

# **Relazione Progetto di Algoritmi e Strutture Dati Giocatore di ConnectX “Telos”**

**Giarrusso Lorenzo**

Mail istituzionale: [lorenzo.giarrusso@studio.unibo.it](mailto:lorenzo.giarrusso@studio.unibo.it)

Numero di matricola: 0001077543



**ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA**

Progetto di ASD A.A. 2022-2023

Corso di Laurea in Informatica

Dipartimento di Informatica – Scienza e Ingegneria

Università di Bologna

# INDICE

<b>1</b>	<b>Introduzione .....</b>	<b>3</b>
1.1	(M,N,K)-Game .....	3
1.2	Il progetto .....	3
<b>2</b>	<b>Scelte implementative.....</b>	<b>3</b>
2.1	selectColumn() .....	3
2.2	Minimax con alpha-beta pruning e move ordering .....	4
2.3	La funzione di valutazione euristica.....	4
2.4	Iterative Deepening .....	6
<b>3</b>	<b>Costi asintotici in termini di tempo.....</b>	<b>6</b>
3.1	Funzioni con costo costante .....	6
3.2	Costi di eval_sub(), eval(), update_eval() .....	7
3.3	Costo del minimax.....	8
3.4	Costo dell'iterative deepening.....	8
3.5	Costo di selectColumn() .....	8
<b>4</b>	<b>Idee per miglioramenti.....</b>	<b>9</b>
4.1	Miglioramento del costo.....	9
4.2	Miglioramento della valutazione euristica .....	9
4.3	Transposition table e specularità .....	9
4.4	Memorizzazione delle migliori mosse iniziali .....	10

# 1 Introduzione

## 1.1 (M,N,K)-Game

Il progetto riguarda una versione specifica del più generale problema del (M,N,K)-Game, ovvero un gioco composto da una matrice di M righe e N colonne in cui lo scopo di ogni giocatore è allineare consecutivamente almeno K dei propri gettoni in orizzontale, verticale e/o diagonale, ove ogni turno i giocatori si alternano a marcare una cella della tabella con un proprio gettone. In particolare, nel nostro caso si tratta di una generalizzazione del gioco Forza 4, che aggiunge dei vincoli alle mosse possibili. Infatti, in questo contesto ogni giocatore non è libero di scegliere una cella qualsiasi, bensì, selezionata una colonna, il gettone sarà posizionato nella posizione più bassa possibile che non sia già occupata da un altro gettone.

## 1.2 Il progetto

Il progetto richiede lo sviluppo di un giocatore software per il gioco sopra descritto, con (M,N,K) qualsiasi, tale che il giocatore sia in grado di scegliere, ad ogni turno, la migliore mossa possibile per raggiungere la vittoria. Ad ogni turno, la mossa deve essere selezionata entro un dato tempo limite. A causa di questo limite temporale, assume una particolare importanza nel contesto del progetto riuscire a limitare i costi degli algoritmi utilizzati, in modo da poter svolgere ‘più lavoro possibile’ nel tempo limite, e gestire il caso in cui si raggiunga il tempo limite durante la visita dell’albero di gioco.

# 2 Scelte implementative

## 2.1 selectColumn()

Il metodo selectColumn(), che restituisce ad ogni turno la colonna scelta in cui inserire il proprio gettone, opera in base a due casi:

- se stiamo effettuando la nostra prima mossa della partita, seleziona la colonna centrale. Questa è sempre la migliore prima mossa poiché la posizione centrale è quella più vantaggiosa per quanto riguarda i potenziali allineamenti. Selezionare la prima mossa ‘a priori’ ci permette inoltre di evitare di visitare una quantità enorme di nodi del game tree, poiché ignoriamo direttamente (N-1) sottoalberi figli del nodo di partenza.
- altrimenti, seleziona la colonna migliore tramite Iterative Deepening, a sua volta basato su Minimax con alpha-beta pruning.

