ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS E INGENIERÍA DE COMPUTADORES

Curso Académico 2022/2023

Trabajo Fin de Grado

DESARROLLO DE COMPORTAMIENTOS DINÁMICOS APLICADO A AGENTES ARTIFICIALES

Autor: Iván Rodríguez García

Directores: Julio Guillén García

"A mi familia por haberme apoyado en la realización del TFG animándome y estando ahí cuando siempre que les necesito.

A mis amigos que siempre me han escuchado.

A mis compañeros de clase.

A mi tutor por estar siempre apoyándome,

Muchas gracias"

Resumen

Para la realización de este trabajo se ha desarrollado una herramienta de planificación de acciones orientada a objetivos capaz de generar planes de acciones que permiten a multitud de agentes artificiales lograr unos objetivos concretos. Se ha creado de forma genérica para C# y específica para Unity logrando aprovechar sus características facilitando así su implementación y su uso.

Se ha realizado una exhaustiva investigación previa con el objetivo de adquirir todos los conocimientos relacionados con la herramienta, desde sus predecesores, el contexto histórico en el que surgió y su evolución con el paso de los años. Además, se han estudiado diversas implementaciones realizadas tanto en grandes compañías como versiones más simples de carácter independiente.

Finalmente, en pos de evaluar el funcionamiento de la herramienta desarrollada y demostrar sus capacidades se han realizado dos ejemplos, uno de carácter genérico y otro de carácter específico de Unity para evaluar los objetivos conseguidos y recopilar posibles mejoras que ayuden a mejorar este proyecto en un futuro.

Palabras Clave:

Inteligencia artificial

Planificación de acciones

Unity Engine

Orientada a objetivos

Herramienta para el desarrollo de videojuegos

STRIPS

Abstract

A Goal-Oriented Action Planner tool has been developed to conduct this project. It allows multiple artificial agents accomplish specific goals. A generic C# version and a concrete Unity version have been developed to take advantage of the main characteristics of each one being easier to implement and to use.

Extensive research has been done with the aim of acquiring all the knowledge related to the tool, from its predecessors, the historic context in which it arose and its evolution over to the years. In addition, various implementations carried out both in large companies and indie ones have been studied.

Finally, in order to evaluate the operation of the developed tool and demonstrate its capabilities, two examples have been made, one generic example and other Unity-specific example to evaluate the objectives achieved and collect possible improvements that will help to improve this project in a future.

Keywords:

Artificial intelligence

Action planning

Unity Engine

Goal Oriented

Tool for Videogame development

STRIPS

Índice de contenidos

Índice de ilustraciones

Índice de tablas

Glosario

Instanciar	Acción de crear una instancia de un objeto en un lenguaje de programación orientado a objetos, C# en este caso.
Scriptable Object	Tipo de objeto de Unity fácilmente editable que define los datos de un objeto instanciable.
Debug	Acción de comprobar los posibles errores de un programa.
GameObject	Objeto de Unity capaz de almacenar componentes en su interior y que siempre posee un Transform que define su posición, rotación y escala en el entorno del videojuego.
Asset	Recurso empleado en Unity que puede ser de cualquier tipo, desde ScriptableObjects hasta imágenes, modelos, animaciones o incluso sonidos.
Package	Paquete instalable en el entorno de Unity, consiste en un archivo que incorpora todos los archivos necesarios para la instalación de una herramienta como assets, clases o prefabs.
Frame	Fotograma de videojuego y es una imagen renderizada del entorno. Se considera normalmente que una simulación adecuada debe contar con 30 frames cada segundo como mínimo.
Tick	Unidad de medida de tiempo que representa el tiempo requerido para procesar y actualizar los elementos de un videojuego. Comúnmente empleado como sinónimo de frame.

Capítulo 1

Introducción

1.1 Contextualización

En el mundo de los videojuegos existen multitud de algoritmos relacionados con inteligencia artificial, capaces de modelar la inteligencia de los personajes artificiales independientemente de si se tratan de enemigos o de acompañantes del jugador.

Algunos de ellos son muy conocidos como las **FSM**, máquinas de estados que definen el comportamiento artificial dependiendo del estado en el que se encuentren; los **árboles de comportamiento**, que define un comportamiento a través de diferentes tipos de nodos y de condiciones; y los **sistemas de utilidad** que permiten escoger entre diferentes comportamientos tratando de satisfacer distintas utilidades.

Uno de ellos recibe el nombre de **planificador de acciones orientado a objetivos** (en inglés, Goal-Oriented Action Planning (**GOAP**)) que causó mucho furor cuando salió en F.E.A.R en el año 2005 debido a la gran atmósfera que consiguió, en gran parte, haciendo uso de esta inteligencia artificial que resultó ser más realista y reactiva que todo aquello que se había apreciado en los videojuegos hasta entonces.

El uso de GOAP resulta interesante y puede llegar a abrir un mundo de posibilidades debido al gran potencial de esta herramienta, aunque también posee ciertas dificultades que se deben afrontar, pues es la planificación puede llegar a suponer una gran carga computacional que debe definirse adecuadamente cuando y cuantas veces va a realizarse para que la simulación no se vea interrumpida y los agentes reaccionen con la rapidez suficiente como para resultar verídico.

1.2 Estado del arte

Los videojuegos han ido evolucionando con el paso de los años dotándose cada vez más con gráficos hiperrealistas y de animaciones que tratan de ser lo más verosímiles posible. En este contexto, se requiere también de unos agentes artificiales capaces de estar a la altura de lo que

espera el jugador, capaz de dar lugar a comportamientos sorprendentes e interesantes que aumenten la inmersión del videojuego [1].

Para ello se empiezan a emplear técnicas que permiten dotar al agente artificial de comportamientos orientados principalmente a uno o varios objetivos y recibe el nombre de **Comportamiento Orientado a Objetivos** (en inglés, Goal-Oriented Behaviour (**GOB**)). Son muchos los videojuegos cuyos personajes toman decisiones basadas en objetivos, pero se trató de alcanzar un paso más allá con lo que se conoce como GOAP [2]. Es descrito como un comportamiento GOB con la peculiaridad de que es capaz de planificar secuencias de acciones con la finalidad de alcanzar un objetivo [1]. Esta secuencia es definida como un plan de acciones que satisface uno o varios de los objetivos del agente.

GOAP dota de inteligencia a los agentes artificiales de un videojuego usando la corriente psicológica GOB como referencia, relacionada con el carácter conductual de los seres vivos a través de los objetivos y las metas. Para ello emplea un algoritmo que es capaz de generar una secuencia de acciones con alta probabilidad de éxito cuya implementación ha ido evolucionando con el paso del tiempo.

El origen de este tipo de algoritmos se da con el **Solucionador de Problemas del Instituto de Investigación de Stanford** (en inglés, Stanford Research Insitute Problem Solver (**STRIPS**)), y en un campo no orientado únicamente a videojuegos, sino que también está altamente relacionado con la robótica. Consiste en un solucionador de problemas cuya principal característica reside en contar con una representación simbólica del mundo definida por un conjunto de literales [3].

STRIPS toma inspiración de la demostración de teoremas descrita por Green en 1969, que es el método que emplea para definir si una secuencia de acciones es válida o no lo es. No obstante, a diferencia de la empleada por Green, se emplea una estrategia de tipo Solucionador de problemas genérico (en inglés, General Problem Solving (GPS)) con el objetivo de poder utilizar modelos de mundo mucho más complejos y generales de lo que se habían podido utilizar hasta el momento [3].

De esta manera surge un solucionador de problemas genérico, a diferencia de la inteligencia artificial más específica empleada hasta el momento como, por ejemplo, un solucionador de

puzles (**Torres de Hanoi**) o una inteligencia artificial que juega al ajedrez. Con STRIPS por primera vez se pueden solucionar diferentes tipos de problemas siempre que se definan ciertos elementos principales a raíz de estos.

- Representación simbólica del mundo inicial: un conjunto de literales con valores booleanos que permiten representar parte de la información del mundo relevante para el problema que se trata de solucionar.

- Acciones que cambiar el estado del mundo: en STRIPS se hacían por acciones de ADD y de REMOVE, y permiten cambiar el valor de un literal, definidas como **Fórmulas Bien Finas** (en inglés, Well Fined Formulas (**WFF**)) por otro distinto cambiando el estado actual del mundo. Esta acción para poderse aplicar debe contar con unas precondiciones, es decir, parte de la información del mundo debe poseer varios valores.

- Objetivos: la parte más importante, pues es lo que define el tipo de algoritmo orientado a objetivos.

Acerca de cómo se definen estas WFF en STRIPS se destaca el hecho de que las acciones presentan parámetros que forman parte de un conjunto previamente definido entre los cuales puede escoger.

Respecto a cómo STRIPS realiza la búsqueda del mejor plan consiste en establecer qué cambios afectan al objetivo y cuales pueden darse por ignorados. Se ignoran aquellos que no tengan ninguna consecuencia en el objetivo que se desea tener y mientras se van definiendo nuevos objetivos necesarios para poder realizar esas acciones que tienen consecuencia directa en el objetivo. Combina ventajas tanto de la búsqueda directa como de la búsqueda regresiva y aplica como heurística, ya que se trata de un A* con ciertas modificaciones, el número de objetivos que tiene cada nodo escogiendo el que menor tenga, para ello los va almacenando después de cada operador.

Debido a que se usa la demostración automática de teoremas se deben tomar algunas cosas en consideración, como que los parámetros de las acciones al ser cogidos de un conjunto esto debe ser tomado en cuenta para todas las reglas de un tipo, por ejemplo, que un parámetro t es igual

que p debe ser tomado en cuenta para todos. De esta manera se verifica si un parámetro es válido o no lo es.

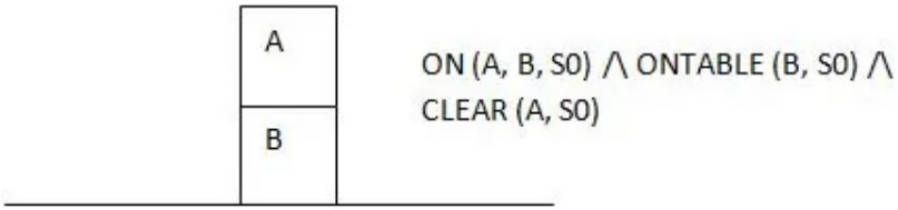


Ilustración 1 - Representación de un estado del mundo en STRIPS

De esta manera, se definen las acciones en STRIPS en tres partes, las precondiciones que se definen como el resto de WFF. Y luego una lista de "delete" donde se establecen los WFF que serán falsos en el nuevo modelo y "add" los cuales serán ciertos en el nuevo modelo.

En definitiva, consiste en intentar buscar alguna secuencia de acciones que encuentra un modelo sin diferencias con el objetivo en cuestión.

STRIPS funcionaba de forma bastante eficaz, pero el empleo de demostración de teoremas podría llegar a ser tedioso y algo dependiente de una estructura muy rígida y definida. En realidad, se podían utilizar variables, pero los efectos de las acciones se veían muy limitados eliminando y añadiendo reglas.

La razón por la que STRIPS necesita eliminar un literal antes de asignarle un nuevo valor reside en que realmente no tiene constancia de las reglas que están previamente escritas, simplemente se añaden. Por esa razón, si solo se agrega un literal con el valor contrario se está creando una contradicción y el objetivo nunca podrá ser subsanado.

Antes de GOAP existen videojuegos que emplean comportamientos basados en objetivos pero que carecen de planificación, como es el caso del videojuego **No One Lives Forever 2** (NOLF2). La principal diferencia entre el GOB utilizado en NOLF2 y GOAP consiste en la forma en la que son tratados los objetivos de cada uno de ellos; en NOLF2 se distinguían diferentes tipos de objetivos y cuando uno de ellos era activado le seguía la ejecución de un

comportamiento previamente establecido, GOAP en cambio define sus objetivos únicamente como condiciones que deben cumplirse y la forma en la que estas sean satisfechas se determina en **tiempo real** [2].

Con el tiempo surge GOAP como una abstracción de STRIPS con ciertas simplificaciones y añaden nuevas funcionalidades o simplifican otras [4].

- **Coste por acción**: aplicando el algoritmo de A*, la introducción de los costes en los operadores se emplea para preferir el uso de ciertas acciones frente a otras. Esto se puede establecer de forma dinámica y se aplica en el pathfinding, pues si hay unas acciones que se pueden realizar sin que esto lleve más tiempo caminando pues se preferirán las que se estime que tarden menos en ejecutarse.

- **Desaparecen las listas de adición y desecho**: para lograrlo ya no se emplea el lenguaje formal empleado en STRIPS que se basaba en la demostración de teoremas. Con GOAP se simplifica al máximo definiendo un conjunto fijo de literales (que pueden tomar cualquier valor), con el objetivo de que solo se toman en cuenta esos posibles literales y los valores que pueden adoptar.

- **Efectos y precondiciones procedurales**: finalmente se añade la posibilidad de que los efectos y las precondiciones puedan ser procedurales y de esa manera tener una funcionalidad concreta diferente a la de tener el valor deseado. De esa manera se puede definir que una acción asignará al valor At el que tenga designado en el objetivo actual (ej.: acción “GoTo”) lo que simplifica la forma en la que se definen las acciones.

GOAP además cuenta con distintas variaciones entre las que se encuentran **Utility GOAP** y **Smelly GOAP** [1]. Estas consisten en versiones anteriores a GOAP que se caracterizaban por tomar en cuenta todos los objetivos del agente por igual buscando los planes que satisfagan en mayor medida el mayor número de objetivos. No obstante, finalmente esto termino aumentando la complejidad significativamente y se buscaron nuevas perspectivas.

Se define el **GOAP con IDA***, esta versión de GOAP presenta grandes ventajas y un único inconveniente significativo ya que debe centrarse en un único objetivo. Esta es la versión que

se emplea en este proyecto ya que permite encontrar planes rápidamente aplicando una heurística adecuada.

A pesar de que GOAP es una herramienta bastante veloz y efectiva gracias a la incorporación de diversas optimizaciones esto no termina por ser suficiente para lo que ciertos estudios plantean para sus videojuegos. El principal inconveniente es que esta planificación no toma en cuenta múltiples agentes trabajando en conjunto.

Se emplea en esta situación la **Red de Tareas Jerárquicas** (en inglés, Hierarchical Task Network (**HTN**)) para solucionar este problema, en conjunto con GOAP [5]. Al principio HTN solo se empleaba en el ámbito de la producción, para organizar tareas. Con este nuevo planteamiento se introduce su uso dentro del ámbito de los videojuegos.

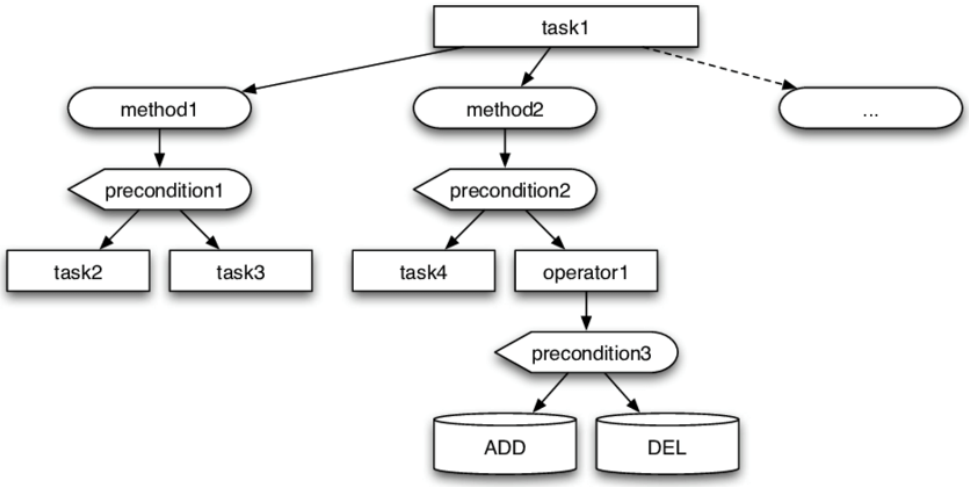


Ilustración 2 - Posible representación de elementos usados en HTN

El principal objetivo de incluir el uso de HTN en GOAP es el poder aplicar comportamientos de equipo más generales, es decir, de un nivel superior. De esta manera a bajo nivel estaría la lógica de GOAP específica para cada **agente** mientras que a nivel superior se establece el comportamiento de distintos agentes en conjunto.

1.3 En la actualidad

En los últimos años GOAP se ha mantenido estable sigue compitiendo con otras IA más sencillas ya sean árboles de comportamiento y conviviendo con las FSM desde su primera aparición. Múltiples juegos se han visto influenciado por esta tecnología y cada uno realiza sus propias interpretaciones de GOAP para conseguir un resultado más cercano a lo que cada uno requiere.

Esto es debido a que GOAP es una herramienta que carece de estándares fijos, más allá de sus componentes elementales se puede modificar y cada uno puede realizar su propia interpretación de ella. Por esta razón consiste en una herramienta muy potencial que puede ser extendida o explotada de diferentes maneras, según lo requiera cada uno.

Son muchos los juegos que usan GOAP para lograr que sus juegos se vean más realistas e inteligentes. A continuación, se puede apreciar una recopilación de algunos juegos que han confirmado el uso de GOAP entre sus algoritmos de inteligencia artificial.

Tabla 1 – Videojuegos que han confirmado públicamente el uso de GOAP

Título	Estudio	Plataforma	Año
F.E.A.R.	Monolith Productions	X360, PS3, PC	2005
Condemned: Criminal Origins	Monolith Productions / SEGA	X360, PC	2005
S.T.A.L.K.E.R.: Shadow of Chernobyl	GSC Game World / THQ	PC	2007
Mushroom Men: The Spore Wars	Red Fly Studio	Wii	2008

Ghostbusters	Red Fly Studio	Wii	2008
Silent Hill: Homecoming	Double Helix Games / Konami	X360, PS3	2008
Fallout 3	Bethesda Softworks	X360, PS3, PC	2008
Empire: Total War	Creative Assembly / SEGA	PC	2009
F.E.A.R. 2: Project Origin	Monolith Productions / Warner Bros	X360, PS3, PC	2009
Demigod	Gas Powered Games / Stardock	PC	2009
Just Cause 2	Avalanche Studios / Eidos Interactive	PC, X360, PS3	2010
Transformers: War for Cybertron	High Moon Studios / Activision	PC, X360, PS3	2010
Trapped Dead	Headup Games	PC	2011
Deus Ex: Human Revolution	Eidos Interactive	PC, X360, PS3	2011
Tomb Raider	Square Enix	PC, PS4, Xbox One, X360, PS3	2013
Middle-Earth: Shadow of Mordor	Monolith Productions	PC, PS4, Xbox One	2017

Assassin's Creed Odyssey	Ubisoft	PC, PS4, Xbox One	2018
Immortals Fenyx Rising	Ubisoft	PC, Nintendo Switch, PS4, PS5, Xbox One, Xbox Series X/S	2020

Como se puede GOAP ha estado muy presente en el ámbito de las inteligencias artificiales desde que apareció hasta la actualidad, aunque siempre con sus modificaciones específicas que han ido variando con lo largo de los años [6].

Se aprecia que en un principio la tendencia de GOAP predominaba en el género del survival horror, se entiende debido a que es uno de los géneros que más necesita sumergir al jugador en el contexto y una inteligencia artificial más realista produce más inmersión y por lo tanto más terror. Con el paso del tiempo ha dado paso a todo tipo de géneros predominando en los juegos de aventuras y estrategia .

En una de las conferencias más recientes se expone una forma de extender GOAP para que sea capaz de establecer una comunicación cooperativa entre múltiples agentes [7]. La idea principal reside en que se añaden al planificador de GOAP acciones que solo pueden ser realizadas por otros agentes, y que requieren cooperación.

Estas serán tomadas en cuenta en el momento en el que el coste del plan más bajo requiera una de estas acciones, por lo que a través de un sistema de mensajes se preguntan los agentes les preguntan a otros si podrían realizar ciertas acciones por ellos y esperan una respuesta. Esto conlleva a dos escenarios, en el caso de que puedan se seguirá con la planificación y en el caso de que no tendrá que buscar otras alternativas.

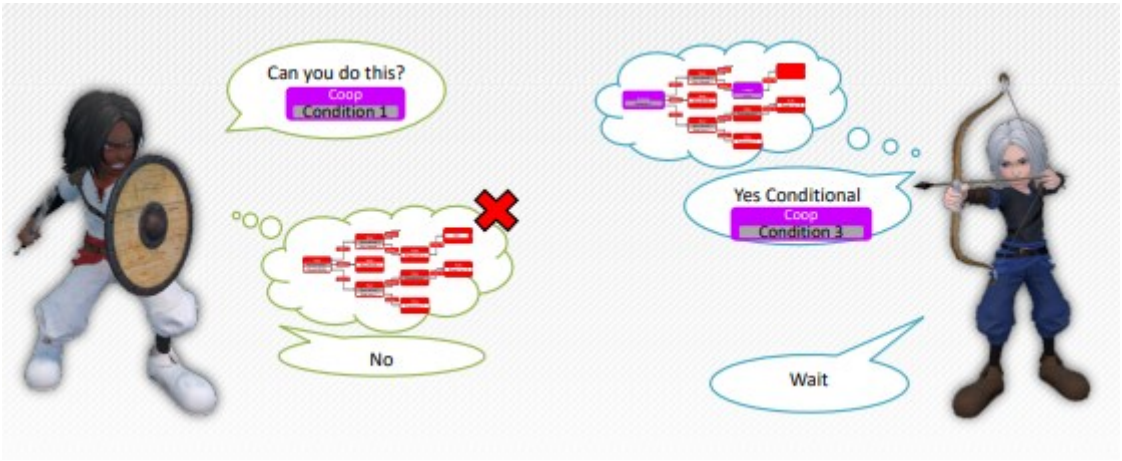


Ilustración 3 - Comunicación entre agentes con GOAP

También se puede dar el caso de que otro agente pueda cumplir con esa tarea solo si el primer agente realiza otra acción por él, en el caso de que la respuesta sea si entonces queda resuelto el conflicto. Finalmente, si no puede realizarlo no significa directamente que no pueda realizar esa acción, el segundo agente podrá preguntar a un tercer agente por esa acción que necesita y en el caso de que se pueda entonces le responderá al primer agente con una respuesta afirmativa.

En conclusión, GOAP sigue siendo útil y sigue evolucionando desde el momento en el que surgió y cuenta con muchas variantes y extensiones distintas que demuestran que se trata de una herramienta muy útil y adaptable.

Capítulo 2

Objetivos

2.1 Descripción del problema

Con este trabajo se pretende desarrollar una herramienta capaz de resolver problemas dentro de entornos simulados, logrando que agentes artificiales determinen planes de acciones que los lleven a cumplir ciertos objetivos.

