

# Implementación de nodos consulta en árboles de comportamiento \*

Ismael Sagredo-Olivenza, Gonzalo Flórez-Puga  
Marco Antonio Gómez-Martín and Pedro A. González-Calero

Departamento de Ingeniería del Software e Inteligencia Artificial  
Universidad Complutense de Madrid, España

**Abstract.** Los árboles de comportamiento son la tecnología más utilizada en videojuegos comerciales para programar el comportamiento de los personajes no controlados por el jugador, aunando una gran expresividad, con cierta facilidad de uso que permite la colaboración entre programadores y diseñadores de videojuegos.

En este artículo describimos un nuevo tipo de nodos en los BTs que representan consultas que devuelven sub-árboles en tiempo de ejecución y explicamos cómo se pueden integrar en Behavior Bricks, un sistema que hemos desarrollado para la creación y ejecución de BTs.

## 1 Introducción

La creación de comportamientos para personajes con controlados por el jugador (en inglés NPC, *Non Playable Characters*) involucra, entre otros, a los diseñadores y los programadores de inteligencia artificial (o usando las siglas inglesas, AI). Los diseñadores son los encargados de idear y describir las reglas del juego (¿Cómo se juega? y ¿A qué se juega?) con el objetivo de que éste sea divertido, mientras que los programadores son los encargados de llevar esas ideas a cabo. De entre todas las tareas de diseño, la especificación del comportamiento de los NPCs es una parte vital ya que afecta directamente a cuestiones como la dificultad o las mecánicas de juego [1]. En este proceso de intercambio de ideas entre ambos se producen problemas de entendimiento y comprensión por falta de una formalización de lo que debe hacer el NPC o cómo se debe comportar. Dicho proceso, que es común a la mayoría de problemas en la fase de análisis de la ingeniería del software, se agrava en el desarrollo de videojuegos ya que la especificación ni siquiera está completa cuando se inicia el desarrollo del juego y ésta va evolucionando con el tiempo, para hacer que el juego sea divertido o tenga una correcta aceptación entre el público.

En este contexto con requisitos tan volátiles, es crucial minimizar en lo posible los problemas derivados de la comunicación entre programador y diseñador, intentando darle al diseñador las herramientas necesarias para que sea él mismo el que implemente los comportamientos, haciendo que el programador intervenga lo menos posible. Un editor visual que le ayude a programar comportamientos de

---

\* Financiado por el Ministerio de Educación y Ciencia (TIN2014-55006-R)

forma gráfica, sobre un modelo de ejecución fácil de comprender puede ayudar a esta tarea. Tradicionalmente se han utilizado muchas técnicas que intentan presentar la lógica de estos comportamientos de una forma más simple para el diseñador o para programadores menos cualificados. Desde utilizar lenguajes de script, más sencillos de entender, para extender los comportamientos, hasta utilizar modelos de ejecución como las máquinas de estado o árboles de comportamiento que pueden ser representados gráficamente y además manejan conceptos más cercanos a la forma de pensar de un diseñador. En los últimos años, uno de los modelos más utilizados son los llamados árboles de comportamiento (o en inglés *Behavior Trees* o BTs) por su versatilidad sobre otras técnicas como describiremos en la sección 2. Tanto esta técnica como las tradicionales máquinas de estado, utilizan un conjunto de tareas primitivas de bajo nivel que son las que interactúan con el entorno del NPC realmente, ya que el ejecutor simplemente decide cuales de todas ellas debe ejecutar en un instante concreto. El problema surge cuando el juego va creciendo y se van creando nuevas tareas primitivas que van afectando a más NPCs, por lo que añadir una nueva forma de realizar una tarea, puede implicar modificar el comportamiento de varios tipos de NPCs. O en un caso aún más extremo, que ni siquiera el diseñador sepa expresar, usando las herramientas, qué es lo que debe hacer el NPC o cómo debe hacerlo. En estos casos, se necesita un proceso iterativo de prueba y error que implica una modificación de la especificación tanto en el lado del diseñador, que debe cambiar su diseño teórico del comportamiento, como en el del programador, que debe cambiar la implementación de dichos comportamientos, añadir nuevas tareas primitivas, etc.

