Inteligencia artificial

Jordi Duch i Gavaldà
Heliodoro Tejedor Navarro

PID_00188528

UOC
Universitat Oberta de Catalunya

Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es

Índice

Introducción 5

Objetivos 6

1. Inteligencia Artificial en los videojuegos 7
1.1. Historia de la IA en los videojuegos 8
1.2. IA, *scripting* y lógica 10
1.3. La curva de dificultad 11
1.4. Uso de la IA 12

2. Técnicas de movimiento 15
2.1. Movimientos cíclicos y basados en patrones 15
2.2. Búsqueda de caminos 19
2.2.1. Recorrido en profundidad 20
2.2.2. Recorrido en amplitud 22
2.2.3. Dijkstra 23
2.2.4. A* 25
2.2.5. Otras variaciones de *pathfinding* 27
2.3. Movimientos complejos 28
2.3.1. Movimientos autónomos 29
2.3.2. Movimientos colectivos (*flocking*) 31
2.3.3. Movimientos programados 38

3. Toma de decisiones 39
3.1. El proceso de toma de decisiones 41
3.2. Técnicas de IA para tomar decisiones 41
3.2.1. Sistemas de reglas 42
3.2.2. Máquinas de estados finitos 45
3.2.3. Árboles de decisión 47
3.2.4. Lógica difusa 50
3.2.5. Redes bayesianas 51
3.2.6. Mapas de influencia 55
3.2.7. Árboles de comportamiento 57

4. Técnicas avanzadas de IA 60
4.1. Aprendizaje 61
4.1.1. Redes neuronales 61
4.2. Evolución 65
4.2.1. Algoritmos genéticos 66
4.3. Comportamientos colectivos 69
4.3.1. Sistemas multi-agente 69

Resumen... 71

Actividades... 73

Glosario... 74

Bibliografía... 75

Introducción

Un juego debe ofrecer retos al usuario. Para poder introducir estos retos en un videojuego son necesarias una serie de técnicas que se engloban dentro del estudio de la Inteligencia Artificial (IA). Estas técnicas deciden cuáles son las mejores opciones que pueden tomar los elementos del juego a partir de las condiciones del entorno que los rodea.

Será responsabilidad de los programadores llegar a un compromiso en la dificultad de los retos para que el jugador no considere el juego aburrido por su sencillez o por su complejidad. Además, la dificultad de estos retos ha de ir aumentando de una manera acorde a la evolución del jugador, para mantener la intensidad desde el principio hasta el final del juego.

Existe una gran multitud de técnicas de IA que se han desarrollado paralelamente en varios campos a lo largo de los años: algoritmos genéticos, sistemas expertos, búsquedas de caminos, redes neuronales, sistemas multiagente, lógica, etc. Las técnicas que deberemos utilizar para un juego dependen mucho del tipo de juego que estemos diseñando, de la importancia que le queramos dar a la IA dentro de nuestro juego, así como de la cantidad de recursos que tengamos disponibles para ella.

En este módulo vamos a introducir el concepto de Inteligencia Artificial, su historia y su relación con los principales módulos del sistema. Después analizaremos con detalle las principales técnicas que se utilizan en los videojuegos. Veremos algunos ejemplos prácticos del uso de estas técnicas en géneros concretos.

Hemos agrupado estas técnicas en cuatro grandes grupos, aunque algunas de ellas las podríamos clasificar en más de uno:

- Cómo movernos por el entorno.
- Cómo decidir qué hacer en cada momento.
- Cómo aprender del entorno y de los errores.
- Cómo trabajar en equipo.

Normalmente existe siempre un balanceo entre la precisión y el tiempo necesario para calcularla, así que dejaremos siempre abierta la elección de la tecnología más adecuada para cada caso particular.

Objetivos

En este módulo didáctico, presentamos al alumno los conocimientos que necesita para alcanzar los siguientes objetivos:

- Entender la relación existente entre la Inteligencia Artificial y los otros componentes del videojuego.

- Saber cómo mover los elementos en un mundo virtual, tanto a nivel individual como colectivo.

- Implementar sistemas que permitan decidir cuál es la mejor acción que debe tomarse en cada momento a partir de la información que tengamos disponible del sistema de una manera rápida.

- Conocer las últimas tendencias en tecnologías de IA aplicadas en los videojuegos y cómo se pueden utilizar estas tecnologías para mejorar la experiencia de juego del usuario.

1. Inteligencia Artificial en los videojuegos

Según la definición del diccionario de la Real Academia de la Lengua Española, la inteligencia artificial (o IA) es el "desarrollo y utilización de ordenadores con los que se intenta reproducir los procesos de inteligencia humana". Se trata de un campo de estudio muy extenso dentro de la informática actual, con aplicaciones en todo tipo de disciplinas como la medicina, el mundo militar o la economía.

En los videojuegos, la IA nos permite hacer que el jugador crea que los personajes que se encuentran en el mundo donde se desarrolla la acción tienen un comportamiento con un cierto grado de inteligencia (con más o menos acierto, dependiendo de la temática del juego). También se puede utilizar para ayudar en la toma de decisiones al usuario (indicar en un juego de fútbol a qué jugador se le puede pasar el balón). La necesidad de utilizar la IA en los videojuegos ha existido desde el principio, pero no ha sido hasta los últimos años cuando la industria se ha tomado muy seriamente la necesidad de incorporar una muy buena IA para ofrecer unos juegos más realistas y competitivos.

Ejemplo

Quizá el ejemplo más clásico de la IA aplicada a los videojuegos (y a los juegos en general) es el caso de los simuladores de ajedrez. El ajedrez es uno de los pocos juegos en los que no se ha conseguido todavía programar un sistema basado en IA que sea invencible. Aunque se hayan ganado varias partidas, la cantidad de recursos necesarios para calcular la IA necesaria para ganar a los mejores jugadores "humanos" está muy lejos de los recursos que tenemos en los procesadores personales de hoy en día.

No obstante, la IA de los juegos se diferencia en algunas cosas de la IA clásica. A no ser que la IA sea el punto más fuerte de nuestro juego, como por ejemplo si llevamos a cabo un juego de estrategia, en ocasiones hemos de conseguir un punto intermedio entre lo inteligentes que puedan ser nuestros jugadores virtuales y el tiempo que necesitemos para calcular su comportamiento. En la implementación de la IA para juegos en ocasiones se realizan ciertos trucos o trampas para que el sistema pueda parecer lo más "vivo" posible, pero siempre alcanzando un compromiso entre la calidad de las decisiones y el tiempo necesario para calcularlas.

Comportamientos inteligentes

Algunos puristas de la IA consideran que en los juegos no existen comportamientos inteligentes, sino que se trata solamente de reglas y heurísticas para ofrecer una buena experiencia de juego.

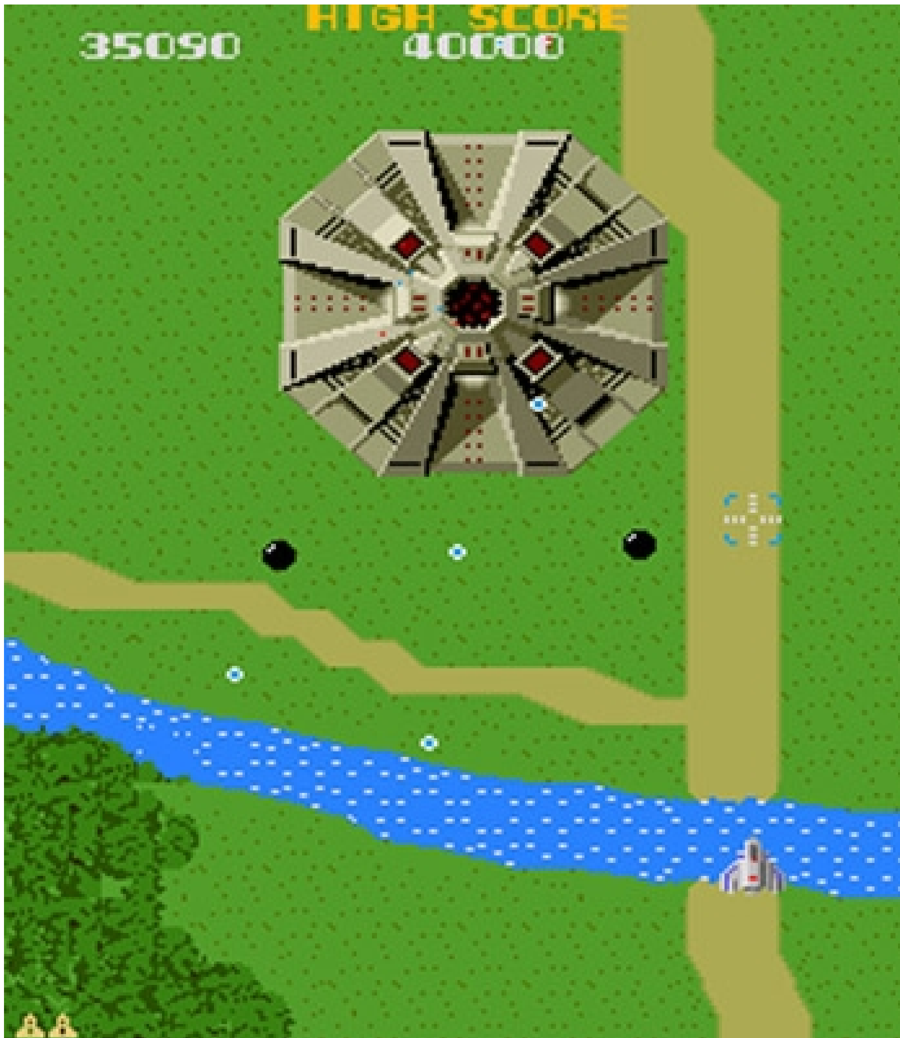
Otra diferencia que existe con la IA clásica es que, en ocasiones, debemos lograr que la IA sea más humana. Por ejemplo, conseguir que de vez en cuando nuestro sistema sea capaz de fallar, aun cuando el sistema es capaz de calcular la trayectoria perfecta para acertar siempre. En otro caso, el jugador se aburriría si ve que nunca es capaz de superar a un enemigo controlado por la IA. La calidad de la IA del juego se controla muchas veces mediante diferentes nive-

les de dificultad, que básicamente indican hasta qué punto debemos dotar de inteligencia a nuestros elementos y cuántas probabilidades existen de que el comportamiento que calculemos como el mejor se lleve a cabo efectivamente.

1.1. Historia de la IA en los videojuegos

Las necesidades de IA de los primeros juegos (desarrollados a lo largo de los años setenta y ochenta) eran bastante mínimas. Los juegos de plataformas clásicos simplemente tenían una serie de *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 © 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 empezó a utilizar máquinas de estados finitos (que ya explicamos en el módulo anterior), 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.

Técnica de búsqueda de caminos

Una técnica de búsqueda de caminos genera una lista de posiciones por donde un avatar debe moverse para llegar a un determinado punto. Este tipo de algoritmos los estudiaremos en el siguiente capítulo.

Dune II: Battle for Arrakis y Age of Empires II: The Age of Kings © Westwood Studios y Ensemble Studios respectivament

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, éstas son muy útiles para modelar ciertos comportamientos que no podemos crear usando técnicas deterministicas como *scripts* o patrones de reglas.

Battlecruiser 3000AD y Black & White © Take Two Software y Lionehad Studios respectivamente

Ya en los últimos años 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 LucasAsrts 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 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. IA, *scripting* y lógica

La IA se encuentra muy relacionada con dos de los aspectos que hemos visto en el módulo anterior, los lenguajes de *scripting* y el motor lógico del mundo.

La mayor parte de la implementación de la IA en el juego se realiza normalmente a base de *scripts*, por varias razones:

- Muchas veces los encargados de programar el comportamiento de los elementos controlados por la IA (denominados agentes inteligentes o simplemente agentes) no son los programadores experimentados, sino el di-

señador de niveles u otro perfil con unos conocimientos de programación no muy elevados.

- La creación de diferentes niveles de dificultad es mucho más fácil, ya que podemos probarlos y modificarlos mientras tenemos el juego en ejecución.

- La definición e implementación de grafos de estado es mucho más sencilla y flexible, lo que facilita la implementación de algunas de las técnicas que veremos en el próximo apartado.

- En el caso de tener que corregir un fallo es bastante más sencillo modificar un fichero de *script* que un ejecutable.

Por otro lado, la definición del mundo lógico es clave para poder proporcionar la información contextual a la IA para que este sistema pueda tomar decisiones. Como explicamos en el capítulo anterior, el mundo lógico contiene toda la información del estado del sistema.

En primer lugar, necesitaremos que el mundo se encuentre *discretizado* con algunas de las técnicas que hemos comentado, para que los agentes controlados por IA puedan localizar diferentes objetos y puedan saber cómo moverse por el mundo. El nivel de *discretización* utilizado permitirá a la IA acceder a más o menos información del mundo.

En segundo lugar, es importante establecer bien los parámetros de todos los elementos del juego en el mundo lógico y hacerlos públicos para todos los participantes. Tanto los jugadores reales como los controlados por IA deben tener acceso a la misma información para tomar decisiones, ya que no queremos que ni unos ni otros tengan ventaja sobre los agentes porque sepan más cosas.

Finalmente, es importante comunicar las decisiones de la IA de manera que el motor lógico las pueda entender. Lo más normal es que estas decisiones se codifiquen de una manera parecida a las decisiones del jugador para que el motor lógico las pueda interpretar de modo parecido.

1.3. La curva de dificultad

Uno de los elementos clave que definen el éxito de un juego es su curva de dificultad. En un eje de coordenadas situamos en las abcisas (eje X) el tiempo y en las ordenadas (eje Y) la dificultad del juego (que de alguna manera podemos cuantificar de fácil a difícil). La curva que describe la dificultad del juego a medida que se avanza en él la denominamos curva de dificultad.

Proporciona dos informaciones muy importantes, cómo de difícil hemos de hacer el juego en cada momento y cómo debemos cambiar la dificultad según el progreso del jugador. Cabe tener en cuenta que se trata de una medida relativa al nivel que muestre el jugador y simplemente indica cómo debemos ir adaptando la dificultad del juego según las habilidades del jugador.

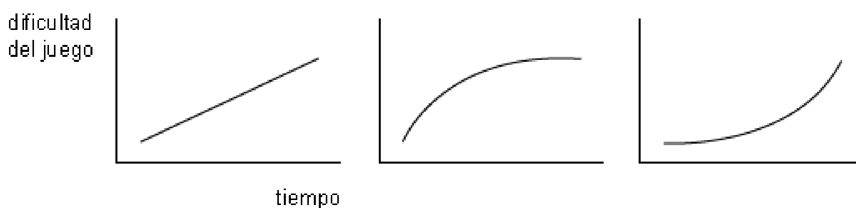
El objetivo de esta curva es muy simple: evitar las emociones negativas creadas por la dificultad del juego y potenciar las positivas. Es decir, si un juego es muy fácil, puede volverse aburrido muy rápidamente y si es muy difícil, puede volverse frustrante. Si los retos presentan un equilibrio justo, el jugador siente la recompensa del esfuerzo que está realizando.

Existen varios tipos de curvas de nivel utilizadas en juegos. Las más comunes son las siguientes:

- Las curvas lineales, donde la dificultad aumenta siempre del mismo modo.

- Las curvas convexas, donde existe más dificultad para empezar, pero a partir de un determinado punto el juego ya parece más fácil.

- Las curvas cóncavas, donde el juego al principio es muy fácil para facilitar las cosas al jugador, pero pasado cierto punto la dificultad crece exponencialmente.



Ejemplos de tres curvas de dificultad: lineal, convexa y cóncava

El diseño de la curva de dificultad implica la observación de diferentes aspectos, como la organización de los elementos en el juego o el número de enemigos y retos que proponemos. Muchas veces, la implementación de esta dificultad también radica en la calidad de nuestra IA y, por lo tanto, es muy importante ajustarla adecuadamente para obtener la curva deseada.

1.4. Uso de la IA

Pero, ¿cuándo y dónde hemos de usar la IA? El tema que nos interesa es la creación de videojuegos, así que no necesitamos generar un algoritmo que pase el test de Tuning, sólo necesitamos simular una serie de comportamientos que permitan una experiencia positiva de entretenimiento a nuestros jugadores.

El primer baremo que debemos tener en cuenta es la relación entre la jugabilidad y el realismo que queremos conseguir. Si pretendemos simular un comportamiento lo más real posible, lo más seguro es que el juego sea injugable para la mayoría de los jugadores y estaríamos hablando de un software de simulación (útil para pilotos, soldados o médicos, pero no para la mayoría de nuestros potenciales clientes). Dependiendo del tipo de experiencia que queramos proporcionar al usuario, decidiremos si queremos maximizar el realismo, si queremos maximizar la jugabilidad, o si queremos encontrar un término intermedio.

Reflexión

En un juego FPS donde los ataques de los enemigos estuviesen bien coordinados o siempre acertasen en el blanco, podría llegar a ser imposible jugar. Lo más razonable es provocar que exista una probabilidad asociada a acertar en el blanco, que pueda cambiarse según la dificultad elegida por el jugador (o incluso adaptarse de manera automática a su nivel, tal y como veremos posteriormente).

Otro punto que cabe tener en cuenta es la complejidad de los algoritmos de la IA. Por un lado, cuanta más complejidad, necesitaremos más tiempo de procesamiento. Si consideramos que nuestra aplicación presenta unos tiempos de ejecución estipulados, hemos de ceñir el tiempo de la IA a un porcentaje que permita una jugabilidad fluida y que los otros componentes del videojuego (gráficos, sonido, motor de lógica, física, etc.) no pierdan rendimiento. Además, dentro del tiempo de ejecución de la IA, normalmente puede que hayamos de calcular el comportamiento de varios elementos, con lo que también tendremos que decidir cómo vamos a repartir el tiempo de cálculo de IA entre todos ellos.

La complejidad de la IA

Debéis tener en cuenta que la complejidad de la IA no es proporcional al grado de satisfacción del jugador. Es posible que con una IA ajustada podamos generar una sensación de juego muy satisfactoria a nuestros clientes.

