UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Soluzione Intelligente del Cubo di Rubik 2x2x2

Progetto di Ingegneria della Conoscenza

Studente:
Luca Ardito
Matricola: 777818
l.ardito12@studenti.uniba.it

Anno Accademico:
2024-2025

Docente:
Nicola Fanizzi

Repository GitHub:
`https://github.com/CrickCrock28/Prolog-Python-Rubik-2x2x2`

Indice

1 Sommario

Il progetto si propone di sviluppare un sistema per la risoluzione del Cubo di Rubik 2x2x2, utilizzando un approccio basato su conoscenza. L'applicazione integra tecniche di rappresentazione della conoscenza, ragionamento automatico e interazione grafica, sfruttando linguaggi complementari come Python.

La rappresentazione dello stato del cubo è implementata in Prolog utilizzando clausole di Horn, che permettono di definire vincoli, regole e trasformazioni in modo formale e dichiarativo. La risoluzione del problema è modellata come una ricerca nello spazio di stati, applicando un algoritmo di ricerca bidirezionale con profondità limitata per trovare una sequenza ottimale di mosse. Inoltre, il sistema include funzionalità per la validazione delle configurazioni, la generazione di stati casuali e il calcolo della soluzione.

L'interfaccia grafica, sviluppata in Python tramite la libreria `tkinter`, consente all'utente di interagire visivamente con il cubo, modificandone lo stato, mescolandolo e calcolando la sequenza di mosse per riportarlo alla configurazione risolta. Questa integrazione tra logica dichiarativa (Prolog) e programmazione imperativa (Python) rappresenta un esempio concreto di utilizzo diversi di strumenti per risolvere un problema noto per la sua complessità computazionale.

In relazione agli argomenti del corso, il progetto affronta i seguenti temi chiave:

- **Ricerca di soluzioni e spazi di stati**: il Cubo di Rubik è stato modellato come uno spazio di stati, esplorato tramite tecniche di ricerca bidirezionale. Le mosse sono definite come trasformazioni applicabili agli stati del cubo.
- **Rappresentazione e ragionamento relazionale**: lo stato del cubo è rappresentato come una stringa codificata in Prolog, con regole di trasformazione formalizzate tramite clausole di Horn.
- **Ragionamento automatico**: utilizzo di Prolog per implementare algoritmi di risoluzione, validazione degli stati e generazione di configurazioni casuali, sfruttando la potenza dell'inferenza logica.

Il progetto dimostra come i concetti teorici affrontati nel corso, tra cui la rappresentazione della conoscenza e il ragionamento automatico, possano essere applicati per risolvere un problema pratico e ben definito. La combinazione di tecniche e strumenti complementari evidenzia come questo approccio possa essere utilizzato per affrontare problemi complessi in modo efficiente e modulare.

2 Notazione del Cubo di Rubik 2x2

2.1 Sommario

Per comprendere pienamente il funzionamento del sistema sviluppato, è necessario introdurre la notazione utilizzata per descrivere le mosse e le configurazioni del Cubo di Rubik 2x2x2. Questa sezione presenta le convenzioni adottate, inclusa la notazione standard delle mosse e il modo in cui sono rappresentate nello stato del cubo.

2.2 Notazione delle mosse

Le mosse del Cubo di Rubik 2x2x2 sono indicate utilizzando una notazione standard che identifica le facce del cubo con le seguenti lettere:

- **U** (Up): faccia superiore.
- **D** (Down): faccia inferiore.
- **F** (Front): faccia frontale.
- **B** (Back): faccia posteriore.
- **L** (Left): faccia sinistra.
- **R** (Right): faccia destra.

Ogni sequenza di mosse è composta da una serie di lettere, ciascuna delle quali rappresenta una rotazione di una faccia del cubo. Le mosse possono essere di due tipi:

- **Lettera semplice** (es. `R`): Una rotazione di 90° in senso orario della faccia corrispondente.
- **Lettera seguita da un apostrofo** (es. `R'`): Una rotazione di 90° in senso antiorario.

Ad esempio, la sequenza `R U R' U'` rappresenta:

- Una rotazione di 90° in senso orario della faccia destra.
- Una rotazione di 90° in senso orario della faccia superiore.
- Una rotazione di 90° in senso antiorario della faccia destra.
- Una rotazione di 90° in senso antiorario della faccia superiore.

3 Ricerca di soluzioni e spazi di stati

3.1 Sommario

Il problema del Cubo di Rubik 2x2x2 è stato modellato come uno spazio di stati, in cui ogni stato rappresenta una configurazione del cubo, mentre le mosse rappresentano le transizioni tra stati. La risoluzione del cubo consiste nel trovare una sequenza di mosse che porti il cubo da una configurazione iniziale, anche casuale, allo stato risolto. Per affrontare questo problema, è stato implementato un algoritmo di ricerca su grafo in Prolog, con alcune ottimizzazioni specifiche per ridurre la complessità dello spazio di ricerca, tra cui la fissazione di un angolo come riferimento e l'uso della ricerca bidirezionale.

3.2 Modellazione del problema

Riduzione dello spazio di stati tramite fissazione di un angolo Il Cubo di Rubik 2x2x2 possiede 8 angoli, ciascuno dei quali può essere orientato in 3 modi. Teoricamente, questo comporterebbe uno spazio di configurazioni pari a:

$$N_{\text{configurazioni}} = 8! \times 3^8 = 88.179.840$$

Tuttavia, a causa di vincoli strutturali del cubo:

- La somma degli orientamenti degli angoli deve essere un multiplo di 3.
- Il numero totale di permutazioni deve essere pari.

