

# MNK-GAME

Relazione

**Cheikh Ibrahim · Zaid**

Matricola: 0000974909

**Xia · Tian Cheng**

Matricola: 0000975129

Anno accademico

2020 — 2021

Corso di Algoritmi e Strutture Dati  
Alma Mater Studiorum · Università di Bologna

## Introduzione

Il progetto MNK-Game consiste nella realizzazione di un programma in grado di giocare a una versione generalizzata del Tris.

L'algoritmo deve avere, come requisito minimo, la capacità di effettuare scelte qualitativamente accettabili in tempi ragionevoli.

La criticità maggiore risiede nella valutazione dei possibili scenari che crescono esponenzialmente nel progredire del gioco, rendendo impossibile la risoluzione del problema tramite forza bruta.

## Scelte progettuali

### Classi implementate

Le classi implementate dall'algoritmo sono le seguenti:

<b>OurPlayer</b>	Implementa l'interfaccia <b>MNKPlayer</b>
<b>Node</b>	Rappresenta un nodo dell'albero di gioco
<b>GameTree</b>	Contiene l'albero di gioco e implementa i metodi per manipolarlo
<b>Matrix</b>	Rappresenta una configurazione della griglia di gioco
<b>BoardStatus</b>	Permette di ricavare informazioni su una configurazione di gioco
<b>Coord</b>	Rappresenta una coordinata
<b>EstimatedPosition</b>	Descrive una possibile mossa quantificata da un punteggio

## Funzionamento generale

Data l'impossibilità di generare tutti i possibili scenari, si rende necessario trovare e valutare solo le mosse realmente proficue tramite funzioni euristiche. Inoltre, bisogna limitare la generazione in altezza dell'albero per mantenere accettabile il tempo di risposta dell'algoritmo, per questo motivo è necessario valutare lo stato delle foglie contenenti configurazioni di gioco intermedie ed estendere, quando necessario, i nodi non ancora terminali.

L'algoritmo implementato genera l'albero iniziale con un'altezza dispari, in questo modo un cammino foglia-radice contiene un numero equo di mosse del giocatore e dell'avversario, rendendo la valutazione euristica più bilanciata. Per la stessa ragione, anche l'estensione dei nodi intermedi deve mantenere l'altezza dell'albero dispari.

Inoltre, il numero di figli per ciascun nodo viene limitato con un valore di soglia per evitare una crescita esponenziale incontrollata durante la generazione.

Per valutare le configurazioni di gioco intermedie, si utilizzano punteggi ricavati dalle informazioni riguardanti il numero di possibili allineamenti dei due giocatori.

## Quantificazione degli allineamenti

### Descrizione

Per quantificare i possibili allineamenti del giocatore o dell'avversario, viene utilizzato il metodo `getScoresArray` della classe `BoardStatus` che implementa un algoritmo basato sulla programmazione dinamica che prende in input il vettore  $I[0..n-1]$  contenente una configurazione di gioco rispetto ad una riga/colonna/diagonale e restituisce il vettore  $S[0..n-1]$ , ove  $S[i]$  contiene la tupla di interi:

(celle allineabili, mosse necessarie, punto inizio allineamento)

che rappresentano le informazioni sulla possibile mossa all' $i$ -esima cella.

Le seguenti equazioni di ricorrenza descrivono le varie casistiche previste dall'algoritmo:

