

Minimax e Forza-4

Progetto per il corso di Intelligenza Artificiale

Daniele Turato¹

2 settembre 2010

¹M: 621522 - E: dturato@studenti.math.unipd.it

Indice

1	Introduzione	2
1.1	Stima dei tempi del progetto	2
1.2	Precisazioni sui sorgenti	2
2	Alpha-beta pruning	3
2.1	Condizioni di test	7
2.2	Disattivazione del pruning	7
2.3	Modifica della soglia	7
2.4	Ordine delle mosse	9
2.5	Ricerca di quiescenza	10
2.6	Risultati	14
3	La funzione euristica	15
3.1	Versione originale	15
3.2	Modifiche	20
3.3	Risultati	25
4	Conclusioni e possibili sviluppi	27
	Bibliografia	28

Capitolo 1

Introduzione

L'idea di base del progetto è quella di sperimentare le varie caratteristiche dell'algoritmo minimax, applicato nella fattispecie al gioco forza-4 (detto anche 4-in-a-row o Connect-4). Per effettuare questi esperimenti siamo partiti da una versione del gioco realizzata nel linguaggio Java da Sean Bridges, e reperibile all'indirizzo <http://www.it.uu.se/edu/course/homepage/ai/vt07/C4.zip>. Si sono rese necessarie diverse modifiche iniziali dei sorgenti originari, principalmente al fine di adattarli a far “scontrare” tra loro intelligenze artificiali con caratteristiche diverse, per rendere facilmente configurabili quest'ultime e per salvare i risultati su file csv in modo da poterli organizzare e analizzare con maggior comodità. Per quel che riguarda la produzione dei file csv ci siamo affidati alla libreria open source [OpenCsv](#).

1.1 Stima dei tempi del progetto

La realizzazione del progetto ha richiesto circa 100 ore, suddivise nel modo seguente:

- 50 ore per la comprensione del funzionamento dell'applicazione e per la realizzazione delle modifiche necessarie alla fase successiva (gioco con scontro tra intelligenze artificiali, salvataggio dati csv);
- 40 ore per la realizzazione della parte sperimentale sul pruning e per l'implementazione della ricerca di quiescenza e dell'euristica avanzata;
- 10 ore per riordinare i risultati e stendere la relazione.

1.2 Precisazioni sui sorgenti

Al fine di documentare con maggior precisione le operazioni di modifica effettuate rispetto al programma originale, ciascuna di esse è stata commentata in dettaglio contestualmente nel file sorgente stesso. Al fine di rendere più rapida la ricerca di tali modifiche all'interno dei file si sono marcati i commenti in modo differente:

- i classici “//” oppure “/*” sono stati utilizzati per i commenti utili a comprendere il funzionamento del programma;
- “//+” è stato utilizzato come marcatore all'inizio dei commenti che riguardano una modifica del codice originale.

Capitolo 2

Alpha-beta pruning

Nel codice 2.1 possiamo osservare il cuore del programma stesso, ossia l'implementazione dell'algoritmo minimax. In particolare le ultime due funzioni (*expandMaxNode* e *expandMinNode*) implementano l'espansione e l'analisi dei nodi per il giocatori min e max.

Codice 2.1: il file *MinimaxPlayer.java*

```
1 import java.io.FileWriter;
import java.io.IOException;
3
import au.com.bytecode.opencsv.CSVWriter;
5
// MinimaxPlayer.java
7
9 /**
 *
 * @author Sean Bridges
 * @version 1.0
13
 * The MinimaxPlayer uses the minimax algorithm to determine what move it should
15 * make. Looks ahead depth moves. The number of moves will be on the order
 * of  $n^{\text{depth}}$ , where  $n$  is the number of moves possible, and depth is the number
17 * of moves the engine is searching. Because of this the minimax player periodically
 * polls the thread that calls getMove(..) to see if it was interrupted. If the thread
19 * is interrupted, the player returns null after it checks.
 */
21 public class MinimaxPlayer extends DefaultPlayer
{
23
//-----
25 //instance variables
27 //the number of levels minimax will look ahead
private int depth = 1;
29 private Player minPlayer;
//+ aggiunto tipo giocatore odd/even
31 public boolean oddeven = false;
//+
33
//-----
35 //constructors
37 /** Creates new MinimaxPlayer */
public MinimaxPlayer(String name, int number, Player minPlayer)
39 {
    super(name, number);
41
    this.minPlayer = minPlayer;
43
}
45
// *INIZIO*
// aggiungo un costruttore senza l'avversario, per risolvere
47 // il problema in C4Applet (altrimenti non posso dichiarare
// due giocatori artificiali
49
public MinimaxPlayer(String name, int number)
```

```

51         {
52             super(name, number);
53         }
54
55         // *FINE*
56     }
57     //instance methods
58
59     //+ metodo necessario a impostare il giocatore avversario durante l'inizializzazione
60     public void setMinPlayer(Player minPlayer)
61     {
62         this.minPlayer = minPlayer;
63     }
64     //+
65
66     /**
67      * Get the number of levels that the Minimax Player is currently looking
68      * ahead.
69      */
70     public int getDepth()
71     {
72         return depth;
73     }
74
75     /**
76      * Set the number of levels that the Minimax Player will look ahead
77      * when getMove is next called
78      */
79     public void setDepth(int anInt)
80     {
81         depth = anInt;
82     }
83
84     /** Passed a copy of the board, asked what move it would like to make.
85      *
86      * The MinimaxPlayer periodically polls the thread that makes this call to
87      * see if it is interrupted. If it is the player returns null.
88      *
89      * Looks ahead depth moves.
90      */
91     public Move getMove(Board b)
92     {
93         MinimaxCalculator calc = new MinimaxCalculator(b, this, minPlayer);
94         //+ aggiunta parte necessaria a inserire un delay fra le mosse
95         try
96         {
97             Thread.sleep(this.movedelay);
98         }
99         catch (InterruptedException ex)
100         {
101             System.out.println("Problemi nella gestione del thread...");
102         }
103         //+
104         return calc.calculateMove(depth);
105     }
106
107 }//end MinimaxPlayer
108
109 /**
110  * The MinimaxCalculator does the actual work of finding the minimax move.
111  * A new calculator should be created each time a move is to be made.
112  * A calculator should only be used once.
113  */
114 final class MinimaxCalculator
115 {
116
117     //-----
118
119     //instance variables
120
121     //the number of moves we have tried
122     private int moveCount = 0;
123     private long startTime;
124
125     private Player minPlayer;
126     private Player maxPlayer;

```

```

127     private Board board;

129     private final int MAX_POSSIBLE_STRENGTH;
130     private final int MIN_POSSIBLE_STRENGTH;

131
132     //-----
133     //constructors
134     MinimaxCalculator(Board b, Player max, Player min)
135     {
136         board = b;
137         maxPlayer = max;
138         minPlayer = min;
139
140         MAX_POSSIBLE_STRENGTH = board.getBoardStats().getMaxStrength();
141         MIN_POSSIBLE_STRENGTH = board.getBoardStats().getMinStrength();
142     }
143
144     //-----
145     //instance methods
146
147     /**
148      * Calculate the move to be made.
149      */
150     public Move calculateMove(int depth)
151     {
152         startTime = System.currentTimeMillis();
153
154         if(depth == 0)
155         {
156             System.out.println("Error, 0 depth in minumax player");
157             Thread.dumpStack();
158             return null;
159         }
160
161         Move[] moves = board.getPossibleMoves(maxPlayer);
162         int maxStrength = MIN_POSSIBLE_STRENGTH;
163         int maxIndex = 0;
164
165         for(int i = 0; i < moves.length; i++)
166         {
167             if(board.move(moves[i]))
168             {
169                 moveCount++;
170
171                 int strength = expandMinNode(depth - 1, maxStrength);
172                 if(strength > maxStrength)
173                 {
174                     maxStrength = strength;
175                     maxIndex = i;
176                 }
177                 board.undoLastMove();
178             } //end if move made
179
180             //if the thread has been interrupted, return immediately.
181             if(Thread.currentThread().isInterrupted())
182             {
183                 return null;
184             }
185
186         } //end for all moves
187
188         long stopTime = System.currentTimeMillis();
189         System.out.print("MINIMAX: Number of moves tried:" + moveCount);
190         System.out.println(" Time:" + (stopTime - startTime) + " milliseconds");
191
192         //+ aggiungo il codice necessario a tracciare su un file csv i dati
193
194         FileWriter fwriter;
195         try {
196             // notare il booleano a true per impostare l'append
197             fwriter = new FileWriter("dati_esecuzione.csv", true);
198             CSVWriter writer = new CSVWriter(fwriter, '\t');
199
200             // feed in your array (or convert your data to an array)
201             String[] entries = new String[2];
202             entries[0] = Integer.toString(moveCount);

