UNIVERSIDAD
DE MÁLAGA

E.T.S.
INGENIERÍA
INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMATICA
GRADO EN INGENIERÍA DEL SOFTWARE

DESARROLLO DE UN AGENTE 3D CON USO DE ALGORITMOS EVOLUTIVOS Y MINERÍA DE DATOS

DEVELOPMENT OF A 3D AGENT USING EVOLUTIVE ALGORITHM AND DATA MINING

Realizado por
D. Pedro Jesús Reyes Santiago

Tutorizado por
Prof. Dr. Marlon Nuñez Paz

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, (24/11/14)

Fecha de lectura: diciembre del 2014

El secretario del Tribunal

Resumen:

En este Trabajo Fin de Grado se lleva a cabo la implementación de un mundo 3D a través del uso del entorno Unity en el se cual realizará el desarrollo de un agente 3D el cual interactúe con el entorno que le rodea. Para ello haremos uso de algoritmos relacionado con la inteligencia artificial así como aplicación de algoritmos relacionados con la minería de datos tales como redes neuronales basando su aprendizaje en algoritmos evolutivos o arboles de decisión, respectivamente. Así pues, el objetivo de este proyecto es la creación de un agente 3D el cual sea capaz de adaptarse al entorno que le rodea, siendo hostiles algunos de estos entornos. Habrá principalmente 2 entornos los cuales serán una ciudad donde el agente deberá recoger clientes en su rol de taxista y soltarlas reconociendo a través de una serie de variables que personas son de fiar y cuales no. El segundo entorno es una cancha de baloncesto donde el agente deberá aprender a lanzar a canasta y reconocer con qué estados meteorológicos es viable jugar.

Palabras clave: algoritmos evolutivos, minería de datos, red neuronal, videojuegos

Abstract:

This Final Project carry out the implementation of a 3D world through the use of the Unity environment in which it´ll realize the development of a 3D agent who interact with the things around him. For that purpose it´ll be used algorithm related to artificial intelligence and data mining such as neuronal networks based his learning on evolutive algorithm and decision trees, respectively. Hence, the target of this project is the development of a 3D agent who is able to adapt himself to his environment, being unfriendly some aspects of this environment. There will be 2 main environments which will be a city where the agent should pick up clients as a taxi driver and take them recognizing by some variables who clients are trustworthy and who are not. The second aspect of the whole environment will be a basketball court where the agent should learn how to throw a ball and recognizing which meteorological conditions are viable to play

Keywords: evolutive algorithm, data mining, neural networks, video games, agent systems

◉ Índices

◉ Cuerpo del TFG

1. Introducción.

Las motivaciones que incitaron a la realización de este proyecto fue la posibilidad de crear un mundo 3D, un campo de pruebas donde poder realizar simulaciones en las que se pudieran visualizar casos reales de la vida cotidiana en los que se pudiese aplicar el apasionante mundo de la inteligencia artificial. Así pues, entre dichos objetivos a simular se llevará a cabo dos escenarios basados en dos aspectos fundamentales de la vida de cualquier ser humano: el trabajo y la diversión. Ambos aspectos están presentes en la vida de todos y cada uno de nosotros de un modo u otro y es por ello que se han elegido los escenarios que a continuación se presentan.

Nuestro primer escenario, relativo a la diversión, se basa en el desarrollo de una cancha de baloncesto donde el objeto 3D que encarna a la inteligencia artificial desarrollada debe realizar lanzamientos a una de las canastas en dicha cancha de baloncesto. Esta cancha de baloncesto no estará siempre abierta al publico debido a factores meteorológicos tales como la lluvia, la nieve o simplemente ha anochecido y por tanto no existe luminosidad suficiente como para jugar al baloncesto. Así pues, el objeto 3D que encarna esta IA, a partir de ahora llamado agente debido a sus similitudes con la teoría general asociada al concepto de sistemas agentes, en cuyo concepto se profundizará más adelante en el capitulo Estado del arte, jugará siempre y cuando las condiciones meteorológicas lo permitan. En cualquier otro caso, este agente no podrá jugar.

Con respecto a nuestro segundo escenario, el cual esta asociado al trabajo, el agente llevará a cabo su ejecución bajo el rol de conductor de un taxi. Bajo este rol, el agente ira recibiendo peticiones de clientes para ser recogidos y una vez recogidos la dirección a la que desean ir. Una vez se ha recibido dicha petición y se ha llegado al punto de recogida del cliente, así como todo taxista en la vida real debe llevar a cabo, nuestro agente deberá discernir que clientes son aptos para la entrada al taxi el cual, de acuerdo a la ley, esta permitido a no dar servicios a aquellos clientes que se encuentren bajo los efectos de alguna droga o alcohol, es decir, factores que pueden influenciar en la dirección del coche o seguridad del propio conductor. Así pues, en caso de que un cliente sea clasificado por la inteligencia artificial asociada a nuestro agente como un posible cliente peligroso en potencia, este cliente será descartado en ese momento, no abriendo la puerta del coche y eligiendo un nuevo cliente a quien dar servicio.

En los capítulos posteriores se explicarán los aspectos del entorno en los que vive el agente, algunos de estos hostiles, como por ejemplo: cambios climáticos, nivel de delincuencia del barrio y sus ciudadanos. El objetivo será que el agente autónomo se adapta a su entorno

intentando cometer el error mínimo tanto en su profesión como taxista como en sus actividades de entretenimiento. En el caso de su profesión aprenderá a partir de las malas experiencias obtenidas con ciudadanos que lo han atacado y ciudadanos que no lo han hecho aprendiendo qué ciudadanos son los más confiables. Por otra parte, en lo referente a sus actividades de entretenimiento, en este caso el lanzamiento a canasta, dado que su intención es tener la satisfacción de encestar aprenderá a hacerlo reduciendo el error del lanzamiento a canasta así como saber bajo que condiciones meteorológicos su actividad de entretenimiento es más efectiva.

Habiendo sentado hasta aquí las motivaciones y objetivo de este trabajo de fin de grado se procederá a explicar la estructura de este documento para así orientar al lector. Los capítulos de este documento están divididos en 5 partes fundamentales: introducción, estado del arte, solución, resultados, conclusión y posibles lineas futuras de investigación. Durante la introducción se ha realizado una breve aproximación de lo que se va a realizar así como las motivaciones de este proyecto. En el capítulo del estado del arte abarcaremos aspectos tales como: el género de este mundo 3D que en principio podría enmarcarse en el mundo de los videojuegos, algoritmos relacionados con la búsqueda de caminos así como de aprendizaje automático, culminando con plataformas y frameworks accesibles hoy día para el desarrollo de videojuegos así como para el uso de algoritmos de inteligencia artificial.

2. Estado del arte.

A continuación se describirá el estado del arte de los aspectos los cuales han sido tocados durante la realización de este proyecto con el objetivo de proporcionar al lector un encuadre donde este sea consciente de cuales son las posibilidades que a día de hoy tenemos con respecto a puntos tales como géneros de videojuegos, aprendizaje automático, pathfinding, plataformas de desarrollo de videojuegos y frameworks posibles a usar para el desarrollo de inteligencia artificial.

2.1. Géneros de juegos.

Así pues, comenzaremos en primer lugar analizando los posibles tipos de juegos que existen a día de hoy en el mercado. En primer lugar nos encontramos con los juegos de acción para los cuales el jugador debe hacer uso de sus reflejos, puntería y habilidad, a menudo en un contexto de combate o de superación de obstáculos y peligros. Estos juegos se dividen principalmente en las siguientes categorías:

- Lucha: recrean combates entre personajes controlados tanto por un jugador como por la computadora. Algunos de los juegos más conocidos de este género serían Street Fighter o Mortal Kombat.
- Beat ‘em up: similares a los de lucha, también llamados videojuegos de lucha a progresión, son videojuegos en los que los jugadores deben combatir un gran número de individuos mientras avanzan a lo largo de varios niveles con la posibilidad de jugar con más jugadores normalmente. Una sub-categoría de éstos podríamos denominarla como Hack and Slash cuando el protagonista va equipado con una espada o un hacha en cuya categoría podemos citar juegos tan conocidos como Dark Souls o God of War.
- Arcade: más que un género en sí, se trata de un calificativo bajo el que se engloban todos aquellos juegos típicos de las maquinas recreativas. Space Invaders o Pac-Man son algunos ejemplos de este género.
- Plataformas: en estos juegos el jugador controla a un personaje que debe avanzar por el escenario evitando obstáculos físicos, ya sea saltando, escalando o agachándose. Algunos juegos de este tipo serían Super Mario o Donkey Kong.

Otro género muy conocido es el género shooter. Entre ellos encontramos los siguientes tipos:

- Disparos en primera persona: conocidos también como FPS(First Person Shooter), las acciones básicas son mover al personaje y usar un arma, la arma se presenta en la pantalla en primer plano y el jugador puede interactuar con ésta. Los gráficos en tres dimensiones aumentan la impresión que se pretende dar de identificación fuerte con el carácter al usar este tipo de perspectiva. Entre juegos FPS encontramos Medal of Honor o Battlefield.

- Disparos en tercera persona: conocidos también como TPS(Third Person Shooter), se basan en la alternancia entre disparos y pelea o interacción con el entorno, a diferencia de los FPS los TPS se juegan con un personaje visto desde atrás o, en ocasiones, desde una perspectiva isométrica. Dentro de este género se encuentran videojuegos tales como Gears of War o Tomb Raider.
- Shot 'em up: es un tipo de juego de disparos con perspectiva en 2D, donde habitualmente manejamos una nave espacial, aunque también puede ser otro tipo de vehículo o un personaje. Aún siendo juegos en 2D, a veces se incluyen o combinan elementos en 3D para dar efectos de profundidad, explosiones o mayor efecto visual, sobre todo con los jefes. Space Invaders o Gradius son algunos ejemplos de juegos desarrollados bajo este género.

Dentro de otra categoría ampliamente desarrollada y jugada dentro del mundo gamer así como una en las que el uso de áreas referentes a la inteligencia artificial se hace más evidente llegando incluso a ser en ciertas ocasiones de gran dificultad vencer a los personajes enemigos en esta categoría podemos dilucidar los siguientes géneros:

- Estrategia en tiempo real: en este tipo de juegos la acción se realiza sin ninguna pausa encontrándose en dicho genero juegos tan conocidos como Age Of Empires o Starcraft.
- Estrategia por turnos: como su nombre indican se basa en la premisa de que cada jugador posee un turno para decidir qué hacer. Podemos citar dentro de este género juegos tales como Final Fantasy o Total War, entre otros.
- Otros subgéneros: podemos encontrar otros subgéneros destacables dentro de los juegos de estrategias tales como: de táctica en tiempo real, de táctica por turnos, de guerra, de construcción de imperios, de artillería, MOBA(Multiplayer Online Battle Arena), o Tower defense.

A continuación, hablaremos de los juegos de simulación cuyo marco podríamos decir se asemeja más a las características del proyecto que se ha realizado. No obstante, no existe como tal un género que defina en su totalidad lo que en este proyecto se ha querido realizar. Así pues, podemos encontrar los siguientes géneros dentro de la categoría de juegos de simulación:

- Simulación de vehículos: son videojuegos que permiten al jugador operar una variedad de vehículos de forma más o menos realista. Como consecuencia esta categoría se divide en subgéneros tales como simulación de conducción (Gran Turismo), de vuelo (Microsoft Flight Simulator), de vuelo de combate (Lock On: Modern Air Combat), de combate de vehículos (Battlezone), ferroviarios (Microsoft Train Simulator), náuticos (Ship Simulator), de submarinos (Silent Hunter).

- Simulación de construcción: son juegos muy populares en PC, donde el programa proporciona al usuario todas las herramientas para construir un proyecto, en el cual se consideran desde gastos de construcción y mantenimiento, hasta una linea de tiempo, física y clima que afecta todas las decisiones que tome. Encontramos muchos subgeneros dentro de la simulación de construcción tales como construcción de ciudades, de imperios, de economía, de Dios, de política, de granja, de gestión deportiva o sandbox, caracterizado este último por una gran libertad creativa y de desarrollo no lineal donde podemos poner como ejemplo el conocido juego comprado por Microsoft, Minecraft.
- Simulación de vida: este genero a menudo llamado también juego de simulación social se enfoca en controlar un personaje con capacidades y emociones humanas, y controlando todos los aspectos de su vida, desde donde vivirá, qué estudiará, hasta con quién se casará. Probablemente este es el género que más se asemeja a lo que se pretende hacer en este proyecto con la diferencia de que todos estos aspectos se pretende que sean llevados a cabo por la máquina en sí misma, sin control alguno de un jugador tal y cómo podemos ver en juegos de esta categorías tales como The Sims donde la inteligencia artificial puede dejar un poco que desear surcando un poco algunas paginas relativas a este tema donde podemos encontrar personajes no jugadores quemándose mientras corren de un lado a otro en la orilla de una playa.
- Simulación de combate: en este género se recrean situaciones de guerra más allá de lo que lo puedan hacer los populares juegos de disparos. Se caracteriza por el elevado realismo en todos los aspectos relevantes, ya sea en el manejo de un soldado o un cuadrilla militar. Un destacado exponente de este subgénero lo encontramos en Operation Flashpoint y su secuela Armed Assault.
- Otros: otros subgéneros dentro de los videojuegos de simulación son los simuladores de zoológicos, de parques de atracciones, de médicos, de citas, de caza o de animales.

Para finalizar, podemos encontrar otros tipos de géneros de videojuegos tales como de deporte, de carreras, de aventuras o rol.

Una vez hemos abarcado el género en el que actualmente y, vuelvo a citar, no con gran precisión podemos situar a este proyecto el cual sería simulación de vida, me gustaría acuñar como concepto un nuevo término el cual sería más preciso para lo que se pretende hacer en este proyecto el cual quedaría definido como ALS, es decir, Autonomous Life Simulation. Este nuevo género, simulación de vida autónoma, quedaría definido como el entorno en el que independientemente de que haya personajes jugadores o no, es decir, humanos tras las acciones, todos los personajes de este entorno son capaces de bajo ciertas acciones del mundo en el que se encuentran y de las cuales tienen percepción en su totalidad o parcialmente de las mismas son capaces de realizar acciones ante dichas acciones perceptibles por éstos.

2.2. Aprendizaje automático.

Proseguiremos ahora con el estado del arte en las áreas en las cuales se ha hecho uso dentro de la rama de la inteligencia artificial en este proyecto. A día de hoy, el aprendizaje automático o aprendizaje de maquinas, conocido más bien como machine learning, del cual se ha hecho uso en la realización de este proyecto es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender. De forma más concreta, se trata de crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos. Encontramos diferentes tipos de algoritmos dentro de los algoritmos de aprendizaje supervisado tales como:

- Aprendizaje supervisado: el algoritmo produce un función que establece una correspondencia entre las entradas y las salidas deseadas del sistema. Un ejemplo de este tipo de algoritmo es el problema de clasificación, donde el sistema de aprendizaje trata de etiquetar (clasificar) una serie de vectores utilizando categorías o clases. Este tipo de aprendizaje puede llegar a ser muy útil en problemas de investigación biológica, biología computacional y bioinformática.
- Aprendizaje no supervisado: todo el proceso de modelado se lleva a cabo sobre un conjunto de ejemplos formado tan sólo por entradas al sistema. No se tiene información sobre las categorías de esos ejemplos. Por lo tanto, en este caso, el sistema tiene que ser capaz de reconocer patrones para poder etiquetar nuevas entradas.
- Aprendizaje semi-supervisado: este tipo de algoritmos combinan los dos algoritmos anteriores para poder clasificar de manera adecuada. Se tiene en cuenta los datos marcados y los no marcados.
- Aprendizaje por refuerzo: el algoritmo aprende observando el mundo que le rodea. Su información de entrada es el feedback o retroalimentación que obtiene del mundo exterior como respuesta a sus acciones. Por lo tanto, el sistema aprende a base de ensayo-error.
- Aprendizaje por transducción: similar al aprendizaje supervisado pero no construye de forma explícita una función. Trata de predecir las categorías de los futuros ejemplos basándose en los ejemplos de entrada, sus respectivas categorías y los ejemplos nuevos al sistema.
- Aprendizaje multitarea: métodos de aprendizaje que usan conocimiento previamente aprendido por el sistema de cara a enfrentarse a problemas parecidos a los ya vistos.

El aprendizaje automático es un proceso automático que todo ser humano tiene. Sin embargo, hasta ahora no ha sido tan sencillo poder imitar un proceso tal como el de nuestro cerebro para el aprendizaje. En el aprendizaje automático aplicado a día de hoy podemos obtener 3 tipos de conocimiento, que son:

- Crecimiento: es el que se adquiere de lo que nos rodea, el cual guarda la información en la memoria como si se dejara huellas.
- Reestructuración: al interpretar los conocimientos el individuo razona y genera nuevo conocimiento al cual se le llama reestructuración.
- Ajuste: es el que se obtiene al generalizar varios conceptos o generar los propios.

Estos tres tipos de obtención de conocimiento se efectúan durante un proceso de aprendizaje automático pero la importancia de cada tipo de conocimiento depende de las características de lo que se está tratando de aprender.

El aprendizaje supervisado se caracteriza por contar con información que especifica qué conjuntos de datos son satisfactorios para el objetivo del aprendizaje mientras que el aprendizaje supervisado el programa no cuenta con datos que definan que información es satisfactoria o no. La información obtenida por un algoritmo de aprendizaje no supervisado debe ser posteriormente interpretada por una persona para darle utilidad.

Algunas aplicaciones del aprendizaje automático son:

- Motores de búsqueda.
- Diagnóstico médico.
- Detección de fraudes con el uso de tarjetas de crédito
- Análisis del mercado de valores.
- Clasificación de secuencias de ADN.
- Reconocimiento del habla.
- Robótica.
- Minería de datos.
- Big Data.

Determinar la estructura de la función adecuada para resolver y el problema y la técnica de aprendizaje correspondiente. Por ejemplo, se podría optar por utilizar una red neuronal artificial o un árbol de decisión los cuales han sido usados durante la realización de este proyecto. En los siguientes sub-capítulos se habla sobre estas dos últimas técnicas mencionadas.

2.2.1.Redes neuronales.

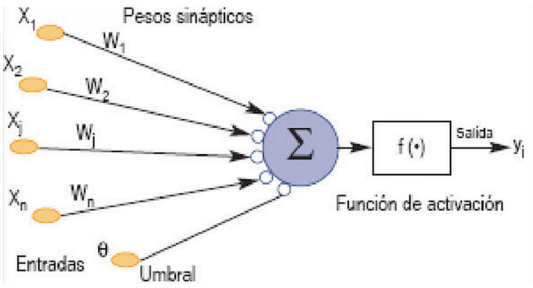
A continuación, se introducirá el estado del arte de las redes neuronales de las cuales se ha hecho amplio uso a lo largo de este proyecto para poder resolver problemas de clasificación y aproximación.

Las redes de neuronas artificiales (denominadas habitualmente como RNA o en ingles como ANN) son un paradigma de aprendizaje y procesamiento automático inspirado en la forma en que funciona el sistema nervioso de los animales. Se trata de un sistema de interconexión de neuronas que colaboran entre sí para producir un estímulo de salida.

Cabe recordar que que las propiedades básicas que constituyen a una red neuronal son un conjunto de unidades denominadas neuronas donde cada una de estas unidades recibe una serie de entradas a través de interconexiones y emite un salida. Esta salida viene dada por tres funciones:

- Una función de propagación: conocida como función de excitación que por lo general consiste en el sumario de cada entrada multiplicada por el peso de su interconexión (valor neto). Si el peso es positivo, la conexión se denomina excitadora; si es negativo, se denomina inhibitoria.
- Una función de activación: la cual modifica a la función de propagación y que puede darse la posibilidad de que no exista siendo en este caso la salida la misma que la de la función de propagación.
- Una función de transferencia: la cual se aplica al valor devuelto por la función de activación. Se utiliza para acotar la salida de la neurona y generalmente viene dada por la interpretación que queramos darle a dichas salidas. Algunas de las más utilizadas son la función sigmoide (para obtener valores en el intervalo [0, 1]) y la tangente hiperbólica (para obtener valores en el intervalo [-1, 1])

En la siguiente imagen se puede observar lo explicado anteriormente:



En lo que respecta al tipo de estructuras que pueden llegar a formarse son muchos los modelos que existen siendo una serie de ellos los que prevalecen en el mundo académico y en el mundo especializado en esta materia tales como:

- Perceptron
- Adaline
- Perceptron multicapa
- Memorias asociativas
- Maquina de Botzmann
- Maquina de Cauchy
- Propagación hacia atras (backpropagation)
- Redes de Elman
- Redes de Hopfield
- Red de propagación
- Redes de neuronas de base radial
- Mapas Autoorganizados (RNA) (Redes de Kohonen)
- Crecimiento dinamico de células
- Gas Neuronal Creciente
- Redes ART (Adaptative Resonance Theory)

Dado que el propósito de este documento no es ilustrar al lector con uno y cada uno de los diferentes tipos de redes neuronales existentes hoy en día y de lo que en principio, podría ser más importante a día de hoy, que son las funciones utilizadas para calcular las salidas de cuyas funciones hemos estado hablando antes realizaremos al menos un análisis sobre una de las técnicas utilizadas para optimización de problemas usada en este proyecto de tal modo que se haga patente que no sólo la estructura sino las funciones usadas determinan la capacidad de nuestra red neuronal para resolver problemas. Una de estas técnicas orientada a la optimización de problemas y usada en este proyecto ha sido el algoritmo de Levenberg-Marquardt.

