Relazione Progetto Algoritmi

Koci Erik

September 2021

1 Intestazione

Nome del progetto: Progetto ASD A.A. 2020-2021 (M,N,K)-game
cognome e nome autore: Koci Erik
Matricola: 0000997662

2 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

3 Scelte progettuali

Nello sviluppo progettuale del metodo **selectCell()** inizialmente ho utilizzato l'algoritmo noto di ricerca **alpha-beta pruning**, il quale riduce notevolmente il numero di mosse da valutare nel gioco in questione. Questa tipologia di algoritmo viene spesso utilizzata nei giochi a turni tra due o più giocatori.

3.1 Funzionamento dell'algoritmo

L'algoritmo **alpha-beta pruning** si basa principalmente su due valori, **alpha** e **beta**, i quali in ogni punto dell'albero, rappresentano la posizione migliore e peggiore che è possibile raggiungere. Se **A** è il giocatore **massimizzante** e **B** il giocatore **minimizzante** accade che:

- α è il punteggio minimo che **A** può raggiungere, a partire dalla posizione in esame; all'inizio dell'algoritmo viene posto a -∞. Durante il calcolo, α coincide con il valore della **migliore mossa** possibile attualmente calcolata per **A**.
- β è il punteggio massimo che **B** può raggiungere a partire dalla stessa posizione; all'inizio dell'algoritmo viene posto a $+\infty$. Durante il calcolo, β coincide con il valore della **migliore mossa** possibile attualmente calcolata per **B**.

Se durante la ricerca, per un dato nodo α diventa **maggiore** di β, la **ricerca** al di sotto di quel nodo **cessa** e il programma passa ad un altro sottoalbero perché da quella posizione in poi, A **perderebbe** anche se giocasse per vincere. [1] Una volta terminata una eseguzione di gioco, la funzione `evaluate` determinerà il punteggio da attribuire a seconda del `game state`.

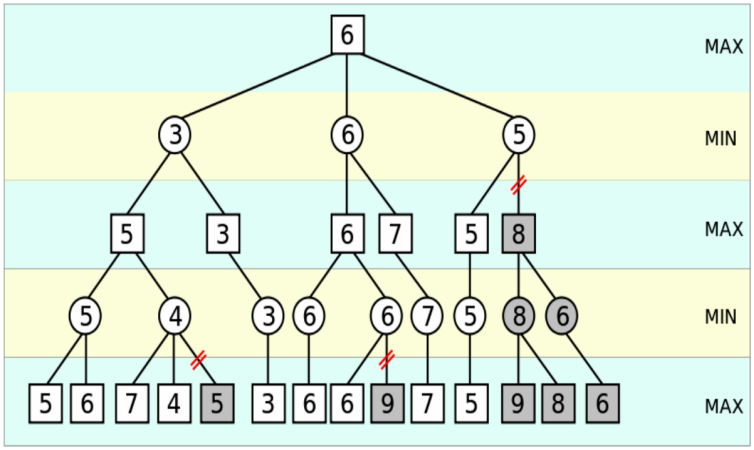

Figure 2: alpha-beta Pruning

3.2 Complessità computazionale

Con un fattore di ramificazione **b** e una profondità di ricerca di **d** strati, quando l'ordine di spostamento è **pessimo** la complessità computazionale è $O(b \cdot b \cdot ... \cdot b) = O(b^d)$.
Se l'ordine delle mosse per la ricerca è **ottimale**, il numero di posizioni del nodo foglia valutate è circa $O(b \cdot 1 \cdot b \cdot 1 \cdot ... \cdot b)$ per profondità dispari è $O(b \cdot 1 \cdot b \cdot 1 \cdot ... \cdot 1)$ per una profondità uniforme $O(b^{d/2}) = O(\sqrt{b^d})$.
In quest'ultimo caso, dove la tela di una ricerca è pari, il fattore di ramificazione effettivo è ridotto alla sua **radice quadrata**. [2]

Worst-case performance: $O(b^d)$
Best-case performance: $O(\sqrt{b^d})$

3.2.1 Pseudocodice:

```
function alphabeta(node, depth, a, b, maximizingPlayer) {
 if depth = 0 or node is a terminal node then
   return the heuristic value evaluate(value)
   if maximizingPlayer then
     value := -infinity
     for each goodMoves do
       value := max(value, alphabeta(child, depth-1,a,b,
           false))
       a := max(a, value)
       if value >= b then
         break (* b cutoff *)
         return value
    else
      value := +infinity
      for each goodMoves do
        value := min(value, alphabeta(child, depth-1,a,b,
            true))
        b := min(b, value)
        if value <= a then
          break (* a cutoff *)
      return value
}
```

```
function evaluate(MNKBoard B) {
    if(state == OPEN) return 1;
    else if(state == DRAW) return 0;
    else if(state == myWin) return 10;
    else return -10;
}
```

3.3 strategie nell'implementazione

A livello di strategie di implementazione, oltre all'**euristica** dell'algoritmo citato precedentemento, ho utilizzato anche altre tre funzioni, le quali ci permettono di:

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

3.3.1 Pseudocodice:

```
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 ho più **probabilità** di selezionare una cella vincente.

3.3.2 Pseudocodice:

```
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. Riduzione del tempo di eseguzione 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.

3.3.3 Pseudocodice:

```
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.

3.4 Ulteriori osservazioni

Nell'implementazione della mia **'AI'** ho inserito degli `if statements` in particolari condizioni di gioco, questo per rendere più efficiente l'algoritmo in certe **situazioni di gioco**.
A seconda della grandezza della board inoltre, viene effettuata una esplorazione a **profondità differente** dell'albero di gioco, questo per rendere più **rapida l'esecuzione dell'algoritmo**, a scapito della mossa migliore.

References

[1] Wikipedia. *Alpha–beta pruning.*

[2] Wikipedia. *Potatura alfa-beta - Alpha–beta pruning.*