Extensión de los grafos de dependencia para incrementar la rejugabilidad*

Marco Antonio Gómez-Martín, Gonzalo Flórez-Puga, Pedro Pablo Gómez-Martín and Pedro A. González-Calero

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
email: {marcoa, gflorez, pedrop, pedro}@fdi.ucm.es

Abstract. *Carlos, Rey Emperador* es un juego de estrategia ambientado en la época de Carlos V. Además de la simulación del reino y de cómo las políticas utilizadas afectan a la fidelidad del pueblo, el juego presenta al usuario eventos históricos (reales o inventados para el juego) que debe intentar solucionar y cuyo resultado afecta a los siguientes sucesos en el juego. Para aumentar la rejugabilidad evitando caer en partidas repetidas con la misma sucesión de eventos, se han extendido los grafos de dependencias utilizados en otros juegos.

1 Introducción

La generación de contenido para videojuegos supone un gran porcentaje del tiempo invertido por el equipo de desarrollo. Una vez que éste pasa a la etapa de producción, desarrolladores, diseñadores y artistas se concentran en construir los niveles, personajes y comportamientos que harán que experiencia de juego sea lo más larga posible.

Con los años, los diseñadores de juegos han ideado distintas formas de alargar las horas de juego de un título eliminando la necesidad de construir niveles nuevos. Es lo que se conoce como *rejugabilidad*, o lo que es lo mismo, intentar que el usuario juege más de una vez a cada nivel.

Existen dos mecanismos muy simples utilizados tradicionalmente. El primero es hacer que el comportamiento del juego (y por lo tanto los niveles a los que se enfrenta el usuario) tengan una componente aleatoria (piénsese por ejemplo en el Tetris). Otro es proporcionar distintos niveles de dificultad, con la esperanza de que el jugador que termina el juego en un nivel fácil vuelva a intentar completarlo en un nivel de dificultad más complicado.

Otro mecanismo muy utilizado es poblar los niveles de *coleccionables* que el jugador debe intentar recoger pero que no comprometen el avance del jugador pues podrá completar el juego sin recogerlos todos. El mero hecho de comprobar que a pesar de terminar el juego no se han recogido todos los objetos hace que muchos usuarios intenten los niveles de nuevo hasta conseguirlos todos. Esta estrategia tiene tanto éxito que las distintas plataformas de juegos manejan

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

sistemas de *logros* que, a modo de medallas virtuales, permiten a un jugador mostrar los éxitos conseguidos en los distinto títulos.

En el mundo de la narración interactiva y de los juegos de aventuras es habitual encontrarse con juegos que ofrecen *más de un final* de forma que las acciones realizadas por el jugador en momentos concretos del juego determinan cuál será el *camino de sucesos* que seguirá el juego. Este camino de sucesos prefijado puede ser además adornado con eventos aleatorios que el juego introduce para hacer de cada partida una experiencia única.

Sin embargo, esta aleatoriedad puede estar reñida con la historia que el diseñador quiere contar en ese camino de sucesos. Esto es especialmente delicado en juegos basados en hechos históricos reales.

En este artículo presentamos nuestra experiencia en el desarrollo del juego *Carlos, Rey Emperador* disponible para plataformas móviles (Android e iOS) y el modo en el que, utilizando una extensión de los grafos de dependencias, logramos incrementar la rejugabilidad garantizando que el resultado final del camino de sucesos que percibirá el usuario se adhiere a lo ideado por el diseñador.

La siguiente sección describe cómo los grafos de dependencia han sido utilizados en juegos. La sección 3 explica los retos que planteó la implementación de *Carlos, Rey Emperador* y la extensión de los grafos de dependencia que la hizo posible. La siguiente sección explica los mecanismos de ejecución necesarios para esos grafos extendidos. Tras ella, la sección 5 entra un poco más en detalle en la implementación de los grafos y su uso en *Carlos, Rey Emperador*. La última sección presenta unas pequeñas conclusiones.

2 Grafos de dependencias en videojuegos

Los grafos han demostrado ser una estructura de datos muy útil en el mundo de los videojuegos. Los desarrolladores de juegos tiene entre su vocabulario conceptos como máquinas de estados, búsqueda de caminos, A* o árboles de comportamiento, por mencionar algunos ejemplos.

Los grafos han sido estudiados intensivamente y categorizados según sus propiedades. Los conocidos como DAG (del inglés, grafo dirigido acíclico, *directed acyclic graph*) son grafos cuyas aristas son dirigidas (tiene un origen y un destino) y no tienen ciclos.

