ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería del Software

Hibridación de bots 'humanizados' de Unreal Tournament 2004 mediante técnicas de Inteligencia Computacional.

Hybridization of 'humanized' bots from Unreal Tournament 2004 using Computational Intelligence techniques.

Realizado por
Álvaro Gutiérrez Rodríguez
Tutorizado por
Antonio José Fernández Leiva y Antonio M. Mora García
Departamento
Departamento de Lenguajes y Ciencias de la Computación.

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Junio de 2017

Fecha defensa:
El Secretario del Tribunal

Resumen: Este proyecto presenta múltiples hibridaciones de dos de los mejores bots de la competición Botprize de la edición de 2014, competición que buscaba el bot con el mejor comportamiento humano jugando al famoso videojuego del género First Person Player, Unreal Tournament 2004. Para ello se evaluaban a los participantes mediante un Test de Turing aplicado a videojuegos. El trabajo considera los códigos de MirrorBot (el ganador) y NizorBot (el segundo) y los combina en dos enfoques diferentes, con el objetivo de obtener un bot que pueda mostrar el mejor comportamiento humano. También se ha realizado una versión evolutiva de Mirrorbot, el cuál fue optimizado mediante un algoritmo genético. Tanto los bot originales como las hibridaciones creadas, han sido evaluadas mediante un Test de Turing aplicado a videojuegos cuyos resultados muestran que la versión evolutiva de Mirrorbot ha superado a la versión original en cuanto a tener un comportamiento humano. Además, una de las hibridaciones ha obtenido un nivel de humanidad bastante alto.

Palabras claves: Inteligencia artificial (IA), inteligencia computacional (IC), computación evolutiva, algoritmo genético, test de Turing, test de Turing aplicado a videojuegos.

Abstract: This project presents multiple hybridizations of the two best bots on the BotPrize 2014 competition, which sought for the best humanlike bot playing the First Person Shooter game Unreal Tournament 2004. To this aim the participants were evaluated using a Turing test in the game. The work considers MirrorBot (the winner) and NizorBot (the second) codes and combines them in two different approaches, aiming to obtain a bot able to show the best behaviour overall. There is also an evolutionary version on MirrorBot, which has been optimized by means of a Genetic Algorithm. The new and the original bots have been tested in a new, open, and public Turing test whose results show that the evolutionary version of MirrorBot apparently improves the original bot, and also that one of the novel approaches gets a good humanness level.

Keywords: Artificial Inteligent (AI), computational inteligent (CI), evolutive computation, genetic algorithm, Turing test, Turing test applied to videogames.

Índice

1. Introducción

Este proyecto se inició con la idea de mejorar la experiencia vivida por los jugadores de videojuegos. La industria del videojuego es una industria que está creciendo año tras año, los consumidores de videojuegos esperan tener experiencias cada vez más realistas y para ello se desarrollan técnicas de IA cada vez más complejas. Al comienzo, esta industria estaba bastante limitada por la tecnología del momento (a nivel de software y hardware) y la jugabilidad y experiencia de juego no requerían de una IA muy avanzada. Pero a medida que la tecnología ha ido avanzando, esta jugabilidad y experiencia ha ido evolucionando con ella y ha llegado un momento en el que se necesita de una IA más avanzada para poder alcanzar esa jugabilidad y experiencia de juego.

Basándonos en la definición de Inteligencia Artificial (IA) que realizan Peter Norvig y Stuart Russell en su libro *Artificial Intelligence: A Modern Approach* [Russell, Stuart J. y Norvig, Peter, 2003], la IA incluye el diseño de agentes capaces de percibir su entorno, procesar los datos percibidos y llevar a cabo acciones de forma autónoma a partir de dichas percepciones. Existen dos formas de reconocer cuándo una acción es inteligente: lo es cuando la lleva a cabo un humano o lo es cuando es racional. Estos dos conceptos son diferentes ya que, normalmente, lo racional lleva a encontrar la solución óptima mientras que el comportamiento humano no siempre es óptimo.

La Inteligencia computacional [Inteligencia computacional, 2016] es una rama de la inteligencia artificial centrada en el estudio de mecanismos adaptativos para permitir el comportamiento inteligente de sistemas complejos y cambiantes, tratando de no confiar en algoritmos heurísticos tan habituales en la IA más tradicional. Dentro de la IC podemos encontrar ramas de la inteligencia computacional como las Redes Neuronales, Computación Evolutiva, Sistemas Inmunes Artificiales o Sistemas difusos. En este proyecto se ha escogido la rama de la computación evolutiva y más concretamente los algoritmos genéticos.

Los algoritmos genéticos [Algoritmos genéticos, 2016] pertenecen a la computación evolutiva y son métodos de optimización y búsqueda de soluciones basados en la evolución genética. De igual modo que la evolución genética, los algoritmos genéticos mantienen un conjunto de entidades que representan posibles soluciones a un problema y éstas posibles soluciones se mezclarán y competirán entre sí de tal manera que las mejores soluciones o las más aptas prevalecerán a lo largo del tiempo, evolucionando hacia mejores soluciones al problema que se enfrenta.

El test de Turing [Russell, Stuart J. y Norvig, Peter, 2003] es aquel que planteó Alan Turing en 1950 para proporcionar una definición estándar de inteligencia. Esta definición no iba a contener una lista de cualidades que debiera tener un agente, sino que el agente fuera capaz de actuar de manera que fuera imposible determinar si es o no es humano. La prueba consiste en una conversación entre un humano y un computador (mediante chat por computadora) y un juez humano quién va a visualizar la conversación. Este juez, debe determinar quién es el humano y quién no. Si la

computadora actúa de manera que el juez no puede dar una respuesta, la computadora ha pasado la prueba.

Con la intención de crear agentes con un comportamiento más humano, nos hemos basado en los agentes Nizorbot y Mirrorbot [Mihai Polcenau y otros, 2016] para crear nuevas 'hibridaciones' que tengan un comportamiento lo más humano posible.

Los agentes Mirrorbot y Nizorbot son bots humanizados que fueron creados para el videojuego Unreal Tournament 2004, también conocido como UT2004 o UT2K4. UT2004 es un videojuego de acción en primera persona orientada a la experiencia multijugador On-line [UT2004,2016].

Mirrorbot y Nizorbot fueron presentados en el campeonato internacional denominado "The 2K Botprize" [Botprize, 2016]. Este campeonato pretende poner en práctica técnicas de IA para definir el comportamiento de un agente y éste deberá superar un test de Turing aplicado a videojuegos.

El test de Turing aplicado a videojuegos se basa en la misma idea, pero en lugar de llevar a cabo una conversación, se llevan a cabo una serie de partidas sobre un videojuego.

El campeonato The 2K Botprize planteaba el siguiente test de Turing aplicado al videojuego UT2004: se jugarán una serie de partidas en las cuales habrá bots y humanos; además, habrá una serie de jueces(humanos) quienes al final de cada partida tendrán que votar quién es humano y quién no. El bot que consiga más votos como humano, gana la competición.

Mirrorbot fue el ganador de este campeonato en las ediciones de 2012 y 2014, mientras que Nizorbot fue subcampeón de la edición de 2014.

El comportamiento de Mirrorbot se basa en imitar a los jugadores humanos, para ello los identificaba (comprobando si disparan o no) e imita sus movimientos. Mientras no imite, su comportamiento no es especial, es más, se diseñó para que cometiera fallos.

Por otro lado, Nizorbot fue un proyecto realizado en la Universidad de Málaga que a su vez se basó en un bot Experto desarrollado por alumnos de la Universidad de Granada [R. Caballero y otros, 2013].

Éstas técnicas han sido de gran utilidad en el desarrollo de este proyecto ya que queríamos optimizar la solución de nuestro problema en un espacio de soluciones de tamaño considerable y la IA clásica no podría darnos una solución en un tiempo razonable o si quiera darnos una solución. Se ha abordado el problema que se plantea aplicando técnicas de inteligencia computacional y para comprobar la efectividad de este abordaje se ha llevado a cabo un Test de Turing aplicado a videojuegos. Para poder entender al completo el problema que abarca este proyecto y las soluciones a las que se ha llegado, se desarrollarán más ampliamente todos los temas mencionados anteriormente.

2. Conceptos Preliminares

En este capítulo se van a describir de forma breve algunos conceptos que han sido necesarios conocer para poder entender el problema que se nos plantea y la resolución llevada a cabo.

2.1. Computación evolutiva

Para la explicación de este término nos basamos en la estructura y el contenido del artículo [Computación evolutiva, 2016] y en el contenido de la tesis [Carlos Javier López Turiégano, 2015].

La computación evolutiva es una rama de la inteligencia artificial que engloba técnicas que simulan la evolución natural, constituyendo una alternativa para abordar problemas complejos de búsqueda, optimización y aprendizaje a través de modelos computacionales que simulan procesos evolutivos.

Los organismos vivos pueden resolver problemas que se le presentan y obtener habilidades para ello a través del mecanismo de la evolución natural. Esta evolución se produce como consecuencia de dos procesos primarios: la selección natural y la reproducción (o cruce).

La evolución natural es un proceso de cambio sobre una población reproductiva con una gran variedad de individuos con características heredables, lo que permite crear nuevos individuos partiendo de dos individuos que ya existían en la población, pero con diferentes características cada uno.

Toda esta base evolutiva parte del trabajo de Darwin en 1859 con el libro “El origen de las especies” [Charles Darwin y Gillian Beer, 1996]. Este contenía una serie de teorías que planteaban un mundo formado por un conjunto de individuos los cuales competían y evolucionaban para poder subsistir y perpetuar la especie en el tiempo. Las especies se crean, evolucionan y desaparecen si no se adaptan al medio de forma que sólo los que mejor se adapten al medio sobreviven y perpetúan la especie y sus aptitudes que le hacen adaptarse al medio.

En estas teorías, la computación ve un proceso claro de optimización ya que se toman una serie de soluciones aparentemente óptimas del espacio de soluciones de un problema – individuos mejor adaptados-, se mezclan estas soluciones-cruce de estos individuos-, se generan nuevas soluciones -nuevos individuos- que contienen información de las soluciones de las que fueron creadas – contienen parte del código genético de sus antecesores- y de esta manera el promedio de las soluciones de acercarse a la solución óptima mejora.

Éste trabajo de adaptación a la computación fue desarrollado por Bremermann, Friedberg, Box, entre otros, pero durante tres décadas se abandonaron todos estos trabajos debido a una falta de tecnología computacional para poder poner en práctica

estos trabajos teóricos. Hasta que con la llegada de nuevos trabajos como los de Holland, Rechenberg, Shwefel y Fogel, permitieron cambiar el escenario hasta que hoy en día el empleo de estos trabajos en la ciencia está creciendo exponencialmente.

Volviendo al campo de la optimización, un problema de optimización requiere hallar una serie de parámetros que cumplan un criterio de calidad, es decir, maximizar o minimizar una cierta función de evaluación f(x) dada.

Estos parámetros se encuentran en el espacio de soluciones del problema que son todas variables que pueden tomar valor en la función de evaluación. Pero hay una serie de funciones o problemas que son muy difíciles de resolver o que llevan una gran cantidad de tiempo poder resolverlos, ya sea porque el espacio de soluciones del problema es de un tamaño demasiado grande como para poder ser abarcado, por la fuerte no linealidad o diferenciabilidad de la función, etc. Pero, para este tipo de problemas, existen una serie de algoritmos y metaheurísticas que son especialmente útiles y uno de estos algoritmos son los algoritmos evolutivos.

2.1.1. Algoritmos evolutivos

Los algoritmos evolutivos son empleados en problemas de optimización o búsqueda de soluciones basándose en algún mecanismo de evolución natural, por lo tanto, su estructura intentará asemejarse lo máximo posible a este mecanismo de evolución natural.

Estos algoritmos trabajan con una población de individuos que representan un conjunto de soluciones a un problema dado. Esta población es sometida a un proceso de recombinación de individuos y a un proceso de selección de los mejores individuos. Cada vez que se termina este proceso se obtiene una nueva población llamada generación. Se espera que después de un número de generaciones exista un individuo (que será el mejor de la población) que se acerque o sea igual a la solución óptima del problema. Ésta búsqueda de la solución óptima combina una búsqueda aleatoria (los cruces de individuos son aleatorios) y una búsqueda dirigida (se seleccionan los mejores individuos en cada generación).

Los principales componentes de cualquier algoritmo evolutivo son:

- **Población de individuos:** una representación de posibles soluciones al problema.
- **Procedimiento de selección:** se basa en lo bueno que sean las aptitudes del individuo para resolver el problema.
- **Procedimiento de transformación:** construirá nuevos candidatos a partir de los anteriores.

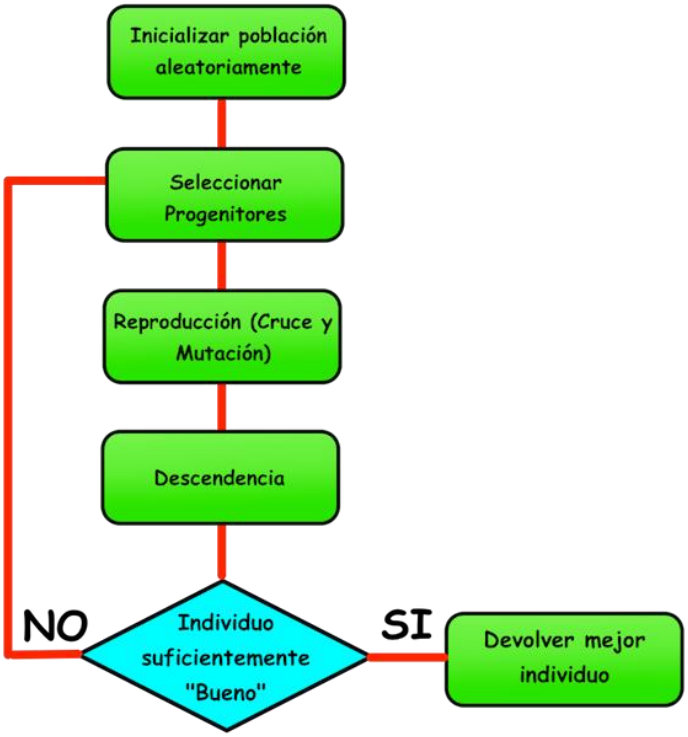


Figura 2.1. Estructura general de un algoritmo evolutivo

La imagen superior nos indica la estructura general de un algoritmo evolutivo. Existen varias ramas dentro de los algoritmos evolutivos y se diferencian por la manera de representar a los individuos, entre otros aspectos. Estas ramas son: **Estrategias evolutivas** (vectores reales), **Programación evolutiva** (máquinas de estados finitos), **Algoritmos genéticos** (cadenas binarias o de reales) y **Programación genética** (basada en la evolución de programas o árboles).

El problema de este proyecto ha sido abordado con algoritmos genéticos debido a su sencillez de implementación, por lo que a continuación se procederá a ver en qué consisten los algoritmos genéticos.

2.1.2. Algoritmos genéticos

Los algoritmos genéticos son una rama de los algoritmos evolutivos los cuales representan la población de individuos mediante cadenas de números, pudiendo ser binaria, entera o real.

Los algoritmos genéticos, al igual que las demás ramas de los algoritmos evolutivos, son usados en la búsqueda de soluciones de problemas de optimización combinatoria, ya sea para encontrar una solución o para optimizarla. Por esta razón se utiliza el concepto de cromosoma ya que un cromosoma está compuesto por genes y cada gen tendrá información sobre una variable en particular.

Usar el concepto de cromosoma como estructura de datos es bastante útil para problemas de combinatoria ya que un cromosoma, computacionalmente, es un array de valores binarios o reales y su tratamiento es más sencillo.

En cualquier algoritmo genético, por tanto, la población de individuos es representada mediante un conjunto de cromosomas la cual será evolucionada un

número de generaciones hasta obtener un individuo que se acerque lo máximo posible a la solución que se está buscando. Los individuos se diferencian entre sí por su nivel de aptitud el cual dependerá del problema en el que nos encontremos. El tipo de problema también influirá en los tipos de cromosomas y los valores que pueden tomar sus genes (así como el número de genes que va a tener).

La evolución de la población creará nuevas generaciones de poblaciones las cuales podrán estar formadas por los hijos de los individuos de la anterior generación. Estos hijos nacen a partir de dos individuos de una generación mediante los operadores biológicos de cruce y mutación. La selección tanto de los padres como de los individuos de una siguiente generación y el tipo de recombinación, pueden ser aleatorias o seguir una estrategia.

El nivel de aptitud de un cromosoma se mide a partir de la función de aptitud o la función fitness la cual se deberá ajustar al problema que se esté tratando. Por ejemplo, si estamos maximizando el equilibrio de daño que hace un bot y el daño recibido, deberemos definir una función fitness que tenga en cuenta el daño recibido y el daño realizado y dar un peso a cada valor. Un ejemplo de esta función sería:

$$f(x) = 2*Daño\_recibido – 0.5*Daño\_realizado.$$

Esta función será aplicada a cada individuo de la población y nos dirá la calidad de la solución que propone el individuo.

A continuación, se pasará a explicar los tipos de operadores de selección de padres y de los individuos de una siguiente generación usados en el proyecto:

- **Selección elitista:** sólo se escogen aquellos individuos con un valor de aptitud o fitness más alto. Normalmente no se suele usar un elitismo puro, sino que se usa una vez generado los nuevos individuos y mantener así en la población los mejores de la anterior.
- **Selección proporcional a la aptitud:** los individuos con un valor de aptitud o fitness más alto tienen mayor probabilidad de ser elegidos.
- **Selección por torneo:** se divide la población en subgrupos y los miembros de cada subgrupo compiten entre ellos eligiendo a un solo individuo de cada subgrupo.
- **Selección por estado estacionario:** los descendientes que se hayan generado reemplazarán a los miembros menos aptos de la generación anterior conservando algunos individuos entre generaciones.

Una vez se han escogidos los individuos padres, se pasa aplicar los operadores genéticos de cruce y mutación:

- **Cruce:** este operador genético consiste en intercambiar ciertos genes de dos padres generando un hijo con el mismo tamaño de genes que sus padres. La elección de los genes de cada uno de los padres se puede hacer de tres maneras

1. **Seleccionando un punto de corte** en el padre más apto y a partir de ese punto los siguientes genes estarán formados por los genes restantes del otro padre.
2. **Seleccionando un rango** en el padre más apto y rellenar ese rango con los valores análogos de los genes del otro padre.
3. **Seleccionando un número de genes sueltos** de uno de los padres más apto y sustituyendo esos genes por los del padre menos apto.

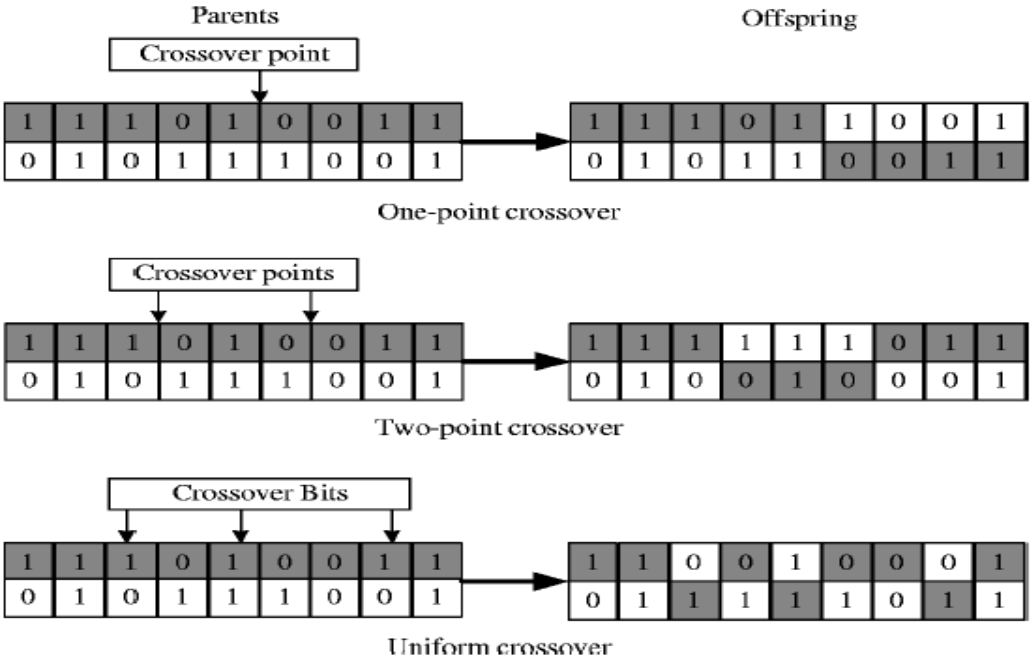


Figura 2.2. Algunos tipos de cruce.

- **Mutación**: este operador genético se aplica una vez se ha generado un hijo y consiste en recorrer el hijo gen por gen y aplicar una cierta probabilidad de mutación. Si se da la probabilidad de mutación en un gen, el valor de ese gen se verá modificado. En caso de tener cromosomas binarios la mutación convierte los 0 en 1 y en el caso de cromosomas reales se aplicaría la política que mejor convenga para el problema.

Otro concepto a tener en cuenta en los algoritmos genéticos es la estructura de los individuos de una población o si van a ser binarios o reales. La elección del tipo de estructura dependerá del problema en el que nos encontremos.

