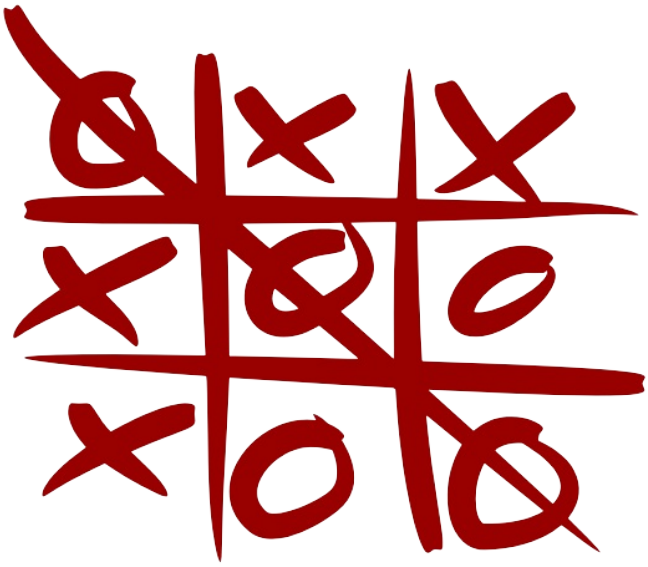
Table of Contents

Tic-Tac-Toe: progetto di Intelligenza Artificiale



Questo documento ha lo scopo di descrivere il progetto *Tic-Tac-Toe*, sviluppato dagli studenti **Davide Rigoni** e **Giovanni Mazzocchin** per il corso di *Intelligenza Artificiale*. Lo scopo del progetto era l'implementazione di un programma in grado di giocare intelligentemente al classico gioco da tavolo *Tris*, grazie a diverse varianti dell'algoritmo **Minimax** e a **funzioni euristiche** di complessità variabile.

Il *software* è stato sviluppato in **Java** e, visto lo scopo del progetto e l'ammontare di ore ad esso dedicato, come interfaccia utente è stata scelta la riga di comando.

Interfaccia di gioco

La semplice interfaccia a riga di comando permette all'utente di scegliere, tramite codici numerici, una configurazione contenente i seguenti dati:

1. Numero (*0* oppure *1*) del giocatore che esegue la prima mossa;
2. Dimensione del tavolo da gioco (può essere sia pari sia dispari);
3. Numero di celle bloccate;
4. Possibilità di giocare o meno contro l'intelligenza artificiale implementata nel progetto (il giocatore *computer* è sempre associato al codice numerico *1*);
5. Variante di *Minimax* utilizzata dall'intelligenza[1];
6. Funzione di valutazione utilizzata dall'intelligenza;
7. Profondità massima raggiungibile da *Minimax*.

Dopo il completamento della configurazione viene rappresentato il tavolo da gioco, sotto il quale viene richiesto l'inserimento delle coordinate corrispondenti alla mossa scelta. Se l'avversario scelto dall'utente è l'intelligenza artificiale, ad ogni mossa dell'utente seguirà una mossa (con relativa rappresentazione) del computer.

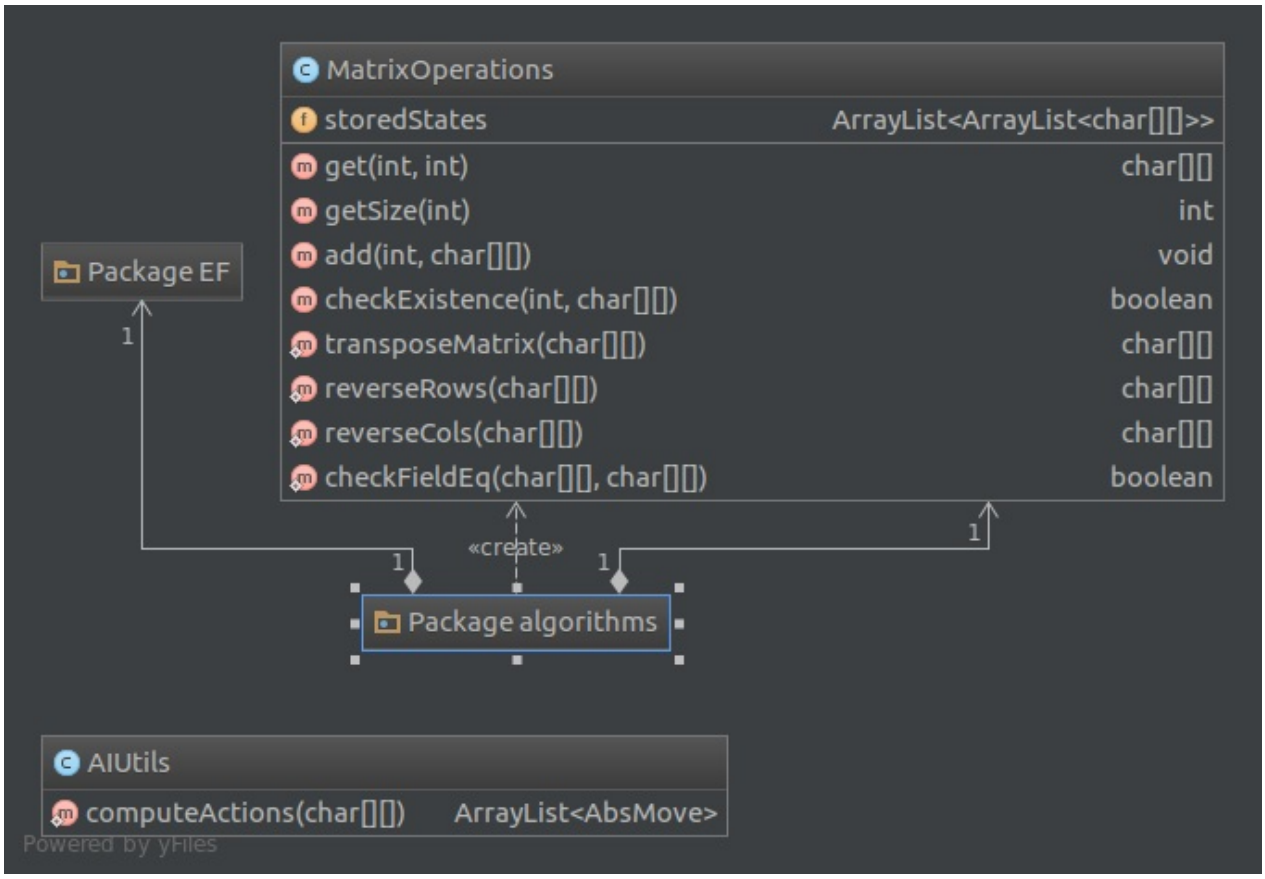
> [1]. Come vedremo, esse sono Minimax semplice, con alpha-beta pruning, con rottura delle simmetrie e con la combinazione di questi due miglioramenti. ↩

