ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

UNA WEBAPP PER DIMOSTRAZIONI IN DEDUZIONE NATURALE PER LA LOGICA DEL PRIMO ORDINE

Relatore:
Chiar.mo Prof.
CLAUDIO SACERDOTI COEN

Presentata da:
ORAZIO ANDREA CAPONE

I Sessione
Anno accademico 2023/2024

ABSTRACT

Questo elaborato descrive l'implementazione e i casi d'uso di una web app per la creazione e la modifica di alberi di deduzione, sia proposizionali che al prim'ordine, mirata a risolvere le problematiche riscontrate nell'uso di metodi tradizionali (carta e penna) e in progetti software passati.

INDICE

Capitolo 1: Introduzione

1.1: Contesto

La deduzione naturale proposizionale e al prim'ordine

La deduzione naturale è un sistema deduttivo utilizzato per formalizzare argomenti e dimostrazioni mediante formule logiche e regole di inferenza. Questo approccio consente di rappresentare il processo di ragionamento in modo rigoroso e strutturato, facilitando la derivazione di conclusioni corrette a partire da premesse date. Grazie alla sua chiarezza, rigore e intuitività, la deduzione naturale è ampiamente impiegata nell'insegnamento della logica e nella verifica della validità degli argomenti logici.

Dimostrazioni tradizionali vs Alberi di deduzione

Le dimostrazioni possono essere rappresentate in vari modi, ma due delle forme più comuni sono le dimostrazioni tradizionali lineari e le dimostrazioni sotto forma di alberi di deduzione.

Dimostrazioni tradizionali

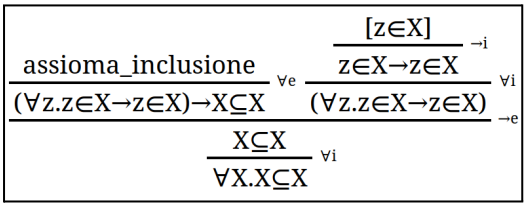
Le dimostrazioni tradizionali vengono presentate come una sequenza lineare di passi. Ogni passo rappresenta un'inferenza logica e si basa sui precedenti per arrivare alla conclusione finale. Il testo viene scritto tipicamente in linguaggio naturale, anche se alcuni software riconoscono solamente un linguaggio controllato.

Alberi di deduzione

Un albero di deduzione rappresenta la struttura logica della dimostrazione in modo grafico. Ogni nodo dell'albero rappresenta una formula e ogni linea rappresenta un inferenza logica. Questo metodo offre una visione più intuitiva delle relazioni logiche tra i vari passaggi e facilita l'individuazione di errori o passi mancanti.

```
theorem esempio1: ∀X. X⊆X.
  Sia X un insieme, dobbiamo dimostrare che X⊆X,
  ovvero ∀z.z∈X→z∈X per l'assioma di inclusione.
  Sia z un insieme tale che (z∈X)(H1),
  dobbiamo dimostrare z∈X, da H1 ovvio.
  qed.
```

Dimostrazione tradizionale



Albero di deduzione

1.2: Motivazioni

Problemi attuali

L'assenza di strumenti digitali adeguati per la creazione e la modifica di alberi di deduzione ha portato studenti e docenti a lavorare principalmente con metodi tradizionali come carta e penna. Questo approccio può presentare diverse problematiche:

- Feedback ritardato: Gli studenti ricevono feedback sui loro lavori solo dopo la revisione da parte dei docenti, ritardando il processo di apprendimento;
- Errore umano: La manipolazione manuale degli alberi di deduzione è soggetta a errori, soprattutto in dimostrazioni più grandi nelle quali aumenta il numero di ipotesi e ramificazioni;
- Fruibilità limitata: I lavori svolti su carta non sono facilmente condivisibili o modificabili, limitando la collaborazione e la revisione;
- Accessibilità: Le prove cartacee non sono fruibili da persone non vedenti.

Obiettivi del progetto

Il progetto ha come obiettivo principale la creazione di un'applicazione web che risolva le problematiche sopra menzionate, offrendo una serie di vantaggi sia per gli studenti che per i docenti.

- Fruibilità e disponibilità: Un'applicazione web è accessibile da qualsiasi dispositivo connesso ad Internet, permettendo agli utenti di lavorare sui loro alberi di deduzione sempre e ovunque;
- Accessibilità: Un'applicazione web ha il potenziale di diventare accessibile, investendovi lavoro aggiuntivo;
- Feedback immediato: L'applicazione fornisce un feedback immediato sulla correttezza degli alberi di deduzione, permettendo agli studenti di esercitarsi e comprendere meglio i principi della deduzione naturale;
- Supporto guidato: L'applicazione aiuta gli utenti nei vari step della dimostrazione, suggerendo le regole applicabili e le ipotesi che si hanno a disposizione in un determinato ramo. Inoltre, grazie a delle funzioni di autocompletamento aiuta sia gli utenti inesperti a capire quali sarebbero i passi giusti, sia gli utenti più esperti a risparmiare tempo nella creazione e nella modifica di un albero;
- Salvataggio su server: Gli utenti possono salvare i loro lavori sul server, rendendoli sempre accessibili e facilitando anche la collaborazione tra studenti e docenti.

Vantaggi per i docenti

Per i docenti l'applicazione rappresenta uno strumento didattico molto utile che può essere utilizzato per assegnare esercizi, monitorare il progresso degli studenti grazie agli strumenti di valutazione automatica e fornire un feedback immediato. Questo non solo migliora l'efficienza del processo di insegnamento, ma permette anche di individuare rapidamente aree di difficoltà comuni tra gli studenti.

Vantaggi per gli studenti

Gli studenti traggono vantaggio dall'avere un ambiente di apprendimento interattivo. L'applicazione non solo facilita la comprensione dei concetti attraverso la pratica ma consente anche di sperimentare con vari approcci e di ricevere un feedback istantaneo, accelerando il processo di apprendimento.

Capitolo 2: Confronto con altri strumenti

2.1: Matita

Matita[1] è uno strumento sviluppato presso l'Università di Bologna che offre un ambiente di lavoro per la scrittura e la verifica delle dimostrazioni lineari scritte in un linguaggio controllato. L'applicazione permette anche di mostrare a schermo gli alberi di deduzione ma la funzione non è stata più utilizzata per vari motivi.
Innanzitutto, la rappresentazione grafica degli alberi di deduzione in Matita causava significativi rallentamenti dell'applicazione. Questo problema era particolarmente evidente quando si lavorava con dimostrazioni complesse o di grandi dimensioni. Inoltre, la necessità di definire l'albero di deduzione tramite codice si è rivelata un ostacolo per molti utenti.
Un ulteriore vincolo di Matita è rappresentato dai requisiti di sistema, poiché si tratta di uno strumento desktop che richiede un ambiente Linux per essere eseguito.

Vediamo un esempio di albero di deduzione descritto in Matita. L'albero in forma grafica viene mostrato in una seconda finestra:

```
theorem esempio1: ∀X.X⊆X.

apply rule (prove ( ∀X.X⊆X ));
apply rule (∀#i {X} ( X⊆X ));
apply rule (⇒#e ( (∀z.z∈X→z∈X)→X⊆X ) ( ∀z.z∈X→z∈X ));
  [ apply rule (∀#e {X} ( ∀B.(∀z.z∈X→z∈B)→X⊆B) ));
    apply rule (∀#e {X} ( ∀A.∀B.(∀z.z∈A→z∈B)→A⊆B) ));
  | apply rule (∀#i {z} ( z∈X→z∈X ));
    apply rule (⇒#i [h1] ( z∈X ));
    apply rule (discharge [h1]);
  ]
```

L'esempio sopra riportato evidenzia come il procedimento di descrizione dell'albero sia macchinoso e risulti complicato seguire le varie diramazioni. Ogni nodo e ramo deve essere definito manualmente tramite codice, il che rende il processo lungo e soggetto a errori. Questa complessità rende difficile mantenere una visione d'insieme della dimostrazione, soprattutto quando l'albero cresce in dimensioni e profondità.

[1] An Interactive Theorem Prover for a variant of the Calculus of (Co)Inductive Constructions.

2.2: LogicPlayer

LogicPlayer è un'applicazione Android sviluppata da Danilo Berardinelli[2] e Stefano Mezza[3] nel 2014 come progetto di tesi di laurea. L'applicazione era un modo di ovviare le problematiche di Matita e offrire un'interfaccia utente dedicata con la quale poter lavorare sugli alberi di deduzione naturale. Quest'ultima prevede le funzionalità minime per svolgere e completare un esercizio utilizzando tutte le regole della logica proposizionale classica. Il problema principale, attualmente, è che il progetto è stato concluso con del codice ancora instabile; inoltre, trattandosi di un'app sviluppata per Android, essa presentava già all'inizio varie sfide: in primis perché doveva supportare vari modelli di smartphone e tablet, e in secundis perché il sistema Android era (ed è ancora oggi) in continuo sviluppo, motivo per il quale richiedeva (e richiede) un assiduo lavoro di manutenzione nel tentativo di garantire la compatibilità con le nuove versioni. In effetti l'applicazione, non più mantenuta una volta che gli studenti si sono laureati, è diventata inutilizzabile sulle nuove versioni prima ancora di essere testata in laboratorio con gli studenti.

2.3: Progetto attuale

La nuova web app è stata pensata appositamente per avere un'interfaccia user-friendly e cross-platform, poiché è intuitiva e può essere utilizzata su qualunque dispositivo dotato di browser e non richiede ulteriori installazioni. Dà la possibilità all'utente di lavorare sia con la logica proposizionale sia al prim'ordine. Inoltre, grazie ai diversi livelli di assist guida l'utente a partire dalle dimostrazioni più semplici fino a quelle più complesse.

[2] Berardinelli, Danilo (2014) *Implementazione del backend di una app Android per la didattica della deduzione naturale*.
[3] Mezza, Stefano (2014) *Implementazione del frontend di una app Android per la didattica della deduzione naturale*.

Capitolo 3: La deduzione naturale

Logica proposizionale

La deduzione naturale proposizionale cattura i ragionamenti sotto forma di variabili proposizionali (A, B, …), che rappresentano ciò che potrebbe o meno valere, e di connettivi logici come $\wedge$ *(e)*, $\vee$ *(o)*, $\rightarrow$*(implica)*, $\neg$ *(non)*. Inoltre, abbiamo due formule che rappresentano valori di verità assoluti, ovvero, $\top$*(top)* che rappresenta ciò che vale sempre e $\bot$*(bottom)* che rappresenta ciò che non vale mai.
Facciamo un esempio di cattura di un ragionamento:

"Se oggi piove allora prendo l'ombrello,
se prendo l'ombrello non mi bagno.
Quindi se oggi piove allora non mi bagno"

Può essere formalizzato come : **(P→O)** $\wedge$ **(O→¬B)** → **(P→¬B)**

Sintassi

Un albero di deduzione naturale per $\Gamma \vdash F$ è una struttura dati arborescente, tale che:

- I nodi sono etichettati con formule;
- Le foglie sono formule scaricate [G] (ipotesi locali), oppure formule non scaricate G (ipotesi globali);
- La radice è etichettata con F;
- Le foglie non scaricate sono etichettate con formule appartenenti a Γ;
- I nodi interni, oltre alla formula, sono etichettati con delle regole di inferenza.

