Inteligencia artificial para videojuegos

PID_00270634

Jordi Duch i Gavaldà
Heliodoro Tejedor Navarro
Samir Kanaan Izquierdo
Gerard Escudero Bakx

Tiempo mínimo de dedicación recomendado: 9 horas

Universitat Oberta de Catalunya

Jordi Duch i Gavaldà

Heliodoro Tejedor Navarro

Samir Kanaan Izquierdo

Gerard Escudero Bakx

La revisión de este recurso de aprendizaje UOC ha sido coordinada por el profesor: Carles Ventura Royo (2019)

Segunda edición: septiembre 2019

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

1. Inteligencia artificial en los videojuegos

¿Cuál es la característica más importante para que un videojuego tenga éxito? Seguramente que entretenga a los jugadores, de forma que mantenga una amplia base de usuarios y así el juego siga atrayendo a nuevos jugadores. ¿Cómo se consigue que un videojuego entretenga a los jugadores? Hay muchos aspectos importantes (idea original, diseño del juego, gráficos y sonido), pero un aspecto fundamental es que ofrezca al usuario una experiencia excitante, que no resulte aburrido por ser demasiado fácil ni frustrante por ser muy difícil.

Pero las cosas no siempre son tan sencillas como ajustar el nivel de dificultad. En videojuegos sencillos (tipo Tetris, Candy Crush y similares) la dificultad se puede ajustar simplemente acelerando el juego o restringiendo el tiempo disponible. Pero eso no es aplicable a otros juegos más complejos, en los que el usuario espera retos que pongan a prueba su inteligencia y habilidad para conseguir vencer en el juego.

Y aquí entra el objetivo principal de la inteligencia artificial (IA) en los videojuegos: dotar a las diferentes entidades con las que interactúa el usuario (soldados enemigos, coches en una carrera, otro equipo de fútbol…) de un comportamiento que parezca inteligente, de modo que a su vez el jugador necesite esforzarse para superar al ordenador. Pero que no sea simplemente subiendo los puntos de vida de los enemigos, o acelerando los otros coches, sino que el jugador tenga la sensación, si le ganan, de que le han ganado limpiamente porque son mejores; así, cuando sea el jugador el que gane en el juego, su satisfacción será mayor y, por tanto, su afición por el juego crecerá.

Agentes

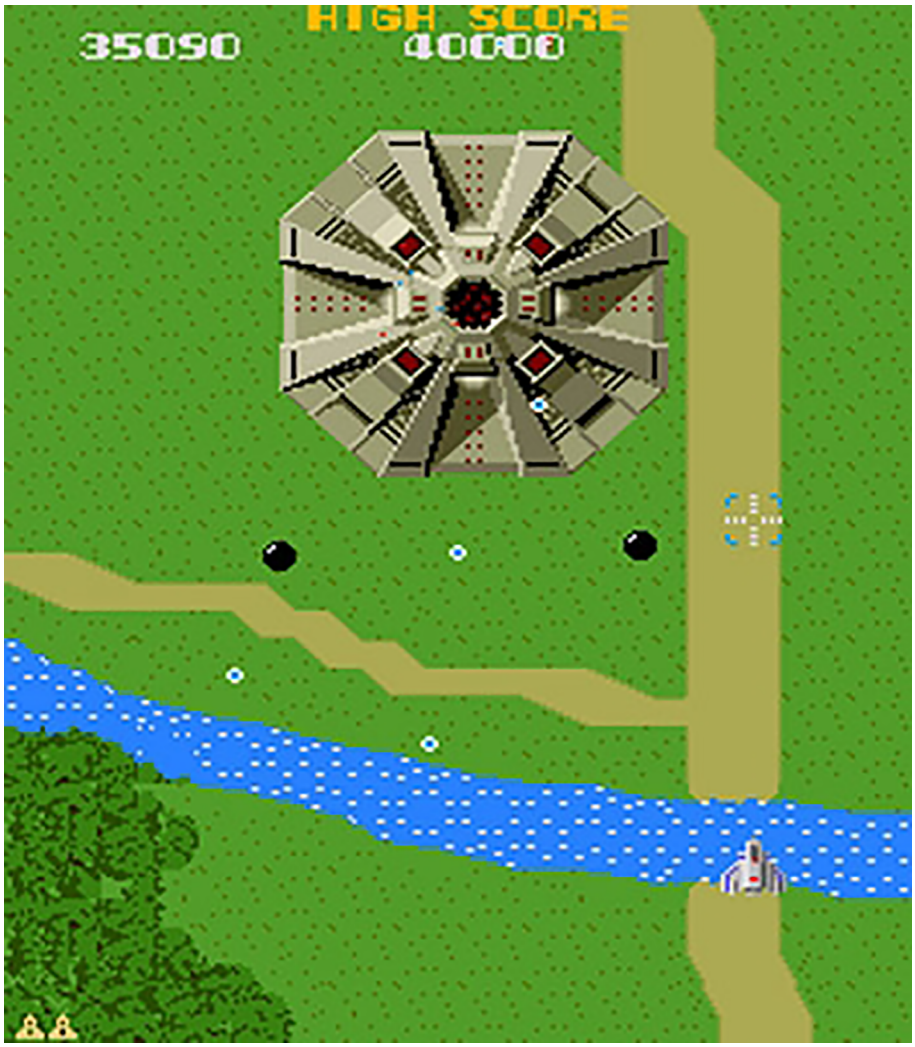
Denominaremos **agentes** a las entidades del juego que deban mostrar algún tipo de autonomía o comportamiento inteligente.

1.1. Historia de la inteligencia artificial en los videojuegos

Las necesidades de IA de los primeros videojuegos (desarrollados a lo largo de los años setenta y ochenta) eran bastante mínimas. Los juegos de plataformas clásicos simplemente tenían una serie de pequeños programas o *scripts* (de diferente complejidad dependiendo de lo «listo» que hiciéramos al juego) que definían comportamientos simples y repetitivos de los enemigos.

Muchas veces, para conseguir ganar a un oponente (por ejemplo, lo que se conocía por «el monstruo final de fase»), bastaba con fijarse durante un tiempo en el patrón de acciones que realizaba de manera repetitiva y después debíamos ajustar nuestras acciones para contrarrestar el patrón.

Xevious



Fuente: © Namco.

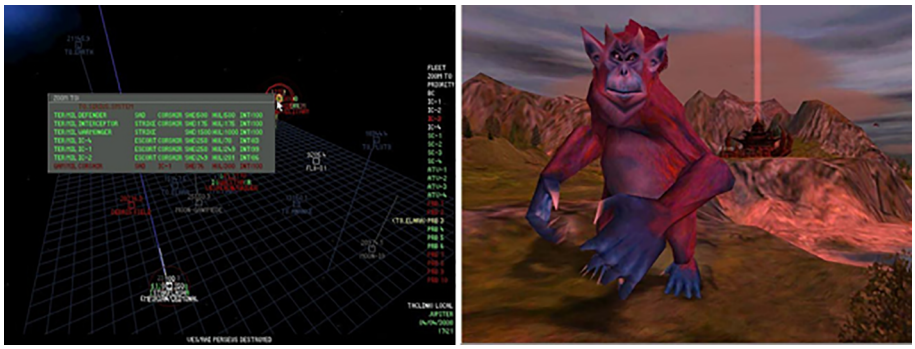
A comienzos de los años noventa aparecieron nuevos géneros con muchas más necesidades de IA que los juegos de plataformas que habían existido hasta la fecha. Un ejemplo claro de esto es el juego Dune II, el primer juego de estrategia en tiempo real, el cual requería una IA capaz de planificar una estrategia en tiempo real y adaptarse a las diferentes situaciones que le iba planteando el jugador. Para este tipo de juegos se empezaron a utilizar máquinas de estados finitos (que explicaremos en el apartado 3), que se combinaban con técnicas de búsqueda de caminos y técnicas de planificación de estrategias. La introducción de esas técnicas incrementó la popularidad de este tipo de juegos considerablemente a lo largo de los noventa, con sagas como Starcraft, Warcraft, Command & Conquer o Age of Empires. El uso de estas técnicas también se extendió a otros géneros en los que la IA desempeña un papel clave.

Dune II: Battle for Arrakis y Age of Empires II: The Age of Kings

Fuente: © Westwood Studios y Ensemble Studios respectivamente.

Juegos posteriores empezaron a añadir técnicas más complejas de IA que se utilizan principalmente en otros campos, como por ejemplo el uso de redes neuronales en el juego Battlecruiser 3000AD o el uso de comportamientos emergentes (o evolutivos) en juegos como Creatures o Black & White. Aunque la utilización de estas técnicas no es muy común, estas son muy útiles para modelar ciertos comportamientos que no podemos crear usando técnicas determinísticas como *scripts* o patrones de reglas.

Battlecruiser 3000AD y Black & White



Fuente: © Take Two Software y Lionehad Studios respectivamente.

Más recientemente se han incluido nuevas técnicas en la IA de videojuegos. Uno de los elementos que se ha potenciado es el comportamiento en grupo, es decir, las interacciones y la coordinación entre agentes inteligentes. Un ejemplo de este tipo de comportamiento lo podemos encontrar en juegos como Far Cry, Halo o F.E.A.R, donde los enemigos se coordinan para rodear al jugador o prepararle emboscadas, llegando a utilizar tácticas militares que dan mucho más realismo al comportamiento de los enemigos.

Otro de los últimos avances en IA ha consistido en intentar aumentar la reactividad de los agentes inteligentes con el entorno que los rodea, siguiendo la línea de tomar las decisiones según un patrón de acciones predeterminadas en *scripts*. La principal gracia de este tipo de sistemas es que no se repite nunca dos veces la misma secuencia de acciones. Uno de los ejemplos más destacados de esta tecnología se encuentra en las demostraciones del motor Euphoria

del juego Star Wars: The Force Unleashed, donde los enemigos son capaces de asirse a una madera para no caer o son capaces de salvar a un compañero que se encuentre cerca.

Far Cry y Star Wars: The Force Unleashed

© Ubisoft y LucasArts respectivamente.

La IA sigue mejorando día a día, principalmente porque su aplicación es multidisciplinar, con lo que las teorías se pueden copiar de un campo a otro y, además, existe mucha más gente implicada mejorando las teorías y los métodos existentes. Posiblemente no estamos muy lejos del día en el que no podremos diferenciar si estamos jugando contra otros jugadores o contra un personaje controlado por un ordenador.

El test de Turing

El test de Turing es un test que nos permite identificar la existencia de inteligencia en una máquina. Si no somos capaces de diferenciar si nos comunicamos con una persona o con una máquina al usar este test, entonces se considera que la máquina es «inteligente».

1.2. Aplicaciones de la inteligencia artificial en los videojuegos

Si bien es posiblemente la aplicación más visible, la IA en los videojuegos no solo se utiliza para manejar los agentes contra los que van a jugar los usuarios, sino que tiene otras muchas aplicaciones igualmente importantes:

- La primera, como ya hemos dicho, es controlar a los agentes con los que compite el jugador.

- En muchos juegos también controla agentes que acompañan o ayudan al jugador, como en los compañeros que pueden elegirse en Diablo o los soldados aliados de videojuegos como Call of Duty.

- Generalmente los agentes del juego esperan a que llegue el jugador para activar su plan de acción, son reactivos; sin embargo, en algunos juegos, como en Alien: Isolation, es el agente el que sale a la caza del jugador. Eso implica un diseño diferente de los objetivos de la IA.

- Un aspecto básico de muchos juegos es conseguir que las unidades se muevan por el nivel de forma creíble y eficiente; es lo que se conoce como búsqueda de caminos.

- En las etapas finales de desarrollo del juego, es necesario ajustar sus parámetros (vida de las unidades, potencia de los coches, etc.) para conseguir un grado de dificultad apropiado; este problema puede ser muy complica-

do por la estrecha interrelación entre parámetros, por lo que se necesita aplicar técnicas de optimización.

- Algunos juegos, como la serie Forza (carreras de coches), aprenden de los jugadores para crear un agente que conduzca como ellos, poniéndolo a disposición de otros jugadores para conseguir adversarios más realistas.

- También veremos que últimamente se utilizan técnicas innovadoras de IA para que sea el ordenador el que juegue a juegos (es decir, que haga de jugador). De momento esto son experimentos y prueba de técnicas, pero seguramente por este camino se revolucionará la idea que tenemos de IA para videojuegos.

- Finalmente, las técnicas que veremos en este módulo son muy cercanas o incluso las mismas que se aplican en otros problemas, como en robótica, visión artificial, conducción automática, organización de empresas y producción, economía, etc.

Agente que acompaña y ayuda al jugador en Diablo III

© Blizzard Entertainment.

1.3. Consideraciones sobre la IA en videojuegos

La IA que se utiliza en videojuegos tiene varias características propias que no suelen darse en otros sistemas que utilizan IA. En este apartado describimos algunas de ellas.

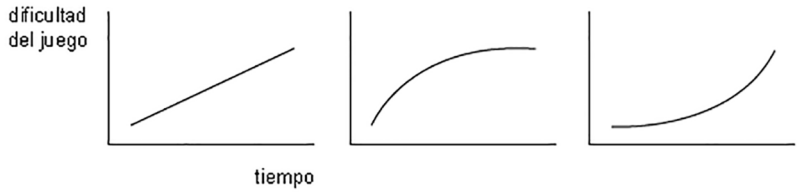
1.3.1. Ajuste de dificultad

En la mayoría de las situaciones en las que se aplica la IA se desea conseguir los mejores resultados posibles (clasificar radiografías, predecir huracanes, conducir un coche, decidir qué acciones comprar, etc.). Al diseñar la IA para un videojuego es fácil caer en la tentación de intentar lograr una IA perfecta. Eso es correcto, pero la equivocación es pensar que una IA perfecta para un videojuego debe jugar perfectamente, ya que si lo hace, el juego resultará imposible

para los jugadores humanos. Como se ha dicho antes, el diseño de agentes contra los que jugarán los usuarios debe lograr un grado de dificultad justo, ni muy bajo ni muy alto. Y así hay que diseñar la IA, por chocante que parezca en un principio, para que alguna vez se equivoque.

Esto tiene mucha relación con lo que se conoce como **curva de dificultad** del juego, que representa gráficamente cómo aumenta la dificultad del juego a medida que se avanza en él. Habitualmente se trata de curvas lineales, convexas (logarítmicas) o cóncavas (exponenciales). Es importante decidir el tipo de curva deseada y ajustar la IA, junto con los demás parámetros del juego, para que la experiencia del jugador sea la esperada.

Ejemplos de curvas de dificultad: lineal, logarítmica y exponencial



1.3.2. Restringir el conocimiento del juego

En la mayoría de las aplicaciones de IA fuera del ámbito de los videojuegos, se intenta que el sistema de IA tenga acceso a toda la información que le pueda ayudar a ejecutar sus funciones y conseguir sus objetivos. La IA de un videojuego, sin embargo, no debe tener acceso a toda la información (por ejemplo, no debe ver todo el mapa en un juego de estrategia, o no debe oír al jugador hasta que esté a una cierta distancia), sino que debe usar la misma información que está disponible para los jugadores; de lo contrario, parecerá que la IA hace «trampas» (se denomina *cheating AI*, en inglés) y los jugadores se indignarán con el juego (con razón).

> En el juego Civilization se eliminaron las alianzas entre agentes de IA porque funcionaban tan bien que los jugadores pensaban que el ordenador hacía trampas. La IA no solo no debe ser tramposa, sobre todo no debe parecerlo.

Para conseguir esta apariencia de legalidad en la IA, se debe evitar que los agentes accedan a información privilegiada del motor de juego; en su lugar, deben usar información «sensorial», lo que un humano podría ver u oír en su situación.

También puede considerarse trampa que la IA se emplee a fondo y, por ejemplo, realice un número desorbitado de acciones por minuto. La IA debe parecer humana, no sobrehumana.

Juegos de estrategia

En juegos de estrategia como Starcraft, de los que hay competiciones internacionales de *e-sports* (*electronic sports*) con gran seguimiento e importancia, se toman las acciones por minuto que puede ejecutar un jugador a modo de indicador de su velocidad con el juego. El récord mundial lo tiene Park Sung-Joon, con 818 acciones por minuto. Nosotros tampoco nos lo explicamos.

1.3.3. Tiempo real, coste computacional y completitud

Un aspecto fundamental de la IA en un videojuego es que, en la mayoría de los juegos (salvo en los juegos por turnos), debe reaccionar en tiempo real a la situación del juego, y tiene un tiempo limitado para dar su respuesta, así como recursos computacionales limitados: mientras la IA trabaja, el resto del juego debe seguir moviéndose, así que no puede ocupar todo el procesador. Por tanto, hay que tener en cuenta algunos factores:

Los procesadores gráficos

Con el extraordinario desarrollo de los procesadores gráficos, los procesadores centrales quedan relativamente descargados al ejecutar videojuegos, en especial si tienen varios núcleos. Así que quedan más recursos para la IA. El problema es que los métodos más potentes de IA, como *deep learning*, que se verá en el apartado 5, ¡también quieren ejecutar sus cálculos en el procesador gráfico!

- Quizá no puedan usarse los mejores métodos disponibles, sino los más sencillos y rápidos.

- Hay que diseñar bien dónde se ejecutará la IA. Habitualmente se le dedicará un hilo de ejecución (*thread*) aparte para que no bloquee el resto del juego.

- En cualquier caso, hay que tener en cuenta el hardware existente: múltiples núcleos, procesador gráfico, memoria, etc. En algunos casos habrá que ajustar el consumo de la IA en función de los recursos disponibles en cada máquina.

- En los videojuegos el bucle principal del juego es el que dicta los tiempos, que a su vez suelen depender del dibujo de cada fotograma (*frame*), que es variable porque depende del número de triángulos en pantalla y de otros efectos. Por tanto, no hay un tiempo fijo para ejecutar una iteración, con lo que la IA tiene estas opciones:
  - Utilizar un diseño flexible que la haga capaz de dar un resultado aproximado si no dispone de tiempo suficiente para terminar.
  - Utilizar un método con una duración máxima inferior al tiempo mínimo entre iteraciones.
  - Ejecutarse en paralelo al bucle principal de juego y ser capaz de dar resultados bajo demanda (ejecución asíncrona).

1.4. Arquitectura de la IA

Desde un punto de vista de su organización, ¿cómo es la IA de un videojuego? Básicamente hay dos arquitecturas, y a menudo el propio tipo de juego invita a utilizar una o la otra.

1.4.1. Arquitectura centralizada

Se programa una IA a imagen de un jugador humano, que puede controlar diferentes elementos (fichas, unidades, etc.) y coordina su uso desde un plan general. En esta arquitectura se sigue una estrategia de arriba abajo (*top-down*), en la que la IA decide un plan de acción general y da todas las órdenes pertinentes para que se lleve a cabo ese plan.

Si la IA controla a varios jugadores, se crearán varias instancias de la arquitectura centralizada.

Es la solución preferida para juegos de tablero o de estrategia.

1.4.2. Arquitectura distribuida: agentes, sistemas multiagente, IA emergente

En esta arquitectura, cada unidad del juego (cada soldado, piloto, deportista, robot, etc.) debe mostrar un comportamiento autónomo e inteligente, así que para cada agente del juego habrá una instancia de la IA, es decir, un método con sus propios datos; se pueden usar distintos métodos para distintos tipos de agentes. Los agentes perciben su entorno, actúan en él y se pueden comunicar con otros agentes. Por tanto, se acaba obteniendo un sistema **multiagente**.

La estrategia resultante es de abajo arriba (*bottom-up*), ya que empieza en cada agente individual y sus acciones se suman en el conjunto del juego. Si la interacción entre los distintos agentes da lugar a un comportamiento coordinado que supera a la suma de las partes, se habla de IA **emergente**.

Es la solución preferida en juegos en los que se distinguen las unidades individuales y los jugadores poseen un comportamiento autónomo: juegos en primera persona, de deportes, etc.

Requiere establecer mecanismos de comunicación entre agentes.

Los sistemas emergentes

Los **sistemas emergentes** son aquellos sistemas compuestos de entidades relativamente sencillas pero que al interactuar dan lugar a comportamientos complejos. Ejemplos: las neuronas, las hormigas.

2. Técnicas de movimiento

Probablemente la primera característica de los agentes de un juego que dé la sensación de vida propia, y por tanto de inteligencia, es el movimiento. En la mayoría de los videojuegos aparecen distintos elementos (enemigos o competidores, objetos que hay que atrapar, etc.) que deben moverse de una forma u otra para dar sentido al juego. También hay que tener en cuenta que la IA del juego puede verse en la obligación de mover a las unidades del jugador, como por ejemplo en los juegos de estrategia en tiempo real, en los que el jugador selecciona algunas unidades y marca en el mapa a dónde quiere que vayan.

En este apartado veremos técnicas que permiten mover los elementos del juego de manera adecuada, teniendo en cuenta los siguientes criterios de diseño del movimiento:

- **Predictibilidad**: según el tipo del juego y su diseño, el movimiento puede ser predecible, como en un juego de plataformas donde los enemigos siempre hacen el mismo recorrido (ya que el objetivo del juego es encontrar una secuencia de movimientos que permita superar el nivel), o impredecible como en un juego de disparos en primera persona tipo «arena», en el que utilizar agentes que se movieran de forma predecible arruinaría el juego completamente.

- **Realismo**: el tipo de movimiento debe adecuarse al tipo de entidad; no se mueve igual un camión que un ratón (en el sentido de que no realizan el mismo tipo de recorridos, no ya en sentido de que la animación sea diferente). Los agentes del juego pueden andar, correr, volar, nadar, etc., y en cada caso el trazado de caminos será diferente.

- **Comportamiento**: ¿qué tipo de comportamiento se quiere representar? El movimiento de los agentes debe reflejar su intención; como tipos de movimiento, según este criterio, podemos encontrar movimiento casual (paseo), búsqueda, persecución, huida, acecho, movimiento grupal, etc.

- **Evitación**: cada nivel de un juego tiene una serie de zonas por las que el movimiento está permitido y otras por las que no; esto puede ser estático (paredes) o dinámico (puertas que se abren, barriles que se pueden romper); además, cada tipo de terreno puede tener un coste implícito asociado que recomiende evitarlas (pendientes, barro, etc.), o bien puede convenir evitarlas por estar controladas por jugadores/IA enemigos.

Para dar respuesta a los criterios anteriores, veremos diferentes tipos de técnicas de movimiento:

- **Movimientos cíclicos y basados en patrones:** al diseñar el nivel del juego se diseñan una serie de recorridos para los agentes, que seguirán de forma determinista y repetitiva.

- **Búsqueda de caminos:** cuando los agentes necesitan encontrar un camino de forma dinámica, la IA debe realizar una búsqueda en el mapa.

- **Movimientos complejos:** a veces el movimiento que se desea no es simplemente «ir de A a B», sino alcanzar objetivos dinámicos, como interceptar a otra unidad en movimiento (por ejemplo, una nave dispara un rayo láser a otra), o como coordinarse con otras unidades que están en movimiento (grupos de enemigos que atacan juntos, por ejemplo). Esas situaciones conllevan unos problemas específicos para los que se verán algunas soluciones.

Antes de comenzar con las técnicas de movimiento hay que definir algunos conceptos generales que se usarán en este apartado:

- **Zonas de movilidad**: al diseñar un nivel de juego, es necesario definir por dónde se pueden mover las unidades y por dónde no; es decir, la IA debe saber que hay una pared, o un río, o cualquier otro obstáculo. La forma habitual de representar esta información es añadiendo al mapa de cada nivel una especie de capa invisible que indica por dónde se pueden mover las unidades. Dependiendo del diseño del juego y del entorno de desarrollo, las zonas de movilidad se deben crear manualmente o es el propio entorno el que las calcula. En cualquier caso, solo se deben tener en cuenta los obstáculos permanentes, ya que recalcular las zonas de movilidad de un nivel puede ser muy costoso.

Unity

Afortunadamente, Unity facilita la tarea de movimiento de las unidades mediante la **Navigation Mesh** (o simplemente **NavMesh**), en la que Unity calcula por qué zonas se pueden mover los agentes en el nivel, y así esas zonas de movilidad ya calculadas se pueden usar en el juego para controlar el movimiento de los agentes. Más adelante se habla de NavMesh.

- **Obstáculos**: en algunos juegos puede haber obstáculos temporales (barriles, cajas o puertas, unidades moviéndose, bloqueos hasta cumplirse alguna condición) que por razones de eficiencia no conviene incorporar en las zonas de movilidad, sino tratar como obstáculos dinámicamente en el momento de calcular los movimientos de las unidades.

- **Características de los agentes**: para poder calcular el movimiento de un agente por un nivel hay que conocer su tamaño. Generalmente, los agentes bípedos se representan mediante un cilindro por lo que respecta al movimiento, por lo que sus características definitorias son el radio y la altura. También se suele tener en cuenta lo máximo que pueden saltar y la máxima pendiente que pueden subir. Otros tipos de agente (vehículos, animales) suelen representarse mediante cajas.

Finalmente, situando el control del movimiento de los agentes en el contexto general de la IA del juego, en general el movimiento es un instrumento básico para que los agentes puedan llevar a cabo sus objetivos generales en el juego: atacar al jugador, cooperar con él, o competir contra él, entre otros. De modo que los agentes tomarán una serie de decisiones de alto nivel, lo que llamaríamos la «estrategia» del agente, y el encontrar el camino para alcanzar sus objetivos es uno de los primeros elementos que se deben proporcionar al agente para que la IA en el juego funcione.

2.1. Movimientos cíclicos y basados en patrones

2.1.1. Movimientos predefinidos

La manera más sencilla de dotar de movimiento a los agentes del juego es definir una trayectoria y hacer que la sigan. El caso más básico, dentro de esta categoría, consiste en «dibujar» la trayectoria del agente al diseñar el nivel. Aunque esta técnica tenga muchas limitaciones, tiene su utilidad en juegos de tipo plataforma o arcade, en los que el jugador espera que los agentes sigan siempre el mismo movimiento.

Generalmente, este tipo de movimiento se define sobre rutas cíclicas. También se puede hacer que cuando el agente llegue a un extremo del recorrido dé la vuelta y lo recorra en sentido contrario. De otra forma, el agente se detendría al terminar su recorrido y la dinámica del juego se vería alterada.

2.1.2. Trayectorias lineales

Una alternativa sencilla que no requiere dibujar el recorrido completo consiste en indicar al agente cuáles son los puntos de inicio y fin de su recorrido y hacerle seguir la línea recta que lleva de un punto a otro. La ejecución de esa trayectoria consiste en calcular, para cada instante de tiempo en el que se quiera actualizar la posición del agente (habitualmente para cada fotograma del juego), la posición del agente sobre la recta. Suponiendo que la velocidad es constante y que t es una variable entre 0 y 1 que indica la proporción de camino recorrido (0 al inicio, 1 al acabar), la expresión que calcula la posición del agente en el momento t del recorrido es:

$$P(t) = (1\text{-}t)\ P_{inicio} + tP_{fin}$$

Es decir, se calcula una interpolación lineal entre los dos extremos del recorrido.

La velocidad de un cuerpo

Estrictamente hablando, la velocidad de un cuerpo es un vector de tres componentes que indica en qué dirección se mueve y con qué «velocidad» lo hace, que es el módulo del vector velocidad, y que para distinguirlo se denomina celeridad.

Interpolación

Se llama interpolación a la estimación de una función que pase sobre un conjunto de puntos dados. Por ejemplo, la recta que trazamos entre dos puntos es una interpolación lineal. Sin embargo, si hay más de dos puntos no siempre se puede trazar una recta que pase por todos ellos; por ese motivo se recurre a funciones de otro tipo, en general con

algún tipo de curvatura que permita ajustarse mejor a todos los puntos aunque no estén en línea recta.

El agente estará orientado en la misma dirección del movimiento. Generalmente, la dirección de los agentes se expresa como un vector unitario (de módulo 1); de modo que si la velocidad del agente es V, estará orientado en la dirección:

$$D(t) = \frac{V(t)}{|V(t)|}$$

2.1.3. Trayectorias curvas

Cuando se desea definir una trayectoria que pase por más de dos puntos se pueden ir encadenando trayectorias lineales, pero el resultado será bastante forzado porque hará que el agente se vaya moviendo en líneas rectas y gire en ángulos bruscamente, lo que le restará realismo al juego (a no ser que el agente sea un robot o algo por el estilo). Por esa razón, cuando se define una trayectoria con más de dos puntos, se utiliza algún tipo de curva que recorra los puntos sin generar ángulos.

En informática gráfica y videojuegos, las interpolaciones mediante curvas suelen hacerse con las curvas llamadas ***spline***, que se definen a partir de un conjunto de **puntos de control**.

Hay diferentes tipos de *splines*, como las curvas Bézier o las curvas Catmull-Rom. En algunos tipos de curvas puede ocurrir que la curva no pase exactamente por los puntos de control, sino cerca de ellos. Eso puede ser un inconveniente en un juego, y por esa razón usaremos las curvas Catmull-Rom, que siempre pasan por los puntos de control.

Las curvas Catmull-Rom

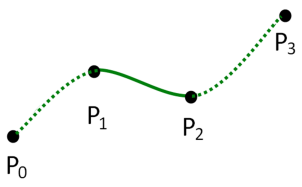
Las curvas Catmull-Rom deben su nombre a sus creadores, Edwin Catmull y Raphael Rom. Edwin Catmull, entre otras cosas, es el creador de Pixar Animation Studios; solo esto ya nos debería convencer del interés de esas curvas.

Las curvas Catmull-Rom se definen mediante cuatro puntos de control $\{P_0,P_1,P_2,P_3\}$, aunque la curva en sí solo se dibuja entre los dos puntos centrales $\{P_1,P_2\}$. Si se desea dibujar una curva Catmull-Rom sobre más puntos, hay que calcular lo que se conoce como una **cadena** de curvas, desplazando los puntos de control dentro de la secuencia de puntos, es decir, calculando la curva para cada subconjunto de puntos $\{P_{n-1},P_n,P_{n+1},P_{n+2}\}$.

Spline

Originalmente una *spline* era una tira flexible de madera o metal que se usaba como guía para dibujar curvas.

Un aspecto importante del movimiento a lo largo de una curva definida por varios puntos de control es que la velocidad del agente al recorrer la curva debe ser constante (a no ser que el juego requiera otra cosa); como los puntos de control no tienen por qué ser equidistantes unos de otros, no se puede calcular la velocidad (la celeridad) independientemente en cada tramo, ya que eso provocaría que el agente se moviera más rápido en los tramos más largos.



Curva Catmull-Rom con cuatro puntos de control

Fuente: Hadunsford - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=28956755.

La solución consiste en calcular la longitud total del camino y calcular un factor de compensación para cada tramo.

Tenemos una *spline* con 5 puntos de control (es decir, 4 tramos) de longitudes 2, 4, 3 y 1 metros. La suma total de longitudes es, por tanto, 10 metros. Si toda la *spline* se debe recorrer en, digamos, 20 segundos, la celeridad debería ser en todo momento de 10/20 = 0,5 m/s.

El tiempo invertido en recorrer cada tramo deberá ser diferente, proporcional a la longitud total y a la duración deseada para el recorrido. Así, el tramo de 2 metros se debe recorrer en 2 m / (0,5 m/s) = 4 s, el de 4 metros en 4 m / (0,5 m/s) = 8 s, etc. Así se mantiene una celeridad constante.

El siguiente código en C# para Unity calcula la curva Catmull-Rom a partir de sus cuatro puntos de control.

