Relazione per il progetto del corso
Algoritmi e Strutture di Dati a.a. 2022/23

Membri del gruppo:
- Enrico Ferraiolo 0001020354

Descrizione del problema:
Ci troviamo a sfidare un giocatore in una partita di Forza 4 generalizzata, ovvero in un tabellone composto da M righe, N colonne e X gettoni da allineare.
Il problema richiede di creare un giocatore software il più *capace* possibile a scegliere la mossa migliore dato uno stato di gioco, ovvero:
ci troviamo ad affrontare l'avversario durante tutta la partita e per ogni sua mossa dobbiamo controbattere con la migliore mossa possibile, aspirando alla vittoria, o se essa non è possibile alla patta.

Il problema non è banale, il numero di stati di gioco cresce in modo esponenziale all'aumentare delle variabili M,N,X, ovvero alla dimensione della matrice di gioco e dei gettoni da allineare.

Inoltre, un altro fattore cruciale per trovare la mossa migliore è il tempo a noi disponibile per giocare, difatti ci occupiamo di trovare la miglior soluzione in un tempo accettabile fissato a 10 secondi.

Scelte progettuali:
Il problema ha una struttura ottima per implementare una risoluzione con un algoritmo di decisione su alberi per un gioco a turni, ovvero il Minimax.
Difatti per poter usare il Minimax abbiamo bisogno di un albero che andrà a rappresentare tutti i possibili stati di gioco a cui possiamo arrivare, tale albero è anche chiamato Game Tree.
Ho scelto di implementare Minimax poiché esso minimizza la massima perdita possibile.
C'è da notare una cosa: tale algoritmo è esatto quando possiamo visitare l'intero albero di gioco, cosa che però diventa praticamente impossibile al minimo aumento di una delle variabili M,N o X.

Il Minimax ha costo computazionale pari a:
$O(m^n),\ m = numero\ di\ mosse\ possibili,\ n = turni$

Per comodità, scambieremo i turni con la profondità, quindi la formula diventa:
$O(m^d),\ m = numero\ di\ mosse\ possibili,\ d = profondità$

Tale algoritmo si comporta estremamente bene su partite con M,N,X molto piccoli, poiché fa in modo di riuscire a visitare l'intero albero in un tempo molto breve, ma il problema è un altro: come facciamo a far funzionare Minimax nel miglior modo possibile all'aumentare di M,N,X, ovvero quando non è possibile visitare l'intero Game Tree?

Per rispondere a questa domanda introduciamo due variabili α e β, esse saranno rispettivamente il punteggio minimo ottenibile dal giocatore A (massimizzante) e il punteggio massimo ottenibile dal giocatore B (minimizzante).

Se ad un certo punto dell'esecuzione del mio algoritmo dovesse accadere che $\beta \leq \alpha$ allora il sottoalbero, rispetto alla mia posizione nel Game Tree, non potrebbe contenere una soluzione migliore di quella già trovata.
Tale ottimizzazione si chiama AlphaBeta Pruning, difatti ciò che facciamo è potare uno o più rami del mio albero senza visitarli.

Notiamo però che nell'implementazione effettiva di Minimax con AlphaBeta Pruning ho anche aggiunto un limite alla profondità. Esso ci permette gestire alberi di gioco troppo grandi tramite un'euristica sugli stati di gioco che non siano finali. Inoltre limitare la profondità ci aiuta a migliorare la valutazione generale poiché in questo modo il nostro giocatore software riesce a decidere quale tra due mosse è quella che ci porta alla vittoria più velocemente, nonché ad una sconfitta più lenta.

Minimax con AlphaBeta Pruning ha il seguente costo computazionale:
$O(m^d)$ nel caso pessimo
$O(\sqrt{m^d})$ nel caso ottimo

Quindi l'ottimizzazione AlphaBeta ha il medesimo costo di Minimax, tranne nel caso ottimo. Difatti in tale situazione otteniamo uno speed-up quadratico, ciò ci permette di visitare nodi dell'albero ad una profondità doppia di quella precedente.

A questo punto rimangono due problemi in sospeso: il primo sta nel trovare una soluzione in un tempo ragionevole, il secondo nell'implementare un'euristica su stati di gioco non finali.