Guardando all'esempio di prima: (P→O); (O→¬B) ⊢ (P→¬B)

$$\dfrac{\dfrac{O\rightarrow\neg B \qquad \dfrac{P\rightarrow O \quad [P]}{O}\,{\rightarrow}e}{\neg B}\,{\rightarrow}e}{P\rightarrow\neg B}\,{\rightarrow}i$$

Abbiamo come radice (P→¬B) e come foglie (O→¬B) e (P→O) che sono ipotesi globali e [P] che è un'ipotesi scaricata; i nodi interni, invece, sono passi di inferenza che analizzeremo nella prossima sezione.

Passi di inferenza

Ci sono due tipi di passi di inferenza: le regole di introduzione, che ci dicono i modi in cui concludere una formula istanza di un determinato connettivo, e le regole di eliminazione, che ci dicono i modi in cui utilizzare un'ipotesi istanza di un determinato connettivo.

Regole di introduzione

Introduzione dell'implica (→i)

Per dimostrare (A→B) assumiamo [A] (che avremo tra le ipotesi scaricabili) e passiamo a dimostrare B.

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A\to B}\ {\to}i$$

Introduzione dell'and (∧i)

Per dimostrare (A∧B) passiamo a dimostrare sia A che B.

$$\frac{A \quad B}{A\wedge B}\ {\wedge}i$$

Introduzione dell'or (∨i_L, ∨i_R)

Per dimostrare (A∨B) è sufficiente passare a dimostrare A, introducendo l'or a sinistra, oppure B, introducendo l'or a destra.

$$\frac{A}{A\vee B}\ {\vee}i\_L \qquad \frac{B}{A\vee B}\ {\vee}i\_R$$

Introduzione del top (⊤i)

Per dimostrare (⊤) non abbiamo bisogno di fare nulla, poiché (⊤) rappresenta un'affermazione sempre vera.

$$\frac{}{\top}\ \top i$$

Introduzione del not (¬i)

Per dimostrare (¬A) assumiamo [A] e passiamo a dimostrare l'assurdo *(⊥)*.

$$\frac{\begin{array}{c}[A]\\ \vdots\\ \bot\end{array}}{\neg A}\ \neg i$$

Riduzione all'assurdo (RAA)

Per dimostrare (A) possiamo assumere [¬A] e passare a dimostrare l'assurdo *(⊥)*. Si noti che questa regola appartiene alla logica classica, quindi se in una dimostrazione viene utilizzata la RAA si passa da una prova intuizionista a una classica.

$$\frac{\begin{array}{c}[\neg A]\\ \vdots\\ \bot\end{array}}{A}\ RAA$$

Regole di eliminazione

Eliminazione dell'implica (→e)

Se sto dimostrando B e ho (A→B) allora posso ridurmi a dimostrare A.

$$\frac{A\to B \quad A}{B}\ {\to}e$$

Eliminazione dell'and (∧e)

Se sto dimostrando F e ho (A∧B) posso continuare a dimostrare F avendo [A] e [B] come ipotesi.

$$\frac{A \wedge B \quad \overset{[A][B]}{\vdots} \quad F}{F}\ \wedge e$$

Se ciò che stiamo dimostrando è uguale a una delle due formule della congiunzione possiamo concludere direttamente utilizzando l'eliminazione a sinistra *(∧e_L)* se, sto dimostrando A, o l'eliminazione a destra *(∧e_R)* se, sto dimostrando B.

$$\frac{A \wedge B}{A}\ \wedge e\_L \qquad \frac{A \wedge B}{B}\ \wedge e\_R$$

Eliminazione dell'or (∨e)

Se sto dimostrando F e ho (A∨B), procediamo per casi, dimostrando F avendo come ipotesi [A], e dimostrando F avendo come ipotesi [B].

$$\frac{A \vee B \quad \overset{[A]}{\vdots} \quad F \quad \overset{[B]}{\vdots} \quad F}{F}\ \vee e$$

Eliminazione del bottom

Dato che dall'assurdo segue qualunque cosa, per dimostrare qualunque formula F possiamo ridurci a dimostrare un assurdo.

$$\frac{\bot}{F}\ \bot e$$

Eliminazione del not

Per dimostrare l'assurdo basta dimostrare qualcosa e il suo contrario.

$$\frac{\neg F \quad F}{\bot}\ \neg e$$

Deduzione naturale per la logica del prim'ordine

La logica del prim'ordine estende le proposizioni della logica proposizionale che abbiamo dato precedentemente. Poiché vengono aggiunti solamente i quantificatori universale (∀) ed esistenziale (∃) è sufficiente introdurre le apposite regole di introduzione ed eliminazione.

Introduzione del quantificatore universale (∀i)

Per dimostrare (∀x.P) è sufficiente sostituire x con una variabile y sulla quale non sappiamo nulla e poi dimostrare P[y/x]. Ovviamente è sufficiente scegliere una variabile y ancora mai usata.

$$\frac{P[y/x]}{\forall x.P}\ \forall i$$

Introduzione del quantificatore esistenziale (∃i)

Per dimostrare (∃x.P) bisogna scegliere un termine t per il quale P[t/x] valga e dimostrarlo.

$$\frac{P[t/x]}{\exists x.P}\ \exists i$$

Eliminazione del quantificatore universale ($\forall$ e)

Per dimostrare P[t/y] è sufficiente dimostrare ($\forall$ x.P), poiché se P vale per qualsiasi x, allora vale anche per qualunque termine t.

$$\frac{\forall x.P}{P[t/x]}\ \forall e$$

Eliminazione del quantificatore esistenziale ($\exists$ e)

Se sto dimostrando F e ho ($\exists$ x.P), posso continuare a dimostrare F assumendo P per una variabile generica y.

$$\frac{\exists x.P \quad \begin{array}{c}[P[y/x]] \\ \vdots \\ F\end{array}}{F}\ \exists e$$

Difficoltà ed errori tipici

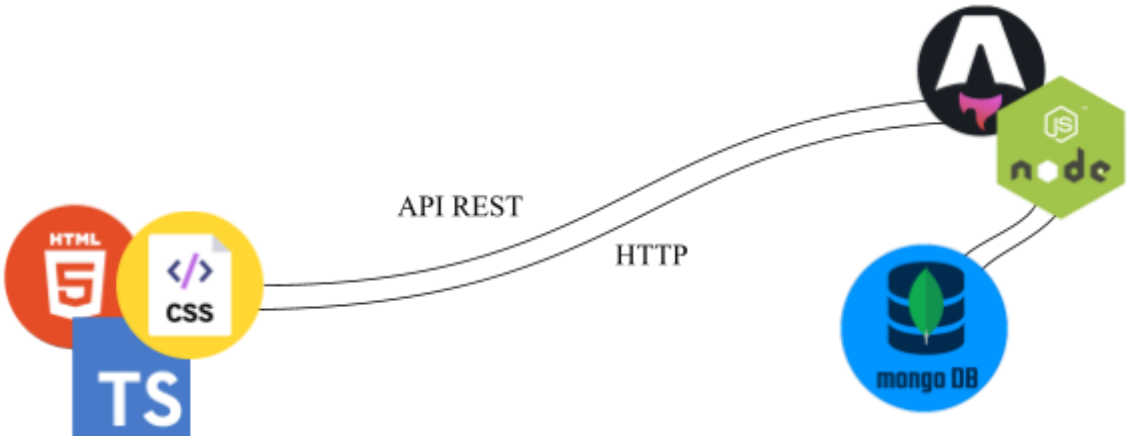
Studiare e applicare la deduzione naturale può presentare diverse sfide per gli studenti, tra le difficoltà più comuni abbiamo:

- Comprensione delle regole di inferenza: Le numerose regole di inferenza possono essere difficili da memorizzare e distinguere. Ogni regola ha specifiche condizioni d'applicazione e scopi che devono essere compresi a fondo. Inoltre, anche una volta memorizzate, applicare correttamente le regole richiede pratica e attenzione ai dettagli. Errori comuni includono la confusione tra le regole di introduzione ed eliminazione e l'uso improprio delle premesse.
- Gestione delle ipotesi: La gestione delle ipotesi durante la dimostrazione può risultare complicata, soprattutto nel caso di più diramazioni.
- Manipolazione dei quantificatori: Nella deduzione naturale del prim'ordine bisogna distinguere tra variabili legate (che sono vincolate da un quantificatore) e variabili libere. Errori nella gestione delle variabili possono portare a dimostrazioni non valide.

Capitolo 4: Architettura software del progetto

Panoramica

L'architettura del software dell'applicazione web si divide principalmente in: backend, realizzato con Node.js e Astro, che si occupa di gestire le richieste da parte dell'utente e di interagire con il database; il database, realizzato con MongoDB, che conterrà tutti i dati degli utenti e i vari alberi di deduzione; una API REST, che permette al frontend e al backend lo scambio di dati; infine, il frontend, responsabile dell'interazione con l'utente, realizzato principalmente con HTML, CSS, e Typescript per evitare problemi futuri di compatibilità con dipendenze varie. In particolare l'editor degli alberi di deduzione è stato realizzato come single-page-application in modo da ridurre al minimo il carico sul server e utilizzare al meglio la potenza di calcolo distribuita.



Backend

Come runtime di backend è stato scelto Node.js perché è facilmente scalabile e permette di avere una base di codice condivisa tra backend e frontend. Come framework è stato scelto Astro poiché porta una serie di vantaggi: permette una facile gestione delle pagine e delle API REST; permette di utilizzare componenti di altri framework (come React, Vue, Svelte e Solid); supporta l'utilizzo di Typescript; grazie al Server-Side-Rendering e la compilazione del sito web (tramite Vite) permette di ridurre al minimo le dimensioni delle pagine e quindi velocizzare il caricamento da parte del client.
Il backend riceve e gestisce le varie richieste degli utenti, servendo le pagine e esponendo le API; si occupa dell'autenticazione, effettuata tramite JSON Web Token, che è uno standard web per lo scambio di dati, esso permette di cifrare e firmare il contenuto del token che poi verrà salvato come cookie server-only per garantire il massimo della sicurezza; si occupa anche della comunicazione con il database e della gestione delle query tramite il connettore di MongoDB.

Database

Come database è stato scelto MongoDB, un DBMS non relazionale orientato ai documenti. La ragione principale di questa scelta è stata che tutti i dati utilizzati dall'applicazione sono in formato JSON, che permette di gestire facilmente lo scambio di dati tra client e server ed è di facile conversione tra testo e oggetti. Dato che MongoDB utilizza un formato a documenti abbiamo una corrispondenza uno a uno tra i dati salvati e quelli richiesti dall'applicazione.

API

Le API si occupano di facilitare lo scambio di dati tra server e client dopo il caricamento della pagina. Tra le funzioni principali troviamo il salvataggio e il caricamento degli alberi; la gestione di login e logout; la richiesta di dati come la lista degli esercizi, la lista degli studenti iscritti a un corso, ecc…
Questa struttura permette di richiedere a runtime solo i dati necessari per una funzione riducendo il carico sul server e il traffico al minimo.

