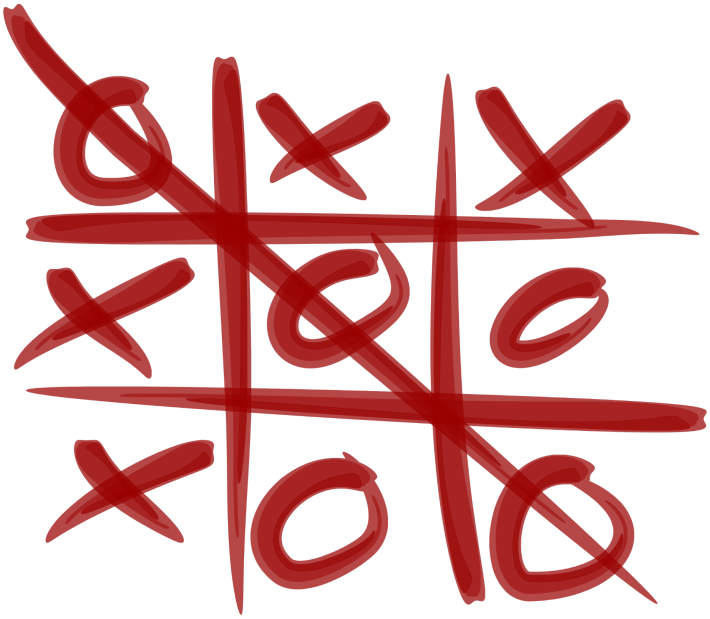
37635 - ALGORITMI E STRUTTURE DI DATI

Relazione Progetto ASD A.A. 2020-2021

(M,N,K)-Game

CrazyPlayerPazzissimo

11/02/2022



A cura di

Arto Manuel (#974775), Andrea Napoli (#988997)

Indice

Introduzione

MNK-Game è un gioco a turni in cui due giocatori si alternano nel marcare una cella vuota in una griglia di dimensione ***MxN*.**
Il primo giocatore che riesce ad allineare ***K*** elementi consecutivi in qualunque direzione vince. La partita termina in pareggio solo nel caso in cui tutte le celle della griglia risultino marcate e nessun giocatore ha vinto.

Interfaccia di gioco

Viene da subito fornito un package completo scritto in linguaggio *Java* per l'esecuzione di un *MNK-Game* con M,N,K personalizzabili. L'obiettivo è quello di implementare l'interfaccia *mnkgame.MNKPlayer* realizzando una classe *Player* che attraverso il metodo **selectCell(...)** restituisce la migliore mossa tra quelle possibili.
L'interfaccia di gioco fa uso di un timeout di 10 secondi di attesa, che serve a ricordare che bisogna puntare su soluzioni più efficienti in termine di risorse di calcolo.

Package CrazyPlayer

Prima di iniziare con la descrizione del problema e le scelte implementative effettuate, ecco una lista delle principali classi *java* contenute all'interno del package *CrazyPlayer:*

- **CrazyPlayer**: classe che implementa l'interfaccia *mnkgame.MNKPlayer*
- **util.AIHelper**: classe di helper contenente algoritmi per la risoluzione del game tree, metodi per l'ordinamento delle celle e metodi per la gestione del *timeout*
- **util.Heuristic:** classe contenente la funzione di evaluate che ritorna un valore relativo alla board di gioco e all'ultima cella giocata
- **util.BitBoard:** classe di helper che fornisce metodi statici di supporto per la ricerca di board simmetriche e la conversione di una board in BitBoard
- **model.TranspositionTable:** classe che gestisce *cache* di configurazioni di BitBoard già calcolate in precedenza

Descrizione del problema

Il problema è equivalente a quello della risoluzione del **Game tree:** albero che rappresenta tutte le possibili partite in un gioco a turni.
Un *game tree* **completo** permette di determinare la mossa migliore in ogni situazione, ma il carico di risorse utilizzate per generare tale game tree aumenta in modo esponenziale all'aumentare della griglia di gioco.

Minimax

Come primo approccio abbiamo utilizzato l'algoritmo **minimax:** algoritmo ricorsivo per la risoluzione di un game tree, ovvero permette di individuare la miglior mossa attraverso una funzione di **evaluate()** e una configurazione iniziale.