Además, se pretende realizar una herramienta que sea **sencilla** de definir y de utilizar pues así se pueden crear escenarios más **elaborados** y **vivos** con mayor facilidad. Cabe destacar el hecho de que debe cumplir con los estándares básicos de una implementación de este tipo, ya que, al ser una herramienta desarrollada para entornos de videojuegos debe ser compatible con la ejecución en **tiempo real**, esto significa que no debe interrumpir la ejecución de la simulación por demasiado tiempo.

Para ello se realiza un estudio y una investigación exhaustiva acerca de GOAP en donde se pondrá el foco en el contexto en el que se originó siendo F.E.A.R. un caso de estudio muy importante y también describir como Orkin define la herramienta y cómo ha evolucionado con el paso de los años a raíz de que no posee un estándar normalizado.

A raíz de toda esta información se podrán extraer las principales características que la herramienta debe tener con la finalidad de obtener un algoritmo genérico que también puede ser implementado en Unity de forma eficiente, es decir, que pueda realizar muchos planes simultáneos sin que suponga un detrimento de la eficiencia de la aplicación en la que la herramienta es usada.

Finalmente se realizará un diseño de GOAP en el que se pueda definir con claridad en que consiste para finalmente validarlo con unos ejemplos prácticos que muestren su potencial y sus capacidades.

2.2 Objetivos

Estos son los objetivos planteados para el presente trabajo:

- Realizar un estudio de los antecedentes y las aplicaciones actuales de la planificación de acciones orientada a objetivos.
- Definir en que consiste GOAP y extraer las características más usadas en la herramienta comparando diferentes implementaciones entre sí.
- Diseñar una herramienta de GOAP funcional y eficiente.
- Implementar una herramienta de GOAP de carácter general específica para Unity.
- Diseñar e implementar ejemplos que permitan validar la herramienta.

2.3 Estudio de alternativas

Un algoritmo GOAP puede ser implementado prácticamente en cualquier lenguaje orientado a objetos. Entre todos ellos se ha decidido implementar la herramienta en el lenguaje C Sharp debido su alta popularidad, formando parte del top 5 de lenguajes de programación más utilizados en los últimos años y también debido a su alto rendimiento que ayuda a procesar la planificación realizada por el algoritmo.

Aunque principalmente se ha usado C# debido a su alta integración en otras aplicaciones, y más importante, en los motores de videojuegos más utilizados hasta el momento [8]. Unity es uno de ellos y se ha decidido realizar una implementación especifica de este algoritmo de GOAP porque posee una gran comunidad y se considera que esta se beneficiaría de una implementación de este tipo pudiendo crear sus propias inteligencias artificiales de manera sencilla.

Cabe destacar que Unreal Engine también hubiera sido una gran opción a la hora de implementar el algoritmo, debido a su eficiencia y que ofrece un acabado artístico muy bueno de forma muy sencilla, pero se ha decidido priorizar la usabilidad y el alcance que puede ofrecer Unity frente a esto.

2.4 Metodología empleada

Se plantea una metodología de **desarrollo iterativo e incremental**, más concretamente **SCRUM** debido a que se espera tener funcionalidades acabadas en plazos de una a dos semanas. Se ha aplicado tanto para realizar la memoria como para la implementación de la herramienta en C Sharp y en Unity. Para ello se ha hecho uso de la herramienta online Trello que permite organizar un tablero en diferentes listas de tareas.

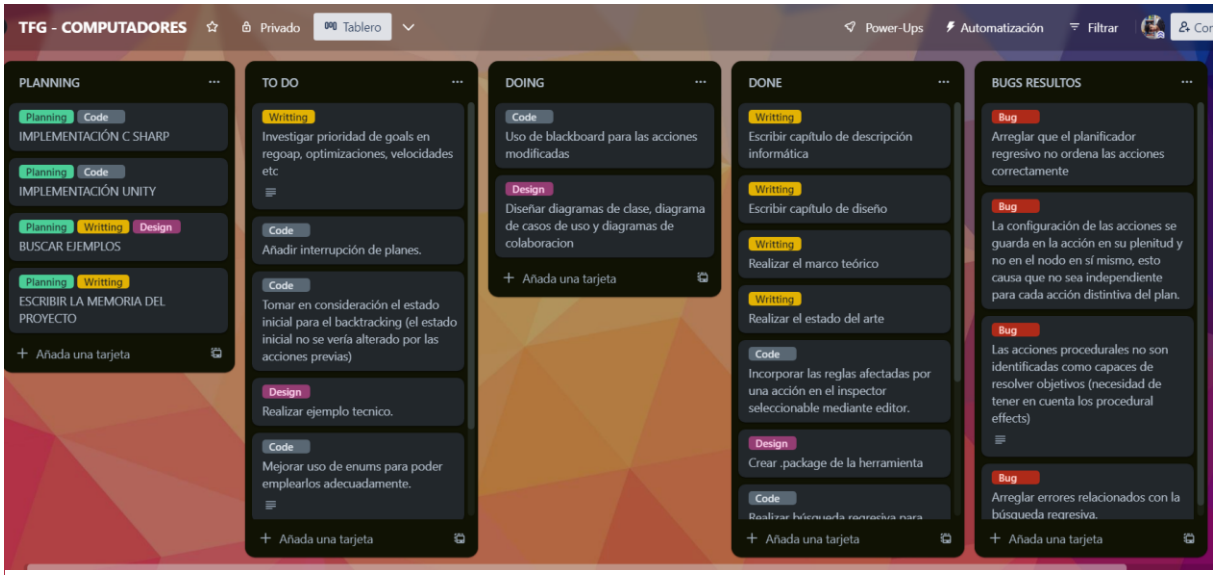


Ilustración 4 - Tablero de Trello aplicando SCRUM

Estas listas incluyen las típicas necesarias para un desarrollo SCRUM; “to do”, tareas pendientes por empezar; “doing”, actualmente en desarrollo y “done”, tareas terminadas. Por otro lado, se añade la lista de “planning” que hace la función de historias de usuario para englobar conjuntos de tareas y la lista de “bugs resueltos” empleada para almacenar tareas de tipo *bug* que hayan sido resueltas.

He de destacar respecto a las tareas la existencia de cuatro etiquetas principales; *code* para tareas que implican programación, *writting* que describir tareas relacionadas con la escritura de la memoria, *design* que involucran diseño ya sea para la creación de diagramas o la creación de los ejemplos y finalmente *bug* para errores encontrados en el funcionamiento de la aplicación.

2.5 Planificación

Para la realización de este trabajo se han realizado diferentes tareas en plazos de una o dos semanas cada vez, para poder programar y planificar el desarrollo del proyecto a lo largo del tiempo se ha realizado un **diagrama de Gantt** en Excel.

Nombre de tarea	Fecha de inicio	Duración en días	Fecha de fin
Investigación del contexto y evolución de GOAP	06-mar	27	02-abr
Redacción de la memoria del trabajo	03-abr	91	03-jul
Recopilación de casos de uso	12-jun	7	19-jun
Desarrollo de las clases principales del algoritmo	06-abr	14	20-abr
Implementación de A*	21-abr	14	05-may
Diseño del ejemplo de las Torres de Hanoi	06-may	7	13-may
Testeo 1	14-may	7	21-may
Creación de ScriptableObjects para Unity	22-may	7	29-may
Creación de inspectores personalizados	30-may	7	06-jun
Implementación de búsqueda regresiva	07-jun	14	21-jun
Creación del ejemplo de Unity	22-jun	7	29-jun
Testeo 2	30-jun	7	07-jul
	Inicio proyecto	06-mar	
	Fin proyecto	07-jul	

Ilustración 5 - Planificación de tareas

Primero se especifican las tareas realizadas y el tiempo que ha abarcado realizar cada una de ellas, se aprecian dos vertientes, la del desarrollo de la aplicación y la de la redacción de la memoria que se han llevado a cabo al mismo tiempo en el tiempo. Finalmente, a raíz de estos valores se crea el diagrama de Gantt que representa esta información de forma más clara.

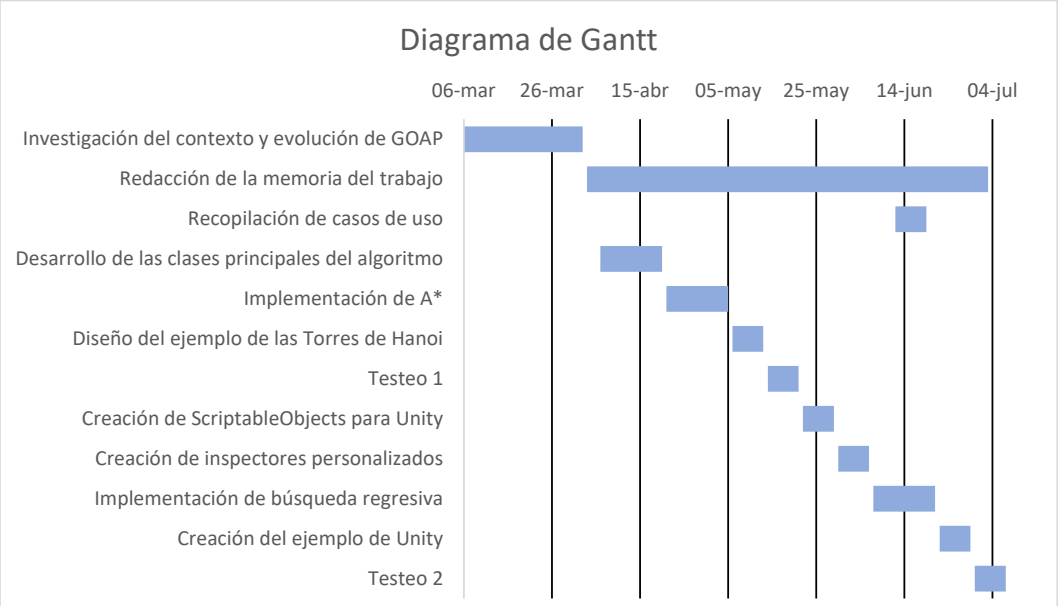


Ilustración 6 - Diagrama de Gantt

Capítulo 3

Marco teórico

3.1 Qué es GOAP

Son numerosos los juegos que presentan personajes capaces de tomar decisiones respecto a un objetivo relevante, pero GOAP añade un mayor grado de sofisticación al introducir la capacidad de decidir, no solo qué van a hacer, sino cómo van a hacerlo. El principal objetivo de esta arquitectura es dotar al personaje de una mayor **variabilidad de comportamientos**, lo que los hace más impredecibles [2].

El principal propósito de GOAP es generar un plan de acciones de forma dinámica para lograr un objetivo específico. Al utilizar GOAP los agentes artificiales del videojuego, también conocidos como **Personajes No Jugables** (en inglés, Non-Playable Characters (**NPC**)), adquieren una mayor inteligencia y capacidad de resolver problemas de forma estratégica.

A pesar de que GOAP no cuenta con un estándar y sus implementaciones varían mucho desde unas implementaciones a otras, define un conjunto de elementos básicos que se encuentran presentes en todas y cada una de estas implementaciones.

- **Objetivo**: es cualquier condición que el agente desea satisfacer. Cada agente puede contar con uno o con varios objetivos y la forma en la que se decanta entre uno u otro varía según la implementación. Esta prioridad puede ser predefinida previamente en diseño o que se defina en tiempo real según el estado en el que se encuentre el mundo o el agente en un preciso momento.

- **Acción**: es un paso atómico perteneciente al plan que hace que el personaje haga "*algo*". La duración de cada acción puede variar entre ser inmediata y ser infinitamente larga.

Cada acción se establece de forma de que se conoce perfectamente cuándo puede ejecutarse y qué efectos tendrá en el estado del mundo, esto lo consigue a través de sus precondiciones y de sus efectos de número indeterminado. Así se puede establecer la consecuencia de unas acciones con otras pues puede ser que las precondiciones de una acción sean los efectos de otra acción distinta, y esto será aprovechado por el planificador.

- **Plan**: define una secuencia de acciones que satisface un objetivo específico si puede llevar al personaje desde un estado inicial hasta otro en el que el objetivo queda satisfecho.

Las principales ventajas que se describen usando GOAP frente a previos comportamientos basados en objetivos son principalmente las siguientes.

- **Independencia entre acciones y objetivos**: GOAP se encarga de buscar el camino de acciones adecuado para cumplir con las condiciones que caracterizan los objetivos lo que permite que se adapte dinámicamente a los cambios de su entorno.

De esa manera ya no es necesario que se diseñen cada una de las excepciones que podrían ocurrir a la hora de ejecutar las acciones de un objetivo cómo ocurría en NOLF2 que no usaba GOAP, donde un conjunto de acciones era ejecutado, tomando en cuenta algunas condiciones, y se daba por cumplido el objetivo.

GOAP previene de fallos solucionándolos dinámicamente y se implementan funcionalidades sin que eso afecte a las previamente desarrolladas.

- **Estructura escalable y adaptable**: ya que las acciones están definidas por precondiciones y efectos y los objetivos por precondiciones hace que sean muy modulares y que distintos agentes puedan tener distintos objetivos y acciones o también tener algunos de ellos en común.

Permite una gran variabilidad de comportamientos realistas e inmersivos sin que un diseñador tenga que pensar en cada posible situación que se pueda dar, llegando incluso a comportamientos que ni los propios creadores imaginaban que pudieran darse.

- **Replanificación automática y manejo de dependencias**: en el momento en el que un plan no puede darse a cabo, se realiza otro plan para el objetivo o si no se encuentra ninguno para el objetivo de siguiente prioridad [9]. El agente de esta manera siempre tiene un comportamiento en mente y es capaz de adaptarse a los cambios inesperados que pueden darse en el mundo. Además, debido a su capacidad de adaptarse es capaz

también de resolver las dependencias que puedan surgir, logrando realizar cualquier acción siempre resolviendo primero todas las acciones previas necesarias.

Los planes se formulan a través de lo que se conoce como un planificador, que será el encargado de buscar en el espacio de acciones un plan que sea capaz de cumplir con un objetivo. Una vez encontrado un plan factible el NPC lo realizará hasta que se finalice, se invalide hasta que otro objetivo termine por ser más relevante. En estos dos últimos casos el personaje abundará el plan formulando uno nuevo [2].

Existen diferentes posibilidades a la hora de establecer el enfoque con el que el planificador encuentra el mejor plan para el objetivo del agente. Orkin propone el uso de A* de forma que se almacene para cada nodo el **coste real** y el **coste heurístico** o estimado con respecto al objetivo. El coste heurístico puede ser desde el número de condiciones que no se cumplen respecto al objetivo o ser más complejas tomando en cuenta la "distancia" a la que se encuentran estas condiciones si están formadas por variables que poseen **valores numéricos**.

Las acciones suponen **transiciones** entre los distintos **nodos** y cada nodo supone un **estado distinto del mundo** y el camino se puede formar desde el punto de vista de dos planteamientos:

- **Búsqueda directa**: en la que se parte desde el estado actual y se trata de encontrar un estado en el que las condiciones del objetivo queden satisfechas. La desventaja principal es que se toman en cuenta todas las acciones posibles que cumplan con las precondiciones lo cual puede dar pie a que el número de nodos explorados se eleve con rapidez.

- **Búsqueda hacia atrás**: en la que se parte desde el estado objetivo y se emplean acciones que puedan satisfacer dichas condiciones sin tener en cuenta las precondiciones pues pasarán a formar parte del objetivo una vez son empleadas. De esa manera se emplean únicamente acciones que colaboran con resolver los conflictos y finalmente se da el plan por encontrado cuando no quede ninguna condición por satisfacer.

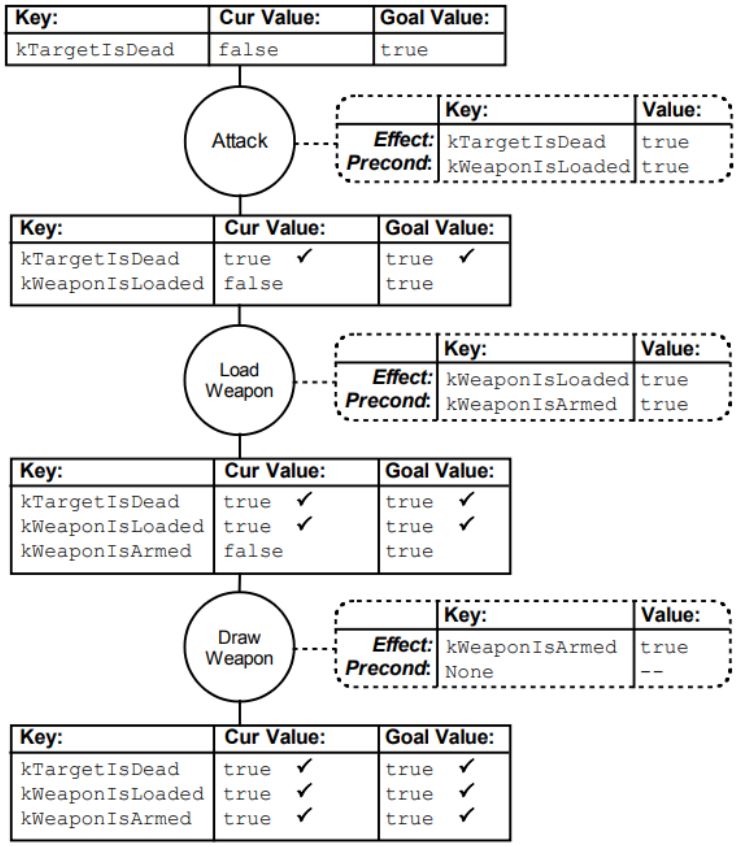


Ilustración 7 - Planificación back tracking según Orkin

Esta planificación queda reflejada en la Ilustración 3. A pesar de ello se debe tomar en consideración que según lo presenta Orkin parece entenderse que cuando una acción satisface una condición esta queda satisfecha para siempre a lo largo del plan.

Esto no es siempre cierto, porque puede que haya una acción que requiera validar una misma condición que ya ha validado otra acción previa, pudiera parecer que no tendría problema y que podría ejecutarse, pero entonces se estaría olvidando el hecho de que se está realizando el plan hacia atrás, es decir, **acciones previas** suponen acciones en el **futuro**, por lo que **no pueden** darse por **cumplidas** las **condiciones** que ya han sido **resueltas**.

Para solucionarlo se debe entender que, si una precondición vuelve a aparecer, vuelve a tomarse en cuenta y solo será resuelta por otra acción posterior que afecte a dicha propiedad en cuestión. O si bien las últimas precondiciones del objetivo ya estaban saciadas por el estado inicial.

Finalmente, se debe explicar que una propiedad es un valor que forma parte de la representación simbólica del estado del mundo. Un estado del mundo no tiene por qué estar conformado por toda la información que se crea importante, sino que se suele hacer una selección de características relevantes que se reducen en lo que se conoce como propiedades.

Las **propiedades** pueden ser muy **sencillas**, siendo todas valores booleanos con valores verdadero o falso, pero también pueden estar formadas por **variables numéricas** o incluso por referencias a objetos. Poseerán una id que indicará a qué personaje pertenece, luego la clave que diferencia unas propiedades de otras y finalmente el **valor** que está **relacionado** con esta propiedad, cualquier tipo de valor [2].

3.2 Casos de estudio

Como ya se ha visto GOAP es una técnica que ha sido utilizada y extendida por muchos años. Se realiza un estudio para apreciar su implementación en diferentes ejemplos y estudiar estas variaciones y qué otras herramientas de inteligencia artificial se implementan.

3.2.1. F.E.A.R.

En la conferencia GDC 2005 hizo una lista de las técnicas de inteligencia artificial más aplicadas en los videojuegos, resultando en que las más empleadas eran A* como algoritmo de búsqueda y Máquinas de Estados Finitos (en inglés, Finite State Machines (FSM)) [4].

Ilustración 8 - Frame del videojuego F.E.A.R. 2005

En este contexto aparece F.E.A.R. como el primer videojuego en implementar la técnica GOAP como parte de su sistema de inteligencia artificial. Aprovechó sus ventajas principales para lograr mostrar unos NPCs inteligentes y reactivos al entorno y al jugador.

Además de las principales características de GOAP, incorpora ciertas innovaciones con respecto al comportamiento de equipos. Estos comportamientos se dividen en dos tipos, los simples y los complejos.

Los **comportamientos de equipo simples** son gestionados por el comportamiento de equipo (en inglés, Squad Behaviour) y se componen de cuatro pasos principales:

1. Encontrar agentes que puedan realizar las tareas requeridas.

2. Enviar órdenes a los miembros del equipo creado.

3. Si la orden recibida se considera una prioridad para el agente en ese momento entonces cumple con las órdenes recibidas.

4. El Squad Behaviour comprueba en cada tick el progreso de cada agente que seguirán las órdenes a no ser que hayan muerto o hayan sufrido cualquier otra interrupción.

El **Squad Behaviour** no tiene que controlar exactamente hacia dónde se dirigirá cada agente, sino comprobar que dos agentes no coincidirán en un mismo sitio. Los agentes únicamente cumplen con el objetivo de seguir las órdenes encomendadas y el comportamiento se considerará éxito si todos los agentes involucrados han cumplido con su parte, si por el contrario ha surgido una fuerza mayor que haya obligado a priorizar otros objetivos entonces terminará en fracaso.

Respecto a los **comportamientos de equipo complejos** la realidad es que F.E.A.R. no incorpora ninguno en específico, sino que la interpolación entre las decisiones realizadas a nivel de equipo y la toma de decisiones propia de cada agente crean la ilusión de que realmente existen.

Otro aspecto importante incorporado por F.E.A.R. consiste en la comunicación otorgada a los agentes, esto permite conocer mejor lo que los agentes artificiales están realizando e incluso pueden dar una mayor sensación de inteligencia de la que realmente está programada [10]. Tal como explica Orkin “cuando un agente detecta que se encuentra solo dirá que necesita refuerzos, pero no existe ningún comportamiento que realmente haga aparecer más enemigos, sino que es común que el jugador avance por el nivel y acaben apareciendo más soldados contra los que luchar, entonces el jugador ha percibido un nivel de inteligencia superior al que se ha establecido” [4].

Finalmente cabe destacar que el algoritmo de búsqueda empleado es A* que aprovecha para dotar de costes a las acciones según el nivel de preferencia entre unas acciones y otras y además utiliza una máquina de estados finitos con tres únicos estados: *Goto*, *Animate* y *Use Smart Object*.

