Report progetto Algoritmi e Strutture Dati 2022/2023

Elena Lippolis, 0000881754

1 Introduzione

Lo scopo del progetto di Algoritmi e Strutture dati dell'anno accademico 2022/23 è quello di sviluppare un giocatore software per il gioco Connect 4, utilizzando una versione del gioco che si svolge su una matrice di dimensione M x N, con l'obiettivo di allineare X gettoni in modo orizzonatale, verticale o diagonale.
Nel caso specifico del Connect 4 classico, i parametri impostati sono: M=6, N=7 e X=4.
L'obiettivo principale di questo progetto è che il giocatore sviluppato riesca a vincere il più possibile ogni partita contro o un avversario umano, un Computer, tenendo conto di un timer di 10 secondi per mossa.
Nel corso del report, presenterò l'analisi dei requisiti e l'implementazione del player, valutandone il costo computazionale e eventuali possibili miglioramenti o sviluppi futuri.

2 Connect 4

Questo gioco permette a due giocatori di sfidarsi cercando di allineare un certo numero di gettoni consecutivi in una griglia rettangolare. L'obiettivo è completare una fila verticale, orizzontale o diagonale di X gettoni, dove X rappresenta un numero specificato, solitamente 4.
La partita viene giocata su una matrice di dimensione M x N, dove M rappresenta il numero di righe e N il numero di colonne. Ogni giocatore a turno posiziona un gettone nella colonna di sua scelta.
La sfida consiste nel prevedere e impedire le mosse dell'avversario, cercando al contempo di vincere.
Le prestazioni del giocatore implementato verranno paragonate con un certo numero di giocatori a livello crescente, che vanno da un Livello 0 a un Livello 6.
Verranno inoltre testate 35 configurazioni, di cui ognuna viene fatta giocare sia come primo giocatore che come secondo (70 partite totali per coppia).
Per quanto riguarda lo schema di punteggi, in caso di vittoria (regolare o a tavolino) 3 punti, in caso di pareggio 1, in caso di sconfitta 0.

3 Scelte progettuali

Per riuscire a visitare più stati di gioco possibili, e quindi analizzare più nodi possibili di un albero che rappresenta tutte le possibili partite in un gioco a turni, ho deciso di sviluppare l'algoritmo di decisione su alberi Minimax insieme alla tecnica di ottimizzazione Alpha-Beta Pruning.

3.1 Minimax

L'algoritmo di decisione Minimax è un algoritmo ricorsivo per individuare la migliore mossa possibile in un gioco secondo il criterio di minimizzare la massima perdita possibile. Questo algoritmo prevede però l'esplorazione completa dell'albero, che con una limitazione di tempo del timeout di 10 secondi per mossa risulta impossibile. Per risolvere questo problema ho deciso di dare un bound alla profondità dell'albero: non controllerà tutti i nodi possibili ma solo quelli che hanno una profondità minore di quella data. Inoltre ho utilizzato una tecnica di ottimizzazione del minimax, l'Alpha-Beta Pruning.

3.2 Alpha-Beta Pruning

L'Alpha-Beta Pruning sfrutta il fatto che con l'algoritmo Minimax il giocatore che massimizza (max) e il giocatore che minimizza (min) si alternano nel fare le mosse. Durante l'esplorazione dell'albero, l'Alpha-Beta Pruning aggiorna due valori: l'**alpha**, il valore finora migliore per il max, e il **beta**, il valore migliore finora per il min, e utilizza questi valori per "potare" (pruning) i rami dell'albero che influenzeranno la decisione finale.
L'applicazione di Alpha-Beta Pruning consente di ridurre il numero di nodi inesplorati, migliorando l'efficienza dell'algoritmo Minimax. Ne do un esempio nel codice seguente.

```
private int alphabeta(CXBoard board, int depth, int
    alpha, int beta, boolean maximizingPlayer){
        int eval = 0;
        Integer[] columns = board.getAvailableColumns();

        if(depth<=0 || board.gameState()!=CXGameState.
            OPEN || (System.currentTimeMillis()-START)
            /1000.0 > TIMEOUT*(99.0/100.0)){
             eval = evaluateGame(board);
             return eval;
        }

        if(maximizingPlayer){
             eval = Integer.MIN_VALUE;
             for(int col : columns){
                  if(!board.fullColumn(col)){
```

```
                    board.markColumn(col);
                    eval = Math.max(eval, alphabeta(board
                        , depth-1, alpha, beta, false));
                    board.unmarkColumn();
                    alpha = Math.max(alpha, eval);
                    if(alpha >= beta){
                        break;
                    }
                }

            }
        } else {
            eval = Integer.MAX_VALUE;
            for (int col : columns) {
                if (!board.fullColumn(col)) {
                    board.markColumn(col);
                    eval = Math.min(eval, alphabeta(board
                        , depth - 1, alpha, beta, true));
                    board.unmarkColumn();
                    beta = Math.min(eval, beta);
                    if (alpha >= beta) {
                        break;
                    }
                }
            }
        }
        return eval;
    }
```

