Diseño e implementación de un agente inteligente Mario A.I.

Yuchen Du
Ingeniería de Telecomunicación
Universidad Carlos III de Madrid
100073084@alumnos.uc3m.es

Virginia Izquierdo Bermúdez
Ingeniería de Telecomunicación
Universidad Carlos III de Madrid
100072580@alumnos.uc3m.es

RESÚMEN

En este artículo se detalla el diseño y la implementación de un agente inteligente capaz de jugar al videojuego de Mario así cómo las técnicas utilizadas para este diseño.

Categorías y descriptores

Java [Lenguaje de programación]

Términos generales

Árbol de decisión, agente inteligente

Palabras claves

Mario, enemigos, agente, inteligencia, escenario

1. INTRODUCCIÓN

Para la implementación de este agente inteligente analizaremos el uso de técnicas utilizadas frecuentemente en aplicaciones de Inteligencia Artificial, tales como árboles de decisión. Después, construiremos nuestro propio árbol de decisión para la solución del problema de agente inteligente planteado en la página web de 2011 Mario AI Championship.

A continuación, haremos un estudio detallado del paquete de software proporcionado por dicha página web, sobre el cual se apoya nuestra implementación del agente de Mario inteligente. Para ello, analizaremos agentes básicos proporcionados dentro del paquete de software, tales como el ForwarJumpingAgent el cual simplemente salta hacia delante sin prestar ninguna atención a los distintos elementos presentes en escenario.

Adicionalmente, comprobaremos la funcionalidad de diversos métodos y variables implementados en otras clases dentro del paquete de software que nos serán de gran ayuda para la construcción en código Java de nuestro agente inteligente.

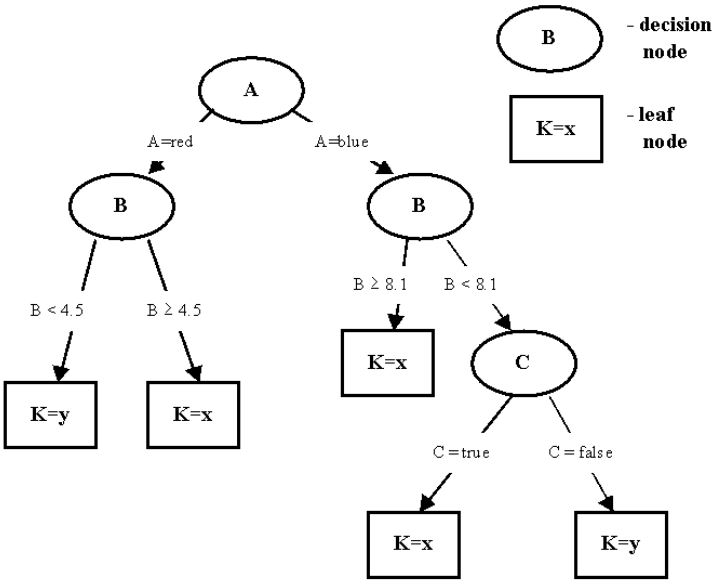
Por último, procedemos a implementar y programar el árbol de decisión construido previamente y a comprobar su funcionamiento real en diversos escenarios utilizando las distintas opciones de simulación disponibles.

2. ÁRBOLES DE DECISIÓN

Se trata de modelos de predicción en los cuales se utilizan técnicas mediante las que se pueden analizar decisiones secuenciales basadas en el uso de resultados. Estos árboles son utilizados para generar sistemas expertos, árboles de juegos o búsquedas binarias. Mediante su uso, un sistema dotado de Inteligencia Artificial podrá tomar decisiones en situaciones previamente definidas.

En un árbol de decisión, se dispone de unas ciertas entradas o situaciones, a partir de las cuales se devuelve un resultado, convergiendo así en una nueva situación donde una nueva decisión ha de ser tomada. Esta acción será realizada tantas veces como sea necesario, hasta llegar a un punto final, llamado hoja en nuestro árbol de decisión, donde se determina la acción a realizar tras el camino tomado, aquella que se ajusta a la situación requerida.

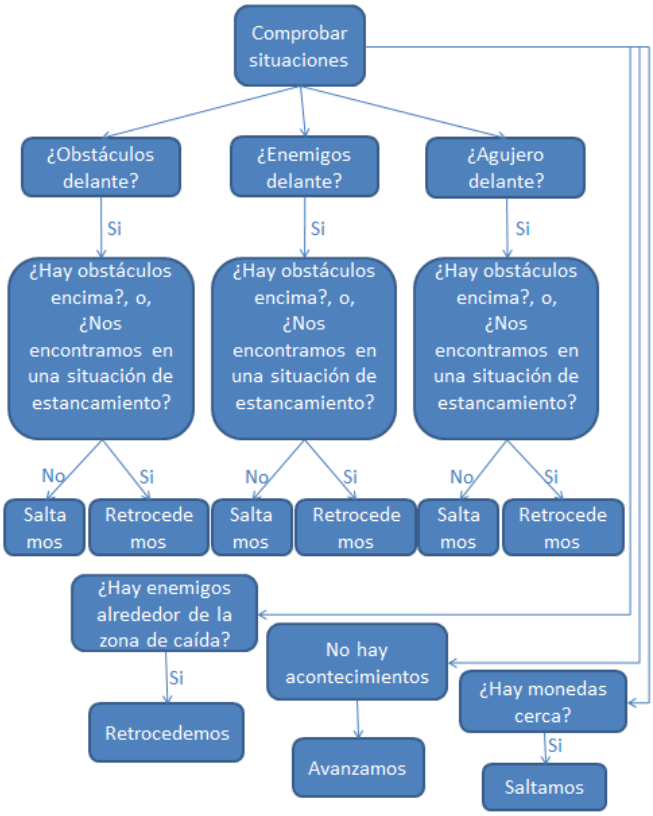
Un ejemplo de árbol de decisión, binario en este caso, se muestra a continuación.



