Relazione del Progetto Algoritmi e Strutture Dati

Marco Coppola: 0001020433

A.A. 2021/2022

Indice

1. Introduzione

2. Esplorazione dell’albero di gioco

3. Schemi riassuntivi e pseudocodice

1. Introduzione

Il gioco

Possiamo vedere l'**(M,N,K)-Game** come una generalizzazione del classico tris: una griglia di dimensioni MxN dove due giocatori si sfidano a chi per primo riesce ad allineare K simboli uno di seguito all'altro, orizzontalmente, verticalmente o diagonalmente.

Una proprietà del tris, che si applica anche all' (M,N,K)-Game, è quella di essere entrambi **giochi a somma zero**, ovvero giochi dove il vantaggio (o lo svantaggio) di un determinato giocatore è perfettamente bilanciato da una perdita (o un guadagno) del giocatore avversario[1]. Sommando quindi il vantaggio del primo giocatore con la perdita del secondo (o viceversa) otteniamo appunto zero.

Questa proprietà come vedremo è molto importante in quanto ci permette di applicare l'algoritmo **MiniMax** (o una qualsiasi sua variante), ovvero un metodo per minimizzare la massima perdita possibile ad ogni turno[2].

Il progetto

Lo scopo del progetto è riuscire ad implementare un giocatore software che riesca (mediante opportune euristiche ed algoritmi) a determinare, per ogni turno, la mossa migliore possibile sulla griglia e portarlo di conseguenza ad una vittoria assicurata (ove possibile) oppure ad un pareggio (nel caso in cui anche il secondo giocatore abbia una strategia ottima e la griglia di gioco non avvantaggi nessuno dei due giocatori).

Limitazioni

Risulta evidente che, nonostante un implementazione astuta non sia di tipo bruteforce, effettuare un numero molto elevato di valutazioni sulla griglia di gioco (in crescita esponenziale rispetto alle dimensioni di quest'ultima) è cruciale per ottenere una buona strategia.
Per griglie molto grandi i tempi di ricerca potrebbero essere elevatissimi, pertanto ad ogni mossa è impostato un limite massimo di 10 secondi.
Il progetto non consente l'utilizzo di alcun tipo di parallelizzazione, quindi tutta la ricerca andrà fatta in un unico thread, il che renderà sì l'algoritmo più lento rispetto ad un approccio multi-thread, ma che permetterà un'implementazione ottimizzata al massimo per riuscire ad esplorare il più possibile l'albero di gioco e rientrare nel timeout di 10 secondi.

2. Esplorazione dell'albero di gioco

MiniMax

Come anticipato in apertura, avendo a che fare con un gioco a somma zero, un algoritmo che risolve alla perfezione questo tipo di problema è senza ombra di dubbio il **MiniMax** o una qualsiasi sua variante.

Complessità di MiniMax

Purtroppo per ottenere una strategia perfetta con MiniMax sin dalla prima mossa di gioco, è necessario che l'algoritmo visiti l'intero albero di gioco avente come radice la griglia vuota.
Supponiamo di avere un classico tris, dove la griglia di gioco ha dimensioni 3x3. Allora al primo turno il primo giocatore avrà 9 possibili mosse a disposizione, al secondo turno il secondo giocatore avrà 8 mosse a disposizione e così via. È facile rendersi conto che procedendo in questo modo otteniamo al massimo 9!=362880 foglie, quindi tutto sommato non è un'ipotesi così assurda visitare l'intero albero di gioco ed essere certi di scegliere ad ogni turno la migliore mossa possibile.

Già con griglie di dimensioni 5x5 ci rendiamo però conto che visitare più di 25! nodi risulta non solo impossibile in 10 secondi, ma anche in svariate ore.
E basti pensare che é necessario solamente ingrandire la griglia di pochi spazi per far aumentare esponenzialmente questo numero.

