PROGETTO ASD 2023

1) INTESTAZIONE

Nome progetto: LavaHound
Membri del gruppo:
- **Lorenzo Casalini 0001069403**
- **Gianluca Casaburi 0001078525**

2) DESCRIZIONE DEL PROBLEMA

Il progetto si focalizza sulla realizzazione di un giocatore artificiale per il gioco Connect X. La sfida principale consiste nello sviluppare un algoritmo in grado di selezionare la mossa migliore possibile ad ogni turno del gioco, tenendo conto delle mosse precedenti e cercando di massimizzare le proprie opportunità di vincita o minimizzare quelle dell'avversario.
Tutto questo deve essere fatto entro il tempo limite messo a disposizione per ogni turno.

4) SCELTE PROGETTUALI

- Ricerca nel Game Tree: Abbiamo implementato un algoritmo di ricerca del game tree basato sull'algoritmo Alpha-Beta Pruning visto a lezione, che ci consente di esplorare le possibili mosse e di valutarle per determinare la migliore da eseguire.

- Iterative Deepening: Per gestire il tempo di calcolo limitato abbiamo utilizzato un altro approccio visto a lezione ovvero l'Iterative Deepening, che ci consente di aumentare progressivamente la profondità della ricerca fino a quando non raggiungiamo il limite di tempo prestabilito.

- Transposition Table: Per migliorare l'efficienza della ricerca e dei calcoli, abbiamo implementato una tabella di trasposizione che memorizza le valutazioni dei game state già esplorati, riducendo così la necessità di ricalcolarle.

- Valutazione del game state: Abbiamo sviluppato una funzione di valutazione che assegna un punteggio a ciascun game state in base a diverse caratteristiche,
come la presenza di minacce di vittoria, opportunità di creare trappole per l'avversario e la presenza di sequenze di pedine già presenti. Questa funzione guida l'algoritmo
nella scelta della mossa.

- Gestione del Tempo: Abbiamo implementato un sistema di gestione del tempo per evitare di superare il limite concesso per effettuare la mossa. L'algoritmo si interrompe quando ci si avvicina al limite di tempo, restituendo così la mossa migliore finora trovata.

5) PROBLEMI INCONTRATI

- Ideare una funzione per valutare i vari game state in modo accurato ma allo stesso tempo efficiente
La nostra idea iniziale era quella di combinare varie strategie di gioco come creazione di trappole per l'avversario, contare la parità o disparità del numero di celle e contare le connessioni già formate. Questa combinazione di strategie avrebbe in teoria consentito una scelta delle mosse molto precisa, ma i calcoli da fare erano troppi e quindi l'algoritmo non riusciva a dare la risposta in tempo.

Allora abbiamo provato a fare l'opposto, ovvero limitarci ad una funzione per valutare i game state molto banale ovvero contare il numero di sequenze formate nelle righe, colonne e diagonali. Questo approccio però, nonostante fosse molto efficiente, era troppo impreciso e spesso attribuiva punteggi alti a sequenze che non potevano portare a vittoria o attribuiva punteggi bassi a sequenze di pedine che non erano contigue.

Quindi abbiamo optato per una via di mezzo, ovvero una funzione in grado di valutare in tempi ragionevoli le sequenze presenti nella griglia di gioco. Verrà descritta meglio dopo.

- Ottimizzare il costo computazionale e i tempi di calcolo per evitare errori di timeout
Con tavole di gioco molto grandi abbiamo notato che l'algoritmo non riusciva a trovare le mosse corrette da fare perchè doveva ogni volta rieffettuare i calcoli. Abbiamo quindi deciso di utilizzare le transposition tables per memorizzare le posizioni di gioco già esaminate e le loro relative valutazioni e velocizzare i calcoli. Verranno descritte meglio dopo.

6) SCELTE IMPLEMENTATIVE

public int selectColumn(CXBoard B)

Questa funzione è il cuore dell'implementazione del progetto. Prende come argomento il game state attuale e restituisce la colonna in cui eseguire la prossima mossa.

