



UNIVERSITÀ DEGLI STUDI
DI SALERNO

FONDAMENTI DI INTELLIGENZA ARTIFICIALE

Connect4IA

Luca Del Bue - 0512116173

Salvatore Di Martino - 0512116932

Anno Accademico 2024 - 2025

Link GitHub: <https://github.com/saladm04/Connect4IA.git>

Indice

Introduzione	2
1 Analisi del Sistema	3
1.1 Obiettivo	3
1.2 Specifica PEAS dell'ambiente	3
1.2.1 Caratteristiche dell'ambiente	4
1.3 Analisi del problema	4
2 Soluzioni del problema	5
2.1 Tecnologie utilizzate	5
2.2 Prima soluzione: minimax puro	6
2.3 Seconda soluzione: minimax con potatura alfa-beta	7
2.4 Terza soluzione: decisioni imperfette in tempo reale	8
2.5 Miglioramenti: ordinamento dinamico delle mosse, approfondi- mento iterativo e potatura in avanti	9
2.6 Definizione dei livelli di difficoltà	11

Introduzione

Connect 4, noto anche come Forza 4, è un popolare gioco da tavolo strategico per due giocatori, che si gioca su una griglia verticale di sette colonne e sei righe. L'obiettivo del gioco è semplice ma coinvolgente: allineare quattro pedine del proprio colore in una fila continua, che può essere orizzontale, verticale o diagonale, prima dell'avversario. Inventato negli anni '70, Connect 4 è stato ufficialmente introdotto sul mercato da Milton Bradley, oggi parte del marchio Hasbro, nel 1974, e ha rapidamente conquistato un pubblico vasto e intergenerazionale grazie alle sue regole facili da comprendere e alla sua profondità strategica.

La meccanica del gioco consiste nell'inserire una pedina in una delle colonne, dove cadrà fino a raggiungere la posizione più bassa libera, creando combinazioni e bloccando le mosse dell'avversario. A differenza di giochi simili come il tris, Connect 4 si distingue per l'elemento gravitazionale che limita le mosse possibili e aggiunge complessità strategica, poiché i giocatori devono anticipare sia le proprie mosse sia quelle dell'altro per impedire eventuali vittorie dell'avversario.



Connect 4 nella sua versione classica

1

Analisi del Sistema

1.1 Obiettivo

L'obiettivo principale è quello di sviluppare un'applicazione desktop in python che permetta di giocare al gioco Connect Four contro un agente di intelligenza artificiale.

Prima di iniziare una partita, l'applicazione permette all'utente di selezionare un livello di difficoltà, che va a modificare i parametri dell'algoritmo, influenzando le performance dell'agente.

1.2 Specifica PEAS dell'ambiente

L'ambiente in cui l'agente opera viene descritto dalla specifica **PEAS**:

- **Performance:** La misura di prestazione adottata prevede la minimizzazione dei tempi di ricerca della miglior mossa possibile in base ai parametri impostati dalla difficoltà selezionata.
- **Enviroment:** L'ambiente in cui opera l'agente è costituito da tutte le possibili combinazioni della griglia di gioco.
- **Actuators:** L'agente agisce sull'ambiente eseguendo la miglior mossa calcolata.
- **Sensors:** L'agente riceve le percezioni tramite la griglia che rappresenta lo stato del gioco.

1.2.1 Caratteristiche dell'ambiente

L'ambiente è caratterizzato dalle seguenti proprietà:

- **Completamente osservabile:** L'agente ha accesso a tutte le informazioni rilevanti sull'ambiente.
- **Deterministico:** Le azioni dell'agente determinano completamente lo stato successivo dell'ambiente.
- **Sequenziale:** Ogni azione dell'agente influenza gli stati futuri.
- **Statico:** L'ambiente rimane invariato durante le decisioni dell'agente.
- **Discreto:** Lo spazio degli stati e quello delle azioni sono finiti.
- **Multi-Agente:** Nell'ambiente operano due agenti.

1.3 Analisi del problema

Nel contesto del gioco Connect Four, l'interazione avviene tra due agenti: uno rappresentato dal giocatore umano e l'altro dall'agente intelligente. A differenza dei problemi di ricerca tradizionali, in cui un singolo agente cerca di raggiungere un obiettivo senza opposizione, questo scenario richiede decisioni strategiche che tengano conto delle possibili mosse dell'avversario. Connect Four è un gioco deterministico con informazione perfetta e a somma zero, dove i valori di utilità, alla fine partita, sono sempre uguali ma di segno opposto.

