

M-N-K Game

Relazione del progetto per l'insegnamento di
Algoritmi e Strutture Dati

Erik Koci M. 0000997662
Paolo Ceroni M. 0000978232



Dipartimento di Informatica
Università di Bologna
A.A 2020/2021

Contents

1	Problema	2
2	Scelte progettuali	2
2.1	Negamax	3
2.1.1	Pseudocodice Negamax	3
2.2	Zobrist Hashing	4
2.2.1	Pseudocodice Zobrist Hashing	4
2.3	Tabella di trasposizione	4
2.4	Iterative Deepening	5
2.4.1	Pseudocodice Iterative Deepening	5
2.5	Valutazione euristica	6
2.5.1	Pseudocodice markWinCell:	6
2.6	Possibili euristiche precedentemente applicate	7
2.6.1	Pseudocodice $\leq K$	7
2.6.2	Pseudocodice nearCell	7
3	Conclusioni	8
3.1	Complessità computazionale	8
3.2	Alternative strategie progettuali	8

1 Problema

- Sviluppare un giocatore software in grado di giocare in modo ottimale a tutte le istanze possibili del (M,N,K)-game.
- Il numero di mosse intelligenti cresce esponenzialmente rispetto alla dimensione della matrice di gioco ed il numero di simboli da allineare.
- Tramite una dimostrazione per furto di strategia si può dimostrare che il secondo giocatore non può avere una strategia che gli assicuri la vittoria.

M	N	K	Risultato (assumendo due giocatori con strategia ottima)
3	3	3	Patta
4	3	3	Vittoria (Primo giocatore)
4	4	3	Vittoria (Primo giocatore)
4	4	4	Patta
5	4	4	Patta
5	5	4	Patta
5	5	5	Patta
6	4	4	Patta
6	5	4	Vittoria (Primo giocatore)
6	6	4	Vittoria (Primo giocatore)
6	6	5	Patta
6	6	6	Patta
7	4	4	Patta
7	5	4	Vittoria (Primo giocatore)
7	6	4	Vittoria (Primo giocatore)
7	7	4	Vittoria (Primo giocatore)
7	5	5	Patta
7	6	5	Patta
7	7	5	Patta
7	7	6	Patta
7	7	7	?
8	8	4	Vittoria (Primo giocatore)
10	10	5	?
50	50	10	?
70	70	10	?

Figure 1: Esempi di configurazione m,n,k

2 Scelte progettuali

Nello sviluppo progettuale del metodo **selectCell()** inizialmente abbiamo optato per un algoritmo di ricerca **negamax**, il quale è una variante dell'algoritmo *minmax*.

Questa tipologia di algoritmo viene spesso utilizzata nei giochi a turni tra due o più giocatori basandosi sulla somma di punteggi di ogni mossa. Successivamente le valutazioni effettuate dall'algoritmo saranno salvate in una matrice la quale ci permetterà di avere una **cache** interna di gioco, nella quale potremmo accedere nel caso trovassimo una combinazione valutata. Questo tipo di implementazione si basa sulla **programmazione dinamica**.

2.1 Negamax

L'algoritmo **negamax** (1) si basa sul fatto che $\max(a.b) = -\min(-a, -b)$ più precisamente, il valore di una posizione per il giocatore *A* in questa tipologia di gioco è la negazione del valore per il giocatore *B*.

Pertanto, il giocatore cercherà una mossa che massimizzi la negazione del valore risultante della mossa. Quest'ultima deve essere stata valutata per definizione dall'avversario.

Il ragionamento applicato funziona indipendentemente dal fatto che *A* o *B* faccia una mossa. Ciò implica che è possibile utilizzare un'unica **procedura** per valutare entrambe le posizioni.

Negamax è una semplificazione più elegante dell'algoritmo **minimax** che richiede che *A* selezioni la mossa con il valore di massimo mentre *B* seleziona la mossa con il valore di minimo.

Una volta terminata una esecuzione di gioco, la funzione *evaluate* determinerà il punteggio da attribuire a seconda del *game state*.

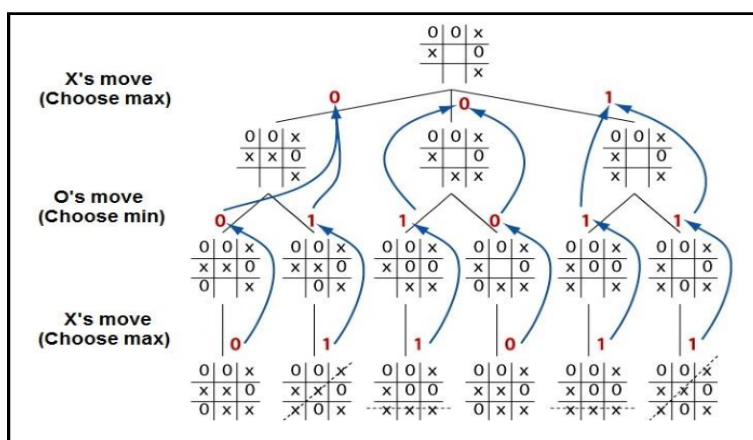


Figure 2: Negamax

2.1.1 Pseudocodice Negamax

```

1  function negamax(node, depth, color) is
2      if depth = 0 nodo terminale then
3          return color * valore euristico
4      value := -infinity
5      for each child of node do
6          value := max(value, -negamax(child, depth-1, -color))
7      return value