Struttura del software

Il progetto è stato sviluppato interamente in **Java SE 7** e pertanto è conforme alle classiche direttive di ogni progetto *Java*, quali la suddivisione in *package* e *classi*.

Ovviamente la progettazione delle gerarchie di classi è stata conforme ai dettami base dell'Ingegneria del Software, con conseguente eliminazione del codice duplicato grazie ad un uso oculato di interfacce e classi astratte.

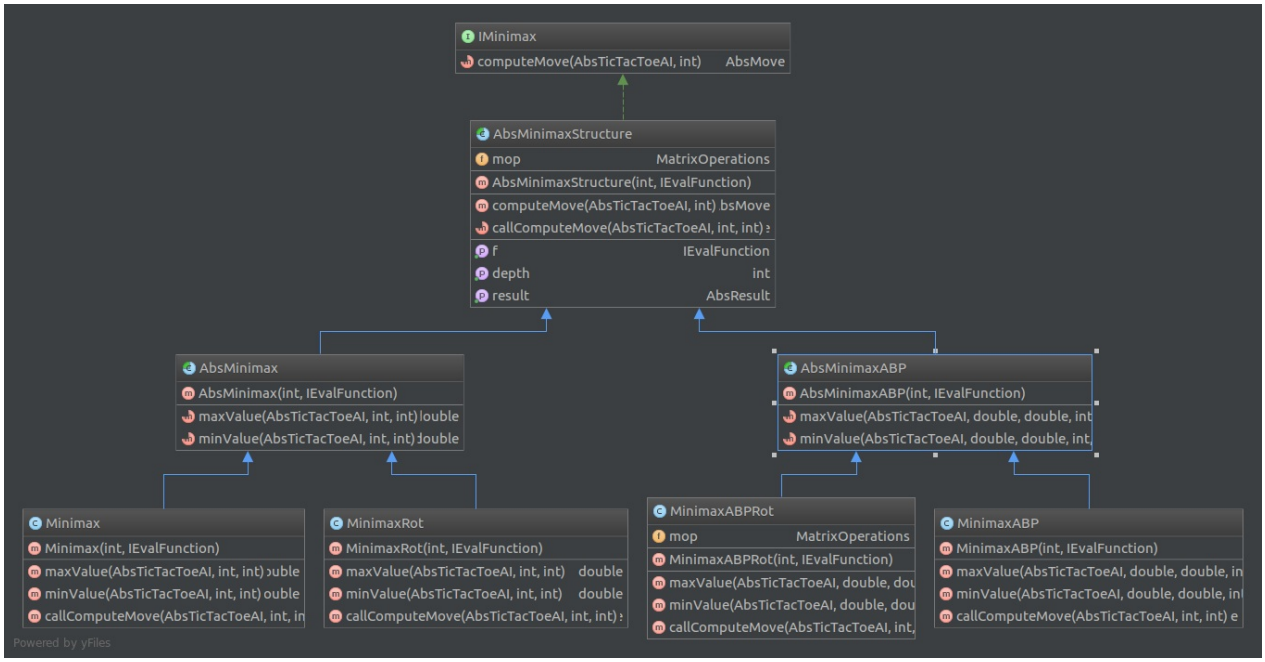
Il package *AI*



Oltre a contenere i due *sottopackage*, esso contiene nel suo livello più alto due classi:

- ***AIUtils***: i suoi metodi statici di utilità eseguono calcoli su una data configurazione, quali il punteggio e le mosse eseguibili a partire da essa;
- ***MatrixOperations***: i suoi metodi (sempre statici) implementano alcune operazioni elementari sulle matrici, quali la trasposizione e l'inversione di righe o colonne, operazioni indispensabili nella fase di rotazione del campo e rottura delle simmetrie.