Por último, veremos las ventajas y desventajas de los algoritmos genéticos. Entre las ventajas encontramos: **paralelismo** (se busca la solución a un problema moviéndose simultáneamente en diferentes partes del espacio de soluciones), **manipulación simultánea** (manipulan muchos parámetros simultáneamente permitiendo manejar más de una solución al problema a la vez) y el **desconocimiento del problema** (permite que el algoritmo se mueva por el espacio de soluciones "aleatoriamente" y así poder abarcar el máximo número posible de combinaciones).

Algunas de las desventajas de los algoritmos genéticos son: la **representación**, **función fitness** (definir una función fitness adecuada para un problema y que sea lo justamente discriminatoria es difícil y crítico), **tamaño de la población** (no elegir adecuadamente un tamaño de población puede no encontrar una solución consistente) y el **ritmo de cruzamiento y mutación** (escoger un mal porcentaje de mutación o cruzamiento puede hacer que algoritmo nunca acabe).

2.2. Inteligencia artificial en los videojuegos

La inteligencia artificial nació con la idea de resolver problemas cotidianos o triviales para el ser humano, pero también para resolver problemas complejos que el ser humano no era capaz de resolver en un tiempo razonable o simplemente no podía resolver.

La inteligencia artificial general (AIG) [Inteligencia artificial general, 2013] es aquella con la capacidad de poder realizar con éxito cualquier tarea intelectual que pueda hacer un humano. También se conoce como IA fuerte o IA completa. Se suele asociar con rasgos como la consciencia, sensibilidad, inteligencia u auto-conciencia que tienen los seres vivos.

No hay una definición exacta o que satisfaga a toda la comunidad científica de inteligencia, pero si se ha llegado a un consenso sobre qué funciones debería realizar cualquier ente inteligente. Las funciones principales que debería tener son:

- Razonar
- Representar conocimiento y sentido común
- Realizar estrategias
- Aprender
- Comunicarse mediante lenguaje natural
- Hacer un uso correcto de estas funciones para alcanzar metas.

Además, se suelen incluir las capacidades de sentir y actuar en el mundo donde se vaya a situar el ente inteligente, así como la capacidad de imaginar o ser autónomo.

Actualmente existen sistemas, como la creatividad computacional o computación evolutiva, que son capaces de realizar numerosas funciones que se han mencionado anteriormente, pero todavía no alcanzan niveles humanos.

La **IA para videojuegos** intenta cumplir no sólo con las funciones que definen la inteligencia sino imitar el comportamiento del mundo real, es decir, si un coche en un videojuego choca con otro se espera que haya una reacción natural al choque.

Arthur Samuel fue un informático quien en 1959 escribió el libro “Some studies in machine learning using the game of checkers” y gracias a este trabajo programó una IA para el juego de las damas. Este trabajo fue el punto de partida de posteriores juegos de un jugador tales como Qwak, pusuit o hunt the Wumpus.

Gracias a los avances tecnológicos se pudo avanzar en el campo de la IA y en los años 70 se publicaron los juegos space invaders (1978) y Galaxian (1979), los cuales tenían enemigos que “aprendían” del jugador a través de funciones hash, y en los años 80 se publicaron los juegos Pac-Man (1980) y Kárate Champ (1984), los cuales introdujeron cierta personalidad a los enemigos.

Pero el avance más significativo de la IA en videojuegos se produjo en los años 90 con la salida de consolas más potentes que permitían implementar máquinas de estados, decisiones en tiempo real, toma de caminos (pahtfinding) o aprendizaje. Destacar en esta década el videojuego Half life (1999) la IA era capaz de crear estrategias para “ganar” al jugador utilizando varios NPCs.

Ya en el siglo XXI la tecnología ha avanzado tanto que permite crear IA con una potencia muy superior a todas las mencionadas anteriormente. Cabe destacar en este siglo las de los juegos:

- **Halo** creó IAs que identificaban y reaccionaban ante amenazas tales como granadas, precipicios o vehículos.
- **Far Cry** por su parte creó IAs que eran capaces de crear estrategias de acorralamiento hacia el jugador sin conocer la posición del mismo en cada momento, usando únicamente la última posición en la que fue visto.
- **FEAR** empleó por primera vez la planificación en tiempo real usando **GOAP** [Goal-Oriented Action Planning, 2017] permitiendo a las IAs adaptarse a todos los cambios que se producen en el entorno en tiempo real.
- **Batman Arkham City** dotó de una personalidad muy fuerte a los NPCs haciendo que reaccionasen emocionalmente ante sucesos que ocurrían en tiempo real.

En los comienzos de la aplicación de IAs para videojuegos, se implementaban con “trampa”, es decir, las IAs tenían información que un jugador humano no podía saber, como por ejemplo conocer la posición del contrario en todo momento.

Estas “trampas” se llevaban a cabo porque la tecnología del momento no permitía aplicar técnicas avanzadas de IA y se necesitaba de ellas para poder tener un juego el cual el usuario se divirtiera jugándolo. Además, esta limitación tecnológica no permitía aplicar ciertas técnicas de IA ni desarrollar nuevas.

En el momento en el que la tecnología avanzó ya se pudieron implementar todas estas técnicas útiles para el desarrollo de IAs para videojuegos.

A continuación, vamos a ver algunas de las estrategias y técnicas más usadas:

- **Chase and evade (capturar y huir):** es una técnica sencilla en la que un NPC sólo tiene dos objetivos: capturar al jugador o huir de él. Se divide en dos partes: primero decide si está en condiciones de ir a por el jugador o huye y, segundo, lleva a cabo la acción y para ello tiene que trazar una estrategia.

- **Pathfinding (o búsqueda de caminos):** esta técnica se utiliza para determinar el camino que debe seguir un NPC para ir de un punto A, a un punto B del escenario teniendo en cuenta las características del mismo, elevaciones, obstáculos, “niebla de guerra”, etc.
- **Redes neuronales:** es una técnica que sigue el modelo de las redes neuronales biológicas, por lo que está compuesto de una serie de entradas a la red, las salidas correspondientes y las neuronas que conforman las interconexiones de las entradas con las salidas.
- **Máquinas de estados finitos:** las máquinas de estados finitos son sistemas formados por un conjunto de diferentes estados y las transiciones entre cada uno de ellos. La transición entre estados se produce por eventos que ocurren en el entorno.
- **Agente experto:** un agente es un sistema capaz de percibir su entorno y actuar en consecuencia. Un agente experto divide su rutina de comportamiento en tres fases: percepción, planificación y actuación.

Estas técnicas no se emplean únicamente sobre los NPCs de un videojuego, también se usan en sistemas (dentro del videojuego) que le dan más realismo. Estos sistemas permiten tener un entorno vivo el cuál no está formado únicamente por personajes no controlados por un jugador, sino que tiene una serie de elementos, orgánicos o inorgánicos, que deben cambiar ante las acciones del jugador. Estos cambios deben ajustarse al tipo de elemento al que se le aplique un cambio, es decir, si se tira una papelera al suelo se espera que no empiece a rebotar como si se tratara de una pelota.

Pero, por otra parte, en el campo de los videojuegos muchas veces no se quiere tener un comportamiento real de personaje o entorno, si se está desarrollando un videojuego que ocurre en un mundo en el que la gravedad no es como el de la tierra, la física de ese mundo deberá ajustarse a esa gravedad y por tanto todos los movimientos y acciones de los personajes también se verán afectados y la IA que controle esas físicas deberá saber ajustar todos esos cambios al mundo.

2.3. Test de Turing

El test de Turing es aquel que planteó Alan Turing en 1950 para proporcionar una definición estándar de inteligencia. Alan Turing [Alan Turing, 2015] fue uno de los padres de los campos de la computación, la inteligencia artificial y de la informática actual. En 1936 publicó el diseño de una máquina capaz de resolver cualquier problema matemático que pudiera representarse con un algoritmo, que posteriormente se conocería como máquina de Turing. Se centró en el estudio de la inteligencia artificial creando en 1950 una prueba llamada “Test de Turing”. Este test no se centra en una lista cualidades que un agente debería cumplir para poder ser considerado inteligente, sino que se centraría más en la actuación del agente, es decir, si este es capaz de actuar de manera que es imposible determinar si es humano o no.

La prueba consiste en una conversación entre un humano y un computador (mediante chat por computadora) y un juez humano quién va a visualizar la conversación. Este juez, debe determinar quién es el humano y quién no. Si la computadora actúa de manera que el juez no puede dar una respuesta, la computadora ha pasado la prueba. [Russell, Stuart J. y Norvig, Peter, 2003].

Esta prueba ha sido criticada por el hecho de dejar a una persona (el juez) quien determina si una máquina tiene un comportamiento humano o no. Esta persona puede tener sus propias percepciones sobre cómo es el comportamiento humano y de la misma definición de inteligencia. Pero Turing no planteó esta prueba como una medida de inteligencia o de una cualidad humana, planteó esta prueba como una alternativa a la definición de la palabra pensar y de esta manera poder continuar con el estudio de este campo.

A partir de esta prueba se han ido creado pruebas alternativas de las cuales destacamos una en concreto por el contexto del problema en el que nos encontramos. Esta alternativa es el Test de Turing Aplicado a Videojuegos.

2.3.1. Test de Turing aplicado a videojuegos

El sector de los videojuegos es un sector que está creciendo año tras año y por lo tanto el público exige una cierta calidad de juego. Como hemos visto anteriormente, los primeros videojuegos estaban limitados por la tecnología, pero con los avances tecnológicos de hoy en día, para el público ya no es excusa poner las limitaciones tecnológicas como un impedimento para hacer juegos más “reales”. Por lo tanto, estar jugando contra un NPC o bot que juega de forma mediocre hace que el videojuego en sí también lo sea para el público. Por este motivo, si un bot es capaz de superar esta prueba significa que tiene un comportamiento humano y por lo tanto mejorará directamente la calidad del juego.

Un bot en un videojuego deberá interactuar con otros jugadores en un entorno multijugador. Esto significa que deberá tomar decisiones óptimas y humanas y tener un comportamiento humano hacia el jugador, todo al mismo tiempo.

En esta variación del test de Turing original el juez ya no tiene que visualizar una conversación entre una persona y una máquina, sino que deberá observar el comportamiento de los personajes en una partida en tiempo real de un videojuego. Por lo tanto, deberá observar otro tipo de aspectos para medir la “humanidad” de los personajes que aparecen en la partida, como por ejemplo el tipo de errores que cometen (o no), la navegación por el escenario, las estrategias que sigue para salir airoso de cualquier situación, ver que no hay monotonía en acciones, etc.

Esta variante fue propuesta por el profesor asociado Philip Hingston de la Universidad de Edith Cowan en 2008 y se implementó a través del torneo 2K Botprize. Se propuso con la idea de avanzar en los campos de la Inteligencia Artificial e Inteligencia Computacional en relación a los videojuegos. También sirvió para desacreditar la afirmación errónea de que “la IA para juegos es un problema resuelto”.

Este test se realiza a modo de torneo dentro del videojuego que sirve como base para poder poner en práctica las diferentes técnicas de IA “experimentales” aplicadas a cada bot. Este torneo tendrá los siguientes objetivos:

- Hay tres tipos de participantes: un jugador humano, un bot y un juez.
- El bot deberá parecer más humano que el jugador humano. Para ello habrá un sistema de calificaciones del 1 al 5 (1= No es humano, 5=Humano) que usará el juez para calificar a cada jugador.
- Los tres participantes serán indistinguibles salvo por el nombre, que será aleatorio. De esta forma, el juez sólo estará influido por el comportamiento de los jugadores y no por el nombre o apariencia.
- No hay chat durante la partida.
- Los bots no pueden tener más información de la que tendría un jugador humano. Sólo reaccionan ante estímulos de la partida.

El torneo 2K Botprize fue el primero en llevar a cabo esta variación de test de Turing con la estructura mencionada anteriormente. La primera edición se realizó el 17 de diciembre 2008 en Perth, Australia, como parte del simposio IEEE 2008 sobre Inteligencia Computacional y Juegos (CIG 2008).

En este torneo se jugaban partida en el videojuego Unreal Tournament 2004. Se llevaban a cabo varias rondas utilizando partidas Deathmatch de 10 minutos de duración. Los jueces se unen los últimos a la partida y sólo observan una vez a cada jugador y bot.

Desde la primera edición de 2008 hasta la edición de 2011, no se consiguieron puntuaciones representativas, pero fueron mejorando hasta que, en la edición de 2012, dos bots consiguieron superar en puntuaciones de humanidad a los jugadores humanos. Estos bots fueron: UT^2 y Mirrorbot.

El bot UT2 fue creado por un equipo de la Universidad de Texas aplicando técnicas que imitaban el comportamiento humano y la neuroevolución. Por otro lado, Mirrorbot fue creado por Mihai Polceanu [Mihai Polcenau, 2013], un estudiante de doctorado de Rumanía.

Dentro de las diferencias que existen entre el test de Turing original y esta variante, debemos destacar las siguientes:

- Los humanos actúan para ganar la partida, al igual que los bots.
- El escoger un determinado juego se debe por la disponibilidad comercial y facilidad para incluir material nuevo o personalizado en el juego y por lo tanto estará limitado por el número de acciones que se pueden hacer dentro del juego. Mientras que el test de Turing tradicional, al ser una conversación, las preguntas son más numerosas que el conjunto de acciones del juego.
- Los bots no oyen ni ven, sólo interactúan con datos y eventos.

- Los jueces no pueden incluir nuevos elementos dentro de la partida, cambiando la jugabilidad. Mientras que el test tradicional el juez puede incluir preguntas que deben responder humano y máquina.

Algunas de estas diferencias muestran ciertas carencias en esta variación a las que debemos añadir el hecho de que las veces que se ha llevado a cabo esta variación mediante el torneo Botprize, las puntuaciones medias de humanidad que han recibido los jugadores humanos ha sido de un 41,4% y Turing puso como condición de superación del test que se debería conseguir al menos un 50%. Por lo tanto, si con estas condiciones actuales ni un humano puede superar el test, no es del todo fiable esta prueba como medidor de humanidad de un bot.

Por otro lado, también demuestra la dificultad de imitar el comportamiento humano ya que nos indica que este comportamiento es más complicado y cuantitativa de lo que se pensaba. Por este motivo, los organizadores de Botprize aumentarán la dificultad del torneo en futuras ediciones para ser más exhaustivos.

2.4. Unreal Tournament 2004

Unreal Tournament 2004 o UT2K4, UT2004, es un videojuego de disparos en primera persona (FPS) con ambiente futurista, competitivo, multijugador On-Line desarrollado por Epic Games y Digital Extremes.

Es la tercera entrega de la subsaga de Unreal Tournament la cual forma parte de la franquicia Unreal. UT2K4 es considerado por una gran mayoría como una extensión de los trabajos realizados en la entrega anterior, Unreal Tournament 2003 (UT2003). Mantendría la mayoría de los mapas y personajes, mejoró ciertos fallos que aparecieron en UT2003 y agregó casi 50 nuevos mapas, combate en vehículo y nuevos modos de juego. Fue lanzado el 16 de marzo de 2004 para PC y el 31 de marzo de 2004 para Mac OS X.

UT2004 dispone de una gran cantidad de modos de juego los cuales algunos de ellos son **Deathmatch**, **Team Deathmatch** o **capturar la bandera** entre otros.

De todos ellos, el único modo que nos interesa para el proyecto ha sido el modo **Deathmatch** ya que, por un lado, el test de Turing aplicado a este videojuego se lleva a cabo en el modo **Deathmatch** y, por otro lado, porque los bots desarrollados en este proyecto han sido puestos en práctica bajo este modo de juego.

Deathmatch es un tipo de juego cuyo objetivo es o bien llegar a un cierto número de frags (muertes), o bien terminar la partida con el número de frags más alto.

Este modo de juego consiste en tener el mayor número de muertes (o frags) en un cierto tiempo límite o ser el primero en alcanzar un cierto número de frags.

2.5. Tecnología empleada

A continuación, se mostrarán las librerías, así como Software necesario para poder desarrollar este proyecto.

2.5.1. Pogamut

Pogamut [Pogamut] es un middleware Java que permite el control de agentes virtuales en múltiples entornos proporcionados por motores de juego. Actualmente son compatibles Unreal Tournament 2004 (UT2004), UnrealEngine2 RuntimeDemo (UE2), Unreal Development Kit (UDK) y el juego DEFCON Pogamut proporciona una API Java para la creación y control de agentes virtuales y una interfaz gráfica (plugin para NetBeans) que simplifica la depuración de los agentes.

El principal objetivo es simplificar la parte "física" de la creación del agente. La mayoría de las acciones en el entorno (incluso los complicados, como búsqueda de caminos y la recolección de información en la memoria del agente) se puede realizar en uno o dos comandos. Esto permite al usuario concentrar sus esfuerzos en las partes que más le interese.

Algunas características principales:

- Biblioteca de Java para la codificación de los robots en Unreal Tournament 2004.
- Un entorno de desarrollo integrado (IDE) con el apoyo de depuración (plugin para NetBeans) con la siguiente funcionalidad:
    - *Gestión de los bots que se ejecutan*
    - *Visualización 3D del mapa*
    - *Editor visual de planes POSH*
- Ejecución automática de partidas en UT2004 (Deathmatch y Captura-la-bandera)
- Dos modos de juego personalizados para UT2004: Tag y Hide and Seek.

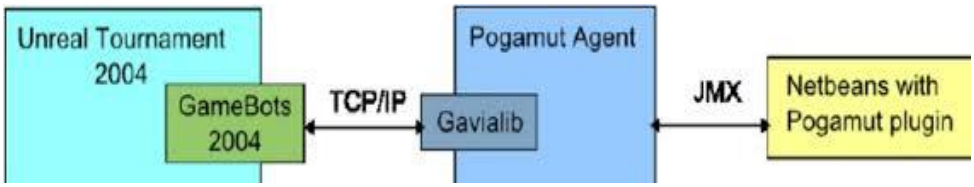


Figura 2.3. Arquitectura de Pogamut.

La información del entorno de UT2004 se exporta a través de TCP / IP por el protocolo de texto GameBots2004. Estos mensajes de texto son procesados por la librería Java Gavialib, por lo que el agente Pogamut puede trabajar con objetos Java.

El agente Pogamut se puede depurar de forma remota a través del protocolo JMX usando el plugin para Netbeans de Pogamut. GameBots 2004 es una modificación (MOD) para el UT2004.

GameBots, escrito en UnrealScript, proporciona un protocolo de red basado en texto para la conexión a UT2004 y el control en el juego de los avatares (los bots). Con GameBots el usuario puede controlar los bots con comandos de texto y, al mismo tiempo, recibir información sobre el entorno del juego en mensajes de texto.

El objetivo principal de GameBots es proporcionar un entorno para el desarrollo de agentes virtuales para UT2004, permitiendo una fácil conexión a UT2004 través de su protocolo de texto.

El objetivo principal de GameBots es proporcionar un entorno para el desarrollo de agentes virtuales para UT2004, permitiendo una fácil conexión a UT2004 través de su protocolo de texto.

2.5.2. Maven

El proyecto Apache Maven [Maven, 2016] es una herramienta de gestión y compresión de proyectos software, desarrollado por la Apache Software Foundation. Tiene un propósito similar a la herramienta de Apache Ant, pero funciona diferente y utiliza conceptos diferentes. Funciona en proyectos de lenguaje Java, C#, Ruby y Scala, entre otros lenguajes.

Maven utiliza un POM (Project Object Model) para describir la construcción de un proyecto Software, por lo que incluye las dependencias con módulos y componentes externos, así como el orden de construcción de los elementos. Además, realiza las operaciones de compilado y empaquetado como tareas predefinidas en cualquier proyecto.

Maven tiene un funcionamiento On-Line ya que descarga de uno o más repositorios, principalmente el repositorio central de Maven 2, las diferentes librerías Java y Plugins de Maven necesarios para la construcción del proyecto.

Maven es el sistema de dependencias usado por Pogamut, por lo que ha sido necesario su aprendizaje para poder usar esta librería.

2.5.3. Extensible Markup Language (XML)

XML o Extensible Markup Language, es un lenguaje de marcado para documentos basado en SGML (Standard Generalized Markup Language). Desarrollado por World Wide Web Consortium (W3C).

Se utiliza para describir datos de manera estructurada e independiente de la plataforma en la que se esté trabajando. Esta característica permite que diferentes plataformas puedan intercambiar información e incluso comunicarse entre ellas sin tener en cuenta el lenguaje en el que trabajen cada plataforma o la propia estructura de cada una.

Este lenguaje es el que utiliza Maven para crear los POM de construcción de los proyectos Maven.

2.5.4. Google App Engine

Google App Engine es un servicio web PaaS (Platform as a Service) que presta Google de forma gratuita permitiendo ejecutar aplicaciones web sobre su infraestructura alojada en la Google Cloud.

Como cualquier servicio Paas, Google App Engine facilita una infraestructura propia a la que el cliente debe adecuarse. Por ello, cualquier cliente que quiera hacer uso de este servicio debe adecuarse a la tecnología que soporta dicha infraestructura, así como a las políticas de equilibrado de carga, escalado y replicación, máquinas virtuales, entre otras, que tenga. Un servicio PaaS ofrece al cliente un IDE de desarrollo el cual incluye:

- Un entorno de desarrollo (SDK).
- Un entorno de ejecución local, para simular el entorno cloud en el que se desplegará la aplicación web
- Una serie de herramientas de despliegue.
- Unos servicios de almacenamiento.

