NisshokuPlayer

Relazione del progetto MNKgame

cs.unibo - Anno: 2021 / 2022

Autori: Francesco Testa - 0001029406

Pietro Sami - 0001020804

Indice

Panoramica generale

Il progetto consiste nel creare un algoritmo in java che, in modo intelligente, riesca a giocare ad un **MNKgame**, generalizzazione del celebre TicTacToe.
NisshokuPlayer è la classe principale che contiene la funzione per la scelta (public MNKCell selectcell()). Nisshoku player si basa sull'algoritmo studiato a lezione dell "'**Iterative deepening**", in aggiunta abbiamo implementato: una classe per la valutazione delle situazioni di gioco (**Heuristic**), una classe per la scelta delle celle durante iterative deepening e alphabeta pruning (**HashMapNFC**) ed infine una classe per gestire le configurazioni che abbiamo già valutato nel corso dell'algoritmo (**TranspositionTable**).

AlphaBeta pruning

L'algoritmo si basa sul **MINIMAX** ottimizzato con **AlphaBeta** pruning per minimizzare il numero di nodi visitati.
Si utilizzano due valori, **alpha** e **beta**, dove alpha è il punteggio minimo ottenibile dal giocatore che massimizza, beta il punteggio massimo ottenibile dal giocatore che minimizza. La potatura dell'albero avviene nel caso beta ≤ alpha, in quanto i valori generabili dal sottoalbero ignorato non coincideranno sicuramente con la soluzione ottima. Il costo del minimax con alpha beta pruning è:

- Caso pessimo: $O(b^d)$, con *b* fattore di diramazione e *d* altezza dell'albero.
- Caso medio: $O(b^{\frac{d}{2}})$ con *b* fattore di diramazione e *d* altezza dell'albero.

Iterative Deepening

Il tempo per scegliere la mossa è ovviamente limitato. Dunque per interrompere l'algoritmo entro un limite di tempo si utilizza l'iterative deepening. In questo modo si ricorre ad una **visita in ampiezza** rispetto ad una visita in profondità dell'a-b pruning. Così facendo, se la ricerca ad altezza d non viene ultimata per limiti di tempo, si sceglierà la mossa migliore ad altezza d-1.
Il funzionamento è molto semplice: un ciclo for da 1 a max_depth in cui viene chiamato ogni volta l'a-b pruning.
Il costo di tale algoritmo è:

caso pessimo: $O(b^d)$, con *b* fattore di diramazione e *d* altezza dell'albero.

Nella versione utilizzata in NisshokuPlayer sono stati aggiunti:

- **Cutoff** e **ritorno immediato** nei casi in cui il player trova un sottoalbero che lo porta matematicamente alla vittoria.
- Ad altezza 1 la scelta della cella viene aggiornata durante l'iterazione.
- Se ad altezza 1 non si sono trovate soluzioni in cui evitare o rallentare la sconfitta, si ritorna subito
- Se l'algoritmo ad altezza *d* rileva la sconfitta per ogni cella, si preferisce non aggiornare la casella e mantenere la scelta fatta con altezza *d-1*. Questo affinché venga rallentata la sconfitta.

TranspositionTable

Per evitare che l'algoritmo analizzasse inutilmente determinate situazioni di gioco già visitate in precedenza (che infatti sarebbero valutate sempre nello stesso modo) è stata implementata la classe **Transposition Table**.
L'idea è quella di:

1) creare una struttura dati dizionario che contenga le valutazioni di configurazioni già analizzate.
2) associare univocamente a una determinata situazione un valore numerico e utilizzarla come Key per la struttura dati del punto 1.

