Práctica 2
Algoritmos Genéticos y Evolutivos

Competición de Inteligencia Artificial Genérica aplicada a Videojuegos

Universidad
Carlos III de Madrid

López Rayo, Raúl
100073776@alumnos.uc3m.es

Jerez Garrido, Daniel
100303628@alumnos.uc3m.es

Rodríguez Maximiano, Rubén
100303579@alumnos.uc3m.es

Poblete Sandoval, Juan
100303554@alumnos.uc3m.es

Buceta Ojeda, Luis
100303573@alumnos.uc3m.es

Rodríguez Grillo, Adrián
100316457@alumnos.uc3m.es

Diciembre 2016

Índice

Índice de figuras

1. Introducción

La utilización de la inteligencia artificial en los videojuegos es uno de los elementos más representativos de estos. Durante muchos años, distintas técnicas de inteligencia artificial han sido utilizadas para modificar el estado de los personajes no controlados (o NPCs) de los juegos, y han ayudado a dar una capa de dificultad y reto a los videojuegos. Sin embargo, exceptuando en algunos géneros, la utilización de técnicas de inteligencia artificial para el control de personajes jugables, originalmente controlados por un jugador real, son mínimas.

La gran diferencia entre controlar a un personaje jugable frente a un NPC es la abstracción de decisión en las decisiones a realizar conociendo el estado del juego, simulando así el proceso de razonamiento del humano que juega al videojuego, y que es capaz de tener una visión global acerca del estado en el que se encuentra la partida gracias a la información mostrada por pantalla. Es esta capacidad de los humanos las que nos permite aprender a jugar a juegos que no conocemos a priori, o que podemos estar jugando por primera vez.

La simulación de este comportamiento, la posibilidad de aprender a jugar a videojuegos de distintos géneros, objetivos y estilos, es el objetivo de esta práctica. Para conseguir este objetivo, haremos uso de distintas técnicas basadas en la computación evolutiva, sobre un framework que se apoya en la ejecución de agentes creados con distintas técnicas ante un grupo variado de videojuegos de distintos géneros, con distintos objetivos. Este framework se conoce como **GVG-AI** (*General Video Game AI Competition*), que nos permite crear controladores (también denominados agentes) que simulen el comportamiento de un jugador real que es capaz de aprender antes un set de juegos que no conoce.

Para la realización de la práctica, tenemos que tener en cuenta ciertas restricciones ante las que nos tenemos que adherir para conseguir un buen agente, como puede ser el **tiempo de ejecución** (nuestro agente debe escoger la acción a realizar en 40ms; si en 50ms no ha conseguido realizar una acción, se descalificará al controlador y la prueba terminará), o las **funciones de los controladores** (necesitan tener un constructor llamado *Agent*, y una función de elección de acciones llamado *act*). Estas limitaciones están descritas tanto en el enunciado de la práctica, como en la documentación del framework **GVG-AI**.
Dado que el framework contiene ciertos agentes de ejemplo, al igual que heurísticas de elección de acciones básicas, para la realización de la práctica hemos utilizado estos ejemplos, y hemos definido y creado **4 agentes** basados en los conocimientos encontrados en estos agentes y heurísticas, que nos ayudarán a conseguir un rendimiento algo mayor ante el problema en el que nos encontramos, y así conseguir un agente final que sea capaz de realizar esta simulación lo mejor posible.

En esta memoria, comentaremos la planificación que el grupo ha llevado a cabo para conseguir llegar al objetivo final definido; además del proceso de modelado, creación y ejecución de los agentes con los que hemos trabajado, y que se detallarán uno a uno en sus correspondientes apartados.
Por último, para ayudarnos a la hora de elegir uno para presentar a la competición entre grupos de trabajo descrita en el enunciado de la práctica, tendremos un apartado de pruebas, donde analizaremos el rendimiento de los agentes desarrollados en una situación real (donde puedan enfrentarse entre ellos) y decidiremos cuál es el mejor.

2. Planificación

Al comenzar la práctica, nos hemos encontrado con distintas fases bien diferenciadas, cada una con una importancia clara en relación al resto, y que hace que todas sean necesarias para la correcta evolución del trabajo. La división de la práctica en estas fases es clave para organizar óptimamente al equipo, y conseguir un desarrollo lógico durante la realización de la práctica. Estas fases se basan en un modelo básico de análisis del problema, resolución, análisis de las soluciones y un procedimiento iterativo de mejoría de los modelos hasta obtener un resultado destacable.
A continuación se indican las distintas fases que se han considerado para la resolución de la práctica:

- **Comprensión del código inicial:** estudio del framework **GVG-AI**, su funcionamiento y las restricciones que nos impone, tanto el código como el enunciado de la práctica.
- **Estudio de los ejemplos dados:** fase donde debemos entender las características, virtudes y problemas de los controladores y heurísticas dados como ejemplo dentro del código inicial, además de su rendimiento al ser usado en los juegos.
- **Planteamiento de sistemas evolutivos a utilizar:** estudio de todos los conocimientos obtenidos en las anteriores fases, y su utilización óptima con técnicas basadas en sistemas evolutivos.
  Con esta fase, conseguimos reducir 4 el número de posibles agentes a realizar: **tres algoritmos genéticos** y una **optimización por colonia de hormigas**.
- **Creación y evolución de los nuevos agentes:** en esta fase nos dividiremos después de realizar las tres fases anteriores juntos, y comenzaremos a desarrollar correctamente los 4 controladores descritos, en paralelo.
  - **Mejoría del algoritmo genético de ejemplo:** intentaremos mejorar el algoritmo genético que viene por defecto, sin cambiar su codificación.
  - **Algoritmo genético evolutivo:** crearemos un nuevo algoritmo genético, basado en la concatenación de acciones a realizar en su codificación.
  - **Politicas de reemplazo evolutivas:** crearemos un nuevo agente que contenga un algoritmo basado en politicas de reemplazo evolutivas.
  - **Optimización por colonias de hormigas:** crearemos un último agente basado en optimización por colonias de hormigas.
- **Pruebas de agentes:** es esta fase, ejecutaremos los distintos agentes terminados anteriormente, y decidiremos cual es el mejor de los 4.

Además de estas fases, hay que destacar que, para ayudar a la planificación, hemos hecho uso de herramientas que nos han permitido realizar cambios que podían ser fácilmente vistos por el resto de miembros del grupo; y que también nos han facilitado enormemente la organización y planificación de tareas.
Para planificarnos y dividir tareas hemos utilizado *Trello*, para el control de versiones del código hemos utilizado *Git*, y para la realización de esta memoria hemos utilizado *ShareLatex* (que permite que varios miembros realicen a la vez cambios en la memoria, en lenguaje de marcado *Latex*).

Tras esta planificación, en los siguientes apartados definiremos cuál es el rol de cada integrante dentro del equipo, y, por ende, que tareas realizará.

3. Roles designados a los componentes del equipo

Dada la complejidad de la práctica, los roles del equipo han ido variando constantemente durante la duración de esta, aunque es posible determinar concretamente el rol con el que más hemos identificado a cada integrante, describiendo sus capacidades de cara a la práctica.

Hay que destacar dos roles principales, con los que hemos estado trabajando, y en donde podemos, de una manera algo más genérica, colocar a cada miembro del grupo:

- **Desarrollador de agentes:** encargados de realizar trabajo en Java, modelando y programando los agentes a realizar. En este rol incluimos a **Daniel Garrido, Adrián Rodríguez, Rubén Rodríguez, Luis Buceta** y **Juan Poblete**.
- **Pruebas:** rol encargado de realizar la última fase del proyecto, deberá probar todos los algoritmos y decidir cuál es el mejor. En este rol tenemos a **Raúl López**.