El algoritmo de Levenberg-Marquardt es un método de aproximación a una función simple pero robusto el cual consiste básicamente en resolver la siguiente ecuación:

$$(J^tJ + \lambda I)\delta = J^tE$$

Donde J es la matriz Jacobiana del sistema, λ es el factor damping o factor de amortiguación de Levenberg usado a efectos prácticos para realizar aproximaciones a través del algoritmo de Gauss-Newton o a través del gradiente dependiendo del crecimiento y decrecimiento de esta variable resolviendo problemas no lineales a través de este paso de algoritmos de diferente comportamiento. El parámetro δ es el vector actualizado de los pesos que nosotros queremos

encontrar para la red neuronal y E es el vector error contenido en las salidas que estamos obteniendo con los parámetros actuales usando los vectores de entrada en el entrenamiento de la red. El parámetro δ nos dice cuánto se debería cambiar los pesos de nuestro red neuronal para lograr posiblemente la mejor solución. La matriz J^tJ es la ya conocida matriz Hessiana.

Como ya se cito anteriormente, el factor damping o de amortiguamiento λ es ajustado en cada iteración de la red neuronal, tantas iteraciones como se desee, guiando el proceso de optimización. Si la reducción del parámetro E es muy rápida, un valor más pequeño de λ puede ser usado, trayendo como consecuencia una aproximación más cercana a través del uso del algoritmo de Gauss-Newton, mientras que si una iteración da una insuficiente reducción en este valor residual E, λ puede ser incrementada, dando un solución más próxima a la dirección del gradiente. Así pues, este algoritmo una vez resumida la formula en la que esta basado consistiría en:

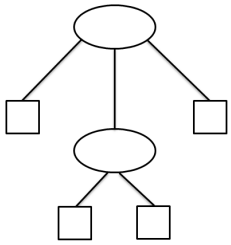
- *1 - Computar la matriz Jacobiana.*
- *2 - Computar el error gradiente a traves de la fórmula: $g = (J^t)E$.*
- *3 - Aproximar la Hessiana usando la matriz Jacobiana: $H = (J^t)J$.*
- *4 - Resolver $(H + \lambda I)\delta = g$ para encontrar δ.*
- *5 - Actualizar los pesos w de la red neuronal usando δ.*
- *6- Calcular de nuevo la suma de los errores cuadrados usando los pesos actualizados.*
- *7 - Si la suma de los errores cuadrados no ha disminuido se descarta los nuevos pesos, se incrementa λ usando el vector v y volvemos al paso 4. El valor v citado anteriormente es un factor de ajuste normalmente definido con un valor por defecto de 10 el cual si λ se ha de incrementar, λ es multiplicado por este valor v, y si λ disminuye, λ es dividido por este valor v.*
- *8 - En otro caso disminuimos el valor de λ usando v y paramos.*

El algoritmo de Levenberg-Marquardt como todo algoritmo tiene sus limitaciones, una de ellas es que es muy sensible a los pesos iniciales en la red. También hay que tener en cuenta que este algoritmo no considera outliers en los datos, para los poco expertos en estadística un outlier son datos sospechosos de no pertenecer al conjunto de datos de donde proceden, o ser producto de algún suceso sumamente extraño pudiendo existir valores outliers moderados y extremos. Este defecto del algoritmo de Levenberg-Marquardt hace posible la existencia de un overfitting noise o dicho en pocas palabras, que la aproximación tengan en cuenta en la descripción de su solución estos valores cuando lo que debería de hacer es ignorarlos. No obstante, para resolver este problema puede usarse una técnica conocida como regularización.

2.2.1. Árboles de decisión y regresión.

Un árbol de decisión es un modelo de predicción utilizado en el ámbito de la inteligencia artificial. Dada una base de datos se construyen diagramas de construcciones lógicas, muy similares a los sistemas de predicción basados en reglas, que sirven para representar y categorizar una serie de condiciones que ocurren de forma sucesiva, para la resolución de un problema.

Estos árboles de decisión dependiendo del algoritmo que decidamos usar pueden lidiar con variables continuas o discretas. Estos árboles están formados por un nodo principal, una serie de nodos internos los cuales hacen las veces de condiciones en el árbol de decisión, y unos nodos finales los cuales indican el resultado final o decisión que se ha de tomar como se indica en la siguiente figura:



Así pues existen dos algoritmos principales dentro de este área los cuales se habla en detalle a continuación.

- ID3

El algoritmo ID3 es utilizado dentro del ámbito de la inteligencia artificial para usos tales como la búsqueda de hipótesis o reglas usando, como ya se ha indicado anteriormente, un conjunto de ejemplos para tal fin.

El conjunto de ejemplos que se proporciona a este árbol de decisión esta formado por una serie de tupas de valores, cada uno de ellos denominados atributos, en el que uno de ellos es el atributo a clasificar y el cual se caracteriza por ser de tipo binario, es decir, solo podrá tener dos posibles valores: 0 o 1, No o Sí, Negativo o Positivo, Mentira o Verdad.

El algoritmo ID3 se basa en el siguiente esquema de ejecución:

ID3(Ejemplos, Atributo-objetivo, Atributos)
Si todos los ejemplos son positivos devolver un nodo positivo
Si todos los ejemplos son negativos devolver un nodo negativo
Si Atributos está vacío devolver el voto mayoritario del valor del atributo objetivos en Ejemplos

En otro caso

Sea atributo A el MEJOR de Atributos

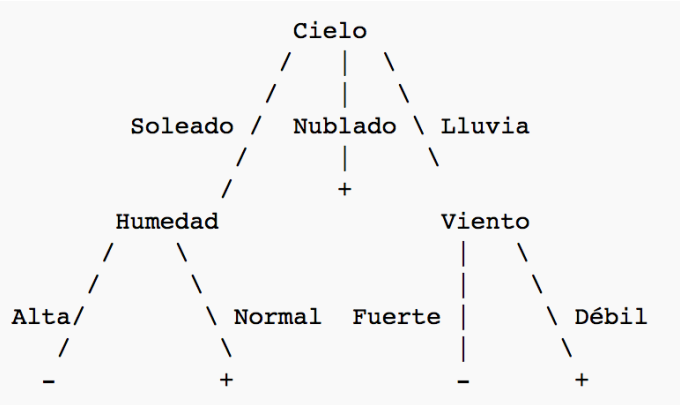
Para cada valor v del atributo hacer

Sea Ejemplos(v) el subconjunto de ejemplos cuyo valor de atributo A es v

Si Ejemplos(v) esta vacío devolver un nodo con el voto mayoritario del Atributo-objetivo de Ejemplos

Sino Devolver ID3(Ejemplos(v), Atributo-objetivo, Atributos/{A})

Un ejemplo muy común en la literatura sobre este algoritmo es de una variable objetivo denominada Jugar basada en factores meteorológicos tales como la lluvia o el sol obteniendo un árbol de decisión como el que sigue:



- C4.5

C4.5 es un algoritmo usado para generar un árbol de decisión desarrollado por Ross Quinlan. C4.5 es una extensión del algoritmo ID3 anteriormente descrito el cual puede ser usado para clasificación. En esta extensión se introducen mejoras tales como:

- Manejo de atributos continuos y discretos. Con el objetivo de manejar atributos continuos, C4.5 crea un umbral y luego se divide la lista en aquellos cuyo valor de atributo es superior al umbral y los que son menores o iguales a él.
- Manejo de los datos de formación con valores de atributos faltantes - C4.5 permite que los valores de los atributos sean marcados como faltantes. Los valores faltantes de los atributos simplemente no se usan en los cálculos de la ganancia y la entropía.
- Manejo de atributos con costos diferentes.
- Podando árboles después de la creación - C4.5 se remonta a través del árbol una vez que ha sido creado e intenta eliminar las ramas que no ayudan, reemplazándolas con nodos de hoja.

El pseudocódigo del algoritmo C4.5 sería el siguiente:

- *1 - Comprobar los casos base*

- *2 - Para cada atributo a encontrar la ganancia de información normalizada de la división de a*
- *3 - Dejar que a_best sea el atributo con la ganancia de información más alta*
- *4 - Crear un nodo de decisión que divida a_best*
- *5 - Repetir en las sublistas obtenidas por división de a_best y agregar estos nodos como hijos de nodo.*

Como nota al pie cabe decir que existe una implementación open source en Java de este algoritmo bajo el nombre de J48 en la herramienta weka de minería de datos la cual ha sido probada y ofrece buenos resultados. Además, en caso de que este algoritmo no satisfaga del todo las necesidades que se buscan Quilan continuo con la creación del C5.0 y See5 (C5.0 para Unix / Linux, See5 para Windows) con fines comerciales los cuales ofrecen una serie de mejoras con respecto a su antecesor C4.5. Algunas de estas son:

- Velocidad - C5.0 es significativamente más rápido que el C4.5.
- El uso de memoria - C5.0 es más eficientes que el C4.5.
- Árboles de decisión más pequeños - C5.0 obtiene resultados similares a C4.5 con arboles de decisión mucho más pequeños.
- Soporte para boosting - Boosting mejora los árboles y les da una mayor precisión.
- Ponderación - C5.0 le permite ponderar los distintos casos y tipos de errores de clasificación.
- Winnowing - una opción automática de C5.0 para eliminar aquellos atributos que puede ser de poca ayuda.

Los fuentes de una versión para Linux de un único subproceso de C5.0 pueden encontrarse disponibles bajo licencia GPL.

- M5P

M5P es una reconstrucción del algoritmo M5 de Quinlan usando inducción y modelos de regresión. M5P combina un árbol de decisión convencional con la posibilidad de funciones de regresión lineal en los nodos.

Primero, un algoritmo de inducción de arboles de decisión es usado para construir el árbol, pero en lugar de maximizar la información ganada en cada nodo interno, un criterio de división es usado para minimizar la varianza dentro de los subconjuntos posibles en los valores de las clases en cada rama. El proceso de división de M5P para si los valores de las clases de todas las instancias o en su totalidad alcanzan un variación muy ligera.

Segundo, el árbol es podado hacia atrás desde cada hoja.

Tercero, para evitar discontinuidades grandes entre los arboles un proceso de suavizado es aplicado el cual combina el modelo de predicción de la hoja con el de los nodos en el camino hacia atrás que finaliza en la raíz, suavizando en cada nodo a través de la combinación del valor predicho por el modelo lineal del nodo en particular.

M5P puede lidiar como valores faltantes usando una técnica de nominada “surrogate splitting” que encuentra otro atributo a dividir en lugar del original y lo usa en su lugar. Cuando el proceso de división termina todos los valores faltantes son reemplazados por el los valores medios de los correspondientes atributos de la fase de entrenamiento alcanzados en las hojas. Si se testea el valor de un atributo no conocido éste es reemplazado por el valor medio de todos las instancias de entrenamientos que alcancen dicho nodo, con efecto de elegir siempre el nodo más popular.

M5P genera modelos compactos y relativamente comprensibles.

2.3.Pathfinding.

Otro aspecto importante en el movimiento de un objeto en cualquier mundo, ya sea este 2D o 3D, y que ha sido utilizado para el movimiento de nuestro agente a través de las calles de nuestra ciudad es el Pathfinding el cual es el trazado del camino más corto entre dos puntos. Este campo de investigación esta fuertemente basado en el algoritmo de Dijkstra para encontrar el camino más corto en un grafo de pesos. Entre los algoritmos usados en pathfinding encontramos:

- El algoritmo de Dijkstra
- El algoritmo de búsqueda A*
- El algoritmo D*, siendo éste una familia de algoritmos de pathfinding dinámicos para problemas en los cuales las restricciones varían a lo largo del tiempo. En esta familia encontramos los siguientes algoritmos de búsqueda incremental:
    - El algoritmo D* original realizado por Anthony Stentz, algoritmo de búsqueda incremental informado.
    - El algoritmo focused D*, algoritmo de búsqueda heurística incremental informado desarrollado por Anthony Stentz que combina las ideas usadas en A* y el algoritmo D* original.
    - El algoritmo D* Lite, algoritmo de búsqueda heurístico incremental por Sven Koenig y Maxim Likhachev que construye un LPA*, un algoritmo de búsqueda heurístico incremental que combina ideas de A* y SWSF-FP dinámico.

En concreto, en la realización de este proyecto se ha realizado una implementación del algoritmo de búsqueda heurística A*, el cual es basado en la siguiente formula:

$$f(n)= g(n) + h(n),$$

donde g(n) indica la distancia del camino desde el nodo origen s al n y h(n) expresa la distancia estimada desde el nodo n hasta el nodo destino t.

Este algoritmo sigue un proceso de ejecución similar al que se explica a continuación:

- *1 - Establecer el nodo s como origen. Hacer f(s)=0, y f(i)=∞ para todos los nodos i diferentes del nodo s. Iniciar el conjunto Q vacío.*
- *2 - Calcular el valor de f(s) y mover el nodo s al conjunto Q .*
- *3 - Seleccionar el nodo i del conjunto Q que presente menor valor de la función f(i) y eliminarlo del conjunto Q.*
- *4 - Analizar los nodos vecinos j's de i. Para cada enlace (i, j) con coste c_{ij} hacer*
    - *4.1 - Calcular: f(j)'=g(i)+c_{ij} +h(j)*
    - *4.2 - Si f(j)'<f(j),*
    - *4.3 - Actualizar la etiqueta de j y su nuevo valor será: f(j) = g(i) + c_{ij} + h(i)*
    - *4.4 - Insertar el nodo j en Q*
    - *4.5 - Si f(j)' ≥ f(j)*
    - *4.6 - Dejar la etiqueta de j como está, con su valor f(j)*
- *5 - Si Q está vacío el algoritmo se termina. Si no está vacío, volver al paso 3.*

En este tipo de algoritmos heurísticos es muy importante tener un buen estimador h(n) ya que este es el que informa la distancia al nodo objetivo por lo que pueden ocurrir los siguientes casos, siendo h'(x) un estimador de h(x) donde h(x) es el estimador o función perfecta que establece la relación de coste exacta entre donde nos encontramos y a donde queremos llegar:

- Si h'(x) hace una estimación perfecta de h(n), A* converge inmediatamente al objetivo.
- Si h'(x) = 0, la función g(x) controla la búsqueda.
- Si h'(n) = 0 y g(x) = 0 la búsqueda será aleatoria.
- Si h′(x) = 0 y g(x) =1 o constante la búsqueda será Primero en Anchura.
- Si h′(x) nunca sobrestima a h(x) (o subestima), se garantiza encontrar el camino óptimo, pero se desperdicia esfuerzo explorando otras rutas que parecieron buenas.
- Si h′(x) sobrestima a h(x), no puede garantizarse la consecución del camino del menor coste

Por todos estos casos, es importante saber elegir nuestro h'(x) lo mejor que podamos ya que de esta función dependerá en prácticamente su totalidad la eficiencia de nuestro algoritmo heurístico A*.

2.4.Plataformas de desarrollo de videojuegos.

En lo que respecta a motores gráficos y lenguajes de desarrollo actuales a poder utilizar para el desarrollo de videojuego existen muchos, entre los más conocidos pueden ser citados las siguientes:

- Unity3D, posiblemente uno de los motores gráficos más conocidos a día de hoy. Robusto, fácil de usar, potente, versátil tanto si eres artista como si eres programador, compatible con una gran cantidad de plataformas, innovador en el modo que afronta el desarrollo de videojuegos y con un gran comunidad de usuarios los cuales hacen fácil resolver dudas y problemas que van surgiendo durante el aprendizaje de dicho motor gráfico. Además cuenta con una serie de tutoriales ofrecidos por los propios creados de la plataforma muy completos junto con una documentación excelente. Por todo ello, Unity3D ha sido elegido como plataforma de desarrolla para nuestro proyecto.
- Shiva 3D, definido como el motor para videojuegos más compatible del mundo, es capaz de soportar desarrollos para Windows, MacOS, Linux, iPhone, Android, BlackBerry, Palm, Wii y iPad. La manera de acercarse a un desarrollo es un tanto peculiar y se aleja de las fórmulas más conocidas que emplean Unity o el UDK sin ir más lejos. Puede representar un problema a la hora de entrar en la aplicación y acostumbrarse a su uso pero una vez superado parece ser muy cómoda y versátil. La parte negativa es que no encontramos una comunidad tan grande como en otros motores más extendidos.
- GameMaker Studio, es un clásico ya para el desarrollo para iOS, Android y Web. Una plataforma a la que es muy sencillo entrar y que dispone de las suficientes herramientas como para que los menos avezados en el desarrollo se sientan cómodos y puedan adelantar sus prototipos. Sin duda, si es un juego 2D lo que se piensa hacer y no existe reticencia a aprender lo mínimo de programación, diseño y arte como para intentar desarrollar en GameMaker, esta es la opción más viable.
- Torque, es probablemente el más difícil para entrar en su desarrollo el cual se basa en un sistema WYSIWYG (What You See Is What You Get) ofreciendo grandes prestaciones y un motor muy versátil. Lo mejor quizás de Torque es que ha evolucionado mucho y ahora ofrece tres variantes distintas que pueden adaptarse a nuestros proyectos. Existe Torque3D, mejorado con las últimas tecnologías, Toque2D para desarrollar juegos en 2D y el relativamente nuevo iTorque2D especialmente diseñado para dispositivos móviles iOS.
- XNA Game Studio 4.0, es un entorno de programación que permite usar Visual Studio para crear juegos para Windows Phone, la consola Xbox 360 y equipos basados en Windows. XNA Game Studio incluye XNA Framework, que es un conjunto de bibliotecas diseñadas para el desarrollo de juegos basados en Microsoft .NET Framework 2.0. Podemos encontrar una documentación para esta plataforma de desarrollo en su pagina

oficial la cual contiene visiones generales sobre tecnología, tutoriales y materiales de referencia relacionados con XNA Game Studio.

En cualquier caso, si lo que deseamos no es estar lidiando con entornos de desarrollo sino que preferimos usar la API para el desarrollo de videojuegos nos encontramos una gran amplitud de posibilidades dependiendo del lenguaje en el que nos encontremos más cómodos tales como:

- Ogre 3D, un potente motor renderizado en C++ pero con muchos ports a distintos lenguajes de programación.
- Panda 3D, en Python.
- jMonkeyEngine en Java.
- Cocos3D para el desarrollo en dispositivos iOS del cual están realizando un port para Android.
- AndEngine, exclusivamente para Android.
- UDK, que te ofrece un entorno visual que te permiten desarrollar tanto para sobremesas como para iOS.
- Java3D, es una herramienta muy poderosa para hacer diferentes clases de juegos en 3D, el cual adquirió fuerza en los últimos años (incluso se ha convertido en un contendiente de gran fuerza en la creación de juegos) … gracias a muchas herramientas, librerías de open-source y algunas IDE fantásticas que se han desarrollado. En Java puedes hacer juegos de pantalla completa, rápidos y con aceleración del hardware además de aprovechar la plataforma java.

Por otro lado si lo que se desea desarrollar va a estar básicamente enfocado en la web existen frameworks en Javascript y HTML5 como Isogenic Engine o LimeJS. Cabe comentar también que Panda 3D permite incrustar desarrollos utilizando sus librerías en paginas web mediante el uso de un plug-in.

2.5.Herramientas software y frameworks de inteligencia artificial.

Entre herramientas software y frameworks los cuales existen hoy día en el mercado tanto comerciales como gratuitos y de entre los cuales algunos surgieron como posibilidad para el desarrollo de la inteligencia artificial en este proyecto podemos encontrar los siguientes:

- Los usados:
    - Aforge.NET: es un framework en C# diseñado por desarrolladores e investigadores el cual contiene un conjunto de librerías en el campo de la visión artificial, la inteligencia artificial, procesamiento de imágenes, redes neuronales, algoritmos evolutivos, machine learning, robótica, etc. Ésta ha sido la librería que ha sido utilizada principalmente junto con Accord.NET para el desarrollo de la inteligencia asociada a nuestro agente por su gran amplitud en cuánto a algoritmos que abarcan diferentes aspectos de la inteligencia artificial, así como estadísticos, proporcionándonos una gran movilidad ante diferentes problemas presentados durante el proceso de desarrollo de este proyecto. Además, ha sido vital el hecho de que esta librería haya sido escrita en C# ya que el entorno utilizado ha sido Unity el cual solo permite 3 lenguajes (Boo, JavaScript, C#) para los cuales fue de extremada dificultad encontrar librerías de similares características tales como este completo framework en lenguajes soportados por dicha plataforma. Así pues, por disponibilidad, rapidez, amplitud de opciones en el desarrollo de una posible solución a un problema así como el evitar la pérdida de tiempo, en no tener que aprender un lenguaje nuevo tal como Boo o profundizar mis conocimientos en JavaScript para Unity, y la pérdida de dinero, ya que esta librería es totalmente gratuita, se optó finalmente, por el uso en el desarrollo de los aspectos relativos al campo de la inteligencia artificial en general en este proyecto, de esta librería a excepción de algunos algoritmos y mejoras implementadas a parte para el correcto funcionamiento de los algoritmos desarrollados en esta librería.
    - Accord.NET: es un framework para computación científica en .NET. Este framework se ha construido sobre Aforge.NET incorporando nuevos algoritmos para procesamiento de datos, machine learnings, visión por computador, distribución de probabilidades, test de hipótesis y soporte para las más populares técnicas de medidas de rendimiento. Entre alguno de estos nuevos algoritmos se encuentran el algoritmo de Levenberg-Marquardt y Resilient Backpropagation.