Dado que la IA ha cobrado más y más importancia dentro de los videojuegos actuales, con algoritmos más realistas pero más complejos, los programadores cada vez le dedican más recursos al cálculo de ésta. Si pretendemos generar un algoritmo complejo para la IA, una de las posibilidades que tenemos es utilizar los siguientes recursos de la máquina:

- Multiprocesadores: como ya vimos en el módulo de introducción, cada vez es más común tener varios procesadores (tanto las ultimas videoconsolas como los ordenadores llevan actualmente por defecto más de uno), por lo que podemos reservar uno o varios de ellos para tareas específicas de IA.

- Memoria: podemos reservar una zona de memoria de la máquina para guardar datos que podamos aprovechar en el futuro, como por ejemplo los caminos mínimos entre varios puntos decisivos de nuestro terreno. Cuanta más memoria tengamos dedicada para almacenar datos intermedios, menos cálculos habremos de efectuar y, por lo tanto, más rápidos podremos realizar nuestros algoritmos.

La implementación de los algoritmos de la IA podemos llevarla a cabo de dos maneras diferentes:

- Usando un hilo de ejecución independiente. En este caso deberemos crear sistemas de sincronización para poder dar la información al motor de lógica, pero evitando todos los bloqueos. Normalmente se utiliza en entornos multiprocesador, y puede tener tantos hilos como procesadores disponibles.
  Otra opción es tener el sistema de IA ejecutándose durante todo momento en este hilo paralelo, intentando encontrar la mejor decisión posible, y cuando el motor lógico le pregunta a la IA cuál es su decisión, ésta responde con la mejor que ha conseguido pensar y se pone a decidir el siguiente movimiento.

- En el bucle principal. Para ello, hemos de asegurarnos de que el código de la IA se ejecute siempre en un tiempo determinado y de que la sensación de continuidad por parte del jugador sea óptima. Esta opción reduce la complejidad de la implementación, ya que evitamos tener que sincronizar procesos y, muchas veces, es suficiente para juegos que no requieren un diseño de IA muy sofisticado.

2. Técnicas de movimiento

Las técnicas más sencillas que se utilizan en un videojuego son las relacionadas con el movimiento. Es importante generar algoritmos que permitan moverse a los distintos elementos móviles del juego por la escena.

Los objetivos que persigue el programador son:

- Aportar sensaciones al usuario a partir de los movimientos de los elementos. Según el tipo de juego querremos que los movimientos sean predecibles (en un arcade) o impredecibles (juego de rol).

- Lograr que los elementos imiten en lo posible un comportamiento natural (andar, correr, conducir, volar, etc.).

- Implementar comportamientos complejos para añadir dificultad a los retos del juego (persecución, huida, emboscadas, etc.)

Dividimos en tres los diferentes tipos de movimientos que podemos programar:

- Movimientos cíclicos y basados en patrones
- Búsqueda de caminos

2.1. Movimientos cíclicos y basados en patrones

Este tipo de movimiento es el más sencillo y el primero que se utilizó en los videojuegos. Se basa en definir el camino a partir de una curva y que el elemento se mueva siguiéndolo.

La implementación más sencilla sería definir el camino como una línea recta y que el elemento se mueva por ella a velocidad constante. Una vez llegue al final del camino, podemos hacer que el elemento retroceda por la misma línea hasta el inicio.

A partir de dos puntos, podemos definir la ecuación de la recta que los une. Provocar que un elemento siga la recta equivale a encontrar una serie finita de puntos repartidos entre los dos puntos (inicio y fin). Si suponemos una velocidad constante, podemos definir la siguiente función para generar los puntos intermedios, siendo *t* un valor entre cero y uno:

$$P(t) = (1-t) \cdot P_{begin} + t \cdot P_{end}$$

Este método es una interpolación lineal entre dos valores. Existen más métodos para realizar una interpolación entre dos o más valores que permiten crear curvas más complejas. Entre los métodos que definen este tipo de curvas tenemos el objeto matemático *spline*. Las dos curvas *spline* más utilizadas son las curvas de Bézier y las de Catmull-Rom.

Nota

Los puntos que definen una "spline" no tienen por qué pertenecer a la curva. Si pretendemos definir un camino que pase por determinados puntos, deberemos utilizar curvas que incluyan estos puntos en el camino, como las curvas *spline* de Catmull-Rom.

A partir de todos los puntos que nos definen un camino, podemos generar el algoritmo que nos interpole todas las posiciones por las que ha de pasar el elemento. Es necesario tener en cuenta su velocidad a la hora de moverlo para evitar saltos erróneos. Si tenemos un camino definido por dos puntos cercanos (una distancia de, por ejemplo, diez metros) y un tercero que diste bastante (a cien metros, por ejemplo), según cómo implementemos la interpolación mediante la ecuación anterior provocará que el elemento se mueva casi diez veces más rápido entre el segundo y el tercer punto que en el primer segmento. Para evitar este error deberemos conocer la longitud de cada tramo del camino, hacer que los tramos sean de la misma longitud o implementar otro modo de interpolar los puntos.

Presentamos a continuación un ejemplo de cómo interpolar varios puntos que forman un camino que cumpla el requisito de la proporción distancia/velocidad. La clase `Path` implementará la interpolación entre varios puntos (usando, por ejemplo, una *spline* Catmull-Rom). La clase `Walker` utilizará el camino para andar sobre él a una velocidad constante:

Path.h

```
class Path
{
public:
     Path();
     void add(const Point& point);
     Point interpolate(float t) const; // t must lie between 0.0 and 1.0
     const Point& getFirstPosition() const;
};
```

Walker.h

```
class Walker
{
public:
     Walker(const Path& path, float celerity, float resolution)
          : m_path(path)
          , m_celerity(celerity)
```

```
            , m_position(path.getFirstPoint())
            , m_nextPosition(path.getFirstPoint())
            , m_resolution(resolution)
            , m_nextT(0.0f)
        { };
        const Point& getPosition() const { return m_position; }
        void walk(double seconds);
        bool finish() const { return m_nextT > 1.0f; }

private:
        const Path& m_path;
        float m_celerity;
        Point m_position;
        Point m_nextPosition;
        Vector m_direction;
        float m_resolution;
        float m_nextT;
};
```

Walker.cpp

```
#include <Walker.h>

void Walker::walk(double seconds)
{
        // En remains guardamos los metros que nos quedan para llegar al siguiente punto
        float remains = (m_nextPosition * m_position).length();
        // Mientras no lleguemos al final (nextT = 1.0f significa que está al final del camino)
        // y que lo que nos queda para llegar al siguiente es casi cero (damos un pequeño
        // margen)
        while (m_nextT < 1.0f && remains < m_celerity * m_resolution)
        {
            // Le pedimos al camino el siguiente punto (a partir de nextT)
            m_nextT += m_resolution;
            m_nextPosition = m_path.interpolate(m_nextT);
            // Nos guardamos el vector que nos indica en qué dirección tenemos que caminar
            m_direction = m_nextPosition * m_position;
            float distance = m_direction.length();
            m_direction.normalize();
            remains = distance;
        }
        // Si no hemos llegado al final, la siguiente posición será la actual más
        // el vector dirección (normalizado) por la celeridad multiplicado por el número
        // de segundos que han transcurrido desde la última vez
        if (m_nextT <= 1.0f)
            m_position += m_direction * m_celerity * seconds;
```

```
}
```

Terminología

Velocidad es un vector y celeridad, su módulo. Lo que comúnmente denominamos velocidad realmente es la celeridad.

Al código anterior podemos implementarle una nueva funcionalidad para que el camino sea un bucle cerrado. De esta manera, podemos provocar que el objeto ande sobre un camino cíclico. El ejemplo más claro sería utilizar este nuevo código para implementar el movimiento de una ronda de un soldado. Por otro lado, también podemos utilizar el vector "direction" para orientar el objeto, es decir, que mire en la dirección en la que anda (a no ser que queramos que ande de espaldas o de lado).

Lo más importante es definir correctamente los puntos que definen el camino. Podemos aprovechar las *splines* para definir el camino, pero necesitaremos un algoritmo que interpole entre los puntos proporcionalmente para poder controlar la velocidad de los objetos que se mueven.

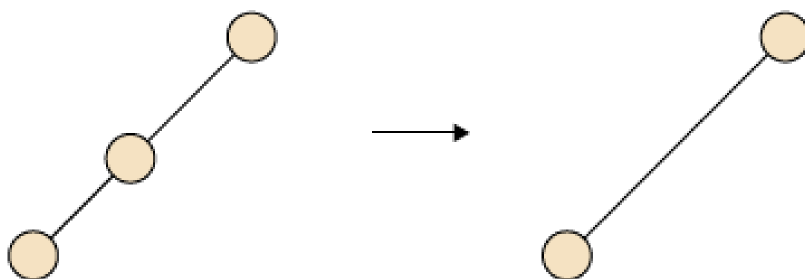
Procedimiento

Una vez generados todos los puntos de un camino, podemos guardarlos en un vector y evitar cálculos durante el juego.

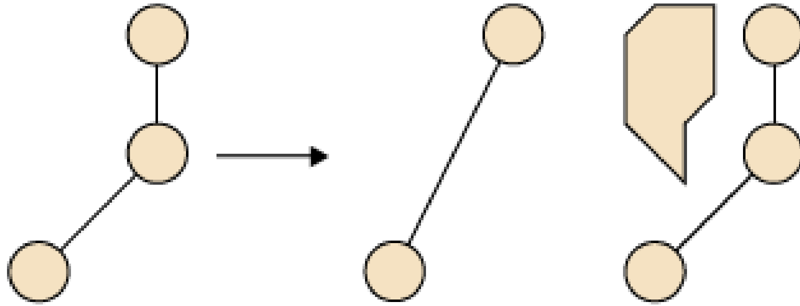
Los movimientos cíclicos y los basados en patrones nos serán útiles para realizar movimientos sencillos; sin embargo, en la mayoría de los casos necesitaremos algoritmos más complejos para decidir hacia dónde queremos que vaya el elemento.

Deberemos implementar nuevos algoritmos que nos calculen los puntos hacia los que tenemos que mover el elemento. A partir de estos puntos, podremos seguir trabajando con el algoritmo presentado en este punto.

Otro algoritmo que podemos aplicar una vez conocidos los puntos por los que transcurre el camino es el de optimización de puntos y de recorrido. Por un lado, tenemos que si tres puntos consecutivos están situados sobre una misma línea imaginaria (o con un error aceptable) podemos eliminar el punto intermedio:



Por otro lado, en el caso de que estemos construyendo un camino para andar sobre un terreno que puede contener obstáculos, es probable que el algoritmo que hemos utilizado no sea del todo óptimo. Si hemos usado un *grid* para calcular el camino, es posible que el camino generado se pueda optimizar, como en el siguiente supuesto de la izquierda, pero no en el de la derecha:



2.2. Búsqueda de caminos

La búsqueda de un camino (en inglés, *pathfinding*) es un problema que nos encontraremos en la mayoría de las aplicaciones que desarrollemos. El problema lo podemos plantear de la siguiente manera:

Tenemos un grafo que representa un espacio y queremos calcular la serie de nodos y aristas que se dan entre dos nodos. A cada camino le podemos asignar un coste (número de aristas recorridas, suma de longitudes de las aristas, etc.). El camino con menor coste será el camino mínimo entre esos dos puntos.

No siempre será necesario buscar el camino mínimo, solamente con encontrar uno óptimo nos será suficiente: podremos ahorrar recursos y evitaremos que el jugador prevea nuestros pasos.

A partir de la representación del mundo que hemos estudiado en el módulo "Lógica de un videojuego" podemos generar un *grid* o un grafo que represente todos los caminos por los que puede moverse un elemento.

En un determinado momento, podemos necesitar saber el camino que debe recorrer un elemento para ir de un punto A a un punto B. Los algoritmos más utilizados son:

- Recorrido en profundidad
- Recorrido en amplitud
- Dijkstra
- A*

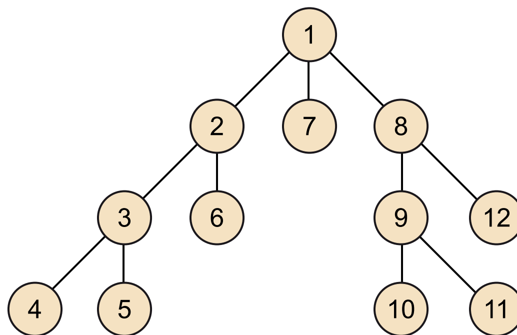
2.2.1. Recorrido en profundidad

El recorrido en profundidad (en inglés, *depth first search*, DFS) es el algoritmo más sencillo para buscar un camino en un grafo y consiste en visitar todos los nodos que podamos a partir del inicial. En este caso, tendremos una lista de nodos a los que hemos llegado y, a medida que vayamos tratándolos, iremos introduciendo aquellos vecinos de cada uno de los nodos. Debemos tener la precaución de guardarnos una lista de todos los nodos visitados para evitar ciclos.

Una búsqueda en profundidad significa que visitaremos primero los nodos a los que podemos llegar desde un determinado nodo, antes que los nodos que tenemos en la lista por visitar. Si vemos el grafo como un árbol, visitaremos antes los nodos hijos que los hermanos.

Procedimiento

Utilizaremos una interfaz "IGraph" que nos permitirá trabajar con el terreno por donde queremos movernos.



Ejemplo del orden en que se visitarían los nodos de este árbol en un recorrido en profundidad empezando por el nodo 1. Fuente: Wikipedia

A continuación presentamos un ejemplo en código en C++ que realiza una búsqueda en profundidad usando un algoritmo recursivo:

```
IGraph* graph = Model::singleton().getTerrain().generateGraph();
std::list<int> path;
bool found;
std::set<int> closed;

void depthSearch(int from, int to)     // guardará el camino en "path"
{
     path.clear();
     closed.clear();
     found = visit(from, to);     // visit nos devolverá si ha encontrado o no el camino
}
bool visit(int current, int to)
{
     if (current == to) // ¡ ha encontrado el camino !
     {
          path.push_front(to);
          return true;
     }
```

```
        // si el nodo que estamos visitando ya lo hemos visitado previamente quiere decir
        // que por aquí no hace falta continuar...
        if (closed.find(current) != closed.end())
             return false;
        // estamos visitando un nodo nuevo. Nos lo guardamos en la lista de visitados
        closed.insert(current);

        // Vamos a visitar los nodos vecinos
        std::list<int> neighbors = graph->getNeighbors(current);
        for(std::list<int>::iterator it = neighbors.begin(); it != neighbors.end(); it++)
        {
             // llamada recursiva, si desde este vecino hemos encontrado un camino hasta
             // el destino, añadimos el nodo actual e informamos que hemos encontrado el
             // camino
             if (visit(*it, to))
             {
                  path.push_front(current);
                  return true;
             }
        }
        // Ninguno de nuestros vecinos llega al destino
        return false;
   }
```

Podemos implementar este algoritmo sin utilizar la recursividad manteniendo en una lista ordenada los nodos que podemos visitar. Necesitaremos también una lista en la que guardemos el camino que hemos ido construyendo. Básicamente, el código sería:

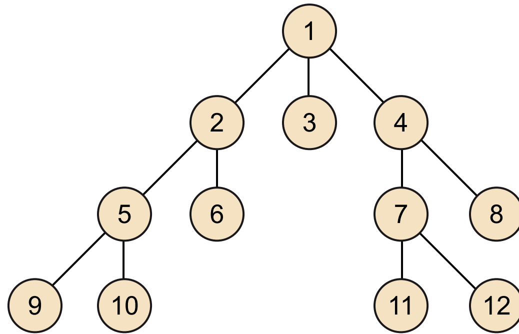
```
   IGraph* graph = Model::singleton().getTerrain().generateGraph();
   std::list<int> depthSearch(int from, int to)
   {
        std::list<int> ret;
        std::list<int> open;
        std::list<int> closed;
        std::map<int, int> parent;
        open.push_front(from);
        bool found = false;
        // En open tenemos todos los nodos a los que sabemos llegar, pero todavía no hemos
        // tratado
        while (!found && open.size() > 0)
        {
             // Vamos a visitar el primer nodo de la lista open
             int current = open.front();
             open.pop_front();
             if (current == to)
             {
```

```
            // Hemos llegado al final, vamos a construir el camino
            found = true;
            ret.push_front(to);
            // Mientras no lleguemos al principio...
            while (current != from)
            {
                // Guardamos el nodo actual y vamos para atrás
                ret.push_front(current);
                current = parent[current];
            }
        }
        // Si el nodo actual ya lo hemos tratado, simplemente lo descartamos
        else if (closed.find(current) != closed.end())
        { }
        else
        {
            // Si el nodo que estamos visitando no es el destino,
            // cogeremos a todos sus vecinos y los introduciremos en la
            // lista open (aquellos a los que sabemos llegar). También
            // guardaremos de donde venimos (en el map parent).
            std::list<int> neighbors = graph->getNeighbors(current);
            for(std::list<int>::iterator it = neighbors.begin();
                it != neighbors.end(); it++)
            {
                open.push_back(*it);
                parent[*it] = current;
            }
        }
    }
    return ret;
}
```

Los dos códigos anteriores no aseguran un camino mínimo, para conseguirlo deberíamos guardar cada uno de los caminos que fuéramos encontrando y continuar con el algoritmo. Una vez consiguiésemos todos los caminos, habríamos de seleccionar el que menor coste hubiese generado. Lógicamente, este algoritmo es ineficiente y deberemos utilizar uno de los algoritmos de los siguientes puntos.

2.2.2. Recorrido en amplitud

El recorrido en amplitud (en inglés, *breadth first search*, BFS) es un algoritmo análogo al anterior. Aquí la prioridad de búsqueda a partir de un nodo del árbol son primero sus vecinos y después sus hijos.



Ejemplo del orden en que se visitarían los nodos de este árbol en un recorrido en amplitud empezando por el nodo 1. Fuente: Wikipedia

Implementar este algoritmo usando la recursividad añade una complejidad innecesaria y poco didáctica. Por otro lado, su implementación de manera iterativa es muy parecida al recorrido en profundidad. Tan sólo es necesario modificar el modo de introducir los nodos adyacentes a la lista "open", que se introducirán al inicio y no al final.