$$S[0] \leftarrow \begin{cases} (0, 0, -1) & \text{se } I[0] = \text{OPPONENT} \\ (1, 0, -1) & \text{se } I[0] = \text{PLAYER} \\ (1, 1, -1) & \text{se } I[0] = \text{FREE} \end{cases}$$
$$S[i] \leftarrow \begin{cases} (0, 0, -1) & \text{se } I[i] = \text{OPPONENT} \\ (S[i-1].aligned + 1, S[i-1].moves, -1) & \text{se } S[i-1].aligned < K \text{ AND } I[i] = \text{PLAYER} \\ (S[i-1].aligned + 1, S[i-1].moves + 1, -1) & \text{se } S[i-1].aligned < K \text{ AND } I[i] = \text{FREE} \end{cases}$$
$$\text{se } I[i] = \text{PLAYER} \Rightarrow S[i] \leftarrow \begin{cases} (K, S[i-1].moves - 1, i - (K - 1)) & \text{se } I[i - K] = \text{FREE} \\ (K, S[i-1].moves, i - (K - 1)) & \text{se } I[i - K] = \text{PLAYER} \end{cases}$$
$$\text{se } I[i] = \text{FREE} \Rightarrow S[i] \leftarrow \begin{cases} (K, S[i-1].moves, i - (K - 1)) & \text{se } I[i - K] = \text{FREE} \\ (K, S[i-1].moves + 1, i - (K - 1)) & \text{se } I[i - K] = \text{PLAYER} \end{cases}$$

Per trovare il numero di mosse ottimali per ciascuna cella, alla posizione  $i$ -esima, avviene una fase di propagazione del punteggio, basato sul seguente pseudocodice:

```
// Se all' $i$ -esima posizione è possibile allinearne  $K$ :  
for  $j \leftarrow 1$  to  $K - 1$  do  
    // Le mosse precedenti sono migliori, non serve propagare  
    if  $S[i-j].aligned = K$  AND  $S[i-j].moves < S[i].moves$  then  
        | break  
    end  
    // Propagazione  
     $S[i-j] = S[i]$   
end
```

Per calcolare i valori in relazione all'avversario è sufficiente invertire `PLAYER` e `OPPONENT`.

### Costo computazionale

L'algoritmo deve necessariamente iterare per intero il vettore  $I$  con un costo di  $\Theta(n)$ , in aggiunta, per ciascun ciclo c'è la possibilità di dover propagare la tupla calcolata alle celle antecedenti. Nel caso pessimo, quindi, si ha un costo computazionale di  $O(n \cdot K)$ , ovvero quando la propagazione avviene per ogni posizione.

## Generazione tuple

### Descrizione

La classe `BoardStatus` implementa le funzioni `generateMovesToWinAt` e `generateGlobalMovesToWin` che utilizzano il metodo `getScoresArray` per generare le tuple rispetto ad una riga, colonna o diagonale.

In particolare:

- `generateMovesToWinAt` prende in input una coordinata e genera le tuple della riga, colonna e diagonali che passano per quel punto
- `generateGlobalMovesToWin` genera le tuple per tutte le posizioni rispetto a tutte le direzioni

L'output viene memorizzato in delle matrici interne alla classe per evitare di dover rigenerare le tuple nel caso si dovesse accedere a celle adiacenti.

### Costo computazionale

La funzione `generateMovesToWinAt`, nel caso pessimo, ha costo  $O(\max\{M, N\} \cdot K) = O(MK + NK)$  dato dalla necessità di iterare tutte le direzioni rispetto ad una coordinata (quindi ha maggior peso la riga/colonna con più celle).

Nel caso ottimo, invece, ha costo  $\Theta(1)$ , ovvero quando le tuple sono già state generate e memorizzate nelle matrici.

La funzione `generateGlobalMovesToWin`, nel caso pessimo, ha costo  $O(MNK)$  perché bisogna iterare l'intera griglia di gioco.

Analogamente, nel caso ottimo, il costo è  $\Theta(1)$ .

## Scenari possibili di vittoria

### Descrizione

La classe `BoardStatus` implementa le funzioni `getAllPossibleWinningScenariosCount` e `getAllPossibleWinningScenariosCountAt` che restituiscono un vettore contenente all' $i$ -esima posizione il numero di possibili modi per vincere che necessitano  $i$  mosse.

Per calcolare questi valori vengono usate le tuple, sfruttando l'indice di inizio allineamento.

### Costo computazionale

La funzione `getAllPossibleWinningScenariosCount` ha complessità  $O(MN)$  dovendo iterare per intero le matrici contenenti le tuple.