Per ovviare a questo problema, ho implementato alcuni metodi che permettono di tagliare notevolmente i nodi da visitare all'interno dell'albero di gioco, oppure quando l'albero è talmente grande da non permetterne l'esplorazione entro il timeout, una funzione di valutazione parziale riuscirà a predire quasi con certezza la mossa che porterà il giocatore ad avere il maggior vantaggio possibile.

Problemi di MiniMax

Durante l'applicazione di MiniMax ho riscontrato due grandi problemi:

1. **L'algoritmo effettua una ricerca in profondità (DFS)**.
    a. Questo risulta molto scomodo in caso di griglie di dimensioni elevate, in quanto il timeout di 10 secondi molto probabilmente scatta quando MiniMax sta ancora cercando di arrivare ad una foglia del primo sottoalbero (generato dalla prima mossa disponibile per il giocatore).
        i. Questo problema può essere raggirato inserendo una profondità massima di ricerca che evita di scendere troppo in profondità in ogni sottoalbero, non permettendo la ricerca nei rimanenti.
2. **Impostando una profondità massima di ricerca si rischia di non sfruttare al meglio il timeout di 10 secondi**.
    a. È molto difficile infatti stimare il tempo di ricerca sull'albero, si rischia quindi o di dedicare troppo tempo ad un cammino verso una singola foglia e non arrivare ad eventuali foglie più importanti (profondità troppo alta) oppure di avere una profondità troppo ridotta che termina l'esplorazione prima del timeout, non sfruttando a pieno il tempo concesso

La prima idea che mi è venuta in mente è stata quindi quella di convertire la ricerca da DFS a BFS, ma cercando online[3] e ragionando su come funziona in pratica una ricerca in ampiezza, è facile rendersi conto dell'impossibilità di questa soluzione.
Infatti implementare una BFS su un albero che potenzialmente potrebbe presentare miliardi di nodi ad una certa altezza significherebbe aggiungerne altrettanti ad una coda, il che renderebbe non solo l'algoritmo impossibile da eseguire per la quantità di memoria necessaria, ma paradossalmente più lento di una ricerca DFS in quanto ha bisogno di allocare continuamente in memoria una struttura dati di enormi dimensioni.

Iterative Deepening

L'approccio che ho trovato avvicinarsi il più possibile ad una ricerca in ampiezza è quello di utilizzare l'iterative deepening. Questa tecnica consiste sempre in una ricerca depth-first, ma a differenza di una DFS pura in questo caso aumenta progressivamente la profondità di ricerca (partendo da 1) in modo da dare ad ogni cella iniziale della griglia la stessa

importanza. In questo modo sono riuscito a risolvere entrambi i problemi elencati in precedenza, nonostante molti nodi venissero visitati più volte adesso l'algoritmo effettua una ricerca più ampia e più omogenea sull'intero albero, rientrando a pieno nei 10 secondi.

Alpha-Beta pruning

Chiaramente l'algoritmo Alpha-Beta pruning permette di ridurre drasticamente il numero di nodi da esplorare con MiniMax. Il funzionamento è molto semplice: interrompiamo immediatamente la ricerca in un determinato sottoalbero quando in precedenza abbiamo ottenuto un risultato migliore. In questo caso continuare nell'esplorazione risulterebbe solamente una perdita di tempo inutile.

La funzione eval

Non sempre (anzi quasi mai) si riesce ad esplorare l'intero albero di gioco, pertanto capita molto spesso che l'esecuzione dell'algoritmo debba fermarsi ad un nodo non foglia, dove in generale non possiamo dire con immediatezza se la partita potrà essere vinta da un giocatore o da un altro.

Viene quindi subito in mente l'idea di assegnare ad ogni nodo un valore, in base alla situazione sulla griglia. Un modo molto simile è utilizzato negli scacchi, nonostante sia un gioco estremamente complesso rispetto ad un MNK game, una buona funzione di eval sarebbe quella di assegnare ad ogni pezzo un valore e sommare i valori di tutti i pezzi in gioco in quel momento (utilizzando valori negativi per i pezzi del giocatore avversario).