```

```

203         entries[1] = Long.toString(stopTime - startTime);
204         writer.writeNext(entries);
205         writer.close();
206     } catch (IOException e) {
207         // TODO Auto-generated catch block
208         e.printStackTrace();
209     }
210
211     //+
212
213     return moves[maxIndex];
214
215 }
216
217 /**
218  * A max node returns the max score of its descendants.
219  * parentMinimum is the minimum score that the parent has already encountered.
220  * if we find a score that is higher than this, we will return that score
221  * immediately rather than continue to expand the tree, since
222  * the min node above us only cares if we are lower than its current
223  * min score.
224  */
225 private int expandMaxNode(int depth, int parentMinimum)
226 {
227     //base step
228     if(depth == 0 || board.isGameOver())
229     {
230         return board.getBoardStats().getStrength(maxPlayer);
231     }
232
233     //recursive step
234     Move[] moves = board.getPossibleMoves(maxPlayer);
235     int maxStrength = MIN_POSSIBLE_STRENGTH;
236
237     for(int i = 0; i < moves.length; i++)
238     {
239         if(board.move(moves[i]))
240         {
241             moveCount++;
242             int strength = expandMinNode(depth - 1, maxStrength);
243
244             // $ commentando il seguente if (e il corrispondente in
245             //   expandMinNode) disattiviamo il pruning
246             if(strength > parentMinimum)
247             {
248                 board.undoLastMove();
249                 return strength;
250             }
251
252             if(strength > maxStrength)
253             {
254                 maxStrength = strength;
255             }
256             board.undoLastMove();
257             //end if move made
258         }
259     }
260     //end for all moves
261
262     return maxStrength;
263 }
264
265 /**
266  * The min node chooses the smallest of its descendents.
267  * parentMaximum is the maximum that the parent max node has already found,
268  * if we find something smaller than this, return immediatly, since the
269  * parent max node will choose the greatest value it can find.
270  */
271 private int expandMinNode(int depth, int parentMaximum)
272 {
273     //base step
274     if(depth == 0 || board.isGameOver())
275     {
276         return board.getBoardStats().getStrength(maxPlayer);
277     }

```

```

279      //recursive step
280      Move[] moves = board.getPossibleMoves(minPlayer);
281
282      // $ questa parte Ã quella che implementa il pruning
283      int minStrength = MAX_POSSIBLE_STRENGTH;
284
285      for(int i = 0; i < moves.length; i++)
286      {
287          if(board.move(moves[i]))
288          {
289              moveCount++;
290              int strength = expandMaxNode(depth -1, minStrength);
291
292              // $ commentando il seguente if (e il corrispondente in
293              //      expandMaxNode) disattiviamo il pruning
294              if(strength < parentMaximum)
295              {
296                  board.undoLastMove();
297                  return strength;
298              }
299
300              if(strength < minStrength)
301              {
302                  minStrength = strength;
303              }
304              board.undoLastMove();
305          } //end if move made
306
307      } //end for all moves
308
309      return minStrength;
310      // $
311  } //end expandMaxNode
312
313  }

```

2.1 Condizioni di test

Impostiamo sempre i test in modo da utilizzare due intelligenze artificiali che arrivano fino a 6-ply. Da notare che in diverse esecuzioni con le stesse caratteristiche, mentre il numero di mosse analizzate rimane costante, i tempi di esecuzione cambiano (in particolare la prima misurazione risulta poco significativa, e segnala tempi molto più alti).

2.2 Disattivazione del pruning

Se osserviamo il listato precedente alle righe da 245 a 249 e da 293 a 297 possiamo notare gli spezzoni di codice che permettono di effettuare il pruning. Un primo esperimento effettuato è stato quello di commentare tali spezzoni al fine di verificare l'efficacia di tale meccanismo. Nella tabella 2.1 possiamo osservare il numero di mosse analizzate e il tempo necessario ad effettuare la scelta della contromossa per ogni prova effettuata: in particolare nella prima colonna possiamo osservare i risultati ottenuti disattivando il pruning.

2.3 Modifica della soglia

Osservando nuovamente il codice dei test alle righe specificate in precedenza possiamo notare come questi non considerino l'uguaglianza: si può intuire quindi come aggiungendo l'uguaglianza sia possibile aumentare ancora il numero di nodi tagliati, rendendo il pruning più efficace. Una conferma di questo è visibile alla terza colonna della tabella 2.1.

Mossa	Pruning disattivato		Pruning attivato		Con uguaglianza		Prima mosse centrali	
	Possibilità analizzate	Tempo (ms)	Possibilità analizzate	Tempo (ms)	Possibilità analizzate	Tempo (ms)	Possibilità analizzate	Tempo (ms)
1	299592	817	73221	33	46657	22	13249	8
2	299591	139	53853	27	39621	18	15881	8
3	297182	141	52176	25	40738	32	7946	4
4	297531	152	43662	23	38340	20	11700	9
5	296320	139	66915	32	48562	23	26101	15
6	298400	153	60105	32	55681	28	12431	7
7	285244	137	40189	20	35220	18	14443	7
8	278024	145	39424	20	33702	21	18576	10
9	270464	134	29989	15	24148	12	16058	8
10	228469	123	7803	4	6936	4	29877	17
11	287062	148	43209	22	35621	18	9161	5
12	276755	200	36431	23	30997	16	14730	8
13	269688	132	46348	27	34529	20	30835	15
14	221171	114	39630	20	29427	16	33426	19
15	217964	108	30985	15	8504	6	27878	14
16	209946	110	44640	27	29657	16	27262	14
17	180614	89	113210	57	22138	11	89556	48
18	221241	121	36357	21	28774	15	17598	10
19	187715	92	41781	22	29951	15	27131	14
20	186152	99	31054	17	25069	13	22555	12
21	145766	78	52457	27	7187	4	38121	21
22	128223	71	26370	16	3319	2	22291	12
23	136264	70	63324	34	4136	2	61775	32
24	96248	52	5148	3	1082	0	19095	13
25	111394	57	62691	53	2067	1	56663	30
26	46461	26	13289	7	366	0	14920	10
27	35787	21	25271	15	220	0	25703	14
28	34076	19	12190	7	29	0	14531	12
29	15641	10	9248	5	22	0	10372	6
30	9357	6	11	0	11	0	265	0

Figura 2.1: tabella riassuntiva dei risultati

2.4 Ordine delle mosse

Nella quarta colonna della tabella 2.1 possiamo osservare come sia effettivamente possibile incrementare ulteriormente le prestazioni del pruning: sappiamo infatti che l'alpha-beta pruning ha la sua massima efficacia se le possibili mosse vengono analizzate in ordine di utilità, ossia se analizziamo prima le mosse con il valore di utilità maggiore. Il problema è quello di sapere in anticipo quali siano tali mosse: in forza-4 un semplice principio afferma che conviene sempre cercare di conquistare prima le posizioni centrali. Il codice qui sotto rappresenta la funzione di espansione per il giocatore Max: alla riga 21 abbiamo sostituito l'indice i con la variabile *index*, il cui valore iniziale è quello della prima colonna centrale, ossia 3. Tale valore viene poi aggiornato a ogni iterazione del ciclo tramite le righe di codice dalla 38 alla 42: tali righe fanno in modo che il valore di *index* segua la successione 3, 4, 2, 5, 1, 6, 0, 7 (in questo modo la ricerca si sposta man mano verso le mosse che posizionano la pedina sulle colonne più laterali). Ancora una volta possiamo osservare un calo del numero di possibilità analizzate e dei tempi impiegati per il calcolo.

