UNIVERSITÀ DEGLI STUDI
DI SALERNO

Corso di Laurea in Informatica

Forza 4 (meno uno)

Antonio Ferraro, Carmine Iaucci, Daria Simonetti

Marzo 2024

Indice

1 Definizione del Problema

1.1 Introduzione

Il Progetto "Forza 4 (meno uno)" nasce con l'obiettivo di osservare un'inteligenza artificiale quando si scontra in un gioco competitivo con un avversario umano e per comprendere i retroscena e i meccanismi da cui è caratterizzata. Per farlo abbiamo deciso di osservare gli algoritmi di ricerca con avversari applicato a uno dei giochi da tavolo più famosi di sempre: Forza 4.

1.2 Obiettivi

Lo scopo del progetto è quindi quello di realizzare un AI capace di "giocare" a Forza 4. Si tratta di un gioco da tavolo che consiste in una sfida di strategia e abilità per due giocatori. È composto da una griglia di 7 colonne e 6 righe, ognuna delle quali può contenere una serie di dischi contrassegnati con un colore differente per ciascun giocatore.L'obiettivo del gioco è allineare quattro dischi dello stesso colore in orizzontale, verticale o diagonale sulla griglia, prima dell'avversario. Forza 4 richiede una combinazione di strategia, pianificazione e attenzione per prevenire il successo dell'avversario mentre si cerca di raggiungere la propria vittoria.

1.3 Caratteristiche dell'ambiente

- **Completamente Osservabile**: I sensori dell'agente gli danno accesso allo stato completo dell'ambiente in ogni momento.
- **Deterministico:** Una volta che un disco viene posizionato in una colonna, non ci sono variabili casuali che influenzano la sua posizione finale. Ogni configurazione di gioco è completamente determinata dalle scelte fatte dai giocatori.
- **Sequenziale:** Ogni mossa è influenzata dalle mosse precedenti e influenzerà a sua volta le mosse future, creando così una sequenza di eventi che porta alla conclusione della partita.
- **Statico**: L'ambiente non pu'o cambiare mentre l'agente sta decidendo come agire.
- **Discreto:** L'ambiente ha un numero finito di stati distinti. Anche le mosse dei giocatori avvengono in modo discreto, in quanto ogni mossa consiste nel posizionare un disco in una specifica colonna.
- **Multi-Agente Competitivo**: Nell'ambiente abbiamo due agenti, l'IA e il giocatore umano.

1.4 Specifica P.E.A.S

PEAS (Performance Measure, Environment, Actuators, Sensors) è una frormulazione utilizzata per descrivere ambienti operativi. Di seguito è riportata la specifica PEAS dell'ambiente di Forza 4:

- **Performance Measure (Misura delle Prestazioni):**
  Vittoria: Allineare quattro dischi dello stesso colore in orizzontale, verticale o diagonale.
  Pareggio: Riempire completamente la griglia di gioco senza che si verifichi una vittoria da parte di nessun giocatore.
  Sconfitta: Consentire all'avversario di allineare quattro dischi dello stesso colore prima dell'AI
  Metriche Specifiche: Assegnare punteggi più alti a combinazioni che possono avvicinare alla vittoria.

- **Environment (Ambiente)**:
  L'ambiente in cui opera l'agente è costituito dall'insieme di tutte le possibili configurazioni che il tabellone puo' assumere. La griglia di gioco è composta da 7 colonne e 6 righe dove i giocatori posizionano i propri dischi.

- **Actuators (Attuatori):**
  I giocatori possono posizionare un disco nella colonna di loro scelta nella griglia di gioco. Una volta posizionato, il disco scivola verso il basso fino a raggiungere la posizione più bassa disponibile o fino a quando non incontra un altro disco.

- **Sensors (Sensori):**
  I sensori rilevano il tabellone di gioco per determinare la migliore mossa da fare, tenendo conto della posizione dei dischi già presenti.