Ipotesi per una buona funzione di eval

1. Controllare quanti simboli un giocatore controlla in un unica riga/colonna/diagonale
    a. Più simboli ci sono più siamo vicini alla vittoria
        i. Non consideriamo però il fatto che potremmo avere simboli alternati tra p1 e p2 e quindi non raggiungere mai una vittoria
2. Contare quanti simboli consecutivi in un'unica riga/colonna/diagonale
    a. Abbiamo più probabilità di vincita se controlliamo più simboli consecutivi
        i. L'avversario potrebbe avere un suo simbolo proprio nel k-esimo spazio consecutivo che ci avrebbe portato alla vittoria, rendendo tutti i simboli piazzati inutili
3. L'idea finale è stata allora di NON tenere in conto quanti simboli venissero piazzati in una determinata riga/colonna/diagonale, ma piuttosto quante possibilità di vittoria avesse ogni giocatore in ognuna di esse.
    a. Sommando tutte le possibilità di vittoria dei due giocatori su ogni riga/colonna/diagonale riusciamo a capire con certezza chi abbia il vantaggio sulla griglia
        i. Es.1: Se il nostro avversario ha 0 possibilità di vittoria allora questo stato è molto vantaggioso
        ii. Es.2: Se esplorando il game tree ci rendiamo conto di avere 0 possibilità di vittoria ci conviene scartare questa strada (a patto che esistano altre strade più vantaggiose)

Una buona funzione di eval non deve essere però troppo difficile da calcolare in quanto porterebbe l'esecuzione dell'algoritmo generale ad essere molto più lenta e quindi impossibilitata ad esplorare molti nodi dell'albero di gioco.

WinCounter

Per evitare allora di scansionare tutte le righe/colonne/diagonali ad ogni passo, ho deciso di mantenere traccia delle possibili vittorie man mano che l'algoritmo esplora in profondità

l'albero di gioco. Ho creato quindi una classe chiamata ***WinCounter***. Ogni riga/colonna/diagonale sulla quale è possibile ottenere almeno una vittoria, è associata ad un corrispettivo **WinCounter**. Questo "contatore" indica appunto il punteggio di ogni riga/colonna/diagonale, in base a quanto un determinato giocatore è vicino alla vittoria. Ogni volta che viene aggiunto un simbolo consecutivo e la vittoria è ancora possibile, il punteggio si alza seguendo la formula: *PunteggioGiocatore += 10^(simboli consecutivi)*.
Nel caso in cui un giocatore abbia molti simboli consecutivi ma non è possibile raggiungere la vittoria per quella riga/colonna/diagonale, il punteggio del WinCounter sarà 0, in quanto non siamo interessati ad avere molteplici simboli consecutivi se non portano a nessuna possibile vittoria.

Questa classe ha permesso un notevole miglioramento dell'algoritmo, in quanto:

1. Quando l'algoritmo non riesce ad arrivare a tutti i nodi foglia, permette una buona valutazione dello stato di gioco, preferendo sempre stati di gioco dove si è in netto vantaggio rispetto all'avversario
2. Non impiega molto tempo per essere calcolata in quanto invece di esaminare tutta la griglia ad ogni passo semplicemente analizza le righe/colonne/diagonali influenzate dall'ultima mossa

Transposition Table e Hashing

Potrebbe capitare, anzi, **è molto probabile** (visto anche il funzionamento dell'iterative deepening) che durante una ricerca nell'albero di gioco si incontrino due stati gioco equivalenti (la griglia di gioco presenta gli stessi simboli nelle stesse posizioni). In questo caso è evidente che è inutile riprocedere e valutare lo stesso stato due volte. Per ovviare a questo problema ho allora implementato delle Transposition Table, che non sono altro che delle Tabelle Hash dove ad ogni stato di gioco viene associata la propria valutazione, così da avere un accesso immediato ad una valutazione (**O(1)**) nel caso in cui sia già stata valutata in precedenza.