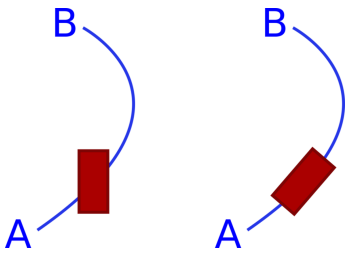
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Catmull : MonoBehaviour {

//Use GameObject in 3d space as your points or define array with desired points
   public GameObject[] points;

   //Store points on the Catmull curve so we can visualize them
   List<Vector2> newPoints = new List<Vector2>();

   //How many points you want on the curve
   float amountOfPoints = 10.0f;

   //set from 0-1
   public float alpha = 0.5f;

   ////////////////////////////

   void Update()
   {
      CatmullRom();
   }

   void CatmullRom()
   {
      newPoints.Clear();

      Vector2 p0 = new Vector2(points[0].transform.position.x, points[0].transform.position.y);
      Vector2 p1 = new Vector2(points[1].transform.position.x, points[1].transform.position.y);
      Vector2 p2 = new Vector2(points[2].transform.position.x, points[2].transform.position.y);
      Vector2 p3 = new Vector2(points[3].transform.position.x, points[3].transform.position.y);

      float t0 = 0.0f;
      float t1 = GetT(t0, p0, p1);
      float t2 = GetT(t1, p1, p2);
```

```
        float t3 = GetT(t2, p2, p3);

        for(float t=t1; t<t2; t+=((t2-t1)/amountOfPoints))
        {
            Vector2 A1 = (t1-t)/(t1-t0)*p0 + (t-t0)/(t1-t0)*p1;
            Vector2 A2 = (t2-t)/(t2-t1)*p1 + (t-t1)/(t2-t1)*p2;
            Vector2 A3 = (t3-t)/(t3-t2)*p2 + (t-t2)/(t3-t2)*p3;

            Vector2 B1 = (t2-t)/(t2-t0)*A1 + (t-t0)/(t2-t0)*A2;
            Vector2 B2 = (t3-t)/(t3-t1)*A2 + (t-t1)/(t3-t1)*A3;

            Vector2 C = (t2-t)/(t2-t1)*B1 + (t-t1)/(t2-t1)*B2;

            newPoints.Add(C);
        }
    }

    float GetT(float t, Vector2 p0, Vector2 p1)
    {
        float a = Mathf.Pow((p1.x-p0.x), 2.0f) + Mathf.Pow((p1.y-p0.y), 2.0f);
        float b = Mathf.Pow(a, 0.5f);
        float c = Mathf.Pow(b, alpha);

        return (c + t);
    }

    //Visualize the points
    void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        foreach(Vector2 temp in newPoints)
        {
            Vector3 pos = new Vector3(temp.x, temp.y, 0);
            Gizmos.DrawSphere(pos, 0.3f);
        }
    }
}
```

Referencia

Fuente: CC Wikipedia https://en.wikipedia.org/wiki/Centripetal_Catmull%E2%80%93Rom_spline.

Por otra parte, cuando un agente se mueve a lo largo de una curva, generalmente tiene que ir orientándose en el sentido de la marcha para que su movimiento sea creíble. Calcular la orientación de un agente a lo largo de una curva no es tan sencillo como a lo largo de una recta. En este caso, su orientación vendrá dada por el vector tangente a la curva en el punto en el que se encuentra el agente en ese momento.



Trayectoria y orientación: si el coche quiere ir de A a B siguiendo la trayectoria, aunque su destino final sea B no debe estar orientado hacia B sino tangente a la trayectoria que va a seguir.

Para indicar a Unity que un agente debe ajustar su orientación al camino seguido, simplemente hay que activar su propiedad *updateRotation*:

```
using UnityEngine;
using System.Collections;

public class MoveTo : MonoBehaviour {

    void Start () {
       NavMeshAgent agente = GetComponent<NavMeshAgent>();
       agente.transform.updateRotation = true
   }
}
```

2.2. Búsqueda de caminos

En la mayoría de los videojuegos, controlar el movimiento de los agentes no es tan sencillo como predefinir una trayectoria, ya que a menudo se encontrarán con obstáculos, tanto relativamente estáticos (cajas, barriles, puertas cerradas) como dinámicos (otros agentes o unidades controladas por el jugador).

Además, no siempre es posible predefinir una trayectoria al diseñar el nivel, ya que muchas veces el objetivo que debemos alcanzar dependerá del desarrollo del juego, y en muchos casos los niveles son generados dinámicamente o bien los jugadores pueden crear nuevos niveles.

Por todas esas razones, con frecuencia es necesario calcular el recorrido de los agentes de forma dinámica, buscando el camino más adecuado en cada momento para alcanzar el objetivo. En la búsqueda de caminos pueden influir muchos factores:

- La propia estructura del nivel: obstáculos fijos e insalvables (paredes, montañas, ríos, etc.) y zonas de movilidad.

- Los puntos de origen y destino del movimiento; este último puede cambiar durante el propio recorrido del camino, por ejemplo porque se trate de alcanzar a un agente en movimiento.

- Los obstáculos temporales, que pueden aparecer y desaparecer a lo largo del desarrollo del juego, como cajas, puertas que se abren y cierran, barricadas, etc.

- Los otros agentes del juego, o las unidades controladas por los jugadores.

- Las características del terreno: en algunos juegos el coste de pasar por unos tipos de terreno puede variar (por ejemplo, cuesta más caminar sobre arena o barro, o cuesta menos caminar sobre caminos o carreteras). O también puede que se tenga en cuenta la pendiente del terreno para incrementar el coste de subir cuesta arriba.

- La situación del juego: en un juego de estrategia, una unidad no debe cruzar por una zona controlada por el enemigo para ir a otra zona.

- Las características del agente: su tamaño, las pendientes máximas que puede subir, si puede nadar o volar, la penalización por cada tipo de terreno, etc.

- Posibles atajos, como teletransportadores o lugares en los que se puede saltar a otro nivel.

En este apartado estudiaremos cómo generar una representación de los niveles del juego que permita aplicar diferentes algoritmos para encontrar caminos. Además, esos caminos deberán tener en cuenta las características aplicables a cada juego, de entre las listadas anteriormente.

2.2.1. Representación de los niveles como grafos

Como se verá en los apartados siguientes, los métodos de obtención de caminos son realmente métodos de búsqueda de caminos en grafos. Pero «¿qué tiene que ver un grafo con el nivel de mi juego?», os preguntaréis. «Yo tengo ríos, montañas, puentes, habitaciones, escaleras…».

Ejemplo de mapa de juego con zonas transitables y no transitables. Generado con el editor de mapas de Battle for Wesnoth (libre). Las marcas azules que se ven a ambos lados del río en la zona inferior son portales de teletransporte.

Grafo

Un grafo es una estructura de datos que contiene dos tipos de elementos: por una parte están los nodos o vértices (los círculos en la imagen), y por otra, los enlaces o aristas (las líneas). Los grafos permiten representar en el ordenador conjuntos de entidades (vértices) relacionadas entre sí (aristas). Por ejemplo, los usuarios de una red social pueden ser los vértices y sus relaciones de amistad, las aristas. Las aristas de un grafo pueden tener un valor asociado, por ejemplo en un grafo en el que las ciudades de un país son los vértices y las aristas son las autopistas que los unen, con la distancia kilométrica asociada a cada arista.

Para conseguir que los agentes busquen sus caminos en los niveles del juego hay que construir un grafo de movilidad; y para crearlo hay que proceder en dos pasos:

1) Encontrar todas las zonas de movilidad **convexas** que haya en el nivel. Cada zona será un vértice del grafo que representa la movilidad en el nivel.

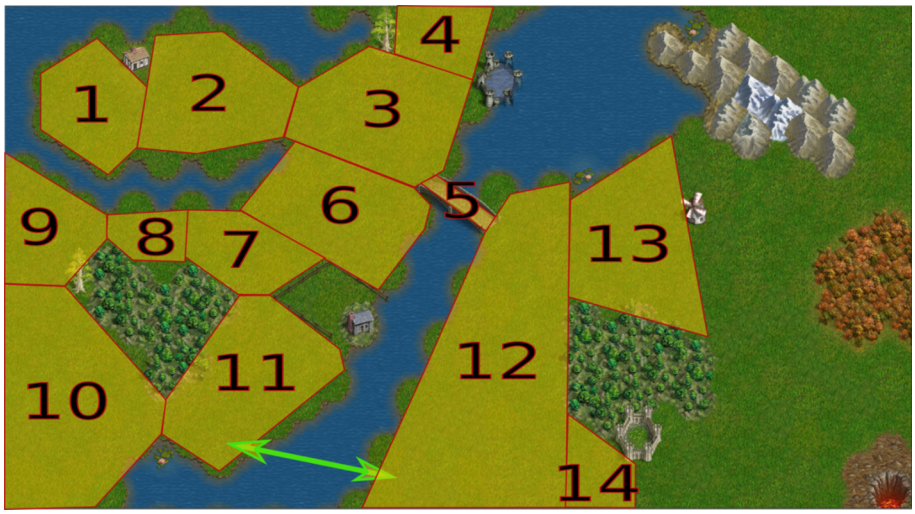
2) Determinar qué zonas están conectadas (es posible moverse directamente de una a otra); cada posible conexión será una arista entre los vértices correspondientes a esas zonas.

¿Qué quiere decir que una zona de movilidad es convexa? Intuitivamente, se dice que una superficie es convexa cuando su límite no tiene zonas que se metan «hacia dentro» creando entradas o bahías; más formalmente, una superficie es convexa cuando es posible unir dos puntos cualesquiera de su interior mediante una línea recta sin salirse de la superficie.

Precisamente esa propiedad es la que nos interesa, ya que garantiza que un agente se pueda mover libremente dentro de la zona; por tanto, el problema queda reducido a conseguir que el agente vaya de zona en zona hasta llegar a la zona que contenga el punto de destino. Y ahí es donde nos conviene tratar las zonas como vértices de un grafo y aplicar algoritmos de búsqueda de caminos en grafos, para navegar de zona en zona.

Este planteamiento permite añadir saltos entre zonas no contiguas, como portales, saltos entre diferentes niveles, etc., simplemente añadiendo otras aristas al grafo de movilidad.

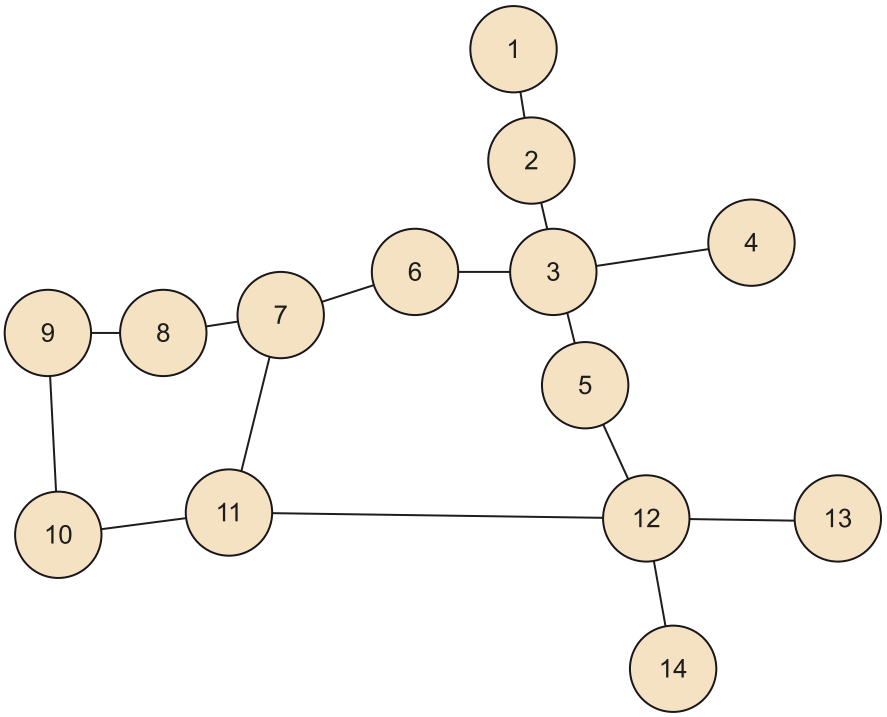
En la siguiente figura pueden verse las zonas de movilidad para el mapa anterior; por no extender excesivamente las explicaciones posteriores no se han generado las zonas para el mapa completo, pero la idea sería similar. Nótese cómo cada zona es convexa, es decir, no hay ningún ángulo obtuso en los vértices de los polígonos.



Mapa anterior con las zonas de movilidad superpuestas. Se resalta con una flecha la conexión entre los portales.

La traducción del mapa de movilidad anterior a un grafo se realiza generando un vértice para cada zona y enlazando zonas adyacentes mediante enlaces entre vértices. Así, el grafo resultante sería:

Grafo de movilidad correspondiente al mapa de ejemplo.



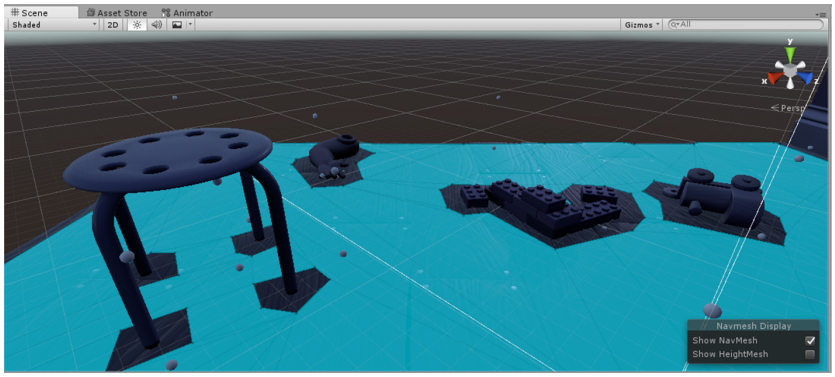
Notad que la situación de los vértices no tiene por qué coincidir con las de las zonas en el mapa; lo único que importa son los enlaces entre zonas.

Grafos de movilidad

Unity calcula grafos de movilidad y los utiliza para calcular el movimiento de los agentes. Es el llamado NavigationMesh, o NavMesh para los amigos, en el que Unity calcula el grafo a partir de la geometría del nivel y a partir de ahí se puede usar para mover los agentes. Hay un apartado del manual que explica cómo generar el NavMesh:

https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html.

NavMesh



NavMesh (azul claro) en el proyecto *Survival Shooter* de Unity.

Solo queda por ver cómo encontrar el mejor camino entre dos vértices de un grafo para poder calcular el camino que debe seguir un agente para ir de A a B. A continuación veremos diferentes algoritmos que resuelven este problema, desde los más sencillos (e ineficientes) hasta los más eficientes, y finalmente veremos algunas variantes para tener en cuenta aspectos adicionales como el coste de caminar sobre diferentes tipos de terreno, la necesidad de evitar obstáculos móviles, o de evitar las zonas que controle el enemigo, por poner algunos ejemplos.

2.2.2. Recorrido en profundidad

Una manera bastante directa de recorrer completamente un grafo es hacerlo por **profundidad**: si partimos de un vértice (1 en la figura del grafo de movilidad), la estrategia consiste en ir al primer vértice vecino (2), y de ese a su primer vecino, y así sucesivamente. Para no entrar en ciclos y repetir vértices, se mantiene una lista en la que se anotan todos los vértices que ya se han visitado.

Nota

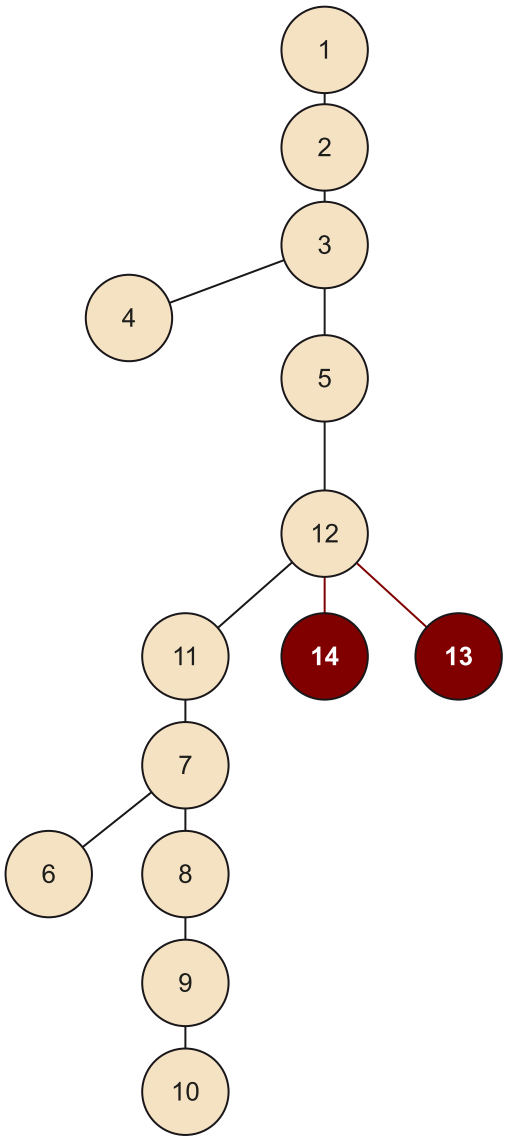
En este ejemplo y los sucesivos ejemplos sobre el grafo de movilidad, supondremos que los vecinos están ordenados numéricamente, es decir, el primer vecino de cada vértice es el vecino con menor valor numérico. Por ejemplo, los vecinos del vértice 3 son los vértices 4, 5 y 6, en este orden.

Aplicándolo a la búsqueda de caminos, para ir de un vértice A a otro B, el recorrido en profundidad simplemente comenzaría en el vértice A y devolvería el camino que encuentre para llegar a B, es decir, la lista de vértices que ha visitado partiendo de A hasta que B aparece en el recorrido.

Al recorrer el grafo evitando pasar dos veces por el mismo vértice, se provoca que solo haya un enlace entre cada par de vértices; es decir, el recorrido en sí tiene una estructura de **árbol**.

Supongamos que en el mapa de zonas de movilidad queremos ir de la zona 1 a la 10. ¿Qué nos propone el algoritmo de recorrido en profundidad? La siguiente figura muestra el resultado del recorrido en profundidad, en el que se accede a cada vecino y de ahí se

busca en sus vecinos hasta que o bien se alcanza el vértice final, o bien se alcanza un vértice que no tiene vecinos no visitados.



Recorrido en profundidad del grafo de movilidad para ir de la zona 1 a la 10. Los nodos en rojo son nodos no visitados por el algoritmo.

Como puede verse, al llegar a la zona 3 primero explora la 4 (que no tiene vecinos disponibles, así que no sigue) y luego la 5; el menor vecino de la 5 es la 12, así que cruza el puente y de ahí pasa a la 11 por el teletransporte. Aunque la zona 11 está junto a la 10, como el orden de los vecinos es numérico tiene que visitar primero la 7, y dar toda la vuelta al bosquecillo hasta llegar a la 10. En ese punto el algoritmo termina y devuelve como ruta la siguiente: [1, 2, 3, 5, 12, 11, 7, 8, 9, 10]. Bastante mejorable, ciertamente.

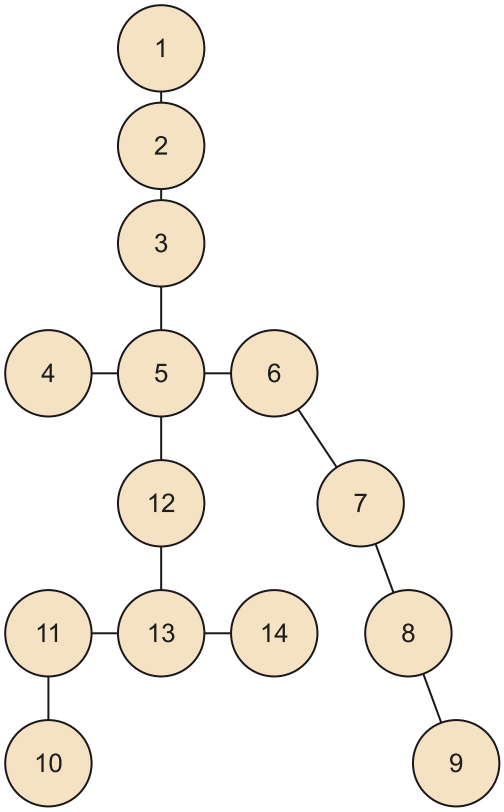
La ventaja de este algoritmo es que es fácilmente programable como una función recursiva que toma un vértice y recorre sus vecinos; solo requiere mantener la lista de los vértices ya visitados.

El problema de este algoritmo es que, si hay varios caminos posibles, solo encuentra uno de ellos, y posiblemente no sea el mejor, lo que hará que el agente camine mucho más de lo necesario en muchas ocasiones, y lo que es peor, su movimiento resultará errático a la vista del jugador.

2.2.3. Recorrido en amplitud

Otra manera de recorrer un grafo es en amplitud: partiendo de un vértice inicial, en primer lugar se visita cada uno de sus vecinos; una vez hecho eso, se salta al primero de los vecinos y se recorren todos sus vecinos (salvo los ya visitados, como en el recorrido en profundidad); así sucesivamente. En este algoritmo, por tanto, se descubren a la vez todos los vértices que están a la misma distancia (número de saltos) del origen. En principio, en el recorrido en amplitud se exploran todos los nodos del grafo.

En el grafo de movilidad de ejemplo, el recorrido en amplitud para ir de la zona 1 a la 10 daría la siguiente estructura:



Recorrido en amplitud del grafo de movilidad para ir de la zona 1 a la 10.

Así, el camino resultante es [1, 2, 3, 5, 12, 11, 10]; también cruza el puente y luego usa el teletransporte pero al menos ahora no da la vuelta al bosquecillo. Pero ¿por qué da este rodeo? Si contamos las zonas, se visitan menos zonas con este camino que yendo por el camino más lógico ([1, 2, 3, 6, 7, 8, 9, 10]); y de momento suponemos que ir de una zona a cualquier otra zona vecina cuesta lo mismo. Así que ha encontrado el mejor camino posible, al menos teniendo en cuenta lo que sabe.

La programación de este método es más compleja que en el caso del recorrido en profundidad, ya que –entre otras cosas– requiere mantener una lista con los vértices que se han visitado pero que tienen vecinos pendientes de visitarse.

Por otra parte, si el peso asociado a cada arista es constante, es decir, si cuesta lo mismo moverse entre cualquier par de zonas vecinas porque tienen todas el mismo tamaño y no se diferencia por el tipo de terreno o similares, entonces este algoritmo sí es capaz de encontrar el camino mínimo entre dos nodos.

Sin embargo, esta es una limitación importante que lo restringe a juegos de tipo tablero o en los que el nivel esté estructurado en celdas homogéneas (por ejemplo, juegos de estrategia con mapa hexagonal), ya que en un nivel diseñado arbitrariamente tendremos zonas más grandes que otras y por tanto habrá diferentes costes para ir de una a otra, como en el ejemplo anterior.

2.2.4. Algoritmo de Dijkstra

Precisamente en la mayoría de los juegos nos encontraremos casi siempre con grafos de movilidad cuyas aristas llevan asociado un número o peso, que representa el coste de moverse de un vértice (zona) a otro, es decir, se trata de un grafo **ponderado**. Este coste puede ser simplemente la distancia en línea recta entre los centros de las dos zonas, o puede tener en cuenta otros factores, como la pendiente o la dificultad del tipo de terreno.

Grafo ponderado

En un grafo **ponderado** cada arista tiene asociado un valor numérico que por ejemplo representa el coste de moverse de un vértice a otro.

Como acabamos de explicar, ni el recorrido en profundidad ni el recorrido en amplitud pueden encontrar un camino óptimo entre dos vértices en un grafo ponderado. Edgser Wybe Dijkstra, matemático holandés y uno de los científicos más influyentes en el desarrollo de la programación moderna, desarrolló en 1959 un algoritmo para encontrar caminos óptimos entre vértices de grafos ponderados. Este algoritmo, que describiremos a continuación, recibe el nombre de algoritmo de Dijkstra.

La descripción del algoritmo es la siguiente:

Entrada: un grafo de N vértices, de los que x es el nodo origen.

1) Crear un vector D con N elementos que almacenará las distancias de x al resto de los vértices. Inicializar todos sus valores a infinito (distancia desconocida), salvo la entrada del propio vértice x, que valdrá 0.

2) Se toma como vértice actual $a = x$.

3) Recorrer todos los vértices vecinos de a, salvo los vértices marcados como completos.

4) Calcular la distancia tentativa desde el vértice inicial x hasta todos los vecinos del vértice actual a. Es decir, para cada vecino de a su distancia tentativa será la distancia desde x hasta a más la distancia de a a v_i: $dt(v_i) = D_a + d(a,v_i)$. Si esa distancia tentativa es menor que la distancia que había hasta ese momento para el vértice en el vector D, entonces se guarda esa distancia en D: $Dv_i = dt(v_i)$ (Recordemos que inicialmente las distancias en D son infinitas, así que al encontrar el primer camino a un vértice, se tomará como el mejor camino posible).

5) Marcar el vértice a como completo.

6) Tomar como siguiente vértice actual el de menor distancia en D y volver al paso 3 hasta que todos los vértices estén marcados como completos.

Salida: un vector D con N elementos, que son las distancias mínimas de x al resto de los vértices del grafo (en realidad también se devolvería el camino en sí, es decir otro vector con listas).

El algoritmo de Dijkstra genera todos los caminos posibles entre un vértice origen y el resto de los vértices del grafo. La ventaja de este algoritmo es, por tanto, que siempre encuentra el camino óptimo, aunque si el vértice origen cambia es necesario volver a ejecutar el algoritmo desde el principio.

El principal inconveniente es que al explorar todos los caminos posibles, incurre en un coste computacional bastante grande $O(|V|^2)$, donde $|V|$ es el número de vértices del grafo, en la solución básica. Eso hace que, si el diseño del juego lo permite, este algoritmo se utilice durante el desarrollo para calcular todos los caminos posibles, que quedarán almacenados para usarse más adelante. Sin embargo, esta estrategia no se puede aplicar en todos los juegos.

2.2.5. Algoritmo A*

Este algoritmo (se lee «A estrella» o «*A star*» en inglés) pretende aprovechar las ventajas del recorrido en profundidad y en amplitud para conseguir encontrar el camino óptimo con el mínimo número de operaciones. Además, es aplicable a grafos ponderados, lo que lo convierte en el algoritmo de búsqueda de caminos más utilizado en videojuegos.

Como el algoritmo en sí es algo complejo, empezaremos describiendo de manera más intuitiva los elementos que lo componen y cómo se comporta para ir buscando el camino óptimo. Los elementos fundamentales de A* son:

- Un grafo (posiblemente ponderado).
- Un vértice origen y un vértice destino.
- Una función $g(n)$ que devuelve el coste real de llegar desde el vértice origen al vértice n.
- Una función $h'(n)$ que devuelve un coste estimado de llegar desde el vértice n hasta el vértice final. Esta función recibe el nombre de **heurístico**, y en el caso de A* suele ser la distancia en línea recta entre el vértice n y el vértice final; el sentido de esta función es estimar de manera rápida un límite inferior a la distancia entre n y el vértice final, ya que si en línea recta hay una distancia d (digamos la distancia a vuelo de pájaro), la distancia real teniendo en cuenta los obstáculos será igual o mayor, pero nunca menor.
- La función de evaluación de un vértice n será $f(n) = g(n) + h'(n)$, es decir, el coste real de ir del origen a n más el coste mínimo estimado de ir de n hasta el final.
- Se mantiene una cola de prioridad de los vértices *abiertos* (aún no visitados), ordenados por su valor de $f(n)$; el siguiente vértice que se evalúa siempre es el de menor $f(n)$, y a cada paso del algoritmo se calcula la $g(n)$ de todos sus vecinos abiertos.
- Se mantiene una estructura de datos con los vértices *cerrados*, aquellos que ya se han evaluado.
- Se llegará al vértice final desde el vértice vecino con menor $f(n)$. Como ese vértice está junto al final, casi todo el valor de $f(n)$ se deberá al coste de llegar desde el origen, es decir, la parte $g(n)$. En otras palabras: se alcanzará el destino por el camino con menor coste desde el origen, que es lo que se buscaba.

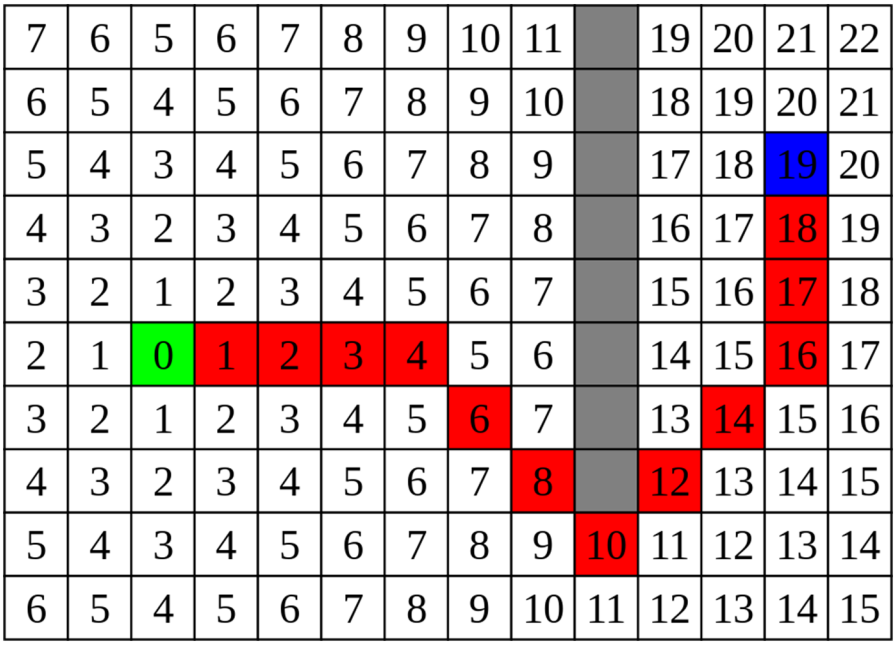
Función heurística

Una función **heurística** es una función inventada que sirve para calcular de manera aproximada el valor de otra función que no se conoce o resulta muy costoso calcular.

Un detalle importante de este algoritmo es que los vértices cuya $f(n)$ sea alta posiblemente no se evalúen nunca, ya que el algoritmo siempre da prioridad al nodo con menor $f(n)$. Ahí estriba la ventaja de A*.

La siguiente figura muestra un ejemplo de ejecución de A* sobre un mapa dividido en zonas cuadradas y con coste de movimiento constante para simplificar la figura.

Ejemplo de ejecución del algoritmo A*, partiendo de la celda verde hasta llegar a la azul. Se supone que el coste entre celdas adyacentes es siempre el mismo.



Fuente: CC WikiMedia Commons.

El principal inconveniente de A* es su relativamente elevado coste en memoria. Existen variantes subóptimas como IDA* (*iterative deepening A**) o SMA* (*simplified memory-bound A**) que sacrifican la optimalidad a cambio de reducir notablemente el uso de memoria.

1) A* en Unity

Unity utiliza A* para buscar los caminos óptimos en el grafo de movilidad (el NavMesh). El NavMesh se compone de un conjunto de polígonos convexos que representan las diferentes zonas de movilidad. A cada zona se le puede asignar un coste específico, de manera que cueste más avanzar por arena que por asfalto, por ejemplo.

Una vez definido el NavMesh, hay que crear un *script* para indicarle al agente que se mueva a la posición de un objeto determinado. El código que realiza esa operación es el siguiente:

```
// MoveTo.cs
   using UnityEngine;
   using System.Collections;

   public class MoveTo : MonoBehaviour {

      public Transform destino;

      void Start () {
         NavMeshAgent agente = GetComponent<NavMeshAgent>();
         agente.destination = destino.position;
```

