MNKGame - Signora Carla

Simone Sanna Matricola 0000792616
Ilaria Palestini Matricola 0000788888

Alma Mater Studiorum - Università di Bologna

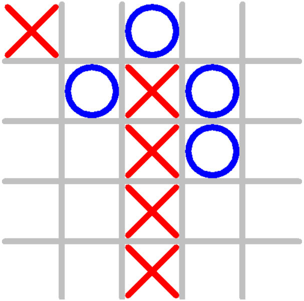
Presentazione Progetto di Algoritmi e Strutture Dati

Table of Contents

1. Introduzione
    - Il gioco
    - L'algoritmo
2. Sviluppo
    - Difficoltà incontrate
    - Scelte progettuali
3. Conclusioni
    - Analisi dei costi
    - Considerazioni
4. Riferimenti
    - Riferimenti

Il gioco

MNK consiste in una versione generalizzata del tris in una matrice di dimensione $M\ X\ N$, nella quale si devono allineare K simboli uguali.



- Esempio di matrice 5 X 5 con vittoria per 4 simboli

L‘algoritmo

L’algoritmo usato è chiamato *Iterative Deepening*, e serve per risolvere un GameTree, ovvero un albero che rappresenta tutte le possibili partite di un gioco a turni.

Viene implementata una *ricerca in ampiezza*, nella quale il numero totale di nodi è determinato da

$$\sum_{k=0}^{m} = \frac{m^{n+1} - 1}{m - 1}$$

dove m rappresenta il numero di mosse e n il numero massimo di turni possibili.

Table of Contents

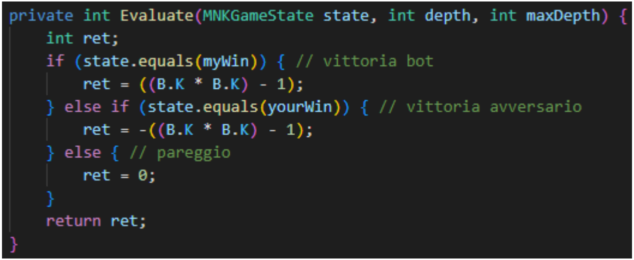
1. Introduzione
    - Il gioco
    - L'algoritmo
2. Sviluppo
    - Difficoltà incontrate
    - Scelte progettuali
3. Conclusioni
    - Analisi dei costi
    - Considerazioni
4. Riferimenti
    - Riferimenti

Difficoltà incontrate

- Il primo ostacolo incontrato, appena iniziato lo sviluppo del progetto, è stato l'approccio alla struttura del programma fornito.
- Una volta entrati nell'ottica delle classi e dei loro metodi, abbiamo analizzato attentamente le classi $\boxed{\text{RandomPlayer}}$ e $\boxed{\text{QuasiRandomPlayer}}$ per prendere spunto e iniziare così la progettazione.
- Il compito più impegnativo, però, è stato costruire una funzione euristica adatta al nostro caso per valutare le configurazioni non-finali. Dopo varie ricerche online, abbiamo preso spunto da "Developing a Memory Efficient Algorithm for Playing MNK games"$^{pag.19}$

Scelte progettuali

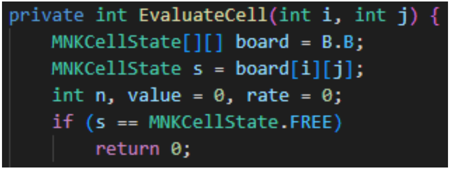
Il metodo `Evaluate` viene usato per assegnare un punteggio alle foglie dell'albero



```
private int Evaluate(MNKGameState state, int depth, int maxDepth) {
    int ret;
    if (state.equals(myWin)) { // vittoria bot
        ret = ((B.K * B.K) - 1);
    } else if (state.equals(yourWin)) { // vittoria avversario
        ret = -((B.K * B.K) - 1);
    } else { // pareggio
        ret = 0;
    }
    return ret;
}
```

In caso di vittoria del giocatore ha come valore di return il massimo assegnabile, in caso di vittoria dell'avversario il minimo, e in caso di pareggio 0.

Abbiamo utilizzato il metodo EvaluateCell per decidere quali fossero le celle migliori da marcare.

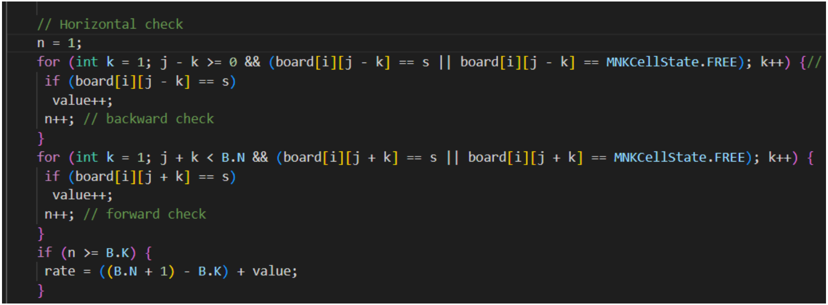


```
private int EvaluateCell(int i, int j) {
    MNKCellState[][] board = B.B;
    MNKCellState s = board[i][j];
    int n, value = 0, rate = 0;
    if (s == MNKCellState.FREE)
        return 0;
```

Il metodo controlla quante celle adiacenti a una cella marcata siano vuote o marcate dal giocatore stesso, e assegna un punteggio in base a quante celle sono disponibili per la vittoria.

