UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

8-Puzzle Solver

Progetto di Ingegneria della Conoscenza

Studente: Marco Ferrara
Matricola: 782407
Email: m.ferrara62@studenti.uniba.it

Anno Accademico:
2024–2025

Docente:
Prof. Nicola Fanizzi

Repository GitHub:
https://github.com/imb0ru/8puzzle-solver

Indice

Capitolo 1

Introduzione

1.1 Il Problema dell'8-Puzzle

L'8-puzzle rappresenta uno dei problemi classici nell'ambito dell'Intelligenza Artificiale e dell'Ingegneria della Conoscenza. Questo rompicapo a scorrimento, inventato nel 1874 da Noyes Palmer Chapman, consiste in una griglia 3×3 contenente otto tessere numerate da 1 a 8 e uno spazio vuoto. L'obiettivo è riordinare le tessere dalla configurazione iniziale a quella finale attraverso una sequenza di mosse che spostano le tessere adiacenti nello spazio vuoto.

La semplicità apparente del problema nasconde una complessità computazionale significativa. Lo spazio degli stati dell'8-puzzle contiene $\frac{9!}{2} = 181.440$ configurazioni raggiungibili, divise in due classi di equivalenza basate sulla parità delle inversioni. Questa proprietà matematica determina la risolvibilità di una configurazione: solo le configurazioni con un numero pari di inversioni possono essere trasformate nello stato obiettivo attraverso mosse legali.

Il problema dell'8-puzzle si presta particolarmente bene come caso di studio per l'Ingegneria della Conoscenza per diverse ragioni. Prima di tutto, la sua formulazione intuitiva permette di concentrarsi sugli aspetti algoritmici senza la complessità di domini specialistici. Inoltre, la dimensione limitata ma non banale dello spazio degli stati permette sperimentazione e validazione empirica degli algoritmi. La presenza di una metrica naturale (distanza dallo stato obiettivo) facilita lo sviluppo di euristiche ammissibili. Infine, l'esistenza di soluzioni ottimali note per molte configurazioni permette la valutazione oggettiva delle prestazioni algoritmiche.

1.2 Obiettivi del Progetto

Il progetto si propone di realizzare un sistema per la risoluzione automatica dell'8-puzzle. L'obiettivo principale è dimostrare come i concetti teorici dell'Ingegneria della Conoscenza possano essere applicati efficacemente a un problema concreto, evidenziando le caratteristiche distintive di ciascun approccio algoritmico.

Gli obiettivi specifici del progetto includono:

1. **Implementazione di algoritmi di ricerca nello spazio degli stati**: Realizzare versioni ottimizzate di A*, BFS e Greedy Best-First Search per la riso-

luzione dell'8-puzzle, permettendo un confronto diretto delle loro prestazioni in termini di ottimalità, completezza ed efficienza computazionale.

2. **Sviluppo di euristiche ammissibili e consistenti**: Progettare e implementare funzioni euristiche che guidino efficacemente la ricerca mantenendo le garanzie teoriche di ammissibilità.

3. **Implementazione di una Knowledge Base in Prolog per la rappresentazione e il ragionamento**: Sviluppare una Knowledge Base completa che integri rappresentazione dichiarativa della conoscenza del dominio (stati, transizioni, obiettivi) con meccanismi di ragionamento automatico (inferenza, unificazione, backtracking).

La realizzazione di questi obiettivi permette di esplorare in profondità le sfide dell'Ingegneria della Conoscenza, dalla rappresentazione formale del problema alla sua risoluzione efficiente, dalla validazione teorica alla verifica empirica delle prestazioni.

Capitolo 2

Strumenti Utilizzati

2.1 Sommario

Il sistema 8-puzzle è stato sviluppato seguendo un approccio multi-paradigma che integra programmazione logica e programmazione orientata agli oggetti. La scelta di utilizzare sia Prolog che Python riflette la volontà di sfruttare i punti di forza di ciascun linguaggio: Prolog per la sua naturale capacità di esprimere problemi di ricerca nello spazio degli stati attraverso regole logiche e backtracking automatico, Python per la sua versatilità nella gestione di interfacce grafiche, analisi dati e integrazione di sistemi eterogenei.

2.2 Python

Python 3.11 costituisce il framework principale per l'orchestrazione del sistema, scelto per la sua capacità di fungere da linguaggio "collante" tra componenti diverse. L'architettura orientata agli oggetti implementata in Python gestisce l'intero ciclo di vita dell'applicazione, dall'inizializzazione dei componenti alla presentazione dei risultati.

2.2.1 Interfaccia Grafica

L'interfaccia utente è realizzata con Tkinter, la libreria GUI standard di Python. La scelta di Tkinter garantisce portabilità cross-platform senza dipendenze esterne, permettendo deployment immediato su Windows, macOS e Linux. L'interfaccia implementa il pattern Model-View-Controller (MVC) per separare la logica di business dalla presentazione, facilitando manutenzione ed estensibilità.

2.2.2 Integrazione con Prolog

L'integrazione con il motore Prolog avviene attraverso la libreria pyswip, che fornisce un binding Python per SWI-Prolog. Questa libreria permette di interrogare la knowledge base Prolog direttamente da Python, mantenendo una comunicazione bidirezionale efficiente tra i due ambienti. La gestione delle query è incapsulata in classi

dedicate che gestiscono conversione dei tipi, gestione degli errori e ottimizzazione delle performance.

2.2.3 Gestione dei Dati e Analisi

Per l'analisi delle prestazioni, il sistema utilizza pandas per elaborare e aggregare i risultati dei test. I dati vengono esportati in tre formati complementari: CSV per l'analisi dettagliata, JSON per l'integrazione programmatica, e report Markdown con tabelle comparative delle prestazioni algoritmiche, includendo metriche quali tempo di esecuzione, nodi esplorati e ottimalità delle soluzioni.

2.3 Prolog

SWI-Prolog 9.0 fornisce il motore di inferenza per l'implementazione degli algoritmi di ricerca. La scelta di SWI-Prolog è motivata dalla sua maturità, dalla ricca libreria standard e dall'eccellente supporto per l'integrazione con altri linguaggi. Il paradigma dichiarativo di Prolog si adatta naturalmente alla rappresentazione di problemi di ricerca, permettendo implementazioni concise ed eleganti degli algoritmi.