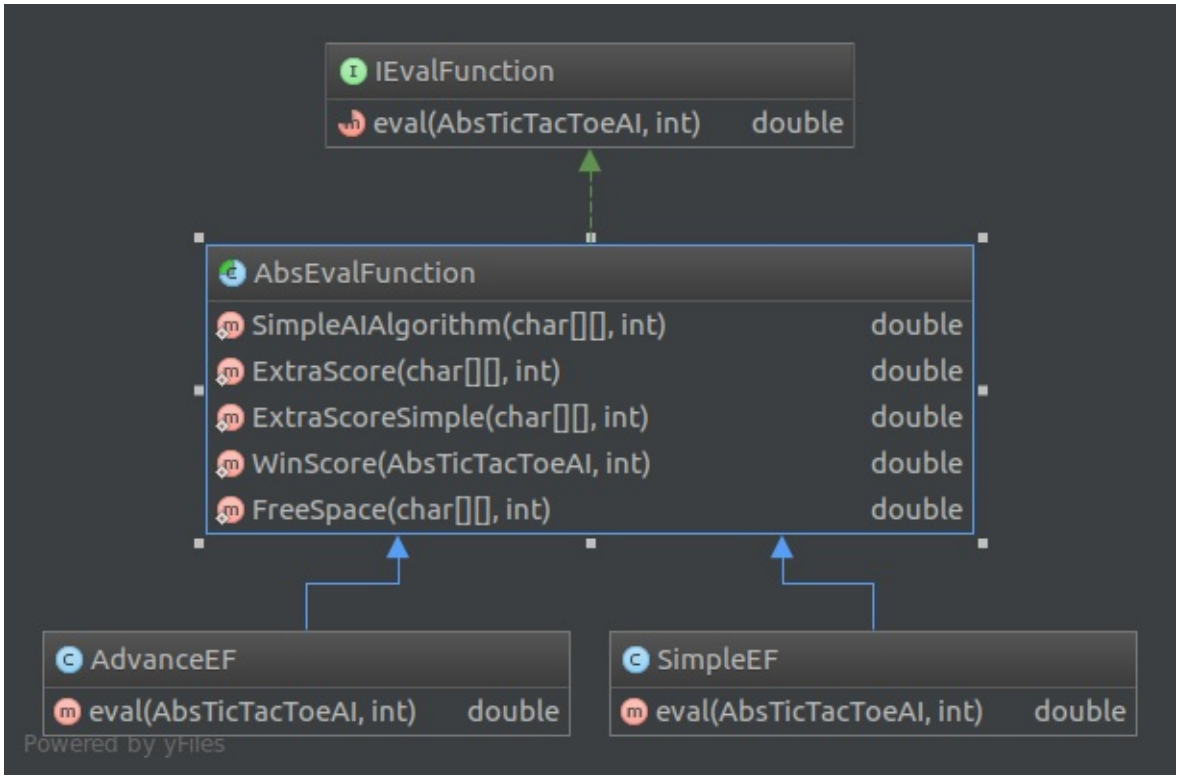
Il package *AI.Algorithms*



È costituito da una gerarchia di classi implementanti le quattro versioni dell'algoritmo *Minimax* utilizzate dall'intelligenza, enumerate qui di seguito:

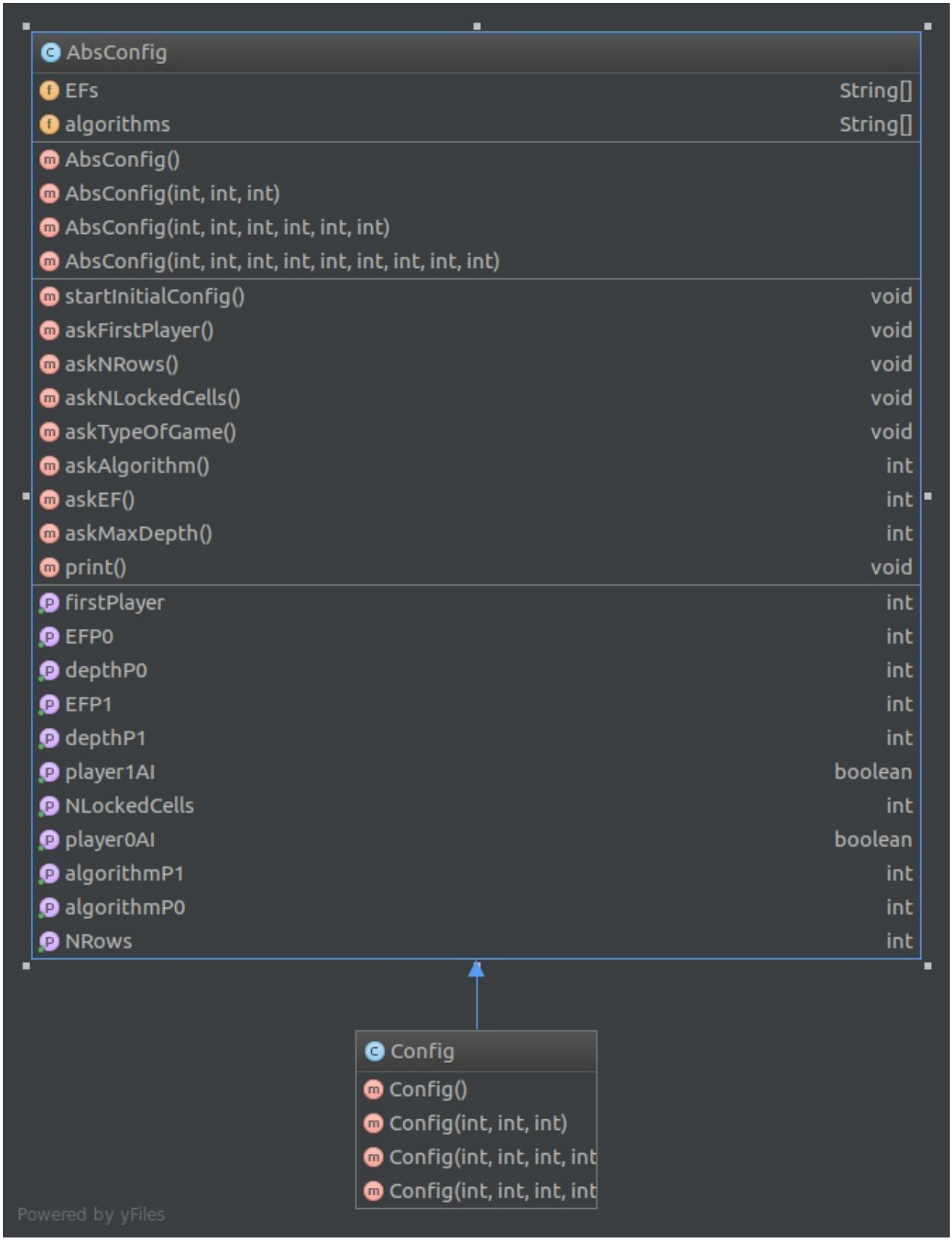
1. *Minimax* (classe ***Minimax***) standard, ossia l'algoritmo di base senza alcun miglioramento in termini di efficienza;
2. *Minimax* standard con rottura delle simmetrie (classe ***MinimaxRot***), ossia l'algoritmo senza *alpha-beta pruning* ma con una *lookup table* che permette di evitare l'espansione di nodi rappresentanti rotazioni di campi già visitati;
3. *Minimax* con *alpha-beta pruning* (classe ***MinimaxABP***) e senza rottura delle simmetrie;
4. *Minimax* con *alpha-beta pruning* e rottura delle simmetrie (classe ***MinimaxABPRot***).