1. Prima di eseguire la ricerca nel game tree, selectColumn si occupa di verificare i casi in cui la scelta della mossa può essere fatta immediatamente (quando la partita è appena iniziata e quando c'è una sola colonna disponibile).
2. Successivamente viene fatta la ricerca nel game tree con alcune caratteristiche chiave:
    - Applicazione dell'Iterative Deepening per garantire che l'algoritmo rimanga entro il limite di tempo stabilito.
    - Utilizzo di Alpha-Beta Pruning per esplorare l'albero di gioco in modo efficiente.
    - Utilizzo di una funzione ausiliaria per ritornare l'indice della colonna dato il punteggio fornito da Alpha-Beta
    - Incremento della profondità al termine di ogni ciclo

private double alphabeta(CXBoard B, boolean myTurn, int depth, double alpha, double beta)

Questa funzione utilizza l'algoritmo Alpha-Beta Pruning ovvero una variante dell'algoritmo Minimax, comunemente utilizzato nei giochi a due giocatori a somma zero.
L'obiettivo di Minimax è trovare la mossa che massimizza il guadagno del giocatore attuale, supponendo che l'avversario giochi sempre in modo ottimale.
Prende come argomenti il game state attuale, il giocatore che deve scegliere la mossa, la profondità e due parametri alpha e beta che rappresentano rispettivamente il valore minimo che il giocatore massimizzante può ottenere e il valore massimo che il giocatore minimizzante può di ottenere. Inizialmente alpha è impostato a meno infinito e beta a più infinito.
L'algoritmo opera ricorsivamente esplorando l'albero di gioco in profondità, decrementandola ad ogni chiamata.
Durante la ricerca, mantiene e aggiorna i valori di alpha e beta in modo da evitare l'esplorazione di rami dell'albero che possono essere sicuramente scartati.
Il funzionamento dell'Algoritmo Alpha-Beta Pruning è il seguente:

1. Se si sta esplorando un nodo massimizzante (il nostro turno), si cerca di massimizzare il valore di alpha. Qualsiasi valore beta minore di alpha può essere ignorato, poiché l'avversario non sceglierà mai una mossa che porti a un risultato peggiore di beta.
2. Se si sta esplorando un nodo minimizzante (il turno dell'avversario), si cerca di minimizzare il valore di beta. Qualsiasi valore alpha maggiore di beta può essere ignorato, poiché non sceglieremo mai una mossa che porti a un risultato peggiore di alpha.
3. Durante la ricerca ricorsiva, se in un qualsiasi punto beta diventa minore o uguale ad alpha, l'algoritmo può interrompere l'esplorazione del ramo corrente, poiché è già stato determinato che questa mossa non è conveniente per il giocatore attuale.
4. Se raggiungiamo una profondità <= 0, raggiungiamo un nodo foglia oppure terminiamo il tempo disponibile allora viene valutato il game state attuale.
5. Grazie all'uso delle transposition tables, vengono calcolati con alphabeta solamente game state mai incontrati fino a quel momento.

private int getBestMove(CXBoard B, boolean myTurn, int depth, double targetScore)

Questa funzione ha il compito di determinare la mossa corrispondente al punteggio ritornato da alphabeta.
Prende come argomenti il game state attuale, il giocatore che deve scegliere la mossa, la profondità e il punteggio da trovare.
Utilizza lo stesso principio dell'Iterative Deepening ovvero effettua una ricerca approfondita solo fino a una certa profondità dell'albero di gioco:

1. La funzione inizia scegliendo casualmente una delle colonne disponibili come mossa iniziale. Questo serve nel caso in cui getBestMove non riesca a trovare la mossa giusta nel tempo disponibile.
2. La funzione itera attraverso le colonne disponibili e, per ciascuna colonna, simula di effettuare la mossa e calcola il punteggio previsto utilizzando alphabeta.

Successivamente, viene annullata la mossa per mantenere lo stato del gioco invariato.