Questi vincoli riducono le configurazioni possibili a:

$$N_{\text{effettive}} = \frac{8! \times 3^7}{2} = 3.674.160$$

Per ridurre ulteriormente la complessità dello spazio di ricerca, nel progetto è stato deciso di fissare un angolo come riferimento. La fissazione di un angolo comporta due effetti principali:

- La posizione e l'orientamento di quell'angolo restano invariati, eliminando la necessità di considerare configurazioni equivalenti ottenute tramite rotazioni dell'intero cubo.
- Il numero di mosse possibili in ogni stato si riduce da 12 a 6, in quanto le mosse che coinvolgerebbero l'angolo fisso non sono considerate.

Questa scelta riduce significativamente lo spazio di ricerca. Il nuovo numero di configurazioni diventa:

$$N_{\text{ridotte}} = \frac{7! \times 3^7}{2} = 367.416$$

Inoltre, la riduzione delle mosse possibili per stato dimezza il branching factor, portandolo da $b = 12$ a $b = 6$. Questo dimezzamento ha un impatto significativo sulla complessità della ricerca, che segue una crescita esponenziale con il branching factor b.

Scelta della ricerca bidirezionale L'algoritmo di risoluzione utilizza una strategia di ricerca bidirezionale. In questo approccio, lo spazio di stati viene esplorato simultaneamente in due direzioni:

- A partire dalla configurazione iniziale (`scrambled`).
- A partire dalla configurazione finale (`solved`).

La ricerca bidirezionale riduce drasticamente il numero di stati esplorati rispetto a una ricerca unidirezionale. Supponendo un branching factor $b = 6$ e una profondità massima $d = 14$, il numero di stati esplorati per una ricerca unidirezionale sarebbe:

$$N_{\text{unidirezionale}} = b^d = 6^{14} \approx 7.529.536$$

Con la ricerca bidirezionale, ogni lato esplora fino a una profondità massima di $d/2 = 7$. Il numero totale di stati esplorati diventa:

$$N_{\text{bidirezionale}} = 2 \times b^{(d/2)} = 2 \times 6^7 = 2 \times 279.936 = 559.872$$

Questo approccio riduce il numero di stati esplorati di un fattore pari a:

$$\frac{N_{\text{unidirezionale}}}{N_{\text{bidirezionale}}} \approx 13,44$$

La scelta della ricerca bidirezionale permette quindi di risolvere il cubo in modo più efficiente, riducendo significativamente i tempi computazionali e la memoria richiesta.

Limite di profondità fissato a 7 Nel caso del Cubo di Rubik 2x2x2, il numero massimo di mosse necessarie per risolvere qualsiasi configurazione è noto essere pari a 14 (vedi `https://en.wikipedia.org/wiki/Pocket_Cube`). Dato che la ricerca è bidirezionale, ogni lato esplora fino a una profondità massima pari a:

$$\frac{14}{2} = 7$$

Questo limite garantisce che le due ricerche si incontrino sempre, coprendo l'intero spazio di stati senza esplorare inutilmente stati più profondi. Inoltre, il limite di profondità è sufficiente per risolvere il cubo in tutte le configurazioni possibili, preservando l'efficienza computazionale.

3.3 Algoritmi di ricerca

L'algoritmo principale implementato è una ricerca su grafo bidirezionale con profondità limitata. Questo approccio è stato scelto per bilanciare il costo computazionale e la completezza della soluzione. Di seguito si riassumono le caratteristiche principali:

- **Costruzione dello spazio di stati**: Lo spazio di stati viene costruito a partire da due origini: lo stato risolto e lo stato iniziale. Gli stati generati sono memorizzati tramite il predicato `sphere/4`, che associa a ogni stato il livello di profondità e la sequenza di mosse che lo ha generato.
- **Ricerca bidirezionale**: Per ridurre la complessità, viene effettuata una ricerca sia a partire dallo stato iniziale (scrambled) sia dallo stato risolto (solved). Questo approccio permette di fermare la ricerca non appena si trova uno stato comune tra i due alberi di ricerca.
- **Ottimizzazione tramite limite di profondità**: La profondità massima di 7 è stata scelta per garantire che il cubo sia risolto in tutte le configurazioni possibili, minimizzando al contempo gli stati esplorati.

3.4 Integrazione Python-Prolog

L'algoritmo di ricerca è integrato con l'interfaccia Python tramite la libreria `pyswip`, che consente di inviare query a Prolog e ricevere i risultati. Il modulo `CubeLogic.py` si occupa di:

- Convertire la configurazione del cubo in una stringa compatibile con Prolog.
- Eseguire la query `solve/2` per calcolare la sequenza di mosse necessaria per risolvere il cubo.
- Interpretare e visualizzare il risultato tramite l'interfaccia grafica.

3.5 Limitazioni e sviluppi futuri

L'approccio attuale non sfrutta euristiche avanzate per ottimizzare ulteriormente la ricerca. Sviluppi futuri potrebbero includere:

- Implementazione di algoritmi di ricerca informata come A*.
- Estensione del sistema per supportare cubi di dimensioni maggiori.
- Integrazione di tecniche di apprendimento per suggerire mosse ottimali basate su dati storici.

4 Rappresentazione della conoscenza

4.1 Sommario

Nel progetto, la rappresentazione della conoscenza è stata implementata utilizzando il linguaggio logico Prolog, con una struttura basata su clausole di Horn. Lo stato del Cubo di Rubik 2x2x2 è rappresentato come una stringa codificata, che descrive la disposizione e l'orientamento di 7 angoli, in quanto l'ottavo è stato fissato come detto in precedenza.

La scelta di utilizzare Prolog è motivata dalla sua capacità di rappresentare problemi in modo dichiarativo, utilizzando regole e predicati logici per definire vincoli, trasformazioni e inferenze.