3. DISEÑO DEL AGENTE

Para el diseño de nuestro agente, haremos uso de un árbol de decisión, no binario en éste caso, donde se tomarán decisiones dependiendo del escenario al que se enfrenta Mario en cada momento, indicándole la acción a realizar para tratar de llegar lo menos dañado posible al final del nivel.

Aquí mostramos el árbol de decisión que hemos diseñado para resolver el presente problema:



Cada nodo consiste en una posible situación a la que se podría enfrentar el agente a lo largo de una partida. A continuación, veremos detalladamente en qué consiste cada una de ellas, para tener especial cuidado a la hora de implementar nuestro sistema, evitando implementaciones que puedan dar lugar a comportamientos erróneos del agente inteligente.

3.1 Obstáculos delante

Si mientras caminamos observamos un objeto el cual no es franqueable, necesitaremos saltar para evitar situaciones en las que nos encontraríamos estancados. Será conveniente adelantarnos a esta situación y no detectar el obstáculo cuando se encuentra justo a nuestro lado, sino que convendrá intentar predecir la presencia de obstáculos a distancias un poco mayores. De ésta manera se mejorará el rendimiento del sistema al tratar de adelantarse en la resolución de problemas que pueden ser evitados fácilmente con un simple salto a tiempo.

3.2 Agujero delante

Al igual que en el caso anterior, será necesario predecir cuándo hay agujeros para tratar de evitar caer en ellos y consecuentemente perder de forma definitiva la partida. Para superar estos agujeros, tomaremos la misma acción que anteriormente, es decir, saltar en el momento necesario para evitar situaciones indeseadas una vez más.

3.3 Enemigos delante

Los enemigos también han de ser evitados para no ser dañados innecesariamente. Por ello, cuando veamos un enemigo próximo y siempre que sea posible (antes de saltar siempre habría que comprobar si dicha acción es conveniente, véase 3.4), saltaremos hacia delante para evitar ser alcanzados por un enemigo que pueda dañar a Mario y reducir su nivel en un rango (disparo → grande → pequeño → fin de partida).

3.4 Obstáculos encima

Será necesario también comprobar si cuando es necesario saltar, es decir, si nos encontramos en las situaciones 3.1, 3.2 o 3.3, esto es conveniente. En concreto, necesitamos comprobar si cuando se requiere realizar un salto, hay algún tipo de enemigo u obstáculo infranqueable por encima de Mario. Si se diese alguna de estas situaciones, lo más sensato sería no saltar, sino retroceder, ya que de lo contrario estaríamos arriesgando a ser dañados de forma innecesaria (enemigo encima) o saltar en vano (objeto infranqueable) y estar expuesto de nuevo a los peligros detectados anteriormente.

3.5 Estancados

Puede darse la situación en la que tratamos de ir hacia delante y saltar para evitar algún tipo de obstáculo, pero no podemos franquearlo por encontrar nuevos impedimentos y quedarnos estancados, por ejemplo si chocamos con un objeto que se encuentra justo encima nuestro y a la vez tenemos una pared o una tubería delante nuestro no podríamos avanzar con un simple salto hacia delante. Por ello, la acción correcta a realizar si esta situación ocurre es la descrita en 3.7, es decir, retroceder y probar a saltar de nuevo.

3.6 Saltar

Si hemos llegado a esta hoja del árbol, la acción final a realizar es saltar hacia delante. La duración y por tanto la altura del salto depende del propósito de dicho salto, ya que depende de si estamos intentando evitar un obstáculo o si queremos recoger más monedas, nos interesa saltar alturas diferentes.

3.7 Retroceder

Finalmente, si llegamos a esta hoja, ya sea para evitar una situación de estancamiento, o para escapar de un enemigo cuando no era conveniente saltar directamente, retrocederemos hacia detrás durante un tiempo determinado para después poder saltar hacia delante sin chocarnos con ningún tipo de obstáculo que podríamos habernos encontrado.

4. IMPLEMENTACIÓN DEL AGENTE

Para la implementación del agente en Java, con ayuda del código proporcionado por 2011 Mario AI Championship necesitaremos analizar ciertas funciones y variables ya implementadas o definidas, que nos serán de apoyo a la hora de programar.

4.1 Escenario

En primer lugar, y como información más importante será necesario observar el escenario en el que se mueve Mario. Según las descripciones de la página web de la competición, este escenario consiste en una matriz de 22 x 22 en el que Mario se encuentra siempre en el centro de la perspectiva, es decir, en esta matriz siempre aparece en la columna 11 y en la fila 11.

En esta matriz, podemos encontrar todo tipo de información acerca de los objetos que aparecen en pantalla, tales como enemigos, obstáculos o monedas.

Existen diversos métodos que permiten acceder a la información de esta matriz en todo momento con distinto nivel de detalle, pudiendo seleccionar el tipo de información que se quiere conocer, bien sea los enemigos, los obstáculos o ambas a la vez.

