Corrado il Conquistatore

Relazione del Progetto MNKgame 2021-2022

Componenti

Alessandro Ravveduto 0001020539 Saverio Govoni 0001029472

Indice

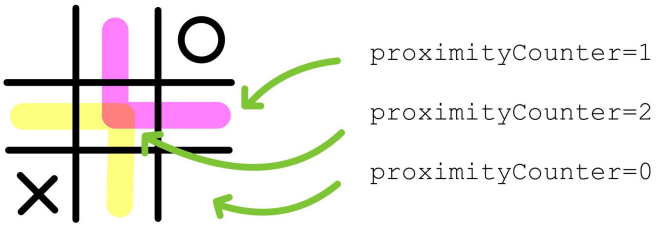
Introduzione

Il fine del progetto è creare un player virtuale in grado di giocare, nel modo migliore possibile, ad un MNK-game ovvero una generalizzazione del classico tris.
La classe principale è PlayerNostro, dove risiede il centro del nostro giocatore. In particolare troviamo la funzione `selectCell()` che permette di scegliere quale mossa compiere. Quest'ultima utilizza il noto algoritmo iterative Deepening che per semplicità abbiamo indicato con la funzione `iterDeep()`. Inoltre, in questa classe troviamo la funzione `evaluateProMax()`, fondamentale per valutare ogni stato di gioco. Per migliorare le prestazioni abbiamo aggiunto le classi: AVLtree che permette di ordinare le mosse, Node su cui si basa la struttura dell'albero, BooleanCombination e BooleanBoard che estendono la MNKCell e MNKBoard, rispettivamente, con variabili essenziali per la valutazione.

BooleanCombination e BooleanBoard

La classe BooleanCombination è un'estensione della classe MNKCell in cui abbiamo aggiunto le variabili booleane `Vertical`, `Orizontal`, `Diagonal`, `Antidiagonal` e la variabile intera `proximityCounter`.

Le variabili di tipo booleano sono necessarie ai fini della valutazione in quanto ci permettono di verificare se una data cella sia stata già controllata per un allineamento di tipo verticale, orizzontale, diagonale e antidiagonale. In tal modo possiamo risparmiare tempo evitando di controllare lo stesso allineamento iniziato da celle differenti.
La variabile `proximityCounter` indica il numero di celle segnate nell'intorno di quella cella, questa verrà utilizzata come chiave nel AVLtree.



La classe BooleanBoard è un'estensione della classe MNKBoard in cui utilizziamo una matrice del tipo BooleanCombination. Inoltre, in questa classe sono presenti le funzioni `ChangeVertical()`, … e `GetVertical()`, … che permettono di cambiare il booleano di una determinata cella secondo un parametro dato e di restituire tale parametro. Inoltre è presente la funzione addProximity che non fa altro che modificare il proximitycounter delle celle attorno alla cella data e ritornare una lista di queste con in testa la cella data.

Costi computazionali

`ChangeVertical - … - ChangeAntidiagonal`: $O(1)$
`GetVertical - … - GetAntidiagonal`: $O(1)$
`addProximity`: $O(1)$

AlphaBeta e Iterative Deepening

L'idea alla base di Corrado il Conquistatore è utilizzare il MiniMax, un algoritmo ricorsivo per individuare la migliore mossa possibile in un gioco secondo il criterio di minimizzare la massima perdita possibile. Siccome non è sempre possibile visitare l'intero Game Tree utilizziamo l'algoritmo ottimizzato AlphaBeta. Il concetto dietro a quest'ultimo è di minimizzare il numero di nodi valutati da MiniMax interrompendo la visita su un sottoalbero quando siamo sicuri di non poter trovare una soluzione migliore di quella attuale. Per implementare tutto ciò è necessario introdurre α e β, rispettivamente il punteggio minimo ottenibile dal giocatore che massimizza e il punteggio massimo ottenibile dal giocatore che minimizza, quando $\alpha \geq \beta$ si effettua una potatura del relativo sottoalbero.