En nuestro caso el SDK de desarrollo (así como el entorno de ejecución) ha sido la aplicación Eclipse Luna, la herramienta de despliegue ha sido mediante empaquetamiento WAR y el servicio de almacenamiento ha sido la Google Cloud Datastore, que es con la que trabaja la infraestructura ofrecida por Google App Engine.

Google Cloud Datastore es una base de datos noSQL. Trabaja con entidades en lugar de tablas y las consultas se realizan mediante SQL-like.

Este servicio ha sido usado en el proyecto para alojar la aplicación web desarrollada para la realización del test de Turing aplicado a videojuegos que se ha llevado a cabo.

2.5.5. Java Server Faces

Java Server Faces (JSF) [Java Server Faces, 2016] es un framework que usa el patrón MVC (Modelo Vista Controlador) y que está basado en la API de Servlets la cual proporciona un conjunto de componentes en forma de etiquetas definidas en páginas XHTML mediante el framewrok Facelets. Facelets es el elemento fundamental de JSF ya que proporciona características de plantillas y de creación de componentes compuestos.

JSF usa páginas Facelets como vista, objetos Javabean como modelos y los métodos de esos objetos como controladores. Por ello, JSF tiene las siguientes características destacables:

- Incorpora componentes de interfaz de usuario, gestión de eventos, validación de datos en el servidor, etc.
- Define el flujo de navegación entre páginas.
- Utiliza librerías de etiquetas propia sobre JSP.

- Tiene soporte para internacionalización.
- A partir de la versión 2.0 soporta una comunicación Ajax entre vista y servidor.

Un proyecto web que use este framework se divide en dos partes: una parte java en la que guarda la lógica de la aplicación (los modelos y controladores) y otra parte WAR en la que se guardan las vistas en formato XHTML.

Este framework ha sido el que se ha usado para el desarrollo de la aplicación web. La elección de este framework ha sido porque ya sabía manejarlo y porque Google App Engine lo soportaba.

2.5.6. Windows Movie Maker

Windows Movie Maker [Windows Movie Maker, 2016] es un software de edición de video creado por Microsoft. Fue incluido por primera vez en el año 2000 con Windows ME. Contiene características tales como efectos, transiciones, títulos o créditos, pista de audio, narración cronológica, etc.

Este software ha sido usado para la edición de los vídeos mostrados en el test de Turing aplicado videojuegos que se ha llevado a cabo en este proyecto.

2.5.7. Bandicamp

Bandicam [Bandicamp, 2016] es un programa del género screencast para aplicaciones 3D o videojuegos teniendo la posibilidad de hacer capturas, tanto de video como de pantalla. Además, permite grabar videos flash en el navegador, igual que capturar dispositivos de HDMI o webcam.

Este software ha sido usado para grabar las partidas jugadas por los bots desarrollados en Unreal Tournamnet 2004, los cuales han sido mostrados posteriormente en el test de Turing llevado a cabo.

3. Descripción del problema

Los bots que se han desarrollado en este proyecto (y que se explicarán en el siguiente punto) son hibridaciones de los bots Mirrorbot y Nizorbot.

Por este motivo, antes de explicar el desarrollo de las hibridaciones, en este punto vamos a describir estos bots.

3.1. Mirrorbot

Este bot fue desarrollado por Mihai Polceanu un estudiante de doctorado de Rumanía. Para explicar el funcionamiento y el desarrollo de este bot, me voy a basar en el artículo [Mihai Polcenau, 2013].

Mirrorbot fue desarrollado para el concurso Botprize del cuál fue ganador en las ediciones de 2012 y 2014. En la edición de 2012 consiguió un porcentaje del 52,2% lo que significó que fue la primera vez que un bot superaba el test de Turing planteado en el concurso.

El planteamiento para el desarrollo de este bot se basa en un tipo de neurona que tiene el ser humano denominada “neurona imitadora”. Con esta neurona cualquier ser humano puede aprender a partir de la imitación y es capaz de adaptarse al medio.

Si extrapolamos la imitación a cualquier tipo de contexto social, como, por ejemplo, una sociedad en la que se saluda de una manera particular y otro ser humano imita ese saludo no se verá negativamente si no que será aceptado socialmente.

En un juego FPS también existen interacciones sociales ya que cuando dos jugadores se enfrentan hay un comportamiento común para los dos como podría ser esquivar los ataques del contrario y para ello lo común es agacharse, esconderse detrás de un parapeto, etc.

Mirrorbot intenta explotar esa aceptación en la que los jugadores asumen que te pueden imitar hasta un cierto grado con el fin de evitar la inconsciencia del bot. El bot fue diseñado con la asunción de que, si el jugador imitado es humano, Mirrorbot obtiene el nivel de humanidad de quién está siendo imitado. Esto es doblemente humano ya que el hecho de tener un comportamiento humano ya te da un nivel de humanidad y, además, el proceso de imitar también le da un nivel de humanidad.

El planteamiento seguido para el desarrollo del bot se basa en la manera en el que ser humano y un bot perciben el entorno virtual y cómo se mueven por él.

Estas diferencias son las que determinan el comportamiento de uno y de otro. En el caso del jugador humano, sus movimientos tienen un cierto grado de retraso y fallo a la hora de coordinar la mano del ratón y la mano del teclado.

Mientras que, en un bot, sus movimientos son más precisos, perfectamente coordinados pero su percepción del mundo hace que tenga problemas para evitar obstáculo o utilizar

ascensores. Por este motivo, se intentó desarrollar Mirrorbot de manera que la percepción y la reacción estuvieran separadas.

Mirrorbot se compone de dos módulos los cuales se irán activando durante el juego. Estos módulos son: el módulo default y el módulo mirroring.

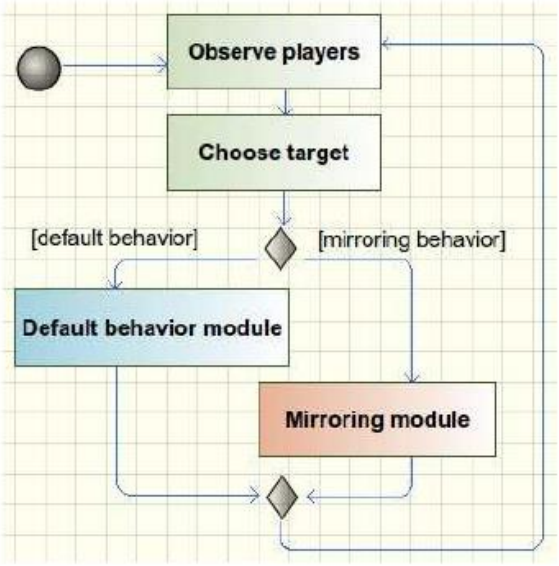


Figura 3.1. Arquitectura Mirrorbot

3.1.1. Módulo default

El módulo default implementa el comportamiento por defecto que tiene cualquier jugador de UT2004, por lo que se encargará de la navegación, apuntado, disparo, elección de objetivo y reconocimiento del enemigo. En este módulo se intenta separar las acciones de percepción del entorno e interacción con él. Por este motivo, sólo dispara cunado el bot está orientado de manera que puede hacer daño a un enemigo mientras que el apuntado se encarga del posicionamiento en el entorno para poder hacer daño.

3.1.1.1. Observar jugadores

Es la principal entrada de información de las acciones de los demás jugadores. Estudia la dirección de disparo de los otros jugadores y el nivel de agresividad con la que está disparando. Este módulo afecta a la elección del objetivo, activación del módulo espejo y al comportamiento evasivo.

3.1.1.2. Puntería

El módulo de puntería se basa en el ajuste sucesivo de la orientación del bot hacia el oponente teniendo en cuenta su nivel de agresividad y un rencor limitado por tiempo. Una vez esté asignado, el objetivo se mantendrá hasta que pase el tiempo de rencor. El nivel de rencor se rellena si el enemigo responde con fuego. Además, la orientación tiene en cuenta la futura posición en la que estará en base a su velocidad y dirección. Si no se tiene un objetivo, apunta a la interpolación lineal entre los dos siguientes puntos de navegación, determinado por el módulo de navegación. Esto

permite al bot salir de una esquina mirando en la dirección correcta en caso de que aparezca un enemigo.

3.1.1.3. Disparo

El módulo de disparo es responsable de elegir el disparo primario o secundario y qué arma va a usar, basado en la eficiencia del arma y la munición disponible.

3.1.1.4. Esquiva

Tiene el objetivo de proteger al bot, utiliza el módulo de observación y evita ser visto por los enemigos. En combate, esquivará los disparos del enemigo calculando la dirección de disparo del enemigo. Esto se cumple para todos los jugadores que estén disparando al mismo tiempo, por lo que el bot se moverá esquivando las direcciones de disparo de los jugadores.

3.1.1.5. Navegación

En el caso en que el bot no pueda salir de una posición debido a un mal funcionamiento del algoritmo de navegación en grafo, se activará un segundo módulo de navegación basado en raycasting. Este módulo de raycasting utiliza 24 *rays*, dividido en 16 rays horizontales (detectan colisiones) que salen desde el centro del personaje y 8 rays (detectan agujeros o precipicios) que salen a 45 grados desde el mismo punto y hacia abajo. El módulo principal de navegación es una mejora del que trae por defecto [Pogamut Logue, 2009], modificado con trayectorias más suavizadas, recogidas de recursos más avaricioso y una esquiva de obstáculos adicional usando los mismos rays que la navegación anterior.

Figura 3.2. Módulo RayCasting de navegación Mirrorbot

3.1.1.6. Juez

El mecanismo de juicio de humanidad del bot se basa en la opinión de la mayoría. Los datos se recogen usando el módulo de observación el cual graba los juicios de los demás jugadores. Mientras obtenga más datos, el bot cambiará su decisión en cada jugador, y cuando cambie de decisión, Mirrorbot ajustará la votación hacia ese jugador que ha sido cambiado por otro.

3.1.2. Módulo Espejo

El módulo mirroring se encarga de grabar las acciones de un oponente y después de un pequeño tiempo reproduce las acciones desde la perspectiva del bot. Cuando este módulo se activa es porque ya tiene un candidato a imitar y se desactiva el módulo *default*. A partir de ese momento empieza a grabar los movimientos y disparos del jugador.

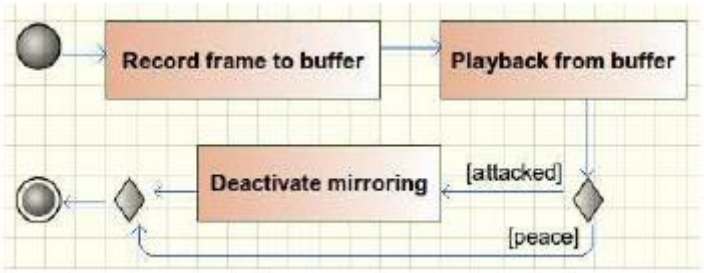


Figura 3.3. Arquitectura módulo espejo

La elección del candidato se hace en función de su agresividad y de su orientación hacia el bot. Una vez que el módulo espejo está activado, la información del jugador se guarda como fotogramas clave que contienen el estado exacto del jugador objetivo. Cada estado consiste en la posición del jugador, velocidad, rotación, salto, si está agachado, modo de disparo y el arma que lleva. Cuando se ejecuta un frame clave, el estado de Mirrorbot cambia al frame clave que se está ejecutando.

La posición del objetivo se usa para establecer un eje de simetría con Mirrorbot. Al usar este eje, todas las acciones son recreadas como si fueran vistas en un espejo. Las acciones de avanzar o retroceder se llevan a cabo al revés para mantener la distancia con el objetivo en caso de que avance o retroceda.

Este comportamiento de imitación se desactivará cuando hayan pasado ocho segundos o si está siendo atacado por otro jugador o han pasado 3 segundos sin ver al objetivo.

La desventaja de este sistema se da cuando Mirrorbot se cruza con otro bot, ya que si los movimientos a imitar son robóticos o no son nada humanos los imitará igualmente. Se intentó solucionar añadiendo un sistema de control de estancamiento de enemigo y un conjunto de pequeños movimientos aleatorios en las reproducciones.

Pero esta solución sólo aminora su impacto. Por lo que la solución definitiva sería tener un sistema que estudie los movimientos de los jugadores y sea capaz de identificar quién es humano y quién bot.

Mirrorbot no se centra en tener el máximo de puntos posibles sino en tener el mayor número de comportamientos humanos posibles. Por este motivo los resultados de Mirrorbot han mostrado que este sistema puede ser aplicado a cualquier contexto de interacción humana, es decir, se podría aplicar en contextos más allá de los videojuegos ya que al ser un módulo independiente y poco costoso computacionalmente puede ser acoplado a otros módulos diferentes.

3.2. Nizorbot

Nizorbot es el resultado de aplicar un AG interactivo al bot creado por Francisco Aisa García y Ricardo Caballero Moral, estudiantes de la Universidad de Granada, denominado ExpertBot.

El flujo de la IA de ExpertBot [José Luis Jiménez López, 2015] se basa en el conocimiento de un jugador humano experto, Me$$!@h, perteneciente al mejor clan español de UT2004, Trauma Reactor. Este jugador ha diseñado e implementado el flujo principal del comportamiento del bot. Este flujo de IA se ha implementado mediante dos niveles de máquinas de estados finitas (FSMs).

ExpertBot está formado por dos capas [R. Caballero y otros, 2013]: una capa cognitiva encargada de elegir los estados y sub-estados del bot a partir de los estímulos recibidos mediante sensores del entorno y una capa reactiva la cual no lleva a cabo ningún tipo de razonamiento, sólo reacciona ante estímulos durante la partida.

Además, usa una base de datos en SQLite de una sola tabla a modo de memoria del bot. Esta memoria le servirá para almacenar las localizaciones de los diferentes objetos del mapa, así como el mapa en el que se encuentran.

Los puntos fuertes de este bot son la transición de los estados y sub-estados, así como el razonamiento que sigue para elegir esos estados y sub-estados.

La navegación y la información sensorial utilizada es la proporcionada por Pogamut. Por este motivo no se ha podido usar la máquina de estados finita que viene en el bot nativo de UT2004 y se tuvo que implementar una arquitectura completa.

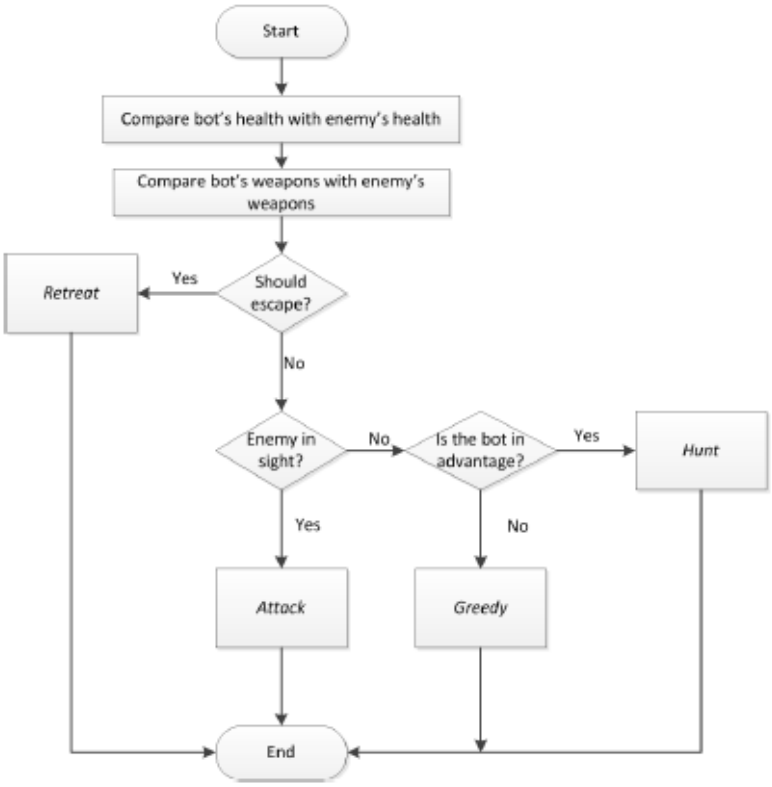


Figura 3.4. Diagrama de flujo general de Bot Experto.

3.2.1. Máquinas de estados finitas

El módulo de IA considera dos niveles de estados: primarios, corresponden a las acciones generales para llevar a cabo y secundarios, se dedican a realizar tareas adicionales dentro del flujo principal (cómo buscar vida).

Los estados primarios son: **Attack** (ataca al enemigo y si no lo ve se mueve pendularmente esperándolo), **Hunt** (busca al enemigo a partir de la última posición conocida y los ruidos que escuche. Si lo ve, lo persigue), **Retreat** (huye del enemigo si lo está viendo si no, huirá del enemigo a partir de la última posición conocida e inferirá su posición a partir de los ruidos que haga), **Greedy** (recoge todos los ítems que vea por el escenario), **Camp** (mantiene al bot agazapado esperando al enemigo)

Los estados secundarios son: **Disabled** (estado secundario está desactivado, el control total lo tiene el estado primario), **DefensiveProfile** (se aleja del enemigo si está a la vista), **OfenisveProfile** (se acerca al enemigo si lo ve), **PickupWeapon** (recoge todas las armas que encuentre), **PickupAmmo** (recoge toda munición que encuentre), **PickupHealth** (recoge todo ítem de vida que encuentre), **CriticalHealth** (hará uso de la base de datos para ir inmediatamente al ítem más cercano conocido) y **CritialWeaponry** (hará uso de la base de datos para ir inmediatamente al arma más cercana conocida.)

El estado camp no se incluye al ser un estado especial. La elección del estado se realiza en base a la información percibida por los sensores, el estado de salud, vida del bot, las armas que tiene el bot y las que tiene el contrario, así como la vida que tiene el bot y el contrario, etc.

Mientras se ejecutan los estados se continúa comprobando las condiciones para cambiar de estados secundario y primario cuando se necesite.

En general, el comportamiento del bot será defensivo cuando su salud sea más baja que la del enemigo, a menos que este tenga una salud crítica lo que significará que intentará matarlo. Por el contrario, si tiene igual o más salud comparará las armas que tiene el bot con las del enemigo. El flujo general se muestra en la siguiente figura.

3.2.2. Algoritmo genético interactivo

Existen numerosos problemas en los cuales formular la función de fitness es bastante complejo y es más sencillo que un humano fuera evaluando a los individuos. Pero, si el humano tuviera que evaluar a cada individuo de la población el tiempo de ejecución sería muy elevado, por lo que se ejecuta un algoritmo genético en el que tanto la máquina cómo el humano evalúan a los individuos.

Las características de este algoritmo eran las siguientes:

3.2.2.1. Cromosoma

El cromosoma de los individuos se divide en 6 bloques utilizando un total de 26 genes, los cuales eran:

- **Genes del 0 al 2:** genes de distancia, los cuales consideran la distancia con el enemigo y atacar/defender con un arma u otra.
- **Genes del 3 al 11:** genes de prioridad de arma. Un gen por cada arma del juego, les dará un valor de prioridad a ser cogida o usada.
- **Genes del 12 al 13:** genes de salud que determinan un comportamiento ofensivo, defensivo o neutral.
- **Genes del 14 al 18:** genes con valores de riesgo de salud dispuesto a sacrificar el bot.
- **Gen 19:** gen cuyo valor determina el tiempo que considera el bot que ha perdido de vista a un enemigo.
- **Genes del 20 al 25:** genes de prioridad de ítem. Dan prioridad para recoger un ítem u otro.

3.2.2.2. Fitness

La función fitness es la siguiente:

$$f(fr, d, dmgG, dmgT, s1, s2, tS, tL) = \begin{cases} (fr - d) + s2 + \frac{s1}{2} + \log((dmgG - dmgT) + 1), fr \geq d \\ \frac{fr}{d} + s2 + \frac{s1}{2} + \log\big((dmgG - dmgT) + 1\big) \\ , fr < d \end{cases}$$

fr es el número de frags, **d** el número de bajas, **dmgG** daño total infligido, **dmgT** daño total recibido, **s1** número de escudos pequeños recogidos, **s2** número de escudos grandes recogidos, **tS** tiempo acumulado con el Shock Rifle en posesión, **tL** tiempo acumulado con la Lightning Gun en posesión.

Debido a la naturaleza no determinista del juego se optó por usas factores que estabilizaran la función fitness, ya que el bot puede haber recogido los items y armas correctamente y haber jugado una buena partida pese a no conseguir un buen número de frags.

Estos factores sirven para evaluar otros aspectos del bot, sobre todo su comportamiento a la hora de recoger objetos, ya que los objetos juegan un papel muy importante en UT2004.

Con la función fitness descrita anteriormente se consigue evaluar estos factores que son **número de escudos recogidos, daño total hecho y recibido** y **tiempo total de posesión de armas**.

3.2.2.3. Mecanismo de selección

Se utilizó un algoritmo de selección por ruleta. Se elige al individuo proporcionalmente a su valor fitness. Además, se ha hecho uso de elitismo 5 el cual coge siempre para cada generación a los 5 mejores individuos de la generación anterior sin modificación alguna.