Minimax, come intuito dal nome, punta a **minimizzare** la massima perdita possibile e, in alternativa, a **massimizzare** il minimo guadagno.

L'algoritmo minimax risulta perfetto ma, come specificato nel paragrafo precedente, all'aumentare della griglia di gioco aumenta il costo computazionale della visita di ogni nodo del game tree.

Grandezza del game tree: assumo *m* mosse per giocatore, visitare l'albero fino a profondità *d* costa $\mathbf{O(m^d)}$

Bisogna perciò puntare a una soluzione efficiente in termini di risorse di calcolo, cercando di trovare la mossa migliore visitando meno nodi possibile del game tree.

Ci sono due modi per ridurre i tempi durante la generazione del game tree:

- Ridurre la **profondita** di ricerca;
- Ridurre l'**ampiezza** (pruning o valutare solo alcune mosse)

Scelte implementative

Durante la realizzazione del progetto ci siamo accorti che per produrre una soluzione in tempi più efficienti, vi erano principalmente due tipi possibili di scelte implementative:

- **Ottimizzazione** dell'algoritmo (pruning, memoization, ecc..),
- Utilizzo di **euristiche** sulla board di gioco e sull'ultima cella selezionata

Ottimizzazione

Scelte che puntano a ottenere una soluzione ottima cercando di visitare solo parzialmente il game tree

Alpha-Beta Pruning

Algoritmo di ricerca per minimizzare il numero di nodi valutati da minimax attraverso il **pruning** di alcune parti del game tree.

Il pruning avviene utilizzando due valori **α** e **β** che rappresentano rispettivamente il punteggio minimo che il giocatore può ottenere e il punteggio massimo che l'avversario può ottenere. Il valore di un nodo **eval** è così definito: $\alpha \leq eval \leq \beta$.
Se ad un certo punto $\boldsymbol{\beta} \leq \boldsymbol{\alpha}$ allora il sottoalbero relativo non può contenere una soluzione ottima (**pruning**).

Costo computazionale AlphaBeta, assumo b = fattore di diramazione e d = profondità

- caso pessimo: $O(b^d)$ = minimax
- caso ottimo (se la prima scelta è sempre la migliore): $\mathbf{O(\sqrt{b^d})}$

Riduzione mosse possibili

Un'altra soluzione per ridurre l'ampiezza durante la visita del game tree è quella ridurre le mosse possibili.

Abbiamo perciò deciso di richiamare l'algoritmo di AlphaBeta solo sulle celle libere nell'intorno $\pm$ 1 di tutte le celle già occupate.

Questo permette di arrivare prima a una possibile vittoria o a un possibile blocco di una vittoria dell'avversario risparmiando utili risorse di calcolo.

Method reference: *AIHelper::getClosedCells*(...)

Ordinamento mosse

Abbiamo visto come l'algoritmo di AlphaBeta pruning risulti ottimo nel caso in cui le mosse siano ordinate, portando a più potature dell'albero di gioco.

Le mosse possibili ottenute dal punto precedente vengono ordinate in base al risultato di una funzione di evaluate che assegna un valore data una configurazione di board e l'ultima cella giocata.

La struttura dati che abbiamo utilizzato per gestire tale ordinamento è **TreeSet,** che usa un AVL (albero binario di ricerca bilanciato) e permette di ordinare un elemento con complessità O(log n).
Inoltre essendo l'albero sempre bilanciato, permette di iterare sugli elementi **preservando** l'ordine di sorting.

Method reference: *AIHelper::getBestMoves*(...)

Transposition Table

Le transposition tables sono utilizzate per salvare stati di gioco già analizzati in un altro ramo del game tree. Ciò ci permette di evitare il calcolo di nodi già visitati precedentemente.

La TT è gestita attraverso una **HashMap** utilizzando come chiave la **BitBoard** relativa alla configurazione di gioco da salvare.

