Trabajo Fin de Grado
grado en ingeniería informática

Desarrollo de una arquitectura reactiva y deliberativa usando planificación en el entorno de juegos GVGAI

Autor
Vladislav Nikolov Vasilev

Director
Juan Fernández Olivares

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, Julio de 2020

Desarrollo de una arquitectura reactiva y deliberativa usando planificación en el entorno de juegos GVGAI

Vladislav Nikolov Vasilev

Palabras clave: Inteligencia Artificial, Planificación automática, Videojuegos

Resumen

A pesar de que la planificación automática ha sido integrada con éxito en muchas aplicaciones reales, los videojuegos siguen suponiéndole un gran reto debido a lo dinámicos y complejos que son. Esto ha llevado a que muchos autores hayan intentado integrar arquitecturas deliberativas basadas en planificación en videojuegos. Sin embargo, las arquitecturas propuestas se centran solamente en un único juego. Por tanto, en este trabajo presentamos una novedosa arquitectura semiautomática que combina una componente reactiva con una componente deliberativa basada en planificación en el entorno de juegos GVGAI. Esta arquitectura está diseñada de manera que permita resolver múltiples juegos del entorno. Se reciben como entrada un dominio de planificación PDDL y un archivo de configuración YAML que contiene la correspondencia entre elementos del juego y los predicados PDDL definidos en el dominio y un conjunto de objetivos a alcanzar. A partir del fichero de configuración, el sistema genera automáticamente problemas PDDL. Utilizando un planificador en la nube que recibe el dominio PDDL y el problema generado, se obtienen planes para alcanzar los objetivos propuestos. La ejecución del plan es monitorizada por la componente reactiva, comprobando la existencia de discrepancias en dicho plan.

Development of a reactive and planning-based deliberative architecture in the GVGAI game environment

Vladislav Nikolov Vasilev

Keywords: Artificial Intelligence, Automated Planning, Video Games

Abstract

Despite the fact that automated planning has been successfully integrated into many real-world applications, video games are still considered to be very challenging due to their dynamism and complexity. This has led many authors to try to integrate planning-based deliberative architectures into video games. However, the proposed architectures focus only on a single game. Therefore, in this work we present a novel semi-automatic architecture that combines a reactive component with a planning-based deliberative component in the GVGAI game environment. This architecture has been designed so that many of the environment's games can be solved. It receives a PDDL planning domain and a YAML configuration file that contains both the correspondence between the game's elements and PDDL predicates defined in the domain and a set of goals to be reached. The system automatically generates PDDL problem files thanks to the configuration file. By using a planner in the cloud that receives the PDDL domain and the generated problem, the system obtains plans that allow it to reach the proposed goals. The plan's execution is monitored by the reactive component, checking the existence of discrepancies in that plan.

Yo, **Vladislav Nikolov Vasilev**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con NIE X8743846M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Vladislav Nikolov Vasilev

Granada a 6 de julio de 2020.

D. **Juan Fernández Olivares**, Profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Desarrollo de una arquitectura reactiva y deliberativa usando planificación en el entorno de juegos GVGAI***, ha sido realizado bajo su supervisión por **Vladislav Nikolov Vasilev**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 6 de julio de 2020.

El director:

Juan Fernández Olivares

Agradecimientos

Quiero empezar dándole las gracias a Ignacio Vellido Expósito por la creación del dominio simplificado del juego *Boulderdash*. También quiero darle las gracias a mi tutor, el Dr. Juan Fernández Olivares por darme la oportunidad de trabajar en un proyecto de este tipo. Y por último, pero no por ello menos importante, quiero darles las gracias a mis padres y a todos mis amigos, los cuales me han dado todo el apoyo necesario para continuar adelante, incluso en los momentos más duros.

Índice general

Índice de figuras

CAPÍTULO 1

Introducción

La planificación automática es una rama de la Inteligencia Artificial cuyo objetivo último es proporcionar planes que permitan resolver problemas [8]. Para ello se hace uso de un ***planificador***, que es un programa que recibe como entradas un ***dominio de planificación***, es decir, una representación del mundo, y un ***problema de planificación***, que es un estado concreto de ese mundo que debe ser resuelto. El resultado obtenido es una secuencia ordenada de acciones que permiten alcanzar el objetivo descrito en el problema desde el estado inicial que se describe en él.

A pesar de que la planificación automática ha sido integrada con éxito en muchas aplicaciones reales como por ejemplo la robótica o los vehículos no tripulados, el campo de los videojuegos sigue suponiéndole un gran reto. Esto se debe a que los videojuegos presentan, en líneas generales, entornos dinámicos y complejos, los cuales requieren una respuesta rápida ante los cambios que se producen y presentan espacios de búsqueda muy grandes debido al elevado número de acciones que existen en éstos. El proceso de planificación es costoso y lento, por lo que los planificadores no se ven capaces de ofrecer una respuesta rápida a los cambios que se producen en su entorno.

El reto que supone integrar una arquitectura deliberativa basada en planificación en un videojuego ha abierto muchas líneas de investigación. Sirva como ejemplo las arquitecturas que se han desarrollado en torno al juego StarCraft [2,

4, 5, 11] debido a la enorme dinamicidad y complejidad que se puede encontrar en dicho juego. Existen, no obstante, otros juegos que se podrían utilizar para dichas pruebas. `GVGAI` (*General Video Game AI*)[10] es un entorno de videojuegos que va a servir de banco de pruebas en este TFG y que dispone de más de cien juegos diferentes. Entre ellos existen juegos que pueden ser resueltos muy fácilmente mediante técnicas de planificación, y también hay otros juegos con entornos dinámicos que pueden llegar a suponer un auténtico reto para los planificadores.

Si bien es cierto que `GVGAI` tiene una gran cantidad de juegos que pueden resultar interesantes desde el punto de vista de la planificación, hay que tener en cuenta que el entorno no fue concebido para la creación de arquitecturas deliberativas basadas en técnicas de planificación. Principalmente, el entorno surgió para permitir el desarrollo y la competición de arquitecturas reactivas basadas en técnicas como el Aprendizaje por Refuerzo o MCTS (*Monte Carlo Tree Search*). Por tanto, crear agentes basados en planificación para los distintos juegos de `GVGAI` puede llegar a ser un proceso costoso que además se sale de la propuesta original del entorno. Es precisamente en este contexto que surge el presente trabajo.

El objetivo principal de este trabajo es la creación de una arquitectura que combina un componente reactivo con uno deliberativo basado en planificación en el entorno de juegos `GVGAI`. Además, a diferencia de las anteriores arquitecturas, las cuales estaban creadas para un juego en específico, la propuesta que se hace en este trabajo tiene también como objetivo ser lo suficientemente general como para permitir resolver cualquier juego del entorno que se desee mediante planificación, siempre y cuando éste pueda ser resuelto utilizando este tipo de técnicas.

Esta propuesta puede llegar a ser bastante útil por dos motivos principales. Por un lado abre nuevas vías para experimentar con técnicas de planificación en `GVGAI`, lo cual es de sumo interés para el desarrollo de arquitecturas *online* que integran un planificador que dirige el comportamiento deliberativo de un agente. Por otro lado, puede utilizarse como herramienta docente para enseñar cómo se representan los problemas de planificación y para comprender mejor el funcionamiento de la planificación.

El trabajo sigue la siguiente estructura. En el capítulo 2 se hará una introducción, comentando los trabajos relacionados e introduciendo los conceptos fundamentales. Después, en el capítulo 3 se detallará el plan de trabajo seguido. Una vez hecho esto, en el capítulo 4 se presentará la arquitectura general del sistema, y seguidamente, en el capítulo 5 se comentará la arquitectura en más detalle. En el capítulo 6 se hablará de la creación y del formato del archivo de configuración. Una vez hecho esto pasaremos al capítulo 7, donde se hará una descripción general de la implementación *software* que se ha llevado a cabo. En el capítulo 8 se presentarán una serie de experimentos que se han realizado con el sistema. Por último, en el capítulo 9 se presentarán las conclusiones que se pueden extraer y trabajos futuros.

CAPÍTULO 2

Antecedentes

2.1. Trabajos relacionados

Hasta donde sabemos, no se ha desarrollado una arquitectura de planificación para controlar el comportamiento deliberativo de un agente en `GVGAI`. No obstante, se han desarrollado otras arquitecturas de planificación para otros juegos, como es el caso de StarCraft. Algunas de estas arquitecturas son, por ejemplo, `Darmok` [2], basada en casos; `UalbertBot` [4], basada en la búsqueda heurística de planes concurrentes; `EISBot` [5], basada en planificación reactiva; y una arquitectura basada en `PELEA` [3] que combina *goal reasoning* [11] con planificación clásica.

A diferencia de las anteriores, nuestra arquitectura está diseñada para resolver problemas en varios dominios de planificación y juegos en vez de solo en un dominio de planificación y un juego, que en este caso sería StarCraft. Por esa razón, presentamos una metodología semiautomática para la integración de un planificador en cualquier juego de `GVGAI` que pueda ser resuelto mediante planificación.

2.2. Planificación automática

La planificación automática es un área de la IA que se encarga del estudio de técnicas que permiten resolver problemas en los que, dado un objetivo determinado, se debe buscar una serie de acciones que permitan alcanzarlo [8].

Para resolver problemas de planificación se requieren tres elementos:

- Un **dominio de planificación**, el cual representa un entorno y las acciones que puede realizar un actuador en ese entorno, además de cómo dichas acciones afectan al entorno.
- Un **problema de planificación**, en el cual representa un estado inicial concreto de un entorno y un objetivo que debe ser alcanzado por el actuador.
- Un **planificador**, que es un programa que recibe como entradas un dominio y un problema de planificación y devuelve un conjunto de acciones ordenadas que debe realizar el agente para alcanzar el objetivo desde el estado inicial descrito en el problema de planificación.

La planificación ha supuesto un gran impacto en la IA, ya que ha permitido el desarrollo de agentes cuya componente deliberativa se basa en dichas técnicas. Estos agentes, normalmente, siguen una arquitectura basada en **planificación**, **percepción** y **ejecución**:

- La componente de **planificación** permite obtener una serie de acciones que permitan al agente alcanzar un objetivo determinado.
- La componente de **percepción** permite al agente obtener información sobre su entorno, utilizando para ello sensores. Estos sensores pueden ser reales (por ejemplo una cámara) o simulados (por ejemplo, en un videojuego se puede simular lo que ve un agente, simulando así una cámara).
- La componente de **ejecución** permite ejecutar el plan y controlar su ejecución en todo momento, de forma que se puede saber por ejemplo cuando detener la ejecución. Normalmente de esto último se encarga un módulo de **monitorización**, el cual está incluido en la componente de ejecución.

A pesar de que la planificación ha sido integrada con éxito en ciertas áreas, como por ejemplo la robótica, los vehículos no tripulados o los sistemas de producción, hay un área en el cual su integración es más complicada: los videojuegos. El motivo de esto se debe a que la mayoría de videojuegos presentan entornos dinámicos en

cambio continuo y/o con un número extremadamente alto de acciones entre las que escoger. Planificar suele ser un proceso costoso, ya que los espacios de búsqueda suelen ser bastante grandes y por tanto, suelen ofrecer una respuesta relativamente lenta. Al estar estos entornos en cambio continuo, un plan que se haya obtenido deja de ser válido al cabo de poco tiempo, lo cual implica estar replanificando constantemente. En un estudio realizado [7] se ha visto que en entornos de este tipo (muy dinámicos) o muy grandes, un agente reactivo es capaz de ofrecer una respuesta similar a la que permite obtener una arquitectura basada en planificación solo que en un tiempo muchísimo menor.

2.2.1. `PDDL`

`PDDL` (*Planning Domain Definition Language*) [1] es uno de los lenguajes más utilizados para la representación de dominios y problemas de planificación. Surgió a finales de los 90 como un intento de estandarizar los lenguajes que se utilizan para describir los dominios y problemas de planificación. Desde su aparición, el lenguaje ha experimentado una serie de cambios que han añadido funcionalidad nueva al lenguaje. Además, a partir de él han surgido otros lenguajes de planificación.

Un dominio de planificación de `PDDL` puede contener los siguientes elementos:

- Un nombre de dominio. Este elemento es obligatorio
- Unos requisitos del dominio, denotados por `:requirements`.
- Uno o varios tipos, que representan de forma general los elementos que aparecen en el entorno. Se denotan por `:types`, y su uso, aunque no es obligatorio, simplifica mucho la legibilidad del dominio.
- Una serie de predicados, los cuales expresan características de los elementos y del entorno. Se denotan mediante `:predicates`. Su uso es obligatorio.
- Funciones, las cuales son predicados que almacenan un valor numérico. Se denotan mediante `:functions` y su uso no es obligatorio.
- Acciones, las cuales representan la manera en la que el agente interactúa con el entorno y los elementos de éste. Se denotan utilizando `:action`.

  Una acción se compone de unos **parámetros** (representados por `:parameters`), que son una serie de instancias concretas u objetos; unas **precondiciones** (`:precondition`), que son una serie de predicados que se tienen que cumplir en el estado actual del problema cuando se sustituyen las variables de dichos predicados por las instancias concretas recibidas como parámetro; y una lista de **efectos** (`:effect`), la cual indica como cambia el entorno cuando el agente realiza dicha acción.

Por otro lado, un problema de planificación de PDDL se compone de los siguientes elementos:

- Un nombre de problema.
- El dominio PDDL que se utiliza en el problema, denotado por `:domain`.
- Una lista de instancias que componen el estado del problema, denotada por `:objects`.
- Una lista con los predicados instanciados que indican el estado inicial del problema. Esta lista se denota mediante `:init`.
- Una serie de objetivos que se deben alcanzar, denotados por `:goal`.

Para entenderlo mejor, vamos a ver un pequeño ejemplo. Supongamos que queremos representar un dominio en el que hay dos tipos de elementos: monedas y frascos. Una persona puede coger una moneda, la cual se encuentra en el suelo, y puede ponerla en un frasco. Solo se puede coger una moneda a la vez. Además, se quiere llevar la cuenta de cuántas monedas hay en un frasco.

El dominio PDDL que representaría dicha descripción sería el siguiente:

```
(define (domain COINS)
  (:requirements :strips :typing :fluents)
  (:types
    Coin Jar
  )
  (:predicates
    (in ?c - Coin ?j - Jar)
    (on-floor ?c - Coin)
    (taken ?c - Coin)
    (empty-hand)
  )
  (:functions
    (coins-in-jar ?j)
  )
  (:action pick
     :parameters (?c - Coin)
     :precondition (and
       (empty-hand)
       (on-floor ?c)
     )
     :effect (and
```

```
        (not (empty-hand))
        (not (on-floor ?c))
        (taken ?c)
      )
  )
  (:action store
     :parameters (?c - Coin ?j - Jar)
     :precondition (taken ?c)
     :effect (and
       (not (taken ?c))
       (in ?c ?j)
       (empty-hand)
       (increase (coins-in-jar ?j) 1)
     )
  )
)
```

Listing 2.1: Ejemplo de dominio `PDDL`.

Ahora, vamos a suponer que queremos representar un problema en el que hay tres monedas (`c1`, `c2` y `c3`) y un frasco (`j1`). Las tres monedas están en el suelo y no hay ninguna en el frasco. Queremos conseguir que en el frasco haya dos monedas y que una de ellas, `c1`, esté dentro del frasco. El problema `PDDL` que representaría la descripción anterior es el siguiente:

```
(define (problem CoinProblem)
  (:domain COINS)
  (:objects
    c1 c2 c3 - Coin
    j1 - Jar
  )
  (:init
    (empty-hand)
    (on-floor c1)
    (on-floor c2)
    (on-floor c3)
    (= (coins-in-jar j1) 0)
  )
  (:goal
    (AND
      (in c1 j1)
      (= (coins-in-jar j1) 2)
    )
  )
```

```
)
```

Listing 2.2: Ejemplo de problema PDDL.

Como es de suponer PDDL contiene muchos otros elementos. De entre ellos se quieren destacar los efectos condicionales. Estos son efectos de una acción que tienen lugar siempre y cuando las condiciones sean satisfechas en el estado actual del problema. Un ejemplo de esto se puede ver en la siguiente acción, la cual se ha extraído de un dominio escrito para un juego de GVGAI llamado *Ice and Fire*:

```
(:action pick-boots
  :parameters (?a - Avatar ?c - Cell ?b - Boots)
  :precondition (and
    (at ?a ?c)
    (at ?b ?c)
  )
  :effect (and
    (when
      (fire-boots ?b)
      (has-fire-boots ?a)
    )
    (when
      (ice-boots ?b)
      (has-ice-boots ?a)
    )
    (not (at ?b ?c))
  )
)
```

Listing 2.3: Ejemplo de efecto condicional.