Codice 2.2: modifica della funzione `expandMaxNode` del file `MinimaxPlayer.java` per favorire le mosse centrali

```
private int expandMaxNode(int depth, int parentMinimum)
{
    //base step
    if(depth == 0 || board.isGameOver())
    {
        return board.getBoardStats().getStrength(maxPlayer);
    }

    //recursive step
    Move[] moves = board.getPossibleMoves(maxPlayer);
    int maxStrength = MIN_POSSIBLE_STRENGTH;
    //+ modifica per scegliere prima le mosse centrali
    int index = 3;

    for(int i = 0; i < moves.length; i++)
    {
        //+ modifica per scegliere prima le mosse centrali
        //+ if(board.move(moves[i]))

        if(board.move(moves[index]))
        {
            moveCount++;
            int strength = expandMinNode(depth - 1, maxStrength);

            if(strength > parentMinimum)
            {
                board.undoLastMove();
                return strength;
            }
            if(strength > maxStrength)
            {
                maxStrength = strength;
            }
            board.undoLastMove();
        } //end if move made

        //+ modifica per scegliere prima le mosse centrali
        if ((i % 2) == 0)
            index = index + (i+1);
        else
            index = index - (i+1);

    } //end for all moves

    return maxStrength;
} //end expandMaxNode
```

2.5 Ricerca di quiescenza

Come è noto non è possibile visitare tutto l'albero delle mosse risultante dall'approccio Minimax, poichè il tempo richiesto sarebbe enorme. Come abbiamo visto l'applicazione descritta permette di limitare l'analisi a un ben preciso numero di livelli dell'albero stesso, specificando tale valore tramite un menù prima di iniziare la partita. Il problema di questa tecnica è che anche se un cammino sembra promettente fino al punto in cui avviene il taglio e si richiama la funzione di valutazione, può darsi che in realtà al livello successivo avremmo ottenuto dei valori che avrebbero abbassato di molto l'utilità complessiva del suddetto cammino, causando la scelta di un'altra mossa da parte del giocatore. Un possibile rimedio a questo fenomeno è l'utilizzo di una ricerca di quiescenza: come possiamo osservare nel codice 2.3, nel momento in cui viene effettuata la valutazione del livello subito precedente al taglio, una volta ottenuti i valori dalla funzione di utilità per tutte e 8 le possibili mosse, anzichè restituirli direttamente al chiamante, si selezionano i due valori più promettenti e si aziona un ulteriore passo ricorsivo che approfondisce ulteriormente l'analisi. In questo modo se sono presenti dei crolli dei valori di utilità ai livelli subito sottostanti (nel nostro caso abbiamo scelto di approfondire ancora di 3 ply), questi vengono rilevati, e il valore restituito sarà quindi alla fine più basso, permettendo una valutazione più corretta da parte dei livelli superiori. Tutto questo viene effettuato mantenendo comunque accettabile il livello di risorse richiesto, in quanto la ricerca non approfondisce la valutazione dell'albero per ogni nodo foglia, ma solo per i più promettenti.

Codice 2.3: funzioni modificate per l'implementazione della ricerca di quiescenza (file *MinimaxPlayer_qs.java*)

```
1 public Move calculateMove(int depth)
2 {
3     startTime = System.currentTimeMillis();
4
5     if(depth == 0)
6     {
7         System.out.println("Error, 0 depth in minumax player");
8         Thread.dumpStack();
9         return null;
10    }
11
12    Move[] moves = board.getPossibleMoves(maxPlayer);
13    int maxStrength = MIN_POSSIBLE_STRENGTH;
14    int maxIndex = 0;
15
16    for(int i = 0; i < moves.length; i++)
17    {
18        if(board.move(moves[i]))
19        {
20            moveCount++;
21
22            int strength = expandMinNode(depth - 1, maxStrength, false);
23            if(strength > maxStrength)
24            {
25                maxStrength = strength;
26                // questo è l'indice che permette di selezionare la mossa
27                // migliore
28                maxIndex = i;
29            }
30            board.undoLastMove();
31        } //end if move made
32
33        //if the thread has been interrupted, return immediately.
34        if(Thread.currentThread().isInterrupted())
35        {
36            return null;
37        }
38    } //end for all moves
39
40    long stopTime = System.currentTimeMillis();
41    System.out.print("MINIMAX: Number of moves tried:" + moveCount);
```

```

43     System.out.println(" Time:" + (stopTime - startTime) + " milliseconds");
44
45     //+ aggiungo il codice necessario a tracciare su un file csv i dati
46
47     FileWriter fwriter;
48     try {
49         // notare il booleano a true per impostare l'append
50         fwriter = new FileWriter("dati_esecuzione.csv", true);
51         CSVWriter writer = new CSVWriter(fwriter, '\t');
52
53         // feed in your array (or convert your data to an array)
54         String[] entries = new String[2];
55         entries[0] = Integer.toString(moveCount);
56         entries[1] = Long.toString(stopTime - startTime);
57         writer.writeNext(entries);
58         writer.close();
59     } catch (IOException e) {
60         // TODO Auto-generated catch block
61         e.printStackTrace();
62     }
63
64     //+
65
66     return moves[maxIndex];
67 }
68
69 /**
70  * A max node returns the max score of its descendants.
71  * parentMinimum is the minimum score that the parent has already encountered.
72  * if we find a score that is higher than this, we will return that score
73  * immediately rather than continue to expand the tree, since
74  * the min node above us only cares if we are lower than its current
75  * min score.
76  */
77 private int expandMaxNode(int depth, int parentMinimum, Boolean qflag)
78 {
79     //base step
80     if(depth == 0 || board.isGameOver())
81     {
82         return board.getBoardStats().getStrength(maxPlayer);
83     }
84
85     //recursive step
86     Move[] moves = board.getPossibleMoves(maxPlayer);
87     int maxStrength = MIN_POSSIBLE_STRENGTH;
88     //+ questa serve per memorizzare il secondo migliore valore
89     int secondMaxStrength = maxStrength;
90     //+ questi servono per memorizzare gli indici delle due migliori mosse
91     int primaScelta = 0, secondaScelta = 0;
92
93     for(int i = 0; i < moves.length; i++)
94     {
95         if(board.move(moves[i]))
96         {
97             moveCount++;
98             int strength = expandMinNode(depth - 1, maxStrength, qflag);
99
100             //$ commentando il seguente if (e il corrispondente in expandMinNode)
101             disattiviamo il pruning
102             if(strength > parentMinimum)
103             {
104                 board.undoLastMove();
105                 return strength;
106             }
107
108             if(strength > maxStrength)
109             {
110                 secondaScelta = primaScelta;
111                 secondMaxStrength = maxStrength;
112                 primaScelta = i;
113                 maxStrength = strength;
114             }
115             else if (strength < secondMaxStrength)
116             {
117                 secondaScelta = i;