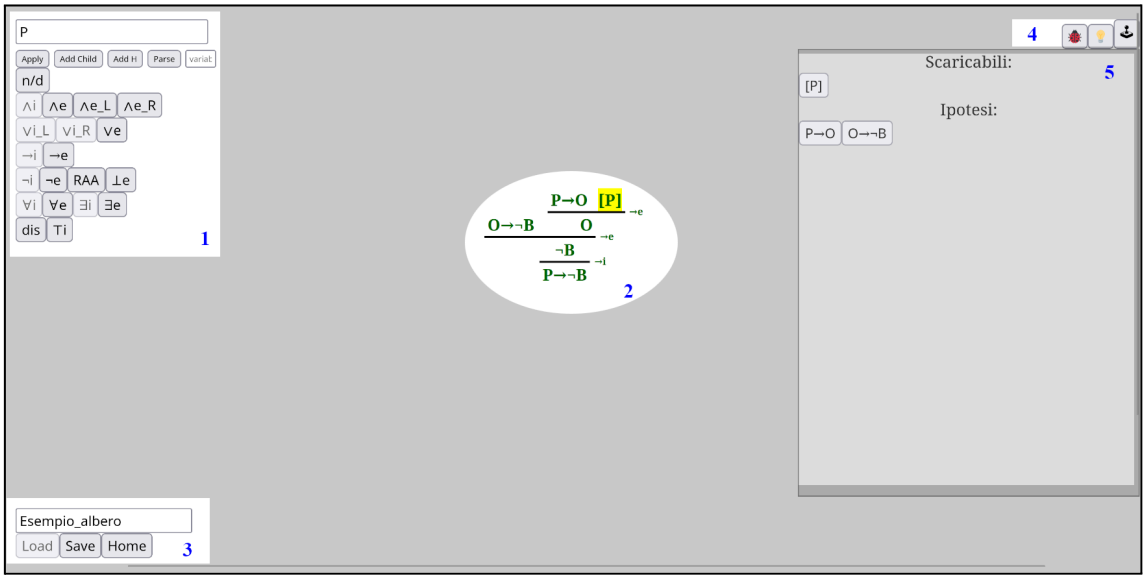
Frontend

Nel frontend si è cercato di limitare il più possibile l'utilizzo di framework per ridurre al minimo eventuali problemi di compatibilità futuri e rendendo il codice il più modulare possibile in modo da consentire facilmente il passaggio a un ambiente diverso.

La pagina dell'editor è stata realizzata esclusivamente in HTML, CSS e Typescript, mentre per le altre pagine è stato utilizzato anche Bootstrap per dare una resa grafica migliore. Nel prossimo capitolo approfondiremo tutte le funzioni disponibili all'utente e le modalità di interazione.

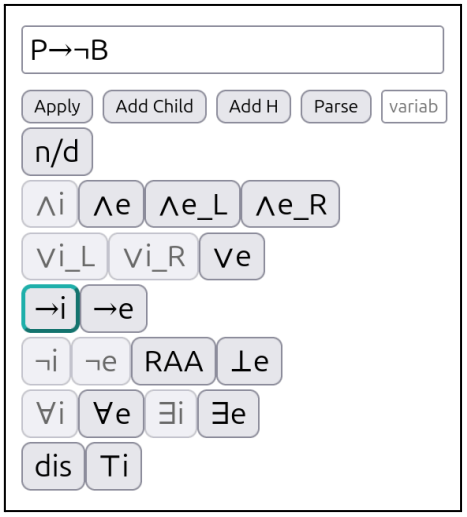
Capitolo 5: L'interfaccia utente

Iniziamo con il descrivere la pagina dell'editor degli alberi che è il fulcro dell'applicazione.



In alto a sinistra (**1**) troviamo l'editor delle formule, al centro (**2**) la visualizzazione dell'albero, in basso a sinistra (**3**) la sezione per caricare e salvare i file e tornare alla Home, e in alto a destra (**4**) i bottoni per cambiare la modalità dell'inspector (**5**) tra debugging, ipotesi e opzioni.

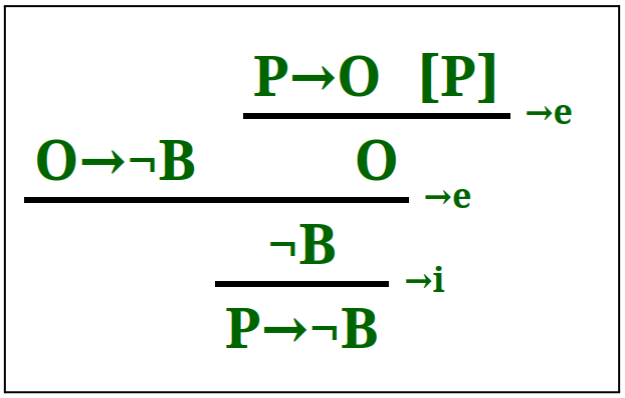
Editor formule



L'editor delle formule permette di modificare il nodo dell'albero che si ha attualmente in focus. È costituito da: un input box, nella quale si può scrivere una formula. Esso implementa anche

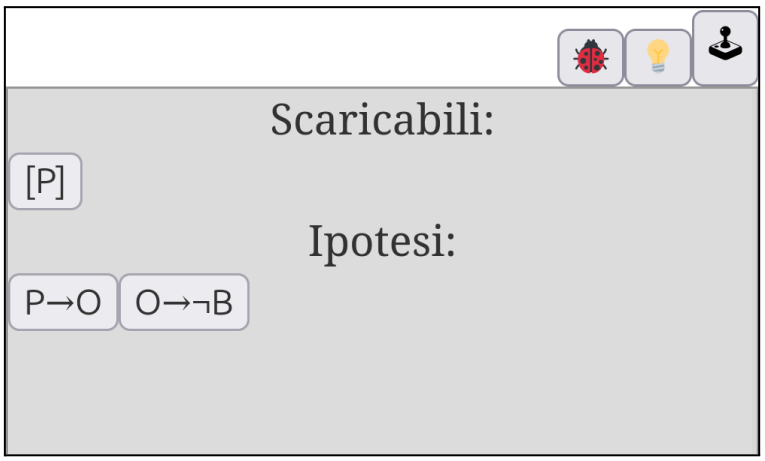
una funzione di sostituzione automatica dei simboli, per esempio scrivendo \forall si ottiene il simbolo ∀; il bottone Apply consente di applicare le modifiche al nodo dell'albero selezionato, aggiornando il contenuto; il bottone Add Child consente di aggiungere un nodo figlio al nodo attualmente selezionato; il bottone Add H consente di aggiungere l'ipotesi attualmente digitata all'insieme delle ipotesi globali; Il bottone Parse esegue la funzione di pretty-parsing sulla formula digitata, rimuovendo eventuali parentesi e simboli non necessari; i bottoni rimanenti servono ad applicare la relativa regola di inferenza al nodo selezionato

Visualizzazione dell'albero



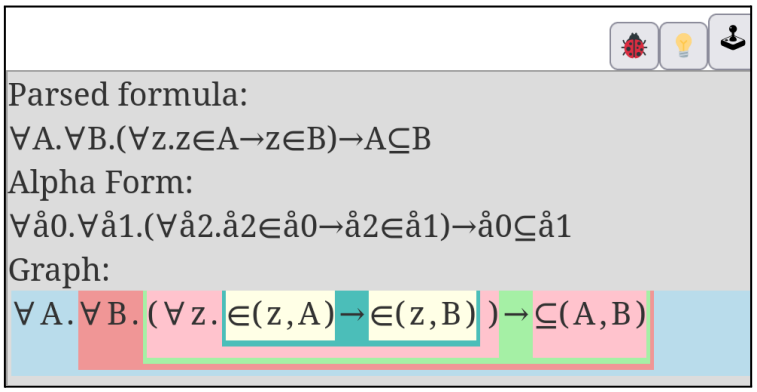
Al centro della pagina abbiamo la visualizzazione dell'albero. Possiamo spostare la visuale tramite la rotellina del mouse o attraverso le gestures. Cliccando un nodo questo verrà evidenziato, insieme a tutti i suoi figli, e diventerà il focus dell'editor delle formule, consentendoci di modificarlo.

Inspector Ipotesi



L’inspector delle ipotesi ci mostra le ipotesi che abbiamo a disposizione al nodo selezionato; questo ci aiuta a tenere traccia delle ipotesi scaricabili che acquisiamo nel corso della dimostrazione nei vari rami. Facendo clic su un’ipotesi questa viene applicata automaticamente al nodo selezionato. Inoltre, tenendo premuto SHIFT e cliccando un’ipotesi, se abbiamo attivato l’apposito assist, verranno creati in automatico eventuali passaggi intermedi che vanno dall’ipotesi alla conclusione che stiamo dimostrando.

Inspector Debugging



L’inspector del debugging ci mostra alcune informazioni in più sulla formula che stiamo digitando. Possiamo vedere in tempo reale come viene effettuato il parsing della formula, così da capire se stiamo commettendo degli errori e stiamo digitando una formula non valida. Nel caso di utilizzo di quantificatori viene mostrata anche l’alpha form, che ci permette di riconoscere formule uguali, ma con variabili diverse, rappresentandole con la loro forma canonica secondo la convenzione di Barendregt. Nella sezione Graph abbiamo un grafico colorato che ci aiuta a capire la gerarchia della formula digitata, per evitare errori con le precedenze dei connettivi.

Inspector Opzioni

L'inspector delle opzioni ci permette di cambiare il comportamento dell'editor. "LiveValid" ci permette di vedere in tempo reale se l'albero è corretto o meno, colorando di verde le parti corrette e in rosso quelle scorrette o ancora non dimostrate. "ShowIpotesi" permette o meno l'utilizzo dell'inspector delle ipotesi (a uso didattico). Sotto "assist" troviamo tutte le voci per abilitare o disabilitare i vari assist che andremo a descrivere nella sezione successiva.

Assists

addChildsOnRule

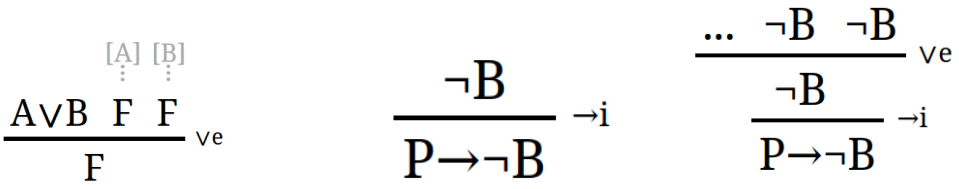
Questo assist, quando applichiamo una regola di inferenza tramite l'editor delle formule, aggiunge in automatico il numero di figli richiesti dalla regola.
Per esempio, se applichiamo $\vee$e (or eliminazione) verranno aggiunti 3 figli.

$$\frac{\neg B}{P \rightarrow \neg B}\ \rightarrow i \qquad\qquad \frac{\begin{array}{c}\cdots\ \ \cdots\ \ \cdots \\ \hline \neg B\end{array}\ \vee e}{P \rightarrow \neg B}\ \rightarrow i$$

autoCompileOnRule

Questo assist, quando applichiamo una regola di inferenza tramite l'editor delle formule, compila in automatico le formule dei figli di cui è possibile determinare il valore in modo assoluto. Per funzionare richiede che sia attivo anche addChildsOnRule.

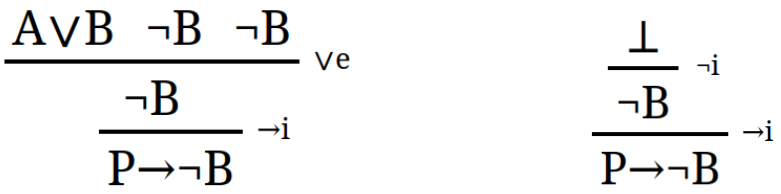
Per esempio, sempre applicando ∨e (or eliminazione), se guardiamo alla regola di inferenza possiamo notare che due dei figli hanno sempre lo stesso valore.



autoDeleteOnRule

Questo assist, quando applichiamo una regola di inferenza tramite l'editor delle formule, rimuove in automatico eventuali figli in più rispetto a quelli richiesti dalla regola applicata.