1.5 Analisi del problema

Considerando che nel gioco Forza 4 abbiamo solamente due agenti, di cui uno (non necessariamente) umano, e che l'intelligenza artificiale cerca attivamente di sconfiggere l'avversario, si è deciso di optare per una soluzione basata su alberi di ricerca. Nel nostro caso, parleremo di albero di gioco, cioè di un albero di ricerca che segue ogni sequenza di mosse fino a raggiungere uno stato terminale. L'agente preleva la mossa migliore dall'albero utilizzando l'algoritmo minimax; l'algoritmo minimax "puro", così com'è, non è adatto per raggiungere gli obiettivi prefissati in tempi brevi, e per questo motivo è stato necessario migliorarlo implementando la tecnica chiamata "Potatura Alpha-Beta". Tale tecnica si è rivelata fondamentale poichè ci ha permesso di esplorare l'albero di gioco in tempi decisamente minori. L'algoritmo usato prevede un punteggio da valutare in base a un determinato stato. Approfondiremo in seguito la nostra funzione di valutavzione.

2 Soluzione del Problema

Per sviluppare un agente intelligente capace di giocare a Forza 4 si è rivelato come prima cosa necessario implementare un tabellone virtuale che i giocatori possono consultare: il giocatore umano lo consulterà tramite una user interface, l'ia consulterà consulterà invece uan bitmap del tabellone. È stato quindi necessario implementare una Graphical User Interface (GUI). Nelle sezioni successive verranno brevemente esplorati questi aspetti del progetto per poi lasciare spazio all'implementazione effettiva dell'agente intelligente.

2.1 Tecnologie Usate

Il progetto è stato sviluppato utilizzando Python e Pygame. Python è un linguaggio di programmazione versatile e potente, ampiamente utilizzato nello sviluppo di giochi e applicazioni grazie alla sua sintassi chiara e della vasta gamma di librerie disponibili. Pygame è una libreria Python specializzata nello sviluppo di giochi 2D, che fornisce funzionalità per la gestione di grafica, input utente e altro ancora, che abbiamo impiegato per la realizzazione della GUI.

2.2 Rappresentazione della posizione

Nel gioco forza 4 le posizioni vengono rappresentate attraverso bitboard. Una bitboard è una struttura dati che utilizza interi binari per rappresentare la posizione degli elementi di un gioco da tavolo. Ogni bit corrisponde a una casella sulla griglia di gioco e può essere utilizzato per rappresentare lo stato (vuoto, occupato da un giocatore) di quella casella. Nel nostro caso abbiamo tre bitboard:

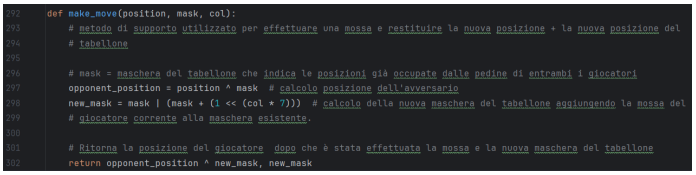
- **"game_position":** Poichè ha dimensioni 6x7 ci sono 42 posizioni totali. Questa bitboard di 42 bit tiene traccia di tutte le celle occupate nel tabellone.

- **"ai_position":** bitboard di 42 bit che tiene traccia delle posizioni occupate dall'AI sul tabellone.

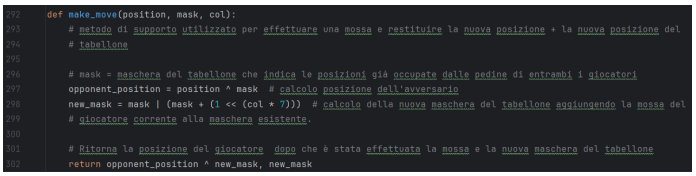
- **"player_position":** questa bitboard, sempre di 42 bit viene calcolata all'interno del programma. La posizione del giocatore umano viene ottenuta eseguendo uno XOR tra le posizioni delle pedine dell'IA e le posizioni totali sul tabellone di gioco. Il risultato di questa operazione è una nuova bitboard in cui sono impostati a 1 solo i bit corrispondenti alle posizioni delle pedine del giocatore umano, mentre i bit corrispondenti alle posizioni delle pedine dell'IA e le celle vuote saranno impostati a 0.