- Otros:
    - Emergent: simulador de redes de neuronas.
    - FANN: Fast Artificial Neural Network es una librería open source implementada en C.
    - Neural Designer: es una aplicación la cual transforma datos simples en conocimiento, descubriendo relaciones complejas, reconociendo patrones ocultos o prediciendo tendencias.
    - NeuroIntelligence: esta es una aplicación del mismo índole que Neural Designer la cual tiene como propósito mejorar la productividad a través de una visualización y un aprendizaje de los mismo que permita al cliente incrementar su productividad o manera de llevar a cabo una tarea en particular.
    - Neuroph: este framework proporciona una serie de clases escritas en Java, siendo un proyecto de código abierto en SourceForge bajo licencia Apache, para la creación y formación de redes neuronales.
    - NeuroSolutions: esta es una aplicación que se encuentra dentro del mismo tipo que oferta aplicaciones anteriores citadas tales como NeuroIntelligence o NeuroDesigner.
    - Synapse: mismas características que NeuroSolutions.
    - OpenNN: es una biblioteca informática escrita en C++ que implementa redes neuronales. La biblioteca es de código abierto, esta alojada en SourceForge y ha sido distribuida bajo Licencia Pública General Reducida de GNU.

Durante el próximo capítulo se tratará el sistema que se ha creado para que el mundo que rodea al agente y cómo éste mundo influye él sea totalmente independiente del agente de tal modo que podamos desarrollar ambos aspectos de manera simultánea, es decir, nuestro mundo a pesar de que no existiese ningún agente podría seguir corriendo sin problemas ya que este mundo tiene aspectos que escapan al control del ser humano y, por tanto, deben escapar también al de nuestro agente como son factores tales como cuándo llueve o cuándo nieva.

3. Implementación del entorno.

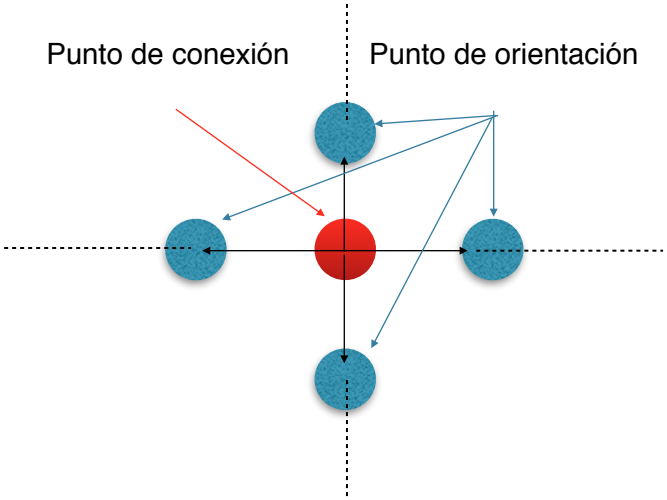
Para la implementación del entorno se ha realizado el desarrollo de un sistema generador de entornos el cual es un sistema dentro del mundo 3D que ejecuta todos los aspectos generales de este mundo 3D simulando aspectos que se podrían dar en la vida real de tal modo que, las acciones por las cuales el agente se vea influenciado, son independientes del mismo. De este modo lo único que hay que centrarse a la hora de programar la inteligencia artificial de este agente es únicamente la manera en la que va a percibir estas influencias o actos por el mundo que le rodean o que hacen que el mundo cambie de un estado A a un estado B.

Así pues, se pasara a explicar a continuación el desarrollo de estos aspectos controlados de manera independiente y como el agente percibe las acciones del mundo en caso de que se le haya dado dicha habilidad.

3.1.Creación del mapa 3D.

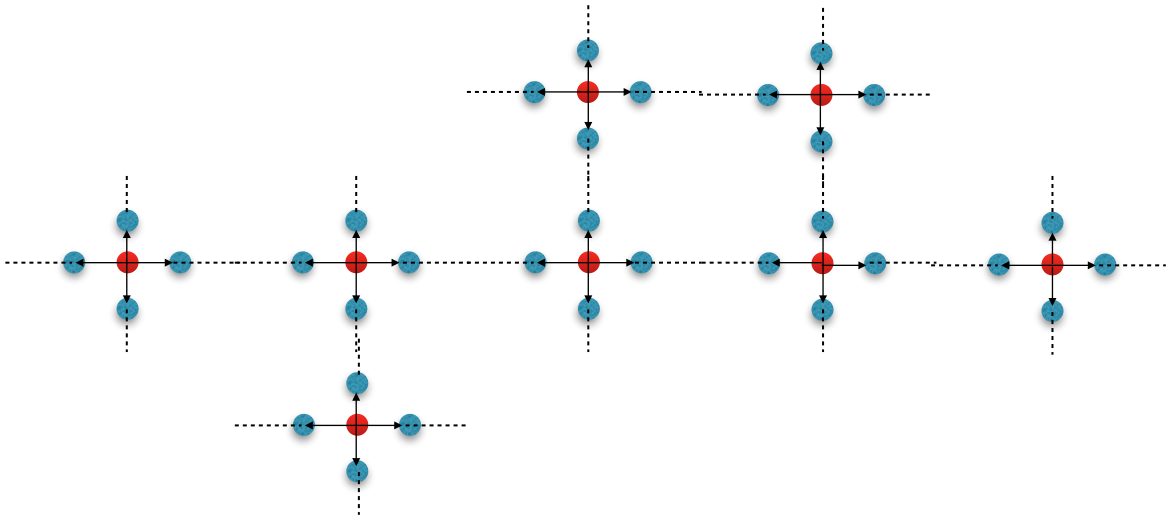
Para el recorrido que nuestro agente debe realizar para llegar de un punto a otro necesitábamos del uso de una estructura la cual nos permitiese adquirir un mapa en una estructura fácil de iterar. Para ello se basó la creación de nuestros mapas en una estructura fácil, basada en un punto de conexión y uno o más puntos de orientación, la cual se colocaba antes de iniciar el sistema y a través de dicha estructura base crear el mapa así como los objetos necesarios para que ese mapa cobre vida.

Así pues, nuestra estructura base, a un nivel abstracto, sería la siguiente:



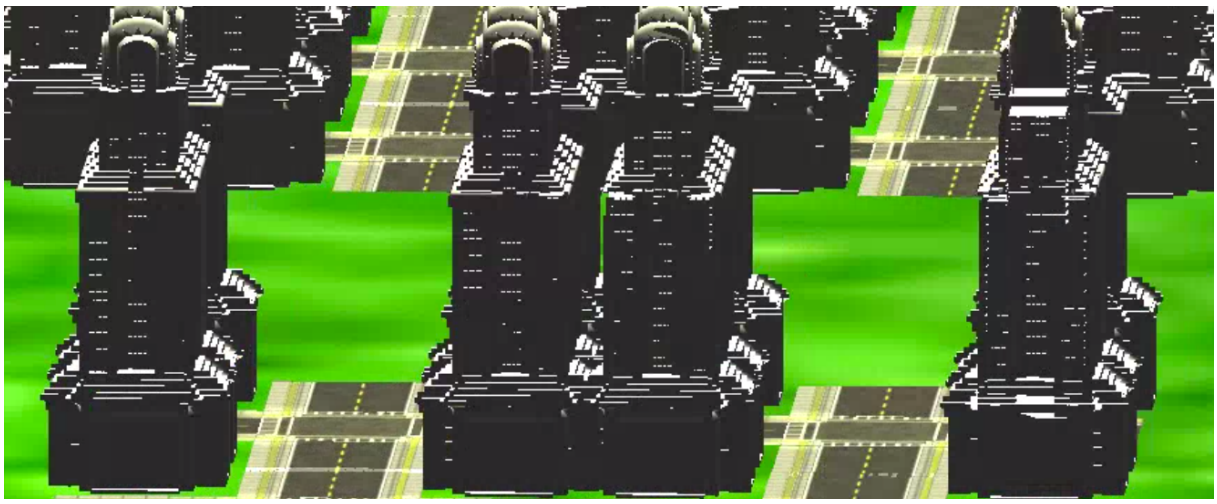
Donde el punto de conexión indica el lugar donde el agente puede basar su movimiento para ir de un punto de conexión A a otro punto de conexión B. Los puntos orientativos son utilizados para, una vez nos encontramos en un punto de conexión A, usar estos puntos de orientación para saber donde se encuentra el punto de conexión B.

Con esto lo que conseguimos es que no importa como sea nuestro mapa ya que lo único que cambiaría serían nuestros puntos de orientación que indican hacia donde lanzar un vector director en la búsqueda de otro punto de conexión en esa dirección. Con esta estructura base, la cual como ya se ha dicho podría cambiar en cualquier momento, cambiando, añadiendo o eliminando puntos de orientación, podríamos crear cualquier tipo de estructuras para probar a nuestro agente siendo además una manera más fácil de crear mapas 3D. Con esto, usando una estructura base como la anterior podemos crear mapas simplemente copiando dicha estructura una y otra vez creando mapas como el siguiente:

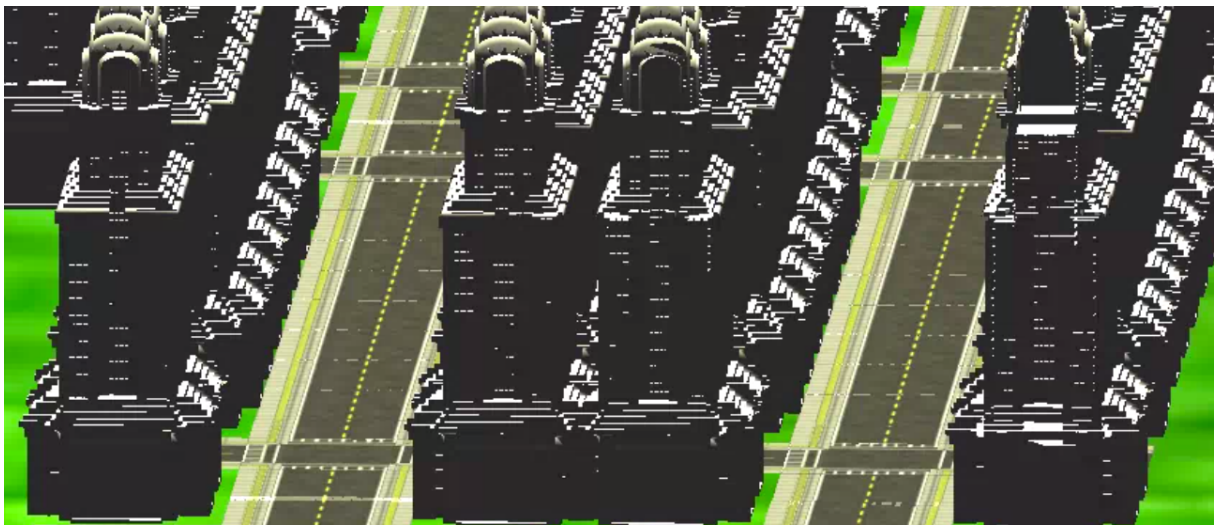


Aunque en este caso el agente que tenemos realizará búsquedas en 2D, con este mismo principio se podrían crear mapas 3D en los que nuestro agente pudiese "volar" o simplemente moverse en el eje Y y no solo en el eje X y Z como hasta ahora mismo hace ya que, en definitiva, la manera de plasmar el mundo 3D en la estructura utilizada es la misma. Sencillamente habrá puntos de conexiones en diferentes lugares del eje Y no estarán todos a la misma altura como en este momento se encuentran en nuestro proyecto.

Una vez este mapa ha sido creado dentro de nuestra estructura, es decir, hemos transformado en una estructura iterativa a nivel de código lo que tenemos inicialmente en nuestro mundo 3D, surge el problema de rellenar esos huecos con estructuras que puedan dar continuidad a una calle en un ciudad ya que si nuestra estructura dentro del mundo 3D inicialmente es la siguiente:



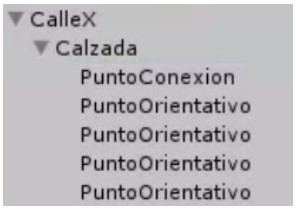
Al crear una conexión entre dos calles separadas tendríamos un hueco entre dichas dos calles. Para solventar este problema de estética una parte de nuestro generador automático de entornos se encarga de crear en tiempo de ejecución las calles y edificios necesarios entre dos calles de tal modo que lo único de lo que nos tengamos que encargar nosotros es de las calles, las intersecciones y demás puntos importantes y no de colocar manualmente edificios, asfalto y calzada entre dichos puntos de interés que conforman nuestro mapa final. Para conseguir esto, se va recorriendo la estructura de puntos que tenemos de puntos de conexión creando todo este engranaje de detalles tales como edificios y calzadas a través de la unión que conforman dos puntos de conexión usando para ello vectores directores normalizados. Con estos vectores director normalizado, comenzando desde cualquiera de los puntos de conexión los cuales están conectados entre sí, iterando uno por uno, y sabiendo cual es el tamaño de los modelos que debemos disponer en el mundo 3D en tiempo de ejecución entre dichos puntos de conexión se crean los elementos intermedios que pudiesen haber entre dos calles. Esto supone una gran libertad para el que quiera desarrollar la parte de mapa ya que el modelo que ponemos entre las ciudades podría cambiarse en cualquier momento así como el asfalto que se esta usando también podría ser modificado por un modelo de arena del desierto o lo que se nos ocurriese. Así pues, dicho lo anterior, usando la estructura de puntos de conexión y de orientación obtendríamos en tiempo de ejecución que el hueco que teníamos al inicio del sistema se rellena con edificios, asfalto y calzadas como vemos en la siguiente imagen:



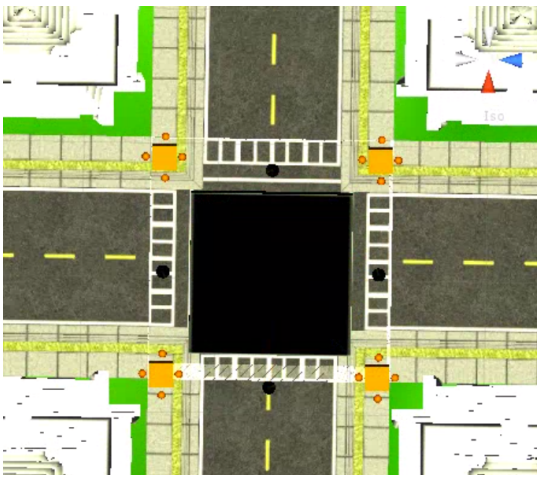
De este modo nos ahorramos una cantidad de tiempo considerable editando nuevos mapas 3D ya que no lo tendremos que hacer manualmente sino que este sistema independiente lo hará por nosotros en tiempo de ejecución. Además, obtenemos una estructura fácil de iterar a nivel de código para programar nuestros algoritmos basada en un punto de conexión y cuáles son los puntos de conexión con los que esta conectado el punto de conexión en el que nos encontramos.

El cómo se hace esto implica una serie de clases así como el uso del árbol de Unity en la representación de objetos del mundo 3D. No es el objetivo de este documento comentar las clases desarrolladas para la realización del proyecto pero en este caso en concreto se hará un inciso debido a la importante influencia en la que el modo en que se ha elegido realizar el mapeado ha tenido sobre el proyecto.

Así pues, comenzamos por explicar la unidad básica en la cual se basa nuestro mapeado siendo esta PuntoConexion.cs la cual contiene el punto dentro del mundo 3D en el cual se encuentra el objeto al que va asociada la instancia de dicha clase así también como una serie de puntos de enlace, tantos como nosotros queramos, con otros objetos haciendo uso de puntos de orientación para poder establecer dichos enlaces, es decir, ser consciente de estos enlaces, almacenándolos en nuestra estructura, los cuáles no son más que otros puntos de conexión. Basándonos en esta estructura y plasmando dicha estructura en el árbol de creación de objetos de Unity en el editor antes de la ejecución del mapa obtenemos un árbol como el siguiente donde una calzada esta representada por un punto de conexión y cuatro orientativos:

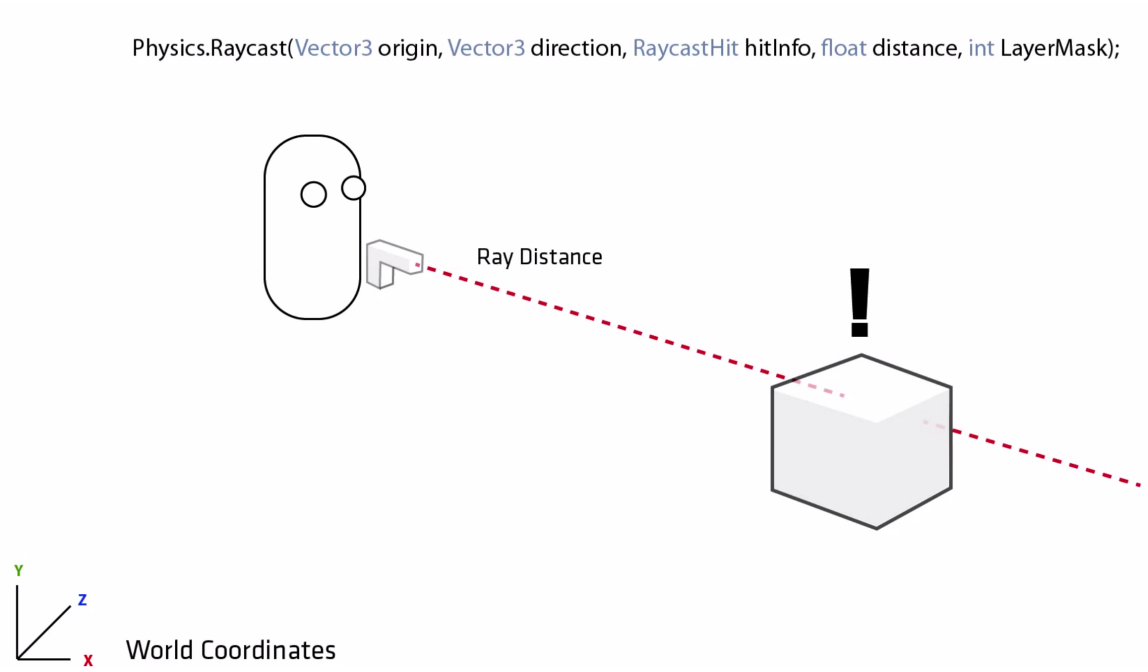


Esta estructura quedaría finalmente representada por lo que vemos en la siguiente imagen donde los círculos negros serían nuestros puntos orientativos y el gran cuadrado negro central nuestro punto de conexión:



Así pues, una vez tenemos situados en los sitios correctos estos modelos básicos como un cuadrado y varios círculos dentro del árbol de Unity podemos usar dicho árbol para establecer su conexión con el código y a través de dicho código realizar lo anteriormente comentado como son la creación de un mapa así como los objetos 3D para dar un aspecto de ciudad al mapa que hemos realizado.

Hasta aquí tendríamos la estructura esencial utilizada y su representación antes de la ejecución en el editor de Unity. Cabe decir que el lugar donde colocamos los círculos negros o la cantidad de ellos es variable por lo que podríamos crear ciudades lo más complejas que se nos ocurra. Así pues, una vez se conoce la estructura básica que se esta utilizando en el mundo 3D para crear ciudades y que, como se ha dicho anteriormente, puede ser modificada en cualquier momento pasamos a como se crean las conexiones entre estas estructuras. A través del uso de la clase RayCast la cual en términos simples nos sirve para dado un vector origen y destino se lanza un rayo que nos devuelve los nombres de los objetos con los que choca a su paso.



Con el uso de esta clase RayCast vamos plasmando recorriendo cada punto de conexión del árbol de Unity utilizando los puntos de orientación de cada punto de conexión las conexiones con otros puntos de conexión.

Hasta aquí hemos realizado la disertación del cómo realizamos nuestro mapa tanto a nivel de código así como a nivel 3D. A continuación, se explicará el cómo se pobla éste mundo 3D de ciudadanos.

3.2.Control de población.

Una parte importante de todo mundo 3D que se precie y más aún hablando de una ciudad, son los ciudadanos que la integran. Para este propósito este generador automático de entornos controla como parte vital del sistema el número actual de ciudadanos que ha generado, pudiendo establecer además el máximo y mínimo de ciudadanos dentro de la ciudad así como la posibilidad de que se generen nuevos ciudadanos o no. Esta probabilidad de creación de nuevos ciudadanos es debida a que por motivos de rendimiento no podemos generar 1000 ciudadanos de una tajada ya que supondría una alta carga de trabajo en CPU. Para evitar esto existe un atributo adicional bajo el nombre de ProbabilidadCreacionNuevoIndividuo cuyo valor ronda entre 0.0 y 1.0 indicando cuán probable es que se cree un nuevo individuo cada cierto tiempo, cuyo tiempo es indicado bajo el nombre de la variable TiempoEntreChequeos. De este modo podemos controlar cada cuánto tiempo podemos crear un nuevo ciudadano y la probabilidad de que dicho individuo sea creado o no.

3.3.Control meteorológico.

