Teoria dei giochi e intelligenza artificiale: il caso del gioco Quoridor

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione
a.a. 2025/2026

Andrea Altieri, Andrea Contursi, Andrea Flaiani

Sommario

Indice delle figure

1 Introduzione e Contesto Teorico

Negli ultimi anni la teoria dei giochi è diventata uno degli strumenti concettuali più importanti per descrivere e analizzare situazioni in cui più agenti – umani o artificiali – interagiscono in modo strategico. In ambito informatico, e in particolare nell'intelligenza artificiale, i giochi da tavolo astratti sono stati spesso utilizzati come "laboratorio" controllato in cui mettere alla prova algoritmi di ricerca, pianificazione e apprendimento: gli esempi classici sono gli scacchi, la dama, il Go. In questo lavoro si inserisce un gioco meno noto al grande pubblico ma molto interessante dal punto di vista computazionale: Quoridor.

L'obiettivo del nostro progetto è progettare e realizzare un agente artificiale in grado di giocare a Quoridor in modo competente, sfruttando concetti e tecniche tipiche dell'IA per giochi a due giocatori: modellazione come problema di ricerca, definizione di funzioni di valutazione, utilizzo di algoritmi come minimax o sue varianti, ecc. Per dare un quadro coerente al progetto, in questo capitolo si propone innanzitutto un breve richiamo alla teoria dei giochi, si presenta il gioco Quoridor e le sue regole, se ne discute la struttura e la complessità, e infine si esplicitano gli obiettivi specifici della nostra implementazione.

1.1 L'Intelligenza Artificiale e la Teoria dei Giochi

La **teoria dei giochi** studia le interazioni strategiche tra più agenti razionali, ovvero situazioni in cui l'esito per ciascun partecipante dipende non solo dalle proprie decisioni, ma anche da quelle altrui. Formalmente, un gioco è spesso descritto specificando:

- un insieme di **giocatori**;
- un insieme di **stati** e di **mosse** possibili in ciascuno stato;
- delle **regole di transizione**, che indicano come le mosse trasformano lo stato corrente in uno nuovo;
- una **funzione di utilità** (o payoff) che assegna un valore a ciascun esito possibile per ogni giocatore.

In questo contesto si analizzano le **strategie**, cioè regole che dicono al giocatore che azione intraprendere in ogni situazione, e si cercano concetti di equilibrio e di ottimalità, che descrivono comportamenti ragionevoli per agenti razionali.
Una prima distinzione classica è tra **giochi cooperativi** e **giochi non cooperativi**: nei primi i giocatori possono formare coalizioni vincolanti, nei secondi ciascuno agisce per conto proprio. Un'altra distinzione importante è quella tra giochi a **somma zero** e giochi a somma generale: in un gioco a somma zero il guadagno di un giocatore corrisponde esattamente alla perdita dell'altro; è il caso tipico dei giochi competitivi "puri" come scacchi, dama, Go e anche Quoridor.
Dal punto di vista dell'informazione, si distinguono inoltre i giochi a **informazione perfetta**, in cui tutti gli agenti conoscono completamente lo stato corrente (non ci sono carte coperte, elementi nascosti o casualità non ancora rivelata), dai giochi a informazione imperfetta o incompleta. I giochi a informazione perfetta e senza aleatorietà sono particolarmente adatti a essere affrontati con algoritmi di ricerca nello spazio degli stati, perché, in linea di principio, è possibile costruire (almeno concettualmente) l'intero albero delle partite possibili.
Quoridor rientra esattamente in questa famiglia: è un gioco **discreto**, **deterministico**, **a due giocatori**, **a somma zero** e a **informazione perfetta**, e quindi un esempio di gioco combinatorio tradizionale in cui si possono applicare tecniche standard di intelligenza

artificiale per giochi, come minimax, alpha-beta pruning, ricerca con profondità limitata e funzioni di valutazione euristica.

1.2 Descrizione del gioco Quoridor

Quoridor è un gioco da tavolo astratto ideato da Mirko Marchesi e pubblicato per la prima volta alla fine degli anni '90. Nella sua versione più comune, il gioco si gioca su una **scacchiera 9×9** (81 caselle) ed è adatto a **due o quattro giocatori**; in questo progetto ci concentriamo sulla versione a **due giocatori**, che è più semplice da modellare e studiare in ottica di IA.
Ogni giocatore controlla un singolo **pedone**, che all'inizio della partita è posizionato al centro della propria linea di base: nel caso standard, il primo giocatore parte dalla casella centrale della fila inferiore, il secondo dalla casella centrale della fila superiore. Oltre al pedone, ogni giocatore dispone di un numero fissato di **barriere** (o "fences", "walls"), tipicamente 10 ciascuno nella versione a due giocatori.
L'**obiettivo** del gioco è molto semplice da enunciare: essere il primo a portare il proprio pedone su una qualunque casella della linea opposta rispetto a quella di partenza.
A ogni turno, un giocatore deve scegliere una tra due azioni possibili:

1. **Muovere il pedone**:
    - Il pedone si sposta di **una casella alla volta** in senso ortogonale (su, giù, sinistra, destra), ma **mai in diagonale**.
    - Il movimento è vincolato dalla presenza delle barriere, che non possono essere attraversate: il pedone deve quindi "aggirare" i muri posizionati sulla scacchiera.
    - Se i due pedoni si trovano su caselle adiacenti frontalmente e non vi è alcuna barriera tra di loro, il giocatore di turno può **saltare** il pedone avversario, avanzando di due caselle in linea retta. Se invece dietro il pedone avversario è presente una barriera o il bordo del tabellone, è possibile spostarsi in una delle caselle laterali accanto al pedone avversario.
2. **Posizionare una barriera**:
    - Le barriere sono segmenti che occupano lo spazio tra due coppie di caselle adiacenti e vengono posizionate lungo la griglia, "chiudendo" parzialmente un passaggio.
    - Una volta posizionata, la barriera **rimane fissa** per tutta la partita: non può essere rimossa né spostata da nessuno dei due giocatori.
    - Le barriere possono essere usate in maniera **difensiva**, per rallentare o deviare il percorso dell'avversario, oppure in modo **offensivo**, per ottimizzare il proprio tragitto creando corridoi favorevoli.
    - È vietato posizionare barriere in modo tale da **bloccare completamente** uno dei giocatori: deve sempre esistere almeno un cammino possibile, anche se molto tortuoso, che consenta a ciascun pedone di raggiungere il lato opposto.

La partita termina non appena uno dei due pedoni raggiunge una qualunque casella della riga opposta rispetto alla sua linea di partenza; quel giocatore viene dichiarato vincitore. La durata tipica di una partita reale è relativamente breve (10–20 minuti), ma come si vedrà tra poco lo spazio delle configurazioni possibili è estremamente ampio.

1.3 Analisi del gioco: struttura e complessità

Dal punto di vista dell'intelligenza artificiale, Quoridor è interessante perché combina due componenti complementari: da un lato la ricerca di un percorso più breve verso il traguardo, dall'altro l'uso strategico delle barriere per modificare la struttura del grafo su cui tale ricerca avviene. In altre parole, ogni mossa influisce sia sullo stato attuale del gioco sia sull'evoluzione futura degli spazi di movimento per entrambi i giocatori.

In termini di teoria dei giochi e di complessità, Quoridor è classificabile come **gioco deterministico, sequenziale, a due giocatori, a somma zero e a informazione perfetta**. Ciò lo colloca nella stessa famiglia di giochi come scacchi e Go dal punto di vista formale, ma con regole molto più semplici da spiegare a un giocatore umano.

Studi specifici sulla complessità del gioco hanno mostrato che la **state-space complexity** (cioè il numero di posizioni legalmente raggiungibili) e la **game-tree complexity** (il numero di partite possibili) di Quoridor sono estremamente elevate. Una stima classica quantifica la complessità dello spazio degli stati intorno a 10^42 e quella dell'albero delle partite addirittura intorno a 10^162, valori paragonabili e in alcuni casi superiori a quelli degli scacchi. In tabelle comparative sulla complessità dei giochi, Quoridor risulta avere una state-space complexity simile a quella degli scacchi ma una game-tree complexity addirittura più alta, e viene generalmente collocato fra i giochi "difficili" da risolvere esattamente.

Per effetto di questa complessità, un approccio di **brute force** che cerchi di esplorare in modo esaustivo tutte le possibili sequenze di mosse è impraticabile, anche con hardware moderno: è necessario ricorrere a strategie di **potatura dell'albero di ricerca** e **a funzioni di valutazione** che stimino rapidamente la bontà di una posizione senza arrivare fino alle foglie dell'albero. In letteratura sono stati proposti diversi agenti per Quoridor basati su minimax con potatura alpha-beta, arricchiti da euristiche che combinano elementi come la lunghezza minima del percorso verso il traguardo, il numero di barriere rimanenti e la presenza di "collo di bottiglia" creati sulla scacchiera.

Dal punto di vista strutturale, un elemento chiave è che la scacchiera di Quoridor può essere vista come un grafo i cui nodi sono le caselle e i cui archi rappresentano mosse legali (che però dipendono dinamicamente dalla disposizione delle barriere). Questo apre naturalmente la strada all'impiego di algoritmi classici di **pathfinding** come BFS o A*, utilizzati sia per calcolare il cammino minimo per ciascun giocatore sia come componente di una funzione di valutazione in un algoritmo di ricerca su albero.

Infine, va notato che, nonostante la regola che impedisce di bloccare completamente l'avversario, la possibilità di posizionare barriere consente una notevole varietà di configurazioni: la partita può assumere fasi più "tattiche", in cui si gioca principalmente sul posizionamento dei muri, e fasi più "corsistiche", in cui la priorità è correre il più velocemente possibile lungo il corridoio migliore. Questa alternanza di fasi rende il gioco dinamico e complica ulteriormente la definizione di una funzione di valutazione efficace e general-purpose.

1.4 Obiettivi del progetto

Alla luce di quanto descritto, il progetto si pone l'obiettivo di **progettare e implementare un agente artificiale in grado di giocare a Quoridor** in maniera ragionevolmente competitiva, partendo dai concetti visti nel corso di intelligenza artificiale. Più nel dettaglio, gli obiettivi principali possono essere così sintetizzati:

1. **Modellazione formale del gioco**
    - Definire una rappresentazione interna dello stato di gioco (posizioni dei pedoni, disposizione delle barriere, numero di barriere rimanenti).
    - Implementare il generatore di mosse legali, tenendo conto di tutte le regole, inclusa la verifica che almeno un cammino verso il traguardo rimanga sempre disponibile per entrambi i giocatori.
2. **Definizione e implementazione degli algoritmi di ricerca**
    - Modellare Quoridor come problema di ricerca su albero, esplorando approcci come **minimax** con o senza **potatura alpha-beta**.
    - Introdurre limiti di profondità e/o limiti di tempo adeguati a ottenere una giocabilità interattiva, compatibilmente con la complessità del gioco.
3. **Progettazione di funzioni di valutazione euristica**
    - Sviluppare una o più funzioni di valutazione che stimino la qualità di una posizione tenendo conto, ad esempio, della lunghezza del cammino minimo verso il traguardo per ciascun giocatore, del numero di barriere disponibili e della possibilità di creare o distruggere "colli di bottiglia" strategici.
    - Confrontare diverse scelte di euristiche, valutandone l'impatto sulle prestazioni dell'agente in termini di forza di gioco e tempi di risposta.
4. **Analisi sperimentale e valutazione dei risultati**
    - Svolgere partite di test tra agenti con configurazioni differenti (profondità di ricerca, euristiche, parametri) e confrontare le prestazioni.
    - Confrontare l'agente con giocatori umani, per valutare in modo qualitativo la qualità delle decisioni e la "stile" di gioco dell'IA.

In sintesi, Quoridor rappresenta un caso di studio ideale per mettere in pratica i concetti di teoria dei giochi e di intelligenza artificiale visti nel corso: pur avendo regole semplici, il gioco nasconde una notevole profondità strategica e una complessità combinatoria paragonabile a quella di giochi molto più famosi. Il progetto si propone quindi non solo di costruire un agente che "gioca bene", ma anche di utilizzare questo contesto per comprendere più a fondo le difficoltà e le potenzialità dei metodi di ricerca e valutazione applicati ai giochi a due giocatori.

