

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria
Dell'Informazione per Videogame e Realtà Virtuale

Interazioni Strategiche in una Variante di Forza Quattro Multi-Agente: Gestione dell'Incertezza e Modelli Decisionali Probabilistici.

Strategic Interactions in a Multi-Agent Variant of Connect Four: Managing Uncertainty and Probabilistic Decision Models.

Relatore:

Prof. Aldo Franco Dragoni

Candidato:

Michele Candigliota

Anno Accademico 2025–2026

Indice

Introduzione **1**

1 Inquadramento Teorico **4**

1.1 Intelligenza Artificiale e Giochi da Tavolo 4

1.1.1 I giochi da tavolo come base per l'Intelligenza Artificiale 4

1.1.2 Le origini della teoria computazionale 4

1.1.3 L'evoluzione della ricerca avversariale 5

1.1.4 Forza 4 e la risoluzione di Allis 6

1.2 Tassonomia dei Giochi e il Modello del Forza 4 Classico 7

1.2.1 Dimensioni tassonomiche secondo la teoria dei giochi 7

1.2.2 Formalizzazione matematica di un gioco a due giocatori 8

1.2.3 Vincoli cinematici e fisici di Forza 4 9

1.3 Spazio degli Stati, Albero di Gioco e Complessità di Ricerca 10

1.3.1 Lo spazio degli stati e l'albero di gioco 10

1.3.2 Branching factor, profondità ed esplosione combinatoria 11

1.4 L'Algoritmo Minimax 12

1.4.1 Principio di induzione a ritroso e razionalità perfetta 12

1.4.2 Formulazione ricorsiva e funzione di valore 12

1.4.3 Funzionamento procedurale dell'algoritmo 13

1.4.4 Proprietà teoriche e complessità computazionale 14

1.4.5 Il limite dell'orizzonte e la valutazione euristica 14

1.5 La Potatura Alpha-Beta 15

1.5.1 Principi matematici della potatura e conservazione dell'ottimalità 15

1.5.2 Meccanismo dei Cutoff 16

1.5.3 Procedura di esecuzione e gestione della finestra $[\alpha, \beta]$ 16

1.5.4 Analisi della complessità computazionale 17

1.5.5 L'importanza delle tecniche di Move Ordering 18

1.6 Dai Giochi a Somma Zero ai Sistemi Multi-Agente Asimmetrici 18

1.6.1 Transizione ai sistemi multi-agente 18

1.6.2 L'algoritmo MaxN e le difficoltà di potatura 19

1.6.3 L'Approccio Paranoid 19

1.6.4 Introduzione al Progetto 20

2 Architettura e Sviluppo del Sistema **23**

2.1 Descrizione della Variante di Forza 4: Regole, Asimmetria e Meccaniche di Gioco 23
2.1.1 Geometria e dimensionamento della griglia estesa 23
2.1.2 Ruoli e asimmetria dei tre agenti 23
2.1.3 Il ciclo a quattro fasi 24
2.2 Paradigma Logico e Architettura di Sistema 25
2.2.1 Motivazioni per l'uso della programmazione logica nei giochi da tavolo 25
2.2.2 L'integrazione di sistema tramite Janus 26
2.2.3 Struttura modulare del software 26
2.3 Rappresentazione dello Stato: Bitboard 27
2.3.1 Limiti della rappresentazione matriciale 27
2.3.2 Codifica della griglia a 9 bit e bit sentinella 27
2.3.3 Verifica delle quaterne vincenti con shift bitwise 28
2.3.4 Operazioni di gioco in tempo costante $\mathcal{O}(1)$ 29
2.4 Esplorazione dello Spazio di Ricerca e Ottimizzazioni Avanzate 30
2.4.1 Ordinamento delle Mosse a due livelli 30
2.4.2 La Tabella delle Trasposizioni 31
2.4.3 Filtri tattici preventivi alla radice 32
2.5 Funzione Euristica: Threat-Space Analysis 32
2.5.1 Ispirazione teorica e fondamento combinatorio 32
2.5.2 Classificazione delle finestre di allineamento 33
2.5.3 Valutazioni tattiche avanzate 33
2.5.4 Composizione e pesi della funzione di valutazione 34
2.6 Algoritmo Decisionale: Paranoid Minimax 35
2.6.1 Il modello Paranoid nel ciclo a quattro fasi 35
2.6.2 Approfondimento Iterativo 35
2.6.3 La ricerca di quiescenza 36
2.7 Livelli di Difficoltà e Telemetria Prestazionale 36
2.7.1 Configurazione dei livelli di difficoltà dell'agente 36
2.8 Interfaccia Grafica e Front-end con Pygame 37
2.8.1 Architettura dell'interfaccia grafica 37
2.8.2 Pannello di controllo e telemetria 38
2.8.3 Elaborazione dell'IA e gestione del ciclo di rendering 38
2.8.4 Dall'architettura del sabotatore alla decisione sotto incertezza . . 39

3 Modellazione dell'Incertezza **41**

3.1 Dalla Certezza delle Regole all'Incertezza delle Decisioni 41
3.1.1 Determinismo del dominio e incertezza comportamentale 41
3.1.2 Fonti di incertezza 42
3.1.3 Informazione osservabile e limiti dell'inferenza 43
3.2 Modelli dei Possibili Comportamenti Umani 43
3.2.1 Razionalità limitata e scelta rumorosa 43
3.2.2 Profili di competenza 44
3.2.3 Indicatori estratti dalle mosse 44
3.2.4 Interazione tra i due giocatori 45
3.3 Rappresentazione dell'Incertezza 45
3.3.1 Alberi di possibilità e attribuzione iniziale dei pesi 45
3.3.2 Trasformazione dei valori euristici in probabilità 47
3.3.3 Incertezza sulle probabilità 47
3.4 Valutazione degli Scenari e Gestione del Rischio 48
3.4.1 Valore atteso 48
3.4.2 Perdita nel caso peggiore e rischio di coda 49
3.4.3 Valutazione corretta per il rischio 49
3.4.4 Controllo della complessità 50
3.5 Profilazione Bayesiana dei Giocatori 51
3.5.1 Stato delle credenze 51
3.5.2 Aggiornamento mediante il teorema di Bayes 52
3.5.3 Costruzione della verosimiglianza 52
3.5.4 Stabilità, oblio e cambiamento di stile 53
3.6 Valutazione delle Mosse e Sintesi del Modello 53
3.6.1 Flusso decisionale proposto 53
3.6.2 Considerazioni conclusive 54

4 Risultati Sperimentali, Conclusioni e Sviluppi Futuri **57**

4.1 Risultati e Metriche Sperimentali 57

4.1.1 Analisi degli esiti per livello di difficoltà 57

4.1.2 Profilazione computazionale 58

4.1.3 Evoluzione dello stato delle 153 tuple 59

4.2 Evoluzione del Progetto e Scelte Progettuali 60

4.2.1 Prima versione basata su MaxN 60

4.2.2 Sperimentazione della pedina blu 60

4.2.3 Adozione del modello Paranoid 61

4.2.4 Ottimizzazione della versione conclusiva 61

4.3 Prospettive di Sviluppo Futuro e Conclusioni 62

4.3.1 Integrazione della Monte Carlo Tree Search 62

4.3.2 Reti Neurali e Apprendimento per Rinforzo 62

4.3.3 Ottimizzazione automatica dei pesi euristici 63

4.3.4 Conclusioni 63

Bibliografia **65**

Elenco delle figure

1.1 Effetto della gravità sulle mosse legali in una configurazione di Forza 4. 10
1.2 Struttura generale di un albero di gioco e dei suoi principali elementi. . 11
1.3 Esempio di propagazione dei valori nell'algoritmo Minimax. 13
1.4 Esempio di potatura Alpha-Beta. 17

2.1 Architettura ibrida del sistema. 27
2.2 Codifica della bitboard e scostamenti associati. 29
2.3 Rappresentazione schematica delle principali configurazioni. 34
2.4 Struttura dell'interfaccia grafica del progetto sviluppato. 38

3.1 Albero degli scenari con probabilità associate. 46
3.2 Rappresentazione dell'effetto del parametro λ sulla distribuzione softmax. 47
3.3 Esempio di aggiornamento bayesiano delle probabilità dei profili. 52

4.1 Distribuzione degli esiti e lunghezza media delle partite in funzione della difficoltà. 57
4.2 Progressione delle 153 tuple vincenti durante una partita a difficoltà Estremo. 59

Elenco delle tabelle

1.1 Confronto tassonomico tra i principali giochi da tavolo. 8

2.1 Pesi ponderali della funzione euristica basata su Threat-Space Analysis. 35
2.2 Configurazione dei quattro livelli di difficoltà dell'agente sabotatore. . . 37

3.1 Principali fonti di incertezza considerate. 42
3.2 Interpretazione dei profili comportamentali. 44
3.3 Relazione tra condizioni osservate, rischio e comportamento strategico. . 50

4.1 Metriche computazionali e profilazione dell'algoritmo decisionale. 58

Introduzione

I giochi da tavolo rappresentano da tempo un importante ambito di studio per l'Intelligenza Artificiale. La presenza di regole definite, azioni discrete e condizioni di vittoria chiaramente identificabili consente infatti di analizzare problemi di pianificazione, ricerca e scelta strategica all'interno di ambienti controllati. Anche nei giochi deterministici e a informazione perfetta, tuttavia, l'elevato numero di configurazioni possibili rende spesso impraticabile un'esplorazione completa dello spazio degli stati, richiedendo l'impiego di algoritmi di ricerca e funzioni euristiche.

Il presente lavoro affronta la progettazione di una variante di Forza 4 caratterizzata dalla presenza di tre partecipanti con obiettivi differenti. Due giocatori competono secondo le regole tradizionali, cercando di realizzare una sequenza di quattro pedine del proprio colore. Il terzo agente, denominato sabotatore, non possiede invece una propria condizione di vittoria: il suo compito consiste nell'impedire il successo di entrambi i giocatori e nel condurre la partita al pareggio. L'introduzione di questo ruolo asimmetrico modifica in modo significativo la struttura strategica del gioco. Il sabotatore deve valutare contemporaneamente le minacce costruite dai due avversari e scegliere mosse capaci di ostacolarle senza favorire involontariamente uno dei partecipanti. A questa difficoltà strategica si aggiunge la necessità di mantenere tempi di elaborazione compatibili con una partita interattiva.

L'obiettivo principale della tesi è quindi progettare e valutare un agente artificiale capace di prendere decisioni efficaci in tale contesto. La soluzione sviluppata combina tecniche di rappresentazione efficiente dello stato, valutazione euristica delle configurazioni e ricerca avversariale. Il sistema utilizza un'architettura ibrida nella quale la logica decisionale è implementata in SWI-Prolog, mentre Python e Pygame gestiscono l'interfaccia grafica e l'interazione con gli utenti.

Il processo decisionale si basa su una variante del Minimax adatta a un ambiente multi-agente, affiancata da tecniche di potatura e ottimizzazione dello spazio di ricerca. L'agente analizza la posizione corrente, individua le minacce più rilevanti e seleziona la mossa ritenuta più efficace per limitare le possibilità di vittoria degli altri giocatori. Una parte del lavoro considera inoltre il problema dell'incertezza relativa al comportamento umano, proponendo un possibile modello adattivo capace di tenere conto delle differenti strategie e competenze dei partecipanti.

L'efficacia del sistema viene esaminata attraverso una fase sperimentale nella quale sono confrontati diversi livelli di difficoltà e vengono osservati sia gli esiti delle partite sia le prestazioni computazionali dell'algoritmo. L'analisi permette di valutare la capacità difensiva del sabotatore e l'effetto delle principali scelte progettuali sul comportamento complessivo dell'agente.

La tesi è organizzata in quattro capitoli. Il primo introduce i principali riferimenti teorici relativi all'Intelligenza Artificiale applicata ai giochi, alla ricerca avversariale e ai sistemi multi-agente. Il secondo descrive le regole della variante, l'architettura del sistema e l'algoritmo sviluppato. Il terzo approfondisce il tema della decisione sotto incertezza e presenta una possibile estensione basata sulla modellazione del comportamento dei giocatori. Il quarto espone i risultati sperimentali, ricostruisce l'evoluzione del progetto e presenta le conclusioni e le possibili direzioni di sviluppo futuro.

Il sistema sviluppato è disponibile nella repository pubblica:
`https://github.com/zGlace/Forza-Quattro-Sabotatore`

1 Inquadramento Teorico

1.1 Intelligenza Artificiale e Giochi da Tavolo

1.1.1 I giochi da tavolo come base per l'Intelligenza Artificiale

Fin dalle origini dell'informatica, i giochi da tavolo hanno svolto un ruolo primario nello sviluppo, nella sperimentazione e nella verifica degli algoritmi di Intelligenza Artificiale. I giochi astratti a informazione perfetta offrono infatti un ambiente di prova ideale, essendo mondi discreti, deterministici e completamente regolati, in cui le condizioni di vittoria o sconfitta sono ben definite e lo stato della partita è interamente noto a tutti i partecipanti. Questa chiarezza strutturale ha permesso ai ricercatori di studiare i meccanismi fondamentali della decisione automatica, della pianificazione strategica e dell'apprendimento, senza dover gestire le incertezze sensoriali o l'incompletezza delle informazioni tipiche del mondo reale.

Da una prospettiva teorica, lo studio computazionale dei giochi ha reso evidente la differenza tra razionalità teorica e razionalità limitata (*bounded rationality*), introdotta dallo scienziato cognitivo ed economista Herbert Simon (1957). Da un punto di vista strettamente matematico, in base al teorema di Zermelo (1913), formalizzato ed esteso ai giochi in forma estesa da Kuhn (1953), ogni gioco finito a informazione perfetta ammette una strategia ottimale determinabile tramite induzione a ritroso. Tuttavia, nella pratica computazionale ci si scontra subito con il problema dell'esplosione combinatoria: il numero di configurazioni possibili cresce a un ritmo tale da superare qualsiasi capacità di memoria o di calcolo disponibile. Di conseguenza, la sfida principale dell'IA nei giochi consiste nel trovare mosse efficaci ed equilibrate in tempi ragionevoli, combinando l'esplorazione parziale dell'albero delle mosse con opportune funzioni di valutazione euristica.

1.1.2 Le origini della teoria computazionale

Il legame tra teoria dei giochi e calcolatori elettronici è stato tracciato da figure storiche fondamentali: John von Neumann, Claude Shannon e Alan Turing. Nel 1928, John von Neumann formulò il celebre Teorema del Minimax, approfondito poi nel libro *Theory of Games and Economic Behavior* (1944) scritto insieme a Oskar Morgenstern. Von Neumann dimostrò che, in qualsiasi gioco finito a due persone a somma zero e a

informazione perfetta, esiste una strategia ottimale che garantisce a ciascun giocatore il miglior risultato possibile nel caso peggiore. In questo scenario, un giocatore cerca di massimizzare il proprio guadagno minimo, mentre l'avversario cerca di minimizzare la perdita massima.

Il teorema di von Neumann forniva però una dimostrazione teorica dell'esistenza di una soluzione, senza indicare un algoritmo efficiente per calcolarla su un elaboratore reale. Fu Claude Shannon, nel celebre articolo *Programming a Computer for Playing Chess* (1950), a descrivere per primo la struttura concreta di un programma per giocare a scacchi. In tale lavoro, Shannon distinse due strategie di ricerca che mantengono tuttora piena validità didattica e applicativa.

Le strategie di Tipo A (basate sulla ricerca esaustiva o *brute-force*) esplorano sistematicamente tutti i nodi dell'albero di gioco fino a una certa profondità prefissata, valutando ogni mossa legale senza alcuna selezione preventiva per poi calcolare il valore della radice tramite Minimax. Shannon calcolò che un albero completo degli scacchi conterrebbe circa 10^{120} possibili sequenze di gioco (Numero di Shannon), dimostrando l'impossibilità pratica di una ricerca esaustiva e la necessità di arrestare l'esplorazione a una profondità finita affiancata da una funzione euristica.

Al contrario, le strategie di Tipo B (ricerca selettiva) cercano di imitare il ragionamento umano selezionando solo le mosse più rilevanti ed escludendo a priori i rami secondari o poco promettenti. L'algoritmo concentra il tempo di calcolo sulle linee tattiche cruciali, come le catture o le minacce immediate, anticipando concetti moderni quali la potatura euristica e la ricerca di quiete (*Quiescence Search*).

Negli stessi anni (1948–1950), Alan Turing progettò *Turochamp*, considerato il primo algoritmo pratico per il gioco degli scacchi. Non avendo a disposizione un elaboratore sufficientemente potente per eseguirlo, Turing ne simulò personalmente il funzionamento eseguendo i calcoli manualmente su carta. L'algoritmo impiegava oltre mezz'ora per ogni decisione, valutando il punteggio materiale dei pezzi, la loro mobilità e la sicurezza del re. Questo esperimento dimostrò che la logica decisionale poteva essere descritta come un algoritmo autonomo, indipendentemente dall'hardware su cui veniva eseguito.

1.1.3 L'evoluzione della ricerca avversariale

Negli anni successivi, i progressi dell'IA applicata ai giochi hanno segnato tappe determinanti per l'informatica. Nel 1959, Arthur Samuel sviluppò presso i laboratori IBM un programma per la dama che rappresentò uno dei primi esempi di apprendimento automatico applicato ai giochi. Il programma di Samuel aggiornava i coefficienti della propria funzione di valutazione tramite l'esperienza acquisita giocando contro sé stesso, riuscendo a competere con ottimi giocatori umani.

Nel 1975, la formalizzazione della potatura Alpha-Beta da parte di Donald Knuth e Ronald Moore, insieme all'uso di tabelle di trasposizione per memorizzare gli stati già visitati, ha consentito un incremento notevole della profondità di calcolo. Il culmine di questa evoluzione classica è stato raggiunto nel maggio 1997 con la vittoria del supercomputer *IBM Deep Blue* sul campione del mondo Garry Kasparov. *Deep Blue* combinava un'architettura di calcolo parallela con circuiti integrati dedicati, arrivando a esaminare fino a 200 milioni di posizioni al secondo e raggiungendo una profondità tipica di 12-14.

Accanto ai sistemi basati su euristiche e profondità limitata, la ricerca ha ottenuto risultati storici anche nella risoluzione esatta dei giochi. Nel 2007, il gruppo di ricerca di Jonathan Schaeffer annunciò la risoluzione completa del gioco della Dama inglese (8×8) con il programma *Chinook*. Grazie a un calcolo distribuito durato quasi due decenni e alla memorizzazione di database di finali con oltre $3{,}9 \times 10^{13}$ posizioni, Schaeffer dimostrò che con un gioco perfetto la partita termina sempre in parità.

Nel decennio più recente si è affermato un nuovo paradigma, introdotto da DeepMind con *AlphaGo* (Silver et al. 2016) e *AlphaZero* (2017). Invece di affidarsi a euristiche create manualmente dagli esperti, *AlphaZero* combina la ricerca su albero Monte Carlo (MCTS) con reti neurali profonde addestrate unicamente tramite *self-play* e apprendimento per rinforzo. Conoscendo soltanto le regole di base del gioco, *AlphaZero* ha raggiunto prestazioni superiori a quelle umane e ai motori tradizionali negli scacchi, nello shogi e nel Go, dimostrando la straordinaria potenza dell'apprendimento automatico moderno.

1.1.4 Forza 4 e la risoluzione di Allis

All'interno della famiglia dei giochi da tavolo astratti, il Forza 4 (noto in lingua inglese come *Connect 4*, ideato da Howard Wexler e Ned Strongin e pubblicato da Milton Bradley nel 1974) possiede caratteristiche scientifiche di grande rilievo. Rispetto a giochi come gli scacchi o il Go, che hanno spazi degli stati enormi, e rispetto al Tris, che è facilmente risolvibile anche a mano, il Forza 4 classico (su griglia di 6 righe per 7 colonne) si colloca a un livello intermedio molto interessante. È abbastanza complesso da richiedere algoritmi e strutture dati efficienti, ma sufficientemente contenuto da permettere uno studio matematico completo.

Nel 1988, il ricercatore olandese Victor Allis, nella sua tesi di laurea intitolata *A Knowledge-based Approach of Connect-Four*, annunciò la risoluzione computazionale completa del gioco classico, risultato raggiunto indipendentemente nello stesso periodo anche dall'informatico statunitense James Allen (1990).