En este artículo se describe una extensión de los BTs que facilita el proceso iterativo de diseño del comportamiento. Mediante técnicas de razonamiento basado en casos (*Case-based reasoning*, CBR), permite a los diseñadores crear comportamientos aportándoles ejemplos. Estos ejemplos pueden generarse bien interactivamente entrenando al NPC, o bien a partir de diferentes implementaciones de la tarea a aprender, usandolas en diferentes contextos. La decisión de cuál de las posibles alternativas se ejecutará, se relega al momento de ejecución. Además de asistir al diseñador, esta técnica permite crear comportamientos menos previsible desde el punto de vista del jugador, lo que incrementa la sensación de que los NPCs tienen un comportamiento más humano y menos predecible. Adicionalmente, este nuevo modelo de ejecución diferido soporta otras características como permitir que los comportamientos soporten nuevas formas de resolver una tarea sin modificar su estructura o crear comportamientos que vayan aprendiendo con el tiempo la mejor forma de resolver una tarea.

El resto del artículo se estructura de la siguiente manera: en la sección 2 se describe qué son los árboles de comportamiento, en la sección 3 se revisa el trabajo relacionado, en la sección 4 se explica qué es Behavior Bricks, nuestra herramienta de creación de comportamientos, en la sección 5 mostramos nuestra propuesta de extensión de BT y Behavior Bricks y finalmente en la sección 6 describimos las conclusiones y trabajo futuro.

## 2 Behavior Trees

Los BTs [2] modelizan la inteligencia artificial de los NPCs de una forma similar a las máquinas de estados (*Finite State Machine* o FSM) [3] que tradicionalmente se han utilizado con este fin. La diferencia principal entre ambos radica en que los BTs son mucho más escalables ya que en las máquinas de estados, el número de las transiciones entre los estados se dispara cuando el comportamiento a implementar crece. Además, los BTs se adaptan mejor a las necesidades de los diseñadores ya que éstos suelen preferir modelos de ejecución que permitan describir los comportamientos guiados por objetivos o metas a conseguir [4]. Los BTs reemplazan las transiciones de las máquinas de estado con información de control adicional mediante unos nodos intermedios, que son los que deciden el orden en el que deben ejecutarse las tareas primitivas. Estas tareas primitivas son aquellas partes del comportamiento que tienen acceso al entorno virtual al que pertenece el NPC y pueden ser de dos tipos:

- Acciones: Son tareas que modifican de alguna forma el entorno del NPC, por ejemplo moverlo, atacar a un enemigo, saltar, etc. Las acciones se sitúan en los nodos terminales (hojas) de los BTs.
- Condiciones: Simplemente comprueban el estado del entorno sin modificarlo. Estas tareas pueden usarse en los nodos hoja de los BTs o bien en ciertos nodos intermedios especiales que toman decisiones de control.

Cuando la ejecución de un nodo termina, éste notifica a su nodo padre si la ejecución de la tarea tuvo éxito (Success) o no (Failure). Los nodos intermedios utilizarán esta información proveniente de sus nodos hijo para tomar la decisión de cuáles serían los siguientes nodos a ejecutar (dependiendo del tipo de nodo intermedio). Algunos de los nodos intermedios más utilizados son los siguientes:

- Secuencia: ejecuta una lista de comportamientos hijo del nodo en el orden que se han definido. Cuando el comportamiento que está ejecutándose actualmente termina con éxito, se ejecuta el siguiente. Si termina con fallo, él también lo hace. Devolverá Success si todos se han ejecutado con éxito.
- Selector: tiene una lista de comportamientos hijo y los ejecuta en orden hasta encontrar uno que tiene éxito. Si no encuentra ninguno, termina con fallo. El orden de los hijos del selector proporciona el orden en el que se evalúan los comportamientos.
- Parallel: ejecuta a la vez todos sus comportamientos hijo. Esta ejecución no debe necesariamente ser paralela (en varios núcleos), sino que puede realizarse entrelazada; lo importante es que ocurre simultáneamente dentro de la misma iteración del bucle de juego. Los nodos parallel pueden tener como política de finalización la del nodo secuencia (acabar si todos los hijos lo hacen), o la de los selectores (acabar si uno de sus nodos hijos lo hace).
- Selector con prioridad: los nodos selector llevan incorporada una cierta prioridad en el orden de ejecución, pero no re-evalúan nodos que ya han terminado. El selector con prioridad (priority selector) se diferencia del selector

normal en que las acciones con más prioridad siempre intentan ejecutarse primero en cada iteración. Si una acción con mayor prioridad que otra que ya se estaba ejecutando puede ejecutarse, se interrumpe la acción que se estaba ejecutando anteriormente.