2 Richiami Teorici

2.1 Algoritmo Minimax

L' algoritmo Minimax è un pilastro fondamentale dell'intelligenza artificiale classica e della teoria dei giochi. Formalizzato matematicamente per la prima volta da John von Neumann nel 1928, questo algoritmo di backtracking è progettato per risolvere problemi decisionali complessi in contesti competitivi. Il suo obiettivo primario è determinare la mossa ottimale per un giocatore, operando sotto il presupposto cruciale della razionalità perfetta: si assume, cioè, che anche l'avversario sia in grado di prevedere le mosse future e che giochi sempre per massimizzare il proprio vantaggio. L'algoritmo trova la sua applicazione ideale nei cosiddetti giochi a somma zero, dove il guadagno di un partecipante corrisponde esattamente alla perdita dell'altro (come negli scacchi, nel tris o nel Go). In questi scenari, non esiste spazio per la cooperazione: ogni mossa è un tentativo di sottrarre risorse o posizioni all'avversario. Inoltre, il Minimax eccelle nei giochi a informazione perfetta, ovvero dove entrambi i giocatori hanno piena visibilità dello stato attuale del gioco e di tutte le mosse possibili, eliminando l'incertezza tipica dei giochi di carte o di fortuna, i cosidetti giochi non deterministici.
Di seguito si riassumono le fasi principali di questo algoritmo:

1. **Generazione dello Spazio degli Stati**: Si costruisce l'albero delle mosse espandendo ogni nodo a partire dallo stato attuale. Idealmente, l'espansione procede fino ai nodi terminali (vittoria, sconfitta o pareggio). Tuttavia, nei giochi complessi, l'espansione viene limitata a una profondità massima fissata per ragioni di tempo e memoria.
2. **Valutazione degli Stati Foglia**: Si applica la funzione di utilità **U(s)** ai nodi foglia **(s)**. Se la ricerca è stata interrotta prima della fine del gioco, si applica una **funzione euristica** che stima quanto sia vantaggiosa quella posizione specifica per il giocatore MAX.
3. **Fase di Backpropagation**: Si risale l'albero calcolando i valori dei nodi superiori (genitori) in base ai valori dei nodi inferiori (figli), seguendo la logica della scelta ottimale, ossia:
   - Nei nodi **MAX:** L'utilità del nodo superiore è data dal valore massimo tra i suoi figli:

     $$U(nodo\_sup) = MAX\{U(nodo\_inf)$$

     Qui MAX sceglie la mossa che massimizza il proprio guadagno.
   - Nei nodi **MIN**: L'utilità del nodo superiore è data dal valore **minimo** tra i suoi figli:

     $$U(nodo\_inf) = Min\{U(nodo\_inf)$$

     Qui MIN sceglie la mossa che minimizza il guadagno dell'avversario (ovvero massimizza il proprio).
4. **Decisione Finale:** Una volta che i valori sono risaliti fino alla radice dell'albero, il giocatore MAX seleziona la mossa che porta al nodo figlio con il valore di utilità più alto.

$$\begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node} \end{cases}$$

Figura 1 - Funzione di Valutazione Ricorsiva usata nel Minimax

Quindi possiamo concludere che in pratica il lavoro dell'algoritmo minimax è di esplorare tutti i possibili nodi dell'albero di gioco: il numero medio di nodi figlio per ogni nodo in esame è detto fattore di diramazione, ed è pari al numero medio di mosse possibili in una generica posizione del gioco. Pertanto, il numero di nodi da valutare cresce esponenzialmente con la profondità di ricerca ed è pari al fattore di diramazione elevato alla profondità di ricerca. La complessità computazionale dell'algoritmo minimax quindi è **NP completa**, il che lo rende poco pratico per ottenere una soluzione definitiva per una certa tipologia di giochi. Nel nostro caso, lo spazio delle mosse in Quoridor è immenso. A ogni turno, un giocatore può muovere il proprio pedone in più direzioni o piazzare una barriera in molte posizioni diverse. Questo crea un fattore di ramificazione molto alto. Prendendo una stima molto conservativa:

- Fattore di ramificazione: 60 (stima bassa)
- Profondità: 30 turni totali (partita breve)

Valutare quindi tutte le possibili sequenze fino alla fine della partita richiederebbe una potenza di calcolo astronomica, oltre a un'esponenziale quantità di memoria necessaria per memorizzare la struttura dell'albero. Anche per un computer, l'algoritmo Minimax "puro" è troppo lento per prendere decisioni in tempi utili.

2.2 Alpha-beta Pruning

Come evidenziato nel paragrafo precedente, la crescita esponenziale dello spazio di ricerca in determinati giochi (come appunto Quoridor) rende impraticabile l'esplorazione completa di tutti i rami dell'albero di gioco con l'algoritmo Minimax base. Per gestire questa complessità computazionale, è necessario introdurre tecniche di potatura (*pruning*) che permettano di eliminare sezioni dell'albero di ricerca senza compromettere l'ottimalità della soluzione finale. La potatura alpha-beta raggiunge questo obiettivo sfruttando una semplice intuizione: se una mossa è chiaramente peggiore di un'alternativa già scoperta, non vale la pena esplorarne ulteriormente le conseguenze. In particolare, i l cuore dell'algoritmo risiede) nella gestione di due parametri che rappresentano i limiti di utilità per l'esplorazione:

- **Alpha (α):** è il miglior valore che il massimizzatore può attualmente garantire a quel livello o ai livelli superiori. Inizialmente è impostato a - ∞.
- **Beta(β):** è il miglior valore che il minimizzatore può attualmente garantire a quel livello o ai livelli inferiori. Inizialmente è impostato a + ∞.

La logica del processo **continuo del rapporto tra queste due soglie:**

nel momento esatto in cui il valore di Alpha diventa maggiore o uguale a quello di Beta, scatta il criterio di interruzione. Questa condizione indica che abbiamo trovato un percorso che il giocatore avversario non ci permetterebbe mai di imboccare, poiché egli ha già a disposizione un'alternativa migliore per i suoi interessi in un altro ramo precedentemente analizzato. Questa logica si traduce operativamente nei seguenti passaggi di valutazione:

1. **Valutazione dei Nodi MAX (Il Massimizzatore):** Per ogni mossa possibile (nodo figlio), il Massimizzatore cerca di alzare la sua asticella del profitto aggiornando Alfa tramite la formula:

   $$\alpha = \text{Max}(\alpha, \text{valore}).$$

   Tuttavia, se durante questa ricerca scopre che Alfa ha raggiunto o superato il valore di Beta (**$\alpha \geq \beta$**), smette immediatamente di valutare le restanti mosse in quel ramo. Questo evento, noto come taglio Beta o come β-cutoff, avviene perché il Massimizzatore ha trovato un'opzione talmente vantaggiosa che il Minimizzatore, nei livelli superiori dell'albero, agirà sicuramente per impedirgli di raggiungerla, rendendo inutile ogni ulteriore calcolo.
2. **Valutazione dei Nodi MIN (Il Minimizzatore):** Simmetricamente la logica è la stessa descritta precedentemente, il Minimizzatore analizza i suoi figli cercando di abbassare il punteggio tramite $\beta = \text{Min}(\beta, \text{valore})$. Se in questo processo il valore di Beta scende al di sotto o eguaglia quello di Alfa **($\beta \leq \alpha$)**, scatta il "taglio Alfa" (o α-cutoff). L'algoritmo interrompe la valutazione dei figli rimanenti poiché è chiaro che il Massimizzatore, ai livelli superiori, non sceglierebbe mai di scendere in questo ramo, avendo già garantita una mossa migliore altrove. In entrambi i casi, l'interruzione permette di scartare scenari irrealistici e ottimizzare drasticamente la ricerca della mossa ottimale.

L'efficacia della potatura Alfa-Beta è strettamente legata all'ordine in cui vengono esaminate le mosse, poiché la condizione di interruzione tra le due soglie si attiva con frequenza diversa a seconda della qualità dei rami analizzati per primi. Nel caso peggiore, ovvero quando le mosse vengono valutate nell'ordine meno favorevole per il giocatore di turno, la potatura risulta quasi inefficace e l'algoritmo è costretto a esplorare l'intero albero, mantenendo la complessità esponenziale tipica del Minimax puro, espressa come $O(b^m)$. Al contrario, in uno scenario di ordinamento perfetto dove la mossa migliore viene individuata immediatamente in ogni nodo, la potatura diventa massimamente aggressiva e la complessità temporale si riduce drasticamente a $O(b^{\frac{m}{2}})$. Questo miglioramento matematico permette all'algoritmo di raddoppiare la profondità della propria ricerca a parità di risorse, trasformando la scansione delle mosse da un calcolo esaustivo a una selezione strategica estremamente rapida.

2.3 Random Search

L'algoritmo Random Search è una tecnica semplice che non si basa su calcoli complessi per decidere la direzione migliore. Invece di analizzare approfonditamente la posizione (come fa il Minimax), si affida a tentativi perlopiù casuali per trovare una mossa decente a breve termine. Quest'algoritmo funziona a "step" o turni di decisione. Ad ogni step, il processo è il seguente:

1. **Genera opzioni casuali**: Dalla posizione corrente di gioco, l'algoritmo algoritmo non effettua una ricerca esaustiva, ma genera un sottoinsieme di mosse candidate selezionate casualmente. Questo campionamento riduce drasticamente

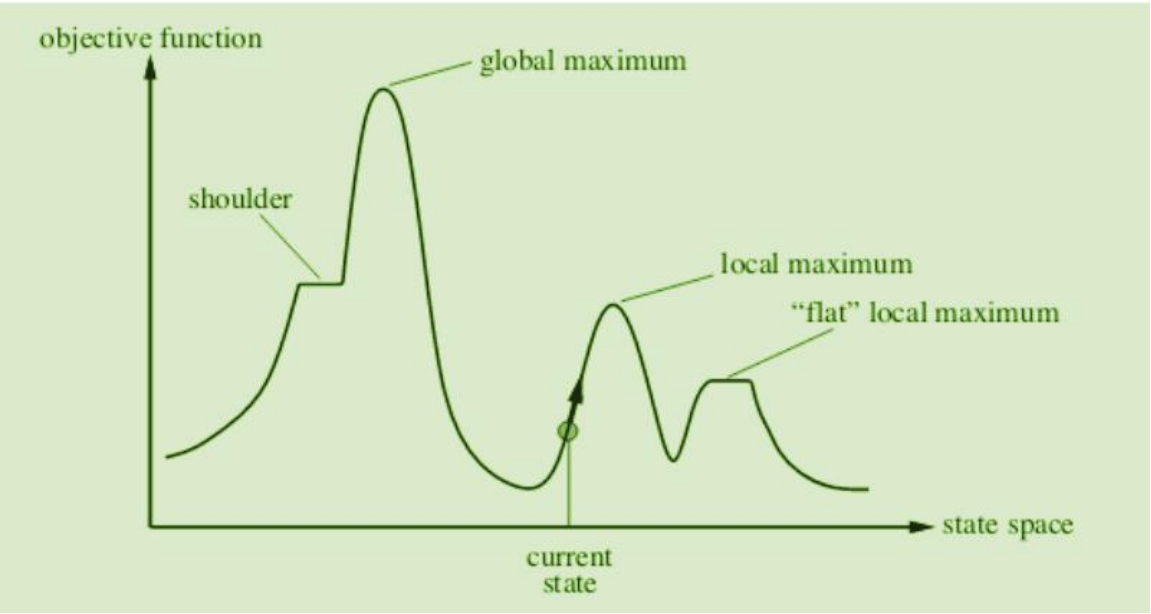
l'ampiezza dell'albero decisionale, limitando l'analisi a un numero predefinito di direzioni casuali che fungono da vettori di aggiornamento per lo stato corrente.
2. **Valuta velocemente le opzioni**: Per ognuna di queste mosse casuali, calcola rapidamente una "punteggio" usando una semplice funzione di valutazione (euristica). In Quoridor, questo punteggio potrebbe essere, ad esempio, la lunghezza del nuovo percorso più breve verso la vittoria dopo quella mossa.
3. **Scegli la migliore**: Tra tutte le mosse casuali provate, viene scelta ed eseguita esclusivamente la mossa che minimizza la funzione di costo (percorso più breve verso l'obiettivo) o che massimizza l'ostacolo per l'avversario. Se nessuna delle direzioni casuali prodotte genera un miglioramento rispetto allo stato attuale, l'algoritmo può rigenerare il campione o attestarsi sulla mossa meno penalizzante.

L'algoritmo Random Search è stato posizionato esclusivamente nel livello Easy del nostro simulatore di Quoridor per un motivo fondamentale: la sua essenza operativa. A differenza di avversari più avanzati, questo algoritmo non gioca una partita strategica; piuttosto, agisce secondo un impulso immediato. Di fatto la sua forza risiede solo nella velocità e nella semplicità di implementazione, qualità che lo rendono un perfetto avversario introduttivo: un benchmark basilare contro cui testare il motore di gioco e un punto di partenza accessibile per chi muove i primi passi in Quoridor, prima di confrontarsi con intelligenze artificiali che pensano davvero.

2.4 Algoritmo Greedy

Gli algoritmi greedy rappresentano una classe di algoritmi che, a ogni passaggio, effettuano la scelta localmente ottima con la speranza che questa conduca a una soluzione ottima globale. In ogni fase del processo, l'algoritmo seleziona l'opzione che appare migliore in quel preciso momento, senza preoccuparsi delle conseguenze future. Per facilitare questa selezione, spesso si ricorre all'ordinamento dei dati (sorting) o all'utilizzo di code con priorità (priority queues), strumenti che permettono di individuare rapidamente l'elemento successivo più vantaggioso. Una volta effettuata una scelta, si verifica il rispetto dei vincoli e si procede iterativamente fino al raggiungimento della soluzione.

L'essenza del greedy è la **scelta miope**. A differenza della programmazione dinamica, che guarda avanti e valuta tutte le possibili combinazioni future, il greedy non torna mai sui suoi passi. Una volta presa una decisione, questa è definitiva. Dove il greedy fallisce, o comunque non è consigliato?



Come si vede spesso nelle rappresentazioni grafiche, un algoritmo greedy potrebbe scalare la collina più vicina (ottimo locale) convinto di raggiungere la vetta più alta del mondo, ignorando che in realtà vi è un ottimo locale che il greedy non riesce a intravedere. Il problema di quest'algoritmo (e quindi del suo utilizzo) risiede proprio nella sua "miopia": q uando raggiunge la cima di una collina, cioè un massimo locale, si ferma perché ogni passo vicino sembra peggiorare la situazione, anche se da qualche parte più lontano esiste una collina più alta, il massimo globale. Allo stesso modo, se incontra una zona piatta o una "spalla", può rallentare o bloccarsi perché non percepisce un miglioramento immediato.

3 Progettazione ed implementazione

3.1 Architettura complessiva del sistema

La realizzazione del progetto prevede un'architettura a tre livelli: un **motore logico** implementato in Prolog (quoridor.pl), un **server HTTP** (server.pl) che funge da intermediario, e **un'interfaccia web client-side** sviluppata con HTML, CSS e JavaScript. Questa decomposizione permette una netta separazione tra la logica del gioco, la strategia dell'IA e la presentazione visuale, facilitando sia il debug che eventuali estensioni future.

Il flusso di comunicazione procede come segue: l'utente interagisce con la scacchiera tramite l'interfaccia web (client), che invia una richiesta HTTP POST al server, specificando la mossa desiderata e il livello di difficoltà; il server Prolog riceve la richiesta, valida la mossa, l'applica allo stato di gioco, esegue la logica dell'IA per computare la contromossa, e infine restituisce il nuovo stato al client in formato JSON. Questo schema permette al client di mantenere uno stato sincronizzato con il server e di visualizzare in tempo reale l'evoluzione della partita.

