

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 algoritmo in grado di giocare a una versione generalizzata del Tris.

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

Contemporaneamente però, l'algoritmo deve avere, come requisito minimo, la capacità di effettuare scelte qualitativamente accettabili.

Scelte progettuali

Classi implementate

Le classi implementate dall'algoritmo sono le seguenti:

OurPlayer	Implementa l'interfaccia MNKPlayer
Node	Rappresenta un nodo dell'albero di gioco
GameTree	Contiene l'albero di gioco e implementa i metodi per manipolarlo
Matrix	Rappresenta una configurazione della griglia di gioco
BoardStatus	Permette di ricavare informazioni su una configurazione di gioco
Coord	Rappresenta una coordinata
EstimatedPosition	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, rendendo quindi necessario valutare lo stato delle foglie contenenti configurazioni di gioco intermedie.

(((((Le funzioni euristiche di questo algoritmo si basano sul numero di allineamenti effettivi e possibili del giocatore e dell'avversario)))))).

Quantificazione degli allineamenti

Descrizione

Per quantificare i possibili allineamenti del giocatore e 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 la configurazione di gioco di ciascuna cella rispetto ad una direzione rappresentata da **PLAYER**, **OPPONENT**, **FREE** e restituisce il vettore $S[0..n-1]$, ove $S[i]$ contiene la tupla di interi:

(celle allineabili, mosse necessarie **alla vittoria**, 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:

$$S[0] \leftarrow \begin{cases} (0, 0, -1) & \text{se } I[0] = \text{OPPONENT} \\ (1, 0, -1) & \text{se } I[0] = \text{PLAYER} \\ (1, 1, -1) & \text{se } I[0] = \text{FREE} \end{cases}$$