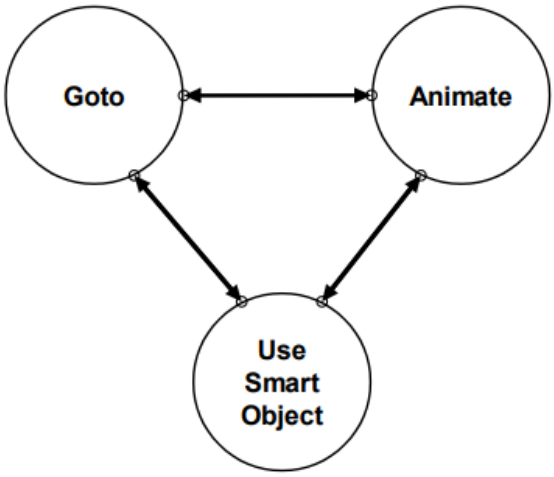


Ilustración 9 - Máquina de estados de F.E.A.R.

La principal diferencia que reside entre estos estados es que *Goto* realiza animaciones mientras se está desplazando hacia un punto concreto del mundo, *Animate* realiza una animación estando en el sitio y *Use Smart Object* hace una animación que involucra un objeto del mundo. Es decir, se manejan automáticamente según las acciones que esté llevando a cabo.

Es la principal novedad que supuso este juego respecto a otras implementaciones de inteligencia artificial, pues la planificación es realizada completamente por GOAP dejando la máquina de estados resumida en funciones básicas principales que se relacionan más con el resultado visual que con la lógica detrás de cada acción.

3.2.2. Middle Earth: Shadow of Mordor

La principal característica que se presenta en el GOAP utilizado por el videojuego de *Middle Earth: Shadow Of Mordor* es que la optimización cobra un papel fundamental, alejando de la planificación de GOAP muchos de los elementos que mayor consumo poseen y lo derivan en subsistemas externos y en sensores [11].

Gracias a estos subsistemas externos el planificador se aleja de trivialidades como qué animación usar o decidir a cuál de todos los enemigos va a atacar. De eso se encargan unos subsistemas que se encargan de elegir cuál es la mejor opción y el planificador solo se centra en aquellos agentes o entidades escogidas. Esto también aplica a la realización de cualquier acción, pues el planificador solo debe conocer que esa acción será realizada, pero no tiene por qué conocer cómo lo hará.

Respecto a cómo funciona el comportamiento de los agentes, se caracteriza por establecer roles tomados en cuenta únicamente por el planificador, de esta manera el planificador tomará decisiones en base a ese rol en específico y permite comportamientos adaptables y variados.

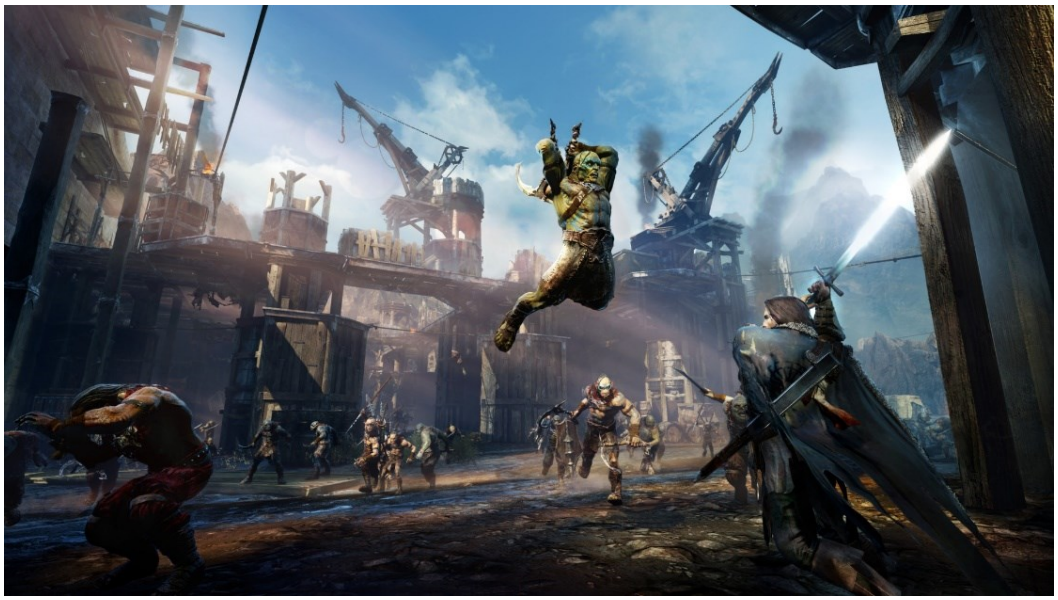


Ilustración 10 - Frame del videojuego Middle Earth: Shadow of Mordor

Este comportamiento optimizado de GOAP permitió integrar lo que se conoce como Sistema Némesis que fue considerado una revolución en cuanto a Inteligencia Artificial [12]. Con este sistema los enemigos aprenden del jugador y tratan de explotar sus debilidades para vencerle y, además, recuerdan al jugador. Por ejemplo, si se derrota a un orco, pero no se le mata este puede aparecer más tarde con cicatrices y con sed de venganza.

Se genera de esta manera, y con uso de técnicas de machine learning, una narrativa de forma procedural y única para cada jugador pues la forma de jugar de cada uno se reflejará en el mundo y creará historias únicas. Esto se había conseguido previamente a través de relaciones previamente establecidas, pero esta es la primera vez que se consigue de forma procedural.

En definitiva, se estableció que cualquier funcionamiento que pueda ser stand-alone puede ser extraído del planificador para acelerar la carga de recursos de la IA en gran medida y así se implementó en conjunto con sistemas globales como el sistema némesis que almacenaba información de cada enemigo para después usarse como una importante herramienta narrativa e inmersiva.

3.2.3. Tomb Raider (2015)

La IA de este juego se centró en los objetivos y en las acciones sin apenas modificar la librería de GOAP, añadiendo ciertas características adicionales para el gameplay. Con respecto a las **acciones**, implementan costes dinámicos para que las acciones sean más conscientes del entorno y para que no siempre se elijan las mismas acciones constantemente.

Por otro lado, respecto al manejo de los **objetivos** presenta un sistema interesante, tomando inspiración de Los Sims 4 hace uso de “motives” o motivos que guían la disponibilidad de ciertos objetivos u acciones. Esto permite que ciertos objetivos solo se tomen en cuenta cuando un motivo tiene ciertos valores o que ciertos objetos sean tomados en cuenta para realizar las acciones necesarias. Además, estos motivos pueden ser modificados por distintas razones, ya sea por su cuenta, o cuando se realizan ciertas acciones.

Implementa también la posibilidad de gestionar más de un plan candidato, permitiendo que un objetivo o una acción pueda incorporar más de un set de requisitos, en lugar de solo uno. Así

se evalúan los distintos requerimientos como planes independientes para decidir cuál es el más conveniente.

Esto permite que una acción genérica, como usar objeto, tenga multitud de posibilidades dependiendo del objeto que sea usado como **parámetro de esta acción**. Añade bastantes opciones al planificador y dará como resultado un comportamiento más inteligente. *Ej.: una acción que pueda utilizar el arma equipada o que requiera equipar un arma mejor primero, estas dos opciones serán tenidas en cuenta por el planificador* [11].

Ilustración 11 - Frame del videojuego Tomb Raider de 2013

Otra característica que destacar es la supervisión de acciones y de objetivos en plena ejecución, de manera que dinámicamente se compruebe que los requisitos siguen estando presentes como para continuar con ello. Además, el hecho de que las acciones o los objetivos puedan comprobar el estado actual del agente permite utilizarlo en su favor para realizar acciones intermedias mientras una principal se está ejecutando.

Por último, se describen acciones con **terminación abierta**, esto significa que mientras que no haya un plan mejor y la acción a realizar sea viable, dentro de un coste razonable, continuará ejecutándola hasta verse interrumpida o que surja otro plan menos costoso. Esto permite emplear distintos planes para un objetivo que se establece y seguirá concluyendo en que quede satisfecho.

3.3 Análisis de REGOAP: una herramienta de GOAP

Al igual que se han analizado distintos videojuegos y sus implicaciones relacionadas con GOAP, se analiza una implementación de GOAP genérica conocida como REGOAP, analizando cuáles son sus características principales y también de cuales carece.

Esta implementación de GOAP para Unity se caracteriza por hacer uso de la concurrencia de manera que todas sus clases hacen uso de hilos y aceleran en gran medida los cálculos realizados por el planificador.

Emplea cinco clases destacadas como importantes: estado del mundo, acción, objetivo, memoria y sensor.

- **Estado del mundo (*State*)**: se implementa a través de reglas que están conformadas por un diccionario, siendo la clave el identificador de la regla y el valor de tipo variable. Se destaca una fábrica de estados que permite reciclar estados que han dejado de utilizarse lo que aumenta considerablemente el rendimiento.

- **Acción (*Action*)**: se componen principalmente de precondiciones y de efectos, cada uno de ellos representando, haciendo uso de la clase State. Además, cada acción hereda de una misma clase base con el objetivo de poder describir mediante código lo que diferencia cada acción de otra distinta compartiendo funcionalidades básicas en lo que respecta al planificador.

- **Objetivo (*Goal*)**: de forma similar a las acciones se representan a través de un único State que recoge las condiciones que caracterizan al objetivo, además de un valor que indica la prioridad del objetivo en concreto.

- **Memoria (*Memory*)**: almacena todo lo que conoce el agente y mantiene una visión actualizada de distintos aspectos del mundo en general. Principalmente almacena la información más reciente obtenida por los sensores y que será utilizada por las acciones en concreto.

Esto permite obtener acciones más genéricas y descarga el proceso que tiene que realizar el planificador pues es un subsistema que funciona por otro lado.

- **Sensor**: sistema a través del cual se obtiene la información del mundo necesaria para ser almacenada en la memoria del agente. Suele representar alguno de los cinco sentidos del ser humano principalmente la vista o el oído.

Estos elementos son usados por la clase GoapAgent que establece las acciones y los objetivos que seguirán al agente. No obstante, toda la implantación se realiza a nivel de código teniendo que heredar de las clases base para crear cada uno de los distintos agentes.

Respecto a la planificación empleada hace uso de un A* personalizado que almacena los nodos en una cola rápida, concretamente la cola rápida de BlueRaja disponible en GitHub.

Poniendo un especial foco en las acciones utilizadas por esta implementación emplean lo que se conoce como **ajustes** (en inglés, **Settings**) que establecen **parámetros dinámicos** que serán tomados en cuenta a la hora de ejecutar la acción en concreto. Esto permite por ejemplo usar como parámetro de la acción un valor extraído del objetivo en concreto que debe utilizarse.

En el apartado de comprobación de errores esta herramienta incluye una funcionalidad muy interesante, un **debugger** de planes de acciones que muestra el plan realizado por un agente en un instante. Se adapta automáticamente al GameObject seleccionado y muestra la información en forma de nodos en los que se aprecia el objetivo, las acciones realizadas y los distintos estados que atraviesa el agente hasta concluir en el objetivo alcanzado.

Finalmente hay que destacar que la planificación emplea **planificación hacia atrás**, es decir, el enfoque de búsqueda que parte desde el objetivo resolviendo las condiciones que van surgiendo, para eso almacena las acciones respecto a los efectos y a las propiedades a las que afectan.

3.4 Conclusiones

Muchas de las características de GOAP son compartidas a lo largo de los años, pero otras van evolucionando, cambiando y reemplazándose con el paso del tiempo. Desde comportamientos cooperativos, hasta comportamientos optimizados para usarse en conjunto con unos sistemas más complejos.

Se llega a la conclusión de que el principal foco a la hora de desarrollar GOAP es obtener una inteligencia que aparente ser más de lo que pudiera ser, a través de técnicas y optimizaciones que resulten en ofrecer una experiencia más inmersiva e interesante sin tener que planearla y desarrollarla previamente.

Las experiencias obtenidas no siempre serán las mismas aumentando la rejugabilidad y la inmersión de forma considerable con un algoritmo que se ha optimizado con una planificación A* haciendo uso de la búsqueda hacia atrás.

Capítulo 4

Diseño del Algoritmo

Se presenta de forma genérica el algoritmo GOAP para establecer una base genérica que se puede adaptar a gusto del desarrollados. Además, se explican algunas funcionalidades extra que pueden aportar bastante eficiencia al algoritmo y finalmente un ejemplo con el que comprobar que GOAP se ha implementado correctamente.

4.1 Descripción de la herramienta

El objetivo principal consiste en **implementar** un algoritmo de GOAP capaz de generar adecuadamente los distintos planes que requieren los NPC. Se definen las distintas clases y relaciones de clases necesarias para poder crear un GOAP plenamente funcional.

En primer lugar, es necesario definir cómo se representará la información almacenada en el mundo, pues a raíz de esta información se definen el resto de los elementos que componen GOAP. Para ello se puede crear una clase PropertyGroup que almacenará el conjunto de propiedades relevantes para definir los distintos estados del mundo.

Estas propiedades se compondrán principalmente de dos valores, de un identificador y de un valor. En una abstracción sencilla estos serían un String y un valor booleano quedando definido de la siguiente manera.

```
Struct Property(){
        Key::String
        Value::bool
}
```

Aunque también se puede utilizar cualquier tipo de dato como valor de la propiedad contando incluso con diferentes tipos de propiedades en caso de que sea necesario.

Una vez definida la clase PropertyGroup será utilizada como la **principal componente** del resto de clases que conforman el algoritmo de GOAP. Comenzando por los objetivos (Goal), que se conforman únicamente por un PropertyGroup que define las condiciones que deben darse para

dar por cumplido el objetivo. Contará además con una variable que indique la prioridad de dicho objetivo.

Por otro lado, la clase acción también hará uso de PropertyGroups que definen sus precondiciones y sus efectos, pero debido a que cada acción tendrá unos efectos distintos en el mundo es necesario heredar de una clase definida como base.

Esta permite añadir **funciones personalizadas** que permiten indicar si existen precondiciones y efectos procedurales, así como lo que hará la acción cuando sea ejecutada dentro de un plan creado. Además, también llevará una variable coste que servirá para influir en si una acción es preferible contra otras o no. Queda una clase similar a la siguiente.

```
Class abstract Action(){
        Preconditions::PropertyGroup
        Effects::PropertyGroup
Cost::float
        Abstract CheckProceduralEffects()::bool
        Abstract ApplyProceduralEffects()::void
        Abract Execute()::void
}
```

Todos estos elementos se interrelacionan a la hora de crear el planificador que es el eje central de GOAP. El planificador genera nodos a raíz de un nodo inicial que consiste en el estado inicial del mundo, estos nodos son generados a raíz de las distintas acciones disponibles por el agente.

Se deben identificar los **elementos** que formarán parte del planificador. A los creados anteriormente se le añaden las siguientes clases.

- **Nodo**: cada uno representará un estado del mundo a través de un PropertyGroup, es creado por el planificador y posee información relacionada con su origen, como la acción que ha dado lugar a ese estado concreto, el objetivo actual resultante de haber ejecutado dicha acción sobre el mundo y una referencia al nodo a partir del cual ha sido originado, denominado padre. También debe contener un valor que indique cual es el coste total del nodo, para que el planificador pueda escoger el de menor coste.

Class Node(){
WorldState::PropertyGroup
PreviousAction::Action
CurrentGoal::Goal
TotalCost::float
}

- **Generador de nodos**: es el encargado de generar y de seleccionar entre todos los nodos generados el siguiente que será expandido, entendiendo expandir como generar todos sus nodos hijos creados a partir de las acciones que puedan ejecutarse. Para hacerlo emplea un algoritmo de búsqueda a elección por el desarrollador.

Class NodeGenerator(){
CreateInitialNode::Node
GetNextNode::Node
AddChildToParentNode::void
}

- **Planificador**: se encarga de crear un plan capaz de satisfacer un objetivo determinado haciendo uso del NodeGenerator. Puede ser de dos tipos, de búsqueda hacia adelante o de búsqueda hacia atrás. Almacena todas las acciones disponibles por el agente y se encarga de seleccionar las acciones disponibles para cada nodo generado.

Class Planner(){
GeneratePlan::Plan
}

Un plan se conforma de un conjunto de acciones ordenados según su orden de ejecución según se ha establecido en la planificación.

Por último, debe existir un agente que sea capaz de emplear el planificador para obtener el plan de acciones que posteriormente tratará de realizar. Contará con un conjunto de acciones y de objetivos que, tras seleccionar el objetivo más prioritario hará uso del planificador para obtener el plan más adecuado en cada ocasión.

Este contará con los subsistemas necesarios para poder ejecutar las acciones establecidas y hará uso de esta inteligencia artificial conseguida empleando GOAP.

4.2 Funcionalidades añadidas

Como bien se ha establecido, el algoritmo de GOAP no tiene un estándar fijo que lo defina y son muchas las modificaciones y nuevas funcionalidades que se le puede agregar para obtener el máximo partido posible a sus ventajas.

Se ha descrito un GOAP que toma como identificadores de las reglas valores String asociados a valores booleanos, pero GOAP puede emplear la mayoría de las ocasiones diferentes tipos de valores en sus datos.

Para lograrlo se pueden extender las clases de GOAP para emplear dos tipos genéricos, uno para los identificadores y otro para los valores de manera que se pueda emplear la herramienta de la forma que mejor se prefiera.

Un ejemplo de esto sería utilizar un enum para identificar los diferentes tipos de reglas en lugar de String, y por otro lado emplear el valor genérico object para definir diferentes tipos de datos para los valores de las distintas reglas, obteniendo así un GOAP con muchas más posibilidades.

Otra funcionalidad añadida consiste en crear un sistema que sea capaz de **evaluar la importancia de un objetivo** según el estado en el que se encuentre el agente en cierto momento. Así en lugar de tener objetivos prefijados con una prioridad pues se puede establecer que estas puedan variar con el paso del tiempo, por ejemplo, aumentando en una cierta cantidad cada cierto tiempo.

Finalmente, se hace uso de una funcionalidad que toma su base en las otras explicadas, se mejora el uso de los valores de las propiedades, concretamente para las precondiciones tanto de los objetivos como de las acciones y los efectos de las acciones. Cuando los valores se tratan únicamente como **booleanos** las condiciones solo pueden **cumplirse** o **no cumplirse** al igual que los efectos solo establecen el nuevo valor sin mayor complicación.

Al emplear valores variables se pueden añadir nuevos tipos de condiciones y de efectos dependiendo del tipo de valor que haya sido empleado. De esa manera los valores comparables

(ej.: int, float, e incluso string) pueden satisfacer condiciones más genéricas y los valores numéricos pueden realizar operaciones aritméticas a la hora de aplica los efectos. Para conseguirlo se puede emplear un enum que identifique el tipo de comparación o de efecto que se va a aplicar. A continuación, se expresan las funcionalidades añadidas:

- *Condiciones*: **igual a (=),** diferente a (¡=), mayor que (>), menor que (<), mayor o igual que (>=), menor o igual que (<=).

- *Efectos*: **establecer (set),** sumar (+), restar (-), multiplicar (*), dividir (/), módulo (%).

Se observa el enum de cada tipo establecido y cuando se comprueban condiciones o cuando se aplican efectos se toma en cuenta el tipo de dato de la propiedad que va a ser comparada y finalmente se da la condición por satisfecha o se aplica el efecto específico.

4.3 Ejemplo de las torres de Hanoi

Para ilustrar mejor toda esta información se va a **definir** en qué consisten las **torres de Hanoi** y cuáles son los pasos para seguir que permitan, haciendo uso de GOAP, resolver el problema de las torres de Hanoi.

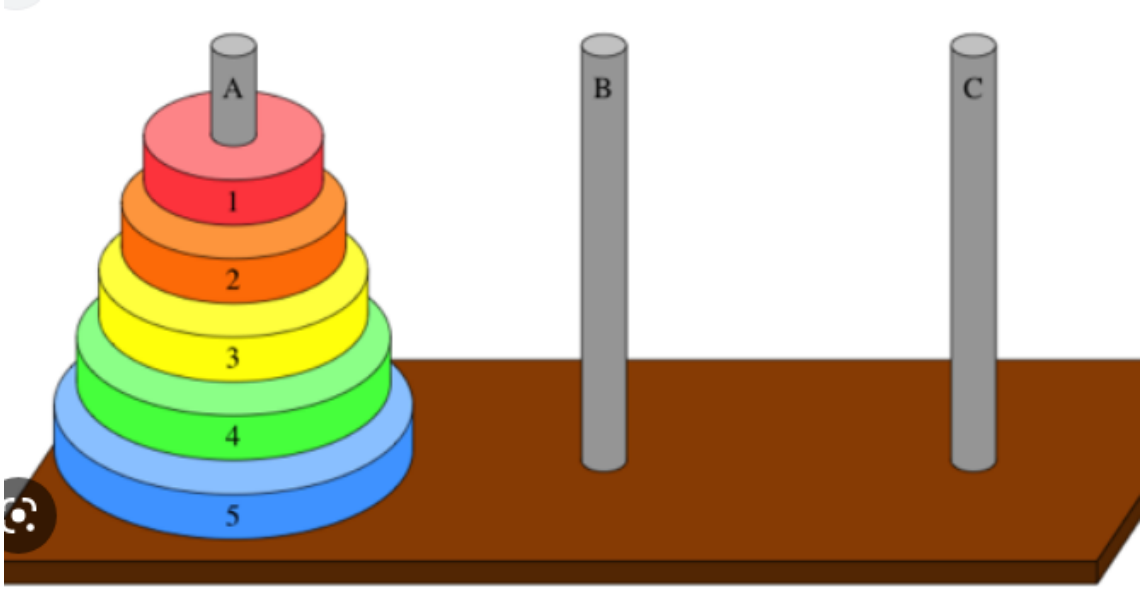


Ilustración 12 - Representación del problema de las Torres de Hanoi

Las torres de Hanoi consisten en un problema que se conforma de tres varillas denominadas A, B y C. En la primera varilla se encuentran una serie de discos apilados uno tras otro desde el 1 hasta el N, desde el más pequeño hasta el más grande, siendo N el número de discos del problema.

El objetivo de este problema consiste en lograr que todos los discos se encuentren apilados en el disco C en vez de en el disco A.

Además, hay dos reglas más que agregan cierta dificultad al problema; solo se puede mover un disco cada vez y nunca puede apilarse un disco sobre otro que sea de menor tamaño.

Los pasos para resolver cualquier problema haciendo uso de GOAP son definir las propiedades que definen el estado del mundo, establecer las acciones a realizar por el agente y definir sus objetivos.

Se comienzan por definir las **propiedades** que definen el estado del mundo, para ello se emplean claves string para la resolución de este problema, y los valores solo podrán ser verdadero o falso. Se van a definir dos realidades, las piezas que se encuentran una sobre otra con la clave On(N, M) y las piezas que se encuentran libres sin ninguna pieza encima con la clave Clear(N). Se emplea un conjunto de elementos que representan cada una de las piezas siempre que N y M no sean la misma pieza.