Per la gestione delle posizioni quando viene aggiunta una nuova pedina nel tabellone abbiamo creato due funzioni. Una che viene invocata quando esegue

la mossa l'AI e prende in input la posizione corrente delle pedine dell'IA, la maschera di gioco corrente e la colonna in cui l'IA intende posizionare la pedina. Quindi, calcola la nuova maschera di gioco, aggiungendo una pedina nella colonna desiderata. Successivamente, esegue uno XOR tra la nuova maschera di gioco e la maschera originale per ottenere la nuova posizione delle pedine dell'IA. La seconda funzione viene richiamata quando effettua la mossa un gio-



catore umano e semplicemente riporta sulla maschera di gioco la pedina appena aggiunta. In entrambi i casi, le funzioni calcolano la nuova maschera di gioco

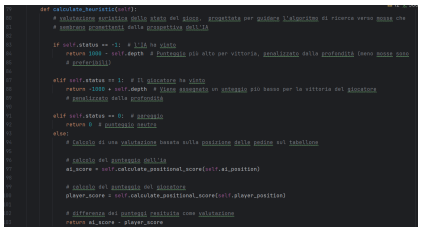


aggiungendo una pedina nella colonna desiderata e poi eseguono uno XOR tra la nuova maschera di gioco e la maschera di gioco originale. Questo aggiornamento delle posizioni è effettuato in modo efficiente utilizzando operazioni bitwise, consentendo una rapida manipolazione delle strutture dati e delle posizioni sul tabellone di gioco.

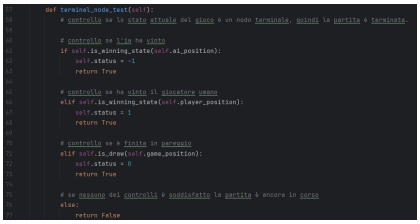
2.3 Funzione di Valutazione Euristica

Di solito è impossibile prendere in considerazione l'intero albero di gioco con l'algoritmo minimax a causa della sua complessità temporale esponenziale. Nell'esaminare completamente l'albero delle mosse possibili fino alla fine del gioco, il numero di configurazioni da considerare crescerebbe in modo significativo con ogni mossa aggiunta. Di conseguenza, diventa necessario tagliare la ricerca a un certo punto e applicare una funzione di valutazione euristica che fornisce la stima dell'utilità di uno stato, in questo caso il posizionamento della pedina in una determinata cella del tabellone.
La funzione euristica è progettata per guidare l'algoritmo di ricerca verso mosse promettenti dalla prospettiva dell'AI nel gioco, senza però peggiorare la complessità della ricerca
La funzione di valutazione che abbiamo definito mira a valutare ogni stato dell'albero di ricerca e ad ognuno assegna un punteggio. In particolare ci siamo concentrati su tre aspetti fondamentali:

- **Valutazione Esiti:** Abbiamo deciso di considerare i tre esiti finali possibili della partita e assegnare ad ognuno un punteggio.
  Se l'AI vince viene assegnato un punteggio più alto, con una penalizzazione in base alla profondità dell'albero di ricerca. Questo punteggio incoraggia l'IA a cercare una vittoria con il minor numero possibile di mosse.
  Se vince il giocatore umano viene assegnato un punteggio più basso, anche qui con una penalizzazione basata sulla profondità. Questo punteggio dissuade l'IA dallo scegliere mosse che conducono a una sconfitta imminente.
  Se la partita termina in pareggio, viene restituito un punteggio neutro. Per verificare se un determinato stato è uno stato terminale abbiamo definito una funzione ad hoc che sarà poi richiamata dall'algoritmo di ricerca.



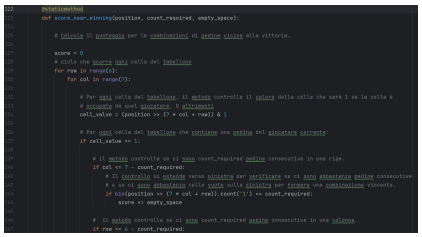
(a) funzione di valutazione euristica.



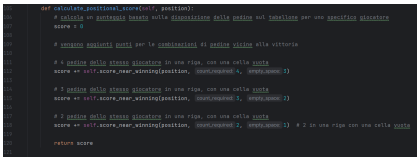
(b) funzione test nodo terminale.