Il package *AI.EF*



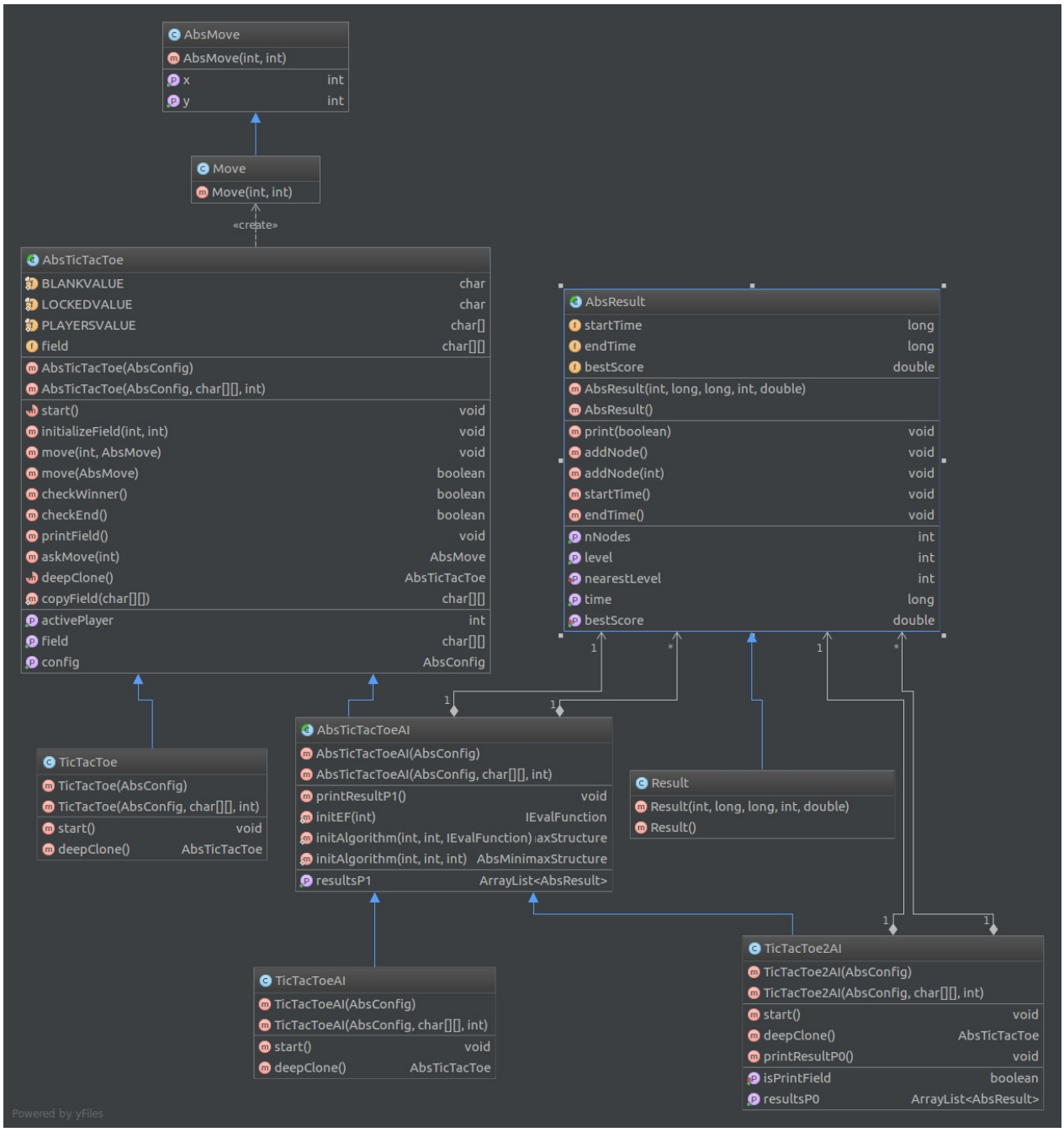
Questo *package* contiene due classi (implementanti la stessa interfaccia), una per ciascuna funzione di valutazione euristica utilizzata dall'intelligenza.

