Escuela Técnica Superior de
Ingeniería Informática

Master Universitario en Lógica, Computación
e Inteligencia Artificial

Trabajo fin de Master: IA para el ajuste dinámico de la dificultad en videojuegos

Autor:
Daniel Márquez Quintanilla

Tutor:
D. Joaquín Borrego Díaz

Índice general

Índice de figuras

Todo el código de la memoria se encuentra alojado en el siguiente repositorio `https://github.com/DaniM/mscthesis`

Capítulo 1

Introducción

1. Motivación

En los últimos años la industria del videojuego se ha convertido en el sector del entretenimiento que más beneficios genera, por encima incluso de la música y cine juntos[1].

A medida que el mercado de los videojuegos crece, surgen metodologías para abarcar a un público mayor haciendo que los videojuegos se adapten a los diversos tipos de jugadores (ya sean ocasionales o habituales) ofreciéndoles a todos una experiencia atractiva.

Una de estas metodologías, con una clara tendencia creciente es el Ajuste Dinámico de la Dificultad (o Balanceo Dinámico de la Dificultad del Juego), que no sólo tiene como meta abarcar a un mayor número de jugadores para generar más ingresos, también tiene como objetivo alargar la vida útil del videojuego -¿para qué 30 horas de juego si el usuario lo deja a las 5?- haciéndolo más rejugable, así como una mejora sustancial de la jugabilidad (y por ende de la experiencia del usuario y de la calidad del título[13]) manteniendo al jugador en una zona de concentración durante más tiempo. Además, debido a que muchos adultos han crecido jugando a videojuegos, éstos tienen un amplio interés como herramienta de enseñanza (*Edutainment* o *Serious Games*)[23] [2]. Por ejemplo, podríamos tener a un niño concentrado aprendiendo un concepto como la suma

[1] `http://www.hoytecnologia.com/noticias/videojuegos-rivalizan-Hollywood-caja/165569`

[2] `http://www.wired.com/wired/archive/14.04/wright.html`

¡sin que el mismo se diese cuenta! [3].

Todo esto hace que un estudio más exhaustivo de dicha metodología sea necesaria para producir una nueva generación de juegos, más enfocados hacia todos los públicos y que ofrezcan una sensación más complaciente de cara al usuario final.

2. Descripción

Ser capaz de medir las habilidades de un jugador durante una partida y adaptarla de modo de que no sea ni demasiado aburrida ni demasiado difícil sin que éste lo perciba (*flow*). Este es el principal objetivo de la adaptabilidad en los videojuegos, en concreto aquí trataremos el ajuste dinámico de la dificultad (*Dynamic Difficulty Adjustment*, DDA) o balanceo dinámico de la dificultad del juego (*Dynamic Game Difficulty Balancing*). Un sistema que cambia automáticamente los parámetros, escenarios y comportamientos en un videojuego de acuerdo a las capacidades del jugador [36].

A pesar de haberse usado desde los 80 [11], el ajuste dinámico de la dificultad no ha irrumpido con fuerza hasta que juegos como *Flow* (Jenova Chen y Nicholas Clark, 2006) o *Left 4 Dead* (*Valve*,2008) -que aquí son analizados y estudiados- lo han aplicado. Los cuales han obtenido unas grandes críticas por parte de la prensa especializada, jugadores y desarrolladores, así como altos beneficios económicos. No obstante, y a pesar de la tendencia creciente, se trata de una metodología que se encuentra todavía en su infancia [26], donde no existe una documentación exhaustiva y en la que es necesario establecer una serie de metodologías o guías para poder implementarse en el mayor número de juegos posibles.

3. Objetivos

Como primer objetivo se pretende dar una introducción y estado del arte en lo que a videojuegos adaptativos se refiere. De esta forma el lector tendrá unos conocimientos amplios sobre el tema y estará capacitado para implementar ciertos mecanismos de aprendizaje usados en juegos. Además estudiaremos una serie de juegos en los que se ha aplicado con éxito el ajuste dinámico de la dificultad.

Tras esto, podremos entender el sistema propuesto, nuestro segundo objetivo: en un intento de sentar una metodología del ajuste dinámico de la dificultad

para un genéro de juego concreto. Una primera pieza sobre la que trabajar en el futuro y arrojar un poco más de luz a un campo que a pesar de estar en auge, y por razones obvias, no existe una gran documentación.

4. Estructura de la memoria

Esta memoria está dividida en los siguientes capítulos:

Introducción Se describe el trabajo a realizar, y el por qué de este.

Trabajos relacionados Se introducen una serie de tematicas que tienen algún tipo de relación con el ajuste dinámico de la dificultad.

Ajuste dinámico de la dificultad y videojuegos adaptativos Se abarca una amplia temática sobre aprendizaje y adaptabilidad en videojuegos. Se presentan de manera extensa varias técnicas usadas, con pseudocódigo y ejemplos reales. Definiremos qué es el ajuste dinámico de la dificultad, sus dificultades y errores. Se describirán los distintos tipos de ajuste dinámico de la dificultad conformando así el estado del arte.

Casos de estudio Se analizan una serie de juegos comerciales que han aplicado el ajuste dinámico con éxito.

Sistema propuesto Donde se propone un sistema de ajuste dinámico de la dificultad para un juego de narrativa lineal de acción.

Conclusiones y trabajos futuros Que conclusiones se extraen del trabajo y que lineas abiertas a seguir.

5. Código fuente

Todo el código de la memoria se encuentra alojado en el siguiente repositorio `https://github.com/DaniM/mscthesis`

Capítulo 2

Trabajos Relacionados

Hablar de ajuste dinámico de la dificultad, es hablar más tarde o más temprano de *Left 4 Dead* (L4D de aquí en adelante, *Valve*, 2008) y su figura del *Game Director* (Director de juego de aquí en adelante). De hecho el comienzo de esta memoria parte de ahí. Si bien es cierto que existen juegos previos que se ha usado con éxito un ajuste dinámico de la dificultad como *Max Payne* (*Remedy Entertainment*, 2001), *Half-Life 2* (*Valve*, 2004). No es hasta la aparición de este juego lo que ha hecho que este concepto sea reconocido ampliamente por jugadores y desarrolladores.

No obstante, se debe aclarar que el director de juego de L4D es algo más complejo que un ajuste dinámico de la dificultad. Está compuesto por una serie de subsistemas entre los que se encuentra el encargado de adaptar el ritmo de la partida haciendo aparecer más o menos enemigos durante el juego [7], que es lo que nos encargaremos de estudiar. A partir de aquí, hemos realizado un proceso de documentación y recapitulado una serie de trabajos e investigaciones que están relacionados de alguna forma con el ajuste dinámico de la dificultad. Las enumeraremos a continuación.

1. Emergencia

El concepto de emergencia describe las propiedades, comportamientos y estructuras que ocurren a alto nivel en un sistema, pero que no están presentes o no son predecibles a un bajo nivel. A partir de interacciones de entidades simples entre ellas y con el entorno local se crea algo nuevo que es más que la

suma de sus partes.

El ejemplo clásico es el cerebro, el cual no es sólo un conjunto de neuronas, es una máquina de pensar. Una persona no es sólo un conjunto de sistemas conectados, es un ente viviente. Una sociedad no es sólo un grupo de personas localizadas en un sitio, es una red capaz de realizar grandes cosas. El todo que es creado a partir de estas partes o colecciones tiene nuevas propiedades, comportamientos y estructuras. Esto es emergencia.

La emergencia puede ocurrir en diferentes grados y a diferentes niveles. Una distinción que vale la pena destacar es la diferencia entre emergencia local y emergencia global. La primera se trata de un comportamiento colectivo que aparece localizado en una pequeña parte del sistema, pudiéndose ser deducido a partir del comportamiento a bajo nivel de las entidades. Mientras que la segunda es cuando el comportamiento colectivo atañe al sistema como conjunto, donde difícilmente podremos deducir el comportamiento a partir del bajo nivel.

El potencial de la emergencia cobra más importancia cuando los elementos del sistema tienen capacidad de adaptación y son capaces de aprender.

Existen muchos campos donde se utiliza o estudia la emergencia, en el caso que aquí nos ocupa, videojuegos, podemos mencionar los trabajos de Penelope Sweetser [33],[34] o Harvey Smith [29] (entre otros) donde se tratan de establecer mecanismos y herramientas que faciliten o den lugar a situaciones no previstas de antemano en la partida.

2. Emergencia en videojuegos

La emergencia en videojuegos puede conseguirse con una serie de reglas de comportamiento global simples y definiendo propiedades en las interacciones de los objetos del mundo. Esta emergencia se produce cuando las acciones del jugador o las interacciones entre los objetos del mundo generan un comportamiento racional o aceptable, que no estaba planeado. Existen diversas técnicas para conseguir la emergencia:

- Razonamiento aproximado.
- Aprendizaje automático.
- Sistemas complejos (los autómatas celulares son usados frecuentemente para dotar de emergencia al mundo).

- Vida artificial.
- *Smart Objects* y *Smart Terrains*. Objetos y porciones de terreno los cuales emiten sus propiedades y guían los comportamientos de los agentes.

La emergencia a nivel local son comportamientos nuevos que no cambian la historia del juego, juegos como *Portal* (*Valve*, 2007) o *The Sims* (*Maxis*, 2000) son ejemplos de ello. También cabe destacar *Minecraft* (*Mojang*, 2011) como uno de los juegos con un comportamiento más emergente, donde los jugadores pueden crear historias a partir de los elementos "básicos" proporcionados por los desarrolladores.

La emergencia a nivel global se produce cuando las reglas y propiedades de bajo nivel crean un juego distinto con sus interacciones. Juegos como *Vampire: The Masquerade-Bloodlines* (*Troika Games*, 2004) o *The Elder Scrolls IV* (*Bethesda*, 2006) y *V* (*Bethesda*, 2011), en los que hay muchos personajes, organizaciones y misiones independientes. En los que el mundo es extenso con muchos objetos, enemigos, objetos y lugares. Donde existe un amplio abanico de interacciones posibles haciendo que dos partidas no sean iguales. Aunque normalmente sigue existiendo linealidad con una historia central que el jugador debe seguir para completar el juego, no obstante hay casos como *Mass Effect* (*Bioware*, 2007), *Vampire: The Masquerade* o *Façade* (*Procedural Arts*, 2005), donde dependiendo de las acciones y decisiones del jugador, el final del juego puede variar.

Los juegos emergentes son muy rejugables, ya que cada vez que el jugador comienza una partida, el juego realiza diferentes decisiones, las cuales cambian en su conjunto con las diferentes acciones que realice el usuario. Hacen el mundo del juego más interactivo y reactivo creando un amplio abanico de acciones y estrategias. Permite al jugador experimentar con más libertad y reducir el sentimiento de estar siguiendo un camino prefijado por los desarrolladores. Sweetser presenta en [33] cómo la emergencia en los juegos influye en su modelo de *Flow*, una especie de medidor de disfrute del jugador, siendo este un concepto muy importante y muy relacionado con el ajuste de la dificultad. Aunque a pesar de todo lo anterior, la emergencia todavía no ha sido explotada en los juegos debido a su dificultad para ser controlada[12].

Es necesario mencionar a modo de recordatorio (y para unir conceptos), que un ajuste dinámico de la dificultad produce un cambio a alto nivel del juego dependiendo de las interacciones del jugador.

[1] `http://aigamedev.com/open/article/chaos-theory-emergent-behaviors/`

[2] `http://en.wikipedia.org/wiki/Emergent_gameplay#Glitch_or_quirk-based_strategies`

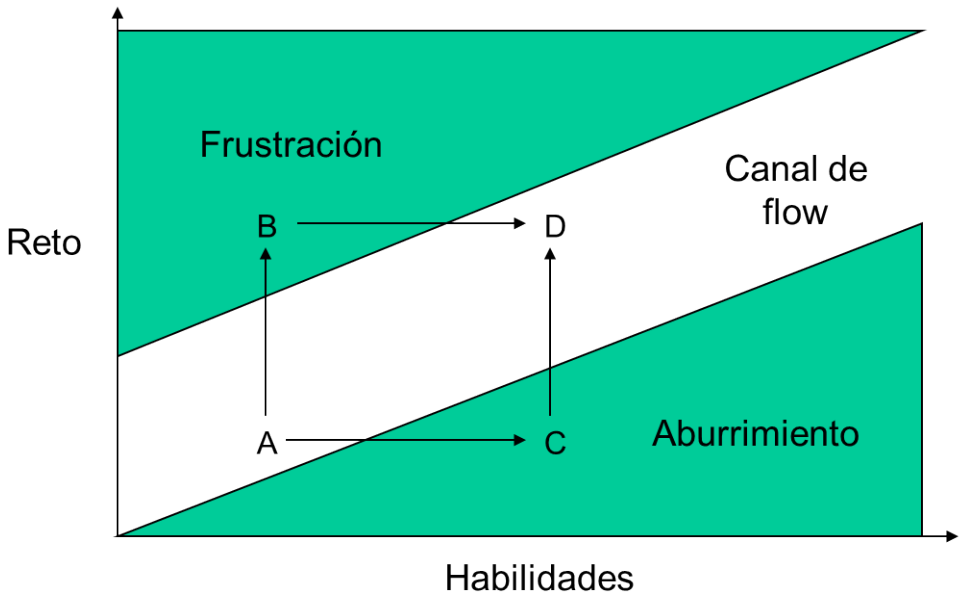


Figura 2.1: Definición de la zona de flow

3. El concepto de Flow

Hace 20 años, con la intención de explicar la felicidad, Mihaly Csikszentmihalyi estableció el concepto de *Flow*, una sensación de concentración completa en una actividad, con un alto nivel de satisfacción y realización [10].

Csikszentmihalyi desarrolló una serie de teorías para ayudar a las personas a entrar en su estado de *Flow*. Desde entonces, estas teorías se han usado en varios campos con el fin de diseñar mejores experiencias interactivas. Uno de los logros más reconocidos es la definición de la zona de *Flow* (figura 2.1)[10].

Para mantener a una persona dentro de dicha zona, la actividad necesita alcanzar un equilibrio entre el reto que presenta y las habilidades del participante. Si el reto que supone supera sus habilidades, la actividad se convierte en algo aplastante y generará ansiedad. Si el reto es más bajo que la habilidad, provocará aburrimiento. Sweetser [33] [34] presenta el modelo de *GameFlow* que relaciona los elementos del Flow de Csikszentmihalyi con otros entre los que destacamos los siguientes, por estar relacionados con el juego y que aparecen en los juegos emergentes:

- Concentración
- Desafío

- Habilidades del jugador
- Control
- Feedback

Sweetser afirma que la emergencia en los juegos tiene el potencial para aumentar la satisfacción del jugador apoyándose en los elementos del *GameFlow* y permitiendo unas interacciones más intuitivas, consistentes y emergentes con el mundo en el que se desarrolla el juego.

Sobre este mismo concepto y otro de los trabajos más famosos es la tesis de Jenova Chen [10], donde se presenta un juego, *Flow* (Chen & Clark, 2006) que puede jugarse desde navegador y que estudiaremos más adelante.

4. Emergencia Narrativa

La narrativa en un videojuego es la historia que está siendo contada, descubierta o creada por el jugador durante su progreso de la partida. Puede ser lineal y revelada al jugador en determinados puntos del juego, o puede ser el producto de las interacciones del jugador con el mundo. De acuerdo con Sweetser [33], si examinamos desde la perspectiva del jugador las formas de estructura narrativa en los videojuegos, podemos identificar tres categorías:

- Jugador como receptor. La línea argumental está escrita y simplemente es transmitida al jugador, sus acciones no pueden cambiar el devenir de ésta. Predomina en juegos lineales como la saga de *Call of Duty* (*Activision*, 2003) o *Gears of War* (*Epic*, 2006), por citar algunos.
- Jugador como descubridor. Los jugadores tienen descubrir la trama argumental que puede no ocurrir siempre en el mismo orden. Es más interactiva que la anterior ya que los jugadores tienen que desvelarla de forma más activa (hablando con otros personajes, explorando, completando misiones). Suelen tener distintas bifurcaciones y diferentes finales. Juegos con este tipo de estructura narrativa son *Heavy Rain* (*Quantic Dream*, 2010) y la saga *Might and Magic*.
- Jugador como creador. El argumento es una función de las acciones e interacciones del jugador, no se recibe o se descubre. El jugador crea una nueva historia al jugar. Juegos que generalmente se consideran que no tienen una trama definida entran en esta categoría. Algunos ejemplos son

SimCity (*Maxis*, 1989), *The Sims* (*Maxis*, 2000), *Civilization* (*MicroProse*, 1991) o *Minecraft* (*Mojang*, 2011)

Narrativa frente a emergencia

Existen según Sweetser [33], dos componentes principales para crear narrativa en un juego: las conversaciones y la historia. Para producir una narrativa emergente necesitamos emergencia en alguno de estos dos elementos. Del primero de los componentes, ya que quizás se aleja demasiado de la temática de esta memoria, tan sólo mencionar trabajos como el de *Façade* (*Procedural Arts*, 2005) y *The Restaurant Game* (Orkin & Roy, 2009) donde se realizan un procesamiento de lenguaje natural con bastante acierto.

La historia es la trama general, así como sus subtramas que se representan hasta el final del juego. Como hemos mencionado anteriormente, la historia puede ser recibida, descubierta o creada por el jugador, y normalmente suele presentarse en forma de *cutscenes* (cinemáticas) durante el juego.

Si consideramos las interacciones del jugador con personajes y objetos como los elementos a bajo nivel del mundo, la historia sería el comportamiento a alto nivel de éste. Si la historia es recibida (en contra posición de descubierta o creada), las acciones son irrelevantes en la trama, no hay conexión entre las interacciones a bajo nivel y el comportamiento a alto nivel. Cuando una historia es descubierta, las interacciones del jugador están forzadas a encajar con el comportamiento a alto nivel del juego. Podría decirse que es un enfoque top-down, donde las interacciones están determinadas por la historia [33].

Mientras en las historias que son creadas, las acciones e interacciones del jugador a bajo nivel forman la trama general del juego. Es aquí donde puede ocurrir la emergencia. Hay varios componentes que se usan para crear una historia en videojuegos [33], destacamos una por ser la que guarda una cierta relación con esta memoria: la narrativa. La narrativa es usada para dar una información adicional sobre la trama general.

Para conseguir una narrativa emergente debe haber un intercambio entre las acciones a bajo nivel y la trama a alto nivel para dar al jugador un rol en la creación de la historia y su impacto en el mundo. Un enfoque lineal podría ser por ejemplo, una suma ponderada y un umbral de activaciones. Cada vez que el jugador hiciera o dijese algo, movería el juego más cerca o más lejos de algunos puntos de la trama (figura 2.2). Una vez que la suma ponderada de dichas acciones superan un umbral de la trama, el argumento se dirige hacia esa dirección.

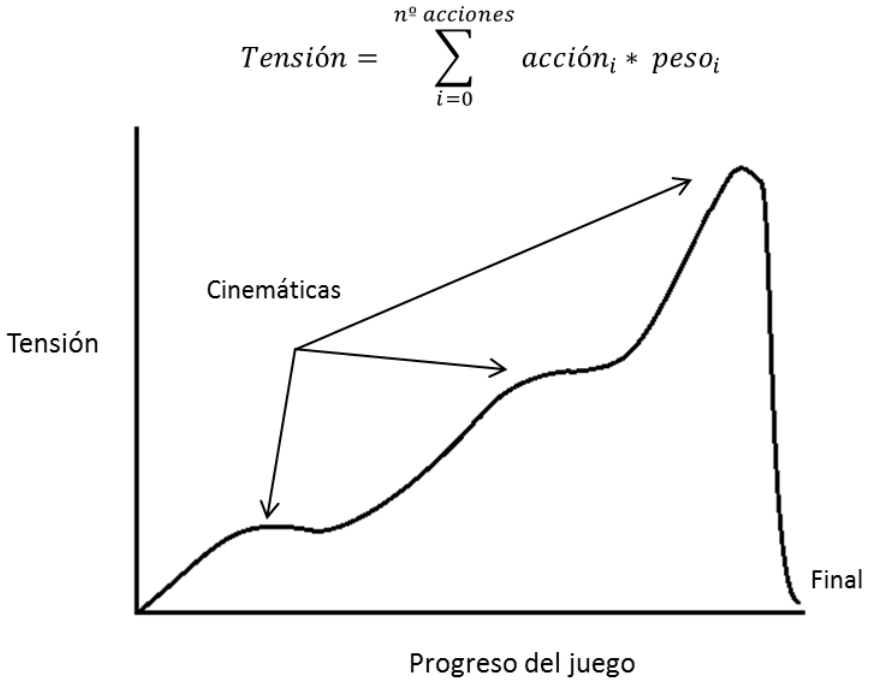


Figura 2.2: Progreso del juego como una suma de las acciones del jugador

Podríamos incluso usar un algoritmo que guie la trama y ayude a seguir un patrón de tensión dramática como en L4D [7]. Basado en la longitud del juego, las acciones del jugador, los niveles de acción del juego y las interacciones, se puede llegar a determinar cuándo es el momento para que algo cambie o algo nuevo ocurra. Hay más formas de generar una historia emergente aparte de la descrita anteriormente. La clave es conectar las acciones, interacciones y elecciones del jugador en el juego de modo que tengan cierto impacto en la historia que está siendo contada [34]. Existen herramientas software como *StoryBricks*[3][4](aunque a fecha de hoy se ha convertido en un estudio de desarrollo) o Erasmatron[5] las cuales permiten generar una emergencia narrativa. Existe una rama más específica sobre narrativa e inteligencia artificial en videojuegos, que no sólo trata la emergencia, también otros aspectos como la gestión del drama o la autoría procedural. Esta rama se denomina Inteligencia Narrativa.

[3]http://aigamedev.com/premium/interview/storybricks/
[4]http://aigamedev.com/open/interview/storybricks-preview/
[5]http://www.erasmatazz.com/

5. Inteligencia Narrativa

A finales de los 90 nació un grupo en el cual exploraban temas que intersectaban entre la inteligencia artificial y la teoría literaria. Al grupo se le puso el nombre de Inteligencia Narrativa (*Narrative Inteligence*, NI) y pronto comenzó a combinar ideas de filosofía, teoría de la comunicación y psicología con teorías computacionales sobre la mente y medios de comunicación [20]. Se puede decir que la Inteligencia Narrativa es una inteligencia artificial "humanística" [20]. Podemos considerar la inteligencia narrativa como la habilidad de una entidad para organizar y explicar las experiencias en términos narrativos (Mateas & Sengers, 1999). La inteligencia narrativa es clave en el proceso cognitivo empleado en un amplio espectro de experiencias desde el entretenimiento al aprendizaje. De ello se deduce que los sistemas computacionales que poseen inteligencia narrativa podrían ser capaces de interactuar con los seres humanos de una manera más natural ya que entienden contextos colaborativos como la narrativa emergente y son capaces de expresarse contando historias [18].

El alto grado de componente narrativo y comunicativo en los videojuegos hace que esta rama sea de interés. Siendo uno de los objetivos el de generar situaciones nuevas cada vez que se juega una partida.

Para Ian Bogost [6], este campo emergente en los videojuegos busca algo más que diversión. Es lo que da lugar a géneros como *advergaming*, *newsgaming*, *politicalgaming* y *educationalgaming*. Realiza una definición de retórica procedural, la forma en la que un videojuego da cuerpo a una ideología en su estructura computacional. Afirma que entendiendo esta retórica se pueden empezar a considerar aspectos como la enseñanza o las ideas políticas en los videojuegos.

Podemos mencionar los trabajos de Michael Mateas sobre la gestión del drama y como se han aplicado en el juego *Façade* (*Procedural Arts*, 2005) un drama interactivo en el que dependiendo de nuestras preguntas/respuestas (posee un procesador de lenguaje natural para frases cortas bastante efectivo) e interacciones irá generando diferentes situaciones y finales tratando de hacer una autoría procedural (es el juego el que va construyendo la historia). Esto último él lo define como IA Expresiva (*Expressive AI*), cuyo fin es el de crear agentes más ricos que simples NPCs (*Non Playable Character*, Personajes No Controlables), agentes con sus propios objetivos y deseos, que cambien conforme avanza el juego, con la interacción del jugador y en respuesta a la historia [6].

Versu[6] es un simulador de drama interactivo parecido a los libros de "elige tu

[6] `http://www.versu.com/`

propia aventura" con imágenes estáticas y una serie de acciones a elegir en cada situación (incluso no hacer nada). Pero a pesar de lo determinista que parece, los actores de este drama realizan sus acciones de manera independiente en función de sus propias creencias y deseos. Según los desarrolladores puedes jugar el mismo episodio realizando las mismas decisiones y puedes obtener distintos resultados.

L4D también vuelve a ser un ejemplo, aunque los creadores usaron el término de *Narrativa Procedural*[7] por el que generaban situaciones de tensión dependiendo del progreso del jugador en la partida.

A fin de generar un mayor número de situaciones y puesto que la Inteligencia Narrativa está asociada con el sentido común [18] se están utilizando técnicas de *crowdsourcing* para generar un amplio número de situaciones. Por ejemplo el juego The Restaurant Game (Orkin & Roy, 2009) donde los jugadores interpretan los diferentes roles en un restaurante cuyos comportamientos serán usados para hacer que los agentes aprendan comportamientos sociales. Los dos objetivos que pretendía afrontar este juego eran la dificultad de modelar un comportamiento adecuado con los sistemas de IA tan "artesanales" existentes, y poner de manifiesto que el testeo por parte de gente ajena a una desarrolladora llega demasiado tarde como para tener un impacto significativo en la calidad [8]. Cabe destacar la tésis sobre el crowdsourcing del propio Orkin [25] como trabajo de investigación.

Finalmente es necesario mencionar el trabajo de Chris Crawford y su herramienta Storytron/Erasmatron un motor para ejecutar narraciones interactivas[9] [20].

6. Videojuegos Adaptativos

La idea detrás de los juegos adaptativos es, como su propio nombre indica, adaptarse de alguna manera al jugador con el fin de proporcionar un reto apropiado o mejorar la experiencia. Ello suele implicar (aunque no es obligatorio) técnicas de aprendizaje. Casi cualquier aspecto del juego puede adaptarse, los comportamientos de los agentes que son personajes no controlables (NPCs), el ritmo dramático, el contenido, la narrativa...Este campo es el que ahondaremos con más detenimiento en el siguiente capítulo.

[7] http://www.edge-online.com/features/gabe-newell-writes-edge/

[8] http://web.media.mit.edu/~jorkin/restaurant/

[9] http://en.wikipedia.org/wiki/Chris_Crawford_(game_designer)#After_the_Dragon

El aprendizaje en videojuegos es algo que lleva intentándose aplicar durante años, pero realmente no es tan común en los videojuegos como se pudiera esperar [21]. Se tiene por norma que el aprendizaje se produzca durante el desarrollo del juego, nunca después de este. Esto atañe la adaptación directa [19] ya que al no tener control sobre los nuevos comportamientos generados se pueden dar lugar a situaciones indeseadas o aprender un concepto determinado que luego sea muy difícil o imposible de "desaprender". Técnicas de aprendizaje directo suelen ser la optimización (como los algoritmos genéticos), el aprendizaje por refuerzo (como el *Q-Learning*) y el aprendizaje mediante imitación (redes neuronales o árboles de decisión).

No obstante sí que existe a fecha de este trabajo un uso más común de técnicas de aprendizaje indirecto [19]. Se trata de mecanismos más simples que se basan en extraer estadísticas del juego para modificar el comportamiento del agente. Juegos como *Castlevania: Lords of Shadow* (*Mercury Steam*, 2010)[10] o *Batman: Arkham Asylum* (*Rocksteady Studios*, 2009) [11] por citar algunos modernos, ya lo incluyen. Este tipo de aprendizaje será en lo que se basará nuestra propuesta de sistema de ajuste dinámico de dificultad.

Existen numerosas publicaciones (libros, revistas, artículos) sobre aprendizaje y adaptabilidad en videojuegos, la literatura es muy amplia (los *AI Game Programming Wisdom* por citar alguno). Pero la realidad todavía es, que salvo en juegos puntuales donde el aprendizaje es la mecánica principal del juego como *Black & White* (*Lion Head*, 2001) el cual usa aprendizaje por refuerzo, existe todavía cierto escepticismo (y probablemente seguirá existiéndolo[11]) a la hora de introducir este tipo de tecnologías[12].

[10] `http://aigamedev.com/premium/interview/castlevania-lord-shadows/`
[11] `http://aigamedev.com/open/article/chaos-theory-emergent-behaviors/`
[12] `http://aigamedev.com/open/article/little-used-tools/`

Capítulo 3

Ajuste Dinámico de la Dificultad

1. Adaptabilidad en Videojuegos

La adaptabilidad en videojuegos se presenta cuando el juego se amolda o se adapta (valga la redundancia) al jugador de alguna manera. Se usa para mejorar aspectos de la experiencia de juego, como por ejemplo, proporcionar un reto adecuado. Casi cualquier aspecto de un juego puede ser adaptado: los comportamientos de los NPCs, el ritmo dramático, el contenido del juego, la narrativa, etc.

La adaptación por tanto es online, y aunque pueda implicar técnicas de aprendizaje automático, no es obligatorio. Basta con hacer un uso de adecuado de estadísticas in-game (aunque dependiendo del lector esto puede ser también aprendizaje automático), realizando lo que se conoce como un aprendizaje indirecto [19]. En el caso de utilizar técnicas de aprendizaje automático (o directo según [19]), la gran mayoría de este se realiza offline. Una vez lanzado el juego se produce el aprendizaje online, aunque como ya hemos dicho es una porción mucho menor.

Veremos con detalle varias técnicas que se han usado para implementar en los juegos. No pretendemos enumerar todas, tan sólo aquellas más famosas que se aplican o que se han aplicado con éxito a la hora de producir una adaptación en el juego.

1.1. La adaptación en la práctica

Esta sección describe con un poco más de profundidad los dos tipos de aprendizaje y adaptación que pueden darse en los juegos. La primera, es la llamada *adaptación indirecta*, la cual consiste en extraer estadísticas del juego para usarse en sobre la IA del agente y modificar sus comportamientos. Qué estadísticas usar y su interpretación en términos de cambios de comportamiento son decisiones tomadas por los diseñadores de la IA. La segunda técnica es la *adaptación directa*, que aplica algoritmos de aprendizaje al comportamiento de los agentes, no requiere mucha intervención humana más allá de especificar qué aspectos del comportamiento adaptar.

1.1.1. Adaptación Indirecta

La adaptación indirecta ocurre cuando un agente extrae información sobre el juego y con ella adapta su comportamiento. Por ejemplo, un *bot* en un juego de acción, puede aprender cual es el mejor sitio para vencer al jugador. Puede usar su IA "convencional" modificando el algoritmo de búsqueda de caminos para visitar dichas localizaciones en el futuro. El rol del aprendizaje se restringe a extraer información por lo que no toma parte en ningún cambio de los comportamientos del agente.

Actualmente (a fecha de la memoria) es la técnica más usada y la más recomendada, ya que ofrece las siguientes ventajas:

- Normalmente la información del mundo en las que se basan los cambios, puede ser extraída fácilmente y además suelen ser fiables. Lo que resulta en una adaptación rápida y efectiva.
- Como los cambios en los comportamientos están hechos por la propia IA del agente (la IA "convencional"), éstos están bien definidos y controlados, lo que lo hace más fácil de depurar y probar.

La mayor desventaja de esta técnica es que requiere tanto la información a aprender como los cambios que ocurrirán en los comportamientos en respuesta a éstos, los cuales están definidos a priori por los diseñadores.

Se pueden obtener una gran cantidad de comportamientos usando esta técnica, desde aprender lugares ventajosos y guiar la búsqueda de caminos hacia ellos, a aprender cuánto tiempo le lleva al jugador lanzar su primer ataque en un juego

de estrategia en tiempo real y decidir si expandir militarmente o económicamente, o que frutas del entorno son venenosas o buenas y guiar la elección de éstas. En cada uno de estos casos, el aprendizaje se basa simplemente en recoger estadísticas e interpretarlas para adaptar los comportamientos.

1.1.2. Adaptación Directa

Los algoritmos de aprendizaje se pueden usar para adaptar el comportamiento del agente de manera directa, normalmente probando dichas modificaciones y viendo si se producen mejoras. En la práctica esto se hace parametrizando de alguna forma el comportamiento del agente y usando un algoritmo de optimización o de aprendizaje por refuerzo para buscar dichos parámetros (y por tanto comportamientos) que ofrezcan el mejor rendimiento. Pongamos de nuevo el ejemplo del juego de acción, un bot puede tener una regla que controle el rango para usar o no un arma determinada. Esto puede ser parametrizado para decidir cuándo cambiar de arma.

La adaptación directa tiene un número de desventajas comparada con la adaptación indirecta:

- El rendimiento de los agentes con diferentes parámetros son evaluados empíricamente mediante su efectividad en el juego. Esto puede resultar problemático ya que:
  - Se debe desarrollar una fórmula que mida el rendimiento del agente y que refleje de manera real el objetivo del aprendizaje durante el juego.
  - El rendimiento de cada agente debe ser evaluado durante un periodo de tiempo sustancial para minimizar el impacto de los eventos aleatorios.
  - Se requieren demasiadas mediciones para la medición del rendimiento de cada agente sobre una muestra representativa de jugadores.
- La adaptación suele ser menos controlable que en el caso indirecto, haciéndola más difícil de probar y depurar. Esto incrementa el riesgo de que un comportamiento descubra una limitación del motor del juego, o un máximo inesperado en la medición de su rendimiento

El último de estos efectos puede paliarse restringiendo el alcance de la adaptación a pequeños aspectos del comportamiento del agente, y limitando la adaptación de éste.

A pesar de las desventajas, un gran punto a favor que ofrece la adaptación directa es que puede eliminar todas las desventajas anteriores, ya que puede desarrollar comportamientos nuevos. Por ejemplo, es posible producir un juego sin ninguna IA predeterminada, pero que use algún tipo de *algoritmo genético* (u otra técnica similar) que haga evolucionar las reglas de control del agente a medida que el juego avanza. Este sistema podría considerarse como una IA definitiva en el sentido que:

- Todos los comportamientos desarrollados por el agente se aprenderían a partir de la experiencia de éste en el mundo, y por tanto, no estaría restringido a las ideas preconcebidas de los diseñadores.

- La evolución de la IA sería abierta, es decir, no habría límites para la complejidad y sofisticación del conjunto de reglas, y por lo tanto, los comportamientos podrían evolucionar.

Por supuesto tal IA no es posible ya que no hay nada que nos garantice que el juego siga siendo jugable una vez se ha comenzado la adaptación. Nótese que esta objeción no puede ser lanzada si esto se realiza durante el desarrollo del juego ya que los resultados pueden ser validados mediante las prueba comunes de testeo y validación.

En resumen, la adaptación directa ofrece una alternativa a la adaptación indirecta, y que puede ser usada para adaptar ciertos aspectos del comportamiento del agente. Pero cuando no se sabe con certeza la forma exacta de adaptación establecida por los diseñadores. Focalizar la adaptación directa a un conjunto limitado y específico de comportamientos es la clave para que funcionen en la práctica.