La funzione `getAllPossibleWinningScenariosCountAt` ha complessità  $O(\max\{M, N\})$  in quanto deve iterare la riga, la colonna e le diagonali che passano per un punto.

## Euristica su configurazioni intermedie

### Descrizione

La classe `GameTree` implementa il metodo `setHeuristicScoreOf` che imposta ad un dato nodo un punteggio euristico.

L'implementazione prevede di richiamare, sia per il giocatore che per l'avversario, la funzione `getAllPossibleWinningScenariosCount` e considerare gli scenari che necessitano da 1 a  $x$  mosse con i quali calcolare il punteggio finale assegnando un peso per ciascuna tipologia.

Vengono quindi gestite tre casistiche:

1. Se è il turno del giocatore e ha la possibilità di vincere immediatamente, viene assegnato il punteggio vincente
2. Se è il turno dell'avversario e ha la possibilità di vincere immediatamente, viene assegnato il punteggio perdente
3. Altrimenti viene assegnata la differenza tra il punteggio del giocatore e quello dell'avversario.

### Costo computazionale

La funzione `setHeuristicScoreOf` ha costo, nel caso pessimo, di  $O(MNK)$  dato dalla chiamata al metodo `generateGlobalMovesToWin`.

## Ricerca della mossa successiva

### Descrizione

Nella classe `GameTree` viene implementato il metodo `getAdjacency` che, a partire da un nodo dell'albero, scansiona le celle vuote adiacenti a tutte le mosse effettuate e restituisce una coda con priorità contenente tutte le mosse analizzate ordinate in base all'importanza.

Per marcare le celle visitate si utilizza una hash table che associa ad una coordinata un booleano, per evitare di dover allocare un'intera matrice di dimensione  $\Theta(MN)$ , avendo comunque un tempo di accesso medio di  $O(1)$ .

Le mosse vengono valutate utilizzando il numero di mosse mancanti alla vittoria e si basano sul seguente ordine di priorità:

Priorità 1 (critico)	Mossa immediatamente vincente
Priorità 2 (critico)	Blocca una mossa immediatamente vincente dell'avversario
Priorità 3 (critico)	Piazza una mossa che crea un vicolo cieco per l'avversario (una mossa che apre più scenari di vittoria immediata)
Priorità 4 (critico)	Blocca la creazione di un vicolo cieco da parte dell'avversario
Altrimenti	Piazza una mossa che aumenta un allineamento del giocatore dando priorità ad allineamenti più lunghi e in grado di bloccare la sequenza maggiore dell'avversario

## Costo computazionale

Il costo computazionale è  $O(h(MK + NK + \log h))$ , dove  $h$  è l'altezza dell'albero di gioco. Il costo è dato dal ciclo di costo  $\Theta(h)$  che ripercorre l'albero fino alla radice e per ciascuna iterazione analizza al più un numero costante di 8 celle (tutte le possibili direzioni). Il costo maggiore all'interno del ciclo è dato dalla chiamata alla funzione `generateMovesToWinAt` di costo  $O(MK + NK)$ .

Inoltre ogni mossa elaborata viene inserita in una coda con priorità che ha un costo computazionale logaritmico rispetto alla dimensione della coda. Ipotizzando che ad ogni iterazione si inserisca sempre nella coda, il costo è il seguente (utilizzando l'approssimazione di Stirling del fattoriale):

$$\sum_{i=1}^h \log i = \log h! = \log \sqrt{2\pi h} \left(\frac{h}{e}\right)^h = \cancel{\log \sqrt{2\pi}} + \cancel{\log \sqrt{h}} + h \log \frac{h}{e} = O(h \log h)$$

## Generazione dell'albero di gioco

### Descrizione

Il metodo `createTree` nella classe `GameTree` prende in input un nodo dell'albero e un intero rappresentante il numero di livelli da generare e crea l'albero di gioco radicato in quel nodo. Il funzionamento si basa sul seguente pseudocodice:

```
Function createTree(nodo, depth)
  generateMovesToWinAt(coordinate della mossa nel nodo)
  if partita terminata then
    | imposta punteggio reale
  else if  $depth \leq 0$  then
    | setHeuristicScoreOf(nodo)
  else
    | PriorityQueue mosse  $\leftarrow$  getAdjacency(a partire dal nodo)
    | while ci sono mosse promettenti do
    |   | createTree(nodo da visitare, depth-1)
    |   end
  end
end
```