```

2.2 Zobrist Hashing

Prima di parlare della *tabella di trasposizione* è necessario capire la funzione di hashing utilizzata. L'**hashing di Zobrist** (2) inizia generando casualmente interi di tipo *long*, per ogni possibile elemento posizionabile nella nostra tabella di trasposizione. In seguito, ogni volta che una cella verrà *marcata* o *demarcata* andremo a *ricalcolare* il valore di hashing effettuando una operazione di **xor** tra la cella marcata della *board* e il valore di *hashing random* generato precedentemente.

2.2.1 Pseudocodice Zobrist Hashing

```
1 function init_zobrist():
2     # riempimento con numeri random
3     table := a 2-d array
4     for each cell on the board :
5         table[i][j] := random_long()
6
7 function findhash(board):
8     hash = 0
9     for each cell on the board :
10        if cell is not empty :
11            piece = board[cell]
12            # esegue lo xor
13            hash ^= table[cell][piece]
14    return hash
```

2.3 Tabella di trasposizione

La tabella di trasposizione (3) è un **database** che memorizza i risultati delle ricerche eseguite in precedenza. È un modo per ridurre notevolmente lo spazio di ricerca di un albero di gioco con scarso impatto negativo.

Normalmente gli algoritmi risolutivi per i giochi finiti, come nel nostro caso, utilizzano metodi di **forza-bruta**, dove vengono analizzate tutte le possibili mosse durante la loro ricerca.

In questo modo vengono incontrate più e più volte le stesse posizioni, ma da sequenze di mosse differenti. Le tabelle di trasposizione quindi, **limitano** queste inutili **iterazioni**.

Per salvare i valori all'interno della tabella è necessario utilizzare una funzione di **hashing**. Le funzioni *hash* convertono le posizioni in un numero scalare quasi unico, consentendo un rapido calcolo dell'indice e una verifica delle posizioni memorizzate.

Esistono diverse tecniche di hashing le più diffuse sono:

- Zobrist Hashing
- BCH Hashing

Nel nostro algoritmo è stato utilizzando l'hashing di **Zobrist** aggiornando in modo incrementale il valore durante la creazione e l'annullamento della mossa. Una volta calcolato il valore di hashing andremo a inserire nella tabella di trasposizione i seguenti dati da memorizzare:

1. Profondità di ricerca
2. Migliore mossa trovata
3. Valore della migliore mossa trovata
4. Tipo di valore trovato

2.4 Iterative Deepening

Un notevole miglioramento che permette la scelta di una mossa migliore è dato dall'*iterative deepening* (4), il quale inizia con una ricerca in un solo strato, e successivamente **incrementa** la **profondità** di ricerca ed esegue un'altra ricerca.

Questo processo viene ripetuto fino all'esaurimento del **tempo** assegnato.

In caso di *ricerca incompleta*, avremo sempre la possibilità di ripiegare sulla mossa selezionata nell'ultima iterazione della ricerca.

Ogni qualvolta una nuova mossa verrà trovata, essa andrà a **sovrascrivere** la precedente ormai diventata superflua.

In questo modo possono essere accettati anche i risultati delle **ricerche parziali**, anche se, in caso di un forte calo del punteggio euristico ottenuto è opportuno dedicare un po' di tempo in più, poiché la prima alternativa è spesso una cattiva mossa.

2.4.1 Pseudocodice Iterative Deepening

```
1 function IterativeDeepening(root , goal){ #profondita max
2   for(profondita = 1; root!=goal; profondita++){
3     root = DLS(root , goal , profondita)
4   }
5
6   function DLS(nodo , goal , profondita){
7     if(profondita >= 0):
8       if(nodo == goal): return nodo
9       foreach(child in visita(nodo)):
10         DLS(child , goal , profondita -1)
11   }
```

2.5 Valutazione euristica

TODO

Riduzione del tempo di esecuzione nel caso in cui sia presente una mossa vincente per l'AI o per l'avversario, **evitando** così di eseguire **l'intero algoritmo** alpha-beta pruning.

2.5.1 Pseudocodice markWinCell:

```
1 function markMyWinningCell(MNKBoard B) {
2   for each freeCell do
3     if(markCell(i,j) == myWin)
4       return cell
5 }
6
7 function markEnemyWinningCell(MNKBoard B) {
8   markCell(i,j)
9   for each freeCell do
10    if(markCell(i,j) == yourWin)
11      B.unmarkCell()
12      B.unmarkCell()
13      B.markCell(i,j)
14      return cell(i,j)
15   else
16     B.unmarkCell()
17 }
```

Costo computazionale: $\Theta(n)$ dove n indica il numero di **celle libere** nella board.

