TRABAJO FIN DE MÁSTER

MÁSTER CIENCIA DE DATOS E INGENIERÍA DE COMPUTADORES

Planificando en Videojuegos

Una Metodología para Generar Dominios de Planificación a partir de Descripciones de Dinámica en Videojuegos

Autor
Ignacio Vellido Expósito

Director
Juan Fernández Olivares

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, Septiembre de 2021

Una Metodología para Generar Dominios de Planificación a partir de Descripciones de Dinámica en Videojuegos

Ignacio Vellido Expósito

Palabras clave: *Ingeniería del Conocimiento*, *Planificación Automática*, *Inteligencia Artificial General*

Resumen

Partiendo de resultados previos en modelos de acciones jerárquicas, en este trabajo se propone una metodología para la generación automática de modelos de acciones a partir de una descripción de la dinámica de un entorno de videojuegos, al igual que su integración con un agente capaz de actuar en dicho entorno. Los modelos de acciones generados permiten a un planificador guiar el comportamiento deliberativo de un agente en una importante cantidad de videojuegos y, junto a un módulo reactivo, resolver niveles tanto deterministas como no deterministas. La experimentación realizada valida la metodología y demuestra que el esfuerzo requerido por un ingeniero del conocimiento en la definición de estos dominios tan complejos se reduce en gran medida. Adicionalmente, hemos generado un conjunto de benchmarks sobre los dominios que pueden ser de interés a la comunidad internacional para la evaluación de planificadores en competiciones internacionales de planificación.

A Methodology for the Automatic Generation of Planning Domains from Videogame Descriptions

Ignacio Vellido Expósito

Keywords: *Knowledge Engineering*, *Automated Planning*, *Artificial General Intelligence*

Abstract

Following previous work in hierarchical task networks, this project proposes a methodology for the automatic generation of action models from video game dynamics descriptions, as well as its integration with a planning agent for the execution and monitoring of the plans. Planners use these action models to get the deliberative behaviour for an agent in many different video games and, combined with a reactive module, solve deterministic and no-deterministic levels. Experimental results validate the methodology and prove that the effort put by a knowledge engineer can be greatly reduced in the definition of such complex domains. Furthermore, benchmarks of the domains has been produced that can be of interest to the international planning community to evaluate planners in international planning competitions.

Yo, **Ignacio Vellido Expósito**, alumno de la titulación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 79056166Z, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Ignacio Vellido Expósito

Granada, a 5 de septiembre de 2021

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

Informa:

Que el presente trabajo, titulado ***Una Metodología para Generar Dominios de Planificación a partir de Descripciones de Dinámica en Videojuegos***, ha sido realizado bajo su supervisión por **Ignacio Vellido Expósito**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 5 de septiembre de 2021.

El tutor:

Juan Fernández Olivares

Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto RTI2018-098460-B-I00 y por el fondo de garantía juvenil de la Junta de Andalucía.

También agradecemos a Carlos y Vladis todo el apoyo aportado en la realización de este proyecto.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los videojuegos llevan siendo desde hace décadas un buen campo de pruebas para evaluar técnicas de IA antes de ser empleadas en casos reales. Los investigadores han puesto un gran esfuerzo en llevar la IA a los videojuegos ya que estos se modelan, visualizan y comprenden de forma sencilla al mismo tiempo que comparten muchas características comunes con problemas de la vida real.

De la misma forma, la planificación automática [Ghallab et al., 2016] ha demostrado ser una rama potente de la IA en la búsqueda de secuencias de acciones para agentes inteligentes. Con solo conocimiento previo de las dinámicas del mundo y su estado actual, las técnicas de planificación son capaces de encontrar (si existe) la secuencia de acciones que alcanza un determinado objetivo.

En su enfoque en videojuegos, la planificación automática se ha adoptado previamente para guiar el comportamiento tanto de jugadores como personajes no jugables (NPCs) [Kelly et al., 2008, Hoang et al., 2005] mostrando múltiples ventajas, entre ellas que el agente cuenta con razonamiento deliberativo que le ayuda a perseguir objetivos en el videojuego, aumentando sus capacidades cognitivas.

La mayor problemática en este campo es que la creación de un dominio de planificación que represente con exactitud que represente fielmente la dinámica del mundo (en nuestro caso la dinámica de un videojuego) es una tarea larga, difícil y que debe ser tratada cuidadosamente. Este es un proceso de ingeniería del conocimiento que requiere modelar y escribir en un lenguaje de planificación (como PDDL [McDermott et al., 1998], actual estándar de facto) tanto los objetos del entorno, sus propiedades y relaciones, como las acciones que pueden ejecutarse y como estas transforman el mundo.

La construcción de un dominio por tanto conlleva realizar un proceso meticuloso de ingeniería del conocimiento de forma que se obtenga la representación más adecuada (*i.e.* simple) para el planificador y que en muchos casos, cuando

la complejidad del problema resulta excesiva, es en la práctica excesivamente costosa de obtener. Esto implica que si es necesario modelar más de un dominio para un campo de aplicación el proceso de representación y validación del conocimiento puede consumir más tiempo que del que disponga el ingeniero del conocimiento.

Por estos motivos creemos que utilizar un lenguaje próximo al campo de aplicación, combinado con un proceso de compilación hacia un lenguaje de planificación, reduce en gran medida el esfuerzo necesario en la integración de técnicas de planificación para resolver problemas. De esta manera un ingeniero del conocimiento solo debe centrarse en la abstracción y codificación de la información expresada en el lenguaje de partida como conocimiento de planificación, mientras que los expertos pueden dedicarse directamente a definir los requisitos de los problemas en su lenguaje habitual.

Para probar esta idea, este proyecto se centra en la aplicación de técnicas de planificación para resolver problemas de actuación en videojuegos, usando para ello un lenguaje simple pero expresivo como VGDL (**V**ideo **G**ame **D**escription **L**anguage) [Schaul, 2013] para describir tanto la dinámica del juego como los distintos escenarios (niveles). Este lenguaje ha sido utilizado previamente para definir más de 100 videojuegos en el entorno GVGAI [Perez-Liebana et al., 2016], un framework enfocado en evaluar técnicas de Inteligencia Artificial General (AGI).

El lenguaje de planificación que usaremos será el estándar aceptado por la comunidad, PDDL (**P**lanning **D**omain **D**efinition **L**anguage) [McDermott et al., 1998], con el que dispondremos una alta expresividad y gran gama de planificadores con los que experimentar.

Nuestra principal contribución es triple: (1) presentamos un proceso automático de ingeniería del conocimiento que, partiendo de la descripción de un videojuego en formato VGDL, produce un dominio PDDL representando los objetos del juego, las relaciones entre ellos y las dinámicas de estos modeladas como acciones; (2) esta arquitectura se integra con un agente deliberativo basado en planificación que permite alcanzar objetivos en los juegos, acompañado de un módulo que monitoriza la ejecución en busca de inconsistencias en situaciones no deterministas; (3) dada la naturaleza desafiante de los dominios producidos se incluye un estudio comparativo de planificadores siguiendo las pautas de la International Planning Competition 2018 con la esperanza de que los resultados sean de interés en futuras líneas de investigación.

Esta metodología abre la puerta a utilizar cualquier planificador basado en PDDL en una cantidad inmensa de videojuegos, comparándolos con aquellas técnicas de AGI que se han evaluado previamente en GVGAI. Adicionalmente, dada su naturaleza compleja y desafiante, los dominios pueden de resultar de interés a la comunidad internacional como testbed en futuras competiciones internacionales de planificación.

La estructura de esta memoria es la siguiente. En el capítulo 2 introducimos conceptos necesarios para lo comprensión de este trabajo junto a un análisis del estado del arte. Seguidamente, se muestra el proceso de desarrollo que ha seguido el proyecto. En el capítulo 4 se incluye una descripción global del funcionamiento del software y las partes que lo forman. En el capítulo 5 se detalla la implementación realizada para la construcción de cada una de las partes. Por último, en los dos últimos capítulos se muestra la experimentación realizada sobre cada caso de estudio, y se realiza un análisis de los logros del proyecto, las contribuciones que aporta y los posibles trabajos futuros que derivan de este.

Capítulo 2

Antecedentes y trabajos relacionados

En este capítulo se procede a introducir los conceptos necesarios para la comprensión de esta memoria, seguido de un análisis comparativo del estado del arte frente a nuestra aproximación al problema.

2.1. Antecedentes

En las siguientes subsecciones se describen los tres pilares sobre los que se sustenta nuestro proyecto, comenzando con el campo de la planificación automática y el estándar actual PDDL; continuando con el entorno de ejecución y el lenguaje de definición de videojuegos que se emplea; y terminando con el sistema utilizado para parsear los archivos de entrada.

2.1.1. Planificación automática

La planificación automática es un área de la IA dedicada al estudio de técnicas encargadas de resolver problemas de actuación de agentes, tanto físicos (e.g. robots), como virtuales (e.g. jugadores en videojuegos) [Ghallab et al., 2016].

Existen dos elementos necesarios para la resolución de problemas de planificación. Estos son:

- **Dominio de planificación**: En él se representa el modelo de acciones que el agente puede realizar, al igual que la dinámica del entorno, es decir, cómo las acciones del agente producen cambios que afectan al mundo.
- **Problema de planificación**: En él se representa (1) un estado preliminar que describe la situación inicial en la que se encuentra el agente, así como los objetos del entorno; y (2), un objetivo indicando la situación que debe alcanzar el agente.

Ambos elementos se proporcionan como entrada a un programa, llamado **planificador**, que lleva a cabo un proceso de búsqueda para generar un plan de actuación. Este plan indica la secuencia de acciones que, partiendo de la situación inicial, alcanza la situación objetivo.

El esquema de representación del conocimiento usado para describir tanto dominios como problemas de planificación es la **lógica de predicados**. A lo largo de los años se ha establecido un lenguaje estándar, denominado PDDL [McDermott et al., 1998], que permite la representación de dominios y problemas de planificación.

En PDDL los dominios se dividen en las siguientes partes:

- **Tipos** (objects): Clases de objetos presentes en el mundo, estructuradas de manera jerárquica.
- **Predicados** (predicates): Relaciones o características de los objetos.
- **Funciones** (functions): Predicados que almacenan un valor numérico.
- **Primitivas** (actions): Operadores que cambian el mundo. Cuentan con tres partes: (i) una lista de **parámetros** implicados, indicando las instancias y sus tipos; (ii) una sección de **precondiciones**, formada por predicados que deben ser ciertos cuando se produzca la acción, o por funciones que deben tener un valor concreto especificado en la precondición[1]; (iii) y otra de **efectos**, representando los cambios que se producen en el estado del mundo.

En la figura 2.1 podemos ver un ejemplo de las diferentes partes definidas para un dominio concreto. Se puede apreciar cómo la definición de tipos sigue el formato **[lista de tipos] - padre**; cómo tanto a los predicados como a las funciones se le asocian unos tipos concretos, usando el esquema **?nombre - tipo**; y cómo se define una acción junto a la utilización de sus parámetros.

Por otro lado, un problema PDDL cuenta con tres partes:

- **Objetos** (objects): Lista de instancias presentes en el problema.
- **Estado inicial** (init): Hechos que son ciertos al inicio del problema (compuestos por predicados y asignaciones de valores a las funciones).
- **Objetivo** (goal): Condiciones que deben ser ciertas al final del plan.

En la figura 2.2 tenemos un ejemplo de problema para el dominio de la figura 2.1, instanciando cuatro objetos (uno de tipo *jugador* y tres de tipo *oro*). Se puede apreciar cómo se afirman los predicados en la sección **:init** y cómo se establece un objetivo en la sección **:goal**.

[1] El uso de funciones está limitado a versiones de PDDL superiores a 2.1.

```
(:types
    ...
    jugador - jugador
    oro diamantes - monedas
)

(:functions
    ...
    (cantidad_oro ?j - jugador)
)

(:predicates
    ...
    (misma_posicion ?j - jugador ?o - oro)
)

(:action coger-oro
    :parameters (?j - jugador ?o - oro)
    :precondition (and
                      (misma_posicion ?j ?o)
                  )
    :effect (and
                (increase (cantidad_oro ?j) 1)
            )
)
```

Figura 2.1: Ejemplo de las diferentes partes de un dominio PDDL.

```
(:objects
    ...
    jugador1 - jugador
    oro1 oro2 oro3 - oro
)

(:init
    ...
    (misma_posicion jugador1 oro2)
)

(:goal (and
            (= (cantidad_oro jugador1) 3)
            ...
        )
)
```

Figura 2.2: Ejemplo de las diferentes partes de un problema PDDL.

Nuevas versiones de PDDL han ido extendiendo la expresividad y funcionalidad del lenguaje con el paso de los años. Añadidos como predicados numéricos, acciones temporales o continuas han permitido abarcar un mayor rango de problemas de manera más cómoda [Fox and Long, 2003].

Aunque algunas de estas características pueden ayudarnos a alcanzar una representación del conocimiento más simplificada, para mantener compatibilidad con la mayor cantidad de planificadores decidimos emplear la versión 1.2 original.

2.1.1.1. Ingeniería del conocimiento en planificación

La definición de un dominio que modele mundos complejos es una tarea complicada para el diseñador. Este debe contar con suficiente conocimiento como para poder definir todos los requisitos de manera precisa y sin errores. Las técnicas de Ingeniería del Conocimiento dentro de este campo pretenden facilitar y ayudar al diseñador en el proceso de construcción de dominios de mayor calidad.

Formalmente, la Ingeniería del Conocimiento en planificación (**K**nowledge **E**ngineering in **P**lanning and **S**cheduling, KEPS) [Vaquero et al., 2011] se encarga de la adquisición, formulación y validación de conocimiento para la definición de modelos de dominio en planificación.

Un **modelo de dominio en planificación** es una abstracción del contenido invariante de un dominio en los problemas sobre los que se puede emplear, generalmente correspondiendo a las diferentes partes que lo forman, como predicados, objetos, acciones, etc.

Tal y como explican [McCluskey et al., 2016], este modelo de dominio va más allá del uso clásico en otras ramas del desarrollo del software, donde este tipo de modelos se suelen representar mediante diagramas cuyo objetivo es descubrir los requisitos del software a producir. Los modelos de dominio en KEPS se suelen definir en un lenguaje formal independiente del usado para los dominios.

Gracias a estos modelos de dominio, podemos utilizar las técnicas de Ingeniería del Conocimiento para la producción de grandes cantidades de dominios con características comunes entre sí.

2.1.2. GVGAI

GVGAI (**G**eneral **V**ideo **G**ame **AI**) es un entorno para la creación y el testeo de agentes en múltiples entornos [Perez-Liebana et al., 2016]. GVGAI cuenta en su interior con más de 100 juegos de distinto tipo, formados por descripciones de dinámicas y niveles de juegos en el lenguaje VGDL. Aunque este framework está orientado a abordar problemas de Inteligencia Artificial General, creemos que puede ser un buen entorno de pruebas para técnicas de planificación, concretamente integrando planificación con actuación.

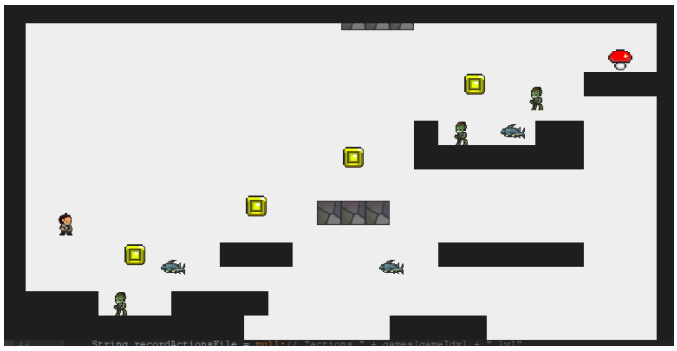


Figura 2.3: Imagen de Mario Bros en GVGAI.

El lenguaje subyacente que utiliza GVGAI es una variante en Java de VGDL, la cual contiene la mayoría de tipos definidos en la versión VGDL original y expande algunas características adicionales para adaptar otros tipos de juegos.

2.1.3. VGDL

VGDL (**V**ideo **G**ame **D**escription **L**anguage) [Schaul, 2013] es un lenguaje de alto nivel para la descripción de juegos, que se especializa en velocidad y simplicidad.

Existen tres variantes de este lenguaje: la original, implementada en Python [Schaul,]; su reimplementación [Vereecken, , VGDL 2.0], que actualmente sigue con soporte; y [GVGAI, , java-vgdl], la versión utilizada por GVGAI en su framework.