Per il primo problema **l'hashtable** si è rivelata la struttura dati ideale dato che ha costo medio = O(1) per le operazioni di Search, Insert e Delete; utilizzate per le funzioni che regolano il popolamento della tabella: **containsData e storeData**.
Ogni Key è associata ad un'istanza della classe **DataHash** che contiene i campi: **depth** (a che profondità dell'albero è stata valutata la configurazione)**, evaluation** (valutazione della configurazione) infine **flag** (Exact, Upperbound, Lowerbound).
La memorizzazione dei dati nella TraspositionTable è affidata allo **storeData(),** a cui viene passato come alpha **l'AlphaOrigin**, ovvero il valore iniziale di alpha in una chiamata di AlphaBeta(). Servirà successivamente per distinguere la **flag**.
In particolare questa viene così assegnata:

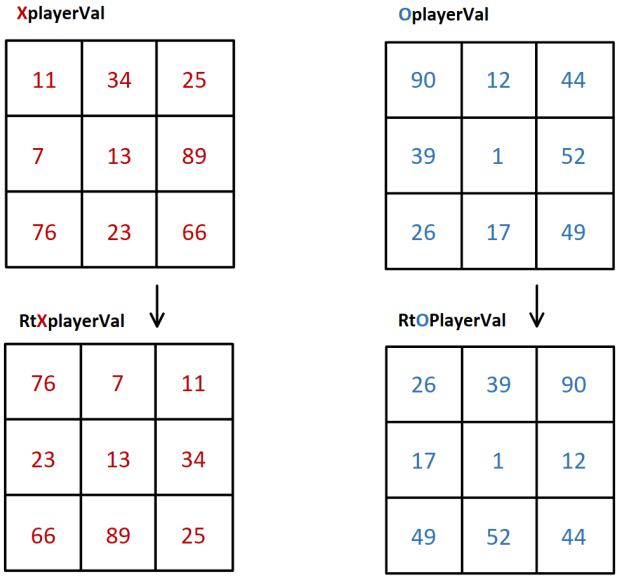
- **Exact**: se la valutazione ottenuta è maggiore di alphaOrigin e minore di beta
- **UpperBound**: se la valutazione ottenuta è minore di alphaOrigin. Poiché alpha rappresenta il valore minimo garantito per il giocatore che massimizza, i valori sottostanti non sono interessanti. Dunque alpha è un upperbound per le valutazioni inferiori.
- **LowerBound**: se la valutazione ottenuta è maggiore di beta. Si usa un ragionamento speculare a quello dell'upperbound.

Ad inizio chiamata di AlphaBeta() si analizza se la situazione di gioco è già stata incontrata. In caso positivo si controlla in base all'altezza se può essere interessante. Successivamente possono avvenire tre azioni in base al **flag**:

- **Exact**: il valore memorizzato nel dato è contenuto nel range alpha e beta, dunque lo ritorniamo direttamente
- **UpperBound**: si aggiorna beta al minimo tra beta e la valutazione presente nel dato. Poiché il valore esatto della configurazione sarà sicuramente minore o uguale ad alphaOrigin, posso restringere il range alpha beta.
- **LowerBound**: si aggiorna alpha al massimo tra alpha e la valutazione presente nel dato. Si usa un ragionamento speculare a quello dell'upperbound.

Zobrist

Per risolvere il secondo punto punto si è partiti dalla funzione di **Zobrist()**: vengono generati M*N *long* randomicamente e salvati all'interno di **due array bidimensionali: XplayerVal e OplayerVal.** (uno per le celle che saranno marcate con X e una per le O). Se M == N viene creata anche una matrice aggiuntiva ottenuta ruotando di 90° a destra quella precedente (**Rt_XplayerVal, Rt_OplayerVal**). Ogni cella della board avrà quindi associata univocamente due valori numerici che permetteranno, attraverso una serie di operazioni, di rappresentare una configurazione di gioco attraverso un'unica codifica.