4.2 Modellazione dello stato del cubo

Il Cubo di Rubik 2x2x2 è rappresentato come una stringa che descrive i colori di 7 angoli, ciascuno composto da 3 colori. L'ottavo angolo è considerato fisso e non è incluso nella rappresentazione dello stato. Ad esempio, lo stato risolto è codificato come:

```
'wgr wrb wbo ygo yob ybr yrg'
```

Ogni tripla rappresenta un angolo, con i colori disposti in un ordine specifico per indicare l'orientamento. Ad esempio:

- `wgr` rappresenta un angolo con i colori bianco, verde e rosso.
- `ygo` rappresenta un angolo con i colori giallo, verde e arancione.

L'ordine dei pezzi nella stringa corrisponde a:

```
UFR, URB, UBL, DFL, DLB, DBR, DRF
```

Quindi, il primo angolo sarà quello all'intersezione tra la faccia superiore, la faccia frontale e la faccia destra, e così via.

La scelta di escludere un angolo dalla rappresentazione è motivata dal fatto che fissare un angolo riduce lo spazio di stati, semplificando al contempo i calcoli per le trasformazioni. Poiché l'ottavo angolo è vincolato dagli altri sette (sia nella posizione che nell'orientamento), non è necessario rappresentarlo esplicitamente.

Il pezzo fissato è quello in posizione `UFL`, con colori `wgo`. Quindi, nella rappresentazione:

- Il colore bianco è associato alla faccia superiore.
- Il colore verde è associato alla faccia frontale.
- Il colore arancione è associato alla faccia sinistra.

Di conseguenza:

- Il colore giallo è associato alla faccia inferiore.
- Il colore rosso è associato alla faccia destra.
- Il colore blu è associato alla faccia posteriore.

4.3 Predicati di base per la rappresentazione

Per rappresentare e manipolare lo stato del cubo, sono stati definiti i seguenti predicati principali:

Fatti

- `state_zero(State)`: Definisce lo stato risolto del cubo.

```
state_zero('wgr wrb wbo ygo yob ybr yrg').
```

- `move(Move)`: Specifica le mosse valide del cubo.

```
move(down_clockwise).
move(down_counterclockwise).
move(right_clockwise).
move(right_counterclockwise).
move(back_clockwise).
move(back_counterclockwise).
```

- `reversal(Move, Reversal)`: Definisce la relazione tra una mossa e la sua controparte inversa.

```
reversal(down_clockwise, down_counterclockwise).
reversal(right_clockwise, right_counterclockwise).
...
```

Regole Le regole implementano la logica per trasformare lo stato del cubo, validare configurazioni e trovare soluzioni. Esempi includono:

- **Trasformazioni delle mosse** (`move_transform/3`): Trasforma uno stato applicando una mossa.

```
move_transform(In_state, Move, Out_state).
```

- **Validazione dello stato** (`valid_state/1`): Verifica che lo stato rispetti i vincoli del cubo.

```
valid_state(State).
```

- **Controllo dei colori** (`check_color_count/3`): Conta e verifica il numero di colori nello stato.

  ```
  check_color_count(State, Color, Count).
  ```

- **Validazione delle permutazioni** (`valid_permutation/1`): Garantisce che lo stato contenga angoli validi.
- **Applicazione delle mosse** (`apply_path/2`): Applica una sequenza di mosse a uno stato.
- **Inversione delle mosse** (`reverse_all_moves/2`): Calcola le mosse inverse di una sequenza.
- **Concatenazione delle liste** (`concat/3`): Concatena due liste di mosse.

Motore di inferenza Prolog agisce da motore di inferenza, applicando le regole per rispondere alle query, ad esempio:

```
?- solve('ygo yob ...', Path).
?- valid_state('wgr wrb ...').
?- shuffle(RandomState).
```

4.4 Validazione dello stato

La validazione dello stato del cubo è un aspetto cruciale per garantire che le configurazioni generate siano effettivamente risolvibili. I vincoli di validazione includono:

- **Vincoli di colore**: Ogni colore (bianco, giallo, rosso, arancione, verde e blu) deve apparire un numero fisso di volte:

  ```
  bianco = 3,  giallo = 4,  rosso = 4,

  arancione = 3,  verde = 3,  blu = 4
  ```

  Questi vincoli sono verificati utilizzando il predicato `check_color_count/3`. I colori bianco, arancione e verde appaiono 3 volte poiché l'angolo fisso ha proprio con questi colori e non è incluso nella rappresentazione dello stato.

- **Permutazione valida degli angoli**: Le configurazioni devono rispettare le regole del Cubo di Rubik, che limitano le permutazioni e gli orientamenti degli angoli. Questo è garantito dal predicato `valid_permutation/1`, che verifica che ogni pezzo corrisponda a uno degli angoli validi dello stato risolto.

- **Somma degli orientamenti**: La somma degli orientamenti dei 7 angoli rappresentati, insieme all'angolo fisso, deve essere un multiplo di 3. Questo vincolo è rispettato implicitamente dalla scelta di fissare un angolo e dalla rappresentazione logica.

4.5 Trasformazioni e mosse

Le mosse del cubo (`down_clockwise`, `right_counterclockwise`, ecc...) sono rappresentate come trasformazioni logiche che agiscono sullo stato. Ogni mossa è definita tramite il predicato `move_transform/3`, che descrive come lo stato viene modificato dall'applicazione di una mossa.

Le mosse inverse (come `right_counterclockwise`) sono definite utilizzando il predicato `reversal/2`, che associa ogni mossa alla sua controparte inversa.

Le trasformazioni sono state implementate nel file `cube_utils.pl`, dove ogni mossa modifica specificamente gli angoli coinvolti, mantenendo inalterato l'angolo fisso.