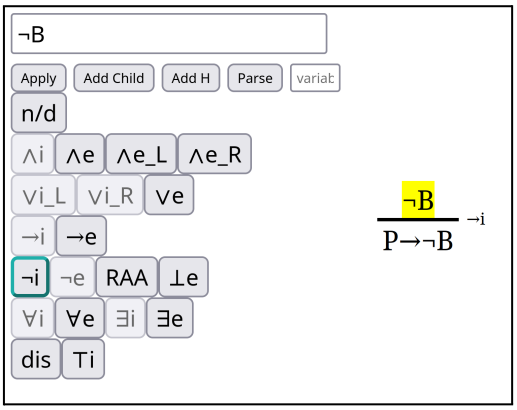
Per esempio, passando da ∨e, che richiede tre figli, a ¬i, che ne richiede soltanto uno, i due figli in eccesso vengono eliminati automaticamente.



autoDisableRules

Questo assist, quando selezioniamo un nodo, disattiva le regole non applicabili, inoltre, evidenzia la regola consigliata per quel connettivo logico.

Per esempio, se stiamo dimostrando ¬B, ci viene consigliato di applicare ¬i (l'introduzione del not).



autoCompileOnApply

Questo assist, quando andiamo a modificare la formula in un nodo, nel momento in cui premiamo Apply va ad aggiornare in automatico i nodi circostanti in modo da rispettare la formula.

Per esempio, se abbiamo →e (eliminazione dell'implica), andando a modificare la premessa dell'implica da A a C viene aggiornato anche il nodo adiacente.

$$\frac{A \to B \quad A}{B}\ \to e \qquad \frac{C \to B \quad C}{B}\ \to e$$

autoChangeVariable

Questo assist, quando andiamo a modificare un nodo che contiene una regola di inferenza riguardante un quantificatore, se premiamo SHIFT+Apply andrà a sostituire in automatico la variabile legata dal quantificatore con quella inserita nel campo di input variabile (accanto al bottone Parse).

Per esempio se abbiamo ($\forall x.P(x)$) con $\forall i$ (introduzione del quantificatore universale), e nel campo variabile abbiamo inserito y, premendo SHIFT+Apply la formula del figlio verrà aggiornata sostituendo x con y.

$$\frac{\dots}{\forall x.P(x)}\ \forall i \qquad \frac{P(y)}{\forall x.P(x)}\ \forall i$$

autoApplyHypotesis

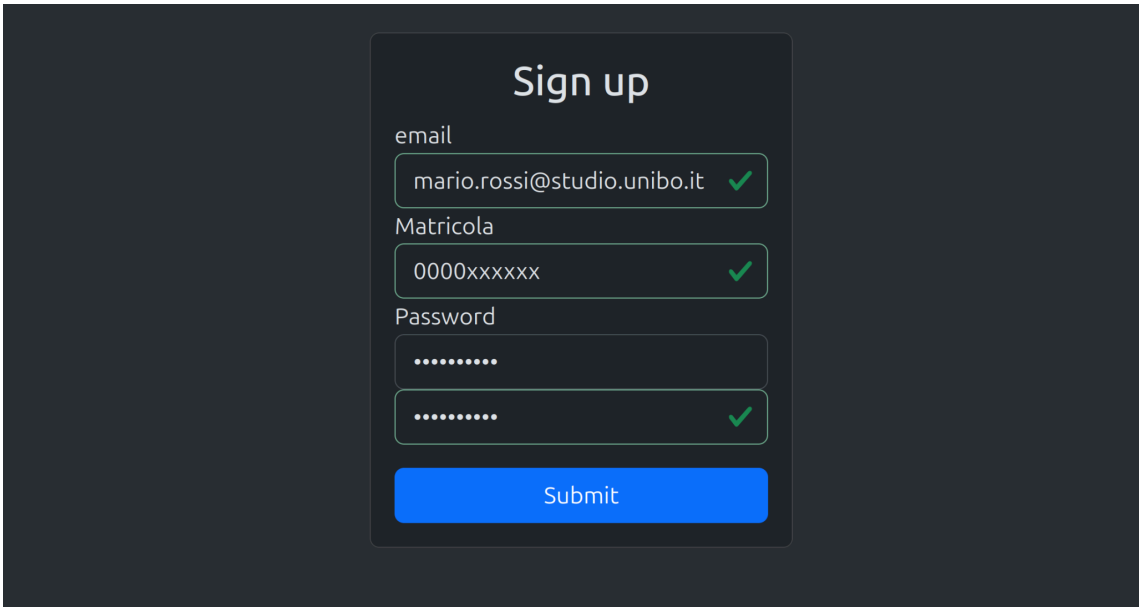
Questo assis ci permette di applicare le ipotesi costruendo in maniera automatica i passi di inferenza che vanno dall'ipotesi a quello che stiamo dimostrando (nei limiti di quello che è deducibile in modo deterministico).

Per esempio, se stiamo dimostrando A e abbiamo come ipotesi (D→C→A∧B), tenendo premuto SHIFT e cliccando il corrispettivo bottone dell'ipotesi, verranno applicati in modo ricorsivo tutti i passi intermedi.

$$\dfrac{\dfrac{\dfrac{\dfrac{D\to C\to A\wedge B \quad D}{C\to A\wedge B}\to e \qquad C}{A\wedge B}\to e}{A}\wedge e\_L}{A\vee F}\vee i\_L \qquad \dfrac{\dfrac{A}{A\vee F}\vee i\_L}$$

Altre pagine

Pagina di Sign Up



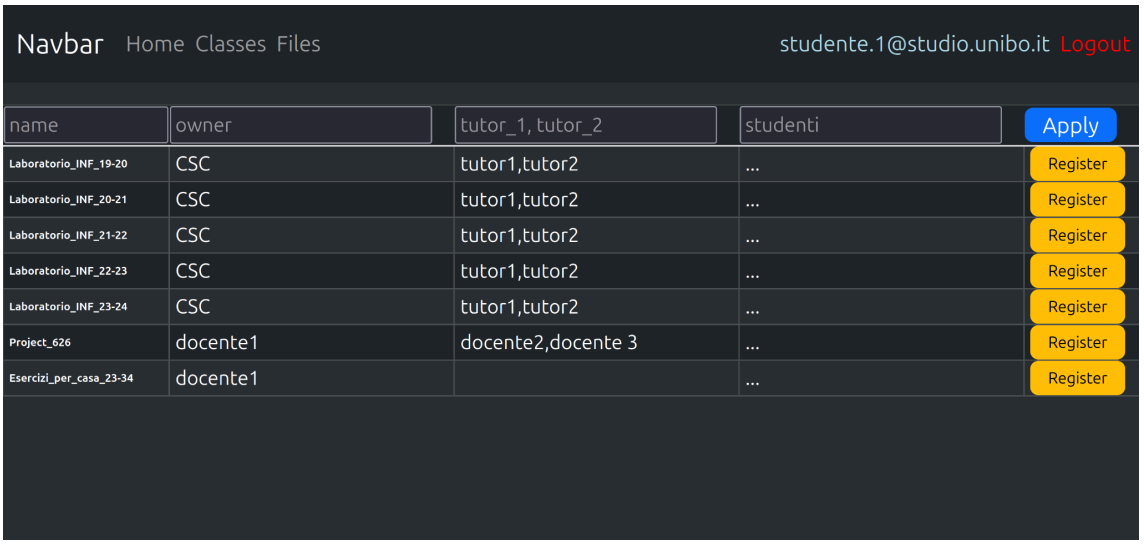
In questa pagina è possibile registrare un nuovo utente, vengono richiesti email istituzionale e numero di matricola per facilitare la ricerca degli studenti all'interno dei corsi. Il metodo ideale

per la registrazione degli studenti all'applicativo sarebbe importare in massa i profili dell'ateneo.
Ci sono diversi livelli di account utente: ADMIN, DOCENTE, TUTOR e STUDENTE.
Un utente appena registrato viene impostato di default come STUDENTE. Questo può essere facilmente cambiato da un ADMIN.

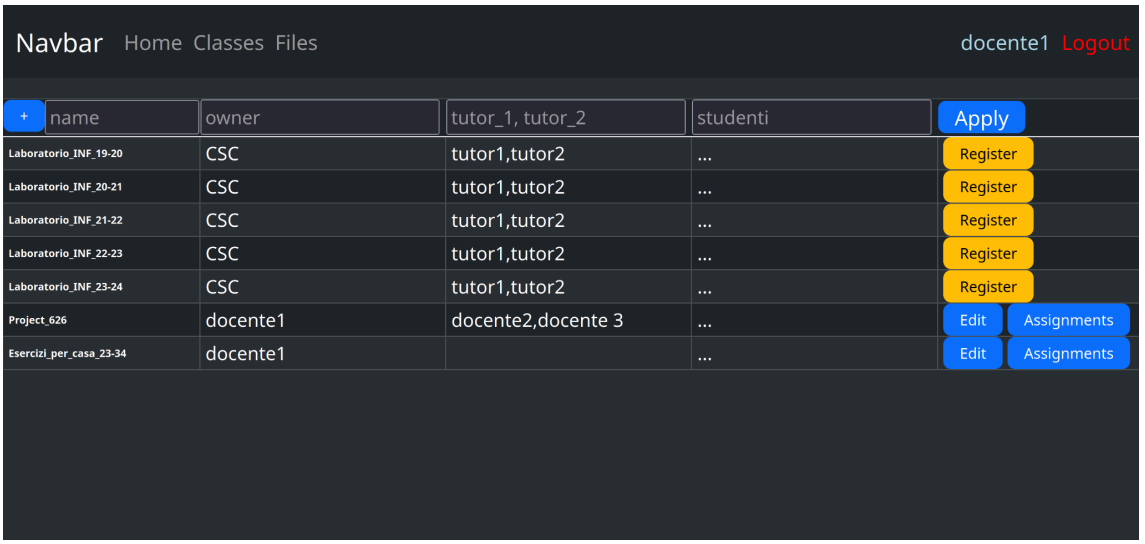
Navbar

Dopo aver effettuato il login ci comparirà in cima ad ogni pagina la Navbar. Questa ci permette di navigare facilmente tra le varie pagine del sito e ci mostra l'account con cui abbiamo effettuato il login sulla destra.