In tutto vengono create quattro matrici: quelle di partenza XplayerVal e OplayerVal sono riempite con numeri casuali in un range da 1 a 2^{63} (nell'esempio utilizziamo numeri da 1 a 100 per comodità). La probabilità che vengano generati due numeri identici per caselle differenti è molto bassa ($1/2^{63}$) e non influirà sulle prestazioni dell'algoritmo.

im 1.0

All'interno della TranspositionTable sono presenti 8 parametri legati alla memorizzazione delle codifiche:

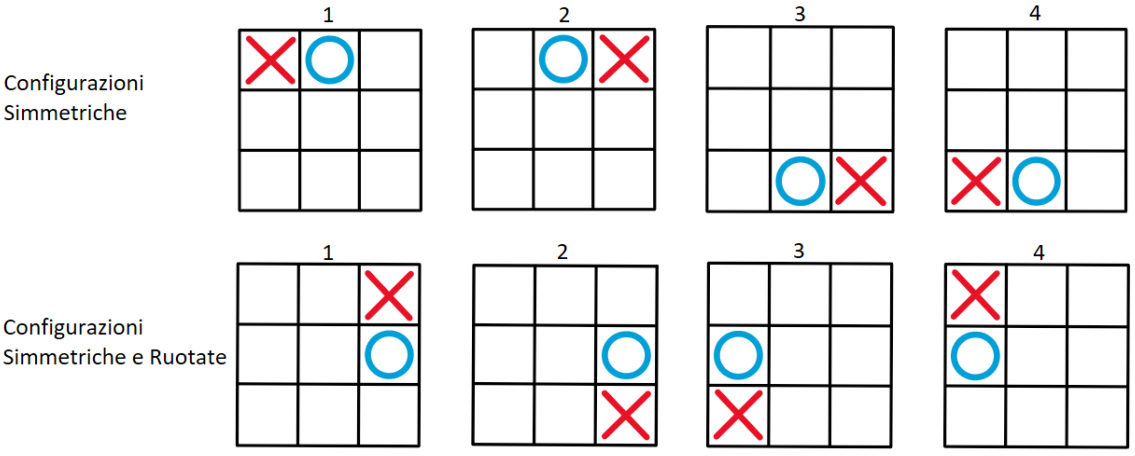
- **Zobby, spec_Zobby, bot_Zobby, bot_Spec_Zobby** memorizzano le codifiche di tutte le configurazioni simmetriche a quelle di partenza.

- **rtZobby, rtspec_Zobby,rt bot_Zobby,rt bot_Spec_Zobby** memorizzano codifiche di tutte le configurazioni simmetriche rispetto a quella di partenza, con un rotazione di 90 gradi in senso orario. Affinché la rotazione abbia senso la Board di gioco deve essere quadrata (M == N).

Di fatto, configurazioni simmetriche a quella principale e alla sua ruotata, rappresentano tutte le possibili configurazioni con la stessa valutazione(*im. 1.1*). Nella **Transposition Table** verrà mappata una sola chiave (**ZobristKey**) che rappresenterà tutte le 8 configurazioni. Attraverso la funzione **findRightKey()** avviene l'associazione tra la codifica di una configurazione e il corrispettivo speculare mappato nella Transposition Table attraverso **ZobristKey**.

Si utilizza il metodo **generateKeys()** per calcolare tutti i valori (8 nel caso M==N, 4 altrimenti). Si sfruttano le proprietà dello **XOR**; in particolare la proprietà di **involuzione**: x XOR y XOR y= x. Dunque le chiavi si possono aggiornare al passo con i markCell() e unmarkCell(), senza dover ricalcolarle da 0. Ad ogni markcell() corrisponde uno XOR tra la codifica corrente e il valore , preso dalle matrici sopra citate ,della cella appena marcata. Specularmente avviene con l'unmarkCell(). In seguito ad un markCell() e ad un unmarkCell() della stessa casella, la codifica risultante sarà uguale a quella precedente alle due operazioni, grazie alla proprietà di involuzione.