Desde los años 60 se están utilizando DAG como grafos de dependencias, siendo los diagramas PERT el ejemplo prototípico. En ellos cada nodo representa una tarea y una arista (dirigida) indica que la tarea asociada al nodo origen debe terminarse antes de que pueda comenzar la del destino. Para que el grafo de dependencias tenga sentido, éste no debe contener ciclos (de ahí que sea un DAG) pues en otro caso una tarea no podría empezar hasta que ella misma no estuviera terminada.

Los grafos de dependencias se han utilizado en juegos como herramienta de diseño. Su uso es muy natural en juegos de aventuras como la mítica serie *Monkey Island*, *Grim Fandango* o *The Day of the Tentacle*. En ellos, cada uno de los nodos representa un puzzle que debe ser resuelto por el jugador lo que

abre la puerta (por diseño) a que pueda abordar los siguientes puzzles. Las dependencias suelen darse debido a que resolver un puzzle proporciona un objeto o un conocimiento necesario para poder terminar otro.

Aunque un DAG general puede tener varios nodos sin dependencias, lo normal es que en estos juegos exista un único nodo "raíz"[1] que simboliza el principio del juego. De forma equivalente, los nodos terminales son puzzles resueltos que no abren otros puzzles; en ausencia de varios finales, únicamente hay uno de esos nodos que simboliza el *juego completo*.

Cuando existe un único recorrido en orden topológico del DAG[2] el usuario percibirá un juego completamente lineal y la rejugabilidad se verá comprometida. Lo normal, sin embargo, es que existan puzzles que puedan completarse en orden arbitario, lo que da como resultado distintos órdenes topológicos. Por poner un ejemplo, en el grafo de dependencias de un juego como *The Day of the Tentacle* (con un único principio y un único final), el número de recorridos topológicos distintos supera los miles de millones [1]. Eso hace virtualmente imposible que dos jugadores superen el juego en el mismo orden. Sin embargo el problema de la rejugabilidad sigue estando ahí, pues el jugador que supera el juego no encontrará nada nuevo en siguientes partidas; podrá terminarlo en otro orden, pero los puzzles a los que se enfrentará serán fundamentalmente los mismos.

Para añadir variabilidad se pueden añadir distintos finales al grafo. El alcance, no obstante, es limitado. Por un lado, tener varios finales requiere un gran esfuerzo de generación de contenidos. Por otro lado, el hecho de que el jugador pueda seguir avanzando por las distintas ramas o historias alternativas hace que en la práctica un usuario pueda explorar en una única partida prácticamente la totalidad del juego. Eso es debido a que tal y como hemos descrito el DAG, éste no tiene nodos disyuntivos que bloqueen el acceso a algunos de los vértices sucesores y den acceso a otros.

3 Extensión de los grafos de dependencias para su uso en ejecución

Los grafos de dependencias son también útiles en otros juegos no tan centrados en puzzles marcando el progreso del juego y determinando su final. Pensamos por ejemplo en juegos de simulación o estrategia en donde el objetivo principal del juego no es resolver todos los retos presentados sino terminar con una serie de recursos o puntos.

En estos juegos de estrategia, a la parte de simulación se le une la aparición de situaciones que el usuario debe resolver. Estas situaciones o retos pueden haber sido fijadas de antemano por el diseñador para aparecer en ese momento concreto del juego o pueden formar parte de un conjunto de eventos que el motor dispara de forma aleatoria.

[1] En el sentido de no tener dependencias, no en el sentido de nodo especial en árboles.

[2] El recorrido topológico es aquel que visita todos los nodos del grafo de tal forma que todas las dependencias de un nodo han sido visitadas antes que el propio nodo.

En el caso del juego *Carlos, Rey Emperador* desarrollado por los autores, queríamos que esa parte de eventos tuviera gran peso desde el punto de vista del diseño. En concreto, entre los objetivos que teníamos en mente estaban:

- Que cada partida jugada correspondiera con una época histórica concreta de la vida del rey Carlos V.
- Que la simulación, por lo tanto, comenzara en unas fechas históricas dadas.
- Que cada partida fuera distinta, presentando al jugador situaciones o sucesos distintos en cada una de ellas.
- Que, a pesar de esa necesaria aleatoreidad, cada partida no incurriera en contradicciones al presentar situaciones incoherentes con el momento histórico, estado de la simulación o sucesos ocurridos anteriormente en la partida.
- Que algunos de los eventos tuvieran rigor histórico (es decir, fueran eventos reales acaecidos en las fechas en las que se encontrara la simulación).
- Que el diseñador pudiera crear otros eventos inventados asignándoles unas fechas históricas plausibles en los que podrían haber ocurrido.