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

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

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

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

```

for  $j \leftarrow i - 1$  to  $i - K + 1$  do
    // L'allineamento viene interrotto da una mossa dell'avversario
    if  $I[j] = \text{OPPONENT}$  then
        | break
    end
    // Le mosse precedenti sono migliori, non serve propagare
    if  $S[j].aligned = K$  AND  $S[j].moves < S[i].moves$  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 $\Theta(I.length)$, 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 $O(I.length \cdot K)$, ovvero **quando** la propagazione avviene per ogni posizione.

Generazione **punteggi 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 **i punteggi le tuple** della riga, colonna e diagonali che passano per quel punto
- `generateGlobalMovesToWin` genera **il punteggio 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 **gli score sono già stati generati e memorizzati le tuple già stati state generate e memorizzate** nelle matrici.

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

Euristica su configurazioni intermedie

Descrizione

La classe `GameTree` implementa il metodo `setHeuristicScoreOf` che prende in input un nodo dell'albero, un oggetto `BoardStatus` e un flag per indicare chi deve eseguire la prossima mossa e imposta a quel nodo un punteggio euristico.

L'implementazione prevede di generare il numero di mosse necessarie per vincere a tutte le celle e di ricavare, sia per il giocatore che per l'avversario, il numero di possibili scenari vincenti che necessitano di piazzare da 1 a n mosse e con queste calcola 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 `BoardStatus` 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 valutate Le mosse vengono valutate utilizzando il numero di mosse mancanti alla vittoria e si basano sul seguente ordine di priorità:

Priorità 1	Mossa immediatamente vincente
Priorità 2	Blocca una mossa immediatamente vincente dell'avversario
Priorità 3	Piazza una mossa che crea un vicolo cieco per l'avversario (ovvero una mossa che apre più scenari di vittoria immediata)
Priorità 4	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

Le mosse con priorità ≤ 4 sono considerate critiche in quanto permettono di aprire scenari a vittoria o sconfitta certa.

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 scorre per intero l'albero 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 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 genera 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 ≤ 0 then
        | imposta punteggio euristico
    else
        PriorityQueue mosse ← 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, valuto al più un numero fissato.

Costo computazionale

La funzione si basa sulla seguente equazione di ricorrenza:

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

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

Assumendo che il numero di iterazioni p sia in media coerente con il valore soglia stabilito,

possiamo trattarlo come una costante, quindi la complessità computazionale della terza equazione del caso ricorsivo è:

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

Risolvendo per iterazione, otteniamo che il costo è:

$$\begin{aligned}
&= (h(MK + NK) + h \log h + \log q) \cdot (\text{depth} - 1) + \cancel{MNK} = \\
&= \text{depth} \cdot h(MK + NK) + \text{depth}(h \log h + \log q)
\end{aligned}$$

FIXAREEE

Dimostriamo, applicando la definizione in modo diretto, che $\log q = O(h \log h)$, ovvero, vogliamo provare che $\exists c > 0, n_0 \geq 0$ t.c. $\forall n \geq n_0 : \log q \leq c \cdot (h \log h)$.

Possiamo porre $c = 1$ e mostrare che per un $n_0 \geq 0$ si ha che $\log q \leq h \log h$.

Ricordando che h è l'altezza dell'albero di gioco e q è il numero di possibili mosse, è facile convincersi che il numero di mosse aumenta con l'aumentare dell'albero e inizia a diminuire una volta raggiunto un punto di "saturazione", quindi, si è provato che $\log q = O(h \log h)$.

Dimostriamo, applicando la definizione, 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 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 $q = O(h)$.

La complessità ottenuta è quindi:

$$\begin{aligned} depth \cdot h(MK + NK) + depth(h \log h) &= \\ &= depth \cdot h(MK + NK + \log h). \end{aligned}$$

Il costo computazionale della funzione `createTree` è $O(\text{depth} \cdot h(MK + NK + \log h))$, dove h è l'altezza dell'albero e depth è il livello di profondità fino al quale generare il numero di livelli da generare.

Di seguito, le robe brutte:

Valutazione mosse "interessanti"

DA RIFARE TUTTA DA CAPO :)

Per ricercare mosse successive a partire da un nodo il cui stato di gioco non sia terminale, viene richiamata la funzione `getInterestingPositions` che restituisce una coda con priorità di posizioni organizzate in ordine crescente di punteggio.

La funzione `getInterestingPositions(Node node, BoardStatus board)` prende in input un nodo ed effettua la scansione di tutte le celle adiacenti alle posizioni già marcate e di queste ne calcola il punteggio euristico relativo sia al giocatore che all'avversario e li inserisce nella coda.

Quindi, all'interno della funzione `createTree` vengono estratte dalla coda tutte le mosse con un punteggio minore o uguale a `SCORE.THRESHOLD` ed eventualmente altre mosse fino al raggiungimento del valore minimo `MIN_EVAL` o fino a coda vuota.

Interfaccia MNKPlayer

DA AGGIUSTARE (POSIZIONE)

L'interfaccia `MNKPlayer` viene implementata dalla classe `OurPlayer` che contiene, tra gli attributi, un oggetto `GameTree`.

La funzione `selectCell`, quindi, restituisce la mossa da eseguire basandosi sullo stato dell'oggetto `GameTree`:

```
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
```

Albero di gioco (FORSE SCOMPARE)

Per la valutazione e la scelta della mossa da eseguire viene implementato un albero di gioco gestito nella classe `GameTree`.

Ciascun nodo dell'albero contiene:

- La descrizione del nodo rappresentato
- Il riferimento al nodo padre e una lista concatenata contenente i figli
- Un punteggio euristico

Generazione parziale dell'albero

Il numero dei nodi generati dell'albero di gioco cresce in modo esponenziale all'aumentare dell'altezza; per tale ragione è impossibile generare l'intero albero per configurazioni di gioco con un elevato numero di celle.

La generazione dell'albero deve essere quindi limitata sia in altezza che nel numero di nodi da elaborare.

All'interno della classe `GameTree` sono quindi presenti le costanti `MAX_DEPTH` e `MIN_EVAL` rispettivamente l'altezza massima generabile e il numero indicativo di figli di ciascun nodo (variabile in base al numero di scenari favorevoli o sfavorevoli).

Al termine di ogni fase di generazione dell'albero, la struttura viene elaborata dall'algoritmo **AlphaBeta Pruning** per propagare i punteggi delle foglie alla radice ed eventualmente tagliare determinati sottoalberi inconvenienti.

CONTINUARE

Per mantenere costante in altezza l'albero, in seguito alla selezione di una mossa, viene richiamata la funzione `extendLeaves(Node root)` che estende le foglie dell'albero radicato nel nodo in input.

Conclusione

Il progetto è stato caruccio ma non lo rifarei.