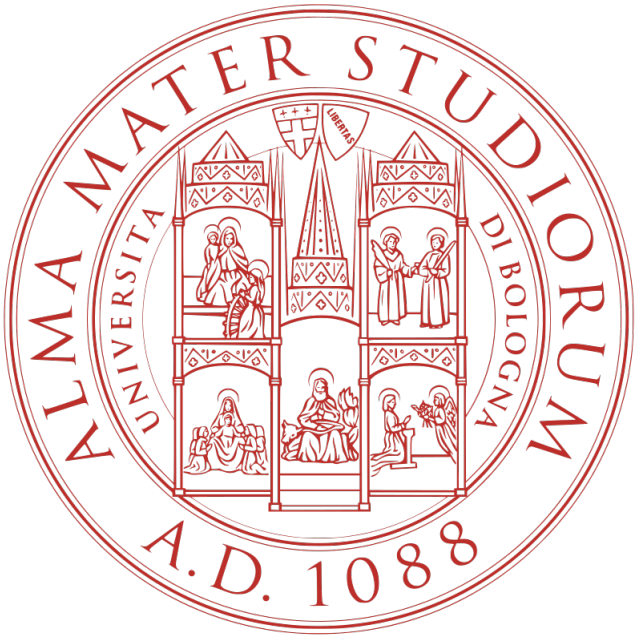
Progetto di Algoritmi e Strutture di Dati
Dipartimento di Informatica

Salvatore Lervini: 0001030390
Mattia Lodi: 0001030390
Federico Vallese: 0000983811

Settembre 2022



Indice

1 Descrizione del problema e obiettivi **3**
1.1 L'algoritmo iterative deepening 3

2 Scelte progettuali **4**
2.1 Implementazione di Iterative deepening 4
2.2 Funzione bestCells 4
2.3 evaluation e evaluate 4
2.4 Timeout 6

3 Costo computazionale **6**

1 Descrizione del problema e obiettivi

Il progetto consiste nella creazione di un giocatore virtuale capace di trovare la strategia ottima data una qualunque configurazione dell'$(M, N, K) - game$ in un limite di tempo prestabilito. L'$(M, N, K) - game$ è la generalizzazione del tris, nel quale i due giocatori, a turno, si sfidano in una matrice di dimensioni $M \times N$ al fine di allineare K simboli uguali orizzontalmente, verticalmente o diagonalmente.
Data la necessità intrinseca del gioco di valutare le possibili mosse future, sia proprie sia avversarie, in maniera efficiente in relazione al limite di tempo, abbiamo ritenuto che l'algoritmo migliore presente in letteratura fosse l'*iterative deepening*.

1.1 L'algoritmo iterative deepening

L'*iterative deepening* consiste nell'effettuare varie chiamate dell'algoritmo *AlphaBeta Pruning* con profondità crescente. Il principale miglioramento rispetto a quest'ultimo sta nel fatto che tale ricerca può essere interrotta in relazione al tempo disponibile, al termine del quale è restituita la mossa migliore trovata finora.
L'*AlphaBeta Pruning* è una versione migliorata dell'algoritmo *MiniMax*, un algoritmo ricorsivo che individua la migliore mossa possibile in un gioco secondo il criterio di *minimizzare la massima perdita possibile*. Questo criterio consiste nel valutare tutte le possibili continuazioni, e di queste scegliere la migliore per l'avversario in modo da poter rispondere efficacemente. Così facendo è possibile contrastare anche un giocatore con strategia ottima. Il primo giocatore massimizza e vince con -1, l'altro minimizza e vince con 1.
L'algoritmo *MiniMax* è troppo dispendioso di risorse, poiché visita sempre l'intero *game tree* (albero che contiene tutte le possibili partite in un gioco a turni), e ciò non è quasi mai possibile in tempi ragionevoli. Tale problema viene parzialmente risolto dall'*AlphaBeta Pruning* che, avvalendosi dei coefficienti α e β, riesce a "potare" in maniera opportuna alcuni rami dell'albero, riducendo quindi la complessità computazionale. I coefficienti α e β sono i parametri che memorizzano la valutazione della configurazione in base alla strategia di cui sopra. Aggiornando opportunamente tali coefficienti, quando α diventa maggiore o uguale a β non si ha più interesse nel visitare gli altri scenari, ciò comporta la suddetta potatura dell'albero.

2 Scelte progettuali

I nostri obiettivi principali sono stati:

- lavorare su un sottinsieme delle possibili mosse in modo da non dover valutare configurazioni inutili
- definire una funzione capace di valutare la configurazione corrente in relazione a quale giocatore si trovi in vantaggio, tenendo conto del numero massimo (opportunamente pesato) di simboli allineati.
- una gestione accurata delle tempistiche.