Nella mia implementazione dell'algoritmo Minimax con Alpha-Beta Pruning, passo come parametri lo stato di gioco (CXBoard board), la profondità, i valori di alpha e beta e se il giocatore deve massimizzare o no (maximizingPlayer viene settato a True se deve massimizzare, a False se invece deve minimizzare).
Se il nodo passato come parametro è una foglia (board.gameState() != CXGameState.OPEN), se il tempo sta per terminare o se ha raggiunto il limite di profondità assegnato, viene restituita una valutazione della situazione di gioco calcolata dalla funzione evaluateGame(board).
Se invece il nodo passato non è una foglia, e non vengono soddisfatti gli altri requisiti, per tutte le colonne ancora disponibili, la funzione aggiunge una tessera, da una valutazione alla nuova situazione di gioco raggiunta chiamando ricorsivamente la funzione alphabeta stessa, toglie la tessera e restituisce la variabile eval. Questa variabile in caso il giocatore voglia massimizzare corrisponde al massimo tra il suo valore precedentemente assegnato e il risultato della ricorsione di alphabeta, nel caso invece di minimizzazione, è il valore minore.

3.3 Strutture dati

Non ho utilizzato strutture dati complesse in quanto non necessarie per un migliore funzionamento del mio programma, se non array per memorizzare tutte le colonne ancora non piene e in cui si potevano aggiungere tessere, e il concetto di alberi. La radice è lo stato iniziale di gioco, i nodi sono gli stati di gioco successivi fino alla vittoria, sconfitta o pareggio di uno dei due giocatori. L'algoritmo Minimax esegue una visita in profondità sull'albero dei possibili stati di gioco. Durante la ricerca, l'algoritmo esplora l'albero dei possibili stati di gioco fino a una determinata profondità, valutando i nodi terminali o i nodi alla massima profondità raggiunta.

3.4 Altre implementazioni

La funzione 'alphabeta' viene chiamata in selectColumn, e seleziona le colonna in cui inserire la tessera. Setto in questa funzione la variabile bestMove a -1 (mossa impossibile in quanto la board va da 0 a N) e bestScore al valore minore possibile.

```
if(board.numOfMarkedCells()==0 && first && board.M==6 &&
    board.N==7){
            return(3);
        } else if(board.numOfMarkedCells()==0 && !first
           && board.M==6 && board.N==7){
            return(1);
        } else if(board.numOfMarkedCells()==0){
            return(board.getAvailableColumns()[board.
                getAvailableColumns().length / 2]);
        } else if(board.numOfMarkedCells()==1){
            CXCell lm = board.getLastMove();
            return(lm.j);
        }
```

In questo frammento di codice controllo se è il primo turno e sono il primo giocatore, inserisco la tessera al centro, mentre se sono il secondo giocatore la inserisco sopra la tessera appena giocata. In questo modo risparmio risorse e tempo di calcolo impiegati inutilmente nell'inserimento della prima mossa di gioco. Inoltre, dal momento che per matrici di dimensioni specifiche è meglio inserire la tessera in una posizione invece che in un'altra, invece di inserirla al centro la inserisco nella posizione che può più facilmente condurre alla vittoria. Inizio poi a iterare su tutte le colonne disponibili, e in base al numero di celle disponibili, quindi allo stato di gioco: se ci sono più celle libere significa che sono all'inizio della partita e quindi dovrò lavorare con una profondità minore, se invece sono poche significa che sono verso la fine, e trovare le mosse migliori risulta più veloce, e posso quindi aumentare il livello di profondità.
In seguito eseguo la prima chiamata della funzione alphabeta come se stessi massimizzando: setto quindi maximizingPlayer a false e prendo il punteggio

più alto tra quello precedentemente salvato e quello restituito da alphabeta. Salvo la colonna della mossa migliore, e se la mossa risultante ha un numero inferiore a 0 o maggiore di N (numero massimo di colonne), o se il timer sta scadendo, restituisce una mossa al centro della board. Se invece non ci sono problemi restituisce la mossa associata al punteggio migliore. Implementazione principale di selectColumn:

```
public int selectColumn(CXBoard board) {
        START = System.currentTimeMillis(); // Save
            starting time
        int bestMove = -1;
        int bestScore = Integer.MIN_VALUE;
        int alpha = Integer.MIN_VALUE;
        int beta = Integer.MAX_VALUE;


[...]


 board.markColumn(col);
                int score = alphabeta(board, depth-1,
                    alpha, beta, false);
                board.unmarkColumn();
                if(score > bestScore){
                    bestScore = score;
                    bestMove = col;
                }
            }
        }

        if(bestMove <0 || bestMove>board.N || ((System.
            currentTimeMillis()-START)/1000.0 > TIMEOUT
            *(99.0/100.0))){
            bestMove = board.getAvailableColumns()[board.
                getAvailableColumns().length / 2];

        }
        return bestMove;
```