```

```

117         secondMaxStrength = strength;
118     }
119     board.undoLastMove();
120 } //end if move made
121
122 } //end for all moves
123
124 //+ qflag serve ad evitare di ripetere per sempre la ricerca di quiescenza
125 if(depth == 1 && !qflag)
126 {
127     depth = 3;
128
129     for(int i = 0; i < moves.length; i++)
130     {
131         if((i == primaScelta || i == secondaScelta) && board.move(moves[i]))
132         {
133             moveCount++;
134             //+ qflag a true per evitare di eseguire la ricerca all'
135             //infinito
136             int strength = expandMinNode(depth - 1, maxStrength, true);
137
138             // $ commentando il seguente if (e il corrispondente in
139             // expandMaxNode) disattiviamo il pruning
140             if(strength > parentMinimum)
141             {
142                 board.undoLastMove();
143                 return strength;
144             }
145
146             if(strength > maxStrength)
147             {
148                 maxStrength = strength;
149             }
150
151             board.undoLastMove();
152         } //end if move made
153     } //end for all moves
154 }
155
156 return maxStrength;
157 } //end expandMaxNode
158
159
160 /**
161  * The min node chooses the smallest of its descendents.
162  * parentMaximum is the maximum that the parent max node has already found,
163  * if we find something smaller than this, return immediately, since the
164  * parent max node will choose the greatest value it can find.
165  */
166 private int expandMinNode(int depth, int parentMaximum, Boolean qflag)
167 {
168     //base step
169     if(depth == 0 || board.isGameOver())
170     {
171         return board.getBoardStats().getStrength(maxPlayer);
172     }
173
174     //+ proviamo a implementare la ricerca di quiescenza
175     //+ all'ultimo passo anzichè fermarci andiamo avanti per altre 4 ply
176     //+ in questo modo se il valore si modifica sarà restituito il nuovo
177     //+ valore
178
179
180     //recursive step
181     Move[] moves = board.getPossibleMoves(minPlayer);
182
183     // $ questa parte è quella che implementa il pruning
184     int minStrength = MAX_POSSIBLE_STRENGTH;
185     //+ questa serve per memorizzare il secondo migliore valore
186     int secondMinStrength = minStrength;
187     //+ questi servono per memorizzare gli indici delle due migliori mosse
188     int primaScelta = 0, secondaScelta = 0;

```

```

191  for(int i = 0; i < moves.length; i++)
192  {
193      if(board.move(moves[i]))
194      {
195          moveCount++;
196          int strength = expandMaxNode(depth -1, minStrength, qflag);
197
198          //$ commentando il seguente if (e il corrispondente in expandMaxNode)
199          disattiviamo il pruning
200          if(strength < parentMaximum)
201          {
202              board.undoLastMove();
203              return strength;
204          }
205
206          if(strength < minStrength)
207          {
208              secondaScelta = primaScelta;
209              secondMinStrength = minStrength;
210              primaScelta = i;
211              minStrength = strength;
212          }
213          else if (strength < secondMinStrength)
214          {
215              secondaScelta = i;
216              secondMinStrength = strength;
217          }
218
219          board.undoLastMove();
220      } //end if move made
221  } //end for all moves
222
223  //+ qflag serve ad evitare di ripetere per sempre la ricerca di quiescenza
224  if(depth == 1 && !qflag)
225  {
226      depth = 3;
227
228      for(int i = 0; i < moves.length; i++)
229      {
230          if((i == primaScelta || i == secondaScelta) && board.move(moves[i]))
231          {
232              moveCount++;
233              //+ qflag a true per evitare di eseguire la ricerca all'
234              infinito
235              int strength = expandMaxNode(depth -1, minStrength, true);
236
237              //$ commentando il seguente if (e il corrispondente in
238              expandMaxNode) disattiviamo il pruning
239              if(strength < parentMaximum)
240              {
241                  board.undoLastMove();
242                  return strength;
243              }
244
245              if(strength < minStrength)
246              {
247                  minStrength = strength;
248              }
249
250              board.undoLastMove();
251          } //end if move made
252      } //end for all moves
253
254  }
255
256  return minStrength;
257  //$
258  } //end expandMaxNode

```

2.6 Risultati

Questo approccio ha fornito buoni risultati nello scontro con l'intelligenza artificiale originale del gioco e, a parità di profondità, quest'ultima è risultata sempre battuta, come possiamo osservare in figura 2.2.

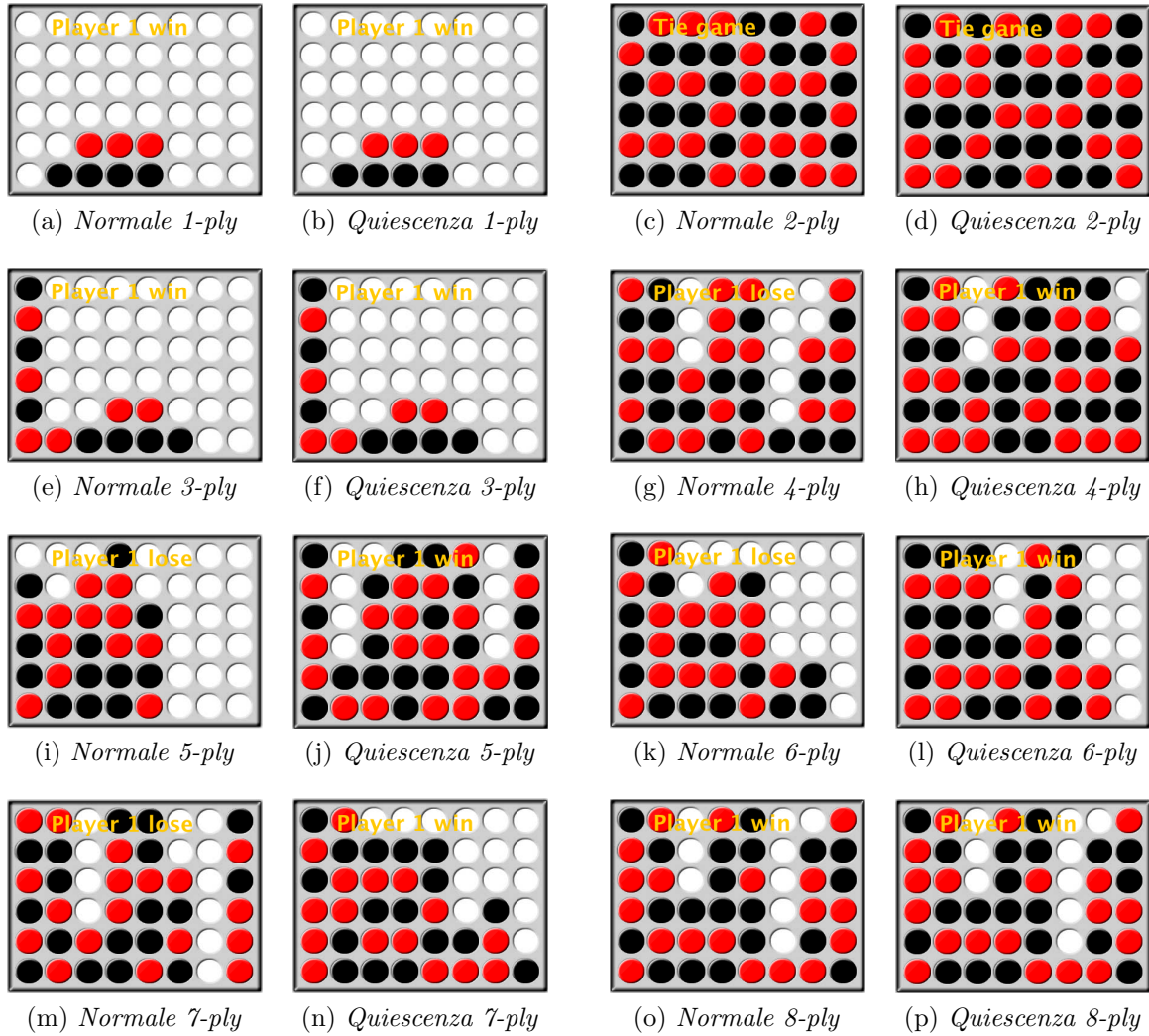


Figura 2.2: confronto fra le giocate nelle partite con entrambi i giocatori Minimax standard, e con primo giocatore (nero) con ricerca di quiescenza