Una *bitboard* è un array di bit utilizzato per rappresentare in modo *memory-efficient* una configurazione di gioco. Sono gestite all'interno del programma tramite la struttura dati **BitSet.**

Nella ricerca AlphaBeta non sempre viene trovato un valore esatto, a causa dei cut. Per questo all'interno della TT vengono salvati un insieme di dati che permettono di capire se tale valore è un **exact value**, un **lower bound** oppure un **upper bound**.

Come si vede dallo pseudocodice che segue, se il *lookup* sulla TT da match il valore salvato viene ritornato solo se quest'ultimo è un *exact value*, altrimenti riduciamo la finestra di ricerca tramite `α e β in caso di lower/upper bound`.

```
Transposition Table Lookup
1    ttEntry := transpositionTableLookup(node)
2    if ttEntry is valid and ttEntry.depth ≥ depth then
3        if ttEntry.flag = EXACT then
4            return ttEntry.value
5        else if ttEntry.flag = LOWERBOUND then
6            α := max(α, ttEntry.value)
7        else if ttEntry.flag = UPPERBOUND then
8            β := min(β, ttEntry.value)
9       if α ≥ β then
10           return ttEntry.value

Transposition Table Store
1    ttEntry.value := value
2    if value ≤ alphaOrig then
3        ttEntry.flag := UPPERBOUND
4    else if value ≥ β then
5        ttEntry.flag := LOWERBOUND
6    else
7        ttEntry.flag := EXACT
8    ttEntry.depth := depth
9    transpositionTableStore(node, ttEntry)
```

Board Simmetriche

Durante la fase di *TT Lookup* possiamo verificare se è già stata salvata una configurazione simmetrica a quella che si sta cercando.

Abbiamo così implementato due diversi metodi *findMIrroredBoard(..)* e *findRotatedBoard(..)* (solo nel caso M=N) che permettono di verificare se una configurazione simmetrica alla board in analisi è già presente all'interno della transposition table.

Iterative Deepening

Un algoritmo noto che abbiamo deciso di implementare nel nostro progetto è *Iterative deepening depth-first search,* un algoritmo di ricerca che permette di richiamare in modo iterativo la funzione alphabeta incrementando ogni volta la profondità di ricerca.

Tale algoritmo risulta molto efficace se si utilizza una configurazione di alphabeta con *memorization.* Durante ogni iterazione salviamo la miglior mossa possibile e tale algoritmo procede fin quando la profondità massima non viene raggiunta oppure fin quando non scade il *TIMEOUT* settato.

Costo computazionale IDDFS, assumo b = fattore di diramazione e d = profondità: **$O(b^d)$**

Euristiche

Scelte che puntano a ottenere una soluzione ottima cercando di visitare solo parzialmente il game tree.

Funzione di evaluate

La funzione di evaluate permette di assegnare un valore data una configurazione di board e l'ultima cella giocata.

I primi controlli verificano lo stato della board:

- *DRAW:* return 0
- *Vittoria*: return +Infinity
- *Sconfitta*: return -Infinity

Nel caso di una configurazione *OPEN* il valore ritornato è dato dalla quantità di elementi consecutivi creati o da quelli bloccati dell'avversario.
Per ognuno di tali allineamenti, chiamati **threat**, è stato deciso un punteggio da assegnare.
Nel caso, controllando i vari allineamenti, ci si accorge che l'ultima cella giocata blocca una possibile vittoria dell'avversario la valutazione ritorna un valore predefinito, solamente inferiore alla valutazione su una cella che porta alla vittoria.

Method reference: Heuristic*::evaluate*(...)

Fonti

Ottimizzazioni

- Alpha-Beta - Chessprogramming wiki
- Transposition Table - Chessprogramming wiki
- Iterative deepening depth-first search - Wikipedia
- https://www.chessprogramming.org/Bitboards

Euristiche

- Generic heuristic for the mnk-game (*tipi di threats*)

Algoritmi

- Alpha-Beta - Chessprogramming wiki
- Iterative deepening depth-first search - Wikipedia

Strutture Dati

- Javarevisited: Difference between PriorityQueue and TreeSet in Java? Example
- A Guide to TreeSet in Java | Baeldung
- Bitboard - Wikipedia
- A Guide to BitSet in Java | Baeldung