Il teorema della risoluzione del Forza 4 formalizza che il gioco standard su griglia 6×7 a due giocatori è un gioco a informazione perfetta a somma zero il cui esito teorico con gioco perfetto corrisponde alla vittoria forzata del primo giocatore. In particolare,

il primo giocatore dispone di una strategia vincente deterministica se e solo se gioca la sua prima pedina nella colonna centrale (colonna 4), riuscendo a chiudere la partita al più tardi entro la quarantunesima mossa. Se invece il primo giocatore apre la partita in una delle due colonne adiacenti al centro, l'esito teorico con gioco perfetto da entrambi i giocatori è il pareggio; aprendo infine in una delle quattro colonne esterne, il secondo giocatore acquisisce il vantaggio e ha una vittoria forzata.

La soluzione di Victor Allis non si basava unicamente sulla forza bruta del calcolatore, ma introdusse un metodo innovativo: la *Threat-Space Analysis*. Allis realizzò il programma *VICTOR*, un sistema esperto basato su nove regole logiche in grado di identificare configurazioni di minaccia decisive senza dover esplorare l'albero fino agli stati terminali. Questo approccio logico-deduttivo rappresenta una delle basi teoriche di questo lavoro di tesi, in quanto i principi della *Threat-Space Analysis* sono stati tradotti nel paradigma dichiarativo di SWI-Prolog per realizzare la funzione di valutazione euristica dell'agente artificiale.

1.2 Tassonomia dei Giochi e il Modello del Forza 4 Classico

1.2.1 Dimensioni tassonomiche secondo la teoria dei giochi

Nella letteratura sull'Intelligenza Artificiale e sulla Teoria dei Giochi, un gioco viene descritto e classificato in base a una serie di proprietà formali che ne definiscono l'ambiente informativo, le dinamiche di transizione e gli obiettivi dei partecipanti. Le dimensioni tassonomiche principali sono:

- **Deterministico vs. Stocastico:** In un ambiente deterministico, ogni azione intrapresa da un giocatore produce un unico stato successivo ben definito, senza alcuna componente casuale (come dadi o mazzi di carte).
- **Informazione Perfetta vs. Imperfetta:** Un gioco è a informazione perfetta se tutti i giocatori hanno visibilità completa dello stato corrente della partita e della cronologia delle mosse giocate, senza carte coperte o informazioni riservate.
- **A Somma Zero vs. A Somma Non-Zero:** Nei giochi a somma zero, la somma delle utilità assegnate a tutti i partecipanti al termine della partita è costante (generalmente posta pari a zero). Di conseguenza, il guadagno di un giocatore coincide esattamente con la perdita dell'avversario. Nei giochi a somma non-zero, invece, gli interessi possono essere parzialmente coincidenti o misti, aprendo la strada a forme di cooperazione, stallo o penalizzazione congiunta.
- **Sequenziale vs. Simultaneo:** Nei giochi sequenziali, i giocatori muovono uno dopo l'altro potendo osservare la mossa appena effettuata dal predecessore.

Nei giochi simultanei, tutti i giocatori prendono la propria decisione nello stesso momento, senza conoscere in anticipo la scelta altrui.

- **Finito vs. Infinito:** Un gioco si definisce finito quando ha un numero finito di stati possibili, un numero finito di mosse legali a ogni turno e una durata limitata a un numero massimo di mosse.

La Tabella 1.1 riassume queste caratteristiche, confrontando il Forza 4 classico con gli altri giochi da tavolo maggiormente studiati nella letteratura scientifica.

Gioco	Numero Giocatori	Natura Transizioni	Tipo Informazioni	Dinamica Turni	Spazio Stati ($\log_{10}$)
Tris	2	Deterministico	Perfetta	Alternata (1-1)	$\approx 3{,}7$ (5 478)
Forza 4 (6×7)	2	Deterministico	Perfetta	Alternata (1-1)	$\approx 12{,}6$ ($4{,}53 \times 10^{12}$)
Scacchi	2	Deterministico	Perfetta	Alternata (1-1)	≈ 47 (10^{47})
Dama Inglese (8×8)	2	Deterministico	Perfetta	Alternata (1-1)	$\approx 20{,}7$ ($5{,}00 \times 10^{20}$)
Go (19×19)	2	Deterministico	Perfetta	Alternata (1-1)	≈ 170 (10^{170})
Battaglia Navale	2	Deterministico	Imperfetta	Alternata (1-1)	≈ 12 (10^{12})
Morra Cinese	2	Deterministico	Imperfetta	Simultanea	$\approx 0{,}95$ (9)
Backgammon	2	Stocastico (Dadi)	Perfetta	Alternata (1-1)	≈ 20 (10^{20})
Poker	2-10	Stocastico (Carte)	Imperfetta	Sequenziale/Blinds	Dipende dalle varianti

Tabella 1.1. Confronto tassonomico tra i principali giochi da tavolo.

1.2.2 Formalizzazione matematica di un gioco a due giocatori

Formalmente, un gioco a due giocatori deterministico a somma zero e a informazione perfetta viene descritto da una tupla a sette componenti:

$$\Gamma = \langle S, s_0, P, A, T, \text{Terminal}, U \rangle \tag{1.1}$$

I simboli della tupla hanno il seguente significato:

- $\boldsymbol{S}$: Lo spazio degli stati è l'insieme di tutte le configurazioni possibili della griglia e delle relative informazioni di stato.
- $\boldsymbol{s_0 \in S}$: Lo stato iniziale della partita che corrisponde alla griglia completamente vuota.
- $\boldsymbol{P: S \to \{1,2\}}$: La funzione del giocatore associa a ciascuno stato non terminale l'identificativo del giocatore a cui spetta il turno di gioco.
- $\boldsymbol{A: S \to \mathcal{P}}(\text{Actions})$: La funzione delle azioni legali definisce l'insieme delle mosse lecite ammissibili nello stato s, indicato con $A(s)$.
- $\boldsymbol{T: S \times} \text{Actions} \boldsymbol{\to S}$: La funzione di transizione associa a uno stato corrente s e a un'azione valida a il conseguente stato successivo: $s' = T(s, a)$.

- Terminal$\boldsymbol{:\ S \rightarrow \{}$true,false$\boldsymbol{\}}$: La funzione booleana stabilisce se la partita è terminata.
- $\boldsymbol{U:\ \{s \in S}$ | Terminal$\boldsymbol{(s)}$ = true$\boldsymbol{\} \times \{1,2\} \rightarrow} \mathbb{R}$: La funzione di utilità assegna un valore numerico a ciascun giocatore negli stati finali. Trattandosi di un gioco a somma zero a due giocatori, vale la relazione di complementarità opposta:

$$U(s,1) + U(s,2) = 0 \implies U(s,2) = -U(s,1), \quad \forall s \in S_{\text{term}} \tag{1.2}$$

Nei giochi con esiti discreti semplici (vittoria, sconfitta, pareggio), i punteggi di utilità per i giocatori vengono generalmente normalizzati nell'insieme $\{-1, 0, +1\}$, corrispondenti rispettivamente a sconfitta, patta e vittoria.

1.2.3 Vincoli cinematici e fisici di Forza 4

Rispetto ad altri giochi di allineamento su griglia (come il Tris o il Gomoku), la particolarità di Forza 4 consiste nel vincolo fisico della gravità. Si consideri una griglia formata da R righe (numerate dal basso verso l'alto come $r \in \{0, 1, \ldots, R-1\}$) e da C colonne (numerate da sinistra a destra come $c \in \{0, 1, \ldots, C-1\}$). Ogni singola casella (r, c) può assumere uno di tre valori:

$$\text{cell}(r,c) \in \{\text{vuoto (0), Giocatore 1 (1), Giocatore 2 (2)}\} \tag{1.3}$$

La caduta per gravità impone che nessuna pedina possa trovarsi sospesa nel vuoto. Matematicamente, una casella alla riga $r > 0$ può essere occupata solo se la casella sottostante nella medesima colonna c è già occupata da una pedina, a prescindere dal colore. Formalmente, la griglia soddisfa la seguente proprietà di monotonicità verticale:

$$\text{cell}(r,c) \neq 0 \implies \text{cell}(r',c) \neq 0, \quad \forall r' < r \tag{1.4}$$

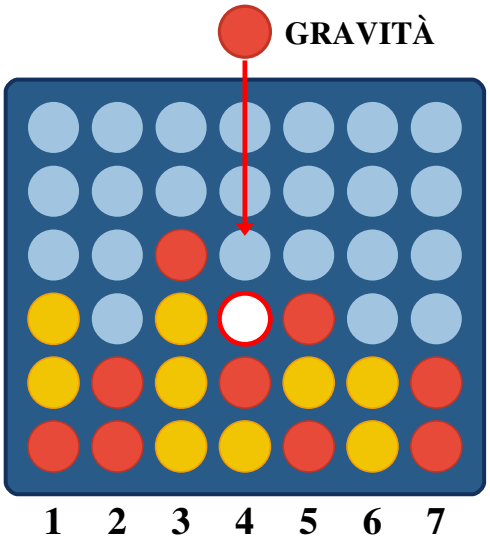


Figura 1.1. Effetto della gravità sulle mosse legali in una configurazione di Forza 4.

Questo vincolo strutturale comporta due importanti conseguenze per la ricerca algoritmica.

In primo luogo, determina una drastica riduzione del fattore di ramificazione (*branching factor*): mentre in un generico gioco su griglia privo di gravità con k celle occupate le mosse possibili sarebbero pari a $(R \times C) - k$, nel Forza 4 una mossa legale consiste unicamente nella scelta di una colonna non ancora piena. Pertanto, il numero di mosse disponibili a ogni turno è superiormente limitato dal numero di colonne della griglia: $b(s) \leq C$, indipendentemente dal numero totale di caselle libere.

In secondo luogo, la caduta per gravità genera una stretta interdipendenza verticale tra le mosse. Poiché l'inserimento di una pedina nella casella (r, c) rende giocabile la casella sovrastante $(r + 1, c)$ al turno successivo, i giocatori devono costantemente valutare se una propria mossa non rischi di fungere da supporto per l'avversario. Questa interazione verticale rappresenta l'aspetto più ricco e delicato dell'analisi tattica delle minacce.

1.3 Spazio degli Stati, Albero di Gioco e Complessità di Ricerca

1.3.1 Lo spazio degli stati e l'albero di gioco

Nello studio algoritmico di un gioco è fondamentale distinguere tra la totalità delle configurazioni teoriche della griglia e l'insieme delle posizioni effettivamente raggiungibili durante una partita. Uno stato di gioco, indicato con $s \in S$, include sia la disposizione spaziale delle pedine sulla griglia, sia le informazioni sul turno corrente e sull'eventuale fase attiva.

Si distinguono formalmente due insiemi di configurazioni:

- **Spazio degli Stati Teorico (S_{teorico}):** Rappresenta l'insieme puramente matematico ottenuto assegnando a ogni casella uno dei tre valori possibili (vuoto, rosso, giallo), senza considerare le regole di caduta, la raggiungibilità dallo stato iniziale o la parità tra le mosse. La cardinalità teorica è pari a $|S_{\text{teorico}}| = 3^{R \times C}$.
- **Spazio degli Stati Legali ($S_{\text{legali}} \subset S_{\text{teorico}}$):** È il sottoinsieme delle configurazioni a cui si può arrivare a partire dalla griglia vuota s_0 tramite una sequenza valida di mosse legali. Questo insieme esclude le configurazioni fisicamente impossibili e gli stati che violano le regole del gioco.

L'albero di gioco (*Game Tree*) è la struttura ad albero orientato che descrive graficamente tutte le possibili evoluzioni della partita. In tale albero si ha:

- **Il nodo radice:** Rappresenta lo stato iniziale s_0 o lo stato corrente della griglia da cui l'algoritmo avvia la ricerca.
- **I nodi interni:** Corrispondono agli stati intermedi della partita. A ogni livello dell'albero corrisponde una semimossa (*ply*) effettuata dal giocatore di turno.
- **Gli archi orientati:** Rappresentano le azioni legali che portano da uno stato genitore s a uno stato successore $s' = T(s, a)$.
- **I nodi foglia:** Corrispondono a stati terminali in cui la partita finisce, oppure a stati intermedi in cui la ricerca si interrompe per raggiunti limiti di profondità, valutati tramite una funzione euristica $h(s)$.

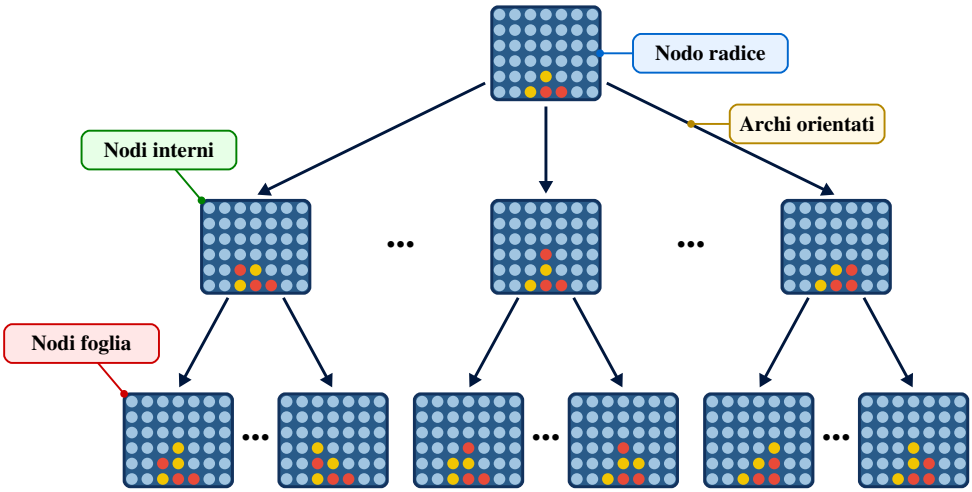


Figura 1.2. Struttura generale di un albero di gioco e dei suoi principali elementi.

1.3.2 Branching factor, profondità ed esplosione combinatoria

La difficoltà computazionale dell'esplorazione dell'albero di gioco è determinata principalmente da due fattori: il branching factor e la profondità dell'albero.

Il branching factor istantaneo $b(s)$ indica il numero di mosse legali disponibili nello stato s. Poiché tale numero diminuisce quando alcune colonne si riempiono, si definisce il branching factor medio come la media delle scelte disponibili lungo le partite.

La profondità d misura la lunghezza della sequenza di semimosse tra la radice e un determinato nodo dell'albero.

In un albero con branching factor costante b e profondità d, il numero di nodi al livello d e il numero complessivo di nodi generati dall'intero albero sono espressi dalle formule:

$$N(d) = b^d, \qquad N_{\text{tot}}(d) = \sum_{k=0}^{d} b^k = \frac{b^{d+1} - 1}{b - 1} = \mathcal{O}(b^d) \tag{1.5}$$

Questa relazione illustra il problema dell'esplosione combinatoria: ogni semimossa aggiuntiva moltiplica il numero di nodi da esaminare per un fattore pari a b. Ne consegue che una ricerca esaustiva completa diventa impraticabile già per valori modesti di profondità.

1.4 L'Algoritmo Minimax

1.4.1 Principio di induzione a ritroso e razionalità perfetta

L'algoritmo Minimax costituisce il modello fondamentale per la ricerca decisionale nei giochi a due giocatori deterministici e a somma zero. Il suo funzionamento si basa sul concetto di induzione a ritroso (*Backward Induction*) e sull'ipotesi che entrambi i giocatori scelgano sempre la mossa migliore a loro disposizione (razionalità perfetta).

I due partecipanti vengono tradizionalmente identificati come giocatore MAX e giocatore MIN. Il giocatore MAX è colui che muove per primo (o dal cui punto di vista si analizza la partita) e mira a massimizzare il valore del risultato finale. Il giocatore MIN rappresenta l'avversario: trattandosi di un gioco a somma zero, il massimo guadagno per MAX coincide con la massima perdita per MIN, per cui MIN sceglierà a ogni turno l'azione che rende minimo il punteggio conseguibile da MAX.

1.4.2 Formulazione ricorsiva e funzione di valore

Dato il gioco formale Γ definito nella Sezione 1.2.2, si definisce la funzione valore Minimax, indicata con $V(s)\colon S \to \mathbb{R}$, che associa a ogni stato s il risultato ottenibile da MAX a partire da s nell'ipotesi di gioco ottimale da entrambe le parti. La funzione è

definita in modo ricorsivo:

$$V(s) = \begin{cases} U(s), & \text{se Terminal}(s) = \text{true} \\ \max_{a \in A(s)} V(T(s,a)), & \text{se } P(s) = \text{MAX} \\ \min_{a \in A(s)} V(T(s,a)), & \text{se } P(s) = \text{MIN} \end{cases} \tag{1.6}$$

L'equazione (1.6) riassume il principio dell'induzione a ritroso, nel quale il valore di uno stato intermedio viene determinato propagando i valori dai nodi foglia verso la radice (approccio *bottom-up*). Nei nodi in cui tocca a MAX, il valore del genitore è il massimo tra i valori dei figli; nei nodi in cui tocca a MIN, il valore del genitore è invece il minimo.

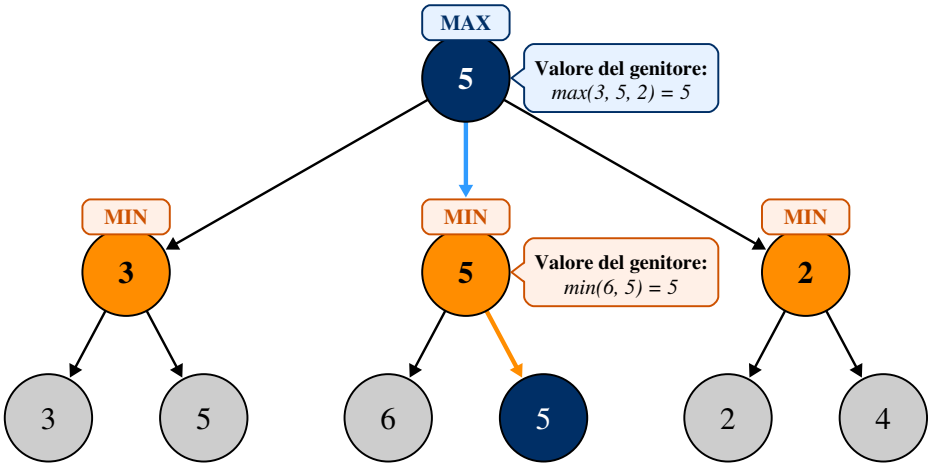


Figura 1.3. Esempio di propagazione dei valori nell'algoritmo Minimax.

1.4.3 Funzionamento procedurale dell'algoritmo

Dal punto di vista pratico, l'algoritmo Minimax viene implementato tramite una funzione ricorsiva basata su visita in profondità. A ogni chiamata sullo stato corrente, l'algoritmo verifica innanzitutto le condizioni di arresto: se la partita è terminata o se la profondità residua si è azzerata, la ricorsione si interrompe restituendo il valore esatto dello stato terminale $U(s)$ o la stima fornita dalla funzione euristica $h(s)$.

Negli stati intermedi, il valore migliore provvisorio viene inizializzato a $-\infty$ per i nodi MAX e a $+\infty$ per i nodi MIN. La funzione genera quindi l'elenco delle mosse legali $A(s)$ e, per ciascuna di esse, richiama ricorsivamente sé stessa riducendo la profondità residua di un'unità e alternando il ruolo del giocatore. I valori restituiti dai figli vengono aggregati aggiornando la stima del genitore tramite l'operatore di massimo o di minimo a seconda del livello, fino a restituire al nodo radice il punteggio ottimale e la mossa migliore da eseguire sulla griglia.

1.4.4 Proprietà teoriche e complessità computazionale

L'algoritmo Minimax soddisfa proprietà teoriche fondamentali di completezza e ottimalità. Sotto il profilo della completezza, se l'albero di gioco è finito, Minimax trova sempre la soluzione esplorando tutti i possibili esiti. Riguardo all'ottimalità, l'algoritmo assicura la scelta migliore contro un avversario razionale perfetto; se l'avversario commette errori o gioca mosse subottimali, Minimax garantisce comunque un risultato almeno pari a quello teorico di equilibrio.

Dal punto di vista computazionale, poiché l'algoritmo visita ogni nodo dell'albero fino alla profondità d con branching factor b, la complessità temporale asintotica cresce in modo esponenziale:

$$T_{\text{Minimax}}(b, d) = \mathcal{O}(b^d) \tag{1.7}$$

Per quanto riguarda la complessità spaziale, implementando la visita mediante ricerca in profondità (*Depth-First Search, DFS*) con backtracking, è sufficiente mantenere in memoria soltanto il percorso corrente dalla radice al nodo esaminato insieme ai nodi fratelli ancora da valutare. La complessità spaziale risulta pertanto lineare rispetto alla profondità di ricerca:

$$S_{\text{Minimax}}(b, d) = \mathcal{O}(b \cdot d) \tag{1.8}$$

Il consumo ridotto di memoria rende la ricerca in profondità la tecnica standard per i giochi su griglia. Anche a profondità elevate, l'occupazione di memoria primaria rimane modesta, concentrando la sfida computazionale sul tempo di calcolo.