```
            }
        }
```

2.2.6. Otras variaciones de la búsqueda de caminos

Si bien A* se utiliza habitualmente en juegos con niveles estándar, hay algunas situaciones específicas en las que no resulta práctico aplicarlo directamente, o bien resultaría muy caro computacionalmente.

1) Búsqueda de caminos jerárquica

Cuando el juego (o aplicación, en general) tiene un mapa muy extenso (pensemos en MMORPG como World of Warcraft, con continentes enteros, o en aplicaciones como Google Maps), es inviable aplicar A* a todo el mapa de juego a la vez. Por ello, lo que se hace es aplicar una jerarquía a los mapas o niveles del juego, por ejemplo dividiendo el mundo en continentes, cada continente en regiones y cada región posiblemente en varias zonas; además, dentro de cada casa o cada cueva se definirán nuevas zonas. De esa manera, para encontrar el camino para ir de un punto a otro del mundo, en primer lugar se buscaría cómo ir de un continente a otro, una vez en el continente de destino se buscaría la región de destino, etc. En cada nivel de complejidad se puede aplicar A*, ya que el número de vértices del grafo será limitado. Y posiblemente el significado de las aristas entre vértices sea diferente según el nivel en la jerarquía porque, por ejemplo, la conexión entre continentes puede referirse a la existencia de barcos para moverse entre ellos.

MMORPG

Un MMORPG es un juego de rol masivo multijugador en línea (*massive multiplayer online role playing game*). Lo de *masivo* se refiere a que puede haber una gran cantidad (miles por servidor) de jugadores conectados simultáneamente. El software de los servidores de este tipo de juegos y el protocolo de comunicación requieren un diseño muy cuidadoso para permitir que los jugadores interactúen en tiempo real y sin colapsar el sistema.

2) Búsqueda de caminos dinámica

En muchas situaciones un agente no puede calcular su camino completo cuando se le da la orden de moverse. En muchos juegos de estrategia, por ejemplo, existe la llamada «niebla de guerra» (*fog of war*), que oculta las zonas de mapa que aún no se han explorado. Sin embargo, el jugador puede clicar en una zona oculta y ordenar a su agente que vaya allí (por ejemplo, para explorar). Se supone que la IA del juego no ha de hacer trampas y debe respetar la niebla de guerra y otras limitaciones que también tenga el jugador, así que irá descubriendo camino a medida que vaya entrando en terreno desconocido.

En otros casos, puede haber obstáculos móviles que impidan el movimiento que se había calculado (una unidad bloqueando un puente, por ejemplo) y que obliguen a recalcular la ruta sobre la marcha. Este tipo de algoritmos también se aplican frecuentemente en robótica para planificar los movimientos del robot.

En estas situaciones se aplican algoritmos de búsqueda de caminos dinámica, como por ejemplo el D* (*Dynamic A**). Sin entrar en mucho detalle, las características de este algoritmo, del que existen diferentes variantes, son las siguientes:

- Se asume que la zona desconocida no tiene obstáculos y se empieza el camino con esta suposición.
- Su estructura básica es similar al A*.
- Los vértices, además de estar abiertos y cerrados, pueden encontrarse en otros estados: nuevo (aún no considerado de ninguna manera), subido (su coste ha subido desde la última evaluación por los cambios dinámicos) y bajado (su coste ha bajado). De esa forma se gestiona la dinámica del mapa.
- A diferencia de A*, D* comienza la exploración desde el vértice destino.

Ejemplo de niebla de guerra en el juego de estrategia por turnos Battle for Wesnoth. (libre)

Fuente: CC WikiMedia Commons.

2.3. Movimientos complejos

Hasta ahora hemos tratado la gestión de movimientos suponiendo que un agente quiere moverse a una posición determinada, pero en los juegos a menudo los agentes se deben mover unos en relación con otros. Ejemplos de estas situaciones: interceptar un balón en movimiento, disparar a un avión calculando el punto en el que se encontrará, perseguir al jugador por una habitación, mover un pelotón de soldados de forma flexible para adaptarse a la an-

chura de las calles y para cambiar su formación al llegar al encontrarse con el enemigo. En este apartado estudiaremos algunos de estos tipos de movimiento y cómo se pueden resolver.

2.3.1. Movimientos de dirección

Los movimientos de dirección (en inglés, *steering behaviours*) son aquellos en los que el movimiento ha de permitir interceptar, perseguir o evadir a otros agentes, o simplemente vagabundear de una manera que resulte natural.

1) Perseguir y huir

Una de las situaciones más habituales en los juegos consiste en hacer que persigan al jugador o a otro agente. Es decir, el destino del movimiento no es un punto fijo del nivel de juego sino la posición de otro agente.

«Suelo»

Aunque gran parte de los juegos son en tres dimensiones, en la mayoría de los casos hay un «suelo» de referencia que hace que el movimiento se calcule en realidad sobre las dos dimensiones del suelo. Solo en juegos en los que hay mayor libertad de movimientos en altura, como en juegos con aviones o naves espaciales, se utilizan plenamente las tres dimensiones para calcular los movimientos.

La posición, la velocidad y la aceleración de los agentes en el juego se representan mediante vectores de dos o tres componentes según las dimensiones en las que se desarrolle el movimiento.

En cualquiera de los dos casos, al poner una flecha sobre una letra ($\vec{A}$) queremos indicar que se trata de un vector.

Supongamos un agente A que quiere perseguir a otro agente B. Por tanto, el destino del movimiento de A será la posición de B, $\vec{P}_B$. Para calcular la dirección en la que deberá moverse A (o sea, su vector de velocidad), es necesario tener en cuenta su posición actual y su destino, es decir:

$$\overrightarrow{V_A} = \overrightarrow{P_B} - \overrightarrow{P_A}$$

Aunque como A tendrá una celeridad (módulo de su velocidad) máxima V_{MAX}, esta expresión se debe ajustar de la siguiente manera:

$$\overrightarrow{V_A} = \frac{\overrightarrow{P_B} - \overrightarrow{P_A}}{|P_B - P_A|} \cdot V_{MAX}$$

En el caso de querer huir de un agente, simplemente hay que cambiar el signo del vector $\overrightarrow{V_A}$ para que A se mueva en dirección opuesta a B y, por tanto, huya de él.

2) Interceptar

En el ejemplo de perseguir o huir hemos supuesto que el agente destino, B, estaba quieto. Sin embargo, si B se mueve, cuando A llegue a su posición B ya no estará allí y tendrá que volver a moverse hacia la nueva posición de B, que mientras tanto seguirá moviéndose, y así sucesivamente. En definitiva, dará una sensación muy torpe a los jugadores que lo estén viendo. Además, es especialmente importante tener en cuenta ese detalle cuando se quiere lanzar un objeto tipo proyectil para que acierte en un objetivo móvil.

La idea básica para interceptar un objeto en movimiento es estimar su posición futura a partir de su velocidad actual, y calcular en qué punto el agente puede alcanzar el objetivo teniendo en cuenta la velocidad máxima que puede alcanzar.

Sea $\overrightarrow{P_A}$ la posición del agente A, $\overrightarrow{P_B}$ la del agente objetivo B y $\vec{V}_B$ su velocidad; lo que queremos calcular es la velocidad necesaria $\overrightarrow{V_A}$ para que A coincida con B en el punto de interceptación, es decir, que los dos agentes estén en el mismo punto $\vec{P}_X$.

$$\begin{cases} \overrightarrow{P_X} - \overrightarrow{P_A} = t \cdot \overrightarrow{V_A} \\ \overrightarrow{P_X} - \overrightarrow{P_B} = t \cdot \overrightarrow{V_B} \end{cases}$$

Como también deben coincidir en el tiempo, *t* debe ser la misma en las dos ecuaciones, así que igualamos y se obtiene:

$$\overrightarrow{V_A} = \frac{\overrightarrow{P_A} - \overrightarrow{P_B}}{t} + \overrightarrow{V_B}$$

De esta forma se obtiene la velocidad que debe adoptar A para alcanzar a B.

3) Movimientos de dirección en Unity

Cuando trabajemos con niveles con obstáculos en Unity, que requieran el uso del NavMesh, no calcularemos directamente los vectores de velocidad, sino que adaptaremos los movimientos de dirección a la búsqueda de caminos de NavMesh. Así, en el caso de la persecución, lo que tendremos que hacer será actualizar constantemente el destino del movimiento a la posición actual del objetivo.

El método *Update()* del comportamiento de los agentes de Unity se llama en cada refresco de fotograma; simplemente se puede recalcular el movimiento a partir de la última posición conocida del objetivo:

```
// MoveTo.cs
   using UnityEngine;
   using System.Collections;
```

```
    public class MoveTo : MonoBehaviour {

       public Transform objetivo;

       void Start () {
// En Start() no hace falta hacer nada porque al primer Update() ya se
// calculará el camino
       void Update () {
         NavMeshAgent agente = GetComponent<NavMeshAgent>();
         agente.destination = objetivo.position;
       }
      }
```

Solo hay que tener en cuenta el coste computacional que puede implicar esta solución, ya que posiblemente en cada fotograma se recalcula el camino; si el número de zonas es elevado y hay muchos agentes haciendo esto, puede acabar resultando una sobrecarga notable.

Posibles mejoras: no recalcular en cada fotograma, sino cada *n* fotogramas o cada *p* milisegundos, y también almacenar la posición del destino y solo recalcular el camino cuando haya cambiado significativamente.

Por lo que respecta a la interceptación, necesitamos averiguar la velocidad del objetivo además de su posición. Si por ejemplo se está utilizando el motor de física de Unity, los objetos controlados por el motor son de clase *Rigidbody* y la propiedad *velocity* es el vector de velocidad del objeto; a partir de ahí podemos determinar su trayectoria y su posición en el futuro. Por otra parte, si se está utilizando NavMesh, es la propiedad *NavMeshAgent.velocity* la que devuelve el *Vector3* con la velocidad del agente a lo largo de su camino.

2.3.2. Movimientos colectivos (*flocking*)

Los agentes de juego no siempre funcionan de manera independiente, sino que en ocasiones se mueven varios agentes en grupo: rebaños o bandadas de animales, pelotones de soldados o de deportistas, formaciones de aviones o naves, etc.

En ese tipo de situaciones no es viable calcular un camino independientemente para cada agente del grupo por dos motivos:

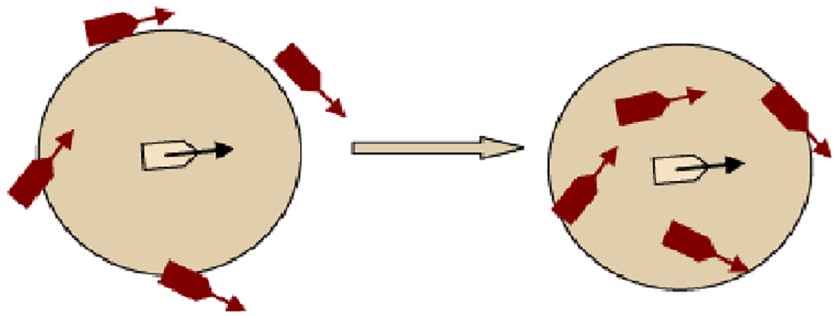
- El excesivo coste computacional repitiendo básicamente la misma operación (mismo origen y destino, con un pequeño desplazamiento).

- Si cada agente siguiera su camino, a menudo chocarían entre ellos y se entorpecerían; además, sería imposible definir un movimiento ordenado, tipo formación.

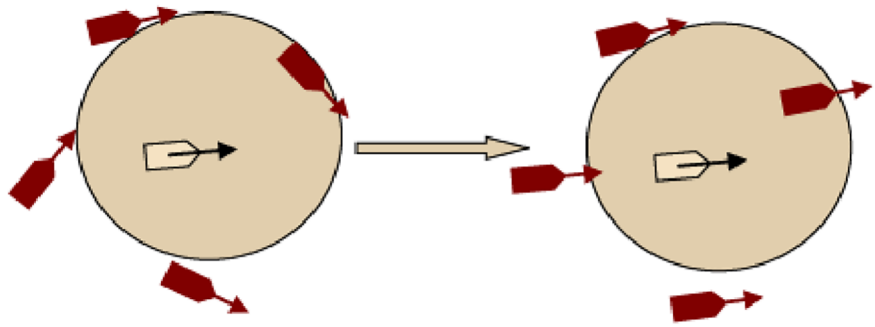
Por otra parte, tampoco se puede considerar que el grupo sea un único agente de mayor tamaño, ya que a veces el grupo tendrá que pasar por zonas más estrechas y adaptarse, y en general es deseable que el movimiento de los agentes resulte natural y no sea completamente rígido.

Por esos motivos es necesario utilizar otras estrategias que permitan modelar el movimiento esperado en grupos de agentes. En 1987 Craig Reynolds definió el primer algoritmo de movimiento colectivo (*flocking*), en el que bautizó como *boids* a los agentes que forman el grupo. El algoritmo de Reynolds da muy buenos resultados y tiene como característica que no hay un líder del grupo. La belleza de este algoritmo proviene en gran medida de su simplicidad, pues funciona según estos tres principios:

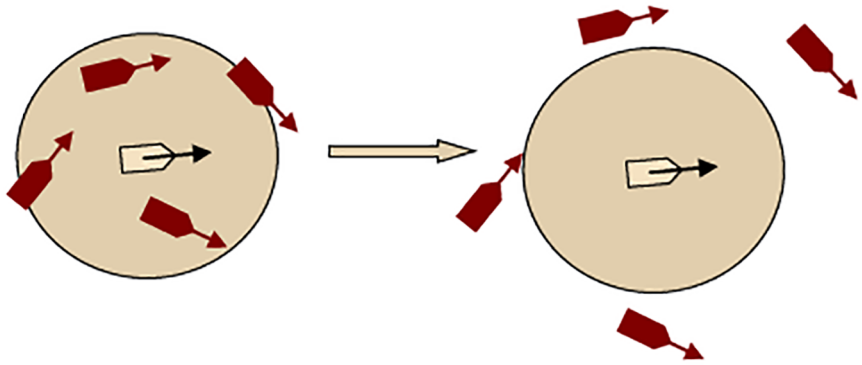
1) Cohesión: cada unidad se dirige hacia la posición media de sus vecinos.



2) Alineamiento: cada unidad se alinea según la alineación media de sus vecinos.



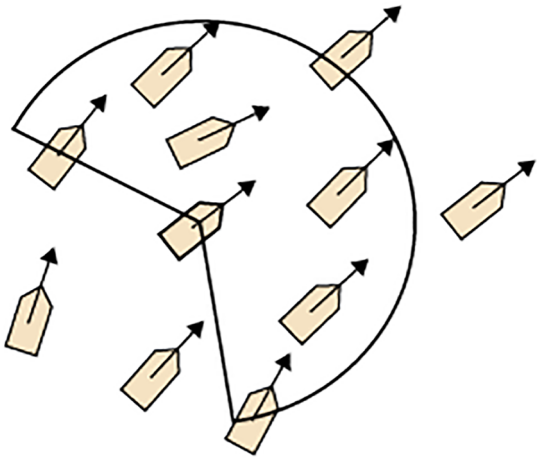
3) Separación: cada unidad ajusta su dirección para evitar colisionar con sus vecinos.



Nota

Se ha demostrado científicamente que los pájaros deciden su movimiento mirando a sus cinco vecinos más cercanos.

Para que los agentes puedan cumplir esas propiedades, necesitan conocer la posición de sus vecinos en todo momento; para conseguir un comportamiento realista, se define un **campo de visión** (*field of view*) que controla qué vecinos puede ver el agente, y por tanto según qué agentes se guiará.



Si el campo de visión es ancho, los agentes formarán grupos anchos; por el contrario, con campos de visión más estrechos los agentes tenderán a ir en grupos estrechos, tipo fila o caravana.

Web recomendada

El algoritmo de *flocking* para Unity se puede encontrar en http://wiki.unity3d.com/index.php?title=Flocking.

3. Toma de decisiones

Sin duda una de las características de la IA de un juego que más impresionará a los jugadores es su capacidad de tomar decisiones como si se tratara de una persona, o al menos demostrando inteligencia. Un ejemplo muy claro de ello son los juegos de estrategia: si la IA se muestra torpe y toma decisiones contraproducentes todo el rato, se arruinará la gracia del juego.

Además, la toma de decisiones no solo se produce en los agentes contra los que juega el jugador humano; muchas veces hay elementos controlados por la IA que colaboran con el jugador, como acompañantes, y las decisiones que tomen pueden ser críticas para el desarrollo del juego.

Nota

Conviene recordar que, en última instancia, el objetivo de la IA en un juego es conseguir que el jugador se divierta jugando; y para lograrlo, en general, es necesario que la IA sea capaz de tomar decisiones acertadas y que den la impresión de inteligencia reaccionando adecuadamente a las acciones del jugador.

En prácticamente todos los juegos la IA debe tomar decisiones constantemente: cuándo usar el turbo, cuándo activar el escudo de fuerza, atacar o huir, construir un cañón o un tanque, golpear el balón con el pie o con la cabeza, pasarlo a otro agente o tirar a portería…

Hay que tener en cuenta, de todos modos, que en algunos casos los agentes de juego pueden seguir un comportamiento fijo, con una serie de pasos que se repiten exactamente, y en los que la dificultad y el interés para los jugadores estriba en aprenderse bien los pasos y perfeccionar la estrategia. Un ejemplo claro de esto son los jefes (*bosses*) de World of Warcraft, cuyo comportamiento es siempre igual y pasa por las mismas fases, bien conocidas y documentadas.

Combate con un jefe de *raid* en World of Warcraft

Fuente: © Blizzard Entertainment.

Hay algunos aspectos generales que conviene considerar en la toma de decisiones. Por una parte, se suelen diferenciar según la escala temporal en la que tendrá efecto la acción:

- Decisiones **estratégicas**: son las de mayor alcance en el tiempo, y normalmente afectan al desarrollo completo de una partida. Esto implica que son las decisiones más complejas de tomar, ya que requieren tener en cuenta más datos de entrada y han de prever sus posibles consecuencias más lejos en el futuro. Ejemplos: en un partido de fútbol si se va a jugar a la defensiva o al ataque; en un juego de estrategia si se va a fortificar una posición, se va a atacar rápidamente o se va a intentar conquistar varias posiciones alrededor del jugador.

- Decisiones **tácticas**: son decisiones en un marco de tiempo intermedio, pensadas para llevar a cabo la estrategia decidida previamente. Normalmente se toman muchas decisiones tácticas en cada partida. Ejemplos: construir una muralla o unos barracones, parar en boxes o intentar dar una vuelta más en una carrera de fórmula 1, asignar un jugador para marcar a otro en un partido de baloncesto.

- Decisiones **operacionales**: decisiones muy concretas para llevar a cabo las decisiones tácticas; pueden tomarse muchas de ellas cada segundo y dependen de información local muy específica del momento. Ejemplos: usar un poder especial en un momento dado, tirar el balón a portería o pasárselo a un compañero.

Otro aspecto importante que se debe tener en cuenta es la **información** que conoce la IA. Se supone que debe «jugar limpio» y evitar usar información privilegiada, como por ejemplo saber con detalle todo lo que hay en el mapa mientras que el jugador no lo sabe si sale a explorar. No obstante, algunos juegos se aprovechan de toda la información disponible (qué está haciendo el jugador en cada momento) como forma fácil de mejorar las prestaciones de la IA sin necesidad de utilizar técnicas muy complejas; es lo que se conoce como IA **omnisciente**.

También puede haber decisiones **proactivas**, en las que el agente toma la iniciativa y sabe que debe tomar una decisión, y **reactivas**, en las que a causa de un cambio externo el agente reacciona y genera una respuesta.

Por último, hay que considerar el **determinismo** que queremos que ofrezca el juego, es decir, si la respuesta de los agentes siempre será la misma en las mismas condiciones o si, por el contrario, hay un cierto factor de **aleatoriedad** en la respuesta para que no sea siempre igual. Esta decisión depende del tipo de juego que se quiera crear y de dónde se quieren poner los retos a los jugadores.

Yendo un paso más allá, en el capítulo 4 veremos métodos de IA que son capaces de aprender de lo que hace el jugador y adaptarse a su estilo de juego para conseguir ofrecer siempre una experiencia emocionante. En ese caso estaríamos hablando de comportamientos **aprendidos**.

3.1. El proceso de toma de decisiones

En líneas generales, los pasos que debe seguir la IA para tomar una decisión son siempre los mismos, adaptándolos, eso sí, a las características propias de cada juego, ya que en algunos sistemas algunos pasos se omitirán.

1) Recopilar la información necesaria para identificar la situación actual y así saber cuáles son las opciones posibles. El tipo y cantidad de información que hay que recopilar dependerá del nivel de complejidad de la decisión que se debe tomar.

2) Identificar los criterios de decisión y asignarles una ponderación o prioridad. Por ejemplo, una nave espacial debe atender al mantenimiento general de sus sistemas, pero en combate debe dar prioridad a la información relacionada (naves enemigas detectadas, etc.).

3) Determinar las posibles alternativas. En este punto es donde más difieren los métodos que se presentarán a continuación. En general, hay que calcular qué posibles decisiones son viables teniendo en cuenta los dos puntos anteriores.

4) Valorar las alternativas encontradas en el punto anterior, asignándoles una puntuación que permita decidir entre una u otra. En el caso de sistemas deterministas siempre se elegirá la alternativa de mayor puntuación; en sistemas más aleatorios habrá una probabilidad de elegir otra opción.

5) Ejecución de la decisión, llevando a cabo las acciones necesarias (apretar el freno, disparar, pasar el balón…).

6) Evaluar los resultados: los algoritmos de aprendizaje (apartado 4) valoran los resultados de sus acciones y aprenden de ellos. Los métodos presentados en este apartado en general no realizan este último paso.

3.2. Sistemas de reglas

Los sistemas de reglas son una de las técnicas más sencillas para programar el comportamiento de los agentes, si bien se utilizan con mucha frecuencia, ya que en numerosas situaciones dan una respuesta rápida y efectiva. Los elementos que componen estos sistemas son tres:

- Un conjunto de condiciones posibles en el juego.
- La respuesta esperada para cada una de las condiciones anteriores.
- Un conjunto de instrucciones *if…else* que evalúa cada una de las condiciones posibles y, en caso de cumplirse, ejecuta la respuesta.

El comportamiento de un monstruo en un juego de disparos en primera persona se podría definir mediante las siguientes condiciones y acciones:

- Si no hay ningún jugador cerca, vagabundea.
- Si oyes ruidos, ve a investigar.
- Si ves a un jugador, ve hacia él.
- Si estás junto a un jugador, atácalo.
- Si tu salud está por debajo del 25 %, huye y escóndete.

Puede ocurrir que en un momento dado se cumplan varias reglas (el monstruo está junto a un jugador y su salud está por debajo del 25 %), lo que requiere que se aplique alguna estrategia de resolución de conflictos para decidir qué regla elegir. Hay diferentes estrategias de resolución de conflictos:

- Ejecutar la primera regla del subconjunto de reglas en conflicto.
- Ejecutar aleatoriamente cualquiera de las reglas del subconjunto.
- Asignar una valoración (fija o dependiente de factores externos) a cada regla y seleccionar la regla con mayor valoración.

En cualquiera de estos casos, las reglas se están utilizando en el sentido habitual de una instrucción *if…then*: si se cumple la condición, se ejecuta la regla; esto es lo que se conoce como **encadenamiento hacia adelante** (*forward chaining*).

No obstante, el conocimiento almacenado en el conjunto de reglas también se puede aprovechar en sentido contrario, para que la IA pueda deducir información sobre el estado del juego. En el ejemplo anterior, puede saber que si otro monstruo está atacando al jugador es que su salud está por encima del 25 %, por ejemplo. Esta técnica se conoce como **encadenamiento hacia atrás** (*backward chaining*) y es muy útil para que la IA deduzca información (cuando no es omnisciente). Por ejemplo, en los juegos de estrategia suele haber tecnologías estructuradas en forma de árbol, de manera que para poder utilizar una tecnología es necesario haber desarrollado previamente otras tecnologías, por ejemplo la pólvora para poder fabricar cañones. Aplicando *backward chaining*, si la IA ve que el jugador tiene cañones puede deducir que también tiene pólvora, información que podrá usar en futuras decisiones.

Los sistemas basados en reglas funcionan bien en situaciones relativamente sencillas (principalmente en decisiones tácticas y operacionales). Otra ventaja es que son fáciles de programar. Por el contrario, cuando el número de reglas aumenta puede resultar difícil gestionarlos y evitar contradicciones o situaciones imprevistas. Además, si no se encuentra una condición que se cumpla, el sistema no puede generar una respuesta razonable: no hay lugar para la improvisación.

3.3. Máquinas de estados finitos

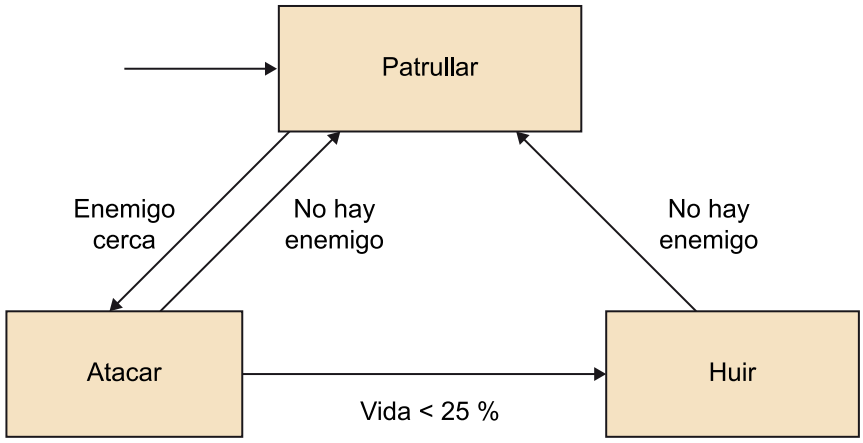
Las máquinas de estados finitos (*finite state machines*, FSM) son modelos de comportamiento que se componen de dos elementos:

- Un conjunto de estados (finito, no puede haber infinitos estados), que son las diferentes situaciones en las que puede encontrarse el agente.

- Un conjunto de transiciones, que son condiciones que hacen que el agente pase del estado A al B. A diferencia de los sistemas basados en reglas, la transición que se haga depende del estado actual del agente.

La siguiente figura muestra un sencillo diagrama de estados de un soldado en un juego. La flecha que viene de ninguna parte y apunta a «Patrullar» indica que ese es el estado inicial del agente.

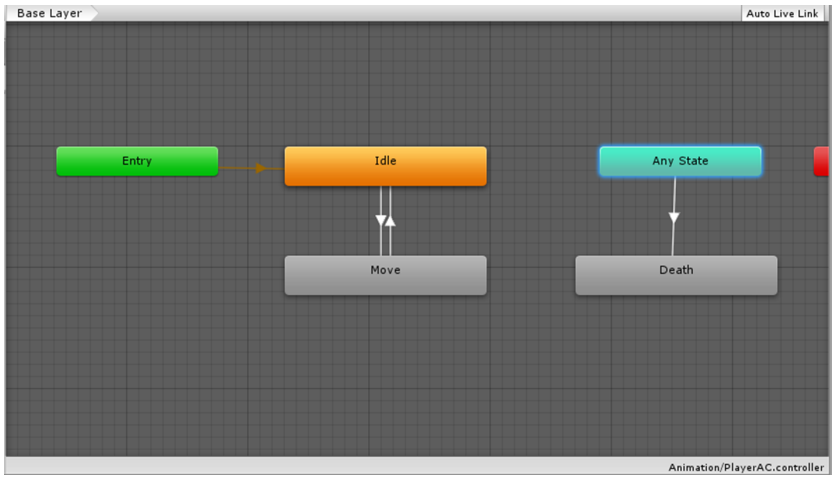
Diagrama de estados de un soldado en un juego.



Los FSM

Los FSM son un patrón de diseño de software que se utiliza en innumerables situaciones en las que se quiera modelar un sistema que tiene un estado interno que cambia en reacción a diferentes condiciones.

El propio Unity utiliza los FSM en otros lugares, como por ejemplo los *Animator*, en ese caso para modelar las diferentes animaciones de un modelo y en qué condiciones se ha de pasar de una a otra.



Es posible (aunque no recomendable desde un punto de vista de ingeniería del software) utilizar la maquinaria de estados de las animaciones para controlar el comportamiento del agente.

Otra ventaja de los FSM es que ayudan a diseñar correctamente el comportamiento de un agente, ya que obligan a definir sus posibles estados y a analizar en qué circunstancias se pasa de un estado a otro. Es decir, son a la vez una herramienta de diseño de software y una forma de realizar ese diseño en el lenguaje de programación deseado. Como se verá en el código del ejemplo siguiente, la estructura de clases utilizada para programar un FSM organiza los datos y operaciones, dividiéndolos en una parte común a todos los estados y otra particular de cada estado.

Por su especial importancia vamos a ver el código de un FSM para el diagrama propuesto en la figura anterior (estados Patrullar, Atacar, Huir).

En primer lugar se define una clase, FSMSoldado, que modela el comportamiento del soldado mediante el FSM de la figura. Dentro de la clase se definen los tres estados del soldado más un atributo que indica cuál es el estado actual. El método *Start()* se aprovecha para asignar el estado inicial (*EstadoPatrullar*), y en el método *Update()* se pide al estado actual que revise su situación por si es necesario cambiar a otro estado.

```
using UnityEngine;
using System.Collections;

public class FSMSoldado: MonoBehaviour
{
    // Aquí iría la información del soldado común a todos los estados



    // Acceso a los estados
    [HideInInspector] public IEstadoSoldado estadoActual;
    [HideInInspector] public EstadoPatrullar estadoPatrullar;
    [HideInInspector] public EstadoAtacar estadoAtacar;
    [HideInInspector] public EstadoHuir estadoHuir;
```

```
    // En este método se crean los objetos que representan cada uno de los tres estados
    private void Awake()
    {
        estadoPatrullar = new EstadoPatrullar (this);
        estadoAtacar = new EstadoAtacar (this);
        estadoHuir = new EstadoHuir (this);

        navMeshAgent = GetComponent<NavMeshAgent> ();
    }

    // Aquí se indica cuál es el estado inicial
    void Start ()
    {
        estadoActual =  estadoPatrullar;
    }

    // En cada fotograma se comprueba si hay que cambiar a otro estado
    void Update ()
    {
        estadoActual.ActualizaEstado();
    }

    private void OnTriggerEnter(Collider other)
    {
        currentState.OnTriggerEnter (other);
    }
}
```

Como vamos a crear tres clases diferentes (para cada uno de los tres estados), pero el atributo *estadoActual* puede referirse a cualquiera de ellas, vamos a definir un *interface* de C# que incluya las operaciones comunes a todos los estados, que son cuatro: una para la llamada de actualización que se hace desde el método *Update()* del FSM, y tres para ejecutar la transición a cada una de los estados.

```
using UnityEngine;
using System.Collections;

public interface IEstadoSoldado
{
    void ActualizaEstado();

    void AEstadoPatrullar();

    void AEstadoAtacar();
```

```
    void AEstadoHuir();
}
```

El estado de Patrullar no cambia hasta que no se detecta a un enemigo durante la patrulla. Como se acaba de explicar, todos los estados implementan el *interface* IEstadoSoldado.

```
using UnityEngine;
using System.Collections;

public class EstadoPatrullar : IEstadoSoldado
{
    // Referencia al FSM para poder ordenarle que cambie de estado
    private readonly FSMSoldado fsm;

    // Aquí iría otra información útil: siguiente punto en el recorrido del soldado, etc.

    // Anotar el objeto FSM para poder pedirle cambios de estado
    public EstadoPatrullar (FSMSoldado fsmSoldado)
    {
        fsm = fsmSoldado;
    }

    // Aquí hacer todo lo que se supone que hace el soldado al patrullar: ir moviéndose, etc.
    public void ActualizaEstado()
    {
       // Elegir siguiente punto de la ruta, moverse, vigilar; si se detecta al jugador, ir a por él.
        if(se detecta al jugador de la forma que sea) {
            AEstadoAtacar();
        }
    }

    // Métodos que ejecutan las transiciones a otros estados
    public void AEstadoPatrullar()
    {
        Debug.Log ("No se puede cambiar al mismo estado.");
    }

    public void AEstadoAtacar()
    {
        fsm.estadoActual = fsm.estadoAtacar;
    }

    public void AEstadoHuir()
    {
        fsm.estadoActual = fsm.estadoHuir;
    }
```

```
}
```

El estado Atacar tiene dos transiciones posibles, a Huir o a Patrullar.

```
using UnityEngine;
using System.Collections;

public class EstadoAtacar : IEstadoSoldado
{
   private readonly FSMSoldado fsm;

   // Otra información útil para este estado: tipo de ataque, etc.

   // Anotar el objeto FSM para poder pedirle cambios de estado
   public EstadoAtacar (FSMSoldado fsmSoldado)
   {
      fsm = fsmSoldado;
   }

   // Aquí hacer todo lo que se supone que hace el soldado al atacar
   public void ActualizaEstado()
   {
      // Realizar el ataque en sí
      // Transiciones
      if(se pierde de vista al jugador) {
         AEstadoPatrullar();
      }
      else if(la salud < 25%) {
         AEstadoHuir();
      }
   }

   // Métodos que ejecutan las transiciones a otros estados
   public void AEstadoPatrullar()
   {
      fsm.estadoActual = fsm.estadoPatrullar;
   }

   public void AEstadoAtacar()
   {
      Debug.Log ("No se puede cambiar al mismo estado.");
   }

   public void AEstadoHuir()
   {
      fsm.estadoActual = fsm.estadoHuir;
   }
```

```
}
```

Por último, el estado Huir solo tiene una transición, volver a Patrullar.

```
using UnityEngine;
using System.Collections;

public class EstadoHuir : IEstadoSoldado
{
   private readonly FSMSoldado fsm;

   // Otra información útil para este estado: hacia dónde se huye, etc.

