Ottimizzazione della strategia di scelta per un gioco a turni di tipo Connect-X

Partecipanti del gruppo:
Mat. 0001081166 - Alessio Prato
Mat. 0001069415 - Andrea Santilli
Mat. 0001078029 - Leonardo Pinna

Abstract

- Il problema prevede la selezione della mossa migliore tra quelle a disposizione in un gioco a turni di tipo connectx, dove data una matrice di gioco con M righe e N colonne, l'obiettivo è allineare X oggetti o pedine in fila orizzontalmente, verticalmente, o diagonalmente. Due giocatori si sfidano e dopo la prima mossa del giocatore 1, a turno scelgono una colonna dove inserire la pedina, impilata dal basso verso l'alto, fino al termine della partita con la vittoria di uno dei due giocatori (X pedine connesse), il pareggio (nessuna mossa disponibile) o la sconfitta a tavolino per uno degli errori di gioco o di timeout. La scelta della mossa deve essere effettuata entro un certo tempo limite dato in input al programma.
- Per risolvere il problema stato è implementato un algoritmo ricorsivo MiniMax di valutazione dell'albero di gioco ad una profondità data in input, con ottimizzazione AlphaBeta Pruning per migliorare il caso medio, con iterazione dell'algoritmo MiniMax ad una profondità crescente (Iterative Deepening) e con una ulteriore ottimizzazione finale di ordinamento delle mosse dalla più promettente alla meno promettente per rendere più efficace l'ottimizzazione AlphaBeta Pruning.
- Particolare attenzione è stata data allo sviluppo di soluzioni algoritmiche che minimizzino la possibilità di errore di gioco, come la selezione di mosse non valide, o errore di timeout, che comportano l'immediata sconfitta e annullano qualsiasi risultato intermedio ottenuto.
- È stata creata una funzione di valutazione dello stato di gioco che prevede la valutazione del potenziale di ogni pedina giocata da entrambi i giocatori, valutando la posizione di ogni elemento della matrice di gioco in funzione dello stato degli elementi della matrice adiacenti all'elemento valutato nelle direzioni utili (orizzontale, verticale, diagonale)
- È stata studiata la complessità computazionale della soluzione proposta.
- Sono state studiate soluzioni alternative al problema, in particolare è stato sperimentato un approccio di ricerca di tipo Monte Carlo che prevede, per ogni mossa possibile in uno stato di gioco, un elevato numero di partite simulate in maniera casuale, l'attribuzione di un punteggio in funzione del risultato ottenuto per ogni mossa valutata
- Sono stati definiti possibili ulteriori sviluppi di ottimizzazione per la risoluzione del problema dato, come il miglioramento della funzione di valutazione per uno stato di gioco e l'utilizzo di tecniche di apprendimento automatico per l'ottimizzazione della scelta.

Introduzione

In questa sezione, si introducono le regole di gioco sottostante al problema di ottimizzazione della scelta.

Il tipo di gioco: ConnectX

Un gioco di tipo ConnectX è di tipo a turni, con superficie di gioco rappresentata da una matrice di M righe e N colonne. Il gioco prevede due giocatori, il giocatore P1 che ha il primo turno, e il giocatore P2. Ogni giocatore ha una propria tipologia di pedina. Uno dopo l'altro, i giocatori selezionano una colonna dove inserire la propria pedina. Lo scopo del gioco è allineare X pedine di uno stesso giocatore prima dell'altro giocatore. Se non ci sono più mosse a disposizione, la partita finisce in pareggio.