3. Per valutare i punteggi in modo più efficiente viene utilizzata una "soglia di valutazione dei punteggi" (scoreThreshold). Questa soglia consente di accettare una mossa anche se il punteggio ottenuto non coincide esattamente con il punteggio target. Questo aiuta a evitare di esaminare inutilmente mosse che potrebbero essere molto simili in termini di vantaggio.
4. Durante l'iterazione attraverso le colonne disponibili, la funzione tiene traccia della mossa che ha ottenuto il punteggio più vantaggioso. A seconda di quale giocatore sta giocando (myTurn), il punteggio migliore viene inizializzato a Double.NEGATIVE_INFINITY se è il turno del nostro giocatore o a Double.POSITIVE_INFINITY se è il turno dell'avversario.
5. Se la funzione trova una mossa che si avvicina al punteggio target entro il margine di errore specificato dalla soglia, allora restituirà immediatamente quella mossa. Altrimenti se una mossa trovata ottiene un punteggio migliore di quello precedente, la funzione aggiorna la mossa migliore e il suo punteggio.
6. Questo processo continua fino a quando non viene trovata la mossa, tutte le colonne disponibili sono state esaminate o fino a quando il tempo limite di ricerca è stato superato.

private double evaluate(CXBoard B)

Questa funzione è responsabile della valutazione dello stato attuale del gioco, assegnando un punteggio a seconda di quanto sia favorevole o sfavorevole per il giocatore.
È divisa in due parti, la prima parte valuta stati di gioco terminali mentre la seconda valuta stati di gioco intermedi:

- Valutazione di stati di gioco terminali

1. Vittoria del nostro giocatore (myWin): Se lo stato attuale del gioco fa vincere il nostro giocatore, la funzione ritorna Double.POSITIVE_INFINITY. Questo indica una situazione altamente vantaggiosa per il giocatore e rappresenta una mossa che porta direttamente alla vittoria.
2. Vittoria dell'avversario (yourWin): Se lo stato attuale del gioco fa vincere l'avversario, la funzione ritorna Double.NEGATIVE_INFINITY. Questo indica una situazione altamente svantaggiosa per noi e rappresenta una mossa che porterebbe alla sconfitta immediata.
3. Pareggio (CXGameState.DRAW): Se lo stato attuale del gioco è un pareggio, la funzione ritorna 0.0. In questa situazione, nessuno dei giocatori ha un vantaggio significativo, quindi il punteggio è neutro.

- Valutazione di stati gioco intermedi

Le condizioni sopra descritte coprono situazioni di vittoria, sconfitta o pareggio immediato, ma molto spesso il gioco è in uno stato intermedio. In questi casi, la funzione evaluate prende in considerazione altri fattori:

1. Minacce del nostro giocatore (myThreats): Viene utilizzata la funzione ausiliaria *checkForMajorThreats* per identificare se il nostro giocatore ha configurazioni di gioco che potrebbero portare a una vittoria al turno successivo. Se tali minacce sono

presenti, la funzione ritorna Double.POSITIVE_INFINITY, indicando una posizione altamente vantaggiosa.
2. Minacce dell'avversario (yourThreats): Analogamente al punto precedente, la funzione verifica se l'avversario ha configurazioni di gioco che potrebbero portare a una vittoria nel turno successivo. In caso positivo, la funzione ritorna Double.NEGATIVE_INFINITY, indicando una posizione altamente svantaggiosa per noi.
3. Conta delle connessioni (connections): Se nessuna delle condizioni precedenti si verifica, la funzione esegue una valutazione basata sulle sequenze di pedine attualmente presenti nella tavola grazie alla funzione ausiliaria *countConnections*. Questa funzione, a seconda delle potenziali connessioni del nostro giocatore e del giocatore avversario, ritorna un punteggio che rappresenta il potenziale del game state.

private boolean checkForMajorThreats(CXBoard B, boolean myTurn)

Questa funzione è responsabile di controllare se sono presenti possibili minacce gravi, ovvero mosse potenzialmente vincenti del giocatore corrente:

1. Vengono esaminate tutte le colonne disponibili nel gioco utilizzando un ciclo for. Per ogni colonna non piena, la funzione simula la mossa in quella colonna e successivamente verifica se questa mossa porterebbe a uno stato di gioco vincente per il giocatore.
2. In caso di rilevamento di una minaccia grave, la funzione rimuove la mossa simulata e ritorna true. Questo indica che almeno una colonna contiene una mossa potenzialmente vincente per il giocatore corrente.
3. Se nessuna minaccia grave viene rilevata dopo aver esaminato tutte le colonne disponibili, la funzione ritorna false. Questo indica che il giocatore corrente non ha alcuna opportunità di vincere nel prossimo turno attraverso una mossa immediata.

private double countConnections(CXBoard B, boolean myTurn)

Questa funzione è responsabile della valutazione delle connessioni di pedine nell'attuale game state. Essa esamina tutte le colonne disponibili nel gioco alla ricerca di connessioni di pedine orizzontalmente, verticalmente e diagonalmente:

1. Viene inizializzata una variabile bestScore che rappresenta il punteggio migliore trovato durante l'analisi delle colonne disponibili.
2. La funzione esamina tutte le colonne disponibili nel gioco utilizzando un ciclo for. Per ogni colonna non piena, la funzione simula una mossa in quella colonna e calcola il punteggio della colonna corrente (colScore) chiamando tre funzioni ausiliarie:
    - *countRows*: Questa funzione analizza le sequenze di pedine orizzontali, restituendo il punteggio relativo.
    - *countDiagonals*: Questa funzione analizza le sequenze di pedine diagonali, restituendo il punteggio relativo.
    - *countColumns*: Questa funzione conta le connessioni di pedine verticali, restituendo il punteggio relativo.
3. Dopo aver calcolato i punteggi relativi alle connessioni di pedine per il nostro giocatore, viene ripetuto il processo anche per le pedine dell'avversario e poi si

sottrae quel punteggio al nostro punteggio calcolato precedentemente.
Questo passaggio è fondamentale poiché ci permette di non considerare solamente mosse offensive ma anche difensive.

4. Successivamente, la funzione confronta il punteggio della colonna corrente con il punteggio migliore e lo aggiorna solo se è migliore. Questo passaggio determina la colonna con maggiore potenziale.
5. Infine, la funzione rimuove la mossa simulata per ripristinare il game state e continua ad esaminare le altre colonne disponibili per poi ritornare il punteggio migliore

7) STRUTTURE DATI E ALGORITMI NOTI UTILIZZATI

- **Alpha-Beta Pruning**

- **Iterative Deepening**

- **Transposition tables (Hash map)**

Le transposition tables vengono utilizzate per memorizzare le posizioni di gioco già esaminate e le loro relative valutazioni, consentendo così all'algoritmo di evitare di ricalcolare le stesse posizioni più volte durante la ricerca.
Questa classe fornisce un meccanismo per memorizzare e recuperare valutazioni di posizioni di gioco in base a un valore di hash associato.
La struttura dati sottostante utilizzata per memorizzare le valutazioni è una HashMap<Long, Double>. In questa mappa, la chiave è un valore di hash (long) associato ad un game state specifico, mentre il valore è una valutazione (double) di quel game state.
La scelta di utilizzare una HashMap è stata fatta perché consente un accesso estremamente efficiente alle valutazioni memorizzate.

La classe TranspositionTable presenta tre metodi:

1. *store*(long hash, double score): Questo metodo consente di memorizzare una valutazione associata ad una chiave specifica. Viene utilizzato quando l'algoritmo di ricerca valuta una posizione di gioco non ancora incontrata, e la valutazione viene memorizzata nella tabella per evitare di ricalcolarla in futuro.
2. *contains*(long hash): Questo metodo verifica se una chiave specifica è già presente nella tabella. Se è già presente, significa che la posizione di gioco associata è stata precedentemente valutata e memorizzata.
3. *retrieve*(long hash): Questo metodo recupera la valutazione associata a una chiave specifica dalla transposition table.

- **Zobrist Hashing**

Lo Zobrist Hashing è una tecnica utilizzata nell'implementazione di algoritmi di giochi da tavolo per ridurre il tempo di calcolo necessario per valutare posizioni di gioco, consentendo così la memorizzazione efficiente e la ricerca rapida di posizioni già valutate.
Abbiamo utilizzato questa tecnica per calcolare una chiave hash per ogni game state.