AlphaBeta ha uno speed-up quadratico rispetto a MiniMax se riusciamo a trovare un ordine ottimo di valutazione delle mosse. A parità di calcoli, con AlphaBeta possiamo visitare nell'albero una profondità doppia rispetto a MiniMax.

Il problema di impiegare solo AlphaBeta è legato al tempo in quanto un requisito che il giocatore deve rispettare è di scegliere una mossa entro il timeout. Per risolvere questa complicazione utilizziamo l'algoritmo Iterative Deepening, il quale implementa una ricerca in ampiezza a profondità via via crescente, allo scadere del tempo se l'analisi della profondità attuale non è terminata restituisce la mossa trovata alla profondità precedente.

Costi computazionali

AlphaBeta e Iterative Deepening:

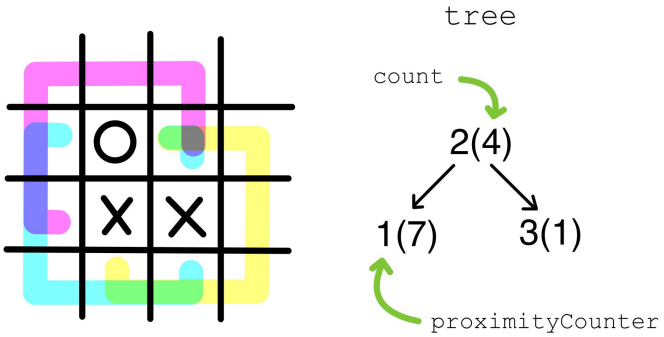
Caso pessimo: $O\left(b^d\right)$ con b fattore di diramazione e d altezza dell'albero.

Caso ottimo: $O\left(\sqrt{b^d}\right)$

In termini di memoria Iterative Deepening ha un costo lineare rispetto alla profondità di ricerca.

Albero AVL

Nella classe AVLtree implementiamo un albero binario bilanciato con nodi duplicati. L'albero è del tipo della classe Node, che oltre a contenere le variabili ordinarie per un albero AVL contiene anche `count`, indica il numero di nodi con la stessa chiave, e una lista `ListNodi` di celle del tipo BooleanCombination. L'albero utilizza come chiave la variabile `proximityCounter` in modo tale privilegiare come mosse le celle libere con intorno di celle segnate maggiore.



In questa classe sono presenti le funzioni fondamentali per inserire ed eliminare un nodo nell'albero, le quattro funzioni per ruotare i sottoalberi sbilanciati. Inoltre è presente la funzione `PossibleMoves()` che ritorna una lista ordinata di mosse. In particolare `PossibleMoves()` chiama la funzione `NotInOrder()`, la quale visita in ordine l'albero partendo però dal nodo maggiore e, invece che visitare il nodo, va ad aggiungere tutte le celle di `ListNodi` in una lista che verrà poi memorizzata in una variabile locale di `PossibleMoves()`.
Impossibile non parlare della funzione `updateTree()` che data una lista di tipo BooleanCombination permette di aggiornare l'albero andando ad eliminare i nodi vecchi ed inserendo i nodi con `proximityCounter` aggiornato. Per inserire ed eliminare i nodi da `tree`, l'albero su cui andiamo effettivamente a lavorare, utilizziamo le funzioni `InsertTree()` e `DeleteTree()`, le quali vanno a richiamare rispettivamente `insertNode()` e `deleteNode()` che prendono in input la radice, la chiave e la cella. Queste ultime due funzioni differiscono dal classico insert e delete dell'albero AVL in quanto inserendo o eliminando dei nodi va a modificare il `count`, e aggiungere o eliminare la cella in input all'interno di `ListNodi`.