La plausibilidad histórica en la aleatoriedad recuerda a los grafos de dependencia de la sección anterior. Igual que en un juego como The Day of the Tentacle primero hay que encontrar el laboratorio del Dr. Fred antes de conseguir el sello que te permitirá mucho después enviar una carta, en un juego como *Carlos, Rey Emperador* el jugador tiene primero que conseguir el beneplácito de su madre, la reina Juana, antes de conseguir que su hermano Fernando le reconozca como el verdadero sucesor. Y si el juego no decide activar el reto del beneplácito de Juana no debe después disparar el de la fidelidad del hermano.

Sin embargo, los grafos de dependencias descritos no permiten implementar todos los puntos anteriores. Para hacerlo posible, planteamos una extensión de los mismos consistente en enriquecer la información de los nodos y definir dos tipos de dependencias distintas entre ellos.

Antes de entrar en detalle sobre la extensión, debemos destacar el cambio en tanto en el uso del grafo de dependencias como en la interpretación que le daremos a los nodos.

En primer lugar, el grafo de dependencias descrito en la sección anterior era fundamentalmente una herramienta *de diseño*. Las dependencias entre los distintos puzzles se daban porque al resolver un puzzle se habría la posibilidad de resolver el siguiente. Por ejemplo, cuando el jugador consigue entrar en el laboratorio del Dr. Fred antes mencionado, podrá encontrar oculto en él el sello que añadirá a su inventario. Pero durante la ejecución del juego no existe el grafo de dependencias explícitamente. El juego no necesariamente conoce qué nodos (puzzles) han sido ya resueltos, cuáles están "abiertos" y pueden ser solucionados y cuales siguen bloqueados. Como veremos más adelante, el uso de los grafos que proponemos mantiene en memoria qué nodos están siendo ejecutados en cada momento y monitorizan su terminación.

En segundo lugar, en esos grafos un nodo representa un puzzle que el jugador tiene que superar para avanzar. El puzzle está ahí esperando a ser resuelto por el jugador y seguirá estando hasta que éste lo complete. En el caso de juegos de

estrategia o simulación en los que se aplica nuestra extensión, los nodos representan eventos o retos que presentaremos al jugador, como tener que conseguir el beneplácito de su madre. No son puzzles que estén disponibles en cualquier momento, sino retos que se disparan en momentos concretos y que el jugador debe resolver en un espacio limitado de tiempo antes de desaparecer.

3.1 Precondiciones

En los grafos de dependencias descritos anteriormente cada uno de los nodo-puzzle tenían un único tipo de dependencia (o precondición): aquella relacionada con la finalización de otros nodo-puzzles del propio grafo.

Sin embargo, la naturaleza del propio nodo puede imponer unas condiciones adicionales. La primera extensión a los grafos de dependencia, pues, consiste en añadir información adicional a los nodos con las condiciones *externas al grafo* que deben cumplirse para que un nodo pueda activarse.

Esas precondiciones del nodo dependen del estado de la simulación. En el caso de juegos con una fuerte componente histórica, son especialmente importantes las precondiciones temporales que pueden limitar las fechas en las que puede activarse el nodo. De esta forma, existirán nodos con todas todas las dependencias satisfechas (todos los nodos de los que depende han sido completados) pero que no puede ser seleccionada porque su fecha de activación ya ha expirado (o aún no se ha llegado a ella).

3.2 Aristas

En juegos de tipo puzzle como los mencionados en la sección anterior, no se distingue entre puzzle no resuelto y puzzle no intentado. Es decir, los puzzles no se *consumen* al ser intentados y no resueltos, sino que el jugador tiene la opción de intentar superarlo una y otra vez. Por ejemplo, si un puzzle consiste en conseguir abrir una puerta (y para eso hay que encontrar distintos objetos por el escenario), la puerta o bien está abierta (puzzle resuelto) o bien está cerrada (puzzle aún no resuelto); pero no hay un estado en el que el jugador ha intentado abrir la puerta y al no conseguirlo la ha bloqueado para siempre.