Procedimiento

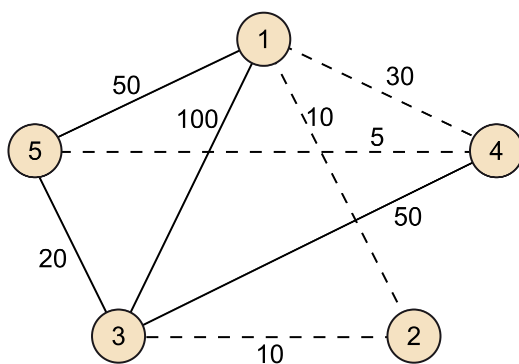
Solo es necesario cambiar la línea "open.push_back(*it)" por "open.push_front(*it)"

En el caso de que el coste entre nodos sea fijo, este algoritmo nos asegura encontrar el camino mínimo. Por ejemplo, si discretizamos el terreno utilizando un *grid* donde las distancias entre nodos sean constantes, este algoritmo nos dará el camino mínimo.

Si utilizamos otro tipo de discretización del mundo o añadimos un coste a cada arista utilizando otro tipo de cuantificación (por ejemplo, contemplando la pendiente del camino, el tipo de terreno, etc.) necesitaremos un algoritmo más elaborado para obtener el camino mínimo.

2.2.3. Dijkstra

Edsger Dijkstra, científico holandés, diseñó en 1959 un algoritmo de búsqueda de caminos mínimos en un grafo. A este algoritmo se lo conoce como el algoritmo de Dijkstra y consiste en generar un árbol en el que la raíz es el vértice de inicio y va ramificándose por todos los nodos del grafo sin crear ciclos. El algoritmo nos asegura que, utilizando estas ramificaciones, el camino entre un vértice y cualquier otro sea mínimo.



Ejemplo de los caminos mínimos desde el nodo 1 al resto de nodos utilizando el algoritmo de Dijkstra. El número del nodo indica el orden en que son añadidos y las líneas discontinuas, los caminos para llegar a ellos.

El algoritmo para calcular el árbol es el siguiente:

```
IGraph* graph = Model::singleton().getTerrain().generateGraph();
void calculate(int from)
{
     // De momento no sabemos llegar a ningún nodo con lo que la distancia
     // desde el origen a él es infinita
     for (IGraph::iterator it = graph->getNodes().begin();
          it != graph->getNodes().end(); it++)
     {
          it->distance = INFINITE;
          it->predecesor = NULL;
     }
     // Pero al nodo origen sabemos llegar y su distancia es cero
     graph->getNode(from).distance = 0;
     // En remainNodes guardaremos los nodos que faltan de tratar
     remainNodes = graph->getNodes();
     while (remainNodes.size() > 0)
     {
          // Extraemos de la lista de nodos que faltan de tratar el que tenga
          // una distancia mínima (la primera vez será el nodo origen puesto que los
          // demás tienen una distancia infinita)
          int current = remainNodes.extractMinDistance();
          // Recorremos todos sus vecinos y por cada uno recalcularemos su distancia como
          // la distancia entre el nodo origen al que estamos tratando más la distancia que
          // hay entre el nodo actual y su vecino.
          // Es posible que el nodo vecino ya tenga calculada una distancia (tenga otro
          // camino ya tratado que llegue hasta él). En este caso, debemos elegir el
          // camino que tenga una distancia menor al origen
          std::list<int> neighbors = graph->getNeighbors(current);
          for(std::list<int>::iterator it = neighbors.begin(); it != neighbors.end(); it++)
          {
               float distance = graph->calculateDistance(current, *it);
               if (current.distance + distance < it->distance)
               {
                    it->distance = current.distance + distance;
                    it->predecesor = current;
               }
          }
     }
}
```

Y para determinar un camino entre el origen y un destino:

```
std::list<int> dijkstra(int to)
{
     std::list<int> path;
```

```
        int current = to;
        while (current != from)
        {
            path.push_front(current);
            current = graph->getNode(curernt).predecesor;
        }
        path.push_front(from);
        return path;
    }
```

Cabe tener en cuenta que, si cambia el origen, se ha de volver a calcular todo el árbol.

Este algoritmo es muy costoso porque calcula todos los caminos posibles a partir de un nodo. Si queremos saber el camino entre un punto y otro, estaremos generando demasiada información que no necesitamos.

Nota

Normalmente tenemos puntos intermedios y puntos de destino, y sólo necesitaríamos calcular los caminos entre el nodo origen y los destinos, pero no con los intermedios.

Podemos utilizar este algoritmo para calcular, al inicio del juego, los posibles caminos que se deben seguir en nuestro espacio. A partir de aquí, un algoritmo que planifique determinados movimientos puede utilizar esta información sin tener que volver a calcularla. Por ejemplo, si un orco ha de caminar desde el mundo de las tinieblas a la comarca de la tierra media, puede obtener el camino a partir de la información que poseemos sin necesidad de volver a calcularla. Por el contrario, si el mundo se va modificando (se destruyen puentes, se desbordan ríos, se construyen túneles, etc.) deberíamos ir calculando la nueva información de los caminos; lo cual generaría bastante coste de proceso.

2.2.4. A*

A* es otro algoritmo que nos permite buscar un camino entre dos nodos de un grafo. Fue presentado en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael.

El algoritmo se basa en la búsqueda de caminos en profundidad y en amplitud. Mientras que los dos primeros presentan unas reglas fijas sobre qué elementos contemplar primero (hijos o hermanos), el algoritmo A-Star deduce en todo momento qué nodo es el que ha de visitarse mediante una función heurística.

Básicamente, se posee una lista de nodos que se han visitado y unos nodos a los que se sabe cómo llegar (a partir de los visitados). De los nodos a los que se puede llegar, se elige aquel que la función de heurística nos proporcione como el de menor coste y que, en teoría, es el camino más probable.

Si, a medida que vamos profundizando por un camino, éste se vuelve más costoso que sus alternativas, el algoritmo de selección aparcará el camino y continuará por otro nuevo y más prometedor.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	**19**	20
4	3	2	3	4	5	6	7	8		16	17	**18**	19
3	2	1	2	3	4	5	6	7		15	16	**17**	18
2	1	**0**	**1**	**2**	**3**	**4**	5	6		14	15	**16**	17
3	2	1	2	3	4	5	**6**	7		13	**14**	15	16
4	3	2	3	4	5	6	7	**8**		**12**	13	14	15
5	4	3	4	5	6	7	8	9	**10**	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Ejemplo de una búsqueda de camino mediante el algoritmo A*. Los números indican la distancia al nodo inicial. A cada paso de tiempo se busca aquel vecino que incremente la distancia al origen, pero que a la vez nos acerque más al destino.

El algoritmo siguiente implementa la búsqueda A*:

```
IGraph* graph = Model::singleton().getTerrain().generateGraph();
std::list<int> astar(int from, int to)
{
     // En la lista open tendremos todos los nodos a los que sabemos llegar. La primera
     // vez será el nodo origen
     std::list<int> path;
     std::list<int> open;
     std::list<int> closed;
     open.push_back(from);
     bool found = false;
     while (! found && open.size() > 0)
     {
          // Extraeremos el nodo que creemos que está más cerca del destino de la
          // lista open (a los que sabemos llegar). La distancia es una estimación, así
          // que es posible que extraigamos un nodo que no sea válido.
          int current = open.extractMinDistance();
          if (current == to)
          {
               // Si encontramos el destino, reconstruimos el camino y lo retornamos
               path.push_front(to);
               while (current != from)
               {
                    current = parent[current];
                    path.push_front(current);
               }
               found = true;
          }
          else
```

```
            {
                  // Calculamos la nueva distancia de todos los vecinos con el siguiente
                  // cálculo: la distancia desde el origen hasta el nodo actual (conocida)
                  // más la distancia entre el nodo actual y su vecino (en este ejemplo
                  // es uno) más una estimación desde el nodo vecino al destino.
                  float newDistance = distance[current] + 1.0f;
                  std::list<int> neighbors = graph->getNeighbors(current);
                  for(std::list<int>::iterator it = neighbors.begin();
                        it != neighbors.end(); it++)
                  {
                        float newEstimated = graph->calculateDistance(*it, to);
                        if (closed.contains(*it))
                        {
                              float total = distance[*it] + estimated[*it];
                              float newTotal = newDistance + newEstimated;
                              if (total > newTotal)
                              {
                                    distance[*it] = newDistance;
                                    estimated[*it] = newEstimated;
                                    parent[*it] = current;
                              }
                        }
                        else
                        {
                              closed.push_back(*it);
                              distance[*it] = newDistance;
                              estimated[*it] = newEstimated;
                              parent[*it] = current;
                        }
                  }
            }
      }
      return path;
}
```

2.2.5. Otras variaciones de *pathfinding*

Aparte de los algoritmos básicos expuestos en las secciones anteriores, existen otras alternativas para calcular el camino entre dos puntos. La mayoría de ellos son variaciones del algoritmo A* que se adaptan a situaciones típicas de los juegos, como mapeados enormes que no pueden ser explorados al detalle, o que utilizan el máximo de información disponible en el mundo para optimizar el proceso de cálculo del camino (por ejemplo, añadiendo datos extra que faciliten el proceso).

La primera variación y la más utilizada es el *pathfinding* jerárquico. Este funciona de la misma forma que planificamos los humanos nuestras rutas (y como funcionan sistemas del tipo Google Maps). Consiste en utilizar descripciones en diferente resolución del mundo y usar la más adecuada dependiendo de dónde nos queremos mover. Por ejemplo, usando el mismo ejemplo de Google Maps, deberíamos tener codificado primero un mundo con ciudades y autopistas, un segundo nivel más local con pueblos y carreteras y un tercer nivel dentro de una ciudad con sus calles.

Para implementar un *pathfinding* jerárquico necesitamos dos requerimientos especiales. Primero, debemos tener descrito el mundo en diferentes niveles de discretización con sus respectivas conexiones entre zonas. Y segundo, necesitamos un algoritmo que permita hacer *pathfinding* dentro de un nivel y saber cuándo cambiar de niveles.

Otra variación de algoritmos de *pathfinding* existente son aquellos algoritmos que intentan aplicar el sistema A* en condiciones de memoria limitada. Existen varias versiones, como el IDA* (*iterative deepening* A*) o el SMA* (*simplified memory-bounded* A*) que trabajan descartando aquellos caminos cuyas probabilidades son bajas en lugar de almacenar todo lo recorrido con anterioridad. El uso de estos algoritmos puede reducir la velocidad de cálculo, pero que, si no se encuentra el camino, hay que recalcular y pueden llevar a caminos menos óptimos que con un A*.

Finalmente, la última variación importante la encontramos en los algoritmos de *pathfinding* dinámicos. En estos casos partimos del hecho de que el sistema está en continuo movimiento o tenemos información incompleta del mapa. Por lo tanto, cada vez que dé un paso puede ser que la situación cambie. Esto no nos permite calcular todo el camino de antemano y a cada paso hay que reevaluar la dirección a tomar. Para este tipo de problemas se puede utilizar el algoritmo D* (de *dynamic* A*), que es un tipo de algoritmo que permite la búsqueda dinámica.

2.3. Movimientos complejos

Muchas veces los juegos están compuestos por grupos de entidades que se mueven por el mundo. En ocasiones queremos que el movimiento de estas unidades se encuentre relacionado entre ellas, de modo que en lugar de dar una posición de origen y una de destino, lo que queremos es que se siga un camino que tenga una función concreta, como por ejemplo:

- Interceptar a otro elemento móvil

- Perseguir a otro elemento móvil

- Evadir o huir de otro elemento móvil

- *Flocking* (movimientos colectivos donde se intenta mantener un grupo compacto de unidades)

- Movimientos programados (sincronización de varios elementos)

Para llevar a cabo todas estas tareas, existen una serie de algoritmos optimizados para cada una de estas rutas. Vamos a analizar los casos que hemos comentado.

2.3.1. Movimientos autónomos

En este apartado vamos a incluir todos los algoritmos que nos servirán para mover elementos según un determinado perfil: interceptar, evadir o perseguir. A este tipo de movimientos también nos referimos como movimientos de dirección o, en inglés, *steering behaviors*.

La idea parte de que tenemos un elemento autónomo en el terreno de juego que debe tener un determinado comportamiento. Este tipo de comportamiento lo calcularemos según los algoritmos del siguiente capítulo (toma de decisiones) y de momento no entraremos en este detalle.

Los comportamientos más comunes son:

- Interceptar a otro elemento móvil

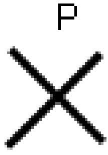
- Perseguir a otro elemento móvil

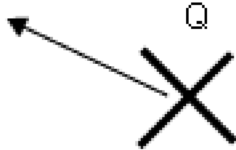
- Evadir o huir de otro elemento móvil

Existen bastantes más tipos de *steering behaviors*, como por ejemplo, adaptar la velocidad de movimiento (o la dirección) de un objeto a una referencia, vagabundear (moverse de forma semialeatoria sin rumbo definido), esquivar objetos, mirar hacia una dirección, ... La implementación de la mayoría de estos comportamientos se puede deducir a partir de los más comunes que veremos a continuación.

Interceptar

Es el comportamiento más sencillo. A partir de nuestra posición y la posición del contrario, debemos modificar nuestra velocidad (celeridad y dirección) para interceptarlo.





La velocidad la podemos calcular con la siguiente función:

$$\vec{v} = P - Q$$

Pero si tenemos un vehículo con una velocidad máxima, la función se convierte en:

$$\vec{v} = \frac{P - Q}{|P - Q|} \cdot V_{max}$$

Perseguir

A partir de la posición de un elemento móvil y de su velocidad, podemos pronosticar sus movimientos en un futuro cercano. Partiendo de nuestros cálculos, podemos determinar qué velocidad debemos aplicar a nuestro vehículo para encontrarnos al contrario. Si suponemos un terreno donde el vehículo en persecución está en el punto P, su velocidad es $\vec{v}$ y nuestro vehículo está en el punto Q:

Podemos calcular la velocidad $\vec{v}_2$ con la siguiente ecuación:

$$\begin{cases} R - P = t \cdot \vec{v} \\ R - Q = t \cdot \vec{v}_2 \end{cases}$$

Lectura de la fórmula

Donde P, Q y V son valores conocidos, y podemos calcular V2 en función de *t*: cuanto más grande sea *t*, nuestra velocidad se parecerá a la del perseguido porque el punto R estará más lejos y nuestras trayectorias serán casi paralelas

$$\vec{v}_2 = \frac{Q - P}{t} - \vec{v}$$

Con esta ecuación podemos averiguar la velocidad (celeridad y dirección) que debemos aplicar a nuestro vehículo para interceptar al contrario en *t* segundos. Como nuestro vehículo seguramente tendrá unas limitaciones físicas (velocidad máxima, por ejemplo), deberemos de recalcular este valor a partir de este nuevo valor, $v_{máx}$, con la siguiente ecuación:

$$\begin{cases} \vec{v_2} = \frac{Q-P}{t} - \vec{v} \\ |\vec{v_2}| = V_{max} \end{cases}$$

Evadir o huir

Este comportamiento sería análogo a los dos anteriores. En vez de sumar la velocidad, habremos de restarla para escaparnos del punto donde estará nuestro contrincante en un determinado momento.

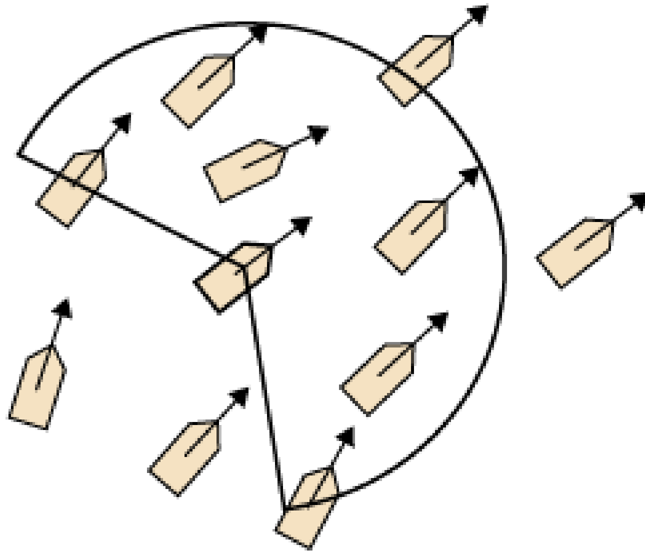
2.3.2. Movimientos colectivos (*flocking*)

Muchas veces en los videojuegos los elementos se mueven en grupos cohesivos en lugar de moverse de manera independiente. Imaginemos un ejército: el movimiento óptimo de un soldado es siempre lo más cerca posible de los compañeros para mejorar la defensa global (si cada uno siguiera su camino podría ser bastante desastroso). Este tipo de movimiento colectivo se denomina *flocking* y se han creado un gran número de algoritmos que intentan imitar los comportamientos de grupos en la naturaleza, como enjambres de abejas, manadas de búfalos o bandadas de aves.

Uso de los algoritmos de *flocking*

Los algoritmos de *flocking* se utilizan mucho en efectos especiales de las películas, como por ejemplo en las batallas épicas de las trilogía *El Señor de los anillos* para controlar a miles de soldados simultáneamente.

El algoritmo más básico de *flocking*, propuesto por C. Reynolds en 1987, contempla cada elemento como un individuo independiente que sigue una dirección y que a la vez puede observar la dirección de algunos de sus vecinos, utilizando la información de éstos para decidir en cada momento hacia dónde se va a dirigir.



Ejemplo de una unidad moviéndose en una dirección, con su radio de observación de sus vecinos

En el diseño de un sistema de *flocking* intervienen algunos parámetros que son importantes para que el comportamiento observado reproduzca un sistema real:

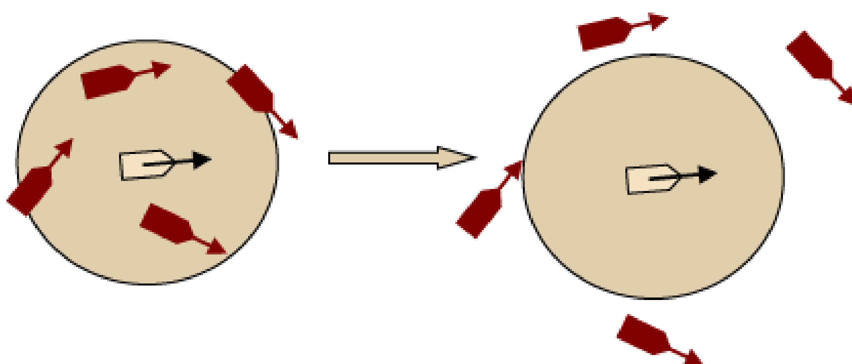
- La dirección global del grupo: nos indica la tendencia global que todos los elementos deben seguir para llegar a su objetivo. Se encuentra definida mediante el punto de destino, pero no definimos la ruta completa que se ha de seguir.