Un approccio ampiamente utilizzato per affrontare problemi di questo tipo è l'algoritmo minimax. Questo metodo prevede l'esplorazione completa dell'albero di gioco, che rappresenta tutti i possibili stati raggiungibili durante la partita. L'obiettivo principale è determinare una sequenza di mosse che massimizzi il punteggio per un agente, assumendo che l'avversario giochi in modo ottimale, cioè adottando sempre la strategia che minimizza il punteggio del primo agente.

L'algoritmo valuta i nodi terminali dell'albero di gioco attribuendo loro un punteggio in base all'esito della partita. Questi valori vengono poi propagati verso l'alto nell'albero, consentendo di identificare la mossa iniziale che garantisce il miglior risultato possibile nel caso peggiore. Tale analisi presuppone che l'avversario adotti un comportamento infallibile, riflettendo quindi una visione pessimistica ma robusta per la presa di decisioni ottimali.

2

Soluzioni del problema

Per consentire all'utente di sfidare l'agente intelligente, verrà implementata un'interfaccia grafica che mostrerà la griglia 6x7 per l'inserimento delle pedine. Prima di iniziare la partita, l'utente avrà la possibilità di inserire il proprio nome e di selezionare la difficoltà con cui andrà a sfidare l'agente.

2.1 Tecnologie utilizzate

Il gioco verrà implementato in *python*, sfruttando la libreria *pygame* per la gestione dell'interfaccia grafica e degli input dell'utente.

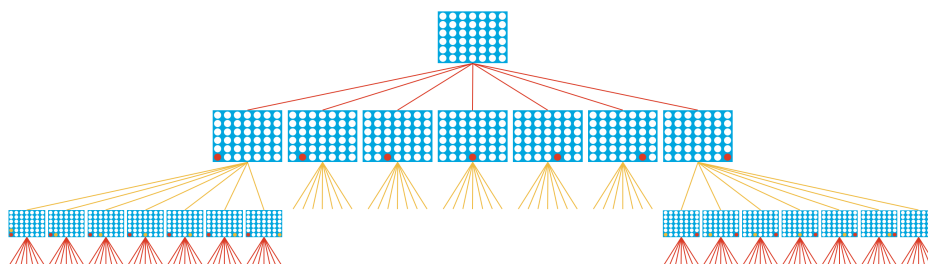
Il progetto verrà strutturato nel seguente modo:

- **board**: modulo dedicato alla gestione della griglia di gioco, contenente costanti e funzioni specifiche.
- **algorithms**: directory che include le implementazioni degli algoritmi di intelligenza artificiale utilizzati nel gioco.
- **states**: directory che raccoglie i moduli per rappresentare i diversi stati del gioco. Ogni modulo integra sia la logica che l'interfaccia grafica relativa allo stato specifico.
- **main**: modulo principale responsabile dell'avvio del gioco.
- **game_assets**: directory contenente gli elementi grafici utilizzati per costruire l'interfaccia utente.
- **utils**: modulo che fornisce funzioni ausiliarie per la gestione dell'interfaccia grafica e per la selezione del livello di difficoltà.
- **tests**: directory che contiene il modulo per il test.

2.2 Prima soluzione: minimax puro

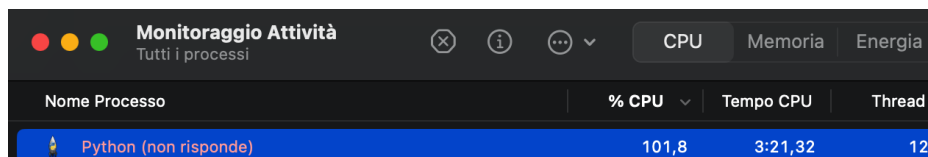
La prima soluzione adottata è il minimax puro. L'algoritmo viene definito "puro" perché non considera né funzione euristica, né viene impostata una profondità massima. Quindi l'algoritmo esplora l'intero albero di gioco, calcolando per ogni nodo il valore minimax. Questo valore corrisponde all'utilità di trovarsi nello stato corrispondente, assumendo che entrambi gli agenti giochino in modo ottimo da lì alla fine della partita.

L'algoritmo è stato implementato nel modulo *minimax.py* all'interno della directory *algorithms*. Nel modulo è definito il metodo *pure_minimax* che simula tutte le possibili sequenze di gioco fino alla conclusione, valutando se la mossa porta ad una vittoria dell'agente intelligente (+10000), dell'utente (-10000) o un pareggio (0). Il metodo *find_best_move* utilizza *pure_minimax* per simulare ogni mossa iniziale e scegliere quella con il punteggio migliore, garantendo decisioni ottimali per l'agente.