2.4 Librerie e Framework

Tabella 2.1: Principali dipendenze Python del progetto

Libreria	**Versione**	**Utilizzo**
pyswip	0.2.10	Bridge per comunicazione Python-Prolog
tkinter	Built-in	Interfaccia grafica utente
pandas	2.0+	Analisi dati e statistiche
json	Built-in	Serializzazione configurazioni
threading	Built-in	Esecuzione asincrona
pathlib	Built-in	Gestione percorsi file system
datetime	Built-in	Timestamp e misurazioni temporali

Capitolo 3

Algoritmi di Ricerca

3.1 Fondamenti Teorici

La ricerca in spazi di stati costituisce un paradigma fondamentale per la risoluzione sistematica di problemi. Un problema di ricerca è caratterizzato da uno spazio degli stati finito o infinito, un insieme di operatori che definiscono le transizioni ammissibili, uno stato iniziale e un criterio per identificare gli stati obiettivo. La complessità computazionale degli algoritmi di ricerca dipende dal fattore di ramificazione, dalla profondità della soluzione e dalle proprietà dell'euristica utilizzata. Nel contesto dell'8-puzzle, lo spazio degli stati può essere formalizzato come un grafo $G = (V, E)$ dove V rappresenta l'insieme delle configurazioni valide del puzzle e E l'insieme delle transizioni legali tra configurazioni.

Lo spazio degli stati dell'8-puzzle è definito dalla quintupla $\langle S, A, T, s_0, G\rangle$ dove:

- $S = \{\text{permutazioni di } \{0, 1, 2, 3, 4, 5, 6, 7, 8\}\}$ con $|S| = 9! = 362.880$
- $A = \{\text{UP, DOWN, LEFT, RIGHT}\}$ rappresenta le azioni possibili
- $T : S \times A \to S$ è la funzione di transizione
- $s_0 \in S$ è lo stato iniziale
- $G = \{[1, 2, 3, 4, 5, 6, 7, 8, 0]\}$ è l'insieme degli stati obiettivo

La peculiarità dell'8-puzzle è che lo spazio degli stati è partizionato in due componenti connesse disgiunte di uguale dimensione. Una configurazione dell'8-puzzle è risolvibile se e solo se il numero di inversioni nella sua rappresentazione lineare è pari, dove un'inversione è una coppia (i, j) tale che $i < j$ ma tile$[i]$ > tile$[j]$ (escludendo lo spazio vuoto).

3.2 Scelta degli Algoritmi di Ricerca

La selezione degli algoritmi implementati nel sistema 8-puzzle è stata guidata da considerazioni teoriche e pratiche specifiche che coprono l'intero spettro delle strategie di ricerca nello spazio degli stati.

A* è stato scelto come algoritmo principale per la sua garanzia di ottimalità combinata con efficienza computazionale. In domini come l'8-puzzle dove esiste un'euristica ammissibile naturale (distanza di Manhattan), A* fornisce il miglior compromesso tra qualità della soluzione e tempo di calcolo. La sua completezza e ottimalità lo rendono ideale per applicazioni dove la qualità della soluzione è prioritaria.

Breadth-First Search è stato incluso come baseline di riferimento per validare l'ottimalità delle soluzioni A*. Essendo naturalmente ottimale per costi unitari, BFS permette di verificare che le soluzioni A* siano effettivamente di lunghezza minima. Inoltre, la sua implementazione semplice e le garanzie teoriche forti lo rendono un punto di confronto essenziale per valutare l'efficacia delle ottimizzazioni euristiche.

Greedy Best-First Search è stato aggiunto per esplorare il trade-off tra velocità e ottimalità. In scenari real-time o con risorse computazionali limitate, la capacità di Greedy di trovare soluzioni rapide (anche se subottimali) rappresenta un valore pratico significativo. La sua inclusione permette di quantificare il beneficio delle garanzie di ottimalità rispetto alla velocità pura.

Questa triade algoritmica permette di esplorare l'intero spettro delle strategie di ricerca, evidenziando come la conoscenza del dominio (euristica) influenzi le prestazioni e la qualità delle soluzioni.

3.3 A* con Euristica Combinata

L'algoritmo A* rappresenta il gold standard per la ricerca ottimale in spazi di stati con costi non negativi. La sua efficacia deriva dalla combinazione di ricerca best-first con una funzione di valutazione che stima il costo totale del percorso attraverso ogni nodo. A* utilizza una funzione di valutazione $f(n) = g(n) + h(n)$ dove $g(n)$ rappresenta il costo del percorso dallo stato iniziale al nodo n e $h(n)$ è la stima euristica del costo dal nodo n all'obiettivo.

Il predicato `solve_astar/5` implementa A* utilizzando una priority queue rappresentata come lista ordinata per f-score. La struttura del nodo è definita come `node(State, Path, G, H, F)` dove `F` è calcolato come `F is G + H`.

L'algoritmo principale `astar_search/4` mantiene una struttura dati essenziale composta da una lista `Open` ordinata per f-score crescente, un insieme `Closed` di stati visitati gestito tramite `assert(stato_visitato(State))`, e un contatore `NodesExplored` per raccogliere statistiche di esecuzione. L'espansione dei nodi è orchestrata dal predicato `expand_node/5`, che genera tutti i successori di uno stato attraverso `successors/2`, calcola il nuovo g-score incrementando di uno il costo del percorso (`NewG is G + 1`), e invoca `combined_heuristic/2` per ottenere la stima euristica. Il sistema filtra automaticamente gli stati duplicati controllando `stato_visitato/1` e inserisce i nuovi nodi nella lista aperta mantenendo l'ordinamento per f-score.

3.4 Breadth-First Search (BFS)

Il predicato `solve_bfs/5` implementa BFS utilizzando una coda FIFO per garantire esplorazione per livelli. L'implementazione non usa euristiche, esplorando sistematicamente tutti gli stati a distanza k prima di quelli a distanza k+1.

