



ugr | Universidad
de Granada

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Desarrollo de un videojuego y de un agente
inteligente basado en razonamiento con
restricciones, planificación automática y búsqueda
heurística

Autor
Israel Puerta Merino

Directores
Pablo Mesejo Santiago
Jesús Giráldez Crú



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, septiembre de 2023

Desarrollo de un videojuego y de un agente inteligente basado en razonamiento con restricciones, planificación automática y búsqueda heurística

Israel Puerta Merino

Palabras clave: Planificación Automática, PDDL, Razonamiento con Restricciones, MiniZinc, Búsqueda Heurística, Inteligencia Artificial Simbólica.

Resumen

Este Trabajo de Fin de Grado (TFG) explora la hibridación de técnicas de Razonamiento con Restricciones (RR), Planificación Automática (PA) y Búsqueda Heurística (BH) y su aplicación en la resolución de problemas complejos. Para ello, se desarrolla un agente inteligente capaz de resolver un videojuego, creado también como parte de este TFG, consistente en un mundo laberíntico con visibilidad parcial en el que se ubican una serie de objetos de distinto valor, sobre los cuales se establecen restricciones (relacionadas con el peso máximo de los objetos que se puede portar y el orden en que son recogidos), y cuyo objetivo es alcanzar lo antes posible el portal de salida a la vez que se maximiza el valor de los objetos capturados. Este TFG utiliza GVGAI (plataforma de creación y evaluación de videojuegos y agentes inteligentes), MiniZinc (lenguaje de modelado de problemas de RR) y PDDL (lenguaje de modelado de problemas de PA), además de diversos algoritmos de BH, incluyendo la búsqueda offline (A^*), búsqueda incremental ($D^* \text{ Lite}$) y búsqueda en tiempo real (RTA^* , $LRTA^*$ y $LRTA^*(k)$).

RR, PA y BH son tres ramas clásicas de la Inteligencia Artificial cuyo uso combinado ha sido escasamente explorado. No obstante, parece razonable pensar que existen numerosos problemas cuya resolución pueda requerir, simultáneamente, satisfacer u optimizar una serie de restricciones, diseñar un plan que permita alcanzar un objetivo, y optimizar caminos en un espacio de búsqueda. El videojuego diseñado e implementado es un ejemplo de ello, específicamente ideado para mostrar la capacidad de resolución del agente inteligente desarrollado. Para ello, se ha experimentado con siete configuraciones diferentes de BH, observando que el agente obtiene, en todos los casos de esta batería experimental preliminar, mejores resultados que otras técnicas como *Monte Carlo Tree Search* o *Reinforcement Learning*.

Development of a video game and an intelligent agent based on heuristic search, reasoning with constraints and automated planning

Israel Puerta Merino

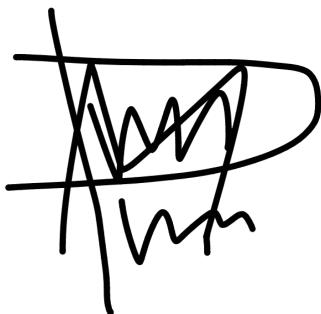
Keywords: Automated Planning, PDDL, Reasoning with Constraints, Mi- niZinc, Heuristic Search, Symbolic Artificial Intelligence.

Abstract

This Bachelor Thesis explores the hybridization of Reasoning with Constraints (RC), Automated Planning (AP) and Heuristic Search (HS) techniques and their application in complex problems solving. To this purpose, it is developed an intelligent agent capable of solving a video game, also created as part of this thesis, consisting of a labyrinthine world with partial visibility where there is a series of objects with different values, on which restrictions are established (related to the maximum weight of the objects that can be carried and the order in which they are collected), and whose objective is to reach the exit portal as soon as possible while maximizing the value of the captured objects. This thesis uses GVGAI (platform for creating and evaluating video games and intelligent agents), MiniZinc (RC problem modeling language) and PDDL (PA problem modeling language), as well as several BH algorithms, including offline search (A^*), incremental search (D^* Lite) and real-time search (RTA^* , $LRTA^*$ and $LRTA^*(k)$).

RC, AP and HS are three classic branches of Artificial Intelligence whose combined use has been scarcely explored. Nevertheless, it seems reasonable to think that there are numerous problems whose resolution may require, simultaneously, to satisfy or optimize a set of constraints, to design a plan to achieve a goal, and to optimize paths in a search space. The video game designed and implemented is an example of that, specifically devised to show the resolution capacity of the intelligent agent developed. For this purpose, it has been experimented with seven different HS configurations, observing that the agent obtains, in all cases of this preliminary experimental battery, better results than other techniques such as *Monte Carlo Tree Search* or *Reinforcement Learning*.

Yo, **Israel Puerta Merino**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75906386S, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

A handwritten signature in black ink, appearing to read "ISRAEL PUERTA MERINO". The signature is fluid and cursive, with a large, stylized letter "I" at the beginning.

Fdo: Israel Puerta Merino

Granada a 6 de septiembre de 2023.

D. Pablo Mesejo Santiago, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

D. Jesús Giráldez Crú, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Desarrollo de un videojuego y de un agente inteligente basado en búsqueda heurística, razonamiento con restricciones y planificación automática*, ha sido realizado bajo su supervisión por **Israel Puerta Merino**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 6 de septiembre de 2023.

Los directores:



Pablo Mesejo Santiago



Jesús Giráldez Crú

Agradecimientos

Quiero empezar agradeciendo a mis tutores, Pablo y Jesús, quienes han estado apoyándome, enseñándome y mostrándome el camino en todo momento, cada vez que me sentía perdido en el desarrollo de este proyecto y en la vida en general. Gracias a ellos he aprendido una infinidad de cosas sobre el desarrollo de proyectos, y se han convertido para mí en todo un referente de cómo hay que enseñar, razón por la que los lo llevaré siempre conmigo.

Quiero dar las gracias a los muchos familiares y amigos que han estado ahí para mí; ayudándome, escuchándome y animándome durante todo el transcurso (de casi un año) del desarrollo de este proyecto. Destacar aquí a mis padres y a mis compañeros de piso, quienes me han levantado tras cada caída y me han permitido seguir adelante en los momentos que sentía imposible avanzar.

Por último, me gustaría agradecer a todos profesores con los que he aprendido a amar esta carrera. Hablo de quienes no solo me han transmitido conocimientos de informática, sino también me han enseñado a disfrutarla y, sobre todo, me han hecho crecer como persona. No es necesario nombrarlos, ellos saben quienes son.

En general, agradezco mucho a la Universidad de Granada, pues esta institución, y esta ciudad en su conjunto, me han hecho crecer mucho, mucho más de lo que esperaba, a nivel académico y a nivel personal.

Índice general

1. Introducción	1
1.1. Descripción del Problema	2
1.2. Motivación	7
1.3. Objetivos	9
1.4. Estructura del Documento	9
2. Planificación	11
2.1. Metodología de Trabajo	11
2.2. Flujo de Trabajo	11
2.3. Planificación Temporal	13
2.4. Planificación Económica	15
3. Fundamentos Teóricos	17
3.1. Razonamiento con Restricciones	17
3.1.1. Problema de Satisfacción de Restricciones	17
3.1.2. Problema de Optimización de Restricciones	18
3.1.3. Resolución de Problemas con Restricciones	18
3.2. Planificación Automática	19
3.3. Búsqueda Heurística	22
4. Métodos y Tecnologías	25
4.1. Entorno para el Desarrollo de Videojuegos y Agentes Inteligentes	25
4.1.1. GVGAI	26
4.2. Modelado y Resolución de Problemas de Razonamiento con Restricciones	26
4.2.1. MiniZinc	27
4.3. Modelado y Resolución de Problemas de Planificación Automática	30
4.3.1. Planning Domain Definition Language	30
4.3.2. Planificadores	34
4.4. Algoritmos de Búsqueda Heurística	34
4.4.1. Búsqueda Offline	34

4.4.2. Búsqueda Incremental	37
4.4.3. Búsqueda en Tiempo Real	44
5. Diseño e implementación	49
5.1. Diseño	49
5.1.1. Videojuego: El Mochilero	49
5.1.2. Agente Inteligente	51
5.2. Implementación	55
5.2.1. Videojuego en VGDL	56
5.2.2. Agente Inteligente	62
6. Experimentación	71
6.1. Estudio Cualitativo	75
6.1.1. Módulo MiniZinc	75
6.1.2. Módulo PDDL	76
6.1.3. Módulo Heurístico	77
6.2. Estudio Cuantitativo	80
6.2.1. Módulos MiniZinc y PDDL	81
6.2.2. Módulo Heurístico	83
6.3. Estudio Comparativo de los algoritmos de Búsqueda Heurística	89
6.3.1. Comparación entre los Algoritmos en Tiempo Real . .	89
6.3.2. Análisis Comparativo General	90
6.4. Comparación con otras técnicas	92
6.4.1. Monte Carlo Tree Search	93
6.4.2. Aprendizaje por Refuerzo	95
7. Conclusiones y Trabajos Futuros	98
Bibliografía	104

Índice de figuras

1.1.	Ejemplo de mapa del videojuego desarrollado.	3
1.2.	Estado del mundo al inicio del juego.	3
1.3.	Esquema general del funcionamiento del sistema.	6
1.4.	Comparativa entre el número de publicaciones por año utilizando RR, PA y BH y sus combinaciones.	8
2.1.	Ejemplo gráfico de una red de flujo de trabajo Gitflow	13
2.2.	Planificación inicial del proyecto.	14
2.3.	Planificación final del proyecto.	15
3.1.	Estado inicial del problema del mundo de bloques.	20
3.2.	Objetivo del problema del mundo de bloques.	20
3.3.	Ejemplo de acción del problema del mundo de bloques.	20
3.4.	Ejemplo de problema de BH.	24
3.5.	Ejemplo de problema de búsqueda de caminos.	24
4.1.	Problema del coloreado de regiones de Australia.	28
4.2.	Problema del mono y el plátano.	33
4.3.	Simulación de ejecución del algoritmo A*.	37
4.4.	Ejemplo de ejecución del algoritmo A*.	37
4.5.	Simulación de ejecución del algoritmo D* Lite.	42
4.6.	Ejemplo de problema de BH tras eliminar una de sus aristas. .	43
4.7.	Simulación de ejecución del algoritmo D* Lite cuando detecta un cambio en algún coste.	43
4.8.	Simulación del algoritmo LRTA*.	45
4.9.	Simulación de ejecución del algoritmo RTA*.	46
4.10.	Simulación de ejecución del algoritmo LRTA*(k) con $k = 2$. .	48
5.1.	Ilustración que refleja, a nivel conceptual, cómo percibe el mundo el jugador.	50
5.2.	Arquitectura general del agente inteligente.	53
5.3.	<i>Level Description File</i> del nivel 1 del videojuego <i>El Mochilero</i> . .	59
5.4.	Ejemplo de representación de un recurso en forma de lingotes y de monedas.	60

5.5. Ejemplo de representación final escogida para los recursos.	60
5.6. Conjunto de todos los modelos utilizados para representar los recursos en el videojuego <i>El Mochilero</i>	60
5.7. Diagrama de secuencia del método plan()	65
5.8. Diagrama de secuencia del método findplan()	66
5.9. Diagrama de clases de los algoritmos de BH implementados. .	67
5.10. Diagrama de secuencia del método act() del controlador con A*.	68
5.11. Diagrama de secuencia del método act() del controlador con D* Lite.	69
5.12. Diagrama de secuencia del método act() del controlador con RTA*.	70
6.1. Mapa del nivel 2.	72
6.2. Mapa del nivel 3.	73
6.3. Mapa del nivel 4.	74
6.4. Resultado devuelto por el módulo MiniZinc.	75
6.5. Situación sobre la que se ilustran los comportamientos de los diferentes algoritmos de BH.	78
6.6. Replanificaciones realizadas por D* Lite y A* en la situación mostrada en la Figura 6.5.	79
6.7. Media y desviación típica de los tiempos de ejecución de Mi- niZinc y PDDL.	83
6.8. Mapa del nuevo nivel simple creado.	92
6.9. Esquema representativo del proceso iterativo de MCTS.	94
6.10. Modelo general de entrenamiento de un agente mediante AR	96
6.11. Estructura básica de un agente inteligente que implementa DQL.	97

Índice de tablas

2.1. Presupuesto del proyecto.	16
6.1. Simulación de ejecución del plan obtenido por el módulo PDDL.	77
6.2. Tiempos de ejecución del módulo MiniZinc, utilizando diferentes mapas y configuraciones de BH en el agente inteligente.	81
6.3. Tiempos de ejecución del módulo PDDL, utilizando diferentes mapas y configuraciones de BH en el agente inteligente.	81
6.4. Tiempos de ejecución del módulo MiniZinc, utilizando diferentes números de objetos.	82
6.5. Tiempos de ejecución del módulo PDDL, utilizando diferentes números de objetos.	82
6.6. Tiempos de ejecución de cada uno de los algoritmos de BH en cada uno de los mapas.	84
6.7. Número de veces que planifican cada uno de los algoritmos de BH en cada uno de los mapas.	84
6.8. Tiempo medio (en ms) que tarda en planificar cada uno de los algoritmos de BH en cada uno de los mapas.	85
6.9. Número de veces que expande nodos cada uno de los algoritmos de BH en cada uno de los mapas.	86
6.10. Número máximo de nodos en memoria que almacena cada uno de los algoritmos de BH en cada uno de los mapas.	87
6.11. Número de ticks que se emplean para terminar el juego con cada uno de los algoritmos de BH en cada uno de los mapas.	87
6.12. Datos extraídos experimentalmente sobre los algoritmos de BH utilizados.	91
6.13. Resultados obtenidos por el agente inteligente desarrollado en el nivel simple creado.	93
6.14. Resultados obtenidos utilizando MCTS en el nivel simple creado.	95

Capítulo 1

Introducción

Este Trabajo de Fin de Grado (TFG) consiste en el desarrollo de un agente inteligente que hibrida técnicas de tres de las principales ramas de la Inteligencia Artificial (IA): Razonamiento con Restricciones (RR), Planificación Automática (PA) y Búsqueda Heurística (BH); y su uso para resolver de forma automática un problema complejo, diseñado específicamente para este proyecto. Este problema ha sido modelado como un videojuego, al ser este un medio flexible y visual a la hora de simular problemas del mundo real en un contexto más simple y acotado. De esta forma, un agente (máquina o humano) puede resolver el problema mediante un procedimiento general: el agente recibe la información del estado del mundo, por medio de sus sensores, y reacciona de la forma que considere más adecuada, por medio de sus actuadores. Esta generalización del modelo de interacción permite realizar un primer desarrollo y evaluación del comportamiento de agentes inteligentes antes de ser aplicados a problemas reales, con el objetivo de pillar su comportamiento y detectar errores en una fase más depurada y menos costosa. En concreto, el juego desarrollado consiste en un mapa laberíntico donde el avatar recibe información parcial del mundo (esto es, sólo conoce la posición de los obstáculos cercanos, una forma de modelar el alcance de los sensores de visibilidad) y hay una serie de objetos, de cierto valor y peso, que se pueden coger y soltar. Entre los objetos, existe un orden de precedencias (es necesario haber cogido un objeto para poder obtener el siguiente) y el avatar tiene una “mochila” con un peso máximo de objetos que puede transportar. El objetivo del jugador es salir del laberinto en el menor tiempo posible, portando en la mochila un conjunto de objetos con el valor óptimo. El agente inteligente será capaz de, con las restricciones de peso máximo y orden de precedencia impuestas sobre los objetos, calcular un conjunto de objetos óptimo, generar un plan con la secuencia de acciones (coger o soltar) necesarias para alcanzar dicho conjunto, y realizar estas acciones de forma eficiente, optimizando los movimientos necesarios para completarlas.

En este capítulo se describe el problema de la creación del agente inteligente, la hibridación de las técnicas consideradas y el desarrollo del videojuego; se exponen las motivaciones para su estudio; y se enumeran los objetivos a alcanzar en este TFG. Finalmente, se describe la estructura del documento, con una breve introducción del contenido de cada uno de los capítulos.

1.1. Descripción del Problema

Un agente inteligente [1] es un sistema informático que percibe su entorno, procesa la información recibida y selecciona de forma autónoma la acción a realizar en cada momento, con el fin de cumplir unos objetivos determinados. En este TFG, el agente inteligente se diseña para combinar técnicas de las ramas anteriormente mencionadas (RR, PA y BH), de modo que el proceso de resolución de un problema se aborde siguiendo una secuencia de tres pasos: “Optimización → Planificación → Búsqueda”. Más adelante en este capítulo se profundiza en esta secuencia, explicando el funcionamiento del agente inteligente y cada uno de sus módulos; pero antes se procede a introducir el videojuego desarrollado, que será el problema a resolver por dicho agente, diseñado e implementado con el objetivo de mostrar el funcionamiento del agente inteligente desarrollado, exemplificar su uso y estudiar sus características.

El videojuego desarrollado consiste en un mundo cuadriculado laberíntico donde se encuentran distribuidos una serie de objetos, además de la salida del laberinto. Los objetos tienen un valor y peso asociados, además de un orden de precedencia entre ellos, y el jugador¹ dispone de una mochila con un límite de peso sobre los objetos que puede llevar al mismo tiempo. Inicialmente, el jugador conoce la posición de la salida y la de cada uno de los objetos, pero no la distribución del laberinto. Para descubrir dicha distribución, el avatar tiene una zona de visión a su alrededor, de forma que las casillas de esta zona sí son perceptibles, revelando qué se encuentra en ellas (muros o casillas transitables). Para mostrar esta última característica, y para ilustrar visualmente un nivel del juego, en la Figura 1.1 se puede ver un ejemplo con un nivel completamente visible y en la Figura 1.2 se muestra este mismo nivel desde el punto de vista del avatar. De esta forma, conforme el jugador se mueve por el mapa, va descubriendo nueva información sobre el laberinto a resolver. El objetivo del juego es salir del laberinto con el conjunto de objetos que mayor valor le proporcione, en el menor tiempo posible. Para lograrlo, el jugador puede desplazarse por el mapa, coger objetos del suelo y soltar objetos de su mochila.

¹El jugador puede ser controlado por un humano o por un agente inteligente.



Figura 1.1: Ejemplo de mapa del videojuego desarrollado, con el estado del mundo completamente visible (es decir, sin mostrar la visibilidad parcial del avatar). Las casillas en color verde oscuro son casillas transitables mientras que las de color verde claro son muros. El portal de salida se encuentra en la zona inferior centro, los cuadrados con 3 números representan objetos (con su posición en el orden de precedencias, su valor y su peso). Finalmente el avatar está en el centro del mapa, y una pequeña flecha rosa indica su orientación (derecha), permitiendo saber hacia donde se soltará un objeto en caso de realizarse dicha acción.



Figura 1.2: Estado del mundo al inicio del juego en el mapa ilustrado en la Figura 1.1, según la visibilidad parcial del avatar. Desde el principio se ven las posiciones de los diferentes objetos y de la salida, pero sólo se conoce la distribución del laberinto en las casillas cercanas al avatar.

Como ya se ha mencionado, el agente inteligente desarrollado resuelve el problema dividiéndolo en tres pasos (optimización, planificación y búsqueda), que se explican en detalle a continuación.

Primero, en el proceso de optimización, el agente inteligente resuelve un problema de restricciones indicado por el usuario, utilizando para ello la información proporcionada por el entorno del videojuego² (el conjunto de

²De esta forma, se simula la información que puede ser recopilada por los sensores de

objetos existentes en el mundo junto con su valor y peso asociados). Para ello, se emplean técnicas de RR [2, 3]. Este campo de la IA se ocupa de la resolución de problemas modelados por medio de un conjunto de variables y un conjunto de restricciones que deben ser satisfechas u optimizadas. Si las restricciones solo han de ser satisfechas, hablamos de un Problema de Satisfacción de Restricciones (PSR), y hallar una solución consiste en encontrar un conjunto de valores para las variables que cumpla todas las restricciones impuestas. Si se incluye una función objetivo (función a maximizar o minimizar), hablamos de Problemas de Optimización de Restricciones (POR), y el objetivo es encontrar una solución que, satisfaciendo todas las restricciones impuestas, optimice además dicha función objetivo. En esta primera fase, el agente inteligente modela un POR considerando los objetos existentes, sus valores y sus pesos, y calcula el conjunto óptimo de objetos que maximizan el valor recogido sin sobrepasar el peso máximo de la mochila.

A continuación, en el proceso de planificación, el agente inteligente fija la solución dada por el módulo anterior como objetivo final (es decir, el conjunto óptimo de objetos), e idea un plan para alcanzar dicho objetivo (es decir, una secuencia de acciones de coger y soltar objetos), utilizando para ello la información proporcionada por el entorno del videojuego (es decir, las acciones disponibles y el estado inicial del problema). En este punto se utilizan técnicas de PA [4]. Esta rama de la IA aborda la generación de planes que lleven de un estado inicial a un objetivo dado, utilizándose generalmente la siguiente representación del problema: los estados se definen como una sucesión de predicados lógicos, el objetivo es un conjunto concreto de condiciones que deben cumplirse, y las transiciones entre estados se modelan mediante acciones. Una acción especifica un conjunto de precondiciones (estado del mundo necesario para poder ejecutar la acción) y postcondiciones (estado del mundo tras la ejecución de la acción). De esta forma, el plan generado es una sucesión de acciones necesarias para, dado el estado inicial, alcanzar el objetivo. En el problema abordado, como consecuencia de la precedencia entre objetos, puede ser necesario obtener objetos que preceden a los que conforman el conjunto óptimo calculado en la fase anterior para posteriormente soltarlos. En esta fase, el plan resultante es una secuencia de acciones de coger y soltar objetos que permite alcanzar dicho conjunto óptimo considerando las restricciones de precedencias y el peso máximo en todo momento.

