SufferingDoge

Alma Mater Studiorum - Università di Bologna
Algoritmi e Strutture di Dati

Francesco Licchelli #0001041426
Nico Wu #0001028979

A.Y. 2022-2023

Indice

1 Introduzione

Lo scopo del progetto è definire un'implementazione dell'interfaccia `MNKPlayer` in grado di giocare ad un `MNKGame` ed essere capace di rispondere in maniera ottimale alle mosse generate dall'avversario. Per raggiungere tale obiettivo viene usato l'algoritmo *MiniMax* con *AlphaBeta pruning*. Vengono inoltre definiti due algoritmi euristici per ridurre lo spazio di ricerca di *MiniMax* e quindi diminuire notevolmente il tempo necessario per determinare la miglior mossa possibile.
La prima euristica delle due, "intermedia", per ogni nodo di cui stiamo calcolando i possibili stati futuri, scarta i figli meno promettenti, lasciandone solo un numero costante. In tal modo riduciamo notevolmente la quantità di nodi visitati, permettendo così di focalizzarci sui nodi migliori e di raggiungere stadi più avanzati del gioco. La seconda, "finale", valuta complessivamente tutta la board, assegnandole un punteggio che verrà gestito da *AlphaBeta*.

2 Classi utilizzate

Classe	**Descrizione**
`SufferingDoge.Board`	Classe derivata di `mnkgame.MNKBoard`; gestisce le sequenze di entrambi i giocatori nelle quattro direzioni. Memorizza inoltre anche il numero di tali sequenze.
`SufferingDoge.Coord`	Classe utilizzata per iterare tra le celle delle matrici presenti in `Board`
`SufferingDoge.Node`	Sottoclasse di `mnkgame.MNKCell`; viene utilizzata per rappresentare i nodi dell'albero di gioco e per confrontare la bontà dei nodi.
`SufferingDoge.SufferingDoge`	Implementazione di `mnkgame.MNKPlayer`.

3 Sequenze massime

Tra i possibili criteri di valutazione di ogni cella, abbiamo deciso di basarci, per entrambi i giocatori, sulla lunghezza della sequenza valida più estesa che coinvolge la cella stessa.
Tali informazioni vengono salvate in un tensore `Board.seqBoard[2][4][M][N]`, cosi da avere una matrice diversa per ogni direzione e giocatore. Inoltre, dopo ogni mossa, viene aggiornato `Board.seqCount[2][K]`: una matrice in cui nella cella $[i, j]$ è presente il numero di sequenze del giocatore $i \in \{0, 1\}$ di lunghezza $j \in [0..K[$. Ogni volta che viene effettuata una mossa, sia essa simulata o reale, `Board.seqCount` e tutte le matrici di `Board.seqBoard` vengono aggiornate dal metodo `Board.updateSequencesInDir`, basato su due metodi: `Board.calcVectStartToEnd` e `Board.calcVectEndToStart`.

3.1 I metodi `Board.calcVectStartToEnd` e `Board.calcVectEndToStart`

Tali metodi si occupano di determinare la nuova sequenza più lunga possibile per ogni cella entro $K-1$ posizioni in una data direzione.
Il metodo `Board.calcVectStartToEnd` ha i seguenti parametri formali:

- `int[] s`: il vettore a cui assegnare il valore delle sequenze massime considerando l'ultima cella selezionata;
- `int startVect`: indice di partenza della scrittura di dati in `s`;
- `int endVect`: indice di interruzione: corrisponde con la posizione della nuova mossa;
- `Coord startMat`: la coordinata di inizio nella matrice che mappa la posizione di `startVect` sulla matrice delle configurazioni `MNKBoard.B`;
- `Coord dir`: contiene gli indici di incremento, utilizzati per muoversi in `MNKBoard.B`;
- `MNKCellState target`: il proprietario della matrice da visitare.