.	Codifiche	Codifiche (matrici ruotate)
1 - Principale	11 ^ 12 = 7	76 ^ 39 = 107
2 - Speculare	25 ^ 12 = 21	90 ^ 39 = 125
3 - Sottosopra	76 ^ 17 = 93	66 ^ 52 = 118
4 - Spec. e Sott.	66 ^ 17 = 83	25 ^ 52 = 45



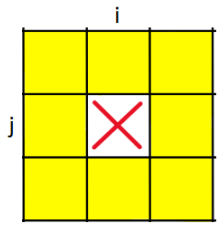
im 1.1

I valori delle matrici casuali fanno riferimento all'immagine 1.0

HashMapNFC

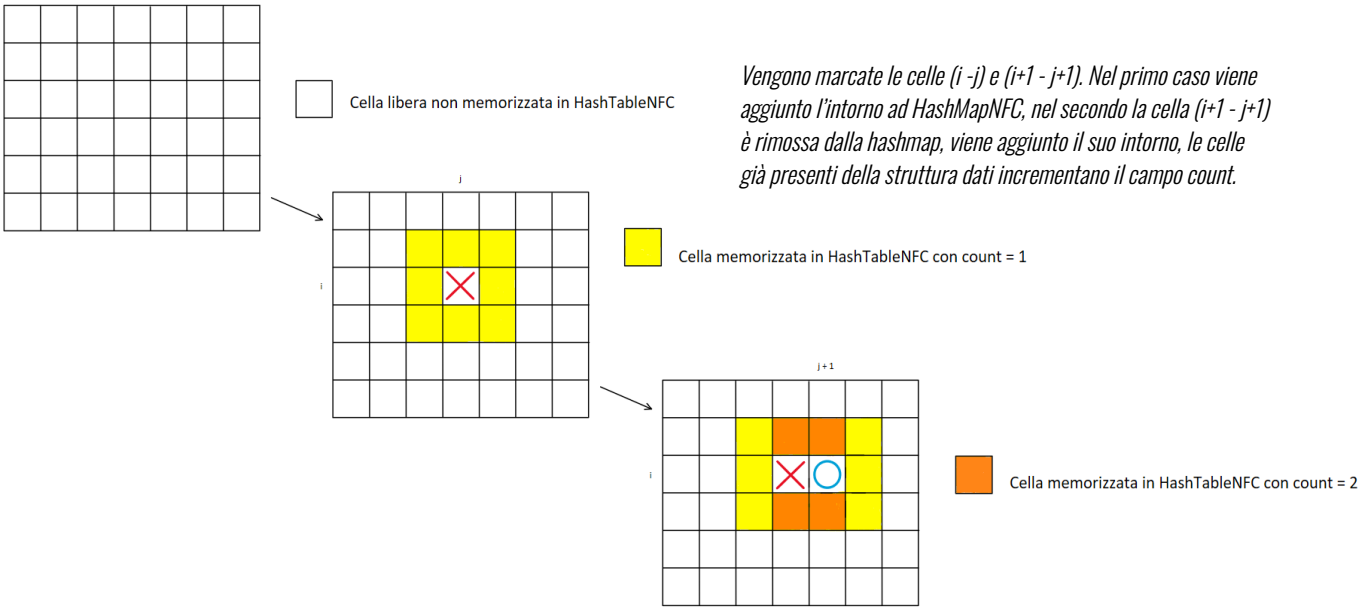
La popolazione della Board durante l'algoritmo di Iterative Deepening e AlphaBeta può essere personalizzata evitando la scelta casuale delle caselle man mano che sviluppiamo l'albero di gioco. Inoltre permette all' Iterative deepening di raggiungere profondità accettabili senza analizzare ogni volta M*N celle.
HashMapNFC (Near Free Cells) è una struttura dati creata ad hoc per la questione sopra citata; si basa su un Hashmap di **NFCell**. La NFCell contiene il parametro **count** che verrà spiegato in seguito.
Ogni cella della board sarà identificata univocamente da una Key dell'hash table (es: la cella 2 -2 è associata alla chiave 22) il costo per la ricerca nella HashMapNFC sarà O(1). Oltre all'implementazione di metodi base (add, delete, contains...) il cuore della classe sta nelle funzioni **fillNFCplus** e **deleteNFCplus**.

FillNFCplus permette di aggiungere all HashMap l'intorno di una cella marcata; dove per intorno si intendono le celle libere adiacenti alla marcata.