El funcionamiento del juego se explicará más detalladamente en capítulos posteriores. De aquí lo único que hace falta entender es que con esta acción un avatar intenta coger unas botas cuando ambos están en la misma casilla y, dependiendo del tipo de las botas, se indica que el avatar posee uno u otro tipo de botas.

2.2.2. Planning.Domains

Planning.Domains [9] es una página *web* que contiene una colección de recursos y herramientas que permiten trabajar con dominios y problemas de planificación de manera sencilla y cómoda. Se puede acceder a dicha página mediante la siguiente URL: http://planning.domains/.

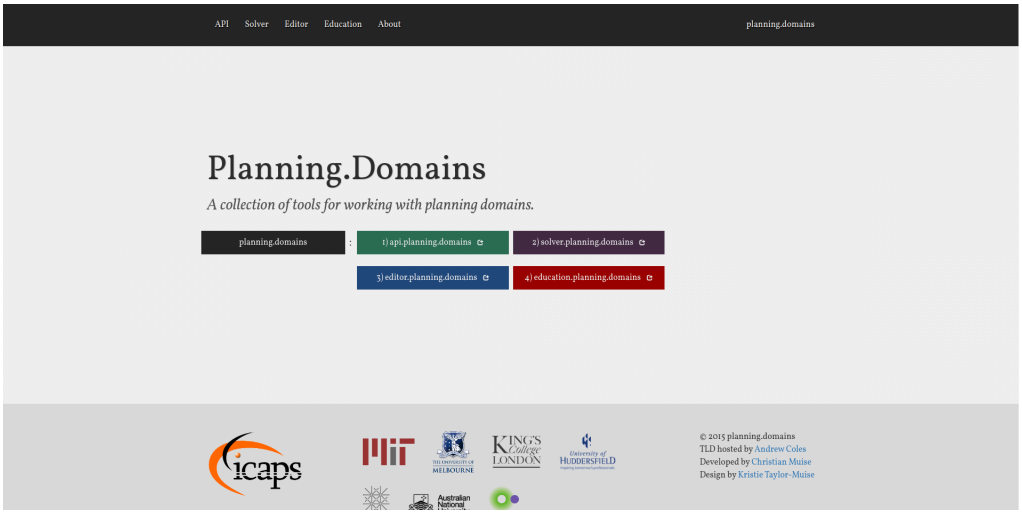


Figura 2.1: Página principal de `Planning.Domains`.

Las herramientas que ofrece dicha página son las siguientes:

- `api.planning.domains`: Permite acceder a una gran cantidad de problemas y dominios de planificación `PDDL` mediante una `API`. Se puede acceder mediante la siguiente URL: `http://api.planning.domains/`.
- `solver.planning.domains`: Ofrece un planificador en la nube que permite obtener planes y validarlos. Se puede acceder a dicho recurso mediante esta URL: `http://solver.planning.domains/`.
- `editor.planning.domains`: Editor en línea que permite definir y resolver dominios y problemas de planificación `PDDL` y resolverlos. Se puede acceder a este recurso mediante la siguiente URL: `http://editor.planning.domains/`.
- `education.planning.domains`: Contiene una serie de recursos para aprender y enseñar sobre la planificación automática. Se puede acceder mediante la siguiente URL: `http://education.planning.domains/`.

De entre todos estos recursos queremos destacar `solver.planning.domains` porque es el planificador que se ha utilizado en el sistema que hemos construido. Se ha escogido éste sobre otros planificadores debido a lo simple que resulta su uso y a que el post-procesamiento que realiza de los planes obtenidos permite obtener mucha más información de la que pueden llegar a ofrecer otros planificadores. Por ejemplo, en la misma respuesta que ofrece se pueden obtener las precondiciones y los efectos instanciados de una acción, lo cual es muy útil para operaciones posteriores que se vayan a hacer con dicha información.

2.3. GVGAI

GVGAI (*General Video Game AI*) [10] es un entorno de juegos que permite el desarrollo de agentes basados en Inteligencia Artificial General, es decir, agentes que sean capaces de resolver múltiples juegos en vez de solo uno en concreto. Cuenta con más de 100 juegos, los cuales están definidos en VGDL y presentan diferentes tipos de retos para un agente.

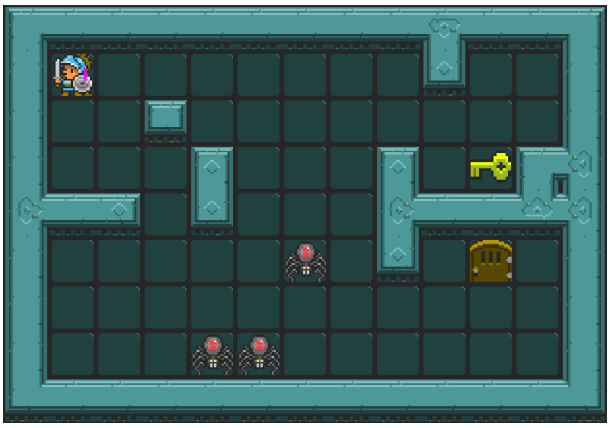


Figura 2.2: Juego *Zelda* en GVGAI

.

El entorno está pensado para el desarrollo de agentes reactivos, los cuales se basan por ejemplo en Aprendizaje por Refuerzo, Algoritmos Genéticos o MCTS (*Monte Carlo Tree Search*). Esto, entre otras cosas, implica que los tiempos de respuesta de los agentes tienen que ser pequeños. Por tanto, los agentes deliberativos se encuentran con un problema aquí, ya que suelen tardar más tiempo en ofrecer una respuesta. Y, como sabemos, los planificadores tardan algún tiempo en dar con una solución. Pero además, existe otro problema, que es la representación de los juegos y de las acciones del agente en dominios de planificación. Este proceso puede llegar a ser extremadamente complejo dependiendo del juego, ya que hay comportamientos que no pueden ser modelados de forma sencilla dentro del juego. Además, supone cierto esfuerzo crear los dominios y validarlos, ya que hay que entender muy bien las reglas del juego y las interacciones que pueden suceder. Esto, sin embargo, no significa que integrar una arquitectura basada en planificación en el entorno sea imposible, pero sí que requiere algo más de trabajo.

2.3.1. VGDL

VGDL (*Video Game Description Language*) [6] es un lenguaje que se utiliza para la descripción de juegos 2D.

La descripción de un juego consiste de dos componentes:

- **Descripción del nivel**: Se describe la disposición de un nivel y la posición de los objetos que están en dicho nivel. Se trata de un archivo de texto que compuesto por líneas de la misma longitud, donde cada carácter se corresponde con un objeto en una posición que viene dada por la fila y la columna en la que se encuentra el carácter.

- **Descripción del juego**: Aquí se describe la dinámica del juego y las interacciones entre los distintos objetos que se puedan encontrar en él. Un fichero de este tipo contiene los siguientes elementos:

  - `LevelMapping`: Especifica una correspondencia entre los caracteres del archivo de descripción del nivel y uno o más objetos que se hayan especificado para cada uno de ellos.

  - `SpriteSet`: Especifica los objetos del juego y sus propiedades. Los objetos se pueden definir de forma jerárquica mediante la indentación. Los objetos hijo heredan las propiedades de su padre y solo pueden tener un único padre.

  - `InteractionSet`: Describe las interacciones que tienen lugar cuando dos objetos colisionan. El primer objeto causa la interacción y el segundo la recibe.

  - `TerminationSet`: Describe las formas en las que puede terminar el juego. Cada línea representa un criterio de terminación.

```
wwwwwwwwwwww
wA........w..w
w..w........w
w...w...w.+ww
www.w2..wwwww
w........w.g.w
w.2..........w
w.....2.....w
wwwwwwwwwwww
```

(a) Fichero descripción del nivel.

```
BasicGame
  SpriteSet
    floor > Immovable randomtiling=0.9 img=oryx/floor3 hidden=True
    goal  > Door color=GREEN img=oryx/doorclosed1
    key   > Immovable color=ORANGE img=oryx/key2
    sword > OrientedFlicker limit=5 singleton=True img=oryx/slash1
    movable >
      avatar  > ShootAvatar   stype=sword frameRate=8
        nokey   > img=oryx/swordman1
        withkey > color=ORANGE img=oryx/swordmankey1
      enemy >
        monsterQuick > RandomNPC cooldown=2 cons=6 img=oryx/bat1
        monsterNormal > RandomNPC cooldown=4 cons=8 img=oryx/spider2
        monsterSlow > RandomNPC cooldown=8 cons=12 img=oryx/scorpion1
      wall > Immovable autotiling=true img=oryx/wall3


  LevelMapping
    g > floor goal
    + > floor key
    A > floor nokey
    1 > floor monsterQuick
    2 > floor monsterNormal
    3 > floor monsterSlow
    w > wall
    . > floor


  InteractionSet
    movable wall  > stepBack
    nokey goal    > stepBack
    goal withkey  > killSprite scoreChange=1
    enemy sword > killSprite scoreChange=2
    enemy enemy > stepBack
    avatar enemy > killSprite scoreChange=-1
    nokey key     > transformTo stype=withkey scoreChange=1 killSecond=True

  TerminationSet
    SpriteCounter stype=goal   win=True
    SpriteCounter stype=avatar win=False
```

(b) Fichero descripción del juego.

Figura 2.3: Descripción en VGDL del juego *Zelda*.

En la figura 2.3 se muestra un ejemplo de un fichero de descripción del nivel y de descripción del juego para el juego *Zelda*, el cual puede verse en la figura 2.2.

Los ficheros VGDL de un juego son bastante útiles a la hora de construir un dominio PDDL para dicho juego, ya que permiten conocer qué objetos hay en ese juego y algunas de las posibles interacciones entre ellos. Además, estos ficheros son utilizados para generar ficheros de configuración para el sistema, aunque eso se verá más adelante.

CAPÍTULO 3

Plan de trabajo

3.1. Metodología de trabajo seguida

Desde el comienzo del proyecto se ha seguido una metodología de desarrollo ágil basada en Scrum. Cada semana o cada dos semanas se celebraba una reunión, la cual se realizaba de manera presencial o telemática mediante Skype o Google Meet. En dichas reuniones se exponía el trabajo desarrollado desde la última reunión, los problemas que habían surgido durante su desarrollo, se proponían algunas posibles soluciones para dichos problemas y se proponían una nueva serie de objetivos que debían cumplirse antes de la siguiente reunión. En dichos nuevos objetivos se consideraban, como era de esperar, los imprevistos que podían haber surgido durante el desarrollo de los objetivos anteriores.

Seguir una metodología ágil como ésta ha facilitado el desarrollo del trabajo de manera sustancial. Gracias a ella se descompuso un trabajo que a priori era bastante grande en pequeños objetivos semanales, los cuales eran más abarcables y servían como guía para no perder de vista el objetivo principal del proyecto.

3.2. Temporización

En la figura 3.1 se muestra la temporización seguida a lo largo del casi año que ha durado este proyecto. En dicho diagrama aparecen reflejadas las tareas que se han llevado a cabo durante este tiempo, las cuales se han agrupado según el tipo de trabajo que había que realizar.

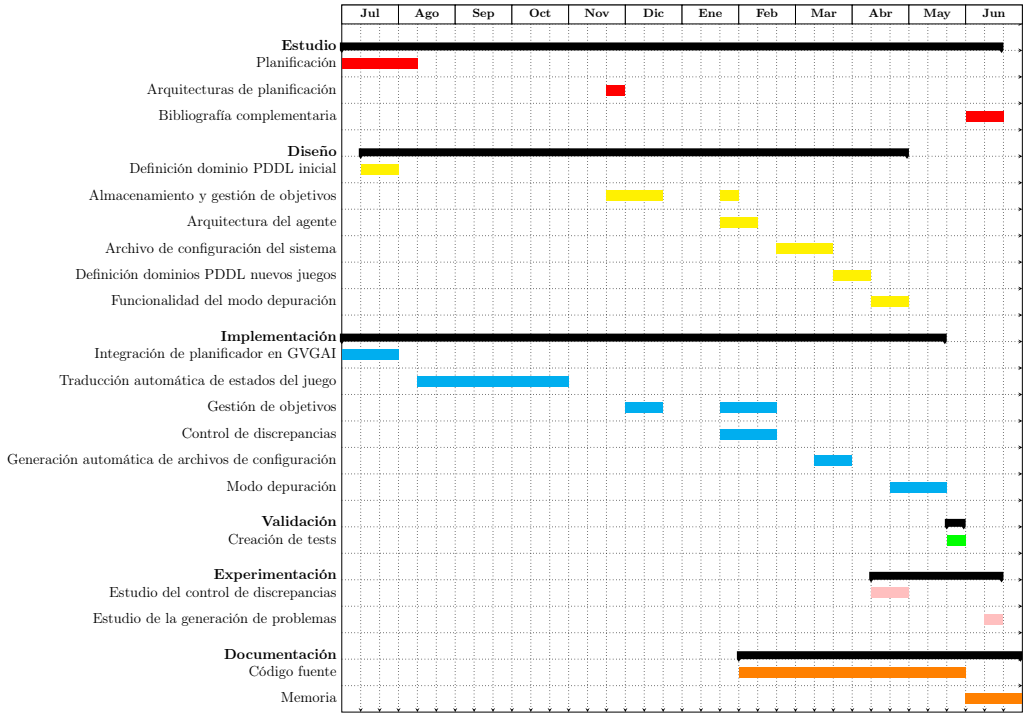


Figura 3.1: Diagrama de Gantt que muestra la temporización del proyecto.

3.2.1. Estudio

El estudio fue una tarea que se llevó a cabo durante casi todo el ciclo de vida del proyecto.

Primero se hizo un estudio de algunos trabajos que trataban el tema de la planificación, además de estudiar el funcionamiento de diversos planificadores para elegir el más adecuado para el problema que se intentaba resolver, aunque al final, para crear el sistema, nos decantamos por otro planificador que no se estudió en esta fase pero sí que era conocido.

Después, a medida que el proyecto iba tomando forma, se hizo un estudio de

una arquitectura reactiva y deliberativa basada en planificación que había sido propuesta por el director del proyecto. Dicha arquitectura sirvió como base para desarrollar la arquitectura final que hemos propuesto en este trabajo.

Por último, a la vez que se comenzó con la redacción de este documento, se llevó a cabo un estudio de otros trabajos relacionados con éste, de forma que se pudiera ver dónde se situaba la propuesta de sistema que habíamos hecho con respecto a las propuestas que han hecho otros autores en el ámbito de la planificación en videojuegos. También se realizó la consulta de diversas fuentes, las cuales permitieron obtener información para la redacción de la parte de antecedentes presentada en el capítulo anterior.

3.2.2. Diseño

El diseño comenzó con la creación del primer dominio de planificación `PDDL`, el cual se usó en las primeras fases del proyecto. A medida que el proyecto iba tomando forma, y gracias al diseño propuesto por el director, se comenzó con el diseño de una estructura de datos que permitiese la gestión de los objetivos de manera sencilla. Solapándose con este proceso se comenzó también el desarrollo de la arquitectura general del agente, ya que ya se tenía una base lo suficientemente grande como para definirla.

Una vez hecho esto, se comenzó con el diseño del archivo de configuración, el cual juega un papel muy importante en el sistema. Para ello se consideraron diversas propuestas, tanto de formato como de generación, hasta que se dio con el formato y el método más adecuados.

Posteriormente, teniendo ya una forma de generar la información necesaria para ejecutar otros juegos de manera sencilla, se comenzó el diseño de dominios de planificación `PDDL` para una serie de nuevos juegos.

Finalmente, a la vista de que el sistema estaba en sus últimas fases, se comenzó con el diseño de un modo depuración, el cuál permitiría obtener más información mientras se ejecutaba un juego.

3.2.3. Implementación

La implementación es, sin lugar a dudas, la tarea que más tiempo ha requerido a lo largo de todo el desarrollo.

En una primera fase, el objetivo era integrar un planificador dentro de `GVGAI` para un juego en concreto, determinando manualmente la traducción de los ele-

mentos del juego a predicados PDDL dentro del código. De esta forma, se podían generar problemas de planificación PDDL a partir de las observaciones del juego.