Para terminar, mencionar un caso de éxito de adaptación directa, en concreto de aprendizaje por refuerzo, en [32] aplicado a la *Robocup*.

1.2. N-Gramas

Los N-Gramas son una técnica de aprendizaje indirecto que se usa para la predicción de acciones por parte del jugador. Un N-Grama realiza un seguimiento la frecuencia (de la cual podemos obtener la probabilidad) de cada secuencia de caracteres (entiéndase como un carácter un comando del juego, por ejemplo). Cada vez que ocurre una acción, el N-Grama registra las últimas n acciones, actualizando las frecuencias.

$$muestra(Ventana) = \sum_{a \in acciones} observaciones(Ventana, a)$$

$$P(q|Ventana) = \frac{observaciones(Ventana, q)}{muestra(Ventana)}$$

(Nótese como un 1-Grama normalizado son las probabilidades simples)

Por ejemplo, un 3-Grama mantendrá un histograma de todas las secuencias de tres opciones que haya visto. A la hora de realizar una predicción, los dos primeros elementos forman la ventana, y el 3-Grama elige como tercer elemento aquel que aparezca con más frecuencia (el más probable) o no hará predicción ninguna si las frecuencias son demasiado bajas. A continuación se expone un pseudocódigo de un N-Grama. La memoria también incluye una implementación básica hecha en python.

```
class KeyDataRecord:
    """Simplemente
        un diccionario en dónde se almacenan las frecuencias"""
    counts = dict()
    total = 0

class NGramPredictor:
    """N-Grama básico creado a partir del tamaño de una ventana y
        una función hash"""
    data = dict()
    _hash
                _windowSize

    def registerSequence(actionsSequence):
        """Registra una secuencia de acciones"""
        if _windowSize == 0:
            value = actionsSequence[0]
        else:
            value = actionsSequence[_windowSize]
        # usamos la función hash para generar la clave
        key = _hash(actionsSequence[0:_windowSize])

        # Si no se ha registrado esta entrada todavía, lo hacemos
        if not key in data:
            data[key] = KeyDataRecord()

        keyData = data[key]

        # Si el valor de la entrada no se ha registrado aún lo
            inicializamos a 0
        if not value in keyData.counts:
```

```
            keyData.counts[value] = 0

        keyData.counts[value] = keyData.counts[value] + 1
        keyData.total = keyData.total + 1

    def getMostLikely(previousActions, num_likely):
        """Coge las num_likely acciones más frecuentes dada una
           ventana de acciones"""
        key = _hash(previousActions)
        if not key in data:
            return None
        else:
            sorted_actions = sorted(data[key],reverse=True)
            return sorted_actions[0:num_likely]
```

1.2.1. Tamaño de la ventana

Incrementando el tamaño de la venta, podemos conseguir, en un principio, mejorar el rendimiento (predicción) de nuestro algoritmo. Aunque llegado un determinado tamaño sólo conseguiremos hacer peores predicciones que si hiciésemos decisiones aleatorias. Esto ocurre debido a que las acciones futuras son precedidas por una serie de acciones que raramente implican un proceso causal muy largo. Existe un equilibrio entre una ventana lo suficientemente grande que nos permita capturar de manera bastante precisa la forma en la que unas acciones influencian otras, y que al mismo tiempo no sea demasiado larga para "confundirse" por las acciones aleatorias del jugador. A medida que dichas acciones se hacen más aleatorias, es necesario reducir el tamaño de las ventanas. La figura 3.1 muestra la precisión de un N-Grama para diferentes tipos de ventanas.

Existen modelos matemáticos que estiman la calidad de un N-Grama a la hora de predecir, y que suelen usarse para encontrar el tamaño óptimo de ventana. La contrapartida de ganar un mayor poder predictivo es la mayor cantidad de memoria requerida y el mayor número de muestras para conseguir que el predictor sea lo suficientemente potente y alcanzar el rendimiento óptimo, lo que implica un mayor tiempo de aprendizaje. Aunque esto último puede resolverse usando la técnica que comentaremos a continuación, los N-Gramas jerárquicos.

1.2.2. N-Gramas jerárquicos

Cuando un N-Grama se usa online, hay un equilibrio entre el poder predictivo y el comportamiento del algoritmo. Un tamaño mayor de ventana puede

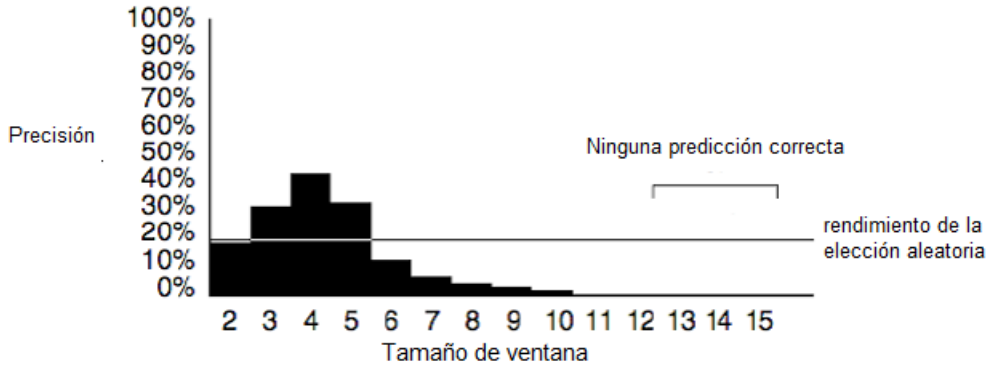


Figura 3.1: Precisión de un N-Grama para diferentes tipos de ventana usando 1000 muestras en un juego de elegir una de cinco opciones. Se puede apreciar como el mejor rendimiento lo da el 4-Grama y como a partir del 5-Grama se obtiene un peor rendimiento que elección aleatoria.

conseguir una mejora de rendimiento, pero también conlleva más tiempo el obtener uno razonable. Un N-Grama jerárquico tiene un conjunto de N-Gramas funcionando en paralelo cada uno de ellos con un tamaño de ventana más grande. Por ejemplo un 3-Grama jerárquico tendrá un 1-Grama, un 2-Grama y un 3-Grama trabajando sobre los mismos datos.

Cuando una predicción es solicitada, el algoritmo busca las acciones de la ventana en el 3-Grama. Si hay suficientes ejemplos, usa dicho 3-Grama para generar su predicción. Si no hay suficientes mira en el 2-Grama, si no tiene suficientes en el 1-Grama, y si definitivamente el N-Grama no tiene suficientes ejemplos, no devuelve nada o realiza una predicción aleatoria. No existe un umbral correcto para el número de entradas requerido para la confianza de la predicción, suele hacerse mediante ensayo/error. No obstante en el aprendizaje online y adaptación, es común que la IA haga decisiones con información incompleta, así que el umbral puede ser pequeño. A pesar de que el ámbito más académico suele ser más estrictos con la precisión de este umbral, en los videojuegos se suele relajar y utilizar un umbral relativamente pequeño.

Cabe mencionar que aunque el número de muestras dé un buen comportamiento en general, es sólo una aproximación. En lo que estamos interesados es en la confianza de un N-Grama en las predicciones que realiza y el número de muestras es sólo uno de los factores que intervienen en ésta. La confianza es una cantidad definida en la teoría de la probabilidad que mide la verosimilitud de haber llegado a algo por casualidad. Si la probabilidad de haber llegado a una situación de casualidad es baja, entonces la confianza es alta.

A continuación se expone un pseudocódigo de un N-Grama jerárquico. La memoria también incluye una implementación básica hecha en python.

```
class HierarchicalNGramPredictor:

    ngrams
    threshold
    _hash

    def registerSequence(actionsSequence):
        # Sólo actualizamos aquellos n-gramas que tengan un tamaño
            de ventana válido para esa muestra
        # Ojo, para actualizar correctamente n-gramas más pequeños
            que la muestra tendremos que llamar
        # varias veces a este método
        for i in len(actionsSequence):
            ngrams[i].registerSequence(actionsSequence[0:ngrams[i].
                windowSize+1])

    def getMostLikely(self, previousActions, num_likely):
        for i in range(len(self._ngrams)-1, -1, -1):
            # Comenzamos por el n-grama de tamaño mayor y vamos
                troceando la secuencia de acciones
            subsequence = previousActions[len(previousActions) -
                ngrams[i].windowSize : len(previousActions)]
            key = _hash(subsequence)
            if ngrams[i].data[key] and
            ngrams[i].data[key].total >= threshold:
                return ngrams[i].getMostLikely(subsequence,
                    num_likely)
        return None
```

1.3. Mapas de influencia

Herramienta de análisis táctico usada principalmente en juegos de estrategia que mantienen las zonas de influencia militar en el nivel. Existen muchos factores que atañen a dicha influencia: la proximidad de una unidad enemiga, una base bien defendida, la ventaja del terreno de alrededor...Incluso el clima. En la figura 3.2, se muestra un ejemplo de influencia militar entre dos bandos [21].

Los mapas de influencia también se usan para aprender IA táctica como una técnica de adaptación indirecta. Mantiene el equilibrio de la influencia actual en cada zona del nivel. Estos tipos de influencia pueden ser: lugares concurridos, lugares peligrosos (donde se han producido más muertes), lugares seguros, etc. En la mayoría de juegos en los mapas de influencia se aplica la siguiente sim-

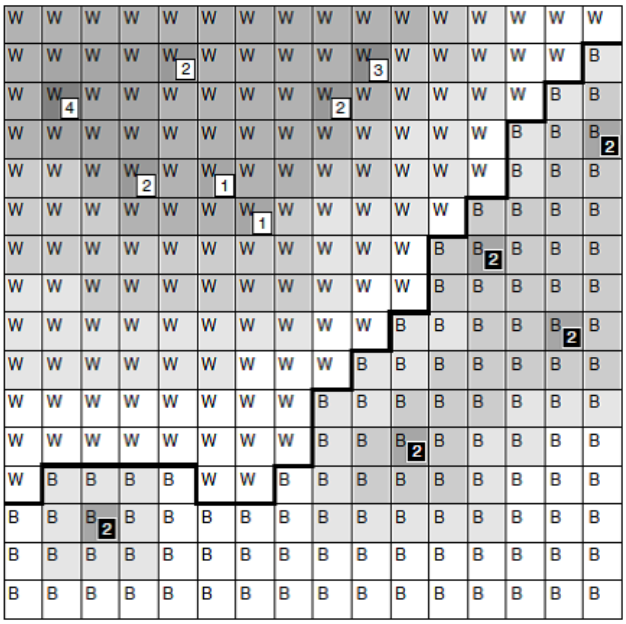


Figura 3.2: Ejemplo de mapa de influencias para todas las localizaciones en un pequeño ejemplo de juego de estrategia. Existen dos bandos blancos y negros (W y B). La influencia de cada unidad está representada por un número. El borde representa el area de control de cada uno.

plificación: la influencia de los elementos depende principalmente de un factor de proximidad. Si pensamos en el poder como una cantidad numérica, éste decae con la distancia: mientras más lejos, menor será dicha influencia, hasta que no sea perceptible. Podemos usar distintas fórmulas para modelizar esto, por ejemplo, una lineal (a doble de distancia, obtenemos la mitad de la influencia):

$$I_d = \frac{I_0}{d+1}$$

Donde I_d es la influencia dada la distancia, d, siendo I_0 la influencia en la distancia 0. En lugar de una función lineal, podemos usar otras para acentuar la caída como:

$$I_d = \frac{I_0}{(1+d)^2}$$

o también

$$I_d = I_0 e^{-d}$$

O para acentuar la caída pero aumentando el rango:

$$I_d = \frac{I_0}{\sqrt{1+d}}$$

Existen otros parámetros que se usan para modificar el rango como es el factor de decrecimiento δ. Normalmente este parámetro se usa para controlar la influencia dependiendo del tamaño del mapa: un valor bajo para mapas grandes y estratégicos, y un valor alto para mapas concretos y usados para decisiones tácticas [2]. Así las fórmulas ya mencionadas tendrían el siguiente aspecto:

$$I_d = \frac{I_0}{(1+d\delta)^2}$$

$$I_d = I_0 e^{-d\delta}$$

$$I_d = \frac{I_0}{\sqrt{1+d\delta}}$$

Otro parámetro que suele aparecer es el impulso α el cual controla o mezcla el valor de la influencia actual con el antiguo (tiene un comportamiento parecido a la tasa de aprendizaje), haciendo, por ejemplo, interpolación lineal entre ambos valores. Un valor del impulso alto (cercano a 1) hará que el algoritmo tienda a dar más importancia al "historico" que al valor actual, lo que resulta muy útil para almacenar estadísticas. De manera opuesta un valor bajo (cercano a 0) hará que el algoritmo tienda a usar los valores actuales de la influencia calculada, de modo que la propagación es más rápida y la predicción es más precisa.

También es posible que cada unidad tenga su propia fórmula o parámetros, lo cual reduciría la necesidad de tener varios mapas divididos por tipos de influencia o fórmulas, pero que, como veremos más adelante en el pseudocódigo, aumentan la complejidad del cálculo.

1.3.1. Cálculo de la influencia

Para calcular el mapa, necesitamos considerar cada fuente de influencia del juego para cada localización del nivel. Normalmente, el primer paso del algoritmo es establecer éstas en el mapa. Las fuentes de influencia pueden ser:

- Entidades, soldados aliados y enemigos, torretas, etc.

- Eventos como explosiones de granadas, fuego de balas, daño, etc.

Las influencias a considerar pueden ser aditivas (se añaden a la influencia actual) o totales (establecen el valor base), el tipo dependerá del juego, aunque normalmente los eventos temporales suelen ser aditivos, mientras que las entidades son totales.

Como este cálculo puede ser demasiado grande (miles de unidades por miles o millones de áreas) necesitamos usar una serie de técnicas para limitar este problema: el radio de efecto, los filtros de convolución y la inundación del mapa.

1. **Radio de efecto**
   La primera técnica es limitar el radio de efecto de cada unidad. Junto con la influencia básica, cada unidad tiene un radio máximo. Más allá de este radio, la unidad no puede ejercer ninguna influencia. El radio máximo puede ser establecido para cada unidad o podemos usar un umbral, es decir, a partir de una influencia mínima se considera cero. Por ejemplo para la fórmula de decrecimiento lineal, dado un umbral mínimo I_t, el radio de influencia sería:
   $$r = \frac{I_0}{I_t} - 1$$
   Este enfoque nos permite iterar sobre las unidades del juego, añadiendo su influencia sólo a los lugares que se encuentren dentro de su radio de alcance. La principal desventaja de esta técnica es que las pequeñas influencias acumuladas no aparecen en las distancias largas. Por ejemplo, un lugar rodeado por unidades de infantería pero fuera de su radio de influencias, puede no ser detectada ésta a pesar de estar rodeado de tropas.

2. **Filtros de convolución**

   La segunda forma de cálculo (la cual se proporciona un pequeño ejemplo hecho en NetLogo) aplica técnicas comunes de los gráficos por computador. Empezamos con un mapa de influencia vacío dónde sólo las unidades localizadas están anotadas. Entonces el algoritmo pasa a través de todas las áreas cambiando su valor, incorporando el suyo propio y el de sus vecinos. Esto tiene el efecto de difuminar los puntos iniciales, a mayor valor inicial, este efecto de difuminado llegará más lejos.

   Esta técnica usa un filtro, esto es, una regla que dice como el valor de un lugar es afectado por sus vecinos. Dependiendo del filtro, podemos obtener distintos efectos de difuminado. El más común y el que se usa en la implementación de ejemplo adjuntada es el filtro *Gaussiano*, el cual es

útil por sus propiedades matemáticas que lo hacen más fácil de calcular. Para realizar dicho filtrado, cada área del mapa necesita ser actualizada usando esta regla.

Todos los filtros de convolución tienen la misma estructura básica: una matriz que nos dice como se actualiza el valor de un área del mapa, basándose en su valor actual y en el de sus vecinos. Un ejemplo de filtro, en concreto un filtro Gaussiano, es el siguiente:

$$M = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Esto se interpreta de la siguiente forma: el elemento central de la matriz (que por lo tanto de tener un número impar de filas y columnas) se refiere al valor del área en el que estamos interesados. Con el valor actual de esa localización y la de sus vecinos, multiplicando cada elemento por su valor correspondiente en la matrix y sumando los resultados, tenemos el nuevo valor.

Repitiendo el proceso para cada una de las áreas del mapa obtenemos el nuevo mapa. No obstante cabe advertir que, si recorremos la matriz usando los nuevos valores que se van calculando en lugar de los antiguos tendremos una asimetría en el mapa, por lo que lo más común es tener dos copias de éste. Una que hace de fuente y que contiene los valores antiguos y es de solo lectura, y otra que hace de destino que almacena los nuevos valores (esta técnica se conoce como *double buffering*)

Para asegurarnos que las influencias se propagan a todas las áreas del mapa, es necesario repetir este proceso varias veces. Por cada repetición, establecemos el valor de la influencia en cada área que haya una unidad. No obstante y debido a que en un juego los tiempos de procesado son muy reducidos, es común realizar una pasada cada frame o incluso a través de múltiples de ellos. Una implementación en pseudocódigo del algoritmo de convolución es el siguiente (también se proporciona una demo hecha en NetLogo en la memoria):

```
class GridInfluenceMap:
    """Mapa de influencias representado por una matriz, cuya
       propagación se hace mediante un filtro de convolución
       """
    mfilter
    imap
    imap_b
    height
    width
```

```
def propagate():
"""Propaga la influencia de los valores del mapa"""
    # Iteramos todas las áreas
    # y aplicamos el filtro de convolución
    for i in xrange(height):
        for j in xrange(width):
            imap_b[i][j] = convolve(i,j)
    # Intercambiamos las variables
    imap, imap_b = imap_b, imap

def convolve(i,j):
"""Aplica
    el filtro de convolución a la celda correspondiente"""
    # Vamos a obviar el tema de los bordes y esquinas
    convResult = 0
    # La matriz de convolición es cuadrada
    size = mfilter.numRows() / 2
    for k in xrange(mfilter.numRows()):
        for m in xrange(mfilter.numColumns()):
            convResult += imap[i-size+k][j-size+m]
                          * mfilter[k][m]
    return convResult
```

3. **Inundación del mapa**

La última técnica utiliza una simplificación extra: la influencia en cada área es igual a la influencia mayor recibida de uno de sus vecinos. Esta simplificación puede conllevar errores, por ejemplo, la IA puede asumir que un gran número de unidades débiles pueden ser vencidas por una sola unidad más fuerte.

El algoritmo de propagación en sí es bastante simple, existen diferentes variantes, aunque este es un buen punto de partida (se adjunta en la memoria una implementación en python junto con un ejemplo):

```
def propagate():
"""Propaga la influencia de los valores del mapa"""
    # Abstraemos el bucle de las áreas ya que en este caso
        no tiene por qué ser una matriz de celdas
    for location in areaGraph:
        maxInfluence = 0
        neighbors = neighbors(location)
        for n in neighbors:
            distance = distance(location,n)
            # La función puede ser cualquiera de las que
                hemos mencionado previamente
```

```
                influence = decayFunction(imap[location],
                    distance,decay)
                maxInfluence = max(maxInfluence,influence)
            # Nótese el uso del impulso
            newInfluence = momentum * imap[location] + (1-
                momentum) * maxInfluence
            imap_bk[location] = newInfluence
        # doble buffering
        imap, imap_b = imap_b, imap
```

El hecho de que esta técnica sea la más fácil y rápida de implementar hace que los desarrolladores opten por ésta como implementación inicial. Este algoritmo es útil más allá de la simple propagación de la influencia, puede también incorporar análisis del terreno mientras realiza los cálculos.

Independientemente del algoritmo elegido, el equilibrio de "poder" o la influencia, no cambia drásticamente entre ciclo y ciclo (*frame*) por lo que es normal ejecutar estos algoritmos de forma distribuida a lo largo de varios frames ya que son fácilmente interrumpibles. También podemos elegir la frecuencia con la que ejecutarlos. Incluso cuando el mapa de influencias nunca esté completamente actualizado, incluso ejecutándose una vez cada 10 segundos, los datos suelen ser lo suficientemente útiles para la IA del juego.

1.3.2. Aprendizaje como IA táctica

Como ya hemos mencionado, esta técnica también puede usarse como aprendizaje, en concreto suele usarse para aprender IA táctica. Empezando por un mapa vacío, durante el juego, cuando un evento que consideremos interesante ocurre, cambiamos los valores de algunos sitios del mapa.

Por ejemplo, si intentamos evitar que nuestro agente caiga en una misma trampa repetidas veces, deberíamos en que sitio el jugador nos prepara con más frecuencia éstas. A pesar de que existen algoritmos que calculan los puntos de escondite o emboscada, estos no suelen adaptarse, además el propio jugador suele encontrar formas más creativas de preparar una trampa.

Para este problema podemos usar un "frag-map" (un mapa en el que apuntamos donde se ha producido el frag o muerte). Cada vez que la IA vea un personaje (incluido él mismo) ser golpeado, añade (o resta según lo midamos concretamente) un número, éste puede ser proporcional a la cantidad de daño o simplemente una cantidad fija para cuando un personaje cae.

De manera similar (usando otro mapa o usando la cantidad opuesta) si el

agente ve a un personaje alcanzar a otro, puede añadir (o restar) una cantidad proporcional al daño o fija para los golpes fatídicos.

Tras un tiempo construiremos un mapa que nos dirá qué zonas del juego son más peligrosas para visitar o cuales son buenas para vencer a enemigos. Este mapa es independiente de todo análisis, son datos aprendidos a partir de la experiencia.

Estos frag-maps también pueden usarse offline, durante la fase de pruebas para construir una buena aproximación de puntos potenciales en un nivel. Una aplicación común de este mapa offline es usarlo como base de otro online, que gradualmente cambie los valores (desaprenda) para adaptarse al estilo de juego del jugador. Si vamos reduciendo estos valores iniciales de forma gradual hasta llegar a cero, después de un rato el conocimiento estará basado en la información aprendida del jugador.

Para este caso, resulta muy importante la propagación de influencias, puesto que permite al agente generalizar a partir de los lugares que conoce sobre los que no tiene experiencia ninguna.

1.3.3. Aplicaciones

Los mapas de influencias nos permiten ver que zonas del juego son seguras o peligrosas, que además de usarlos para la toma de decisiones, podemos combinarlo con el algoritmo de búsqueda de caminos (*pathfinding*) para guiar los movimientos de los agentes cambiando las rutas que llevan a un destino con el fin de evitar (o pasar) áreas concretas.

1.4. Árboles de decisión

Los árboles de decisión pueden aprenderse, es decir, podemos construirlos a partir de un conjunto de observaciones y acciones. Estos árboles son usados para tomar decisiones durante el juego. Existe un amplio abanico de algoritmos de aprendizaje de árboles de decisión usados para clasificación, predicción y análisis estadístico. Los usados en videojuegos suelen estar basados en el ID3 de Quinlan, con el añadido de que soportan una actualización incremental del árbol (online).

1.4.1. ID3

El algoritmo básico de ID3 usa un conjunto de (entrenamiento) ejemplos de observación-acción. Las observaciones son llamadas normalmente atributos. El algoritmo empieza con una única hoja en el árbol de decisión y va asignándole ejemplos. Si no puede clasificarlos bien en un solo nodo, entonces divide el actual (en este caso el raíz) colocando los ejemplos es los nuevos grupos. Esta división se basa en el atributo que nos produzca el árbol más eficiente. Una vez se realiza la división, se siguen asignando ejemplos y se repite el mismo proceso cada vez que un nodo no pueda clasificar bien todos sus ejemplos.

El algoritmo es recursivo: empezando desde un nodo inicial va reemplazándolo con decisiones hasta que todo el árbol ha sido creado. En la creación de cada rama, se divide el conjunto de ejemplos entre los hijos, hasta que todos los ejemplos estén de acuerdo en una misma acción. Una vez se alcanza ese punto, no es necesario generar más ramas.

El proceso de división calcula en cada atributo la ganancia de información para cada posible división. La división que produzca la mayor ganancia se elige como la decisión para ese nodo. El pseudocódigo[1] del algoritmo se muestra en 1, también se incluye en la memoria una implementación en python.

1.4.2. Aprendizaje incremental de árboles de decisión

El algoritmo ID3 aprende el árbol mediante un único proceso, proporcionándole un conjunto completo de ejemplos, éste devuelve un árbol de decisión completo listo para usar. Esto está bien para un aprendizaje offline, pero la adaptación implica online, es decir, van a generarse nuevos ejemplos durante el juego y el árbol de decisión tiene que ser modificado para acomodarlos.

La forma más fácil de soportar esto, es simplemente volviendo a lanzar ID3 cada vez que se nos proporcione un nuevo ejemplo. Esto nos garantiza que árbol sea el mejor posible en cada momento. Desafortunadamente este proceso no es muy rápido y con el subsecuente crecimiento de la base de datos de ejemplos, puede ser prohibitivo.

Los algoritmos incrementales actualizan el árbol de decisión basándose en la nueva información, sin necesidad de tener que reconstruir todo el árbol desde el principio.

[1]Pseudocódigo extraído de las transparencias del tema 4 de la asignatura Aprendizaje Automático del Máster LCIA

Algoritmo 1 Algoritmo ID3

Entrada: Un conjunto de Ejemplos, un Atributo-objetivo, la lista de Atributos
Salida: Árbol de decisión

```
1: si todos los Ejemplos tienen el mismo valor para el Atributo-objetivo entonces
2:    devolver un nodo etiquetado con dicho valor.
3: si no, si Atributos está vacio entonces
4:    devolver un nodo etiquetado con el valor más frecuente del Atributo-
      objetivo en Ejemplos.
5: si no
6:    Sea A el atributo de Atributos que MEJOR clasifica Ejemplos.
7:    Crear Árbol con un nodo etiquetado con A.
8:    para todo los posibles valores v de A hacer
9:       Añadir un arco a Árbol etiquetado con v
10:      Sea Ejemplos(v) el subconjunto de Ejemplos con del atributo A igual
         a v.
11:      si Ejemplos(v) está vacío entonces
12:         Colocar como nodo hijo del árbol en el arco etiquetado con v el valor
            más frecuente de Atributo-objetivo en Ejemplos.
13:      si no
14:         Colocar como nodo hijo en el arco etiquetado con v el subárbol
            ID3(Ejemplos(v), Atributo-objetivo,Atributos-{A})
15:      fin si
16:   fin para
17: fin si
```

La aproximación más simple podría ser tomar un nuevo ejemplo y usar sus observaciones para recorrer el árbol. Cuando alcancemos un nodo terminal (hoja), comparamos la acción con la de nuestro ejemplo. Si son iguales, entonces no es necesaria ninguna actualización y simplemente podemos añadir el nuevo ejemplo a esa hoja. Si las acciones no son iguales, entonces la hoja se convierte en un nodo de decisión y separa los ejemplos como en ID3.

Este enfoque está bien como primer intento, pero tiene el problema que siempre añade los ejemplos al final del árbol, creando árboles con muchas ramas secuenciales. Idealmente, deberíamos poder crear árboles lo más "plano" posibles, dónde la acción a llevar a cabo pueda ser determinada lo más rápidamente posible.

1.4.3. ID4

El algoritmo incremental más simple y útil es el ID4, que como su propio nombre indica está relacionado con el ID3 básico.

Empezamos con un árbol de decisión creado por ID3. Cada nodo del árbol de decisión también mantiene un registro de todos los ejemplos que atañen a ese nodo. Los ejemplos que son pasados a ramas diferentes del árbol son almacenados en otra parte.

Para soportar un aprendizaje incremental, preguntamos a cualquier nodo del árbol que se actualice dado el nuevo ejemplo. Cuando hacemos esto pueden ocurrir tres cosas:

1. Si es un nodo terminal (representa la acción a tomar) y si el nuevo ejemplo comparte la misma acción, entonces el ejemplo se añade a la lista de ejemplos de ese nodo.
2. Si es un nodo terminal, pero el ejemplo no coincide con la acción a tomar. Usamos ID3 a partir de este nodo para determinar el nuevo árbol.
3. Si el nodo no es terminal (es un nodo con hijos), determinamos el mejor atributo que divida el conjunto de ejemplos junto con la nueva instancia.
    i Si el atributo es el mismo que el del nodo actual, determinamos a que nodo hijo corresponde y actualizamos éste recursivamente.
    ii Si el atributo es distinto, significa que el nuevo ejemplo hace que la decisión sea distinta, por lo que toda esa rama pasa a ser inválida. A partir de aquí construimos de nuevo el árbol usando ID3.

El pseudocódigo se muestra en 2, también se incluye en la memoria una implementación con una pequeña demo en python.

Algoritmo 2 Algoritmo ID4

Entrada: un Árbol de decisión, un conjunto de Ejemplos, un nuevo Ejemplo, un Atributo-objetivo, la lista de Atributos
Salida: Árbol de decisión actualizado con el nuevo ejemplo clasificado

```
1: si es un nodo hoja entonces
2:    Hacer Ejemplos = Ejemplos + {Ejemplo}
3:    si todos los Ejemplos pertenecen a la misma clase entonces
4:       Devolver Árbol.
5:    si no
6:       Árbol = ID3(Ejemplos, Atributo-objetivo, Atributos)
7:    fin si
8: si no
9:    Calcular el MEJOR atributo A, que clasifica Ejemplos + {Ejemplo}
10:   si A es la raíz del Árbol entonces
11:      Determinar a que nodo hijo pertenece el nuevo Ejemplo y actualizar
         hijo con ID4(hijo, Ejemplos(v), Ejemplo, Atributo-objetivo, Atributos
         - A)
12:   si no
13:      Hacer Ejemplos = Ejemplos + {Ejemplo}
14:      Árbol = ID3(Ejemplos, Atributo-objetivo,Atributos)
15:   fin si
16: fin si
```

Recapitulando, en cada nodo del árbol, ID4 comprueba si el atributo de éste sigue proporcionando la mayor ganancia de información. Si es así, continúa recursivamente en el hijo correspondiente. Si no, el árbol es reconstruido a partir de ese punto, haciendo en el peor de los casos el mismo trabajo que ID3.

En la práctica, para un tamaño sensato de ejemplos, ID4 es considerablemente más rápido que llamar repetidas veces a ID3 y generará árboles más planos que la implementación incremental básica propuesta en el punto anterior.

Problemas con ID4

ID4 y otros algoritmos similares pueden ser muy efectivos a la hora de crear árboles de decisión óptimos. Conforme empiezan a añadirse ejemplos, el árbol será reconstruido en gran parte y a medida que la base de datos de ejemplos crezca los cambios en el árbol serán más pequeños, manteniendo la velocidad de ejecución.

No obstante, podemos enfrentarnos a conjuntos de ejemplos en los cuales el orden de comprobación de los atributos en el árbol es patológico: el árbol necesita ser reconstruido casi siempre. Esto puede llegar a ser incluso más lento que lanzar el ID3 en cada paso. ID4 es a veces incapaz de aprender ciertos conceptos. Esto no signifique que genere árboles inválidos, simplemente que no son estables conforme aparecen nuevos ejemplos.

A pesar de ello, no es muy común sufrir de este problema en videojuegos. Los datos tienden a estabilizarse bastante rápido e ID4 acaba siendo significativamente más rápido que ID3. Otros algoritmos incrementales son ID5, ID6 o ITI. Éstos usan trasposiciones y registros estadísticos para cada nodo de decisión u operaciones adicionales de reestructuración del árbol con el fin de evitar reconstruirlo repetidamente.

1.5. Aprendizaje por refuerzo. Q-Learning

Aprendizaje por refuerzo es el nombre que se le da a un determinado número de técnicas que usan aprendizaje basado en la experiencia. Suelen estar compuestas por tres componentes principales: una exploración, una función de recompensa y una ecuación de actualización del aprendizaje que une las dos anteriores.

Se trata de un método de adaptación directa, en el que el objetivo es aprender la relación entre la acción tomada por el agente en un estado determinado y su bondad, es decir, la bondad del par estado-acción.

Por ser la más simple de implementar y de ajustar sin necesidad de entender en profundidad sus propiedades teóricas, aquí trataremos una de las técnicas perteneciente a la familia diferencias temporales, el *aprendizaje-Q*. Es llamado así por mantener el valor o la calidad (valor Q) para cada posible par estado-acción que ha probado. El valor Q representa como de buena cree el agente que es esa acción.

1.5.1. El problema

Tenemos un agente el cual nos gustaría que mejorase la elección de sus acciones con el tiempo. Qué define a una acción como buena, puede ser difícil de decidir a priori, ya sea para los diseñadores o para el experto. Depende de la manera de actuar del jugador o del entorno en el que se encuentra el agente.

En un principio el agente puede tener libre elección de las acciones y calcular

que acciones son las mejores para una situación. La complicación viene cuando la calidad de una acción no está clara en el momento que se realiza y el agente debe aprender acciones que conlleven a un evento, pero que en el momento de realizarla no proporcionan feedback.

1.5.2. La representación del mundo en el aprendizaje-Q

El aprendizaje-Q trata al entorno como una máquina de estados finita. En cualquier instante de tiempo, el algoritmo se encuentra en algún estado. El estado debe codificar todos los detalles relevantes sobre el entorno y los datos internos.

En un juego los estados están formados por muchos factores: posición, proximidad del enemigo, nivel de salud, etc. El aprendizaje-Q no necesita entender todos los elementos que componen el estado, simplemente puede ser un entero, el número del estado. Por su parte el juego debe de ser capaz de trasladar el estado a un número que pueda usar el algoritmo de aprendizaje.

El aprendizaje-Q es lo que se dice un método libre de modelo ya que no intenta construir el modelo de cómo funciona el entorno, todo lo trata como estados, lo que es más fácil de implementar.

Para cada estado, el agente debe saber que acciones tiene disponibles. En muchos juegos todas las acciones están disponibles todo el tiempo. No obstante, para entornos más complejos, algunas acciones pueden no estar disponibles cuando el agente está en un estado determinado.

Después de que un agente realice una acción en el estado actual, la función recompensa debe devolver algún tipo de feedback, que puede ser positivo, negativo o cero si no está muy claro como de buena es la acción tomada. Aunque no hay límites en los valores de las recompensas normalmente suelen estar en el intervalo [-1,1].

No existe ningún tipo de restricción que obligue a que la recompensa sea la misma cada vez que la amima cción se lleve a cabo en un estado determinado. Puede haber otra información contextual no usada para crear dichos estados, es decir, omitimos cierta información a la hora de crear éstos con el óbjetivo de reducir el número. Aunque el algoritmo no pueda aprovecharse de esto al no ser parte del estado, puede tolerar sus efectos y aprender el éxito general de una acción en lugar de basarse en un solo intento. De la misma forma, llevar a cabo repetidas veces la misma acción en el mismo estado no implica que se alcance siempre el mismo estado siguiente. Puede haber varios factores influenciándolo.

Una de las bondades de este algoritmo (y de la mayoría de los pertenecientes al aprendizaje por refuerzo) es que puede lidiar con este tipo de incertidumbres.

Estos cuatro elementos: el estado inicial, la acción tomada, la recompensa y el estado siguiente. Forman lo que se llama la tupla de experiencia, a menudo escrita como $\langle s, a, r, s' \rangle$.

