ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Informatica
Dipartimento di Informatica - Scienza e Ingegneria
Università di Bologna

Relazione MNK Game
Progetto Algoritmi e Strutture Dati 2021/2022

Heisenberg

Emanuele Pischetola 0001019620,
Alessandro Neri 0001087314,
Saad Farqad Medhat 0001027683

5 febbraio 2023

Sommario

L'MNK game è una generalizzazione del gioco tic-tac-toe o tris. M, N e K sono rispettivamente le dimensioni della board di gioco, righe e colonne, e il numero di simboli da conseguire in fila. Lo scopo del progetto è quello di sviluppare un giocatore software in grado di giocare in tempo limitato nel modo migliore possibile su tutte le istanze possibili di un (M, N, K)-game.

Indice

1 Introduzione

Nell'individuazione di una soluzione al problema abbiamo vagliato diversi approcci. Alcuni sono stati soddisfacenti per i risultati ottenuti come metodo di gioco, ma non per il tempo di decisione; altri viceversa, davano speed-up alla scelta della mossa, ma avevano un metodo di gioco non soddisfacente. Abbiamo anche combinato i punti di forza degli approcci più interessanti per ottenere le prestazioni migliori. Da questa ricerca abbiamo ottenuto i nostri 3 player:

- Heisenberg-sorting(NostroPlayer2.java): che applica l'algoritmo AlphaBeta alle mosse precedentemente ordinate secondo un criterio di importanza.
- Heisenberg-cutoff(NostroPlayer.java): che applica l'algotirmo AlphaBeta solo alle celle nelle vicinanze di ogni cella marcata.
- Heisenberg(NostroPlayer3.java): che applica l'algoritmo AlphaBeta alle mosse ordinate nelle board più piccole. Applica invece l'algoritmo AlphaBeta con il cutoff delle celle lontane da quelle marcate nelle board grandi.

NostroPlayer3.java sarebbe il giocatore che riteniamo migliore e che vorremmo venisse usato nel torneo.

2 AlphaBeta Pruning

Abbiamo fatto uso dell'algoritmo AlphaBeta Pruning, allo scopo di ridurre la quantità di nodi visitati, effettuando una potatura dell'albero di gioco, e permettendoci così di ottenere vantaggi sulla complessità computazionale. Infatti, è possibile ottenere uno speed-up quadratico rispetto a MiniMax, nel caso in cui si riesca a trovare un ordine ottimo di valutazione delle mosse. Detto ciò, nel caso medio, AlphaBeta visita comunque meno nodi dell'algoritmo MiniMax. Il costo computazionale di AlphaBeta è $\mathcal{O}(n)$ dove n è il numero di nodi dell'albero di gioco, inoltre sappiamo che al livello n ci sono $\mathcal{O}(n!)$.

3 IterativeDeepening

Abbiamo deciso di ottimizzare l'algoritmo AlphaBeta facendo uso dell'algoritmo IterativeDeepening, allo scopo di ottenere una gestione migliore del tempo. In questo caso, infatti, non si rende necessario il calcolo di una profondità massima, ma si visita il livello più profondo raggiungibile nel limite di tempo, e all'imminente scadere del timeout, scegliamo la mossa migliore individuata alla profondità precedentemente valutata. Rispetto all'algoritmo AlphaBeta, è leggermente più lento, ma questo svantaggio viene bilanciato dalla possibilità di avere maggiore controllo sulla visita dell'albero di gioco, nel caso in cui venga imposto un limite di tempo entro il quale scegliere la mossa da effettuare. Il costo asintotico rimane comunque lo stesso di AlphaBeta.

Nella funzione `selectCell()` è presente un'implementazione di IterativeDeepening.