En una segunda fase se propuso conseguir una traducción automática de los estados del juego a predicados PDDL, la cual permitiese al usuario definir cómo se traducía cada elemento del juego, desacoplando por tanto dicha correspondencia entre elementos del juego y predicados PDDL del código. Este objetivo suponía un primer paso hacia la creación de una arquitectura que permitiese resolver múltiples juegos. En esta primera propuesta se utilizaron una serie de archivos en formato JSON que definían un conjunto de correspondencias que permitían la traducción automática de estados de observación del juego.

Una vez conseguido esto, se comenzó a trabajar en la gestión automática de objetivos, aunque todavía no se habían conseguido desacoplar del código. Paralelamente a esto se comenzó a implementar el control de discrepancias en el sistema. De esta forma, se consiguió crear una estructura de datos que pudiese gestionar los objetivos en función de lo que sucedía dentro del juego.

Después de estudiar algunos diseños para el archivo de configuración, se comenzó con la implementación de un generador que permitiese obtener un archivo de configuración plantilla a partir de un dominio de planificación PDDL especificado por el usuario y una descripción en VGDL del juego. Es aquí donde finalmente se desacoplaron los objetivos del código, permitiendo al usuario establecer los objetivos que quiera.

En las últimas fases se implementó un modo depuración, el cual permitiría obtener maś información del sistema mientras se estuviese ejecutando el juego.

3.2.4. Validación

Una vez que el sistema fue completamente implementado, se realizó una validación de éste creando una serie de tests unitarios que debían pasar distintos componentes del sistema. Estos tests permitieron comprobar el correcto funcionamiento del código implementado, además de que gracia a ellos se pudieron detectar algunos fallos menores que habían pasado desapercibidos.

3.2.5. Experimentación

Teniendo el sistema casi totalmente implementado, se hizo un primer estudio para ver qué tal respondía el agente a las discrepancias que se podían dar a la hora de ejecutar un plan. Posteriormente, se llevó a cabo un segundo estudio que

pretendía poner de manifiesto la rapidez del sistema para generar problemas de planificación `PDDL` a partir de estados de observación del juego.

3.2.6. Documentación

Por último, resta hablar de la documentación. Este es un proceso muy importante en cualquier proyecto, ya que permite conocer qué se ha llevado a cabo y cómo se ha desarrollado el proyecto.

La tarea de documentación comenzó con la documentación del código fuente. Se decidió comenzar con dicha tarea cuando una gran parte del sistema ya estaba implementado, de forma que no fuese necesario reescribirla demasiado.

Esta tarea concluyó con la redacción de este documento, el cual pone de manifiesto todo el trabajo que ha sido desarrollado desde el comienzo del proyecto hasta el final del mismo.

CAPÍTULO 4

Arquitectura general del sistema

El objetivo del sistema es, dados un archivo que contiene un dominio de planificación `PDDL` que describe un determinado juego y un archivo de configuración que especifica una serie de parámetros del sistema, resolver un nivel de dicho juego generando los problemas de planificación `PDDL` hasta los objetivos especificados de forma automática a partir de los estados de observación del juego.

Este sistema está estructurado en una serie de módulos lógicos que representan una tarea de alto nivel. Por lo pronto, los módulos lógicos principales que se han considerado a la hora de definir la arquitectura son los siguientes:

- Un módulo de interacción con el usuario (*User interaction*).
- Un módulo de planificación (*Planning*).
- Un módulo de monitorización y ejecución del plan (*Plan Execution & Monitoring*).

Un módulo lógico se compone a su vez de diversos módulos funcionales o procesos, los cuales llevan a cabo una tarea concreta y se comunican con otros módulos funcionales del mismo módulo lógico para llevar a cabo dicha tarea de alto nivel.

La tarea que un módulo funcional lleva a cabo puede realizarse de forma automática o puede llegar a requerir de la interacción con el usuario para poder ser

completada. Cabe destacar además que, en determinados casos, los módulos funcionales de un módulo lógico pueden comunicarse con los módulos funcionales de otros módulos lógicos, lo cual se describirá más detenidamente más adelante.

En la figura 4.1 pueden verse los distintos módulos lógicos y funcionales que componen el sistema, además de cómo están organizados, cómo se comunican y qué información se envían entre ellos.

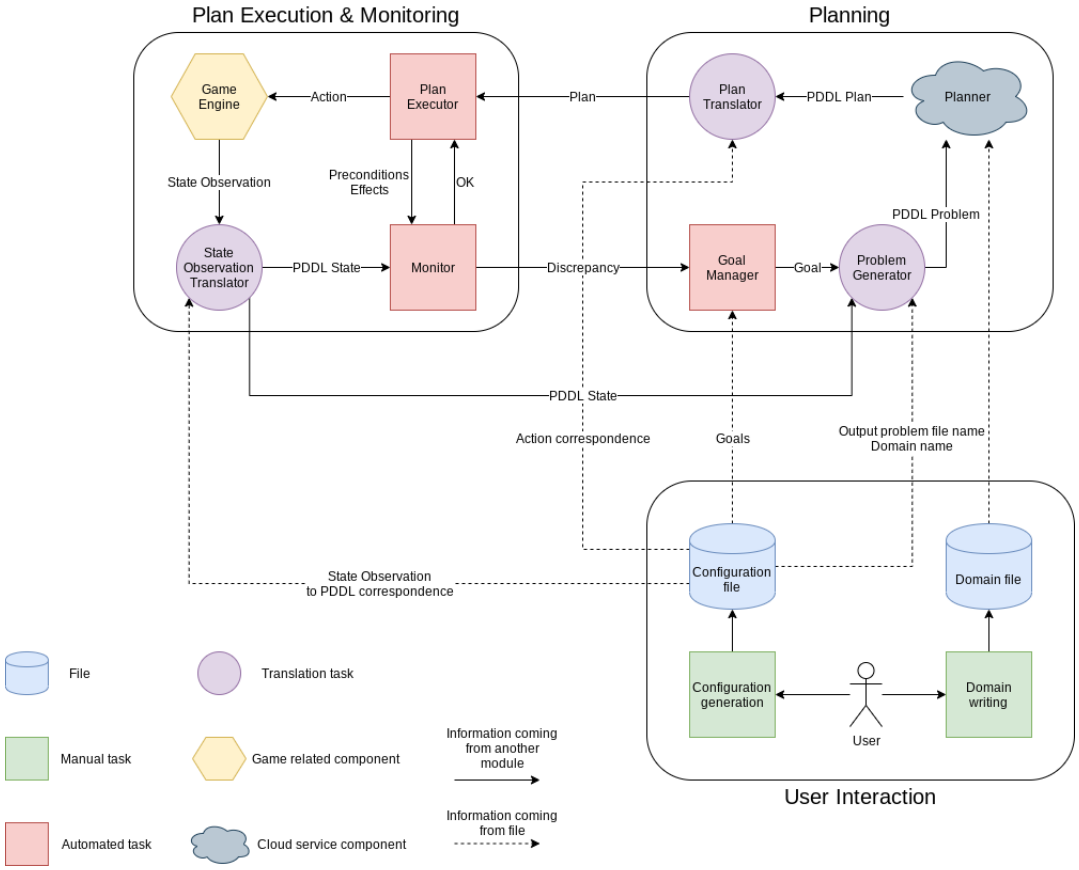


Figura 4.1: Esquema de la arquitectura general del sistema.

Tomando en consideración dicho esquema, vamos a comentar brevemente cual es la funcionalidad básica de cada módulo lógico y cómo interactúan entre ellos. En el próximo capítulo desglosaremos cada uno de ellos y estudiaremos los módulos funcionales que los componen, de forma que se obtendrá una mejor visión de cómo funciona cada componente y de cómo interactúa con las demás.

Empecemos por la parte fundamental, la cual es el **módulo de interacción con el usuario** (*User Interaction*). Como se puede suponer y por lo que se ha

comentado anteriormente, este es el módulo menos automatizado, ya que es con el que interactúa directamente el usuario. Es aquí donde se encuentran dos de los procesos más importantes: la creación del dominio (*Domain writing*) y la creación del archivo de configuración (*Configuration generation*) que utilizará el sistema.

Por una parte, el usuario tiene que definir el dominio de planificación `PDDL`. Este dominio está compuesto por una serie de predicados y tipos de objetos que intentan modelar el juego (por ejemplo conexiones entre casillas, objetos del juego, información sobre la posición de los elementos, etc.). También forman parte del dominio una serie de acciones que, con sus precondiciones y efectos, pretenden reflejar las acciones del juego. No obstante, también se pueden tener otras acciones que no tengan ningún tipo de correspondencia, las cuales podrían considerarse acciones auxiliares. Como se puede suponer, el sistema tendrá que traducir el estado del juego para adaptarse a la representación en formato `PDDL` que ha creado el usuario.

Por otra parte, el usuario debe especificar una serie de parámetros en el archivo de configuración que se utilizarán en el sistema para realizar una serie de tareas. Entre dichas tareas se incluyen:

- La traducción de estados de observación del juego a estados `PDDL` (*State Observation Translation*) mediante una serie de correspondencias entre los elementos de los estados de observación (obtenidos automáticamente del *GVGAI Engine*) y predicados `PDDL` (*State Observation to `PDDL` correspondence*). Esta funcionalidad se describe en la sección 5.3.2.
- La traducción de acciones `PDDL` a acciones de `GVGAI` (*Plan Translator*) mediante una serie de correspondencias entre las acciones de `PDDL` y las de `GVGAI` (*Action correspondence*). Esta funcionalidad se encuentra descrita en la sección 5.2.4.
- La gestión de los objetivos (*Goal Manager*) teniendo en cuenta los objetivos proporcionados por el usuario (*Goals*). Esta funcionalidad se encuentra descrita en la sección 5.2.1.
- La generación de los problemas `PDDL` (*Problem Generator*) especificando el nombre del fichero de problema de salida (*Output problem file name*) y el nombre del dominio (*Domain name*). Esta funcionalidad se ha descrito en la sección 5.2.2.

Otro módulo lógico que se encuentra en el sistema es el **módulo de planificación** (*Planning*). Este módulo se encarga de gestionar los objetivos (*Goals*) cuando sea necesario, estableciendo el objetivo actual y determinando cuales se han cumplido y cuales no (*Goal Manager*). También es el módulo responsable de generar

un archivo de problema (*Problem Generator*), de obtener un plan válido hasta un predicado objetivo dado (*Planner*) y de traducir posteriormente dicho plan para que pueda ser ejecutado en el entorno de juego (*Plan Translator*).

Por último tenemos el **módulo de ejecución del plan y de monitorización** (*Plan Execution & Monitoring*). Como su propio nombre indica, este módulo se encarga de ejecutar el plan obtenido por el módulo de planificación (*Plan Executor*) y de monitorizar dicha ejecución (*Monitoring*), determinando en el proceso si se ha producido alguna discrepancia, y si por tanto hace falta replanificar. Una ***discrepancia*** implica que se ha producido un cambio en el estado del juego que no se esperaba a la hora de obtener el plan original, y por tanto éste ha dejado de ser válido y hace falta encontrar uno nuevo. En el proceso de comprobar si se ha producido una discrepancia se tiene que realizar la traducción del estado de observación del juego a predicados `PDDL` (*State Observation Translator*), ya que esta información se utiliza por el monitor para estudiar la existencia de discrepancias. Para ejecutar el plan se mandan las acciones al motor del juego (*Game Engine*), el cual se encargará de procesarlas de la forma adecuada.

En cuanto a las interacciones entre los módulos, el módulo de ejecución del plan y de monitorización y el de planificación interactúan directamente entre ellos. El primero comunica al segundo si se ha producido alguna discrepancia, y el segundo debe determinar un nuevo objetivo y encontrar un plan hasta dicho objetivo. Posteriormente, el plan obtenido es comunicado al primer módulo, el cual se encargará de hacer las operaciones pertinentes con él.

El módulo de interacción con el usuario se comporta como un proveedor con los otros dos módulos, ya que les proporciona la información necesaria para que éstos puedan llevar a cabo ciertas tareas. Al funcionar como un proveedor, la comunicación es unilateral, ya que no obtiene ningún tipo de respuesta de ellos.

CAPÍTULO 5

Descripción detallada del sistema

Una vez que se ha visto cuál es la arquitectura general del sistema y cómo se agrupan y relacionan los diferentes módulos lógicos y componentes funcionales, vamos a pasar a estudiar dichos elementos de manera más detallada, haciendo especial hincapié en los componentes funcionales.

5.1. Módulo de interacción con el usuario

En este módulo es donde el usuario define el dominio y crea el archivo de configuración, que son los dos elementos que va a utilizar el sistema cuando entre en funcionamiento. Más que un componente software dentro del sistema, este módulo representa un proceso general que es llevado a cabo por el usuario, el cual le permite modificar directamente el funcionamiento del sistema.

En la figura 5.1 se puede ver qué pasos se llevan dentro de este módulo lógico. Se comienza con la definición del dominio `PDDL` (*Domain writing*), el cual representará las acciones que puede ejecutar el agente y la dinámica del juego en términos de precondiciones y efectos. Después de ese proceso vendría la definición del archivo de configuración (*Configuration generation*). A la hora de crear dicho archivo, el usuario puede darse cuenta de que no ha considerado ciertos elementos en su dominio, lo cual implicará que se deberán hacer ciertas modificaciones sobre

dicho dominio. Por último, con los dos ficheros ya creados, el usuario ejecutaría el sistema (*System execution*). En función de la salida obtenida, el usuario podría modificar o bien el dominio creado para adaptarlo mejor al juego, o bien modificar el archivo de configuración para por ejemplo establecer nuevos objetivos o corregir algún error a la hora de crearlo.

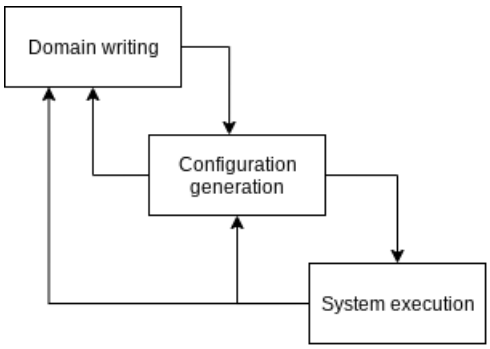


Figura 5.1: Ciclo de vida dentro del módulo de interacción con el usuario.

Visto este esquema, pasemos a estudiar las componentes funcionales que forman el módulo.

5.1.1. Creación del dominio

La creación del dominio de planificación es llevada a cabo completamente por el usuario. Implica definir una representación del juego en el lenguaje `PDDL` (descrito previamente en la sección 2.2.1) que incluya los elementos que se pueden encontrar en éste, las acciones del agente (con sus precondiciones y efectos) y una serie de predicados que definen el funcionamiento del juego. Dicho dominio, en la medida de lo posible, debería ser lo más parecido posible al juego original, de manera que casi todo lo que se haga en el juego se pueda hacer en el dominio.

5.1.2. Creación de la configuración

La creación del archivo de configuración, a diferencia de la creación del dominio, se ha diseñado para que tenga el máximo grado de automatización posible y facilitar así la tarea de integración al usuario. Se dispone de un *script* que permite generar parte de ésta a partir del dominio creado en el paso anterior del proceso (ver figura 5.1) y de la definición en `VGDL` del juego. La otra parte la tendrá que generar el usuario.

La creación de la configuración será descrita con mucho más detalle en el próximo capítulo, donde se comentará el proceso y cómo es un archivo de configuración.

5.2. Módulo de planificación

El funcionamiento del módulo de planificación se explicó en el capítulo anterior. En resumen, el módulo se encarga de gestionar los objetivos proporcionados por el usuario en el archivo de configuración, de elegir un objetivo y de obtener un plan válido hasta dicho objetivo, asegurándose además que el plan pueda ser interpretado por el entorno de juego. La figura 5.2 ilustra el funcionamiento del módulo de forma resumida. En líneas generales, el funcionamiento del módulo está totalmente automatizado, si bien es cierto que requiere que el usuario haya definido el dominio `PDDL` y el archivo de configuración. Se comunica con el módulo de ejecución del plan y monitorización, recibiendo de él una discrepancia en caso de que se haya producido y proporcionándole un plan válido hasta el objetivo actual.