- Decoradores: son nodos especiales que sólo tienen un hijo y que permiten modificar el comportamiento o el resultado de ese hijo. Algunos ejemplos son decoradores que repiten la ejecución del hijo un número de veces determinado, los que invierten el resultado del hijo (de éxito a fracaso y viceversa) o aquellos que poseen una condición adicional de forma que se ejecutará o no el hijo en base a si se cumple esa condición.

### 3 Trabajo relacionado

El Razonamiento basado en casos (en inglés *Case-based reasoning*, CBR) [5] es el proceso de solucionar nuevos problemas basándose en las soluciones de problemas anteriores y es un conjunto de técnicas ampliamente utilizadas en multitud de dominios en inteligencia artificial. En CBR necesitamos mantener una base de casos en la que debemos almacenar qué tarea o acción se ejecutó en el pasado y en qué contexto se hizo. Con esos datos, el CBR trata de inferir cuál de estos resultados es el más apropiado de usar en el problema actual, asumiendo que si aplicamos soluciones buenas en el pasado a problemas similares al actual, esas soluciones deberían de ser buenas también en el problema actual. La información almacenada en la base de casos puede provenir de conocimiento experto o ser generada almacenando las acciones que el experto está realizando mientras resuelve el problema, generando lo que se suele denominar como *trazas*. Estas trazas no son más que ejemplos generados por un experto que sirven de información al CBR para tomar sus decisiones.

La idea de extender los BTs con nodos terminales que no ejecutan una tarea concreta sino un conjunto de posibles tareas, aplazando la decisión de cuál ejecutar al momento de ser ejecutada, ya se ha explorado en trabajos previos con la inclusión de nodos consulta [6] para poder recuperar implementaciones de comportamientos similares en base a una descripción ontológica realizada por el propio diseñador, así como para recuperar planes en juegos de estrategia como Starcraft [7] donde se seleccionaban BTs con diferentes implementaciones de una acción del juego en base a información semántica proporcionada por expertos.

Estos trabajos previos se basaban en la idea de que la información semántica que permitía seleccionar el BT a ejecutar, proviene de una descripción semántica de dicho problema mediante una ontología. Esta información semántica describe el contexto idóneo en el que se debe utilizar dicho comportamiento. Esta información ha de ser introducida por el diseñador, lo cual es poco flexible y requiere de un trabajo extra por su parte que es, además, propenso a errores. Esta nueva aproximación pretende proporcionar mecanismos para que la generación de esa información semántica y los valores correctos de la misma, se aprendan automáticamente. De esta forma se simplifican las tareas que el diseñador o los programadores deben realizar para mantener una ontología actualizada.

También otros autores han utilizado otras técnicas de aprendizaje automático con otros enfoques para intentar seleccionar el BT que mejor resolvía un problema usando algoritmos genéticos [8] y otras técnicas de aprendizaje automático como el Q-Learning [9].

## 4 Behavior Bricks

Behavior Bricks<sup>1</sup> es una herramienta cuya finalidad es el desarrollo de comportamientos inteligentes para videojuegos. Permite la incorporación de tareas primitivas desarrolladas por los programadores para ser usadas en árboles de comportamiento, contruidos por los diseñadores a través de un editor visual integrado, lo que permite la colaboración entre ambos, minimizando problemas de comunicación y permitiendo que ambos trabajen en paralelo.

En Behavior Bricks se define como *tarea primitiva* al mínimo fragmento de código ejecutable por un comportamiento, que son las acciones y condiciones de los BTs. Estas primitivas se definen mediante un nombre (único en toda la colección) que determina la *semántica* de la acción o condición y opcionalmente, una lista de parámetros de entrada (Y también de salida en el caso de las acciones). Estas primitivas deben ser escritas por los programadores en C#.

Las acciones primitivas constituyen los “ladrillos” básicos para la creación de comportamientos por parte de los diseñadores. La posibilidad de parametrizar estas primitivas las dota de generalidad, lo que permite que un estudio de videojuegos pueda, con el tiempo, construirse una biblioteca de acciones y condiciones reutilizables entre proyectos. Para permitir la comunicación entre las diferentes tareas, los parámetros de entrada y salida de las mismas leen y escriben de un espacio de memoria compartido al que denominamos *la pizarra* del comportamiento. Estos parámetros se deben asociar a las variables de la pizarra en el diseño del comportamiento, permitiendo interrelacionarlas. La información almacenada en estas pizarras es lo que denominamos *contexto* del comportamiento.

