Kebabo

IA per il gioco Connect X

Corso di Algoritmi e Strutture di Dati

Lorenzo Peronese (0001081726), Omar Ayache (0001068895)

Alma Mater Studiorum
Bologna
3 settembre 2023

Indice

1 Introduzione **1**
1.1 Configurazioni di gioco 2

2 La nostra implementazione **3**
2.1 MiniMax-AlphaBeta 3
2.2 Move explorator order 5
2.3 Tabella delle trasposizioni 5
2.4 Iterative Deepening 6
2.5 Euristica 7

3 Conclusioni **7**

1 Introduzione

Connect X è una variante del celebre gioco Connect 4 (Forza 4), in cui le dimensioni della matrice di gioco e il numero di pedine da collegare variano; Connect 4 è infatti una configurazione di Connect X, la 6 7 4.

A turno due giocatori fanno cadere delle pedine lungo le colonne della matrice di gioco, con l'obiettivo di allinearne in orizzontale, verticale o diagonale un certo numero; quello che lo differenzia dal gioco Tic-Tac-Toe (Tris) è la gravità: una pedina messa in una certa colonna della matrice va ad occupare la posizione libera più in basso.

Connect 4 è un gioco risolto, ovvero esiste una sequenza di mosse che porta sempre e inevitabilmente alla vittoria di uno dei giocatori oppure a una patta; ovviamente, Connect X non può esserlo a causa della grande varietà di configurazioni diverse; alcune di esse sono però risolte, come ad esempio le board 4 4 4 e 4 6 4 che, giocate perfettamente da entrambi i giocatori, porteranno sempre rispettivamente a un pareggio e a una vittoria del secondo giocatore.

Il progetto richiedeva lo sviluppo di un algoritmo in grado di giocare alcune configurazioni e selezionare sempre la mossa migliore entro un certo tempo limite; l'algoritmo avrebbe poi giocato due partite, una come primo giocatore e una come secondo, per ogni board, contro ogni altro player; una vittoria (regolare o per errore avversario) porta 3 punti, un pareggio 1 punto e una sconfitta 0 punti. Le configurazioni sono riportate nella pagina seguente.

I due errori più comuni che terminano immediatamente la partita sono timeout, ovvero la scadenza del tempo per fare una mossa, e mossa non legale.

É stata poi fornita dai docenti l'interfaccia di gioco CXGame, che si occupa di inizializzare la partita, sia in modalità testuale, sia grafica; l'implementazione deve essere fornita come package, e la classe che implementa l'interfaccia CXPlayer deve contenere 3 metodi: initPlayer, selectColumn e playerName.

L'algoritmo adottato ha diverse ottimizzazioni che sono state aggiunte progressivamente; ogni qual volta veniva implementata una nuova versione, questa veniva testata contro la precedente per verificarne la bontà; alcune implementazioni sono state mantenute perchè migliori delle precedenti, mentre altre sono state scartate perchè non apportavano un sigificativo miglioramento.

Per questo progetto è inoltre stato fatto uso della potenza di calcolo delle macchine di laboratorio, per il testing delle varie versioni del player.

1.1 Configurazioni di gioco

M	N	X
4	4	4
5	4	4
6	4	4
7	4	4
4	5	4
5	5	4
6	5	4
7	5	4
4	6	4
5	6	4
6	6	4
7	6	4
4	7	4
5	7	4
6	7	4
7	7	4
5	4	5
6	4	5
7	4	5
4	5	5
5	5	5
6	5	5
7	5	5
4	6	5
5	6	5
6	6	5
7	6	5
4	7	5
5	7	5
6	7	5
7	7	5
20	20	10
30	30	10
40	40	10
50	50	10

2 La nostra implementazione

La versione definitiva del giocatore utilizza l'algoritmo MiniMax con potatura Alpha-Beta per esplorare l'albero delle mosse possibili. Per aumentare i tagli di AlphaBeta, è stato aggiunto un semplice move explorator order, che garantisce spesso di trovare la mossa migliore tra le prime analizzate e fa uso inoltre di una abella delle trasposizioni per memorizzare il punteggio delle posizioni che vengono incontrate.

La ricerca Iterative Deepening permette poi di effettuare una visita a livelli dell'albero, aumentando gradualmente la profondità e garantendo così la massima efficienza possibile.

Nelle board più grandi, questo non è sufficiente per esplorare l'intero albero di gioco, perciò si rende necessaria una funzione di valutazione dei nodi non terminali, che si occupa di analizzare la posizione e assegnare un punteggio in base alle proprie minacce e quelle dell'avversario.