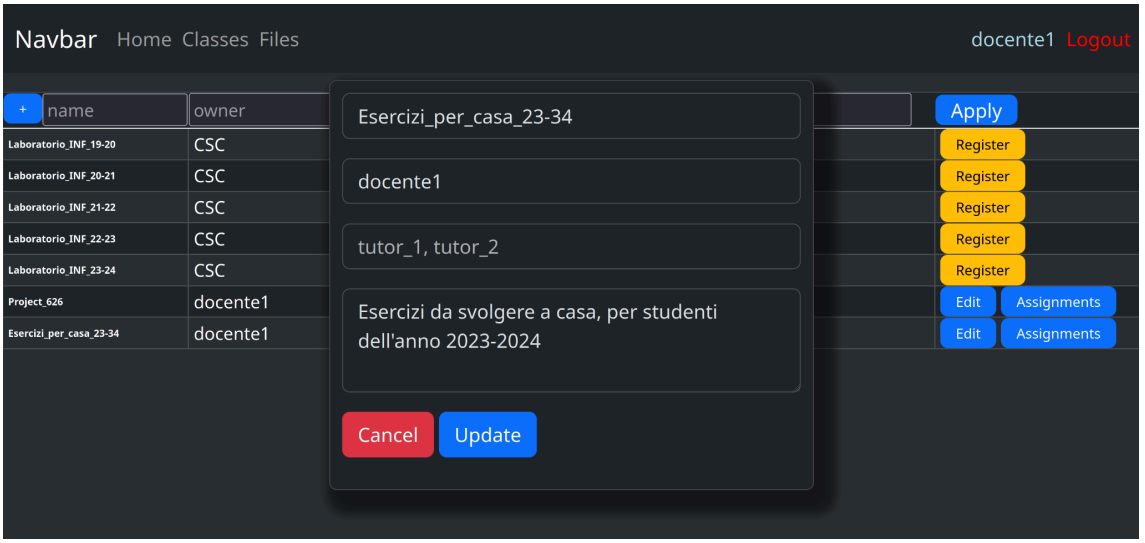
Classes



La pagina Classes ci mostra l'elenco dei vari corsi a cui possiamo partecipare. Se siamo degli studenti possiamo registrarci a un corso, inoltre tramite i campi nell'intestazione della tabella e il tasto Apply possiamo filtrare i risultati che ci vengono mostrati.
I corsi possono essere ad accesso libero oppure possono richiedere un codice per la registrazione.

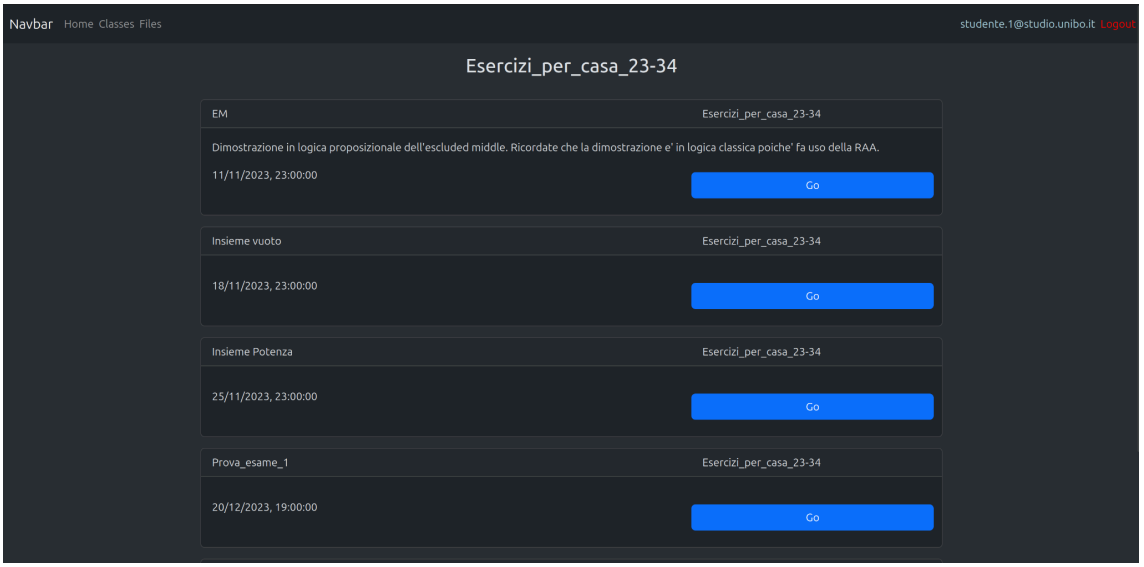


Se siamo dei docenti o dei tutor possiamo modificare i dettagli di un corso tramite il bottone "Edit", oppure crearne uno nuovo tramite il bottone "+" situato nella parte sinistra dell'intestazione della tabella.

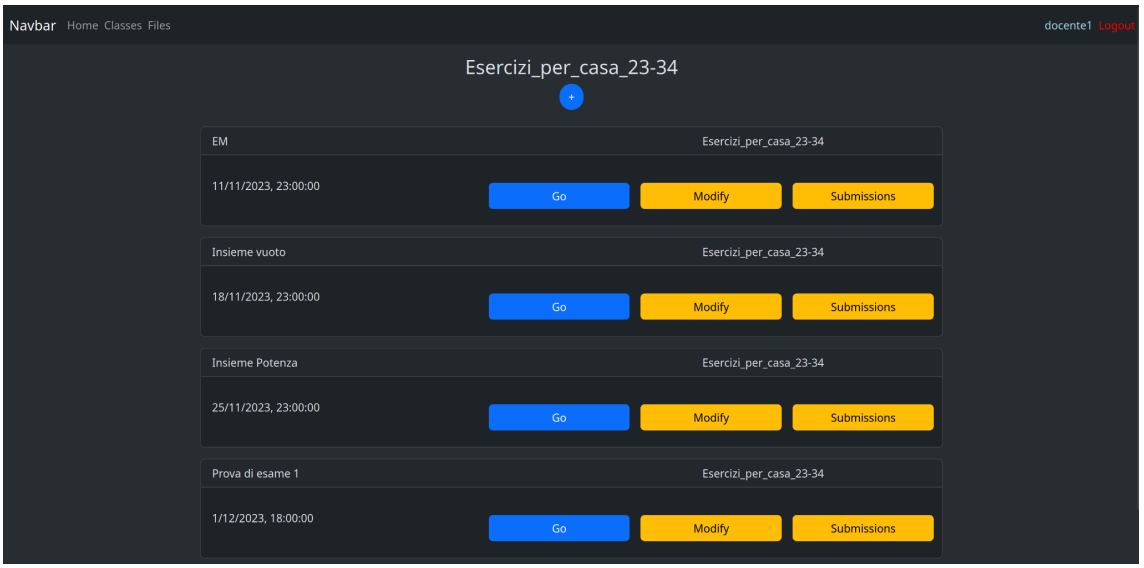


Cliccando il bottone "+" ci appare questo dialog tramite il quale possiamo compilare le informazioni del corso che vogliamo creare. Lo stesso dialog è utilizzato anche per modificare i dettagli di un corso quando si preme "Edit".

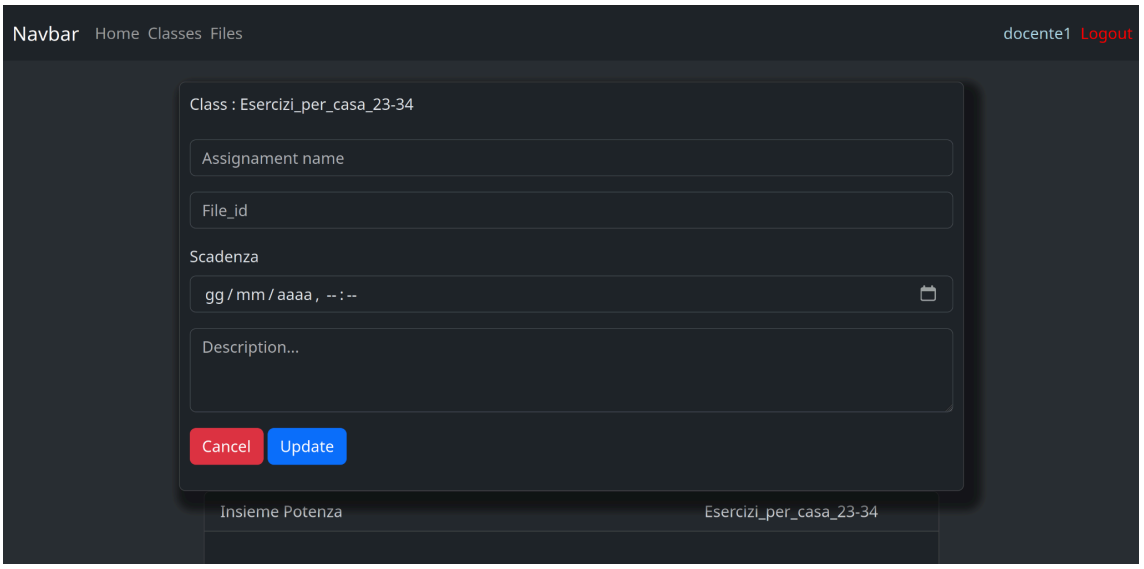
Assignments



La pagina Assignments è accessibile tramite il corrispettivo bottone presente nell'elenco dei corsi. Se siamo degli studenti ci mostra l'elenco degli esercizi che possiamo svolgere.
Ogni esercizio ha un nome, una descrizione (opzionale), una data di scadenza (opzionale) e un bottone "Go" che ci porterà all'interno dell'editor permettendoci di completare l'esercizio.

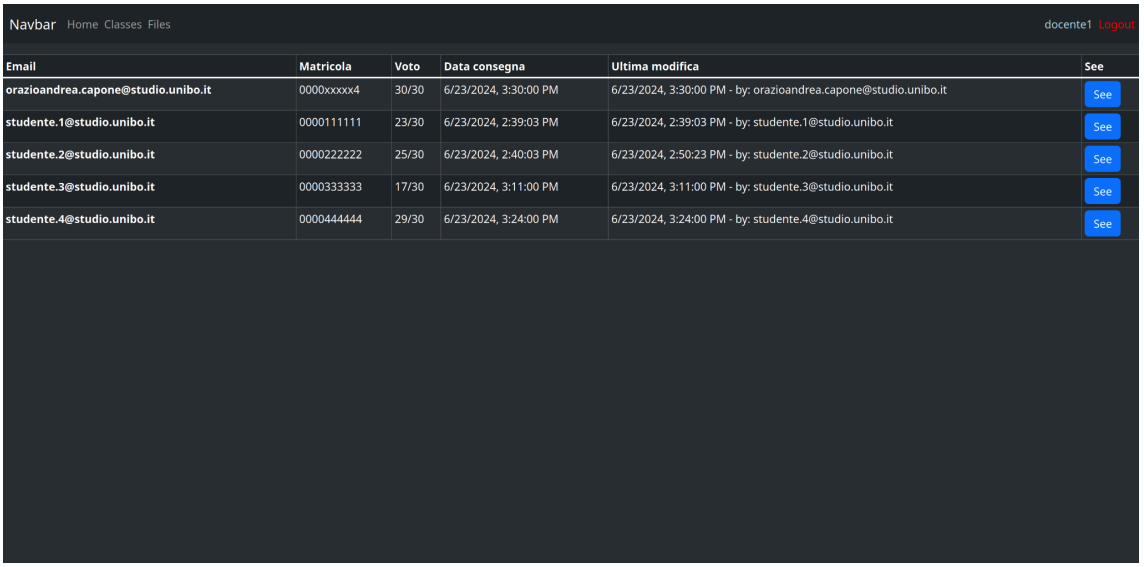


Dal punto di vista di un docente o tutor (se hanno permessi sul corso) possiamo modificare i dettagli di un esercizio tramite "Modify", vedere l'elenco delle sottomissioni tramite "Submission" oppure creare un nuovo esercizio tramite "+".



Premendo "+" (o "Modify") ci appare il dialog per inserire i dettagli di un esercizio. Possiamo scegliere il nome, la data di scadenza e la descrizione. Tramite il campo File_id possiamo indicare uno dei nostri file come template dell'esercizio, l'applicazione ne creerà una copia e sarà il punto di partenza per ogni studente. Così facendo è possibile sia definire le ipotesi iniziali e la conclusione da dimostrare, che creare anche degli esercizi particolari come degli alberi incompleti oppure contenenti degli errori da risolvere.