Al igual que las primitivas, los comportamientos contruidos por los diseñadores también pueden tener parámetros de entrada y de salida. Esto permite que dichos comportamientos puedan ser reutilizados en otros BTs como tareas primitivas.

La extensión que proponemos en este trabajo surge de forma natural gracias a esta última característica: si los BTs ya contruidos pueden colocarse en otros BTs más generales como nodos hoja, podemos ver esos BTs como definiciones de tareas primitivas implementadas con árboles en lugar de directamente en código. La tarea implementada con ese BT puede ser un nuevo tipo de tarea primitiva que pasa a estar disponible en la colección o una implementación distinta de una tarea primitiva ya existente.

Por lo tanto, dada una tarea, podemos tener un conjunto de posibles implementaciones de dicha tarea que podemos utilizar. Para poder hacer esta asociación, los parámetros de entrada y de salida de la tarea a implementar y el comportamiento que la implementa deben coincidir en número y tipo.

---

<sup>1</sup> <http://www.padaonegames.com/bb/>

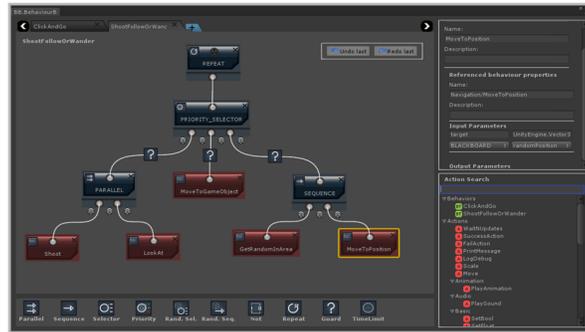


Fig. 1: El editor de Behavior Bricks

#### 4.1 El editor de comportamientos de Behavior Bricks

Behavior Bricks se compone de dos partes: el motor de ejecución de comportamientos y el editor de esos comportamientos. Aunque el motor de ejecución es asíptico y puede ser utilizado en cualquier juego que sea capaz de ejecutar código en C#, el editor está hecho como plug-in para Unity3D <sup>2</sup>.

La figura 1 muestra el aspecto del editor, que está implementado usando UHotDraw [10], un framework desarrollado por los autores que simplifica la creación de editores en Unity.

#### 4.2 Creación de primitivas en Behavior Bricks

Las acciones primitivas en Behavior Bricks son implementadas por los programadores usando C#. Para proporcionar tanto el nombre, como los parámetros de entrada y salida se usa *atributos* de C#. Las acciones se implementan heredando de una superclase con un conjunto de métodos que deben ser sobrescritos y que son invocados por el intérprete durante el *ciclo de vida* de la acción. De ellos, el más importante es `OnUpdate()`, llamado en cada iteración del ciclo del juego, y que debe devolver si la acción ha terminado ya (con éxito o fracaso) o debe seguir siendo ejecutada en la próxima iteración.

Las condiciones evalúan el estado del mundo permitiendo determinar si se cumplen ciertas reglas. Al igual que las acciones, las condiciones primitivas son también implementadas por los programadores utilizando los mismos mecanismos (atributos de C# y herencia). En este caso, el método sobrescrito será `Check()`, que devolverá un valor booleano.

#### 4.3 Metodología de uso de Behavior Bricks

En la industria hay cierto recelo a la hora de permitir a los diseñadores crear comportamientos, incluso si disponen de una herramienta que les facilite la