Finalmente, en el proceso de búsqueda, el agente inteligente ejecuta, una por una, la sucesión de acciones dada por el módulo anterior. Así, se establece la primera acción como objetivo parcial a corto plazo e, iterativamente, cuando se alcanza un objetivo parcial, se establece la siguiente acción como nuevo objetivo parcial. Si la acción es trivial, se utiliza un algoritmo simple

un agente inteligente en el mundo real.

para generar la sucesión de decisiones que el agente inteligente ha de tomar. En caso contrario, cuando la acción es compleja, se utiliza un algoritmo de BH [5]. Este campo se ocupa del estudio de algoritmos de búsqueda caracterizados por aprovechar información heurística sobre el medio para, dado un estado inicial y un objetivo, encontrar la sucesión de estados que permite llegar hasta algún estado que cumpla dicho objetivo. En estos algoritmos, la heurística funciona como una forma de pronosticar la distancia a la que se encuentra un cierto estado del estado final, de forma que se prioriza la exploración de los estados más prometedores. Es por ello que la eficacia de un algoritmo de BH está altamente condicionada por la capacidad predictiva de la información heurística que se tiene sobre el problema. La sucesión de decisiones que toma el agente inteligente en este caso es la que permite pasar, en cada momento, del estado en el que se encuentra al siguiente. En el problema abordado, este módulo será el encargado de decidir las acciones de bajo nivel (movimientos en el mapa) del agente inteligente, considerando las soluciones calculadas por los módulos anteriores así como la información del mapa. Recuérdese que el agente inteligente sólo tiene información parcial del mapa, no conociendo a priori la posición de los obstáculos existentes. De esta forma, este módulo de BH persigue conseguir una navegación eficiente a través del mapa.

En resumen, el objetivo del videojuego se puede dividir en tres partes bien diferenciadas, cada una dentro del marco de uso de una de las ramas que implementa el agente inteligente (RR, PA y BH), por lo que este debería ser capaz de resolver satisfactoriamente el problema de forma adecuada. Estas tres partes son:

1. Encontrar el conjunto de objetos que proporciona un mayor valor sin superar el peso máximo de la mochila. Este problema lo resuelve el primer módulo, mediante RR.
2. Diseñar un plan, en forma de secuencia de acciones (coger o soltar objeto) para obtener dicho conjunto. Este problema lo resuelve el segundo módulo, mediante PA.
3. Llevar a cabo, en orden, cada una de las acciones del plan, moviéndose por el mapa de forma eficiente. Este problema lo resuelve el tercer módulo, mediante BH.

Asimismo, debe destacarse que, además del diseño e implementación del agente inteligente para resolver este problema complejo, un aspecto fundamental de este TFG se centra en el propio desarrollo del videojuego, para ejemplificar su uso, así como ilustrar y analizar su comportamiento. Para esto último, también es necesario seleccionar un entorno de desarrollo a utilizar, que se encargue de controlar los estados de juego y de hacer de interfaz

con el agente inteligente, proporcionándole la información pertinente sobre el estado en cada instante y modificando el mundo en función de las acciones este le indique.

Para explicar de forma más ilustrativa el funcionamiento y flujo de información entre los diferentes componentes descritos en esta sección, se muestra en la Figura 1.3 un esquema general del funcionamiento del sistema. En primer lugar se define el videojuego, y se envía la información necesaria para definir las reglas del juego y el mapa al entorno, que hace de motor del juego y de interfaz con el agente inteligente. Este último recibe, por parte del usuario, la información necesaria para definir el problema de RR y el problema de PA a resolver; y, por parte del entorno, el estado del mundo. El agente inteligente divide el problema en tres pasos: (1) optimización (utilizando la información del estado del mundo y el problema de RR indicado, obtiene el conjunto de recursos objetivo a obtener); (2) planificación (con el estado del mundo y el conjunto de recursos objetivo, resuelve el problema de PA y obtiene un plan para obtener dicho objetivo); y (3) búsqueda (para cada objetivo parcial que compone el plan obtenido, se calcula la ruta a seguir). Para decidir la acción a realizar, el agente inteligente inteligente seguirá la lista de decisiones proporcionada por el último módulo. El agente inteligente volverá a llamar a dicho módulo si se acaba la lista, si alguna acción no es realizable, si se ha obtenido información nueva del entorno que se considere importante para la toma de decisiones o si se ha alcanzado el objetivo parcial (en ese caso, se establece un nuevo objetivo antes de realizar la llamada). El agente le indica al entorno, una a una, las acciones a realizar para, secuencialmente, ir alcanzando todos los objetivos parciales.

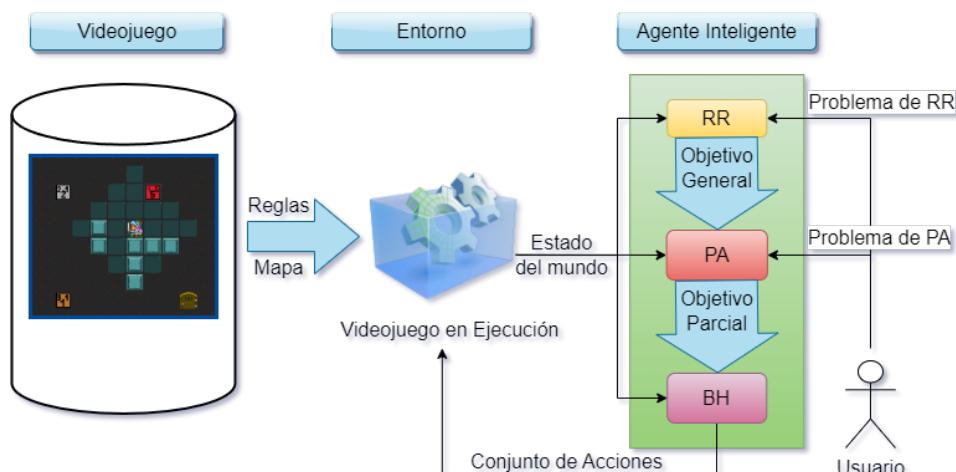


Figura 1.3: Esquema general del funcionamiento del sistema.

1.2. Motivación

RR, PA y BH son ramas clásicas de la IA, capaces de resolver adecuadamente aquellos problemas que se encuentren dentro de sus marcos teóricos y de aplicación. No obstante, es posible encontrarse problemas complejos que no se ajusten específicamente al marco de ninguno de estos campos en concreto, sino que se correspondan con una composición de estos. Nos referimos a problemas en los que se requiera optimizar una solución, generar un plan para alcanzarla, y realizar un proceso de búsqueda heurística que permita navegar por un espacio de forma eficiente para ejecutar cada una de las acciones del plan anterior. En estos casos, sería ideal que técnicas de cada una de estas ramas trabajasen conjuntamente, de forma que cada una se encargue de la resolución de una parte del problema. Por ello, en este TFG se exploran los resultados de hibridar estas tres ramas que, empleadas conjuntamente, pueden ser de gran utilidad y efectividad en la resolución de ciertas tareas complejas. La hibridación de estas ramas, además, está considerablemente menos exploradas que cada una de estas tres ramas de forma aislada, pues el número de publicaciones existentes sobre su uso conjunto es notablemente inferior al de su uso individual. Esto se puede ver reflejado en el análisis realizado, utilizando la base de datos bibliográfica Scopus.³ En ella, se han buscado artículos que contuvieran en el título, resumen o palabras clave, las consultas detalladas a continuación. Se han limitado los resultados al ámbito de las Ciencias de la Computación. A continuación, se muestra una lista de las consultas, todas realizadas el 02/07/2023, junto al número de publicaciones obtenidas:

- RR - Consulta: *constraint programming* - 58.512 resultados.
- BH - Consulta: *heuristic search* - 34.998 resultados.
- PA - Consulta: *automated planning* - 9.666 resultados.
- RR y BH - Consulta: *constraint programming heuristic search* - 1.755 resultados.
- PA y RR - Consulta: *constraint programming automated planning* - 250 resultados.
- PA y BH - Consulta: *automated planning heuristic search* - 221 resultados.

Como se puede ver, aunque el número de artículos sobre cada una de estas ramas es del orden de miles o decenas de miles, el número de artículos que los explora conjuntamente se encuentra en un orden bastante inferior

³Página web de Scopus: <https://www.scopus.com>

para cualquier combinación de estas.⁴ En la Figura 1.4 se muestra de forma visual la diferencia entre el número de artículos sobre cada una de las ramas y el número de publicaciones en las que se explora conjuntamente con otra de las ramas mencionadas. En todos los casos, el número de documentos en los que se utilizan dos ramas conjuntamente es notablemente inferior al de su uso de forma aislada. Además, a pesar de que estas ramas son cada vez más estudiadas (el número de documentos en los que aparecen de forma individual sigue una tendencia creciente con los años), sus apariciones en conjunto apenas aumenta. Estos resultados sugieren la hibridación de RR, PA y BH es un campo notablemente inexplorado.

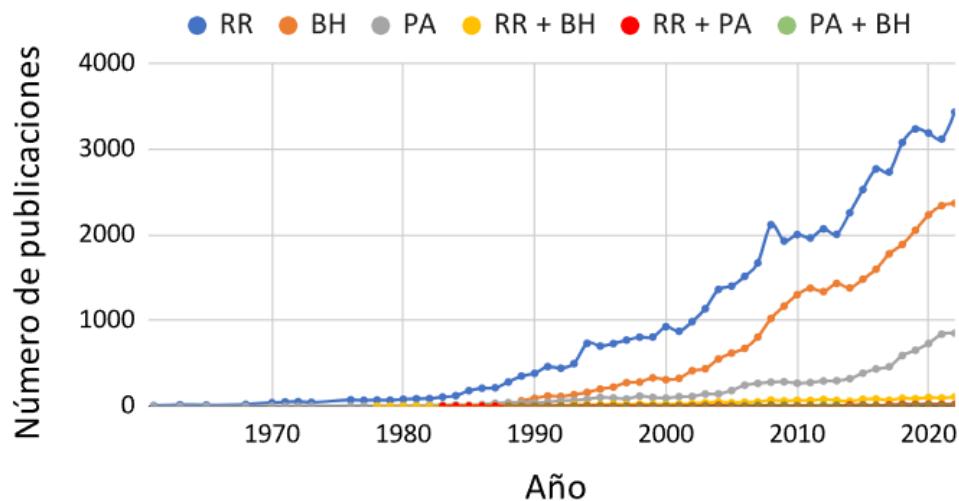


Figura 1.4: Comparativa entre el número de publicaciones por año utilizando RR, PA y BH y sus posibles duplas.

⁴Las búsquedas realizadas sobre las ramas en conjunto se han hecho con el objetivo de abarcar todos los resultados posibles (buscando minimizar el número de artículos sobre hibridación que se queden fuera). En consecuencia, aumenta el riesgo de falsos positivos (artículos que no tratan la hibridación pero aparecen en esta búsqueda), por lo que en realidad estamos estableciendo una cota superior del número real de artículos que abarcan la hibridación. Esto, no obstante, no entorpece el análisis ni sus conclusiones porque, aún así, el número de resultados es notablemente inferior al de las ramas estudiadas por separado.

1.3. Objetivos

El objetivo global de este TFG es el **desarrollo de un nuevo agente inteligente híbrido**, resultado de integrar técnicas de los campos de RR, PA y BH, de forma que este las use coordinadamente **para resolver un problema complejo, modelado en forma de un videojuego**. Para ello, será necesario crear un sistema con la estructura mostrada en la Figura 1.3. Para cumplir este objetivo general, se han definido los siguientes objetivos parciales:

1. Diseñar e implementar el videojuego al que se enfrentará el agente inteligente.
2. Diseñar e implementar el agente inteligente, que hibride técnicas de las RR, PA y BH, y las utilice para superar el juego. Esto, a su vez, requiere seleccionar e implementar las técnicas concretas a utilizar; crear las interfaces entre estas y el agente; y, por último, crear el controlador que interaccione con el entorno y las interfaces de forma adecuada.
3. Realizar una amplia validación experimental para evaluar los resultados que dicho agente inteligente obtiene a la hora de resolver el problema diseñado.

1.4. Estructura del Documento

Esta memoria se estructura en un total de siete capítulos. Estos son:

1. **Introducción:** Es el capítulo actual. En él se ha hecho una descripción general de los principales elementos del proyecto y del problema, se ha expuesto motivación para la realización de este proyecto, se han enumerado los objetivos a alcanzar y, por último, en esta misma sección se explica la estructura general de todo el proyecto.
2. **Planificación:** En este capítulo se explican las decisiones tomadas, a nivel organizativo, para el desarrollo del proyecto: método y flujo de trabajo utilizados, así como la planificación temporal y económica del proyecto.
3. **Fundamentos teóricos:** En este capítulo se exponen con detalle todos los conceptos necesarios para entender el resto de la memoria. Estos son, en definitiva, los cimientos sobre los que se sustenta todo el proyecto.

4. **Métodos y Tecnologías:** En este capítulo se expone el estado en el que se encuentran, en el momento de realización de este trabajo, las diferentes técnicas que se abarcan, y se explican los algoritmos y herramientas utilizadas para este proyecto.
5. **Diseño e implementación:** En este capítulo se exponen las decisiones tomadas para el diseño del videojuego y el agente inteligente, su estructura y sus características, así como los detalles de implementación y el funcionamiento concreto de todo el software desarrollado.
6. **Experimentación:** En este capítulo se hace un estudio de las características del agente inteligente desarrollado, analizando sus rendimientos en diferentes entornos y con diferentes algoritmos, así como comparándolo con otras técnicas utilizadas en el desarrollo de agentes inteligentes.
7. **Conclusiones y trabajos futuros:** En este último capítulo se hace un resumen del trabajo realizado, exponiendo las conclusiones de los resultados, y se enumeran posibles mejoras y trabajos futuros que se podrían realizar a raíz este proyecto.

Capítulo 2

Planificación

2.1. Metodología de Trabajo

La metodología define el procedimiento a seguir para la realización de una tarea. En este caso, se opta por emplear el desarrollo ágil de software [6, 7], que se basa en la idea de que un desarrollo debe ser iterativo e incremental.

Esta metodología prioriza la creación de un software funcional desde el principio, sobre el que, en cada iteración se pueda construir y pulir. Además, sobrepone la comunicación cara a cara entre los diferentes integrantes del proyecto por encima de la abundante documentación. Todo esto se consigue con iteraciones frecuentes, llamadas *sprints*, tras los que se concierta una reunión donde se expone el producto acabado, se habla sobre el estado del proyecto, y se eligen los objetivos a priorizar para la próxima reunión, bajo el criterio de cuáles aportan, directamente, más valor al trabajo.

Para este TFG, se opta por realizar *sprints* de 3 semanas. En cada reunión, el alumno expone, con el apoyo de diapositivas, el producto realizado esa iteración; a continuación se habla sobre el producto y sus posibles mejoras o ampliaciones; y por último se seleccionan las tareas de mayor prioridad y se fija la próxima reunión, para la cual se intentará tener dichas tareas terminadas.

2.2. Flujo de Trabajo

El flujo de trabajo indica los procesos a seguir durante la realización de las tareas. Estos son los mecanismos que un trabajador debe seguir a la hora de estructurar, realizar, ordenar y sincronizar las tareas. En general, define cómo fluye la información del proyecto, para poder hacer un fácil seguimiento.

En este caso, y ya que se utiliza Git como herramienta de control de versiones¹ se optó por utilizar el flujo de trabajo Gitflow [8]. Este define un modelo de creación de ramas en Git, indicando qué ramas crear, cuándo crearlas y cuándo fusionarlas. Las ramas existentes son:

1. **Rama principal (Main):** En ella, se mantiene un seguimiento de las versiones oficiales. Cada vez que una rama se fusiona en ella, será para generar una nueva versión oficial del proyecto, por lo que cada integración en Main estará etiquetado con un nuevo número de versión.
2. **Rama de desarrollo (Develop):** Es una rama complementaria a Main, donde los desarrolladores subirán todas las modificaciones del sistema que hagan. Cuando se crea, se hace mediante una copia del estado de Main en ese momento.
3. **Ramas de funcionalidades (Feature):** Cada vez que un desarrollador vaya a implementar una funcionalidad nueva para el sistema, debe crear una nueva rama, nacida de Develop. En ella, se trabajará en la implementación y, cuando la funcionalidad está terminada, se volverá a fusionar en Develop.
4. **Ramas de publicaciones (Release):** Cuando se acerque el momento de publicar una nueva versión, se creará una nueva rama, a partir de Develop, para preparar el lanzamiento. En ella no se deben añadir nuevas funcionalidades, sólo se trabaja en pulido, solución de errores y, en general, preparar el sistema para su nueva versión. Llegado el momento, se fusiona tanto en Main como en Develop, de forma que ambas ramas se ponen al mismo nivel. Para este proyecto, se ha considerado que los momentos de publicación eran las diferentes reuniones concertadas.
5. **Ramas de correcciones (Hotfix):** Cuando se encuentra, en el programa ya publicado, un error que es prioritario solucionar, se hace mediante estas ramas. Se crean desde Main, y cuando se corrige el error se fusiona nuevamente con Main y Develop.

En la Figura 2.1 se muestra un ejemplo gráfico del funcionamiento de este flujo de trabajo. Como se puede ver, a la rama Main solo se cargan nuevas versiones desde Hotfix (ramas que nacen para arreglar rápidamente errores emergentes) o Release (donde se revisa y prepara una versión estable para el salto de versión). Todos los cambios se realizan en la rama Develop o en ramas Feature que nacen de ella, permitiendo implementar de forma paralela varias características sin afectarse entre ellas

¹URL del repositorio en GitHub de este trabajo: <https://github.com/Corkiray/TFG>

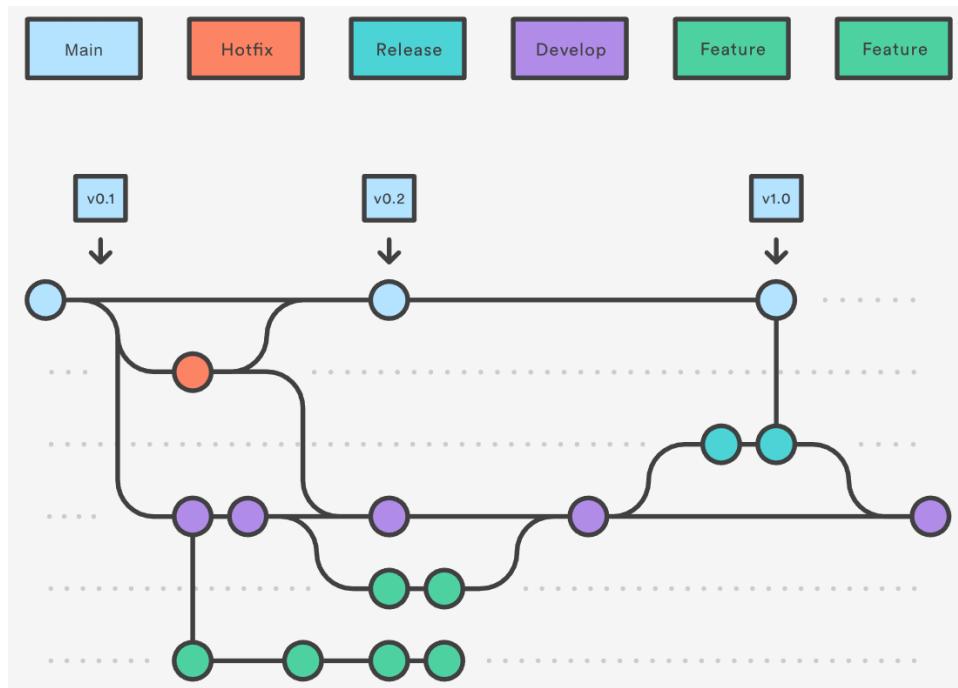


Figura 2.1: Ejemplo gráfico de una red de flujo de trabajo Gitflow. Imagen extraída de <https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>.

2.3. Planificación Temporal

El proyecto se ha desarrollado a lo largo del curso académico 2022/2023, comenzando en noviembre de 2022 y terminando en julio de 2023, con el objetivo inicial de presentarlo en la convocatoria ordinaria.

Para la planificación temporal, y siguiendo la filosofía de la metodología ágil, se ha dividido el proyecto en 6 grandes subproductos que lo conforman. Estos son (1) el videojuego, (2) las interfaces con RR y PA, (3) los algoritmos de BH, (4) el controlador inteligente, (5) la experimentación y (6) la memoria. Estos bloques se han fragmentado en un total de 20 *issues*, pequeñas tareas en forma de problemas a resolver, repartidas a lo largo de 10 *sprints*, cada uno de aproximadamente 3 semanas. En la Figura 2.2 se muestra el diagrama de Gantt que resume esta planificación inicial. Esta planificación se realizó en un primer *sprint*, y la división se hizo en vista de tener margen de tiempo suficiente para la posibilidad de realizar de *sprints* extra, consecuencia de los problemas y alteraciones que pueden emerger a lo largo de un desarrollo. A esta planificación se le añade el proceso de documentación, que es esencial durante todo el transcurso del desarrollo de un proyecto.