Capitolo 3

La funzione euristica

3.1 Versione originale

Nel codice 3.1 possiamo osservare le 3 funzioni che stanno alla base dell'euristica utilizzata nella versione originale dell'applicazione: la funzione *getStrength*, applicata su uno dei due giocatori, ritorna la differenza in termini quantitativi tra il grado di bontà dell'insieme delle posizioni occupate da quest'ultimo e quello del giocatore avversario. Tale quantità viene calcolata restituendo il massimo valore possibile se risulta che il giocatore è riuscito a inserire 4 delle proprie pedine in un gruppo di 4 posizioni adiacenti. Altrimenti la quantità restituita viene calcolata sulla base della seguente formula:

$$strength = 32 \cdot n_3 + 4 \cdot n_2 + n_1$$

dove:

- n_3 è il numero di gruppi di 4 posizioni adiacenti nelle quali siano presenti 3 pedine del giocatore in questione;
- n_2 è il numero di gruppi di 4 posizioni adiacenti nelle quali siano presenti 2 pedine dello stesso;
- n_1 è il numero di gruppi di 4 posizioni adiacenti nelle quali sia presente 1 pedina dello stesso.

Per comprendere la corrispondenza di tale formula con il codice alle righe da 13 a 17 bisogna ricordare che l'operazione di shift a sinistra (" $<<$ ") di una posizione equivale a moltiplicare per 2 il numero stesso e che le costanti `SHIFT_3_IN_A_ROW` e `SHIFT_2_IN_A_ROW` valgono rispettivamente 5 e 2. La scelta di utilizzare l'operazione di shifting al posto della reale moltiplicazione consente di ottenere maggiori prestazioni nel calcolo stesso.

Codice 3.1: funzione di valutazione della situazione di un giocatore (file *C4Stats.java*)

```
private int firstPlayerStrength()  
2 {  
    if (p1_4InARow != 0)  
4    {  
        return Integer.MAX_VALUE;  
6    }  
    else if (p2_4InARow != 0)  
8    {  
        return 0;  
10   }  
    else  
12   {  
        return (p1_2InARow << SHIFT_2_IN_A_ROW) +  
14        (p1_3InARow << SHIFT_3_IN_A_ROW) +
```



```

16         }
17     }
18     private int secondPlayerStrength()
19     {
20         if (p2_4InARow != 0)
21         {
22             return Integer.MAX_VALUE;
23         }
24         else if (p1_4InARow != 0)
25         {
26             return 0;
27         }
28         else
29         {
30             return (p2_2InARow << SHIFT_2_IN_A_ROW) +
31                    (p2_3InARow << SHIFT_3_IN_A_ROW) +
32                    p2_1InARow;
33         }
34     }
35     public int getStrength(Player aPlayer)
36     {
37         if (aPlayer.getNumber() == PLAYER1)
38         {
39             return firstPlayerStrength() - secondPlayerStrength();
40         }
41         else
42         {
43             return secondPlayerStrength() - firstPlayerStrength();
44         }
45     }
46 }

```

Una volta compreso come l'euristica venga calcolata a partire dai parametri che descrivono la posizione delle pedine del giocatore, dobbiamo capire in che modo tali parametri vengano aggiornati ogni volta che viene effettuata una mossa da parte di uno dei giocatori.

Bisogna innanzitutto capire come i vari oggetti che fanno parte del programma contribuiscano al calcolo del valore di utilità della mossa. Quando viene creata una nuova board questa crea contestualmente anche un oggetto per il calcolo dell'euristica (*C4Stats*). Successivamente tramite la funzione *initSlots* vengono creati tutti i gruppi di 4 posizioni vincenti (ossia adiacenti) possibili (84 gruppi, si può facilmente verificare anche visivamente provando a tracciarli in una tabella e contandoli), ai quali viene assegnato il suddetto oggetto. Ogni gruppo non è altro che un oggetto *C4Row*: tale oggetto viene aggiunto come listener di ogni posizione (*C4Slot*) che fa parte del gruppo. Tutti i gruppi sono inseriti in un vettore. Quando il *GameMaster* ordina l'esecuzione della mossa scelta dal giocatore, non fa altro che richiamare il metodo *move* di *C4Board*, che utilizza il metodo *SetContents* di *C4Slot* per impostare il contenuto dello slot al numero del giocatore che ha scelto la mossa stessa. *SetContents* notifica il nuovo contenuto per ogni listener, tramite il metodo *notifyListenersContentsChanged*, sempre della classe *C4Slot*, il quale utilizza il metodo *contentsChanged* dei listener *C4Row*, di cui possiamo vedere l'implementazione nel codice 3.2. Questa funzione rappresenta il cuore dell'intero meccanismo di valutazione dello stato dei giocatori, poichè si occupa di aggiornare tutti i contatori dell'oggetto di calcolo *C4Stats* visto in precedenza, in modo che questo possa restituire la giusta valutazione. Per capire meglio possiamo fare un semplice esempio, corrispondente alle righe 20-22 del sorgente: se abbiamo un gruppo di 4 celle adiacenti nelle quali sappiamo che sono già presenti 2 pedine del primo giocatore e nessuna del secondo, nel momento in cui viene aggiunta un'altra pedina del primo giocatore dobbiamo decrementare il contatore relativo alle coppie per il giocatore 1 (*p1_2InARow*) e incrementare quello per le triplete (*p1_3InARow*).

Codice 3.2: funzione che aggiorna i campi necessari al calcolo dell'euristica (file *C4Row.java*)

```

2 public void contentsChanged(int oldContents, int newContents)
3 {
4     //case 1, moving from null to first player
5     if (oldContents == C4Board.NULLPLAYER.NUMBER &

```

```

newContents == C4Board.FIRST_PLAYER_NUMBER)
{
    //if we had no second player tokens, we now have a better score
    //for the first player, tell the stats.
    if(numPlayer2 == 0)
    {
        switch(numPlayer1)
        {
            case 0: stats.pl_1InARowInc();
            break;

            case 1: stats.pl_1InARowDec();
                    stats.pl_2InARowInc();
            break;

            case 2: stats.pl_2InARowDec();
                    stats.pl_3InARowInc();
            break;

            case 3: stats.pl_3InARowDec();
                    stats.pl_4InARowInc();
            break;

            default: errorPrint();
        } //end switch player 1
    } //end if numPlayer2 = 0

    //if we have some second player tokens, and
    //we had no first player tokens, we are no longer populated
    //exclusively by second player tokens, change the stats
    else if(numPlayer1 == 0)
    {
        switch(numPlayer2)
        {
            case 0:
            break;

            case 1: stats.p2_1InARowDec();
            break;

            case 2: stats.p2_2InARowDec();
            break;

            case 3: stats.p2_3InARowDec();
            break;

            default: errorPrint();
        } //end switch player 1
    }

} //end if old = null, new = player 1

//case 2, moving from null to second player
else if(oldContents == C4Board.NULLPLAYER_NUMBER &
newContents == C4Board.SECONDPLAYER_NUMBER)
{
    //if we had no first player tokens, we now have a better score
    //for the second player, tell the stats.
    if(numPlayer1 == 0)
    {
        switch(numPlayer2)
        {
            case 0: stats.p2_1InARowInc();
            break;

            case 1: stats.p2_1InARowDec();
                    stats.p2_2InARowInc();
            break;

            case 2: stats.p2_2InARowDec();
                    stats.p2_3InARowInc();
            break;

            case 3: stats.p2_3InARowDec();
                    stats.p2_4InARowInc();