Este proyecto se centra, aunque no es exclusivamente, en java-vgdl. De esta manera existe la posibilidad de probar los dominios generados (y sus respectivos planes) mediante el entorno GVGAI a través de un agente basado en planificación.

Se ha escogido VGDL como lenguaje base por su facilidad de compresión sin comprometer una alta capacidad expresiva. Existe una alta gama de tipos predefinidos en VGDL, yendo desde tipos estáticos hasta varias clases de NPCs y agentes. Esto le permite representar una gran variedad de juegos, incluyendo no solo aquellos con mundos cuadriculados sino también con físicas 2D. Este lenguaje también cuenta con una gran potencialidad, pues en apenas 52 líneas puede definir juegos suficientemente complejos como el Mario Bros.

Un juego en VGDL se especifica con un archivo de descripción de juego (Game Description File. A partir de ahora, **GDF**), detallando las dinámicas; y un archivo por cada nivel indicando las posiciones iniciales de los objetos en cada uno de ellos (Level Description File. A partir de ahora, **LDF**).

```
SpriteSet
   user    > VerticalAvatar
   boulder > Missile        orientation=DOWN
```

(a) Definición de objetos y sus parámetros

```
LevelMapping
   u > user
   b > boulder
```

(b) Caracteres que representan cada objeto en el archivo de definición de nivel

```
InteractionSet
   user boulder > killIfFromAbove
```

(c) Ejemplo de interacción entre objetos

```
TerminationSet
   SpriteCounter stype=user limit=0 win=False
```

(d) Criterio de parada para el juego

Figura 2.4: Las cuatro partes en las que se compone una descripción de juego en VGDL.

El archivo GDF se estructura en cuatro partes:

- **SpriteSet**: Especificación de los objetos que participan en el juego (avatares, enemigos, misiles, etc.). Sigue el formato:

  ```
  nombre > tipo [lista_de_parámetros]
  ```

  En la figura 2.4a se muestran ejemplos de objetos y sus atributos, donde se puede ver que *boulder* es un misil que solo se mueve hacia abajo; y que *user* es un *VerticalAvatar*, es decir, un agente con dos movimientos posibles (*up* y *down*). En esta parte se permite la declaración de objetos de forma jerárquica, de forma que los hijos puedan heredar atributos y comportamientos de los padres. Esta herencia no es múltiple, un hijo solo puede derivar de un único padre. Los niveles de la jerarquía se definen en base a la indentación, yendo de padres a hijos.

- **LevelMapping**: Aquí se indican los caracteres que representan a los objetos en un archivo de descripción de nivel, donde cada línea de esta parte sigue el formato:

  ```
  carácter > lista_de_tipos
  ```

  La figura 2.4b muestra como *boulder* y *user* se simbolizan mediante las letras *b* y *u* respectivamente. Estos serán usados en la descripción de un nivel de juego (detallado más adelante).

- **InteractionSet**: En esta sección se indican las interacciones entre objetos, producidas mediante la colisión de ellos, siguiendo el formato:

  ```
  sprite1 sprite2 > tipo [lista_de_parámetros]
  ```

  Como ejemplo, en la figura 2.4c vemos una interacción que involucra a un objeto *boulder* y a un *user*, llamada *killIfFromAbove*. Es importante tener en cuenta que el orden en el que se definen los objetos es relevante, siendo

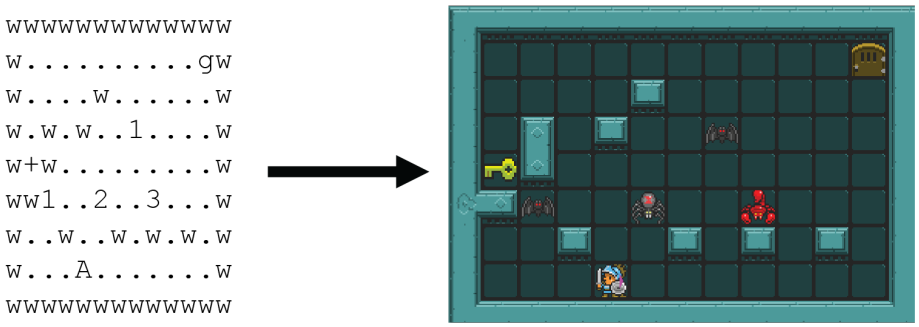


Figura 2.5: Representación de un nivel en GVGAI. Se puede ver cómo cada carácter representa una instancia en el juego. Por ejemplo: *w* significa *wall* (muro), *A* significa *avatar* y cada número un enemigo diferente.

el primero el que produce la interacción y el segundo el principal receptor de sus efectos, aunque en algunos casos más de un objeto se puede ver afectado por la interacción.

La definición de múltiples interacciones se puede resumir siguiendo el formato de GVGAI-VGDL, en el que se indican todos los objetos que producen una misma interacción para uno dado.

- **TerminationSet**: Se constituye por una serie de criterios que indican las condiciones de parada del juego. Cada uno de ellos sigue el formato:

```
condición [lista_de_parámetros] win=<True/False>
```

Podemos ver en la figura 2.4d un ejemplo, donde se especifica que cuando el número de objetos *user* actualmente presentes en el nivel llega a cero, el juego termina y el agente pierde la partida.

Un archivo de **descripción de nivel** (LDF) se compone de una matriz 2D de caracteres en cuyas celdas se define la posición inicial de una instancia de un objeto. Cada carácter se asocia con su objeto siguiendo las reglas definidas en el *LevelMapping*. La figura 2.5 muestra una transformación desde una descripción de un nivel a la visualización que produce GVGAI.

2.1.4. ANTLR

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) [Parr and Quong, 1995] es un generador de intérpretes que, siguiendo un lenguaje especial de definición de gramáticas, permite procesar y traducir textos o archivos binarios.

En este proyecto hemos elegido ANTLR para parsear la información de los ficheros VGDL debido a que resulta más sencillo de aprender que otros lenguajes similares. Su uso permite obtener un programa (llamado **listener**) capaz de leer un archivo de descripción de juego en VGDL y extraer a partir de él los datos relevantes en el proceso de parseo.

ANTLR tiene muchas características en común con el lenguaje usado en LEX. Principalmente se centra en la asociación de expresiones regulares a patrones que almacenan la información encontrada. Además, se permite la inclusión de código a ejecutar cuando se detecta cada una de las reglas.

Existen tres términos de relevancia a mencionar:

- **Lexer rules**: Reglas para detectar los tokens (unidad de información básica) de la gramática.
- **Parser rules**: Reglas para el reconocimiento de expresiones. Adicionalmente, se pueden asignar etiquetas a las diferentes opciones de las parser rules para poder referenciarlas en el listener resultante. De forma similar, también se permite asignarle nombres a conjuntos de tokens dentro de una regla.
- **Fragments**: Elementos reutilizables que ayudan al reconocimiento de tokens en las lexer rules, de cara a simplificar la definición de la gramática.

ANTLR puede devolver el intérprete en múltiples lenguajes, y en nuestro caso hemos especificado la salida a Python. Para este proyecto se ha utilizado la última versión de ANTLR en el momento de su uso, la 4.7.2.

2.2. Trabajos relacionados

Respecto a videojuegos, la planificación automática se considera desde hace tiempo una buena tecnología habilitadora en el modelado de comportamientos deliberativos en agentes. [Černỳ et al., 2016] muestra un análisis en detalle de trabajos basados en planificación en este campo. Pese a ello, hasta donde alcanza nuestro conocimiento, la mayoría de los trabajos se centran en evaluar el rendimiento de los planificadores, y los dominios son modelados manualmente. Habitualmente se define un único dominio y el planificador es testeado en varios problemas (*i.e.* escenarios, niveles). Por otra parte, aproximaciones como [Geffner and Geffner, 2015] van más allá, mostrando que los algoritmos de planificación

son competitivos respecto a técnicas estándar en juegos de Atari, pero ausentes de un modelo PDDL compacto para esos juegos.

En lo que se refiere a entornos de videojuegos, múltiples arquitecturas de planificación han sido integradas en StarCraft [Martínez and Luis, 2018] o Minecraft [Bonanno et al., 2016], con la idea de evaluar el rendimiento de arquitecturas de planificación con un único dominio. Respecto a GVGAI, algunas aproximaciones muestran controladores reactivos aprendiendo a jugar con relativo éxito [Torrado et al., 2018]. Sin embargo, estos agentes carecen de capacidades deliberativas y emplean modelos de caja negra, por lo que es imposible comprender el razonamiento tras las acciones del jugador.

Tal y como hemos dicho, estamos centrados en el entorno de videojuegos GVGAI. El único trabajo basado en planificación clásica que conocemos es [Couto Carrasco, 2015], donde los autores manualmente definen un controlador para juegos de puzles mediante PDDL. Nuestra propuesta va un paso más allá automatizando la generación de dominios de planificación para teóricamente cualquier juego descrito en VGDL, junto a un agente capaz de ejecutar y monitorizar los planes en GVGAI.

Previamente nos hemos enfocado en la generación de modelos HTN (Hierarchical Task Networks) a partir de descripciones de videojuegos [Vellido et al., 2020] aunque, al contrario que las aproximaciones de planificación clásicas, los modelos HTN necesitan una estrategia que debe ser codificada manualmente por un experto (o en el mejor de los casos, aprendida) para que el agente pueda actuar adecuadamente en un videojuego. De cara a evitar este problema en este proyecto hemos optado por usar un lenguaje no jerárquico como PDDL de manera que el planificador sea responsable de elegir la mejor secuencia de acciones para el agente sin necesidad de conocimiento experto.

Por otro lado, el problema de la generación de dominios de planificación a partir de descripciones en otros lenguajes se ha abordado previamente en el campo del eLearning [Castillo et al., 2010], de la gestión de procesos de negocio (BPM) [González-Ferrer et al., 2013], y de la representación de guías de práctica clínica [Fdez-Olivares et al., 2011]. En todos los casos se parte de un lenguaje próximo al dominio de aplicación (BPMN en un caso, Asbru en otro) y se generan dominios de planificación que se emplean para ayudar a expertos en la toma de decisiones. La novedad en nuestra propuesta reside en el uso de plantillas abstractas que individualizan el conocimiento del lenguaje de aplicación en conocimiento de planificación, permitiendo generar rápidamente tantos casos de estudio diferentes como el lenguaje de partida sea capaz de expresar.

Finalmente, es importante destacar que este trabajo supone una continuación de mi Trabajo de Fin de Grado [Vellido and Olivares, 2020], donde para la validación y experimentación de la metodología se ha utilizado y extendido el TFG de Vladislav Nikolov Vasilev [Vasilev and Olivares, 2020].

Capítulo 3

Plan de Trabajo

3.1. Metodología

El proyecto sigue un esquema clásico del ciclo de vida de un proceso de ingeniería del conocimiento, durante el cuál se ha seguido una metodología basada en Scrum. Se han realizado reuniones cada poco tiempo especificando objetivos realizables a corto plazo, revisando y debatiendo continuamente lo realizado en los pasos anteriores.

Esta metodología de trabajo nos permitió tener un control cercano de la evolución del proyecto sin necesidad de haber especificado con precisión todos los requisitos a comienzos de este.

Recordando que el objetivo principal era conseguir planificar en videojuegos (más concretamente, traducir automáticamente descripciones VGDL en dominios de planificación que permitan resolver juegos) se destacaron los siguientes subobjetivos principales:

OBJ-1 Estudio y análisis del lenguaje PDDL.

OBJ-2 Extensión de la base de conocimiento para el lenguaje PDDL.

OBJ-3 Optimización del conocimiento y de los dominios.

OBJ-4 Integración con la arquitectura de planificación y ejecución.

OBJ-5 Extracción automática de objetivos.

OBJ-6 Validación del conocimiento integrado.

OBJ-7 Experimentación con múltiples planificadores.

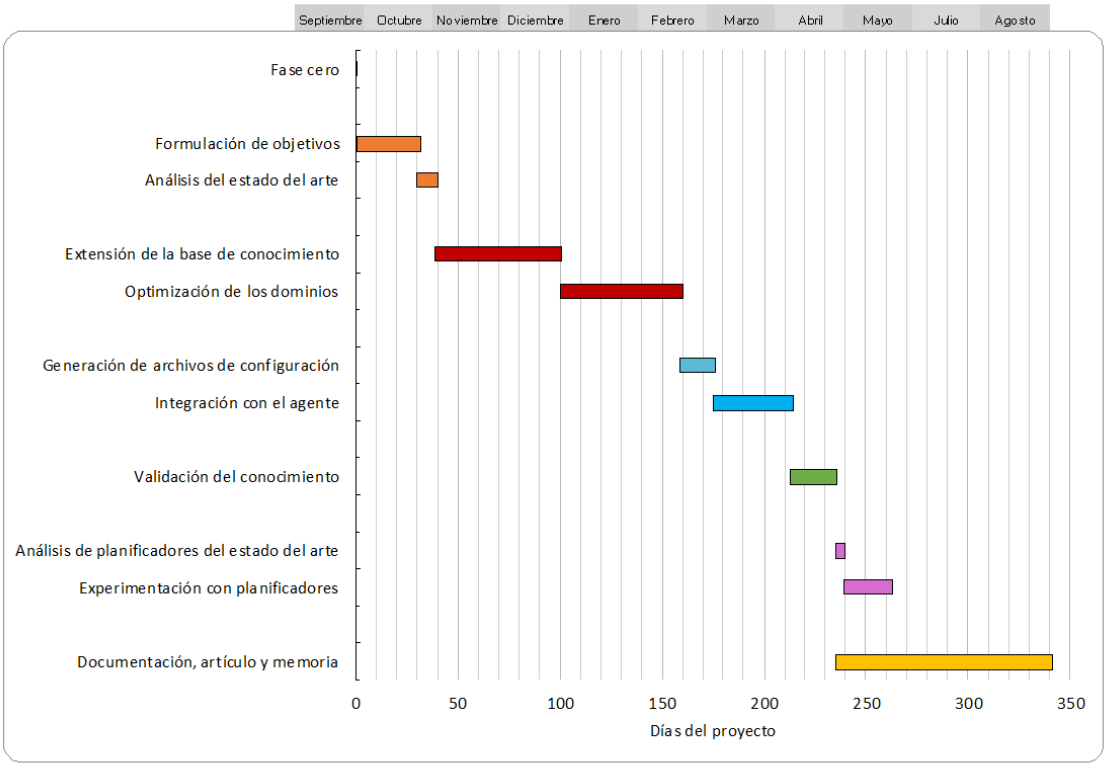


Figura 3.1: Diagrama de Gantt del proyecto.

3.2. Temporización

El plan de trabajo siguió la siguiente temporización, resumida en los diagramas 3.1 y 3.2. La duración del proyecto, contando las fases previas a este TFM, lleva más de 2 años.

3.2.1. Fase cero

Se procede a resumir el estado del proyecto previo al comienzo de este TFM.

Durante el primer año de desarrollo nos dedicamos a implementar un compilador de VGDL hacia HPDL, una variante jerárquica de la planificación clásica. Con una metodología altamente similar[1] a la mostrada en los siguientes apartados, obtuvimos una arquitectura funcional pero ausente de estrategia inteligente, de forma que se necesitaba aportar conocimiento adicional tras el proceso de traducción de cara a poder actuar en los juegos. Adicionalmente, se realizó un

[1] La idea era (y sigue siendo) demostrar que la metodología, pese a ser usada en un ámbito de videojuegos, puede extenderse a otros lenguajes.

TAREA	FECHA DE INICIO	FECHA DE TERMINACIÓN	DÍA DE COMIENZO	DÍAS DE DURACIÓN
Fase cero				
Fase cero	5/7	23/10	0	1
Primera fase				
Formulación de objetivos	23/9	23/10	1	31
Análisis del estado del arte	23/10	1/11	30	10
Segunda fase				
Extensión de la base de conocimiento	1/11	1/1	39	62
Optimización de los dominios	1/1	1/3	100	60
Tercera fase				
Generación de archivos de configuración	1/3	17/3	159	17
Integración con el agente	17/3	24/4	175	39
Cuarta fase				
Validación del conocimiento	24/4	16/5	213	23
Quinta fase				
Análisis de planificadores del estado del arte	16/5	20/5	235	5
Experimentación con planificadores	20/5	12/6	239	24
Documentación				
Documentación, artículo y memoria	16/5	29/8	235	106

Figura 3.2: Tareas y fechas del proyecto.

esquema de replanificación sencillo para poder emplear los dominios en sistemas no deterministas.

