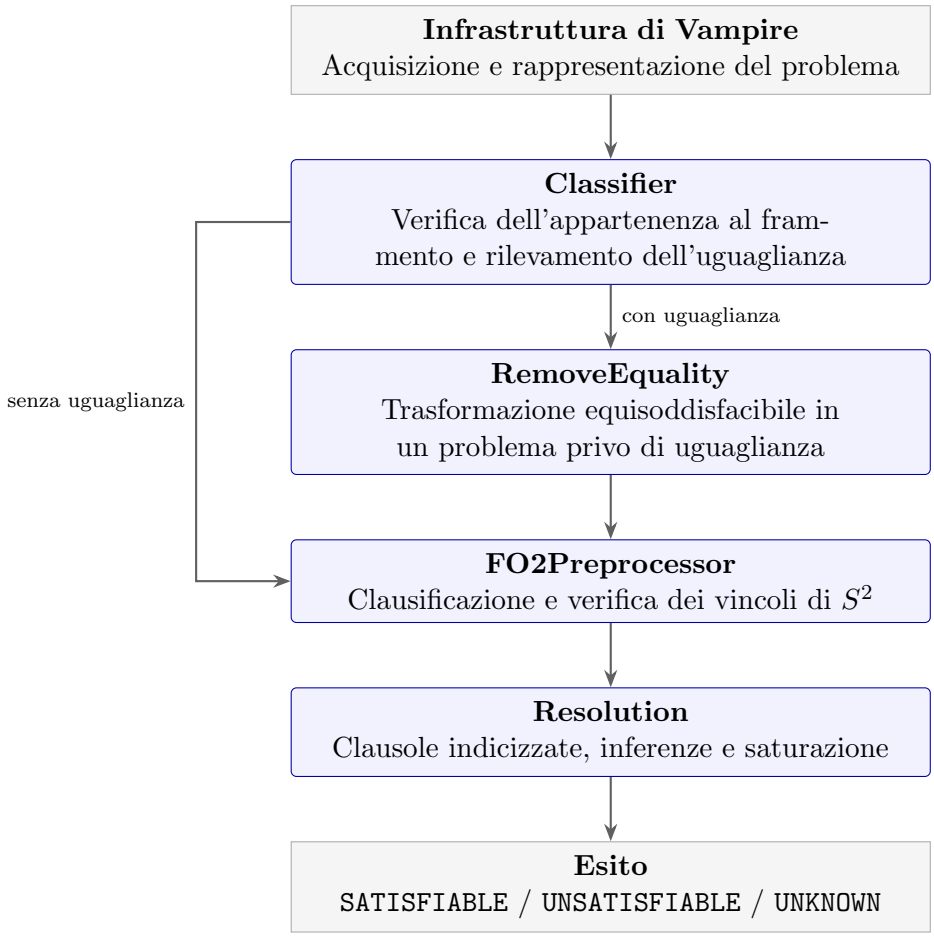


Figura 3.1: Scomposizione funzionale dell'estensione per $\mathrm{FO}^2_=$ e flusso principale di elaborazione. Lo schema rappresenta il percorso degli input riconosciuti come appartenenti al frammento.

3.3.1 Classificazione del frammento

Il componente `Classifier` costituisce il punto di ingresso della procedura specializzata. Il suo compito è analizzare il problema e stabilire se la struttura sintattica delle formule sia compatibile con $\mathrm{FO}^2_=$. I controlli riguardano il numero di variabili distinte, i simboli funzionali ammessi e la presenza di uguaglianze e disuguaglianze.

Il risultato della classificazione comprende dunque due informazioni concettualmente distinte: l'ammissibilità del problema e la necessità di applicare la trasformazione per l'eliminazione dell'uguaglianza. Quest'ultima informazione permette di evitare una fase di elaborazione non necessaria per gli input già privi del simbolo di identità.

La responsabilità del classificatore termina con l'analisi dell'input: il componente non applica le trasformazioni dei lemmi e non genera inferenze. Tale separazione evita di sovrapporre il riconoscimento del dominio di applicazione con la modifica delle formule.

3.3.2 Eliminazione dell'uguaglianza

Il componente `RemoveEquality` è responsabile della prima fase della procedura decisionale. Riceve un problema appartenente al frammento con uguaglianza e lo trasforma in un problema equisoddisfacibile privo del simbolo di identità. Dal punto di vista progettuale, il componente raccoglie le trasformazioni descritte dai sei lemmi richiamati nel Capitolo 2.

La scelta di isolare questa responsabilità in un componente autonomo risponde a due esigenze. In primo luogo, le riscritture agiscono su formule e vincoli logici che precedono la saturazione del problema. In secondo luogo, il motore di risoluzione può essere progettato per operare su clausole prive di uguaglianza, senza incorporare direttamente le trasformazioni necessarie a eliminarla.

Il contratto in uscita del componente richiede che il problema trasformato conservi la soddisfacibilità dell'input. Le eventuali costanti e i predicati ausiliari introdotti durante la trasformazione fanno parte della rappresentazione intermedia e devono essere gestiti dalle successive fasi di preparazione, secondo le condizioni stabilite dalla procedura teorica.

3.3.3 Preparazione e validazione delle clausole

Il componente `FO2Preprocessor` collega la trasformazione delle formule al calcolo di risoluzione. La sua responsabilità è preparare il problema in forma clausale e accertare che le clausole soddisfino le condizioni strutturali richieste dalla successiva fase inferenziale.

La preparazione comprende le trasformazioni logiche necessarie alla clausificazione, mentre la validazione prende come riferimento le proprietà della classe S^2: numero limitato di variabili, assenza di termini funzionali annidati, vincoli sui letterali ground e copertura delle variabili da parte dei termini e dei letterali previsti dalla definizione. La distinzione tra trasformazione e validazione è importante: la prima produce la rappresentazione destinata al calcolo, la seconda verifica che tale rappresentazione rispetti le precondizioni del decisore.

Il preprocessore si colloca dopo l'eventuale eliminazione dell'uguaglianza e prima dell'introduzione delle informazioni di indicizzazione. In questo modo, il motore inferenziale riceve clausole già predisposte per l'applicazione del calcolo specializzato.

3.3.4 Rappresentazione indicizzata e calcolo di risoluzione

Il sottosistema `Resolution` comprende la rappresentazione delle clausole indicizzate, le operazioni inferenziali e la gestione della ricerca. La progettazione distingue queste responsabilità, pur mantenendole all'interno della stessa fase logica della pipeline.

Rappresentazione delle clausole. Le clausole di VAMPIRE non contengono, nella loro rappresentazione ordinaria, tutte le informazioni richieste dal calcolo S^{2+i}. È quindi prevista una rappresentazione che associa a ciascun letterale un indice appartenente a $\{0, 1\}$, conservando il collegamento con il letterale originario. Tale scelta permette di applicare la selezione Σ_2 senza modificare il significato logico della clausola: gli indici disciplinano le inferenze ammesse, ma non ne fanno parte della semantica.

Inferenze e semplificazioni. Il componente `FO2Inferences` raccoglie le operazioni utilizzate dal calcolo: risoluzione indicizzata, fattorizzazione, sussunzione e gestione dello splitting. La responsabilità assegnata a questo componente consiste nell'esporre operazioni che rispettino i vincoli del calcolo teorico; la definizione formale delle regole resta quella presentata nel capitolo precedente.

Coordinamento della saturazione. Il componente `FO2Solver` gestisce il processo decisionale, coordinando l'elaborazione delle clausole e richiamando le operazioni inferenziali. La separazione tra `FO2Solver` e `FO2Inferences` consente di distinguere il controllo della ricerca dalla logica delle singole inferenze.

Il risultato del processo è espresso mediante tre possibili esiti: `SATISFIABLE`, `UNSATISFIABLE` e `UNKNOWN`. I primi due corrispondono a conclusioni logiche sul problema, mentre il terzo segnala che l'elaborazione non ha prodotto un esito conclusivo nelle condizioni operative previste. Il significato e le cause concrete di quest'ultimo caso sono approfonditi nel capitolo dedicato all'implementazione.

3.4 Interazioni e contratti tra i componenti

Oltre alla suddivisione delle responsabilità, l'architettura definisce le condizioni che devono essere rispettate nei passaggi tra i componenti. Tali condizioni costituiscono il collegamento tra le proprietà teoriche della procedura e la sua organizzazione software.

I contratti descritti nella Tabella 3.1 permettono di individuare il punto nel quale verificare ciascuna proprietà. L'appartenenza al frammento viene controllata prima delle trasformazioni; l'equisoddisfacibilità deve essere preservata durante l'eliminazione dell'uguaglianza; le condizioni clausali devono essere rispettate prima della

Componente	**Input previsto**	**Condizione in uscita**
`Classifier`	Problema acquisito da Vampire	Esito dell'analisi di appartenenza a $\mathrm{FO}^2_=$ e indicazione della presenza di uguaglianza
`RemoveEquality`	Problema ammesso con uguaglianza	Problema equisoddisfacibile privo di uguaglianza
`FO2Preprocessor`	Problema del frammento privo di uguaglianza	Insieme di clausole predisposto per la verifica dei vincoli di S^2
`FO2Resolution`	Clausole conformi ai requisiti del calcolo S^{2+i}	Esito della ricerca specializzata

Tabella 3.1: Responsabilità e condizioni di interfaccia dei componenti principali.

risoluzione indicizzata. Il motore di saturazione assume quindi che l'input abbia superato le verifiche previste dalle fasi precedenti.

Questa organizzazione limita la sovrapposizione delle responsabilità tra i componenti e rende esplicite le precondizioni della procedura specializzata. Nel capitolo successivo sono descritte le strutture dati, le interfacce effettive e le soluzioni adottate per realizzare i componenti all'interno di Vampire.

— 4 —

Implementazione della Procedura in Vampire

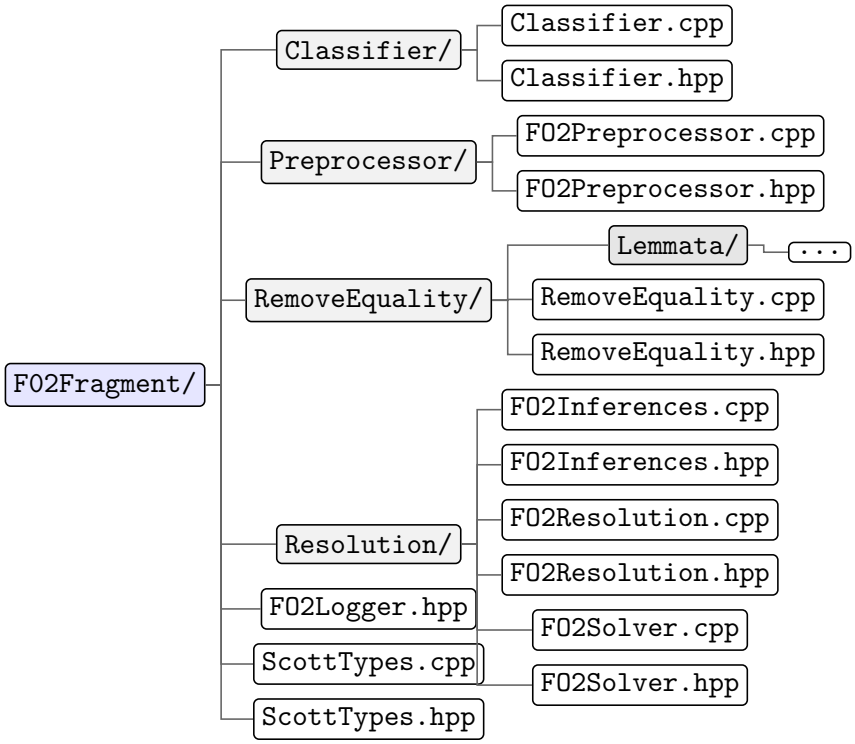


Figura 4.1: Albero strutturale dei moduli implementati all'interno della directory `F02Fragment`.

Definiti i vincoli e le specifiche architetturali nel capitolo precedente, questo capitolo descrive l'implementazione dei moduli sviluppati per il frammento $FO^2_=$ nel codice sorgente di Vampire.

La descrizione segue le trasformazioni logiche applicate alle unità di `Kernel::Problem`, dalla manipolazione delle formule fino alla saturazione delle clausole. Sono indicate anche le forme che il codice lascia invariate e i controlli che non modificano il problema.

Le sezioni seguenti descrivono il comportamento dei moduli e i passaggi eseguiti dal codice.

4.1 Modulo di Preprocessing e Validazione Sintattica

L'algoritmo principale di questo modulo è PREPROCESS (Algoritmo 1). La procedura accetta un'istanza di `Kernel::Problem`, conserva temporaneamente le clausole già presenti e converte le unità formula in clausole.

Le clausole già presenti vengono conservate e raccolte insieme ai risultati della clausificazione. Per ogni unità formula, PREPROCESS esegue in sequenza:

- semplificazione delle costanti vero/falso;
- conversione in NNF e nuova semplificazione;
- rettificazione delle variabili, appiattimento della formula;
- clausificazione con `Shell::NewCNF(1)`, che esegue anche la skolemizzazione.

La formula viene rimossa dalla lista delle unità dopo la clausificazione. Una formula ridotta a falso produce la clausola vuota; una formula ridotta a vero viene eliminata. Al termine, tutte le clausole raccolte vengono reinserite nel problema. Questa fase prepara la forma clausale usata dai controlli successivi.

4.1.1 Validazione dei Vincoli del Frammento $\mathcal{S}_2$

Terminata la clausificazione, PREPROCESS controlla tutte le clausole con VALIDATECLAUSE (Algoritmo 2). La validazione restituisce un booleano diagnostico. Se trova una violazione, la clausola resta nel problema: la funzione non applica lo splitting dell'articolo e non lancia un'eccezione.

La funzione VALIDATECLAUSE controlla i quattro vincoli della Definizione 6 dell'articolo:

1. **Vincolo sulle Variabili**: la clausola non deve contenere più di 2 variabili distinte.
2. **Proprietà dei Termini Funzionali (*Shallow* e *Covering*)**: Per ciascun argomento di ogni letterale della clausola che sia un termine composto $t = f(t_1, \ldots, t_k)$, vengono verificati due sotto-requisiti:
   - *Profondità ridotta (Shallow)*: Il termine t non deve contenere simboli di funzione nidificati (t.isShallow() = true).
   - *Copertura (Covering)*: Se t non è ground, esso deve contenere tutte le variabili presenti nella clausola complessiva. (coversAllVariables($t, clauseVars$))
3. **Controllo su Clausole Ground**: se contiene almeno un letterale senza variabili (anche proposizionale/nullario), tutti i letterali devono essere ground.
4. **Copertura Bivariata**: se la clausola contiene due variabili distinte, almeno un letterale deve contenere entrambe. Il controllo considera le variabili presenti negli argomenti del letterale, non richiede che il simbolo predicativo abbia arità esattamente due.

La verifica della copertura delle variabili è implementata da coversAllVariables (Algoritmo 3) e da containsAllVariables (Algoritmo 4).

coversAllVariables estrae con `VariableIterator` le variabili del termine funzionale e verifica che l'insieme ottenuto contenga tutte le variabili della clausola. I termini ground non richiedono questo controllo. Se una condizione fallisce, validateClause registra il motivo e restituisce `false`; la clausola resta nel problema. Il valore restituito da preprocess segnala l'esito della validazione, ma non trasforma né rifiuta le clausole non conformi e non esegue lo splitting.

Algorithm 1 FO2 Preprocessing Main Pipeline

```
    Signature: PREPROCESS : Problem& → Void
 1: function PREPROCESS(prb)
 2:     it ← UnitList::DelIterator(prb.units())
 3:     clauses ← Stack⟨Clause*⟩()
 4:     newCnf ← Shell::NewCNF(1)
 5:     allClauses ← Stack⟨Clause*⟩()
 6:     while it.hasNext() do
 7:         u ← it.next()
 8:         if ¬u → isClause() then
 9:             fu ← static_cast⟨FormulaUnit*⟩(u)
10:             fu ← simplifyFalseTrue(fu)
11:             fu ← nnf(fu)
12:             fu ← simplifyFalseTrue(fu)
13:             fu ← rectify(fu)
14:             if fu.formula() = false then
15:                 allClauses.push(emptyClause(fu)); it.del(); continue
16:             end if
17:             if fu.formula() = true then
18:                 it.del(); continue
19:             end if
20:             fu ← flatten(fu)
21:             clauses.reset()
22:             newCnf.clausify(fu, clauses)
23:             while ¬clauses.isEmpty() do
24:                 cl ← clauses.pop()
25:                 allClauses.push(cl)
26:             end while
27:             it.del()
28:         end if
29:     end while
30:     while ¬allClauses.isEmpty() do
31:         prb.units().push(allClauses.pop())
32:     end while
33:     valid ← true
34:     uit ← UnitList::Iterator(prb.units())
35:     while uit.hasNext() do
36:         u ← uit.next()
37:         if u → isClause() then
38:             cl ← static_cast⟨Clause*⟩(u)
39:             err ← NULL
40:             if ¬VALIDATECLAUSE(cl, err) then
41:                 valid ← false
42:             end if
43:         end if
44:     end while
45:     return valid
46: end function
```

Algorithm 3 Variable Coverage Check

Signature: coversAllVariables : Term $*$ $\times$DHSet$\langle$unsigned$\rangle$& $\rightarrow$ Boolean

```
1: function coversAllVariables(t, clauseVars)
2:     termVars ← ∅
3:     vit ← VariableIterator(t)
4:     while vit.hasNext() do
5:         v ← vit.next()
6:         termVars.insert(v.var())
7:     end while
8:     return containsAllVariables(termVars, clauseVars)
9: end function
```

Algorithm 2 S2 Clause Validation

Signature: validateClause : Clause $*$ $\times$String& $\rightarrow$ Boolean

```
1: function validateClause(cl, errorMessage)
2:     clauseVars ← ∅
3:     vit ← cl → getVariableIterator()
4:     while vit.hasNext() do
5:         clauseVars.insert(vit.next())
6:     end while
7:     if |clauseVars| > 2 then
8:         errorMessage ← "violates S2 variable bound (¿2 vars)"
9:         return false
10:    end if
11:    hasGroundLiteral ← false
12:    hasNonGroundLiteral ← false
13:    hasTwoVarLiteral ← false
14:    for i ← 0 to cl → length() − 1 do
15:        lit ← (∗cl)[i]
16:        litVars ← getVariables(lit)
17:        if |litVars| = 0 then
18:            hasGroundLiteral ← true
19:        else if |litVars| > 0 then
20:            hasNonGroundLiteral ← true
21:        end if
22:        if |clauseVars| = 2 ∧ |litVars| = 2 then
23:            hasTwoVarLiteral ← true
24:        end if
25:        for a ← 0 to lit → arity() − 1 do
26:            tl ← lit → nthArgument(a)
27:            if tl → isTerm() then
28:                t ← tl → term()
29:                if ¬t → isShallow() then
30:                    errorMessage ← "nested function symbols"
31:                    return false
32:                end if
33:                if ¬t → ground() ∧ ¬coversAllVariables(t, clauseVars) then
34:                    errorMessage ← "non-covering functional term"
35:                    return false
36:                end if
37:            end if
38:        end for
39:    end for
40:    if hasGroundLiteral ∧ hasNonGroundLiteral then
41:        errorMessage ← "clause containing ground literal is not entirely ground"
42:        return false
43:    end if
44:    if |clauseVars| = 2 ∧ ¬hasTwoVarLiteral then
45:        errorMessage ← "2-variable clause has no literal containing all variables"
46:        return false
47:    end if
48:    return true
49: end function
```

Algorithm 4 Variable Coverage Check 2

Signature: containsAllVariables : Set⟨Var⟩ × Set⟨Var⟩ → Boolean
1: **function** containsAllVariables($vars, required$)
2: **if** $|vars| < |required|$ **then**
3: **return false**
4: **end if**
5: **for each** $v \in required$ **do**
6: **if** $v \notin vars$ **then**
7: **return false**
8: **end if**
9: **end for**
10: **return true**
11: **end function**

4.2 Eliminazione dell'Uguaglianza: il Modulo `RemoveEquality`

La classe `RemoveEquality` coordina la trasformazione. Espande prima gli eventuali gruppi di oggetti distinti TPTP, poi usa `Classifier::isFO2` per controllare il frammento e rilevare la presenza dell'uguaglianza. Se il controllo FO^2 fallisce, la procedura termina senza applicare i Lemmi. Se l'uguaglianza non è presente, salta la riduzione.

La funzione principale `RemoveEquality::removeEquality(Kernel::Problem &prb)` orchestra l'intera trasformazione mediante la seguente sequenza di chiamate statiche:

Algorithm 5 FO2 Equality Removal Main Procedure

```
    Signature: removeEquality : Problem& → Void
 1: function removeEquality(prb)
 2:     if prb.hasDistinctGroups() then
 3:         prb ← DistinctGroupExpansion(prb)
 4:     end if
 5:     hasEq ← false
 6:     if ¬isFO2(prb.units(), hasEq) then
 7:         return
 8:     end if
 9:     if ¬hasEq ∧ ¬prb.hasEquality() then
10:         return
11:     end if
12:     prb ← normalizeEqualitiesInProblem(prb)
13:     prb ← Rectify(prb)
14:     applyLemma1(prb)
15:     applyLemma2(prb)
16:     applyLemma3(prb)
17:     applyLemma4(prb)
18:     applyLemma5(prb)
19:     applyLemma6(prb)          ▷ La sostituzione proxy dell'uguaglianza non è
    invocata.
20: end function
```

I Lemmi aggiornano o rimpiazzano le unità del problema.

4.2.1 Lemma 1: Eliminazione delle Costanti Individuali

Il Lemma 1 realizza la trasformazione della Tabella 1 dell'articolo per le forme di letterale trattate dal codice. Una mappa associa ogni costante incontrata a un predicato unario fresco p_c. In un'uguaglianza con una o due costanti, la routine introduce variabili per le posizioni costanti, aggiunge i corrispondenti atomi p_c e conserva il letterale di uguaglianza con la polarità originaria. Nei letterali predicativi di arità uno o due, le posizioni costanti vengono trattate allo stesso modo e il predicato originale mantiene polarità e simbolo. La ricorsione visita connettivi, negazioni e quantificatori. Gli atomi nullari e i letterali non di uguaglianza di arità maggiore di due restano invariati; perciò l'implementazione copre la forma prevista per il frammento, ma non riscrive termini costanti in modo generale.

L'architettura del Lemma 1 si articola su tre livelli di trasformazione:

1. **Orchestrazione del Problema (applyLemma1)**: La funzione principale scansiona tutte le unità del problema (`prb.units()`) e le suddivide in base alla loro natura:

   - **Formule (`FormulaUnit`)**: invoca la funzione replaceInFormula sulla formula radice per processarne ricorsivamente l'albero sintattico.

- **Clausole (`Clause`)**: invoca la funzione replaceInClause per convertire i letterali della clausola ed elaborarli.

Se durante la scansione sono state individuate costanti, la mappa `constantsMap` contiene l'associazione tra ciascun simbolo di costante e il predicato fresco p_c. Per ogni voce la funzione aggiunge all'inizio del problema l'assioma $\exists x(p_c(x) \wedge \forall y(p_c(y) \rightarrow x = y))$, che impone esistenza e unicità.

2. **Elaborazione delle clausole**: ogni letterale viene avvolto in una formula atomica e passato alla stessa trasformazione ricorsiva. Se almeno un letterale cambia, la clausola viene rimpiazzata da una `FormulaUnit` contenente la disgiunzione dei letterali trasformati; se nessuno cambia, l'unità clausola resta invariata.

3. **Sostituzione ricorsiva nelle formule**: attraversa gli operatori binari, le congiunzioni, le disgiunzioni, le negazioni e i quantificatori. Solo i nodi atomici vengono sottoposti ai casi della Tabella 1:

   - *Letterali di uguaglianza*: se uno o entrambi gli argomenti sono costanti, introduce una variabile per ogni posizione costante, il relativo atomo p_c e un quantificatore esistenziale. Un'uguaglianza senza costanti resta invariata.
   - *Letterali non di uguaglianza*: per arità uno e due sostituisce le posizioni costanti con variabili esistenziali e congiunge gli atomi p_c al letterale predicativo trasformato. Se non vi sono costanti o l'arità non è supportata, il letterale resta invariato.

4. **Generazione degli assiomi di unicità**: Per garantire l'equivalenza di soddisfacibilità, per ogni predicato di costante p_c introdotto viene generato l'assioma che ne impone l'esistenza e l'unicità dell'interpretazione:

$$\exists x \, (p_c(x) \wedge \forall y(p_c(y) \implies x = y))$$

Algorithm 6 Get Predicate for Constant

Signature: getPredForConstant : unsigned × DHMap& → unsigned

```
1: function getPredForConstant(constantFunctor, constants)
2:     if constants.find(constantFunctor, predicateFunctor) then
3:         return predicateFunctor
4:     end if
5:     constName ← env.signature.getFunction(constantFunctor).name()
6:     predName ← "p_" + constName
7:     predicateFunctor ← env.signature.addFreshPredicate(1, predName)
8:     constants.insert(constantFunctor, predicateFunctor)
9:     return predicateFunctor
10: end function
```

Algorithm 7 Check Constant Term

Signature: isConstantTerm : TermList → bool

```
1: function isConstantTerm(tl)
2:     return ¬tl.isVar() ∧ tl.term().arity() = 0
3: end function
```

Algorithm 8 Replace Constants in Atomic Formula

Signature: replaceinAtomicFormula : Formula* × DHMap& → Formula*

```
1: function replaceinAtomicFormula(formula, constants)
2:     if formula = NULL then
3:         return NULL
4:     end if
5:     lit ← formula.literal()
6:     if lit = NULL then
7:         return formula
8:     end if
9:     arity ← lit.arity()
10:    if arity = 0 ∨ arity > 2 then
11:        return formula
12:    end if
13:    if arity = 1 then
14:        d ← lit.nthArgument(0)
15:        if ¬isConstantTerm(d) then
16:            return formula
17:        end if
18:        p_d ← getPredForConstant(d, constants)
19:        L ← replaceArgument(lit, 0, x)
20:        return ∃x (p_d(x) ∧ L)
21:    end if
22:    a_0 ← lit.nthArgument(0)
23:    a_1 ← lit.nthArgument(1)
24:    c_0 ← isConstantTerm(a_0)
25:    c_1 ← isConstantTerm(a_1)
26:    if ¬c_0 ∧ ¬c_1 then
27:        return formula
28:    end if
29:    if c_0 ∧ c_1 then
30:        p_0 ← getPredForConstant(a_0, constants)
31:        p_1 ← getPredForConstant(a_1, constants)
32:        if lit.isEquality() then
33:            L ← createEquality(lit.polarity(), x, y)
34:        else
35:            L ← replaceArguments(lit, x, y)
36:        end if
37:        return ∃x∃y (p_0(x) ∧ p_1(y) ∧ L)
38:    end if
39:    if c_0 then
40:        i ← 0
41:        a ← a_0; b ← a_1
42:    else
43:        i ← 1
44:        a ← a_1; b ← a_0
45:    end if
46:    p ← getPredForConstant(a, constants)
47:    v ← chooseVariableDifferentFrom(b)
48:    if lit.isEquality() then
49:        if i = 0 then
50:            L ← createEquality(lit.polarity(), v, a_1)
51:        else
52:            L ← createEquality(lit.polarity(), a_0, v)
53:        end if
54:    else
55:        L ← replaceArgument(lit, i, v)
56:    end if
57:    return ∃v (p(v) ∧ L)
58: end function
```

Algorithm 9 Generate Uniqueness Axiom

Signature: makeUniquenessAxiom : unsigned → Formula∗

```
1:  function makeUniquenessAxiom(predicateFunctor)
2:      x ← TermList::var(0)
3:      y ← TermList::var(1)
4:      predY ← createUnaryAtom(predicateFunctor, y)
5:      eq ← createEquality(true, x, y)
6:      implication ← BinaryFormula(IMP, predY, eq)
7:      universalY ← ∀y (predY → eq)
8:      predX ← createUnaryAtom(predicateFunctor, x)
9:      conjunction ← predX ∧ universalY
10:     finalAxiom ← ∃x conjunction
11:     return finalAxiom
12: end function
```

Algorithm 10 Recursive Constant Replacement in Formula

Signature: replaceInFormula : Formula$*$ $\times$DHMap& $\rightarrow$ Formula$*$

```
1: function replaceInFormula(formula, constants)
2:     if formula = NULL then
3:         return NULL
4:     end if
       formula.connective()
       LITERAL
5:     return replaceinAtomicFormula(formula, constants)
       IMP, IFF, XOR
6:     left ← replaceInFormula(formula.left(), constants)
7:     right ← replaceInFormula(formula.right(), constants)
8:     if left ≠ formula.left() ∨ right ≠ formula.right() then
9:         return BinaryFormula(formula.connective(), left, right)
10:    end if
11:    return formula
       AND, OR
12:    newArgs ← FormulaList()
13:    changed ← false
14:    for all arg ∈ formula.args() do
15:        newArg ← replaceInFormula(arg, constants)
16:        if newArg ≠ arg then
17:            changed ← true
18:        end if
19:        newArgs.push(newArg)
20:    end for
21:    if changed then
22:        return JunctionFormula(formula.connective(), newArgs)
23:    end if
24:    return formula
       NOT
25:    newArg ← replaceInFormula(formula.uarg(), constants)
26:    if newArg ≠ formula.uarg() then
27:        return ¬newArg
28:    end if
29:    return formula
       FORALL, EXISTS
30:    newSub ← replaceInFormula(formula.qarg(), constants)
31:    if newSub ≠ formula.qarg() then
32:        return QuantifiedFormula(formula.connective(), formula.vars(), newSub)
33:    end if
34:    return formula
35:    return formula
36: end function
```

Algorithm 11 Replace Constants in Clause

Signature: replaceInClause : Clause $*$ $\times$DHMap& $\rightarrow$ Unit$*$

```
 1: function replaceInClause(cl, constants)
 2:     modified ← false
 3:     processedLiterals ← Stack⟨Formula*⟩()
 4:     for i ← 0 to cl.length() − 1 do
 5:         lit ← cl[i]
 6:         litFormula ← AtomicFormula(lit)
 7:         processedFormula ← replaceInFormula(litFormula, constants)
 8:         if processedFormula ≠ litFormula then
 9:             modified ← true
10:         end if
11:         processedLiterals.push(processedFormula)
12:     end for
13:     if ¬modified then
14:         return cl
15:     end if
16:     disjuncts ← FormulaList()
17:     while ¬processedLiterals.isEmpty() do
18:         disjuncts.push(processedLiterals.pop())
19:     end while
20:     fullDisjunction ← JunctionFormula(OR, disjuncts)
21:     return FormulaUnit(fullDisjunction)
22: end function
```

Algorithm 12 Lemma 1 – Elimination of Individual Constants

Signature: applyLemma1 : Problem& → Void

```
1:  function applyLemma1(prb)
2:      constants ← DHMap⟨unsigned, unsigned⟩()
3:      it ← UnitList::DelIterator(prb.units())
4:      while it.hasNext() do
5:          unit ← it.next()
6:          if ¬unit → isClause() then
7:              fu ← static_cast⟨FormulaUnit*⟩(unit)
8:              originFormula ← fu.formula()
9:              processedFormula ← replaceInFormula(originFormula, constants)
10:             if processedFormula ≠ originFormula then
11:                 newUnit ← FormulaUnit(processedFormula)
12:                 it.replace(newUnit)
13:             end if
14:         else
15:             cl ← static_cast⟨Clause*⟩(unit)
16:             processedUnit ← replaceInClause(cl, constants)
17:             if processedUnit ≠ cl then
18:                 it.replace(processedUnit)
19:             end if
20:         end if
21:     end while
22:     if ¬constants.isEmpty() then
23:         newUnits ← UnitList()
24:         for all (constantId, predicateId) ∈ constants do
25:             uniquenessFormula ← makeUniquenessAxiom(predicateId)
26:             uniquenessUnit ← FormulaUnit(uniquenessFormula, AXIOM)
27:             newUnits.push(uniquenessUnit)
28:         end for
29:         prb.units() ← UnitList::concat(newUnits, prb.units())
30:     end if
31: end function
```

4.2.2 Lemma 2: Normalizzazione di Scott e Rinominazione Sottoformule Complesse

Il Lemma 2 lavora sulle sole unità formula; le clausole già presenti non vengono elaborate. Per ogni formula, chiama prima `Shell::Rectify::rectify` e poi `Shell::NNF::nnf`. Il `DefinitionBuilder` nomina in modo ricorsivo le sottoformule e accumula le definizioni. La trasformazione completa avviene solo quando il predicato radice può essere asserito come formula chiusa; se la radice non è nullaria o una sottoformula supera il limite di due variabili libere, l'unità originale viene lasciata nel problema e le definizioni provvisorie non vengono aggiunte.