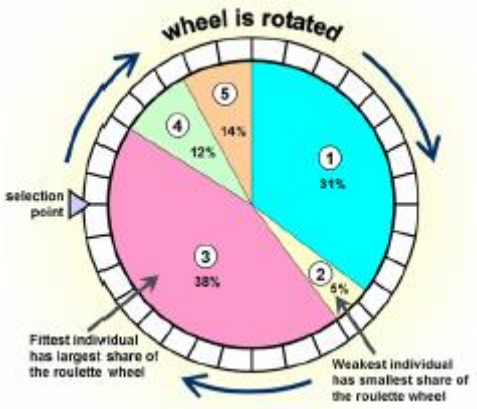


Figura 3.5. Algoritmo de selección por ruleta

3.2.2.4. Operadores genéticos

Se ha optado por la opción del cruce uniforme. Este tipo de cruce implica una mayor diversidad del individuo, donde cada gen del individuo descendiente tiene la misma probabilidad de ser heredado de uno de los dos progenitores.

3.2.2.5. Datos empleados para la ejecución del algoritmo

El rival utilizado durante las partidas de evolución ha sido el Bot Experto original. Además, las partidas eran de 5 minutos en el mapa DM-DE-Ironic, el tamaño de población era de 30 individuos, se generaban hasta 50 generaciones y la variación en la mutación era del 10% y la probabilidad de mutación del 33%

Además, la evolución de los individuos se realizó mediante la supervisión del jugador experto que visualizaba las partidas de los individuos. Cabe mencionar que no tuvo que ver todas las partidas disputadas, sino que se marcaban ciertos valores o condiciones de parada del algoritmo para preguntar al experto si se estaba evolucionando por el buen camino. Además, este podía decidir si un gen ya estaba optimizado, bloqueándolo, y a partir de ese momento ese gen ya no se vería modificado.

4. Desarrollo del proyecto

En este punto se abordará el desarrollo de las distintas hibridaciones creadas a partir de Mirrorbot y Nizorbot así como la justificación de los mismos.

4.1. Abordaje

Tras estudiar en detalle los comportamientos de Mirrorbot y Nizorbot, se han descubierto ciertas carencias en ellos y con ciertas hibridaciones de ambos bots se conseguirían otros bots con comportamientos más humanos. Este estudio se llevó a cabo realizando partidas contra los bots (humano vs bot) y generando los casos extremos de fallo en ambos bots. Un ejemplo de caso extremo era llevar a los bots a un ascensor ya que Nizorbot tiene problemas para cogerlos y Mirrorbot no.

En el caso de Nizorbot, su sistema de disparo es demasiado perfecto, es difícil que falle un disparo y en un combate uno contra uno, un humano tiene más porcentaje de fallo. Si además de ese detalle añadimos que dispara siempre a la cabeza (es donde se hace más daño), más se acentúa el hecho de que no parece humano. Sin embargo, la manera que tiene de elegir el arma para entrar en combate o qué ítem (de vida, munición, arma o protección) necesita en un momento dado, se consideró que es bastante racional.

En el caso de Mirrorbot, su algoritmo de pathfinding no sólo determina el mejor camino, sino que define cómo se va moviendo, es decir, no va siempre corriendo a todos lados y mirando hacia adelante, sino que corre, da saltos y va mirando a su alrededor mientras llega a su destino, lo que se consideró un detalle bastante humano. Por contraposición, su sistema de disparo es demasiado imperfecto. La tasa de fallo que tiene a la hora de disparar se concluyó que es demasiado alta. La detección de estos defectos y virtudes se comprobó observando, como espectador, una serie de partidas entre Mirrorbot, Nizorbot y ExpertBot.

Después de estudiar las virtudes y defectos de cada bot, se han planteado tres hibridaciones:

- **Mizorbot**: va a tener todo el comportamiento de Nizorbot pero con el pathfinding de Mirrorbot.
- **Mirzorbot**: va a tener el comportamiento de Mirrorbot pero con los estados de Nizorbot y el disparo de Mirrorbot. Es decir, las decisiones de a dónde va y a dónde mira lo determinará el motor de Mirrorbot y la decisión de si tiene que huir o necesita un ítem en un momento dado, lo determinará el motor de Nizorbot.
- **Mirrorbot-Evolutivo**: va a ser Mirrorbot pero al que se la va a aplicar un algoritmo genético no interactivo. Para ello, primero se determinarán los parámetros y sus rangos, se diseñará el algoritmo genético a llevar a cabo, la función fitness y, por último, se llevará a cabo la ejecución del algoritmo jugando partidas contra el Bot-Experto. Al finalizar, se escogerán los mejores valores de

parámetros y se pasará al estudio del comportamiento del bot mediante una serie de partidas.
- Para las pruebas de Mizorbot y Mirzorbot, se realizarán partidas para comprobar fallos de programación y para el estudio de la hibridación y su rendimiento. En el caso de Mirrorbot-Evolutivo, se realizarán partidas para comprobar su rendimiento.

4.2. Mizorbot

Esta hibridación contiene todo el comportamiento de Nizorbot pero se ha redefinido su pahtfinding usando el de Mirrorbot. Esta redefinición se debe a que uno de los puntos flojos que tuvo Nizorbot durante la edición de la Botprize en 2014 era su pathfinding (no era capaz de saltar), mientras que para Mirrorbot era uno de sus puntos fuertes.

A partir de las evaluaciones que se hicieron en la botprize de 2014 y estudiando el comportamiento del bot en partidas, se decidió que el pathfinding de Mirrorbot era mejor que el de Nizorbot y, además, se intentaría modificar lo menos posible el código original.

Para poder usar el pathfinding de Mirrorbot hubo que comprender qué clases utilizaba y cómo las llamaba a la hora de crear el bot.

Pogamut viene con una serie de clases predefinidas de las que se puede hacer uso sin necesidad de definir. Una de estas clases predefinidas se utiliza para el pathfinding las cuales son [Bida, M. y otros, 2012]: *LoqueNavigator*, *UT2004PathExecutor*, *KeifikRunner* y *UT2004Navigation*.

LoqueNavigator hereda de la clase abstracta *AbstractUT2004PathNavigator* la cual implementa la interfaz *IUT2004PathNavigator*. Esta clase se encarga de mover al bot hasta su destino a través del camino pre-computado, evitando obstáculos, abriendo puertas, esperando ascensores, saltar sobre obstáculos, etc. Esta clase se "sustituyó" por la clase *MyNavigator* implementada en Mirrorbot la cual sólo tuvo que heredar de *AbstractUT2004PathNavigator* para poder ser reconocida como una implementación propia y válida para Pogamut.

UT2004PathExecutor se encarga de manejar el estado actual en el que se encuentra la ejecución. Implementa la interfaz *IUT2004PathNavigator*. La redefinición que hace Mirrorbot de esta clase es *MyPathExecutor* la cual hereda de *UT2004PathExecutor* e implementa la interfaz *IUnrealPathExecutor*.

KeifikRunner es la redefinición de *LoqueRunner* la cual se usaba en versiones anteriores de Pogamut como clase por defecto. Esta clase se encarga de llevar al bot hasta su destino en el menor tiempo posible y evitando obstáculos y

colisiones. Implementa la interfaz *IUT2004PathRunner*. La redefinición de Mirrorbot la realiza en la clase *MyRunner* la cual implementa también la interfaz *IUT2004PathRunner*.

UT2004Navigation implementa la interfaz *IUT2004Navigation* y se encarga de todo el sistema de navegación de UT2004. Para manejar todos los posibles casos de navegación, hace uso de las clases *IUT2004PathExecutor*, *FloydWarshallMap*, *IUT2004RunStraight* y *IUT2004GetBackToNavGraph*.

Una vez que hemos visto las clases encargadas de la navegación del bot, vamos a ver cómo llamarlas.

La clase *UT2004BotModuleController* de Pogamut es la que se encarga del instanciamiento y control de un bot para UT2004. Para ello viene con una serie de funciones y procedimientos definidos los cuales se pueden redefinir sin problemas. La clase T800 de Nizorbot hereda de esta clase (al igual que la clase *Mirrorbot4* de Mirrorbot) y redefine muchas de estas funciones. Entre las más importantes destacamos el procedimiento Logic pues es en este procedimiento es donde se va a ejecutar todo el comportamiento del bot. Pero, para inicializar el módulo de navegación del bot se hace uso del procedimiento *initializePathFinding*. En este procedimiento se inicializarán las clases *MyNavigator* y *MyPathExecutor* las cuales se las pasará a la clase *UT2004Navigation* que viene por defecto y de esta manera ya hemos redefinido el pathfinding.

```
protected void initializePathFinding(UT2004Bot bot)
{
            //System.out.println("SUCCESS HACK");
            java.util.logging.Logger myLog = java.util.logging.Logger.getAnonymousLogger();
            if (myLog == null)
            {
                    myLog = log;
            }
            else
            {
                    myLog.setLevel(Level.OFF);
            }

            pathPlanner  = new UT2004AStarPathPlanner(bot);
            fwMap        = new FloydWarshallMap(bot);
            myNav        = new MyNavigator(bot, myLog);
            pathExecutor = new MyPathExecutor<ILocated>(bot, myNav);

            pathExecutor.addStuckDetector(new UT2004TimeStuckDetector(bot, 3000, 100000)); // if the bot
            pathExecutor.addStuckDetector(new UT2004PositionStuckDetector(bot));           // watch over
            pathExecutor.addStuckDetector(new UT2004DistanceStuckDetector(bot));           // watch over

            getBackToNavGraph = new UT2004GetBackToNavGraph(bot, info, move);
            runStraight = new UT2004RunStraight(bot, info, move);
            navigation = new UT2004Navigation(bot, pathExecutor, fwMap, getBackToNavGraph, runStraight);
}
```

Figura 4.1. Inicialización pathfinding.

La Figura 4.1. muestra el código que inicializa el pathfinding del bot. Este procedimiento hubo que incluirlo en la clase *T800* de Nizorbot ya que en la versión

original no hacía uso de este procedimiento. Además, en el archivo POM de Nizorbot hubo que incluir el proyecto de Mirrorbot para poder hacer uso de las clases `MyNavigator` y `MyPathExecutor` y de esta manera modificar lo menos posible el proyecto de Nizorbot.

Otra modificación sobre esta clase fue eliminar el procedimiento `initializeRayCasting` ya que en este procedimiento se definió un raycasting propio. Fue sustituido por el raycasting que hace Mirrorbot en la clase `RayData` la cual hace uso la clase `MyRunner`. Para Nizorbot se definió 8 *rays* que coinciden con todos los puntos cardinales, mientras que en Mirrorbot se definió 24 *rays*, 16 horizontales y 8 verticales. Dentro de los 16 *rays* horizontales se incluyen los 8 puntos cardinales y hubo que averiguar cuáles de los 16 rays coincidían con cada punto cardinal que definió Nizorbot. Para averiguarlo se comprobó los valores de los *rays* que ponían cada uno y las equivalencias de dirección de los *rays*. Las equivalencias fueron las siguientes:

Dirección del ray	Valor de Nizorbot	Valor de Mirrorbot
Norte	(1,0,0)	(0.995, 0,0)
Noreste	(1,1,0)	(0.703, 0.703,0)
Este	(0,1,0)	(0, 0.995,0)
Sureste	(-1,10)	(-0.703, 0.703,0)
Sur	(-1,0,0)	(-0.995,0, 0)
Suroeste	(-1,-1,0)	(-0.703, -0.703,0)
Oeste	(0,-1,0)	(0, -0.995,0)
Noroeste	(1,-1,0)	(0.703, -0.703, 0)

Esta definición de raycasting la realiza la clase `RayData` en su constructor. Esta clase se encarga del control de raycasting, controla todas las colisiones de todos los *rays*. La clase `MyRunner` hace un uso interno de esta clase y `MyNavigator` hace un uso interno de la clase `MyRunner`. Pero estos usos internos son sólo para el módulo de pathfinding de Mirrorbot y Nizorbot hace un uso interno de su propio raycasting en las clases que definen los estados `DefensiveProfile`, `Attack` y `SecondaryState`. En estas clases, originalmente, no se hace uso de la clase `RayData` para el control del raycasting si no que se usa el control interno que hace Pogamut y este control lo realiza a través de la variable raycasting de la clase `UT2004BotModuleController`.

Esta variable almacena todos los *rays* que se utilizarán para el raycasting así como informar del estado de colisión de cada *ray*. Es decir, para Nizorbot, en la función `initializeRayCasting` añadía a la variable raycasting cada *ray* que se puede ver en la tabla de arriba.

Originalmente, para que pudieran hacer uso de esta variable se les pasaba como argumento en sus constructores. Se podría pensar que hubo que modificar

estos constructores incluyendo una instancia de la clase `RayData`, pero no hizo falta hacerlo ya que la clase `RayData` hace uso de esta variable y la inicializa en su constructor, lo único que cambia es el trato interno de las colisiones de los *rays*. Esto quiere decir que en las clases que hagan uso de la variable raycasting, ya tendrán los valores que define la clase `RayData`. Lo que hubo que modificar fue el acceso a estos *rays* ya que se hace a partir del nombre asociado a cada *ray* y Mirrorbot le da unos nombres diferentes a los de Nizorbot. Por lo que hubo que cambiar estos nombres de acceso en las clases *`DefensiveProfile`*, *`Attack`* y *`SecondaryState`*. Además, en la clase abstracta *`SecondaryState`*, de la cual heredan todas las clases que definen cada estado secundario de Nizorbot, no almacenaba la variable raycasting en una variable a la que pudieran acceder sus hijos, por lo que hubo que hacerla accesible a estos. El acceso original de estos hijos se realizaba a través de un array de *rays* que define Nizorbot y el cual se le pasa como argumento a todos los estados de Nizorbot. La inicialización de los estados se realiza en el procedimiento *`prepareBot`*.

4.3. Mirzorbot

Esta hibridación contiene todos los puntos que se han considerado fuertes de Mirrorbot y Nizorbot. Estos puntos fuertes se han elegido en base a la observación del comportamiento de los bots en ejecución, las consideraciones que se dieron en la Botprize de 2014 (ambos participaron quedando Mirrorbot en primer lugar y Nizorbot en segundo) y, en los comportamientos comunes cómo la comparación de los arsenales propio y del enemigo, por la mejor implementación.

Los puntos de fuertes de Nizorbot se consideran que son: las máquinas de estados finitos jerárquica con las que funciona y la elección del arma.

Los puntos fuertes de Mirrorbot se consideran que son: el módulo *mirror*, el pathfinding, la elección de hacia dónde ir y hacia dónde mirar y el disparo. Por lo tanto, el esquema de comportamiento de Mirzorbot sería:

1) Elección de destino del bot y la dirección de su mirada. Comportamiento de Mirrorbot.

2) Comprobar si está imitando:

a. Si está imitando no hace nada más que imitar hasta que muera o pare de imitar. Comportamiento de Mirrorbot.

b. En caso de no imitar determina los estados primarios y secundarios, comparará los arsenales, ejecutará los estados y disparará si fuera necesario.

Este comportamiento es íntegramente el de Nizorbot. Esta hibridación se ha conseguido uniendo funciones y procedimientos de ambas implementaciones, eliminando sentencias que fueran contradictorias y modificando ciertos usos de clases

de Pogamut debido a la diferencia de versión que usa uno y otro. En el caso de Nizorbot usa una versión más nueva que Mirrorbot.

La base del proyecto que contiene a Mirzorbot es de Mirrorbot por lo que el POM es de Mirrorbot y se le ha añadido aquellas líneas de dependencias que necesita Nizorbot para poder ser construido. Además, se han añadido las carpetas de Nizorbot que contienen los estados y las clases auxiliares con las que funcionan los estados.

La clase Mirzorbot es la que ejecuta todo el comportamiento descrito anteriormente. Esta clase, al igual que la clase *T800* de Nizorbot, hereda de la clase `UT2004ModuleController` y hace uso de los mismos procedimientos que Mirrorbot y Nizorbot, es más, esta clase es una mezcla de las clases *T800* y Mirrorbot de Nizorbot y Mirrorbot respectivamente. Esto es porque tanto Nizorbot como Mirrorbot redefinen algunos procedimientos de la clase padre.

Antes de ver la construcción de la clase Mirzorbot veamos primero las modificaciones que hubo que realizar en los estados de Nizorbot, los cambios en el módulo de pathfinding de Mirrorbot y las modificaciones de la clase brain de Mirrorbot.

Las modificaciones en los estados de Nizorbot surgen por el uso del pahtfinding de Mirrorbot y por el sistema de disparo y *Mirroring* de Mirrorbot. Por ello, las modificaciones que se han hecho para acoplar el pahtfinding de Mirrorbot han sido las mismas que para las que hubo que hacer en Mizorbot. Las nuevas modificaciones relacionadas con el sistema de disparo y *Mirroring* de Mirrorbot sólo afectan al estado `PrimaryState`.

Este estado es donde se ejecuta el sistema de disparo de Nizorbot y por lo tanto donde se tendrá que ejecutar el sistema de disparo de Mirrorbot. El procedimiento que se encarga de este sistema de disparo es `engage` y recibe como parámetros un objeto de tipo `Player` (es el enemigo) y un `float` indicando el tiempo transcurrido entre frames (necesario para el sistema *Mirroring*).

El contenido de este procedimiento fue sustituido por completo por el sistema de disparo de Mirrorbot que se encontraba en la clase `Brain` de Mirrorbot en el procedimiento `doBehaviour`. Con esta sustitución, el bot estaría listo para ejecutar el sistema de disparo y para activar el módulo Mirroring de Mirrorbot si se cumplen las condiciones para ello.

Debido a que las versiones de Pogamut con las que trabajan Nizorbot y Mirrorbot son diferentes, hubo que modificar la clase `MyRunner` de Mirrorbot. Las modificaciones no fueron significativas sólo que la manera de modificar los objetos de tipo `Location` eran diferentes en las versiones de Pogamut de Mirrorbot y Nizorbot. Por lo que, cada vez que se quería modificar los valores de un objeto `Location`, se reconstruía el objeto pasando como parámetros al constructor los nuevos valores de x, y, z.

La clase *brain* de Mirzorbot unifica las clases *Skynet* y *Brain* de Nizorbot y Mirrorbot respectivamente. No las unifica en su totalidad, sino que unifica aquellas partes necesarias e importantes para conseguir un comportamiento unificado de Mirrorbot y Nizorbot. Veamos cada modificación que hubo que hacer.

En primer lugar, hubo que añadir tres variables privadas de tipo *Individual*, *Location* y booleano denominadas *testIndividual*, *spamLocation* y '*entra*' respectivamente. La primera variable contiene el cromosoma de Nizorbot y la segunda y tercera las veremos más adelante.

El constructor de la clase hubo que modificarlo añadiendo un parámetro de entrada de tipo *Individual* el cual le dará valor a la variable *testIndividual*. Por lo demás es el mismo constructor.

El procedimiento *execute* hubo que cambiarlo a una función que devuelve un array de tipo *Location*. Además de añadir una serie de parámetros necesarios para el comportamiento de Nizorbot. Esta modificación se debe a que ahora esta función determinará a dónde moverse y dónde mirar, pero no ejecuta ni el movimiento ni la rotación del bot. Además, se hace uso de la variable *entra* la cual nos sirve para resolver un problema que surge al unificar Mirrorbot y Nizorbot. Este problema es el siguiente, originalmente en esta función se determina si Mirrorbot tiene que imitar o tiene que ejecutar su comportamiento base. Por ello, la función se divide en dos partes:

1. Si está imitando sigue imitando hasta que finalice el tiempo de imitación.
2. Y, en caso de no imitar, ejecuta su comportamiento base.

Pero, cuando está imitando no determina ni la posición a la que desplazarse ni la dirección hacia la que mirar, por lo que, si entra en la parte de imitar y determina que tiene que dejar de imitar, no determina una posición a la que moverse ni la dirección a la que mirar y cuando llegue a ejecutar la parte de Nizorbot, la cual se encargará de ejecutar el movimiento y la rotación del personaje (porque no está imitando), saltará una excepción de tipo *NullPointerException*. Por lo que, esta variable tiene un valor false hasta que empieza a imitar la cual se pondrá a true y antes de devolver un valor de posición y dirección se preguntará si ya no está imitando y si entró a determinar si seguía imitando o no. Si se da el caso en el que entró a imitar y determinó que deja de imitar, se cumpliría la condición de fallo y nos aseguramos que se devuelve un array no nulo.

El procedimiento *doBehaviour* de Mirrorbot es dónde se ejecuta su comportamiento base, es decir, dónde determina hacia dónde ir, mirar y disparar en caso de necesitarlo y es ejecutado en el procedimiento *execute* original de Mirrorbot. En el caso de Mirzorbot, este procedimiento se ha convertido a una función que devuelve un array de tipo *Location* el cual contendrá hacia donde ir y hacia dónde mirar. Es ejecutado en la función *execute* y tiene la misma estructura que el original con la diferencia de que ya no ejecuta el desplazamiento ni la rotación del bot, además

de no ejecutar el sistema de disparo de Mirrorbot (de esto ya se encarga el estado *PrimaryState* y *SecondaryState* de Nizorbot).

Cómo esta clase aúna todas las funciones necesarias para ejecutar los comportamientos de Mirrorbot y Nizorbot, hubo que añadir una serie de funciones de Nizorbot las cuales son:

- ***compareArsenals***: da un valor de lo bueno que es arsenal del bot con respecto al enemigo.
- ***estimateProfile***: determina el estado secundario del bot en base a una serie de riesgos que puede tomar el bot para vencer al enemigo.
- ***behave***: determina los estados primarios y secundarios del bot (haciendo uso de las funciones *compareArsenals* y *estimateProfile*).
- ***incomingProjectile***: determina si la posición desde la que lanzaron un proyectil actualmente es falsa o no. En caso de no serlo, rompemos un combo de ataque que se puede hacer con el *ShockRifle*.