- El radio y la distancia de visión de un elemento: se utiliza para saber cuál es la dirección del grupo. Mayores radio y distancia proporcionan más información; sin embargo, pueden volver más complicada la toma de decisiones (y más costoso computacionalmente).

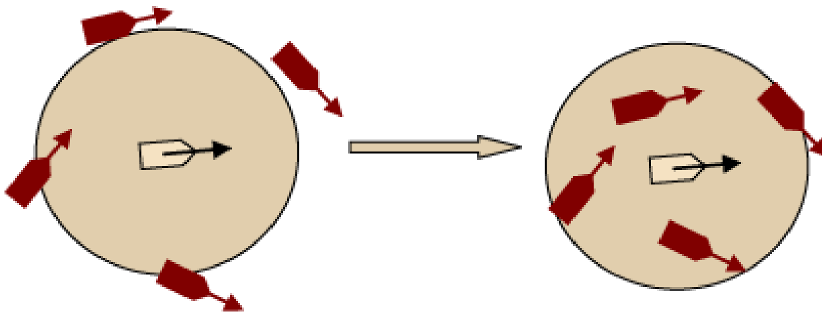
- Un factor de separación: para evitar que el elemento se acerque demasiado al resto y así evitar colisiones.

Nota

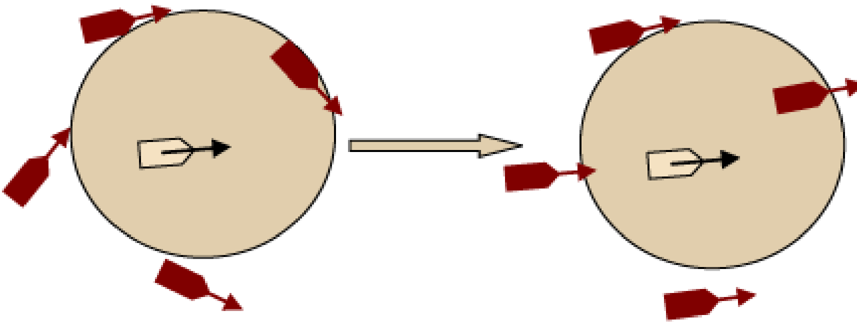
Por ejemplo, se ha demostrado científicamente que los pájaros deciden su movimiento mirando a sus cinco vecinos más cercanos.



- Un factor de cohesión: para mantener todos los elementos juntos y que no se separen del grupo.



- Un factor de alineamiento: para que todos los elementos sigan en una misma dirección.



Normalmente todos estos parámetros se incluyen en un algoritmo que decide en cada paso cuál es la dirección que se debe seguir, si hemos de ir en la dirección del grupo o si en cambio hemos de corregirla. El algoritmo más básico utiliza una media de las direcciones de los elementos colindantes, junto con el control del alineamiento, la separación y la cohesión.

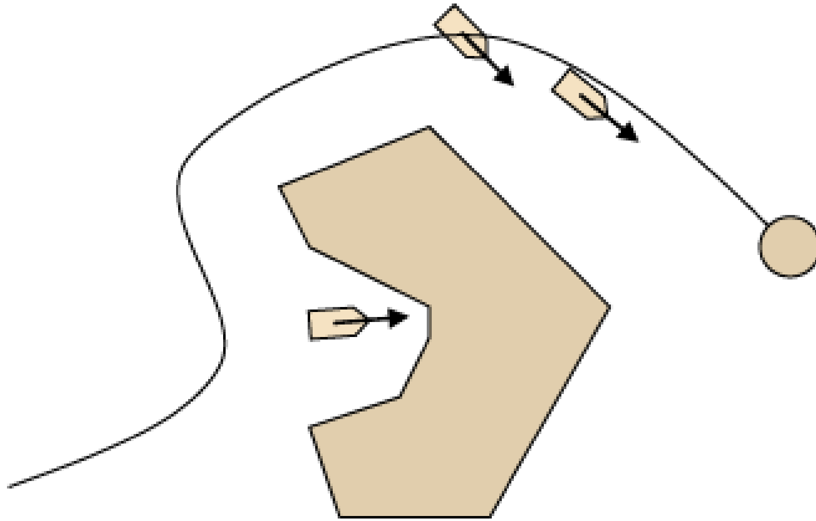
¿Cómo podemos utilizar los algoritmos de *flocking*

Tenemos una serie de individuos que queremos que vayan a un lugar determinado. Podemos planificar un camino para cada uno, pero en este caso tenemos los siguientes inconvenientes:

- Tenemos un gasto de cálculo proporcional al número de individuos y puede llegar a ser muy costoso.

- Los individuos pueden estar en la misma posición en un mismo instante, situación que la mayoría de veces no es correcta.

Podemos tratar a todos los individuos como un grupo y planificar el camino para el mismo. Para ello tenemos una serie de puntos intermedios a los que deben dirigirse sus integrantes. En este caso, si los individuos se hallan distantes entre ellos (algún individuo se ha rezagado), es posible que se encuentren con la situación descrita en la siguiente figura y les sea imposible llegar al destino.



El círculo indica el siguiente punto al que deben dirigirse, pero el individuo que hay en el centro del dibujo no podrá llegar nunca

Para evitar este problema, podemos definir un grupo como una serie de individuos que se encuentran cercanos. En el momento en el que un individuo deja de estar cerca de un grupo, lo deberemos tratar como otro grupo y planificar su camino por separado. En el caso de que se acerquen dos grupos, pasarán a ser uno y sólo se tendrá en cuenta uno de sus caminos.

Ejemplo de una implementación de *flocking*

En el siguiente ejemplo, implementamos el comportamiento del movimiento de un soldado que depende del comportamiento de la tropa en su conjunto. La clase Soldier tiene como atributos la posición actual, la dirección (orientación) y la dirección a la que debe ir.

El soldado sólo debe caminar según le indique la dirección, el código sería:

Soldier.cpp

```
void Soldier::walk(float timeSinceLastFrame)
{
     m_direction += (m_desiredDirection - m_direction) * factor;
     m_direction.normalize();
     m_position += m_direction * timeSinceLastFrame;
}
```

Hemos incluido un factor que nos indicará cómo de rápido se girará el soldado.

Una tropa contendrá una serie de soldados que otro algoritmo nos habrá asociado. La tropa tendrá un punto al que ha de ir, al cual denominaremos destino (aunque sólo sea un punto intermedio del camino que debe recorrer). El código para hacer caminar a la tropa sería:

Troop.cpp

```
void Troop::walk(float timeSinceLastFrame)
{
     // Recorremos todos los soldados de la tropa
     for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
     {
          Soldier& soldier = *it;
          // La dirección a la que debe caminar el soldado será hacia la que
          // le lleve al destino
          Vector4 desiredDirection m_destination - soldier.getPosition();
          desiredDirection.normalize();
          soldier.setDesiredDirection(desiredDirection);
     }

     // Una vez calculado la dirección a todos los soldados, miremos los factores
     // de Flocking
     updateSeparation();
     updateCohesion();
     updateAlignment();

     // Una vez ajustada la dirección deseada a cada soldado, hagámosles caminar
     for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
          it->walk(timeSinceLastFrame);
}
```

En el ejemplo anterior sólo hemos indicado al soldado la dirección hacia la que debe caminar. Para ajustarla hemos llamado a tres funciones que modifican estos valores según el comportamiento de toda la tropa.

El código del comportamiento según el factor de separación es:

Troop.cpp

```
void Troop::updateSeparation()
{
     // Iteramos por todos los soldados
     for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
     {
          Soldier& soldier = *it;
```

```
            Vector4 desiredDirection = soldier.getDesiredDirection();
            Vector4 newDirection(0.0f, 0.0f, 0.0f, 0.0f);
            // Por cada soldado, miraremos de modificarle la dirección deseada para
            // que se aleje de sus compañeros
            for(std::list<Soldier>::iterator it = m_soldiers.begin();
                 it != m_soldiers.end(); it++)
            {
                 Soldier& neighbor = *it;
                 // Un soldado no se puede separar de si mismo!
                 if (neighbor != soldier)
                 {
                      // Modificaremos la dirección deseada restando la
                      // distancia entre el vecino y el soldado, pero multiplicado
                      // por un factor inversamente proporcional a la distancia
                      Vector4.distance = neighbor.getPosition() soldier.getPosition();
                      float length = distance.length();
                      float factor = length * length;
                      if (factor > 0.0f)
                           newDirection -= distance / factor;
                 }
            }
            // Si tenemos que ajustar la dirección al soldado
            if (newDirection.length() > 0.0f)
            {
                 // A la dirección deseada le sumamos la dirección calculada
                 // multiplicada por un factor de separación definido
                 newDirection.normalize();
                 desiredDirection += newDirection * separationFactor;
                 desiredDirection.normalize();
                 soldier.setDesiredDirection(desiredDirection);
            }
       }
}
```

El factor de cohesión, sería:

Troop.cpp

```
void Troop::updateCohesion()
{
     if (m_soldiers.size() == 0)
          return;

     // Primero de todo, buscamos el centro del grupo
     Vector4 center;
     for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
          center += it->getPosition();
```

```
        center = center / m_soldiers.size();

        // Una vez calculado, de cada soldado miramos la distancia que tiene al centro
        // y le sumamos un vector proporcional a la dirección deseada
        for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
        {
             Vector4 newDirection = (center - it->getPosition());
             if (newDirection.length() > 0.5f) {
                  newDirection.normalize();
                  newDirection = newDirection * cohesionFactor;
                  newDirection += it->getDesiredDirection();
                  newDirection.normalize();
                  it->setDesiredDirection(newDirection);
             }
        }
   }
```

Por último, el factor de alineamiento sería:

Troop.cpp

```
   void Troop::updateAlignment()
   {

        if (m_soldiers.size() == 0)
             return;

        // Calculamos la media aritmética de la dirección de los soldados
        Vector4 avgDirection;
        for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
             avgDirection += it->getDesiredDirection();
        avgDirection = avgDirection / m_soldiers.size();

        // De forma análoga al ejemplo anterior, le añadimos a la dirección deseada
        // de cada soldado un factor del promedio
        for(std::list<Soldier>::iterator it = m_soldiers.begin(); it != m_soldiers.end(); it++)
        {
             Vector4 newDirection = avgDirection - it->getDesiredDirection();
             if (newDirection.length() > 0.5f) {
                  newDirection.normalize();
                  newDirection = newDirection * alignmentFactor;
                  newDirection += it->getDesiredDirection();
                  newDirection.normalize();
                  it->setDesiredDirection(newDirection);
             }
        }
```

```
}
```

2.3.3. Movimientos programados

Podemos aprovechar los algoritmos de búsqueda de caminos para precalcular los movimientos de una serie de elementos y que el resultado sea: óptimo, coherente e, incluso, bonito.

La idea consiste en planificar el movimiento de un elemento desde una posición inicial a una de destino, pero incluiremos en el grafo una dimensión nueva que representará el tiempo. De este modo, tendremos planificado el camino del elemento y en qué nodo se hallará en un determinado instante.

A partir de esta información, planificaremos el siguiente elemento, sin embargo, no tratará aquellos nodos (posición y tiempo) por donde pasa el primer elemento. De esta manera, tenemos el segundo movimiento planificado y sin colisión alguna con el primer elemento. Por otro lado, también tendremos los dos caminos mínimos.

Si iteramos este algoritmo por todos los elementos del grupo, obtendremos como resultado que todos los movimientos parecen sincronizados. Por ejemplo, es posible que un soldado se quede parado a la espera de que pase otro soldado por delante de él y después continuará la marcha.

Otro ejemplo muy visual consiste en cambiar de una formación a otra un escuadrón de aviones de combate.

3. Toma de decisiones

El rol principal de la inteligencia artificial en los videojuegos es el de tomar decisiones acerca de qué debe realizar un sistema inteligente durante el transcurso de la partida. Del resultado de estas acciones dependerá que seamos capaces de ofrecer un reto adecuado al nivel del jugador.

El proceso de cómo se razonan las decisiones ha sido objeto de estudio en la psicología humana durante muchos años. Dentro del campo de la IA existen muchas maneras de aproximarse a este problema.

En este punto vamos a analizar qué aspectos hemos de tener en cuenta para decidir qué técnica es la más apropiada para realizar el proceso de toma de decisión.

En primer lugar, a la hora de tomar una decisión nos podemos encontrar en dos situaciones diferentes dependiendo de la cantidad de información de la que dispongamos:

- Situación de certidumbre o certeza: cuando tenemos conocimiento total de la situación y sabemos todos los resultados posibles que existen a partir de nuestras acciones. Al tomar la decisión sólo hemos de elegir aquella que al final nos vaya a proporcionar el máximo beneficio. Un ejemplo de esto sería un juego de mesa como el Tic-Tac-Toe, donde conocemos todas las alternativas y sus resultados.

- Situación de incertidumbre: cuando sólo disponemos de un conjunto de información incompleta que no nos permite garantizar que la decisión que tomemos sea la que vaya a beneficiarnos o a solucionar nuestro problema. Por ejemplo, en un juego de ajedrez no sabemos si a la larga una decisión va a ser buena o mala.

En segundo lugar, debemos tener en cuenta a qué escala temporal va a afectar nuestra decisión. En este caso tenemos tres tipos de sistemas de decisión:

- Decisión estratégica: se trata de aquellas decisiones que se toman y tienen efecto en un largo período de tiempo. Necesitan normalmente muchos datos de entrada, ya que del éxito de esta decisión puede depender el resultado final de la partida. Por ejemplo, en un juego de fútbol, la estrategia puede ser si vamos a jugar ofensiva o defensivamente.

- Decisión operacional: se trata de aquellas decisiones que tienen un efecto inmediato. Utilizan muy poca información, pero se realizan muchas más decisiones de este tipo que en los niveles superiores, así que se han de

calcular en un tiempo muy rápido. En el caso de un juego de fútbol, una decisión operacional sería decidir si se pasa el balón o si se chuta a portería.

- Decisión táctica: se trata de un nivel intermedio entre la estrategia y la operación, donde se considera un subgrupo de los elementos del juego y se decide su comportamiento. El objetivo de este nivel es aplicar el plan diseñado en el nivel estratégico de la mejor manera. En el mismo ejemplo del fútbol, la decisión táctica sería cómo llevar a cabo la estrategia que tenemos planteada. Por ejemplo, si es defensiva, decidiremos si marcaremos en zona o individualmente.

Finalmente, otro aspecto que debemos contemplar para saber qué método es el más adecuado es la frecuencia con la que vamos a tener que tomar una determinada decisión:

- Decisiones programadas: se trata de aquellas decisiones rutinarias que se toman frecuentemente. En este caso es más fácil poseer un método bien establecido para tomar una decisión rápida. Por ejemplo, el movimiento rutinario de patrulla de un soldado en un juego se repite de manera regular.

- Decisiones no programadas: aquellas decisiones que hemos de realizar menos a menudo, normalmente asociadas a situaciones más excepcionales. En este caso, es necesario un estudio más detallado de la situación para poder intentar decidir cuál es la decisión óptima que podemos tomar. Por ejemplo, un ataque contra el soldado que se encuentra patrullando es una acción puntual que debe estudiar muchos factores para saber cómo actuar.

Reflexión

Dependiendo de si queremos utilizar un cierto porcentaje de azar en nuestras decisiones, necesitaremos elegir entre programar un sistema inteligente determinista o no determinista.

Por un lado, los sistemas deterministas nos ayudan a tomar decisiones sin que intervenga el azar. Se trata de un sistema en el que se evalúa la información y se extrae una única decisión posible. Se trata del sistema más clásico utilizado en los primeros videojuegos, ya que es el más rápido en calcular y nos da directamente una única solución para cada decisión que debamos tomar. El problema de este tipo de sistemas es que el comportamiento resultante es muy mecánico y repetitivo.

Por otro lado, los sistemas no deterministas incorporan el azar en la toma de decisiones. En este caso, el resultado del algoritmo de decisión suele ser una serie de posibles acciones con una determinada probabilidad de elección asociada a cada una. Entonces se elige un valor aleatorio y se ejecuta la acción relacionada con este número. Normalmente este proceso es más costoso, ya que generamos varias soluciones que hemos de ponderar, pero el resultado es más "humano", ya que, por ejemplo, nos es mucho más fácil modelar la posibilidad de que los adversarios se equivoquen en sus decisiones.

Existen multitud de técnicas que cubren todos los aspectos detallados en esta introducción. En primer lugar, vamos a explicar un sistema general de toma de decisiones, y después entraremos en detalle con algunas de las técnicas principales que se utilizan en la toma de decisiones de la IA de los videojuegos.

3.1. El proceso de toma de decisiones

Dentro del proceso de tomar una decisión, existe una serie de etapas comunes que es necesario seguir en todos los algoritmos:

- Identificar y analizar la situación: en primer lugar, debemos reunir información sobre la situación en la que nos encontramos y observar qué tipo de acciones son adecuadas en dicha situación. Dependiendo del nivel de la decisión necesitaremos acumular más o menos volumen de información.

- Identificar los criterios de decisión y ponderarlos: a partir de la información que obtenemos, debemos decidir qué partes de esta información nos interesan y asignarles una ponderación, es decir, una importancia relativa.

- Generar las alternativas de solución: en este punto es donde difieren principalmente todos los diferentes métodos de la IA que presentaremos en este apartado. El objetivo de esta etapa es buscar una o varias posibles decisiones para la situación actual a partir de los datos ponderados en el punto anterior.

- Evaluar las alternativas: a continuación debemos elaborar un estudio detallado de las diferentes decisiones que podemos tomar, como mirar sus ventajas y desventajas y asignarles una ponderación.

- Elección de la mejor alternativa: dependiendo de la capacidad de cálculo de nuestro sistema, podemos simplemente elegir la primera decisión que nos dé un resultado razonable de la manera más rápida, o podemos efectuar una selección más detallada para optar por la mejor decisión posible.

- Implementación de la decisión: una vez hemos elegido qué decisión se ha de llevar a cabo, la enviamos al motor lógico del juego para que la ejecute.

- Evaluación de los resultados: el último punto es valorar el resultado obtenido al ejecutar esta decisión. El resultado es importante para ciertos algoritmos, sobre todo para aquellos que permiten el aprendizaje de las acciones mientras se juega.