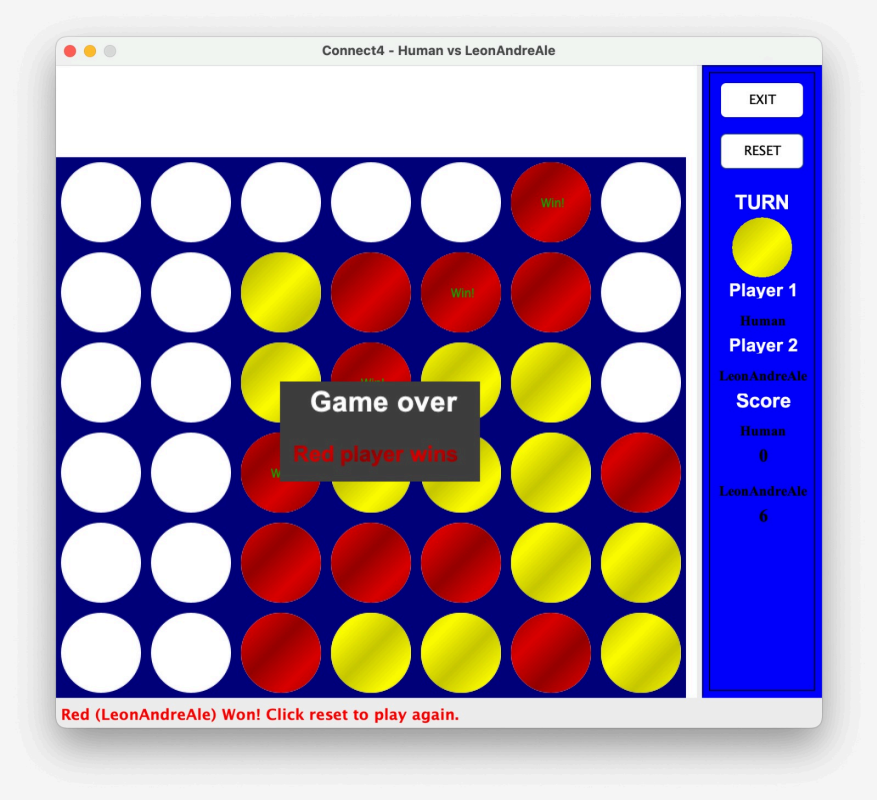

Figura 1: l'algoritmo ha vinto contro un giocatore umano in una partita 6x7x4.

Superficie di gioco

La superficie di gioco è rappresentata da una matrice di M righe e N colonne, e ogni elemento (i, j) della matrice rappresenta una pedina che può assumere uno dei 3 stati di gioco disponibili.
Ogni pedina della MxN pedine ha posizione (i, j) con i=0,…M-1 e j=0,…N-1.

Giocatori

Il gioco prevede due giocatori, il giocatore P1, che ha il primo turno, e il giocatore P2. Ogni giocatore ha una propria tipologia di pedina. Uno dopo l'altro, i giocatori selezionano una

colonna dove inserire la propria pedina che assume la posizione della prima riga libera a partire dal basso.

Stati di gioco

Gli stati di gioco possibili sono 4:
- WIN1: il giocatore 1 vince
- WIN2: il giocatore 2 vince
- DRAW: pareggio (non ci sono più mosse disponibili e nessuno ha vinto)
- OPEN: partita aperta

Pedine

Ogni cella della superficie di gioco in posizione (i, j) (i-esima riga, j-esima colonna) ha 3 possibili stati:
- P1: occupata da una pedina del giocatore 1
- P2 occupata da una pedina del giocatore 2
- FREE: cella libera

Tempo limite

Il gioco prevede per ogni mossa un tempo limite entro il quale effettuare una scelta. Se la mossa non è selezionata entro il tempo limite, si perde la partita. Il tempo limite è un parametro in input che è utilizzabile dalla funzione checkTime() per valutare il tempo residuo.

Turni di gioco

Ogni partita prevede che i due giocatori inseriscano a turno una pedina all'interno della matrice di gioco.
La pedina può essere inserita in una delle colonne disponibili e, scelta la colonna j tra quelle disponibili (da 0 a N-1), la pedina viene posizionata nella prima riga i disponibile a partire dal basso (dall'indice M-1 fino a 0). Una colonna j che presenta la cella (0, j) non libera è piena e non è più disponibile per la selezione.

Errori di gioco

Se non si pongono vincoli alla valutazione delle mosse e alla scelta della colonna, è possibile incorrere in un errore di gioco che comporta la sconfitta immediata. Una lista (non esaustiva) dei principali errori di gioco è riportata qui:
- incide fuori dall'intervallo corretto (IndexOutOfRange): si sceglie una colonna j minore di zero o maggiore o uguale a N, si sceglie una riga i con i minore di 0 o i maggiore o uguale a M.
- Colonna non disponibile o piena (IllegalMove): si sceglie una colonna già piena per la prossima mossa
- Eccessivo utilizzo di tempo (TimeOut): il superamento del tempo limite comporta la sconfitta immediata per il giocatore.

Scopo del lavoro

Lo scopo del presente lavoro è fornire uno o più algoritmi ottimizzati di ricerca per la scelta della migliore mossa da giocare, in funzione dello stato attuale della matrice di gioco.