3.2 Motore logico: rappresentazione dello stato e validazione delle mosse

Il cuore del sistema risiede nel modulo Prolog `quoridor.pl`, che implementa la rappresentazione dello stato di gioco e tutti gli algoritmi di controllo e ricerca. Lo stato è rappresentato come una struttura compositiva `game(P1, P2, Walls, Turn)`, dove `P1` e `P2` sono termini `player(X, Y, WallsLeft)` che memorizzano la posizione cartesiana del pedone e il numero di barriere rimanenti, `Walls` è una lista di termini `wall(X, Y, Orientation)` che traccia tutte le barriere posizionate sulla scacchiera, e `Turn` è un atomo `(p1 o p2)` che indica il giocatore corrente.

La validazione delle mosse è affidata al predicato `is_valid_move/2`, che verifica due tipi di azioni: movimento del pedone e posizionamento di una barriera. Per il movimento del pedone, il predicato controlla che la posizione di destinazione sia all'interno dei confini della scacchiera, adiacente alla posizione corrente in senso ortogonale (non diagonale), e che non sia bloccata da una barriera tramite il predicato `wall_blocks/3`. Particolare attenzione è stata dedicata al caso speciale in cui i due pedoni si trovano adiacenti: in questa situazione, il giocatore può saltare l'avversario e avanzare di due caselle in linea retta, oppure spostarsi in una delle caselle laterali adiacenti, se libere.

Per il posizionamento di barriere, il sistema verifica che il giocatore disponga ancora di barriere, che la posizione sia valida (all'interno della griglia 8×8 dedicata ai muri, cioè sugli spazi tra le caselle), che non vi siano già barriere sovrapposte o intersecanti, e – il vincolo più critico – che l'aggiunta della barriera non blocchi completamente alcuno dei due giocatori, impedendogli di raggiungere la linea di arrivo. Quest'ultimo controllo è implementato mediante ricerca in ampiezza (BFS) nel predicato `has_path_to_goal/2`, che verifica l'esistenza di almeno un cammino per ciascun giocatore verso la propria linea di arrivo. Grazie a questo meccanismo, il gioco mantiene intrinsecamente l'equità: nessun giocatore può essere bloccato, qualunque sia la strategia dell'avversario.

3.3 Strategie di ricerca: una gerarchia di quattro livelli di difficoltà

La strategia di gioco dell'IA è implementata tramite il predicato `choose_move/3`, che accetta tre parametri: il livello di difficoltà, lo stato di gioco corrente, e restituisce la mossa scelta. La particolare rilevanza di questa sezione risiede nel fatto che ogni livello implementa una strategia di ricerca qualitativamente diversa, passando da un approccio completamente casuale a un algoritmo di ricerca sofisticato con restrizioni intelligenti. Nei paragrafi successivi vengono descritti in dettaglio i vari livelli di difficoltà.

3.3.1 Livello Easy: ricerca casuale

La difficoltà **easy** implementa la strategia più semplice: il predicato `choose_move(easy, State, Move)` genera l'insieme completo di tutte le mosse valide (sia movimenti di pedone che posizionamenti di barriera) tramite il predicato ausiliario `findall_valid_moves/2`, e seleziona una mossa a caso tramite `random_member/2`.

Questa strategia non impiega alcun algoritmo di ricerca o valutazione: ogni mossa ha uguale probabilità di essere scelta, indipendentemente dal suo impatto sul gioco. Il livello easy è principalmente utile per test iniziali e per permettere all'utente di **imparare le regole del gioco**, senza affrontare una competizione seria.

3.3.2 Livello Medium: ricerca greedy con BFS

La difficoltà **medium** impiega una **strategia golosa (greedy)** che priorizza il movimento del pedone e valuta le mosse in base alla distanza immediata verso la linea di arrivo. L'algoritmo procede in due fasi:

1. **Generazione dei candidati**: il predicato `findall_pawn_moves/2` genera tutti i movimenti di pedone validi nello stato corrente, filtrandoli dal controllo della validità. Si noti che questa fase **esclude esplicitamente i posizionamenti di barriera**, forzando l'IA a prioriz̀zare il movimento diretto verso la vittoria.
2. **Valutazione e selezione**: per ogni mossa di pedone candidata, il sistema applica il movimento tramite `make_move/3` e valuta la posizione risultante calcolando la **funzione di valutazione euristica** definita nel predicato `evaluate/2`. Questa funzione è stata descritta in dettaglio più avanti (cfr. Sez. 3.4), ma brevemente: `Score = DistanzaP1 - DistanzaP2`, dove le distanze sono i cammini minimi dalla posizione corrente alla linea di arrivo, calcolati mediante **ricerca in ampiezza (BFS)** nel predicato `bfs_dist/4.`
3. **Selezione del massimo**: tra tutte le mosse valutate, il predicato `best_greedy_move/3` ordina le mosse per score decrescente e seleziona il primo (migliore) candidato.

La logica greedy è efficace per le fasi intermedie di gioco: spinge l'IA a progredire verso la vittoria, e in molti casi fornisce una difesa ragionevole contro l'avversario.

3.3.3 Livello Hard: Minimax con Alpha-Beta Pruning a profondità 2

La difficoltà **hard** implementa il primo livello di **ricerca e pianificazione**, utilizzando l'algoritmo **minimax con alpha-beta pruning** a una profondità limitata di **2 livelli** (ovvero, l'IA guarda due turni in avanti: la sua prossima mossa e la possibile risposta dell'avversario). L'algoritmo è strutturato in tre componenti, elencati di seguito:

A. **Fase tattica di riconoscimento difensivo**: prima di eseguire minimax, il sistema valuta la situazione tramite il predicato `should_place_defensive_wall/3`. Se l'IA ha ancora barriere disponibili e il pedone dell'avversario dista meno di quattro caselle, l'IA entra in "modalità difensiva". In questa modalità:
    a. genera tutti i posizionamenti di barriera validi tramite `findall_wall_moves/2`.
    b. valuta rapidamente ciascun muro calcolando lo score della posizione risultante.
    c. ordina i muri per score decrescente e seleziona il migliore.

   Questa fase è cruciale perché riconosce situazioni critiche in cui bloccare il cammino dell'avversario è più importante che avanzare.

B. **Minimax standard per mosse di pedone**: se la fase tattica non attiva la difesa, l'IA ricorre al predicato `minimax/6` per analizzare le mosse di pedone. Il minimax esplora l'albero del gioco fino a profondità 2, alternando turni di massimizzazione (turno dell'IA) e minimizzazione (turno dell'avversario). L'alpha-beta pruning elimina rami non promettenti: se durante la ricerca la IA trova una mossa che garantisce uno score minore o uguale ad Alpha (soglia inferiore), interrompe l'esplorazione del resto dei fratelli (pruning).
C. **Calcolo del valore minimax**: ogni nodo foglia (profondità 0) è valutato mediante `evaluate/2`, che calcola `Score = DistanzaP1 - DistanzaP2` tramite BFS. Un valore positivo è favorevole all'IA `(P2)`, uno negativo favorevole all'avversario. L'algoritmo propaga questi valori verso l'alto dell'albero: ogni nodo di massimizzazione prende il valore massimo dei figli, ogni nodo di minimizzazione il valore minimo. La profondità 2 è un compromesso cruciale: permette all'IA di "vedere" le conseguenze della sua mossa sulla reazione dell'avversario, fornendo decisioni ragionevoli, mentre mantiene i tempi di calcolo entro 1-2 secondi per turno su hardware moderno. Il limite di profondità è controllato dal parametro passato a `minimax/6`: nel caso hard, è `Depth = 2`

3.3.4 Livello Extreme: Selective Minimax con restrizione del fattore di ramificazione

La difficoltà extreme implementa una strategia ibrida sofisticata che combina la valutazione rapida di molte mosse con una ricerca profonda su pochi candidati promettenti. Segue una spiegazione dell'algoritmo.

Innanzitutto, l'IA genera tutti i movimenti di pedone validi e li valuta tramite il predicato `evaluate/2`, ordinando i risultati per score decrescente. Successivamente, tramite il predicato ausiliario `take_n/3`, estrae i **top 5 migliori movimenti**, che rappresentano le opzioni più promettenti dal punto di vista della progressione verso il traguardo. Inoltre, la strategia riconosce che le barriere diventano meno utili nelle fasi finali di gioco. Pertanto, se l'IA dispone ancora di più di 3 barriere (indicativo di trovarsi ancora in fase intermedia), genera e valuta tutti i posizionamenti di barriera validi, ordinandoli per score e estraendo i top 3 migliori. Questi tre muri rappresentano le opzioni difensive di qualità più elevata. I 5 movimenti migliori e i 3 muri migliori vengono poi combinati in un insieme ridotto di al massimo 8 mosse candidate. Se invece l'IA ha esaurito le barriere (meno di 4 disponibili), la

strategia privilegia esclusivamente il movimento, interpretando correttamente che la priorità è correre verso la vittoria immediata.

Nella fase successiva, anziché applicare minimax a tutte le mosse possibili – il che porterebbe a un'esplosione combinatoria computazionalmente proibitiva – l'IA applica l'algoritmo minimax tramite il predicato `minimax_limited/6` **esclusivamente sulle 8 mosse candidate filtrate nelle fasi precedenti**. Questo permette di esplorare una profondità di 5 livelli (equivalente a 2.5 turni in avanti, circa 5 semiturni), scoprendo combinazioni tattiche e controcombinazioni che il livello hard, limitato a profondità 2, non può percepire.

Il beneficio strategico di questo approccio è considerevole: i top 5 movimenti sono già stati validati come promettenti in termini di avanzamento verso il traguardo, mentre i top 3 muri offrono opzioni difensive di qualità documentata. **La ricerca a profondità 5 non spreca risorse su mosse mediocri**, ma concentra il potere computazionale su candidati di alto valore. Di conseguenza, l'IA riesce a pianificare sequenze tattiche sofisticate: riconosce quando una mossa apparentemente priva di senso (ad esempio, spostarsi lateralmente) in realtà apre la strada a una combinazione vincente due turni più avanti, o quando una barriera in posizione apparentemente non ottimale blocca effettivamente l'unico cammino alternativo dell'avversario.

Dal punto di vista computazionale, il trade-off temporale è accuratamente calibrato. La fase di filtrazione costa circa *O(100 × 81) ≈ O(8100)*, una frazione di secondo su hardware moderno. Il minimax selettivo su 8 candidati a profondità 5 esplora nominalmente *O(8^5) = O(32768)* nodi di ricerca; tuttavia, con l'alpha-beta pruning e la lunghezza limitata della ricerca, il numero effettivo di nodi esplorati si riduce a *1000-5000*, **mantenendo tempi di risposta ragionevoli entro 5-10 secondi per mossa**. Questa latenza è accettabile nel contesto di un gioco da tavolo, dove i giocatori umani impiegano molto più tempo a decidere una mossa.

3.3.5 Verifica della vittoria

La determinazione del vincitore è implementata tramite il predicato `game_over/2` nel modulo Prolog `quoridor.pl`. Questo predicato verifica se uno dei due giocatori ha raggiunto la linea di arrivo e, in caso affermativo, unifica il secondo argomento con l'identificativo del vincitore.

La logica è semplice e coincide con le regole del gioco: il giocatore P1 (umano) vince quando il suo pedone raggiunge la riga 1 (la riga opposta rispetto alla sua posizione iniziale in riga 9). Il giocatore P2 (IA) vince quando il suo pedone raggiunge la riga 9 (corrispondentemente, la riga opposta rispetto alla posizione iniziale in riga 1). Nel codice:

```
game_over(game(player(_, 1, _), _, _, _), p1).
game_over(game(_, player(_, Y, _), _, _), p2) :-
    board_size(Y).
```

La verifica avviene **dopo ogni mossa** nel flusso server-side. Nello specifico, dopo che il giocatore umano ha eseguito la sua mossa e il server ha calcolato la contromossa dell'IA, il

server invoca `game_over/2` per determinare se la partita è conclusa. Se la partita è conclusa, il server restituisce al client un campo winner nel payload JSON di risposta (contenente "player" o "ai"); il client JavaScript legge questo campo e, se non è `null`, visualizza un messaggio che dichiara il vincitore e offre la possibilità di iniziare una nuova partita.
Questo meccanismo garantisce che la vittoria sia sempre **verificata lato server**, evitando che il client possa falsificare il risultato.

3.4 Funzione di valutazione euristica

La **funzione di valutazione** è definita nel predicato `evaluate/2` come:

```
evaluate(game(P1, P2, Walls, _), Score) :-
    shortest_path_len(P1, Walls, p1, D1),
    shortest_path_len(P2, Walls, p2, D2),
    Score is D1 - D2.
```

La semantica è semplice ma efficace:

- D1 è la lunghezza del cammino minimo da P1 (il giocatore umano) alla linea di arrivo (riga 1).
- D2 è la lunghezza del cammino minimo da P2 (l'IA) alla linea di arrivo (riga 9).
- `Score = D1 - D2` rappresenta la "differenza di distanza".

Un valore di Score positivo favorisce l'IA: significa che P1 è più lontana dal traguardo che P2. Un valore negativo favorisce l'avversario. Uno score di 0 indica parità. Un valore estremo come +8 significa che l'IA è a 1 mossa dalla vittoria mentre l'avversario è a 9 mosse, una situazione vincente.
I cammini minimi sono calcolati tramite **BFS con distanza**, implementata nel predicato `bfs_dist/4`, che è una variante del classico BFS che traccia anche la distanza dal punto di partenza. La complessità di BFS su una griglia 9×9 è O(81), rendendo ogni valutazione computazionalmente poco costosa (ideale per valutare centinaia di nodi durante la ricerca).

Comunque, è bene sottolineare che questa euristica è semplice e perde informazioni strategiche importanti, come la configurazione delle barriere future o la "qualità" di un cammino. Tuttavia, la semplicità è un vantaggio: permette di valutare posizioni molto rapidamente. In futuro, potrebbe essere arricchita con fattori aggiuntivi, come il numero di barriere rimanenti o la misura di "collo di bottiglia" creato dalle barriere.

3.5 Server HTTP e conversione JSON

Il modulo `server.pl` implementa un server http, utilizzando la libreria `Prolog library(http/thread_httpd)`, ed espone i tre seguenti endpoint principali:

- **GET /game/start**: inizializza una nuova partita, recupera lo stato iniziale dal predicato `initial_state/1` e lo serializza in JSON.