La selezione delle mosse promettenti è basato sul seguente criterio:

- Se la mossa è critica, valuto tutte quelle equivalenti (ad esempio se c'è la possibilità di vincere immediatamente, non è necessario valutare mosse di tipologia diversa)
- Se la mossa non è critica ne valuto al più un numero fissato.

## Costo computazionale

La funzione si basa sulla seguente equazione di ricorrenza:

$$T(\text{depth}) = \begin{cases} MK + NK & \text{se partita terminata} \\ (\cancel{MK + NK}) + MNK & \text{se depth} \leq 0 \\ (\cancel{MK + NK}) + h(MK + NK + \log h) + p(M + N) + p \log q + pT(\text{depth} - 1) & \text{altrimenti} \end{cases}$$

Dove  $h$  è l'altezza dell'albero,  $p$  è il numero di mosse promettenti (iterazioni del ciclo **while**) e  $q$  è il numero di possibili mosse (dimensione della coda con priorità).

Assumendo che il numero di iterazioni  $p$  sia mediamente coerente con il valore soglia fissato, possiamo trattarlo come costante. Quindi la complessità computazionale del caso ricorsivo è:

$$\begin{aligned} h(MK + NK + \log h) + p(\cancel{M + N}) + p \log q + pT(\text{depth} - 1) = \\ = h(MK + NK) + h \log h + \log q + pT(\text{depth} - 1) \end{aligned}$$

Risolvendo per iterazione, otteniamo che il costo è:

$$p^{\text{depth}} (h(MK + NK) + h \log h + \log q + \cancel{MNK})$$

Dimostriamo che  $q = O(h)$ . Fissiamo due successioni  $q_n$  e  $h_n$ , rappresentanti rispettivamente il numero di possibili mosse e l'altezza dell'albero nel progredire della partita, e proviamo che  $\exists c > 0, n_0 \geq 0$  t.c.  $\forall n \geq n_0 : q_n \leq c \cdot h_n$ .

Intuitivamente l'altezza dell'albero di gioco cresce in maniera lineare nel corso della partita, mentre il numero di possibili mosse cresce per poi diminuire una volta raggiunto un punto di "saturazione".

Quindi, ponendo  $c = 1$ , esiste un  $n_0$  tale che  $\forall n \geq n_0 : q_n \leq h_n$ , provando che  $q = O(h)$  e ottenendo che la complessità è:

$$p^{\text{depth}} \cdot h(MK + NK + \log h).$$

Il costo computazionale della funzione **createTree** è quindi  $O(p^{\text{depth}} \cdot h(MK + NK + \log h))$ , dove  $h$  è l'altezza dell'albero,  $p$  è il numero medio di figli per nodo e  $\text{depth}$  è il numero di livelli da generare.

## Estensione dell'albero di gioco

### Descrizione

La funzione **extendLeaves** permette di estendere di un determinato numero di livelli tutte le foglie dell'albero che contengono configurazioni di gioco intermedie.

Per estendere un nodo viene usata la funzione ausiliaria **extendNode** che prende in input un nodo e il numero di livelli ulteriori da generare e richiama la funzione **createTree** su quel nodo.

### Costo computazionale

La funzione `extendNode` ha costo computazionale  $O(p^{\text{depth}} \cdot h(MK + NK + \log h))$  dato dalla chiamata a `createTree`.

Il costo, nel caso pessimo, di `extendLeaves` è quindi  $O(n_f \cdot (p^{\text{depth}} \cdot h(MK + NK + \log h)))$  dove  $n_f$  è il numero di foglie dell'albero.