Uno degli errori più difficili da risolvere che ho incontrato durante lo sviluppo è stato quello di salvare qualsiasi valutazione all'interno delle TranspositionTable. Questo porta a false assunzioni per MiniMax in quanto salvare valutazioni parziali (quando non siamo arrivati effettivamente ad una foglia, ma l'algoritmo si è interrotto prima utilizzando come punteggio quello del WinCounter) porterà nelle valutazioni future a scartare vie che anche se inizialmente più svantaggiose rispetto ad altre, potevano invece portare alla vittoria.

SqueezedNode

Per ovviare al problema precedente ho allora introdotto il concetto di **SqueezedNode** (Nodo spremuto). Un nodo viene considerato "spremuto" nel momento in cui, a partire da esso, siamo arrivati unicamente a nodi foglia oppure a tagli di alpha-beta pruning. Un nodo quindi **non** è considerato "spremuto" quando anche solo uno dei nodi raggiungibili a partire da esso è stato valutato attraverso una valutazione parziale. In questo modo è evidente che aggiungere unicamente nodi "spremuti" all'interno delle tabelle hash, ci dà l'assoluta certezza che non è più necessario visitare cammini a partire da quel nodo ma che possiamo direttamente leggere la sua valutazione attraverso la tabella.

Hash Differenziale

Durante l'implementazione dell'algoritmo di hashing ho realizzato che analizzare l'intera griglia di gioco ad ogni mossa era un enorme dispendio di tempo. Infatti, se da un particolare stato di gioco di cui abbiamo già calcolato l'hash, volessimo effettuare una mossa (o se volessimo tornare indietro di una) ci basterà modificare l'hash precedente con la nuova mossa per ottenere il nuovo: passiamo così da calcolare l'hash in **O(m*n)** a calcolarlo in

solo **O(1)**. La complicazione di mantenere ogni volta l'hash precedente é gestita perfettamente dall'algoritmo di MiniMax che quando chiama se stesso ricorsivamente passerà tra gli argomenti anche l'hash dello stato precedente.

Simmetrie

Possiamo facilmente espandere il numero di stati con la stessa valutazione facendo una semplice osservazione: ruotare la griglia di gioco di 180 gradi non influenza minimamente la partita in corso, in quanto preserviamo lo stato di gioco e gli eventuali vantaggi ottenuti da un certo giocatore. Nel caso di griglie quadrate il numero di simmetrie aumenta ulteriormente, ottenendo dalla stessa griglia altre 3 con la stessa valutazione (effettuando 3 rotazioni di 90 gradi). Ogni volta che aggiungevo una valutazione in una Transposition Table ne aggiungevo quindi anche le corrispettive simmetrie con la medesima valutazione.

3. Schemi riassuntivi e pseudocodice

Per facilitare la comprensione delle varie tecniche utilizzate, ecco degli schemi riassuntivi sul funzionamento delle parti principali dell'algoritmo.

Select Cell

Algorithm 1: SelectCell

```
maxDepth=1;
bestMove = null;
bestVal = -∞;
while !timeOut do
    bestIterationMove = null ;
    bestIterationVal = -∞;
    for c in FreeCells do
        marca cella c;
        aggiorna WinCounters(c);
        moveVal = MiniMax(maxDepth);
        resetta WinCounters(c);
        undo marca cella c;
        if moveVal > bestIterationVal then
            bestIterationMove = c;
        end
    end
    if iterazione finita prima del timeout then
        //Assumiamo che a profondita' maggiore, questa valutazione sia piu' accurata
        //anche se ha valore minore della precedente bestMove
        bestMove = bestIterationMove;
    end
    maxDepth++;
end
return BestMove
```