- **Valutazione Disposizioni e Blocco Avversari:** come possiamo vedere dalla *figura a*, nel caso in cui lo stato analizzato non sia uno stato terminale, abbiamo deciso di fornire una valutazione della posizione delle pedine sul tabellone, assegnando punteggi, in modo da definire una strategia per l'AI. Più nello specifico abbiamo creato due funzioni: score_near_winning (*figura a c*) e calculate_positional_score (*figura b d*). La prima funzione si occupa di controllare se ci sono le configurazioni di pedine che abbiamo

scelto di considerare nelle varie direzioni possibili: orizzontale, verticale e diagonale; la seconda funzione invece assegna effettivamente i punteggi alle configurazioni individuate. Abbiamo considerato le configurazioni di 4, 3 e 2 pedine dello stesso giocatore allineate e affiancate da una o più celle vuote.



(a) (c) funzione score_near_winning.



(b) (d) calculate_positional_score.

Con queste due funzioni abbiamo potuto anche sviluppare così il blocco indiretto dell'avversario nel gioco che avviene attraverso la funzione calculate_positional_score. Poichè essa calcola un punteggio basato sulla disposizione delle pedine sul tabellone per uno specifico giocatore, la funzione attribuisce punteggi più alti a combinazioni di pedine che possono portare a una vittoria o che possono bloccare le mosse dell'avversario.
Ad esempio, se l'IA posiziona una pedina in una colonna che blocca una combinazione potenziale di pedine dell'avversario, la funzione assegnerà un punteggio più alto a quella posizione, incoraggiando così l'IA a scegliere quella mossa. In questo modo, l'IA cerca di influenzare indirettamente il gioco bloccando le mosse dell'avversario e cercando allo stesso tempo di creare combinazioni vincenti per sé. L'obbiettivo primario della funzione non è quello appena descritto ma può influenzare indirettamente le decisioni dell'IA in modo da impedire o mitigare le mosse vantaggiose dell'avversario.

Il punteggio finale dell'euristica è la differenza tra il punteggio calcolato per l'IA e quello calcolato per il giocatore. Questo fornisce una stima del vantaggio della situazione attuale per l'IA. La funzione euristica implementata fornisce una valutazione accurata e ragionata della situazione attuale del gioco Forza 4, consentendo all'algoritmo di ricerca di effettuare scelte informate e strategicamente vantaggiose.
Passiamo ora ad analizzare l'algoritmo di ricerca

2.4 Algoritmo di Ricerca: Potatura Alpha-Beta

In questa sezione analizzaremo nel dettaglio l'algoritmo di ricerca utilizzato dalla nostra AI.

2.4.1 Concetto di Base

L'algoritmo Alpha-Beta è una tecnica di potatura utilizzata per migliorare l'efficienza dell'algoritmo Minimax. Si basa sulla potatura di rami dell'albero di ricerca che sono sicuramente meno promettenti rispetto ad altri, senza compromettere la correttezza dell'algoritmo. Si è scelto di utilizzare direttamente questa tecnica poichè, a seguito di alcune ricerche, abbiamo constatato che l'algoritmo Minimax puro avrebbe richiesto tempo di calcolo esponenziale per analizzare l'intero albero di gioco. Infatti l'Alpha-Beta pruning consente di esplorare un numero minore di nodi nell'albero di gioco rispetto al Minimax puro. Questo porta a una riduzione significativa del tempo di calcolo richiesto per prendere una decisione.
Un'altro vantaggio è sciuramente lo spazio di ricerca ridotto. Infatti, utilizzando la potatura Alpha-Beta, è possibile evitare di esplorare interi sottoalberi che non influenzerebbero la decisione finale e ciò avviene quando viene rilevato che il valore dell'azione corrente non migliorerà il risultato dell'azione precedente. Così facendo si riduce in modo significativo lo spazio di ricerca, consentendo all'algoritmo di concentrarsi su mosse più promettenti.

