ALGORITMI E STRUTTURE DATI

MNK-Game Player

Speziali, Filippo 0000978355
Staffolani, Stefano 0000987575

Anno accademico 2020-2021

1 Intro

Il progetto richiedeva di sviluppare un giocatore di MNK-game per la verifica delle conoscenze al termine del corso di Algoritmi e Strutture di Dati del primo anno di Informatica, presso l'università di Bologna. Il progetto è stato sviluppato in Java.

2 Problema

Il problema affrontato è computazionalmente complesso, infatti dagli algoritmi noti in letteratura ovvero Minimax e AlphaBeta-pruning sappiamo che nel caso pessimo il costo computazionale è $\theta(n)$ dove n è il numero di nodi dell'albero d-ario completo e h è la sua altezza:

$$h = log(n+1) - 1 = \theta(log(n))$$

$$n = \sum_{i=0}^{h} d^i = d^{h+1} - 1$$

dunque la complessità è esponenziale. Per AlphaBeta possiamo assumere un numero medio di mosse m e una massima profondità di ricerca d, dunque la complessità computazionale diventa $O(m^d)$. Dobbiamo cercare di visitare prima i nodi più promettenti, poichè così facendo nel caso ottimo la complessità diventerebbe $O(m * 1 * m * 1...) = O(m^{d/2})$.
Per valutarele mosse più promettenti sono necessarie delle euristiche, inoltre per non rivalutare delle configurazioni già viste in precedenza possiamo utilizzare una Transposition Table.

3 Scelte Progettuali

Inizialmente per ridurre il numero di celle su cui effettuare la ricerca della mossa abbiamo usato una matrice di HeuValue, tipo di dato che contiene gli indice della cella e un valore, dove teniamo traccia dello stato di gioco. L'obiettivo è quello di considerare solo le celle libere adiacenti a celle già marcate per la ricerca della mossa migliore. Se matrix[i][j] è occupata dal player1 diamo valore -1, Se matrix[i][j] è occupata dal player2 diamo valore -2, Se matrix[i][j] è libera diamo valore 0. Successivamente per ogni cella che è stata marcata vengono aumentati i valori delle celle adiacenti di 1, se sono libere. Ogni volta che viene fatta o tolta una mossa tramite il metodo $update\_matrix()$ andiamo ad aggiornare i valori della matrice considerando solo la mossa che abbiamo aggiunto/tolto. a questo punto inseriamo nel max heap solo le celle che hanno valore diverso da 0, -1 e -2. Attraverso le funzioni di evaluate regoliamo i valori contenuti nel max heap a seconda di importanza. Utilizzando il metodo $extract\_max$ verranno poi scelti per la selezione delle celle.

3.1 AlphaBeta Pruning

Per l'analisi delle mosse cerchiamo di ricostruire il game tree fino a una certa profondità. I nodi rappresentano una situazione di gioco, gli archi sono le mosse giocabili a partire da un nodo e le foglie sono gli stati finali di gioco. Per esplorare questo albero e valutare le mosse migliori da compiere utilizziamo l'algoritmo di minimax. L'algoritmo di minimax è un algoritmo ricorsivo e usa il criterio di minimizzare la massima perdita possibile. Inoltre per minimizzare il numero di nodi valutati da minimax utilizziamo l'algoritmo alpha beta pruning. Questo algoritmo fermerà la ricerca in un punto dell'albero se il sottoalbero non può contenere una soluzione ottima. Per minimizzare utlteriormente la ricerca abbiamo usato delle funzioni di euristica che permettono di iniziare a visitare l'albero dalle mosse che crediamo essere più promettenti

3.2 Transposition Table

La Transposition Table è una HashTable dove le chiavi vengono hashate con una specifica funzione che prende in input la board di gioco(le celle marcate). Nella Transposition Table salviamo un HeuValue che contiene lo score e la mossa migliore di quella configuarazione. Questa funzione di hash è detta Zobrist Hash ed è qui implementata in pseudocodice.