L'algoritmo `bfs_search/3` gestisce una coda FIFO implementata attraverso liste Prolog, un set di stati visitati mantenuto via asserzioni dinamiche, e un tracking implicito del livello di esplorazione attraverso la lunghezza del percorso. L'espansione sistematica per livelli viene garantita dall'estrazione sequenziale dei nodi dalla testa della coda, seguita dal controllo dell'obiettivo tramite `goal_state/1`. Per ogni nodo estratto, il sistema genera tutti i successori attraverso `successors/2`, filtra gli stati duplicati, e accoda i nuovi nodi validi utilizzando `append/3` per mantenere l'ordine FIFO.

La garanzia di ottimalità deriva dal fatto che il primo percorso trovato verso lo stato obiettivo ha necessariamente lunghezza minima.

3.5 Greedy Best-First Search

Il predicato `solve_greedy/5` implementa Greedy Best-First utilizzando solo il valore euristico per l'ordinamento. La struttura del nodo è `node(State, Path, H)` dove `H` è calcolato tramite `combined_heuristic/2`.

L'algoritmo `greedy_search/4` mantiene una lista aperta ordinata esclusivamente per h-score crescente, un insieme di stati visitati per la prevenzione di cicli, e implementa una strategia best-first che seleziona sempre il nodo con il minimo valore euristico. La strategia greedy si materializza attraverso l'ordinamento continuo della lista tramite `sort_by_h_greedy/2`, la selezione sistematica del nodo in testa, e l'espansione immediata senza considerazione del costo del percorso g-score. I successori vengono inseriti nella lista aperta mantenendo l'ordinamento per valore euristico.

L'assenza del fattore g-score porta a esplorazione rapida ma potenzialmente subottimale. Il predicato `expand_greedy_nodes/4` gestisce l'espansione calcolando solo il valore euristico per i nuovi stati.

3.6 Funzioni Euristiche

Le funzioni euristiche costituiscono l'elemento critico che distingue gli algoritmi di ricerca informata da quelli non informata. Nel contesto dell'8-puzzle, una buona euristica deve fornire una stima accurata della distanza dall'obiettivo pur rimanendo computazionalmente efficiente.

3.6.1 Manhattan Distance

La Manhattan Distance, anche nota come distanza L_1 o distanza di taxi, calcola per ogni tessera la somma delle distanze orizzontali e verticali dalla sua posizione attuale alla posizione obiettivo, ovvero il numero di mosse necessarie per portare ogni tessera nella sua posizione finale.

Definizione 3.1 (Manhattan Distance per l'8-Puzzle)**.** Dato uno stato s e lo stato obiettivo g, la Manhattan Distance è definita come:

$$h_{\text{MD}}(s) = \sum_{i=1}^{8} |x_i^s - x_i^g| + |y_i^s - y_i^g|$$

dove (x_i^s, y_i^s) e (x_i^g, y_i^g) sono rispettivamente le coordinate della tessera i nello stato s e g.

Il predicato `manhattan_distance/2` implementa questa formula attraverso `manhattan_distance_a` con accumulatore per efficienza. Il calcolo sfrutta `goal_positions/1` per il mapping tessera-posizione e `tile_manhattan/3` per il calcolo delle distanze individuali.

3.6.2 Misplaced Tiles

L'euristica Misplaced Tiles conta semplicemente il numero di tessere che non si trovano nella loro posizione finale.

Definizione 3.2 (Misplaced Tiles per l'8-Puzzle)**.**

$$h_{\text{MT}}(s) = |\{i : s[i] \neq g[i] \wedge s[i] \neq 0\}|$$

dove $s[i]$ rappresenta il contenuto della posizione i nello stato s.

Il predicato `misplaced_tiles/2` implementa questa definizione attraverso `misplaced_tiles_acc/4` con pattern matching parallelo di stato corrente e `goal_state/1`. L'implementazione esclude automaticamente lo spazio vuoto (tessera 0) dal conteggio.

3.6.3 Euristica Combinata

La scelta di sviluppare un'euristica combinata è motivata dalla complementarità delle caratteristiche informative della Manhattan Distance e delle Misplaced Tiles. Mentre la Manhattan Distance cattura accuratamente la distanza spaziale delle tessere dalle loro posizioni obiettivo, le Misplaced Tiles forniscono una misura diretta del progresso verso la configurazione finale.

Definizione 3.3 (Euristica Combinata)**.**

$$h_{\text{comb}}(s) = \max(h_{\text{MD}}(s), h_{\text{MD}}(s) + \frac{h_{\text{MT}}(s)}{2})$$

Teorema 3.1 (Ammissibilità dell'Euristica Combinata)**.** Se h_1 e h_2 sono euristiche ammissibili, allora $h(n) = \max(h_1(n), h_2(n))$ è anch'essa ammissibile.

Il predicato `combined_heuristic/2` implementa questa formula attraverso `manhattan_distance/2`, `misplaced_tiles/2` e la selezione del massimo per preservare l'ammissibilità.

Capitolo 4

Implementazione in Prolog

4.1 Scelta del Paradigma Logico

L'implementazione del motore di ricerca in Prolog è giustificata da diverse caratteristiche che si allineano perfettamente con i requisiti del problema. Il paradigma logico-dichiarativo permette di esprimere naturalmente le relazioni tra stati del puzzle come fatti e regole, riducendo la complessità implementativa e migliorando la manutenibilità del codice. Il motore di unificazione nativo di Prolog ottimizza automaticamente il matching di pattern negli stati, operazione critica per l'efficienza degli algoritmi di ricerca.

Il **backtracking automatico** elimina la necessità di gestire esplicitamente stack e strutture dati per l'esplorazione dello spazio degli stati. Questa caratteristica è particolarmente vantaggiosa nell'8-puzzle dove la ricerca può dover retrocedere frequentemente da vicoli ciechi. La gestione automatica della memoria e il cleanup delle variabili temporanee riducono significativamente la probabilità di memory leak e semplificano il debugging.

L'approccio **knowledge-based** di Prolog permette di codificare la conoscenza del dominio in modo dichiarativo. Le regole che definiscono mosse valide, stati obiettivo e funzioni euristiche possono essere espresse in forma logica naturale, facilitando la verifica formale della correttezza e l'estensione del sistema con nuove euristiche o vincoli.