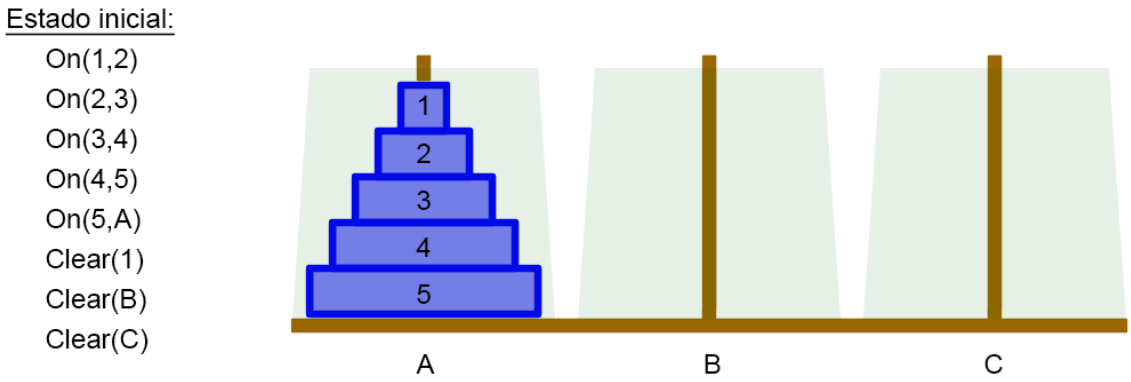


Ilustración 13 - Estado inicial del problema de las Torres de Hanoi

Así se genera un conjunto de reglas booleanas que permitirán definir cada uno de los diferentes estados del problema.

A continuación, se definen las **acciones** que serán ejecutadas por el agente. En este caso el agente solo podrá mover una pieza a cualquier otra aguja definida como Move(x,b,y) siendo "x" la pieza que se va a mover, "b" la pieza que posee debajo e "y" la pieza destino donde "x" se va a colocar.

Esto será posible siempre que ninguna de las piezas tenga otra pieza encima, esto se define con las precondiciones Clear(x) y Clear(y) y después se comprueba que "x" efectivamente está encima de "b" y que será tomada en cuenta en los efectos con la precondición On(x, b).

Para tomar en cuenta la condición clave de que la pieza sobre la que se coloca la pieza es de mayor tamaño que la que se esté movimiento se podría conseguir estableciendo como precondición las reglas de que la pieza "x" debe ser menor que la pieza "y". No obstante, serían

reglas que siempre se mantienen a lo largo de todo el problema y lo importante de GOAP es mantener en los estados del mundo la menor cantidad de información relevante posible. Por esta razón se simplificará la información del estado generando únicamente acciones Move(x,b,y) en las que "x" siempre sean de menor tamaño que "y".

Cabe mencionar que la acción deberá poseer un coste mayor que cero en caso de que se busque el plan con el menor número de acciones posibles, ya que si se le da un coste 0 el planificador puede interpretar que se pueden realizar muchas otras acciones antes de elegir aquella que más se acerque al objetivo.

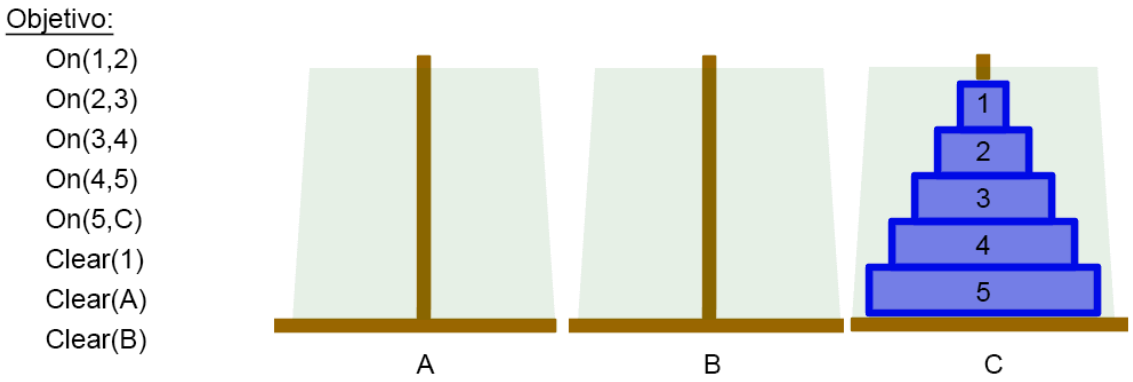


Ilustración 14 - Estado objetivo del problema de las Torres de Hanoi

Por último, se deben definir el **estado inicial** y el **estado objetivo**, estos se definen haciendo uso de las reglas establecidas previamente. El estado inicial se define empleando las reglas apreciadas en la Figura 9 y el estado final en la Figura 10 que poseen la única diferencia de que la pieza más grande se encuentra sobre la aguja C y que la A se encuentra libre, el resto de las reglas permanecen debido a las implicaciones del problema.

Ahora el agente hará uso del planificador GOAP haciendo uso de esta información para un plan de acciones capaz de resolver el problema. Si todo está en orden el plan obtenido estará formado por un total de $2^N - 1$ acciones, siendo N el número de discos empleado.

Capítulo 5

Descripción informática

Se va a describir en este capítulo todos aquellos aspectos relacionados con el trabajo realizado para este TFG en concreto. En el apartado de análisis se recopilarán los requisitos esperados tanto funcionales como no funcionales. En el apartado de diseño se mostrarán los distintos diagramas de clase y así como distintos casos de uso de la herramienta [13]. Finalmente, se describe la implementación específica realizada de GOAP para Unity.

5.1 Análisis

5.1.1 Requisitos funcionales

RF1. La base de la herramienta podrá ser implementada en cualquier aplicación C# pudiendo adaptarse a cualquier aplicación.

RF2. Se define una implementación de GOAP aprovechando funcionalidades de Unity.

RF3. Será capaz de definir estados del mundo bien diferenciados entre sí.

RF4. Las acciones podrán contar con condiciones y efectos definidos tanto con *PropertyGroups* como pudiéndose establecer de forma procedural.

RF5. Cada objetivo será definido con un *PropertyGroup* concreto y su prioridad será independiente de unos agentes a otros.

RF6. El coste de las acciones será independiente para cada tipo de agente definido.

RF7. Cada acción, objetivo y estado inicial podrá ser definido como *ScriptableObjects*.

RF8. Las reglas de los estados del mundo emplean valores tipo object pudiéndose emplear cualquier tipo de dato.

RF9. Los objetivos y las precondiciones de las acciones permiten definir **comparadores lógicos** respecto a un valor establecido.

RF10. Los efectos de las acciones permiten definir **funciones aritméticas**.

RF11. La herramienta gestiona múltiples agentes simultáneamente.

RF12. El planificador GOAP genera planes válidos continuamente y los agentes no se quedan inmóviles.

RF13. Cuando un plan de un agente falla, o termina, se genera un nuevo plan inmediatamente.

RF14. Se pueden tomar valores del objetivo actual en el planificador como parámetro para una acción en específico.

RF15. Los agentes de GOAP se definen con un componente que funciona correctamente indistintamente del *GameObject* al que haya sido adherido.

RF16. El planificador realiza la búsqueda hacia atrás para agilizar el procesamiento.

RF17. Heurísticas empleadas por el planificador modificables a gusto del desarrollador.

5.1.2 Requisitos no funcionales

RFN1. Cada plan viable no tardará más de 2s en generarse y contará con un máximo de 500 acciones.

RNF2. Se emplearán diccionarios clave valor para definir los *PropertyGroup* acelerando la comprobación de reglas.

RNF3. Las propiedades que definen los estados del mundo son establecidas con Enums para evitar errores de nombramiento y acelerar su búsqueda dentro de los diccionarios.

RNF4. Se define el tipo de dato establecido para cada valor del Enum que define los nombres de las reglas para aumentar la cohesión del proyecto.

RNF5. Los valores de las distintas reglas, condiciones y efectos de los ScriptableObjects pueden ser definidos fácilmente a través del editor de Unity.

RNF6. Depuración del planificador a través de la consola de Unity.

5.2 Diseño

A continuación, se muestran distintos diagramas que ofrecen una visión más amplia y específica de la herramienta desarrollada. Estos serán los diagramas de clase, los distintos casos de uso, así como sus respectivos diagramas y un diagrama de secuencia que ejemplifica el funcionamiento general de la herramienta. Para su formalización se empleará el estándar Lenguaje Unificado de Modelado (en inglés, Unified Modeling Language (UML)).

5.2.1 Diagramas de clase

Los diagramas de clase permiten comprender la estructura de las distintas clases creadas en un simple vistazo. Se han estructurado según los directorios en los que se encuentran cada una siguiendo un nivel de abstracción cada vez mayor. Los recuadros azules pertenecen a un nivel de abstracción menor y los naranjas son parte de la librería de Unity.

Debido al gran tamaño de estos diagramas se ubican en el Anexo I – Diagramas de clase.

5.2.2 Casos de uso

En esta sección se describen algunos casos de uso de la herramienta para poder comprobar que se pueden realizar las funciones esperadas y para aprender cómo debe ser utilizada correctamente.

UC1. Instalar UGoap en un proyecto

- **Actor principal**: Usuario
- **Precondiciones**: Poseer el archivo “UGoap.package”.
- **Resultado esperado**: UGoap es instalado con todas sus funciones disponibles.
- **Flujo de acciones**:
    1. Abrir el menú desplegable de la barra superior “Assets” y seleccionar la opción “Import Package > Custom Package”.
    2. Se abre la ventana del explorador de Windows.
    3. Encontrar el directorio donde se encuentra almacenado el archivo y abrirlo.
    4. Seleccionar todos los archivos y finalmente confirmar presionando “Import”.
- **Flujos adicionales**:

4.1. No se importan todos los archivos necesarios.

a) Eliminar por completo la carpeta “UGoap” de Assets.

b) Realizar de nuevo todos los pasos del flujo de acciones.

UC2. Definir las Property Keys y el tipo de sus valores

- **Actor principal**: Usuario
- **Precondiciones**: Haber instalado UGoap correctamente y tener un editor de texto o de código en el equipo.
- **Resultado esperado**: Las propiedades de UGoap quedan bien definidas empleando los tipos de valores indicados.
- **Flujo de acciones**:
  1. Abrir el script “UGoapPropertyManager.cs” ubicado en “UGoap/Unity”.
  2. Escribir en el enum PropertyKey los nombres de las reglas que se deseen establecer.
  3. Añadir al diccionario PropertyTypes los pares clave/valor siendo las claves los distintos nombres escritos en el paso anterior y el valor uno de los tipos definidos en el enum PropertyType.
- **Flujos adicionales**:

  3.1. Si el tipo de valor escogido es un Enum.

  a) Añadir en el diccionario EnumNames un conjunto de strings para cada propiedad definida como Enum.

UC3. Definir un estado inicial

- **Actor principal**: Usuario
- **Precondiciones**: Paquete UGoap instalado y haber definido correctamente tanto las *PropertyKeys* como sus tipos de valor.
- **Resultado esperado**: Se obtiene un *ScriptableObject* que define un estado inicial de UGoap.
- **Flujo de acciones**:
  1. Navegar por las carpetas de la ventana “Project” hasta encontrar el lugar donde se quiera crear el estado inicial.

2. Realizar clic derecho en un espacio vacío para mostrar el menú desplegable y seleccionar “Create > Goap Items > State”.
3. Dotar al estado de un nombre característico con F2.
4. Añadir todas las propiedades deseadas en el campo “Properties” seleccionando las PropertyKeys deseadas y el valor inicial que poseerán.

UC4. Definir un objetivo

- **Actor principal**: Usuario
- **Precondiciones**: Paquete UGoap instalado y haber definido correctamente tanto las *PropertyKeys* como sus tipos de valor.
- **Resultado esperado**: Obtener el ScriptableObject de la acción creada plenamente funcional y lista para su uso.
- **Flujo de acciones**:
  1. Navegar por las carpetas de la ventana “Project” hasta encontrar el lugar donde se quiera crear el objetivo.
  2. Realizar clic derecho en un espacio vacío para mostrar el menú desplegable y seleccionar “Create > Goap Items > Goal”.
  3. Dotar al objetivo de un nombre característico con F2.
  4. Añadir todas las condiciones deseadas en el campo “Properties” seleccionando las *PropertyKeys* deseadas, la función de comparación deseada y el valor inicial que poseerán.
- **Flujos adicionales**:
  4.1. Se establece una función de comparación no compatible con el tipo de valor establecido para una *PropertyKey* en concreto.
    a) Cambiar por una función de comparación compatible para evitar resultados no deseados en la planificación.

UC5. Crear y definir una acción de UGoap

- **Actor principal**: Usuario
- **Precondiciones**: Paquete UGoap instalado y haber definido correctamente tanto las *PropertyKeys* como sus tipos de valor.
- **Resultado esperado**: Obtener el ScriptableObject de la acción creada plenamente funcional y lista para su uso.
- **Flujo de acciones**:
    1. Navegar por las carpetas de la ventana "Project" hasta encontrar el lugar donde se quiera crear la clase que defina la acción deseada.
    2. Realizar clic derecho en un espacio vacío para mostrar el menú desplegable y seleccionar "Create > C# Script".
    3. Definir una clase con el nombre que identifique la acción deseada que herede de la acción base "UGoapAction" e implementar los métodos requeridos.
    4. Agregar el atributo "CreateAssetMenu" a la clase definiendo sus parámetros "filename" como el nombre de la acción y "menuName" cómo se podrá encontrar esta acción en el menú desplegable.
    5. Escribir el código que debe ejecutarse cuando la acción sea realizada en el método "Execute".
    6. Crear en el directorio que se quiera el ScriptableObject de la acción definida con clic derecho y seleccionándola en el desplegable que se ubicará donde previamente se ha establecido en "menuName".
    7. Definir el coste de la acción.
    8. Definir las precondiciones de la acción en "Preconditions" estableciendo la PropertyKey, la función de comparación y el valor con la que será comparado el estado del mundo.
    9. Definir los efectos de la acción en "Effects" estableciendo la PropertyKey, la función aritmética y el valor con el que se realizará dicha función con respecto al estado del mundo.
- **Flujos adicionales**:

    5.1. La acción posee condiciones procedurales.

a) Escribir en el método “ProceduralConditions” una función que evalúe las precondiciones necesarias para la acción.

5.2. La acción posee efectos procedurales.

a) Escribir en el método “GetProceduralEffects” una función que devuelva un PropertyGroup que defina los efectos que se han generado de forma dinámica.

6.1. No se encuentra la acción creada.

a) Revisar que el parámetro “CreateMenuAsset” posee los nombres bien puestos y las barras que definen subcarpetas en el menú desplegable son barras orientadas hacia la derecha.

7.1. La acción debería calcular su coste de forma dinámica.

a) Sobrescribir el método “GetCost” de la clase de la acción y definir una función que devuelva el coste que obtendrá la función empleando el estado actual del mundo y el objetivo próximo.

9.1. La acción posee efectos procedurales.

a) Seleccionar las PropertyKeys que se verán modificadas por los efectos procedurales en “Affected Keys”.

UC6. Crear una Entidad de UGoap

- **Actor principal**: Usuario
- **Precondiciones**: Poseer UGoap instalada correctamente en el proyecto.
- **Resultado esperado**: Entidad creada y funcional dentro del sistema de UGoap.
- **Flujo de acciones**:
  1. Crear el GameObject que se desee convertir en Entidad de UGoap.
  2. Añadir el componente UGoapEntity.
  3. Definir un nombre sí que identifique la entidad.
- **Flujos adicionales**:

3.1. Se requiere un estado inicial concreto.

a) Definir un Scriptable Object de estado que describa la situación inicial del agente en concreto.

b) Arrastrar el objeto creado a “Initial State”.

UC7. Crear un Agente funcional de UGoap

- **Actor principal**: Usuario
- **Precondiciones**: Scriptable Objects de los objetivos y acciones necesarios creados y listos para usarse.
- **Resultado esperado**: Obtener un agente de UGoap que realiza planes de forma autónoma.
- **Flujo de acciones**:
    1. Crear el GameObject que se desee convertir en Agente de UGoap.
    2. Añadir el componente UGoapAgent al GameObject.
    3. Asignar los objetivos deseados arrastrando los ScriptableObjects de los objetivos hacia “Goal Objects” y definir sus prioridades siendo las más altas las primeras para las que se buscará un plan asequible.
    4. Asignar las acciones que poseerá el agente arrastrando sus ScriptableObjects en “Action Objects”.
- **Flujos adicionales**:

    4.1. Se requiere un estado inicial concreto.

        a) Definir un Scriptable Object de estado que describa la situación inicial del agente en concreto.
        b) Arrastrar el objeto creado a “Initial State”.

A continuación, se muestra un diagrama que refleja el flujo de los diferentes casos de uso para una visión más rápida de todos ellos.

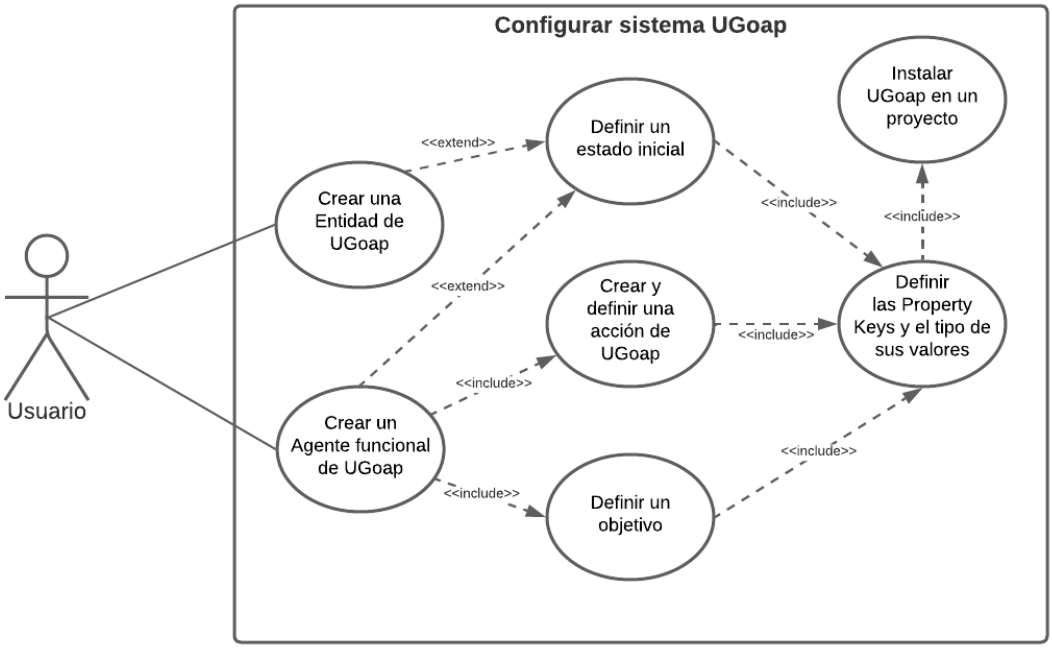


Ilustración 15 - Diagrama UML de casos de uso

5.2.3 Diagramas de secuencia y de colaboración

Para comprender mejor cómo funciona internamente el algoritmo se realiza un diagrama de secuencia que permite observar el tiempo necesario para generar un plan y un diagrama de colaboración que muestra en un rápido vistazo los diferentes elementos relacionados entre sí.

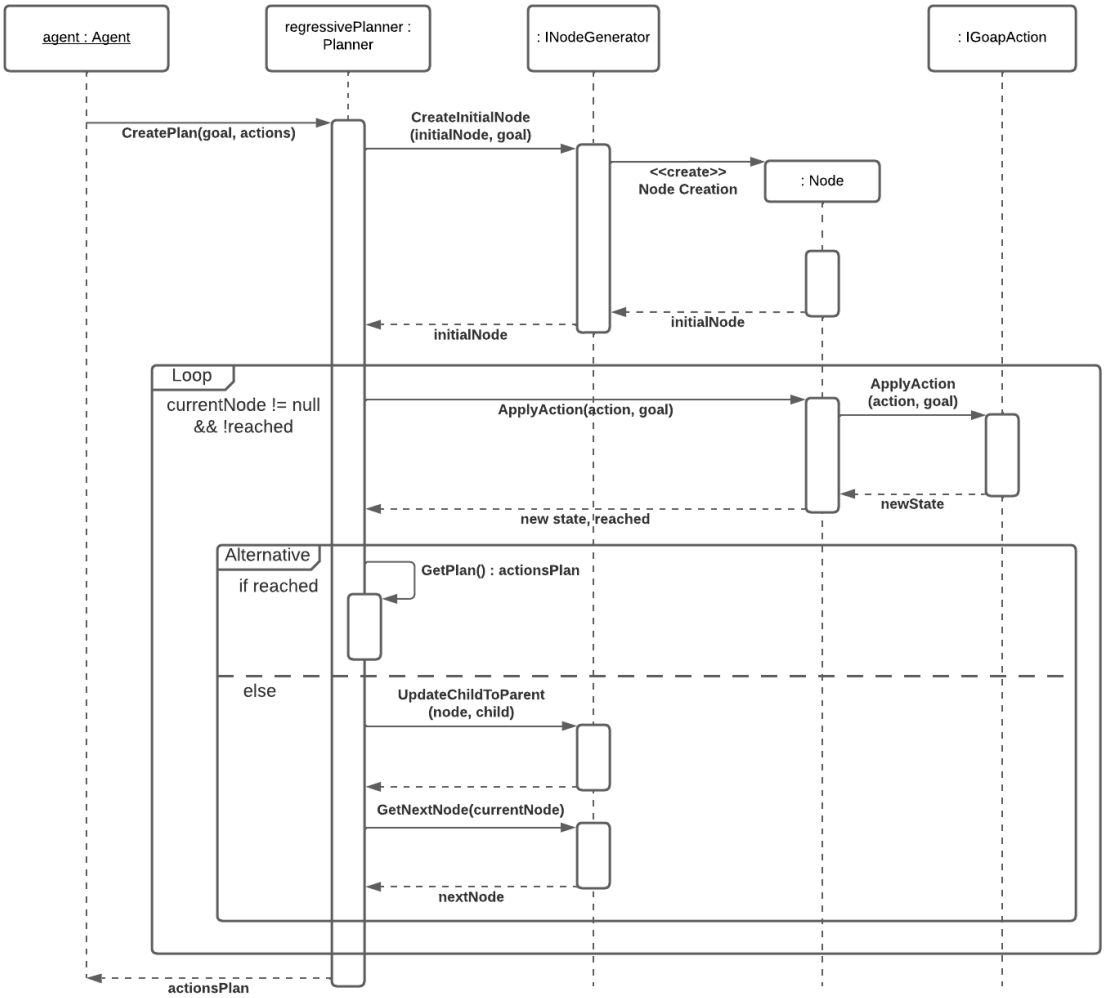


Ilustración 17 - Diagrama de secuencia de generación de un plan

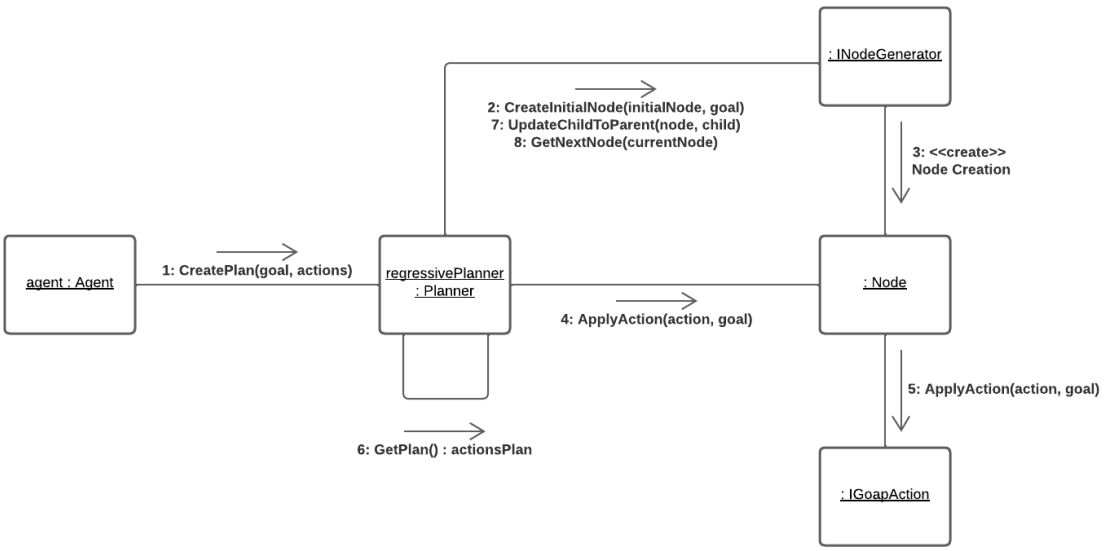


Ilustración 16 - Diagrama de colaboración de generación de un plan

5.3 Implementación

En este apartado se explica de principio a fin cómo se ha desarrollado la herramienta de UGoap hasta llegar al estado actual de la misma. Así se puede explicar el porqué de las decisiones realizadas a lo largo de su creación.

