Aprendizaje por refuerzo aplicado a juegos de peleas

José Emmanuel Fuentes Cortés
Fac. de Matematicas - UADY
fuencort@gmail.com

16 de diciembre de 2015

Índice

1. Introducción

Existen algunas formas de aprendizaje aplicadas a agentes inteligentes. Entre las más relevantes están el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo.

El aprendizaje por refuerzo consiste en aprender a decidir, ante una situación determinada, que acción es la más adecuada para lograr un objetivo, tal técnica es adecuada cuando no existe un conocimiento .ª priori"del entorno o este es demasiado complejo como para utilizar otros métodos.

A lo largo de este proyecto se desarrollará esta técnica aplicada a resolver lo problemas que pudieran presentarse durante un videojuego de peleas. Se pretende que el agente sea lo suficientemente inteligente como para derrotar de manera autónoma a un contendiente cuyos movimientos son elegidos al azar.

Se eligió como interfaz el juego **FIGHTING ICE**, proporcionado por *INTELLIGENT COMPUTER ENTERTAINMENT LAB., RITSUMEIKAN UNIVERSITY* para su competencia internacional *FIghting Game AI Comptetition*.

2. Marco Teorico

Para este proyecto se implementara una versión del algortimo Q-learning, tomado de [1]. También se harán uso de las herramientas aprendidas a lo largo del curso de Inteligencia Artificial impartido durante el semestre actual en la Facultad de Matemáticas de la Universidad Autónoma de Yucatán.

El algoritmo Q-learning se desarrolla a grandes rasgos de la siguiente manera:

- El agente percive un conjunto finito, S, de estados distintos en su entorno, y dispone de un conjunto finito, A, de acciones para interactuar con él.
- El tiempo avanza de forma discreta, y en cada instante de tiempo, t, el agente percibe un estado concreto, s, selecciona una acción posible, a, y la ejecuta, obteniendo un nuevo estado, $s^+ = a(s)$.
- El entorno responde a la acción del agente por medio de una recompensa, o castigo, $r(s, a)$.
- La recompensa y el estado siguiente obtenido, no tiene porqué ser conocido a priori por el agente, y dependen únicamente del estado actual y la acción tomada.

Si el agente supiera a priori los valores de todos los posibles pares (estado, acción) podría usar esta información para seleccionar la acción adecuada para cada estado. El problema es que al principio el agente no tiene esta información, por lo que su primer objetivo es aproximar lo mejor posible esta asignación de valores.

Por lo tanto:

- Si una acción en un estado determinado causa un resultado no deseado, hay que aprender no aplicar esa acción en ese estado.
- Si una acción en un estado determinado causa un resultado sí deseado, hay que aprender a aplicar esa acción en ese estado.
- Si todas las acciones que se pueden tomar desde un estado determinado dan resultado negativo, aprender a evitar ese estado.
- Si cualquier acción en un determinado estado da un resultado positivo, aprender que es conveniente llegar a él.

Para hacer posible esto, nos apegamos a la siguiente ecuación:

$$Q[s,a] = Q[s,a] + \alpha(r + \gamma max_d\, Q[a',s'] - Q[a,s])$$

Donde Q[s,a] es la pareja estado-acción actuales, α una variable de aprendizaje, r la recompensa y $max_d\, Q[a',s']$ el valor de la acción más alta posible en s'.

Formalizando lo anterior, el algoritmo Q-learning es el que sigue:

función AGENTE-APRENDIZAJE-Q (*percepción*) **devuelve** una acción
  entradas: *percepción*, una percepción indica el estado actual s' y la señal de recompensa r'
  estática: Q, una tabla de valores de acción indexada por el estado y la acción
    N_{sa}, una tabla de frecuencias de los pares estado-acción
    s, a, r, el estado, la acción y la recompensa previa, inicialmente nulos

  si s no es nulo **entonces hacer**
    incrementar $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[a,s] + \alpha\,(N_{sa}[s,a])\,(r + \gamma\, \max_{a'} Q[a',s'] - Q[a,s])$
  si TERMINAL?[s'] **entonces** s, a, r ← nulo
  si no $s, a, r \leftarrow s'$, $\text{argmax}_{a'}\, f(Q[a',s'], N_{sa}[a',s'], r')$
  devolver a

Figura 1: Algoritmo Q-learning

3. Diseño del Experimento

Según la información proporcionada por *FIghting Game AI Comptetition* la interfaz de pelea cuenta con varias funciones preestablecidas para proporcionar la información necesaria para ejecutar casi cualquier rutina de inteligencia artificial que se desee.

3.1. DataIO.java

Para que la información no se perdiera al término de cada ejecución del programa, se implementó un sistema de almacenamiento y recuperación de información en dos archivos .txt, tales fueron nombrados "data.txtz "qdata.txt", lo cuales almacenan la información de los estados conocidos y los valores numéricos de sus acciones respectivamente.