2.1 Implementazione di Iterative deepening

Per ridurre il numero di figli da analizzare creiamo l'array BC e lo riempiamo chiamando la funzione *bestCells*. Utilizziamo quindi due cicli, quello esterno agisce sulla profondità massima incrementandola, laddove quello interno applica l'algoritmo *AlphaBeta* alle configurazioni di BC, riempiendo parallelamente un array con i risultati ritornati da *AlphaBeta*, in modo che all' i-esimo valore dell'array corrisponda l'i-esima configurazione di BC.
In caso di possibilità di vittoria immediata si ritorna direttamente la mossa vincente evitando controlli inutili. Altrimenti, se il tempo sta finendo, grazie all'uso di iterative deepening, si ritorna la cella migliore tra quella trovata con l'ultima chiamata terminata e la migliore trovata finora nella chiamata corrente.
Una volta riempito l'array dei risultati, se il nostro giocatore è primo allora vincerà con -1, quindi selezioniamo la cella con valore minore; altrimenti, vincerà con $+1$ e in tal caso andremo quindi a considerare il maggiore di questi.

2.2 Funzione bestCells

La funzione *bestCells* riduce il numero di celle da dare in input a *iterative deepening*, che gioca un simbolo in ognuna di quelle celle, valuta la relativa configurazione e ritira tale mossa per procedere allo stesso modo con la successiva.
Per la valutazione di cui sopra, ci avvaliamo di un *array dinamico* per la memorizzazione di queste tramite la funzione *findFreeCells*, che considera le eventuali celle libere adiacenti a quelle marcate negli ultimi 10 turni.
Nel caso in cui *findFreeCells* non trovi celle se ne considerano 3 prese a caso da FC, vettore contenente tutte le celle libere.
Se il giocatore di cui si sta analizzando il turno si trova ad una mossa dalla vittoria, allora la si pone in cima all'array in modo da essere sempre valutata anche in caso di timeout.

2.3 evaluation e evaluate

evaluation chiama la funzione *evaluate*, una volta per il nostro giocatore ed una volta per l'altro, al fine di restituire un valore compreso tra -1 e 1, che sta ad

indicare qual è il giocatore che si trova in vantaggio. Il valore suddetto è ottenuto usando la funzione arcotangente normalizzata, poichè questa è monotona crescente e tende ad 1 al tendere dell'ascissa a $+\infty$.

$$-1 < \frac{Math.atan(max\_p2) - Math.atan(max\_p1)}{Math.PI/2} < 1$$

evaluate ispeziona la matrice di gioco esaminando le righe, le colonne, le diagonali e le antidiagonali al fine di trovare il numero massimo (opportunamente pesato) di simboli allineati dal giocatore preso in analisi. Facciamo ciò avvalendoci di un array dinamico di tipo *Streak*, una nuova classe da noi creata al fine di memorizzare la lunghezza della fila, il valore dei moltiplicatori ad essa associata e il numero di buchi che presenta.
Evaluate tollera fino ad una cella libera di fila nella streak e cambia tale valore grazie a due moltiplicatori che dipendono dal fatto che prima e/o dopo ci sia una cella libera o meno. Facciamo ciò dal momento che tollerare una n-upla, con $n > 1$, di celle libere di fila non risulterebbe efficiente poichè, in tal caso, l'avversario potrebbe contrastare facilmente la nostra strategia riempiendo uno di quei buchi.
La streak è conclusa quando incontriamo:

- il bordo
- un simbolo avversario
- almeno due celle vuote di fila

Per l'ultimo caso usiamo una variabile booleana inizializzata a *false*.
Tali moltiplicatori possono assumere i seguenti valori:

- 1 se la cella è libera
- 0.9 altrimenti

Nel caso in cui entrambi i moltiplicatori valgano 0.9 e la lunghezza della streak sia minore di K meno il numero di buchi presenti, allora tale streak non viene considerata.
Ogni altra streak viene pesata con i suoi moltiplicatori.
Una volta valutate tutte le file in questo modo, ritorniamo il valore più alto.

2.4 Timeout

Per ovviare ai limiti di tempo, all'interno di *initPlayer*, in base al numero di celle della matrice di gioco ($M \times N$), assegnamo una profondità massima che abbiamo empiricamente verificato essere la più adeguata per non causare timeout e ritornare una mossa in tempi ragionevoli.

3 Costo computazionale

Chiamando la dimensione dell'input $n := M \times N$ e $d := max\_depth$, il costo computazionale totale dell'algoritmo è

$$O(n^d) + O(n^2) + O(n) + c = O(n^d).$$

Infatti le funzioni *evaluateRows*, *evaluateCols*, *evaluateDiags* e *evaluateAntiDiags* costano $O(n)$ per via dei due cicli *for* annidati, all'interno dei quali ci sono solo operazioni di costo costante.
La funzione *bestCells* ha, nel caso peggiore, costo $O(n^2)$ per via di un ciclo su tutte le celle libere, a partire da ognuna delle quali è applicata la funzione *isntAlready*, che verifica che tale cella non sia stata già inserita nell'*array*.
Il costo totale è dominato asintoticamente dal costo dell'algoritmo *iterative deepening*, che richiama *AlphaBeta* d volte con profondità crescente.

$$O\left(\sum_{i=1}^{d} n^i\right) = O(n^d)$$

AlphaBeta ha un costo esponenziale poichè a partire da una configurazione visita un numero esponenziale di nodi:

$$1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! = \sum_{k=0}^{n} \frac{n!}{(n-k)!} = O(n!) < O(n^d)$$

In realtà, grazie alla funzione *bestCells*, poniamo un limite al numero di mosse valutate in ogni livello (massimo 80), quindi mediamente le prestazioni sono migliori di quelle previste.