La **rappresentazione relazionale** consente di modellare le relazioni complesse tra configurazioni del puzzle, rendendo l'implementazione più vicina alla formalizzazione matematica del problema di ricerca nello spazio degli stati.

4.2 Architettura Modulare del Sistema Prolog

4.2.1 Modulo Solver (solver.pl)

Il modulo `solver` implementa i tre algoritmi di ricerca e le strutture dati associate. Ogni algoritmo è realizzato come predicato ricorsivo che manipola strutture dati appropriate alla strategia di ricerca. Il modulo espone predicati uniformi (`solve_astar/5`, `solve_bfs/5`, `solve_greedy/5`) che accettano lo stato iniziale e restituiscono percorso, statistiche e tempi di esecuzione.

La scelta di separare i predicati per algoritmo facilita la manutenzione e permette ottimizzazioni specifiche per ciascuna strategia di ricerca. Sono incluse funzionalità ausiliarie per la generazione di puzzle casuali, la verifica di risolvibilità e la gestione della memoria.

4.2.2 Modulo Heuristics (heuristics.pl)

Il modulo `heuristics` incapsula le funzioni di valutazione, permettendo facile modifica e testing delle euristiche senza impattare gli algoritmi di ricerca. L'interfaccia del modulo espone tre diverse funzioni di valutazione implementate: la distanza di Manhattan, il conteggio delle tessere malposizionate, e un'euristica combinata che integra entrambe le metriche. Sono incluse anche funzionalità di visualizzazione degli stati per debugging e analisi.

4.3 Analisi della Knowledge Base

L'implementazione Prolog del sistema 8-puzzle realizza una Knowledge Base (KB) che organizza i fatti e le regole di inferenza. I fatti statici definiscono gli elementi fondamentali del problema, includendo obiettivi attraverso `goal_state/1`, posizioni delle tessere tramite `goal_positions/1`, e configurazioni valide del puzzle. Le regole di inferenza codificano la logica per movimenti validi, la generazione dinamica di successori, e l'implementazione delle funzioni euristiche che guidano la ricerca. I predicati dinamici gestiscono informazioni temporanee durante l'esecuzione, tracciando stati visitati, statistiche di performance, e risultati intermedi. La conoscenza procedurale è incorporata negli algoritmi di ricerca stessi, implementati come regole Prolog che codificano strategie specifiche di esplorazione dello spazio degli stati.

4.3.1 Principali Fatti della Knowledge Base

Fatti Statici:

- `goal_state/1` - Definisce lo stato obiettivo [1,2,3,4,5,6,7,8,0] (definito in solver.pl ed esportato a heuristics.pl)
- `goal_positions/1` - Lista delle posizioni obiettivo per ogni tessera

Predicati Dinamici:

- `stato_visitato/1` - Memorizza gli stati già visitati durante la ricerca
- `nodes_explored/1` - Contatore dei nodi esplorati
- `nodes_frontier/1` - Dimensione massima della frontiera
- `optimal_cost/1` - Costo ottimale trovato finora

4.3.2 Principali Predicati e Regole della Knowledge Base

Predicati di Movimento e Transizione:

- `find_blank/2` - Trova la posizione dello spazio vuoto nello stato
- `successors/2` - Genera tutti gli stati successori validi
- `move/2` - Applica una mossa valida per generare un nuovo stato
- `valid_move/2` - Determina le mosse valide dalla posizione dello spazio vuoto

Predicati Euristici:

- `manhattan_distance/2` - Calcola la distanza di Manhattan totale
- `misplaced_tiles/2` - Conta il numero di tessere fuori posto
- `combined_heuristic/2` - Euristica combinata che integra Manhattan e Misplaced

Predicati di Validazione e Generazione:

- `is_solvable/1` - Verifica se un puzzle è risolvibile (controllo parità inversioni)
- `generate_random_puzzle/2` - Genera un puzzle casuale risolvibile con difficoltà specificata

Predicati degli Algoritmi di Ricerca:

- `solve_astar/5` - Risolve il puzzle usando A* con euristica combinata
- `solve_bfs/5` - Risolve il puzzle usando Breadth-First Search
- `solve_greedy/5` - Risolve il puzzle usando Greedy Best-First Search

4.3.3 Meccanismi di Ragionamento della KB

La Knowledge Base implementa meccanismi di ragionamento che sfruttano le capacità native del motore Prolog per l'esplorazione dello spazio degli stati. L'inferenza deduttiva costituisce il meccanismo fondamentale attraverso cui il sistema deriva stati successori dalle regole di movimento, applicando trasformazioni logiche che seguono il principio per cui da uno stato valido e una mossa legale si può inferire deterministicamente un nuovo stato raggiungibile.

Il motore Prolog esplora sistematicamente lo spazio degli stati attraverso unificazione e backtracking automatici. L'unificazione fornisce matching automatico di pattern per identificare stati specifici e applicare le regole appropriate, mentre il backtracking garantisce esplorazione sistematica di tutte le alternative quando un particolare ramo di ricerca fallisce o si rivela improduttivo. Questo meccanismo elimina la necessità di gestire esplicitamente stack e strutture dati per l'esplorazione, semplificando significativamente l'implementazione degli algoritmi.

Il reasoning euristico valuta gli stati attraverso funzioni di stima specializzate. La `manhattan_distance/2` fornisce stime accurate della distanza dall'obiettivo basate sulla geometria del problema, mentre `misplaced_tiles/2` conta direttamente le tessere malposizionate per una valutazione complementare. L'euristica `combined_heuristic/2` integra queste metriche multiple per massimizzare l'informatività mantenendo l'ammissibilità teorica.

La prevenzione di cicli avviene attraverso la verifica continua di stati già visitati tramite `stato_visitato/1`, mentre la gestione efficiente della memoria è garantita dal predicato `cleanup_prolog/0` che rimuove tutte le informazioni temporanee al termine di ogni ricerca, prevenendo accumulo di memoria tra esecuzioni successive.

4.4 Rappresentazione degli Stati

Il sistema implementa la rappresentazione degli stati del puzzle utilizzando liste Prolog di 9 elementi che mappano direttamente la griglia 3×3 in formato lineare. La rappresentazione lineare permette un accesso efficiente agli elementi attraverso predicati come `nth0/3` e facilita le operazioni di pattern matching che sono fondamentali per l'unificazione Prolog.

