ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ALGORITMO ALPHABETAPRO

Progetto di ASD 2021/2022

Gabellini Luca (0001020370)

Largura Andrea (0001032111)

Sagliano Giacomo (0000890256)

1 - Introduzione al problema

1.1 – (M, N, K) - game

(M, N, K) - game è un gioco in cui due giocatori si alternano nel selezionare una cella libera in una griglia di dimensione M x N, con l'obiettivo di allineare K simboli orizzontalmente, verticalmente o diagonalmente per vincere. Se la griglia viene completamente riempita prima che uno dei due riesca a vincere, allora la partita si conclude in parità.

1.2 – Punteggi

Si utilizza il seguente schema di punteggi per ogni partita giocata:

- 3 punti in caso di vittoria come secondo giocatore, ma non a tavolino;
- 2 punti in caso di vittoria come primo giocatore o a tavolino;
- 1 punto in caso di pareggio;
- 0 punti in caso di sconfitta.

La vittoria a tavolino avviene in presenza di errori dell'avversario (ad esempio: time-out, mossa non legale, etc.).

1.3 – Obiettivo

Lo scopo del progetto è quello di sviluppare un algoritmo efficiente in *Java* che simuli il comportamento di un giocatore reale e che ottenga il miglior punteggio possibile sulle seguenti configurazioni di gioco:

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	?

Figura 1: configurazioni del (M, N, K) - game

1.4 – Il package *mnkgame*

Al fine di sviluppare l'algoritmo, sono state fornite delle classi ausiliarie che definiscono l'interfaccia del gioco ed il suo funzionamento. In particolare, la realizzazione del giocatore si basa sull'implementazione dell'interfaccia *MNKPlayer*, contenente un metodo per la selezione delle mosse e uno di inizializzazione del giocatore.

2 – Scelte progettuali

2.1 – Classi e criteri di gioco utilizzati

AlphaBetaPro è progettato attorno ad un sistema di generazione e valutazione di strategie di gioco. Per illustrare questo meccanismo, ci avvaliamo di alcuni esempi:

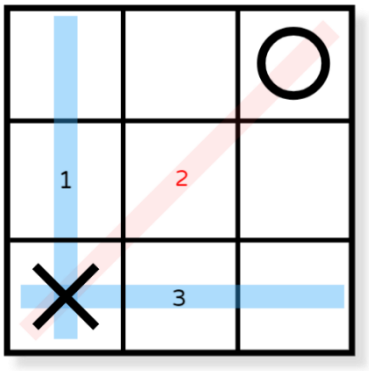

Figura 2: *generazione di strategie in un esempio di partita a configurazione (3, 3, 3). Si noti la 2 invalidata.*

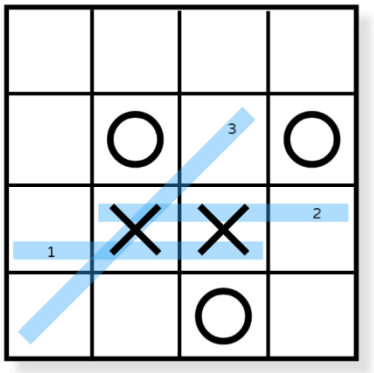

Figura 3: *generazione di strategie in un esempio di partita a configurazione (4, 4, 3).*

Ognuno dei due giocatori memorizza le proprie strategie all'interno di una lista. Nel momento in cui una strategia viene bloccata dall'avversario *(fig.2)*, questa viene invalidata e rimossa dalla lista.

Ad ogni cella è assegnata una priorità, calcolata in funzione del numero di strategie (di entrambi i giocatori) che la attraversano. Celle ad alta priorità sono quelle che contemporaneamente avvicinano il giocatore alla vittoria e bloccano possibili strategie vincenti dell'avversario. L'algoritmo sfrutta questo parametro per scegliere un ordine il più vantaggioso possibile di valutazione della qualità delle mosse.

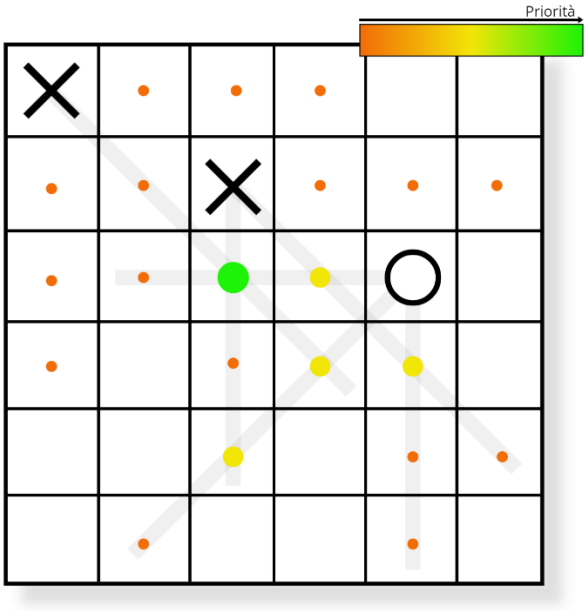

Figura 4: esempio di assegnamento di priorità alle celle in una configurazione (6, 6, 4). Sono evidenziate le strategie che portano alcune mosse a crescere di priorità.