1.4.5 Il limite dell'orizzonte e la valutazione euristica

Nei giochi da tavolo più complessi, la rapida crescita esponenziale dell'albero di ricerca rende impraticabile l'applicazione dell'algoritmo Minimax fino agli stati terminali della partita. Per consentire all'agente di selezionare una mossa entro limiti di tempo ragionevoli, è quindi necessario arrestare l'esplorazione a una profondità massima prefissata, indicata convenzionalmente con d_{cut}.

Quando la ricerca raggiunge la profondità d_{cut}, il valore dello stato viene stimato tramite una funzione euristica statica $h(s)$:

$$h(s) \approx V(s), \qquad h \colon S \to [-M, +M] \tag{1.9}$$

La funzione $h(s)$ valuta la qualità della posizione assegnando un punteggio positivo alle configurazioni favorevoli e penalizzando quelle pericolose. Tuttavia, l'interruzione

della ricerca a una profondità fissa può causare un problema noto come effetto orizzonte (*Horizon Effect*).

Questo si verifica quando un evento decisivo (come una vittoria forzata dell'avversario) si colloca appena oltre il limite massimo di profondità d_{cut} impostato per la ricerca. Non potendo vedere cosa accade oltre l'orizzonte, l'algoritmo può scegliere mosse che ritardano momentaneamente la minaccia di una singola semimossa (ad esempio sacrifici o blocchi poco efficaci), con l'illusione di averla evitata, mentre in realtà peggiora la propria posizione complessiva. Per mitigare questo effetto, i motori decisionali moderni non interrompono la ricerca in posizioni precarie (come scacchiere con minacce immediate di vittoria), ma prolungano selettivamente l'esplorazione fino a raggiungere stati stabili, in cui la funzione euristica statica risulta davvero attendibile.

1.5 La Potatura Alpha-Beta

1.5.1 Principi matematici della potatura e conservazione dell'ottimalità

L'algoritmo Minimax, pur essendo formalmente corretto, presenta lo svantaggio pratico di dover visitare tutti i nodi dell'albero di ricerca fino alla profondità prestabilita. La potatura Alpha-Beta (*Alpha-Beta Pruning*), analizzata in modo sistematico da Donald Knuth e Ronald W. Moore nel 1975, costituisce la più importante tecnica di ottimizzazione esatta per la ricerca avversariale.

Il principio alla base dell'Alpha-Beta Pruning è intuitivo e rigoroso: durante l'esplorazione dell'albero, è possibile tralasciare interi sotto-alberi qualora si possa dimostrare matematicamente che le posizioni in essi contenute non potranno in alcun caso influenzare la scelta finale alla radice. Questa è la proprietà cardine dell'algoritmo, definita come perfetta conservazione dell'ottimalità della soluzione.

L'algoritmo opera mantenendo e aggiornando durante la discesa ricorsiva due valori scalari, tradizionalmente indicati con le lettere greche α (Alpha) e β (Beta). Il parametro α rappresenta il limite inferiore del punteggio che il giocatore MAX ha già la certezza matematica di poter conseguire lungo un percorso precedentemente esplorato; inizializzato a $-\infty$ alla radice, α può solo crescere monotonamente nei nodi MAX. Dualmente, il parametro β rappresenta il limite superiore del punteggio che il giocatore MIN è disposto a concedere a MAX, avendo già individuato una contromossa migliore per la propria difesa; inizializzato a $+\infty$ alla radice, β può solo decrescere nei nodi MIN.

1.5.2 Meccanismo dei Cutoff

La potatura avviene non appena i due valori della finestra di ricerca si incrociano, ovvero quando si verifica la condizione:

$$\alpha \geq \beta \implies \text{Cutoff (interruzione immediata della visita dei rami fratelli)} \tag{1.10}$$

Si distinguono due tipi di taglio lungo l'albero di ricerca:

- **Beta-Cutoff (nodi MIN):** Si consideri un nodo n di tipo MIN avente come genitore un nodo MAX il cui valore garantito è almeno α. Se durante l'esplorazione dei nodi figli di n uno di essi restituisce un valore $v \leq \alpha$, il valore definitivo del nodo n non potrà mai superare v (essendo n un nodo di minimizzazione): $V(n) \leq v \leq \alpha$. Poiché il genitore MAX dispone già di una scelta che gli assicura almeno α, MAX non sceglierà mai la mossa che porta a questo nodo n. L'esplorazione dei rimanenti figli di n è quindi superflua e viene interrotta immediatamente.
- **Alpha-Cutoff (nodi MAX):** In modo speculare, si consideri un nodo n di tipo MAX avente come antenato un nodo MIN il cui limite massimo accettabile è β. Se un figlio di n restituisce un valore $v \geq \beta$, il valore finale del nodo MAX sarà almeno $V(n) \geq v \geq \beta$. L'antenato MIN, potendo limitare il punteggio a β tramite un'altra mossa già esaminata, eviterà di consentire la transizione verso il nodo n. L'espansione dei restanti rami viene quindi bloccata all'istante.

1.5.3 Procedura di esecuzione e gestione della finestra $[\alpha, \beta]$

Dal punto di vista implementativo, l'algoritmo Alpha-Beta arricchisce il flusso ricorsivo di Minimax gestendo dinamicamente la finestra di ricerca. Alla radice la ricerca viene inizializzata con la finestra massimamente aperta ($\alpha = -\infty$ e $\beta = +\infty$). A ogni nodo intermedio, le mosse lecite vengono ordinate secondo criteri euristici, poiché valutare prima le opzioni più promettenti massimizza la probabilità di innescare tagli precoci. Durante la discesa ricorsiva, i nodi MAX e MIN aggiornano rispettivamente i valori di α e β in base ai ritorni dei sotto-alberi, interrompendo immediatamente la visita dei nodi fratelli non appena si manifesta la condizione di cutoff $\alpha \geq \beta$.

La Figura 1.4 schematizza un caso di Beta-Cutoff e la conseguente eliminazione dei rami superflui.

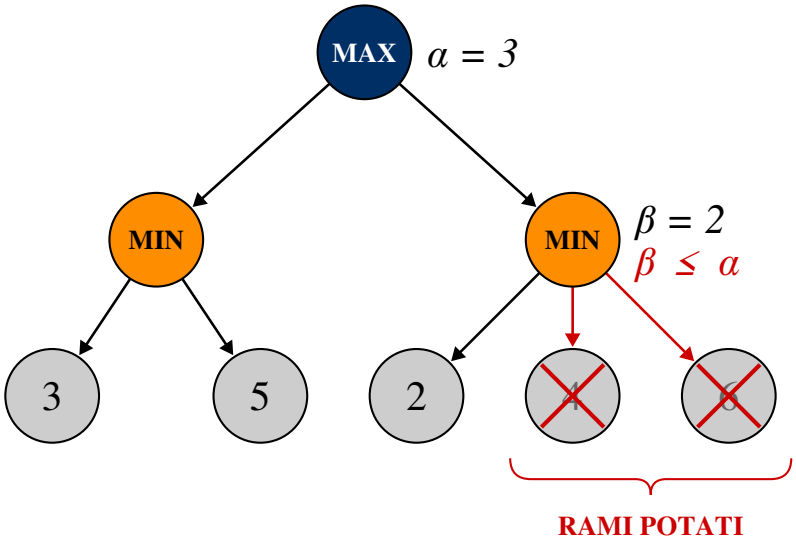


Figura 1.4. Esempio di potatura Alpha-Beta.

1.5.4 Analisi della complessità computazionale

L'efficacia della potatura dipende fortemente dall'ordine con cui vengono esaminate le mosse figlie a ogni livello (*Move Ordering*):

- **Caso Peggiore:** Se le mosse vengono esaminate nell'ordine peggiore (dalla meno efficace alla migliore), la condizione $\alpha \geq \beta$ non si verifica mai in anticipo. Di conseguenza, nessun ramo viene potato e l'algoritmo esplora l'intero albero di gioco: $T_{\text{worst}}(b, d) = \mathcal{O}(b^d)$, con prestazioni identiche a quelle del Minimax semplice.
- **Caso Medio:** Ipotizzando un ordinamento casuale delle mosse a ciascun nodo, la complessità asintotica stimata si attesta su:

$$T_{\text{avg}}(b, d) \approx \mathcal{O}\left(\left(\frac{b}{\ln b}\right)^d\right) \tag{1.11}$$

- **Caso Ottimo:** Se a ogni nodo la mossa migliore viene valutata per prima, il numero di foglie da esaminare si riduce al minimo teorico:

$$N_{\text{opt}}(b, d) = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \approx 2 \cdot b^{d/2} = \mathcal{O}(b^{d/2}) \tag{1.12}$$

L'equazione (1.12) evidenzia il grande vantaggio di Alpha-Beta, ossia quello di dimezzare l'esponente asintotico da d a $d/2$. A parità di nodi elaborati e di tempo di calcolo, l'algoritmo può spingersi al doppio della profondità di previsione.

1.5.5 L'importanza delle tecniche di Move Ordering

L'efficacia reale di Alpha-Beta dipende in larga misura dall'ordine in cui vengono analizzate le mosse. Per avvicinarsi il più possibile al caso ottimo $\mathcal{O}(b^{d/2})$, vengono impiegate diverse tecniche di Move Ordering. In Forza 4 assume un ruolo di primo piano l'ordinamento statico posizionale proposto da John Tromp: le colonne centrali offrono un numero di opportunità di allineamento nettamente superiore rispetto a quelle periferiche, poiché una pedina al centro partecipa a più combinazioni rispetto a quelle laterali.

A questo criterio geometrico si affianca l'impiego della Tabella delle Trasposizioni: memorizzando le posizioni già esaminate in una struttura hash, il motore può estrarre la mossa migliore individuata nelle iterazioni precedenti o a profondità inferiori (*Principal Variation Move*) e valutarla come prima opzione al nodo corrente, massimizzando la probabilità di cutoff immediati.

1.6 Dai Giochi a Somma Zero ai Sistemi Multi-Agente Asimmetrici

1.6.1 Transizione ai sistemi multi-agente

Il modello classico basato su Minimax e Alpha-Beta presuppone che il gioco sia a due giocatori e a somma zero. Estendendo il modello a tre o più agenti, questa assunzione decade e ciascun partecipante possiede una propria funzione di utilità individuale $U_i \colon S_{\text{term}} \to \mathbb{R}$. Il gioco cessa di essere a somma costante e assume una natura a motivazioni miste (*mixed-motive*): un'azione che penalizza un avversario non garantisce un beneficio diretto a chi la compie, ma può avvantaggiare un terzo contendente. Viene così a mancare il principio cardine del Minimax secondo cui la scelta peggiore per l'avversario coincide con la migliore per noi e l'utilità degli stati assume una forma vettoriale:

$$\sum_{i=1}^{n} U_i(s) \neq \text{costante}, \qquad \vec{U}(s) = (U_1(s), U_2(s), \ldots, U_n(s)) \tag{1.13}$$

Di conseguenza, lo stato del gioco non può più essere sintetizzato da una singola funzione di valutazione scalare. Per estendere la logica decisionale oltre lo schema bipolare si adottano paradigmi di ricerca multi-agente, tra i quali l'algoritmo MaxN e il modello Paranoid.

1.6.2 L'algoritmo MaxN e le difficoltà di potatura

La generalizzazione più nota del Minimax per giochi a n partecipanti è l'algoritmo MaxN, proposto da C. A. Luckhardt e K. B. Irani nel 1986. In MaxN, il valore di ogni nodo non è un singolo numero scalare, ma un vettore a n componenti:

$$\vec{V}(s) = (u_1(s), u_2(s), \ldots, u_n(s)) \in \mathbb{R}^n \tag{1.14}$$

Quando la ricerca raggiunge un nodo in cui tocca al giocatore i, l'algoritmo esamina i vettori restituiti dai nodi figli e sceglie la mossa che rende massima la componente i-esima del vettore:

$$\text{best_child} = \underset{\text{child}}{\text{argmax}} \left(\vec{V}(\text{child})_i \right) \tag{1.15}$$

Sul piano teorico, MaxN rappresenta una formulazione rigorosa. Con orizzonte completo, garantisce la convergenza verso un punto di equilibrio. Tuttavia, come evidenziato da Luckhardt e Irani, l'algoritmo manifesta un grave collo di bottiglia computazionale dovuto alla drastica perdita di efficacia dei meccanismi di potatura. Nei giochi a somma non costante, infatti, i punteggi dei vari partecipanti non sono legati da una relazione algebrica rigida e speculare; di conseguenza, il vantaggio conseguito da un agente lungo un percorso non implica una perdita proporzionale per gli altri, precludendo l'impiego della potatura tradizionale Alpha-Beta.

Sebbene la letteratura abbia proposto estensioni dei tagli per MaxN (in particolare lo *shallow pruning* formulato da Korf nel 1991), tali condizioni di taglio si innescano raramente nella pratica. L'algoritmo riesce a potare solo una frazione trascurabile di nodi, rimanendo vincolato a una complessità temporale pressoché identica al caso peggiore non potato, pari a $\mathcal{O}(b^d)$. In giochi con un branching factor significativo non banale ($b \geq 7$), questa inefficienza limita la profondità raggiungibile a sole 3 o 4 semimosse entro tempi di calcolo accettabili, rendendo l'algoritmo inadeguato per garantire decisioni tatticamente solide in tempo reale.

1.6.3 L'Approccio Paranoid

Per superare i limiti di MaxN, la letteratura sull'IA nei giochi multi-agente offre una soluzione molto efficiente: l'approccio Paranoid (Sturtevant & Korf, 2000). Questa tecnica riconduce un gioco a n partecipanti a una sfida a due entità attraverso un'ipotesi cautelativa pessimistica: l'agente che pianifica la mossa assume che tutti gli altri giocatori formino una coalizione implicita orientata unicamente a minimizzare la sua utilità, ignorando qualsiasi rivalità reciproca.

Sul piano matematico, l'approccio proietta la funzione di utilità vettoriale $U(s) \in \mathbb{R}^n$ su una funzione scalare $U_i(s) \in \mathbb{R}$ calcolata dal punto di vista dell'agente i. Si definisce quindi la funzione valore ricorsiva:

$$V_{\text{paranoid}}(s) = \begin{cases} U_i(s), & \text{se Terminal}(s) = \text{true} \\ \max_{a \in A(s)} V_{\text{paranoid}}(T(s,a)), & \text{se } P(s) = i \\ \min_{a \in A(s)} V_{\text{paranoid}}(T(s,a)), & \text{se } P(s) \neq i \end{cases} \tag{1.16}$$

In questo modo, la partita a tre agenti viene semplificata ad uno scontro bipolare: l'agente i assume il ruolo MAX, mentre la totalità degli altri contendenti viene aggregata in un'unica coalizione fittizia MIN avversaria. Questa riduzione scalare offre due vantaggi metodologici e operativi.

In primo luogo, ripristina la piena applicabilità della potatura Alpha-Beta. Considerando l'utilità della coalizione MIN come l'esatto opposto di quella di MAX, il gioco viene ricondotto al modello a somma zero. È quindi possibile riutilizzare la finestra di ricerca $[\alpha, \beta]$ e il meccanismo di potatura descritto nelle Sezioni 1.5.2 e 1.5.3.

In secondo luogo, l'approccio garantisce una solida robustezza nel caso peggiore. Pianificare contro una coalizione avversa rende l'agente prudente, poiché anche qualora i giocatori umani commettano errori, agiscano in modo disgiunto o adottino strategie subottimali, l'agente si è già coperto contro la combinazione di mosse a lui più sfavorevole.

1.6.4 Introduzione al Progetto

Il percorso teorico sviluppato in questo capitolo ha messo in luce una netta distinzione tra i giochi classici a somma zero e gli scenari multi-agente. Nei contesti tradizionali a due partecipanti, come il Forza 4 classico, l'esistenza di un'opposizione diametrale tra i giocatori e la presenza di funzioni di utilità speculari consentono di fare affidamento su garanzie matematiche precise e su tecniche di ricerca consolidate. Al contrario, quando il numero di agenti cresce o gli scopi dei partecipanti non risultano perfettamente simmetrici, i presupposti di ottimalità lineare vengono meno, rendendo l'esplorazione dello spazio degli stati una sfida computazionalmente più complessa.

È proprio in questo spazio concettuale che si colloca la motivazione del presente lavoro di tesi. Piuttosto che riproporre una classica sfida binaria, l'obiettivo del progetto è indagare le dinamiche emergenti dall'inserimento di un terzo agente con un ruolo volutamente asimmetrico: il *Sabotatore*. Invece di competere per allineare le proprie pedine e vincere la partita in modo convenzionale, questo agente persegue una funzione di disturbo continuo, orientata a neutralizzare le iniziative avversarie e a spingere l'esito del confronto verso un pareggio.

Una simile architettura di gioco impone un ripensamento critico degli strumenti decisionali visti finora. L'agente non può fare affidamento su assunzioni semplicistiche sul comportamento degli altri giocatori, né può permettersi un calcolo esaustivo incompatibile con i vincoli di una partita in tempo reale. Prima di poter tradurre queste considerazioni in specifici algoritmi di ricerca euristica e nelle strutture logiche destinate a guidare l'agente, diviene essenziale definire l'ambiente formale in cui si realizza tale interazione.

2 Architettura e Sviluppo del Sistema

2.1 Descrizione della Variante di Forza 4: Regole, Asimmetria e Meccaniche di Gioco

2.1.1 Geometria e dimensionamento della griglia estesa

La variante di Forza 4 sviluppata in questo lavoro di tesi introduce una modifica significativa alle dimensioni della griglia rispetto al modello classico standard da 6 righe e 7 colonne. La presente implementazione adotta uno schema 8×9, per una superficie complessiva di 72 caselle giocabili.

Questo ampliamento dimensionale (+71,4% di caselle disponibili) è stato scelto per rendere tatticamente equilibrata l'interazione a tre agenti. In una griglia classica 6×7 a due giocatori, lo spazio si riduce rapidamente. Entro le prime 15–20 mosse le colonne centrali tendono a riempirsi, limitando le possibilità di manovra e riducendo il tempo a disposizione per costruire allineamenti articolati. L'inserimento di un terzo agente avrebbe provocato sulla griglia standard una rapida saturazione del gioco, riducendo le partite a semplici sequenze di blocchi difensivi.

L'estensione adottata risponde a precise esigenze di progettazione del gioco. In primo luogo, l'aggiunta di due colonne rispetto alla griglia originale crea una fascia centrale a tre vie (colonne 4, 5 e 6) anziché un singolo perno centrale. Questo assetto moltiplica le combinazioni geometriche possibili ed evita che un singolo blocco sull'asse mediano paralizzi l'iniziativa dei giocatori. In secondo luogo, la disponibilità di 72 caselle complessive rende possibile l'estensione della partita fino a 18 cicli completi a quattro fasi (dettagliati nella Sezione 2.1.3). Ciò offre ai giocatori lo spazio necessario per sviluppare strategie complesse.

2.1.2 Ruoli e asimmetria dei tre agenti

La caratteristica distintiva del progetto riguarda la definizione dei ruoli dei tre partecipanti, che presentano obiettivi e vincoli operativi differenti:

- **Giocatore 1:** Il primo giocatore umano con il colore rosso. Il suo obiettivo è allineare quattro pedine del proprio colore in senso orizzontale, verticale o diagonale per vincere la partita.

- **Giocatore 2:** Il secondo giocatore umano con il colore giallo. Persegue specularmente lo stesso scopo del Giocatore 1, completando una quaterna di pedine del proprio colore.
- **Sabotatore:** Agente autonomo che, a differenza dei giocatori umani, non possiede un colore proprio e non può vincere completando un proprio allineamento. Il suo compito è impedire che uno dei due umani realizzi una quaterna di pedine, conducendo la partita al pareggio per riempimento completo delle 72 caselle della griglia. Poiché non possiede pedine proprie, il sabotatore interviene sul tabellone alternando l'uso di pezzi rossi e gialli all'interno della sequenza di gioco (come formalizzato nella Sezione 2.1.3).

Per ragioni di sintesi e chiarezza espositiva, i tre agenti d'ora in avanti verranno indicati rispettivamente con le sigle G_1, G_2 e G_S.