Viene utilizzata la programmazione dinamica per scrivere nel vettore `s`, negli indici compresi tra `startVect` ed `endVect`, la lunghezza della sequenza massima che coinvolge ogni cella considerata.
Segue un breve esempio in un caso semplificato di `calcVectStartToEnd` in cui viene considerata solamente la direzione orizzontale.
Dato il sottovettore `subV[]` della matrice `MNKBoard.B` contenente la configurazione di una riga in direzione orizzontale, inizializziamo l'iteratore `iter` assegnandovi il valore di `startMat` (per semplificare nel caso orizzontale supponiamo che sia rappresentata da un indice, anche se sarebbe stata una coordinata); sono tre gli scenari possibili:

1. caso base: se `iter` = `startMat`, allora la sequenza massima corrisponde con il numero delle celle il cui `state` è uguale a `target` nel raggio di $K-1$.
2. caso base: se non esiste la $(K-1)$-esima cella oppure se essa è stata già scelta dall'avversario, allora la sequenza massima è quella precedentemente calcolata.
3. caso induttivo: se esiste la $(K-1)$-esima cella, allora la sequenza massima che comprende la cella `iter` sarà il massimo tra la sequenza massima che comprende la cella `iter`-1 (che a sua volta comprenderà anche la cella `iter` considerata) ed il numero di celle comprese tra `iter` e `iter`$+(K-1)$ appartenenti a `target`.

$$\begin{cases} s[i] \leftarrow (\sum_{j=start}^{start+(K-1)} subV[j]) & \textbf{if } \texttt{i} = \texttt{start} \\ s[i] \leftarrow s[i-1] & \textbf{else if } \nexists \texttt{ subV[i+(K-1)] } \textbf{or } \texttt{subV[i+(K-1)]} = \texttt{opp(target)} \\ s[i] \leftarrow max\{s[i-1], (\sum_{j=i}^{i+(K-1)} subV[j])\} & \textbf{otherwise} \end{cases}$$

`Board.CalcVectEndToStart` è lo speculare `Board.CalcVectStartToEnd`.
Dopo aver introdotto i due metodi possiamo spiegare il funzionamento di `Board.updateSequencesInDir`.

3.2 Il metodo `Board.updateSequencesInDir`[1]

`Board.updateSequencesInDir` aggiorna il tensore delle sequenze `Board.seqBoard` rispetto alla direzione e al target considerati e `Board.seqCount`.
Tra i suoi parametri formali vi sono:

- `int[][] seqBoardDir`: matrice in cui nella cella $a_{i,j}$ è presente la lunghezza della massima sequenza di cui fa parte anche $a_{i,j}$;
- `MNKCellState mark`: il giocatore che ha markato la cella;
- `Coord pos`: la posizione della cella markata;
- `Coord dir`: contiene gli indici di incremento, utilizzati per muoversi in `MNKBoard.B`;
- `MNKCellState target`: il "proprietario" della matrice da aggiornare.

Più in dettaglio, markando una cella sono possibili due scenari:

1. `mark` = `target`: se il proprietario della matrice corrisponde con il giocatore che sta facendo la mossa allora bisogna considerare le possibili nuove sequenze che si possono formare nell'intorno centrato nella cella selezionata con raggio $K-1$, per ogni cella compresa nell'intervallo si prende il valore massimo tra il valore preesistente e la lunghezza massima della nuova sequenza che coinvolge la cella appena markata. I risultati aggiornati vengono generati da `Board.calcVectStartToEnd` e `Board.calcVectEndToStart`.

2. `mark` $\neq$ `target`: se il proprietario della matrice è l'avversario del giocatore che effettua la mossa allora sarà necessario considerare la possibilità che la nuova cella marcata blocchi oppure decrementi la lunghezza delle sequenze del proprietario della matrice; sono considerate sequenze non valide tutte quelle sequenze che non possono raggiungere una lunghezza di almeno K celle, poiché limitate dalla scacchiera oppure da mosse dell'avversario. In `Board.seqBoard` le sequenze non valide sono identificate con il valore `-1`. Nel caso in cui non si formi alcuna sequenza non valida sarà necessario ricalcolare le sequenze senza considerare tutte le celle che sono state "tagliate via" in seguito alla selezione.

In entrambi i casi, ogni modifica a `Board.seqBoard` deve essere riflessa anche in `Board.seqCount`. Per mantenere una coerenza tra la Board di gioco e `Board.seqBoard` sono stati implementati i metodi `Board.saveState(MNKCell)` e `Board.restoreState(MNKCell)`, che si occupano rispettivamente di salvare lo stato delle sequenze prima che la cella in questione venisse markata e di ripristinare `Board.seqBoard` e `Board.seqCount` ad uno stato precedente.
L'algoritmo proposto nel caso pessimo ha costo $O(2(K-1)^2) = O(K^2)$.