L'implementazione si articola su tre algoritmi principali:

1. **Gestione del problema**: scorre le unità formula, rettifica e converte in NNF, quindi aggiunge al problema le definizioni generate.

2. **Costruzione ricorsiva delle definizioni**: `DefinitionBuilder::nameFormula` restituisce direttamente i letterali atomici. Per le congiunzioni e disgiunzioni crea un predicato sulle variabili libere e aggiunge clausole universalmente quantificate che esprimono entrambe le direzioni della definizione. Per i quantificatori costruisce le clausole che collegano il predicato definitorio al corpo; il verso che richiede un testimone o un controesempio usa una formula con prefisso $\forall x \exists y$ quando la sottoformula ha una variabile libera, e con il solo esistenziale quando non ne ha.

   - calcola le variabili libere, escludendo quelle legate dai quantificatori;
   - crea un predicato fresco di arità pari al numero di variabili libere, ammesso che tale numero sia al massimo due;
   - aggiunge le clausole definitorie e restituisce il letterale positivo che rappresenta la sottoformula.

3. **Asserzione radice**: quando il letterale radice è nullario, viene asserito sotto un quantificatore esistenziale su una variabile non usata nella matrice. Il codice inserisce le formule definitorie come assiomi e rimpiazza l'unità originaria con l'asserzione della radice.

Algorithm 13 Lemma 2 – Conversione nella forma normale di Scott

Signature: applyLemma2 : Problem& → Void

```
1: function applyLemma2(prb)
2:     definitions ← UnitList()
3:     it ← UnitList::DelIterator(prb.units())
4:     while it.hasNext() do
5:         unit ← it.next()
6:         if unit → isClause() then
7:             continue
8:         end if
9:         original ← FormulaUnit(unit)
10:        normalized ← NNF(Rectify(original))
11:        builder ← DefinitionBuilder()
12:        matrix ← normalized.formula()
13:        builder.nextVariable ← nextUnusedVariable(matrix)
14:        root ← builder.nameFormula(matrix)
15:        if root = NULL then
16:            continue
17:        end if
18:        assertion ← builder.rootAssertion(root)
19:        if assertion = NULL then
20:            continue
21:        end if
22:        it.replace(FormulaUnit(assertion))
23:        while ¬builder.definitions.empty() do
24:            definition ← builder.definitions.popBack()
25:            definitions.push(FormulaUnit(definition, AXIOM))
26:        end while
27:    end while
28:    prb.units() ← UnitList::concat(definitions, prb.units())
29: end function
```

Algorithm 14 Naming ricorsivo delle sottoformule di Scott

Signature: nameFormula : Formula$*$ $\rightarrow$ Literal$*$

```
 1: function nameFormula(formula)
 2:     if formula = NULL then
 3:         return NULL
 4:     end if
 5:     if formula.connective() = LITERAL then
 6:         return formula.literal()
 7:     end if
 8:     if formula.connective() ∈ {TRUE, FALSE} then
 9:         name ← newNullaryPredicate()
10:         lit ← name if TRUE, otherwise ¬name
11:         addUniversalClause({lit}, ∅)
12:         return name
13:     end if
14:     formula ← splitBundledQuantifiers(formula)
15:     vars ← variablesOf(formula)
16:     if |vars| > 2 then
17:         return NULL
18:     end if
19:     name ← newPredicateLiteral(vars)
20:     if formula.connective() ∈ {AND, OR} then
21:         children ← [nameFormula(a) : a ∈ formula.args()]
22:         if NULL ∈ children then
23:             return NULL
24:         end if
25:         if formula.connective() = AND then
26:             D ← {{¬name, c} : c ∈ children} ∪ {{name} ∪ {¬c : c ∈ children}}
27:         else
28:             D ← {{¬name} ∪ children} ∪ {{name, ¬c} : c ∈ children}
29:         end if
30:         for all cl ∈ D do
31:             addUniversalClause(cl, vars)
32:         end for
33:         return name
34:     end if
35:     if formula.connective() ∈ {EXISTS, FORALL} then
36:         y ← firstBoundVariable(formula)
37:         body ← nameFormula(formula.qarg())
38:         if body = NULL then
39:             return NULL
40:         end if
41:         allVars ← withVariable(vars, y)
42:         if formula.connective() = EXISTS then
43:             w ← universallyThenExistentially(¬name ∨ body, vars, y)
44:             addUniversalClause({name, ¬body}, allVars)
45:         else
46:             addUniversalClause({¬name, body}, allVars)
47:             w ← universallyThenExistentially(name ∨ ¬body, vars, y)
48:         end if
49:         if w ≠ NULL then
50:             definitions.push(w)
51:         end if
52:         return name
53:     end if
54:     return NULL
55: end function
```

Algorithm 15 Costruzione della forma universale di Tipo 3

Signature: universallyQuantified : Formula $*$ ×Vector¡unsigned¿ → Formula$*$

```
1: function universallyQuantified(matrix, vars)
2:     while |vars| < 2 do
3:         vars.push(freshVariable())
4:     end while
5:     if |vars| > 2 then
6:         return NULL
7:     end if
8:     for all v ∈ reverse(vars) do
9:         matrix ← ∀v matrix
10:    end for
11:    return matrix
12: end function
```

Algorithm 16 Costruzione della forma universale-esistenziale di Tipo 2

Signature: universallyThenExistentially : Formula $*$ ×Vector¡unsigned¿ × unsigned → Formula$*$

```
1: function universallyThenExistentially(matrix, freeVars, y)
2:     if |freeVars| > 1 then
3:         return NULL
4:     end if
5:     matrix ← ∃y matrix
6:     if |freeVars| = 1 then
7:         x ← freeVars[0]
8:         matrix ← ∀x matrix
9:     end if
10:    return matrix
11: end function
```

Algorithm 17 Asserzione della formula radice in forma di Tipo 1

Signature: rootAssertion : Literal$*$ → Formula$*$

```
1: function rootAssertion(root)
2:     if root = NULL then
3:         return FalseFormula()
4:     end if
5:     if root.arity() = 0 then
6:         x ← freshVariable()
7:         return ∃x root
8:     end if
9:     return NULL
10: end function
```

4.2.3 Lemma 3: Gestione dell'Uguaglianza

Il Lemma 3 visita le unità formula e scende ricorsivamente nei congiunti esterni. Le unità clausola vengono saltate. Il riconoscimento è strutturale: per Tipo 2 richiede un quantificatore universale seguito da uno esistenziale; per Tipo 3 raccoglie due variabili legate da quantificatori universali consecutivi e richiede che il corpo finale non abbia altri quantificatori. Per Tipo 1 tratta una formula esistenziale e semplifica le uguaglianze riflessive nel suo corpo.

Nel Tipo 2, la semplificazione valuta le uguaglianze tra le due variabili del prefisso e quelle riflessive. I letterali di uguaglianza che coinvolgono altri termini o variabili non riconosciuti rimangono nella matrice. Il codice poi antepone il letterale nativo $x \neq y$ alla matrice con una congiunzione. Nel Tipo 3, semplifica la matrice sotto l'ipotesi che le due variabili siano distinte e aggiunge il letterale nativo $x = y$ in disgiunzione: se il corpo è una congiunzione, il letterale viene aggiunto a ogni suo congiunto. Il Lemma 3 non converte le formule in CNF; presuppone la forma prodotta dal Lemma 2 e conserva l'uguaglianza positiva per i passaggi seguenti.

- simplifyEqualityUnderDistinct: valuta le uguaglianze tra variabili reflexive e quelle tra le due variabili del prefisso di Tipo 2; le altre uguaglianze non riconosciute restano invariate.

- transformScottFormula: riconosce i prefissi di Tipo 1, 2 e 3 e applica rispettivamente la semplificazione reflexive, il guardiano negativo o il guardiano positivo.

- addEqualityToCnf: antepone il guardiano a ciascuna clausola quando la matrice è una congiunzione.

- applyLemma3: visita le unità formula e trasforma ciascun congiunto Scott.

Algorithm 18 Lemma 3 – Semplificazione dell'uguaglianza nei casi diagonale e distinto

Signature: simplifyEqualityUnderRelation : Formula $*$ $\times$unsigned^2 $\times$ bool $\rightarrow$ Formula$*$

function simplifyEqualityUnderRelation($formula, x, y, pairEqual$)
  if $formula$ = NULL **then**
    return NULL
  end if
  if $formula$ e' un letterale di uguaglianza su variabili a, b **then**
    if $a = b$ **then**
      $eqTrue \leftarrow$ **true**
    else if $(a, b) = (x, y) \vee (a, b) = (y, x)$ **then**
      $eqTrue \leftarrow pairEqual$
    else
      return $formula$
    end if
    $literalTrue \leftarrow eqTrue$ se il letterale e' positivo,
    altrimenti $literalTrue \leftarrow \neg eqTrue$
    return $\top$ se $literalTrue$, altrimenti $\bot$
  else if $formula$.connective() = NOT **then**
    $sub \leftarrow formula$.uarg()
    $u \leftarrow$ simplifyEqualityUnderRelation($sub, x, y, pairEqual$)
    return negazione di u con semplificazione di $\top$ e $\bot$
  else if $formula$.connective() $\in \{$AND, OR$\}$ **then**
    Semplifica ricorsivamente tutti i congiunti o disgiunti
    Elimina gli elementi neutri e valuta quelli assorbenti
    return la formula semplificata, o $formula$ se invariata
  end if
  return $formula$
end function

Algorithm 19 Lemma 3 – Introduzione della guardia di uguaglianza

Signature: addEqualityToCnf : unsigned^2 $\times$ Formula$*$ $\rightarrow$ Formula$*$

function addEqualityToCnf(x, y, cnf)
  $eq \leftarrow$ makeEquality(**true**, x, y)
  if cnf.connective() $\neq$ AND **then**
    return $eq \vee cnf$
  end if
  $clauses \leftarrow [\,]$
  for all $clause \in cnf$.args() **do**
    $clauses$.push($eq \vee clause$)
  end for
  return $\bigwedge clauses$
end function

Algorithm 20 Lemma 3 – Trasformazione dei tre tipi di Scott

Signature: transformScottFormula : Formula$*$ $\to$ Formula$*$

function transformScottFormula($formula$)
  if $formula$ = NULL **then**
    return $formula$
  end if
  if $formula$.connective() = EXISTS **then** ▷ Tipo 1: $\exists x\, M$
    $x \leftarrow$ firstBoundVariable($formula$); $M \leftarrow formula$.qarg()
    $D \leftarrow$ simplifyEqualityUnderRelation($M, x, x,$ **true**)
    if $D \neq M \wedge \neg$containsEquality(D) **then**
      return $\exists x\, D$
    end if
    return $formula$
  end if
  if $formula$ ha prefisso $\forall x \exists y$ **then** ▷ Tipo 2
    $M \leftarrow$ matrice di $formula$; $(x, y) \leftarrow$ variabili del prefisso
    $D \leftarrow$ simplifyEqualityUnderRelation($M[y \mapsto x], x, y,$ **true**)
    $O \leftarrow$ simplifyEqualityUnderRelation($M, x, y,$ **false**)
    if D o O non sono CNF prive di uguaglianza **then**
      return $formula$
    end if
    return $\forall x \left(D \vee \exists y \left(x \neq y \wedge O\right)\right)$
  end if
  if $formula$.connective() = FORALL **then** ▷ Tipo 3
    $prefix \leftarrow formula$; $vars \leftarrow [\,]$
    while $prefix \neq$ NULL $\wedge\, prefix$.connective() = FORALL **do**
      $vars$.appendAll($prefix$.vars())
      if $|vars| > 2$ **then**
        return $formula$
      end if
      $prefix \leftarrow prefix$.qarg()
    end while
    if $|vars| \neq 2 \vee prefix =$ NULL **then**
      return $formula$
    end if
    if $prefix$ e' ancora una formula quantificata **then**
      return $formula$
    end if
    $(x, y) \leftarrow (vars[0], vars[1])$; $M \leftarrow prefix$
    $D \leftarrow$ simplifyEqualityUnderRelation($M[y \mapsto x], x, y,$ **true**)
    $O \leftarrow$ simplifyEqualityUnderRelation($M, x, y,$ **false**)
    if D o O non sono CNF prive di uguaglianza **then**
      return $formula$
    end if
    $G \leftarrow$ addEqualityToCnf(x, y, O)
    return $\left(\forall x\, D\right) \wedge \left(\forall x \forall y\, G\right)$
  end if
  return $formula$
end function

Algorithm 21 Lemma 3 – Trasformazione ricorsiva dei congiunti di Scott

Signature: transformScottConjuncts : Formula$*$ $\rightarrow$ Formula$*$

```
1: function transformScottConjuncts(formula)
2:     if formula = NULL ∨ formula.connective() ≠ AND then
3:         return transformScottFormula(formula)
4:     end if
5:     newArgs ← [ ]; changed ← false
6:     for all arg ∈ formula.args() do
7:         newArg ← transformScottConjuncts(arg)
8:         changed ← changed ∨ (newArg ≠ arg)
9:         newArgs.push(newArg)
10:    end for
11:    if ¬changed then
12:        return formula
13:    end if
14:    return ⋀ newArgs
15: end function
```

Algorithm 22 Lemma 3 – Applicazione al problema

```
    Signature: applyLemma3 : Problem& → Void
 1: function applyLemma3(prb)
 2:     generated ← UnitList()
 3:     it ← UnitList::DelIterator(prb.units())
 4:     while it.hasNext() do
 5:         unit ← it.next()
 6:         if unit → isClause() then
 7:             continue
 8:         end if
 9:         fu ← static_cast⟨FormulaUnit∗⟩(unit)
10:         transformed ← transformScottConjuncts(fu.formula())
11:         if transformed = fu.formula() then
12:             continue
13:         end if
14:         if transformed.connective() = AND then
15:             first ← true
16:             for all conjunct ∈ transformed.args() do
17:                 newUnit ← FormulaUnit(conjunct, unit)
18:                 if first then
19:                     it.replace(newUnit); first ← false
20:                 else
21:                     generated.push(newUnit)
22:                 end if
23:             end for
24:         else
25:             it.replace(FormulaUnit(transformed, unit))
26:         end if
27:     end while
28:     prb.units() ← UnitList::concat(generated, prb.units())
29: end function
```

4.2.4 Lemma 4: Saturazione Ristretta per Clausole di Tipo 3

Il Lemma 4 applica la risoluzione ristretta della Definizione 2 dell'articolo. Prima rettifica le unità formula e rinomina le variabili di ciascuna clausola con indici consecutivi, così che i due argomenti del pivot corrispondano agli indici canonici 0 e 1. Le clausole vengono divise in Tipo 3, se contengono un letterale non di uguaglianza che contiene entrambe le variabili, e clausole ordinarie.

Per le unità formula, il codice riconosce il Tipo 3 contando due variabili quantificate universalmente e clausificando il corpo se è composto da letterali, disgiunzioni e congiunzioni. Il Tipo 2 è riconosciuto solo con un universale e un esistenziale, ciascuno con una sola variabile, e quando la matrice contiene come congiunto il letterale negativo $x \neq y$. Quel guardiano viene separato; gli altri congiunti sono convertiti in clausole e associati all'unità formula da cui provengono. Le unità che non corrispondono a queste forme restano invariate.

- normalizeClauseVariables: rinomina le variabili di ciascuna clausola con indici consecutivi. Clausole con più di due variabili non vengono rinominate.

- containsBothVariables e areComplementaryUnderRename: individuano i pivot non di uguaglianza che contengono entrambe le variabili; i letterali sono complementari solo se hanno stesso predicato e arità, polarità opposta e argomenti identici oppure scambiati.

- La risoluzione ristretta non unifica né istanzia termini: ammette solo l'identità delle variabili o il loro scambio. Elimina i due pivot, applica lo scambio ai residui quando necessario, rimuove i duplicati e scarta i risolventi tautologici. La saturazione delle clausole di Tipo 3 prosegue finché non si producono nuovi risolventi.

- extractType3 e extractType2: raccolgono le clausole dalle matrici nelle forme riconosciute sopra. Il guardiano del Tipo 2 non entra tra le clausole da risolvere.

- applyLemma4: dopo la saturazione di Tipo 3, risolve ciascuna clausola risultante con le clausole di Tipo 2. Semplifica i risolventi di Tipo 2 valutando le uguaglianze in base al guardiano di distinzione e li aggiunge alla matrice dell'unità proprietaria. Nel risultato conserva le clausole di Tipo 3 che non hanno letterali non di uguaglianza contenenti entrambe le variabili; elimina quelle che ne hanno almeno uno.

Algorithm 23 Lemma 4 – Estrazione delle clausole di Tipo 3

```
    Signature: extractType3 : FormulaUnit * ×vector⟨Clause*⟩& → bool
 1: function extractType3(unit, clauses)
 2:     prefix ← unit.formula()
 3:     boundCount ← 0
 4:     while prefix ≠ NULL ∧ prefix.connective() = FORALL do
 5:         boundCount ← boundCount + countBoundVariables(prefix)
 6:         prefix ← prefix.qarg()
 7:     end while
 8:     if boundCount ≠ 2 ∨ prefix = NULL ∨ isQuantified(prefix) then
 9:         return false
10:     end if
11:     extracted ← [ ]
12:     if ¬appendCnfClauses(prefix, unit, extracted) then
13:         return false
14:     end if
15:     for all cl ∈ extracted do
16:         hasGuard ← false
17:         for all lit ∈ cl do
18:             if ¬lit.isEquality() then
19:                 continue
20:             end if
21:             if ¬lit.isPositive() ∨ ¬isGuardXY(lit) then
22:                 return false
23:             end if
24:             hasGuard ← true
25:         end for
26:         if ¬hasGuard then
27:             return false
28:         end if
29:     end for
30:     clauses.insertAll(extracted)
31:     return true
32: end function
```

Algorithm 24 Lemma 4 – Estrazione delle clausole di Tipo 2

Signature: EXTRACTTYPE2 : FormulaUnit$*\times$vector$\langle$Clause$*\rangle\&\times$unsigned$\&\times$unsigned$\&\rightarrow$ bool

```
1:  function EXTRACTTYPE2(unit, clauses, x, y)
2:      forall ← unit.formula()
3:      if forall = NULL ∨ forall.connective() ≠ FORALL then
4:          return false
5:      end if
6:      if ¬GETSINGLEBOUNDVARIABLE(forall, x) then
7:          return false
8:      end if
9:      exists ← forall.qarg()
10:     if exists = NULL ∨ exists.connective() ≠ EXISTS then
11:         return false
12:     end if
13:     if ¬GETSINGLEBOUNDVARIABLE(exists, y) then
14:         return false
15:     end if
16:     conjuncts ← TOPLEVELCONJUNCTS(exists.qarg())
17:     extracted ← [ ]; foundGuard ← false
18:     for all conj ∈ conjuncts do
19:         lit ← FORMULALITERAL(conj)
20:         if ISDISTINCTGUARD(lit, x, y) then
21:             foundGuard ← true
22:             continue
23:         end if
24:         if ¬APPENDCNFCLAUSES(conj, unit, extracted) then
25:             return false
26:         end if
27:     end for
28:     if ¬foundGuard then
29:         return false
30:     end if
31:     for all cl ∈ extracted do
32:         if CONTAINSEQUALITY(cl) then
33:             return false
34:         end if
35:     end for
36:     clauses.insertAll(extracted)
37:     return true
38: end function
```

Algorithm 25 Lemma 4 – Risoluzione ristretta a due variabili

Signature: resolveRestricted : Clause $*$ $\times$Clause $*$ $\times$vector$\langle$Clause$*\rangle$& $\rightarrow$ Void

```
1: function resolveRestricted(left, right, resolvents)
2:     for all pivotLeft ∈ left do
3:         if pivotLeft.isEquality() ∨ ¬containsBothVariables(pivotLeft) then
4:             continue
5:         end if
6:         for all pivotRight ∈ right do
7:             if pivotRight.isEquality()∨¬containsBothVariables(pivotRight) then
8:                 continue
9:             end if
10:            swap ← false
11:            if ¬areComplementaryUnderRename(pivotLeft, pivotRight, swap) then
12:                continue
13:            end if
14:            remainder ← left \ {pivotLeft}
15:            for all lit ∈ right \ {pivotRight} do
16:                remainder.push(swap ? swapVariables(lit) : lit)
17:            end for
18:            seen ← ∅; normalized ← [ ]
19:            tautology ← false
20:            for all lit ∈ remainder do
21:                if complementaryLiteral(lit) ∈ seen then
22:                    tautology ← true
23:                    break
24:                end if
25:                if lit ∉ seen then
26:                    seen.insert(lit); normalized.push(lit)
27:                end if
28:            end for
29:            if ¬tautology then
30:                resolvents.push(makeClause(normalized, left, right))
31:            end if
32:        end for
33:    end for
34: end function
```

Algorithm 26 Lemma 4 – Semplificazione di una clausola di Tipo 2 per $x \neq y$

Signature: simplifyType2ClauseUnderDistinct : Clause $*$ $\times$unsigned $\times$ unsigned $\rightarrow$ Clause$*$

```
1: function simplifyType2ClauseUnderDistinct(clause, x, y)
2:     remaining ← [ ]
3:     for all lit ∈ clause do
4:         if ¬lit.isEquality() then
5:             remaining.push(lit)
6:             continue
7:         end if
8:         (a, b) ← arguments(lit)
9:         if ¬areVariables(a, b) then
10:            remaining.push(lit)
11:            continue
12:        end if
13:        reflexive ← (a = b)
14:        guardedPair ← ((a = x ∧ b = y) ∨ (a = y ∧ b = x))
15:        if ¬reflexive ∧ ¬guardedPair then
16:            remaining.push(lit)
17:            continue
18:        end if
19:        equalityTrue ← reflexive                    ▷ Ipotesi: x ≠ y
20:        literalTrue ← lit.isPositive() ? equalityTrue : ¬equalityTrue
21:        if literalTrue then
22:            return NULL                              ▷ Clausola già vera
23:        end if
                                  ▷ Un letterale di uguaglianza falso è omesso
24:    end for
25:    return makeClauseWithInference(remaining, clause.inference())
26: end function
```

Algorithm 27 Lemma 4 – Saturazione ristretta ed eliminazione dei letterali binari

```
    Signature: applyLemma4 : Problem& → Void
 1: function applyLemma4(prb)
 2:     Rectify(prb.units())
 3:     for all unit ∈ prb.units() with unit.isClause() do
 4:         unit ← normalizeClauseVariables(unit)
 5:     end for
 6:     type3, type2, ordinary, retained, type2Units ← [ ]
 7:     additions, knownType2 ← empty maps
    Classificazione ed estrazione delle clausole
 8:     for all unit ∈ prb.units() do
 9:         if unit.isClause() then
10:             append unit to type3 if it has a non-equality two-variable literal, else to ordinary
11:         else if extractType3(unit, extracted3) then
12:             type3.insertAll(extracted3)
13:         else if extractType2(unit, extracted2, x, y) then
14:             type2 ← type2 ++ [(cl, unit, x, y) : cl ∈ extracted2]
15:             knownType2[unit] ← {clauseKey(cl) : cl ∈ extracted2}
16:             type2Units.push(unit)
17:         else
18:             retained.push(unit)
19:         end if
20:     end for
    Saturazione delle clausole di Tipo 3
21:     passive3 ← uniqueQueue(type3); active3 ← [ ]
22:     knownType3 ← {clauseKey(cl) : cl ∈ passive3}
23:     while ¬passive3.empty() do
24:         given ← passive3.popFront()
25:         for all cl ∈ active3 do
26:             resolvents ← [ ]
27:             resolveRestricted(given, cl, resolvents)
28:             for all r ∈ resolvents with clauseKey(r) ∉ knownType3 do
29:                 knownType3.insert(clauseKey(r)); passive3.pushBack(r)
30:             end for
31:         end for
32:         active3.push(given)
33:     end while
    Risoluzione mista Tipo 2 / Tipo 3
34:     pending2 ← indexQueue(type2)
35:     while ¬pending2.empty() do
36:         (cl, owner, x, y) ← type2[pending2.popFront()]
37:         for all u ∈ active3 do
38:             resolvents ← [ ]
39:             resolveRestricted(u, cl, resolvents)
40:             for all r ∈ resolvents do
41:                 r′ ← simplifyType2ClauseUnderDistinct(r, x, y)
42:                 if r′ ≠ NULL ∧ clauseKey(r′) ∉ knownType2[owner] then
43:                     knownType2[owner].insert(clauseKey(r′))
44:                     type2.push((r′, owner, x, y)); additions[owner].push(r′)
45:                     pending2.pushBack(type2.size() − 1)
46:                 end if
47:             end for
48:         end for
49:     end while
    Ricostruzione: conserva soltanto le clausole di Tipo 3 prive di letterali binari non identitari
50:     result ← retained ++ ordinary
51:     result ← result ++ [addType2Resolvents(u, additions[u]) : u ∈ type2Units]
52:     result ← result ++ [cl ∈ active3 : ¬isType3RemovalCandidate(cl)]
53:     prb.units() ← result
54: end function
```

4.2.5 Lemma 5: Decomposizione in Componenti Unarie

Il Lemma 5 inizia rettificando le unità e sostituendo le uguaglianze negative con un unico predicato binario fresco `neq`, se ne trova almeno una. Per le clausole sostituisce ogni letterale di uguaglianza negativo con `neq` positivo e aggiunge la clausola universale $\neg neq(x, x)$. Nelle formule gestisce sia i letterali di uguaglianza a polarità negativa sia la forma esplicita di negazione di un'uguaglianza positiva. Questo assioma impone l'irriflessività di `neq`; il codice non aggiunge in questa fase assiomi di simmetria o una definizione completa di `neq` come disuguaglianza.