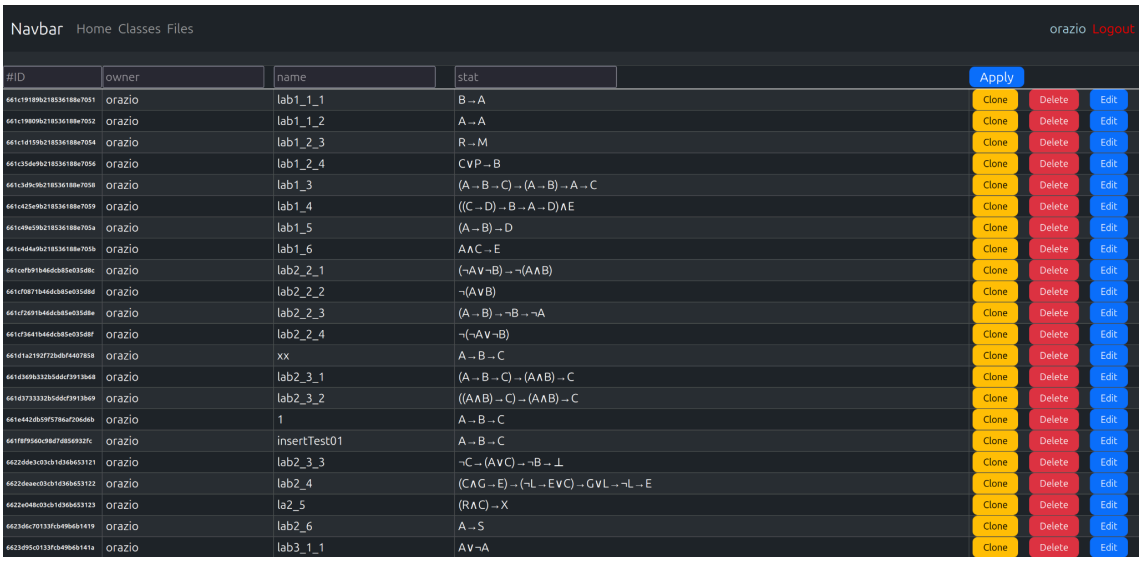
Submissions



La pagina delle submission è accessibile da docenti o tutor tramite il bottone "Submission" presente nella pagina degli Assignments. Ci presenta l'elenco delle consegne per ogni studente, mostrandoci email, matricola, voto, data di consegna e data di ultima modifica. Una volta superata la data di scadenza uno studente può visualizzare l'albero da lui creato ma non può

effettuare modifiche. I docenti e i tutor invece possono sempre modificare i file, anche dopo la data di scadenza, per risolvere eventuali errori. Per tale ragione, viene salvata anche la data dell'ultima modifica insieme al responsabile.

Files



La pagina Files ci mostra l'elenco dei nostri file personali; questi possono essere filtrati tramite l'intestazione della tabella. Qui possiamo creare, modificare, clonare o eliminare un file.

Capitolo 6: Implementazioni

Qui di seguito verranno esposte le implementazioni principali sulle quali si basa il funzionamento dell'app. È stato usato un modello a oggetti in modo da poter sfruttare al massimo l'ereditarietà e rendere il codice il più modulare possibile. La struttura degli oggetti è stata divisa in due categorie principali: quelli che si occupano della parte "visuale", che hanno come padre la classe Foglia, e quelli che si occupano della parte "logica" che hanno come padre la classe REGOLA e la classe FORMULA.

Classe Foglia

La classe Foglia descrive come è formato un nodo dell'albero e si occupa della sua visualizzazione a schermo e dell'interazione con l'utente. Vediamo i suoi attributi principali:

- string text: contiene la conclusione del nodo sotto forma di testo, ed è utile per la visualizzazione a schermo e per il parsing in JSON dell'albero. Inoltre, quando il suo valore viene modificato viene aggiornata in automatico la struttura logica e viene ricalcolata la visualizzazione dell'albero;
- Foglia parent: è un attributo di tipo foglia che contiene un riferimento al nodo padre;
- Foglia[] childs: è un attributo che contiene la lista dei nodi figli;
- FORMULA formula: è un attributo di tipo FORMULA che contiene la conclusione del nodo sotto forma di oggetto, e viene aggiornata in modo automatico quando si modifica il campo text (viene effettuato il parsing da testo a oggetto);
- REGOLA regola: è un attributo di tipo REGOLA che contiene la regola di inferenza del nodo sotto forma di oggetto, e viene modificato in automatico quando si applica una regola.
- Draw(): è il metodo che si occupa della resa visiva del nodo;
- RecursiveUpDraw(): è un metodo che chiama ricorsivamente se stesso sul nodo padre fino ad arrivare al nodo radice; dopodiché chiama il metodo RecursiveDownDraw(). Ciò permette di aggiornare la resa visiva di tutto l'albero;
- RecursiveDownDraw(): è un metodo che si occupa di chiamare il metodo Draw() per il nodo corrente; dopodiché chiama se stesso in tutti i nodi figli;
- AddChild(Foglia f): è un metodo che prende in input una Foglia f e la aggiunge come figlio al nodo corrente;
- RemoveChild(Foglia f): è un metodo che prende in input una Foglia f e, se fa parte dei figli del nodo, la rimuove;
- Delete(): è un metodo che elimina il nodo corrente;

- OnConclusionClick(): è un metodo che viene chiamato quando viene cliccata la conclusione del nodo corrente. Si occupa di impostare il focus sul nodo cliccato;
- root(): è un metodo che restituisce la Foglia che è alla radice dell'albero;
- SetRegola(REGOLA r): è un metodo che prende in input una REGOLA r e si occupa di applicarla al nodo corrente. Viene chiamato quando si clicca il bottone della corrispettiva regola di inferenza;
- collapsed(bool state): è un metodo che prende in input un booleano; se il valore è vero collassa il sottoalbero del nodo corrente, se è falso viene espanso;
- addIpotesi(string text): è un metodo che prende in input una formula sotto forma di testo e la aggiunge come ipotesi globale all'albero;
- toJSON(): è un metodo che restituisce il risultato della conversione da Foglia a JSON. Viene chiamato in modo ricorsivo sui figli;
- fromJSON(): è il metodo complementare a toJSON(), ergo restituisce il risultato della conversione da JSON a Foglia.

Classe FORMULA

La classe FORMULA è la classe padre di tutte le formule, essa ha un metodo per essere convertita in testo (toString()) e un metodo che si occupa del confronto con un'altra formula (Equal(FORMULA f)). Inoltre, tiene traccia del simbolo del connettivo e del tipo di operatore.

```
export class FORMULA{
    public sym=undefined;
    public type:op_type=undefined;
    constructor(){
    }

    get string():string{
        return this.toString();
    }
    public toString():string{
        return "n/d";
    }

    public Equal(f:FORMULA):boolean{
        if(!f)
            return false;
        const r:string = Parse(this.toString()).toString();
        const l:string = Parse(f.toString()).toString();

        return (r==l);
    }

}
```

Da essa vengono ereditate tutte le classi che rappresentano un connettivo logico. Vediamo di seguito qualche esempio.

Classe VAR (extends FORMULA)

Serve a rappresentare una variabile proposizionale e ha un attributo che contiene il nome della variabile. Il getter exisestial() serve a capire se la variabile è esistenziale o meno.

```
export class VAR extends FORMULA{

    public f:string=undefined;

    constructor(f:string){
        super();
        this.f=f;
    }

    public toString():string{
        return this.f+"";
    }

    get existential():VAR{
        const s = this.toString();
        if(s && s[0]=="?")
            return new VAR(s.slice(1, s.length));
        else
            return undefined;
    }

}
```

Classe AND (extends FORMULA)

Ha due attributi di tipo FORMULA di nome left e right, che contengono i membri sinistri e destri del connettivo. Il metodo toString() viene sovrascritto per restituire (left.toString() ∧ right.toString()). Le classi OR, IMPLY e DOUBLEIMPLY sono costruite in modo analogo.

```
export class AND extends FORMULA{

    public left:FORMULA;
    public right:FORMULA;
    public sym="∧";
    public type:op_type=op_type.BIN;

    constructor(left:FORMULA=undefined, right:FORMULA=undefine
        super();
        this.left=left;
        this.right=right;
    }

    public toString():string{
        const l= this.left ? this.left.toString() : undefined;
        const r= this.right ? this.right.toString() : undefine
        return (""+l+"∧"+r+"");
    }

}
```

Classe FORALL (extends FORMULA)

Ha un attributo di tipo VAR che contiene la variabile legata dal quantificatore e un attributo di tipo FORMULA che contiene a sua volta la formula legata. Il metodo ChangeVariable(FORMULA new_v) data una variabile in input sostituisce la variabile corrente con la nuova variabile all'interno della formula legata. alphaForm() restituisce l'alpha form della formula.

GetSwappedVariable(FORMULA f) confronta il FORALL con una formula f e restituisce la variabile che se sostituita rende uguale il FORALL con la formula f . La classe EXISTS è analoga.

```
export class FORALL extends FORMULA{
    public v:VAR;
    public p:FORMULA;
    public sym="∀";
    public type:op_type=op_type.BINDER;

    constructor(variable:VAR=undefined, property:FORMULA=undefined){
        super();
        this.v = variable;
        this.p=property;
    }

    public toString():string{
        return ( "∀"+this.v?.string+"."+ this.p?.string +"" );
    }

    public ChangeVariable(new_V:FORMULA):FORMULA{
        var p=this.p;
        if(!new_V.Equal(this.v)){
            p = RemoveShadowing(this.p, new_V, 0) as FORALL;

        }
        return  Parse(SwapVariable(p, this.v, new_V).toString());


    }

    public GetSwappedVariable(f:FORMULA):FORMULA{
        return FindSwappedVariable(this.p, f, this.v);
    }

    public Equal(f: FORMULA): boolean {
        if(!f)
            return false;
        const r:string = Parse(this.alphaForm.toString()).toString();
        var l:string;

        f = RemoveOuterBrackets(f);
        if(f instanceof FORALL){
            l = Parse(f.alphaForm.toString()).toString();

            return (r==l);
        }else
            return false;
    }

    get alphaForm():FORALL{
        return Parse(ToAplphaForm(this, {}).toString()) as FORALL;
    }
}
```

Classe PROPERTY (extends FORMULA)

È la classe che si occupa di rappresentare una proprietà P. Ha un attributo che contiene il nome della proprietà e un array che contiene i termini ai quali la proprietà è applicata.
Per esempio P(x, y) avrà come nome P e come array di termini [x, y]. Da questa classe derivano tutte le classi che descrivono una proprietà, come $\in$, =, $\subseteq$, $\cup$, ecc…

```
export class PROPERTY extends FORMULA{

    public p:string;
    public termini: termine[];
    public type:op_type=op_type.PROPERTY;

    constructor(p:string=undefined, termini:termine[]=undefined){
        super();
        this.p=p;
        this.termini=termini;
    }

    public toString():string{
        if(!this.termini||!this.p)
            return undefined;
        var arr:string = "";
        for(var i=0; i<this.termini.length; i++){
            arr+=this.termini[i]?.toString();
            if(i<this.termini.length-1)
                arr+=",";
        }
        return ( this.p+"("+arr+")" );
    }

}
```

Facciamo un esempio di come viene rappresentata una formula sotto forma di oggetto.
La formula $(A \rightarrow A \wedge B)$ sarà rappresentata come IMPLY(A, AND(A, B))

Classe REGOLA

La classe regola è la classe padre di tutte le regole di inferenza. Vediamo i suoi attributi principali:

- FORMULA conclusione: contiene la formula rappresentante la conclusione della regola;
- REGOLA[] premesse: è un array che contiene i sottoalberi rappresentanti le premesse della regola;
- FORMULA[] scaricabili: è un array che contiene le ipotesi scaricabili che si hanno a disposizione a questo punto della dimostrazione;
- FORMULA[] ipotesi: è un array che contiene le ipotesi globali della dimostrazione;
- AddScaricabili(FORMULA f): è un metodo che prende in input una formula e la aggiunge alle ipotesi scaricabili;

- AddIpotesi(FORMULA f): è un metodo che prende in input una formula e la aggiunge alle ipotesi globali;
- Charge(): è un metodo che si occupa di calcolare eventuali ipotesi scaricabili che derivano dall'applicazione della regola e di aggiungerle ai corretti sottoalberi;
- ValidRule(): è un metodo che si occupa di verificare se la regola è applicata correttamente.