2.6 Possibili euristiche precedentemente applicate

A livello di strategie di implementazione, oltre all'**euristica** citata precedentemente, sono state implementate anche altri metodi con scarsi risultati:

1. Ricavare e analizzare solo le celle vicine a **distanza** minore-uguale a **K**.

2.6.1 Pseudocodice $\leq K$

```
1 function removeBadMoves(MNKBoard B) {
2   for each markedCell do
3     for each k-distance Free-cell do
4       if (i+k < B.M) add(i+k, j)
5       if (j+k < B.N) add(i, j+k)
6       if (i-k >= 0) add(i-k, j)
7       if (j-k >= 0) add(i, j-k)
8       if (i+k < B.M and j+k < B.N) add(i+k, j+k)
9       if (i+k < B.M and j-k >= 0) add(i+k, j-k)
10      if (i-k >= 0 and j+k < B.N) add(i-k, j+k)
11      if (i-k >= 0 and j-k >= 0) add(i-k, j-k)
12   return possibleValue;
13 }
```

Costo computazionale: $O(m \cdot k)$ dove m indica il numero di **celle** **marcate** e k indica il numero di **celle** allineate per **vincere**.

2. scegliere una mossa **vicina** a una cella avversaria nel caso in cui non si riesca a ricavare una mossa "vincente". (in questo modo con configurazioni m, n, k di grandi dimensione abbiamo più **probabilità** di selezionare una cella vincente.

2.6.2 Pseudocodice nearCell

```
1 function getBestMoves(MNKBoard B) {
2   for each freeCell do
3     if (cellState(i+1, j) != FREE) add(i, j);
4     if (cellState(i, j+1) != FREE) add(i, j);
5     if (cellState(i-1, j) != FREE) add(i, j);
6     if (cellState(i+1, j+1) != FREE) add(i, j);
7     if (cellState(i+1, j-1) != FREE) add(i, j);
8     if (cellState(i-1, j+1) != FREE) add(i, j);
9     if (cellState(i-1, j-1) != FREE) add(i, j);
10    if (cellState(i, j-1) != FREE) add(i, j);
11   return singolCellUseful;
12 }
```


Costo computazionale: $\Theta(n)$ dove n indica il numero di **celle libere** nella board.

3 Conclusioni

In conclusione gli algoritmi applicati hanno dato un riscontro positivo sull'efficacia del player implementato. Dal punto di vista computazionale è stata data una stima appropriata dovuta alla tipologia di algoritmo, poiché non è possibile definire un numero massimo di nodi visitati.

3.1 Complessità computazionale

La complessità computazionale del metodo *selectCell()* deriva maggiormente dall'algoritmo *negamax* e *iterative deepening*. Negamax rispettivamente ha costo computazionale pari a $O(m^d)$ dove m sono le mosse per giocatore e d indicano la profondità della mossa, mentre iterative deepening ha un costo trascurabile di $O(d)$ dove d indica la profondità della soluzione più vicina alla radice.

La tabella di trasposizione implementata assieme alla funzione di hashing di Zobrist invece ha un costo costante.

Un altro fattore importante da osservare è l'euristica implementata, la quale viene chiamata ad ogni iterazione influenzando sul costo computazionale, pari a $O(???)$. La complessità totale dunque raggiunge un costo di $O(???)$.

3.2 Alternative strategie progettuali

Possibili miglioramenti/strategie applicabili possono migliorare ulteriormente l'efficacia di gioco del nostro giocatore. Metodi e strategie potrebbero essere:

- L'uso di una migliore euristica di gioco, come ad esempio le seguenti (5):
 - Euristica di **ABDOULAYE** dove viene implementato è gestito il concetto di **threat**.
 - Euristica di **Shevchenko**, dove vengono analizzate le celle presenti per righe, colonne e diagonali
 - Euristica di **Chua Hock Chuan**, in cui ci basiamo sugli allineamenti di entrambi i giocatori in tutte le direzioni.
- Adottare il metodo di **Monte Carlo** (6), dove andiamo a campionare casualmente delle combinazioni di tabelle per ottenere dei risultati numerici
- La **Ricerca quiescenza** (7) dove una volta effettuata la valutazione delle mosse, essa viene rinviata senza considerare le mosse future della posizione, fino a quando non è sufficientemente stabile per essere valutata.

References

- [1] <https://en.wikipedia.org/wiki/Negamax>
- [2] <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/>
- [3] https://www.chessprogramming.org/Transposition_Table
- [4] https://www.chessprogramming.org/Iterative_Deepening
- [5] <http://www.cari-info.org/Actes-2018/p276-286.pdf>
- [6] https://en.wikipedia.org/wiki/Monte_Carlo_method
- [7] https://en.wikipedia.org/wiki/Quiescence_search