2.1.3 Il ciclo a quattro fasi

I turni di gioco non seguono una rotazione semplice a tre tempi, ma sono organizzati secondo una sequenza ciclica a quattro fasi, che nel prosieguo verranno indicate rispettivamente F_1, F_2, F_3 e F_4. In ogni fase viene assegnato all'agente di turno un colore di pedina e un preciso ruolo tattico:

- $F_1 - G_1$ **(Rosso):** il ciclo si apre con il turno del primo giocatore umano, che inserisce sulla griglia una pedina rossa. In questa fase il ruolo del giocatore è prettamente costruttivo e offensivo, orientato a impostare minacce dirette per arrivare alla vittoria.
- $F_2 - G_S$ **(Giallo):** il secondo turno del ciclo vede l'intervento dell'agente artificiale, vincolato dalle regole a piazzare una pedina gialla. Il suo compito tattico immediato consiste nell'interferenza difensiva contro G_1, sfruttando la pedina gialla per ostacolare l'allineamento rosso appena abbozzato o per occupare caselle strategiche che impediscano l'espansione dell'avversario.
- $F_3 - G_2$ **(Giallo):** a metà ciclo interviene il secondo giocatore umano, che gioca a sua volta con una pedina gialla. Analogamente al primo giocatore, il suo obiettivo è offensivo e mira alla costruzione di una quaterna vincente gialla, dovendo però adattare il proprio piano tattico alle conseguenze della mossa appena eseguita dal sabotatore.
- $F_4 - G_S$ **(Rosso):** il ciclo si conclude con il secondo intervento del sabotatore, che in questa fase deve invece inserire una pedina rossa. Il ruolo tattico dell'IA si ribalta specularmente in una difesa attiva mirata a contenere l'iniziativa di G_2 mediante l'impiego del colore rosso.

Nel corso del testo, l'intera rotazione verrà indicata sinteticamente con la notazione R-G-G-R (Rosso-Giallo-Giallo-Rosso), in cui le posizioni dispari corrispondono ai turni dei giocatori umani, mentre le posizioni pari identificano gli interventi del sabotatore.

La struttura dello schema R-G-G-R presenta vantaggi rilevanti sotto il profilo delle regole di gioco.

Innanzitutto, garantisce la perfetta parità numerica dei colori sulla griglia. Infatti, in ciascun ciclo di quattro semimosse vengono inserite due pedine per ciascun colore. La griglia mantiene così una perfetta parità numerica tra pedine rosse e gialle.

In secondo luogo, intervenendo in due fasi su quattro, G_S controlla esattamente la metà delle mosse totali. Tale frequenza di intervento è fondamentale: per contenere due avversari umani su una griglia di 72 caselle, una presenza limitata a un terzo delle mosse non sarebbe bastata a contrastare l'iniziativa combinata dei due giocatori.

Infine, la sequenza impone all'IA la gestione della pedina invertita. In F_2 e F_4, il sabotatore piazza rispettivamente una pedina gialla e una rossa. Ciò richiede un'attenta valutazione tattica, poiché quando l'IA colloca una pedina gialla in F_2 per spezzare un attacco di G_1, posiziona al tempo stesso un pezzo potenzialmente utile per G_2. L'algoritmo decisionale deve quindi bilanciare l'efficacia del blocco immediato con il rischio di favorire involontariamente il giocatore successivo.

2.2 Paradigma Logico e Architettura di Sistema

2.2.1 Motivazioni per l'uso della programmazione logica nei giochi da tavolo

La scelta del linguaggio per il motore logico e decisionale influisce direttamente sulla chiarezza e sulla facilità di manutenzione del sistema. Sebbene molti motori commerciali usino linguaggi imperativi come C++ per massimizzare il numero di posizioni calcolate al secondo, in questo lavoro di tesi si è scelto di impiegare SWI-Prolog (Wielemaker et al. 2012) per la gestione delle regole e per l'algoritmo di ricerca del sabotatore.

La programmazione logica dichiarativa si adatta in modo naturale alla modellazione dei giochi a regole formali, poiché la computazione procede attraverso la dimostrazione automatica di predicati logici. Un primo vantaggio essenziale risiede nella corrispondenza diretta tra le regole di gioco e la loro implementazione, dal momento che la dinamica dei turni e le transizioni di stato non richiedono complesse strutture di controllo imperativo (come diramazioni condizionali annidate o cicli di controllo espliciti), ma vengono espresse come relazioni pure.

Oltre alla chiarezza espressiva nella formalizzazione delle regole, il paradigma logico offre due ulteriori caratteristiche determinanti per l'efficacia del motore di gioco: l'unificazione sintattica e il backtracking. L'unificazione sintattica e il pattern matching permettono di estrarre e manipolare i dati di gioco in modo rapido e compatto. Strutture

composte come `turn(Phase, PlayerId, Kind, Piece)` vengono istanziate e verificate direttamente nella testa delle clausole, senza richiedere codice di conversione aggiuntivo.

Parallelamente, il meccanismo di risoluzione SLD di Prolog integra nativamente il backtracking, rivelandosi ideale per l'esplorazione dell'albero Minimax. Durante la discesa ricorsiva, al completamento dell'analisi di un ramo o al manifestarsi di un taglio, il motore ripristina automaticamente lo stato precedente per valutare la mossa alternativa.

2.2.2 L'integrazione di sistema tramite Janus

Se da un lato Prolog si distingue nell'inferenza logica e nella rappresentazione dichiarativa, dall'altro presenta limitazioni pratiche nella gestione della grafica interattiva, delle animazioni e dell'interfaccia utente. Per evitare queste limitazioni, il progetto adotta la libreria Janus, introdotta ufficialmente a partire da SWI-Prolog 9.2.

Janus stabilisce un collegamento a livello C tra l'interprete Python e il runtime di SWI-Prolog, operando all'interno del medesimo spazio di memoria. Da Python è possibile interrogare i predicati Prolog tramite la funzione `janus.query_once(Query)` che permette di convertire i tipi di dato Prolog direttamente nelle corrispondenti strutture dati Python.

Questa architettura consente una netta suddivisione dei ruoli, dove Prolog gestisce il modello formale, la verifica delle regole e la ricerca dell'IA, mentre Python si occupa dell'interfaccia utente, delle animazioni grafiche e della visualizzazione dei dati di telemetria.

2.2.3 Struttura modulare del software

L'architettura del sistema è organizzata in moduli distinti, ciascuno con una responsabilità ben definita:

- **Modulo `bitboard.pl`:** Implementa la rappresentazione della griglia a 81 bit, le maschere di colonna e la verifica delle condizioni di vittoria.
- **Modulo `threats.pl`:** Implementa la Threat-Space Analysis estesa alla griglia 8×9. Indicizza le 153 tuple vincenti, individua le minacce dirette e differite, calcola le penalità e restituisce il punteggio posizionale per l'IA.
- **Modulo `ai.pl`:** Contiene l'algoritmo Paranoid Minimax con potatura Alpha-Beta, la gestione della Tabella delle Trasposizioni e l'ordinamento delle mosse (Tromp statico e minacce dinamiche).
- **Modulo `engine.pl`:** Espone le funzioni principali richiamate da Python quali: inizializzazione della partita, esecuzione della mossa, calcolo della mossa dell'IA e impostazione della difficoltà.

- **Moduli Python `gui.py` e `main.py`:**
  - `main.py` gestisce l'inizializzazione del bridge Janus e la configurazione iniziale.
  - `gui.py` gestisce il rendering a schermo con Pygame, l'animazione della caduta delle pedine, l'evidenziazione delle minacce e l'interfaccia con i dati di telemetria.

La Figura 2.1 riassume l'architettura ibrida del sistema e le principali dipendenze tra il front-end Python e il motore logico SWI-Prolog.

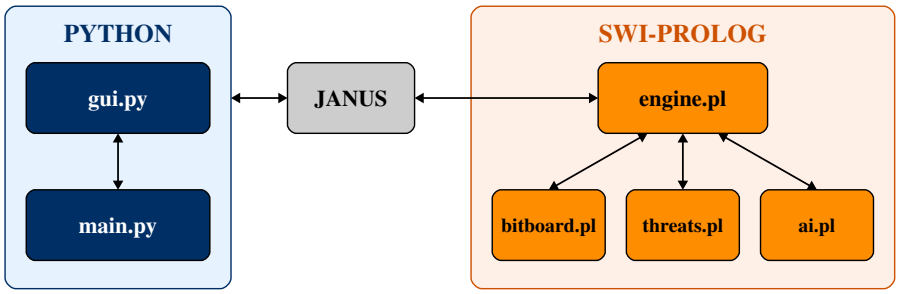


Figura 2.1. Architettura ibrida del sistema.

2.3 Rappresentazione dello Stato: Bitboard

2.3.1 Limiti della rappresentazione matriciale

Durante la ricerca avversariale con potatura Alpha-Beta su griglia 8×9, il motore decisionale può arrivare a valutare decine di migliaia di nodi al secondo. A ogni singolo nodo dell'albero è necessario eseguire quattro compiti fondamentali: verificare le colonne ancora disponibili; applicare la caduta della pedina per gravità; verificare se la nuova mossa ha generato un 4-in-linea vincente; valutare la qualità euristica della posizione.

Se la griglia fosse memorizzata come una matrice classica (ad esempio una lista di liste in Prolog), ciascuna di queste operazioni richiederebbe scansioni iterative con cicli annidati su righe e colonne, con una complessità pari a $\mathcal{O}(R \times C)$. Ripetere queste iterazioni milioni di volte avrebbe rallentato sensibilmente il motore, impedendogli di raggiungere profondità utili in tempi compatibili con un'interazione in tempo reale. Dunque, per ottimizzare le prestazioni, nel progetto è stata adottata la rappresentazione a bitboard ideata da John Tromp per il Forza 4 classico, adattata ed estesa per gestire la griglia a 8 righe per 9 colonne.

2.3.2 Codifica della griglia a 9 bit e bit sentinella

La tecnica della bitboard associa biunivocamente le caselle della griglia ai singoli bit di un registro numerico intero. L'intera configurazione di gioco su griglia 8×9 viene

memorizzata attraverso tre numeri interi a 81 bit. Il primo registro, `BoardP1`, rappresenta la maschera delle pedine rosse, in cui il bit i-esimo è pari a 1 se la casella corrispondente contiene una pedina rossa e 0 altrimenti. In modo del tutto speculare, `BoardP2` descrive la maschera delle pedine gialle. Infine, il registro `Mask`, calcolato come unione logica bit a bit (Mask = BoardP1 $\vee$ BoardP2), traccia la totalità delle caselle occupate sulla griglia indipendentemente dal colore della pedina.

La griglia è composta da 9 colonne. Per ciascuna colonna viene allocato uno spazio di memoria (*stride*) di 9 bit: i primi 8 bit corrispondono alle 8 celle fisiche della colonna (dalla riga 0 sul fondo alla riga 7 in cima), mentre il nono bit (indice 8 relativo alla colonna) funge da bit sentinella ed è mantenuto sempre a zero.

La relazione che converte le coordinate bidimensionali (riga $R \in \{0, \ldots, 7\}$, colonna $C \in \{1, \ldots, 9\}$) nell'indice di bit corrispondente BitIndex $\in \{0, \ldots, 80\}$ è:

$$\text{BitIndex} = (C - 1) \times 9 + R \tag{2.1}$$

I bit sentinella occupano gli indici 8, 17, 26, 35, 44, 53, 62, 71 e 80. La loro presenza è essenziale: quando si eseguono operazioni di scorrimento (*shift*) orizzontale o diagonale per verificare gli allineamenti, il bit sentinella posto a zero evita che i bit della parte alta di una colonna vadano a interferire con la base della colonna successiva.

2.3.3 Verifica delle quaterne vincenti con shift bitwise

Il controllo della presenza di una quaterna vincente è l'operazione più frequente della ricerca, venendo eseguita milioni di volte. Con le bitboard, la verifica viene effettuata simultaneamente su tutta la griglia per ciascuna delle quattro direzioni possibili, sfruttando gli shift definiti dallo stride a 9 bit:

- **Direzione Verticale (shift = 1):** Due caselle contigue nella stessa colonna si trovano a distanza di 1 bit.
- **Direzione Orizzontale (shift = 9):** Due caselle adiacenti sulla stessa riga in colonne consecutive distano esattamente 9 bit.
- **Direzione Diagonale Ascendente / (shift = 10):** Uno spostamento di una colonna a destra (+9) e di una riga in alto (+1) corrisponde a uno scostamento di $9 + 1 = 10$ bit.
- **Direzione Diagonale Discendente \ (shift = 8):** Uno spostamento di una colonna a destra (+9) e di una riga in basso (-1) corrisponde a uno scostamento di $9 - 1 = 8$ bit.

Il predicato ausiliario `check_dir(Board, Shift)` verifica la presenza di quattro pedine allineate nella direzione indicata mediante due shift e due operazioni logiche

AND:

```
check_dir(Board, Shift) :-
                          M is Board /\ (Board >> Shift),
                          Shift2 is 2 * Shift,
                          Win is M /\ (M >> Shift2),
                          Win =\= 0.
```
(2.2)

La Figura 2.2 sintetizza l'organizzazione di ciascuna colonna nella bitboard e gli scostamenti utilizzati per individuare gli allineamenti nelle quattro direzioni.

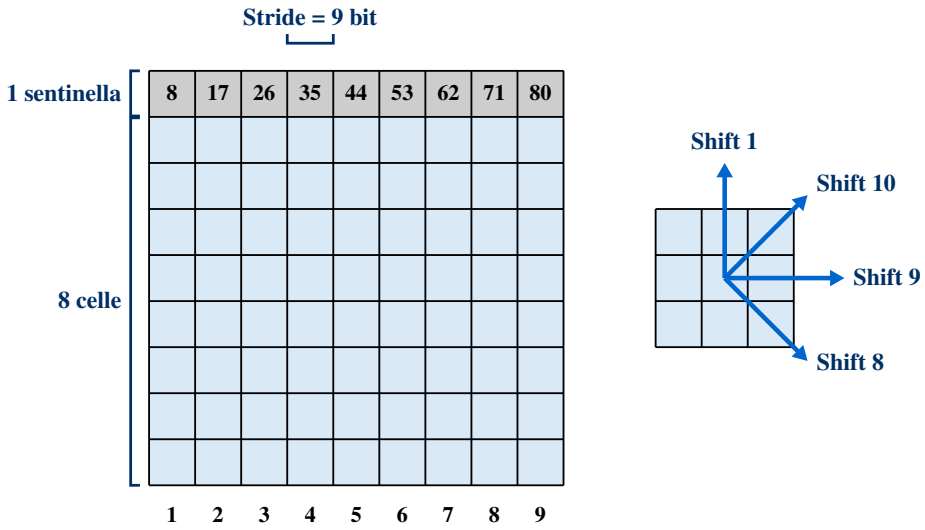


Figura 2.2. Codifica della bitboard e scostamenti associati.

2.3.4 Operazioni di gioco in tempo costante $\mathcal{O}(1)$

Grazie alla rappresentazione bitboard, le operazioni essenziali sulla griglia vengono risolte in tempo costante $\mathcal{O}(1)$ attraverso istruzioni logico-aritmetiche a livello di bit. Nel modulo `bitboard.pl` sono implementate le seguenti funzioni principali:

- **`can_play/2`:** Verifica della giocabilità di una colonna. Una colonna C può accogliere un'altra pedina se e solo se la sua casella più alta (riga 7) è ancora vuota. Dato che l'indice di bit corrispondente è $1 \ll ((C-1) \times 9 + 7)$, il controllo richiede una semplice operazione AND logica:

$$\text{can_play}(\text{Mask}, \text{Col}) \iff (\text{Mask} \wedge (1 \ll ((\text{Col} - 1) \times 9 + 7))) == 0 \tag{2.3}$$

- **`drop_piece/9`:** Caduta della pedina per gravità. Nel metodo Tromp, l'individuazione della casella libera più bassa di una colonna viene risolta sfruttando il

riporto binario dell'addizione. Aggiungendo a `Mask` il bit di fondo della colonna Bottom $= 1 \ll ((\text{Col} - 1) \times 9)$, il riporto supera tutte le celle già occupate e imposta a 1 esattamente la prima posizione vuota. Il bit della mossa `MoveBit` viene così isolato e aggiunto alla maschera del giocatore di turno con un'operazione OR.

- **`board_full/1`:** Verifica della saturazione della griglia. La griglia è completamente piena quando la casella superiore di tutte le 9 colonne è occupata. Questo controllo viene effettuato verificando simultaneamente i 9 bit di cima (7, 16, 25, 34, 43, 52, 61, 70, 79).

2.4 Esplorazione dello Spazio di Ricerca e Ottimizzazioni Avanzate

2.4.1 Ordinamento delle Mosse a due livelli

Come illustrato nel Capitolo 1, la potatura Alpha-Beta riduce la complessità temporale da $\mathcal{O}(b^d)$ a $\mathcal{O}(b^{d/2})$ solo se l'ordine di visita delle mosse consente di valutare le opzioni più promettenti fin dall'inizio dell'esplorazione. Qualora l'ordine di esplorazione fosse casuale o sfavorevole, il numero di nodi visitati si avvicinerebbe al caso peggiore, vanificando i vantaggi dei tagli. Nel modulo `ai.pl`, l'efficienza della ricerca è affidata a tre ottimizzazioni coordinate: l'ordinamento dinamico delle mosse, la memorizzazione degli stati visitati tramite Tabella delle Trasposizioni (Sezione 2.4.2) e una serie di filtri tattici preventivi alla radice (Sezione 2.4.3).

In questo quadro, la selezione dell'ordine con cui esaminare le mosse combina un criterio geometrico statico e una valutazione tattica dinamica.

Il primo livello si fonda sull'ordinamento geometrico statico di Tromp: sulla griglia a nove colonne il valore intrinseco delle caselle varia in funzione della loro posizione orizzontale. Una pedina collocata nella colonna centrale partecipa al massimo numero di allineamenti orizzontali, verticali e diagonali, mentre man mano che ci si sposta verso i bordi le opportunità di allineamento diminuiscono sensibilmente. Il predicato `tromp_order/1` definisce la sequenza di esplorazione in base alla centralità decrescente: l'esplorazione inizia sistematicamente dalla colonna centrale 5, procedendo verso i bordi.

Il secondo livello sovrappone a questo criterio statico una valutazione tattica dinamica gestita da `move_order_key/6`: prima di ordinare le colonne in base al centro, il motore simula l'applicazione di ciascuna mossa lecita e ne valuta l'effetto immediato attraverso il predicato `terminal_priority/3`. La priorità viene assegnata come segue: Priorità 0 (massima) alle mosse che provocano una vittoria immediata nei nodi MIN (che l'avversario umano giocherebbe sicuramente) o che completano il riempimento della

griglia nei nodi MAX (raggiungendo il pareggio favorevole all'IA); Priorità 1 alle mosse ordinarie (*quiet moves*), che vengono poi ordinate in base al criterio di centralità di Tromp.

2.4.2 La Tabella delle Trasposizioni

In Forza 4, la commutatività delle mosse in colonne differenti genera frequenti trasposizioni, ossia situazioni in cui sequenze di mosse diverse possono produrre la stessa identica configurazione della griglia. Senza memorizzare le posizioni già esaminate, l'algoritmo Alpha-Beta finirebbe per ricalcolare più volte lo stesso sotto-albero lungo rami differenti.

Nel sistema sviluppato, la memorizzazione è gestita tramite la Tabella delle Trasposizioni, memorizzata in Prolog tramite fatti dinamici indicizzati `tt_entry/8`:

```
tt_entry(Thread, BoardP1, BoardP2, Mask,
         Phase, Depth, Flag, StoredValue).
```
(2.4)

I campi registrati per ciascuna voce sono così articolati:

- **Chiave di Posizione (`BoardP1`, `BoardP2`, `Mask`, `Phase`):** La disposizione delle pedine è descritta dalle tre maschere bitboard, mentre `Phase` (da 1 a 4) assicura che configurazioni graficamente identiche, ma con turno o colore differente, vengano distinte correttamente.
- **Profondità di Ricerca (`Depth`):** Indica la profondità con cui è stato calcolato il valore memorizzato. Nell'implementazione proposta, la consultazione tramite unificazione esatta richiede che la profondità registrata coincida esattamente con quella richiesta nella ricerca corrente (`StoredDepth == CurrentDepth`).
- **Tipologia del Valore (`Flag`):** Definisce la natura del punteggio rispetto alla finestra di ricerca Alpha-Beta. Un flag `exact` attesta che la ricerca ha completato la valutazione del nodo senza tagli, per cui il valore corrisponde al vero punteggio Minimax all'interno dell'intervallo ($\alpha <$ Valore $< \beta$). Il flag `lower` segnala il verificarsi di un taglio Beta (Valore $\geq \beta$), rappresentando un limite inferiore certo; viceversa, il flag `upper` indica che nessun ramo ha superato il valore α, memorizzando un limite superiore valido per future potature.

