DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Implementazione dell'algoritmo Minimax al gioco del tris e adattamento a matrici n x m.

Relatore: Prof. Antonio Giunta

Laureando: Claudio Padoan

ANNO ACCADEMICO 2021 – 2022

Data di laurea: 21/09/2022

Ringrazio il mio relatore,
per avermi seguito nella stesura della tesi,
Ringrazio la mia famiglia,
per avermi invogliato da sempre a studiare,
Ringrazio Ilaria,
per tutti questi anni assieme
e per essermi accanto in ogni situazione,
Ringrazio i miei colleghi e amici,
senza di loro questo percorso non sarebbe stato lo stesso.

Indice

1. La teoria dei giochi........ Pag. 5
    1.1 Somma........ Pag. 5
    1.2 Tris........ Pag. 6
2. Minimax........ Pag. 7
    2.1 Algoritmo........ Pag. 7
    2.2 Negamax........ Pag. 9
    2.3 Potatura alfa-beta........ Pag. 9
3. Progetto........ Pag. 12
    3.1 Descrizione........ Pag. 12
    3.2 File........ Pag. 13
        3.2.1 Field........ Pag. 13
        3.2.2 Game........ Pag. 14
            3.2.2.1 AlphaBeta........ Pag. 15
            3.2.2.2 Minimax........ Pag. 15
    3.3 Tabella dei tempi........ Pag. 15
4. Limiti e approcci futuri........ Pag. 19
5. Bibliografia........ Pag. 20
6. Sitografia........ Pag. 20
7. Appendice........ Pag. 21

1. La teoria dei Giochi

La *Teoria dei Giochi* può essere definita come *lo studio dei modelli matematici di conflitto e cooperazione tra "decision-makers" razionali*. Essa è in grado di fornire delle metodologie matematiche generali volte all'analisi di situazioni in cui due o più individui, talvolta denominati *giocatori* o *agenti*, prendono decisioni, le quali andranno ad influenzare la scelta dell'altro (Myerson, 1997).

A discapito di quanto suggerisca il nome, la *teoria dei giochi* e chi, di conseguenza, la studia, non vanno ad occuparsi esclusivamente di attività ricreative: i campi di applicazione sono molteplici e riguardano, ad esempio, vari campi delle scienze sociali, oppure svariati ambiti di logica, teorie di sistemi ed Informatica. In particolar modo, al giorno d'oggi, è generalmente applicata nelle scienze delle decisioni logiche negli esseri umani e nei calcolatori (Aumann, 2008).

La premessa indispensabile, per poter applicare gli strumenti della *teoria dei giochi*, è quella di arrivare alla vittoria. Tutti i giocatori devono conoscere le modalità di gioco, sia per quanto riguarda le regole che per quanto riguarda la conseguenza di ogni singola mossa, per sé stessi e per il continuo del gioco stesso. L'insieme delle mosse che un individuo intende fare vengono chiamate *strategie*. A seconda, poi, della strategia adottata dal singolo agente, ad esso verrà attribuito un *pay-off* o *vincita*, che può essere positivo, negativo o nullo a seconda degli effetti che tale strategia porrà al gioco stesso.

1.1 Somma

Un aspetto importante riguarda la *somma*; con questo termine andiamo a delineare una classificazione di giochi ben distinta.

Un gioco si definisce *a somma costante* se per ogni vincita di un giocatore vi è una corrispondente perdita per altri. Ci concentreremo in particolar modo ad un caso particolare: i giochi *a somma zero*, in cui la costante (il payoff complessivo) risulta nulla, ovverosia in cui le vincite risultanti siano uguali ed opposte (Gaburri, Norvig, Russell, 2005).

Un esempio esplicativo è dato dalla seguente situazione:

		GIOCATORE C	
		SCELTA A	SCELTA B
GIOCATORE R	SCELTA A	(-2 , +2)	(+2 , -2)
	SCELTA B	(+2 , -2)	(-2 , +2)

Nella prima cella i due giocatori, R e C, effettuano la stessa scelta. Per come abbiamo definito i *giochi a somma zero*, la risultante dei payoff dev'essere nulla quindi, preso come payoff positivo +2 e come payoff negativo -2, in base alle regole del gioco attribuiamo la *vincita* a R e la *sconfitta* a C.
Si procede allo stesso modo per le altre celle.

1.2 Tris

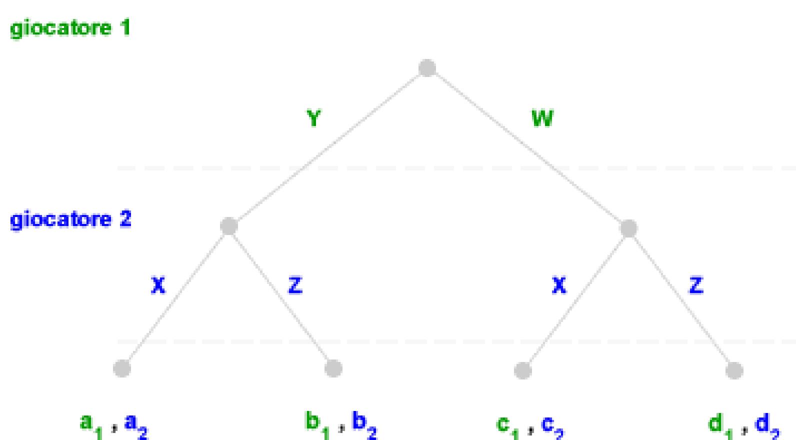
Il *Tris*, noto anche come *Filetto* o *TicTacToe*, è un popolare gioco da tavolo a *informazione completa*, ossia un gioco in cui la conoscenza delle strategie e delle mosse precedenti su ogni giocatore è condivisa da tutti gli altri.
Le regole del gioco sono banali: ad ogni giocatore è associato un simbolo, una 'X' ed un 'O'; Il giocatore avente 'X' comincia per primo e deve posizionare il simbolo in una delle nove celle di una griglia 3 x 3. L'avversario dovrà poi rispondere posizionando il rispettivo simbolo in una delle celle rimanenti. Vince il giocatore che avrà disposto tre simboli uguali in riga, colonna oppure diagonale. Nel caso in cui la griglia venisse riempita senza ottenere allineamenti tra simboli, la partita risulterà patta.
Per come li abbiamo definiti, il gioco del *Tris* appartiene sicuramente alla categoria dei *giochi a somma zero*, e per esso valgono quindi tutte le loro proprietà.

2. Minimax

Identifichiamo come *minimax* il teorema fondamentale della *teoria dei giochi*. Esso afferma che *qualsiasi gioco finito, a somma zero, e con due giocatori ha strategie miste ottimali* (Neumann, 1928).

Il *minimax* risulta essere quindi un metodo in grado di massimizzare (minimizzare) la minima (massimo) perdita (guadagno) possibile. Viene applicato, nella teoria dei giochi, sia in caso di giochi a turni che in caso di giochi simultanei, sempre basandosi sull'ipotesi che ciascun individuo è razionale ed effettua le proprie decisioni allo scopo di ottenere il massimo guadagno possibile.

Il metodo analizza le mosse migliori a partire dalla fine del gioco fino a risalire alla posizione corrente, in cui è avvenuta l'invocazione: l'analisi avviene attraverso l'*albero di gioco*.