Per le implementazioni non banali viene fornito lo pseudocodice.

2.1 MiniMax-AlphaBeta

Essendo Connect X un gioco a somma zero, ovvero la somma dei punteggio dei due giocatori è sempre uguale a zero, il punto di partenza è stato l'algoritmo MiniMax studiato a lezione. Questo algoritmo permette di minimizzare la massima perdita possibile e massimizzare il minimo guadagno: l'algoritmo cerca la mossa migliore dal fondo dell'albero e risale fino alla posizione attuale, supponendo che entrambi i giocatori facciano sempre le scelte migliori.

Se l'algoritmo riesce ad esplorare l'intero Game Tree, sappiamo per certo che sceglierà la mossa migliore, ma purtroppo, soprattutto con configurazioni grandi, è molto difficile che questo accada; infatti, il numero di nodi visitati in una partita con m mosse e n turni è limitato da $O(m^n)$.

La potatura AlphaBeta permette di tagliare alcuni rami dell'albero di gioco che non sono interessanti: ad esempio se il giocatore A massimizza e la prima mossa gli da un certo punteggio, se il primo nodo figlio della seconda mossa ha un punteggio minore della mossa 1, sappiamo per certo che la mossa 2 avrà uno score minore di quello del primo figlio (perchè il giocatore B minimizza), perciò non ha senso finire l'esplorazione di quel ramo perchè la mossa ad esso collegato non verrà mai scelta, e possiamo quindi potarlo.

Questa miglioria incide sulle prestazioni sulla base di un fattore: l'ordinamento delle mosse. Visitando prima le mosse che si riveleranno poi migliori, il numero dei tagli sarà molto maggiore, mentre con l'ordinamento opposto non ci sarà nessun taglio e il costo computazionale sarà uguale a quello di MiniMax. Infatti, ipotizzando un ordinamento delle mosse ottimale, il numero di nodi visitati in una partita con m mosse e n turni ha come upper bound $O(\sqrt{m^n})$.

Algorithm 1 MiniMax-AlphaBeta

```
procedure alphaBetaSearch(board, depth)
    bestMove ← −1
    for move in possibleMoves() do
        score ← AlphaBetaMin(board, −∞, ∞, depth)
        if score > alpha then
            alpha ← score
            bestMove ← move
        end if
    end for
    return bestMove
end procedure

procedure AlphaBetaMax(board,alpha,beta,depth)
    if depth = 0 or node is a leaf then
        return Evaluate(board)
    end if
    for move in possibleMoves() do
        score ← AlphaBetaMin(board, alpha, beta, depth − 1)
        if score ≥ beta then                          ▷ potatura AlphaBeta
            return beta
        end if
        if score > alpha then
            alpha ← score
        end if
    end for
    return alpha
end procedure

procedure AlphaBetaMin(board,alpha,beta,depth)
    if depth = 0 or node is a leaf then
        return Evaluate(board)
    end if
    for move in possibleMoves() do
        score ← AlphaBetaMax(board, alpha, beta, depth − 1)
        if score ≤ alpha then                         ▷ potatura AlphaBeta
            return alpha
        end if
        if score < beta then
            beta ← score
        end if
    end for
    return beta
end procedure
```

2.2 Move explorator order

Per usufruire al massimo dei vantaggi del pruning di AlphaBeta, abbiamo utilizzato un array che contiene l'ordine con cui esplorare le mosse, con l'obiettivo di iniziare dalle mosse più promettenti. L'idea che sta alla base è che le colonne centrali sono quelle più strategiche e da cui passano più collegamenti, perciò è spesso vero che le mosse migliori sono quelle vicino al centro.

L'array viene riempito in questa maniera:

$$moveOrder[i] = columns/2 + (1 - 2 \cdot (i\%2)) \cdot (i+1)/2, \forall i \in \{0, columns - 1\}$$

Il risultato sarà questo:

- $moveOrder[0] \leftarrow columns/2$
- $moveOrder[1] \leftarrow columns/2 - 1$
- $moveOrder[2] \leftarrow columns/2 + 1$
- ...
- $moveOrder[columns - 2] \leftarrow 0$
- $moveOrder[columns - 1] \leftarrow columns - 1$

2.3 Tabella delle trasposizioni

Nell'analizzare tutti i possibili stati di una partita, MiniMax può trovare più nodi con la stessa posizione delle pedine, ma raggiunta con un ordine di mosse differente (*trasposizione* in gergo scacchistico). La tabella delle trasposizioni viene adottata proprio per evitare di sprecare tempo a ricalcolare il punteggio della trasposizione di una mossa già calcolata.