Il package *config*



Costituito da una semplice gerarchia verticale con una classe astratta come padre, le classi di questo *package* si occupano di salvare i dati inseriti dall'utente all'inizio del gioco. I dati richiesti all'utente sono stati presentati nel capitolo "*Interfaccia di gioco*".

Il package *ticTacToe*



- La gerarchia contenente **AbsTicTacToe** e **TicTacToe** rappresenta tramite i suoi campi dati alcune delle caratteristiche necessarie alla creazione di un gioco completo, quali i caratteri utilizzati dai giocatori, il giocatore corrente ed il campo da gioco. La classe astratta della gerarchia implementa inoltre alcuni metodi utili per la stampa del campo e per l'esecuzione di una mossa a partire da una configurazione data;
- La gerarchia contenente **AbsMove** e **Move** è semplicemte un *wrapper* di una coppia di coordinate rappresentanti una mossa sul campo di gioco;
- La gerarchia contenente **AbsResult** e **Result** si occupa della raccolta dei dati statistici

prodotti in una singola esecuzione del programma. Questi dati comprendono il numero di nodi generati, il tempo di esecuzione, la profondità della soluzione meno profonda e il miglior punteggio ottenuto.

Algoritmi di Intelligenza Artificiale

Varianti di Minimax

I *Minimax* semplice e con *pruning* seguono fedelmente lo pseudocodice contenuto nel libro utilizzato nel corso, mentre le versioni con il controllo dei campi ruotati sono state implementate da zero come spiegato nel paragrafo seguente.

Controllo delle rotazioni

Le rotazioni possibili del campo sono tre:

1. *90 gradi a destra*, realizzata tramite la trasposizione (matriciale) del campo e la conseguente inversione delle colonne;
2. *90 gradi a sinistra*, realizzata tramite la trasposizione del campo e la conseguente inversione delle righe;
3. *180 gradi*, realizzata tramite l'inversione delle righe e la conseguente inversione delle colonne.

Il *software* mantiene dunque una tavola di *lookup* con le configurazioni di gioco raggiunte fino all'istante corrente, permettendo così di sapere se un nuovo stato candidato all'espansione va espanso effettivamente: in pratica, se una qualsiasi rotazione di questo candidato esiste già nella tavola, lo stato candidato non viene espanso (ovviamente questi controlli riducono la dimensione dell'albero ma provocano tempi di esecuzione molto maggiori). Si tratta di un controllo molto semplice, contenuto nello *snippet* seguente:

```
boolean matchFound = mop.checkExistence(currFieldConf); //esegue le rotazioni e cerca
nella tavola di lookup
if (matchFound == false){     //la chiamata ricorsiva viene eseguita solo in caso di a
ssenza di match
   double max = maxValue(newState, depthP - 1);
   if (max < v){
       v = max;
   }
}
```

Funzioni di valutazione

Euristica semplice

Questa funzione di valutazione dà un punteggio al campo visitato unicamente in base al numero di caratteri allineati del giocatore interessato. In sostanza, se in un campo il numero di caratteri allineati su una riga, colonna o diagonale è maggiore del numero corrispondente in un altro campo, il primo campo riceve un punteggio maggiore. Questa euristica non si preoccupa minimamente di bloccare le mosse dell'avversario, portando dunque a performance spesso scarse da parte del giocatore computer. Nello *snippet* seguente viene implementato il controllo di continuità ed il conteggio dei segni sulle righe:

```
for(int column = 0; column < n; column ++){
    int cRow = 0;
    for(int row = 0; row < n; row ++){
        if(field[row][column] == AbsTicTacToe.PLAYERSVALUE[player]){
            cRow ++;
        } else{
            if(field[row][column] != AbsTicTacToe.BLANKVALUE) {
                cRow = 0;
                break;
            }
          }
     }
     if(cRow > maxVal){
        maxVal = cRow;
     }
}
```

Euristica avanzata

Questa funzione di valutazione assegna punteggi maggiori, nell'ordine, alla posizione centrale e agli angoli, e successivamente calcola il punteggio contando il numero di segni allineati (come per l'euristica precedente). Tuttavia, a differenza del caso semplice, qui viene considerato anche il punteggio dell'avversario: utilizzando questa strategia è possibile notare come l'intelligenza sia in grado di difendersi bloccando le mosse intelligenti dell'avversario.

Consuntivo orario

Questo progetto inizialmente prevedeva la partecipazione di tre persone, ma uno dei tre componenti ha lasciato il gruppo dopo la fase di progettazione, pertanto i due membri rimasti hanno dovuto eseguire del lavoro aggiuntivo.

Il totale delle ore dedicate al progetto ammonta a circa 32 a testa, così distribuite:

- **Fase di progettazione**: 6 ore;
- **Fase di codifica**: 21 ore;
- **Fase di verifica**: 5 ore.

Davide Rigoni si è occupato principalmente della logica del gioco e delle euristiche, mentre Giovanni Mazzocchin ha lavorato sulle varianti di *Minimax* e sulla relazione finale.

Estratti di codice

Di seguito vengono presentati succintamente i codici di alcuni algoritmi facenti parte dell'intelligenza del progetto.

