PROGETTO MNK-GAME ASD 2022

INTESTAZIONE

1) Nome del Progetto: MrRobot
2) Nome giocatore MrRobot
3) Progetto realizzato da: Irene Burri (916395) e Beatrice Zamagna (914649)

DESCRIZIONE DEL PROBLEMA

1) SCOPO DEL PROGETTO
   Lo scopo del progetto è quello di implementare un algoritmo che riesca a valutare le mosse migliori in un MNK Game, ovvero una generalizzazione del comune tris in cui però abbiamo una griglia di dimensione MxN e in cui per vincere bisogna allineare K simboli.

2) PRINCIPALE ALGORITMO ADOTTATO E OTTIMIZZAZIONI
   Il giocatore implementato sfrutta uno dei metodi più noti della teoria delle decisioni, ovvero il *minimax*, utile per minimizzare la massima perdita in giochi che prevedono mosse alternative come il tris o gli scacchi. In questi casi una delle due parti tenta di massimizzare il punteggio (nostro giocatore) mentre l'altro tende a minimizzare (l'avversario). Tale algoritmo è stato ottimizzato grazie alla potatura alphabeta che ha l'obiettivo di ridurre i nodi da valutare nell'albero. Infine, per cercare di garantire una buona efficienza, abbiamo deciso di limitare la ricerca ad un numero prefissato di livelli, stabilendo dei range basati sul numero di celle libere in quella configurazione. Per le board molto grandi abbiamo dovuto fare delle scelte per cercare di bilanciare sia la precisione delle mosse, che l'efficienza in tempo (timeout di 10 secondi).

3) CONSEGUENTI PROBLEMI DA RISOLVERE
   Dopo aver implementato l'algoritmo fondamentale del nostro progetto (minimax con potatura alphabeta), il principale problema che abbiamo dovuto affrontare è stato quello della valutazione delle configurazioni. Imponendo un numero massimo di livelli all'albero, la valutazione di una "foglia" poteva prevedere sia configurazioni finali che configurazioni aperte.

   Per le configurazioni finali abbiamo deciso di assegnare punteggio di 1000 nel caso in cui rappresentasse una vittoria per il nostro giocatore, -1000 nel caso rappresentasse una configurazione di vittoria per l'avversario oppure 0 nel caso si trattasse di una patta. Tale punteggio varia a seconda del livello in cui si trova tale configurazione, in modo da dare la precedenza alle vittorie più vicine e alle sconfitte più lontane.

   Per le configurazioni non finali abbiamo dovuto ideare una euristica in modo da assegnare ad ogni configurazione un punteggio in base alla validità dell'ultima mossa eseguita, sempre considerando come obiettivo quello di vincere nel minor numero di mosse possibili. La soluzione che abbiamo implementato prevede l'identificazione dell'ultima cella segnata e per essa viene fatta una valutazione sulla riga, colonna e diagonali a cui appartiene in modo da capire la validità di tale mossa. Questa valutazione viene fatta su tutte le celle libere per poi scegliere quella migliore.

4) VALUTAZIONE DELL'ARRAY (riga, colonna, diagonale, antidiagonale)
   Dopo aver trasformato la riga (o la colonna o le diagonali) in un vettore era necessario capire come valutare la situazione in quell'array considerando l'ultima mossa fatta. Abbiamo quindi identificato dei casi generali e assegnato ad ognuno di loro un punteggio (positivo nel caso in cui quella configurazione fosse favorevole al nostro giocatore, negativa altrimenti). Ad ogni array viene assegnata la somma di tutte le casistiche riscontrate e ad ogni cella viene assegnato il punteggio massimo considerando riga, colonna e diagonali di appartenenza.

I casi che abbiamo identificato sono i seguenti

- Valuto una mia ipotetica mossa, ovvero lastCell.state == myMove
  Configurazioni favorevoli:
  1. Ho un sottovettore max di lunghezza >= k, di cui ha occupate più di k/2 celle: +30
  2. Ho un sottovettore max di lunghezza >= k, di cui ha occupate meno di k/2 celle: +20
  3. Blocco un suo sottovettore di lunghezza >= k, di cui ha occupate più di k/2 celle: +10
  4. Blocco un suo sottovettore di lunghezza >= k, di cui ha occupate meno di k/2 celle: +5
  5. Ho bloccato un suo sottovettore lungo k-1 utilizzando justone(): return 1000

  Configurazioni sfavorevoli:

  5bis. Non sono riuscita a bloccare un suo sottovettore lungo k-1 dell'avversario, al turno dopo vincerà utilizzando riskySituation(): return -1000
- Turno avversario (valuto una mia ipotetica mossa, ovvero lastCell.state == yourMove):
  Configurazione sfavorevoli:
  1. Ha un sottovettore max di lunghezza >= k, di cui ha occupate più di k/2 celle: -30
  2. Ha un sottovettore max di lunghezza >= k, di cui ha occupate meno di k/2 celle: -20
  3. Bloccato mio sottovettore di lunghezza >= k, di cui ha occupate più di k/2 celle: -10
  4. Bloccato mio sottovettore id lunghezza >= k, di cui ha occupate meno di k/2 celle: -5
  5. Ha bloccato un mio sottovettore lungo k-1 utilizzando justone: return -1000

  Configurazioni favorevoli:

  10bis. Non è riuscito a bloccare un mio sottovettore lungo k-e dell'avversario, al turno dopo vincerò utilizzando riskySituation(): return 1000

I casi 5, 5bis, 10 e 10bis sono stati aggiunti per fare un lavoro ancora più preciso nei casi in cui tali configurazioni ci avrebbero portato, alla mossa successiva, direttamente a una nostra vittoria o ad impedire una vittoria avversaria. Infatti notiamo che hanno punteggi prioritari.

