Report progetto MNK-game

Michele Cucci
michele.cucci@studio.unibo.it
0000874481

Tommaso Bindi
tommaso.bindi2@studio.unibo.it
0000881736

Università di Bologna, Laurea Triennale in Informatica

Febbraio 2022

Indice

1 Overview

1.1 Panoramica del gioco

Mnk-game è un gioco da tavolo astratto in cui due giocatori, a turno, eseguono una mossa su una tavola m x n.
Il vincitore è il giocatore che per primo ottiene k mosse proprie in una riga, in orizzontale, in verticale o in diagonale. In caso nessun giocatore dovesse ottenere k mosse allineate si avrà un pareggio. Per esempio, il tris è il gioco 3 x 3 con k=3.

1.2 L'obiettivo

Lo scopo del progetto è lo sviluppo di un'intelligenza artificiale in Java per l'implementazione di un giocatore di m,n,k-game. L'implementazione della intelligenza artificiale dovrà cercare di effettuare una scelta intelligente per una possibile mossa ottima/quasi-ottima/**accettabile** da giocare.

1.3 MNKGame

L'interfaccia di gioco su cui la nostra intelligenza artificiale può operare è MNKGame. Questa interfaccia lascia un intervallo di tempo di 10 secondi all'intelligenza per far si che questa esprima la miglior risposta alla mossa effettuata dal giocatore avversario.

2 Analisi del problema

È importante considerare che nel problema del mnk-game non è possibile trovare una strategia che permetta al secondo giocatore di vincere. Questo argomento è noto come furto di strategia.

2.1 Furto di strategia

Il primo giocatore inizia eseguendo una mossa arbitraria e si dia per assurdo che il secondo giocatore abbia una strategia vincente. Allora si può convertire questa strategia vincente in una strategia vincente per il primo giocatore nel seguente modo:

1. il primo giocatore gioca come prima mossa una mossa casuale.
2. Dalla seconda mossa il primo giocatore gioca scegliendo la propria mossa applicando la strategia vincente ad una partita ipotetica, in cui non sia mai stata fatta la mossa del punto 1 e invece la prima mossa sia stata fatta dal secondo giocatore.
   In tale situazione il primo giocatore diventa il secondo e può dunque applicare la strategia vincente.
3. Qualora la strategia vincente disponesse di muovere proprio nella prima casella, il primo giocatore può invece muovere in una nuova casella libera scelta a caso, e questa sarà la casella ignorata d'ora in poi al punto 2.

Per ciò il primo giocatore vincerà la partita ipotetica di cui al punto 2 e così anche la partita reale, poiché avere una pedina in più in gioco non porta svantaggio. In altre parole, quella appena descritta è una strategia vincente per il primo giocatore. Quindi, l'esistenza di una strategia vincente per il secondo giocatore implica l'esistenza di una strategia vincente per il primo giocatore, il che porta ad una contraddizione.

2.2 Valori di gioco per l'mnk-game

Un'altra nozione importante da stabilire è cosa è un "weak (m,n,k) game".
Con "weak-(m,n,k)" si intende una tabella in cui k mosse in fila del secondo giocatore non portano a terminare il gioco con una vittoria del secondo giocatore.
Se un gioco weak-(m,n,k) termina con un pareggio, anche la diminuzione di **m** o **n** o l'aumento di **k** risulteranno in un pareggio.
Al contrario, se weak-(m,n,k) termina in una vittoria, qualsiasi weak-(m,n,k) maggiore terminerà in una vittoria.

2.2.1 Soluzioni specifiche conosciute

Di seguito si elencano weak-(m,n,k) speciali che producono sempre lo stesso risultato.

- Con k = 1 e k = 2 vince sempre il primo giocatore.

- Con k = 3 è un pareggio per (3,3,3) (vedi Tris) e (m,n,3) è un pareggio se m < 3 o n < 3. altrimenti è vittoria per il primo giocatore se m ≥ 3 e n ≥ 4 o m ≥ 4 e n ≥ 3.
- (5,5,4) è un pareggio, il che significa che (m,n,4) è un pareggio per m ≤ 5 e n ≤ 5, e (6,5,4) è una vittoria, il che significa che (m,n,4) è una vittoria per m ≥ 6 e n ≥ 5 o m ≥ 5 e n ≥ 6.
- Se k ≥ 3 e m < k o n < k, la partita è patta.
- Se k ≥ 8 è un pareggio: è stato dimostrato che quando k è almeno 8, il secondo giocatore può forzare il pareggio anche su una scacchiera infinita, e quindi su qualsiasi scacchiera finita. Questo significa che quando il tabellone è infinito il gioco andrà avanti per sempre con gioco perfetto, mentre se è finito il gioco finirà in parità.
- (9,6,6) e (7,7,6) sono entrambi pareggi tramite accoppiamenti.
- Non è noto se il secondo giocatore può forzare un pareggio quando k è 6 o 7 su una tavola infinita.