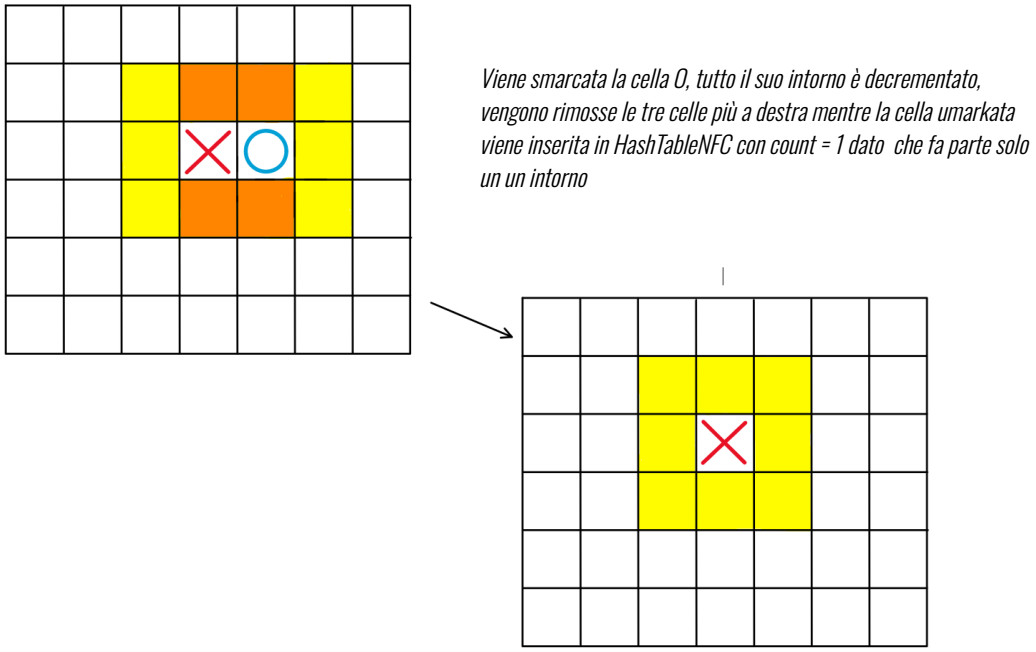
Intorno della cella i - j

Durante l'alphabeta il popolamento della tabella partirà dalle celle nelle vicinanze di quelle marcate, aumentando la probabilità di analizzare situazioni critiche e evitando la scelta finali di celle troppo distanti dalla zona della board in cui si sta sviluppando la partita.
Grazie al parametro count della NFCell il riempimento e lo svuotamento della tabella hash viene eseguito in modo corretto. Se una cella libera è considerata più volte, dato che è presente nell'intorno di due o più celle marcate diverse, allora count aumenta.



Vengono marcate le celle (i -j) e (i+1 - j+1). Nel primo caso viene aggiunto l'intorno ad HashMapNFC, nel secondo la cella (i+1 - j+1) è rimossa dalla hashmap, viene aggiunto il suo intorno, le celle già presenti della struttura dati incrementano il campo count.

Lo svuotamento (**deleteNFCplus**) avviene in questo modo: quando viene eseguito un unmark, decrementa count in tutto l'intorno della cella unmarkata, solo quanto count == 0 la cella può essere considerata può essere rimossa dalla HashTableNFC .
Infine aggiunge ad HashMapNFC la cella che sta venendo smarcata. Il parametro count di questa cella è inizializzato dal metodo **numberNFC**, che in base alle celle marcate adiacenti riesce a risalire a quanti intorni appartiene.



Viene smarcata la cella O, tutto il suo intorno è decrementato, vengono rimosse le tre celle più a destra mentre la cella umarkata viene inserita in HashTableNFC con count = 1 dato che fa parte solo un un intorno

Costi computazionali:

I costi di HashMapNFC sono relativamente bassi: ricerca, aggiunta ed eliminazione di una cella hanno costo medio uguale a **O(1)** come studiato a lezione. Le funzioni fillNFCPlus e deleteNFCPlus hanno entrambe un numero costante di operazioni quindi il costo il costo è Θ(1).