5.3.1 Estructuración inicial

Se comienza por elaborar una herramienta de GOAP desarrollada para C# independiente de Unity asegurando su correcto funcionamiento. Se implementan las clases base de PropertyGroup, Goal y Action que conforman los elementos más fundamentales con los que funciona el planificador.

A la hora de implementarse el planificador se debe hacer uso de un algoritmo de búsqueda que permita encontrar un plan de acciones que satisfaga el objetivo actual. Se escoge el algoritmo de A* que permite aplicar una función heurística para determinar lo lejos que se encuentra cada nodo generado del objetivo y que además toma en cuenta el coste de las sucesivas acciones entre sí.

Respecto a la implementación de este algoritmo es necesario gestionar dos colecciones de nodos; un almacén de nodos expandidos, que no deben volver a ser explorados, y la lista abierta, que ordena de menor a mayor los nodos hijos de los nodos previamente expandidos según el coste total obtenido.

En primer lugar, se define la lista de nodos expandidos como una tabla Hash de nodos de tal manera que nunca vayan a expandirse dos nodos iguales generando bucles innecesarios. Cada nodo del planificador se conforma por un estado del mundo característico *PropertyGroup* por lo que dos nodos serán diferentes si sus PropertyGroups lo son.

Para definir las reglas características de cada estado del mundo se hace uso de un SortedDictionary que emplea de clave un tipo genérico TKey y de valor un tipo genérico TValue. De esta manera las reglas siempre serán recorridas en el mismo orden lo que permite identificar de forma mucho más rápida si dos estados son iguales o diferentes entre sí [14] [15].

Se sobrescriben los métodos Equals y GetHashCode para poder identificar rápidamente si dos nodos son iguales o distintos. En Equals se establece una comparación exhaustiva para asegurar que todas las propiedades importantes coinciden. Cabe destacar que se deben ignorar aquellas reglas cuyo valor sea el valor por defecto de su tipo en concreto considerándose regla desinformada.

La fórmula empleada para el GetHashCode hace uso del orden propio de las distintas reglas, de un número arbitrario por el que será multiplicado el código cada regla y la función AND entre la clave y el valor que posee de esta manera quedan ligadas cada clave con su valor y se toma en cuenta el número total de reglas relevantes obteniendo una función casi única para cada PropertyGroup.

$$Hash = 18 * Hash + key.GetHashCode() \text{ ^ } value.GetHashCode()$$

Finalmente, para definir la lista abierta de nodos se hace uso de esta propiedad de manera que cada Nodo hará uso de los métodos Equals y GetHashCode de PropertyGroup. Se hace uso de un SortedSet con el objetivo de aumentar considerablemente la eficacia del planificador.

En un primer momento se empleaba una lista normal que se ordenaba cada vez que un nuevo nodo era añadido y tenía una complejidad de O(n*log(n)). Aprovechando la característica de que nunca dos nodos iguales van a almacenarse en la lista se emplea un SortedSet que posee una complejidad de O(log(n)) reduciéndose considerablemente la complejidad a mayor número de nodos. En ambos casos la extracción del nodo que se va a expandir es O(1). Así queda creado el algoritmo A* que supone el eje central del planificador de GOAP.

Respecto al agente debe ser configurado manualmente con los objetivos y acciones necesarios para generar los planes de acciones necesarios dotándolos de inteligencia. Esto se realiza desde código a través de la clase principal del programa.

5.3.2 Implementación de UGoap para Unity

Tras haber desarrollado una implementación de GOAP genérica en C# se procede a extender su funcionalidad a Unity aprovechando las características que ofrece el motor. Para ello se hacen implementaciones específicas de los componentes de GOAP para conseguir una buena integración de la herramienta.

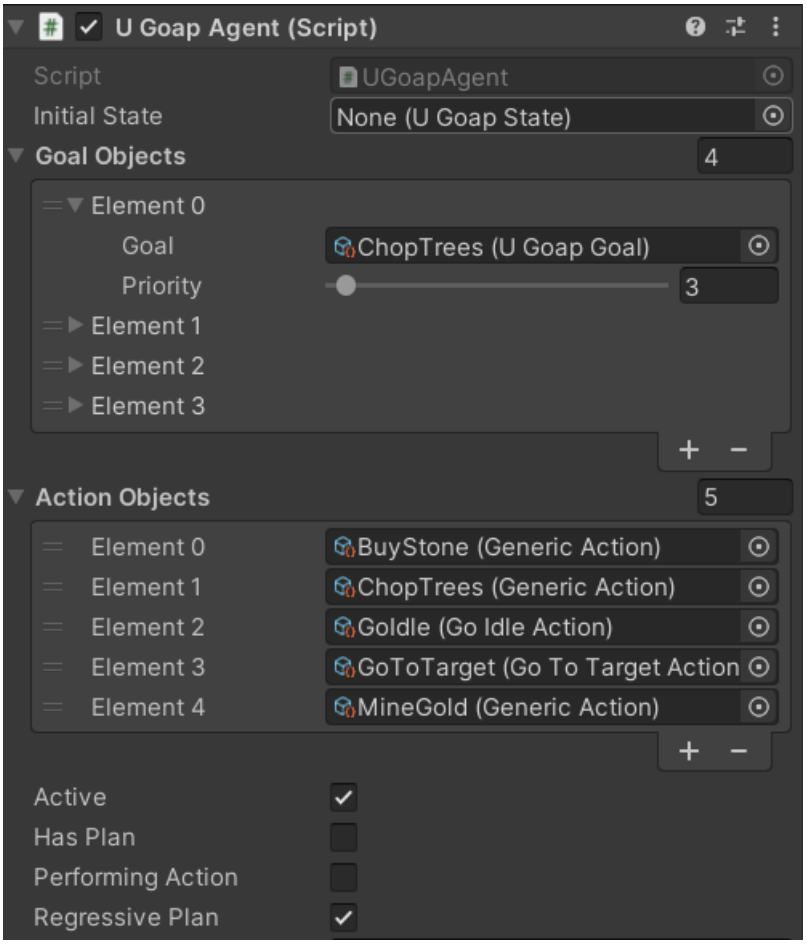


Ilustración 18 - Componente U Goap Agent en la ventana inspector de Unity

Se crea un agente de GOAP que herede de MonoBehaviour llamado UGoapAgent, de esta manera el comportamiento de agente puede aplicarse a cualquier GameObject deseado y se facilita la configuración del agente pues puede realizarse a través de la ventana del inspector de Unity.

Como se aprecia en la Ilustración 14 el agente puede configurarse arrastrando el estado inicial deseado, y los objetivos junto con sus prioridades y las acciones específicos que este agente será capaz de realizar.

Además, se sobrescriben los tipos genéricos que se usaban en la base creada de GOAP para C# siendo así el tipo TKey un enum llamado PropertyKey y TValue de tipo object de forma de que cada clave específica pueda tener un tipo distinto específico. Se indica el tipo que posee cada clave en un diccionario PropertyTypes que relaciona cada clave con un enum del tipo PropertyType.

```
[Serializable]
121 usages  IvanSploy  8 exposing APIs
public enum PropertyKey {
    WoodCount,
    StoneCount,
    GoldCount,
    Target,
    IsInTarget,
    StateOfTarget,
    IsIdle
}
```

```
[Serializable]
20 usages  IvanSploy  6 exposing APIs
public enum PropertyType
{
    Boolean = 0,
    Integer = 1,
    Float = 2,
    String = 3,
    Enum = 4
}
```

```
private static readonly Dictionary<PropertyKey, PropertyType> PropertyTypes = new()
{
    { PropertyKey.WoodCount, PropertyType.Integer },
    { PropertyKey.StoneCount, PropertyType.Integer },
    { PropertyKey.GoldCount, PropertyType.Float },
    { PropertyKey.Target, PropertyType.String },
    { PropertyKey.IsInTarget, PropertyType.Boolean },
    { PropertyKey.StateOfTarget, PropertyType.Enum },
    { PropertyKey.IsIdle, PropertyType.Boolean }
};

public static Dictionary<PropertyKey, string[]> EnumNames = new()
{
    { PropertyKey.StateOfTarget, new [] { "Reached", "Going", "Ready" }}
};
```

Ilustración 19 - Configuración de PropertyKeys y de tipos de valor

En la Ilustración 15 también se puede apreciar la forma en la que se pueden definir diferentes tipos de enums a través del diccionario EnumNames que serán fácilmente configurables en el inspector de Unity.

También se añaden diferentes tipos de comparaciones a las precondiciones de las acciones y de los objetivos, así como nuevas operaciones a los efectos de las acciones. Así se abre el abanico de posibilidades que ofrece la herramienta pudiendo definir de forma menos rígida los diferentes elementos.

```
[Serialization]
15 usages  IvanSploy  9 exposing APIs
public enum ConditionType {
    Equal,
    NotEqual,
    LessThan,
    LessOrEqual,
    GreaterThan,
    GreaterOrEqual
}
```

```
[Serialization]
15 usages  IvanSploy  9 exposing APIs
public enum EffectType {
    Set,
    Add,
    Subtract,
    Multiply,
    Divide,
    Modulo
}
```

Ilustración 20 - Definición de operaciones condicionales y operadores de efectos.

Para facilitar la definición de los elementos fundamentales de UGoap (estados, objetivos y acciones) se han realizado tres tipos distintos de ScriptableObjects; UGoapState, UGoapGoal y UGoapAction. Consisten en especificaciones de State, Goal y Action que heredan de la clase ScriptableObject lo que permite editar estos objetos desde editor y ser preconfigurados a gusto del desarrollador.

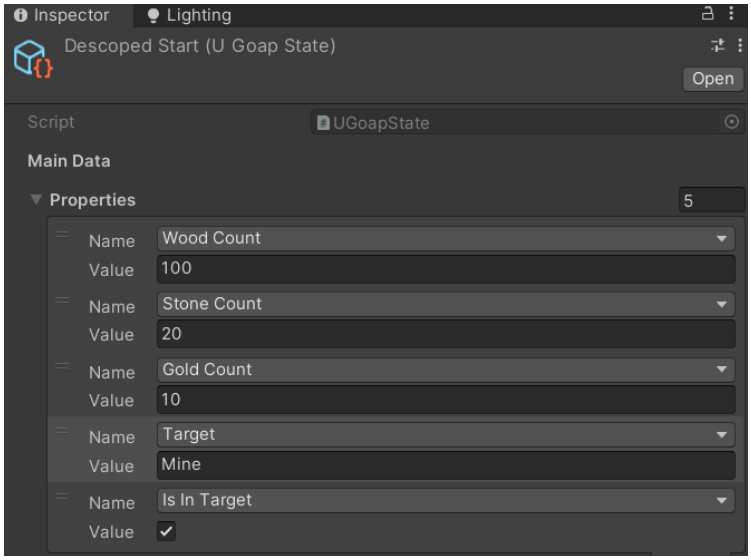


Ilustración 21 - Configuración de State mediante Inspector

Para que se muestren correctamente las distintas propiedades en el Inspector se ha necesitado modificar la forma en la que se muestra haciendo uso del atributo CustomEditor de Unity. Además de realizar una visualización customizada de cada propiedad para que se muestre dentro de una lista de elementos haciendo uso del atributo CustomPropertyDrawer.

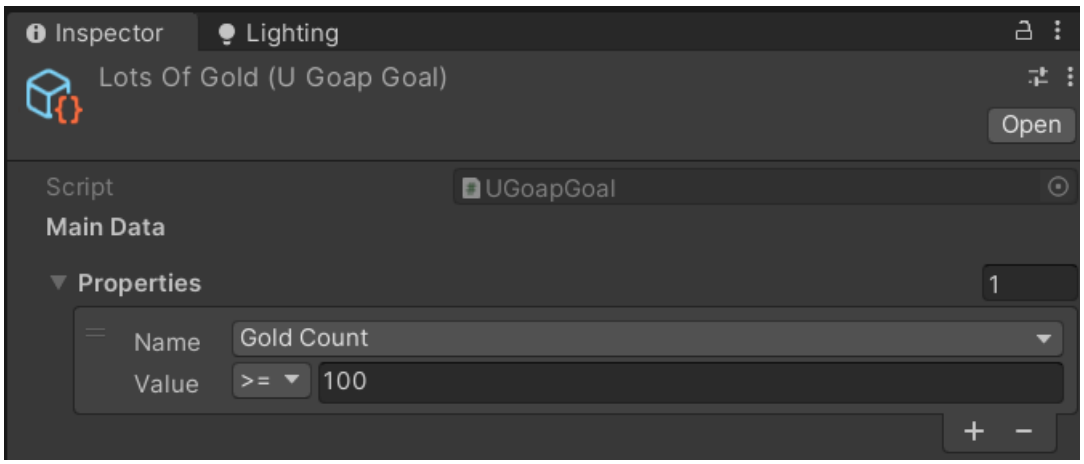


Ilustración 22 - Configuración de Goal mediante Inspector

No obstante, se necesitan crear diferentes tipos de funcionalidades para las acciones, más allá de definir un conjunto u otro de precondiciones y de efectos debido a que las acciones pueden incorporar código adicional para definir precondiciones, efectos, coste e incluso definir a nivel de código los efectos que se tendrán cuando se ejecute el plan.

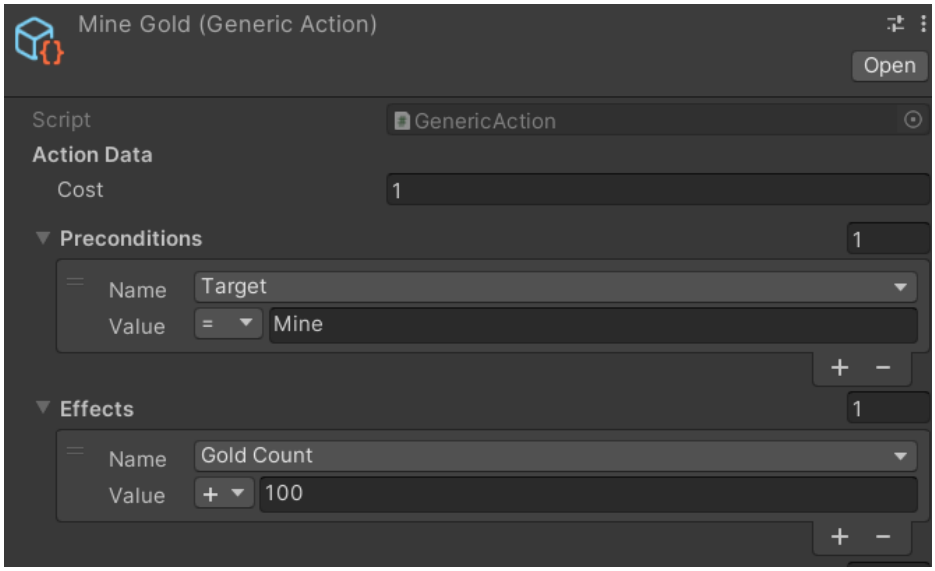


Ilustración 23 - Configuración de Generic Action a través del Inspector

Para ello las diferentes acciones heredan de la clase padre UGoapAction que a su vez es definida como un ScriptableObject. Esto permitirá configurar diferentes acciones que ejecuten el mismo código haciéndolo especificable para cada tipo de agente. Cabe destacar que si un efecto procedural desde código afecta a una clave en concreto deberá indicarse en AffectedKeys lo que permitirá al planificador regresivo encontrar esta acción como una posible solución.

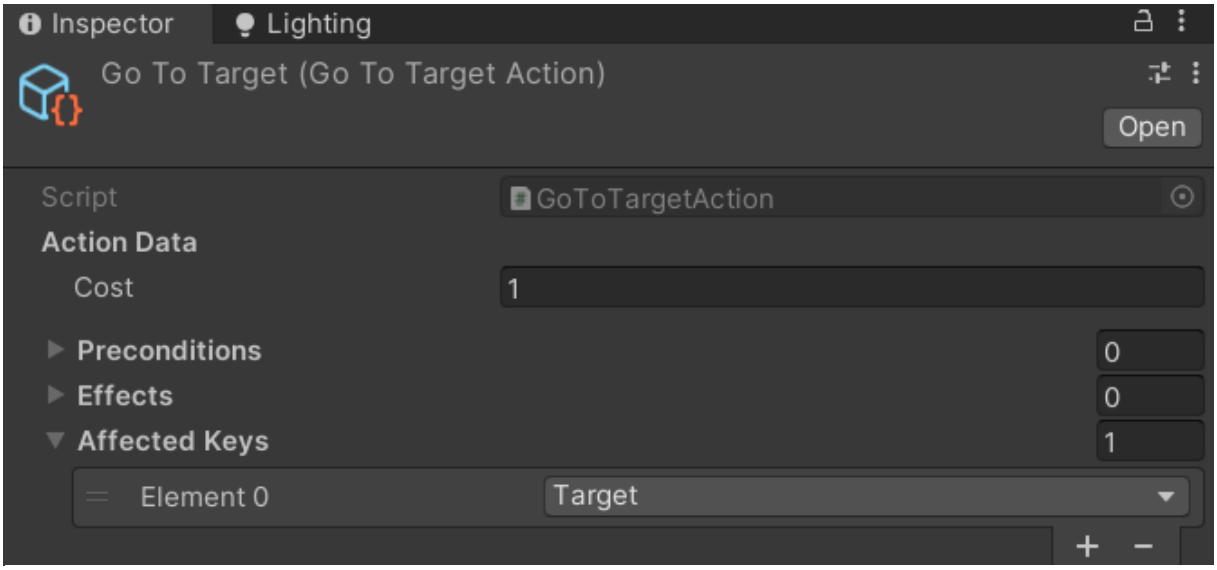


Ilustración 24 - Configuración de GoToTargetAction en Inspector

5.3.3 Optimizaciones

Finalmente se realizan optimizaciones necesarias para lograr reducir el tiempo de carga empleado. En un inicio se realiza un planificador directo que comienza buscando desde el nodo inicial y prueba todos los nodos tratando de encontrar el camino de acciones cuyo coste hasta encontrar el nodo objetivo sea menor.

Para reducir significativamente el elevado tiempo de búsqueda de un plan asequible se comienza por mejorar la heurística empleada en A*. Usar el número de condiciones incumplidas como heurística principal puede ser útil cuando los valores que pueden tomar estas reglas son true o false, pero en UGoap, al utilizarse tipos de valores más amplios como int o float puede hacerse uso de una heurística más eficiente.

Definida en UGoapData y empleada por los nodos de A* para estimar el coste hasta el nodo objetivo se define la heurística para que tome en cuenta los valores de tipo int y float midiendo la distancias hasta el valor requerido por la condición.

Hace uso del valor absoluto y la diferencia del valor actual y el valor esperado como el coste estimado la heurística para esa propiedad en concreto. En el caso de otros tipos de datos se sigue el estándar habitual, si la condición no se cumple se le suma 1 al coste estimado por cada regla incumplida.

Así el planificador es capaz de centrarse en los nodos que más se acerquen al objetivo deseado pues es capaz de cuantificar los valores de cada propiedad reduciendo considerablemente el tiempo de búsqueda.

No obstante, aunque a través de heurísticas se ha mejorado considerablemente el tiempo de ejecución de la planificación el plan generado no toma en cuenta el coste de trasladarse a través del mapa y el tiempo que supone. Para ello se le añade a la acción GoTo un coste procedural que se representará con el módulo de la distancia entre su posición y su destino.

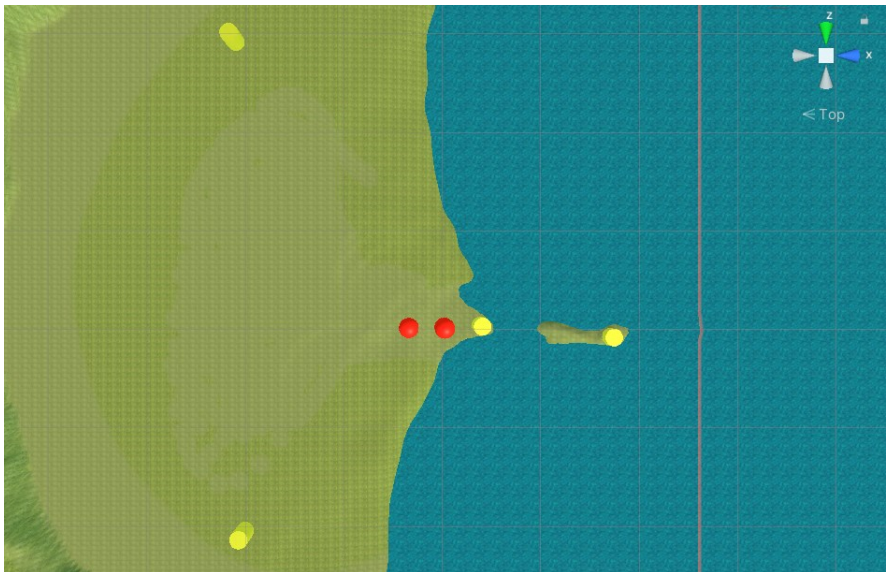


Ilustración 25 - Ubicación de entidades y agentes

Se crea un planificador capaz de realizar una búsqueda regresiva con el objetivo de minimizar el tiempo de ejecución y mejorar la secuencia de acciones obtenida.