La clase DataIO se encargó de este cometido y es mostrada a continuación:

```
1  import java.io.*;
2  import java.util.*;
3  import java.net.*;
4
5  public class DataIO{
6
7          /**
8           * DataIO se encarga de la creación de la tabla a partir de un archivo
                txt y también
9           * del almacenaje de la misma al finalizar su uso. La tabla se
                almacenará en el mismo
10          * archivo del cual los datos fueron leódos, tal archivo es definido al
                momento de
11          * instanciar el objeto DataIO según sea el parómetro que se le pase.
12          *
13          * Recibe como entrada un String con el nombre/ubicación del archivo a
                utilizar.
14          *
15          * Mótodos:
16          *
17          * + public ArrayList<ArrayList<String>> readFile()
18          * + public void writeFile(ArrayList<ArrayList<String>> tabla)
19          */
20
21
22         File base = null;
23         File datafile= null;
24
25         public DataIO(){
26                 try{
27                         this.base = new File(DataIO.class.getProtectionDomain()
                                .getCodeSource().getLocation().toURI()).
                                getParentFile();
28                         this.datafile =  new File(base,"data.txt");
29                 }
30                 catch(URISyntaxException o){
31                         System.err.println(o);
32                 }
33         }
34
35         public DataIO(String nombreArchivo){
36                 try{
```

```
37                      this.base = new File(DataIO.class.getProtectionDomain()
                            .getCodeSource().getLocation().toURI()).
                            getParentFile();
38                      this.datafile =  new File(base, nombreArchivo);
39              }
40              catch(URISyntaxException o){
41                      System.err.println(o);
42              }
43
44      }
45
46      public ArrayList<ArrayList<String>> readFile() {
47
48              /**
49               *      readFile se encarga de tomar los datos almacenados en
                        un archivo llamado
50               *  según el parómetro de entrada de la clase y lo procesa para
                        que se integren
51               *  al programa ordenados en una matriz (ArrayList).
52               */
53
54              ArrayList<ArrayList<String>> tabla = new ArrayList<ArrayList<
                    String>>();
55
56              try{
57
58                      BufferedReader br = new BufferedReader(new FileReader(
                            this.datafile));
59
60                      String line = null;
61                      String[] lineDividida = null;
62                      while ((line = br.readLine()) != null) {
63                              lineDividida = line.split(" ");
64                              ArrayList <String> arrlTemp = new ArrayList <
                                    String>();
65                              for (String i: lineDividida){
66                                      arrlTemp.add(i);
67                              }
68                              tabla.add(arrlTemp);
69                      }
70
71                      br.close();
72                      System.out.println("Archivo leido");
73              }
74              catch(IOException e){
75                      System.err.println(e);
76              }
77
78
79              return tabla;
80      }
81
82      public void writeFile(ArrayList<ArrayList<String>> tabla){
83
84              /**
85               * writeFile se encarga de almacenar todos los datos de la
                        tabla pasada por parametro
86               * en el mismo archivo que se haya utilizado para crear la
                        tabla durante el mótodo
```

```
87                  * readFile. El archivo se define al instanciar el objeto
                        DataIO, ya sea según el
88                  * parametro que le sea pasado.
89                  */
90
91                 try{
92                         FileOutputStream fos = new FileOutputStream(this.
                            datafile);
93                         BufferedWriter bw = new BufferedWriter(new
                            OutputStreamWriter(fos));
94
95                         String line = "";
96                         for (ArrayList<String> i: tabla) {
97                                 for(String j: i){
98                                         line += j + " ";
99                                 }
100                                bw.write(line);
101                                bw.newLine();
102                                line = "";
103                        }
104
105                        bw.close();
106                        fos.close();
107                        System.out.println("Archivo escrito");
108                }
109                catch(IOException e){
110                        System.err.println(e);
111                }
112        }
113 }
```

3.2. StateHandle.java

Para la definición de nuevos estados, es decir, los parámetros que serían tomados en cuenta y su forma de almacenamiento durante el programa se utilizó una clase llamada StateHandle.

Las parámetros tomados en cuenta fueron:

- Distancia en X absoluta entre los dos jugadores, discretizada en cerca, distancia media y lejos.
- Valor booleano indicador de si se esta ganando o perdiendo.
- Estado del oponente, dividido en STAND, AIR y CROUCH.
- Acción tomada por el oponente.

A partir de estos valores llegamos a un total de 441 estados conocidos hasta el momento. Debido a que los estados se generan de manera automática, este número podría incrementarse a lo largo del tiempo.
El código utilizado se muestra a continuación:

```java
import java.util.*;

public class StateHandle {
        /**
         * Esta clase fue creada con el motivo de modificar el estado recibido para adecuarlo
         * a un estado discretizado y por lo tanto, mós fócil de manipular.
         *
         * La discretización se da principalmente en la distancia medida en X entre el jugador
         * 1 y el jugador 2, en donde a partir de un valor int [0, >400] se reduce a un
         * int [1, 3], según se considere cerca, medio, o lejos.
         *
         */
        public ArrayList<String> discretizador(ArrayList<String> stateString){

                ArrayList<String> stateDiscret = new ArrayList<String>();

                //evalua distancia en X
                if(Integer.parseInt(stateString.get(0))>= 0 && Integer.parseInt(stateString.get(0)) < 60){
                        stateDiscret.add("1"); //cerca
                }
                else if(Integer.parseInt(stateString.get(0))>= 60 && Integer.parseInt(stateString.get(0)) < 200){
                        stateDiscret.add("2"); //medio
                }
                else{stateDiscret.add("3");} //lejos

                //evalua mi hp
                if (Integer.parseInt(stateString.get(1)) >= Integer.parseInt(stateString.get(2))){
                        stateDiscret.add("1");
                }
                else{stateDiscret.add("0");}

                //evalua hp enemigo
                if (Integer.parseInt(stateString.get(2)) >= Integer.parseInt(stateString.get(1))){
                        stateDiscret.add("1");
                }
                else{stateDiscret.add("0");}

                stateDiscret.add(stateString.get(3));
                stateDiscret.add(stateString.get(4));


                return stateDiscret;
        }
}
```

3.3. Qlearning.java

La clase en donde está implementado todo comportamiento relacionado directamente con el algoritmo Q-learning fue denominada "Qlearning".

Aquí se generaron diferentes métodos de clase para solucionar diferentes problemas, entre los cuales están:

- Obtener el valor numérico más alto entre las acciones posibles según el estado actual. Tal método es llamado **getMAxDouble()**.
- Obtener un string útil para que la interfaz pudiera interpretar la acción que debía tomar el personaje. Tal método es llamado **getMaxString()**.
- Generar la recompensa necesaria para ejecutar la ecuación del algoritmo Q-learning. Tal método es llamado **getReward()**.
- La definición de las acciones posibles. El método utilizado es **getAction()**.
- y finalmente, La implementación de la ecuación a partir de todos los datos recabados anteriormente. Tal método fue llamado **learn()**.

Es importante destacar que el personaje no era libre de elegir entre cualquiera de las acciones posibles en la interfaz de Fighting ice, sino que las acciones posibles fueron disminuidas por motivos de simplicidad del algoritmo y de una más fácil depuración del mismo.

A continuación se muestra el método **getAction()** para ilustrar de mejor manera lo anteriormente mencionado:

```
1  public String getAction(int index){
2
3                  String max = new String();
4
5                  if(index == 0){ max = "STAND_FB"; } // patada
6                  else if (index == 1){ max = "STAND_D_DF_FA"; } // poder a
                       distancia
7                  else if (index == 2){ max = "9 6_B";} // salto al frente y
                       patada
8                  else if (index == 3){ max = "6";} //hacia adelante
9                  else if (index == 4){ max = "CROUCH_FB";} // patada abajo
10                 else if (index == 5){ max = "STAND_F_D_DFA";} // poder de golpe
                        hacia arriba
11                 else if (index == 6){ max = "7 4_B"; } // salto hacia atrós y
                       patada
12                 else{ System.out.println("Error, acción no encontrada");}
13
14                 return max;
15         }
```

Se observa que las acciones utilizadas por el personaje son únicamente 7, contrario a las más de 20 acciones posibles según la interfaz oficial de Fighting ice.

