Catalini Yonas Ernesto: #0000987636 - Rovelli Gianmaria: #0000970936

Algoritmi e Strutture Dati
Progetto ASD 2021 - MNK Game

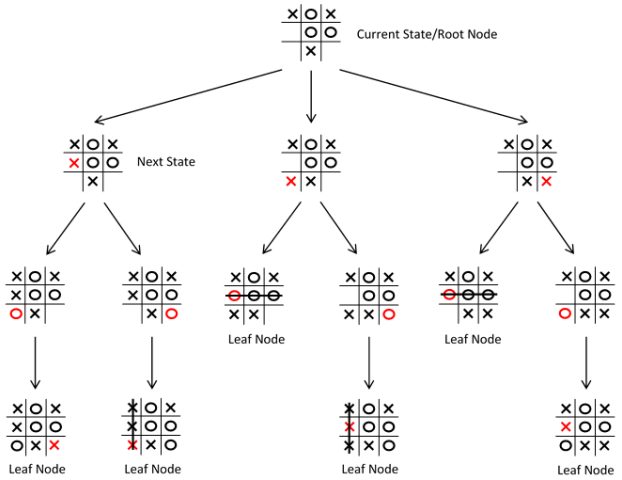
Introduzione

Lo scopo del progetto è quello di sviluppare un algoritmo che sia capace di giocare correttamente a **MNK Game**: un gioco in cui due partecipanti si alternano nel piazzare il loro simbolo su una tabella di grandezza M*N, cercando di creare una fila orizzontale, verticale o diagonale di lunghezza K, nel contempo impedendo all'avversario di fare lo stesso.

MNK è un gioco a **somma zero** e ad informazione **completa** e **perfetta**, ovvero ogni giocatore conosce le possibili strategie dell'avversario, ed è a conoscenza di tutte le mosse effettuate fino a quel punto nella partita. Queste caratteristiche permettono di utilizzare, con lo scopo di scegliere la mossa più opportuna, vari tipi di algoritmi.

Sviluppo e Scelte Progettuali

Minimax e Alpha-Beta Pruning



Alla base delle scelte del player abbiamo il **minimax con potatura alfa-beta:** questo algoritmo di ricerca ricorsivo va ad analizzare, partendo da uno stato del gioco, la bontà di ogni possibile mossa.
L'intera partita viene vista in maniera astratta come un albero, dove ogni nodo esprime una mossa possibile a partire dal nodo precedente. Il principio del minimax si occupa di minimizzare la massima perdita possibile visitando l'albero in profondità e valutando ogni nodo in base ai suoi figli.

Essendo la **complessità computazionale** dell'algoritmo minimax **NP completa**, esso è poco pratico per ottenere una soluzione definitiva in tempo utile, specialmente quando il tabellone di gioco supera le dimensioni del normale tris. Per questo si limita la ricerca attraverso l' alpha-beta pruning e l'impostazione di una profondità di ricerca massima.

La potatura alfa-beta si occupa di ridurre drasticamente il numero di nodi visitato dall'algoritmo, andando appunto a "tagliare" rami che sarebbe inutile visitare. L'efficacia è direttamente proporzionale all'ordinamento dell'albero.

Una volta raggiunta la profondità massima di ricerca, un apposito metodo di valutazione si occupa di valutare il nodo raggiunto. Sarà in base a questi valori che il minimax sceglierà la direzione più opportuna da prendere.

```
01 FUNZIONE alfa_beta(nodo, profondità, α, β, massimizza)
02      SE profondità = 0 O nodo è terminale
03          RESTITUISCI valore euristico del nodo
04      SE massimizza
05          v := -∞
06          PER OGNI figlio del nodo
07              v := max(v, alfa_beta(figlio, profondità - 1, α, β, FALSO))
08              α := max(α, v)
09              SE β ≤ α
10                  INTERROMPI IL CICLO (* taglio secondo β *)
11          RESTITUISCI v
12      ALTRIMENTI
13          v := +∞
14          PER OGNI figlio del nodo
15              v := min(v, alfa_beta(figlio, profondità - 1, α, β, VERO))
16              β := min(β, v)
17              SE β ≤ α
18                  INTERROMPI IL CICLO (* taglio secondo α *)
19          RESTITUISCI v
```

Ordinamento dell'albero di gioco

Per aumentare l'efficacia della potatura, abbiamo utilizzato diverse tecniche che ci permettono di valutare preventivamente le mosse immediatamente disponibili, in modo tale da poterle poi ordinare.

La qualità dei passi disponibili viene valutata in base a diversi criteri, tra cui il numero di celle consecutive ottenute, la somma delle vittorie possibili (del player e dell'avversario), e la presenza di eventuali vittorie immediate.

Il valore che identifica la qualità di ogni mossa viene salvato in una matrice equidimensionale al tabellone di gioco, e poi utilizzato come chiave all'interno di una priority queue utilizzata per ordinare le mosse.