Quando il motore interroga la tabella tramite il predicato `tt_lookup/10`, se è presente una voce registrata alla medesima profondità, una voce `exact` restituisce direttamente il punteggio memorizzato potando l'intero sotto-albero; una voce `lower` consente di alzare il parametro α; una voce `upper` permette di abbassare β. Se l'aggiornamento porta alla condizione $\alpha \geq \beta$, scatta un cutoff immediato senza dover generare o valutare i nodi figli.

2.4.3 Filtri tattici preventivi alla radice

Prima di avviare la ricerca ricorsiva lungo l'albero, il motore esegue alcuni controlli preliminari alla radice, gestiti dal predicato `select_move/11`. Il primo controllo riguarda l'intercettazione e il blocco delle vittorie immediate: tramite `winning_moves/3`, l'agente verifica se l'avversario umano di turno successivo dispone di una mossa che gli consentirebbe di completare una quaterna vincente al turno seguente. Nei livelli di gioco Medio, Difficile ed Estremo, in caso positivo, il sabotatore restringe subito l'insieme delle mosse candidate alle sole colonne che intercettano e bloccano la minaccia avversaria; al livello Facile, invece, la parata non è vincolata in modo deterministico, come descritto nella Sezione 2.7.1.

Il secondo controllo attua l'esclusione categorica delle mosse suicide tramite il predicato `suicide_moves/5`. Poiché in F_2 e F_4 il sabotatore piazza rispettivamente pedine gialle e rosse, una mossa è definita suicida quando inserire la pedina del colore correntemente giocato andrebbe a chiudere direttamente una quaterna vincente di quel colore; il predicato interroga `winning_moves/3` ed esclude tassativamente queste colonne.

Infine, il predicato `setup_moves/6` esclude le mosse di setup sfavorevole, caratteristiche della meccanica a gravità di Forza 4, che consistono nel piazzare una pedina alla riga R che funga da supporto per la casella soprastante $(R+1, C)$. Se l'avversario possiede già tre pedine allineate su quella casella superiore, giocare sotto di essa gli offrirebbe il supporto per vincere al turno successivo. Riconoscendo tali trappole, il motore scarta le mosse che preparerebbero una vittoria immediata per G_1 o G_2.

2.5 Funzione Euristica: Threat-Space Analysis

2.5.1 Ispirazione teorica e fondamento combinatorio

Nei giochi in cui la ricerca completa non è praticabile, la qualità delle scelte dell'agente dipende dall'efficacia della funzione di valutazione euristica $h(s)$. Una funzione basata sul semplice conteggio delle pedine non sarebbe in grado di cogliere le minacce articolate di Forza 4. Nel progetto, il modulo `threats.pl` adatta ed estende in Prolog i principi della Threat-Space Analysis (TSA) formulata da Victor Allis (1988).

La Threat-Space Analysis si basa sull'identificazione preventiva di tutte le possibili quaterne di caselle contigue e allineate presenti sulla griglia, definite tuple vincenti o finestre di allineamento. Mentre nel Forza 4 classico esistono 69 tuple vincenti, sulla griglia estesa 8×9 il numero sale a 153, di cui 48 orizzontali, 45 verticali, e 60 diagonali totali (ascendenti e discendenti).

All'avvio dell'applicazione, il predicato `init_threat_windows/0` calcola in anticipo ciascuna delle 153 tuple, estraendone la maschera bitboard a 81 bit e memorizzandola

come fatto dinamico precalcolato:

```
cached_window(Id, Mask, [B1, B2, B3, B4], Dir, B1).        (2.5)
```

2.5.2 Classificazione delle finestre di allineamento

A ogni chiamata dell'euristica per uno stato s, il motore esamina le 153 tuple precalcolate tramite il predicato `scan_all_windows/10`, determinando quante pedine rosse e quante gialle sono presenti in ciascuna finestra attraverso operazioni di popcount sulle maschere bitwise. In base a questo conteggio, la tupla viene inserita in una di quattro categorie operative.

La prima categoria comprende le tuple neutralizzate o bloccate, che contengono pedine di entrambi i colori, le quali, non potendo più essere completate da alcun giocatore, rappresentano per il sabotatore un risultato positivo permanente verso il pareggio. La seconda riguarda le minacce dirette (tre pedine con casella vuota immediatamente accessibile sul fondo o su una pedina già presente), che permetterebbero all'avversario di vincere al turno successivo e vengono penalizzate pesantemente. La terza include le minacce differite (tre pedine con casella vuota sospesa per mancanza di supporti sottostanti), la cui urgenza è quantificata da `support_info/4` in base ai pezzi mancanti per l'attivazione. La quarta categoria rileva infine le finestre aperte a due pedine, che segnalano l'avvio di una potenziale linea offensiva.

2.5.3 Valutazioni tattiche avanzate

La sola analisi delle singole tuple non è sufficiente per contrastare schemi di gioco avanzati. Il modulo `threats.pl` integra controlli mirati a individuare combinazioni tra più tuple. Il primo controllo riguarda il rilevamento di doppie minacce (due minacce dirette su colonne distinte) o fork tramite `fork_setup_count/4`. Con quest'ultimo termine si indica una mossa che genera una doppia minaccia. Poiché il sabotatore può intervenire in una sola colonna alla volta, la comparsa di un fork garantisce la vittoria all'avversario al turno successivo. Il predicato individua se una mossa attiva due tuple vincenti ad altezza giocabile, assegnando una penalità molto severa all'euristica per impedire la formazione di tale configurazione.

La Figura 2.3 confronta schematicamente le principali configurazioni riconosciute dalla Threat-Space Analysis.

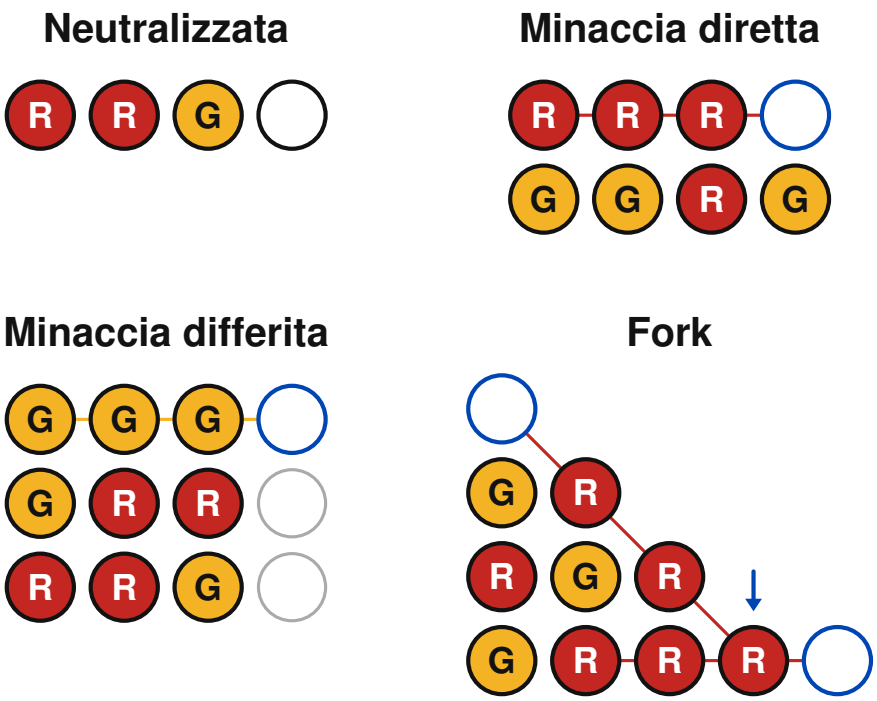


Figura 2.3. Rappresentazione schematica delle principali configurazioni.

Il secondo controllo identifica i supporti collegati mediante `linked_support_pairs/2`, riconoscendo coppie di minacce differite che condividono la stessa colonna di supporto. Giocare in quella colonna per bloccare la prima minaccia rischierebbe di abilitare immediatamente la seconda alla riga superiore. Infine, il predicato `phase_urgency_multiplier/4` pondera il peso delle minacce in funzione della vicinanza del turno dell'avversario nella sequenza R-G-G-R.

2.5.4 Composizione e pesi della funzione di valutazione

I diversi fattori analizzati confluiscono in un punteggio complessivo calcolato dal predicato `evaluate_saboteur/5`. I pesi assegnati ai singoli contributi, riassunti nella Tabella 2.1, sono stati calibrati in funzione della gravità e dell'urgenza temporale di ciascuna minaccia.

Il valore risultante viene infine limitato nell'intervallo $[-90\,000, +90\,000]$. In questo modo si garantisce che nessuna valutazione euristica, per quanto positiva o negativa, possa sovrapporsi ai valori assegnati agli stati terminali certi ($+100\,000$ per la vittoria dell'IA e $-100\,000$ per la sconfitta).

Parametro Valutato dall'Euristica	Peso	Descrizione Funzionale
Tuple Neutralizzate	+35	Premia l'eliminazione definitiva di linee vincenti
Minacce Dirette	−425	Penalizza pericoli di sconfitta al turno successivo
Minacce Differite	−140	Penalizza la presenza di minacce su supporti futuri
Finestre Aperte a 2 Pedine	−25	Scoraggia la formazione di nuovi allineamenti
Doppie minacce	−1250	Previene la creazione di doppie minacce
Mossa Preparatoria (Fork)	−350	Penalizza mosse che preparano un fork
Urgenza di Supporto Verticale	−35	Penalizza minacce vicine all'attivazione verticale
Supporti sulla Stessa Colonna	−120	Penalizza colonne con doppio innesco
Controllo di Fase della Minaccia	−90	Penalizza minacce differite a favore dell'umano
Pericolo Umano Dominante	−50	Penalizza l'iniziativa dell'umano più pericoloso
Controllo Umano Centro	−10	Penalizza l'occupazione umana delle colonne centrali

Tabella 2.1. Pesi ponderali della funzione euristica basata su Threat-Space Analysis.

2.6 Algoritmo Decisionale: Paranoid Minimax

2.6.1 Il modello Paranoid nel ciclo a quattro fasi

Il motore decisionale in `ai.pl` implementa il modello Paranoid descritto nella Sezione 1.6.3. Nel contesto della sequenza R-G-G-R, ciascuna fase viene classificata determinando se si tratta di un nodo MAX o MIN.

La ricerca ricorsiva alterna i livelli MIN e MAX. Nei livelli MAX (F_2 e F_4), il sabotatore massimizza l'euristica aggiornando il limite inferiore α; nei livelli MIN (F_1 e F_3), si ipotizza che l'avversario umano compia la scelta più sfavorevole per il sabotatore, minimizzando il punteggio e aggiornando il limite superiore β. La condizione di taglio $\alpha \geq \beta$ elimina i rami non influenti.

2.6.2 Approfondimento Iterativo

Nei livelli di difficoltà avanzati (Difficile ed Estremo), il motore non esegue direttamente la ricerca alla massima profondità, ma applica la tecnica dell'Approfondimento Iterativo (*Iterative Deepening Search*, IDS), gestita dal predicato `iterative_best_loop/11`:

$$d \in \{1, 2, 3, \ldots, \text{MaxDepth}\} \tag{2.6}$$

L'algoritmo avvia una serie di ricerche Alpha-Beta partendo da profondità 1 e incrementando il limite di un'unità per volta fino a `MaxDepth`. Sebbene questa procedura rivaluti i primi livelli, il tempo impiegato nelle iterazioni preliminari è contenuto, poiché,

per via della crescita geometrica dei nodi, l'ultima iterazione a profondità d assorbe la maggior parte del tempo complessivo.

L'Approfondimento Iterativo offre rilevanti vantaggi operativi nell'economia della ricerca. Il beneficio primario consiste nell'ordinamento guidato dalla mossa migliore. Nello specifico, la colonna valutata come migliore al termine dell'iterazione a profondità d viene registrata e valutata come primissima opzione all'iterazione successiva a profondità $d+1$ tramite il predicato `prefer_move/3`. Esaminare subito la mossa migliore consente di provocare tagli precoci lungo i rami successivi, avvicinando le prestazioni pratiche alla frontiera ideale del caso ottimo $\mathcal{O}(b^{d/2})$.

Inoltre, mentre in linea teorica l'Approfondimento Iterativo può essere impiegato come algoritmo *anytime* interrompibile allo scadere di un timer, nella presente implementazione (`iterative_best_loop/11`) l'esplorazione avanza in modo deterministico fino al raggiungimento del `MaxDepth` prefissato, senza interruzioni premature su base temporale. Il beneficio risiede dunque nell'ordinamento ottimizzato dell'albero ad ogni iterazione, che accelera sensibilmente la convergenza della ricerca più profonda.

2.6.3 La ricerca di quiescenza

Per limitare il rischio dell'effetto orizzonte (descritto nella Sezione 1.4.5), il sistema include una ricerca di quiescenza tramite `quiescence/8`. Quando la profondità principale si esaurisce, se la posizione presenta mosse forzate, il motore estende l'esplorazione per un massimo di due ulteriori semimosse: nei nodi MIN, vengono considerate forzate le mosse che permettono la chiusura immediata di una quaterna vincente; nei nodi MAX, vengono considerate forzate le mosse necessarie a bloccare una quaterna dell'avversario.

Se la configurazione è stabile, la ricerca di quiescenza si ferma e restituisce il valore euristico calcolato da `threats.pl`. In caso contrario, la ricerca di quiescenza estende selettivamente l'analisi degli stati tatticamente instabili fino al raggiungimento di una posizione sufficientemente stabile oppure fino all'esaurimento del budget massimo di estensione, fissato nell'implementazione a due livelli aggiuntivi.

2.7 Livelli di Difficoltà e Telemetria Prestazionale

2.7.1 Configurazione dei livelli di difficoltà dell'agente

Nello sviluppo di un sistema di gioco basato su Intelligenza Artificiale, la possibilità di regolare la difficoltà rappresenta un aspetto fondamentale sia per l'esperienza utente sia per condurre test sperimentali controllati. Un agente che giochi costantemente alla massima profondità algoritmica può risultare eccessivamente impegnativo per utenti non esperti, mentre un agente privo di profondità non offre elementi utili per analizzare

il comportamento strategico. Nel file `ai.pl`, le capacità decisionali del sabotatore sono parametrizzate su quattro livelli di difficoltà, illustrate nella Tabella 2.2.

Livello	Profondità	Criterio	Scopo
Facile	3 semimosse	Scelta casuale tra le top 5 mosse	Principiante; introduce imprecisioni controllate
Medio	4 semimosse	Blocco deterministico + scelta top 3	Bilanciato; blocca minacce ovvie ma ammette sviste tattiche
Difficile	5 semimosse	Deterministica con IDS	Competitivo; previene combinazioni complesse
Estremo	6 semimosse	Deterministica con IDS profondo	Ottimale; massimizza i cutoff

Tabella 2.2. Configurazione dei quattro livelli di difficoltà dell'agente sabotatore.

Nei livelli Facile e Medio, l'uso del predicato `random_member/2` introduce una componente casuale che rende le partite più varie ed evita sequenze ripetitive. In particolare, al livello Facile le eventuali mosse di blocco forzato vengono semplicemente unite alle migliori cinque mosse classificate, estraendo poi una mossa a caso dal pool risultante. La parata della minaccia è quindi probabile ma non matematicamente garantita, simulando possibili sviste da parte di un avversario principiante. Al contrario, al livello Medio il blocco è sempre deterministico, mentre nei livelli Difficile ed Estremo l'agente sceglie rigorosamente la mossa ottimale.

2.8 Interfaccia Grafica e Front-end con Pygame

2.8.1 Architettura dell'interfaccia grafica

Mentre la logica decisionale e le regole di gioco risiedono interamente in SWI-Prolog, l'interazione con i giocatori umani e la presentazione visiva sono gestite da un front-end Python sviluppato con la libreria Pygame (modulo `gui.py`).

L'architettura grafica si basa su un game loop sincronizzato a 60 fotogrammi al secondo (FPS). A ogni ciclo, l'applicazione gestisce anzitutto gli eventi dell'utente, intercettando i movimenti del mouse, i clic sulle colonne e le interazioni con i menu laterali. In secondo luogo, aggiorna la simulazione fisica della caduta delle pedine, offrendo un riscontro visivo naturale. Infine, tramite rendering a doppio buffer, tutti gli elementi visivi vengono disegnati su una superficie di memoria secondaria e proiettati a schermo contemporaneamente, evitando sfarfallii.

2.8.2 Pannello di controllo e telemetria

L'interfaccia grafica è stata concepita anche come strumento di analisi didattica e sperimentale, rendendo visibili a schermo i calcoli eseguiti dal motore Prolog. Sulla griglia di gioco, l'interfaccia evidenzia visivamente l'ultima pedina inserita mediante un anello circolare concentrico, consentendo ai giocatori di individuare subito la casella appena occupata. Le statistiche sulle minacce non appesantiscono la griglia con contorni grafici sulle celle, ma vengono riassunte in forma quantitativa nel pannello laterale.

Il pannello laterale di telemetria (HUD) presenta un riquadro informativo scuro ad alto contrasto che mostra in tempo reale la fase attiva del ciclo R-G-G-R, il colore della pedina corrente, il livello di difficoltà impostato, il tempo impiegato dall'IA nell'ultima decisione, i nodi esplorati nell'albero e la valutazione euristica della posizione.

La Figura 2.4 mostra l'organizzazione dell'interfaccia di gioco, nella quale la griglia 8×9 è affiancata dal pannello di controllo e dai principali dati di telemetria.

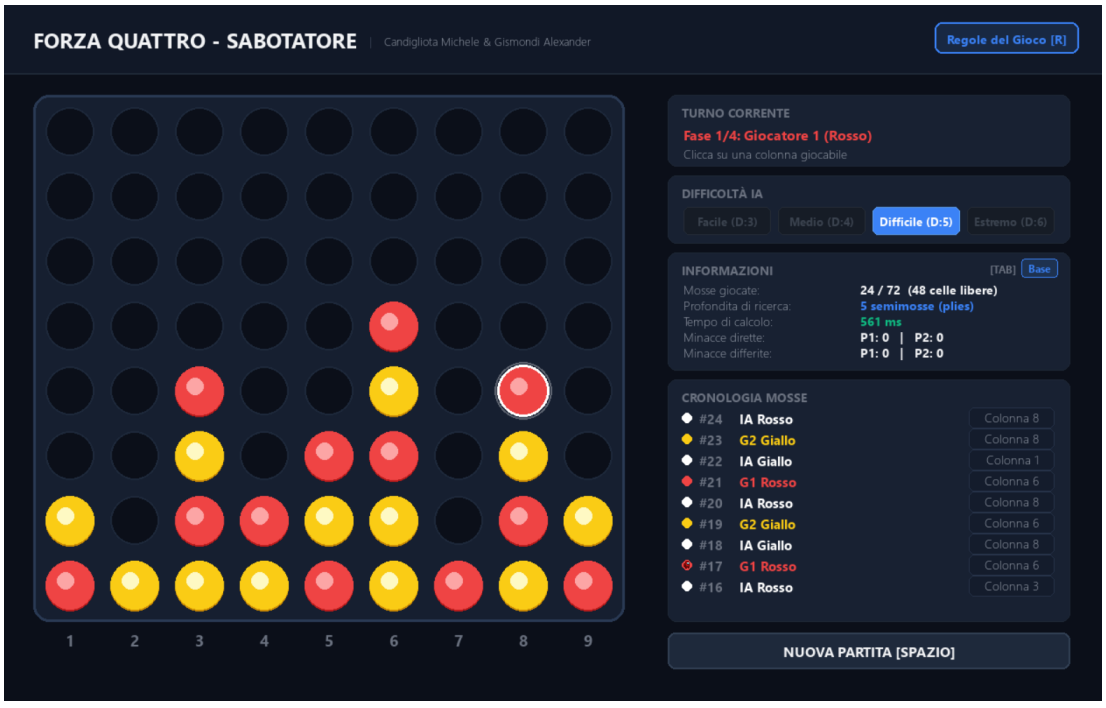


Figura 2.4. Struttura dell'interfaccia grafica del progetto sviluppato.