Successivamente esamina le unità una per una. Per una clausola cerca prima un'uguaglianza positiva tra due variabili distinte; le eventuali altre uguaglianze positive sono accettate solo se riguardano la stessa coppia. I letterali rimanenti devono contenere al più una delle due variabili: quelli riferiti alla prima formano γ, quelli riferiti alla seconda formano δ. Un letterale senza variabili può essere assegnato a γ. Se un letterale non rispetta queste condizioni, la clausola rimane invariata. Per le formule, l'estrazione è più ristretta: riconosce disgiunzioni di letterali (eventualmente sotto quantificatori universali) e non formule arbitrarie in CNF.

Quando la decomposizione riesce, le variabili della coppia vengono rinominate negli indici 0 e 1. Il codice costruisce la disgiunzione dei tre rami della trasformazione, usando le formule γ e δ e il ramo di esistenza dell'unico elemento che non soddisfa γ. La nuova formula viene inserita come unità assiomatica al posto dell'unità originale.

Le routine usate in questa trasformazione sono:

- isUnaryOnVar e renameVariable: escludono i letterali di uguaglianza dalla classificazione, verificano che non compaiano variabili diverse da quella assegnata alla componente e rinominano tale variabile nell'indice canonico 0. I letterali nullari soddisfano il controllo per entrambe le componenti e vengono assegnati alla prima.

- extractLiteralsFromFormula estrae letterali atomici, negazioni di atomi e disgiunzioni; attraversa anche quantificatori universali. createDisjunctionFromLiterals ricostruisce la formula, restituendo falso se la lista è vuota.

- decomposeLiterals individua la coppia x, y dall'uguaglianza positiva prima di classificare gli altri letterali, così non dipende dall'ordine con cui compaiono nella clausola. Se non trova la coppia o non riesce ad assegnare un letterale a una componente, non trasforma l'unità.

- applyLemma5: per i casi riconosciuti sostituisce la formula con i tre rami della trasformazione:

$$\forall x.\gamma(x) \vee \forall x.\delta(x) \vee (\forall x(\gamma(x) \iff \delta(x)) \wedge \exists x(\neg\gamma(x) \wedge \forall y(\neg\gamma(y) \implies x = y)))$$

Algorithm 28 Lemma 5 – Trasformazione delle clausole con uguaglianza positiva

Signature: applyLemma5 : Problem& → Void

```
1:  function applyLemma5(prb)
2:      Rectify(prb.units())
3:      eliminateNegativeEqualities(prb)
4:      newUnits ← UnitList()
5:      it ← UnitList::DelIterator(prb.units())
6:      while it.hasNext() do
7:          unit ← it.next()
8:          γLits ← [ ]; δLits ← [ ]
9:          canTransform ← false
10:         if unit → isClause() then
11:             canTransform ← decomposeType3Clause(unit, γLits, δLits)
12:         else
13:             lits ← [ ]
14:             if extractLiteralsFromFormula(unit.formula(), lits) then
15:                 canTransform ← decomposeLiterals(lits, γLits, δLits)
16:             end if
17:         end if
18:         if ¬canTransform then
19:             newUnits.push(unit)
20:             continue
21:         end if
22:         γ(x) ← createDisjunctionFromLiterals(γLits)
23:         δ(x) ← createDisjunctionFromLiterals(δLits)
24:         γYLits ← [ ]
25:         for all lit ∈ γLits do
26:             γYLits.push(renameVariable(lit, 0, 1))
27:         end for
28:         γ(y) ← createDisjunctionFromLiterals(γYLits)
29:         branch1 ← ∀x γ(x)
30:         branch2 ← ∀x δ(x)
31:         unique ← ∃x (¬γ(x) ∧ ∀y (¬γ(y) → x = y))
32:         coincidence ← ∀x (γ(x) ↔ δ(x))
33:         branch3 ← unique ∧ coincidence
34:         finalFormula ← branch1 ∨ branch2 ∨ branch3
35:         newUnits.push(FormulaUnit(finalFormula, AXIOM))
36:     end while
37:     prb.units() ← UnitList::reverse(newUnits)
38: end function
```

Algorithm 29 Lemma 5 – Decomposizione dei letterali in γ e δ

Signature: DECOMPOSELITERALS : LiteralList×LiteralList&×LiteralList& → bool

```
1: function DECOMPOSELITERALS(lits, γLits, δLits)
2:     hasPositiveEq ← false
3:     for all lit ∈ lits do
4:         if lit rappresenta u = v con u ≠ v variabili then
5:             if hasPositiveEq ∧ {u, v} ≠ {xVar, yVar} then
6:                 return false
7:             end if
8:             if ¬hasPositiveEq then
9:                 (xVar, yVar) ← (u, v)
10:            end if
11:            hasPositiveEq ← true
12:        end if
13:    end for
14:    if ¬hasPositiveEq then
15:        return false
16:    end if
17:    for all lit ∈ lits do
18:        if lit e' l'uguaglianza positiva tra xVar e yVar then
19:            continue
20:        else if ISUNARYONVAR(lit, xVar) then
21:            γLits.push(RENAMEVARIABLE(lit, xVar, 0))
22:        else if ISUNARYONVAR(lit, yVar) then
23:            δLits.push(RENAMEVARIABLE(lit, yVar, 0))
24:        else
25:            return false
26:        end if
27:    end for
28:    return true
29: end function
```

Algorithm 30 Lemma 5 – Eliminazione preliminare delle uguaglianze negative

```
    Signature: eliminateNegativeEqualities : Problem& → Void
 1: function eliminateNegativeEqualities(prb)
 2:     hasDisequality ← false
 3:     for all unit ∈ prb.units() do
 4:         if unit contiene un letterale di disuguaglianza then
 5:             hasDisequality ← true
 6:         end if
 7:     end for
 8:     if ¬hasDisequality then
 9:         return
10:     end if
11:     neqPredicate ← env.signature.addFreshPredicate(2,"neq")
12:     it ← UnitList::DelIterator(prb.units())
13:     while it.hasNext() do
14:         unit ← it.next()
15:         if unit → isClause() then
16:             changed ← false
17:             newLits ← Stack⟨Literal*⟩()
18:             for all lit ∈ unit do
19:                 if lit e' ¬(s = t) then
20:                     lit ← neqPredicate(s, t)
21:                     changed ← true
22:                 end if
23:                 newLits.push(lit)
24:             end for
25:             if changed then
26:                 it.replace(ClauseFromStack(newLits))
27:             end if
28:         else
29:             fu ← static_cast⟨FormulaUnit*⟩(unit)
30:             f' ← replaceNegativeEquality(fu.formula(), neqPredicate)
31:             if f' ≠ fu.formula() then
32:                 it.replace(FormulaUnit(f'))
33:             end if
34:         end if
35:     end while
36:     axiom ← Clause(¬neqPredicate(x, x))
37:     prb.units().push(axiom)
38: end function
```

4.2.6 Lemma 6: Eliminazione dei Quantificatori d'Unicità ed Assiomi di Congruenza

Il Lemma 6 visita le unità formula, non le clausole, e riconosce una forma sintattica precisa: un esistenziale il cui corpo è una congiunzione di $\neg\zeta(x)$ e di un universale

con implicazione da $\neg\zeta(y)$ all'uguaglianza $x = y$. Il controllo richiede gli indici canonici 0 e 1, la polarità positiva dell'uguaglianza e l'ordine indicato degli argomenti. Le formule equivalenti ma rappresentate con ordine o struttura diversi non sono riconosciute.

Per ogni occorrenza riconosciuta, il codice crea una costante fresca e_i e genera una congiunzione con $\zeta(e_i)$. Per ogni predicato non di uguaglianza di arità uno aggiunge $\forall x(\zeta(x) \rightarrow (p(x) \leftrightarrow p(e_i)))$. Per ciascun predicato binario aggiunge due formule universali, una per la prima posizione e una per la seconda. I predicati di arità diversa da uno o due non ricevono assiomi di congruenza. La sottoformula di unicità viene sostituita da questa congiunzione e l'unità formula modificata viene reintrodotta come assioma. Il codice non riapplica i Lemmi 1 e 2 e non verifica che tutte le uguaglianze siano state rimosse.

Nel codice sono presenti queste routine:

- La sostituzione ricorsiva rimpiazza una variabile con il termine costante fresco nelle formule riconosciute.

- LEMMA5ZETA: richiede un esistenziale con corpo AND formato esattamente da un elemento negato e da un universale la cui matrice è un'implicazione; il conseguente deve essere l'uguaglianza $x_0 = y_1$. Se la struttura o gli indici non coincidono, restituisce nessun risultato.

- REPLACEUNIQUENESSOCCURRENCES: visita connettivi e quantificatori delle formule, sostituisce le occorrenze riconosciute e conserva le unità che non cambiano.

- La generazione degli assiomi aggiunge per ogni costante fresca e_i l'asserzione $\zeta_i(e_i)$ e gli assiomi di congruenza per i predicati unari e binari della signature:

$$\forall x.(\zeta_i(x) \implies (p(x) \iff p(e_i)))$$

$$\forall x \forall y.(\zeta_i(x) \implies (q(x,y) \iff q(e_i,y))) \quad \wedge \quad \forall x \forall y.(\zeta_i(x) \implies (q(y,x) \iff q(y,e_i)))$$

- APPLYLEMMA6: sostituisce le formule modificate e conserva le altre unità. Non rilancia altri Lemmi né certifica autonomamente che tutte le uguaglianze siano state eliminate.

Algorithm 31 Lemma 6 – Eliminazione dei vincoli di unicità

```
    Signature: applyLemma6 : Problem& → Void
 1: function applyLemma6(prb)
 2:     newUnits ← UnitList()
 3:     it ← UnitList::DelIterator(prb.units())
 4:     while it.hasNext() do
 5:         unit ← it.next()
 6:         if ¬unit → isClause() then
 7:             fu ← static_cast⟨FormulaUnit*⟩(unit)
 8:             changed ← false
 9:             cleanFormula ← replaceUniquenessOccurrences(fu.formula(), changed)
10:             if changed then
11:                 cleanUnit ← FormulaUnit(cleanFormula, AXIOM)
12:                 newUnits.push(cleanUnit)
13:                 continue
14:             end if
15:         end if
16:         newUnits.push(unit)
17:     end while
18:     prb.units() ← UnitList::reverse(newUnits)
19: end function
```

Algorithm 32 Sostituzione ricorsiva dei vincoli di unicità

Signature: replaceUniquenessOccurrences : Formula $*$ $\times$bool& $\rightarrow$ Formula$*$

```
1: function replaceUniquenessOccurrences(formula, changed)
2:     if formula = NULL then
3:         return NULL
4:     end if
5:     ζ ← lemma5Zeta(formula)
6:     if ζ ≠ NULL then
7:         functor ← env.signature.addFreshFunction(0, "e_")
8:         setConstantType(functor)
9:         e ← createConstantTerm(functor)
10:        axioms ← FormulaList()
11:        generateCongruenceAxioms(ζ, e, axioms)
12:        changed ← true
13:        return conjunction(axioms)
14:    end if
15:    if formula.connective() = NOT then
16:        arg ← replaceUniquenessOccurrences(formula.uarg(), changed)
17:        return rebuildIfChanged(formula, NOT, arg)
18:    else if formula.connective() ∈ {AND, OR} then
19:        newArgs ← FormulaList(); localChange ← false
20:        for all arg ∈ formula.args() do
21:            newArg ← replaceUniquenessOccurrences(arg, changed)
22:            localChange ← localChange ∨ (newArg ≠ arg)
23:            newArgs.push(newArg)
24:        end for
25:        if localChange then
26:            return junction(formula.connective(), newArgs)
27:        end if
28:    else if formula.connective() ∈ {IMP, IFF, XOR} then
29:        left ← replaceUniquenessOccurrences(formula.left(), changed)
30:        right ← replaceUniquenessOccurrences(formula.right(), changed)
31:        return rebuildIfChanged(formula, formula.connective(), left, right)
32:    else if formula.connective() ∈ {FORALL, EXISTS} then
33:        arg ← replaceUniquenessOccurrences(formula.qarg(), changed)
34:        return rebuildIfChanged(formula, formula.connective(), formula.vars(), arg)
35:    end if
36:    return formula
37: end function
```

Algorithm 33 Generazione degli assiomi di congruenza Θ_i

Signature: generateCongruenceAxioms : Formula $*$ $\times$TermList $\times$ FormulaList& $\rightarrow$ Void

```
1: function generateCongruenceAxioms(ζ, e, axioms)
2:     ζ_e ← replaceVarWithTerm(ζ, 0, e)
3:     axioms.push(ζ_e)
4:     for all p ∈ env.signature.predicates() do
5:         if isEqualityPredicate(p) then
6:             continue
7:         end if
8:         if arity(p) = 1 then
9:             axioms.push(∀x (ζ(x) → (p(x) ↔ p(e))))
10:        else if arity(p) = 2 then
11:            axioms.push(∀x ∀y (ζ(x) → (p(x, y) ↔ p(e, y))))
12:            axioms.push(∀x ∀y (ζ(x) → (p(y, x) ↔ p(y, e))))
13:        end if
14:    end for
15: end function
```

4.3 Modulo di Risoluzione e Decision Procedure (`Resolution`)

Il modulo `Resolution` comprende l'indicizzazione delle clausole, le inferenze indicizzate e il solver. Le sue regole di risoluzione e factoring seguono i vincoli di selezione descritti nell'articolo. La selezione effettiva è definita dai test riportati sotto; il comparatore per profondità e indice è presente nel codice, ma non partecipa a quei test.

La procedura restituisce `SATISFIABLE`, `UNSATISFIABLE` oppure `UNKNOWN`. La garanzia teorica di terminazione dell'articolo richiede che gli invarianti e le condizioni di input siano rispettati; l'implementazione ha inoltre un limite di iterazioni e controlli che possono restituire `UNKNOWN`.

4.3.1 Indicizzazione $\mathcal{S}_2 + i$, Ordinamento $<_2$ e Selezione Σ_2 (`FO2Resolution`)

La classe `FO2Resolution` (file `FO2Resolution.hpp` e `FO2Resolution.cpp`) fornisce l'infrastruttura per la gestione delle clausole indicizzate (`IndexedClause`) e l'implementazione della funzione di selezione Σ_2.

1. **Indicizzazione delle Clausole (fromClause)**: Come mostrato nell'Algoritmo 34, `IndexedClause::fromClause` raccoglie le variabili della clausola e assegna a ogni letterale un indice $i \in \{0, 1\}$:

   - Se la clausola contiene due variabili distinte, ricevono indice 1 i letterali che contengono entrambe; tutti gli altri ricevono indice 0.

- Se la clausola contiene al più una variabile libera, tutti i letterali ricevono l'indice 0.

2. **Profondità**: le funzioni ausiliarie calcolano la profondità massima dei termini in un letterale. È presente anche un comparatore per profondità e indice, ma la selezione implementata non lo usa per ordinare i letterali.

3. **Funzione di Selezione Σ_2 (isSelected)**: `IndexedClause::isSelected` restituisce falso nei due casi seguenti e vero negli altri:

   - A non contiene termini funzionali, ha indice 0 e nella clausola compare almeno un letterale di indice 1.
   - A non contiene termini funzionali e nella clausola compare almeno un termine funzionale.

   Il controllo non consulta `compareGroundIndexedLiterals`; quindi la presenza di quella funzione non significa che l'implementazione ordini i letterali o verifichi la condizione di lifting della Definizione 5.

Algorithm 34 Costruzione di una clausola indicizzata

Signature: fromClause : Clause$*$ $\rightarrow$ IndexedClause

```
1:  function fromClause(cl)
2:      if cl = NULL then
3:          return IndexedClause()
4:      end if
5:      clauseVars ← collectVariables(cl)
6:      numClauseVars ← |clauseVars|
7:      indexedLits ← Vector⟨IndexedLiteral⟩()
8:      for i ← 0 to cl.length() − 1 do
9:          lit ← cl[i]
10:         litVars ← collectVariables(lit)
11:         idx ← 0
12:         if numClauseVars = 2 ∧ |litVars| = 2 then
13:             idx ← 1
14:         end if
15:         indexedLits.push(IndexedLiteral(lit, idx))
16:     end for
17:     return IndexedClause(indexedLits, cl)
18: end function
```

Algorithm 35 Verifica della selezionabilità secondo Σ_2

Signature: isValidSelectionCandidate : size_t → bool

```
1: function isValidSelectionCandidate(litIndex)
2:     if litIndex ≥ |_literals| then
3:         return false
4:     end if
5:     clauseHasFunctional ← false
6:     clauseHasIndexOne ← false
7:     for all ilit ∈ _literals do
8:         if hasFunctionalTerms(ilit.literal) then
9:             clauseHasFunctional ← true
10:        end if
11:        if ilit.index = 1 then
12:            clauseHasIndexOne ← true
13:        end if
14:    end for
15:    target ← _literals[litIndex]
16:    targetHasFunc ← hasFunctionalTerms(target.literal)
17:    if ¬targetHasFunc ∧ target.index = 0 ∧ clauseHasIndexOne then
18:        return false
19:    end if
20:    if ¬targetHasFunc ∧ clauseHasFunctional then
21:        return false
22:    end if
23:    return true
24: end function
```

Algorithm 36 Confronto tra letterali indicizzati ground

Signature: compareGroundIndexedLiterals : IndexedLiteral × IndexedLiteral → int

```
1: function compareGroundIndexedLiterals(ilitA, ilitB)
2:     depthA ← getLiteralDepth(ilitA.literal)
3:     depthB ← getLiteralDepth(ilitB.literal)
4:     if depthA < depthB then
5:         return −1
6:     else if depthA > depthB then
7:         return 1
8:     end if
9:     if ilitA.index < ilitB.index then
10:        return −1
11:    else if ilitA.index > ilitB.index then
12:        return 1
13:    end if
14:    return 0
15: end function
```

4.3.2 Regole di Inferenza e Semplificazione (FO2Inferences)

La classe FO2Inferences (file FO2Inferences.hpp e FO2Inferences.cpp) applica risoluzione, factoring e sussunzione alle clausole indicizzate. La sostituzione calcolata da RobSubstitution viene applicata ai letterali residui; ogni letterale residuo mantiene l'indice assegnato nella clausola d'origine.

- **Risoluzione indicizzata**: entrambi i pivot devono essere selezionati e avere polarità opposta. La routine rende positive le copie usate per l'unificazione, usa banche di variabili distinte per le due clausole e scarta la coppia se non è unificabile. Nel risultato rimuove duplicati identici per letterale e indice. Se due letterali residui sono complementari, il risolvente viene scartato come tautologico, anche quando i loro indici differiscono.

- **Fattorizzazione**: richiede che il primo letterale sia selezionato e che i due letterali abbiano la stessa polarità e siano unificabili nella stessa banca di variabili. Elimina il primo letterale, applica la sostituzione al secondo e ai residui e mantiene i rispettivi indici. La routine non richiede che i due indici coincidano.

- **Sussunzione Indicizzata (subsumes)**: La ricerca abbina ricorsivamente ogni letterale della prima clausola a un letterale della seconda con lo stesso indice, polarità e simbolo predicativo, usando il matching dei termini. Una clausola più lunga della seconda non può sussumerla; la clausola vuota sussume ogni clausola.

- **Splitting**: non è un'inferenza di FO2Inferences; il solver cerca una partizione della clausola in due parti con variabili disgiunte. Se separa letterali ground da non ground, crea tali due parti; altrimenti usa le componenti connesse del grafo di condivisione delle variabili.

Algorithm 37 Risoluzione tra clausole indicizzate

Signature: RESOLVE : IndexedClause × size_t × IndexedClause × size_t × IndexedClause& → bool

```
1: function RESOLVE(C_1, i_1, C_2, i_2, outResolvent)
2:     if i_1 ≥ C_1.length() ∨ i_2 ≥ C_2.length() then
3:         return false
4:     end if
5:     if ¬C_1.isSelected(i_1) ∨ ¬C_2.isSelected(i_2) then
6:         return false
7:     end if
8:     L_1 ← C_1[i_1].literal; L_2 ← C_2[i_2].literal
9:     if L_1 = NULL ∨ L_2 = NULL ∨ POLARITY(L_1) = POLARITY(L_2) then
10:        return false
11:    end if
12:    σ ← RobSubstitution()
13:    if ¬σ.unify(POSITIVE(L_1), 0, POSITIVE(L_2), 1) then
14:        return false
15:    end if
16:    R ← [ ]
17:    for all (C, b, pivot) ∈ {(C_1, 0, i_1), (C_2, 1, i_2)} do
18:        for j ← 0 to C.length() − 1 do
19:            if j = pivot then
20:                continue
21:            end if
22:            I ← (σ.apply(C[j].literal, b), C[j].index)
23:            if I ∉ R then
24:                R.push(I)                ▷ Duplicati: letterale e indice uguali
25:            end if
26:        end for
27:    end for
28:    if ∃(L, a), (¬L, b) ∈ R then
29:        return false                     ▷ Tautologia, indipendentemente dagli indici
30:    end if
31:    outResolvent ← IndexedClause(R)
32:    return true
33: end function
```

Algorithm 38 Fattorizzazione indicizzata

Signature: FACTOR : IndexedClause × size_t × size_t × IndexedClause& → bool

```
1:  function FACTOR(C, i_1, i_2, outFactor)
2:      if i_1 ≥ C.length() ∨ i_2 ≥ C.length() ∨ i_1 = i_2 then
3:          return false
4:      end if
5:      if ¬C.isSelected(i_1) then
6:          return false
7:      end if
8:      L_1 ← C[i_1].literal; L_2 ← C[i_2].literal
9:      if L_1 = NULL ∨ L_2 = NULL ∨ POLARITY(L_1) ≠ POLARITY(L_2) then
10:         return false
11:     end if
12:     σ ← RobSubstitution()
13:     if ¬σ.unify(POSITIVE(L_1), 0, POSITIVE(L_2), 0) then
14:         return false
15:     end if
16:     F ← [ ]
17:     for j ← 0 to C.length() − 1 do
18:         if j = i_1 then
19:             continue                    ▷ Si elimina il letterale selezionato
20:         end if
21:         I ← (σ.apply(C[j].literal, 0), C[j].index)
22:         if I ∉ F then
23:             F.push(I)
24:         end if
25:     end for
26:     outFactor ← IndexedClause(F)
27:     return true
28: end function
```

Algorithm 39 Verifica della sussunzione indicizzata

Signature: SUBSUMES : IndexedClause × IndexedClause → bool

```
1:  function SUBSUMES(C_1, C_2)
2:      if C_1.length() > C_2.length() then
3:          return false
4:      end if
5:      if C_1.length() = 0 then
6:          return true
7:      end if
8:      σ ← RobSubstitution()
9:      return MATCHSUBSUMPTIONLITERALS(0, C_1, C_2, σ)
10: end function
```

Algorithm 40 Matching ricorsivo dei letterali per la sussunzione

Signature: matchSubsumptionLiterals : size_t × IndexedClause × IndexedClause × RobSubstitution& → bool

```
1: function matchSubsumptionLiterals(i, C1, C2, σ)
2:     if i ≥ C1.length() then
3:         return true
4:     end if
5:     (L1, a1) ← C1[i]
6:     for j ← 0 to C2.length() − 1 do
7:         (L2, a2) ← C2[j]
8:         if a1 ≠ a2 ∨ polarity(L1) ≠ polarity(L2) then
9:             continue
10:        end if
11:        if functor(L1) ≠ functor(L2) then
12:            continue
13:        end if
14:        checkpoint ← σ.saveBacktrackState()
15:        if σ.match(positive(L1), 0, positive(L2), 1) then
16:            if matchSubsumptionLiterals(i + 1, C1, C2, σ) then
17:                σ.commit(checkpoint)
18:                return true
19:            end if
20:        end if
21:        σ.restore(checkpoint)
22:    end for
23:    return false
24: end function
```

4.3.3 Engine di Saturazione e Risoluzione (FO2Solver)

La classe FO2Solver (file FO2Solver.hpp e FO2Solver.cpp) gestisce prima le partizioni di clausola e poi esegue la saturazione **Given-Clause** sui rami che non richiedono ulteriori partizioni. Le partizioni sono esplorate con una lista di problemi pendenti, senza chiamate ricorsive del solver.

Per ogni problema pendente il solver cerca la prima clausola partizionabile. Se la clausola contiene letterali ground e letterali con variabili, crea una parte con tutti i letterali ground e una con quelli non ground. Negli altri casi costruisce le componenti connesse dei letterali collegati da variabili comuni: la prima componente forma una parte, le altre la seconda. Le variabili delle due parti sono disgiunte. Il solver costruisce due problemi derivati, in ognuno rimpiazza la clausola originale con una delle due parti, e mette entrambi nella lista di lavoro. Un ramo SAT conclude la ricerca con SAT; per concludere UNSAT devono risultare insoddisfacibili tutti i rami.

Su ciascun ramo che non ha altre partizioni, la procedura segue questi passaggi:

1. **Inizializzazione**: prima di saturare un ramo, il solver controlla la presenza di uguaglianze built-in nelle clausole; se ne trova, quel ramo riceve esito UNKNOWN. Nella saturazione verifica inoltre l'appartenenza iniziale a $\mathcal{S}_{2+i}$ e che i simboli funzionali abbiano arità uno; una violazione produce UNKNOWN. Le clausole ammesse vengono normalizzate rispetto agli indici delle variabili e inserite in passive. Una clausola vuota iniziale dà UNSATISFIABLE.

2. **Ciclo di Saturazione**: per ogni ramo non scindibile, finché la coda passive contiene clausole:

   - Estrae la clausola data (given). Se è la clausola vuota, interrompe la ricerca restituendo UNSATISFIABLE.
   - **Forward Subsumption**: verifica se given è sussunta da una clausola già attiva. In caso affermativo, la scarta.
   - **Backward Subsumption**: elimina dall'insieme delle clausole attive (active) quelle sussunte dalla clausola given.
   - **Factoring**: calcola tutti i fattori generabili da given ed aggiunge i risultati a passive.
   - **Risoluzione**: per ogni clausola attiva e per ogni coppia di pivot selezionati, prova entrambi gli ordinamenti delle clausole madri. Normalizza le variabili dei risolventi prima di inserirli in passive.
   - Inserisce la clausola given nell'insieme active.

3. **Terminazione**. La procedura termina con esito UNSATISFIABLE quando viene derivata la clausola vuota. Se il ciclo di saturazione esaurisce le clausole passive senza produrre una contraddizione, restituisce SATISFIABLE. Tale conclusione si fonda sulle proprietà di correttezza e completezza del calcolo di risoluzione indicizzata descritto da de Nivelle e Pratt-Hartmann e presuppone che l'implementazione rispetti le regole inferenziali, la funzione di selezione e gli invarianti

strutturali della classe $\mathcal{S}_{2+i}$ durante l'intero processo di saturazione. Per limitare l'utilizzo delle risorse, l'implementazione introduce inoltre un limite complessivo di 50.000 iterazioni, condiviso tra la saturazione e l'esplorazione dei rami generati dallo splitting. Al raggiungimento del limite, qualora la ricerca non sia stata completata, viene restituito `UNKNOWN`. Lo stesso risultato viene utilizzato quando vengono rilevate condizioni che impediscono l'applicazione della procedura specializzata. Nella gestione dello splitting, un ramo soddisfacibile permette di concludere la soddisfacibilità del problema iniziale. L'insoddisfacibilità viene invece restituita soltanto quando tutti i rami sono stati chiusi senza incontrare esiti indeterminati.

enqueue nel codice indica la routine interna del solver che filtra tautologie e duplicati, applica la sussunzione sulle clausole pendenti e aggiorna code e indici.