   // Anotar el objeto FSM para poder pedirle cambios de estado
   public EstadoHuir (FSMSoldado fsmSoldado)
   {
      fsm = fsmSoldado;
   }

   // Aquí hacer todo lo que se supone que hace el soldado al huir
   public void ActualizaEstado()
   {
      // Determinar a dónde huir, buscar el camino, correr por el camino
      // Transición
      if(se pierde de vista al jugador) {
         AEstadoPatrullar();
      }
   }

   // Métodos que ejecutan las transiciones a otros estados
   public void AEstadoPatrullar()
   {
      fsm.estadoActual = fsm.estadoPatrullar;
   }

   public void AEstadoAtacar()
   {
      fsm.estadoActual = fsm.estadoAtacar;
   }

   public void AEstadoHuir()
   {
      Debug.Log ("No se puede cambiar al mismo estado.");
   }
}
```

Nota

Si hay muchos agentes que usan el mismo FSM, se puede ahorrar memoria creando cada estado solo cuando se necesite en lugar de crearlos y mantenerlos todos. Mejor aún, se pueden definir los estados como *Singleton* y pasarles la referencia del FSM al llamar a su *ActualizaEstado()*.

Como se puede ver en el código, el método *ActualizaEstado*() de cada estado es el responsable de ir ejecutando el comportamiento esperado del agente de ese estado y de decidir si es necesario cambiar a otro estado. En cada estado solo se programan las salidas posibles (transiciones a otros estados), no tiene importancia de qué estado se venga.

Como posibles desventajas de los FSM se encuentra que no es posible combinar diferentes estados, lo que impide modelar algunos comportamientos que resultan de combinaciones de varios estados o características. Por ejemplo, en un juego de estrategia la IA puede estar construyendo un aserradero y a la vez entrenando soldados, pero no hay ningún estado que represente esas dos variables, ya que las combinaciones posibles son tantas que serían necesarios muchísimos estados. Además, los estados y las transiciones, en general, se predefinen al escribir el programa, es decir, que no se pueden añadir estados nuevos dinámicamente.

Los FSM son deterministas, es decir, partiendo de un estado y si se dan las mismas condiciones, el modelo siempre cambiará al mismo estado. Esto puede provocar que las respuestas del juego sean demasiado predecibles. Se pueden generalizar los FSM añadiendo una probabilidad a cada transición, de modo que si en un momento dado hay varias transiciones posibles desde el estado actual, se elija una de ellas al azar.

Los FSM se pueden aplicar a diferentes niveles de decisión (estratégica, táctica, operacional). De hecho, es posible anidar un FSM dentro de un estado de otro FSM. Algo así es lo que se conoce como **árboles de comportamiento**, que veremos más adelante.

3.4. Exploración en árbol

Los **árboles** son estructuras en las que se parte de un elemento llamado **nodo raíz**, del que parten dos (puede que más) **enlaces** que conducen a otros **nodos**, que o bien pueden ser **internos** (o ramas) si de ellos parten nuevos enlaces, o bien **nodos hoja** o **terminales** si de ellos no parten enlaces. A diferencia de los grafos, en los árboles no puede haber ciclos: los nodos están organizados jerárquicamente y no es posible crear un enlace que vuelva a un nodo previo. Habitualmente un nodo que proviene de otro se denomina nodo **hijo** del primero, y este lógicamente es su **padre**.

La programación de videojuegos

La programación de videojuegos ofrece numerosas situaciones que se resuelven de manera natural mediante estructuras con forma de árbol. Un ejemplo fuera de la IA son las escenas y sus objetos: una escena en Unity es la raíz del árbol, cada objeto de la escena es un nodo que proviene de la escena, a su vez los objetos asociados a otros objetos (las ruedas de un coche, la espada de un guerrero) son nodos hijos de los objetos a los que van asociados.

En este apartado veremos diferentes aplicaciones de los árboles para dotar de inteligencia a nuestros juegos.

3.4.1. Árboles de decisión

Los **árboles de decisión** son un método de decisión o clasificación que permite encadenar un conjunto de condiciones (como las de los sistemas basados en reglas) de manera estructurada y jerarquizada. Generalmente se empieza planteando la condición más importante (la que permite distinguir el mayor número de casos), y dentro de cada alternativa se van planteando otras condiciones, en las que a su vez habrá otras condiciones, y así sucesivamente.

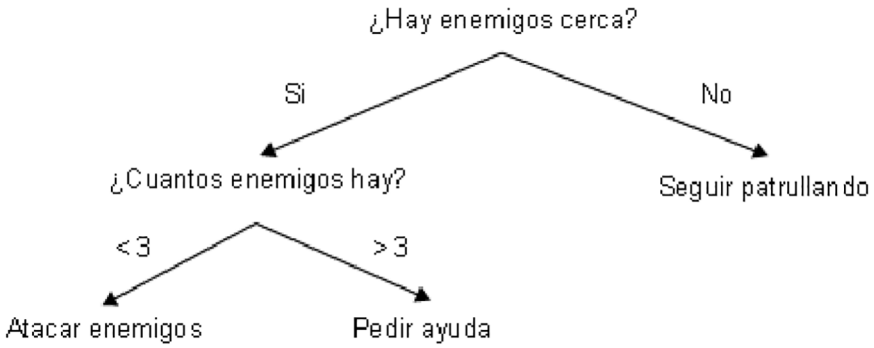
Desde el punto de vista de la programación, un árbol de decisión puede programarse mediante un conjunto de *if…else* anidados. La secuencia de decisiones, representada gráficamente, da lugar a un árbol, de ahí el nombre del método.

Métodos de clasificación

Los árboles de decisión, así como otros métodos que iremos estudiando, son **métodos de clasificación**, que son capaces de clasificar una muestra de datos a partir de sus características. Por ejemplo, clasificar un coche en deportivo, familiar, todo terreno, etc., en función de sus características: tipo de motor, tamaño de rueda, etc. Se puede aplicar esa capacidad al problema de la toma de decisiones, haciendo que los métodos tomen como entrada la situación actual del juego y den como resultado la clase de acción que se debe ejecutar.

Un ejemplo de árbol de decisión sería el siguiente:

Ejemplo sencillo de árbol de decisión.



Los árboles de decisión tienen dos tipos de elementos:

- Los nodos **internos**, que son las condiciones que debe evaluar el método. Los nodos internos pueden ser **deterministas** –si en las mismas situacio-

Nota

En este apartado hemos visto árboles de decisión **prediseñados**, en los que el programador fija las condiciones y el orden en el que se ejecutarán; el programa las ejecuta tal cual. En el apartado 4 veremos que los árboles de decisión pueden entrenarse para que aprendan por ellos mismos qué decisiones les conviene tomar en cada momento.

nes siempre toman la misma decisión–, o **probabilísticos** –si su decisión depende parcial o totalmente del azar.

- Los nodos **finales** u hojas, que son las decisiones finales que tomará el árbol, es decir, el resultado de la evaluación del árbol.

Si queremos utilizar un árbol de decisión en nuestro juego, debemos tener en cuenta que se distinguen dos etapas:

1) Diseño del árbol, etapa en la que se ha de decidir cuáles son las decisiones más importantes y ponerlas al inicio del árbol, y dejar las decisiones menos influyentes hacia el final. Esto se hace al programar el juego.

2) Ejecución del árbol durante el juego, para aplicar la cadena de decisiones durante el desarrollo de la partida.

Las ventajas de los árboles de decisión son las siguientes:

- Permiten estructurar la toma de una gran cantidad de decisiones.

- Pueden ser más eficientes que las secuencias de condiciones de los sistemas de reglas, ya que en cada ejecución solo se evalúa una línea de condiciones, no todas.

- Controlando la profundidad del árbol se puede controlar el tiempo de ejecución del método de una manera sencilla.

- Es posible seguir la secuencia de decisiones que lleva al árbol a tomar una decisión final.

3.4.2. Árboles de juego. Algoritmo Minimax

Los **árboles de juego** (árboles con todas las **jugadas posibles** de un juego) son muy populares para los juegos en los que participan dos jugadores, sobre todo en los llamados juegos de mesa. Los elementos de un árbol de juegos son estos:

- Cada nodo representa un estado posible del juego. El nodo raíz es el inicio del juego. Los nodos hoja son los diferentes finales del juego.

- Cada enlace representa una jugada de un jugador, que lógicamente hace que el juego cambie de un estado (nodo) a otro.

- Por tanto, todos los enlaces que están al mismo nivel (misma distancia al nodo raíz) son las jugadas posibles en un momento dado, y todos los nodos del mismo nivel son los estados posibles del juego tras la jugada

correspondiente. Por ejemplo, los nodos de nivel 3 (suponiendo que el raíz es nivel 0) son los estados posibles tras tres jugadas.

- Al ser juegos de dos jugadores, cada nivel de enlaces corresponde a las jugadas de cada uno de los dos jugadores alternativamente (la primera fila de enlaces son las jugadas posibles del jugador 1; la segunda, las del jugador 2; la tercera, las del jugador 1 de nuevo, etc.)

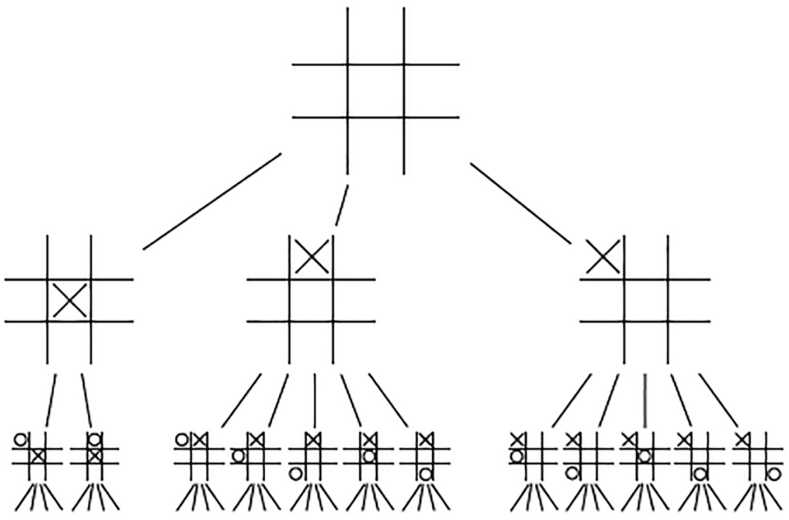
- Asimismo, los niveles de nodos corresponden al estado del juego tras la jugada de cada jugador alternativamente (nivel 1 estado tras jugada del jugador 1, etc.).

Normalmente la generación de árboles de juego está limitada por los recursos del sistema, por lo que, en general, no podremos llegar hasta la situación final del juego (ganar, perder o empatar, por ejemplo), salvo en juegos muy sencillos.

El tres en raya es un ejemplo perfecto para utilizar un árbol de juego. Partiendo del tablero vacío (que será el nodo raíz), y suponiendo que empiece el jugador con las fichas marcadas como «X», se genera un nodo hijo para cada jugada posible del jugador, en este caso colocar la «X» en cada una de las 9 casillas del tablero (recordemos que es la primera jugada).

A su vez, en respuesta a cada una de las 9 jugadas posibles el contrincante puede colocar su ficha «O» en alguna de las 8 casillas que quedarán libres, por lo que cada nodo generará a su vez 8 nodos hijo con cada respuesta posible del jugador «O»; cada uno de estos nodos, a su vez, permitirá 7 posiciones posibles al jugador «X», y así sucesivamente hasta finalizar el juego (por ocupar todas las casillas o porque uno de los dos jugadores haga tres en raya) en los nodos hoja.

Ejemplo de árbol de juego para el tres en raya.



Tres en raya

Como se ve, el número de nodos crece rápidamente; en el caso del tres en raya, que es un juego muy sencillo, las jugadas posibles son del orden de 9! = 362880; en realidad, menos, ya que algunas ramas acaban antes de completar todas las posibilidades, cuando se consigue hacer tres en raya.

En otros juegos más complejos el número de niveles máximo del árbol normalmente es bastante pequeño. Por ejemplo, el supercomputador Deep Blue que ganó a Gary Kasparov solo era capaz de introducir en el árbol los doce siguientes movimientos para después aplicar una heurística sobre ellos.

El objetivo de utilizar árboles de juego es que la IA elija la jugada que más posibilidades de ganar le proporcione. Una manera sencilla sería contar cuántas ramas derivadas de cada una de las opciones actuales llevan a la victoria (o qué proporción de ellas: por ejemplo, 100 me dan la victoria y 20 me hacen perder). No obstante, este algoritmo tan sencillo tiene dos limitaciones importantes:

- Requiere que se pueda generar el árbol de juego completo (o que se pueda evaluar si una rama conducirá a la victoria con una cierta probabilidad).

- Supone que el contrincante juega eligiendo al azar, ya que solo mira el número de ramas que dan la victoria, pero para que ese sencillo cálculo de probabilidades funcione es necesario que a su vez el contrincante no muestre ninguna inteligencia.

Por ello, es necesario utilizar algoritmos más avanzados de gestión de los árboles de juego. Uno de los más utilizados es el algoritmo **Minimax**, que intenta buscar la mejor estrategia dentro del conocimiento del árbol, asumiendo que ambos jugadores van a jugar siempre su mejor movimiento, es decir, asumiendo que el contrincante es inteligente. El jugador A (el que usa el algoritmo) ha de intentar conseguir el valor máximo (el que le da el mayor beneficio), mientras que el jugador B (su adversario) jugará para que A obtenga un valor mínimo (menor beneficio). En otras palabras, el funcionamiento de Minimax puede resumirse como elegir el mejor movimiento para uno mismo suponiendo que el contrincante seleccionará el peor para nosotros.

La primera versión formal de Minimax

La primera versión formal de Minimax fue publicada por John Von Neumann (el mismo que definió la arquitectura moderna de los ordenadores, que permitió que fueran programables) en 1928, y asume que se aplica en los llamados juegos de **suma cero**, en los que la pérdida de puntos o equivalente por un jugador supone la misma ganancia por parte del otro jugador, y viceversa. El ajedrez y el póker son ejemplos de juegos de suma cero.

El funcionamiento del algoritmo es bastante simple. En primer lugar, se debe definir una **función de utilidad**, que asigne un valor numérico a un nodo del árbol. Por ejemplo, en juegos como el tres en raya hay tres posibles valores: +1 (ganar), –1 (perder), 0 (empatar). Otros juegos pueden tener un rango mayor de valores (por ejemplo, los puntos conseguidos).

Una vez generado el árbol de juego, se aplica la función de utilidad a cada nodo hoja (que, recordemos, implica el fin de la partida).

Suponiendo un juego de dos jugadores, cada nivel de nodos de juego corresponde alternativamente a cada uno de los jugadores (es como queda el juego tras la jugada de uno de los jugadores). De ahí que, desde el punto de vista del jugador que ejecuta el algoritmo (y se supone que quiere ganar el juego), habrá dos tipos de niveles:

- **Niveles «max»:** niveles que corresponden al jugador, en los que se debe **maximizar la ganancia**, ya que es el jugador el que elige la jugada.

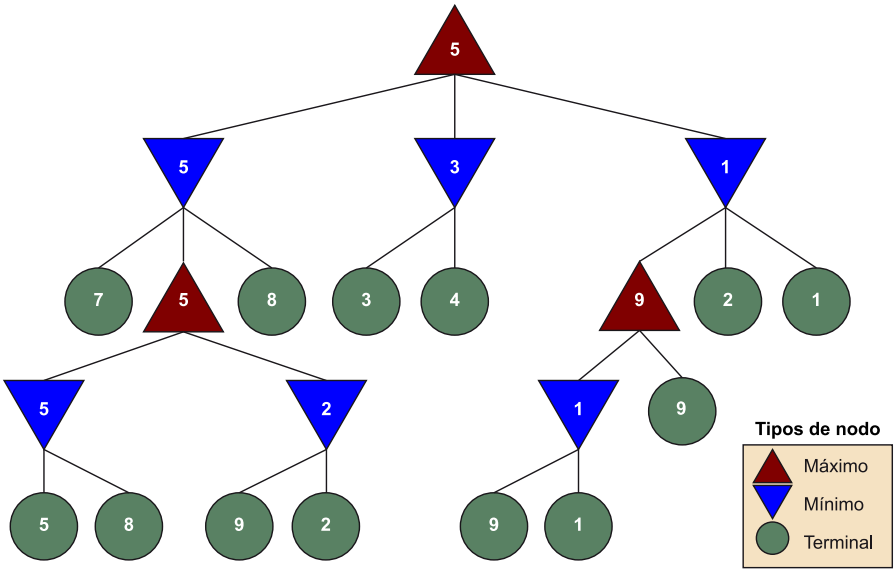
- **Niveles «min»:** niveles que corresponden al adversario, en los que se debe **minimizar la pérdida**, ya que es el adversario el que elige la jugada y lo único que podemos hacer es protegernos para no perder mucho haga lo que haga el adversario.

  En el ajedrez, un nivel «max» bien aprovechado podría implicar comerle una pieza al adversario (digamos una torre), pero si a cambio nos hemos arriesgado mucho, en el siguiente nivel (que será «min») podemos perder más (que nos coma la reina, que vale más).

Resumiendo: se deben maximizar las ganancias y minimizar las pérdidas.

En el ejemplo siguiente puede verse un ejemplo de aplicación de Minimax, en el que tenemos un juego con un resultado posible (función de utilidad) entre 1 y 9.

- En primer lugar, se calculan esos valores (círculos verdes), que son los posibles resultados finales del juego.
- A continuación se va rellenando el árbol hacia arriba, teniendo en cuenta si estamos en un nivel «max» o «min», es decir, si elige el jugador o el adversario.
- Si elige el adversario («min», triángulos azules que apuntan hacia abajo), elegirá la opción con menos puntos de entre los nodos hijos (por ejemplo el 5 de la penúltima fila, izquierda, ya que puede elegir entre hacer una jugada que nos dé 5 puntos y una que nos dé 8, y él quiere que consigamos la menor cantidad de puntos posible).
- Si elegimos nosotros («max», triángulos rojos apuntando hacia arriba), entonces elegiremos la jugada que nos dé la máxima puntuación. Por ejemplo, el triángulo de la tercera fila, segundo por la izquierda, que elige 5 ya que puede ir a un estado con 5 puntos y a otro con 2).
- Una vez completado el árbol, se empieza el juego y se van tomando las mejores decisiones según lo visto en el árbol.



Fuente: CC Wikimedia Commons, por ArthaSphinx.

1) Mejoras

El principal problema de los algoritmos de análisis de estos árboles es que son computacionalmente muy costosos. En un juego real normalmente no se evalúan todas las soluciones porque en algunos casos podemos ver desde el principio que nos van a llevar a un fracaso. En este caso, se dice que «podamos» la rama.

Una de las técnicas más utilizadas es la **poda alfa-beta**, que consiste en una simple modificación del Minimax para descartar aquellas ramas que no pueden mejorar el resultado actual.

Siguiendo con el ejemplo anterior, en el nivel 0 (raíz) estamos buscando el máximo. Analizamos las ramas desde la izquierda y vemos que en la primera obtenemos un 5. En la segunda rama obtenemos un 3. No es necesario analizar en profundidad la tercera rama (la del 1), ya que aunque tiene una rama que nos puede dar 9, nunca conseguiremos ese 9 porque es un nivel «min» y el adversario nos dará el 1; así que la rama del 9 y sus

descendientes se pueden eliminar de la exploración porque el adversario nunca nos las dará, pues tiene la alternativa del 1, pero tenemos algo mejor en otro nodo «max» (el 5).

3.4.3. Árboles de comportamiento

Las máquinas de estados finitos vistas anteriormente permiten modelar el comportamiento de un agente pero tienen algunos inconvenientes:

- Si el comportamiento que se desea modelar es complejo, diseñar una máquina de estados finitos puede convertirse en una tarea muy compleja y proclive a errores.

- Por situaciones inesperadas o fallos de diseño suele ocurrir que los agentes controlados por máquinas de estados finitos se queden atascados en un estado y sean incapaces de salir.

Como siempre que un sistema se vuelve muy complicado por tener un número de componentes demasiado elevado como para ser manejable, hay que organizar y estructurar los componentes para conseguir que funcione como esperamos. Los **árboles de comportamiento** (AC) son una alternativa a las máquinas de estados finitos que permiten crear modelos de comportamiento más complejos. Se empezaron a utilizar en juegos como Halo 2 y Bioshock, juegos que destacaban por la IA de sus NPC.

Los AC tienen los siguientes elementos:

1) Los nodos hoja reciben el nombre de **ejecutores**, y representan acciones que el agente puede llevar a cabo (disparar, andar, abrir una puerta, acelerar…). Cuando reciben la orden para ejecutarse pueden devolver uno de estos tres resultados posibles:

- **Ejecutando**: la acción correspondiente se está ejecutando pero no ha terminado. Por ejemplo, el agente está caminando.

- **Éxito**: la acción correspondiente se ha ejecutado y ha terminado con éxito. Por ejemplo, el agente estaba caminando y ha llegado a su destino.

- **Fallo**: la acción correspondiente no se ha podido iniciar o al intentarlo se ha fallado. Por ejemplo, no ha podido abrir la puerta.

2) Unos nodos **controladores**, que siempre tienen hijos (de tipo ejecutor o controlador). Su función es organizar la ejecución de los nodos ejecutores según diferentes tipos de comportamiento. Básicamente representan una tarea más o menos compleja que debe llevar a cabo el agente. Cuando reciben la orden para ejecutarse proceden a reenviar la orden de ejecución a sus hijos según alguno de los comportamientos siguientes:

- **Secuencia**: el nodo controlador envía la orden de ejecución a sus nodos hijos de izquierda a derecha para realizar una operación que requiere seguir una serie de pasos. Cuando un hijo devuelve «éxito», el controlador pasa al siguiente; mientras el hijo devuelva «ejecutando», permanece en él; si un hijo falla, el controlador falla.

- **Selector**: el nodo controlador envía la orden de ejecución a sus nodos hijos de izquierda a derecha. Cuando un hijo devuelve «éxito», el controlador devuelve «éxito» y termina. Si el hijo devuelve «ejecutando», el controlador devuelve «ejecutando». Si un hijo falla, el controlador pasa al siguiente; si todos fallan, devuelve «fallo».

3) Un nodo **raíz**. Su función es, cada cierto tiempo, enviar una señal (*tick*) a su nodo hijo, que es una orden para que se ejecute. El nodo raíz solo tiene un hijo, de tipo controlador (ver a continuación).

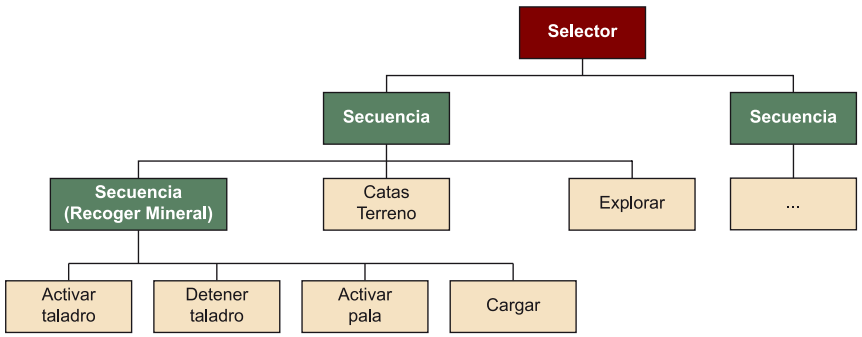
AND y OR y otro tipo de controladores

Podemos ver los controladores de secuencia como «AND» para tareas en las que es necesario seguir todos los pasos (recoger troncos, mover al carpintero a la posición deseada, construir la casa). Los controladores selectores, por su parte, son una «OR» en la que cualquiera de las acciones es suficiente para resolver la tarea (abrir la puerta con la manecilla, con una llave o tirarla abajo). Hay otros tipos de controladores, como los inversores, repetidores, selectores aleatorios, etc.

Por tanto, una IA controlada por AC debe generar un evento cada cierto tiempo (poco, inferior al segundo) para que se vayan ejecutando las tareas. La propia estructura del AC permite modelar un comportamiento mediante un controlador general: ¿la IA debe seguir unos pasos o simplemente realizar una tarea que elegirá entre varias? A continuación, en otro nivel de detalle, se definen las tareas fundamentales que deben llevarse a cabo; esas tareas se pueden dividir en otras tareas más concretas, y así hasta alcanzar acciones del juego, que serán los nodos de ejecución.

Un robot que recorre un planeta en busca de minerales valiosos tendrá al principio un nodo selector para elegir entre dos tareas: recoger minerales y mantenerse en funcionamiento. La tarea principal es la de recoger minerales, así que será la primera y siempre que sea posible se ejecutará esa. A su vez, la tarea selectiva de recoger minerales se compondrá de varias subtareas: tomar mineral, hacer catas en el terreno, explorar. Notad que el orden es el contrario al esperado porque se especifica la prioridad de las tareas: si el robot tiene minerales identificados a su alcance, los debe recoger. Si no lo sabe, debe catar el terreno a ver si hay algo que valga la pena. Y si no, seguir explorando. La tarea de tomar el mineral es una tarea representada por un controlador secuencial: activar taladro, detener taladro, activar pala, cargar en contenedor. Así sucesivamente con el resto de las tareas.

Ejemplo de árbol de comportamiento de un robot minero. Los nodos granate son los selectores; los verdes, los de secuencia; y los rosa, los de acciones.



Propiedades de los AC:

- Como un nodo controlador puede tener a otros nodos controladores como hijos, la ejecución del AC puede ser recursiva.

- Para su correcta ejecución, los nodos del AC deben recordar su estado entre una invocación (*tick*) y el siguiente; si un agente está caminando debe recordarlo.

- Permiten representar comportamientos muy complejos con gran nivel de detalle pero con una visión global que evita atascos en estados pobremente diseñados.

- Resulta sencillo tomar una parte de un AC para reutilizarla en otro; son modulares.

Behavior Designer

El *asset* de Unity llamado Behavior Designer permite diseñar y utilizar árboles de comportamiento fácilmente.

Las principales limitaciones de los AC son estas:

- Si los árboles son muy extensos, puede resultar costoso computacionalmente ejecutarlos.

- La calidad final del comportamiento depende de la calidad de las condiciones programadas en cada nodo, lo que en muchos casos sigue siendo un problema sin resolver.

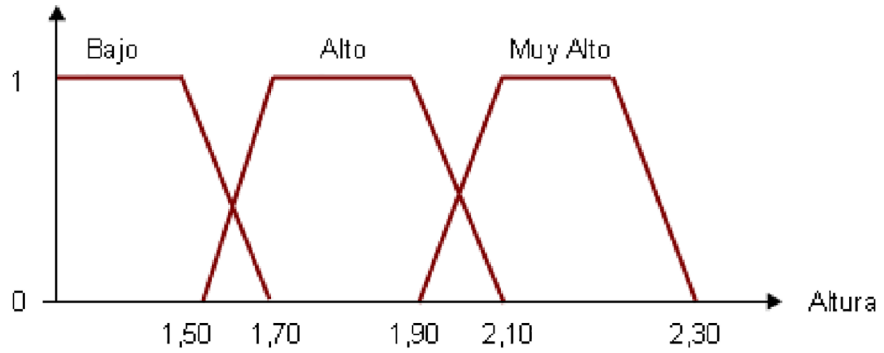
3.5. Lógica difusa

Una proposición en la lógica clásica solo admite dos valores posibles: verdadero o falso. No obstante, en la vida real las observaciones no son tan claras como para poder afirmar que una proposición es verdadera o falsa, sino que puede haber diferentes puntos de vista que sean relativos al observador.

La lógica difusa es una alternativa que nos permite cuantificar esta incertidumbre, nos da la opción de asignar cierta probabilidad a cada uno de los posibles valores y, por lo tanto, añadir la relatividad del observador. Según su creador, la idea original que se halla detrás de la lógica difusa es la de imitar el funcionamiento del razonamiento humano, el cual normalmente no trabaja con valores exactos sino con valores relativos.

Imaginad que queremos jugar un partido de baloncesto y hemos de colocar a cada jugador en su posición. Para esta tarea, clasificamos a la gente como alta, baja, lenta o rápida, pero nunca utilizamos medidas exactas, como que mide 1,80 metros o que corre los 100 metros en 15 segundos. La lógica difusa nos permite introducir estos conceptos, más relativos a los que estamos acostumbrados dentro de un sistema de IA.

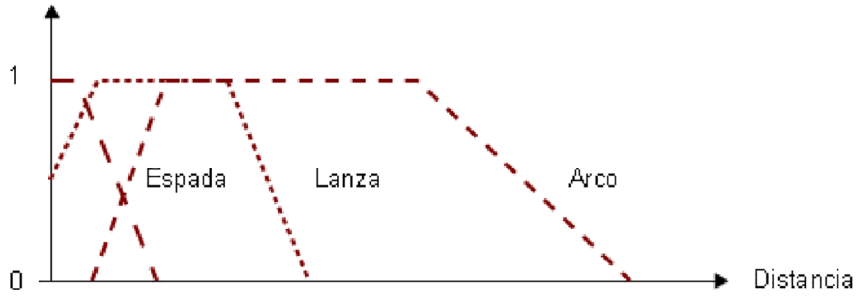
Un valor en la lógica difusa puede encontrarse entre verdadero y falso, pudiendo tener valores de mucho, un poco... Esta conversión de un valor cuantitativo a un valor cualitativo se realiza mediante lo que se conoce como un subconjunto difuso (*fuzzy set*), y representa la probabilidad de que alguien asigne un cierto valor cualitativo a uno cuantitativo.



En la figura anterior tenemos una forma para convertir los valores. Para alguien que mida menos de 1,50, la probabilidad de que lo llamemos bajo es 1, pero si en cambio mide 1,60, existirá un cierto porcentaje de gente que lo pueda clasificar como alto y otro como bajo.

La lógica difusa no es una técnica como las que hemos presentado hasta el momento, que nos permiten tomar una decisión a partir de la observación del sistema; más bien la podemos considerar como un sistema complementario a los vistos, que provoca que las decisiones que se tomen tengan una vertiente más relativista y por tanto más «humana».

1) Si combinamos la lógica difusa en un sistema de reglas *if...then*, podemos hacer que las reglas parezcan más realistas. Imaginemos un caso en el que hemos de elegir el arma con la que atacar (espada, lanza o arco) dependiendo de la distancia del objetivo. En las reglas básicas podemos definir tres rangos donde vamos a decidir el arma que queremos llevar (por ejemplo, para menos de cinco metros usaremos una espada, entre cinco y veinte metros, una lanza, y a partir de aquí, el arco). En cambio, si queremos que sea más realista, podemos introducir un subconjunto difuso como el siguiente y después elegir el arma según la probabilidad de cada una.



2) Si combinamos la lógica difusa con las máquinas de estado, obtenemos lo que se conoce en IA como máquinas de estado difusas. Este tipo de máquinas permiten que el sistema se encuentre en más de un estado a la vez, es decir, tenemos una serie de estados activos con una cierta probabilidad y en cada momento decidimos cuál es el que nos interesa utilizar.

Método IA de utilidad

La lógica difusa ha ganado actualidad a través del método llamado **IA de utilidad** (Utility AI), en el que se combinan varias medidas difusas con las diferentes opciones que tiene un agente y lo útiles que le resultan en un momento dado para así elegir la opción que le proporcione más utilidad. Este método tiene la ventaja de conseguir un comportamiento flexible y resulta fácil de diseñar, ya que no es necesario prever todas las situaciones (estados) posibles, sino que siempre da una respuesta.

Utility AI está disponible para Unity mediante el *asset* Apex Utility AI (de pago).

3.6. Mapas de influencia

Los mapas de influencia son una representación discreta del conocimiento que tiene el jugador acerca del mundo. Son un elemento necesario para procesos de decisión estratégicos y tácticos.

Popularidad de los mapas de influencia

Son muy populares en todo tipo de juegos de estrategia, desde juegos de tablero a juegos de estrategia en tiempo real, ya que permiten estudiar la situación e influencia de las piezas del enemigo.

Los mapas de influencia se pueden utilizar para diferentes objetivos:

- detectar aquellas regiones que nos interesan, aquellas otras regiones que debemos evitar y el lugar donde se encuentra la frontera entre estas regiones,
- buscar puntos débiles del adversario, analizando la localización de sus defensas,
- guiar el movimiento de un elemento,
- cuantificar si nuestro oponente tiene más fuerzas que nosotros. Si los valores negativos indican la fuerza del contrario y los positivos las nuestras, la suma de todos los valores nos indicará quién es más fuerte.

Para crear un mapa de influencia, lo primero que hemos de hacer es segmentar el mapa del mundo en regiones discretas. Podemos utilizar tanto los *grids* como las mallas poligonales vistas en el módulo anterior y no tienen por qué coincidir con la geografía real del mundo.

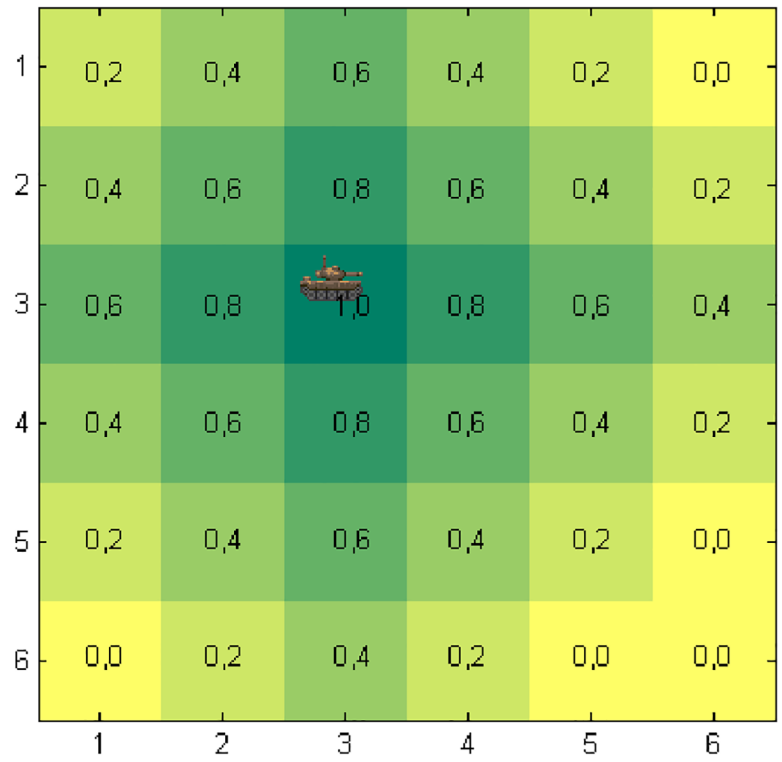
Después debemos asignar un valor numérico a cada una de las zonas del mapa de influencia que tiene relación con la partida que se esté desarrollando (fuerza estratégica de la posición, número de unidades enemigas en la zona, contenido de esa zona en recursos, etc.).

El cálculo del valor de esta posición es un elemento clave para posteriormente elegir las decisiones de manera correcta, así que será necesario el diseño de una buena ecuación que tenga en cuenta dos grupos de elementos:

- Todos los elementos que se encuentren en la zona y que influyan en su valor puntual.

- La influencia de los elementos que se encuentren en las zonas colindantes. Cuanto más lejos se encuentren, menor será su influencia.

Ejemplo de un mapa de influencia en un *grid* cuadrado



Una manera bastante utilizada para calcular mapas de influencia es calcular un mapa por cada zona que contenga algún elemento interesante (por ejemplo, una unidad o un edificio) y añadir la influencia de este elemento en las demás zonas. Posteriormente, agregamos todos estos mapas para obtener el mapa final.

Los mapas de influencia son elementos dinámicos. Al principio del juego podemos calcular el valor inicial de las zonas, pero cada vez que se dé algún movimiento deberemos recalcular un nuevo mapa con la nueva situación. Cabe tener en cuenta que no es necesario recalcular todas las casillas, solamente aquellas afectadas por el cambio, con lo que el proceso es menos costoso de lo que parece.

Podemos definir un comportamiento sencillo a modo de ejemplo. Supongamos que el valor de cada celda del mapa de influencia es un valor entre –1 y 1 que define la «fuerza» de un oponente. Si es un valor positivo, indica que tenemos más fuerza que nuestro oponente en ese punto, mientras que en caso contrario, el oponente tendrá más que nosotros.

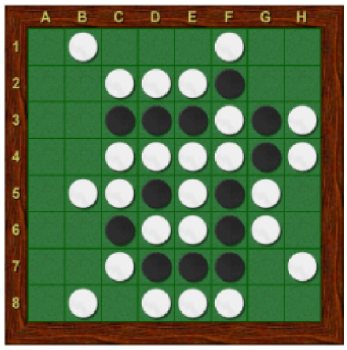
Por otro lado, cada elemento (soldado, tanque, etc.) tiene un valor de fuerza entre –1 y 1, pero su significado cambia un poco. Si el valor es negativo, indica que tiene problemas y quiere huir del enemigo (podríamos definir otro comportamiento, como ir en busca de ayuda, alimentos, etc.). Si el valor es positivo, nos señalará que está en condiciones de atacar.

El mapa de influencia lo hemos definido a base de cuadrados. Así, el elemento solo puede moverse a una celda de sus ocho celdas adyacentes. De estas ocho, elegirá aquella que tenga un valor menor que el suyo si está en condiciones de atacar, o un número mayor si tiene problemas.

Un ejemplo más simple del uso de mapas de influencia lo encontramos en la planificación de estrategias en juegos de tablero.

Por ejemplo, en el juego del Othello (también conocido como Reversi), que consiste en colocar las fichas en el tablero para poder conseguir capturar las del contrincante, a cada posición del tablero se le puede asignar un peso que nos indica qué beneficio obtenemos si ponemos una pieza en cada casilla.

Ejemplo de una partida, junto con su mapa de influencia correspondiente



34		4	6	6		-1	34
-2	-9					-8	-1
6	2						
3	1						
3							5
6	2						4
-2	-9					-7	
34		8				-5	34

Esta información es crucial para poder desarrollar una buena estrategia, para intentar colocar las piezas en las posiciones más favorables y evitar aquellas posiciones menos ventajosas.

4. Aprendizaje