Heuristic

L'euristica utilizza 4 hashset: **hor, ver, diag e antidiag**. La scelta è ricaduta su questa struttura dati a causa del costo di add() e contains() , entrambi O(1). Viene utilizzato anche il clear() dal costo di O(n), dove n è la grandezza dell'hashset.
L'euristica si basa su due tipi di valutazioni: una per le minacce (vedi *Minacce nell'Euristica*), l'altra per la parte di gioco che viene prima di queste. Entrambe sono gestite da **evaluate()**.

Evaluate()

Restituisce dei valori standard in caso di fine partita (10000000 in caso di WinP1, -10000000 in caso di WinP2, 0 in caso di Draw), altrimenti la valutazione passa **all'euristica**.
Viene eseguito un ciclo for su tutte le celle marcate . Su ogni cella, in caso non siano contenute negli hashset (e dunque non siano state visitate), vengono chiamate le funzioni: **evaluateHorThreat(),evaluateVerThreat(), evaluateDiagThreat(), evaluateAntiDiagThreat()**, che valutano la relativa minaccia sulla Board.
Definendo :

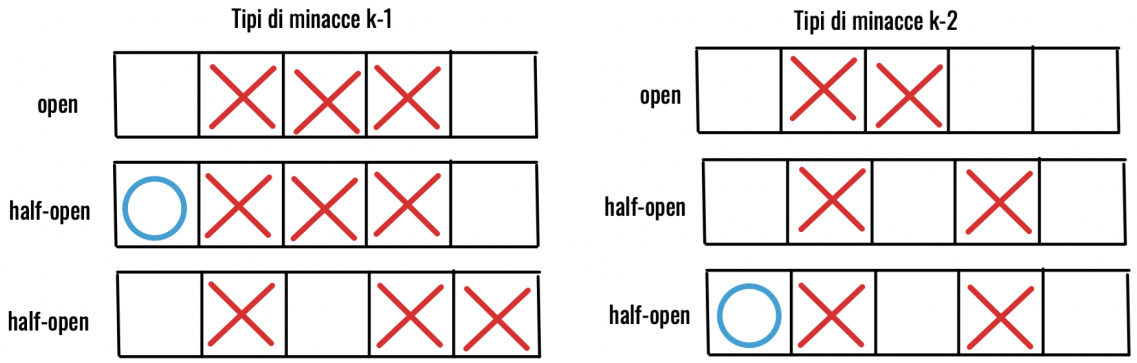
- **Salto** il caso in cui ci sia una casella vuota tra due elementi dello stesso giocatore (X o O);
- **MNKCell start e MNKCell arrive** le celle alle estremità dell'allineamento (che serviranno poi per controllare il tipo di minaccia),

distinguiamo due tipi di minacce:

- **Minaccia aperta (Open Threat)**: caso in cui ci siano *n* elementi in fila, le cui estremità siano vuote e in cui non sia avvenuto alcun salto.
- **Minaccia semi-aperta (Half-open threat)**: caso in cui ci siano *n* elementi in fila, le cui estremità possono non essere vuote e può essere avvenuto un salto.

(in entrambi i casi n è il numero di elementi che cerchiamo, esempio k-2 o k-1)

Esempio (considerando k = 4) :



Il Valore ritornato sarà assegnato al current player se la cella valutata è del player corrente, altrimenti in opponent player.

evaluate*Threat()