La planificación regresiva emplea únicamente acciones cuyos efectos estén relacionados con el objetivo deseado. Por esta razón al construirse una acción se determinan las PropertyKeys para las que esa acción en concreto realizará un cambio. El objetivo de la búsqueda regresiva es conseguir que el objetivo tenga todas sus condiciones cumplidas para definir que un nodo se considera objetivo.

Debido a que UGoap dispone de tipos que constan de muchos valores se especifican ciertas reglas a la hora de establecer si una acción es válida o no lo es que expande la explicación realizada por Orkin [2].

- Si para satisfacer una condición del objetivo se debe aplicar una precondición cuya Key ya está presente en el objetivo entonces la acción no podrá aplicarse. Esto es debido a que no se puede asegurar que se vayan a poder satisfacer ambas condiciones.

- Si un efecto satisface una precondición y luego surge otra precondición similar volverá a tomarse en cuenta en el objetivo hasta que se vuelva a satisfacer. Esto debido a que en la planificación regresiva se está yendo desde el futuro hasta el presente entonces no se puede asegurar que una precondición ya satisfecha lo esté en el pasado.

5.3.4 Funcionalidades adicionales

Finalmente, con el objetivo de aprovechar al máximo la herramienta se incorporan ciertas funcionalidades que aportan flexibilidad a la hora de diseñar las acciones de UGoap.

La búsqueda regresiva no solo aporta ventajas a nivel de optimización debido a que las condiciones de los objetivos ahora se resuelven de forma inmediata, encontrándose acciones que resuelvan sus precondiciones, y se pueden conocer de antemano los valores necesarios para satisfacerlos. De esta manera, se expande la funcionalidad de los efectos procedurales de las acciones para que puedan acceder a las precondiciones del objetivo actual que permite obtener "parámetros" que simplifica la creación de acciones significativamente.

Por ejemplo, en lugar de crear una acción GoTo con cada posible destino siendo solo una la acción posible, se crea una acción que toma como parámetro el valor requerido por una PropertyKey del objetivo y se especifica como destino.

```
protected override PropertyGroup<PropertyKey, object>
    Frequently called  0+1 usages  IvanSploy *
    GetProceduralEffects(GoapStateInfo<PropertyKey, object> stateInfo)
{
    PropertyGroup<PropertyKey, object> proceduralEffects = new PropertyGroup<PropertyKey, object>();

    _target = stateInfo.CurrentGoal[PropertyKey.Target];
    proceduralEffects[PropertyKey.Target] = _target;
    return proceduralEffects;
}
```

Ilustración 26 - Ejemplo de parametrización de la acción GoToTarget

Cada agente posee un estado del mundo concreto que determina lo que conoce de su entorno, siendo propio de cada agente. Para ampliar esta funcionalidad se crean las entidades llamadas UGoapEntity conocidas por todos los agentes y accesibles en cualquier momento.

Al igual que los agentes poseen un estado del mundo que define lo que conoce del mundo las entidades también poseen un estado del mundo con lo que él conoce. La principal diferencia que existe entre las entidades y las acciones es que las acciones generan planes y son capaces de realizar acciones y tener objetivos mientras que las entidades solo actualizan su información del mundo y son empleadas por las acciones de los agentes.

De esa manera las entidades una vez creadas son agregadas automáticamente a la Working Memory, accesible a través de la Working Memory Manager (UGoapWMM). Almacenando información relevante como su nombre, posición, dirección o incluso su UGoapEntity pudiendo acceder a su estado del mundo.

Finalmente, al igual que los agentes, las entidades se crean a través del componente UGoapEntity y se crea añadiéndole un estado inicial si es necesario y un nombre que lo identifique que será buscado por las acciones que requieran usar esta entidad.

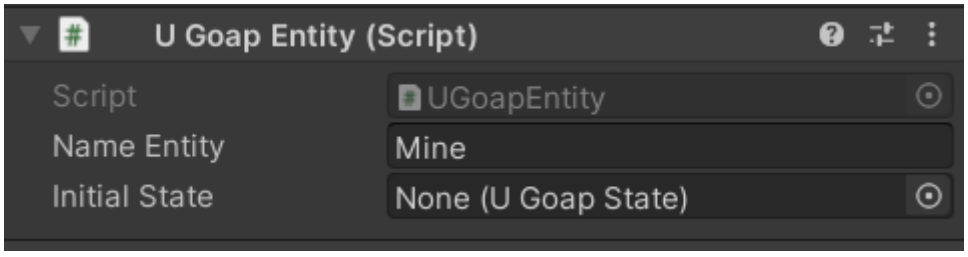


Ilustración 27 - Configuración de una entidad a través del inspector

Capítulo 6

Validación

6.1 Resultado final

Se obtiene una herramienta GOAP, de nombre UGoap, implementada de forma genérica en C#, y extendida para Unity permitiendo aprovechar las ventajas del motor gráfico. Esta se puede instalar fácilmente como un paquete de Unity y permite crear estados, objetivos y acciones fácilmente configurables a través del uso de ScriptableObjects.

También posee dos componentes principales que permiten hacer uso de la herramienta, UGoapAgent para definir un agente capaz de realizar sus planes de acciones y UGoapEntity para definir elementos relevantes para la planificación en la simulación. Estos se configuran haciendo uso de los ScriptableObjects previamente creados.

Se abstrae de la programación en mayor parte a excepción de las acciones UGoapActions que implementan su funcionalidad a través de clases heredadas permitiendo relacionarse con otros componentes de Unity, así como la posibilidad de definir precondiciones y efectos procedurales que permiten definir especificaciones más complejas y a la vez más fáciles de implementar.

Así se obtiene la capacidad de tener NPCs con una inteligencia aparentemente realista alejándose de la complejidad que supone diseñar y especificar cada posible situación que pueda darse entre diferentes acciones.

Para apreciar las características de esta herramienta se realiza una comparación con las características que trae una herramienta similar como ReGoap.

Tabla 2 - Comparativa entre la herramienta ReGoap y la herramienta desarrollada UGoap

Herramienta	Definición de elementos	Definición de planificador	Uso de operadores	Uso de concurrencia	Debug
ReGoap	Componentes de Unity	Componente de Unity	Mediante código procedural	Threads de .NET	Ventana de Unity
UGoap	Scriptable Objects	Clase estática – No requerido	Incorporados en Inspector	No	Consola

Como se puede apreciar en la Tabla 2 la herramienta de ReGoap puede resultar más compleja a la hora de definir los diferentes elementos. Esto debido a que depende de los componentes que pueden llegar a sobrecargar significativamente la ventana del Inspector de Unity llegando a saturar de información. Por el contrario, el uso de Scriptable Objects por parte de UGoap permite mantener la información separada y accesible en caso de que sea necesario, además de permitir la reutilización de objetivos y acciones para diferentes tipos de agentes.

También es más sencillo parametrizar las acciones debido a la selección de operadores y valores en las propiedades a través del Inspector lo que resulta en una implementación más fácil e intuitiva que la que frece ReGoap. A excepción, de parámetros externos al estado del mundo, que podrían facilitar los efectos procedurales, pero en UGoap deben definirse desde código.

Por otro lado, la incorporación de Threads en ReGoap optimiza considerablemente la búsqueda de planes sobre todo cuando múltiples agentes se encuentran a la vez planeando, agilizando la respuesta por parte del planificador. La inclusión de una ventana de Debug por parte de ReGoap sirve de gran ayuda para visualizar rápidamente el plan realizado por el agente seleccionado, en contraposición con los Logs emitidos por UGoap por cada agente que pueden llegar a sobrecargar la consola.

Además, respecto a la eficiencia se aprecian ciertas diferencias respecto a la implementación pues en ReGoap para almacenar los nodos de la lista abierta se hace uso de una PriorityQueue

mientras que en UGoap se ha decidido hacer uso de un SortedSet ya que nunca deberían existir nodos repetidos en las listas, y si ese es el caso, se actualizan los costes en caso de que se haya encontrado un camino más rápido hasta alcanzarlo.

6.2 Experimentación

Se realizan dos ejemplos haciendo uso de esta herramienta con el objetivo de verificar su correcto funcionamiento y también para poder apreciar su alcance y flexibilidad. El primer ejemplo se crea para verificar el funcionamiento más básico UGoap mientras que el segundo ejemplo, un poco más complejo, verifica su capacidad de ser empleada en aplicaciones de tiempo real como lo podría ser un videojuego, centrándose en la implementación realizada para Unity.

6.2.1 Implementación de las Torres de Hanoi

Se resuelve el problema de las Torres de Hanoi haciendo uso de la implementación genérica de UGoap para C#. Se emplean strings para definir las claves de las propiedades y todas tendrán valores booleanos para definir el escenario. Esto da lugar a que el número de acciones posible se eleve considerablemente por cada posible parámetro de la acción de mover siendo las precondiciones y los efectos más estrictos.

```
Key: CLEAR(PIECE1) | Valor: True
Key: CLEAR(ROD2) | Valor: True
Key: CLEAR(ROD3) | Valor: True
Key: ON(PIECE1,PIECE2) | Valor: True
Key: ON(PIECE2,PIECE3) | Valor: True
Key: ON(PIECE3,PIECE4) | Valor: True
Key: ON(PIECE4,ROD1) | Valor: True
```

```
Key: CLEAR(PIECE1) | Valor: True
Key: CLEAR(ROD1) | Valor: True
Key: CLEAR(ROD2) | Valor: True
Key: ON(PIECE4,ROD3) | Valor: True
```

Ilustración 28 - Objetivos inicial y final de Torres de Hanoi para 4 discos y 3 varillas

Se define el estado inicial del problema y el objetivo al que se desea llegar, empleando las claves CLEAR y ON, la primera indicando que la pieza no tiene ninguna otra encima y la segunda qué pieza se encuentra encima de qué otra pieza.

Se crean todas las acciones posibles para el escenario de N discos y M varillas obteniéndose, por ejemplo, un total de 90 acciones si se emplean 4 discos y 3 varillas o 150 acciones para 5 discos y 3 varillas.

```
MOVE(PIECE1,PIECE2,ROD2)
MOVE(PIECE2,PIECE3,ROD3)
MOVE(PIECE1,ROD2,PIECE2)
MOVE(PIECE3,PIECE4,ROD2)
MOVE(PIECE1,PIECE2,PIECE4)
MOVE(PIECE2,ROD3,PIECE3)
MOVE(PIECE1,PIECE4,PIECE2)
MOVE(PIECE4,ROD1,ROD3)
MOVE(PIECE1,PIECE2,PIECE4)
MOVE(PIECE2,PIECE3,ROD1)
MOVE(PIECE1,PIECE4,PIECE2)
MOVE(PIECE3,ROD2,PIECE4)
MOVE(PIECE1,PIECE2,ROD2)
MOVE(PIECE2,ROD1,PIECE3)
MOVE(PIECE1,ROD2,PIECE2)
```

```
Tiempo total: 95
Objetivo: 1
Total de acciones del plan: 15
Key: CLEAR(PIECE1) | Valor: True
Key: CLEAR(ROD1) | Valor: True
Key: CLEAR(ROD2) | Valor: True
Key: ON(PIECE1,PIECE2) | Valor: True
Key: ON(PIECE2,PIECE3) | Valor: True
Key: ON(PIECE3,PIECE4) | Valor: True
Key: ON(PIECE4,ROD3) | Valor: True
```

Ilustración 29 - Resultado de planificación de Torres de Hanoi para 4 discos y 3 varillas

Tras ejecutar la herramienta de UGoap se obtiene un plan de acciones adecuado conformado por 15 acciones, por lo que se verifica el funcionamiento del planificador ya que como se ha mencionado en el capítulo 5, el número mínimo de acciones para solucionar las torres de Hanoi es de $2^N - 1$. Se concluye el ejemplo añadiendo que este plan se ha realizado en un total de 95 ms con planificación hacia adelante.

6.2.2 Implementación de poblado en Unity

Con el objetivo de abordar casi todas las funcionalidades que ofrece la herramienta se prepara un ejemplo más elaborado que el de las Torres de Hanoi que sirva para demostrar y pulir el potencial de UGoap tanto en términos de diseño y flexibilidad como de eficiencia y escalabilidad.

En este ejemplo se diseña un pequeño poblado conformado por agentes que poseen el papel de ciudadanos. Estos podrán elegir cómo aportar recursos a la comunidad para posteriormente vender estos recursos obtenidos y finalmente emplear el dinero conseguido en comida, bebida y ocio. Se aprecia así la creación de un sistema cerrado, en el que los ciudadanos dedicarán su

tiempo a trabajar o a tiempos de ocio o de alimentación pudiendo apreciar las decisiones que realiza cada uno de ellos a lo largo del tiempo.

Así se han definido dos tipos de agentes, cada uno con sus objetivos y sus acciones concretas:

- **Civil**: realiza el ciclo descrito anteriormente, trabaja para conseguir dinero que emplea en alimentarse y en disfrutar de la vida para volver al trabajo. Aunque lo realizará al mejor coste posible, además, pondrá su supervivencia por encima de todo.
- **Obrero**: realiza únicamente acciones de trabajo por lo que nunca dedica tiempo a ocio. También tiene instinto de supervivencia, pero posee una ligera probabilidad de convertirse en peligro público debido al cansancio mental.

Se comienzan por definir las PropertyKeys del ejemplo junto con sus tipos de valor en la clase UGoapPropertyManager:

- **Target**: de tipo string, define **dónde se encuentra** el agente referenciando al nombre de un UGoapEntity.
- **Seeds**: de tipo int, define el número de **semillas** recolectadas por el agente.
- **Fish**: de tipo int, define el número de **peces** pescados por el agente.
- **Money**: de tipo float, define el **dinero** que posee el agente.
- **Happiness**: de tipo float, define el **índice de felicidad** que posee el agente.
- **Fatigue**: de tipo float, define el **índice de cansancio** que posee el agente.
- **MentalState**: de tipo enum, con valores **"Good"**, **"Tired"** y **"Burnout"**, define si el agente se encuentra **saludable** mentalmente, **cansado** o está **sobresaturado**.

A continuación, se crean tres objetivos usados en el escenario de ejemplo, su definición se puede estudiar en la Ilustración 30:

- **Seek Happiness**: objetivo que prioriza la obtención de felicidad y que trata de mantener una fatiga más reducida.
- **Seek Work**: objetivo que prioriza el trabajo duro, tratando de conseguir más dinero y permitiendo una fatiga mucho más elevada.
- **Survive**: objetivo prioritario que trata de mantener al agente en un estado mental saludable.

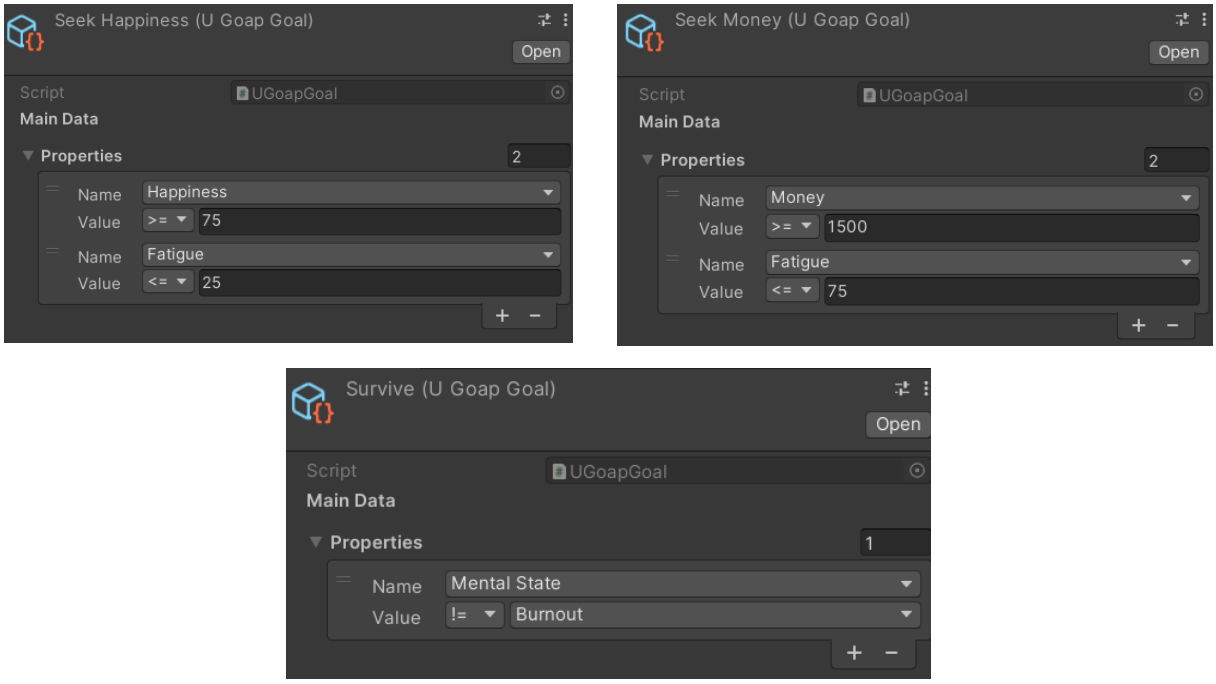


Ilustración 30 - Definición de los objetivos del ejemplo del poblado

Se procede a definir las acciones empleadas por los distintos agentes, se han creado un total de **4 acciones específicas**, cada una con implementación específica y **6 acciones genéricas** que únicamente hacen uso del estado del mundo sin código adicional.

Con respecto a las acciones específicas:

- **WalkToTarget**: acción que permite desplazarse hasta un punto en específico con velocidad intermedia.
- **RunToTarget**: similar a la anterior con la diferencia de que se desplaza a máxima velocidad, requiere encontrarse a más de 10 metros del objetivo para poder emplearla.
- **SellAllFish**: da al ayuntamiento todo el pescado que posee y a cambio recibe una interesante suma de dinero.
- **SellAllSeeds**: vende todas las semillas encontradas al ayuntamiento y a cambio recibe una pequeña compensación.

Con respecto a las acciones genéricas:

- **AttendTherapy**: acude al hospital para recibir atención médica, soluciona problemas de fatiga y convierte el estado mental del agente en bueno.
- **IntensiveTherapy**: acción de mayor coste que la anterior que ofrece al agente un tratamiento intensivo en el hospital para solucionar sus problemas de fatiga, felicidad y de salud mental.
- **DrinkCoffee**: el agente se toma un café en la cafetería recuperando felicidad, disminuyendo la fatiga a cambio de una cierta cantidad de dinero.
- **WatchMovies**: similar a la anterior, el agente ve una película en el cine del pueblo recuperando bastante felicidad y reduciendo mucha fatiga a cambio de una considerada suma de dinero.
- **GoFishing**: el agente decide trabajar pescando peces en el mar, obteniendo así peces en contraposición de aumentar la fatiga.
- **LookForSeeds**: se recolectan semillas por el bosque a cambio de aumentar ligeramente la fatiga.

Una información más detallada acerca de estas acciones se puede encontrar en el Anexo II – Acciones Detalladas.

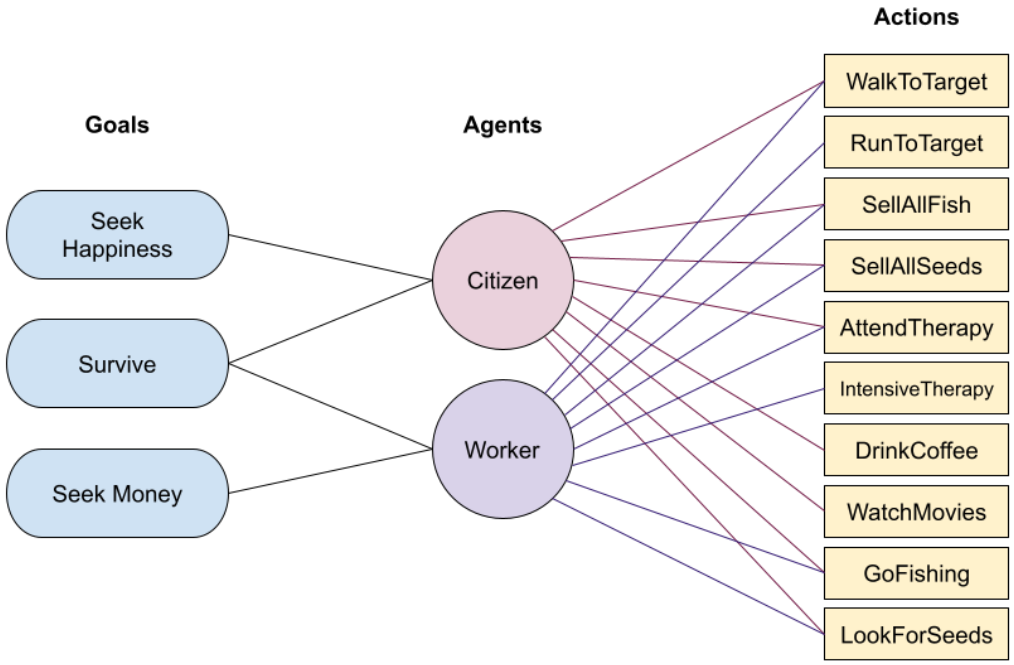


Ilustración 31 - Relación entre agentes, objetivos y acciones en el ejemplo

Los dos tipos de agentes creados poseerán algún objetivo diferente y harán uso de diferente conjunto de acciones tal y como se puede apreciar en la Ilustración 31.

Una vez definido el ejemplo y configurados todos sus elementos se procede a ejecutar la simulación para comprobar su funcionamiento. Los agentes comienzan a realizar sus planes automáticamente y se desplazan por el entorno dotándolo de **vida** y sensación de **inteligencia**.

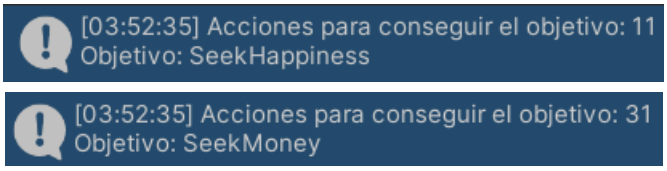


Ilustración 32 - Planes generados para el ejemplo del poblado

Además, se realizan pruebas para uno, dos y más agentes comprobando la capacidad del programa de gestionar múltiples planificaciones simultáneas y, con capacidad de mejora, se consigue en mayor medida para planes cortos.