1.5.3. Aprendizaje

La tupla de experiencia está dividida en dos partes. Los dos primeros términos (estado y acción) se usan para buscar el valor Q almacenado. Los otros dos elementos (la recompensa y el nuevo estado) se usan para actualizar el valor Q basándose en como de buena es la acción y como de bueno es el siguiente estado.

La fórmula de actualización es la siguiente:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

Aunque también podemos encontrárnosla escrita así:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(a,s))$$

Dónde α es la tasa de aprendizaje (*learning-rate*) y γ el factor de descuento, ambos parámetros del algoritmo. $\gamma \max_{a'}(Q(s',a'))$ mira el nuevo estado de la tupla de experiencia de entre todas las acciones posibles de ese estado y eligiendo aquella con el valor Q más alto

La ecuación de aprendizaje-Q combina linealmente dos componentes usando la tasa de aprendizaje, cuyo rango es $[0,1]$. El primer componente $Q(s,a)$ es simplemente el valor Q actual para ese estado y acción. Manteniendo parte del valor actual nunca perdemos conocimiento adquirido previamente.

La segunda componente también tiene dos elementos. El valor r la recompensa de la tupla de experiencia. Si la función refuerzo fuese:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha r$$

Entonces estaríamos combinando el valor Q antiguo y el nuevo feedback de la acción.

El segundo elemento $\gamma \max_{a'}(Q(s', a'))$ mira el nuevo estado de la tupla de experiencia de entre todas las acciones posibles de ese estado y eligiendo aquella con el valor Q más alto. Esto ayuda a propagar el éxito de una acción: si el siguiente estado es bueno, el actual también debería serlo un mínimo.

El factor de descuento controla cuanto influye el estado futuro. Un valor muy alto generará una gran atracción hacia los estados considerados como buenos, mientras que un valor bajo prácticamente sólo afectará a estados cercanos al éxito. El factor de descuento debe de estar en el intervalo $[0, 1]$.

Recapitulando, el valor Q es una combinación del valor actual con el nuevo, el cual también es influido por la calidad del estado futuro.

1.5.4. Exploración

Las técnicas de aprendizaje por refuerzo también requieren una estrategia de exploración: una política para seleccionar las acciones a tomar en un estado determinado. Suele llamarse simplemente política.

En el sentido más estricto, seguir una estrategia no es parte del algoritmo de aprendizaje-Q, aunque sí es muy común. Por ejemplo, la estrategia de exploración para el algoritmo básico es parcialmente aleatoria. La mayoría del tiempo elige la acción con el valor Q más alto desde el estado en el que se encuentra, pero otras veces lo hará de manera aleatoria (esto suele controlarse mediante un parámetro).

Una alternativa puede ser incorporar acciones del jugador, generando tuplas basadas en su forma de jugar.

1.5.5. Algoritmo y parámetros de ajuste

Si el problema se mantiene estático y las recompensas son consistentes (lo cual ocurre a menudo a menos que se basen en eventos aleatorios del juego), los valores Q acabarán convergiendo y continuar ejecutando el algoritmo no conllevará cambios. Llegado a este punto ha aprendido el problema completamente.

Para pequeños problemas de juguete esto se puede conseguir en unas cuantas de miles de iteraciones, pero en problemas reales conlleva mucho más. En una

aplicación práctica no habrá tiempo suficiente para alcanzar la convergencia, por lo que es común que el agente comience a actuar bajo la influencia de los valores aprendidos antes de que se complete todo el proceso de aprendizaje.

Para aclarar cómo funciona el algoritmo de aprendizaje-Q, presentamos un pseudocódigo [2] y comentaremos los diversos parámetros.

El Aprendizaje-Q en el algoritmo 3 divide el aprendizaje en episodios, esto es, transicionar entre estados hasta llegar a uno terminal

Algoritmo 3 Aprendizaje-Q episódico

Entrada: Un Problema, del cual podemos saber el estado actual s y la recompensa r, Q una tabla de valores de acción indexada por el estado y la acción, un número de Episodios, una tasa de aprendizaje α, un factor de descuento γ, un factor de aleatoriedad en la exploración ρ

Salida: Q actualizada

```
1:  para todo Episodios hacer
2:    hacer s un estado aleatorio de Problema
3:    mientras s no sea un estado terminal hacer
4:      sea A el conjunto de acciones disponibles de s
5:      sea r un número aleatorio entre [0,1)
6:      si r < ρ entonces
7:        seleccionar de manera aleatoria una acción a de A
8:      si no
9:        hacer a la acción cuyo valor Q[s,a] es el mayor
10:     fin si
11:     sea s',  el resultado de aplicar la acción a, al estado s de Problema
12:     sea r,  la recompensa de Problema[s,a,s'] de transicionar a ese estado
13:     sea Q, Q[s,a]
14:     sea QMax, el valor del mejor par acción-estado Q[s',a']
15:     Q(s,a) = (1 − α)Q + α(r + γQMax)
16:     hacer s = s'
17:   fin mientras
18: fin para
```

Como es posible que en un juego no existan estados finales muy definidos, podemos aplicarlo como en el algoritmo 4.

Si el aprendizaje-Q se diseña para operar online, debemos cambiar la función de aprendizaje para que sólo realiza una iteración por llamada y mantenga el estado actual así como sus valores Q.

[2] También se presenta una implementación en python con un ejemplo de prueba sacado de [27]

Algoritmo 4 Aprendizaje-Q

Entrada: Un Problema, del cual podemos saber el estado actual s y la recompensa r, Q una tabla de valores de acción indexada por el estado y la acción, un número de Iteraciones, una tasa de aprendizaje α, un factor de descuento γ, un factor de aleatoriedad en la exploración ρ, un factor de longitud de secuencia de estados ν

Salida: Q actualizada

```
hacer s un estado aleatorio de Problema
para todo Iteraciones hacer
    sea r un número aleatorio entre [0,1)
    si r < ν entonces
        hacer s un estado aleatorio de Problema
    fin si
    sea A el conjunto de acciones disponibles de s
    sea r un número aleatorio entre [0,1)
    si r < ρ entonces
        seleccionar de manera aleatoria una acción a de A
    si no
        hacer a la acción cuyo valor Q[s,a] es el mayor
    fin si
    sea s',  el resultado de aplicar la acción a, al estado s de Problema
    sea r, la recompensa de Problema[s,a,s'] de transicionar a ese estado
    sea Q, Q[s,a]
    sea QMax, el valor del mejor par acción-estado Q[s',a']
    Q(s,a) = (1 − α)Q + α(r + γQMax)
    hacer s = s'
fin para
```

Para el almacenaje de datos podemos usar usar un diccionario indexado por el par acción-valor. Sólo aquellas tuplas que se hayan explorado estarán guardadas en la estructura de datos, en caso de no estar tienen el valor por defecto de 0. El problema de esta aproximación es que no nos devolverá la mejor acción sino hemos visitado todas las duplas (supongamos que todas las visitadas tienen valores negativos y al no tener constancia de las no visitadas, la mejor será aquella cuyo valor negativo sea menor).

No obstante, el aprendizaje-Q está diseñado para pasar por todos los estados y acciones posibles (y varias veces), por lo que una matriz de valores Q inicializada a 0 puede ser una mejor solución, más rápida y tendrá en cuenta todas las posibles opciones.

La tasa de aprendizaje α. Controla la influencia del feedback actual sobre el valor Q. Su rango es $[0, 1]$. Un valor de 0 da como resultado un algoritmo que no aprende. Un valor de 1 descarta toda experiencia previa.

En muchos algoritmos de aprendizaje automático, la tasa de aprendizaje disminuye conforme pasa el tiempo. Inicialmente comienza con un valor relativamente alto (por ejemplo, 0.7) y tras un tiempo se reduce gradualmente hasta un valor bajo (digamos 0.1). Esto permite aprender más rápido los cambios de los valores Q cuando hay poca información y "protegerlos" de los nuevos datos aprendidos más tarde.

El factor de descuento, γ. Controla cuanto influye en el valor Q de una acción, el estado al que lleva. Su rango es $[0, 1]$. Un valor de 0 implica que sólo se tiene en cuenta la recompensa directa, haciendo que el algoritmo no aprenda estrategias a largo plazo que supongan una secuencia de acciones. Un valor de 1 daría la misma importancia a la recompensa de la acción actual que al valor Q del estado al que se llega.

Valores altos favorecen secuencias más largas de acciones, pero tardan más en aprenderse. Valores bajos se estabilizan antes, pero sólo soportan secuencias relativamente cortas.

Aleatoriedad en la exploración, ρ. Este parámetro controla con que frecuencia el algoritmo tomará una acción aleatoria en lugar de la mejor acción conocida hasta el momento. Su rango es $[0, 1]$. Un valor de cero hará que se centre en explotar el conocimiento que tiene, mientras que un valor de uno, desembocará en una estrategia continua de exploración (siempre intentará pro-

bar nuevas cosas). Este balance es bastante dependiente de si el algoritmo se utiliza offline u online.

Si el aprendizaje va a realizarse online, el jugador querrá ver algún tipo de comportamiento inteligente, por lo que el algoritmo debería explotar su conocimiento. Por lo tanto ρ deberá ser bajo (0.1 o menos, estaría bien).

Para un aprendizaje offline, querremos que aprenda el máximo posible. No obstante, a pesar de que es preferible valores mas altos, también existe un contrapunto. Si hemos llegado a un estado bueno, entonces otros estados y acciones similares también pueden ser buenos, es decir, guiar la exploración hacia valores Q altos es una buena estrategia para encontrar otros pares acción-valor buenos.

Longitud de la secuencia de estados, ν. La longitud de la secuencia controla el número de iteraciones que se llevarán a cabo como una secuencia conectada de acciones. Al igual que los anteriores su rango es $[0, 1]$. Un valor de cero significa que el algoritmo siempre usará el estado alcanzado en la iteración anterior para la siguiente. Esto tiene la ventaja de poder descubrir secuencias de acciones que lleven al éxito, pero tiene la desventaja de permitir que el algoritmo se estanque en un número relativamente pequeño de estados en el cual no hay salida salvo mediante una secuencia de acciones cuyo valor Q es bajo (lo que se conoce como mínimo local).

Un valor de uno implica que en cada iteración se empieza por un estado aleatorio. Si todos los estados y acciones son igualmente posibles, entonces esta es la estrategia óptima ya que explora el mayor número de estados y acciones en el menor tiempo posible. Aunque en la práctica, unos estados son más interesantes que otros y deben tener preferencia a ser explorados sobre otros. Esto se consigue dejando al algoritmo realizar secuencias de acciones.

Muchas estrategias de exploración de aprendizaje por refuerzo, ni siquiera tienen en cuenta este parámetro y asumen que tiene el valor cero. Por ejemplo en un aprendizaje online, el estado usado en el algoritmo, está directamente controlado por el estado del juego, por lo que es imposible saltar a un estado nuevo aleatorio.

1.6. Redes Neuronales

Antes de nada hemos de decir que muy poca teoría sobre las redes neuronales es aplicada a los juegos. Algunas de ellas son reducir el número de estados a

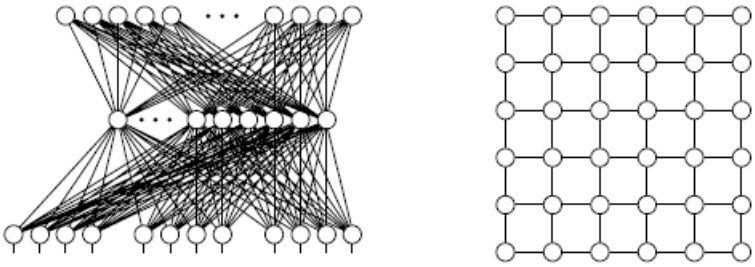


Figura 3.3: Dos tipos de topologías o arquitecturas de redes neuronales. A la izquierda el perceptrón multicapa, a la derecha una red de Hopfield

almacenar para un aprendizaje por refuerzo [21], modelar deseos de los agentes a través del perceptrón simple [12], controladores [8]. No obstante, merece ser mencionada, ya que una de las líneas de investigación del ajuste dinámico (y de la generación de contenido) de la dificultad se basa en ellas [37][28][9]. También aclarar que las redes neuronales en sí son un amplio campo de investigación que está resurgiendo en los últimos tiempos [3] y que en la memoria nos ceñiremos a los modelos más básico que son el perceptrón simple y el perceptrón multicapa.

1.6.1. Visión General

Una red neuronal consiste en una red de nodos relativamente simples, cada uno de ellos ejecutando el mismo algoritmo. Estos nodos son las neuronas que se comunican con un subconjunto de ellas en la red. Se conectan de una determinada manera, siguiendo un patrón que corresponde al tipo de red. A estos patrones característicos se les conoce como la arquitectura o topología de la red. La figura 3.3 muestra dos arquitecturas típicas de una red neuronal. Normalmente (figura 3.3 izquierda), los nodos suelen estar agrupados en capas.

En muchos tipos de redes neuronales (como es el caso del perceptrón multicapa), algunas conexiones son específicas de entrada y otras de salida, este tipo de redes se conocen como *feedforward*. Las entradas suelen ser proporcionadas por el programador o usuario, y la salida suele proporcionar o hacer algo útil. Las redes feedforward pueden tener bucles, conexiones que se encuentren más adelante en la red pueden conectarse de nuevo con otras de capas anteriores. A este tipo de arquitectura se le conoce como redes recurrentes, las cuales suelen ser producir comportamientos muy complejos e inestables, siendo mucho más difíciles de controlar.

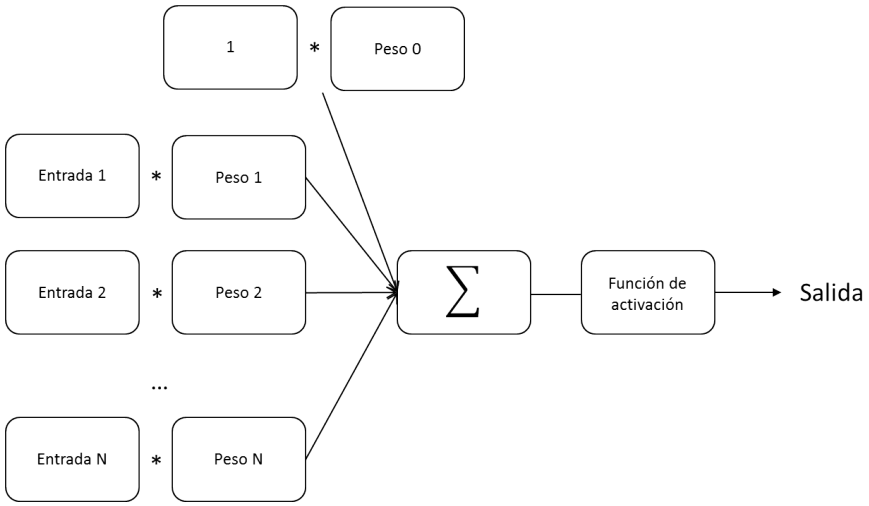
[3] `https://www.coursera.org/course/ml`

Otros tipos de redes pueden no tener conexiones de entrada y salida específicas. Cada conexión es entrada y salida al mismo tiempo (Hopfield, figura 3.3 derecha).

Las redes neuronales suelen usarse para clasificación y regresión o aproximación de funciones. Durante el resto de la sección nos referiremos principalmente al problema de clasificación.

1.6.2. El perceptrón simple

Se trata de la unidad básica de una red neuronal, genera una salida basada en las entradas. El funcionamiento del perceptrón, aparece resumido en la figura 1.6.2.



Cada entrada tiene asociado un peso θ correspondiente por el que es multiplicado, además se suele añadir un peso adicional de guía (que es equivalente a añadir un valor de entrada adicional que siempre valga uno). La suma de las entradas multiplicadas por el peso se pasan a la función de activación o hipótesis, h_θ, la cual determina la salida del perceptrón.

Dependiendo del tipo de problema h_θ puede implementarse de diversas maneras, no obstante, las más comunes suelen ser: La función lineal

$$h_\theta = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n$$

para regresión, y la función sigmoide

$$h_\theta = \frac{1}{1 + e^{-z}}$$

para clasificación.

Para aplicar el aprendizaje en nuestro perceptrón partimos de una serie de ejemplos con su salida correspondiente y unos pesos que queremos aprender, de modo que la salida de la función de activación o hipótesis corresponda con la de los ejemplos. El algoritmo más simple que se puede aplicar es el del descenso por el gradiente, en el cual tratamos de minimizar el error (a menudo llamado función de coste) $J(\Theta)$ cometido en las salidas del perceptrón a partir del conjunto de ejemplos[4].

Algoritmo 5 Descenso por el gradiente

Entrada: Un vector de pesos Θ, un conjunto de entrenamiento $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$, una función de coste J, un número de iteraciones ITS y una tasa de aprendizaje α

Salida: Θ actualizados

1: **repetir**
2: inicializar Δ a un array del tamaño de Θ a 0
3: **para** i=1 **hasta** m **hacer**
4: $coste = J(\Theta, x^{(i)}, y^{(i)})$
5: $\Delta = \Delta + J'(\Theta, x^{(i)}, y^{(i)})$
6: **fin para**
7: $\Theta = \Theta - \alpha * \Delta$
8: **hasta que** número de iteraciones igual a ITS **or** el coste sea inferior a un umbral determinado

Aclarar la línea 5 del algoritmo, en la que para economizar espacio, se ha puesto que J' devuelve el vector gradiente para ese ejemplo. Una forma más "iterativa" sería: $\Delta_j = \Delta_j + \frac{\partial}{\partial \theta_j} J(\Theta, x^{(i)}, y^{(i)})$ para cada uno de los pesos θ_j que conforman Θ

Este pseudocódigo del descenso por el gradiente, describe un procesamiento por lotes, es decir, actualizamos los pesos tras procesar todos los ejemplos del conjunto de entrenamiento-Otra de las formas posibles sería iterar sobre el conjunto de entrenamiento y actualizar los pesos tras procesar cada ejemplo. Nótese como el algoritmo va acercándose en dirección al mínimo (local[5]) poco a poco con el gradiente J' y el factor α, que tiene la misma función que la tasa de aprendizaje vista en el aprendizaje por refuerzo. Las funciones de coste (y

[4]Recordar al lector que el perceptrón simple no puede clasificar de manera correcta conjuntos que no son linealmente separables, recuérdese el problema XOR

[5]Recuérdese los problemas de este algoritmo en caso de existir mínimos y máximos locales

sus derivadas parciales) suelen ser [6]:

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m} = \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_j}J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\cdot x_j^{(i)}$$

Siendo m el tamaño del conjunto de ejemplos e $y(i)$ la salida esperada para el ejemplo i-ésimo, es decir, el error cuadrático medio. Esto respecto a regresión, para clasificación

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

$$\frac{\partial}{\partial\theta_j}J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\cdot x_j^{(i)}$$

Pongamos un ejemplo extraído de [12] (y que se usó para Black & White, por lo tanto es un ejemplo real) que se basa en un agente *BDI* cuyos deseos están modelados mediante perceptrones simples:

Supongamos que el deseo de nuestro agente para comer algo es el hambre, los sabroso que este el objeto a comer y como de infeliz esté. Supongamos que $\alpha = 0{,}1$ y todos los pesos están inicializados a 0.5. La tabla 3.1 [7] nos muestra cinco iteraciones del descenso por el gradiente para un conjunto de cinco ejemplos.

Cuadro 3.1: Cinco iteraciones del descenso por el gradiente, con sus correspondientes correcciones de pesos

Iteración	Bias (peso 0)	Hambriento	Sabroso	Infeliz
1	0,96900891	0,5012154	0,49215675	0,47440597
2	0,93863429	0,50263303	0,48453163	0,44912619
3	0,90888176	0,50425329	0,47712569	0,42416721
4	0,87975585	0,5060762	0,46993958	0,39953503
5	0,85126005	0,50810146	0,46297361	0,37523506

Nótese como el feedback va reduciendo el peso de sentirse infeliz conforme pasan las iteraciones, con el objetivo de reflejar más la importancia de estar hambriento o que el objeto sea sabroso con el deseo de comer, o también, que un agente infeliz no tendrá ganas de comer.

[6] las fórmulas que aquí aparecen son las del cálculo del error en lote, para hacerlo de forma individual basta con eliminar el m de la ecuación

[7] Se adjunta un código fuente para que el lector los compruebe por si mismo

1.6.3. El problema

Antes de entrar en el perceptrón multicapa conviene recapitular que problema intentamos solucionar a través de ésto. Queremos clasificar un conjunto de valores de entrada, para poder actuar de una manera determinada por cada agrupación que obtengamos. Tras leer esto (sino antes), cabe preguntarse ¿por qué no usar un árbol de decisión?

Si las clasificaciones son lo suficientemente complejas, las interacciones de los factores pueden ser demasiado difíciles de identificar y resultará imposible generar reglas mediante este método. Los árboles de decisión son precisos, cuando se les proporciona una situación que no está representada en los ejemplos, toman una decisión basándose en ellos. Debido a esto no son buenos generalizando y extrapolando.

Con las redes neuronales ocurre lo contrario, no son tan precisas, incluso pueden dar respuestas erróneas para ejemplos del conjunto de entrenamiento. No obstante, son mucho mejores generalizando y extrapolando. Es este equilibrio entre precisión y generalización la clave para elegir una técnica u otra.

1.6.4. El perceptrón multicapa

El problema del perceptrón simple es que sólo puede expresar o aprender conjuntos linealmente separables. Es por eso que usamos redes con más capas, para ser capaces de expresar decisiones no lineales. Como ya hemos mencionado con anterioridad nos centraremos en el perceptrón multicapa como tipo de red.

Aprendizaje

El propósito del aprendizaje es reemplazar los pesos de las conexiones inicializados de manera aleatoria por unos cerca del óptimo. Las redes neuronales difieren en la forma en la que implementan el aprendizaje. En concreto, el perceptrón multicapa opera de dos formas:

- El modo normal (feedforward), el cual comparte con el perceptrón simple, que consiste en darle una serie de datos de entrada, operar de alguna forma, y obtener una salida. Este proceso no involucra aprendizaje.

- El modo aprendizaje, cuya forma más común es el algoritmo de retropropagación (también parecido al del perceptrón simple). Dónde la red opera

hacia delante, con cada capa generando su salida a partir de la capa anterior, y luego hacia atrás para corregir el error. Se trata de minimizar el error producido en el conjunto de ejemplos dado.

De modo que necesitamos un método que, al igual que un árbol de decisión, aprenda a partir de un conjunto de entrenamiento o ejemplos. El algoritmo debe ser capaz de generalizar con el fin de abarcar todas las eventualidades, y también debe ser capaz de adaptarse, permitir añadir nuevos ejemplos y aprender de los errores.

Algoritmo de retropropagación

El algoritmo de retropropagación aprende los pesos de un perceptrón multicapa dada un conjunto de neuronas e interconexiones fijas. Al igual que en el perceptrón simple se apoya en un descenso por el gradiente para minimizar el error producido entre la salida de la red y sus correspondientes ejemplos.

Al tratar ahora con multiples salidas en lugar de sólo una como ocurría con el perceptrón simple, la función de coste o error cambia ligeramente, pasando a ser

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - h_\Theta(x^{(i)})_k)\right]$$

para clasificación y

$$J(\Theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}(h_\Theta(x^{(i)})_k - y_k)^2\right]$$

para aproximación de funciones. Siendo K el número de salidas posibles de la red.

Una de las principales diferencias con el perceptrón en lo que a espacio de búsqueda se refiere, es que el error puede tener multiples mínimos locales. Lo que conlleva que el descenso por el gradiente pueda converger a uno de estos mínimos.

Conviene aclarar ciertas notaciones usadas:

Algoritmo 6 Algoritmo de retropropagación para un problema de clasificación

Entrada: Un número de capas L y los pesos Θ correspondientes a cada capa $\Theta^1, ..., \Theta^L$, un conjunto de entrenamiento $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$, una función de coste J y una tasa de aprendizaje α

Salida: Θ actualizados

1: **repetir**
2: $\Delta_{ij}^{(l)} = 0$ (para todo l, i, j)
3: **para** i=1 **hasta** m **hacer**
4: Hacer $a^{(1)} = x^{(i)}$
5: Realizar una propagación feedforward para calcular $a^{(l)}$ desde $l = 2, 3, \ldots, L$
6: A partir de $y^{(i)}$, calcular $\delta^{(L)} = a^{(L)} - y^{(i)}$ (el error de la salida)
7: Calcular $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
8: Hacer $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_i^{(l)} + \delta_j^{(l+1)}$ para cada neurona de i de la capa l que conecta con cada una de las neuronas j de la capa siguiente $l + 1$
9: **fin para**
10: Hacer $D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}$
11: Hacer $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$
12: **hasta que** cumplir algún tipo de condición

- $a^{(l)}$ representa el vector de salida de la capa l. Siendo $l = 1$ la capa de entrada y $a^{(L)}$ la salida de la red. Aclarando un poco más sobre esta representación vectorial, $a_j^{(l)}$ representa la salida de la neurona j de la capa l.

- $\delta^{(l)}$ calcula el error de cada neurona en la capa l[8]. Esta fórmula puesta de manera no vectorial, para un problema cuya función de activación es sigmoide, es la siguiente:

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_{j \in salidas} \theta_{ij}^{(l)} \delta_j^{(l+1)}$$

El algoritmo 6 es la versión por lotes del descenso por el gradiente. Podemos apreciar como a partir de una red fija iteramos sobre el conjunto de entrenamiento. Por cada ejemplo aplicamos la red en feedforward para obtener la salida y una vez hemos procesado todos los ejemplos actualizamos los pesos. Este proceso sobre el conjunto de entrenamiento se repite hasta que los resultados obtenidos por la red sean satisfactorios.

[8]Dependiendo del texto, el cálculo de $\delta^{(L)}$ tiene distintas implementaciones en [24], por ejemplo hace también la derivada de la salida $\delta^{(L)} = a^{(L)}(1 - a^{(L)})(a^{(L)} - y^{(i)})$. Recuérdese que la derivada de una sigmoide $g(x)$ es $g(x)(1 - g(x))$

El procedimiento de actualización de pesos es bastante parecido al del perceptrón simple con la ligera variación de que en las neuronas de las capas ocultas no tenemos un valor de salida y para calcular el error. Es por eso que sumamos los errores producidos por las neuronas de la siguiente capa en las cuales influye ésta, de forma que la hacemos "responsable" de una manera ponderada del error de salida de la red.

Existen diversas formas de aplicar este algoritmo, tanto de manera supervisada como no supervisada, online y offline. Aparte de ellas, cabe destacar una forma más de aplicarlo, que respecta a como operamos con el conjunto de entrenamiento. Estos tipos de aprendizaje son:

- **Entrenamiento incremental.** Actualiza los pesos por cada ejemplo de dado, el descenso por el gradiente suele conocerse también como descenso estocástico. Esto implica que la convergencia no queda determinada a un número de valores fijos. No obstante, hace posible el aprendizaje online, ya que si procesamos de nuevo todos los ejemplos tendríamos una situación parecida a lo que ocurría con ID3. Se ejecuta el algoritmo de retropropagación hasta que se obtienen resultados satisfactorios. Para prevenir que el algoritmo se estanque en un subóptimo, se suele añadir un factor de impulso o momento a las diferencias de los pesos.

  El principal problema de esta técnica es la selección de la tasa de aprendizaje, que puede ser algo específico de la aplicación. Una solución a esto es la aproximación estocástica o el enfriamiento, que reduce paulatinamente la tasa de aprendizaje para forzar más la covergencia al mínimo global.

- **Entrenamiento por lotes o *batch*.** Es el que se muestra en el algoritmo 6. Se procesa todo el conjunto de entrenamiento antes de actualizar los pesos. Este proceso se aplica offline, ya que si el conjunto de entrenamiento es muy grande, el tiempo se dispararía. Esta técnica cuenta con una gran ventaja, es posible incluso prescindir de la tasa de aprendizaje y puede converger automáticamente a una solución.

Es necesario aclarar, que existen combinaciones de ambos como un incremental con un tamaño más reducido de los lotes, o usar un entrenamiento por lotes offline previo para partir de una base y luego aplicar el incremental de manera online.

1.6.5. El aprendizaje Hebbiano

Para terminar la sección vamos a introducir brevemente una técnica de aprendizaje no supervisado para redes neuronales y que parece que puede resultar de utilidad en el mundo de los videojuegos [21][33], el aprendizaje Hebbiano. A pesar de que se puede usar en cualquier tipo de red, el aprendizaje Hebbiano suele aplicarse en redes con estructura de "rejilla" (figura 3.3, derecha), donde cada nodo está conectado con sus vecinos.

El funcionamiento normal (el que no aprende), es el mismo que hemos estado viendo en los apartados anteriores: se suman de manera ponderada las entradas y se decide el estado a través de la función de activación. En este caso las entradas se toman de las neuronas vecinas en lugar de la capa anterior.

La regla de este aprendizaje se basa en que si un nodo tiende a compartir el mismo estado que su vecino, el peso entre estos dos nodos debe incrementarse. De manera opuesta, si un nodo tiende a un estado diferente, hay que decrementar el peso.

Si dos nodos suelen compartir el mismo estado, entonces puede existir alguna razón por la que estén relacionados, es por eso que incrementamos el peso entre dos neuronas vecinas si una de ellas está activa. Si por el contrario, no existe ninguna correlación, las neuronas no compartirán el mismo estado tan frecuentemente, y el peso de sus conexiones se incrementará con la misma frecuencia que se decrementará. No se debilitará o reforzará una conexión de manera global.

El aprendizaje Hebbiano es usado para encontrar patrones y relaciones en los datos más que para generar una salida. Puede ser usado para regenerar la pérdida o falta de información en los datos.

Pongamos por ejemplo un juego de estrategia en tiempo real (figura 3.4), donde se tiene un conocimiento parcial de las fuerzas enemigas debido a la llamada niebla de guerra (una característica común de estos juegos). Podríamos usar una red neuronal con una estructura de rejilla y aplicar el aprendizaje Hebbiano. La rejilla representaría el mapa de juego y los vecino serían dependiendo del juego 1, 4, 9.

El estado de cada neurona indica si la celda correspondiente es segura o no. Con un conocimiento completo, la red puede entrenarse a partir del conjunto completo de celdas seguras o peligrosas generadas por un mapa de influencias.

Tras una serie de partidas, la red puede predecir los patrones de zonas seguras a partir de los estados de las neuronas. La IA establece los estados de

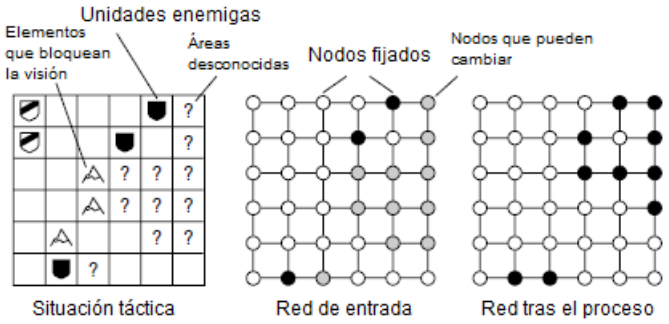


Figura 3.4: Aprendizaje Hebbiano como mapa de influencia

las neuronas que se corresponden con celdas visibles, éstas quedan fijadas y no cambian. Tras esto, el resto de la red comienza entonces su proceso de suma y activación hasta llegar a una situación estable que determina cuales de las zonas no visibles son seguras o cuales deben ser evitadas.

2. Qué es el Ajuste Dinámico de la Dificultad

El ajuste dinámico de la dificultad (*Dynamic Difficulty Adjustment*, DDA), también conocido como balanceo dinámico de la dificultad (*Dynamic Game Balancing*, *DGB*), es el proceso por el cual se cambian de manera automática y en tiempo real parámetros, escenarios y comportamientos en un videojuego, basándose en las habilidades del jugador. El objetivo que se pretende con esto es evitar que el jugador se aburra (demasiado fácil) o se frustre (demasiado difícil) para así mantenerlo concentrado durante toda la partida a la vez que se le proporciona un nivel de desafío más a medida. Además gracias a esta podemos marcar el ritmo del juego (ritmo adaptativo) y controlar o mantener una tensión dramática [26][7] como en la figura 3.5.

Tradicionalmente, la dificultad de un juego se incrementaba a lo largo de la partida de manera suave o a través de fases o niveles. Los parámetros de estos incrementos solo podían modularse al principio del juego seleccionando un nivel de dificultad. No obstante, esto puede conllevar experiencias frustrantes tanto para jugadores noveles como aquellos más experimentados, ya que intentan seguir una curva de aprendizaje o dificultad predefinida. El ajuste dinámico de la dificultad intenta remediar esto intentando crear una experiencia más adaptada al jugador. A medida que éste mejora sus habilidades, así lo hace el reto. Sin embargo, implementar tales elementos en el juego no es una tarea fácil, y como resultado, estos métodos no están muy extendidos [36].

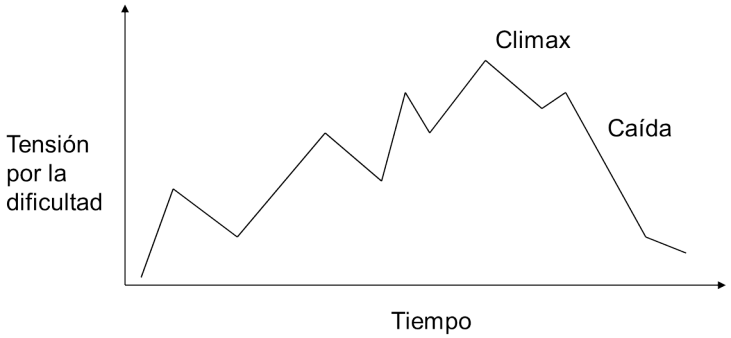


Figura 3.5: Curva de ejemplo de cómo influir en la tensión del jugador gracias a la dificultad

Mencionar, a modo de curiosidad, que esta técnica existe desde 1982, siendo *Xevious* (*Namco*, 1982) [11] el primer juego que lo implementa. Aunque no siempre es comentado o documentado.

2.1. Controversias y dificultades del ajuste dinámico de la dificultad

2.1.1. Controversias

En géneros donde la progresión del jugador es una parte fundamental del juego, el ajuste dinámico de la dificultad se encuentra en entredicho. Estos tipos de juegos están basados en un mundo con una amplia variedad de enemigos, con su dificultad acoplada a la pericia del jugador para fortalecer a su personaje.

Esto motiva al jugador a mejorar su efectividad general, ya sea subiendo de nivel o con ítems, para acceder o solventar los retos que se le presentan en las partes más difíciles del mundo. La motivación y satisfacción producida por la progresión de nuestro avatar queda reducida cuando el entorno se adapta reduciendo automáticamente su propia efectividad o dificultad para hacer el juego más accesible en base al estado actual del personaje. En lugar de permitir al jugador esforzarse por incrementar la efectividad del personaje para superar el reto.

Llegado a un punto y en el peor de los casos, el jugador podría no tener ningún tipo de sensación de logro o superación o de mejora de su personaje, animando al usuario a seguir el "camino de la mediocridad" donde rebajar sus

habilidades conlleva a un curso más fácil en el discurrir del juego [36] [9]. No obstante un ajuste de la dificultad no tiene po rque implicar necesariamente una reducción de dificultad, Hunicke y Chapman [16] proponen una variante en el cual el sistema se adapte para mantener al jugador en la situación más difícil posible (discomfort) y así aumentar el reto.

