



Adaptive Bitboard Engine

Report Tecnico CRISP-DM

Versione 2.1 - Eliminazione Duplicati e Correzione Argomenti

Informazioni Generali

Corso di Laurea:	Informatica
Università:	Università degli Studi di Salerno (UNISA)
Docente:	Chiar.mo Prof. PALOMBA
Data:	13/02/2026
Anno Accademico:	2025/2026

Membri Del Gruppo

MANFREDINI Umberto	Matricola 0512119797
ROMANO Pino Fiorello	Matricola 0512120259

Revision History

Data	Vers.	Descrizione	Autore
30/01/2026	1.0	Stesura iniziale	Gruppo
05/02/2026	1.1	Capitoli Architettura e Modeling	Gruppo
12/02/2026	2.0	Ristrutturazione CRISP-DM	Gruppo
13/02/2026	2.1	Eliminazione duplicati e Correzione Argomenti	Gruppo

Indice

1 Business Understanding	5
1.1 Visione del Progetto	5
1.2 Architettura del Sistema (Pattern MVC)	5
1.3 Obiettivi Tecnici	5
2 Data Understanding	6
2.1 Bitboard: Rappresentazione dello Stato	6
2.1.1 Mappatura Spaziale	6
2.1.2 Maschere di Stato	6
2.1.3 Top Mask e Gestione Gravità	6
2.2 Persistenza Dati: SQLite e UPSERT	7
2.2.1 Schema Relazionale	7
2.2.2 Logica UPSERT	7
3 Data Preparation	8
3.1 Pattern Recognition tramite Bit-Shifting	8
3.1.1 Rilevamento Vittorie (4-in-fila)	8
3.1.2 Threat Mask Avanzata	8
3.2 Filtri di Validità Fisica	8
3.2.1 Playable Mask	9
3.2.2 Alive Pairs Logic	9
3.3 Accelerazione Hardware: POPCNT	9
4 Modeling	10
4.1 Motore di Ricerca: Minimax e Alpha-Beta Pruning	10
4.1.1 Ottimizzazione Alpha-Beta	10
4.1.2 Depth-Weighted Scoring	10
4.2 Ottimizzazione della Memoria: Transposition Table	10
4.3 Opening Manager: Algoritmo UCB1	11
4.4 Opponent Profiling	11
4.4.1 Rilevamento e Smoothing	11
4.4.2 Bias Cooling (Meccanismo di Raffreddamento)	12
4.4.3 Dinamiche Predatorie: Threat Inflation	12
4.4.4 Defense Relaxation	12
5 Evaluation	12
5.1 Metodologia di Testing e Archetipi Indotti	13
5.1.1 Iniezione di Rumore	13

5.2	Metriche di Analisi e Monitoraggio	13
5.3	Risultati dei Benchmark	14
6	Deployment	15
6.1	Architettura dell'Interfaccia (FSM)	15
6.2	Effettistica Vettoriale e Rendering Procedurale	15
6.3	Feedback Decisionale in Real-Time	16
7	Conclusioni e Sviluppi Futuri	17
7.1	Risultati Raggiunti	17
7.2	Sviluppi Futuri	17
	Bibliografia	18

1. Business Understanding

1.1. Visione del Progetto

Il progetto **Adaptive Bitboard Engine** nasce per superare i limiti delle intelligenze artificiali "statiche" nel gioco del Forza 4. L'obiettivo primario non è la creazione di un agente imbattibile in senso assoluto (problema già risolto matematicamente da Allis [6]), ma lo sviluppo di un sistema **adattivo** capace di profilare l'avversario in tempo reale.

Il sistema deve essere in grado di evolvere la propria strategia, passando dalla pura ottimizzazione matematica allo sfruttamento dei bias cognitivi dell'umano (es. cecità diagonale), rendendo l'esperienza di gioco dinamica e personalizzata secondo i concetti dell'Opponent Modelling [4].