3.2. Técnicas de IA para tomar decisiones

En este apartado vamos a analizar un amplio conjunto de técnicas de IA que nos facilitarán la toma de decisiones. En ocasiones podemos combinar algunos de estos sistemas o utilizar varios de ellos para diferentes tipos de entidades inteligentes.

Todas las técnicas se pueden implementar de varias maneras. En este apartado sólo comentaremos la idea principal que se halla detrás de cada una de las técnicas, pero no entraremos en detalles de implementación.

Finalmente, debéis recordar que el éxito de las técnicas depende de varios factores, como la capacidad de cálculo que tengamos o la correcta reunión y codificación de la información.

3.2.1. Sistemas de reglas

Los sistemas de reglas son quizá el tipo de IA más utilizado en la historia de los videojuegos. En su forma más simple consisten en un conjunto de instrucciones *"If ... then ..."* que se utilizan para evaluar estados y tomar decisiones.

Nota

Los sistemas de reglas utilizan mucho los operadores booleanos AND, OR y NOT. Estos se pueden combinar para obtener operaciones adicionales (NAND, NOR, XOR, XNOR,...) y expresiones lógicas más complejas (¡tened cuidado con los paréntesis y la precedencia!).

Los sistemas de reglas presentan dos ventajas principales respecto al resto de sistemas de IA. En primer lugar, se trata de un sistema que intenta reproducir la manera de pensar y razonar de los humanos ante una situación a partir de la información que tienen de la misma. Y en segundo lugar, es un sistema muy fácil de programar.

Por otro lado, el principal problema de un sistema de reglas es que es poco escalable. Cuando queremos implementar muchas reglas, el sistema se vuelve difícil de gestionar y el tiempo necesario para el estudio de las reglas puede provocar un efecto negativo en el rendimiento del proceso de IA. Además, los sistemas de reglas tampoco son muy efectivos cuando se dan situaciones poco comunes que no se encuentran especificadas dentro de las reglas.

Un sistema de reglas está formado por tres componentes principales:

- La memoria de trabajo: aquí incluimos todos los hechos y todas las afirmaciones que tengan relación con la situación actual, tanto aquello que leemos directamente desde el entorno (como, por ejemplo, que se hallan cinco enemigos cerca) como aquello que inferimos nosotros mediante las reglas (como, por ejemplo, que existe un elevado riesgo de ataque).

memoria_trabajo.h

```
enum condicion {cierto, falso, probable, desconocido};

int num_enemigos_tierra;
int distancia_media_enemigos_tierra;
int cantidad_defensas_terrestres;
condicion ataque_inminente_tierra;
```

- El conjunto de reglas: se trata de secuencias del tipo *"If ... then ..."*, con una serie de condiciones, y que pueden dar como resultado o bien inferir más

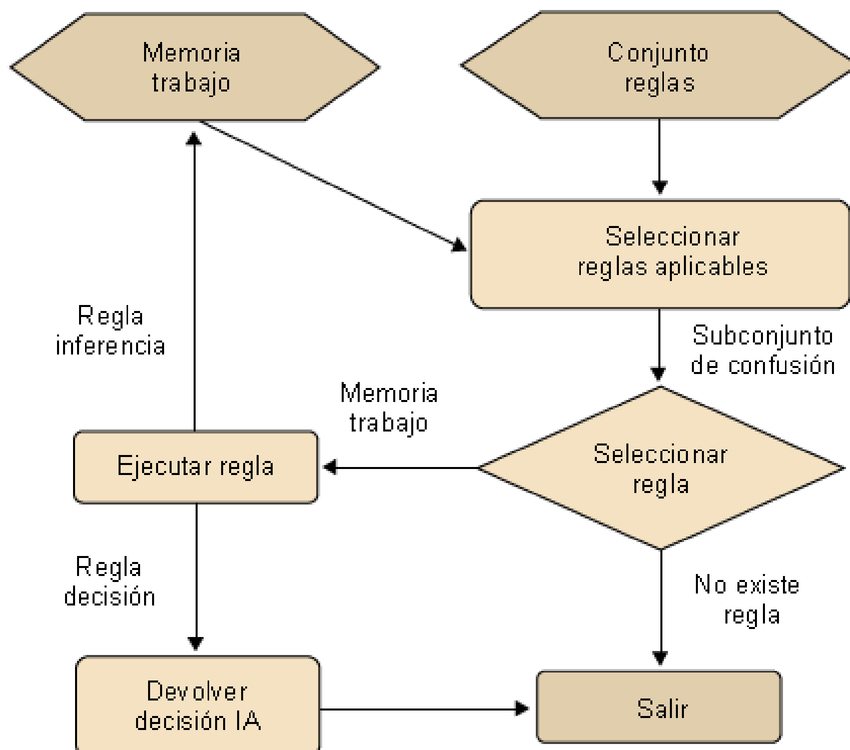
información para la memoria de trabajo, o bien encontrar la decisión final sobre qué hemos de realizar en ese momento.

conjunto_reglas.c

```
if(num_enemigos_tierra > 5 && distancia_media_enemigos_tierra < 100)
     then inferir(ataque_inminente=probable)

if(num_enemigos_tierra > 10 && distancia_media_enemigos_tierra < 50)
     then inferir(ataque_inminente=cierto)

if(ataque_inminente_tierra == probable && cantidad_defensas_terrestres < 10)
     then ejecutar(construir_defensas_terrestres)

if(ataque_inminente_tierra == cierto)
     then ejecutar(mover_defensas_terrestres)
```

- Una condición que determine si se ha hallado la decisión adecuada para la situación o si no existe ninguna dentro del sistema. Esto es necesario para terminar el bucle de decisión del sistema de reglas.

Existen varias maneras de realizar la inferencia utilizando el subconjunto de reglas. Un posible proceso de inferencia en un sistema de reglas es el siguiente, conocido como "forward chaining":



En primer lugar, a partir de la información contenida en la memoria de trabajo y el conjunto de reglas, encontraremos el subconjunto de reglas que podemos aplicar en este momento, lo que se conoce por el "subconjunto de conflicto".

Si este subconjunto contiene más de una posible regla aplicable a la situación actual, debemos decidir cuál de ellas es la preferida. Existen varias estrategias para decidir qué regla vamos a seleccionar:

- Podemos elegir la primera que encontremos. Es el método más rápido, pero no nos garantiza que sea la mejor opción que tenemos.

- Podemos optar por una de ellas aleatoriamente. Esto nos dará cierta incertidumbre que puede ser beneficiosa para el juego.

- Podemos seleccionar la regla más específica, es decir, aquella que cumpla más condiciones.

- Podemos tomar la menos utilizada para garantizar que todas ellas se ejecuten alguna vez y así evitar comportamientos muy repetitivos.

- Finalmente, si tenemos la posibilidad de asignar alguna ponderación a cada una de ellas, podemos decidirnos por la que nos dé el valor más alto.

El resultado de una regla puede ser aumentar la memoria de conocimiento:

```
if(num_enemigos_tierra > 5 && distancia_media_enemigos_tierra < 100)
     then inferir(ataque_inminente=probable)
```

O puede darnos la decisión de qué vamos a realizar en este turno:

```
if(ataque_inminente_tierra == cierto)
     then ejecutar(mover_defensas_terrestres)
```

En el primer caso, volveremos a repetir el proceso y en el segundo devolveremos la acción o acciones que creemos más convenientes para la situación actual, avisando al sistema de que debe finalizar el bucle de inferencia.

Reflexión

Un caso particular de los sistemas de reglas es lo que conocemos por sistemas expertos. En los sistemas expertos se busca una mejor calidad y rapidez en las respuestas, y se utilizan en multitud de campos, como la diagnosis médica o en los sistemas de gestión de finanzas.

Un sistema experto es un sistema de reglas que incorpora muchas más reglas (hasta miles de ellas). En este caso se necesita optimizar todo el proceso de inferencia para hacerlo más rápido. En los videojuegos, el uso de sistemas expertos se halla limitado a ocasiones puntuales donde se requiera un proceso extra para tomar ciertas decisiones críticas (desde un punto de vista táctico o estratégico), pero no está recomendado para las decisiones operacionales por su elevado consumo de recursos.

3.2.2. Máquinas de estados finitos

Tal y como hemos introducido en el módulo anterior, las máquinas de estados finitos (en inglés, *machine finite state*, MEF) nos permiten modelar el comportamiento de un sistema, especificando una serie de estados y de condiciones que deben cumplirse para realizar las transiciones entre ellos. El sistema se encuentra siempre en un estado activo y, a partir de la información que leamos en el sistema y los condicionales de las transiciones, hemos de decidir si debemos cambiar hacia otro estado.

Uso de las máquinas de estado

Las máquinas de estado se han utilizado desde los primeros videojuegos. Por ejemplo, los fantasmas del Pac Man original eran una máquina de estados finitos.

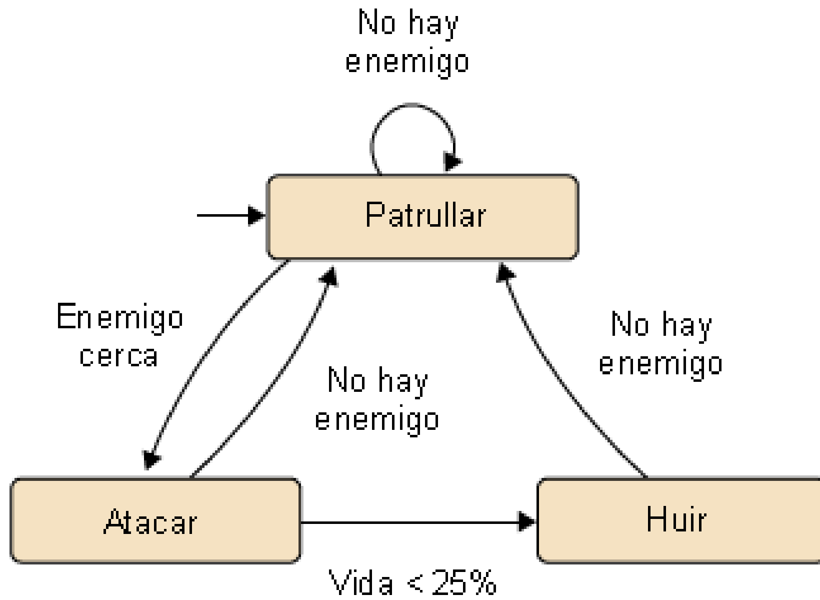
Algunas de las ventajas asociadas a la utilización de MEF para el cálculo de las decisiones de la IA son las siguientes:

- Su simplicidad las hace perfectas para desarrolladores con poca experiencia, ya que se diseñan e implementan de manera rápida.
- En el caso de las MEF deterministas se pueden predecir fácilmente las situaciones que provocan la transición, lo cual permite un mayor control de la depuración de la IA.
- Se trata de un sistema flexible, fácil de extender con nuevos estados y transiciones sin complicar el funcionamiento global.
- Proporcionan una representación gráfica del comportamiento que nos permite saber fácilmente cómo ir de un estado a otro y qué condiciones son necesarias para la transición.
- Los recursos necesarios para ejecutar una MEF son muy pocos.

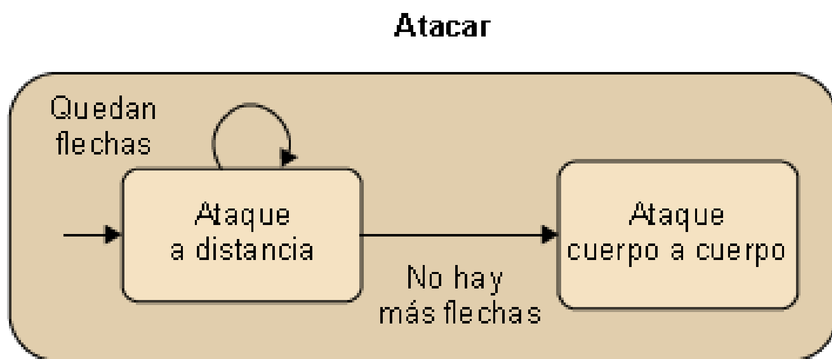
Por otro lado, las desventajas de este tipo de sistemas son las siguientes:

- Normalmente son sistemas muy predecibles, sobre todo en el caso de una MEF determinista.
- Sin un buen diseño podemos tener problemas a la hora de su implementación, sobre todo si se trata de una máquina con muchos estados diferentes.
- Sólo sirve para aquellos casos en los que podemos separar el comportamiento en diferentes estados independientes y donde podemos definir unas transiciones claras entre estados.

Las MEF pueden utilizarse en diferentes niveles de la programación de la IA. Lo más normal es utilizarlas para la programación de decisiones estratégicas o tácticas:



No obstante, podemos asimismo utilizarla para decisiones operacionales, incluso podemos poseer diferentes niveles jerárquicos, en los que dentro de cada estado podemos tener otra MEF completa para las decisiones operacionales:



Reflexión

Como ya hemos comentado, la transición entre estados puede depender de condiciones deterministas (por ejemplo, si nuestra salud es mayor de un 50% debemos cambiar de estado) o probabilísticas (por ejemplo, la probabilidad de cambiar de estado es proporcional a la salud que tengamos).

3.2.3. Árboles de decisión

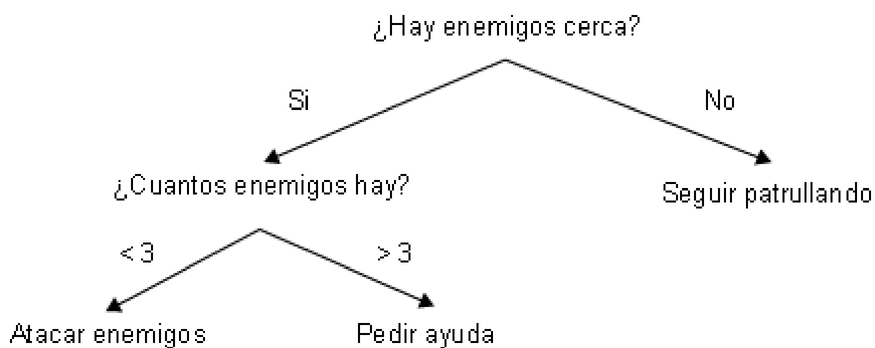
Un árbol de decisión es otro sistema utilizado para analizar las diferentes opciones que podemos llevar a cabo. Se trata de una estructura en forma de árbol donde colocamos todas las decisiones que podemos realizar y sus posibles consecuencias.

Árboles de decisión

En algunos casos se considera que los árboles de decisión son sólo representaciones visuales de un sistema basado en reglas, aunque realmente son muy útiles para representar cierto tipo de decisiones.

La decisión en este tipo de árboles se lleva a cabo recorriéndolo desde la raíz hasta las hojas, para alcanzar así una decisión. El árbol de decisión suele contener cuatro tipos de elementos diferentes:

- Nodos internos deterministas: contienen un test sobre la información de la situación actual que nos permitirá decidir qué rama elegir.
- Nodos internos probabilísticos: dependiendo de un evento aleatorio, decidiremos la siguiente rama.
- Hojas del árbol: representan el resultado que devolverá el árbol de decisión.
- Ramas del árbol: describen los posibles caminos que se extienden de acuerdo con la decisión tomada en los nodos internos.



Los árboles de decisión presentan algunas ventajas interesantes respecto a otros sistemas:

- Son mucho más fáciles de entender e interpretar gracias a la representación gráfica de los otros elementos.
- Podemos decidir el nivel de profundidad de nuestra decisión de una manera más clara, con lo que podemos ajustarla a la capacidad computacional.
- Se puede combinar con otras técnicas de decisión para responder a cada una de las preguntas de los nodos intermedios.

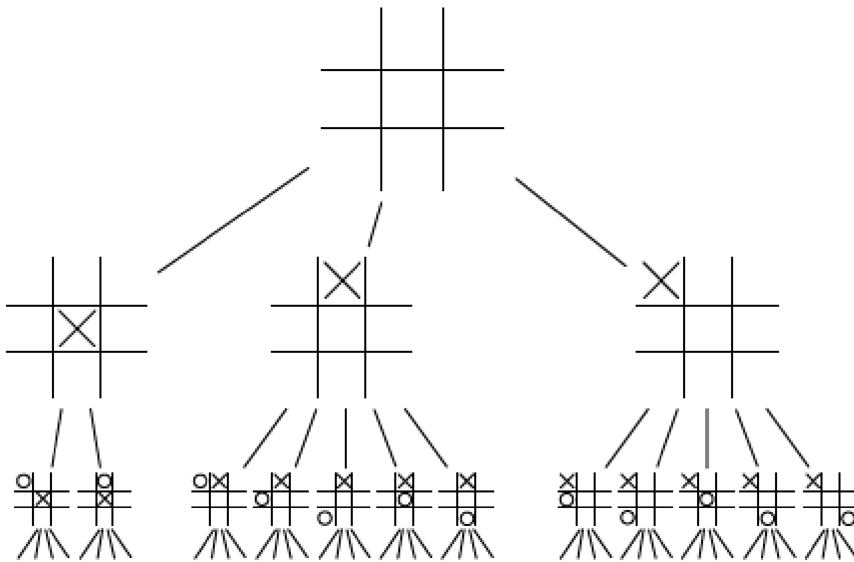
La implementación de los árboles de decisión se realiza en dos fases:

- Fase de diseño: durante la creación del juego hemos de introducir en el árbol todas las condiciones o reglas que vamos a tratar y las acciones que se encuentran en las hojas. Todos estos elementos los guardaremos en una estructura de datos en forma de árbol.

- Fase de análisis: en segundo lugar, debemos programar un algoritmo de análisis del árbol. El algoritmo ha de ir descendiendo por el árbol, evaluando las diferentes condiciones que se encuentre hasta que llegue a una hoja. Debéis tener en cuenta que la evaluación puede ser determinista o probabilística, dependiendo del tipo de nodo interno utilizado.

El algoritmo Minimax

Los árboles de decisión son muy populares para los juegos en los que participan dos jugadores, sobre todo en todos aquellos que se desarrollan encima de un tablero. En este tipo de juegos, cada nivel del árbol representa las acciones que puede tomar un jugador, y el siguiente nivel representa las acciones del oponente en el caso de que se haya ejecutado uno de los movimientos. Esto se repite sucesivamente un cierto de número de pasos, que normalmente están limitados por los recursos del sistema, por lo que, normalmente, no podremos llegar hasta la situación final del juego (ganar, perder o empate, por ejemplo).