Por simplicidad utilizaremos los siguientes métodos: getLevelSceneObservationZ (int level), para ver únicamente los obstáculos en el escenario; getEnemiesObservationZ (int level), en este caso para ver solo los obstáculos; y por último, getMergedObservationZZ(int level, int level) en el cual se pueden observar ambas informaciones.

Es importante mencionar que, a diferencia de la matriz de perspectiva completa de dimensiones 22 x 22, estas funciones que simplifican la tarea de obtención de información dividiéndola según su clase, nos devuelven una matriz de perspectiva de dimensiones 19 x 19 con la información requerida según la función invocada. Al igual que antes, Mario siempre estará situado en el centro de perspectiva, que en este caso pasa a ser la fila 10 y la columna 10.

La variable level permite modificar el nivel de detalle de los objetos en la pantalla que queremos obtener. Cuando se le asigna el valor 2 a esta variable de entrada, los datos que devuelve la función tendrán la forma más simple (mayoritariamente 0s y 1s) facilitando su procesamiento y manipulación posterior, mientras que si se le asigna 0 a esta variable, los datos obtenidos tendrán la máxima precisión, siendo posible la distinción entre distintos tipos de enemigos u obstáculos. A la hora de programar, dependiendo del propósito y las necesidades de la función determinada, utilizaremos un nivel de detalle mayor o menor, para adaptarnos mejor a la situación concreta.

4.2 Variables utilizadas

A parte de estos tres métodos que utilizamos para obtener información acerca del entorno en el que se desenvuelve Mario, aprovecharemos también otros atributos que se encuentran distribuidos en el paquete de software Mario AI:

4.2.1 isMarioOnGround

Indica si en un momento determinado Mario está sobre una plataforma fija, o si está en medio del aire. Este atributo nos será de gran utilidad para determinar si estamos preparados para detectar nuevos enemigos u obstáculos y saltar de nuevo, o si por el contrario tenemos que prestar mayor atención a la zona donde Mario va a caer, comprobando la proximidad a enemigos y decidiendo si se debe retroceder hacia atrás para evitar un posible choque y daño indeseado.

4.2.2 Mario.fire

Este atributo nos permite saber si Mario aún mantiene en su situación inicial con capacidad de disparo, o si ya ha sufrido daños y consecuentemente es grande o pequeño, inhabilitado para disparar. Esto es útil ya que, en el caso de que podamos disparar, sería muy interesante disparar continuamente para destruir los enemigos que vienen de enfrente y por tanto para aumentar las posibilidades de sobrevivencia. Nótese que, al tratarse el botón de disparar el mismo que sirve para correr, mientras Mario disponga de la opción de disparar, lo hará repetidamente mientras que va avanzando rápidamente por el nivel. Después de sufrir algún daño y perder la capacidad de disparo, perdemos protección contra los enemigos que vienen de frente, por lo que dejaremos de correr para mejorar nuestra reacción contra las distintas situaciones adversas que puedan tener lugar.

4.2.3 marioEgoRow y marioEgoCol

Estas dos variables indican la posición de Mario en la matriz de perspectiva de escenario en un momento determinado.

4.2.4 action

A parte de los atributos anteriores que utilizamos para obtener información acerca del escenario presente, también cabe mencionar que, para indicar las acciones que debe tomar Mario en cada momento, modificamos el array de variables boolean *action*, que contiene la información acerca de qué botones deben ser pulsados durante una ronda determinada. Los que utilizamos principalmente son:

- action[Mario.Key_RIGHT]
- action[Mario.Key_LEFT]
- action[Mario.Key_JUMP]
- action[Mario.Key_SPEED]

Que tienen nombres bastante intuitivos.

4.3 Enemigos

Siempre que sea posible, como dijimos en la parte de diseño, al encontrarnos con un enemigo al lado nuestro trataremos de saltar para no ser dañados. En caso de que esto no fuera posible, bien porque hay más enemigos encima o porque tenemos algún obstáculo que nos lo impide, nos desplazaremos hacia atrás para poder huir de él y tomar distintas decisiones más adelante. Por lo general, esta funcionalidad no es utilizada muy frecuentemente, pero en ocasiones resulta de vital importancia ya que nos evita ser dañados por un estrecho margen. También se da la situación cuando estamos cayendo después de un salto, que Mario hace contacto muchas veces (sin llegar a saltar encima) con un enemigo justo cuando está a punto de tocar el suelo, por lo que es imposible

realizar otro salto para evitar ser dañado (para realizar un salto es necesario que Mario esté sobre alguna plataforma firme). Para prevenir y solucionar este peligro, analizamos la posibilidad de que algún enemigo se encuentre alrededor de la zona de aterrizaje, y cuando se dé el caso, trataremos de retroceder o bien frenar la trayectoria y así evitar algunos enemigos que podrían habernos atacado si simplemente hubiéramos seguido el salto con nuestra trayectoria inicial.

4.4 Obstáculos o Monedas

Aparte de los enemigos, a la hora de comprobar obstáculos, nos interesa diferenciar si nos encontramos ante un objeto sólido, al que haría falta realizar un gran salto por encima para superarlo y así poder seguir avanzando en el nivel, o de monedas, que en vez de esquivarlas lo que realmente nos interesa es recolectar la mayor cantidad posible, realizando saltos pequeños para intentar coger todas las monedas cercanas. En este caso, esta tarea es relativamente fácil, pues utilizando un nivel de detalle 2, bajo la presencia de monedas, el método para obtener información de escenario devuelve *2*s en aquellas posiciones donde éstas se encuentran, diferenciándolas fácilmente de los obstáculos indeseados, que corresponderían a '1's, evitando franquearlas de forma innecesaria.