Además, hubo que añadir funciones *getter* y *setter* de las variables *MirrorModule*, *ShootingData* y *PlayerData* y una función *resetTempInfo* necesaria para Nizorbot que resetea información temporal en cada frame.

Una vez hemos visto todas estas modificaciones veamos ahora la construcción de la clase Mirzorbot la cual ejecutará todo el comportamiento y hará uso de las modificaciones descritas anteriormente.

En primer lugar, las variables locales de esta clase es una unificación de las variables locales de Mirrorbot y Nizorbot necesarias para su ejecución.

En la función *PrepareBot* que nos sirve para inicializar las variables locales, se ha unificado la implementación de esta función de Mirrorbot y Nizorbot.

La función *getInitializeCommand* es íntegramente la implementación de Mirrorbot. Se eligió esta implementación porque la que realiza Nizorbot está más orientado para un algoritmo genético, esto se debe a que aquí se pone un nombre de juego al bot y en el algoritmo genético de Nizorbot, el nombre correspondía con la generación actual en la que se encontraba el algoritmo. Mientras que Mirrorbot lo único que hace es poner un nombre cualquiera al bot.

La función *botInitialized* es una unificación integra de las implementaciones que realizan Mirrorbot y Nizorbot con la excepción de que no se llama a la función *initializeRayCasting* de Nizorbot ya que la inicialización del raycasting la realiza Mirrorbot en la clase *RayData* usada en la clase *MyRunner*, la cual ha sido inicializada en la función *botInitialized*.

La función *botFirstSpawn* es íntegramente la implementación de Nizorbot ya que el único que la implementa.

La función *Logic* es la función que ejecuta todo el comportamiento de Mirzorbot, la hará uso de la clase *brain* y dónde se ejecutarán los diferentes estados del bot y el módulo Mirror. Esta función se estructura de la siguiente manera:

- En primer lugar, determinamos el tiempo que ha transcurrido entre frames.
- En segundo lugar, ejecutamos la función *execute* de *brain* la cual nos determinará si empieza a imitar o si determina hacia dónde ir y mirar.
- En tercer lugar, preguntamos si estamos imitando.
  - En caso de estar imitando, no se hace nada más.
  - Si se ha dejado de imitar, entramos en el comportamiento de Nizorbot. Este comportamiento determina los estados primarios y secundarios, cambiará a la mejor arma disponible, ejecutará el desplazamiento hacia la posición determinada por `execute` así como la dirección a la que mirar, disparará si tuviera que hacerlo (haciendo uso del sistema de disparo de Mirrorbot) y reseteará la información temporal. Es el mismo orden de ejecución que en Nizorbot con la diferencia que la clase que determina los estados es `brain` en lugar de la clase `Skynet` de Nizorbot.

La función *botKilled* es una unificación de las implementaciones de Mirrorbot y Nizorbot.

La función `main` es dónde se ejecuta el bot y dónde se conectará a la partida de UT2004. Tanto Nizorbot como Mirrorbot tienen su propia implementación, pero se ha escogido íntegramente la implementación de Mirrorbot porque, por un lado, daba menos problemas a la hora de conectarse y, por otro lado, porque mientras se esté ejecutando intentará conectarse a una partida mientras que Nizorbot si a la primera no lo consigue termina su ejecución.

Por último, la función *initializePathFinding* necesaria para poder inicializar todo el pathfinding de Mirrorbot. Además, hay una serie funciones que hace uso Nizorbot pero Mirrorbot no las cuales le da un comportamiento más completo a Mirzorbot.

4.4. Mirrorbot-Evo

Este bot es el resultado de aplicar un algoritmo genético a Mirrorbot con el fin de mejorar su actitud ofensiva mientras no está imitando y de mejorar las condiciones que se tienen que dar para que imite.

El comportamiento de Mirrorbot se estudió realizando partidas contra él o poniéndolo a jugar contra otros bots y se llegó a la conclusión de que su comportamiento ofensivo mientras no imita era demasiado poco ofensivo. Esto es, no dispara, al contrario, aunque lo tuviera delante y le disparara, se quedaba quieto mirando al contrario o se movía alrededor sin hacer nada.

Por estos motivos se estudió cuáles eran los parámetros usados para determinar su comportamiento ofensivo y una vez localizados y estudiados optimizarlos.

Por otro lado, se estudió los parámetros utilizados para determinar si debe imitar al contrario o no ya que se consideró que esperaba demasiado para imitar al contrario.

Una vez estudiados los parámetros usados para ambos fines se definió el siguiente cromosoma:

- **Gen1:** tiempo que está imitando. Rango de valores (en milisegundos) [1000, 10000], valor original 8136.
- **Gen2:** tiempo que considera que se ha perdido de vista al objetivo. Rango de valores (en milisegundos) [1000, 9000], valor original 2982.
- **Gen3:** límite de violencia, es las veces que ve a un jugador en un estado de violencia. Si se sobre pasa el valor de este gen, descarta al individuo para ser imitado (considerando que es un bot y no un humano). Rango de valores [1, 10], valor original 7.
- **Gen4:** delay para imitar. Rango de valores (en milisegundos) [0, 500], valor original 115.
- **Gen5:** último momento en el que se vio en movimiento al objetivo a ser imitado. Rango de valores (en milisegundos) [0, 1000], valor original 842.
- **Gen6:** votos hacia un candidato para ser imitado. Si el objetivo alcanza este valor, se pasa a imitarlo. Rango de valores [1, 7], valor original 5.
- **Gen7:** la distancia media entre el bot y el jugador candidato. Para poder observar el comportamiento del candidato sin ser atacado por este. Rango de valores [1200, 2000], valor original 2000.
- **Gen8:** tiempo que el bot mantiene a un jugador como archienemigo. Esto significa que en el momento en el que lo vea le atacará. Rango de valores (en milisegundos) [1000, 9000], valor original 8211.
- **Gen 9:** tiempo el cual si se sobrepasa se limpia la lista de archienemigos. Rango de valores (en milisegundos) [1000, 5000], valor original 2142.
- **Gen 10:** tiempo en el cual dispara sin control, tiroteando. Rango de valores (en milisegundos) [100, 500], valor original 300.
- **Gen 11:** distancia larga entre el bot y el jugador objetivo. Rango de valores [500, 5000], valor original 3500.
- **Gen 12:** distancia corta entre el bot y el jugador objetivo. Rango de valores [100, 800], valor original 600.

Tras definir el gen se planteó un algoritmo genético estacionario, ya que, en los problemas ruidosos como este, los estacionarios hacen menos exploración y dan menos variedad a la población. Este hecho hace que se elimine un poco de ruido y la evolución coja una dirección a seguir.

Este algoritmo es el siguiente:

- El tamaño de la población es de 30 individuos.
- La función fitness usada es: f(x) = (DañoRealizado – DañoRecibido/10) + (Muertes*50 – VecesMuerto*5)
- Se evalúa la generación y se escogen dos padres al azar. Para ello, se escogen primero dos padres aleatorios de los cuales nos quedamos con el que tenga mejor valor fitness y hacemos lo mismo para el segundo padre.
- Una vez tenemos los padres creamos el hijo cogiendo el valor de los genes de los padres, es decir, por cada gen hay un 50% de coger el valor del gen del mejor padre (de 0 a 49 cogemos al mejor padre, de 50 a 100 al peor).
- Una vez tenemos creado al hijo, hay una probabilidad por cada gen de 1/12 (teniendo todos la misma probabilidad) de que un gen mute. Si se diera la probabilidad de que un gen mute, cogería un valor aleatorio del rango de valores que puede coger ese gen, es decir, si ese gen coge valores entre 100 y 1000 y muta, tendría un nuevo valor aleatorio perteneciente a ese rango.
- Una vez el hijo ha pasado las fases de creación y mutación, se le evalúa. Si tiene un valor superior al peor gen de la generación (previamente habremos ordenado la generación de mejor a peor valor fitness) sustituye a ese individuo y se reordena la generación de nuevo de mejor a peor valor fitness.

El algoritmo se desarrolla en las clases `EvolutionClass` y `ClaseEjecutora`. La primera clase se encarga de ejecutar el orden del algoritmo y la segunda clase se encarga de ejecutar cada paso del algoritmo. Es decir, `EvolutionClass` ejecuta las funciones de `ClaseEjecutora` en el orden descrito anteriormente y `ClaseEjecutora` es la que se encarga de reordenar la población, crear al hijo, escoger a los padres, ejecutar las partidas, guardar la generación en un archivo xml al generar una población nueva y cargar una población desde un archivo xml.

La información que se guarda de cada individuo es: los valores de sus genes, la media de su fitness hasta el momento, el daño hecho, daño recibido, número de muertes causadas, número de muertes propias y número de evaluaciones a las que ha sido sometido. Se guarda aquella información que es relevante para el cálculo del fitness y para valorar la calidad del individuo.

Para ejecutar las partidas sin gráficos se reutilizó las librerías usadas en el proyecto `EvoGUI`, proyecto que se desarrolló para evolucionar Nizorbot.

Las modificaciones han sido las mismas en todos los casos: añadir una variable de tipo `Individual` a cada clase y, donde antes estaba el valor original del parámetro poner el valor del gen. Por lo que todos los constructores fueron modificados para recibir un parámetro de tipo `Individual`.

La única clase que ha recibido más modificaciones ha sido la clase *Mirzorbot*. Esto se debe a que el algoritmo genético ejecuta individuos de tipo *Mirzorbot* al que

les pasa un valor de cromosoma cuando ejecuta las partidas, esto es, pasarle una clase de tipo `Individual` que le de valor a su cromosoma. Este valor se da por medio de un socket de conexión, por lo que se cogió la misma función que usaba Nizorbot para poder obtener el valor del cromosoma mediante un *socket* y se introdujo en la clase *Mirzorbot*.

En la Figura 4.2. se muestra la función usada para leer los datos del individuo recibidos a través de un Socket.

```
private void cargarIndividual(){
    try {
        //Get current individual ID from server
        workClient = new WorkQueueClient(4000);
        if (workClient != null) {
            synchro.SyncMessage id = workClient.readMessage(null);
            testIndividual = (IndividualV1) id.getData();
            //   id.setStatus(Job.Estado.Running);
            workClient.sendMessage(id);
            number = id.getId();

        }
    } catch (IOException ex) {
        throw new PogamutException("No DB server found", ex);
    } catch (ClassNotFoundException ex) {
    }
}
```

Figura 4.2. Función de lectura de socket

Este procedimiento es llamado en el procedimiento `botInitilized` y el resultado de leer la información del *socket* se lo dará a la clase `Brain` y de esta forma ya tenemos un individuo cargado para poder ser evaluado.

Una vez que se terminó de ejecutar el algoritmo y obtuvimos un valor definitivo de cromosoma, ya no hacía falta hacer uso de esta función pues cuando se va a jugar una partida normal, el bot no recibe ningún de valor de cromosoma, por lo que el valor del cromosoma de obtiene de un archivo xml. Por lo tanto, donde antes llamábamos a la función de lectura del socket ahora ejecutamos una lectura de archivo xml e igual que antes al cargar el valor del cromosoma, se le pasa a la clase `Brain`.

Por otro lado, al hacer uso de las librerías de evolución que se desarrollaron para el proyecto Nizorbot, hubo que modificar la clase `IndividualV1` (la cual es la que define un cromosoma y la función fitness que va a usar). Hubo que modificar los arrays que contienen los valores mínimos y máximos de cada gen poniendo los nuevos valores de cada rango, crear un nuevo constructor que recibe sólo la función fitness a usar y definir una nueva función la cual crea un cromosoma aleatorio con los nuevos rangos de valores para cada gen.

5. Experimentos: Resultados y análisis.

En este apartado veremos, por un lado, los resultados de comportamiento obtenido de cada hibridación y, por otro, el diseño test de Turing así cómo los resultados del mismo y del algoritmo genético ejecutado y se analizarán posteriormente estos resultados intentando extraer conclusiones generales.

5.1. Test de Turing

Una vez se terminaron las fases de desarrollo y testeo de las hibridaciones, se pasó a diseñar un test de Turing el cual nos serviría para comprobar la validez del nivel de humanidad de las hibridaciones.

Por motivos de tiempo y recursos, se planteó hacer un test de Turing a través de una página web en la cual se alojarían una serie de vídeos de partidas en UT2004 y una serie de opciones las cuales permiten poder votar el nivel de humanidad de los jugadores mostrados en los vídeos, permitiendo realizar juicios TPA (Third Person Assessment).

Lo primero que se empezó a hacer fueron los vídeos a mostrar. Para ello se grabaron partidas en las que todos los bots jugaban contra los demás bots (partidas uno contra uno, tanto como jugador como enemigo) y partidas en las que cada bot jugó contra un humano (cómo enemigos, debido a problemas de reproducción de partidas jugándolas en lugar de ser espectador). Las partidas son de dos minutos y con un límite de frags de 1000. Se grabaron tanto vídeos en primera persona como en tercera persona.

Una vez se grabaron todas las partidas se pasó a analizarlas y recortar de cada vídeo las partes más interesantes a mostrar. Estas partes debían cumplir unas series de condiciones para ser recortadas. Estas condiciones consisten en la situación de vida en la que se encuentran cada jugador, el armamento de cada jugador, la situación de los jugadores en el mapa, su movimiento a través de él y la resolución del enfrentamiento.

Una vez se consiguieron todas las partes candidatas a ser mostradas, se pasó a escoger dos vídeos por bot, es decir, cada bot debía aparecer en dos vídeos, ya fuera como jugador o como enemigo, contra bot o contra humano, para poder estar todos los bots en igual de condiciones.

Finalmente, se escogieron siete vídeos los cuales tenían como jugador (personaje en primer plano) y cómo enemigo:

Vídeo	Jugador	Enemigo
Vídeo1	Humano	Mirrorbot-Evo
Vídeo2	Nizorbot	Mizorbot
Vídeo3	Humano	Nizorbot
Vídeo4	Mirzorbot	Mirrorbot-Evo
Vídeo5	Humano	Mirrorbot
Vídeo6	Mizorbot	Mirrorbot
Vídeo7	Humano	Mirzorbot

Los vídeos no duraban más de 20 segundos puesto que si tuvieran una duración más larga, los votantes no terminarían el test y tendríamos posibles resultados erróneos.

5.1.1. Primera versión

Una vez colgados los vídeos en youtube, se creó la página web (http://1-dot-proyecto-tfg.appspot.com/) usando los servicios Google Cloud para alojarla. Esta web tendría una página de inicio en la que se explicaría la finalidad del test, siete páginas las cuales contiene cada una un vídeo y las opciones para votar y una página final de agradecimiento.

En cada página que contenía un vídeo se planteaba la pregunta ¿Quién es el humano?, y para poder responder se mostraban cuatro opciones: ambos, ninguno, pepe y pepito. Pepe siempre era el contrincante y pepito el jugador en primer plano (en el caso de los ´videos en tercera persona, en los vídeos en primera persona era la vista que se mostraba).

Ambos	Ninguno	Pepe	Pepito	Vídeo
18	21	35	54	Video1
15	15	25	50	Video2
22	12	27	28	Video3
24	24	15	20	Video5
26	11	14	26	Video6
12	24	14	25	Video7
22	17	16	16	Video8

Cómo se puede observar en la tabla, los resultados no son fiables puesto que el número de votaciones que obtuvo en total el primer vídeo fueron de 128, mientras que el último vídeo recibió 71 votos. Esto indica que el último vídeo no ha recibido el mismo número de votos y por lo tanto no parten en las mismas condiciones para ser estudiados los resultados. Por lo que se pasó a analizar por qué había personas que no terminaban el test. Se pasó a una fase de testeo más en profundidad y también a preguntar a votantes conocidos por posibles problemas que tuvieran a la hora de votar.

Algunos votantes indicaron que no había botón para volver atrás para cambiar su votación y esto produjo que dieran al botón de volver a la página anterior del navegador enviando de nuevo el formulario anterior y registrando una votación doble en una opción que posiblemente no vaya a volver a votar.

5.1.2. Segundo diseño

Finalmente, como solución al problema se dejó la página de presentación inicial y se enlazó con un formulario de google, el cual nos aseguramos que si votan deben votar todos los vídeos para poder registrar el voto y poder cambiar el voto sin tener que reenviar dos veces un formulario.

Como segundo diseño del test, se añadieron nuevas opciones de votación y un apartado final opcional para poner comentarios. Las opciones nuevas fueron: NS/NC (si no se podía dar una respuesta) e incongruente (si no se entendía lo que se veía en el vídeo).

Una vez diseñado el nuevo test, creado el formulario de google y enlazado con la web, se abrió un segundo y definitivo periodo de votación de un mes y medio el cual transcurrió sin ningún incidente y registrando votaciones totalmente válidas.

5.2. Resultados Mizorbot

El planteamiento principal de esta hibridación fue resolver la carencia que tenía Nizorbot en su algoritmo de pathfinding (es el que trae de base Pogamut). Esto provocaba que tuviera problemas para coger ascensores, subir ciertos tipos de escaleras e incluso de quedarse estancado en una pared.

Decir que estas carencias fueron resueltas sin incidentes teniendo un movimiento por el escenario mucho más fluido y sin estancamientos de ningún tipo. Tras una serie de partidas jugadas contra el bot y jugadas contra otros bots, no se observaron más incidencias de este tipo. Estas partidas de prueba se realizaron en diversos mapas para comprobar la fiabilidad de la solución, en el modo Deathmatch todos contra todos teniendo como contrincantes a Nizorbot, Mirrorbot, Mirzorbot, Mirrorbot-Evolutivo y ExpertBot, sin límite de tiempo y frags y observando las partidas como espectador.

Pero, se sigue teniendo el problema de quedarse quieto en una misma posición mirando hacia el enemigo, aunque no lo tenga a la vista. Decir que este problema no tiene que ver con el algoritmo de pathfinding sino con la toma de decisiones de Nizorbot. En esas mismas series de partidas de testeo de la hibridación se pudo ver este fallo.

5.3. Resultados Mirzorbot

El planteamiento de esto bot fue unificar en un solo bot las mejores partes de Nizorbot y de Mirrorbot. El resultado ha conseguido un comportamiento unificado resolviendo las carencias observadas en ambos bots.

Al igual que en Mizorbot, este bot ya no tiene los problemas de navegación de Nizorbot y, además, ya no tiene el problema de quedarse fijo en un punto del mapa esperando al otro jugador y mirándole sin tenerlo a la vista ya que estas decisiones de hacia dónde ir y mirar las toma Mirrorbot.

Todas estas mejoras se observaron llevando a cabo una serie de partidas de prueba en diversos mapas para comprobar la fiabilidad de la solución, en el modo Deathmatch todos contra todos teniendo como contrincantes a Nizorbot, Mirrorbot, Mizorbot, Mirrorbot-Evolutivo y ExpertBot, sin límite de tiempo y frags y observando las partidas como espectador.

Por otro lado, las decisiones de coger los ítems que necesita en un momento dado el bot, cambiar al arma correcta para resolver un enfrentamiento y el comportamiento ofensivo en ese enfrentamiento, fueron resueltos como carencias que tenía Mirrorbot. Esto se debe a que ahora estas decisiones las toma Nizorbot quién las resuelve de manera eficiente y lógica.

Pero se observa una pequeña carencia en su comportamiento ofensivo que puede dar la sensación de estar jugando contra una máquina. Esto se debe a que el sistema de disparo (que es el que usa Mirrorbot) entra en conflicto con ese aumento de actitud ofensiva. La razón se debe a que ahora el bot es más agresivo, por lo que dispara más y con menos condiciones para hacerlo (que las que necesitaba Mirrorbot para entrar en conflicto) y si a eso le unimos el hecho de que el sistema de disparo está diseñado para tener un grado de fallo, observamos a un jugador bastante agresivo, de continuo disparo, pero que apenas acierta sus disparos. En jugadores humanos es normal que se tengan bajones de rendimiento por cansancio en algunos momentos de la partida. Pero, en el caso de Mirzorbot, esto no sólo se observa en un par de enfrentamientos que se tenga contra él, sino que se observa en cada enfrentamiento en cualquier partida. Es más, en varias ocasiones en las que el jugador se queda quieto sin disparar, sólo observando, Mirzorbot era incapaz de dañar al jugador de forma considerable e incluso pasado un par de minutos era incapaz de matar al jugador.

5.4. Resultados algoritmo genético

Para comenzar con la ejecución del algoritmo se creaba una generación aleatoria (la cual sería la misma para cada ejecución del algoritmo) y a partir de ella se comenzaba la evolución. Por motivos de rendimiento del equipo donde se ejecutó el algoritmo, hubo que realizar un guardado de la generación actual del algoritmo para, en caso de que el equipo dejara de responder, continuar por la última generación realizada del algoritmo y no comenzar desde el principio.

El algoritmo se ejecutaba hasta haber generado 50 generaciones y, a su vez, se hicieron 10 ejecuciones para comprobar la validez del algoritmo. Tras haber ejecutado las diez iteraciones, se escogieron diez candidatos en base al número de generaciones que “sobrevivieron” y tuvieran el fitness más alto de entre los individuos más longevos. Se planteó de esta manera ya que un individuo que haya sobrevivido un número alto de generaciones nos da indicios de que es un buen candidato a tener en cuenta, no ha sido casualidad que haya tenido una o dos partidas buenas.