- **POST /game/move**: riceve una mossa dal client (formato JSON), la valida, l'applica allo stato di gioco, esegue la logica dell'IA per computare la contromossa, e restituisce il nuovo stato e l'eventuale vincitore.
- **GET / (wildcard)**: serve i file statici (HTML, CSS, JavaScript) dalla cartella `/web`.

Un aspetto critico è la **conversione bidirezionale tra il formato logico Prolog e JSON**. Ad esempio, lo stato Prolog `game(player(5, 9, 10), player(5, 1, 10), [], p1)` viene trasformato nel codice JSON presente di seguito:

```
{
  "p1": {"x": 5, "y": 9, "walls": 10},
  "p2": {"x": 5, "y": 1, "walls": 10},
  "walls": [],
  "turn": "p1"
}
```

Questa serializzazione è gestita dai predicati `state_to_json/2` e `json_to_state/2`, che operano mappature reversibili. Analogamente, le mosse JSON come `{"type": "move", "x": 5, "y": 8}` vengono convertite nel termine Prolog `move(5, 8)` tramite il predicato `json_to_move/2`. Un meccanismo di coercizione (`ensure_number/2`) gestisce le eventuali incoerenze di tipo tra la rappresentazione JSON e gli atomi Prolog, garantendo robustezza in caso di invio di dati numerici come stringhe.

3.6 Interfaccia grafica

L'interfaccia è costituita da tre componenti che lavorano insieme per fornire un'esperienza utente coerente e reattiva.

I file `index.html` e `style.css` forniscono la struttura e lo stile della pagina. L'**HTML** definisce una sezione modale iniziale per la scelta della difficoltà, un contenitore centrale per la scacchiera 9×9, e una barra laterale con informazioni di stato (barriere rimanenti, turno corrente). Il **CSS** implementa il rendering tramite CSS Grid, creando una griglia di celle con spaziatura tra loro per accogliere visualmente le barriere.

Il file `game.js` è il cuore della logica client-side e gestisce effettivamente il ciclo di gioco e la comunicazione con il server. Le funzioni principali sono:

- `startGame(level)`: invia una richiesta GET a `/game/start`, riceve lo stato iniziale dal server e triggera il primo rendering della scacchiera.
- `renderBoard()`: ripulisce il DOM e ricostruisce completamente la visualizzazione della scacchiera in base allo stato corrente, posizionando celle, pedoni, barriere e i clickable hitbox per il posizionamento di muri. È invocata ogni volta che lo stato cambia.
- `handleMove(x, y)` e `handleWall(x, y)`: gestiscono gli input dell'utente, creano il payload della mossa (pedone o barriera) e lo inviano al server via POST tramite `sendAction()`.
- `sendAction(move)`: effettua la richiesta POST a `/game/move` con la mossa e lo stato corrente, attende la risposta del server (nuovo stato + eventuali vincitori), aggiorna la variabile globale `gameState` e comporta un nuovo rendering della scacchiera.

4 Esempi di esecuzione

4.1 Introduzione

In questo capitolo si presentano alcuni esempi di esecuzione del gioco sviluppato, con l'obiettivo di illustrare in modo concreto il comportamento del sistema nelle varie fasi della partita. Si mostra come l'utente avvia una nuova partita tramite l'interfaccia web, come vengono gestite le mosse del giocatore e dell'IA, in che modo il sistema rileva la fine della partita e come cambia il comportamento dell'agente al variare della difficoltà selezionata. Questa parte ha lo scopo di collegare la progettazione logica e algoritmica descritta nei capitoli precedenti con l'esperienza effettiva di gioco, evidenziando sia gli aspetti di usabilità sia quelli legati alle prestazioni e alla profondità di ricerca dell'IA.

4.2 Avvio ed esecuzione della partita

All'apertura della pagina web, l'utente viene accolto da una schermata iniziale, mostrata in Figura 2, che consente di scegliere il livello di difficoltà dell'avversario artificiale. Una volta selezionata la difficoltà, il client invia una richiesta al server per inizializzare una nuova partita. Il server costruisce lo stato iniziale del gioco, posizionando i due pedoni nelle rispettive caselle di partenza come in Figura 3 - Stato iniziale della partita (riga 9 per il giocatore, riga 1 per l'IA) e assegnando il numero iniziale di barriere.

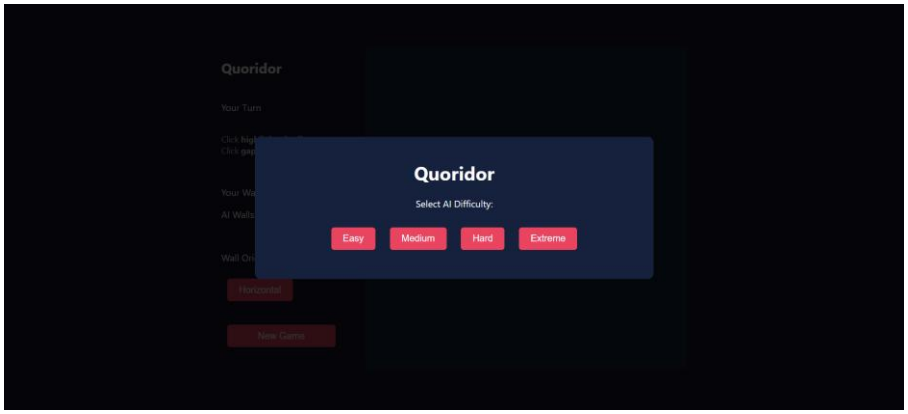


Figura 2 - Schermata di avvio

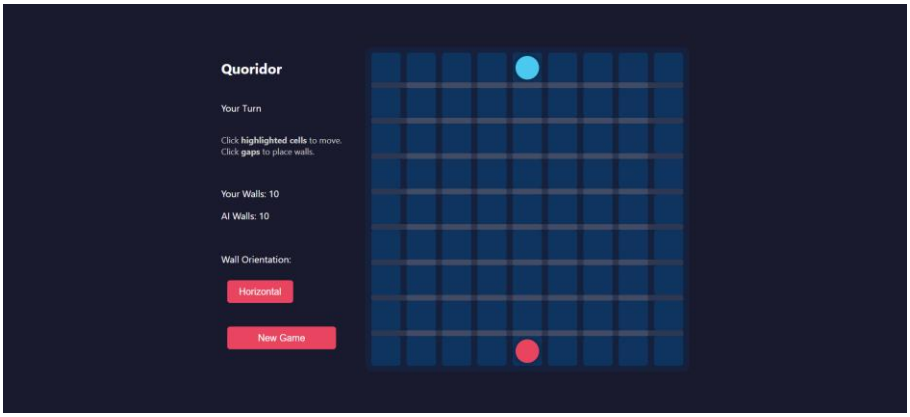


Figura 3 - Stato iniziale della partita

Lo stato iniziale viene serializzato in formato JSON e restituito al client, che provvede a renderizzare la scacchiera 9×9, i due pedoni e il contatore delle barriere rimanenti. Da questo

momento in poi, la partita procede a turni alterni: l'utente effettua una mossa cliccando su una casella per muovere il proprio pedone oppure su uno spazio tra due caselle per posizionare una barriera. Ogni azione genera una richiesta HTTP POST verso il server, che si occupa di validare la mossa tramite il predicato `is_valid_move/2`, aggiornare lo stato e calcolare la risposta dell'IA prima di restituire il nuovo stato al client.

Se la mossa dell'utente non è valida (ad esempio, un muro sovrapposto o un movimento che blocca completamente l'accesso alla meta), il server restituisce un errore e l'interfaccia notifica l'utente senza alterare lo stato visualizzato. Se la mossa è valida, l'interfaccia aggiorna la scacchiera mostrando la mossa del giocatore seguita immediatamente dalla risposta calcolata dall'IA.

4.2.1 Analisi di una partita a livello Extreme

Per illustrare le capacità strategiche del sistema, analizziamo lo svolgimento di una partita contro l'agente di livello **Extreme**. In questa modalità, l'IA non utilizza semplici regole reattive, ma impiega un algoritmo di **Minimax Selettivo** a profondità 5. Questo approccio filtra le mosse, considerando solo i migliori 5 movimenti di pedone e i migliori 3 posizionamenti di muro, permettendo una pianificazione profonda senza esplodere nella complessità computazionale.

Dopo aver avviato il gioco, la situazione iniziale vede i pedoni al centro delle rispettive basi (Figura 2). A differenza dei livelli inferiori, l'IA (pedone Blu) adotta fin da subito un approccio proattivo. Già nelle prime fasi, come visibile in Figura 4 - Turno cinque, l'IA inizia a posizionare barriere per modellare il grafo della scacchiera a suo favore. Il contatore "AI Walls" scende rapidamente (nell'immagine a 7), mentre il giocatore umano (pedone Rosso) ne ha piazzate meno (rimanenti 8).

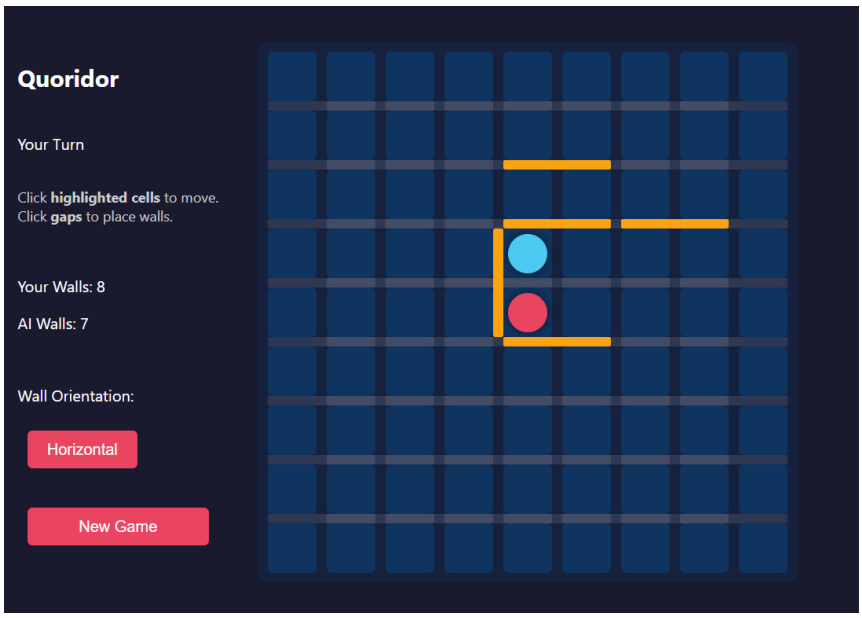


Figura 4 - Turno cinque

Questa aggressività è dettata dalla funzione di valutazione `evaluate/2` che mira a massimizzare la differenza dei cammini minimi `Score = D_player - D_AI`: posizionare

muri precoci spesso aumenta drasticamente `D_player`. Proseguendo nella partita Figura 5 - Fase centrale, la strategia dell'IA crea una struttura complessa a "imbuto". Mentre il giocatore umano è costretto a un percorso tortuoso sulla sinistra per aggirare le barriere, l'IA si è garantita un corridoio libero verso il basso.

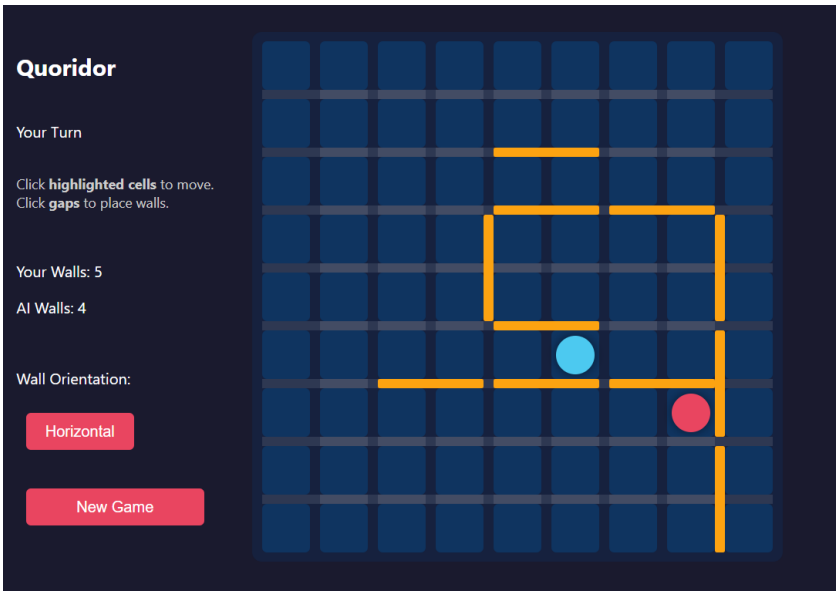


Figura 5 - Fase centrale

L'algoritmo a profondità 5 ha calcolato che il sacrificio delle barriere in fase iniziale avrebbe garantito un vantaggio di tempo (numero di mosse per arrivare a meta) superiore nel lungo periodo. Un aspetto cruciale dell'implementazione Extreme emerge nel finale di partita, mostrato in Figura 6 - Fase finale. Quando il numero di barriere dell'IA scende a 3 o meno, il codice attiva una condizione di "cut-off": `(W1 > 3 -> ... ; CombinedMoves = PawnMoves).`

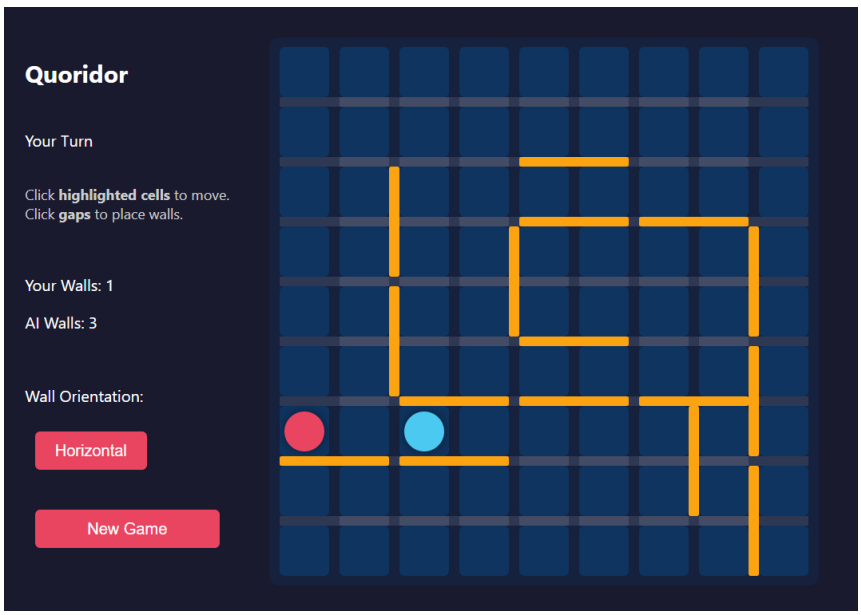


Figura 6 - Fase finale

Come si nota nell'immagine, avendo l'IA esattamente tre muri residui, l'algoritmo smette di considerare il posizionamento di nuove barriere per risparmiare risorse di calcolo e

focalizzarsi sulla corsa verso il traguardo. Avendo intrappolato il giocatore in un percorso lungo (il pedone rosso è ancora alla riga 3), l'IA sfrutta il vantaggio posizionale per correre verso la vittoria senza ulteriori distrazioni difensive.
La partita si conclude con il successo dell'agente intelligente, come visibile in Figura 7 - Termine della partita. Il server rileva la condizione di vittoria tramite `game_over/2` e il client visualizza il messaggio "AI Wins!", confermando l'efficacia della pianificazione a lungo termine del livello Extreme rispetto alla strategia umana.