Soluzione proposta al problema computazionale affrontato

Il problema della scelta delle mosse disponibili

Il problema richiede la scelta della prossima mossa migliore tra le mosse disponibili. Il problema si riduce alla scelta della colonna dove inserire la pedina che deve essere effettuata tra le $N - Z$ colonne disponibili, dove N è il numero di colonne e Z è il numero di colonne piene e ritornata attraverso il metodo `selectColumn()` della classe player.
Il pacchetto connectx fornisce a partire da una classe di tipo `CXBoard` che rappresenta la matrice di gioco un metodo per ottenere un array di interi che rappresenta la lista di mosse disponibili (`getAvailableColumns()`) che è stato utilizzato come punto di partenza per la scelta delle mosse da valutare, perchè contiene tutte e sole le mosse possibili per un determinato stato di gioco.
Per ciascuna mossa, è stato implementato un algoritmo di assegnazione di un punteggio secondo l'algoritmo di ricerca MiniMax, con ottimizzazione AlphaBeta Pruning e, per le gestione del tempo limite, è stata utilizzata una strategia Iterative Deepening e un ordinamento intermedio delle mosse dalla più promettente alla meno promettente ad ogni iterazione completata. Una volta assegnato il punteggio alle singole mosse valutate, risulta triviale scegliere la mossa che risulta essere quella con punteggio più favorevole.
Per convenzione, abbiamo definito che il "nostro" giocatore è il giocatore che massimizza e non quello che minimizza, e quindi la scelta della mossa risulta essere quella che ha ottenuto un punteggio di valutazione più alto.

Algoritmo MiniMax

È stato implementato un algoritmo ricorsivo MiniMax che in input riceve i seguenti parametri:

- la matrice di gioco di tipo `CXBoard`
- Un valore booleano che indica se è il turno del giocatore che massimizza il punteggio (true), o se è il turno del giocatore che minimizza il punteggio (false)
- La profondità massima raggiunta prima di effettuare una valutazione

L'algoritmo MiniMax è un algoritmo ricorsivo che può essere assimilato ad una visita in profondità per una struttura dati di tipo albero ($N - Z_d$)-ario (ogni nodo ha un numero di figli pari al numero di colonne disponibili Z_d che varia in funzione della posizione), che visita tutti i sotto-alberi figli e se incontra una foglia, ritorna un punteggio intero.
Ogni chiamata ricorsiva si effettua selezionando una mossa tra quelle disponibili nella matrice di gioco attuale, invertendo il valore di maxPlayer (perchè è il turno dell'avversario) e riducendo la profondità di una unità per poter arrivare al caso base della struttura ricorsiva.
Per convenzione, abbiamo definito che il "nostro" giocatore è il giocatore che massimizza. Se il giocatore attuale è maxPlayer, allora si inizializza il valore bestScore come l'intero minimo (In Java Integer.MIN_VALUE) e il punteggio migliore, tra i punteggi delle mosse disponibili, è il più alto. Se il giocatore attuale non è maxPlayer, allora si inizializza il valore bestScore come l'intero massimo (In Java Integer.MAX_VALUE) e il punteggio migliore, tra i punteggi delle mosse disponibili, è il più basso.
Le foglie dell'albero, che rappresentano i casi base della funzione ricorsiva, si raggiungono in diverse situazioni:

- lo stato di gioco non è più aperto (!OPEN): la partita è terminata e non è necessario valutare ulteriori mosse per quella partita. Si assegna un punteggio in funzione dello stato di gioco: vittoria di un giocatore o pareggio.
- Si è raggiunta la profondità massima di valutazione (`depth == 0`): se il gioco è ancora aperto alla profondità valutata, si ritorna un punteggio valutato dalla funzione `evaluatePosition()` che in input prende la posizione corrente della matrice di gioco.

```
if (B.gameState() == myWin) {
    return Integer.MAX_VALUE; // Vittoria immediata
}
if (B.gameState() == yourWin) {
    return Integer.MIN_VALUE; // Sconfitta immediata
}
if (B.gameState() == CXGameState.DRAW) {
    return 0; // Pareggio
}
if (depth == 0) {
    // Raggiunto il limite di profondità della ricerca, valuta la posizione corrente
    return evaluatePosition(B);
}
```