## 2.2 Minimax con alpha-beta pruning e move ordering

La selezione della colonna migliore si basa sull'algoritmo Minimax, utilizzato per minimizzare la massima perdita in ogni situazione dato un game tree. L'implementazione del Minimax nel giocatore prevede delle ottimizzazioni dovute all'Alpha-Beta pruning ed al move ordering. Data l'enorme quantità di possibili situazioni di gioco, specialmente per valori di M,N,K elevati, l'algoritmo Minimax dovrebbe infatti visitare una quantità di nodi tale da rendere impossibile qualsiasi valutazione accettabile entro il tempo limite. La potatura Alpha-Beta è un meccanismo che permette di ridurre drasticamente il numero di nodi esplorati e valutati dall'algoritmo, ignorando i sottoalberi che sappiamo non influenzerebbero la valutazione del nodo nel Minimax; permette dunque il raggiungimento di profondità maggiori nel caso generale. La potatura Alpha-Beta brilla in particolar modo quando vengono considerate prima le mosse migliori per il giocatore corrente, poiché questo aumenta il numero di potature e dunque il guadagno di tempo rispetto al normale minimax. Per rendere questa eventualità più probabile, è previsto un riordinamento dei nodi figli di quello attualmente visitato. Questo riordinamento viene fatto in base ad una valutazione euristica di ciascun nodo figlio, secondo l'idea che, se una situazione è migliore a bassa profondità, allora probabilmente lo è anche a profondità maggiori. Le mosse, rappresentate in una lista di coppie <mossa, punteggio>, vengono ordinate in ordine decrescente per il giocatore che massimizza e crescente per quello che minimizza. Infine, va notato che, per scelta implementativa, il nostro giocatore è quello che massimizza.

## 2.3 La funzione di valutazione euristica

Data l'impossibilità di visitare completamente il game tree entro il tempo limite, si presenta la necessità di fornire una valutazione di una situazione (eventualmente) non terminale del gioco che permetta di stabilire se tale situazione sia vantaggiosa per uno dei due giocatori. L'implementazione di tale funzione si basa su alcune semplici considerazioni:

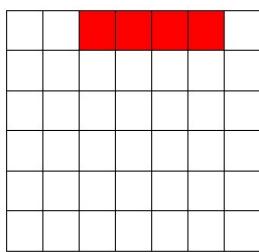
- la condizione ‘minima’ di vittoria è allineare K gettoni, dunque ci interessa studiare le sottosequenze di celle della tabella di gioco che abbiano lunghezza K.
- tali sottosequenze possono essere orizzontali, verticali, diagonali discendenti (“da nordovest a sudest”) o diagonali ascendenti (“da sudovest a nordest”, ovvero antidiagonali).
- righe e colonne vuote possono essere ignorate. Questo non migliora il costo nel caso pessimo, ma praticamente risulta in un grande miglioramento delle prestazioni nelle situazioni iniziali con (M,N) grandi poiché ci permette di ignorare anche le relative sottosequenze orizzontali e verticali.
- poiché una cella può essere occupata solo se essa appartiene alla riga più in basso oppure se la cella sottostante è già occupata, possiamo ignorare tutte le sottosequenze verticali in cui la cella più in basso è vuota, poiché abbiamo la certezza che lo siano anche tutte le altre.

- analizzando le diagonali ascendenti, sappiamo di poter ignorare gli angoli della matrice in alto a sinistra e in basso a destra, poiché in tali angoli le sottodiagonali hanno lunghezza  $< K$ . Analogamente per le diagonali discendenti e gli angoli in alto a destra e in basso a sinistra.
- una sottosequenza è particolarmente interessante se presenta almeno un gettone di un giocatore e nessun gettone del suo avversario; questo infatti implica che tale sequenza potrebbe potenzialmente portare alla vittoria del giocatore se tutte le sue  $K$  celle fossero occupate dal giocatore.

