UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Cluedo Solver

Risolutore automatico di partite di Cluedo

Studente

Davide Carella, 775492, d.carella12@studenti.uniba.it

Anno accademico

2024/2025

Repository

https://github.com/ITHackerstein/CluedoSolver

Indice

Capitolo 1

Introduzione

1.1 Scopo del progetto

Ogni partita di Cluedo comincia con l'omicidio di Samuel Black che viene trovato morto nella sua villa. I giocatori devono aiutare i detective a risolvere il delitto di cui si sa che:

- l'assassino è uno fra i seguenti sospetti: *Green*, *Mustard*, *Orchid*, *Peacock*, *Plum* e *Scarlet*.
- l'arma del delitto è una fra le seguenti: *Candeliere*, *Pugnale*, *Tubo di piombo*, *Pistola*, *Corda* e *Chiave inglese*.
- il luogo del delitto è una fra le seguenti stanze: *Sala da biliardo*, *Sala da ballo*, *Sala da pranzo*, *Serra*, *Ingresso*, *Cucina*, *Biblioteca*, *Soggiorno* e *Studio*.

L'obiettivo del gioco è scoprire chi ha commesso l'omicidio, con quale arma e in quale stanza.

Il numero di giocatori può andare da 2 a 6 e ad ognuno di essi viene distribuito, in maniera casuale, un numero uguale di carte (ricordando che 3 di tipo diverso vanno riservate alla soluzione), se questo non è possibile si cerca di distribuirle nella maniera più equa possibile.

A turno, ogni giocatore farà un'*ipotesi* sulla soluzione, a quel punto, partendo dal prossimo giocatore, ogni giocatore controllerà se possiede una delle tre

carte dell'ipotesi e, in caso positivo, risponde al giocatore che ha fatto l'ipotesi facendo vedere quale carta possiede.

Se uno dei giocatori non risponde e nell'ipotesi non si è inserita nessuna delle proprie carte, allora si è trovata la soluzione della partita.

Il sistema utilizzerà le ipotesi formulate durante la partita per ridurre l'insieme delle possibili configurazioni della partita e, alla fine, fornire all'utente le soluzioni più probabili.

1.2 Argomenti affrontati

- **Ricerca su spazi di stati**: Cluedo non è altro che uno spazio di stati in cui ogni stato rappresenta una possibile configurazione della partita e la ricerca su tale spazio permette di trovare le soluzioni più probabili;
- **Ragionamento con vincoli**: lo spazio di stati di Cluedo può essere ridotto utilizzando i vincoli che si ottengono dalle ipotesi formulate durante la partita;
- **Ragionamento su modelli di conoscenza incerta**: le carte che i giocatori possiedono sono incerte, il sistema deve essere in grado di gestire questa incertezza, inoltre lo spazio di stati è spesso troppo grande per essere esplorato completamente per cui si deve fare un ragionamento approssimato.

Capitolo 2

Ricerca su spazi di stati

Cluedo può essere visto come uno spazio di stati in cui ogni stato rappresenta una possibile configurazione della partita. Durante la partita vengono acquisite nuove informazioni che permettono di escludere alcune configurazioni non consistenti con le osservazioni fatte. Le transizioni da uno stato all'altro avvengono ogni volta che si apprendono nuovi dati, restringendo progressivamente il numero di possibilità valide.

Tuttavia, come abbiamo visto in precedenza, il numero totale di stati è estremamente elevato, rendendo un'esplorazione esaustiva di tutte le configurazioni possibili proibitiva. Dunque è necessario adottare metodi di esplorazione efficienti per trovare le soluzioni più probabili.

2.1 Strategia di riduzione degli stati

In Cluedo abbiamo un totale di 21 carte (6 sospetti, 6 armi e 9 stanze), 3 delle quali sono riservate alla soluzione del gioco. Ogni giocatore possiede un numero casuale di carte, in media 3, e le carte che non sono state distribuite ai giocatori vengono mescolate e distribuite in maniera casuale tra i giocatori.