1.2. Architettura del Sistema (Pattern MVC)

Per garantire modularità e scalabilità, il software adotta rigorosamente il paradigma **Model-View-Controller (MVC)**:

- **Engine (Model):** Rappresenta il cervello logico del sistema. Gestisce la rappresentazione *bitwise* della scacchiera, le regole di validazione e ospita il motore decisionale (Minimax + Profiler). Non ha alcuna dipendenza dalle librerie grafiche.
- **View (Vista):** Implementata tramite Pygame, è responsabile esclusivamente del rendering grafico (in stile Cyberpunk/Neon) e del feedback visivo (es. "Tug of War" bar per il vantaggio).
- **Controller (Logica di Flusso):** Agisce da orchestratore. Intercetta gli input dell'utente, gestisce la macchina a stati finiti (FSM) per i menu e sincronizza i turni tra l'umano e il bot.

1.3. Obiettivi Tecnici

- **Trasparenza (White Box):** A differenza delle reti neurali "Black Box", ogni decisione dell'IA deve essere giustificabile tramite parametri leggibili (pesi euristici) [5].
- **Performance:** Il motore deve garantire una profondità di ricerca elevata ($d \geq 4$) in tempo reale (< 2s per mossa).
- **Persistenza:** Capacità di salvare lo storico delle partite e i profili psicologici degli avversari su database relazionale.

2. Data Understanding

In questa fase del ciclo CRISP-DM [8], analizziamo l'architettura logica del sistema e le strutture dati fondamentali utilizzate per rappresentare il dominio del problema. La scelta di strutture dati efficienti è il prerequisito per un motore decisionale performante.

2.1. Bitboard: Rappresentazione dello Stato

La rappresentazione classica a matrice 6×7 è inefficiente per la ricerca profonda. Il sistema adotta i **Bitboards** [2]: due interi a 64-bit (*unsigned long long*) che mappano l'intera griglia di gioco.

2.1.1. Mappatura Spaziale

La scacchiera è mappata in memoria in ordine *Column-Major* (colonna per colonna). Per una griglia $H = 6$, $W = 7$, ogni colonna occupa 7 bit (6 per le celle + 1 bit "sentinella" per evitare overflow verticali).

L'indice del bit per la cella (*riga, colonna*) è calcolato come:

$$\text{Index}(r, c) = r + (c \cdot (H + 1)) \quad (1)$$

Questa struttura permette di calcolare la mossa successiva semplicemente incrementando l'indice bit: `next_cell = current_cell + 1`.

2.1.2. Maschere di Stato

Lo stato è definito dalla coppia di bitboard (P, M):

- **Position (P):** Maschera delle pedine del giocatore corrente.
- **Mask (M):** Maschera di *tutte* le pedine occupate (Giocatore + Avversario).

Per ottenere le pedine dell'avversario, è sufficiente un'operazione XOR:

$$\text{Opponent} = P \oplus M \quad (2)$$

2.1.3. Top Mask e Gestione Gravità

Per gestire la "fisica" del gioco (le pedine cadono in basso), il sistema mantiene un array `heights[7]` che punta al primo bit libero di ogni colonna. Inoltre, una costante `TOP_MASK` identifica la sesta riga di ogni colonna. La verifica "Colonna Piena" diventa un'operazione atomica [2]:

```
if (TOP_MASK & (1 << heights[col])): return False; // Colonna piena
```

2.2. Persistenza Dati: SQLite e UPSERT

Per supportare l'apprendimento continuo, il sistema utilizza un database relazionale embedded (**SQLite**).

2.2.1. Schema Relazionale

1. **Tabella games:** Storico delle partite.

- **id** (PK), **timestamp**, **opponent_id**, **result**.
- **biases_json**: Snapshot dei parametri del Profiler al termine della partita, serializzati in formato JSON per flessibilità.

2. **Tabella opening_book:** Conoscenza delle aperture.

- **state_hash** (PK): Hash univoco dello stato (stringa bitboard).
- **move_col** (PK): La mossa effettuata (0-6).
- **visits**: Numero di volte che la mossa è stata esplorata.
- **total_score**: Somma dei reward (Win/Loss/Draw) accumulati.