Poiché l'albero viene sempre esteso per un numero costante di livelli e ciascun nodo ha in media un numero fissato di figli, il costo è  $O(h(MK + NK + \log h))$ .

### Operazioni sull'albero di gioco

#### Descrizione

Le operazioni previste dalla classe `GameTree` sull'albero di gioco sono le seguenti:

Funzione	Descrizione
<code>generate</code>	Genera l'albero di gioco iniziale
<code>setOpponentMove</code>	Sposta la radice dell'albero al figlio che contiene la mossa corrispondente a quella dell'avversario. Eventualmente estende l'albero di gioco e valuta con <b>AlphaBeta Pruning</b> .  Nel caso la mossa non fosse prevista, viene creato un nuovo nodo contenente quella mossa e generato il sottoalbero radicato in tale nodo.
<code>nextMove</code>	Seleziona e sposta la radice al figlio contenente la mossa migliore del giocatore. Eventualmente estende l'albero di gioco e valuta con <b>AlphaBeta Pruning</b>

### Costo computazionale

Funzione	Costo computazionale (caso pessimo)
<code>generate</code>	$O(h(MK + NK + \log h))$ dato dalla chiamata a <code>createTree</code>
<code>setOpponentMove</code>	$O(h(MK + NK + \log h))$ dato dalla chiamata a <code>extendLeaves</code> o <code>extendNode</code>
<code>nextMove</code>	$O(h(MK + NK + \log h))$ dato dalla chiamata a <code>extendLeaves</code>

**Nota:** Il costo computazionale per le tre funzioni risulta identico ma bisogna ricordare che differiscono di una costante non sempre uguale.



## Costo computazione di AlphaBeta Pruning

Il costo computazionale di **AlphaBeta Pruning** è, nel caso pessimo,  $O(m^d)$  dove  $m$  è il numero medio di figli per nodo e  $d$  è la profondità di ricerca. Poiché nell'algoritmo sia la profondità di ricerca che il numero medio di figli è mantenuta costante, si ha che il costo dell'esecuzione di **AlphaBeta Pruning** risulta costante su tutte le chiamate.

In aggiunta, per come viene generato l'albero, si ha che **AlphaBeta Pruning** elabora sempre il nodo che, secondo la valutazione euristica, è il più promettente, permettendo di ottenere, in molti casi, un aumento quadratico delle prestazioni.

## Interfaccia MNKPlayer

### Descrizione

L'interfaccia **MNKPlayer** viene implementata dalla classe **OurPlayer** che istanzia un oggetto **GameTree** su cui esegue le operazioni per generare e manipolare l'albero di gioco.

La funzione **selectCell**, quindi, restituisce la mossa estraendola dall'oggetto **GameTree** basandosi sul seguente pseudocodice:

```
MNKCell mossaScelta ← null
if gameTree.isEmpty() then
  if gioco per primo then
    mossaScelta ← centro della griglia
    gameTree.generate(mossaScelta)
  else
    gameTree.generate(mossa dell'avversario)
    mossaScelta ← gameTree.nextMove()
  end
else
  gameTree.setOpponentMove(mossa dell'avversario)
  mossaScelta ← gameTree.nextMove()
end
```

### Costo computazionale

La funzione **selectCell** ha costo  $O(h(MK + NK + \log h))$  dato dalla chiamata a **generate** o **setOpponentMove** o **nextMove**.

## Conclusione

L'algoritmo implementato è in grado di giocare in modo accettabile sulle configurazioni note di MNK-Game. Sono presenti criticità date dal fatto che l'albero di gioco è generato in modo parziale in altezza, rendendo possibile il verificarsi di scenari in cui non è possibile prevedere, con sufficiente anticipo, vicoli ciechi, ovvero situazioni di sconfitta certa.

Un possibile miglioramento dell'algoritmo è quello di continuare la generazione dell'albero di gioco anche durante l'attesa della mossa dell'avversario.

Un'ulteriore possibilità è quella di generare i sottoalberi dei vari nodi in modo parallelo, utilizzando, ad esempio, i *thread*.