Algorithm 1 Zobrist

```
procedure INIT_TABLE(int M, int N, int[ ][ ][ ] table)
    for (int i ← 0; i < M; i++) do
        for (int j ← 0; j < N; j++) do
            table[i][j][0] = Random.Integer();
            table[i][j][1] = Random.Integer();
        end for
    end for
end procedure

procedure HASH(MNKCell[ ] MC)                ▷ array di celle marcate
    int h ← 0;
    for MNKCelld : MC do
        if (d.state == Player1) then    ▷ La cella  marcata da Player1
            h ← h ⊕ table[d.i][d.j][0];
        else                            ▷ La cella  marcata da Player2
            h ← h ⊕ table[d.i][d.j][1];
        end if
    end for
    return h;
end procedure
```

Come possiamo vedere Zobrist fa uso di una tabella MxNx2 dove ogni cella è occupata da un intero random. La funzione di hash scorre le celle marcate

e in base al giocatore che ha marcato la cella calcola lo xor. Il risultato di questa funzione sarà un intero e sarà anche la chiave che useremo per la Transposition Table. La transposition table viene svuotata quando si riempie.

3.3 MaxHeap

Il MaxHeap viene utilizzato per prendere i valori più promettenti per fare AlphaBeta-pruning, i valori più promettenti sono tutte le celle adiacenti a quelle marcate. Quindi le celle promettenti sono al più otto per ogni cella marcata, ovvero tutte quelle intorno. Quanto può essere al massimo la dimensione dell'heap quindi?

Lemma 1. *Il numero massimo di celle marcate che massimizza la grandezza dell'heap è dato dalla funzione*

$$f(M,N) = \left\lceil \frac{M}{3} \right\rceil * \left\lceil \frac{N}{3} \right\rceil$$

Dimostrazione. Banalmente cerco di inserire il minor numero di celle sull'altezza e sulla lunghezza, dato che l'area marcata sarà al più un quadrato $3x3$, le celle marcate in altezza saranno $M/3$ arrotondato per eccesso, mentre il lunghezza $N/3$ (sempre arrotondato per eccesso). Il risultato è ottenuto come la formula dell'area del rettangolo, ovvero $base * altezza$. □

Lemma 2. *Sia*

$$g(M,N) = \frac{(M*N) - f(M,N)}{(M*N)} = 1 - \frac{f(M,N)}{(M*N)}$$

la funzione che calcola il rapporto tra la dimensione dell'heap e la dimensione della board, il valore massimo di questa funzione è $\frac{8}{9}$.

Dimostrazione. Per dimostare il valore massimo di g è $\frac{8}{9}$ valutiamo 4 casi distinti, il dominio di M,N è $D : Z \geq 3$:

1. $(3 \mid M) \wedge (3 \nmid N)$

   $$g(M,N) = 1 - \frac{\frac{N}{3} * (\frac{N}{3} + 1)}{M*N} = 1 - \frac{(N+3)}{9N} = 1 - \frac{1}{9} - \frac{3}{9N} = \frac{8}{9} - \frac{1}{3N}$$

   che con l'aumentare di N tenderà a $\frac{8}{9}$

2. $(3 \nmid M) \wedge (3 \mid N)$ analogo a 1.

3. $(3 \mid M) \wedge (3 \mid N)$

   $$g(M,N) = 1 - \frac{\frac{M}{3} * \frac{N}{3}}{M*N} = 1 - \frac{1}{9} = \frac{8}{9}$$

4. $(3 \nmid M) \wedge (3 \nmid N)$

   $$g(M,N) = 1 - \frac{(\frac{M}{3}+1)*(\frac{N}{3}+1)}{M*N} = 1 - \frac{(M+3)(N+3)}{9MN} = 1 - \frac{1}{9} * (1+\frac{3}{M}) * (1+\frac{3}{N})$$

   che ponendola $\leq \frac{8}{9}$ otteniamo:

   $$1 - \frac{1}{9}(1+\frac{3}{M})(1+\frac{3}{N}) \leq \frac{8}{9}$$

   $$(1+\frac{3}{N})(1+\frac{3}{M}) \geq 0$$

   che risulta vera $\forall N, M \in D$

Abbiamo quindi dimostrato che il valore massimo di g è $\frac{8}{9}$ che si ottiene quando $(3 \mid M) \wedge (3 \mid N)$. □

Dunque la dimensione massima dell'heap sarà $g(M,N) * (M * N)$ che può avere valore massimo di $\frac{8(MN)}{9}$ e questa dimensione diminuirà dopo che saranno state marcate $f(M,N)$ *celle*, quindi dopo $f(M,N)$ *turni*.