La struttura dati HashMap di Java memorizza score e profondità di una posizione; la funzione di hash è la Zobrist Hashing, che associa ad ogni cella della matrice 3 interi casuali, uno per ogni possibile valore che può assumere (`P1, P2, FREE`). L'hash della posizione viene calcolato facendo la `XOR` di tutti i valori Zobrist associati alla matrice di gioco, in base alla posizione raggiunta.

Quando viene chiamato MiniMax, prima di iniziare ad analizzare le mosse possibili viene controllato in tempo lineare se la posizione da calcolare si trova già nella HashMap e se la profondità con cui è stata calcolata è maggiore o uguale a quella richiesta; se è presente, viene ritornato direttamente quel valore, altrimenti MiniMax calcola il punteggio, che viene poi salvato nella tabella.

2.4 Iterative Deepening

L'algoritmo MiniMax visita l'albero in profondità, in post-ordine; se non è in grado di visitare l'intero Game Tree entro il tempo limite, allora non è riuscito ad analizzare le ultime mosse possibili, rischiando così di sbagliare l'analisi e restituire una mossa non ottimale.

É di vitale importanza che il giocatore abbia analizzato ogni mossa e abbia una visione totale della board, anche se questo significa perdere un po' in profondità; questo problema viene risolto dall'algoritmo IterativeDeepening, che implementa una visita a livelli dell'albero di gioco.

Grazie a questa ottimizzazione, viene calcolata la mossa migliore a profondità crescenti e viene restituito l'ultimo valore calcolato prima dello scadere del tempo.

In realtà, questo processo avviene solamente durante la prima mossa della partita, mentre nelle successive la profondità non viene più inizializzata a 0, ma alla profondità raggiunta durante la mossa precedente (-2 per essere sicuri di avere una mossa valida da ritornare).

Inoltre, se al crescere della profondità il giocatore si accorge che non ha modo di evitare la sconfitta, gioca l'ultima mossa migliore valida per cercare di prolungare il più possibile la partita, sperando in un errore avversario.

Algorithm 2 Iterative Deepening

```
procedure IterativeDeepening(board)
    bestmove ← −1
    depth ← 0
    while thereIsTime() do
        depth ← depth + 2
        bestMove ← alphaBetaSearch(board, depth)
    end while
    return bestMove
end procedure
```

2.5 Euristica

Alla scadenza del tempo, l'algoritmo lancia un'eccezione che blocca tutte le chiamate ricorsive e deve valutare i nodi raggiunti: se questi sono foglie, la valutazione è triviale: INTEGER.MAXVALUE se quella serie di mosse gli permette di vincere, INTEGER.MINVALUE se lo portano a una sconfitta e 0 se la partita si conclude con un pareggio.

Se però la partita non si conclude con l'ultima mossa, è necessario poter confrontare diverse posizioni e stabilire la migliore, assegnando ad ognuna di esse un punteggio.

La strategia che abbiamo adottato è stata quella di contare le serie aperte di pedine, ovvero quelle che in futuro potrebbero portare alla vittoria, e dare un punteggio pari al numero di pedine collegate elevato al quadrato. Lo score della posizione è dato dunque dalla differenza tra il punteggio delle serie aperte del giocatore e quelle dell'avversario.

Oltre a questo, un'altra idea è stata quella di dare un numero di punti ad ogni cella della matrice in base alla posizione strategica, quindi più punti alle celle centrali e meno a quelle più esterne.

Una volta implementata la matrice di punteggi, il cambiamento non si è notato particolarmente perchè di per sè già la funzione di valutazione include questo tema: infatti le celle centrali fanno parte di diverse serie che possono portare alla vittoria, al contrario di quelle più esterne, quindi controllarle porta più punti; pertanto abbiamo deciso di mantenere la funzione originale ed eliminare questa parte.

Algorithm 3 EURISTICA

```
procedure EVALUATE(board)
    if MYWIN() then
        return ∞
    else if YOURWIN() then
        return −∞
    else if ISDRAW() then
        return 0
    else
        return ROWSCORE(board) + COLUMNSCORE(board) + DIAGONALSCORE(board) + ANTIDIAGONALSCORE(board)
    end if
end procedure
```

METTO LO PSEUDOCODICE DELL'EURISTICA? FORSE È UN PO' LUNGHETTO

3 Conclusioni