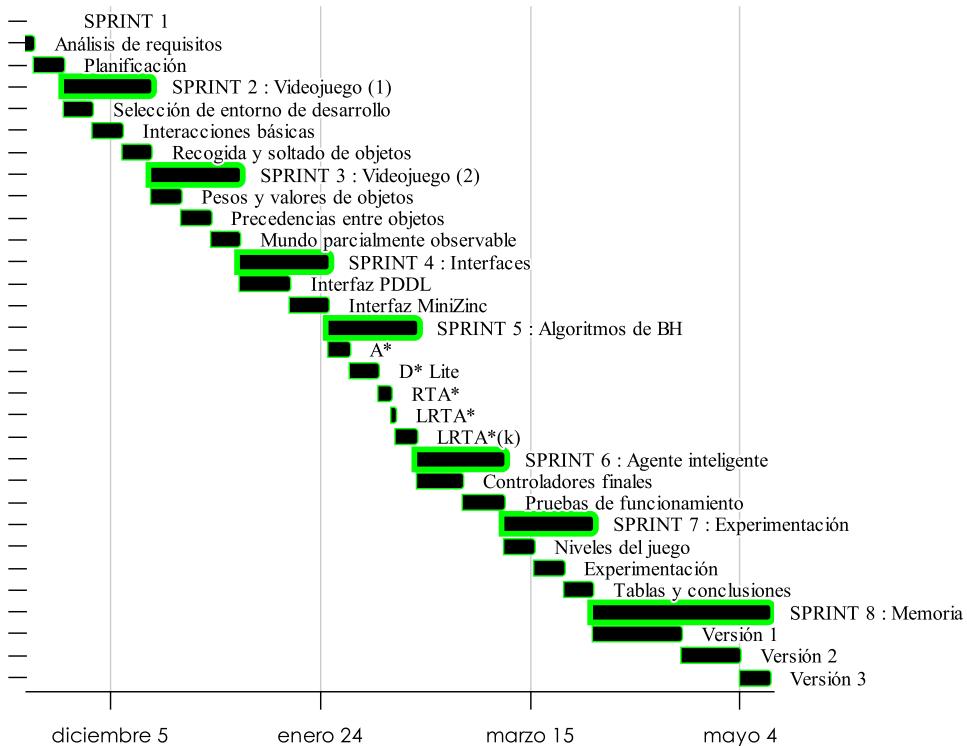


Figura 2.2: Planificación inicial del proyecto. Los bloques con borde fino representan el tiempo planificado para la resolución de una *issue* o tarea concreta. Los bloques más gruesos representan la duración planificada para cada uno de los *sprints*. Estos, además, tienen indicado el subproducto en el que se estará avanzando al completarlo.

Esta planificación no pudo ser cumplida, principalmente, por las siguientes razones:

- La carga de trabajo del primer cuatrimestre fue mucho mayor de lo esperado. Al ser el TFG una asignatura de 12 créditos correspondientes enteramente al segundo cuatrimestre, la diferencia de carga de trabajo entre las dos mitades del curso fue notable: se ha tenido que afrontar la primera mitad del desarrollo a la vez que se cursaban cinco asignaturas (30 créditos ECTS, 750 horas de trabajo); frente a las tres asignaturas (18 créditos, 450h) de la segunda mitad.
- El desarrollo del videojuego fue inesperadamente largo, pues las limitaciones del entorno utilizado dificultaron el proceso de implementación de algunas de las características. Las más problemáticas fueron la visibilidad parcial del mundo y el soltado de objetos, al no existir ningún precedente en los videojuegos proporcionados por el entorno utilizado.

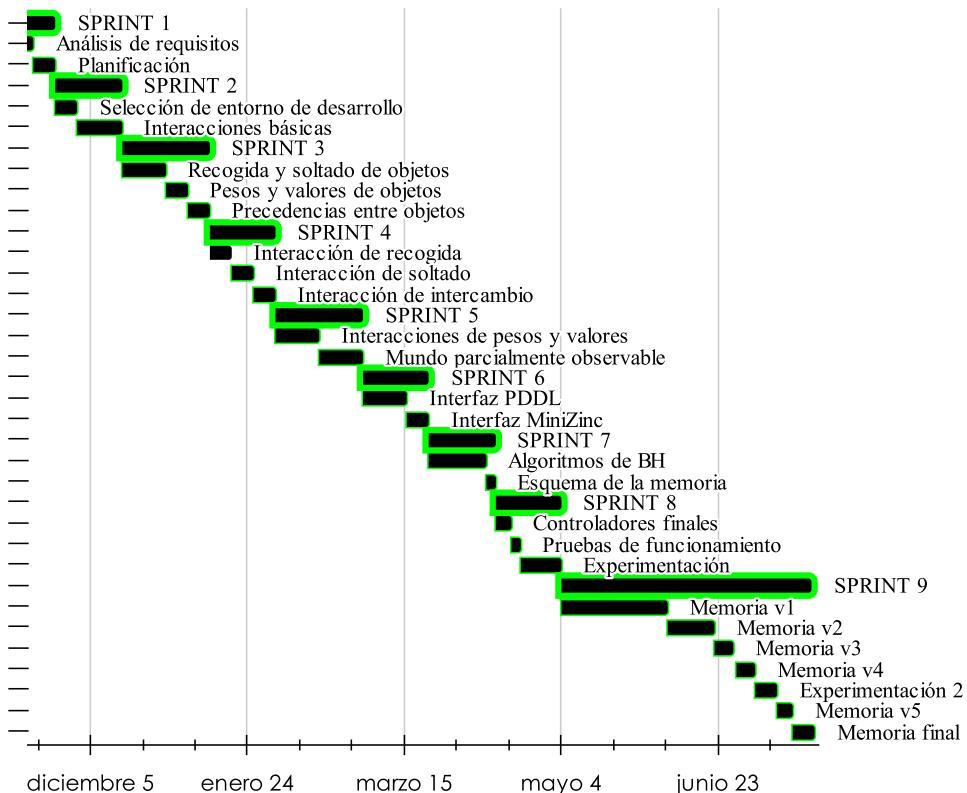


Figura 2.3: Planificación final del proyecto. Los bloques con borde fino representan el tiempo empleado para la resolución de una *issue* o tarea concreta. Los bloques más gruesos representan la duración de para cada uno de los *sprints* empleados.

En la Figura 2.3 se muestra el diagrama de Gantt con la planificación final del proyecto. Como se puede ver, finalmente se decidió presentar este TFG en la convocatoria extraordinaria.

2.4. Planificación Económica

Las herramientas de software utilizadas han sido MiniZinc, Metric-FF y GVGAI, todas completamente gratuitas, por lo que para el presupuesto solo es necesario tener en cuenta el material físico utilizado (un ordenador de gama media - 900€) y el tiempo invertido. Para cuantificar esto último, se toma como referencia el sueldo de un Investigador Senior o Responsable de I+D de una empresa tecnológica, con un salario medio de 35€/h. Como el TFG es una asignatura de 12 créditos ECTS, y cada crédito equivale (en teoría) a 25 horas de trabajo, inicialmente se asumieron 300 horas de trabajo. No obstante, finalmente fueron muchas más: con 270 días de trabajo, a una

media de 6h diarias, hace un total de 1.620 horas de trabajo, lo que se traduce en un total de 56.700€. El presupuesto final (véase Tabla 2.1) es de 57.600€.

Ítem	Costo
Salario	56.700€
Ordenador de Gama Media	900€
Total	57.600€

Tabla 2.1: Presupuesto del proyecto.

Capítulo 3

Fundamentos Teóricos

3.1. Razonamiento con Restricciones

El RR trata la resolución de problemas modelados por medio de un conjunto de variables y un conjunto de restricciones. Estos problemas pueden ser de dos tipos, un PSR o un POR, por lo que en los siguientes subapartados, se explican las características de cada uno de ellos.

3.1.1. Problema de Satisfacción de Restricciones

Un PSR queda definido por 3 elementos:

- **Variables (X).** Conjunto de elementos que conforman el problema y que pueden tener múltiples estados.
- **Dominios (D).** Dominio de cada variable, es decir, el conjunto de posibles estados que puede adoptar cada una de ellas.
- **Restricciones (C).** Conjunto de restricciones sobre las variables.

Así, el problema se reduce encontrar una asignación para cada variable tal que todas las restricciones sean satisfechas. A continuación, se puede observar un ejemplo simple de modelado de un PSR, donde el problema sería “encontrar dos enteros positivos tales que el primero es mayor que el segundo”.

```
1 X = { x , y }
2 D = { [1, inf) , [1, inf) }
3 C = { x > y }
```

Al proceso de encontrar una solución a un PSR se le llama Satisfacción de Restricciones (SR).

3.1.2. Problema de Optimización de Restricciones

Un POR es una extensión del PSR donde también se indica una función objetivo a maximizar o minimizar. El objetivo ahora será encontrar, de entre todas las posibles soluciones que satisfagan las restricciones, aquella que maximice (o minimice) la función objetivo. A continuación, se puede observar un ejemplo simple de modelado de un POR. Este es análogo al ejemplo anterior pero añadiendo como función objetivo “encontrar los números cuya suma sea lo más pequeña posible”.

```

1 X = { x , y }
2 D = { [1, inf) , [1, inf) }
3 C = { x > y }
4 f(x,y) = x+y
5 minimizar f(x,y)

```

Al proceso de encontrar una solución al POR se le llama Optimización de Restricciones (OR).

3.1.3. Resolución de Problemas con Restricciones

El RR aúna conceptualmente la SR y la OR, pues la forma de afrontar sus problemas es muy similar. El proceso de encontrar la solución a un problema de RR suele dividirse en dos partes, explicadas a continuación [9]:

- **Modelado.** Representar el problema como un conjunto de restricciones interpretables por un programa. Para ello, se requiere de un lenguaje en el que este se pueda definir formalmente (llamado lenguaje de modelado de restricciones) y que, posteriormente, una herramienta de modelado traduzca el problema definido a un modelo.
- **Resolución.** Dado un problema ya modelado, buscar la solución que satisfaga las restricciones (y optimice la función objetivo, si es un POR). A la herramienta utilizada para ello se la llama resolutor. Para resolver un modelo, bien se puede diseñar de forma que su estructura concuerde con la entrada del resolutor a utilizar, o bien se puede utilizar un traductor que transforme el modelo al formato que se requiera.

3.2. Planificación Automática

La PA [4] es una rama de la IA que se centra en el desarrollo de planes para solucionar problemas. Como estos problemas pueden ser hasta problemas reales, se ha de hacer una representación fidedigna del entorno en el que se encuentra el problema y todas las características que lo definen. A este entorno se le llama mundo, y un estado en PA representa la definición completa de un posible estado de dicho mundo. Con esto, se puede definir un problema de PA mediante tres elementos, los cuales son descritos a continuación, acompañados de un ejemplo ilustrativo donde se muestran estos elementos dentro de un problema concreto que consiste en, dado una serie de bloques apilados, reapilarlos en el orden correcto:

- Un **estado inicial** del mundo (Figura 3.1).
- Un **objetivo** a alcanzar, de forma que se considera resuelto un problema cuando se consigue llegar a un estado en el que se cumpla dicho objetivo (Figura 3.2).
- Un **conjunto de acciones** que se pueden realizar y que modifican el estado del mundo cuando se realizan (Figura 3.3).

Así pues, el objetivo de la PA es encontrar una sucesión de acciones que lleve del estado inicial a un estado que cumpla el objetivo. Para ello, su principal característica diferenciadora es su forma de representar los problemas, basada en condiciones:

- Un estado se define como una sucesión de condiciones.
- El objetivo es un conjunto de condiciones que se deben cumplir.
- Las acciones tienen una serie de precondiciones (condiciones que ha de cumplir el estado para que esta sea realizada) y postcondiciones (efecto que tendrá la acción en el estado, que pueden ser condiciones eliminadas y/o nuevas condiciones establecidas).

Aunque idealmente una representación del problema debería reflejar completamente el mundo en el que se encuentra, esto no siempre es posible: si queremos utilizar PA para resolver un problema del mundo real, este es inviablemente grande, además de que nunca podemos asegurar saber todos los efectos que una acción tiene en el mundo real. Es por ello que es necesario hacer algunos supuestos sobre el mundo, para simplificar la definición del estado. Uno de los más comunes y destacables es la Hipótesis del Mundo Cerrado (HMC), que consiste en asumir como falso todo aquello que no sea conocido. De esta forma, se procede a obviar todas las características

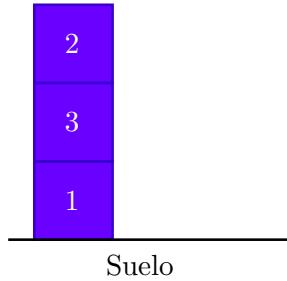


Figura 3.1: Estado inicial del problema del mundo de bloques. En él, los únicos elementos del mundo son el suelo y tres bloques enumerados que se encuentran apilados sobre él.

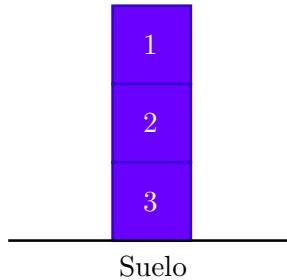


Figura 3.2: Objetivo del problema del mundo de bloques. Para alcanzarlo, hay que cumplir las condiciones que se ven en la imagen: que los tres bloques estén apilados en orden (el bloque 1 sobre el 2, y este sobre el 3), y que este último se encuentre sobre el suelo. La posición del resto de elementos, si los hubiera, no sería relevante para la solución.

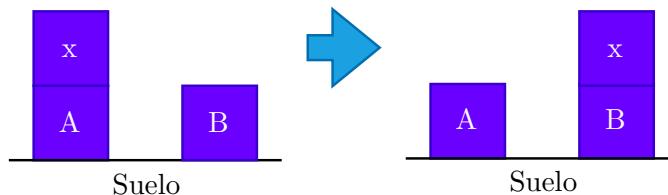


Figura 3.3: Ejemplo de acción del problema del mundo de bloques. En este caso, solo existe una acción posible, que es desplazar un bloque desde la parte superior de una pila hasta otra o al suelo. En el ejemplo ilustrado, se mueve el bloque x de la pila A a la pila B .

del mundo que no se consideren necesarias para el problema, así como todas las consecuencias que no se conozcan de una acción. Bajo este supuesto, se pueden formular las condiciones mediante predicados (afirmaciones simples sobre el mundo al que pertenecen) y expresiones lógicas entre ellos. Un predicado puede definir una característica del mundo, una propiedad de un objeto o una relación entre varios de ellos. Ejemplos de predicados son “x es azul”, “x es mayor que y” o sencillamente “llueve”. Estas afirmaciones pueden ser verdaderas, si se encuentra el predicado en el estado del mundo, o falsas, en caso contrario. De esta forma, el estado inicial se puede definir como un conjunto inicial de predicados; el objetivo es una expresión lógica entre predicados que se busca volver verdadera; las precondiciones son expresiones lógicas que deben ser verdaderas para que se pueda ejecutar la acción; y las postcondiciones son predicados que se añaden al estado del mundo (se vuelven verdaderos) o se eliminan de este (se vuelven falsos). El problema del mundo de bloques, utilizado como ejemplo, se puede modelar como predicados, por ejemplo de la siguiente manera:

```
1  Predicados relevantes (no ignorados por la HMC):
2      Sobre(x, y) : El objeto x se encuentra sobre el objeto y
3      Libre(x) : El bloque x no tiene otro encima.
4
5  Estado inicial:
6      Sobre(2,3)
7      Sobre(3,1)
8      Sobre(1,S)
9
10 Objetivo:
11     (Sobre(1,2) AND Sobre(2,3) AND Sobre(3,S))
12
13 Acciones:
14     Mover(x, A, B): Mover un objeto de una pila a otra.
15     Precondicion:
16         (Libre(x) AND Libre(B) AND Sobre(x,A))
17     Nuevos predicados:
18         Libre(A)
19         Sobre(x,B)
20     Predicados eliminados:
21         Libre(B)
22         Sobre(x,A)
23
24     Mover(x, A): Mover un objeto de una pila al suelo.
25     Precondicion:
26         (Libre(x) AND Sobre(x,A))
27     Nuevos predicados:
28         Libre(A)
29         Sobre(x,S)
30     Predicados eliminados:
31         Sobre(x,A)
```

Para resolver un problema de PA son necesarios dos pasos bien diferenciados:

- **Representar el problema** en un modelo bien definido que pueda ser interpretable por un programa. Al lenguaje utilizado para definir un modelo se le llama lenguaje de planificación. Un ejemplo de lenguaje de planificación es PDDL [10].
- **Obtener el plan.** Al algoritmo que resuelve un problema de PA se le llama planificador. Este es un algoritmo que, dado un estado inicial, un objetivo y todas las acciones que tiene disponible (así como sus precondiciones y postcondiciones) devuelve el plan a realizar para alcanzar el objetivo. Un ejemplo de planificador es FF [11].

Generalmente, también es necesario utilizar un programa que conozca tanto el lenguaje de planificación utilizado como el planificador a ejecutar, para que actúe a modo de interfaz entre ellos. Por ejemplo, Metric-FF [12] es un sistema encargado de ejecutar FF para un problema escrito en PDDL.

3.3. Búsqueda Heurística

Los algoritmos de búsqueda engloban a todos aquellos algoritmos que están diseñados para resolver un problema de búsqueda. Esto es, todo problema que se pueda expresar de la forma “dado un objeto x encontrar la estructura y ”. Esta definición es demasiado general, por lo que comúnmente se formalizan como un problema de espacios estados [1, 5]. En ellos, un problema consiste en, dado un estado inicial, llegar a un estado objetivo. Para ello, un problema de búsqueda puede quedar definido mediante 4 elementos:

- **Conjunto de Estados:** Es el objeto o espacio en el que se está explorando. Define todos los posibles estados.
- **Estado Inicial:** Es el estado desde el que se inicia la búsqueda.
- **Estado Objetivo:** Es el estado al que se pretende llegar. Este podría ser en realidad un conjunto de estados objetivos, de forma que llegar a cualquiera de ellos se consideraría solución, o una función test, que evalúa un estado e indica si es objetivo o no.
- **Función de Sucesión:** Una función que indique, para cada estado, todos aquellos a los que es posible desplazarse directamente desde él.

Bajo esta premisa, un problema puede entenderse como un árbol, en el que el estado inicial es el nodo raíz y la función de sucesión genera los nodos

hijos de un nodo dado. De esta forma, todo algoritmo de búsqueda se puede reducir, de forma general, al siguiente comportamiento:

1. Partir del nodo raíz.
2. Expandir el nodo actual (esto es, llamar a la función de sucesión para obtener los nodos hijos).
3. Determinar qué nodo explorar a continuación.
4. Continuar hasta que se llegue a un nodo objetivo.

Un ejemplo simple es la búsqueda de un número concreto en un vector de enteros ordenado. Este problema se puede expresar de la forma anteriormente mencionada: “dado un objeto (el vector), llegar a un estado objetivo (el número concreto)”. También se puede modelar mediante un problema de espacios de estados, donde el conjunto de estados es el vector, el objetivo es un estado que contenga dicho número, el estado inicial es el primer elemento del vector y la función de sucesión siempre genera el elemento inmediatamente siguiente del vector.

Los algoritmos de búsqueda se pueden dividir en informados (es decir, que conocen alguna información de utilidad sobre el problema y la utilizan para deducir qué decisión es potencialmente mejor) y no informados. Los algoritmos de búsqueda informada, o BH [5], modelan su comportamiento en función de la información recibida. Para ello, se utiliza una función heurística que proporciona una puntuación a cada estado. Su uso busca aumentar la velocidad a la que se alcanza el objetivo, minimizando la cantidad de veces que es necesario expandir un estado. Tanto la función heurística utilizada, como la estructura del algoritmo de búsqueda en sí, definen la forma en la que cada algoritmo explorará el conjunto de estados en busca del objetivo. En la Figura 3.4 se muestra un ejemplo de problema de BH. Como se puede ver, consiste en un grafo con pesos en las aristas e información heurística sobre los nodos. En la Figura 3.5 se puede ver otro ejemplo, en este caso un problema de búsqueda de caminos en un mapa de casillas. Esto se puede representar mediante espacios de estados, y resolver interpretándolo como un grafo en el que los nodos son las casillas y las aristas las posibles transiciones entre ellas.

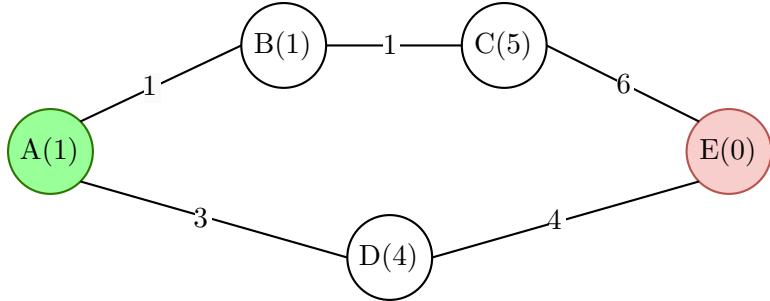


Figura 3.4: Ejemplo de problema de BH. Consiste en un grafo cuyas aristas tienen un peso asociado (esto es una abstracción del coste que conlleva desplazarse entre los nodos; que en un problema real podría referirse al tiempo de ejecución, distancia, dinero, etc). Además, todos los nodos tienen, entre paréntesis un valor asociado que corresponde a su valor heurístico $h(n)$, que es una estimación del coste que tendrá alcanzar el objetivo desde él. Esta estimación puede llegar a ser completamente acertada (como en C) o bastante mala (como en A). El nodo verde es el estado inicial y el rojo el objetivo.

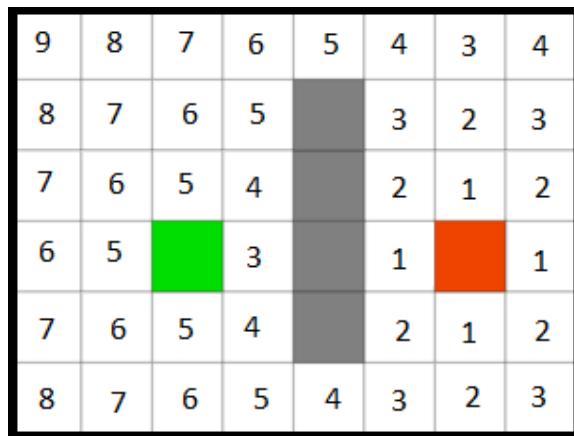


Figura 3.5: Ejemplo de problema de búsqueda de caminos. Cada cuadrícula (excepto las grises) es un nodo transitable, la roja es el nodo objetivo y la verde el nodo inicial. El número que hay anotado en cada casilla es su puntuación según la función heurística, que en este caso corresponde a la distancia Manhattan entre ella y la casilla objetivo. La función de sucesión de cada casilla devuelve sus adyacentes.

Capítulo 4

Métodos y Tecnologías

4.1. Entorno para el Desarrollo de Videojuegos y Agentes Inteligentes

Para el desarrollo de un agente inteligente es conveniente utilizar un entorno adecuado que proporcione las herramientas necesarias para que el agente pueda interactuar con su ambiente, así como modelar fácilmente el mundo en el que se mueve. Hay muchas herramientas que dan estas facilidades; frameworks y APIs que se utilizan para el desarrollo e investigación en el campo de los agentes inteligentes. Algunos ejemplos son Griddly, PyGame, MiniGrid, DeepMind Lab2D y GVGAI, que se explican brevemente a continuación.