<sup>2</sup> <http://unity3d.com/>

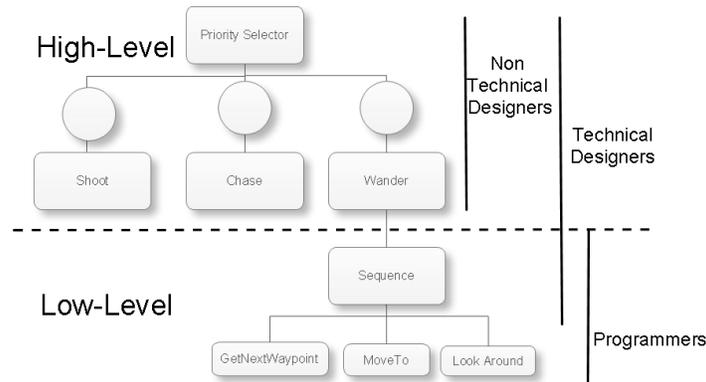


Fig. 2: Comportamientos de alto y bajo nivel y su asignación de responsabilidades

tarea, soliendo encargarse principalmente de supervisar el comportamiento [4]. Basándonos en algunos experimentos, hemos detectado que con una metodología adecuada y formación mínima, los diseñadores son capaces de realizar comportamientos generales sin necesidad de tener demasiados conocimientos de programación. Para ello hemos definido una metodología de uso de Behavior Bricks en la que definimos varios roles a la hora de crear estos comportamientos según el tipo de usuario de la herramienta.

Entre los profesionales dedicados al diseño de videojuegos existen algunos con un perfil más técnico y otros que prácticamente no tienen conocimientos informáticos. Los diseñadores no técnicos podrían tener problemas para crear un comportamiento completo usando el editor, por lo que nuestra metodología propone dividir los comportamientos generales en comportamientos a *bajo nivel* y comportamientos a *alto nivel*.

Cuando pensamos en un comportamiento a *alto nivel* lo hacemos viendo dicho comportamiento como una descripción simple del comportamiento general de un NPC, similar a lo que los diseñadores suelen hacer en los primeros prototipos del juego [11]. Estos comportamientos a *alto nivel* pueden contener otros comportamientos más específicos y más cercanos a los problemas típicos de implementación a los que denominamos comportamientos de *bajo nivel* y que deberían ser creados por diseñadores técnicos o por programadores. De esta forma, podemos dividir tareas y asignar responsabilidades dependiendo de las características del equipo de desarrollo, permitiendo la cooperación entre ambos.

Para clarificar estos conceptos, la figura 2 muestra un esquema del reparto de responsabilidades que define nuestra metodología de uso de Behavior Bricks.

## 5 Extensión de Behavior Bricks con Query Nodes

En el proceso de creación del comportamiento, asumimos que el diseñador sabe cómo debe comportarse el NPC, pero vamos a suponer que estamos en un estado temprano de desarrollo del juego en el que los comportamientos de los

NPCs aún no están claros. Incluso con la separación de tareas descrita en nuestra metodología, la comunicación entre programador y diseñador debe ser fluida y constante en estos primeros compases. Imaginemos que nuestra biblioteca de tareas primitivas por sucesivos proyectos desarrollados, empieza a ser de un gran tamaño y está llena de comportamientos similares pero que tiene distintos matices de implementación. ¿Cuál de ellos debe usar el diseñador? Primeramente puede que ni siquiera lo tenga claro; puede incluso que el programador esté ocupado haciendo otras tareas y el diseñador no disponga de conocimientos suficientes como para crear comportamientos a bajo nivel que se adapten a sus necesidades. ¿Qué alternativas tiene el diseñador en este contexto? Puede esperar a que esas tareas a bajo nivel sean implementadas, puede intentar ir probando todas las tareas similares que encuentre en la biblioteca o aventurarse a implementar una nueva tarea por sí solo. Nosotros proponemos una extensión de Behavior Bricks y de los BTs en general, basada en la idea de incorporar un nuevo nodo hoja denominado *Query Node*, descrito en [6], en la que aprovechando las posibilidades que ofrece Behavior Bricks para crear jerarquías de tareas y comportamientos, el diseñador introduzca como nodo hoja, un nodo especial al que denominamos nodo *Query* en el que se define el *tipo* de tarea que queremos ejecutar, pero no cuál de sus posibles implementaciones se ejecutará. Dicha decisión se llevará a cabo en tiempo de ejecución, seleccionando aquella que mejor se adapte al entorno donde finalmente se utilice.