Otra cosa para destacar sobre el algoritmo es que se modificó ligeramente la ecuación para que, en lugar de modificar el estado actual a partir de un estado siguiente, se modificara el estado anterior a partir del estado actual. Esto por los obvios motivos de que al ser una interacción en tiempo real, es imposible predecir el estado posterior pues no se sabe aún de qué manera cambiará el ambiente.

Se deja el el código completo para mejor documentación del lector:

```
1  import java.util.*;
2
3
4  public class Qlearning {
5
6          public static final double ALFA = 0.15;
7          public static final double GAMMA = 0.9;
8
9          public Double getMaxDouble(ArrayList<Double> acciones){
10         /**
11          * Esta función recibe un vector de acciones posibles en
12          * sus valores numóricos y elige el valor mós alto.
13          */
14                 Double maxDouble;
15
16                 maxDouble = acciones.get(0);
17                 for(Double j: acciones){
18                         maxDouble = (j>maxDouble) ? j : maxDouble;
19                 }
20
21                 return maxDouble;
22
23         }
24
25         public String getMaxString(ArrayList<Double> acciones){
26
27                 /**
28                  * Recibe un vector de acciones posibles en su valor numórico,
29                  * selecciona el valor numórico mós alto y apartir de ese
30                  * valor se elije un string que determina la acción a tomar,
31                  * la cual seró la que se enviaró a la interfaz de Fighting Ice
32                  * para que ejecute la acción.
33                  */
34
35                 String max = new String();
36                 Double maxDouble;
37                 Random explorar = new Random();
38                 Random accionAleatoria = new Random();
39                 int index;
40
41                 maxDouble = acciones.get(0);
42                 for(Double j: acciones){
43                         maxDouble = (j>maxDouble) ? j : maxDouble;
44                 }
45
```

```
46                  /*
47                   * ya elegido el valor numórico mós alto, se hace un tiro de
48                   * moneda en donde se busca que el 70% se las veces se ejecute
49                   * esta acción. El otro 30% de las vecs se elegiró una acción
50                   * aleatoria.
51                   *
52                   * La función get acción es la encargad de regresar el string
53                   * a partir del óndice del valor seleccionado.
54                   */
55                  if(explorar.nextDouble() > 0.7){
56                          index = accionAleatoria.nextInt(acciones.size());
57                          max = getAction(index);
58                          System.out.println("*****EXPLORACION*****");
59                  }
60                  else {
61                          index = acciones.indexOf(maxDouble);
62                          max = getAction(index);
63                  }
64
65                  System.out.println("maxAccion = " + max);
66                  return max;
67
68          }
69
70          public int getReward(ArrayList<String> estado, int newHpAbs, int
                lastHpAbs){
71                  int reward = 0;
72
73                  /**
74                   * Se elige la recompensa, recibe:
75                   *
76                   * estado : El estado actual del agente.
77                   * newHpAbs: el Hp absoluto actual (myHp - enemyHp).
78                   * lastHPAbs: Hp absoluto del estado anterior.
79                   *
80                   * el parómetro estado no se utiliza en esta versión final de
81                   * recompensas, sin embargo se utilizó para pruebas iniciales y
82                   * decide dejarse en caso de que sea de utilidad en un futuro.
83                   */
84
85                  if(newHpAbs > lastHpAbs){
86                          if (newHpAbs < 0){ reward = 100; }
87                          else if (newHpAbs >= 0 && newHpAbs < 50){ reward = 5; }
88                          else{ reward = 1000; }
89                  }
90                  else{
91                          if (newHpAbs < 0){ reward = -200; }
92                          else if (newHpAbs >= 0 && newHpAbs < 50){ reward = -50;
                                }
93                          else{ reward = -10; }
94                  }
95
96
97
98                  System.out.println("lastHp = " + lastHpAbs + " newHp = "+
                        newHpAbs + " reward = "+ reward);
99                  return reward;
100         }
101
```

```
102         public void learn(ArrayList<Double> lastAction, int indexLastMaxAction
                , Double maxNewAction, int reward){
103 
104                 /**
105                  * Se actualiza el Q(s, a) anterior a partir del Q(s, a) actual
                        .
106                  */
107 
108                 Double lastQ_A = lastAction.get(indexLastMaxAction);
109                 lastQ_A = lastQ_A + ALFA*(reward + GAMMA*maxNewAction - lastQ_A
                        );
110                 lastAction.set(indexLastMaxAction, lastQ_A);
111 
112         }
113 
114         public String getAction(int index){
115 
116                 /**
117                  * Esta función se llama en public String getMaxString(
                        ArrayList<Double> acciones),
118                  * recibe el óndice de la acción seleccionada y regresa un
                        string necesario para
119                  * que la acción sea reconocida por la interfaz de Fighting Ice
                        .
120                  */
121 
122                 String max = new String();
123 
124                 if(index == 0){ max = "STAND_FB"; } // patada
125                 else if (index == 1){ max = "STAND_D_DF_FA"; } // poder a
                        distancia
126                 else if (index == 2){ max = "9 6_B";} // salto al frente y
                        patada
127                 else if (index == 3){ max = "6";} //hacia adelante
128                 else if (index == 4){ max = "CROUCH_FB";} // patada abajo
129                 else if (index == 5){ max = "STAND_F_D_DFA";} // poder de golpe
                        hacia arriba
130                 else if (index == 6){ max = "7 4_B"; } // salto hacia atrós y
                        patada
131                 else{ System.out.println("Error, acción no encontrada");}
132 
133                 return max;
134         }
135 
136 }
```

3.4. ClubIA.java

La implementación del algoritmo en la interfaz se llevo a cabo en la clase ClubIa, tal clase sigue un formato proporcionado por competencia. El método de clase nombrado **processing()** el llamado tras cada *frame* y por lo tanto debe considerarse como el *loop* principal del videojuego.