Strutture dati usate

- **Array:** sono stati utilizzati array bidimensionali per descrivere la mappa di gioco e la "mappa delle vittorie", ovvero la matrice che contiene i valori sfruttati per ordinare l'albero. Degli array unidimensionali vengono utilizzati per tenere traccia delle mosse effettuate, di quelle disponibili e per tenere conto del numero di file di lunghezza *i* ottenibili con una determinata mossa.
  La scelta è ricaduta su questa struttura dati in quanto permette di accedere direttamente agli elementi con costo O(1)
- **Liste:** sono state usate all'interno del minimax per la loro semplicità ed efficienza dell'inserimento e rimozione di un valore
- **Priority Queue:** è stata scelta per permetterci di ordinare l'albero delle mosse, in quanto la struttura dati più efficiente disponibile ed adeguata al nostro scopo

Strategie Utilizzate

Le strategie scelte per rendere l'algoritmo in grado di effettuare scelte ottimale in tempi contenuti sono :

- Ordinamento dell'albero prima dell'esecuzione vera e propria del minimax tramite euristiche
- Il metodo che verifica la vittoria non si limita solo a ciò, ma memorizza anche le combinazioni ottenibili con quella determinata mossa (file da 3/4/...)
- Sono state individuate due particolari situazioni che portano particolare vantaggio al giocatore o all' avversario, chiamate "secure" e "almost_secure"
    - **secure** : situazione di vittoria sicura, si verifica quando in una partita otteniamo una fila di lunghezza K-1 con estremità libere

    - **almost_secure** : situazione di vittoria quasi sicura, si verifica quando in una partita otteniamo una fila di lunghezza K-2 con estremità libere e capacità di raggiungere la lunghezza K. Se una casella ci permette di ottenere **due almost_secure**, quest'ultima verrà considerata come una **secure**.
- Se **M>= K +2** e **N >= K +1** o **viceversa** e siamo i primi a giocare la scelta ottimale come prima mossa della partita sarà la casella **2,2**, questa euristica è stata trovata tramite numerose partite e prove. Il che ci permette almeno per la prima mossa (che è anche quella computazionalmente più costosa) di ridurre il costo dell'algoritmo a O(1)

Pseudocodice

```
01 FUNZIONE selectCell(FC, MC)
02      SE unica mossa disponibile
03          RESTITUISCE questa mossa
04      SE prima mossa della partita & rispetta dei criteri euristici
05          RESTITUISCI mossa in posizione 2,2
06      Inizia il timer
07      Calcolo mappa delle vittorie
08      Creo un queue di FC ordinata in base alla mappa delle vittorie
09      Verifico vittorie/perdite immediate e secure mie/nemiche
10      SE M <= 9 & N <= 9 ALLORA MAX_DEPTH = 5
11      ALTRIMENTI MAX_DEPTH = 3
12      r = -INF
13      SE M <= 15 && N <= 15
14          PER OGNI mossa libera
15               a = alphabeta(mossa, depth+1, -INF, +INF)
16               a += (valore in mappa delle vittorie) / 10
17               r = max(r, a) // cambia anche mossa scelta
18      ALTRIMENTI
19          Scelgo la mossa in cima alla pQueue
20      RESTITUISCI mossa scelta
```

Complessità

isWinningCell: controlla in tutte e 4 le direzioni possibili per verificare se mettendo un simbolo in una determinata cella si termini la partita, oppure quanti simboli consecutivi si hanno. Nel caso peggiore questa operazione va a verificare quasi completamente ogni direzione, soltanto per trovare una vittoria nell'ultima. Questa operazione avrebbe costo O(3(k-1)+k)=O(4k)=O(k)

Controlla poi se si sono verificate delle secure o almost secure, con costo O(1).
Esegue infine un ciclo O(k).
Costo totale: O(k).

count: Count utilizza isWinningCell (costo O(k)) , poi countMieMosse (costo O(m+n)) e infine un ciclo che ripete comandi O(1) k volte, (costo O(k)).
Costo totale: O(m+n+2k)=O(m+n+k). In quanto k<=min(m,n), O(m+n+k)=O(m+n+min(m,n))=O(m+n)

mappaVittorie: Questo metodo utilizza count su ogni casella. Il costo totale è quindi $O(mn(m+n))=O(m^2n+mn^2)$

countMieMosse: controlla in tutte le direzioni a partire da una singola cella, e si ferma soltanto quando trova una casella nemica o il limite della mappa. Nel caso peggiore andrà a controllare fino ai limiti del tabellone in tutte e 4 le possibili direzioni, con costo O(m+n+2min(m,n))=O(m+n).
Costo totale: O(m+n)

eval: utilizza un ciclo O(k) e due volte isWinningCell (costo 2*O(k)=O(k)).
Costo totale: O(k)

alphabeta: Nel caso pessimo, ovvero in un albero completamente disordinato, il costo è pari a $O(b^d)$, dove b è il fattore di diramazione e d è la profondità di ricerca. Nel caso peggiore la profondità è pari a 5, e all'inizio della partita il fattore di diramazione medio è pari a:

$$b = \frac{\sum_{i=0}^{d-1}(mn-i)}{d} = \frac{mn+(mn-1)+\ldots+(mn-(d-1)))}{d} =$$

$$\frac{dmn-(\frac{(d-1)d}{2})}{d} = mn - \frac{d-1}{2} \leq mn - 2 = O(mn)$$

$$b^d = O((mn)^5)$$

Tuttavia nel caso si utilizzi un albero ordinato, il costo del minimax con alpha-beta pruning scende a $O\sqrt{(mn)^5}$

selectCell: oltre a varie assegnazioni O(1), all'interno del metodo troviamo:

- mappaVittorie: O(mn(m+n))
- Un ciclo che che si ripete per ogni casella libera (quindi O(mn) volte). All'interno del ciclo:
    - Viene aggiunta la casella libera alla priorityQueue, con costo O(log(mn)) nel caso peggiore.
    - Si utilizza il metodo isWinningCell(), che ha costo O(k)
    - Viene richiamata la funzione eval() con costo O(k)
- Viene lanciato l'algoritmo alfabeta (si rimuovono anche degli elementi dalla priority queue ma il costo viene oscurato da quello dell'algoritmo di ricerca) con costo $O((mn)^5)$.

Il costo totale di selectCell è quindi $O((mn)^5)$nel caso peggiore, e $O\sqrt{(mn)^5}$nel caso ottimale.