Progetto Algoritmi a.s. 2021/2022

Team

Xuanqiang Huang : 0001030271
Giovanni Spadaccini : 0001021270

Riassunto

Abbiamo utilizzato un algoritmo di Minimax euristico con alpha-beta pruning per la risoluzione del *gioco mnk*, una forma generalizzata del tris. L'algoritmo utilizza l'euristica di valutazione per scoprire l'ordine di esplorazione di un numero limitato di nodi e li esplora fino a un livello di profondità prefissato, dopo il quale ritorna un valore euristico, nel caso la board non sia terminale, o un valore finale, nel caso in cui la board sia terminale.

Dai test fatti in locale, l'algoritmo sembra avere capacità simili o superiori a quelle di un umane per le tavole accessibili ai limiti umani (ossia minori di ~15).

Introduzione

Il gioco proposto è una versione generalizzata del gioco tris, o gomoku in Giappone: un gioco a due giocatori a turni alterni su una tavola simile a quanto in immagine, in cui bisogna allineare un certo numero di pedine del proprio giocatore al fine di vincere.

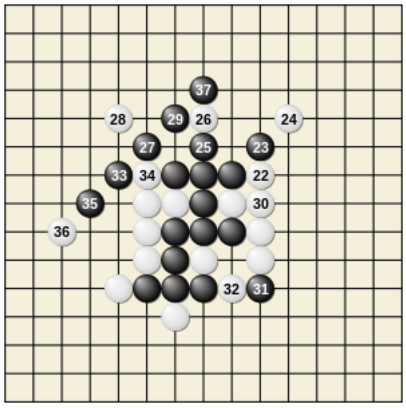

Figure 1: Esempio di tavola di gioco di gomoku in cui il giocatore nero ha vinto

Seguendo la caratterizzazione di un ambiente di gioco di Russel e Norvig[1] possiamo descrivere il gioco come un ambiente deterministico, multigiocatore, con informazioni complete, sequenziale, statico, conosciuto e discreto. Questa descrizione ci ha permesso di avere una prima idea di quali algoritmi potessero essere utilizzati per la risoluzione del gioco, in quanto in letteratura questo

problema, e problemi simili, sono stati risolti con tecniche che ormai possiamo considerare *classiche*.

Sotto questa logica abbiamo scelto l'implementazione di un minimax euristico.

L'algoritmo ad alto livello

Il nostro algoritmo minimax euristico fa una stima iniziale di quanto può esplorare, dopodiché utilizza una euristica per decidere l'ordine di eslorazione di un numero limitato di mosse. Quest'ultime saranno visitate in un numero limitato di nodi, determinato da due costanti che indicano rispettivamente il numero di nodi da espandere in ampiezza e la profondità di esplorazione.

Argomenti trattati

- *Markcell e unmarkCell*
  - Versione più veloce e cache-friendly per segnare le celle come marcate e per annullare quanto marcato
- *Euristica*
  - Spiegazione, calcolo e utilizzo dell'euristica per il nostro minimax
- *Ordinamento mosse*
  - Sull'algoritmo utilizzato per ordinare le mosse a seconda del valore restituito dall'euristica
- *Timer test*
  - Sull'algoritmo usato per avere una stima della quantità di nodi esplorabili
  - Sui metodi di allocazione di un numero di mosse preciso alle celle scelte per l'esplorazione
  - Sulla scelta dei valori di ramificazione e profondità
- *Analisi del costo*
  - Sulla complessità temporale e spaziale dell'algoritmo

Markcell e unmarkCell

Problema

Marcare e smarcare le celle nella maniera più veloce possibile.

Algoritmi considerati

1. Riallocazione ad ogni nodo di esplorazione un array con le `freeCell` meno quella appena marcata (nel caso di markCell), o di creazione di un array con tutte le `freeCell` con anche l'ultima mossa eseguita (nel caso di unmarkCell), costo $O(n)$
2. utilizzo di una linked list che contenga le celle libere. Questo approccio nonostante avesse le operazioni di insert e remove in tempo constante performava peggio rispetto al riallocamento in un array ad ogni nodo di

esplorazione (che ha costo $O(n)$) (pensiamo che questo sia dovuto alle ottimizzazioni cache degli array)

3. utilizzo di hashset: ha un costo lineare nel caso pessimo oltre a delle costanti molto alte nel caso medio.