Esempio di albero di gioco per Connect Four

L'algoritmo così definito è però inefficiente per via della sua complessità temporale, determinata dalla dimensione dell'albero di gioco. Come mostrato nell'immagine seguente, il calcolo richiede troppo tempo, rendendo impossibile giocare in tempo reale e bloccando il computer durante l'elaborazione.



Esecuzione di minimax puro

Per risolvere il problema, si potrebbe pensare di applicare la potatura alfa-beta all'algoritmo minimax per ridurre il numero di nodi da esplorare.

2.3 Seconda soluzione: minimax con potatura alfa-beta

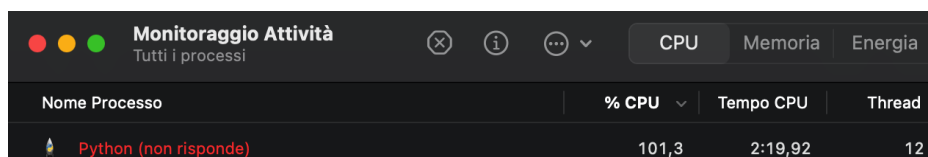
La potatura alfa-beta ottimizza l'algoritmo minimax riducendo i nodi da esplorare, senza alterare il risultato finale. Elimina i rami dell'albero di gioco che non influenzeranno la decisione, utilizzando due parametri:

- Alfa, aggiornato nel turno dell'agente intelligente, rappresenta il miglior valore che può garantire.
- Beta, aggiornato nel turno dell'utente, rappresenta il miglior valore che può garantire.

Il confronto tra alfa e beta durante l'esplorazione guida l'eliminazione dei rami superflui. Ogni volta che si scopre che il valore di un ramo non migliorerà oltre i limiti di alfa o beta, quel ramo può essere eliminato, poiché non influenzerà il risultato finale. Questo consente di ridurre significativamente il numero di nodi da esplorare.

L'algoritmo è stato implementato nel modulo *minimax_alpha_beta.py* all'interno della directory *algorithms*. Nel modulo è definito il metodo *minimax_alpha_beta*, che esegue le stesse operazioni del metodo *pure_minimax*, ottimizzando il processo con la potatura alfa-beta per ridurre il numero di stati esplorati. Il metodo *find_best_move* resta perlopiù invariato, inizializzando i valori di alfa e beta a $-\infty$ e ∞ .

Purtroppo, anche in questo caso, l'algoritmo risulta essere impraticabile nonostante la potatura. Infatti, essendo l'albero di gioco eccessivamente ampio, la potatura da sola non riesce a diminuire il numero di nodi da esplorare. Non ci resta che introdurre una profondità massima nell'esplorazione dell'albero, che permette di tagliare la ricerca ad un certo punto, e una funzione di valutazione euristica che fornisce la stima dell'utilità.



Monitoraggio Attività Tutti i processi			
CPU Memoria Energia			
Nome Processo	% CPU	Tempo CPU	Thread
Python (non risponde)	101,3	2:19,92	12

Esecuzione di minimax con potatura alfa-beta

2.4 Terza soluzione: decisioni imperfette in tempo reale

L'algoritmo minimax genera l'intero spazio di ricerca, mentre l'alfa-beta ne pota una parte significativa. Tuttavia, anche alfa-beta deve esplorare fino agli stati terminali, almeno in una parte dello spazio, il che è spesso impraticabile perché le mosse devono essere calcolate in tempi ragionevoli. Impostando una profondità massima, si interrompe la ricerca prima delle foglie e si utilizza una funzione di valutazione euristica per stimare il guadagno atteso in una posizione.

La funzione di valutazione dovrebbe essere veloce da calcolare e avere una forte correlazione con la probabilità reale di vincere la partita, basandosi sulle caratteristiche di uno stato. Nel nostro caso, la funzione *score_position* si basa sul numero di pedine dello stesso colore presenti all'interno di tutte le possibili "finestre" di 4 celle consecutive contenute nella griglia. Ad esempio:

- Se la finestra contiene 4 pezzi del giocatore: +10000
- Se la finestra contiene 3 pezzi del giocatore e 1 cella vuota: +100
- Se la finestra contiene 2 pezzi del giocatore e 2 celle vuote: +50
- Se la finestra contiene 3 pezzi dell'avversario e 1 cella vuota: -100
- Se la finestra contiene 2 pezzi dell'avversario e 2 celle vuote: -45

Inoltre, la funzione euristica premia il posizionamento delle pedine verso il centro della griglia, aumentando le opportunità di combinazioni vincenti. Ad esempio, i coefficienti per le sette colonne possono essere definiti come: [3,4,6,7,6,4,3]. La profondità di "taglio" viene fornita come parametro al metodo *find_best_move* in modo da poter personalizzare il comportamento e l'efficienza dell'algoritmo.