El código que esta dentro del método **processing()** se encarga de llamar a todas las clases anteriormente mencionadas para que en su conjunto cumplan con el objetivo principal, que es que tras cada frame se cree un estado nuevo de no existir, elegir una acción a partir de dicho estado, evaluar el resultado en el *frame* siguiente y modificar el valor de $Q(s, a)$.

```
1  import structs.FrameData;
2  import structs.GameData;
3  import structs.Key;
4  import gameInterface.AIInterface;
5  import commandcenter.CommandCenter;
6  import enumerate.Action;
7  import structs.MotionData;
8  import java.util.*;
9
10 public class Club_IA implements AIInterface {
11
12         Key inputKey;
13         boolean playerNumber;
14         FrameData frameData;
15         CommandCenter cc;
16
17         ArrayList<ArrayList<String>> tablaEstados;
18         ArrayList<ArrayList<String>> tablaAcciones;
19         ArrayList<ArrayList<Double>> tablaAccionesDouble;
20
21         ArrayList<String> lastState;
22         ArrayList<String> newState;
23         ArrayList<Double> lastAction;
24         ArrayList<Double> newAction;
25         Double maxDouble;
26         String maxString;
27         int newHpAbs;
28         int lastHpAbs;
29
30
31
32         DataIO ioEstados = new DataIO();
33         DataIO ioAcciones = new DataIO("qdata.txt");
34         StateHandle sth = new StateHandle();
35         Qlearning ql = new Qlearning();
36
37         Random ran = new Random();
38
39         @Override
40         public int initialize(GameData gameData, boolean playerNumber){
41                 this.playerNumber = playerNumber;
```

```
42                  this.inputKey = new Key();
43                  cc = new CommandCenter();
44                  frameData = new FrameData();
45
46                  tablaEstados = ioEstados.readFile();
47                  for(ArrayList<String> i: tablaEstados){
48                          for(String j: i){ System.out.print(j + " "); }
49                          System.out.println(" ");
50                  }
51
52
53                  tablaAcciones = ioAcciones.readFile();
54                  tablaAccionesDouble = new ArrayList<ArrayList<Double>>();
55
56                  for(ArrayList<String> i: tablaAcciones){
57                          ArrayList<Double> nAction = new ArrayList<Double>();
58                          for(String j: i){
59                                  Double d  = Double.parseDouble(j);
60                                  System.out.print(d + " ");
61                                  nAction.add(d);
62                          }
63                          System.out.println(" ...nActions");
64                          tablaAccionesDouble.add(nAction);
65                  }
66                  newState = new ArrayList<String>();
67                  newAction = new ArrayList<Double>();
68                  lastState = tablaEstados.get(0);
69                  lastAction = tablaAccionesDouble.get(0);
70                  newHpAbs = 0;
71                  lastHpAbs = 0;
72
73                  return 0;
74          }
75
76          @Override
77          public void getInformation(FrameData frameData){
78                  this.frameData = frameData;
79                  cc.setFrameData(this.frameData, playerNumber);
80          }
81
82          @Override
83          public void processing() {
84
85                  if(!frameData.getEmptyFlag() && frameData.getRemainingTime() >
                        0){
86                          if(cc.getskillFlag()){ inputKey = cc.getSkillKey(); }
87                          else{
88                                  inputKey.empty();
89                                  cc.skillCancel();
90
91                                  newState = new ArrayList<String>();
92                                  newAction = new ArrayList<Double>();
93
94                                  // agrega información del estado en string
95                                  newState.add(String.valueOf(cc.getDistanceX()))
                                        ; // distancia en X
96                                  newState.add(String.valueOf(cc.getMyHP())); //
                                        mi HP
```

```
97                  newState.add(String.valueOf(cc.getEnemyHP())); // enemigo HP
98                  newState.add(cc.getEnemyCharacter().state.toString()); // estado del enemigo
99                  newState.add(cc.getEnemyCharacter().action.toString()); // accion del enemigo
100                 newState = sth.discretizador(newState);
101
102
103
104                 if(!newState.equals(lastState)){
105
106                         if(tablaEstados.contains(newState)){ // se ejecuta si el estado actual ya existe en la tabla
107                             System.out.println("newState en tablaEstados.");
108
109
110                             int indexNewState = tablaEstados.indexOf(newState);
111                             int indexLastState = tablaEstados.indexOf(lastState);
112
113                             int indexNewAction = indexNewState;
114                             int indexLastAction = tablaAccionesDouble.indexOf(lastAction);
115
116
117                             System.out.println("indexLastState = " + indexLastState);
118                             System.out.println("indexNewState = " + indexNewState);
119                             System.out.println("indexLastAction = " + indexLastAction);
120                             System.out.println("indexNewAction = " + indexNewAction);
121
122                             // copia el estado y acción desde la tabla
123                             newState = tablaEstados.get(indexNewState);
124                             newAction = tablaAccionesDouble.get(indexNewAction);
125
126                             // control** , imprime valores numóricos de las acciones
127                             for(Double i: newAction){
128                                     System.out.print(i + " ");
129                             }
```

```
                System.out.println(" ");

                maxDouble = ql.getMaxDouble(newAction); //regresa el valor numórico mós alto
                maxString = ql.getMaxString(newAction); //regresa string para ejecutar acción
                System.out.println("maxDouble = " + maxDouble);
                int indexNewMaxAction = newAction.indexOf(maxDouble);

                cc.commandCall(maxString); //ejecuta acción
                int myHp = cc.getMyHP();
                int enemyHp = cc.getEnemyHP();
                newHpAbs = myHp - enemyHp;

                //actualiza Q(s, a) anterior al actual
                ql.learn(lastAction, indexNewMaxAction, maxDouble, ql.getReward(newState, newHpAbs, lastHpAbs));

                lastState = newState;
                lastAction = newAction;
                lastHpAbs = newHpAbs;

            }
            else{ //se ejecuta si no existe el estado actual en la tabla
                System.out.println("newState NO en tablaEstados.");


                //%%%%%%%%%%%%%%%crea Q(s, a) y lo agrega a tabla
                tablaEstados.add(newState);
                ArrayList<Double> newActions = new ArrayList<Double>();
                for(Double i: lastAction){
                    newActions.add(ran.nextDouble());
                }
                tablaAccionesDouble.add(newActions);

                //%%%%%%%%%%%%%%%%%%%%%


                // a partir de aquó es el mismo algoritmo que sigue cuando existe un estato en la tabla
                int indexNewState = tablaEstados.indexOf(
```

```
                newState);
167             int indexLastState =
                    tablaEstados.indexOf(
                    lastState);
168
169             int indexNewAction =
                    tablaAccionesDouble.indexOf(
                    newActions);
170             int indexLastAction =
                    tablaAccionesDouble.indexOf(
                    lastAction);
171
172             System.out.println("
                    indexLastState = " +
                    indexLastState);
173             System.out.println("
                    indexNewState = " +
                    indexNewState);
174             System.out.println("
                    indexLastAction = " +
                    indexLastAction);
175             System.out.println("
                    indexNewAction = " +
                    indexNewAction);
176
177             newState = tablaEstados.get(
                    indexNewState);
178             newAction = tablaAccionesDouble
                    .get(indexNewAction);
179
180             for(Double i: newAction){
181                     System.out.print(i + "
                            ");
182             }
183             System.out.println(" ");
184
185             maxDouble = ql.getMaxDouble(
                    newAction);
186             maxString = ql.getMaxString(
                    newAction);
187             System.out.println("maxDouble =
                     " + maxDouble);
188             int indexNewMaxAction =
                    newAction.indexOf(maxDouble)
                    ;
189
190             cc.commandCall(maxString);
191             int myHp = cc.getMyHP();
192             int enemyHp = cc.getEnemyHP();
193             newHpAbs = myHp - enemyHp;
194             ql.learn(lastAction,
                    indexNewMaxAction,
                    maxDouble, ql.getReward(
                    newState, newHpAbs,
                    lastHpAbs));
195
196             lastState = newState;
197             lastAction = newAction;
198             lastHpAbs = newHpAbs;
```

```
199                                         }
200
201                                 }
202
203
204                         }
205                 }
206         }
207
208         @Override
209         public Key input(){
210                 return inputKey;
211         }
212
213         @Override
214         public void close(){
215                 ioEstados.writeFile(tablaEstados);
216
217                 tablaAcciones.clear();
218                 for(ArrayList<Double> i: tablaAccionesDouble){
219                         ArrayList<String> axxion = new ArrayList<String>();
220                         for(Double j: i){
221                                 axxion.add(String.valueOf(j));
222                         }
223                         tablaAcciones.add(axxion);
224                 }
225                 ioAcciones.writeFile(tablaAcciones);
226
227                 System.out.println("cerrando");
228         }
229
230         @Override
231         public String getCharacter(){
232                 return CHARACTER_ZEN;
233         }
234
235         }
```