A modo de ejemplo, imaginemos que el diseñador dispone de multitud de implementaciones diferentes de la acción *Attack*. Por ejemplo: puede atacar cuerpo a cuerpo, desde lejos, puede intentar rodear al enemigo antes de atacarle, o puede autodestruirse si ve que sus posibilidades de victoria son escasas. ¿Cuál es la más adecuada? En [6] o [7] para tomar esta decisión se usaba conocimiento experto descrito mediante una ontología o descripción semántica, tanto de los comportamientos almacenados como del árbol donde queríamos integrar dicho nodo *Query*. Nuestra nueva aproximación es adquirir este conocimiento mediante un proceso de aprendizaje, intentando que el diseñador tenga que introducir el mínimo conocimiento semántico posible para que sea fácil de mantener. Para ello nos enfrentamos a varios problemas a resolver.

Por un lado, no todos los comportamientos son utilizables en todos los NPCs. Tampoco el diseñador quiere que sus NPCs puedan atacar de todas las formas implementadas posibles. Así que debe haber algún tipo de prerequisite para poder usar una tarea o un comportamiento.

En la definición de las tareas primitivas añadimos un nuevo método: *bool CheckPrerequisites()* que informa al comportamiento que lo ejecuta de las restricciones de dicha tarea. Imaginemos por ejemplo en el caso de la acción *Attack*, que para poder usar el ataque a distancia, el NPC debe disponer de un arma a distancia, o solo ciertos enemigos de cierto nivel son capaces de rodear antes de atacar o que para poder autodestruirse, el NPC debe tener una bomba. Este mecanismo permite de forma sencilla para el programador de la acción describir qué restricciones de uso son las que precisa dicha acción. Estas restricciones pueden ser supervisadas por el diseñador e incluso añadir nuevas restricciones

adicionales de diseño en el nodo Query, (por ejemplo que la acción a ejecutar sea del nivel adecuado para el NPC o dependiendo de la dificultad del juego) usando una condición previamente programada que el nodo chequea para cada posible comportamiento a elegir. No es necesario definir los prerrequisitos de los comportamientos complejos creados por los diseñadores ya que estos se pueden inferir de las tareas primitivas y otros subcomportamientos que utilicen los comportamientos seleccionados por el nodo Query, simplemente chequeando todos los prerrequisitos de todas las tareas que contenga el comportamiento.

Así pues, cuando el nodo Query se ejecute, deberá buscar en la base de casos aquellos que implementen la tarea a ejecutar que ha sido establecida en tiempo de diseño y de entre ellos, preseleccionar aquellos que son ejecutables cumpliendo por un lado sus prerrequisitos y por otro lado la condición definida en el nodo Query. De todos estos candidatos se debe seleccionar cuál es el más adecuado. Si no hay ningún comportamiento recuperado, entonces se ejecutará la tarea primitiva establecida.

## 5.1 Métodos de selección del comportamiento más adecuado

Cuando el nodo Query ha seleccionado el conjunto de posibles comportamientos a ejecutar, debe decidir cuál de ellos ejecuta. Para ello el algoritmo recurre a una base de casos donde recupera cuál de los candidatos a ejecutar se utilizó en unas circunstancias similares a las actuales. Para ello debemos guardar en esta base de casos la acción ejecutada y el contexto del árbol de comportamiento que la ejecutó (su pizarra o parte de ella). La información almacenada en la base de casos tendrá entonces una lista de atributos con su valor en el momento de ejecutar el comportamiento. Utilizando una medida de similitud entre los parámetros almacenados y los de la pizarra del comportamiento, seleccionaremos el comportamiento que se ejecutó en un contexto más parecido al nuestro.

El problema aquí es hacer la correspondencia entre los parámetros almacenados en las trazas de la base de casos y el estado del contexto del Query Node, ya que se necesita hacer una correspondencia semántica entre el nombre del atributo almacenado en la base de casos y el nombre del atributo de la pizarra del comportamiento que ejecuta el nodo Query. Así que para poder hacer esta correspondencia, necesitamos etiquetar los parámetros con una etiqueta semántica común para todos los comportamientos. Estas etiquetas semánticas son creadas por los diseñadores en forma de una simple lista o relacionándolas por sinónimos mediante un *tesauro* si queremos una semántica más rica. De esta forma, en vez de almacenar los nombres de la pizarra del comportamiento que genera la traza para la base de casos, se almacena su valor semántico. En la tabla 1 podemos ver la pizarra del comportamiento que ejecuta el nodo Query Attack.