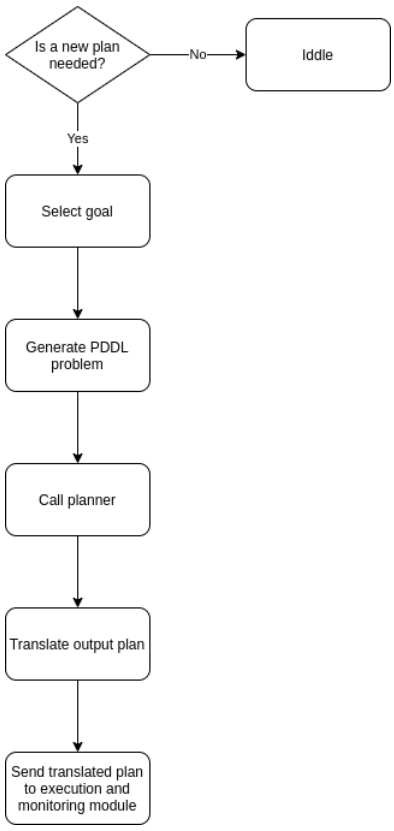


Figura 5.2: Diagrama de flujo del módulo de planificación.

5.2.1. Gestor de objetivos

La principal tarea de este módulo funcional es gestionar los objetivos que le han sido proporcionados en el archivo de configuración y de establecer un objetivo actual. Para hacer dicha gestión, se almacenan los objetivos en una estructura de datos similar a una agenda. Los objetivos están organizados por prioridad, la cual indica si un objetivo debe completarse antes de que se intente buscar un plan para otros objetivos, aunque esto se comentará con más detalle más adelante. Además de eso, un objetivo puede estar en uno de los siguientes estados:

- **No planificado**: no se ha buscado ningún plan hasta ese objetivo.
- **Alcanzado**: el objetivo ha sido alcanzado satisfactoriamente.
- **Detenido**: el plan hasta el objetivo se ha detenido debido a una discrepancia.
- **Actual**: objetivo que ha decidido el gestor que debe alcanzarse.

La transición entre los distintos estados se puede ver en la figura 5.3. Un objetivo comienza en el estado no planificado (*Not planned*), ya que no existe un plan hasta dicho objetivo. Si es seleccionado, el objetivo pasa a ser el objetivo actual (*Current goal*). Estando en dicho estado, y tras haber obtenido un plan hasta él, si se alcanza dicho objetivo éste pasa a estar alcanzado (*Reached*). Si en cambio se produce una discrepancia, el objetivo pasa a estar detenido (*Preempted*). Desde dicho estado, el objetivo puede volver a ser seleccionado como objetivo actual si se dan una serie de condiciones, como por ejemplo que el objetivo detenido tiene que ser completado antes que cualquiera de los objetivos no planificados. Es importante destacar que un objetivo puede pasar, debido a una discrepancia, desde el estado no planificado o detenido al de alcanzado. Esto puede suceder si, mientras se está ejecutando el plan hasta el objetivo actual, se alcanza accidentalmente algún objetivo que esté en este estado, debido por ejemplo a los efectos de una acción.

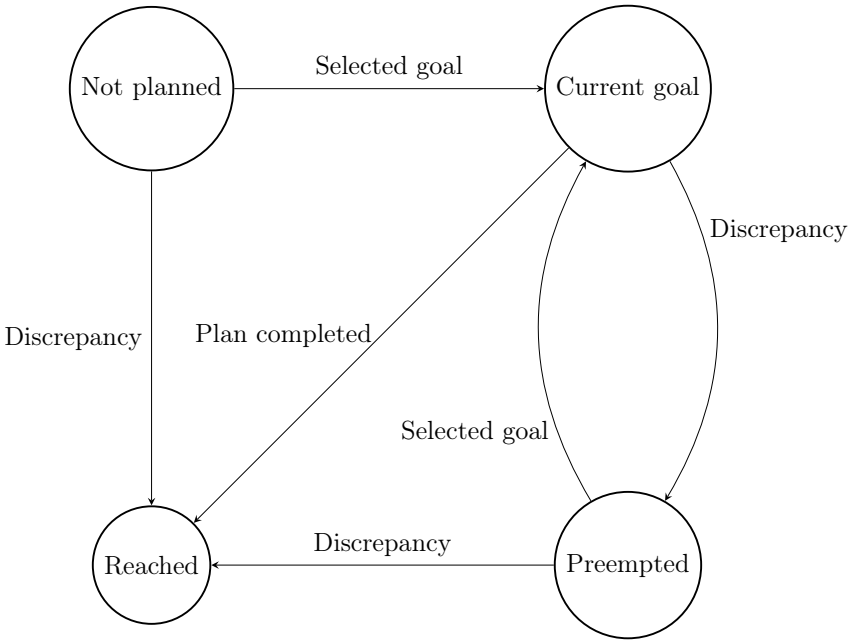


Figura 5.3: Diagrama de transición del estado de un objetivo.

El módulo recibe como entradas los objetivos definidos por el usuario en el archivo de configuración, y cuando sea el caso, una discrepancia que se ha producido a la hora de ejecutar el plan, proporcionada por el monitor. La salida que genera es el objetivo actual, el cual es el nuevo objetivo para el que el planificador debe obtener un plan. Esta salida se envía al generador de problemas, que se va a encargar de generar un problema para ese objetivo.

5.2.2. Generador de problemas

Este módulo funcional se encarga de generar los archivos de problema `PDDL` que necesita el planificador. Recibe como entrada el objetivo actual que ha seleccionado el gestor de objetivos, el estado del juego en formato `PDDL` del traductor de estados de observación y el nombre del fichero de problema de salida y el nombre del dominio del archivo de configuración.

5.2.3. Planificador

El planificador tiene la tarea de obtener un plan desde el estado actual, representado en el problema `PDDL`, hasta el objetivo actual. Podría decirse que este componente es el corazón tanto del módulo lógico como de todo el sistema. Toda

la arquitectura se ha montado alrededor de éste, con lo cual eliminarlo haría que ésta fuese inútil.

Recibe como entradas el archivo con el dominio PDDL y el archivo de problema obtenido por el generador de problemas. La salida que genera es un plan compuesto por acciones PDDL hasta el objetivo, el cual se pasa al traductor de planes.

5.2.4. Traductor de planes

Este componente funcional tiene la tarea de traducir los planes generados por el planificador, los cuáles están formados de acciones PDDL a planes que sean interpretables por el entorno de juego, es decir, planes formados por acciones GVGAI. Para cada una de estas acciones se encarga de extraer también sus precondiciones y efectos instanciados, es decir, con los objetos concretos del estado PDDL del juego. Esta información se puede utilizar luego en el monitor para el estudio de las discrepancias.

El módulo funcional recibe como entrada el plan obtenido por el planificador y la correspondencia entre acciones PDDL y GVGAI, la cual proviene del fichero de configuración. Genera como salida el plan traducido a acciones de GVGAI junto con las precondiciones y efectos instanciados de cada acción del plan. Dicha salida es enviada al ejecutor del plan.

5.3. Módulo de ejecución del plan y monitorización

El funcionamiento general del módulo fue puesto de manifiesto en el capítulo anterior. En resumidas cuentas, este módulo lógico se encarga de ejecutar el plan obtenido por el módulo de planificación y de monitorizar dicha ejecución, de manera que se detecten las discrepancias que se puedan producir. El funcionamiento de este módulo está totalmente automatizado, aunque tal y como pasaba en el módulo anterior, se requiere del archivo de configuración creado por el usuario. Se utiliza la información de dicho fichero para automatizar la traducción de los estados de observación del juego a estados PDDL. Se comunica con el módulo de planificación para comunicarle una discrepancia y para recibir de él el plan a ejecutar.

5.3.1. Motor del juego

Este es el componente que visualiza y ejecuta el juego. En cada turno del juego, se obtiene a partir de él un estado de observación, el cual representa el

estado actual del juego. Dicha salida es utilizada por el traductor del estado de observación. Además de eso, en cada turno recibe una acción del ejecutor del plan, que es la que va a ejecutar el agente.

5.3.2. Traductor del estado de observación

La finalidad de este módulo funcional es generar, a partir de un estado de observación obtenido del motor del juego, una representación en formato `PDDL` del estado de juego actual. Para poder hacer esto de forma automática necesita, aparte de recibir como entrada el estado de observación del motor del juego, una correspondencia entre los elementos y los predicados `PDDL` correspondientes, la cual proviene del archivo de configuración que ha definido el usuario. La salida que produce, el estado de juego en formato `PDDL`, es enviada al monitor y, en caso de que éste detecte una discrepancia, al generador de problemas.

En el algoritmo 1 se puede ver el funcionamiento del traductor de estados de observación. Recibe como entrada el estado de observación del juego, el cual es proporcionado por el motor del juego. Se genera una lista vacía que representa el estado del juego en forma de predicados `PDDL` y un conjunto vacío que contendrá los predicados asociados a las conexiones de las celdas (líneas 2 y 3). Para cada celda del estado de observación se generan los predicados que representan las relaciones de conectividad y se añaden al conjunto (líneas 5-6). A continuación, para cada observación que exista en la celda se obtienen los predicados asociados a dicho elemento del juego (línea 8). Cada uno de estos predicados es instanciado y añadido a la lista de predicados del estado de observación (líneas 10 y 11). Adicionalmente, si la observación que se está procesando es la del avatar y en el juego se utilizan orientaciones, se genera el predicado correspondiente a la orientación del avatar y se añade a la lista (líneas 13-15). Una vez que se ha terminado esto se añaden a la lista de predicados las relaciones de conectividad entre celdas en formato `PDDL` (línea 19) y, en caso de que sea necesario, los predicados asociados a los objetivos alcanzados que deben ser guardados (línea 20).

Algorithm 1 Función que representa el funcionamiento del traductor del estado de observación.

```
1:  function TranslateStateObservation(stateObservation)
2:      pddlState ← Array()
3:      pddlCellConnections ← Set()
4:      for each cell ∈ stateObservation do
5:          connectionsPredicates ← GeneratePDLLConnections(cell)
6:          pddlCellConnections.insert(connectionsPredicates)
7:          for each observation ∈ cell.getObservations() do
8:              correspondenceList ← correspondences[observation]
9:              for each predicate ∈ correspondenceList do
10:                 predicate ← InstantiatePredicate(predicate)
11:                 pddlState.add(predicate)
12:             end for
13:             if observation = avatar and useOrientations then
14:                 orientationPredicate ← GetPDDLOrientation()
15:                 pddlState.add(orientationPredicate)
16:             end if
17:         end for
18:     end for
19:     pddlState.add(pddlCellConnections)
20:     pddlState.add(savedReachedGoalPredicates)
21:     return pddlState
22: end function
```

5.3.3. Monitor

El monitor se encarga de supervisar la ejecución del plan, detectando en el proceso si se ha producido alguna discrepancia. Como se ha comentado anteriormente, una discrepancia se produce cuando el estado actual del juego no es el esperado debido a que ha sucedido algún evento impredecible dentro del juego. Con "estado esperado" nos referimos al que se indica en las precondiciones de la acción, ya que éstas se tienen que satisfacer para que dicha acción pueda ser ejecutada.

De esta forma, el monitor recibe como entrada el estado del juego en formato `PDDL` del traductor de estados de observación y las precondiciones y los efectos de la acción a ejecutar del ejecutor del plan. Si no hay ninguna disconformidad entre las precondiciones y el estado del juego, el monitor le indica al ejecutor que puede proceder. Si en cambio hay algún tipo de disconformidad, el monitor detendría la ejecución del plan e indicaría al gestor de objetivos que se ha producido una discrepancia, de forma que se tendría que seleccionar un nuevo objetivo y buscar un plan hasta él. Adicionalmente, con los efectos de la acción se puede

determinar si un objetivo se ha alcanzado antes de tiempo (también de forma accidental). Esto también generaría una discrepancia, ya que no se esperaba que se alcanzase un objetivo antes de tiempo. Sin embargo, esta discrepancia no obligaría a detener la ejecución del plan actual. Lo único que haría sería indicarle al gestor de objetivos que marque como alcanzado el objetivo alcanzado antes de tiempo. Todo el procedimiento que acaba de describirse puede verse en la figura 5.4.

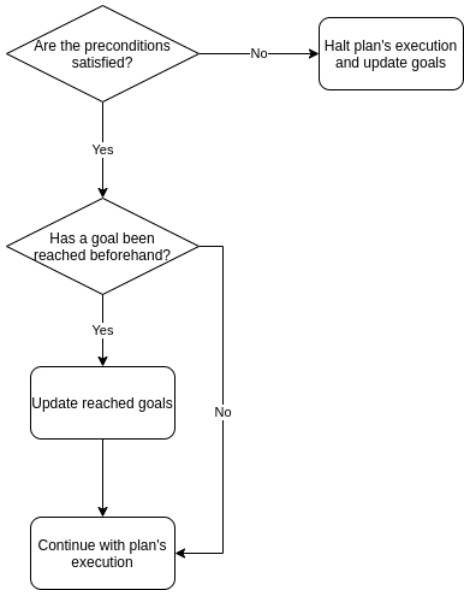


Figura 5.4: Diagrama de flujo del funcionamiento del monitor.

5.3.4. Ejecutor del plan

Este módulo se encarga de ejecutar el plan que le ha sido proporcionado por el módulo de planificación. En cada turno se tiene que ejecutar una acción de dicho plan. No obstante, antes de eso hace falta comprobar que el estado de juego actual es el correcto y no se ha producido ninguna discrepancia, ya que en caso contrario el plan dejaría de ser válido. De esta parte se encarga el monitor, el cual le comunica si todo está en orden o no.

Por tanto, este componente recibe como entrada el plan del traductor de planes y envía al monitor las precondiciones y los efectos de la acción que va a ejecutar a continuación. Si no hay ningún tipo de problema, el ejecutor del plan manda al motor de juego la siguiente acción que se tiene que ejecutar.

CAPÍTULO 6

Descripción del archivo de configuración

Como se ha podido ver hasta ahora, el archivo de configuración definido por el usuario juega un papel clave en el sistema. Es gracias a él que la mayoría de operaciones dentro del sistema pueden ser automatizadas, como por ejemplo la generación de problemas, la traducción de estados de observación a estados `PDDL`, la gestión de objetivos y la traducción de planes. Por tanto, es importante conocerlo más a fondo. En las secciones posteriores se describirá el proceso de creación del archivo de configuración, el formato que tiene y su estructura mediante un ejemplo concreto, el cual facilitará su comprensión.

6.1. Creación del archivo

Como se comentó brevemente en el capítulo anterior, la creación del archivo de configuración está semiautomatizada. Se tiene un *script* que, dado un archivo que contiene el dominio `PDDL` y el archivo de descripción del juego en formato `VGDL`, genera un archivo de configuración plantilla para que el usuario lo pueda rellenar. Algunos de los campos de ese archivo ya estarán rellenados, pero se deja total libertad al usuario para que los modifique. También existen ciertos campos que son opcionales, aunque se concretará más cuáles son cuando se describa la estructura del archivo.

6.2. Formato del archivo de configuración

El archivo de configuración está en formato `YAML` (*YAML Ain't Markup Language*), que es un lenguaje para la serialización de datos creado de forma que sea muy fácil de leer tanto por humanos como por la máquina. Es esta legibilidad lo que hace que `YAML` sea ideal para la creación de archivos de configuración, ya que de manera sencilla y clara se pueden establecer los parámetros que configuran un sistema.

El lenguaje está formado por tipos básicos como cadenas de texto, enteros, números reales y valores booleanos. También tiene tipos más avanzados, como listas y relaciones clave-valor.

6.3. Estructura del archivo de configuración

Para explicar la estructura del archivo vamos a partir de un archivo de configuración plantilla generado para el juego *Boulderdash*. A partir de este archivo plantilla se explicarán los campos que lo forman y cómo se rellenarían. El juego como tal se explicará más detalladamente en capítulos posteriores, pero la idea general es coger 9 de las gemas que hay en el mapa y salir del nivel, esquivando a los enemigos (cangrejos y mariposas) y picando las rocas para despejar el camino. El nivel de ejemplo que se va a utilizar se puede ver en la figura 6.1.