Lo stato obiettivo `[1,2,3,4,5,6,7,8,0]` rappresenta la configurazione target con le tessere ordinate e lo spazio vuoto (0) nell'ultima posizione. Questa rappresentazione compatta minimizza l'overhead di memoria e semplifica le operazioni di confronto tra stati.

4.5 Implementazione delle Transizioni

Le transizioni tra stati sono implementate attraverso il predicato `successors/2` che genera tutti gli stati raggiungibili da una configurazione data. Il processo di generazione identifica prima la posizione dello spazio vuoto, determina le mosse valide basate su quella posizione, e applica ciascuna mossa per generare i nuovi stati.

4.5.1 Validazione delle Mosse

Il sistema implementa controlli rigorosi per garantire che solo mosse legali siano applicate. I vincoli di bordo impediscono movimenti impossibili (es. spostamento a sinistra dalla colonna 0), mentre la validazione della posizione assicura che lo spazio vuoto possa effettivamente muoversi nella direzione specificata. Il predicato `valid_moves/2` codifica queste regole per ciascuna delle 9 posizioni possibili dello spazio vuoto.

4.6 Ottimizzazioni Implementative

L'efficienza del sistema è stata ottimizzata attraverso diverse strategie implementative che bilanciano performance e chiarezza del codice.

4.6.1 Gestione della Memoria

La gestione della memoria in Prolog presenta sfide uniche dovute alla natura del garbage collection e alla persistenza delle asserzioni dinamiche. Il sistema implementa un meccanismo di cleanup che utilizza `retractall/1` per rimuovere stati visitati al termine di ogni ricerca, prevenendo accumulo di memoria tra esecuzioni successive. L'uso di predicati tail-recursive dove possibile permette al compilatore Prolog di ottimizzare la ricorsione in iterazione, riducendo l'uso dello stack.

4.6.2 Ottimizzazione delle Euristiche

Le funzioni euristiche sono state ottimizzate per minimizzare il costo computazionale mantenendo l'accuratezza. L'uso di accumulatori nei predicati ricorsivi evita ricostruzioni di liste costose, mentre il caching implicito attraverso l'unificazione riduce ricalcoli ridondanti. La pre-computazione delle posizioni obiettivo elimina lookup ripetuti durante la valutazione euristica.

4.7 Validazione

Il sistema include meccanismi robusti per la validazione della correttezza e della risolvibilità dei puzzle.

4.7.1 Controllo Risolvibilità

Il predicato `is_solvable/1` verifica che un puzzle sia risolvibile calcolando il numero di inversioni. Solo configurazioni con parità di inversioni uguale a quella dello stato obiettivo sono risolvibili. Questo controllo previene tentativi di risoluzione su configurazioni impossibili, risparmiando risorse computazionali.

4.7.2 Generazione Test Cases

Il sistema include un generatore di puzzle casuali che garantisce la risolvibilità. Il predicato `generate_random_puzzle/2` parte dallo stato obiettivo e applica una sequenza di mosse casuali valide, garantendo per costruzione che il puzzle risultante sia risolvibile. Il parametro di difficoltà controlla il numero di mosse applicate, correlando con la complessità attesa della risoluzione.

Capitolo 5

Integrazione Python-Prolog

5.1 Configurazione e Inizializzazione

L'integrazione tra Python e Prolog avviene attraverso la libreria pyswip, che fornisce un'interfaccia Foreign Function Interface (FFI) per SWI-Prolog. La classe `PuzzleLogic` gestisce l'inizializzazione del motore Prolog, il caricamento dei moduli e la gestione del ciclo di vita delle query.

L'inizializzazione segue una sequenza precisa: verifica della disponibilità di SWI-Prolog nel sistema, configurazione delle variabili d'ambiente necessarie, creazione dell'istanza Prolog con parametri ottimizzati, e caricamento dei moduli solver e heuristics. La gestione degli errori è robusta, con fallback e messaggi diagnostici dettagliati.

5.1.1 Protocollo di Comunicazione

La comunicazione tra Python e Prolog segue un protocollo ben definito che garantisce type safety e gestione errori. Le query Prolog sono costruite dinamicamente con escape appropriato dei caratteri speciali, i risultati sono parsati e convertiti in tipi Python nativi, e le eccezioni Prolog sono catturate e tradotte in eccezioni Python meaningful.

Il marshalling bidirezionale è gestito da metodi specializzati nella classe `PuzzleLogic` che convertono tra rappresentazioni Python (liste, dizionari) e termini Prolog (liste, strutture).

5.2 Conversione di Rappresentazioni

La conversione tra le rappresentazioni Python e Prolog è critica per la correttezza del sistema. Gli stati del puzzle sono convertiti da liste Python a liste Prolog e viceversa, i percorsi sono trasformati da liste di stati Prolog a sequenze di mosse Python, e le statistiche sono estratte come dizionari Python da termini Prolog strutturati.

Il sistema gestisce automaticamente le differenze di indicizzazione (0-based in Python, potenzialmente 1-based in Prolog) e i tipi di dato (interi, float, atomi).

Capitolo 6

Utilizzo del Sistema

6.1 Struttura del Progetto

Il progetto segue una struttura modulare che separa chiaramente le diverse componenti:

```
8-puzzle-solver/
|
|-- app.py                # Entry point principale
|-- requirements.txt      # Dipendenze Python
|-- README.md             # Panoramica e istruzioni sul progetto
|
|-- docs/                 # Documentazione progetto
|
|-- gui/
|   |-- puzzle_gui.py     # Interfaccia grafica Tkinter
|
|-- logic/
|   |-- puzzle_logic.py   # Logica e bridge Python-Prolog
|
|-- prolog/
|   |-- solver.pl         # Implementazione algoritmi in Prolog
|   |-- heuristics.pl     # Funzioni euristiche
|
|-- test/
|   |-- test.py           # Suite di test
|
|-- results/              # Report generati dai test e dall'app
```

6.2 Prerequisiti e Installazione

6.2.1 Verifica dei Prerequisiti

Prima dell'installazione, è necessario verificare la presenza dei componenti richiesti:

```
# Verifica versioni
python --version  # >= 3.9
swipl --version   # >= 8.4
```

Il sistema richiede Python 3.9+ e SWI-Prolog 8.4+ per garantire compatibilità con PySwip e prestazioni ottimali.