2.8.3 Elaborazione dell'IA e gestione del ciclo di rendering

Ai livelli di difficoltà più elevati, l'elaborazione dell'algoritmo decisionale in Prolog può richiedere da qualche centinaio di millisecondi a oltre un secondo di calcolo. Nel front-end, la chiamata a Prolog avviene in modo sincrono all'interno del metodo `execute_ai_turn()` nel ciclo principale di Pygame.

Durante il turno di G_S, per informare l'utente senza provocare blocchi improvvisi, l'interfaccia attiva lo stato `ai_thinking` e introduce una breve pausa visiva, consentendo a Pygame di renderizzare a schermo un messaggio informativo per quanto riguarda il processo decisionale del sabotatore. Durante l'effettiva ricerca svolta da Janus e Prolog, il thread principale rimane impegnato nel calcolo, con una momentanea sospensione della gestione degli eventi fino al termine della ricerca. Questa soluzione a singolo thread garantisce una totale stabilità ed evita complessità di sincronizzazione concorrente tra Python e il runtime C di SWI-Prolog.

2.8.4 Dall'architettura del sabotatore alla decisione sotto incertezza

L'architettura descritta e implementata in questo capitolo ha tradotto la complessità della variante a tre agenti in un sistema software modulare ed efficiente. L'impiego della rappresentazione a bitboard a 81 bit con operazioni bit a bit in tempo costante $\mathcal{O}(1)$, l'euristica fondata sulla Threat-Space Analysis e l'algoritmo Paranoid Minimax consentono al Sabotatore di elaborare decisioni tattiche solide e tempestive, perfettamente compatibili con la reattività richiesta da una partita in tempo reale.

Il comportamento dell'agente si basa tuttavia su un modello prudenziale, nel quale le possibili decisioni degli altri partecipanti vengono valutate assumendo che essi scelgano le mosse più sfavorevoli per il sabotatore. Questa impostazione garantisce una strategia robusta nel caso peggiore, ma non rappresenta completamente la varietà dei comportamenti che possono manifestarsi durante una partita reale. I giocatori umani, infatti, possono possedere livelli di esperienza differenti, commettere errori, non riconoscere una minaccia, adottare strategie subottimali oppure scegliere tra più alternative apparentemente equivalenti. Inoltre, poiché perseguono obiettivi individuali, possono ostacolarsi reciprocamente senza agire come una coalizione stabile contro l'agente artificiale.

Sebbene le regole del gioco siano deterministiche e lo stato della griglia sia interamente osservabile, rimane quindi un'incertezza relativa alle decisioni future degli altri agenti. Per affrontarla non è sufficiente individuare la singola continuazione peggiore, ma diventa necessario considerare più comportamenti possibili, confrontare gli scenari che ne derivano e valutare il rischio associato a ciascuna scelta. Una stessa mossa del Sabotatore potrebbe infatti risultare molto efficace contro un giocatore inesperto, ma eccessivamente rischiosa contro un avversario capace di riconoscere e sfruttare configurazioni tattiche più complesse.

Il capitolo seguente approfondirà pertanto il problema della decisione sotto incertezza, esaminando come rappresentare i possibili comportamenti degli altri partecipanti e come attribuire pesi o probabilità alle diverse mosse che essi potrebbero effettuare.

3 Modellazione dell'Incertezza

3.1 Dalla Certezza delle Regole all'Incertezza delle Decisioni

I Capitoli precedenti hanno descritto un gioco deterministico, a informazione perfetta e con uno stato completamente osservabile. La griglia, la fase del ciclo R-G-G-R, il colore della pedina da inserire e l'insieme delle mosse legali sono sempre noti. Una volta scelta una colonna, la gravità determina senza ambiguità la cella occupata e il nuovo stato. In questo senso non esistono eventi casuali interni alle regole, carte nascoste o percezioni incomplete.

La certezza della transizione non coincide però con la certezza della decisione. Prima che un giocatore umano muova, il sabotatore conosce le alternative disponibili ma non sa quale verrà selezionata. Il modello Paranoid presentato nella Sezione 1.6.3 e implementato nella Sezione 2.6.1 risolve il problema assumendo che ogni turno umano produca la continuazione più sfavorevole per l'agente artificiale. Tale ipotesi offre una garanzia nel caso peggiore, ma sostituisce la variabilità del comportamento umano con un avversario astratto, perfettamente razionale e implicitamente coordinato.

Come anticipato nella Sezione 2.8.4, l'approfondimento proposto in questo capitolo sposta quindi l'attenzione dalla sola ricerca avversariale alla decisione sotto incertezza. L'obiettivo non è rendere casuali le regole del gioco, ma rappresentare ciò che il sabotatore non conosce: il criterio con cui ciascun umano valuta la posizione, la probabilità che riconosca una minaccia, la profondità con cui pianifica e la propensione a scegliere una mossa rischiosa o subottimale.

3.1.1 Determinismo del dominio e incertezza comportamentale

È utile distinguere il dominio dal modello degli agenti. Il dominio rimane deterministico perché la funzione di transizione $T(s,a)$ associa a ogni stato s e a ogni azione legale a un solo stato successivo. L'incertezza riguarda invece la politica decisionale $\pi_i(a \mid s)$ del giocatore i, cioè la distribuzione delle scelte che l'agente potrebbe effettuare nello stato corrente:

$$T(s,a) = s' \quad \text{mentre} \quad \pi_i(a \mid s) \in [0,1] \quad \text{e} \quad \sum_a \pi_i(a \mid s) = 1 \tag{3.1}$$

Questa separazione evita un'ambiguità frequente. Dire che la decisione umana è incerta non significa affermare che ogni mossa sia equiprobabile o priva di motivazioni, ma significa ammettere che il sabotatore possiede una conoscenza incompleta del processo con cui l'avversario sceglie. Le probabilità diventano quindi una rappresentazione delle credenze dell'agente artificiale, non una proprietà fisica della griglia.

Nella teoria dei giochi con informazione incompleta, Harsanyi formalizza situazioni nelle quali i partecipanti non conoscono alcuni parametri rilevanti, ma mantengono distribuzioni soggettive sulle possibilità alternative. Nel progetto, il parametro ignoto può essere interpretato come il tipo comportamentale del giocatore: capacità tattica, profondità di previsione, regolarità delle scelte e sensibilità alle minacce.

3.1.2 Fonti di incertezza

Le principali fonti di incertezza derivano dalla distanza tra il comportamento teoricamente ottimale e quello effettivamente osservato. Esse agiscono su livelli differenti e non devono essere riunite in un'unica variabile indistinta.

Fonte	Oggetto Incerto	Effetto sulla Ricerca
Competenza	Riconoscimento di vittorie, blocchi, fork e supporti	Modifica la probabilità delle mosse forti e degli errori evidenti
Capacità di previsione	Numero di semimosse considerate dal giocatore	Cambia la capacità di anticipare minacce differite
Preferenze posizionali	Centralità, costruzione offensiva e avversione al rischio	Distingue mosse con valore tattico simile
Variabilità	Coerenza delle scelte nel corso della partita	Determina quanta fiducia attribuire al profilo stimato
Interazione tra umani	Conflitto o convergenza temporanea degli obiettivi	Impedisce di trattare stabilmente i due giocatori come una coalizione
Limite del modello	Errore dell'euristica e profondità finita	Introduce incertezza anche nella valutazione prodotta dal sabotatore

Tabella 3.1. Principali fonti di incertezza considerate.

La Tabella 3.1 separa l'incertezza sull'avversario dall'incertezza del valutatore. La prima riguarda ciò che faranno gli umani, la seconda riguarda la qualità con cui il motore stima le conseguenze di una mossa. Anche una distribuzione perfettamente calibrata sui comportamenti non elimina infatti l'effetto della profondità limitata o dell'errore euristico descritto nella Sezione 1.4.5.

3.1.3 Informazione osservabile e limiti dell'inferenza

Il sistema può osservare la sequenza degli stati, le colonne scelte, le mosse legali ignorate e le caratteristiche tattiche calcolabili tramite il modulo `threats.pl`. Non può invece osservare direttamente l'intenzione del giocatore, il tempo mentale dedicato alla scelta o il piano che egli ritiene di seguire. Il profilo deve quindi essere dedotto da indicatori indiretti.

Una singola mossa non è sufficiente per classificare un avversario. Una scelta debole può dipendere da inesperienza, distrazione, esplorazione o da una strategia che l'euristica non riconosce. Allo stesso modo, un principiante può selezionare casualmente la mossa migliore. Il modello deve quindi accumulare evidenze nel tempo, mantenere una misura di fiducia e conservare probabilità non nulle per più ipotesi finché le osservazioni non diventano discriminanti.

La profilazione descritta nelle sezioni successive non pretende dunque di misurare una competenza assoluta. Produce una stima operativa, valida per la partita corrente e limitata alle capacità rilevanti per questa variante di Forza 4.

3.2 Modelli dei Possibili Comportamenti Umani

3.2.1 Razionalità limitata e scelta rumorosa

Per valutare scenari alternativi è necessario specificare quali comportamenti siano plausibili. L'ipotesi di razionalità perfetta del Minimax assegna tutta la scelta alla migliore risposta. Una distribuzione uniforme compie l'errore opposto, perché tratta allo stesso modo una vittoria immediata e una mossa palesemente perdente. Un modello utile deve collocarsi tra questi estremi: le mosse migliori devono risultare più probabili, ma gli errori devono rimanere possibili.

Il concetto di razionalità limitata consente di rappresentare giocatori che perseguono un obiettivo coerente senza disporre di calcolo, conoscenza o attenzione illimitati. McKelvey e Palfrey propongono un modello di risposta quantale nel quale le strategie con utilità maggiore vengono scelte più frequentemente, ma non in modo deterministico. Questo principio è adatto al progetto perché trasforma il punteggio tattico di una mossa in una preferenza probabilistica.

Il modello non richiede di assumere che l'essere umano calcoli la stessa funzione euristica del sabotatore ma fornisce una scala interna con cui ordinare le alternative e stimare quanto una scelta appaia evidente. Può farlo sintetizzando il comportamento che il sistema prevede per il giocatore.

3.2.2 Profili di competenza

Per rendere interpretabile la previsione si possono introdurre tre profili latenti: principiante, intermedio ed esperto. Il termine intermedio sostituisce una classificazione genericamente mediocre e indica un giocatore capace di riconoscere la tattica immediata, ma non sempre le conseguenze differite. I profili non costituiscono etichette permanenti, ma sono ipotesi probabilistiche aggiornate nel corso della partita.

Profilo	Comportamento Atteso	Errori Ancora Plausibili	Concentrazione delle Probabilità
Principiante	Preferisce mosse visibili e può seguire la centralità senza analisi profonda	Mancati blocchi, supporti concessi e fork non riconosciuti	Sparsa su molte colonne
Intermedio	Riconosce vittorie e minacce dirette con buona regolarità	Valutazione incompleta di minacce differite e sequenze a più turni	Prevalente sulle prime mosse classificate
Esperto	Considera conseguenze tattiche, turnazione e interazione tra minacce	Errori rari o scelte tra alternative quasi equivalenti	Concentrata sulle mosse migliori

Tabella 3.2. Interpretazione dei profili comportamentali.

I profili devono essere definiti rispetto a eventi misurabili. Un esperto non è semplicemente il giocatore che vince, perché l'esito dipende anche dalle mosse dell'altro umano e dagli interventi del sabotatore. È invece il giocatore che, nelle situazioni in cui dispone di una scelta tatticamente dominante, la seleziona con elevata frequenza e riduce gli errori coerentemente osservabili.

3.2.3 Indicatori estratti dalle mosse

Ogni turno umano può generare un vettore di osservazioni. Le caratteristiche più informative sono già in larga parte calcolabili con le strutture presentate nel Capitolo 2, perché la Threat-Space Analysis identifica vittorie immediate, minacce dirette, minacce differite, fork e supporti:

- **Vittoria immediata:** Si verifica se il giocatore sceglie una colonna che completa una quaterna quando almeno una vittoria è disponibile.
- **Blocco necessario:** Misura se viene neutralizzata una minaccia immediata dell'altro umano quando il blocco è legalmente possibile.

- **Prevenzione del fork:** Valuta la capacità di evitare una mossa che consenta una doppia minaccia alla fase successiva.
- **Qualità relativa:** Confronta il valore euristico della colonna scelta con il valore della migliore alternativa disponibile dal punto di vista del giocatore.
- **Coerenza strategica:** Rileva se le scelte successive privilegiano sistematicamente centro, costruzione offensiva, blocco o rischio.
- **Errore grave:** Registra una mossa che concede una vittoria immediata evitabile o ignora una vittoria certa.

Gli indicatori binari, come il riconoscimento di una vittoria, possono essere combinati con misure continue. La perdita di qualità è data dalla differenza ΔQ tra la valutazione della migliore mossa e quella scelta. Un errore di un punto tra alternative equivalenti ha un significato diverso da una scelta che trasforma una posizione stabile in una sconfitta forzata.

3.2.4 Interazione tra i due giocatori

I due umani devono essere modellati separatamente. Il Giocatore 1 (G_1) persegue la vittoria rossa e il Giocatore 2 (G_2) quella gialla, quindi una mossa utile al primo può danneggiare il secondo. La temporanea convergenza contro il sabotatore non implica una coalizione stabile, come già osservato nel confronto tra Max-N e modello Paranoid nella Sezione 1.6.

Si mantengono, quindi, dei profili e delle distribuzioni distinte per ogni giocatore; l'interazione riappare solamente nella valutazione dello scenario: la mossa prevista per G_1 modifica lo stato nel quale agiranno G_S e G_2.

3.3 Rappresentazione dell'Incertezza

3.3.1 Alberi di possibilità e attribuzione iniziale dei pesi

Una volta definiti i comportamenti possibili, l'incertezza deve essere tradotta in quantità concrete, utilizzabili dal motore. Per fare ciò a ogni nodo umano dell'albero si associa una distribuzione sulle colonne legali, dove la somma delle probabilità è pari a uno e le mosse impossibili ricevono probabilità nulla.

Uno scenario è una sequenza di mosse future accompagnata dal relativo peso. Se al primo turno umano la mossa a_1 ha probabilità p_1 e al turno successivo la mossa a_2 ha probabilità condizionata p_2, il peso del percorso è $p_1 p_2$. La valutazione finale dello scenario deriva dallo stato di frontiera mediante la funzione `evaluate_saboteur/4` descritta nella Sezione 2.5.4.

L'albero mantiene nodi decisionali nei turni di G_S e nodi previsionali nei turni umani, dove nei primi l'agente confronta le proprie azioni e nei secondi combina le continuazioni secondo i pesi assegnati alle possibili scelte. Questa rappresentazione non impone uno specifico algoritmo di ricerca ma definisce soltanto il significato dei valori che devono essere propagati.

La Figura 3.1 rappresenta l'alternanza tra decisioni del sabotatore e risposte probabilistiche dei giocatori umani, mostrando come il peso di uno scenario derivi dal prodotto delle probabilità associate al percorso.

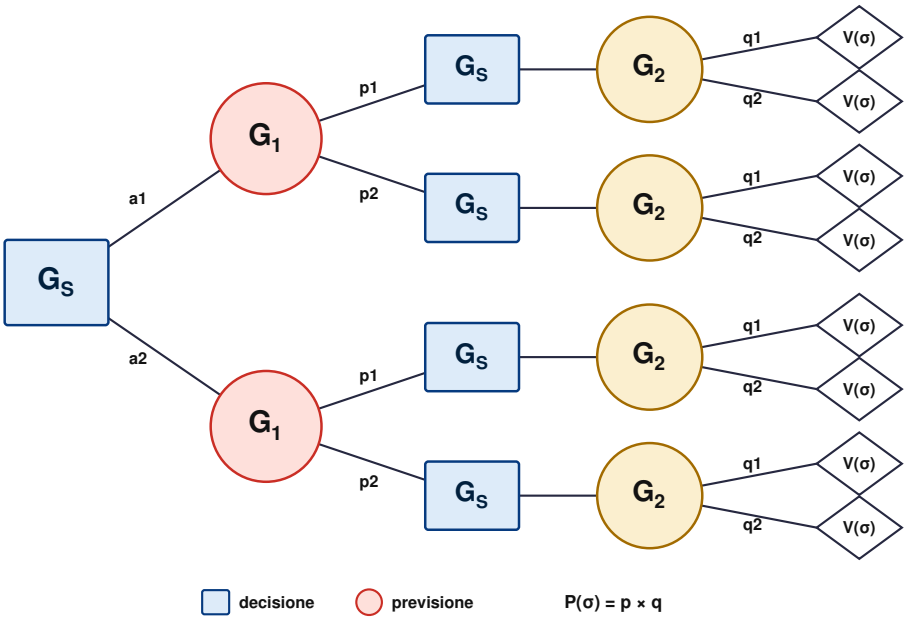


Figura 3.1. Albero degli scenari con probabilità associate.

All'inizio della partita, le informazioni sul giocatore sono ancora limitate. Il sistema può quindi adottare un prior uniforme, assegnando la stessa probabilità iniziale ai tre profili, oppure un prior informato, basato sul livello dichiarato dall'utente o su dati aggregati raccolti in precedenza. Le probabilità associate ai profili vengono poi aggiornate progressivamente sulla base delle mosse osservate.

Questa distribuzione descrive la probabilità che il giocatore appartenga a ciascun profilo, ma non permette ancora di prevedere direttamente quale mossa sceglierà. A tale scopo, il sistema associa a ogni profilo una distribuzione sulle colonne disponibili: profili diversi attribuiscono probabilità differenti alle mosse in funzione della loro qualità tattica. La previsione complessiva si ottiene combinando tali distribuzioni e pesandole mediante le credenze correnti sui profili.

3.3.2 Trasformazione dei valori euristici in probabilità

Per costruire la distribuzione delle mosse associata a ciascun profilo, occorre trasformare i rispettivi valori euristici in probabilità. Una possibile trasformazione è la funzione softmax. Per un giocatore i, a ogni mossa legale a viene assegnato un valore $Q_i(s,a)$, calcolato dal suo punto di vista. Il parametro λ_k, specifico del profilo k, controlla quanto la distribuzione si concentra sulle mosse con valore più elevato:

$$P(a \mid s, k) = \frac{\exp(\lambda_k Q_i(s, a))}{\sum_b \exp(\lambda_k Q_i(s, b))} \tag{3.2}$$

Quando λ è vicino a zero, le differenze tra i valori incidono poco e la distribuzione è quasi uniforme. All'aumentare di λ, la probabilità si concentra progressivamente sulle mosse con valore più elevato. Il profilo principiante utilizza quindi un parametro più basso, quello intermedio un valore maggiore e quello esperto un parametro elevato. Per evitare una falsa precisione, λ deve essere calibrato su partite osservate e non scelto soltanto per convenienza.

Come mostrato nella Figura 3.2, l'aumento del parametro di razionalità concentra progressivamente la probabilità sulle mosse valutate come migliori.

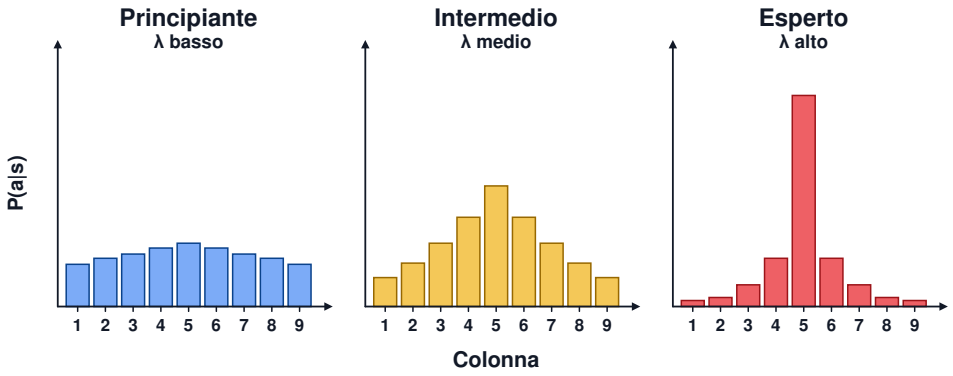


Figura 3.2. Rappresentazione dell'effetto del parametro λ sulla distribuzione softmax.

Le mosse terminali richiedono un trattamento prioritario: se una colonna produce una vittoria immediata, la sua probabilità dovrebbe essere molto elevata per i profili competenti, mentre se una mossa causa una sconfitta evitabile, la probabilità deve ridursi drasticamente. Non deve però essere necessariamente azzerata, perché un modello adattivo deve poter rappresentare anche l'errore umano.