Una parte muy importante dentro de un mundo realista es la creación de factores meteorológicos los cuales puedan influir dentro de las acciones que se pueden realizar dentro de dicho mundo. En este caso se ha usado la meteorología como factor decisivo para jugar al baloncesto en una cancha de baloncesto outdoor o, dicho de otro modo, no cubierta. Nuestro generador automático de entornos llevará a cabo esta tarea basándose en una serie de parámetros cuyos valores son establecidos a nuestra elección siendo los siguientes:

- duración de un día en minutos
- probabilidad de lluvia durante un día
- probabilidad de nieve durante un día
- tanto por ciento de un día que durará la lluvia en caso de que llueva
- tanto por ciento de un día que durará la nieve en caso de que nieve
- tiempos entre chequeos de estos atributos

con estos 6 atributos principales controlamos como de lluvioso, soleado o nevado será nuestro mundo. Con estos atributos básicos y los prefabs y modelos necesarios para la simulación de los factores meteorológicos creamos un sistema con su propia meteorología totalmente independiente de la inteligencia que se desarrolle con la posibilidad de poder ampliarla en cualquier momento debido a su independencia de la inteligencia que se desarrolla para el agente, ya que este último solo accederá a este control meteorológico si le interesa.

3.4.Control de duración del día y la noche.

Para dar la continuidad de la cual un sistema agente debe presumir se ha establecido dentro de nuestro control meteorológico el guardado de una variable que establezca que tanto por ciento del día ha sido completado. Así pues, cuando el agente decida que por la noche no es una situación ideal para jugar al baloncesto y se vaya a trabajar con el taxi, el tiempo transcurrido de una escena a otra sea exactamente el mismo tanto por ciento transcurrido. Con esto, si un día dura 1 minuto, 30 segundos serán de día y otros 30 de noche por lo que cuando se haga de noche y nuestro agente se encuentre en la cancha de baloncesto y vaya a trabajar porque es de noche encontraremos que también es de noche cuando vayamos a trabajar, siendo ese tiempo de noche los 30 segundos restantes del día. Evidentemente, el personaje podrá dejar de jugar por otros motivos en cualquier momento como por ejemplo por nieve y si en ese caso solamente transcurrió 10 segundos del día, quedarán 50 segundos para que pase un día cuando el personaje se vaya a trabajar de los cuales 30 serán de noche y 20 de día.

3.5.Orientación del cielo.

Con esto conseguimos dar un cierto realismo al mundo teniendo unas nubes que se mueven y no un cielo el cual permanece estático en todo momento. Este es uno de los aspectos más sencillos y que no afectan al agente. La orientación del cielo es simplemente una parte del sistema la cual lleva a cabo la rotación en los ángulos X, Y, Z que elijamos a nuestro gusto del objeto en el mundo 3D el cual es portador de una textura con el cielo.

Aquí concluimos este capítulo del documento. Durante el siguiente capítulo profundizaremos en la implementación del agente autónomo que se ha desarrollado.

3.6.Implementación del ciudadano autónomo.

En este capítulo se llevará a cabo la descripción de los ciudadanos que pueblan nuestra ciudad. Estos ciudadanos una vez son creados por nuestro generador automático de entornos son ellos mismos quienes completan su estado inicial aleatorio dentro del sistema. Esto quiere decir que el propio ciudadano elige si quiere ser una persona de fiar o no dentro del sistema así como la posición origen, donde quiere ser recogido por nuestro agente en su rol de taxista, y la posición destino, donde quiere que el agente lo lleve.

Una de las partes que engloba a este ciudadano es la creación de un modelo de peligrosidad dentro del cual el propio ciudadano aleatoriamente se establece si en base a dicho modelo es una persona de fiar o no o, dicho de otro modo, si es una persona peligrosa o no. Este modelo es el que a posteriori nuestro agente, el cual no sabe nada a cerca de este modelo, deberá hallar para poder establecer por sí mismo que ciudadanos son de fiar y cuáles no lo son.

Queda decir que la información en la cual el ciudadano se establece que es de fiar o no así como su punto de recogida y punto de destino es enviado por dicho ciudadano a nuestro agente de tal modo que nuestro agente tiene una lista de todos los ciudadanos a los que se debe de dar servicio y donde debe recogerlos y dejarlos en la ciudad que se ha creado como mundo 3D.

3.6.1.Navegación.

Estos ciudadanos serán los que veamos merodeando por la ciudad utilizando un mapa de aceras el cual ha sido creado con el mismo método que se ha utilizado para crear las calzadas, cuyo método ha sido explicado anteriormente en el capitulo 3.1. Creación del mapa 3D, en la parte de nuestro generador automático de entornos encargado de la creación de la estructura de mapas. En el momento de creación del ciudadano éste realiza una copia del mapa de aceras que el generador de entornos ha creado y lo utilizará para moverse por la ciudad de tal modo que no es necesario la intervención de nuestro generador automático de entornos para realizar esta tarea sino que cada ciudadano controla su movimiento por sí mismo.

Por motivos de rendimientos se ha hecho que estos ciudadanos solo se muevan en caso de que tengan una visualización del agente a su alcance de tal modo que estos objetos dentro del mundo 3D no consumen recursos sino cabe la posibilidad de que en algún momento sean visualizados por el agente. Esto quiere decir que si el agente se encuentra en la misma calle que un ciudadano este ciudadano se moverá aunque no se encuentre visualizado en la pantalla ya que en cualquier momento el agente podría darse la vuelta y sí que aparecería en pantalla. En caso de no hacerse esto tendríamos ciudadanos que se están moviendo por el resto de la ciudad y los cuales, a nuestros ojos, están consumiendo recursos sin sentidos de la CPU en su movimiento al no ser éstos visualizados en pantalla.

Queda decir que la información en la cual el ciudadano establece su punto de recogida y punto de destino es enviado por dicho ciudadano a nuestro agente de tal modo que nuestro agente tiene una lista de todos los ciudadanos a los que se debe de dar servicio y donde debe recogerlos y dejarlos en la ciudad que se ha creado como mundo 3D.

3.6.2.Nivel de peligrosidad.

Para esta parte del ciudadano se ha creado una modelo sobre el que el ciudadano se apoya, cuyo modelo es totalmente desconocido por nuestro agente, para establecerse dicho ciudadano a sí mismo si es una persona de fiar o no.

Para establecer este modelo de nivel de peligrosidad de un ciudadano se ha llevado a cabo la creación de una clase denominada EstimadorComportamiento.cs en la cual, al ser esto una simulación de la realidad, establecemos los parámetros que hacen que un ciudadano sea de fiar o no. En nuestro caso usamos la presión sanguínea y las pulsaciones. Evidentemente se requiere el uso de muchas más variables tales como el sexo, edad, nivel de testosterona para evaluar algo tan subjetivo pero lo que se pretende con esto es dar un punto de vista de cómo se podría utilizar la minería de datos para evaluar tales factores aplicados a la vida real. Además, supondría un mayor número de ejemplos, es decir, de pruebas previas dentro del

mundo 3D para evaluar si una persona es de fiar o no al tener que evaluar muchas más variables que simplemente dos lo cual por motivos de tiempo y para el propósito de este proyecto carece de sentido.

Así pues, cada ciudadano de nuestro mundo creado a través del generador de entornos ejecuta su propio código de manera independiente realizando la creación de una instancia EstimadorComportamiento.cs y a través de ella establece si es de fiar o no en base a las características anteriormente citadas así como un atributo de entrada el cual hace referencia al nivel de delincuencia del barrio en el que el ciudadano va a ser recogido.

Esta información referente al nivel de peligrosidad es enviada de igual modo a nuestro agente a través de un mensaje para que este pueda evaluar por sí mismo si esta persona es de fiar o no.

A continuación se realizará un análisis más en profundidad de estos parámetros utilizados así como del modelo que, a posteriori, el agente deberá discernir de algún modo si es que no quiere ser atracado muchas veces.

El modelo se ha basado acorde a variables tales como la presión sanguínea, pulsaciones y lugar de recogida del cliente. Se han elegido las anteriores variables como método de decisión por las siguientes razones:

- Un cliente el cual no se encuentre bajo estados de embriaguez, drogas o con una intención de hacer daño no verá su presión sanguínea ni pulsaciones aceleradas y acorde a la situación legal en la que nos encontramos estas serían las únicas razones por las que un taxista podría hacer uso de su derecho de no dar servicio a un cliente. Bajo estos términos se entenderá como unos niveles de presión sanguínea y pulsaciones normales, es decir, niveles propios de “una persona de fiar”, los siguientes:
    - Pulsaciones = [72, 80] pulsaciones por minuto
    - Presión sanguínea = [90, 120] mmHg
- El lugar de recogida de un cliente influye. No es lo mismo recoger, si ponemos como ejemplo Málaga, a alguien en La Palmilla que en Marbella ya que los niveles de delincuencia son distintos. Es por esto que el lugar de recogida se ha marcado con 3 posibles niveles que marcarán el nivel de delincuencia:
    - Nivel de delincuencia bajo: implicará que el lugar de recogida del cliente no tiene importancia alguna y se deberá decidir si ese cliente es de fiar o no basándose en su presión sanguínea y pulsaciones por minuto.
    - Nivel de delincuencia medio: habrá un 50% de posibilidades de que la presión sanguínea y las pulsaciones sean decisivas y otro 50% de que sea el lugar de recogida. Así como la condición humana goza del libre albedrío, los datos en los

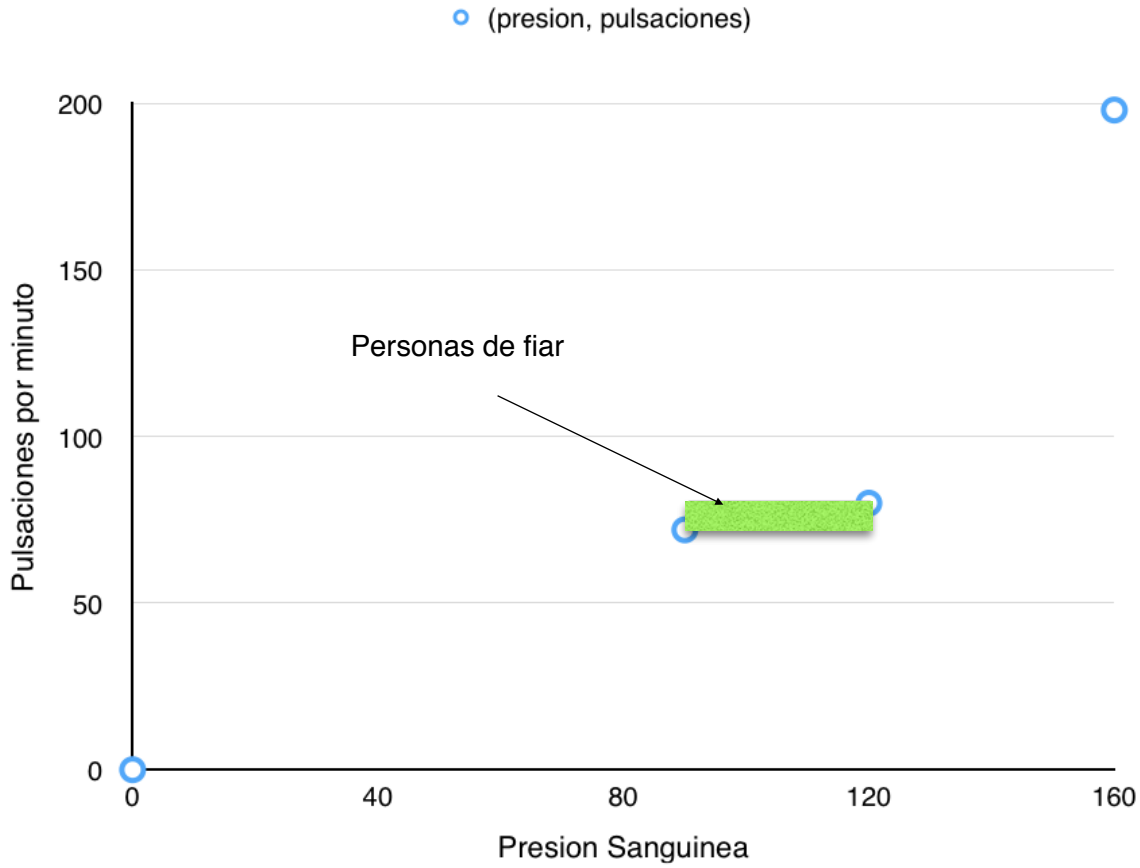
que se base esta maquina deben serlo también y dichos datos no pueden reflejar blanco o negro. Es por ello que se ha decidido incluir este nivel en el que la decisión de la maquina podría ser errónea.

- Nivel de delincuencia alto: implicará que por el lugar de recogida la persona que se esta recogiendo no es de fiar automáticamente.

Así pues, la inteligencia artificial del agente deberá aprender toda esta variedad de situaciones que se presentan en nuestro modelo en las cuales discernir que casos son importante tener en cuenta el lugar de recogida del cliente y en cuales es importante clasificarlos por sus pulsaciones y presión sanguínea.

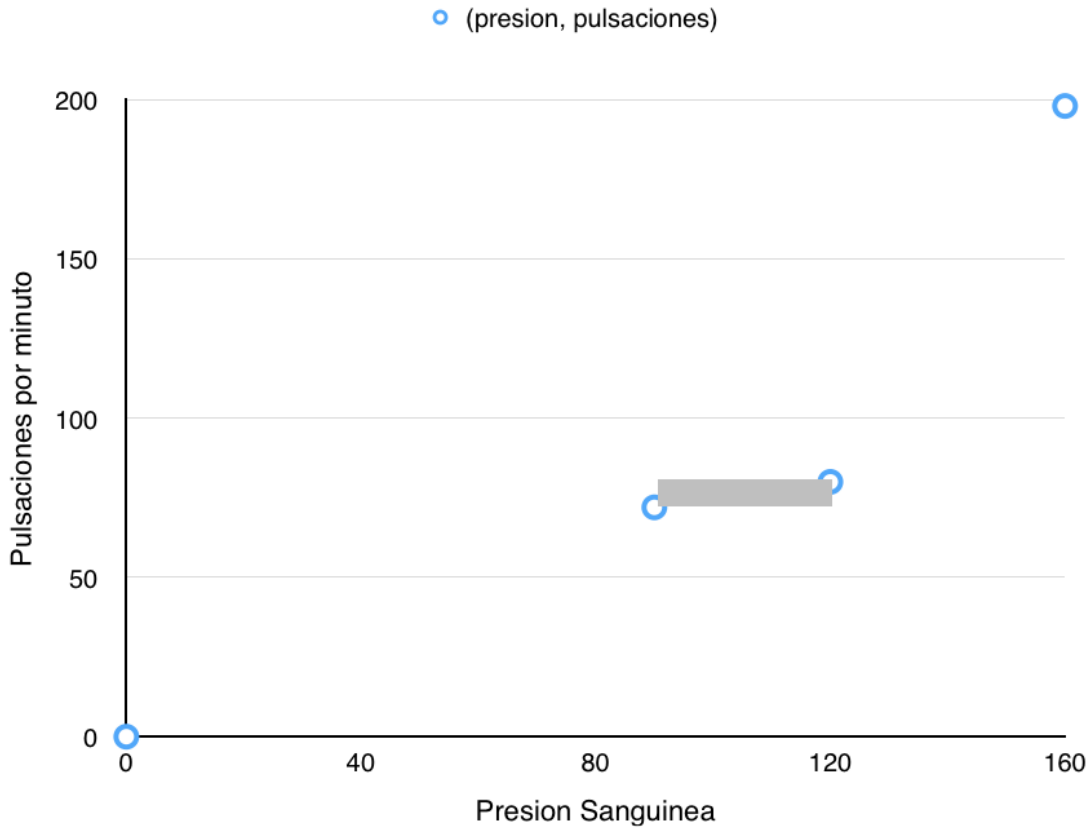
Para plasmar esta situación o factores dentro de los ciudadanos de nuestra ciudad así como de los niveles de delincuencia dependiendo del punto de recogida se ha llevado a cabo como se ha citado anteriormente el desarrollo de la clase EstimadorComportamiento.cs. Esta clase en base a un factor de entrada denominado como nivelDelincuenciaZona y en unos parámetros establecidos acorde a las variables citadas anteriormente crea una instancia la cual contiene los atributos esDeFiar, nivel de delincuencia del barrio, pulsación y presión sanguínea con valores aleatorios estos dos últimos. En base a estos valores generados aleatoriamente y al nivel de delincuencia del lugar de recogida se establece el valor del atributo esDeFiar. Una vez esta instancia es creada se envía toda esta información al agente junto con el lugar de recogida y de destino que el cliente desea ir.

En base a estos valores el agente deberá discernir que datos son importantes de este conjunto de 3 datos para saber si el ciudadano que desea entrar en el taxi es de fiar o no. Como consecuencia nuestro agente deberá ser capaz de obtener un modelo parecido al nuestro el cual se ve representado en la siguiente imagen para al menos discernir a las personas que son de fiar basando su decisión únicamente en sus signos vitales:



La parte coloreada de la figura superior muestra la franja de valores donde nuestro EstimadorComportamiento.cs sitúa a las personas de fiar basándose en sus signos vitales y cuya información es, obviamente, totalmente desconocida para nuestro agente.

Una vez este primer reto haya sido superado, el agente no solo deberá discernir este rango dentro del conjunto de valores máximos y mínimos que hemos ofrecido que pueden llegar a tener estas dos variables de nuestro modelo sino que ademas deberá, en función del nivel de delincuencia del punto de recogida, establecer en que casos esta información referente a los signos vitales es relevante. Es decir, la gráfica pasaría a ser algo como lo que sigue acorde a lo explicado anteriormente sobre los niveles de delincuencia y su influencia en como se detecta si una persona es de fiar o no:



Nivel de importancia de los signos vitales en función del nivel de delincuencia:

- Nivel bajo - su decisión debe basarse en los signos vitales.
- Nivel medio - su decisión se basará en los signos vitales y en el nivel de delincuencia.
- Nivel alto - su decisión debe basarse en el nivel de delincuencia.

3.6.3.Líneas futuras de investigación.

Una vez un ciudadano es creado, con perspectivas a un futuro poder otorgar de inteligencia a estos individuos, los cuales ahora mismo acorde a la teoría de sistemas agentes formarían parte del conjunto que engloba a los objetos no pudiendo llegar a la categoría de agente dado que estos ciudadanos no tienen ninguna lógica la cual les permite percibir cambios en el entorno, se ha decidido que estos actúen u operen una vez hayan sido creados por nuestro generador automático de entornos de manera independiente de tal modo que aislamos la posible inteligencia futura que estos pudiesen llegar a tener del sistema que los crea y, además, simplificamos a largo plazo la codificación

Durante el próximo capítulo se muestra como todo lo que se ha realizado durante el desarrollo de los anteriores capítulos se ve plasmado en la aplicación que hemos desarrollado dando finalmente con dicho capítulo la visualización global de todo el proyecto realizado en este Trabajo de Fin de Grado.

4. Implementación del agente autónomo.

Durante el desarrollo de los próximos sub-capítulos de este capítulo se llevará a cabo la descripción de nuestro agente autónomo. Cabe recordar que el objetivo de este agente será adaptarse al entorno, en ciertas ocasiones peligroso, al que se ve sometido realizando análisis de las personas a las que da servicio a través de factores tales como el nivel de delincuencia del barrio o aprender con qué condiciones meteorológicas es más recomendable jugar al baloncesto. Así pues, todo este entramado de habilidades, destrezas y aspectos que hacen de nuestro agente ser como su nombre indica, un agente autónomo, se llevará a cabo su desarrollo durante los próximos sub-capítulos.

4.1. Entretenimiento del agente: lanzar a canasta.

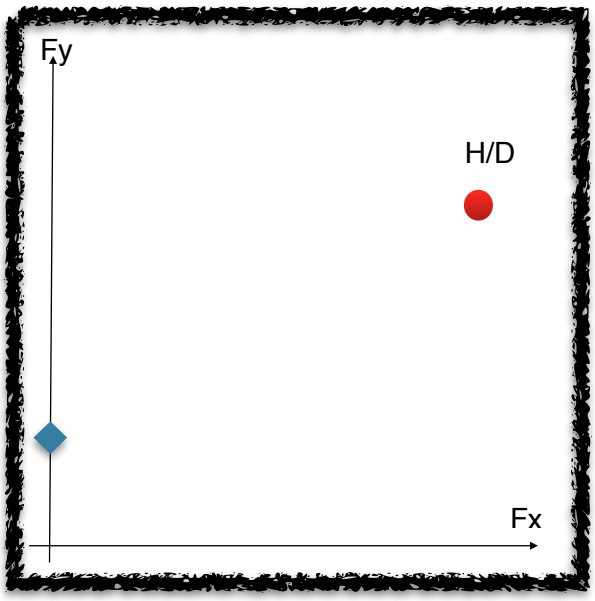
En este capítulo el objetivo es la realización de una inteligencia tal que lleve a cabo un lanzamiento a canasta del modo más preciso posible. En el desarrollo de este capítulo realizaremos la descripción de las inteligencias utilizadas, ventajas y desventajas de las soluciones usadas así como otras posibles soluciones para dar solución al objetivo del capítulo en el que nos encontramos. En particular, hablaremos de dos soluciones desarrolladas durante la realización de este proyecto. Una de estas soluciones esta basada en la interpolación de fuerzas para alcanzar una distancia u altura concreta mientras que la segunda la cual ha sido finalmente la utilizada en la aproximación orientada a alcanzar el error mínimo en el lanzamiento a canasta.

4.1.1. Aproximación orientada a la interpolación de fuerzas.

Así pues, comenzaremos por la primera aproximación que se implementó la cual parecía la más obvia y cercana a lo que una persona haría en la vida real. Esta solución esta basada en la interpolación y extrapolación de fuerzas para conseguir una distancia o una altura en concreto.