3.5. Información extra

La información más importante para recavar información relevante es mostrada en un PDF que será incrustado a continuación:

Key

Public access variable name	Value type	Description
U	boolean	If the value is set to true, then the ”Upward” button will be pressed.
L	boolean	If the value is set to true, then the ”Left” button will be pressed.
R	boolean	If the value is set to true, then the ”Right” button will be pressed.
D	boolean	If the value is set to true, then the ”Down” button will be pressed.
A	boolean	If the value is set to true, then the ”A” button will be pressed.
B	boolean	If the value is set to true, then the ”B” button will be pressed.
C	boolean	If the value is set to true, then the ”C” button will be pressed.
getLever	int	Returns the value indicating the direction key inputted by the player using the numeric keypad.

GameData

Method name	Value type	Description
getStageXMax	int	Returns the horizontal length of the battle stage.
getStageYMax	int	Returns the vertical length of the battle stage.
getPlayerOneMaxEnergy	int	Returns the energy limit of the first player’s character.
getPlayerTwoMaxEnergy	int	Returns the energy limit of the second player’s character.
getPlayerOneMotionData	Vector<MotionData>	Returns MotionData of the first player’s character.
getPlayerTwoMotionData	Vector<MotionData>	Returns MotionData of the second player’s character.
getPlayerOneCharacterName	String	Returns the name of the first player’s character.
getPlayerTwoCharacterName	String	Returns the name of the second player’s character.
getMyMaxEnergy(boolean player)	int	Returns the maximum energy of your player character with the parameter player, returned from the method initialize in AIInterface.

getOpponentMaxEnergy(boolean player)	int	Returns the maximum energy of the opponent's player character with the parameter player, returned from the method initialize in AIInterface.
getMyMotion(boolean player)	Vector<MotionData>	Returns MotionData of your player character with the parameter player, returned from the method initialize in AIInterface.
getOpponentMotion(boolean player)	Vector<MotionData>	Returns MotionData of the opponent's player character with the parameter player, returned from the method initialize in AIInterface.
getMyName(boolean player)	String	Returns the name of your player character with the parameter player, returned from the method initialize in AIInterface.
getOpponentName(boolean player)	String	Returns the name of the opponent's player character with the parameter player, returned from the method initialize in AIInterface.

MotionData

Method name	Value type	Description
getFrameNumber	int	Returns the number of frames in this motion.
getCancelAbleFrame	int	Returns the value of the first frame that the character can cancel this motion. If the current motion has reached this timing, it can be canceled with a motion having a lower value of motionLevel. If this motion has no cancelable period, the returned value will be -1.
getSpeedX	int	Returns the speed value in the horizontal direction that will be applied to the character when it does this motion.
getSpeedY	int	Returns the speed value in the vertical direction that will be applied to the character when it does this motion.
getHit	HitArea	Returns the information on the hit box (boundary box in other games), as shown in Fig.1 below.
getState	State	Returns the resulting state after running this motion.
getAttackHit	HitArea	Returns the information on the attack hit box, as shown in Fig.1 below.
getAttackSpeedX	int	Returns the horizontal speed of the attack hit box.
getAttackSpeedY	int	Returns the vertical speed of the attack hit box.
getAttackStartUp	int	Returns the number of frames in Startup.
getAttackActive	int	Returns the number of frames in Active.

getAttackHitDamage	int	Returns the damage value to the unguarded opponent hit by this skill.
getAttackGuardDamage	int	Returns the damage value to the guarded opponent hit by this skill.
getAttackStartAddEnergy	int	Returns the energy value added to the character when it uses this skill. If this value is negative and your character's energy is less than the absolute value of this value, you cannot use this skill.
getAttackHitAddEnergy	int	Returns the energy value added to the character when this skill hits the opponent.
getAttackGuardAddEnergy	int	Returns the energy value added to the character when this skill is blocked by the opponent.
getAttackGiveEnergy	int	Returns the energy value added to the opponent when it is hit by this skill.
getAttackImpactX	int	Returns the change in the horizontal speed of the opponent when it is hit by this skill.
getAttackImpactY	int	Returns the change in the vertical speed of the opponent when it is hit by this skill.
getAttackGiveGuardRecov	int	Returns the number of frames that the guarded opponent needs to resume to its normal status after being hit by this skill.
getAttackType	int	Returns the value of the attack type. 1 = high 2 = middle 3 = low 4 = throw
isAttackDownProperty	boolean	Returns the flag whether this skill can push down the opponent when hit.
getCancelAbleMotionLevel	int	Returns the value of the level that can cancel this motion. During cancelable frames, any motion whose level is below this value can cancel this motion.
getMotionLevel	int	Returns the value of this motion's level.
isControl	boolean	Returns the flag whether this character can run this motion with the motion's command.
isLandingFlag	int	Returns the flag whether a landing motion can cancel this motion.