3.4 Heuristic

Le euristiche che abbiamo implementato si basano sulle possibili threat che un giocatore può compiere. Una threat è una mossa che se non viene ostacolata può portare alla vittoria. Ci sono diversi tipi di threat che abbiamo considerato e a seconda della loro pericolosità viene assegnato un valore di euristica più o meno alto. Per illustrare le possibili threat che abbiamo deciso di valutare considero il caso del gioco Gomoku (15x15) dove k = 5.
La nostra funzione evaluate per ogni cella libera che si trova vicina a una cella già marcata controlla sulla riga, sulla colonna e sulle diagonali la quantità di caselle giocate dallo stesso giocatore. Se il numero di caselle dello stesso tipo raggiunge k-2 considero 2 casi: Nel caso in cui la sequenza abbia entrambe le estremità libere assegno 100 come valore di euristica. Nel caso in cui la sequenza abbia almeno un estremità libera assegno 10 Se il numero di caselle dello stesso tipo raggiunge k-1 considero 2 casi: Nel caso in cui la sequenza abbia entrambe le estremità libere assegno 1000000 (mossa vincente) Nel caso in cui la sequenza abbia almeno un estremità libera assegno 10000 per il giocatore avversario vengono dati gli stessi valori ma negativi infine la mia funzione evaluate ritorna la somma dei valori di euristica del primo giocatore A più la somma del secondo giocatore B, f = A + B (dove B è un numero minore o uguale a 0).
Le funzioni che mi determinano l'importanza di una cella libera vengono sia usate su tutta la board per la funzione evaluate chiamata quando Alpha Beta Pruning raggiunge depth = 0, sia per valutare singolarmente le celle libere e disporle nell'heap da cui verranno estratte per valutarle nella ricerca della mossa vincente. Nella seguente immagine mostriamo le threat che abbiamo considerato:

(a) corrisponde a k-1 + la cella che stiamo valutando = k (1000000)

(b) analogo di a per entrambe le posizionio considerate

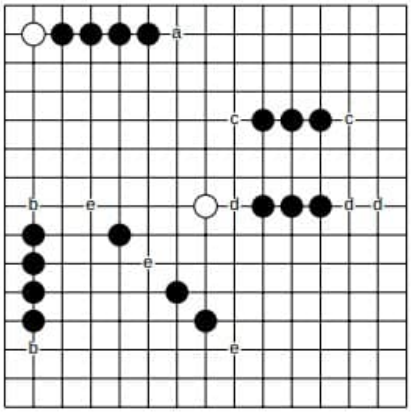
(c) corrisponde a k-2 + la cella che stiamo valutando = k-1 con entrambe le estremità libere (1000000)

(d) partendo da sinistra:

1. corrisponde a k-2 più la cella che stiamo valutando = k-1 con una sola estremità libera (10000)
2. corrisponde a k-2 più la cella che stiamo valutando = k-1 con entrambe le estremità libere (1000000)
3. non viene valutato nella nostra euristica

(e) partendo dall'alto:

1. corrisponde a k-4 + la cella che stiamo valutando = k-3 (0)
2. corrisponde a k-2 + la cella che stiamo valutando = k-1 con entrambe le estremità libere (100000)
3. corrisponde a k-3 + la cella che stiamo valutando = k-2 con entrambe le celle libere (1000)