SCELTE PROGETUALI

1) selectCell(): Se nella board è rimasta una sola cella disponibile allora viene restituita. Altrimenti viene scelta una cella in modo casuale tra quelle libere e poi viene richiamata la vera e propria funzione di scelta della cella migliore: si controllano tutte le celle libere in una configurazione e ognuna di esse viene marcata di volta in volta come occupata e si chiama *alphabeta* a partire da quella nuova configurazione. Alphabeta tornerà il valore di tale configurazione e la cella verrà demarcata. La scelta ricadrà sulla cella con valore più alto (dato che il nostro giocatore deve massimizzare il punteggio).

2) alphabeta(): è una funzione ricorsiva che va in profondità fino a raggiungere il caso base. Se rientriamo in questa casistica significa che siamo arrivati all'ultimo livello dell'albero e dobbiamo valutare le "foglie". Per questo chiamiamo la funzione *score*. La profondità massima dell'albero è stata decisa in base al numero di celle libere e al tempo di computazione necessario.

3) score(): se ci troviamo in una configurazione di vittoria per il nostro giocatore il punteggio assegnato è 1000, in modo che venga scelta come configurazione prioritaria da perseguire. Abbiamo aggiunto come dettaglio quello di diminuire il punteggio a seconda del livello a cui si trova la configurazione finale, in modo che venga data maggiore priorità alle vittorie più vicine. Discorso opposto per il caso in cui la configurazione che stiamo analizzando è di vittoria per l'avversario; ad esso viene dato un punteggio di -1000 che verrà modificato a seconda della profondità in cui ci troviamo, in questo modo diamo la precedenza alle sconfitte più lontane. Nel caso di pareggio diamo punteggio 0. In alternativa a questi tre casi richiamiamo la funzione *scoreNotLeaf.*

4) scoreNotLeaf(): come prima operazione prende l'ultima cella segnata, ovvero quella che sta analizzando, e crea un array per ciascuna riga, colonna, diagonale principale e diagonale inversa (le diagonali attraverso l'ausilio di funzioni ausiliarie). Su ognuno di questi array viene chiama la funzione *scoreLine().*

5) scoreLine(): ha il compito di assegnare un punteggio all'array che viene passato in input. La mossa che stiamo considerando potrebbe creare situazioni diverse nei vari array di cui fa parte; per questo noi abbiamo deciso di sommare i punteggi positivi e negativi che avevamo per valutare la situazione complessiva e quindi la validità della mossa. Ad esempio aggiungiamo 20 o 30 punti nel caso in cui andiamo ad allungare un sottovettore composto da celle marcate dal nostro giocatore, diminuiamo il punteggio se invece è l'avversario ad allungare un suo sottovettore.

   Fatto questo su riga, colonna, diagonale principale e diagonale inversa scoreNotLeaf prende questi quattro valori e sceglie l'alternativa con punteggio maggiore. Quella sarà il valore associato alla mossa.

   La funzione appena descritta, scoreLine, sfrutta tre funzioni. La prima è *maxSubVector*, poi abbiamo *riskySituation* e infine *justOne*.

6) maxSubVetor è una funzione fondamentale per i casi che cerchiamo di analizzare. Infatti, come si può vedere la maggior parte si basano proprio su un controllo del sottovettore massimo.

7) riskySituation e infine justOne sono due funzioni molto simili e che modificano leggermente maxSubVector, infatti non analizzano il sottovettore massimo ma tutti i possibili sottovettori contenuti in una riga e controllano che rispettino certi requisiti. La prima ci permette di controllare se un giocatore ha un sottovettore di lunghezza k-1 con ai lati due celle libere. La seconda ci permette di controllare se un giocatore ha un sottovettore di lunghezza k in cui k-1 celle sono marcate da lui e una cella è marcata dal giocatore opposto.

Nel codice si possono trovare ulteriori commenti esplicativi.

ALGORITMI UTILIZZATI

1) alphabeta pruning
2) ricerca del sottovettore massimo

STRUTTURE DATI UTILIZZATE

1) java.util.Arrays di Java
2) java.util.ArrayList di Java

STRATEGIE ORIGINALI

1) utilizzo del sottovettore massimo per valutare il punteggio e quindi la priorità di una mossa nel caso in cui non siamo ancora in una configurazione finale.
2) valutazione di una mossa analizzando la riga, la colonna e le diagonali di cui fa parte
3) scelta delle casistiche per le analisi citate sopra

COSTI COMPUTAZIONALI (considerando L=max{M, N})

1) justone / riskySituation / maxsubvector = O(L)
2) scoreLine = O(L) per via delle chiamate alle funzioni al punto precedente
3) matrixDiag / diagOpp = O(M*N)
4) scoreNotLeaf = O(M*N) per via della chiamata matrixDiag e diagOpp. Il costo relativo alle chiamate di scoreLine vengono "inglobate" nel costo totale O(M*N)

5) score =O(M*N) per la chiamata a scoreNotLeaf
6) alphabeta ha costo nel caso pessimo m^{depth} (con m numero medio di mosse e depth profonditá di ricerca massima).

   In conclusione quindi il costo del nostro algoritmo sarà O(max(m^{depth} , M*N)