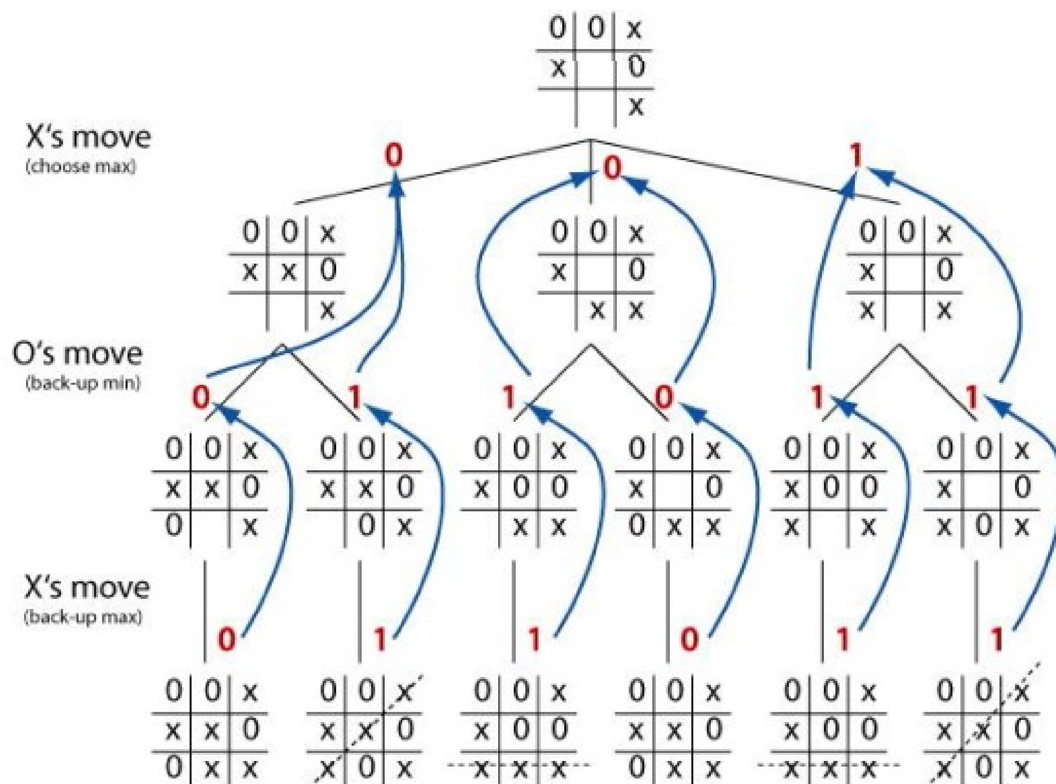
Si evidenziano il *nodo-madre*, associato alla situazione iniziale, e le varie *ramificazioni*, corrispondenti alle possibili situazioni del gioco. Se un qualsiasi nodo è *terminale*, allora identifica lo stato finale del gioco.

Limitandoci ai giochi a turni, il principio del minimax assume la forma di *algoritmo minimax*.

2.1 Algoritmo

L'algoritmo *minimax* è costituito da una funzione che ha lo scopo di valutare la posizione corrente ed indica quanto è desiderabile per il dato giocatore raggiungere quella posizione; viene quindi eseguita la mossa che minimizza il valore della migliore posizione raggiungibile dall'altro giocatore.

Attraverso questa sequenza di minimizzazioni/massimizzazioni, si arriva alla sequenza ottimale di mosse per il giocatore corrente nella sua attuale posizione. Per questi motivi, l'algoritmo viene applicato solo in giochi in cui il numero di stati è finito.
Qui di seguito viene riportato un esempio, applicato al gioco del tris:



Si noti come ogni nodo alterni la scelta del minimo o del massimo tra i loro figli, come descritto sopra, e venga quindi dato un *percorso ottimale* rispetto alla situazione iniziale alla luce di queste considerazioni. In questo esempio è, inoltre, ben evidenziata la dinamica dei payoff:

- +1 in caso di vittoria della X;
- -1 in caso di vittoria di O;
- 0 in caso di pareggio.

Passiamo quindi a dare una visione più *informatica* dell'algoritmo, mostrandone lo pseudocodice:

```
function minimax(nodo, profondità)
    SE nodo è un nodo terminale OPPURE profondità = 0
        return il valore euristico del nodo
    SE l'avversario deve giocare
        α := +∞
        PER OGNI figlio di nodo
            α := min(α, minimax(figlio, profondità-1))
    ALTRIMENTI dobbiamo giocare noi
        α := -∞
        PER OGNI figlio di nodo
            α := max(α, minimax(figlio, profondità-1))
    return α
```

Per quanto riguarda il valore euristico (terza riga) da assegnare al nodo, ci si riferisce semplicemente al payoff, poiché ci troviamo in una situazione terminale. È possibile, tuttavia, attribuire un punteggio ad un nodo anche se non si trova in una situazione terminale attraverso una *funzione euristica* specifica di un gioco, ma questo esula gli scopi della tesi (vedi capitolo 4).

Dopo aver mostrato la struttura e lo pseudocodice di *minimax*, si procede con la descrizione di alcune sue varianti.

2.2 Negamax

Quando ci si riferisce a *negamax*, si parla di una versione leggermente modificata di *minimax*. Esso si basa sulla proprietà dei *giochi a somma zero* per la quale il valore della posizione del giocatore A in un certo gioco è l'opposto del valore della posizione del giocatore B. Ciò implica che la differenza evidenziata dall'*if* della quarta riga può essere omesso. Al suo posto, e quindi anche al posto della differenziazione delle assegnazioni, verrà effettuata la chiamata ricorsiva in un modo simile a questo:

```
cur = -NegaMax(gamePosition, depth-1);
```

con la successiva valutazione del massimo tra questo valore ed *alpha*.

È stata evidenziata questa differenza poiché il *negamax* sarà ampiamente usato nel codice di questa tesi.

2.3 Potatura alpha-beta

È quindi chiaro che l'algoritmo presentato attui una ricerca in profondità nell'albero di gioco; pertanto, si ha una crescita esponenziale dei nodi da analizzare in funzione della profondità stessa: il gioco del tris, partendo da una griglia vuota, ha 9! combinazioni di gioco (~ 360 mila

nodi terminali), per cui al crescere dei nodi terminali aumenta la complessità spaziale e, soprattutto, temporale dell'algoritmo.