Algorithm 1 IterativeDeepening

```
1: bestMove = globalBestMove ← randomMove
2: for depth = 0, 1, ... do
3:     currentDepth ← initial_depth + depth
4:     isMaximizing ← true
5:     int searchResult ← alphabeta(Board, currentDepth, −∞, +∞, isMaximizing)
6:
7:     if timedOut then                          ▷ Se non ho finito di esplorare l'albero termino
8:         break
9:     end if
10:    globalBestMove ← bestMove
11:
12:    if isTreeCompleted or winfound then
13:        break
14:    end if
15:    isTreeCompleted ← true
16: end for
17:
18: Board.markCell(globalBestMove.i, globalBestMove.j)
19: return globalBestMove
```

4 Evaluate

Il metodo di valutazione delle mosse che abbiamo utilizzato consiste nel valutare l'intera board mossa per mossa. Questo metodo si basa sull'idea che ogni nuova mossa fatta può influire sul punteggio dell'intera board solo cambiando le sue vicinanze. Infatti il controllo avviene a distanza K in ogni direzione a partire dall'ultima mossa. Abbiamo un set di sequenze e criteri di valutazione predefiniti, che sono:

- **WIN**: K simboli, ovvero una vittoria
- **THREAT**: K-1 simboli e uno spazio vuoto, quindi la minaccia di vittoria (esempio con K=4 per il player 1: [FREE P1 P1 P1])
- **OPEN END**: K-2 simboli e spazi vuoti, ovvero una sequenza che si avvicina ad un threat (esempio con K=4 per il player 1: [FREE P1 P1 FREE FREE])
- **WEIGHT**: non è una sequenza di simboli, ma piuttosto il valore di ogni cella marcata

Abbiamo utilizzato un array, chiamato `evalWeights`{*WIN*: 1000000, *THREAT*: 100, *OPEN END*: 10, *WEIGHT*: 15}, che contiene il peso di ognuna delle precedenti sequenze.

4.1 Inizializzazione delle sequenze

Per prima cosa durante la chiamata di `initPlayer()` , ci occupiamo di inizializzare una matrice di interi per i weight, di dimensione $M \times N$, assegnando a ciascuna cella un valore intero, che corrisponde al numero di sequenze di lunghezza K che possono passare per quella cella: quelle per cui passano più sequenze (e.g. la cella centrale) hanno un peso più alta. L'operazione appena descritta ha una complessità computazionale equivalente ad $\mathcal{O}(M \times N)$.

Ad esempio nella configurazione 4 4 3:

3	4	4	3
4	7	7	4
4	7	7	4
3	4	4	3

Per ogni altra sequenza inizializziamo invece degli array che la rappresentano.

4.2 Valutazione della nuova mossa

Ad ogni mossa, effettuiamo un ricalcolo della valutazione della board, sommando il peso della cella al totale precedente e andando a cercare delle potenziali sequenze che si sono formate nel suo intorno. La board può anche subire un decremento della valutazione, nel caso in cui la nuova mossa blocchi una sequenza precedentemente creata. Un metodo fondamentale per la valutazione è la funzione `match()`, che si occupa di ritornare *True* se esiste una sequenza alle coordinate date. Quest'ultima ha una complessità computazionale $\mathcal{O}(K)$ nel caso pessimo. Invece per determinare la nuova valutazione dell'intera board viene utilizzato il metodo `calculateIncidence()`, che ha un costo computazionale di $\mathcal{O}(K^2)$.

Quando poi si capita nel caso base della ricorsione, ovvero stato di gioco finale o profondità raggiunta, viene restituita la valutazione della board calcolata fino a quel punto. Per questo viene utilizzato il metodo `eval()`, che ha costo computazionale $\mathcal{O}(1)$.

4.3 Scelta implementativa

L'altra opzione che abbiamo considerato era valutare l'intera board una volta raggiunto uno stato di gioco finale, con un costo computazionale di circa $\mathcal{O}(M * N * K)$. Abbiamo quindi deciso di non percorrere questa strada, perché il numero di stati finali è asintoticamente lo stesso degli stati presenti nei livelli precedenti dell'albero. Di conseguenza crediamo sia più efficiente fare una valutazione meno costosa negli stati non finali.