En la tabla 2 podemos ver un ejemplo de una base de casos para la acción Attack con diferentes valores semánticos de los atributos y el resultado de similitud con el contexto actual mediante una distancia euclídea simple como ejemplo ilustrativo de como realizar dicha recuperación. Los marcados con guión no están almacenados para el caso concreto ya que no son relevantes. En este ejemplo,

<i>Attribute</i>	<i>SemanticTag</i>	<i>Value</i>
Distance	EnemyDistance	13
Bullet	Bullets	25
Life	PlayerLive	12
EnemyInCover	BehindCover	False

Table 1: Pizarra del comportamiento que ejecuta el nodo Query Attack

nos quedaríamos con la distancia mínima que determina la mayor similitud con el contexto actual, que en este caso es el comportamiento *AttackToDistance*.

<i>EnemyDistance</i>	<i>PlayerLive</i>	<i>Bullets</i>	<i>BehindCover</i>	<i>Behavior</i>	<i>Task</i>	<i>Similarity</i>
12	5	50	-	AttackToDistance	Attack	2.88
1	-	0	-	MeleeAttack	Attack	23.28
2	1	12	-	AutoDestroy	Attack	12.16
10	12	0	True	RoundAndMelee	Attack	11.27

Table 2: Base de casos de ejemplo con las diferentes implementaciones de Attack

Para disminuir posibles errores al seleccionar el comportamiento a ejecutar se pueden seleccionar los K comportamientos más similares (KNN, del inglés *k-Nearest Neighbors algorithm*) [12] y quedarnos con el que más se repite.

Al seleccionar la acción en base a la similitud, el diseñador no puede garantizar como se va a comportar el NPC por lo que pueden surgir comportamientos emergentes que van a ofrecer una cierta impredecibilidad en el comportamiento del NPC, que puede ser interesante desde el punto de vista del jugador y su percepción de la experiencia jugable pero que no es siempre del agrado de los diseñadores que normalmente prefieren mantener el control del juego. Por lo que hay que intentar mantener dichos comportamientos emergentes bajo control para que el NPC no realice comportamientos sin sentido o alejados de la idea que el diseñador tiene del juego.

## 5.2 Generación de la base de conocimiento

Para generar la base de casos, definimos un nuevo decorador llamado *Record* que se asigna al nodo del árbol que queremos sustituir por el nodo Query posteriormente y que utilizaremos para almacenar las trazas. Este árbol generado por el diseñador generará trazas guardando los atributos del comportamiento que el diseñador seleccione. Las trazas se guardarán en una base de casos que puede ser alimentada con diferentes implementaciones del sub-árbol a aprender.

Otra alternativa para generar la base de casos es establecer un nuevo tipo de nodo terminal que denominamos *InteractiveRecord*. Este nodo permite ejecutar los comportamientos a petición de un usuario, asignando cada comportamiento a ejecutar a un control. De esta forma, cuando el NPC decida ejecutar

la acción a aprender, avisará el diseñador para que este pulse el control asociado al comportamiento correspondiente que él considere mejor en ese momento, de forma que enseña al NPC como debe comportarse. Dado que el diseñador puede equivocarse al ejecutar el comportamiento, se puede asignar a cada una de los comportamientos a ejecutar una función de valoración que determine cómo de buenas han sido estas trazas, pudiendo hacer una selección de la base de casos descartando aquellas decisiones erróneas del diseñador ejecutando el juego.

### 5.3 NPCs con aprendizaje adaptativo

Modificando ligeramente el comportamiento del nodo Query, podemos permitir aprendizaje durante el juego, dotando el NPC de la capacidad de adaptarse al comportamiento del jugador y aprender nuevas estrategias. El NPC puede partir de una base de casos configurada por el diseñador, pero que se irá modificando en base a lo que el NPC vaya aprendiendo durante el tiempo de juego con el jugador. Para ello, el nodo Query dispone de un parámetro que determina la probabilidad de seleccionar un comportamiento de la base de casos o uno al azar de entre los disponibles. La probabilidad de elegir uno u otro puede ir variando en función de múltiples aspectos como la diversidad de comportamientos existente en la base de casos o los resultados obtenidos por el NPC. Si el NPC detecta que sus estrategias no consiguen vencer al jugador, este puede incrementar la probabilidad de elegir nuevas formas de implementar un comportamiento, dándole más probabilidad de elegir uno de forma aleatoria. Las trazas en este aprendizaje inline deben de estar puntuadas para saber si el comportamiento que se ha seleccionado ha cumplido su objetivo (por ejemplo para la tarea de atacar, si ha dañado al jugador y cuánto daño le ha hecho). Cuando el nodo Query seleccione un comportamiento de la base de casos, seleccionara de entre los  $K$  más parecidos, aquel que tenga una valoración más alta.