Sapendo questo è possibile calcolare il numero di soluzioni che sarà:

$$6 \cdot 6 \cdot 9 = 324$$

in quanto ogni soluzione ha sempre una carta di ogni categoria.

Possiamo poi calcolare il numero di modi con i quali si possono distribuire le carte ai giocatori che sarà:

$$(21 - 3)! = 18! = 6,402,373,705,728,000$$

Quindi, in totale, il numero di configurazioni possibili sarà:

$$(6 \cdot 6 \cdot 9) \cdot 18! = 2,074,369,080,655,872,000$$

Per ridurre il numero di stati da esplorare, si possono sfruttare i vincoli derivanti dalle informazioni acquisite durante la partita.

Ad esempio, supponiamo di scoprire che un determinato giocatore possiede una carta specifica. Questa informazione elimina immediatamente tutte le configurazioni in cui quella carta non è assegnata a quel giocatore e quelle in cui la soluzione contenga quella carta. Vediamo nel dettaglio come cambio lo spazio di stati in base alla categoria della carta di cui si è venuti a conoscenza:

1. La carta che possiede il giocatore è un sospetto, allora il numero di configurazioni possibili si riduce a:

   $$5 \cdot 6 \cdot 9 \cdot 17! = 96,035,605,585,920,000$$

2. La carta che possiede il giocatore è un'arma, allora il numero di configurazioni possibili si riduce a:

   $$6 \cdot 5 \cdot 9 \cdot 17! = 96,035,605,585,920,000$$

3. La carta che possiede il giocatore è una stanza, allora il numero di configurazioni possibili si riduce a:

   $$6 \cdot 6 \cdot 8 \cdot 17! = 102,437,979,291,648,000$$

Dunque, notiamo come anche la semplice conoscenza di una singola carta permetta di ridurre lo spazio delle configurazioni possibili di circa 20 volte. Questo principio si applica a ogni altra informazione raccolta per restringere progressivamente lo spazio di ricerca per convergere alla soluzione più probabile.

Nel prossimo capitolo, verrà analizzato nel dettaglio il modo in cui ogni vincolo viene trattato e come il sistema sfrutta tale informazioni per dedurre nuovi fatti.

2.2 Strategia di ricerca

Poiché esplorare l'intero spazio di stati rimane comunque impraticabile anche dopo l'applicazione dei vincoli, il sistema adotta una strategia di ricerca probabilistica per stimare le soluzioni più probabili. In particolare, si applicherà una simulazione stocastica basata su tecniche Monte Carlo per esplorare in modo approssimato lo spazio di stati.

L'idea alla base di questa strategia è di generare numerose configurazioni valide, rispettando i vincoli appresi, e osservare la frequenza con cui determinate soluzioni appaiono. Invece di verificare tutte le configurazioni possibili, il sistema ne esaminerà solo un sottoinsieme signfiicativo ricavando un'approssimazione.

In questo modo non si troverà la soluzione esatta ma, a patto che venga scelto un numero di simulazioni sufficientemente elevato, se ne otterà una con un altro grado di affidabilità. Al contrario, se il numero di simulazioni è troppo basso si potrebbe avere una stima poco precisa.

Il tutto verrà discusso nel dettaglio successivamente.

Capitolo 3

Ragionamento con vincoli

Come accennato in precedenza, durante ogni partita di Cluedo i giocatori, a turno, fanno delle ipotesi che danno luogo a dei vincoli che possono essere utilizzati per ridurre lo spazio di stati della partita. Il sistema definirà una KB che è in grado di acquisire le informazioni durante la partita e la aggiornerà seguendo un processo di inferenza automatico che permette di dedurre nuove informazioni.

In questo capitolo affronteremo come vengono rappresentate le informazioni, come vengono definiti i vincoli e come vengono dedotte nuove informazioni.