- **Griddly** [13]. Es un proyecto en Python que busca facilitar la investigación en mundos basados en grids (rejillas). Proporciona herramientas para facilitar la creación de videojuegos (además de una selección de juegos pre-creados), la interacción de los agentes con ellos y una interfaz con *OpenAI Gym* [14] para facilitar la creación de agentes que utilicen aprendizaje por refuerzo.
- **PyGame** [15]. Es un conjunto de módulos Python que está más centrado en el desarrollo de videojuegos. Proporciona mas herramientas de desarrollo que permiten la creación de videojuegos complejos. Por otra parte, no dispone de herramientas para facilitar la creación de agentes ni interfaz con *OpenAI Gym* [14].
- **MiniGrid** [16]. Es una librería, también desarrollada en Python para el desarrollo de técnicas de aprendizaje por refuerzo. Cuenta con una colección de entornos fácilmente configurables donde poner a prueba al agente inteligente.

- **DeepMind Lab2D** [17]. Es un entorno destinado a la creación de mundos basados en grids, dirigido a la investigación con agentes de Aprendizaje Automático. Tiene API para C y para Python.
- El grupo de investigación *Game AI* de la Universidad *Queen Mary* de Londres¹ tiene múltiples plataformas destinadas a la investigación con agentes. Algunos de estos son **The Tabletop Games Framework** [18], entorno para juegos de mesa; **Stratega** [19], para juegos de estrategia; o **General Video Game AI (GVGAI)** [20], para mundos basados en grids, que es el framework utilizado para este proyecto.

4.1.1. GVGAI

GVGAI es un entorno de desarrollo de videojuegos y agentes inteligentes que nace para una serie de competiciones llamadas *The General Video Game Competition*. En ellas, los concursantes intentan implementar el agente inteligente que mejor juegue a una serie de videojuegos desconocidos para él. Este entorno parece estar abandonado, pues la última competición fue en 2020, a día de hoy la página web oficial no funciona y sus redes sociales están inactivas. Aun así, se ha decidido utilizar este entorno, porque es un framework completo y sencillo de utilizar y porque ya existe un precedente de integración en GVGAI que se puede utilizar como base. Este es GVGAI-PDDL [21], un proyecto en el que se realiza integración de PA en dicho entorno. En él, se crea un agente basado en PA que utiliza plenamente técnicas de planificación para jugar, por lo que es una referencia muy útil para nuestro objetivo actual.

4.2. Modelado y Resolución de Problemas de Razonamiento con Restricciones

Por lo general, para el RR se ofrecen *toolkits* (sets de herramientas) que integran todas las partes del proceso en un mismo software, el cual hace, además, de interfaz entre la herramienta de modelado y el resolutor. Estos sets suelen funcionar de la siguiente manera [9]:

1. Incluyen su propio lenguaje de modelado de restricciones, en el que el usuario puede definir el problema.
2. La herramienta de modelado transforma el problema en un modelo general.

¹Enlace a su página web: <https://gaigresearch.github.io>

3. Según el resolutor indicado por el usuario, el traductor transforma el modelo para que encaje con su entrada.
4. Finalmente, se ejecuta el resolutor y se muestra la salida, traducida al lenguaje natural.

En <http://www.constraint.org/en/tools/> podemos encontrar una amplia recopilación de herramientas utilizadas para RR. MiniZinc, Eclipse, la *toolchain* (cadena de herramientas) *Conjure - Savile Row - Minion* [9], Curry, Gecode, Choco, OR-tools, CPLEX, y CSPlib son los recursos más famosos.

4.2.1. MiniZinc

MiniZinc [22] es la herramienta de RR utilizada en este proyecto, por lo que es de interés explicar su funcionamiento.

El software MiniZinc es una cadena de herramientas cuyo corazón es el compilador, que traduce los modelos (escritos en un lenguaje con el mismo nombre) a FlatZinc, un lenguaje intermedio preparado para que pueda ser entendido por una gran variedad de resolutores (la lista completa está disponible en la web <https://www.minizinc.org/software.html>).

4.2.1.1. El Lenguaje de Modelado de Restricciones de MiniZinc

El lenguaje MiniZinc permite modelar un PSR o un POR de forma cómoda e intuitiva, al tener una sintaxis cercana a los lenguajes de programación convencionales. Todo problema se define en un archivo de modelo, que se identifica por la extensión “.mzn”. Un modelo MiniZinc se define como una sucesión de objetos, separados entre ellos por punto y coma. Es importante destacar que MiniZinc no tiene en consideración el orden en el que se definen los objetos, por lo que se pueden declarar en cualquier secuencia. A continuación, se muestran los principales objetos que se pueden definir en archivo de modelo (información extraída del manual de usuario de MiniZinc <https://www.minizinc.org/doc-2.7.5/en> [23]):

- **Include:** – `include <filename>;` – Similar a otros lenguajes de programación, sirve para insertar el contenido de otros archivos y ampliar la funcionalidad con librerías.
- **Definición de conjuntos:** – `set of <type>: <name>= { <expr-1>, ..., <expr-n>}` – Permite definir conjuntos de elementos, que se puede emplear como dominios complejos para las variables.

- **Declaración de variables:** – `var <domain>: <name>;` – El dominio puede ser sencillamente un tipo de datos (`int`), un rango de datos (`1..10`) o un conjunto definido.
- **Declaración de parámetros:** – `<type>: <name>;` – Análogo a las variables, pero sin “var” al principio. Los parámetros no soportan dominios específicos, solo el tipo de dato.
- **Asignar valor a variables o parámetros:** – `<name>= <exp>;` – Esta línea se puede integrar con la declaración en un solo objeto que haga, al mismo tiempo, la función de definición y la de asignación.
- **Restricciones:** – `constraint <Boolean expression>;` – Permite añadir una restricción al modelo. La solución encontrada deberá hacer verdadera la expresión booleana indicada.
- **Solve:** – Especifica el tipo de solución que buscamos, hay 3 posibilidades: minimizar una función, maximizarla o solo satisfacer las restricciones:
`solve satisfy;`
`solve maximize <arithmetic expression>;`
`solve minimize <arithmetic expression>;`
- **Output:** – `output [<string expression>, ..., <string expression>];` – Permite formatear la salida. Por ejemplo, `output ["x=\\"(x)\", y=\\"(y)"];` hace que se devuelvan variables `x` e `y` precedidas por su nombre (por ejemplo “`x = 5, y = 1`”).

Para mayor claridad, en la Figura 4.1 se muestra un ejemplo ilustrativo de un PSR y, seguidamente, dicho problema modelado en MiniZinc.

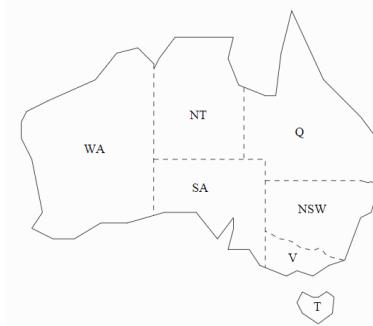


Figura 4.1: Problema del coloreado de regiones de Australia [23]. El objetivo es colorear el mala utilizando n colores, de forma que las regiones adyacentes estén siempre coloreadas de diferente color.

```

1 % Colouring Australia using nc colours
2 int: nc = 3;
3
4 var 1..nc: wa;    var 1..nc: nt;    var 1..nc: sa;    var 1..nc: q;
5 var 1..nc: nsw;   var 1..nc: v;     var 1..nc: t;
6
7 constraint wa != nt;
8 constraint wa != sa;
9 constraint nt != sa;
10 constraint nt != q;
11 constraint sa != q;
12 constraint sa != nsw;
13 constraint sa != v;
14 constraint q != nsw;
15 constraint nsw != v;
16 solve satisfy;
17
18 output ["wa=\(wa)\t nt=\(nt)\t sa=\(sa)\n",
19      "q=\(q)\t nsw=\(nsw)\t v=\(v)\n",
20      "t=", show(t), "\n"];

```

4.2.1.2. El Compilador de MiniZinc

El compilador puede ser llamado mediante el entorno de desarrollo incluido en el propio software o desde terminal, especificando el archivo de modelo y el resolutor a utilizar. Desde terminal, su uso es interesante porque permite hacer la asignación de valores a los parámetros mediante un archivo de datos externo (con la extensión .dzn). Así pues, se puede utilizar un archivo de modelo con parámetros sin definir y añadir, *a posteriori*, el archivo de datos que las defina. Para exemplificar su uso, se resuelve el problema mostrado en la Figura 4.1. El archivo de modelo es análogo, pero con el número de colores declarado y sin asignar (*int : nc;*). Para la asignación, el archivo de datos utilizado es el siguiente:

```
1 nc = 3;
```

Si se ejecuta el compilador con estos dos archivos y el resolutor Gecode, se obtiene el siguiente resultado:

```

1 wa=3      nt=2      sa=1
2 q=3       nsw=2     v=3
3 t=1
4 -----

```

4.3. Modelado y Resolución de Problemas de Planificación Automática

Para la Planificación Automática, sí que hay un lenguaje estándar *de facto*, aunque también existen algunas alternativas, sucesores o extensiones. Este es Planning Domain Definition Language (PDDL) [10].

4.3.1. Planning Domain Definition Language

En este lenguaje de propósito general, un problema queda definido por dos archivos: el archivo de dominio y el archivo de problema, los cuales se explican en las siguientes secciones.

4.3.1.1. El Archivo de Dominio

En él se definen los tipos de objetos que puede haber en el problema, así como los predicados en los que pueden verse contenidos. Además, se definen todas las acciones con sus respectivas precondiciones y postcondiciones. A continuación se muestran los principales componentes que suele contener un Dominio PDDL [24] (lista completa en <https://planning.wiki/ref/pddl/domain>):

- **Nombre de dominio**, denotado por `domain`. En él se da nombre al dominio que estamos definiendo.
- **Requisitos**, denotado por `:requirements`. En él se especifican las capacidades que un planificador deberá tener para poder resolver un problema en este dominio. En la práctica, es similar al “import” o “include” en otros lenguajes. La lista completa de requisitos que pueden especificarse se encuentra también en la página web anteriormente citada [24].
- **Tipos**, denotados por `:types`. En él se especifican los diferentes tipos de objetos que se pueden crear. Estos se pueden jerarquizar, creando a su vez subtipos de un tipo.
- **Constantes**, denotados por `:constants`. En él se pueden declarar objetos que están presentes en todos los problemas que se vayan a modelar bajo este dominio.
- **Predicados**, denotados por `:predicates`. En él se definen todas las propiedades que puede tener un objeto o las relaciones que puede haber entre varios de ellos. Es importante denotar que aquí se definen todas las posibles propiedades y relaciones, de forma que ni en las acciones

que veremos a continuación, ni en el archivo de problema, se puede hacer referencia a predicados que no estén declarados en este apartado.

Para añadir claridad, algunos ejemplos de predicados serían: (*encendido ?x - interruptor*), para indicar que un objeto de tipo interruptor puede estar encendido o (*padre ?x ?y - persona*) para indicar que una persona ?x es puede ser padre de una persona ?y.

- **Acciones**, denotadas por :**actions**. En él se definen todas las acciones que el planificador puede utilizar en este dominio. Cada una de estas, a su vez, queda definida por 3 apartados:
 - **Parámetros**, denotados por :**parameters**. Es el conjunto de objetos que intervienen en la acción, indicando el tipo al que ha de pertenecer cada uno.
 - **Precondiciones**, denotadas por :**preconditions**. Es el conjunto de predicados (o de expresiones lógicas entre estos) que debe cumplirse para que se pueda ejecutar la acción.
 - **Postcondiciones**, denotados por :**effects**. Es, en realidad, el conjunto de efectos que tiene la ejecución de la acción sobre el estado en el que se encuentra. Esto se expresa como un conjunto de predicados que son creados o eliminados.

4.3.1.2. El Archivo de Problema

En él se define el problema concreto a resolver, indicando el estado inicial y el objetivo, ambos en forma de predicados. A continuación se muestran los componentes que definen un Problema PDDL [25]:

- **Dominio**, denotado por **:domain**. En él, se indica el nombre del dominio bajo el que se define el problema.
- **Objetos**, denotado por **:objects**. En él, se listan los objetos concretos que existen en el ámbito del problema, con la posibilidad de especificarles un tipo.
- **Estado inicial**, denotado por **:init**. En él se listan los predicados que conforman el estado inicial del problema.
- **Objetivo**, denotado por **:goal**. En él se describe el objetivo a alcanzar. Se representa como un conjunto de predicados (o de expresiones lógicas entre estos) que debe cumplirse para que el objetivo se considere alcanzado.

Para mayor claridad, se añade un ejemplo ilustrativo de su uso, para resolver el problema que se muestra en la Figura 4.2. Primero se define el archivo de dominio. Para ello, se añade toda la información general (nombre, requerimientos, tipos, constantes y predicados) y a continuación todas las acciones que se pueden realizar (como estas siguen siempre la misma estructura, solo se muestra una al completo):

```

1  (define (domain mono)
2      (:requirements :strips :typing :negative-preconditions)
3      (:types
4          movible localizacion - object
5          mono caja - movible
6      )
7      (:constants
8      )
9      (:predicates
10         (en ?obj - movible ?x - localizacion)
11         (tienePlatano ?m - mono)
12         (sobre ?m - mono ?c - caja)
13         (platanoEn ?x - localizacion)
14     )
15
16     (:action moverCaja
17         :parameters (?m - mono ?c - caja ?x ?y - localizacion)
18         :precondition
19         (and
20             (en ?m ?x)
21             (en ?c ?x)
22             (not (sobre ?m ?c))
23         )
24         :effect
25         (and
26             (en ?m ?y)
27             (not (en ?m ?x))
28             (en ?c ?y)
29             (not (en ?c ?y))
30         )
31     )
32
33     (:action cogerPlatano
34         ...
35     )
36
37     (:action desplazarse
38         ...
39     )
40
41     (:action subir
42         ...
43     )
44
45     (:action bajar
46         ...
47     )
48 )

```

En este caso no se utilizan constantes, por lo que ese apartado queda vacío. Por último, se define el archivo de problema:

```

1 (define (problem monosp1)
2   (:domain mono)
3   (:objects
4     mono1 - mono
5     caja1 - caja
6     localizacion1 localizacion2 localizacion3 - localizacion
7   )
8   (:init
9     (en caja1 localizacion3)
10    (platanoEn localizacion2)
11    (en mono1 localizacion1)
12  )
13   (:goal
14     (and
15       (tienePlatano mono1)
16     )
17   )
18 )

```

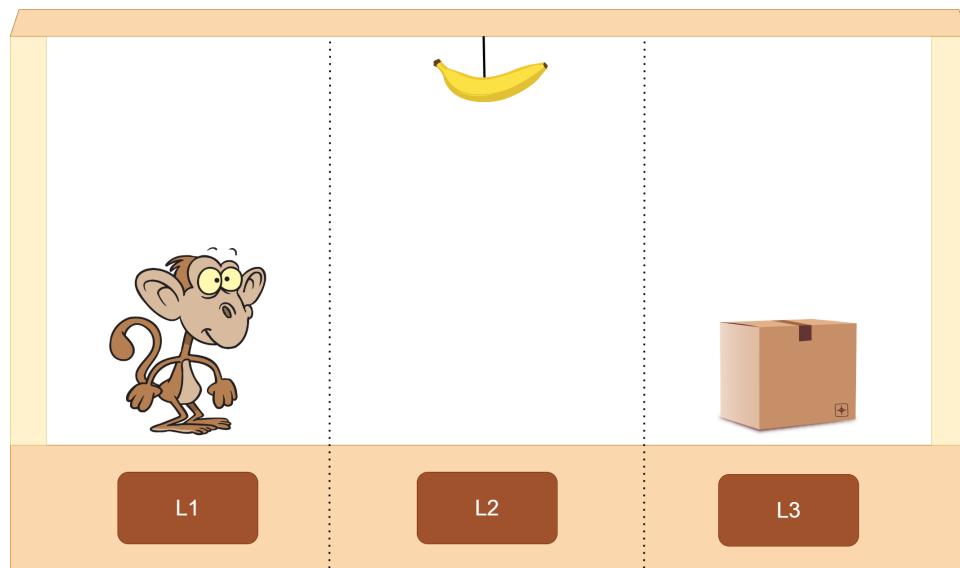


Figura 4.2: Problema del mono y el plátano. En la localización 1 (L1) hay un mono, en la localización 2 (L2) un plátano y en la localización 3 (L3) una caja. El objetivo del mono es obtener el plátano, y para conseguirlo sus posibles acciones son: desplazarse, mover la caja, subirse a la caja, bajarse de la caja y coger el plátano, todas con sus correspondientes restricciones (por ejemplo, solo el mono solo puede mover la caja si ambos están en la misma localización).

4.3.2. Planificadores

En cuanto a planificadores, hay gran variedad, cada uno con su propia estrategia y heurística. Para utilizarlos, será necesario un programa que haga de interfaz entre los archivos de dominio y problema PDDL y el planificador. Por ello, estos están generalmente integrados en su correspondiente sistema de planificación, un programa que recibe los archivos PDDL, los interpreta, planifica y devuelve la salida correspondiente.

En <https://planning.wiki/ref/planners/atoz> encontramos una extensa lista de sistemas de planificación. Algunos ejemplos destacables son planning.domain [26], el utilizado por *GVGAI-PDDL*, ENHSP [27], Fast Downward [28] y Metric-FF [12], que ejecuta el planificador FF (Fast-Forward) [11]. Este último es el sistema de planificación utilizado para este proyecto.

Siguiendo el ejemplo anterior, tras ejecutar el Metric-FF con los archivos de problema y dominio mostrados, se obtiene el siguiente plan:

```

1   0: DESPLAZARSE MONO1 LOCALIZACION1 LOCALIZACION3
2   1: MOVERCAJA MONO1 CAJA1 LOCALIZACION3 LOCALIZACION2
3   2: SUBIR MONO1 CAJA1 LOCALIZACION2
4   3: COGERPLATANO MONO1 CAJA1 LOCALIZACION2

```

Esta es la lista de acciones que deberían realizarse para que el mono termine cogiendo el plátano.

4.4. Algoritmos de Búsqueda Heurística

Los algoritmos de búsqueda heurística, en lo referido al uso en agentes inteligentes, se pueden dividir en dos grandes categoría: búsqueda offline (generan un plan en la primera llamada que el agente les realiza y se sigue ciegamente a lo largo de todas las demás) y online (generan varios planes a lo largo de las múltiples llamadas que el agente les realiza). Estos últimos se pueden dividir, a su vez, en algoritmos de búsqueda incremental y algoritmos de búsqueda en tiempo real. Cada uno de estos grupos se explica con más detalle en las secciones siguientes.

4.4.1. Búsqueda Offline

Los algoritmos de búsqueda offline generan un plan para llegar hasta el nodo objetivo, y se espera que este plan sea completo, realizable y, generalmente, también óptimo. Para poder asegurar esto, no obstante, se necesita de información absoluta del entorno, lo cual en muchos casos no es posible. En este TFG, el algoritmo de búsqueda offline utilizado es el A*.

4.4.1.1. A*

El algoritmo A* [1, 29] es uno de los algoritmos de búsqueda más extendidos, y el padre de todos los que veremos a continuación. La propiedad más destacable de este algoritmo es que, bajo ciertas condiciones, puede asegurar haber encontrado “la mejor solución”, lo cual es especialmente interesante para resolver problemas en los que se busca optimizar la búsqueda.

Para encontrar “la mejor solución”, es necesario definir antes una métrica que permita comparar la calidad entre ellas. Esto se hace mediante la función $c(n, n')$, que dados dos nodos directamente conectados por una arista, devuelve el coste de desplazarse del nodo n al nodo n' . De esta forma, el objetivo es encontrar un camino desde el nodo inicial hasta un nodo objetivo, tal que que minimice la sumatoria de los costes de realizar todos los desplazamientos que contiene. Sabiendo esto, se puede demostrar que este algoritmo siempre obtiene el camino óptimo si aseguramos una heurística admisible (esto es, $h(n) \leq C(n)$ para todo n nodo, siendo $C(n)$ el coste mínimo real para alcanzar el objetivo desde el nodo n ; es decir, que la heurística nunca sobreestime el coste).

Una vez visto esto, se explica el funcionamiento de A*. Para ello, se definen a continuación tres funciones y dos listas que son utilizadas en su lógica:

- La **función heurística** $h(n)$, que ya se ha explicado en el Capítulo 3. Devolverá el valor heurístico del nodo n dado.
- La **función de coste** $g(n)$, que devolverá el coste que tiene ir desde el nodo inicial hasta el nodo n , por el mejor camino encontrado hasta el momento. Así, el objetivo O se simplifica a encontrar un camino tal que minimice $g(O)$. Este valor es inicialmente infinito para todos los nodo, y va disminuyendo conforme se encuentran mejores caminos para llegar hasta él.
- La **función de evaluación** $f(n)$. Es la función indica lo prometedor que es un nodo. A menor valor de $f(n)$, más probable es que el nodo pertenezca al camino final. En este caso, $f(n) = g(n) + h(n)$.
- La lista de **abiertos**. En tiempo de ejecución, contiene los nodos evaluados que están pendientes de expandir. Estos son los que, o bien han sido evaluados pero no expandidos, o bien han sido expandidos pero su valor de f ha disminuido desde la última vez que se expandió.
- La lista de **cerrados**. En tiempo de ejecución, contiene los nodos ya expandidos. Todo nodo tiene que pasar por *abiertos* para llegar a esta lista. Aun así, un nodo que se encuentre en *cerrados* puede volver a *abiertos*, si su valor de f disminuye en alguna evaluación.