Funzione *minValue* standard

```
    protected double minValue(TicTacToe state, int depthP) {
        this.res.addNode();

        if (state.checkEnd() || depthP == 0){
            double fvalue = this.getF().eval(state);

            res.setBestScore(fvalue);
            res.setNearestLevel(this.getDepth() - depthP -1);
            return fvalue;
        }
        double v = Double.POSITIVE_INFINITY;
        ArrayList< AbsMove> actions = AIUtils.computeActions(state.getField());

        for (int i = 0; i < actions.size(); i++){
            TicTacToe newState = state.deepClone();
            newState.move(actions.get(i));
            double max = maxValue(newState, depthP - 1);
            if (max < v){
                v = max;
            }
        }

        return v;
    }
```

Funzione *minValue* con *pruning*

```
    protected double minValue(TicTacToe state, double alpha, double beta, int depthP)
{
        this.res.addNode();

        if (state.checkEnd() || depthP == 0){
            double fvalue = this.getF().eval(state);

            res.setBestScore(fvalue);
            res.setNearestLevel(this.getDepth() - depthP -1);
            return fvalue;
        }

        double v = Double.POSITIVE_INFINITY;
        ArrayList< AbsMove> actions = AIUtils.
                computeActions(state.getField());
        for (int i = 0; i < actions.size(); i++){
            TicTacToe newState = state.deepClone();
            newState.move(actions.get(i));

            double max = maxValue(newState, alpha, beta, depthP - 1);
            if (max < v){
                v = max;
            }

            if (v <= alpha)
                return v;
            if (v < beta)
                beta = v;
        }

        return v;
    }
```

Funzione euristica avanzata

```
    public double eval(AbsTicTacToe state) {
        char[][] field = state.getField();
        int length = field.length;
        double result = 0;

        if(length%2 == 1){
            int index = (length/2);
            if(field[index][index] == AbsTicTacToe.PLAYERSVALUE[1]){
                result = result + length;
            } else{
                if(field[index][index] == AbsTicTacToe.PLAYERSVALUE[0]){
                    result = result - length;
                }
            }
        }

        if(field[0][0] == AbsTicTacToe.PLAYERSVALUE[1]){
            result = result + 0.25;
        }else{
            if(field[0][0] == AbsTicTacToe.PLAYERSVALUE[0])
                result = result - 0.25;
        }
        }
        [...]

        if(state.checkEnd()){
            if(state.checkWinner()){
                if(state.getActivePlayer() == 1) {
                    return length * 4 + result;
                }else{
                    return -(length * 4) + result;
                }
            } else{
                return 0;
            }
        } else{

            //Evaluate the fields from both the AI and player
            double valAI = AIUtils.SimpleAIAlgorithm(field,1);
            double valPlayer = AIUtils.SimpleAIAlgorithm(field,0);

            return result + valAI - valPlayer ;
        }

    }
```

Alcuni dati numerici

Presentiamo qui di seguito alcune statistiche sulle performance del programma ottenute considerando come avversari due intelligenze artificiali.

Confronto tra euristiche

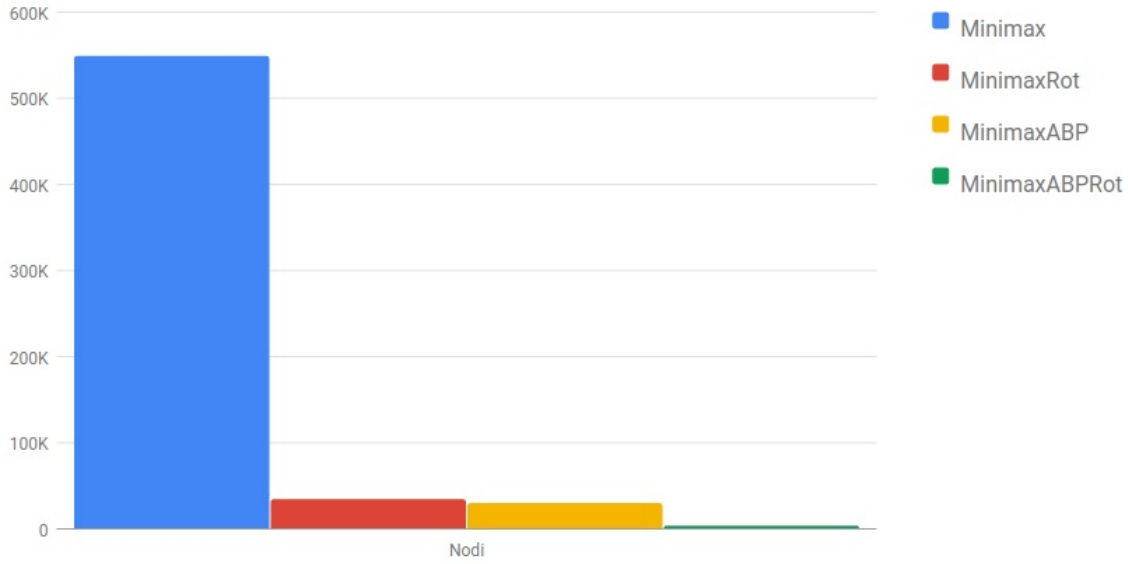
Abbiamo testato le due euristiche associate al *minimax* standard con profondità pari a due su un campo 3x3: facendole giocare una contro l'altra il risultato finale è di **patta**.

Giocando a profondità significativamente diverse tra loro si nota come il risultato finale sia di **vittoria** da parte dell'euristica associata alla profondità maggiore.