Partendo dal primo, ricordiamo che abbiamo un limite di tempo pari a 10 secondi per eseguire la nostra mossa.
Per risolvere tale problema introduciamo l'algoritmo Iterative Deepening. Esso ci permette di eseguire una visita in ampiezza del nostro sottoalbero radicato, fino ad una profondità d.
Dato un timeout (nel nostro caso di 10 secondi) e un nodo che stiamo valutando, è possibile affermare dove si trova la mossa migliore: se mentre eseguiamo la visita di tale nodo il tempo scade, allora la mossa migliore sarà a profondità $d - 1$.

Iterative Deepening ha il seguento costo computazionale:
$O(m^d)$
Esso è lo stesso di AlphaBeta poiché il numero di nodi a livello d domina l'intero costo.
Noteremo però che nel codice, Iterative Deepening, è più dispendioso di tempo rispetto ad AlphaBeta Pruning.

Ora dobbiamo risolvere il problema relativo all'euristica:
ho scelto di introdurre una valutazione degli stati di gioco in base al numero di gettoni allineati da ciascun giocatore all'interno del tabellone.
Ogni stato non finale ottiene la seguente valutazione:
$GettoniAllineatiMiei - GettoniAllineatiAvversario$ ciò mi permette di ottenere una valutazione a grandi linee di quello che sta succedendo nel nodo.

Inoltre valuto anche gli stati finali, ovvero patta, vittoria giocatore 1 e vittoria giocatore 2.

In caso di vittoria viene restituito:

- Vittoria mia: $ValoreVittoria$
- Vittoria avversario: $-\ ValoreVittoria$.

Similmente per le patte viene restituito:

- Se massimizzo: $ValorePatta$
- Se minimizzo: $-\ ValorePatta$

A questo punto possiamo leggere il codice e mettere insieme i vari dettagli per ottenere i seguenti costi computazionali per le funzioni:

- selectColumn:
  - $L\ *\ O(IterativeDeepening)\ +\ O(1),\ L\ =\ colonne\ disponibili\ nel\ mio\ turno$
- iterativeDeepening:
  - $O(checktime)\ +\ d\ *\ (O(timeIsRunningOut)\ +\ O(alphaBeta))\ +\ O(1),\ d\ =\ profondità$
- checktime:
  - $O(1)$
- timeIsRunningOut:
  - $O(1)$
- alphaBeta:
  - $O(evaluateBoard)\ +\ O(m^d),\ m\ =\ numero\ di\ mosse\ possibili,\ d\ =\ profondità$
- evaluateBoard:
  - $2\ *\ O(countAlignedCells)\ +\ O(1)$
- countAlignedCells:
  - $2\ *\ O(MN)\ +\ O(MNK)\ =\ O(MNK),\ K\ =\ min(M,N)$

Quindi

- selectColumn:
  - $L\ *\ O(dm^d)\ +\ O(1)\ =\ O(Ld\ *\ m^d)$
    $m\ =\ numero\ di\ mosse\ possibili,\ d\ =\ profondità,\ L\ =\ colonne\ disponibili\ nel\ mio\ turno$
- iterativeDeepening:
  - $O(1)\ +\ d\ *\ (O(1)\ +\ O(m^d))\ +\ O(1)\ =\ O(dm^d)$,
    $m\ =\ numero\ di\ mosse\ possibili,\ d\ =\ profondità$
- checktime:
  - $O(1)$
- timeIsRunningOut:
  - $O(1)$
- alphaBeta:
  - $O(MNK)\ +\ O(m^d)\ =\ O(m^d),\ m\ =\ numero\ di\ mosse\ possibili,\ d\ =\ profondità$
- evaluateBoard:
  - $2\ *\ O(MNK)\ +\ O(1)\ =\ O(MNK)$
- countAlignedCells:
  - $O(MNK),\ K\ =\ min(M,N)$

Quindi il costo computazionale finale sarà quello di selectColumn, ovvero:
$O(Ld * m^d)$,
$m = numero\ di\ mosse\ possibili,\ d = profondità,\ L = colonne\ disponibili\ nel\ mio\ turno$

Bibliografia:

- Slide del corso
- AlphaBeta Pruning
- Iterative Deepening