A* prioriza la exploración de nodos con menor valor de f . Para ello, el funcionamiento del algoritmo es el siguiente:

1. Se empieza por el nodo inicial.
2. En cada iteración:
 - a) Se añade el nodo actual a *cerrados* y se expande (es decir, se obtienen sus sucesores).
 - b) Para cada sucesor n , si a través del nodo actual se encuentra un mejor camino a él, se actualiza su valor $g(n)$, se evalúa $f(n)$ y se añade a *abiertos*.
 - c) Se selecciona el nodo de menor valor $f(n)$ de la lista de abiertos, y se elimina.
3. El algoritmo termina cuando se llega a un nodo objetivo o cuando no quedan nodos en abiertos que expandir (en este caso, significa que no existe solución).

En el Listado 4.1 se muestra una descripción en pseudocódigo de A*; y en la Figura 4.3 se simula su ejecución, donde se muestra el comportamiento del algoritmo para resolver el problema mostrado en la Figura 3.4 del capítulo anterior. Adicionalmente, para contextualizar la búsqueda heurística en entornos más cercanos al videojuego implementado en este TFG, en la Figura 4.4 se muestra también un ejemplo visual en el que A* resuelve el problema de búsqueda de caminos presentado en la Figura 3.5 del capítulo anterior.

Listado 4.1: Pseudocódigo del algoritmo A*.

```

1 g : función que es inicialmente infinito para todo valor.
2 abiertos : lista con prioridad ordenada de menor a mayor valor de f.
3 g(inicial) := 0
4 abiertos := [inicial]
5 while !abiertos.empty() and abiertos.top() != objetivo:
6     actual := abiertos.pop()
7     foreach nodo in actual.sucesores():
8         g_aux := g(actual) + c(actual, nodo)
9         if g_aux < g(nodo) and abiertos.contains(nodo):
10             g(nodo) := g_aux
11             abiertos.update(nodo)
12             if g_aux < g(nodo) and cerrados.contains(nodo):
13                 g(nodo) := g_aux
14                 cerrado.remove(nodo)
15                 abiertos.add(nodo)
16             if !abiertos.contains(nodo) and !cerrados.contains(nodo):
17                 g(nodo) := g_aux
18                 abiertos.add(nodo)

```

Para obtener el camino, solo hay que partir del nodo inicial e, iterativamente, desplazarse hacia el nodo sucesor n' con menor valor de $g(n') + c(n, n')$, hasta llegar al objetivo.

It.	Actual	Sucesores	Abiertos	Cerrados
0			A(1)	
1	A	B(2), D(7)	B(2), D(7)	A(1)
2	B	C(7), A(3)	D(7), C(7)	A(1), B(2)
3	C	B(4), E(8)	D(7), E(8)	A(1), B(2), C(7)
4	D	A(7), E(7)	E(8), E(7)	A(1), B(2), C(7), D(8)
5	E	Fin	Coste: 7	Camino: A-D-E

Figura 4.3: Simulación de ejecución del algoritmo A* para resolver el problema mostrado en la Figura 3.4. Se empieza con el nodo inicial (A) en *abiertos*; y en cada iteración (It.) se extrae, como *actual*, el nodo de menor valor f de *abiertos* (en caso de empate, se elige alfabéticamente), se obtienen sus sucesores, y se actualizan las listas de *abiertos* y *cerrados*. Los nodos muestran, entre paréntesis, el valor f con el que se almacenan en la correspondiente lista. El algoritmo termina cuando se expande el nodo objetivo (E).

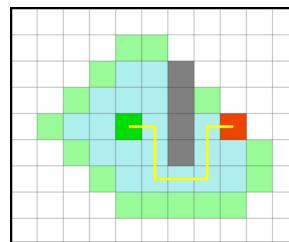


Figura 4.4: Ejemplo de ejecución del algoritmo A* para resolver el problema mostrado en la Figura 3.5. Las casillas de color celeste son conforman la lista de cerrados y las de verde agua conforman la de abiertos en el momento de finalizar la ejecución. La línea amarilla representa el camino encontrado.

4.4.2. Búsqueda Incremental

Para que un algoritmo de BH offline encuentre el camino, requiere información completa del entorno, pero hay 2 tipos de situaciones en las que esto no es posible, que son:

- **Entornos parcialmente conocidos**, donde el algoritmo solo dispone de información parcial sobre el estado del entorno. Es decir, existen valores de $c(n, n')$ desconocidos.
- **Entornos dinámicos**: donde la información que el algoritmo tiene sobre el entorno puede variar entre ejecuciones. Es decir, existen valores de $c(n, n')$ que pueden variar con el tiempo.

Ante estos casos, la solución más básica que se puede implementar es utilizar un algoritmo de búsqueda offline (A^* , por ejemplo) y, cada vez que su plan se vuelva imposible de continuar, generar un nuevo plan desde cero (esta es la versión de A^* implementada en este TFG). El problema de esto es que, en cada ejecución, hace un procesamiento completo de la evaluación de estados. Esto es un desperdicio de tiempo, pues se están recalculando valores de f para nodos que no se han visto afectados por el cambio. Para solucionar esto, nacen los algoritmos de búsqueda incremental, los cuales utilizan la información obtenida en ejecuciones anteriores para acelerar el proceso de búsqueda actual. El algoritmo de búsqueda incremental utilizado en este TFG es D^* Lite, pero para explicar su funcionamiento se introducen previamente LPA^* y D^* , pues son los algoritmos en los que este se basa.

4.4.2.1. LPA*

LPA^* (Lifelong Planning A*) [30] es una versión incremental de A^* . La idea principal es que, cuando se detecta un cambio en un nodo (esto es, que varía el coste de desplazarse hacia él), solo es necesario volver a explorar dicho nodo y aquellos a los que, consecuentemente, su cambio les afecte. El funcionamiento de este algoritmo es una adaptación y extensión del algoritmo A^* , con la misma base pero añadiendo las siguientes consideraciones:

- Todo nodo tiene una lista de predecesores, nodos que tienen una arista que lo conducen hasta él. Formalmente, n' pertenece a $predecesores(n)$ si existe un valor de $c(n', n)$.
- Todo nodo tiene una lista de sucesores, hasta los que se puede llegar desde él mediante una arista. Formalmente, n' pertenece a $sucesores(n)$ si existe un valor de $c(n, n')$.
- Se define una nueva función $rhs(n)$, que representa un valor anticipado de $g(n)$, calculado en base a sus predecesores: $rhs(n) = \min_{n'}(g(n') + c(n', n))$, con n' predecesor de n .
- Un nodo n se considera localmente consistente cuando $g(n) = rhs(n)$. Cuando todos los nodos son localmente consistentes, el camino encontrado por este algoritmo es equivalente al de A^* .
- Cuando un nodo n se vuelve localmente inconsistente (como consecuencia de un cambio coste de una arista), se añade a una cola con prioridad, análoga a *abiertos* de A^* , pero con la siguiente llave:

$$\begin{aligned} K(n) &= [K_1(n); K_2(n)] \\ K_1(n) &= \min(g(n), rhs(n)) + h(n) \\ K_2(n) &= \min(g(n), rhs(n)) \end{aligned}$$

Esto quiere decir que los nodos estarán ordenados de menor a mayor valor de K_1 y, en los casos de empate, se ordenarán de menor a mayor valor de K_2 .

- Cuando un nodo n se evalúa, se recalcula $rhs(n)$, lo que puede llevar a 3 situaciones:
 - Si el nodo era localmente inconsistente y se ha vuelto localmente consistente, se elimina de la lista.
 - Si el nodo era localmente consistente y se ha vuelto localmente inconsistente, se añade a la lista.
 - Si el nodo era localmente inconsistente y se mantiene así, pero el valor de su llave ha cambiado, se actualiza su posición en la lista.
- Cuando un nodo n se expande, se pueden dar tres casos:
 - $rhs(n) = g(n)$. El nodo es localmente consistente, por lo que se elimina de la cola.
 - $rhs(n) < g(n)$. Esto significa que se ha encontrado un padre a través del cual se mejora el valor de g . Se iguala $g(n)$ a $rhs(n)$, volviéndolo localmente consistente y se le elimina de la cola.
 - $rhs(n) > g(n)$. Esto significa que el valor de g que teníamos ya no es válido. $g(n)$ se establece en infinito y se actualiza su posición en la cola.

Como se puede ver, cuando el nodo es localmente inconsistente, se modifica su valor de $g(n)$. Esto puede, a su vez, afectar a la consistencia local de sus sucesores, de forma que todos ellos también deberán ser evaluados. Esta es la forma que tiene este algoritmo de propagar la inconsistencia cuando se genera.

- La condición de parada es que el nodo objetivo sea localmente consistente, y que el primer nodo de la cola de prioridad tenga un mayor valor $K(n)$ que el nodo objetivo.

Establecido todo esto, el funcionamiento de LPA* es el siguiente:

- **Bucle principal:** Es análogo al de A*. En cada iteración:
 1. Se expande el nodo actual.
 2. Se evalúan sus sucesores.
 3. Se selecciona el primer nodo de la lista con prioridad.
 4. Si se cumple la condición de parada, o no quedan nodos en la lista (en este caso, significa que no existe solución), el bucle termina.

- **Primera ejecución:** Este algoritmo no tiene conocimiento previo, por lo que se inicializa $rhs(inicial)$ a 0 y $g(inicial)$ a infinito. Para el resto de nodos, ambos valores son inicialmente infinito. De esta forma, el nodo inicial empieza siendo el único nodo localmente inconsistente, y la primera ejecución consistirá en propagar y solventar dicha inconsistencia.
- **Cambio de un coste:** Cuando se detecta una variación en un valor $c(n, n')$, se revalúa n' y se vuelve a ejecutar el bucle principal.
- Al igual que en A*, para obtener el camino hay que partir del nodo inicial e, iterativamente, desplazarse hacia el nodo sucesor n' con menor valor de $g(n') + c(n, n')$, hasta llegar al objetivo.

En el Listado 4.2 se puede ver una descripción en pseudocódigo del algoritmo LPA*.

Listado 4.2: Pseudocódigo del algoritmo LPA*.

```

1 g : funcion que es inicialmente infinito para todo valor.
2 rhs : funcion que es inicialmente infinito para todo valor.
3 Inconsistentes : lista con prioridad ordenada de menor a mayor key.
4 rhs(inicial) = 0
5 CalcularKey(inicial)
6 inconsistentes.add(inicial)
7 CalcularCamino()
8 Cuando se modifica un un valor c(n,n'):
9     Actualizar(n')
10    CalcularCamino()

11
12 CalcularCamino():
13     CalcularKey(objetivo)
14     while Inconsistentes.Top().key < objetivo.key
15     OR rhs(objetivo) != g(objetivo)
16         actual = U.Pop()
17         if g(actual) > rhs(actual):
18             g(actual) = rhs(actual)
19         else:
20             g(actual) = infinito
21             Actualizar(actual)
22             for all sucesor in sucesores(actual):
23                 Actualizar(sucesor)

24 Actualizar(n):
25     if Inconsistentes.contains(n):
26         Inconsistentes.remove(n)
27     if n != inicial:
28         rhs(n) = min(g(n')+c(n',n)) for n' in predecesores(n)
29     if g(n) != rhs(n):
30         CalcularKey(n)
31         Inconsistentes.add(n)

32 CalcularKey(n):
33     n.key = [min(g(n),rhs(n)) + h(n); min(g(n),rhs(n))]
```

4.4.2.2. D*

D* [31] sigue la filosofía de búsqueda de A* pero cambia el enfoque, partiendo del nodo objetivo y buscando el camino hacia el nodo inicial. Esto nos la siguiente ventaja a la hora utilizarlo en agentes, donde es común recalcular una ruta desde un nuevo nodo: En A*, como todos los valores de g se han calculado a partir del nodo inicial, al recalcularse desde otro nodo, los valores de g calculados ya no son fidedignos, al no representar el coste real de ir desde el nodo actual hasta el nodo dado. En cambio, en D*, como g se calcula en referencia al nodo objetivo, su información seguirá siendo acertada mientras este no varíe.

Además, D*, al igual que LPA* aprovecha la información que ya se calculó en anteriores ejecuciones. De hecho, D* y LPA* son algoritmos similares, siendo su comportamiento prácticamente equivalente pero cambiando el punto de referencia (LPA* empieza la exploración desde el nodo inicial y D* desde el objetivo). No obstante, como nacieron de forma independiente, su implementación es muy diferente. En este documento no se entra en detalle en su implementación ya que no será necesaria para implementar D* Lite.

4.4.2.3. D* Lite

D* Lite [32] es un algoritmo basado en LPA*, pero cuyo comportamiento es equivalente al de D*. Conceptualmente, este algoritmo consiste en utilizar LPA* pero cambiando el punto de referencia, expandiendo desde el objetivo en vez del nodo inicial. En consecuencia, D* Lite se comporta exactamente igual que D*, pero su implementación resulta más intuitiva y sencilla, y se implementa en menos líneas de código. Además, D* Lite tiene mejor rendimiento que el D* original. En los Listados 4.3 y 4.4, se muestra una descripción en pseudocódigo del algoritmo D* Lite; y en la Figura 4.5 se simula su ejecución, donde se muestra el comportamiento del algoritmo para resolver el problema mostrado en la Figura 3.4 del capítulo anterior.

Listado 4.3: Pseudocódigo del funcionamiento principal algoritmo D* Lite.

```

1 g : función que es inicialmente infinito para todo valor.
2 rhs : función que es inicialmente infinito para todo valor.
3 Inconsistentes : lista con prioridad ordenada de menor a mayor key.
4 rhs(Objetivo) = 0
5 CalcularKey(Objetivo)
6 inconsistentes.add(Objetivo)
7 CalcularCamino()
8 Cuando se modifica un valor c(n,n'):
9     Actualizar(n')
10    CalcularCamino()
```

Listado 4.4: Pseudocódigo de las funciones utilizadas por D* Lite.

```

1 CalcularCamino():
2     CalcularKey(inicial)
3     while Inconsistentes.Top().key < inicial.key
4     OR rhs(inicial) != g(inicial):
5         actual = Inconsistentes.Pop()
6         if g(actual) > rhs(actual):
7             g(actual) = rhs(actual)
8         else:
9             g(actual) = infinito
10            Actualizar(actual)
11            for all predecesor in predecesores(actual):
12                Actualizar(predecesor)
13
14 Actualizar(n):
15     if Inconsistentes.contains(n): Inconsistentes.remove(n)
16     if n != objetivo:
17         rhs(n) = min(g(n')+c(n,n')) for n' in sucesores(n)
18     if g(n) != rhs(n):
19         CalcularKey(n)
20         Inconsistentes.add(n)
21
22 CalcularKey(n):
23     n.key = [min(g(n),rhs(n)) + h(n); min(g(n),rhs(n))]
```

It.	Actual	Sucesores	Inconsistentes	g(A); rhs(A)	g(B); rhs(B)	g(C); rhs(C)	g(D); rhs(D)	g(E); rhs(E)
0			E(0;0)	$\infty; \infty$	$\infty; \infty$	$\infty; \infty$	$\infty; \infty$	$\infty; 0$
1	E	C, D	-	=	=	=	=	0; 0
	C	B, E	C(11;6)	=	=	$\infty; 6$	=	=
	D	A, E	C(11;6), D(8;4)	=	=	=	$\infty; 4$	=
2	D	A, E	C(11;6)	=	=	=	4; 4	=
	A	B, D	C(11;6), A(8;7)	$\infty; 7$	=	=	=	=
	E	No se actualiza rhs del nodo objetivo		=	=	=	=	=
3	A	B, D	C(11;6)	7; 7	=	=	=	=
	B	A, C	C(11;6), B(9;8)	=	$\infty; 8$	=	=	=
	D	A, E	=	=	=	=	=	=
4	B	Fin: A es consistente y A.Key < B.Key		Caminos: A-D-E		Coste: 7		

Figura 4.5: Simulación de ejecución del algoritmo D* Lite para resolver el problema mostrado en la Figura 3.4. Se empieza con todos los valores $g(n)$ y $rhs(n)$ a infinito, excepto $rhs(\text{objetivo})$, que es igual a 0; y con el nodo objetivo (E) en *inconsistentes*. En cada iteración (It.), se extrae, como *actual*, al nodo con menor *Key* (en caso de empate, se elige alfabéticamente), se actualiza su valor de g y se exploran sus predecesores (en este ejemplo, $\text{predecesores}(n) = \text{sucesores}(n)$ para todo n , por lo que se utiliza la misma columna para ambos). En estas exploraciones (zonas marcadas de azul), se obtienen los sucesores del nodo explorado y se actualiza su valor *rhs* en función de estos. El algoritmo termina cuando se cumple la condición de parada del algoritmo. Los nodos en *inconsistentes* muestran, entre paréntesis, la *Key* con la que se han almacenado.

Adicionalmente, para visualizar el mecanismo de replanificación de D* Lite, se modifica una arista del ejemplo dado (Figuras 4.6) y se simula su comportamiento cuando detecta dicho cambio (Figura 4.7).

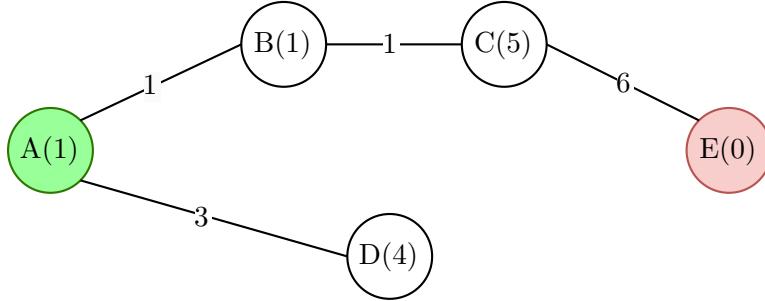


Figura 4.6: Ejemplo del problema de BH mostrado en la Figura 3.4, tras eliminar una arista. La arista que une D y E ha sido eliminada, por lo que los valores $c(D, E)$ y $c(E, D)$ se han eliminado.

It.	Actual	Sucesores	Inconsistentes	$g(A); \text{rhs}(A)$	$g(B); \text{rhs}(B)$	$g(C); \text{rhs}(C)$	$g(D); \text{rhs}(D)$	$g(E); \text{rhs}(E)$
0	E		No se actualiza rhs del nodo objetivo	=	=	=	=	=
	D	A	C(11;6), B(9;8), D(8;4)	=	=	=	4; 10	=
			C(11;6), B(9;8), D(8;4)	7; 7	∞ ; 8	∞ ; 6	4; 10	0; 0
1	D	A	C(11;6), B(9;8), D(8;4)	=	=	=	∞ ; 10	=
	A	B, D	C(11;6), B(9;8), D(14;10), A(8;7)	7; ∞	=	=	=	=
2	A	B, D	C(11;6), B(9;8), D(14;10)	∞ ; ∞	=	=	=	=
	B	A, C	C(11;6), B(9;8), D(14;10)	=	∞ ; ∞	=	=	=
	D	A	C(11;6), D(14;10)	=	=	=	∞ ; ∞	=
3	C	B, E	-	=	=	6; 6	=	=
	B	A, C	B(8;7)	=	∞ ; 7	=	=	=
	E		No se actualiza rhs del nodo objetivo	=	=	=	=	=
4	B	A, C	-	=	7; 7	=	=	=
	A	B, D	A(9;8)	∞ ; 8	=	=	=	=
	C	B, E	-	=	=	=	=	=
5	A	B, D	-	8; 8	=	=	=	=
	B	A, C	-	=	=	=	=	=
	D	A	D(15;11)	=	=	=	∞ ; 11	=
6	D	Fin: A es consistente y A.Key >= D.Key		Camino: A-B-C-E		Coste: 8		

Figura 4.7: Simulación de ejecución del algoritmo D* Lite cuando detecta un cambio en algún coste $c(n, n')$. En este caso, se ha modificado una arista del problema original, obteniendo así la Figura 4.6. Cuando esto ocurre, se revalúan los nodos (D y E) afectados (zona marcada de amarillo) y se vuelve a ejecutar el algoritmo principal. En la iteración 1, se propaga la inconsistencia generada, de D a A; en la iteración 2 se solucionan las sobreestimaciones de g en A y D; y en el resto de iteraciones, se busca el nuevo camino de forma análoga a la ejecución inicial.

4.4.3. Búsqueda en Tiempo Real

Los algoritmos vistos hasta ahora están diseñados para encontrar un plan al completo, invirtiendo todo el tiempo que sea necesario; pero, cuando el tiempo disponible por ejecución es muy bajo, es necesario utilizar algoritmos de búsqueda en tiempo real (BTR). Esta rama aúna aquellos algoritmos que se basan en resolver los problemas asegurando un bajo tiempo de respuesta. Se basan en la premisa de que, en cada ejecución, no es necesario calcular el plan al completo, sino solo la primera parte de este. Así, un algoritmo de este tipo sólo encontrará la solución tras múltiples llamadas, realizadas por un agente encargado de ejecutar las acciones que el algoritmo le devuelve. Todo algoritmo heurístico de BTR consta de 2 espacios (conjuntos de nodos) clave:

- **Espacio local de búsqueda:** Es el conjunto de nodos que se le permite explorar en cada ejecución. Sirve para acortar el tiempo de búsqueda, impidiendo que el algoritmo expanda nodos más allá de los límites impuestos. El tamaño del plan que es capaz de devolver el algoritmo está, pues, delimitado por el tamaño del espacio local de búsqueda.
- **Espacio local de aprendizaje:** Es el conjunto de nodos sobre los que, en cada ejecución, se actualiza la información que se tiene.

Como estos algoritmos están diseñados para ejecutarse múltiples veces, es necesario que en cada ejecución utilicen la información aprendida sobre las ejecuciones anteriores. Esta información se suele expresar mediante la función heurística, cuyos valores son actualizados a lo largo de las ejecuciones. La forma en que se modifica la heurística de los nodos se denomina regla de aprendizaje.

Por tanto, el espacio local de aprendizaje es el conjunto de nodos a los que se les modifica la función heurística en cada ejecución.