Così facendo, l'agente risponde in tempi ragionevoli. Infatti, modificando la profondità massima si può ottenere un giusto compromesso tra efficienza e numero di stati esplorati. Test effettuati con Intel i7-1165G7 @ 2.80GHz.

Profondità massima	4	5	6	7
Tempo medio di risposta	0.245	1.039	4.542	14.706

Tempi medi in secondi, calcolati su 7 esecuzioni iniziali, una per colonna

2.5 Miglioramenti: ordinamento dinamico delle mosse, approfondimento iterativo e potatura in avanti

Queste tecniche permettono di migliorare ulteriormente l'efficienza dell'algoritmo riducendo il numero di nodi da visitare nell'albero di gioco. Inoltre è possibile sfruttare la parametrizzazione dei valori per modificare il comportamento dell'algoritmo.

L'efficacia della potatura alfa-beta dipende fortemente dall'ordinamento in cui gli stati sono esaminati. L'idea è quindi quella di ordinare le mosse più promettenti all'inizio del processo di esplorazione favorendo un numero maggiore di potature precoci. L'ordinamento viene definito in base alla funzione euristica *score_position* descritta precedentemente.

L'approfondimento iterativo consente di esplorare l'albero di gioco a profondità crescente, fornendo una mossa valida anche se non è stato completato l'intero processo di ricerca. Se viene raggiunto il limite di tempo, l'algoritmo restituisce la mossa migliore trovata fino a quel momento. Per garantire una risposta rapida, il limite di tempo è stato impostato ad un secondo.

La potatura in avanti consiste nell'esplorare solo determinate mosse ritenute particolarmente promettenti, tagliando le altre. L'approccio adottato si basa sulla beam search, che considera solo le prime k migliori mosse in base alla funzione di valutazione euristica.

Nella seguente tabella sono riportate le medie dei tempi di esecuzione iniziali dell'algoritmo al variare della profondità massima e del parametro k della beam search. Per mostrare l'effetto dei miglioramenti introdotti, non è stato considerato il limite di tempo dell'approfondimento iterativo (1 sec).

Profondità max Parametro k	4	5	6	8	10
4	0.130	0.327	0.769	3.997	16.77
5	0.198	0.520	1.655	9.810	44.41
6	0.278	0.761	2.758	15.397	>60
7	0.376	1.174	4.271	26.211	>60

Misurazioni effettuate con CPU Intel i7-1165G7 @ 2.80GHz

Si osserva come il parametro k influenzi significativamente i tempi di esecuzione, permettendo di scartare mosse meno rilevanti. L'ultima riga della tabella riporta i tempi dell'algoritmo senza potatura in avanti, che considera tutti e sette gli stati generati dalle possibili mosse. In questo caso, i tempi risultano leggermente superiori rispetto alla versione precedente dell'algoritmo, a causa dell'overhead introdotto dalle nuove funzionalità implementate.

L'ottimizzazione diventa evidente quando si imposta un valore di k inferiore a sette. Questo approccio consente di ridurre i tempi di calcolo mantenendo una profondità maggiore rispetto a quella considerata accettabile nella versione precedente. In pratica, la combinazione di un k più basso e una maggiore profondità migliora l'efficienza complessiva dell'algoritmo.

Infine, i parametri k e la profondità massima, insieme ai punteggi della funzione di valutazione euristica, possono essere modificati per simulare diversi livelli di difficoltà dell'agente. Ad esempio, valori più bassi di k rendono l'agente più conservativo, mentre valori più alti ne aumentano l'aggressività e la complessità strategica.

2.6 Definizione dei livelli di difficoltà

I parametri utilizzati per definire i tre livelli di difficoltà (facile, medio e difficile) sono:

- **Profondità massima:** livello massimo di esplorazione dell'albero.
- **Larghezza k della beam search:** ad ogni livello di ricerca vengono considerate solo le k mosse con il miglior punteggio, basato sull'euristica.
- **Pesi dell'euristica:** valori che stimano l'utilità di uno stato per prevedere il guadagno atteso.
- **Limite di tempo:** tempo massimo per ogni iterazione di ricerca.
- **Moltiplicatori delle mosse centrali:** coefficienti che favoriscono il posizionamento delle pedine nelle colonne centrali.

Questi parametri sono definiti nel modulo *utils.py*, che associa ogni livello di difficoltà a un set di valori tramite un dizionario. Il comportamento dell'algoritmo è personalizzato passando i parametri al metodo *find_best_move*.