4.5 Flores enemigas

Por lo general, para obtener la matriz con la información del escenario será suficiente con un nivel de detalle 2, pero en ocasiones, como por ejemplo cuando tratamos de evitar las flores enemigas que se esconden en las tuberías, nos interesara saber específicamente si los enemigos que encontramos delante se tratan de dichas flores o no, ya que dependiendo de eso actuaremos de una forma u otra: en el caso de que se trate de enemigos comunes, no habría ningún problema y por tanto actuaremos igual que ante el resto de enemigos, mientras que cuando detectamos flores enemigas, es mucho más seguro esperar a que se refugien de nuevo en las tuberías antes de avanzar de nuevo hacia delante.

Del código fuente del paquete software, podemos saber que cuando se utiliza un nivel de detalle 0, una flor enemiga será "traducida" con el valor de *91* cada vez que se invoque el método getEnemiesObservationZ(...).

De esta forma, cuando detectamos un obstáculo delante, y cuando no observamos ningún impedimento para saltar por la existencia de obstáculos, procedemos a comprobar la existencia de flores enemigas en las celdas siguientes. En caso de encontrarlas, nos pegaremos a la tubería, esperaremos a que dichas flores bajen y cuando ya el camino está libre, saltaremos hacia delante.

5. RESULTADOS

Una vez implementado nuestro sistema podemos comprobar cómo este funciona aceptablemente, siendo capaz de terminar diversos escenarios, con distintas dificultades y obstáculos sin ser apenas dañados. También es cierto que nuestro agente no es infalible y en ocasiones comete fallos que le suponen la pérdida de fuerza, sin embargo las consecuencias no son tan graves en comparación con la dificultad de la implementación de soluciones a dichos problemas.

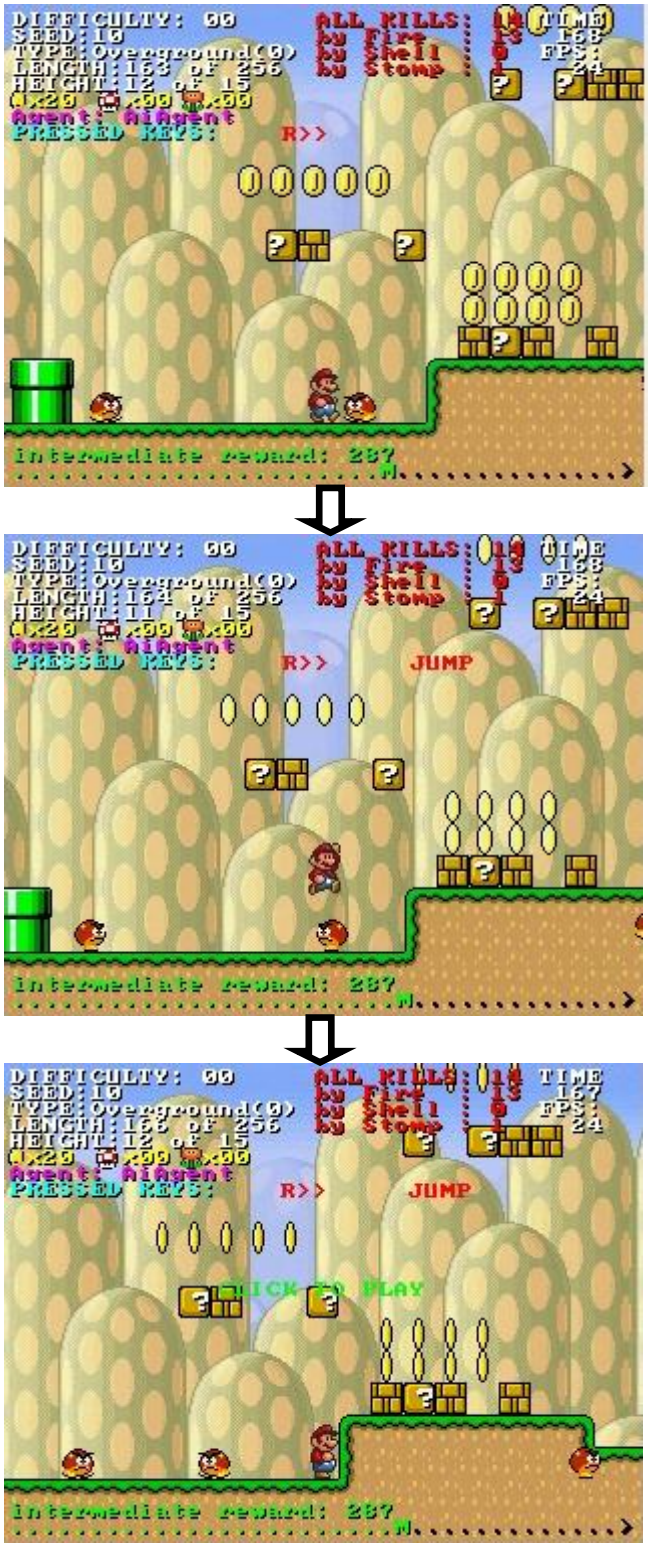
Por tanto, hemos tratado de llegar a una implementación en la que exista un compromiso entre funcionalidad y dificultad, consiguiendo, en nuestra opinión una solución muy acertada en cuanto a resolución de problemas por parte de Mario y la dificultad a la que su implementación está sometida.