Figura 2: casi base dell'algoritmo miniMax

Ottimizzazione AlphaBeta Pruning

Per ridurre il numero di foglie visitate dall'algoritmo Minimax, è stato introdotta un'ottimizzazione di tipo AlphaBeta Pruning, che nel caso medio ritorna lo stesso valore di MiniMax effettuando meno operazioni perchè non valuta le configurazioni che non possono ottenere un punteggio migliore rispetto a quello già ottenuto da un'altra mossa.

```
int bestScore;
if (maxPlayer) {
    bestScore = Integer.MIN_VALUE;
    Integer[] moves = B.getAvailableColumns();
    for (int move : moves) {
        B.markColumn(move);
        int score = evaluateMove(B, maxPlayer:false, depth - 1, alpha, beta);
        B.unmarkColumn();
        bestScore = Math.max(bestScore, score);
        if (bestScore == Integer.MAX_VALUE) break;
        alpha = Math.max(alpha, score);
        if (beta <= alpha) {
            break; // Beta cut-off
        }
    }
} else {
    bestScore = Integer.MAX_VALUE;
    Integer[] moves = B.getAvailableColumns();
    for (int move : moves) {
        B.markColumn(move);
        int score = evaluateMove(B, maxPlayer:true, depth - 1, alpha, beta);
        B.unmarkColumn();
        bestScore = Math.min(bestScore, score);
        if (bestScore == Integer.MIN_VALUE) break;

        beta = Math.min(beta, score);
        if (beta <= alpha) {
            break; // Alpha cut-off
        }
    }
}

return bestScore;
```

Figura 3: chiamata ricorsiva di MiniMax con ottimizzazione AlphaBeta

L'algoritmo AlphaBeta implementato tiene traccia di due parametri (`alpha` e `beta`, inizializzati come $-\infty$ e $+\infty$ rispettivamente nel campo degli interi) aggiornati per ogni punteggio calcolato e, nel caso in cui beta dovesse essere inferiore ad alpha, la ricerca delle mosse successive viene interrotta perchè non è possibile trovarne di migliori.

Algoritmo Iterative Deepening

Uno dei problemi principali dell'implementazione dell'algoritmo MiniMax ha riguardato la gestione del tempo limite. Infatti, una scelta arbitraria della profondità massima di MiniMax nel caso migliore porta ad una selezione sub-ottima della mossa, nel caso peggiore porta a non completare l'algoritmo di ricerca entro il tempo limite e quindi ad una sconfitta immediata.
Per non incorrere nell'errore TimeOut si è resa necessaria l'implementazione di un algoritmo che permettesse di ottenere risultati sub-ottimi, ma sempre più accurati, in un tempo accettabile. La soluzione proposta è l'esecuzione di un algoritmo MiniMax a profondità sempre crescenti che al termine di ogni esecuzione del ciclo salvi la mossa migliore ottenuta in un parametro che, in caso di TimeOut, rappresenta il valore ritornato da `selectColumn()`. Iterative Deepening è quindi l'applicazione dell'algoritmo MiniMax per profondità sempre crescenti in maniera iterativa.
Idealmente, non è necessario stabilire un valore di massima profondità per l'iterazione `MAX_DEPTH`, posto che la valutazione del tempo restante tramite `checkTime()` sia effettuata con frequenza accettabile all'interno dell'algoritmo.

```java
public int iterativeDeepening(CXBoard B, int depth) throws TimeoutException {
        // Io sono il giocatore che massimizza
        int bestScore = Integer.MIN_VALUE;

        // Inizializzazione parametri
        Integer[] Q = B.getAvailableColumns();
        int nextMove = Q[rand.nextInt(Q.length)];
        int[] scores = new int[Q.length];

        for (int d = 1; d <= depth; d++) {

            // Inizializzo i punteggi a 0
            for (int i = 0; i < Q.length; i++) {
                scores[i] = 0;
            }
            // Valuto ogni mossa disponibile
            for (int i = 0; i < Q.length; i++) {
                B.markColumn(Q[i]);
                // Metto falso perchè avendo giocato la mia mossa simulata, è il turno dell'avversario.
                scores[i] = evaluateMove(B, maxPlayer:false, d, Integer.MIN_VALUE, Integer.MAX_VALUE);
                B.unmarkColumn();
            }
            // Assegno la mossa migliore a nextScore
            for (int i = 0; i < Q.length; i++) {
                if (scores[i] < bestScore) {
                    bestScore = scores[i];
                    nextMove = Q[i];
                }
            }
            // Aggiorno la mossa migliore trovata in queste iterazione
            this.currentBestMove = nextMove;
        }
        return this.currentBestMove;
    }
```