3.1 Rappresentazione delle informazioni

Nella fase di progettazione del sistema, sono state vagliate diverse possibilità per rappresentare lo stato della partita e le sue informazioni. Questa che viene riportata di seguito è quella attuale e che si è ritenuta più valida per lo scopo del progetto.

3.2 Carte

Per rappresentare le carte del gioco sono state definite due principali enumerazioni: `CardCategory`, che contiene le tre possibili categorie, e `Card`, che contiene tutte le carte.

In aggiunta a queste due enumerazioni, sono stati inseriti metodi necessari a recuperare informazioni sulle carte come, ad esempio, la categoria di una determinata carte, il numero di carte contenute in una categoria oppure tutte le carte di una categoria.

Inoltre, si è visto, durante la progettazione, che serviva un'astrazione per un insieme di carte: `CardSet`. Inizialmente questa classe era implementato con un *hash set* di carte, ma in fase di sperimentazione si è visto che era molto inefficiente e quindi si è passati ad una implementazione con un *bit set*. In ogni caso le possibili operazioni che è possibile effettuare con questa classe sono quelle classiche di un insieme: inserimento/rimozione di un elemento, controllo di appartenenza, unione e intersezione.

3.3 Partita

Per tutto ciò che riguarda la partita si sono trovate le due principali astrazioni: `Player` e `Solver`.

3.3.1 Classe `Player`

La classe `Player` rappresenta un giocatore della partita ed ha i seguenti attributi:

- `card_count`: il numero di carte che il giocatore possiede;
- `cards_in_hand`: l'insieme di carte che il giocatore possiede sicuramente;
- `cards_not_in_hand`: l'insieme di carte che il giocatore non possiede sicuramente;
- `possibilities`: la lista di *possibilità* del giocatore, dove ogni possibilità è un insieme di carte di cui il giocatore ha sicuramente una carta.

Inoltre, la classe possiede diversi metodi utili all'acquisizione di informazioni sulle carte che possiede o meno il giocatore.

3.3.2 Classe `Solver`

La classe `Solver` rappresenta il sistema che risolve una partita e possiede tutti i dati necessari a rappresentarla, ovvero la lista dei suoi giocatori. Principalmente la classe avrà due compiti principali:

- l'acquisizione di informazioni sulla partita e, di conseguenza, sui giocatori;
- la ricerca delle soluzioni più probabili

È importante notare che la classe `Solver`, per comodità di implementazione, rappresenta la soluzione come un particolare giocatore, nascosto all'utente, che avrà sempre 3 carte di categoria distinta (informazione utile per la ricerca delle soluzioni).

3.4 Vincoli

3.4.1 Un giocatore possiede una carta

Nel caso in cui si sappia che il giocatore possiede una carta, non si farà altro che aggiornare il suo insieme `cards_in_hand` inserendo tale carta con il metodo `add_in_hand_card`.

Il metodo `add__in_hand_card`, oltre che aggiungere la carta all'insieme, eseguirà diversi metodi che tenteranno di deduure nuove informazioni, ovvero:

- `remove_superfluous_possibilities`

  Questo metodo rimuoverà le possibilità duplicate o quelle che sono un soprainsieme di un'altra possibilità.

- `simplify_possibilities_with_card`

  Questo metodo rimuoverà tutte quelle possibilità che contengono la carta in questione.

- `check_if_all_cards_in_hand`

  Questo metodo controllerà se la dimensione dell'insieme `cards_in_hand` è pari al numero di carte del giocatore e, in caso positivo, aggiungerà tutte le carte rimanenti all'insieme `cards_not_in_hand`.

Il sistema, inoltre, se il giocatore in questione è la soluzione, imparerà che non può possedere nessun'altra carta appartente alla categoria della carta in questione.

3.4.2 Un giocatore non possiede una carta

Nel caso in cui si sappia che il giocatore non possiede una carta, non si farà altro che aggiornare il suo insieme `cards_not_in_hand` inserendo tale carta con il metodo `add_not_in_hand_card`.