Los roles de cada integrante fueron designados tras la tercera fase del trabajo, es decir, tras saber que íbamos a realizar. Una vez conocidos los roles, podemos definir las tareas de cada uno.

4. Relación de tareas realizadas por cada miembro del equipo

Las primeras tres fases (comprensión del código inicial, estudio de los ejemplos dados y el planteamiento de los sistemas evolutivos nuevos) fueron realizadas por todos los integrantes del equipo, y es en las siguientes dos fases donde dividimos la fuerza operativa. A continuación, se detallarán las tareas de cada miembro del equipo en estas últimas dos fases:

- **Raúl López:** pruebas de agentes.
- **Daniel Jerez:** creación de un nuevo algoritmo genético.
- **Adrían Rodríguez:** mejora del algoritmo genético de ejemplo, creación de un nuevo algoritmo basado en politicas de reemplazo evolutivas.
- **Rubén Rodríguez:** creación de un nuevo algoritmo basado en colonias de hormigas.
- **Juan Poblete:** creación de un nuevo algoritmo basado en colonias de hormigas.
- **Luis Buceta:** mejora del algoritmo genético de ejemplo.

5. Estudio de las heurísticas

Antes de comenzar a explicar el desarrollo de la práctica y de las técnicas utilizadas, creemos que es necesario definir las heurísticas que utilizaremos, puesto que son un elemento clave en el rendimiento de los agentes, ya que son las principales responsables de determinar cómo de buena puede ser realizar una acción ante una situación de juego concreta.

Para empezar, hay que destacar que tenemos unas ciertas **heurísticas de ejemplo**, que son utilizadas por los agentes de ejemplo, y que realizan una evaluación mínima del estado del juego. Tenemos dos heurísticas originales:

- **SimpleStateHeuristic:** estudia dos elementos del estado del juego: **las consecuencias de cara al final del juego de una acción** (es decir, si provoca la victoria o derrota de algún jugador) y **la distancia a ciertos elementos del mapa** (en concreto, los portales, que suelen ser metas, y a los NPCs que se encuentran en el mapa del juego).
  Esta heurística utiliza la **puntuación real del juego** (la que es controlada por la lógica del juego) junto con los **acontecimientos dados en relación a los dos elementos anteriores**, y determina una puntuación de evaluación (o *fitness*) de dicha acción sobre el estado de juego.
  La obtención de estas características se produce a través de la simulación del estado del juego en caso de que se de una acción (utilizando la función *advance()* del *framework*) sin que se de realmente, puesto que al final no es más que una mera simulación de lo que sucedería. **La mejor evaluación será la que ofrezca el mayor valor de la función heurística**. Este valor puede contribuir a mejorar la puntuación del agente, y por ende ganar el juego con la siguiente acción; o bien conseguir que la distancia con los objetos importantes del juego sea pequeña.

- **WinStateHeuristic:** más sencilla que SimpleState, utiliza la situación del final del juego para determinar un ganador. Es decir, sólo comprueba si al simular una acción, el juego terminaría ahí. En caso afirmativo, comprueba quién gana, si es el jugador controlado le ofrece una buena puntuación de evaluación, mientras que si pierde le ofrece una mala evaluación. Durante el resto de momentos, no devuelve una puntuación concreta, y espera que el funcionamiento del programa escoja una acción aleatoria hasta llegar a un movimiento final para determinar que realizar.

Como podemos ver, ambas heurísticas son bastante simples, y no son muy adecuadas para un alto número de juegos. SimpleState está más enfocada a juegos donde debemos llegar a un portal meta o interactuar con NPCs, mientras que WinState solo se preocupa del agente cuando va a ganar o perder, ignorando lo que haga el contrincante. Viendo esto, hemos decidido aportar funciones heurísticas adicionales, que puedan ser algo más detalladas y que ayuden al agente a realizar las acciones necesarias en cada momento. Entre ellas distinguimos:

- **Heuristica Sample:** evolución de **SimpleStateHeuristic**. Se basa en los mismos principios que esta última, utilizando la simulación de movimientos para estudiar el estado del juego, y, según las consecuencias del movimiento en el final del juego o la distancia a elementos del mapa, determinar una puntuación. La gran diferencia entre esta y la original es que hemos reducido el número de variables a calcular, junto con algunas pre-condiciones dadas en los *if* para que el tiempo de evaluación sea menor.
  Al ser menor, podemos aprovechar mejor los 40ms que tenemos para determinar una acción, y permitimos que el agente aprenda más y mejor en este tiempo. Con ello conseguimos mejores resultados a la hora de decidir acciones.

- **Heurística avara:** también basada en **SimpleState**; la amplía para hacerla más compleja. Tiene en cuenta características de **SimpleState**, pero también considera las variaciones en

la puntuación real del juego del jugador (si aumenta, mejor valor de la función heurística) y la distancia a nuevos elementos del mapa (en concreto, añadimos la posibilidad de calcular la distancia a los recursos necesarios para ganar el juego, como llaves, banderas, etc.).
A pesar de este aumento en complejidad, el tiempo de evaluación se mantiene, por lo que conseguimos realizar evaluaciones más complejas en el mismo tiempo que antes, consiguiendo mejores resultados.

- **Heurística de Dennis Soemers:** una evolución de **WinState**. De nuevo, mira a ver los estados finales del juego, y comprueba si el jugador llega a alguno de ellos. En caso afirmativo, comprueba el resultado del juego, y evalúa en relación a este (siempre buscando que el jugador realice acciones que le lleven a ganar). En este caso, contemplamos también casos como empates, o el final de los ticks de un juego, donde el resultado puede ser favorable al jugador controlado. La mejor evaluación es la que tenga mayor puntuación tras aplicar este estudio.

Tras la descripción de estas heurísticas, nos pondremos a describir el funcionamiento de los agentes creados, y ver como hemos utilizado estas heurísticas para hacerlos funcionar correctamente.

6. Descripción de la técnica(s) implementada(s)

6.1. Mejora del algoritmo genético de ejemplo

En este apartado, estudiaremos el trabajo realizado sobre el algoritmo original que viene dentro del código inicial de la práctica, denominado **sampleAG**. Este agente, originalmente, aplica un algoritmo genético y una heurística de las de por defecto, pero en el agente mejorado se ha cambiado la heurística por una más adecuada.

6.1.1. Algoritmo original

El algoritmo original, tal y como se detalla en el párrafo anterior, es el llamado **sampleAG**. Este agente aplica un algoritmo genético, soportado por la evaluación de una de las heurísticas que vienen poder defecto, y de manera iterativa consigue lograr que el agente reaccione a tiempo para no llegar al límite de los 40ms.
La codificación de la población se ve dentro del array *cuatridimensional genome*, donde sus dimensiones se ven detalladas por el **número de jugadores** (en el caso de esta práctica, dos), la **cantidad de acciones disponibles** en el juego actual, el **tamaño de la población** y la **profundidad de la simulación**.
La población se rellena con las acciones a realizar por cada personaje en cada momento de la simulación (de ahí tener la profundidad como dimensión del array).

El algoritmo, teniendo una población creada, realiza un proceso de elección de acción, que básicamente se fundamenta en la selección de los mejores elementos de la población, mediante un sistema de torneos, que comienza a evaluar a la población. Esta evaluación se realiza con la heurística **SimpleStateHeuristic** que viene por defecto, aunque es fácilmente cambiable por **WinStateHeuristic**. Sin embargo, no utiliza únicamente la heurística, sino que añade un punto de aleatoriedad a la evaluación utilizando un *random* que sube o baja el *fitness* obtenido, haciendo que la evaluación sea prácticamente aleatoria.