2.1.2. Dificultades

Una de las principales dificultades que tiene el DDA es el hecho de que requiere una gran inversión centrada en las pruebas [10] ya que se han de introducir jugadores nuevos al juego durante el ciclo de prueba de éste. Esta incorporación continúa de personal, además del número que se requiere puede resultar muy costosa económicamente.

Otra dificultad es el balanceo propio del juego, que trata de ajustar los parámetros de éste de forma que ofrezca un reto justo al mayor tipo de jugadores [11]. Este proceso se da en todos los juegos, pero puede verse incrementado aún más si tratamos de implementar un DDA, dónde tendremos que especificar, por ejemplo, no sólo que un enemigo de tipo A es más fuerte que uno de tipo B, sino cuanto (de una manera cuantitativa) más fuerte. Aquí también entrarían ajustes de heurísticas (como la función de "reto"), los parámetros a medir del jugador, los pesos de estos, etc. Ya que se pretende que sea algo más que dar más "botiquines" al jugador cuando esté bajo de salud [12].

La monitorización del juego tiene que ser lo más fiable posible ya que son los parámetros que usará la heurística para determinar la pericia del jugador [5].

Para terminar una muy obvia, hacer que el ajuste sea invisible para el jugador, ya que no sólo se rompe la experiencia[13], sino que también podría aprovecharlo en su favor y explotarlo [26][11][14]. Es por eso que para mantener esa invisibilidad también es necesario saber cuándo realizar los ajustes [16] y no sólo eso, deben detectarse rápido [Andrade et al]. Además el jugador debe poder ganar o perder aún con un sistema de ajuste que controle todo con exactitud

[9] http://www.gamefeil.com/gdc2001.html

[10] http://aigamedev.com/plus/article/procedural-director/

[11] http://gamasutra.com/blogs/AlexanderKerezman/20100430/87204/Structure_or_AI_Director.php

[12] http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml

[13] http://dukenukem.typepad.com/game_matters/2004/01/autoadjusting_g.html

[14] http://dukenukem.typepad.com/game_matters/2004/01/autoadjusting_g.html

[11]. Yendo más allá y relacionándolo con el apartado anterior de controversia, Spronck propone que el DDA sea algo opcional a elegir por el propio jugador [31] y Pfeifer [26] que directamente desaparezca si no está bien hecho, ya que un ajuste de la dificultad mal hecho puede arruinar el juego.

2.2. Errores

Existen bastantes juegos que han implementado un ajuste dinámico de la dificultad que ha acabado de alguna forma no tan satisfactoriamente como debería. A continuación expondremos ciertas "pifias" que se cometen al implementar este sistema.

2.2.1. Escalar demasiados parámetros

En el juego *Oblivion* (*Bethesda*, 2006) los enemigos escalaban su nivel basándose en nivel del jugador. Por ejemplo, los guardias de la ciudad siempre tenían entre 2 y 5 niveles más, los bandidos de 2 a 5 menos, y así con los demás personajes. Lo que conllevaba que el jugador pudiese ir a cualquier lugar del mundo, proporcionándole un reto ajustado. El problema fue que escalaron demasiados parámetros del juego como el nivel, el tipo de enemigo, lo que llevaban consigo éstos (o lo que dejaban al ser derrotados) o los ítems que se podían vender. De modo que llegado a un momento del juego ciertos elementos desaparecían de éste y encontrabas en su lugar enemigos mucho más fuertes con las mejores armaduras del juego. ¿Qué problemas generó esto? Pues que submisiones que consistían en matar a un tipo de enemigo, no podían ser completadas al no existir éstos más en el juego debido a que habían sido sustituidos por otros más poderosos[35].

2.2.2. Escalar el nivel del juego indiscriminadamente

Encontrado en Oblivion también y debido al escalado de nivel mencionado al principio del punto anterior. Nos encontrábamos con submisiones en los que tenías que mantener con vida un grupo de aliados resultaban imposibles ya que los enemigos podían vencerlos de un golpe, lo que hacía perder el sentido de progresión al subir de nivel [35].

Final Fantasy VIII (*SquareEnix*, 1997) también tenía problemas con el escalado de nivel, dándose el problema de "nivel 100" donde el juego tornaba más

difícil al tener el nivel máximo [15].

En Oblivion, también teníamos el caso totalmente opuesto, si elegíamos no subir nunca de nivel el juego pasaba a ser excesivamente sencillo.

Otro de los fallos más famosos del DDA en referente a escalar nivel del juego, fue en *Homeworld 2* (*Relic*, 2003), donde el número de tipos de naves de la flota enemiga en la misión actual estaba basado en el número de naves que habías tenido al final de las misiones anteriores. Lo que conllevo que en muchas guías y estrategias del juego recomendaban destruir toda tu flota al final de cada misión para evitar ser apabullados en la siguiente [35].

2.2.3. La goma

El fallo más común de IA en los juegos de carreras. De manera sencilla, la goma da una ventaja determinada a la IA dependiendo de la posición del jugador. Si el jugador lo hace mal, la IA irá más despacio para adaptarse a éste. Si lo hace bien, todos los coches (o al menos uno) irán más rápido. Esto se hace con el objetivo de hacer la carrera más ajustada, pero se obtiene un sentimiento opuesto, el de una carrera injusta y frustrante que da la sensación de que la computadora hace trampas ya que realmente no se tiene en cuenta la pericia del jugador. Es en estos géneros donde existen más detractores del ajuste dinámico de la dificultad [35]. Juegos famosos que usan esta técnica son *Burnout* (*Electronic Arts*) y *Mario Kart* (*Nintendo*). Para paliar un poco este fallo existen una serie de mejoras "simples" que hacen la goma menos obvia y que simplemente las mencionaremos [1]:

- Dividir en diversos grupos la carrera: la cabeza de carrera, los medios y los lentos; y que exista cierta competición en estos grupos.
- No aumentar la velocidad directamente, en su lugar hacerlo en trozos mediante eventos.
- Con respecto a los tiempos que está realizando en qué posición debería estar de acuerdo a la heurística de reto o pericia, y determinar si es necesario realizar un ajuste. ¿Está mejorando o empeorando los tiempos? Y usar los eventos mencionados anteriormente.

[15] `http://www.gamefaqs.com/ps/197343-final-fantasy-viii/answers?qid=150470`

2.3. Usos recientes en videojuegos

Max Payne (*Remedy*, 2001) ajustaba de manera muy leve una ayuda para apuntar más rápido, y subía o bajaba un poco la vida de los enemigos dependiendo de cómo de bien lo esté haciendo el usuario en ese nivel [35].

Half-Life 2 (*Valve*, 2004) también usaba uno de los más simples ajustes de la dificultad. Modificaba el contenido de las cajas dependiendo del estado del protagonista al romperlas. Si se encontraba con una salud alta, recibía botiquines pequeños o munición aleatoria. Si la salud era baja, botiquines más grandes aparecían [35]. El lector podrá deducir lo fácilmente explotable que era este sistema, bastaba no coger ningún botiquín en las cajas cercanas para recuperar por completo la salud. A pesar de ello, resultó muy efectivo al ser un sistema tan simple.

El videojuego *Flow* (el cual expondremos como caso de estudio) popularizó la aplicación de inmersión o concentración (el mencionado flow del capítulo 1) en videojuegos. El diseño del videojuego estaba basado en la tésis de Jenova Chen (uno de los propios autores del juego) y se lanzó primero de manera gratuita en *Flash*. Más tarde tuvo una adaptación a *PlayStation 3* [36].

SiN Episodes (*Ritual*, 2006) contaba con el "Sistema de Reto Personal" (*Personal Challenge System*) un sistema parecido al de Max Payne donde el número y resistencia de los enemigos que se enfrentaba el jugador variaban dependiendo de cómo lo estuviese haciendo éste. Así aseguraba un nivel de reto y una progresión del ritmo del juego adecuada [36].

God Hand (*Clover*, 2006) tiene un medidor que regula la inteligencia y fuerza de los enemigos durante la partida. El medidor, dividido en cuatro niveles de dificultad, se incrementa cuando el jugador esquiva y ataca a los oponentes, y decrece cuando recibe golpes. El juego también cuenta con tres niveles de dificultad que actúan como límite del medidor. Por ejemplo, en el nivel fácil, el medidor no pasa de su segundo nivel; es sólo en el nivel difícil cuando el medidor puede llegar a su nivel máximo. Este sistema también ofrece recompensas mejores cuando derrotas a los enemigos en dicho nivel límite.

Left 4 Dead (*Valve*, 2008) destacó por usar una IA nueva a la que llamaban "Director de IA". Éste generaba proceduralmente diferentes experiencias cada vez que alguien jugaba, monitorizando la actuación de los integrantes de la partida de manera individual y colectiva. A partir de ellas determinaba el número de zombis que atacaban a los jugadores y la localización de ciertos enemigos especiales. Además el director iba más allá de un sistema de ajuste de dificultad (y ritmo de juego), también controlaba ciertos aspectos visuales y sonoros

para hacer (o al menos hacer más propensos) que los jugadores estuviesen en un estado de ánimo determinado (encuentro con un jefe de nivel o llamar la atención sobre un área determinada). La propia Valve llamaba a esta manera de generar situaciones "Narrativa Procedural" [16] ya que no sólo ajustaba la dificultad, también según el progreso del jugador generaba ciertos eventos que proporcionaban cierto sentido narrativo [7].

Su segunda parte *Left 4 Dead 2* (*Valve*, 2009) usó el mismo sistema pero con algunas mediciones extra que lo hacían más sofisticado. Algunas de éstas eran la modificación del escenario de acuerdo con el progreso de los jugadores o proporcionar mejor equipamiento al ir por caminos más difíciles [36].

En *Resident Evil 5* (*Capcom*, 2009) se empleaba un sistema llamado "escalado de la dificultad", el cual puntuaba a los jugadores de uno a diez, y ajustaba el comportamiento y ataques de los enemigos, así como su daño y resistencia, basándose en la actuación del jugador (número de muertes, ataques críticos, etc.). El nivel de dificultad elegido restringía al jugador a determinados números (algo parecido a lo que ocurría en God Hand), por ejemplo en el nivel de dificultad normal, sólo se permitía estar entre un dos (bajaba si lo hacías mal) como mínimo y un siete (subía si lo hacías bien) como máximo [36].

En *Fishdom* (*Playrix Entertainment*, 2008), un juego de puzzles de conectar tres piezas de un mismo color. El tiempo límite se ajustaba a lo bien que lo hacía el jugador. El tiempo límite se veía incrementado cada vez que el usuario fallaba un nivel, haciendo posible que casi cualquier jugador pudiese pasárselo tras una serie de intentos [36].

Homeworld y *Homeworld 2* (*Relic*) el número de naves con el que la IA comenzaba cada misión dependía del número con las que terminó el jugador las misiones anteriores. De esta forma, jugadores "buenos" que terminaban las misiones con un gran número de naves en su flota, se les proporcionaba un reto mayor a medida que progresaban en el juego [36].

En Fallout 3 (Bethesda, 2008) y Fallout: New Vegas (Obsidian, 2010), se presentaba un sistema parecido al de Oblivion donde a medida que el jugador subía de nivel, variantes más poderosas de los enemigos con mejores estadísticas y armas, o directamente nuevos enemigos aparecían para mantener una dificultad constante, que era ajustable mediante una barra deslizadora. En el caso de Fallout 3, usar el deslizador para aumentar la dificultad conllevaba unos mayores premios [36].

La saga *Mario Kart* (*Nintendo*) se caracteriza por proporcionar ítems que

[16] `http://www.edge-online.com/features/gabe-newell-writes-edge/`

ayudan a los corredores durante las carreras. Dichos ítems son distribuidos basándose en la posición que ocupa el corredor. Conductores que estén en la cola de la carrera tienen más posibilidades de obtener objetos que aumenten la velocidad o disminuyan la de los demás, mientras que los que estén en la cabeza de carrera rara vez obtendrán ayudas de este tipo, se le proporcionarán ítems mucho peores. Además de esto, la saga usa el ajuste de dificultad ya mencionado anteriormente, conocido como la goma [36].

3. Tipos de DDA

Vamos a presentar, las diversas formas en las que se puede presentar un ajuste dinámico de la dificultad [9]. Existen varios tipos de implementación, desde una simple fórmula hasta mecanismos de aprendizaje que eligen el comportamiento o comportamientos (ya sea del juego o de los NPCs) más adecuados. Cabe mencionar que cada una de las formas es un problema ortogonal, es decir, no son excluyentes y se pueden presentar varias en el mismo juego.

3.1. DDA mediante generación automática del nivel

Se utiliza en juegos donde el jugador controla un avatar y uno de los objetivos principales es ir desde un punto A a otro B, realizando una serie de acciones. En el caso de los juegos de plataformas como *Super Mario Bros* (*Nintendo*, 1985), serían saltar los barrancos, esquivar enemigos, coleccionar monedas etc. En [37], los autores han desarrollado un sistema que genera contenido de manera automática para juegos de plataformas usando como base el juego Infinite Mario (Markus Pearson), un clon de Super Mario Bros de dominio público y que usa una tecnología de *Generación de Contenido Procedural* (*Procedural Content Generation*, PCG). El PCG permite crear contenido de un juego, tales como objetos gráficos u elementos de juego, sin la necesidad de tener a un diseñador que los sitúe de manera manual.

Tradicionalmente el PCG se ha usado offline, es decir, antes de que salga el juego o el jugador empiece a jugar. Una de sus ventajas es que reduce el tamaño del juego considerablemente. Por ejemplo *The Sentinel* (Geoff Crammond, 1986) contenía 10000 niveles generados de forma procedural en 48 KB [17]. Actualmente suele usarse para generar contenido simplemente para rebajar costes y tiempos al equipo de desarrollo. Una herramienta usada para esto sería *SpreadTree*, un

[17] `http://en.wikipedia.org/wiki/The_Sentinel_(video_game)#Limitations_of_the_game_engine`

middleware que genera un gran número de árboles proceduralmente[18] usado en multitud de juegos AAA y películas[19].

No obstante con las menores restricciones de memoria que existen en la actualidad el PCG se está usando online para generar contenido, tales como mapas, en tiempo real a medida que el juego es jugado. En el ya mencionado Infinite Mario Bros, los niveles son creados automáticamente por el jugador mientras juega, dando la posibilidad de un juego infinito que cambia cada vez que lo juegas. En [28] los autores modifican Infinite Mario Bros para proceduralmente adaptar los niveles al jugador partiendo de los datos de 327 personas que jugaban. Los datos se dividían en las siguientes categorías:

1. Aspectos controlables del juego, como el número y anchura media de los barrancos.

2. Estadísticas del juego, como el número de saltos, intentos, enemigos matados que realiza el jugador.

3. La experiencia subjetiva del jugador en de forma numérica teniendo en cuenta diversión, reto y frustración.

Con estos datos, los aspectos controlables que afectan a la percepción del usuario eran determinados mediante un perceptrón. Tras esto se entrenaba a un perceptrón multicapa para que dado un conjunto de características del nivel (número de barrancos) y de juego (número de saltos, de enemigos matados etc), aprendiese que experiencia podría tener el jugador. Así que una vez entrenado, a partir de los datos del nivel, el perceptrón multicapa podría dar información sobre como de divertido, desafiante y frustrante iba a ser el nivel para el jugador. La adaptación tenía lugar de la siguiente forma: el primer nivel que juega el usuario es generado de manera aleatoria; tras esto, las características de juego reunidas de esa sesión de juego junto con las controladas del nivel, se usan como entrada de la red neuronal (perceptrón multicapa). Con estos datos la red decide, dado el último nivel jugado y la forma en la que el usuario jugó, que clase de nivel le resultará más divertido y lo genera, repitiendo este proceso continuamente. De modo que mientras más niveles juegue el usuario, mejor entrenado estará el perceptrón multicapa y mejor determinará los niveles que le resulten más divertidos al jugador.

[18] `http://en.wikipedia.org/wiki/Procedural_generation#Middleware`
[19] `http://www.speedtree.com/video-game-development.php`

3.1.1. Críticas

El citado trabajo [28] presenta un importante avance en la generación de niveles de manera procedural para juegos de plataformas, ya que podría ser portado de forma relativamente fácil a otros juegos aparte de Infinite Mario (actualmente y debido a burocracia legal, el trabajo ahora se hace llamar *Infinite Tux*). Basta con identificar las partes controlables del nivel, y determinar mediante testing que características contribuyen más a la experiencia del usuario. No obstante quizás le falta más evaluación por parte de usuarios "humanos". No solo eso, la estructura de un nivel juega un papel muy importante en la experiencia global del usuario en juegos de plataformas, pero ¿cómo afectaría esto a un juego de acción? Esto podría ser un posible foco de investigación futura.

A pesar de que a fecha de la memoria existen multitud de juegos con contenido generado proceduralmente (como Minecraft por ejemplo, o cualquier juego roguelike) [20], no es usado para generar niveles conforme a la pericia del jugador, sí para modificarlo (adaptación del contenido) algo que entraría en una categoría que veremos más adelante.

3.2. DDA mediante la modificación de la IA

En muchos juegos, el nivel de dificultad que el usuario percibe está directamente influenciado por la manera en que su oponente juega. Esto es especialmente cierto en juegos de estrategia, donde el usuario normalmente se enfrenta a un agente, cuyo único objetivo es avanzar derrotando a toda oposición. Ejemplos de estos casos, son las populares sagas *Civilization* (*Firaxis*) y *Command & Conquer* (*Electronic Arts*).

En cualquier juego, la IA funciona de acuerdo con un algoritmo predefinido; la "fuerza" de la IA será determinada por la calidad del algoritmo subyacente. La dificultad elegida por el jugador determinará que algoritmo usará la IA en la partida. El problema de ajustar la dificultad de manera dinámica para un oponente controlado por la IA, puede ser formulado como la selección apropiada del algoritmo que usará la IA contra un jugador determinado, online y de acuerdo con las habilidades del jugador. Existen diferentes métodos para ajustar la IA de un juego dinámicamente, los cuales expondremos a continuación.

[20] `http://en.wikipedia.org/wiki/Procedural_generation#Games_with_procedural_levels`

3.2.1. Scripting Dinámico

Basado en el aprendizaje por refuerzo, el scripting dinámico es una técnica de aprendizaje on-line, no supervisado que pretende adaptar los comportamientos de los NPCs. Esta técnica propuesta por Spronck [30] está enfocada a los juegos de rol con combates (CRPG) como *Baldur's Gate* (*Bioware*, 1998), *Neverwinter Nights* (*Bioware*, 2002) o *Dragon Age* (*Bioware*, 2009)

En muchos juegos el control de los NPCs está basado exclusivamente en scripts. Durante la fase de desarrollo éstos son adaptados manualmente para asegurar de que exhiben en comportamiento deseado. Una vez que el juego es lanzado, los scripts y dichos comportamientos asociados permanecen sin cambios a menos que haya una actualización o parche del juego. Conviene aclarar que el scripting en los videojuegos suele referirse a una secuencia de acciones del propio juego en lugar de código en un lenguaje concreto. Para hacernos una mejor idea, sería algo parecido al código escrito por el autor de la memoria[21], o a la herramienta de scripting visual de UDK, *Kismet*.

El scripting dinámico consiste en mantener una serie de bases de reglas para cada tipo de enemigo en el juego. Estas reglas son usadas para crear nuevos scripts que controlan los comportamientos de los oponentes que se generan durante la partida. Las reglas que componen un script se extraen de la base de reglas correspondiente al tipo de oponente (por ejemplo, de la base de reglas de "magos" si el oponente es un mago) (figura 3.6). La probabilidad de que una regla sea seleccionada por un script viene dada por el peso asociado a ésta. Las bases de reglas se adaptan cambiando los pesos dependiendo del éxito o fracaso de las correspondientes reglas en el script. El cambio de los pesos viene determinado por la función *actualizar-pesos*.

El proceso de se ilustra en la figura 3.7 [30]. La base de reglas asociada a cada oponente contiene reglas diseñadas manualmente usando el dominio específico de conocimiento. La base de reglas de un personaje controlado por la IA se divide en categorías, dentro de las cuales están agrupadas dichas reglas (figura 3.8). De modo que podemos seleccionar las reglas más orientadas hacia los objetivos, entendiendo por objetivo una categoría de la base de reglas.

Al comienzo de un encuentro o batalla, un nuevo script es generado para cada oponente, seleccionando un número aleatorio de reglas de su base de reglas asociada. Existe una relación lineal entre la probabilidad de que una regla sea elegida y su peso. El orden en que las reglas son colocadas es de dominio específico del juego. Puede usarse algún tipo de mecanismo de prioridad para

[21] `http://thelastdoor.com/forums/viewtopic.php?f=3&t=27`

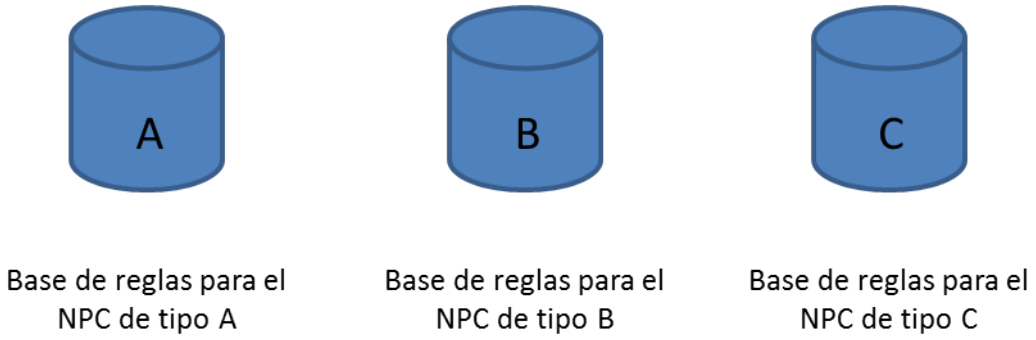


Figura 3.6: Cada tipo de enemigo tendrá asociado una base de reglas de dominio específico del juego y son las que usará para sus scripts.

permitir que ciertas reglas tomen precedencia sobre otras.

En el scripting dinámico el aprendizaje procede de la siguiente forma:

- Tras el enfrentamiento los pesos de las reglas usadas se actualizan dependiendo de su contribución al devenir del combate.
- Las reglas que han conllevado un éxito son premiadas con un aumento en su peso, mientras que aquellas que conllevaron un fracaso son penalizadas con un decremento.
- Las reglas restantes se actualizan de forma que el total de todos los pesos de la base de reglas permanezca igual.

Scripts

Este lenguaje de scripting está diseñado para construir reglas compuestas una sentencia condicional, que es opcional y una acción. La condición está compuesta de una o más condiciones combinadas mediante operadores lógicos. Se pueden referir a diferentes variables, como la distancia al enemigo o la salud del personaje.

Las reglas en el script se ejecutan secuencialmente. Por cada regla, se comprueba su condición (si es que tiene alguna). Si esta se cumple, la acción es ejecutada si es útil en la situación actual (aquí entra en juego esa orientación a objetivos mencionada anteriormente). Si ninguna acción es seleccionada, la acción por defecto de 'pass' es usada.

En el scripting dinámico las reglas se seleccionan con una probabilidad determinada por el peso de éstas. Para establecer un orden, Spronck propone

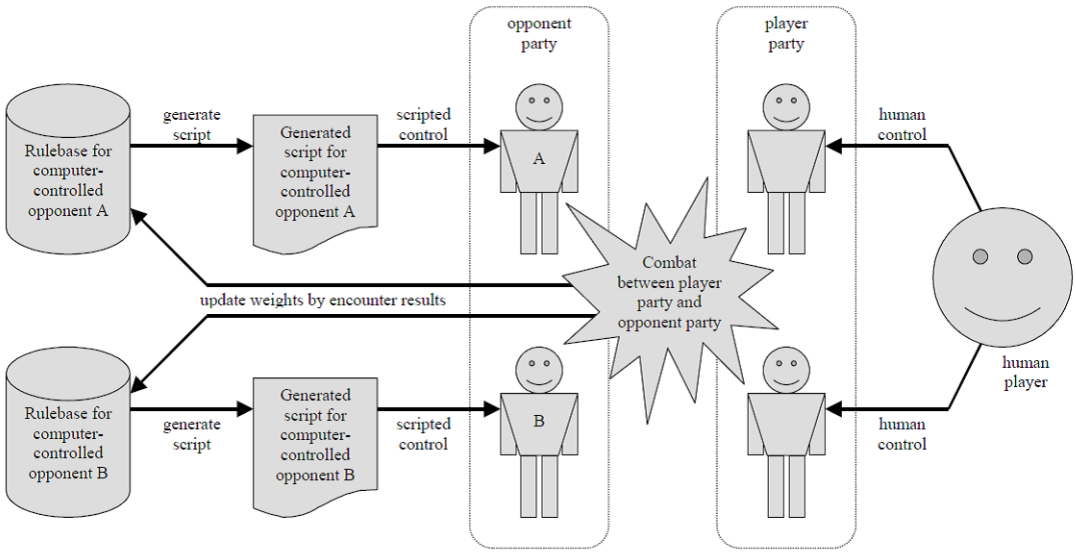


Figura 3.7: Proceso del scripting dinámico en un juego CRPG. Por cada oponente controlado por la máquina se genera un nuevo script al principio del combate. Una vez el combate haya terminado, se adaptan los pesos para reflejar los resultados del enfrentamiento.

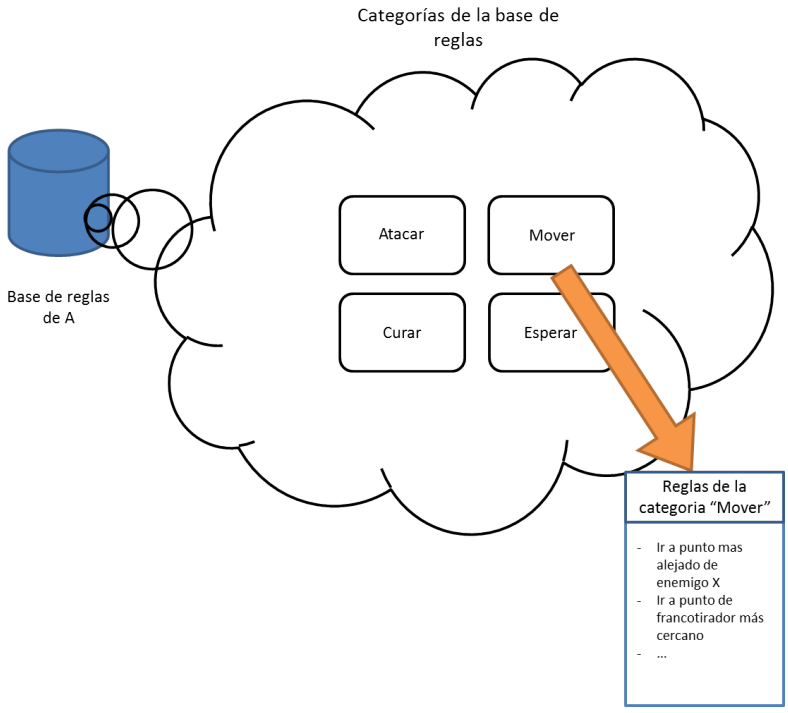


Figura 3.8: Organización de una base de reglas.

asignar a cada regla una prioridad, donde las que tienen un valor más alto tienen precedencia a las que tienen un valor más bajo.

Para las reglas con la misma prioridad, se cogerán antes aquellas que tengan un peso mayor. En el caso de que ambos sean iguales, el orden se determinará de manera aleatoria.

Actualización de pesos

La función de actualización de pesos propuestas por Spronck está basada en dos funciones (ya que está enfocado en un CRPG) de *fitness*: una función de fitness para el grupo, y otra para individuo (si los agentes no combaten a nivel de grupo podríamos obviar la primera). Como ejemplo ilustrativo, mencionaremos éstas de manera más específica a continuación.

La función de fitness del grupo devuelve un valor en el rango [0,1]. Cero si el grupo ha perdido el combate y 0.5 más la media de la salud restante de todos los miembros que componen el grupo que ha ganado. El fitness F de un grupo p (consistente en cuatro miembros) queda definida formalmente de la siguiente forma:

$$F(p) = \begin{cases} 0 & \{\forall n \in p | h(n) \leq 0\} \\ 0{,}5 + 0{,}125 \sum_{n \in p} \frac{h(n)}{mh(n)} & \{\exists n \in p | h(n) > 0\} \end{cases}$$

Donde $mh(n)$ es una función que devuelve la salud del personaje n al principio del encuentro (un número natural mayor que cero) y $h(n)$ devuelve la salud al final del combate del personaje n, $0 \leq h(n) \leq mh(n)$.

La función de fitness de un personaje c devuelve un valor entre [0,1] basada en cuatro factores:

1. La media de la salud restante de todos los miembros del grupo, incluyendo c.
2. La media del daño hecho al grupo oponente.
3. La salud restante del personaje c o si c cayó, el momento.
4. El fitness del grupo.

La función fitness de un personaje c, miembro de un grupo p, queda definida así:

$$F(p,c) = 0{,}05 \sum_n \begin{cases} 0 & \{n \in p \wedge h(n) \leq 0\} \\ 0{,}5 + 0{,}5\frac{h(n)}{mh(n)} & \{n \in p \wedge h(n) > 0\} \\ 1 & \{n \notin p \wedge h(n) \leq 0\} \\ 0{,}5 + 0{,}5\frac{h(n)}{mh(n)} & \{n \notin p \wedge h(n) > 0\} \end{cases} + \begin{cases} \frac{min(dc(c),100)}{1000} & \{h(c) \leq 0\} \\ 0{,}2 + 0{,}1\frac{h(c)}{mh(c)} & \{h(c) > 0\} \end{cases} + 0{,}3F(p)$$

Donde n es cualquiera de los personajes del combate (de un total de ocho), $dc(c)$ es el momento en el que el personaje c cae o muere. Las demás funciones son las mismas que las de la anterior. La función de fitness del individuo, asigna un valor de recompensa alto a éste si su grupo gana, incluso si dicho individuo cae durante el combate. A esta se añade una recompensa más pequeña por la propia supervivencia del individuo, otra un poco más pequeña por la supervivencia de los compañeros, y otra por el daño inflingido a los oponentes.

La función de actualización de pesos traslada el fitness de un personaje en pesos para las reglas del script. Sólo las reglas en el script que se ejecutan durante un encuentro son penalizadas o recompensadas. La función de actualizar pesos se define formalmente así:

$$W = \begin{cases} max\left(0, W_{org} - MP \cdot \frac{b - F(p,c)}{b}\right) & \{F(p,c) < b\} \\ min\left(W_{org} + MR \cdot \frac{F(p,c) - b}{1-b}, MW\right) & \{F(p,c) \geq b\} \end{cases}$$

Donde W es el nuevo peso, W_{org} es el peso original de la regla, MP es la penalización máxima, MR la recompensa máxima, MW el peso máximo y b es el punto de equilibrio. El punto de equilibrio se alcanza cuando no se producen cambios en los pesos. Para mantener constante la suma de todos los pesos de una base de reglas, los cambios en éstos son ejecutados a través de una redistribución de todos los pesos.

Mejoras

Cabe destacar dos optimizaciones propuestas por Spronck [31], para generar unos scripts más variados -una mayor exploración.

Recorte de los pesos: el valor máximo de un peso W_{max} determina el máximo nivel de optimización que puede alcanzar una táctica aprendida. Un valor alto de W_{max} permitiría pesos muy altos que tras un rato conllevaría que las reglas más efectivas (con mayor peso) se seleccionasen siempre. Lo que tendría como resultado scripts más cerca del óptimo. Un valor más bajo de

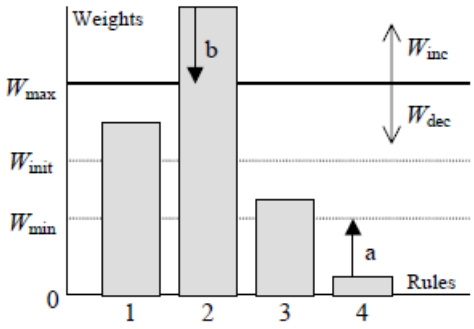


Figura 3.9: Ilustración de los procesos de recorte de pesos y bloqueo de los mejores, junto con sus parámetros.

W_{max} restringiría el crecimiento de los pesos, forzando una mayor diversidad de scripts los cuales no su mayoría no sería óptimos.

El recorte de pesos cambia de manera dinámica el valor de W_{max}, con el fin de forzar un juego más justo. Se trata de tener un valor bajo cuando el ordenador gana de forma frecuente, y un valor alto cuando pierde de manera continuada. La implementación sería así:

Si el ordenador gana un combate, W_{max} es decrementado por un porcentaje W_{dec} pero siempre manteniendo que $W_{init} \leq W_{max}$. Si el ordenador pierde, W_{max} es incrementado por W_{inc} por ciento.

La figura 3.9 ilustra el proceso de recorte de pesos y los parámetros. Las barras sombreadas son los pesos de las reglas las cuales están representadas en el eje horizontal. El peso máximo cambia por valor de W_{dec} o W_{inc} dependiendo del resultado del combate. Después del ajuste de pesos, en la figura 3.9 el peso de la regla 4 es demasiado bajo y se incrementará hasta W_{min} (la flecha 'a'), mientras que la regla 2 es demasiado alta y decrecerá a W_{max} (la flecha 'b').

Bloqueo de los mejores: el bloqueo de los mejores es similar al recorte de pesos. Emplea el mismo mecanismo de adaptación para el peso máximo W_{max}. La diferencia entre esta técnica y la anterior es que esta permite que los pesos tengan un valor mayor de W_{max}. No obstante, las reglas que superen W_{max} no se elegirán para los scripts. En consecuencia, cuando un oponente controlado por el ordenador gana de manera frecuente, las reglas más efectivas excederán W_{max} y no podrán ser seleccionadas, por lo que se comenzarán a usar otras tácticas a priori más débiles. De manera opuesta, cuando pierde continuamente, las reglas con pesos altos vuelven a ser seleccionables y podrán usarse las tácticas más fuertes otra vez.

En la figura 3.9, al contrario que en el recorte de pesos, la regla 2 permanecería igual (la acción representada por la flecha 'b' no se llevará acabo), pero no estaría disponible para ser elegida para la próxima selección puesto que excede W_{max}.

Críticas

Una desventaja de esta técnica es que a medida que el juego avanza en su desarrollo, la base de reglas tiende a ser demasiado grande. Esto provoca que sea más susceptible de errores, más difícil de construir y más difícil de mantener. Además, el adversario está limitado por el diseño de las reglas que generan el agente más inteligente, el cual podría incluso no significar reto alguno para jugadores más avanzados o más habilidosos.

Otra crítica que podríamos hacerle al modelo presentado por Spronck es que está principalmente enfocado al combate en juegos de rol, en cosas como la selección de ataques. Afortunadamente (o desafortunadamente) este tipo de situaciones no son especialmente problemáticas para la aleatoriedad, de hecho podrían ser divertidas. No obstante existen otros problemas en la IA de los juegos mucho más sensibles a la aleatoriedad, como es la predicción de acciones, en los que esta técnica está más limitada.