5 Ordinamento delle mosse

Le mosse sono ordinate in base al loro peso. Inizialmente a tutte le celle viene assegnato un peso equivalente a zero. Successivamente, ogni volta che viene effettuata una mossa, le celle intorno a quest'ultima, vengono aumentate di peso. Il peso viene attribuito in base alla distanza dalla cella in cui è stata effettuata l'ultima mossa: alle celle nell'immediata vicinanza viene assegnato il peso più alto K, mentre il peso delle celle più lontane viene incrementato di $(K - d)$, in cui d indica la distanza dalla cella e $(1 \leq d \leq K - 1)$.

Come per altre soluzioni abbiamo vagliato più ipotesi prima di prendere una decisione su come operare. Subito abbiamo pensato di ordinare le mosse ogni volta che fosse necessario. Abbiamo poi concluso che sarebbe stato più efficiente mantenersi una struttura dati la quale ci permettesse di non dover ripetere più volte l'ordinamento. Ci si è presentato quindi il bivio della scelta di una struttura dati. Abbiamo ipotizzato di mantenere un array di mosse ordinate ed eseguire un riordinamento mirato delle singole celle che cambiavano di peso, con operazioni simili al bubble sort. Inoltre abbiamo pensato a memorizzare un heap e nello specifico abbiamo vagliato le ipotesi dell'heap binario e di quello di Fibonacci. Infine abbiamo optato per una struttura dati ad hoc.

5.1 Liste di trabocco

Il peso dato alle celle, che individua la loro importanza, è un numero intero. Abbiamo deciso di sfruttare questa caratteristica per creare un array che contiene in ogni posizione una lista delle celle di peso uguale alla posizione stessa. In questo modo l'ordine delle celle viene mantenuto in memoria e i costi computazionali sono limitati:

inserimento: inserimento in testa a una lista. Costo $\mathcal{O}(1)$.

eliminazione: i nodi delle liste sono contemporaneamente inseriti in una matrice che rappresenta la board; non è quindi necessaria la ricerca del nodo prima dell'eliminazione. Costo $\mathcal{O}(1)$.

modifica al peso: eliminazione da una lista e inserimento in un'altra. Costo $\mathcal{O}(1)$.

to array: scansione dell'intero array e per ogni cella che contiene una lista visita dell'intera lista. Considerando il numero di elementi complessivo N si può dire che il costo è $\mathcal{O}(N)$.

Le liste utilizzate per realizzare questa struttura dati sono doppiamente concatenate per maggiore comodità nelle operazioni. Inoltre è stata aggiunta una primitiva che permette di iterare comodamente la lista.

6 Cut off

Le celle vuote sono possibili mosse che l'algoritmo può prendere in considerazione, ma un algoritmo molto computazionalmente costoso come l'AlphaBeta può non riuscire a terminare se richiamato su tutte le celle della board, soprattuto se questa non è di piccole dimensioni. Per soddisfare i vincoli progettuali e ottenere un algoritmo rapido, sotto i 10 secondi, e il più possibile efficiente si può eliminare la valutazione di celle poco interessanti. Inizialmente abbiamo pensato di considerare come interessanti solo le celle nelle vicinanze di quelle marcate, senza ulteriori distinzioni.

Abbiamo poi optato per una selezione di celle data dall'allineamento con quelle marcate. Per ogni cella vuota si controlla se nelle 8 direzioni attorno; nord, nord-est, est, sud-est, sud, sud-ovest, ovest, nord-ovest; è presente una cella marcata o meno. Se è presente, la cella vuota dalla quale è partito il controllo viene considerata interessante, altrimenti no. Solo le celle interessanti verranno poi considerate come ipotetiche mosse nell'AlphaBeta.

Abbiamo proceduto, quindi, a selezionare le celle vicine, e di queste solo quelle allineate con altre già marcate. Queste selezioni portano a una drastica riduzione delle celle da coinvolgere nell'algoritmo AlphaBeta:

distanza 1: si controllano tutte le celle a distanza 1 con entrambi i metodi.

distanza 2: si risparmiano 8 delle 16 celle a distanza 2, la metà dei controlli, con il metodo dell'allineamento.

distanza 3: si risparmiano 16 delle 24 celle a distanza 3, $\frac{2}{3}$ dei controlli, con il metodo dell'allineamento.

