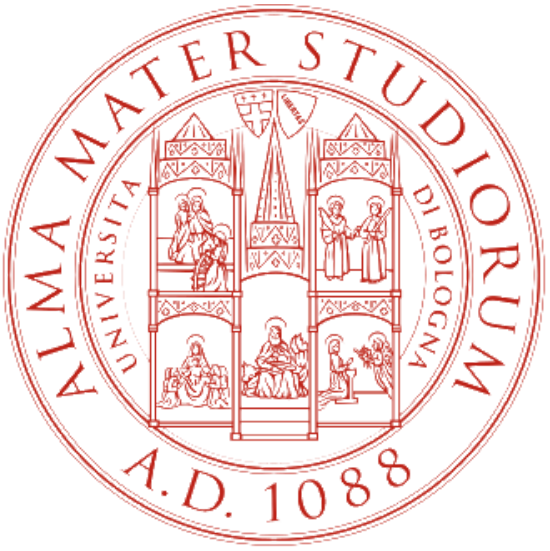
Relazione progetto di Algoritmi e Strutture Dati
Giocatore di ConnectX “PojamDesi”

Ivan De Simone - 0001069314
ivan.desimone@studio.unibo.it
Payam Salarieh - 0001077673
payam.salarieh@studio.unibo.it



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Progetto di ASD A.A. 2022-23
Corso di Laurea in Informatica
Dipartimento di Informatica - Scienza e Ingegneria
Università di Bologna

1 - Introduzione

1.1 Il progetto

Lo scopo del progetto è quello di implementare un giocatore software in grado di giocare nel miglior modo possibile su tutte le istanze (ragionevolmente) possibili di un ConnectX, rispettando un limite di tempo.

ConnectX è un gioco a turni, generalizzazione del popolare Connect4 (per noi Forza4). Si sviluppa su un numero arbitrario di righe e colonne, in cui bisogna allineare X pedine verticalmente, orizzontalmente oppure in diagonale per vincere.

1.2 Problema

Per effettuare la mossa ottimale ad ogni turno, sarebbe necessario valutare tutte le possibili mosse successive, che crescono esponenzialmente in numero ad ogni giocata. Per questo motivo il principale problema da affrontare è il limite di tempo per compiere la propria mossa, che non permette di analizzare l'intero albero di gioco nella fase iniziale ed intermedia della partita.

La soluzione adottata è una visita in ampiezza piuttosto che in profondità dell'albero di gioco, che consente di trovare la mossa più vantaggiosa esplorando completamente il maggior numero di livelli prima che scada il tempo.

2 - Scelte progettuali

2.1 selectColumn

La colonna portante del giocatore è il metodo selectColumn(), che data una configurazione di gioco restituisce la colonna valutata ottimale in cui giocare la pedina. Il metodo esegue una serie di controlli per decretare la mossa migliore.

Per prima cosa verifica se è la prima mossa della partita per il giocatore, e nel caso restituisce la colonna di mezzo, la più vantaggiosa ad inizio gioco.

Se non è la prima mossa, controlla tramite immediateMove() se ci sono mosse a disposizione che portano ad una vittoria diretta, e nel caso gioca una di queste. Contrariamente, nell'ArrayList A vengono inserite le possibili mosse che non conducono ad una sconfitta diretta. Se A è vuoto allora la sconfitta è inevitabile, se contiene un solo elemento allora è l'unica mossa giocabile. Se A contiene più di una mossa fattibile si chiama il metodo iterativeDeepening() per analizzare le mosse successive e scegliere quindi la più vantaggiosa.

Se il tempo dovesse scadere prima della decisione restituisce una colonna a caso.

2.2 Algoritmi noti

L'implementazione del giocatore sfrutta due algoritmi già rodati per questo genere di problema.

Il primo è il MiniMax con limite di profondità, che consente di assegnare ricorsivamente una valutazione ad ogni nodo, rappresentante la bontà di una mossa, basandosi sui nodi figli fino ad una profondità stabilita come parametro. In particolare MiniMax viene usato con ottimizzazione alphabeta-pruning, che interrompe la visita se non si può ottenere un valore migliore di quello trovato. Questo algoritmo fa uso di una visita in profondità, che non permette di valutare tutte le possibili mosse a disposizione del giocatore nei limiti di tempo.

Il secondo algoritmo noto utilizzato è l'Iterative Deepening, che sfrutta il MiniMax incrementando il limite di profondità raggiungibile ad ogni iterazione. Questo algoritmo non consente ugualmente di visitare l'intero albero, ma implementa una visita equa di tutte le mosse a disposizione del giocatore.

2.3 Valutazione delle configurazioni

Alle configurazioni di gioco (nodi) viene assegnato un valore dal metodo evaluate().

Se la configurazione di gioco è terminale viene assegnato un valore ≥ 1 in caso di vittoria, un valore ≤ -1 in caso di sconfitta oppure 0 (zero) in caso di patta.
A parità di stato (vittoria o sconfitta) viene privilegiata con un punteggio maggiore la mossa che fa vincere più velocemente o perdere più lentamente, sfruttando la profondità della configurazione nell'albero di gioco.