No obstante, comentaremos posteriormente ciertas mejoras que pueden realizarse para conseguir un agente mucho más inteligente y capaz de superar problemas mayores a los que nuestro agente puede enfrentarse.

Antes de eso, mostraremos algunas de nuestras implementaciones más destacadas, demostrando su correcto funcionamiento, obteniendo el efecto deseado según lo explicado anteriormente, atendiendo a las decisiones que se toman tras recorrer nuestro árbol de decisión definido.

5.1 Enemigos

La tarea básica para hacer que nuestro agente funcione es tratar de esquivar enemigos que se acercan hacia nosotros, a nivel del suelo. Para ello, antes de que estos nos ataquen, daremos un salto evitándolos de forma exitosa, tal y como se muestra en las imágenes a continuación:

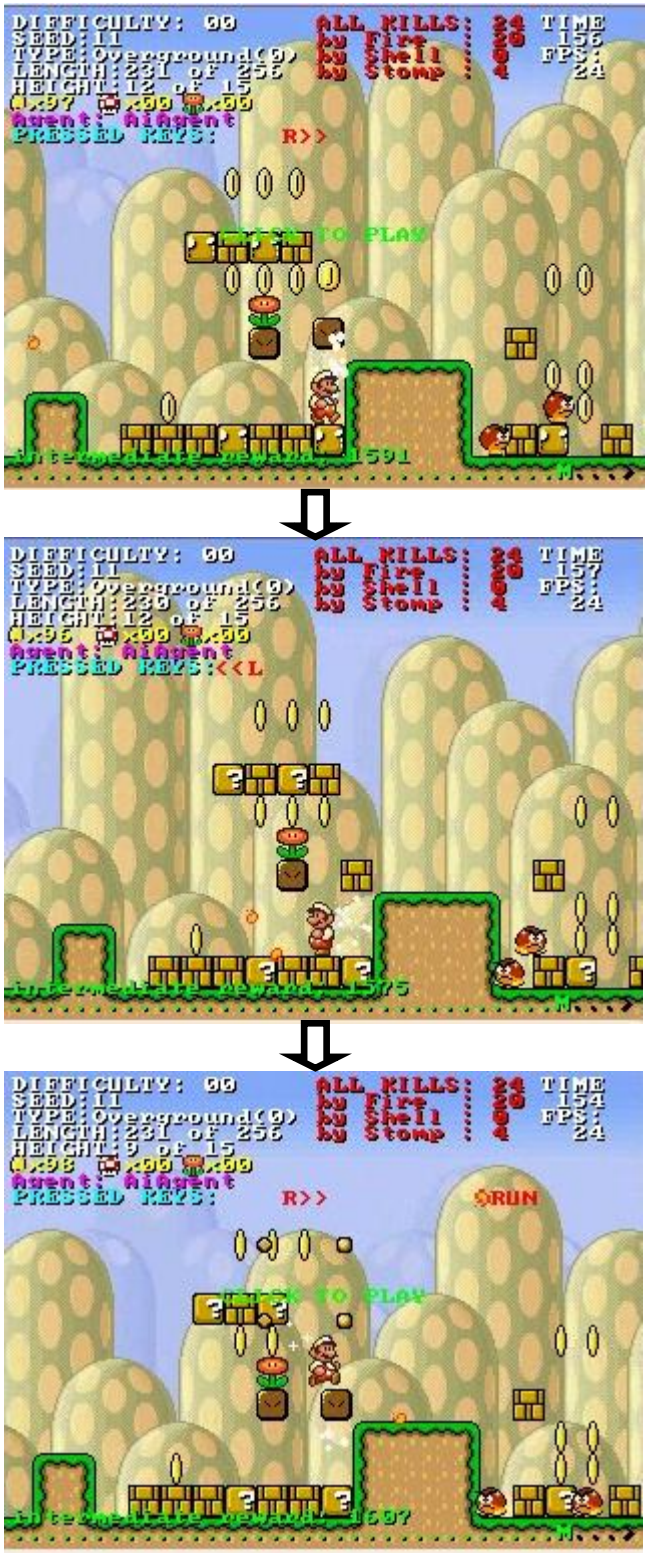


5.2 Flores enemigas

Otra acción a realizar, es esquivar flores enemigas, emergente de las tuberías del escenario de Mario. Para ello, ya hemos explicado que trataremos de esperar, hasta que esta se ha escondido para saltar, evitando que la flor nos dañe:

5.3 Situaciones de estancamiento

Otra situación muy incómoda es cuando Mario se encuentra atascado entre varios bloques irrompibles, ya que si no se toman medidas, Mario se mantendrá en la misma posición infinitamente, pues ni puede seguir hacia delante ni puede saltar, al chocar con algún bloque de su alrededor. Por ello, será necesario hacer que Mario retroceda cuando se encuentre en esta situación, pudiendo completar el escenario tal y como se muestra:



6. POSIBLES MEJORAS

Debido a limitaciones tales como el tiempo disponible, nuestra implementación no resulta ser la más idónea, ya que existen detalles en el funcionamiento del sistema que podrían ser mejorados realizando implementaciones más complejas.