Costi computazionali

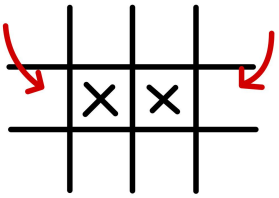
`rightRotate - leftRotate`: $O(1)$
`insertNode - deleteNode`: $O(log\ n)$ dove n è il numero di nodi presenti nell'albero
`updateTree`: $O(c\ log\ n)$ dove c è la cardinalità di `ListCell` , ovvero della lista di celle il cui proximity counter deve essere aggiornato

Valutazione

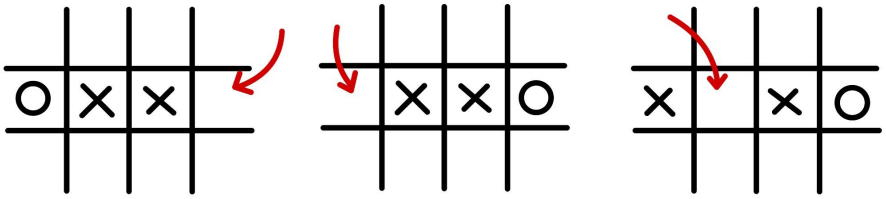
Prima di andare nel dettaglio sulla valutazione dobbiamo introdurre alcune definizioni.
Un **threat** (minaccia) è un allineamento di celle segnate dallo stesso giocatore.

Ci sono tre tipi di minacce:

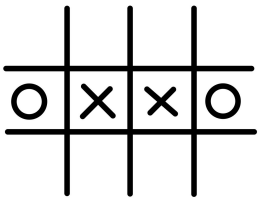
- **Open**: entrambe le celle adiacenti all'estremità dell'allineamento sono libere.



- **Half-Open**: solo una delle due celle adiacenti all'estremità dell'allineamento è libera, l'altra è segnata dall'avversario oppure è presente un salto, ovvero una cella libera tra le celle segnate dallo stesso giocatore.



- **Closed**: entrambe le celle adiacenti all'estremità dell'allineamento sono segnate dall'avversario.



L'idea della valutazione di una configurazione di gioco è molto semplice: $f = A - B$
dove A è il valore assegnato alla valutazione della board rispetto al nostro giocatore e B rispetto all'avversario. In particolare definiamo A e B come segue:

$$A = \begin{cases} \sum_{i=1}^{k-3} (a_{2i-1}p_{i,1} + a_{2i}p_{i,2}) + a_{2(k-2)-1}p_{k-2,1} + 100p_{k-2,2} + 80p_{k-1,1} + 250p_{k-1,2} + 1000000p_k & \text{if } k > 3 \\ a_1p_{k-2,1} + 100p_{k-2,2} + 80p_{k-1,1} + 250p_{k-1,2} + 1000000p_k & \text{if } k = 3 \end{cases}$$

$$B = \begin{cases} \sum_{i=1}^{k-3} (a_{2i-1}q_{i,1} + a_{2i}q_{i,2}) + a_{2(k-2)-1}q_{k-2,1} + 1300q_{k-2,2} + 2000q_{k-1,1} + 5020q_{k-1,2} + 1000000q_k & \text{if } k > 3 \\ a_1q_{k-2,1} + 1300q_{k-2,2} + 2000q_{k-1,1} + 5020q_{k-1,2} + 1000000q_k & \text{if } k = 3 \end{cases}$$

dove:

- $p_{i,1}$ indica il numero di allineamenti di tipo half-open di lunghezza i.
- $p_{i,2}$ indica il numero di allineamenti di tipo open di lunghezza i.
- p_i numero di allineamenti senza interruzioni di lunghezza i.

Il parametro p si riferisce al nostro giocatore mentre q, definito in modo analogo a p, si riferisce all'avversario.
Il calcolo della configurazione di gioco f viene effettuato dalla funzione `evaluateProMax()` in cui il calcolo di A e B viene affidato alla funzione `threats()`. Per il calcolo di p e q utilizziamo la funzione `getThreat()`, la quale somma i risultati ottenuti dalle funzioni