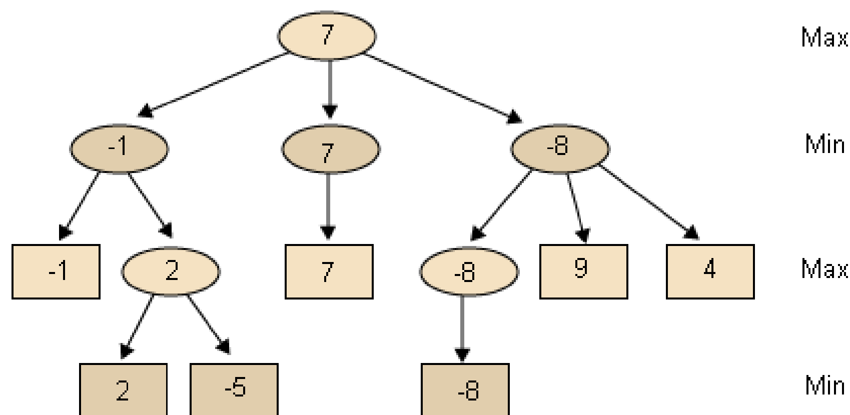


Dentro de los algoritmos que analizan las diferentes opciones que se presentan en el árbol, uno de los más utilizados es el algoritmo Minimax. Este algoritmo intenta buscar la mejor estrategia dentro del conocimiento del árbol, asumiendo que ambos jugadores van a jugar siempre su mejor movimiento.

Nota

El número de niveles máximo del árbol normalmente es bastante pequeño. Por ejemplo, el supercomputador Deep Blue que gano a Garry Kasparov sólo era capaz de introducir en el árbol los 12 siguientes movimientos, para después aplicar una heurística sobre ellos.

El funcionamiento del algoritmo es bastante simple. En primer lugar, lo que debemos llevar a cabo es asignar un valor numérico a cada una de las posibles situaciones del juego y poner este valor en los nodos y las hojas del árbol, utilizando lo que se conoce por una función de utilidad. Uno de los jugadores ha de intentar conseguir el valor máximo (el que le da el mayor beneficio) mientras que el adversario jugará para que el jugador obtenga un valor mínimo (el que le dé 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.



En la figura anterior vemos un ejemplo del funcionamiento del algoritmo Minimax. En el tercer nivel, vemos cómo el jugador "Max" toma el valor 7 porque es el que mayor beneficio le dará. En el segundo nivel, el jugador "Min" elegirá uno de los valores –1, 8 y 7. Finalmente, el jugador "Max" opta por el mayor de todos estos, el valor 7, con la garantía de que en las tres próximas rondas este valor máximo está garantizado.

Reflexión

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 primer nivel estamos buscando el máximo. Analizamos la primera rama y obtenemos como máximo el –1. Analizamos la segunda y obtenemos como máximo el 7, con lo que mejoramos y nos guardamos esta rama como la mejor. Cuando empezamos a recorrer la tercera, ya vemos que el oponente nos va a dar un –8 sin necesidad de seguir mirando los valores 9 ó 4, así que podemos dejar de analizarla y quedarnos con el 7.

3.2.4. Lógica difusa

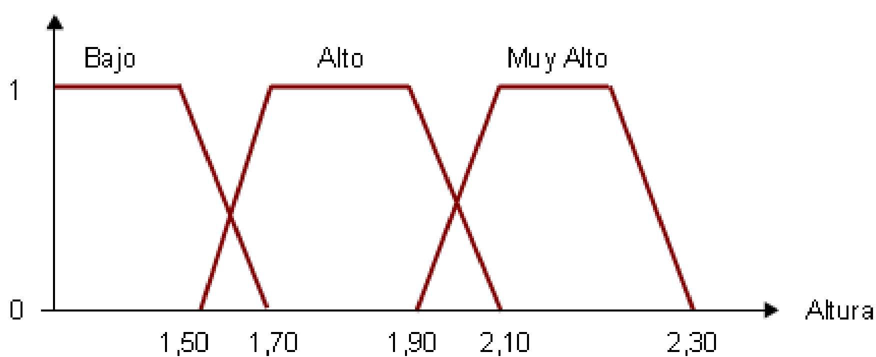
Una proposición en la lógica clásica sólo admite dos valores posibles: verdadero o falso. No obstante, en la vida real las observaciones no son tan claras como para poder afirmar que una proposición es verdadera o falsa, sino que puede haber diferentes puntos de vista que sean relativos al observador.

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.

Ejemplo

Funcionamiento de la lógica difusa: 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.

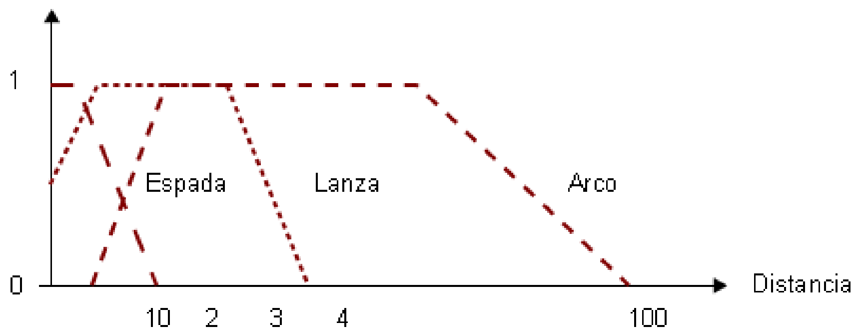
Un valor en la lógica difusa puede encontrarse entre verdadero y falso, pudiendo tener valores de mucho, un poco... Esta conversión de un valor cuantitativo a un valor cualitativo se realiza mediante lo que se conoce por un subconjunto difuso (*fuzzy set*), y representa la probabilidad de que alguien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

- 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.



- 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.

3.2.5. Redes bayesianas

En el caso de que queramos introducir más tipos de incertidumbre en el juego, podríamos utilizar un sistema basado en probabilidades. La idea es muy simple: cuando queremos tomar una decisión asignamos una serie de probabilidades a cada una de las opciones que tenemos a partir de la observación y sobre estas probabilidades tomamos la decisión.

Una de las mejores técnicas para modelar la incertidumbre de las decisiones son las redes bayesianas. Las redes bayesianas son grafos dirigidos y acíclicos que representan la relación entre diferentes variables y sus relaciones de dependencia. Estas redes están basadas en la regla de Bayes.

Grafo acíclico

Un grafo es acíclico si no podemos encontrar un camino de arcos donde un nodo sea origen y destino.

Reflexión

La regla o teorema de Bayes se utiliza para relacionar probabilidades condicionales de dos elementos.

La probabilidad condicional de dos sucesos P(B|A) representa cuál es la probabilidad de que, si ha ocurrido el suceso A, entonces ocurra el suceso B. Matemáticamente se describe mediante $P(B|A) = P(A \cap B)/P(A)$, donde P(A) es la probabilidad de que ocurra el suceso A y $P(A \cap B)$ es la probabilidad de que ocurran los dos sucesos a la vez.

El teorema de Bayes nos permite relacionar las probabilidades condicionales de dos sucesos de la siguiente manera, $P(B|A)\ P(A) = P(A|B)\ P(B)$.

Dos variables son independientes si no existe ninguna relación entre ellas, es decir, si $P(B|A) = P(B)$.

En una red bayesiana, cada nodo representa una condición que tiene asociada una cierta probabilidad y los arcos representan causalidad, es decir, que es necesario que sea cierto el nodo origen para poder analizar la condición del nodo destino. Vamos a estudiar cómo utilizar las redes bayesianas con uno de los ejemplos típicos que podemos encontrar en los libros de IA clásica.

Tenemos un sistema de alarma en casa y sabemos que puede activarse si alguien quiere robarnos, o si un gato se cuela por alguna ventana. A partir de estadísticas y experiencias propias, podemos definir las siguientes probabilidades:

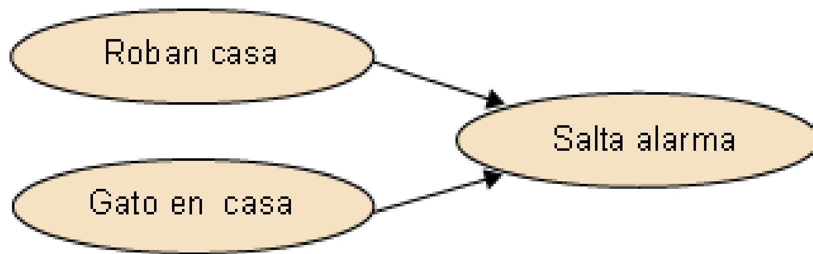
P(R): Probabilidad de que nos roben: 0,001
P(G): Probabilidad de que entre un gato: 0,002

También podemos calcular la probabilidad de que salte la alarma si ocurren los dos eventos a la vez, si sólo ocurre uno de ellos o si no ocurre ninguno:

Tabla

¿nos roban?	¿entra un gato?	P(A|R,G): probabilidad alarma
No	No	0,001
No	Sí	0,75
Sí	No	0,95
Sí	Sí	0,99

Estas probabilidades las podemos resumir en una red bayesiana en la que representamos cómo cada probabilidad de que se produzca un hecho afecta a la probabilidad de que se produzca otro.



Ahora nos podemos hacer la pregunta inversa: si estamos en el trabajo y el sistema de alarmas nos envía un aviso a nuestro teléfono porque se ha activado, ¿qué probabilidad hay de que nos estén robando? Para ello utilizaremos justamente el teorema de Bayes, calculando P(A) = P(A|R,G)*P(R)*P(G):

Tabla

P(R)	P(G)	P(A|R,G)	P(A)
No = 0,999	No = 0,998	0,001	0,0010
No = 0,999	Sí = 0,002	0,75	0,0015
Sí = 0,001	No = 0,998	0,95	0,0009
Sí = 0,001	Sí = 0,002	0,99	0,0001

Como los valores de P(A) no suman nunca 1, podemos normalizar este valor multiplicándolo por un factor para que su suma sea 1, obteniendo así el porcentaje de que no ocurra ninguna de las dos acciones, una de ellas, o las dos. En este ejemplo hemos de multiplicar la última columna por 268 para obtener estas probabilidades: que la alarma suene si nos roban y entra un gato será del 2,9% (porque que se den las dos cosas a la vez es muy poco probable). Si suena porque ha entrado un gato, será del 25,7%. Si realmente nos están robando, será del 42,8%, y que suene sin que pase nada, del 28,6%. Con lo que podemos decir que un 43% de las veces nuestra alarma nos avisará de que roban, y en un 57% de las veces será una falsa alarma.

Ahora podemos complicar un poco más el ejemplo. Imaginemos que, una vez recibimos el aviso, llamamos al vecino y nos dice que había un gato merodeando alrededor de nuestra casa y lo más seguro es que se haya metido en ella, ¿qué probabilidad existe de que nos estén robando?

Tabla

P(R)	P(G)	P(A|R,G)	P(A)	P(A) Normalizado
No = 0,999	No = 0	0,001	0	0
No =0,999	Sí = 1	0,75	0,7492	0,9988

P(R)	P(G)	P(A|R,G)	P(A)	P(A) Normalizado
Sí = 0,001	No = 0	0,95	0	0
Sí = 0,001	Sí = 1	0,99	0,0009	0,0012

La probabilidad de que nos estén robando baja hasta el 0,12%. Si somos capaces de modelar los comportamientos y las acciones de los elementos del juego utilizando redes bayesianas, podemos generar todo tipo de comportamientos más realistas basados en el estudio de las probabilidades y de los elementos que conocemos. Por lo tanto, para diseñar una red bayesiana hemos de realizar tres tareas:

- Definir las interacciones entre los diferentes elementos de la red (qué probabilidades influyen en otras).
- Asignar los valores a cada una de las probabilidades de los posibles eventos. Las probabilidades pueden ser variables dependiendo de la situación y de la información disponible, tal y como hemos comprobado con la extensión del ejemplo.
- Decidir qué preguntas queremos formularle a la red, como en nuestro caso, que queríamos saber cuál era la probabilidad de que hubiera una falsa alarma.

Ejemplo

En el libro *AI Game Programming Wisdom* los autores muestran como ejemplo la detección de un intruso utilizando este tipo de algoritmos. En este caso, nos hablan de un guardián que está patrullando. Éste escucha un sonido, cuyo origen puede ser, con una cierta probabilidad, de unas ratas o, con otra probabilidad, de un enemigo. A partir de la red bayesiana y de las probabilidades de que ocurra cada evento, el soldado decide si ha de ignorar el sonido o no.

Al trabajar con probabilidades, el soldado tendrá un comportamiento más natural. Por ejemplo, si es un ruido flojo, la probabilidad de que piense que es un enemigo es muy baja y muy posiblemente lo ignorará. Si, en cambio, el enemigo realiza un ruido fuerte, la probabilidad de que el guardia piense que es un enemigo aumentará, con lo que la reacción posiblemente será la de ponerse en alerta.

Como se puede apreciar, el uso de las redes bayesianas es técnicamente más complejo que el resto de técnicas vistas en este capítulo, con lo que son recomendables tan sólo para problemas de decisión muy específicos. En otros casos pueden existir mejores soluciones que éstas, más simples e igual de efectivas.

3.2.6. 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.

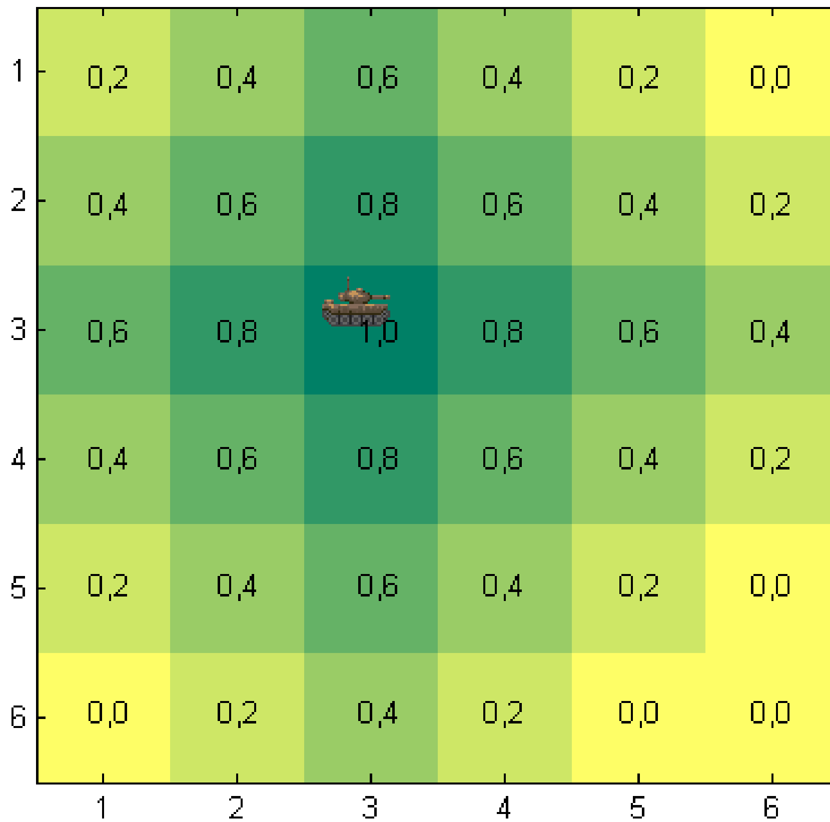
Para crear un mapa de influencia, lo primero que hemos de llevar a cabo es segmentar el mapa del mundo en regiones discretas. Podemos utilizar tanto los *grids* como las mallas poligonales vistas en el módulo anterior y no tienen por qué coincidir con la geografía real del mundo.

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 forma 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 añadimos 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.

Ejemplo

Podemos definir un comportamiento sencillo a modo de ejemplo. Supongamos que el valor de cada celda del mapa de influencia es un valor entre menos uno y uno 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.

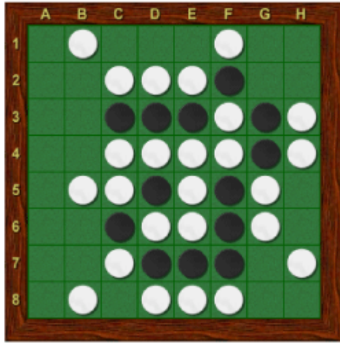
Por otro lado, cada elemento (soldado, tanque, etc.) tiene un valor de fuerza entre menos uno y uno, pero su significado cambia un poco. Si el valor es negativo, indica que tiene problemas y quiere huir del enemigo (podríamos definir otro comportamiento, como ir en busca de ayuda, alimentos, etc.). Si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



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

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

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.

3.2.7. Árboles de comportamiento

Se trata de una técnica relativamente nueva que combina diferentes opciones de las vistas anteriormente y que nos permite añadir más variaciones en los comportamientos de forma flexible, mientras que a su vez optimizamos el rendimiento general de la IA.

Esta técnica intenta solventar la creciente complejidad y variedad de comportamientos que puede tener un elemento en un juego. Por ejemplo, podemos tener un orco en un juego que pueda tener diferentes comportamientos según las siguientes condiciones:

- Si está descansando, buscando o atacando.
- De la cantidad de vida restante.
- De su nivel: si es un jefe, un general o un simple guerrero.
- Dependiendo de dónde se encuentra: en una playa, en un bosque o en una caverna.
- Según su tipo de arma: si lleva lanza, arco o espada.

Y así podríamos continuar con todo lo que se le ocurra a los diseñadores del juego. Implementar todas las posibilidades en un árbol de decisión o una máquina de estados finita nos llevaría mucho tiempo, no sería muy escalable y posiblemente cometeríamos errores difíciles de detectar.

Los árboles de comportamiento basan su funcionamiento en dos elementos principales: los datos codificados que describen el estado del personaje en cada momento y un árbol que usando estos dados es capaz de decidir qué acción llevar a cabo en cada momento.

Los datos del personaje tienen que estar codificados de la forma más simple (si puede ser, de forma numérica) y compacta posible (todos en la misma estructura).

Datos Orco

Arma = 2 // 1 = *Espada*, 2 = *Arco*,...