4.6 Integrazione con la risoluzione

La rappresentazione della conoscenza si integra strettamente con gli algoritmi di ricerca. Lo stato iniziale e quello finale sono rappresentati come stringhe compatibili con i predicati logici. Le trasformazioni sono applicate per esplorare lo spazio di stati, mentre i vincoli garantiscono che ogni stato generato sia valido.

Questa rappresentazione logica consente di utilizzare Prolog per inferire direttamente la sequenza di mosse necessaria per risolvere il cubo, sfruttando la potenza del motore di inferenza logica.

4.7 Vantaggi e limiti della rappresentazione

Vantaggi

- La rappresentazione logica è dichiarativa, compatta e facilmente estendibile ad altri problemi simili.
- L'uso di clausole di Horn permette di definire in modo esplicito i vincoli e le trasformazioni dello stato.
- La fissazione di un angolo riduce la complessità, semplificando la validazione e la generazione di stati.

Limiti

- La rappresentazione si basa su stringhe codificate, che richiedono operazioni di parsing e conversione per interagire con altri moduli.
- Il sistema attuale non supporta configurazioni di cubi di dimensioni maggiori, che richiederebbero una rappresentazione più complessa.

4.8 Sviluppi futuri

Un possibile sviluppo futuro consiste nell'estendere la rappresentazione per includere cubi di dimensioni maggiori, integrando ontologie o knowledge graph per rappresentare le configurazioni in modo più flessibile. Inoltre, l'uso di modelli probabilistici potrebbe consentire di suggerire mosse ottimali basate su dati storici o euristiche.

5 Ragionamento automatico

5.1 Sommario

Il ragionamento automatico è un elemento centrale del progetto, poiché permette di validare configurazioni, risolvere il Cubo di Rubik 2x2x2 e generare stati casuali. Prolog, utilizzato come motore logico, implementa i predicati necessari per rappresentare e trasformare gli stati del cubo, applicando regole logiche e vincoli. Questo approccio sfrutta la potenza del ragionamento dichiarativo per gestire la complessità del problema.

5.2 Validazione e vincoli logici

La validazione di uno stato è fondamentale per garantire che le configurazioni generate o ricevute come input siano effettivamente risolvibili. Questo è ottenuto tramite una serie di regole logiche definite in Prolog:

- **Vincoli sui colori**: Ogni colore (bianco, giallo, rosso, arancione, verde e blu) deve apparire un numero specifico di volte, come verificato dal predicato `check_color_count/3`.
- **Permutazioni valide**: Gli angoli rappresentati nello stato devono corrispondere a configurazioni fisicamente possibili del cubo. Questo è garantito dal predicato `valid_permutation/1`.
- **Orientamenti degli angoli**: La somma degli orientamenti dei 7 angoli rappresentati, combinata con l'angolo fisso, deve essere un multiplo di 3. Questo vincolo è rispettato implicitamente grazie alla rappresentazione logica.

La validazione è integrata nell'algoritmo di risoluzione, che rigetta configurazioni non valide e fornisce feedback all'utente tramite l'interfaccia Python.

5.3 Risoluzione del cubo

La risoluzione del Cubo di Rubik 2x2x2 è modellata come un problema di ricerca nello spazio di stati, utilizzando il predicato `solve/2` in Prolog. L'algoritmo applica una ricerca bidirezionale con limite di profondità, come descritto nella sezione precedente. La logica di risoluzione è strutturata nei seguenti passi:

1. **Costruzione dello spazio di stati**: Gli stati vengono generati a partire dalla configurazione iniziale (`scrambled`) e da quella finale (`solved`), utilizzando il predicato `build_sphere/4`.
2. **Espansione degli stati**: Ogni stato esplorato genera nuovi stati applicando le mosse definite nel predicato `move_transform/3`.
3. **Trovare un percorso**: Quando uno stato comune è trovato tra lo spazio di ricerca iniziale e quello finale, il predicato `find_path/1` calcola la sequenza di mosse necessaria per risolvere il cubo.

L'algoritmo restituisce una sequenza di mosse ottimale (minimo numero di mosse) per portare il cubo dallo stato iniziale a quello risolto. Questo è possibile grazie alla ricerca bidirezionale, che riduce significativamente il numero di stati da esplorare.

5.4 Generazione di stati casuali

Per testare il sistema e offrire funzionalità all'utente, è stato implementato un algoritmo per generare stati casuali del cubo. Questo è ottenuto tramite il predicato `shuffle/1`, che:

- Parte dallo stato risolto (`state_zero/1`).
- Applica 30 mosse casuali con (`move_transform/3`), garantendo che lo stato generato sia sempre valido.

La generazione casuale è utile sia per simulare configurazioni realistiche, sia per fornire all'utente un punto di partenza casuale per testare il sistema di risoluzione.

5.5 Integrazione con Python

Il modulo Python `CubeLogic.py` comunica con Prolog per eseguire il ragionamento automatico. Le funzionalità principali includono:

- **Validazione degli stati**: Tramite il predicato `valid_state/1`, il sistema verifica che lo stato inserito dall'utente sia corretto.
- **Calcolo della soluzione**: La funzione `solve/2` in Prolog restituisce la sequenza di mosse per risolvere il cubo. Questa sequenza è interpretata e mostrata all'utente tramite l'interfaccia grafica.
- **Generazione casuale**: La funzione `shuffle/1` in Prolog genera stati casuali, che sono inviati e rappresentati nell'interfaccia Python.

5.6 Vantaggi e limiti del ragionamento automatico

Vantaggi

- Il ragionamento dichiarativo di Prolog è ideale per rappresentare i vincoli e le regole del cubo.
- L'inferenza logica permette di implementare algoritmi di ricerca complessi in modo semplice e leggibile.
- L'integrazione Python-Prolog offre un ambiente flessibile e potente per combinare logica dichiarativa e programmazione imperativa.