Algoritmo utilizzato

Abbiamo ideato un sistema che è in grado di eseguire le operazioni di `markCell`, `unmarkCell` in tempo costante nel caso ottimo, pessimo e medio, senza considerare i checks aggiuntivi per verificare lo stato del gioco e l'aggiornamento dell'euristica.

Al fine di raggiungere questa velocità, l'insieme delle mosse eseguite è tenuto alla fine di un array che contiene tutte le mosse, in modo simile a quanto fa una heap al momento di rimozione.

Mano a mano che le celle vengono utilizzate, esegue uno swap con l'ultima cella dell'array marcata come libera e si memorizza la posizione in cui era, così facendo, quando viene chiamata l'unmarkCell, riesce a riposizionarsi nella sua vecchia posizione. Questa implementazione migliora rispetto alla board del codice iniziale nel caso pessimo, in quanto non deve più necessitare di un hashtable, il cui caso pessimo è $O(n)$ con n la grandezza della table.

Nota: tutte le celle contengono un **index** che indica la posizione dell'array in cui è contenuta la mossa, altrimenti, se questa non è ancora stata rimossa indica la posizione di ritorno.

Pseudocodici per queste due funzioni sono presenti in appendice

L'Euristica

In questo progetto sono fondamentali le euristiche utilizzate al fine del successo del Player. Esiste una unica euristica che viene utilizzata, in modi diversi, sia per la valutazione della board sia per la scelta delle mosse.

Cosa rappresenta

L'euristica utilizzata mi restituisce un valore di importanza della singola cella sia per me sia per il mio avversario.

La somma dei due valori di importanza mi dà una stima di criticità della singola cella.

Con questo valore numerico è possibile ordinare le mosse secondo un ordine di priorità.

Questa euristica è una versione modificata dell'euristica proposta da Nathaniel Hayes and Teig Loge [3] adattata per board di dimensione maggiore di 3x3 o 4x4, insieme a una versione, sempre modificata, dell'euristica di Chua Hock Chuan [4], per la valutazione di allineamenti critici di K - 1 e K - 2.

Calcolo dell'euristica con le sliding window

> Definiamo **Sliding-window** un insieme di celle allineate in una direzione di lunghezza `K`

L'euristica calcola per ogni direzione di una singola cella e per entrambi i player i seguenti valori:

1. Il numero di celle amiche presenti
2. Il numero di sliding windows che passano per una cella
3. Massimo numero delle sliding-window con minor numero di celle necessarie per la vittoria
4. Il numero di sliding-window massime

Per fare ciò aggiorniamo tutte celle nelle 4 direzioni possibili fino a una distanza massima `K`, ed andiamo ad riaggiornare i valori in di queste celle nella direzione attraverso la quale si allineano con la cella modificata. Per riaggiornare questi valori chiamiamo la funzione `updateDirectionValue`.

La funzione che aggiorna una singola cella per una direzione è implementata in `computeCellDirectionValue`.

Questa funzione esplora la cella attuale, in direzione orizzontale o verticale, e si allarga fin quanto può verso una direzione (al massimo di `K - 1`), una volta ragginto il limite in questa direzione, si espande nella direzione opposta, mantenendo la sliding window nel caso sia stata creata. Mentre si espande anche dall'altra parte, finchè non va oltre le `k-1` celle o finchè non trova una cella dell'altro player, si aggiorna i valori della sliding window corrente, e aggiorna i valori della cella di cui sta facendo l'update.

Rilevamento dei doppio-giochi e fine-giochi

Con il sistema a sliding window possiamo anche rilevare con molta facilità alcune celle *critiche* ossia situazioni di doppi giochi oppure giochi ad una mossa dalla fine.

> Definiamo **fine-giochi** le celle per cui esiste almeno una sliding window a cui manca 1 mossa per vincere

È chiaro come queste celle siano molto importanti sia per noi, al fine della vittoria, sia per il nemico, al fine del blocco.

> Definiamo **doppi-giochi banali** le celle per cui esistono due o più sliding window per cui mancano 2 mosse per vincere

Se abbiamo una tale configurazione, si può notare come muovere su quella cella riduce le mosse per vincere di 1 in entrambe le sliding window. Abbiamo allora due sliding window in cui manca una mossa per vincere, per cui il nemico può bloccare al massimo una, garantendoci la vittoria sull'altra.