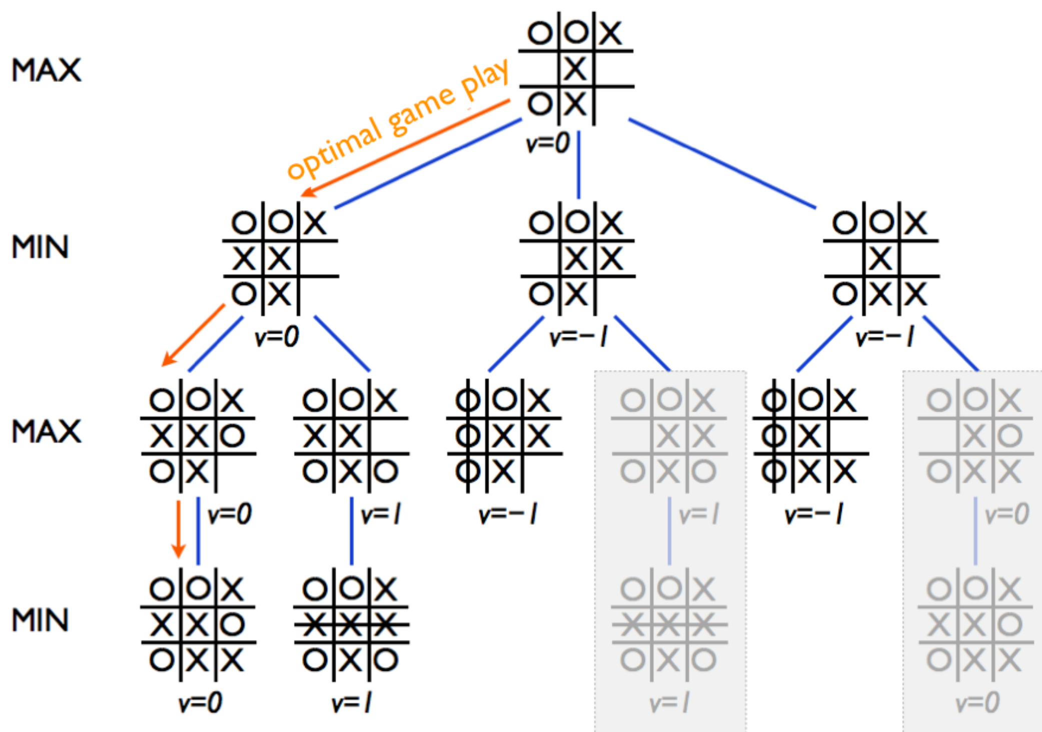
Per ridurne, quindi, la complessità si può pensare di adottare tecniche di *potatura logica* ossia l'eliminazione di una o più ramificazioni dell'albero di gioco da parte di un algoritmo di ricerca.

La potatura che verrà utilizzata nel codice sorgente è la *potatura alpha-beta*. L'idea alla base consiste nell'interrompere l'esplorazione in profondità dei nodi quando questi non sarebbero comunque selezionati dai giocatori perché peggiori rispetto alle alternative possibili.

Il seguente esempio ne descrive efficientemente l'idea alla base. Supponiamo di trovarci in un nodo interno: siamo nella situazione in cui si *massimizza* ed abbiamo Max (6, Min (3, X)); il risultato sarà sempre e comunque 6, per cui il ramo può essere eliminato. Situazione analoga può essere data da una situazione in cui si *minimizza* ed abbiamo Min (1, Max (3, Y)); allo stesso modo avremo come risultato sempre 1, per cui l'altro ramo può essere scartato.

L'algoritmo fa uso di due variabili, dette appunto *alpha* e *beta*, che hanno lo scopo di individuare questi casi, in modo che qualsiasi valore minore di *alpha* o maggiore di *beta* sarà automaticamente scartato, senza intaccare i risultati dell'albero di ricerca. La seguente rappresentazione dà un'idea

più concreta di quanto descritto:



Di seguito ne viene riportato lo pseudocodice:

```
FUNZIONE alfa_beta(nodo, profondità, α, β, massimizza)
    SE profondità = 0 O nodo è terminale
        RESTITUISCI valore euristico del nodo
    SE massimizza
        v := -∞
        PER OGNI figlio del nodo
            v := max(v, alfa_beta(figlio, profondità - 1, α, β, FALSO))
            α := max(α, v)
            SE β ≤ α
                INTERROMPI IL CICLO (* taglio secondo β *)
        RESTITUISCI v
    ALTRIMENTI
        v := +∞
        PER OGNI figlio del nodo
            v := min(v, alfa_beta(figlio, profondità - 1, α, β, VERO))
            β := min(β, v)
            SE β ≤ α
                INTERROMPI IL CICLO (* taglio secondo α *)
        RESTITUISCI v
```

3. Progetto

Il programma in appendice è stato sviluppato in ANSI C, versione standard del linguaggio di programmazione C, ed implementa un approccio risolutivo del gioco del Tris esteso a griglie m x n con k simboli da allineare per arrivare alla vittoria, utilizzando l'algoritmo *minimax*. Il progetto mette in luce le dipendenze temporali dell'algoritmo con il numero di nodi da analizzare, mostrandone limiti e differenze con la variante *alpha-beta pruning*.
L'eseguibile è generato dalla compilazione di *Game_AlphaBeta.c* (o *Game_Negamax.c*) e di *Field.c*, con il successivo linkaggio di *Game_AlphaBeta.o* con *Field.o*.

3.1 Descrizione

Il programma si esegue e visualizza da riga di comando, ponendo come parametri di ingresso: numero di righe, numero di colonne, numero di simboli da allineare.
Esempio: `Game 3 3 3` aprirà il classico gioco del Tris, quindi si giocherà usando una griglia 3 x 3 e con tre simboli da allineare per arrivare alla vittoria.
Mediante i parametri di ingresso 3 3 3, la grafica del programma apparirà in questo modo:

```
    1   2   3
1      |   |
    ---+---+---
2      |   |
    ---+---+---
3      |   |
```

Mediante i parametri di ingresso 7 5 5, la grafica del programma apparirà in questo modo:

```
    1   2   3   4   5
1      |   |   |   |
    ---+---+---+---+---
2      |   |   |   |
    ---+---+---+---+---
3      |   |   |   |
    ---+---+---+---+---
4      |   |   |   |
    ---+---+---+---+---
5      |   |   |   |
    ---+---+---+---+---
6      |   |   |   |
    ---+---+---+---+---
7      |   |   |   |
```

In seguito all'apertura verrà chiesto l'inserimento di riga e colonna, come indicati nella rappresentazione, per poter posizionare il proprio simbolo nella cella corrispondente, a cui seguirà una risposta da parte del programma con l'altro simbolo.
Il programma termina con l'allineamento dei k simboli indicati come parametri da riga di comando oppure con il riempimento della griglia di gioco, mostrando a schermo il vincitore della partita o l'eventuale patta. Le seguenti immagini mostrano i tre possibili risultati di una partita in una griglia 3 x 4 con tre simboli da allineare:

```
   1   2   3   4
1  x | o | x |
  ---+---+---+---
2  o | x |   |
  ---+---+---+---
3  x |   | o |

HA VINTO X!!
```

```
   1   2   3   4
1  x | x | o | o
  ---+---+---+---
2  o | o | x | x
  ---+---+---+---
3  x | x | o | o

PAREGGIO!!
```

```
   1   2   3   4
1  x | o | o | o
  ---+---+---+---
2    | x | x |
  ---+---+---+---
3    | x | o |

HA VINTO O!!
```

3.2 File

Dopo aver descritto input ed output del progetto, segue una parte descrittiva dei file sviluppati.