Classe and_e (extends REGOLA)

È la classe che rappresenta l'eliminazione dell'and. Il metodo ValidRule() viene sovrascritto in modo da verificare che la prima premessa sia di tipo AND e che la seconda premessa sia uguale alla conclusione. Il metodo Charge() aggiunge alle ipotesi scaricabili del sottoalbero della seconda premessa il membro sinistro e destro dell'AND.

```
export class and_e extends REGOLA{

    constructor(conclusione:FORMULA=undefined, premesse:REGOLA[]=[]){
        super(conclusione, premesse);
        this.label="∧e";
        this.required_childs=2;
    }

    public Charge(){
        super.Charge();

        if(this.premesse.length<2 || !this.premesse[0] || !this.premesse[1])
            return;

        var f:FORMULA=this.premesse[0].conclusione;
        if(!f || !(f instanceof AND))
            return;

        if(f instanceof DOUBLEIMPLY){
            f= new AND(new IMPLY(f.left, f.right), new IMPLY(f.right, f.left));
        }


        const and_f:AND=f as AND;

        this.premesse[1].AddScaricabili(and_f.left);
        this.premesse[1].AddScaricabili(and_f.right);

    }

    public ValidRule(): valid_state {
        const and_f : FORMULA = this.premesse[0]?.conclusione;
        const p1 : FORMULA = this.premesse[1]?.conclusione;
        if(!and_f || !p1)
            return valid_state.NP;

        if(and_f instanceof AND && p1.Equal(this.conclusione))
            return valid_state.V
        else
            return valid_state.NV;
    }
}
```

Classe imply_i (extends REGOLA)

È la classe che rappresenta l'introduzione dell'implica. Il metodo ValidRule() viene sovrascritto in modo da verificare che la conclusione sia di tipo IMPLY e che la premessa sia uguale alla conclusione dell'implica. Il metodo Charge() aggiunge alle ipotesi scaricabili la premessa dell'implica.

```
export class imply_i extends REGOLA{

    constructor(conclusione:FORMULA=undefined, premesse:REGOLA[]=[]){
        super(conclusione, premesse);
        this.label="→i";
        this.required_childs=1;
    }

    public Charge(): void {
        super.Charge();

        if(this.premesse.length<1 || !this.premesse[0])
            return;

        const f:FORMULA = this.conclusione;
        if(!f || !(f instanceof IMPLY))
            return;

        const imply_f:IMPLY=f;
        this.premesse[0].AddScaricabili(imply_f.left);
    }

    public ValidRule(): valid_state {
        const imply_f : FORMULA = this.conclusione;
        const p0 : FORMULA = this.premesse[0]?.conclusione;

        if(!imply_f || !p0)
            return valid_state.NP;

        if(imply_f instanceof IMPLY && p0.Equal(imply_f.right))
            return valid_state.V
        else
            return valid_state.NV;
    }
}
```

Le altre regole sono implementate in modo analogo.

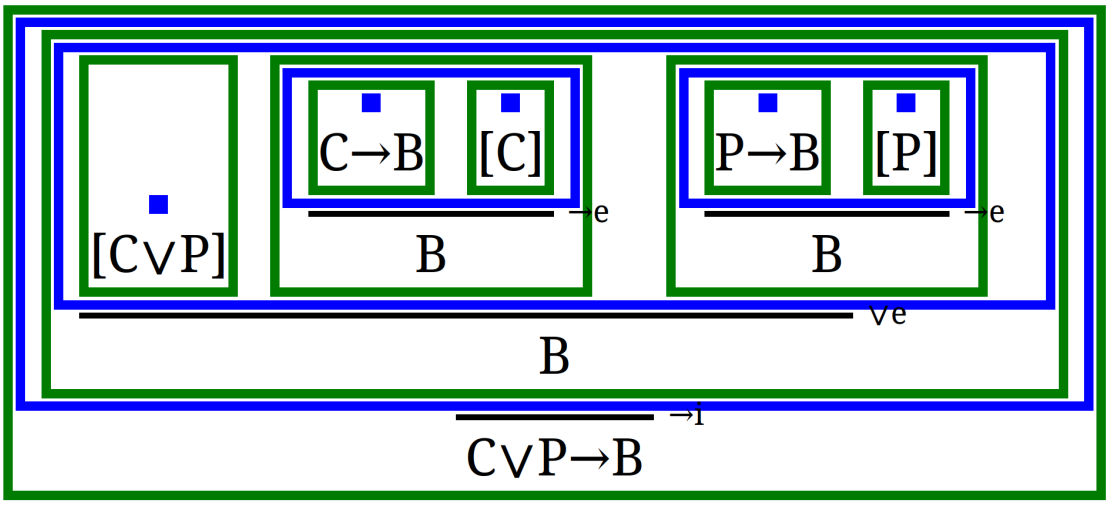
Algoritmo di resa dell'albero

Un punto chiave nella realizzazione del progetto è l'algoritmo di resa dell'albero. Nei progetti precedenti, la gestione e la visualizzazione degli alberi di deduzione hanno spesso presentato sfide significative, influenzando negativamente l'usabilità e la performance dell'applicazione.
L'idea che sta alla base dell'algoritmo è quella di sfruttare al meglio gli strumenti che la piattaforma ci mette a disposizione e far fare il grosso del lavoro a HTML e CSS.

Considerando che la struttura del DOM[4] generato da HTML è essa stessa un albero, notiamo subito che possiamo in qualche modo descrivere il nostro albero di deduzione.

```
<div id="radice" class="radice">
  <div class=" v"> flex
    <p style="white-space: nowrap;">CvP→B</p> event overflow
    <div class=" div_hover_conclusion " style="visibility: collapse;">type: IMPLY</div> overflow
    <div class=" line" style="width: 59.95px; visibility: visible; right: 0px;"></div> overflow
    <div class=" divRule" style="visibility: hidden;">…</div> overflow
    <div class=" h"> flex
      <div class=" v" style="margin-right: 5px;"> flex
        <p style="white-space: nowrap;">B</p> event overflow
        <div class=" div_hover_conclusion " style="visibility: collapse;">div_hover</div> overflow
        <div class=" line" style="width: 239.567px; visibility: visible; right: 27.55px;"></div> overflow
        <div class=" divRule" style="visibility: hidden;">…</div>
        <div class=" h"> flex
          <div class=" v" style="">…</div> flex
          <div class=" v" style="margin-right: 18.1833px;">…</div> flex
          <div class=" v" style="margin-right: 18.1833px;">…</div> flex
```

Pensiamo ai div come dei contenitori: ogni nodo dovrebbe contenere una conclusione, una linea, una regola e una serie di sottoalberi; pertanto, avremo un contenitore verticale **(v)**, che racchiude la conclusione, la linea, la regola, e un contenitore orizzontale **(h)**, che contiene i vari sottoalberi, i quali a loro volta sono contenitori verticali **(v)**. Descrivendo in nodi in questo modo è molto semplice definire l'albero in maniera ricorsiva; inoltre, il riferimento ai contenitori è salvato all'interno di Foglia, in modo da avere sempre a disposizione rappresentazione virtuale della struttura della pagina (idea simile al Virtual DOM di React). Questo consente di ottimizzare l'algoritmo andando ad aggiornare il DOM solo quando necessario.



[4] Document Object Model

I due tipi di contenitori (v e h) sono definiti come due classi CSS. Sono due flexbox: una con direzione verticale e una con direzione orizzontale.

```css
.v{
    margin: 0px;
    display: flex;
    flex-direction: column-reverse;
    align-items: center;
    justify-content: end;
    margin-inline: 5px;
    /*border : solid green 3px;*/
}
```

```css
.h{
    margin: 0px;
    display: flex;
    flex-direction: row;
    justify-content: center;
    margin-inline: 1px;
    /*border: solid blue 3px;*/
}
```

Tutto questo ci da già una buona rappresentazione dell'albero che, essendo composto da flexbox, si adatta in automatico seguendo le regole CSS quando andiamo a modificare un nodo. Ora dobbiamo adattare correttamente dimensione e posizione di linee e regole. Poiché quando modifichiamo un nodo, varia di conseguenza la dimensione del sottoalbero a cui appartiene, dobbiamo effettuare il calcolo ricorsivamente, partendo dal nodo modificato fino alla radice.
Iniziamo dal caso base, ovvero quello in cui il nodo modificato è una foglia dell'albero.

$$\dfrac{[C \vee P] \qquad \dfrac{C \to B \quad [C]}{B} \to e \qquad \dfrac{P \to B \quad [P]}{B} \to e}{B} \vee e$$

$$\dfrac{B}{C \vee P \to B} \to i$$

Abbiamo (C→B) e [C] come premesse e (B) come conclusione. La linea deve partire dal punto più a sinistra dell'elemento contenente (C→B) e finire nel punto più a destra dell'elemento contenente [C]. Inoltre, a essa viene aggiunto del margine sia a sinistra che a destra. La regola viene posizionata a destra della linea. Poiché è impossibile conoscere a priori la dimensioni di questi elementi, a causa di fattori variabili come la scala e la dimensione del font, ci avvaliamo di nuovo degli strumenti offerti dal DOM, che tramite il metodo Element.getBoundingClientRect() ci permette di sapere posizione e dimensioni effettive degli elementi visualizzati a schermo. Con queste informazioni a nostra disposizione possiamo calcolare facilmente come devono essere posizionate la linea e la regola.
Estendendosi al caso più generale, la linea deve partire dal punto più a sinistra della conclusione del primo figlio e finire nel punto più a destra della conclusione dell'ultimo figlio. Nel caso in cui l'elemento rappresentante la conclusione abbia una larghezza maggiore rispetto alla linea calcolata precedentemente, essa viene estesa portando l'inizio nel punto più a sinistra e la fine nel punto più a destra della conclusione.

Ripetendo questo procedimento ricorsivamente fino ad arrivare alla radice dell'albero (utilizzando il metodo RecursiveUpDraw()) otteniamo una corretta visualizzazione dell'albero.