Se llega a la conclusión de que el planificador puede ser mucho más **optimizado** para evitar la mayoría de los **tiempos de carga** que ocurren cuando las planificaciones son demasiado extensas. Esto se podrá conseguir haciendo uso de **podas**, **búsquedas por horizonte** e incluso **programación concurrente**.

Ilustración 33 - Muestra visual del ejemplo en funcionamiento

Capítulo 7

Conclusiones

7.1 Logros alcanzados

Se evalúa el éxito alcanzado con respecto a los objetivos planteados al principio de la realización del proyecto, analizando los distintos apartados desarrollados en esta memoria.

- Realizar un estudio de los antecedentes y las aplicaciones actuales de la planificación de acciones orientada a objetivos.

En el capítulo 1.2 "Estado del arte" se realiza una recopilación de los antecedentes de GOAP en el ámbito de los algoritmos GOB, desde STRIPS, apreciando algunas variaciones de GOAP e incluso comparándose con HTN, y de las diferentes aplicaciones actuales se ha hablado en el capítulo que comparte este mismo nombre, el capítulo 1.3 "En la actualidad".

Además, en el capítulo 3.2 "Casos de estudio" se aprecian distintas implementaciones realizadas por videojuegos en los últimos años y en el capítulo 3.3 "Análisis de REGOAP" se analiza una implementación de la herramienta de código abierto y disponible para todo el mundo.

Se destaca también las numerosas referencias a distintos documentos, libros y conferencias relacionados con la herramienta reflejadas en la bibliografía.

Por lo tanto, se considera que este objetivo ha sido **cumplido satisfactoriamente**.

- Definir en que consiste GOAP y extraer las características más usadas en la herramienta comparando diferentes implementaciones entre sí.

Se describe en el capítulo 3.1 "Qué es GOAP" los elementos principales que conforman la herramienta, así como sus características más importantes definidas por su creador Orkin. Se complementa con la información del capítulo 4.1 "Descripción de la herramienta" que contiene la información necesaria para realizar una implementación propia de la herramienta.

En el capítulo 3.2 "Casos de estudio" se describen diversas implementaciones de GOAP que se toman en cuenta en la implementación de la herramienta y en el capítulo 3.3 "Análisis de REGOAP" se analiza una implementación de GOAP que luego es comparada con la herramienta desarrollada en el capítulo 6.1 "Resultado final" destacando las diferencias positivas y negativas la una respecto a la otra.

Se considera así el objetivo **cumplido satisfactoriamente**.

- Diseñar una herramienta de GOAP funcional y eficiente.

Se describe en el capítulo 4.1 "Descripción de la herramienta" cómo debe realizarse una implementación genérica de la herramienta y se expande esta información con funcionalidades adicionales en el capítulo 4.2 "Funcionalidades añadidas" que pueden aportar mayor funcionalidad y eficiencia al algoritmo.

Además, en el capítulo 5.1 "Análisis" se recopilan una serie de requisitos funcionales y no funcionales que debe poseer la herramienta de GOAP que se pretende desarrollar para obtener una herramienta usable y eficiente.

Para cumplir con esos requisitos funcional se definen en el capítulo 5.2.1 "Diagramas de clase" los diagramas de clase que describen la interrelación de los diferentes objetos que necesita la herramienta para funcionar, se presentan visualmente estos diagramas en el Anexo I "Diagramas de Clase". Afianzando la relación entre los diferentes elementos en el capítulo 5.2.3 "Diagramas de secuencia y de colaboración" describiendo es espacio y tiempo el funcionamiento del planificador de acciones.

Finalmente, se describe cómo se debe emplear la herramienta en el capítulo 5.2.2 "Casos de uso" describiendo los pasos a seguir para hacer uso de sus funcionalidades.

Se concluye que se ha realizado un diseño adecuado completando el **objetivo satisfactoriamente**.

- Implementar una herramienta de GOAP de usable y reactiva para Unity.

El capítulo 5.3 "Implementación" se explica el desarrollo de la herramienta desde su estructuración inicial, que posee un carácter más general pudiendo adaptarse a cualquier entorno que haga uso de C# como lenguaje hasta la incorporación de esta herramienta a Unity como un *package* instalable en cualquier proyecto de este motor.

Se explica también el desarrollo de ciertas funcionalidades adicionales que permiten aumentar la usabilidad de la herramienta para el desarrollador, pero se considera que no se han implementado todas las posibles mejoras en cuanto a eficiencia que podría incorporar el planificador mejorando su reactividad (puede quedarse más tiempo del necesario razonando un plan), por lo que queda como tarea pendiente para más adelante.

Se considera este objetivo **cumplido en gran medida**, pero con bastante margen de mejora con respecto a la reactividad de la herramienta desarrollada.

- Diseñar e implementar ejemplos que permitan validar la herramienta.

Se diseña un ejemplo genérico en el capítulo 4.3 "Ejemplo de las torres de Hanoi" que permite comprobar el funcionamiento de la herramienta de GOAP para cualquier aplicación desarrollada.

Adicionalmente se implementa este mismo ejemplo para UGoap en el capítulo 6.2.1 "Implementación de las Torres de Hanoi" donde también se evalúan los resultados obtenidos verificando el correcto funcionamiento de la herramienta genérica.

También se comprueba el funcionamiento de la herramienta específica para Unity en el capítulo 6.2.2 "Implementación de poblado en Unity" donde se describen las diferentes inteligencias artificiales planteadas y el funcionamiento de todas ellas en conjunto.

Así se considera que el objetivo ha sido **cumplido satisfactoriamente**.

En resumen, se considera que cuatro objetivos han sido cumplidos y uno de ellos cumplido en gran medida, pero con bastante capacidad de mejorar en el futuro.

7.2 Lecciones aprendidas

La investigación y recopilación de información de la planificación de acciones orientada a objetivos seguida de su posterior desarrollo e implementación de una herramienta ha concluido en la realización de un trabajo **más que satisfactorio** por parte del alumno. Se ha adquirido una **visión** mucho más **amplia** en el ámbito de las inteligencias artificiales y se han afrontado **numerosos retos** hasta alcanzar el resultado esperado.

Con respecto a la investigación realizada se destaca el uso de **conferencias** que han servido de gran ayuda pues han servido para obtener información actualizada de GOAP ya que la mayoría de los artículos acerca de este algoritmo abarcan principalmente los primeros años en los que surgió. Esto ha enseñado al alumno nuevas formas de obtener **fuentes de información** y ampliar el rango de búsqueda a la hora de realizar una investigación.

A la hora de realizar la implementación de la herramienta se considera que se podría haber llegado al resultado esperado en **menos tiempo** del que al final ha supuesto debido a que se ha tratado de implementar la aplicación sin tener suficientemente claro cómo se quería realizar el desarrollo. Se considera que se debe centrar mayor esfuerzo en la **investigación previa** con el objetivo de alcanzar la versión deseada con más rapidez ya que se ha tenido que rehacer gran parte del código que al final era necesario que funcionara de manera distinta.

También se ha aprendido a aplicar correctamente las **interfaces** de programación, para poder implementar los dos tipos de búsquedas, la hacia delante y la hacia atrás y se ha separado correctamente el código para **evitar dependencias innecesarias**, así los elementos base nunca poseen referencias al planificador y todo queda encapsulado dentro de su propio nivel. Mencionar respecto a la organización del código que hubiera estado mejor haber abstraído la **generalización de los tipos base** de la aplicación para evitar repetir constantemente en el código los tipos específicos,

Se destaca el **aprendizaje** obtenido acerca de las funciones que ofrece Unity con el objetivo de aumentar la **usabilidad** de la aplicación, haciendo uso de los componentes que permiten modificar la forma en la que se muestran los datos de un objeto en la ventana de Inspector y el uso de ScriptableObjects para configurar los distintos elementos almacenando la información

sin necesidad de tener que definirlo a través de código. Aprender acerca de estas funcionalidades ha aumentado los conocimientos adquiridos respecto a Unity y respecto a herramientas que permiten ofrecer una mayor usabilidad a los desarrolladores de una aplicación.

Finalmente, respecto a la eficiencia de la herramienta se considera que se podría haber **paralelizado el planificador** para agilizar la búsqueda de los planes, sobre todo cuando múltiples agentes realizan esta búsqueda al mismo tiempo. Además, se podría haber ofrecido una alternativa más visual que permita solucionar errores derivados del diseño del uso de esta herramienta, pues actualmente la única forma de comprobar estos errores es a través de observar el camino de acciones obtenido a través de la consola.

7.3 Líneas futuras

Se han alcanzado los objetivos propuestos, pero aún quedan muchas otras funcionalidades que se puedes desarrollar para aumentar las posibilidades que esta herramienta puede llegar a ofrecer. Estas quedan pendientes para implementarse en un futuro.

Para ofrecer un **mayor control** de la planificación y del estado concreto de cada agente en cada momento se podría desarrollar una ventana visual en Unity que a través de nodos haciendo uso de EditorWindow como lo realiza ReGoap. Se adaptaría a la implementación realizada en UGoap y permitiría tener un mejor manejo y conocimiento de lo que está ocurriendo en la simulación permitiendo detectar errores y solucionar comportamientos no deseados de forma mucho más sencilla.

Con respecto a la **eficiencia del algoritmo** se puede mejorar la planificación repartiendo la carga de trabajo entre varios frames, con el objetivo de aumentar la reactividad de la aplicación y evitar un descenso en los frames. Otra medida que se puede realizar para optimizar la aplicación sería repartiendo el trabajo entre diferentes hilos de forma concurrente, para ello habría que hacer que las distintas clases sean *thread safe*.

También se podría explorar la **comunicación entre distintos agentes** para resolver problemas en conjunto tal y como se expresó en la conferencia de Square Enix de 2023 [7]. En la que se

crea un sistema de intercambio de mensajes en el que los agentes pueden esperar una respuesta de otros agentes mientras realizan una planificación.

Además, se podría realizar un **mejor uso de las entidades** internándolas dentro del planificador para conseguir acciones que se ejecuten únicamente en determinados contextos o relacionados a estas entidades. Es decir, incorporar a las condiciones y efectos la posibilidad de afectar a otra entidad de forma intrínseca a la planificación, pues actualmente se debe realizar de forma externa con los efectos procedurales [16].

A esto último también se le podría añadir la **implementación de sensores** capaces de percibir información acerca del mundo simulando alguno de los cinco sentidos humanos. Actualmente todos los agentes tienen pleno conocimiento del mundo de juego, con la adición de estos sensores

Finalmente, se podría agregar la posibilidad de que UGoap fuera **capaz de aprender** mejorando así las planificaciones sin que requiera intervención del desarrollador. Actualmente el funcionamiento de la planificación depende principalmente de las configuraciones y el diseño previo realizado por el desarrollador de los objetivos y las acciones que el agente es capaz de realizar. El aprendizaje podría consistir en que el agente, dependiendo de los planes que va realizando y los objetivos que realmente se terminan por cumplir, podría variar los costes de ciertas acciones según su porcentaje de éxito para preferir dinámicamente acciones que tienen mayor probabilidad de cumplir con el objetivo deseado.

Las **posibilidades del aprendizaje son muy variadas**, también se podría lograr que el agente sea capaz de observar los planes realizados por otros agentes y que esto influya en su toma de decisiones, o incluso, que sea capaz de aprender de las decisiones tomadas por el jugador, o el enemigo, dado el caso. De esta manera se terminaría por conseguir un algoritmo de inteligencia artificial muy completo capaz de ser reactivo y de aumentar la inmersión del jugador en el entorno de juego.

Bibliografía

[1] I. Millington, AI for Games, Third Edition, 2019.

[2] J. Orkin, Applying Goal-Oriented Action Planning to Games, 2004.

[3] N. J. N. Richard E. Fikes, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, 1971.

[4] J. Orkin, Three States and a Plan: The A.I. of F.E.A.R., 2006.

[5] G. Wissing, *Multi-agent planning using HTN and GOAP,* 2007.

[6] S. Girard, «Postmortem: AI action planning on Assassin's Creed Odyssey and Immortals Fenyx Rising,» Game Developer, Quebec, 2021.

[7] Square Enix CO., LTD., «Multi-Agent Cooperation With GOAP,» de *Let Your Agents Plan Together*, San Francisco, 2023.

[8] Bytehide, «bytehide.com,» 3 Febrero 2022. [En línea]. Available: https://www.bytehide.com/blog/c-wants-to-become-the-most-popular-programming-language-in-2022. [Último acceso: 1 Julio 2023].

[9] TheHappieCat, «YouTube,» 16 Julio 2016. [En línea]. Available: https://www.youtube.com/watch?v=nEnNtiumgII. [Último acceso: 18 Abril 2023].

[10] S. Rabin, Game AI Pro 2: Collected Wisdom of Game AI Professionals, New York, 2013.

[11] Crystal Dynamics, Monolith Productions, Center of Research, Saint-Cyr Military Academy, «Goal-Oriented Action Planning: Ten Years Old and No Fear!,» de *Goal-Oriented Action Planning: Ten Years of AI Programming*, San Francisco, 2015.

[12] T. Thompson, «The Nemesis System of Shadow of Mordor,» modl, 2022.

[13] Creately, «creately.com,» 26 Junio 2023. [En línea]. Available: https://creately.com/blog/es/diagramas/tutorial-diagrama-caso-de-uso/. [Último acceso: 29 Junio 2023].

[14] Ian, «stackoverflow.com,» 11 Junio 2009. [En línea]. Available: https://stackoverflow.com/questions/980390/consistent-hashcodes-for-dictionaries-in-c-sharp. [Último acceso: 11 Abril 2023].

[15] G. S. polygenelubricants, «stackoverflow.com,» 30 Abril 2010. [En línea]. Available: https://stackoverflow.com/questions/2738886/what-is-a-best-practice-of-writing-hash-function-in-java. [Último acceso: 12 Abril 2023].

[16] G. Maggiore, C. Santos, D. Dini, F. Peters, H. Bouwknegt y P. Spronck, LGOAP: adaptive layered planning for real-time videogames, 2013.

[17] P. H. a. E. J. Chris Conway, «Goal-Oriented Action Planning: Ten Years of AI Programming,» GDC, California, 2017.

[18] Ubisoft, «AI Action Planning on Assassin's Creed Odyssey and Immortals Fenyx Rising,» de *AI Action-Planning on Assassin*, San Francisco, 2021.

Ludografía

F.E.A.R.	Monolith Productions	2005
Condemned: Criminal Origins	Monolith Productions / SEGA	2005
S.T.A.L.K.E.R.: Shadow of Chernobyl	GSC Game World / THQ	2007
Mushroom Men: The Spore Wars	Red Fly Studio	2008
Ghostbusters	Red Fly Studio	2008
Silent Hill: Homecoming	Double Helix Games / Konami	2008
Fallout 3	Bethesda Softworks	2008
Empire: Total War	Creative Assembly / SEGA	2009
F.E.A.R. 2: Project Origin	Monolith Productions / Warner Bros	2009
Demigod	Gas Powered Games / Stardock	2009
Just Cause 2	Avalanche Studios / Eidos Interactive	2010
Transformers: War for Cybertron	High Moon Studios / Activision	2010
Trapped Dead	Headup Games	2011
Deus Ex: Human Revolution	Eidos Interactive	2011
Tomb Raider	Square Enix	2013
Middle-Earth: Shadow of Mordor	Monolith Productions	2017
Assassin's Creed Odyssey	Ubisoft	2018
Immortals Fenyx Rising	Ubisoft	2020