3.2.1 Field

Il primo file definisce, nel rispettivo header, i due tipi di dato strutturato che utilizzeremo nel codice: Field e Move.
Il tipo strutturato Field è composto da sei campi:

- `char** table`, indica la griglia di gioco, che è aggiornata dopo ogni mossa ed è utilizzata negli algoritmi di ricerca;
- `int row`, indica il numero di righe della griglia di gioco. Viene inserito da riga di comando;
- `int col`, indica il numero di colonne della griglia di gioco. Viene inserito da riga di comando;
- `int numSym`, indica il numero di simboli da allineare per far terminare la partita. Viene inserito da riga di comando;

- `int moveCounter`, indica il numero di mosse effettuate. Risulta efficiente per valutare la situazione di eventuale pareggio di una partita senza dover valutare se ogni cella di gioco è occupata;
- `Move lastMove`, variabile di tipo strutturato che va ad indicare l'ultima mossa eseguita sia da input che da algoritmo. Risulta efficiente nel valutare l'eventuale fine di una partita andando a controllare effettivamente se l'ultima mossa inserita risulta decisiva, senza dover controllare l'intero campo di gioco.

Useremo questo tipo strutturato per tutte quelle operazioni che andranno ad effettuare calcoli o confronti per una griglia di gioco.

Il secondo tipo strutturato, Move, è composto da:

- `int row`, indica la riga;
- `int col`, indica la colonna.

Viene usato per rendere più comoda l'analisi delle celle di un campo da gioco.

Nel Field.c vengono definite le funzioni necessarie per poter lavorare su una griglia di gioco e poterla rappresentare a schermo con un'opportuna formattazione.

3.2.2 Game

In questo file vengono definite tutte le funzioni per poter effettuare la partita. Qui si trova il *main*, per cui l'eseguibile sarà generato a partire da questo sorgente.

Prima di tutto viene inizializzato il terreno da gioco, dichiarando l'opportuna variabile di tipo Field e passando i parametri inseriti da riga di comando. È necessario sottolineare che non viene effettuato nessun tipo di controllo sui parametri, per cui l'utente dovrà curarsi di inserire l'input nel formato corretto.

Il passo successivo è quello di far partire il gioco: qui seguirà un ciclo *while* che rimarrà valido finché non ci sarà un vincitore o un'eventuale patta. Il gioco è strutturato a turni, per cui a seguito di un turno, che sia dell'utente o del computer, la variabile che identifica il giocatore attuale cambierà stato, indicando il rispettivo avversario; verrà inoltre aggiornato il *contamosse*, definito nella variabile strutturata Field.

La funzione che permette di eseguire la mossa del giocatore corrente dev'essere capace di distinguere se il giocatore è l'utente, permettendo quindi un input da tastiera, oppure il computer, facendo partire l'analisi del campo da gioco con la rispettiva ricerca della mossa migliore da parte degli algoritmi: *minimax* oppure *alphaBeta.*

3.2.2.1 AlphaBeta

Come già descritto in precedenza, *alpha-beta pruning* rappresenta un miglioramento di *minimax*.
Il codice in appendice è stato sviluppato seguendo questo approccio, in modo da avere risultati ottimi in termini temporali anche per griglie maggiori di quella convenzionale. Le relazioni tempo/nodi sono mostrate nella sezione successiva.

3.2.2.2 Minimax

L'algoritmo *minimax*, pur essendo alla base dell'*alpha-beta pruning*, con l'aumentare del numero di nodi da analizzare diventa molto lento, per cui il codice in appendice mostrerà una parte con questa strategia implementativa solo a titolo di esempio.

3.3 Tabelle dei tempi

Di seguito sono riportate le dipendenze temporali di *alpha-beta pruning* dal numero di livelli analizzati. Occorre sottolineare che i risultati sono ottenuti cercando di far "lavorare" il più possibile l'algoritmo: nel caso in cui si individuasse una situazione che potrebbe rivelarsi vincente, al fine di ottenere una relazione "pulita", verrà (ove possibile) evitata.
Segue un esempio:

```
Tempo trascorso: 17.974000
    1   2   3   4   5
 1  o | x |   | o |
   ---+---+---+---+---
 2    |   | o |   | x
   ---+---+---+---+---
 3    |   |   | x | o
   ---+---+---+---+---
 4    |   |   |   | x
   ---+---+---+---+---
 5  o |   |   |   |
```

```
Tempo trascorso: 25.183000
    1   2   3   4   5
 1  o | x |   | o |
   ---+---+---+---+---
 2    |   | x |   | x
   ---+---+---+---+---
 3    |   |   | o | o
   ---+---+---+---+---
 4    |   |   |   | x
   ---+---+---+---+---
 5  o |   |   |   |
```

```
Tempo trascorso: 31.368000
    1   2   3   4   5
 1  o | x | o | o |
   ---+---+---+---+---
 2    |   |   |   | x
   ---+---+---+---+---
 3    |   |   |   | o
   ---+---+---+---+---
 4    | x |   |   | x
   ---+---+---+---+---
 5  o |   |   |   |
```

È evidente, infatti, la differenza di tempo nei tre casi. Il primo caso è quello con tempo minore, difatti l'algoritmo riesce a trovare la soluzione migliore nella cella (2,3), che nell'ispezione è valutata prima delle altre. Il secondo caso, invece, trova la soluzione migliore in (3,4) che nell'ispezione è valutata successivamente. Il terzo caso mostra una situazione arbitraria, come spiegato prima, in cui la 'x' viene posta in (4,2) evitando perciò un caso di *quasi vittoria*.

Per la valutazione dei tempi quindi, nei limiti della situazione di gioco, verrà adottato l'approccio utilizzato per il terzo caso.

Nelle tabelle sono riportati unicamente i tempi > 1 ms, per cui i livelli che vengono risolti con un tempo minore non vengono indicati.

Come si evince nelle tabelle, esistono casi in cui un livello può essere risolto con maggior tempo rispetto al livello precedente: questo è uno dei casi in cui il livello precedente aveva una griglia nella quale l'inserimento portava ad una *quasi vittoria*, risolvibile quindi in tempo minore.

Le misurazioni sono avvenute in due fasi: nella prima fase iniziava 'x', per cui non si valutava la tabella vuota ma con un livello minore, l'altra fase analizzava il campo vuoto partendo con 'o'.

Nel 5x5, non sono stati analizzati tutti i livelli ma si partiva con una data situazione iniziale.

- **K = 5**

Livelli	t(s)
19	1881,77
18	375,19
17	130
16	28,22
15	12,63
14	7,745
13	0,759
12	9,819
11	0,013
10	0,467
9	0,003

5x5

Livelli	t(s)
19	2473,25
18	1152,072
17	146,476
16	48,476
15	7,771
14	14,458
13	0,379
12	12,002
11	0,0226
10	0,587

4x5

Livelli	t(s)
15	5,214
14	1,959
13	0,346
12	0,102
11	0,29
10	0,009
9	0,003
8	0,002

3x5

- **K = 4**

Livelli	t(s)
19	315,53
18	107,054
17	31,368
16	30,48
15	2,32
14	3,93
13	0,144
12	0,031
11	0,004
10	0,006

5x5

Livelli	t(s)
19	520,87
18	155,58
17	47,8
16	33,76
15	3,083
14	10,86

4x5

Livelli	t(s)
15	9,443
14	4,351
13	0,972
12	0,112
11	0,017
10	0,006

3x5

Livelli	t(s)
16	23,339
15	4,144
14	4,227
13	0,302
12	0,379
11	0,016
10	0,041
9	0,003
8	0,016

4x4

Livelli	t(s)
12	0,1
11	0,013
10	0,03

3x4

Livelli	t(s)
9	0,003
8	0,001

3x3

- **K = 3**

  Nota: in questa situazione viene indicata solo il gioco classico 3x3, poiché in tabelle di dimensione superiore la vittoria è assicurata per il primo ad effettuare la mossa.