Il procedimento per dedurre nuove informazioni è lo stesso del vincolo precedente se non fosse che il metodo `simplify_possibilities_with_card` rimuoverà la carta dalle possibilità che la contengono e, se dopo la rimozione la possibilità ha solo una carta, impara che il giocatore a tale carta e rimuove tale possibilità.

3.4.3 Un giocatore ha una ed una sola carta di un insieme

Nel caso in cui si sappia che il giocatore possiede una ed una sola carta fra quelle contenute in un insieme, non si farà altro che aggiungere tale insieme a `possibilities`. Una volta inserito si richiameranno i metodi `remove_superfluous_possibilities` e `check_if_all_cards_in_hand` per dedurre nuove informazioni.

3.4.4 Un giocatore fa un'ipotesi durante la partita

Questo è il vincolo più complesso fra tutti e fa utilizzo di quelli precedenti. Come spiegato inizialmente, un'ipotesi sarà costituita dalle seguenti informazioni:

- `suggesting_player`: il giocatore che ha fatto l'ipotesi;
- `suspect`: il sospetto dell'ipotesi;
- `weapon`: l'arma dell'ipotesi;
- `room`: la stanza dell'ipotesi;
- `responding_player`: il giocatore che ha risposto all'ipotesi, se esiste;

- `response_card`: la carta che il giocatore ha mostrato, se la si conosce.

In questo caso partendo dal giocatore successivo a `suggesting_player` si effettueranno le seguenti operazioni:

- se si è raggiunto il `suggesting_player` allora si termina;
- se si è raggiunto il `responding_player` allora si termina e possiamo concludere che:
  - se si conosce `response_card`, il giocatore possiede tale carta;
  - se non la si conosce, il giocatore possiede una carta tra `suspect`, `weapon` e `room`.
- per ogni altro giocatore che si incontra prima della terminazione, si sa che non ha nessuna carta tra `suspect`, `weapon` e `room`.

3.5 Dedurre nuove informazioni

Per ogni vincolo, dopo questo che viene inserito, viene richiamato il metodo `infer_new_information` che si occupa di trovare nuove informazioni analizzando lo stato attuale della partita.

Il metodo può essere diviso in tre parti:

1. la prima parte, per ogni carta, controlla se:
   - un giocatore la possiede, allora si deduce che tutti gli altri non ce l'hanno;
   - nessun giocatore la possiede tranne uno, allora si deduce che quel giocatore la deve possedere.
2. la seconda parte cerca di dedurre informazioni sulla soluzione sapendo che le sue carte hanno categoria distinta: per ogni categoria non ancora conosciuta della soluzione, controlla se c'è una ed una sola carta che non è posseduta da alcun giocatore; in tal caso si può dedurre che allora deve essere la soluzione per quella categoria.
3. la terza parte verifica se ci sono giocatori che hanno possibilità in comune, se ce ne sono e il numero di gicatori è maggiore di quello del-

la dimensione della possibilità, allora si può dedurre che nessun altro giocatore possiede le carte della possibilità.

3.6 Osservazioni

La rappresentazione scelta per rappresentare le informazioni sulla partita è quella che si è comportata meglio in termini di efficienza e facilità d'uso, infatti permette di aggiungere nuovi vincoli in maniera semplice e intuitiva.

La rappresentazione scelta per rappresentare le informazioni sulla partita presenta un piccolo limite che sta proprio nella lista di possibilità del giocatore: questa è memorizzata come una lista dinamica di `CardSet` in quanto non si conosce la sua lunghezza priori. Questo comporta che venga memorizzata nell'*heap* piuttosto che nello *stack* che causa una maggiore lentezza nelle operazioni di gestione della memoria. Una possibile soluzione potrebbe esssere quella di analizzare, in media, quale sia la lunghezza di questa lista è utilizzare tale valore come soglia al di sotto della quale si memorizzerà la lista nello stack altrimenti si passa all'allocazione dinamica. (si veda Small vector optimization).