Limiti

- L'attuale implementazione non sfrutta algoritmi di ricerca informata, che potrebbero ulteriormente ottimizzare la risoluzione.
- La generazione casuale si basa su sequenze di mosse casuali e non utilizza una distribuzione uniforme sugli stati del cubo.
- Il sistema è specifico per il Cubo di Rubik 2x2x2 e non è facilmente estensibile a cubi più grandi senza modifiche significative.

5.7 Sviluppi futuri

Tra i possibili sviluppi futuri si includono:

- L'implementazione di algoritmi di ricerca informata, come A*, per ridurre ulteriormente il numero di stati esplorati.
- Estendere il sistema per supportare cubi di dimensioni maggiori, con una rappresentazione più complessa e un motore logico più avanzato.
- Migliorare l'algoritmo di generazione casuale per garantire una distribuzione uniforme degli stati del cubo.

6 Strumenti utilizzati

6.1 Sommario

Il progetto combina diversi strumenti e tecnologie per implementare un sistema di risoluzione del Cubo di Rubik 2x2x2. Questa sezione descrive i principali strumenti utilizzati e il loro ruolo nel progetto.

6.2 Python

Python è stato scelto per gestire la logica ad alto livello, l'interfaccia grafica e l'integrazione con Prolog. In particolare:

- **Interfaccia grafica**: La libreria `tkinter` è stata utilizzata per sviluppare un'applicazione grafica che consente agli utenti di interagire con il cubo, visualizzare la configurazione attuale e ottenere la soluzione calcolata. `tkinter` è semplice da utilizzare, altamente personalizzabile e integrato nativamente in Python.
- **Integrazione con Prolog**: La libreria `pyswip` è stata utilizzata per interfacciarsi con Prolog, permettendo di inviare query, ricevere risultati e utilizzare la logica dichiarativa direttamente da Python.
- **Gestione dello stato del cubo**: Il modulo `CubeLogic` in Python traduce la configurazione del cubo in una stringa compatibile con i predicati Prolog, gestendo la comunicazione tra l'interfaccia grafica e il motore logico.

6.3 Prolog

Prolog è stato utilizzato come motore logico per rappresentare la conoscenza e implementare gli algoritmi di risoluzione. Le sue caratteristiche principali sono:

- **Rappresentazione dello stato**: Lo stato del cubo è descritto da una stringa codificata, manipolata tramite clausole di Horn per rappresentare le regole e i vincoli del cubo.
- **Algoritmi di ricerca**: L'intero spazio di stati è esplorato utilizzando predicati logici, come `solve/2`, che implementano una ricerca bidirezionale con profondità limitata.
- **Validazione**: I predicati `valid_state/1` e `check_color_count/3` verificano che uno stato sia valido, rispettando i vincoli strutturali del Cubo di Rubik.
- **Generazione casuale**: Il predicato `shuffle/1` genera configurazioni casuali applicando sequenze di mosse casuali allo stato risolto.

Prolog è stato scelto per la sua capacità di rappresentare problemi in modo dichiarativo, semplificando l'implementazione di vincoli e algoritmi complessi.

6.4 Librerie e framework principali

- `tkinter` **(Python)**: Utilizzata per creare l'interfaccia grafica, permettendo agli utenti di interagire con il cubo in modo visivo.
- `pyswip` **(Python)**: Libreria che consente di integrare Prolog con Python, facilitando lo scambio di informazioni tra i due ambienti.
- **Prolog (SWI-Prolog)**: Motore logico utilizzato per implementare il ragionamento dichiarativo.

6.5 Strumenti di sviluppo e versionamento

- **Editor di testo/IDE**: Visual Studio Code è stato utilizzato per lo sviluppo del progetto, grazie alla sua compatibilità con Python e Prolog, e al supporto per il debug.
- **Sistema di versionamento**: GitHub è stato utilizzato per il controllo delle versioni. Il repository contiene tutto il codice sorgente e la documentazione.

6.6 Vantaggi degli strumenti scelti

- La combinazione di Python e Prolog unisce la flessibilità e la semplicità della programmazione imperativa con la potenza della logica dichiarativa.
- La libreria `tkinter` offre un'interfaccia grafica intuitiva e facilmente estensibile.
- L'integrazione con Prolog tramite `pyswip` permette di sfruttare appieno le capacità di inferenza logica senza sacrificare la versatilità di Python.

6.7 Limiti degli strumenti utilizzati

- `pyswip`, pur essendo potente, può risultare meno documentata rispetto ad altre librerie, rendendo più complessa l'integrazione.
- SWI-Prolog è altamente efficiente per problemi logici, ma la rappresentazione dello stato come stringa richiede parsing e conversioni aggiuntive.
- `tkinter` è funzionale per interfacce grafiche semplici, ma non offre la stessa flessibilità o modernità di framework più avanzati (come PyQt o Tkinter avanzato con temi personalizzati).

6.8 Sviluppi futuri

Tra i possibili sviluppi futuri:

- Migliorare l'interfaccia grafica utilizzando librerie più avanzate come `PyQt` o `Kivy`.

- Integrare altri motori logici o librerie IA per problemi più complessi.

7 Scelte di progetto

7.1 Sommario

Nel corso dello sviluppo del progetto sono state prese diverse decisioni progettuali per garantire un bilanciamento tra semplicità implementativa, efficienza computazionale e chiarezza della rappresentazione. Questa sezione illustra le principali scelte effettuate, con le relative motivazioni.

7.2 Rappresentazione dello stato del cubo

La rappresentazione dello stato del cubo come una stringa codificata di 7 angoli, anziché 8, è stata una decisione chiave del progetto. Fissare un angolo come riferimento ha permesso di ridurre significativamente lo spazio di stati, eliminando la necessità di considerare configurazioni equivalenti ottenute tramite la rotazione dell'intero cubo. Questa scelta:

- Riduce il numero di permutazioni possibili da $8! \cdot 3^7$ a $7! \cdot 3^7$.
- Semplifica la validazione dello stato, dato che l'angolo fisso vincola implicitamente gli altri angoli.
- Dimezza il branching factor da 12 a 6, rendendo la ricerca nello spazio di stati molto più efficiente.