Riguardo i doppi-giochi non banali, ossia doppi giochi in cui abbiamo bisogno di 3 o più mosse per vincere ci basiamo sulla capacità del minimax di scovarli, non siamo riusciti a trovare un modo per codificare questo caso tramite le sliding-window.

Punteggi per configurazioni di doppio-gioco e fine-gioco

Sono assegnati alcuni punteggi speciali alle celle di doppio-gioco o fine-gioco.

Queste configurazioni non sono espressamente visibili dall'euristica ispirata da Nathaniel Hayes and Teig Loge, per cui abbiamo assegnato dei valori fissi a *gradini*, ossia in qualunque modo si calcoli l'euristica precedentemente nominata, il valore euristico calcolato da quest'ultimo non può superare il valore assegnato da una cella di doppio gioco, e quest'ultima non può superare il valore di una cella di fine-gioco.

Quindi in ordine di importanza abbiamo:

1. Cella di fine-gioco
2. Cella di doppio-gioco banale
3. Cella valutata dall'euristica di Nathaniel Hayes and Teig Loge + punteggi allineamento e vicinanza.

Punteggi per allineamento e vicinanza

Con prove empiriche abbiamo notato che l'euristica come spiegata fino a ora non è in grado di valutare correttamente alcune situazioni di allineamento, per cui abbiamo aggiunto dei moltiplicatori di punteggi per l'allineamento di celle e per favorire le mosse vicine ad alcune celle già allineate.

Si possono osservare i valori di questi moltiplicatori `MY_CELL_MULT` e `ADIACENT_MULT` rispettivamente nel file di `DirectionValue` e `HeuristicCell`.

Questi valori si sono rilevati fondamentali per il gioco intelligente del player.

Ordinamento delle mosse

Problema

Ad ogni momento dalla board abbiamo la necessità di trovare le migliori `q` celle ordinate in modo decrescente, che determineranno l'ordine di ricerca.

Su un array di `n` celle libere, dobbiamo ordinare le prime `q` che hanno il valore più alto

Approcci presi in considerazione

1. Il primo sorting normale che andava in $O(n \log n)$ dove n sono le celle libere

2. Quick select, che separa correttamente i `q` elementi più grandi, ma non li ordina. Questo sarebbe costato $O(n + q \log q)$ nel caso medio e e tempo d'esecuzione peggiore però sarebbe stato $O(n^2 + q \log q)$ dove $O(n^2)$ nel caso peggiore di quick-select e $O(q \log q)$ per ordinare le celle.

Algoritmo Utilizzato

Il metodo che abbiamo utilizzato sfrutta una leggera variazione dell'algoritmo di heap-select: si scorre l'array delle celle libere tenendosi una heap di massimo `q` elementi di questa, e infine svuota la heap e la mette in un array che contiene le prime `q` celle sortate.

Costo computazionale $O(n \log q)$ in quanto eseguiamo una operazione di inserimento di costo $O(\log q)$ nella heap, $O(n)$ volte

Questo miglioramento sui primi test è riustito a far esplorare la board 5 o 6 volte più mosse a parità di tempo in input.

Un esempio di utilizzo di questo algoritmo lo si può trovare in `updateCellDataStruct` della Board. In questo caso utilizziamo la variabile `branchingFactor` per tenere il valore di `q` molto basso.

Algorithm 1: Algoritmo di ordinamento delle mosse

Input : int l, numero elementi della heap
Input : array: array di elementi comparabili di lunghezza n t.c. n > l
Output: un array con le l elementi maggiori di array

```
1  queue = new MinPriorityQueue();
2  for element in array do
3  |  if queue.size() < l then
4  |  |  add element to queue;
5  |  else if get first element of queue < element then
6  |  |  delete top of queue;
7  |  |  add element to queue;
8  |  end
9  end
   // dopo il ciclo abbiamo esattamente l elementi nella queue
10 out = new Array of l elements;
11 int i = l - 1; while queue.size() > 0 do
12 |  out[i] = get first element of queue;
13 |  delete top of queue;
14 |  i = i - 1;
15 end
16 return out;
```

Timer Test

Problema

Al fine di utilizzare tutto il tempo computazionale a disposizione era presente la necessità di trovare un metodo che permettesse di esplorare i nodi interessanti sfruttando al meglio il tempo a disposizione.