3.3.3 Incertezza sulle probabilità

Attribuire una probabilità non elimina l'incertezza ma la sposta nel modello. Le stime dipendono dal prior, dalla funzione Q, dal parametro di razionalità λ_k e dal numero di

osservazioni. È quindi necessario distinguere la probabilità prevista dalla fiducia nella previsione.

All'inizio della partita il sistema dispone di poche informazioni sul giocatore. Può quindi assegnare una probabilità ai diversi profili, ma questa prima stima deve essere considerata poco affidabile. Se le mosse successive mostrano un comportamento coerente, il sistema accumula nuove informazioni e attribuisce una probabilità sempre maggiore al profilo che descrive meglio il giocatore. Al contrario, se il giocatore alterna mosse efficaci ed errori oppure cambia stile di gioco, le probabilità rimangono distribuite tra più profili. Il livello di incertezza della stima può essere misurato attraverso l'entropia: un valore elevato indica che il sistema non è ancora in grado di individuare chiaramente il profilo del giocatore, mentre un valore basso segnala una maggiore sicurezza nella classificazione.

La classificazione non deve comunque diventare definitiva; il sistema deve poter modificare la propria stima quando emergono nuovi comportamenti. A tale scopo, è possibile ridurre gradualmente l'importanza delle mosse più lontane nel tempo oppure ammettere il passaggio da un profilo all'altro, così che il modello possa adattarsi a un giocatore che migliora durante la partita, cambia strategia o viene sostituito da un'altra persona.

3.4 Valutazione degli Scenari e Gestione del Rischio

3.4.1 Valore atteso

La distribuzione sulle mosse rende possibile valutare più scenari, ma non stabilisce da sola quale strategia scegliere; il sabotatore, infatti, deve combinare qualità attesa, gravità degli esiti negativi e affidabilità del modello. Una decisione fondata soltanto sulla media può preferire una mossa con molti esiti moderatamente positivi e una piccola probabilità di sconfitta immediata perché nel progetto, tale rischio non è accettabile quando esiste un'alternativa sicura.

Il valore atteso permette di stimare la convenienza di una mossa del sabotatore considerando tutti gli scenari che potrebbero verificarsi dopo la sua scelta. Ogni scenario viene valutato in base a due elementi: la probabilità che si verifichi e il vantaggio o lo svantaggio che produrrebbe per il sabotatore. Indicando con $S(a)$ l'insieme degli scenari successivi alla mossa a, con $P(\sigma \mid a)$ il peso dello scenario σ e con $V(\sigma)$ la sua valutazione, si ottiene:

$$\mathbb{E}[V \mid a] = \sum_{\sigma \in S(a)} P(\sigma \mid a) V(\sigma) \tag{3.3}$$

La formula moltiplica il valore di ogni scenario per la sua probabilità e somma tutti i risultati. Gli scenari più probabili influenzano quindi maggiormente la valutazione finale, mentre quelli meno probabili hanno un peso inferiore. Grazie a questo criterio si possono confrontare più alternative tenendo conto non soltanto delle loro conseguenze, ma anche della probabilità con cui tali conseguenze potrebbero verificarsi.

Il valore atteso, tuttavia, rappresenta una media e non mostra da solo quanto una scelta sia rischiosa. Due mosse possono avere lo stesso valore atteso pur presentando conseguenze molto diverse: una può produrre risultati abbastanza stabili, mentre l'altra può offrire un grande vantaggio ma anche comportare una piccola probabilità di sconfitta immediata. Per questo motivo, il valore atteso deve essere affiancato da una valutazione del rischio e degli scenari peggiori.

3.4.2 Perdita nel caso peggiore e rischio di coda

Per il sabotatore, la sconfitta corrisponde alla vittoria di uno dei due umani e riceve un valore terminale fortemente negativo. La funzione di rischio deve quindi dare rilievo alla probabilità di tali esiti, anche quando è bassa. Una mossa con un buon valore atteso potrebbe comunque essere poco conveniente se lascia aperta una possibilità concreta di vittoria per uno degli avversari.

Considerare esclusivamente il caso peggiore, tuttavia, può rendere il comportamento del sabotatore eccessivamente prudente. Lo scenario più negativo potrebbe infatti avere una probabilità molto bassa oppure dipendere da una previsione poco affidabile. Per ottenere una valutazione più equilibrata è quindi possibile analizzare il rischio di coda, cioè l'insieme degli scenari più sfavorevoli anziché soltanto il peggiore in assoluto.

Il rischio di coda può essere rappresentato attraverso la probabilità complessiva di una sconfitta oppure mediante il valore medio degli scenari peggiori. In questo modo, il sabotatore può distinguere tra una conseguenza negativa quasi impossibile e un gruppo di esiti sfavorevoli che, nel loro insieme, costituiscono un rischio significativo. La decisione finale tiene così conto sia del rendimento medio della mossa sia della gravità e della probabilità delle sue conseguenze peggiori.

3.4.3 Valutazione corretta per il rischio

Una volta stimata la distribuzione delle possibili mosse del giocatore, il G_S deve decidere quanto affidarsi a tale previsione. A questo scopo, può adottare un criterio ibrido che consideri sia il valore medio delle continuazioni previste sia il loro esito peggiore, penalizzando inoltre le mosse che possono condurre alla vittoria di uno dei giocatori umani.

Il parametro ρ determina il peso attribuito allo scenario peggiore: quanto maggiore è il suo valore, tanto più prudente sarà il comportamento del sabotatore. Il coefficiente κ

stabilisce invece quanto severamente penalizzare una mossa associata a una possibile sconfitta. In questo modo, l'agente può sfruttare le previsioni probabilistiche senza trascurare continuazioni meno probabili ma particolarmente pericolose:

$$J(a) = (1-\rho)\mathbb{E}[V \mid a] + \rho \min_{\sigma \in S(a)} V(\sigma) - \kappa P(\text{sconfitta} \mid a) \tag{3.4}$$

Quando ρ è prossimo a uno, il comportamento si avvicina al modello prudenziale del Capitolo 2, invece quando ρ diminuisce, il G_S attribuisce maggiore importanza agli scenari probabili e può scegliere mosse opportunistiche. Il parametro non deve dipendere soltanto dal profilo stimato ma anche dalla fase della partita, dal numero di mosse sicure e dalla fiducia nel modello.

Condizione	**Interpretazione**	**Impostazione Strategica**
Minaccia terminale disponibile	Un umano può vincere al turno successivo	Vincolo di sicurezza e blocco obbligatorio
Profilo incerto	Posterior disperso o poche osservazioni	ρ elevato e maggiore peso al caso peggiore
Profilo esperto probabile	Alta probabilità di riconoscere la tattica	Strategia prudente sulle minacce profonde
Profilo principiante probabile	Errori tattici osservati con regolarità	Maggiore peso al valore atteso senza rimuovere i vincoli
Alternative quasi equivalenti	Differenze di valore contenute	Preferenza per la mossa meno sensibile all'errore del modello

Tabella 3.3. Relazione tra condizioni osservate, rischio e comportamento strategico.

La probabilità non deve sostituire le regole tattiche certe. I filtri preventivi descritti nella Sezione 2.4.3 continuano ad avere precedenza: una vittoria immediata dell'umano deve essere bloccata, una mossa suicida deve essere esclusa e una configurazione che offre un supporto vincente certo deve essere evitata quando esistono alternative legali.

La valutazione probabilistica opera dopo questi controlli, tra le mosse che rispettano i vincoli di sicurezza; in questo modo l'adattamento influenza la strategia senza indebolire conoscenze certe.

3.4.4 Controllo della complessità

La valutazione di più comportamenti possibili aumenta il numero di scenari che il sistema deve analizzare. A ogni turno umano, infatti, ciascuna mossa legale può generare nuove continuazioni, facendo crescere rapidamente l'albero di ricerca ed esaminare tutte le

alternative fino alla stessa profondità potrebbe quindi richiedere tempi non compatibili con lo svolgimento di una partita in tempo reale.

Per limitare il costo computazionale, le mosse degli altri giocatori possono essere ordinate considerando sia la probabilità stimata sia la loro importanza tattica. Il sistema può quindi approfondire per prime le risposte più plausibili o pericolose e interrompere l'espansione quando gli scenari selezionati raggiungono una determinata probabilità cumulativa; per esempio, potrebbe analizzare le continuazioni che, nel loro insieme, coprono il 90% delle scelte previste. Le mosse escluse dall'espansione non devono però essere considerate impossibili. La loro probabilità deve essere conservata come massa residua, in modo da rappresentare l'incertezza non analizzata e impedire che il sistema attribuisca una fiducia eccessiva agli scenari approfonditi. Inoltre, le risposte che conducono a una vittoria immediata o a una grave minaccia devono essere valutate anche quando il modello assegna loro una probabilità ridotta.

Un'ulteriore soluzione consiste nell'utilizzare il modello probabilistico soltanto nei primi livelli dell'albero, nei quali la previsione del comportamento umano è più attendibile. Nei livelli successivi può essere nuovamente impiegato il Paranoid Minimax. Questa strategia ibrida permette di adattare le decisioni del sabotatore alle mosse più probabili degli avversari, mantenendo al tempo stesso un comportamento prudente per gli scenari più lontani e incerti. In questo modo, il costo della ricerca rimane compatibile con i requisiti di reattività del sistema.

3.5 Profilazione Bayesiana dei Giocatori

3.5.1 Stato delle credenze

Il modello bayesiano fornisce un meccanismo coerente per aggiornare il profilo del giocatore man mano che vengono osservate nuove mosse, dove il sistema non sceglie subito un'etichetta definitiva, ma mantiene una probabilità per ciascuna categoria.

Per ogni giocatore i si definisce la variabile K_i appartenente all'insieme $\{P, I, E\}$: prima della prima mossa, il sabotatore dispone di un prior $P(K_i)$ e dopo t osservazioni, la credenza diventa il posterior $P(K_i \mid o_{1:t})$. I due giocatori conservano distribuzioni distinte perché possono avere competenze e stili differenti. Il posterior rappresenta ciò che il sistema crede alla luce delle evidenze disponibili. Se i valori sono, ad esempio, 0,15 per principiante, 0,60 per intermedio e 0,25 per esperto, la classificazione più probabile è intermedio, ma l'incertezza residua rimane esplicita. Il motore può quindi usare l'intera distribuzione invece di ridurla prematuramente a una singola classe.

3.5.2 Aggiornamento mediante il teorema di Bayes

Quando viene osservata una nuova mossa o_t, il sistema valuta quanto essa sia compatibile con ciascun profilo. La verosimiglianza $P(o_t \mid K_i = k)$ viene moltiplicata per il posterior precedente e il risultato viene normalizzato; questo principio deriva dal teorema di Bayes, pubblicato nel lavoro di Bayes e Price (1763):

$$P(K_i = k \mid o_{1:t}) = \frac{P(o_t \mid K_i = k)P(K_i = k \mid o_{1:t-1})}{\sum_j P(o_t \mid K_i = j)P(K_i = j \mid o_{1:t-1})} \tag{3.5}$$

Le verosimiglianze possono essere definite inizialmente da regole trasparenti; ad esempio, se un giocatore sceglie una vittoria immediata, l'osservazione è compatibile con tutti i profili ma più probabile sotto quello esperto; in seguito, se ignora ripetutamente blocchi obbligatori, cresce la probabilità del profilo principiante; se riconosce minacce differite e previene fork, aumenta il peso del profilo esperto.

Le probabilità condizionate non devono assumere valori zero o uno in modo rigido, questo perché un singolo errore non rende impossibile il profilo esperto e una singola mossa corretta non dimostra competenza elevata; proprio per questo l'impiego di valori smussati impedisce che il posterior collassi a causa di un'osservazione isolata.

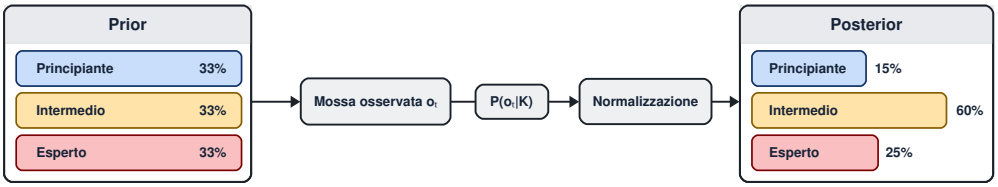


Figura 3.3. Esempio di aggiornamento bayesiano delle probabilità dei profili.

3.5.3 Costruzione della verosimiglianza

La verosimiglianza può essere ricavata dal modello softmax presentato nella Sezione 3.3.2. Per ogni profilo, il sistema calcola la probabilità delle mosse legali in base al loro valore tattico. La probabilità assegnata alla mossa effettivamente scelta viene quindi utilizzata come verosimiglianza nell'aggiornamento bayesiano. La valutazione può essere completata considerando eventi tattici particolarmente significativi, come il riconoscimento di una vittoria immediata, il blocco di una minaccia o la prevenzione di un fork. Questi eventi devono avere un peso maggiore rispetto a preferenze meno importanti, come la scelta tra due colonne con valore simile.

Il posterior sui profili viene infine trasformato nella previsione concreta delle mosse. La probabilità di giocare una colonna è ottenuta mediando le distribuzioni specifiche

dei profili con i rispettivi pesi posteriori:

$$P(a \mid s, o_{1:t}) = \sum_{k \in \{P,I,E\}} P(a \mid s, K_i = k) P(K_i = k \mid o_{1:t}) \tag{3.6}$$

Questa formula permette una transizione graduale da parte del G_S, infatti esso non passa bruscamente da una strategia per principianti a una strategia per esperti, ma modifica progressivamente le proprie previsioni.

3.5.4 Stabilità, oblio e cambiamento di stile

Un aggiornamento puramente cumulativo attribuisce alle prime osservazioni un'influenza permanente; questo può essere indesiderato in vari contesti quali il giocatore che apprende durante la partita o cambia comportamento. Si può quindi introdurre un fattore di oblio $\gamma \in (0,1]$, che riduca gradualmente il peso delle evidenze passate prima di incorporare la nuova osservazione per dare maggiore dinamicità alla lettura della tipologia di giocatore.

Il fattore non deve essere troppo basso, altrimenti il profilo oscilla dopo ogni mossa. Una soluzione pratica consiste nel mantenere il posterior stabile nelle prime osservazioni, applicare una soglia minima di evidenza prima di modificare sensibilmente la strategia e consentire un adattamento più rapido soltanto quando compaiono più segnali coerenti.

3.6 Valutazione delle Mosse e Sintesi del Modello

3.6.1 Flusso decisionale proposto

Le componenti precedenti possono confluire tutte in una procedura decisionale. La strategia deve prima garantire la sicurezza tattica, poi prevedere le risposte umane, valutare gli scenari e infine selezionare la mossa con il miglior compromesso tra rendimento e rischio:

1. **Generazione:** Il motore produce le colonne mediante la rappresentazione bitboard descritta nella Sezione 2.3.
2. **Filtraggio:** Vengono applicati blocchi obbligatori, esclusione delle mosse suicide e controllo dei supporti pericolosi (Sezione 2.4.3).
3. **Previsione:** Per ogni nodo umano si calcola la distribuzione delle mosse a partire dal posterior del giocatore attivo.
4. **Espansione:** Gli scenari vengono esplorati in ordine di probabilità e rilevanza tattica, entro il limite di profondità e di massa cumulativa.
5. **Valutazione:** Ogni stato di frontiera riceve il punteggio della Threat-Space Analysis e gli stati terminali mantengono i valori estremi definiti nel Capitolo 2.

6. **Correzione per il rischio:** Il valore atteso viene combinato con caso peggiore, probabilità di sconfitta e affidabilità del modello.
7. **Selezione:** Il sabotatore sceglie la colonna con il valore strategico corretto per il rischio più elevato.

La procedura conserva la modularità esistente, dove `bitboard.pl` e `threats.pl` continuano a fornire rappresentazione e caratteristiche tattiche; `ai.pl` ospita la propagazione dei valori ed un nuovo modulo teorico potrebbe mantenere i posterior e restituire la distribuzione delle mosse; `engine.pl` esporrebbe a Python soltanto i nuovi dati di telemetria necessari.

Il punteggio finale deve mantenere separati tre contributi: qualità posizionale, previsione comportamentale e rischio. Indicando con $H(a)$ il valore euristico immediato, con $\mathbb{E}[V \mid a]$ il valore atteso degli scenari e con $R(a)$ la penalità di rischio, una formulazione generale è:

$$\text{Score}(a) = w_h H(a) + w_e \mathbb{E}[V \mid a] - w_r R(a) \tag{3.7}$$

I pesi w_h, w_e e w_r possono dipendere dalla profondità residua, dalla fase R-G-G-R e dalla fiducia nel modello. Se il posterior è disperso, aumenta w_r; se il profilo è ben identificato e le mosse sicure sono numerose, cresce il contributo del valore atteso.

Gli stati terminali restano dominanti. Nessuna combinazione di pesi deve rendere preferibile una sconfitta certa a una posizione non terminale. Il vincolo già adottato nel Capitolo 2, che separa l'intervallo euristico dai valori ± 100.000, deve essere mantenuto anche nella nuova funzione.

3.6.2 Considerazioni conclusive

I vincoli descritti completano il passaggio dalla previsione del comportamento umano alla scelta della mossa del sabotatore. Nel complesso, il capitolo ha mostrato che l'incertezza non riguarda lo stato del gioco, poiché le regole sono deterministiche e la griglia è completamente osservabile, ma le decisioni future dei due giocatori umani. La difficoltà deriva quindi dalla varietà delle azioni che essi possono scegliere durante la partita.

Per rappresentare questa incertezza, il modello non assegna la stessa probabilità a tutte le risposte e non presume che i giocatori scelgano sempre la mossa ottimale. La probabilità di ciascuna azione dipende dal suo valore tattico e dal profilo di competenza attribuito al giocatore. L'aggiornamento bayesiano modifica durante la partita le probabilità associate ai profili principiante, intermedio ed esperto. Le credenze così aggiornate permettono di stimare le risposte più plausibili, costruire scenari alternativi e valutarli considerando sia il risultato atteso sia il rischio. Il sabotatore può quindi

adattare la propria decisione alle evidenze osservate senza affidarsi esclusivamente all'ipotesi che gli altri partecipanti agiscano sempre nel modo per lui più sfavorevole.

La procedura costituisce una possibile estensione dell'architettura realizzata. La sua integrazione completa richiederebbe la calibrazione dei parametri che distinguono i profili, la definizione empirica delle probabilità associate alle mosse e una successiva fase di sperimentazione. Il contributo del capitolo consiste pertanto nella definizione di un modello con cui il G_S potrebbe passare da una strategia esclusivamente prudenziale a una strategia capace di adattarsi al comportamento osservato dei giocatori.

4 Risultati Sperimentali, Conclusioni e Sviluppi Futuri

4.1 Risultati e Metriche Sperimentali

4.1.1 Analisi degli esiti per livello di difficoltà

La metrica principale per misurare l'efficacia del sabotatore è la percentuale di partite concluse in parità, che coincide con il raggiungimento del suo obiettivo.

La Figura 4.1 riassume i risultati ottenuti su un totale di 400 partite controllate (100 partite per ciascun livello di difficoltà):

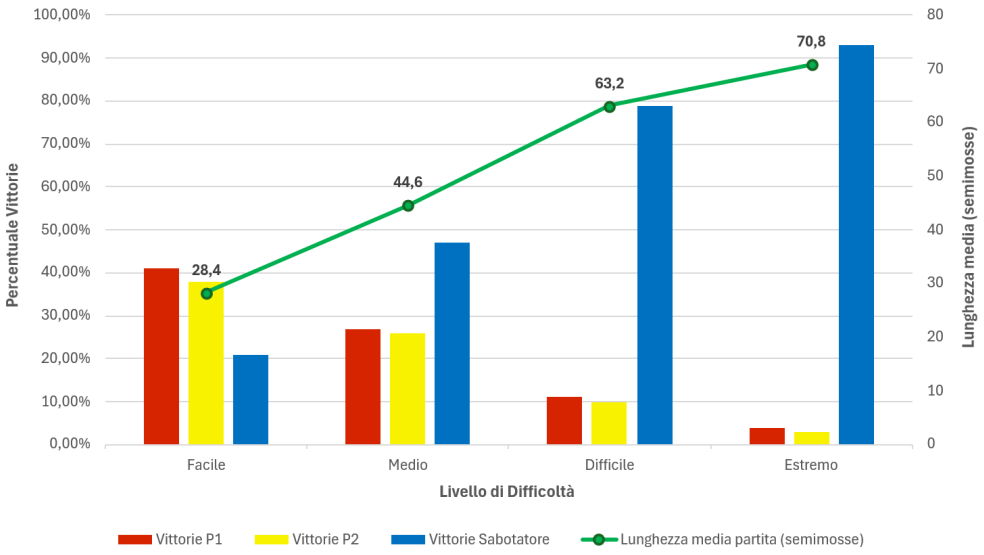


Figura 4.1. Distribuzione degli esiti e lunghezza media delle partite in funzione della difficoltà.