Se la configurazione non è terminale il valore viene calcolato da una funzione euristica che assegna un valore float tra 0 e 1 (1 escluso) in base a quante pedine del giocatore sono adiacenti a quella giocata per ultima.

Non c'è ambiguità tra una mossa che porta al pareggio (eval = 0) e una con totale indecisione (nessuna pedina adiacente, eval = 0) poichè il pareggio si scopre solo arrivando in fondo all'albero di gioco, quindi si hanno abbastanza informazioni per non trovare una mossa con totale indecisione.

2.4 Contributi originali

Le mosse giocabili inserite nell'ArrayList A sono in realtà delle coppie < colonna, qualità della mossa >. Ad ogni turno del giocatore, prima dell'utilizzo di iterativeDeepening() viene effettuato un ordinamento delle mosse dalla più promettente alla meno promettente basato sulla valutazione data dalla funzione euristica. Questo viene eseguito con lo scopo di sfruttare al meglio l'ottimizzazione alphabeta-pruning.

Per ri-valutare il minor numero di nodi possibili in iterativeDeepening(), alla prima iterazione si usa come profondità limite il massimo valore fra 3 (le mosse a profondità 1 e 2 le analizza immediateMove()) e la profondità massima valutata completamente al turno precedente meno 2 (che sono le mosse giocate per arrivare alla configurazione attuale). Intuizione alla base: se il giocatore valuta il livello d completamente, al turno successivo (con due livelli in meno da visitare), di sicuro riesce a valutare interamente quello che prima era il livello d, quindi risparmia il tempo di ri-valutazione di tutti i nodi sopra.

3 - Costi computazionali

Siano M e N rispettivamente il numero di righe e colonne della matrice di gioco.

3.1 Funzioni di costo costante

Le seguenti funzioni eseguono un numero finito di passi senza ricorsione o cicli e pertanto hanno un costo costante O(1):

- initPlayer()
- playerName()
- checkTime()
- isLeaf()
- evaluate()
- coinsAround()

3.2 Costo di immediateMove

Il metodo immediateMove() nel caso pessimo non fa altro che controllare tutte le mosse del giocatore (O(N)), e per ognuna analizza le contromosse dell'avversario (O(N)). Ne risulta che il costo in termini di tempo è $O(N^2)$.

3.3 Costo di alphabeta

Nel caso pessimo l'ottimizzazione alphabeta-pruning non evita nessun ramo, perciò il numero di nodi visitati dall'algoritmo è lo stesso del semplice MiniMax. Il costo nel caso pessimo dell'algoritmo MiniMax, come visto a lezione, corrisponde al numero totale di nodi visitati. Sia d la profondità limite a cui arriva alphabeta(). Possiamo trovare un upper bound per il numero di nodi visitati, considerando anche le configurazioni di gioco illegali, come il numero di mosse possibili elevato al livello massimo raggiunto, quindi $O(N^d)$.

3.4 Costo di iterativeDeepening

Il costo di iterativeDeepening() è dato dall'utilizzo di alphabeta() con profondità limite incrementata ad ogni iterazione fino a d, profondità massima raggiungibile. Detto questo il costo in termini di tempo è dato da $O(N)+O(N^2)+ ... +O(N^d) = O(N^d)$

3.5 Costo di selectColumn

Il costo del metodo selectColumn() è dominato dal costo di immediateMove(), $O(N^2)$, e di iterativeDeepening(), $O(N^d)$, dove d è la profondità massima raggiunta. Essendo di interesse il caso pessimo si assume $d > 1$. Come conseguenza di questa assunzione si ottiene che il costo nel caso pessimo è $O(N^d)$.

4 - Miglioramenti

4.1 Funzione euristica

La funzione euristica utilizzata non è affidabile in quanto non sempre un buon punteggio è indicatore di una situazione di gioco favorevole. Detto questo, una funzione non totalmente affidabile è migliore della totale indecisione, in assenza di ulteriori informazioni. Un miglioramento si potrebbe avere verificando la presenza di allineamenti nella matrice invece della semplice adiacenza all'ultima mossa, facendo attenzione a non gravare troppo sul costo computazionale.

4.2 Configurazioni già valutate

Durante la partita numerose configurazioni di gioco si ripresentano tramite combinazioni di mosse differenti. Si potrebbe avere un miglioramento tenendo traccia delle configurazioni già valutate con associato il valore calcolato.

Si è provato a sviluppare questa idea tramite una tabella hash. A seguito di analisi sperimentale questo metodo si è rivelato inefficiente in quanto i calcoli per mantenere l'HashMap rallentavano il giocatore.

Un altro problema generato da questa soluzione è che configurazioni di gioco differenti potrebbero avere lo stesso valore hash e quindi essere associate alla stessa valutazione, portando così ad inconsistenza dei dati.