6.2.2 Clonazione e Configurazione

La procedura di installazione segue questi passaggi:

1. **Clonazione del repository**:

   ```
   git clone https://github.com/imb0ru/8puzzle-solver.git
   cd 8puzzle-solver
   ```

2. **Creazione ambiente virtuale**:

   ```
   python -m venv venv
   source venv/bin/activate       # Linux/macOS
   venv\Scripts\activate.bat       # Windows
   ```

3. **Installazione dipendenze**:

   ```
   pip install -r requirements.txt
   ```

6.3 Modalità di Esecuzione

L'applicazione principale si avvia tramite:

```
python app.py
```

Per ottenere informazioni complete sui parametri disponibili:

```
python app.py --help
```

Per l'esecuzione di test:

```
python test/test.py --puzzles N
```

dove N specifica il numero di puzzle da testare per ogni difficoltà (minimo 10, default 10). Il sistema genera automaticamente:

- N puzzle Easy (10-20 mosse dalla soluzione)
- N puzzle Medium (20-40 mosse dalla soluzione)
- N puzzle Hard (40+ mosse dalla soluzione)
- N puzzle Mixed (distribuzione casuale)

Per ogni puzzle generato, tutti gli algoritmi (A*, BFS, Greedy) vengono testati, producendo un totale di N×4×3 test individuali.

6.4 Formati di Output

Il sistema genera automaticamente output in tre formati:

File CSV (Dati Grezzi)

Contiene tutti i dati di test individuali per analisi statistica avanzata:

```
difficulty_category,puzzle_id,algorithm,success,time,path_length,
nodes_explored,nodes_frontier,memory,puzzle_manhattan
```

File JSON (Statistiche Aggregate)

Metriche aggregate per algoritmo in formato machine-readable:

```
{
  "astar": {
    "success_rate": 100.0,
    "avg_time": 0.021,
    "avg_moves": 9.4,
    "avg_nodes": 70,
    "total_tests": 4000,
    "successful_tests": 4000
  }
}
```

Report Markdown (Human-Readable)

Documenti formattati con tabelle comparative, analisi per difficoltà e conclusioni:

- Sommario esecutivo con overview dei test
- Tabelle comparative delle prestazioni globali
- Analisi dettagliata per ogni livello di difficoltà
- Identificazione del miglior algoritmo per diversi criteri

6.5 Troubleshooting

6.5.1 Problemi Comuni

Errori di Integrazione PySwip:

```
# Verifica installazione SWI-Prolog
swipl --version
# Reinstallazione PySwip
pip uninstall pyswip
pip install pyswip --no-cache-dir
```

Problemi di Performance: - Verificare disponibilità di memoria per test su larga scala - Considerare riduzione del numero di test per hardware limitato

Errori di Parsing Output: - Verificare encoding dei file (UTF-8)

Capitolo 7

Testing e Valutazione

7.1 Framework di Testing Sistematico

Il framework di testing implementato permette la valutazione quantitativa e comparativa degli algoritmi attraverso metriche standardizzate. La classe `PuzzleTest` orchestra l'intero processo di testing, dalla generazione dei dataset alla produzione dei report finali.

Il framework garantisce riproducibilità attraverso seed configurabili per la generazione casuale, completezza delle metriche con raccolta di tempi, mosse, nodi e memoria, e facilità di analisi attraverso export in formati standard (CSV, JSON, Markdown).

7.2 Generazione Dataset e Metriche

La generazione di puzzle utilizza una strategia di scrambling controllato dallo stato obiettivo, garantendo risolvibilità per costruzione. Le difficoltà sono mappate a intervalli di mosse: easy (10-20), medium (20-40), hard (40-80). Il sistema raccoglie metriche complete includendo tempo di esecuzione, lunghezza soluzione, nodi esplorati, dimensione frontiera e utilizzo memoria stimato.

7.3 Risultati Sperimentali

Basandosi sui test eseguiti sui file presenti nella cartella `results/`, emerge un quadro prestazionale chiaro degli algoritmi implementati. Il sistema è stato sottoposto a sei sessioni di test distinte, ciascuna con caratteristiche specifiche per valutare diversi aspetti delle prestazioni algoritmiche.

7.3.1 Test 20250817_190151 (40 puzzle)

Primo test di validazione su un campione ridotto (10 puzzle per difficoltà):

Tabella 7.1: Risultati globali Test 20250817_190151

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	15ms	9.2	50
BFS	1.27s	9.2	5040
Greedy	11ms	22.6	60

Tabella 7.2: Stratificazione per difficoltà Test 20250817_190151

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	30ms	5.0	6	2ms	5.0	96	¡1ms	5.0	6
Medium	3ms	10.2	33	1.78s	10.2	6304	6ms	22.2	59
Hard	15ms	12.0	94	2.51s	12.0	9513	15ms	31.2	79
Mixed	13ms	9.7	68	807ms	9.7	4250	21ms	32.1	96

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.2 Test 20250817_190210 (100 puzzle)

Test su campione intermedio (25 puzzle per difficoltà):

Tabella 7.3: Risultati globali Test 20250817_190210

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	35ms	9.7	73
BFS	8.08s	9.7	8261
Greedy	15ms	25.5	76

Tabella 7.4: Stratificazione per difficoltà Test 20250817_190210

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	2ms	6.4	8	5ms	6.4	211	6ms	13.4	32
Medium	2ms	9.5	18	783ms	9.5	2276	10ms	22.7	67
Hard	103ms	14.2	193	27.70s	14.0	24324	31ms	45.1	146
Mixed	35ms	8.8	72	3.82s	8.8	6233	11ms	21.0	59

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.3 Test 20250817_190219 (200 puzzle)

Test di conferma su campione esteso (50 puzzle per difficoltà):

Tabella 7.5: Risultati globali Test 20250817_190219

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	18ms	9.4	64
BFS	6.01s	9.4	7466
Greedy	14ms	23.9	75