Una de las posibles mejoras que a priori parece asequible es la siguiente: del motor de generación de niveles automático que utiliza el paquete de software, muchas veces genera escenarios en el cual hay muchos enemigos en distintos niveles moviéndose hacia la izquierda (hacia Mario) y cayéndose. Utilizando el agente que hemos implementado, se necesita mucha suerte para poder esquivar todos, ya que no es nada fácil evitar contacto con 5 o más enemigos cayendo. A simple vista, una posible solución sería: cuando se detecta una gran cantidad de enemigos por delante, en vez de seguir hacia delante, Mario puede permanecer quieto o incluso retroceder un poco, y esperar hasta que los enemigos hayan caído sobre plataforma firme para seguir avanzando, utilizando nuestro árbol de decisión para esquivar los enemigos.

Sin embargo, esta solución no es óptima del todo. Uno de los problemas generados por intentar esquivar los enemigos saltándolos es que, cuando hay una presencia numerosa de enemigos, es complicado evitar el contacto con todos. Nosotros intentamos detectar si en el aterrizaje de un salto es posible hacer colisión con algún enemigo y así cambiar la trayectoria del salto. Pero esto no evita colisiones al 100% y muchas veces perdemos vida justo en los instantes siguientes de un salto.

Además, en los niveles avanzados que se generan con el paquete de software, son muy comunes los enemigos con alas que pueden moverse en todas las direcciones. Dado el caso, no sería de utilidad esperar a que todos los enemigos aterricen, ya que es algo que no ocurriría.

Para hacer frente a esta situación y resolver los problemas que puedan presentarse, es necesario diseñar una implementación más compleja: haría falta conocer las posiciones de todos los enemigos

y los obstáculos presentes y, teniendo en cuenta las direcciones de movimiento de los enemigos, calcular hacia que posiciones sería conveniente saltar (ya sea un hueco entre los enemigos o saltar sobre algún enemigo, aunque esta última opción implicaría un pequeño salto forzado después de matar al enemigo, que habría que tener en cuenta para contemplar posibles resultados). Al mismo tiempo, habría que tener conocimiento sobre las capacidades de Mario, es decir, sus velocidades de movimiento y de salto, para ver de la lista de posiciones seguras hacia dónde es factible saltar. Y eso no es todo: habría que tener un control exhaustivo en todo momento sobre las posiciones futuras de los enemigos y de Mario para así evitar cualquier sorpresa.

Como podemos imaginar, esta solución, que funcionaría de forma correcta en todas las situaciones posibles, requiere una enorme complejidad ya que se necesitaría estudiar ecuaciones sobre los movimientos de Mario y los enemigos, y se escapan del rango de requerimientos de este trabajo. En cualquier caso, el sistema para evitar enemigos que hemos implementado funciona razonablemente bien para niveles de dificultad limitada, superando incluso a las capacidades de un jugador humano con poca experiencia.

También existen otros aspectos en donde se puede implementar mejoras, tales como recolectar champiñones y flores para mejorar el estado de Mario, así como colectar todas las monedas que aparecen en pantalla. Sin embargo, todo eso aumentaría en gran medida la complejidad del sistema, por lo que para el presente diseño se ha decidido por un sistema más simple y al mismo tiempo veloz, que es capaz de resolver problemas de cierta magnitud.