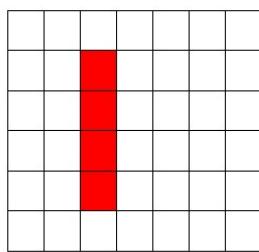
La funzione eval, dunque, verifica se la situazione studiata è terminale; se sì, restituisce il valore relativo (massimo se è una vittoria per il nostro giocatore, minimo se per l'avversario, 0 per una parità). Si noti che la valutazione delle situazioni terminali prende in considerazione la profondità a cui il nodo è stato visitato; in particolare, il valore della vittoria del nostro giocatore (che massimizza) viene diminuita del valore della profondità, mentre per l'avversario (che minimizza) aumenta. Così facendo, l'algoritmo tende a preferire vittorie a profondità minore (ovvero “che succedono prima”) e, nel caso diinevitabile sconfitta, cerca di ritardarla il più possibile.

Se invece la situazione non è terminale, l'algoritmo procede con la valutazione euristica. Questa consiste nel considerare ogni sottosequenza di  $K$  celle consecutive nella matrice in senso orizzontale, verticale, diagonale e antidiagonale. Ad ogni sottosequenza resta assegnato un punteggio il cui valore iniziale è il numero di nostri gettoni meno il numero di gettoni avversari. Per ogni sottosequenza, se essa contiene gettoni di un solo giocatore, il punteggio viene moltiplicato per  $2^{(\text{numero di gettoni del giocatore})}$ . La valutazione euristica della tabella di gioco è semplicemente la somma di tutti i punteggi delle sottosequenze.

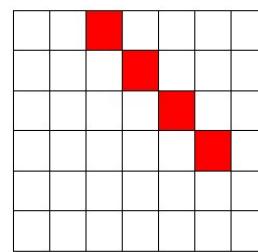
#### Esempi di sottosequenze con $K=4$



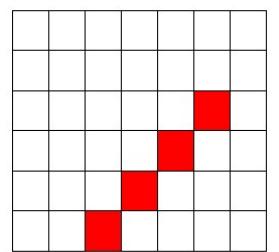
Orizzontale



Verticale



Diagonale



Antidiagonale

Come spiegato nella sezione 3.2, il costo di eval() è  $O(MNK)$ . Poiché vogliamo ottenere una valutazione euristica dei figli di ogni nodo per poter effettuare il move ordering, siamo interessati a diminuire il costo della valutazione in tali casi. Per la valutazione dei figli di un dato nodo si usa dunque un'altra funzione, update\_eval(), che, data la tabella di gioco e la colonna in cui si intende inserire il gettone, restituisce la valutazione euristica del nodo figlio semplicemente considerando le sottosequenze che vengono modificate dall'inserimento. Questo ha costo  $O(K^2)$ , come si dimostrerà nella sezione 3.2.

## 2.4 Iterative Deepening

L'algoritmo Minimax è essenzialmente un algoritmo di visita depth-first, dunque, data la presenza di un limite di tempo, se implementato senza ulteriori accorgimenti esso non sarebbe in grado di fornire sempre una scelta affidabile. Infatti, specialmente con  $(M, N, K)$  grandi, l'algoritmo raggiungerebbe il limite di tempo prima di aver effettivamente considerato tutte le mosse possibili. Per compensare a ciò, le principali opzioni sono:

- stimare in anticipo la massima profondità raggiungibile, che richiede una prima visita Minimax con profondità ‘sicura’ (es. 1 o 2) di cui considerare il tempo di esecuzione per ogni nodo, che permette di stabilire approssimativamente una profondità accettabilmente visitabile ma il cui valore deve essere tenuto aggiornato nel caso di timeout futuri e che comunque non garantisce un utilizzo ottimizzato di tutto il tempo a disposizione;
- usare il meccanismo dell'Iterative Deepening, una forma di visita simile a BFS che consiste nell'effettuare iterativamente delle visite con Minimax utilizzando profondità massima ogni volta maggiore. Nel caso di timeout quando si effettua la visita con profondità massima  $d$ , si utilizza la mossa migliore trovata nella visita con profondità massima  $d-1$  (a quel punto già completata).