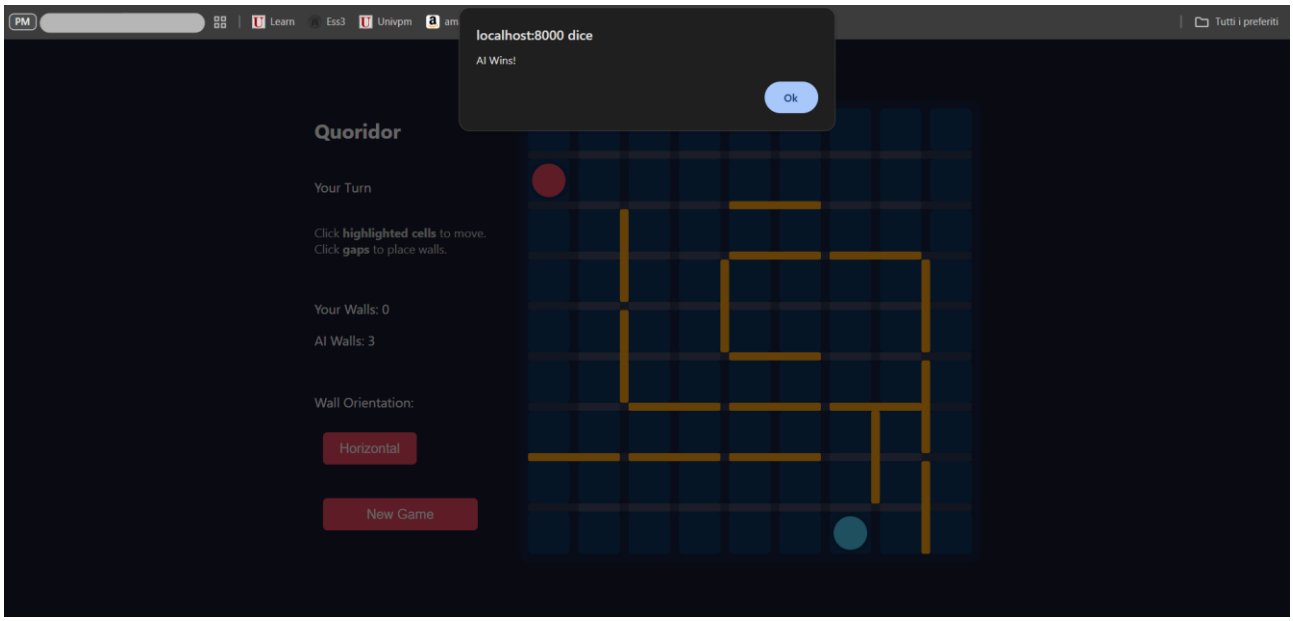


Figura 7 - Termine della partita

4.3 Analisi delle Prestazioni Temporali

Un aspetto critico nella progettazione di un agente per giochi in tempo reale risiede nel bilanciamento tra la profondità di ricerca e la latenza di risposta. La complessità computazionale di Quoridor, unita alla necessità di mantenere l'interazione fluida per l'utente, ha richiesto un'attenta calibrazione degli algoritmi implementati. Durante le sessioni di test, sono state monitorate le prestazioni per ciascun livello di difficoltà, evidenziando come le diverse scelte architetturali impattino sui tempi di esecuzione.
Per quanto riguarda i livelli *Easy* e *Medium*, le strategie adottate garantiscono tempi di risposta pressoché istantanei, inferiori ai 100 millisecondi. Nel caso del livello *Easy*, ciò è dovuto alla natura banale della selezione casuale, mentre per il livello *Medium* l'efficienza deriva dalla bassa complessità dell'algoritmo di ricerca in ampiezza (BFS). Poiché la scacchiera è modellata come un grafo di dimensioni contenute (9 X 9, ovvero 81 nodi), il calcolo del cammino minimo ha un costo computazionale trascurabile ($O(V + E)$), permettendo all'agente Greedy di valutare le mosse immediate senza percepibili ritardi.
Il livello *Hard*, che introduce l'algoritmo Minimax con potatura Alpha-Beta a una profondità di 2, ha mantenuto tempi di risposta fluidi, attestandosi mediamente tra 1 e 2 secondi su hardware standard. L'analisi del codice rivela che questa efficienza è ottenuta limitando drasticamente il fattore di ramificazione dell'albero di ricerca. A meno che non si attivino specifiche condizioni difensive, l'algoritmo esplora esclusivamente le mosse di spostamento del pedone, ignorando la vasta combinatoria legata al posizionamento dei muri. Questa

scelta progettuale permette di beneficiare di una minima capacità di previsione senza incorrere in colli di bottiglia computazionali.
Il risultato più significativo, tuttavia, riguarda il livello *Extreme*. Nonostante la profondità di ricerca sia stata estesa a 5 livelli, permettendo all'IA di "vedere" fino a 2.5 turni nel futuro, i tempi di risposta sono rimasti contenuti in un intervallo accettabile di 5-10 secondi. Questo risultato conferma l'efficacia dell'approccio *Selective Minimax* implementato nel predicato `choose_move/3`. Invece di esplorare l'intero spazio degli stati, che crescerebbe esponenzialmente, l'algoritmo filtra i candidati all'origine, selezionando tramite il predicato `take_n` solo i 5 migliori movimenti e i 3 migliori posizionamenti di muro. Questa potatura euristica preventiva riduce drasticamente il numero di nodi visitati, rendendo praticabile una ricerca profonda che altrimenti sarebbe impossibile in tempi ragionevoli.

4.4 Confronto qualitativo delle strategie

L'osservazione empirica delle partite svolte ha permesso di delineare profili comportamentali distinti per ciascun livello di difficoltà, validando le scelte algoritmiche descritte nel capitolo relativo all'implementazione. Il confronto tra le diverse modalità di gioco evidenzia come la variazione dell'orizzonte di ricerca e delle euristiche trasformi radicalmente lo "stile" dell'agente.
Una prima distinzione fondamentale emerge confrontando la "miopia" del livello *Medium* con la capacità di pianificazione dei livelli superiori. L'agente *Medium*, guidato da una logica puramente Greedy, tende a ignorare i pericoli futuri pur di avanzare immediatamente verso la meta. Al contrario, i livelli *Hard* ed *Extreme* dimostrano la capacità di evitare vicoli ciechi e trappole. In particolare, il livello *Extreme* è l'unico in grado di effettuare mosse controintuitive, come muoversi lateralmente o addirittura indietro, qualora la ricerca profonda riveli che il percorso apparentemente più breve verrebbe bloccato dall'avversario nei turni successivi.
La differenza più marcata, tuttavia, risiede nella gestione delle barriere. L'agente di livello *Hard* adotta un approccio prettamente reattivo: come definito nel predicato `should_place_defensive_wall`, esso considera l'uso dei muri solo quando l'avversario si trova a una distanza inferiore a quattro caselle dal traguardo. Questo comportamento simula un giocatore che "corre" per gran parte della partita e passa alla difesa solo in situazioni di emergenza. Diametralmente opposto è l'approccio del livello *Extreme*, che utilizza le barriere in modo proattivo. Grazie all'inclusione dei muri nell'albero di ricerca principale fin dalle fasi di apertura, l'agente è in grado di modellare la topologia della scacchiera a proprio vantaggio, creando percorsi obbligati per l'avversario molto prima che la minaccia diventi imminente.
Inoltre, la logica condizionale implementata prevede un cambio di strategia nel finale di partita: quando le barriere residue scendono sotto la soglia critica di 4, l'IA smette di valutare posizionamenti difensivi per concentrare tutte le risorse computazionali sulla corsa finale verso la vittoria.
Infine, l'analisi ha evidenziato anche i limiti dell'attuale funzione di valutazione. Basandosi esclusivamente sulla differenza dei cammini minimi (`Score = D_player - D_AI`), l'IA tende talvolta a sottostimare il valore strategico del possesso delle barriere in sé, spendendole talvolta troppo rapidamente nelle fasi iniziali. Questa osservazione suggerisce che, in sviluppi futuri, l'euristica potrebbe essere raffinata integrando un peso specifico per il numero di muri residui, rendendo l'agente ancora più conservativo e difficile da battere.

Appendice A

Nelle prossime pagine, viene riportato il codice sorgente completo utilizzato per il progetto.

quoridor.pl

```
% ========================================
% MODULO QUORIDOR - Logica del Gioco
% ========================================
% Questo modulo implementa tutte le regole del gioco Quoridor,
% inclusi i movimenti dei pedoni, il posizionamento dei muri,
% e l'intelligenza artificiale con diversi livelli di difficoltà.

:- module(quoridor, [
    initial_state/1,        % Stato iniziale del gioco
    is_valid_move/2,        % Verifica se una mossa è valida
    make_move/3,            % Applica una mossa e genera il nuovo stato
    game_over/2,            % Controlla se il gioco è terminato
    choose_move/3,          % AI: sceglie la mossa in base alla difficoltà
    get_valid_pawn_moves/2  % Restituisce le mosse valide del pedone
]).

:- discontiguous choose_move/3.

% ========================================
% CONFIGURAZIONE DELLA GRIGLIA
% ========================================
% Qui puoi modificare le dimensioni della griglia:
% - Per griglia 9x9 (standard): board_size(9), initial_walls(10)
% - Per griglia 5x5 (veloce):   board_size(5), initial_walls(5)
board_size(9).       % Dimensione della griglia (9x9)
initial_walls(10).   % Numero di muri disponibili per ogni giocatore

% ========================================
% RAPPRESENTAZIONE DELLO STATO DI GIOCO
% ========================================
% Lo stato del gioco è rappresentato come: game(P1, P2, Walls, Turn)
%
% Dove:
%   - P1, P2 = player(X, Y, WallsLeft)
%       X, Y: posizione del pedone sulla griglia
%       WallsLeft: numero di muri rimanenti
%   - Walls = [wall(X, Y, Orientation), ...]
%       Lista di tutti i muri posizionati
%       Orientation può essere 'h' (orizzontale) o 'v' (verticale)
%   - Turn = p1 o p2 (indica di chi è il turno)
%
% STATO INIZIALE:
% - Giocatore 1 (P1) parte dalla riga 9 (in basso)
% - Giocatore 2 (P2) parte dalla riga 1 (in alto)
% - Entrambi partono dalla colonna 5 (centro)
% - Entrambi hanno 10 muri disponibili
% - Il primo turno è del giocatore 1
initial_state(game(player(5, 9, 10), player(5, 1, 10), [], p1)).