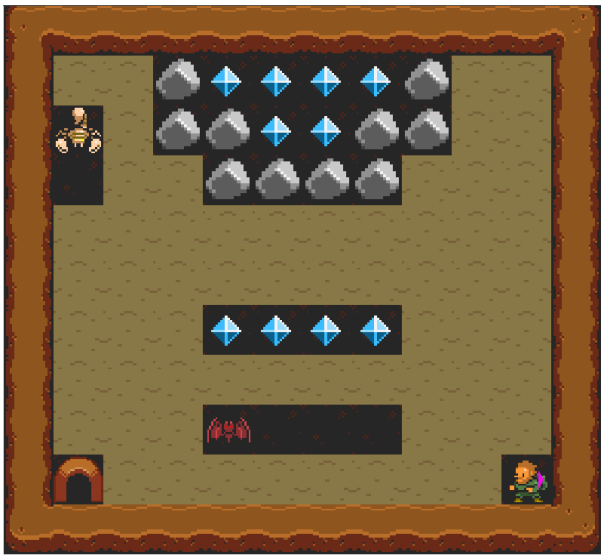


Figura 6.1: Nivel de ejemplo del juego *Boulderdash*.

```
(:types
  Bat Scorpion - Enemy
  Gem Player Boulder Enemy Exit - Locatable
  Cell
)

(:predicates
  (at ?l - Locatable ?c - Cell)
  (oriented-up ?p - Player)
  (oriented-down ?p - Player)
  (oriented-left ?p - Player)
  (oriented-right ?p - Player)
  (connected-up ?c1 ?c2 - Cell)
  (connected-down ?c1 ?c2 - Cell)
  (connected-left ?c1 ?c2 - Cell)
  (connected-right ?c1 ?c2 - Cell)
  (terrain-ground ?c - Cell)
  (terrain-wall ?c - Cell)
  (terrain-empty ?c - Cell)
  (got ?g - Gem)
  (exited-level)
  (occupied ?c - Cell)
)

(:action turn-up
  ... )

(:action turn-down
  ... )

(:action turn-left
  ... )

(:action turn-right
  ... )

(:action move-up
  ... )

(:action move-down
  ... )
```

```
(:action move-left
  ... )

(:action move-right
  ... )

(:action move-up-get-gem
  ... )

(:action move-down-get-gem
  ... )

(:action move-left-get-gem
  ... )

(:action move-right-get-gem
  ... )

(:action dig-up
  ... )

(:action dig-down
  ... )

(:action dig-left
  ... )

(:action dig-right
  ... )

(:action exit-level
  ... )
```

Listing 6.1: Dominio `PDDL` del juego *Boulderdash*.

Es importante también conocer el dominio `PDDL` que se ha creado para el juego, el cual puede verse de forma resumida en el listado 6.1 y se utilizará en la explicación que se va a realizar. Se pueden ver los tipos que representan a los elementos del juego, los predicados y las acciones que puede realizar, aunque éstas se muestran de forma simplificada (solo el nombre). A continuación se ofrece más información sobre lo que representan los predicados:

- El predicado `(at ?l - Locatable ?c - Cell)`, que indica que un objeto

determinado se encuentra en una determinada celda.

- Los predicados `(oriented-up ?p - Player)`, `(oriented-down ?p - Player)`, `(oriented-left ?p - Player)` y `(oriented-right ?p - Player)`, los cuales expresan la orientación del jugador.

- Los predicados `(connected-up ?c1 ?c2 - Cell)`, `(connected-down ?c1 ?c2 - Cell)`, `(connected-left ?c1 ?c2 - Cell)` y `(connected-right ?c1 ?c2 - Cell)`, los cuales sirven para representar las relaciones de conectividad entre las casillas. Por ejemplo, el primero de ellos indica que la celda `?c1` tiene por encima de ella a la casilla `?c2`. Algo similar sucede con el siguiente predicado, donde se indica que `?c2` estaría debajo de `?c1`. Y así con el resto de predicados, los cuales tienen una semántica igual a estos dos casos.

- Los predicados `(terrain-ground ?c - Cell)`, `(terrain-wall ?c - Cell)` y `(terrain-empty ?c - Cell)` indican el tipo de terreno de la celda (con tierra, muro o excavada, respectivamente).

- El predicado `(got ?g - Gem)` indica que se ha cogido la gema `?g`.

- El predicado `(exited-level)` indica que se ha salido el nivel una vez que se han cogido las gemas necesarias.

- El predicado `(occupied ?c - Cell)` indica que la celda está ocupada por una roca o un enemigo.

Por último, es importante comentar brevemente qué representa cada acción:

- Las acciones `turn-up`, `turn-down`, `turn-left` y `turn-right` representan el giro que puede realizar el agente para cambiar su orientación a la que indique el nombre de la acción.

- Las acciones `move-up`, `move-down`, `move-left` y `move-right` implican el movimiento del agente de una casilla a otra que esté conectada con la primera, siempre y cuando el agente esté orientado de la forma correcta y exista una conexión entre las dos casillas. Por ejemplo, si se quiere mover hacia arriba (`move-up`), tendrá que estar orientado hacia arriba y tendrá que existir un predicado `(connected-up c1 c2)` que indique que existe la correspondiente conexión entre ambas celdas.

- Las acciones `move-up-get-gem`, `move-down-get-gem`, `move-left-get-gem` y `move-right-get-gem` son parecidas a las cuatro anteriores, solo que si hay una gema en la casilla destino, el agente la coge en el proceso.

- Las acciones `dig-up`, `dig-down`, `dig-left` y `dig-right` implican cavar en la dirección que indican. Para ello, el agente tiene que estar orientado previamente de forma correcta.
- La acción `exit-level` se lleva a cabo cuando el agente va a salir del nivel, una vez que ha cogido todas las gemas necesarias.

Una vez descritos el juego y el dominio `PDDL` que se ha creado para el juego, pasemos a comentar la estructura del archivo de configuración. A continuación se ofrece el contenido de la plantilla que se ha obtenido a la hora de ejecutar el *script*:

```
domainFile: domains/boulderdash-domain.pddl
problemFile: problem.pddl
domainName: BoulderDash
cellVariable: null
avatarVariable: null
gameElementsCorrespondence:
  background:
  - null
  wall:
  - null
  sword:
  - null
  dirt:
  - null
  exitdoor:
  - null
  diamond:
  - null
  boulder:
  - null
  avatar:
  - null
  crab:
  - null
  butterfly:
  - null
variablesTypes:
  ?variable: Type
orientationCorrespondence:
  UP: null
  DOWN: null
  LEFT: null
  RIGHT: null
```

```
connections:
  UP: null
  DOWN: null
  LEFT: null
  RIGHT: null
actionsCorrespondence:
  TURN-UP: ACTION_UP
  TURN-DOWN: ACTION_DOWN
  TURN-LEFT: ACTION_LEFT
  TURN-RIGHT: ACTION_RIGHT
  MOVE-UP: ACTION_UP
  MOVE-DOWN: ACTION_DOWN
  MOVE-LEFT: ACTION_LEFT
  MOVE-RIGHT: ACTION_RIGHT
  MOVE-UP-GET-GEM: ACTION_UP
  MOVE-DOWN-GET-GEM: ACTION_DOWN
  MOVE-LEFT-GET-GEM: ACTION_LEFT
  MOVE-RIGHT-GET-GEM: ACTION_RIGHT
  DIG-UP: ACTION_UP
  DIG-DOWN: ACTION_DOWN
  DIG-LEFT: ACTION_LEFT
  DIG-RIGHT: ACTION_RIGHT
  EXIT-LEVEL: null
goals:
- goalPredicate: null
  priority: 0
  saveGoal: false
  removeReachedGoalsList:
  - null
```

Los campos que aparecen en el fichero son los siguientes:

1. `domainFile`: Nombre del fichero de dominio `PDDL`. Este campo se genera automáticamente a partir de los parámetros utilizados a la hora de ejecutar el *script*.

2. `problemFile`: Nombre del fichero de problema de salida. Tiene como valor por defecto `problem.pddl`, pero el usuario puede darle el nombre que quiera.

3. `domainName`: Nombre del dominio. Se genera automáticamente y se obtiene a partir del archivo de dominio `PDDL`.

4. `cellVariable`: Variable que se utilizará para representar las celdas. Una variable viene precedida por el símbolo "`?`" y se le da el nombre que quiera el

usuario. El uso del símbolo de interrogación es para que tenga cierta semejanza con la manera en la que se hacen las definiciones de variables en `PDDL`. Por ejemplo, en este caso se ha decidido que dicha variable sea la siguiente:

```
cellVariable: ?c
```

Supongamos que hemos asociado el predicado `(at ?l - Locatable ?c - Cell)` del domino `PDDL` a una serie de elementos del juego. En este caso, la variable `?c` que se ha definido en el dominio se correspondería con la variable que hemos definido en el fichero de configuración, la cual es `?c`. Es importante destacar que no tienen por qué tener el mismo nombre. Por ejemplo, podríamos haberle puesto el nombre `?celda` en el archivo de configuración. Cuando el traductor se encuentre con alguno de los elementos a los que está asociado el predicado, sustituirá en este predicado la variable `?c` por su instancia correspondiente, la cual será `c_x_y`, donde `x` es la columna e `y` la fila donde se encuentra dicho elemento del juego.

5. `avatarVariable`: Variable que se utilizará para representar al avatar/agente. Por ejemplo, en este caso se ha decidido que sea la siguiente:

   ```
   avatarVariable: ?p
   ```

   Esta variable es especial, ya que cuando se encuentre dicha variable en alguno de los predicados asociados al avatar se va a sustituir simplemente por el nombre que le ha dado el usuario. Por ejemplo, en este caso se sustituiría por `p`. Se ha hecho de esta forma debido a que simplifica mucho la monitorización del plan, ya que tener una variable que constantemente cambia sus coordenadas x, y solo induciría a errores.

6. `gameElementsCorrespondence`: Este campo sirve para asociar una lista de predicados a cada observación (elemento) del juego, de forma que cuando el traductor de estados de observación se encuentre con dicha observación, generará los correspondientes predicados instanciando las variables de la misma forma que se describió en el punto 4, a excepción como se comentó en el punto anterior de las variables asociadas al jugador, las cuales se instancian de forma especial. Las observaciones se obtienen a partir del fichero que define el juego en formato `VGDL`, el cual recordemos que se pasa como parámetro a la hora de generar la plantilla del archivo de configuración. Una observación que aparece siempre es `background`, la cual representa una celda en la que no hay nada. Para este juego en concreto se han definido las siguientes correspondencias:

   ```
   gameElementsCorrespondence:
     background:
     - (terrain-empty ?c)
     wall:
   ```

```
  - (terrain-wall ?c)
  sword:
  - (terrain-empty ?c)
  dirt:
  - (terrain-ground ?c)
  exitdoor:
  - (at ?e ?c)
  - (terrain-empty ?c)
  diamond:
  - (at ?g ?c)
  - (terrain-empty ?c)
  - (occupied ?c)
  boulder:
  - (at ?boulder ?c)
  - (terrain-empty ?c)
  - (occupied ?c)
  avatar:
  - (at ?p ?c)
  - (terrain-empty ?c)
  crab:
  - (at ?s ?c)
  - (terrain-empty ?c)
  - (occupied ?c)
  butterfly:
  - (at ?b ?c)
  - (terrain-empty ?c)
  - (occupied ?c)
```

Los predicados que aparecen, como es de suponer, deben ser los que se encuentran en el dominio `PDDL` y deben ser correctos. Otra cosa importante es que se debe mantener un mínimo de consistencia. Por ejemplo, si se ha definido que las celdas se representan mediante la variable `?c`, en todos aquellos predicados donde aparezcan celdas debería aparecer dicha variable. En resumen, la nomenclatura que se utiliza para definir las variables debe ser consistente, ya que si no se hace de esta forma, se pueden llegar a producir errores a la hora de traducir las observaciones del juego.

7. `variablesTypes`: Este campo representa una correspondencia entre las variables que aparecen en los predicados del campo anterior y los tipos de `PDDL` que el usuario haya definido en el dominio. Para este juego en concreto se han definido las siguientes asociaciones:

```
variablesTypes:
  ?e: Exit
  ?p: Player
```

```
?boulder: Boulder
?g: Gem
?b: Bat
?s: Scorpion
?c: Cell
```

Aunque anteriormente se ha definido cuáles son las variables asociadas a las celdas y al agente, sigue siendo necesario especificar su tipo, ya que el usuario podría darle cualquier nombre al tipo asociado a dichas variables al no haber un estándar.

8. `orientationCorrespondence`: Este campo es opcional y solo se utiliza en aquellos juegos donde la orientación del personaje sea un factor a tener en cuenta. A cada clave se le tiene que asociar el correspondiente predicado que indique que el agente está orientado en esa dirección (`UP` para arriba, `DOWN` para abajo, `LEFT` para izquierda y `RIGHT` para derecha). En este caso, los siguientes predicados indican la orientación:

```
orientationCorrespondence:
  UP: (oriented-up ?p)
  DOWN: (oriented-down ?p)
  LEFT: (oriented-left ?p)
  RIGHT: (oriented-right ?p)
```

9. `connections`: Indica la relación de conectividad entre las celdas del juego. Esto obliga a que en el dominio se represente el mapa del juego como una cuadrícula. Por tanto, tienen que haberse declarado una serie de predicados que indiquen las conexiones entre las celdas. La clave `UP` indica que una celda está encima de la otra, `DOWN` que una está debajo de la otra, `LEFT` que una está a la izquierda de la otra y `RIGHT` que una está a la derecha de la otra. Un ejemplo de esto se puede ver a continuación:

```
connections:
  UP: (connected-up ?c ?u)
  DOWN: (connected-down ?c ?d)
  LEFT: (connected-left ?c ?l)
  RIGHT: (connected-right ?c ?r)
```

Es importante destacar que las variables que aparecen en estos predicados no tienen nada que ver con las que se han definido en anteriores campos. Cuando se recorra el mapa para generar los relaciones de conectividad la variable `?c` representará la celda actual, la cual está en la posición (x, y). La variable `?u` representará la celda superior a la actual situada en la posición $(x, y-1)$, `?d` la inferior respecto a la actual situada en la posición $(x, y+1)$, `?l` la izquierda respecto a la actual situada en la posición $(x-1, y)$ y `?r` la derecha respecto a la actual situada en la posición $(x+1, y)$.

10. `actionsCorrespondence`: Correspondencia entre las acciones definidas en el dominio `PDDL` y las acciones de `GVGAI`. Las acciones `PDDL` se obtienen a partir del archivo de dominio. Si el nombre de la acción `PDDL` contiene ciertos patrones, se produce una traducción automática a una acción de `GVGAI`:

    - Si contiene *up*, la acción se traduciría como `ACTION_UP`.
    - Si contiene *down*, la acción se traduciría como `ACTION_DOWN`.
    - Si contiene *left*, la acción se traduciría como `ACTION_LEFT`.
    - Si contiene *right*, la acción se traduciría como `ACTION_RIGHT`.
    - Si contiene *use*, la acción se traduciría como `ACTION_USE`.
    - Si no contiene ninguno de los patrones anteriores, se traduciría como `null`, dejándole al usuario la tarea de asignarle un acción de `GVGAI`, si es que la hay.

    Obviamente, el usuario puede modificar las traducciones a su gusto. Las acciones que tienen como traducción `null` no se tendrán en cuenta a la hora de traducir los planes. También, si no se quiere que se tenga en cuenta una acción, bien porque no tenga traducción o bien por otro motivo, simplemente bastaría con eliminar dicha acción.

11. `goals`: Representa la lista de objetivos definida por el usuario. Un objetivo está compuesto por un predicado `PDDL` que representa lo que se quiere alcanzar, la prioridad del objetivo, una opción que indica si guardar el objetivo o no una vez que este ha sido alcanzado y un campo opcional que contiene una lista de objetivos alcanzados y guardados que deben ser eliminados. Cuanto menor sea el valor de la prioridad, más prioridad tendrá dicho objetivo. Puede darse el caso de que haya más de un objetivo con la misma prioridad, lo cual implica que no importa el orden en el que se completen dichos objetivos. Guardar un objetivo sirve para incluirlo en los próximos estados `PDDL` que genere el traductor de estados de observación, ya que puede ser que al alcanzar dicho objetivo se produzca un cambio en el juego que debe ser considerado para alcanzar objetivos posteriores. Eliminar un objetivo guardado una vez que se ha alcanzado otro implica que algo que anteriormente era verdad ha dejado de serlo, y por tanto, el objetivo a eliminar no será incluido en posteriores estados `PDDL`. Obviamente, para que esto suceda se tiene que haber alcanzado el objetivo a eliminar previamente. A continuación se puede ver un ejemplo de esto:

```
goals:
- goalPredicate: (got g_7_6)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_6_6)
```

```
  priority: 1
  saveGoal: no
- goalPredicate: (got g_5_6)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_4_6)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_4_1)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_5_1)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_6_1)
  priority: 1
  saveGoal: no
- goalPredicate: (got g_7_1)
  priority: 1
  saveGoal: no
- goalPredicate: (exited-level)
  priority: 2
  saveGoal: no
- goalPredicate: (got g_6_2)
  priority: 1
  saveGoal: no
```

Aquí se indica que el agente primero debe recoger 9 gemas en cualquier orden (debido a que tienen la misma prioridad) y que, una vez que las ha obtenido, debe salir del nivel. Ninguno de estos objetivos tiene que ser guardado, y ninguno tiene una lista de objetivos a eliminar una vez que se haya alcanzado. Es importante destacar que, a la hora de indicar los objetivos, las variables de los predicados se sustituyen por objetos (instancias concretas de las variables). Por ejemplo, si nos fijamos en el primer objetivo, vemos que el predicado con la variable instanciada es `(got g_7_6)` (hay una gema en la posición $x = 7, y = 6$), mientras que el predicado como tal sería `(got ?g - Gem)`. Por tanto, es necesario conocer cómo se instancian las variables, así como también la posición de los elementos en el mapa.