Una vez escogidos los diez candidatos se creó una generación la cual estaría formada por los diez individuos más longevos más Mirrorbot (este último no es más que un cromosoma con los valores originales de los parámetros a evolucionar). Se ejecutaría de nuevo el algoritmo hasta tener diez generaciones con la diferencia de que no se crearían nuevas generaciones, si no que jugarían cada individuo diez partidas y se haría una media de sus fitness al jugar las diez partidas. Todo este proceso se planteó para que los candidatos pelearan entre ellos por demostrar quién era el más apto teniendo la media del fitness más alto, indicándonos quién es el más constante en sus valores fitness.

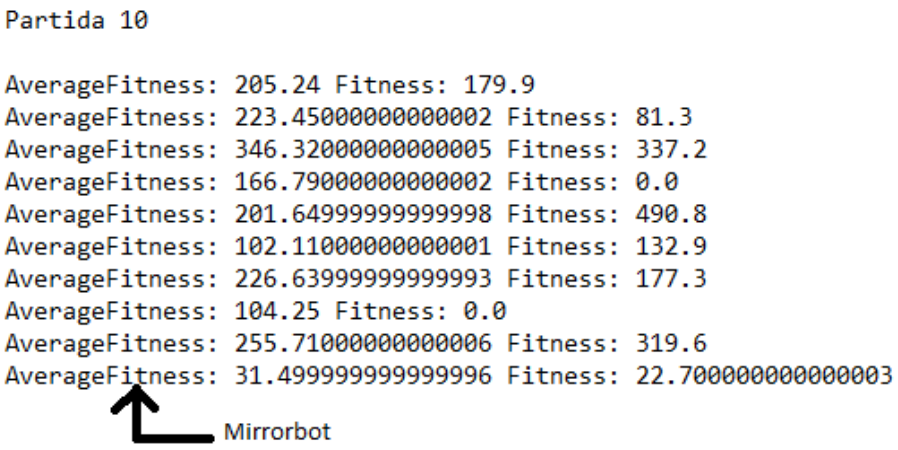


Figura 5.1. Resultados última ejecución de los candidatos

Una vez jugaron las diez partidas y se obtuvieron las medias (Figura 5.1.), se escogió al individuo con la media más alta y se estudió su comportamiento en partidas uno contra uno contra Bot-Experto. Este estudio se centraba en comprobar que tuviera un comportamiento más agresivo en combate que el Mirrorbot original.

Tras el estudio, se consideró que su media era buena pero su comportamiento en partida era demasiado robótico, es decir, era más ofensivo que Mirrorbot pero sus movimientos eran demasiados estáticos y muchas veces no se movía. Por lo que se estudió al siguiente candidato (de la misma manera que el anterior) con la media más alta y su comportamiento no era tan ofensivo como el anterior, pero tenía un buen equilibrio entre movimientos, actitud ofensiva y toma de decisiones lógicas. Por lo que este candidato se escogió como resultado del algoritmo y tiene los siguientes valores:

- **Gen1, valor optimizado 6231, valor original 8136.**
- **Gen2, valor optimizado 5364, valor original 2982.**
- **Gen3, valor optimizado 1, valor original 7.**
- **Gen4, valor optimizado 25, valor original 115.**
- **Gen5, valor optimizado 612, valor original 842.**
- **Gen6, valor optimizado 2, valor original 5.**
- **Gen7, valor optimizado 1560, valor original 2000.**
- **Gen8, valor optimizado 3194, valor original 8211.**
- **Gen9, valor optimizado 2880, valor original 2142.**
- **Gen10, valor optimizado 447, valor original 300.**
- **Gen11, valor optimizado 3398, valor original 3500.**
- **Gen12, valor optimizado 245, valor original 600.**

Por último, la estructura del proyecto es la misma que la de Mirrorbot, mismo POM y estructura de clases. Pero, para poder realizar las evoluciones, hubo que modificar las clases que iban a contener los valores de los parámetros optimizados. Hubo que modificar la clase *`Mirrorbot`* (que ahora se llama Mirzorbot), *`Brain`*, *`MirrorModule`* y *`PlayerData`*.

En el apartado gráficas del anexo se pueden ver las gráficas de evolución de los fitness más altos en cada generación de cada ejecución. A continuación, se mostrará la gráfica en la Figura 5.2. de la ejecución tres como ejemplo:

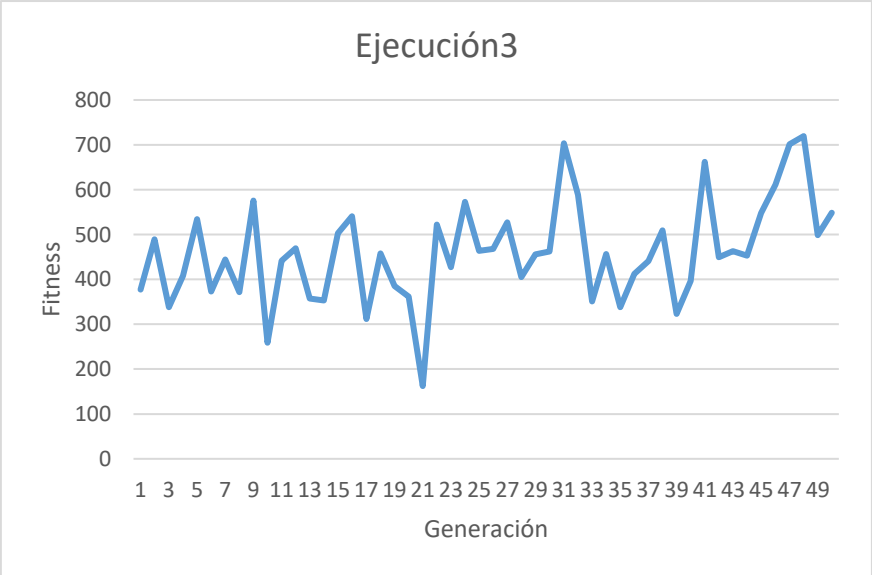


Figura 5.2. Mejores fitness ejecución 3

Como se puede observar en la gráfica los valores fitness tienden a una mejora, aunque sigue habiendo mucho ruido en los valores fitness. Esto se debe mayormente a la naturaleza del problema, ya que estamos hablando de un entorno no determinista.

Pero este hecho se puede atenuar añadiendo parámetros que dirijan la evolución o realizando una evolución interactiva.

En definitiva, estamos buscando un comportamiento más agresivo, pero de forma humana ya que en el caso de Nizorbot éste tiene un comportamiento extremadamente agresivo lo cual lleva a pensar que es un bot, lo que rompe la barrera de la incertidumbre.

A pesar de todo esto, se ha obtenido un resultado cercano a lo que se esperaba conseguir. Se ha obtenido una mejora de agresividad en el bot, pero, por otro lado, se ha tenido que reducir su comportamiento "espejo" para poder ser más agresivo lo que rompe un poco con la idea con la que se creó a Mirrorbot.

5.5. Resultados test de Turing

Este test fue realizado para medir el nivel de humanidad de los bots desarrollados. Las evaluaciones fueron de tipo TPA, es decir, no ha habido un único juez que ha jugado una serie de partidas con los bots y ha dado un voto al finalizar (como pasaba en anteriores ediciones de la Botprize). En este test se han tenido varios jueces que han dado su voto tras observar una serie de partidas jugadas por bots y por humanos (cómo en las últimas ediciones de la Botprize). Lo ideal hubiera sido que cada uno de los jueces que han participado en el test hubiera podido jugar contra cada uno de los bots, pero por problemas de disponibilidad de los jueces y por tiempo no se ha podido hacer. Por este motivo se decidió hacer valoraciones de tipo TPA.

Cómo se mencionó anteriormente, el test fue colocado en una página web a la que podía acceder cualquier persona y la cual estaba tanto en inglés como en español para así abarcar el máximo número posible de participantes. La difusión se llevó a cabo mediante las redes sociales y por difusión oral en algunos casos.

La última versión del test no dio tantos problemas como la primera, es decir, los participantes ya no abandonaban el test a mitad por falta de opciones y, por otro lado, se entendía mejor qué es lo que se estaba pidiendo.

Tras un periodo de votación de un mes y medio y con un total de 61 votantes, se han obtenido los siguientes resultados:

Video	Pepito	Pepe	pepe	pepito	ambos	ninguno	NS/NC	Incongruente
Video1	Humano	MirrorbotEvo	9	25	14	9	2	2
Video2	Nizorbot	Mizorbot	15	29	5	8	1	3
Video3	Humano	Nizorbot	15	30	8	5	2	1
Video4	Mirzorbot	MirrorbotEvo	17	20	13	8	0	3
Video5	Humano	Mirrorbot	14	21	15	8	2	1
Video6	Mizorbot	Mirrorbot	12	20	6	10	7	6
Video7	Humano	Mirzorbot	5	28	13	9	4	2

La fórmula elegida para calcular la humanidad en este test es la siguiente:

$$H = \frac{nV + nA}{122}$$

Dónde **nV** es el número de votos recibido por cada bot, **nA** el número de votos recibidos en la opción 'ambos' en cada vídeo que estuviera el bot y 122 porque al ser 61 votantes y cada bot sale en dos vídeos, puede recibir como máximo 122 votos.

Tras el periodo de votación se aplicó la fórmula a cada bot obteniendo los resultados que se muestran en la siguiente tabla:

Bot	Votos como humano	Nivel humanidad
Nizorbot	57	46,72131148
Mirrorbot	47	38,52459016
MirrorbotEvo	53	43,44262295
Mizorbot	46	37,70491803
Mirzorbot	51	41,80327869

Como se puede observar, el bot que ha obtenido un mayor porcentaje de humanidad ha sido Nizorbot con un 46%. A pesar de haber obtenido un porcentaje mayor que el resto de bots, tanto Nizorbot como los demás bots, no han superado la prueba pues se debe obtener un porcentaje mayor o igual a 50 para poder superar la prueba. Este hecho también sucede en la Botprize, por lo que no hay que tomarlo como algo negativo, al contrario, en la misma Botprize los jugadores humanos no superaban la prueba teniendo incluso peores puntuaciones que los bots. Esto nos demuestra que es una prueba muy difícil de superar y bastante subjetiva de puntuar.

A la vista de los resultados, no se ha conseguido superar el nivel de humanidad de Nizorbot, pero sí a Mirrorbot que ha quedado en penúltima posición y quedando sólo por encima de Mizorbot.

Se puede observar que a pesar de que hay vídeos en primera y tercera persona se ha tendido a votar más al jugador (pepito) que al enemigo (pepe). Se debe a que, tanto si es en primera como en tercera persona, no se puede ver con claridad las acciones del enemigo, por lo que el voto se centra más en lo que se puede ver con claridad. Por este motivo se añadió la opción incongruente para, por si se elegía esta opción más de un 30% de los votos, descartar el vídeo y los votos que hubiera en él.

Por otro lado, se les ha dado más validez a los votos en las opciones de ambos y NS/NC puesto que si se consigue un porcentaje alto de votos en estas opciones significa que el votante ha sido confundido por el bot haciéndole errar con el voto en ambos y con el voto NS/NC no ha sabido diferenciar cuál de los jugadores era el humano y cuál era el bot.

Un detalle a tener en cuenta es el voto 'ninguno' en aquellos vídeos en los que hay un humano, puesto que ha superado en todos los casos al voto NS/NC.

Esto significa que el comportamiento humano puede hacer errar también en las decisiones de otros humanos.

Como se puede observar, el jugador humano siempre estaba en primera persona contra un bot y los vídeos en tercera eran siempre dos bots, esto puede hacer pensar que es fácil descifrar que el humano siempre es el que está en primera persona. Pero, a la vista de los resultados, ha dado igual si es en tercera o primera persona puesto que los votos se han concentrado más en el jugador que en el enemigo y, por otro lado, y más importante, la suma de las opciones ninguno y ambos, en muchos casos, han igualado o superado a los votos que ha recibido el humano del vídeo. Lo cual el bot y el comportamiento del humano han hecho dudar o errar a un número considerable de votantes.

Esto nos lleva a que paralelamente al periodo de votación del test, se hicieron test de Turing con juicios FPA, esto es, al grabar las partidas de humanos contra bot, a los humanos se les hizo dar un voto de humanidad al bot después de haber jugado contra él y se les pidió dar un voto después de ver la grabación de la partida. Los resultados fueron interesantes ya que, en diversos casos, dieron unos votos de humanidad bajos después de haber jugado contra ellos y, tras ver la grabación de la partida, aumentaron el nivel de votación argumentando los propios jugadores que ellos daban más la impresión de bot que el propio bot.

6. Conclusiones y trabajo futuro

En este proyecto se han implementado dos hibridaciones y una optimzación mediante EA's a partir de los bots Mirrorbot y Nizorbot con la intención de conseguir bots con un comportamiento más humano.

A la vista del resultado del test de Turing realizado podríamos decir Nizorbot es el bot con el comportamiento más humano de todos, pero en el test de Turing presencial que se realizó, Nizorbot fue el que peor votación recibió. Por lo que este test ha servido de guía para mejorar ciertos aspectos del test como el formato de los vídeos, los tipos de votos, por ejemplo.

En términos generales, las hibridaciones han servido como guía para futuros abordamientos pues este ha sido un proyecto experimental. Las hibridaciones se han planteado en base al estudio de los comportamientos de Mirrorbot y Nizorbot y se ha intentado crear nuevos comportamientos que cubran sus carencias con las partes positivas y características del otro bot. Por lo que, no se sabía si iban a dar buenos resultados. El hecho de que cada hibridación haya superado el período de depuración y pruebas no nos indica que la hibridación haya sido un éxito puesto que se trataba de medir su nivel de humanidad.

Medir el nivel de humanidad de un bot no es una tarea sencilla puesto que no hay ninguna función matemática que lo haga, al depender ésta de factores subjetivos. Se pueden definir funciones que midan el nivel de agresividad, de pasividad, pero siempre son referidos a: cantidad de daño hecha, recibida, muertes ocasionadas, etc, y el comportamiento humano se mide con valores subjetivos difíciles de medir. Este tipo de funciones nos sirven de guía para simular un comportamiento determinada del ser humano, pero no hay ningún tipo de función que nos indique que un bot tiene un comportamiento humano, dado que ni siquiera el ser humano es capaz de definir cómo es un comportamiento humano.

Una de las conclusiones que podemos sacar tras haber realizado este proyecto es que la inteligencia artificial aplicada a los videojuegos, por ahora, debe centrarse más en conseguir un nivel alto de diversión en la experiencia que tienen los jugadores, pero, entraríamos en otra incógnita difícil de responder y medir: ¿Qué es divertido y cuánto lo es?, ¿Cómo se puede medir la diversión de una persona en tiempo real?

El planteamiento que expone Mirrorbot es una muy buena dirección a seguir pues se basa en el concepto de imitación que tiene el ser humano pero llevado a un extremo abstracto de imitación. Esta abstracción de la imitación la plantea en un sentido más de comportamientos estandarizados en grupos sociales, es decir, las expresiones usadas o el simple hecho de la manera de hablar en ese grupo social. Si hiciéramos un bot que use expresiones de ese grupo social nadie dudaría de que fuera humano. Llevado a otro contexto, si en ese grupo social se anda de una manera determinada o se tiene un saludo característico e hiciéramos un bot que imitara

únicamente la forma de anda y de saludar, pero no hablara, ya sería aceptado por ese grupo social.

Este concepto llevado al entorno de los videojuegos no es más que determinar los comportamientos estandarizados en cada género e imitarlos pues, al ser un comportamiento común en ese género, todos los jugadores pensarían que es un ser humano y habríamos alcanzado el objetivo. Pero, en el entorno de los videojuegos, existen ciertos géneros en los que prioriza más el hecho de divertir antes que tener bots con unos comportamientos humanos. Se puede pensar que al jugar contra humanos ya hay diversión, pero en muchos casos no es así pues la mayoría de las veces los jugadores más expertos buscan retos y jugar contra jugadores humanos menos experimentados que ellos no les produce un reto y no se divierten.

Llegados a este punto un posible trabajo futuro sería el de aplicar el módulo de imitación de Mirrorbot a otros géneros de videojuegos o incluso aplicarlo a entornos diferentes de los videojuegos. Por ejemplo, aplicar este sistema a un entorno de atención al público pues la IA de este sistema podría aprender a empatizar con los clientes aprendiendo mediante la imitación. O aplicar este sistema a un videojuego en el cual hay a un compañero que sigue al jugador y lo imita para poder seguirlo por el escenario.

Por otro lado, en este proyecto no se llevaron a cabo ciertas hibridaciones por falta de tiempo, por lo que se dejaron para trabajos futuros. Estas hibridaciones son:

- Nizorbot, pero con el módulo de imitación de Mirrorbot. Es una hibridación compleja de realizar pues habría que añadir un nuevo estado a su máquina de estado y ver las repercusiones que tiene con respecto a los demás.
- Mirzorbot con el sistema de disparo de Nizorbot atenuado para que no sea tan perfecto.
- Aplicar de nuevo un algoritmo genético a Mirrorbot pero con otro cromosoma, otra función fitness y con alguna función auxiliar que atenúe el ruido de la evolución.

6.1. Aprendizaje personal

Este proyecto me ha aportado mucha experiencia y formación personal pues he aprendido a:

- Manejar y entender Pogamut.
- Usar Maven.
- Primer acercamiento a técnicas de inteligencia artificial avanzadas.
- Programar algoritmos genéticos.
- Implementación de máquinas de estados a un bot experto.
- Crear un test de Turing.

Referencias

[Alan Turing, 2015] http://varinia.es/blog/2015/02/14/quien-fue-que-hizo-alan-turing/ (visto en marzo de 2016)

[Algoritmos genéticos, 2016], http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf, http://www.uv.es/asepuma/X/J24C.pdf (visto en marzo de 2016).

[Bandicamp, 2016] https://es.wikipedia.org/wiki/Bandicam (visto en marzo de 2016)

[Bida, M. y otros, 2012] Bida, M., Cerny, M., Gemrot, J., Brom, C.: Evolution of GameBots project. In: Herrlich, M., Malaka, R., Masuch, M. (eds.) ICEC 2012. LNCS, vol. 7522, pp. 397--400. Springer, Heidelberg, 2012.

[Botprize, 2016] http://botprize.org/ (visto en marzo de 2016).

[Carlos Javier López Turiégano, 2015] http://oa.upm.es/37432/1/TM_LOPEZ_TURIEGANO_CARLOS.pdf (visto en marzo de 2016).

[Charles Darwin y Gillian Beer, 1996] Charles Darwin and Gillian Beer. The origin of species. Oxford University Press,1996.

[Computación evolutiva, 2016] http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeag (visto en marzo de 2016).

[Goal-Oriented Action Planning, 2017] http://alumni.media.mit.edu/~jorkin/goap.html (visto en junio de 2017).

[Inteligencia artificial general, 2013] http://www.it.uc3m.es/jvillena/irc/practicas/13-14/03.pdf (visto en marzo de 2016)

[Inteligencia computacional, 2016] http://computacional.webcindario.com/ (visto en marzo de 2016)

[Java Server Faces, 2016] http://www.jtech.ua.es/j2ee/publico/jsf-2012-13/sesion01-apuntes.html (visto en marzo de 2016).

[José Luis Jiménez López, 2015] Memoria de trabajo de fin de grado “Control de NPCs en Unreal Tournament 2004 mediante computación evolutiva interactiva”, José Luis Jiménez López, Antonio José Fernández Leiva, Antonio Mora García, junio 2015.

[Maven, 2016] https://maven.apache.org/what-is-maven.html (visto en marzo de 2016).

[Mihai Polcenau, 2013] Mihai Polceanu. MirrorBot: Using Human-inspired Mirroring Behavior To Pass A Turing Test. IEEE Conference on Computational Intelligence and Games, Aug 2013, Niagara Falls, Canada. pp.201 |- 208, 2013. <hal-00864153>

[Mihai Polcenau y otros, 2016] Mihai Polceanu, Antonio M. Mora, José L. Jiménez, Cédric Buche, Antonio José Fernández Leiva: The Believability Gene in Virtual Bots. FLAIRS Conference 2016: 346-349 2016.

[Pogamut] Bida, M., Cerny, M., Gemrot, J., Brom, C.: Evolution of GameBots project. In: Herrlich, M., Malaka, R., Masuch, M. (eds.) ICEC 2012. LNCS, vol. 7522, pp. 397--400. Springer, Heidelberg, 2012.

[Pogamut Logue, 2009] Pogamut Loque (J. Gemrot, R. Kadlec, M. Bda, O. Burkert, R. Pbil, J. Havlek, L. Zemk, J. imlovi, R. Vansa, M. tolba, T. Plch, and C. Brom, “Pogamut 3 can assist developers in building ai (not only) for their videogame agents,” in Agents for Games and Simulations, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5920, pp. 1–15.)

[Russell, Stuart J. y Norvig, Peter, 2003] Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentince Hall.

[Russell, Stuart J. y Norvig, Peter, 2003] Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentince Hall

[R. Caballero y otros, 2013] A.M. Mora, F. Aisa, R. Caballero, P. García-Sánchez, J.J. Merelo, P.A. Castillo, and R. Lara-Cabrera. Designing and evolving an unreal tournamenttm 2004 expert bot Springer, pages 312–323, 2013.