Al terminar las iteraciones de torneos necesarias para encontrar al mejor, no realiza ningún tipo de operación de cruce o mutación para consecuentes iteraciones, por lo que es posible que

la población pueda estancarse. Al saber cuál es el mejor, realiza la primera acción que indica el cromosoma, y se la envía al **GVG-AI** para que la interprete como un movimiento de un jugador.

6.1.2. Algoritmo mejorado

Como podemos ver, en el anterior algoritmo, hay un enorme grado de aleatoriedad que cancela cualquier tipo de decisión lógica que pueda realizar. El primer paso realizado al encontrar eso fue eliminar el susodicho *random*, con lo que empezamos a ver una pequeña mejoría en el rendimiento del algoritmo, tanto en tiempo como en elección de acciones a realizar.
Ya que **SimpleState** utiliza una heurística un poco más compleja que **WinState**, podemos ver cómo al utilizar **SimpleState** es capaz de mover al jugador en relación a los elementos del mapa con un poco más de situación que con el otro algoritmo.
WinState funciona mejor con situaciones donde no hay puntuación, y solo puedes encontrarte estados de fin de juego (un ejemplo de juego así es el *Tron*, donde debes evitar el impacto del agente con cualquier elemento del mapa.

Sin embargo, las deficiencias de ambas heurísticas son aparentes en juegos más complejos, donde por ejemplo es necesario almacenar recursos. Es por ello que decidimos crear las heurísticas más complejas vistas en el apartado 5: Estudio de las heurísticas, y probarlas con este mismo algoritmo (sin cambiar la codificación o método de selección), convirtiéndolo en un campo de pruebas. Teniendo las tres heurísticas (**Sample**, **Avara** y la de **Dennis Soemers**), probamos el rendimiento de esta configuración con cada una de ellas. Los resultados permitieron que se percibiera rápidamente una lógica en la elección de los movimientos del jugador controlado no vista con las heurísticas normales (de nuevo gracias a la ampliada complejidad de estas nuevas heurísticas sin sacrificar tiempo de procesamiento). En concreto, la **heurística avara**, que a priori es la más completa para todos los tipos de juegos, es la que mejor resultado da, como era de esperar. Siendo una heurística que se puede amoldar a muchos géneros y reglas de juego, da un buen resultado en prácticamente todos los juegos.
Al testear diversas partidas en las que este agente se enfrenta al algoritmo original, comprobamos que este nuevo algoritmo (con la heurística avara) gana al viejo una gran cantidad de ocasiones, demostrando que el cambio de la heurística es clave para el correcto funcionamiento de un algoritmo de este tipo.

6.2. Algoritmo genético evolutivo

Esta técnica fue implementada por **Daniel Jerez Garrido**. La idea principal detrás de esta implementación era lograr una concatenación de mejores acciones (profundidad en el avance de estados) para elegir la mejor acción dependiente de lo que suceda después de esta.

6.2.1. Parámetros

Los parámetros de los que dispone este diseño son los siguientes:

- **Tamaño de población:** número de individuos que trata el algoritmo en cada generación. El parámetro es variable, donde un valor mayor produce un coste computacional superior. Tras diversas pruebas de rendimiento se fijó este valor a 6 individuos.
- **Genes del cromosoma:** cada gen es una acción disponible por el juego. Los alelos del gen son los rangos de las acciones siendo por defecto en su mayoría numero enteros de 0 a 4. El tamaño del cromosoma (número de genes) es un parámetro variable que indica la profundidad de acciones para un estado origen. El valor por defecto asignado es 8.

- **Tasa de mutación:** probabilidad de que un gen al cruzarse mute. Nos permite meter diversidad genética para evitar una convergencia prematura de la población. No es un parámetro significativo pues no suele haber demasiadas generaciones en cada turno del jugador. Demasiada tasa de mutación genera un agente prácticamente aleatorio, se decidió fijar este parámetro a un 1 %.

- **Tamaño de torneo:** para la selección se decidió usar torneos. El tamaño de estos es un parámetro variable. Dado que el coste computacional depende considerablemente del tamaño de la población, el tamaño del torneo tiene que ser proporcional a esta, pues la evaluación va asociada al índice de comparaciones entre individuos. Se decidió fijar este valor a 2, siendo un torneo de pocas dimensiones, pero suficiente para una población tan pequeña.

- **Cruce uniforme:** para cada gen de la cadena del descendiente existe una probabilidad de que el gen pertenezca al padre, y otra de que pertenezca a la madre. Se decidió utilizar este tipo cruce porque en prácticas anteriores dio buenos resultados y es un cruce que mantiene la diversidad genética en sus hijos.

6.2.2. Funcionalidad

La explicación biológica que sigue esta técnica es selección, cruce y mutación. El constructor del agente se encarga de generar una población inicial, que como se comentó anteriormente esta compuesta de individuos cuyos cromosomas son una concatenación aleatoria de acciones dentro de los rangos disponibles por el juego seleccionado.

El algoritmo se ejecuta en la función *act*, que elige la acción a elegir en el turno. El número de generaciones que se ejecuta es variable respecto al tiempo de cómputo, dando de margen 12ms con respecto a los 40ms disponibles, de tal forma que no se produzcan descalificaciones del agente o acciones nulas.

La primera decisión es guardar el estado actual en el que se encuentra el agente. Posteriormente, dentro de una generación se realiza una selección con torneos (la evaluación del individuo se explica posteriormente) sobre la población actual generando una nueva población con los ganadores, de este conjunto se elige el individuo con mayor *fitness* cuya primera acción a ejecutar (primer gen del cromosoma) se almacena como posible candidato a mejor acción a elegir.
La población resultado se somete a un cruce uniforme donde un padre es el individuo **n** y el otro el **n + 1** (para esto es imprescindible que el número de población sea par) con su posterior mutación de los hijos resultado.
Los hijos pasan a ser la nueva población que pasará a la siguiente generación la cual también perdura por turnos. En cuanto finaliza el tiempo límite se devuelve la acción que mejor *fitness* tuvo con respecto a sus acciones sucesoras.

6.2.3. Evaluación

La evaluación de cada individuo se compone de dos funciones. Los torneos llaman a la función *evaluar* que consiste en una llamada iterativa por genes (acciones concatenadas) a la función *evaluación* (el estado se desplaza con la función *advance*), que da como resultado un *fitness* que se irá sumando, dando como resultado un *fitness* total. Este *fitness* da una ponderación del estado evaluado, y el total una ponderación del conjunto de estados evaluados.
Cuando un individuo se termina de evaluar, se recupera el estado original de partida para que el siguiente sea evaluado desde este (esto requiere un calculo computacional alto debido al uso de la función *copy*).

6.2.4. Diagrama de flujo

El diagrama que representa el flujo del algoritmo implementado es el siguiente.