`CheckThreatVertical()`,`CheckThreatOrizontal()`, `CheckThreatDiagonal()` e `CheckThreatAntidiagonal()` per ogni cella.
Queste quattro funzioni lavorano allo stesso modo, pertanto ci soffermeremo ad analizzare solo la prima. Presi in input una cella, la lunghezza della minaccia e il tipo di allineamento si contano le celle sopra e sotto la cella data attraverso due cicli `while` consecutivi. Le celle visitate vengono marchiate utilizzando i parametri aggiunti dalla classe BooleanCombination e si memorizzano le celle per cui si sono interrotti i cicli, rispettivamente `lastCellx1` e `lastCellx2`. Se il conteggio della lunghezza dell'allineamento corrisponde al parametro dato in input, si effettuerà il controllo su `lastCellx1` e `lastCellx2` rispetto al tipo di allineamento, se anche questo controllo viene passato con successo allora ritorneremo 1, c'è un allineamento verticale che passa per la colonna della cella data.
Le funzioni CheckThreat vengono chiamate su ogni cella marcata dallo stesso giocatore e poiché marchiamo le celle visitate non facciamo operazioni superflue, minimizzando così il costo computazionale della valutazione. Le marcature vengono poi fissate a `false` come ultime operazioni della funzioni `getThreat`.

Costi computazionali

Il costo di `CheckThreatVertical` sarebbe $O(M)$, dove M è il numero di righe, ma siccome non è possibile che sulla board ci siano più di $K + 1$ celle segnate dallo stesso player in una stessa colonna, poiché altrimenti la partita terminerebbe, il costo effettivo di `CheckThreatVertical` è $O(K)$. Analogo ragionamento per le altre funzioni CheckThreat.
`getThreat`: $O(nK)$ dove n è il numero di celle segnate nella board
`threat - evaluateProMax`: $O(nK^2)$ siccome viene chiamata `getThreat` $K - 2$ volte

Gestione del tempo

Per non sforare il time out di 10 secondi sfruttiamo tre variabili e una funzione.
La variabile start di tipo long si aggiorna all'inizio di ogni `selectCell` e segna il momento di inizio del countdown. La variabile `percTime` di tipo double indica quale percentuale dei dieci secondi viene utilizzata, (le varie funzioni devono avere il tempo di concludersi perciò è necessario lasciare uno scarto). Infine la variabile `is_late` di tipo booleano diventa vera quando il tempo sta per finire.
Per prima cosa `percTime` viene inizializzata tramite la seguente espressione: 90.0 - (M*N/1000) in questo modo per le griglie più grandi verrà riservato più tempo per eseguire tutti i return, inoltre `percTime` viene aggiornata alla fine di ogni `selectCell` tramite la funzione `changeTimer`. In particolare, quest'ultima funzione controlla quanto tempo è rimasto per raggiungere i 10 secondi, se è meno di 300 millisecondi allora diminuisce `percTime` di 3, se è più di 600 millisecondi allora aumenta `percTime` di 3, assicurandosi però che `percTime` non superi mai 99. In questo modo se la scelta della cella è andata troppo vicina allo scadere del

tempo diminuirà la percentuale del timer sfruttata così da riservare uno scarto maggiore, se invece avanza una buona porzione di timer aumenta la percentuale in modo da poter sfruttare più tempo di ricerca per la prossima `selectCell`.
I vari controlli del timer vengono effettuati all'inizio delle tre funzioni `selectCell`, `IterDeep` e `AlphaBetaDepht` tramite un if che sottrae il tempo corrente con la variabile start e lo compara al timeout moltiplicato per `percTime`, nel caso in cui il tempo utilizzato superi la percentuale di timer sfruttabile la variabile `is_late` verrà settata su true, e verrà eseguito un break per terminare i cicli.
Una volta ritornato il controllo alla funzione `selectCell` se la variabile `is_late` è vera il ciclo verrà interrotto senza assegnare la cella trovata, in questo modo se una profondità non è stata visitata completamente la mossa migliore alla profondità precedente verrà assegnata.

Bibliografia

Minacce per la valutazione: http://www.cari-info.org/Actes-2018/p276-286.pdf
Albero AVL con nodi duplicati: https://www.geeksforgeeks.org/avl-with-duplicate-keys/