Approcci provati

1. Un check di timeout su ogni nodo del minimax: questo approccio non funzionava perché era presente il rischio che al primo livello venissero esplorati pochi nodi, a causa dell'esplorazione in profondità.
2. Esplorazione di una parte di albero di ampiezza e profondità prefissata: questo approccio portava il rischio del tuning dei parametri di profondità e ampiezza, che potevano cambiare a seconda del calcolatore.

Soluzione utilizzata

Alla fine abbiamo utilizzato idee da entrambi i metodi, creando una simulazione del processo di decisione che esplorasse quanti nodi più possibili e dasse una stima di quanti era possibile visitare, mantenendo sempre delle costanti di profondità e ampiezza prefissati per le varie tipologie di board.

Al fine di avere una stima di quanti nodi di ricerca un qualunque calcolatore potesse elaborare con un limite di tempo prefissato abbiamo utilizziamo la classe `TimingPlayer`, che simula il processo di decisione del nostro algoritmo, tenendosi conto di quanti nodi è riuscito a visitare entro la fine del tempo concesso.

Spartizione e utilizzo del numero di mosse

Problema

Vorremmo che le celle più promettenti abbiano più tempo a disposizione per l'esplorazione. Dobbiamo creare un metodo per distribuire il numero di mosse disponibili durante l'esplorazione del minimax.

Soluzione utilizzata

Abbiamo visto che in seguito all'ordinamento con l'euristica le prime celle sono quelle più importanti da esplorare, dato che favorisce la potatura di altri nodi di ricerca grazie ad alpha-beta pruning. Quindi vogliamo distribuire più mosse alla prima cella, in modo che possa avere una esplorazione più approfondita e una più alta probabilità di potatura.

In `findBestMove` vediamo come sono utilizzati il numero di celle trovate in questo modo:

Nel caso in cui la prima cella non utilizzi tutte le celle date, queste saranno affidate alle celle di esplorazione successive. La cella di esplorazione successiva può, quindi, esplorare un numero di celle uguale a `numero nodi non utilizzati precedenti + addendo di nuove celle da esplorare`. Così per le prime `branchingFactor * 3` con i valori euristici più grandi.

Scelta del fattore di ramificazione e profondità

Problema

I fattori di ramificazione e profondità hanno un inpatto diretto sul tempo di esecuzione dell'algoritmo, e sulla qualità della mossa scelta Diventava quindi molto importante trovare i valori corretti da assegnare per ogni board.

Soluzione utilizzata

Eravamo coscienti della possibilità di utilizzare metodi di apprendimento automatico al fine di trovare in questi valori.

Tuttavia non eravamo a conoscenza dei metodi di applicazione nel nostro ambiente, né se i valori potevano dipendere dal calcolatore su cui veniva eseguito il programma.

Abbiamo quindi deciso di utilizzare alcuni valori fissati, che abbiamo trovato in base a delle prove empiriche.

Analisi del costo

In questa sezione esponiamo un'analisi del costo computazionale del nostro algoritmo passo dopo passo.

markCell e unmarkCell

Sia il mark e l'unmarkCell come prima cosa devono aggiornare le celle libere, cosa che entrambe le funzioni fanno in $O(1)$. Poi devono aggiornare tutti le celle vicine alla cella modificata in $O(k^2)$. Dopodiché aggiornare le celle sortate $O(nm \log branchingFactor)$ (il branching factor è una costante che va da 7 nei casi piccoli a 3 in quelli grandi).

Quindi mark e unmark cell hanno un costo computazionele di $O(mn \log branchingFactor)+$ $O(k^2)$.

minPlayer e maxPlayer

Questi due metodi sono i giocatori della minimax, rispettivamente il player minimo e massimo.

Ognuno di questi esegue operazioni che dipendono dal costo di markCell e unmarkCell, e da `BRANCHING_FACTOR` e `DEPTH` che sono costanti.

Sia quindi $C(k)$ il costo di markCell e unmarkCell, queste operazioni sono eseguite ad ogni nodo fino alla depth fissata. Si ha un costo di $(C(k) \cdot branchingFactor)^{DEPTH-depthRaggiunta}$ per questi due algoritmi nel caso pessimo in cui vengono visitati tutti i nodi e il pruning non viene mai eseguito.