Per questo progetto si è scelto di seguire la via dell'Iterative Deepening; questo presenta infatti il vantaggio di raggiungere sempre una buona profondità senza bisogno di stabilirla a priori. Inoltre, pur essendo una visita BFS, il suo costo in termini di memoria non è esponenziale, bensì lineare, come visto a lezione. Il difetto principale dell'Iterative Deepening deriva dal fatto che esso visita ripetutamente gli stessi nodi in iterazioni successive. Un modo per ridurre tale svantaggio deriva dall'utilizzo di Transposition Tables, qui non implementate, ovvero memorizzare (in tabelle hash) le situazioni già valutate con le relative mosse migliori in modo da non aver bisogno di ri-valutarle ripetutamente. Per ulteriori considerazioni al riguardo, vedasi la sezione 4, dedicata ad idee per eventuali miglioramenti.

## 3 Costi asintotici in termini di tempo

Siano  $M$  il numero di righe,  $N$  il numero di colonne,  $K$  il numero di gettoni da allineare per vincere. **Nota:** si userà l'assunzione che  $K \leq N$  e  $K \leq M$ , che non è necessariamente vera in termini assoluti ma vale per tutte le configurazioni previste per il progetto.

### 3.1 Funzioni con costo costante

Le seguenti funzioni hanno banalmente costo costante poiché servono solo per accedere direttamente ad un dato o ad eseguire una singola operazione:

- `initPlayer()`
- `playerName()`

- `timeIsRunningOut()`
- `isLeaf()`

## 3.2 Costi di eval\_sub(), eval(), update\_eval()

La funzione per la valutazione euristica, `eval`, fa uso del metodo `eval_sub()`, che, dato un array e due index ‘‘start’’ ed ‘‘end’’, restituisce il punteggio euristico relativo a tale sottosequenza. Questo viene fatto visitando le celle in questione, dunque `eval_sub()` ha costo lineare, ovvero  $\Theta(n)$  (si assume che la funzione `Math.pow()` sia  $O(1)$ ). La funzione `eval()`, tramite dei cicli annidati, considera ogni sottosequenza (verticale, orizzontale, diagonale o antidiagonale) di  $K$  celle consecutive nella tabella di gioco, e su ciascuna sottosequenza chiama `eval_sub()`; tale chiamata ha costo  $\Theta(K)$  per quanto appena detto. Si hanno i costi:

- **Allineamenti verticali**

Costo:  $O(MN + N(M-K+1)K) = O(MN + N(MK - K^2 + K)) = O(MN + MNK + NK^2 + NK) = O(MNK)$   
 $(si\ usa\ K \leq M \Rightarrow NK^2 \leq MNK)$

- **Allineamenti orizzontali**

Costo:  $O(M(N-K+1)K) = O(MNK + MK^2 + MK) = O(MNK)$   
 $(come\ sopra)$

- **Allineamenti diagonali**

Costo:  $\Theta((M-K+1)(N-K+1)K) = \Theta(MNK - MK^2 + MK - NK^2 + K^3 - K^2 + NK - K^2 + K) = \Theta(MNK)$   
 $(come\ sopra,\ ma\ anche\ K \leq M \wedge K \leq N \Rightarrow K^3 \leq MNK)$

- **Allineamenti antidiagonali**

Costo:  $\Theta((M-K+1)(N-K+1)K) = \Theta(MNK)$   
 $(uguale\ a\ sopra)$

Dunque, `eval()` ha complessivamente costo pessimo  $\Theta(MNK)$ . Il ciclo di inizializzazione di `isEmptyRow[]` ha costo  $\Theta(M)$ , che viene assorbito da  $\Theta(MNK)$ . Come citato sopra, questo costo avrebbe un enorme impatto sul minimax a causa dell'utilizzo della valutazione euristica per il move ordering. Per tale contesto si usa dunque la funzione `update_eval()`, che semplicemente ‘‘aggiorna’’ la valutazione euristica già calcolata per il padre andando a modificare la valutazione delle sottosequenze influenzate dalla mossa che ha generato il figlio. Il costo di tale funzione può essere trovato senza calcoli facendo delle semplici considerazioni:

- fissata una cella, le sottosequenze che la contengono possono avere 4 direzioni (verticale, orizzontale, diagonale, antidiagonale).
- fissata una direzione, la cella è coinvolta in al più  $K$  sottosequenze (in quanto in ciascuna occupa una delle  $K$  posizioni possibili).
- per ogni sottosequenza, chiamiamo `eval_sub()`, che avrà costo  $\Theta(K)$ .

Considerando che le sottosequenze vengono visitate due volte ciascuna (la prima volta per rimuovere il contributo euristico obsoleto dalla valutazione precedente, la seconda volta per aggiungere il contributo aggiornato), il costo della funzione risulta dunque essere  $\Theta(2^*4*K^2) = \Theta(K^2)$ .

### 3.3 Costo del minimax

Per quanto visto a lezione, sappiamo che l'utilizzo dell'alpha-beta pruning non migliora il costo asintotico dell'algoritmo minimax, poiché nel caso pessimo non si hanno potature e dunque si visitano comunque tutti i nodi. Il massimo numero di nodi in un livello è dato dal numero di possibili mosse ad ogni turno elevato al numero di turni considerato. Il numero di possibili mosse ad ogni turno è  $N$  (scelgo infatti una colonna in ogni turno), mentre il numero di turni considerato è  $d$ , ovvero la profondità massima di ricerca con cui è stato chiamato il minimax. Si hanno dunque  $N^d$  nodi nell'ultimo livello. Ad ogni nodo, il move ordering richiede di effettuare la valutazione euristica dei nodi figli (con `update_eval()`) e di ordinarli. Si hanno al più  $N$  nodi figli, ed assumiamo che la funzione di sorting built-in sia  $O(N\log N)$ , dunque il move ordering ha costo  $O(NK^2 + N\log N) = O(N(K^2 + \log N))$ . Il move ordering viene effettuato per ogni nodo non terminale visitato, dunque il costo complessivo è  $O(N^d N(K^2 + \log N)) = O(N^{d+1}(K^2 + \log N))$ . Poiché ci interessa il comportamento asintotico nel caso pessimo, possiamo assumere che  $d > 1$ , dunque risulta il costo  $O(N^d(K^2 + \log N))$ . Nel caso in cui invece i  $N^d$  nodi debbano essere valutati euristicamente, si ha il costo  $O(N^d MNK) = O(MN^d K)$ , che è maggiore rispetto a quello visto considerando il move ordering e dunque dominante.

### 3.4 Costo dell'iterative deepening

L'iterative deepening semplicemente calcola la valutazione euristica della situazione attuale e poi chiama iterativamente la funzione di minimax con profondità massima incrementata ad ogni iterazione. Detta  $d$  la profondità massima, si avrà dunque  $O(MNK + MNK + \dots + O(MNK + MN^d K) = O(MN^d K)$ .

### 3.5 Costo di selectColumn()

Infine, la funzione che viene chiamata per scegliere la mossa ad ogni turno, `selectColumn()`, semplicemente chiama la funzione di iterative deepening con una certa profondità massima  $d$ . Il costo di `selectColumn` è dunque banalmente anch'esso  $O(MN^d K)$ .