Además existen ciertas limitacione debido al hecho de analizar el fitness de una estrategia después de un encuentro, ya que los jugadores analizan la situación actual en un combate para adaptar su estrategia. Es decir podría reconocer lo que está intentando hacer el oponente y contratacarla. Si esta evaluación del fitness pudiese evaluarse durante el combate, y que afectasen al peso de las reglas durante este mismo, teniendo en cuenta que es lo que está haciendo el oponente sería una herramienta mucho más flexible y útil[22]. También asume en dichas funciones que la salud al final del combate es inferior a la del inicio ¿y si ejecuta una acción de recuperación (un hechizo de cura)?

Añadir además, que a pesar de que Spronck menciona en su trabajo [30] que BioWare estaba investigando en cómo aplicarlo en sus juegos, no se conoce ningún juego el cual haya aplicado scripting dinámico.

Para finalizar, como opinión personal del autor de esta memoria, podríamos mencionar que el scripting dinámico puede adaptarse como un *selector* de un *Behavior Tree* [23] (o de una *HTN*) por lo que podría tener cabida como apoyo a otras técnicas.

[22] `http://aigamedev.com/open/review/online-adaptation-game-opponent/`
[23] `http://aigamedev.com/open/review/online-adaptation-game-opponent/`

3.2.2. Coevolución online y algoritmos genéticos

Se trata de tener una serie de agentes predefinidos, ya sea offline o de manera manual, con buenas características y usar algoritmos genéticos usando como padres estos agentes para que guíen la evolución. De esta forma evolucionamos aquellos que se encuentren más cerca de las habilidades del jugador [9].

Críticas

A pesar de ser un enfoque interesante tiene una serie de limitaciones. Para jugadores muy habilidosos es posible que no exista ningún agente predefinido, o para aquellos que tenga un comportamiento muy poco común jugando, el proceso de aprendizaje podría ser muy lento. Además, la habilidad de los agentes sólo puede crecer (no hay posibilidad de regresión). Así que si las habilidades de un jugador decrecen (por ejemplo, el usuario deja de jugar durante un tiempo) los agentes no se adaptarán a dicho cambio. Para asegurar que los agentes cubren todo tipos de niveles de habilidad, éstos deben comenzar su evolución desde el nivel más fácil. Esto implica un mayor tiempo de evolución hasta alcanzar las habilidades de un jugador experimentado, que podría aburrirse durante el proceso.

3.2.3. Agentes adaptativos. MiniMax adaptativo

Consideremos la situación en el que jugador y agente tienen conjuntos similares de acciones, como podría ser el caso de juegos de tablero (como el tres en raya, el backgammon o el conecta cuatro) o videojuegos "reales" tales como un juego de estrategia en tiempo real (o por turnos).

Dado un estado de juego algunas acciones son mejores que otras, en otras palabras, hay una clasificación natural de las acciones disponibles. Conociendo dicha clasificación el agente puede evaluar la actuación del jugador y elegir sus acciones de acuerdo con éstas. Este enfoque nos permite crear agentes adaptativos (online) que se adapten a su oponente durante el transcurso de una partida.

En [22] se investiga dicha técnica de clasificación para desarrollar agentes adaptativos para el juego Conecta Cuatro o Cuatro en raya. Este juego se escogió debido a que esta técnica requiere dominio específico por lo que era necesario seleccionar un juego que fuese familiar para muchas personas, que se pudiesen jugar varias partidas, que hubiesen suficientes tácticas y estrategias y que no fuese muy complejo desde el punto de vista teórico pero sí más que el

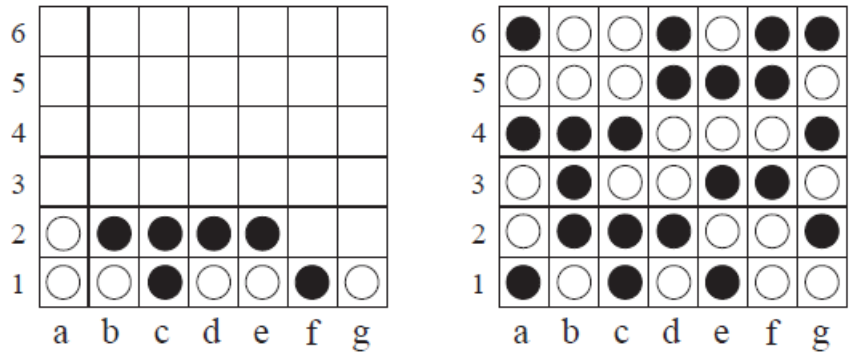


Figura 3.10: Dos posibles estados de una partida de conecta cuatro. La izquierda muestra una situación donde las negras han ganado. La derecha un empate entre ambos jugadores.

tres en raya, dejando más lugar para la adaptación.

Entorno

Conecta Cuatro es un juego para dos jugadores cada uno con 21 fichas iguales de un color determinado. Asumiremos que un jugador tiene sus fichas blancas, el otro negras y que las blancas mueven primero.

El tablero es de 6x7, si dejamos caer una ficha por una de las columnas, caerá hasta la casilla desocupada más baja. Cada movimiento consiste en poner una ficha en una de las columnas que no esté completa.

Cada jugador hace sus movimientos en turnos y el objetivo es conectar cuatro fichas en vertical, horizontal o diagonal. Si se usan las 42 fichas y no se ha creado ninguna conexión de cuatro, se considera empate. La figura 3.10 muestra dos situaciones del juego, la de la izquierda donde el jugador negras gana, la otra donde se da una de las posibles situaciones de empate.

Cabe destacar, que existe un algoritmo óptimo para este juego desarrollado en 1988 por Victor Allis [4]. En dicho algoritmo las blancas siempre ganan y las negras son muy difíciles de vencer.

MiniMax

En términos de teoría de juegos, un juego consiste en un conjunto de jugadores, un conjunto de movimientos o estrategias disponibles para éstos y una especificación de pagos o recompensas (normalmente una matriz de pagos) que

tiene en cuenta cada posible combinación de estrategias.

En Conecta Cuatro el conjunto de jugadores son dos, blancas y negras, el conjunto de estrategias o movimientos está formado por todas las secuencias de movimientos legales.

Definimos los pagos de la siguiente manera:

- En caso en que las blancas ganen, su recompensa o pago es +1, -1 para las negras.
- En caso de empate, la recompensa es 0 para ambos.
- En caso de que las negras ganen, su recompensa es +1 y -1 para las blancas

Viendo la definición de recompensas, el Conecta Cuatro se convierte en un juego por turnos para dos jugadores de suma cero: La suma total de las ganancias y las pérdidas es cero para cualquier final del juego. Este tipo de juegos se pueden resolver aplicando un algoritmo MiniMax.

El algoritmo MiniMax usa un árbol de juego para hacer sus decisiones sobre el movimiento a realizar. Un árbol de juego es un árbol dirigido cuyos nodos son estados del juego y cuyas aristas son transiciones legales entre estos, dicho de otra forma, los movimientos. El árbol de juego completo es aquel que empieza desde el inicio del juego y que contiene todos los movimientos posibles. Con esto podríamos "resolver" el juego, es decir, encontrar una secuencia de movimientos óptima para ambos jugadores que garantice la victoria a uno de ellos o el empate.

Normalmente, en la práctica, el MinMax sólo utiliza un subárbol para tomar sus decisiones. Este subárbol es construido a partir del árbol completo dónde su raíz es el nodo que representa la situación actual del juego, y que continúa explorando hasta un nivel de profundidad fijado, por lo que ya no tenemos información completa. Mientras que las hojas originales del árbol de juego completo poseían su valor exacto, las hojas de este subárbol deben ser evaluadas usando algún tipo de heurística.

La heurística usada en [22] calcula una suma ponderada que tiene en cuenta las líneas abiertas y cerradas y los *forks* dados por un movimiento. Cuatro casillas en el tablero en cualquier dirección tales que 3 o menos sean de un jugador y las demás estén vacías, es una línea abierta. El jugador que tiene una línea abierta produce una amenaza para su oponente. Si el oponente coloca una pieza rompiendo la amenaza, se le llama línea cerrada. La figura 3.11 muestra un ejemplo de línea abierta.

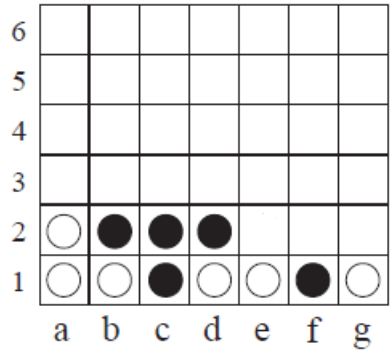


Figura 3.11: Ejemplo de línea abierta (fila 2, columnas b, c, d, e) por parte de negras.

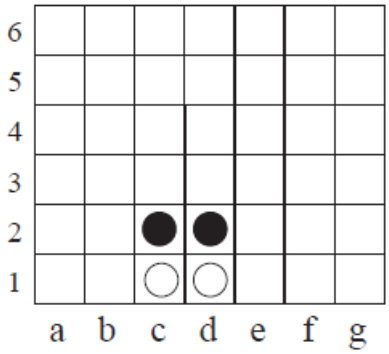


Figura 3.12: Ejemplo de fork. Si negras no cortan la amenaza, las blancas podrían conseguir un 3-fork (fila 1, b o e) que en este caso le darían la victoria.

Un fork es un conjunto de 4 o 5 casillas conectadas de tal modo que:

- Los cuadrados siguientes a sus lados están vacíos.
- Los de un lado son de un color y los de otro del contrario.

La figura 3.12 muestra un fork. Un fork compuesto por dos fichas debe ser cortado tan pronto como sea posible para prevenir que se convierta en un 3-fork. De forma opuesta, para un jugador construir sus propios forks le ofrece una gran ventaja.

El algoritmo opera con tres pesos: w_f, w_{pl}, w_{ol} (el peso de los forks, el peso de las líneas abiertas del jugador y el peso de las líneas abiertas del oponente).

Si asumimos que f_p y l_p son el número de forks y líneas abiertas que el jugador crearía con un movimiento en cuestión, l_o es el número de líneas abiertas que serían rotas al oponente y f_o es el número de forks del oponente que todavía existirían con ese movimiento. Si se hace dicho movimiento, la puntuación quedaría reflejada mediante la siguiente fórmula:

$$s(m) = w_{pl} \cdot l_p + w_f \cdot f_p + w_{ol} \cdot l_o - w_f \cdot f_o$$

Cuando usamos un subárbol el MiniMax ya no podemos estar seguros de que se encuentre el camino óptimo a través de este, ya que se reemplaza la información exacta con valores heurísticos. Si incrementamos la profundidad de la búsqueda mejoraremos el rendimiento del algoritmo, pero también empleará más tiempo para tomar una decisión. La elección de este parámetro representa un balance entre como de bien jugará el MinMax y cuánto tiempo podemos permitirle para que haga un movimiento.

De manera adicional, para mejorar más el rendimiento, podemos reducir el número de nodos visitados usando una poda alfa-beta. La poda alfa-beta termina una evaluación de un movimiento cuando encuentra que su valor es menor que el óptimo actual (el mejor movimiento que ha encontrado hasta el momento). La eficiencia del alfa-beta depende del orden en el que se evalúen los nodos, en condiciones óptimas, cuando el mejor movimiento siempre es considerado el primero, alfa-beta dobla la efectividad de la profundidad de búsqueda del MiniMax.

Algoritmo Adaptativo

La idea que subyace en este tipo de agentes adaptativos es, dado que tanto jugador como agente comparten un conjunto de acciones muy similares, realizar una clasificación de las acciones disponibles junto con las ya realizadas a lo largo de la partida para que el agente estime como está jugando el usuario. Para realizar esta clasificación se usa un algoritmo MiniMax por lo que en [22] deciden llamar a esta técnica MiniMax Adaptativo (*Adaptative MiniMax*, *AMM*).

Supongamos que el algoritmo MiniMax usase el árbol completo de juego dónde sus hojas tendrían asignadas uno de los siguientes valores: $\{+\infty, 0, -\infty\}$. Tras evaluar el árbol el MiniMax construye el conjunto de movimientos disponibles cuyas evaluaciones son uno de los elementos del conjunto $\{+\infty, 0, -\infty\}$. Estas evaluaciones crean la siguiente clasificación:

1. Movimientos ganadores, $+\infty$

2. Movimientos neutrales, 0

3. Movimientos perdedores, $-\infty$

Desafortunadamente esta clasificación es demasiado extremista. Teniendo en cuenta que las acciones neutrales disponibles apenas aparecen, prácticamente nos quedan sólo acciones ganadoras y perdedoras, por lo que necesitamos modificar dicha clasificación.

Para conseguir una clasificación más detallada necesitamos reflejar la longitud del camino desde la raíz hasta la hoja, basándonos en la idea de que un camino más corto que nos lleve a ganar la partida es mejor que uno largo. De manera opuesta, un camino que alargue nuestra derrota es mejor que uno que nos la haga perder inmediatamente.

Para implementar esta reemplazamos los $+\infty$ y $-\infty$ por números lo suficientemente grandes $+L$ y $-L$, es decir, valores que nos permitan modificar las puntuaciones de ganadores/perdedores sin que por ello dejemos de diferenciar fácilmente entre éstos. Los números positivos deben indicar caminos ganadores para el jugador MAX, y los negativos, caminos ganadores para el jugador MIN.

Para incluir información sobre la longitud de la victoria (o derrota) de un camino del jugador MAX restamos 1 al valor previo, añadimos 1 al valor anterior por cada nivel de un camino que lleve a la derrota al jugador MAX (o lo que es lo mismo, la victoria del jugador MIN). De modo que un movimiento con una evaluación de $L + 5$ significa que existe un camino ganador para MAX en cinco movimientos, empezando desde el estado actual. Mientras que $L + 1$ significaría que MAX puede ganar en dos movimientos si eligiese el movimiento actual. Además refinamos la graduación de los movimientos disponibles usando la heurística del MinMax descrita anteriormente.

Este método de clasificación propuesto sirve para determinar las habilidades de un oponente online (entiéndase online en términos de aprendizaje). Antes de evaluar los movimientos disponibles y tomar una decisión el AMM evalúa los movimientos que tenía disponibles su oponente y establece una clasificación aplicando los mismo principios. El movimiento que realizó recibe su puntuación, es decir, su posición de la clasificación. Se calcula la media de la puntuación de todos los movimientos hasta el momento, usándola el agente, para elegir la acción que esté más cerca de dicha puntuación.

Críticas

Este modelo sólo parece encajar en los juegos por turnos, resultando muy

difícil llevarlo a otro tipo de juegos. La principal desventaja de este enfoque, es que el agente sólo podrá jugar como mucho al nivel del MinMax, el cual no es óptimo (la profundidad del árbol es limitada), además obtener el óptimo no es tan fácil en un juego (por ejemplo) de estrategia en tiempo real. Lo que conlleva que el AMM no pueda ser un buen contrincante ante alguien que tenga más habilidad que el árbol.

Otra desventaja es que el AMM requiere una heurística que evalúe los movimientos o estrategias en el juego. Sin ésta el agente podría no construir una clasificación correcta.

Es necesario encontrar una manera de aumentar la profundidad del árbol de juego que se explora.

3.2.4. Selección de acciones dependiendo del reto. Q-Learning

El aprendizaje por refuerzo propuesto por [5] trata de adaptar el comportamiento de los agentes, primero a partir de un aprendizaje offline y luego mediante un aprendizaje online donde el agente elige las acciones a realizar dependiendo de la pericia del jugador. Es necesario por tanto, medir de manera implícita o explícita la dificultad que está afrontando un jugador en un determinado momento. Como se ha mencionado otras veces anteriormente, esta medida se lleva a cabo mediante una heurística –llamada comúnmente función de reto. Esta función mapea un estado o especificación de juego en un momento dado con un valor de dificultad que nos dice como de fácil o difícil le está siendo al usuario el juego. Ejemplos de algunas heurísticas usadas son: el ratio de disparos o golpes acertados, el número de victorias o derrotas, la evolución de los puntos de salud o el tiempo en completar lo que lleva de juego.

Aprendiendo a jugar al nivel del usuario

El problema de cambiar el nivel de dificultad de un juego dinámicamente puede ser abordado mediante aprendizaje por refuerzo, eligiendo la recompensa de manera podemos hacer que el agente aprenda a actuar al mismo nivel que le dictan las habilidades del usuario. Cuando el juego se hace demasiado fácil o difícil se da una recompensa negativa (penalización), en otros caso, positiva [5].

Este enfoque tiene la ventaja de que el modelo matemático de aprendizaje corresponde con los objetivos del agente. No obstante también tiene inconvenientes: el agente podría no ser capaz de rivalizar con un jugador experto ya que tendría que aprender antes (es por eso que primero se realiza un aprendi-

zaje offline), o podría desembocar en comportamientos no creíbles con tal de mantener la partida equilibrada [5].

Selección de acciones dependiendo del reto

Dadas las dificultades anteriores, en [5] presentan la siguiente alternativa. Separar el ajuste de la dificultad en dos problemas: el aprendizaje (construir agentes que puedan aprender estrategias óptimas) y la adaptación (proporcionar mecanismos de selección de acciones para equilibrar el juego, que posiblemente implique usar acciones sub-óptimas).

Debido a que los agentes deben de poder jugar a cualquier nivel dificultad de manera inmediata desde el comienzo del juego. Se necesita de un entrenamiento offline para arrancar el proceso de aprendizaje. Esto se puede conseguir haciendo que el agente juegue contra él mismo (autoaprendizaje) o con otros pre-programados. Luego, podemos aplicar un aprendizaje online para adaptar continuamente estos comportamientos iniciales a un jugador específico para descubrir la mejor estrategia (entiéndase aquí "mejor", como la estrategia que mejor se adapte a las habilidades del jugador) para jugar contra él o ella.

En cuanto a la adaptación, la idea es encontrar la política de acciones adecuada que proporcione un juego bien equilibrado, por ejemplo, acciones que mantengan al agente y al jugador aproximadamente al mismo nivel. Una vez que el agente ha aprendido la política óptima, puede cambiar durante ciertos momentos (con una frecuencia que depende de la función de reto) simplemente eligiendo una acción aleatoria. El problema de este enfoque es que fácilmente puede generar comportamientos poco creíbles. Por ejemplo en el juego de lucha sobre el que usan este sistema [5] desemboca en un puñetazo al aire cuando el oponente está muy lejos, o un salto de huida hacia atrás cuando no hay peligro.

En la implementación de [5], de acuerdo con la dificultad a la que el jugador se esté enfrentando, el agente elige las acciones con un valor esperado alto o bajo. Por ejemplo, dada una situación, si el nivel del juego es demasiado alto, el agente no elige la acción óptima (aquella que tiene el valor más alto) sino que elige progresivamente las sub-óptimas hasta que su nivel se equilibra con el del jugador. Esto implica ir cogiendo la segunda, tercera o cuarta (y así sucesivamente) mejor acción hasta alcanzar el nivel del jugador. De manera opuesta, si el nivel del juego empieza a ser demasiado fácil, comenzará a coger acciones cuya valor sea más alto, pudiendo incluso llegar a usar la política óptima. En la figura 3.13 se muestra una posible configuración del agente el cual actúa cogiendo la segunda mejor acción. La relación de orden queda definida en un estado cualquiera por la función acción-valor, que es construida durante la fase de aprendizaje. A medida que estos valores estiman la calidad de cada

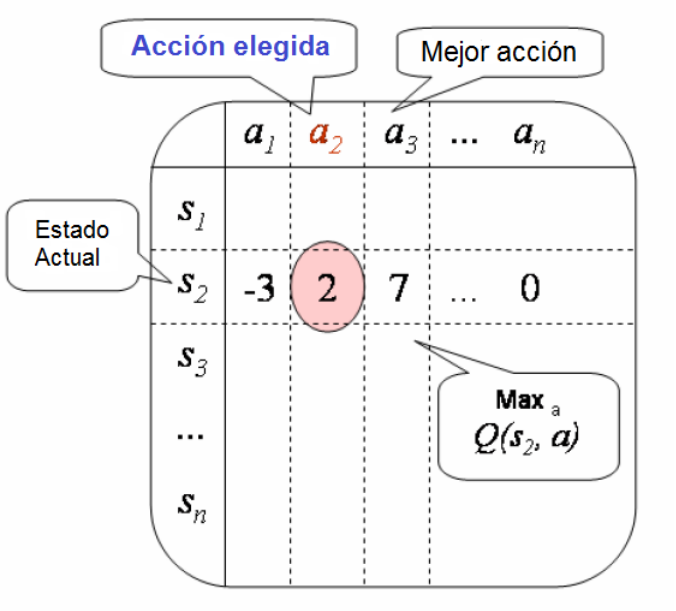


Figura 3.13: Una selección de la segunda mejor de acción por parte del agente

acción de manera individual, es posible tener un control mejor y más rápido sobre el comportamiento de los agentes, y por tanto en el nivel del juego.

Una posible fuente de inestabilidad para este mecanismo de selección de acciones es que los valores que devuelve la función acción-valor expresa la recompensa o valor esperado por la acción actual y sus siguientes y que se va reduciendo conforme al parámetro γ. De modo que si el agente quiere obtener una recompensa a largo plazo, necesitaría continuar la misma política hasta obtener el premio deseado. Lo que significa que es necesario determinar un ciclo que determine cuando el agente debe analizar, y posiblemente cambiar, su forma o nivel de actuar. De hecho, este ciclo debería ser el mismo que el usado por la heurística de reto para medir la dificultad a la que se está enfrentando el jugador.

De modo que en esta técnica el agente evalúa periódicamente si está al mismo nivel del jugador, a través de la función de reto, y de acuerdo con el resultado para que mantenga o cambie su nivel. El agente no cambia su nivel hasta el siguiente ciclo. Este ciclo de evaluación es dependiente del juego en el que actúa el agente, y de las recompensas a largo plazo. Si el ciclo es demasiado corto, el agente puede comenzar a realizar acciones aleatorias; si por el contrario es demasiado largo, tardará demasiado en ajustarse a la habilidad del jugador.

Es importante resaltar que esta técnica sólo cambia el mecanismo de selección de acciones, el agente puede seguir aprendiendo durante todo el juego.

Destacar también que si el mecanismo de selección se apoya sobre un aprendizaje offline previo, puede conseguir adaptarse a mayor velocidad, incluso pudiendo hacer frente desde el principio del juego a un jugador experimentado.

Selección de acciones dependiendo del reto aplicado a un juego de lucha

En [5] como caso de estudio aplican el mecanismo de selección de acciones a un juego de lucha donde dos jugadores compiten en tiempo real dentro de un escenario. El objetivo principal es derrotar al contrario. La pelea termina cuando los puntos de vida de uno de los jugadores llegue a cero (se empieza inicialmente con 100), o después de 1 minuto y 30 segundos (y gana el que más vida conserve), lo que se produzca antes. La zona de combate es un área bidimensional en donde los jugadores pueden moverse libremente en horizontal, y en vertical mediante saltos. Las posibles acciones son dar un puñetazo (fuerte o rápido), dar una patada (fuerte o rápida) y lanzar un ataque especial (una "bola de fuego"). Los puñetazos y patadas también pueden darse en el aire, durante un salto. Las acciones defensivas son bloquear y agacharse, durante esta última también es posible lanzar puñetazos o patadas. Si el agente tiene suficiente maná, puede lanzar el ataque especial. El maná se reduce al realizar dichos ataque especiales y se va rellenando continuamente a una velocidad fija.

Aprendiendo a jugar. La IA del agente se implementa mediante una técnica de aprendizaje por refuerzo (en [5] usando concretamente Q-learning) y por tanto es necesario codificar las percepciones de los agentes, las posibles acciones y la función de recompensa.

La representación del estado queda definida por la siguiente tupla:

$$S = (S_{agent}, S_{opponent}, D, M_{agent}, M_{opponent}, F)$$

S_{agent} es el estado del agente (parado, saltando o agachado). $S_{opponent}$ es el estado del oponente (parada, saltando, agachado, atacando, atancando en el aire, atacando agachado, cubriéndose). D es la distancia al oponente (cerca, medio y lejos). M representa el nivel de maná (necesario para realizar ataques especiales) y F la distancia de la bola de fuego enemiga (cerca, medio, lejos, no existe). Las acciones del agente son las mismas que las de los demás jugadores, descritas en el punto anterior.

La señal refuerzo está basada en la diferencia de vida causada por una acción (la vida quitada al oponente, menos la vida perdida por el agente). Como

resultado, la recompensa siempre está en el intervalo [-100,100]. Recompensas negativas significan una mala elección porque el agente perdería más vida que el daño que inflige. Mientras que recompensas positivas, que el adversario pierda más vida que él en una acción, son el objetivo de aprendizaje del agente.

La inteligencia inicial del agente puede construirse a partir de enfrentamientos con una IA con movimientos aleatorios o contra el mismo (auto-aprendizaje). En [5] usan el agente aleatorio como punto de partida.

Jugando al nivel de usuario. Este mecanismo de selección de acciones propuesto por [5] fue evaluado en un juego de lucha concreto (aunque puede extenderse fácilmente a otros del mismo género). La función de reto está basada en la diferencia de vida durante la pelea. Para motivar al jugador, la función está diseñada para que el agente trate de jugar mejor que él, quedando definida empíricamente así: si la vida del agente es menor que la vida del jugador, es que el nivel está siendo fácil; si su diferencia de vida es menor que el 10 % del total de éstas (en el juego el total de vida es 100), el nivel es medio; en otro caso es difícil.

$$f = \begin{cases} fácil, si & L(agent) < L(player) \\ medio, si & L(agent) - L(player) < 10 \\ dificil, en\ otro\ caso & \end{cases}$$

El funcionamiento es el siguiente. El agente empieza en el nivel medio, para el juego en el que se probó la selección de acciones en [5], éste tenía trece posibles acciones por lo que se establecieron trece niveles de dificultad, siendo el nivel seis el de comienzo del agente. Por lo tanto al principio, elegirá la sexta mejor acción dado por el valor devuelto por la función acción-valor. Si el juego está siendo fácil, el agente pasa a elegir la séptima mejor acción; si es difícil, subirá de nivel al quinto; en otro caso se mantiene en el nivel seis.

Críticas

Las técnicas de aprendizaje por refuerzo son bastante complejas y usar esta misma aproximación de máquina finita de estados en un juego más complejo puede implicar un consumo de memoria prohibitivo. Tendríamos que usar técnicas más sofisticadas para aprendizaje por refuerzo en entornos contínuos [27]. Aparte de esto, la mayor crítica que podemos hacer a este enfoque es sobre los posibles problemas de inestabilidad que pueda tener. Habría que encontrar los valores adecuados para optar por recompensas a largo plazo (ese "retraso"). Lo que además conlleva una selección de la frecuencia del ciclo de evaluación (ya

que pasado un tiempo esa opción a largo plazo puede ser demasiado difícil para el jugador).

3.3. DDA mediante el ajuste del contenido del nivel

Son los DDAs que ajustan la dificultad basándose en la cantidad y tipos de elementos (contenido de nivel) a aparecer durante la partida.

3.3.1. Secuenciador de Pfeifer

El secuenciador propuesto por Borut Pfeifer busca controlar el ritmo de juego y la tensión dramática mediante la dificultad. Trata de adaptar de alguna forma el trabajo hecho en Façade sobre controlar la estructura dramática en una narrativa interactiva, ya que mantiene el mismo objetivo básico: usar un sistema que controle qué eventos son creados en el entorno para ajustar la experiencia del jugador.

Este sistema actúa como un secuenciador, mezclando diferentes eventos para crear un ritmo de juego adecuado al jugador, que mantiene la estructura dramática (momentos más difíciles por ejemplo) y ajustando los retos a las habilidades del jugador.

Midiendo las habilidades del jugador, seríamos capaces de determinar en qué zona de *flow* (ver capítulo 1) podría estar. Pudiendo aumentar el reto con respecto a las mejoras de sus habilidades o también ajustar éste en base a la tensión dramática. Incluso sería posible eliminar la necesidad de implementar distintos niveles de dificultad estática.

Tensión dramática

La tensión dramática viene acuñada de [17] donde se usa el drama como un modelo para la interactividad, en lugar de la narrativa:

- Promulgación. El modelo interactivo se centra en las acciones, las cuales también definen la interactividad. A diferencia de la narrativa, que se centra en las descripciones.
- Intensificación. Los incidentes o eventos se pueden seleccionar y organizar en un espacio de tiempo. La narrativa suele usar la “extensificación” (se

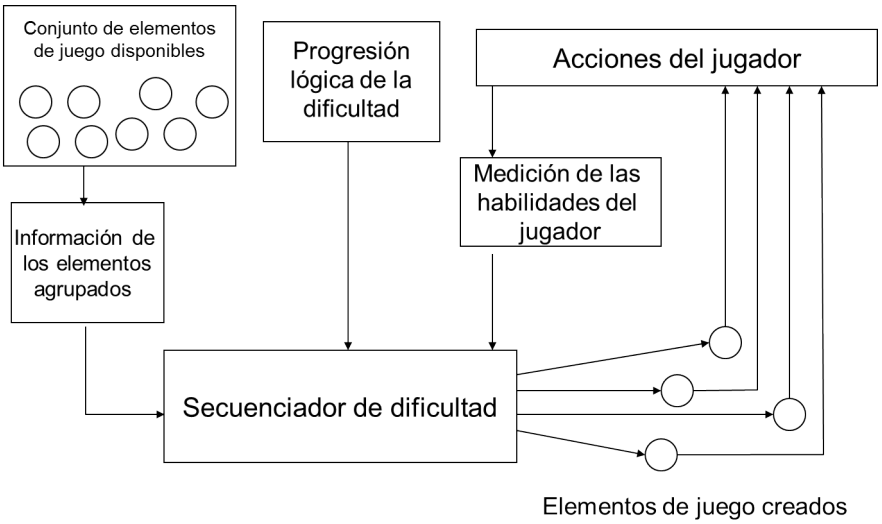


Figura 3.14: Diagrama a alto nivel que describe el funcionamiento del secuenciador y los componentes que intervienen

extiende más en el tiempo), donde una escena puede ser visitada varias veces desde la perspectiva de diversos personajes.

- Unidad de acción. Acciones enlazadas de manera causal con una acción central principal. La narrativa suele usar para esto los capítulos

Componentes del Sistema

La figura 3.3.1 muestra un diagrama a alto nivel de los componentes de este secuenciador que comentaremos a continuación:

- Elementos dinámicos de juego: se trata de objetos del juego que pueden cambiar de un momento a otro y que afectan a la dificultad.
- Progresión lógica de la dificultad: normalmente esto se da tras una etapa de balanceo de juego, determina donde están los puntos fáciles y difíciles de éste.
- Medición de las habilidades del jugador: se trata de una heurística o función de reto que nos puntúa como lo está haciendo el jugador hasta el momento.
- Información de los elementos agrupados: puede haber un componente estético o tener que cumplir una restricción la selección de elementos de juego dinámico.

- Secuenciador de dificultad: selecciona los elementos necesarios para lograr la dificultad deseada con respecto a la información de éstos, la progresión de la dificultad y las habilidades del jugador.

Elementos dinámicos de dificultad. Un ejemplo quizás defina mejor lo que son los elementos dinámicos del juego: los tipos de enemigos o, en un juego de acción, los tipos de arma o botiquines. Los cuales evaluamos o puntuamos su dificultad de alguna forma. Éstos a su vez puede estar compuesto de otros subelementos (un enemigo determinado con un arma o armadura determinada) que necesiten ser evaluados también.

Esta evaluación suele estar establecida por el diseñador que asigna una puntuación de dificultad a cada elemento, más fácil de ajustar para las heurísticas frente una variedad de factores complejos. También tiene definida una escala de dificultad que se corresponde con la medición de las habilidades del jugador (por ejemplo, 1 la más fácil, 10 la más difícil).

Definir mediante una fórmula más "mecánica" este cálculo normalmente es problemático (recordar el apartado de este mismo capítulo dedicado a pifias). Una heurística hecha por un diseñador abarcará varios factores potenciales (daño, rango, precisión, estrategias de IA usadas) sin añadir una excesiva complejidad al sistema. Si la dificultad asociada con los elementos individuales cambia frecuentemente, si es posible que requiramos algo más sofisticado y automático para definir la dificultad de los elementos, lo que conllevará que sea mucho más difícil de ajustar. Una situación donde puede ocurrir esto (de nuevo orientado a un juego de acción) es cuando los enemigos tienen acceso a modificadores que afectan de manera muy pronunciada la fuerza de éstos con respecto al jugador (como coger un arma muy poderos o un escudo), dada esta situación el juego tendría que calcular la dificultad de dichos enemigos (elementos) basándose en combinaciones de diferentes factores de dificultad que los diseñadores hayan dado a cada subelemento.

Progresión de la dificultad. La progresión de la dificultad se puede definir de dos maneras:

- Offline: Los diseñadores proporcionan pistas al sistema sobre dónde están los puntos fáciles o difíciles del juego, creando áreas de triggers asociándole una dificultad, así como la lista de los enemigos disponibles para dicha área, los puntos de aparición de los enemigos, etc. Esto también se conoce como especificación [1], que define cual es el estado del jugador "medio" en esa área y los distintos elementos que puede haber.

- Basado en mecánicas de juego: la dificultad está basada en algún elemento de la experiencia de juego, como por ejemplo la cantidad restante de nivel. A medida que el jugador se acerque a la consecución de todos los objetivos de un nivel, el sistema modulará la dificultad como sea necesario.

Habilidades del jugador. Necesitamos medir las habilidades del jugador para determinar la dificultad, para ello podemos apoyarnos en una serie de factores:

- Tiempo
- Dificultad de los elementos que ha completado
- Número de elementos que el jugador se encuentra en proceso de completar.

En un juego orientado al combate, estos elementos podrían traducirse en los siguientes

- Tiempo en derrotar un enemigo (relación indirecta)
- Dificultad que asignó el diseñador para ese tipo de enemigo (relación directa)
- Número de enemigos que están actualmente enfrentándose al jugador (relación directa)

En una primera aproximación la fórmula propuesta quedaría de la siguiente manera:

$$Medida\ básica\ de\ la\ habilidad\ del\ jugador = \frac{Dificultad\ del\ enemigo}{Tiempo\ en\ derrotarlo}$$

Pero necesitamos una escala similar que nos permita comparar entre habilidad del jugador y dificultad del enemigo. Así que añadimos una constante que representa la unidad básica de dificultad: el tiempo medio que emplearía un jugador novato (nivel de habilidad 1) en derrotar a un enemigo simple (nivel de dificultad 1), con los elementos más básicos (armas o potenciadores) la primera vez que juega al juego:

$$h = \frac{D_e}{T_e * d}$$

Dónde h es la habilidad del jugador, D_e la dificultad del enemigo, T_e el tiempo en derrotarlo y d es la unidad básica de dificultad.

Ahora necesitamos tener en cuenta otras amenazas o desafíos que esté afrontando el jugador también en ese momento. Usamos una constante, que debemos obtener mediante pruebas de usabilidad (ya que puede darse el caso, por ejemplo, de que tengamos armas que hagan daño a varios enemigos a la vez y otras que se centren en solo uno), que escala la dificultad por el número de oponentes.

$$h = \frac{D_e * f}{T_e * d}$$

Dónde f es el factor que escala la dificultad por el numero de oponentes.

