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.
Contemporaneamente, l'algoritmo deve avere la capacità di effettuare scelte qualitativamente accettabili.

Scelte progettuali

Interfaccia `MNKPlayer`

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

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.
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.

Euristica sui punteggi

Tramite la classe `BoardStatus` è possibile rappresentare ed effettuare la stima del possibile esito di una configurazione di gioco.
Tale punteggio euristico viene calcolato tramite un algoritmo basato sulla programmazione dinamica che prende in input il vettore $M[0..n-1]$ contenente la configurazione di gioco di ciascuna cella rappresentata da `PLAYER`, `OPPONENT`, `FREE` e restituisce il vettore $S[0..n-1]$, ove $S[i]$ contiene una coppia di interi rappresentanti il numero di celle allineabili e il numero di mosse necessarie per vincere se si seleziona l'i-esima cella. Le seguenti equazioni di ricorrenza descrivono le varie casistiche previste dall'algoritmo:

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

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

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

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

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

```
for j ← i − 1 to i − K + 1 do
    // L'allineamento viene interrotto da una mossa dell'avversario
    if M[j] = 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
    S[i − j] = S[i]
end
```

Valutazione mosse "interessanti"

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 lista 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.

Conclusione

Il progetto è stato caruccio ma non lo rifarei.