De forma paralela al desarrollo también se escribió un artículo que fue aceptado y presentado en el workshop KEPS de la conferencia ICAPS 2020 [Vellido et al., 2020]. La retroalimentación más destacada que obtuvimos de la conferencia era que la ausencia de estrategia era un impedimento demasiado significativo para emplear el sistema, por lo que decidimos cambiar de lenguaje de planificación objetivo.

3.2.2. Primera fase

Aunque la idea de este proyecto ya estaba planteada, consideramos que la primera fase comienza tras la exposición del artículo, en octubre de 2020.

Durante estas semanas se hizo un análisis del estado del arte ahora desde el punto de vista de la planificación clásica, y se discutieron las diferentes formas de afrontar el nuevo problema. Temas tales como la integración con GVGAI y las posibles técnicas para reducir las restricciones de memoria se plantearon aquí.

3.2.3. Segunda fase

La segunda fase consistió en la extensión de la base del conocimiento para este nuevo lenguaje. A lo largo de estos meses realizamos diferentes pruebas de concepto sobre los dominios. Se generaron casos de estudio en diferentes versiones de PDDL (desde PDDL1.2 hasta PDDL+) y los evaluamos con diferentes planificadores buscando la representación más eficiente de los dominios.

Sobre estas pruebas, destacamos que aunque las características de PDDL+ eran prometedoras (eventos y funciones continuas podían servirnos para modelar en "segundo plano" el manejo del turno y las interacciones) tuvo que ser descartado por no poder adaptarse completamente a las características de un juego VGDL.

Por otro lado, PDDL2.1 y superiores ofrecían soporte a funciones numéricas que sabíamos que simplificaban enormemente la complejidad de los dominios (frente a una aproximación basada en conectividades de celdas). Desafortunadamente, su uso implicaría reducir en gran medida el número de planificadores que podríamos evaluar, puesto que aunque esta versión es la más estandarizada en el campo de la planificación automática, muchos de los planificadores clásicos no se habían adaptado a ella.

No por ello la funcionalidad de los numéricos fue descartada. Tal y como mostramos en la subsección 5.3.3, esta se puede simular perfectamente en la versión original de PDDL.

3.2.4. Tercera fase

Una vez finalizado el proceso de traducción a PDDL integramos la metodología con el agente de replanificación. Para ello, primeramente fue necesario hacer un estudio de su funcionamiento y de las características necesarias para su uso.

Posteriormente, se implementó el proceso de generación automática de archivos de configuración, de los que dependía el agente. También se ampliaron sus capacidades para adaptarse de mejor manera a los dominios optimizados.

Por último, puesto que el agente dependía de un planificador ubicado en la nube, algo que nos resultaba tedioso en la práctica puesto que nos devolvía frecuentemente timeouts en las peticiones al servidor web, se realizó una integración en local del sistema capaz y fácilmente extensible a múltiples planificadores.

3.2.5. Cuarta fase

Como cuarta fase se procedió a validar los dominios generados y a refinar su comportamiento, corrigiendo errores dónde era necesario.

Esta validación se realizó tanto de manera visual (con pruebas unitarias) como mediante la ejecución del agente en múltiples niveles, verificando que podía jugar y ganar en los juegos.

3.2.6. Quinta fase

Una vez validados los dominios se procedió a experimentar con múltiples planificadores del estado del arte para ver si estos se podían proponer como testbed en competiciones futuras. Con este objetivo en mente, se realizó una análisis de múltiples planificadores de diferente índole (con tipos de heurística, búsqueda, etc) y se procedió a evaluarlos en diferentes pruebas y métricas.

Esta tarea llevó más tiempo del que estimábamos, puesto que los planificadores se desarrollaron hace años y frecuentemente colisionaban con las versiones más modernas de software. También se debía a que aunque que restringimos los tiempos máximos de ejecución a 15 minutos (la última competición IPC establecía el límite en 30) el tiempo de computación empleado fue enorme. Llegamos a estimar que la experimentación al completo empleó más de 200 horas de procesamiento.

Al mismo tiempo que se realizaba la experimentación se escribió la documentación y memoria del proyecto. Adicionalmente, se redactó un artículo que ha sido enviado a la revista *The Knowledge Engineering Review*, aún a falta de confirmar su aceptación. Se puede acceder a la preprublicación en el enlace http://arxiv.org/abs/2109.00449.

Capítulo 4

Metodología

En este proyecto hemos generado dominios PDDL a partir de descripciones de videojuegos en el lenguaje VGDL. Los dominios describen en términos de tipos, predicados, acciones, etc. los objetos y las interacciones en el juego de manera precisa, de forma que, tras integrarse en un agente de planificación, es capaz de actuar y resolver niveles de juegos.

Tal y como se muestra en la figura 4.1, el sistema basado en el conocimiento implementado recibe como entrada los archivos GDF y LDF describiendo, respectivamente, un juego y un nivel en el lenguaje VGDL. Como resultado, se obtiene un dominio PDDL y un archivo de configuración que permite integrar el sistema con una arquitectura la cuál, hasta que el nivel es resuelto, continuamente planifica y actúa.

Los dominios producidos son deterministas, ya que modelar incertidumbre añade una complejidad adicional a unos dominios ya duros de por sí. A medida que en los juegos aumenta el número de elementos e interacciones posibles los dominios crecen en dimensión, resultándole más difícil al planificador buscar planes en esos entornos. Añadir control sobre el no determinismo aumenta esta posibilidad, y puede llegar a ser contraproducente. Para paliar esta carencia, consideramos una mejor alternativa incluir un módulo reactivo que controle las posibles inconsistencias entre el estado del planificador y el del juego, replanificando si es necesario.

La metodología se divide un cuatro pasos, descritos brevemente a continuación y en detalle en las siguientes subsecciones.

1. **Extracción de elementos de juego**: Mediante una gramática definida en ANTLR se parsea el GDF y se extraen una serie de elementos del juego (*game elements*), representando los conceptos de interés para la generación de los archivos PDDL.

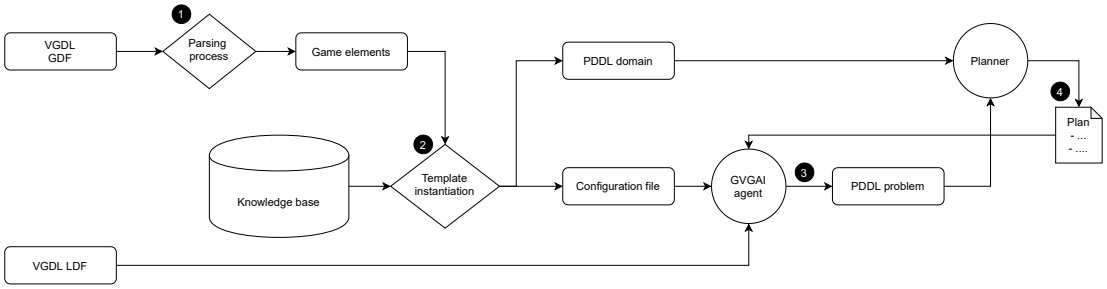


Figura 4.1: Flujo del sistema de compilación.

2. **Generación del dominio**: Estas elementos entran en un módulo responsable de producir el dominio de salida, dónde se almacena la base de conocimiento. Este módulo realiza un proceso de instanciación de plantillas PDDL abstractas obteniendo diferente contenido para el dominio (tipos, predicados y acciones).

3. **Generación del problema**: Junto al dominio la metodología genera un archivo de configuración que aporta el conocimiento necesario para que la arquitectura del agente sea capaz de generar problemas PDDL a partir del LDF cuando sea necesario.

4. **Planificación y actuación**: Una vez obtenidos el dominio y el problema PDDL se llama a un planificador para obtener un plan que resuelva el nivel. De cara a afrontar situaciones no determinístas, un sistema de monitorización vigila la ejecución del plan y replanifica cuando sea necesario.

4.1. Extracción de elementos de juego

Inicialmente el proceso emplea la gramática definida en ANTLR para parsear el GDF, extrayendo una serie de *elementos de juego* (*game elements*). Cada elemento codifica una pieza individual del GDF, independiente del resto de componentes del juego, y será apropiadamente transformada en elementos del dominio PDDL. Los tipos de elementos posibles son:

- **Sprite**: Objeto presente en el juego y sus características.
- **Interaction**: Acción y objetos involucrados.
- **LevelMap**: Carácter y el objeto que representa en un LDF.
- **Termination**: Criterio de terminación y sus características.

El proceso de parseo permite identificar las distintas secciones del archivo GDF, cada sección corresponde con un conjunto de elementos de juego como se muestra a continuación:

- **SpriteSet**: Por cada objeto de este conjunto se genera un elemento de juego de tipo *avatar* o *sprite*, según corresponda o no a un personaje jugable, de la forma (n, t, f, att), donde n es el nombre del elemento; t es su tipo (*e.g.* VerticalAvatar, Flicker); f es su padre en la jerarquía de objetos; y att es una lista de atributos[1]. Estos atributos servirán para especificar las propiedades de los objetos en PDDL.

- **InteractionSet**: Para cada interacción definida en esta sección se genera un elemento de juego en forma de tupla (t, p, r, att), siendo t es el tipo de la interacción (*e.g.* killSprite, flipDirection); p y r los nombres de los objetos en el rol de emisor y receptor, respectivamente; y att los posibles atributos de la interacción.

- **LevelMapping**: Para almacenar las correspondencias entre los caracteres de un LDF y los objetos del GDF se generan elementos de juego de la forma (c, n), donde c es el carácter y n el nombre del objeto tal y como se ha definido en la sección SpriteSet.

  Ya que el agente es el encargado de generar el problema PDDL este conocimiento se pasará mediante el archivo de configuración.

- **TerminationSet**: Por último, respecto a los criterios de parada se forman elementos de forma (t, v, att), siendo t el tipo de criterio (*e.g.* SpriteCounter, Timeout); v un indicativo de si la condición corresponde a una victoria o una derrota; y att la lista de parámetros que pueden acompañar al criterio, como los objetos involucrados o el número de turnos necesario para que haga efecto.

Estos elementos de juego son independientes del lenguaje objetivo hacia el que se traduce. En una versión previa de esta metodología utilizamos el lenguaje de planificación no jerárquico HPDL [Vellido and Olivares, 2020], empleando exactamente el mismo proceso de extracción de elementos de juego. Por tanto es posible ampliar fácilmente la base de conocimiento para otros lenguajes siguiendo los mismos pasos descritos a continuación.

[1] Únicamente se almacenan atributos de carácter funcional (como velocidad o dirección de movimiento), mientras que los centrados en aspecto de visualización son ignorados.

Elemento de juego		Base de conocimiento	
Nombre:	<T>	Acciones:	move_<O>(...) move_stop(...)
Tipo:	Missile (sprite)		
Parámetros:	Orientation: <O>	Predicados:	(missile_<T>_moved ?object) (turn_<T>_move) (finished_turn_<T>_move)
Tipo:	killSprite (interacción)		
Emisor:	<E>	Acción:	<E>_<R>_killSprite(...)
Receptor:	<O>		

Tabla 4.1: Información almacenada en la base de conocimiento para dos elementos de juego: un sprite de tipo *missile* y una interacción de tipo *killSprite*. Cuando se descubran los valores de las variables tras el proceso de parseo las plantillas se instancian y se incluyen en el dominio PDDL. Sobre las acciones se muestra únicamente su cabecera.

4.2. Generación del dominio

Una vez obtenidos los elementos de juego comienza un proceso de compilación de VGDL a PDDL dando como salida un dominio y un archivo de configuración. Este último archivo sirve para pasarle al agente el conocimiento necesario para generar los problemas PDDL y para monitorizar la ejecución del plan. Su composición está escrita con mayor detalle en las secciones 4.3 y 5.3.4.

El dominio se construye mediante la instanciación de plantillas PDDL almacenadas en una base de conocimiento. En base a los diferentes tipos de los elementos de juego, este módulo genera y combina las partes del dominio PDDL (tipos, predicados y acciones).

4.2.1. Base de conocimiento

La base de conocimiento es una colección de plantillas que facilita la traducción de VGDL a PDDL. Cada una de las plantillas se define como una abstracción de código PDDL que permite instanciar los elementos de juego descubiertos en los pasos anteriores. La selección de cada plantilla se realiza en base a los atributos que acompañan a cada una de los elementos.

Para alcanzar la mayor generalización posible cada una de las plantillas se centra en un único concepto de VGDL, independientemente del juego en el que se defina y del resto de objetos que lo formen.

```
(:action <S1>_<S2>_COLLECTRESOURCE
  :parameters(?o1 - <S1> ?o2 - <S2>
              ?x ?y ?r ?r_next - num)

  :precondition (and
      (turn-interactions)

      ; Check objects are different
      (not (= ?o1 ?o2))
      (at ?x ?y ?o1)
      (at ?x ?y ?o2)

      (got-resource-<S1> ?r)
      (next ?r ?r_next)
  )

  :effect (and
      ; Remove resource from map
      (not (at ?x ?y ?o1))
      (object-dead ?o1)

      ; Increase value
      (not (got-resource-<S1> ?r))
      (got-resource-<S1> ?r_next)
  )
)
```

```
(:action SHOES_USER_COLLECTRESOURCE
  :parameters(?o1 - shoes ?o2 - user
              ?x ?y ?r ?r_next - num)

  :precondition (and
    (turn-interactions)

    ; Check objects are different
    (not (= ?o1 ?o2))
    (at ?x ?y ?o1)
    (at ?x ?y ?o2)

    (got-resource-shoes ?r)
    (next ?r ?r_next)
  )

  :effect (and
    ; Remove resource from map
    (not (at ?x ?y ?o1))
    (object-dead ?o1)

    ; Increase value
    (not (got-resource-shoes ?r))
    (got-resource-shoes ?r_next)
  )
)
```

Tabla 4.2: Ejemplo del template de una acción para una interacción (izquierda) y su instanciación (derecha), representando cómo el agente adquiere un recurso al ubicarse en la misma casilla. Cuando se descubren los elementos de juego S1 y S2 se rellena la plantilla con los valores oportunos.

La base de conocimiento se puede agrupar en tres secciones:

- **Plantillas para sprites**: Enfocadas en representar los objetos del juego (a excepción del avatar). Cada sprite acaba transformándose como mínimo en un objeto PDDL y en una serie de predicados, pudiendo expandirse con acciones si son necesarias para modelar su comportamiento. Existen predicados genéricos para todos los tipos, como los que se utilizan para representar la posición de cada objeto, y otros específicos para el tipo concreto del sprite.

  Un ejemplo se muestra en la tabla 4.1. Un objeto de tipo *missile* hará que se generen predicados y acciones que actualicen su posición en cada turno del juego, moviéndose únicamente en la dirección indicada en sus parámetros.

- **Plantillas para avatares**: Los avatares son un subtipo de *sprites* que representan los agentes en el juego. Las acciones disponibles para cada avatar varían con su tipo, y en la mayor parte están limitadas a juegos cuadriculados, incluyendo comúnmente movimientos en las cuatro direcciones cardinales y la posibilidad de usar un *sprite* definido previamente (por ejemplo, una espada con la que atacar enemigos).

La base de conocimiento almacena predicados y templates para cada posible acción de un avatar y conoce cuáles están disponibles para cada tipo.

Es importante indicar que para controlar la posición del avatar en el turno anterior siempre se utilizan predicados de orientación. En caso de ser un avatar no orientado (puede moverse en cualquier dirección sin perder un turno en girar) el propio dominio actualiza la orientación tras cada movimiento.

- **Plantillas para interacciones**: Esta última sección almacena templates para las interacciones VGDL (*i.e.* cambios en el mundo tras la colisión de dos objetos), cada una asociada con una o varias acciones PDDL.

  En la tabla 4.2 se muestra un ejemplo de una plantilla para una primitiva, junto al resultado de su instanciación. Después del proceso de parseo donde se averigua los tipos de los objetos involucrados en la interacción, las variables *S1* y *S2* se sustituyen y se incluye la acción en el dominio PDDL.

El proceso de creación de una plantilla consiste en abstraer el conocimiento del concepto a modelar de forma independiente al resto de elementos de juego. Aunque para un solo dominio repetir este proceso para cada elemento del lenguaje puede no merecer la pena, cuando se van a generar múltiples dominios el esfuerzo se reduce considerablemente. Puesto que cada plantilla es genérica (independiente del juego en el que se utilice), una vez se ha definido y validado se puede reutilizar tantas veces como se quiera sin necesidad de que un ingeniero del conocimiento intervenga.