```

```
% ========================================
% VALIDAZIONE DELLE COORDINATE
% ========================================
% Verifica che una posizione (X, Y) sia valida sulla griglia.
% Una posizione è valida se entrambe le coordinate sono comprese
% tra 1 e la dimensione della griglia.
valid_pos(X, Y) :-
    board_size(Size),
    between(1, Size, X),  % X deve essere tra 1 e Size
    between(1, Size, Y).  % Y deve essere tra 1 e Size

% ========================================
% VALIDAZIONE DELLE MOSSE
% ========================================

% Punto di ingresso principale: verifica se una mossa è valida
% dato lo stato corrente del gioco.
is_valid_move(game(P1, P2, Walls, Turn), Move) :-
    % Determina chi è il giocatore corrente e chi è l'avversario
    (Turn == p1 -> CurrentPlayer = P1, OtherPlayer = P2 ; CurrentPlayer = P2, OtherPlayer = P1),
    % Delega la validazione alla funzione is_valid_action
    is_valid_action(CurrentPlayer, OtherPlayer, Walls, Move).

% --- SALTO SOPRA L'AVVERSARIO ---
% Questa regola permette di saltare l'avversario quando è adiacente.
% Il salto porta il pedone nella casella immediatamente dietro l'avversario.
is_valid_action(player(X, Y, _), player(OX, OY, _), Walls, move(JumpX, JumpY)) :-
    % 1. L'avversario deve essere adiacente al giocatore corrente
    adjacent(X, Y, OX, OY),
    % 2. Non ci devono essere muri tra giocatore e avversario
    \+ wall_blocks(X, Y, OX, OY, Walls),
    % 3. Calcola la destinazione del salto (2 caselle nella stessa direzione)
    %    Esempio: se sei in (3,3) e l'avversario in (3,4), salti in (3,5)
    JumpX is OX + (OX - X),
    JumpY is OY + (OY - Y),
    % 4. La destinazione del salto deve essere una posizione valida
    valid_pos(JumpX, JumpY),
    % 5. Non ci devono essere muri tra avversario e destinazione
    \+ wall_blocks(OX, OY, JumpX, JumpY, Walls).
% --- MOVIMENTO NORMALE DEL PEDONE ---
% Permette di muovere il pedone in una casella adiacente vuota.
is_valid_action(player(X, Y, _), player(OX, OY, _), Walls, move(NX, NY)) :-
    valid_pos(NX, NY),                      % La destinazione deve essere valida
    adjacent(X, Y, NX, NY),
    \+ wall_blocks(X, Y, NX, NY, Walls),
    (NX \= OX ; NY \= OY).                  %

% --- POSIZIONAMENTO DI UN MURO ---
% Permette di piazzare un muro se:
% 1. Il giocatore ha ancora muri disponibili
% 2. La posizione è valida
% 3. Non c'è già un muro in quella posizione
% 4. Il muro non si sovrappone ad altri muri
% 5. Il muro non blocca completamente il percorso verso l'obiettivo
is_valid_action(player(PX, PY, WallsLeft), player(OX, OY, _), Walls, place_wall(X, Y, O)) :-
    WallsLeft > 0,                          % Devi avere muri disponibili
    valid_wall_pos(X, Y),                   % La posizione del muro valida
    \+ member(wall(X, Y, _), Walls),        % Non ci deve essere già un muro qui
```

```
108     \+ wall_overlaps(X, Y, O, Walls),       % I muri non devono sovrapporsi
109     % Verifica che entrambi i giocatori possano raggiungere il loro obiettivo
110     \+ blocks_path(player(PX, PY, WallsLeft), player(OX, OY, _), [wall(X, Y, O)|Walls]).
111
112 % ======================================
113 % PATHFINDING - Ricerca del Percorso (BFS)
114 % ======================================
115 % Queste funzioni verificano se un giocatore può ancora raggiungere
116 % il suo obiettivo. Usa l'algoritmo BFS (Breadth-First Search).
117
118 % Verifica se il posizionamento di un muro blocca il percorso di un giocatore
119 blocks_path(P1, _, Walls) :-
120     \+ has_path_to_goal(P1, Walls, p1),  % Se P1 non ha più un percorso, blocca
121     !.
122 blocks_path(_, P2, Walls) :-
123     \+ has_path_to_goal(P2, Walls, p2).  % Se P2 non ha più un percorso, blocca
124
125 % Verifica se un giocatore ha un percorso valido verso il suo obiettivo
126 has_path_to_goal(player(X, Y, _), Walls, Player) :-
127     goal_row(Player, GoalY),             % Ottiene la riga obiettivo del giocatore
128     bfs([[X, Y]], GoalY, Walls, []).     % Esegue BFS dalla posizione corrente
129
130 % Definisce la riga obiettivo per ciascun giocatore:
131 % - P1 parte dalla riga 9 (in basso) e deve raggiungere la riga 1 (in alto)
132 % - P2 parte dalla riga 1 (in alto) e deve raggiungere la riga Size (in basso)
133 goal_row(p1, 1).
134 goal_row(p2, Size) :- board_size(Size).
135
136 % BFS: Algoritmo di ricerca in ampiezza
137 % Caso base: se la posizione corrente è sulla riga obiettivo, successo!
138 bfs([[_, Y]|_], GoalY, _, _) :- Y == GoalY, !.
139 % Caso ricorsivo: esplora le posizioni adiacenti
140 bfs([[X, Y]|Rest], GoalY, Walls, Visited) :-
141     % Trova tutte le mosse valide dalla posizione corrente
142     findall([NX, NY], (
143         adjacent(X, Y, NX, NY),              % Posizione adiacente
144         valid_pos(NX, NY),                   % Posizione valida sulla griglia
145         \+ wall_blocks(X, Y, NX, NY, Walls), % Nessun muro blocca il movimento
146         \+ member([NX, NY], Visited),        % Non già visitata
147         \+ member([NX, NY], [[X, Y]|Rest])   % Non già in coda
148     ), NextMoves),
149     append(Rest, NextMoves, NewQueue),       % Aggiungi le nuove mosse alla coda
150     bfs(NewQueue, GoalY, Walls, [[X, Y]|Visited]).  % Continua la ricerca
151
152 % ======================================
153 % FUNZIONI HELPER
154 % ======================================
155
156 % Definisce le 4 posizioni adiacenti a (X, Y): su, giù, sinistra, destra
157 adjacent(X, Y, X, NY) :- NY is Y + 1.   % Movimento in alto
158 adjacent(X, Y, X, NY) :- NY is Y - 1.   % Movimento in basso
159 adjacent(X, Y, NX, Y) :- NX is X + 1.   % Movimento a destra
160 adjacent(X, Y, NX, Y) :- NX is X - 1.   % Movimento a sinistra
161 % Verifica che una posizione per un muro sia valida
162 % I muri possono essere posizionati solo tra le caselle, quindi
163 % le coordinate valide vanno da 1 a (Size-1)
164 valid_wall_pos(X, Y) :-
165     board_size(Size),
```

```
    MaxWall is Size - 1,
    between(1, MaxWall, X),
    between(1, MaxWall, Y).

% --- LOGICA DI BLOCCO DEI MURI ---

% Verifica se un muro blocca il movimento verticale (su/giù)
wall_blocks(X, Y, X, NY, Walls) :-
    MinY is min(Y, NY),
    % Un muro orizzontale in (X, MinY) o (X-1, MinY) blocca il movimento
    (member(wall(X, MinY, h), Walls) ; member(wall(X1, MinY, h), Walls), X1 is X - 1).

% Verifica se un muro blocca il movimento orizzontale (sinistra/destra)
wall_blocks(X, Y, NX, Y, Walls) :-
    MinX is min(X, NX),
    % Un muro verticale in (MinX, Y) o (MinX, Y-1) blocca il movimento
    (member(wall(MinX, Y, v), Walls) ; member(wall(MinX, Y1, v), Walls), Y1 is Y - 1).

% --- LOGICA DI SOVRAPPOSIZIONE DEI MURI ---

% Un muro orizzontale si sovrappone se c'è un altro muro orizzontale adiacente
wall_overlaps(X, Y, h, Walls) :-
    member(wall(X1, Y, h), Walls), (X1 is X - 1 ; X1 is X + 1).
% Un muro verticale si sovrappone se c'è un altro muro verticale adiacente
wall_overlaps(X, Y, v, Walls) :-
    member(wall(X, Y1, v), Walls), (Y1 is Y - 1 ; Y1 is Y + 1).
% I muri si sovrappongono anche se si incrociano nella stessa posizione
wall_overlaps(X, Y, h, Walls) :- member(wall(X, Y, v), Walls).
wall_overlaps(X, Y, v, Walls) :- member(wall(X, Y, h), Walls).

% ========================================
% APPLICAZIONE DELLE MOSSE
% ========================================
% Queste funzioni applicano una mossa allo stato corrente
% e generano il nuovo stato di gioco.

% Applica una mossa per il giocatore 1 (P1)
make_move(game(P1, P2, Walls, p1), Move, game(NewP1, P2, NewWalls, p2)) :-
    apply_move(P1, Walls, Move, NewP1, NewWalls).

% Applica una mossa per il giocatore 2 (P2)
make_move(game(P1, P2, Walls, p2), Move, game(P1, NewP2, NewWalls, p1)) :-
    apply_move(P2, Walls, Move, NewP2, NewWalls).

% Applica un movimento del pedone: aggiorna solo la posizione
apply_move(player(_, _, W), Walls, move(NX, NY), player(NX, NY, W), Walls).
% Applica il posizionamento di un muro: aggiunge il muro e decrementa il contatore
apply_move(player(X, Y, W), Walls, place_wall(WX, WY, O), player(X, Y, NW), [wall(WX, WY,
O)|Walls]) :-
    NW is W - 1.

% ========================================
% CONDIZIONI DI VITTORIA
% ========================================
% Il gioco termina quando un giocatore raggiunge la riga obiettivo.

% P1 vince se raggiunge la riga 1 (in alto)
game_over(game(player(_, 1, _), _, _, _), p1).
```

```
% P2 vince se raggiunge la riga Size (in basso)
game_over(game(_, player(_, Y, _), _, _), p2) :-
    board_size(Y).

% ========================================
% INTELLIGENZA ARTIFICIALE (AI)
% ========================================
% Implementa 4 livelli di difficoltà: easy, medium, hard, extreme

% --- DIFFICOLTÀ FACILE (EASY) ---
% Sceglie una mossa casuale tra tutte le mosse valide.
% choose_move(+Difficulty, +State, -Move)
choose_move(easy, State, Move) :-
    findall_valid_moves(State, Moves),  % Trova tutte le mosse valide
    Moves \= [],                        % Verifica che ci siano mosse disponibili
    random_member(Move, Moves).         % Sceglie una mossa a caso

% --- DIFFICOLTÀ MEDIA (MEDIUM) ---
% Usa una strategia greedy: sceglie la mossa che minimizza la distanza
% dall'obiettivo. Preferisce muovere il pedone rispetto ai muri.
choose_move(medium, State, Move) :-
    findall_pawn_moves(State, Moves),    % Trova tutte le mosse del pedone
    (Moves \= [] ->
        % Se ci sono mosse del pedone, scegli la migliore
        best_greedy_move(State, Moves, Move)
    ;
        % Nessuna mossa del pedone, piazza un muro casuale
        findall_wall_moves(State, WallMoves),
        WallMoves \= [],
        random_member(Move, WallMoves)
    ).
% --- DIFFICOLTÀ DIFFICILE (HARD) ---
% Strategia ibrida: usa muri difensivi quando l'avversario è vicino,
% altrimenti usa minimax con profondità 2 per scegliere la mossa ottimale.
choose_move(hard, State, Move) :-
    State = game(P1, P2, _, Turn),
    (should_place_defensive_wall(P1, P2, Turn) ->
        % Cerca di trovare un buon muro difensivo
        findall_wall_moves(State, WallMoves),
        findall(Score-WallMove, (
            member(WallMove, WallMoves),
            make_move(State, WallMove, NextState),
            evaluate(NextState, Score)
        ), ScoredWalls),
        (ScoredWalls \= [] ->
            % Ordina i muri per punteggio e sceglie il migliore
            sort(ScoredWalls, Sorted),
            reverse(Sorted, [_-Move|_])
        ;
            % Nessun muro valido, usa minimax
            minimax(State, 2, -1000, 1000, Move, _)
        )
    ;
        % Usa minimax per il movimento del pedone
        minimax(State, 2, -1000, 1000, Move, _)
    ).

% Decide se l'AI dovrebbe piazzare un muro difensivo
```

```
% Condizioni: avere muri disponibili E avversario vicino (< 4 righe)
should_place_defensive_wall(player(_, Y1, W1), player(_, Y2, _), p2) :-
    W1 > 0,                             % Deve avere muri disponibili
    Distance is abs(Y1 - Y2),
    Distance < 4.                       % L'avversario deve essere vicino

% --- DIFFICOLTÀ ESTREMA (EXTREME) ---
% Strategia avanzata: valuta le migliori 5 mosse del pedone e i migliori 3 muri,
% poi usa minimax con profondità 5 su questo set ridotto per massimizzare
% la qualità della decisione riducendo il tempo di calcolo.
choose_move(extreme, State, Move) :-
    findall_pawn_moves(State, AllPawnMoves),
    (AllPawnMoves \= [] ->
        % Valuta tutte le mosse del pedone e le ordina per punteggio
        findall(Score-M, (
            member(M, AllPawnMoves),
            make_move(State, M, NextState),
            evaluate(NextState, Score)
        ), ScoredPawns),
        sort(ScoredPawns, SortedPawns),
        reverse(SortedPawns, TopPawns),
        take_n(5, TopPawns, BestPawns),          % Prendi le migliori 5 mosse del pedone
        findall(PM, member(_-PM, BestPawns), PawnMoves),

        % Genera le migliori mosse con i muri (solo se strategicamente utile)
        State = game(player(_, _, W1), _, _, _),
        (W1 > 3 ->
            % Se abbiamo più di 3 muri, considera anche i muri
            findall_wall_moves(State, AllWallMoves),
            findall(Score-WM, (
                member(WM, AllWallMoves),
                make_move(State, WM, NextState),
                evaluate(NextState, Score)
            ), ScoredWalls),
            (ScoredWalls \= [] ->
                sort(ScoredWalls, SortedWalls),
                reverse(SortedWalls, TopWalls),
                take_n(3, TopWalls, BestWalls),  % Prendi i migliori 3 muri
                findall(WM, member(_-WM, BestWalls), WallMoves),
                append(PawnMoves, WallMoves, CombinedMoves)
            ;
                CombinedMoves = PawnMoves
            )
        ;
            CombinedMoves = PawnMoves
        ),

        % Usa minimax su questo set ridotto di mosse migliori
        (CombinedMoves \= [] ->
            (minimax_limited(State, CombinedMoves, 5, -1000, 1000, TempMove, _) ->
                Move = TempMove
            ;
                % Minimax fallito, usa la prima mossa migliore
                CombinedMoves = [Move|_]
            )
        ;
            % Nessuna mossa valida trovata, usa qualsiasi mossa del pedone
            AllPawnMoves = [Move|_]
```

```
340         )
341     ;
342         % Nessuna mossa del pedone disponibile, prova qualsiasi mossa valida
343         findall_valid_moves(State, AllMoves),
344         (AllMoves \= [] -> AllMoves = [Move|_] ; fail)
345     ).
346 
347 % ========================================
348 % FUNZIONI HELPER PER L'AI
349 % ========================================
350 
351 % Prende i primi N elementi da una lista
352 take_n(0, _, []) :- !.
353 take_n(_, [], []) :- !.
354 take_n(N, [H|T], [H|R]) :-
355     N > 0,
356     N1 is N - 1,
357     take_n(N1, T, R).
358 % ========================================
359 % ALGORITMO MINIMAX CON MOSSE LIMITATE
360 % ========================================
361 % Versione ottimizzata di minimax che lavora su un set pre-selezionato
362 % di mosse invece di generarle tutte. Usato dalla difficoltà "extreme".
363 
364 % Caso base: profondità 0, valuta lo stato
365 minimax_limited(State, _Moves, 0, _, _, nil, Score) :-
366     evaluate(State, Score), !.
367 
368 % Caso base: gioco terminato
369 minimax_limited(State, _, _, _, _, nil, BestScore) :-
370     game_over(State, Winner),
371     (Winner == p2 -> BestScore = 1000 ; BestScore = -1000), !.
372 
373 % Caso ricorsivo: valuta le mosse limitate
374 minimax_limited(State, Moves, Depth, Alpha, Beta, BestMove, BestScore) :-
375     Depth > 0,
376     Moves \= [],
377     State = game(_, _, _, Turn),
378     (Turn == p2 ->
379         % Turno dell'AI: massimizza il punteggio
380         maximize_limited(Moves, State, Depth, Alpha, Beta, nil, -10000, BestMove, BestScore)
381     ;
382         % Turno del giocatore: minimizza il punteggio
383         minimize_limited(Moves, State, Depth, Alpha, Beta, nil, 10000, BestMove, BestScore)
384     ).
385 
386 % Massimizzazione (AI vuole massimizzare)
387 maximize_limited([], _, _, _, _, BestMove, BestScore, BestMove, BestScore) :- !.
388 maximize_limited([Move|Rest], State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
389 FinalMove, FinalScore) :-
390     (make_move(State, Move, NextState) ->
391         NewDepth is Depth - 1,
392         findall_pawn_moves(NextState, NextMoves),
393         (NextMoves \= [] ->
394             (minimax_limited(NextState, NextMoves, NewDepth, Alpha, Beta, _, Score) ->
395                 true
396             ;
397                 Score = CurrentBestScore
```

```
            )
        ;
            evaluate(NextState, Score)
        ),
        (Score > CurrentBestScore ->
            NewBestMove = Move,
            NewBestScore = Score
        ;
            NewBestMove = CurrentBestMove,
            NewBestScore = CurrentBestScore
        ),
        % Potatura Alpha-Beta
        (NewBestScore >= Beta ->
            FinalMove = NewBestMove,
            FinalScore = NewBestScore
        ;
            NewAlpha is max(Alpha, NewBestScore),
            maximize_limited(Rest, State, Depth, NewAlpha, Beta, NewBestMove, NewBestScore,
FinalMove, FinalScore)
        )
    ;
        % Mossa fallita, salta
        maximize_limited(Rest, State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
FinalMove, FinalScore)
    ).

% Minimizzazione (giocatore vuole minimizzare)
minimize_limited([], _, _, _, _, BestMove, BestScore, BestMove, BestScore) :- !.
minimize_limited([Move|Rest], State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
FinalMove, FinalScore) :-
    (make_move(State, Move, NextState) ->
        NewDepth is Depth - 1,
        findall_pawn_moves(NextState, NextMoves),
        (NextMoves \= [] ->
            (minimax_limited(NextState, NextMoves, NewDepth, Alpha, Beta, _, Score) ->
                true
            ;
                Score = CurrentBestScore
            )
        ;
            evaluate(NextState, Score)
        ),
        (Score < CurrentBestScore ->
            NewBestMove = Move,
            NewBestScore = Score
        ;
            NewBestMove = CurrentBestMove,
            NewBestScore = CurrentBestScore
        ),
        % Potatura Alpha-Beta
        (NewBestScore =< Alpha ->
            FinalMove = NewBestMove,
            FinalScore = NewBestScore
        ;
            NewBeta is min(Beta, NewBestScore),
            minimize_limited(Rest, State, Depth, Alpha, NewBeta, NewBestMove, NewBestScore,
FinalMove, FinalScore)
        )
```

```
    ;
        % Mossa fallita, salta
        minimize_limited(Rest, State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
FinalMove, FinalScore)
    ).
% ========================================
% STRATEGIA GREEDY (per difficoltà media)
% ========================================
% Sceglie la mossa che porta al punteggio più alto secondo la funzione di valutazione.
best_greedy_move(State, Moves, BestMove) :-
    findall(Score-M, (
        member(M, Moves),
        make_move(State, M, NextState),
        evaluate(NextState, Score)
    ), ScoredMoves),
    sort(ScoredMoves, Sorted),
    reverse(Sorted, [_-BestMove|_]).

% ========================================
% FUNZIONI PER GENERARE LE MOSSE
% ========================================

% Trova tutte le mosse valide (pedone + muri)
findall_valid_moves(State, Moves) :-
    findall_pawn_moves(State, PawnMoves),
    findall_wall_moves(State, WallMoves),
    append(PawnMoves, WallMoves, Moves).

% Trova tutte le mosse valide del pedone
findall_pawn_moves(State, Moves) :-
    board_size(Size),
    findall(move(X, Y), (
        between(1, Size, X),
        between(1, Size, Y),
        is_valid_move(State, move(X, Y))
    ), Moves).

% Trova tutte le mosse valide per piazzare muri
findall_wall_moves(State, Moves) :-
    board_size(Size),
    MaxWall is Size - 1,
    findall(place_wall(X, Y, O), (
        between(1, MaxWall, X),
        between(1, MaxWall, Y),
        member(O, [h, v]),
        is_valid_move(State, place_wall(X, Y, O))
    ), Moves).

% ========================================
% ALGORITMO MINIMAX STANDARD
% ========================================
% Implementazione classica di minimax con potatura Alpha-Beta.
% Usato dalle difficoltà "hard" e "medium".

% Caso base: profondità 0, valuta lo stato corrente
minimax(State, 0, _, _, nil, Score) :-
    evaluate(State, Score), !.