HitArea

Method name	Value type	Description
getL	int	Returns the x-coordinate of the character's hit box's left boundary.

getR	int	Returns the x-coordinate of the character's hit box's right boundary.
getT	int	Returns the y-coordinate of the character's hit box's top boundary.
getB	int	Returns the y-coordinate of the character's hit box's bottom boundary.

FrameData

Method name	Value type	Description
getP1	CharacterData	Returns the first character's data.
getP2	CharacterData	Returns the second character's data.
getRemainingTime	long	Returns the remaining time.
getAttack	Deque<Attack>	Returns the projectile data of both characters.
getKeyData	KeyData	Returns the value of input information.
getMyCharacter(boolean player)	CharacterData	Returns CharacterData of your player character with the parameter player, returned from the method initialize in AIInterface.
getOpponentCharacter(boolean player)	CharacterData	Returns CharacterData of the opponent's player character with the parameter player, returned from the method initialize in AIInterface.

CharacterData

Method name	Value type	Description
getHp	int	Returns the character's hit points.
getEnergy	int	Returns the character's energy.
getX	int	Returns the character's most-left x-coordinate as shown in Fig. 2 below.
getY	int	Returns the character's most-top y-coordinate as shown in Fig. 2 below.
getLeft	int	Returns the character's hit box's most-left x-coordinate as shown in Fig. 2 below.
getRight	int	Returns the character's hit box's most-right x-coordinate as shown in Fig. 2 below.
getTop	int	Returns the character's hit box's most-top y-coordinate as shown in Fig. 2 below.
getBottom	int	Returns the character's hit box's most-bottom y-coordinate as shown in Fig. 2 below.
getSpeedX	int	Returns the character's horizontal speed, as shown in Fig. 2.

getSpeedY	int	Returns the character's vertical speed, as shown in Fig. 2.
getState	State	Returns the character's state: stand / crouch / in air / down
getAction	Action	Returns the character's action.
isFront	boolean	Return the character's facing direction.
isControl	boolean	Returns the flag whether this character can run a motion with the motion's command.
getRemainingFrame	int	Returns the number of frames that the character needs to resume to its normal status.
getAttack	Attack	Returns the non-projectile attack data that the character is using.

KeyData

Method name	Value type	Description
getPlayerOne	Key	Returns the first character's input information.
getPlayerTwo	Key	Returns the second character's input information.
getPlayer(int player)	Key	Returns Key of the player specified by the integer parameter player.
getPlayer(boolean player)	Key	Returns Key of the player specified by the boolean parameter player.
getMyKey(boolean player)	Key	Returns Key of your player character with the parameter player, returned from the method initialize in AIInterface.
GetOpponentKey(boolean player)	Key	Returns Key of the opponent's player character with the parameter player, returned from the method initialize in AIInterface.

Attack

Method name	Value type	Description
getHitAreaSetting	HitArea	Returns HitArea's setting information.
getHitAreaNow	HitArea	Returns HitArea's information of this attack hit box in the current frame.
getNowFrame	int	Returns the number of frames since this attack was used.
getPlayerNumber	int	Returns the integer number indicating the player of the attack. (0 for P1 and 1 for P2)
isPlayerNumber	boolean	Returns player's side flag.

getSettingSpeedX	int	Returns the absolute value of the horizontal speed of the attack hit box (zero means the attack hit box will track the character).
getSettingSpeedY	int	Returns the absolute value of the vertical speed of the attack hit box (zero means the attack hit box will track the character).
getSpeedX	int	Returns the horizontal speed of the attack hit box (minus when moving left and plus when moving right)
getSpeedY	int	Returns the vertical speed of the attack hit box (minus when moving up and plus when moving down)
getStartUp	int	Returns the number of frames in Startup.
getActive	int	Returns the number of frames in Active.
getHitDamage	int	Returns the damage value to the unguarded opponent hit by this skill.
getGuardDamage	int	Returns the damage value to the guarded opponent hit by this skill.
getStartAddEnergy	int	Returns the energy value added to the character when it uses this skill.
getHitAddEnergy	int	Returns the energy value added to the character when this skill hits the opponent.
getGuardAddEnergy	int	Returns the energy value added to the character when this skill is blocked by the opponent.
getGiveEnergy	int	Returns the energy value added to the opponent when it is hit by this skill.
getImpactX	int	Returns the change in the horizontal speed of the opponent when it is hit by this skill.
getImpactY	int	Returns the change in the vertical speed of the opponent when it is hit by this skill.
getGiveGuardRecov	int	Returns the number of frames that the guarded opponent needs to resume to its normal status after being hit by this skill.
getAttackType	int	Returns the value of the attack type. 1 = high 2 = middle 3 = low 4 = throw
isDownProperty	boolean	Returns the flag whether this skill can push down the opponent when hit.