2.4.2 Inizializzazione

Quando viene chiamata la funzione alphabeta_search bisogna passare lo stato iniziale del gioco come argomento. all'interno dell'algoritmo vengono definti una serie di valori:

- Vengono definiti i valori iniziali per l'*alpha* e il *beta*, che sono rispettivamente -infinito e infinito. Questi valori rappresentano i limiti inferiori e superiori del valore dell'azione migliore trovata fino a quel momento.
- Viene anche definito il valore *best_score* inizializzato a -infinito, che rappresenta il punteggio migliore trovato fino a quel momento.
- Viene inizializzata la variabile *best_action* a None, che conterrà l'azione migliore trovata durante la ricerca.
- Infine viene anche creato un dizionario *seen* per tener traccia degli stati figli già visitati.

2.4.3 Corpo dell'algoritmo

Andiamo ora a vedere nel dettaglio il funzionamento dell'algoritmo. Ananlizzeremo logicamente i passaggi che avvengono quando viene richiamato l'algoritmo.

- All'interno della funzione principale alphabeta_search, vengono definite due funzioni interne *max_value* e *min_value*. La prima esplora i nodi di tipo MAX nell'albero di gioco e restituisce il valore massimo ottenibile mentre la seconda esplora i nodi di tipo MIN nell'albero di gioco e restituisce il valore minimo ottenibile. Quindi, entrambe le funzioni esplorano ricorsivamente i nodi dell'albero di gioco e calcolano il valore associato ad ogni stato.

- Viene definita una funzione *cutoff_search* che controlla se lo stato corrente è uno stato terminale o se la profondità massima di ricerca è stata raggiunta. Questa funzione viene utilizzata per determinare se la ricerca deve essere interrotta.

- Viene eseguita una iterazione attraverso tutti i figli dello stato corrente generati dalla funzione *generate_children.*
  Per ogni figlio, viene chiamata la funzione min_value per calcolare il valore che rappresenta la valutazione dell'azione per il giocatore corrente.
  Se il valore calcolato è maggiore del best_score corrente, viene aggiornato il best_score e viene memorizzata l'azione corrispondente in best_action.

- Durante l'esplorazione dell'albero di ricerca, vengono eseguite le potature Alpha-Beta per eliminare i rami dell'albero che non influenzeranno la scelta finale dell'azione. Ciò avviene quando viene rilevato che il valore dell'azione corrente non migliorerà il risultato dell'azione precedente. Quindi Se durante la ricerca si trova un nodo che ha una valutazione che non influenzerà la scelta della mossa, viene potato.