Diagramas de Clase

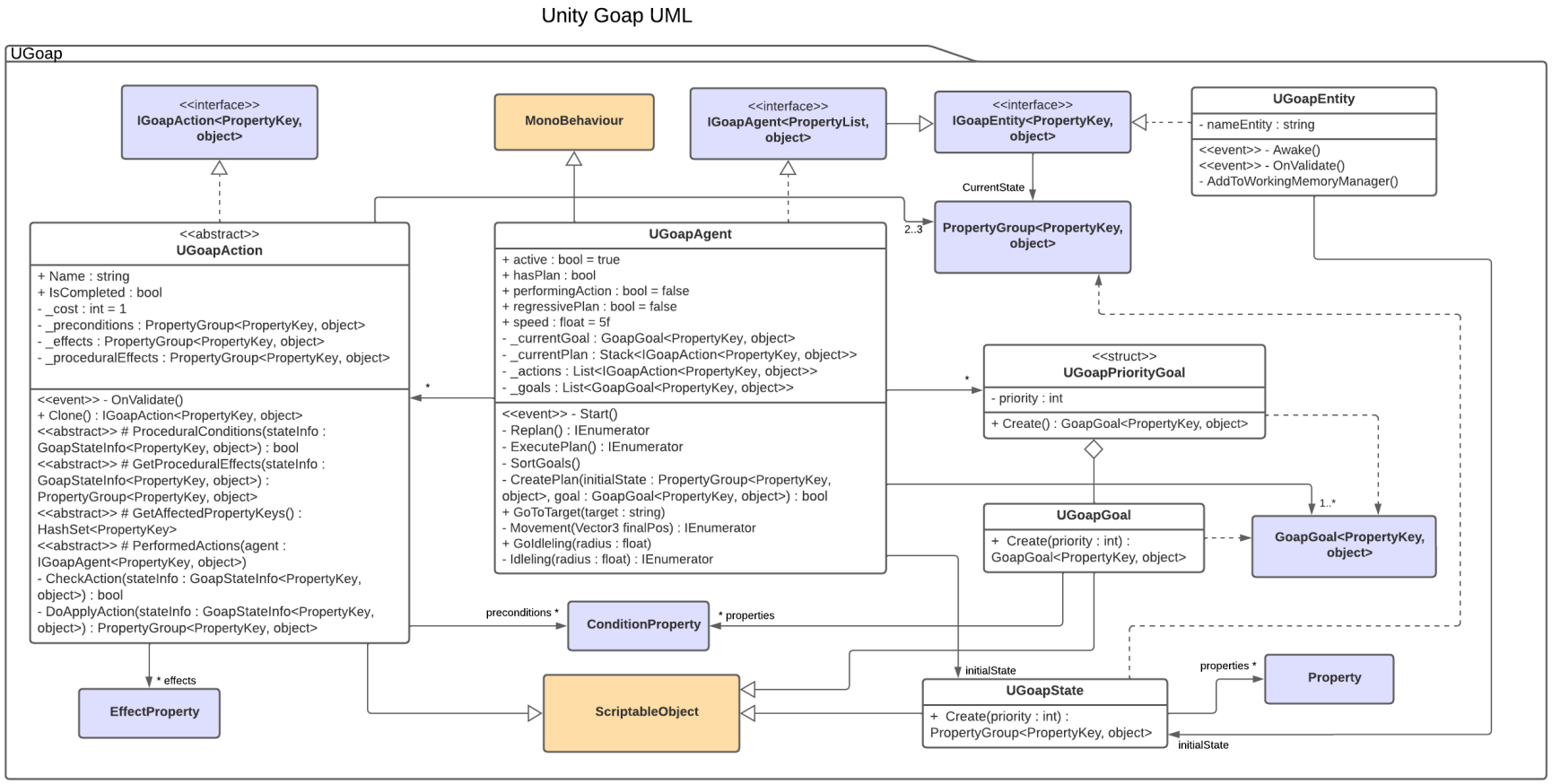


Ilustración 34 - Diagrama de clase de Unity Goap

Base Goap UML

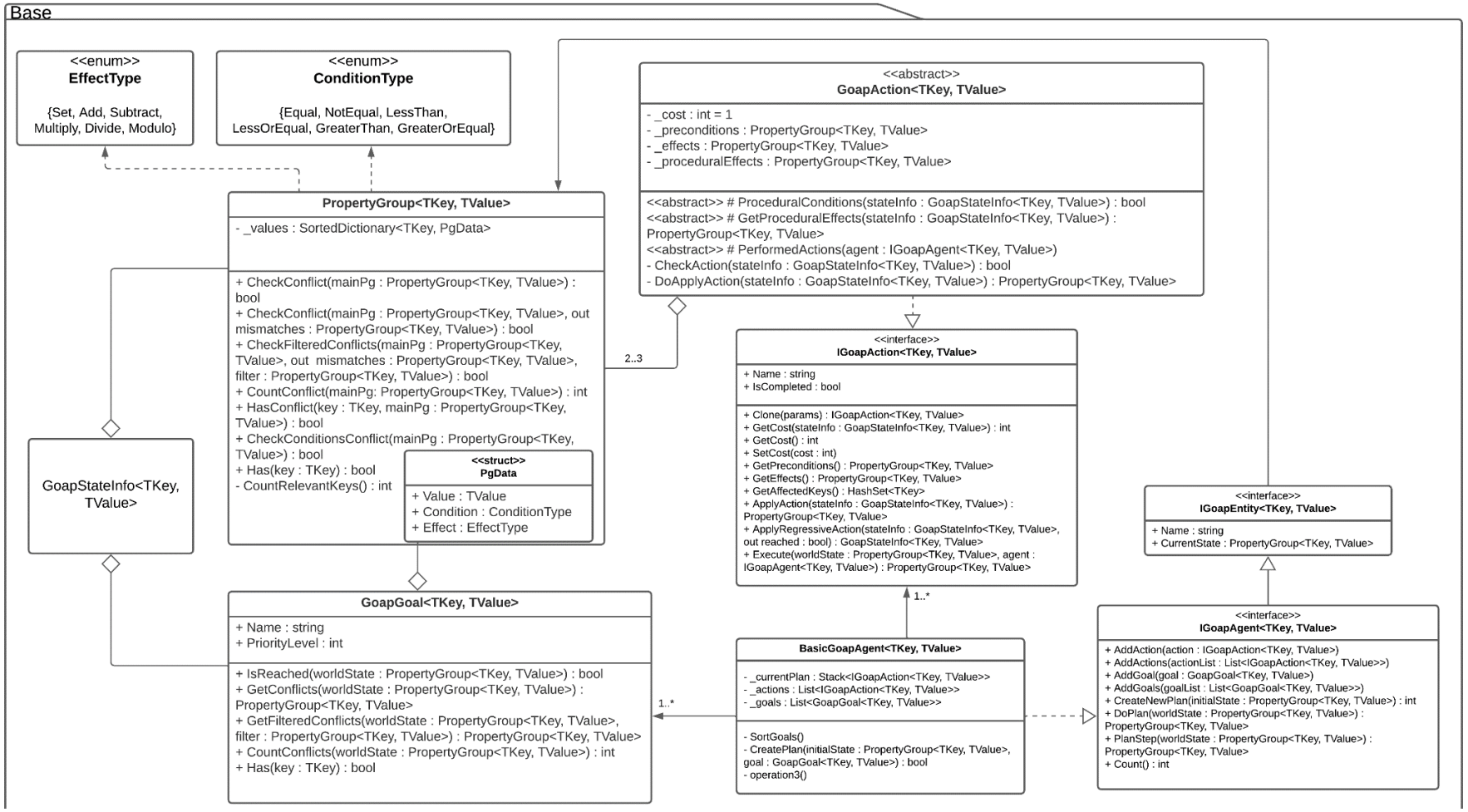


Ilustración 35 - Diagrama de clase de Base Goap

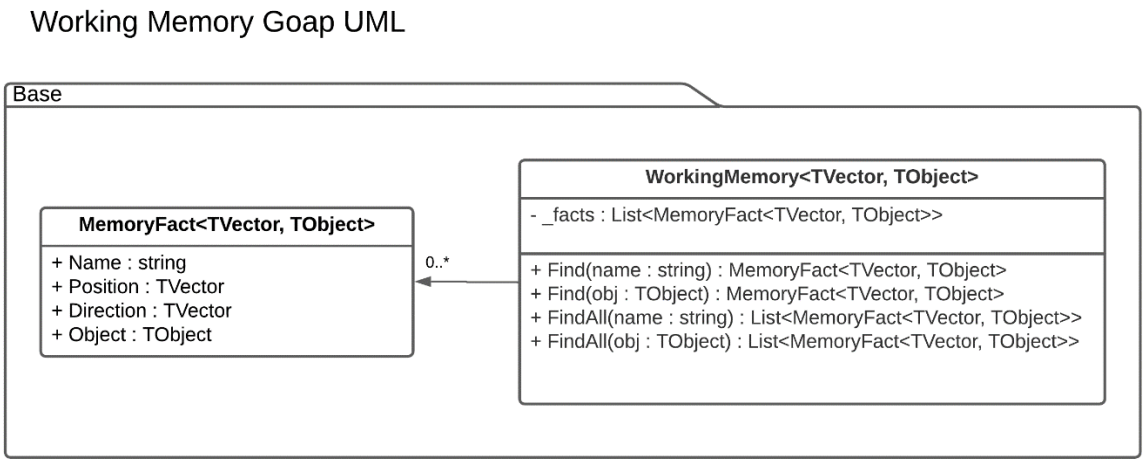


Ilustración 36 - Diagrama de clase del Working Memory Goap

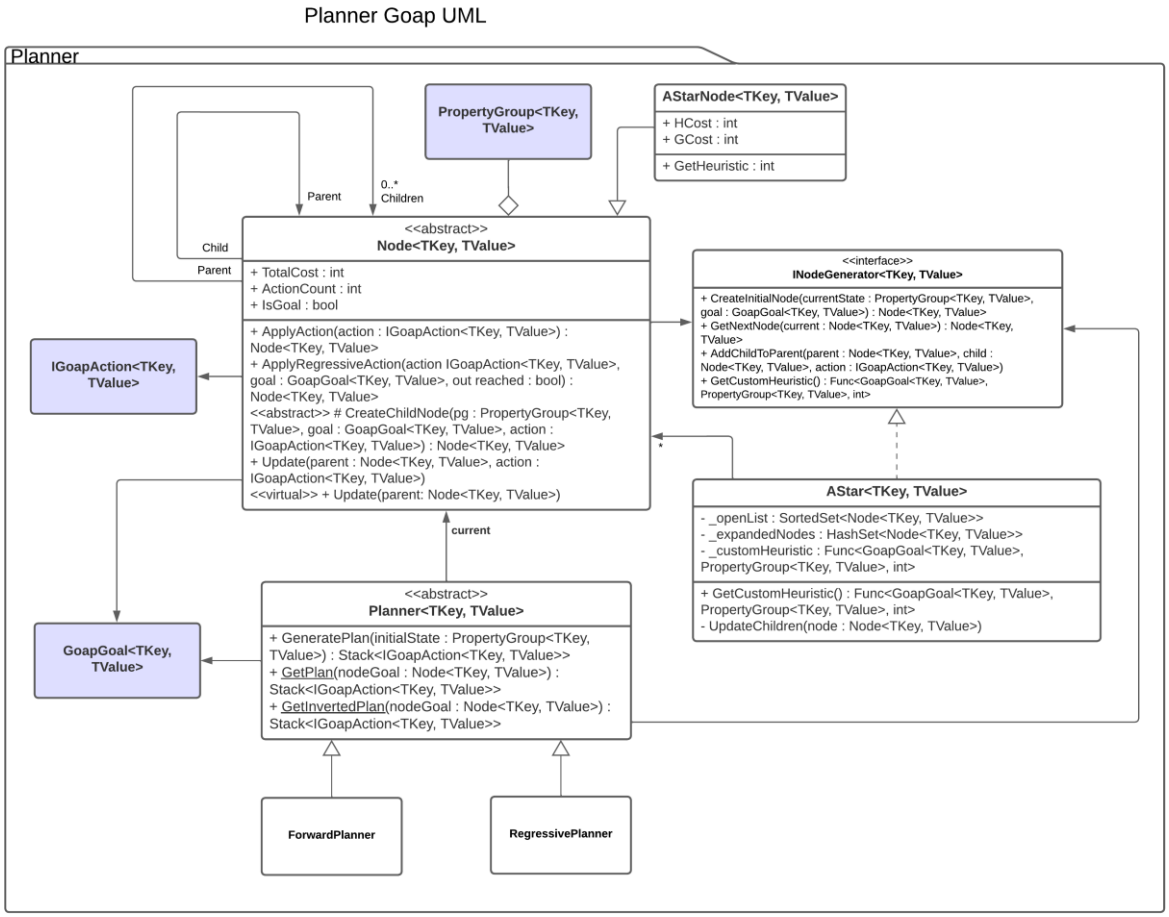


Ilustración 37 - Diagrama de clase del Planner Goap

Static Classes Goap UML

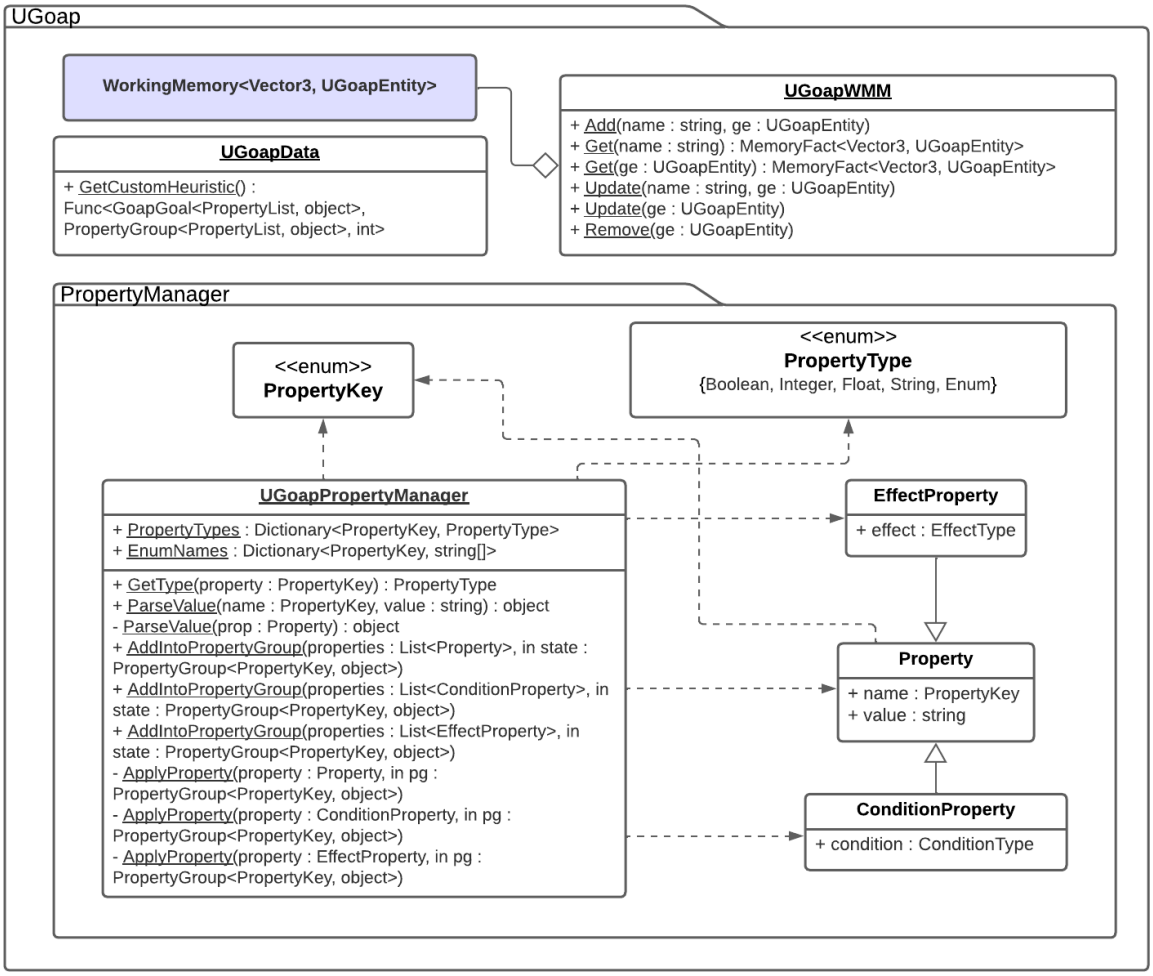


Ilustración 38 - Diagrama de clase de clases estáticas

Anexo II

Acciones Detalladas

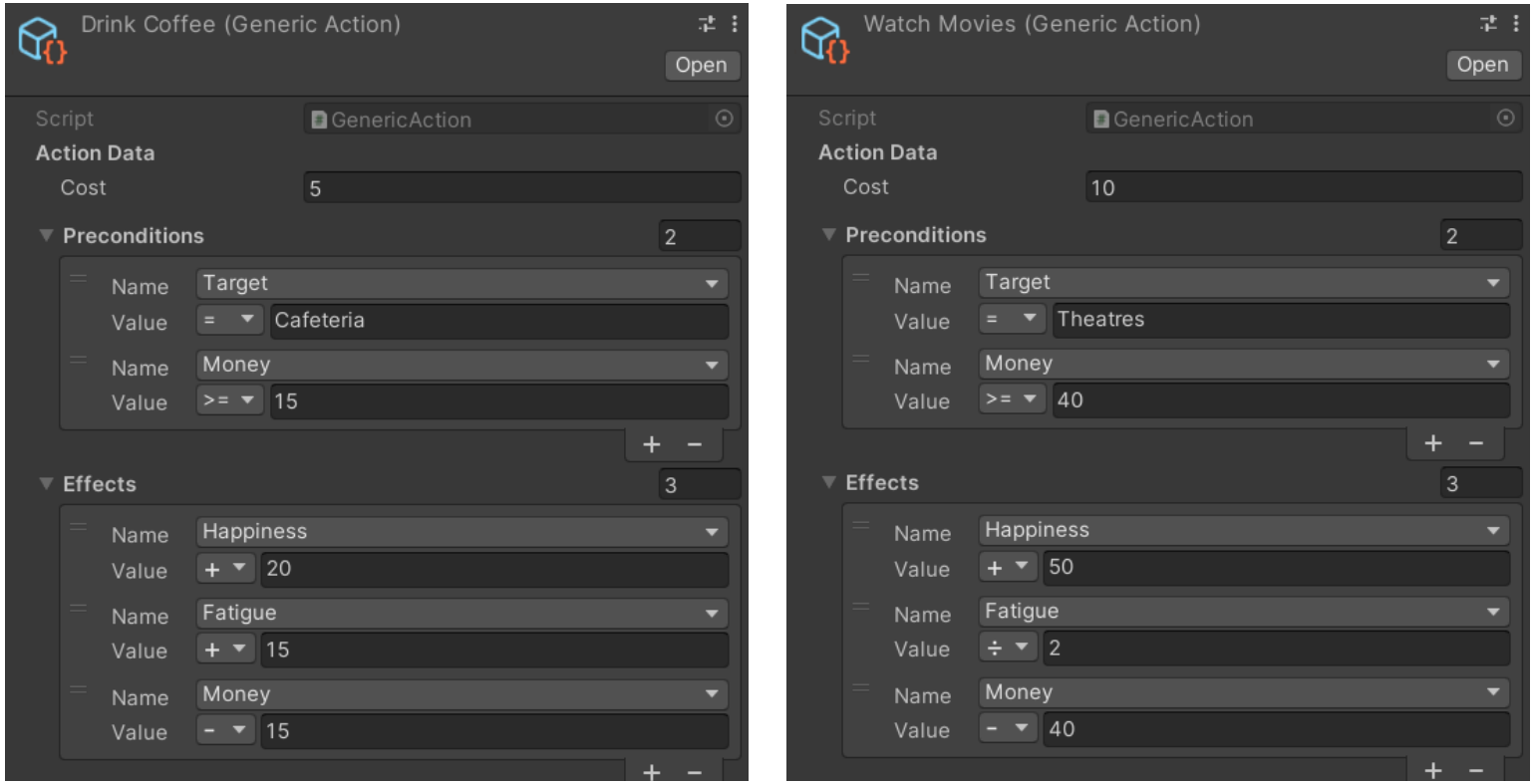
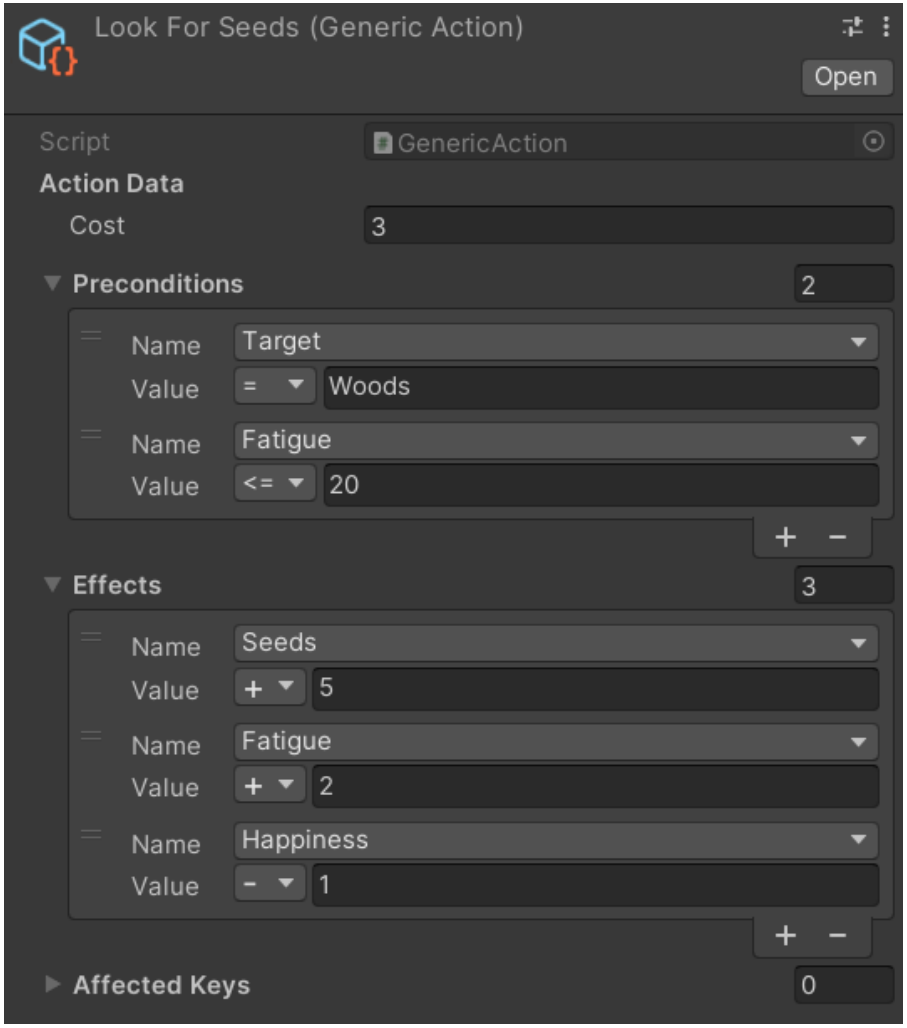


Ilustración 39 - Definición de acciones de ocio



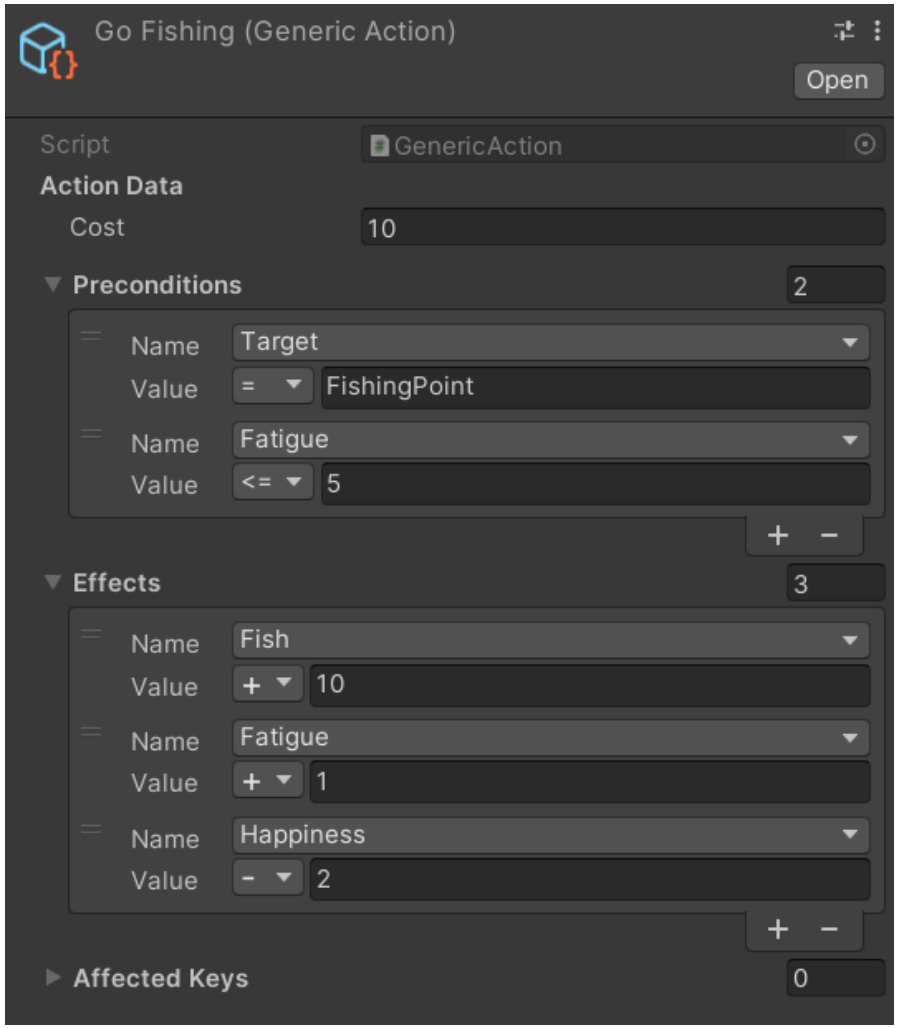


Ilustración 40 - Definición de acciones de trabajo

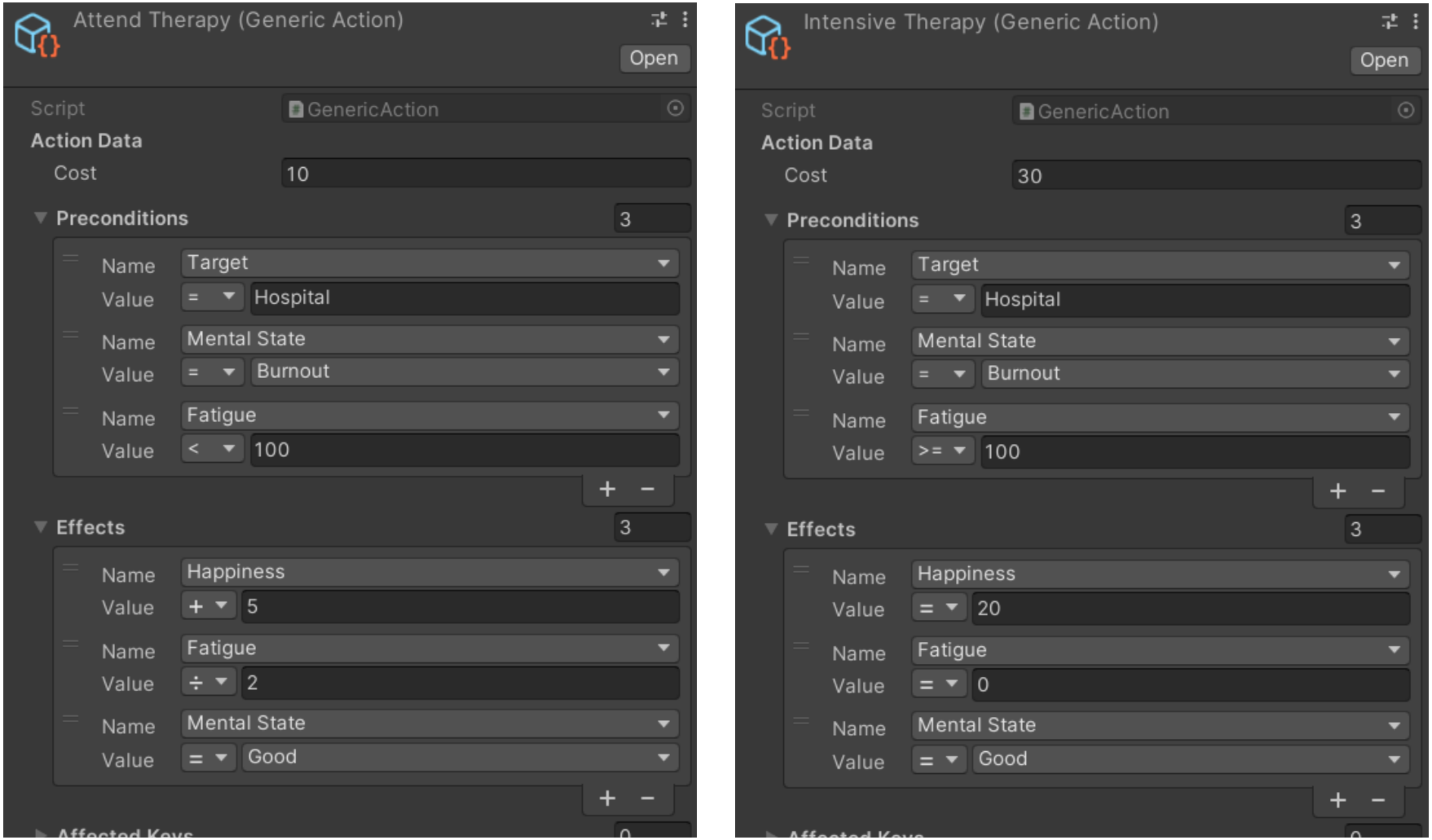


Ilustración 41 - Definición de acciones de salud