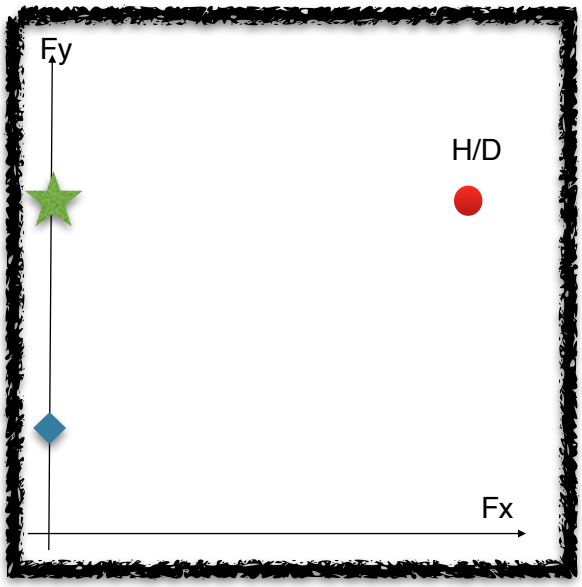
Para la realización de esta aproximación se ha llevado a cabo el desarrollo de un buscador de experiencias el cual decide en qué momento utilizar la inteligencia artificial desarrollada para el objetivo de este capítulo, el lanzamiento a canasta, para interpolar o extrapolar la fuerza necesaria para alcanzar la distancia o la altura a la que se encuentra la cesta ya sea enfocando dicha inteligencia artificial en la fuerza horizontal o en la vertical. Cabe decir, que este algoritmo solo se preocupa por la fuerza vertical u horizontal la cual tenemos que realizar para encestar ya que de por sí nuestro agente siempre que se pare mirará hacia la cesta de tal modo que no es necesario tener en cuenta la fuerza que se aplica en los 3 ejes X, Y y Z sino solo en Z e Y. Así pues, el algoritmo buscador de experiencias desarrollado seguiría el siguiente script de ejecución:

- *El mundo sería el siguiente*



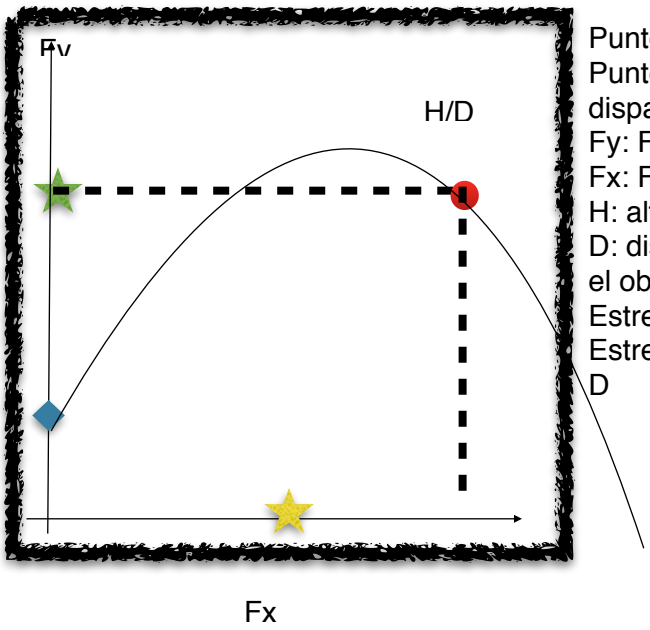
Punto rojo: punto a alcanzar (lugar de la cesta)
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Y
Fx: Fuerza aplicada sobre el eje X
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar

- *Paso 1: En nuestra implementación tendremos siempre como prioridad conseguir la altura. Por tanto, en este paso le proporcionaremos a la red neuronal una serie de fuerzas en el eje y de tal modo que una vez terminadas dichas pruebas nuestra red neuronal tenga cierta información con la que trabajar. Este ha sido sin duda uno de los puntos más importantes para el éxito de la red neuronal en esta aproximación ya que los datos iniciales así como los datos a posteriori que debían ser evaluados/supervisados para su aprendizaje eran vitales para el éxito de la red neuronal. Una vez esta información ha sido asimilada, la red neuronal ajustará el valor de la fuerza necesaria hasta alcanzar la altura H indicada.*



Punto rojo: punto a alcanzar
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Y
Fx: Fuerza aplicada sobre el eje X
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar
Estrella verde: Fy para alcanzar H
Estrella amarilla: Fx para alcanzar D

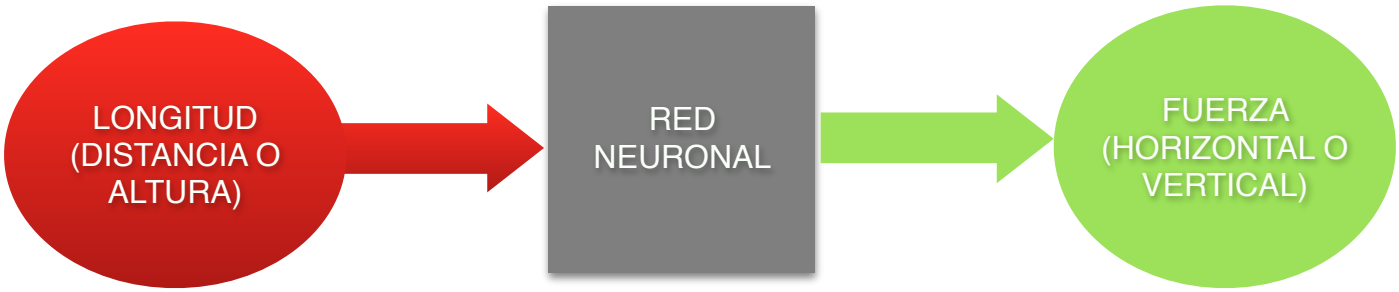
- *Paso 2: Una vez la fuerza en el eje y haya sido hallada por la red neuronal, se re-inicia la red neuronal nuevamente para que realice las mismas pruebas iniciales y ajustes que hicimos en el Paso 1 hasta alcanzar la fuerza en el eje z o x ,según que estemos usando como eje horizontal, necesaria para alcanzar la distancia D indicada dejando la fuerza en y obtenida en el anterior paso como una constante.*



Punto rojo: punto a alcanzar
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Y
Fx: Fuerza aplicada sobre el eje X
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar
Estrella verde: Fy para alcanzar H
Estrella amarilla: Fx para alcanzar D

- *Paso 3: en caso de que altura y distancia hayan sido alcanzadas las fuerzas en el eje x e y permanecerán así hasta que cambiemos algunas de las variables objetivo, ya sea la altura o la distancia. En caso contrario, es decir, en caso de que algunas de las variables objetivos no sean alcanzadas se volverá al paso 1 o al paso 2 dependiendo de la que no haya sido conseguido.*

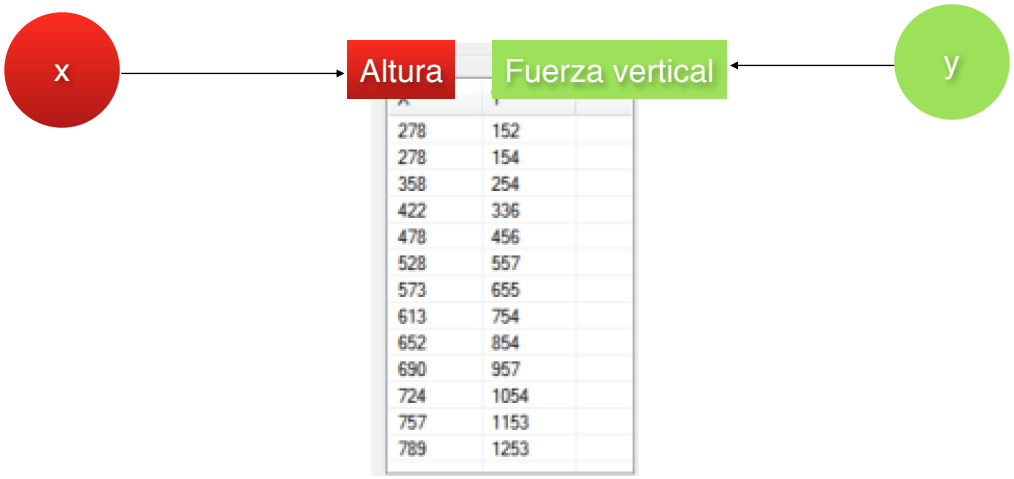
Así pues, como bien se puede deducir del script anterior existirá una única red neuronal con una única entrada y una única salida. Esta red neuronal se irá reiniciando en función de si lo que se desea optimizar es la altura o la distancia de tal modo que nos encontraríamos con una red neuronal del tipo siguiente:



Así pues, será nuestro algoritmo buscador de experiencias el que proporcione los datos a nuestra red neuronal e interprete lo que significan estos datos ya que, en definitiva, el trabajo

de nuestra red neuronal será dado un dato x, que podría representar una altura o una distancia, cual es el dato y, que podría ser la fuerza horizontal o vertical a utilizar, que habría que usar acorde a los otros datos proporcionados. Por tanto, lo que se pretende que haga en esta aproximación la red neuronal no es ni más ni menos que una interpolación y extrapolación de en base a unos datos dados para con ello en unos pocos disparos cuyos datos se suministran a dicha red neuronal fuesen suficientes para que al tercer o cuarto disparo nuestra red neuronal nos de la fuerza necesaria para alcanzar el objetivo.

Para conseguir el objetivo anteriormente propuesto se partió de una serie de datos reales obtenidos a través de la observación en el entorno de desarrollo de tal modo que pudiésemos investigar de las posibles opciones ofrecidas por la librería AForge.NET y Accord.NET cual es el mejor algoritmo para realizar las interpolaciones y extrapolaciones que el objetivo de este capítulo requieren. Así pues, el conjunto de datos del que partiremos serán los siguientes los cuales hacen referencia a la altura y a la fuerza Y que se uso en el entorno para conseguir dicha altura:



278	152
278	154
358	254
422	336
478	456
528	557
573	655
613	754
652	854
690	957
724	1054
757	1153
789	1253

Así pues, tal y como se dijo anteriormente para nuestra red neuronal los datos entrantes serán valores x e y sin sentido alguno. Será nuestro algoritmo buscador de experiencia el que decida cual es el significado de estos. En concreto el valor x podrá significar distancia u altura mientras que el valor y podrá significar fuerza horizontal o fuerza vertical.

Comenzaremos a continuación con las diferentes opciones las cuales podría hacerse uso para llevar a cabo el desarrollo de esta red neuronal cuyo objetivo es la interpolación y extrapolación de unos datos de salida, la fuerza vertical u horizontal.

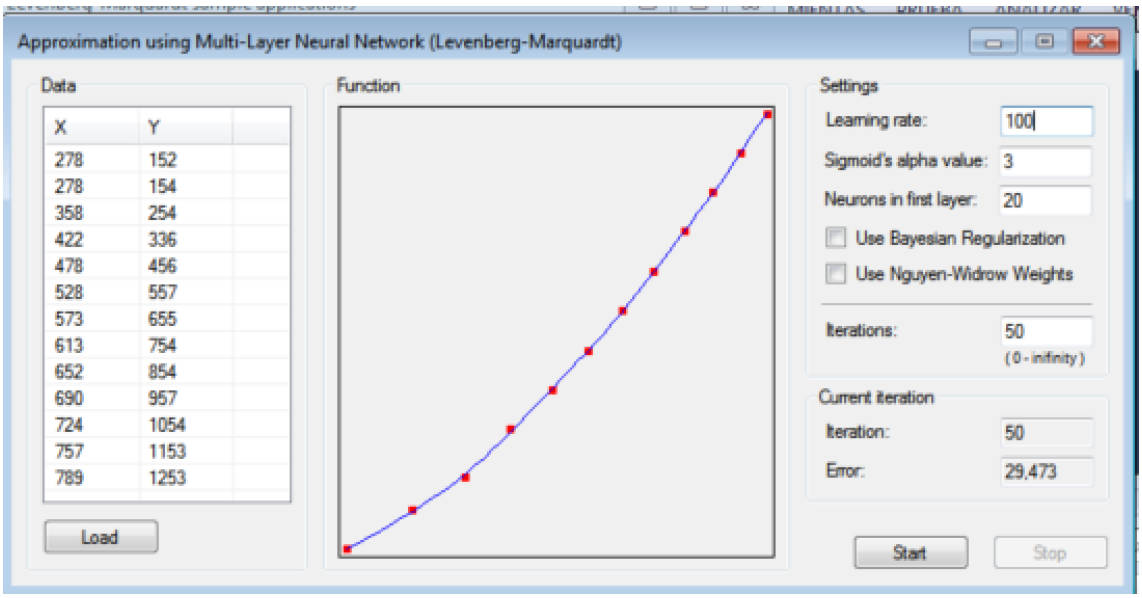
Con el objetivo de orientar al lector para el caso en que este quiera realizar los mismos pasos que se realizarán a continuación se marcarán los lugares donde se encuentran los algoritmos que se irán utilizando con el siguiente formato: Librería utilizada —> Área de desarrollo —> Algoritmo usado. De este modo, el lector puede orientarse dentro del paquete de las dos

librerías utilizadas lo cual, debido a su extensión, puede hacerse difícil dada la gran cantidad de algoritmos implementados en estas dos librerías.

Así pues, se irán indicando en el formato establecido anteriormente los algoritmos con los cuales se intentó dar solución al objetivo de este aspecto de la inteligencia artificial de nuestro agente, es decir, el lanzamiento a canasta. Se adjuntará para cada algoritmo el error cometido para el ejemplo anteriormente citado el cual se obtuvo mediante observación en el que se establecían tuplas del tipo altura-fuerza vertical así también como una gráfica de la interpolación que este algoritmo realizaría en caso de utilizarse en nuestro mundo 3D. Dicho esto, comencemos:

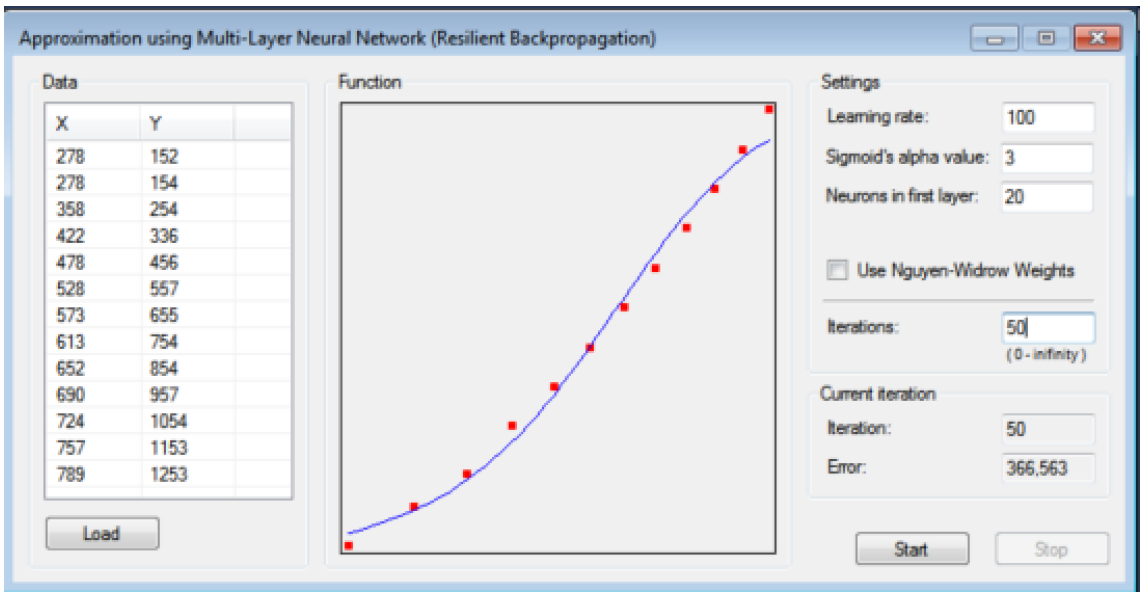
- Aproximación usando Accord.Net—>Neuro—>Levenberg-marquardt

Sin duda esta aproximación es la más viable con un error tan solo de 29.473 usando 50 iteraciones a partir de las cuales el error entra en su valor óptimo y deja de mejorar. En caso de que el lector quiera reproducir esto mismo únicamente ha de seguir el camino indicado, ejecutar el proyecto contenido en dicha ruta, con VisualStudio por ejemplo, pulsar sobre el botón Load, cargar un fichero del tipo .csv, establecer los parámetros de la red neuronal del mismo modo en que aparecen en la siguiente imagen para finalmente pulsar el botón Start obteniendo una gráfica similar a la que ve en la siguiente imagen



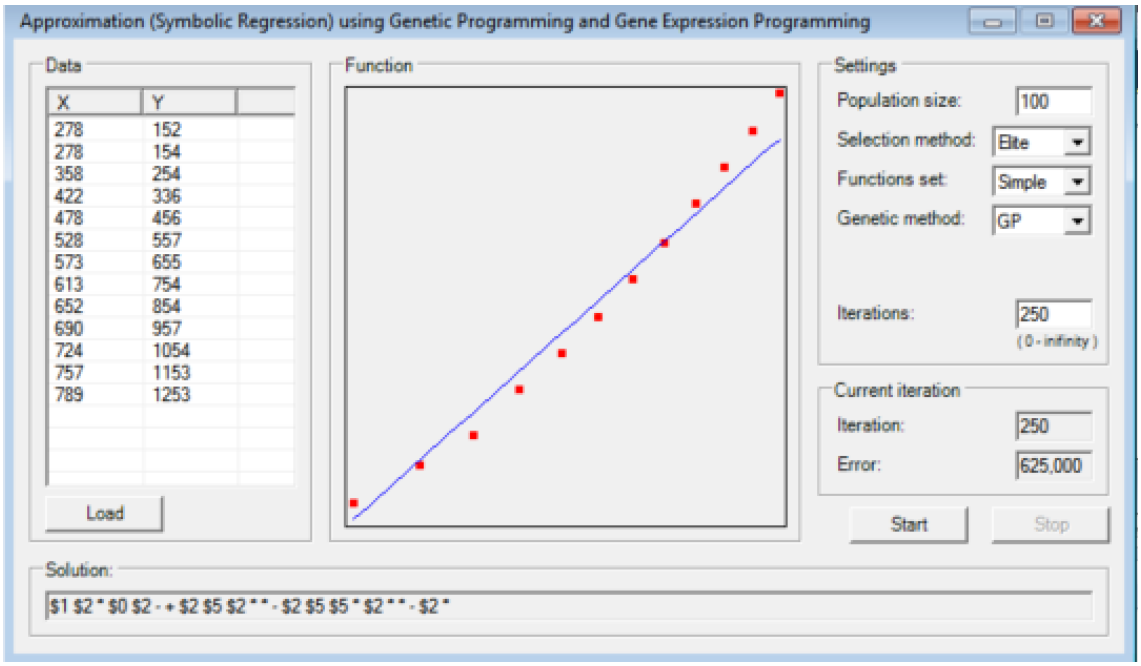
- Aproximación usando Accord.Net —> Neuro —> Resilient-backpropagration

Con el algoritmo Resilient-backpropagation podemos observar que aún pudiendo conseguir valores más o menos aceptables estos son peores para mismo número de iteraciones que con el algoritmo Levenberg-marquardt el cual es la mejor solución que hemos obtenido de entre todos los algoritmos disponibles en las amplias librerías de AForge.NET y Accord.NET. Además, el error con respecto a la mejor solución es de una diferencia de 333 como vemos en la siguiente imagen:



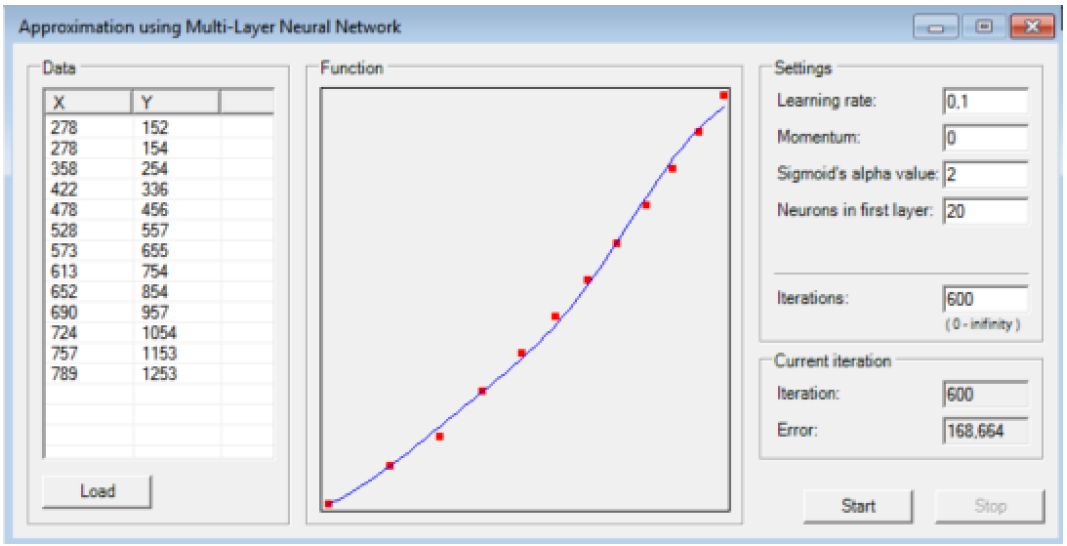
- Aproximación usando AForge.Net —> Genetic —> Approximation

Como podemos ver el error que vemos en la siguiente figura es mayor que el que conseguimos con la mejor solución con el añadido, el cual empeora su elección, de que necesitamos muchas más iteraciones que con los anteriores, es decir, descartamos esta solución por falta de precisión y exceso de tiempo:

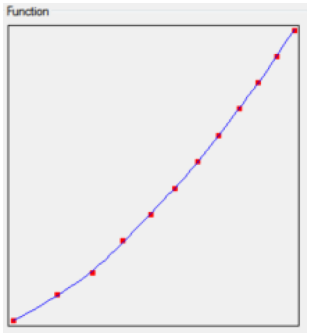


- Aproximación usando AForge.Net —> Neuro —> BackPropagation —> Approximation

Con backpropagation utilizando redes neuronales al igual que ocurría con algoritmos genéticos necesitamos más iteraciones que la mejor solución que se puede obtener, y la precisión conseguida por el error acumulado no es ni de cerca tan mínimo como el que conseguimos con la mejor solución:



Así pues, proseguiremos nuestra solución usando el algoritmo de Levenberg-marquardt para redes neuronales con el cual conseguimos los siguientes valores aceptables para la posterior integración con el algoritmo que hará uso de esta inteligencia para el lanzamiento a canasta para esta aproximación:



Así pues, quedaría explicado hasta aquí los siguientes puntos:

- Cómo funciona el algoritmo buscador de experiencias: buscando fuerzas verticales y horizontales en función del objetivo que se plantee dicho algoritmo optimizar.
- Qué uso se le va a dar a nuestra red neuronal de una entrada y una salida: misma red neuronal para nuestro algoritmo buscador de experiencias la cual interpretará unos datos para con ellos poder realizar interpolaciones y extrapolaciones a través del uso de nuestro

algoritmo buscador de experiencias el cual será, en última instancia, el que decidirá el significado de los datos proporcionados a dicha red neuronal.