4.2.2. Construyendo los dominios

Como se ha indicado previamente, las plantillas devueltas por la base de conocimiento se instancian con la ayuda de los elementos de juego, obteniendo un dominio como combinación de todas las instanciaciones. Cada sección del dominio PDDL se forma de diferente manera:

- **Types**: Para cada sprite hijo se refleja la jerarquía definiendo el padre como el tipo del objeto. Cuando se alcanza un sprite sin padre, el tipo asociado es aquel incluido en la descripción VGDL. Todos los tipos de sprites acabarán heredando del objeto más genérico, denominado *Object*, que nos permite generalizar comportamientos en predicados y acciones.

  Para reducir la complejidad de los dominios (en el número de interacciones e instancias posibles), los objetos estáticos en vez de incluirse como instancias en el problema se modelan mediante predicados *(is-<T> ?x ?y)*, dónde *<T>* representa el tipo del objeto, *?x* la coordenada en el eje horizontal e *?y* la coordenada en el eje vertical. Aunque este método no es una representación fiel de VGDL no perjudica las capacidades de generalización de la metodología. Por el contrario, este conocimiento adicional optimiza

```
(:action BOULDER_MOVE_DOWN
  :parameters (?o - boulder
               ?x ?y ?new_y - num)
  :precondition (and
    (turn-boulder-move)
    (not (boulder-moved ?o))
    (oriented-down ?o)

    (at ?x ?y ?o)
    (next ?y ?new_y)

    ; Check cell is empty
    (not (is-wall ?x ?new_y))
  )
  :effect (and
    (not (at ?x ?y ?o))
    (at ?x ?new_y ?o)

    (boulder-moved ?o)
  )
)

...
```

```
(:action STOP_BOULDER_MOVE
  :parameters ()
  :precondition (and
    (turn-boulder-move)

*   (forall (?o - boulder)
*     (or (object-dead ?o)
*         (boulder-moved ?o))
*   )
  )
  :effect (and
    (forall (?o - boulder)
      (not (boulder-moved ?o))
    )

    (not (turn-boulder-move))
    (finished-turn-boulder-move)
  )
)
```

Tabla 4.3: Primitivas que modelan el comportamiento de objetos tipo *Missile*. La acción de la izquierda se llama repetidamente moviendo cada instancia de forma acorde. Cuando todos los objetos se han actualizado, la predicados señalados en la acción de la derecha se vuelven ciertos y esta permite continuar con la ejecución del turno.

los dominios en gran medida[2], permitiendo obtener planes en niveles más grandes y complejos.

- **Predicates**: Debemos mantener lo siguientes tipos de predicados:
  - Para saber la localización de cada instancia en el mapa.
  - Para conocer la orientación de cada objeto capaz de moverse (avatares y sprites no-estáticos).
  - Predicados específicos para recursos, como monedas, gemas, o botas.
  - Para representar el orden del turno en el dominio (detallado en la sección 4.2.3).
  - Finalmente, puesto que PDDL 1.2 no permite definir valores numéricos para recursos o coordenadas, los simulamos con un nuevo tipo *num* y predicados de orden (*e.g. (next ?n1 ?n2 - num)*, siendo *?n2* el entero sucesor de *?n1*). Más detalles sobre su implementacion se especifican en la sección 5.3.3.

[2] Alcanzamos una reducción hasta del 75% en el número de instancias mediante esta optimización.

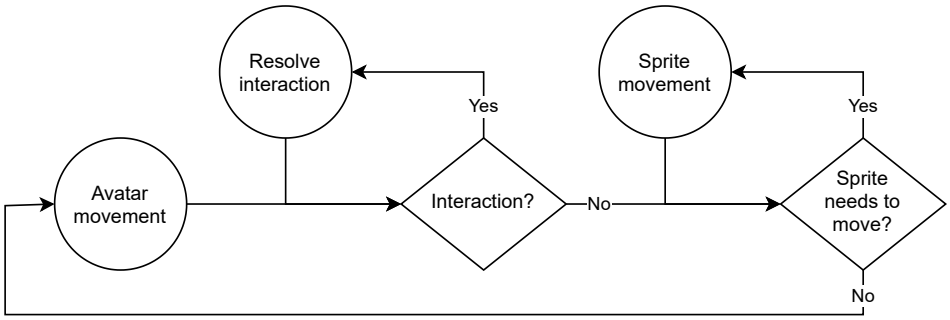


Figura 4.2: Ciclo ordenado de acciones que siguen los dominios. Los círculos indican primitivas que modifican el estado del mundo y los rombos primitivas de control.

- **Actions**: La generación de acciones se estructura de la siguiente forma
  - Por cada posible movimiento del avatar se incluye una nueva acción, añadiendo una adicional para el movimiento nulo (no hacer nada). Esta última se incluye en caso de que el agente deba esperar a la finalización de un evento antes de realizar su movimiento.
  - Para los sprites no-estáticos añadimos acciones que actualicen su estado en cada turno, tal y como se ejemplificó en la tabla 4.3.
  - Por cada interacción se incluye una primitiva que reproduzca sus efectos. Las precondiciones comprobarán que ambas instancias se encuentran en la misma casilla y, según el tipo de interacción, en algunos casos será necesario incluir comprobaciones adicionales.

    En la tabla 4.2 vimos un ejemplo de una primitiva asociada a una interacción, llamada *CollectResource*. Esta interacción involucra a un sprite de tipo *shoes* y a uno de tipo *user*, haciendo que user incremente el número de recursos de shoes y este desaparezca del mapa.
  - Finalmente, se añaden acciones que fuerzan el orden entre las acciones, de forma que el dominio sea coherente con el flujo de control de GVGAI. Estas primitivas se detallan en la sección 4.2.3.

4.2.3. Modelando el turno del juego

Cualquier videojuego sigue un flujo de procesamiento interno que organiza la ejecución de cada uno de los elementos que lo componen. Por tanto, nuestros dominios, cuya intención final es funcionar en el entorno GVGAI, también deben ajustarse a este flujo. Aunque PDDL, a diferencia de un modelo HTN, carece de sistemas explícitos para modelar el orden entre acciones, es posible simularlo añadiendo predicados y acciones adicionales.

Los pasos de un turno en nuestros dominios son los siguientes, tal y como se muestran en la figura 4.2 y se ejemplifican en la tabla 4.4:

1. Primeramente, se elige un movimiento para el agente. Será responsabilidad del planificador y de su heurística decidir, entre aquellas acciones disponibles ese turno, la más orientada en la resolución del nivel.

2. Seguidamente, todas las interacciones se resuelven antes de proseguir con la ejecución del turno. Esto implica que al acabar este paso no habrá un par de objetos con una interacción definida en el GDF que estén actualmente colisionando en el nivel.

3. De la misma manera, otro conjunto de primitivas de control aseguran que aquellos sprites (objetos) que deberían realizar un movimiento lo hagan antes de terminar el turno. Por ejemplo, las precondiciones señaladas con un asterisco (*) en la acción de la derecha en la tabla 4.3 solo serán ciertas una vez que cada instancia de tipo *missile* se haya actualizado.

A diferencia de la versión de HPDL que previamente implementamos en este proyecto se altera el orden entre la ejecución de los movimientos de aquellos objetos diferentes al avatar y la comprobación de interacciones. Esto se debe a que, puesto que un plan PDDL termina cuando el estado indicado en los objetivos del problema es alcanzado, el planificador puede terminar la ejecución sin resolver interacciones que invalidan el estado final.

4.2.4. Traducción de objetivos

Los juegos en VGDL están dirigidos por objetivos (al contrario que orientados por recompensa), por lo que se adaptan bien a la representación de objetivos de la planificación clásica. Aunque existen varios tipos de *TerminationCriteria*, la mayoría son variaciones del llamado *SpriteCounter*. Tal y como mostramos a la izquierda en la tabla 4.5, estos criterios vienen acompañados de un sprite y un contador (generalmente a cero), con el significado de: *Cuándo el número de instancias del tipo T alcance el valor X, finalizar*.

Aunque parezca que usar un solo tipo de criterio es limitado, combinando adecuadamente la gran cantidad de interacciones con las que cuenta VGDL no reducimos la expresividad en esta parte. Por ejemplo, en la tabla 4.5 se comparan objetivos con su representación en VGDL.

Para nuestra representación en PDDL, cuando el parser se encuentre con alguno de estos criterios indica (a través del archivo de configuración) que el *goal* debe revisar que no haya ninguna instancia de ese tipo presente en el nivel.

```
Sokoban: Empujando una caja en un hoyo

+ AVATAR_ACTION_MOVE_DOWN         (avatar n3 n4 n5)
- BOX_AVATAR_BOUNCEFORWARD_DOWN   (box_3_4 avatar n3 n5 n6)
- BOX_HOLE_KILLSPRITE             (box_3_4 hole_3_6 n3 n6)
- END-TURN-INTERACTIONS           ()
# END-TURN-SPRITES                ()
-----------------------------------------------------------------
IceAndFire: Cogiendo un recurso

+ AVATAR_ACTION_MOVE_RIGHT        (avatar n6 n1 n7)
- ICESHOES_AVATAR_COLLECTRESOURCE (iceshoes_7_1 avatar n13 n1 n0 n1)
- END-TURN-INTERACTIONS           ()
# END-TURN-SPRITES                ()
-----------------------------------------------------------------
Boulderdash: Cavando bajo una roca

+ AVATAR_ACTION_USE_RIGHT         (avatar n11 n6 n12)
- DIRT_SWORD_KILLSPRITE           (dirt17 sword1 n12 n6)
- END-TURN-INTERACTIONS           ()
# SWORD_DISAPPEAR                 (sword1 n12 n6)
# STOP_SWORD_DISAPPEAR            ()
# BOULDER_MOVE_DOWN               (boulder3 n12 n5 n6)
# STOP_BOULDER_MOVE               ()
# END-TURN-SPRITES                ()
```

Tabla 4.4: Planes de salida para un turno en varios juegos. (+) indica movimientos del avatar, (-) interacciones y (#) acciones de movimiento para otros sprites.

Sokoban: Poner cada caja en un hoyo	`InteractionSet`
	`  box hole > killSprite`
	
	`TerminationSet`
	`  SpriteCounter stype=box limit=0`
	`  win=True`
		`(:goal`
		`  (forall`
		`  (?o - box)`
		`  (dead ?o)`
		`  )`
		`)`

Boulderdash: Coger 9 gemas e ir a la salida	`InteractionSet`
	`  exit avatar > killIfOtherHasMore`
	`  resource=gem limit=9`
	
	`TerminationSet`
	`  SpriteCounter stype=exit limit=0`
	`  win=True`
		`(:goal`
		`  (forall`
		`  (?o - exit)`
		`  (dead ?o)`
		`  )`
		`)`

Waves: Sobrevivir 50 turnos	`TerminationSet`
	`  Timeout limit=50 win=True`
		`(:goal`
		`  (and`
		`  (turn n50)`
		`  (not`
		`  (dead avatar)`
		`  )`
		`  )`
		`)`

Tabla 4.5: Criterios de parada en VGDL y su traducción a PDDL.

Pese a que técnicamente existe otro tipo de criterio de parada, pocos juegos lo utilizan. Este criterio, llamado *Timeout*, se activa cuando se alcanza el número de turnos indicado en sus parámetros. Su codificación en PDDL se puede incluir fácilmente con un nuevo predicado que cuente cuántos turnos han pasado desde el comienzo de la ejecución.

4.3. Generación del problema

Puesto que buscamos un proceso continuo de replanificación hacemos que el agente se encargue de generar el problema PDDL. Para ello, el compilador VGDL produce un archivo adicional, que denominamos **archivo de configuración**, donde se indican las relaciones entre la lógica del framework GVGAI y la de planificación. De esta forma el agente es capaz de mantener un control de sus recursos y definir los problemas PDDL siguiendo el esquema del dominio.

Versiones previas del agente suponían que los archivos de configuración se generaban a mano, pero gracias a los elementos de juego y a la introducción de conocimiento adicional hemos visto que el parser es capaz de rellenar esta plantilla de manera automática.

No por ello sin cambios. Para adaptar la lógica de los dominios producidos automáticamente, se introducen nuevos campos y se modifican algunos existentes. En la sección 5.3.4 se indican los detalles completos de su implementación.

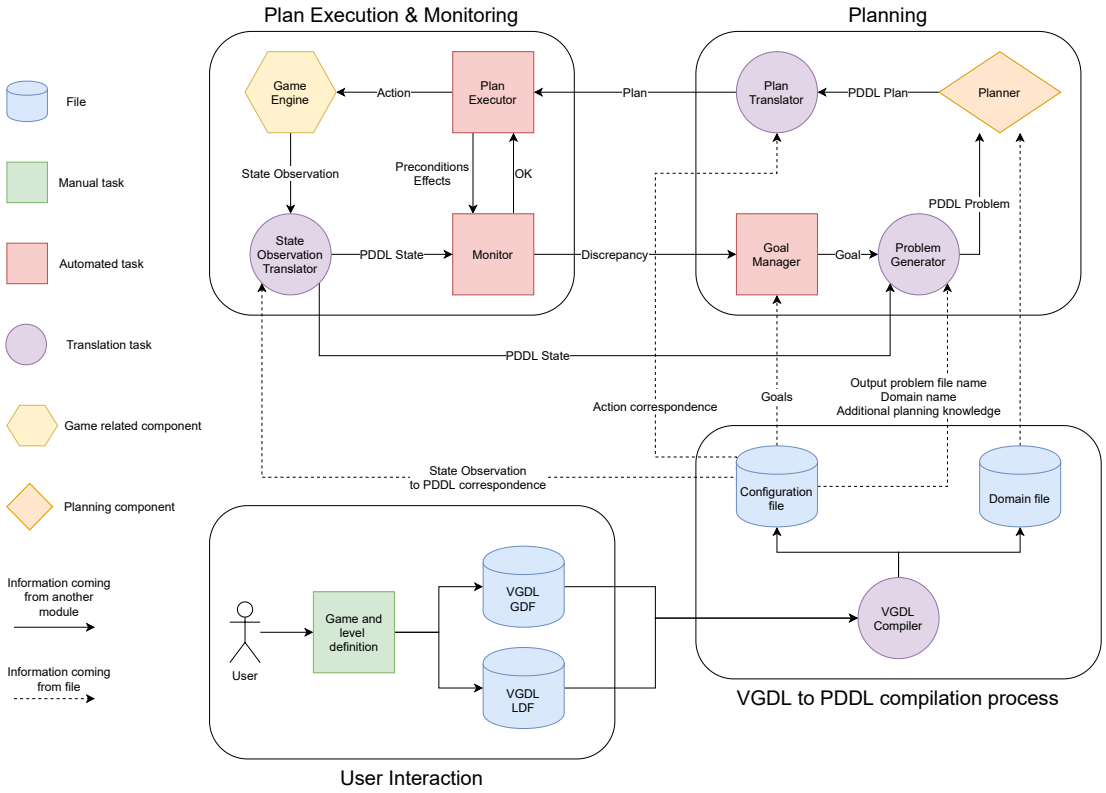


Figura 4.3: Arquitectura final tras la integración del compilador con el sistema de planificación y actuación.

4.4. Planificación y actuación

La metodología detallada anteriormente muestra como generar dominios y problemas automáticamente a partir de descripciones de videojuegos. Estos archivos pueden usarse de por sí como workbench para planificadores debido a su naturaleza desafiante (tal y como se explica en el capítulo 6), pero de cara a utilizarlos para guiar el comportamiento de un agente deliberativo en los juegos se integra con una arquitectura de planificación y actuación.

El resultado final de esta integración se muestra en la figura 4.3, junto al flujo de información entre sus diferentes componentes. La arquitectura de planificación se compone de dos módulos, detallados en las siguientes secciones.

4.4.1. Módulo de planificación

El módulo de planificación cuenta con tres sistemas: un Traductor de Planes (*Plan Translator*); un Generador de Problemas *(Problem Generator)*; y un Gestor de Objetivos (*Goal Manager*).

El gestor de objetivos lee los objetivos indicados en el archivo de configuración y se encarga de determinar cuál es el objetivo actual y cuáles han sido o no previamente alcanzados. Este objetivo actual se envía al generador del problema junto al resto de conocimiento indicado en el archivo de configuración necesario para su definición. Tal y como se ha indicado previamente, las pautas que sigue para generarlo se indican en la sección 5.3.4.