Figura 4: algoritmo Iterative Deepening non ottimizzato

Ottimizzazione di ordinamento delle mosse migliori

Ad ogni ciclo di Iterative Deepening, si ottiene un risultato parziale migliorato rispetto al ciclo precedente (perchè sono analizzati casi a profondità maggiore). In termini di punteggi associati alla mossa migliore, questo significa che ad ogni iterazione si possono ordinare le mosse da valutare (sempre le stesse perchè per ogni iterazione lo stato di gioco da valutare è lo stesso) in ordine crescente rispetto al punteggio ottenuto da ciascuna mossa nell'ordinamento precedente. Per associare le mosse da valuare ai rispettivi punteggi, è stata utilizzata una struttura dati di tipo `Pair` che associa un valore chiave `key`, che rappresenta la mossa i-esima nell'array di mosse o colonne disponibili, ad un valore `value`, che rappresenta il punteggio calcolato per la i-esima mossa per una determinata iterazione di Iterative Deepening.

```java
public class Pair<K, V> {
    private final K key;
    private final V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

Figura 5: Struttura dati che associa una colonna ad un punteggio

Iterative Deepening è stato integrato quindi con l'inserimento dell'i-esima mossa e dell'i-esimo punteggio nella stessa classe Pair, e il successivo inserimento dell'elemento di tipo Pair in una lista List. Gli elementi di tale lista sono ordinati in modo decrescente in funzione del punteggio delle mosse. È possibile salvare la migliore mossa prendendo il valore key del primo elemento della lista.
Inoltre, l'array di mosse disponibili è ordinato e nell'iterazione successiva, le prima mosse valutate saranno quelle più promettenti trovate nell'iterazione appena conclusa.

```java
// Creare una lista di coppie (Q[i], scores[i])
List<Pair<Integer, Integer>> pairList = new ArrayList<>();
for (int i = 0; i < Q.length; i++) {
    pairList.add(new Pair<>(Q[i], scores[i]));
}

// Ordinare la lista in base al punteggio in ordine decrescente
pairList.sort((a, b) -> b.getValue().compareTo(a.getValue()));