4.4.3.1. LRTA*

LRTA* [33] es una versión en tiempo real de A*. En este algoritmo, cada llamada solo devuelve la mejor acción a corto plazo, y es tras múltiples ejecuciones que eventualmente se alcanza el objetivo. En cada iteración observa cuál es el sucesor de menor $h(n)$ y le indica al agente que se mueva hacia él. Para converger hacia el objetivo y no terminar atascado en mínimos locales, LRTA* va aumentando el valor $h(n)$ de los nodos conforme los explora. Esto se hace actualizándolo en base al mejor nodo encontrado. La estructura de cada ejecución es la siguiente:

1. Por cada sucesor n' del nodo actual n : $f(n') = c(n, n') + h(n')$
2. Almacenar el nuevo $h(n)$: $h(n) = \min_{n'} f(n')$
3. Moverse al sucesor n' con menor $f(n')$

Así pues, en este algoritmo el espacio local de búsqueda es el conjunto de sucesores del nodo actual, y el espacio local de aprendizaje es, únicamente, el nodo actual. En el Listado 4.6 se muestra una descripción en pseudocódigo de una ejecución de este algoritmo, y en la Figura 4.8 se simula su ejecución, donde se muestra el comportamiento del algoritmo para resolver el problema mostrado en la Figura 3.4 del capítulo anterior.

Listado 4.5: Pseudocódigo del algoritmo LRTA*.

```

1 sucesor = min(c(actual, n')+h(n')) for n' in sucesores(n)
2 if h(n) < c(n, sucesor) + h(sucesor):
3     h(n) = c(n, sucesor) + h(sucesor)
4 return sucesor

```

Ej.	Actual	Sucesores	Sucesor	$h(A)$	$h(B)$	$h(C)$	$h(D)$	$h(E)$
0				1	1	4	5	0
1	A	B(2), D(8)	B	2	=	=	=	=
2	B	A(3), C(6)	A	=	3	=	=	=
3	A	B(4), D(8)	B	4	=	=	=	=
4	B	A(5), C(6)	A	=	5	=	=	=
5	A	B(6), D(8)	B	6	=	=	=	=
6	B	A(7), C(6)	C	=	6	=	=	=
7	C	B(7), E(6)	E	=	=	6	=	=
8	E	Fin		Coste Total: 12				

Figura 4.8: Simulación de ejecución del algoritmo LRTA* para resolver el problema mostrado en la Figura 3.4. Se inicializan todos los valores $h(n)$ según la heurística dada; y en cada ejecución (Ej.) se obtienen los sucesores del nodo actual, se actualiza $h(actual)$ en función de estos, y se elige el sucesor al que desplazarse en esa ejecución, que será el que menor valor $c(actual, n) + h(n)$ tenga (en caso de empate, se elige alfabéticamente). Los sucesores muestran, entre paréntesis, dicho valor. El algoritmo termina cuando se llega al nodo objetivo (E).

La característica más destacable de LRTA* es que, si la heurística es admisible, converge a la solución óptima, por lo que, si se utiliza múltiples veces para resolver el mismo problema, manteniendo lo aprendido entre usos, cada vez lo hará de forma más eficiente y, eventualmente, encontrará el camino óptimo.

4.4.3.2. RTA*

RTA* [33] es casi idéntico a LRTA*, con la diferencia de que el valor $h(n)$ es actualizado en base al segundo mejor nodo encontrado, no en base al mejor. En el Listado 4.6 se muestra una descripción en pseudocódigo de una ejecución de este algoritmo, y en la Figura 4.9 se simula su ejecución, donde se muestra el comportamiento del algoritmo para resolver el problema mostrado en la Figura 3.4 del capítulo anterior.

Listado 4.6: Pseudocódigo del algoritmo LRTA*.

```

1 sucesor = secondmin(c(actual, n') + h(n')) for n' in sucesores(n)
2 if h(n) < c(n, sucesor) + h(sucesor):
3     h(n) = c(n, sucesor) + h(sucesor)
4 return sucesor

```

Ej.	Actual	Sucesores	Sucesor	$h(A)$	$h(B)$	$h(C)$	$h(D)$	$h(E)$
0				1	1	4	5	0
1	A	B(2), D(7)	B	7	=	=	=	=
2	B	A(8), C(6)	C	=	8	=	=	=
3	C	B(9), E(6)	E	=	=	9	=	=
4	E	Fin	Coste Total: 8					

Figura 4.9: Simulación de ejecución del algoritmo RTA* para resolver el problema mostrado en la Figura 3.4. Se inicializan todos los valores $h(n)$ según la heurística dada. En cada ejecución (Ej.), se obtienen los sucesores del nodo actual, se actualiza $h(actual)$ en función de estos, y se elige el sucesor al que desplazarse en esa ejecución, que será el que menor valor $c(actual, n) + h(n)$ tenga (en caso de empate, se elige alfabéticamente). Los sucesores muestran, entre paréntesis, dicho valor. El algoritmo termina cuando se llega al nodo objetivo (E).

La ventaja de RTA* frente a LRTA* es que escapa de los mínimos locales con mayor rapidez, por lo que en general llega al objetivo más rápido. Esto se puede apreciar comparando las Figuras 4.8 y 4.9, donde la lógica de ambos algoritmos es la misma pero RTA* tarda la mitad de ejecuciones en encontrar la solución, pues LRTA* se queda temporalmente atascado en el mínimo local que hay entre los nodos A y B (ejecuciones 1-4). La desventaja de RTA* es que puede sobreestimar la heurística (puede haber nodos a los que les haya establecido un valor $h(n)$ mayor al coste real), por lo que no converge necesariamente a la solución óptima.

4.4.3.3. LRTA*(k)

LRTA*(k) [34] es una extensión de LRTA* que busca aumentar su rendimiento a costa de hacer más procesado por ejecución. La idea es que hay casos donde A* es demasiado lento, pero LRTA* es innecesariamente rápido (toma la decisión lo más rápido posible, expandiendo solo el nodo actual). Así pues, LRTA*(k) extiende este algoritmo de forma que se expandan nodos hasta un máximo de k veces. De esta forma, el algoritmo LRTA* se puede ver como un caso concreto de LRTA*(k) con $k = 1$.

En el Listado 4.7 se muestra una descripción en pseudocódigo de una ejecución de este algoritmo, y en la Figura 4.10 se simula su ejecución para $k = 2$, donde se muestra el comportamiento del algoritmo a la hora de resolver el problema mostrado en la Figura 3.4 del capítulo anterior.

Listado 4.7: Pseudocódigo del algoritmo LRTA*(k).

```

1 Candidatos := {actual}
2 contador = k-1
3 while !Candidatos.empty():
4     n := Candidatos.pop()
5     if Actualiza1(n):
6         for each sucesor in sucesores(n):
7             if contados > 0:
8                 Candidatos.add(n)
9                 contador = contador - 1
10    sucesor = min(c(actual, n')+h(n')) for n' in sucesores(actual)
11    return sucesor
12
13 Actualiza1(n):
14     sucesor = min(c(n, n')+h(n')) for n' in sucesores(n)
15     if h(n) < c(n, sucesor) + h(sucesor):
16         h(n) = c(n, sucesor) + h(sucesor)
17         return true
18     else: return false

```

Como se puede ver comparando la Figura 4.10 con la Figura 4.8, con aumentar k de 1 a 2, se consigue reducir el número de ejecuciones que se emplean en llegar al objetivo, pues al aumentar el número de expansiones por ejecución se consigue salir antes del mínimo local existente entre los nodos A y B.

Ej.	It.	Actual	Sucesores	Candidatos	$h(A)$	$h(B)$	$h(C)$	$h(D)$	$h(E)$
0					1	1	4	5	0
1	A		A						
	1	A	B(2), D(8)	B, D	2	=	=	=	=
	2	B	A(3), C(6)	D, C	=	3	=	=	=
		A	B(4), D(8)	Sucesor:	B				
2	B		B						
	1	B	A(3), C(6)	A, C	=	=	=	=	=
	2	A	B(4), D(8)	C, D	4	=	=	=	=
		B	A(5), C(5)	Sucesor:	A				
3	A		A						
	1	A	B(4), D(8)	B, D	=	=	=	=	=
	2	B	A(5), C(6)	D, C	=	5	=	=	=
		A	B(6), D(6)	Sucesor:	B				
4	B		B						
	1	B	A(5), C(6)	A, C	=	=	=	=	=
	2	A	B(6), D(8)	C, D	6	=	=	=	=
		B	A(7), C(6)	Sucesor:	C				
5	C		C						
	1	C	B(6), E(6)	B, E	=	=	6	=	=
	2	B	A(7), C(7)	E, A	=	7	=	=	=
		C	B(8), E(6)	Sucesor:	E				
6	E	Fin	Coste Total: 10						

Figura 4.10: Simulación de ejecución del algoritmo LRTA*(k), con $k = 2$, para resolver el problema mostrado en la Figura 3.4. Se inicializan todos los valores $h(n)$ según la heurística dada. Cada ejecución (Ej.) empieza con el nodo actual en la lista de *candidatos*, y se hacen $k = 2$ iteraciones (It.) de expansiones. En cada iteración (zonas marcadas de azul), se extrae el primer nodo de *candidatos*, se obtienen sus sucesores, que se almacenan también en dicha lista, y se actualiza su valor de h en función de estos. Al finalizar las iteraciones, se elige el sucesor al que desplazarse en esa ejecución, que será el que menor valor $c(actual, n) + h(n)$ tenga (en caso de empate, se elige alfabéticamente). La lista de *sucesores* muestra siempre, entre paréntesis, dicho valor para cada nodo en ese momento. El algoritmo termina cuando se llega al nodo objetivo (E).

Capítulo 5

Diseño e implementación

5.1. Diseño

5.1.1. Videojuego: El Mochilero

El videojuego desarrollado se nominado como *El Mochilero*, nombre que proviene del problema de la mochila, tomado como referencia inicial para diseñar la lógica del juego. Este problema consiste en, dada una serie de objetos cada uno con un valor y un peso concretos, y una mochila donde estos se pueden guardar, pero con un peso máximo que puede almacenar, encontrar el conjunto de objetos que guardar tal que se maximice el valor conseguido. *El Mochilero* implementa este problema, al cual se le añaden dos dificultades principales: una capa de restricciones a la hora de obtener objetos y un mapa laberíntico por el que moverse.

El Mochilero, además, transcurre en un mundo parcialmente observable, lo que quiere decir que en cada momento el jugador solo conoce lo que hay en las casillas que estén dentro de la zona de visibilidad (ZVA) de su avatar. De esta forma, el problema no puede ser resuelto en una sola ejecución, y el agente inteligente que resuelva el problema necesita disponer de técnicas de replanificación (es decir, debe poder calcular un nuevo camino a seguir si es necesario). Esto solo afecta a la información relacionada con carácter laberíntico del mundo (es decir, los suelos y muros), por lo que los objetos y la salida son visibles en todo momento. Esto se ha decidido así para mantener la estructura de “Optimización → Planificación → Búsqueda” planteada. Si no se conocen inicialmente los objetos, es necesario emplear movimientos con único objetivo de localizar todos los objetos que hay en el mapa, lo cual sólo aumentaría la complejidad del problema saliéndose del marco de las técnicas que estamos poniendo a prueba. Para ilustrar esta característica, en la Figura 5.1 se muestra un ejemplo de cómo percibe el mundo el jugador.

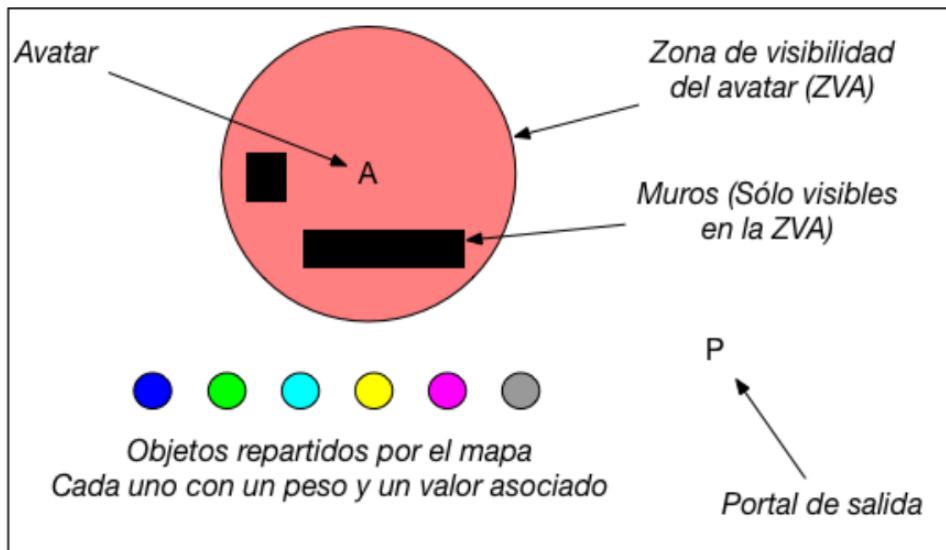


Figura 5.1: Ilustración que refleja, a nivel conceptual, cómo percibe el mundo el jugador. Este siempre conoce la posición de los objetos repartidos por el mapa (círculos de colores) y el portal de salida (P), pero solo conoce la posición de los muros (rectángulos negros) que se encuentran en la ZVA (espacio en rojo), que es un cierto área alrededor del avatar (A). Cuando el avatar se desplaza, la ZVA se mueve con él, revelando los nuevos muros que pueda haber por el mapa.

El juego consiste, pues, en un mapa laberíntico parcialmente observable en el que hay repartidos una serie de objetos, cada uno con un valor y un peso concretos; y un avatar que tiene una mochila que le permite llevarlos. El jugador puede usar el avatar para desplazarse por el mundo, coger los objetos y soltarlos a voluntad, pero con un peso máximo que puede almacenar al mismo tiempo. Para coger los recursos, no obstante, existe un orden de precedencia entre ellos, por lo que, para coger un objeto concreto, el jugador tiene que haber tenido en su poder en algún momento el objeto inmediatamente anterior (y por extensión, todos los anteriores). El objetivo del juego es salir del laberinto con el conjunto de recursos que mayor valor le proporcione, en el menor tiempo posible.

Para modular este comportamiento, el videojuego consta de las siguientes reglas concretas que lo definen:

1. Los objetos tienen un valor y un peso asociados, además un orden de precedencia fijo (el segundo requiere que antes se haya cogido el primero, el tercero que se haya cogido el segundo, etc).
2. El avatar tiene una ZVA a la hora de percibir el entorno. Inicialmente

sabrá dónde está la salida, así como los objetos y sus pesos, valores y órdenes de precedencia; pero no conoce el mapa y, obviamente, tampoco conoce el camino para llegar hasta ellos.

3. Una vez obtenido un objeto, ya siempre será posible obtener el siguiente en el orden de precedencia, sin necesidad de mantenerlo en la mochila.
4. El jugador tiene un peso máximo que es capaz de llevar al mismo tiempo.
5. El objetivo del juego es conseguir el subconjunto de objetos que mayor beneficio proporciona, sin exceder el peso disponible, y salir del laberinto en el menor tiempo posible.
6. El avatar siempre apunta hacia una dirección concreta, que será hacia donde se desplaza o donde se suelte un objeto, si lo indica el jugador.
7. En cada acción, el jugador puede modificar la dirección hacia la que apunta el avatar, desplazarlo hacia adelante (si se mueve hacia un objeto que cumple con las condiciones para obtenerlo, lo recogerá automáticamente) o soltar un objeto (siempre se suelta el primero, en el orden de procedencias, que el avatar tenga en su poder).

Este diseño tiene el objetivo de aprovechar al máximo las potencias del agente inteligente planteado:

- Mediante RR, encontrará el subconjunto de objetos que más convenga obtener (resolverá el problema de la mochila).
- Mediante PA, determinará que orden de acciones (coger / soltar objetos) habrá que llevar a cabo para obtener el subconjunto objetivo (resolverá el problema de precedencias).
- Mediante BH, obtendrá los objetos en el orden determinado, y posteriormente llegará hasta la salida (resolverá el laberinto en un entorno parcialmente observable).

5.1.2. Agente Inteligente

El agente inteligente diseñado se conforma, principalmente, por 3 grandes subsistemas o módulos, cada uno encargado del procesamiento de cada una de las técnicas a integrar. Para el RR, se utiliza MiniZinc [22], al ser versátil, simple, fácil de invocar por terminal y con adaptabilidad en las salidas. Para la PA se utiliza Metric-FF [12], principalmente al ser de los pocos

sistemas que permiten el uso del módulo *fluents*, que permite realizar operaciones numéricas dentro de la lógica a procesar, lo cual es extremadamente conveniente para llevar la cuenta del peso. La idea general de funcionamiento es la siguiente:

1. El **módulo MiniZinc** obtiene la información de los objetos que se encuentran en el mapa, así como sus pesos y valores, y el peso máximo que puede llevar el avatar. Con esa información calcula el subconjunto de objetos que se pueden llevar que maximice el valor total.
2. El **módulo PDDL** obtiene el objetivo del módulo anterior (este es, el subconjunto de objetos que se busca tener) y la información de los objetos que se encuentran en el mapa, así como sus pesos y precedencias. Con esta información, calcula el conjunto de acciones (coger o soltar un objeto) que hay que realizar, finalizando con la acción de salir.
3. El **módulo heurístico** obtiene la lista de acciones del módulo anterior, y las va completando consecutivamente (buscando el objeto indicado, soltándolo, o desplazándose hasta la salida, según la acción que se le indique).
4. El **controlador** es el software encargado de la lógica general del agente. Gestiona la interacción con cada uno de estos módulos y la toma de decisiones, en función del estado del entorno y del propio sistema.

Para que todo este sistema funcione, además, son necesarios 3 archivos informativos que el usuario debe proporcionar. Estos son los siguientes:

- El **archivo de modelo MiniZinc** del problema de optimización a resolver. Este se completará mediante un archivo de datos, generado automáticamente con la información específica del mapa.
- El **archivo de dominio PDDL**, que define el problema de predecesoras a resolver. El archivo de Problema PDDL se genera automáticamente con la información específica del mapa.
- El **archivo de configuración general**, que define toda la información extra necesaria para el procesamiento. Estos son, esencialmente, las rutas en las que se encuentran los archivos y las correspondencias entre un lenguaje y otro.

En la Figura 5.2 se puede ver la arquitectura general del sistema, con las relaciones entre los diferentes módulos.

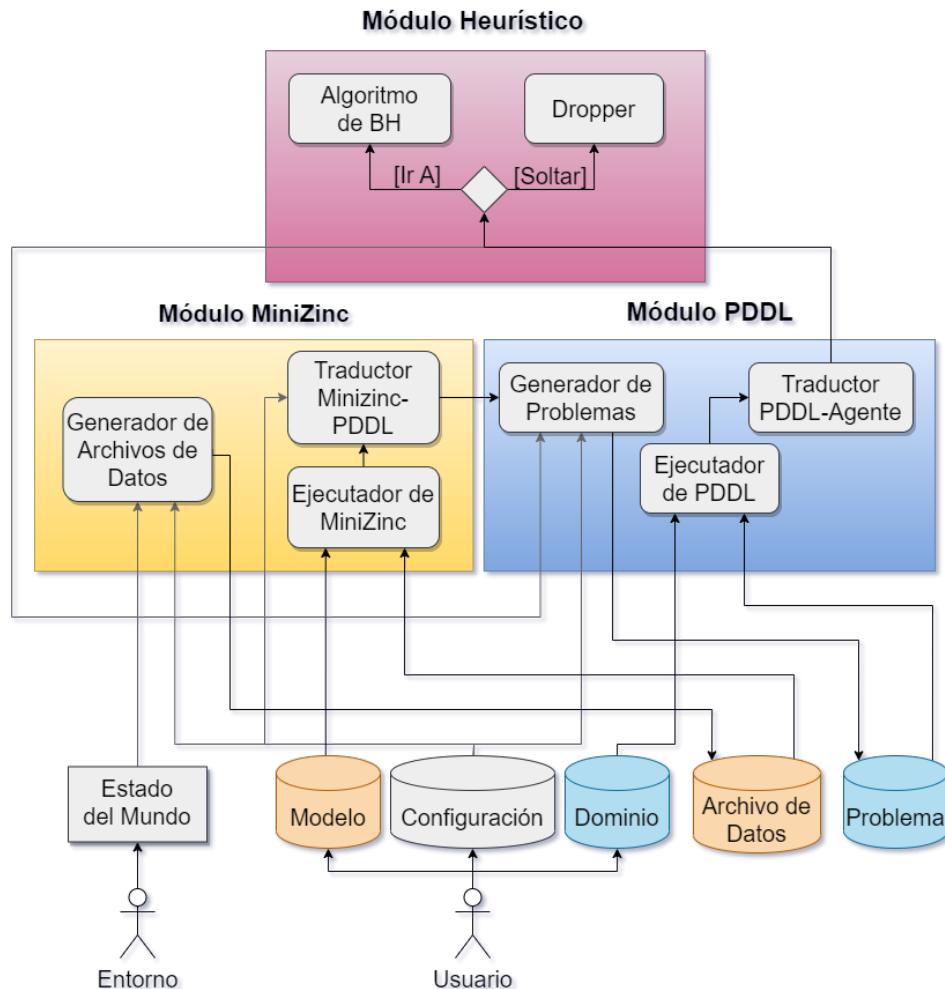


Figura 5.2: Arquitectura general del agente inteligente. El Módulo MiniZinc (Sección 5.1.2.1) resuelve el problema de RR. Para ello, el **Generador de Archivos de Datos** crea el archivo que el **Ejecutor de MiniZinc** utiliza para generar un objetivo que el **Traductor MiniZinc-PDDL** transforma al lenguaje PDDL. A continuación, el Módulo PDDL (Sección 5.1.2.1) resuelve el problema de PA. Para ello, el **Generador de Problemas** crea el archivo que el **Ejecutor de PDDL** utiliza para generar un plan que el **Traductor PDDL-Agente** transforma a una serie de objetivos parciales. Finalmente, el Módulo heurístico (Sección 5.1.2.3) recibe los objetivos parciales uno a uno, y los completa utilizando, en función del objetivo a alcanzar, bien el **Dropper**, un algoritmo simple, o bien el **Algoritmo de BH**. El entorno informa a los generadores del estado del mundo, y el usuario ha de proporcionar los archivos de modelo, dominio y configuración que se utilizan.