Una vez obtenido el problema y junto al dominio recibido anteriormente, el sistema llama a un planificador a la espera de un plan que ejecutar. Versiones previas de la arquitectura empleaban un planificador ubicado en la nube, pero debido a los problemas de comunicación (recibíamos frecuentes timeouts en la comunicación con el servidor) decidimos adaptar el módulo para permitir emplear un planificador local mediante un nuevo traductor de planes.

Este traductor es el único requisito a cambiar de cara a utilizar cualquier otro planificador basado en PDDL en el sistema. Este se encarga de extraer del plan las acciones GVGAI que realizará el agente ayudándose de las correspondencias de acciones indicadas en el archivo de configuración.

4.4.2. Módulo de monitorización

Debido a las características no determinísticas de algunos de los juegos el plan devuelto por el planificador puede fallar. Puede haber casos en el que una roca haya caído en un sitio inesperado o un grupo de enemigos hago el camino actual demasiado peligroso para atravesarlo.

Para este tipo de casos el sistema cuenta con un módulo de monitorización que comprueba antes de indicarle a GVGAI la acción a realizar que las precondiciones del movimiento del agente siguen siendo consistentes con el estado actual del juego. Si en algún momento estas dejan de serlo, se descarta el plan actual y se genera un nuevo problema PDDL, usando el estado del juego (*state observation*) como LDF. Con el nuevo problema y el dominio original se vuelve a llamar al planificador en busca de un nuevo plan para el agente, y se continua con el ciclo de ejecución. Puesto que las veracidad de las precondiciones ha cambiado podemos estar seguros que el mismo movimiento no se volverá a seleccionar de nuevo.

Capítulo 5

Implementación

El código fuente junto a toda la información sobre instalación y uso del software implementado se puede encontrar en la página [Vellido and Nikolov,].

5.1. Gramática ANTLR

La gramática, mostrada en la figura 5.1, representa la sintaxis y los diferentes conceptos que se incluyen dentro de la descripción de un juego en VGDL. Definida en el formato de ANTLR, permite producir la estructura básica del listener, que la recorrerá extrayendo los tokens necesarios para el proceso de compilación.

Existen ciertos puntos de relevancia para su comprensión:

- INDENT y DEDENT indican respectivamente el aumento y el decremento de indentación. De cara a controlar la jerarquía de sprites es necesario incluir código adicional para saber el nivel de indentación en todo momento.
- Las etiquetas de las reglas se definen con el carácter #, que facilita la forma de referenciarlas en el listener.
- Mediante el carácter = se permite asignar nombres a conjuntos de tokens dentro de las parser rules.
- El comando (− > skip) indica que se ignora el token encontrado.
- WS indica cualquier tipo de espacio en blanco. Por la forma de tratar las indentaciones, se prioriza la detección de estas ante el resto de espacios.
- El orden en el que se definen las reglas es relevante. ANTLR permite escribir gramáticas ambiguas, priorizando en ese caso la primera regla definida.
- En spriteDefinition, si la línea no contiene un segundo token WORD, nos indica que es un sprite hoja.

```
... // (Comandos para manejar la indentación en el SpriteSet)

// ----------------- Parser rules -----------------
basicGame
: 'BasicGame' parameter* nlindent
    (spriteSet | levelMapping | interactionSet | terminationSet
     | DEDENT | nlindent | NL )* DEDENT
;

// ----------------------------
spriteSet
: 'SpriteSet' nlindent (NL* spriteDefinition NL*)+ DEDENT
;

spriteDefinition
: name=WORD WS* '>' WS* spriteType=WORD?
    parameter* nlindent (WS* spriteDefinition NL?)+ DEDENT
    #RecursiveSprite
| name=WORD WS* '>' WS* spriteType=WORD? parameter*
    #NonRecursiveSprite
;

// ----------------------------
levelMapping
: 'LevelMapping' nlindent (NL* LEVELDEFINITION NL*)+ DEDENT
;

// ----------------------------
interactionSet
: 'InteractionSet' nlindent (NL* interaction NL*)+ DEDENT
;

interaction
: sprite1=WORD WORD+ WS* '>' interactionType=WORD parameter*
;

// ----------------------------
terminationSet
: 'TerminationSet' nlindent (NL* terminationCriteria NL*)+ DEDENT
;

terminationCriteria
: WORD parameter* 'win='(TRUE | FALSE) parameter*
;

// ----------------------------
parameter
: left=WORD '=' (WORD | TRUE | FALSE ) #NonPathParameter
| left=WORD '=' (WORD '/' WORD)  #PathParameter
;

nlindent    // Nueva línea seguida de indentación
: NL INDENT
;
```

Figura 5.1

```
// ----------------- Lexer rules -----------------

COMMENT // Se ignoran los comentarios
: '#' ~[\r\n]* -> skip ;

NL  // Nueva línea
| ( '\r'? '\n' | '\r' | '\f' ) SPACES?
)

WS  // Se ignoran los espacios en blanco
: [ \t]+ -> skip ;

TRUE
: [Tt] [Rr] [Uu] [Ee]
;
FALSE
: [Ff] [Aa] [Ll] [Ss] [Ee]
;

WORD
: CHAR (CHAR | UNDERSCORE | NUMBER)*
| NUMBER
;

LEVELDEFINITION
: WS* SYMBOL WS* '>' WS* (WORD WS*)+
;

ANY     // Se ignora todo aquello no reconocido
: .
;

// ----------------- Fragments -----------------

fragment SPACES
: [ \t]+ ;
fragment SYMBOL
: ~[ ] ;
fragment CHAR
: [a-zA-Z/.] ;
fragment UNDERSCORE
: '_' ;
fragment NUMBER
: '-'? [0-9]+ ([.,] [0-9]+)? ;  // Cualquier número real
```

Figura 5.1: Gramática ANTLR resumida.

Puesto que la descripción de un nivel en VGDL sigue una conceptualización bastante sencilla, es posible parsearla manualmente sin necesidad de definir ninguna gramática.

5.2. Listener

El listener consta de una clase responsable de recorrer el árbol generado a partir del archivo de descripción de juego y extraer la información de relevancia para el proceso de compilación.

Gracias a la gramática ANTLR, este listener se genera automáticamente con métodos que se activan cuando se entra y se sale de cada nodo del árbol. Estos nodos van asociados a cada parser rule definida en la gramática, y junto a las etiquetas y la asociación de tokens resulta fácil de acceder a las distintas partes de cada regla.

Además de los elementos de juego, detallados más adelante, se extrae la siguiente información:

- **ShortTypes**: Lista de los caracteres que pueden aparecer en un archivo de nivel (LDF).
- **LongTypes**: Tipos de los objetos que pueden aparecer en un LDF.
- **Stypes**: Tipos definidos en el SpriteSet. Este conjunto es mayor que el de LongTypes, pues incluye a los padres y a los tipos de estos.
- **Hierarchy**: Diccionario con los hijos asociados a cada padre.
- **TransformTo**: Objetos que pueden transformase en otros. Son los afectados por la interacción *transformTo*.
- **StepBacks**, **killIfOther**: Objetos involucrados en una interacción stepBack, killIfOtherHasMore o killIfOtherHasLess. Para reducir el tamaño de los dominios y los efectos del grounding, estas interacciones no se definen como acciones, sino que se integran dentro de las acciones de movimiento.

Para sacar alguno de estos elementos el listener analiza el nivel del juego asociado al problema.

5.2.1. Elementos de juego

Como se muestra en la figura 5.2, existen cuatro clases que almacenan la información de cada línea del archivo de descripción de juego. Mayoritariamente de la clases *Sprite* e *Interaction* se sacan los elementos de juego, elementos que serán instanciados en los templates.

Estas clases no implementan funcionalidad por sí misma, sino que encapsulan la información de cara a facilitar el acceso al resto de clases.

```
Sprite
|-- Name            : Nombre del objeto
|-- Type            : Tipo del objeto
|-- Parent          : En caso de que tenga, padre
|__ Parameters      : Parámetros del objeto

LevelMap
|-- Char            : Carácter que representa al objeto
|__ Sprites         : Tipo del objeto representado

Interaction
|-- SpriteName      : Nombre del objeto que produce la interacción
|-- PartnerName     : Objeto que recibe los efectos
|-- Type            : Tipo de interacción
|__ Parameters      : Parámetros de la interacción

Termination
|-- Type            : Criterio de parada
|-- Win             : Si el agente gana o pierde la partida
|__ Parameters      : Parámetros del criterio de parada
```

Figura 5.2: Clases de los elementos de juego y sus atributos.

5.2.2. Base de conocimiento

La base de conocimiento se almacena como un conjunto de clases estructuradas en tres módulos (como se muestra en la figura 5.3). Estos módulos a su vez encapsulan diferentes clases, las cuales reciben los elementos de juego, y cada módulo devuelve templates instanciados de una parte del dominio/problema.

Los módulos se dividen en las clases descritas a continuación.

5.2.2.1. AvatarPDDL

Recibe como entrada la instancia de la clase Sprite (desde los elementos de juego) que corresponde al avatar; la jerarquía; y en caso de poder usar algún objeto, la instancia Sprite correspondiente. Para cada elemento devuelve información diferente:

- **Actions**: Según el tipo del avatar construye e instancia los templates de los movimientos disponibles. Corresponde a primitivas del dominio PDDL.
- **Predicates**: Devuelve predicados en base al tipo del avatar.

```
AvatarPDDL
|-- AvatarActions
|__ AvatarPredicates

SpritePDDL
|-- SpriteActions
|-- SpritePredicates
|__ SpriteLevelpredicates

InteractionPDDL
|__ InteractionActions
```

Figura 5.3: Clases que forman la base de conocimiento relacionadas con VGDL.

5.2.2.2. SpritePDDL

Se genera una instancia con cada tipo diferente de objeto distinto de avatar.

Recibe como entrada la instancia del objeto Sprite; la jerarquía; y el padre en caso de que tenga. Para cada elemento devuelve información diferente:

- **Actions**: Construye y devuelve primitivas en base al tipo de objeto.
- **Predicates**: Genera los predicados necesarios para modelar el orden del turno en las acciones de control, al igual que aquellos asociados a recursos.
- **LevelPredicates**: Predicados para el problema específicos del objeto. En la versión actual del software solo concierne a la orientación de los tipos de misiles, que serán añadidos en el archivo de configuración.

5.2.2.3. InteractionPDDL

Recibe como entrada la instancia de Interaction; las dos instancias Sprite de los objetos involucrados; y la jerarquía. Para cada elemento devuelve una tupla **(interacción, sprite, sprite)** con las tres partes que forman una acción en PDDL, no solo por cada interacción sino por cada pareja de objetos asociada a ella.

Los tres módulos anteriores hacen uso de una clases auxiliares que encapsulan las diferentes partes de una acción PDDL. Esta se almacena en un archivo llamado *typesPDDL* y permite acceder con facilidad a las distintas partes que forman las primitivas.

El uso de un único archivo para una clase tan simple es debido a que mantenemos la misma metodología que con HPDL. En sus caso, *typesHPDL* se forma por la estructura de tareas del lenguaje (task, methods y actions).

5.3. Generación de la salida

La salida producida por nuestra metodología se corresponde con dos archivos, el dominio PDDL y el archivo de configuración utilizado en la arquitectura del agente. En las siguientes subsecciones se detallan las clases y conceptos de interés para generar esta salida.

5.3.1. Domain generator

El domain generator es una clase que, a partir de las plantillas y a las estructuras obtenidas a partir del listener, termina de construir cada parte del dominio. Concretamente se encarga de:

- Definir la sección **:types** reflejando la estructura jerárquica del juego.
- Incluir los predicados recibidos por la base de conocimiento.
- Construir los predicados genéricos de orientación y posición. Añadir predicados necesarios para la evaluación de interacciones, control de orden, y aquellos específicos de ciertos objetos.
- Crear las acciones del avatar, del resto de objetos, y las primitivas para resolver interacciones, junta a las acciones que reflejan el orden del turno.

Como salida devuelve, en forma de cadena de caracteres, cada una de las diferentes estructuras que forman el dominio PDDL (predicates, actions, etc.).

5.3.2. Writers

Por último, existe un script encargado de escribir la salida del generator en pantalla/disco. Este script construye completamente el dominio de salida, en base a las diferentes partes devueltas por el generator.

Principalmente, se encargan de darle el formato necesario del lenguaje PDDL (paréntesis, requisitos, etc.).

5.3.3. Representación de funciones numéricas

Una función numérica es un tipo de predicado que, en vez de almacenar un valor booleano (es o no cierto) codifica un número flotante. Aunque PDDL2.1 integra de manera adecuada el uso de funciones, permitiendo hacer operaciones sobre estas como suma o resta, no todos los planificadores soportan esta versión.

Pese a que el número de planificadores no adaptados a PDDL2.1 no es muy grande consideramos que el esfuerzo necesario para su implementación en PDDL1.2 no era excesivo, y las ventajas de que los dominios utilicen la versión más antigua nos resultaría provechosa[1].

[1] A diferencia de algunos tipos de desarrollo de software, las nuevas versiones de PDDL se construyen sobre las anteriores, añadiendo y no reduciendo la expresividad de estas.

```
(define (domain VGDLGame)
    (:types
        num
    )
    (:predicates
        ; Numerics
        (next ?x ?y - num)
        (previous ?x ?y - num)
    )
)
```

```
(define (problem VGDLGameProblem)
    (:domain VGDLGame)
    (:objects
        n0 n1 n2 n3 n4 n5 - num
    )
    (:init
        (next n0 n1)
        (next n1 n2)
        (next n2 n3)
        (previous n1 n0)
        (previous n2 n1)
        (previous n3 n2)
    )
)
```

Tabla 5.1: Implementación de valores numéricos tanto en un dominio como en un problema PDDL.

Para la implementación fue necesario incluir un nuevo tipo de objeto, **num**, y dos tipos de predicados para controlar el orden de los numéricos. Respecto al problema PDDL, siempre se deben añadir estas instancias a cada nivel con los predicados correctamente inicializados. En la tabla 5.1 tenemos un ejemplo de su implementación.

5.3.4. Config generator

El config generator actúa de manera similar al domain generator, recibiendo información tanto del listener como de la base de conocimiento. La diferencia reside en que este define, en formato YAML, el archivo de configuración necesario para generar problemas PDDL y monitorizar la ejecución del agente. Este tipo de archivo contiene los siguientes campos, ejemplificados en la figura 5.4:

- **domainFile**: Ruta donde encontrar el dominio PDDL.
- **problemFile**: Ruta donde almacenar el problema PDDL.
- **domainName**: Nombre del dominio.
- **gameElementsCorrespondence**: Indica qué predicados incluir cuando se detecta una instancia del tipo GVGAI especificado.
- **variablesTypes**: En este campo se representa la correspondencia entre las variables del campo anterior y los tipos PDDL del dominio. Cada instancia se modelará como un único objeto PDDL en el problema.
- **avatarVariable**: Variable que representa al avatar.
- **orientation**: Tipo de orientación a asignar a cada objeto. Para el agente obligatoriamente siempre se define una, aunque en caso de no ser relevante se indica NONE. Para el resto de objetos la orientación es opcional.

```
domainFile: domains/zelda.pddl
problemFile: problem.pddl
domainName: VGDLGame
gameElementsCorrespondence:
  avatar:
  - (at ?x ?y ?avatar)
  goal:
  - (at ?x ?y ?goal)
  key:
  - (at ?x ?y ?key)
  sword:
  - (at ?x ?y ?sword)
  enemy:
  - (is-enemy ?x ?y)
  monsterQuick:
  - (at ?x ?y ?enemy)
  wall:
  - (is-wall ?x ?y)
variablesTypes:
  ?goal: goal
  ?key: key
  ?sword: sword
  ?avatar: avatar
  ?enemy: enemy
  ?x: num
  ?y: num
avatarVariable: ?avatar
orientation:
  avatar: FIND
orientationCorrespondence:
  UP: (oriented-up ?object)
  DOWN: (oriented-down ?object)
  LEFT: (oriented-left ?object)
  RIGHT: (oriented-right ?object)
fluentsPredicates:
  next: (next ?n0 ?n1)
  previous: (previous ?n1 ?n0)
actionsCorrespondence:
  AVATAR_ACTION_MOVE_UP: ACTION_UP
  AVATAR_ACTION_TURN_UP: ACTION_UP
  AVATAR_ACTION_USE_UP: ACTION_USE
  ...
```

```
additionalPredicates:
- (turn-avatar)
- (got-resource-key n0)
addDeadObjects:
  sword: 1
goals:
- goalPredicate: (forall (?o - goal) (object-dead ?o))
  priority: 1
  saveGoal: no
```

Figura 5.4: Ejemplo resumido del archivo de configuración para Zelda.