3 Scelte progettuali

Il progetto è stato sviluppato tramite l'IDE Intellij Idea con Java come linguaggio di programmazione.
Per l'implementazione del progetto sono state realizzate due classi di supporto alla classe implementativa **AiPvs.java** denominate **Game.java** e **AISearcher.java**.

3.1 AiPvs.java

AiPvs.java è la classe principale del progetto. Essa definisce la classe AiPvs come figlio della classe MNKPlayer, permettendo quindi di eseguire l'Override delle funzioni principali da richiamare.
La funzione initPlayer inizializza le due sottoclassi, Game e AISeacher inserendo i valori necessari per la creazione della partita corrente.
La funzione selectCell invece riceve in input due array contenenti le "celle" marcate e quelle non marcate restituisce in output il valore della mossa "ottimale" da giocare.
Per questa classe si è deciso di non inserire il costo computazionale poiché esso è derivante dalle altre sottoclassi, dove invece verrà mostrato.

3.1.1 Codice

```
public class AiPvs implements MNKPlayer {

    private long time;
    private Game game;
    private AISearcher searcher;

    public void initPlayer(int M, int N, int K, boolean first, int
        timeout_in_secs) {
        game = new Game(M, N, K);
        this.time = (long) timeout_in_secs * 1000;
        searcher = new AISearcher(game, time);
    }

    public MNKCell selectCell(MNKCell[] FC, MNKCell[] MC) {
        if (MC.length > 1)
            searcher.update(MC[MC.length - 2]);
        if (MC.length > 0)
            searcher.update(MC[MC.length - 1]);
        int move = searcher.iterativeDeepening();
        return new MNKCell(game.getRow(move), game.getCol(move));
    }
}
```

3.2 AISearcher.java

La classe **AISearcher** si può definire come il cuore pulsante dell'applicazione, visto che al suo interno si trovano le funzioni di ricerca della mossa migliore e di valutazione della posizione corrente.
La funzione di ricerca si basa sulla potatura alfa-beta, un metodo appartenente alla categoria di forza bruta.
L'algoritmo alfa-beta è stato ottimizzato grazie al restringimento della finestra dei punteggi interessanti, permettendo di ottenere più cutoff.
Per realizzare ciò si è scelto di implementare una ricerca *iterative deepening*, una finestra di aspirazione e una *transposition table* con *Zobrist Hashing*.
Le finestre di aspirazione ('Aspiration Windows') sono un modo per ridurre lo spazio di ricerca in una ricerca alfa-beta. La tecnica consiste nell'utilizzare un'ipotesi del valore atteso (di solito dall'ultima iterazione nell'approfondimento iterativo) e utilizzare una finestra attorno a questo come limiti alfa-beta. Poiché la finestra è più stretta, vengono raggiunti più cutoff beta e la ricerca richiede un tempo più breve. Lo svantaggio è che se il punteggio reale è al di fuori di questa finestra, è necessario effettuare una ricerca costosa.