Lo scopo della funzione è quello di individuare le minacce k-2 e k-1.
Il controllo viene eseguito a partire da una cella. Si esegue un controllo all'indietro (backward) e in avanti (forward); tale controllo si fermerà in caso venga trovata una casella diversa dalla propria dopo aver eseguito un salto, oppure nel caso in cui ci si allontani troppo dalla cella (k celle, dove k è il numero di celle per vincere).
L'assegnazione dei valori avviene secondo tale principio:

minacce k-1:

	open	half-open
Current	250	80
Opponent	5020	1500

minacce k-2:

	open	half-open
Current	150	60
Opponent	1200	1000

Il valore di k-1-open per **l'opponent** è maggiore rispetto a quello del **current** poiché consente all'opponent di diventare matematicamente vincitore qualora il current non sventasse la minaccia.
In generale si può notare come i valori per le minacce dell'opponent Player siano di molto maggiori rispetto a quelli del current. Le cifre più elevate consentono al current di arginare le minacce che gli saranno poste dall'opponent nei turni successivi.
Affinchè non vengano controllate le minacce sulle stesse celle, queste ultime vengono inserite (**add()**) nel corrispettivo hashset durante la scansione.

Per distinguere il tipo di minaccia, le funzioni **is*Open()** prendono in input la cella di **start** e quella di **arrive** e restituiscono quante celle libere ci sono alle estremità. In caso siano **2**, e non si sia eseguito alcun salto, si tratta di minaccia aperta. Il costo è O(1)

Tuttavia, non sempre ci sono minacce sulla Board, dunque si utilizza una valutazione secondaria che viene sommata a quella delle minacce. Quest'ultima viene moltiplicata *150 affinchè la scelta si basi principalmente su di essa.

Check()

L'idea alla base è il conteggio di quante possibili linee vincenti ci sono a partire da un determinato elemento.
A partire da una cella (stessa cella iterata nel for per l'evaluate*Threat), viene contato in ogni direzione il numero di celle diverse da quelle avversarie. Ogniqualvolta si trovi una cella dello stesso tipo, viene incrementato il valore della valutazione totale per dare un bonus alle celle allineate. Se il controllo in una direzione viene stoppato a causa dell'incontro di una cella avversaria, si decrementa il valore della valutazione totale. In caso le celle totali siano >= B.K, si incrementa due volte il valore della valutazione totale; altrimenti la valutazione, per quella direzione, viene azzerata se è positiva, rimane invariata se è

negativa. La valutazione totale di una cella sarà data dalla somma delle scansioni in ogni direzione. Questo valore sarà assegnato al **MaxPlayer**, in caso la cella analizzata fosse del player P1 o al **MinPlayer** in caso contrario.

Costi computazionali:

Nel ciclo for vengono eseguiti, nel caso pessimo, 4 Evaluate*Threat() e un check().
Il costo di Evaluate*Threat nel caso pessimo: O(k) + O(k) + O(1) = O(k).
Il costo di Check() è O(k).
Dunque il costo di un ciclo for, nel caso pessimo, è **O(nk)**, dove *n* è il numero di celle marcate, *k* il numero di celle allineate per la vittoria.
Successivamente viene eseguita la pulizia degli hashset, dal costo O(n).
Il costo finale dell'euristica quindi rimane O(nk)

Altri Metodi

estimateTime()

In seguito a vari test eseguiti su macchine diverse, si è arrivati alla realizzazione di una funzione che adatta il limite di tempo in base a quanto se ne utilizza in ogni selectCell(). Il tempo utilizzabile può diminuire se supera una certa soglia, rischiando quindi di produrre un errore di timeout (ovvero, se il tempo utilizzato è >= TIMEOUT - 0.06, il tempo limite viene diminuito). Al contrario, se il tempo utilizzato è molto distante dal TIMEOUT, si aggiungono centesimi di secondo al tempo limite, non superando il limite 0.06.

bestMove()

Metodo che, dopo aver eseguito un alphabeta, trova la miglior mossa tra il valore ritornato da alpha beta e quelli derivanti dalle iterate precedenti.

Bibliografia

Minacce nell'Euristica: http://www.cari-info.org/Actes-2018/p276-286.pdf

Utilizzo della transposition table con algoritmi di decisione sugli alberi: https://en.wikipedia.org/wiki/Negamax

Idee per la TranspositionTable: http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html

Zobrist: https://youtu.be/QYNRvMoIN20

Complessità computazionale per le Java Collections: https://www.baeldung.com/java-collections-complexity