Algoritmo di parsing

L'algoritmo di parsing si occupa di convertire una formula da testo a un oggetto di tipo FORMULA. Per effettuare questa operazione abbiamo bisogno di alcune informazioni, per cui viene definita una tabella che ne tiene traccia per ogni connettivo: il simbolo testuale (∧, ∀, ¬, ...); la corrispettiva classe (AND, FORALL, NOT, …); il tipo di connettivo (binario, binder, unario, …); l'associatività (sinistra o destra); la precedenza (∧ ha una precedenza maggiore rispetto a ∀).

```
export const operators = {

    "∀" : { op:FORALL, precedence:20, ass:associativity.R, type:op_type.BINDER},
    "∃" : { op:EXIST, precedence:20, ass:associativity.R, type:op_type.BINDER},
    "→" : { op:IMPLY, precedence:20, ass:associativity.R, type:op_type.BIN},
    "⇔" : { op:DOUBLEIMPLY, precedence:20, ass:associativity.R, type:op_type.BIN},

    "⋁" : { op:OR, precedence:30, ass:associativity.L, type:op_type.BIN},
    "∧" : { op:AND, precedence:35, ass:associativity.L, type:op_type.BIN},
    "¬" : { op:NOT, precedence:40, ass:associativity.L, type:op_type.UN},
    "⊥" : { op:NOT, precedence:100, ass:associativity.L, type:op_type.BOTTOM},
    "∈" : { op:IN_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "∉" : { op:NOT_IN_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "=" : { op:EQ_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "≠" : { op:NOT_EQ_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "⊆" : { op:SUBE_RPOPERTY, precedence:50, ass:associativity.L, type:op_type.BIN},
    "⋃" : { op:UNION_PROPERTY, precedence:55, ass:associativity.L, type:op_type.BIN},
    "∩" : { op:INTERSECTION_PROPERTY, precedence:60, ass:associativity.L, type:op_type.BIN},

    "<" : { op:LE_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "≤" : { op:LEQ_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    ">" : { op:GE_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},
    "≥" : { op:GEQ_PROPERTY, precedence:45, ass:associativity.L, type:op_type.BIN},

    "+" : { op:SUM_PROPERTY, precedence:55, ass:associativity.L, type:op_type.BIN},
    "-" : { op:MINUS_PROPERTY, precedence:55, ass:associativity.L, type:op_type.BIN},
    "*" : { op:MUL_PROPERTY, precedence:60, ass:associativity.L, type:op_type.BIN},
    "/" : { op:DIVISION_PROPERTY, precedence:60, ass:associativity.L, type:op_type.BIN},
    ":":{ op:CONCAT_PROPERTY,  precedence:80, ass:associativity.R, type:op_type.BIN}
}
```

L'algoritmo procede ricorsivamente partendo dall'operatore con minore precedenza, che sarà quello più "esterno", fino all'operatore con precedenza maggiore, che sarà quello più "interno". Ovviamente, è possibile cambiare il comportamento tramite l'utilizzo delle parentesi.

Andiamo ad analizzare i vari passi dell'algoritmo su un esempio concreto: Parse("A→B∧C→A∨B∧¬C").

Iniziamo con il trovare l'operatore con la precedenza minore che, nel caso sopra esposto è "→". Poiché esso è associativo a destra, prendiamo in considerazione quello più a sinistra. Dato che è un operatore binario andiamo ad analizzare ricorsivamente ciò che c'è a sinistra e a destra. L'oggetto risultante sarà del tipo IMPLY(Parse("A"), Parse("B∧C→A∨B∧¬C")).

Parse(“A”): restituisce VAR(“A”) poiché non ci sono più connettivi da analizzare, viene restituita una variabile.

Parse(“B∧C→A∨B∧¬C”): ha come operatore con precedenza minore “→”, quindi sarà restituito IMPLY(Parse(“B∧C”), Parse(“A∨B∧¬C”)).

Parse(“B∧C”): ha come unico operatore “∧”, quindi sarà restituito AND(Parse(“B”), Parse(“C”)).

Parse(“A∨B∧¬C”): l’operatore “∨” ha precedenza minore rispetto a “∧”, quindi verrà restituito OR(Parse(“A”), Parse(“B∧¬C”)).

Parse(“B∧¬C”): l’operatore “∧” ha precedenza minore rispetto a “¬”, quindi verrà restituito AND(Parse(“B”), Parse(“¬C”)).

Parse(“¬C”): ha come unico operatore “¬”, che è un operatore di tipo unario, quindi verrà restituito NOT(Parse(“C”)).

Alla fine del procedimento ci ritroveremo con l’oggetto:

IMPLY(VAR(A), IMPLY(AND(VAR(B), VAR(C)), OR(VAR(A), AND(B, NOT(C)))))

Possiamo capire più facilmente la gerarchia guardando al grafico nell’inspector del debugging:

Capitolo 7: Conclusioni e sviluppi futuri

L’applicazione si trova attualmente in uno stadio embrionale, ma comprende già tutte le funzionalità necessarie per creare un albero di deduzione. Nonostante sia ancora in fase di sviluppo, l’applicazione offre un’esperienza utente robusta grazie agli assist che facilitano significativamente l’interazione con il sistema. Inoltre, l’applicazione consente a docenti e tutor di assegnare degli esercizi e visualizzare le soluzioni degli studenti, facilitando il monitoraggio dei loro progressi.

Uno degli aspetti più rilevanti di questa versione è l’algoritmo di resa dell’albero, che ha risolto le problematiche riscontrate nei progetti precedenti. Questo algoritmo assicura una visualizzazione chiara e navigabile degli alberi di deduzione, migliorando l’usabilità e le prestazioni dell’applicazione.

Inoltre, il backend è stato progettato con una serie di controlli rigorosi per prevenire comportamenti scorretti, garantendo così un funzionamento stabile e sicuro del sistema. Grazie a questi miglioramenti, l’applicazione rappresenta un passo avanti significativo rispetto ai progetti precedenti e pone solide basi per futuri sviluppi.

Riguardo l’intero progetto, è stata stimata una lunghezza di 6.125 linee di codice per un lavoro complessivo di 3 mesi/uomo. Ulteriori dettagli sono illustrati nell’appendice "Note sul codice prodotto".

Lo strumento non è ancora stato sottoposto a una fase di validazione pratica con gli studenti. La validazione rappresenta un passo cruciale per garantire che il sistema soddisfi le reali esigenze degli utenti finali e raggiunga gli obiettivi prefissati. Il sistema verrà introdotto e testato nel contesto delle lezioni di laboratorio. Gli studenti saranno invitati a utilizzare il sistema per svolgere esercizi e inviare soluzioni. Durante queste sessioni, verranno raccolti feedback dettagliati dagli studenti e dai docenti, al fine di valutare la facilità d’uso, l’efficacia degli strumenti e la chiarezza delle funzionalità offerte. Inoltre, gli studenti avranno l’opportunità di esplorare e utilizzare il sistema in modo autonomo. Questo approccio consentirà di osservare come il sistema viene utilizzato in scenari reali e di raccogliere ulteriori feedback per identificare aree di miglioramento. Gli studenti saranno incoraggiati a fornire osservazioni e suggerimenti tramite un modulo di feedback che aiuterà a perfezionare il sistema.

Sviluppi futuri

Attualmente, è possibile creare alberi di deduzione naturale per la logica proposizionale e la logica del prim'ordine. Un possibile sviluppo futuro potrebbe consistere nell'implementazione di altri tipi di logica, come la logica del secondo ordine.

Inoltre, si potrebbe espandere il set di regole disponibili, aggiungendone alcune che permettano di ottenere alberi di deduzione più compatti. Ad esempio, l'introduzione di regole dirette per applicare gli assiomi della teoria degli insiemi, che attualmente richiedono vari passaggi per essere utilizzati, consentirebbe di creare alberi di grandi dimensioni in minor tempo, migliorando significativamente la leggibilità.

Un ulteriore sviluppo potrebbe consistere nell'aggiungere livelli aggiuntivi di assistenza automatica. Ad esempio, implementare un algoritmo che cerchi di combinare più ipotesi per compilare ulteriori passaggi automaticamente, o un algoritmo che risolva completamente dei sottoalberi mentre si lavora, evitando la necessità di dimostrare manualmente casi banali. Questi miglioramenti renderebbero il processo di costruzione degli alberi di deduzione ancora più efficiente e user-friendly.

Un'altra idea interessante sarebbe quella di implementare un algoritmo per convertire gli alberi di deduzione naturale in dimostrazioni tradizionali, ad esempio nel linguaggio controllato di Matita, e viceversa. Questa funzionalità permetterebbe di confrontare i due tipi di dimostrazione, avendo un grande impatto a livello didattico, poiché offrirebbe agli studenti la possibilità di apprendere e confrontare diversi approcci alla dimostrazione.

Bibliografia

Studi

Berardinelli, Danilo (2014) *Implementazione del backend di una app Android per la didattica della deduzione naturale*. [Laurea], Università di Bologna, Corso di Studio in Informatica [L-DM270].

Sacerdoti Coen, Claudio (2000). *Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata*. [Laurea], Università di Bologna, Corso di Studio in Informatica.

Sacerdoti Coen, Claudio, & Tassi, Enrico (2009). Natural deduction environment for Matita.

Mezza, Stefano (2014) *Implementazione del frontend di una app Android per la didattica della deduzione naturale*. [Laurea], Università di Bologna, Corso di Studio in Informatica [L-DM270].

Slide presentate nell'anno accademico 2023/2024 dell'insegnamento 93283 - Logica per L'Informatica.

Strumenti

"Astro Docs", https://docs.astro.build

"MDN Web Docs", https://developer.mozilla.org

"Stack Overflow", https://stackoverflow.com

"Bootstrap Docs", https://getbootstrap.com/docs

"Introduction to JSON Web Tokens", https://jwt.io/introduction

"Matita Documentation", http://matita.cs.unibo.it/documentation.shtml

"MongoDB Documentation", https://www.mongodb.com/docs

"TypeScript Documentation", https://www.typescriptlang.org/docs

"Node.js Documentation", https://nodejs.org/docs/latest/api

"W3Schools Online Web Tutorials", https://www.w3schools.com

Appendice: Note sul codice prodotto

In questa appendice analizziamo alcune metriche sul codice prodotto e sugli strumenti utilizzati per lo sviluppo.
L'ambiente di sviluppo scelto è stato Visual Studio Code che, grazie al suo ecosistema di estensioni e alla sua leggerezza, ha semplificato lo sviluppo dell'app. Utilizzando una delle estensioni possiamo calcolare il numero di righe di codice (prendiamo in considerazione solo i file rilevanti). Questo è il risultato del conteggio:

Total: 65 files, 6125 codes, 72 comments, 1874 blanks, all 8071 lines
Typescript: 89.9%, Astro 9.8%, Other: 0.3%

È stato utilizzato un paradigma di programmazione a oggetti, in modo da evitare codice ripetuto e facilitare l'estensione delle funzionalità. Le classi principali includono "Foglia" , FORMULA (da cui derivano 29 classi) e REGOLA (da cui derivano 19 classi).
Il progetto include 7 pagine accessibili dagli utenti e una struttura di API REST composta da 19 chiamate.
Il tempo stimato per la realizzazione del progetto è di 3 mesi/uomo (480 ore/uomo).

Durante lo sviluppo l'ambiente di test principale è stato Firefox, principalmente perché offre molti strumenti di analisi per le pagine web. Ciononostante, sono stati effettuati anche alcuni test anche su altri browser come Chrome ed Edge.

Per eseguire il progetto è necessario aver installato Node.js (versione 8.2.5 o superiore), MongoDB (versione 6.5.0 o superiore) e npm (versione 9.0.1 o superiore).
Il progetto può essere anche racchiuso in un'istanza di docker per evitare problemi di dipendenze e per facilitare la scalabilità.