Livelli	t(s)
9	0,003
8	0,001

3x3

Ora, invece, saranno presentate le dipendenze temporali utilizzando *minimax*. Ovviamente i tempi saranno molto più dilatati (ad esempio, con *alpha-beta* riesco ad ottenere risposta con una matrice 4 x 4 in circa quattro secondi, mentre con *minimax* non ho nessuna risposta) per cui il numero di livelli analizzati sarà ridotto.

- **K = 4**

Livelli	t(s)
13	1481,567
12	161,328
11	9,779
10	0,986
9	0,082
8	0,013
7	0,003

4x4

Livelli	t(s)
12	144,63
11	13,859
10	1,079
9	0,116
8	0,013
7	0,002

3x4

È evidente la dipendenza del tempo con il numero di livelli. Preso, ad esempio, il livello 13, si nota come il tempo sia circa dieci volte il tempo del livello 12, in linea con l'intuizione per la quale il livello 13 impiega 13! unità di tempo. Risulta perciò comprensibile l'impossibilità di fare test di misura su livelli più profondi; quindi, la ricerca si ferma con questo tempo (circa 1500 s) come tempo massimo (per *minimax*). Se provassimo, difatti a misurare il tempo di ricerca per il livello 14, potremmo presumere che il tempo risulti circa dieci volte 1500 secondi, ossia 15000 (circa 250 minuti, cioè 4 ore).

- **K = 3**

Livelli	t(s)
12	28,288
11	3,571
10	0,219
9	0,039
8	0,003

3x4

Livelli	t(s)
9	0,056
8	0,008
7	0,001

3x3

4. Limiti ed approcci futuri

Come si evince dalle misurazioni, i limiti del progetto sono evidenti: nonostante l'utilizzo di *alpha-beta pruning*, infatti, sono state tralasciate le misure al di sopra dei 19 livelli.
Il problema principale riguarda, appunto, il numero di livelli in cui si effettua l'analisi, poiché per ogni livello il numero di nodi aumenta in modo fattoriale, intaccando drasticamente i risultati temporali.
Un primo approccio alla risoluzione di questo problema potrebbe essere quello di fermarsi in nodi non foglia invece di analizzare tutto l'albero: questo diminuirebbe in modo significativo i livelli analizzati ottenendo tempi più brevi. Il problema di questo approccio è definire una *funzione euristica* che, data una griglia di gioco non ancora in una situazione terminale, riesca ad assegnarvi un punteggio, tale da riuscire a distinguere e preferire una situazione piuttosto che un'altra nonostante il gioco non sia ancora terminato; la ricerca di una funzione euristica generale, tuttavia, è piuttosto complicata (Abdoulaye, Abdel-Hafiz & Houndji, Vinasetan & Ezin, Eugène & Aglin, Gael., 2018).
Un ulteriore metodo, invece, per velocizzare l'algoritmo è quello di *memoizzare* il suo funzionamento, ossia di tenere traccia attraverso una struttura dati di situazioni già visitate per non ri-elaborarle in altri nodi nell'albero di gioco. Questo eviterebbe di calcolare nuovamente situazioni ripetute ricercandole all'interno della struttura dati, incrementando ovviamente la velocità dell'algoritmo. Il problema di questo approccio è la struttura dati da sviluppare: sarebbe necessario implementare in C una HashMap, avente come *chiave* il valore della data situazione, e come *valore* la rappresentazione in stringa della stessa situazione di gioco. Vista l'onerosità dello sviluppo di questa struttura dati e lo spreco necessario di memoria dovuto a quest'implementazione, è stato preferito l'incentrarsi sulla struttura del *minimax*.
Infine, si può pensare di adottare altri metodi di ricerca della situazione ottimale invece di *minimax* e delle sue varianti, ottenendo possibilmente risultati migliori, tra i quali possiamo citare *negascout*, *parallel alpha-beta pruning*, *Quiescence search, negaCStar* (Elnaggar, Ahmed & Gadallah, Mahmoud & Mostafa, Mostafa & Eldeeb, Hesham, 2014).

5. Bibliografia

- Abdoulaye, Abdel-Hafiz & Houndji, Vinasetan & Ezin, Eugène & Aglin, Gael. (2018). CARI2018 p265-275.
- Elnaggar, Ahmed & Gadallah, Mahmoud & Mostafa, Mostafa & Eldeeb, Hesham. (2014). A Comparative Study of Game Tree Searching Methods. International Journal of Advanced Computer Science and Applications. 5. 68-77. 10.14569/IJACSA.2014.050510.
- Gaburri, S., Norvig, P., Russell, S. J. (2005). Intelligenza Artificiale: Un Approccio Moderno. Italia: Pearson Education Italia.
- J. von Neumann, *Zur Theorie der Gesselschaftsspiele*, Math. Ann. 100, 1928, p. 295-320.
- Roger B. Myerson, (1991). *Game Theory, Analysis of Conflict*, Harvard University Press.
- R. J. Aumann,(2008). *The New Palgrave Dictionary of Economics*; 1–40.

6. Sitografia

- https://www.okpedia.it/gioco_a_somma_zero, consultato il 05/08/2022.
- https://www.okpedia.it/albero_di_gioco, consultato il 05/08/2022.
- https://www.okpedia.it/algoritmo_minimax, consultato il 05/08/2022.
- https://www.okpedia.it/potatura_alfa_beta, consultato il 05/08/2022.
- https://mathworld.wolfram.com/Tic-Tac-Toe.html, consultato il 05/08/2022.
- https://mathworld.wolfram.com/MinimaxTheorem.html, consultato il 05/08/2022.
- https://it.wikipedia.org/wiki/Minimax, consultato il 05/08/2022.
- https://course.elementsofai.com/it/2/3, consultato il 05/08/2022.
- https://materiaalit.github.io/intro-to-ai/part2/, consultato il 08/08/2022.
- https://it.wikipedia.org/wiki/Potatura_alfa-beta, consultato il 08/08/2022.

7. Appendice

Field.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

/* Struttura Move per facilitare la manipolazione delle posizioni nella
matrice */
typedef struct{
    int row;
    int col;
}Move;

/* Il campo in cui si svolge la partita */
typedef struct{
    char** table; /* matrice contenente {'x', 'o', ' '} */
    int row;
    int col;
    int numSym;
    int moveCounter; /* Contamosse per valutare la fine della parita */
    Move lastMove; /* Tiene conto dell' utima mossa inserita */
}Field;


void initField(Field*, int, int, int);

bool isPositionValid(Field*, Move*);

bool isFieldFull(Field*);

void drawField(Field*);

void freeField(Field*);
```

Field.c

```
#include "Field.h"

/* Inizializza la struttura $Field. */
/*
    IOP f Field da inizializzare;
    IP r Numero di righe del campo da gioco;
    IP c Numero di colonne del campo da gioco;
    IP n Numero di simboli da allineare per avere vittoria.
*/
void initField(Field* f, int r, int c, int n){
    int i, j;
    f->row = r;
    f->col = c;
    f->numSym = n;
    f->moveCounter = 0;
    f->table = (char **) malloc(r*sizeof(char *));
    assert(f->table != NULL);
    for(i = 0; i < r; i++){
            f->table[i]=(char *)malloc(c*sizeof(char));
      assert(f->table[i] != NULL);
      }/* for */
    for(i = 0; i < r; i++)
        for(j = 0; j < c; j++)
            f->table[i][j] = ' '; /* inizializzo ogni elemento della matrice
a '' */