- Decisione migliore: Alla fine della ricerca, viene restituita la mossa migliore, cioè quella che massimizza il vantaggio per il giocatore corrente, considerando le potature Alpha-Beta.

```
def alphabeta_search(state, turn=-1, d=7):
    # algoritmo di ricerca con potatura alpha-beta

    # Definizione delle funzioni che rappresentano i nodi MAX e MIN nell'algoritmo MiniMax. Esplorano ricorsivamente i
    # nodi dell'albero di gioco e calcolano il valore associato ad ogni stato

    # la funzione max_value è responsabile di esplorare i nodi di tipo MAX nell'albero di gioco e restituire il valore
    # massimo ottenibile da tale nodo
    new *
    def max_value(state, alpha, beta, depth):
        if cutoff_search(state, depth):  # controlla se lo stato corrente è uno stato terminale o se la profondità
            # massima di ricerca è stata raggiunta
            return state.calculate_heuristic()  # in caso affermativo viene restituito il valore euristico dello stato

        v = -infinity  # valore massimo che il nodo MAX può ottenere esplorando i suoi figli

        # iterazione sui figli
        for child in state.generate_children(turn):

            if child in seen:  # se il figlio è stato già visto allora si passa al prossimo
                continue

            v = max(v, min_value(child, alpha, beta, depth + 1))  # viene calcolato il valore minimo che il nodo MIN può
            # ottenere esplorando i suoi figli. Il massimo tra il valore corrente di v e il valore calcolato per il
            # figlio viene quindi assegnato a v.

            seen[child] = alpha

            if v >= beta:  # Se il valore di v diventa maggiore o uguale a beta, viene eseguita la potatura Alpha-Beta
                # l'algoritmo restituisce immediatamente il valore di v, poiché il nodo MIN ignorerà completamente
                # questo ramo in quanto il suo valore non può essere superiore a beta.
                return v

            alpha = max(alpha, v)  # alpha viene aggiornata al massimo tra il suo valore corrente e il valore di v

        if v == -infinity:
            # Se il valore di v rimane -infinity, significa che non è stata trovata nessuna vittoria/sconfitta/pareggio
            # fino a questo punto
            return infinity
        return v

    # la funzione esplora i nodi di tipo MIN nell'albero di gioco e restituire il valore minimo ottenibile da tale nodo
    new *
    def min_value(state, alpha, beta, depth):

        if cutoff_search(state, depth):   # controlla se lo stato corrente è uno stato terminale o se la profondità
            # massima di ricerca è stata raggiunta
            return state.calculate_heuristic()  # in caso affermativo viene restituito il valore euristico dello stato

        v = infinity  # valore minimo che il nodo MIN può ottenere esplorando i suoi figli

        for child in state.generate_children(turn):
            if child in seen:
                continue

            v = min(v, max_value(child, alpha, beta, depth + 1))  # calcolo del valore massimo che il nodo MAX può
            # ottenere esplorando i suoi figli. Il minimo tra il valore corrente di v e il valore calcolato per il
            # figlio viene quindi assegnato a v.

            seen[child] = alpha

            if v <= alpha:  # Se il valore di v diventa minore o uguale ad aplha, viene eseguita la potatura Alpha-Beta
                # l'algoritmo restituisce immediatamente il valore di v, poiché il nodo MAX ignorerà completamente
                # questo ramo in quanto il suo valore non può essere inferiore ad alpha.
                return v

            beta = min(beta, v)  # beta viene aggiornata al minimo tra il suo valore corrente e il valore di v

        if v == infinity:
            # Se il valore di v rimane infinity, significa che non è stata trovata nessuna vittoria/sconfitta/pareggio
            # fino a questo punto
            return -infinity
        return v

    # dizionario seen per tenere traccia degli stati già visitati.
    seen = {}

    # corpo dell'algoritmo:

    # funzione cutoff_search restituisce True se lo stato corrente è terminale o se la profondità massima di ricerca è
    # stata raggiunta, altrimenti restituisce False.
    cutoff_search = (lambda state, depth: depth > d or state.terminal_node_test())

    best_score = -infinity
    beta = infinity
    best_action = None

    for child in state.generate_children(turn):  # iterazione su tutti i figli dello stato corrente
        # per ogni figlio:
        v = min_value(child, best_score, beta,  depth 1)  # calcolo del valore che rappresenta la valutazione dell'azione
        # per il giocatore corrente.

        if v > best_score:  # se v è maggiore del best score
            best_score = v
            best_action = child  # l'azione migliore sarà il figlio corrente

    # alla fine del ciclo best-action conterrà l'azione che l'algoritmo ritiene essere la migliore da svolgere
    return best_action
```

Figure 3: funzione alphabeta_search.

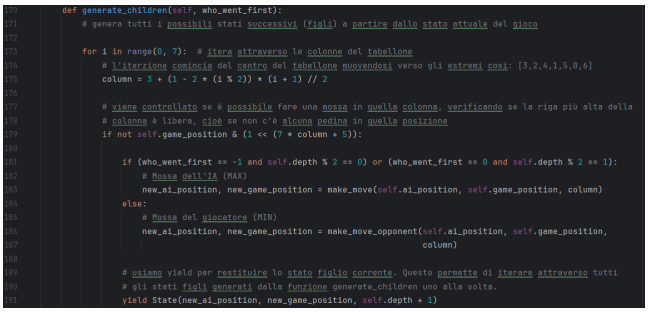
2.4.4 Funzioni esterne richiamate

Vengono richiamate nel corpo dell'algoritmo una serie di funzioni definite ed implementate esternamente:

- **Funzione di Valutazione:** La funzione *calculate_eheuristic* viene utilizzata per calcolare la valutazione euristica di uno stato.
  Se lo stato è uno stato terminale, il punteggio euristico viene impostato in base al risultato della partita (vittoria dell'IA, vittoria del giocatore o pareggio).
  Se lo stato non è uno stato terminale, viene calcolato il punteggio euristico basato sulla posizione delle pedine sul tabellone per entrambi i giocatori.
  L'approfondimento di questa funzione si trova nella sezione 2.3