Si ha un problema simile nella memorizzazione dei nomi dei giocatori (escluso dalla discussione della rappresentazione in quanto si è ritenuto che non fosse necessario) rappresentati come della stringhe di lunghezza variabile. La possibile soluzione per questo problema sarebbe rimuovere i nomi del tutto e memorizzarli separatamente oppure limitarne la lunghezza a un valore fisso in modo da memorizzarle in maniera statica velocizzando le operazioni di gestione della memoria.

Capitolo 4

Ragionamento su modelli di conoscenza incerta

I vincoli descritti nel capitolo precedente permettono di ridurre lo spazio di ricerca, ciò nonostante esplorarlo nella sua interezza per trovare la soluzione più probabile in maniera accurata diventa dispendioso in termini di tempo. Per questo motivo bisogna optare per una simulazione stocastica per trovare un'approssimazione della soluzione più probabile.

4.1 Calcolo delle soluzioni più probabili

La parte del sistema che si occupa di trovare le soluzioni più probabili è il metodo `find_most_likely_solutions` della classe `Solver` che esegue le seguenti operazioni:

1. esclude dalle soluzioni le carte che sappiamo per certo non poterlo essere (quelle contenute nell'insieme `cards_not_in_hand` della soluzione);
2. per ogni possibile soluzione:
   (a) si copia lo stato attuale della partita;
   (b) si memorizza l'insieme delle carte che nessun giocatore possiede;
   (c) si impara che la soluzione è costituita dalle tre carte scelte;
   (d) si imposta il contatore delle iterazioni valide a 0;

(e) per un numero prefissato di iterazioni si assgenano casualmente le carte ai giocatori e si controlla che i vincoli siano rispettati, se lo sono si incrementa il contatore;

(f) si calcola la probabilità di tale soluzione come il rapporto fra il numero di iterazioni valide su quelle totali;

3. si restituiscono le soluzioni dalla più probabile alla meno probabile.

Il numero di iterazioni totale è attualmente impostato a 1,000,000 che si è visto fornire un buon compromesso tra efficienza e accuratezza, due metriche di cui si parlerà in seguito.

4.2 Osservazioni

L'algoritmo descritto sopra ha subito diverse modifiche poiché spesso si incappava nel biasing di alcune soluzioni.

Inizialmente l'algoritmo, durante l'assegnazione delle carte ai giocatori, evitava le assegnazioni che si sapeva già fossero inconsistenti con i vincoli appresi. Questo portava ad un miglioramento delle prestazioni in quanto si riduceva il numero di simulazioni incosistenti fatte portando, però, ad un biasing dei risultati poiché ogni simulazione risultava consistente ma ogni soluzione aveva stessa probabilità. Per risolvere questo problema si è deciso di assegnare le carte in maniera totalmente casuale per ogni simulazione e controllare solo in seguito se questa simulazione fosse consistente.

In seguito, si è però notato che l'algoritmo presentava un problema simile quando assegnavamo la soluzione. Infatti, assegnare la soluzione prima di trovare le carte che è possibile assegnare ai giocatori, portava ad un biasing dei risultati. Si supponga, ad esempio, di essere in una partita nella quale si sa che o un giocatore o la soluzione hanno una carta, selezionando una soluzione che non contiene quella carta prima di trovare le carte che è possibile assegnare, porterà a dedurre che il giocatore ha quella carta, ma non dovremmo avere tale conoscenza in fase di distribuzione delle carte.

Per risolvere il problema dunque bisogna trovare le carte che è possibile assegnare ai giocatori prima di assegnare la soluzione.

Capitolo 5

Valutazione

La valutazione del progetto si è basata su due metriche: accuratezza ed efficienza. In questa sezione esploreremo i risultati ottenuti per queste due metriche.

5.1 Accuratezza

Per ciò che riguarda l'accuratezza, purtroppo non è possibile confrontare le probabilità trovate direttamente con quelle esatte, in quanto quest'ultime, come spiegato inizialmente, sono estremamente difficili da calcolare.