}/* initField */

/* Valuta de la posizione e' valida (cioe` valuta se ci sono gia altri
simbolo nella posizione data).*/
/*
    IP f Field dal quale valutare la correttezza della posizione;
    IP m Move contenente la posizione.
    OR Valore booleano:
        -VERO se la posizione e' valida;
        -FALSO altrimenti.
*/
bool isPositionValid(Field* f, Move* m){
    if (m->row >= f->row || m->col >= f->col) /* se eccede il campo
(possibile solo da input)*/
        return false;
    else if(f->table[m->row][m->col] != ' ') /* se la posizione e' gia
occupata */
        return false;
    return true;
}/* isPositionValid */

/* Valuta se $f ha altre celle libere.*/
/*
    IP f Field da valutare se pieno o no.
    OR Valore Booleano:
        -VERO se $f NON ha altre celle libere;
        -FALSO altrimenti.
*/
bool isFieldFull(Field* f){
    return (f->moveCounter == f->row * f->col); /* qui si vede la forza del
contatore. */
}/* isFieldFull */
```

```
/* Rappresenta $f a schermo con i suoi valori attuali. */
/*
    IP+OV f Field da rappresentare a schermo
*/
void drawField(Field* f){
    int i, j;
    int rows = f->row;
    int columns = f->col;

    printf("  ");
    for(j = 0; j < columns; j++)
        printf(" %2d ", j+1);
    printf("\n");
    for(i = 0; i < rows; i++){
        printf("%2d ",i+1);
        for(j = 0; j < columns-1; j++)
            printf(" %c |", f->table[i][j]);
        printf(" %c\n", f->table[i][j]);
        if (i != rows - 1){
            printf("   ");
            for(j = 0; j < columns-1; j++)
                printf("---+");
            printf("---\n");
        }
    }
    printf("\n\n\n");
}/* drawField */

/* Libera la memoria dalla struttura Field usata. */
/*
    IOP f Field da liberare in memoria.
*/
void freeField(Field* f){
    int i;
    for(i = 0; i < f->row; i++)
        free(f->table[i]);
    free(f->table);
}/* freeField */
```

Game.c

```
#include "Field.h"
#include <time.h>
#include <limits.h>

#define PLAYER_1  1
#define PLAYER_2 -1


/* Dato un carattere in ingresso, che puo' essere 'x' o 'o', ne restituisce
il rispettivo intero
   definito da PLAYER_1 o PLAYER_2 */
/*
    IP c Carattere 'x' o 'o'
    OR PLAYER_1, se 'x', PLAYER_2, se 'o'
*/
int getPlayer(char c){
    return c == 'x' ? PLAYER_1 : PLAYER_2;
}/* getPlayer */

/* Valuta se avviene l'allineamento di $n simboli sulla riga dell'ultima
mossa */
/*
      IP f Field in cui valutare la situazione del terreno di gioco;
    IP square Flag che indica se la matrice è quadrata o meno.
      OR valore intero:
    -PLAYER_1 se ha vinto 'x';
    -PLAYER_2 se ha vinto 'o';
      - 0 se non c'è allineamento nella riga.
*/
int rowCheck(Field* f, bool square){
    int c = f->col;
    int n = f->numSym;
      int j, count = 1;
      int lastRow = f->lastMove.row;
      int lastCol = f->lastMove.col;
      char currPlayer = f->table[lastRow][lastCol]; /* giocatore corrente,
preso dall'ultima mossa eseguita */

      if (square){ /* se matrice quadrata */
            for(j = 0; j < c; j++){ /* parto dalla prima colonna */
              if(f->table[lastRow][j] != currPlayer) /* se trovo un elemento
diverso, blocco l'esecuzione */
                  break;
              if(j == c-1) return getPlayer(currPlayer); /* se ho
controllato tutta la riga senza uscire */
                                                       /* allora vince
$currPlayer */
          }/*for*/
      }/* if */
      else {
            int startCol = lastCol; /* parto dall'ultima colonna inserita */
        if (startCol < c/2){ /* se mi trovo prima della metà */
            while(startCol >= 0 && f->table[lastRow][startCol] ==
currPlayer) startCol--; /* mi porto a sinistra */
            for(j = startCol + 2; j < c; j++){ /* e faccio la valutazione*/
                if(f->table[lastRow][j] == currPlayer) count++;
                else break;
                if(count == n) return getPlayer(currPlayer);
            }/*for*/
        }/*if*/
```

```
        else { /* se sono oltre la metà */
            while(startCol < c && f->table[lastRow][startCol] == currPlayer)
startCol++; /* parto da destra */
            for(j = startCol - 2; j >= 0; j--){ /* e valuto */
                if(f->table[lastRow][j] == currPlayer) count++;
                else break;
                if(count == n) return getPlayer(currPlayer);
            }/*for*/
        }/*else*/
      }/* else */
      return 0; /* no vincita attuale*/
}/* rowCheck */
```

```
/* Valuta se avviene l'allineamento di $n simboli sulla colonna dell'ultima
mossa */
/*
      IP f Field in cui valutare la situazione del terreno di gioco;
    IP square Flag che indica se la matrice è quadrata o meno.
      OR valore intero:
    -PLAYER_1 se ha vinto 'x';
    -PLAYER_2 se ha vinto 'o';
      - 0 se non c'è allineamento nella colonna.
*/
int columnCheck(Field* f, bool square){
      int r = f->row;
    int n = f->numSym;
      int i, count = 1;
      int lastRow = f->lastMove.row;
      int lastCol = f->lastMove.col;
      char currPlayer = f->table[lastRow][lastCol]; /* giocatore corrente,
preso dall'ultima mossa eseguita */

    if(square){ /* se matrice quadrata */
      for(i = 0; i < r; i++){
            if(f->table[i][lastCol] != currPlayer) /* mantengo fissa la
colonna e scorro le righe*/
                break; /* nel caso in cui trovo un carattere diverso, esco
dal ciclo */
            if(i == r-1) return getPlayer(currPlayer); /* se ho controllato
tutta la colonna senza uscire */
                                                        /* allora vince
$currPlayer */
        }/*for*/
      }/* if */
      else {
            int startRow = lastRow;
        if(startRow < r/2){ /* stesse valutazioni della riga */
            while(startRow >= 0 && f->table[startRow][lastCol] ==
currPlayer) startRow--;
            for(i = startRow+2; i < r; i++){
                if(f->table[i][lastCol] == currPlayer) count++;
                else break;
                if(count == n) return getPlayer(currPlayer);
            }/*for*/
        }/*if*/
        else {
            while(startRow < r && f->table[startRow][lastCol] == currPlayer)
startRow++;
            for(i = startRow - 2; i >= 0; i--){
                if(f->table[i][lastCol] == currPlayer) count++;
                else break;
                if(count == n) return getPlayer(currPlayer);
            }/*for*/
        }/*else*/
      }/* else */

      return 0;
}/* columnCheck */

/* Valuta se avviene l'allineamento di $n simboli sulle diagonali relative
all'ultima mossa */
/*
      IP f Field in cui valutare la situazione del terreno di gioco;
    IP square Flag che indica se la matrice è quadrata o meno.
      OR valore intero:
    -PLAYER_1 se ha vinto 'x';
```