// Aggiornare gli array Q e scores con gli elementi ordinati
for (int i = 0; i < pairList.size(); i++) {
    Pair<Integer, Integer> pair = pairList.get(i);
    Q[i] = pair.getKey();
    scores[i] = pair.getValue();
}
```

Figura 6: Associazione per indice di mossa valutata e punteggio ottenuto, e successivo ordinamento dell'array di mosse, all'interno di Iterative Deepening

Scelte progettuali adottate

Il “nostro” giocatore è sempre il giocatore che massimizza

Per scelta convenzionale, il giocatore è sempre il giocatore che massimizza. In maniera simmetrica avremmo potuto definire il “nostro” giocatore come il giocatore che minimizza e invertire in maniera accorta i parametri di MiniMax e degli altri algoritmi.

Controllo del tempo limite

È stato scelto di usare una funzione `checkTime()` che calcola il tempo rimanente dal momento in cui la valutazione della mossa è iniziata e crea un eccezione se il tempo residuo è inferiore all’1% del tempo inizialmente dato.
Il posizionamento all’interno dell’algoritmo di `checkTime()` richiede una attenta valutazione, perchè da un lato se il tempo intercorso tra due chiamate di `checkTime()` è troppo alto si potrebbe ricadere in un errore TimeOut (sconfitta immediata), dall’altro una chiamata troppo frequente di `checkTime()` (che di per sé ha complessità O(1)) potrebbe risultare in un numero complessivo inferiore di posizioni di gioco valutate. La nostra scelta è ricaduta nell’inserire nel ciclo più annidato questa funzione, aumentando il numero di chiamate della funzione ma a favore di sicurezza rispetto al raggiungimento dell’errore TimeOut.

Metodo `copy()` VS `markColumn()/unmarkColumn()`

La classe `CXBoard` fornisce due metodi per gestire le mosse simulate: `copy()` che crea una copia della matrice di gioco, e `markColumn(col)` e `unmarkColumn()` per aggiungere una mossa nella colonna col, e per rimuovere l’ultima mossa giocata.
L’utilizzo esclusivo e indiscriminato di `markColumn()` porta a errori di tipo IllegalMove perchè non si tiene in considerazione che ogni mossa valutata deve essere rimossa prima di valutare la mossa successiva.
In generale, per la strategia di assegnazione e eliminazione della mossa, è necessario che prima di ogni chiamata `markColumn()`, la colonna selezionata sia scelta tra le colonne disponibili in quel momento, ottenute tramite il metodo `getAvailableColumns()`, e che per ogni `markColumn()` sia effettuata una chiamata a `unmarkColumn()` dopo che è stata elaborata la matrice con mossa eseguita per rimuovere tale mossa.
Pertanto dal lato pratico la soluzione `copy()` è apparentemente più semplice.
È stato però previsto e verificato che la creazione di una copia della matrice per ogni nodo non-foglia dell’albero di gioco, già ad una profondità non così elevata, portava ad un errore di tipo OutOfMemory, pertanto sono stati utilizzati i metodi `getAvailableColumns()`, `markColumn()` e `unMarkColumn()` attentamente nelle fasi di valutazione dell’intervallo di mosse disponibili ad una determinata profondità e situazione di gioco.

Gestione dell’errore TimeOut

La gestione dell’errore Time Out è stata effettuata utilizzando il metodo iterativo a profondità crescente (Iterative Deepening), con il salvataggio intermedio dei risultati. Una volta che una iterazione è completata, la colonna migliore è aggiornata con il nuovo risultato ottenuto. Uno dei problemi che si potrebbe incontrare in caso di numero di colonne elevato o tempo a disposizione molto basso, è il non completamento del primo ciclo di Iterative Deepening. In

quel caso l'attributo della classe del giocatore che contiene la colonna ritornata è preventivamente assegnata ad un valore casuale all'inizio del metodo selectColumn(). La strategia di gioco non è ottimizzata in quel caso.

Funzione di valutazione di stato

Uno dei parametri più importanti per la corretta assegnazione di un punteggio alle mosse è la valutazione di una posizione aperta (OPEN) in funzione del posizionamento delle pedine.
La scelta per la creazione di una funzione di valutazione è ampia e pertanto risulta molto difficile definire una strategia univoca ottimale per assegnare un punteggio ad una posizione statica.
In particolare, risulta complesso sia dal punto di vista progettuale che dal punto di vista computazionale valutare la presenza di potenziali combinazioni di gioco *future* che possano essere attivamente cercare nella posizione *attuale* di gioco (senza metodi previsionali), perchè la posizione di gioco alla quale siamo arrivati e che dobbiamo valutare proviene dall'applicazione di mosse tramite l'algoritmo MiniMax e stimare ulteriori mosse significa procedere a passi ulteriori dell'algoritmo MiniMax (non possibile).
La sfida principale riguarda quindi la valutazione di una posizione *statica* nel contesto di una partita *dinamica*, dove una pedina ha un valore superiore o inferiore in funzione del *potenziale* valore di collegamento con le pedine in posizioni vicine ad essa.
La nostra scelta progettuale è ricaduta nella valutazione dello stato di ogni pedina giocata rispetto allo stato delle pedine entro una distanza limite da essa l'assegnazione di un punteggio in funzione del potenziale della pedina.
La distanza limite è stata considerata nell'intervallo $[-X, +X]$ per tenere in considerazione, nell'assegnazione del punteggio, solamente dello stato delle celle *rilevanti* per la posizione valutata, mentre le celle troppo lontane per poter essere connesse in maniera vincente non sono utili per la valutazione del potenziale di una pedina.
Le direzioni valutate sono 7:

- una direzione verticale (solo verso l'alto) (verso il basso non ha significato ulteriore rispetto alla valutazione verso l'alto)
- due direzioni orizzontali (da sinistra a destra, da destra a sinistra)
- quattro direzioni diagonali.

Ad esempio, per un gioco connect-4, sono considerate le 4 pedine prima e dopo ([-4, -3, -2, -1, 1, 2, 3, 4]).
L'intervallo $[-X, +X]$ dipende dalla direzione considerata: può essere lungo le colonne, lungo le righe o entrambi (diagonali). Quindi il parametro k del paragrafo seguente può assumere il valore di i (riga), j (colonna) o $\pm i$ e $\pm j$ della cella in posizione (i, j) considerata.
Partendo dalla posizione $p = k - X$, valuto tutte le pedine consecutive fino alla posizione $p = k + X$. Se trovo una pedina vuota, aggiungo un punteggio. Se trovo una pedina occupata da me, aggiungo un punteggio maggiore. Se trovo una pedina dell'avversario, azzero il punteggio perchè ricomincia la ricerca di pedine connesse. Se trovo una pedina dell'avversario oltre la pedina attuale nella direzione valutata, esco dal loop di ricerca.
Le posizioni considerate sono quelle valide, quindi la valutazione non inizia da p ma da valori che sono entro gli estremi della matrice (0,…M-1 o 0,..N-1) solo quando si raggiunge

una posizione valida (i, j). Se supero un estremo con $p > k$, allora il ciclo finisce perchè non posso procedere oltre la pedina e restare dentro la matrice.