Algorithm 41 fo2solve - Gestione dei rami

Signature: SOLVE: Problem& → FO2Result

```
1: function SOLVE(prb)
2:     pending ← [prb.units()]
3:     sawUnknown ← false
4:     totalIterations ← 0
5:     MAX ← 50000
6:     while pending ≠ ∅ do
7:         if totalIterations ≥ MAX then
8:             sawUnknown ← true
9:             break
10:        end if
11:        totalIterations ← totalIterations + 1
12:        P ← Problem(pending.popBack())
13:        if ¬PROPAGATEUNITS(P) then
14:            continue                    ▷ Ramo chiuso per contraddizione
15:        end if
16:        split ← false
17:        for each clausola C ∈ P do
18:            r1 ← []; r2 ← []
19:            if FO2INFERENCES::PARTITIONCLAUSE(C, r1, r2) then
20:                P1 ← MAKESPLITBRANCH(P, C, r1)
21:                P2 ← MAKESPLITBRANCH(P, C, r2)
22:                pending.pushBack(P2.units())
23:                pending.pushBack(P1.units())
24:                split ← true
25:                break
26:            end if
27:        end for
28:        if split then
29:            continue
30:        end if
31:        result ← SOLVESATURATED(P, totalIterations, MAX)
32:        if result = SATISFIABLE then
33:            return SATISFIABLE
34:        end if
35:        if result = UNKNOWN then
36:            sawUnknown ← true
37:        end if
38:    end while
39:    if sawUnknown then
40:        return UNKNOWN
41:    end if
42:    return UNSATISFIABLE
43: end function
```

Algorithm 42 fo2solve - Saturazione del ramo corrente

```
1: function SOLVESATURATED(P, totalIterations, MAX)   ▷ totalIterations è passato per riferimento
2:     inizializza active, clausole pendenti e indici
3:     givenSteps ← 0
4:     for each clausola C ∈ P do
5:         icl ← INDEXEDCLAUSE::FROMCLAUSE(C)
6:         if TAUTOLOGICAL(icl) then
7:             continue
8:         end if
9:         FO2INFERENCES::NORMALIZEVARIABLES(icl, c)
10:        if c.length() = 0 then
11:            return UNSATISFIABLE
12:        end if
13:        ENQUEUE(c)
14:    end for
15:    while esistono clausole pendenti e totalIterations < MAX do
16:        totalIterations ← totalIterations + 1
17:        givenSteps ← givenSteps + 1
18:        if givenSteps mod 5 = 0 then
19:            estrai la clausola pendente più vecchia in given
20:        else
21:            estrai in given la clausola più corta, a parità di lunghezza la più vecchia
22:        end if
23:        if given.length() = 0 then
24:            return UNSATISFIABLE
25:        end if
26:        semplifica ripetutamente given con le unità attive tramite FO2INFERENCES::SIMPLIFYWITHUNIT
27:        if given.length() = 0 then
28:            return UNSATISFIABLE
29:        end if
30:        if SUBSUMEDBYACTIVE(given) then
31:            continue
32:        end if
33:        if given.length() = 1 then
34:            semplifica le clausole attive candidate con given tramite FO2INFERENCES::SIMPLIFYWITHUNIT
35:            if una semplificazione produce la clausola vuota then
36:                return UNSATISFIABLE
37:            end if
38:            inserisci i risultati con ENQUEUE e rimuovi le clausole attive sostituite
39:        end if
```

Algorithm 42 fo2solve - Saturazione del ramo corrente (continua)

```
40:         rimuovi le clausole attive sussunte da given usando
            FO2Inferences::subsumes
41:         for each i ∈ given.getSelectedLiteralIndices() do
42:             for j ← 0 to given.length() − 1 do
43:                 if i ≠ j e FO2Inferences::factor(given, i, j, f) then
44:                     FO2Inferences::normalizeVariables(f, c)
45:                     if c.length() = 0 then
46:                         return unsatisfiable
47:                     end if
48:                     enqueue(c)
49:                 end if
50:             end for
51:         end for
52:         for each i ∈ given.getSelectedLiteralIndices() do
53:             for each (act, j) ∈ resolutionCandidates(given[i]) do
54:                 for each (A, a, B, b) ∈ [(given, i, act, j), (act, j, given, i)] do
55:                     if FO2Inferences::resolve(A, a, B, b, r) then
56:                         FO2Inferences::normalizeVariables(r, c)
57:                         if c.length() = 0 then
58:                             return unsatisfiable
59:                         end if
60:                         enqueue(c)
61:                     end if
62:                 end for
63:             end for
64:         end for
65:         inserisci given in active e aggiorna gli indici
66:     end while
67:     if esistono ancora clausole pendenti then
68:         return unknown
69:     end if
70:     return satisfiable
71: end function
```

— 5 —

Risultati sperimentali

la parte finale di questo lavoro di tesi, è incentrata sulla valutazione della procedura di decisione implementata, confrontando i risultati ottenuti con la nuova mode `fo2`, rispetto a quelli ottenuti con la versione originale di Vampire. Il principale standard di riferimento usato per il test è stata la libreria TPTP. La libreria standard **TPTP (Thousands of Problems for Theorem Provers)** è una libreria di problemi utilizzati come test per i sistemi ATP (automated theorem proving). Essa fornisce un elenco completo di problemi, al fine di fornire un meccanismo di riferimento per i test, semplice e non ambiguo. TPTP contiene molti problemi in diversi formati, dalla logica del primo ordine alla logica di ordine superiore o problemi con tipi estesi. Ai fini della ricerca effettuata, è stato ristretto il campo d'azione ai soli problemi di tipo *first-order-formula* (FOF) e *clause-normal-form* (CNF).

I problemi FOF sono espressi come lista di formule, utilizzano una sintassi leggibile dove ogni formula è racchiusa all'interno dello statement `fof(nome, role, formula)` dove il `nome` indica un identificatore univoco, `role` è lo scopo della formula (es assioma, congettura, congettura negata ecc) e la `formula` che contiene l'espressione logica. Le variabili sono rappresentate da lettere in maiuscolo `(X1, VarName)`. I connettivi logici includono & ($\wedge$), | ($\vee$), => ($\implies$), <=> ($\iff$) e ~ ($\neg$). I quantificatori sono denotati da ! per il $\forall$ e ? per $\exists$, la lista di variabili quantificate segue i quantificatori all'interno delle parentesi quadre (es, `[X1, X2]`). Le parentesi sono usate per assicurare il corretto ordine delle operazioni e limitarne lo scope, ad esempio la formula $\forall x(P(x) \implies Q(f(x)))$ diventa nel formato TPTP FOF:

```
fof(f1, axiom, ![X] (P(X) => Q(f(X)))).
```

I problemi CNF hanno una struttura simile l'unica differenza è che invece di avere delle formule, esse contengono clausole espresse come disgiunzione di letterali, ad esempio il problema precedente verrebbe rappresentato in questo modo:

```
cnf(c1, axiom, (¬P(X) | Q(f(X)))).
```

I problemi tptp sono raggruppati in sottocartelle in base al loro dominio di interesse per esempio `ALG` per problemi algebrici, `COL` problemi di logica combinatoria, `HWC` creazione hardware o `SYN` per problemi di sintassi.

5.0.1 Creazione del benchmark TPTP

Nello specifico per la classificazione, è stata utilizzata una delle utility ufficiali della libreria TPTP, `tptp2T`. Si tratta di uno script in grado di effettuare data-mining e filtraggio avanzato sull'intero corpo di problemi senza dover scansionare i file manualmente. La libreria TPTP conta più di 25000 problemi, utilizzando tptp2T abbiamo quindi potuto effettuare una prima scrematura iniziale andando ad eliminare a priori i problemi che violavano i requisiti algebrici di base come: presenza

di funzioni, assenza o presenza di uguaglianza, formati diversi da quelli di nostro interesse. I comandi utilizzati sono stati in particolare:
`tptp2T -q2 -pp Form FOF Equality -Propositional FunctionArity - 0`
`tptp2T -q2 -pp Form CNF Equality -Propositional FunctionArity - 0` Il dataset TPTP è stato così ristretto a 770 candidati iniziali per i problemi con uguaglianza in formato FOF e 178 per i problemi con uguaglianza di tipo CNF Successivamente è stato necessario un ulteriore filtraggio, in quanto i problemi individuati dallo script, non erano in grado di soddisfare tutti i requisiti del frammento . Infatti, con tptp2T non è stato possibile verificare la correttezza del numero di variabili (<= 2) in quanto non era possibile selezionare problemi che riutilizzavano le uniche due variabili presenti. Per questo motivo i problemi inizialmente individuati, sono stati processati dal classificatore implementato `fo2_classifier`, il quale oltre il controllo sull'uguaglianza è in grado di verificare la struttura dei quantificatori e assicurarsi che in ogni sottoformula non vengano mai usate più di 2 variabili.
I test sono stati eseguiti su pc dotato di processore Intel Core i5-8250U a 1.60GHz e 12GB di RAM, con sistema operativo Ubuntu 25.10, impostando timer di 60/30/10 secondi.

5.0.2 Classificatore $FO^2_=$

Le varie fasi di parsing e preprocessing comportano un costo computazionale e un incremento della dimensione delle formule, per questo motivo ispezionare preventivamente il problema consente di scartare a priori problemi non appartenenti al frammento di riferimento ($FO^2_=$), evitando quindi di lavorare su formule non decidibili con questa procedura.

Come riportato prima, lo script della libreria tptp2T non ci permette di selezionare da subito problemi completamente adeguati per testare la procedura, per questo motivo la componente classifier si occupa quindi di completare l'analisi sintattica dei problemi, stabilendo se effettivamente appartengono o meno al frammento a due variabili con uguaglianza. in oltre il classificatore è in grado di rilevare se il problema presenta o meno l'uguaglianza, in questo modo sarà possibile determinare a priori se andrà applicata o no la procedura di rimozione dell'uguaglianza vista nei Capitoli 2 e 4.

Il punto di ingresso è il metodo statico: isFO2, il quale preso in input il problema `Problem` va ad analizzare tutte le unità che lo compongono. In base al tipo dell'unità analizzata verrà invocata una specifica funzione ausiliaria:

1. Se l'unità è in forma *Clausale*: invoca la funzione ausiliaria isFO2Clause 44.

2. Se l'unità è una *Formula*: allora invoca la funzione ricorsiva isFO2Formula 45

Algorithm 43 FO2 Main Classification

```
1: signature isFO2 : UnitList ∗ ×Boolean& → Boolean
2: function isFO2(ul, hasEq)
3:     hasEq ← false
4:     while ul ≠ ∅ do
5:         u ← ul.next()
6:         vars ← ∅                          ▷ Insieme resettato per ogni unità
7:         flag ← true
8:         if u.isClause() then
9:             flag ← isFO2Clause(u.asClause(), vars, hasEq)
10:        else
11:            flag ← isFO2Formula(u.getFormula(), vars, hasEq)
12:        end if
13:        if ¬flag then
14:            return false                  ▷ Early exit se non è FO2
15:        end if
16:    end while
17:    return true
18: end function
```

Algorithm 44 FO2 Clause Check

Require: $clause$: Clause$*$, $vars$: DHSet$\langle$unsigned$\rangle$, $hasEq$: Boolean
Ensure: Boolean isFO2Clause: Clause $*$ $\times$DHSet$\langle$unsigned$\rangle$ $\times$ Boolean& $\rightarrow$ Boolean

```
1:  function isFO2Clause(clause, vars, hasEq)
2:      for i ← 0 to clause → length() − 1 do
3:          vars.reset()                ▷ Insieme di variabili resettato per ogni letterale
4:          lit ← (∗clause)[i]
5:          if lit = NULL then
6:              return false
7:          end if
8:          if lit → isEquality() then
9:              hasEq ← true
10:         end if
11:         for a ← 0 to lit → arity() − 1 do
12:             tl ← lit → nthArgument(a)
13:             if tl = NULL then
14:                 return false
15:             end if
16:             if tl → isVar() then
17:                 vars.insert(tl → var())
18:                 if |vars| > 2 then
19:                     return false        ▷ Più di 2 variabili nello stesso letterale
20:                 end if
21:             else if tl → isTerm() then
22:                 t ← tl → term()
23:                 if t = NULL ∨ t → arity() > 0 then
24:                     return false        ▷ Funzioni con arietà > 0 vietate in FO2
25:                 end if
26:             else
27:                 return false
28:             end if
29:         end for
30:     end for
31:     return true
32: end function
```

Algorithm 45 FO2 Formula Recursive Check

Require: f : Formula$*$, $varSet$: DHSet$\langle$unsigned$\rangle$, $hasEq$: Boolean **Signature:**isFO2Formula: Formula $*$ $\times$DHSet$\langle$unsigned$\rangle$& $\times$ Boolean& $\rightarrow$ Boolean

1: **function** isFO2Formula($f, varSet, hasEq$)
2: **if** $f =$ NULL **then**
3: **return false**
4: **end if**
5: $varSet$.reset()
6: **switch** $f \rightarrow$ connective() **do**
7: **case** AND, OR **do**
8: **for each** $sub \in f \rightarrow$ args() **do**
9: $childVars \leftarrow \varnothing$
10: **if** $\neg$isFO2Formula($sub, childVars, hasEq$) **then**
11: **return false**
12: **end if**
13: $varSet \leftarrow varSet \cup childVars$
14: **if** $|varSet| > 2$ **then**
15: **return false**
16: **end if**
17: **end for**
18: **case** FORALL, EXISTS **do**
19: **if** $\neg$isFO2Formula($f \rightarrow$ qarg(), $varSet, hasEq$) **then**
20: **return false**
21: **end if**
22: **for each** $v \in f \rightarrow$ vars() **do**
23: $varSet$.remove(v.first)
24: **end for**
25: **return true**
26: **case** BOOL_TERM **do**
27: $ts \leftarrow f \rightarrow$ getBooleanTerm()
28: **if** $\neg ts$.isVar() **then**
29: **return false**
30: **end if**
31: $varSet$.insert(ts.var())
32: **return** $|varSet| \leq 2$
33: **case** NOT **do**
34: **return** isFO2Formula($f \rightarrow$ uarg(), $varSet, hasEq$)
35: **case** IMP, IFF **do**
36: $rightVars \leftarrow \varnothing$
37: **if** $\neg$isFO2Formula($f \rightarrow$ left(), $varSet, hasEq$) **then**
38: **return false**
39: **end if**
40: **if** $\neg$isFO2Formula($f \rightarrow$ right(), $rightVars, hasEq$) **then**
41: **return false**
42: **end if**
43: $varSet \leftarrow varSet \cup rightVars$
44: **return** $|varSet| \leq 2$

Algorithm 45 FO2 Formula Recursive Check (continua)

```
32: case LITERAL do
33: lit ← f → literal()
34: if lit = NULL then
35:     return false
36: end if
37: if lit → isEquality() then
38:     hasEq ← true
39: end if
40: for i ← 0 to lit → arity() − 1 do
41:     tl ← lit → nthArgument(i)
42:     if tl = NULL then
43:         return false
44:     end if
45:     if tl → isVar() then
46:         varSet.insert(tl → var())
47:         if |varSet| > 2 then
48:             return false
49:         end if
50:     else if tl → isTerm() then
51:         t ← tl → term()
52:         if t = NULL ∨ t → arity() > 0 then
53:             return false
54:         end if
55:     else
56:         return false
57:     end if
58: end for
59: return true
60: case FALSE, TRUE do
61: return true
62: default do
63: return false
64: =0
```

Quindi tutti i problemi ottenuti dallo script TPTP sono stati poi processati dal classificatore il quale individuato tra i problemi candidati 21 problemi processabili nel formato FOF e 56 per il formato CNF.

Tabella 5.1: problemi FOF e CNF con uguaglianza

	FOF	**CNF**
tptp2T	413	178
classificatore fo2	21	18

I problemi con uguaglianza, sono stati confrontati su tre modalità:

- **FO2**: `fo2_remove_equality` ovvero modalità implementata.
- **Otter**: `--saturation_algorithm otter -av off -updr off -bs on -frs off`: che è la modalità classica di Vampire, descritta alla sezione 5.2
- **FO2_remove_equality**: modalità che va a combinare le due modalità precedenti: ovvero, che per ogni problema va prima ad eseguire i passaggi di rimozione dell'uguaglianza implementati nel modulo RemoveEquality e poi prosegue con la risoluzione prevista dalla modalità Otter (no CASC)

Lettura dei grafici

La configurazione *Classic* indica Vampire con saturazione Otter; `FO2` indica la procedura specializzata.

Un problema è conteggiato come deciso quando l'output registra SAT o UNSAT. Timeout e Unknown non sono decisioni. La copertura entro un limite di risorse non misura la completezza teorica del calcolo. Nei cactus plot l'ascissa conta le istanze decise e l'ordinata riporta i tempi individuali ordinati, in secondi e in scala logaritmica; Una curva più estesa indica maggiore copertura. Curve più basse indicano tempi inferiori, ma possono riferirsi a insiemi diversi di problemi. Gli scatter plot qui riportati mostrano, per ciascuna configurazione, il tempo, con colori distinti per SAT, UNSAT, timeout e altri esiti.

Confronto su problemi FOF con uguaglianza

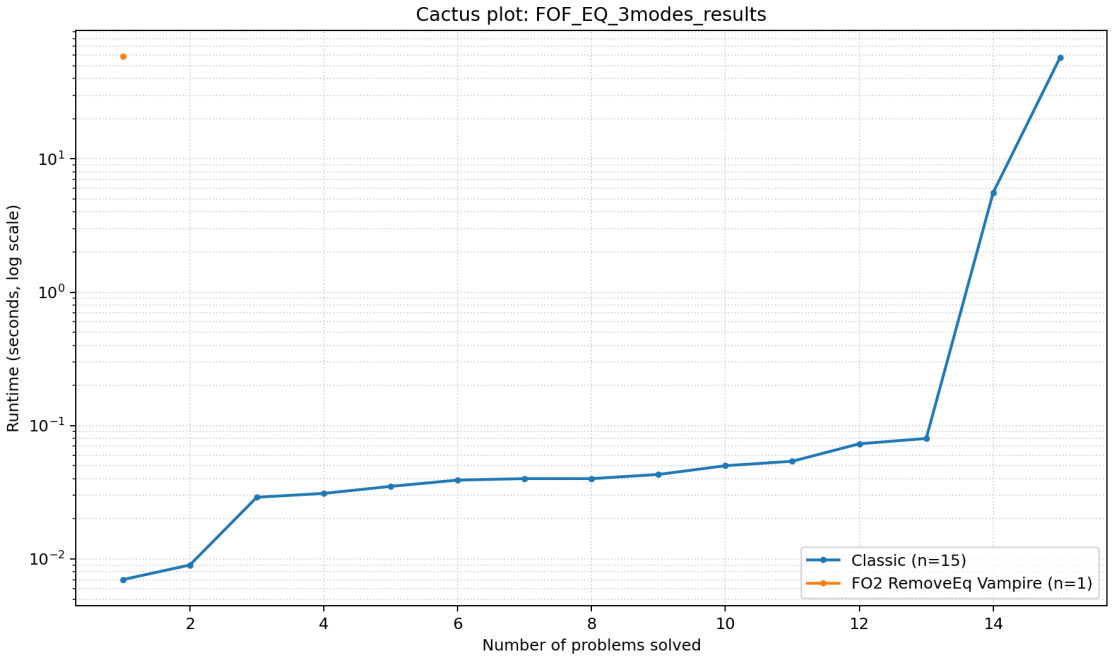


Figura 5.1: Cactus plot delle configurazioni sul benchmark FOF con uguaglianza (18 istanze).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	0	0	12	6	0/18 (0.0%)
Classic	2	13	3	0	15/18 (83.3%)
RemoveEq	1	0	17	0	1/18 (5.6%)

Tabella 5.2: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

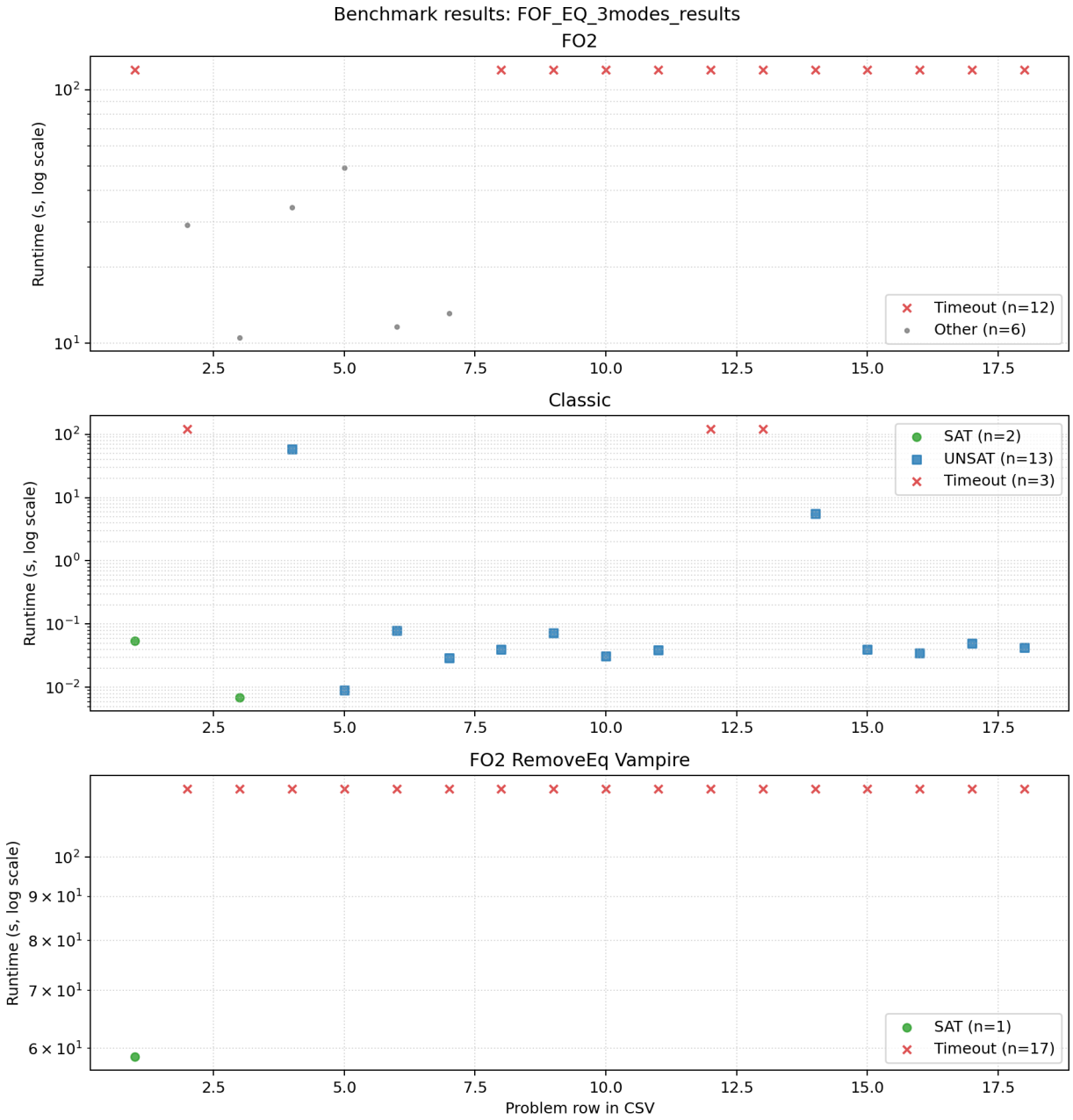


Figura 5.2: Tempi per istanza e categoria di esito sul benchmark FOF con uguaglianza. L'ascissa è l'indice della riga nel CSV; l'ordinata è logaritmica.

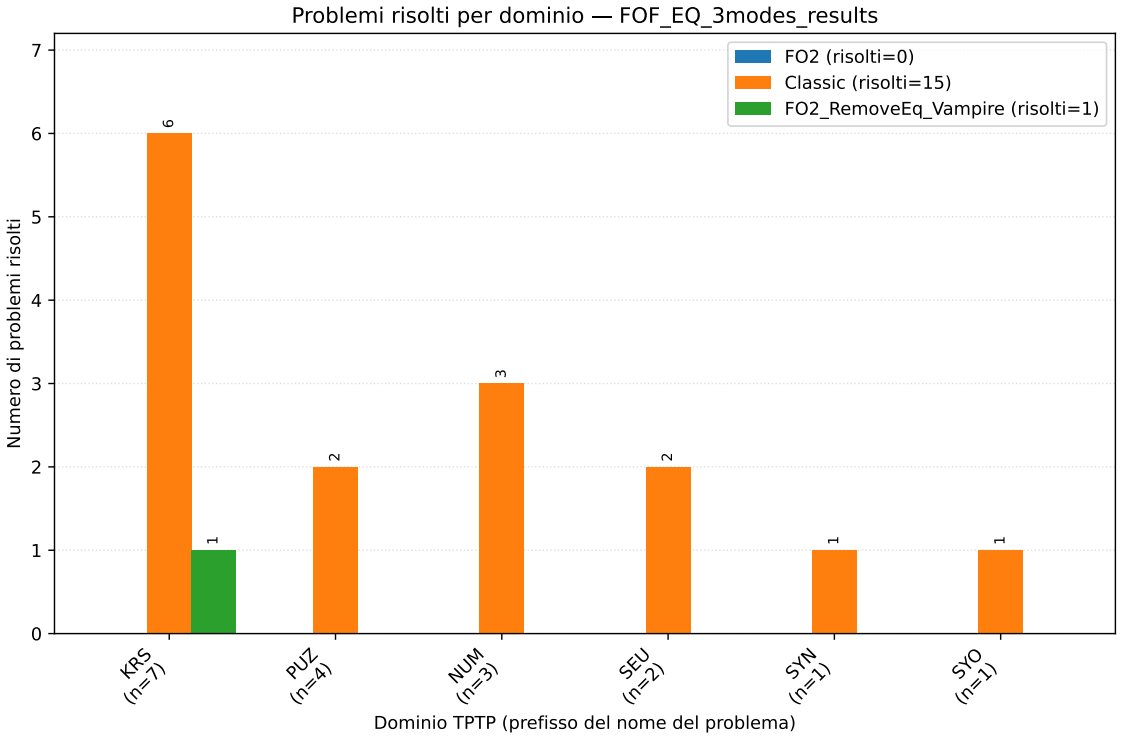


Figura 5.3: Numero di problemi decisi per dominio TPTP sul benchmark FOF con uguaglianza.

Classic decide 15 istanze su 18. FO2 registra 12 timeout e 6 Unknown. RemoveEq decide soltanto `KRS026+1.p`, in 58.668 s, contro 0.054 s di Classic.