5.1.2.1. Módulo MiniZinc

La interacción con MiniZinc consiste en llamadas por terminal desde el entorno y el procesamiento de la información enviada y devuelta. Para ello, será necesario implementar las siguientes funcionalidades o bloques:

- **Generador de Archivos de Datos.** De la información que proporciona el entorno sobre estado inicial del mundo, se extrae la pertinente para resolver el problema de optimización, se transforma al formato de MiniZinc, y se almacena en un archivo de datos. La información sobre qué extraer del estado inicial, así como la transformación que realizarle para tener el formato adecuado, se encuentran especificadas en el archivo de configuración.
- **Ejecutador de MiniZinc.** Se llama a MiniZinc por terminal, proporcionándole el archivo de modelo creado por el usuario y el archivo de datos generado automáticamente. MiniZinc resuelve el problema y devuelve una salida que, a su vez, es lo que devuelve este bloque.
- **Traductor MiniZinc-PDDL.** La salida obtenida por el bloque anterior se transforma en predicados PDDL. La correspondencia entre la salida de MiniZinc y los predicados debe estar especificada en el archivo de configuración. La lista de predicados PDDL es la salida de este bloque, y también de todo este módulo.

5.1.2.2. Módulo PDDL

La interacción con PDDL consiste también en llamadas por terminal desde el entorno. De forma muy similar a MiniZinc, para este subsistema su funcionalidad se divide en 3 bloques. Estos son:

- **Generador de Problemas.** Será el encargado de generar el archivo de problema PDDL, el cual tendrá la siguiente estructura:
 - Para la inicialización (`:init`), se extrae la información necesaria de los datos que proporciona el entorno sobre estado inicial del juego. De nuevo, la información a extraer, así como su correspondencia en forma de predicados, se encuentra especificada en el archivo de configuración.
 - El objetivo (`:goal`) son los predicados generados por el modulo MiniZinc.
 - Los objetos (`:objects`) están vacíos.
 - El nombre del dominio (`:domain`) es también proporcionado por el archivo de configuración.

- **Ejecutador de PDDL.** Se llama a Metric-FF por terminal. Se le proporcionan el archivo de dominio generado por el usuario y el archivo de problema recién generado. De la información devuelta, se extrae con la parte del plan, que será la salida de este bloque.
- **Traductor PDDL-Agente.** El plan obtenido por el bloque anterior es transformado a un lenguaje que pueda entender el agente inteligente: las órdenes son traducidas a órdenes que entiende, y los objetos son traducidos a su correspondiente nombre en el juego, para ser fácilmente identificables. Estas correspondencias también han de ser especificadas en el archivo de configuración.

5.1.2.3. Módulo Heurístico

Una vez se tiene un plan PDDL, es decir, una serie de acciones concretas a realizar en orden, se le proporcionan una a una al agente, que actúa en función de la acción solicitada. Se diferencian, para este juego en concreto, en dos tipos (aunque podrían definirse todas las acciones que se necesitasen, aumentando así la inteligencia del agente):

- **Ir A:** Si el objetivo es alcanzar un recurso concreto, se llama a un algoritmo de BH que encuentre el camino hasta dicho objeto.
- **Soltar:** Si el objetivo es soltar un objeto, se llama a un algoritmo simple, llamado *Dropper* (*Soltador*), cuya funcionalidad es sencillamente soltar un recurso en una casilla válida.

Una vez alcanzado un objetivo, se extrae el siguiente del plan y se llama al algoritmo correspondiente para lograrlo. Este proceso se repite hasta que el plan se completa y el juego se acaba (ambas cosas deberían ocurrir a la vez, pues el último objetivo es siempre llegar a la salida).

5.2. Implementación

A continuación se expone en detalle de todo el software creado a lo largo del proyecto. Este utiliza el proyecto GVGAI-PDDL [21] como framework, de forma que se puede acceder a todas las características que GVGAI brinda, así como a su enfoque de la integración entre GVGAI y PDDL. Por tanto, como lenguaje de programación se utiliza principalmente Java (lenguaje en el que se encuentra dicho entorno), además de *Video Game Definition Language* (VGDL) [35], pues es el que se utiliza para la definición de videojuegos en GVGAI.

El desarrollo de este proyecto se divide en dos partes bien diferenciadas: el agente inteligente y el videojuego en el que se pondrá a prueba. Para mayor facilidad, se va a empezar por describir la creación de este último, y posteriormente se explicarán los detalles de implementación del agente.

5.2.1. Videojuego en VGDL

Para poder ejecutar un juego en VGDL, es necesario definir 2 archivos. Estos son el *Game Description File* (GDF) y el *Level Description File* (LDF), los cuales se explican en las siguientes secciones.

5.2.1.1. Game Description File

En el GDF se especifican todos los objetos del juego (a los que se les llama sprites), las reglas que se aplican cuando dos sprites colisionan (a las que se les llama interacciones) y las condiciones de terminación de este. En este archivo se definen la totalidad de las reglas del juego, basándose en el sistema de interacciones, que tienen siempre la siguiente estructura: cuando dos objetos concretos colisionan → Una regla concreta se ejecuta.

Un GDF se define mediante 4 secciones. A continuación, se explica la función de cada una de ellas, utilizando como ejemplo el videojuego desarrollado.

SpriteSet Es el conjunto de sprites que tendrá el juego. Para *El Mochilero* se utilizan:

- **Avatar.** Es el personaje sobre el que el jugador tiene control. Será de tipo *ShootAvatar*, lo que le concede la característica de generar un objeto cada vez que se pulsa el espacio o se ejecuta la acción *Use*. El avatar suelta el sprite *Dropper*, explicado a continuación. Con él, se consigue que el avatar se suelten los recursos al realizar esta acción.
- **Dropper.** Es el objeto que crea el avatar. Es un *Singleton*, lo que hace que no se puedan tener múltiples droppers a la vez, y es de tipo *Flicker*, lo que hace que al poco tiempo de crearlo se elimina automáticamente, para poder volverlo a invocar en otra casilla diferente. Su función es transformarse en el recurso pertinente cuando el avatar intente soltar un objeto.
- **Suelos.** Los habrá de 3 tipo en función de su estado de visibilidad: visible, desconocido y conocido (si se ha visto con anterioridad pero actualmente no se ve).

- **Paredes.** Ídem a los suelos.
- **Salida.** Al entrar en contacto con ella, se termina el nivel.
- **Recursos.** Habrá 6, todos *Singletones* (para cada recurso, solo puede existir uno a la vez), lo que ayudará a las interacciones.
- **Peso.** Es de tipo *Resource*, lo que permite que el avatar tenga un medidor con el peso que carga. Está limitado a 10.
- **Llaves.** Son las que permiten coger el recurso siguiente (es decir, para obtener el recurso 5 se ha de tener la llave 4). Siempre que se recoja un recurso por primera vez, se recoge también su llave, asegurando así el orden de precedencia.
- Para ayudar con las interacciones, se añaden dos entidades más: el **fondo**, que estará presente en todas las casillas y el recurso **nulo**, que nunca se encontrará presente en el mapa.

TerminationSet Es el conjunto de condiciones que finaliza la partida, también se puede indicar si el jugador ha ganado o perdido. Para *El Mochilero*, la partida finaliza cuando el jugador entra en contacto con la salida. Se considera que gana siempre que termina, al conseguir resolver el laberinto.

InteractionSet Es el conjunto de reglas que definen lo que ocurre cuando los sprites colisionan. En *El Mochilero* se utilizan tanto interacciones predefinidas como interacciones creadas (explicadas en la Sección 5.2.1.4), con el siguiente resultado:

- **Interacciones básicas:**

Cuando el avatar colisiona con una pared, vuelve atrás. Esto es lo que hace intransitables las paredes.

Cuando el *Dropper* colisiona con una pared o un recurso, es inmediatamente destruido, evitando que suelten recursos dentro de la propia pared o que se apilen en la misma cuadrícula.

- **Obtención de Recursos:**

Tanto un recurso como una llave se considerará que “está obtenido” cuando no está en el mapa. Así pues, cuando el avatar entra en contacto con un recurso, se modifica su peso y su puntuación y se destruye el recurso, simulando su recogida. Esta interacción sólo ocurre si no se sobrepasa el peso máximo y si se tiene la llave anterior.

Cuando se pasa por encima de una llave, esta se destruye si ya se tiene la anterior, simulando así haberla recogido y creando un orden de precedencias entre ellas.

Para el recurso 1, se considera que su llave es el sprite *nulo* que, como se ha dicho, nunca se encuentra en el mapa, por lo que se considera siempre obtenido.

- **Soltado de Recursos:**

Cuando el *Dropper* entra en contacto con el suelo, se intenta transformar, por orden, en los recursos (primero en el recurso 1, luego en el 2, etc), pero al ser *Singletones*, esto solo ocurrirá si dicho recurso no está en el mapa. Cuando se encuentra un recurso en el que es posible transformarse, ejecuta la interacción y se disminuyen los valores de peso y puntuación del avatar, acorde al recurso soltado.

- **Mundo Parcialmente Observable:**

Para generar un mundo parcialmente observable, se juega con los 3 tipos de paredes y suelos que existen. Cada vez que uno de estos entra en contacto con el fondo, se ejecutan las siguientes acciones (esto ocurre en cada instante de juego, por lo que, a efectos prácticos se están ejecutando continuamente):

- Los suelos y paredes desconocidos y conocidos se volverán visibles si están a 4 casillas o menos del avatar.
- Los suelos y paredes visibles se volverán conocidos si están a más de 4 casillas del avatar.

LevelMapping Es la sección que define la relación entre los los caracteres en el LDF y los sprites que se encuentran en esa casilla. Los definidos para el juego son:

- “w” para las paredes y “.” para los suelos. Inicialmente, todos son del tipo Desconocido.
- “e” para la salida y “A” para el avatar.
- Los números “1 al 6” para los recursos. En cada una de estas casillas se encuentra inicialmente tanto el recurso como su llave, además del suelo.

En todas estas casillas se añade, además, un sprite de tipo *fondo*.

5.2.1.2. Level Description File

En el LDF se define el estado en el que se encuentra el mapa cuando se inicia cierto nivel concreto. Se indica mediante una matriz 2D de símbolos, donde cada uno representa el conjunto concreto de objetos que se encuentran en esa casilla.

Los símbolos disponibles son los caracteres definidos en la sección *LevelMapping* del GDF correspondiente. De esta forma, una vez definido el funcionamiento del juego, se pueden crear fácilmente todos los niveles que se deseen. Para exemplificar su funcionamiento, en la Figura 5.3 se puede ver un ejemplo LDF, correspondiente al mapa mostrado en el Capítulo 1. Concretamente, en la Figura 1.1 de dicho capítulo se muestra el mapa completo de dicho nivel, representado en GVGAI, y en la Figura 1.2 lo que percibe el jugador a su inicio.

```
wwwwwwwwwwwwwwwwwwww
w.....4..w
w.2....5.....w
w.....wwwww
w...w.A....ww...3w
w...w.www.www.w.www
w...w.w.w.....www
w...w.w.www.ww.w.w
w.1.....ewww...6w
wwwwwwwwwwwwwwwwww
```

Figura 5.3: *Level Description File* del nivel 1 del videojuego *El Mochilero*. Consiste en una cuadrícula de caracteres, cada uno de ellos representando lo que será un elemento concreto en el juego una vez cargado: paredes (w), suelo (.), la salida (e), el avatar (A) y los recursos (números del 1 al 6).

5.2.1.3. Modelado de los Recursos

Para representar los diferentes recursos que se deben obtener, es necesario crear un modelo que informe al jugador, de forma rápida y sencilla, las 3 características de estos (la posición en la lista de precedencias, el peso y el valor). Para esto, se barajaron algunas opciones, como lingotes o monedas apiladas (Figura 5.4). Estas se descartaron por ser poco claras, difícilmente extensibles y nada intuitivas. En su lugar, y como este proyecto no tiene fines artísticos, se optó por la opción más sencilla: un cuadrado que informa de estos 3 valores en forma numérica (Figura 5.5). En la Figura 5.6 se pueden ver los modelos finales utilizados para representar los diferentes recursos. Para el resto de elementos se utilizarán sprites ya dados por el framework.

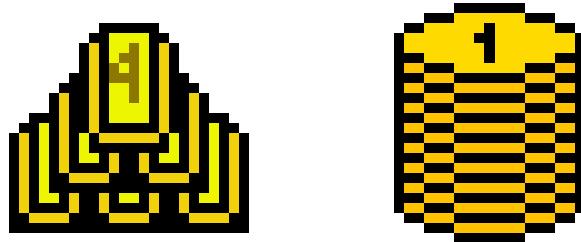


Figura 5.4: Ejemplo de representación de un recurso en forma de lingotes (imagen izquierda) y de monedas (imagen derecha). En ambos casos, el número escrito en la parte superior indica su posición en la lista de precedencias, el número de elementos (lingotes o monedas) su peso y el color su valor. Este último parámetro debía regirse por un código colores que representase materiales con una clara diferencia de valor.

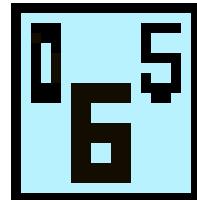


Figura 5.5: Ejemplo de representación final escogida para los recursos. El número central (6) indica la posición de este recurso en el orden de precedencias, el numero superior derecho (5) indica su valor y el número superior izquierdo (0) indica su peso. El color solo es un elemento diferenciador.

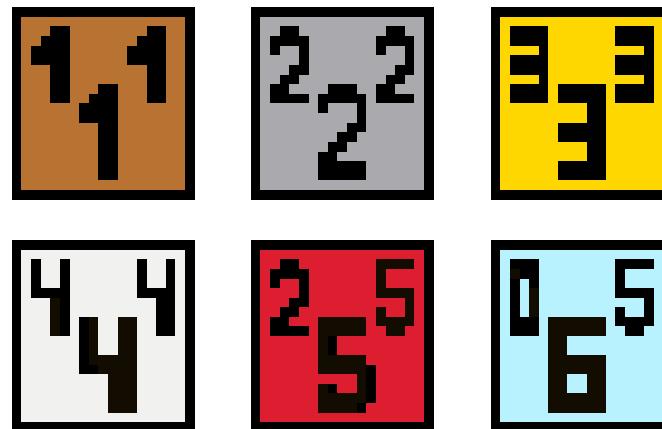


Figura 5.6: Conjunto de todos los modelos utilizados para representar los recursos en el videojuego *El Mochilero*.

5.2.1.4. Interacciones

El problema de VGDL es que las interacciones que proporciona el framework son muy limitadas para el juego que se busca crear. Por ello, fue necesaria la creación de nuevas interacciones. Estas están definidas de forma general, para que puedan ser utilizadas en el desarrollo de otros juegos. Aunque no todas han sido implementadas en la versión final de *El Mochilero*, a continuación se explican las interacciones añadidas:

- **ChangeResourceIfHeld.** Modifica, al primer sprite, el valor de recurso *resource* indicado en la interacción. Esto solo ocurre si no existe en el mapa el sprite *heldResource* y no se supera el valor máximo del recurso.

Modo de uso: *avatar recurso4 > changeResourceIfHeld heldResource=recurso3 resource=peso value=2 scoreChange=2* - Aumenta en 2 el valor de peso y la puntuación del avatar cuando colisiona con el recurso 4, solo si tiene el recurso 3.

- **TransformAndChangeResource.** Transforma el primer sprite al tipo *stype*, y modifica el valor del recurso *resource* al *avatar* indicado. Esta interacción solo ocurre si no se supera el valor máximo del recurso.

Modo de uso: *dropper floor > transformAndChangeResource stype=recurso1 avatar=avatar resource=peso value=-1 scoreChange=-1* - Cuando el *dropper* toca el suelo, se transforma en el recurso 1, y se reduce el valor de peso y la puntuación del avatar en 1.

- **RefreshVisionState.** Actualiza los estados de *invisibilidad* y *ocultación* del primer sprite, en función de su distancia con el *avatar* indicado. Si está demasiado lejos se actualizará a oculto e invisible, y si está cerca a visible y perceptible.

Modo de uso: *perceptible background > refreshVisionState avatar=avatar range=4* - Si un objeto perceptible colisiona con el fondo (esencialmente, en cada tick), se actualiza su estado de visión en función de su estar a más de 4 casillas del avatar o no.

- **TransformIfAvatarNear.** Transforma el primer sprite al tipo *stype*, si el *avatar* indicado está a un rango menor o igual a *range*.

Modo de uso: *unknownFloor background > transformIfAvatarNear avatar=avatar range=4 stype=visibleFloor* - El suelo desconocido se vuelve visible si se encuentra a 4 casillas o menos del avatar.

- **TransformIfAvatarFar.** Transforma el primer sprite al tipo *stype*, si el *avatar* indicado está a un rango mayor a *range*.

Modo de uso: *visibleFloor background > transformIfAvatarFar avatar=avatar range=4 stype=unknownFloor* - El suelo visible se vuelve desconocido si se encuentra a más de 4 casillas del jugador.

- **ExchangeResourceIfHeld.** Transforma el primer sprite al tipo *stype* y modifica el valor del recurso *resource* al *avatar* indicado. Esta interacción solo ocurre si no existe en el mapa el sprite *stype* indicado y no se supera el valor máximo del recurso.

Modo de uso: *recurso2 dropper > exchangeResourceIfHeld stype=recurso1 avatar=avatar resource=peso value=1 scoreChange=1* - Si el recurso 2 colisiona con el dropper, se transforma en el recurso 1 y se aumenta el valor del peso y la puntuación del avatar en 1.

- **killIfHeld.** Destruye el segundo sprite, solo si no existe en el mapa el recurso *heldResource*.

Modo de uso: *avatar r2key > killIfHeld heldResource=r1key* - Si el avatar colisiona con la llave 2, la se destruye únicamente si ya se obtuvo la llave 1.

5.2.2. Agente Inteligente

Siguiendo la arquitectura general del sistema descrita en el Capítulo 5.1.2, el agente inteligente consta de 3 partes bien diferenciadas: la definición de la estructura que tendrán los archivos a utilizar, la implementación de las diferentes interfaces y algoritmos, y la implementación del controlador que interacciona con el resto de elementos del agente inteligente. En las siguientes secciones se explica la implementación de cada una de estas partes.

5.2.2.1. Archivos

Hay un total de 6 archivos que se utilizan a lo largo de la ejecución del agente inteligente. Estos han sido nombrados como *Game Config File*, *Model File*, *Domain File* y *Data File*. Todos se explican, uno por uno, en las secciones siguientes.

Game Config File Es uno de los de los archivos que ha de definir el usuario, y para el correcto funcionamiento del sistema será necesario que incluya la siguiente información: las rutas donde se encuentran el *Model File* y el *Domain File*, las rutas donde se crearán el *Data File* y el *Problem File*, y el nombre del dominio PDDL; además de una serie de secciones que informarán al sistema de las correspondencias entre un lenguaje y otro. Estas secciones se explican a continuación:

- **pddlCorrespondence:**

Define la correspondencia entre los sprites del juego y los predicados PDDL que se añadirán al *Problem File*. El usuario define, para los sprites que desee (no tienen que ser todos), los predicados que se generarán si dicho sprite se encuentra en el mapa.

- **minizincCorrespondence:**

Define la correspondencia entre los sprites del juego y las líneas que se añadirán al *Data File*. El usuario debe define, para los sprites que desee, dos órdenes: la línea de texto que se añadirá al *Data File* si dicho sprite se encuentra en el mapa y la que se añadirá si no se encuentra en él.

- **minizinc_to_PDDL_correspondence:**

Define la correspondencia entre la salida resultante de la ejecución de MiniZinc y los predicados PDDL que conformarán el objetivo en el *Problem File*. El usuario define, para una línea de texto concreta de la salida, el conjunto de predicados que se añadirán al objetivo PDDL. Como se puede formatear la salida de MiniZinc a voluntad en el *Model File*, la idea es que defina en este apartado las transformaciones de las posibles salida a los predicados que le correspondan.

- **pddl_to_agent_correspondence:**

Define la correspondencia entre una acción PDDL y una acción que entienda el agente inteligente. El usuario indica, para cada palabra que conforma una acción, la traducción que tendrá; componiendo así la nueva acción formateada.

Este archivo se escribe en formato YAML, y su estructura de datos se almacena, en tiempo de ejecución, en un objeto del tipo *GameInformation*, clase a través de la cual el sistema puede acceder cómodamente a toda la información.

Model File Es otro de los archivos que ha de definir el usuario. En él, se plante el PSR o el POR y, junto a la información que le proporciona el sistema sobre la información del nivel, se resuelve. El usuario puede definirlo con total libertad, pero es necesario para el correcto funcionamiento del sistema que **todos** los parámetros que no sean definidos en el *Model File* y **solo** los parámetros no definidos, se encuentren en el *Data File* tras su creación.

Data File Es el archivo de datos que generará automáticamente el sistema, utilizando la información del estado del juego y el apartado *minizinc-Correspondence* del *Game Config File*.

Domain File Es el último de los archivos que ha de definir el usuario. En él, se definirá el Dominio PDDL del problema de planificación automática a resolver.

Problem File Es el archivo de Problema PDDL que generará automáticamente el sistema, utilizando la información del estado del juego y el apartado *pddlCorrespondence* del *Game Config File*.

5.2.2.2. Interfaces

Para que el agente inteligente se comunique con el software necesario, se han creado dos clases que actúan a modo de interfaz. Estas se encargan de realizar las llamadas a MiniZinc y a Metric-FF, así como el procesamiento de datos de entrada y salida, para adaptarlos al formato adecuado.

MinizincInterface Su constructor requiere de un objeto *GameInformation*, al cual se almacena una referencia. Consta de los siguientes métodos:

- **generate_dzn.** Dado un estado de juego y la información proporcionada por el *GameInformation*, procesa la información del estado al formato correspondiente, genera el *Data File* y la escribe en él.
- **execute_solver.** Hace una llamada a MiniZinc con el *Model File* indicado y el *Data File* creado, y devuelve la salida en forma de lista de *strings*, cada uno correspondiendo a una línea de la salida.
- **translate_to_PDDL.** Dada una lista de *strings*, y utilizando la información proporcionada por el *GameInformation*, genera la lista de predicados correspondiente.
- **plan.** Realiza todo el procesamiento necesario para, dado un estado de juego, obtener el conjunto de objetivos en forma de predicados PDDL. Utiliza para ello los métodos descritos anteriormente. En la Figura 5.7, se puede apreciar su diagrama de secuencia.