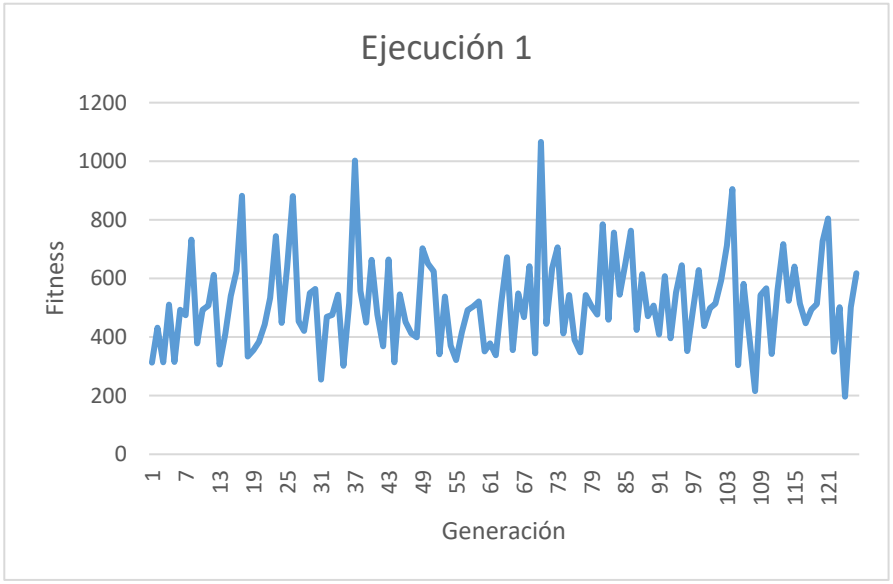
[S. Rosen, y otros, 2002] S. Rosen, L. Musser, and J. Brown, “Reactions of third-graders to recognition allocation after being peer-imitated,” Current Psychology, vol. 21, pp. 319–332, 2002

[UT2004, 2016] https://es.wikipedia.org/wiki/Unreal_Tournament_2004 (visto en marzo de 2016)

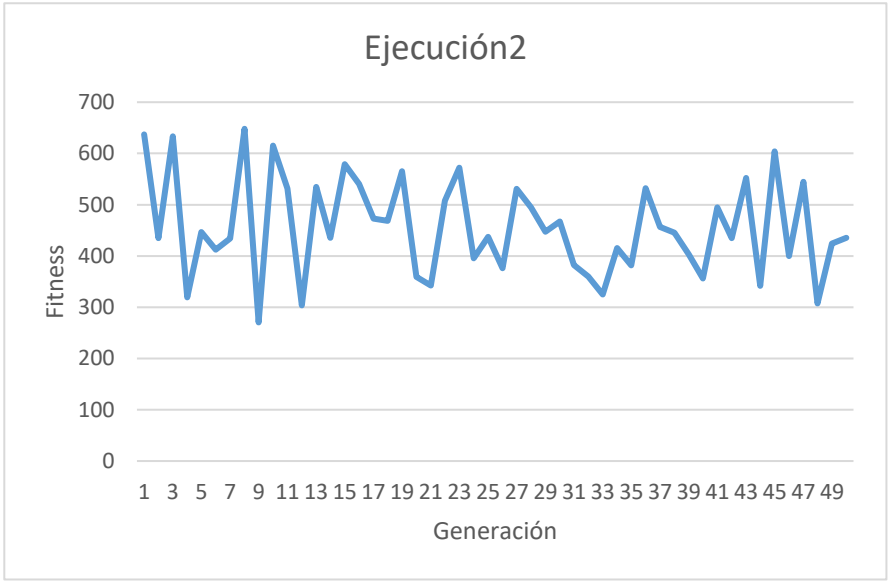
[Windows Movie MAker, 2016] http://movimacker.blogspot.com.es/ (visto en marzo de 2016).