7. CÓDIGO FUENTE

```
/**
 * Inteligencia en Redes de Comunicaciones
 * Curso 2011 - 2012
* Diseño e Implementación de un agente inteligente Mario
A.I.
* @author - Yuchen Du 100073084
 * @author - Virginia Izquierdo Bermúdez 100072580
*/
package ch.idsia.agents.controllers;
import ch.idsia.agents.Agent;
import ch.idsia.benchmark.mario.engine.sprites.Mario;
import ch.idsia.benchmark.mario.environments.Environment;
import
ch.idsia.benchmark.mario.environments.MarioEnvironment;
public class AiAgent extends BasicMarioAIAgent implements
Agent {
        // Variable para controlar el salto
        int trueJumpCounter = 0;
        // Variable para controlar retrocesos
        int trueBufferCounter = 0;

        public AiAgent() {
                super("AiAgent");
                reset();
        }
        /**
         * Metodo de inicializacion Estado inicial: todos
        los botones sin presionar
         */
        public void reset() {
                action = new
                boolean[Environment.numberOfKeys];
                action[Mario.KEY_RIGHT] = false;
                action[Mario.KEY_SPEED] = false;
                action[Mario.KEY_JUMP] = false;
                trueJumpCounter = 0;
                trueBufferCounter = 0;
        }
        /**
         * Metodo para chequear si hay enemigos delante
         *
         * @param environment
         *            : el entorno en el que se
         *            desenvuelve Mario. De ahi sacaremos
         *            la informacion necesaria para
         *            interactuar con el resto de
         *            elementos que aparecen en la
         *            pantalla.
         * @return boolean existencia de un enemigo
         *            cercano Simplemente comprobamos
         *           la existencia de enemigos en la
         *            posicion consecutiva del array
         */
        private boolean checkEnemiesFront(Environment
        environment) {
                byte[][] enemies =
                environment.getEnemiesObservationZ(2);
                if (enemies[marioEgoRow][marioEgoCol + 1]
                != 0) {
                        return true;
                } else {
                        return false;
                }
        }
        /**
         * Metodo para chequear si hay obstaculos
         *
         * @param environment
         *            : el entorno en el que se
         *            desenvuelve Mario.
         * @return boolean existencia de obstáculos
         *            delante Comprobamos la
         *            existencia de elementos bloqueantes
         *            por delante de la posicion
         *            actual de la figura de Mario
         *            Prestamos atencion a si el elemento
         *           se trata de una moneda ya que de ser
         *            asi, no se le considera como
         *           un elemento bloqueante
         */
        private boolean checkObsFront(Environment
        environment) {
                byte[][] obstacles =
                environment.getLevelSceneObservationZ(2);
                if ((obstacles[marioEgoRow][marioEgoCol +
                1] != 0 &&
                obstacles[marioEgoRow][marioEgoCol + 1] !=
                2) || (obstacles[marioEgoRow][marioEgoCol
                + 2] != 0 &&
                obstacles[marioEgoRow][marioEgoCol + 2] !=
                2) || (obstacles[marioEgoRow][marioEgoCol
                + 3] != 0 &&
                obstacles[marioEgoRow][marioEgoCol + 3] !=
                2) ||
                (obstacles[marioEgoRow - 1][marioEgoCol +
                1] != 0 &&
```

```
        obstacles[marioEgoRow - 1][marioEgoCol +
        1] != 2)) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo para chequear si hay un agujero delante
 *
 * @param environment
 *            : el entorno en el que se
 *            desenvuelve Mario.
 * @return boolean existencia un agujero delante
 *            Verificamos si hay en la
 *           columna delante existe una posicion
 *           vacia
 */
private boolean checkGapFront(Environment
environment) {
        byte[][] obstacles =
        environment.getLevelSceneObservationZ(2);
        if (obstacles[marioEgoRow + 1][marioEgoCol
        + 1] == 0) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo para chequear si se puede saltar
 *
 * @param environment
 *            : el entorno en el que se
 *             desenvuelve Mario.
 * @return boolean si debemos saltar Comprobamos
 *            la existencia de enemigos
 *            por encima de la posicion de Mario.
 *            Si se diera la situación, no
 *            se debe saltar para evitar colision.
 */
private boolean checkJump(Environment environment)
{
        byte[][] things =
        environment.getEnemiesObservationZ(2);
        if (things[marioEgoRow - 1][marioEgoCol]
        != 0 ||
         things[marioEgoRow - 2][marioEgoCol] != 0
        ||
        things[marioEgoRow - 3][marioEgoCol] !=
        0) {
                return false;
        } else {
                return true;
        }
}
/**
 * Metodo para chequear si se ha llegado a una
 * situacion de estancamiento
 *
 * @param environment
 *            : el entorno en el que se
 *            desenvuelve Mario.
 * @return boolean si Mario esta estancado en
 *            algun punto Verificamos si
 *            Mario se encuentra en una situacion
 *            de estancamiento, es decir,
 *            si por delante de el hay un muro
 *            infranqueable, pero al mismo
 *           tiempo existe elementos irrompible
 *            por encima de el
 */
private boolean stucked(Environment environment) {
        byte[][] things2 =
        environment.getLevelSceneObservationZ(2);
        if ((things2[marioEgoRow - 2][marioEgoCol]
        == -60 || things2[marioEgoRow -
        3][marioEgoCol] == -60)
                        &&
        (things2[marioEgoRow][marioEgoCol + 1] ==
        1 && things2[marioEgoRow - 1][marioEgoCol
        + 1] == 1)) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo para comprobar si hay monedas cerca
 *
 * @param environment
 *            : el entorno en el que se
 *             desenvuelve Mario.
 * @return boolean si hay monedas cerca
 *            Comprobamos en los alrededores de la
 *            posicion actual de Mario la
 *            existencia de monedas
 */
private boolean areCoins(Environment environment)
{
        byte[][] coins =
        environment.getLevelSceneObservationZ(2);
        if (coins[marioEgoRow - 1][marioEgoCol] ==
        2 || coins[marioEgoRow - 1][marioEgoCol +
        1] == 2) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo para comprobar si hay enemigos    cerca
 * en los instantes finales de
 * un salto
 *
 * @param environment
 *            : el entorno en el que se
 *             desenvuelve Mario.
 * @return boolean la existencia de peligro
 *            cercana Comprobamos si hay
 *            enemigos alrededor de la zona en
 *            donde caeria mario
 */