## 4 Idee per miglioramenti

### 4.1 Miglioramento del costo

Il miglioramento di gran lunga più significativo per questo giocatore sarebbe quello relativo alla funzione di valutazione euristica. Nell'implementazione corrente, infatti, la valutazione euristica consiste nella visita di tutte le possibili sottosequenze di K elementi presenti nella tabella di gioco. Questo, come visto sopra, implica un costo pari a  $\Theta(MNK)$ , che è estremamente alto ed ha un impatto notevole sui costi della selezione della colonna. Un approccio semplice per migliorare la situazione consisterebbe nel memorizzare il valore euristico della situazione attuale in una variabile membro della classe, mantenendo tale valutazione aggiornata a seguito di ogni mossa eseguita dall'avversario e dal giocatore stesso. Questo permetterebbe di abbattere notevolmente i costi dell'iterative deepening e dunque di `selectColumn()`, in quanto l'aggiornamento della valutazione euristica andrebbe da  $\Theta(MNK)$  a  $\Theta(K^2)$ . Di conseguenza, il costo di iterative deepening e `selectColumn()` scenderebbe a  $O(N^d(K^2 + \log N))$ , ovvero un comportamento dominato dai nodi non-terminali, un netto miglioramento rispetto alla situazione attuale.

### 4.2 Miglioramento della valutazione euristica

Un aspetto che non è stato tenuto direttamente in considerazione nell'implementazione della valutazione euristica è il fatto che le sottosequenze favorevoli (cioè occupate da soli gettoni nostri e spazi vuoti) intersecate tra loro forniscono un vantaggio assai superiore rispetto a quelle non intersecate; questo perché permettono di portarsi a situazioni di vittoria sicura anche qualora l'avversario blocchi una possibile vittoria. Questo potrebbe essere implementato raddoppiando il valore delle sottosequenze favorevoli che si intersecano con altre sottosequenze favorevoli.

### 4.3 Transposition table e specularità

Dato l'utilizzo dell'Iterative Deepening, che porta alla ripetuta ri-visita e ri-valutazione delle stesse situazioni di gioco, si avrebbe un buon guadagno dall'utilizzo delle Transposition tables, poiché esse ci permetterebbero di conoscere direttamente il risultato della valutazione precedente della stessa situazione. Va comunque citato che, implementandole tramite tabelle hash, si dovrebbero gestire le problematiche relative alle collisioni di hash (situazioni diverse risultano nello stesso hash) e alle collisioni di indice (hash diversi risultano nello stesso indice). Le collisioni di hash in particolare implicherebbero il bisogno di controllare che tabelle aventi lo stesso hash siano effettivamente uguali; questo potrebbe portare a visitare meno nodi qualora si implementi una valutazione euristica nettamente migliore della nostra. Un ulteriore miglioramento si potrebbe avere considerando anche le situazioni speculari; due situazioni che sono una "specchiata" rispetto all'altra possono infatti essere considerate come equivalenti, sebbene

in tal caso va tenuto in considerazione il fatto che le due tabelle sono fra loro simmetriche rispetto alla colonna centrale. Questo tuttavia implica ulteriori controlli sulle tabelle. Ad esempio:

1	2	3	4	5	6	7
			■	■		
			■	■		

1	2	3	4	5	6	7
			■	■		
			■	■		

Le due situazioni sopra sono fra loro speculari. Infatti, la colonna 1 a sinistra è uguale alla colonna 7 a destra, la colonna 2 a sinistra è uguale alla colonna 6 a destra, ecc. Poiché nella situazione a sinistra la mossa migliore per il rosso è la colonna 3, nella situazione a destra la mossa migliore è la colonna simmetrica alla 3 rispetto alla colonna centrale, dunque 5.

#### 4.4 Memorizzazione delle migliori mosse iniziali

Un enorme miglioramento per l'efficienza delle prime mosse sarebbe possibile stabilendo a priori le migliori prime mosse di una partita, che potrebbero essere memorizzate su file, in modo da poter dirigere la partita in una situazione per noi vantaggiosa sin dall'inizio.

Questo sarebbe dunque un meccanismo analogo alle “aperture” nelle partite di scacchi.

Tuttavia, per implementare questa idea servirebbe uno studio dettagliato delle possibilità del gioco da parte dell'implementatore.