Apéndice

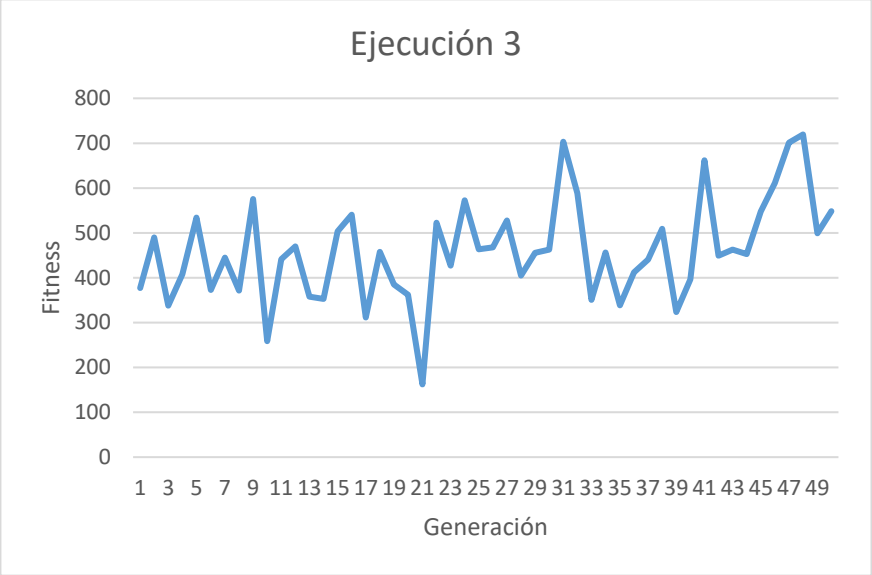
Gráficas



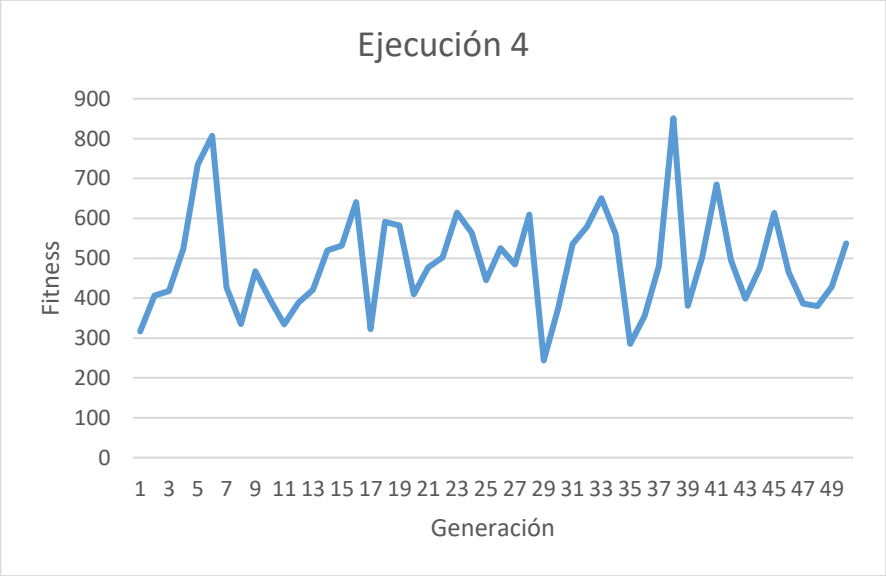
Mejores fitness primera ejecución



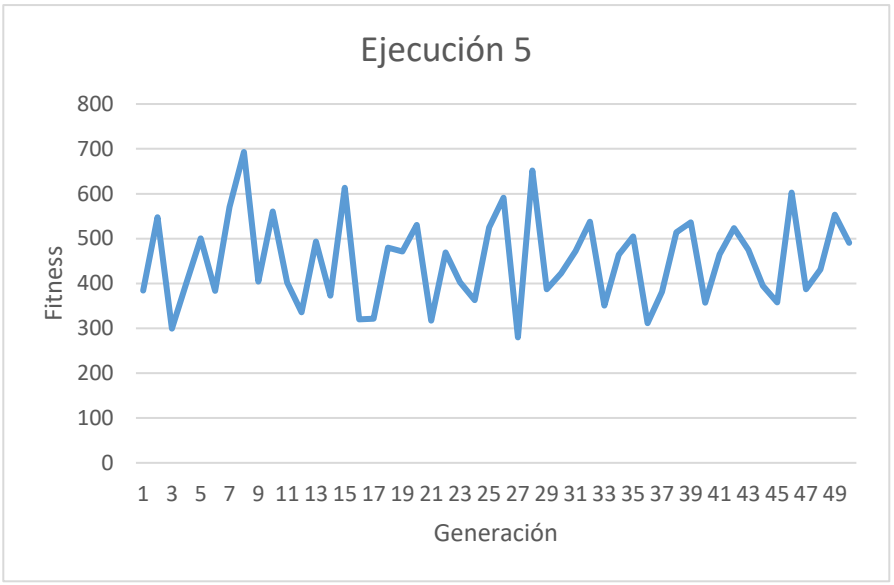
Mejores fitness segunda ejecución



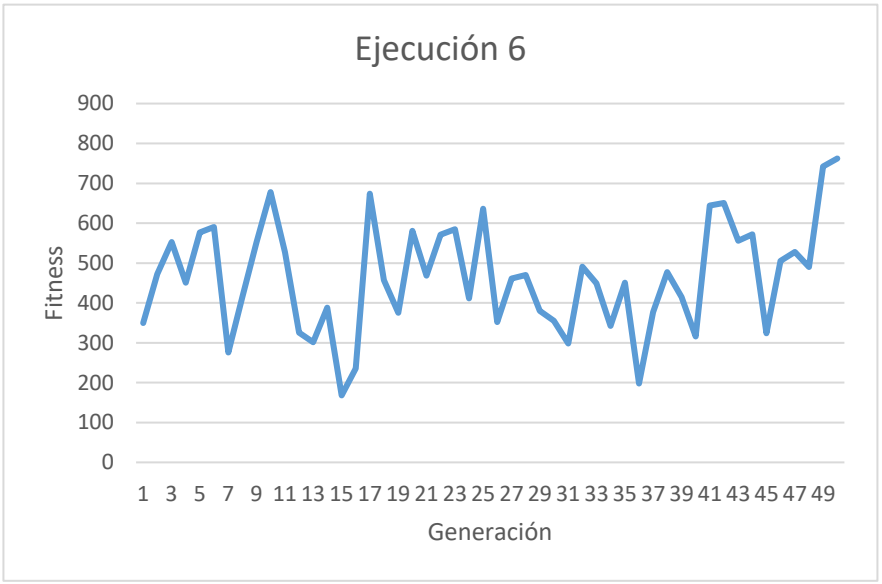
Mejores fitness tercera ejecución



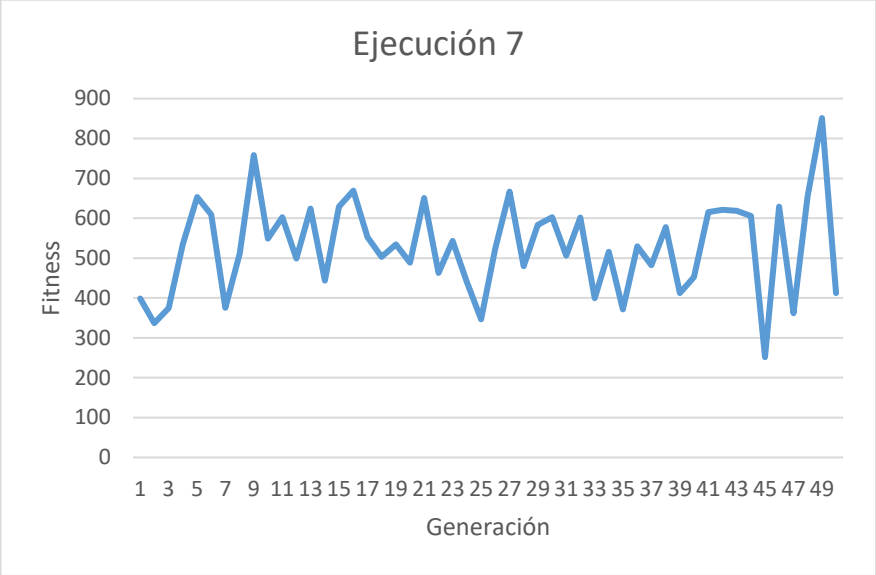
Mejores fitness cuarta generación



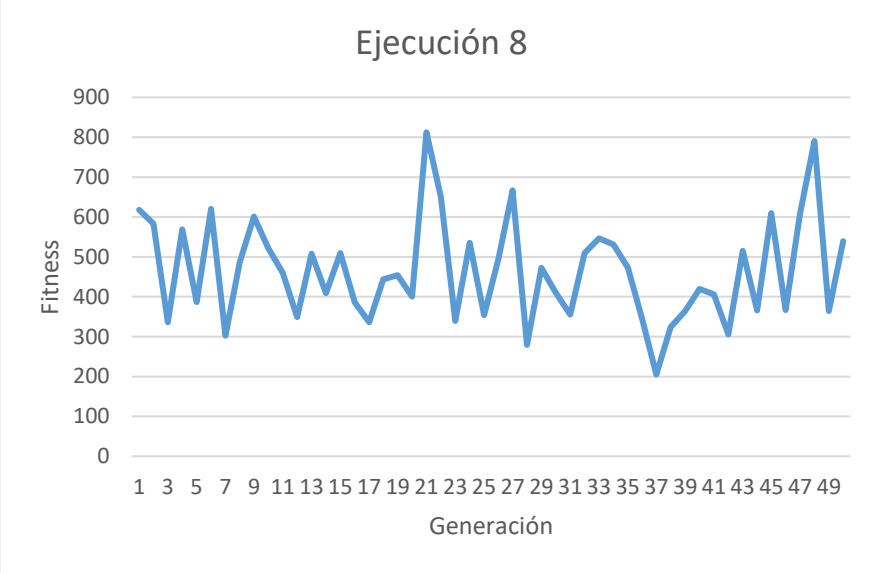
Mejores fitness quinta generación



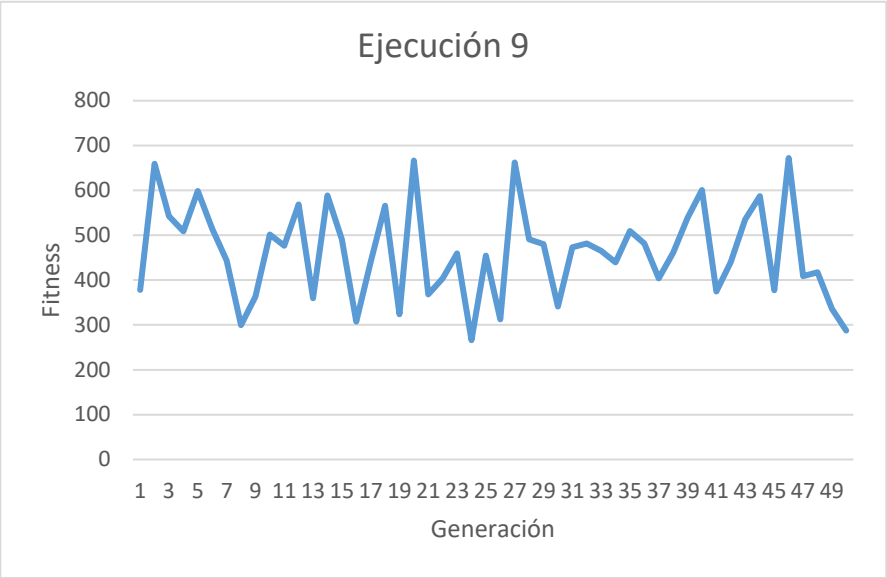
Mejores fitness sexto generación



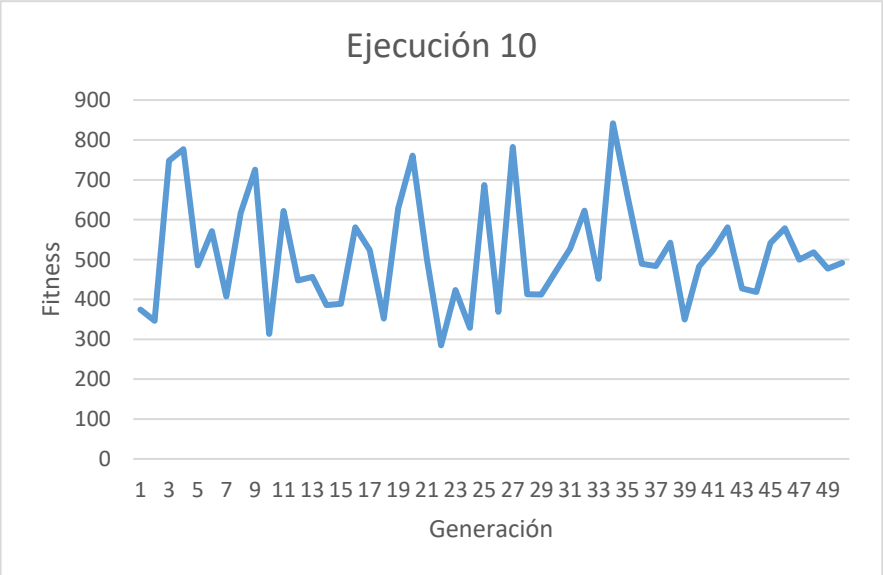
Mejores fitness séptima generación



Mejores fitness octava generación



Mejores fitness novena generación

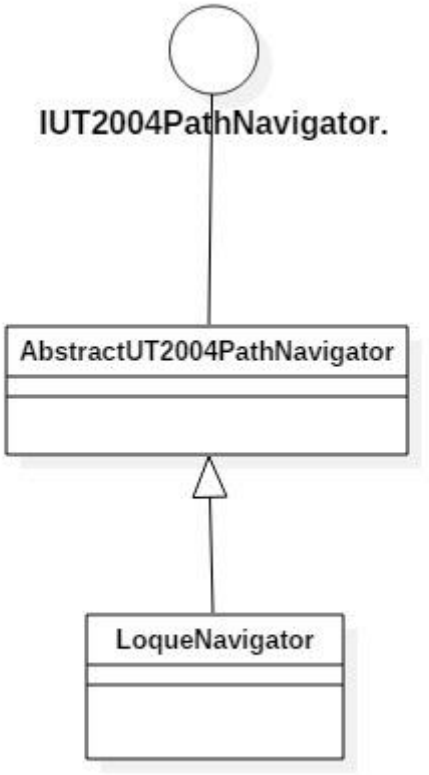


Mejores fitness décima generación

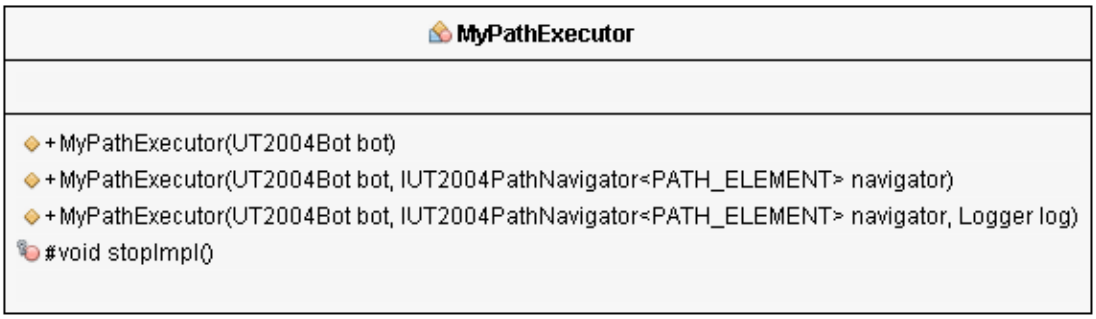
Modelos

Mizorbot

Native Navegation

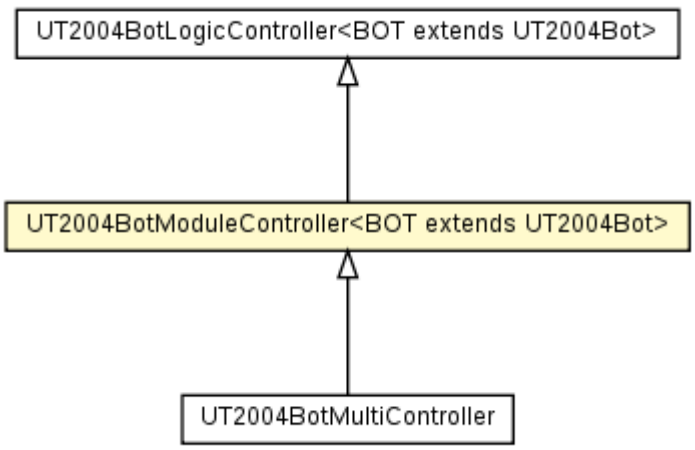


MyPathExecutor

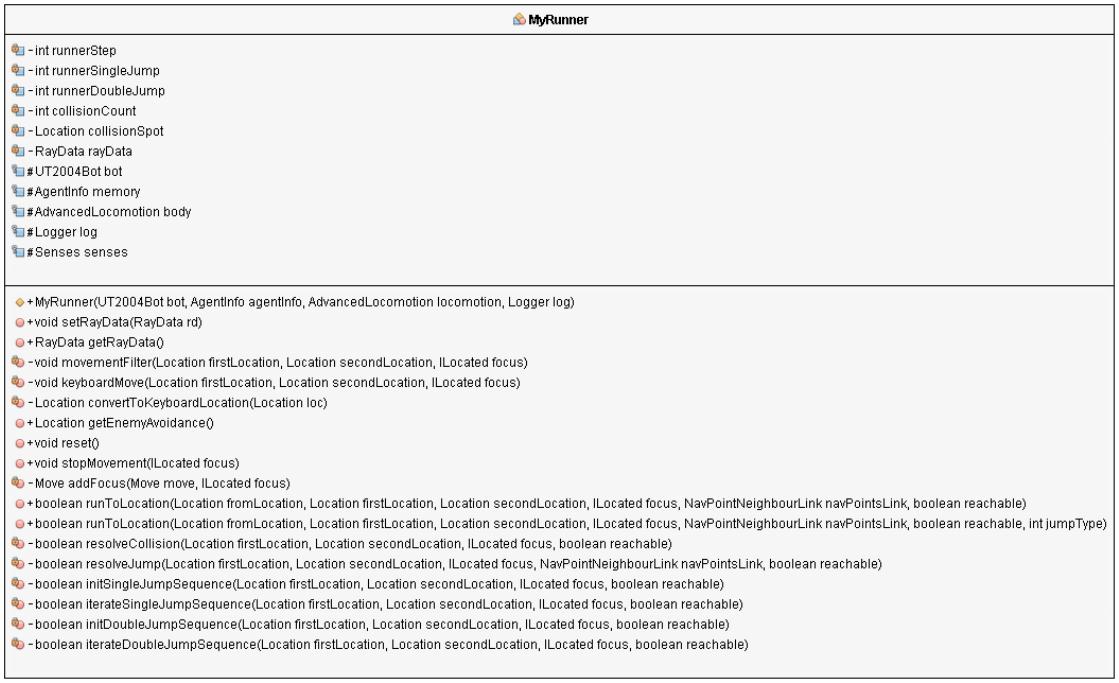


MyNavigator

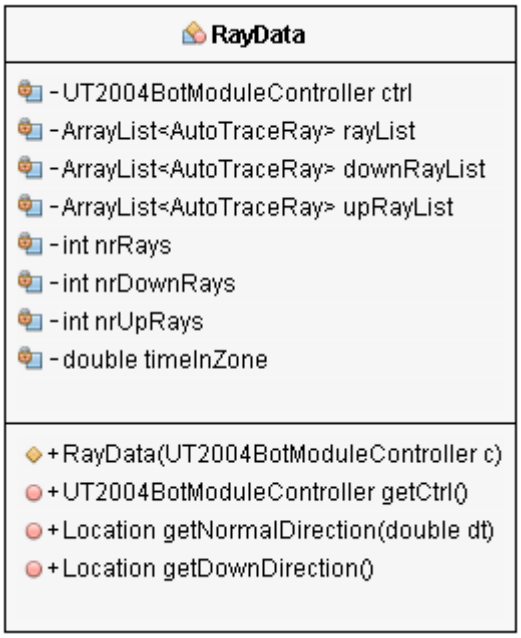
UT2004ModuleController



MyRunner

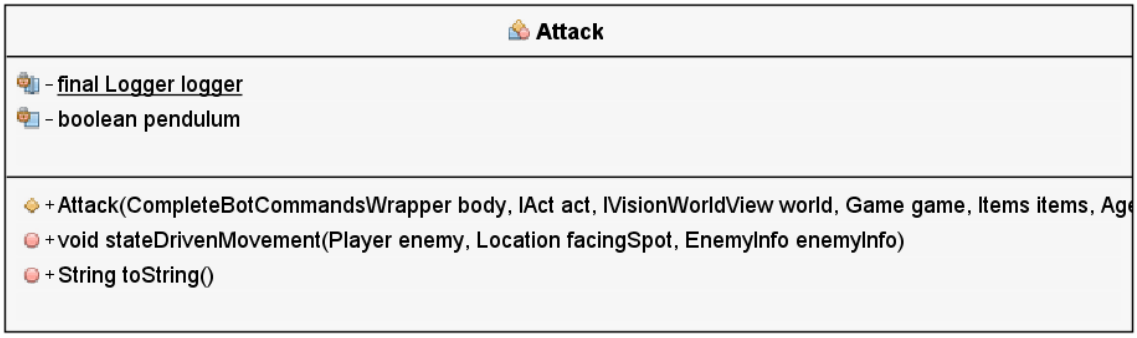


RayData



DefensiveProfile

Attack



SecondaryState

Mirzorbot

Brain

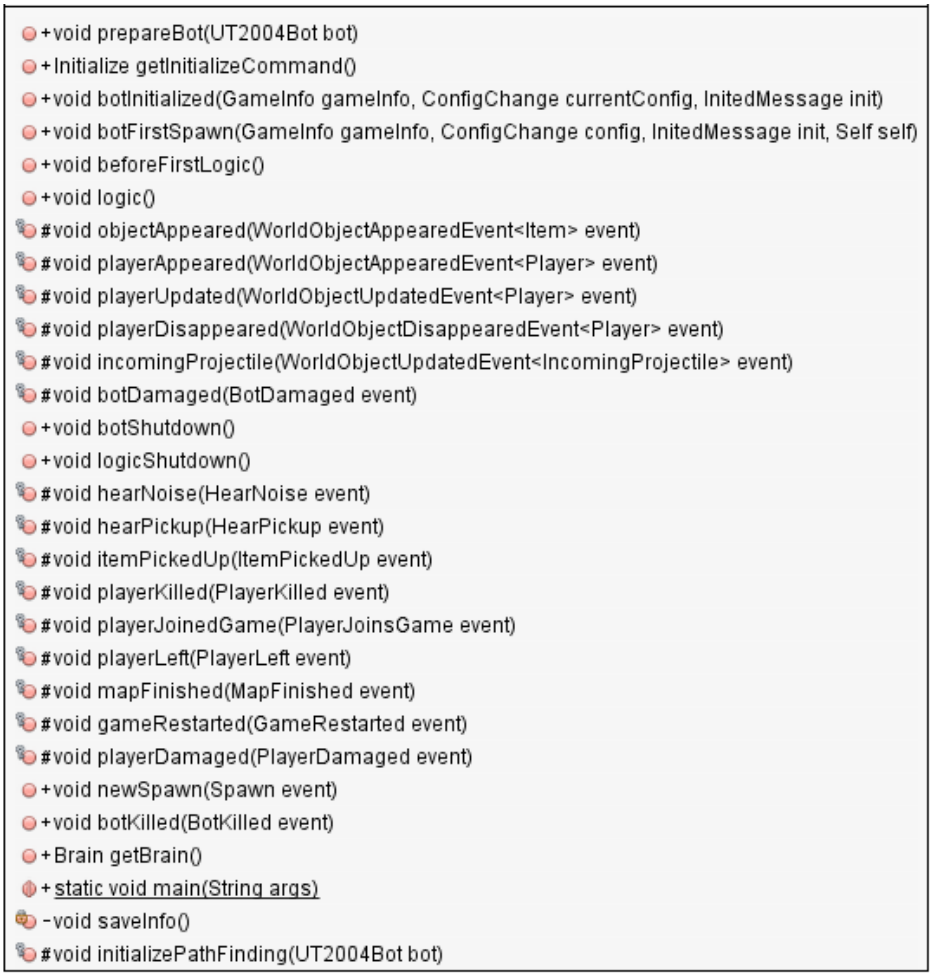
Atributos

Funciones

Mirzorbot

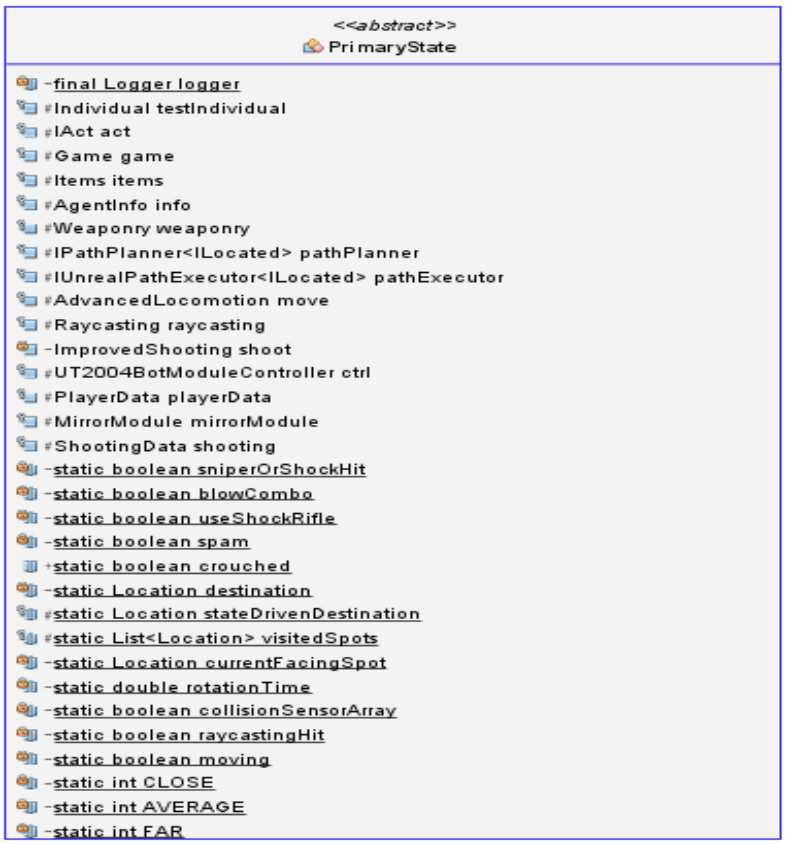
Atributos

Funciones



PrimaryState

Atributos

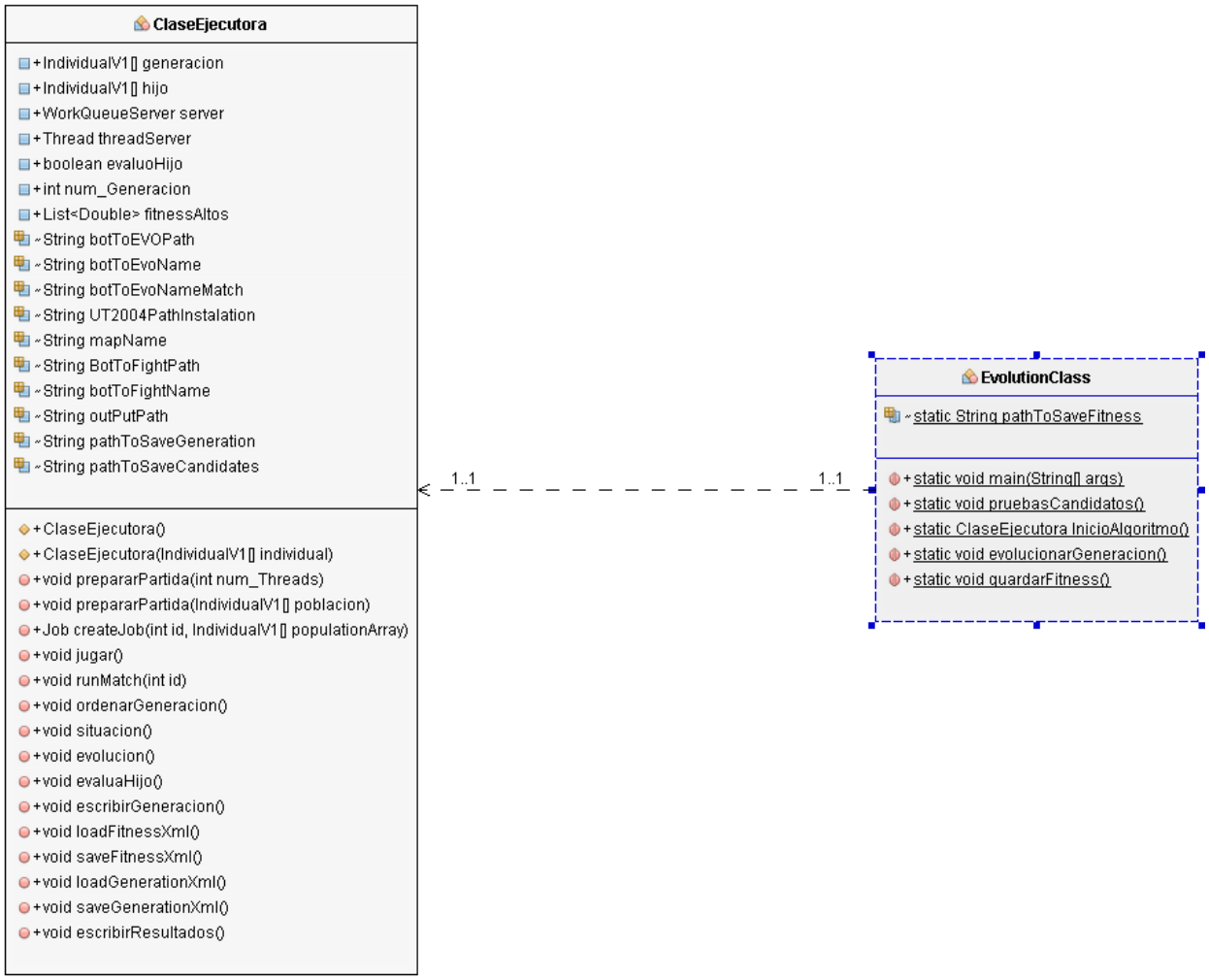


Funciones


#PrimaryState(CompleteBotCommandsWrapper body, IAct act, IVisionWorldView world, Game game, Items items, A
+void executeMovement(SecondaryState subState, Location newDestination, Player enemy, Location facingSpot, En
#void stateDrivenMovement(Player enemy, Location facingSpot, EnemyInfo enemyInfo)
+void destinationReached(SecondaryState subState)
+void stopExecution(SecondaryState subState)
+void botStuck(SecondaryState subState)
+void switchToBestWeapon(Player enemy, EnemyInfo enemyInfo)
+void engage(Player enemy, double dt)
-int estimateWeaponAdvantage(Weapon weapon, Player enemy, EnemyInfo enemyInfo)
+static void resetTempInfo()
+static void feasibleCombo()
+static void feasibleSpam()
+void playerDisappeared()
+void playerDamaged(PlayerDamaged event)
+void hearNoise(HearNoise noise, double clockTime, Player enemy)
+void hearPickup(HearPickup noise, double clockTime, Player enemy)
+void botDamaged(BotDamaged botDamaged, double clockTime, Player enemy)
-void goForward()
+void reactiveMovement()
+static int getFarRange()
+static int getMediumRange()
+static int getShortRange()
+PlayerData getPlayerData()
+void setPlayerData(PlayerData playerData)
+MirrorModule getMirrorModule()
+void setMirrorModule(MirrorModule mirrorModule)
+ShootingData getShooting()
+void setShooting(ShootingData shooting)

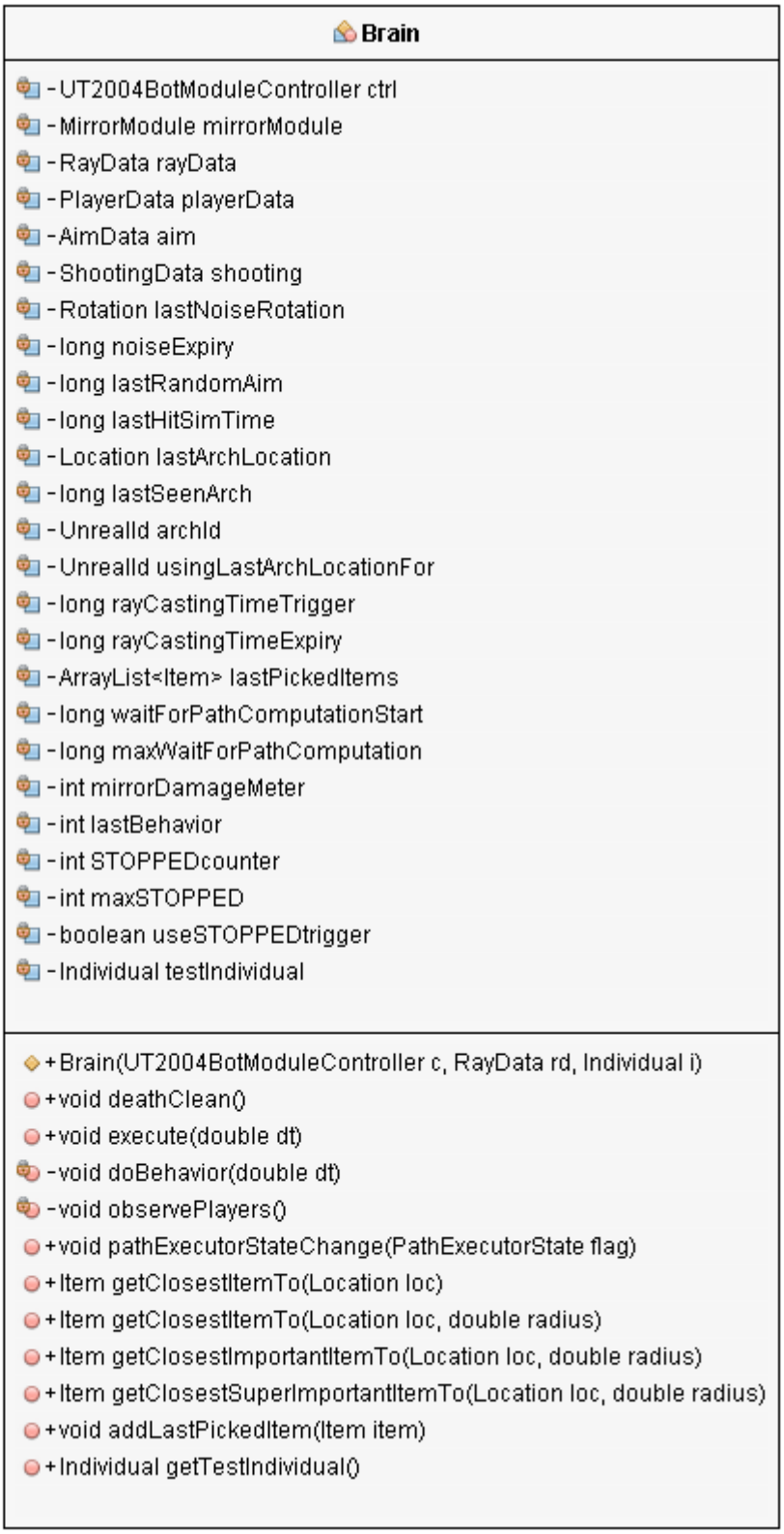
MirrorbotEVO

EvolutionProject

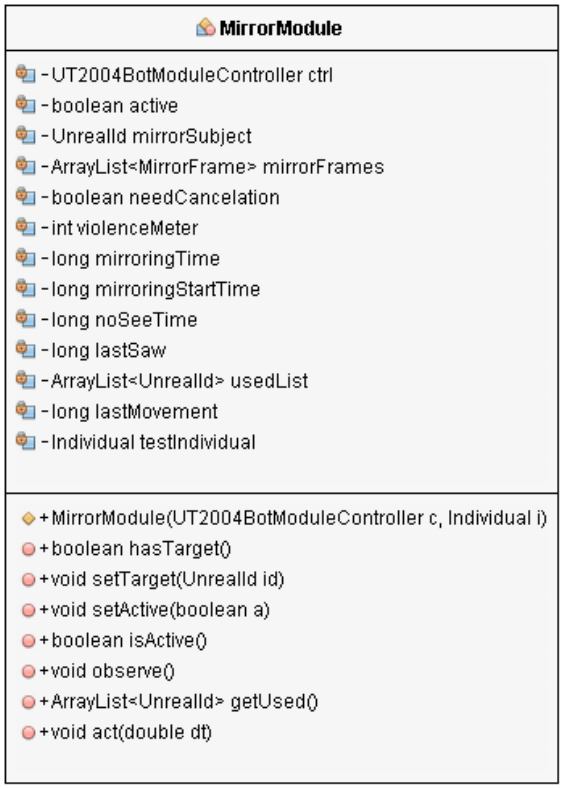


MirrorbotEvo

Brain


Brain
-UT2004BotModuleController ctrl
-MirrorModule mirrorModule
-RayData rayData
-PlayerData playerData
-AimData aim
-ShootingData shooting
-Rotation lastNoiseRotation
-long noiseExpiry
-long lastRandomAim
-long lastHitSimTime
-Location lastArchLocation
-long lastSeenArch
-UnrealId archId
-UnrealId usingLastArchLocationFor
-long rayCastingTimeTrigger
-long rayCastingTimeExpiry
-ArrayList<Item> lastPickedItems
-long waitForPathComputationStart
-long maxWaitForPathComputation
-int mirrorDamageMeter
-int lastBehavior
-int STOPPEDcounter
-int maxSTOPPED
-boolean useSTOPPEDtrigger
-Individual testIndividual
+Brain(UT2004BotModuleController c, RayData rd, Individual i)
+void deathClean()
+void execute(double dt)
-void doBehavior(double dt)
-void observePlayers()
+void pathExecutorStateChange(PathExecutorState flag)
+Item getClosestItemTo(Location loc)
+Item getClosestItemTo(Location loc, double radius)
+Item getClosestImportantItemTo(Location loc, double radius)
+Item getClosestSuperImportantItemTo(Location loc, double radius)
+void addLastPickedItem(Item item)
+Individual getTestIndividual()

MirrorModule



PlayerData