4 Generazione dei figli

Per rendere più efficiente *AlphaBeta* è stato deciso di considerare esclusivamente le celle libere adiacenti a quelle occupate. Per ognuna di esse verrà inizializzato un nodo, il quale verrà valutato e successivamente inserito in una coda con priorità.
Gli attributi di `Node` utilizzati per la comparazione sono:

- `isImportant`: viene impostato a `true` se e solo se il nodo considerato fa parte di una sequenza di $K-1$ celle, di una sequenza di $K-2$ con estremità non bloccate o di una sequenza lunga $K-2$ in più di una direzione;
- La **più lunga sequenza** che passa per il nodo tra le diverse direzioni;
- La **somma** della lunghezza delle sequenze che passano per il nodo nelle diverse direzioni;
- Il giocatore che effettuerà la mossa nel **turno successivo**.

Dati due nodi sono possibili i seguenti scenari, in base ai quali, varia l'ordine delle priorità:

- nel caso in cui i due nodi siano entrambi importanti:
  1. la lunghezza massima maggiore;
  2. giocatore del turno successivo;
  3. il nodo la cui somma delle lunghezze delle sequenze che vi passano è maggiore.
- nel caso in cui nessuno dei due nodi è importante:
  1. il nodo la cui somma delle lunghezze delle sequenze che vi passano è maggiore;
  2. la lunghezza massima maggiore;
  3. giocatore del turno successivo.
- nel caso in cui un solo nodo è importante:

  viene preferito quello in cui `Node.isImportant=true`.

4.1 Chiarimenti sulle priorità diverse

Per l'ordinamento dei nodi sono stati utilizzati due ordini di priorità diversi per valutarne la bontà nelle diverse situazioni, critiche o meno:

1. le situazioni critiche si verificano quando mancano al più quattro mosse per raggiungere uno stato finale (vittoria, sconfitta, pareggio), dunque per facilitare la potatura dei rami diamo più importanza agli attributi che indicano la lunghezza, il giocatore successivo e il proprietario della sequenza, in quanto determinano direttamente il numero esatto di mosse mancanti per terminare la partita, mentre la somma delle lunghezze delle sequenze che passano per il nodo attuale ha minor peso in quanto non direttamente relazionata con il numero minimo delle mosse mancanti per chiudere il gioco;

2. le situazioni non critiche si verificano quando mancano almeno 4 mosse per giungere ad una configurazione finale, quindi si prediligono le mosse che portino alle situazioni critiche assumendo che entrambi giocatori giochino in maniera ottimale. Pertanto la somma delle lunghezze delle sequenze che coinvolgono la cella presa in considerazione ha un peso maggiore, poiché permette di prevedere le mosse ottimali.

4.2 Selezione dei figli

Dopo aver generato tutti i possibili figli, essi vengono inseriti in una coda con priorità dalla quale poi ne verrà estratto un numero variabile: più ci si allontana dalla radice, meno saranno i figli. In questo modo, quando ci si trova vicini alla radice verranno valutati molti scenari diversi, mentre quando si è scesi abbastanza in profondità verranno visitate solamente le mosse migliori. È stato determinato empiricamente che, scendendo per un massimo di nove livelli, il numero di figli ottimale e $\texttt{numFigli} = \max(1 + \texttt{livelliRimanenti}, 5)$.
Valutare prima i nodi "migliori" rende più probabile le potature, abbattendo il tempo di esecuzione dell'algoritmo.

5 Euristica finale[2]

Anche avendo adottato l'euristica appena descritta, non è sempre possibile visitare uno stato finale di gioco, pertanto è stata implementata un'altra euristica in grado di valutare tutta la Board. Essa è stata progettata in maniera tale da non incidere in maniera negativa sulle prestazioni.
La board viene valutata sommando il numero di sequenze proprie e sottraendo il numero di sequenze dell'avversario, dando importanza maggiore alle sequenze più lunghe. In questo modo la valutazione è simmetrica e poiché tale calcolo viene fatto sulla base di `Board.seqCount` la generazione dello score ha costo $O(2K) = O(K)$.