private boolean checkWhileFalling(Environment
environment) {
        byte[][] things =
        environment.getEnemiesObservationZ(2);
        if (things[marioEgoRow + 1][marioEgoCol +
        1] != 0  || things[marioEgoRow +
        1][marioEgoCol + 2] != 0 ||
        things[marioEgoRow + 1][marioEgoCol + 3]
        != 0 || things[marioEgoRow +
        2][marioEgoCol + 3] != 0 ||
        things[marioEgoRow + 2][marioEgoCol + 2]
        != 0) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo para comprobar si ha salido una flor
 * enemiga de alguna de las
 * tuberías cercanas
 *
 * @param environment
 *            : el entorno en el que se
 *            desenvuelve Mario.
 * @return boolean la existencia de una flor
 *            carnivora Chequeamos en los
 *            puntos cercanos si hay alguna flor
 *           enemiga
 */
private boolean checkFlower(Environment
environment) {
        byte[][] things =
        environment.getEnemiesObservationZ(0);
        if (things[marioEgoRow - 2][marioEgoCol +
        1] == 91 || things[marioEgoRow -
        3][marioEgoCol + 1] == 91 ||
        things[marioEgoRow - 4][marioEgoCol + 1]
        == 91 || things[marioEgoRow -
        2][marioEgoCol + 2] == 91 ||
        things[marioEgoRow - 3][marioEgoCol + 2]
        == 91 || things[marioEgoRow -
        4][marioEgoCol + 2] == 91) {
                return true;
        } else {
                return false;
        }
}
/**
 * Metodo auxiliar para imprimir el array por la
 * pantalla
```

```
 *
 * @param environment
 *            : el entorno en el que se
 *            desenvuelve Mario.
 * @return void
 */
private void printout(Environment environment) {
        byte[][] things =
        environment.getLevelSceneObservationZ(2);
        for (int i = 0; i < 19; i++) {
                System.out.println("");
                for (int j = 0; j < 19; j++) {
                System.out.print(things[i][j]+"
                ");
                }
        }
}
/**
 * Metodo para decidir en cada momento las
 * acciones a tomar
 *
 * @return boolean[] el array indicando los
 *         botonos que se deben mantener
 *         pulsado Utilizando los métodos
 *         definidos anteriormente,
 *         comprobamos la situacion en la que se
 *         encuentra Mario en cada
 *         momento y decidimos los botones que
 *        debemos accionar
 */
public boolean[] getAction() {

        MarioEnvironment env =
        MarioEnvironment.getInstance();
        // Si Mario se dispone de la habilidad de
        // disparo,
        // encendemnos y apagamos alternativamente
        // el botón
        // de SPEED y así disparar continuamente
        if (Mario.fire) {
                action[Mario.KEY_SPEED] =
                !action[Mario.KEY_SPEED];
        }
        // En caso contrario, avanzamos a
        // velocidad regular
        else {
                action[Mario.KEY_SPEED] = false;
        }
        // Primero comprobamos si Mario esta
        // retrocediendo
        // para evitar un enemigo o un obstaculo
        if (trueBufferCounter > 0) {
                trueBufferCounter--;
                // Si este contador se ha llegado
                // a 0, dejamos de mover
                // hacia la izquierda y volvemos
                // hacia la derecha
                if (trueBufferCounter == 0) {
                        action[Mario.KEY_RIGHT]
                        = true;
                        action[Mario.KEY_LEFT] =
                        false;
                }
        }
        // Ahora comprobamos si Mario estaba
        // saltando,
        // gracias al contador trueJumpCounter
        if (trueJumpCounter > 0) {
                trueJumpCounter--;
                // Si mario ya se encuentra sobre
                // alguna plataforma,
                // reseteamos el contador
                if (isMarioOnGround) {
                        trueJumpCounter = 0;
                }
                // Si el contador esta a 0,
                // dejamos de pulsar el boton de
                // saltar
                if (trueJumpCounter == 0) {
                        action[Mario.KEY_JUMP] =
                        false;
                }
        }
        // Si no se da lugar ninguna de las
        // situaciones anteriores,
        // Comprobamos si hay enemigos, obstaculos
        // o agujeros delantes
        // y asi determinar las acciones a
        // desarrollar
        else if ((checkEnemiesFront(env) ||
        checkObsFront(env) || checkGapFront(env))
        && trueBufferCounter == 0) {
                if (isMarioOnGround == false) {
                        // Si hay enemigos
                        // cercanos al punto de
                        // caida,
                        // retrocedemos
                if (checkWhileFalling(env)) {
                action[Mario.KEY_RIGHT] = false;
                action[Mario.KEY_LEFT] = true;
                trueBufferCounter = 1;
                        }
                }
                // Si se puede saltar con
                // seguridad y Mario no se
                // encuentra en una situacion de
                // estancamiento, movemos hacia
                // la derecha
                else if (checkJump(env) &&
                !stucked(env)) {
                        action[Mario.KEY_RIGHT]
                        = true;
                        // Si no hay una flor
                        // enemiga fuera de una
                        // tuberia, saltamos
                        // En caso contrario
                        // esperamos hasta que
                        // la flor se refugie
                if (!checkFlower(env)) {

                action[Mario.KEY_JUMP] = true;
                        trueJumpCounter = 7;
                        }
                }
                // Si no es asi, retrocedemos
                // hacia la izquierda
                else {
                action[Mario.KEY_RIGHT] = false;
                action[Mario.KEY_LEFT] = true;
                trueBufferCounter = 3;
                }
        }
        // Si hay monedas cerca
        else if (areCoins(env) &&
        trueBufferCounter == 0) {
                // Si saltar no implica ningun
                // peligro,
                // saltamos un poco para coger
                // mas monedas
                if (checkJump(env)) {
                        action[Mario.KEY_RIGHT]
                        = true;
                        action[Mario.KEY_JUMP] =
                        true;
                        trueJumpCounter = 1;
                }
        }
        // Finalmente si no pasa nada,
        // simplemente
        // movemos hacia la derecha
        else if (trueBufferCounter == 0) {
                action[Mario.KEY_RIGHT] = true;
                action[Mario.KEY_JUMP] = false;
        }
        return action;
    }
}
```

8. REFERENCIAS

[1] Página web de Mario AI Championship 2011: http://www.marioai.org/

[2] Ejemplo del árbol de decisión tomado: http://dms.irb.hr/tutorial/tut_dtrees.php