Los métodos vistos hasta ahora ofrecen soluciones para un amplio abanico de situaciones de juego y son muy utilizadas por su efectividad a la hora de resolver los problemas para los que han sido diseñadas. Pero precisamente ahí radica también su principal limitación: son técnicas que se deben diseñar y ajustar por los desarrolladores del juego, a menudo mediante un costoso proceso de ensayo y error que no siempre da buenos resultados. Y además, su resultado, al estar prefijado en el diseño del juego, resulta bastante predecible. Imaginemos por ejemplo un árbol de decisión en el que la cadena de decisiones se repite miles de veces y, de alguna manera, los jugadores pueden acabar reconociéndola y aprendiéndose las respuestas de los agentes de la IA. Una solución simple para evitar esa respuesta predecible consiste en introducir elementos aleatorios, pero por su propia naturaleza aleatoria eso provoca que los agentes hagan, de vez en cuando, cosas sin mucho sentido. El jugador percibirá una cierta inteligencia en los agentes, pero tampoco mucha.

Lo ideal sería que la IA reaccionara de manera flexible ante las diferentes situaciones del juego e incluso innovara en función del estilo de juego del jugador, para que este realmente se sorprenda al ver a la IA aprendiendo de cómo juega y encontrándole puntos débiles.

Pero para eso es necesario que la IA aprenda durante la ejecución del juego, no se pueden prediseñar todas las respuestas posibles. Para conseguir dotar a los juegos de esa inteligencia avanzada, en este apartado veremos métodos que aprenden a partir de las diferentes situaciones del juego y son capaces de corregir su comportamiento para mejorar sus resultados y, lo que es más importante, seguir entreteniendo al jugador.

4.1. Estrategias de aprendizaje en juegos

En general, los métodos de IA que aprenden (también conocidos como métodos de **aprendizaje automático** o *machine learning*) distinguen dos fases: en primer lugar, una fase de **entrenamiento**, en la que el método recibe un conjunto de datos (a menudo junto con la respuesta correcta) y lo utiliza para aprender ajustando sus parámetros internos y creando un modelo del problema al que se enfrenta. En nuestro caso, los datos serán las diferentes situaciones de juego, y el problema que deberemos resolver consiste en decidir las acciones correctas para que el agente consiga la máxima puntuación en el juego o simplemente ofrezca un reto más interesante a los jugadores. Normalmente son desarrolladores los que juegan contra la IA para entrenarla.

Entrenamiento de la IA

Dependiendo de la organización de la empresa que desarrolla el videojuego, esta labor la pueden llevar a cabo los propios desarrolladores de la IA, los *internal testers* (prueban versiones intermedias del juego) o los *quality assurance testers* (que prueban el juego ya acabado). Incluso muchas veces se abre al público una versión *beta* del juego para aprovechar que jugadores reales usen el juego y así usar ese tiempo de juego para acabar de ajustar la IA.

Una vez entrenado el método, se pasa a la fase de **prueba**, en la que se utiliza el modelo aprendido para reaccionar de la mejor forma posible a cada situación de juego que se presente. En el caso de los videojuegos, la prueba es el uso de los métodos en el juego definitivo contra jugadores reales (no contra desarrolladores que se dedican a entrenar a la IA).

Un agente para un juego de tenis puede entrenar jugando contra personas o contra otros agentes e ir aprendiendo de sus éxitos y sus errores para encontrar estrategias óptimas en cada situación (bola baja, bola alta, bola en la otra parte de la pista...). Esa sería la fase de entrenamiento. La fase de prueba del método sería la puesta en uso del agente en el juego real, utilizando el modelo aprendido previamente para tomar las decisiones necesarias en cada momento.

4.1.1. Momentos del aprendizaje

La fase de entrenamiento de la mayoría de los métodos de IA es relativamente costosa en términos de consumo de memoria y tiempo de ejecución, por lo que en la mayoría de los casos no resulta práctico llevarla a cabo durante la ejecución del juego (aparte de que un jugador quiere un juego que ya sepa jugar, no que todavía tenga que aprender y empiece jugando fatal). De ahí que la fase de entrenamiento se realice durante el desarrollo y ajuste del juego; en ese caso se habla de aprendizaje ***offline***, pues no se hace en tiempo de ejecución del juego final.

Por otra parte, algunos métodos más ligeros pueden aprender durante el juego en sí; en ese caso hablaremos de aprendizaje ***online***.

Por último, existe la posibilidad de hacer un entrenamiento previo en modo *offline* pero dejar los ajustes finales (por ejemplo, para reaccionar al estilo de juego de cada jugador) para hacer algo de entrenamiento *online*.

4.1.2. Modos del aprendizaje

Otro aspecto fundamental de los métodos de aprendizaje automático es cómo se realiza el aprendizaje, es decir, con qué información cuenta el método para ir aprendiendo. Según este criterio, se distinguen cuatro modos fundamentales:

- **Aprendizaje no supervisado:** el método recibe una serie de atributos o características de los ejemplos que debe procesar. Tareas típicas de este modo son el agrupamiento (*clustering*) y la reducción de la dimensionalidad. Por ejemplo, podemos recibir los datos de uso de un juego (tiempo medio

jugado, modos de juego utilizados, puntuación conseguida) y en función de eso formar grupos de perfiles de usuario típicos.

- **Aprendizaje supervisado:** además de los atributos de los ejemplos de entrenamiento, se recibe también una **etiqueta** que indica de qué clase o tipo es ese ejemplo. Tareas típicas de este modo son la clasificación y la detección de anomalías. Por ejemplo, podemos recibir las características de un avión y su clase para que un piloto automático de un simulador de vuelo aprenda si se trata de un avión de combate, de pasajeros, etc.

- **Aprendizaje semisupervisado:** algunos ejemplos de entrenamiento están etiquetados y otros no; se empieza aprendiendo de los etiquetados y luego se supone una etiqueta a los no etiquetados en función de su similitud con los etiquetados. Resulta útil porque a menudo es costoso etiquetar ejemplos (debe hacerlo un humano), y así se aprovecha un conjunto etiquetado pequeño para entrenar con un conjunto de ejemplos mucho mayor. Por ejemplo, si queremos que un sistema de conducción automática detecte si hay peatones en una imagen, podemos etiquetar algunas imágenes (cientos o miles) y además proporcionarle millones de imágenes sin etiquetar para mejorar el entrenamiento.

- **Aprendizaje por refuerzo (*reinforcement learning*):** en muchos problemas no resulta natural hablar de ejemplos y etiquetas, ya que se trata de entrenar a un sistema para que interactúe con un entorno (real o virtual) y vaya mejorando con la práctica. Dado el especial interés de este tipo de aprendizaje para los videojuegos, le dedicaremos un apartado completo.

Aprendizaje por refuerzo

El aprendizaje por refuerzo aplicado a la IA en videojuegos ha adquirido una gran y merecida fama por sus recientes éxitos, como el sistema DQN (de Deep Q-Network, ahora se explicará), que es capaz de jugar a juegos de la consola Atari y ganar a jugadores humanos expertos, o el sistema AlphaGo, que ha ganado al campeón mundial de Go (un juego de tablero de origen chino que es mucho más complejo en número de jugadas posibles que el ajedrez). En ambos sistemas se ha demostrado el potencial de esta técnica.

También se está aplicando con éxito a simuladores de diferentes tipos, sistemas de conducción automática o robótica; en todos estos ámbitos, el agente puede realizar acciones y aprender de sus consecuencias.

Observación

Hay que destacar que el sistema DQN aprende a jugar sin ningún conocimiento previo del juego, solo recibe la imagen de vídeo (lo mismo que vería un jugador humano) y va probando diferentes acciones para ir aprendiendo poco a poco con qué acciones consigue una mayor puntuación. Además, no se le ha programado ningún tipo de conocimiento sobre cada juego en particular, y a pesar de ello ha sido capaz de dominar los juegos e incluso encontrar «trucos», como en el juego Breakout, en el que aprende a hacer un agujero en la fila de bloques para que la bola suba a la parte superior y así destruya muchos bloques rápidamente.

Captura de pantalla del juego Breakout de la consola Atari 2600

Fuente: CC McLoaf para Wikimedia Commons a partir del juego Breakout propiedad de Atari Inc.

Veamos en qué consiste el *reinforcement learning* (RL para abreviar). Tenemos un problema o situación para la que queremos entrenar a un agente de manera que aprenda a realizar las acciones adecuadas en cada momento. Para que el agente sepa que lo está haciendo bien, debe recibir una «recompensa» cuando su acción acierta en el sentido del comportamiento que se desea: jugar bien a un videojuego, llevar bien un coche por una autopista, mover un robot entre dos posiciones esquivando obstáculos, etc. Más detalladamente, los elementos que componen el sistema RL son estos:

- Un conjunto S de **estados** posibles del problema (juego, etc.). En el ejemplo del Breakout, los estados posibles son todas las posibles posiciones de la barra que mueve el jugador, por todas las posibles posiciones y velocidades de la bola, por todas las combinaciones posibles de ladrillos enteros/destruidos. Como se ve, hasta para un juego tan sencillo el número de posibles estados es inmenso.

- Un conjunto A de posibles **acciones**. En el juego Breakout son solo tres: moverse a izquierda, derecha o quedarse quieto. En un juego de estrategia, por ejemplo, serán muchísimas más.

- Una función de **recompensa** R, que devuelve un número real en recompensa por la acción tomada por el agente en un estado determinado, $R: S \times A \rightarrow R$. Es muy importante notar que la recompensa depende de la acción tomada y del estado en el que se encontraba el juego, ya que a veces moverse a la izquierda será bueno pero otras veces no.

- En principio, el agente debe aprenderse una **tabla de recompensas** $Q = S \times A$ para así saber qué hacer en cada posible situación del juego. Lógicamente esto es inviable en casi cualquier juego (que sea más complicado que el tres en raya), de ahí que se hayan propuesto diferentes métodos para aprender una aproximación de Q aceptablemente buena y que sea práctica. Esos métodos reciben el nombre de *Q-Learning*.

Observación

La Q que vemos es la del método DQN mencionado anteriormente; lo de *deep* y *network* se refiere a que el método no se aprende la matriz Q tal cual (sería imposible), sino que aprende a aproximarla mediante una red neuronal (de ahí el *network*) profunda (y de aquí el *deep*). En el apartado 5 se explicará qué son estas dos técnicas y cómo pueden usarse con RL.

El funcionamiento del RL con todos esos elementos se produce en momentos de tiempo discreto, {...,t-2, t-1, t, t+1, t+2,...}, por ejemplo en cada iteración del bucle de juego (cada *frame*). Cuando el agente actúa en el tiempo t, ocurre lo siguiente:

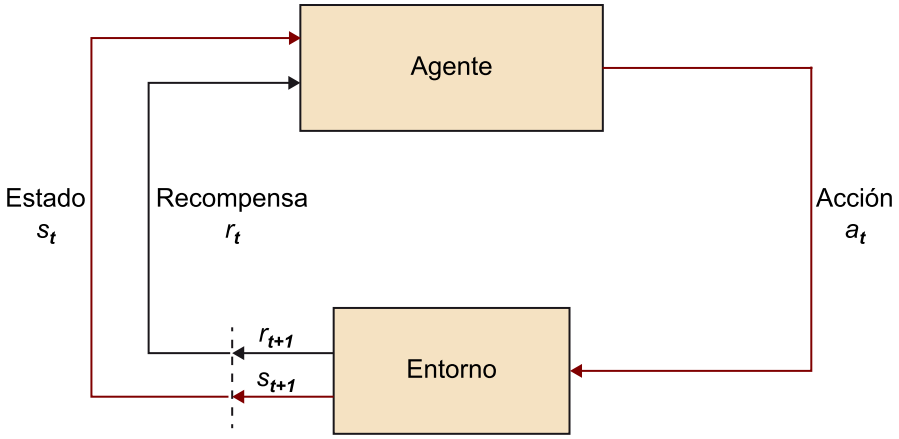
1) Recibe la recompensa de su acción anterior, R_t.

2) Debe identificar cuál es el estado actual S_t a partir de la información que le proporcione el juego.

3) En función del estado actual, el agente debe decidir cuál es la acción A_t que le proporcionará una mayor recompensa en la próxima iteración, R_{t+1}.

4) Al aplicar la acción, el entorno cambia y se inicia una nueva iteración (vuelta al paso 1). La recompensa recibida se asocia al uso de A_t en S_t, es decir, si la recompensa es buena se asociará esa respuesta a ese estado, y si es mala se dejará de lado esa acción como respuesta a ese estado.

Diagrama de control de un sistema RL



Fuente: Adaptado de https://deeplearning4j.org/reinforcementlearning.

Una forma sencilla de programar un agente para que tome una decisión es mediante el algoritmo $\epsilon - greedy$ (voraz – épsilon), que consiste en elegir la acción que dé la mayor recompensa en la siguiente iteración con probabilidad $1-\epsilon$, o una acción al azar en otro caso. En el fondo es una manera simple de tratar con un problema de RL: las recompensas no siempre son inmediatas, sino que pueden llegar al cabo de muchas iteraciones.

> En el juego del Breakout, la pelota está bajando, el agente mueve la barra hacia ella, le da, la pelota rebota y al cabo de unos segundos rompe un bloque y le da 10 puntos al agente. Pero en medio han pasado, por ejemplo, 3 segundos, que pueden ser 90 *frames*, es decir, quizá 90 iteraciones. ¿Cómo hace el método RL para relacionar la acción anterior con la recompensa futura?

Una estrategia más avanzada para afrontar ese problema es la consideración de las recompensas futuras en el algoritmo de aprendizaje. Si en el instante t el juego se encuentra en el estado S_t y se ha realizado la acción A_t, se deben asociar las recompensas recibidas en las siguientes iteraciones $\sum_{i=t}^{t+N} R_i$, no solo la siguiente recompensa; de esa manera se premia una acción aunque la recompensa llegue tarde. En esta estrategia se supone que la recompensa se alcanza tras N iteraciones.

Si no es así y no hay modo de conocer cuándo se alcanzará la recompensa, se suele aplicar la siguiente fórmula, que incluye un descuento $0 < \gamma < 1$ que va reduciendo el peso de la recompensa a medida que se aleja en el tiempo: $\sum_{i=t}^{\infty} \gamma^i R_i$.

En cualquiera de los dos casos, aunque se complica el desarrollo de aprendizaje, se tienen en cuenta recompensas futuras, lo que en la mayoría de las situaciones mejora los resultados del algoritmo.

Bootstrapping

Anteriormente hemos visto que el entrenamiento de un agente de IA para un juego suele hacerse en su mayor parte *offline*, por una parte para no ocupar recursos computacionales cuando se ejecuta el juego, y por otra para no entregar un producto incompleto a los clientes. Pero ¿cómo se lleva a cabo ese entrenamiento cuando se quiere entrenar una IA que juegue contra humanos? ¿Se contrata a equipos de jugadores profesionales para que jueguen contra la IA y le enseñen a jugar? En ocasiones se hace eso precisamente, pero en muchos casos las horas de entrenamiento que necesita la IA son demasiadas y no resulta viable utilizar ese recurso, al menos no exclusivamente.

En este ámbito, la solución se suele denominar *bootstrapping* y consiste en algo muy sencillo: que una IA juegue contra otra. Las dos pueden usar el mismo algoritmo (se recomienda que con ajustes diferentes) o algoritmos distintos. Incluso es interesante poner a competir a la IA con varios algoritmos distintos para que vaya aprendiendo de cada uno de ellos. Y a medida que los dos agentes van mejorando, tienen que esforzarse más para ganar al otro y aún deben mejorar más.

Bootstrapping

En el ámbito más general del aprendizaje automático, *bootstrapping* es una técnica de aprendizaje semisupervisado que consiste en atribuir etiquetas a ejemplos no etiquetados y usarlos como entrenamiento, o incluso a «inventarse» ejemplos parecidos a los existentes para ampliar el conjunto de entrenamiento y mejorar los resultados. En el ámbito del RL, se utiliza el mismo término para la estrategia expuesta aquí por la similitud de usar un agente de IA para generar ejemplos de entrenamiento para otro agente de IA.

Por ejemplo, utilizando esta estrategia se ha conseguido una IA que juega al ajedrez como un experto entrenando solo unas horas con un ordenador personal, a diferencia del famoso Deep Blue que ganó a Kasparov en 1997 y que era un supercomputador que calculaba millones de jugadas posibles. Se puede leer más en https://arxiv.org/pdf/1509.01549v2.pdf.

4.2. Algoritmos de aprendizaje

El objetivo principal de este apartado es dar una visión de los algoritmos de aprendizaje automático y su posible aplicación. Antes de ver los diferentes algoritmos vamos a ver qué problemas abordan, su nomenclatura y cómo los podemos aplicar a los videojuegos.

Todos los algoritmos de aprendizaje se basan en la extracción de conocimiento a partir de conjuntos de datos. El problema más importante en aprendizaje es la clasificación. En clasificación partimos de un conjunto de datos etiquetados

que representan objetos. La clasificación consiste en etiquetar objetos nuevos con las clases pertinentes partiendo del conocimiento extraído de los datos previos.

A modo de ejemplo, imaginemos que disponemos de una colección de correos electrónicos con la información añadida de si son correos *spam* o no. La tarea de la clasificación corresponde a crear un modelo a partir de estos datos y poder aplicar este modelo para construir un clasificador de *spam* con el objetivo de poder etiquetar automáticamente si un conjunto nuevo de correos son *spam* o no.

A estos objetos (en el caso anterior correos) se los suele llamar ejemplos y tienen asociada una etiqueta a la que llamamos clase (en el caso anterior si corresponde a *spam* o no). El vector que representa a cada objeto suele recibir el nombre de conjunto de atributos, atributos a secas o entradas del sistema[1]. La clase puede recibir también el nombre de salida[2]. Si agrupamos todos los vectores de atributos en forma de matriz, recibe el nombre de entradas o *X*; y el vector de todas las clases el de salida o vector *Y*.

(1)En inglés *inputs*.

(2)En inglés *output*.

El problema más importante a la hora de construir un clasificador es la representación de la información. Es decir, cómo guardamos la información de los diferentes objetos en forma de atributos y definimos el conjunto de clases. Imaginemos ahora que tenemos a nuestra disposición los informes médicos de un grupo de pacientes de un hospital y queremos desarrollar un clasificador para diagnosticar de manera automática una cierta enfermedad a partir de estos. El primer paso que se nos plantea es definir el conjunto de clases. Estas podrían ser simplemente si el paciente padece o no la enfermedad. Pero también podríamos pensar en construir un conjunto de clases que nos permitan discernir el grado de la enfermedad: si está completamente sano, si la tiene pero en un estado inicial, etc.

El segundo paso será decidir cómo representamos al paciente en forma de atributos. ¿Qué informaciones son relevantes para detectar la enfermedad? La fiebre, la tensión arterial, la edad, si es varón o mujer…

Una vez decidido que queremos guardar la fiebre, esto puede ser si el paciente tiene fiebre o no, o directamente la temperatura corporal. Podríamos alternativamente pensar en guardar la temperatura cada 8 horas durante la última semana.

Cuando encontramos un atributo representado en forma numérica se dice que el atributo es numérico. Otra alternativa es que sea nominal. Un atributo nominal corresponde a uno en el que sus valores pertenecen a un conjunto de categorías. El ejemplo de si un paciente tiene fiebre o no o el color de un objeto serían ejemplos de atributos nominales. Esto es especialmente importante porque hay muchos algoritmos que solo funcionan con atributos nominales

(3)Las técnicas más usuales son la estandarización y el escalado (en inglés *standardization* y *scaling*).

o solo con atributos numéricos. Se puede dar el caso en el que nuestro problema contenga los dos tipos de atributos. Hay técnicas para convertir un tipo de atributo en el otro: como la creación de intervalos para pasar un atributo numérico a nominal o la asignación de un número a cada valor nominal en el caso contrario. Muy a menudo también es útil la normalización de los valores numéricos cuando los valores de los diferentes atributos son de rangos muy dispares[3].

Todos estos temas los tendremos que pensar en el momento de diseñar el problema. Esto es especialmente relevante en el caso de la programación de videojuegos, ya que no existe mucha documentación al respecto y los resultados dependen en gran medida de este diseño.

En un juego de lucha, podríamos decidir implementar un clasificador que guie a un cierto personaje no jugador prediciendo de alguna forma los movimientos del jugador. Esto obligaría al jugador a ir cambiando su sistema de ataque. Para diseñar el clasificador tendremos que diseñar primero el conjunto de clases. Estas podrían ser el movimiento que tiene que hacer el personaje no jugador: un cierto ataque, una cierta defensa, una secuencia tipos de defensa + ataque... pero también podríamos pensar en que prediga qué movimiento hará el jugador y, por ejemplo, después poder elegir aleatoriamente el movimiento que hacer de entre unos cuantos en función de la predicción. Esto haría que fuera menos predictible. Las siguientes decisiones que deberemos tomar tendrán que ver con la representación de las situaciones. Es decir, con la representación de los ejemplos. Podríamos pensar en los últimos movimientos que ha realizado el jugador en una cierta ventana de tiempo. Normalmente discretizamos el tiempo y establecemos una cierta ventana (x unidades de tiempo). Pero también podríamos pensar en hacer lo mismo con los movimientos tanto del jugador como del oponente no jugador. Podríamos añadir información de cuál de los dos ganó el encuentro o pensar en ponderar los ejemplos en función de si ganó o perdió con mucha diferencia. Todo esto genera un grupo numeroso de posibilidades de diseño en el que habrá muchos diseños que no funcionen y otros que sí. Una vez diseñado todo, tendremos que elegir el algoritmo de aprendizaje a utilizar teniendo en cuenta el tipo de atributos que tenemos, cuestiones de eficiencia computacional, etc.

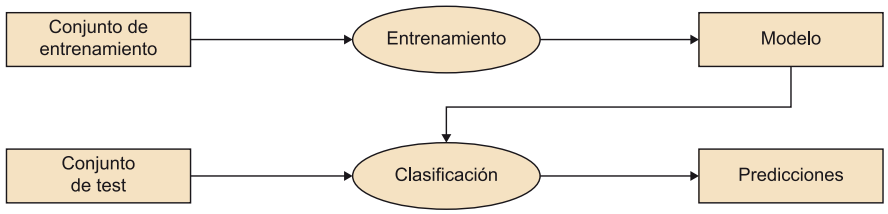
En un juego de rol en el que el jugador controla a una patrulla pequeña de héroes, podríamos diseñar clasificadores a diferentes niveles. Por un lado, podríamos pensar en un clasificador para guiar a los compañeros del héroe que no esté controlando él. En este caso podríamos darle la opción al usuario de definir las características generales de cada uno de ellos. Por otro lado, podríamos diseñar clasificadores para controlar las patrullas enemigas a dos niveles. Podríamos pensar en diseñar un clasificador que guíe la estrategia general de la patrulla. Esta tomaría como entrada los componentes de la patrulla del jugador y los de la suya propia, y establecería la táctica general de la patrulla no jugadora. En este caso tendríamos que definir *a priori* cuáles son estas tácticas y proponerlas como clases. A modo de ejemplo, una de estas tácticas podría

ser concentrar el fuego de los arqueros sobre los magos y evadir en todo momento a los guerreros contrarios; que los magos confundan a los guerreros y lo guerreros ataquen a los arqueros. En un segundo nivel, podríamos definir clasificadores para el control de los personajes individuales. A modo de ejemplo, podríamos tener el clasificador para un arquero. En este caso las clases podrían ser huir, ponerse en actitud defensiva con un escudo y espada, atacar al mago más débil o con menos vida, atacar al enemigo que esté atacando a su compañero más débil… y sus atributos podrían ser la estrategia general que define el clasificador superior, su nivel de vida, el de sus compañeros, el de los miembros de la patrulla enemiga, el nivel de maná o característica que se utilice para ataques especiales… También podríamos pensar en incorporar algo de historia, como lo que se ha explicado anteriormente para el juego de lucha.

En un juego de estrategia o simulador de algún deporte de grupo podemos pensar en una arquitectura de clasificadores a dos niveles tal y como se explica para el caso del juego de rol anterior.

Para acabar la introducción de este apartado, se hace indispensable hablar de las diferentes fases involucradas en el proceso de clasificación y de los temas de eficiencia computacional. La figura siguiente muestra en forma de rectángulo los conjuntos de datos y en forma de óvalos los procesos involucrados.

Fases y datos de un proceso de clasificación



Partimos siempre del conjunto de entrenamiento[4] que corresponde a los datos que representan a los ejemplos y a sus clases correspondientes recopilados previamente. Estos datos son la entrada para el proceso de entrenamiento, aprendizaje o construcción del modelo[5]. La salida de este proceso será el modelo de clasificación.

[4] En inglés, *training set*.

[5] En inglés, *learning* o *training*.

El conjunto de test corresponde a la representación de los ejemplos nuevos. Estos pueden ser, a modo de ejemplo, los objetos nuevos en tiempo de juego. El proceso de clasificación toma como entrada este conjunto y el modelo generado en tiempo de entrenamiento, y da como salida las predicciones de los ejemplos del conjunto de test.

En el campo de la programación de videojuegos hay que tener muy en cuenta la eficiencia computacional. Es muy importante saber dónde poner los dos procesos y qué algoritmos utilizar teniendo en cuenta esto. Dado un problema concreto, podríamos decidir incluir el proceso de entrenamiento en tiempo

de implementación del juego y el de clasificación en tiempo de juego. Esto es posible porque son dos procesos independientes y siempre podemos guardar de alguna manera el modelo de clasificación.

En las secciones siguientes veremos cómo utilizar una librería de aprendizaje automático que se puede utilizar desde Unity y una serie de algoritmos de clasificación incluidos en esta. En el apartado «Análisis del método» de cada uno de los algoritmos adaptaremos los temas computacionales mencionados anteriormente a cada caso concreto.

4.2.1. Accord.NET Framework

El Accord.NET es un sistema de librerías para C# que contiene la implementación de muchos algoritmos de aprendizaje automático y de gestión de los datos asociados a estos. Utilizaremos esta librería durante todo este apartado.

Veremos dos formas de trabajar: una directamente en Visual Studio para el aprendizaje en tiempo de diseño y otra con Unity 3D para el aprendizaje en tiempo de juego. A continuación vienen dos subapartados que indican cómo invocar y/o instalar la librería en estos dos entornos y cómo daremos los datos de salida en cada uno de ellos.

Accord.NET y Visual Studio

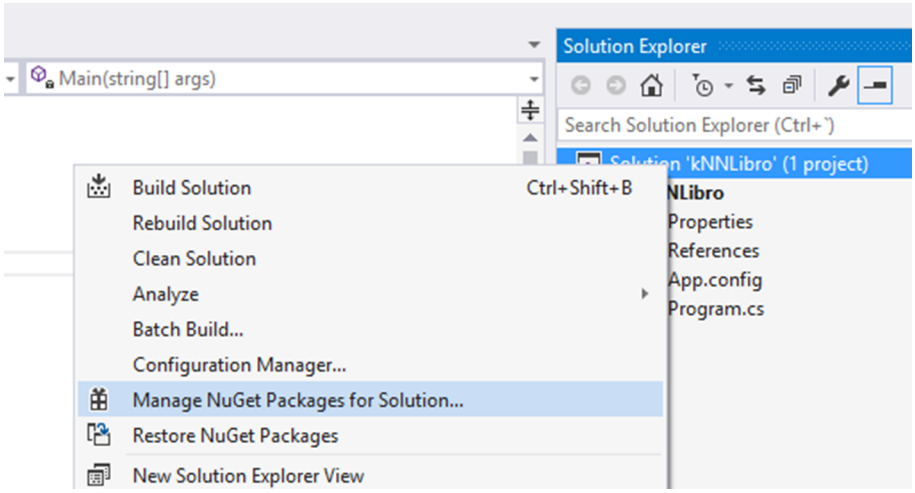
El primer tema que hemos de tener en cuenta es que esta librería se instala a nivel de proyecto. Así, en este apartado vamos a ver cómo se crea un proyecto nuevo en Visual Studio y cómo le instalamos el Accord utilizando una herramienta de gestión de paquetes llamada NuGet.

Al trabajar directamente en Visual Studio trabajaremos con aplicaciones de consola, que son las que se parecen más a los *scripts* de Unity.

Teniendo en cuenta todo esto, empezaremos clicando la opción *File* del menú, *New* y *Project*. Nos aparecerá un cuadro de diálogo en el que seleccionaremos *Console Aplication* dentro de *C#* y elegiremos la ubicación y el nombre que nos interese del proyecto.

Una vez confirmado, nos aparece la plantilla para crear el programa a la izquierda y una ventana a la derecha a la que llama *Solution Explorer*. Para invocar al gestor de paquetes NuGet tenemos que clicar con el botón derecho del ratón sobre el texto superior del árbol que muestra esta ventana (suele llevar el nombre *Solution '<nombre del proyecto>'*). Nos aparecerá un menú de contexto del que tenemos que seleccionar la opción *Manage NuGet Packages for Solution…* como muestra la figura siguiente.

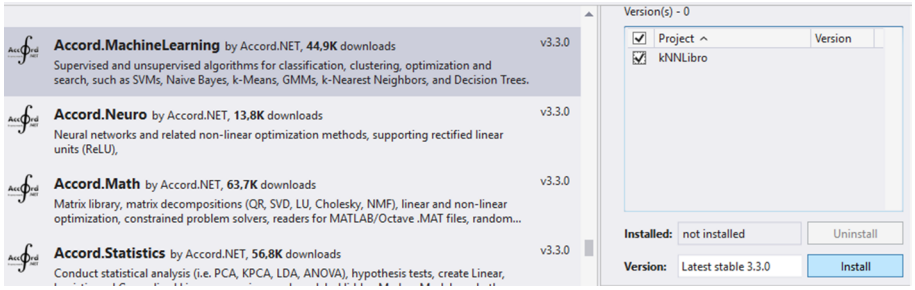
Acceso al NuGet



En la ventana de la izquierda nos aparecerá el gestor de paquetes. En esta ventana introducimos *Accord* en el cuadro de texto y clicamos en *Browse*. Para la mayoría (si no todos) de los ejemplos del apartado tenemos suficiente con instalar el módulo Accord.MachineLearning[6]. Seleccionamos el módulo, activamos a la derecha el proyecto y clicamos el botón de *Install* como muestra la figura siguiente.

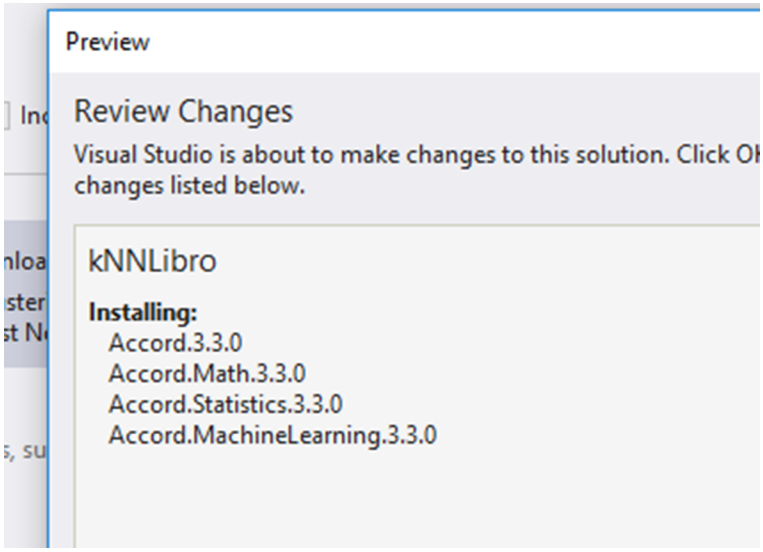
(6)Existe, además, un módulo llamado Accord.MachineLearning.GPL, que no es el que necesitamos.

Instalación de Accord.MachineLearning



Veremos que se nos pide confirmación para instalar las dependencias... (ver figura siguiente).

Dependencias del Accord.MachineLearning



Clicamos en *Ok* y aceptamos la licencia que nos muestra a continuación. Una vez hecho esto, podemos volver al programa y utilizar la librería.

En los ejemplos que veremos en esta sección se utilizará directamente Visual Studio empezando por lo que se acaba de describir y se dará la salida de los programas con *Console.Writeln*. En el caso de querer ejecutarlos en Unity, se tendrá que cambiar esta línea por la que se describe en el siguiente apartado.

Accord.NET y Unity 3D

En este apartado vamos a ver cómo instalar e invocar la librería Accord en un proyecto de Unity y cómo adaptar los ejemplos que aparecen a lo largo de esta sección a este caso.

Antes de empezar, tenemos que hablar de una característica de Unity que afecta en profundidad al uso de librerías externas C# o .NET. Unity está basado en una versión de C# abierta un poco antigua.

Observación

Hace unos años C# no era abierto y disponía de una versión que sí lo era. Unity se basa en esta versión antigua. Hace ya algunos años que Microsoft abrió el C# y por tanto ahora sí que lo es.

Esto hace que solo se puedan utilizar librerías que están compiladas para la versión 3.5 de .NET. Desde la comunidad de desarrolladores se está pidiendo que se actualice la versión de C# incluida en Unity. En teoría, cuando esto ocurra, la instalación del Accord funcionará como se detalla en el apartado anterior, invocando a Visual Studio desde un *script* de Unity y utilizando el gestor de paquetes NuGet.

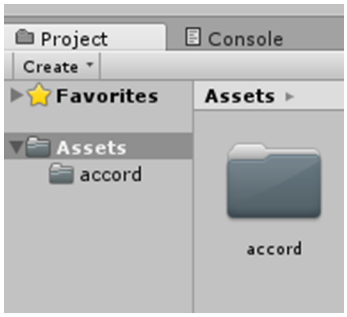
A día del desarrollo de este material esto todavía no es una realidad[7]. A partir de este momento detallamos cómo crear una aplicación muy simple en Unity 3D que invoque a uno de los ejemplos (el del kNN) de las secciones siguientes.

(7)Unity 3D versión 5.4.1f1 Personal.

Empezaremos por crear un proyecto nuevo 2D en Unity. La manera más simple de trabajar con Accord en Unity es copiar las librerías del Accord en una carpeta ubicada en *Assets* (ver figura siguiente). Bajaremos la librería de la Web[8], la descomprimiremos y copiaremos todos los archivos de la carpeta *Release/net35* relacionados con *Accord*, *Accord.Math*, *Accord.Statistics* y *Accord.MachineLearning* en una carpeta del *Assets* del proyecto de Unity.

(8)URL: http://accord-framework.net/index.html

Carpeta dentro de Assets



Ahora tenemos que cambiar el *API Compatibility Level*. Para hacerlo, hemos de mostrar los Player Settings en el *Inspector* clicando en *Edit – Proyect Settings – Player*. Seleccionamos la opción .NET 2.0 del *API Compatibility Level* del apartado *Optimization* como muestra la figura siguiente.

API Compatibility Level

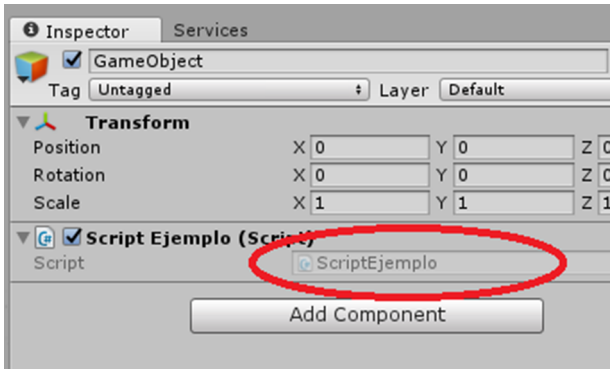
Una vez hecho esto, Unity ya reconoce la librería y la podemos invocar.

Añadiremos a continuación un objeto vacío[9] al proyecto. Una vez hecho esto, le añadimos un *script*[10]. Podemos acceder ahora al *script* con un doble clic en el nombre del *script* como marca la figura siguiente. Esto invocará al Visual Studio con el código del *script* abierto.

(9)Ventana Hierarchy – Create – Create Empty.

(10)En la ventana del *Inspector* teniendo seleccionado el objeto, añadimos el *script Add component*.

Acceso a un *script*

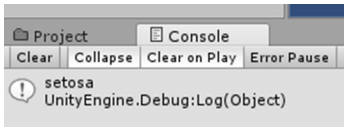


En este punto, copiar el código que aparece a continuación:

```
using Accord.MachineLearning;
using UnityEngine;

public class ScriptEjemplo : MonoBehaviour {
void Start () {
      double[][] inputs =
      {
            new double[] {5.1, 3.5, 1.4, 0.2},
            new double[] {4.9, 3.0, 1.4, 0.2},
            new double[] {6.1, 2.9, 4.7, 1.4},
            new double[] {5.6, 2.9, 3.6, 1.3},
            new double[] {7.6, 3.0, 6.6, 2.1},
            new double[] {4.9, 2.5, 4.5, 1.7},
         };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificación para la salida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

      // Entrenamiento: k=3
      int k = 3;
      var knn = new KNearestNeighbors(k, inputs, outputs);

      // Ejemplo de test
      double[] instance = { 4.9, 3.1, 1.5, 0.1 };

      // Clasificación: resultado int
      int pred = knn.Compute(instance);

      // Escritura del resultado en Visual Studio
      //      Console.WriteLine(codificacionOutputs[pred]);
      // Escritura del resultado en la consola de Unity 3D
      Debug.Log(codificacionOutputs[pred]);
   }
}
```

Después de guardar, compilar y cerrar el Visual Studio, estamos listos para ejecutar el código del ejemplo como muestra la figura siguiente.

Ejecución del ejemplo Unity 3D



En el código anterior tenemos que mencionar la diferencia al dar los resultados de salida con respecto a las versiones de Visual Studio. En este caso utilizaremos el *Debug.Log* y en los de Visual Studio el *Console.WriteLine* y eliminar el *Console.ReadKey()* para la espera de una tecla.

4.2.2. Naïve Bayes

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero, y grosor y color del tronco. La tabla 1 muestra ejemplos de este tipo de datos:

Tabla 1. Conjunto de entrenamiento

cap-shape	*cap-color*	*gill-size*	*gill-color*	*class*
convex	*brown*	*narrow*	*black*	*poisonous*
convex	*yellow*	*broad*	*black*	*edible*
bell	*white*	*broad*	*brown*	*edible*
convex	*white*	*narrow*	*brown*	*poisonous*
convex	*yellow*	*broad*	*brown*	*edible*
bell	*white*	*broad*	*brown*	*edible*
convex	*white*	*narrow*	*pink*	*poisonous*

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

El Naïve Bayes[11] es el representante más simple de los algoritmos basados en probabilidades; está basado en el teorema de Bayes.

(11)El algoritmo de Naïve Bayes lo describen Duda y Hart en su forma más clásica en 1973.

El algoritmo de Naïve Bayes clasifica nuevos ejemplos $x = (x_1,\ldots,x_m)$ asignándole la clase k que maximiza la probabilidad condicional de la clase dada la secuencia observada de entradas del ejemplo. Es decir,

$$argmax_k \quad P(k|x_1, \cdots, x_m) = argmax_k \frac{P(x_1, \cdots, x_m|k)P(k)}{P(x_1, \cdots, x_m)} \approx argmax_k P(k)\prod_{i=1}^{m} P(x_i|k)$$

Donde $P(k)$ y $P(x_i|k)$ se estiman a partir del conjunto de entrenamiento, utilizando las frecuencias relativas (estimación de la máxima verosimilitud[12]).

(12)En inglés, *maximum likelihood estimation.*

Ejemplo de aplicación

La tabla 1 muestra un ejemplo de conjunto de entrenamiento en el que tenemos que detectar si una seta es venenosa o comestible en función de sus propiedades.

Durante el proceso de entrenamiento empezaremos por calcular $P(k)$ para cada una de las clases. Así, aplicando una estimación de la máxima verosimilitud obtenemos $P(poisonous) = 3/7 = 0{,}43$ y $P(\text{edible}) = 4/7 = 0{,}57$. El segundo paso consiste en calcular $P(x_i|k)$ para cada pareja atributo-valor y para cada clase, de la misma forma que en el caso anterior. La tabla 2 muestra los resultados de este proceso. El 1 de la celda correspondiente a la

fila *cap-shape: convex* y columna *poisonous* sale de que los tres ejemplos etiquetados como *poisonous* tienen el valor *convex* para el atributo *cap-shape*.

Tabla 2. Valores de $P(x_i|k)$

atributo-valor	*poisonous*	*edible*
cap-shape: convex	1	0,5
cap-shape: bell	0	0,5
cap-color: brown	0,33	0
cap-color: yellow	0	0,5
cap-color: white	0,67	0,5
gill-size: narrow	1	0
gill-size: broad	0	1
gill-color: black	0,33	0,25
gill-color: brown	0,33	0,75
gill-color: pink	0,33	0

Con esto habremos terminado el proceso de entrenamiento. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 3, tendremos que aplicar la fórmula $argmax_k P(k)\prod_{i=1}^{m} P(x_i|k)$ para clasificarlo.

Tabla 3. Ejemplo de test

cap-shape	*cap-color*	*gill-size*	*gill-color*	*class*
convex	*brown*	*narrow*	*black*	*poisonous*

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Empezamos por calcular la parte del valor *poisonous*. Para ello, tendremos que multiplicar entre sí las probabilidades: *P*(*poisonous*), *P*(*cap - shape*:*convex*|*poisonous*), *P*(*cap - color*:*brown*|*poisonous*), *P*(*gill - size*:*narrow*|*poisonous*) y *P*(*gill - color*:*black*|*poisonous*); que equivale a un valor de 0,05. Realizando el mismo proceso para la clase *edible* obtenemos un valor de 0. Por tanto, la clase que maximiza la fórmula de las probabilidades es *poisonous*; con lo que el método está clasificando correctamente el ejemplo de test, dado este conjunto de entrenamiento.

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using Accord;
using Accord.MachineLearning.Bayes;
using Accord.Math;
using Accord.Statistics.Filters;
using System;
using System.Data;

namespace NaiveBayesLibro
{
   class Program
   {
      static void Main(string[] args)
      {
```

```
            // Creación del conjunto de datos
            // Carga de datos en la variable data
            DataTable data = new DataTable("Mushroom Data");
            data.Columns.Add("cap-shape", "cap-color", "gill-size", "gill-color", "class");
            data.Rows.Add("convex", "brown", "narrow", "black", "poisonous");
            data.Rows.Add("convex", "yellow", "broad", "black", "edible");
            data.Rows.Add("bell", "white", "broad", "brown", "edible");
            data.Rows.Add("convex", "white", "narrow", "brown", "poisonous");
            data.Rows.Add("convex", "yellow", "broad", "brown", "edible");
            data.Rows.Add("bell", "white", "broad", "brown", "edible");
            data.Rows.Add("convex", "white", "narrow", "pink", "poisonous");

            // Codificación del conjunto de datos
            // Generación de las variables inputs i outputs a partir de data
            Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-size",
            "gill-color", "class");
            DataTable symbols = codebook.Apply(data);
          int[][] inputs = symbols.ToArray<int>("cap-shape", "cap-color", "gill-size", "gill-color");
            int[] outputs = symbols.ToArray<int>("class");

            // Entrenamiento
            // Construcción del modelo en la variable nb
            // Se pueden consultar las probabilidades del modelo
            var learner = new NaiveBayesLearning();
            NaiveBayes nb = learner.Learn(inputs, outputs);

            // Creación del ejemplo de test
            // Generación y codificación del ejemplo de test en instance
            int[] instance = codebook.Translate("convex", "brown", "narrow", "black");
            // Clasificación
            // El resultado es numérico
            int pred = nb.Decide(instance);

            // Decodificación de la predicción
            string result = codebook.Translate("class", pred);
            // Escritura del resultado
            Console.WriteLine("Predicción: {0}", result);

            // Espera de una tecla
            Console.ReadKey();
        }
    }
}
```

Análisis del método

Uno de los problemas del Naïve Bayes, que ha sido mencionado frecuentemente en la literatura, es que el algoritmo asume la independencia de los diferentes atributos que representan a un ejemplo. Así, es poco probable que el color del tallo de una seta no esté relacionado con el color de su sombrero.

Otra característica destacable es que cuando el conjunto de entrenamiento no está balanceado[13], tiende a clasificar los ejemplos hacia la clase que tiene más ejemplos dentro del conjunto de entrenamiento.

(13)Un conjunto de entrenamiento está balanceado cuando tiene el mismo número de ejemplos de cada una de las clases que contiene.

Dos de las ventajas que presenta el método son su simplicidad y eficiencia computacional. Las dos fases del aprendizaje son rápidas. La fase de entrenamiento consiste en la carga y/o preparación de los datos y en la estimación de las probabilidades $P(k)$ y $P(x_i|k)$. La fase de test o clasificación consiste en la preparación de los ejemplos de test y en la multiplicación de las probabilidades pertinentes. Hay que tener en cuenta dónde hacemos cada una de las fases, sobre todo la de entrenamiento. Las dos fases pueden estar separadas, ya que disponemos del modelo.

No obstante, y a pesar de sus inconvenientes, este método ha sido muy utilizado históricamente y se han obtenido buenos resultados para muchos conjuntos de datos. Esto ocurre cuando el conjunto de entrenamiento representa bien las distribuciones de probabilidad del problema.

Hasta ahora, hemos hablado exclusivamente de atributos nominales. En el caso de tener un problema representado con algún atributo continuo, como los numéricos, es necesario algún tipo de proceso. Hay diversas formas de hacerlo; una consiste en categorizarlos, dividiendo el continuo en intervalos. Otra opción consiste en asumir que los valores de cada clase siguen una distribución gaussiana y aplicar la fórmula:

$$P(x = v|k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{\left(v-\mu_k\right)^2}{2\sigma_k^2}\right)$$

donde x corresponde al atributo, v a su valor, k a la clase, μ_k al promedio de valores de la clase k y σ_k a su desviación estándar. Esta modificación del algoritmo recibe el nombre de Gaussian Naïve Bayes y es de uso muy común. Lamentablemente, no tenemos esta implementación en la librería Accord.Net.

4.2.3. kNN

Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las

diferentes estanterías del almacén. Las medidas que envía el sistema láser son la longitud y anchura del sépalo y el pétalo de cada flor. La tabla 4 muestra ejemplos de este tipo de datos.

Tabla 4. Conjunto de entrenamiento

sepal-length	*sepal-width*	*petal-length*	*petal-width*	*class*
5,1	3,5	1,4	0,2	*setosa*
4,9	3,0	1,4	0,2	*setosa*
6,1	2,9	4,7	1,4	*versicolor*
5,6	2,9	3,6	1,3	*versicolor*
7,6	3,0	6,6	2,1	*virginica*
4,9	2,5	4,5	1,7	*virginica*

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

En kNN (*k*vecinos más cercanos[14]) la clasificación de nuevos ejemplos se realiza buscando el conjunto de los *k* ejemplos más cercanos de entre un conjunto de ejemplos etiquetados previamente guardados y seleccionando la clase más frecuente de entre sus etiquetas. La generalización se pospone hasta el momento de la clasificación de nuevos ejemplos[15].

(14)En inglés, *k nearest neighbours*.

(15)Por esta razón es por la que en ocasiones es llamado aprendizaje perezoso (en inglés, *lazy learning*).

Una parte muy importante de este método es la definición de la medida de distancia (o similitud) apropiada para el problema que tratar. Esta debería tener en cuenta la importancia relativa de cada atributo y ser eficiente computacionalmente. El tipo de combinación para elegir el resultado de entre los *k* ejemplos más cercanos y el valor de la propia *k* también son cuestiones que decidir de entre varias alternativas.

Este algoritmo, en su forma más simple, guarda en memoria todos los ejemplos durante el proceso de entrenamiento y la clasificación de nuevos ejemplos se basa en las clases de los *k*ejemplos más cercanos[16]. Para obtener el conjunto de los *k* vecinos más cercanos, se calcula la distancia entre el ejemplo que clasificar $x = (x_1, \cdots, x_m)$ y todos los ejemplos guardados $x_i = (x_{i_1}, \cdots, x_{i_m})$. Una de las distancias más utilizadas es la euclidiana:

(16)Por esta razón se lo conoce también como basado en memoria, en ejemplos, en instancias o en casos.

$$de(x, x_i) = \sqrt{\sum_{j=1}^{m} (x_j - x_{i_j})^2}$$

Ejemplo de aplicación

La tabla 4 muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. En este ejemplo, aplicaremos el kNN para valores de *k* de 1 y 3, utilizando como medida de distancia la euclidiana.

El proceso de entrenamiento consiste en guardar los datos; no tenemos que hacer nada. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 5, hemos de

calcular las distancias entre el nuevo ejemplo y todos los del conjunto de entrenamiento. La tabla 6 muestra estas distancias.

Tabla 5. Ejemplo de test

sepal-length	***sepal-width***	***petal-length***	***petal-width***	***class***
4,9	3,1	1,5	0,1	*setosa*

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

Tabla 6. Distancias

0,5	0,2	3,7	2,5	6,1	3,5

Para el 1NN elegimos la clase del más cercano que coincide con el segundo ejemplo (distancia 0,2) que tiene por clase *setosa*. Para el 3NN elegimos los tres ejemplos más cercanos: primero, segundo y cuarto; con distancias respectivas: 0,5, 0,2 y 2,5. Sus clases corresponden a *setosa*, *setosa* y *versicolor*. En este caso asignaremos también a *setosa* por ser la clase más frecuente. En los dos casos el resultado es correcto.

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using System;
using Accord.MachineLearning;

namespace kNNLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunto de entrenamiento: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificación para la salida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

      // Entrenamiento: k=3
      int k = 3;
      var knn = new KNearestNeighbors(k, inputs, outputs);

      // Ejemplo de test
```

```
        double[] instance = { 4.9, 3.1, 1.5, 0.1 };

        // Clasificación: resultado int
        int pred = knn.Compute(instance);

        // Escritura del resultado
        Console.WriteLine(codificacionOutputs[pred]);

        // Espera de una tecla
        Console.ReadKey();
    }
    }
}
```

Análisis del método

Un aspecto que cabe tener en cuenta es la eficiencia computacional, el algoritmo realiza todos los cálculos en el proceso de clasificación. Así, aun siendo un método rápido globalmente, hemos de tener en cuenta que el proceso de clasificación no lo es. Esto puede llegar a ser crítico para juegos que necesiten una respuesta rápida.

En el momento en el que queramos aplicar este algoritmo a un problema, la primera decisión que debemos tomar es la medida de distancia, y la segunda, el valor de *k*. Se suele elegir un número impar o primo para minimizar la posibilidad de empates en las votaciones. La siguiente cuestión que hay que decidir es el tratamiento de los empates cuando la *k* es superior a uno. Algunos heurísticos posibles son: no dar predicción en caso de empate, dar la clase más frecuente en el conjunto de entrenamiento de entre las clases seleccionadas para votar, etc. Se suele elegir el heurístico en función del problema que hayamos de tratar.

Las distancias más utilizadas son la euclidiana y la de Hamming, en función del tipo de atributos que tengamos. La primera se utiliza mayoritariamente para atributos numéricos, y la segunda, para atributos nominales o binarios. La distancia de Hamming está incluida en la librería Accord.NET.

Una de las grandes ventajas de este método es la conservación de excepciones en el proceso de generalización.

4.2.4. Clasificador lineal

Muy parecida a la del algoritmo anterior, otra manera de abordar el problema de la clasificación de flores se basa en el uso de centros de masa o centroides. El modelo de clasificación de este algoritmo consta de un centroide, que representa cada una de las clases que aparecen en el conjunto de entrenamiento[17]. El valor de cada atributo del centroide se calcula como el promedio del

[17]Este método también recibe el nombre de método basado en centroides o en centros de masas.

valor del mismo atributo de todos los ejemplos del conjunto de entrenamiento que pertenecen a su clase. La fase de clasificación consiste en aplicar el 1NN con distancia euclidiana, seleccionando como conjunto de entrenamiento los centroides calculados previamente.

Ejemplo de aplicación

Aplicaremos como ejemplo el algoritmo a los ejemplos de la tabla 4. El proceso de entrenamiento consiste en calcular los centroides. La tabla 7 muestra el resultado. A modo de ejemplo, el 5 del atributo *sepal-length* del centroide *setosa* sale del promedio de 4.9 y 5.1 de la tabla 4.

Tabla 7. Centroides

sepal-length	*sepal-width*	*petal-length*	*petal-width*	*class*
5	3,25	1,4	0,2	*setosa*
5,85	2,9	4,15	1,35	*versicolor*
6,25	2,75	5,55	1,9	*virginica*

A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 5, tenemos que calcular las distancias entre el nuevo ejemplo y todos los centroides para el 1NN. La tabla 8 muestra estas distancias. Al elegir el más cercano (distancia 0,2) que tiene por clase *setosa*, vemos que el ejemplo queda bien clasificado.

Tabla 8. Distancias

0,2	3,1	4,6

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using System;
using Accord.MachineLearning;

namespace CentroidesLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunto de entrenamiento: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
```

```
        int[] outputs = { 0, 0, 1, 1, 2, 2 };
        // Codificación para la salida: 0->"setosa", 1->"versicolor", 2->"virginica"
        string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

        // Entrenamiento
        var learner = new MinimumMeanDistanceClassifier(inputs, outputs);

        // Ejemplo de test
        double[] instance = { 4.9, 3.1, 1.5, 0.1 };

        // Clasificación: resultado int
        int pred = learner.Decide(instance);

        // Espera de una tecla
        Console.WriteLine(codificacionOutputs[pred]);

        // Espera de una tecla
        Console.ReadKey();
    }
    }
}
```

Análisis del método

Las principales ventajas de este método son su simplicidad y eficiencia computacional, tanto en tiempo como en espacio. Sus inconvenientes son que solo sirve para atributos numéricos y que no sirve para abordar problemas muy complejos.

4.2.5. Árboles de decisión

Antes de empezar con este algoritmo, hay que hacer una pequeña reseña explicando la diferencia entre el árbol de decisión explicado en el apartado de toma de decisiones (apartado 3) y el actual. Los dos árboles son básicamente lo mismo: los dos guardan un modelo para la toma de decisiones. En lo que se diferencian es en quién ha generado la estructura. Los árboles de los que se habla en el apartado de toma de decisiones generalmente los han generado los implementadores o diseñadores del juego. En el caso del aprendizaje automático, el árbol se genera automáticamente a partir de los datos y constituye el modelo para un clasificador.

Una vez hecho este apunte, supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recogida de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero, y color del tronco.

Un árbol de decisión es una forma de representar reglas de clasificación inherentes a los datos, con una estructura en árbol n-ario que particiona los datos de manera recursiva. Cada rama de un árbol de decisión representa una regla que decide entre una conjunción de valores de un atributo básico (nodos internos) o realiza una predicción de la clase (nodos terminales).

El algoritmo de los árboles de decisión básico está pensado para trabajar con atributos nominales. El conjunto de entrenamiento queda definido por $S = \{(x_1,y_1),\ldots,(x_n,y_n)\}$, donde cada ejemplo x corresponde a $x_i = \left(x_{i_1}, \cdots, x_{i_m}\right)$, donde m guarda el número de atributos; y $A = \{a_1,\ldots,a_m\}$ el conjunto de atributos, donde $dom(a_j)$ corresponde al conjunto de todos los posibles valores del atributo a_j, y para cualquier valor de un ejemplo de entrenamiento $x_{ij} \in dom\left(a_j\right)$.

El proceso de construcción del árbol es un proceso iterativo, en el que, en cada iteración, se selecciona el atributo que mejor particiona el conjunto de entrenamiento. Para realizar este proceso, tenemos que mirar la bondad de las particiones que genera cada uno de los atributos y, en un segundo paso, seleccionar el mejor. La partición del atributo a_j genera $|dom(a_j)|$ conjuntos, que corresponde al número de elementos del conjunto. Existen diversas medidas para mirar la bondad de la partición. Una básica consiste en asignar a cada conjunto de la partición la clase mayoritaria de este, contar cuántos quedan bien clasificados y dividirlo por el número de ejemplos. Una vez calculadas las bondades de todos los atributos, elegimos el mejor.

Cada conjunto de la mejor partición pasará a ser un nuevo nodo del árbol. A este nodo se llegará a través de una regla del tipo atributo = valor. Si todos los ejemplos del conjunto han quedado bien clasificados, lo convertimos en nodo terminal con la clase de los ejemplos. En caso contrario, lo convertiremos en nodo interno y aplicamos una nueva iteración al conjunto («reducido») eliminando el atributo que ha generado la partición. En caso de no quedar atributos, lo convertiríamos en nodo terminal asignando la clase mayoritaria.

Para realizar el test, exploramos el árbol en función de los valores de los atributos del ejemplo de test y las reglas del árbol hasta llegar al nodo terminal, y damos como predicción la clase del nodo terminal al que lleguemos.

Ejemplo de aplicación

La tabla 9 es una simplificación de la tabla 4. Para construir un árbol de decisión a partir de este conjunto, hemos de calcular la bondad de las particiones de los tres atributos: *cap-shape*, *cap-color* y *gill-color*. El atributo *cap-shape* nos genera una partición con dos conjuntos: uno para el valor *convex* y otro para *bell*. La clase mayoritaria para el conjunto de *convex* es *poisonous* y la de *bell* es *edible*; su bondad es *bondad*(*cap-shape*) = (3+2)/7 = 0,71.

El 7 corresponde al número total de ejemplos. Si fijamos el valor *convex* nos quedan las clases *{poisonous, edible, poisonous, edible, poisonous}* para *cap-shape*; la mayoritaria corresponde a *poisonous*, que está tres veces. Las dos clases de *bell* son *edible*, que está dos veces.

Si realizamos el mismo proceso para el resto de los atributos, obtenemos: *bondad*(*cap-color*) = (1+2+2)/7 = 0,71 y *bondad*(*cap-color*) = (1+3+1)/7 = 0,71.

Tabla 9. Conjunto de entrenamiento

cap-shape	***cap-color***	***gill-color***	***class***
convex	*brown*	*black*	*poisonous*
convex	*yellow*	*black*	*edible*
bell	*white*	*brown*	*edible*
convex	*white*	*brown*	*poisonous*
convex	*yellow*	*brown*	*edible*
bell	*white*	*brown*	*edible*
convex	*white*	*pink*	*poisonous*

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

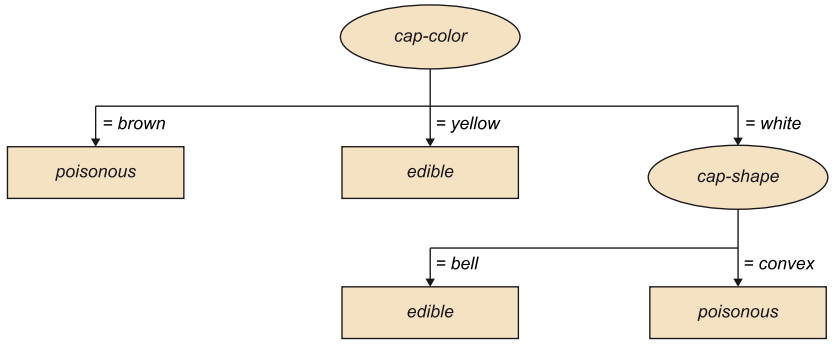
El siguiente paso consiste en seleccionar el mejor atributo. Hemos obtenido un empate entre los tres atributos, así que podemos elegir cualquiera de ellos; escogemos *cap-color*. Los nodos generados por el conjunto de *brown* y *yellow* son terminales y les asignamos las clases *poisonous* y *edible*, respectivamente. Esto se debe a que obtienen los dos conjuntos una bondad de 1. El nodo de *white* lo volvemos a hacer iterar a partir del conjunto que muestra la tabla 10. Este conjunto lo obtenemos de eliminar el atributo *cap-color* de los ejemplos que tienen el valor *white* para el atributo *cap-color*. La bondad de las nuevas particiones es *bondad*(*cap-shape*) = (2+2)/4 = 1 y *bondad*(*gill-color*) = (2+1)/4 = 0,75. El mejor atributo es *cap-shape*, que genera dos nodos terminales con las clases *edible* para *bell* y *poisonous para convex*.