- **orientationCorrespondence**: Predicados que modelan cada una de las orientaciones.
- **fluentsPredicates**: Predicados usados para codificar los objetos numéricos, tal y como se explicaron en la sección anterior.
- **actionsCorrespondence**: Correspondencia entre las acciones definidas en el dominio y los movimientos del agente en GVGAI. Cuando el agente encuentra una de estas acciones en el plan devuelto por el planificador ejecuta la acción indicada en la correspondencia.
- **additionalPredicates**: Este campo añade los predicados indicados al problema. Mayoritariamente utilizado para establecer el comienzo del turno y los recursos iniciales del agente.
- **addDeadObjects**: En algunos casos es necesario que el nivel cuente con instancias de objetos aunque estos no estén inicialmente presentes en él. Principalmente se corresponde con objetos que surgen espontáneamente debido a una interacción a al movimiento del avatar (*e.g.* espadas, picos). Este campo, recibiendo nombre del objeto y un número X, añade X instancias del objeto al problema con el predicado *(object-dead ...)*.
- **goals**: Esta sección se forma por la lista de objetivos que el agente debe resolver. Cada uno de ellos cuenta con tres parámetros: (1) los predicados que se pretenden alcanzar; (2) el nivel prioridad; y (3) si se desea guardar o no el objetivo. Si un objetivo se guarda el estado alcanzado en *goalPredicate* se incluye en los siguientes problemas generados. Esto resulta útil, por ejemplo, si dividimos en subobjetivos y el hecho de conocer cuáles de ellos se han alcanzado es relevante para la obtención del resto.

  Si en el archivo de configuración se indica más de un objetivo el agente define un problema PDDL para cada uno de ellos, y repetirá el proceso de planificación y actuación hasta completarlos todos.

5.3.5. Acciones de control de orden

```
(:action END-TURN-INTERACTIONS
    :parameters ()
    :precondition (and
        (turn-interactions)

        (not (exists (?o1 - goal ?o2 - avatar ?x ?y - num)
            (and
                (not (= ?o1 ?o2))
                (at ?x ?y ?o1)
                (at ?x ?y ?o2)
            )
        )

        (not (exists (?o1 - box ?x ?y - num)
            (and
                (is-hole ?x ?y)
                (at ?x ?y ?o1)
            )
        )
        ...
    )
    :effect (and
        (turn-sprites)
        (not (turn-interactions))
    )
)
```

Figura 5.5: Ejemplo de acción de control para verificar que no hay interacciones pendientes por resolver.

La ausencia de mecanismos en PDDL que nos permitan forzar el orden en la ejecución de algunas acciones nos obliga a introducir primitivas y predicados adicionales que regulan el flujo del turno en el juego.

Es necesario por tanto dos tipos de primitivas: (i) para comprobar que no haya interacciones pendientes; y (ii) para asegurarnos que todas las instancias de objetos no estáticos han realizado su turno.

Para (i) la acción de control comprueba cada par de instancias para asegurarse que no colisionan. Ejemplificada en la figura 5.5, esta primitiva se genera de forma óptima para que compruebe el menor número de pares posibles, asegurándose que solo los tipos que pueden interaccionar se incluyen en la comprobación.

Para (ii), tal y como se muestra en la figura 5.6, bloquea la ejecución del turno hasta que cada una de las instancias haya realizado su movimiento.

Como predicados de control se añade uno para cada fase del juego y, en caso de haber objetos en movimiento, un par de predicados para cada tipo. Podemos ver un ejemplo en la figura 5.7.

```
(:action STOP_BOULDER_MOVE
    :parameters ()
    :precondition (and
        (turn-boulder-move)
        (forall (?o - boulder)
           (or (object-dead ?o)
               (boulder-moved ?o)
           )
        )
    )
    :effect (and
        (forall (?o - boulder)
           (not (boulder-moved ?o))
        )
        (not (turn-boulder-move))
        (finished-turn-boulder-move)
    )
)
```

Figura 5.6: Acción para asegurar la caída de rocas en Boulderdash.

```
(:predicates
    ; Game turn order
    (turn-avatar)
    (turn-sprites)
    (turn-interactions)

    (turn-sword-disappear)
    (finished-turn-sword-disappear)
)
```

Figura 5.7: Predicados para controlar el turno en Zelda.

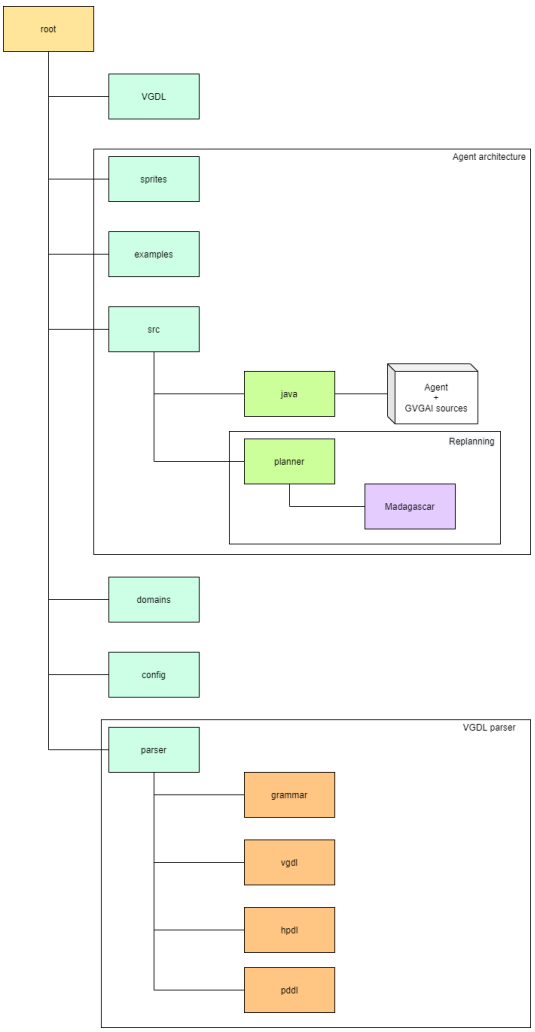


Figura 5.8: Estructura de directorios del sistema completo.

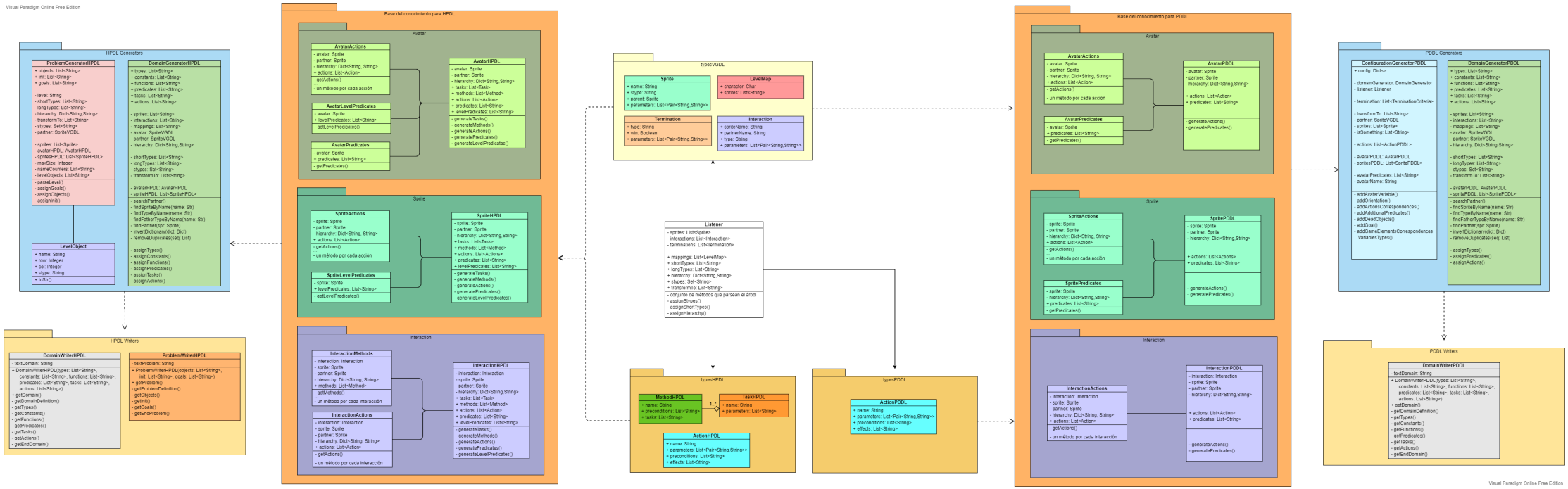


Figura 5.9: Diagrama de clases.

5.4. Estructuración

La estructura de directorios del compilador se diseñó de forma que fuera fácilmente extensible a nuevos lenguajes. Como vemos en la figura 5.8, para integrar PDDL en la arquitectura solo hace falta definir una nueva carpeta bajo el directorio *parser* con los archivos *avatar/sprite/interaction-PDDL.py, *GeneratorPDDL.py y *WriterPDDL.py*.

El sistema al completo, con el agente, se forma con la siguiente jerarquía de directorios, mostrada en la figura 5.8:

- **VGDL**: Archivos fuente de VGDL (GDF y LDF) para cada juego.
- **config**: Carpeta donde almacenar los archivos de configuración generados por el compilador.
- **domains**: Directorio donde almacenar los dominios PDDL generados por el compilador.
- **parser**: Archivos fuentes del traductor, separados en base al lenguaje que se asocia. Se cuenta con la gramática ANTLR y el listener; clases asociadas a la producción de información HPDL y PDDL; y clases con conocimiento exclusivo de VGDL.
- **src**: Scripts que forman el framework GVGAI, con el agente definido en su interior. Existe una carpeta adicional, *planner*, con los archivos necesarios para interpretar y ejecutar la salida del planificador.
- **examples**, **sprites**: Recursos que GVGAI necesita para ejecutarse.

En la figura 5.9 se muestran el conjunto de módulos y las correspondientes clases que componen el proyecto, siguiendo los detalles de implementación explicados en los apartados anteriores.

5.5. Ejecución

El formato de ejecución del compilador queda de la siguiente manera:

```
$ main.py [-h] -l {pddl,hpdl}
               -gi GAMEINPUT
               [-li LEVELINPUT]
               [-go GAMEOUTPUT]
               [-lo LEVELOUTPUT]
               [-vh]

Argumentos opcionales:
 -h,
   --help                       Mensaje de ayuda
 -l {pddl,hpdl},
   --language {pddl,hpdl}       Lenguaje de salida
 -gi GAMEINPUT,
   --gameInput   GAMEINPUT      Archivo de juego de entrada
 -li LEVELINPUT,
   --levelInput  LEVELINPUT     Archivo de nivel de entrada
 -go GAMEOUTPUT,
   --gameOutput  GAMEOUTPUT     Archivo de juego de salida
 -lo LEVELOUTPUT,
   --levelOutput LEVELOUTPUT    Archivo de nivel de salida
 -vh,
   --verboseHelp                Muestra información adicional
```

Aunque se proporciona un archivo Makefile para facilitar el uso el software.

La ejecución del agente, más compleja, se detalla en el repositorio del proyecto [Vellido and Nikolov,].

Capítulo 6

Experimentación

Para la experimentación hemos utilizado diez juegos diferentes, siete de puzles y tres reactivos, con diferentes niveles de determinismo, no determinismo y características tales como las que se muestran en la tabla 6.1.

Esta experimentación es multiobjetivo. Por un lado pretendemos validar la metodología generando varios dominios PDDL a partir de descripciones VGDL de videojuegos. Además, para los dominios resultantes analizamos las características sintácticas y semánticas de los mismos y verificamos mediante pruebas unitarias que las plantillas definidas son correctas.

Por otro lado, estamos interesados en evaluar el rendimiento de varios planificadores del estado del arte con los dominios generados para comprobar la complejidad computacional y la viabilidad (cómo de duros son) de los problemas de actuación en dominios de videojuegos.

En la figura 6.2 mostramos una captura de los juegos en la interfaz gráfica de GVGAI. Estos son:

- **Bait**: Juego en el que el jugador debe obtener una llave para abrir la puerta de salida. Existen hoyos que dificultan el paso, pero pueden ser cubierto empujando cajas sobre ellos.

- **Boulderdash**: El objetivo de este juego es recoger nueve gemas en el mapa y moverse hacia la salida. Existen multitud de enemigos y rocas para dificultar el movimiento del agente en su camino.

  Hay dos tipos de terreno: excavado y vacío. Las rocas se desplazan hacia abajo cuando bajo ellas no se encuentra ningún objeto "rígido" (es decir, un agente, un NPC o terreno excavado). Para abrirse paso, el personaje tiene a su disposición una pala que le permite cavar el terreno, pasando este de excavado a vacío. En ocasiones el uso de esta pala es forzosamente necesario para resolver los niveles, aunque su uso se debe hacer con precaución pues puede hacer caer rocas en posiciones no deseadas.

	Elementos sintácticos				Elementos semánticos		
	Types	Supertypes	Predicates	Actions	Determinismo	Acciones del agente	Interacciones
Bait	7	4	15	15	Yes	4	8
Boulderdash	11	8	21	23	No	12	8
Butterflies	4	3	13	8	No	4	1
CakyBaky	11	4	22	17	No	8	6
ChipsChallenge	19	4	27	26	Yes	4	19
IceAndFire	8	4	18	10	Yes	4	3
SimplifiedBoulder.	7	6	17	20	Yes	12	5
Sokoban	4	3	13	12	Yes	4	5
Zelda	9	7	17	20	No	12	5
Zenpuzzle	5	2	15	8	Yes	4	1

Figura 6.1: Características sintácticas y semánticas de los dominios.

Boulderdash se caracteriza por ser fuertemente no-determinista, multi-objetivo, y en ocasiones irresoluble.

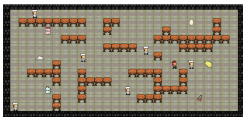
Como alternativa simplificada también se crea una nueva versión, llamada **SimplifiedBoulderdash**, con cinco gemas como objetivo, sin enemigos ni gravedad en las rocas, y donde estas se pueden romper con la pala.

- **Butterflies**: Un juego extremadamente simple. El agente debe colisionar con todas las mariposas del mapa, que se mueven de forma aleatoria y se multiplican cuando se sitúan sobre un capullo.
- **Cakybaky**: Juego reactivo en el que el avatar debe coger seis ingredientes en un orden concreto, mientras al mismo tiempo evita una serie de enemigos que le persiguen por el mapa.
- **ChipsChallenge**: Juego complejo que mezcla ideas de IceAndFire y Sokoban. Con un gran número de objetos e interacciones diferentes, el objetivo es alcanzar la salida oculta tras puertas de múltiples colores. El agente debe ir alcanzado cada una de las llaves que abren cada puerta sin morir, teniendo a su disposición botas y cajas que le ayudan a abrirse camino.
- **IceAndFire**: Un juego laberíntico donde el jugador debe alcanzar la salida. Existen casillas de hielo y fuego que dificultan el objetivo, pero que pueden ser atravesadas si previamente se cogen unas botas de resistencia a cada elemento.
- **Sokoban**: Juego de puzles en el cuál el agente debe empujar todas las cajas en casillas específicas del mapa, sobre las que caen y desaparecen. El avatar no puede agarrar ninguna de las cajas, debe empujarlas en la misma dirección en la que se mueve.
- **Zelda**: El objetivo reside en conseguir una llave para escapar por la puerta, esquivando una serie de enemigos de movimiento aleatorio, pero que pueden ser eliminados usando una espada.

(a) Bait.

(b) Boulderdash.

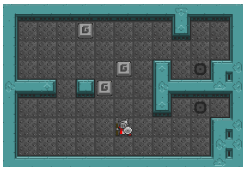
(c) Butterflies.



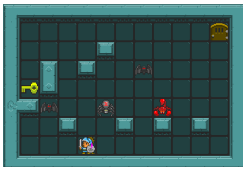
(d) Cakybaky.

(e) ChipsChallenge.

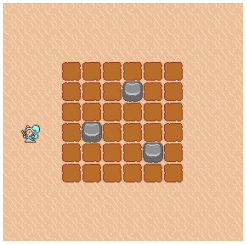
(f) IceAndFire.



(g) Sokoban.



(h) Zelda.



(i) Zenpuzzle.