3.5 SelectCell

La funzione SelectCell prende in input le celle marcate e le celle libere. Come prima cosa controlliamo tramite le celle marcate se la configurazione in input è già stata valutata. Se è già stata valutata e quindi abbiamo già incontrato la configurazione nell'albero di gioco allora ritorniamo il valore della cella da marcare. Altrimenti si tratta di una nuova configurazione su cui dobbiamo esplorare l'albero di gioco e fare una nuova valutazione. Prima di esplorare l'albero dobbiamo ridurre le celle su cui agirà l'algoritmo AlphaBeta. Creiamo una matrice MxN che usiamo per considerare solo le mosse adiacenti a celle già marcate. Successivamente grazie a questa matrice costruiamo un max heap composto dalle sole celle libere adiacenti a celle già marcate. Aggiorniamo i valori del max heap utilizzando delle funzioni

di valutazione delle celle in modo tale da averle ordinate in ordine di importanza. Infine è presente un ciclo dove analizziamo tutte le mosse presenti all'interno dell'heap utilizzando $extract\_max()$. Per ogni mossa che estraiamo chiamiamo alpha beta pruning che ci ritornerà uno score. Alla fine la funzione Select Cell ritorna la mossa che ha ottenuto uno score più alto.

Algorithm 2 SelectCell

```
procedure SelectCell(MNKCell[ ]FC, MNKCell[ ]MC)
    MNKCell ret_value ← new MNKCell(m/2, n/2);
    HeuValue tab_val ← TranspositionTable.get_val(MC);
    if (tab_val.val > MIN_VAL) then                    ▷ valore minimo
        HeuValue tmp ← new HeuValue(tab_val.i, tab_val.j);
        TranspositionTable.remove_val(MC);
        Marca la cella;
    end if
    score ← MIN_VAL;
    bestScore ← score;
    Init Matrix
    MaxHeap max_heap ← new MaxHeap(matrix, m, n)
    for (node ∈ max_heap) do
        Heuristic(node);              ▷ aggiorno i valori tramite evaluate
    end for
    max_heap.heapify();
    while (timenotfinished ∧ max_heap.last ≥ 1) do
        HeuValue e ← max_heap.extract_max();
        Marca la cella e;
        aggiorna matrice;
        score ← AlphaBeta(false, MIN_VAL, MAX_VAL, depth);
        aggiorna matrice;
        Togli la cella marcata precedentemente;
        if (score > bestScore) then
            bestScore ← score;
            ret_value ← (e.i, e.j);
        end if
    end while
    Marca la cella e.i, e.j;
    return ret_value
end procedure
```

4 Analisi Costo Computazionale

4.1 Zobrist

Il costo computazione del calcolo di Zobrist hash è $O(MC[\ ])$ che è un $O(M * N)$

4.2 Heuristic

Il costo di Heuristic equivale a contare fino a k per ogni direzione, quindi 8 direzioni. Il costo sarà dunque $O(8 * K) = O(K)$

4.3 MaxHeap

Abbiamo detto che la dimensione dell'heap sarà al più $O(\frac{8}{9}M * N)$ e tenderà a diminuire, ma per il calcolo consideriamo $O(M * N)$, quindi le operazioni sull'heap avranno costo:

- $fix\_heap() = O(log(M * N))$
- $heapify() = O(M * N)$
- $extraxt\_max() = O(log(M * N))$

4.4 Alphabeta

Dentro AlphaBeta vengono fatte delle chiamate ricorsive, abbiamo impostato che per ogni chiamata vengono fatte le m mosse più promettenti. Ad ogni chiamata viene chiamata Heuristic su ogni valore dell'heap, il costo è di $O(M * N) * O(K) = O(MNK)$. Il numero di chiamate di funzione è m^d. Ad ogni ciclo viene eseguito un $extract\_max()$ con costo $O(log(MN))$ e dunque il costo sarà $O(m * log(MN))$. Considerando tutte le chiamate ricorsive il costo computazionale di AlphaBetaPruning sarà $O(m^d MNK) + O(m^d MNlog(MN))$. Per il calcolo abbiamo assunto che i get dalla transposition table non fallissero mai. Se avessero fallito il costo sarebbe variato di $O(M^2N^2)$ per ogni chiamata, dunque il costo sarebbe stato $O(m^d M^2N^2K) + O(m^d M^2N^2log(MN))$.

4.5 selectCell

Dentro SelectCell viene fatta una chiamata con Heuristic che ha costo $O(K)$, che verrà poi inglobata dalle chiamate di AlpahBeta, che sono $O(HeapSize)$, quindi il costo di SelectCell è $O(m^d M^2N^2log(MN))$.