Un ejemplo de constante para el número de enemigos podría ser el siguiente: +1 enemigo, el factor es ×1,125, +2 enemigos, ×1,25, +3 más enemigos: ×1,5. Si el jugador tiene que derrotar a tres enemigos a la vez, derrotar a uno de ellos es vez y media más difícil que si se enfrentara a ese enemigo solo.

Otros factores a considerar para la medición de la habilidad del jugador son:

- La media de la puntuación de habilidad en los últimos minutos jugados. Es decir no ajustar la dificultad en base a una única muestra, ya que si basamos el ajuste sólo en lo que está haciendo en ese momento, descubrir las mecánicas del sistema pueden resultar muy obvias (el jugador puede intencionadamente tardar más en vencer al enemigo más inmediato con el objetivo de hacer el juego más fácil). Además, es posible que sea conveniente no usar ningún tipo de ponderación, o al menos, no ponderar en función del más reciente ya que previene al jugador de hacer trampas (cambiar rápidamente su manera de actuar con el objetivo de que el sistema baje la dificultad).

- El ajuste de la dificultad debe ser invisible para el jugador, en caso de no serlo, debe descartarse dicho ajuste.

- El tiempo suele ser siempre un factor clave a la hora de medir habilidades, más aún en juegos que tengan algún tipo de acción (entiéndase aquí acción, a algo parecido al género de cine).

- Evitar factores adicionales (efectividad del arma, daño infligido) que no estén directamente asociados con la efectividad. Usando los elementos básicos ya los estamos teniendo en cuenta (una mala puntería con el lanzacohetes no significa que no esté derrotando a enemigos.Por ejemplo, el jugador puede tener mala puntería con un arma muy poderosa y sin embargo aún ser capaz de derrotar a enemigos rápidamente)

Agrupación de los elementos de dificultad. Es un paso previo antes de seleccionar los elementos para una dificultad determinada. Suelen ser una serie de restricciones extras para elegir los elementos que acentúan los objetivos del juego, ya que simplemente seleccionar los objetos de manera aleatoria no dé resultados aceptables. Para Pfeifer en [26], directamente eran las combinaciones posibles de los elementos. Estas restricciones pueden ser estéticas (los enemigos del tipo A siempre salen de dos en dos como ocurre en las películas de acción) o propias de la dificultad del juego (el enemigo del tipo A siempre va acompañado de uno del tipo B o completando el elemento X, se desbloquea el Y). Normalmente la manera más sencilla es haciendo que el diseñador añada información adicional como entrada al sistema. Ésta controla las relaciones específicas entre los elementos.

Secuenciador. El secuenciador elige aleatoriamente de entre los elementos disponibles que cumplan las características deseadas de la agrupación aquel cuya suma no supere el valor de la dificultad objetivo.

$$Dificultad\ objetivo = habilidad\ del\ jugador * modificador\ de\ dificultad$$

El modificador de dificultad viene definido por la progresión de la dificultad dado el lugar y/o el tiempo (muy difícil, normal, fácil, etc.). En caso de que existan subelementos (armaduras, armas, etc), éstos deben tener su propia dificultad. El secuenciador considerará cada una de las combinaciones válidas (enemigo A con armadura 1) y su dificultad será la suma individual de los elementos.

Ejemplo

Supongamos que tenemos 5 niveles de dificultad (discreta) con los siguientes modificadores:

- Muy difícil 2,0
- Difícil 1,5
- Normal 1,0
- Fácil 0,75
- Muy fácil 0,5

Si las habilidades del jugador están en 8, y estamos en una sección muy difícil, la dificultad objetivo es 16.

Podríamos permitir al diseñador especificar los modificadores directamente, pero la propuesta anteriormente (indirecta) tiene dos ventajas: limitar el número de dificultades a estados discretos lo hace más intuitivo de cara al diseñador, que entiende que tipo de resultado debería obtener; y permite cambiar los modificadores de modo que si 2,0 no es un valor suficiente para ser "muy difícil ", lo modifiquemos sin necesidad de ir ajustando las dificultades en cada área del juego. Supongamos que tenemos las siguientes restricciones para la información de las agrupaciones (sería cumplir una de las siguientes condiciones):

- Un enemigo muy fuerte.
- Dos enemigos.
- Tres enemigos, un más fuerte que los otros dos.
- Más de tres enemigos débiles.

Y los elementos de juego disponible son los siguientes:

- Comando, valor de dificultad 10.
- Francotirador, valor de dificultad 5.
- Infantería, valor de dificultad 4.
- Perro guardián, valor de dificultad 2.

Ahora de manera aleatoria el secuenciador elige dos enemigos (segunda restricción) de tipo comando. Pero el sistema no puede crear dicho grupo ya que si bien cumple las restricciones de agrupación, su valor de dificultad ($10+10 = 20$) excede demasiado la dificultad objetivo.

De modo que ahora elige otro agrupamiento distinto: tres enemigos, uno más fuerte que los otros dos. Elige el más fuerte de los disponibles (comando) y los otros dos aleatoriamente, resultando un elemento infantería y un perro guardián. El resultado es 16 justo la dificultad objetivo (no tiene por qué ser el valor justo, basta que esté lo suficientemente cerca) por lo que crea los enemigos de acuerdo con las especificaciones de los diseñadores (puntos de aparición, armas, armaduras, etc).

Ventajas

Uno de los puntos fuertes de este enfoque propuesto por Pfeifer, es que no sólo es fácil de ajustar (no hay que ajustar cada posición del enemigo, el tipo, etc), también es rápido y sencillo de desarrollar. Puede crear un ritmo más consistente, incluso en entornos no lineales. Además el hecho de tener varios tipos de situaciones y que no todas sean percibidas durante una partida, hace el juego más rejugable.

Mejoras

Una mejora sería mantener un seguimiento del porcentaje al cual el jugador cambia sus habilidades, medir cuanto está aprendiendo. También podríamos usar "conceptos" en el módulo de agrupación, que podrían forzar al jugador a aprender nuevas habilidades para mejorar si vemos que están estancadas (como forzar más combates cuerpo a cuerpo). A medida que los elementos dinámicos se hacen más complejos, esa información sobre la selección de éstos sería más rica y cobraría mayor importancia.

Críticas

La principal crítica de este modelo es la restricción de mantener las mecánicas simples, de otro modo podría ser muy difícil de ajustar la dificultad, ya que éstas afectan indirectamente a la experiencia del jugador. Además estas mecánicas de dificultad deben ser invisibles al jugador (simples e invisibles, más difícil aún).

Otra de las críticas es el hecho de que el sistema se sustente demasiado en una heurística un tanto semiarbitraria para definir la escala de dificultad.

3.3.2. La dificultad como un sistema de inventario

Robin Hunicke [16] (junto a Chapman) [15] ha hecho importantes avances en lo que a conseguir un sistema de DDA se refiere que implemente el ajuste del contenido de nivel. En [16] se describe el desarrollo del sistema de DDA llamado "Hamlet".

Hamlet está construido sobre el motor del juego Half-Life (*Valve*, 1998). Usa técnicas de teoría de inventario e investigación operativa para analizar y ajustar la provisión y demanda de objetos del juego con el fin de tener un control general de la dificultad.

Un sistema correcto de DDA debe mantener un equilibrio interno del juego y mecanismos de feedback, tienen que ser discretos e inescrutables. En la medida que sistemas comerciales han sido propuestos o publicados, según [16] hay dos enfoques principales.

- Manual (principalmente). Los diseñadores anotan las tareas y obstáculos con información sobre su dificultad [26]. Aunque esta información suele ser obtenida mediante pruebas de usuario.
- Mediante minería de datos y análisis offline[24]. Hunicke y Chapman proponen una técnica probabilística que evalúa dinámicamente la dificultad de un obstáculo basándose en la actuación del jugador, mientras está jugando.

MDA

En diseño de juegos el framework MDA (*Mechanics*, *Dynamics*, *Aesthetics*) es una herramienta usada para analizar juegos. Formaliza el consumo del juego dividiéndolo en tres componentes: mecánicas, dinámicas y estéticas. Estas tres palabras han sido usadas de manera informal durante años para describir varios aspectos de los juegos, pero el framework MDA proporciona definiciones precisas para estos términos y buscan explicar cómo se relacionan entre ellas y cómo influyen en la experiencia de usuario[25].

Hamlet

Usando el framework MDA como guía se crea el sistema Hamlet pensado para

[24] `http://www.gamasutra.com/view/feature/3017/designers_notebook_positive_.php`

[25] `http://en.wikipedia.org/wiki/MDA_framework`

juegos de acción en primera persona, que ayuda a reducir el feedback negativo innecesario sin que éste se interponga en la experiencia de juego.

El Hamlet regula las Mecánicas de inventario del juego (salud, munición, armas y escudos), los cuales afectan a las Dinámicas principales del juego (exploración y combate), mientras mantiene el ciclo general de la acción (que contribuye al ritmo frenético que tienen los juegos de acción, Estética). Es importante resaltar que Hamlet no cambia las mecánicas fundamentales o los objetivos estéticos del juego. Los jugadores pueden morir o fallar, ya que es una parte crítica del diseño general de éste, y es por eso que el balanceo y mecanismos de feedback originales son preservados. Hamlet está diseñado para mantener al jugador en la zona de flow propiciando ciertos estados y evitando otros, manteniendo al jugador en ciclos de interacción atractivos y apropiados dadas las habilidades del jugador y la experiencia de juego.

Arquitectura. El sistema Hamlet es principalmente un conjunto de librerías empotradas en el motor de juego de Half-Life, entre las que se incluyen:

- Monitorización de las estadísticas de juego de acuerdo a unas métricas predefinidas.
- Definición de acciones y políticas de ajuste.
- Mostrar datos y ajustes del sistema de control.
- Generar trazas de sesiones de juego.

A medida que el jugador se mueve por el entorno, Hamlet usa métricas estadísticas para monitorizar los datos del juego entrantes. Tras un tiempo, Hamlet estima el estado futuro del jugador basándose en estos datos. Cuando un estado no deseado pero evitable se predice, el sistema interviene y ajusta los parámetros necesarios del juego.

En el sentido más general el objetivo del algoritmo de DDA de Hamlet es asociar el estado actual del mundo con un conjunto de acciones de ajuste. Esto se realiza mediante una función de evaluación (que asocia el estado actual con la evaluación de la actuación del jugador) y una política de ajustes (que asocia la evaluación con un conjunto de ajustes a realizar). En un principio estas funciones pueden ser bastante complejas, no obstante, enfocándolo como un problema de productor-consumidor (como puede ser la salud del avatar del jugador), podemos simplificarlas (lo veremos más adelante).

Básicamente trata de predecir cuándo va a fallar el jugador y cuando detectemos dicho comportamiento ayudar al jugador a progresar en el juego.

Objetivos. Existen muchas formas de determinar cuando este tipo de ajustes son necesarios o deseables para una Buena experiencia de juego. Sin entrar en debates sobre estética y diseño de juegos, podemos parametrizar los objetivos del ajuste de la siguiente manera:

- Modificar el juego para estimular el entretenimiento del juego tanto a corto como a largo plazo, apoyando al jugador pero sin eliminar el feedback negativo que hace que el juego sea demasiado fácil y predecible.
- Intervenir sólo lo justo y necesario para que el comportamiento del sistema sea relativamente estable y predecible en el tiempo.

De esta forma:

- Valoramos cuando es necesario un ajuste.
- Determinamos que cambios o ajustes deberían hacerse.
- Ejecutamos los cambios ocasionando los menores problemas posibles.

Valoraciones

Como ya se ha dicho anteriormente el objetivo del Hamlet es determinar cuándo un jugador está fallando. En muchos juegos de acción (sobre todo en juegos de acción en primera persona), esto se caracteriza por un repetido déficit de objetos de inventario (sitios donde el jugador no logra obtener los objetos "demandados").

Observando las tendencias del uso de objetos de inventario por parte del jugador, podemos ver potenciales déficits de éstos y en consecuencia, determinar con precisión oportunidades potenciales de ajustes. Nuestra primera tarea es establecer métricas de valoración de datos estadísticos del juego. Empezaremos con una observación directa de la salud del jugador, más específicamente el daño que recibe a lo largo del tiempo.

Estimando el daño. Comenzamos asumiendo que tenemos una secuencia de mediciones aleatorias del daño $x(t)$, cada una de las cuales tiene una probabilidad de distribución p_d.

El daño total dado un punto en el tiempo es la suma de estas variables aleatorias. Por el Teorema Central del Límite esta suma converge a una distribución Gaussiana. Como estamos sumando distribuciones

$$\sum_{i=1}^{t} x(i)$$

La media μ resultante de la suma es la suma de la medias

$$\mu = E\big(\sum_{i=1}^{t} x(i)\big) = tE(x(t)) = t\mu_d$$

Y la varianza σ^2 es la suma de las varianzas

$$\sigma^2 = V\big(\sum_{i=1}^{t} x(i)\big) = t\sigma_d^2$$
$$\sigma = \sqrt{t}\sigma_d$$

La distribución del daño se convierte en:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/2\sigma^2} dx$$

Ahora podemos establecer la función de distribución acumulada de la distribución Gaussiana (también conocida como *error integral*):

$$F(d) = \int_{-\infty}^{d} p(x)dx = 1/\left[\sigma\sqrt{2\pi}\right] \int_{-\infty}^{d} e^{(x-\mu)^2/2\sigma^2} dx$$

$F(d)$ representa la probabilidad de recibir un daño d o menos en un tick (unidad de tiempo) dado. Normalmente esta función suele estar implementada como $erf(x)$ (*error function*) en las librerías estándares de matemáticas (como la de C++).

Con la distribución del daño, podemos aproximar los niveles de inventario para la salud del (avatar del) jugador.

Inventario. En cualquier sistema, los niveles de inventario dependen del ritmo del flujo de entrada y salida. Esto a menudo es tratado en términos de oferta y demanda.

En el dominio de los juegos de acción, el inventario del jugador varía continuamente. Se le proporcionan ítems al jugador a medida que exploran el entorno (dentro de cajas por ejemplo), y que reducen al realizar interacciones con éste (munición y salud se gastan en combate, por ejemplo).

Usando ecuaciones de teoría de inventario, podemos modelar el inventario global del jugador así como el flujo si hay una entrada y salida de objetos de inventario en el sistema.

Detectando déficit. Para predecir un déficit en el inventario, sea z el nivel del inventario en un momento determinado. El déficit esperado es la probabilidad acumulada del daño superior al nivel inicial o P_s , que se calcula mediante la siguiente fórmula:

$$\begin{aligned} P_s = P(x(t) > z) &= 1 - P(x(t) < z) = 1 - F(z) \\ &= 1 - \frac{1}{\sigma\sqrt{2\pi}} \int_z^{\infty} e^{-(t-\mu)^2/2\sigma^2} dt \end{aligned}$$

Con esta ecuación y la función *erf*, pordemos calcular el déficit como una función de la vida/salud inicial, media y desviación estándar de la pérdida de vida a través del tiempo.

Podemos usar este cálculo de déficit de salud inminente como un indicador de cuando jugador esté necesitado. Entonces podemos pasar de la valoración al ajuste, analizando los sistemas que afectan al objeto de inventario en cuestión y ajustando el juego de manera adecuada.

Ajustes

Acciones de ajustes combinadas con estimaciones de costes forman las políticas de ajustes. Hamlet posee dos políticas de ajustes.

Acciones. Cuando esté completado, Hamlet soportará dos tipos de acciones de ajuste.

- **Zona de confort.** Esta política mantiene al jugador en un rango de salud todo el tiempo. Las entidades se ajustarán para que disparen con menor frecuencia y con menos precisión y los botiquines están fácilmente disponibles. El objetivo es mantener al jugador sobre el 50 % de su salud, ocasionalmente llevándolos cerca del 25 % o subiéndolo al 75 %.

  El ensayo/error es importante en esta política –los enemigos sólo serán ajustados si vencen con comodidad repetidamente al jugador. El comportamiento es parecido al de una niñera, interviene con frecuencia pero deja lugar para errores. En general, la política estará caracterizada por la predecible oferta y la constante demanda.

- **Zona de disconfort.** Esta política está diseñada para jugadores más experimentados. Las entidades en el juego son muy precias y la munición y botiquines escasean. La idea es mantener al jugador en el límite de sus posibilidades, constantemente alerta de enemigos y encontrándose siempre en un rango de salud del 15

  Esta política es la que podríamos llamar "sargento". Establece unas cotas altas, pero proporcionando el suficiente feedback positivo como para mantener al jugador motivado. Esta política está caracterizada por una oferta alta y esporádica y por una demanda escasa orientada al objetivo.

Ejemplo

Como ejemplo de funcionamiento del Hamlet pondremos el expuesto en [16].

Cuando el jugador llega a un punto determinado del juego, hay cuatro enemigos y éste tiene la salud al 45 %. Hamlet observa las estadísticas del inventario para ver si el jugador está fallando, lo que en este caso se traduce como morir repetidas veces antes de completar el enfrentamiento, y normalmente sobreviviendo la mayoría de los enemigos.

En respuesta inmediata a este fallo, el sistema puede:

- Añadir un botiquín en algún lado de la escena.
- Aumentar la fuerza del arma del jugador.
- Reducir la precisión o fuerza de los ataques enemigos.

Dependiendo del éxito de estas acciones individuales, el Hamlet puede intervenir de nuevo. Si transcurrido un tiempo, el jugador requiere ajustes más significativos que los niveles iniciales propuestos por el diseñador, entonces también puede:

- Reducir la fuerza inicial y vida de los enemigos pendientes.
- Añadir más botiquines y salud a los cuerpos de los enemigos derrotados.
- Cambiar los enemigos por los de otro tipo en los que el jugador haya sido capaz de vencer con más facilidad.

La clave está en intervenir iterativamente, permitiendo ensayo y error. El juego cambiará gradualmente y se acomodará al jugador actual.

Críticas

Se trata de una técnica muy interesante para ajustar contenido del nivel y que parece que potencialmente podría llevarse a otros géneros además de los juegos de acción, aunque sería de un gran interés verlo aplicado. Juegos de rol, por ejemplo, donde los elementos del juego varían mucho más que simples enemigos e ítems, como por ejemplo, el tipo de magia disponible.

No obstante la principal crítica que se le puede achacar es que el Hamlet a fecha de la memoria, todavía no está terminado o no existe (el autor de la memoria no ha sido capaz de encontrarlo por ninguna parte) lo que lo hace muy difícil de evaluar fuera de los resultados de los artículos científicos.

4. Estado del Arte

Para finalizar el capítulo vamos a tratar los trabajos más actuales en lo que a ajuste dinámico de la dificultad se refiere. En [23] se presenta una formulación del ajuste de la dificultad como un problema de aprendizaje online sobre conjuntos parcialmente ordenados y un algoritmo que está considerado (a fecha de la memoria) como estado del arte.

La idea de este algoritmo es proporcionar un mecanismo “universal” en lugar de heurísticas, las cuales requieren un testeo excesivo y son muy específicas del dominio. A través de una adaptación online, a partir de unos datos específicos del juego, el “historial” (acciones previas, acciones, reacciones, etc) del jugador en la partida y las modificaciones posibles (que pueden ser determinadas explícitamente mediante un conjunto finito o implícitamente mediante un conjunto infinito), produce una salida con las modificaciones apropiadas para ajustar la dificultad.

O. Missura y T.Gärtner proponen en [23] una formalización teórica del ajuste dinámico de la dificultad como un problema de aprendizaje.

Algoritmo POSM

Como hemos venido comentando a lo largo del trabajo el objetivo de un ajuste de la dificultad es proporcionar una dificultad adecuada para los diferentes jugadores. El trabajo de O. Missura y T.Gärtner, pretende esto mismo, apoyándose en técnicas de aprendizaje automático ya que la dificultad depende de muchos factores y no es la misma para todos los jugadores.

A lo largo de la memoria hemos mencionado que la manera tradicional de ajustar la dificultad mediante niveles está muy limitada. Por un lado si hay pocos niveles (de dificultad), la elección de la dificultad puede no ser difícil, pero es poco probable que sea satisfactoria. Por otro lado, si el número de niveles (de dificultad) es muy grande, es más probable que se nos proporcione un ajuste satisfactorio, pero el problema ahora sería elegir ese nivel concreto. Además el hecho de estar creando configuraciones de juego apropiadas para dichos niveles es una tarea que consume mucho tiempo. El enfoque adoptado en [23] es formalizar el ajuste de la dificultad como un *meta-juego* entre un maestro o director y un jugador, donde el primero trata de predecir el estado de juego más acorde con la dificultad, haciendo una analogía, podría ser el director de juego (*game master*) de una partida de rol.

Las acciones del jugador y sus respuestas al estado del juego actual proporcionan al director la información necesaria para realizar él sus movimientos (cambiar el nivel de dificultad de manera apropiada). La figura 3.15 muestra un diagrama de cómo sería este meta-juego. No obstante, se debe aclarar que en [23] simplemente se asume que tenemos una información suficiente que le permita al director saber si el nivel de dificultad es el correcto. Está información suele ser muy dependiente del juego (velocidad de resolución de puzles, el jugador está perdiendo vidas muy rápido, etc) pero como ya hemos dicho, obviamos que debe ser incluido y que no, simplemente asumimos que tenemos acceso a ella de alguna manera.

El ajuste de la dificultad del meta-juego se hace en un conjunto parcialmente ordenado que refleja la relación 'más difícil que' entre el conjunto de los estados de juego. Pongamos como ejemplo que existen dos entradas a un castillo, una con una puerta cerrada y otra abierta, pero vigilada por guardias. Para un jugador cuyas habilidades de infiltración son buenas (o que tiene el objeto para abrir la puerta) pero las de combate no, la situación primera es más fácil, de manera opuesta para un jugador más habilidoso con el combate (o que no puede abrir la cerradura siquiera), la situación dos será la más fácil. De manera general, no podemos comparar los estados ya que la dificultad no sólo depende de unas habilidades específicas dentro del juego (infiltración, combate, etc) que

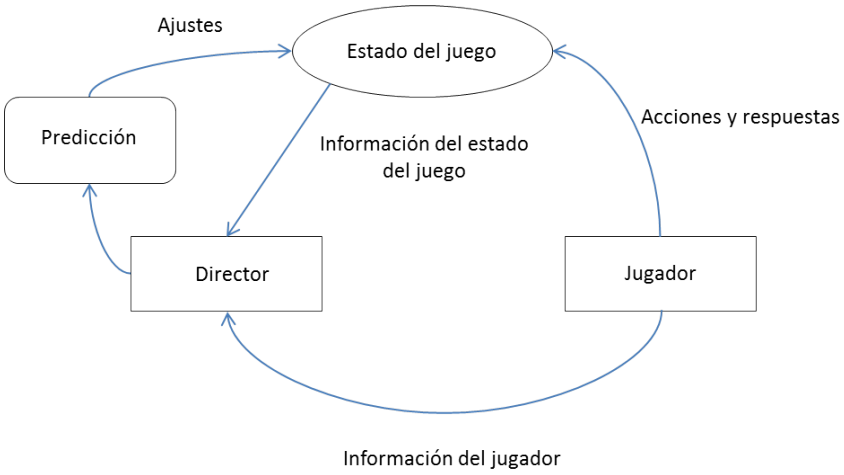


Figura 3.15: Funcionamiento del meta juego. El director realiza una predicción del estado futuro en base a la información del estadoa actual y la del jugador. Tras esto realiza los ajustes que sean necesarios.

sí podríamos comparar, sino también de cualidades intrínsecas de la persona (capacidad estratégica, paciencia, etc) que son muy difíciles o imposibles de incluir en el modelo. Es por ello por lo que decimos que los estados del juego constituyen un conjunto parcialmente ordenado.

En este conjunto parcialmente ordenado, el problema de ajustar la dificultad se reduce a predecir un *vertex cut* (número de vértices para dejar un grafo inconexo) desconocido entre los estados 'demasiado difíciles' y 'demasiado fáciles'.

4.0.3. Formulación del problema

Para poder investigar el problema desde un punto de vista teórico, lo hemos establecido como un juego entre un director y un jugador, que se juega en conjunto parcialmente ordenado que refleja la relación 'más difícil que'. Donde además, se juega por turnos y cada uno de ellos tienen los siguientes elementos:

1. El director elige una dificultad

2. El jugador juega una ronda en esa dificultad

3. El director calcula si el nivel era 'demasiado difícil', 'justo' o 'demasiado fácil' para el jugador

El director tratará cometer el menor número de errores en su intento por elegir la dificultad 'justa'.

Para simplificar el análisis se realizan las siguientes asunciones:

- El conjunto de dificultades es finito.
- En cada turno se respeta el orden parcial (oculto) de los niveles de dificultad, es decir:
  - No existe ningún estado X 'más difícil' que otro estado Y que sea 'demasiado difícil' y que al mismo tiempo X sea 'justo' o 'demasiado fácil'.
  - No existe ningún estado que sea 'más difícil' que otro estado 'justo' y 'demasiado fácil' al mismo tiempo.
- En cada turno, por cada par de niveles donde uno es 'demasiado difícil' y otro 'demasiado fácil' existe un nivel de dificultad entre ellos 'justo'.

Como no podemos hacer ningún tipo de asunciones con respecto al jugador, compararemos el algoritmo con el mejor nivel de dificultad estática elegido. Si el juego determina un subconjunto lógico de niveles de dificultad, el algoritmo se limitará a elegir entre éstos y comparar con la mejor dificultad estática de estos niveles. Asumiremos también que estos subconjuntos están linealmente ordenados, es decir, forman una cadena[26].

Bajo estas premisas que acabamos de mencionar, podemos ver el conjunto parcialmente ordenado como un grafo dirigido sin ciclos (DAG), en cada turno etiquetado por tres colores (rojo para 'demasiado difícil', verde para 'justo' y azul para 'demasiado fácil') de tal manera que:

- Por cada camino en el grafo entre dos vértices etiquetados con el mismo color, todos los vértices que conforman el camino tienen el mismo color.
- No existe un camino de un vértice verde a otro rojo, y no existen caminos de un vértice azul a un rojo o a uno verde.
- Por cada camino de un vértice rojo a otro azul, existe al menos un vértice verde, que son los que forman el vertex cut.

[26] `http://en.wikipedia.org/wiki/Antichain`

La coloración puede cambiar en cada turno siempre que se cumplan las restricciones anteriores. El algoritmo de aprendizaje (el director) no ve los colores, pero debe de elegir los vértices verdes lo más frecuentemente posible. El feedback recibido contiene el verdadero color del vértice elegido. Se trata de aprender el corte mediante el feedback.

Si ignoramos el orden parcial de los niveles de dificultad, tenemos un problema de *n-bandidos* [27]. Tenemos N palancas a las cuales un adversario les asigna un valor de pérdida en cada iteración (0 para las palancas 'justas', 1 para las demás). El objetivo del algoritmo es elegir en cada iteración la palanca que minimice las pérdidas. La dificultad del aprendizaje viene por el hecho de que al algoritmo sólo se le revela la pérdida de la palanca elegida. La medida de ejecución es la diferencia entre la pérdida de la palanca elegida por el algoritmo y la mejor elegida en "retrospectiva", es decir, si supiésemos cual es la mejor palanca. El mejor algoritmo que se conoce hasta la fecha, que no usa ningún tipo de información adicional, es la *Estrategia del Bandido Mejorada* (*Improved Bandit Problem*). Su límite superior de la medida de ejecución es el orden de $\sqrt{NT\ln(T)}$, donde T es la cantidad de iteraciones.

Algoritmo

Vamos a presentar el algoritmo para predecir un vértice que se corresponda con el nivel de dificultad 'justoén un conjunto finito parcialmente ordenado $(K, \succ)$ de niveles de dificultad. El orden parcial cumple que para cada par $i, j \in K$ escribimos $i \succ j$ si el nivel de dificultad i es 'más difícil que' j. El conjunto de cadenas que el entorno puede elegir en cada turno se denota por C y la tasa de aprendizaje por β. La respuesta o_t que el algoritmo puede observar es $+1$ si la dificultad elegida era 'demasiado fácil', 0 si era la 'justa' y -1 si era 'demasiado difícil'. El algoritmo guarda unas creencias w de cada vértice para ser el nivel 'justo' y actualiza estas creencias si la respuesta observada fue 'demasiado fácil' o 'demasiado difícil'.

La idea principal del algoritmo (de coste exponencial) 7 es, en cada turno, una vez que se ha elegido una cadena, asegurarnos de actualizar la mayor cantidad de creencia posible. Para asegurarnos de ellos, calculamos para cada nivel k la creencia por arriba y por abajo en la cadena. A_t recoge las creencias de todos los niveles de la cadena que son 'más difícil' que k y B_t las creencias de los que son 'más fácil'. Dependiendo de la observación, podremos actualizar las creencias de los que están por arriba o por debajo del nivel elegido, que considerando el caso peor, se puede garantizar una cantidad mayor o igual que $min\{Bt(k), At(k)\}$. Para conseguir el mejor rendimiento, elegimos el k que nos

Algoritmo 7 Algoritmo Partially-Ordered-Set Master (POSM) para el ajuste de la dificultad

Entrada: Un parámetro $\beta \in (0,1)$, K niveles de dificultaf K, un orden parcial $\succ$ en K, C subconjuntos C de K cada formando una cadena en el conjunto parcialmente ordenado $(K, \succ)$ y una secuencia de observaciones $o_1, o_2, ...$

1: $\forall k \in K$: sea $w_1(k) = 1$
2: **para** cada turno $t = 1, 2, ...$ **hacer**
3: Coger una cadena $c_t \in C$
4: Sea $A_t(k) = \sum_{x \in c_t : x \succeq k} w_t(k), \forall k \in c_t$
5: Sea $B_t(k) = \sum_{x \in c_t : x \preceq k} w_t(k), \forall k \in c_t$
6: Predecir $k_t = \arg\max_{k \in c_t} \min\{B_t(k), A_t(k)\}$
7: Observar $o_t \in \{-1, 0, +1\}$
8: **si** $o_t = +1$ **entonces**
9:

$$\forall k \in K : \ sea\ w_{t+1}(k) = \begin{cases} \beta w_t(k) & sik \succeq k_t \\ w_t(x) & en\ otro\ caso \end{cases}$$

10: **fin si**
11: **si** $o_t = -1$ **entonces**
12:

$$\forall k \in K : \ sea\ w_{t+1}(k) = \begin{cases} \beta w_t(k) & sik \preceq k_t \\ w_t(x) & en\ otro\ caso \end{cases}$$

13: **fin si**
14: **fin para**

dé el mejor caso peor.

La forma de elegir la cadena será dependiente del juego. No obstante, en ausencia de otros supuestos o conocimientos sobre el juego, una posible estrategia es muestrear de manera aleatoria uniforme una cadena del camino mínimo recubridor del conjunto (parcialmente ordenado).

Mencionar que por simplicidad en [23] se limitan a un solo caso de orden lineal de niveles de dificultad, es decir, a una sola cadena.

Análisis teórico

A continuación analizaremos los límites del algoritmo relativos al número de errores que el nivel de dificultad estático realizaría en cada cadena. Aunque ya hemos mencionado que en los experimentos de [23] usan una sola cadena, el análisis se hará suponiendo que no existe esta limitación.

El análisis se fundamente sobre la noción del camino recubridor de K, es decir, el conjunto de caminos que cubren K. Un camino es un subconjunto de K que está totalmente ordenado. Se dice que un conjunto de caminos cubren K si la unión de dichos caminos es igual a K. Podemos elegir cualquier camino recubridor, pero el mínimo será el que dé el límite más ajustado. Éste se puede encontrar en tiempo polinomial $|K|$ y su tamaño es igual al tamaño de la anticadena[27] más grande en $(K, \succ)$. Denotaremos al conjunto de caminos como C.

Para todo $c \in C$, la cantidad de creencia en cada cadena es $W_t^c = \sum_{x \in c} w_t(x)$.

En cada iteración en el que el algoritmo selecciona un nivel de dificultad inapropiado, actualizamos al menos la mitad de los pesos de la cadena elegida. Para demostrar esto, primero hemos de demostrar que $max_{k \in c_t} min\{A_t(k), B_t(k)\} \geq W_t^{c_t}/2$. Para ello hacemos:

$i = argmax_{k \in c_t}\{B_t(k) | B_t(k) < W_t/2\}$ y $j = argmin_{k \in c_t}\{B_t(k) | B_t(k) \geq W_t/2\}$

De esta manera obtenemos $i, j \in c_t$ para los cuales $B_t(i) < W_t/2 \leq B_t(j)$, que además son consecutivos, es decir $\nexists k \in c : i \prec k \prec j$. Tales i, j existen y son únicos ya que $\forall x \in K : w_t(x) > 0$. Tenemos entonces $B_t(i) + A_t(j) = W_t^{c_t}$ y por tanto también $A_t(j) > W_t/2$. Lo que implica

[27] `http://en.wikipedia.org/wiki/Antichain`

$$W_t/2 \leq min\{A_t(j), B_t(j)\} \leq max_{k \in c_t}\ min\{A_t(k), B_t(k)\}$$

Lo que prueba lo declarado anteriormente sobre la actualización de pesos.

Para cada error cometido en la cadena c_t, el peso de la cadena cambia de la siguiente manera

$$W_{t+1}^{c_t} \leq W_t^{c_t}/2 + \beta W_t^{c_t}/2 = W_t^{c_t}(1+\beta)/2$$

Así tenemos $W_T^c \leq W_0^c(1+\beta)_c^m/2^{m_c}$. Como la longitud de la cadena está acotada por el número de niveles de dificultad, tenemos

$$W_T^c \leq K(1+\beta)^{m_c}/2^{m_c}$$

El peso de cada nivel de dificultad se actualiza sólo si las respuesta observada implica que el nivel no era 'justo'. Por lo tanto, β^{M_c} es el límite inferior del peso de un nivel de dificultad en la cadena y por consiguiente

$$W_T^c \geq \beta^{M_c}$$

. Resolviendo

$$\beta^{M_c} \leq K(1+\beta)^{m_c}/2^{m_c}$$

Para m_c, tenemos

$$m_c \leq \left\lfloor \frac{\log_2 K + M_c \log_2 1/\beta}{\log_2 \frac{2}{1+\beta}} \right\rfloor$$

Crítica

A pesar de ser algo muy novedoso, el trabajo de Missura y Gärtner da la impresión de estar más orientado a juegos de tablero (o aquellos de teoría de juegos) que en videojuegos. Estaría por ver una aplicación a un videojuego, y si es posible aplicarse a un videojuego que no esté dividido en turnos.

Capítulo 4

Casos de estudio

1. Flow

A continuación pasamos a analizar la tesis y juego de Jenova Chen *Flow* [10] en la cual se implementa un ajuste dinámico de la dificultad. Se ha elegido como caso de estudio ya que representa una vuelta de tuerca a los DDA orientándolos más al usuario

Para Chen, un DDA es una parte más de los elementos fundamentales que conforman el concepto de Flow (que ya vimos en el capítulo uno). El enfoque particular de Chen se centra, en lugar de diseñar un DDA para juegos, en diseñar un sistema general de Flow a partir de sus tres elementos fundamentales:

- El juego es intrínsicamente reconfortante, premia, y el jugador está en disposición de jugarlo.
- El juego ofrece una cantidad de retos acorde con las habilidades del jugador que le permite profundizar más en el juego.
- El jugador necesita sentir un control personal sobre el juego

El resultado de estos elementos es la pérdida de la noción del tiempo y cohibición (Flow).