Confronto su problemi CNF con uguaglianza

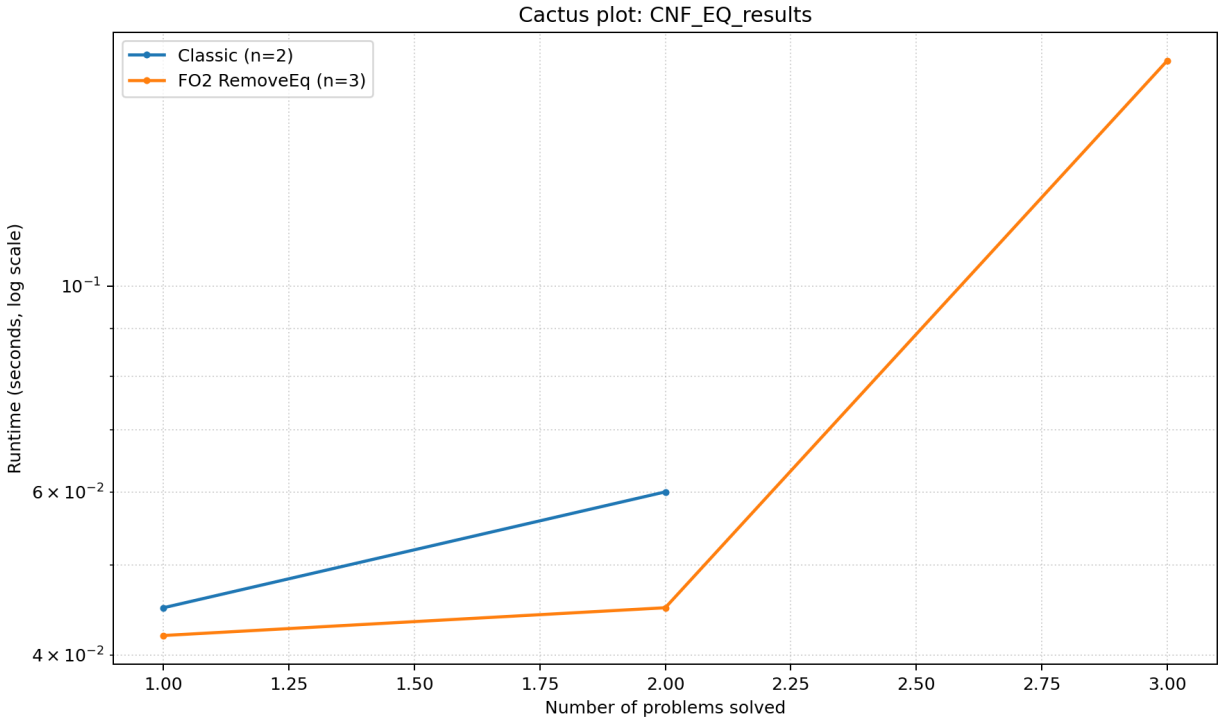


Figura 5.4: Cactus plot delle configurazioni sul benchmark CNF con uguaglianza (18 istanze).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	0	0	18	0	0/18 (0.0%)
Classic	0	2	16	0	2/18 (11.1%)
RemoveEq	0	3	15	0	3/18 (88.9%)

Tabella 5.3: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

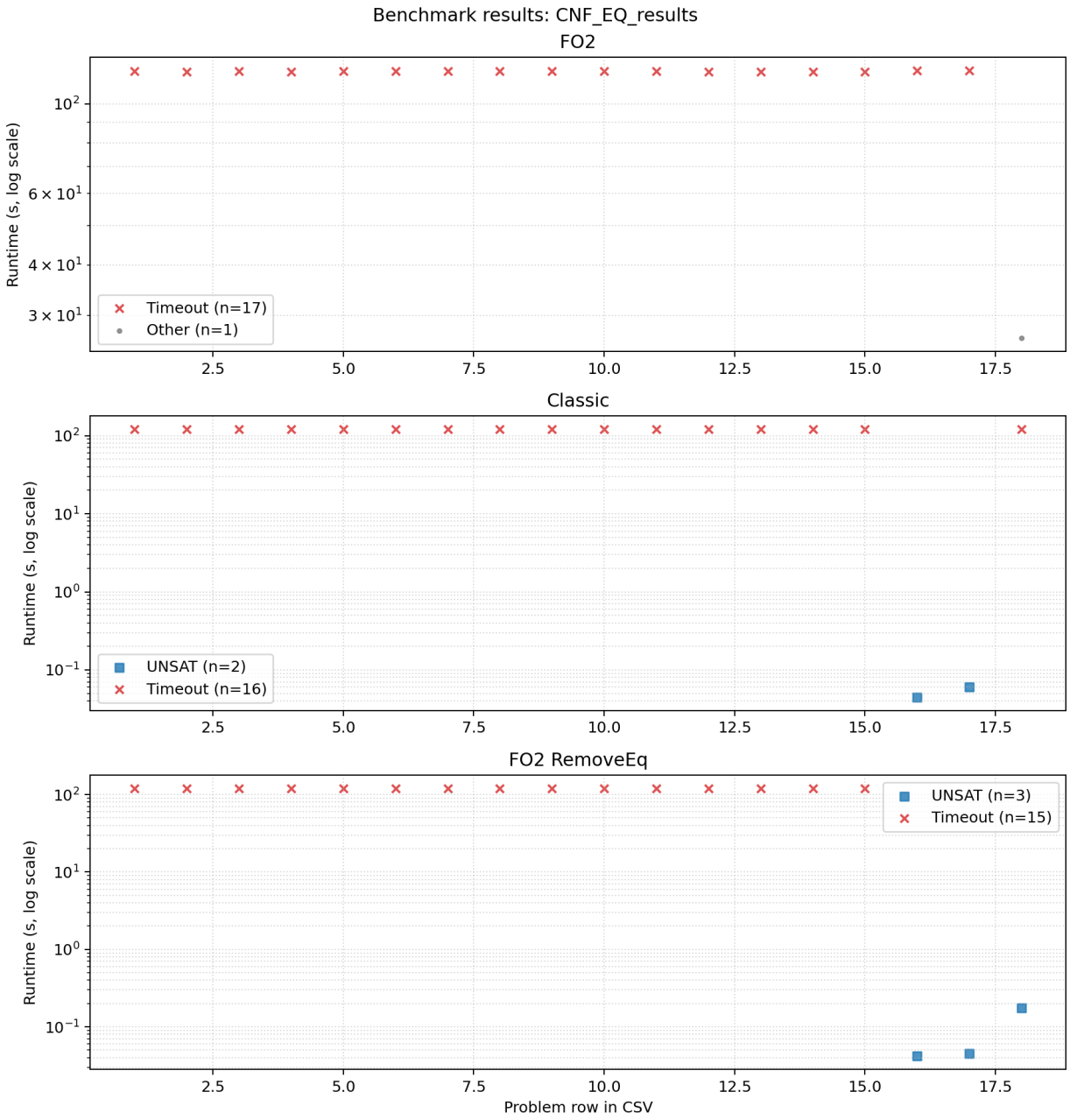


Figura 5.5: Tempi per istanza e categoria di esito sul benchmark CNF con uguaglianza. L'ascissa è l'indice della riga nel CSV; l'ordinata è logaritmica.

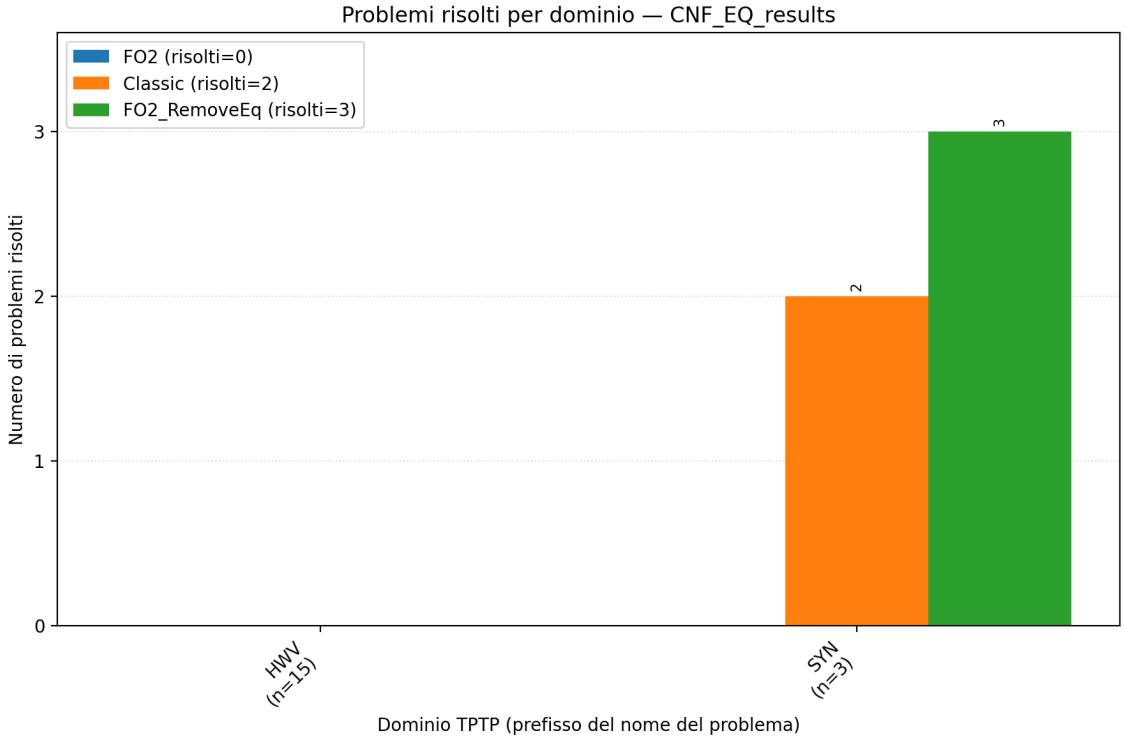


Figura 5.6: Numero di problemi decisi per dominio TPTP sul benchmark CNF con uguaglianza.

FO2 esaurisce il tempo su tutte le 18 istanze; Classic decide soltanto due problemi mentre RemoveEq decide queste due formule e in più solo un'altra, per un totale di 3/18. Il vantaggio osservato riguarda quindi il problema trasformato seguito da Otter.

Essendo che il nostro lavoro prevedeva l'individuazione di una procedura di decisione anche per il frammento a due variabili senza uguaglianza, è stato effettuato lo stesso lavoro di duplice filtraggio anche per i problemi senza uguaglianza, ottenendo con tptp2t 1157 problemi FOF e 711 problemi CNF e dopo l'applicazione del classificatore implementato abbiamo ottenuto 359 problemi FOF e 51 problemi CNF.

Tabella 5.4: problemi FOF e CNF senza uguaglianza

	FOF	**CNF**
tptp2T	1156	710
classificatore fo2	349	51

Confronto su problemi FOF senza uguaglianza

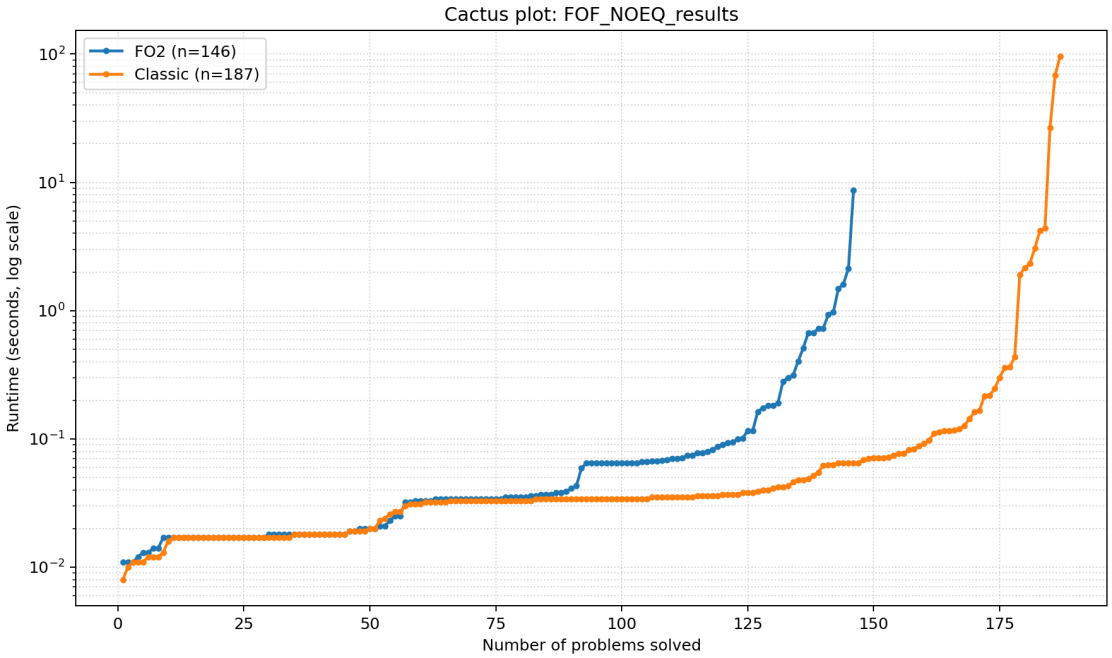


Figura 5.7: Cactus plot delle configurazioni sul benchmark FOF senza uguaglianza (343 istanze).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	32	114	59	138	146/343 (42.6%)
Classic	38	149	156	0	187/343 (54.5%)

Tabella 5.5: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

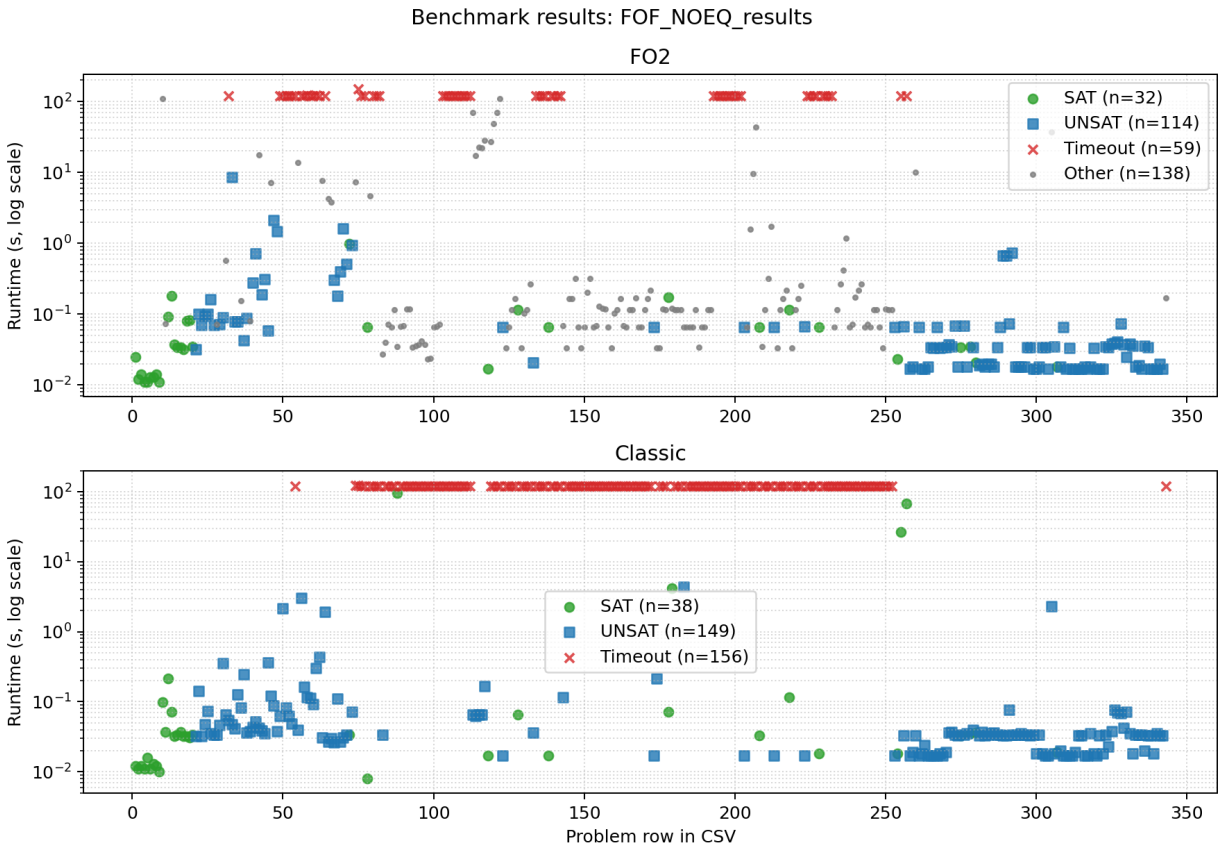


Figura 5.8: Tempi per istanza e categoria di esito sul benchmark FOF senza uguaglianza. L'ascissa è l'indice della riga nel CSV; l'ordinata è logaritmica.

FO2 e Classic decidono congiuntamente 146 istanze e si registrano 0 disaccordi SAT/UNSAT.

Classic decide 187/343 problemi (54.5%), FO2 146/343 (42.6%). Tutte le decisioni di FO2 appartengono all'insieme deciso da Classic: il divario è di 41 istanze.

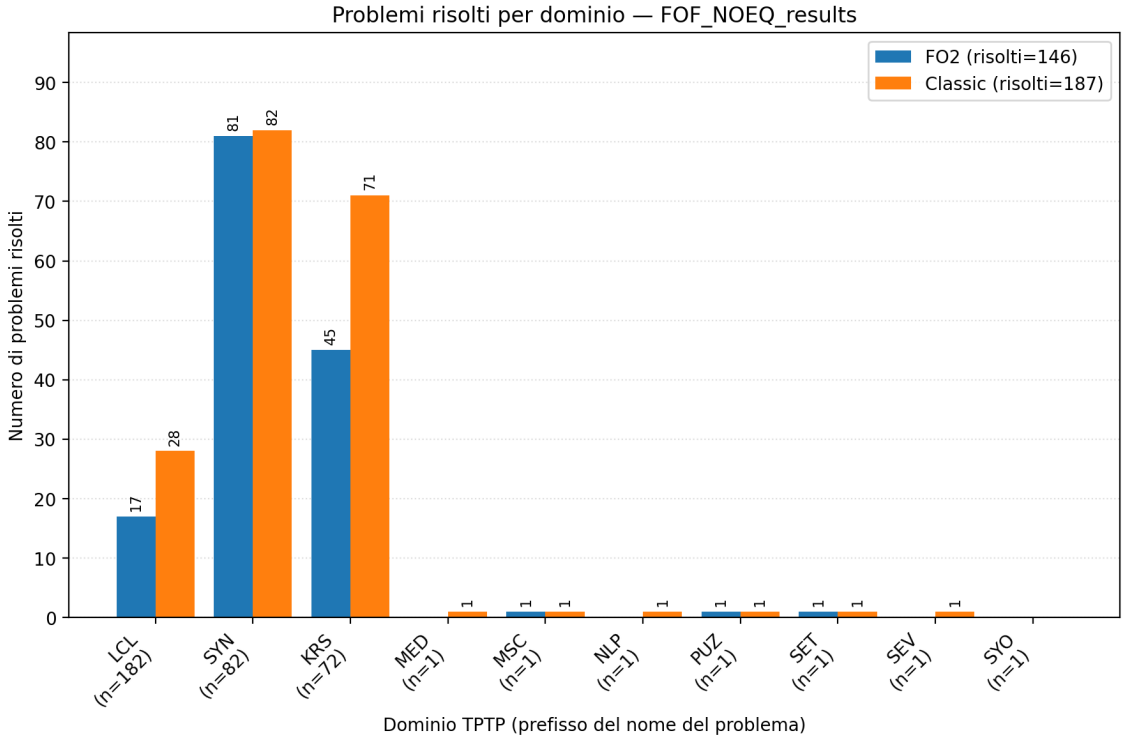


Figura 5.9: Numero di problemi decisi per dominio TPTP sul benchmark FOF senza uguaglianza.

Confronto su problemi CNF senza uguaglianza

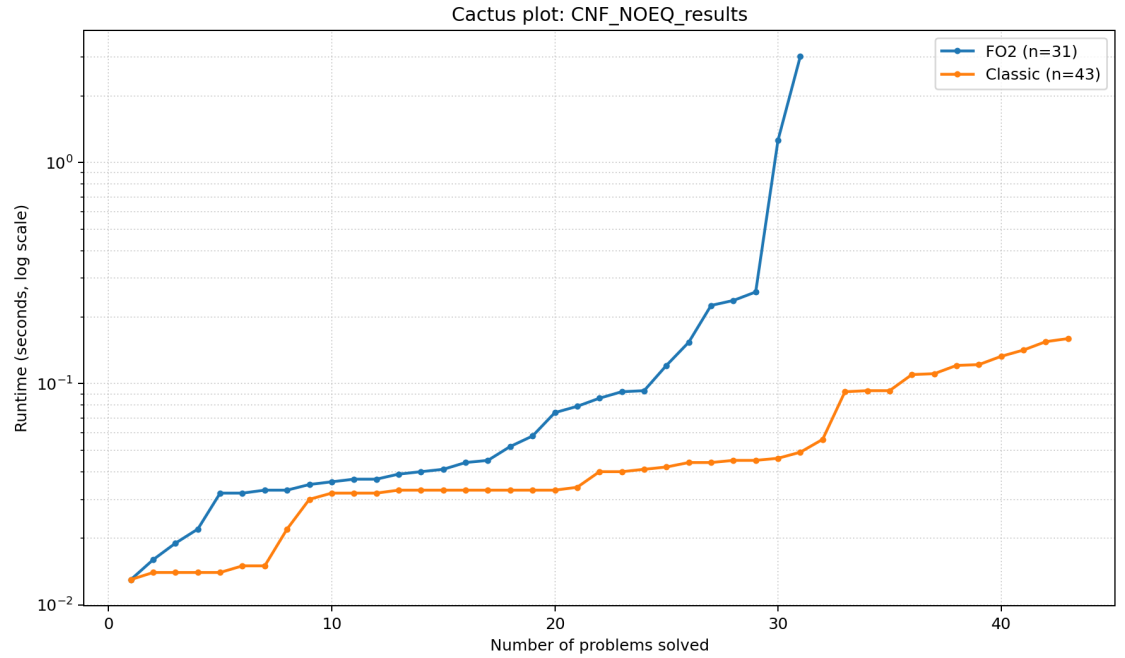


Figura 5.10: Cactus plot delle configurazioni sul benchmark CNF senza uguaglianza (51 istanze).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	5	26	19	1	31/51 (60.8%)
Classic	13	30	8	0	43/51 (84.3%)

Tabella 5.6: Esiti registrati nel CSV per CNF_NOEQ con limite temporale di 2 minuti; *Altri* comprende Unknown ed Error.

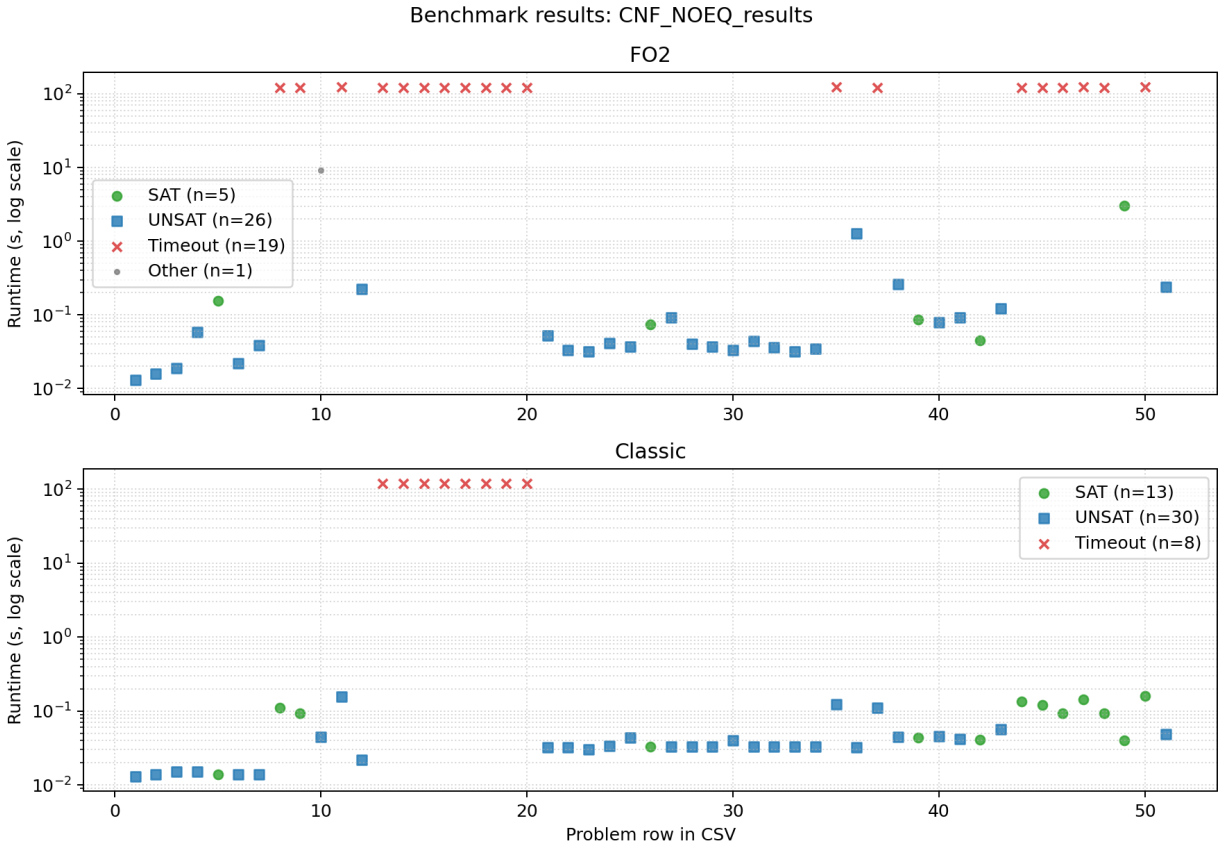


Figura 5.11: Tempi per istanza e categoria di esito sul benchmark CNF senza uguaglianza. L'ascissa è l'indice della riga nel CSV

FO2 e Classic decidono congiuntamente 31 istanze; 0 disaccordi SAT/UNSAT.

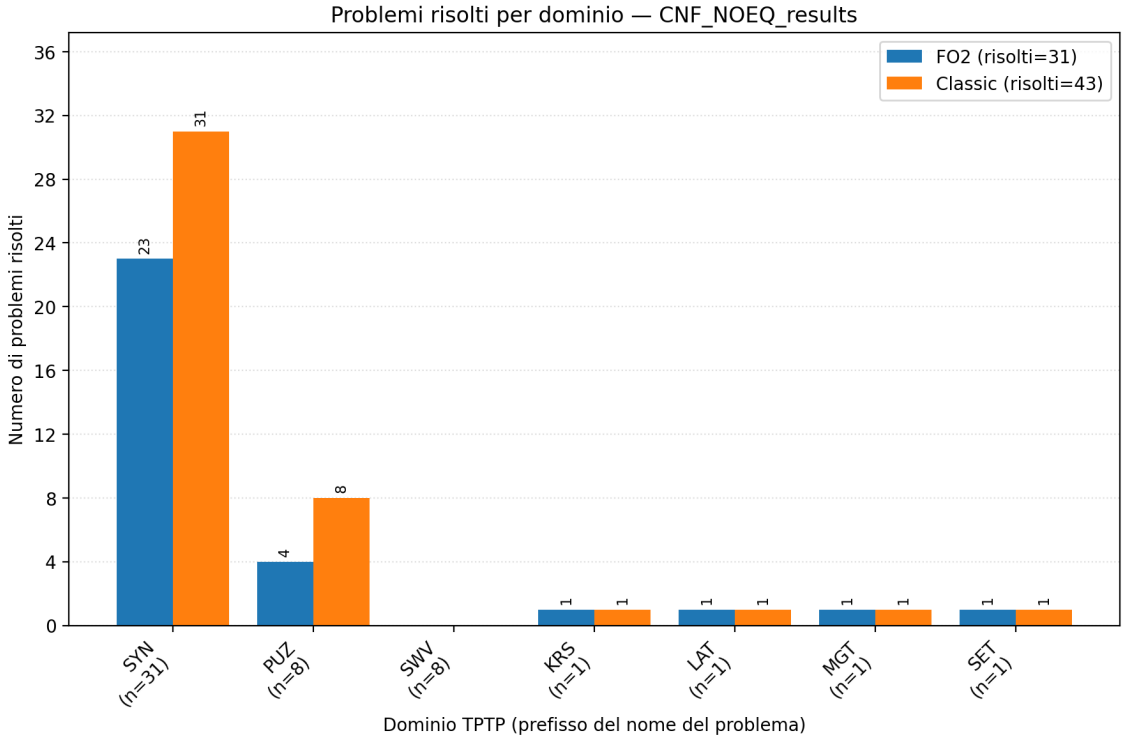


Figura 5.12: Numero di problemi decisi per dominio TPTP sul benchmark CNF senza uguaglianza.

Entrambe le configurazioni decidono le medesime 43 istanze: 30 UNSAT e 13 SAT. Gli otto timeout appartengono a SWV. La mediana sul campione comune è maggiore per FO2; tale differenza descrive il costo complessivo dell'implementazione e non identifica da sola quale componente lo produca.

5.1 Dataset sintetici generati

Oltre all'estrazione di problemi di letteratura dalla libreria TPTP, l'analisi sperimentale si avvale di una suite di problemi sintetici generati automaticamente in modo parametrico. Questi dataset provengono dal lavoro di tesi di **Manuel Borgia** (*"Generazione automatica di formule per frammenti decidibili"*)[Bor], incentrato sullo sviluppo di un generatore configurabile di formule FO^2 in grado di sollecitare in modo specifico le diverse componenti dei solutori automatici di teoremi.

I problemi sintetici sono generati definendo una struttura di configurazione `GenConfig` che controlla la modalità di generazione (`GenMode`), la profondità dell'albero di sintassi astratta (`depth`), la dimensione e la tipologia del vocabolario dei predicati (`vocab`), il budget di quantificatori e connettivi, e la dimensione del dominio per la ricerca di modelli finiti (`domainSize`).