```
    -PLAYER_2 se ha vinto 'o';
      - 0 se non c'è allineamento nelle diagonali.
*/
int diagonalCheck(Field* f, bool square){
      int r = f->row;
    int c = f->col;
    int n = f->numSym;
      int i, j, count = 1;
      int lastRow = f->lastMove.row;
      int lastCol = f->lastMove.col;
      char currPlayer = f->table[lastRow][lastCol]; /* giocatore corrente,
preso dall'ultima mossa eseguita */

      if(square){ /* se matrice quadrata */
            for(i = 0, j = 0; i < r && j < c; i++, j++){ /* diagonale
decrescente [0][0]->[r-1][r-1] */
            if(f->table[i][j] != currPlayer)
                break;
            if(i == r-1 && j == c-1) return getPlayer(currPlayer);
        }/*for*/
        for(i = r-1, j = 0; i >= 0  && j < c; i--, j++){ /* diagonale
crescente [0][r-1]->[r-1][0]*/
            if(f->table[i][j] != currPlayer)
                break;
            if(i == 0 && j == c-1) return getPlayer(currPlayer);
        }/*for*/
      }/* if */
      else{
            /* 1.Diagonale decrescente: */
        int startRow = lastRow; /* sia righe che colonne iniziali */
        int startCol = lastCol;
        for(i = lastRow-1, j = lastCol-1; i >= 0 && j >= 0 && f->table[i][j]
== currPlayer ; i--, j--){
            startRow = i; /* aggiorno finchè non trovo un elemento diverso
*/
            startCol = j;
        }/* for */
        if(startRow + n <= r && startCol + n <= c) /* se ho abbastanza celle
in diagonale per i simboli */
            for (i = startRow+1, j = startCol+1; i < r && j < c; i++, j++){
/* stessa valutazione */
                if(f->table[i][j] == currPlayer) count++;
                else count = 0;
                if(count == n) return getPlayer(currPlayer);
            }/* for */

        count = 1; /* resetto il contatore */

            /* 2.Diagonale crescente: */
        startRow = lastRow; /* sia righe che colonne iniziali */
        startCol = lastCol;
        for(i = lastRow, j = lastCol; i >= 0 && j < c && f->table[i][j] ==
currPlayer; i--, j++){
            startRow = i; /* aggiorno finchè non trovo un elemento diverso
*/
            startCol = j;
        }/* for */

        if(startRow + n <= r && startCol - n >= -1) /* se ho abbastanza
celle in diagonale per i simboli */
            for (i = startRow + 1, j = startCol - 1; i < r && j >= 0 ; i++,
j--){ /* stessa valutazione */
                if(f->table[i][j] == currPlayer) count++;
```

```
                else count = 0;
                if(count == n) return getPlayer(currPlayer);
            }/* for */
    }/* else */

    return 0;
}/* diagonalCheck */

/* Funzione di valutazione dello stato del gioco. Valuta se c'è un
vincitore, in corso, o patta. */
/*
    IP f Field in cui valutare la situazione del terreno di gioco.
    OR valore intero:
    -PLAYER_1 se ha vinto 'x';
    -PLAYER_2 se ha vinto 'o';
    -2 se il gioco è ancora in corso;
    -0 se il gioco termina in patta.
*/
int checkGameState(Field* f){
    int win;
    bool square = (f->col == f->row && f->row == f->numSym);

    win = rowCheck(f, square);
    if(win!=0) return win;
    else{
      win = columnCheck(f, square);
      if(win!=0) return win;
      else{
            win = diagonalCheck(f, square);
            if(win!=0) return win;
            }/* else */
      } /* else */

    if (!isFieldFull(f)) return 2; /* gioco in corso se ho superato if/else
e la matrice non è piena */

    return 0; /* la matrice è piena e non ho trovato vittorie, quindi patta
*/
}/* checkGameState */
```

```
/* Algoritmo potatura alpha-beta */
/*
    IP f Field in cui eseguire l'algoritmo;
    IP alpha Punteggio minimo che il giocatore massimizzante può
raggiungere;
    IP beta Punteggio massimo che il giocatore minimizzante può raggiungere;
    IP depth Profondita dell'albero di ricerca;
    IP player Giocatore attuale.
    OR miglior/peggior punteggio della data profodita', a seconda del
giocatore attuale.
*/
int alphaBeta(Field* f, int alpha, int beta, int depth, int player) {
    int winner = checkGameState(f); /* prima valutazione sulla matrice */
    int min = INT_MAX, max = INT_MIN; /* inizializzati a valori limite in
modo da essere SICURAMENTE superati */
    Move m;
    int i, j;
    if(winner != 2) return winner*10; /* ritorno il giocatore vincente*10 */

    m.row = -1; /* serve come flag per valutare se si è trovata una mossa
migliore */
    m.col = -1;

    for(i = 0; i < f->row; i++){ /* righe */
        for(j = 0; j < f->col; j++){ /* colonne */
            Move move, temp; /* move cattura la mossa da fare, temp serve
per salvare mosse interne */
            move.row = i;
            move.col = j;
            if(isPositionValid(f, &move)){ /* se è una mossa ammissibile */
                int thisScore;
                if(player == PLAYER_1)f->table[move.row][move.col] = 'x';
                else f->table[move.row][move.col] = 'o';
                temp.row = f->lastMove.row; /* salvo l'ultima mossa
precedente in una variabile temporanea */
                temp.col = f->lastMove.col;
                f->lastMove.row = move.row;
                f->lastMove.col = move.col;
                f->moveCounter++;
                thisScore = alphaBeta(f, alpha, beta, depth+1, -player); /*
ricorsivo con $depth+1 e -player */
                if(player == PLAYER_1 && max < thisScore){ /* se il
punteggio attuale è maggiore di $max */
                    max = thisScore; /* sostituisco */
                    m.row = i; /* salvo la mossa */
                    m.col = j;
                    alpha = (alpha > max) ? alpha : max; /* aggiorno alpha
*/
                    }/* if */
                else if (player == PLAYER_2 && min > thisScore){ /* se il
punteggio attuale è minore di $min */
                    min = thisScore; /* sostituisco */
                    m.row = i; /* salvo la mossa */
                    m.col = j;
                    beta = (beta < min) ? beta : min; /* aggiorno beta */
                    }/* else if */
                f->table[move.row][move.col] = ' '; /* ripristino le
condizioni iniziali */
                f->moveCounter--;
                f->lastMove.row = temp.row;
                f->lastMove.col = temp.col;
                if (beta <= alpha) break; /* caso di potatura, esco dai due
for */
```

```
            }/* for */
            if (beta <= alpha) break;
        }/* for */
    }/* for */
    if(m.row == -1 || m.col == -1) return 0; /* se non ho aggiornato il
punteggio */
    if (player == PLAYER_1) return max - depth; /* se giocatore
massimizzante, $depth rappresenta un peso */
    else return min + depth; /* se giocatore minimizzante, $depth
rappresenta un peso */
}/* alphaBeta */