Vida = 60

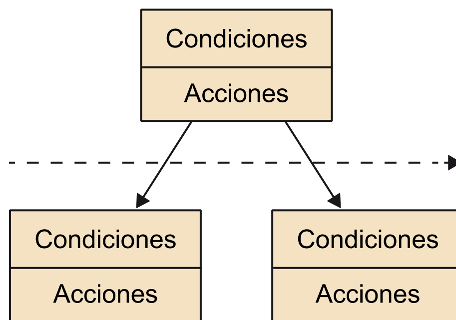
Orcos_cerca = 4

Enemigos_cerca = 0

Tipo_Orco = 1 // 1 = *Soldado*, 2 = *Jefe*,...

...

El árbol de comportamiento está compuesto de nodos que incluyen dos tipos de datos: condiciones y acciones. Las condiciones determinan si el nodo es accesible o no según los datos disponibles, por ejemplo '*IF (ARMA == 1)*'. Las acciones definen el comportamiento especificado si se cumplen las condiciones. Normalmente solo hay acciones definidas en los nodos hoja, aquellos que no tienen hijos colgando.



Esquema básico de un árbol de comportamiento.

El funcionamiento de un árbol de decisión es el siguiente.

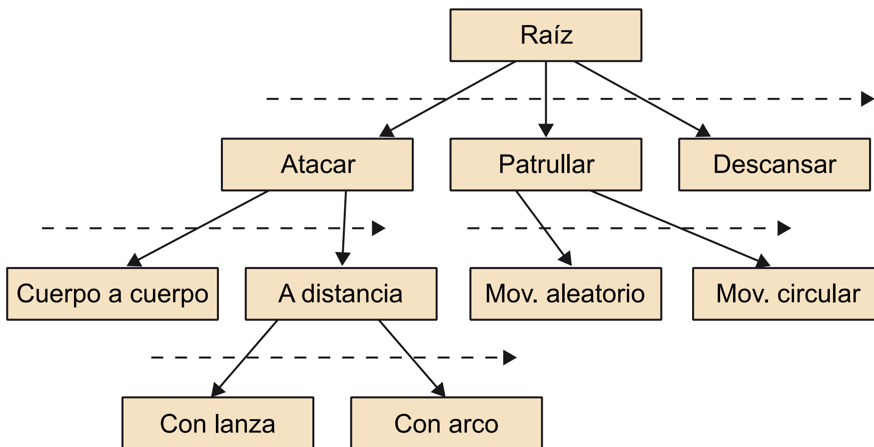
- El sistema evalúa las condiciones de los nodos hijo hasta encontrar uno en que se cumpla. Cuando se encuentra una condición verdadera, ya no es necesario mirar a los otros hijos. Los hijos se evalúan normalmente en el mismo orden (en el esquema anterior de izquierda a derecha), lo que implica de forma implícita que podemos asignar prioridades a los diferentes comportamientos según su ordenación.

Nota

Si existen varios hijos que cumplen su condición, se pueden usar variaciones donde la selección del hijo se puede hacer de forma probabilística o mediante técnicas de tipo ***round-robin***.

- Si el hijo tiene a su vez otros hijos, se repite el proceso. Si no tiene hijos se marca la hoja actual como hoja activa y se ejecuta el comportamiento determinado en las acciones.

- El sistema tiene siempre una hoja activa. Cada vez que hay un cambio en los datos se recalcula todo el proceso hasta determinar cuál es la nueva hoja activa.



Ejemplo de un árbol de comportamiento con sus prioridades.

Los árboles de comportamiento ofrecen varias ventajas respecto a otros sistemas.

1. Permiten realizar modificaciones de forma fácil sin cambios sustanciales en el código. En el ejemplo anterior, si queremos añadir un tipo de arma nueva, crearemos un nodo nuevo y definiremos las condiciones/acciones relacionadas con la nueva arma. Después situaremos este nodo en el árbol junto con las otras hojas de armas, teniendo en cuenta su prioridad.

2. Se pueden encapsular las partes más usadas en subárboles, que se pueden integrar en otros árboles de comportamiento. Esto incluye todas las ventajas relacionadas con programación modular: reducción del tiempo de desarrollo y la complejidad, posibilidad de dividir el trabajo, más fácil de corregir errores, etc.

3. Permiten diseñar tácticas de grupo. Para esto necesitamos un sistema central donde registraremos nuestra disponibilidad a hacer una acción sincronizada/colaborativa (por ejemplo, ayudar a flanquear a un enemigo). También tendremos que añadir en las condiciones necesarias las comprobaciones para ver si se puede entrar en el nodo basándose en lo que esté registrado en el sistema central. En este caso tendremos que avisar a todos los elementos que se tienen que sincronizar que reevalúen su árbol de comportamiento.

4. Técnicas avanzadas de IA

La mayoría de las técnicas que hemos comentado hasta ahora son bastante comunes en la mayor parte de los videojuegos. No obstante, existe otro grupo de técnicas de IA no tan extendidas en los videojuegos, como las redes neuronales o los algoritmos genéticos, que también pueden utilizarse, aunque su aplicación no es tan obvia como las anteriores.

Este grupo de técnicas ofrecen interesantes alternativas para realizar tareas puntuales dentro de la programación de la IA, con lo que están ganando bastante popularidad entre la comunidad de desarrolladores. De cara al usuario, la diferencia más importante es que dotan a los agentes inteligentes de un comportamiento más parecido a la realidad, tanto a nivel individual como colectivo, con lo que se consigue una mejor experiencia de juego.

En este capítulo vamos a tratar tres técnicas que nos permiten el desarrollo de una IA más avanzada mediante técnicas clásicas utilizadas ampliamente y con éxito en otras disciplinas:

- En primer lugar, analizaremos cómo podemos hacer que los elementos inteligentes aprendan qué acciones se deben realizar en cada momento, sin necesidad de tener todas las posibles respuestas programadas dentro del sistema.

- En segundo lugar, vamos a estudiar técnicas que permiten que los elementos inteligentes evolucionen a medida que se desarrolle el juego, de modo que se adapten automáticamente a la dificultad y al entorno del juego.

- Finalmente, analizaremos cómo implementar la comunicación entre agentes para crear comportamientos colectivos más complejos, tales como tácticas de grupo.

Como hemos repetido en varias ocasiones, el uso de estas técnicas dependerá del contexto, ya que son técnicas bastante específicas. Vamos a describir las más utilizadas para el diseño de videojuegos, aunque existen más alternativas que se han utilizado con éxito en otros campos que también se podrían implementar.

4.1. Aprendizaje

Una de las propiedades más importantes que podemos añadir a los elementos inteligentes que participan en el juego es la capacidad de aprender para adaptarse a las condiciones del entorno y así poder actuar de una manera menos mecánica y más humana.

Según el *DRAE*, *aprender* significa "Adquirir el conocimiento de algo por medio del estudio o de la experiencia". En el contexto de los agentes inteligentes que participan en un videojuego, entendemos como aprender ser capaz de asociar una acción con el beneficio (o resultado) obtenido al ejecutarla y almacenar este conocimiento para usarlo posteriormente.

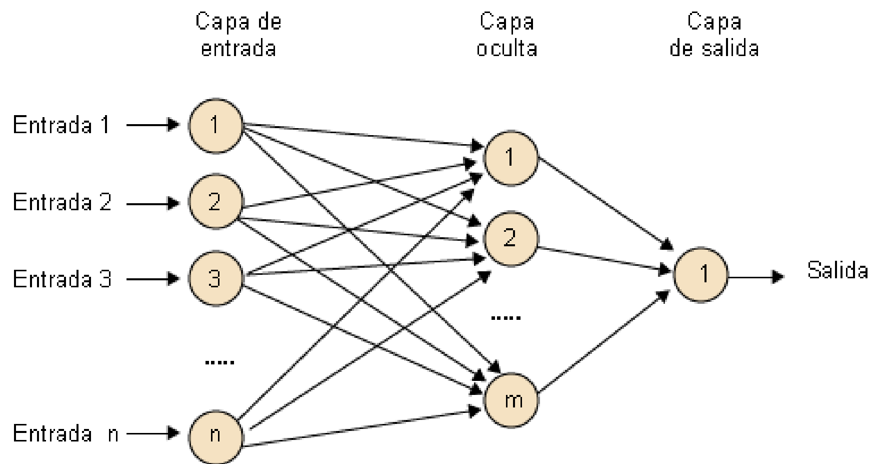
Existen dos tipos principales de aprendizaje automático: el aprendizaje supervisado y el no supervisado. La diferencia entre ellos radica en que en el primer tipo de aprendizaje hay conocimiento de las posibles entradas y salidas del sistema y, en cambio, en el segundo no existe este conocimiento *a priori*. En el caso de los videojuegos se utiliza principalmente el uso de un aprendizaje supervisado, donde los agentes conocen el resultado de sus acciones en el mundo y saben cuantificar el beneficio obtenido al realizarlas.

Existen muchas maneras de introducir la capacidad de aprender en un agente inteligente, pero quizá la más famosa y utilizada ha sido las redes neuronales, ya que nos proporcionan un sistema que intenta emular cómo aprendemos realmente los seres vivos.

4.1.1. Redes neuronales

Nuestros cerebros están compuestos de millones de neuronas entrelazadas entre sí creando un sistema muy complejo capaz adquirir nuevos conocimientos, almacenarlos y tomar decisiones sobre éstos.

Las redes neuronales son un intento de reproducir este tipo de estructura desde un punto de vista muy simplificado. Básicamente, entendemos una red neuronal como un sistema que recibe un conjunto de entradas (percepciones que recibimos del sistema), las procesa por medio de un conjunto de elementos ocultos que se encuentran conectados entre sí (en un sistema parecido a las conexiones de las neuronas) y que produce un resultado de salida (la acción que vamos a tomar).



Cada uno de los elementos (círculos) que vemos en el esquema podemos considerarlo una neurona. Cada neurona tiene una función asociada que nos dice cómo calcular el estado de la neurona a partir del estado en el que se encuentren las neuronas que están conectadas anteriormente a la misma y un peso asignado a cada una de estas conexiones. La salida de una neurona podemos definirla:

$$w_i = \sum_{j=1}^{n} w_j \cdot p_{i\,j}$$

Siendo $w_x$la salida de la neurona x, y $p_{i,j}$el peso asignado a la conexión entre la neurona ***i*** y la neurona ***j***.

Cada vez que hay un cambio en la señal de entrada, esta señal se propaga a través de las diferentes capas ocultas de neuronas (donde se recalcula el nuevo estado para cada neurona) hasta que obtenemos el resultado final.

En las redes neuronales estándar, el valor de una neurona suele ser uno o cero, dependiendo de si se encuentra activa o no. La función asignada a cada neurona genera, a partir de las entradas, un valor continuo entre cero y uno, y lo que hace es fijar, a partir de un umbral de activación, el resultado.

Una segunda opción que tenemos es interpretar el estado de las neuronas como un número (entero o real). Aunque este sistema no refleja la interpretación biológica de una red neuronal, nos permite crear redes más complejas con un mayor rango de posibilidades.

La implementación de una red neuronal tiene dos fases:

- Fase de entrenamiento. En una primera fase, hemos de enseñar a nuestra red neuronal para que sepa relacionar un conjunto de entradas conocidas con una salida esperada que también es conocida. El objetivo de este entrenamiento es definir los pesos de las ecuaciones de estado de todas las neuronas. De esta manera, vamos repitiendo el entrenamiento hasta que demos con los valores que hacen que para las entradas que conocemos el sistema nos dé siempre la salida que queremos.

Nota

Los ejemplos utilizados en la fase de entrenamiento son claves para conseguir que nuestra red nos dé la mejor respuesta para aquellos casos en los que no ha sido entrenada.

- Fase de explotación. Cuando ponemos la red neuronal en un caso real, la red recibe todo tipo de entradas, posiblemente muchas más de las que hemos utilizado durante la fase de entrenamiento. A partir de las fórmulas que tenemos en la red neuronal obtenidas en el entrenamiento, nuestro sistema nos dará la acción que él cree que es la más acertada.

Durante la fase de explotación, la red neuronal puede seguir aprendiendo, ya que la estamos enfrentando a conjuntos de datos de entrada que no había visto con anterioridad. En el caso de que queramos que la red sea moldeable después de la fase de entrenamiento, hemos de permitir que se cambien los parámetros que controlan los estados de las neuronas.

Entrenamiento y aprendizaje de una red neuronal

El método mas utilizado para ajustar los pesos y, por tanto, aprender en una red neuronal se conoce por propagación hacia atrás(en inglés, *back-propagation*). Vamos a estudiar su funcionamiento:

- Para poder entrenar el sistema, primero hemos de asignar un valor numérico a cada una de las acciones que realizamos (por ejemplo, el beneficio económico o la cantidad de vida que le hemos quitado a nuestro enemigo).

- Cuando ponemos un conjunto de entradas, medimos el valor numérico esperado de esta acción con el resultado de la red neuronal, y calculamos la diferencia entre los dos, obteniendo la cantidad de error de nuestra acción.

- Usamos este error para calcular cuánto deberíamos cambiar los pesos dentro de las ecuaciones de las neuronas. Un error pequeño supondrá tan sólo pequeños ajustes, mientras que, para corregir un error grande, necesitaremos cambios más sustanciales en los pesos.

- Este proceso se repite indefinidamente hasta que consigamos que el error se encuentre por debajo de un valor crítico, donde consideramos que aunque el resultado no es el óptimo esperado, es un valor suficiente para dar la decisión de la red neuronal como buena.

En el método de *back-propagation*, el ajuste de los pesos se realiza desde la última capa a la primera, ya que se considera que los pesos que influyen más en el resultado final son los que están más cerca de la salida.

Usos de una red neuronal en los videojuegos

En el caso particular del desarrollo de videojuegos, las redes neuronales ofrecen algunas ventajas respecto a otras técnicas de IA tradicional:

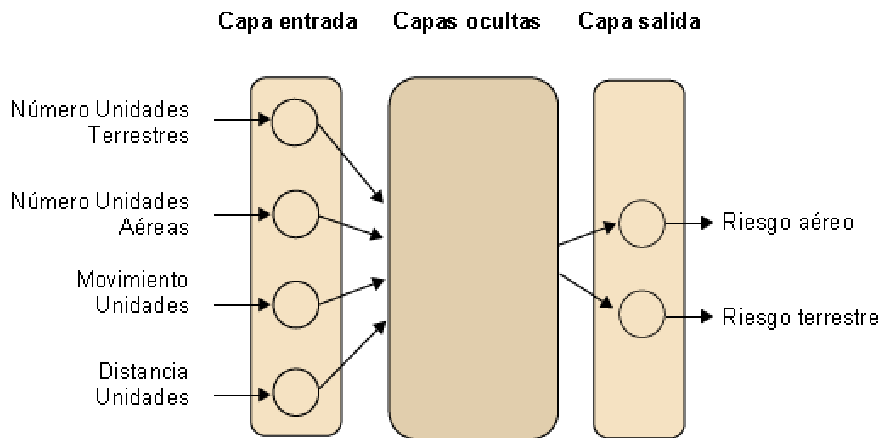
- Permite simplificar la programación de máquinas de estado o sistemas de reglas complejos para decidir el comportamiento del juego. En lugar de esto, podemos implementar una red neuronal más simple y entrenarla para que tome las decisiones de una manera más razonable y rápida.

- Ofrecen la posibilidad de que la IA de los agentes se adapte a medida que se desarrolla el juego, aprendiendo de las acciones que va tomando el agente, y por tanto adaptando la dificultad del juego a las acciones del usuario.

Vamos a presentar algunos ejemplos de cómo podemos utilizar con éxito una red neuronal para tareas concretas de la IA. Veremos tres casos diferentes donde el aprendizaje se realiza durante el desarrollo del juego, mientras el jugador está jugando o en ambos casos. No vamos a entrar en detalles de la implementación de una red en cada caso.

Uno de los principales usos de una red neuronal es para implementar el sistema de decisión del comportamiento de los agentes inteligentes de una manera más rápida y natural. Algunos juegos de coches (por ejemplo, Colin McRae Rally) utilizan las redes neuronales para implementar el modo como conducen los oponentes. Estas redes neuronales se entrenan previamente a partir de cómo conducen los desarrolladores o algunos jugadores expertos, y han conseguido un gran resultado con un comportamiento más impredecible y efectivo.

Otro uso posible de las redes neuronales es el de ayudarnos a decidir el nivel de peligro de una situación. Vamos a analizar su utilización en un juego de estrategia. Podemos crear una red neuronal que reciba como entradas de nuestra red el estado de las unidades de los jugadores: número de unidades terrestres, número de unidades aéreas, movimiento de estas unidades, distancia a los elementos del jugador, etc. El valor de salida de nuestra red puede ser el peligro potencial que tienen los jugadores, tanto aéreo como terrestre.



El entrenamiento de esta red se puede producir previamente para tener una idea básica de qué situaciones entablan alto o bajo riesgo. No obstante, también es interesante que esta red se pueda ir adaptando al modo de jugar de su oponente. Por ejemplo, si vemos que después de varias situaciones que la red considera de alto riesgo no ha habido un ataque, podemos reestructurar la red para que se adapte a este comportamiento específico del jugador.

Finalmente, uno de los mejores usos de las redes neuronales es el de implementar la capacidad de anticipar las acciones de los jugadores. Vamos a poner como ejemplo el caso particular de los juegos de lucha. En estos juegos existe muchas veces la posibilidad de encadenar ataques que consiguen mucho más daño que un ataque normal. Lo que queremos es que la IA sea capaz de anticipar cuál será el siguiente golpe que hará el usuario a partir del historial de golpes que ha utilizado con anterioridad. Así podremos anticiparnos al mismo y contraatacar, bloquear o esquivar el ataque que suponemos que va a realizar.

Para esta tarea podemos programar una red neuronal que recoja los tres últimos golpes y lleve a cabo una predicción de cuál va a ser el siguiente golpe. El entrenamiento de esta red se realiza mientras se juega, de manera que la red va aprendiendo poco a poco los patrones de comportamiento del jugador. Esto añadirá un mayor nivel de realismo al combate y obligará al jugador a que aprenda nuevas maneras para ganar el combate.

4.2. Evolución

Tal y como han demostrado los científicos, los seres vivos evolucionan mediante un proceso denominado selección natural, en el que sólo aquellas especies que se adapten a su entorno sobreviven a lo largo del tiempo.