2.2.2. Logica UPSERT

L'aggiornamento delle statistiche nel libro delle aperture deve essere rapido e sicuro rispetto alla concorrenza. Utilizziamo la clausola **ON CONFLICT** per eseguire "Insert or Update" atomici:

```
INSERT INTO opening_book (...) VALUES (...)  
ON CONFLICT(state_hash, move_col)  
DO UPDATE SET visits = visits + 1, total_score = ...
```

Questo approccio elimina la necessità di lock applicativi e riduce l'overhead di I/O.

3. Data Preparation

In questa fase del ciclo CRISP-DM, i dati grezzi (lo stato dei bitboard) vengono trasformati in *feature* significative (minacce, opportunità, pattern) utilizzabili dagli algoritmi di Modeling. Grazie alla rappresentazione bitboard, questa fase avviene tramite operazioni di algebra booleana ad altissima efficienza.

3.1. Pattern Recognition tramite Bit-Shifting

Il riconoscimento di configurazioni vincenti o pericolose non richiede cicli `for`, ma operazioni di scorrimento (shift) dell'intero registro a 64 bit. Definiamo P come il bitboard del giocatore corrente.

3.1.1. Rilevamento Vittorie (4-in-fila)

Per verificare una vittoria orizzontale, controlliamo se esistono 4 bit consecutivi impostati a 1. L'algoritmo esegue un AND logico tra il bitboard originale e una sua copia shiftata:

$$m = P \& (P \gg 7) \quad (\text{Coppie orizzontali}) \quad (3)$$

$$res = m \& (m \gg 14) \quad (\text{Quartine orizzontali}) \quad (4)$$

Se $res \neq 0$, esiste almeno un allineamento vincente. La stessa logica si applica alle direzioni verticali ($shift = 1$) e diagonali ($shift = 6$ e $shift = 8$).

3.1.2. Threat Mask Avanzata

Oltre alle vittorie immediate, il sistema deve rilevare minacce future. Vengono calcolate maschere specifiche per diversi tipi di pattern:

- **Open-Ended Three (Pattern .XXX.)**: Tre pedine consecutive con spazi liberi ai lati.
- **Split Three (Pattern XX.X)**: Tre pedine con un "buco" interno. Questo pattern è insidioso perché crea due minacce di completamento separate [3].

Queste maschere vengono generate combinando shift multipli e operazioni `XOR` con la maschera globale delle celle occupate.

3.2. Filtri di Validità Fisica

Nel Forza 4, non tutte le celle vuote sono accessibili. Una minaccia alla riga 5 è irrilevante se la riga 4 è ancora vuota. Per pulire i dati di input, vengono applicati due filtri fondamentali.

3.2.1. Playable Mask

Viene generata una maschera che rappresenta solo le celle legalmente giocabili nel turno corrente (il primo bit libero sopra ogni colonna).

$$\text{Playable} = \sum_{col=0}^6 (1 \ll \text{heights}[col]) \quad (5)$$

L'intersezione (\wedge) tra la *Threat Mask* e la *Playable Mask* restituisce le **Minacce Immediate**, mentre l'intersezione con lo shift della Playable Mask rivela le **Minacce Future** (che si attiveranno al prossimo turno).

3.2.2. Alive Pairs Logic

Un errore comune nelle euristiche statiche è sovrastimare le coppie di pedine bloccate dai bordi [3]. Il filtro "Alive Pairs" scarta le coppie che non hanno "spazio vitale" per crescere. Una coppia è considerata valida come feature solo se:

$$\text{Space}(pair) \geq 2 \text{ celle libere adiacenti} \quad (6)$$

Questo pre-processing impedisce all'IA di perseguire strategie vicolo cieco.

3.3. Accelerazione Hardware: POPCNT

Una volta estratte le feature (es. "tutte le minacce diagonali"), è necessario quantificarle numericamente per la funzione di valutazione. Iterare sui bit è lento. Il sistema sfrutta l'istruzione CPU **POPCNT** (Population Count), accessibile in Python 3.10+ tramite il metodo `int.bit_count()`.