Confronto tra algoritmi su campo 3x3

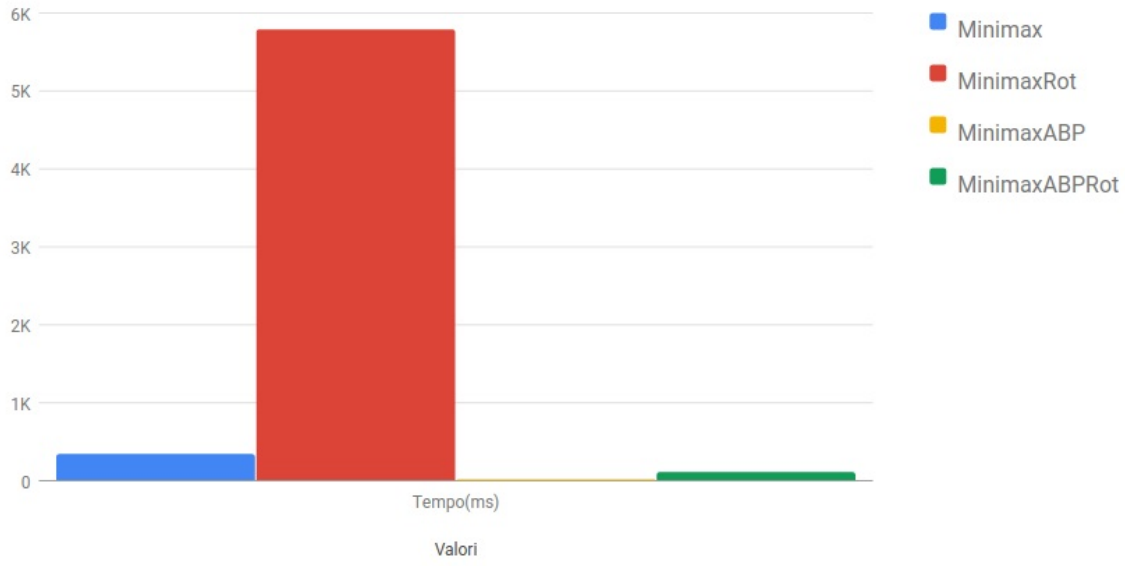
Nodi generati

- **Minimax 0**: 549946;
- **Minimax 1**: 31607;
- **Minimax 2**: 30710;
- **Minimax 3**: 4728.



Tempo (ms)

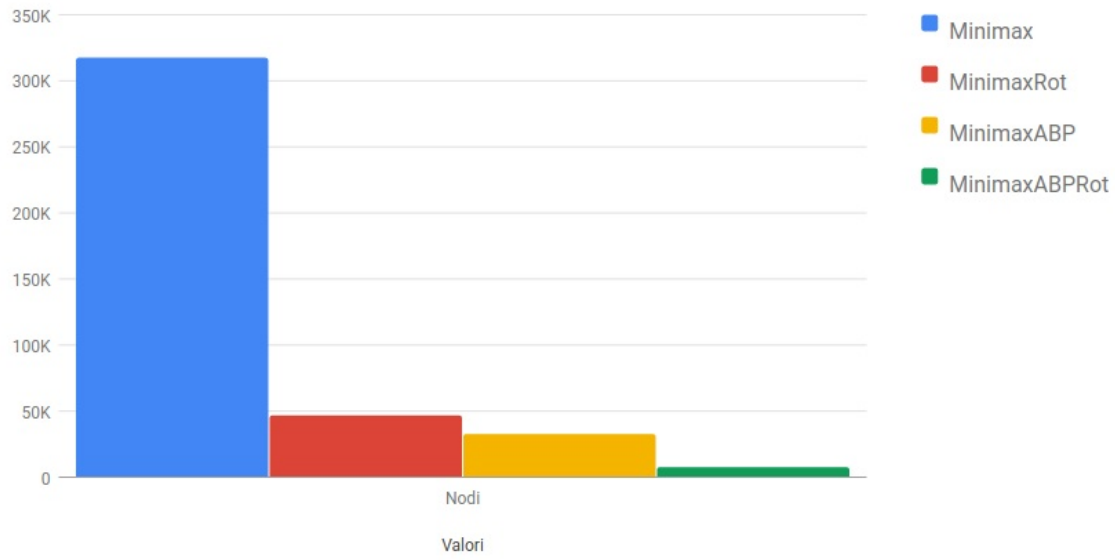
- **Minimax 0**: 351.6;
- **Minimax 1**: 5799.5;
- **Minimax 2**: 19.6;
- **Minimax 3**: 120.1.



Confronto tra algoritmi su campo 5x5 (profondità 4)

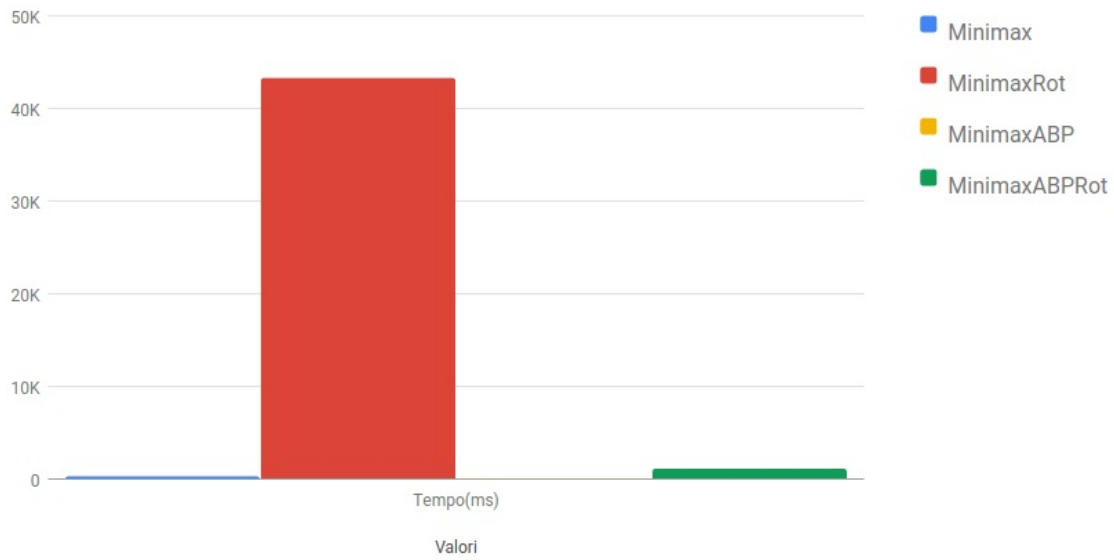
Nodi generati

- **Minimax 0**: 318026;
- **Minimax 1**: 47339;
- **Minimax 2**: 33207;
- **Minimax 3**: 8161.