Figura 6.2: Imágenes de 9 juegos con la interfaz gráfica de GVGAI.

```
1.  (avatar_action_move_right avatar n1 n9 n2)
2.  (diamond_avatar_collectresource diamond_2_9 avatar n2 n9 n0 n1)
3.  (end-turn-interactions )
4.  (turn-sprites )
5.  (stop_sword_disappear )
6.  (stop_boulder_move )
7.  (end-turn-sprites )
8.  (avatar_action_turn_left avatar)
9.  (end-turn-interactions )
10. (turn-sprites )
11. (boulder_move_down boulder_9_10 n9 n10 n11)
12. (stop_sword_disappear )
13. (stop_boulder_move )
14. (end-turn-sprites )
```

Figura 6.3: Porción de un plan devuelto por el planificador, representando dos turnos en *Boulderdash*. Este se ha dividido en los segmentos correspondientes a cada subdivisión de un turno, de forma que las primitivas en verde corresponden al movimiento del avatar, las naranjas a las del resto de objetos y las rojas las interacciones. En azul se muestran las acciones de control de orden.

- **Zenpuzzle**: Engloba una serie de puzles basados en caminos. El avatar debe encontrar la ruta que pasa por cada casilla una única vez. Este juego resulta interesante ya que la heurística del planificador cobra mayor importancia a la hora de encontrar la solución de forma rápida.

6.1. Validación del conocimiento

La validación de un dominio de planificación es una tarea de validación del conocimiento, pero mientras que un sistema basado en el conocimiento clásico suele enfocarse para clasificación o diagnóstico, en planificación se pretende construir planes. Por tanto, el conocimiento de nuestro modelo representa acciones y cómo estas afectan a los objetos del mundo [McCluskey, 2002].

Para asegurarnos que los modelos son correctos y se adaptan a los requisitos del sistema no podemos seguir un formalismo exacto, ya que los requisitos en nuestro sistema son los archivos VGDL y el significados semánticos de las acciones e interacciones de los juegos, conceptos que no pueden ser expresados matemáticamente.

El mecanismo por el que optamos consecuentemente es una validación por inspección mediante pruebas unitarias, metodología muy extendida en la validación de software que consiste en evaluar pequeños trozos de código individuales cuyo funcionamiento esperado se sabe de antemano.

Adaptando este concepto al ámbito de la planificación, realizamos múltiples casos de prueba (test unitarios) sobre cada plantilla en la base de conocimiento, formados por un dominio con la plantilla a verificar (*e.g.* una acción modelando una interacción, el movimiento de un misil, de un avatar, etc.) y el problema más pequeño que nos permita llevar a cabo la prueba. El proceso de validación consistió en verificar que en cada momento el estado de planificación era igual al estado del juego.

Adicionalmente, gracias a la interfaz gráfica de GVGAI realizamos una segunda validación (en este caso puramente visual) del comportamiento del agente, ejecutándolo sobre múltiples niveles y comprobando que las dinámicas de los dominios estaban fielmente representadas y se resolvían los objetivos sin errores.

En el vídeo `https://youtu.be/F7kg6LC-sDI` se muestra mediante la interfaz de GVGAI el comportamiento del agente en varios juegos, tres de puzzles y uno reactivo.

6.2. Benchmarks

El objetivo principal de nuestra experimentación es proponer, siguiendo el formato de la **I**nternational **P**lanning **C**ompetition de 2018 [ipc, 2018], múltiples benchmarks de nuestros dominios, mostrando el comportamiento de planificadores del estado del arte bajo condiciones complejas.

Todas las pruebas se realizaron en un Intel i7-8750H funcionando a 2.20Ghz y con 6GB de memoria RAM, estableciendo un límite de tiempo en 15 minutos[1].

La elección de planificadores se basó en los siguientes motivos:

1. **Facilidad de integración**. Creemos que habría sido interesante añadir muchos más planificadores a los presentados en este proyecto, sobre todo aquellos basados en grounding parcial como IPALAMA o en lifted planning como L-RPG. Desafortunadamente, tras múltiples esfuerzos resultó imposible compilarlos en nuestro ordenador, y la total ausencia de documentación y tiempo no permitió que los adaptáramos a las nuevas versiones.
2. **Variedad en las técnicas**. Se ha intentado que la gama de planificadores sea la más variada posible (con diferentes tipos de heurísticas, de algoritmos de búsqueda, estando o no basados en SAT...).

Con estos criterios, los planificadores seleccionados fueron los siguientes:

- **Madagascar** [Rintanen, 2011], que utiliza técnicas basadas en SAT.
- **Fast Downward** [Helmert, 2006], con búsqueda basada en A* y en LAMA LAMA [Richter and Westphal, 2010] es un sistema multi-heurístico que emplea un algoritmo A* con pesos decrecientes.

[1] En el anexo A se muestran los detalles completos de cada ejecución.

- **Saarplan** [Fickert et al., 2018], basado en *decoupled search*, un algoritmo que intenta reducir el espacio de búsqueda particionando las variables del espacio formando una topología de estrella y buscando únicamente sobre el centro de la topología.
- **LAPKT-DUAL-BFWS** [Frances et al.,], combinando búsqueda primero en anchura con poda.

Los distintos criterios sobre los que se han evaluado los planificadores son:

- **Coverage**: Indica la cantidad de instancias resueltas por el planificador, siendo una buena medida para valorar la versatilidad del planificador en múltiples juegos. La puntuación se calcula como la suma para cada nivel resuelto dentro de los límites de memoria y tiempo, siendo uno si lo resuelve y cero en otro caso.
- **Satisficing**: Mide la capacidad de resolver cada nivel de la forma más óptima, es decir, con el menor número de pasos necesarios (plan más corto). Cada nivel se calcula mediante la fórmula:

$$S = \frac{C}{C*} \tag{6.1}$$

  donde C es la distancia del plan encontrado y C* la del plan óptimo. En nuestro caso, como el no determinismo de los juegos hace que no podamos calcular matemáticamente el plan óptimo cogemos el menor encontrado por cualquiera de los planificadores. Una vez más, la puntuación final de un planificador es la suma de la métrica en cada instancia.
- **Agile**: Valora la rapidez de cada planificador en la búsqueda de planes. Su métrica se define de la siguiente forma, siendo T el tiempo (en segundos) en obtener el plan:

$$A = \begin{cases} 1 & T \leq 1 \\ 1 - \frac{log(T)}{log(900)} & 1 < T \leq 900 \\ 0 & T > 900 \end{cases} \tag{6.2}$$

  La elección del umbral de 900 corresponde al número máximo de segundos del que disponen los planificadores para resolver el problema. La puntuación final de un planificador se calcula como la suma del agile score para cada nivel.

Cada planificador se evaluó sobre cada criterio con 10 niveles de cada juego, cinco predefinidos en GVGAI y otros cinco adicionales que definimos nosotros, variando en tamaño del mapa y en el número de instancias.

Las tablas 6.1, 6.2, 6.3 muestran resultados de los planificadores con cada criterio, y en las tablas 6.5 y 6.6 se indica los tiempos y desviaciones medios para cada juego.

A partir de ellos apreciamos que los resultados de coverage son extremadamente pobres en la mayoría de casos. Aunque los dominios y problemas PDDL están optimizados para reducir el número de instancias y acciones, solo siete juegos se resuelven por más de un planificador.

Los problemas aún así son resolubles, pero encontrar un plan de calidad es duro. DUAL-BFWS consigue resolver el 99% de los niveles, pero la mayoría no son planes de calidad.

Esto es mayormente debido al proceso de grounding[2], explotando en tiempo y memoria. DUAL-BFWS aparenta no verse afectado por la búsqueda en anchura inicial que realiza, obteniendo un plan provisional al comienzo de la ejecución. Pese a ello, estos planes no son óptimos, y en la mayoría de juegos donde un planificador basado en Fast Downward resiste el proceso de grounding estos son capaces de encontrar un plan mejor a mayor velocidad.

Indicamos también que entre los planificadores que hemos seleccionado aquellos que participaron en la IPC 2018 (*i.e.* DUAL-BFWS, Saarplan y FD-LAMA) alcanzaron resultados bastante diferentes frente a los de nuestro testbed.

En la IPC los tres conseguían posiciones contiguas y en lo alto de la tabla, excediendo DUAL-BFWS en satisficing, LAMA en agile y Saarplan en coverage. Mientras que en nuestros resultados, si suprimimos los juegos afectados por el grounding, DUAL-BFWS es el mejor planificador en coverage, LAMA en satisficing y Saarplan en agile, tal y como se muestra en la tabla 6.4

Este es otro motivo más para considerar nuestros dominios como un buen testbed, ya que se evalúan otro tipo de características y da otra perspectiva diferente del comportamiento de los planificadores.

Es importante destacar que un controlador estándar en las competiciones de GVGAI cuenta con 50ms para decidir una acción para el agente. Comparándolo con los 15min a los que no son capaces de ceñirse los planificadores (como hemos indicado, no a causa de una búsqueda larga, sino de un proceso de adecuación del problema al algoritmo de búsqueda demasiado lento), se ve que estos no se pueden utilizar en condiciones donde se combina un número alto de instancias presentes en el mundo y una necesidad de respuesta rápida.

No quiere decir que la planificación no se pueda utilizar en problemas con restricciones temporales. Únicamente indica que el dominio debe estar mejor definido y centrado en el problema.

[2] Transformación del problema desde una representación *lifted* a un equivalente proposicional, de manera que se acelere la búsqueda del plan.

Por tanto, puesto que podemos identificar claramente el problema (el proceso de grounding) concluímos que estos dominios se pueden considerar como un buen testbed para evaluar planificadores. Los resultados indican que el comportamiento de planificadores en problemas con muchas instancias debería ser mejorado, ya que la mejora en velocidad que proporciona el grounding no justifica el impacto negativo que genera en coverage.

Coverage	bait	boulderdash	butterflies	cakybaky	chipschallenge	iceandfire	simplified-bould.	sokoban	zelda	zenpuzzle	SUM
DUAL-BFWS	**10**	**10**	**10**	**10**	**9**	**10**	**10**	**10**	**10**	**10**	99
Saarplan	0	0	10	10	0	10	0	0	10	1	41
FD (A*)	0	0	10	10	0	10	0	0	10	1	41
FD (LAMA)	0	0	8	10	0	10	0	0	10	0	38
Madagascar	1	0	8	0	0	8	0	1	6	8	32

Tabla 6.1: Resultados del Coverage score

Sat score	bait	boulderdash	butterflies	cakybaky	chipschallenge	iceandfire	simplified-bould.	sokoban	zelda	zenpuzzle	SUM
DUAL-BFWS	**10.00**	**10.00**	7.09	8.65	**9.00**	**10.00**	**10.00**	**10.00**	8.34	**9.18**	92.3
FD (A*)	0.00	0.00	**10.00**	**9.99**	0.00	10.00	0.00	0.00	**9.96**	1.00	41.0
FD (LAMA)	0.00	0.00	8.00	9.99	0.00	10.00	0.00	0.00	9.96	0.00	38.0
Saarplan	0.00	0.00	7.84	8.08	0.00	9.83	0.00	0.00	9.59	0.94	36.3
Madagascar	0.32	0.00	7.16	0.00	0.00	7.36	0.00	0.76	5.58	7.68	28.9

Tabla 6.2: Resultados del Satisficing score

Agile score	bait	boulderdash	butterflies	cakybaky	chipschallenge	iceandfire	simplified-bould.	sokoban	zelda	zenpuzzle	SUM
DUAL-BFWS	**5.97**	**5.75**	8.17	5.59	**5.94**	**10.00**	**6.62**	**5.43**	9.44	**10.00**	72.9
Saarplan	0.00	0.00	**9.88**	7.99	0.00	10.00	0.00	0.00	**10.00**	0.95	38.8
FD (A*)	0.00	0.00	7.25	**9.88**	0.00	10.00	0.00	0.00	10.00	0.78	37.9
FD (LAMA)	0.00	0.00	4.46	4.47	0.00	9.71	0.00	0.00	9.75	0.00	28.4
Madagascar	0.68	0.00	1.62	0.00	0.00	1.66	0.00	0.97	0.54	3.44	8.9

Tabla 6.3: Resultados del Agile score

Metric	DUAL-BFWS	Saarplan	FD (LAMA)	FD (A*)	Madagascar
Coverage score	**40**	40	38	40	22
Satisficing score	34.12	35.4	38.0	**40.0**	20.1
Agile score	33.19	**37.9**	28.4	37.1	3.8

Tabla 6.4: Resultados totales de las distintas métricas quitando los juegos altamente afectados por el grounding. Siendo estos: bait, boulderdash, chipschallenge, simplified-boulderdash, sokoban, zenpuzzle.

Mean times (s)	DUAL-BFWS	Saarplan	FD (A*)	FD (LAMA)	Madagascar
bait	119.26	-	-	-	8.94
boulderdash	22.91	-	-	-	-
butterflies	4.48	0.54	71.35	103.42	262.02
cakybaky	22.02	4.02	1.08	43.92	-
chipschallenge	34.18	-	-	-	-
iceandfire	0.46	0.12	0.12	1.22	369.77
simplified-bould.	11.88	-	-	-	-
sokoban	78.78	-	-	-	1.22
zelda	1.48	0.08	0.11	1.18	574.66
zenpuzzle	0.22	1.37	4.34	-	149.69

Tabla 6.5: Tiempos medios para cada uno de los juegos.

Times STD (s)	DUAL-BFWS	Saarplan	FD (A*)	FD (LAMA)	Madagascar
bait	210.55	-	-	-	0.00
boulderdash	13.84	-	-	-	-
butterflies	3.57	0.56	123.87	219.50	126.47
cakybaky	10.20	0.89	0.12	9.91	-
chipschallenge	65.61	-	-	-	-
iceandfire	0.11	0.04	0.03	0.39	237.16
simplified-bould.	4.62	-	-	-	-
sokoban	98.91	-	-	-	0.00
zelda	0.58	0.03	0.02	0.12	258.59
zenpuzzle	0.28	0.00	0.00	-	231.24

Tabla 6.6: Desviación estándar para cada uno de los juegos.

Capítulo 7

Conclusiones

En este memoria hemos presentado un proceso automático de ingeniería del conocimiento para generar una gran variedad de dominios de planificación para videojuegos. Hemos mostrado resultados experimentales en 10 juegos representados en el lenguaje de descripción de videojuegos VGDL, y potencialmente somos capaces de generar cualquier juego descrito en este lenguaje, incluyendo todos los presentes en el framework GVGAI (actualmente, más de cien). Este proceso requiere representar patrones de comportamiento de avatares y objetos en una base de conocimiento que serán usados posteriormente por un proceso de compilación, generando dominios listos para su uso para un planificador clásico. También se ha demostrado que este proceso, combinado con una componente reactiva en un agente basado en planificación, puede actuar y resolver cualquiera de estos juegos con el único conocimiento previo de las dinámicas del juego.

Aunque la abstracción y definición de plantillas requiere tanto conocimiento previo del entorno de juego como del lenguaje de descripción, estimamos que podemos reducir semanas de trabajo para un ingeniero del conocimiento comprometido a diseñar agentes deliberativos basados en planificación que actúen en estos juegos.

Adicionalmente, hemos realizados benchmarks con planificadores del estado del arte mostrando que dominios basados en videojuegos ofrecen casos de estudios diversos, comprensibles y desafiantes que pueden ser de interés para la comunidad internacional a la hora de considerar futuras líneas de mejora. Los experimentos indican resultados extremadamente pobres en problemas con muchas instancias, mayoritariamente causados por el proceso de grounding.

7.1. Trabajos futuros

A pesar de todo lo logrado en el proyecto se aprecia que existe un número de puntos en los que este puede mejorar y podrían ser considerados como trabajos futuros:

- **Representación del no-determinismo**. Este tipo de situaciones aparecen frecuentemente no solo en videojuegos, sino en problemas reales. En algunos casos un proceso continuo de replanificación y una estrategia reactiva puede ser suficientes para enfrentarse a este tipo de problemas. Aunque, por otra parte, si pretendemos crear un único dominio versátil y autosuficiente, creemos que la mejor aproximación sería incorporar técnicas de planificación probabilística.

  Otras aproximaciones interesantes podrían considerar técnicas FOND (Fully Observable Non-Deterministic) como [Daniele et al., 2000, Muise et al., 2014]. En ese caso, los dominios deberían recibir cambios adicionales para describir las características no-deterministas de los juegos.