La extensión que proponemos permite especificar dos resultados a un nodo: completado con éxito y terminado con fracaso. Las aristas que unen los distintos nodos del DAG, pues, están etiquetadas con el tipo de resultado necesario del nodo origen para activar esa arista/dependencia.

3.3 Resultados

Dado que los grafos de dependencia tradicionales no se hacen explícitos durante la ejecución, el resultado de la consecución de un puzzle no se encuentra almacenado en la información asociada al nodo del puzzle.

En nuestro caso los nodos sí se almacenan en memoria y por tanto el responsable de su ejecución puede ser el que aplique sobre la simulación el resultado de completar con éxito o fracaso ese nodo.

4 Ejecución del grafo

La ejecución del grafo puede separarse en dos: la parte encargada de gestionar qué nodos son seleccionables en cada momento (*nodos abiertos* a partir de ahora) y la que se encarga de poner a ejecutar esos nodos disponibles.

Si, como en el caso de *Carlos, Rey Emperador*, los nodos tienen restricciones temporales, la gestión de los nodos abiertos debe tener en cuenta tanto las dependencias del propio grafo como esas restricciones.

4.1 Seguimiento del grafo

El encargado del seguimiento del grafo (o *tracker*) mantiene, además de la información del grafo:

- Fecha actual de la simulación.
- Lista con los nodos abiertos, entendiendo estos los que tienen todas sus dependencias satisfechas (los nodos de los que dependen han tenido el resultado necesario) y además se cumplen sus restricciones temporales (la fecha de la simulación permite su activación).
- Cola de prioridad con los nodos cuyas dependencias se han satisfecho pero cuya fecha mínima de activación aún no ha llegado. Están ordenados por la fecha más baja en la que se pueden activar.
- Información sobre el número de dependencias pendientes de satisfacerse de los nodos que aún no se han abierto. En el arranque coincide con el grado de entrada de cada vértice del grafo y después se va actualizando cuando se van completando los nodos.

La gestión de la simulación es responsable de notificar al *tracker* del cambio de fechas para que éste actualice sus nodos abiertos. El algoritmo que actualiza los nodos aparece en la figura 1.

Para ganar algo de eficiencia en la implementación se podría utilizar también una cola de prioridad para los nodos abiertos ordenando esta vez por la fecha del final del intervalo. No se recomienda esta opción porque dificulta la implementación de otras partes del *tracker*. Una de ellas es la retirada de los nodos abiertos cuando éste es informado de que el juego ha *activado* uno de los nodos y se lo ha presentado al usuario.

Cuando el usuario completa (ya sea con éxito o fracaso) el evento asociado a uno de los nodos activados, el *tracker* debe actualizar los nodos abiertos. Para eso decrementa el número de dependencias de los nodos adyacentes y si alguno queda sin dependencias lo añade a las listas correspondientes. El algoritmo que aparece en la figura 2 se encarga de actualizar el *tracker* cuando el jugador completa con éxito un nodo. La versión de completar un nodo con fracaso es equivalente.

La última responsabilidad del *tracker* es dar al ejecutor la lista de nodos que pueden ser ejecutados en un momento dado. En un grafo libre de precondiciones en los nodos, esa lista coincide con la de los nodos abiertos pues contiene

```
// Date currentDate;
// List<Node> openNodes;
// PriorityQueue<Node> withoutDependencies;

void advance(int days)
{
    currentDate += days;

    // Quitamos los expirados
    foreach (Node n : openNodes)
        if (n.NotAfter < currentDate)
            openNodes.remove(n);

    // Añadimos los nodos sin dependencias
    // cuya fecha se acaba de alcanzar
    while (!withoutDependencies.empty() &&
           withoutDependencies.first().NotBefore <= currentDate) {
        openNodes.Add(withoutDependencies.first());
        withoutDependencies.popFirst();
    }
}
```

Fig. 1: Algoritmo de actualización de nodos abiertos al avanzar la fecha de simulación

todos los vértices cuyas dependencias ya se han satisfecho y además cumplen las restricciones temporales. Sin embargo, si los nodos han sido extendidos con precondiciones relacionadas con el estado de la simulación, el *tracker* filtrará la lista antes de devolverla.

4.2 Ejecutor

Entendemos por ejecutor al elemento responsable de disparar los eventos y presentarlos al usuario. Es él el responsable de decidir, de entre todos los nodos disponibles, cuáles pone a ejecutar.

Su implementación está muy ligada al juego y de hecho puede tener sentido utilizar ejecutores distintos en el mismo juego.