MiniMax

Algorithm 1: MiniMax

```
if gioco finito then
    //Solo se vittoria, sconfitta o pareggio
    return (valuta(B), squeezedChildren:true);
end
if tempo sta per scadere ∨ depth ≤ 0 then
    return (valutazioneParziale(B), squeezedChildren:false);
end
if maximizing then
    int MaxValue = -∞;
    bool SqueezedNode = true;
    for c in FreeCells do
        marca cella c;
        long boardHash;
        if abbiamo hash precedente then
            boardHash = diffHash(hashPrecedente, c);
        end
        else
            boardHash = computeHash(B);
        end
        Integer boardValue = EvaluatedStates.getOrDefault(boardHash);
        if boardValue == null then
            //Dobbiamo procedere con l'algoritmo
            aggiorna WinCounters(c);
            moveVal = MiniMax(depth-1, boardHash, alpha, beta);
            boardValue = moveVal.boardValue;
            if !moveVal.squeezedChildren then
                // Se abbiamo usato una valutazione parziale piu' in profondita'
                // non possiamo considerare questo nodo come spremuto
                SqueezedNode = false;
            end
            resetta WinCounters(c);
        end
        undo marca cella c;
        if boardValue > MaxValue then
            MaxValue = boardValue;;
        end
        //Alpha beta
        alpha = Math.max(alpha, MaxValue);
        if alpha ≥ beta then
            return (MaxValue, squeezedChildren:true);
        end
        // Se a questo punto SqueezedNode e' ancora true allora
        // non abbiamo mai raggiunto una valutazione parziale
        if SqueezedNode then
            //Possiamo procedere a salvare la valutazione nelle tabelle hash
            salvaStato(nodeHash, MaxValue);
        end
        return (MaxValue, squeezedChildren:SqueezedNode);
    end
end
else
    simmetrico per minimizing
end
```

Valutazione Parziale

Algorithm 1: ValutazioneParziale

```
if totalP1Score > 0 ∧ totalP2Score ≤ 0 then
    // Abbiamo possibile vittoria e l'avversario non puo' vincere:
    // nel caso peggiore abbiamo un pareggio
    return MAXVALUE-1;
end
if totalP2Score > 0 ∧ totalP1Score ≤ 0 then
    // Non possiamo vincere e l'avversario ha possibile vittoria
    return -MAXVALUE+1;
end
// Ritorna semplicemente una valutazione complessiva
return totalP1Score - totalP2Score;
```

Nota: in totalP1Score e totalP2Score sono salvate le somme di tutti i winCounters

Update Win Counters

Algorithm 1: UpdateWinCounters

```
// Resetta il punteggio di questo WinCounter
P1Score = 0;
P2Score = 0;
//Inizializza i parametri di valutazione
FreeOrP1Consecutive = 0;
FreeOrP2Consecutive = 0;
P1Consecutive = 0;
P2Consecutive = 0;
//Controlla tutte le celle controllate da questo WinCounter
for c in CellsToCheck do
    // Chi controlla questa cella
    targetCellState = B[c.i][c.j];
    if targetCellState == P1 then
        //Controlla P1, resetta la streak di P2 e incrementa la streak di P1
        FreeOrP2Consecutive = 0;
        P2Consecutive = 0;
        P1Consecutive++;
        FreeOrP1Consecutive++;
    end
    else if targetCellState == P2 then
        //Controlla P2, resetta la streak di P1 e incrementa la streak di P2
        FreeOrP1Consecutive = 0;
        P1Consecutive = 0;
        P2Consecutive++;
        FreeOrP2Consecutive++;
    end
    else
        // Cella libera
        FreeOrP1Consecutive++;
        FreeOrP2Consecutive++;
    end
    //Valutazioni
    if FreeOrP1Consecutive ≥ B.K then
        //Piu probabilita' di vittoria per ogni segno consecutivo
        P1Score += Math.pow(10,P1Consecutive);
        //Resetta streaks
        P1Consecutive = 0;
        FreeOrP1Consecutive = 0;
    end
    if FreeOrP2Consecutive ≥ B.K then
        //Piu probabilita' di vittoria per ogni segno consecutivo
        P2Score += Math.pow(10,P2Consecutive);
        //Resetta streaks
        P2Consecutive = 0;
        FreeOrP2Consecutive = 0;
    end
end
```

Fonti e riferimenti

1. Zero-sum game - Wikipedia
2. Minimax - Wikipedia
3. Why does the adversarial search minimax algorithm use Depth-First Search (DFS) instead of Breadth-First Search (BFS)? - Artificial Intelligence Stack Exchange