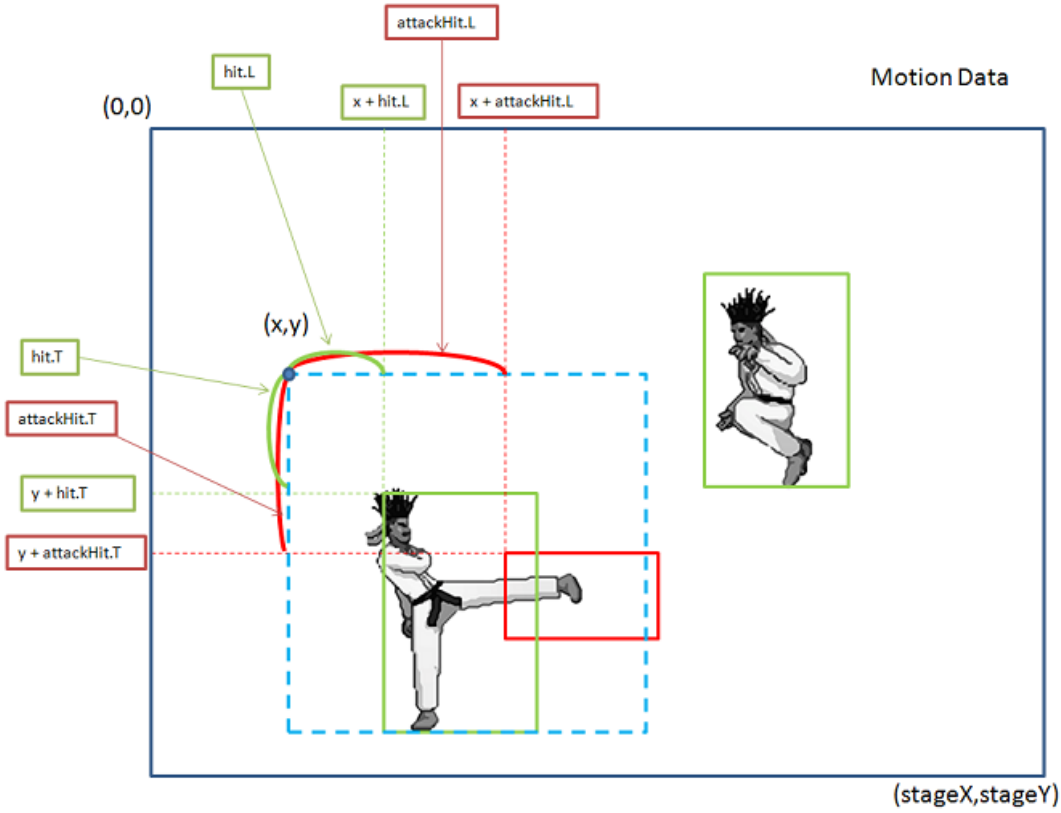


Figure.1: MotionData

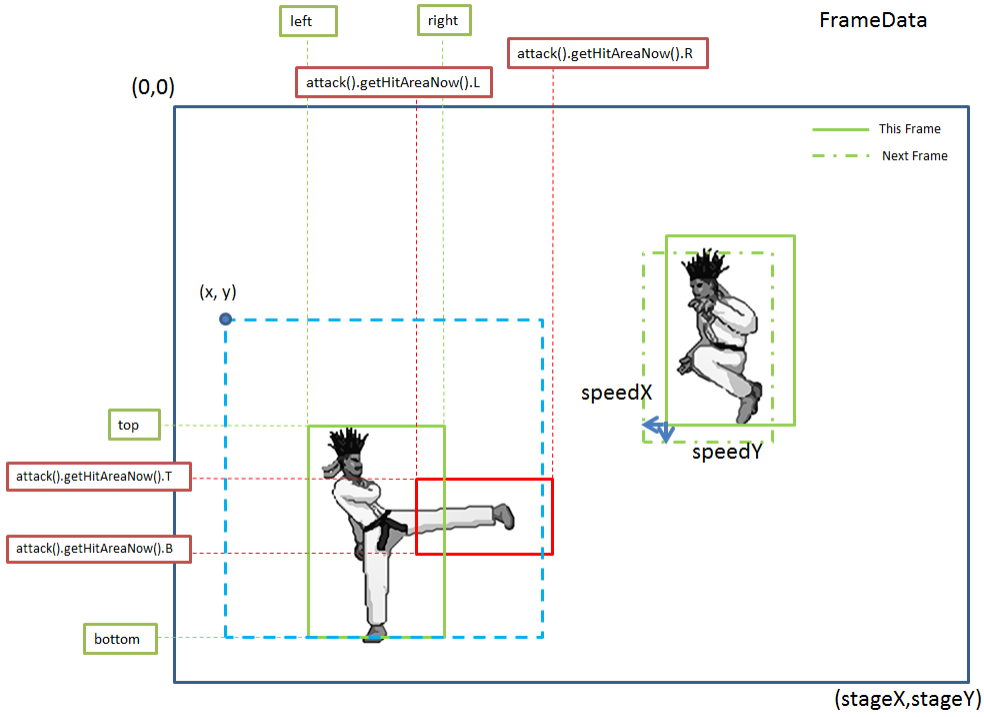
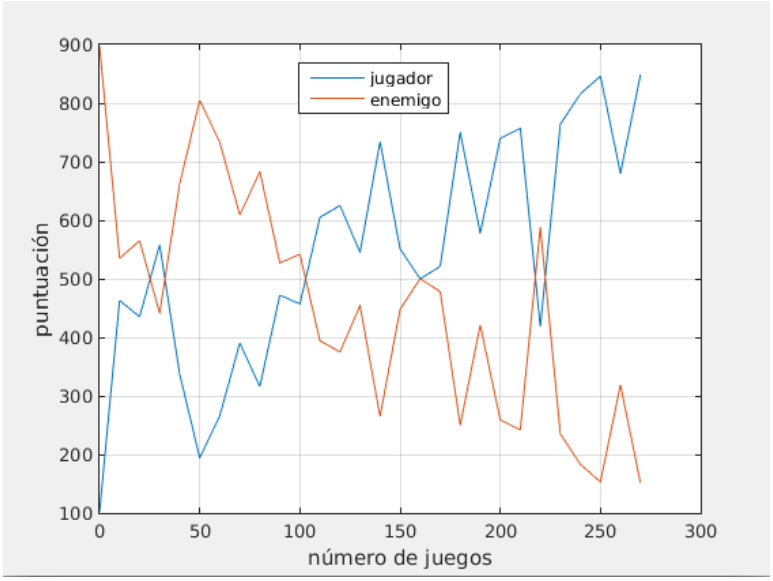