5.1.1 Dimensioni di Scaling Sintattico e Strutturale

La generazione del dataset per il frammento a due variabili (FO^2) è suddivisa in sei dimensioni:

- **Vocabulary Scaling**: Espansione progressiva dei predicati distinti da $5/1,\ 2/2$ fino a $20/1,\ 10/2$ a una profondità fissa pari a 12.
- **Depth Scaling**: Scalatura della profondità dell'albero sintattico attraverso $d \in \{5, 8, 12, 16\}$ impiegando un vocabolario standard $(15/1,\ 10/2)$.
- **Quantifier Scaling**: Incremento del budget dei quantificatori $qb \in \{2, 4, 6, 8\}$, con la profondità regolata dinamicamente come $qb \times 2 + 10$.
- **Equality Scaling**: Incremento progressivo del budget dell'operatore di uguaglianza $eq \in \{2, 4, 6, 8\}$ a profondità 18 con un vocabolario specializzato $(5/1,\ 15/2)$.
- **Finite Domain Scaling (SATBUILD)**: Valutazione della robustezza nella generazione di modelli finiti tramite la scalatura esplicita della dimensione del dominio $ds \in \{1, 2, 3\}$ a profondità 12 con un vocabolario minimale $(2/1,\ 1/2)$.
- **Dense Equality & IFF**: Stress-test su interazioni dense di uguaglianze ed equivalenze con budget elevati $eq \in \{10, 16, 20, 26, 30\}$ a profondità 30 impiegando il vocabolario $2/1,\ 2/2$.

5.1.2 test2Equality

La campagna comprende 370 problemi nominalmente SAT o UNSAT con uguaglianza densa.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	1	8	24	337	9/370 (2.4%)
Classic	111	148	111	0	259/370 (70.0%)
RemoveEq	31	132	207	0	163/370 (44.1%)

Tabella 5.7: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus plot per classe, i verdetti risolti sono FO2: 9, Classic: 259, FO2_RemoveEq: 155.

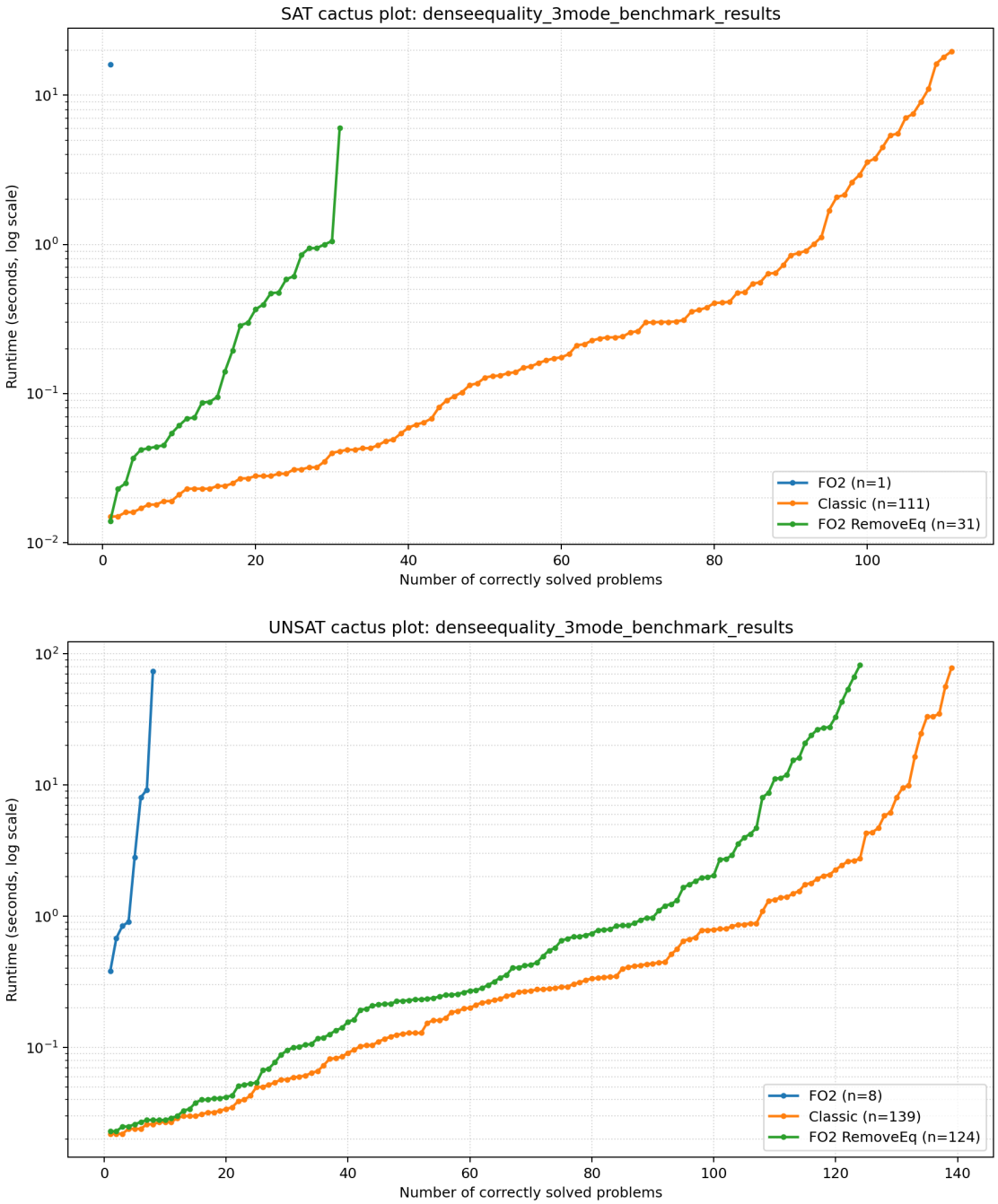


Figura 5.13: Cactus plot di test2Equality (370 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

5.1.3 test2Mix

Dense Equality

370 problemi, con livelli nominali di uguaglianza eq10, eq16, eq20, eq26 ed eq30. La famiglia sollecita le trasformazioni e la saturazione in presenza di numerosi vincoli di uguaglianza.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	0	0	368	2	0/370 (0.0%)
Classic	114	139	117	0	253/370 (68.4%)
RemoveEq	0	1	369	0	1/370 (0.3%)

Tabella 5.8: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti coerenti con il nome del file sono FO2: 0, Classic: 247, FO2_RemoveEq: 1. Le risposte discordanti con la classe nominale sono rispettivamente 0, 6, 0. Una discordanza richiede la verifica della formula e del generatore: non prova da sola un errore del solutore.

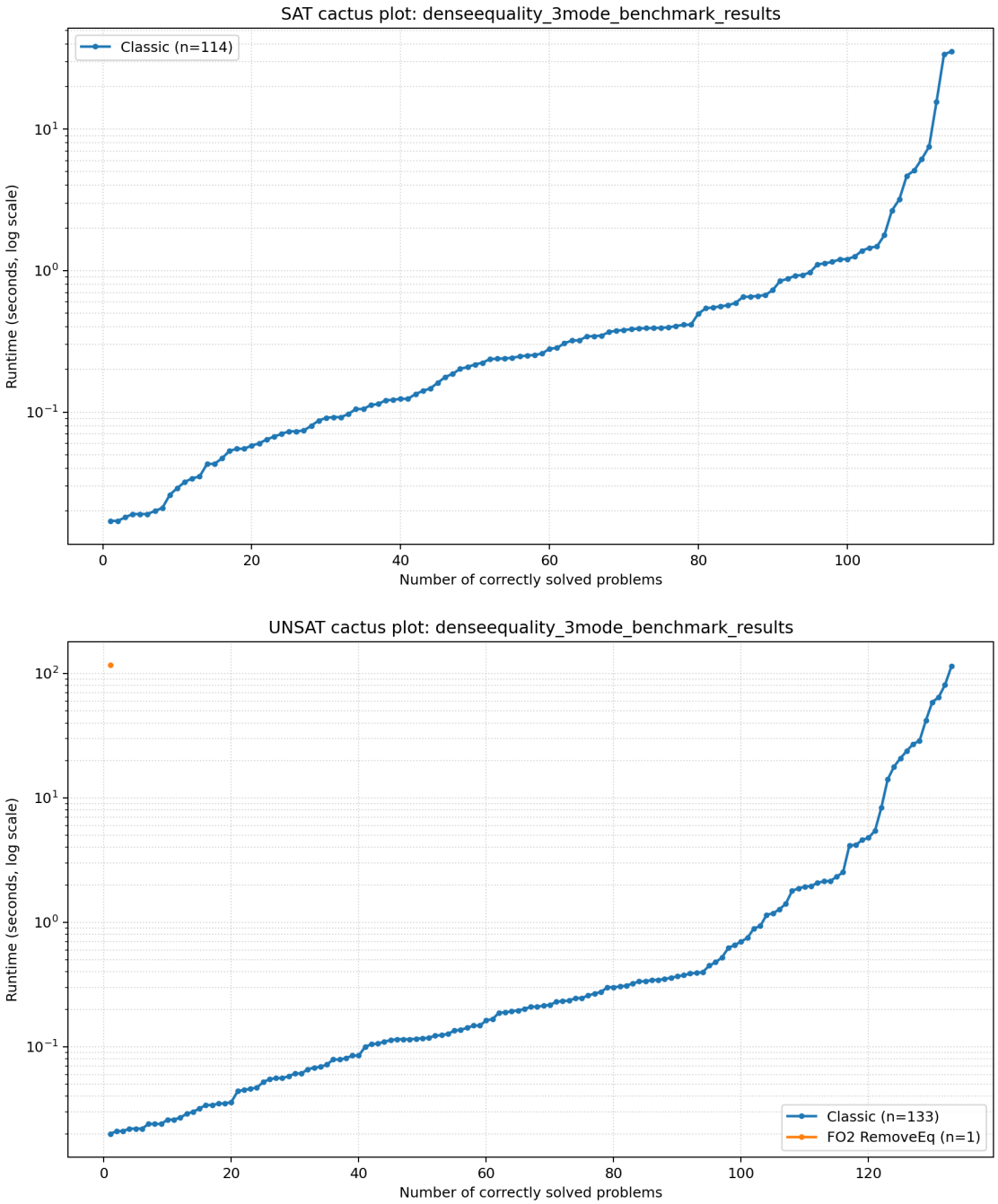


Figura 5.14: Cactus plot di Dense Equality (370 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

Il divario tra Classic e FO2 è ampio; anche RemoveEq decide una sola formula in questa campagna. La densità di uguaglianza è quindi una difficoltà empirica per queste configurazioni, senza che il CSV identifichi quale trasformazione o inferenza domini il costo.

Depth Scaling

296 problemi ai livelli d5, d8, d12 e d16. Il parametro riguarda la profondità sintattica nominale.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	45	54	1	196	99/296 (33.4%)
Classic	140	153	3	0	293/296 (99.0%)
RemoveEq	117	134	45	0	251/296 (84.8%)

Tabella 5.9: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i problemi risolti sono FO2: 98, Classic: 293, FO2_RemoveEq: 253.

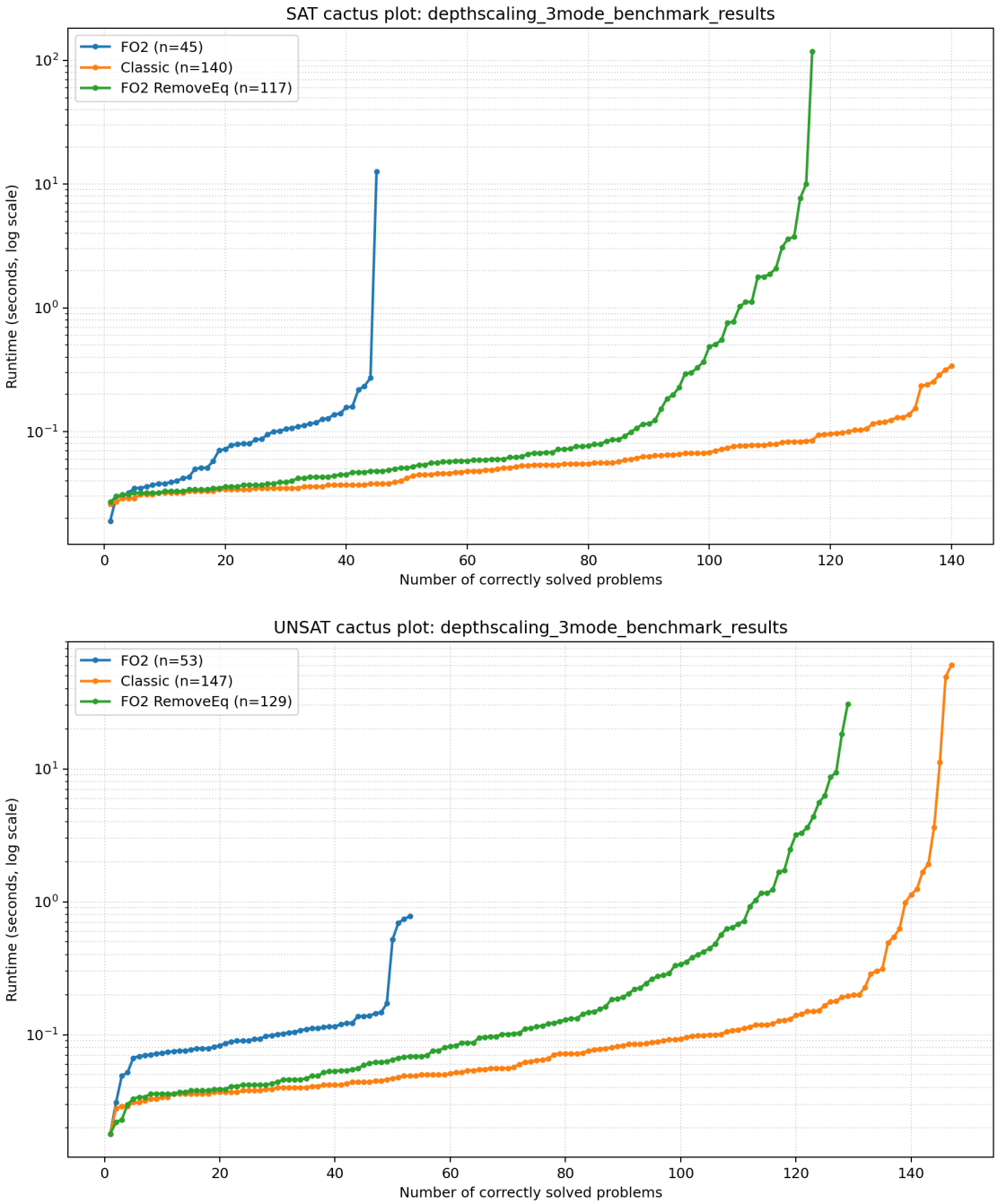


Figura 5.15: Cactus plot di Depth Scaling (296 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

Classic copre quasi tutto il dataset. Fo2 presenta molti risultati di Unknow/Timeout

Equality Scaling

296 problemi ai livelli eq2, eq4, eq6 ed eq8, che variano il budget nominale di uguaglianza.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	0	0	188	108	0/296 (0.0%)
Classic	117	139	40	0	256/296 (86.5%)
RemoveEq	34	69	193	0	103/296 (34.8%)

Tabella 5.10: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti coerenti con il nome del file sono FO2: 0, Classic: 256, FO2_RemoveEq: 103.

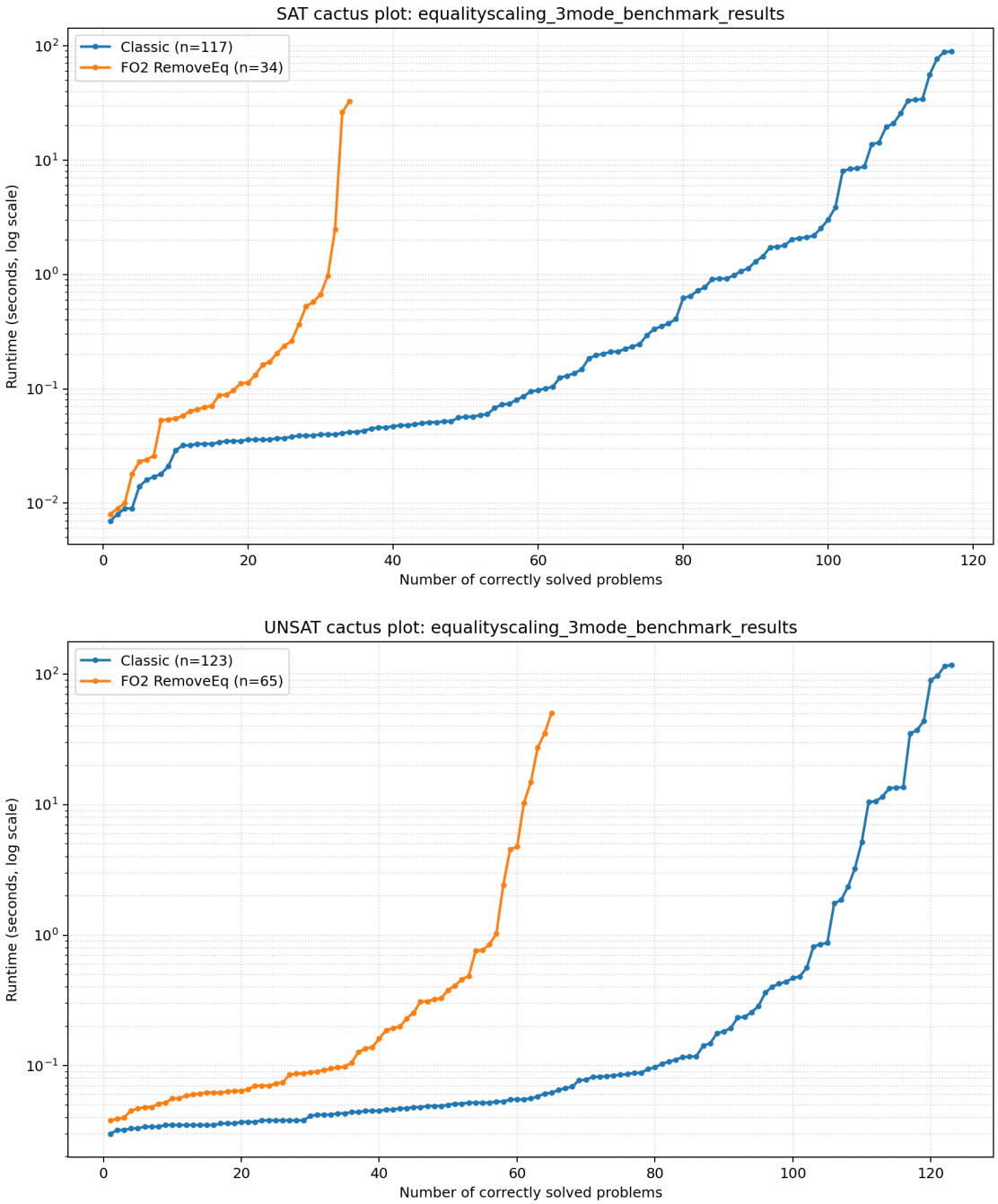


Figura 5.16: Cactus plot di Equality Scaling (296 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

FO2 non produce decisioni in questa esecuzione. RemoveEq recupera parte delle istanze, ma rimane lontano dalla copertura di Classic.

Quantifier Scaling

296 problemi ai livelli q2, q4, q6 e q8. Si modifica il budget nominale dei quantificatori.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	6	17	231	42	23/296 (7.8%)
Classic	125	144	27	0	269/296 (90.9%)
RemoveEq	20	36	240	0	56/296 (18.9%)

Tabella 5.11: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti risolti del file sono FO2: 23, Classic: 269, FO2_RemoveEq: 56.

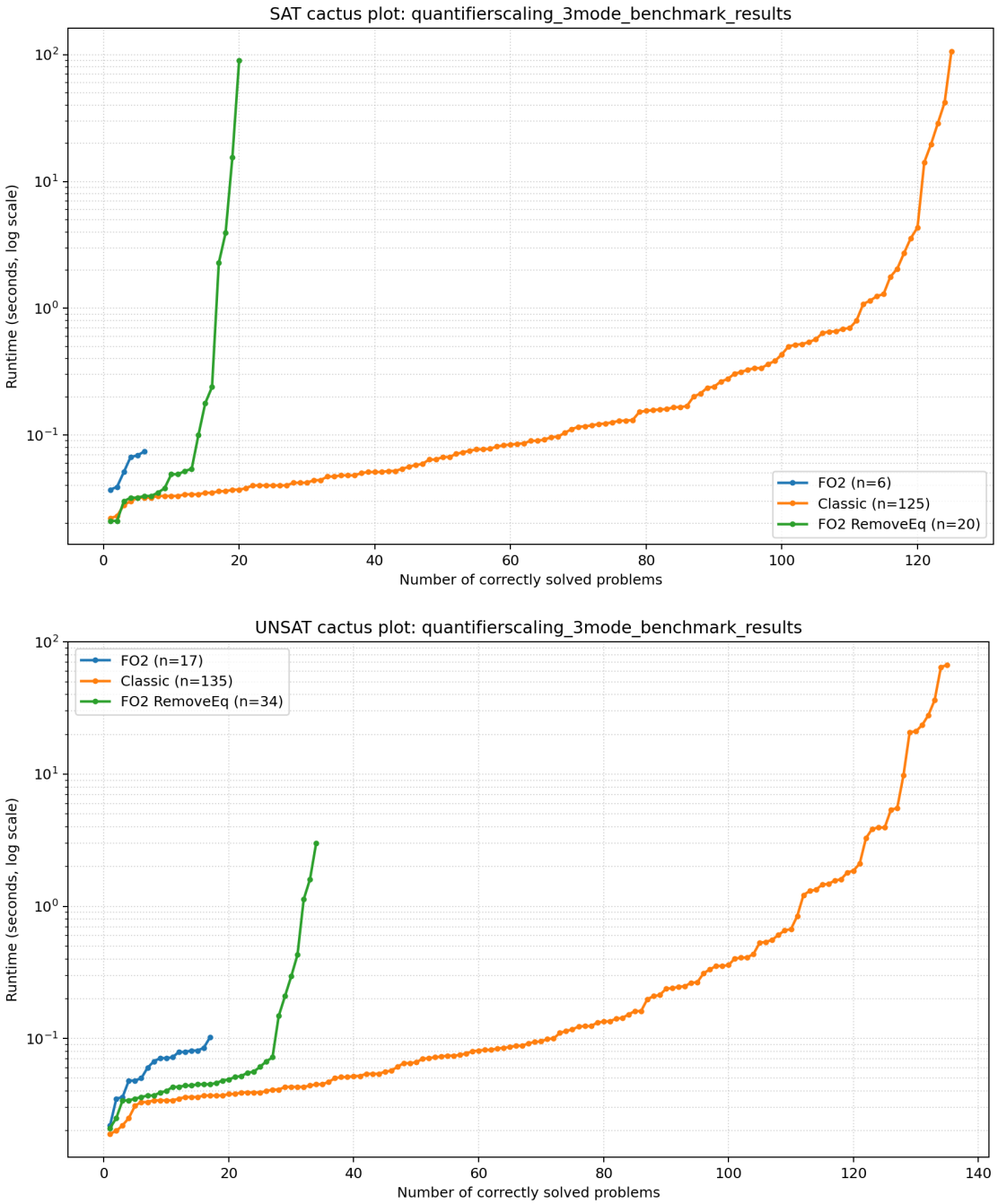


Figura 5.17: Cactus plot di Quantifier Scaling (296 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

Prevalgono i timeout di FO2 e RemoveEq.

Vocabulary Scaling

296 problemi, suddivisi nei profili nominali di 5 predicati unari e 2 binari, 10 e 5, 15 e 8, 20 e 10. Si varia quindi la dimensione del vocabolario relazionale.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	73	93	19	111	166/296 (56.1%)
Classic	137	149	10	0	286/296 (96.6%)
RemoveEq	120	137	39	0	257/296 (86.8%)

Tabella 5.12: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti superati sono FO2: 166, Classic: 286, FO2_RemoveEq: 257.

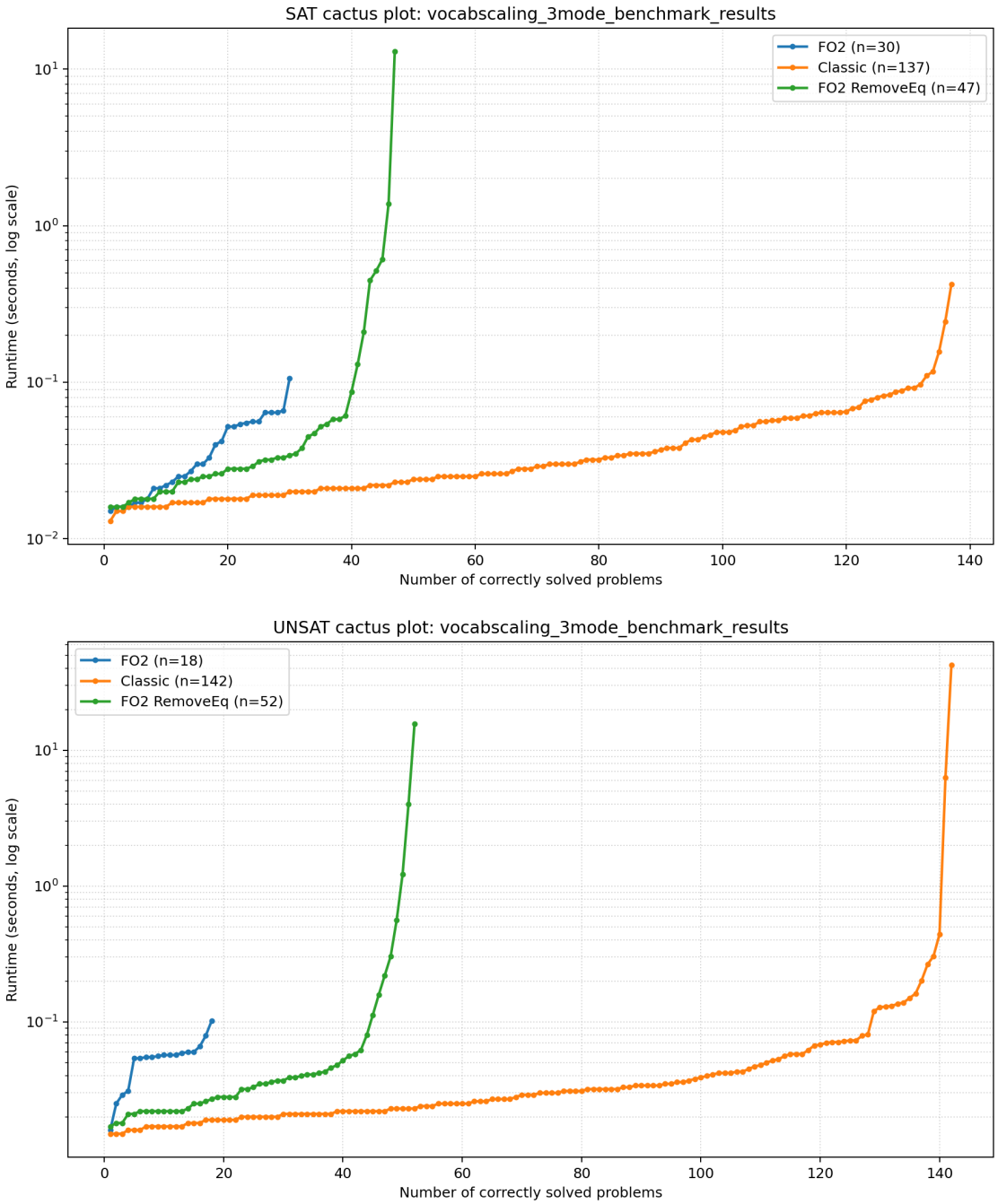


Figura 5.18: Cactus plot di Vocabulary Scaling (296 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

RemoveEq ha una copertura maggiore di FO2 e minore di Classic. Una mediana inferiore su campioni diversi non dimostra un'accelerazione sulle stesse formule.

SATBUILD Scaling

111 problemi nominalmente soddisfacibili, ai livelli dom1, dom2 e dom3. Il parametro riguarda il dominio usato nella costruzione dei problemi SAT.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	8	0	35	68	8/111 (7.2%)
Classic	111	0	0	0	111/111 (100.0%)
RemoveEq	71	0	40	0	71/111 (64.0%)

Tabella 5.13: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti coerenti con il nome del file sono FO2: 8, Classic: 111, FO2_RemoveEq: 71.