Vantaggio Prestazionale: L'istruzione POPCNT conta il numero di bit a 1 in un intero a 64 bit in un singolo ciclo di clock (su architetture x86_64 moderne), rendendo la valutazione euristica ordini di grandezza più veloce rispetto agli approcci basati su stringhe o loop [2].

4. Modeling

In questo capitolo vengono descritti gli algoritmi decisionali (Minimax), le tecniche di apprendimento (UCB1) e il sistema di adattamento dinamico (Profiler) che costituiscono l'*Adaptive Bitboard Engine*.

4.1. Motore di Ricerca: Minimax e Alpha-Beta Pruning

L'agente decisionale è basato su una variante ottimizzata dell'algoritmo **Minimax** con *Alpha-Beta Pruning* [1]. Dato uno stato S_0 , l'algoritmo esplora l'albero di gioco fino a una profondità d , valutando i nodi foglia tramite una funzione euristica $E(s)$.

4.1.1. Ottimizzazione Alpha-Beta

Per mitigare la complessità esponenziale $O(b^d)$ (dove $b \approx 7$ è il fattore di ramificazione), l'algoritmo mantiene due valori, α (il miglior punteggio garantito per il Maximizer) e β (il miglior punteggio garantito per il Minimizer). Un ramo viene "potato" (pruned) non appena si verifica la condizione $\alpha \geq \beta$. Questa tecnica, unita all'euristica *Center-First Move Ordering* [3] (che esplora prima le colonne 3, 2, 4), permette di raddoppiare la profondità di ricerca effettiva a parità di tempo computazionale.

4.1.2. Depth-Weighted Scoring

Per risolvere il problema dell'indecisione in stati vincenti, il punteggio di una vittoria non è costante. Viene calcolato come:

$$Score_{win} = \pm(10^7 + \text{depth}) \quad (7)$$

Questo termine additivo spinge l'IA a preferire le vittorie più rapide (profondità maggiore residua) e a ritardare il più possibile le sconfitte inevitabili.

4.2. Ottimizzazione della Memoria: Transposition Table

Poiché diverse sequenze di mosse possono condurre alla stessa configurazione della scacchiera (trasposizioni), il motore utilizza una **Transposition Table** (TT) per memorizzare i risultati delle valutazioni passate. Per eliminare il rischio di collisioni, la tabella è indicizzata tramite una **chiave sicura basata su tupla immutabile** dello stato bitboard corrente (B_0, B_1). Ogni entry contiene:

- **Value:** Il punteggio Minimax calcolato.
- **Flag:** Tipo di valore (EXACT, LOWERBOUND (α), UPPERBOUND (β))).

- **Depth:** La profondità a cui è stata effettuata la ricerca.

L'algoritmo riutilizza il valore in cache solo se la profondità memorizzata è maggiore o uguale a quella richiesta, trasformando la complessità temporale in spaziale.

4.3. Opening Manager: Algoritmo UCB1

Per superare il determinismo delle aperture, il sistema implementa un *Opening Book* dinamico gestito dall'algoritmo **UCB1 (Upper Confidence Bound)** [9]. Invece di scegliere sempre la mossa con la media di vittorie più alta (Greedy), l'algoritmo assegna un punteggio *UCB* a ogni mossa candidata j :

$$UCB_j = \underbrace{\frac{w_j}{n_j}}_{\text{Exploitation}} + C \cdot \sqrt{\frac{\ln N}{n_j}} \quad \text{Exploration} \quad (8)$$

Dove:

- w_j : Somma dei reward ottenuti dalla mossa j (normalizzati in $[0, 1]$).
- n_j : Numero di volte che la mossa j è stata giocata.
- N : Numero totale di partite giocate da quella posizione.
- C : Costante di esplorazione (tipicamente $\sqrt{2}$).

Questo approccio garantisce che l'IA esplori periodicamente nuove varianti di apertura, evitando di fossilizzarsi su massimi locali sub-ottimali. Sono implementate soglie di sicurezza (fallback al Minimax) se il punteggio UCB scende sotto una soglia critica, per evitare di ripetere errori palesi.