1.1. Diseñar Flow en juegos

Los videojuegos como medio de comunicación suelen verse compuestos por dos componentes principales:

- Contenidos del juego, el "alma" de éste; la experiencia específica a transmitir para la que se ha diseñado el juego.
- Sistema de juego, el "cuerpo"; un software interactivo que hace de puente entre los contenidos del juego y el jugador a través de la propias interacciones, los aspectos visuales y sonoros.

Si asumimos que el contenido es atractivo para el público, diseñar un videojuego tiene mucho que ver con mantener al jugador en la zona de flow, para que, eventualmente, complete el juego. Por lo tanto, para Chen, es importante que el sistema de juego sea capaz de abarcar diferentes experiencias de juego para los jugadores. Desafortunadamente, cada una de las diferentes personas, tienen diferentes habilidades y zonas de Flow, por lo que un juego bien diseñado puede mantener en esta zona a los jugadores "normales". Pero quizás no tanto a los novatos o experimentados (figura 4.1).

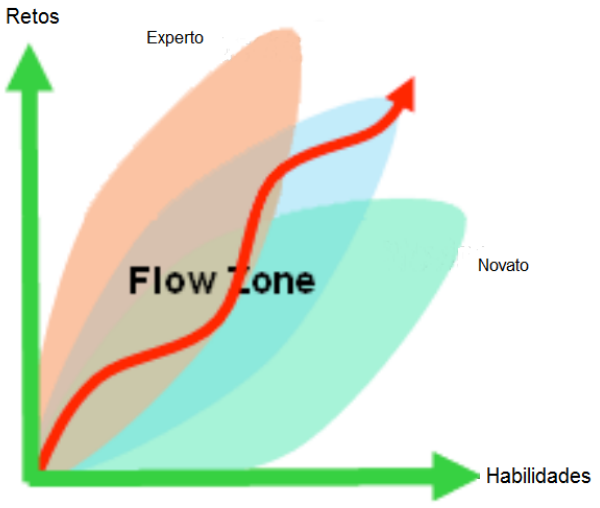


Figura 4.1: Diferentes jugadores y zonas de Flow

Expandir la zona de Flow

Para diseñar un juego capaz de abarcar un mayor público, la experiencia no puede ser lineal o estática. En su lugar, necesita cubrir un amplio espectro

de experiencias potenciales que encajen en las diferentes zonas de flow de los jugadores. Para expandir dicha zona, es el diseño en sí lo que necesita ofrecer la variedad de experiencias: desde tareas extremadamente simples a resolución de problemas complejos. Los propios jugadores siempre deberían poder encontrar el nivel justo de reto que los mantenga su zona de flow.

Estas diferentes opciones deben ser obvias de modo que el jugador pueda identificarlas fácilmente con que corresponde y profundizar en ellas.

Ajuste del juego y Flow estático

Tradicionalmente los diseñadores de niveles, ajustaban los parámetros de juego mediante pruebas de usuario (*playtesting*) hasta que la experiencia conseguida era la adecuada. El ajuste del juego nos indica la rigidez y linearidad de la experiencia final, que es ajustada por estos diseñadores y testers que es muy difícil que representen toda la variedad del público al que va dirigido. Este ajuste es muy estático y no puede adaptarse a los diferentes tipos de jugadores. Para obtener experiencias más óptimas para un público más amplio, no basta simplemente con expandir esa zona de flow (que podría entenderse como los diferentes tipos de dificultad), necesitamos un sistema adaptativo que sea capaz de entrelazar los distintos tipos de experiencias y ajustarlas basándose en los propios jugadores.

Podemos apreciar como también para Chen, la idea de un sistema adaptativo es fundamental para ajustarse a los distintos tipos de jugadores.

Ajuste del Flow Pasivo

Uno de los principales dilemas en el Flow, como ya vimos en el capítulo 2, es la controversia que implica implementar un sistema de ajuste del juego. Muchas investigaciones (entre los que se encuentra el presente trabajo) se centran en diseñar un sistema que ajuste la dificultad basándose en la actuación del jugador. Este tipo de DDAs orientados al sistema, funciona bajo un bucle iterativo de ajustes (figura 4.2), el cual consiste en los siguientes cuatro elementos fundamentales:

1. **Jugador**. Genera los datos brutos mediante el juego.
2. **Sistema de Monitorización**. Elige los datos importantes que reflejen el estado de Flow o reto del jugador y los pasa el Sistema de Análisis.

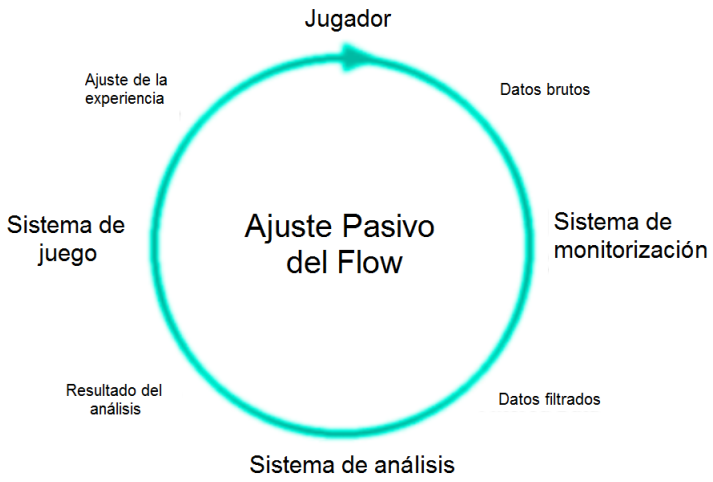


Figura 4.2: Bucle de un DDA orientado al sistema

3. **Sistema de Análisis**. Analiza el estado de Flow y notifican al sistema que necesita ser cambiado.

4. **Sistema de juego**. Aplica los cambios al juego basándose en los resultados del análisis.

No obstante para Chen, este sistema sigue teniendo graves carencias (figura 1.1):

- **Ausencia de datos directos**. A fecha de la memoria no es posible saber o leer lo que está pensando el jugador, la única conexión que tenemos es mediante algún tipo de mando o controlador. Con un número limitado de entradas la posibilidad de observar el estado de concentración del jugador es muy difícil. Tenemos que basarnos en asunciones y estadísticas incompletas.

- **La actuación del jugador no refleja su estado de Flow**. Las investigaciones han conseguido formas que de alguna manera estiman la actuación del jugador basándose en datos límitados como la "precisión" o el "número de intentos". No obstante, estos son datos objetivos, mientras que el Flow es subjetivo. Un jugador puede encontrarse en dicho estado simplemente paseándose por el nivel sin ni siquiera terminarlo, y este tipo de DDAs tendrán problemas tratando de percibir eso.

Figura 4.3: Problemas típicos de un DDA orientado al sistema

- **Análisis basado en supuestos o asunciones**. Asumir ciertas cosas nunca funciona para el gran público. Cuando un jugador disfruta cometiendo un acto suicida absurdo en *Grand Thief Auto* (*Rockstar*) sería un poco absurdo y ridículo que el DDA tomase en cuenta este hecho a la hora de estimar mediante el número de intentos.
- **Cambios que se basan en un diseño rígido**. La manera en que el juego se ajusta está predeterminada por el diseñador. Se puede decidir cuantos cambios deben aplicarse. No obstante, las preferencias de un diseñador nunca lograrán representar a la gran masa.

Ajuste del Flow Activo

La mayoría de los diseños de DDAs orientados al sistema se centran en un aspecto en particular, encontrar el equilibrio entre reto y habilidad. No obstante,

Chen argumenta que ignoran un punto fundamental, que es el hacer sentir al jugador un control sobre la actividad del juego.

En los medios de comunicación pasivos, como es este, tradicionalmente la sensación de control viene a su vez de la sensación de progresión y de la recepción de un feedback positivo. Pero en los videojuegos, el control no sólo se puede obtener de la progresión. También se puede obtener explorando, eligiendo opciones o interacciones sensatas, así que ¿por qué no dar esto a los jugadores y dejar que generen su propia experiencia de Flow?

Para crear juegos así es necesario expandir la zona de Flow, el juego tiene que ofrecer un amplio espectro de actividades y dificultades para los distintos tipos de jugadores. Es el propio jugador el que elige diferentes opciones y diferentes ritmos de juego (figura 4.4).

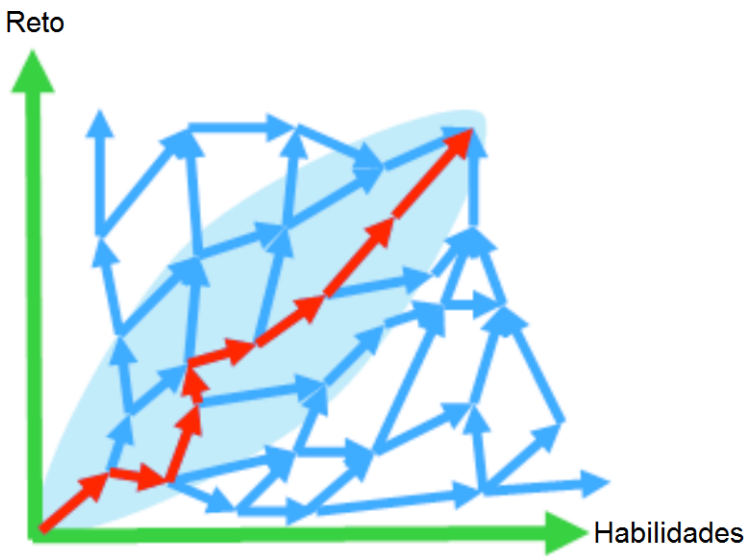


Figura 4.4: Ajuste del Flow mediante elecciones

Añadiendo elecciones en el juego

Chen propone un DDA orientado al jugador (un DDA activo), de modo que éste controle su propia experiencia en el juego. El problema radica en implementar estas decisiones, algo que no es nada trivial.

Para ajustar las experiencias dinámicamente y reducir el ruido, estas elecciones deben aparecer con una frecuencia relativamente alta. Pero estas mismas

elecciones pueden ser potenciales interrupciones cuando el jugador se encuentra en su zona de Flow.

La solución más rápida y fácil podría ser implementar un sistema de monitorización para detectar cuando es buen momento para ofrecer elecciones al jugador. No obstante, esto adolece de lo ya mencionado anteriormente en los DDAs orientados al sistema, no es capaz de medir o detectar algo tan subjetivo como el Flow. La solución propuesta por Chen y que veremos a continuación implementada en su juego *FlOw*, es meter las elecciones como parte del juego, permitir al jugador que trate las decisiones como parte de la partida. Así sus elecciones serán intuitivas y reflejarán sus deseos en ese momento.

1.2. Implementación del Flow en juegos

Para probar su teoría sobre los DDA orientados al jugador y las metodologías para diseñar juegos de esta manera, Chen implementa dos sencillos juegos con estos elementos en mente. De estos dos, nos centraremos en uno de ellos, FlOw, por ser el más complejo y famoso.

Como acabamos de mencionar FlOw se crea para probar un DDA orientado al jugador con elecciones dentro del juego. En éste, el jugador mueve una especie de organismo acuático con el ratón a través de una especie de entorno marino, donde aparecen otros organismos que pueden comerse para así evolucionar a nuestro organismo. El objetivo ir bajando a mayores profundidades hasta alcanzar el nivel más profundo.

El gameplay es intencionalmente simple para que sea fácil de evaluar la eficiencia del DDA. Las únicas acciones realizables por el jugador son: moverse por el entorno y comerse los otros organismos si están delante de su boca.

Existen tres versiones de FlOw a fecha de la memoria, una hecha en Flash que puede jugarse gratuitamente en `http://interactive.usc.edu/projects/cloud/flowing/` y otras dos descargables en PlayStation 3 y PlayStation Portable.

Aplicación de la Metodología

Expansión de la zona de Flow

Con motivo de abarcar a jugadores casuales o no jugadores, el control se reduce al mínimo, simplemente para mover a nuestro avatar a través del "océano".

Para atraer a jugadores más clásicos o expertos, se le proporciona una serie de enemigos a comerse con los que evolucionar el avatar de manera estratégica y con una acción más intensa.

Ajuste del Flow Activo

FlOw tiene 20 niveles, donde a medida que bajamos a lo más profundo de este particular océano nos aparecen criaturas más grandes y más difíciles.

La característica que diferencia a FlOw en este aspecto es que no es necesario completar un nivel para pasar al siguiente, le ofrece al jugador controlar su progreso. Comiendo distintos tipos de presas, roja o azul, los jugadores pueden avanzar o retroceder un nivel respectivamente.

El juego proporciona un leve "castigo" a la hora de morir. Si el jugador muere en un nivel, se le devuelve al nivel anterior con el avatar ligeramente menos evolucionado. El jugador, si quiere, puede evitar los enfrentamientos y saltarse el nivel, para volver más tarde si quiere.

Añadiendo elecciones de DDA al juego

Para Chen, en FlOw los jugadores pueden personalizar su experiencia a través del gameplay de manera natural, nadando y comiendo. Nadando más lejos o más cerca de otros organismos, y comiendo diferentes tipos de comida, los jugadores equilibran subconscientemente su experiencia.

1.3. Conclusiones del trabajo de Chen

Recapitulando las ideas de Jenova Chen, las cuales están más enfocadas al diseño de juegos que a la IA (la unión entre ambas empieza a cobrar más relevancia[1][2]), para conseguir un juego que se adaptativo y que logre mantener al usuario concentrado, en su estado de Flow, existen tres condiciones fundamentales en el campo del diseño de juegos:

1. Se parte de la premisa de que un juego es intrínsicamente gratificante y el jugador está dispuesto a jugarlo.
2. El juego ofrece la cantidad justa de reto de acuerdo con las habilidades del jugador, al cual se le permite profundizar más en el juego.

[1] `http://dankline.wordpress.com/2011/06/29/the-ai-director-of-darkspore/`
[2] `http://chrishecker.com/Structure_vs_Style`

3. El jugador necesita percibir un control sobre la actividad del juego.

Para mejorar esta experiencia, propone una serie de metodologías para hacer los juegos más divertidos y que lleguen a un mayor público:

1. Abarcar un espectro más amplio de gameplay con diferentes dificultades y aspectos.
2. Crear un DDA activo orientado al jugador que permita a los diferentes jugadores jugar a su propio ritmo.
3. Añadir elecciones de DDA como parte del núcleo de mecánicas de juego y permitir al jugador hacer dichas elecciones mientras juega.

2. Left 4 Dead

Llegamos al juego que motivó el presente trabajo (aunque ha acabado divergiendo bastante). Vamos a analizar ciertas características del juego ya que se trata de uno de los pocos éxitos comerciales conocidos del DDA, que es tan solo un elemento más del Director de Juego. A fecha de la memoria el término Director de Juego o de IA, está ligado a este juego de alguna forma ¿pero en qué consiste la dirección de la IA? Podríamos resumirlo como un algoritmo que reemplaza la aleatoriedad y gestiona la experiencia. Es esa gestión de la experiencia lo que vamos a tratar abarcar en esta sección, qué tipo de técnicas se han utilizado para ello.

2.1. Descripción del juego

Comencemos con una breve descripción del juego. *Left 4 Dead* (abreviado L4D) es un juego de acción cooperativo en primera persona, con un "toque" arcade. Ambientado en una pandemia, tomamos el papel de un miembro de un grupo de cuatro supervivientes, tratando de avanzar por el nivel hasta llegar a zonas seguras y finalmente llegando a un punto de evacuación. Durante el juego podremos obtener una serie de armas distintas y de ítems extras como botiquines o armas arrojadizas. Aparte de los ítems y armas, en el juego destacan cuatro elementos principales:

- **El equipo de supervivientes.** El grupo protagonista, bien controlados por jugadores o por IA. Deben colaborar eliminando infectados abriéndose

paso hasta los diferentes puntos de control del juego, para eventualmente, terminar el nivel.

- **Los infectados.** Los zombies "básicos" del juego. Suelen atacar en hordas compuestas por un gran número de ellos. Estas hordas además suelen acompañar a los infectados más "especiales".
- **Los infectados especiales.** Existen cuatro infectados especiales, cada uno con una serie de características (pero irrelevantes en lo que aquí nos concierne). No aparecen de manera tan frecuente como los infectados.
- **Los infectados jefes.** Existen dos tipos de infectados más por encima de los especiales. Aparecen pocas veces a lo largo de la partida, algunas en puntos determinados.

2.2. Gestión de la experiencia y ritmo dramático adaptativo

Una de las características principales de Left 4 Dead es su rejugabilidad, y gran parte de ello tiene la culpa la gestión de la experiencia. Al igual que el secuenciador de Pfeifer parte de una premisa que ellos llaman *Impredecibilidad Estructurada*.

Esta impredecibilidad se basa en tener una colección de posibilidades interesantes a seleccionar en tiempo de ejecución en un momento determinado mediante una serie de restricciones aleatorias pero intencionalmente establecidas por los diseñadores, es un concepto parecido al 'Conjunto de elementos de juego disponiblesé Ínformación de los elementos agrupados'del secuenciador de Pfeifer.

El hecho de tener muchas situaciones "dramáticas" posibles, pero sólo un reducido conjunto semi aleatorio (ya que no son restricciones puramente aleatorias, es un conjunto determinado por los diseñadores) van a experimentarse en la partida hace que aumente la rejugabilidad del juego, incentivando al jugador a pasárselo siendo muy difícil que haya dos partidas iguales.

Ritmo dramático adaptativo

De nuevo aparece el término adaptativo, en este caso lo que se adapta es el ritmo dramático, que como vimos al principio del capítulo 2, está asociado con la dificultad. Left 4 Dead genera mediante el director de IA, picos de tensión

en forma de oleadas de enemigos de manera semi aleatoria (de nuevo), es decir, se producirán dichos picos en diferentes momentos y situaciones.

Para reaccionar a las acciones del equipo de supervivientes y adaptar la generación de enemigos de modo que los jugadores no se sientan superados por la dificultad, se calcula "intensidad emocional" del grupo de supervivientes. Detallaremos el proceso a continuación.

Algoritmo 8 Adaptación del ritmo dramático

Entrada: Estado del juego GS
Salida: Población de infectados modificada

1: Sea $EI_i = estimar-intensidad-emocional(GS, i)$, la intensidad emocional del superviviente i
2: Monitorizar la intensidad máxima de los 4 supervivientes
3: **si** la intensidad estimada es demasiado alta **entonces**
4: Quitar amenazas durante un tiempo
5: **si no**
6: Crear una serie de amenzas (infectados + hordas + infectados especiales).
7: **fin si**

Podemos apreciar en el pseudocódigo 8 como la estimación de la intensidad emocional se realiza mediante una función (al igual que la heurística de reto) que devuelve un valor. En este valor influyen los siguientes factores:

- Las heridas producidas por los infectados, aumentan la intensidad de manera propocional al daño recibido.
- Cuando uno de los jugadores queda incapacitado, es otro factor que hace aumentar la intensidad.
- Cuando un infectado muere cerca de un superviviente, influye en un aumento de la intensidad inversamente proporcional a la distancia de éste. Es decir, muchos infectados muertos cerca implicará un aumento considerable de la intensidad.
- El tiempo hace que la intensidad descienda progresivamente.
- Si hay infectados enfrentándose activamente contra el grupo de supervivientes, mantener la intensidad.

A partir de esta intensidad se modula la aparición de infectados, exceptuando los infectados jefe cuya aparición no está influida por ninguna fómula adaptativa.

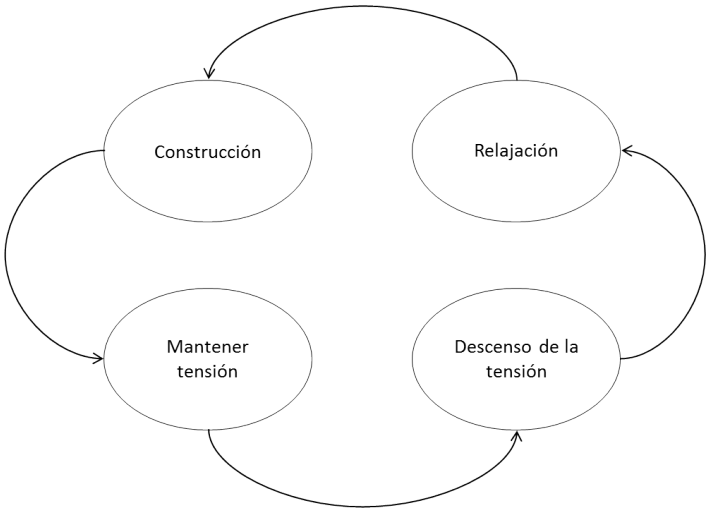


Figura 4.5: Los cuatro estados que usa el director de IA de L4D para adaptar el ritmo de la partida

Esta modulación está implementada como una máquina de estados finita (figura 4.5):

- **Construcción.** Va creando oleadas de amenzas o enemigos hasta que la intensidad de los supervivientes supere un umbral máximo. Estas oleadas consisten en infectados, ordas de infectados o infectados especiales
- **Mantener la tensión.** Continúa creando infectados durante unos 3-5 segundos una vez se ha alcanzado el pico de intensidad.
- **Descenso de la tensión.** Cambiamos a la generación mínima de amenazas y monitorizamos a los supervivientes hasta que la intesidad baja de cierto rango mínimo. Este estado es usado como una espera para que el combate no acabe solapando por completo el periodo de relajación. Este estado no transitará al de relajación hasta que no se produzca esa bajada de tensión.
- **Relajación.** Mantiene la población de infectados mínima durante 30-45 segundos o hasta que los supervivientes estén lo suficientemente lejos de la zona. No más infectados, ordas ni infectados especiales (salvo que ya los haya).

A continuación aparecen una serie de figuras que muestran gráficamente funcionamiento de la modulación de enemigos. Se comienza con una generación procedural, que describiremos con algo más de detalle en la siguiente sección, al

inicio del nivel (figura 2.2) en las zonas más cercanas y como según la intensidad de los jugadores ésta se va modificando (figuras 2.2 y 2.2)

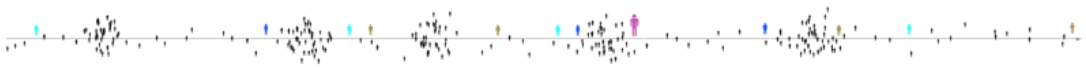


Figura 4.6: Una población de infectados generada proceduralmente por el juego

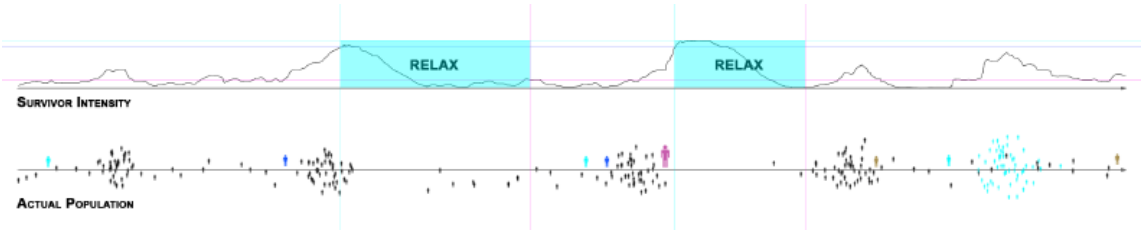


Figura 4.7: Modulación de la población de infectados por parte del director de IA basándose en la intensidad emocional.

2.3. Generación Procedural del Entorno

Left 4 Dead cuenta con un sistema de generación procedural de elementos que si bien no es la técnica más moderna que existe, es bastante efectiva. Ya mencionamos el concepto de impredecibilidad estructurada y es en esto en lo que se basa, teniendo en su haber los siguientes tipos de elementos:

- **Infectados comunes o merodeadores.** El infectado más común, simplemente merodea o permanece en un sitio hasta ser alertado por un superviviente.
- **Hordas.** Grupos de 20-30 infectados comunes que se abalanzan sobre el grupo en momentos inesperados.
- **Infectados especiales.** Infectados con habilidades especiales que atacan peródicamente.
- **Armas.** Armas y munición.
- **Items.** Items de un sólo uso como granadas, botiquines o pildoras

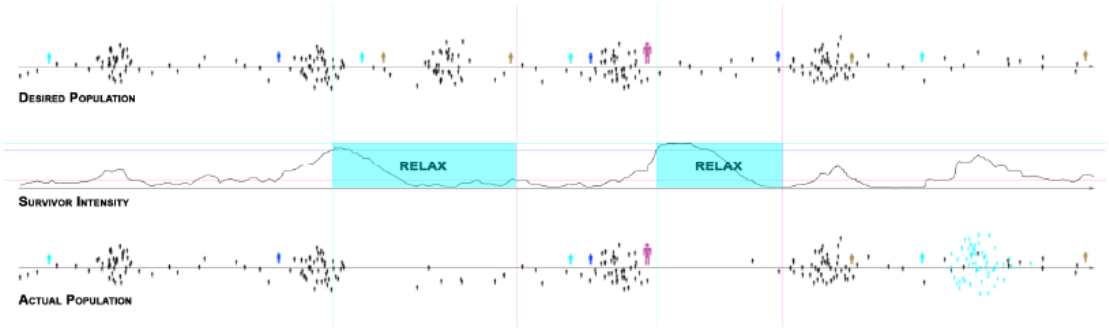


Figura 4.8: Comparación de la población de infectados generada procedural-mente y como se va modulando conforme la intensidad emocional. Nótese la desaparición de un buen número de infectados durante el primer periodo de relajación y el aumento de éstos (en cyan) al final de la gráfica con motivo de aumentar la intensidad emocional.

Aunque también hay un problema añadido, más enfocado a la capacidad o escalabilidad ¿cómo poblamos un mundo con cientos de enemigos? La solución por la que se optó en L4D es en reutilizar un número determinado de entidades, apoyándose en cierto razonamiento espacial. Este razonamiento se basa en sólo generar los elementos de las zonas de los alrededores del equipo de supervivientes, pero que no sean visibles directamente por ellos, en lo que ellos llaman el *Conjunto de Áreas Activas* ().

El conjunto de áreas activas se refiere a las áreas de la malla de navegación (*Navmesh*) que rodean a los supervivientes. El director de IA crea y destruye infectados conforme las áreas activas se mueven con los supervivientes.

Generando infectados

Los infectado se generan a partir de un número N que tiene asociado cada área y que se determina de manera aleatoria al empezar o reiniciar un mapa. Cuando un área entra en el conjunto de activas se intentan crear N infectados. Cuando un área sale del conjunto de activas o se requieren más infectados, lo infectados de éstas son reutilizados.

N pasará a valer 0 cuando un área sea completamente visible o cuando el director esté en el estado de Relajación.

Generando hordas

Durante intervalos “semi” aleatorios, de entre 90 y 180 segundos se crean

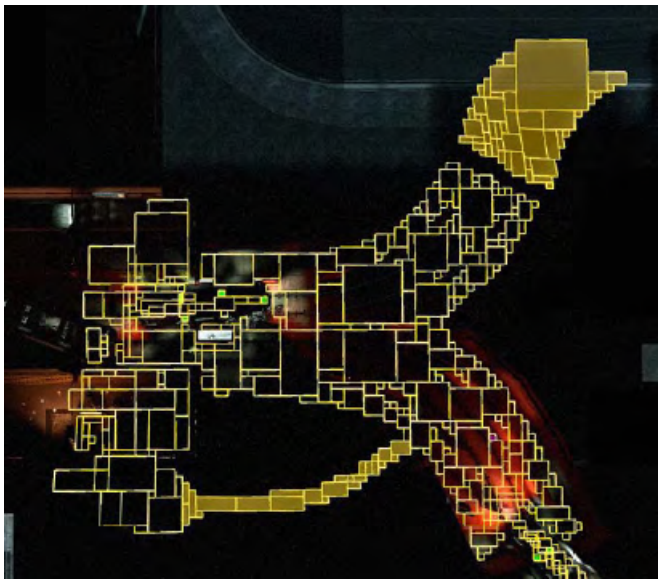


Figura 4.9: Conjunto de áreas activas durante un momento de la partida. Las zonas oscuras marcan las inactivas.

varias hordas. No obstante, un evento de un infectado especial concreto puede resetear el intervalo y forzar la generación de hordas de manera instantánea. El tamaño de las hordas varía desde el mínimo justo nada más generar una, al máximo tras un tiempo para equilibrar la dificultad de las sucesivas hordas durante. Durante el estado de relajación no se generan hordas.

El lugar dónde las hordas se generan es en las áreas activas de detrás de los supervivientes o en las cercanas al superviviente afectado por el infectado especial que resetea los intervalos. En caso de no existir ningún area disponible de éstas, se usa cualquiera área activa.

Generando infectados especiales

Los infectados jefes se crean de manera individual durante intervalos aleatorios y aparecen en cualquier área válida que no haya sido vista por los supervivientes (las áreas despejadas tampoco cuentan). Cada infectado de estos tiene una serie de preferencias de áreas, por ejemplo, uno de ellos preferirá aquellas que estén arriba del grupo.

Generando infectados jefes

Los jefes se crean cada N unidades, +/- una cantidad aleatoria, a través de la ruta de escape, además de ciertos puntos concretos de ésta. Hay dos tipos

de jefes, aunque para aleatorizar más la generación se usa un tercero ficticio, "Ninguno". La creación de estos se hace de forma que no se repitan dos jefes del mismo tipo seguidos (figura 4.10)

Figura 4.10: Generación de los dos tipos de jefes del juego.

Generando Items

La generación de items, no resulta ser tal ya que todos los elementos son puestos manualmente por los diseñadores de niveles. Lo que hace el director es seleccionar cuales de ellos existen realmente en la partida. La razón por la que este proceso es el más manual de todos es para aumentar la carga narrativa, para beneficiar al jugador (al es más predecible puede facilitarle la tarea al jugador) y para evitar artefactos gráficos. Es decir, para que no aparezcan armás atravesando una pared o en posiciones extrañas (items encima de otros).

3. Conclusiones

Tras analizar con detenemiento el juego, se puede llegar a la conclusión de que L4D es un paso más allá en la evolución del secuenciador de Pfeifer. Es un DDA orientado al sistema (y pasivo según Chen) basado en el ajuste del contenido del nivel, con más restricciones a contemplar y una mayor aplicación del contenido procedural. Esta generación de contenido procedural resulta un elemento clave en el éxito de la aplicación del director de IA (¿Sería posible de alguna manera llevarlo a generos que no sean juegos de acción?).

En conjunto, el director de IA se compone de algoritmos simples, incluso su heurística de reto también lo es, pero que en un aspecto global, operando de forma cooperativa, consiguen ser muy eficaces (podríamos hablar de emergencia) en su objetivo de gestionar el ritmo de juego y la dificultad.

Capítulo 5

Sistema propuesto

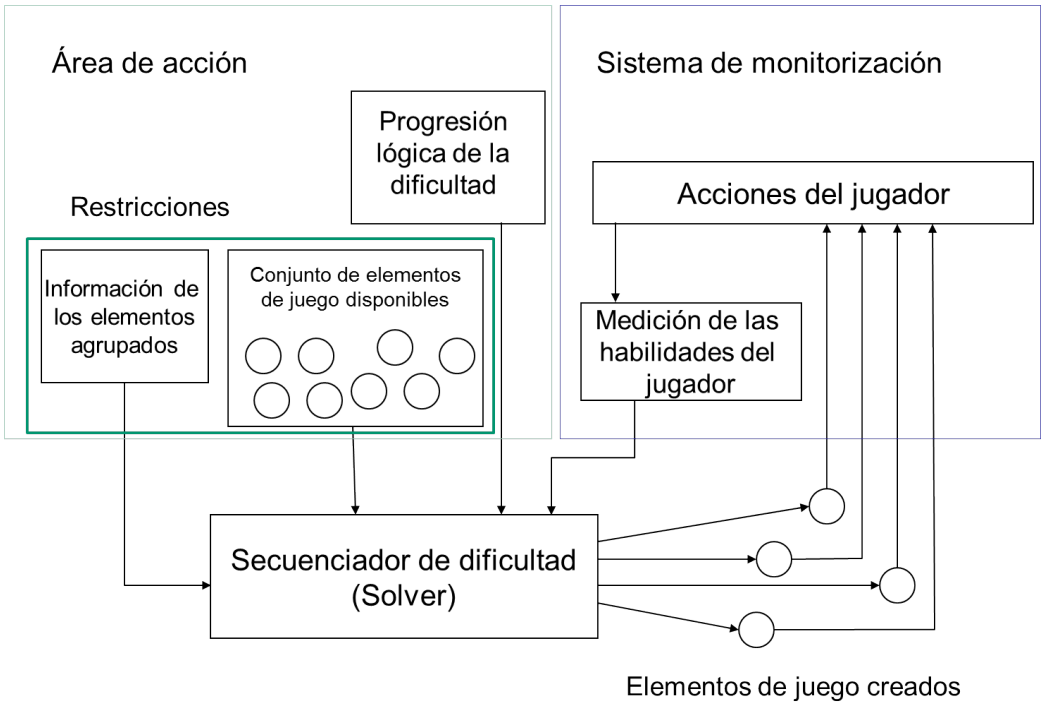
En este capítulo introduciremos una pequeña vuelta de tuerca a los DDA basados en el ajuste del contenido de nivel, tratando de modelizar este proceso como un problema de satisfacción de restricciones. Basándonos en el secuenciador de Pfeifer y la parte del director de IA referente al ajuste de la dificultad y ritmo, propondremos una nueva forma de selección de elementos de juego a aparecer durante la partida.

1. Componentes del sistema de ajuste dinámico de la dificultad

Se trata de una extensión del secuenciador de Pfeifer visto en el capítulo 2 (3.3.1), un DDA de ajuste de contenido de nivel (3.3) y orientado al sistema (1.1). Está pensado para juegos de acción lineales, en los que el jugador tiene que llegar de un punto A (inicial) a otro B (final) del nivel.

Comencemos por una descripción a alto nivel del sistema (figura 1) y comentemos sus diferencias:

- **Area de acción.** Se trata de una discretización del nivel en áreas o zonas. Cada zona está compuesta por los elementos que pueden aparecer en ésta, la información (restricciones) y la dificultad correspondiente.
- **Restricciones** a cumplir por los elementos que el DDA elija para poner en esa zona.



- **Progresión lógica de la dificultad** o **especificación**. Describe el estado del juego ideal,medio o normal del jugador en esa zona.
- **Sistema de monitorización.** Es el encargado de hacer un seguimiento del comportamiento del jugador y el estado del juego.
- **Secuenciador de dificultad.** Es el que se encarga de resolver el problema de satisfacción de restricciones propuesto por el área de acción y el sistema de monitorización. Es aquí dónde introduciremos nuestra pequeña propuesta.