Tabella 7.6: Stratificazione per difficoltà Test 20250817_190219

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	1ms	5.8	9	26ms	5.8	475	6ms	10.7	34
Medium	2ms	8.9	22	157ms	8.8	1681	10ms	20.2	63
Hard	63ms	13.9	184	19.29s	13.8	21031	32ms	42.1	144
Mixed	5ms	9.2	40	4.55s	9.2	6676	9ms	22.4	60

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.4 Test 20250817_190226 (400 puzzle)

Test di scalabilità (100 puzzle per difficoltà):

Tabella 7.7: Risultati globali Test 20250817_190226

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	50ms	9.5	64
BFS	7.19s	9.5	7782
Greedy	14ms	25.1	73

Tabella 7.8: Stratificazione per difficoltà Test 20250817_190226

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	1ms	5.5	7	8ms	5.5	223	3ms	11.3	25
Medium	2ms	8.4	19	393ms	8.4	1624	5ms	17.8	41
Hard	32ms	13.8	125	14.87s	13.8	19103	29ms	43.2	147
Mixed	167ms	10.4	104	13.47s	10.4	10175	19ms	28.0	80

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.5 Test 20250817_190233 (2000 puzzle)

Test intensivo (500 puzzle per difficoltà):

Tabella 7.9: Risultati globali Test 20250817_190233

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	39ms	9.4	71
BFS	4.40s	9.4	7277
Greedy	13ms	25.4	76

Tabella 7.10: Stratificazione per difficoltà Test 20250817_190233

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	1ms	5.4	8	20ms	5.4	227	3ms	8.7	18
Medium	6ms	8.8	28	847ms	8.8	2561	10ms	23.1	65
Hard	126ms	13.9	175	12.16s	13.9	18277	28ms	44.7	146
Mixed	23ms	9.4	74	4.60s	9.4	8063	11ms	25.0	76

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.6 Test 20250817_190238 (4000 puzzle)

Test completo stratificato (1000 puzzle per difficoltà):

Tabella 7.11: Risultati globali Test 20250817_190238

Algoritmo	Tempo Medio	Mosse Medie	Nodi Esplorati
A*	19ms	9.4	64
BFS	3.79s	9.4	7089
Greedy	12ms	25.3	76

Tabella 7.12: Analisi dettagliata per difficoltà - Test 20250817_190238

Difficoltà	A*			BFS			Greedy		
	T	M	N	T	M	N	T	M	N
Easy	1ms	5.3	7	7ms	5.3	185	3ms	8.6	18
Medium	4ms	8.8	27	672ms	8.8	2344	11ms	23.5	68
Hard	51ms	14.0	154	11.24s	14.0	18717	23ms	42.9	138
Mixed	21ms	9.4	67	3.25s	9.4	7110	10ms	26.1	79

Nota: T = tempo medio, M = mosse medie, N = nodi esplorati.

7.3.7 Trend Temporali

L'analisi delle performance temporali attraverso i sei test progressivi (da 40 a 4000 puzzle) rivela pattern di convergenza caratteristici per ciascun algoritmo, con lievi variazioni tra i diversi test.

A* manifesta stabilità notevole con tempi medi che variano da 15ms (test piccoli) a 50ms (test di 400 puzzle), convergendo intorno ai 19-39ms per campioni di dimensioni significative (2000-4000 puzzle). La consistenza delle performance conferma l'efficacia dell'euristica combinata nel guidare la ricerca verso stati promettenti.

BFS mostra comportamento più variabile, da 1.27s per piccoli campioni fino a 8.08s per campioni medi, stabilizzandosi intorno ai 3.79-4.40 secondi per campioni grandi (2000-4000 puzzle). Questo riflette la crescita esponenziale del tempo con la complessità del puzzle.

Greedy presenta le migliori performance temporali assolute e più consistenti, con tempi medi stabili tra 11-15ms indipendentemente dalla dimensione del campione. L'utilizzo dell'euristica combinata garantisce sia velocità che qualità accettabile delle soluzioni.

7.3.8 Evoluzione delle Stratificazioni per Difficoltà

L'analisi dettagliata per livelli di difficoltà evidenzia comportamenti algoritmici distintivi con valori coerenti attraverso i diversi test:

Puzzle Easy (5-6 mosse ottimali): Tutti gli algoritmi convergono rapidamente. BFS mostra overhead contenuto (2-26ms), mentre A* oscilla tra 1-30ms e Greedy tra ¡1-6ms. La differenza in nodi esplorati è già marcata: A* 7-9 nodi, BFS 96-475 nodi, Greedy 6-34 nodi. Interessante notare come per puzzle molto semplici, Greedy eguagli l'ottimalità.

Puzzle Medium (8-10 mosse ottimali): Emergono differenze significative. BFS richiede da 157ms a 1.78s esplorando 1600-6300 nodi. A* mantiene tempi eccellenti (2-6ms) esplorando solo 18-33 nodi. Greedy resta veloce (5-11ms) ma la qualità delle soluzioni degrada moderatamente (18-24 mosse vs 9 ottimali).

Puzzle Hard (13-15 mosse ottimali): Le differenze diventano drammatiche. BFS richiede 2.51-27.70s esplorando 9500-24300 nodi. A* richiede 15-126ms mantenendo l'ottimalità con 94-193 nodi esplorati. Greedy trova soluzioni in 15-32ms con percorsi di 31-45 mosse, circa 2.7-3x l'ottimo.

Puzzle Mixed (9-10 mosse ottimali): Questa categoria bilanciata mostra A* con performance stabili (5-167ms) esplorando 40-104 nodi. BFS richiede 0.8-13.47s con 4200-10175 nodi esplorati. Greedy mantiene tempi rapidi (9-21ms) con soluzioni di lunghezza 21-32 mosse, dimostrando buona consistenza su distribuzioni eterogenee.

7.3.9 Analisi Comparativa Complessiva

I risultati sperimentali confermano le previsioni teoriche sulle caratteristiche degli algoritmi con precisione:

Consistenza Temporale

A* dimostra scalabilità lineare rispetto alla difficoltà del puzzle, con factor di crescita prevedibile. BFS mostra crescita esponenziale del tempo con la profondità della soluzione. Greedy mantiene tempo quasi costante indipendentemente dalla complessità, grazie all'euristica combinata che guida efficacemente la ricerca.