- Las razones del por qué se ha decidido el uso de una red neuronal de Levenberg-Marquardt: el cual tanto por precisión así como rendimiento en tiempo de ejecución para un número de iteraciones reducido realiza una interpolación de los datos proporcionados, con respecto a los reales, un error aceptable en comparación con los otros algoritmos investigados.

Finalmente se explicará a continuación uno de los puntos claves y más problemáticos para el éxito de la red neuronal, la decisión sobre que datos mejorarían y por tanto deberían ser añadidos a los datos con los que la red neuronal debería trabajar y, por otro lado, cuales empeorarían la aptitud y por tanto deberían ser descartados. Para el proceso de selección de individuos (siendo un individuo un par Altura-FuerzaY o un par Distancia-FuerzaZ dependiendo de lo que se este optimizando como ya se ha indicado anteriormente en función de nuestro algoritmo) se observo que para el buen rendimiento de la red neuronal ésta necesitaba de un conjunto de valores entre los mínimos y máximos de fuerzas establecidos el cual estuviera lo mejor distribuido que se pudiera obteniendo con ella una respuesta más rápida al tener que realizar la red neuronal menos intentos. Así pues, para la optimización de la red neuronal a través de la selección concreta de los individuos más aptos se basa en la división del número total de individuos con los que la red neuronal va a trabajar dejando a cada individuo(posición del array) un rango del conjunto de alturas/distancias máximas y mínimas que puedan usarse. De tal modo que si tenemos que la red neuronal va a poder trabajar hasta con 5 individuos y queremos hallar la fuerza *y* para conseguir una altura H teniendo un rango de fuerzas en el eje *y* de 0 como mínimo y 1000 como máximo así como un número de 3 pruebas iniciales con las que aportar información inicial a la red neuronal, tendremos una vez realizadas estas 3 pruebas lo siguiente en la información que se le proporcionara a la red neuronal como array del cual extraerá el conocimiento inicial para posteriormente responder a la altura concreta que se este solicitando:

Posición array	Altura	Fuerza Y
Pos1_Rango[0,200]		
Pos2_Rango[201,400]	2	333
Pos3_Rango[401,600]		
Pos4_Rango[601,800]	4	666
Pos5_Rango[801,1000]	6	999

Recordar que los valores indicados en la tabla anterior son orientativos para dejar el concepto claro dado que las fuerzas usadas pueden ser modificadas.

Una vez la red neuronal tiene estas n primeras pruebas que le indicamos y que se colocan dentro del rango al que pertenezcan, se empieza a rellenar los huecos sobrantes, si es que existen, o a sustituir los antiguos por los valores nuevos y que se consideren mejores que los actuales para alcanzar la altura/distancia objetivo. Estos valores entrantes se consideraran mejores si se cumpliesen algunas de las siguientes condiciones:

- La posición del array tiene fuerza 0 (en un lanzamiento a canasta esta fuerza es inservible) por lo que será sustituida por cualquier valor entrante.
- El valor nuevo se encuentra dentro de un rango y es mejor valor que el valor actual que se encuentra en el rango. Ejemplo: si echamos un vistazo nuevamente a la tabla anterior veremos que las pruebas iniciales nos da un valor de 2 para altura 333. Imaginemos por un momento que nuestra altura objetivo es 2.4 y que para esa altura necesitamos 380 de fuerza en el eje *y*. En este caso, la red neuronal nos advierte que para acercamos a H=2.4 de altura debemos usar 370 pero lo que conseguimos realmente es 2.3. Así pues, este valor, aunque no es aun 2.4, es más cercano a 2.4 que la altura de 2 que conseguíamos con una fuerza de 333 y, por tanto, la posición 2 de nuestra tabla anterior sería sustituida por lo siguiente:

  Pos2_Rango[201,400] —> (Altura=2.3, FuerzaY=370)

  De este modo, mantenemos la información proporcionada a la red neuronal lo más óptima posible con el objetivo de no cebar a la red neuronal con datos muy cercanos a la altura objetivo sino que se mantiene un rango útil de posibles fuerzas en caso de que se cambie dicha altura dejando a cargo de un rango de fuerzas a una posición del array. Así la información que se le pasa a la red neuronal es siempre óptima y su respuesta lo más rápida posible lo cual es de particular interés ya que cada lanzamiento implica una pérdida de tiempo de entorno a 1 segundo o más dependiendo de las fuerzas que se apliquen.

La ventaja de esta red neuronal es que al ser solamente evaluada una sola variable del problema global a la vez la solución que obtendremos es muy rápida pero necesitamos realizar la división en el código de dos partes, una para evaluar la fuerza horizontal y otra para la fuerza vertical lo que lleva a la creación de un código bastante extenso aunque fácil de leer. No obstante, la realización de esta aproximación englobando todas estas operaciones las cuales midiesen, controlasen y ajustasen las fuerzas aplicadas así como el movimiento de la pelota hace que sea difícil de usar este código para tareas de otro índole. Esta última desventaja hizo que se descartase esta aproximación debido a que no sólo se pretende alcanzar la solución más óptima cuando se realizó el desarrollo de la inteligencia artificial de este agente sino también que la misma inteligencia fuese desarrollada de manera lo más abstracta posible para que pudiese ser usada en otros aspectos del agente. Así pues, queda descartada

esta aproximación que a pesar de ofrecer solución óptimas en un corto periodo de tiempo de unos 5 disparos de media reside en ella una complejidad muy alta para poder ser aplicada de manera sencilla a otros aspectos del agente por su carencia de abstracción.

4.1.2. Aproximación orientada al error mínimo.

Nuestra segunda aproximación y la solución la cual se usará para la realización de este aspecto de la inteligencia artificial de nuestro agente se basa en la realización de una única red neuronal la cual tiene dos entradas, siendo éstas las fuerzas en el eje *z* y en el eje *y* utilizadas, y una salida, siendo ésta el error producido por dichas dos entradas que en este caso son las fuerzas usadas.

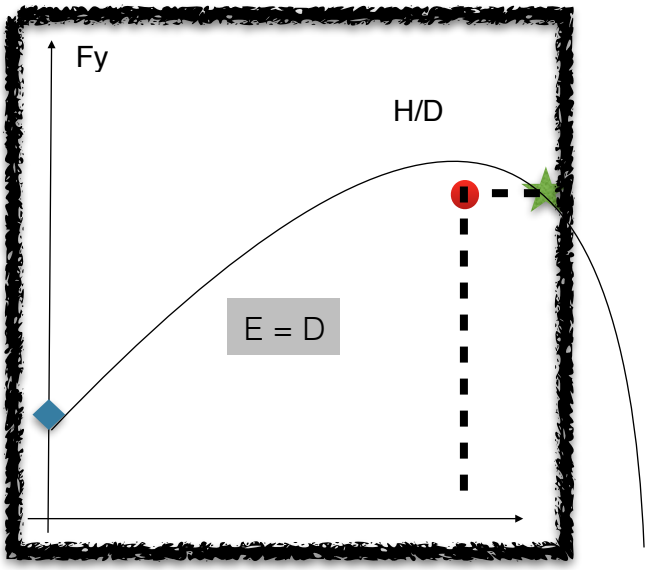
Así como ocurrió en la anterior solución, no es necesario llevar a cabo la búsqueda en el eje *x* dado que nuestro personaje va a mirar siempre hacia la canasta por lo que el eje local *z* apuntará en todo momento hacia dicha cesta y no será necesario, como consecuencia de ello, tener en cuenta el eje *x*.

Para el desarrollo de esta inteligencia se ha realizado la creación de una clase denominada Aproximacion.cs a través de la cual podemos crear una red neuronal ajustada a nuestras necesidades, es decir, podemos elegir desde el número de entradas y salidas hasta cual es el valor sigma utilizado para la red neuronal.

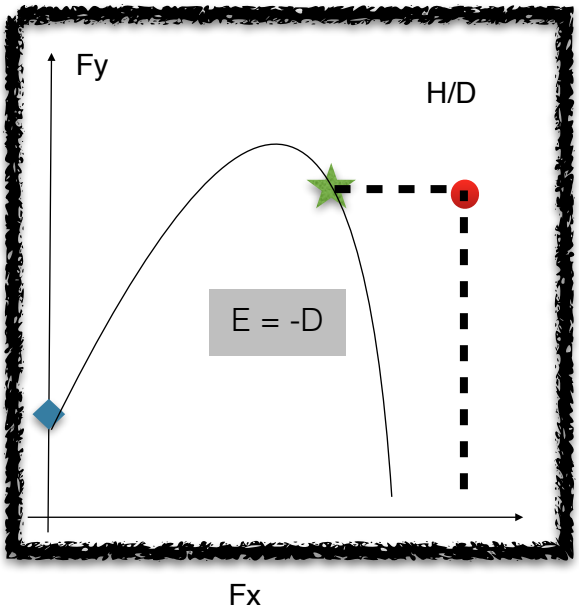
Así pues, con el desarrollo de una red neuronal a través de esta clase con dos entradas y una salida podemos separar nuestra red neuronal del lugar donde conseguimos los datos. Para conseguir los datos llevamos a cabo el uso de otra clase denominada ControlBola.cs la cual mide el error que la bola produce al dispararse con las fuerzas en el eje *z* e *y* sugeridas por la red neuronal de Levenberg-Marquardt usada.

Pasaremos ahora a explicar como se ha realizado el calculo del error para el lanzamiento de la pelota. Dado el lanzamiento de una pelota podemos encontrarnos con las siguientes posibilidades:

- La pelota ha superado la altura mínima para encestar: en este caso el cálculo del error vendrá dado por la distancia a la altura del plano de la pelota a la cesta. Además, dependiendo de si hemos pasado de largo la cesta o no el error será negativo o positivo como bien se indica en al siguiente imagen:



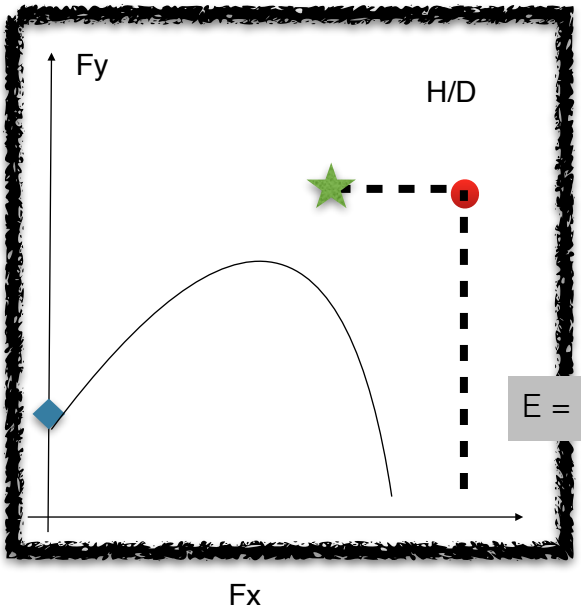
Punto rojo: punto a alcanzar
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Y
Fx: Fuerza aplicada sobre el eje X
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar
Estrella verde: distancia a la cesta



Punto rojo: punto a alcanzar
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Vertical
Fx: Fuerza aplicada sobre el eje Horizontal
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar
Estrella verde: Fy para alcanzar H
Estrella amarilla: Fx para alcanzar D

Así pues, el error rango de valores en los que se moverá el error en caso de que haya superado la altura será [-(longitud de la cancha de baloncesto), + ∞].

- La pelota no ha superado la altura mínima para encestar: en este caso el error será negativo y vendrá dado por la suma de la longitud máxima de la cancha de baloncesto más el alfa provocado por el ángulo de las fuerzas utilizadas



Punto rojo: punto a alcanzar
Punto azul:Posición inicial de disparo
Fy: Fuerza aplicada sobre el eje Vertical
Fx: Fuerza aplicada sobre el eje Horizontal
H: altura que necesitamos alcanzar
D: distancia a la que se encuentra el objeto a alcanzar
Estrella verde: Fy para alcanzar H
Estrella amarilla: Fx para alcanzar D

E = - (Longitud cancha) - (α de las fuerzas)

Por tanto, cuando obtengamos un error como el anterior nuestro rango vendrá dado entre [-∞, -(longitud de la cancha de baloncesto)).

Hasta aquí habríamos desarrollado nuestro algoritmo buscador de experiencias en el que establece que según si la altura ha sido superada por la bola o no nos encontraremos en diferentes rangos de error. Será pues, este algoritmo, el encargado de interpretar los errores que la red neuronal nos proporcione en base a unas pruebas iniciales aleatorias que le hayamos proporcionado anteriormente para que con ello podamos obtener un lanzamiento a canasta certero.

Para comprobar que realmente se ha encestado se ha establecido un collider debajo de la red el cual ha sido usado a modo de sensor de presión de tal modo que si la bola se posa sobre él sabremos que no sólo hemos conseguido obtener un error menor que el establecido sino que además la bola ha pasado literalmente por la red.

Por la abstracción de esta solución así como su simplicidad en el uso de otras posibles futuras implementaciones de aspectos de la inteligencia artificial asociada a nuestro agente se ha elegido esta solución la cual a pesar de no ofrecer resultados en un periodo de tiempo tan corto como el que obtenemos con la solución anteriormente mencionada permite una mayor libertad cuando se programa debido a que basa su rendimiento en un error y que, al igual que en un algoritmo heurístico, dependiendo de cuán bueno sea la formula usada para ese cálculo así de bueno será la implementación que realicemos y por tanto los resultados que

obtengamos. Además, con este algoritmo podemos conocer o, más bien, podemos plantear estados a nuestro algoritmo buscador de experiencias para que los controle en base al error proporcionado. Esto último queda descrito por estados tales como los anteriormente citados entre los que figuran:

- Un error en negativo menor que la (-longitud de la cancha de baloncesto) implica que nos hemos quedado cortos de fuerza vertical.
- Un error positivo que hemos pasado la altura pero que nos hemos pasado con la fuerza horizontal.
- Un error negativo pero mayor que (-longitud de la cancha de baloncesto) implica que hemos pasado la altura pero que nos hemos quedado cortos con la fuerza horizontal.
- Otros posibles: un error muy pequeño podría significar que aunque estemos pasando la bola muy cerca de la cesta le estamos dando tan fuerte que esta rebotando en el panel y no esta entrando en la red por lo que no toca el sensor. En este caso, podríamos usar el error para aumentar la fuerza vertical y disminuir la fuerza horizontal.

Por otro lado, esta solución plantea un problema grave de rendimiento y es que una vez incorporamos todos los datos debemos iterar sobre cada una de las posibilidades de las fuerzas horizontales y verticales para obtener el error de menor valor. Esto evidentemente es un problema grave ya que si nos movemos en limites de fuerzas grandes el obtener el error de menor valor puede suponer una tarea de incluso minutos. Por ello, como solución a este problema se ha utilizado el concepto cluster para la optimización de modo que evitamos esa iteración de N contra M en la que probamos cada una de las N fuerzas verticales contra cada una de las M fuerzas horizontales para obtener el error menor. Así pues, si tenemos unas fuerzas las cuales se mueven en el eje horizontal de 0 a 1000 y en el vertical de 0 a 1000 dividimos ambos rangos en el número de clusters que se indiquen, por ejemplo en 10 como vemos en la siguiente figura:

Fuerza horizontal	[0, 100]	[101, 200]	[201, 300]	[301, 400]	[401, 500]	[501, 600]	[601, 700]	[701, 800]	[801, 900]	[901, 1000]
Fuerza Vertical	[0, 100]	[101, 200]	[201, 300]	[301, 400]	[401, 500]	[501, 600]	[601, 700]	[701, 800]	[801, 900]	[901, 1000]

Una vez divididos se prueba cada combinación de los clusters de las fuerzas horizontales y verticales cogiendo la fuerza media de cada cluster. Dependiendo de cual sea la combinación ganadora, es decir, con la que obtengamos un menor error pasaremos a adentrarnos más y más en cada iteración dentro de ese conjunto de clusters. Por consiguiente, imaginemos que hemos seleccionado como fuerza horizontal 50 y como fuerza vertical 950 obteniendo el menor error de las 10x10 posibles combinaciones. De este modo, en la siguiente iteración el siguiente cluster que se analizará será el cluster 1 de las fuerzas horizontales y el 10 de las fuerzas

verticales. Es decir, pasaríamos a realizar el mismo proceso pero para la siguiente división de las fuerzas:

Fuerza horizontal	[0, 10]	[11, 20]	[21, 30]	[31, 40]	[41, 50]	[51, 60]	[61, 70]	[71, 80]	[81, 90]	[91, 100]
Fuerza Vertical	[900, 910]	[911, 920]	[921, 930]	[931, 940]	[941, 950]	[951, 960]	[961, 970]	[971, 980]	[981, 990]	[991, 1000]

De este modo, controlamos el tiempo ya que podemos establecer cuántas iteraciones realizar de este proceso así como el número de clusters a realizar en cada iteración evitando el grave problema de rendimiento que sería hacer un N contra M comprobando el error de la red neuronal para cada combinación. Tras terminar todas las iteraciones la fuerzas elegidas serán las mejores hasta el momento, es decir, con las que se ha conseguido un error esperado menor ya que el hecho de que la red neuronal espere que el error sea 0.5 no implica que éste sea el error real. Esto es un aspecto que se va mejorando con cada iteración en la que se aprende de cada error, es decir, cada vez que la red neuronal esperaba un error pero este es mayor o menor.

Queda decir que se anota como un tanto cuando el error con el que encestamos es menor que un máximo establecido y se toca la base de un objeto circular situado debajo de la red asegurando de este modo que realmente se ha hecho un tanto en la canasta. Este objeto circular hace las veces de sensor de modo que sabemos exactamente si la bola ha conseguido pasar el aro o no tal y como se ha explicado anteriormente. Además, de este modo se controla que no sólo vale con que el error sea inferior al establecido sino que realmente el agente enceste literalmente la canasta.

4.1.3.Otras posibles soluciones.

Otras posibles soluciones las cuales son más complejas de realizar o con más desventajas serían las siguientes:

Posible solución 1	Altura	Distancia	Fuerza en Y	Fuerza en X

En esta solución necesitaríamos muchos más individuos que 5 para poder tener un conjunto con el cual la red neuronal pudiera trabajar y dar valores consistentes o, en cualquier otro caso, reducir el rango de fuerzas a poder utilizar lo cual limitaría nuestras posibilidades en caso de realizar tiros más largos/cortos a posteriori.

Posible solución 2	Altura=constante (habría que hallarla a ojo o usando una IA para calcular con un entrada y una salida como la que hemos explicado)	Distancia	Fuerza en Y	Fuerza en X

En esta solución solamente pasaríamos a la red neuronal valores cuando alcanzase la altura que nosotros deseamos para lo cual, en principio, se precedería de dar una ayuda un poco más precisa que quitaría autonomía en el lanzamiento a nuestro agente.

En el próximo capítulo se entrará en profundidad en cómo esta actividad recién explicada que nuestro agente realiza se puede ver condicionada por los factores meteorológicos.

4.2.Análisis meteorológico.

En este capítulo del proyecto enfocaremos nuestra atención a la realización de un control meteorológico según el cual nuestro agente deberá condicionar las acciones que este realizando en función del estado de este control meteorológico. En particular, el agente verá condicionado su juego en la cancha de baloncesto en función del tiempo meteorológico. Así pues, como en la vida real podría ocurrir, carece de sentido en una pista de baloncesto outdoor estar jugando ya que factores meteorológicos tales como la lluvia, la nieve o el viento podrían interferir en el juego de manera negativa, tanto en el movimiento del balón como en el de sus jugadores.