```

```

82         break;
83     default: errorPrint();
84 } //end switch player 2
85 } //end if numPlayer1 = 0
86
87 //if we have some first player tokens,
88 //and we had no second player tokens, then we are no
89 //longer populated exclusively by first player tokens
90 else if(numPlayer2 == 0)
91 {
92     switch(numPlayer1)
93     {
94         case 0:
95             break;
96
97         case 1: stats.pl_1InARowDec();
98             break;
99
100        case 2: stats.pl_2InARowDec();
101            break;
102
103        case 3: stats.pl_3InARowDec();
104            break;
105
106        default: errorPrint();
107    } //end switch player 1
108 }
109 } //end if old = null, new = player 2
110
111 //case 3, moving from player 1 to null
112 else if(oldContents == C4Board.FIRST_PLAYER_NUMBER &
113        newContents == C4Board.NULL_PLAYER_NUMBER)
114 {
115     //if no player 2 tokens, we have a worse score
116     if(numPlayer2 == 0)
117     {
118         switch(numPlayer1)
119         {
120
121             case 1: stats.pl_1InARowDec();
122                 break;
123
124             case 2: stats.pl_2InARowDec();
125                     stats.pl_1InARowInc();
126                 break;
127
128             case 3: stats.pl_3InARowDec();
129                     stats.pl_2InARowInc();
130                 break;
131
132             case 4: stats.pl_4InARowDec();
133                     stats.pl_3InARowInc();
134                 break;
135
136             default: errorPrint();
137         } //end switch player 1
138
139     } //end if numPlayer2 == 0
140     //if there was only 1 player 1 token, then we now have a score for p2
141     if(numPlayer1 == 1)
142     {
143         switch(numPlayer2)
144         {
145             case 0:
146                 break;
147
148             case 1: stats.p2_1InARowInc();
149                 break;
150
151             case 2: stats.p2_2InARowInc();
152                 break;
153
154             case 3: stats.p2_3InARowInc();
155                 break;
156

```

```

158             default: errorPrint();
159                 } //end switch player 2
160         } //end if numPlayer1 == 1
161     } //end old = p1, new = null
162
163     //case 4, moving from player 2 to null
164     else if (oldContents == C4Board.SECOND_PLAYER_NUMBER &
165             newContents == C4Board.NULL_PLAYER_NUMBER)
166     {
167         //if no player 1 tokens, we have a worse score
168         if (numPlayer1 == 0)
169         {
170             switch (numPlayer2)
171             {
172
173                 case 1: stats.p2_1InARowDec();
174                     break;
175
176                 case 2: stats.p2_2InARowDec();
177                         stats.p2_1InARowInc();
178                     break;
179
180                 case 3: stats.p2_3InARowDec();
181                         stats.p2_2InARowInc();
182                     break;
183
184                 case 4: stats.p2_4InARowDec();
185                         stats.p2_3InARowInc();
186                     break;
187
188                 default: errorPrint();
189             } //end switch player 2
190
191             //end if numPlayer1 == 0
192             //if there was only 1 player 2 token, then we now have a score for p1
193             if (numPlayer2 == 1)
194             {
195                 switch (numPlayer1)
196                 {
197                     case 0:
198                         break;
199
200                     case 1: stats.p1_1InARowInc();
201                         break;
202
203                     case 2: stats.p1_2InARowInc();
204                         break;
205
206                     case 3: stats.p1_3InARowInc();
207                         break;
208
209                     default: errorPrint();
210                 } //end switch player 1
211             } //end if numPlayer2 == 1
212         } //end old = p2, new = null
213
214
215     if (newContents == C4Board.FIRST_PLAYER_NUMBER)
216     {
217         numPlayer1++;
218     }
219     else if (newContents == C4Board.SECOND_PLAYER_NUMBER)
220     {
221         numPlayer2++;
222     }
223     else if (newContents == C4Board.NULL_PLAYER_NUMBER)
224     {
225         if (oldContents == C4Board.FIRST_PLAYER_NUMBER)
226         {
227             numPlayer1--;
228         }
229         else if (oldContents == C4Board.SECOND_PLAYER_NUMBER)
230         {
231             numPlayer2--;
232         }
233     }

```

```

234      }
    }
  } //end contentsChanged

```

3.2 Modifiche

Nel 1988 Victor Allis ha risolto e specificato in modo formale il gioco Forza 4 nella sua tesi [1]. In tale documento sono descritte in dettaglio molte strategie che possono essere sia utilizzate dal giocatore umano sia implementate tramite algoritmi per essere utilizzate da intelligenze artificiali. Per questioni di tempo è stato possibile utilizzare nel progetto solo una piccola parte del contenuto del documento: si tratta della regola strategica delle minacce utili. Per capire di cosa si tratta dobbiamo innanzitutto introdurre alcuni concetti:

- per *minaccia* si intende un gruppo di 4 posizioni adiacenti nelle quali sono presenti 3 pedine di uno stesso giocatore, e pertanto rimane una posizione vuota che se completata con un'ulteriore pedina consente la vittoria al suddetto giocatore;
- abbiamo una *minaccia dispari* quando la posizione vuota è in un numero di riga dispari;
- abbiamo una *minaccia pari* quando la posizione vuota è in un numero di riga pari.

Secondo quanto descritto da Allis al giocatore che muove per primo (nel nostro caso il nero) è più utile creare minacce dispari, mentre a quello che muove per secondo sono più utili quelle pari. Sulla base di questo principio abbiamo modificato le parti dei sorgenti descritte nella sezione precedente. In particolare nel codice 3.3 abbiamo modificato la formula utilizzata per il calcolo della bontà della posizione di un giocatore, che è diventata per il primo giocatore:

$$strength = 32 \cdot n_{3d} + 16 \cdot n_3 + 4 \cdot n_2 + n_1$$

mentre per il secondo:

$$strength = 32 \cdot n_{3p} + 16 \cdot n_3 + 4 \cdot n_2 + n_1$$

dove n_{3d} è il numero di minacce dispari, mentre n_{3p} quello di minacce pari. Questo ha comportato l'aggiunta dei corrispondenti contatori, *p1_3InARow_hpd* e *p2_3InARow_hpd* e delle corrispondenti funzioni di incremento e decremento nel file *C4Stats*, che utilizzano il nuovo parametro *SHIFT_3_IN_A_ROW_LOW* nello shift del conteggio delle minacce totali. Analogamente è stato necessario modificare la funzione *contentsChanged*, la cui nuova implementazione è visibile nel codice 3.4, per fare in modo che richiamasse quando necessario le rispettive funzioni di incremento e decremento di tali contatori. A tal proposito è stato necessario creare una funzione che permetta di stabilire se una minaccia è pari o dispari (codice 3.5), nonché modificare la classe *C4Slot* per distinguere tra posizioni pari e dispari della board.

