Relazione progetto *M-N-K game* 2021/2022

A questo progetto hanno partecipato gli studenti:

Girotti Lorenzo 0001020884

Ruggiero Simone 0001021768

Problema affrontato

Il problema che il nostro algoritmo va a risolvere è quello della decisione della cella da marcare per portare al risultato migliore possibile durante la partita del gioco di tipo *M-N-K*.

Documentazione utilizzata

Per realizzare questo progetto ci siamo basati su una documentazione specifica:

"Developing a Memory Efficient Algorithm for Playing m, n, k Games" di Nathaniel Hayes e Teig Loge

http://www.micsymposium.org/mics2016/Papers/MICS_2016_paper_28.pdf

Questa documentazione spiega le basi dietro un algoritmo pensato per non perdere mai. E' un algoritmo greedy che assegna un valore specifico ad ogni cella libera chiamato **helpfulness**, tramite il quale si determina, inserendo le celle con i rispettivi valori all'interno di una coda con priorità, la cella più utile.

Questo valore è dato dalla somma di:

- numero di possibili raggruppamenti di K celle che comprendono quella cella(non sono considerati i raggruppamenti che racchiudono una o più celle dell'avversario);
- numero di celle già marcate comprese in questi raggruppamenti.

Questo valore determina il livello di utilità di quella cella per il giocatore.

Per avere un calcolo più accurato dell'helpfulness, nella documentazione viene consigliato di calcolare il valore di importanza di una cella, sia dal lato del nostro giocatore, sia dal lato

del giocatore avversario. Sommando questi due risultati si ottiene un valore il più reale possibile, così da considerarne l'utilità per poter bloccare delle possibili mosse vincenti da parte dell'avversario.

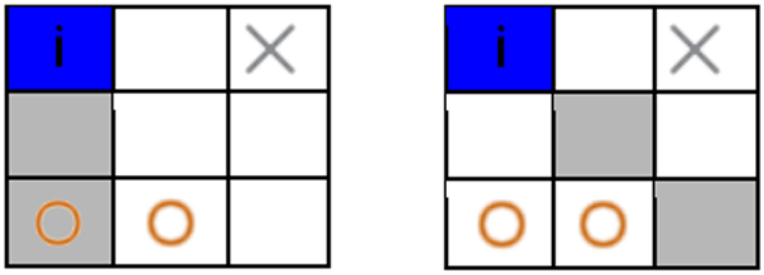
La cella che alla fine avrà il valore maggiore sarà quella che, al momento, costituisce la mossa più redditizia e quindi quella da scegliere durante il gioco.

Implementazione

Per la realizzazione di questo algoritmo abbiamo deciso di salvare il valore di helpfulness di ogni cella all'interno di una coda con priorità massima, in modo tale da avere sempre a portata di mano, in cima alla coda, la cella con il valore dell'helpfulness maggiore.

Per ogni cella i (sia libera che già marcata) controlliamo:

- Raggruppamento orizzontale verso destra di k celle partendo dalla cella i
- Raggruppamento verticale verso il basso di k celle partendo dalla cella i
- Raggruppamento della diagonale principale verso il basso di k celle partendo dalla cella i
- Raggruppamento della diagonale secondaria verso il basso di k celle partendo dalla cella i



Questo lavoro è svolto dalle funzioni:

- `horizontalCheck`
- `verticalCheck`
- `mainDiagonalCheck`

- `secondDiagonalCheck`

Per ognuno di questi raggruppamenti calcoliamo il valore dell'helpfulness utilizzando la funzione `getFreeCellsHelpfulness` e lo aggiungiamo, non solo alla cella che stiamo considerando, ma a tutte le celle libere all'interno del raggruppamento.

In questo modo possiamo considerare solo la metà dei raggruppamenti nei quali una cella può essere compresa, ma avere comunque lo stesso valore reale.

In accordo con la documentazione, quando viene controllata una cella che, se selezionata porta alla vittoria del giocatore, l'algoritmo provvede ad assegnargli un valore pseudo-infinitesimale.

Mentre se porta alla vittoria dell'avversario le viene assegnato un valore ragionevolmente più basso ma che comunque non sia raggiungibile dal valore di altre celle.

Ulteriori modifiche

La prima modifica originale apportata all'algoritmo è stata l'utilizzo di un oggetto di tipo `Cell` all'interno della coda con priorità massima.