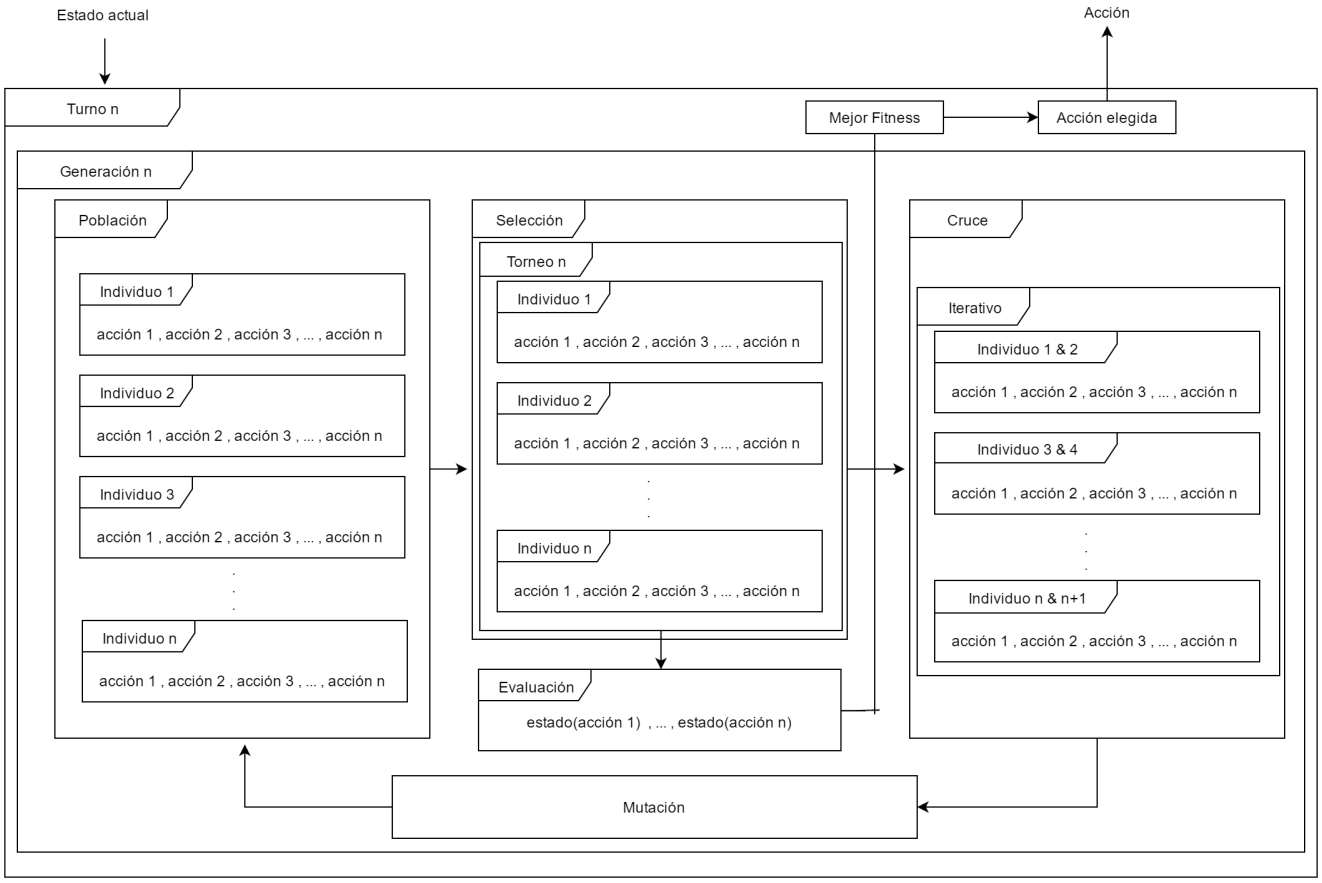

Figura 1: GA - Diagrama implementado

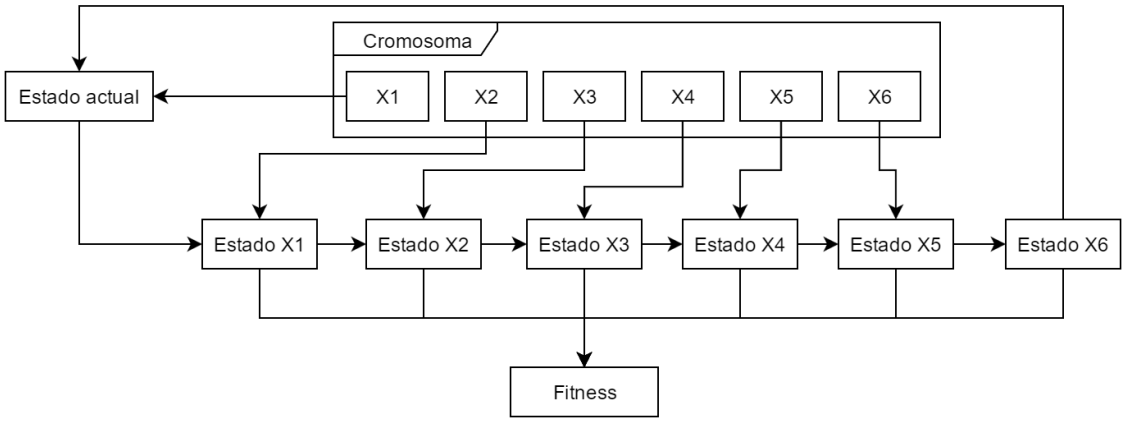

Figura 2: GA - Profundidad de estados

6.3. Optimización por Colonia de hormigas

Tras estudiar el programa nos encontramos con que existe una posibilidad de modelarlo utilizando el método de colonias de hormigas.

Este algoritmo nos aportaría un balance eficiente entre exploración y explotación. Gracias a la **exploración** proporcionada por las hormigas, evitaríamos caer en sub-óptimos locales utilizando la capacidad de mostrar varias partes del espacio de búsqueda. Por otro lado, un buen uso de la **explotación** nos será útil para obtener un posible óptimo global una vez que consideramos que nos hemos aproximado suficientemente al mismo.

6.3.1. Modelado

Diseñamos un grafo en el cuál los **nodos** representan **estados del juego** y los **enlaces** entre los mismos interpretan las **acciones posibles a realizar**. De esta manera, cada hormiga debe escoger una acción a realizar en cada iteración del algoritmo, llevándole ésta a un nuevo estado del juego.

Para esclarecer la idea que subyace en este diseño, suponiendo que existieran tres acciones posibles para el jugador y que la profundidad de predicción de nuestro modelo fuera de 3 turnos (longitud del camino de las hormigas), el grafo representativo sería el siguiente:

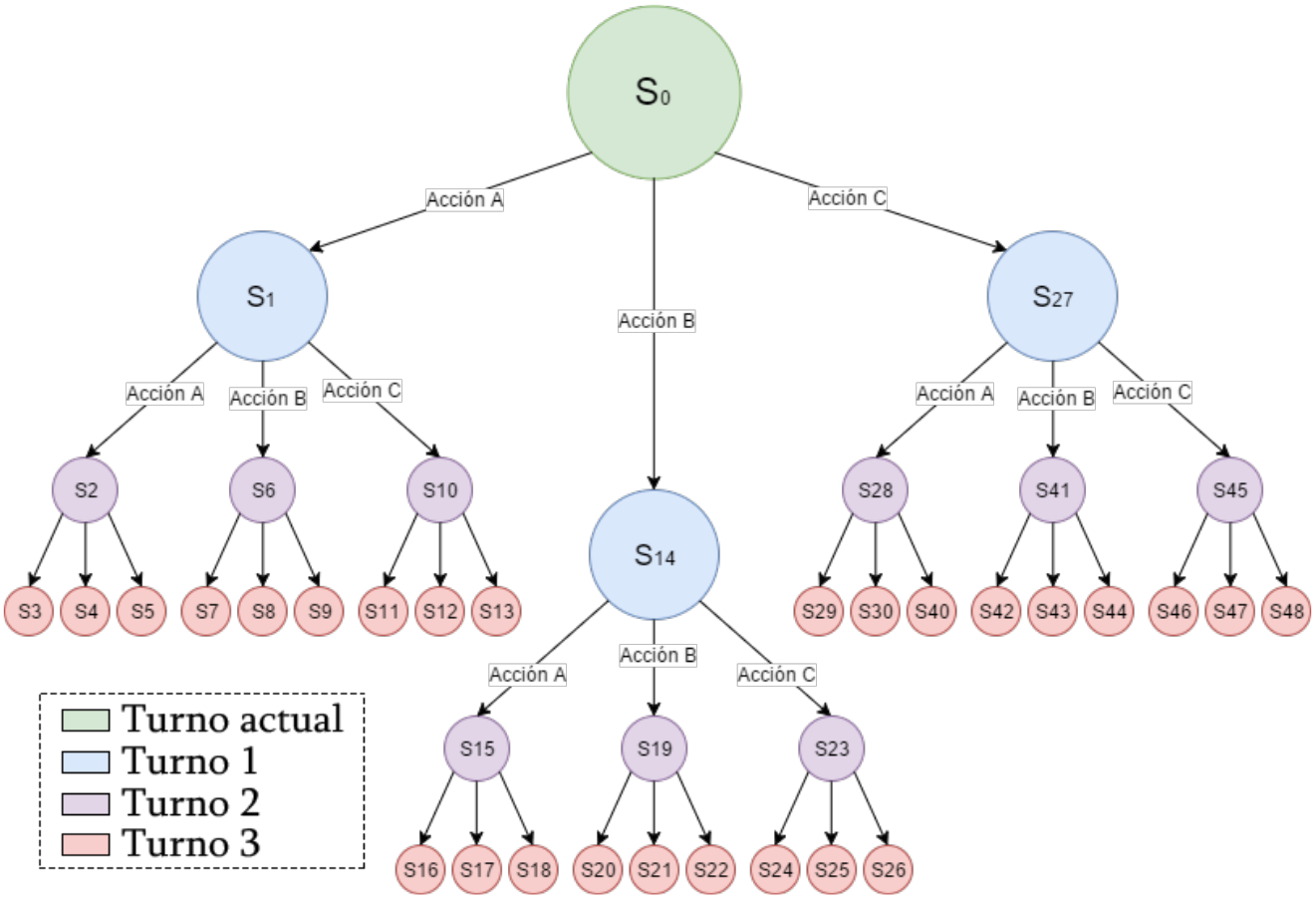

Figura 3: ACO - Grafo modelado

El aprendizaje que se produce durante la ejecución del algoritmo consiste en la cantidad de feromona depositada en los estados más avanzados respecto al nodo que representa el estado actual. Evidentemente, este grafo es distinto en cada iteración del juego, ya que debe representar

las opciones que tienen las hormigas de avanzar el estado actual a través de diversas simulaciones. Cuando comienza un nuevo turno, los tramos que acercan a estados futuros ya poseen feromona generada por la ejecución del algoritmo en iteraciones anteriores.
Es importante señalar que cuando se realiza una primera simulación del estado actual, con cualquier acción, se llega a un estado del enemigo. Por lo que para tener un mínimo de información sobre qué camino tomar (gracias a la iteración del algoritmo de Optimización por Colonia de Hormigas) es necesario llegar al menos a tres niveles de profundidad en la simulación.

Los parámetros que podemos utilizar para este algoritmo se pueden resumir en:

- **Número de hormigas**: indica el número de hormigas que se lanzan en la ejecución del algoritmo de Optimización por Colonia de Hormigas.
- **Profundidad**: indica el número de nodos a alcanzar por cada hormiga, que se traduce en profundidad de simulación.
- **Grado de importancia de la feromona**: peso que se le da a la feromona para la toma de decisiones de las hormigas.
- **Grado de importancia de la evaluación**: peso que se le da al valor de evaluación de cada estado en la toma de decisiones de las hormigas.

6.3.2. Tablas de decisión

En todo modelo basado en optimización por colonias de hormigas debe existir una tabla de decisiones que será evaluada por cada hormiga antes de desplazarse hacia un nuevo nodo del grafo. En nuestro caso, esta tabla se define en base a dos parámetros:

- **Evaluación:** asigna un valor de calidad a un estado de juego en base a una función heurística. La ideada para este caso se basa en la puntuación del jugador y en el hecho de si alguien gana la partida. Cuanto mayor sea el valor de evaluación, mayor será la probabilidad de que la hormiga escoja este camino. Sin embargo, se propone la experimentación con diversas funciones heurísticas que midan la calidad de un estado para el agente.
- **Feromonas:** cada hormiga emite una cantidad de feromona a medida que camina. Cuanto más intenso sea el rastro de feromonas, mayor será la probabilidad de que sea seguido.

En nuestro diseño damos la opción de darle un peso diferente a estos dos parámetros, es decir, existe la posibilidad de variar la importancia que se le da a cada uno de ellos a la hora de crear la tabla de decisión.

Con el fin de ampliar nuestro grado de exploración, damos un pequeño factor de aleatoriedad a las decisiones tomadas por las hormigas, de forma que cada vez que deben desplazarse a otro grafo traducen la tabla de decisión del mismo en una tabla de probabilidades de utilizar cada uno de estos caminos.

Para optimizar el tiempo de cómputo, en nuestro modelo sólo se calculan las tablas cuando una de las hormigas necesita utilizarla para desplazarse. El propósito de esto es ahorrar cálculos innecesarios sabiendo que, si fueran evaluadas las del grafo completo en cada iteración del algoritmo, es muy probable que muchas no fueran utilizadas por ninguna hormiga.

6.3.3. Implementación

Para lograr una implementación eficiente dentro de la plataforma, se ha optado por un diseño basado en el uso de *arrays*, estructuras muy simples y que presentan una gran eficacia en cuanto al acceso de lectura/escritura de su información.
El *array* tendría un tamaño proporcional al número de estados alcanzables por el agente en el estado actual del juego. Dicha proporción sigue la regla de que cada estado alcanzable necesita de 5 posiciones en el *array*. A continuación se indica la información que se almacena en cada una de dichas posiciones para cada estado:

- **Estado**: representación del estado del juego. Para evitar cargar el *array* con información innecesaria, se ha optado por la creación de una clase auxiliar que nos permita recoger las características que representan a un estado, que son:
  - Acción del jugador que permitió alcanzar el estado.
  - Última acción ejecutada por el rival.
  - Vida actual del jugador.
  - Puntuación actual del jugador.
  - Jugador ganador de ese estado en el caso de que lo hubiera.
- **Nodos hijos hacia los que se puede transitar**: *array* que contiene el set de estados alcanzables por el estado correspondiente.
- **Feromonas**: cantidad de feromonas que posee el estado.
- **Evaluación**: valor de la función heurística de evaluación para el estado.
- **Tabla de probabilidades**: no existe como tal una estructura que represente la tabla de probabilidades de cada hormiga. Esto se simula durante el funcionamiento del algoritmo. Para sustentar esta funcionalidad, en su lugar, este campo posee un valor que indica la importancia que tiene dicho estado para todas las hormigas, en función de la feromona y la evaluación del estado. Como se ha indicado previamente, la combinación de ambos factores se basa en la aplicación de un peso para uno de dichos factores, permitiendo realizar experimentos en los que se opta por la explotación o por la exploración.

Como se ha mencionado anteriormente, el grafo se va expandiendo en base a los caminos que van tomando las hormigas. Si una hormigas está en un nodo, esta debe poder observar la importancia que tienen todos los estados alcanzables para ella en dicho estado. Dado que utilizamos *arrays*, lo que sucede cuando un estado del array es visitado; es decir, se le deposita feromona, es que se crea una referencia a un *array* con las mismas características que el que posee la posición desde la que este nuevo *array* es apuntado.

Después de lanzar las hormigas para visitar los distintos estados, se realiza la elección de la acción que acerque al estado más lejano alcanzado, con mayor valor de evaluación.