De este modo, el agente tendrá que aprender que condiciones meteorológicas son las oportunas para la acción en concreto a realizar. Por tanto, se establece que para jugar en una

cancha de baloncesto no cubierta partiendo de la base de que tendríamos los siguientes factores meteorológicos posibles en nuestro mundo 3D: soleado, lluvioso, nevado. La situación ideal para poder jugar al baloncesto sería que estuviese soleado, no lluvioso y no nevado. Este estado ideal es el que nuestro agente debe alcanzar por sí mismo. Cabe decir que para esta acción en concreto, jugar al baloncesto, el estado ideal meteorológico que se ha proporcionado ha sido este pero podría ser cualquier otro.

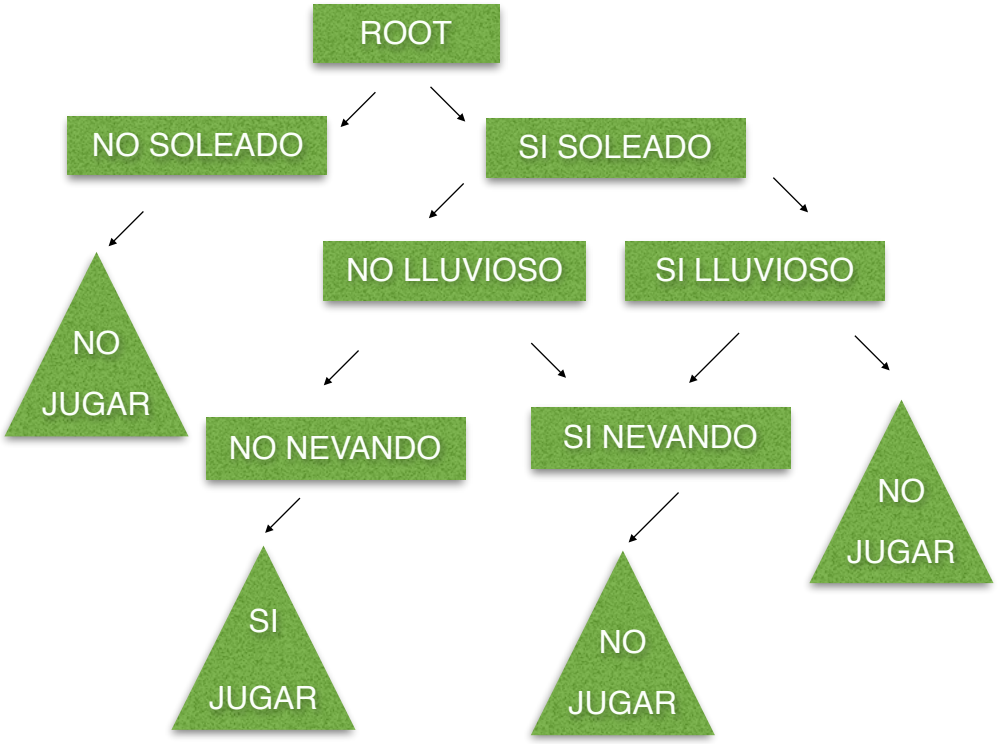
Así pues, nuestro personaje deberá aprender que el estado ideal de entre las 8 combinaciones posibles del tiempo meteorológico solo 1 de ellas es la correcta para poder jugar al baloncesto. Para ello se ha hecho uso de la minería de datos definida esta última como campo de las ciencias de la computación referido al proceso que intenta descubrir patrones en grandes volúmenes de conjunto de datos. Más en profundidad dentro de este campo, se ha hecho uso de uno de los modelos de predicción utilizados dentro de este ámbito conocidos como arboles de decisión. Siendo más exactos en la ejecución de nuestro árbol de decisión se ha hecho uso de algoritmo C4.5 para generar el árbol de decisión, cuyo algoritmo es una extensión del ya existente ID3, el cual incluye algunas mejoras como el manejo de atributos continuos y discretos, podando el árbol después de la creación, manejo de atributos con costos diferentes y manejo de datos con valores de atributos que, por alguna razón, se deseen marcar como faltantes de tal modo que estos atributos faltantes no son usados en los cálculos de la ganancia y la entropía.

A lo largo del proyecto me he topado con ciertos inconvenientes o problemas de diferente índole relacionados con el desarrollo de la inteligencia. Uno de ellos ha sido la cantidad de datos a almacenar o, dicho de otro modo, el número de ejemplos máximo con los que los diferentes algoritmos utilizados durante el desarrollo de las inteligencias artificiales podían contar para desarrollar su aprendizaje. En este caso en particular cómo almacenar, cuánto y qué cantidades no ha sido un problema debido a la sencillez de los posibles 8 casos que se nos presenta ante este problema. Así pues, con un mínimo de 8 ejemplos o datos tipo como los que vemos a continuación:

double[ESTADO SOL, ESTADO LLUVIA, ESTADO NIEVE, ESTADO PLAY]

se ha podido conseguir sin ningún tipo de problema y almacenando los casos según surgían en los que el árbol de decisión fallaba ya fuera por el hecho de que no tenía ningún dato que hiciese referencia a este caso o en el caso de que se pusiese más de 8 ejemplos máximos los cuales no serían necesarios en principio porque no contuviese una cantidad importante de este estado como para aprenderlo de manera contundente.

Como consecuencia del uso de este árbol de decisión lo ideal sería que se obtuviese un árbol similar al siguiente:



Hasta este punto hemos explicado ya uno de las acciones condicionadas que realiza nuestro agente basada en el juego dentro de una cancha de baloncesto en función de las condiciones meteorológicas. En el próximo capítulo se realizará la descripción de la próxima acción condicionada de nuestro agente basada en la recogida de clientes con un taxi y condicionar su entrada en base a factores tales como presión sanguínea o pulsaciones que pudiesen indicar un estado no apto para entrar a un taxi y por cuyos factores la entrada sería prohibida o permitida a un cliente en un taxi. Cabe recalcar como comentario que esto es algo que podríamos realizar en la vida real ya que según la ley un taxista esta permitido a no ofrecer un servicio en caso de que el cliente al que vaya a dar dicho servicio presente síntomas relativos a haber consumido drogas o a tener una cantidad importante de alcohol en sangre.

4.3.Trabajo del agente: taxista.

Durante este capítulo y el próximo hablaremos del aspecto laboral que nuestro agente realiza dentro del mundo 3D. Este aspecto laboral se ve representado bajo el rol del manejo de un taxi el cual nuestro agente dirige recogiendo clientes y llevándolos al lugar de destino que estos deseen. Como detalle el cual se entrará más en profundidad más adelante actualmente todos los aspectos del mundo 3D son realizados de manera independiente, es decir, los clientes generados así como sus estados de animo, tiempo meteorológico y demás factores los cuales son percibidos por nuestro agente de manera autónoma, actuando éste mismo en una continuidad en el tiempo, son controlados por una entidad totalmente independiente a la inteligencia del agente en sí mismo de manera que se esta realizando a través de este proyecto un mundo en el cual se esta simulando casos reales dados en la vida real. Nuestro agente recibe por tanto a través de mensajes de esta entidad independiente dentro del mundo 3D información de clientes actuales y de entre ellos nuestro agente va eligiendo según estos van llegando. Así pues, comentado lo dicho nuestro agente dará servicio como si de una FIFO se tratase, First Input First Output, recogiendo de la estructura de datos proporcionada en el mensaje la información la cual hace referencia a la posición en la que hay que recoger al cliente y a donde hay que llevarlo.

Una vez la entidad independiente anteriormente descrita ha creado el mapa, nuestro agente consulta si este mapa ha sido definitivamente creado y realiza una copia del mismo posicionándose en un punto inicial, eligiendo a un primer cliente y obteniendo el punto de recogida de dicho cliente. Una vez esto ha sido realizado nuestro agente debe llevar a cabo el uso de un algoritmo para ir desde el punto en el que se encuentra al punto donde se encuentra el cliente. Este algoritmo es el algoritmo A* del cual se ha hecho una desarrollado una implementación propia al haber hecho uso de la estructura anteriormente citada la cual hace uso de la clase PuntoConexion.cs. Una vez el algoritmo A* ha encontrado un camino para llegar al punto objetivo desde la posición actual, no necesariamente el óptimo como se dijo anteriormente, usaremos este camino en el mundo 3D donde el agente ira recorriendo punto por punto el camino que se le pasa como salida del algoritmo A*.

Para dejar claro que lo que se esta realizando es recoger y soltar clientes en puntos del mapa se colocará a una entidad al lado del coche de tal modo que se experimenta un mínimo de esa sensación. No obstante, no es en ningún caso el objetivo de este TFG realizar un mundo 3D donde los gráficos o animaciones sean de vital importancia sino la inteligencia que el agente ejecuta sobre este mundo.

Se había pensado en la posibilidad de realizar la búsqueda de camino de un punto A a otro B dentro de esta estructura a través del algoritmo de enjambre conocido como algoritmo de optimización de colonia de hormigas el cual tuvo que ser descartado ya no sólo porque no

cumpliría el propósito de que el camino debe de buscarse a través de una búsqueda única ya que en el algoritmo colonia de hormigas se precisa de más de un individuo en principio para obtener soluciones apropiadas sino que además suponía un exceso de carga de trabajo muy grande dentro del modo de funcionamiento de Unity el cual realiza toda su ejecución en un único thread, un Main, no existiendo la posibilidad de lanzar threads independientes que pudiesen minimizar esta teórica sobrecarga que se daría para mapas grandes. Además, como inconveniente añadido al rendimiento del algoritmo colonia de hormigas dentro de un mapa grande podría derivar, en caso de una mala heurística en las hormigas, en la posibilidad de ni si quiera encontrar una solución al recorrido de un punto A a un punto B lo cual no se presenta con el algoritmo A* que por muy ineficiente que este camino pueda ser al menos obtendremos una solución. Este último detalle fue vital ya que es esencial que el agente, ante cualquier problema que se le plantee dentro de este mundo, encuentre una solución medianamente eficiente o, si es posible, la más eficiente.

Hasta aquí ya conocemos bastante sobre el agente, su afición condicionada por el tiempo meteorológico que es jugar al baloncesto así como su manera de moverse por el mundo 3D y como se relaciona a través de mensajes con otros personajes y digo bien, personajes, ya que los ciudadanos hasta ahora citados los cuales se mueven por la ciudad no gozan del mismo grado de inteligencia artificial que nuestro agente. Durante la lectura del siguiente capítulo el lector conocerá cómo evalúa nuestro agente durante el tiempo que esta trabajando qué clientes poseen un estado estable a los que prestar servicio y cuáles no.

4.4.Análisis de peligrosidad de clientes.

En este capítulo se tratará el desarrollo del aspecto más humano se podría decir que el agente realiza en el que éste analiza cuando una persona es de fiar o no.

Para el desarrollo de este aspecto se ha llevado a cabo el uso de una red neuronal de Levenberg-Marquadt la cual contiene tres entradas y una salida. El motivo de estas entradas ha sido debido a que el modelo utilizado de peligrosidad, como bien se indica en el capítulo sobre la creación del modelo de nivel de peligrosidad de los ciudadanos, contiene tres parámetros principales para establecer si un ciudadano es de fiar o no. Estos parámetros son el nivel de delincuencia del barrio en el que se va a recoger al cliente así como la presión sanguínea y pulsaciones del cliente que se va a recoger. El parámetro de salida de esta red neuronal será un valor fijado por 1 o 0 que indicará si la persona es de fiar o no, respectivamente.

Así pues, cuando vayamos a recoger al cliente el cual nos envío toda su información a través de esta red neuronal utilizando el algoritmo de Levenberg-Mardquadt nuestro agente deberá discernir si es una persona de fiar o no.

Obviamente en un principio la decisión de esta inteligencia no es determinante cuando se monta a alguien en un taxi ya que en última instancia quien tiene la decisión es el taxista pero puede servir como método para conductores o taxistas poco experimentados con poco conocimiento de la zona para ayudarles a conocer que rutas o zonas, en base a históricos relacionados con atracos, niveles de violencia u otro tipo de crímenes, son menos recomendables. Se necesitan de muchos más datos para poder estimar si una persona es de fiar basando nuestra decisión únicamente en datos puros y duros sin tener en cuenta detalles tales como las apariencias que el ser humano lleva a cabo de manera automática sin darse cuenta. No obstante, se ha intentado llevar el concepto de una manera humilde a una situación tan cotidiana como es el montarse a un taxi de cómo la recopilación de datos y su tratamiento a través de la minería de datos o la toma de decisiones mediante algoritmos relacionados con el área de la inteligencia artificial podrían ser aplicados para sino hacer una sociedad más segura al menos una sociedad más consciente de los peligros que la rodean.

Hasta aquí se habría terminado todos los comportamientos de los cuales el agente goza y que hacen de él un todo. Durante el próximo capítulo se tratarán los resultados donde mostraremos la pincelada final de este proyecto con imágenes del agente funcionando, de las diferentes partes de la interfaz gráfica así como el diseño UML de los scripts de mayor importancia para el desarrollo del proyecto.

5. Resultados.

Este capítulo se mostraran screenshots de todo el proyecto como pincelada final para que el lector de este escrito tenga una visual global de todos los puntos que se han ido detallando.

Así pues, dividiremos esta parte del documento en dos, una para la escena del lanzamiento a canasta y otro para la escena del taxista.

Comenzaremos por la escena del taxista donde nos encontramos con una primera imagen como la siguiente al inicializar el sistema:

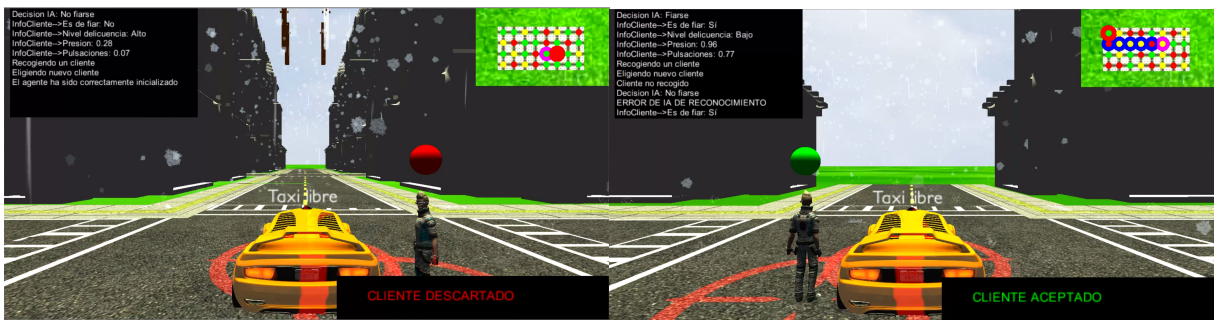
Al inicio de esta escena el agente se inicia en una posición dentro del mapa creado y espera a que llegue información de un cliente al que ir a recoger:

Ofreciendo en todo momento la información más relevante que el agente esta realizando en un mensaje de alerta más grande abajo a la derecha:

Estos mensajes pueden ser los siguientes en principio:

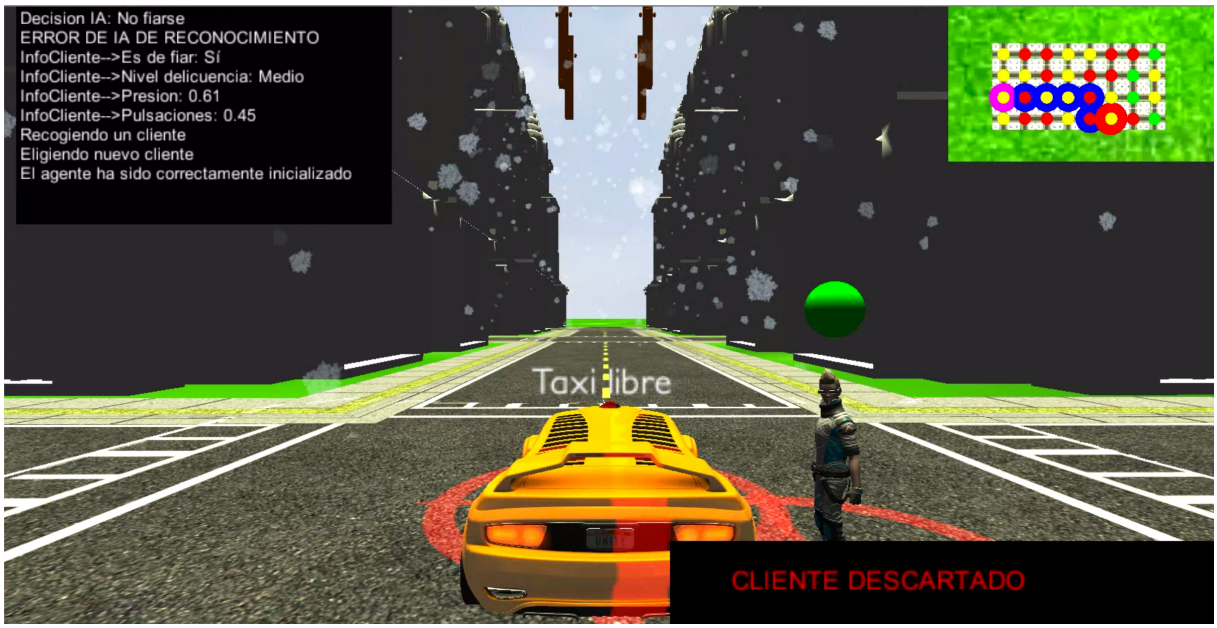
- Recogiendo cliente
- Cliente aceptado
- Cliente descartado
- Horas laborales completadas, este último se ha establecido para marcar una cota temporal bajo la cual el agente/taxista no puede estar trabajando más horas de las 8 horas establecidas.

Para una mayor visualización por parte del usuario que visualiza el sistema con el objetivo de no tener que estar leyendo la consola con todos los datos que se esta teniendo en cuenta por parte de la inteligencia sino simplemente basarse en el mensaje situado abajo a la derecha en letras mayúsculas junto con la visualización del propio programa se ha hecho que bajo el ciudadano que va a ser recogido situar una bola. El color de esta bola indica si esta persona es de fiar o no, siendo una bola verde una persona de fiar y una bola roja una persona de la que no fiarse. De este modo viendo el color de la bola sabemos el resultado que debería darse:



Como se dijo anteriormente la decisión de la inteligencia no es decisiva pero dado que esta es una simulación si el personaje se equivoca se llevará su equivocación a cabo, aprendiendo de

ella, es decir, es posible que alguno de los clientes de los que se deba fiar el agente no lleguen a su destino como puede verse en la siguiente imagen:



Como último detalle a destacar de esta interfaz hacer notar que los niveles de delincuencia según el punto de recogida se han recogido de dentro de un minimapa arriba a la derecha de modo que como ya se ha dicho antes no es necesario para el usuario leer nada, simplemente viendo el minimapa y el mensaje de alerta puede conocer cuales son las decisiones que debería tomar la inteligencia artificial del agente. Así pues, se ha marcado en el minimapa la información de delincuencia como sigue:

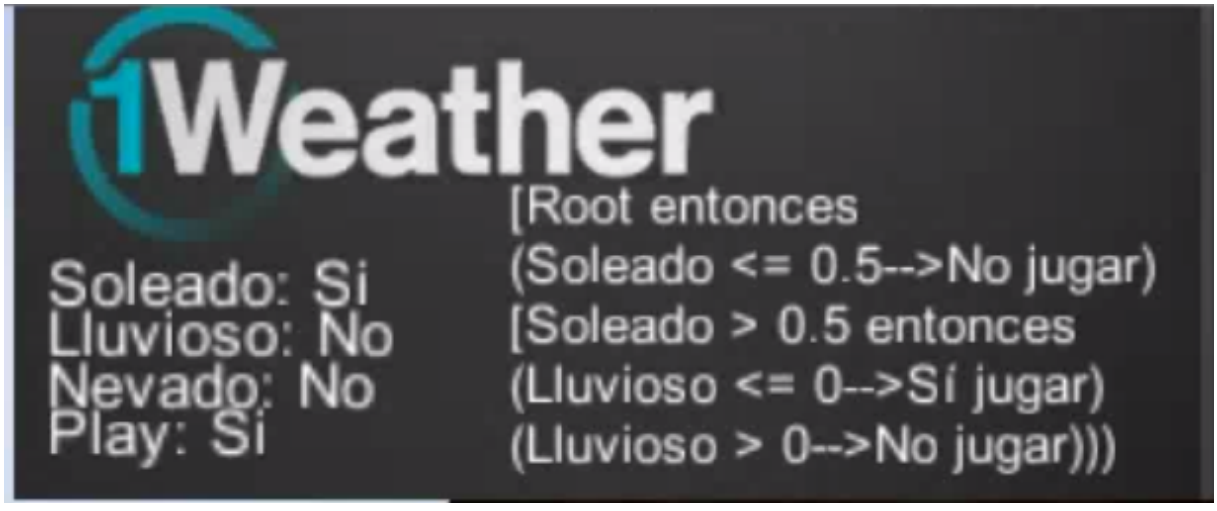
- Color rojo para zonas de nivel de delincuencia alta.
- Color amarillo para zonas de nivel de delincuencia media.
- Color verde para zonas de nivel de delincuencia bajo.