Per ovviare a questo problema si è deciso di verificare l'affidabilità dei risultati confrontandoli con i risultati di partite già concluse. L'idea è quella di osservare il comportamento del sistema in scenari pratici e valutare se le probabilità stimate si allineano con l'esito effettivo del gioco. Tuttavia, anche questa strategia presenta delle difficoltà: non esiste un dataset pubblico di partite di Cluedo finite che possa essere utilizzato per testare il sistema in modo rigoroso e su un'ampia varietà di situazioni.

Per questo motivo si è deciso di adottare un approccio più pratico e sperimentale. Sono state giocate diverse partite di Cluedo, sia reali che simulate, annotando con precisione tutte le informazioni raccolte durante il gioco e verificando se il sistema fosse in grado di riprodurre correttamente il ragionamento dei giocatori. In alcuni casi, sono state costruite ad hoc delle partite "finite", nelle quali si sono impostate situazioni specifiche con lo sco-

po di verificare la risposta del sistema a particolari combinazioni di vincoli e informazioni parziali.

Questi test hanno permesso di valutare il comportamento del sistema in condizioni realistiche e di individuare eventuali casi in cui le probabilità risultavano distorte o non coerenti con l'andamento della partita. Sebbene non si possa garantire un'accuratezza assoluta, i risultati ottenuti mostrano che il sistema riesce a fornire una buona approssimazione nella maggior parte dei casi, confermando l'efficacia dell'approccio scelto.

5.2 Efficienza

Per valutare l'efficienza, invece, ci si è concentrati sul tempo di esecuzione del metodo `find_most_likely_solutions` della classe `Solver`. Questo metodo, come spiegato in precedenza, esegue un numero prefissato di simulazioni per trovare le soluzioni più probabili. L'obiettivo è di trovare un buon compromesso tra accuratezza e tempo di esecuzione, in modo da ottenere risultati affidabili in tempi ragionevoli.

Poiché il sistema deve essere in grado di processare le informazioni in tempo reale durante una partita, il tempo di calcolo per ogni aggiornamento delle probabilità deve rimanere sufficientemente basso da non risultare penalizzante per l'esperienza dell'utente. Un tempo di risposta troppo elevato renderebbe il sistema poco pratico, soprattutto nel caso in cui venga utilizzato come assistente decisionale durante una partita in corso.

Per valutare le prestazioni, sono stati condotti i test su una CPU AMD Ryzen 5 5600X (6 core, 12 thread @ 3.7GHz) e RAM DDR4 16GB 3600 MT/s con una build ottimizzata del codice. I risultati hanno mostrato che il metodo impiega circa 100ms, un valore considerato accettabile per garantire un'analisi rapida senza sacrificare troppo la precisione. Questo tempo consente all'applicazione di rimanere fluida e reattiva, evitando ritardi frustranti per l'utente.

Per ottenere questo livello di efficienza e individuare i principali colli di bottiglia del codice, è stata eseguita una profilazione approfondita del metodo `find_most_likely_solutions`. Il processo di profilazione ha permesso di analizzare nel dettaglio quali sezioni del codice fossero più dispendiose in termini di tempo di calcolo, permettendo così di ottimizzarle in modo mirato.

Gli strumenti utilizzati per la profilazione e l'analisi delle prestazioni, vengono discussi successivamente.

Capitolo 6

Conclusioni

Il progetto Cluedo Solver ha dimostrato come sia possibile applicare tecniche avanzate di ragionamento con vincoli e simulazione Monte Carlo per risolvere problemi complessi con informazioni incomplete.

Grazie all'uso di vincoli logici, il sistema riduce in modo significativo lo spazio delle soluzioni possibili, mentre la simulazione Monte Carlo permette di gestire l'incertezza e identificare le soluzioni più probabili in modo efficiente. Questo approccio garantisce un buon compromesso tra accuratezza e prestazioni, evitando l'esplorazione esaustiva dello spazio di stati, che sarebbe computazionalmente proibitiva.