Para mejorar la eficacia de nuestra implementación, se ha diseñado un método de poda del grafo, de tal manera que aquellos nodos que no han sido alcanzados (y a los que ya no se puede llegar, dado que es un grafo dirigido), son eliminados del propio grafo.
Para simular este proceso, cuando comienza un turno, el primer paso es localizar el estado al que se ha llegado finalmente; recordemos que aunque el algoritmo de Optimización por Colonias de Hormigas le indica al agente qué acción hacer, el enemigo puede hacer una acción que desconocemos a priori. Este estado se localiza a través de la clase auxiliar que consiste en una representación del estado del juego (con las características indicadas anteriormente).

Realizando una comparativa de las características del estado actual con las de los conjuntos de estados del *array*, podemos identificar el estado en el que nos encontramos. Este estado pasa a ser el primer nodo de nuestro grafo, de tal manera que nuestro *array* inicial ahora apunta al *array* seleccionado como equivalente del estado actual.

6.4. Problemas y codificación

Pese al diseño realizado en el presente documento, no se ha llegado a una versión funcional de un agente que utilice el algoritmo propuesto. Los motivos de este hecho se escapan de los objetivos de esta memoria, pero se fundamentan en la complejidad de ciertos aspectos del algoritmo (mantenimiento del grafo, simulación en base a las necesidades de las hormigas) en relación con el tiempo con el que se ha dispuesto para realizarlo.
Sin embargo, se pretende dar un par de comentarios en cuanto a las dificultades que tendría la ejecución del algoritmo, que por desgracia no ha podido ser experimentado. El principal problema con el que tiene que lidiar este algoritmo en este problema es con el tiempo de cómputo permitido para la elección de la acción a ejecutar. Dado que una hormiga puedellegar a viajar por varios estados, es necesario hacer una simulación de la ejecución de distintas acciones para el mismo estado. Esto provoca que cada vez que se quiera realizar una simulación con un alternativa distinta, haya que salvaguardar el estado desde el que se parte. Esto sólo es posible a través de la función *copy* de la clase *StateObservationMulti*. Esta es una de las operaciones más costosas que se pueden realizar durante la ejecución del programa actual. Por lo que desconocemos si no se podría llegar ni si quiera a un mínimo de aprendizaje dado el número de copias de estados que serían necesarias.

6.5. Políticas de reemplazo evolutivas

Esta técnica fue desarrollada por **Adrián Rodríguez Grillo** y consistió en usar la política $(\mu + \lambda) - EE$ de estrategias evolutivas para controlar dos tipos de agentes automáticos, uno con **búsqueda aleatoria** y, otro, con un **árbol de exploración**.

6.5.1. Funcionamiento general

El funcionamiento del algoritmo basado las políticas de reemplazo estudiadas en estrategia evolutiva es muy similar para ambos agentes, con pequeñas diferencias en las estructuras utilizadas para cada implementación, que detallaremos más adelante. Comenzaremos explicando los **parámetros** que pueden ser modificados en el juego:

- **Tamaño de la población (*tamPoblacion*):** es el tamaño de la población que participará en el algoritmo, es decir, el número de individuos, que serán el conjunto de movimientos a realizar desde la posición actual.
- μ **- Mu (*muSize*):** es el número de individuos que pasarán de una generación a la siguiente, es decir, del conjunto de individuos de progenitores, los μ individuos con mejor *fitness* pasarán a la siguiente generación.
- λ **- Lambda (*lamSize*):** es el número de individuos de una generación que son sustituidos por los descendientes de dicha generación, es decir, los λ peores individuos serán sustituidos directamente por los descendientes de dicha generación.
- **Profundidad de exploración (*profundidad*):** será el número de generaciones que se desarrollará el algoritmo, añadiéndose por cada generación una acción.

La inspiración biológica que sigue este método es únicamente la reproducción y la sustitución, donde un individuo, que será la situación actual en la que se encuentra el juego, se "reproduce"produciendo como sucesores los estados a los que puede llegar realizando una acción de las disponibles.

Este algoritmo se ejecutará en cada turno del jugador desde el comienzo, es decir, en cada turno se iniciará la población con la acción actual en la que se encuentra el jugador, y devolverá la acción a realizar tras realizar las evaluaciones. El motivo por el cuál no se guarda el conjunto de acciones exploradas es que la técnica de búsqueda no tiene en cuenta los movimientos de otros jugadores y cambios en la partida. El funcionamiento de la técnica será el siguiente:

1. Se crea la **población inicial** con un conjunto de individuos iguales que contendrán la situación actual de la partida.
2. Se realiza la primera generación donde cada individuo se reproduce, obteniendo un único hijo, que es el siguiente estado de la partida tras realizar una acción, cuya selección dependerá del método utilizado (al azar o con árboles de decisión).
3. Se **evalúa** cada individuo.
4. La población es ordenada en relación a su *fitness* colocándose primero el individuo con mejor valoración y, así, sucesivamente.
5. Se **reproducen** los mejores individuos generando λ sucesores.
6. Los λ peores individuos de la generación son **sustituidos** por los sucesores.

7. Se vuelve al **paso 3** hasta que se alcanza el límite de tiempo o de profundidad.
8. Se devuelve la **primera acción que tiene el mejor individuo resultante del proceso**, que será la que realice el agente.

La ejecución de este proceso se realizará íntegramente en el método *act*, aunque la inicialización de diferentes variables de control se realizará en el método de creación del agente (constructor). Estas variables de control serán el conocimiento de las acciones disponibles y del *id* del jugador.

Con respecto a la reproducción (aunque depende de los parámetros y del agente escogido) los hijos pertenecen mayoritariamente o únicamente del individuo con más *fitness*. Esto, aunque puede reducir la diversidad de la población, lo hemos decidido así debido a la cantidad de movimientos disponibles, con los que generamos un sucesor por cada uno.

Es decir, ya que en la mayoría de las situaciones λ es menor que el rango de acciones disponibles en el juego, decidimos crear la población de sucesores con individuos que provengan del mejor de la generación anterior.

La parte clave de este método es la función de evaluación de los individuos, que será la que nos indique el conjunto de acciones a explorar para obtener el mejor resultado. Para estos agentes, aunque es compatible con las diferentes heurísticas explicadas anteriormente, hemos decidido utilizar la **avara** ya que recoge más información que las demás.

Este detalle es muy importante para este método, ya que la evaluación sólo se realiza sobre el nuevo estado y, por tanto, si la información obtenida de los estados es nula la evolución pasaría a ser puro azar, ya que no podrían ordenarse los individuos en base a su calidad.

Por último, el proceso de sustitución dependerá más de cada agente elegido, es decir, el *random* tiende a contar más con el presente, y el basado en *árboles de decisión* también tiene en cuenta los pasos realizados anteriormente.

6.5.2. Agente random

Este agente se denomina así debido a que la selección de los sucesores de los individuos se hace de forma aleatoria, es decir, la acción que genera el sucesor se selecciona de forma aleatoria de entre las disponibles. Con respecto al agente basado en árboles, aunque el proceso de generación de sucesores es más rápido en este caso; ya que no hay que controlar a los nuevos individuos, este puede producir individuos iguales que reducen la efectividad de la búsqueda al repetirse evaluaciones.

Como hemos comentado anteriormente, este agente se enfoca más en el presente, es decir, no tiene cuenta las acciones pasadas para calcular el *fitness*. Esto lo hemos decidido así para que el proceso pueda volver a momentos anteriores en el caso de que las acciones elegidas para generar los sucesores no hayan sido buenas. Es decir, gracias a esta decisión, si los hijos de un individuo no son mejores que los ya existentes, estos no serán elegidos para las siguientes generaciones.