Figure.2: CharacterData

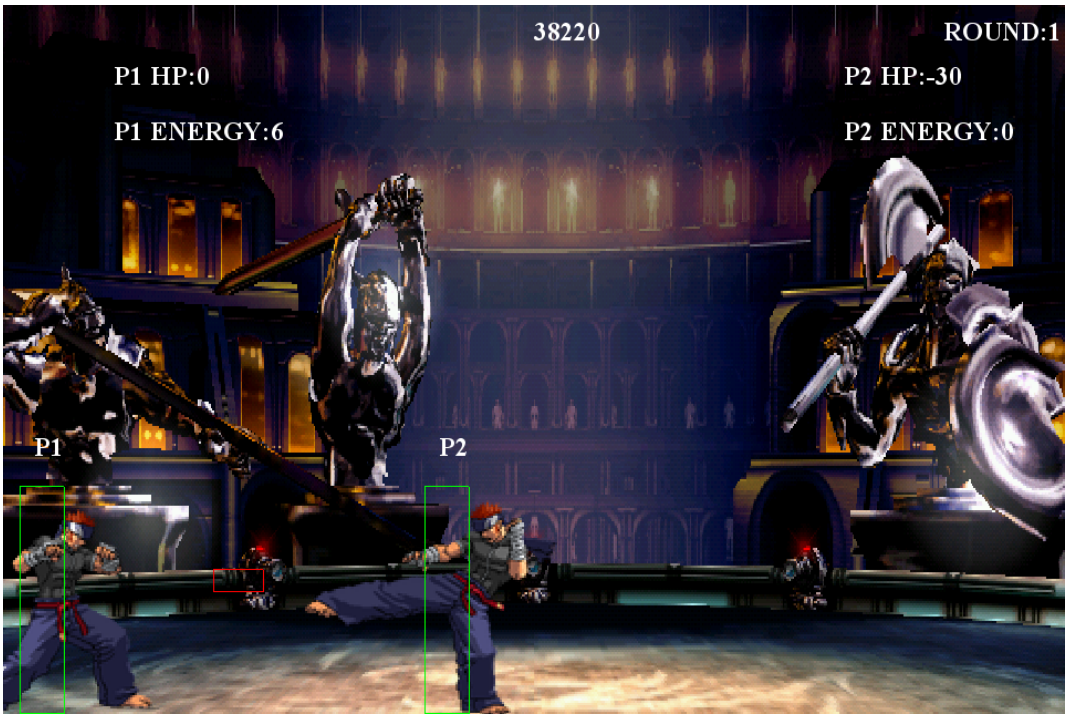
4. Resultados

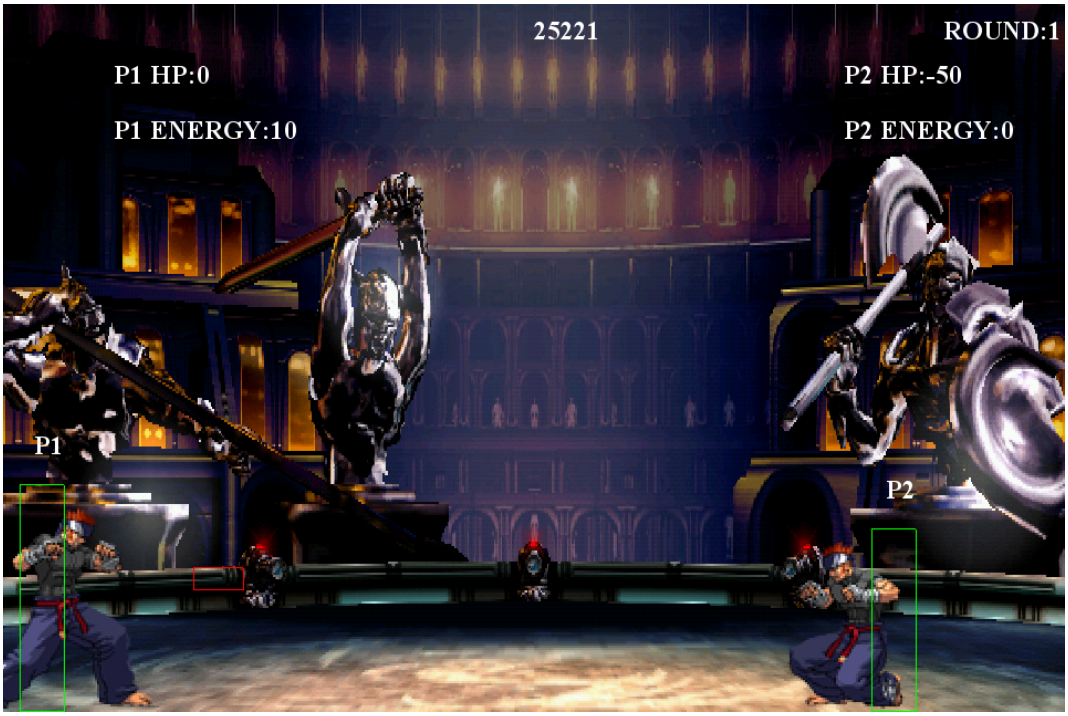
A pesar de la casi exagerada discretización de estados y acciones el algoritmo respondió positivamente. El jugador aprendió a atacar mayormente con poderes a distancia y a mantenerse a una distancia media. Esta estrategia se mantuvo efectiva ya que su contrincante era manipulado por un algoritmo que le dejaba tomar acciones aleatoriamente, por lo tanto, el contrincante no solía buscar agresivamente al jugador permitiendo que los ataques a distancia surtieran un efecto contundente.

A continuación se muestra una gráfica en donde se muestran los resultados de cada 10 peleas, promediados entre los 3 *rounds*:



Se puede apreciar la clara tendencia en el incremento de las puntuaciones del jugador conforme se va incrementando el número de peleas.


38220
ROUND:1
P1 HP:0
P2 HP:-30
P1 ENERGY:6
P2 ENERGY:0
P1
P2


25221
ROUND:1
P1 HP:0
P2 HP:-50
P1 ENERGY:10
P2 ENERGY:0
P1
P2

5. Conclusiones

El algoritmo Q-learning puede ser utilizado para resolver una cantidad casi infinita de problemas. El aprendizaje por refuerzo se ha mostrado más que adecuado en la actualidad para enfrentar retos en la comunidad científica.

A pesar de que esta implementación tiene un enfoque mucho menos espectacular, los resultados positivos demuestran el poder que los algoritmos actuales pudieran adquirir.

El algoritmo queda aún en etapas tempranas teniendo un futuro prometedor; para comenzar pudiera reducirse la discretización de los estados para tener predicciones más finas, también pudiera quitarse la restricción de las acciones permitiendo al agente elegir cualquier acción disponible en la interfaz. Con tan sólo estas medidas, tras entrenar el algoritmo debería de ser capaz de obtener aún mejores resultados.

Posteriormente la expansión vendría de la mano del incremento de variables de estado; pudieran tomarse en cuenta factores como: el tiempo restante del round, la cantidad de energía disponible, la cantidad de *frames* restantes de x movimiento, etc. Por supuesto también se podría mejorar enfrentándolo a oponentes de mayor inteligencia o dificultad.

Referencias

[1] Peter Norvig Stuart Russell. *Inteligencia Artificial: un enfoque moderno*. Prentice Hall, 2da edición edition, 2008.