7.3 Algoritmo di risoluzione

È stata adottata una strategia di ricerca bidirezionale con limite di profondità per risolvere il cubo. Le motivazioni di questa scelta includono:

- **Efficienza computazionale**: La ricerca bidirezionale esplora simultaneamente lo spazio di stati a partire dalla configurazione iniziale (`scrambled`) e dallo stato risolto (`solved`), riducendo il numero di stati esplorati da b^d a $2 \cdot b^{d/2}$.
- **Limite di profondità**: È stato fissato a 7 poiché il numero massimo di mosse per risolvere il cubo è 14. Il limite di profondità garantisce che la ricerca si incontri sempre, esplorando solo lo spazio necessario.
- **Generazione di stati comuni**: Il predicato `find_path/1` consente di identificare uno stato condiviso tra le due ricerche, calcolando la sequenza di mosse ottimale.

7.4 Integrazione Python-Prolog

Per combinare la flessibilità di Python con la potenza logica di Prolog, è stata utilizzata la libreria `pyswip`. Le motivazioni principali di questa scelta includono:

- **Separazione delle responsabilità**: Python gestisce l'interfaccia grafica e l'integrazione, mentre Prolog è dedicato alla logica dichiarativa e agli algoritmi di risoluzione.
- **Facilità di sviluppo**: `pyswip` consente di inviare query Prolog direttamente da Python, riducendo la complessità dell'integrazione.
- **Modularità**: La comunicazione tra Python e Prolog si basa su una rappresentazione stringa dello stato del cubo, rendendo il sistema altamente modulare e facile da estendere.

7.5 Interfaccia grafica

L'interfaccia grafica è stata sviluppata utilizzando la libreria `tkinter`, scelta per la sua semplicità e la sua integrazione nativa con Python. Le principali decisioni relative alla GUI includono:

- **Visualizzazione dello stato del cubo**: Ogni faccia del cubo è rappresentata come un insieme di pulsanti colorati, facilmente modificabili dall'utente.
- **Palette di colori**: Una selezione di colori è stata aggiunta per permettere all'utente di personalizzare lo stato del cubo.
- **Pulsanti per azioni principali**: Sono stati implementati pulsanti per resettare il cubo, mescolarlo o calcolare la soluzione.

7.6 Gestione delle trasformazioni

Le trasformazioni del cubo (mosse) sono state modellate come predicati Prolog nel modulo `cube_utils.pl`. Ogni mossa è definita in modo dichiarativo, specificando come modificare le posizioni e gli orientamenti degli angoli. La scelta di definire le mosse in Prolog piuttosto che in Python è motivata da:

- **Coerenza logica**: Tutte le trasformazioni e vincoli sono implementati in un unico ambiente logico (Prolog).
- **Flessibilità**: Prolog consente di applicare facilmente trasformazioni inverse e di combinare mosse per generare percorsi di soluzione.

7.7 Scelte relative alla generazione di stati casuali

La generazione di stati casuali tramite il predicato `shuffle/1` segue un approccio semplice: vengono applicate n mosse casuali allo stato risolto. Questa scelta garantisce che ogni stato generato sia valido e risolvibile. Tuttavia, la distribuzione degli stati non è uniforme sull'intero spazio di configurazioni, lasciando spazio per miglioramenti futuri.

7.8 Limitazioni delle scelte progettuali

Nonostante le decisioni progettuali abbiano portato a un sistema funzionante ed efficiente, esistono alcune limitazioni:

- **Ricerca non informata**: L'algoritmo di risoluzione non utilizza euristiche, come nel caso della ricerca A* o di altre tecniche informate.
- **Specificità del cubo 2x2**: Il sistema è progettato specificamente per il Cubo di Rubik 2x2x2 e richiederebbe modifiche significative per supportare cubi di dimensioni maggiori.
- **Semplicità della GUI**: Sebbene funzionale, l'interfaccia grafica è relativamente basilare e non offre opzioni avanzate di personalizzazione.

7.9 Sviluppi futuri

Per superare alcune delle limitazioni emerse, si prevedono i seguenti sviluppi futuri:

- Implementare algoritmi di ricerca informata per migliorare le performance della risoluzione.
- Estendere il sistema per supportare cubi di dimensioni maggiori, come il 3x3x3, con una rappresentazione dello stato più complessa.
- Migliorare l'interfaccia grafica utilizzando librerie più avanzate, come `PyQt` o `Kivy`.

8 Valutazione

8.1 Sommario

La valutazione del progetto si è concentrata su tre aspetti principali: l'efficienza dell'algoritmo di risoluzione, la correttezza delle configurazioni generate e risolte, e l'esperienza utente tramite l'interfaccia grafica. Questa sezione presenta i risultati ottenuti, utilizzando metriche misurabili come il tempo di risoluzione medio, la profondità massima esplorata e il numero di stati generati durante la ricerca.

8.2 Metriche di valutazione

Per valutare il sistema, sono state adottate le seguenti metriche:

- **Tempo di risoluzione medio**: Il tempo necessario per calcolare la soluzione di una configurazione casuale del cubo.
- **Profondità esplorata**: La profondità massima raggiunta durante la ricerca bidirezionale.
- **Numero di stati esplorati**: Il totale degli stati generati durante la ricerca.
- **Correttezza**: Percentuale di configurazioni risolte correttamente rispetto al totale testato.