Este método depende de dos estructuras de datos para su corrector funcionamiento:

Individuo_Antecesor: esta estructura será el individuo de la población, formado por dos elementos:

- **Estado:** que será el estado de la partida que representa el individuo, es decir, en la primera generación el estado será el estado de la partida actual y, al reproducirse el individuo, este se irá actualizando.

- **Antecesor:** Que será la acción que realizará el agente si el individuo resulta el más apto de la población al terminar el proceso.

puntuacionIndividuo: que será la estructura donde se guarde el *fitness* de un individuo y que permita ordenarlos en cuanto al valor de este. Esta formado por dos elementos:

- **Puntuación:** que será el resultado de la evaluación del individuo.
- **Índice del individuo:** posición del individuo en la población, que es una lista.

El funcionamiento del algoritmo será el indicado en el apartado anterior (6.5.1) con los detalles explicados anteriormente, es decir, con la selección de sucesores y con el cálculo de la función de evaluación comentados.

6.5.3. Agente basado en árboles de decisión

Este agente se denomina así ya que los sucesores del individuo se obtienen teniendo en cuenta todas las acciones que el individuo es capaz de realizar. A diferencia con el agente *random*, los sucesores de un individuo nunca pueden ser iguales porque son controlados para que no se repitan. Esto es bueno ya que no se repiten evaluaciones de individuos con el mismo estado, sin embargo, los mecanismos de control hacen que se pierda tiempo de exploración.

Por tanto, para obtener los sucesores de un individuo se toman las λ primeras acciones que se encuentren en la lista de posibles acciones. Esto planteaba dos problemas, el primero es que si la heurística no devolvía información de los estados (todos los individuos con el mismo *fitness*) siempre se elegía la misma acción para crear los individuos, haciendo que se devolviese siempre esa acción y el jugador se quedase bloqueado haciendo siempre lo mismo. Para solucionar esto, para cada nueva generación, la lista de acciones posibles es modificada para tener diferente orden y, así, que no siempre salga la misma como primera acción.

El segundo problema dependía del tamaño de λ, cuando este era mayor al número de acciones disponibles, donde se empezarían a repetir individuos si se tomase únicamente el mejor de la población. Para solucionar este problema, cada vez que se generen todos los posibles sucesores de un individuo, se pasaría al siguiente en *fitness* para producir sus sucesores, haciendo que la diversidad de los sucesores aumente.

Otra de las diferencias con respecto al agente anterior dependía de la importancia que se daba a las acciones pasadas, es decir, la influencia del conjunto de acciones. Si bien en el caso del agente *random* no se tenía en cuenta el pasado, en este, se va almacenando el *fitness* de generación en generación, es decir, los hijos adquieren en la reproducción el resultado de la función de evaluación de sus padres y añaden el suyo. Esto hace que sea más difícil retroceder en el árbol si la solución escogida acaba resultando mala, pero por otro lado, es más agresiva al navegar por el conjunto de nodos con más *fitness*, cosa útil debido al escaso tiempo de procesamiento que disponemos.

Este agente depende de dos elementos para su correcto funcionamiento; por un lado, la lista de posibles acciones disponibles en el juego, que es utilizada para obtener a los hijos y, por otro lado, de **Individuo_Antecesor** que será el objeto que represente a los individuos de la población. Este será similar al que existe en el agente *random* pero contendrá más elementos, que serán los siguientes:

- **Estado:** al igual que en el anterior, será el estado de la partida que representa el individuo.

- **Antecesor:** que será la acción que realizará el agente si el individuo resulta el más apto de la población al terminar el proceso.
- **Puntuación:** que será el *fitness* del individuo, que se transmitirá de generación en generación y que permite ordenar a los individuos en la población en base a su valor.

Su funcionamiento, como se ha indicado anteriormente, será el explicado en el punto 6.5.1 con las características explicadas en este apartado.

7. Pruebas

Para explicar las pruebas debidamente, se aporta un listado de los agentes desarrollados que han sido subidos a la plataforma, con el nombre de usuario y el que tiene originalmente en el proyecto desarrollado por el equipo. También se añade una pequeña descripción para indicar a qué algoritmo se corresponde.

- **darwinner:** Algoritmo genético desarrollado por *Daniel Jerez Garrido*.
  Nombre en el proyecto: *darwinnerGeneticNew*.
- **darwinnerTree:** Algoritmo basado en Árboles de Decisión con política de reemplazo $(\mu + \lambda) - EE$ y heurística avara por **Adrián Rodríguez Grillo**.
  Nombre en el proyecto: *darwinnerTree*
- **darwinnerSample:** Algoritmo basado en acciones aleatorias con política de reemplazo $(\mu + \lambda) - EE$ y heurística sample por **Adrián Rodríguez Grillo**.
  Nombre en el proyecto: *darwinnerRandom* (Utilizando la clase *HeuristicaSample*).
- **rurodrig:** Algoritmo basado en acciones aleatorias con política de reemplazo $(\mu + \lambda) - EE$ y heurística avara por **Adrián Rodríguez Grillo**.
  Nombre en el proyecto: *darwinnerRandom* (Utilizando la clase *HeuristicaAvara*).
- **buislu:** Agente basado en el algoritmo genético de ejemplo utilizando la heurística avara, por *Luis Buceta Ojeda*
  Nombre en el proyecto: *darwinnerSample* (Utilizando la clase *HeuristicaAvara*).

Para realizar una comparativa entre los distintos agentes desarrollados en esta práctica, se ha optado por observar los resultados que nos ofrece la plataforma *The GVG-AI Competition*. Dado que queremos decidir cuál es el mejor agente para la resolución del problema, se han recopilado los resultados de partidas entre nuestros propios agentes.

Es importante señalar que en el momento en el que los datos han sido recopilados aún no se habían producido todos los enfrentamientos entre nuestros agentes. Este motivo es el que provoca que algunos agentes como *buislu* tengan un conjunto de enfrentamiento poco variado (poca cantidad de agentes distintos en el mismo set de enfrentamientos).

A continuación se muestran diversas tablas resumen de los enfrentamientos recopilados.

- **Tabla de victorias:**
  muestra el número de victorias de cada agente (fila) frente a otro (columna).

Figura 4: Tabla - victorias

Victorias	darwinner	darwinnerTree	rurodrig	darwinerSample	buislu
darwinner	-	63	7	19	0
darwinnerTree	71	-	20	24	0
rurodrig	7	20	-	43	0
darwinerSample	16	21	38	-	9
buislu	0	0	0	7	-

- **Tabla detotales:**
  muestra el número total de partidas de cada agente (fila) frente a otro (columna).

Figura 5: Tabla - total partidas

Total partidas	darwinner	darwinnerTree	rurodrig	darwinerSample	buislu
darwinner	-	172	20	40	0
darwinnerTree	172	-	58	60	0
rurodrig	20	58	-	118	0
darwinerSample	40	60	118	-	20
buislu	0	0	0	20	-

- **Tabla de tasas de victoria:**
  muestra el porcentaje de victoria de cada agente (fila) frente a otro (columna).

Figura 6: Tabla - tasa de victoria

Tasa victoria	darwinner	darwinnerTree	rurodrig	darwinerSample	buislu
darwinner	-	0,3662790698	0,35	0,475	0
darwinnerTree	0,4127906977	-	0,3448275862	0,4	0
rurodrig	0,35	0,3448275862	-	0,3644067797	0
darwinerSample	0,4	0,35	0,3220338983	-	0,45
buislu	0	0	0	0,35	-

Para colaborar con la comprensión de estos resúmenes de enfrentamientos, aportamos una serie de gráficas resumen.
En la esta gráfica se observa la acumulación de tasas de victorias de cada agente, frente al resto. Se puede observar cómo claramente existe un desequilibrio entre el número de enfrentamientos que ha habido entre los distintos agentes.