```

```
% Caso base: il gioco è terminato
minimax(State, _, _, _, nil, BestScore) :-
    game_over(State, Winner),
    (Winner == p2 -> BestScore = 1000 ; BestScore = -1000), !.

% Caso ricorsivo: genera e valuta tutte le mosse del pedone
minimax(State, Depth, Alpha, Beta, BestMove, BestScore) :-
    findall_pawn_moves(State, Moves),
    (Moves \= [] ->
        State = game(_, _, _, Turn),
        (Turn == p2 ->
            % Turno dell'AI: massimizza
            maximize(Moves, State, Depth, Alpha, Beta, nil, -10000, BestMove, BestScore)
        ;
            % Turno del giocatore: minimizza
            minimize(Moves, State, Depth, Alpha, Beta, nil, 10000, BestMove, BestScore)
        )
    ;
        % Nessuna mossa disponibile, valuta lo stato
        evaluate(State, BestScore),
        BestMove = nil
    ).

% Massimizzazione: cerca la mossa con il punteggio più alto
maximize([], _, _, _, _, BestMove, BestScore, BestMove, BestScore).
maximize([Move|Rest], State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
FinalBestMove, FinalBestScore) :-
    make_move(State, Move, NextState),
    NewDepth is Depth - 1,
    minimax(NextState, NewDepth, Alpha, Beta, _, Score),
    (Score > CurrentBestScore ->
        NewBestScore = Score, NewBestMove = Move
    ;
        NewBestScore = CurrentBestScore, NewBestMove = CurrentBestMove
    ),
    % Potatura Beta: se il punteggio supera Beta, interrompi
    (NewBestScore >= Beta ->
        FinalBestMove = NewBestMove, FinalBestScore = NewBestScore
    ;
        NewAlpha is max(Alpha, NewBestScore),
        maximize(Rest, State, Depth, NewAlpha, Beta, NewBestMove, NewBestScore, FinalBestMove,
FinalBestScore)
    ).
% Minimizzazione: cerca la mossa con il punteggio più basso
minimize([], _, _, _, _, BestMove, BestScore, BestMove, BestScore).
minimize([Move|Rest], State, Depth, Alpha, Beta, CurrentBestMove, CurrentBestScore,
FinalBestMove, FinalBestScore) :-
    make_move(State, Move, NextState),
    NewDepth is Depth - 1,
    minimax(NextState, NewDepth, Alpha, Beta, _, Score),
    (Score < CurrentBestScore ->
        NewBestScore = Score, NewBestMove = Move
    ;
        NewBestScore = CurrentBestScore, NewBestMove = CurrentBestMove
    ),
    % Potatura Alpha: se il punteggio scende sotto Alpha, interrompi
    (NewBestScore =< Alpha ->
        FinalBestMove = NewBestMove, FinalBestScore = NewBestScore
```

```
    ;
        NewBeta is min(Beta, NewBestScore),
        minimize(Rest, State, Depth, Alpha, NewBeta, NewBestMove, NewBestScore, FinalBestMove,
FinalBestScore)
    ).

% ========================================
% FUNZIONE DI VALUTAZIONE
% ========================================
% Calcola quanto è favorevole uno stato per l'AI (P2).
% Euristica: Score = (Distanza di P1 dall'obiettivo) - (Distanza di P2)
%
% - Se P2 è vicino all'obiettivo (distanza piccola), lo Score aumenta
% - Se P1 è lontano dall'obiettivo (distanza grande), lo Score aumenta
% - L'AI cerca di massimizzare questo punteggio

evaluate(game(P1, P2, Walls, _), Score) :-
    shortest_path_len(P1, Walls, p1, D1),  % Distanza di P1 dall'obiettivo
    shortest_path_len(P2, Walls, p2, D2),  % Distanza di P2 dall'obiettivo
    Score is D1 - D2.                       % Punteggio finale

% Calcola la lunghezza del percorso più breve verso l'obiettivo
shortest_path_len(player(X, Y, _), Walls, Player, Dist) :-
    goal_row(Player, GoalY),
    bfs_dist([[X, Y, 0]], GoalY, Walls, [], Dist).

% BFS con calcolo della distanza
% Caso base: raggiunta la riga obiettivo
bfs_dist([[_, Y, D]|_], GoalY, _, _, D) :- Y == GoalY, !.
% Caso ricorsivo: continua la ricerca
bfs_dist([[X, Y, D]|Rest], GoalY, Walls, Visited, Dist) :-
    D1 is D + 1,  % Incrementa la distanza
    findall([NX, NY, D1], (
        adjacent(X, Y, NX, NY),
        valid_pos(NX, NY),
        \+ wall_blocks(X, Y, NX, NY, Walls),
        \+ member([NX, NY], Visited),
        \+ member([NX, NY, _], [[X, Y, D]|Rest])
    ), NextMoves),
    append(Rest, NextMoves, NewQueue),
    bfs_dist(NewQueue, GoalY, Walls, [[X, Y]|Visited], Dist).
% ========================================
% INTERFACCIA PER L'UI
% ========================================
% Restituisce le mosse valide del pedone per l'interfaccia utente.
get_valid_pawn_moves(State, Moves) :-
findall([X, Y], is_valid_move(State, move(X, Y)), Moves)

```

server.pl

```
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).
:- use_module(library(http/http_files)).
:- use_module(library(http/http_path)).
:- use_module(library(http/http_parameters)).

:- use_module(quoridor).

% --- Server Setup ---

% Salva la directory del progetto durante il caricamento
:- dynamic web_directory/1.
:- prolog_load_context(directory, Dir),
   atom_concat(Dir, '/web', WebDir),
   assertz(web_directory(WebDir)).

% Handler per i file statici
:- http_handler(root(.), serve_static_files, [prefix]).
:- http_handler(root(game/start), start_game_handler, []).
:- http_handler(root(game/move), move_handler, []).

server(Port) :-
    http_server(http_dispatch, [port(Port)]).

% --- Handlers ---

% Handler per servire i file statici dalla cartella 'web'
serve_static_files(Request) :-
    web_directory(WebDir),
    http_reply_from_files(WebDir, [], Request).

start_game_handler(_Request) :-
    initial_state(State),
    state_to_json(State, JSON),
    reply_json(JSON).

move_handler(Request) :-
    http_read_json_dict(Request, Dict),
    % Dict contains: difficulty, state (optional, or we track it?), move
    % For simplicity, let's assume the client sends the current state and
the move.
    % Ideally, server should track state, but stateless is easier for
dev.
    % Let's expect: { "difficulty": "easy", "move": { "type": "move",
"x": 5, "y": 8 }, "state": ... }
    % Actually, simpler: Client sends move, Server applies, then Server
runs AI, returns NEW state.

    atom_string(Difficulty, Dict.difficulty),
    CurrentState = Dict.state,
    PlayerMove = Dict.move,

    json_to_state(CurrentState, State),
```

```
    json_to_move(PlayerMove, Move),

    format(user_error, '~n=== DEBUG ===~n', []),
    format(user_error, 'Received Move: ~w~n', [Move]),
    format(user_error, 'Reconstructed State: ~w~n', [State]),
    format(user_error, 'Testing is_valid_move...~n', []),

    (is_valid_move(State, Move) ->
        format(user_error, 'Move is valid, applying...~n', []),
        make_move(State, Move, StateAfterPlayer),
        format(user_error, 'StateAfterPlayer: ~w~n', [StateAfterPlayer]),
        (game_over(StateAfterPlayer, _) ->
            format(user_error, 'Game over after player move~n', []),
            FinalState = StateAfterPlayer,
            Winner = "player"
        ;
            format(user_error, 'Game continues, AI turn...~n', []),
            % AI Turn
            (choose_move(Difficulty, StateAfterPlayer, AIMove) ->
                format(user_error, 'AI chose: ~w~n', [AIMove]),
                make_move(StateAfterPlayer, AIMove, FinalState),
                format(user_error, 'FinalState: ~w~n', [FinalState]),
                (game_over(FinalState, _) -> Winner = "ai" ; Winner =
null)
            ;
                % AI cannot move? Should not happen if game not over
                format(user_error, 'AI cannot move!~n', []),
                FinalState = StateAfterPlayer, Winner = null
            )
        ),
        format(user_error, 'Converting FinalState to JSON...~n', []),
        state_to_json(FinalState, ResponseState),
        format(user_error, 'ResponseState: ~w~n', [ResponseState]),
        format(user_error, 'Move ACCEPTED~n==========~n~n', []),
        reply_json(_{valid: true, state: ResponseState, winner: Winner})
    ;
        format(user_error, 'Move REJECTED~n==========~n~n', []),
        reply_json(_{valid: false})
    ).

% --- JSON Conversion Helpers ---

state_to_json(game(player(X1, Y1, W1), player(X2, Y2, W2), Walls, Turn),
_{
    p1: _{x: X1, y: Y1, walls: W1},
    p2: _{x: X2, y: Y2, walls: W2},
    walls: WallList,
    turn: Turn,
    valid_moves: []
}) :-
    findall(_{x: WX, y: WY, o: O}, member(wall(WX, WY, O), Walls),
WallList).

json_to_state(Dict, game(player(X1, Y1, W1), player(X2, Y2, W2), Walls,
Turn)) :-
    ensure_number(Dict.p1.x, X1), ensure_number(Dict.p1.y, Y1),
ensure_number(Dict.p1.walls, W1),