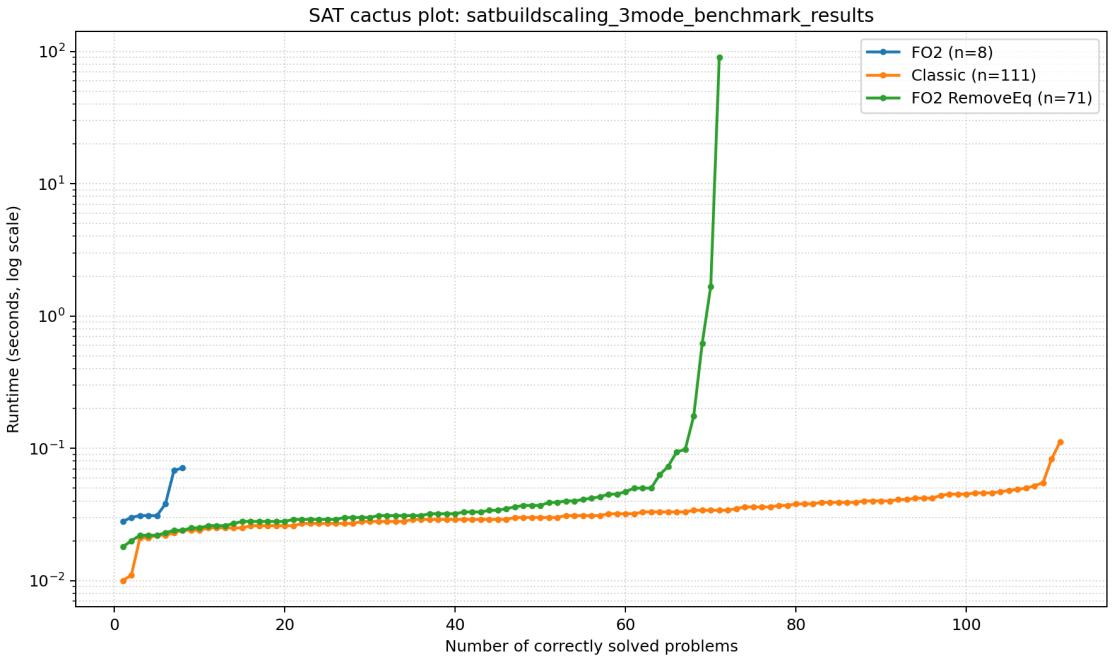


Figura 5.19: Cactus plot di SATBUILD Scaling (111 problemi): sola classe SAT. Sono incluse le decisioni coerenti con la classe nominale.

Classic decide tutte le istanze. FO2 e RemoveEq lasciano una quota consistente non decisa: l'assenza di casi UNSAT non rende uniforme il comportamento delle configurazioni.

Unknown candidates

25 casi selezionati da altre famiglie per approfondire gli esiti indeterminati.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	13	0	0	12	13/25 (52.0%)
Classic	25	0	0	0	25/25 (100.0%)
RemoveEq	23	0	2	0	23/25 (92.0%)

Tabella 5.14: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti coerenti con il nome del file sono FO2: 13, Classic: 25, FO2_RemoveEq: 23.

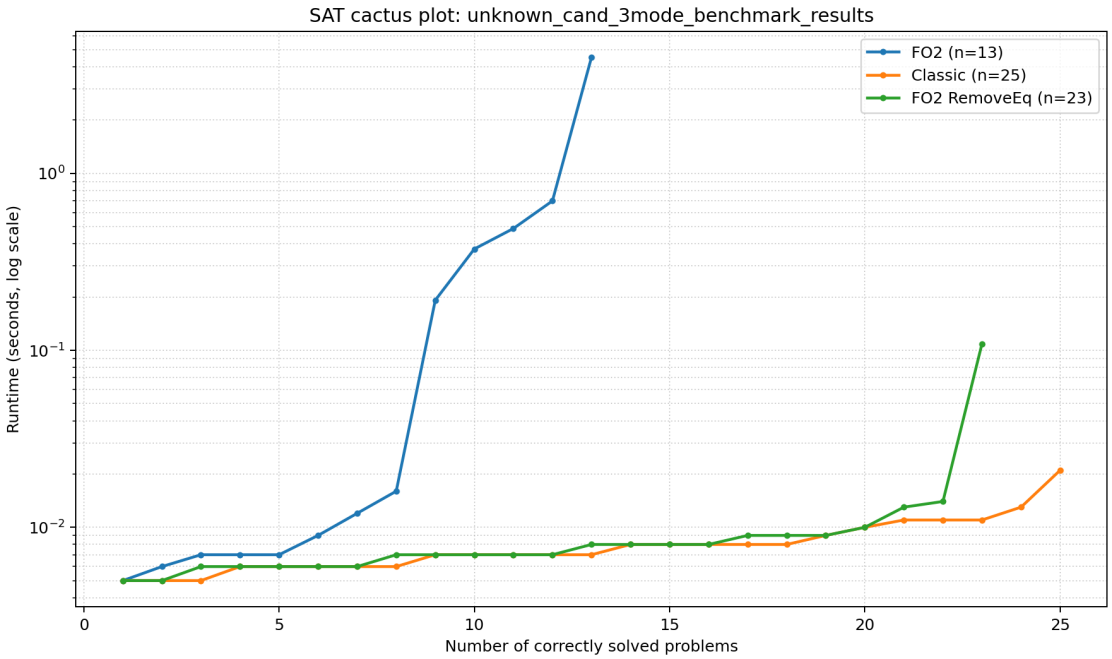


Figura 5.20: Cactus plot di Unknown candidates (25 problemi): sola classe SAT. Sono incluse le decisioni coerenti con la classe nominale.

Essendo un sottoinsieme selezionato, non è un campione indipendente. Il nome della famiglia non implica che ogni istanza restituisca Unknown nella versione misurata.

Timeout candidates

67 casi selezionati da altre famiglie per approfondire gli arresti per tempo.

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	0	7	52	8	7/67 (10.4%)
Classic	0	27	40	0	27/67 (40.3%)
RemoveEq	0	14	53	0	14/67 (20.9%)

Tabella 5.15: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Per il cactus separato per classe, i verdetti coerenti con il nome del file sono FO2: 7, Classic: 27, FO2_RemoveEq: 14.

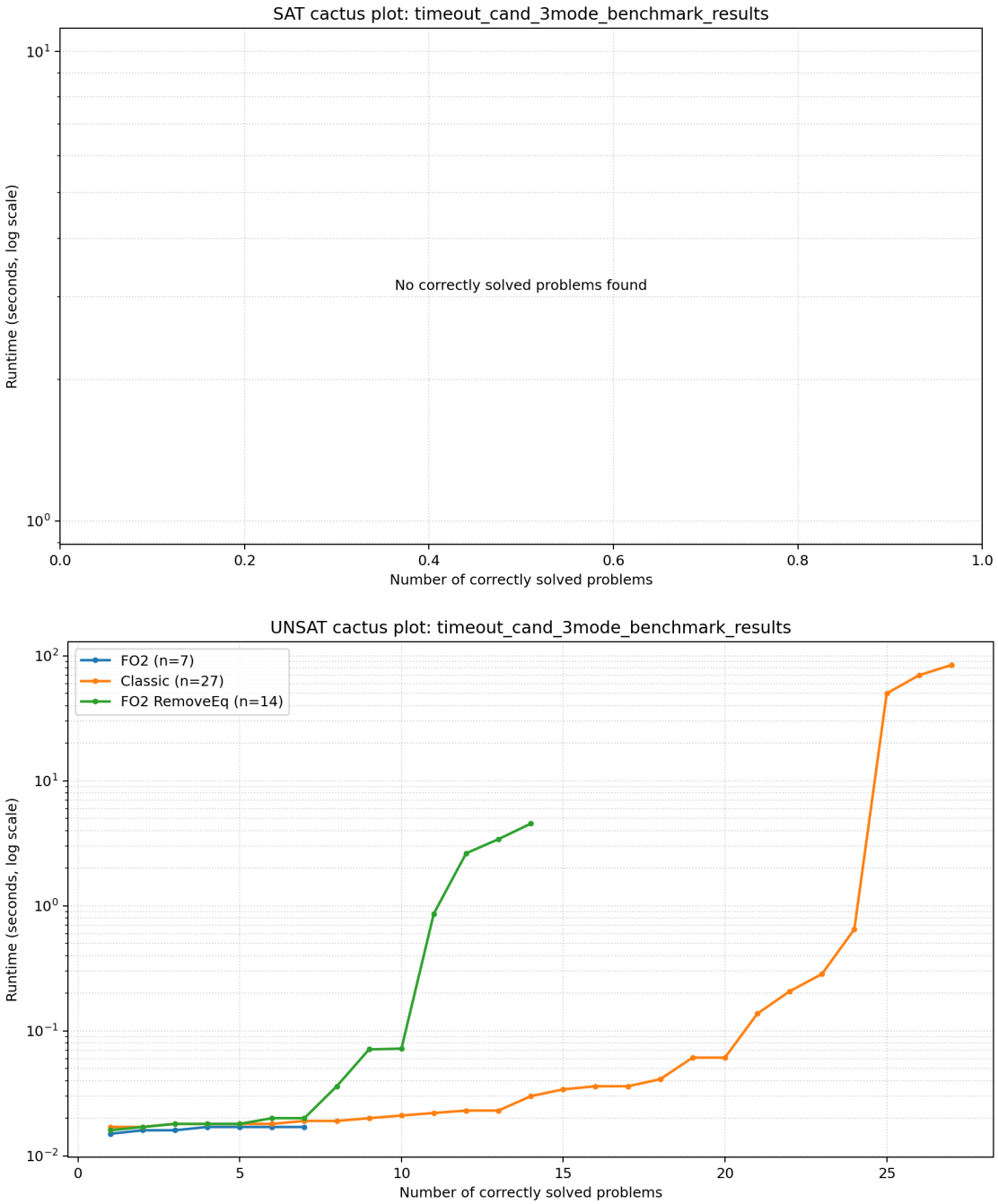


Figura 5.21: Cactus plot di Timeout candidates (67 problemi): SAT in alto e UNSAT in basso. Sono incluse le decisioni coerenti con la classe nominale.

5.1.4 Benchmark generati senza uguaglianza

La suite senza uguaglianza comprende cinque famiglie di 96 problemi, per 480 istanze complessive. Ogni famiglia ha quattro livelli nominali di 24 problemi, ciascuno con 12 file SAT e 12 file UNSAT. Si confrontano FO2 e Classic Otter; non si aggiunge RemoveEq, poiché manca l'operatore da eliminare.

Le figure aggregano SAT e UNSAT e contano tutte le decisioni registrate.

Alternation Scaling senza uguaglianza

I quattro profili nominali sono fa2_ex1, fa2_ex3, fa3_ex2 e fa4_ex4: variano i budget universali ed esistenziali. Il nome non certifica il numero esatto di alternanze lungo ogni ramo; questa misura richiederebbe l'analisi delle formule. La quantificazione può introdurre dipendenze dei testimoni e aumentare il lavoro di clausificazione e saturazione.

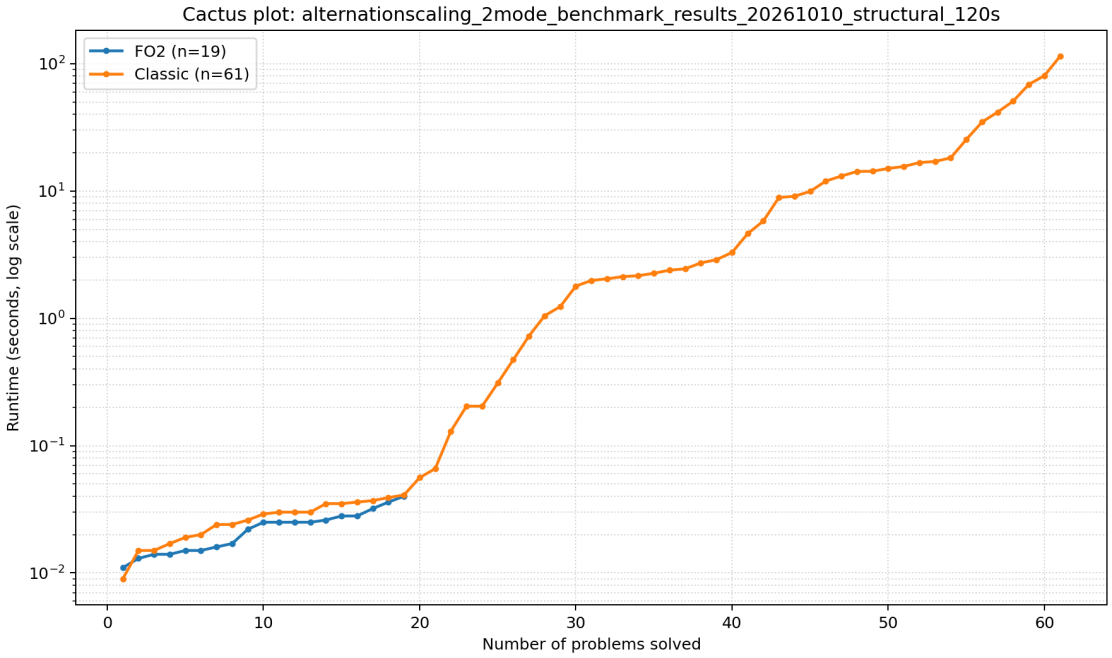


Figura 5.22: Cactus plot di Alternation Scaling senza uguaglianza (96 istanze, SAT e UNSAT aggregati; limite di 120 secondi per modalità).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	9	10	77	0	19/96 (19.8%)
Classic	37	24	35	0	61/96 (63.5%)

Tabella 5.16: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Le 19 istanze decise da FO2 sono tutte decise anche da Classic; si registrano 0 disaccordi tra i due verdetti. Le mediane su questo campione comune sono 0.025 s per FO2 e 0.029 s per Classic.

Le decisioni per livello, nell'ordine FO2/Classic e su 24 istanze ciascuno, sono: fa2_ex1: 6/15; fa2_ex3: 3/16; fa3_ex2: 7/18; fa4_ex4: 3/12.

Il preprocessing strutturale separa le definizioni quantificate, ma il fatto che non compaiano Unknown non implica che la saturazione sia rapida. I timeout restano il principale limite osservato.

Branching Scaling senza uguaglianza

I livelli branch4, branch8, branch12 e branch16 modificano il budget nominale di ramificazione/connettivi. Più ramificazione può produrre più clausole o alternative di ricerca; equivalenze e semplificazioni possono però ridurre il problema, quindi non si presume una crescita monotona della difficoltà.

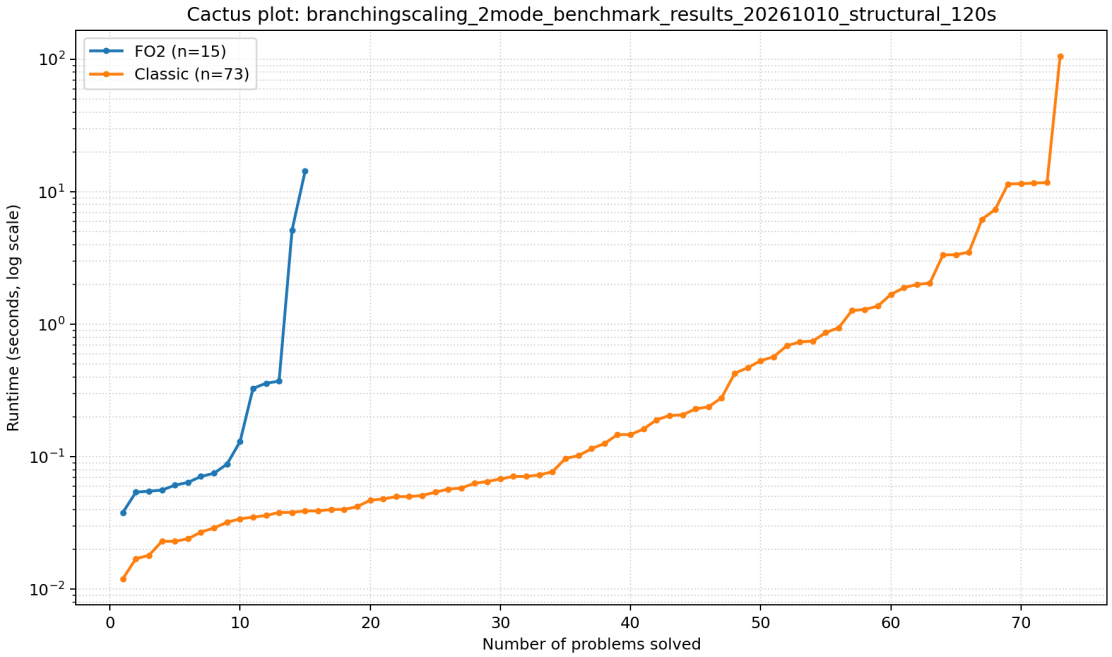


Figura 5.23: Cactus plot di Branching Scaling senza uguaglianza (96 istanze, SAT e UNSAT aggregati; limite di 120 secondi per modalità).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	7	8	81	0	15/96 (15.6%)
Classic	39	34	23	0	73/96 (76.0%)

Tabella 5.17: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Le 15 istanze decise da FO2 sono tutte decise anche da Classic; si registrano 0 disaccordi tra i due verdetti. Le mediane su questo campione comune sono 0.075 s per FO2 e 0.035 s per Classic.

Le decisioni per livello, nell'ordine FO2/Classic e su 24 istanze ciascuno, sono: branch12: 2/20; branch16: 1/18; branch4: 5/20; branch8: 7/15.

La copertura di FO2 è bassa; i conteggi per livello non sono monotoni. Il risultato è compatibile con una crescita della ricerca, ma non dimostra una relazione causale con il solo numero di connettivi.

Depth Scaling senza uguaglianza

I livelli d6, d10, d14 e d18 aumentano la profondità sintattica nominale. Una maggiore profondità consente più annidamento di connettivi e quantificatori, senza determinare da sola il numero finale di clausole.

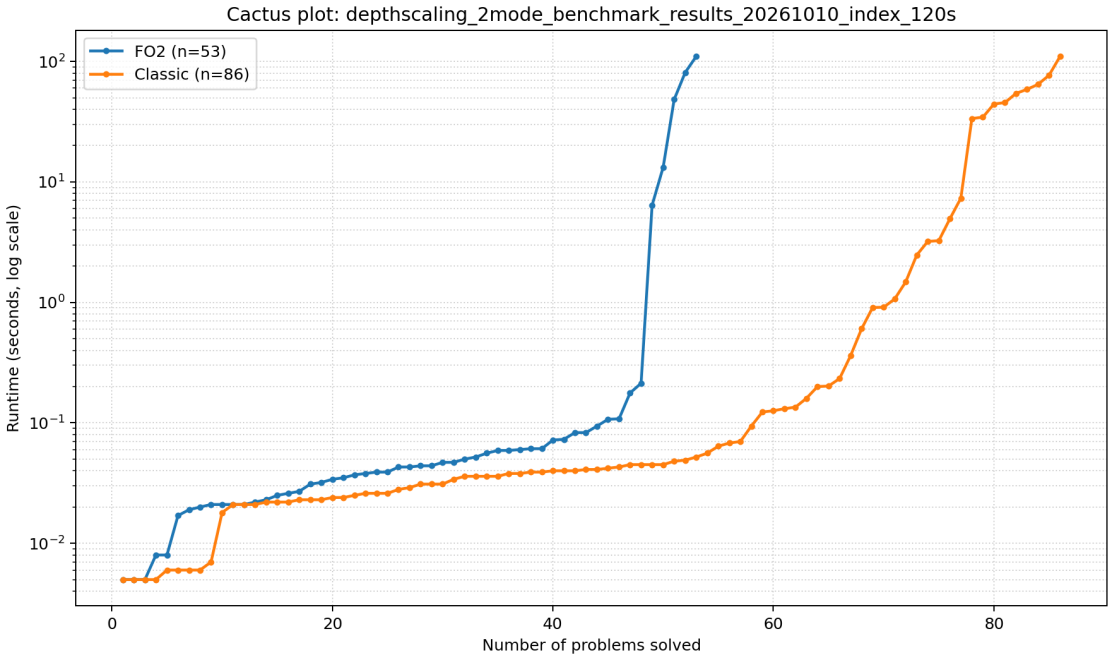


Figura 5.24: Cactus plot di Depth Scaling senza uguaglianza (96 istanze, SAT e UNSAT aggregati; limite di 120 secondi per modalità).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	25	28	43	0	53/96 (55.2%)
Classic	43	43	10	0	86/96 (89.6%)

Tabella 5.18: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Le 53 istanze decise da FO2 sono tutte decise anche da Classic; si registrano 0 disaccordi tra i due verdetti. Le mediane su questo campione comune sono 0.043 s per FO2 e 0.034 s per Classic.

Le decisioni per livello, nell'ordine FO2/Classic e su 24 istanze ciascuno, sono: d10: 16/23; d14: 6/18; d18: 7/21; d6: 24/24.

FO2 decide tutti i casi a d6, poi la copertura cala nettamente. Il lieve recupero a d18 rispetto a d14 impedisce di descrivere l'andamento come strettamente monotono. Classic mantiene una copertura maggiore a tutti i livelli più profondi.

I verdetti discordanti con la classe indicata dal nome sono 2/2 (FO2/Classic). La concordanza tra motori non risolve l'eventuale incoerenza della classe nominale.

Quantifier Scaling senza uguaglianza

I livelli q2, q4, q6 e q8 variano il budget nominale dei quantificatori. Quantificazione e dipendenze introdotte dalla skolemizzazione possono ampliare lo spazio delle inferenze anche senza uguaglianza.

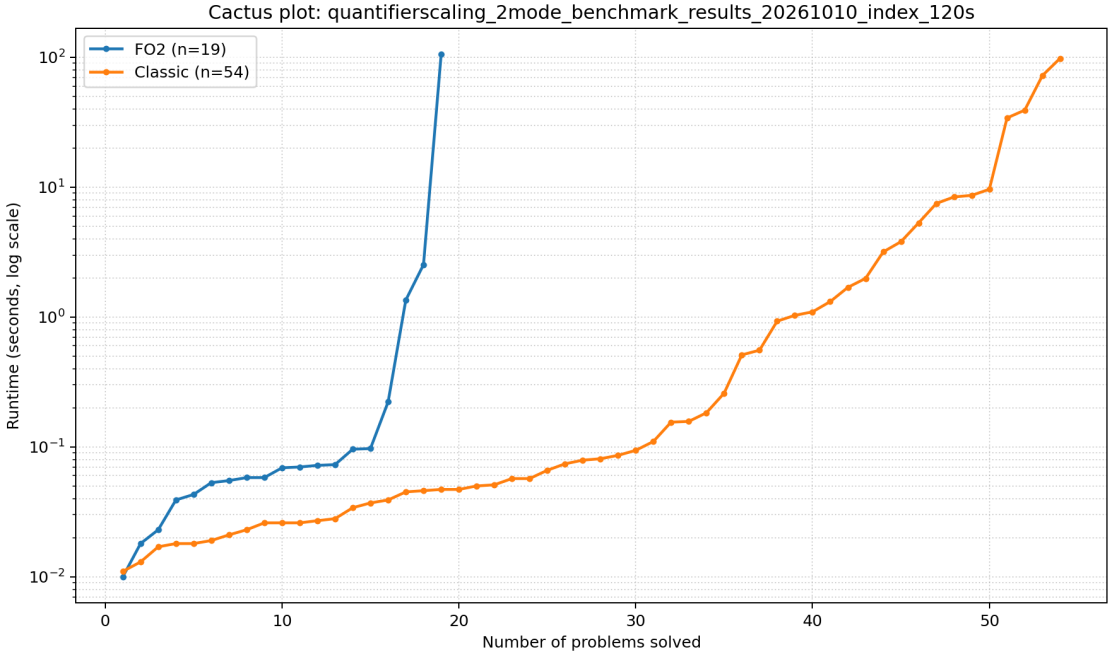


Figura 5.25: Cactus plot di Quantifier Scaling senza uguaglianza (96 istanze, SAT e UNSAT aggregati; limite di 120 secondi per modalità).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	8	11	77	0	19/96 (19.8%)
Classic	29	25	42	0	54/96 (56.2%)

Tabella 5.19: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Le 19 istanze decise da FO2 sono tutte decise anche da Classic; si registrano 0 disaccordi tra i due verdetti. Le mediane su questo campione comune sono 0.069 s per FO2 e 0.026 s per Classic.

Le decisioni per livello, nell'ordine FO2/Classic e su 24 istanze ciascuno, sono: q2: 10/24; q4: 5/18; q6: 3/7; q8: 1/5.

La copertura cala a ogni livello per entrambi i motori. È il segnale più netto di difficoltà crescente nella suite, ma non isola il costo dei quantificatori da eventuali variazioni della struttura generata.

I verdetti discordanti con la classe indicata dal nome sono 0/0 (FO2/Classic). La concordanza tra motori non risolve l'eventuale incoerenza della classe nominale.

Vocabulary Scaling senza uguaglianza

I profili nominali sono 6 predicati unari e 3 binari, 12 e 6, 18 e 10, 24 e 14. Un vocabolario più ampio modifica i possibili accoppiamenti tra letterali; può aumentare la varietà delle clausole ma anche ridurre le coppie complementari, perciò non implica automaticamente maggiore difficoltà.

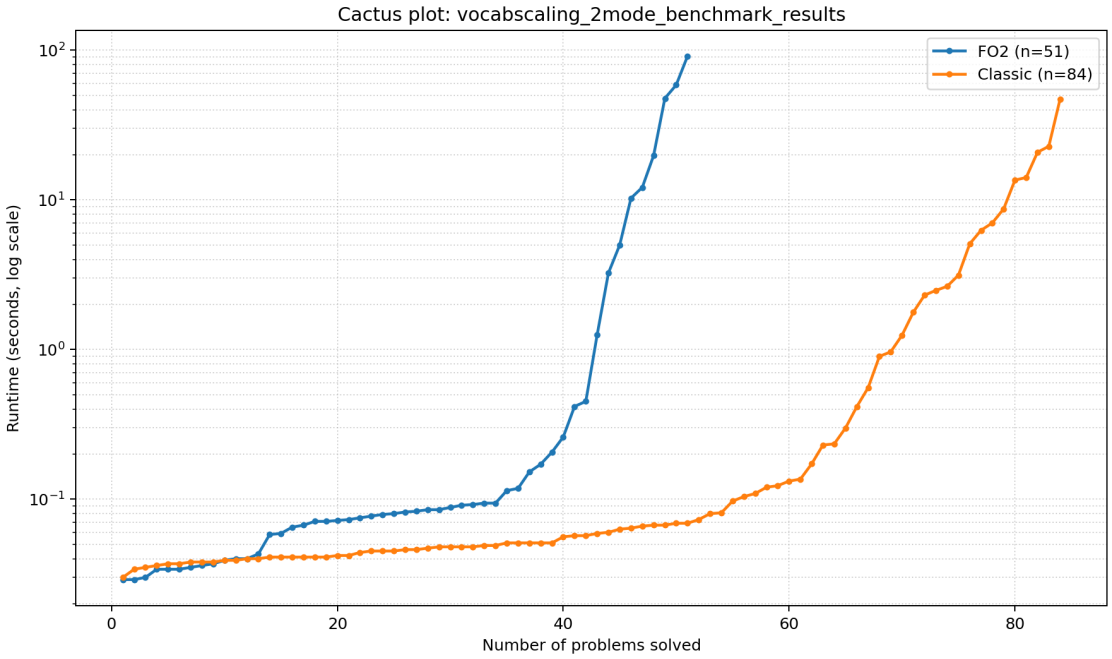


Figura 5.26: Cactus plot di Vocabulary Scaling senza uguaglianza (96 istanze, SAT e UNSAT aggregati; limite di 120 secondi per modalità).

Configurazione	SAT	UNSAT	Timeout	Altri	Decisi
FO2	26	25	45	0	51/96 (53.1%)
Classic	43	41	12	0	84/96 (87.5%)

Tabella 5.20: Esiti registrati nel CSV; *Altri* comprende Unknown ed Error.

Le 51 istanze decise da FO2 sono tutte decise anche da Classic; si registrano 0 disaccordi tra i due verdetti. Le mediane su questo campione comune sono 0.082 s per FO2 e 0.046 s per Classic.

Le decisioni per livello, nell'ordine FO2/Classic e su 24 istanze ciascuno, sono: p12_16_2: 10/23; p18_110_2: 10/17; p24_114_2: 18/24; p6_13_2: 13/20.

FO2 decide più della metà dei problemi, ma rimane sotto Classic. Il CSV appartiene alla campagna base: le differenze rispetto alle altre famiglie non devono essere attribuite soltanto al vocabolario, poiché manca un controllo uniforme della versione del solutore.

I verdetti discordanti con la classe indicata dal nome sono 1/1 (FO2/Classic). La concordanza tra motori non risolve l'eventuale incoerenza della classe nominale.

5.2 Metodologia di confronto e controllo delle risorse

Per garantire la rigorosità scientifica dei risultati nella tesi, l'esecuzione di OtterVampire non viene lasciata alle impostazioni predefinite da competizione CASC, ma viene uniformata mediante uno specifico protocollo di **Fair Comparison**.

5.2.1 Parametri di Confronto Simmetrico

Le euristiche di default di Vampire introducono asimmetrie fondamentali rispetto alla procedura decisionale al primo ordine puro. Per ripristinare la parità di condizioni, OtterVampire viene eseguito con i seguenti flag:

1. **Algoritmo di Saturazione Otter (`--sa otter`)**: La strategia predefinita LRS (*Limited Resource Strategy*) scarta clausole passive in base al tempo CPU stimato residuo, introducendo non-determinismo legato al carico della macchina. L'algoritmo Otter mantiene l'invariante di completezza deterministica su entrambi i motori.