```
// Valutazione della colonna (verso l'alto)
for (int index = - getK() ; index <= getK(); index++) {

    // Arrivo oltre il limite superiore della matrice. Interrompo la ricerca per colonna.
    if ( x - index  <= -1) {
        // Zero punti perchè non posso fare molto.
        break;
    }

    // Se non sono al di sotto della matrice, considero le pedine entro il range di valutazione.
    if (!(x - index >= getRows())) {
        if (B.cellState(x - index, y) == CXCellState.FREE) {
            // La cella sopra è libera: vuol dire che potenzialmente posso continuare la streak
            counter += emptyVal;
        } else if (B.cellState(x - index, y) == B.cellState(x,y) ) {
            // La cella sopra è uguale a questa cella: ottengo molti punti.
            counter += columnVal;
        } else {
            // La cella sopra non è uguale alla cella attualmente valutata; la cella è bloccata verso l'alto.
            // Zero punti.
            counter = 0;
            if (index > 0) {break;}
            break;
        }
    }
}
```

Figura 7: Esempio di valutazione di una cella (x,y) con un indice index che va da -X a +X

Gli step della funzione di ricerca sono i seguenti:

1. Sono inizializzati i punteggi di giocatore 1 e di giocatore 2 uguali a 0.
2. Per ogni pedina la funzione di valutazione calcola un punteggio.
3. Il punteggio di ogni pedina è sommato al punteggio totale del giocatore che ha giocato quella pedina.
4. La funzione di valutazione ritorna il punteggio della posizione come punteggio giocatore 1 meno punteggio giocatore 2.

Si ottiene un punteggio che viene ritornato dal caso base dell'algoritmo MiniMax e valutato rispetto ai punteggi delle altre mosse ad una determinata profondità.

Analisi del costo computazionale

I parametri principali che determinano la complessità di gioco sono 3: M il numero di righe, N il numero di colonne, X il numero di "pedine" da allineare per vincere.
Questi parametri determinano la complessità computazionale complessiva dell'algoritmo di ricerca e/o di valutazione delle mosse.

Analisi della complessità temporale

Partendo dalle funzioni più annidate, si risale al costo computazionale temporale della soluzione proposta.

Analisi della funzione di valutazione

La funzione di valutazione prevede:

- un ciclo for che esegue T cicli, con T pari al numero di celle non FREE, ovvero il numero di mosse eseguite fino a quel momento, ovvero la profondità massima raggiunta da MiniMax più le mosse giocate prima del lancio di MiniMax per il turno attuale. Ogni ciclo esegue tutti i 7 cicli for evidenziati in seguito.
- 7 cicli for che eseguono ciascuno $2X$ cicli, ciascuno di costo costante $O(1)$ (alcune operazioni if-else e comparazioni tra interi o chiamate a funzioni con costo costante), quindi il costo di questi cicli è $O(2X) = O(X)$.

Il costo complessivo della funzione di valutazione nel caso pessimo è $O(TX)$, dove T è il numero di mosse giocate fino a quel momento, e X è il numero di pedine da connettere per vincere la partita.
Nel caso medio e nel caso ottimo, il costo computazionale è lo stesso del caso pessimo. Tuttavia, sono stati inseriti alcuni break che interrompono i cicli quando si valutano pedine ad esempio in posizioni vicino al bordo, o in particolari posizioni i cicli for annidati sono interrotti prima di essere completati.

Analisi dell'algoritmo MiniMax

L'algoritmo MiniMax ha un costo dominato dal costo computazionale relativo alle foglie a profondità massima, ovvero se D è la profondità massima in input all'algoritmo MiniMax, il costo computazionale è superiormente limitato da $(N - Z)^D$, dove N è il numero di numero di colonne e Z è il numero di colonne piene, quindi $O((N - Z)^D)$. Ogni nodo dell'albero di gioco di MiniMax infatti valuta al massimo $N - Z$ mosse, fino ad una profondità D. Per $D < M$, si può considerare un costo computazionale pari a $O(N^D)$, in quanto sicuramente nessuna colonna sarà stata riempita ($Z = 0$).
L'ottimizzazione AlphaBeta Pruning aiuta la valutazione dell'algoritmo MiniMax fornendo lo stesso risultato e riducendo nel caso medio il numero di nodi visitati. Inoltre, l'ulteriore ottimizzazione dell'ordinamento delle mosse dalla più promettente alla meno promettente rende il caso medio più efficiente perchè il "taglio" del ramo di AlphaBeta è più probabile.