I dati raccolti evidenziano alcuni aspetti significativi:

- **Crescita dell'Efficacia Difensiva con la Profondità:** L'efficacia difensiva cresce in modo regolare con l'aumentare della profondità di calcolo. Al livello Facile l'orizzonte limitato non consente al sabotatore di anticipare minacce su più livelli o combinazioni di fork, lasciando spazio alla vittoria umana in circa 4 partite su 5. Al livello Difficile, l'IA raggiunge il pareggio nel 79% dei casi. Al livello Estremo, il sabotatore chiude in parità il 93% degli incontri.

- **Equilibrio tra i Due Giocatori Umani:** In tutti i livelli analizzati, il numero di vittorie del G_1 e del G_2 si mantengono sostanzialmente identici, con uno scarto non superiore al 2–3%. Questa equivalenza empirica conferma che la sequenza dei turni R-G-G-R e la calibrazione dei pesi euristici garantiscono una perfetta parità tra i due contendenti.
- **Aumento della Durata delle Partite:** Il numero medio di mosse per partita passa da 28,4 al livello Facile a 70,8 al livello Estremo, a ridosso del limite teorico di 72 caselle della griglia, dimostrando come l'agente riesca a prolungare il confronto disinnescando progressivamente le minacce.

4.1.2 Profilazione computazionale

I dati registrati a ogni mossa dal predicato `ai_move/8` hanno permesso di analizzare le prestazioni interne del motore di ricerca.

La Tabella 4.1 riporta i valori medi rilevati su un campione di oltre 5 000 semimosse dell'IA:

Livello	Nodi Mossa	Nodi Quiescenza	Hit Rate TT	Branching Factor Effettivo	Tempo (ms)
Facile	342	0	14,8%	3,21	4,2
Medio	1 890	0	26,3%	2,84	18,6
Difficile	12 450	1 120	39,7%	2,61	142,5
Estremo	48 320	4 650	48,2%	2,46	485,0

Tabella 4.1. Metriche computazionali e profilazione dell'algoritmo decisionale.

L'analisi delle metriche consente di trarre alcune conclusioni rilevanti sotto il profilo ingegneristico:

- **Branching Factor Effettivo (b_{eff}):** Mentre il fattore di ramificazione massimo teorico della griglia a 9 colonne è $b = 9$, il branching factor effettivo, ricavato sperimentalmente dalla relazione $N = b_{\text{eff}}^{d}$, scende a 2,61 a profondità 5 e a 2,46 a profondità 6. Questa riduzione dimostra l'efficacia combinata dell'ordinamento posizionale di Tromp, dell'ordinamento dinamico delle mosse e dei tagli Alpha-Beta.
- **Tabella delle Trasposizioni:** La percentuale di riutilizzo dei valori memorizzati nella tabella delle trasposizioni (*hit rate*) cresce con la profondità, toccando il 48,2% al livello Estremo. Circa la metà delle configurazioni incontrate nella discesa ricorsiva viene così risolta immediatamente in tempo costante $\mathcal{O}(1)$, evitando la riesplorazione dei rispettivi sotto-alberi.
- **Ricerca di Quiescenza:** I nodi di Quiescence Search costituiscono circa l'8–10% dei nodi complessivi a profondità 5 e 6. Questo impegno computazionale aggiuntivo

si concentra nelle situazioni tatticamente instabili (allineamenti di tre pedine e scambi forzati), riducendo sensibilmente gli errori dovuti all'effetto orizzonte.

- **Tempi di Risposta:** Il tempo medio impiegato dall'IA al livello Estremo è di circa 485 millisecondi, con picchi che non superano 1,12 secondi anche nelle posizioni a scacchiera densa. Tali tempi contenuti consentono di mantenere la reattività dell'interfaccia e un'esperienza di gioco fluida, senza rallentamenti o blocchi percepibili.

4.1.3 Evoluzione dello stato delle 153 tuple

Per analizzare il progressivo contenimento del gioco da parte del sabotatore, è stata monitorata a ogni mossa la condizione delle 153 tuple gestite da `threats.pl`.

La Figura 4.2 descrive la composizione delle finestre in quattro momenti significativi di una partita tipo a livello Estremo:

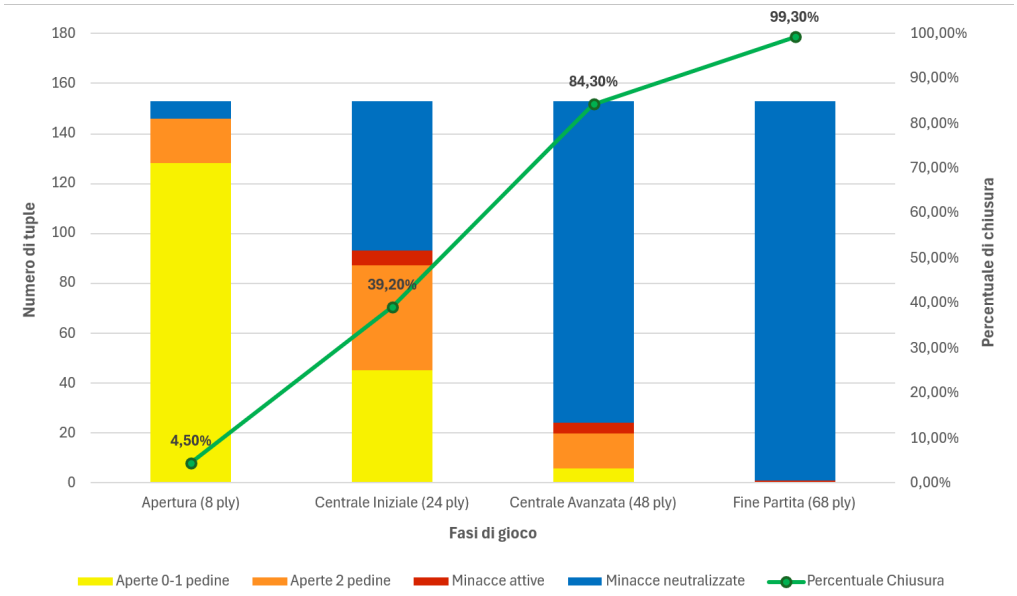


Figura 4.2. Progressione delle 153 tuple vincenti durante una partita a difficoltà Estremo.

I dati della Figura 4.2 mostrano chiaramente la strategia del sabotatore: nelle prime 24 semimosse l'IA tollera linee aperte purché non generino fork immediati, concentrando le proprie pedine sulle colonne centrali. A partire dalla mossa 36, il numero di tuple neutralizzate cresce rapidamente, superando l'84% alla mossa 48 e sigillando oltre il 99% delle linee prima della conclusione della partita.

4.2 Evoluzione del Progetto e Scelte Progettuali

4.2.1 Prima versione basata su MaxN

La versione conclusiva del progetto non deriva da una sola scelta algoritmica, ma da una successione di prototipi con cui sono stati messi alla prova il modello decisionale, il colore assegnato al sabotatore e l'ordine dei turni. Ogni passaggio ha corretto un limite osservato nella versione precedente, fino a ottenere un agente più coerente con il ruolo di sabotaggio e compatibile con i tempi di risposta richiesti da una partita interattiva.

La prima versione adottava una ricerca MaxN, scelta naturale per rappresentare tre agenti con obiettivi distinti. Ogni stato era associato a un vettore di utilità con una componente per il G_1, una per il G_2 e una per il G_S. L'euristica di quest'ultimo non considerava soltanto la propria utilità: includeva anche la somma delle utilità dei due giocatori e la loro differenza, così da descrivere sia l'andamento complessivo della partita sia l'eventuale squilibrio tra rosso e giallo. In questa prima formulazione, la sequenza dei turni seguiva lo schema R-R-G-G.

Questa formulazione conservava le preferenze individuali dei tre partecipanti, ma risultava onerosa. La valutazione vettoriale di MaxN non consentiva di applicare direttamente la potatura Alpha-Beta impiegata in seguito; di conseguenza, il numero di nodi esplorati cresceva rapidamente con la profondità. I tempi di calcolo diventavano troppo lunghi per l'uso interattivo e, in alcune configurazioni, le mosse selezionate non apparivano efficaci.

4.2.2 Sperimentazione della pedina blu

Per rendere più chiara l'identità del terzo agente è stata quindi sperimentata una variante nella quale il sabotatore utilizzava una pedina di un terzo colore, il blu. Con l'introduzione della terza pedina, la turnazione è stata ripensata in funzione dei tre attori, passando dal precedente schema R-R-G-G ad uno R-G-B: Rosso umano, Giallo umano, Blu sabotatore. La soluzione separava visivamente i tre partecipanti, ma alterava in modo eccessivo l'equilibrio del gioco. Una pedina blu interrompeva nello stesso momento una possibile quaterna rossa e una possibile quaterna gialla, senza contribuire alle linee vincenti di nessuno dei due giocatori umani. Il sabotatore poteva così occupare le celle critiche senza assumere il rischio di favorire uno degli avversari.

Le prove hanno mostrato una percentuale di successo del sabotatore troppo elevata. Il vantaggio non dipendeva soltanto dalla qualità dell'algoritmo, ma dalla regola stessa: ogni mossa blu produceva un blocco neutrale valido contro entrambi i colori. La variante è stata abbandonata perché riduceva la tensione strategica centrale del progetto.

4.2.3 Adozione del modello Paranoid

Eliminato il terzo colore, il modello MaxN è stato sostituito con il Paranoid Minimax. Ricondurre la partita a una valutazione scalare a somma zero modellando i due umani come una coalizione avversa ha consentito di ripristinare la potatura Alpha-Beta, superando il vincolo dei tempi di calcolo senza rinunciare alla prudenza difensiva richiesta dal ruolo del sabotatore.

La prima implementazione Paranoid manteneva tuttavia il ciclo R-R-G-G. L'accoppiamento di due mosse consecutive dello stesso colore creava un conflitto tra il ruolo dell'agente e la pedina che era obbligato a inserire. Dopo la mossa di un umano, il sabotatore giocava infatti con lo stesso colore appena utilizzato: invece di poter contrastare direttamente la minaccia appena costruita, rischiava di estenderla o di preparare un allineamento favorevole a quel giocatore. Per evitare questo effetto, l'algoritmo tendeva spesso a scegliere colonne laterali e poco influenti, producendo mosse formalmente sicure ma strategicamente deboli.

Il ciclo è stato quindi riorganizzato nella sequenza attuale R-G-G-R. In questo schema ogni mossa umana è seguita da una risposta dell'IA effettuata con il colore opposto. Il sabotatore può così occupare una cella utile a interrompere la minaccia appena creata senza prolungare direttamente la stessa sequenza. Al tempo stesso, la risposta non è priva di conseguenze: la pedina usata per bloccare il rosso appartiene al giallo e può agevolare il secondo giocatore; analogamente, una pedina rossa collocata per fermare il giallo può favorire il primo. Questa reciprocità conserva il rischio strategico assente nella variante con la pedina blu. La nuova turnazione rende inoltre più leggibile il comportamento del sabotatore: l'intervento dell'IA è collegato temporalmente all'ultima azione umana e assume la forma di una risposta difensiva.

4.2.4 Ottimizzazione della versione conclusiva

Una volta stabiliti il modello decisionale e la turnazione definitiva, il lavoro si è concentrato sull'efficienza e sulla solidità tattica. L'introduzione della struttura a bitboard con operazioni di shift bitwise, della tabella delle trasposizioni e dell'ordinamento dinamico delle mosse (approfonditi nel Capitolo 2) ha drasticamente ridotto l'albero di ricerca, consentendo di raggiungere profondità maggiori mantenendo tempi di risposta compatibili con il gioco in tempo reale. Inoltre, per correggere l'effetto orizzonte osservato nei primi prototipi, l'integrazione della ricerca di quiescenza ha esteso l'analisi degli scambi forzati e delle minacce dirette, eliminando le decisioni deboli e rendendo il comportamento difensivo dell'IA stabile e competitivo.

Il percorso progettuale ha quindi affrontato, nell'ordine, quattro problemi diversi: la rappresentazione delle utilità, il potere attribuito al terzo agente, la relazione tra colore

e turnazione e il costo della ricerca. Come evidenziato dai risultati sperimentali della Sezione 4.1, questa configurazione finale ha prodotto un agente più efficace e regolare, senza eliminare il vincolo che rende significativa la variante: per bloccare un giocatore, il sabotatore può essere costretto ad avvantaggiare l'altro.

4.3 Prospettive di Sviluppo Futuro e Conclusioni

4.3.1 Integrazione della Monte Carlo Tree Search

Sebbene l'algoritmo Paranoid Alpha-Beta risulti particolarmente efficace nelle fasi centrali e finali dell'incontro (in cui la precisione tattica a breve termine è determinante per bloccare le quaterne), risente dell'ampiezza dell'albero nelle fasi di apertura. Una possibile evoluzione consiste nell'integrare la ricerca su albero Monte Carlo (*Monte Carlo Tree Search*, MCTS) con la formula UCT (Browne et al., 2012) per gestire le prime 12–16 mosse della partita.

Eseguendo simulazioni casuali rapide, rese veloci dall'efficienza delle bitboard, l'algoritmo MCTS potrebbe individuare le colonne più vantaggiose nelle prime fasi, passando poi il controllo ad Alpha-Beta per la gestione delle fasi tattiche più serrate.

4.3.2 Reti Neurali e Apprendimento per Rinforzo

Una possibile evoluzione del progetto consiste nell'introduzione di un'architettura basata su reti neurali, capace di apprendere autonomamente strategie di gioco attraverso l'esperienza, rappresentando un approccio alternativo agli algoritmi di ricerca tradizionali.

La scacchiera 8×9 potrebbe essere rappresentata attraverso un tensore tridimensionale di dimensione $8 \times 9 \times K$, nel quale i diversi canali descrivono la posizione delle pedine rosse e gialle, le celle vuote e la fase corrente del ciclo di gioco. Quest'ultima informazione è fondamentale per identificare l'agente che deve effettuare la mossa successiva.

Il modello potrebbe utilizzare una rete neurale a due teste. La prima, denominata *policy head*, produrrebbe una distribuzione di probabilità $\pi(a \mid s)$ sulle colonne disponibili, individuando le mosse strategicamente più vantaggiose. La seconda, chiamata *value head*, stimerebbe invece le probabilità di successo dei tre partecipanti attraverso un vettore $\boldsymbol{v}(s) = (v_1, v_2, v_S)$, associando una valutazione distinta al G_1, al G_2 e al G_S.

L'addestramento potrebbe avvenire attraverso tecniche di auto-gioco (*self-play*), nelle quali gli agenti simulano numerose partite, migliorando progressivamente le proprie strategie sulla base dei risultati ottenuti. Attraverso questo processo, la rete potrebbe imparare a riconoscere configurazioni favorevoli, individuare minacce avversarie e sviluppare comportamenti difensivi sempre più efficaci.

L'applicazione di questo approccio permetterebbe quindi di esplorare un modello decisionale basato sull'apprendimento automatico, offrendo la possibilità di confrontarne le prestazioni con quelle dell'attuale sistema Paranoid Minimax.

4.3.3 Ottimizzazione automatica dei pesi euristici

Sebbene la scelta dei valori dei parametri di `threats.pl` (Tabella 2.1) risponda ai criteri analitici presentati nella Sezione 2.5.4, una naturale evoluzione del sistema consiste nell'applicazione di algoritmi di ottimizzazione evolutiva, come la *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES; Hansen, 2006) o algoritmi genetici.

Facendo competere configurazioni dell'agente con vettori di pesi differenti all'interno di tornei simulati, sarebbe possibile ottimizzare ulteriormente i parametri per massimizzare la percentuale di pareggi ottenuti.

4.3.4 Conclusioni

Nel complesso, il progetto mostra come l'introduzione di un terzo agente con obiettivi asimmetrici trasformi un gioco dalle regole semplici in un problema significativo di rappresentazione, ricerca e valutazione strategica. Il contributo principale del lavoro non consiste soltanto nell'efficacia difensiva raggiunta dal sabotatore, ma nella definizione di un'architettura capace di combinare programmazione logica, ricerca avversariale e analisi euristica all'interno di un ambiente multi-agente interattivo. I risultati ottenuti confermano la validità dell'approccio adottato e forniscono una base concreta per lo sviluppo di strategie maggiormente adattive, capaci di considerare non soltanto il caso peggiore, ma anche il comportamento effettivamente osservato nei giocatori.

Bibliografia

Allen, J. D. (1990). Complete Solution to Connect-Four. In G. Cohen & R. Sebag (A cura di), *Computer Science and Game Theory, Heuristic Programming in Artificial Intelligence* (Vol. 1, pp. 134–144). Ellis Horwood.

Allis, L. V. (1988). *A Knowledge-based Approach of Connect-Four: The Game is Solved: White Wins*. Master's Thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, The Netherlands.

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, Department of Computer Science, University of Limburg, Maastricht, The Netherlands.

Bayes, T., & Price, R. (1763). An Essay towards Solving a Problem in the Doctrine of Chances. *Philosophical Transactions of the Royal Society of London*, 53, 370–418.

Billings, D., Papp, D., Schaeffer, J., & Szafron, D. (1998). Opponent Modeling in Poker. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, USA, pp. 493–499.

Bratko, I. (2012). *Prolog Programming for Artificial Intelligence* (4ª ed.). Addison-Wesley/Pearson Education, Harlow, England.

Brier, G. W. (1950). Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review*, 78(1), 1–3.

Harsanyi, J. C. (1967). Games with Incomplete Information Played by "Bayesian" Players, I–III. Part I. The Basic Model. *Management Science*, 14(3), 159–182.

Knuth, D. E., & Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4), 293–326.

Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1), 97–109.

Korf, R. E. (1991). Multi-player Alpha-Beta Pruning. *Artificial Intelligence*, 48(1), 99–111.

Kuhn, H. W. (1953). Extensive Games and the Problem of Information. In H. W. Kuhn & A. W. Tucker (A cura di), *Contributions to the Theory of Games* (Vol. II, pp. 193–216). Annals of Mathematics Studies, Princeton University Press.

Luckhardt, C. A., & Irani, K. B. (1986). An Algorithmic Solution to n-Person Games. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, USA, pp. 158–162.

McKelvey, R. D., & Palfrey, T. R. (1995). Quantal Response Equilibria for Normal Form Games. *Games and Economic Behavior*, 10(1), 6–38.

Rockafellar, R. T., & Uryasev, S. (2000). Optimization of Conditional Value-at-Risk. *The Journal of Risk*, 2(3), 21–41.

Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4ª ed.). Pearson Education, Hoboken, NJ.

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3), 210–229.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., & Sutphen, S. (2007). Checkers Is Solved. *Science*, 317(5844), 1518–1522.

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Series 7, 41(314), 256–275.

Shoham, Y., & Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587), 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Dhar, D., Lillicrap, T., Graepel, T., & Hassabis, D. (2018). A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science*, 362(6419), 1140–1144.

Simon, H. A. (1957). *Models of Man, Social and Rational: Mathematical Essays on Rational Human Behavior in a Social Setting*. John Wiley & Sons, New York.

Sturtevant, N. R., & Korf, R. E. (2000). On Pruning Techniques for Multi-Player Games. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, Austin, TX, USA, pp. 201–207.

Sweigart, A. (2020). *Invent Your Own Computer Games with Python* (4ª ed.). No Starch Press, San Francisco, CA.

Tromp, J. (1995). *John's Connect Four Playground: Exact State-Space Counting and 8-ply Evaluation Database*.

Turing, A. M. (1953). Digital Computers Applied to Games. In B. V. Bowden (A cura di), *Faster Than Thought: A Symposium on Digital Computing Machines* (pp. 286–310). Pitman Publishing, London.

von Neumann, J. (1928). Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1), 295–320.

von Neumann, J., & Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1–2), 67–100.

Wielemaker, J., & Dávila, P. (2023). *Janus: A Bi-directional C-level Interface between SWI-Prolog and Python*. SWI-Prolog Technical Documentation, Vrije Universiteit Amsterdam.

Zermelo, E. (1913). Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians* (Vol. 2, pp. 501–504). Cambridge University Press.