- **Valutazione della posizione delle pedine:** La funzione *calculate_positional_score*calcola il punteggio basato sulla disposizione delle pedine sul tabellone per uno specifico giocatore.
  Vengono aggiunti punti per le combinazioni di pedine vicine alla vittoria, con punteggi differenti in base al numero di pedine allineate e allo spazio vuoto tra di esse.
  L'approfondimento di questa funzione si trova nella sezione 2.3

- **Generazione dei figli:** La funzione *generate_children* genera tutti i possibili stati successivi (figli) a partire dallo stato attuale del gioco.
  Viene verificato se è possibile fare una mossa in una colonna specifica e vengono generati gli stati successivi corrispondenti a tale mossa.
  Nello specifico la funzione prende in input lo stato attuale del gioco e chi ha mosso per primo.
  Per ciascuna colonna del tabellone, la funzione genera una nuova mossa potenziale. La sequenza delle colonne è determinata da un pattern che inizia dal centro e si espande verso gli estremi, alternando la direzione.
  Per ogni colonna, la funzione verifica se la mossa è valida controllando se la colonna non è piena. Se la colonna è già occupata, la mossa non viene generata.
  Se la mossa è valida, la funzione calcola il nuovo stato del gioco dopo che il giocatore corrente ha effettuato la mossa.
  La funzione genera lo stato successivo utilizzando yield, che permette di restituire uno stato alla volta senza dover generare e memorizzare tutti gli stati contemporaneamente. Ciò consente di risparmiare memoria, in particolare quando il numero di mosse è elevato.



```
    def generate_children(self, who_went_first):
        # genera tutti i possibili stati successivi (figli) a partire dallo stato attuale del gioco

        for i in range(0, 7):  # itera attraverso le colonne del tabellone
            # l'iterzione comincia dal centro del tabellone muovendosi verso gli estremi così: [3,2,4,1,5,0,6]
            column = 3 + (1 - 2 * (i % 2)) * (i + 1) // 2

            # viene controllato se è possibile fare una mossa in quella colonna, verificando se la riga più alta della
            # colonna è libera, cioè se non c'è alcuna pedina in quella posizione
            if not self.game_position & (1 << (7 * column + 5)):

                if (who_went_first == -1 and self.depth % 2 == 0) or (who_went_first == 0 and self.depth % 2 == 1):
                    # Mossa dell'IA (MAX)
                    new_ai_position, new_game_position = make_move(self.ai_position, self.game_position, column)
                else:
                    # Mossa del giocatore (MIN)
                    new_ai_position, new_game_position = make_move_opponent(self.ai_position, self.game_position,
                                                                            column)

                # usiamo yield per restituire lo stato figlio corrente. Questo permette di iterare attraverso tutti
                # gli stati figli generati dalla funzione generate_children uno alla volta.
                yield State(new_ai_position, new_game_position, self.depth + 1)
```

Figure 4: funzione generate_children.

3 Risultati

L'algoritmo implementato ci ha permesso di creare un'intelligenza artificiale che è perfettamente capace di giocare al gioco Forza 4 in tempi decisionali adeguati. Abbiamo però notato che il tempo decisionale aumenta leggermente nel momento in cui l'AI muove il suo gettone per ostacolare il giocatore umano nel caso in cui quest'ultimo stia per vincere.
Giocando 20 partite contro l'AI siamo riusciti ad arrivare a una situazione di pareggio solo una volta, nella fase di sviluppo in cui l'AI non era ancora in grado di agire considerando la possibilità di bloccare il giocatore avversario.
Una volta risolto questo problema, ancora non siamo riusciti a battere l'AI.

4 Conclusioni

La creazione e lo sviluppo di questo progetto ci hanno permesso di capire ancora meglio l'applicazione dell'intelligenza artificiale nel mondo dei giochi, permettendoci di esplorare più approfonditamente il campo dei giochi con avversari. Abbiamo trovato l'esperienza divertente ed educativa. Ovviamente, ci sono ancora molte cose da migliorare e argomenti da approfondire; sarebbe ad esempio interessante applicare algoritmi di reinforcement learning in modo che l'AI impari dalle sue esperienze.