MNKGame – Progetto di algoritmi e strutture dati a.a. 2022-2023

PlayerMandus

Rocco Pastore 0000923786
Pasquale Ricciulli 0000923980

Il problema affrontato consiste nel creare un giocatore di tic tac toe che sia in grado di fare la scelta ottimale in base alle condizioni della configurazione di gioco attuale. Per fare ciò, abbiamo utilizzato l'algoritmo MiniMax con l'ottimizzazione Alpha Beta Pruning.
I punti cardine del codice riguardano la funzione selectCell, evaluate e calculateCombination.

selectCell()

Come dice il nome stesso, questa funzione seleziona la prossima cella da marcare. In ordine controlla prima se il nostro giocatore può vincere in una sola mossa, e nel caso la effettua. Lo stesso controllo viene fatto anche sull'avversario, bloccando la sua mossa vincente. Se non si verificano le situazioni appena citate, entra in gioco l'algoritmo Alpha Beta Pruning, che analizza tutte le possibili mosse simulando un albero con profondità data da getDepth(), che lavora in base al numero di celle della board. La profondità decrescerà con l'aumentare dei nodi, per evitare di simulare alberi troppo grandi. L'analisi, grazie alla funzione evaluate, associa un valore ad ogni combinazione, in modo da determinarne la migliore. Il costo di selectCell dipende dalla ricerca della possibile vittoria dell'avversario, $O(x^2)$dove x è il numero di celle, e dal costo di Alpha Beta, ovvero $O(m^d)$ nel caso pessimo, e $O(\sqrt{m^d})$ nel caso ottimo.

evaluate()

La funzione evaluate determina un punteggio da assegnare ad ogni nodo dell'albero, e quindi ad ogni combinazione.Viene assegnato un punteggio di 1000 in caso di vittoria, - 1000 in caso di sconfitta e 0 per il pareggio, mentre, se siamo in una configurazione ancora aperta e non terminata, calcoliamo il punteggio in base a tutte le righe, colonne e diagonali.

Per ognuna di esse richiamiamo la funzione calculateCombination, che facendo uso di programmazione dinamica calcola le occorrenze consecutive dei nostri simboli e delle celle vuote.

```
for(int i=1; i < combination.length;i++)
    {
        if(combination[i] == MNKCellState.P1 ||
combination[i] == MNKCellState.FREE)
        {
            if(combination[i] == MNKCellState.P1)
            {
                occurrenciesP1[i] =
occurrenciesP1[i-1] + 1.5;
            }
            else{
                occurrenciesP1[i] =
occurrenciesP1[i-1] + 1;
            }
            occurrenciesP2[i] = 0;
        }
    }
```

Parallelamente, viene fatta la stessa cosa per le occorrenze dei simboli avversari, in modo da dosare il punteggio scegliendo la combinazione che porta più velocemente alla vittoria e che nello stesso tempo favorisca meno l'avversario. In particolare, consideriamo maggiormente le celle marcate dai propri simboli, in modo da favorire quelle linee che sono quasi pronte per contenere un allineamento.

`score += (maxP1 - maxP2) * 10;` dove maxP1 e maxP2, sono rispettivamente il maggior numero di celle libere e marcate consecutive di entrambi i giocatori.

Quindi Mandus procede con una strategia prettamente offensiva, cercando di posizionare più velocemente possibile simboli adiacenti, o che possono potenzialmente creare un allineamento con una cella già marcata, che non favoriscano l'avversario. Questa scelta si basa sul fatto che, soprattutto in tabelle molto grandi e con un valore di K basso, se si marcano celle adiacenti si ha più possibilità di creare un allineamento vincente in meno tempo.
Il costo della funzione evaluate() è di $O(M * N * K)$, dovuta all'analisi delle diagonali.