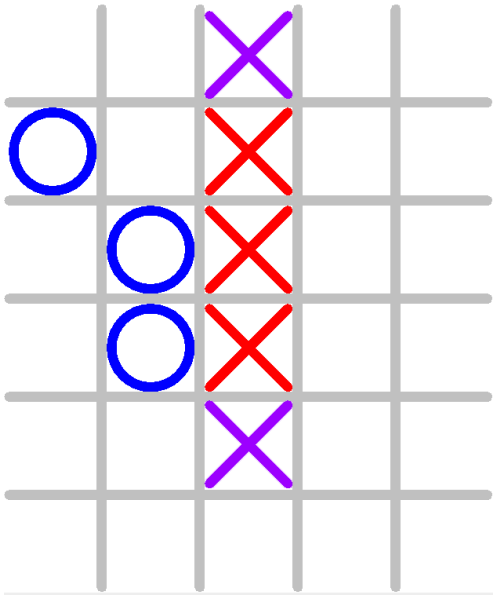
Questo ci ha permesso di salvare le seguenti proprietà relative ad ogni cella:

- `Count` (chiave): indica il valore di helpfulness della cella
- `playerCells`: indica il numero di celle marcate del giocatore in tutti i raggruppamenti in cui è compresa la cella
- `opponentCells`: indica il numero di celle marcate dell'avversario in tutti i raggruppamenti in cui è compresa la cella
- `Cell`: indica la cella

La cella scelta dall'algoritmo è quella con il valore di count più elevato, ma quando ce n'è più di una i parametri `PlayerCells` e `OpponentCells` vengono utilizzati per decidiere quali tra queste è la migliore.

Doppio gioco

> Il doppio gioco è una situazione in cui sono presenti più celle che portano un giocatore alla sconfitta e di conseguenza è impossibile impedire la vittoria del giocatore che lo mette in atto.



Nell'esempio di doppio gioco qui sopra bisogna allineare 4 simboli uguali, quindi il giocatore `X` ha la possibilità di vincere scegliendo una delle due celle viola.

Un'altra scelta implementativa aggiuntiva rispetto all'algoritmo spiegato nella documentazione, è quella di andare a prevenire il più possibile situazioni di "doppio gioco" favorevoli all'avversario e di individuare tutte quelle che, invece, sono favorevoli al giocatore.

Per ogni raggruppamento che contiene almeno $K - 2$ celle marcate appartenenti allo stesso giocatore viene controllata la possibilità di eseguire un "doppio gioco" utilizzando le celle libere del raggruppamento.

Per farlo viene controllata sia la cella precedente alla prima del raggruppamento, sia quella successiva all'ultima. In base al caso viene definito se c'è un pericolo di "doppio gioco" oppure no e nel caso ci sia, si incrementa il valore di helpfulness di tutte le celle del raggruppamento, sommando un valore predefinito.

Costo dell'algoritmo

Il costo computazionale della funzione `selectCell` è interamente dato dal costo della funzione `calculateHelpfulness`.

Calcolo del costo computazionale

Sia f = numero free cells

La funzione `isWinningCell` ha costo:

$$O(K)$$

La funzione `getFreeCellsHelpfulness` ha costo:

$$O(f + 2 * log(f) + K) =$$

$$O(f + K)$$

Le funzioni `horizontalCheck`, `verticalCheck`, `mainDiagonalCheck` e `secondDiagonalCheck` le consideriamo tutte con lo stesso costo, ovvero :

$$O(K * (M * N + f) + f + M * N + f + f + K * (f + K)) =$$

$$O(K * M * N + K * f + f + M * N + K * (f + K)) =$$

$$O(K * M * N + K * (f + K))$$

$$O(K * M * N + K^2)$$

essendo che M è nel peggiore dei casi grande quanto il più piccolo tra M ed N allora si può semplificare:

$$O(K * M * N)$$

quindi il costo generale dell'algoritmo è

$$O(\sum_{i=1}^{f} f * \log i) + O(2 * M * N * (4(K * M * N))) + O(\sum_{i=1}^{f} f * i) =$$

$$O(\sum_{i=1}^{f} f * i) + O(M * N * K * M * N) =$$

$$O(\sum_{i=1}^{f} f * i) + O(M^2 * N^2 * K) =$$

$$O(f * \frac{f * (f+1)}{2}) + O(M^2 * N^2 * K) =$$

$$O(f^3 + M^2 * N^2 * K)$$

dato che nel peggiore dei casi $f^3 = (M * N)^3$, non si può semplificare ulteriormente.