Configurando este parámetro de “aleatoriedad” podemos generar comportamientos más emergentes y más *sorprendentes* para el jugador, lo que ayuda a reducir la sensación de IA *scriptada* (repetitiva) ya que el NPC puede cambiar de estrategia si detecta que la elegida no es eficaz.

## 6 Conclusiones y trabajo futuro

Con esta extensión de los BTs permitimos a los diseñadores un mayor grado de autonomía, pudiendo prescindir en muchas ocasiones del programador para implementar comportamientos a bajo nivel así como ayudar al diseñador a que pueda entrenar al NPC de forma interactiva, lo que le facilita la tarea de crear comportamientos, incluso si no dispone de conocimientos de programación. Esto le permite desarrollar comportamientos a bajo nivel que según nuestra metodología, deberían ser realizados por diseñadores técnicos o programadores. Hay que tener en cuenta que el aprendizaje automático requiere de una implicación activa del diseñador en el mismo por lo que puede resultar tedioso llevarla a cabo.

Como trabajo futuro queremos poner en práctica estas técnicas y validarlas experimentalmente para ver cómo funcionan y qué aceptación tienen entre los diseñadores y si realmente simplifican la creación de comportamientos complejos, así como testear la capacidad de aprendizaje del NPC con la técnica de aprendizaje diseñada y poder aprender comportamientos de arriba a abajo, es decir ir aprendiendo comportamientos de bajo nivel y posteriormente, usando los nodos Query ya aprendidos, aprender comportamientos de más alto nivel hasta permitir aprender el comportamiento completo del NPC.

Además queremos estudiar qué funciones de similitud obtienen mejores resultados en la recuperación así como qué algoritmos de selección de atributos podemos utilizar para simplificar la base de casos, así como estudiar la posibilidad de simular la ejecución de los comportamientos en la fase de selección de comportamientos candidatos a ser ejecutados, de forma especulativa para que sólo se tenga en cuenta los preresquisitos de las acciones que realmente se ejecutarán en el contexto del Query Node que las ejecuta.

## References

1. Rouse III, R.: *Game design: Theory and practice*. Jones & Bartlett Learning (2010)
2. Champandard, A.J.: 3.4. In: *Getting Started with Decision Making and Control Systems*. Volume 4 of *AI Game Programming Wisdom*. Course Technology (2008) 257–264
3. Rabin, S.: 3.4. In: *Implementing a State Machine Language*. Volume 1 of *AI Game Programming Wisdom*. Cengage Learning (2002) 314–320
4. Champandard, A.J.: Behavior trees for next-gen ai. In: *Game Developers Conference*. (2005)
5. Aamodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications* **7**(1) (1994) 39–59
6. Flórez-Puga, Gómez-Martín, Díaz-Agudo, González-Calero: Query enabled behaviour trees. *IEEE Transactions on Computational Intelligence And AI In Games* **1**(4) (2009) 298–308
7. Palma, R., Sánchez-Ruiz, A.A., Gómez-Martín, M.A., Gómez-Martín, P.P., González-Calero, P.A.: Combining expert knowledge and learning from demonstration in real-time strategy games. In: *Case-Based Reasoning Research and Development*. Springer (2011) 181–195
8. Colledanchise, M., Parasuraman, R., Ögren, P.: Learning of behavior trees for autonomous agents. *arXiv preprint arXiv:1504.05811* (2015)
9. Dey, R., Child, C.: Ql-bt: Enhancing behaviour tree design and implementation with q-learning. In: *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, IEEE (2013) 1–8
10. Sagredo-Olivenza, I., Flórez-Puga, G., Gómez-Martín, M.A., González-Calero, P.A.: UHotDraw: a GUI framework to simplify draw application development in Unity 3D. In: *Primer Simposio Español de Entretenimiento Digital*. (2013)
11. Hudson's, K.: The ai of bioshock 2: Methods for iteration and innovation. In: *Game Developers Conference*. (2010)
12. Dasarathy, B.V.: Nearest neighbor (NN) norms: NN pattern classification techniques. (1991)