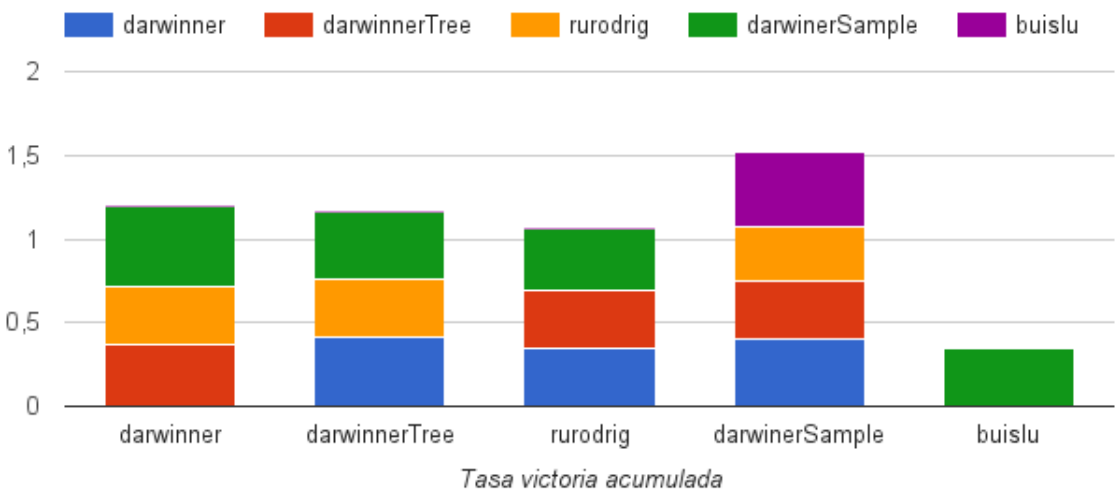

Figura 7: Gráfica - tasa de victoria acumulada

Esta gráfica nos permite visualizar mejor cuál de los agentes gana en más ocasiones al resto de agentes.
Darwinner es el agente que gana en más ocasiones a un agente (*darwinnerSample*), y el resto de enfrentamientos presenta valores muy similares a los porcentajes de victorias que presentan otros agentes.
No podemos justificar que el mejor agente de todos sin duda sea *darwinner*, ya que presenta unos resultados muy similares a los del resto de agentes, pero sí que es ligeramente mejor. Por ello decidimos escoger este como **mejor modelo generado** para la resolución del problema.

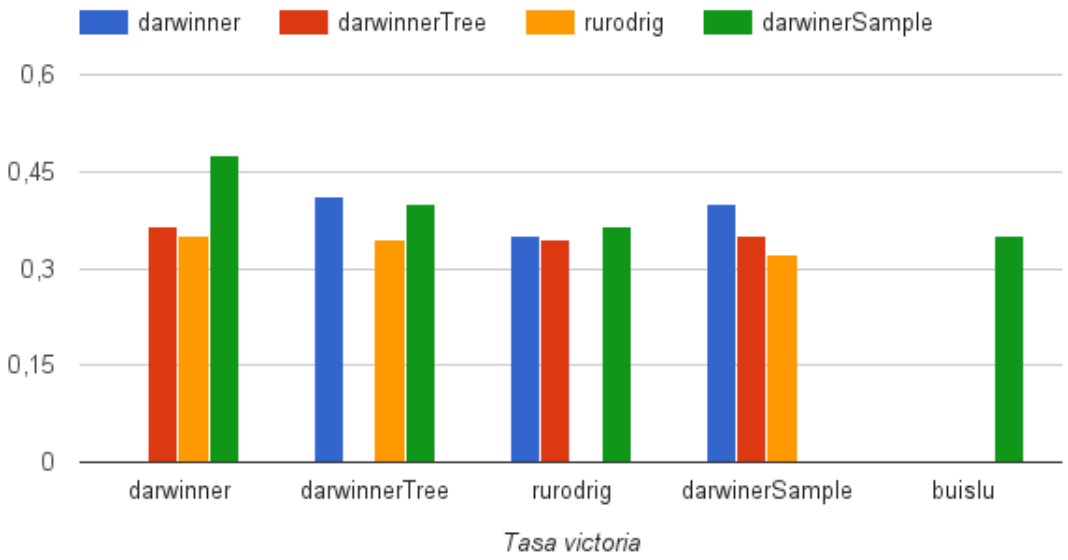

Figura 8: Gráfica - tasa de victoria

8. Conclusiones

8.1. Problemas encontrados

El principal que hemos encontrado ha sido la falta de documentación del framework, primeramente al tener que estudiar por nuestra parte el funcionamiento de los algoritmos y, por otro lado, al desarrollar las heurísticas, no existe ningún lugar donde encontrar información y significado de los diferentes objetos involucrados en el juego, como el significado de los atributos de los NPCs.

Otro de los problemas encontrados ha sido las excepciones que saltaban al ejecutar pruebas de manera automática como la de round robin (todos contra todos) para recoger toda la información en un .txt.

8.2. Comentarios personales

Ha resultado muy interesante el desarrollo de la práctica debido a los conocimientos que nos ha proporcionado.

Todos los algoritmos utilizados fueron estudiados de forma teórica durante las clases de la asignatura. Sin embargo, el aspecto práctico fue brevemente desarrollado a través de una serie de ejercicios. Es mediante la realización de estos agentes donde hemos aumentado nuestra capacidad de utilizar estos algoritmos en problemas reales como es por ejemplo un controlador de videojuegos. Además, por primera vez nos hemos enfrentado a la necesidad de implementar estos algoritmos en un lenguaje de programación.

9. Anexo I: comentarios acerca del trabajo en equipo

A continuación se aporta un breve resumen de comentarios que los integrantes del grupo han considerado necesarios, sobre el trabajo realizado en equipo:

- Dadas las condiciones de la práctica y pese a las dimensiones de la misma, se ha podido abordar su resolución por distintas vías, con la esperanza de encontrar buenos resultados en alguna de ellas. Esto ha a beneficiado el trabajo global, ya que cada alumno ha podido trabajar en mayor o menor medida en su propia vía sin la preocupación de influir en la tarea realizada por el resto.

- Haber colaborado en un grupo tan grande ha permitido que se pudiera intentar desarrollar algoritmos como la Optimización por Colonia de Hormigas. Esto suponía una apuesta arriesgada, no tanto por su diseño pero sí por su implementación. Sin embargo, gracias al reparto de tareas entre los miembros del equipo se han conseguido agentes funcionales viables a pesar de no haber llegado a una versión funcional de la Optimización por Colonia de Hormigas.

Referencias

[1] The General Video Game AI Competition - 2016.

http://www.gvgai.net/index.php

[2] Sample Genetic Algorithm Controller (2 Player).

https://github.com/EssexUniversityMCTS/gvgai/wiki/Sample-Genetic-\
Algorithm-Controller-(2-Player)

[3] Github: repositorio de GIT utilizado para el control de versiones.

https://github.com/

[4] Trello: plataforma web para la gestión y planificación del equipo, basada en modelo Canvas.

https://trello.com/

[5] ShareLatex: plataforma web de desarrollo colaborativo de documentos en Latex.

https://www.sharelatex.com/

[6] Heurísticas de inspiración, basadas en los agentes de Wells Lucas Santo.

https://github.com/Devking/GVGAI-Controllers

[7] Heuristica basada en el agente MaastCTS2, de Dennis Soemers.

https://github.com/DennisSoemers/MaastCTS2/tree/master/Two-Player/src/
MaastCTS2/heuristics/states

https://dennissoemers.github.io/jekyll/update/2016/09/29/
the-general-video-game-agent-maastcts2.html