2. **Disattivazione di AVATAR (`-av off`)**: Disabilita lo splitting proposizionale gestito da un solutore SAT/SMT esterno, consentendo il confronto puro tra motori al primo ordine clausale.

3. **Disattivazione Euristiche Asimmetriche di Preprocessig (`-updr off`)**: Disabilita la *Unused Predicate Definition Removal* che andrebbe ad alterare la struttura del problema in modo disomogeneo rispetto al pre-processing del solutore generico.

4. **Subsumption simmetrica (`-bs on -fsr off`)**: Il solutore FO2 implementa esplicitamente *forward subsumption* e *backward subsumption*, ma non la *subsumption resolution*. In OtterVampire la forward subsumption è attiva di default (`-fs on`) e viene mantenuta, mentre di default la backward subsumption è disattivata (`-bs off`) e la forward subsumption resolution è attiva (`-fsr on`). Si attiva quindi la backward subsumption (`-bs on`) e si disattiva la forward subsumption resolution (`-fsr off`), così che entrambi i motori applichino lo stesso insieme di tecniche di sussunzione.

5.2.2 Limiti della simmetria

Il confronto resta simmetrico solo rispetto alle opzioni sopra elencate. OtterVampire applica altre semplificazioni attive di default (ad esempio la demodulazione) e usa indicizzazione e selezione dei letterali proprie, mentre il solutore FO2 usa la selezione Σ_2

5.2.3 Limiti temporali e memoria

I benchmark qui selezionati adottano un limite di 120 secondi per problema e modalità. Gli script prevedono un limite di memoria di 4096 MB per processo; il CSV

da solo non certifica il valore effettivo di ogni invocazione. Per la suite senza uguaglianza i tempi sono misure di tempo reale e gli script possono eseguire più worker in parallelo: carico e scheduling influenzano soprattutto i confronti su tempi molto brevi. Piccoli superamenti del limite registrati nel CSV possono dipendere dal controllo del processo e non vanno interpretati come decisioni oltre il budget.

5.2.4 Interpretazione e riproducibilità

I commenti prima di ciascuna famiglia indicano il CSV usato per rigenerare le figure. Non si sostituiscono risultati di esecuzioni storiche con quelli ottenuti da versioni successive. I CSV conservano conteggi ed esiti, ma non bastano a identificare sempre revisione del binario, hardware, seed e configurazione del generatore: una riproduzione completa richiede anche queste informazioni.

Le spiegazioni strutturali proposte sono ipotesi compatibili con i dati. Per attribuire il costo a clausificazione, eliminazione dell'uguaglianza, crescita delle clausole o indicizzazione occorrono misure separate di tali fasi. La suite senza uguaglianza mostra che la sola assenza di uguaglianza non elimina i limiti pratici della ricerca, soprattutto nelle famiglie di quantificazione e ramificazione.

— 6 —

Conclusioni

Il presente lavoro di tesi ha riguardato l'implementazione, all'interno del dimostratore automatico Vampire, di una procedura di decisione specializzata per il frammento della logica del primo ordine a due variabili con uguaglianza ($FO^2_=$), basata sull'approccio proposto da de Nivelle e Pratt-Hartmann. Il lavoro si è concentrato sulla realizzazione delle trasformazioni necessarie per l'eliminazione dell'uguaglianza e sull'integrazione di un motore di saturazione fondato sulla risoluzione indicizzata, con l'obiettivo di valutare l'applicabilità pratica della procedura teorica.

Uno degli aspetti più significativi emersi durante lo sviluppo e la valutazione della procedura riguarda il costo computazionale associato all'eliminazione dell'uguaglianza. Sebbene questa trasformazione consenta di ricondurre il problema a un frammento per il quale è disponibile un calcolo di risoluzione terminante, essa non comporta necessariamente una riduzione della complessità del problema dal punto di vista operativo. L'applicazione dei lemmi richiede infatti l'introduzione di nuovi predicati, formule ausiliarie e assiomi di congruenza, che possono aumentare considerevolmente la dimensione della rappresentazione iniziale. Inoltre, le fasi di clausificazione e risoluzione ristretta possono produrre ulteriori clausole, ampliando l'insieme su cui deve operare il motore di saturazione. Questo aspetto rappresenta un possibile fattore esplicativo delle difficoltà osservate nei benchmark con uguaglianza, soprattutto nelle formule caratterizzate da un numero elevato di occorrenze del simbolo di identità. I risultati sperimentali evidenziano infatti che la trasformazione preliminare non si traduce automaticamente in un miglioramento delle prestazioni rispetto alla risoluzione standard di Vampire.

Un'ulteriore considerazione emerge dall'analisi dei problemi appartenenti al frammento privo di uguaglianza. In questo caso non è necessario applicare le trasformazioni per l'eliminazione del simbolo di identità, permettendo di valutare più direttamente il comportamento del motore di risoluzione specializzato. Nei benchmark TPTP (senza uguaglianza) in formato CNF, la procedura FO2 riesce a decidere un grande numero di problemi, quasi alla pari di Vampire, mentre nel formato FOF il motore standard ottiene una copertura maggiore. Questo risultato suggerisce che, almeno su alcune istanze già in forma clausale, il calcolo specializzato riesce a raggiungere risultati confrontabili con quelli di Vampire, pur senza offrire un vantaggio significativo in termini di prestazioni. Le differenze diventano invece più marcate nei benchmark sintetici, soprattutto nelle famiglie caratterizzate da un aumento della ramificazione e della complessità della quantificazione. Tali risultati mostrano che le difficoltà dell'implementazione non dipendono esclusivamente dal costo della rimozione dell'uguaglianza, ma riguardano anche la gestione del processo di saturazione. In particolare, la crescita delle clausole generate, il numero di inferenze e l'efficienza delle strategie di selezione e semplificazione rappresentano possibili fattori che

contribuiscono ai tempi di esecuzione.

Risulta quindi evidente che la specializzazione del calcolo a un frammento decidibile non garantisce, da sola, prestazioni superiori rispetto a un dimostratore generale come Vampire, che dispone di strategie e ottimizzazioni sviluppate per un insieme molto più ampio di problemi.

Considerando congiuntamente i risultati ottenuti sui problemi con e senza uguaglianza, emergono pertanto due aspetti distinti su cui intervenire: da un lato, il costo delle trasformazioni necessarie per eliminare l'uguaglianza; dall'altro, l'efficienza del motore di risoluzione indicizzata, che presenta margini di miglioramento anche quando opera direttamente su formule prive di uguaglianza.

— A —

Risultati sperimentali TPTP

In questa appendice sono riportati i tempi di esecuzione registrati sui problemi TPTP utilizzati nella valutazione sperimentale. I risultati sono suddivisi in quattro categorie: FOF e CNF, con e senza uguaglianza.

I tempi sono espressi in secondi. Il tempo registrato non implica necessariamente che il problema sia stato risolto: gli esiti SAT, UNSAT, UNKNOWN e timeout sono analizzati nel Capitolo 5.

Tabella A.1: Risultati TPTP FOF senza uguaglianza (343 problemi)

Problema	**FO2**	**Vampire Classic**
KRS018+1	0.025	0.012
KRS019+1	0.012	0.011
KRS020+1	0.014	0.012
KRS021+1	0.011	0.011
KRS022+1	0.011	0.016
KRS023+1	0.013	0.011
KRS024+1	0.013	0.013
KRS025+1	0.014	0.012
KRS027+1	0.011	0.010
KRS041+1	110.779	0.098
KRS050+1	0.073	0.037
KRS053+1	0.093	0.217
KRS054+1	0.181	0.071
KRS055+1	0.037	0.032
KRS056+1	0.034	0.034
KRS058+1	0.034	0.037
KRS059+1	0.032	0.032
KRS060+1	0.080	0.032
KRS061+1	0.082	0.031
KRS062+1	0.035	0.034
KRS064+1	0.032	0.032
KRS065+1	0.100	0.143
KRS066+1	0.070	0.032
KRS067+1	0.094	0.048
KRS068+1	0.101	0.074
KRS090+1	0.163	0.035
KRS091+1	0.070	0.034
KRS092+1	0.071	0.034
KRS093+1	0.071	0.046
KRS094+1	0.090	0.359
KRS100+1	0.567	0.065
KRS103+1	120.111	0.055
KRS104+1	8.641	0.048
KRS105+1	0.078	0.041
KRS123+1	0.078	0.127
KRS124+1	0.156	0.083
KRS125+1	0.043	0.246
KRS126+1	0.087	0.036
KRS128+1	0.081	0.038
KRS130+1	0.279	0.043
KRS131+1	0.723	0.052
KRS132+1	17.694	0.042

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
KRS134+1	0.190	0.039
KRS135+1	0.313	0.035
KRS136+1	0.059	0.362
KRS137+1	7.128	0.120
KRS139+1	2.134	0.088
KRS141+1	1.485	0.038
KRS146+1	121.102	0.062
KRS147+1	120.376	2.144
KRS148+1	120.049	0.082
KRS149+1	120.075	0.063
KRS150+1	120.062	0.049
KRS151+1	120.052	120.056
KRS152+1	13.756	0.040
KRS153+1	120.065	3.072
KRS154+1	120.499	0.162
KRS155+1	122.500	0.116
KRS156+1	120.113	0.113
KRS157+1	120.277	0.092
KRS158+1	121.435	0.300
KRS159+1	120.045	0.434
KRS160+1	7.611	0.031
KRS161+1	120.160	1.904
KRS164+1	4.270	0.027
KRS165+1	3.813	0.030
KRS166+1	0.301	0.026
KRS168+1	0.182	0.110
KRS169+1	0.401	0.027
KRS170+1	1.611	0.031
KRS171+1	0.508	0.034
KRS173+1	0.973	0.034
LCL636+1.001	0.931	0.071
LCL636+1.005	7.346	122.956
LCL636+1.010	150.779	124.089
LCL636+1.015	120.139	120.562
LCL636+1.020	120.086	120.593
LCL637+1.001	0.065	0.008
LCL637+1.005	4.633	120.301
LCL637+1.010	120.133	120.580
LCL637+1.015	120.144	120.576
LCL637+1.020	120.095	120.412
LCL638+1.001	0.027	0.034
LCL638+1.005	0.040	120.093

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
LCL638+1.010	0.072	120.118
LCL638+1.015	0.066	120.134
LCL638+1.020	0.116	120.107
LCL639+1.001	0.035	95.969
LCL639+1.005	0.067	120.093
LCL639+1.010	0.068	120.104
LCL639+1.015	0.117	120.094
LCL639+1.020	0.118	120.089
LCL640+1.001	0.034	120.078
LCL640+1.005	0.036	120.094
LCL640+1.010	0.037	120.111
LCL640+1.015	0.042	120.161
LCL640+1.020	0.037	120.137
LCL641+1.001	0.023	120.079
LCL641+1.005	0.024	120.134
LCL641+1.010	0.066	120.097
LCL641+1.015	0.067	120.080
LCL641+1.020	0.072	120.086
LCL642+1.001	120.430	120.672
LCL642+1.005	120.823	120.256
LCL642+1.010	120.510	120.104
LCL642+1.015	120.993	120.086
LCL642+1.020	120.537	120.033
LCL643+1.001	120.288	120.197
LCL643+1.005	120.339	120.148
LCL643+1.010	120.389	120.046
LCL643+1.015	120.162	120.024
LCL643+1.020	120.236	120.016
LCL644+1.001	69.301	0.065
LCL644+1.005	17.291	0.063
LCL644+1.010	22.761	0.065
LCL644+1.015	22.026	0.065
LCL644+1.020	28.339	0.167
LCL645+1.001	0.017	0.017
LCL645+1.005	27.094	120.175
LCL645+1.010	49.123	120.101
LCL645+1.015	70.188	120.163
LCL645+1.020	109.896	120.122
LCL646+1.001	0.065	0.017
LCL646+1.005	0.033	120.329
LCL646+1.010	0.065	120.320
LCL646+1.015	0.115	120.266

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
LCL646+1.020	0.166	120.234
LCL647+1.001	0.116	0.065
LCL647+1.005	0.033	120.567
LCL647+1.010	0.104	120.304
LCL647+1.015	0.115	120.254
LCL647+1.020	0.266	120.284
LCL648+1.001	0.021	0.036
LCL648+1.005	120.008	120.069
LCL648+1.010	120.015	120.231
LCL648+1.015	120.105	120.567
LCL648+1.020	120.140	120.563
LCL649+1.001	0.065	0.017
LCL649+1.005	120.006	120.070
LCL649+1.010	120.024	120.322
LCL649+1.015	120.108	120.304
LCL649+1.020	120.235	120.528
LCL650+1.001	0.068	0.116
LCL650+1.005	0.033	120.012
LCL650+1.010	0.166	120.010
LCL650+1.015	0.166	120.012
LCL650+1.020	0.318	120.013
LCL651+1.001	0.065	120.291
LCL651+1.005	0.033	120.008
LCL651+1.010	0.065	120.013
LCL651+1.015	0.204	120.013
LCL651+1.020	0.316	120.023
LCL652+1.001	0.065	120.107
LCL652+1.005	0.068	120.038
LCL652+1.010	0.128	120.024
LCL652+1.015	0.122	120.039
LCL652+1.020	0.117	120.038
LCL653+1.001	0.066	120.095
LCL653+1.005	0.033	120.035
LCL653+1.010	0.104	120.045
LCL653+1.015	0.116	120.072
LCL653+1.020	0.166	120.089
LCL654+1.001	0.068	120.057
LCL654+1.005	0.065	120.050
LCL654+1.010	0.116	120.059
LCL654+1.015	0.116	120.068
LCL654+1.020	0.168	120.076
LCL655+1.001	0.065	120.043

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
LCL655+1.005	0.065	120.058
LCL655+1.010	0.116	120.048
LCL655+1.015	0.166	120.024
LCL655+1.020	0.216	120.031
LCL656+1.001	0.065	0.017
LCL656+1.005	0.033	0.216
LCL656+1.010	0.033	120.104
LCL656+1.015	0.117	120.290
LCL656+1.020	0.116	120.318
LCL657+1.001	0.174	0.072
LCL657+1.005	0.121	4.198
LCL657+1.010	0.116	120.153
LCL657+1.015	0.116	120.340
LCL657+1.020	0.115	120.297
LCL658+1.001	0.065	4.384
LCL658+1.005	0.065	120.167
LCL658+1.010	0.065	120.088
LCL658+1.015	0.066	120.077
LCL658+1.020	0.116	120.030
LCL659+1.001	0.034	120.069
LCL659+1.005	0.065	120.117
LCL659+1.010	0.065	120.101
LCL659+1.015	0.116	120.084
LCL659+1.020	0.116	120.049
LCL660+1.001	120.333	120.259
LCL660+1.005	120.374	120.192
LCL660+1.010	120.350	120.160
LCL660+1.015	120.445	120.148
LCL660+1.020	120.363	120.088
LCL661+1.001	120.210	120.155
LCL661+1.005	120.229	120.187
LCL661+1.010	120.284	120.196
LCL661+1.015	120.230	120.179
LCL661+1.020	120.262	120.021
LCL662+1.001	0.065	0.017
LCL662+1.005	0.066	120.118
LCL662+1.010	1.576	120.125
LCL662+1.015	9.650	120.277
LCL662+1.020	43.795	120.188
LCL663+1.001	0.065	0.033
LCL663+1.005	0.035	120.148
LCL663+1.010	0.115	120.176

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
LCL663+1.015	0.316	120.217
LCL663+1.020	1.724	120.219
LCL664+1.001	0.065	0.017
LCL664+1.005	0.033	120.377
LCL664+1.010	0.115	120.201
LCL664+1.015	0.166	120.195
LCL664+1.020	0.216	120.150
LCL665+1.001	0.116	0.117
LCL665+1.005	0.033	120.331
LCL665+1.010	0.115	120.160
LCL665+1.015	0.166	120.170
LCL665+1.020	0.254	120.141
LCL666+1.001	0.067	0.017
LCL666+1.005	120.004	120.042
LCL666+1.010	120.023	120.158
LCL666+1.015	120.083	120.101
LCL666+1.020	120.223	120.239
LCL667+1.001	0.065	0.018
LCL667+1.005	120.006	120.048
LCL667+1.010	120.021	120.193
LCL667+1.015	120.096	120.107
LCL667+1.020	120.153	120.224
LCL668+1.001	0.065	120.031
LCL668+1.005	0.116	120.011
LCL668+1.010	0.266	120.011
LCL668+1.015	0.416	120.010
LCL668+1.020	1.187	120.024
LCL669+1.001	0.065	120.119
LCL669+1.005	0.072	120.011
LCL669+1.010	0.172	120.020
LCL669+1.015	0.215	120.010
LCL669+1.020	0.266	120.012
LCL670+1.001	0.065	120.068
LCL670+1.005	0.071	120.054
LCL670+1.010	0.065	120.024
LCL670+1.015	0.116	120.021
LCL670+1.020	0.115	120.050
LCL671+1.001	0.066	120.058
LCL671+1.005	0.033	120.017
LCL671+1.010	0.117	120.038
LCL671+1.015	0.115	120.069
LCL671+1.020	0.115	120.133

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
LCL680+1.001	0.065	0.017
LCL681+1.001	0.023	0.018
MED011+1	120.081	26.634
MSC011+1	0.067	0.033
NLP263+1	120.121	68.268
PUZ128+1	0.017	0.017
SET043+1	0.018	0.019
SEV515+1	9.956	0.033
SYN048+1	0.065	0.017
SYN051+1	0.017	0.018
SYN052+1	0.017	0.024
SYN053+1	0.018	0.017
SYN054+1	0.034	0.017
SYN055+1	0.033	0.017
SYN056+1	0.065	0.017
SYN057+1	0.033	0.018
SYN058+1	0.034	0.017
SYN060+1	0.034	0.019
SYN061+1	0.037	0.036
SYN062+1	0.035	0.037
SYN063+1	0.068	0.034
SYN315+1	0.018	0.033
SYN316+1	0.034	0.033
SYN317+1	0.069	0.034
SYN318+1	0.018	0.035
SYN322+1	0.035	0.035
SYN323+1	0.034	0.040
SYN324+1	0.021	0.036
SYN325+1	0.019	0.033
SYN355+1	0.020	0.037
SYN357+1	0.018	0.033
SYN358+1	0.020	0.034
SYN367+1	0.020	0.035
SYN368+1	0.018	0.036
SYN372+1	0.034	0.034
SYN373+1	0.066	0.034
SYN374+1	0.668	0.033
SYN375+1	0.667	0.033
SYN376+1	0.074	0.077
SYN377+1	0.728	0.034
SYN383+1	0.018	0.034
SYN385+1	0.018	0.034

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
SYN396+1	0.018	0.035
SYN397+1	0.018	0.034
SYN398+1	0.034	0.034
SYN399+1	0.034	0.033
SYN401+1	0.017	0.034
SYN402+1	0.017	0.018
SYN403+1	0.018	0.034
SYN404+1	0.034	0.018
SYN408+1	0.034	0.017
SYN412+1	0.017	0.017
SYN723+1	37.194	2.337
SYN724+1	0.035	0.018
SYN725+1	0.018	0.019
SYN921+1	0.018	0.020
SYN922+1	0.066	0.017
SYN923+1	0.017	0.017
SYN924+1	0.033	0.017
SYN925+1	0.017	0.017
SYN926+1	0.017	0.019
SYN927+1	0.017	0.033
SYN928+1	0.017	0.033
SYN929+1	0.018	0.017
SYN930+1	0.017	0.017
SYN931+1	0.018	0.035
SYN934+1	0.033	0.018
SYN935+1	0.017	0.017
SYN936+1	0.017	0.033
SYN937+1	0.017	0.018
SYN942+1	0.035	0.034
SYN951+1	0.034	0.023
SYN952+1	0.038	0.038
SYN953+1	0.039	0.077
SYN955+1	0.041	0.070
SYN956+1	0.074	0.069
SYN957+1	0.037	0.042
SYN958+1	0.025	0.071
SYN959+1	0.038	0.035
SYN960+1	0.036	0.018
SYN962+1	0.018	0.034
SYN963+1	0.019	0.034
SYN964+1	0.017	0.034
SYN968+1	0.036	0.020

Risultati TPTP FOF senza uguaglianza (343 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
SYN971+1	0.034	0.033
SYN972+1	0.017	0.033
SYN973+1	0.017	0.018
SYN974+1	0.017	0.035
SYN975+1	0.020	0.033
SYN976+1	0.017	0.033
SYO607+1	0.169	120.006

Tabella A.2: Risultati TPTP CNF senza uguaglianza (51 problemi)

Problema	**FO2**	**Vampire Classic**
KRS004-1	0.013	0.013
LAT267-2	0.016	0.014
MGT041-2	0.019	0.015
PUZ001-1	0.058	0.015
PUZ001-3	0.154	0.014
PUZ002-1	0.022	0.014
PUZ003-1	0.039	0.014
PUZ028-3	121.683	0.110
PUZ028-4	122.329	0.092
PUZ029-1	9.170	0.045
PUZ030-1	124.490	0.155
SET043-5	0.226	0.022
SWV424-1.050	120.963	120.068
SWV424-1.100	121.709	120.067
SWV424-1.200	121.500	120.068
SWV424-1.300	121.132	120.075
SWV424-1.350	120.551	120.066
SWV424-1.400	120.483	120.052
SWV424-1.450	120.595	120.062
SWV424-1.500	120.938	120.064
SYN051-1	0.052	0.032
SYN052-1	0.033	0.032
SYN053-1	0.032	0.030
SYN054-1	0.041	0.034
SYN055-1	0.037	0.044
SYN056-1	0.074	0.033
SYN057-1	0.092	0.033
SYN058-1	0.040	0.033
SYN060-1	0.037	0.033
SYN061-1	0.033	0.040
SYN062-1	0.044	0.033

Risultati TPTP CNF senza uguaglianza (51 problemi) (continua)

Problema	**FO2**	**Vampire Classic**
SYN063-1	0.036	0.033
SYN063-2	0.032	0.033
SYN095-1.002	0.035	0.033
SYN096-1.008	123.594	0.122
SYN099-1.003	1.263	0.032
SYN100-1.005	122.083	0.111
SYN101-1.002.002	0.260	0.045
SYN317-1	0.086	0.044
SYN318-1	0.079	0.046
SYN321-1	0.093	0.042
SYN322-1	0.045	0.041
SYN323-1	0.121	0.056
SYN490-1	122.097	0.133
SYN492-1	122.109	0.121
SYN495-1	121.935	0.093
SYN516-1	123.410	0.142
SYN521-1	121.842	0.093
SYN523-1	3.027	0.040
SYN527-1	124.481	0.160
SYN724-1	0.238	0.049

Tabella A.3: Risultati TPTP FOF con uguaglianza (18 problemi)

Problema	**FO2**	**Vampire Classic**	**FO2 + RemoveEq**
KRS026+1	120.000	0.054	58.668
KRS040+1	29.187	120.000	120.000
KRS057+1	10.518	0.007	120.000
KRS102+1	34.278	57.558	120.000
KRS163+1	49.307	0.009	120.000
KRS174+1	11.622	0.080	120.000
KRS175+1	13.154	0.029	120.000
NUM395+1	120.000	0.040	120.000
NUM531+1	120.000	0.073	120.000
NUM531+2	120.000	0.031	120.000
PUZ001+1	120.000	0.039	120.000
PUZ073+1	120.000	120.000	120.000
PUZ074+1	120.000	120.000	120.000
PUZ075+1	120.000	5.575	120.000
SEU123+1	120.000	0.040	120.000
SEU139+1	120.000	0.035	120.000
SYN071+1	120.000	0.050	120.000
SYO561+1	120.000	0.043	120.000

Tabella A.4: Risultati TPTP CNF con uguaglianza (18 problemi)

Problema	**FO2**	**Vampire Classic**	**FO2 + RemoveEq**
HWV039-1	120.135	120.042	120.154
HWV040-1	120.103	120.081	120.099
HWV041-1	120.157	120.039	120.148
HWV042-1	120.109	120.045	120.101
HWV043-1	120.143	120.054	120.111
HWV044-1	120.260	120.076	120.136
HWV045-1	120.184	120.055	120.201
HWV046-1	120.186	120.049	120.111
HWV047-1	120.156	120.060	120.102
HWV048-1	120.160	120.052	120.127
HWV049-1	120.140	120.070	120.111
HWV050-1	120.124	120.045	120.113
HWV051-1	120.112	120.044	120.122
HWV065-1	120.057	120.086	120.041
HWV066-1	120.041	120.050	120.032
SYN071-1	120.849	0.045	0.042
SYN072-1	120.728	0.060	0.045
SYN914-1	26.385	120.045	0.175

List of Algorithms

1 FO2 Preprocessing Main Pipeline 32
3 Variable Coverage Check 33
2 S2 Clause Validation 34
4 Variable Coverage Check 2 35
5 FO2 Equality Removal Main Procedure 36
6 Get Predicate for Constant 38
7 Check Constant Term 38
8 Replace Constants in Atomic Formula 39
9 Generate Uniqueness Axiom 40
10 Recursive Constant Replacement in Formula 41
11 Replace Constants in Clause 42
12 Lemma 1 – Elimination of Individual Constants 43
13 Lemma 2 – Conversione nella forma normale di Scott 45
14 Naming ricorsivo delle sottoformule di Scott 46
15 Costruzione della forma universale di Tipo 3 47
16 Costruzione della forma universale-esistenziale di Tipo 2 47
17 Asserzione della formula radice in forma di Tipo 1 47
18 Lemma 3 – Semplificazione dell'uguaglianza nei casi diagonale e distinto 49
19 Lemma 3 – Introduzione della guardia di uguaglianza 49
20 Lemma 3 – Trasformazione dei tre tipi di Scott 50
21 Lemma 3 – Trasformazione ricorsiva dei congiunti di Scott 51
22 Lemma 3 – Applicazione al problema 52
23 Lemma 4 – Estrazione delle clausole di Tipo 3 54
24 Lemma 4 – Estrazione delle clausole di Tipo 2 55
25 Lemma 4 – Risoluzione ristretta a due variabili 56
26 Lemma 4 – Semplificazione di una clausola di Tipo 2 per $x \neq y$ 57
27 Lemma 4 – Saturazione ristretta ed eliminazione dei letterali binari . 58
28 Lemma 5 – Trasformazione delle clausole con uguaglianza positiva . . 60
29 Lemma 5 – Decomposizione dei letterali in γ e δ 61
30 Lemma 5 – Eliminazione preliminare delle uguaglianze negative . . . 62
31 Lemma 6 – Eliminazione dei vincoli di unicità 64
32 Sostituzione ricorsiva dei vincoli di unicità 65
33 Generazione degli assiomi di congruenza Θ_i 66
34 Costruzione di una clausola indicizzata 67
35 Verifica della selezionabilità secondo Σ_2 68
36 Confronto tra letterali indicizzati ground 68
37 Risoluzione tra clausole indicizzate 70
38 Fattorizzazione indicizzata 71
39 Verifica della sussunzione indicizzata 71
40 Matching ricorsivo dei letterali per la sussunzione 72
41 fo2solve - Gestione dei rami 75

42 fo2solve - Saturazione del ramo corrente 76
42 fo2solve - Saturazione del ramo corrente (continua) 77
43 FO2 Main Classification 80
44 FO2 Clause Check 81
45 FO2 Formula Recursive Check 82
45 FO2 Formula Recursive Check (continua) 83

Bibliografia

[Bor] Manuele Borgia. «Generazione automatica di formule per frammenti decidibili». Lavoro di Tesi in Informatica AA2025/2026.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. 2nd. Graduate Texts in Computer Science. Springer-Verlag New York, 1996. ISBN: 978-0-387-94593-4.

[GKV97] Erich Grdel, Phokion G. Kolaitis e Moshe Y. Vardi. «On the Decision Problem for Two-Variable First-Order Logic». In: *The Bulletin of Symbolic Logic* 3.1 (1997), pp. 53–69.

[KV13] Laura Kovcs e Andrei Voronkov. «First-Order Theorem Proving and VAMPIRE». In: *Tutorial / Overview Paper on First-Order Theorem Proving*. Springer, 2013.

[NP01] Hans de Nivelle e Ian Pratt-Hartmann. «A Resolution-Based Decision Procedure for the Two-Variable Fragment with Equality». In: *Automated Reasoning – First International Joint Conference, IJCAR 2001*. A cura di Rajeev Gor, Alexander Leitsch e Tobias Nipkow. Vol. 2083. Lecture Notes in Computer Science (LNAI). Springer, 2001, pp. 211–225.

[Sut17] G. Sutcliffe. «the TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0». In: *Journal of Automated Reasoning* (2017).