- **Reducir efectos del grounding**: Hemos visto que para obtener un plan que resuelva el nivel completo los efectos del grounding pueden ser devastadores. La sobrecarga en tiempo y memoria ocasionada por todas las posibles instanciaciones vuelve en algunos casos el uso de planificación contraproducente. Si lo único que queremos es modificar el planificador, creemos que estrategias de *lifted planning* o grounding parcial, como [Ridder and Fox, 2014, Gnad et al., 2019], pueden resultar una buena opción a considerar.

- **Selección de subobjetivos**: Los objetivos producidos en estos dominios se enfocan en la resolución completa del nivel. Esto acarrea dos problemas: por una parte, dada la complejidad del dominio los recursos de tiempo y memoria necesarios pueden ser totalmente inviables; por otro, en juegos no deterministas los planes dejan de ser válidos tras unos pocos pasos, siendo innecesaria la planificación tan a largo plazo.

  Una mejor aproximación para estos problemas es la división y selección inteligente de subobjetivos. Esta tarea no es sencilla, puesto que ya no estamos considerando un único proceso de parseo. Es necesario analizar las mecánicas del juego y comprenderlo.

  Actualmente existen múltiples propuestas para la selección de subobjetivos en videojuegos, tales como [Núñez-Molina et al., 2020], que aprenden mediante Deep-Q Learning el mejor orden de subobjetivos para la resolución completa del nivel. Creemos que esta arquitectura puede funcionar perfectamente en nuestro entorno y la consideramos como un posible trabajo futuro.

- **Mayor cantidad de lenguajes**. Queda claro que los lenguajes usados en el proceso de compilación son muy influyentes en la calidad de los resultados. VGDL y PDDL han sido propuestos como candidatos por su fácil integración con el entorno GVGAI. Como vimos con HPDL, este proceso se puede generalizar fácilmente para otros lenguajes y otras descripciones de juegos e interacciones que mantengan características similares a las mencionadas en este proyecto.

Bibliografía

[ipc, 2018] (2018). International planning competition 2018 - classical tracks. https://ipc2018-classical.bitbucket.io/.

[Badia et al., 2020] Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D., and Blundell, C. (2020). Agent57: Outperforming the atari human benchmark.

[Bonanno et al., 2016] Bonanno, D., Roberts, M., Smith, L., and Aha, D. W. (2016). Selecting subgoals using deep learning in minecraft: A preliminary report. In *IJCAI Workshop on Deep Learning for Artificial Intelligence*.

[Castillo et al., 2010] Castillo, L., Morales, L., González-Ferrer, A., Fdez-Olivares, J., Borrajo, D., and Onaindia, E. (2010). Automatic generation of temporal planning domains for e-learning problems. *Journal of Scheduling*, 13:347–362.

[Černỳ et al., 2016] Černỳ, M., Barták, R., Brom, C., and Gemrot, J. (2016). To plan or to simply react? an experimental study of action planning in a game environment. *Computational Intelligence*, 32(4):668–710.

[Couto Carrasco, 2015] Couto Carrasco, M. (2015). Creació d'un controlador automàtic pel concurs gvg-ai. http://hdl.handle.net/10230/25487.

[Daniele et al., 2000] Daniele, M., Traverso, P., and Vardi, M. Y. (2000). Strong cyclic planning revisited. In Biundo, S. and Fox, M., editors, *Recent Advances in AI Planning*, pages 35–48, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Fdez-Olivares et al., 2011] Fdez-Olivares, J., Castillo, L., Cózar, J. A., and García Pérez, O. (2011). Supporting clinical processes and decisions by hierarchical planning and scheduling. *Computational Intelligence*, 27(1):103–122.

[Fdez-Olivares et al., 2006] Fdez-Olivares, J., Castillo, L., García-Pérez, Ó., and Palao, F. (2006). Bringing users and planning technology together. experiences in siadex. pages 11–20.

[Fickert et al., 2018] Fickert, M., Gnad, D., Speicher, P., and Hoffmann, J. (2018). Saarplan : Combining saarland ' s greatest planning techniques.

[Fox and Long, 2003] Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[Frances et al.,] Frances, G., Geffner, H., Lipovetzky, N., and Ramirez, M. Best-first width search in the ipc 2018: Complete, simulated, and polynomial variants. page 5.

[Geffner and Geffner, 2015] Geffner, T. and Geffner, H. (2015). Width-based planning for general video-game playing. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Ghallab et al., 2016] Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press, USA, 1st edition.

[Gnad et al., 2019] Gnad, D., Torralba, Á., Domínguez, M., Areces, C., and Bustos, F. (2019). Learning how to ground a plan – partial grounding in classical planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(0101):7602–7609.

[González-Ferrer et al., 2013] González-Ferrer, A., Fernández-Olivares, J., and Castillo, L. (2013). From business process models to hierarchical task network planning domains. *The Knowledge Engineering Review*, 28(2):175–193.

[GVGAI,] GVGAI. java-vgdl. https://github.com/GAIGResearch/GVGAI/wiki/VGDL-Language.

[Haslum et al., 2019] Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187.

[Helmert, 2006] Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246. arXiv: 1109.6051.

[Hoang et al., 2005] Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. (2005). Hierarchical plan representations for encoding strategic game ai. In *AIIDE*, pages 63–68.

[Höller et al., 2019] Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., and Alford, R. (2019). Hddl – a language to describe hierarchical planning problems.

[Kelly et al., 2008] Kelly, J. P., Botea, A., Koenig, S., et al. (2008). Offline planning with hierarchical task networks in video games. In *AIIDE*, pages 60–65.

[Martínez and Luis, 2018] Martínez, M. and Luis, N. (2018). Goal-reasoning in starcraft: brood war through multilevel planning. In *XVIII Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 2018) 23-26 de octubre de 2018 Granada, España*, pages 107–113. Asociación Española para la Inteligencia Artificial (AEPIA).

[McCluskey, 2002] McCluskey, T. (2002). Knowledge engineering: issues for the ai planning community.

[McCluskey et al., 2016] McCluskey, T. L., Vaquero, T., and Vallati, M. (2016). Issues in planning domain model engineering.

[McDermott et al., 1998] McDermott, D., Ghallab, M., Howe, A., Knoblock, C. A., Ram, A., Veloso, M., Weld, D. S., and Wilkins, D. (1998). Pddl-the planning domain definition language.

[Muise et al., 2014] Muise, C., McIlraith, S., and Belle, V. (2014). Non-deterministic planning with conditional effects. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24(1):370–374.

[Nau et al., 2003] Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

[Núñez-Molina et al., 2020] Núñez-Molina, C., Nikolov, V., Vellido, I., and Fernández-Olivares, J. (2020). Goal reasoning by selecting subgoals with deep q-learning.

[Parr and Quong, 1995] Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

[Perez-Liebana et al., 2016] Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couëtoux, A., Lee, J., Lim, C., and Thompson, T. (2016). The 2014 general video game playing competition.

[Richter and Westphal, 2010] Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

[Ridder and Fox, 2014] Ridder, B. and Fox, M. (2014). Heuristic evaluation based on lifted relaxed planning graphs. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24:244–252.

[Rintanen, 2011] Rintanen, J. (2011). Heuristic planning with sat: Beyond uninformed depth-first search. In Li, J., editor, *AI 2010: Advances in Artificial Intelligence*, pages 415–424, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Schaul,] Schaul, T. A video game description language (vgdl) built on top of pygame. https://github.com/schaul/py-vgdl.

[Schaul, 2013] Schaul, T. (2013). A video game description language for model-based or interactive learning. pages 1–8.

[Segura-Muros et al., 2017] Segura-Muros, J. A., Pérez, R., and Fernández-Olivares, J. (2017). Learning htn domains using process mining and data mining techniques.

[TIN2015-71618-R,] TIN2015-71618-R. Plan miner: Integración de planificación automática y minería de procesos para el aprendizaje de dominios de planificación jerárquica a partir de la experiencia almacenada en registros de actividad.

[Torrado et al., 2018] Torrado, R. R., Bontrager, P., Togelius, J., Liu, J., and Perez-Liebana, D. (2018). Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

[Vaquero et al., 2011] Vaquero, T., Silva, J., and Beck, C. (2011). A brief review of tools and methods for knowledge engineering for planning & scheduling. pages 7–14.

[Vasilev and Olivares, 2020] Vasilev, V. N. and Olivares, J. F. (2020). Desarrollo de una arquitectura reactiva y deliberativa usando planificación en el entorno de juegos gvgai. page 83.

[Vellido,] Vellido, I. Vgdl to htn parser. https://github.com/IgnacioVellido/VGDL-to-HTN-Parser.

[Vellido et al., 2020] Vellido, I., Fdez-Olivares, J., and Pérez, R. (2020). A knowledge based process for the generation of htn domains from vgdl video game descriptions. *ICAPS 2020 Workshop on Knowledge Engineering for Planning (KEPS2020)*.

[Vellido and Nikolov,] Vellido, I. and Nikolov, V. Vgdl-pddl: A planning-based agent capable of playing from only vgdl game descriptions. https://github.com/IgnacioVellido/VGDL-PDDL.

[Vellido and Olivares, 2020] Vellido, I. and Olivares, J. F. (2020). Generación de dominios de planificación jerárquica a partir de descripciones de videojuegos en vgdl. page 62.

[Vereecken,] Vereecken, R. Vgdl 2.0. https://github.com/rubenvereecken/py-vgdl.

Anexo A

Tablas de experimentos

Madagascar	time (s)										
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle	
0	8.94		505.53			295.71				35.05	
1						335.76				740.08	
2			258.99			885.99					
3									377.76	56.53	
4			158.88					1.22	555.86	214.47	
5			385.23			507.30					
6			266.64			215.17			124.26	20.56	
7			205.66			366.21			710.67	1.07	
8			63.19			4.35			829.02	69.42	
9			252.04			347.65			850.38	60.34	
Mean	8.94	#DIV/0!	262.02	#DIV/0!	#DIV/0!	369.77	#DIV/0!	1.22	574.66	149.69	
Std	0	#DIV/0!	126.46918	#DIV/0!	#DIV/0!	237.16399	#DIV/0!	0	258.58669	231.23772	
Coverage	1	0	8	0	0	8	0	1	6	8	**32**

Figura A.1: Tiempos con Madagascar.

FD (A*)	time (s)										
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle	
0			4.56	1.07		0.13			0.11		
1			369.38	1.17		0.14			0.11		
2			244.98	1.02		0.15			0.12		
3			88.55	1.05		0.13			0.09		
4			3.73	1.12		0.17			0.08		
5			0.64	1.01		0.11			0.11		
6			0.42	1.40		0.11			0.10		
7			0.33	1.03		0.11			0.11		
8			0.34	0.99		0.08			0.11	4.34	
9			0.55	0.94		0.07			0.15		
Mean	#DIV/0!	#DIV/0!	71.35	1.08	#DIV/0!	0.12	#DIV/0!	#DIV/0!	0.11	4.34	
Std	#DIV/0!	#DIV/0!	123.86686	0.122169	#DIV/0!	0.0274561	#DIV/0!	#DIV/0!	0.0183234	0	
Coverage	0	0	10	10	0	10	0	0	10	1	**41**

Figura A.2: Tiempos con FD(A*).

FD (LAMA)	time (s)										
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle	
0			61.86	40.35		1.31			1.13		
1				55.22		1.17			1.23		
2				39.67		1.37			1.30		
3			681.57	41.29		1.33			1.11		
4			48.83	45.47		2.24			0.90		
5			10.01	42.57		0.93			1.31		
6			6.70	68.04		1.18			1.18		
7			4.87	39.04		1.00			1.21		
8			5.40	34.79		0.70			1.30		
9			8.09	32.77		1.00			1.12		
Mean	#DIV/0!	#DIV/0!	103.42	43.92	#DIV/0!	1.22	#DIV/0!	#DIV/0!	1.18	#DIV/0!	
Std	#DIV/0!	#DIV/0!	219.50303	9.9142659	#DIV/0!	0.3930176	#DIV/0!	#DIV/0!	0.11624	#DIV/0!	
Coverage	0	0	8	10	0	10	0	0	10	0	**38**

Figura A.3: Tiempos con FD(LAMA).

Dual	time (s)										
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle	
0	0.01	42.49	6.07	18.01	11.00	0.48	11.36	275.71	2.07	0.53	
1	6.65	30.74	10.24	44.81	29.80	0.38	13.38	69.22	2.52	0.08	
2	0.16	25.81	0.09	19.57	218.07	0.50	10.32	14.94	1.32	0.95	
3	17.60	49.46	9.79	18.94	0.74	0.55	12.29	0.19	1.43	0.07	
4	29.07	2.62	6.22	13.15		0.71	0.52	0.07	0.73	0.09	
5	0.13	15.93	0.04	35.99	19.47	0.43	12.18	246.53	0.84	0.05	
6	40.25	23.03	0.04	21.56	1.79	0.44	19.40	130.52	1.39	0.06	
7	512.67	11.99	4.36	24.51	15.77	0.38	11.27	10.25	0.82	0.05	
8	20.80	12.73	3.20	11.32	3.88	0.30	11.42	27.86	1.51	0.27	
9	565.20	14.27	4.69	12.33	7.09	0.38	16.63	12.49	2.13	0.04	
Mean	119.26	22.91	4.48	22.02	34.18	0.46	11.88	78.78	1.48	0.22	
Std	210.55084	13.84217	3.5680929	10.197232	65.612416	0.1069738	4.6249011	98.905896	0.5753883	0.2836092	
Coverage	10	10	10	10	9	10	10	10	10	10	**99**

Figura A.4: Tiempos con Dual.

Saarplan	time (s)										
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle	
0			0.39	4.02		0.11			0.05		
1			2.00	5.45		0.13			0.14		
2			1.13	5.01		0.17			0.08		
3			0.51	4.36		0.15			0.07		
4			0.25	4.24		0.20			0.05		
5			0.24	4.84		0.11			0.08		
6			0.23	3.22		0.11			0.05		
7			0.21	3.13		0.09			0.09		
8			0.24	2.54		0.08			0.09	1.37	
9			0.21	3.39		0.09			0.10		
Mean	#DIV/0!	#DIV/0!	0.54	4.02	#DIV/0!	0.12	#DIV/0!	#DIV/0!	0.08	1.37	
Std	#DIV/0!	#DIV/0!	0.5555889	0.8881203	#DIV/0!	0.0367202	#DIV/0!	#DIV/0!	0.0254964	0	
Coverage	0	0	10	10	0	10	0	0	10	1	**41**

Figura A.5: Tiempos con Saarplan.

Madagascar	plan length									
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle
0	65		163			187				146
1						262				185
2			136			364				
3									153	147
4			98					38	133	181
5			142			241				
6			93			154			118	124
7			111			199			148	113
8			106			78			258	94
9			169			187			358	117

Figura A.6: Longitudes de planes con Madagascar.

FD (A*)	plan length									
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle
0			142	307		187			153	
1			175	532		238			208	
2			130	346		256			208	
3			134	493		208			148	
4			89	445		379			118	
5			103	310		199			148	
6			93	316		154			98	
7			99	505		199			138	
8			106	550		72			258	88
9			136	289		187			358	

Figura A.7: Longitudes de planes con FD(A*).

FD (LAMA)	plan length									
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle
0			142	307		187			153	
1				532		238			208	
2				346		256			208	
3			134	493		208			148	
4			89	445		379			118	
5			130	310		199			148	
6			93	316		154			98	
7			99	505		199			138	
8			106	550		72			258	
9			136	289		187			358	

Figura A.8: Longitudes de planes con FD(LAMA).

Dual	plan length									
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle
0	21	258	178	355	350	187	173	353	243	140
1	121	208	397	607	315	238	177	208	223	185
2	42	175	148	541	790	256	196	148	258	211
3	82	349	236	490	66	208	212	57	253	123
4	137	102	158	472		379	86	29	133	172
5	77	163	148	460	181	199	136	193	163	157
6	327	203	135	343	120	154	173	308	128	136
7	336	227	99	526	165	199	191	212	148	191
8	227	254	139	559	230	72	211	207	288	112
9	313	175	196	367	146	187	176	330	343	132

Figura A.9: Longitudes de planes con Dual.

Saarplan	plan length									
GVGAI level	bait	boulder	butterflies	cakybaky	chips	iceandfire	simpBould	sokoban	zelda	zenpuzzle
0			172	364		187			158	
1			472	685		238			208	
2			274	502		256			233	
3			194	622		208			158	
4			116	481		379			118	
5			109	601		241			148	
6			93	328		154			98	
7			99	613		199			163	
8			139	565		72			258	94
9			136	370		187			363	

Figura A.10: Longitudes de planes con Saarplan.