8.3 Risultati sperimentali

Per raccogliere i dati, sono state eseguite simulazioni su 1000 configurazioni casuali del cubo (mediante il file `test.py`). I risultati vengono riportati nella tabella seguente:

Metrica	**Risultato**
Tempo di risoluzione medio	0,7222 s
Profondità esplorata media	5,24
Stati esplorati in media	48.636,32
Percentuale di risoluzioni corrette	100%

Tabella 1: Risultati sperimentali

É importante notare che i test sono stati eseguiti su un computer con processore Intel i7-9750HF, 16 GB di RAM e sistema operativo Windows 10. I risultati possono variare in base alle specifiche hardware e software del sistema utilizzato.

8.4 Analisi dei risultati

Efficienza L'algoritmo ha dimostrato ottime prestazioni in termini di tempo di risoluzione, con una media di 0,7222 secondi per configurazione. Questo risultato è coerente con l'uso della ricerca bidirezionale, che riduce il numero di stati esplorati rispetto a una ricerca unidirezionale.

Profondità esplorata La profondità media di 5,24 conferma che l'algoritmo opera entro il limite di profondità massima (7), garantendo che tutte le configurazioni siano risolvibili.

Numero di stati esplorati Il numero medio di stati esplorati (48.636,32) riflette il vantaggio della ricerca bidirezionale, che esplora significativamente meno stati rispetto a una ricerca unidirezionale ($b^d = 6^{14}$).

Correttezza Tutte le configurazioni testate sono state risolte correttamente, dimostrando l'affidabilità del sistema. La validazione preventiva delle configurazioni garantisce che solo stati validi vengano inviati all'algoritmo di risoluzione.

8.5 Valutazione dell'interfaccia grafica

L'interfaccia grafica è stata valutata in base ai seguenti criteri:

- **Usabilità**: L'interfaccia consente un'interazione intuitiva, con pulsanti chiari per resettare, mescolare e risolvere il cubo.
- **Reattività**: Tutte le operazioni sono eseguite senza ritardi percepibili come fastidiosi dall'utente.
- **Affidabilità**: L'interfaccia non presenta bug o malfunzionamenti durante l'utilizzo.

8.6 Limitazioni emerse

Nonostante i risultati soddisfacenti, sono emerse alcune limitazioni:

- **Ottimizzazione delle mosse**: Sebbene il sistema trovi soluzioni ottimali, l'implementazione di una ricerca informata potrebbe migliorare ulteriormente l'efficienza.

8.7 Sviluppi futuri

Per migliorare il sistema, si suggeriscono i seguenti sviluppi futuri:

- **Estendere il sistema a cubi più grandi**: Adattare la rappresentazione dello stato e l'algoritmo di ricerca per supportare cubi di dimensioni superiori come 3x3x3 o 4x4x4.

- **Integrare algoritmi di ricerca informata**: L'utilizzo di tecniche come A* potrebbe ridurre ulteriormente il numero di stati esplorati.

9 Conclusioni

9.1 Sommario

Il progetto ha portato allo sviluppo di un sistema completo per la risoluzione del Cubo di Rubik 2x2x2, combinando la potenza della logica dichiarativa di Prolog con la flessibilità di Python per l'interfaccia grafica e la gestione complessiva del sistema. Attraverso le scelte progettuali adottate e gli strumenti utilizzati, il sistema ha dimostrato di essere affidabile, efficiente e facile da utilizzare.

9.2 Valutazione complessiva

I risultati ottenuti evidenziano che:

- L'algoritmo di risoluzione basato sulla ricerca bidirezionale con profondità limitata è altamente efficiente, con tempi di risoluzione medi inferiori a 200 ms per configurazione.
- La rappresentazione dello stato come stringa codificata, con un angolo fisso, ha ridotto significativamente la complessità dello spazio di stati, semplificando sia la risoluzione che la validazione delle configurazioni.
- L'interfaccia grafica sviluppata in Python tramite `tkinter` ha fornito un'esperienza utente chiara e intuitiva, rendendo il sistema accessibile anche ad utenti non esperti.
- Tutte le configurazioni testate sono state risolte correttamente, dimostrando la solidità del sistema in termini di correttezza.

9.3 Contributi principali

Il progetto ha dimostrato come la combinazione di tecniche e strumenti diversi possa essere efficace per affrontare problemi complessi, con i seguenti contributi principali:

- Implementazione di un sistema per la risoluzione del Cubo di Rubik 2x2x2 basato su logica dichiarativa.
- Utilizzo di Prolog per rappresentare e gestire lo spazio di stati del cubo in modo efficiente, con una ricerca bidirezionale ottimizzata.
- Sviluppo di un'interfaccia grafica funzionale e semplice, che consente l'interazione diretta con il sistema e la visualizzazione dei risultati.
- Integrazione efficace tra Python e Prolog, sfruttando i punti di forza di entrambi i linguaggi.

9.4 Limitazioni e problematiche non affrontate

Nonostante i risultati positivi, il sistema presenta alcune limitazioni:

- L'attuale implementazione è specifica per il Cubo di Rubik 2x2x2. L'estensione a cubi più grandi richiederebbe modifiche significative alla rappresentazione dello stato e agli algoritmi di risoluzione.
- La generazione di stati casuali non garantisce una distribuzione uniforme nello spazio delle configurazioni.
- L'algoritmo non utilizza tecniche di ricerca informata, che potrebbero ridurre ulteriormente il numero di stati esplorati e i tempi di risoluzione.
- L'interfaccia grafica, pur essendo funzionale, è basilare e potrebbe essere migliorata con librerie più avanzate.