/* Trova la mossa migliore tramite l'algoritmo di alphaBeta */
/*
    IP f Field in cui ricercare la mossa migliore;
    IOP m Mossa migliore trovata da restituire come parametro;
    IP currPlayer Giocatore attuale.
*/
void findBestMove(Field* f, Move* m, int currPlayer){
    int score = INT_MIN; /* valore minimo */
    int i, j;
    for(i = 0; i < f->row; i++){ /* stessa struttura dell'algoritmo */
        for(j = 0; j < f->col; j++){
            Move move, temp;
            move.row = i;
            move.col = j;
            if(isPositionValid(f, &move)){
                int tempScore;
                f->table[move.row][move.col] = 'o'; /* solo la mossa del
computer */
                temp.row = f->lastMove.row; /* salvo i parametri */
                temp.col = f->lastMove.col;
                f->lastMove.row = move.row;
                f->lastMove.col = move.col;
                f->moveCounter++;
                tempScore = -alphaBeta(f, INT_MIN, INT_MAX, 0, -currPlayer);
/* negAlphaBeta */
                f->table[move.row][move.col] = ' '; /* backTracking dei
parametri precedenti */
                f->moveCounter--;
                f->lastMove.row = temp.row;
                f->lastMove.col = temp.col;
                if(tempScore > score) {
                    score = tempScore;
                    m->row = i; /* bestMove che deve fare il computer */
                    m->col = j;
                }/* if */
            }/* if */
        }/* for */
    }/* for */
}/* findBestMove */
```

```
/* Dato il giocatore $player, esegue una mossa da Input se $player =
PLAYER_1, altrimenti esegue la mossa */
/* migliore da $findBestMove
*/
/*
    IOP f Field in cui caricare le mosse;
    IP player Valore intero {1, -1}, che indica il giocatore corrente.
*/
void makeMove(Field* f, int player){
    Move m, bestMove;
    if(player == PLAYER_1){ /* giocatore da Input */
        printf("mossa da fare: \n");
        printf("Riga: ");
        scanf("%d", &m.row);
        m.row--; /* la rendo a al *ccettabile dal $isPositionValid */
        printf("colonna: ");
        scanf("%d", &m.col);
        m.col--; /* come per la riga */
        if(isPositionValid(f, &m)){ /* se mossa ammissibile */
            f->table[m.row][m.col] = 'x'; /* scrivo 'x' nella cella
indicizzata da $m*/
            f->lastMove.row = m.row;
            f->lastMove.col = m.col;
            f->moveCounter++; /* aggiorno il contamosse */
        }/* if */
    }/* if */
    else{
        clock_t t; /* variabile per le misure di tempo */
        t = clock(); /* comincio a tenere il tempo da qui */
        findBestMove(f, &bestMove, player); /* trovo la mossa migliore data
la situazione corrente di $f */
        f->table[bestMove.row][bestMove.col] = 'o'; /* scrivo 'o' nella
cella indicizzata da $bestMove */
        f->lastMove.row = bestMove.row;
        f->lastMove.col = bestMove.col;
        f->moveCounter++; /* aggiorno il contamosse */
        t = clock()-t;
        printf("\nTempo trascorso: %f\n", (double)t/CLOCKS_PER_SEC);
    }/* else */
}/* makeMove */

/* Mostra a schermo il risultato della partita */
/*
    IP winner Intero contenente {1, 0, -1};
    OV Mostra a video il vincitore, indicato da $winner.
*/
void displayWinner(int winner){
    if(winner == -1) printf("\n\nHA VINTO O!!");
    if(winner ==  1) printf("\n\nHA VINTO X!!");
    if(winner == 0) printf("\n\nPAREGGIO!!");
}/* displayWinner */
```

```
/* Faccio partire il gioco */
/*
    IOP f Field che contiene le informazioni per il gioco.
*/
int gameStart(Field* f){
    bool victory = false;
    int currPlayer = PLAYER_1; /* il primo a partire sara il giocatore da
input */
    int winner = 0;

    while(!victory){
        drawField(f); /* disegno la griglia aggiornata mossa per mossa */
        makeMove(f, currPlayer);
        winner = checkGameState(f); /* catturo il vincitore */
        currPlayer = -currPlayer; /* cambio di giocatore */
        if (winner != 2)
            victory = true;
    }/* while */
    drawField(f); /* disegno anche apppena fuori dal while per mostrare la
mossa che ha terminato la partita */
    return winner;
}/* gameStart */

/*
    Gli argomenti da inserire da riga di comando sono:
    n --> numero di righe della matrice;
    m --> numero di colonne della matrice;
    k --> numero di simboli da allineare.
    esempio: Game 3 3 3 fa partire il classico tris 3x3.
*/
int main(int argc, char **argv){
    int winner;
    Field f;
    initField(&f, atoi(argv[1]), atoi(argv[2]), atoi(argv[3])); /*
inizializzo con i valori da riga di comando */
    winner = gameStart(&f);
    displayWinner(winner);
    freeField(&f); /* libero la memoria dalla struttura */
    return 0;
}/* main */
```

Nota: Il codice riportato dal file *Game.c* utilizza la potatura alfa-beta.

Invece, se si volesse utilizzare, *minimax*, basterebbe:

~ sostituire la funzione `alphabeta` con la seguente funzione minimax:

~ e, successivamente modificare la chiamata su `findBestMove`.

```
/* Algoritmo minimax */
/*
    IP f Field in cui eseguire l'algoritmo;
    IP player Giocatore attuale.
    OR miglior/peggior punteggio, a seconda del giocatore attuale.
*/
int minimax(Field* f, int player) {
    int winner = checkGameState(f); /* prima valutazione sulla matrice */
    Move m;
    int score = -2; /* punteggio minimo */
    int i, j;
```

```
    if(winner != 2) return winner*player; /* ritorno il valore del giocatore
vincente*giocatore attuale */

    m.row = -1; /* serve come flag per valutare se si è trovata una mossa
migliore */
    m.col = -1;

    for(i = 0; i < f->row; i++){ /* righe */
        for(j = 0; j < f->col; j++){ /* colonne */
            Move move, temp; /* move cattura la mossa da fare, temp serve
per salvare mosse interne */
            move.row = i;
            move.col = j;
            if(isPositionValid(f, &move)){ /* se è una mossa ammissibile */
                int thisScore;
                if(player == player_1) f->table[move.row][move.col] = 'x';
                else f->table[move.row][move.col] = 'o';
                temp.row = f->lastMove.row; /* salvo l'ultima mossa
precedente in una variabile temporanea */
                temp.col = f->lastMove.col;
                f->lastMove.row = move.row;
                f->lastMove.col = move.col;
                f->moveCounter++;
                thisScore = -minimax(f, -player); /* negamax */
                f->table[move.row][move.col] = ' '; /* ripristino le
condizioni iniziali */
                f->moveCounter--;
                f->lastMove.row = temp.row;
                f->lastMove.col = temp.col;
                if(thisScore > score) { /* se trovo un punteggio migliore*/
                    score = thisScore;
                    m.row = i;
                    m.col = j;
                } /* if */
            }/* if */
        }/* for */
    }/* for */
    if(m.row == -1 || m.col == -1) return 0; /* nel caso non ci fosse
sovrascrittura */
    return score; /* punteggio attuale */
}/* minimax */
```