```

```
    ensure_number(Dict.p2.x, X2), ensure_number(Dict.p2.y, Y2),
ensure_number(Dict.p2.walls, W2),
    atom_string(Turn, Dict.turn), % Ensure Turn is an atom
    % Convert walls list
    maplist(json_wall_to_term, Dict.walls, Walls).

json_wall_to_term(_{x: X, y: Y, o: O}, wall(X, Y, AtomO)) :-
    atom_string(AtomO, O).

json_to_move(_{type: "move", x: X, y: Y}, move(NX, NY)) :-
    ensure_number(X, NX), ensure_number(Y, NY).
json_to_move(_{type: "wall", x: X, y: Y, o: O}, place_wall(NX, NY,
AtomO)) :-
    ensure_number(X, NX), ensure_number(Y, NY),
    atom_string(AtomO, O).

ensure_number(N, N) :- number(N), !.
ensure_number(S, N) :- string(S), number_string(N, S), !.
ensure_number(A, N) :- atom(A), atom_number(A, N), !.
```

index.html

```
1
2   <!DOCTYPE html>
3   <html lang="en">
4   <head>
5       <meta charset="UTF-8">
6       <meta name="viewport" content="width=device-width, initial-
7   scale=1.0">
8       <title>Prolog Quoridor</title>
9       <link rel="stylesheet" href="style.css">
10  </head>
11  <body>
12      <div id="modal" class="modal">
13          <div class="modal-content">
14              <h2>Select Difficulty</h2>
15              <button onclick="startGame('easy')">Easy</button>
16              <button onclick="startGame('medium')">Medium</button>
17              <button onclick="startGame('hard')">Hard</button>
18          </div>
19      </div>
20
21      <div class="container">
22          <div class="sidebar">
23              <h1>Quoridor</h1>
24              <div id="status">Player's Turn</div>
25              <div class="stats">
26                  <p>Your Walls: <span id="p1-walls">10</span></p>
27                  <p>AI Walls: <span id="p2-walls">10</span></p>
28              </div>
29              <div class="controls">
30                  <p>Wall Orientation:</p>
31                  <button id="toggle-orientation"
32  onclick="toggleOrientation()">Horizontal</button>
33              </div>
34              <button onclick="location.reload()">New Game</button>
35          </div>
36          <div id="board" class="board">
37              <!-- Grid generated by JS -->
38          </div>
39      </div>
40
41      <script src="game.js"></script>
42  </body>
43  </html>
```

style.css

```css
:root {
    --bg-color: #1a1a2e;
    --board-bg: #16213e;
    --cell-color: #0f3460;
    --cell-hover: #1a5f7a;
    --p1-color: #e94560;
    --p2-color: #4cc9f0;
    --wall-color: #fca311;
    --text-color: #ffffff;
    --gap: 10px;
    --cell-size: 50px;
}

body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: var(--bg-color);
    color: var(--text-color);
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
}

.container {
    display: flex;
    gap: 40px;
}

.sidebar {
    display: flex;
    flex-direction: column;
    gap: 20px;
}

.board {
    display: grid;
    grid-template-columns: repeat(9, var(--cell-size));
    grid-template-rows: repeat(9, var(--cell-size));
    gap: var(--gap);
    background-color: var(--board-bg);
    padding: var(--gap);
    border-radius: 10px;
    position: relative;
}

.cell {
    width: var(--cell-size);
    height: var(--cell-size);
    background-color: var(--cell-color);
    border-radius: 5px;
    cursor: pointer;
    display: flex;
    justify-content: center;
    align-items: center;
```

```
    position: relative;
}

.cell:hover {
    background-color: var(--cell-hover);
}

.pawn {
    width: 80%;
    height: 80%;
    border-radius: 50%;
    box-shadow: 0 0 10px rgba(0,0,0,0.5);
}

.p1 { background-color: var(--p1-color); }
.p2 { background-color: var(--p2-color); }

/* Walls */
.wall-h, .wall-v {
    position: absolute;
    background-color: var(--wall-color);
    border-radius: 2px;
    z-index: 10;
    pointer-events: none; /* Let clicks pass through for now, logic
handled by gap clicks */
}

.wall-h {
    height: var(--gap);
    width: calc(2 * var(--cell-size) + var(--gap));
    margin-top: calc(var(--cell-size) + var(--gap) / 2 - var(--gap) / 2);
/* Centered in gap */
}

.wall-v {
    width: var(--gap);
    height: calc(2 * var(--cell-size) + var(--gap));
    margin-left: calc(var(--cell-size) + var(--gap) / 2 - var(--gap) /
2);
}

/* Wall placement hitboxes (invisible but clickable) */
.gap-h {
    position: absolute;
    height: var(--gap);
    width: calc(2 * var(--cell-size) + var(--gap));
    background: rgba(255, 255, 255, 0.1);
    cursor: pointer;
    z-index: 5;
}
.gap-h:hover { background: rgba(252, 163, 17, 0.5); }

.gap-v {
    position: absolute;
    width: var(--gap);
    height: calc(2 * var(--cell-size) + var(--gap));
    background: rgba(255, 255, 255, 0.1);
    cursor: pointer;
```

```
    z-index: 5;
}
.gap-v:hover { background: rgba(252, 163, 17, 0.5); }


/* Modal */
.modal {
    position: fixed;
    top: 0; left: 0; width: 100%; height: 100%;
    background: rgba(0,0,0,0.8);
    display: flex;
    justify-content: center;
    align-items: center;
    z-index: 100;
}

.modal-content {
    background: var(--board-bg);
    padding: 40px;
    border-radius: 10px;
    text-align: center;
}

button {
    padding: 10px 20px;
    margin: 10px;
    font-size: 16px;
    cursor: pointer;
    background: var(--p1-color);
    border: none;
    color: white;
    border-radius: 5px;
}
button:hover { opacity: 0.9; }
```

game.js

```
let gameState = null;
let difficulty = 'easy';
let wallOrientation = 'h'; // 'h' or 'v'

const cellSize = 50;
const gapSize = 10;

function startGame(level) {
    difficulty = level;
    document.getElementById('modal').style.display = 'none';

    fetch('/game/start')
        .then(res => {
            if (!res.ok) throw new Error('Server Error: ' + res.statusText);
            return res.json();
        })
        .then(data => {
            gameState = data;
            renderBoard();
        })
        .catch(err => {
            alert("Error starting game: " + err.message + "\nCheck Prolog console for
details.");
        });
}

function toggleOrientation() {
    wallOrientation = wallOrientation === 'h' ? 'v' : 'h';
    document.getElementById('toggle-orientation').innerText =
        wallOrientation === 'h' ? 'Horizontal' : 'Vertical';
}

function renderBoard() {
    const board = document.getElementById('board');
    board.innerHTML = ''; // Clear

    // Render Cells
    for (let y = 1; y <= 9; y++) {
        for (let x = 1; x <= 9; x++) {
            const cell = document.createElement('div');
            cell.className = 'cell';
            cell.dataset.x = x;
            cell.dataset.y = y;
            cell.onclick = () => handleMove(x, y);

            // Render Pawns
            if (gameState.p1.x === x && gameState.p1.y === y) {
                const p1 = document.createElement('div');
                p1.className = 'pawn p1';
                cell.appendChild(p1);
            }
            if (gameState.p2.x === x && gameState.p2.y === y) {
                const p2 = document.createElement('div');
                p2.className = 'pawn p2';
                cell.appendChild(p2);
```

```
            }

            // Highlight Valid Moves
            if (gameState.valid_moves && gameState.turn === 'p1') {
                const isValid = gameState.valid_moves.some(m => m[0] === x && m[1] ===
y);
                if (isValid) {
                    cell.classList.add('valid-move');
                }
            }

            board.appendChild(cell);
        }
    }

    // Render Walls
    gameState.walls.forEach(w => {
        const wall = document.createElement('div');
        wall.className = w.o === 'h' ? 'wall-h' : 'wall-v';
        // Calculate position in pixels relative to board
        // Grid gap logic is tricky in CSS grid, absolute positioning is easier for
walls
        // Cell (x,y) top-left is:
        // Left: (x-1)*(cellSize+gap) + gap
        // Top: (y-1)*(cellSize+gap) + gap

        const left = (w.x - 1) * (cellSize + gapSize) + gapSize;
        const top = (w.y - 1) * (cellSize + gapSize) + gapSize;

        if (w.o === 'h') {
            wall.style.left = left + 'px';
            wall.style.top = (top + cellSize) + 'px'; // In the gap below the row
        } else {
            wall.style.left = (left + cellSize) + 'px'; // In the gap right of the col
            wall.style.top = top + 'px';
        }
        board.appendChild(wall);
    });

    // Render Wall Placement Hitboxes (Gaps)
    // We need clickable areas between cells.
    // Horizontal gaps: between row Y and Y+1, spanning 2 cols (X, X+1)
    // Vertical gaps: between col X and X+1, spanning 2 rows (Y, Y+1)

    for (let y = 1; y <= 8; y++) {
        for (let x = 1; x <= 8; x++) {
            // We can place a wall at (x,y)
            const hitbox = document.createElement('div');
            hitbox.className = wallOrientation === 'h' ? 'gap-h' : 'gap-v';

            const left = (x - 1) * (cellSize + gapSize) + gapSize;
            const top = (y - 1) * (cellSize + gapSize) + gapSize;

            if (wallOrientation === 'h') {
                hitbox.style.left = left + 'px';
                hitbox.style.top = (top + cellSize) + 'px';
            } else {
                hitbox.style.left = (left + cellSize) + 'px';
```

```
                hitbox.style.top = top + 'px';
            }

            hitbox.onclick = (e) => {
                e.stopPropagation();
                handleWall(x, y);
            };
            board.appendChild(hitbox);
        }
    }

    // Update Stats
    document.getElementById('p1-walls').innerText = gameState.p1.walls;
    document.getElementById('p2-walls').innerText = gameState.p2.walls;
    document.getElementById('status').innerText =
        gameState.turn === 'p1' ? "Your Turn" : "AI Thinking...";
}

function handleMove(x, y) {
    if (gameState.turn !== 'p1') return;
    sendAction({ type: 'move', x: x, y: y });
}

function handleWall(x, y) {
    if (gameState.turn !== 'p1') return;
    sendAction({ type: 'wall', x: x, y: y, o: wallOrientation });
}

function sendAction(move) {
    fetch('/game/move', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            difficulty: difficulty,
            state: gameState,
            move: move
        })
    })
        .then(res => {
            console.log('Response status:', res.status);
            return res.json();
        })
        .then(data => {
            console.log('Response data:', data);
            console.log('data.valid:', data.valid);
            if (data.valid) {
                gameState = data.state;
                renderBoard();
                if (data.winner) {
                    setTimeout(() => alert(data.winner === 'player' ? "You Win!" : "AI
Wins!"), 100);
                }
            } else {
                alert("Invalid Move!");
            }
        })
        .catch(err => {
            console.error('Fetch error:', err);
```

```
            alert("Error: " + err.message);
        });
}
```

Appendice B

Dal punto di vista matematico, il gioco Quoridor rappresenta un interessante problema di *pathfinding* su un grafo dinamico, caratterizzato da proprietà topologiche uniche che ne garantiscono la risolubilità.

Modellazione tramite Teoria dei Grafi

Il piano di gioco può essere formalizzato come un grafo non orientato $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, dove:

- V è l'insieme degli 81 vertici (le caselle della scacchiera).
- E è l'insieme degli archi che collegano vertici adiacenti (le mosse possibili).

La peculiarità di Quoridor risiede nella natura dinamica di E. L'azione di posizionare una barriera non aggiunge elementi al sistema, ma agisce come un operatore di **rimozione di archi**. Piazzare un muro corrisponde a eliminare dal grafo gli archi che collegano le caselle separate dal muro stesso, modificando in tempo reale la topologia dello spazio di ricerca.

Teorema di Connettività

Una delle caratteristiche fondamentali di Quoridor è l'impossibilità di raggiungere una situazione di stallo topologico ("deadlock"). Questa proprietà è garantita da una regola esplicita che funge da invariante fondamentale del sistema. Possiamo formalizzare questa proprietà nel seguente enunciato:

Teorema:

Sia G_t la configurazione del grafo al turno t. Sia P_i la posizione del giocatore i e $T_i \subset V$ l'insieme dei nodi che costituiscono il suo obiettivo (la riga opposta). Una mossa M che trasforma il grafo in G_{t+1} è ammissibile se e solo se:

$$\forall i \in \{1,2\}, \exists \pi : P_i \rightsquigarrow t, con\ t \in T_i$$

Ovvero, deve esistere almeno un cammino π nel grafo G_{t+1} che colleghi ogni giocatore al proprio obiettivo.

Questa condizione implica che Quoridor è un gioco in cui la connettività tra punto di partenza e destinazione è preservata per induzione: è vera allo stato iniziale (grafo completo senza muri) e ogni mossa legale mantiene vera la proprietà.

Complessità computazionale

Nonostante le regole semplici, la complessità computazionale di Quoridor è sorprendentemente elevata, tanto da essere classificato come problema **PSPACE-hard**. Studi accademici hanno stimato le dimensioni dello spazio di ricerca:

- **State-space complexity:** il numero di posizioni legali raggiungibili è stimato intorno a 10^{42}.
- **Game-tree complexity:** il numero di partite possibili è stimato nell'ordine di 10^{162}.

Per fornire un termine di paragone, la complessità dell'albero di gioco degli scacchi è stimata intorno a 10^{120}. Questo dato evidenzia come la combinatoria data dal posizionamento dei muri (che possono essere piazzati in 128 posizioni diverse, con orientamento orizzontale o verticale) renda intrattabile qualsiasi approccio di risoluzione esaustiva ("brute force").
La necessità di utilizzare algoritmi di ricerca euristica come **Minimax** con potatura **Alpha-Beta**, come implementato in questo progetto, deriva direttamente da questa esplosione combinatoria: non essendo possibile calcolare la soluzione ottima globale (Strategia di Nash perfetta) in tempi ragionevoli, è necessario approssimarla limitando l'orizzonte di ricerca.