4.4. Opponent Profiling

Il componente più innovativo del modello è il Profiler, che adatta la funzione di valutazione $E(s)$ in base allo stile dell'avversario [4].

4.4.1. Rilevamento e Smoothing

Il sistema monitora gli errori dell'avversario classificandoli in tre categorie: *Lethal* (mancata vittoria), *Strategic* (mancato blocco) e *Correction* (parata corretta). I bias cognitivi (es. `diagonal_weakness`) vengono aggiornati usando un sistema di **Delta-Smoothing**:

$$B_{t+1} = B_t + \eta \cdot (Error_{type} - B_t) \quad (9)$$

Per bilanciare stabilità statistica e reattività tattica, il *learning rate* η assume valori differenziati in base alla gravità dell'evento rilevato:

- $\eta_{lethal} = 0.8$: garantisce un adattamento quasi istantaneo se l'avversario ignora una mossa vincente giocabile.
- $\eta_{strategic} = 0.3$: applicato a errori tattici (es. mancata difesa di un tris) per una crescita costante e solida del bias.
- $\eta_{correction} = 0.1$: riduce il bias con prudenza quando l'avversario corregge il proprio stile, evitando che un singolo blocco casuale azzeri la profilazione accumulata.

4.4.2. Bias Cooling (Meccanismo di Raffreddamento)

Per prevenire l'*over-confidence* derivante da una profilazione errata, il sistema implementa il **Cooling After Loss**. In caso di sconfitta imprevista, l'IA "punisce" i propri bias: se un parametro supera la soglia di arroganza ($Bias > 2.0$), esso viene ridotto del 15%. Questo forza il modello a riconsiderare l'avversario con maggiore prudenza difensiva.

4.4.3. Dinamiche Predatorie: Threat Inflation

Quando un bias supera una soglia di confidenza (es. > 1.5), l'IA attiva la modalità "Predator" [5]. I pesi euristici per quel vettore di attacco vengono aumentati quadraticamente: $Weight_{final} = (Weight_{base})^2$. Questa **Threat Inflation** trasforma una debolezza rilevata in una priorità assoluta per il Minimax, simulando un "Killer Instinct".

4.4.4. Defense Relaxation

Simmetricamente, se il parametro `threat_underestimation` dell'avversario è elevato, l'IA riduce la penalità per le minacce subite.

Logica: "Non serve sacrificare la mia struttura d'attacco per bloccare una minaccia che l'avversario molto probabilmente non vedrà."

Questo permette all'agente di prendersi rischi calcolati che un Minimax standard non considererebbe mai [5].

5. Evaluation

Nella fase di Evaluation si verifica quantitativamente se il modello soddisfa gli obiettivi prefissati. Poiché il nostro sistema mira all'adattività, non è sufficiente misurare la forza bruta contro un motore perfetto; è necessario dimostrare che l'IA impari a sfruttare le debolezze specifiche di avversari sub-ottimali [3].

5.1. Metodologia di Testing e Archetipi Indotti

Per simulare la variabilità umana e testare la reattività del Profiler, è stata sviluppata una suite di bot di allenamento (*Training Evaluators*), ciascuno con un "difetto cognitivo" programmato [4]:

- **Casual Novice (Simulatore Umano):** Un bot basato su Minimax a bassa profondità ($d = 2$). Simula un giocatore inesperto che riconosce le minacce immediate ma non sa pianificare a lungo termine.
- **Diagonal Blinder:** Un bot a profondità media ($d = 4$) in cui il peso della valutazione delle diagonali è artificialmente ridotto. Serve a verificare se la nostra IA è in grado di rilevare questa cecità e innescare la *Threat Inflation* diagonale.
- **Edge Runner:** Un bot con una "fobia del centro". La sua euristica assegna punteggi negativi alle colonne centrali (2, 3, 4) e positivi a quelle laterali (0, 1, 5, 6).