Codice 3.3: modifica di *getStrength* per integrare la distinzione fra minacce pari e dispari (file *C4Row.java*)

```

1 private int firstPlayerStrength_hpd()
2 {
3     if (p1_4InARow != 0)
4     {
5         return Integer.MAX_VALUE;
6     }
7     else if (p2_4InARow != 0)
8     {
9         return 0;
10    }
11    else

```

```

13     {
14         /// abbassiamo il peso delle minacce generiche e inseriamo le minacce dispari
15         /// con un peso più alto
16         return (p1_2InARow << SHIFT_2.IN_A_ROW) +
17             (p1_3InARow_hpd << SHIFT_3.IN_A_ROW) +
18             (p1_3InARow << SHIFT_3.IN_A_ROW_LOW) +
19             p1_1InARow;
20     }
21 }
22
23 private int secondPlayerStrength_hpd()
24 {
25     if (p2_4InARow != 0)
26     {
27         return Integer.MAX_VALUE;
28     }
29     else if (p1_4InARow != 0)
30     {
31         return 0;
32     }
33     else
34     {
35         /// abbassiamo il peso delle minacce generiche e inseriamo le minacce pari
36         /// con un peso più alto
37         return (p2_2InARow << SHIFT_2.IN_A_ROW) +
38             (p2_3InARow_hpd << SHIFT_3.IN_A_ROW) +
39             (p2_3InARow << SHIFT_3.IN_A_ROW_LOW) +
40             p2_1InARow;
41     }
42 }
43
44 public int getStrength(Player aPlayer, Boolean hpd)
45 {
46     if (!hpd)
47     {
48         if (aPlayer.getNumber() == PLAYER1)
49         {
50             return firstPlayerStrength() - secondPlayerStrength();
51         }
52         else
53         {
54             return secondPlayerStrength() - firstPlayerStrength();
55         }
56     }
57     else
58     {
59         if (aPlayer.getNumber() == PLAYER1)
60         {
61             return firstPlayerStrength_hpd() - secondPlayerStrength_hpd();
62         }
63         else
64         {
65             return secondPlayerStrength_hpd() - firstPlayerStrength_hpd();
66         }
67     }
68 }

```

Codice 3.4: nuova versione di *contentsChanged* per l'euristica avanzata (file *C4Row.java*)

```

1     public void contentsChanged(int oldContents, int newContents)
2     {
3         /// case 1, moving from null to first player
4         if (oldContents == C4Board.NULL_PLAYER_NUMBER &
5             newContents == C4Board.FIRST_PLAYER_NUMBER)
6         {
7             /// if we had no second player tokens, we now have a better score
8             /// for the first player, tell the stats.
9             if (numPlayer2 == 0)
10            {
11                /// ricordando che numPlayer1 contiene il numero di pedine del
12                /// primo
13                /// giocatore nell'attuale slot (ossia gruppo di 4 posizioni
14                /// monitorato)
15                switch (numPlayer1)
16                {
17                    case 0: stats.p1_1InARowInc();
18                }
19            }
20        }
21    }

```

```

17         break;
19         case 1: stats.p1_1InARowDec();
                stats.p1_2InARowInc();
                break;
21
23         case 2: stats.p1_2InARowDec();
                stats.p1_3InARowInc();
                //+ nel caso sia anche dispari incrementiamo
                   il relativo indice
                if (!isEvenThreat())
                   this.stats.p1_3InARowInc_hpd();
27         break;
29
31         case 3: stats.p1_3InARowDec();
                stats.p1_4InARowInc();
                //+ decremento l'indice giusto se era dispari
                if (this.evenThreat == -1)
                   this.stats.p1_3InARowDec_hpd();
33         break;
35
37         default: errorPrint();
                }//end switch player 1
            }//end if numPlayer2 = 0
39
41         //if we have some second player tokens, and
43         //we had no first player tokens, we are no longer populated
45         //exclusively by second player tokens, change the stats
47         else if(numPlayer1 == 0)
49         {
51             switch(numPlayer2)
53             {
55                 // già coperto dal caso precedente
57                 case 0:
59                     break;
61
63                 case 1: stats.p2_1InARowDec();
65                     break;
67
69                 case 2: stats.p2_2InARowDec();
71                     break;
73
75                 case 3: stats.p2_3InARowDec();
77                     if (this.evenThreat == 1)
79                         stats.p2_3InARowDec_hpd();
81                     break;
83
85                 default: errorPrint();
87             }//end switch player 1
89         }
    }//end if old = null, new = player 1

    //case 2, moving from null to second player
    else if(oldContents == C4Board.NULLPLAYER.NUMBER &
           newContents == C4Board.SECONDPLAYER.NUMBER)
    {
        //if we had no first player tokens, we now have a better score
        //for the second player, tell the stats.
        if(numPlayer1 == 0)
        {
            switch(numPlayer2)
            {
                case 0: stats.p2_1InARowInc();
                        break;

                case 1: stats.p2_1InARowDec();
                        stats.p2_2InARowInc();
                        break;

                case 2: stats.p2_2InARowDec();
                        stats.p2_3InARowInc();
                        if (this.evenThreat == 1)
                           this.stats.p2_3InARowInc_hpd();
                        break;
            }
        }
    }

```

```

91         case 3: stats.p2_3InARowDec();
92             if (this.eventThreat == 1)
93                 this.stats.p2_3InARowDec_hpd();
94                 stats.p2_4InARowInc();
95         break;
96
97         default: errorPrint();
98     } //end switch player 2
99 } //end if numPlayer1 = 0
100
101 //if we have some first player tokens,
102 //and we had no second player tokens, then we are no
103 //longer populated exclusively by first player tokens
104 else if(numPlayer2 == 0)
105 {
106     switch(numPlayer1)
107     {
108         case 0:
109             break;
110
111         case 1: stats.p1_1InARowDec();
112             break;
113
114         case 2: stats.p1_2InARowDec();
115             break;
116
117         case 3: stats.p1_3InARowDec();
118             if (this.eventThreat == -1)
119                 this.stats.p1_3InARowDec_hpd();
120             break;
121
122         default: errorPrint();
123     } //end switch player 1
124 }
125
126 } //end if old = null, new = player 2
127
128 //case 3, moving from player 1 to null
129 else if(oldContents == C4Board.FIRST_PLAYER_NUMBER &
130         newContents == C4Board.NULL_PLAYER_NUMBER)
131 {
132     //if no player 2 tokens, we have a worse score
133     if(numPlayer2 == 0)
134     {
135         switch(numPlayer1)
136         {
137
138             case 1: stats.p1_1InARowDec();
139                 break;
140
141             case 2: stats.p1_2InARowDec();
142                 stats.p1_1InARowInc();
143                 break;
144
145             case 3: stats.p1_3InARowDec();
146                 if (this.eventThreat == -1)
147                     this.stats.p1_3InARowDec_hpd();
148                 stats.p1_2InARowInc();
149                 break;
150
151             case 4: stats.p1_4InARowDec();
152                 stats.p1_3InARowInc();
153                 if (this.eventThreat == -1)
154                     this.stats.p1_3InARowInc_hpd();
155                 break;
156
157             default: errorPrint();
158         } //end switch player 1
159
160 } //end if numPlayer2 == 0
161 //if there was only 1 player 1 token, then we now have a score for p2
162 if(numPlayer1 == 1)
163 {
164     switch(numPlayer2)
165     {

```



```

167         case 0:
168             break;
169
170         case 1: stats.p2_1InARowInc();
171             break;
172
173         case 2: stats.p2_2InARowInc();
174             break;
175
176         case 3: stats.p2_3InARowInc();
177             if (this.evenThreat == 1)
178                 this.stats.p2_3InARowInc_hpd();
179             break;
180
181         default: errorPrint();
182             } //end switch player 2
183     } //end if numPlayer1 == 1
184 } //end old = p1, new = null
185
186 //case 4, moving from player 2 to null
187 else if (oldContents == C4Board.SECOND_PLAYER_NUMBER &
188     newContents == C4Board.NULL_PLAYER_NUMBER)
189 {
190     //if no player 1 tokens, we have a worse score
191     if (numPlayer1 == 0)
192     {
193         switch (numPlayer2)
194         {
195
196             case 1: stats.p2_1InARowDec();
197                 break;
198
199             case 2: stats.p2_2InARowDec();
200                 stats.p2_1InARowInc();
201                 break;
202
203             case 3: stats.p2_3InARowDec();
204                 if (this.evenThreat == 1)
205                     this.stats.p2_3InARowDec_hpd();
206                 stats.p2_2InARowInc();
207                 break;
208
209             case 4: stats.p2_4InARowDec();
210                 stats.p2_3InARowInc();
211                 if (this.evenThreat == 1)
212                     this.stats.p2_3InARowInc_hpd();
213                 break;
214
215             default: errorPrint();
216                 } //end switch player 2
217
218     } //end if numPlayer1 == 0
219     //if there was only 1 player 2 token, then we now have a score for p1
220     if (numPlayer2 == 1)
221     {
222         switch (numPlayer1)
223         {
224
225             case 0:
226                 break;
227
228             case 1: stats.p1_1InARowInc();
229                 break;
230
231             case 2: stats.p1_2InARowInc();
232                 break;
233
234             case 3: stats.p1_3InARowInc();
235                 if (this.evenThreat == -1)
236                     this.stats.p1_3InARowInc_hpd();
237                 break;
238
239             default: errorPrint();
240                 } //end switch player 1
241     } //end if numPlayer2 == 1
242 } //end old = p2, new = null