Sea como sea, la implementación del ejecutor utilizará el *tracker* explicado anteriormente. Regularmente le pedirá cuáles son los nodos que pueden poner a ejecutarse y decidirá si lanza alguno o no. Si lanza uno, será él el responsable de vigilar su progreso y comprobar si éste se completa con éxito o fracaso para actualizar el *tracker* interno.

Es en el ejecutor donde reside la capacidad de construir partidas distintas y fomentar la rejugabilidad. En un extremo tenemos ejecutores que activan todos los nodos abiertos; en el otro extremo tenemos aquellos que no activan ninguno. Las dos alternativas hacen que todas las partidas sean iguales.

```
void executionEndedWithSuccess(Node n)
{
    foreach (Edge e : graph.Edges(n))
    {
        if (e.IsSuccess)
        {
            Node v = e.target();
            numDependencies[v]--;
            if (numDependencies[v] == 0)
                withoutDependencies.add(v);
        }
    }
    // Abrimos todos los nuevos que hayan podido aparecer
    // reutilizando el método anterior, simulando que no
    // han pasado días.
    advance(0);
}
```

Fig. 2: Algoritmo de actualización cuando un nodo se completa con éxito

Idealmente, las implementaciones deben permitir configurar la probabilidad de selección de un nuevo evento, limitar el número de eventos activos en cada momento y otros parámetros. Si el juego lo requiere, se pueden tener nodos de *ejecución obligatoria* que el ejecutor debe activar siempre, una vez satisfechas sus dependencias y precondiciones y cuya existencia deberá tener en cuenta el ejecutor. Estos sucesos pueden penalizar la rejugabilidad al aparecer en todas las partidas pero son indispensables para la construcción de las fases de tutorial.

5 Implementación en *Carlos Rey Emperador*

La extensión de los grafos de dependencias la hemos puesto en práctica en el juego *Carlos, Rey Emperador*. Es un juego de estrategia en el que el jugador hace las veces de Carlos V, decidiendo políticas, estableciendo alianzas y entrando en guerra con reinos vecinos entre otras cosas.

Durante cada partida, que se juega en un periodo concreto de la vida del monarca, al jugador se le van presentando distintas situaciones o retos que debe ir superando, que se corresponden con los nodos del grafo descrito anteriormente.

La figura 3 muestra una captura del juego en el momento en el que se muestra el reto en el que se pide al usuario que debe conseguir el beneplácito de su madre. Se ve también el resultado del nodo tanto si es completado con éxito como con fracaso.

Para la implementación del juego se utilizó Unity. Para permitir a los diseñadores crear los eventos y el grafo, se hizo una pequeña adaptación del editor que permitía, en una primera fase, definir los eventos o sucesos posibles (figura 4).

Las propiedades configurables son fundamentalmente:

Fig. 3: Captura de *Carlos, Rey Emperador* mostrando uno de los retos

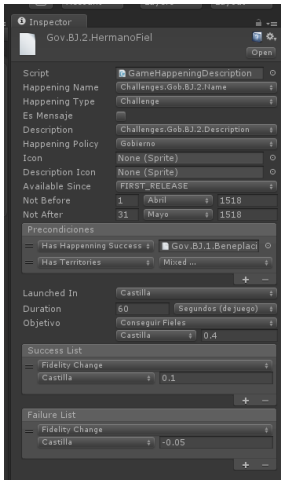

Fig. 4: Definición de un nodo del grafo

- Descripción del evento: incluye el texto a presentar al usuario, tipo de evento (para utilizarlo en el interfaz), etc.
- Intervalo de fechas en el que es aplicable.
- Precondiciones: en ellas se incluyen cosas como "el jugador debe tener en su reino los territorios X e Y". En la implementación, además, es aquí donde ponemos las dependencias del grafo (nodos que han debido completarse con éxito o fracaso), para evitar tener que dibujar el grafo explícitamente.
- Descripción de la ejecución: qué cosas tiene que hacer el jugador para completar el evento y algunos otros parámetros.
- Resultado del nodo: qué ocurre en la simulación cuando el reto se completa, ya sea correctamente o no. Cabe mencionar que aquí no se incluye qué nodos se "abren" al terminar con el reto, pues las aristas se guardan como precondiciones en los nodos destino de las mismas.

En la producción del juego se contó con una historiadora experta en la época de Carlos que supervisó la creación de la colección con los casi 400 eventos tanto históricos reales como inventados para el juego.