Tabla 10. Conjunto en la segunda iteración

cap-shape	***gill-color***	***class***
bell	*brown*	*edible*
convex	*brown*	*poisonous*
bell	*brown*	*edible*
convex	*pink*	*poisonous*

La figura siguiente muestra el árbol construido en este proceso.

Árbol de decisión



Para etiquetar un ejemplo de test como el que muestra la tabla 11 tenemos que recorrer el árbol, partiendo de la raíz, eligiendo las ramas correspondientes a los valores de los atributos de los ejemplos de test. Para este caso, miramos el valor del atributo *cap-color* y bajamos por la rama que corresponde al valor *brown*. Llegamos a un nodo terminal con

clase *poisonous*, con lo que se la asignaremos al ejemplo de test como predicción. Esta predicción es correcta.

Tabla 11. Ejemplo de test

cap-shape	***cap-color***	***gill-color***	***class***
convex	*brown*	*black*	*poisonous*

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using System;
using System.Data;
using Accord;
using Accord.Math;
using Accord.Statistics.Filters;
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;

namespace DTsLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Creación del conjunto de datos
      // Carga de datos en la variable data
      DataTable data = new DataTable("Mushroom Data");
      data.Columns.Add("cap-shape", "cap-color", "gill-color", "class");
      data.Rows.Add("convex", "brown", "black", "poisonous");
      data.Rows.Add("convex", "yellow", "black", "edible");
      data.Rows.Add("bell", "white", "brown", "edible");
      data.Rows.Add("convex", "white", "brown", "poisonous");
      data.Rows.Add("convex", "yellow", "brown", "edible");
      data.Rows.Add("bell", "white", "brown", "edible");
      data.Rows.Add("convex", "white", "pink", "poisonous");

      // Codificación del conjunto de datos
      // Generación de las variables inputs i outputs a partir de data
    Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-color", "class");
      DataTable symbols = codebook.Apply(data);
      double[][] inputs = symbols.ToArray<double>("cap-shape", "cap-color", "gill-color");
      int[] outputs = symbols.ToArray<int>("class");

      // Entrenamiento
      string[] inputColumns = { "cap-shape", "cap-color", "gill-color" };
```

```
        var attributes = DecisionVariable.FromCodebook(codebook, inputColumns);
        DecisionTree tree = new DecisionTree(attributes, 2);
        C45Learning c45 = new C45Learning(tree);
        c45.Learn(inputs, outputs);

        // Creación del ejemplo de test
        // Generación y codificación del ejemplo de test en instance
        int[] instance = codebook.Translate("convex", "brown", "black");
        // Clasificación
        // El resultado es numérico
        int pred = tree.Decide(instance);
        // Decodificación de la predicción
        string result = codebook.Translate("class", pred);
        // Escritura del resultado
        Console.WriteLine("Predicción: {0}", result);

        // Espera de una tecla
        Console.ReadKey();
    }
    }
}
```

Tratamiento de atributos numéricos

Supongamos que ahora queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra la longitud del sépalo de las flores, y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén.

El algoritmo de los árboles de decisión fue diseñado para tratar con atributos nominales, pero existen alternativas para el tratamiento de atributos numéricos. El más común se basa en la utilización de puntos de corte, que son un punto que divide el conjunto de valores de un atributo en dos (los menores y los mayores del punto de corte).

Para calcular el mejor punto de corte de un atributo numérico, se ordenan los valores y se eliminan los elementos repetidos. Se calculan los posibles puntos de corte como el promedio de cada dos valores consecutivos. Como último paso, se calcula el mejor de ellos como aquel con mejor bondad.

Hemos de tener en cuenta que el tratamiento de atributos numéricos genera árboles (y decisiones) binarios. Esto implica que en los siguientes niveles del árbol tendremos que volver a procesar el atributo numérico que acabamos de seleccionar, a diferencia de los atributos nominales.

Ejemplo del cálculo del mejor punto de corte de un atributo numérico

A continuación se expone el cálculo del mejor punto de corte para el atributo *sepal-length* del conjunto de datos que muestra la tabla 4. La tabla 12 muestra estos cálculos: las columnas 1 y 2 muestran la clase y el atributo ordenados por el atributo, la 3 muestra los valores sin los repetidos; la 4, los puntos de corte como el promedio de cada dos valores consecutivos, y la última, las bondades de los puntos de corte. El mejor punto de corte es 5,35, con una bondad de 66,7 %.

Tabla 12. Cálculos para el mejor punto de corte

class	*sepal-length*	sin repetidos	puntos de corte	bondad
setosa	4,9			
virginica	4,9	4,9	5	(1 + 2) / 6 = 0,5
setosa	5,1	5,1	5,35	(2 + 2) / 6 = 0,67
versicolor	5,6	5,6	5,85	(2 + 1) / 6 = 0,5
versicolor	6,1	6,1	6,85	(2 + 1) / 6 = 0,5
virginica	7,6	7,6		

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;
using System;

namespace DTsIrisLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunto de entrenamiento: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificación para la salida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };
```

```
        // Entrenamiento
        // nombres de los atributos
        DecisionVariable[] features =
           {
        new DecisionVariable("sepal length", DecisionVariableKind.Continuous),
        new DecisionVariable("sepal width", DecisionVariableKind.Continuous),
        new DecisionVariable("petal length", DecisionVariableKind.Continuous),
        new DecisionVariable("petal width", DecisionVariableKind.Continuous),
        };

        // Creamos un árbol para 3 clases
        var tree = new DecisionTree(inputs: features, classes: 3);

        // Utilizamos el C4.5 para el entrenamiento
        var teacher = new C45Learning(tree);

        // Inducimos el árbol
        teacher.Learn(inputs, outputs);

        // Ejemplo de test
        double[] instance = { 4.9, 3.1, 1.5, 0.1 };

        // Clasificación
        int pred = tree.Decide(instance);
        Console.WriteLine(codificacionOutputs[pred]);

        Console.ReadKey();
    }
    }
}
```

Análisis del método

Este método es el menos eficiente computacionalmente de los que hemos visto en esta sección, pero tiene la gran ventaja de la facilidad de interpretación del modelo de aprendizaje. Un árbol de decisión nos da la información clara de la toma de decisiones y de la importancia de los diferentes atributos involucrados. El árbol de decisión generado se puede convertir en un conjunto de reglas que se pueden implementar con estructuras alternativas.

Dos de los grandes inconvenientes que plantea son la elevada fragmentación de los datos en presencia de atributos con muchos valores y el elevado coste computacional que esto implica. Esto provoca que no sea un método muy adecuado para problemas con grandes espacios de atributos.

Otro inconveniente que debemos tener en cuenta es que los nodos terminales correspondientes a reglas que dan cobertura a pocos ejemplos de entrenamiento no producen estimaciones fiables de las clases. Tiende a sobreentrenar el conjunto de entrenamiento[18]. Para suavizar este efecto se puede utilizar alguna técnica de poda[19].

(18)Este efecto se conoce en inglés como *overfitting*.

(19)En inglés, *prunning*.

4.2.6. Algoritmo ID3

El algoritmo ID3 es una modificación de los árboles de decisión que utiliza la ganancia de información[20] como medida de bondad para analizar los atributos. La ganancia de información es un concepto que está basado en la entropía. El uso de la entropía tiende a penalizar más los conjuntos mezclados.

(20)En inglés, *information gain*.

La entropía es una medida de la teoría de la información que cuantifica el desorden. Así, dado un conjunto *S*, su fórmula viene dada por:

$$H(S) = -\sum_{y \in Y} p(y) log_2(p(y))$$

donde *p(y)* es la proporción de ejemplos de *S* que pertenece a la clase *y*. La entropía será mínima (0) cuando en *S* hay una sola clase, y será máxima (1) cuando el conjunto es totalmente aleatorio.

A modo de ejemplo, la entropía del conjunto que muestra la tabla 9 viene dada por:

$$H(S) = -\frac{3}{7} log_2 \frac{3}{7} - \frac{4}{7} log_2 \frac{4}{7} = 0{,}985$$

La fórmula de la ganancia de información para el conjunto *S* y el atributo queda como:

$$G(S, a_j) = H(S) - \sum_{v \in a_j} p(v) H(S_v)$$

donde *p(v)* es la proporción de ejemplos de *S* que tienen el valor *v* para el atributo a_j, y S_v es el subconjunto de *S* de los ejemplos que toman el valor *v* para el atributo a_j.

A modo de ejemplo, la ganancia de información del conjunto que muestra la tabla 9 y el atributo *cap-shape* viene dada por:

$$G(S, cs) = H(S) - \frac{5}{7} H(cs = convex) - \frac{2}{7} H(cs = bell) = 0{,}292$$

donde *cs* es *cap-shape*,

$$H(cs = convex) = -\frac{3}{5}log_2\frac{3}{5} - \frac{2}{5}log_2\frac{2}{5} = 0{,}971$$

y

$$H(cs = bell) = -\frac{0}{2}log_2\frac{0}{2} - \frac{2}{2}log_2\frac{2}{2} = 0$$

Poda de los árboles de decisión

El objetivo de la poda de los árboles de decisión es obtener árboles que no tengan en las hojas reglas que afecten a pocos ejemplos del conjunto de entrenamiento. Es deseable no forzar la construcción del árbol para evitar el sobreentrenamiento[21].

[21] En inglés, *overfitting*.

La primera aproximación para abordar la solución de este problema consiste en establecer un umbral de reducción de la entropía. Es decir, pararemos una rama cuando la disminución de entropía no supere dicho umbral. Esta aproximación plantea el problema de que puede que no se reduzca en un cierto paso pero sí en el siguiente.

Otra aproximación, que soluciona el problema de la anterior y que es la que se suele utilizar, consiste en construir todo el árbol y, en una segunda fase, eliminar los nodos superfluos (poda del árbol[22]). Para realizar este proceso de poda recorremos el árbol de forma ascendente (de las hojas a la raíz) y vamos mirando si cada par de nodos incrementa la entropía por encima de un cierto umbral si los juntáramos. Si esto sucede, deshacemos la partición.

[22] En inglés, *prunning*.

Uso en Accord.NET

A continuación tenéis a vuestra disposición un *script* que implementa el ejemplo anterior utilizando la librería Accord:

```
using Accord;
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;
using Accord.Math;
using Accord.Statistics.Filters;
using System;
using System.Data;

namespace ID3Libro
{
   class Program
   {
      static void Main(string[] args)
      {
         // Creación del conjunto de datos
```

```
            // Carga de datos en la variable data
            DataTable data = new DataTable("Mushroom Data");
            data.Columns.Add("cap-shape", "cap-color", "gill-color", "class");
            data.Rows.Add("convex", "brown", "black", "poisonous");
            data.Rows.Add("convex", "yellow", "black", "edible");
            data.Rows.Add("bell", "white", "brown", "edible");
            data.Rows.Add("convex", "white", "brown", "poisonous");
            data.Rows.Add("convex", "yellow", "brown", "edible");
            data.Rows.Add("bell", "white", "brown", "edible");
            data.Rows.Add("convex", "white", "pink", "poisonous");

            // Codificación del conjunto de datos
            // Generación de las variables inputs i outputs a partir de data
            Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-color",
            "class");
            DataTable symbols = codebook.Apply(data);
            int[][] inputs = symbols.ToArray<int>("cap-shape", "cap-color", "gill-color");
            int[] outputs = symbols.ToArray<int>("class");

            // Entrenamiento
            string[] inputColumns = { "cap-shape", "cap-color", "gill-color" };
            var attributes = DecisionVariable.FromCodebook(codebook, inputColumns);
            DecisionTree tree = new DecisionTree(attributes, 2);
            ID3Learning id3learning = new ID3Learning(tree);
            id3learning.Learn(inputs, outputs);

            // Creación del ejemplo de test
            // Generación y codificación del ejemplo de test en instance
            int[] instance = codebook.Translate("convex", "brown", "black");
            // Clasificación
            // El resultado es numérico
            int pred = tree.Decide(instance);
            // Decodificación de la predicción
            string result = codebook.Translate("class", pred);
            // Escritura del resultado
            Console.WriteLine("Predicción: {0}", result);

            // Espera de una tecla
            Console.ReadKey();
        }
    }
}
```

4.3. Ajuste de parámetros

Imaginemos que hemos creado un estupendo juego de estrategia, o de tipo *tower defense*, o cualquier otro en el que haya que balancear diferentes atributos de las unidades (poder de ataque, puntos de vida, etc.). Antes de comercializarlo, es fundamental ajustar muy bien esos parámetros para conseguir que el juego tenga el nivel de dificultad justo para resultar entretenido, y también para conseguir que todos los tipos de unidades resulten útiles, es decir, que no haya un tipo de unidad claramente mejor que las demás, ya que eso acabará haciendo que nadie use las demás unidades y el juego sea mucho más aburrido de lo que pretendíamos.

Nos encontramos ante un problema de **ajuste de parámetros**, en los que existe un conjunto de parámetros interrelacionados y hay que encontrar una combinación adecuada para los objetivos deseados, en este caso ajustar la dificultad y variedad del juego. El problema es que, en general, no se pueden ajustar de uno en uno, ya que al modificar uno el valor de todos los demás debe reajustarse para que el conjunto funcione bien. Por ejemplo, si a un tanque le aumentamos el ataque puede que sea necesario bajarle la defensa, pues de lo contrario será una unidad demasiado poderosa y desequilibrará completamente el juego.

Para conseguir ese buen ajuste de parámetros, podríamos pensar en probar todas las combinaciones posibles y quedarnos con la mejor, pero enseguida veremos que eso es imposible: imaginemos un juego con diez tipos de unidades, con tres atributos cada una (ataque, defensa y vida) y en la que estos atributos pueden tomar un valor entre 1 y 20. Tenemos que ajustar 3 x 10 = 30 parámetros, y como cada uno tiene 20 valores posibles, en total hay 20^{30} combinaciones de parámetros. Evidentemente, no es muy práctico probarlas todas.

Maldición de la dimensionalidad

Se llama maldición de la dimensionalidad (*curse of dimensionality*) el hecho de que cada parámetro que se añade a un problema aumenta el espacio de soluciones posibles en una dimensión, con lo que el crecimiento del número de soluciones posibles es exponencial respecto al número de parámetros.

Para afrontar este tipo de situaciones existen los algoritmos de **optimización**, en los que se utiliza un método que permite encontrar una solución «razonablemente» buena a un problema en un tiempo acotado. La clave es que, como no es posible probar todas las soluciones, se intenta encontrar la mejor solución posible en el tiempo disponible para llevar a cabo la optimización. De

ahí lo de «razonablemente», ya que en general no se puede asegurar que la solución encontrada sea la mejor posible (habría que probarlas todas), pero se espera que al menos sea mejor que una solución cualquiera generada al azar.

Observación

El ajuste de los parámetros del juego es una actividad que se lleva a cabo durante su desarrollo, no se ejecuta en el ordenador de los usuarios mientras están jugando; decimos que es una ejecución *offline*. Aunque se usen algoritmos de optimización muy eficientes, su tiempo de ejecución suele ser elevado, salvo para los problemas más sencillos.

4.3.1. Optimización con algoritmos genéticos

Un método de optimización muy adecuado para resolver problemas de ajuste de parámetros, en los que hay una gran cantidad de parámetros interdependientes, son los **algoritmos genéticos**.

Teoría de la evolución

La teoría de la evolución de Charles Darwin afirma que los individuos mejor adaptados a un entorno son los que tienen más posibilidades de tener descendencia, de modo que sus características se imponen a las de otros individuos peor adaptados. Los algoritmos genéticos se inspiran en esta idea, cambiando «entorno» por «problema», «individuo» por «solución», y «adaptación» por «calidad de una solución».

Con más detalle, dado un problema *P* con un espacio de soluciones *S*, los componentes de un algoritmo genético son:

- Un **individuo**, que lo definimos como una solución concreta. Por ejemplo, en el ajuste de atributos de las unidades del juego, un individuo será una combinación concreta de atributos de todas las unidades. Siguiendo la metáfora evolutiva, cada atributo de un individuo es como un gen, y todos sus atributos (generalmente en un vector) son su genoma completo.

- Una **población**, es decir un conjunto de individuos que compiten entre ellos, ya que los mejor adaptados (las mejores soluciones) serán los que se reproducirán. Al aumentar el tamaño de la población se mejora la exploración de *S*, pero se incrementa el coste computacional. A modo de orientación, se recomienda que el tamaño de la población (número de individuos) sea igual o superior al número de atributos.

- Una **función objetivo** (o función de **error**, si lo que queremos es reducir un error). Ejemplos: duración de una partida, diferencia entre los puntos conseguidos por cada tipo de unidad, tiempo invertido en recorrer un circuito, distancia del punto de impacto de un proyectil a un objetivo. La función objetivo debe ser continua y se debe evitar que tenga cambios bruscos y zonas constantes.

- Un **sentido de operación**: ¿queremos minimizar o maximizar la función objetivo? Dependiendo de cómo la hayamos definido y qué sentido tenga, querremos una cosa u otra.

El esquema genérico de los algoritmos genéticos es como sigue:

1) Se crea una población inicial aleatoriamente.

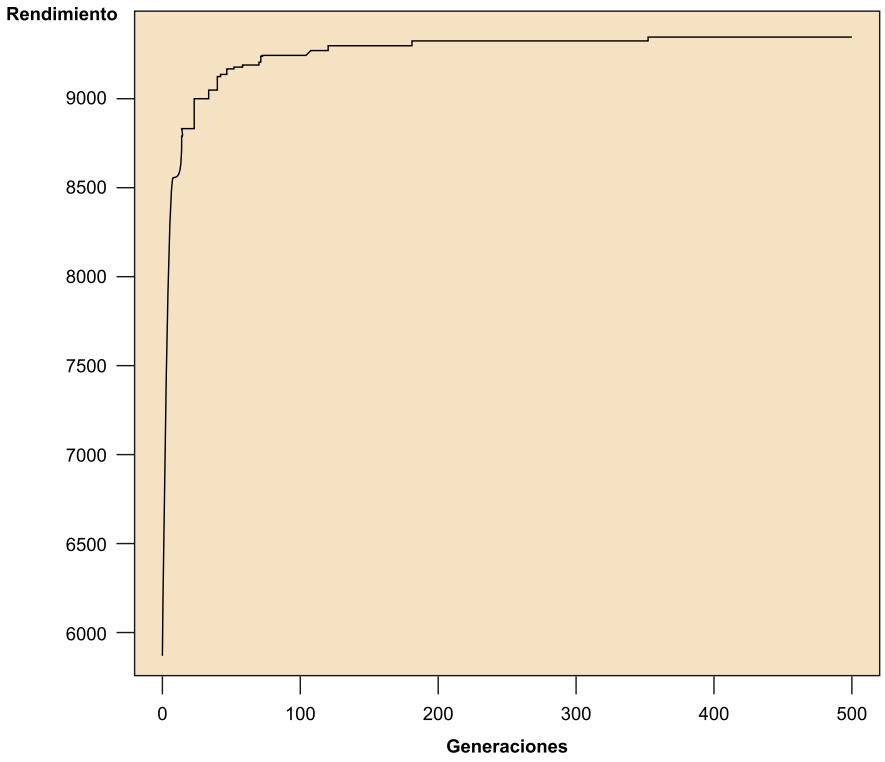
2) Se aplica la función objetivo a cada individuo. Los que obtengan mejores resultados serán los mejor adaptados.

3) Los individuos mejor adaptados se **seleccionan** para reproducirse mediante **cruzamiento**, es decir, combinación de sus genes. Esto da lugar a una nueva **generación** de individuos.

4) Para aumentar la diversidad de la población y así explorar nuevas posibilidades, los nuevos individuos pueden sufrir una **mutación**, esto es, un cambio aleatorio en algunos de sus genes.

5) Volver a 2 hasta que se alcance el número de generaciones (equivalente a iteraciones) deseado.

En principio, cuantas más generaciones se prueben, mejores soluciones se obtendrán, pero a cambio de un coste computacional mayor. Para saber cuándo conviene detenerse interesa observar la función objetivo y ver a partir de qué momento se estanca, es decir, no hay mejora aunque se sigan generando nuevos individuos.



Función objetivo (rendimiento de un sistema) en función del número de generaciones. Obsérvese el rápido crecimiento inicial y el relativo estancamiento a partir de doscientas generaciones.

Algunas de las operaciones mencionadas en el esquema necesitan ser concretadas. La **selección** de individuos hace referencia a qué individuos se eligen para que se reproduzcan y den lugar a una nueva generación. En principio, podrían cogerse los mejores individuos, pero eso haría que en unas cuantas

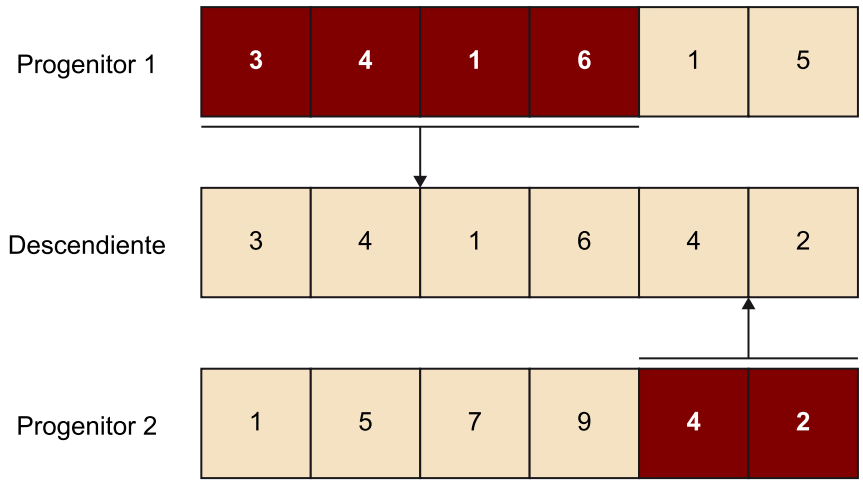
iteraciones todos los individuos fuesen descendientes de unos pocos progenitores y se perdería la diversidad de soluciones, con lo que se empobrecería el rendimiento del algoritmo (¡aquí también es importante la biodiversidad!).

Una alternativa muy utilizada es la **selección por torneo**: se seleccionan dos individuos al azar y se elige el mejor adaptado (mejor valor en la función objetivo); a continuación se selecciona otro individuo al azar y se compara con el vencedor del paso anterior, quedando el mejor adaptado de los dos; este proceso se repite un número de veces (unas tres) y finalmente queda seleccionado el individuo que vence en la última ronda. Todo este proceso se repite tantas veces como sea necesario (en general, si necesitamos generar n nuevos individuos, serán necesarios $2n$ progenitores).

Precisamente el **cruzamiento** de individuos se refiere a cómo se combinan los genes de dos individuos (pueden ser más pero no es habitual) para dar lugar a un descendiente que formará parte de la población de la siguiente generación. En general, se persigue mezclar los atributos (genes) de los dos individuos para obtener un nuevo individuo que, con un poco de suerte, reúna lo mejor de los dos progenitores. Hay varias estrategias de cruzamiento, aunque las más habituales son las siguientes.

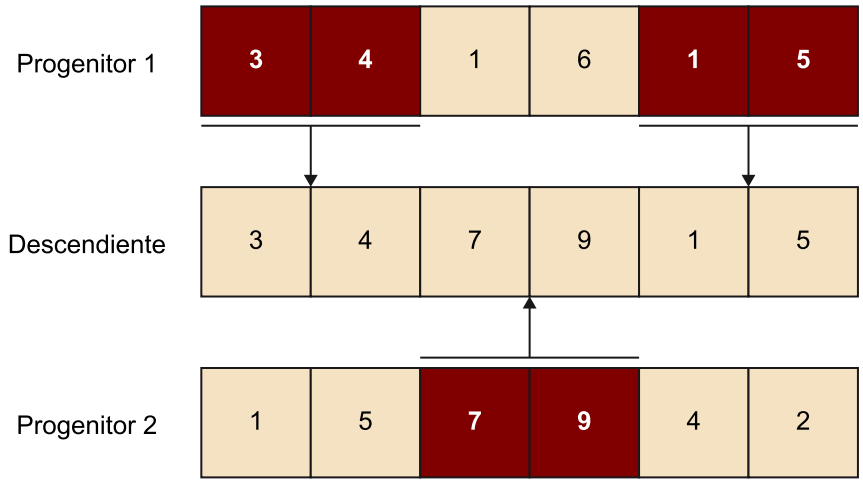
a) Cruzamiento **simple**: se elige un punto del genoma al azar y se cortan por ese punto los genomas de los progenitores, tomando de uno de ellos los atributos hasta el punto de corte y del otro los atributos a partir del punto de corte. Es útil cuando no hay un gran número de atributos.

Ejemplo de cruzamiento simple



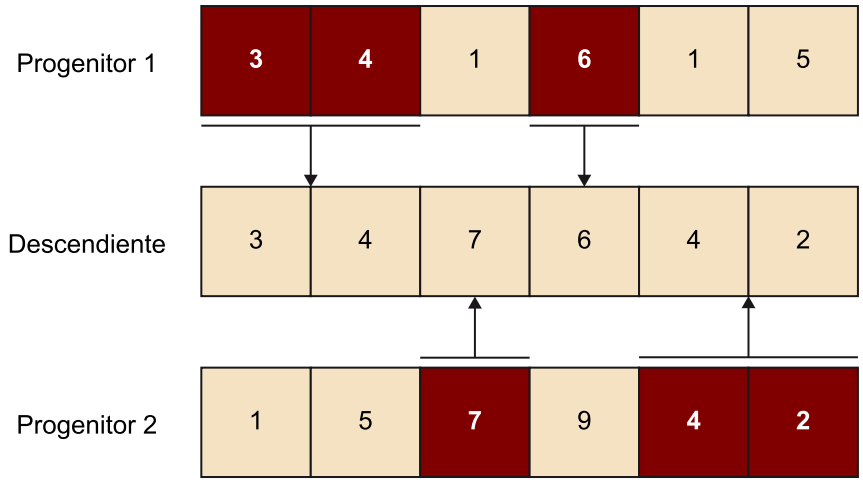
b) El cruzamiento **doble** funciona de manera parecida al simple pero en este caso se eligen dos puntos de corte, y se toman los atributos entre los dos puntos de corte de un progenitor y los restantes del otro. Es útil si hay un gran número de atributos, para lograr más variedad.

Ejemplo de cruzamiento doble



c) Finalmente, en el cruzamiento **uniforme** se toma cada uno de los atributos aleatoriamente de uno u otro de los progenitores. Es útil cuando no hay *localidad* en los atributos, es decir, no es importante mantener fragmentos seguidos de genoma porque no hay ninguna estructura en el vector de atributos.

Ejemplo de cruzamiento uniforme



Una limitación de utilizar solamente el cruzamiento para generar nuevos individuos es que en todo momento se están recombinando valores ya existentes en el acervo genético de la población, pero no se introducen valores nuevos que permitan explorar otras posibilidades. Para superar esta limitación, se define el tercer tipo de estrategia, llamada **mutación**, en la que algunos atributos se modifican aleatoriamente para probar valores nuevos y explorar nuevas zonas del espacio de soluciones.

Las mutaciones suelen ser alteraciones aleatorias cuya forma concreta depende del tipo de datos de los atributos. Los más habituales son estos:

- Atributos binarios: la mutación cambia al otro valor posible (de 0 a 1 y viceversa). Por ejemplo, volar o no volar.

- Atributos categóricos: se elige un nuevo valor aleatoriamente de entre los valores posibles (por ejemplo, el color de una unidad de entre una paleta).

- Atributos numéricos: se suma una cantidad aleatoria al valor actual; si por ejemplo la masa de un objeto es 1.500 kg, se añade o resta una cantidad al azar.

Al diseñar el algoritmo genético se suele ajustar la mutación de acuerdo con dos parámetros:

- Probabilidad de mutación individual: probabilidad de que un individuo sufra mutaciones. Suele ser baja (<10 %), pues de lo contrario la lógica del algoritmo genético se pierde.

- Probabilidad de mutación de atributos: dado un individuo que va a sufrir mutaciones, probabilidad de cada uno de sus atributos de ser mutado. También debería ser baja (<10 %), o de lo contrario el algoritmo genético se acaba convirtiendo en una exploración aleatoria.

Algoritmos genéticos

Se denominan «algoritmos genéticos», en plural, porque hay muchas formas de concretarlos para resolver un problema (codificación de los individuos, diseño de la función objetivo, tamaño de la población, número de generaciones, estrategias de selección, cruzamiento y mutación, etc.), así que se consideran una familia de algoritmos, no un único algoritmo que siempre es igual.

Vamos a utilizar los algoritmos genéticos para balancear las características de diez coches en un juego de carreras tipo Fórmula 1. Cada coche tiene cuatro atributos (velocidad máxima, aceleración, frenado, agarre), por lo que una solución del problema es un vector con valores para los cuatro atributos de los diez coches, es decir, un vector de cuarenta elementos de tipo real.

Aunque no desarrollaremos el ejemplo completo (falta añadir el juego en sí y la IA que conduzca los coches), sí veremos la parte de ajuste de parámetros. Los elementos que definirán el algoritmo genético para este problema serán:

- Individuo: vector de cuarenta atributos reales.
- Población: al menos cuarenta individuos, mejor si pueden ser cien.
- Número de generaciones: conviene probar con unas cien e ir controlando qué ocurre con la función objetivo.
- Selección: por torneo en tres pasos.
- Cruzamiento: puede ser simple o doble. De entrada, no interesa el uniforme porque cada bloque de cuatro atributos tiene un sentido (pertenece al mismo coche).
- Mutación: sumar/restar una cantidad aleatoria al atributo, controlando que se mantenga en el rango permitido (habrá un mínimo y un máximo para cada tipo de atributo).
- Función objetivo: ¿cómo valoramos qué combinación de coches es mejor? Hay muchas formas de hacerlo, aquí van dos propuestas:
  - Analítica: diseñar una función que asigne una puntuación a un coche en función de sus atributos; la función objetivo puede ser la desviación estándar de esa función: si es cero, es que todos los coches puntúan igual. Problema: puede ser difícil diseñar una función así y que sea realista.
  - Experimental: si ya se dispone del juego y de la IA de conducción, se puede ejecutar el juego con cada configuración (individuo) y analizar cómo de igualados llegan los coches. Mejor si se puede ejecutar varias veces para cada individuo y

se comprueba que en promedio quedan bastante igualados. La función objetivo sería la desviación estándar en las posiciones/tiempos obtenidos por los coches.

Biblioteca AForge.NET

Es posible programar el algoritmo genético completo, ya que desde el punto de vista de la programación no son muy complejos. Sin embargo, existen bibliotecas que los contienen (y ya están probados y ofrecen diferentes opciones); en C# los podemos encontrar en la biblioteca AForge.NET.

Hay que tener en cuenta, en cualquier caso, que los componentes de algoritmos genéticos de AForge.NET no están tan desarrollados como en otras bibliotecas, especialmente por lo que se refiere a la documentación y a la flexibilidad para personalizar su funcionamiento. Si se necesita un mejor control sobre el algoritmo se recomienda, por ejemplo, la biblioteca DEAP (Distributed Evolutionary Algorithms in Python), que como es evidente está en Python.

La instalación de AForge.Net en Unity no es sencilla; se recomienda utilizar la adaptación ai4unity e instalar los *assets* Core, Genetic y Math de la misma forma que se explica en la sección 4.2.1. Además, hay que añadir un fichero llamado «gmcs.rsp» a la carpeta *Assets* que simplemente contenga el texto «-unsafe» (sin las comillas). Algunas características de AForge no pueden utilizarse en Unity a causa de que utiliza una versión antigua de C# (2.0).

Webs recomendadas

La página oficial de la biblioteca AForge.NET es: http://www.aforgenet.com/.

Y su página sobre algoritmos genéticos es: http://www.aforgenet.com/framework/features/genetic_algorithms.html.

AI4unity puede obtenerse en:

https://github.com/davidgutierrezpalma/ai4unity.

Volviendo al ejemplo del ajuste de características de los coches de Fórmula 1, el código siguiente utiliza un algoritmo genético para encontrar un ajuste equilibrado. Para poder dar un ejemplo completo, tomaremos una función objetivo sencilla. En primer lugar, la puntuación de un coche será la media de sus cuatro parámetros; la función objetivo será la desviación estándar entre las puntuaciones de los coches de una solución dada, y el objetivo será reducirla a cero.