Nonostante i grandi vantaggi a distanza 3 che abbiamo ottenuto con il miglioramento dato dallo studio dell'allineamento, abbiamo scelto di terminare il taglio con la distanza 2 per non rischiare di renderli vani, soprattutto nelle board intermedie. Nelle board molto grandi invece è possibile che i miglioramenti non si notino proprio per le vaste dimensioni. Si ha infatti che le mosse possono essere maggiormente sparse e il numero delle celle da controllare, nonostante sia controllato, può impennare facilmente. Per prevenire questa eventualità abbiamo adottato nelle board più grandi il criterio di soffermarci alla distanza 1.

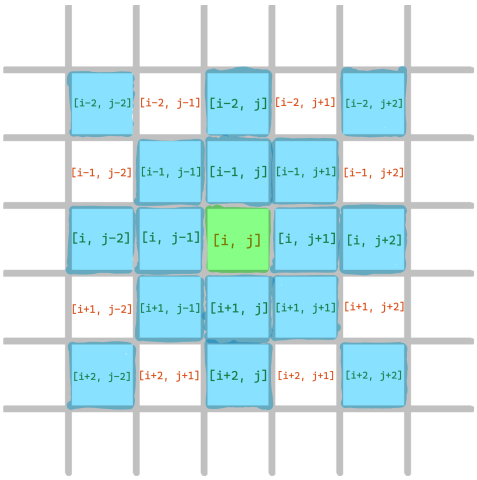

Figura 1: Esempio di cutoff a distanza 2

7 Altre idee vagliate

Come detto abbiamo valutato molte ipotesi, alcune di esse sono però state scartate.

7.1 Funzione euristica

Quando ci siamo accorti che nelle board molto grandi era difficile completare l'albero di chiamate dell'algoritmo AlphaBeta, e arrivare a una profondità accettabile, abbiamo anche tentato un approccio euristico. La funzione prevedeva diversi casi:

- iniziare la partita con la mossa nel centro.
- completare una sequenza vincente.
- bloccare una sequenza perdente.
- bloccare una sequenza destinata a far perdere.
- continuare una sequenza di propri simboli.
- iniziare una sequenza di simboli propri se non ce ne sono altre.

Queste casistiche, però, non permettevano all'algoritmo di essere sufficentemente incisivo, e per questo motivo abbiamo deciso di accantonare questa ipotesi.

8 Miglioramenti

In conclusione vorremmo proporre delle possibili migliorie che si potrebbero apportare.

8.1 Board grandi

Abbiamo notato che, con le dovute differenze, nessuno degli algoritmi studiati e pensati è particolarmente performante nelle board di grandi dimensioni. Il maggior problema riscontrato è rimanere sotto i 10 secondi con un algoritmo come l'AlphaBeta, che a ogni iterazione aumenta fattorialmente i controlli da eseguire. Probabilmente, per risolvere questa difficoltà, sarebbe più semplice adottare un approccio totalmente differente basandosi su un algoritmo diverso.

8.2 Transposition table

Un'idea che abbiamo vagliato, era quella delle tabelle di trasposizione: un sistema per evitare di controllare più volte una stessa configurazione. Per mettere in pratica questo metodo è necessario assegnare un valore univoco a ogni singola board per poterla facilmente individuare una volta che essa si ripresenta, ma in un punto diverso dell'albero di gioco. Sarebbe stato possibile inoltre riconoscere le stesse configurazioni delle board specchiate, ruotate e simmetriche.

Per studiare questo metodo abbiamo preso spunto da algoritmi in grado di giocare a scacchi e il gioco cinese del Go : sembra infatti che sia vastamente utilizzato in quegli ambiti. Dalle nostre ricerche abbiamo capito che per generare il valore univoco da assegnare a una board si utilizzano le Zobrist keys, che fanno parte di un omonimo metodo di hashing.