6.1 Sviluppi futuri

Il sistema in futuro si potrebbe migliorare in diversi modi:

- si potrebbero supportare più linguaggi e piattaforme per consentire un utilizzo più flessibile;
- si potrebbe tentare di modificare l'algoritmo per trovare le soluzioni più probabili utilizzando un algoritmo Markov Chain Monte Carlo in modo tale da trovare prima un'approssimazione della distribuzione di probabilità delle soluzioni e utilizzarla per evitare di effettuare troppe simulazioni inconsistenti migliorando così le prestazioni;

- si potrebbe rendere l'algoritmo *any-time* in modo tale che l'applicazione non venga bloccata durante la ricerca delle soluzioni più probabili ma continui a migliorare le probabilità trovate per ogni soluzione;
- la versione del Cluedo utilizzata è quella classica, ma ne esistono anche altre in cui cambiano il numero delle carte o anche la loro denominazione, si potrebbe generalizzare la rappresentazione delle carte per poterle supportare.

Capitolo 7

Tecnologie utilizzate

Uno degli obiettivi del progetto era cercare di realizzare tutto da zero senza l'utilizzo di librerie poiché si voleva sperimentare nella rappresentazione e nell'ottimizzazione delle strutture dati e gli algoritmi necessari per raggiungere gli scopi del progetto. Nonostante questo, dove si è ritenuto necessario o fuori dallo scopo del progetto, si è fatto uso di alcune librerie. Di seguito verrano riportati tutti le tecnologie utilizzate per la realizzazione del progetto.

7.1 Strumenti di sviluppo

7.1.1 C++

Si è scelto C++ per la realizzazione del progetto nella sua intereza per i seguenti motivi:

- **Ottima conoscenza del linguaggio**

  Questo ha semplificato la scrittura del codice in quanto si lavorava in un ambiente familiare.

- **Linguaggio a basso livello**

  Essendo C++ un linguaggio a basso livello compilato direttamente in codice macchina, si è ritenuto che sarebbe stato più facile ottimizzare il codice per ottenere le migliori prestazioni possibili.

- **Modernità**

  Nonostante il suo essere a basso livello, C++ è un linguaggio moderno che offre molte funzionalità che permettono di scrivere codice pulito, mantenibile e facile da leggere.

7.1.2 CMake

Si è scelto CMake come sistema di build per il progetto per i seguenti motivi:

- **Sintassi dichiarativa di alto livello**

  La sintassi di CMake è facile da leggere e permette di creare dei file di configurazione modulare che scalano meglio rispetto, ad esempio, a dei classici Makefile che possono diventare difficili da gestire.

- **Gestione delle dipendenze**

  CMake ha un supporto integrato per trovare e *link*are librerie esterne semplificando i progetti che non devono gestirle manualmente.

- **Compatibilità cross-platform**

  CMake genera script di build per sistemi diversi (Makefiles, Ninja, Visual Studio, ...) rendendo semplice la compilazione del progetto su altri sistemi senza dover riscrivere gli script.

7.1.3 Doxygen

Si è scelto Doxygen per la generazione della documentazione del progetto per i seguenti motivi:

- **Generazione automatica della documentazione**

  Doxygen permette di generare automaticamente la documentazione inserendo dei commenti speciali all'interno del codice, permettendo quindi di non separare la documentazione dal codice ma sincronizzandoli.

- **Formati di output multipli**

  Doxygen permette di specificare diversi formati di output: HTML, Latex, ...

- **Facilità d'uso**

  Doxygen è facile da usare e permette di generare la documentazione in pochi passaggi.

7.1.4 Git

Per il versionamento del progetto si è usato Git e il repository è stato pubblicato su GitHub.