Nella funzione alphabeta valuto lo stato di gioco con la seguente funzione evaluateGame(board):

```
private int evaluateGame(CXBoard board){
        int eval=0;
        if(board.gameState() == myWin){
            eval += 200;
        } else if (board.gameState()==yourWin){
```

```
            eval -= 200;
        } else if(board.gameState()==CXGameState.DRAW){
            eval = 0;
        } else {
            eval = evalBoard(board) ;
        }
        return eval;
    }
```

Se lo stato di gioco risulta in una vittoria del mio player, restituisco 200, se invece risulta in una sconfitta, restituisco -200. Se è un pareggio restituisco 0, e se la partita è ancora in gioco faccio una valutazione con un'uteriore funzione che valuta la board calcolando quante tessere in fila ci sono sia del mio giocatore che dell'avversario.

```
private int evalBoard(CXBoard board){
        CXCellState player, opponent;

[...]

        for(int row=0; row<board.M; row++){
            for(int col=0; col<board.N-board.X-1; col++){
                pieceCount=0;
                oppPieceCount=0;
                for(int i=0; i<board.X; i++){
                    if(board.cellState(row, col+i)==
                        player){
                        pieceCount++;
                    } else if(board.cellState(row, col+i)
                        ==opponent){
                        oppPieceCount++;
                    }
                }
                if(pieceCount== board.X -1){
                    score +=80;
                } else if(pieceCount== board.X -2){
                    score +=20;
                } else if(pieceCount== board.X -3){
                    score +=15;
                }
                if(oppPieceCount==board.X -1){
                    score -=80;
                }else if(oppPieceCount== board.X -2){
                    score -=20;
                }else if(oppPieceCount== board.X -3){
                    score -=15;
```

```
                }
                if(board.X!=4){
                    if(pieceCount==board.X-4){
                        score +=10;
                    }
                    if(oppPieceCount==board.X -4){
                        score -=10;
                    }
                }
            }
        }
[...]

return score;
```

La funzione evalBoard, di cui ho messo solo la parte di codice del controllo orizzontale, è una funzione che verifica quante tessere minori di X consecutive ci sono nella board. Essa fa 4 controlli: orizzontale (di cui è presente il codice), verticale, diagonale, e della diagonale inversa (che risultano analoghi). Calcola inoltre il punteggio incrementando o decrementando la variabile score nel caso si trovino X-1, X-2, X-3 (e X-4 nel caso che X sia diverso da 4) tessere consecutive del giocatore o dell'avversario. In caso siano dell'avversario i punti vengono decrementati, sennò vengono sommati. I punti aumentano all'aumentare del numero tessere consecutive.

4 Complessità

4.1 selectionColumn e alphabeta

In selectColumn itero al massimo N volte e chiamo alphabeta con una profondità prefissata ma variabile (da 3 a 20).
Nel caso pessimo la complessità in tempo è

$$O(Nb^d) \tag{1}$$

dove b è il branching factor (numero medio di sottoalberi per ogni nodo) e d è la profondità, quindi la complessità varierà da

$$O(Nb^3) \tag{2}$$

a

$$O(Nb^{20}) \tag{3}$$

La funzione alphabeta avrà lo stesso costo computazionale di selectColumn.

4.2 evaluateGame e evalBoard

La complessità di evaluateGame è una complessità lineare sommata alla complessità di evalBoard, che è O(M*N).

5 Possibili miglioramenti

Si potrebbe pensare di implementare l'algoritmo Monte Carlo Search Tree che, per quanto complesso, risulta essere quello che da i risultati migliori per quanto riguarda questo tipo di giochi.
Ulteriori miglioramenti al codice da me implementato potrebbero essere:

- Transposition Tables: consentono di evitare il ricalcolo di valutazioni e mosse già esplorate in precedenza durante la ricerca nell'albero di gioco.
- Algoritmo di ordinamento delle mosse: consentirebbe di esplorare le mosse dalla più promettente a quella meno durante la ricerca nell'albero di gioco, ottimizzando quindi le prestazioni complessive dell'algoritmo di ricerca e ottenendo una soluzione più rapidamente.