CAPÍTULO 7

Diseño *software* del sistema

7.1. Implementación

El sistema completo ha sido implementado utilizando `Python` y `Java`. El primer lenguaje se ha utilizado para crear el *script* que permite automatizar la generación de archivos de configuración plantilla, mientras que el segundo se ha utilizado para implementar el resto de la arquitectura, ya que el entorno `GVGAI` fue originalmente escrito en `Java`. Todo el código fuente se encuentra disponible en un repositorio público de `GitHub` [12], con lo cual cualquiera puede acceder a él.

Si se desean consultar más detalles sobre la implementación, se recomienda consultar el documento adjunto, el cual contiene más información.

7.2. Arquitectura *software* del sistema

El sistema se ha implementado mediante una serie de clases, las cuales posteriormente se han agrupado en un paquete, creando por tanto un módulo *software*. En el diagrama de clases de la figura 7.1 se puede observar qué clases componen el paquete, cuáles son las relaciones entre ellas, qué información guardan y qué funcionalidades exponen.

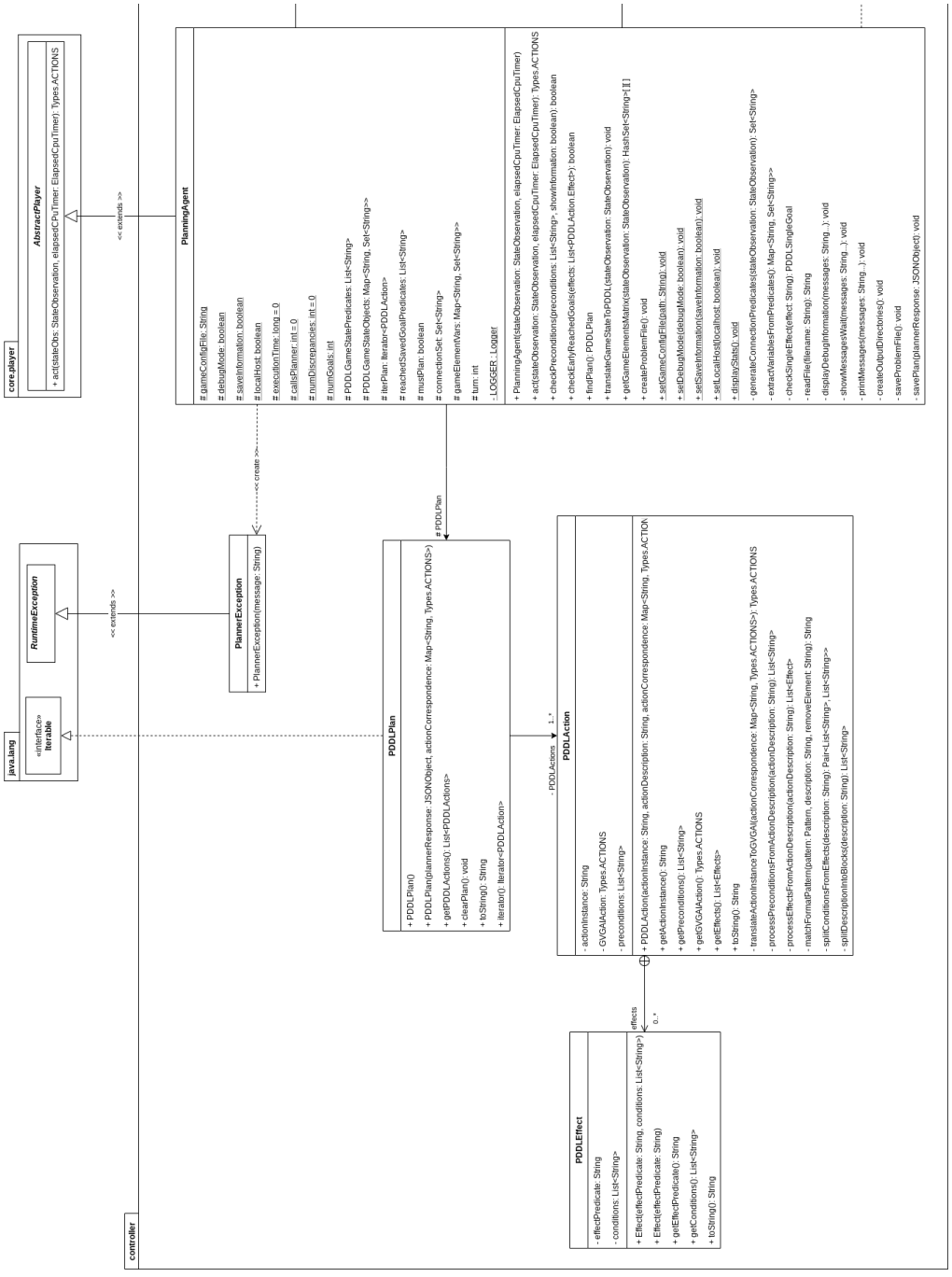
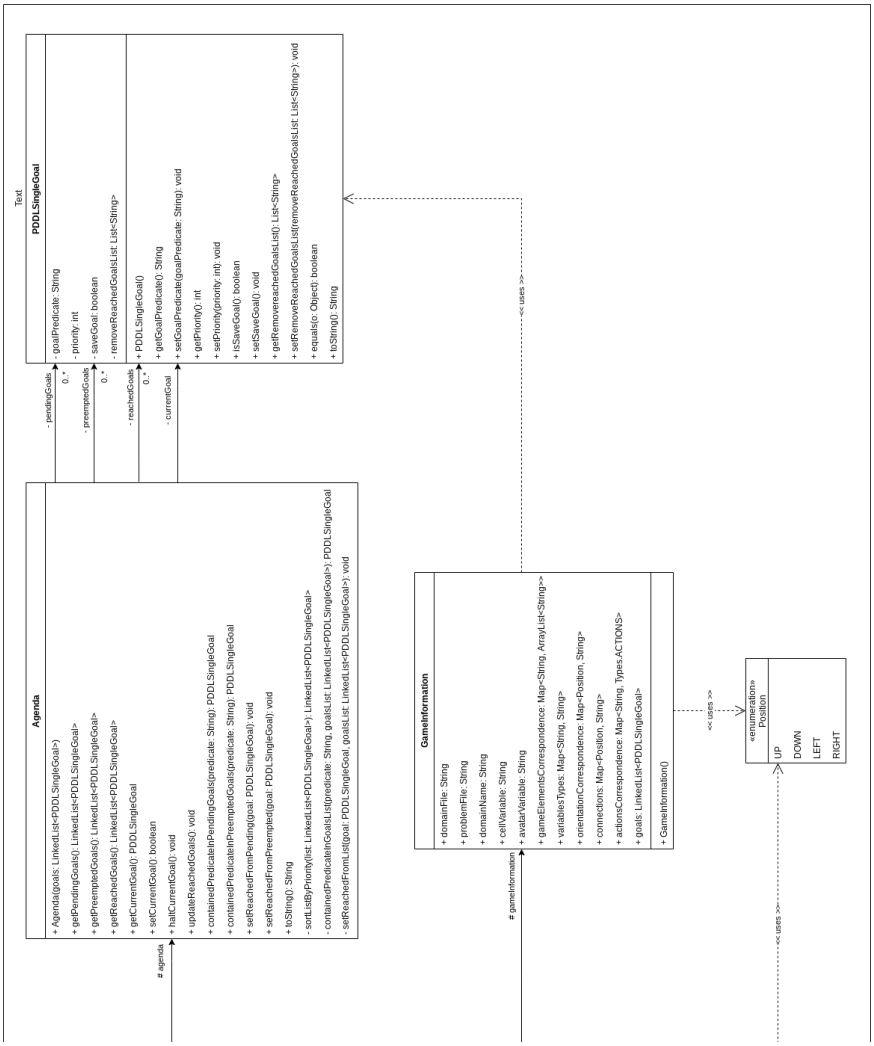


Figura 7.1: Diagrama de clases.


Text
PDDLSingleGoal
- goalPredicate: String
- priority: int
- saveGoal: boolean
- removeReachedGoalsList: List<String>
+ PDDLSingleGoal()
+ getGoalPredicate(): String
+ setGoalPredicate(goalPredicate: String): void
+ getPriority(): int
+ setPriority(priority: int): void
+ isSaveGoal(): boolean
+ setSaveGoal(): void
+ getRemoveReachedGoalsList(): List<String>
+ setRemoveReachedGoalsList(removeReachedGoalsList: List<String>): void
+ equals(o: Object): boolean
+ toString(): String
- pendingGoals
0..*
- preemptedGoals
0..*
- reachedGoals
0..*
- currentGoal
Agenda
+ Agenda(goals: LinkedList<PDDLSingleGoal>)
+ getPendingGoals(): LinkedList<PDDLSingleGoal>
+ getPreemptedGoals(): LinkedList<PDDLSingleGoal>
+ getReachedGoals(): LinkedList<PDDLSingleGoal>
+ getCurrentGoal(): PDDLSingleGoal
+ setCurrentGoal(): boolean
+ haltCurrentGoal(): void
+ updateReachedGoals(): void
+ containedPredicateInPendingGoals(predicate: String): PDDLSingleGoal
+ containedPredicateInPreemptedGoals(predicate: String): PDDLSingleGoal
+ setReachedFromPending(goal: PDDLSingleGoal): void
+ setReachedFromPreempted(goal: PDDLSingleGoal): void
+ toString(): String
- sortListByPriority(list: LinkedList<PDDLSingleGoal>): LinkedList<PDDLSingleGoal>
- containedPredicateInGoalsList(predicate: String, goalsList: LinkedList<PDDLSingleGoal>): PDDLSingleGoal
- setReachedFromList(goal: PDDLSingleGoal, goalsList: LinkedList<PDDLSingleGoal>): void
agenda
<< uses >>
GameInformation
+ domainFile: String
+ problemFile: String
+ domainName: String
+ cellVariable: String
+ avatarVariable: String
+ gameElementsCorrespondence: Map<String, ArrayList<String>>
+ variablesTypes: Map<String, String>
+ orientationCorrespondence: Map<Position, String>
+ connections: Map<Position, String>
+ actionsCorrespondence: Map<String, Types.ACTIONS>
+ goals: LinkedList<PDDLSingleGoal>
+ GameInformation()
# gameInformation
<< uses >>
«enumeration»
Position
UP
DOWN
LEFT
RIGHT
<< uses >>

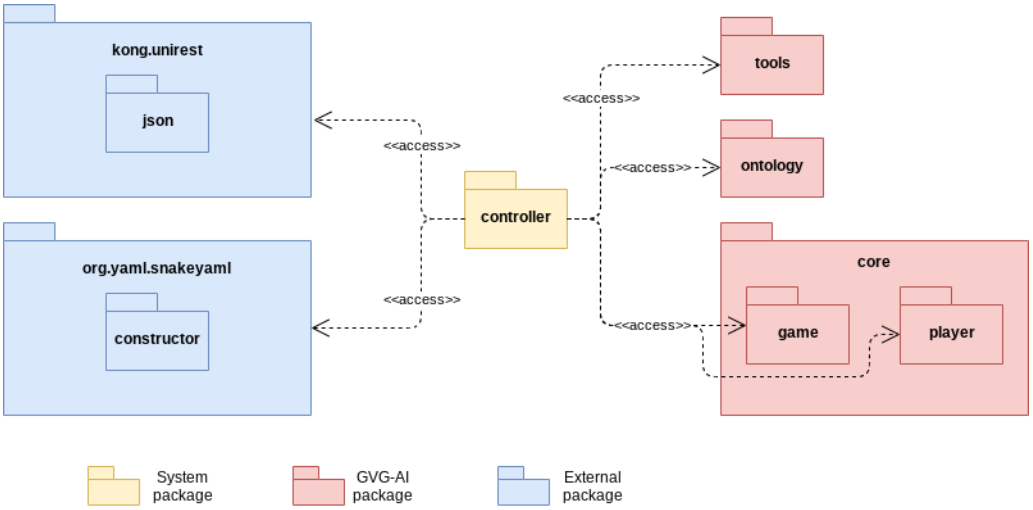


Figura 7.2: Diagrama de paquetes.

En la figura 7.2 se puede apreciar cómo se relaciona el paquete que encapsula el sistema, `controller`, con otros paquetes. Puede verse que el paquete del sistema accede a las funcionalidades que se pueden encontrar en algunos de los paquetes que ofrece el entorno de `GVGAI`. También se puede observar que se accede a funcionalidad de módulos externos, como por ejemplo a `kong.unirest` y a su subpaquete `json` y al paquete `org.yaml.snakeyaml` y a su subpaquete `constructor`.

El paquete `kong.unirest` junto con su subpaquete `json` se han utilizado para hacer las peticiones al planificador, el cual ha sido comentado varias veces que es un componente que se encuentra desplegado en la nube, y que por tanto, ofrece una `API REST` a la cual se pueden hacer peticiones utilizando los métodos de `HTTP`.

El paquete `org.yaml.snakeyaml` junto con su subpaquete `constructor` se han utilizado para cargar el archivo de configuración en formato `YAML` a la correspondiente estructura de datos que se ha definido.

7.3. Descripción de las clases

Una vez que se ha visto cómo se ha estructurado el módulo *software*, vamos a pasar a comentar de forma general las clases que lo conforman.

7.3.1. La clase `PlanningAgent`

Esta es la clase que representa el agente implementado, el cual combina una componente deliberativa (la planificación) con una componente reactiva (el monitor). Hereda de la clase abstracta `AbstractPlayer`, la cual es proporcionada por el entorno de `GVGAI`. Como tal no se instancia si no que se le indica al entorno la ruta de la clase que se quiere ejecutar. Si se quiere configurar de alguna forma o acceder a las estadísticas una vez se ha completado el juego, se deben utilizar los métodos estáticos proporcionados (ver figura 7.1).

Aparte de disponer de atributos de configuración y de estadísticas, la clase dispone de un *logger* (atributo `LOGGER`) que permite guardar información de la ejecución en un *log* si se especifica la opción a la hora de ejecutar el sistema.

La mayoría de atributos están bastante claros, aunque se quieren destacar los siguientes, los cuales son de especial interés:

- `PDDLGameStatePredicates`: Lista que contiene todos los predicados `PDDL` del estado actual del juego. Estos predicados están asociados a las observaciones del juego y a las relaciones de conectividad entre las casillas.
- `PDDLGameStateObjects`: Diccionario que relaciona las variables definidas en el archivo de configuración con los objetos del juego que se han construido utilizando dicha variable (instancias concretas de la variable).
- `iterPlan`: Iterador que permite recorrer de forma sencilla el plan generado.
- `reachedSavedGoalPredicates`: Lista de predicados objetivo alcanzados que deben ser añadidos a los predicados del estado actual.
- `mustPlan`: Variable booleana que indica si se debe planificar o no.
- `connectionSet`: Conjunto que contiene los predicados asociados a las relaciones de conectividad de las celdas. Estos predicados se tienen guardados aparte ya que solo se generan una vez al principio de la partida y posteriormente se añaden a `PDDLGameStatePredicates`.
- `gameElementVars`: Diccionario que relaciona los elementos del juego y las variables asociadas a estos elementos.

De entre los métodos disponibles, se quieren comentar algunos de ellos debido a que son de especial importancia:

- `act`: Este es el método más importante de la clase y de todo el sistema. Se ejecuta en cada turno y devuelve la siguiente acción que tiene que ejecutar

el agente. Podría considerarse que es el **ejecutor del plan** del sistema, ya que es el que manda las acciones a ejecutar al motor del juego. Se encarga también de comunicarse con todos los demás módulos funcionales que se han descrito en los capítulos 4 y 5 para determinar la siguiente acción a ejecutar. Su funcionamiento puede verse en el diagrama de secuencia de la figura 7.3, donde se muestra cómo se comunica con los otros elementos.

- `checkPreconditions`: Este método es una parte del **monitor**, encargándose de determinar si se ha producido una discrepancia debido a que no se cumplen las precondiciones de una acción.