Efficacia Euristica

L'euristica combinata utilizzata sia da A* che da Greedy mostra eccellente efficacia nel ridurre lo spazio di ricerca. Per A, questo si traduce in una riduzione del 99.1% dei nodi esplorati rispetto a BFS mantenendo l'ottimalità. Per Greedy, l'euristica combinata fornisce un buon bilanciamento tra velocità di convergenza e qualità delle soluzioni, con percorsi che sono mediamente 2.7x più lunghi dell'ottimo ma trovati in tempi trascurabili. Il rapporto nodi esplorati A/BFS varia da 1:20-26 (easy) a 1:120-121 (hard), evidenziando l'importanza crescente dell'euristica con la complessità.

Capitolo 8

Conclusioni e Sviluppi Futuri

8.1 Osservazioni sui Risultati Sperimentali

L'analisi empirica condotta su oltre 12.000 istanze di puzzle ha fornito una validazione quantitativa delle previsioni teoriche sulle caratteristiche degli algoritmi implementati. I risultati ottenuti permettono di trarre alcune osservazioni di carattere generale sul comportamento degli algoritmi di ricerca nel dominio dell'8-puzzle.

I test hanno confermato la correttezza delle complessità computazionali attese. **A*** ha dimostrato il comportamento ottimale previsto dalla teoria, con una complessità temporale che cresce linearmente con la lunghezza del percorso ottimo quando guidato da un'euristica ammissibile e consistente. La natura esponenziale di **BFS** è emersa chiaramente nei test su puzzle di difficoltà crescente, mentre **Greedy** ha manifestato l'instabilità tipica degli algoritmi non ottimali ma con tempi di esecuzione sostanzialmente costanti.

Un risultato particolarmente significativo riguarda il valore dell'informazione euristica nella riduzione dello spazio di ricerca. L'euristica combinata Manhattan-Misplaced ha permesso ad A* di esplorare solo una frazione minima dello spazio degli stati rispetto all'esplorazione sistematica di BFS. Questo vantaggio cresce in modo non lineare con la complessità del problema, confermando l'importanza crescente dell'informazione euristica per problemi di dimensioni maggiori.

L'analisi ha rivelato tre distinti profili algoritmici che corrispondono a diversi scenari applicativi:

- **Scenario critico per l'ottimalità**: Quando la qualità della soluzione è prioritaria, A* emerge come scelta obbligata, offrendo garanzie di ottimalità con overhead computazionale accettabile.
- **Scenario con risorse limitate**: In contesti real-time o embedded dove il tempo di risposta è critico, Greedy fornisce soluzioni immediate con qualità degradata ma prevedibile.
- **Scenario di verifica**: BFS mantiene valore come baseline per la validazione della correttezza delle soluzioni ottimali e come riferimento per misurare l'efficacia delle ottimizzazioni euristiche.

8.2 Conclusioni del Progetto

Il progetto ha raggiunto con successo tutti gli obiettivi prefissati, dimostrando l'applicabilità pratica dei concetti dell'Ingegneria della Conoscenza a un problema classico dell'AI. L'implementazione multi-paradigma ha evidenziato come diversi approcci computazionali possano cooperare efficacemente, sfruttando i punti di forza di ciascun linguaggio.

Dal punto di vista didattico, il sistema fornisce uno strumento concreto per comprendere le differenze tra algoritmi di ricerca informata e non informata, l'importanza delle euristiche ammissibili nella ricerca ottimale, e i trade-off tra completezza, ottimalità ed efficienza. L'interfaccia grafica rende accessibili concetti algoritmici complessi attraverso visualizzazione intuitiva e feedback immediato.

Dal punto di vista tecnico, il progetto dimostra l'efficacia del paradigma logico per problemi di ricerca nello spazio degli stati, la fattibilità dell'integrazione tra linguaggi di paradigmi diversi, e l'importanza dell'ingegneria del software nella realizzazione di sistemi AI robusti. La modularità dell'architettura facilita estensioni e modifiche future.

L'implementazione ha affrontato e risolto diverse sfide tecniche significative. La gestione efficiente della memoria in Prolog attraverso cleanup sistematico delle asserzioni dinamiche, l'ottimizzazione del marshalling Python-Prolog per minimizzare l'overhead di comunicazione, la sincronizzazione tra GUI e motore di inferenza per responsività dell'interfaccia, e la validazione della correttezza algoritmica attraverso testing sistematico.

8.3 Sviluppi Futuri

Il sistema presenta numerose opportunità di estensione e miglioramento che possono essere esplorate seguendo diverse direzioni di ricerca e sviluppo.

Una delle estensioni più naturali riguarda la generalizzazione dell'architettura per supportare puzzle di dimensioni arbitrarie (NxN), che richiederebbe un adattamento automatico delle euristiche per mantenere efficacia e ammissibilità. Parallelamente, lo sviluppo di euristiche adattive che si auto-calibrano basandosi su pattern di esplorazione storici potrebbe migliorare significativamente le performance su domini specifici.

L'utilizzo di reti neurali per la predizione di mosse ottimali potrebbe accelerare drammaticamente la ricerca, mentre approcci di reinforcement learning permetterebbero lo sviluppo di agenti che apprendono strategie di risoluzione attraverso l'interazione diretta con il dominio. L'identificazione automatica di configurazioni caratteristiche tramite pattern recognition consentirebbe inoltre una specializzazione euristica dinamica, adattando le strategie di ricerca alle specificità del problema.

Il progetto rappresenta una solida base per ulteriori esplorazioni nell'ambito dell'Ingegneria della Conoscenza, offrendo un framework estensibile per lo studio di algoritmi di ricerca e tecniche di AI. La combinazione di rigore teorico, implementazione pratica e validazione empirica lo rende uno strumento prezioso sia per la didattica che per la ricerca.

La natura open-source del progetto invita contributi dalla comunità accademica e professionale, con potenziale per evolversi in una risorsa educativa di riferimento per lo studio degli algoritmi di ricerca e dell'intelligenza artificiale applicata. La documentazione completa e il codice ben strutturato facilitano l'adozione e l'estensione da parte di altri ricercatori e studenti.