Del mismo modo, se ha usado el mismo minimapa para poder hacer ver al usuario cual es el camino que, a través del algoritmo A* implementado, se va a realizar por el agente marcando el punto rosa como el punto inicial desde el que se parte y el punto rojo más grande el punto en el que nos encontramos así como los puntos azules el camino que se va a seguir:

Con esto se finalizaría una de las dos escenas a mostrar. La siguiente escena trata sobre nuestro agente ejecutando el rol de jugador de baloncesto, ya que decidimos en su momento llevar a cabo una red neuronal de 2 entradas y 1 salida solo se mostrará la interfaz gráfica que se realizo para esta solución. Lo que veríamos dentro de este rol nada más iniciar dicha escena sería lo siguiente:

Es decir, en primera instancia no habría mucho movimiento por parte del agente exceptuando la inicialización de cual es el estado meteorológico y en base a dicho estado cual es la decisión que el agente debe tomar, jugar o no jugar:

En este cuadrado arriba a la derecha debemos tener en cuenta que aparece más información que solo un simple si o no. Entre esta información encontramos cual es el estado meteorológico actual, la decisión de la inteligencia artificial del agente basada en un árbol de decisión así como el árbol de decisión actual en el que ha basado su resultado el agente:



Una vez el agente ha realizado este primer análisis decisivo para la acción a realizar que es la de jugar a baloncesto. El personaje elige una posición dentro de la cancha de baloncesto de manera aleatoria y se le ofrece la distancia y altura a la que se encuentra la cesta. A través de la red neuronal de la cual se ha hablado largo y tendido basada en dos entradas, compuestas por la fuerza horizontal y vertical usadas, y una salida, la cual indicaba el error realizado al

usar dichas fuerzas, se pregunta a dicha inteligencia con cual fuerzas se produce el menor error, se usan dichas fuerzas y se registra el error producido realizando esto hasta que se encuentren las fuerzas con las cuales se enceste ya que no solo basamos un punto en que se cumpla un error menor del establecido sino que además la pelota debe tocar un 'sensor' dentro de la cesta plasmando todo este proceso en la siguiente imagen:

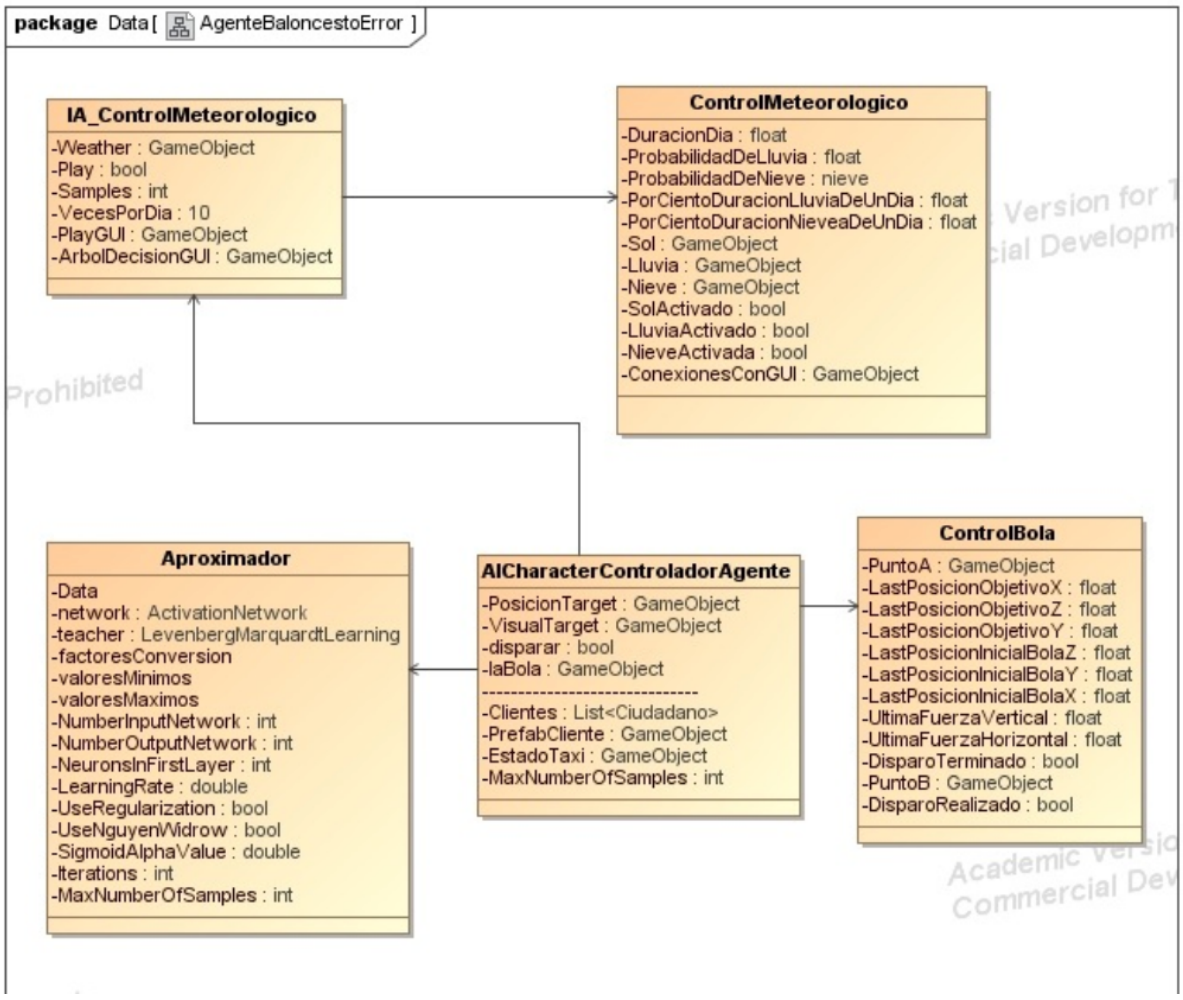
Si se ha conseguido un error menor que el aceptable y se ha tocado dicho sensor aparecerá en la interfaz gráfica lo siguiente:

En el caso de que la meteorología no sea la apta para jugar al baloncesto se producirá un mensaje el cual alertará al usuario de dicho estado y a continuación el sistema procederá a irse a la escena del taxista. Este mensaje se ve a continuación:

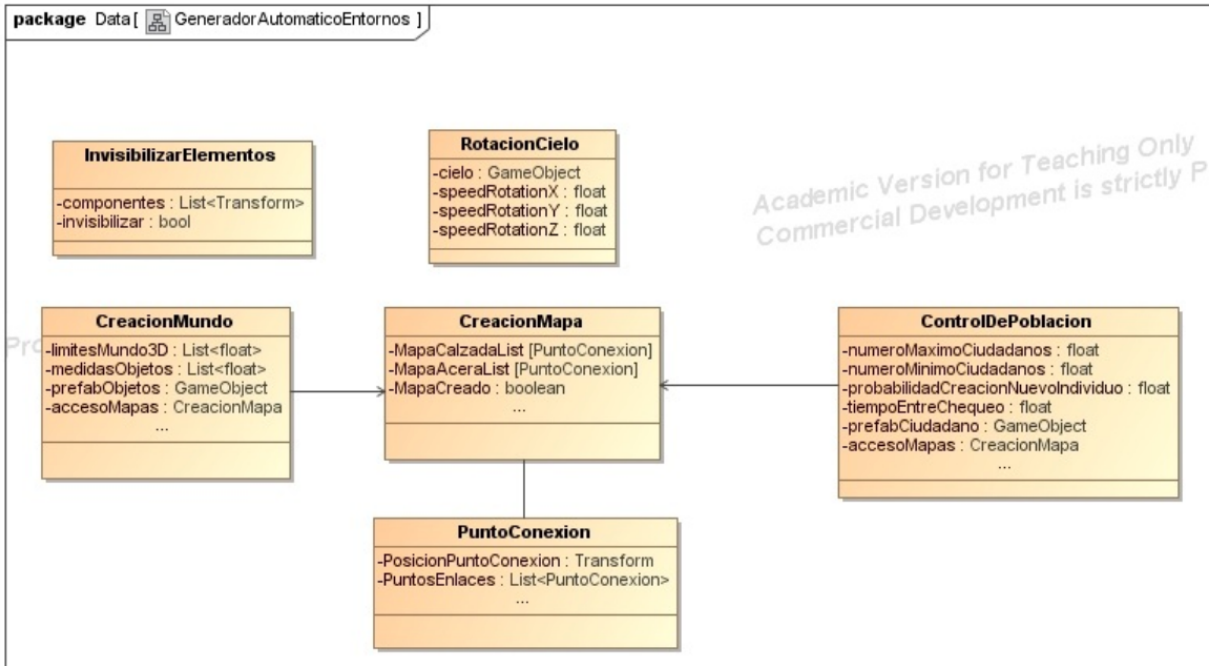
Hasta aquí serían los resultados que obtendríamos de nuestro sistema para cada uno de los casos posibles que pudiesen darse concluyendo así la recogida de todos los algoritmos, implementados y usados por librerías, ventajas y desventajas de las decisiones tomadas para la elección de un algoritmo u otro, así como de la estructura del proyecto y de problemas ocasionados durante la realización del mismo.

Finalmente, concluiremos este capítulo con los diseños UML que reflejan los scripts de mayor importancia desarrollados para este proyecto los cuales muchos de ellos han sido comentados su función a lo largo del desarrollo de esta memoria:

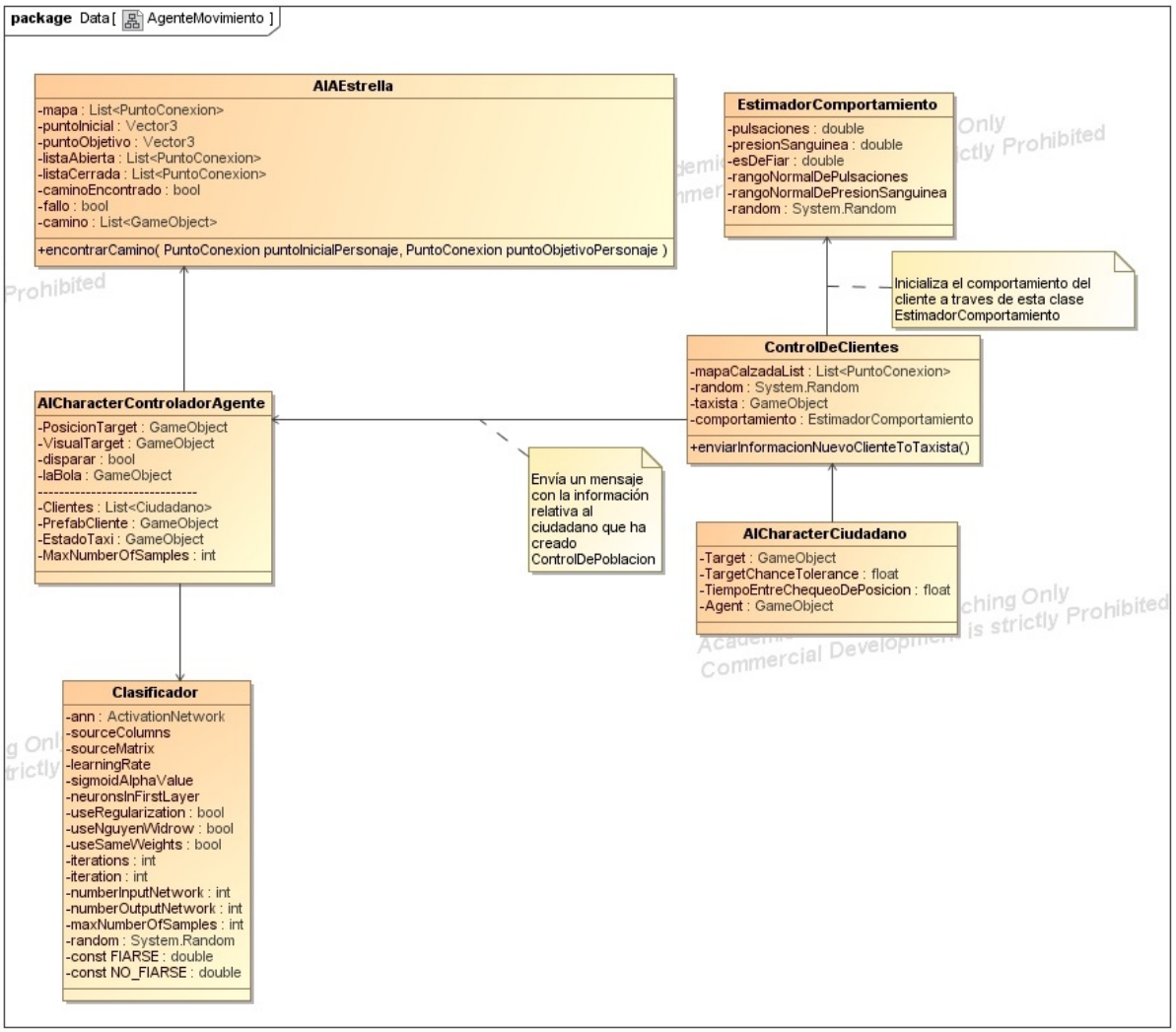
- En la siguiente imagen vemos como el controlador del agente utiliza una instancia de la clase Aproximador.cs para poder realizar la interpolación de fuerzas necesarias para encestar. Cuando esta bola se lance será de la clase ControlBola.cs donde el controlador cogerá el error ya que es esta clase la que se encarga de dicho propósito. Por otro lado, con el árbol de decisión implementado en IA_ControlMeteorologico.cs de cuya clase el controlador hace otra instancia será usada para en función de la clase ControlMeteorologico.cs, la cual se encarga de los factores atmosféricos, saber que decisión debe tomar, es decir, si jugar o no en función del clima actual.



- En el siguiente diagrama UML vemos las clases utilizadas por el generador automático de entornos para rotar el cielo, crear un mapa 3D así como la estructura a iterar en el código para las calzadas y aceras de nuestra ciudad, usando instancias de la clase PuntoConexion.cs, así como el control de la población de la ciudad con el objetivo de controlar cuántos ciudadanos existen en dicha ciudad.



- Finalmente, en este diagrama UML podemos observar los scripts necesarios para el rol de taxista bajo el cual nuestro agente a través de la clase controlador hace uso de una instancia del algoritmo A* desarrollado en la clase AIAestrella.cs. Además, el controlador hace uso de la clase Clasificador.cs donde se encuentra nuestra red neuronal la cual decide si los atributos presión sanguínea, pulsaciones y nivel de delincuencia son o no propios de una persona de fiar. Por otro lado, nos encontramos en el diagrama UML las clases necesarias para controlar y establecer que clase de ciudadano se ha creado. Para ello, se usa la clase EstimadorComportamiento.cs donde se establece el modelo del que se hablo en capítulos anteriores con el fin de establecer si una persona es de fiar. AICharacterCiudadano.cs es la clase encargada de mover a dicho ciudadano haciendo uso de un mapa. Además, nuestro ciudadano contiene a la clase ControlDeClientes.cs usándola para establecer a sí mismo un modelo de un tipo persona de fiar o no.



6. Conclusión y líneas futuras de investigación.

En los capítulos anteriores se ha explicado los aspectos del entorno en los que vive el agente, algunos de estos hostiles, como por ejemplo: cambios climáticos, nivel de delincuencia del barrio y sus ciudadanos. El objetivo era que el agente autónomo se adaptase a su entorno anteriormente descrito intentando cometer el error mínimo tanto en su profesión como taxista como en sus actividades de entretenimiento. En el caso de su profesión aprendió a partir de las malas experiencias obtenidas con ciudadanos que lo han atacado y ciudadanos que no lo han hecho aprendiendo qué ciudadanos son los más confiables. Por otra parte, en lo referente a sus actividades de entretenimiento, en este caso el lanzamiento a canasta, dado que su intención es tener la satisfacción de encestar ha aprendido a hacerlo reduciendo el error del lanzamiento a canasta así como saber bajo que condiciones meteorológicos su actividad de entretenimiento es más efectiva.

En este proyecto se han sentado las bases de lo que podría ser una linea de investigación basada en la creación de simuladores que retraten situaciones reales de la vida real con el objetivo de probar sistemas aplicados a la vida real dentro de estas simulaciones antes de ser usados en entornos reales pudiendo de este modo realizar los algoritmos necesarios y demás aspectos que podrían no necesitar de objetos físicos como robots. Estos aspectos podrían ser el reconocimiento de imágenes, un algoritmo de mapeo de la zona en la que se encuentre el robot para lo cual se podría usar sistema como Unity a modo de simulación o el algoritmo de un quadricóptero el cual controle cual es la fuerza o capacidad de giro que debe darse a los rotores para mantenerse estabilizado. Es sin duda un mundo abierto con infinitas posibilidades el de la inteligencia artificial aplicada a mundos 3D como el que se ha creado en este proyecto donde los límites son los establecidos por tu imaginación.

A través de los diferentes estados de desarrollo de este proyecto se ha aprendido diferentes tipos de algoritmos de inteligencia artificial así como de minería de datos, lo importante que es el uso de uno u otro dependiendo de cuál es la motivación del problema que se quiere resolver. Además, se ha podido observar que la inteligencia artificial ofrecida hasta hoy día carece de muchos factores que no se tienen en cuenta y que, probablemente, sean imposibles tenerlos en cuenta debido a la gran cantidad de factores que intervienen en la creación de un ente con unas ciertas capacidades o habilidades. Estos factores van desde cuales son los datos que dicho ente debiese analizar para su rendimiento más certero así como cuales son los datos más importantes que dicho ente, por sí mismo, debe descartar y cuáles almacenar ya que en muchos casos podríamos toparnos con limitaciones de almacenamiento o tiempo que con lleva conseguir u almacenar un dato de dicho índole.

Además he podido experimentar como el uso de algoritmos relacionados con el área de la inteligencia artificial no implican un éxito inmediato sino que se debe realizar un cierto

estudio de cuales es la mejor manera de sacar partido a dichos algoritmos. Para ello es de vital importancia distinguir qué datos son más importantes que otros o cuales son simplemente inservibles para el propósito de la red neuronal lo cual tiene que ser evaluado por el programador en sí mismo en caso de usar redes neuronales utilizando la implementación del algoritmo de Levenberg-Marquardt el cual no tiene en cuenta outliers por lo que el no tener en cuenta detalles como este o simplemente no saber como funciona el algoritmo usado en la red neuronal o sus limitaciones llevarán a una solución mediocre, eficiente u optimizada o en el peor de los casos todas a la vez.

La inteligencia artificial es sin duda un mundo que mucho más aún después de la realización de este proyecto cobra un mayor interés para mi del que ya de por sí tenía. Las posibilidades que la creación de entornos o mejor dicho del desarrollo de algoritmos que, una vez aplicados a este mundo virtual, pudiesen aplicarse de igual modo en mundos reales me ha abierto un nuevo mundo que explorar y al que, por qué no, dedicarme en un futuro como profesional en el mundo de la creación inteligencia artificial asociada a videojuegos.

Finalmente, destacar que la inteligencia artificial de los ciudadanos entra dentro de uno de los puntos de desarrollo como linea de investigación futura como simulación para la comprensión del comportamiento entre humanos o la interacción hombre-robot y que, por como ha sido implementado, podría ampliarse en un futuro de manera independiente al resto del sistema.

◉ Referencias bibliográficas

1. http://www.turbosquid.com/
2. http://unity3d.com/es
3. http://www.codeproject.com/Articles/16447/Neural-Networks-on-C
4. http://crsouza.blogspot.com.es/2012/01/decision-trees-in-c.html
5. http://travel.amerikanki.com/most-beautiful-places-in-the-world/
6. http://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico
7. http://www.perlmonks.org/index.pl?node_id=638391
8. http://www.intelnics.com/neuraldesigner
9. http://web.archive.org/web/20140528023449/http://www.esp.uem.es/gsi/
10. http://info.fisica.uson.mx/arnulfo.castellanos/archivos_html/quesonredneu.htm
11. http://www.gc.ssr.upm.es/inves/neural/ann2/anntutorial.html
12. http://www.aforgenet.com/
13. http://crsouza.blogspot.com.es/2009/11/neural-network-learning-by-levenberg_18.html
14. http://theory.stanford.edu/~amitp/GameProgramming/
15. http://www.policyalmanac.org/games/aStarTutorial.htm
16. http://www.vidaextra.com/listas/4-motores-graficos-para-perder-el-miedo-y-lanzarse-al-desarrollo-de-videojuegos
17. http://www.genbetadev.com/programacion-de-videojuegos/herramientas-para-desarrollar-videojuegos
18. http://msdn.microsoft.com/es-es/library/bb200104(v=xnagamestudio.40).aspx
19. http://www.aco-metaheuristic.org/
20. http://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm
21. http://www.rulequest.com/Personal/
22. http://www2.cs.uregina.ca/~hamilton/courses/831/notes/ml/dtrees/4_dtrees1.html
23. http://journal.sepln.org/sepln/ojs/ojs/index.php/pln/article/view/800
24. http://www.ii.uam.es/esp/posgrado/proyectos/oscar_perez.pdf
25. http://informatica.ucam.edu/?p=748
26. http://personales.unican.es/zorrillm/PDFs/caepia07.pdf
27. http://www.ia.urjc.es/cms/sites/default/files/userfiles/file/ia4/2011/IA4_%5BRefuerzo%5D(1).pdf

◉ Anexos técnicos

- Información complementaria que no tenga cabida en el cuerpo del TFG, tales como listados, descripciones detalladas, manuales de usuario y programador, etc.