1.1. Discretización del juego

Una parte fundamental de este sistema es determinar cuáles son las zonas en las que pueden o no aparecer objetos. En caso de tener ya disponer de una discretización hecha por la IA (como puede ser la *navmesh* utilizada para la búsqueda de caminos) propia del juego podemos reutilizarla y añadirle una capa de abstracción más (puede incluso que esta abstracción sea algún tipo de clústering) para razonar. La ventaja de disponer de este tipo de discreticación es que no tampoco tenemos que preocuparnos en exceso por los puntos de aparición de los elementos (*spawn points*). En caso de no disponer de esta discretización necesitaremos al menos alguna manera de representar el nivel como un grafo,

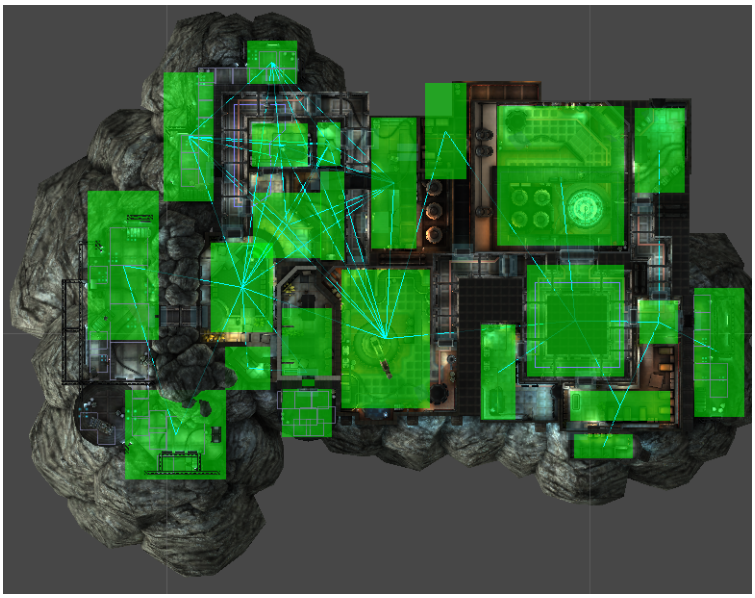


Figura 5.1: Vista cenital de la discretización del nivel en áreas. Los cuadrados verdes representan las áreas de acción. Las líneas cian son las conexiones entre ellas.

por lo que manualmente definiríamos nosotros las áreas y quizás también los puntos en donde aparecerían los elementos.

El nivel de detalle de este grafo será bastante dependiente del juego. En nuestro caso, para una primera implementación sobre juego simple de prueba, nuestra discretización ha sido bastante vaga. Hemos definido manualmente las áreas mediante (*bounding volumes*) y sus conexiones. La figura 5.1 muestra una perspectiva cenital de las áreas del nivel que se correspondería con el siguiente grafo de la figura 5.2.

1.2. Definición de restricciones

Una vez tenemos discretizado el nivel en zonas, es el momento de añadir las restricciones sobre ésta. La idea de esto es modelizar las diferentes zonas como un problema de satisfacción de restricciones a resolver por el DDA.

Siguiendo las ideas Pfeifer y L4D, estas restricciones se basarán principalmente en el tipo de enemigos a poder aparecer, su número máximo o mínimo. Es importante hacer hincapie en esto último ya que la técnica de DDA propuesta se basa en que tenemos almenos una función de optimización. Junto estas aparecerán otras restricciones más de conceptos o estéticas que implique

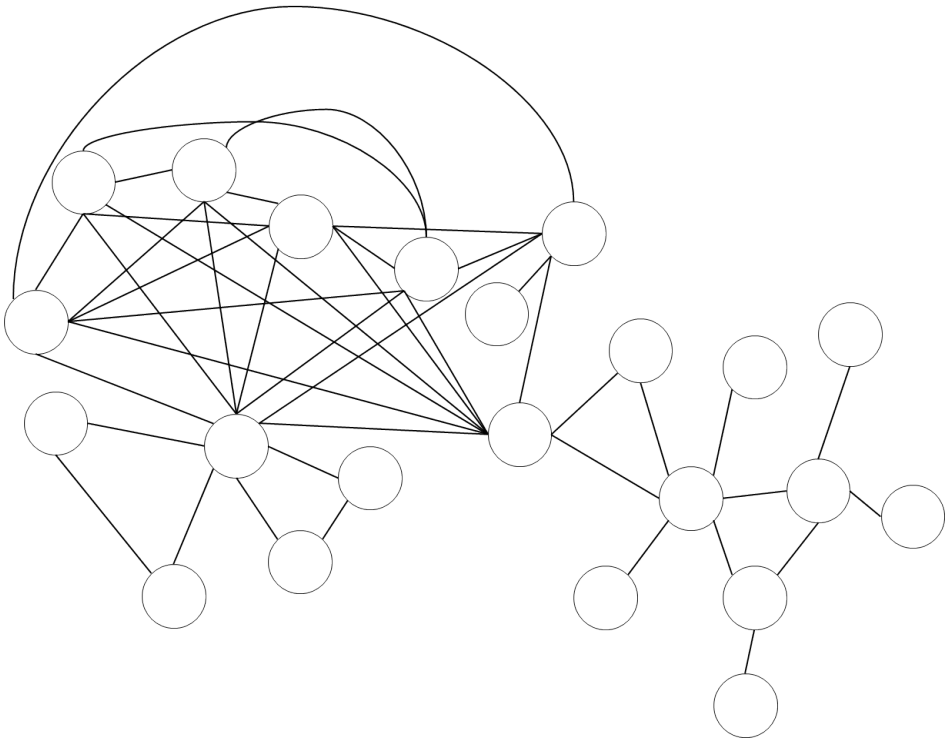


Figura 5.2: Grafo no dirigido correspondiente a las áreas de un nivel

la aparición de cierto enemigo determinado o si aparece un enemigo A tenga que aparecer otro B.

En nuestro juego de ejemplo las restricciones de cada área están relacionadas con mostrar un número de enemigos determinados de forma que el poder de los enemigos no exceda el número devuelto por la heurística de reto. Las restricciones principal sobre la que se apoya el DDA son las siguientes:

$$\text{máx} \sum_{i \in E} v_i x_i$$

$$\sum_{i \in E} w_i x_i \leq C, x_i \in \{0, 1, \ldots, c_i\}$$

Dónde E representa el número de elementos disponibles para ese área y C el valor devuelto por la función de reto (puede que escalado de alguna forma), resultando ser una versión del problema de la mochila acotado donde el valor de un objeto es igual a su peso (v_i=w_i, también conocido como *Subset-sum*). Es importante resaltar que, a pesar de que estamos ante un problema NP-Completo, el tiempo de ejecución no resulta excesivamente problemático

debido a que tenemos un conjunto finito de elementos que no es muy numeroso (posiblemente no lleguen a la veintena).

Como también es posible avanzar en el juego sin enfrentarse apenas a enemigos, vamos a poner una restricción extra:

$$|E| \geq N$$

Que empezará a añadir enemigos extra hasta llegar al número determinado por N con el fin de que el jugador deje de realizar evasivas. La idea es poblar el área de tantos elementos que le resulte muy dificil evitar no hacer frente a los enemigos. Esta restricción se ejecutaría una vez resuelta la de asignación de enemigos para no entrar en conflicto con ésta.

1.3. Definición de especificaciones

Para lo que Pfeifer era un factor, ahora ha pasado ser un estado del juego con el que comparar el actual realizado por el jugador, esto es, una especificación. El estado del juego σ en un instante t podemos definirlo como una N-tupla $\sigma_t = \langle v_1, v_2, \ldots, v_n \rangle$ dónde cada uno de los v_i representa el valor de una de las variables del juego que estamos midiendo (mediante el sistema de monitorización).

En nuestro juego de ejemplo hemos reducido la representación del estado de juego a $e_t = \langle h(t) \rangle$ donde $h(t)$ es la salud del jugador en el instante t.

Normalmente, para definir los valores adecuados de una especificación hacen falta sesiones de juego en los que recolectar estadísticas, por lo que para ello es necesario algún tipo de monitorización. A partir de estas estadísticas podemos definir cuál es el estadio medio del jugador en un punto del juego y ver si es acorde con nuestra especificación inicial o necesitamos balancear más el juego[1] (figura 5.3).

Como ya hemos dicho este DDA está pensado para juegos lineales, en los que asumimos que hay ciertos puntos de acción en el juego, lo que hemos llamado áreas o zonas de acción. Es en estos puntos dónde definiremos el estado ideal (especificación) el cual nos servirá de guía en la medición de las habilidades del jugador. De momento, por simplicidad y para que se entienda mejor el concepto, asumiremos que el estado ideal o especificación que pongamos en un área de acción es la que se corresponde con el estado de un jugador "normal".

[1] `http://aigamedev.com/premium/article/procedural-director/#playtesting`

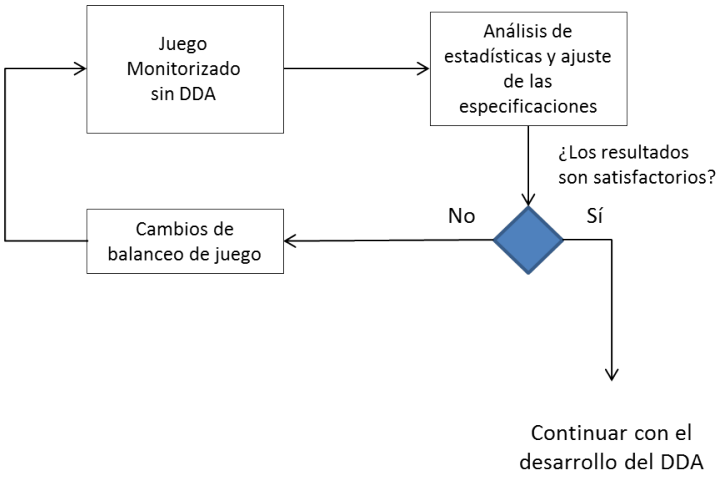


Figura 5.3: Flujo de sesiones playtesting para balanceo y definición de especificaciones

Para hacer una analogía con la implementación de Pfeifer si queremos definir una zona difícil, en lugar de un factor, definimos una especificación en el que el estado ideal corresponda con un estado de juego en el que el jugador se encuentre en un estado crítico (ya sea de vida, munición o ambas). En nuestro caso práctico dónde el estado de la partida se ha basado simplemente en la salud del personaje, bástaría con definir una especificación con la salud baja.

1.4. Sistema de monitorización

Se trata de hacer un seguimiento del estado de la partida ya sea de forma "continua" (tras un intervalo de tiempo) o por eventos (al llegar a determinados puntos de control) o ambos.

El sistema de monitorización puede usarse igualmente independientemente de implementar ningún tipo de ajuste de la dificultad simplemente para obtener datos de juego y realizar análisis de éstos, es dependiente de la aplicación. La relación inversa no se cumple, es decir, es fundamental para la implementación del DDA, y si dentro de este sistema de monitorización también incluimos estadísticas del juego, resulta un elemento imprescindible para cualquier tipo de sistema adaptativo.

Otro de los componentes que forma parte del sistema de monitorización que resulta imprescindible, el más importante y más complicado, para un sistema

de DDA, es la heurística o función de reto. Esta función

$$h(\sigma_t) = N$$

a partir del estado actual del juego σ_t nos devuelve un número N (normalmente entre 0 y 1) que estima cómo de bien está haciéndolo un jugador. También es posible (de hecho es lo recomendable [1]) que esta estimación no sólo se base en el estado actual sino que también se apoye en una serie de estados anteriores $\langle \sigma_t, \sigma_{t-1}, \ldots, \sigma_{t-n} \rangle$.

1.5. El secuenciador de dificultad o Solver

El núcleo principal de nuestro DDA propuesto es un algoritmo de resolución de problemas de satisfacción de restricciones (*CSP*) como $AC3$, puesto que como mencionamos en la sección de restricciones solemos tener un dominio finito, muy reducido y con bastantes soluciones no es necesario disponer de uno muy avanzado.

El uso de algoritmos de CSP aplicado en diseño de niveles es una técnica novedosa cuya pionera es Gillian Smith [14]. No obstante, para esta tarea al realizarse en tiempo de diseño, se podían usar algoritmos independientes sin ninguna integración en el motor o en el juego. Para implementar el sistema propuesto es necesario disponer de un solver integrado en el juego, que pueda ejecutarse y continuar la búsqueda durante varios ciclos de éste en la partida (esto puede no ser necesario si las búsquedas pueden realizarse fácilmente en un sólo ciclo).

Ejemplo

Ilustremos el proceso con una sencilla modelización de ejemplo. Supongamos que un área de acción tiene los siguientes elementos disponibles:

- 3 enemigos de tipo A. Los enemigos tipo A tiene un peso o dificultad d_A que en nuestro caso está representado por la vida de éste
- 2 enemigos de tipo B. Los enemigos tipo B tienen un peso o dificultad d_B que en nuestro casp está representado por la vida de éste.

Supongamos que no hay más restricciones y que el valor de la función de reto devuelto con respecto a la especificación del área es C.

Las variables y restricciones a pasar a nuestro solver son:

$$x_a \in \{0, 1, 2, 3\}$$
$$x_b \in \{0, 1, 2\}$$
$$R1 : \text{máx}\, x_a d_a + x_b d_b$$

$R1$ Es nuestra función objetivo a optimizar.

$$R2 : x_a d_a + x_b d_b \leq C$$

Tan sólo nos queda llamar a nuestro algoritmo de resolución con los parámetros:

$$ac3(V, D, R)$$

Dónde $V = \{x_a, x_b\}$ es el conjunto de variables, $D(x) = \{x_a : \{0, 1, 2, 3\}, x_b : \{0, 1, 2\}\}$ son los dominios de las variables y $R = \{R1, R2\}$ las restricciones.

Si quisiéramos forzar a que apareciese un número mínimo de enemigos tipo A, n_a obligatorio, bastaría con añadir la restricción:

$$R3 : x_a \geq n_a$$

Observaciones

La pregunta inmediate que el lector puede hacerse es qué ocurrirá en caso de fallo. Es necesario aclarar que la asignación de enemigos no fallará nunca debido a que siempre podemos no generar ninguno, el problema vendrá cuando restricciones "extra" a la optimización generén conflictos que hagan el sistema irresoluble.

La solución propuesta para ello se basa en realizar la satisfacción de restricciones en dos pasos:

1. Aplicar las restricciones que "extras" (la que no es la función objetivo a optimizar).
2. Aplicar Sobre las soluciones anteriores la solución al problema de la mochila.

Pero con esto hemos trasladado el problema ¿qué ocurre si el primer paso no encuentra solución? En ese caso, proponemos dejar que el algoritmo resuelva el problema de la mochila sin tener en cuenta las restricciones anteriormente aplicadas.

2. Implementación

Con el objetivo de probar el funcionamiento de esta técnica hemos implementado un sistema de ajuste dinámico de la dificultad sobre un juego acción ya desarrollado y bastante simplificado como es el *Angry Bots* de Unity[2]. El Desarrollo consta de dos partes: una de recogida de estadísticas de juego y otra de generación del contenido del nivel en base a éstas.

La metodología aplicada ha sido la siguiente [3]:

Definición de variables a medir y recopilación de estadísticas

A la hora de recoger estadísticas o definir una especificación lo primero que cabe preguntarse es qué datos o variables necesitamos o vamos a medir. Partiendo de una primera toma de contacto hemos tratado de simplificar las siguientes estadísticas de juego básicas, que hemos ido almacenando durante una semana:

- El número de muertes.
- El número de enemigos vencidos.
- El número de items usados. Hemos modificado ligeramente el juego para incluir este objeto.
- El número de áreas visitadas y completadas.

Que nos servirán para definir restricciones extras en el juego. Con respecto al estado del juego para la definición de especificaciones[4]:

- La salud del jugador al visitar y completar las zonas de acción

[2] `https://www.assetstore.unity3d.com/#/content/12175`

[3] Ni que decir tiene que esto realmente es un proceso iterativo, que por restricciones de tiempo lo hemos reducido a una sola iteración

[4] Es necesario aclarar que las estadísticas de juego también pueden servir para definir especificaciones

- El orden de las áreas visitadas y completadas.

Como hemos partido de un juego ya hecho, no tenemos muchos opción a balancear. Por lo que hemos optado por definir las especificaciones a partir de los resultados de las estadísticas.

Definición de la heurística de reto

A partir de las estadísticas hemos tratado de obtener una fórmula que nos sirva para estimar la actuación del jugador. El hecho de disponer de poco tiempo y de tratar de tener un primer prototipo del sistema para analizar, ha hecho que optemos por reducir la heurística de reto a la propia salud del jugador en el juego.

La heurística de reto normaliza la salud del jugador (la divide entre la salud máxima), llamemos a este número $\hat{h}$. Este número lo escalamos conforme a la especificación del área de la siguiente manera:

1. Calculamos el reto C_{area_i} que le corresponde a la especificación e_{area_i} del área en el que se encuentra. Se trata de a partir de la población inicial generada (antes de realizar cualquier ajuste), sumar los pesos de los enemigos que se encuentran en ésta y establecerlo como el reto de la especificicación.

2. A partir del número anterior calculamos el nuevo reto C mediante una regla de tres con e_{area_i} ($\hat{h}_{area_i}$) y C_{area_i}, resultándonos el valor a utilizar como restricción del problema de la mochila.

Implementación del DDA

Una vez tenemos las especificaciones y la función de reto queda implementar la adaptación o ajuste de dificultad para "validar" los resultados de las dos etapas anteriores. Como ya hemos dicho y aunque no ha sido el caso en este trabajo debido a limitaciones de tiempo, todo este proceso suele ser iterativo y testeado con muchas sesiones de juego.

Las particularidades más destacadas a la hora de implementar el sistema propuesto en AngryBots las detallaremos a continuación.

Figura 5.4: Máquina de estados finita de un área de acción

2.1. Peculiaridades de la implementación

2.1.1. Áreas de acción

Las áreas de acción han sido modeladas con una máquina finita de estados compuesta por tres estados (figura 5.4)

- **Estado Inicial**. Al comienzo del nivel el área no ha sido ni visitada, ni completada por el usuario. Genera una población inicial de elementos conforme a la especificación.

- **Visitado**. El conjunto de elementos del área queda fijado y ya no será modificado más. En un principio se planteó la posibilidad de sólo poder aumentar el número de elementos del área(para no generar inconsistencias), pero por simplicidad se ha optado por no implementarlo y dejarlo como una futura ampliación.

- **Completado**. Se han consumido todos los elementos del área y ésta da por completada. Se realiza una propagación de esto entre sus vecinos con el objetivo de modificar la población de aquellas zonas cuyo estado aún es el inicial (no visitadas).

De una forma parecida al L4D, se ha optado por generar una población de elementos del nivel "ideal" conforme a las especificaciones e ir modificando ésta conforme el transcurso de la partida. El algoritmo 9 describe a grandes rasgos este proceso.

La idea es propagar la actualización. Como las áreas completadas ya no influyen, simplemente dejan pasar la actualización hasta llegar a un área no visitada (la cual se actualizará), o a un área visitada (la cual no hará nada para evitar inconsistencias).

Algoritmo 9 Propagación de del área completada

Entrada: Un estado del juego E, una área A completada y el grafo de áreas de acción.

Salida: Adaptación de la nueva población de elementos.

```
1: Sea Q una cola dónde almacenamos las áreas que quedan pendientes de
   actualizar.
2: Sea V una lista dónde almacenamos las zonas ya visitadas.
3: Añadimos A a Q
4: mientras Q no esté vacía hacer
5:    Hacer a el primer elemento de Q
6:    Poner a en V
7:    si estado de a = Estado Inicial entonces
8:       Actualizar población de a
9:    si no
10:      si estado de a = Completado entonces
11:         expandir(a)
12:      si no
13:         Continuar
14:      fin si
15:   fin si
16: fin mientras
```

Algoritmo 10 Expandir área

```
1: para todo área vecina n de a hacer
2:    si n no ha sido ya visitada entonces
3:       Añadir a Q
4:    fin si
5: fin para
```

Para los puntos de aparición de los elementos hemos optado por la vía más simple que es definir manualmente una cantidad determinada (la suficiente para dar variedad al juego) ya que de por sí el juego no traía ningún tipo de discretización del espacio. La generación automática de estos puntos sería muy específica para este juego y a pesar de que no es una tarea excesivamente compleja, se deja como punto de ampliación. Otro punto de ampliación más complejo, que requeriría modificar parte del núcleo del juego, sería la discretización automática del espacio (ya sea navmesh o celdas o *waypoints*) que conllevaría que la definicón de áreas de acción fuesen mucho más precisas.

Figura 5.5: Editor de las restricciones de un área de acción

Restricciones

Las restricciones las hemos implementado en una sola clase. La hemos restringido al tipo de enemigos disponibles junto a su número máximo, y el número mínimo de elementos a usar. Que es una restricción extra usada al final del juego para forzar al jugador a enfrentarse o vencer a almenos un determinado número de enemigos.

Esta clase hace de interfaz o puente entre restricciones del juego y restricciones del solver. Es decir, se encarga de trasladar los parámetros establecidos para el nivel del juego como restricciones de un CSP. La figura 5.5 muestra un ejemplo como se introducen las restricciones en tiempo de diseño en el nivel.

Como punto de extensión se deja introducir más restricciones al sistema y refactorizar este adaptador[5] para que la creación de futuras restricciones no sea tan artesanal.

Secuenciador de dificultad o solver

Se trata de una encapsulación de la librería de CSP a la cual se ha extendido para poder pausar y reanudar la búsqueda de soluciones. Se ha añadido también una implementación del problema de la mochila, usando programación dinámica para acelerar las búsquedas y con la capacidad de adaptar el tamaño, esto último con motivo de una futura extensión para usar más restricciones como paso previo. No obstante para el juego de ejemplo no es necesario su uso ya que no hay ningún problema de rendimiento.

Para garantizar una mayor variedad de generación de enemigos en caso de existir más de una solución óptima, elegimos aleatoriamente una entre ellas. No obstante, aclaremos que generaremos un número máximo de soluciones deter-

[5] `http://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)`

minadas en tiempo de diseño, no todas las posibles. Como punto de extensión a la librería quedaría determinar si es posible integrar algún tipo de restricción o variables que se solucionen mediante programación dinámica, en lugar de tener éste como un módulo aparte.

La implementación del secuenciador, trata la generación de enemigos en dos pasos:

1. Generación de N soluciones que satisfagan las restricciones previas.
2. Generación de enemigos conforme a una de las soluciones anteriores. En este paso también hemos implementado la restricción "extra" de crear al menos un total de enemigos E independientemente del resultado de la estimación del reto, la cual simplemente es una generación indicriminada de enemigos hasta alcanzar ese número.

El hecho de no haber usado más restricciones hacen que la primera búsqueda de soluciones sea innecesaria. De hecho y como crítica al estado en el que se encuentra la implementación, es posible poder prescindir del solver debido a que no hay más restricciones que la de asignación de elementos. El uso de un mayor número de éstas, las cuales se dejan como posible extensión, daría más sentido y utilidad al sistema propuesto.

Capítulo 6

Conclusiones y trabajos futuros

1. Conclusiones

Hemos comenzado la memoria introduciendo las técnicas básicas de adaptabilidad en videojuegos, donde hemos visto que las más usadas son las que se basan en el uso de un conjunto de estadísticas o de ejemplos. Es lo que se conoce como aprendizaje indirecto, la forma más común (y probablemente seguirá siéndolo) de implementar aprendizaje en videojuegos.

Tras esto hemos introducido el concepto de ajuste dinámico de dificultad con las diferentes técnicas investigadas. Es importante saber que existen diversas formas de atacar este problema, en la memoría nos hemos centrado en una, que es el ajuste de la dificultad mediante la modificación del contenido de nivel, más en auge con la importancia que está cobrando la generación de contenido de manera procedural. Dentro de esta categoría hemos decidido proponer una extensión al secuenciador de Pfeifer el cual también es extendido de alguna forma como parte del Director de IA de Left 4 Dead.

El sistema DDA presentado trata de modelizar el ajuste de la dificultad mediante el contenido de nivel como un problema de satisfacción de restricciones (mayormente restricciones de programación lineal). La principal novedad implementada es crear los elementos del juego resolviendo el problema de la mochila lo que permite una generación de elementos más óptimas que la elección del secuenciador de Pfeifer y más complejas que la generación de enemigos de Left 4 Dead.

La ventaja que ofrece usar un CSP es que permite implementar fácilmente (al menos en teoría) todas las extensiones propuestas por Pfeifer en su secuen-

ciador (como es el uso de conceptos) y que ya se usan en el diseño de niveles. También debería ser posible simular la adaptación de la dificultad de Left 4 Ddead mediante este sistema y en general cualquier juego de acción lineal.

Se ha implementado con éxito el sistema en un juego de ejemplo dónde hemos conseguido ir modificando la población de los elementos del juego conforme al transcurso de la partida sin experimentar ningún tipo de problema de *frame-rate*. No obstante, y debido a restricciones de tiempo en el desarrollo, no se han podido introducir restricciones más avanzadas, ni el uso de más de un estado (es decir, tener en cuenta el historial, cosa que es muy recomendable) a la hora de generar modificar el contenido. Tampoco hemos tratado el diseño de especificaciones y heurísticas de reto más avanzadas (siendo esto último muy dependiente del juego). Todo ello se propone como posible punto de extensión.

Para terminar una serie de observaciones finales que cabe destacar a partir del resultado obtenido.

- Es muy importante realizar ciclos contínuos de pruebas de juego para balancear y obtener datos más fiables, dónde las estadísticas juegan un papel muy importante. Estos ciclos requieren bastante tiempo.

- Tratar de implementar un sistema de dificultad adaptativo partiendo de un juego que no ha partido de esa idea seguramente conllevará una refactorización o reimplementación de mucha lógica del juego. También requerirá una fase extra de balanceo, y la formulación de una buena herística se hará más difícil.

- A pesar de que los resultados son prometedores, se trata todavía de una implementación demasiado *näive* que requiere un mayor estudio para entender sus verdaderas posibilidades.

2. Trabajos futuros

A lo largo del capítulo anterior hemos ido indicando posibles punto de extensión del sistema, tanto para el juego de ejemplo, como otros avances que conllevan usar otro juego algo más complejo como pruebas. Una de las conclusiones extraídas del trabajo, es que la implementación de este sistema (y como hemos visto a lo largo de la memoria, de cualquiera) requiere tiempo por lo que muchas de las propuestas están relacionadas con invertir tiempo en el refinamiento de éste. En general en:

- **Construcción de niveles más pequeños para realizar pruebas**. Un inconveniente encontrado en los datos recogidos en la fase de recopilación de estadísticas han sido la escasa cantidad de estos. Independientemente del número de personas que lo han jugado (una docena), sólo tres han terminado la partida. Lo cual indica que el nviel elegido era excesivamente largo para una recogida de estadísticas de prueba.

  Un ejemplo sería un nivel con cuatro áreas, más corto, rápido y sencillo de medir. No sólo eso, se podrían usar niveles cuyo grafo de área de acción fuese completo (todos los nodos están conectados con todos) para realizar pruebas de estrés o para analizar el comportamiento del sistema.

- **Generación automática de las áreas**. En principio se trataría de obtener la superficie andable del nivel bien para este juego o bien de forma más genérica. Esto en sí es un problema no muy complejo (existen ya librerías de generación automática de las superficies andables) que tendría como dificultad modificar los comportamientos de la IA enemiga para hacer uso de este "suelo".

  Una vez tuviésemos este sistema, las áreas podrían definirse automáticamente, así como los puntos de aparición, quedando por establecer las restricciones y especificaciones y dejando de ser todo tan manual (ya que carece de sentido que el tiempo a realizar el diseño de un nivel aumente sólo por este hecho). Además, partiríamos de la misma discreatización del espacio que en L4D con lo que podríamos tratar de comparar los sistemas de alguna forma.

- **Nuevas variables a medir**. Se trataría de realizar otro ciclo de recopilación de datos tratando de incluir nuevas variables que ayuden a la monitorización, a generar estados y especificaciones más descriptivas y a una nueva fórmula de estimación del reto. Este proceso podría involucrar un balanceo del juego (determinar la dificultad de los elementos) que podría servir de experiencia a la hora de llevar el sistema a juegos más complejos.

  Usando un juego algo más complejo sería necesario un sistema de monitorización más sofisticado y quizás herramientas de análisis estadístico más avanzadas. Sería interesante de aplicar para entender mejor su uso[1]

- **Añadir nuevos elementos de juego**. En relación al punto anterior puede ser interesante añadir nuevos elementos de juego tales como munición o escudos, lo que conllevaría a desarrollar nuevas variables y una nueva función de reto. Acércandose más a lo que sería un juego real.

[1] http://www.gamasutra.com/view/feature/3017/designers_notebook_positive_.php

- **Inclusión de nuevas restricciones**. Usar más restricciones puede dotar de más riqueza al sistema, a la vez que puede servirnos para determinar posibles problemas para juegos más complejos, ya sea por cuello de botella en tiempo de ejecución o por problemas propios de la implementación.

  Un ejemplo de regla nueva sería introducir botiquines en el área, siempre que el peso total de los enemigos generados sea superior a N. También se podrían introducir reglas para forzar al jugador a aprender "conceptos", como por ejemplo hacer establecer una restricción que fuese "cuerpo a cuerpo" y que obligase al sistema a sólo generar enemigos que tengan esa característica.

- **Especificaciones y función de reto** ¿Sería posible implementar la especificación y función de reto actual usando las ideas del *Hamlet* de Robin Hunicke[16]?

- **Iterar la implementación actual**. Existen ciertos puntos fácilmente mejorables en la implementación actual.

  - ¿Es posible integrar la resolución pseudopolinomial del algoritmo de la mochila en la libreria CSP?
  - Ahora mismo el proceso de satisfacción de restricciones consta de dos partes, restricciones previas y resolución del knapsack. Convendría poder dividir las restricciones entre pre-knapsack y post-knapsack. O simplemente integrarlo todo y ver si hay problemas en tiempo de ejecución.
  - Ser menos "reactivo" y modificar el contenido del nivel no sólo conforme al estado actual, también conforme a los estados anteriores.

- **Implementación del sistema presente durante el desarrollo de un juego**. Vendría a ser la prueba definitiva de que el sistema propuesto es realmente viable. En lugar de partir de un juego ya hecho y modificarlo, implementar el sistema conforme se desarrolla éste.

Bibliografía

[1] AIGAMEDEV.COM. Building your own ai director for fun and pacing `http://aigamedev.com/premium/tutorial/building-ai-directors/`.

[2] AIGAMEDEV.COM. The mechanics of influence mapping: Representation, algorithm & parameters`http://aigamedev.com/open/tutorial/influence-map-mechanics/`.

[3] AIGAMEDEV.COM. Mixing ai tools and authoring for progression design`http://aigamedev.com/premium/interview/mixed-initiative/`.

[4] ALLIS, L. V. *A knowledge-based approach of connect-four*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1988.

[5] ANDRADE, G., RAMALHO, G., SANTANA, H., AND CORRUBLE, V. Challenge-sensitive action selection: an application to game balancing. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (Washington, DC, USA, 2005), IAT '05, IEEE Computer Society, pp. 194–200.

[6] BOGOST, I., MATEAS, M., MURRAY, J., AND NITSCHE, M. Asking what is possible: the georgia tech approach to game research and education. *Int'l Digital Media and Arts Association J 2*, 1 (2005), 59–68.

[7] BOOTH, M. Replayable cooperative game design: Left 4 Dead. In *Game Developer's Conference* (2009).

[8] CHAMPANDARD, A. J. The darks art of neural networks. *AI Game Programming Wisdom* (2002), 640–651.

[9] CHANG, D. M. J. Dynamic difficulty adjustment in computer games.

[10] CHEN, J. Flow in games, mfa thesis. *USC Interactive Media. 2007b* (2009).

[11] Colton, S., and Pease, A. Curso de *Ludic Computing*, Imperial College, Londres `http://ccg.doc.ic.ac.uk/wiki/doku.php?id=ludic`. lectura 12, 2012.

[12] Evans, R. Varieties of learning. *AI Game Programming Wisdom* (2002), 567–578.

[13] González Sánchez, J. L. *Jugabilidad. Caracterización de la experiencia del jugador en videojuegos*. PhD thesis, Universidad de Granada, 2010.

[14] Horswill, I., and Foged, L. How to build a constraint propagator in a weekend, 2012.

[15] Hunicke, R. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology* (New York, NY, USA, 2005), ACE '05, ACM, pp. 429–433.

[16] Hunicke, R., and Chapman, V. Ai for dynamic difficulty adjustment in games. In *Challenges in Game Artificial Intelligence AAAI Workshop* (2004), pp. 91–96.

[17] Laurel, B. Computer as theatre: A dramatic theory of interactive experience, 1991.

[18] Li, B., Lee-Urban, S., Appling, D. S., and Riedl, M. O. Crowdsourcing narrative intelligence. *Advances in Cognitive Systems 2* (2012), 25–42.

[19] Manslow, J. Learning and adaptation. *AI Game Programming Wisdom* (2002), 557–566.

[20] Mateas, M., and Sengers, P. *Narrative Intelligence*. John Benjamins Publishing, 2003.

[21] Millington, I., and Funge, J. D. *Artificial Intelligence for Games*. Morgan Kaufmann/Elsevier, 2009.

[22] Missura, O., and Gaertner, T. Online adaptive agent for connect four. In *Proceedings of the Fourth International Conference on Games Research and Development CyberGames* (2008), pp. 1–8.

[23] Missura, O., and Gärtner, T. Predicting dynamic difficulty. In *Advances in Neural Information Processing Systems 24* (2011), J. Shawe–Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds., pp. 2007–2015.

[24] Mitchell, T. M. *Machine learning. WCB*. McGraw-Hill Boston, MA:, 1997.

[25] Orkin, J. D. *Collective artificial intelligence: simulated role-playing from crowdsourced data*. PhD thesis, Massachusetts Institute of Technology, 2013.

[26] Pfeifer, B. Ai to control pacing in games. In *Artificial Intelligence, Interactivity, and Immersive Environments: 2nd Annual Game Development Workshop, University of Texas at Austin.* (2003).

[27] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., and Edwards, D. D. *Artificial intelligence: a modern approach*, vol. 74. Prentice hall Englewood Cliffs, 1995.

[28] Shaker, N., Yannakakis, G. N., and Togelius, J. Towards automatic personalized content generation for platform games. In *AIIDE* (2010).

[29] Smith, H. `http://www.igda.org/articles/hsmith_future`, 2001.

[30] Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E. Online adaptation of game opponent ai in simulation and in practice. In *Proceedings of the 4th International Conference on Intelligent Games and Simulation* (2003), pp. 93–100.

[31] Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E. Difficulty scaling of game ai. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)* (2004), pp. 33–37.

[32] Stone, P. *Layered learning in multiagent systems: A winning approach to robotic soccer*. The MIT press, 2000.

[33] Sweetser, P. *Emergence in Games*. Charles River Media, 2007.

[34] Sweetser, P. Emergence in games.

[35] Tolentino, J. Good idea, bad idea: Dynamic difficulty adjustment `http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml`.

[36] wikipedia. `http://en.wikipedia.org/wiki/Dynamic_game_difficulty_balancing`.

[37] Yannakakis, G. N., and Hallam, J. Towards capturing and enhancing entertainment in computer games. In *Advances in artificial intelligence*. Springer, 2006, pp. 432–442.