Per prima cosa inizializziamo il punteggio a 0, e successivamente iniziamo il conteggio delle celle consecutive nell'intorno della cella del giocatore, in tutte le direzioni possibili.

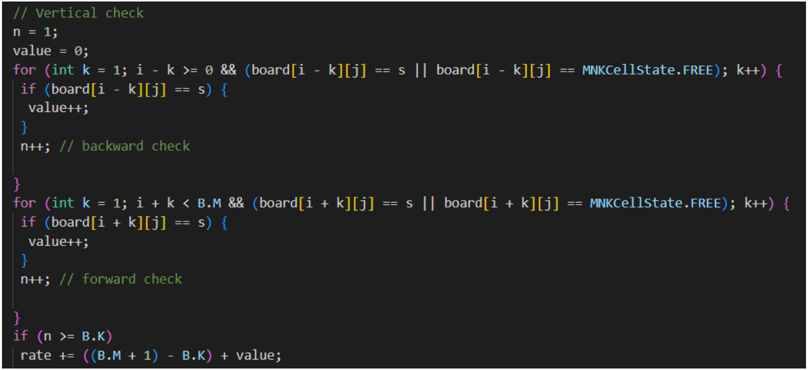
- Controllo orizzontale



```
// Horizontal check
n = 1;
for (int k = 1; j - k >= 0 && (board[i][j - k] == s || board[i][j - k] == MNKCellState.FREE); k++) {//
 if (board[i][j - k] == s)
  value++;
 n++; // backward check
}
for (int k = 1; j + k < B.N && (board[i][j + k] == s || board[i][j + k] == MNKCellState.FREE); k++) {
 if (board[i][j + k] == s)
  value++;
 n++; // forward check
}
if (n >= B.K) {
 rate = ((B.N + 1) - B.K) + value;
}
```

Partendo dalla cella selezionata, controlla le celle precedenti e successive, fermandosi all'inizio o alla fine della matrice. (oppure nel caso in cui venga trovata una cella marcata dall'avversario)

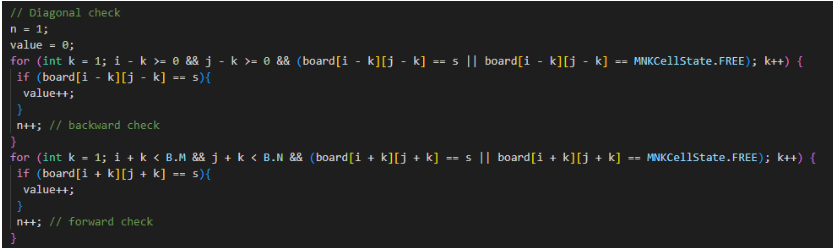
- Controllo verticale



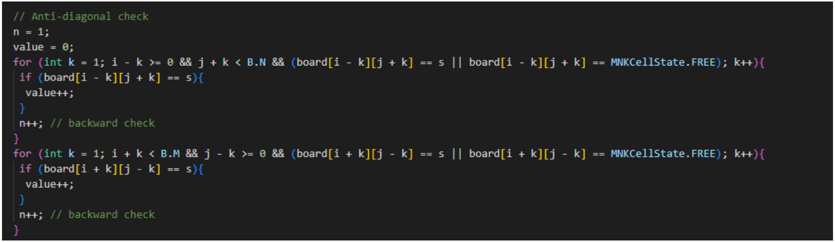
```
// Vertical check
n = 1;
value = 0;
for (int k = 1; i - k >= 0 && (board[i - k][j] == s || board[i - k][j] == MNKCellState.FREE); k++) {
  if (board[i - k][j] == s) {
    value++;
  }
  n++; // backward check

}
for (int k = 1; i + k < B.M && (board[i + k][j] == s || board[i + k][j] == MNKCellState.FREE); k++) {
  if (board[i + k][j] == s) {
    value++;
  }
  n++; // forward check

}
if (n >= B.K)
  rate += ((B.M + 1) - B.K) + value;
```

Controlla le celle sopra e sotto quella selezionata, fermandosi ai limiti della griglia.