3.2.1 Codice

```
public int iterativeDeepening() {
   startTime = System.currentTimeMillis();
   int depth = game.maxDepth();
   int bestScore = 0, bestMove = -1;
   boolean IamP1 = getGame().getCurrentPlayer() == Game.PLAYER_1;
   int alpha = Game.MIN_SCORE, beta = Game.MAX_SCORE;
   Game.IntegerPair partialScore;

   final Game backupGame = game.clone();
   try {
       //iterativeDeepening
       for (int i = 1; i <= depth; i++) {
           partialScore = findBestMove(i, alpha, beta);
           //Aspiration
           if (((partialScore.second() <= alpha && alpha !=
                Game.MIN_SCORE) || (beta != MAX_SCORE &&
                partialScore.second() >= beta)) && i > 1) {
               alpha = Game.MIN_SCORE;
               beta = Game.MAX_SCORE;
               i--;
           } else {
               alpha = partialScore.second() - 100;
               beta = partialScore.second() + 100;
           }
           if ((IamP1 && partialScore.second() >= bestScore) ||
                (!IamP1 && partialScore.second() <= bestScore)) {
               bestScore = partialScore.second();
               bestMove = partialScore.first();
           }
       }
   } catch (TimeoutException ex) {
```

```
            System.out.println("Error, move not found on time.");
        }
        this.game = backupGame;
        return bestMove;
    }
```

```
private int AlphaBeta(int depth, int alpha, int beta) throws
    TimeoutException {
      int val, hashf = hashfALPHA;
      long zobristKey = getGame().computeKey();
      timeCheck();

      EntryTT entry = transpositionTable.get(zobristKey);
      if (entry != null) {
          if (entry.flag == hashfEXACT) {
              return entry.value;
          } else if (entry.flag == hashfBETA) {
              beta = Math.min(beta, entry.value);
          } else if (entry.flag == hashfALPHA) {
              alpha = Math.max(alpha, entry.value);
          }
          if (alpha >= beta) {
              return entry.value;
          }
      }
      if (depth == 0) {
          val = evaluate();
          transpositionTable.put(zobristKey, new EntryTT(depth, val,
              hashfEXACT));
          return val;
      }
      timeCheck();
      for (int move : getGame().generateMoves()) {
          getGame().playMove(move);
          val = -AlphaBeta(depth - 1, -beta, -alpha);
          getGame().unPlayMove();
          if (val >= beta) {
              transpositionTable.put(zobristKey, new EntryTT(depth,
                  beta, hashfBETA));
              return beta;
          }
          if (val > alpha) {
              hashf = hashfEXACT;
              alpha = val;
          }
      }
      timeCheck();
      transpositionTable.put(zobristKey, new EntryTT(depth, alpha,
          hashf));
      return alpha;
  }
```

3.3 Game.java

La classe Game è lo scheletro del programma.
Nella classe vengono definiti i valori principali come la dimensioni della tabella di gioco, il numero di mosse da effettuare per vincere e una cronologia delle mosse effettuate.
Si possono trovare al suo interno varie funzioni per permettere all'*IA* di effettuare mosse virtuali per osservare il possibile proseguimento del gioco e funzioni per l'analisi delle righe/colonne/diagonali per la valutazione di una posizione.
Di particolare importanza va notato come si è deciso di implementare la matrice di gioco come un vettore per potervi accedere successivamente tramite una funzione **Iterable** per poter ottenere uno "stack" contenente tutte le mossi libere.
La classe implementa inoltre il metodo Cloneable che permette una clonazione dell'oggetto **Game** all'inizio della ricerca in modo da eseguire un veloce reset dell'oggetto stesso dopo aver trovato la mossa ottimale.
Si è deciso di formattare tutta la struttura conclusa ogni ricerca per evitare array "sporchi", eliminando tutte le prove effettuate tranne le mosse effettuate dai giocatori in precedenza. Ogni volta che viene lanciata l'IA si effettua un inserimento delle ultime due mosse effettuate.

3.4 Costi computazionali

Per l'analisi dei costi computazionali possiamo osservare i costi iniziali in fase di costruzione e poi di ricerca dell'algoritmo.
In fase di costruzione, vengono istanziati (i costi riportati tra parentesi sono sia temporali che spaziali):

1. un vettore rappresentante la griglia di gioco $\theta(s)$;
2. un vettore rappresentante la cronologia delle posizioni $\theta(s)$;
3. variabili in quantità costante $\theta(1)$.

Mentre in fase di ricerca:

1. ripristino della copia alla fine della ricerca $\theta(s)$;
2. discesa dell'*iterative deepining O(depth) + funzione bestNodeSearch(n)*;
3. ciclo all'interno della funzione bestNodeSearch su nodi ancora disponibili tramite chiamata ricorsiva **AlphaBeta**
4. purtroppo non è possibile calcolare una stima accurata del numero *n(d)* dei nodi ispezionati dalla funzione ricorsiva bestNodeSearch + AlphaBeta con limite di profondità d

Ciò porta al costo computazionale della funzione iterative deepening a $\theta(depth * bestNodeSearch + AlphaBeta)$;

4 Conclusioni

Anche se è stato in grado di giocare in maniera più che ottimale contro un giocatore umano, purtroppo ci è risultato impossibile quantificare le prestazioni complessive.
L'algoritmo è in grado di giocare contro avversari indipendentemente dalla dimensione della matrice, ma purtroppo la complessità dell'algoritmo implementato ci ha reso molto difficile il test approfondito dell'applicazione. Ad oggi l'applicazione è capace di reagire ad attacchi ma purtroppo è ancora cieca davanti a mosse conclusive.