Nel caso pessimo, per ogni nodo foglia l'algoritmo MiniMax chiama la funzione di valutazione che ha costo $O(TX)$. Pertanto l'algoritmo MiniMax ha una complessità complessiva $O(TX(N - Z)^D)$.

Analisi di Iterative Deepening

L'algoritmo Iterative Deepening chiama l'algoritmo MiniMax con profondità crescente fino a profondità D_{MAX}. Fissata una certa profondità massima D_{MAX}, Iterative Deepening ha un costo computazionale pari a

$$\sum_{i=1}^{D_{MAX}} O(TX(N-Z)^i) = O(TX(N-Z)^{D_{MAX}})$$

dove:

- T è il numero di celle non libere, quindi il numero di mosse già effettuate
- X è il numero di celle da allineare per vincere la partita
- N è il numero di colonne della matrice di gioco
- Z è il numero di colonne complete della matrice
- D_{MAX} è il numero massimo di iterazioni per Iterative Deepening

Quando la matrice è quasi completa, vale che $Z \to N$ e $T \to MN$.
In generale, considerando una funzione di valutazione con complessità F_C, la complessità dell'algoritmo di Iterative Deepening è $O(F_C(N-Z)^{D_{MAX}})$.

Analisi della complessità spaziale

Per quanto riguarda la complessità spaziale, l'algoritmo non utilizza uno spazio di memoria eccessivo e quindi il rischio di un errore OutOfMemory è secondo il nostro parere remoto.

Funzione di valutazione

La memoria è richiesta per la matrice B ($O(MN)$), alcune costanti e parametri con costo costante.

MiniMax

Sono effettuate D chiamate ricorsive, con profondità massima pari a D, alcune costanti e parametri.

Iterative Deepening

Sono utilizzati parametri e elementi costanti ($O(1)$).

Cenni sull'algoritmo Monte Carlo

Nella ricerca dell'algoritmo di ottimizzazione, come evoluzione dell'approccio mostrato nelle sezioni precedenti, è stata valutata l'implementazione di un algoritmo di tipo Monte Carlo che, per ogni mossa possibile, esegue la mossa ed effettua un numero G di partite facendo mosse casuali e assegnando un punteggio cumulativo in funzione del risultato della partita (nel caso pessimo $O(MN)$ mosse eseguite per completare il ciclo di valutazione).
Uno dei problemi incontrati è stato relativo alla necessità di "salvare" risultati intermedi in quanto se il tempo a disposizione non è sufficiente per eseguire le $O(GMN)$ operazioni, si incorre nell'errore TimeOut perdendo la partita. Per ovviare a questo problema, la soluzione proposta è eseguire un numero G di partite limitato e ogni volta che sono state simulate G partite, si salva il risultato parziale in un array di punteggio in corrispondenza posizionale 1-1 con l'array di mosse disponibili. In questo modo, più partite casuali sono giocate, più si accumulano i punteggi totali realizzati giocando una delle mosse, aggiornando la mossa migliore ogni G cicli a seconda dei risultati cumulativi.
Un altro dei problemi riscontrati è ancora il posizionamento di `checkTime()` che è stato annidato nel ciclo più interno per essere eseguito con frequenza elevata.
La strategia di tipo Monte Carlo è stata implementata e testata, ma i risultati, seppur soddisfacenti in termini di mosse effettuate, non hanno migliorato i risultati ottenuti con le ottimizzazioni AlphaBeta, ordinamento delle mosse e Iterative Deepening. Pertanto non è risultato la scelta ottimale e non è stato proposto come soluzione.

Prospettive future di miglioramento

In questa sezione, sono evidenziate alcune attività che a nostro avviso potrebbero incrementare il risultato della soluzione da noi proposta o implementazioni evolute e più complesse.

Ottimizzazione della funzione di valutazione

Tra le attività di ottimizzazione, è di primaria importanza testare e affinare sperimentalmente i punteggi assegnati durante l'esecuzione della funzione di valutazione, anche se per profondità e quindi complessità elevate, l'ottimizzazione del punteggio risultante dalla funzione di valutazione fornisce a nostro avviso un miglioramento incrementale rispetto alla maggiore complessità del modello di valutazione o dei test sperimentali necessari.

Approccio con apprendimento automatico

L'algoritmo Monte Carlo si presta bene come step iniziale di uno sviluppo più ampio nel campo dell'apprendimento automatico per il problema valutato: i punteggi ottenuti effettuando combinazioni casuali di mosse potrebbero essere utilizzati come base di partenza per un iniziale apprendimento per la scelta ottimale, che nelle epoche successive potrebbero essere integrate con altre strategie di scelta ottimale.