6 Gestione del tempo

Quando viene invocato il metodo `MNKPlayer.selectCell` viene assegnato ad una variabile il tempo di inizio della scelta della mossa. Durante l'esecuzione di `SufferingDoge.alphabeta` viene calcolato il tempo passato dall'inizio del turno: se questo valore è troppo vicino allo scadere del tempo, allora il nodo corrente valutato da *AlphaBeta* viene considerato come foglia, quindi il suo score dipenderà dall'euristica finale. Tale scelta non impatta troppo negativamente sulle prestazioni di AlphaBeta poiché sviluppando prima le mosse migliori, lasceremo per ultime quelle meno interessanti.

7 Conclusioni

L'attuale implementazione dell'algoritmo *MiniMax* con *potatura AlphaBeta* permette di dare risposte soddisfacenti in tempi ragionevoli, con un costo computazionale nel caso pessimo $O(\texttt{DEPTH!})$ e nel caso ottimo $O(\sqrt{\texttt{DEPTH!}})$, con `DEPTH` il numero livelli visitati da *AlphaBeta*.
Tuttavia, si potrebbero migliorare ancora le prestazioni di *SufferingDoge* gestendo in maniera più precisa il tempo attraverso l'utilizzo di *Iterative Deepening* ed evitando di visitare configurazioni già viste tramite l'impiego di una tabella di trasposizioni.

8 Algoritmi implementati

Segue lo pseudocodice di alcuni degli algoritmi discussi.

Algorithm 1 UpdateSequencesInDir

```
int foward, backward
Coord inizio, fine
boolean bwk, fwk
Contare e iterare le celle valide (al più K − 1) in un verso finché non se ne incontra una dell'avversario
Salvare il valore del contatore in backward e la coordinata di terminazione in inizio
Se è possibile iterare anche la K-esima cella, impostare bwk a true
Contare e iterare le celle valide (al più K − 1) nell'altro verso finché non se ne incontra una dell'avversario
Salvare il valore del contatore in forward e la coordinata di terminazione in fine
Se è possibile iterare anche la K-esima cella, impostare fwk a true
if mark = target and forward + backward + 1 ≥ K then
    size ← forward + backward + 1
    s ← new int[size]
    calcVectStartToEnd(s,0,backward-1, inizio, dir, target),
    calcVectEndToStart(s,backward+1,size-1, fine, dir, target)
    s[backward] ← max(s[backward − 1], s[backward + 1])
    for i ← 0..size − 1 do
        B[i] ← max(B[i], s[i])
    end for
else
    size ← forward + backward + 1, s ← new int[size]
    s[backward] ← −1
    if !bwk then
        for i ← 0..backward − 1 do
            s[i] ← −1
        end for
    else
        calcVectStartToEnd(s,0,backward-1, inizio, dir, target)
    end if
    if !fwk then
        for i ← size − 1..backward + 1 do
            s[i] ← −1
        end for
    else
        calcVectEndToStart(s,backward+1,size-1, fine, dir, target)
    end if
    for i ← 0..size and isValid(inizio) do
        if seqBoardDir[inizio] > 0 and (B[inizio]=FREE or inizio.equals(pos)) then
            seqCount[mark=SufferingDoge.me?1:0][seqBoardDir[inizio]-1]−
        end if
        if s[i] > 0 and (B[inizio]=FREE or inizio.equals(pos)) then
            seqCount[mark=SufferingDoge.me?1:0][s[i]-1]−
        end if
        inizio.add(dir)
    end for
end if
```

Algorithm 2 getScore

if gameState() = MNKGameState.$DRAW$ **then return** 0
end if
if gameState() = SufferingDoge.$myWin$ **then return** ∞
end if
if gameState() = SufferingDoge.$oppWin$ **then return** $-\infty$
end if
$score \leftarrow 0$
$exp \leftarrow 1$
for $i \leftarrow 0..K-3$ **do**
    $score \leftarrow score + 5^{exp} * (seqCount[0][i] - seqCount[1][i])$
    $exp \leftarrow exp + 2$
end for
for $i \leftarrow K-3..K$ **do**
    $score \leftarrow score + 10^{exp} * (seqCount[0][i] - seqCount[1][i])$
    $exp \leftarrow exp + 2$
end for
return score