SelectCell e findBestMove

`SelectCell` chiama in primo momento `findBestMove` per trovare la mossa migliore.

findBestMove esegue delle operazioni costanti e chiama `updateCellDataStruct` che riordina le celle in $O(n \log q)$ come descritto nella sezione riordinamento mosse.

findBestMove riordina le mosse in attraverso l'euristica e chiama il minimax con alpha beta pruning in ordine su quelle che hanno un valore maggiore. Il costo di questo algoritmo è $O(nm \log branchingFactor + branchingFactor \cdot (Costo(minPlayer) + Costo(marckCell) + Costo(unmarkCell))$

il costo del minPlayer nel caso peggiore è $(O(mn \log branchingFactor + k^2) \cdot branchingFactor)^{DEPTH - depthRaggiunta}$

Il costo sia di markCell che unmarkCell nel caso peggiore è $O(mn \log branchingFactor + k^2)$

Quindi il costo nel caso peggiore del nostro algoritmo è $O(nm \log branchingFactor) + O(branchingFactor \cdot ((O(mn \log branchingFactor) + O(k^2)) \cdot branchingFactor + O(mn \log branchingFactor) + O(k^2))^{DEPTH}$

Se consideriamo `branchingFactor` come se fosse una costante, allora abbiamo che il costo nel caso pessimo è $O(nm) + O((O(mn) + O(k^2)) + O(mn) + O(k^2))^{DEPTH} = O(nm) + O(nm + k^2)^{DEPTH} = O(mn + k^2)^{DEPTH}$

Analisi dello costo in memoria

Il nostro algoritmo è molto efficiente dal punto di vista dello spazio utilizzato: $\Theta(mn) + O(DEPTH)$.

Gli unici oggetti che sono memorizzati sono le `HeuristicCell` che rimangono sempre le stesse per una intera partita, senza essere mai distrutte o ricreate nel mezzo.

All'inizio del gioco vengono creati un numero di celle uguale a MN che vengono memorizzate in 3 strutture di dati differenti di grandezza MN.

Rispettivamente sono:

1. `Board`, che contiene tutte le celle in un array bidimensionale di dimensione $M \cdot N$.
2. `allCells`, che contiene le stesse celle in un array unidimensionale di dimensione MN. Questo array è utilizzato per markCell e unmarkCell
3. `sortedAllCells`, che contiene una parte delle celle ordinate in base all'euristica. Questo array è utilizzato per decidere un ordine di esplorazione.

Durante l'esplorazione con la minimax, la board non viene mai ricreata, ma viene modificata e ripristinata ad ogni mossa, garantendo grande efficienza in

termini di memoria. Durante questa esplorazione sono aggiunti alla stack del programma dei frame di chiamata, che influiscono leggermente sul costo in spazio. Ma essendo la `DEPTH` un valore costante, si potrebbe considerare questo apporto ininfluente rispetto al resto.

Per cui possiamo affermare che il costo in memoria sia $\Theta(mn) + O(1) = \Theta(mn)$.

Approcci fallimentari

1. Simulazione di Montecarlo (MCTS), che abbiamo provato dato il grande successo di AlphaGo
    1. Guardava celle che avevano poco valore per la vittoria
    2. Guardava tutti gli stati ad ogni livello, il che pesava molto anche sulla memoria (poichè si teneva tutto il game tree)
    3. Il limite di tempo era troppo basso per avere un numero di simulazioni sufficenti
    4. La capacità dell'hardware incideva molto sui risultati.
2. Puro algoritmo heuristico (anche conosciuto come Greedy best first Search)
    1. Non riusciva ad andare in profondità, dato che selezionava ogni volta la cella con maggiori probabilità di vittoria al primo livello, questo non gli permetteva di pianificare le proprie mosse.
3. alpha beta pruning puro, per i test grandi impiegava troppo tempo d'esecuzione, non riuscendo a fare l'eval di neanche una mossa
4. rule-based strategy[2], basato su 5 steps che riporto qui testualmente: Rule 1 If the player has a winning move, take it. Rule 2 If the opponent has a winning move, block it. Rule 3 If the player can create a fork (two winning ways) after this move, take it. Rule 4 Do not let the opponent create a fork after the player's move. Rule 5 Move in a way such as the player may win the most number of possible ways.
    1. Queste regole sono state molto importanti come guida del nostro progetto, nonostante non siano applicate in modo esplicito, hanno guidato il valore fisso per le celle di doppio-gioco e fine-gioco.
5. Iterative Deepening, come l'alpha-beta pruning non era in grado di esplorare una parte sufficiente dell'albero di ricerca.