La teoría de la evolución explica cómo las características que hacen fuerte a una especie se perpetúan a las siguientes generaciones. En el caso de los seres vivos, estas características se encuentran codificadas en los cromosomas.

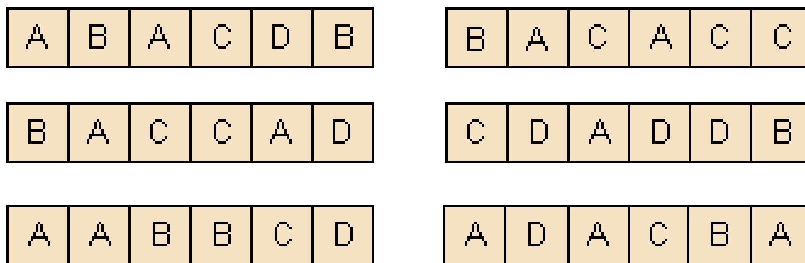
En el ámbito de los videojuegos, la evolución nos proporciona un mecanismo para que los agentes inteligentes se adapten mejor al entorno que los rodea, seleccionando aquellos agentes que se desenvuelven mejor para su posterior reproducción.

4.2.1. Algoritmos genéticos

Los algoritmos genéticos nos permiten modelar el proceso de selección natural para poderlo aplicar a la resolución de todo tipo de problemas. En nuestro caso, vamos a explicar una posible aplicación de éstos: cómo desarrollar un conjunto de agentes inteligentes que se adapten de manera automática al nivel del jugador.

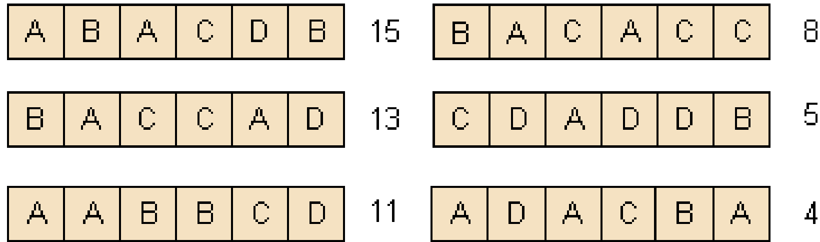
Los algoritmos genéticos también tienen como base de su funcionamiento un cromosoma. En este caso, un cromosoma describe una serie de atributos o propiedades de un elemento perteneciente al sistema. Por otro lado, nuestro sistema se compone de varios elementos descritos cada uno por un cromosoma particular, lo que denominamos la población del sistema.

El funcionamiento simplificado de un algoritmo genético se puede resumir en cuatro fases. En primer lugar, se crea una población inicial donde cada cromosoma tiene unos atributos asignados de manera aleatoria.

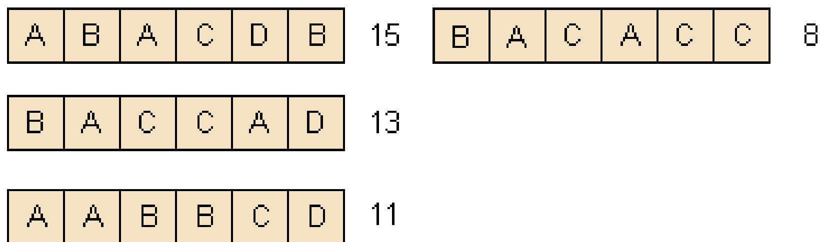


A partir de aquí, el sistema entra en su fase evolutiva, que se repite continuamente:

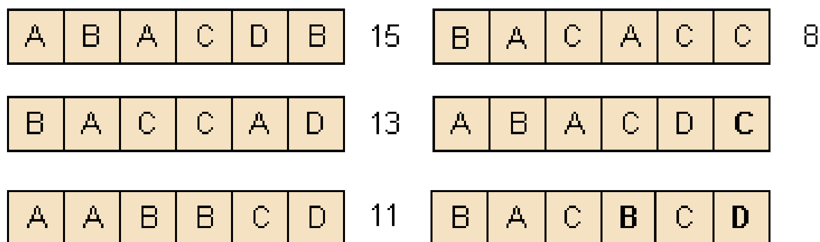
- Lo primero que hacemos es ordenar nuestra población a partir de su *fitness*. Definimos el *fitness* de cada componente como su adaptación al entorno y se calcula a partir de una función que puede tener varios parámetros (beneficios obtenidos, tiempo de vida, etc.). Un elemento que se desarrolle muy bien en este entorno tendrá un *fitness* muy alto, mientras que un elemento débil que no se haya adaptado tendrá un *fitness* muy bajo.



- A partir de esta ordenación de *fitness*, aplicamos un proceso de selección natural en el que eliminamos aquellos elementos más débiles de nuestro entorno. Todos aquellos elementos que no hayan superado un *fitness* mínimo son descartados, mientras que los otros se mantienen en la población.



- Finalmente, volvemos a crear los elementos que hemos eliminado a partir de reproducir los elementos que se han quedado en la población. La reproducción consiste en copiar los cromosomas de los mejores y aplicar lo que se conoce por mutaciones. Una mutación es un cambio en algunos de los atributos del cromosoma, ya sea combinando dos cromosomas de dos diferentes elementos o cambiando aleatoriamente el valor de algún atributo del cromosoma.



Aplicación de los algoritmos genéticos a los videojuegos

En los videojuegos, los algoritmos genéticos se suelen utilizar cuando existe bastante incertidumbre sobre las características del jugador. En este caso, es muy difícil diseñar los adversarios para que sean capaces de encontrarse al mismo nivel que cualquier jugador.

Vamos a estudiar un caso en particular, el de los juegos de rol. En este tipo de juegos, los jugadores pueden elegir personajes con diferentes clases, habilidades y atributos. Además, los personajes pueden llevar un gran número de armas, armaduras y otros complementos que también cambian sus propiedades.

Por lo tanto, en este tipo de juegos los enemigos deben ser capaces de proporcionar un reto que sea difícil independientemente de las combinaciones posibles de atributos. Una posibilidad es modelar los comportamientos de los enemigos mediante el uso de un algoritmo genético, de modo que aquellos comportamientos que se adapten mejor a los atributos del jugador serán los que sobrevivirán, mientras que los comportamientos que resulten inútiles desaparecerán.

El primer paso para poder utilizar el algoritmo genético es codificar la información que queremos que "evolucione" dentro del cromosoma. En el caso que tratamos, podemos incluir por ejemplo la respuesta que debe tener el agente inteligente para cada tipo de acción: qué hacer si nos ataca cuerpo a cuerpo, qué hacer si nos lanza un hechizo, qué hacer si se aproxima, qué hacer si se aleja, cuándo huir, cuándo pedir ayuda, etc.

Cuando empiece el juego, el sistema de IA creará un grupo de adversarios básicos con unos comportamientos basados en los cromosomas que tengamos como definidos inicialmente. Cada vez que uno de estos adversarios entre en juego, el sistema calculará cuál ha sido su *fitness* mirando la efectividad de usar el comportamiento que tenga respecto a la estrategia del jugador. A partir de la escala de estos *fitness*, el sistema irá eliminando los peores y los irá sustituyendo con aquellos más adaptados, hasta que se llegue a un comportamiento que sea capaz de contrarrestar de la mejor manera todas las posibles acciones del jugador.

Reflexión

Otra característica importante de los algoritmos genéticos es que permiten que los agentes inteligentes se adapten a los cambios en el entorno. Siguiendo el ejemplo del rol, si al principio el jugador es mago, los adversarios pueden evolucionar hacia técnicas que esquiven la magia, pero si de repente el jugador cambia y se hace guerrero, veremos que el comportamiento de los agentes cambiará para adaptarse al nuevo tipo de jugador, ya que los comportamientos anteriores ahora recibirán un *fitness* mucho más bajo.

4.3. Comportamientos colectivos

En muchas situaciones, los adversarios de un juego no actúan de modo individual, sino que forman parte de un colectivo que intenta conseguir un objetivo global. Para poder conseguir este objetivo, es necesario que los individuos colaboren entre sí para mejorar sus probabilidades de éxito.

En los juegos actuales, cada vez es más común observar este tipo de comportamientos colectivos en los que los elementos colaboran entre ellos, se coordinan e intentan sacar ventaja de la superioridad numérica. Incluso en algunos casos se utilizan estrategias militares para poder dar mucho más realismo a los jugadores.

Dentro de la IA han existido desde el principio varios métodos para implementar inteligencias colectivas o distribuidas. En el caso de los videojuegos, el más usado son los sistemas multi-agente.

4.3.1. Sistemas multi-agente

Un sistema multi-agente es una de las técnicas de la IA más utilizadas para modelar el comportamiento de un conjunto de agentes, poniendo más énfasis en el objetivo global de todo el sistema que en el objetivo individual de cada uno de los agentes que lo componen.

Aunque no existe una definición formal y precisa de lo que es un agente, por lo general, éstos son vistos como entidades inteligentes, que existen dentro de cierto contexto o ambiente, y que se pueden comunicar por medio de un mecanismo de comunicación inter-proceso, usualmente un sistema de red, utilizando protocolos de comunicación.

Por lo tanto, un sistema multi-agente requiere tres componentes claves:

- Una inteligencia local básica implementada en cada agente que le permite actuar de manera autónoma. Esta inteligencia puede utilizar cualquiera de las técnicas explicadas anteriormente y, básicamente, explica cómo ha de reaccionar el agente a partir de las percepciones que recibe del sistema.

- Un sistema de comunicación que permite enviar mensajes entre los agentes. Para la implementación de esta comunicación podemos utilizar el sistema de mensajería propio del sistema operativo (por ejemplo, el sistema de mensajes de Windows) o podemos diseñarnos nuestro propio sistema de mensajes que esté adaptado a los agentes que tenemos.
  Si optamos por hacernos nuestro propio sistema de mensajería, debemos acordarnos de que hemos de diseñar dos elementos:

- Un gestor de mensajes; es decir, un componente que se encargue de que el mensaje se distribuya entre agente origen y destino de una manera rápida y eficiente.

- Los tipos de mensajes y sus posibles contenidos.

• Un sistema que permita reaccionar a los agentes cada vez que reciban un mensaje. Esta reacción provocará que el agente se deje de comportar de manera individual y pase a actuar en favor del grupo. Por ejemplo, si el agente recibe una llamada de socorro de otro agente cercano, cambiará su rutina, buscará un camino para llegar al agente y después decidirá cómo actuar una vez se encuentre allí.

Uno de los grandes debates que existe cuando implementamos un sistema multi-agente es si es necesaria la existencia de una entidad central que coordine las acciones de todos.

Si tenemos un agente central que coordine las acciones de todos, es mucho más fácil implementar el comportamiento colectivo, ya que este agente central es el que decide la estrategia y va enviando a cada uno de los agentes las ordenes que debe seguir. Este agente necesita una visión más global que los otros.

Si no existe un agente central, son los propios agentes los que se comunican entre ellos. En este caso, los agentes sólo pueden enviar mensajes a aquellos que se encuentran dentro de su radio de conocimiento, y la coordinación debe realizarse entre los propios agentes.

La idea inicial de los sistemas multi-agente es que no debería existir esta unidad central, sino que todos los agentes deberían encontrarse en el mismo nivel. No obstante, dado que en un juego buscamos conseguir una sensación de realismo óptima utilizando el mínimo número de recursos, no es mala idea utilizar un coordinador central de agentes. Dejaremos la elección del tipo más adecuado a las características de cada caso particular.

Un código usando el lenguaje Python de un agente que define el comportamiento de un soldado:

```
def soldier_update(soldier):
     action = Action.WAIT
     neighbors = SoldierManager.getNeighborsOf(soldier)
     for neighbor in neighbors:
          if neighbor.friend and neighbor.life < 0.5 and neighbor.enemies > 0:
               action = Action.bestOf(action, Action.helpTo(neighbor))
          elsif neighbor.enemy and neigbor.life < 0.8 and neighbor.friends < 2:
               action = Action.bestOf(action, Action.attachTo(neighbor))
     soldier.nextAction(action)
```

Resumen

En este módulo hemos estudiado qué es la Inteligencia Artificial y cómo nos puede servir para aportar comportamientos parecidos a los humanos a los diferentes retos que incluyamos en nuestros videojuegos.

Hemos repasado una serie de algoritmos que nos permiten mover elementos por terrenos concretos (desde un tablón de ajedrez a un edificio con habitaciones). Estos algoritmos los hemos clasificado en:

- Movimientos cíclicos y basados en patrones. Donde hemos conocido las curvas *spline* y cómo movernos entre varios puntos.

- Búsqueda de caminos. A partir de un grafo, hemos aprendido a encontrar un camino entre dos puntos. Los algoritmos que hemos tratado son: recorrido en profundidad, recorrido en amplitud, Dijkstra y A*.

- Movimientos complejos. Tenemos una serie de elementos en el terreno de juego y queremos que se comporten de una determinada manera. Hemos estudiado cómo interceptar, perseguir, huir, mover un grupo según unas características (*flocking*) y definir un movimiento de varios individuos con una cierta coreografía.

Pero no nos sirve de nada todo lo que hemos estudiado si no podemos aplicarlo. Para que podamos calcular un camino mínimo a un elemento, primero hemos de decidir que ese elemento tiene que ir de una posición a otra. El tercer punto que hemos estudiado trata esto: la toma de decisiones.

Asimismo, hemos visto varias técnicas utilizas en IA para tomar decisiones:

- Sistema de reglas. Es la técnica más básica. A partir de una serie de entradas, tenemos un algoritmo que nos genera la acción que debemos aplicar (si llueve, abrimos el paraguas).

- Máquinas de estados finitos. Podemos modelar una serie de estados y, a partir de una serie de entradas, ir modificando el estado actual y actuar en consecuencia.

- Árboles de decisión. En este punto vimos algoritmos que se utilizan bastante en juegos de tablero y en los que hay dos contrincantes (por ejemplo, el algoritmo Minimax).

- Lógica difusa. Donde ya no tratamos en términos de cierto o falso, sino que a cada evento se le aplica un porcentaje y tomamos la decisión a partir de cálculos de probabilidad.

- Redes bayesianas. Vimos que es una de las mejores técnicas para modelar la incertidumbre de las decisiones. Con este tipo de redes simulábamos relaciones de dependencia utilizando el teorema de Bayes.

- Mapas de influencia. Generábamos un mapa de celdas sobre nuestro terreno y a cada una le asignábamos un valor numérico que nos informaba de un determinado aspecto del terreno (tenía enemigos cerca, recursos naturales, etc.)

En el último punto hemos tratado las técnicas avanzadas utilizadas en la IA. Estas técnicas están basadas en el aprendizaje (redes neuronales), en la evolución y selección natural (algoritmos genéticos) y en los comportamientos colectivos (sistemas multi-agente).

Actividades

1. ¿Qué técnica de IA utilizaríais para construir un algoritmo que jugase al Connecta-4?

2. Tenemos una serie de agentes que salen de un tren, ¿qué algoritmo modelaríais para que entre todos salgan correctamente y de manera ordenada por las dos salidas del andén?

3. Calculad el movimiento que deben realizar los aviones que componen un escuadrón para que pasen entre las siguientes formaciones:

Más información en

http://
www.patrullaaguila.com

4. Estudiad qué tipo de red neuronal deberíais implementar para poder jugar al juego del Tic-Tac-Toe e ir aprendiendo de los errores.

Glosario

algoritmo genético *m* Técnica que permite solucionar problemas o evolucionar una población a partir de implementar el método de selección natural de las especies.

algoritmo Mimimax *m* Algoritmo que nos permite estudiar la mejor estrategia en un juego entre dos oponentes.

back propagation *f* Véase propagación hacia atrás

cronosoma *m* Cada uno de los componentes de un algoritmo genético que guarda la información de los atributos.

fitness *m* Valor que determina cómo se adapta una especie en un entorno.

flocking *m* Sistema que permite mover colectivamente un grupo de elementos de manera coordinada.

lógica difusa *f* Rama de la lógica que añade la posibilidad de que una proposición pueda tener otros valores aparte de verdadero o falso.

mapa de influencia *m* Representación discreta del mundo que incluye la importancia estratégica de cada zona del mismo.

movimientos de dirección *m* Comportamientos para interceptar, perseguir o huir a un elemento móvil. *en* steering behavior

poda alfa-beta *f* Sistema que nos permite reducir el tiempo de análisis de un árbol Minimax.

propagación hacia atrás *f* Técnica de las redes neuronales para que podamos aprender a partir de los errores cometidos en las decisiones tomadas. *en* back propagation

red neuronal *f* Representación artificial de la conectividad de las neuronas del cerebro para intentar emular el funcionamiento de las mismas.

sistema determinista/no determinista *m* En un sistema determinista no influye el azar en el resultado, mientras que un sistema no determinista sí.

sistema experto *m* Sistema de decisión basado en reglas que permite tomar decisiones a partir de inferencias sobre la una determinada situación.

sistema multiagente *m* Sistema en el que diferentes agentes o entidades inteligentes interactúan entre sí enviándose mensajes y comunicándose para realizar una tarea conjunta.

steering behavior *m* Véase movimientos de dirección

subconjunto difuso *m* Descripción de las probabilidades de asignar un valor cualitativo a uno cuantitativo.

teorema de Bayes *m* Teorema que nos describe cómo relacionar la probabilidad condicional de dos variables.

Bibliografía

Bourg, D. M.; Seemann, G. (2004). *AI for Game Developers*. O'Reilly.

Buckland, M. (2002). AI *Techniques for Game Programming*. Thomson Course Technology.

Buckland, M. (2005). *Programming Game AI by Example*. Wordware Publishing.

Champandard, A. (2003). *AI Game Development: Synthetic Creatures With Learning and Reactive Behaviors*. New Riders.

Deloura, M.; Rabin, S.*AI Game Programming Wisdom*. Volumes 1 to 4. Charles River Media.

Funge, J. (1999). *AI for Computer Games and Animation: A Cognitive Modeling Approach*. AK Peters, Ltd.

Millington, I. (2006). *Artificial Intelligence for Games*. Academic Press.

Russell, S.; Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Sanchez-Crespo Dalmau, D. (2003). *Core Techniques and Algorithms in Game Programming*. Ed. New Riders.

Schwab, B. (2004). *AI Game Engine Programming*. Charles River Media.

Smed, J.; Hakonen, H. (2006). *Algorithms and Networking for Computer Games*. Wiley.