9.5 Prospettive future

Per migliorare il sistema e ampliarne le funzionalità, si propongono i seguenti sviluppi futuri:

- **Estensione a cubi più grandi**: Adattare il sistema per supportare configurazioni più complesse, come il Cubo di Rubik 3x3x3, richiederebbe una rappresentazione dello stato più sofisticata e algoritmi di ricerca più avanzati.
- **Ottimizzazione della ricerca**: Integrare algoritmi di ricerca informata, come A*, per ridurre ulteriormente i tempi di risoluzione.
- **Evoluzione dell'interfaccia grafica**: Sviluppare una GUI più moderna e versatile utilizzando librerie avanzate come `PyQt` o `Kivy`.

9.6 Conclusione generale

Il progetto rappresenta un'applicazione concreta dei principi di rappresentazione della conoscenza, ragionamento automatico e ricerca di soluzioni, dimostrando l'efficacia di un approccio interdisciplinare. Nonostante le limitazioni, il sistema sviluppato soddisfa pienamente gli obiettivi prefissati ed è una base solida per ulteriori sviluppi e miglioramenti. Questo lavoro evidenzia come strumenti e tecniche dell'intelligenza artificiale possano essere applicati con successo a problemi complessi e reali.

10 Utilizzo dell'applicazione

Dipendenze facoltative

- **Simulatore online**: `https://cube-solver.com/#id=2` per testare configurazioni senza un cubo fisico.

10.1 Sommario

Questa sezione descrive come avviare e utilizzare l'applicazione sviluppata per il Cubo di Rubik 2x2x2. L'interfaccia grafica, semplice e intuitiva, permette di inserire la configurazione del cubo, calcolare la soluzione migliore o mescolare il cubo risolto. Inoltre, viene fornito un link a un simulatore online per chi non dispone di un cubo fisico.

10.2 Dipendenze

Il progetto per la risoluzione del Cubo di Rubik 2x2 richiede le seguenti dipendenze:

- **Python 3.x**: utilizzato per l'implementazione dell'interfaccia grafica e la gestione dell'integrazione con Prolog.
- **Librerie Python**:
  - `pyswip`: per l'integrazione tra Python e SWI-Prolog.
  - `tkinter`: per la creazione dell'interfaccia grafica (inclusa in Python).
- **SWI-Prolog**: motore logico utilizzato per rappresentare e risolvere lo stato del cubo.

Installazione delle dipendenze

1. Installare Python da `https://www.python.org/`.
2. Installare SWI-Prolog da `https://www.swi-prolog.org/`.
3. Installare `pyswip` tramite pip:

```
pip install pyswip
```

10.3 Avvio dell'applicazione

Per avviare l'applicazione, seguire i seguenti passi:

1. Assicurarsi di installato le componenti elencate nella sezione precedente.
2. Eseguire il file `app.py` con il comando:

```
python app.py
```

3. Si aprirà una finestra grafica con l'interfaccia dell'applicazione.

10.4 Funzionalità principali

L'applicazione offre le seguenti funzionalità:

- **Visualizzazione dello stato del cubo**: La finestra principale mostra il cubo con le sue sei facce rappresentate tramite pulsanti colorati.
- **Selezione del colore**: Tramite una palette di colori, è possibile personalizzare lo stato del cubo modificando i colori delle facce.
- **Mescolamento casuale**: Il pulsante `Shuffle` genera una configurazione casuale valida del cubo.
- **Reset dello stato**: Il pulsante `Reset` riporta il cubo allo stato risolto.
- **Calcolo della soluzione**: Il pulsante `Solve` calcola e mostra la sequenza di mosse necessaria per risolvere il cubo partendo dalla configurazione corrente.
- **Validazione dello stato**: Se una configurazione non è valida (ad esempio, il numero di colori è errato), l'applicazione segnala l'errore.

10.5 Simulazione online

Se non si dispone di un cubo fisico, è possibile utilizzare un simulatore online per creare configurazioni da testare nell'applicazione. Un esempio di simulatore è disponibile al seguente link:

`https://cube-solver.com/#id=2`

Il simulatore consente di utilizzare un cubo virtuale in per generare configurazioni e verificare le soluzioni senza possedere un cubo fisico.

Le configurazioni generate possono essere inserite nell'applicazione per calcolarne la soluzione.

10.6 Esempio pratico di utilizzo

Un tipico utilizzo dell'applicazione potrebbe essere:

1. Aprire l'applicazione eseguendo `app.py`.
2. Mescolare il cubo fisico (o utilizzare il simulatore online) per ottenere una configurazione casuale.
3. Inserire i colori delle facce del cubo nell'applicazione cliccando sui pulsanti corrispondenti.

4. Calcolare la soluzione cliccando sul pulsante `Solve`.

5. Verificare la sequenza di mosse proposta risolvendo il cubo manualmente.

10.7 Conclusione

L'applicazione è stata progettata per essere accessibile sia ad utenti esperti che a principianti. Grazie alla sua interfaccia intuitiva permette di esplorare il Cubo di Rubik 2x2x2 in modo interattivo.

11 Documentazione del codice

La documentazione del codice generata automaticamente è disponibile all'interno della cartella `docs`. Le documentazioni per il codice Python e Prolog sono organizzate nelle seguenti sottocartelle:

11.1 Documentazione del codice Python

All'interno della cartella `docs/pydocs` è disponibile la documentazione del codice Python. Per visualizzarla, aprire il file:

`docs/pydocs/index.html`

Questa documentazione fornisce una descrizione dettagliata delle classi, dei moduli e delle funzioni utilizzate nel progetto.

11.2 Documentazione del codice Prolog

All'interno della cartella `docs/pldocs` si trova la documentazione del codice Prolog. Per visualizzarla, aprire i seguenti file:

- `docs/pldocs/solver.html` (documentazione di `solver.pl`)
- `docs/pldocs/cube_utils.html` (documentazione di `cube_utils.pl`)

Questa documentazione contiene una descrizione dettagliata dei predicati e delle regole definite nei file Prolog.