Las cuatro características tienen rangos diferentes: la velocidad máxima 200-300, la aceleración 5-10, la frenada 5-20 y el agarre 0-1. Sin embargo, para simplificar el algoritmo se manejarán cuatro valores reales cualesquiera para estas características, y después se adaptarán a los rangos reales en el juego.

Evidentemente, esa función objetivo no es muy realista y probablemente la solución obtenida no muestre una gran jugabilidad; como se ha comentado antes, lo mejor sería utilizar un juego real y hacer competir a los coches para medir su rendimiento en la carrera.

El siguiente código se debe asociar a un objeto vacío (*Empty*) de Unity, creando un *script* como se ha descrito en el apartado 4.2.1:

Observación

La traducción de un parámetro en un rango 0...1 a otro rango es sencilla; sea *min* el valor mínimo del rango deseado y *max* el máximo; si el algoritmo nos da un valor *x*, ese valor traducido al rango *min…max* viene dado por:

$$y = x\,(max\text{-}min) + min$$

Podéis comprobar que si $x = 0$, entonces $y = min$; y que si $x = 1$, $y = max$.

```
using AForge.Genetic;

using AForge.Math;
using System;         // Para Math.Pow
using UnityEngine;


// Clase para definir la función objetivo (calcular la desviación
// estándar de las medias de cada coche)
public class DiferenciasCoches : IFitnessFunction
{
   // Constructor (podría recibir datos necesarios para calcular
   // la calidad de las soluciones)
   public DiferenciasCoches()
   {
   }
```

```
   // Para visualizar los resultados
   public object Translate(IChromosome cromosoma)
   {
      return cromosoma.ToString();
   }

   // Calcular la función objetivo
   public double Evaluate(IChromosome cromosoma)
   {
      // Como AForge considera que un individuo con mayor función
      // objetivo es mejor (es decir, solo maximiza) tenemos que
      // invertir el resultado de la desv. estándar.
      // Sumamos 1 al denominador para evitar divisiones por cero.
      return 1.0 / (1.0 + evalua(cromosoma));
   }

   // Función que calcula la desviación estándar de un cromosoma
   // interpretando que hay n coches con 4 atributos cada uno
   public double evalua(IChromosome cromosoma)
   {
      // Para acceder a los valores propios del array de reales
      // Se toman los valores de 4 en 4 (cada coche) y se calcula la
      // media de cada grupo
      DoubleArrayChromosome miCrom = (DoubleArrayChromosome)cromosoma;
      int numCoches = miCrom.Length / 4;
      double[] medias = new double[numCoches];
      for (int i = 0; i < numCoches; i++)
      {
         // Calcula la media de las 4 componentes de cada coche
         medias[i] = media(miCrom.Value[i * 4],
                     miCrom.Value[i * 4 + 1],
                     miCrom.Value[i * 4 + 2],
                     miCrom.Value[i * 4 + 3]);
      }
      return desvEstandar(medias);
   }

   // Calcula la desviación estándar de las valoraciones
   // de los coches
   public double desvEstandar(double [] valores)
   {
      double suma = 0.0;
      foreach (double x in valores)
         suma += x;
      double media = suma / valores.Length;

      double suma2 = 0.0;
      foreach (double x in valores)
         suma2 += Math.Pow(x - media, 2);
      return Math.Sqrt(suma2 / (valores.Length-1)) ;
   }

   // Calcula la media de los cuatro valores
   public double media(double veloc, double acel, double freno,
   double agarre)
   {
      return (veloc + acel + freno + agarre) / 4.0;
   }
}



public class AlgoritmoGenetico : MonoBehaviour {

// Use this for initialization
void Start () {
      // Función objetivo (calcula la calidad de una solución)
      DiferenciasCoches difCoches = new DiferenciasCoches();

      // Generador de números aleatorios para crear la población,
```

```
        // mutaciones, etc.
        AForge.Math.Random.UniformOneGenerator generador = new
        AForge.Math.Random.UniformOneGenerator();

        // Número de parámetros en el problema (10 coches x 4
        parámetros/coche)
        int numParams = 40;

        // Hay que generar un vector base de 40 reales
        double[] valores = new double[numParams];
        Debug.Log("A================================");
        // Parámetros del constructor de Population
        // -tamaño de la población (número de individuos)
        // -tamaño de un individuo (número de atributos, 4 x 10)
        // -difCoches -> objeto que contiene la función objetivo
        // -RouletteWheelSelection -> función de selección (al azar
        // proporcional a la calidad de cada solución)
        Population population = new Population(50,
              new DoubleArrayChromosome(generador, generador,
              generador, valores),
              difCoches,
              new RouletteWheelSelection());

        // Hacer evolucionar a la población durante un número
        // de generaciones; mostrar resultados cada 10
        for(int i=0; i<1000; i++)
        {
           population.RunEpoch();
           if(i % 10 == 0)
           {
              Debug.Log(i);
              Debug.Log(population.BestChromosome.ToString());
              Debug.Log(difCoches.evalua(population.BestChromosome));
           }
        }

        Debug.Log("Valor final");
        Debug.Log(population.BestChromosome.ToString());
        Debug.Log(difCoches.evalua(population.BestChromosome));

    }

    // Update is called once per frame
    void Update () {

    }
}
```

En la salida por la consola de depuración se puede observar cómo la desviación estándar va descendiendo a medida que avanzan las generaciones, lo que indica que los diferentes coches están balanceados entre ellos y, por tanto, un jugador puede elegir cualquiera de ellos y ganar en el juego.

Resumiendo, los algoritmos genéticos son una herramienta muy útil para resolver problemas de optimización como por ejemplo los de ajuste de parámetros, pero tienen las desventajas de requerir bastante memoria y tiempo de computación, por lo que en general se utilizan en procesos *offline* o en momentos en los que no interfieran con la ejecución del juego (mientras se muestran puntuaciones, etc.)

5. Técnicas avanzadas de IA

En este apartado estudiaremos las últimas técnicas de IA que se están investigando ahora mismo y que prometen revolucionar lo que conocemos por IA, tanto en juegos como en otros ámbitos. A partir de 2015 se han sucedido logros impensables anteriormente, entre los que destacamos:

- AlphaGo: un sistema que ha derrotado al experto en Go (juego entre ajedrez y damas chinas, más complejo que el ajedrez) Lee Se-dol, considerado el segundo mejor jugador del mundo.

- DQN: un sistema que, simplemente viendo las imágenes de un simulador de consola Atari 2600, es capaz de aprender a jugar sin ninguna instrucción previa. Incluso juega mejor que humanos expertos, y es capaz de encontrar trucos para mejorar el juego.

- Un sistema que juega al juego de disparos en primera persona Doom, a partir de las imágenes, y es capaz de ganar a jugadores humanos; de hecho, está especializado en cazar a jugadores humanos (lo que ha suscitado una cierta polémica).

- Un piloto automático de cazas de combate que ha derrotado a pilotos expertos en un simulador de vuelo militar.

- Fuera del ámbito estricto de los videojuegos: sistemas de conducción automática, reconocimiento de voz en móviles, reconocimiento de rostros en redes sociales, mejoras en la traducción automática, etc.

Todos estos avances tienen en común el método en el que están basados, conocido como aprendizaje profundo (*deep learning*), que describiremos en este apartado. En primer lugar veremos sus elementos constituyentes, las redes neuronales, y a continuación veremos cuál es la arquitectura de los sistemas de aprendizaje profundo y qué opciones existen.

5.1. Redes neuronales

El cerebro humano (y del resto de los animales) está formado por diferentes tipos de células. Las neuronas son las células que, estableciendo conexiones entre sí y enviando mensajes por esas conexiones, dan lugar al comportamiento emergente que conocemos por inteligencia.

La meta de las **redes neuronales** es crear una suerte de neuronas computacionales para intentar reproducir la inteligencia humana en un ordenador, si bien la analogía se queda en que existen unos elementos llamados **unidades** –el

equivalente a las neuronas– que se conectan unas a otras y se pasan mensajes mediante **conexiones**; más allá de esto, no hay que pensar que las redes neuronales se parezcan a las neuronas del cerebro.

5.1.1. Componentes de una red neuronal

La arquitectura básica de una red neuronal se compone de los siguientes elementos:

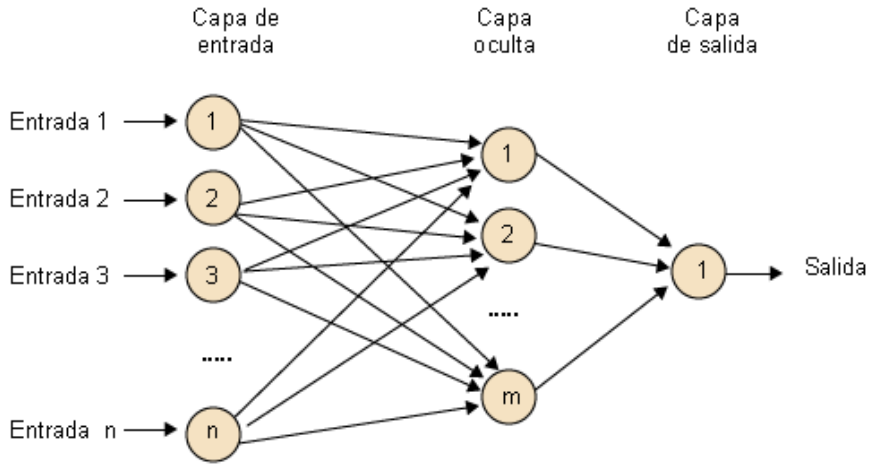
- Una capa de unidades (el equivalente a las neuronas) de **entrada**, que reciben un conjunto de valores del exterior (píxeles de una imagen, posición de los jugadores de un partido de baloncesto, etc.)

- Una capa de unidades de **salida**, compuesta por una o más unidades que producen una salida al exterior, que sería el «resultado» de la ejecución de la red, por ejemplo un valor indicando si el coche debe frenar, si se debe desplazar el muñeco de un juego a izquierda o a derecha, etc.

- Una (o más) capas de unidades **ocultas**, que reciben conexiones de la capa de entrada y se conectan a las unidades de salida.

- El conjunto de **conexiones** entre capas. En la arquitectura más básica, llamada red neuronal prealimentada (*feed forward neural network*) o perceptrón multicapa (*multilayer perceptron*), las conexiones siempre van de las unidades de una capa a la siguiente, es decir, de la capa de entrada a la oculta y de esta a la capa de salida.

Arquitectura básica de una red neuronal de tipo «prealimentada»



El funcionamiento general de las redes neuronales, sin entrar en detalles técnicos, es la **propagación** de señales de una capa a la siguiente. Veamos paso a paso lo que ocurre ante un conjunto de entradas determinado, en el que cada entrada a la red tendrá un valor diferente:

1) Las unidades de la capa de entrada reaccionarán **activando** su salida o no en función de la entrada recibida.

2) Como la salida de las unidades de la capa de entrada está conectada a las entradas de las unidades de la capa oculta, estas reciben un conjunto de señales de entrada (para ellas) frente a las que también deben reaccionar. Para ello, cada unidad de la capa oculta tiene un **vector de pesos**, un valor por cada conexión que le llega, que combina con la entrada recibida y hace que a su vez se active o no la salida de cada unidad oculta.

3) Las unidades de la capa de salida reciben señales de las unidades ocultas y aplican su propio vector de pesos a esas señales para decidir si activan su salida o no, en lo que será el resultado final de la ejecución de la red para el ejemplo recibido.

Redes neuronales

Las redes neuronales son un formalismo que permite aprender una función multivariante (con más de una variable de entrada) y multivaluada (con más de una variable de salida). Con suficientes unidades y capas ocultas es posible aprender cualquier función.

Las unidades de una capa pueden tener conexiones hacia todas las unidades de la siguiente (entonces se habla de capas totalmente conectadas o densas) o, por el contrario, cada unidad puede estar conectada solo a algunas de las unidades de la siguiente capa.

Nota

En este apartado se van a explicar las opciones más utilizadas, aunque existen muchas alternativas que omitiremos, ya que un curso completo sobre redes neuronales requeriría un libro completo (o más).

5.1.2. Funciones de activación

Existen numerosas alternativas para cada uno de los elementos que componen una red neuronal. Por ejemplo, las señales de activación generadas por las unidades pueden ser **binarias** (planteamiento original) o **reales** (uso actual).

El **vector de pesos** de cada unidad suele ser un vector de números reales W, tantos como entradas tenga la unidad, y realmente esto es lo que sería la «memoria» de la red; además, se suele incluir un valor escalar b (por *bias*, es decir, un sesgo o valor de desplazamiento). Así, si el vector de entradas a una unidad es X, se realiza esta operación:

$$z = W^T X + b \qquad (1)$$

(es decir, se multiplica cada entrada por su peso asociado, se suman esos productos y al final se suma b). El valor z así obtenido (un número real) se utiliza entonces como entrada para una función de **activación**, que es la que decide cuál es la salida.

La función de activación más utilizada en unidades ocultas es la llamada *ReLU* (de *rectified linear unit*), que a partir del valor z calculado anteriormente aplica:

$$g(z) = \max\{0, z\}$$

Es decir, la salida es 0 si la entrada es negativa, y es igual a z si z es positiva.

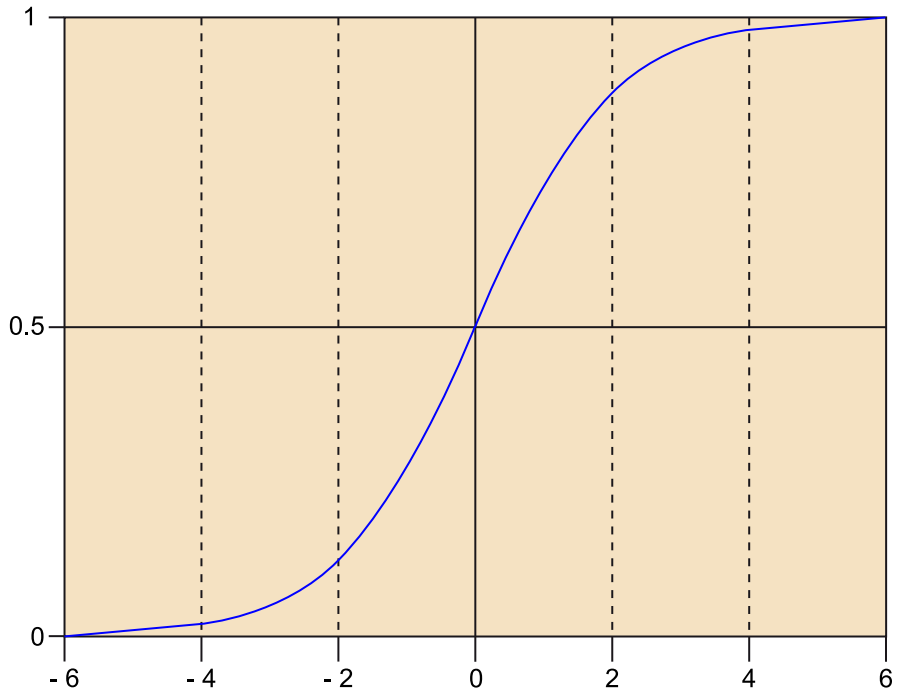
En el caso de unidades de salida, se utilizan otras funciones de activación que dependen del tipo de salida deseada:

- Valor **real** (por ejemplo, cuánto pisar el acelerador): unidades lineales, es decir el valor z calculado según la expresión (1).

- Valores **binarios** (por ejemplo, decidir si saltar o no): unidades **sigmoides**, usando la función logística sigmoide que tiene este aspecto:

Observación

Es mejor que la salida de las unidades *softmax* no sea exactamente 1 en una y 0 en las demás para facilitar el entrenamiento.



- Valores **multiclase** (por ejemplo, al decidir qué tipo de unidad producir en un juego de estrategia): función **softmax**, en la que la salida es un vector con tantos valores como clases posibles se puedan elegir (por ejemplo, un valor para elegir obrero, otro para soldado, otro para arquero, etc.); la salida que corresponde al valor elegido tendrá un valor cercano a 1 y las demás tendrán un valor cercano a 0.

5.1.3. Entrenamiento de una red neuronal

Acabamos de ver cuáles son las operaciones que lleva a cabo una red neuronal en respuesta a un conjunto de entradas, pero hemos dado por supuestos algunos valores que aparecían como por arte de magia en las fórmulas, concretamente los vectores de pesos W de las unidades internas y de salida y los valores de sesgo b. Lógicamente, esos valores no aparecen solos, sino que primero la red los debe aprender. Pues en las redes neuronales pasan por dos fases: una primera fase de **entrenamiento** y otra de **ejecución**; hemos visto cómo operan

en la fase de ejecución para dar la salida correspondiente a una entrada, pero falta ver cómo entrenar la red para que aprenda los pesos y sesgos adecuados si queremos que reaccione correctamente.

En la fase de entrenamiento, a la red se le da un conjunto de ejemplos de entrenamiento que incluyen la salida esperada para cada uno de ellos.

> Estamos desarrollando un juego para móvil o tableta y queremos que reconozca gestos del usuario en la pantalla. El conjunto de entrenamiento sería un conjunto de gestos (la captura de los píxeles activados en el gesto) y el código del gesto correspondiente. Para que la red pueda aprender, será necesario proporcionarle muchos ejemplos de cada clase de gesto.

Al iniciar el entrenamiento, la red se inicializa con valores aleatorios (se recomiendan valores pequeños, en torno a 0,1). A continuación, se le van dando los ejemplos y se analiza la diferencia entre la salida generada por la red y la salida esperada; entonces se vuelve hacia atrás (de la capa de salida hacia la capa de entrada), ajustando los pesos y los sesgos para intentar minimizar la diferencia entre la salida obtenida y la esperada. Este proceso se repite para todos los ejemplos de entrenamiento, hasta que se consigue minimizar la diferencia. Esta vuelta hacia atrás para ir ajustando los pesos se denomina **retropropagación** (*backpropagation*).

El algoritmo de optimización que se suele utilizar para realizar este ajuste de pesos y sesgos es el **descenso de gradientes estocástico** (*stochastic gradient descent*), que consiste, simplificando, en ir probando valores aleatorios (de pesos y sesgos) alrededor del valor actual y tomar el que menor error muestre.

Al final de la fase de entrenamiento la red será capaz de producir una salida acorde con la entrada que reciba según los ejemplos que ha recibido.

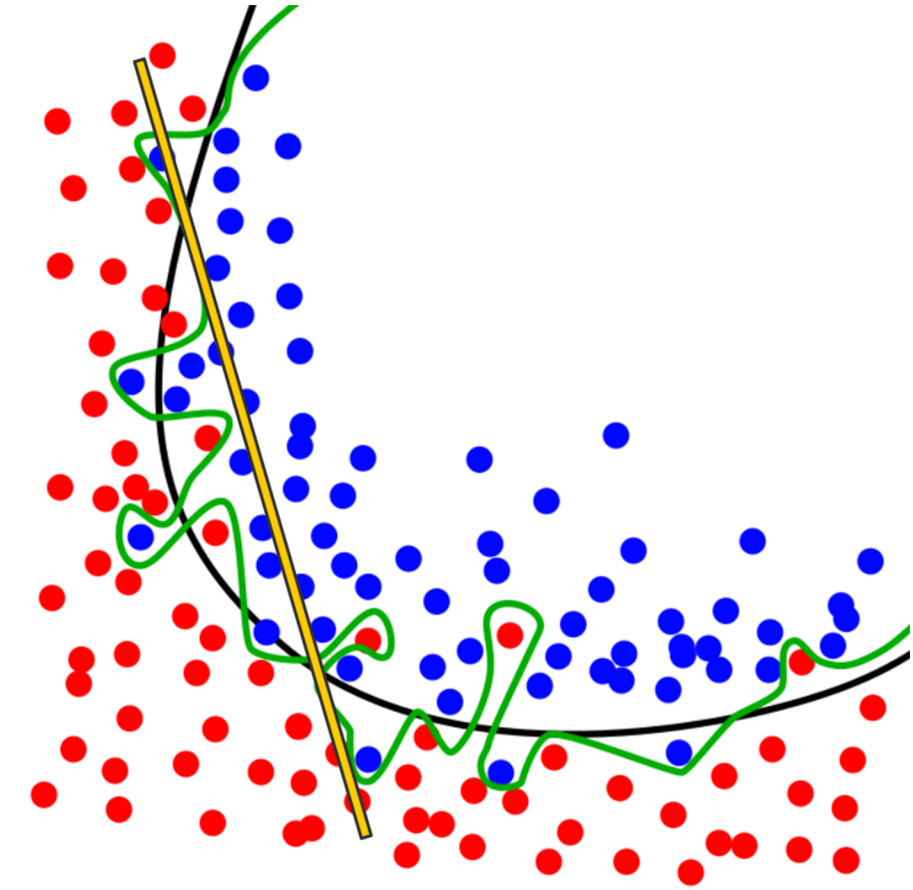
5.1.4. Problemas de aprendizaje

Tanto en las redes neuronales como en otros métodos de aprendizaje nos podemos encontrar con estos dos problemas:

- **Infraajuste** (*underfitting*): si la función que queremos aprender es muy compleja y nuestra red neuronal no tiene suficientes unidades o capas ocultas, no podrá aprender bien la función (no tendrá suficiente «memoria», lo que se denomina la **capacidad** del modelo).

- **Sobreajuste** (*overfitting*): por el contrario, si nuestra red tiene demasiada capacidad (unidades y capas ocultas), la red se aprenderá perfectamente todos los ejemplos de entrenamiento pero no será capaz de generalizar a ejemplos ligeramente diferentes cuando se quiera poner en uso.

Problemas de aprendizaje



Problemas de aprendizaje: infraajuste (línea amarilla), sobreajuste (línea verde), aprendizaje correcto (línea negra). Fuente: Adaptado de CC Wikimedia Commons- By Chabacano - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3610704.

Conclusión: la capacidad de la red ha de ser adecuada a la función que se desea aprender. ¿Cómo se sabe cuál es la capacidad adecuada? Utilizando un protocolo de **entrenamiento** + **validación**.

1) Los datos de entrenamiento se dividen en dos grupos, entrenamiento y validación.

2) La red se entrena con los datos de entrenamiento y se monitoriza el error de entrenamiento (diferencia entre la salida esperada y la obtenida). Mientras ese error se reduzca, es que la red está mejorando el aprendizaje (estamos evitando el infraajuste), conviene aumentar la capacidad de la red.

3) En paralelo se monitoriza el error que produciría la red si se probara con los datos de validación, que equivale a probar los datos en una situación real, con ejemplos que no había visto antes.

4) Si el error de validación empieza a crecer, quiere decir que se está produciendo *sobreajuste* porque la red se está aprendiendo con tanto detalle los ejemplos de entrenamiento que es incapaz de generalizar. Es el momento de dejar de ampliar la red.

Existen otras estrategias para reducir el sobreajuste, en lo que se conoce como **regularización** del modelo para hacerlo más general. En redes neuronales, la estrategia de regularización más utilizada se llama ***dropout*** y, sorprendentemente, consiste en apagar aleatoriamente algunas unidades ocultas durante el entrenamiento. La idea es que así la red será capaz de adaptarse con más facilidad a cambios en los valores de entrada, en lugar de seguir al pie de la letra los ejemplos de entrenamiento, que es en el fondo en lo que consiste el sobreajuste.

5.1.5. Perspectiva histórica y futura

Las redes neuronales son uno de los primeros métodos de IA; de hecho el perceptrón simple (sin capas ocultas) fue propuesto en 1958 por Frank Rosenblatt. Durante su historia han sufrido muchos altibajos: cuando se propusieron parecía que rápidamente iban a conseguir emular la inteligencia humana; al cabo de unos años cayeron en desuso; en los años ochenta del siglo XX volvieron a resurgir con el desarrollo del algoritmo de retropropagación; pero de nuevo decayeron ya que no se produjeron más avances; pero especialmente desde el año 2010 han vuelto a surgir con mucha fuerza gracias a diferentes factores que permiten crear redes con más capas ocultas, más neuronas, mayor estabilidad numérica y que pueden entrenarse y ejecutarse a gran velocidad gracias a los avances en el hardware, especialmente los procesadores gráficos. En el siguiente apartado veremos las características de este renacimiento de las redes neuronales.

5.2. Aprendizaje profundo

Se denomina **aprendizaje profundo** (*deep learning*, DL a partir de ahora) a las diferentes arquitecturas de redes neuronales cuya característica principal es que contienen un gran número de capas ocultas; la profundidad se refiere al número de capas de la red.

El DL ha sido posible por diferentes factores científicos y tecnológicos que han surgido desde 2004 aproximadamente, y entre los que destacan:

- El uso de funciones de activación como ReLU, que facilitan la optimización de las redes neuronales, en contraste con funciones que se utilizaban anteriormente y provocaban problemas numéricos o daban una respuesta pobre.

- La aplicación del descenso de gradientes estocástico como algoritmo de optimización en el entrenamiento de las redes.

- El gran desarrollo de hardware capaz de operar con vectores, concretamente los procesadores gráficos, que pueden dividir por diez o más el tiempo necesario para llevar a cabo un entrenamiento.

- La abundancia de conjuntos de datos de gran tamaño, que resultan adecuados para este tipo de métodos, al contrario de lo que ocurre con otros métodos de aprendizaje automático que requieren crear matrices de *kernel* o similares, lo que no resulta viable cuando hay millones de ejemplos.

La ventaja de tener una red profunda estriba en que cada capa extrae características interesantes de la anterior, y así poco a poco la red va obteniendo cada vez características más sofisticadas sin necesidad de programarla expresamente para ello. Por ejemplo, de una imagen se puede empezar extrayendo los contornos de las figuras, en otro nivel identificar formas básicas, luego unir varias de esas formas para formar caras, etc. De esa manera la red es útil para resolver tareas con nuevos datos.

Por el contrario, tener redes con muchas unidades pero pocas capas (es decir, capas muy anchas) solo sirve para que la red aprenda diferentes combinaciones de los datos de entrada pero sin capacidad de generalizar, es decir, tienden más al sobreajuste.

5.2.1. Arquitecturas

Dentro del DL hay diferentes tipos de arquitecturas, cada una orientada a resolver determinados tipos de problemas:

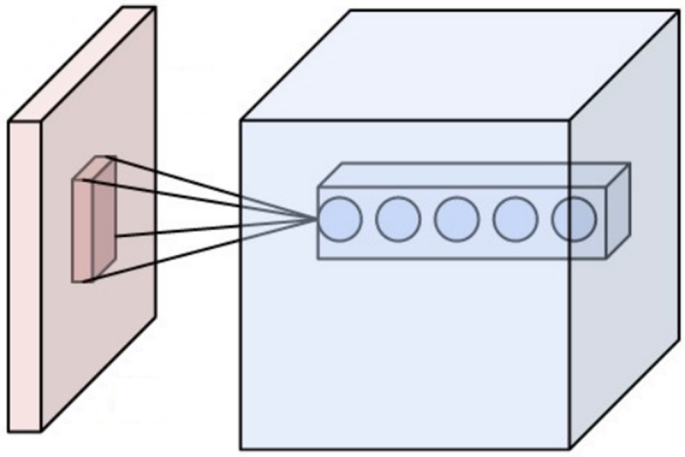
- Redes **prealimentadas** (*feed-forward neural networks*, FNN): se trata de la arquitectura básica, descrita en el apartado anterior. Su característica es que las conexiones siempre van desde la entrada hacia la salida, no hay conexiones entre unidades de la misma capa hacia capas anteriores. Se utilizan en problemas de clasificación de datos, de generación de valores reales, etc.

- Redes neuronales **recurrentes** (*recurrent neural networks*, RNN): en estas redes sí hay algunas conexiones hacia capas anteriores (hacia la entrada) con el objetivo de que la red recuerde ejemplos anteriores y los tenga en cuenta en el aprendizaje. Se utilizan para datos que tienen un sentido temporal, por ejemplo en procesamiento de señales o de lenguaje escrito y hablado.

- Redes neuronales **convolucionales** (*convolutional neural networks*, CNN): en estas redes las unidades de la capa de entrada no están conectadas a todas las unidades de la siguiente capa, sino solo a unas cuantas. La idea es aprovechar la localidad de las entradas, por ejemplo de los píxeles de una imagen: solo los píxeles adyacentes tienen relación entre sí, y por eso las entradas correspondientes a esos píxeles se conectan solo a las unidades que les corresponden. Esto tiene la ventaja de reducir el coste computacional (los vectores de pesos son mucho más pequeños) y permitir que la red detecte características de detalle (por ejemplo, contornos en una imagen) sin necesidad de procesar la imagen entera. Este tipo de redes se utilizan

para procesar imágenes o cualquier tipo de dato en el que haya localidad en sus atributos.

Red neuronal convolucional



Red neuronal convolucional: cada zona de la red solo procesa una parte de la imagen de entrada. Fuente: By Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45659236.

Por otra parte, nos encontramos con los **autocodificadores** (*autoencoders*), que pueden aplicarse a cualquiera de las tres arquitecturas anteriores, y que consisten en crear una red que tenga el mismo número de salidas que de entradas, y obligarla a que las salidas sean iguales a las entradas (por ejemplo, si entra la foto de un gato, que salga la misma foto). ¿Qué sentido tiene esto? Pues que alguna(s) capa(s) ocultas tienen menos unidades que las entradas, lo que obliga a la red a codificar la información con menos datos de modo que luego se pueda reconstruir la original. De esta forma se consigue generar una reducción de la dimensionalidad en la imagen.

Una aplicación del DL de gran importancia para los videojuegos es el llamado Q-Learning, en el que se utiliza una red DL para aprender la matriz de estados y acciones (*Q*) de problemas de aprendizaje por refuerzo (ver apartado 4). Como vimos allí, en casi cualquier videojuego el número de estados posible es inmenso, y si se multiplica por el número de acciones posibles resulta una matriz de posibles acciones para un agente que no se puede almacenar. Pero es posible usar una red DL para que aprenda una aproximación de la matriz *Q*, haciendo que asocie cada entrada (estado del juego) con una acción y una recompensa. Eso es lo que hace, por ejemplo, el algoritmo DQN (Deep Q-Network), el que aprendió a jugar a los videojuegos de Atari.

5.2.2. Librerías para aprendizaje profundo

Es posible utilizar DL en videojuegos, si bien la fase de entrenamiento suele ser muy costosa computacionalmente (y además por velocidad se recomienda ejecutarla en el procesador gráfico, algo complicado durante el juego). Por ese motivo en general se entrenarán las redes *offline* y después se usarán dentro del juego.

Dado que hay bastantes matemáticas y programación vectorial y matricial detrás de los algoritmos de DL, es conveniente utilizar alguna de las librerías existentes.

Lamentablemente en Unity y C# no hay librerías muy desarrolladas:

- En **Accord.Net** existen algunas funciones para realizar operaciones básicas, pero en un estado bastante primitivo.

- En **AForge.Net** hay más desarrollo hecho, si bien sigue siendo bastante limitado en opciones.

Las librerías de DL más desarrolladas actualmente son las siguientes:

- **Theano**: librería en Python que ofrece todo el soporte matemático necesario para DL, con todo tipo de objetos y con código optimizado especialmente para CUDA (lenguaje de programación del procesador gráfico creado por NVidia). No es recomendable usarla directamente si no se tiene un conocimiento profundo de las matemáticas y algoritmos de DL.

- **Lasagne**: librería en Python que facilita enormemente la creación de redes DL sobre Theano; en Lasagne creamos capas de unidades y las conectamos, olvidándonos un poco de vectores y matrices y gradientes. Es una de las mejores opciones actualmente.

- **Keras**: como Lasagne, se trata de otra librería de alto nivel en Python, que en este caso puede funcionar tanto sobre Theano como sobre Tensorflow. Facilita mucho la creación de redes neuronales.

- **nolearn**: librería en Python que trabaja sobre Lasagne para facilitar aún más las cosas, aunque no compensa utilizarla salvo para ejemplos bastante sencillos, pues se pierde fácilmente el control sobre lo que se está haciendo.

- **Tensorflow** es la alternativa de Google a Theano, también en Python. Tiene algunos problemas de eficiencia con matrices grandes.

- **Caffe** es una librería de visión artificial en Python que solo soporta redes convolucionales.

- **Torch** es una librería en Lua. En su favor se puede decir que Facebook y Twitter la utilizan, entre otras compañías.

- **DL4J** es una librería de DL para Java. Sus promotores aducen que es la única librería preparada para el entorno empresarial (se considera que Python pertenece al entorno científico).

Como se ve, hay una gran variedad de opciones y con toda seguridad seguirán apareciendo nuevas librerías, teniendo en cuenta la importancia que está adquiriendo el DL.