Miglioramenti possibili

1. Utilizzare un sistema ad apprendimento automatico per decidere il `BRANCHING_FACTOR` e la `DEPTH_LIMIT` che ora sono di valori fissati, secondo l'esperienza umana.
2. Utilizzare più threads per l'esplorazione parallela dell'albero di ricerca (non possibile per limiti imposti).
3. Update dell'euristica in $O(k)$ invece dell'attuale $O(k^2)$, dove k è il numero di celle da allineare.

Conclusione

Abbiamo osservato come un classico algoritmo Minimax con alpha-beta pruning possa giocare in modo simile, o superiore rispetto a un giocatore umano medio per le board di grandezza adeguata per l'umano, data una euristica che gli permetta di potare ampi rami di albero.

Appendice

Markcell e unmarkCell

Algorithm 2: markCell senza checks sulla board

Result: Cella richiesta della tavola è marcata
Input : int $freeCellsCount$: numero di celle libere, è compreso fra 1 e allCells.length
Input : int $index$: l'index della cella da marcare, compresa fra 0 e $freeCellsCount$
Input : Cell[] $allCells$: array tutte le celle, in cui le prime $freeCellsCount$ sono considerate libere
Output: void: viene modificata allCells

```
1 allCells[freeCellsCount - 1].index = allCells[index].index;
2 swap(allCells[freeCellsCount - 1], allCells[index]);
  // marca la cella come occupata dal giocatore
3 mark(allCells[freeCellsCount - 1]);
4 freeCellsCount = freeCellsCount - 1;
```

Algorithm 3: unmarkCell senza checks sulla board

Result: Viene rimosso l'ultima mossa della tavola
Input : int $freeCellsCount$: numero di celle libere, è compreso fra 0 e allCells.length - 1
Input : Cell[] $allCells$: array tutte le celle, in cui le prime $freeCellsCount$ sono considerate libere

```
  // marca la cella come libera
1 markFree(allCells[freeCellsCount - 1]);
2 swap(allCells[allCells[freeCellsCount].index], allCells[freeCellsCount]);
3 allCells[freeCellsCount].index = freeCellsCount;
4 freeCellsCount = freeCellsCount + 1;
```

Pseudocodice di un minimax euristico

Algorithm 4: Minimax Euristico riadattato da Norvig e Russel [1]

```
1  Function MiniMax-Search(game, state)
       // Cerca fra le mosse la migliore possibile
       // move è l'azione da intraprendere
2      value, move = Max-Value(game, state);
3      return move;

4  Function Max-Value(game, state, alpha, beta)
5      if game.terminalTest(state) then return game.utility(state), null;
6      value, move = -∞;
7      for each action in game.actions(state) limitato da una branching factor
        do
8          v, m = Min-Value(game, game.result(state, action), alpha, beta);
9          if v > value then
10             value, move = v, action;
11             alpha = Max(alpha, value);
12         end
13         if v ≥ beta then
14             return value, move;
15         end
16     end
17     return value, move;

18 Function Min-Value(game, state,alpha, beta)
19     if game.terminalTest(state) then return game.utility(state), null;
20     value, move = ∞;
21     for each action in game.actions(state) limitato da una branching factor
        do
22         v, m = Max-Value(game, game.result(state, action),aplha,beta);
23         if v < value then
24             value, move = v, action;
25             beta = Min(beta, value);
26         end
27         if v ≤ alpha then
28             return value, move;
29         end
30     end
31     return value, move;
```

References

1. Russell, Stuart J., e Peter Norvig. Artificial Intelligence: A Modern Approach. Fourth edition, Global edition, Pearson, 2022.
2. Development of Tic-Tac-Toe Game Using Heuristic Search IOP Publishing, 2nd Joint Conference on Green Engineering Technology & Applied Computing 2020, Zain AM, Chai CW, Goh CC, Lim BJ, Low CJ, Tan SJ
3. Developing a Memory Efficient Algorithm for Playing m, n, k Games, Nathaniel Hayes and Teig Loge, 2016.
4. Chua Hock Chuan, Java games, https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html, 2017