5.1.1. Iniezione di Rumore

Per evitare che l'IA memorizzi passivamente l'esatta sequenza di un'apertura e si adatti per *overfitting* a un singolo percorso deterministico, ai bot di training è stato applicato un fattore di **Stochastic Noise**:

$$Score_{noisy} = Score_{base} \cdot \mathcal{U}(0.8, 1.2) \quad (10)$$

Questa variabilità casuale ($\pm 20\%$) simula l'imprevedibilità umana e forza l'IA (tramite l'algoritmo UCB1) a costruire un albero delle aperture robusto e generalizzabile.

5.2. Metriche di Analisi e Monitoraggio

I test massivi ("Headless Benchmark", fino a 2000 partite per sessione) vengono valutati attraverso un sistema di monitoraggio duale:

1. **Incremental Progress:** Valuta le performance della singola sessione corrente per validare modifiche recenti al codice.
2. **Lifetime History:** Esegue aggregazioni SQL sull'intero database per analizzare la maratona evolutiva del modello.

Le metriche chiave estratte sono:

- **Win Rate (%):** Percentuale di vittorie.
- **Efficienza di Vittoria (Avg Moves):** Misura la durata media delle partite. Un abbassamento di questo valore nel tempo dimostra che l'IA sta imparando a chiudere le partite più in fretta sfruttando i bias.

5.3. Risultati dei Benchmark

L'integrazione del Profiler combinato con l'algoritmo UCB1 ha prodotto risultati stabili e misurabili:

- **vs Casual Novice:** Il *Win Rate* si è stabilizzato nell'intorno dell'**80%**. L'IA esplora varianti di apertura senza cadere in loop deterministici, vincendo prevalentemente tramite controllo del centro.
- **vs Edge Runner:** Consistenza raggiunta al **75-80%**. L'IA ha imparato a cedere momentaneamente i lati per costruire "Torri" centrali inattaccabili.
- **vs Diagonal Blinder:** Il Diagonal blinder in quanto avversario più ostico e vicino ad un bot di minimax perfetto, ha fatto da ottimo elemento di addestramento portando il tasso di vittoria al **60-65%**.

6. Deployment

L'ultima fase del CRISP-DM prevede il rilascio del modello in un ambiente fruibile dall'utente finale. Nel nostro caso, il *Deployment* coincide con la realizzazione dell'interfaccia grafica (GUI) e del sistema di orchestrazione del gioco.

6.1. Architettura dell'Interfaccia (FSM)

Il flusso dell'applicazione è gestito da una Macchina a Stati Finiti (FSM) implementata nel modulo `controller.py`. Gli stati principali sono:

- **MAIN_MENU**: Gestione dell'ingresso e configurazione.
- **BOT_SELECT**: Scelta dell'avversario (Umano vs IA, IA vs IA, selezione dell'archetipo).
- **GAME**: Il core loop. Gestisce il rendering a 60 FPS, il polling degli eventi (click del mouse) e delega i calcoli pesanti dell'IA in modo da non bloccare il thread grafico.
- **GAME_OVER**: Mostra il modal di vittoria e gestisce le transizioni verso la rivincita o il menu principale.

6.2. Effettistica Vettoriale e Rendering Procedurale

L'interfaccia utente è stata sviluppata con Pygame adottando uno stile **Cyberpunk/Neon**. Per garantire la massima scalabilità e nitidezza (indipendenza dalla risoluzione), si è scelto di **non utilizzare asset statici** (PNG o JPG). Ogni elemento grafico è generato proceduralmente (*Procedural Rendering*).

- **Hollow Board Effect**: La scacchiera non è un'immagine solida. Viene creata in memoria una superficie SRCALPHA (con supporto per la trasparenza) di colore Blu Elettrico. I "buchi" vengono generati disegnando cerchi con canale Alpha a zero (*Blending Mode* di sottrazione).
- **Layering (Profondità Visiva)**: Le pedine vengono renderizzate sul livello inferiore (Layer 0), e la maschera della scacchiera viene sovrapposta (Layer 1). Questo crea un'illusione ottica per cui le pedine sembrano fisicamente inserite nei fori.
- **Neon Glow**: Le pedine umane e dell'IA non sono cerchi piatti, ma composte da cerchi concentrici con gradazioni di colore decrescenti, simulando l'alone luminoso tipico dei tubi al neon.