Per implementare il criterio di gioco dato sopra, si fa uso delle seguenti classi ausiliarie:

- *MNKStrategy*: rappresenta una potenziale strategia da seguire a partire da una certa mossa, costituita da *K* celle;
- *StrategySet*: è l'insieme delle *MNKStrategy* associate ad uno specifico giocatore. L'algoritmo mantiene, quindi, due StrategySet: uno per il giocatore che massimizza e uno per quello che minimizza: entrambi sono aggiornati ad ogni mossa scelta;
- *MovesQueue*: implementazione personalizzata di una coda di priorità, in cui vengono memorizzate, secondo il criterio di priorità dato sopra, le possibili mosse da giocare.

2.2 – *Alpha-beta pruning* con memoria

Alpha-beta pruning è un algoritmo ricorsivo per individuare la migliore mossa in un gioco secondo il criterio di minimizzare la massima perdita possibile.

La partita viene rappresentata mediante una struttura ad albero, in cui ogni nodo rappresenta una configurazione di gioco, e le foglie rappresentano una partita conclusa.

L'algoritmo è esatto quando è possibile visitare l'intero *game tree*, riuscendo così a trovare la mossa migliore in ogni situazione. Nel momento in cui si trova una mossa ottima (ovvero non è possibile far di meglio) interviene la potatura *alfa-beta*. Essa ha il beneficio di ridurre drasticamente il numero di nodi visitato dall'algoritmo, andando figurativamente a "potare" rami superflui.

La versione proposta di *Alpha-beta pruning* adotta un sistema di memoria implementato mediante tabelle di trasposizione, che permettono di salvare la valutazione di configurazioni già visitate, riducendo così ulteriormente le valutazioni superflue e offrendo un discreto miglioramento in termini di prestazioni, seppur non sufficiente a permettere di giocare in maniera ottimale nelle configurazioni più grandi di *(M, N, K) – game*.

Oltre a quanto detto, *Alpha-beta Pro* differisce dall'originale perché impone un ordinamento ai figli di un certo nodo del *game tree* secondo la coda con priorità vista sopra. Poiché si assume che l'algoritmo abbia già trovato una mossa ottimale, le mosse fuori dalla coda non vengono nemmeno visitate.

Alpha-beta pruning viene chiamato dalla funzione *Iterative Deepening*: questa permette di eseguire sul game tree una visita in ampiezza che scende gradualmente di profondità e può essere interrotta in qualunque momento allo scadere del tempo. Se questo accade, *Alpha-beta Pro* ritorna semplicemente la mossa migliore trovata fino a quel punto.

Algorithm $AlphaBetaPro(node, \alpha, \beta, depth, maxPlayer)$

```
1: ttEntry ← TranspositionTable.get(node)
2: if ttEntry.valid then
3:     return ttEntry.value
4: end if
5: if depth = 0 or node.isLeaf() then
6:     return eval(node)
7: else if maxPlayer then
8:     value ← +∞
9:     for c ∈ Q.moves() do
10:        Strategies.update(c)
11:        value ← max(value, AlphaBetaPruning(c, α, β, depth − 1, false))
12:        α ← max(value, α)
13:        if α ≥ β then
14:            break                                          ▷ Cutoff
15:        end if
16:    end for
17: else
18:    value ← −∞
19:    for c ∈ Q.moves() do
20:        Strategies.update(c)
21:        value ← min(value, AlphaBetaPruning(c, α, β, depth − 1, true))
22:        β ← min(value, β)
23:        if α ≥ β then
24:            break                                          ▷ Cutoff
25:        end if
26:    end for
27: end if
28: TranspositionTable.put(node)
29: return value
```

Figura 5: pseudocodice della variante di Alpha-beta pruning implementata

2.3 – Metodo *selectCell( )*

Come prima cosa, il metodo controlla la dimensione della matrice di gioco: se questa è inferiore a 4 x 4 allora viene utilizzata la versione standard di *Alpha-beta pruning*. In queste configurazioni, infatti, l'algoritmo si comporta in maniera esatta.