- `checkEarlyReachedGoals`: La otra parte del **monitor**, encargándose de comprobar si algún objetivo se ha alcanzado antes de tiempo.

- `findPlan`: En este método se lleva a cabo la llamada al planificador, lo que se correspondería la tarea realizada por el **planificador**. Dentro de él también se realiza la traducción del plan, aunque no es una tarea que realiza este método directamente.

- `translateGameStateToPDDL`: Este método traduce el estado de observación actual del juego a predicados `PDDL`, aunque solo lo hace para las observaciones. También añadiría los predicados que se encuentran en el atributo `reachedSavedGoalPredicated`. Por tanto, realiza una parte del trabajo que realiza el **traductor del estado de observación**.

- `generateConnectionPredicates`: Este método genera los predicados que representan las relaciones de conectividad entre las celdas. Por tanto, realizaría parte del funcionamiento del **traductor del estado de observación**.

- `getGameElementsMatrix`: Este método transforma el estado de observación que es proporcionado por el motor del juego a una matriz que contiene los nombres de los elementos del juego en cada posición. Puede haber más de un elemento en una misma posición. Estos nombres se corresponden con los que hay en el archivo de configuración. Este método es indispensable para que se pueda hacer correctamente la traducción de los estados de observación. Realiza, por tanto, una parte del trabajo del **traductor del estado de observación**.

- `createProblemFile`: Este método genera el archivo de problemas. Por tanto, su funcionamiento se corresponde con el descrito para el **generador de problemas**.

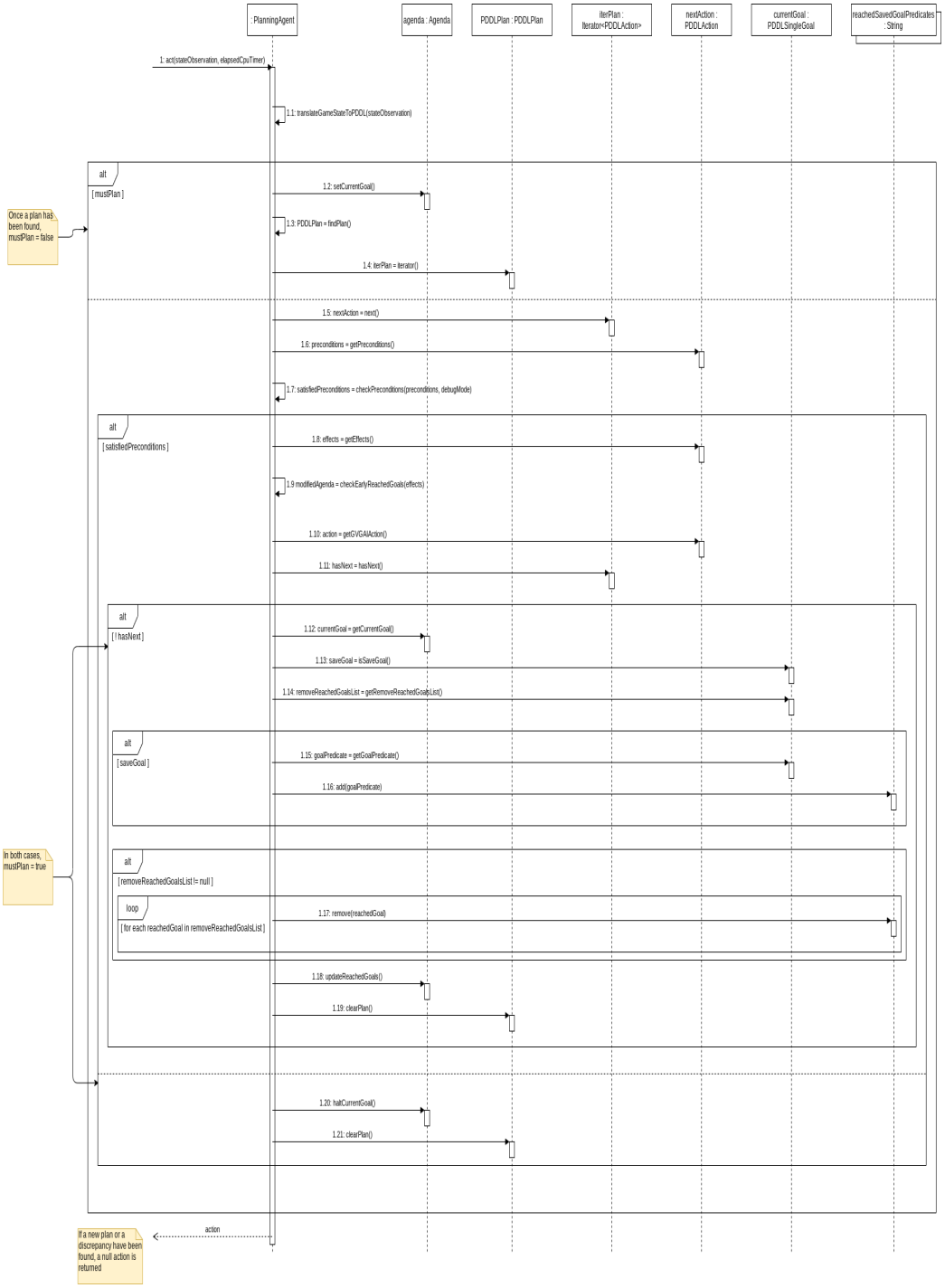


Figura 7.3: Diagrama de secuencia del método `act()`.

7.3.2. La clase `GameInformation`

Esta clase representa la información del juego que se ha definido en el archivo de configuración. Tiene exactamente los mismos campos que dicho archivo. Como son todos públicos, se puede instanciar la clase y darle valores a dichos atributos. Sin embargo, es mucho más sencillo utilizar la funcionalidad ofrecida por el paquete `org.yaml.snakeyaml` para crear directamente una nueva instancia con la información del archivo.

7.3.3. La clase `PDDLSingleGoal`

La clase representa un objetivo a alcanzar. Los atributos que la componen son el predicado objetivo (`goalPredicate`), la prioridad del objetivo (`priority`), un booleano que indica si guardar o no el objetivo (`saveGoal`) y una lista de predicados a eliminar cuando éste sea alcanzado (`removeReachedGoalsList`).

Para crear una instancia se podría llamar al constructor y usar los *setters* para dar valores. Sin embargo, en el sistema lo que se hace es cargar los objetivos directamente cuando se crea la instancia de `GameInformation` a partir del archivo de configuración, ya que éstos aparecen reflejados ahí.

7.3.4. La clase `Agenda`

Esta clase se encarga de gestionar los objetivos, de forma que es la que realiza la tarea del **gestor de objetivos**. Para crear una instancia se necesita una lista de objetivos. Tal y como pasaba antes, esto es muy fácil de hacer cuando se carga el archivo de configuración y se obtiene dicha lista accediendo al atributo `goals` de la clase `GameInformation`.

Tal y como veíamos en el capítulo 5, un objetivo podía estar en diversos estados. Para representar estos estados se tienen tres listas: una lista de objetivos no planificados (`pendingGoals`), una lista de objetivos detenidos (`preemptedGoals`) y una lista de objetivos alcanzados (`reachedGoals`). Adicionalmente, se tiene un atributo que representa el estado de objetivo actual (`currentGoal`). Las listas `pendingGoals` y `preemptedGoals` están ordenadas por prioridad.

Para seleccionar el objetivo actual se utiliza el método `setCurrentGoal`. El objetivo actual se selecciona de la siguiente forma:

1º Si `pendingGoals` contiene objetivos y `preemptedGoals` no, se selecciona el primero de `pendingGoals`.

2º Si `preemptedGoals` contiene objetivos y `pendingGoals` no, se selecciona el primero de `preemptedGoals`.

3º Si ambas listas contienen elementos, se comparan las prioridades del primer objetivo de cada lista y se escoge el de menor prioridad. En caso de empate, se decide escoger el objetivo de `pendingGoals` para así explorar nuevos objetivos.

4º Si no quedan objetivos en ambas listas, significa que se han alcanzado todos los objetivos.

El método `haltCurrentGoal` mueve el objetivo actual a la lista `preemptedGoals` y establece que no hay ningún objetivo actual, de forma que se debe seleccionar uno nuevo.

7.3.5. La clase `PDDLAction`

Esta clase representa una instancia de una acción `PDDL`. Está formada por la acción instanciada (`actionInstance`), la correspondiente acción `GVGAI` (`GVGAIAction`), una lista de precondiciones instanciadas (`preconditions`) una lista de efectos instanciados (`effects`). Cuando se habla de instancias se hace referencia a que las variables han sido sustituidas por elementos concretos del estado de observación actual del juego, en formato `PDDL` como es de suponer. Las precondiciones y los efectos son proporcionados por el planificador como salida.

Una acción se crea cuando mientras se traduce el plan obtenido por el planificador. Dicho proceso se realiza en el constructor de la clase. Por tanto, el constructor realiza parte del trabajo que realiza el **traductor de planes**.

7.3.6. La clase `PDDLEffect`

Esta es una clase interna a `PDDLAction` y representa los efectos instanciados de una acción. Un efecto consta de un predicado de efecto (`effectPredicate`) y de una lista de condiciones (`conditions`). La existencia de condiciones implica que se trata de un efecto condicional, siendo necesario por tanto que se cumplan las condiciones para que el efecto se lleve a cabo.

La existencia de una instancia de esta clase depende de una instancia de `PDDLAction`, ya que si no existe una acción, no pueden existir efectos.

7.3.7. La clase `PDDLPlan`

Esta clase representa un plan traducido, o lo que es lo mismo, una secuencia de instancias de la clase `PDDLAction`. Es el constructor de esta clase el que construye dicho plan, construyendo cada acción de forma individual a partir de la respuesta del planificador. Además, el resultado obtenido es posteriormente filtrado, de manera que las acciones nulas (aquellas que no tienen una correspondencia) son eliminadas del plan. Por tanto, el constructor de la clase realiza parte del trabajo que realiza el **traductor del plan**. El método `iterator` devuelve un iterador que permite recorrer el plan de manera sencilla.

7.4. Validación del *software*

Comprobar el correcto funcionamiento de *software* de este tipo puede llegar a ser complicado y tedioso, ya que implicaría tener que ejecutar cada vez una partida y comprobar qué posibles errores se han producido a la hora de ejecutarlo o, en caso de que no se produzca ninguno, comprobar que el resultado obtenido es el esperado.

Para simplificar todo el proceso, se han creado una serie de tests unitarios que comprueban el correcto funcionamiento de las clases desarrolladas. De esta forma, antes incluso de ejecutar un juego se puede llegar a asegurar el correcto funcionamiento del código implementado y se pueden llegar a detectar errores que podrían afectar a su funcionamiento.

Dichos tests se ejecutan cuando se construye el archivo ejecutable del sistema, aunque también se ha automatizado su ejecución por medio de plataformas de integración continua (CI o *Continuous Integration*) como por ejemplo serían `Travis CI` o las recientemente creadas `GitHub Actions`, las cuales también permiten ejecutar tests. De esta forma, cada vez que se haga un cambio en el repositorio se van a ejecutar los tests definidos, con la ventaja además de que se pueden probar en múltiples sistemas operativos y versiones de éstos. En este caso, se han realizado pruebas en `Ubuntu 16.04`, `Ubuntu 18.04` y `Windows 10`. Desafortunadamente, no se han podido hacer pruebas en `macOS` debido a problemas de estas plataformas a la hora de crear el entorno de pruebas.

Los tests creados realizan pruebas sobre toda la interfaz pública de las clases `Agenda`, `PDDLAction` (y por tanto también se testea la clase `PDDLEffect`), `PDDLPlan` y `PDDLSingleGoal`. De esta forma se puede asegurar que las estructuras de datos se crean de forma correcta y que los métodos implementados tienen el comportamiento esperado.

Para la clase `PlanningAgent` también se han creado una serie de tests que realizan pruebas sobre una serie de métodos de la interfaz pública. Sin embargo, testear el método `act` es complicado, ya que implicaría tener que crear una instancia de un juego determinado, lo cual puede llegar a ser costoso. Para solventar eso, se han hecho pruebas sobre los métodos públicos de la clase que intervienen en la ejecución de `act`. De esta forma, si dichos métodos funcionan correctamente, se puede llegar a afirmar con bastante certeza que el método tendrá el comportamiento esperado. Por tanto, para comprobar que esto es así bastaría con realizar una ejecución con algún juego y comprobar el resultado. Como el resto de funcionalidades han sido testeadas, se puede asegurar que su comportamiento va a ser el esperado y que no van a generar errores durante la ejecución.

CAPÍTULO 8

Experimentación

Una vez que el sistema ha sido implementado y validado, es necesario realizar un estudio de su funcionamiento. Este estudio consiste en llevar a cabo una serie de experimentos, los cuales pretenden poner a prueba dos aspectos fundamentales del sistema:

1. La capacidad que tiene para generar problemas `PDDL` de forma automática a partir de los estados del juego.
2. La capacidad que tiene para responder a los cambios dinámicos que se producen en los juegos.

Para llevar a cabo estos experimentos se han escogido los siguientes juegos:

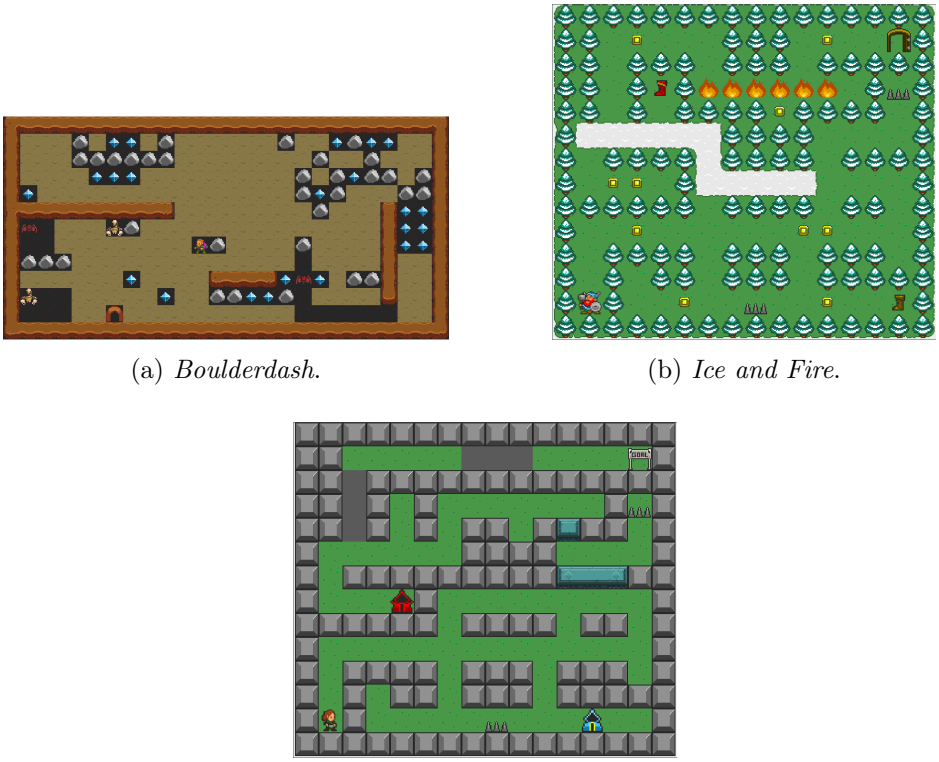
- ***Boulderdash***. El objetivo del juego consiste en recoger 9 de las gemas que hay esparcidas a lo largo de todo el mapa y llegar a la salida sin morir en el intento (ver figura 8.1a). En el mapa hay también esparcidos dos tipos de enemigos: las mariposas y los cangrejos, los cuales matarán al agente si chocan con él. Si dos enemigos chocan entre sí, se generará una gema nueva. En el juego original, las rocas son afectadas por la gravedad, de manera que si la casilla debajo de una de ellas está excavada (es decir, es una casilla vacía), la roca caerá hasta toparse con una casilla que contenga tierra, un muro,

una gema u otra roca. Una roca cayendo puede matar al agente si colisiona con este. Sin embargo, debido a la extrema dificultad que supone simular este sistema de físicas en un dominio PDDL, se trabaja con una versión simplificada del juego donde las rocas no caen y pueden ser excavadas utilizando el pico del que dispone el agente, de forma que estas rocas desaparecen del mapa.

Esta versión del juego, a pesar de ser simplificada, es no determinista, ya que el comportamiento de los enemigos es totalmente aleatorio. Este no determinismo introduce por tanto cierto dinamismo en el juego, ya que los enemigos pueden interponerse en el camino del agente y matarlo. Es aquí donde el monitor va a jugar un papel clave, ya que gracias a él se podrá determinar si algún enemigo se interpone entre el agente y el objetivo al que se dirige.