Con esos eventos creados, el diseñador después seleccionaba el subconjunto que debía utilizarse para cada una de las partidas/misiones. En la figura 5 aparece una vista parcial de la definición de las propiedades de una partida. Como se aprecia, desde el editor de Unity se puede configurar los parámetros del ejecutor, que incluye el tiempo mínimo y máximo que se puede esperar para lanzar un nuevo suceso, el número máximo de eventos lanzados simultáneamente y la probabilidad de lanzar un nuevo evento.

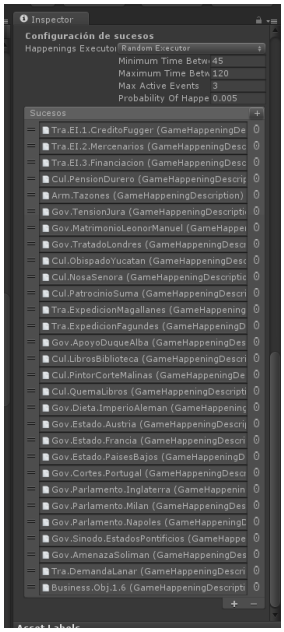

Fig. 5: Definición de los sucesos disponibles en una partida

6 Conclusiones y trabajo relacionado

Los grafos son una herramienta muy útil en el desarrollo de videojuegos, para representar, implicita o explícitamente, información de muy diferente naturaleza. Los *grafos dirigidos acíclicos* (DAG) son útiles para almacenar información de dependencias, de ahí que surjan de manera natural en los juegos de aventura, entre los que se encuentran las sagas clásicas de la desaparecida LucasArts.

En ellos, sin embargo, no se representa de manera natural la posibilidad de que el jugador pueda fallar *de manera irreversible* un determinado puzzle

(nodo), o que éstos tengan precondiciones adicionales a las modelizadas por el propio DAG.

En los juegos de estrategia estas necesidades se hacen más patentes si los *eventos* que ocurren durante un nivel para retar al jugador se almacenan también usando un DAG. Por ejemplo, la puesta en marcha de un evento puede tener sentido únicamente si antes se produjeron otros, y el usuario los completó con éxito. Además, si se quiere aumentar la rejugabilidad, es importante que diferentes partidas tengan diferentes eventos, y el juego deberá garantizar la coherencia de todos ellos en función de sus restricciones.

El método de generar contenido para el juego teniendo en cuenta que ciertos eventos han de haberse producido anteriormente es una forma incompleta de planificación que conecta nuestro trabajo con investigación en narración interactiva [2]. Una técnica habitual de generar contenido narrativo que se ajusta dinámicamente a la interacción con el jugador es identificar "puntos de trama" (*plot points*) que permiten articular la narración, asociando precondiciones a cada punto de trama para así poder decidir en un momento dado qué puntos de narración son aplicables [3]. En cualquier caso, el problema que hemos resuelto de forma práctica en este trabajo es más sencillo que el problema de la narración interactiva que requiere algún tipo de control global o "gestión de drama" [4] y también suele incluir alguna forma de modelado del usuario [5].

En este artículo hemos descrito el modo en el que se representaron los eventos en el juego *Carlos, Rey Emperador* usando un DAG, y cómo se utilizó en ejecución para mantener el control de los eventos a lanzar. En todo momento se tuvo presente la necesidad de asegurar la coherencia, la plausabilidad histórica y un rendimiento óptimo, especialmente importante en el desarrollo de juegos para dispositivos móviles.

References

1. Weinberg, J.: Journey into the DAG: Puzzle dependency charts, tentacles and you. In: Game Developers Conference. (2016)
2. Riedl, M.O., Bulitko, V.: Interactive narrative: An intelligent systems approach. AI Magazine **34**(1) (2013) 67–77
3. Porteous, J., Cavazza, M., Charles, F.: Applying planning to interactive storytelling: Narrative control using state constraints. ACM TIST **1**(2) (2010) 10
4. Magerko, B., Laird, J.E., Assanie, M., Kerfoot, A., Stokes, D.: AI characters and directors for interactive computer games. In McGuinness, D.L., Ferguson, G., eds.: Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA, AAAI Press / The MIT Press (2004) 877–883
5. Sharma, M., Ontañón, S., Mehta, M., Ram, A.: Drama management and player modeling for interactive fiction games. Computational Intelligence **26**(2) (2010) 183–211