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

Titoloooooooooooooooooooooooooooooooooooooooooooooo

Data l'impossibilità di generare tutti i possibili scenari, si rende necessario realizzare una funzione euristica in grado di prendere in considerazione solo le mosse realmente proficue.
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 mancanti 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:

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

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

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

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

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

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

Costo computazionale

L'algoritmo deve necessariamente iterare per intero il vettore I con un costo di $\Theta(I.\text{length})$, 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 $\text{O}(I.\texttt{length} \cdot K)$, ovvero la propagazione avviene per ogni posizione.

Generazione punteggi

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 della riga, colonna e diagonali che passano per quel punto
- `generateGlobalMovesToWin` genera il punteggio per tutte le posizioni rispetto a tutte le direzioni

L'output viene memorizzato in una matrice interna 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 $\text{O}(\texttt{max}\{M, N\} \cdot K) = \text{O}(MK + NK)$ 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 nelle matrici.
La funzione `generateGlobalMovesToWin`, nel caso pessimo, ha costo $\text{O}(MNK)$ perché bisogna iterare l'intera griglia di gioco.

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.
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.

Punteggio euristico a configurazioni di gioco non terminali

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 i punteggi per 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 O(MNK) dato dalla chiamata al metodo `generateGlobalMovesToWin`.

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
```

Conclusione

Il progetto è stato caruccio ma non lo rifarei.