- ***Ice and Fire***. En este juego, el agente tiene que atravesar una especie de laberinto hasta la salida, el cual está lleno de obstáculos como pinchos, fuego y hielo (ver figura 8.1b). Estos obstáculos matarán al jugador si entra en contacto directo con ellos. Sin embargo, las casillas que contienen fuego pueden ser atravesadas si el agente dispone de unas botas de fuego. Análogamente, se pueden atravesar las casillas que contienen hielo si se tienen las botas de hielo. Se pueden llevar equipados los dos tipos de botas simultáneamente, de forma que se pueden atravesar ambos tipos de obstáculos. Los pinchos, por desgracia, no pueden atravesarse. A lo largo del mapa hay también una serie de monedas que pueden ser recogidas para aumentar la puntuación del jugador, aunque esto es más bien secundario. Este juego es totalmente determinista ya que, al no haber enemigos u otros personajes en el mapa, no se pueden llegar a producir eventos exógenos que obliguen a modificar un plan obtenido.

- ***Labyrinth Dual***. Aquí el agente también tiene que atravesar un laberinto hasta la salida esquivando los pinchos y atravesando una serie de casillas para las que requiere un traje en específico de los que se pueden encontrar por el mapa (ver figura 8.1c). El traje rojo permite atravesar los muros de color gris oscuro y el traje azul permite atravesar las muros de color azul. El agente solo puede llevar equipado un traje a la vez. Si ya tiene equipado uno y coge el otro, se cambia de traje, perdiendo la capacidad de atravesar las casillas que antes podía. Por tanto, hay que tener cierto cuidado a la hora de coger un traje, ya que en algunas ocasiones cambiar de traje puede llegar a encerrar al agente, impidiéndole llegar a la meta. A pesar de eso, el juego es totalmente determinista, ya que no hay factores externos que puedan impedir que un plan hasta un objetivo determinado sea llevado a cabo.



(a) *Boulderdash.* (b) *Ice and Fire.*

(c) *Labyrinth Dual.*

Figura 8.1: Juegos con los que se han realizado los experimentos.

Para cada juego se ha creado su dominio `PDDL`, intentando hacer una representación lo más fiel posible del juego. En la tabla 8.1 puede el número de predicados, de tipos y de acciones que conforman cada uno de los dominios. Adicionalmente, debido a que la planificación es un proceso lento, es importante destacar que a la hora de construir el sistema se han modificado tanto el tiempo máximo que tiene el agente para dar una respuesta, el cual ha pasado a ser 40 segundos (originalmente 40 milisegundos), como el tiempo máximo que se espera antes de descalificar al agente, el cual es ahora 500 segundos (anteriormente 50 milisegundos).

Juego	**Predicados**	**Tipos**	**Acciones**
Boulderdash	15	9	17
Ice and Fire	13	10	14
Labyrinth Dual	12	9	14

Cuadro 8.1: Información sobre los dominios creados para cada juego.

8.1. Generación de problemas

Para comprobar la generación de problemas se han hecho pruebas con cada uno de los cinco niveles de los tres juegos anteriormente mencionados. Para cada nivel de cada juego se ha creado un archivo de configuración en el cual se ha especificado la información correspondiente, además de decir los objetivos concretos que se deben alcanzar en cada nivel. En cada ejecución se ha obtenido información del número de predicados y objetos que que conforman el estado inicial del primer archivo de problema generado[1], el tiempo total de ejecución desde que empieza el juego, el tiempo de generación de los predicados de conectividad entre casillas, el tiempo medio de traducción de un estado de observación, el tiempo medio de generación un archivo de problema y la suma de los tres tiempos anteriores, lo cual nos da una idea de cuánto se tarda en generar un problema completo de media. Se han realizado 3 ejecuciones de cada nivel, de modo que los resultados obtenidos para los tiempos son una media de esas tres ejecuciones.

En la tabla 8.2 se pueden ver los resultados obtenidos. Como puede observarse, en todos los casos se generan problemas con más de 1000 predicados y cerca de 350-400 objetos. Puede verse también que el tiempo medio total para generar un problema es del orden de centésimas de segundo. Generar problemas de este tamaño a mano podría llevar días e incluso semanas, además de que se pueden cometer errores a la hora de crearlos, lo cual puede retrasar dicha creación aún más tiempo. Por tanto, hacer uso de esta arquitectura permite ahorrar muchísimo tiempo de trabajo y permite detectar de forma más rápida los posibles errores que puedan aparecer a la hora de crear un archivo de problema.

Si observamos los tiempos individuales que suman dicho tiempo total de generación de problema, se puede ver que las fases de traducción del estado de observación y la generación del archivo de problema tienen un tiempo medio del orden de milésimas de segundo, mientras que el proceso que más tarda es la generación de los predicados de conectividad, el cual tarda un tiempo medio del orden de las centésimas de segundo. Esto es normal, ya que los predicados de conectividad representan una gran parte de los predicados del problema. Para un mapa dado de tamaño $w \times h$, siendo w la anchura y h la altura, el número de predicados de conectividad, PC, viene dado por la expresión (8.1):

$$PC = 4 \cdot 2\ pred + 2((w-2) + (h-2)) \cdot 3\ pred + (w-2)(h-2) \cdot 4\ pred \quad (8.1)$$

[1] En general, va a haber muy poca variabilidad entre dichos números para archivos de problema posteriores, y como en algunos juegos hay más elementos al principio del juego, esto nos da una idea de cuánto se tarda en generar el problema más grande.

Juego	Nivel	Objetivos	Predicados problema inicial	Objetos problema inicial	Tiempo medio ejecución (s)	Tiempo medio generación predicados conectividad (s)	Tiempo medio traducción estado de observación (s)	Tiempo medio generación archivo de problema (s)	Tiempo medio total generación problema (s)
Boulderdash	**0**	10	1735	400	0.4967	0.0313	0.0024	0.0038	0.0375
	1	10	1721	393	0.428	0.0323	0.0029	0.0039	0.0392
	2	10	1717	391	0.5293	0.0347	0.0034	0.0044	0.0425
	3	10	1733	399	0.7487	0.0327	0.0049	0.0048	0.0423
	4	10	1731	398	0.4403	0.0317	0.0030	0.0039	0.0386
Ice and Fire	**0**	3	1215	391	0.417	0.027	0.0026	0.003	0.0326
	1	3	1234	410	0.4827	0.0307	0.0028	0.0056	0.0390
	2	3	1235	411	0.522	0.0273	0.0031	0.0045	0.0349
	3	3	1219	395	0.4587	0.0323	0.0029	0.0061	0.0413
	4	3	1210	386	0.697	0.0273	0.0031	0.0055	0.0359
Labyrinth Dual	**0**	3	1085	369	0.4503	0.0303	0.0029	0.0043	0.0376
	1	2	1069	380	0.3847	0.0303	0.0027	0.0045	0.0376
	2	3	1073	378	0.4107	0.0287	0.0024	0.0029	0.0339
	3	3	1083	376	0.41	0.0297	0.0031	0.0052	0.038
	4	2	1079	363	0.5623	0.029	0.0029	0.0062	0.0381

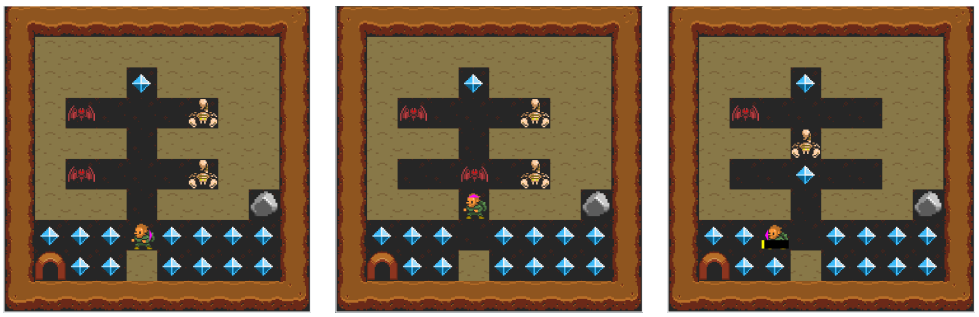
Cuadro 8.2: Resultados de la experimentación.

8.2. Respuesta a cambios dinámicos

Para facilitar el estudio de la respuesta que ofrece el sistema a los cambios dinámicos en los juegos, se ha creado un nuevo nivel del juego *Boulderdash*, que es el único de la lista que contiene no determinismo al tener enemigos que se mueven por el mapa aleatoriamente, y se ha fijado una semilla aleatoria, de forma que el comportamiento de los enemigos fuese el deseado. El nuevo nivel puede observarse en la figura 8.2a. Se ha creado también un archivo de configuración en el cual se han especificado qué gemas tiene que coger el agente antes de salir del nivel. La primera de ellas es la gema que está justo encima suyo, la cual está rodeada de enemigos.

Después de obtener un plan hasta dicha gema, comienza la ejecución de éste. Mientas el agente intenta avanzar, una mariposa se le cruza en el camino, tal y como puede verse en la figura 8.2b. El monitor, al hacer un estudio de las precondiciones de la siguiente acción a ejecutar, detecta una discrepancia, ya que la celda justo arriba del agente está ocupada por un enemigo, algo que no se esperaba. Es entonces cuando se le indica al gestor de objetivos que se ha producido una discrepancia y que se debe seleccionar un nuevo objetivo, que resulta ser la gema que se encontraba a la izquierda de la posición inicial del agente.

La detección de la discrepancia dispara un proceso de replanificación con este nuevo objetivo. Esto genera un nuevo plan hasta dicha gema y, tal y como puede observarse en la figura 8.2c, el agente alcanza el nuevo objetivo que se había propuesto. Vemos por tanto que el agente ha conseguido responder no solo mediante técnicas reactivas, si no autoproponiéndose un nuevo objetivo, a la aparición de un enemigo en su camino, deteniendo la ejecución del plan actual y cambiando de objetivo en el proceso.



(a) Estado inicial del juego. (b) Mariposa bloquea el camino. (c) Agente cambia de objetivo y lo alcanza.

Figura 8.2: Ejemplo de respuesta a cambios dinámicos en el entorno en *Boulderdash*.

CAPÍTULO 9

Conclusiones

En este trabajo hemos propuesto una arquitectura semiautomática que combina un componente deliberativo guiado por la planificación y un componente reactivo que permite la ejecución y monitorización de los planes obtenidos, detectando en el proceso cualquier tipo de situación imprevista que pueda surgir.

Tal y como se ha podido ver, esta nueva arquitectura, a diferencia de otras propuestas diseñadas para manejar un único dominio de planificación para un único juego, permite resolver distintos juegos mediante el uso de planificación. Esto es posible siempre y cuando dichos juegos puedan ser resueltos mediante este tipo de técnicas. Esto se ha conseguido gracias a la definición de un proceso semiautomático para la integración del planificador, basado en archivos de configuración que permiten al usuario establecer tanto los parámetros que va a utilizar el sistema para traducir estados de observación del juego a predicados `PDDL` como los objetivos que se desea alcanzar en la ejecución del sistema.

Un punto fuerte de esta arquitectura es que simplifica mucho la creación de problemas de planificación `PDDL`, ya que permite obtenerlos automáticamente a partir de los estados de observación del juego y en muy poco tiempo, como se ha podido comprobar, incluso para problemas grandes. Si se intentase hacer el proceso de integración de un planificador en un juego, el cual requiere de la definición de un dominio y de la generación de problemas específicos para el juego, se podrían tardar días, e incluso semanas.

Por último, cabe destacar que esta propuesta puede ser utilizada en dos ámbitos. Por una parte, puede utilizarse para experimentar con arquitecturas deliberativas dirigidas por un planificador en el entorno `GVGAI`. Por otra, puede ser utilizada con fines educativos, permitiendo a los estudiantes entender la generación de problemas a partir de dominios de planificación `PDDL` y comprender mejor cómo funciona la planificación.

9.1. Trabajos futuros

A pesar de los éxitos logrados en este trabajo, se considera que todavía hay cierto margen de mejora. En este punto se proponen una serie de ideas que permitirían mejorar el sistema y que pueden servir como trabajos futuros. Es importante destacar que estas propuestas tienen que permitir la resolución de múltiples juegos, de forma que integrar estos cambios en el sistema puede suponer un auténtico reto. Las propuestas de mejora que se hacen son las siguientes:

- **Mejora del comportamiento reactivo**. El comportamiento reactivo que se presenta en este trabajo es simple. Cuando el agente va a ejecutar la siguiente acción, comprueba que sus precondiciones se cumplen, y en caso de que no, cambia de objetivo. Este comportamiento, a pesar de que funciona en la mayoría de los casos, tiene el problema de que no tiene en mayor consideración el entorno del agente. Muchas veces, el agente puede llegar a morir aun sin haberse producido una discrepancia, debido a que por ejemplo ha aparecido un enemigo que no ha podido detectar al comprobar las precondiciones de la acción. Para solventar esto, se puede modificar el comportamiento reactivo del agente, incluyendo alguno de los controladores reactivos que se proporcionan en `GVGAI` o creando uno propio. De esta forma, el comportamiento reactivo podría permitir sortear peligros que se detecten en el entorno del agente, y después aplicar algún método para reparar el plan obtenido por el planificador.

- **Integración de un módulo de *goal reasoning***. Muchas nuevas propuestas de arquitecturas deliberativas basadas en planificación incluyen un módulo de *goal reasoning*, como por ejemplo la que se puede ver en [11]. Este módulo permite generar objetivos de forma automática a partir del estado del juego. Dichos objetivos son comunicados al planificador, el cual se encarga de encontrar un plan hasta ellos. Puede resultar de interés integrar un módulo de este tipo en el sistema ya que, si la arquitectura es capaz de proponerse ella misma los objetivos de forma automática, el usuario solo tendría que especificar qué predicados del dominio de planificación `PDDL` que ha definido se pueden considerar como objetivo y el módulo, a partir de esta información

y de la información de traducción proporcionada, podría plantear objetivos de forma automática. En esta descripción, sin embargo, se omite todo el trabajo que supone entrenar un módulo de este tipo, además de hacerlo lo más general posible para que pueda resolver múltiples juegos.

Bibliografía

[1] Malik Ghallab y col. “PDDL - The Planning Domain Definition Language”. En: (ago. de 1998).

[2] Santiago Ontañón y col. “Case-Based Planning and Execution for Real-Time Strategy Games”. En: *Case-Based Reasoning Research and Development*. Ed. por Rosina O. Weber y Michael M. Richter. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, págs. 164-178. ISBN: 978-3-540-74141-1.

[3] Vidal Alcázar. “pelea : Planning , Learning and Execution Architecture”. En: 2010.

[4] David Churchill y Michael Buro. “Build Order Optimization in StarCraft”. En: *AIIDE*. 2011.

[5] Ben George Weber, Michael Mateas y Arnav Jhala. “Building Human-Level AI for Real-Time Strategy Games”. En: *AAAI Fall Symposium: Advances in Cognitive Systems*. 2011.

[6] Tom Schaul. “A video game description language for model-based or interactive learning”. En: *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (2013), págs. 1-8.

[7] Martin Černý y col. “To Plan or to Simply React? An Experimental Study of Action Planning in a Game Environment”. En: *Comput. Intell.* 32.4 (nov. de 2016), págs. 668-710. ISSN: 0824-7935. DOI: 10.1111/coin.12079. URL: https://doi.org/10.1111/coin.12079.

[8] Malik Ghallab, Dana Nau y Paolo Traverso. *Automated Planning and Acting*. 1st. USA: Cambridge University Press, 2016. ISBN: 1107037271.

[9] Christian Muise. "Planning.Domains". En: *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*. 2016. URL: http://www.haz.ca/papers/planning-domains-icaps16.pdf.

[10] D. Perez-Liebana y col. "The 2014 General Video Game Playing Competition". En: *IEEE Transactions on Computational Intelligence and AI in Games* 8.3 (2016), págs. 229-243.

[11] David W. Aha. "Goal Reasoning: Foundations, Emerging Applications, and Prospects". En: *AI Magazine* 39.2 (jul. de 2018), págs. 3-24. DOI: 10.1609/aimag.v39i2.2800. URL: https://www.aaai.org/ojs/index.php/aimagazine/article/view/2800.

[12] Vladislav Nikolov Vasilev. *gvgai-pddl*. https://github.com/Vol0kin/gvgai-pddl. 2020.