Sono stati progettati tre agenti intelligenti per rappresentare i diversi livelli di difficoltà:

- A, facile
- B, media
- C, difficile

Per garantire una separazione coerente tra le complessità, gli algoritmi sono configurati in modo che:

1. A perda contro B
2. B perda contro C
3. A perda contro C

Un modulo dedicato, *difficulty_test.py*, contenuto nella directory *tests*, consente di testare e ottimizzare rapidamente queste configurazioni. Esso simula un determinato numero di partite tra agenti con difficoltà diverse, raccogliendo dati come le mosse effettuate e il loro numero medio, le vittorie (o pareggi), le scacchiere risultanti e i tempi di risposta. Per rendere la simulazione più realistica, il primo giocatore e la mossa iniziale vengono selezionati casualmente, per imitare il comportamento di un giocatore umano e per verificare quanto la vittoria di un agente è influenzata dall'iniziativa di gioco.

Dopo un'attenta analisi empirica, sono stati scelti i seguenti parametri:

Parametro	Media	Difficile
Profondità massima	5	7
Valore k beam search	4	6
Pesi euristica	10000	10000
	100	200
	50	100
	100	210
	45	100
Limite di tempo	1.0	1.2
Coefficienti mosse centrali	[3,4,5,5,5,4,3]	[6,8,10,14,10,8,6]

I pesi dell'euristica indicano, in sequenza, il valore di una finestra contenente:

- 4 pezzi del giocatore
- 3 pezzi del giocatore e 1 cella vuota
- 2 pezzi del giocatore e 2 celle vuote
- 3 pezzi dell'avversario e 1 cella vuota
- 2 pezzi dell'avversario e 2 celle vuote

L'agente più difficile da battere (C) adotta una strategia avanzata grazie a una profondità di ricerca maggiore e a un valore di k più elevato, che gli permette di considerare un numero maggiore di mosse. I pesi delle euristiche sono configurati per prediligere uno stile di gioco difensivo, con un limite di tempo leggermente superiore che favorisce analisi più approfondite. Inoltre, premia fortemente il posizionamento delle pedine nelle colonne centrali, rendendo la sua strategia più efficace.

L'agente di difficoltà media (B) bilancia attacco e difesa, con valori complessivi più moderati rispetto all'agente C.

L'agente più semplice da battere (A) parte con parametri simili a quelli dell'agente B, ma i suoi valori diminuiscono drasticamente man mano che aumenta il numero di mosse effettuate. Questo lo rende sempre più facile da sconfiggere nel corso della partita.

Numero di mosse	1 - 4	5	6	7	8 o più
Profondità massima	5	4	3	3	2
Valore k beam search	4	2	4	2	2
Pesi euristica	10000	10000	100	100	100
	100	100	25	25	25
	50	50	10	10	10
	100	100	10	5	0
	45	45	0	0	0
Limite di tempo	1.0				
Coeff. mosse centrali	[3, 4, 5, 5, 5, 4, 3]				

Di seguito sono riportati i risultati delle simulazioni su cento partite:

<p>Test Completato!</p> <p>Numero di Partite: 100 Vittorie AI1 (Livello 1): 5 Vittorie AI2 (Livello 2): 81 Pareggi: 14</p> <p>Tempo Medio di Risposta AI1: 0.0477 secondi per mossa Tempo Medio di Risposta AI2: 0.1184 secondi per mossa</p>	<p>Test Completato!</p> <p>Numero di Partite: 100 Vittorie AI1 (Livello 2): 22 Vittorie AI2 (Livello 3): 69 Pareggi: 9</p> <p>Tempo Medio di Risposta AI1: 0.1133 secondi per mossa Tempo Medio di Risposta AI2: 0.9014 secondi per mossa</p>
<p>Test Completato!</p> <p>Numero di Partite: 100 Vittorie AI1 (Livello 1): 2 Vittorie AI2 (Livello 3): 84 Pareggi: 14</p> <p>Tempo Medio di Risposta AI1: 0.0490 secondi per mossa Tempo Medio di Risposta AI2: 0.9217 secondi per mossa</p>	

Le simulazioni mostrano come gli agenti avanzati, configurati con parametri ottimizzati, prevalgano costantemente su quelli più semplici. Questo sottolinea l'importanza di un'euristica ben progettata, una profondità massima adeguata e una corretta parametrizzazione per migliorare le prestazioni. Inoltre, una combinazione equilibrata di strategie offensive e difensive, supportata da tecniche come la potatura alfa-beta e la beam search, garantisce un comportamento strategico robusto e adattabile a diversi livelli di difficoltà.