L'utilizzo di questi strumenti ha permesso di:

- tenere traccia delle modifiche effettuate nel codice;
- tornare indietro in caso di errori;
- effettuare dei backup sulla piattaforma.

7.1.5 Strumenti di profilazione

Per misurare l'efficienza dell'applicazione e la sua profilazione sono stati usati i seguenti strumenti:

- hyperfine: uno strumento da linea di comando per fare benchmark di un processo che permette di eseguire anche analisti statistiche sui risultati;
- Valgrind: una suite di strumenti da linea di comando per il debugging e la profilazione di applicazioni, in particolare è stato usato callgrind che analizza le chiamate a funzioni, le istruzioni eseguite e il tempo speso per ognuna di esse;
- KCachegrind: uno strumento grafico che permette di analizzare i profili generati da Valgrind.

7.2 Librerie esterne

7.2.1 {fmt}

{fmt} è una libreria per la formattazione del testo scelta per la sua modernità e facilità d'uso. Questa, in particolare, è più efficiente di quella della

STL di C++ e verrà probabilmente implementata in uno standard futuro del linguaggio (si veda https://en.cppreference.com/w/cpp/utility/format).

7.2.2 JSON for Modern C++

JSON for Modern C++ è una libreria per gestire file JSON che vengono utilizzati nel progetto per memorizzare le stringhe da tradurre nelle diverse lingue che può supportare il sistema. Si è scelto di utilizzare una libreria per far questo in quanto, nonostante non sia eccessivamente complesso scriverne una da zero, rimane comunque fuori dallo scopo del progetto.

7.2.3 Dear ImGui e SDL

Dear ImGui è una libreria per creare GUI scelta per la sua leggerezza e semplicità grazie alla scelta di un'approccio *immediate* invece che *retained*. Questa supporta diversi backend per la visualizzazione di tali interfacce, in questo caso si è scelto di utilizzare SDL una libreria cross-platform per la gestione a basso livello di audio, video e grafica.

In questo caso sviluppare una libreria da zero che faccia questo è molto complesso e dispendioso in termini di tempo, quindi si preferisce utilizzare una libreria ben supportata e testata.

Capitolo 8

Utilizzo dell'applicazione

Per utilizzare l'applicazione bisogna seguire i passaggi descritti qui di seguito.

8.1 Compilazione

Il primo passo è compilare il progetto o scaricare una release.

8.1.1 Da zero

Per compilare l'applicazione da zero bisogna installare nel proprio sistema i seguenti software:

- GCC o Clang (su Windows non è possibile utilizzare MSVC in quanto non supporta una feature utilizzata nel codice, in alternativa è possibile utilizzare MinGW-w64);
- CMake;
- Git.

Una volta installati bisogna eseguire i seguenti comandi:

```
$ git clone --recursive https://github.com/ITHackerstein/
    CluedoSolver
$ cd CluedoSolver
$ mkdir build
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
$ cmake --build build
```

Su Windows i comandi rimangono gli stessi se non fosse che, nel caso si stia utilizzando MinGW, bisogna modificare il penultimo comando in:

```
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "MinGW
   Makefiles"
```

Per una maggiore efficienza durante la fase di compilazione è possibile specificare il numero di *job* da utilizzare per la compilazione aggiungendo l'opzione `-j` seguita dal numero di *job* (il numero dipende dalle specifiche del sistema che si sta utilizzando).

In alternativa, è possibile utilizzare Ninja se installato nel sistema specificando il seguente flag nel penultimo comando:

```
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G Ninja
```

8.1.2 Download di una release precompilata

In alternativa, è possibile scaricare una release precompilata dal repository. Al momento è presenta un'AppImage per Linux e un eseguibile per Windows.

8.2 Esecuzione

Una volta compilato o scaricata una release basterà aprire l'eseguibile che, nel caso in cui lo si abbia compilato, si troverà nella cartella `build/src/CluedoSolver`.

Capitolo 9

Documentazione del codice

La documentazione del codice è consultabile nel GitHub Pages del repository.