La classe ZobristHashing presenta due metodi:

1. *initializeKeys*(): Durante la creazione di un'istanza della classe ZobristHashing, vengono inizializzate le Zobrist keys. Queste chiavi sono generate casualmente e sono associate ad ogni cella possibile nella griglia di gioco.
   Per ogni cella, vengono create due Zobrist keys, una per il giocatore P1 e una per il giocatore P2.
   Queste chiavi vengono poi memorizzate nella matrice zobrist Keys[M][N][2].
2. *calculateBoardHash*(CXBoard B): Viene utilizzato per calcolare la chiave hash per un dato game state rappresentato della tavola di gioco. Per calcolare questa chiave, la classe esamina ogni cella nella tavola e utilizza le Zobrist keys appropriate in base allo stato della cella.
   Se una cella è occupata dal giocatore P1, viene utilizzata la prima Zobrist key associata a quella cella. Se è occupata dal giocatore P2, viene utilizzata la seconda Zobrist key. Tutte le Zobrist keys vengono infine combinate utilizzando l'operatore XOR per ottenere una chiave hash finale che rappresenta in modo univoco il game state.

8) CONTRIBUTI ORIGINALI

- *getBestMove*

- *checkForMajorThreats*

- *countConnections*

- *countRows*

- *countDiagonals*

- *countColumns*

- *highestColumn*

9) ANALISI COMPLESSITÀ COMPUTAZIONALE

Analisi molto approssimativa della complessità temporale e spaziale
Assumiamo M come numero righe, N numero colonne, X numero pedine per vincere

countRows
- Complessità temporale: O(M * N * X)

countDiagonals
- Complessità temporale: O(M * N * X)

countColumns
- Complessità temporale: O(M * N * X)

countConnections
- Complessità temporale:
    - Caso ottimo: O(1) quando viene trovato uno stato di gioco terminale
    - Caso medio e pessimo: Dipendono dal numero di colonne N e dal costo di *countRows*, *countDiagonals*, *countColumns*
      Quindi O(N * (M * N * X))

checkForMajorThreats
- Complessità temporale:
    - Caso ottimo: O(1) la minaccia viene trovata subito
    - Caso medio e pessimo: O(N) dipendono dal numero di colonne

evaluate
- Complessità temporale:
    - Caso ottimo: O(1) viene trovato uno stati di gioco terminale
    - Caso medio e pessimo: Dipendono dalla complessità di *checkForMajorThreats* e *countConnections*
      Quindi O(N) + O(N * (M * N * X)) ovvero O(N * (M * N * X))

getBestMove
- Complessità temporale: Normalmente dipenderebbe dal costo di alphabeta, ma visto che abbiamo utilizzato le transposition tables per salvarci i game state già visitati, quando getBestMove chiama alphabeta, viene sempre utilizzato un valore già memorizzato e quindi il costo dipende solo dal tempo di ricerca nella transposition table ovvero O(1)

alphabeta
Assumiamo m come numero medio di mosse e d come profondità massima
- Complessità temporale di Alpha-Beta Pruning:
    - Caso ottimo: O(sqrt(m^d))
    - Caso medio e pessimo: O(m^d)

Consideriamo solo il caso medio e pessimo e aggiungiamoci il costo di evaluate:
O((m^d) + (N * (M * N * X))) ovvero O (max((m^d) , (N * (M * N * X))))

selectColumn
Assumiamo m come numero medio di mosse e d come profondità massima
- Complessità temporale:
    - Caso ottimo: O(1) quando ricadiamo nei casi immediati
    - Caso medio e pessimo: Dipendono dalla profondità di ricerca massima dal costo di alphabeta
      Quindi O (max((m^d) , (N * (M * N * X))))

10) FONTI

Connect 4: https://en.wikipedia.org/wiki/Connect_Four

Alpha-Beta Pruning: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Transposition Tables: https://en.wikipedia.org/wiki/Transposition_table

Zobrist Hashing: https://en.wikipedia.org/wiki/Zobrist_hashing