Tempo (ms)

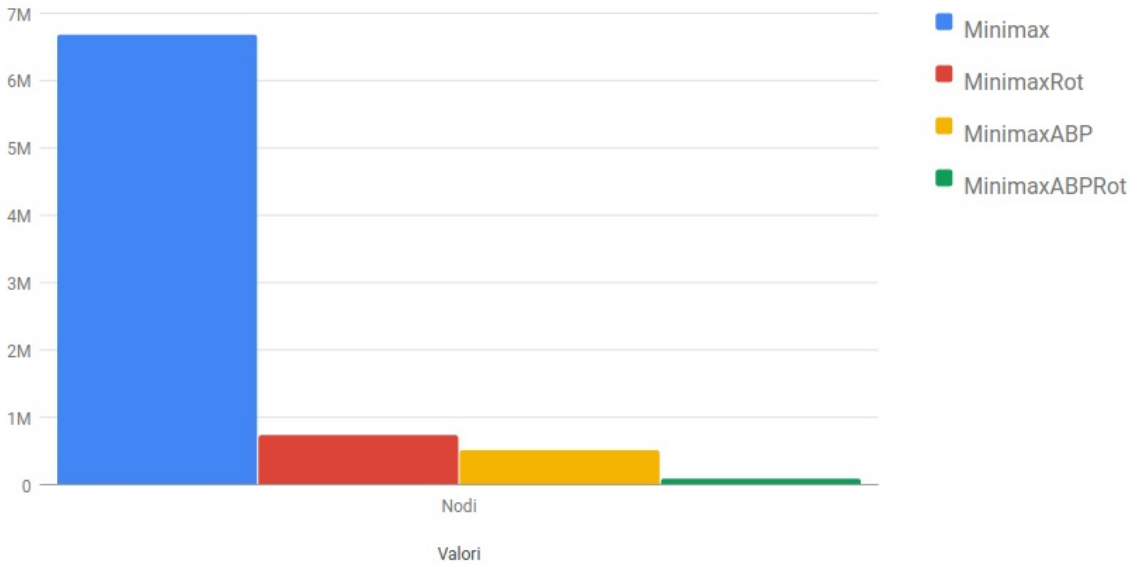
- **Minimax 0**: 287.5;
- **Minimax 1**: 43359.6;
- **Minimax 2**: 31.3;
- **Minimax 3**: 1186.5.


50K
40K
30K
20K
10K
0
Tempo(ms)
Valori
Minimax
MinimaxRot
MinimaxABP
MinimaxABPRot

Confronto tra algoritmi su campo 5x5 (profondità 5)

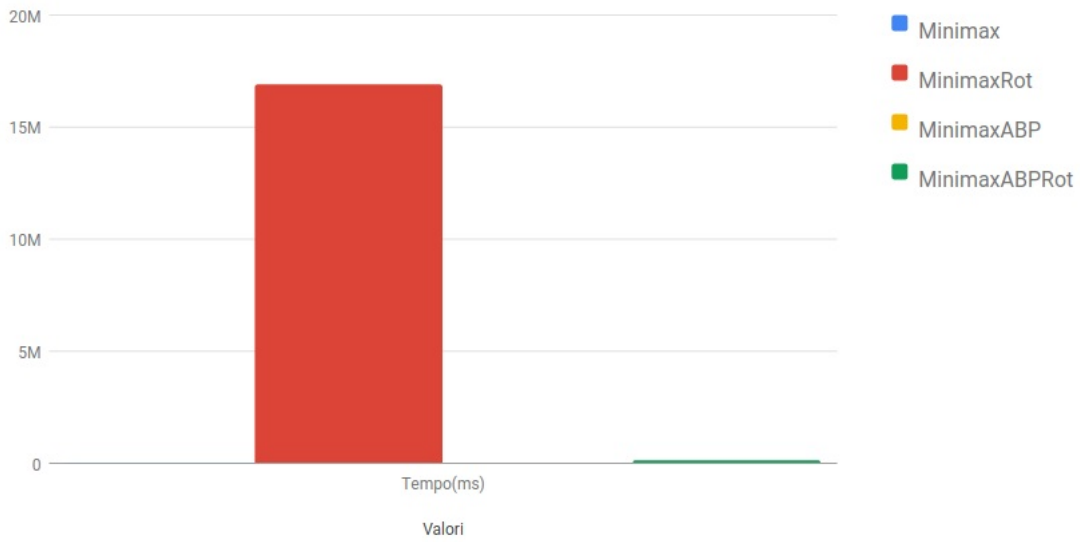
Nodi generati

- **Minimax 0**: 6693626;
- **Minimax 1**: 746402;
- **Minimax 2**: 523291;
- **Minimax 3**: 99598.



Tempo (ms)

- **Minimax 0**: 7667.3;
- **Minimax 1**: 16935842.3;
- **Minimax 2**: 571.8;
- **Minimax 3**: 164668.2.


20M
15M
10M
5M
0
Tempo(ms)
Valori
Minimax
MinimaxRot
MinimaxABP
MinimaxABPRot