- Controlli diagonali



```
// Diagonal check
n = 1;
value = 0;
for (int k = 1; i - k >= 0 && j - k >= 0 && (board[i - k][j - k] == s || board[i - k][j - k] == MNKCellState.FREE); k++) {
 if (board[i - k][j - k] == s){
  value++;
 }
 n++; // backward check
}
for (int k = 1; i + k < B.M && j + k < B.N && (board[i + k][j + k] == s || board[i + k][j + k] == MNKCellState.FREE); k++) {
 if (board[i + k][j + k] == s){
  value++;
 }
 n++; // forward check
}
```

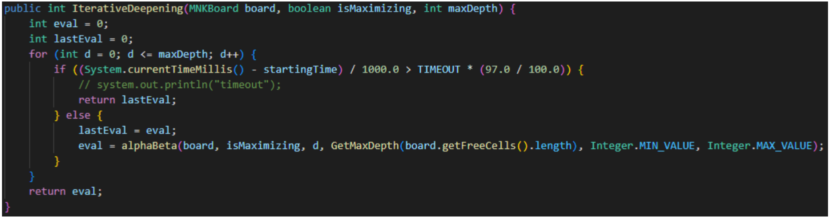


```
// Anti-diagonal check
n = 1;
value = 0;
for (int k = 1; i - k >= 0 && j + k < B.N && (board[i - k][j + k] == s || board[i - k][j + k] == MNKCellState.FREE); k++){
 if (board[i - k][j + k] == s){
  value++;
 }
 n++; // backward check
}
for (int k = 1; i + k < B.M && j - k >= 0 && (board[i + k][j - k] == s || board[i + k][j - k] == MNKCellState.FREE); k++){
 if (board[i + k][j - k] == s){
  value++;
 }
 n++; // backward check
}
```

Check in entrambe le direzioni, anche qua il metodo verifica quante caselle sono libere o a nostro favore.

Il parametro $\boxed{\text{value}}$ incrementa per ogni cella occupata dal giocatore nel caso in cui questa faccia parte di una possibile configurazione vincente.
Dopo ogni controllo direzionale viene invece incrementato il valore $\boxed{\text{n}}$ per ogni cella non occupata dall'avversario.
Il controllo finale $\boxed{\text{n} \geq B.K}$ ci dice se, per quella cella, è possibile raggiungere uno stato di vittoria mettendo B.K simboli in successione.

Il metodo IterativeDeepening visita in ampiezza un sottoalbero.



```java
public int IterativeDeepening(MNKBoard board, boolean isMaximizing, int maxDepth) {
    int eval = 0;
    int lastEval = 0;
    for (int d = 0; d <= maxDepth; d++) {
        if ((System.currentTimeMillis() - startingTime) / 1000.0 > TIMEOUT * (97.0 / 100.0)) {
            // system.out.println("timeout");
            return lastEval;
        } else {
            lastEval = eval;
            eval = alphaBeta(board, isMaximizing, d, GetMaxDepth(board.getFreeCells().length), Integer.MIN_VALUE, Integer.MAX_VALUE);
        }
    }
    return eval;
}
```

Gli viene passata la profondità massima a cui vogliamo arrivare, che poi cerca di raggiungere entro il tempo limite. Nel caso in cui raggiungiamo il timeout prima di arrivare alla fine del sottoalbero, utilizza i risultati trovati alla profondità precedente.

Table of Contents

1 Introduzione
- Il gioco
- L'algoritmo

2 Sviluppo
- Difficoltà incontrate
- Scelte progettuali

3 Conclusioni
- Analisi dei costi
- Considerazioni

4 Riferimenti
- Riferimenti

Analisi dei costi

Per quanto riguarda l'analisi dei costi:

- In termini di memoria l'algoritmo Iterative Deepening dipende dal costo di Alpha Beta, nel caso pessimo, comuqnue, abbiamo O(maxDepth), dove $\boxed{\text{maxDepth}}$ è la profondità di ricerca.
- In termini di tempo, nel caso di gioco con $\boxed{\text{m}}$ mosse e $\boxed{\text{n}}$ turni, limitato a $\boxed{\text{d}}$ turni

$$\begin{aligned} T(m,d) &\leq (d+1) + dm + (d-1)m^2 + \cdots + 2m^{d-1} + m^d = \\ &= m^d \left(1 + 2m^{-1} + \cdots + dm^{-d+1} + (d+1)m^{-d}\right) \\ &= m^d \sum_{i=0}^{d} i \frac{1}{m^i} \leq m^d \sum_{i=0}^{\infty} i \frac{1}{m^i} = m^d \frac{1/m}{(1-1/m)^2} = O(m^d) \end{aligned}$$

Considerazioni

L'algoritmo Iterative Deepening è leggermente più lento rispetto ad AlphaBeta ma ha un controllo maggiore sulla visita del Game Tree quando limitato temporalmente.

- Come possiamo migliorare Signora Carla?
    - L'utilizzo di hashMap per salvare configurazioni già valutate portebbere un guadagno di tempo non indifferente in quanto molti dei sotto-alberi vengono ripetuti e rivalutati. Questa soluzione permetterebbe, con un controllo, di verificare la presenza in memoria di configurazioni già salvate ed utilizzare la loro valutazione senza rivisitare il sottoalbero.

Table of Contents

1. Introduzione
   - Il gioco
   - L'algoritmo
2. Sviluppo
   - Difficoltà incontrate
   - Scelte progettuali
3. Conclusioni
   - Analisi dei costi
   - Considerazioni
4. Riferimenti
   - Riferimenti

Riferimenti

- Developing a Memory Efficient Algorithm for Playing MNK games
- Alpha–beta pruning - Wikipedia
- Iterative deepening - Wikipedia