```

```

243
245         if(newContents == C4Board.FIRST_PLAYER_NUMBER)
246         {
247             numPlayer1++;
248         }
249         else if(newContents == C4Board.SECOND_PLAYER_NUMBER)
250         {
251             numPlayer2++;
252         }
253         else if(newContents == C4Board.NULL_PLAYER_NUMBER)
254         {
255             if(oldContents == C4Board.FIRST_PLAYER_NUMBER)
256             {
257                 numPlayer1--;
258             }
259             else if(oldContents == C4Board.SECOND_PLAYER_NUMBER)
260             {
261                 numPlayer2--;
262             }
263         }
    }
} //end contentsChanged

```

Codice 3.5: funzione per distinguere le minacce pari da quelle dispari (file *C4Row.java*)

```

private boolean isEvenThreat() {
2   boolean result = false;
    // la minaccia è rappresentata dalla casella vuota che manca a completare 4 pedine
    // adiacenti
4   // gli if controllano se ogni possibile minaccia (se è presente) se è pari o no
    if (this.s1.getContents() == C4Board.NULL_PLAYER_NUMBER)
6       result = this.s1.isEven();
    if (this.s2.getContents() == C4Board.NULL_PLAYER_NUMBER)
8       result = this.s2.isEven();
    if (this.s3.getContents() == C4Board.NULL_PLAYER_NUMBER)
10      result = this.s3.isEven();
    if (this.s4.getContents() == C4Board.NULL_PLAYER_NUMBER)
12      result = this.s4.isEven();
    // imposta il relativo campo
14    if (result)
        this.evenThreat = 1;
16    else
        this.evenThreat = -1;
18    return result;
}

```

3.3 Risultati

I risultati sono purtroppo solo leggermente migliorativi rispetto all'euristica originale, consentendo di vincere due partite in più ai valori di lookahead di 4 e 6 ply, ma peggiorando il risultato al valore di 8 ply, situazione nella quale il giocatore con l'euristica modificata perde la partita, diversamente da quanto succedeva nel caso di utilizzo di quella originale.

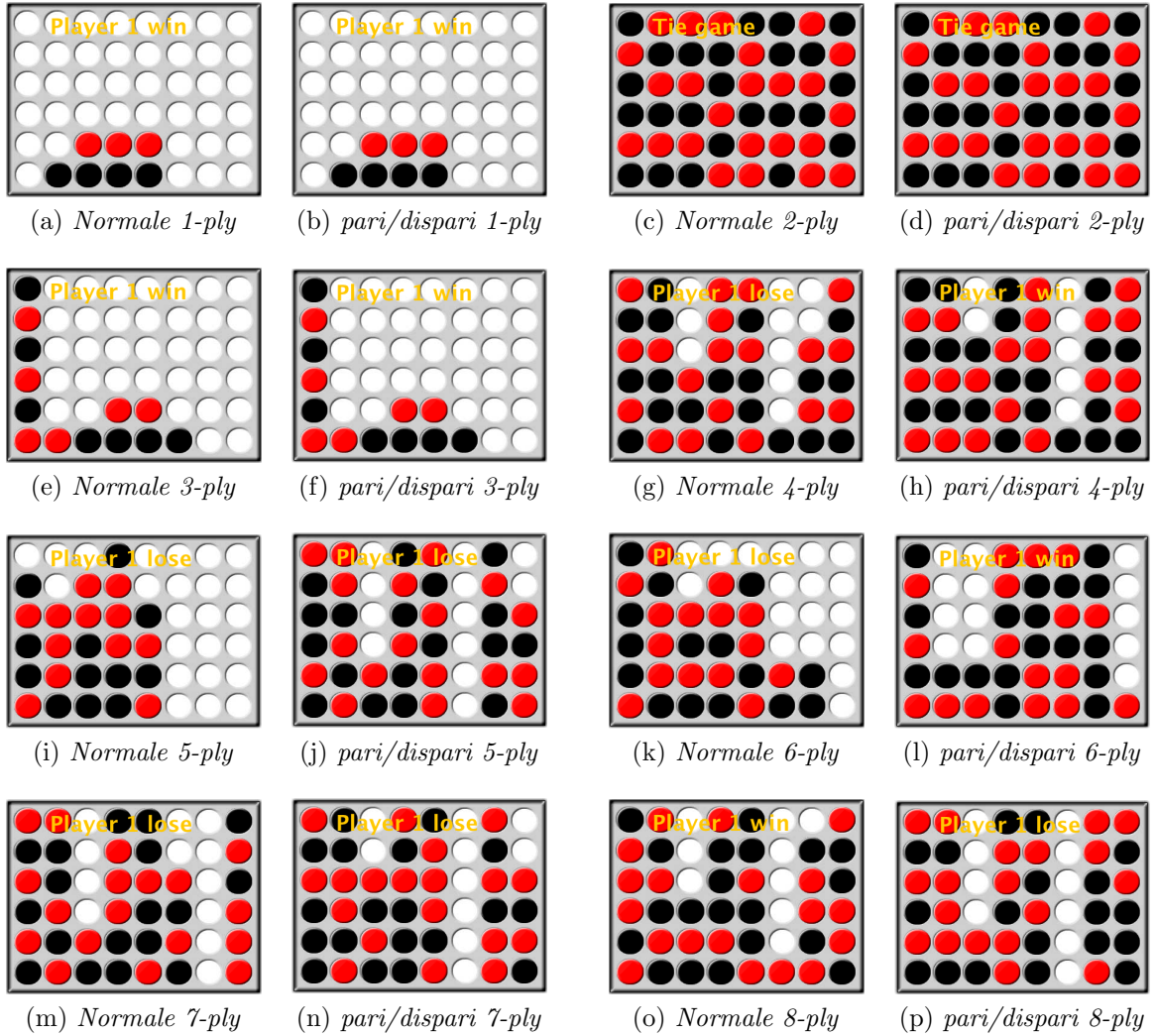


Figura 3.1: confronto fra le giocate nelle partite con entrambi i giocatori Minimax standard, e con primo giocatore (nero) con l'euristica modificata

Capitolo 4

Conclusioni e possibili sviluppi

I risultati ottenuti negli esperimenti sul pruning sono in linea con quanto avremmo potuto attenderci. In particolare lo sviluppo della ricerca di quiescenza ha permesso di aumentare notevolmente le vittorie del giocatore che la utilizza, a parità del valore di lookahead. Non altrettanto efficace è risultata la modifica dell'euristica, e proprio in questo punto risiedono le maggiori possibilità di sviluppo, sia per quel che riguarda il tuning dei parametri utilizzati nelle formule, sia per quel che riguarda l'implementazione di tutte le altre strategie descritte nel documento di Victor Allis.

Bibliografia

- [1] Victor Allis. A knowledge-based approach of connect-four - the game is solved: White wins, 1988.
- [2] Sean Bridges. C4 applet. <http://www.geocities.com/ResearchTriangle/System/3517/C4/C4Conv.html>.