Nel caso di matrici più grandi e, se *AlphaBetaPro* è il primo giocatore, la prima mossa viene posizionata nel centro della matrice. Questo permette di massimizzare il numero di strategie generate. Qualora, invece, si cominci come secondo, la mossa è posizionata diagonalmente rispetto a quella dell'avversario. Per colmare lo svantaggio dovuto al cominciare secondi, si cerca di ostacolare il più possibile le strategie dell'avversario.

Nei turni successivi, *selectCell( )* effettua alcuni controlli sullo stato attuale del set di strategie per individuare eventuali mosse vincenti. In caso contrario, viene eseguito *AlphaBetaPro* attraverso *iterativeDeepening( ).*

Algorithm $SelectCell(MarkedCells, FreeCells)$

```
1:  B ← local MNKBoard
2:  B ← B + last opponent move
3:  if FreeCells.length = 1 then
4:      return FreeCells[0]
5:  end if
6:  if B.size ≤ 16 then
7:      AlphaBetaPruning(B, α, β, false)
8:  else
9:      if MarkedCells.length = 0 then
10:         Play in the middle of the board
11:     end if
12:     if one-move win/loss then
13:         c ← winning/losing move
14:         return c
15:     end if
16:     if MarkedCells.length = 1 or 2 then
17:         Play along the diagonal
18:     end if
19:     for move ∈ Q.moves() do
20:         Strategies.update(move)
21:         evaluateStrategies()
22:     end for
23:     c ← IterativeDeepening(B)
24:     return c
25: end if
```

Figura 6: implementazione in pseudocodice del metodo SelectCell( )

2.4 – Strutture dati utilizzate

- *Array List*: utilizzata nelle seguenti classi:
  - MNKStrategy: memorizza come array le celle della strategia;
  - StrategySet: memorizza le MNKStrategy valide;

  È stata utilizzata principalmente per l'efficacia nell'inserimento (in tempo O(1)) e facilità di iterazione sui suoi elementi.
- *Stack*: utilizzata come struttura d'appoggio dalla classe *StrategySet* per interfacciarsi all'algoritmo *Alpha-beta pruning*.
- *Hash Table*: utilizzata nelle seguenti classi:
  - *AlphaBetaPro*: la tabella di trasposizione è implementata attraverso una tabella hash. La funzione di hash utilizzata è detta *zobrist hash*. Questa tecnica è molto comune negli algoritmi basati su game tree per la sua estrema efficacia;
  - *MovesQueue*: la coda con priorità utilizza due tabelle hash distinte. La prima *hTable* permette di verificare in tempo costante se una cella è presente o meno nella coda. La seconda *pTable* memorizza la priorità di ogni cella della matrice di gioco.
- *Priority Queue*: la classe *MovesQueue* è largamente basata su una coda con priorità.

3 – Analisi del costo computazionale

3.1 – Algoritmi noti

Alpha-beta pruning ha un costo pari a $O(m^d)$, dove *m* è il fattore di branching (ovvero il numero di possibili scelte di mosse disponibili) e *d* la profondità attuale: tuttavia, nel caso di ordinamento ottimo delle mosse, il costo scende fino a $O(\sqrt{m^d})$ grazie alle frequenti operazioni di potatura.
Iterative-deepening ha un costo triviale pari a $O(d)$, dove *d* è la profondità di ricerca.
La tabella di trasposizione con *zobrist hashing* ha costo pari a $O(1)$ sia nelle operazioni di deposito che per la consultazione della tabella.

3.2 – Euristica

Il costo dell'euristica utilizzata è più complesso da quantificare.
In generale, questo dipende dalla dimensione di *StrategySet*, poiché le operazioni sulle strategie sono al più lineari su di essa.
I metodi di *MovesQueue* sono a loro volta lineari sulla dimensione della coda con priorità (la quale a sua volta è legata alla dimensione di *StrategySet*).

Attraverso una serie di test sperimentali, è emerso che la dimensione di *StrategySet* cresce con andamento all'incirca logaritmico rispetto alla tabella di gioco, e in maniera inversamente proporzionale a K. Riassumendo, il costo aggiunto dell'euristica è quindi formulabile come:

$$O(log(\frac{M \cdot N}{K}))$$

4 – Fonti

- https://en.wikipedia.org/wiki/Alpha-beta_pruning
- https://levelup.gitconnected.com/zobrist-hashing-305c6c3c54d0
- https://www.chessprogramming.org/Transposition_Table
- https://www.chessprogramming.org/Zobrist_Hashing