6.3. Feedback Decisionale in Real-Time

Per enfatizzare la natura "White Box" dell'IA, l'interfaccia fornisce all'utente un feedback continuo su cosa il motore sta pensando:

- **Evaluation Bar (Tug of War):** Una barra centrale divisa tra i due colori. Se il punteggio Minimax è favorevole all'IA, la porzione del suo colore si espande, "spingendo" l'altra. Indica chi è in vantaggio posizionale ben prima della fine della partita.
- **Neural Dashboard (Profiler):** Un pannello laterale che mostra, per ogni turno, le stime del Profiler. Tramite barre di progresso, l'utente può vedere i propri bias quantificati (es. "Diagonal Weakness: 2.15"). Questo elemento trasforma il gioco in uno strumento di analisi cognitiva per l'utente stesso.

7. Conclusioni e Sviluppi Futuri

7.1. Risultati Raggiunti

Il progetto *Adaptive Bitboard Engine* ha dimostrato con successo la fattibilità di un'IA basata su Teoria dei Giochi classica capace di svincolarsi dalla rigidità strategica. L'integrazione di una **Rappresentazione Bitwise** ha garantito le performance hardware necessarie per la ricerca Minimax in tempo reale, mentre il **Profiler Comportamentale** e l'uso di euristiche adattive (Threat Inflation) hanno permesso all'agente di imparare dagli errori dell'avversario. Infine, l'algoritmo **UCB1** interfacciato con SQLite ha fornito una soluzione solida al problema del determinismo in fase di apertura, permettendo al sistema di esplorare senza incorrere nell'*overfitting*.

7.2. Sviluppi Futuri

Sebbene l'architettura attuale sia completa e funzionale, il design modulare (MVC) permette diverse estensioni future:

1. **Monte Carlo Tree Search (MCTS)** [7]: Sostituire l'Alpha-Beta Pruning con MCTS, una tecnica di ricerca probabilistica che ben si adatta a spazi di stati enormi e che potrebbe beneficiare nativamente della nostra velocità nell'esecuzione di partite randomiche (*rollouts*) basate su Bitboard.
2. **Multiplayer Online e API REST**: Separare il *Model* (Engine) dalla *View* creando un backend Python (es. FastAPI) per esporre il motore decisionale come servizio web. Questo permetterebbe di sviluppare client web/mobile e raccogliere dati comportamentali su scala globale, alimentando il database di aperture con partite di giocatori umani reali.

Riferimenti bibliografici

- [1] D. E. Knuth and R. W. Moore, *An analysis of alpha-beta pruning*, in *Artificial Intelligence*, vol. 6, no. 4, pp. 293-326, 1975.
- [2] J. Tromp, *The Fhourstones Benchmark*, <http://tromp.github.io/c4/fhour.html>. (Accesso: Febbraio 2026).
- [3] K. Wang et al., *Research on Different Heuristics for Minimax Algorithm: Insight from Connect-4 Game*, in *Proceedings of 2019 International Conference on Computing and Artificial Intelligence*, 2019.
- [4] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, *Opponent Modelling for Case-Based Adaptive Game AI*, in *Entertainment Computing - ICEC 2004*, Springer, 2004.
- [5] H. Silva et al., *Dynamic Difficulty Adjustment through an Adaptive AI*, in *ResearchGate Publications*, 2017.
- [6] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence*, Ph.D. Thesis, University of Limburg, Maastricht, 1994.
- [7] D. Silver et al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, in *Science*, vol. 362, no. 6419, pp. 1140-1144, 2018.
- [8] R. Wirth and J. Hipp, *CRISP-DM: Towards a standard process model for data mining*, in *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, 2000.
- [9] P. Auer, N. Cesa-Bianchi, and P. Fischer, *Finite-time analysis of the multiarmed bandit problem*, *Machine learning*, 47(2), pp. 235-256, 2002.