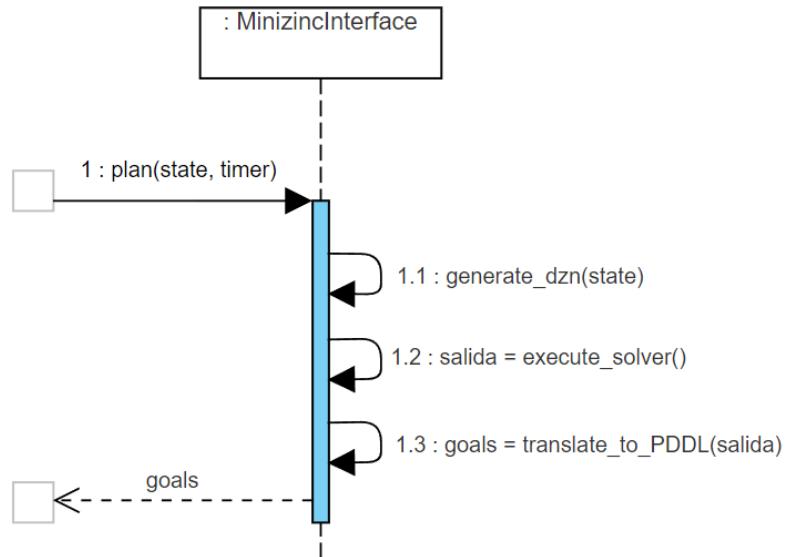


Figura 5.7: Diagrama de secuencia del método `plan()`.

PDDLIface Su constructor requiere de un objeto de tipo *GameInformation*, al cual se almacena una referencia. Consta de los siguientes métodos:

- **translateGameStateToPDDL**. Dado un estado de juego, y utilizando la información proporcionada por el *GameInformation*, procesa la información del estado y la almacena en el objeto *PDDLGameStatePredicates*. Esto es una lista de *strings*, donde cada uno corresponde a un predicado PDDL.
- **createProblemFile**. Crea el archivo de problema PDDL, utilizando los Predicados almacenados en el objeto *PDDLGameStatePredicates* y los Objetivos almacenados en el objeto *goals*.
- **execute_metricff**. Hace una llamada a Metric-FF con el *Domain File* indicado y el *Problem File* creado y filtra la salida, devolviendo únicamente la lista de acciones. Cada acción, a su vez, se representa como una lista de *strings*.
- **translate_to_agent**. Dada una lista de acciones PDDL correspondiente a un plan, y utilizando la información proporcionada por el *GameInformation*, la transforma en una lista de acciones comprensible por el agente inteligente.
- **findplan**. Realiza todo el procesamiento necesario para, dado un estado de juego, obtener el conjunto de acciones que tendrá que realizar

el agente inteligente. Utiliza para ello los métodos descritos anteriormente. En la Figura 5.8, se puede apreciar su diagrama de secuencia.

- **getNextAction.** Funciona a modo de iterador. Cada vez que se le llama, devuelve la siguiente acción del plan, ya traducido al lenguaje del agente inteligente.

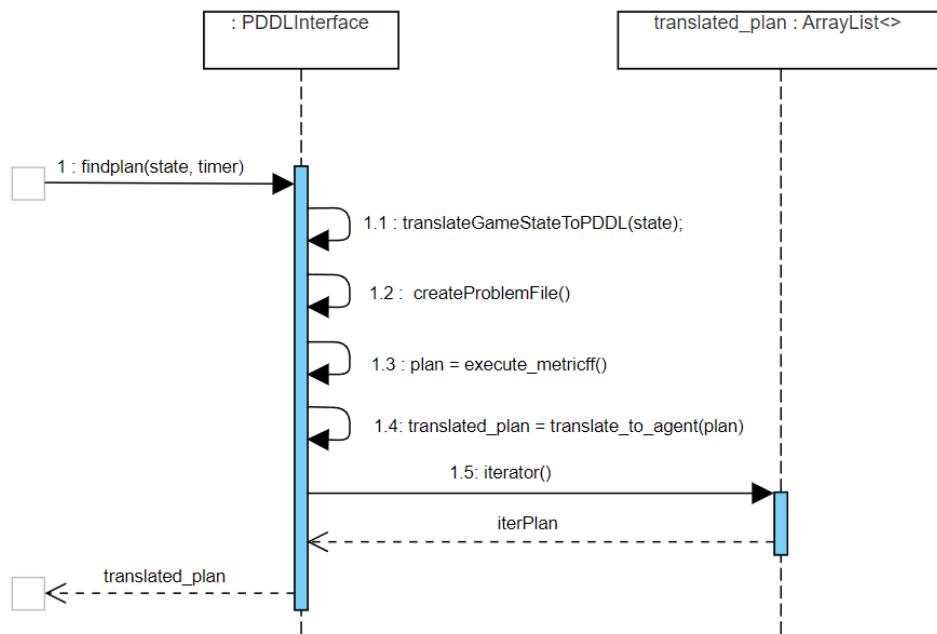


Figura 5.8: Diagrama de secuencia del método `findplan()`.

5.2.2.3. Algoritmos de Búsqueda Heurística y Dropper

Se han implementado los siguientes algoritmos de BH: A*, RTA*, LRTA*, LRTA*(k) y D* Lite. Todos de forma genérica, de forma que para adaptar estos algoritmo a otro videojuego solo es necesario redefinir las funciones del nodo que se utiliza, adaptándolas al funcionamiento del juego en cuestión. Además, se ha creado el objeto *Dropper*, un algoritmo simple que, cuando se le llama, busca la forma de soltar un objeto. Su método `plan()` sencillamente busca entre las casillas colindantes y lo suelta en la primera libre que encuentra, devolviendo el plan como una lista de acciones interpretables por GVGAI.

Para que los algoritmos implementados funcionen correctamente, se utiliza un objeto *Node*, que tiene implementadas todas las variables y métodos que estos utilizan. En la Figura 5.9 se muestra el diagrama de clases de estos objetos.

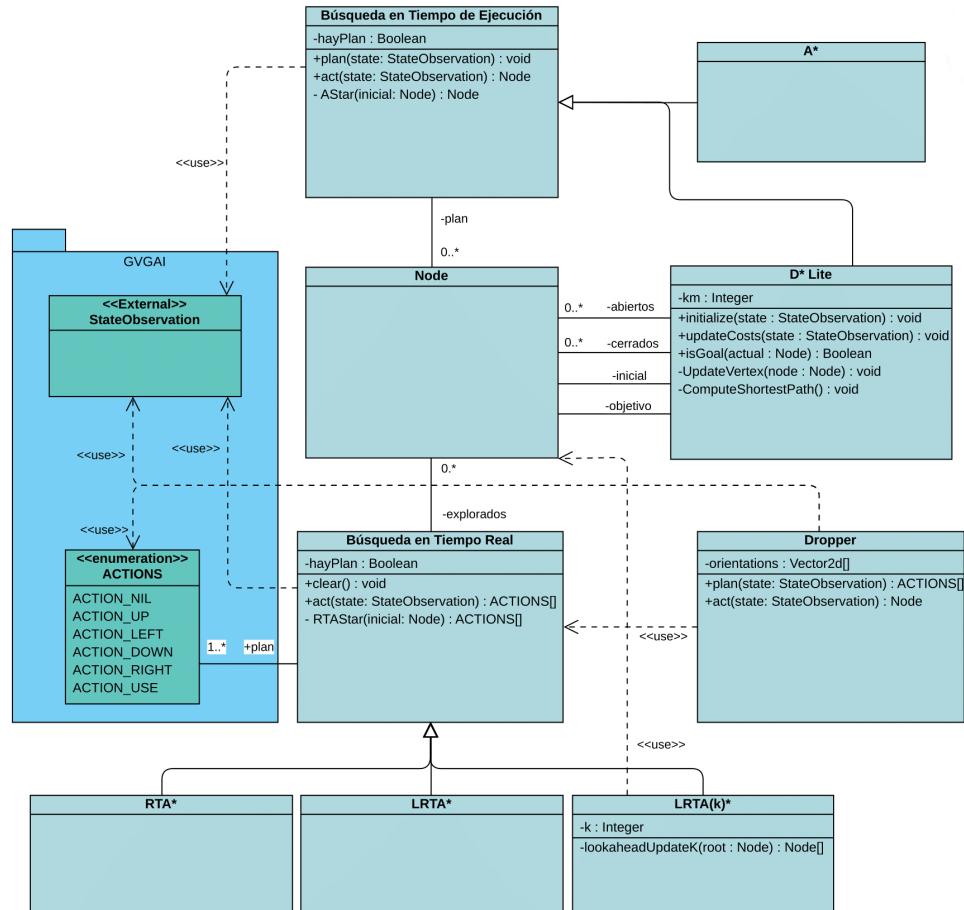


Figura 5.9: Diagrama de clases de los algoritmos de BH implementados.

5.2.2.4. Controladores

El controlador es la parte del agente inteligente encargada de llamar a todos los módulos y comunicarse con el juego. Para esto último, ha de ser una extensión de la clase *AbstractPlayer* de GVGAI, y tener un método `act()` definido, el cual es llamado por el entorno en cada *tick* (turno o instante de tiempo de juego) y devuelve la acción a realizar. Como el funcionamiento de cada algoritmo de BH es diferente, es necesario que el funcionamiento del controlador también lo sea. En las Figuras 5.10, 5.11 y 5.12 se muestra el diagrama de secuencia del método `act()` para los diferentes controladores.

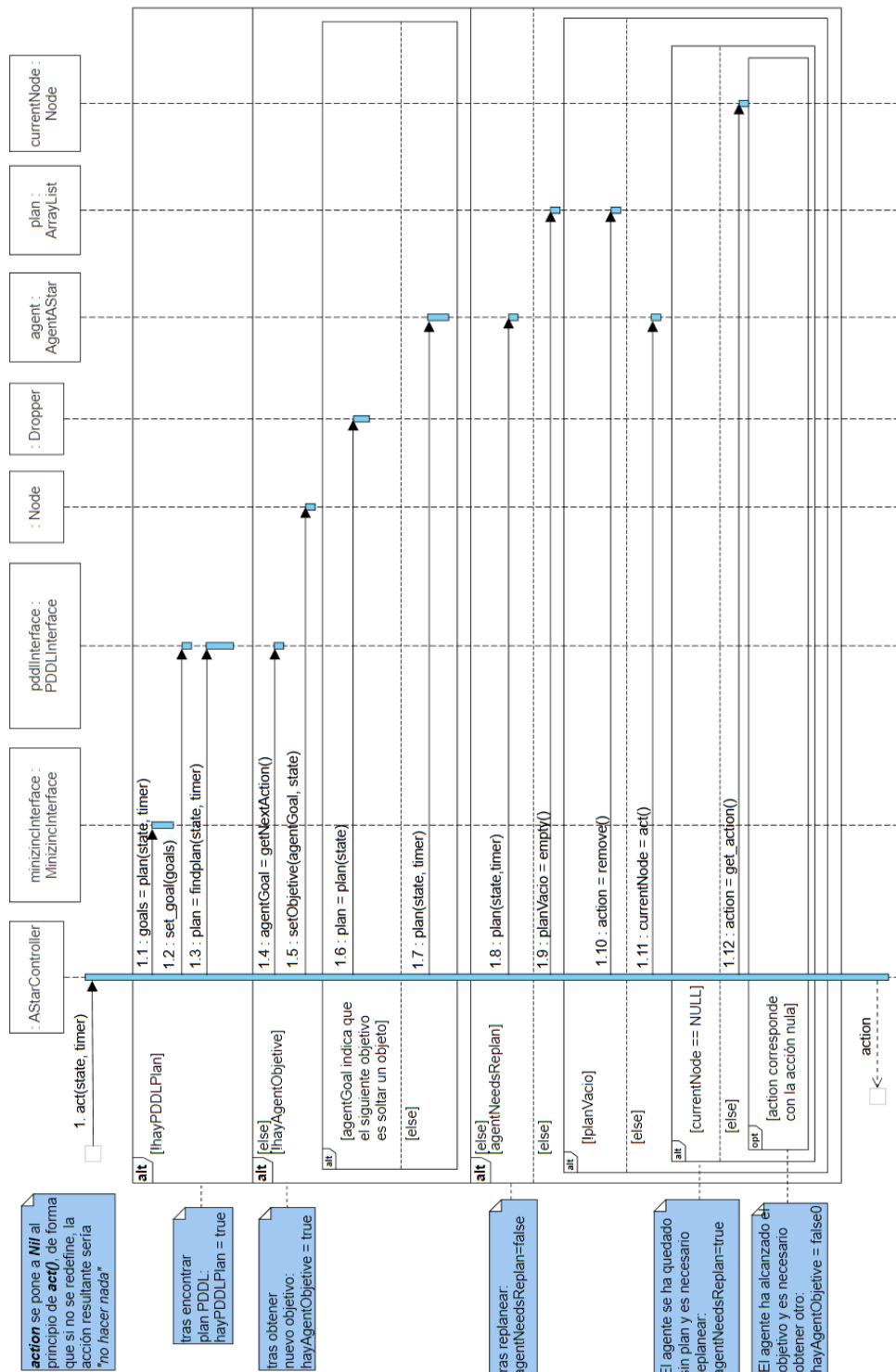


Figura 5.10: Diagrama de secuencia del método `act()` del controlador con A*.

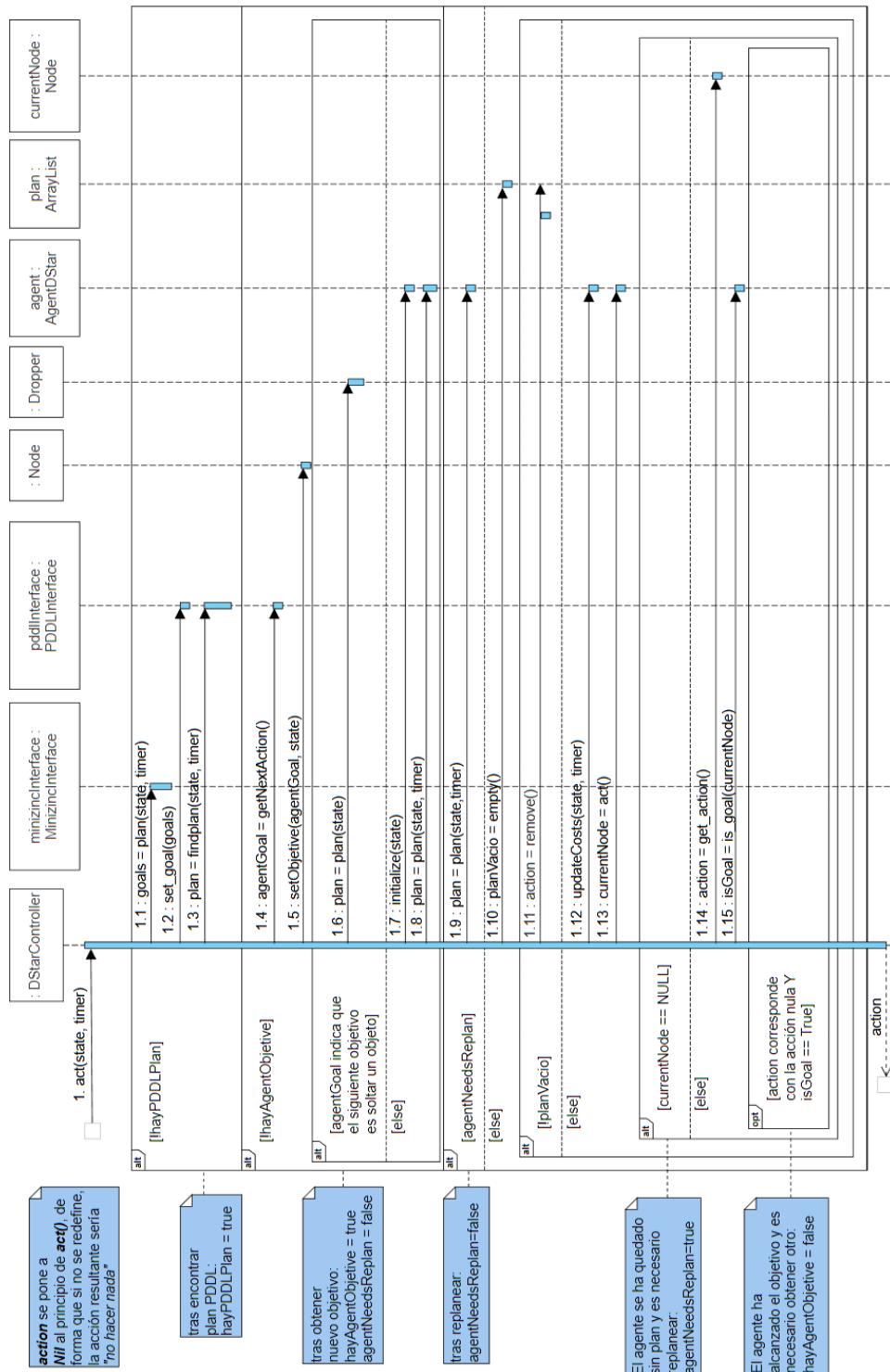


Figura 5.11: Diagrama de secuencia del método **act()** del controlador con D* Lite.

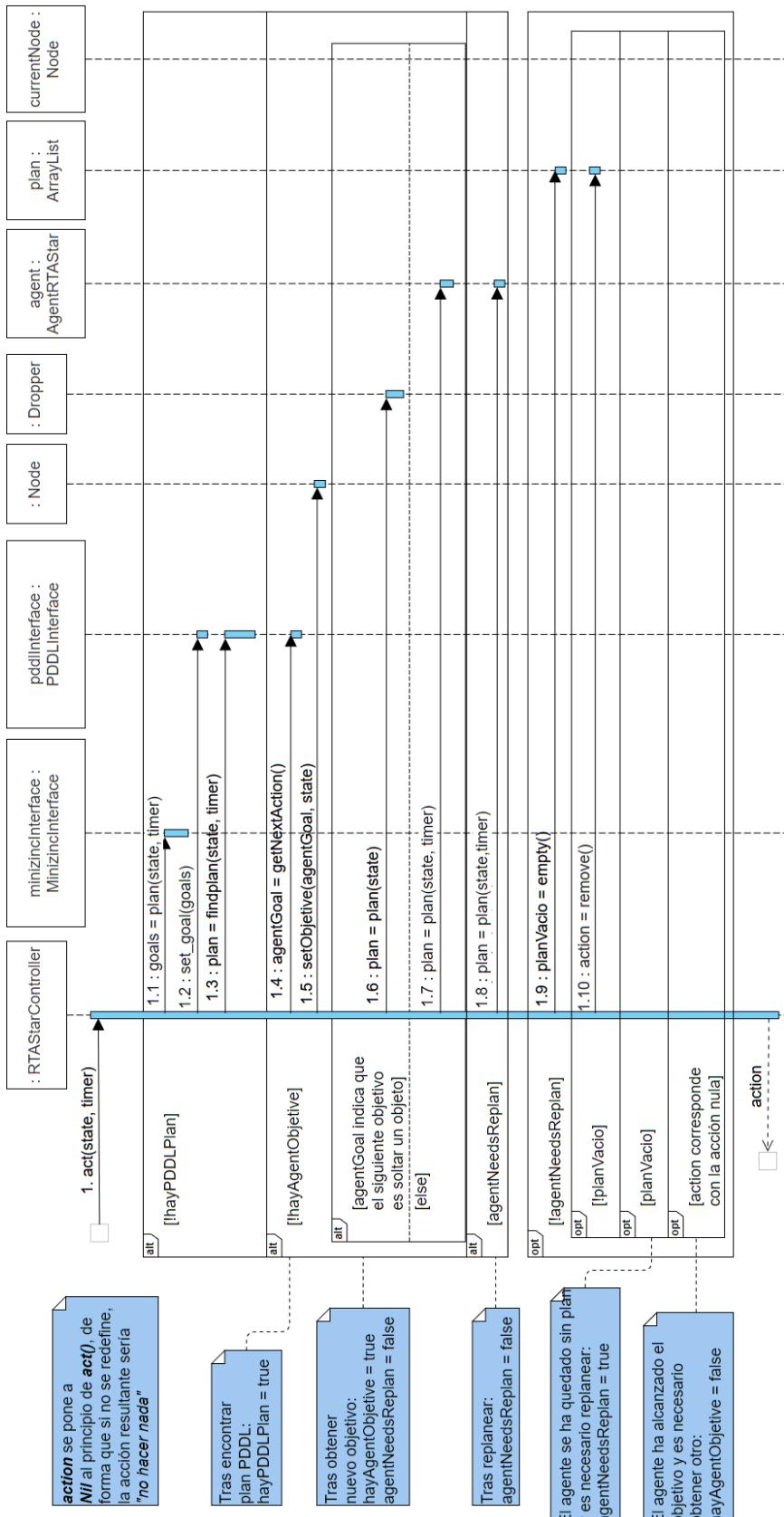


Figura 5.12: Diagrama de secuencia del método `act()` del controlador con RTA*. Son idénticos a él los controladores con LRTA* y LRTA*(k).

Capítulo 6

Experimentación

Para poner a prueba el agente inteligente desarrollado y evaluar su funcionamiento y rendimiento a la hora de jugar al videojuego *El Mochilero*, se han creado cuatro niveles de distinta dificultad. Para ello, se han utilizado como referencias diferentes LDF ya implementados en GVGAI, a los que se le han hecho las adaptaciones necesarias para encajar en lógica del juego creado. Para la selección de niveles, se han buscado mapas que fueran conceptualmente diversos entre ellos, y además con tamaños notablemente diferentes, para estudiar la capacidad del agente para enfrentarse a problemas variados y su evolución conforme aumenta la complejidad del problema (entendida como el tamaño del mapa). En las figuras siguientes se muestra una representación visual de los mapas de dichos niveles.¹ En resumen, los mapas utilizados son:

- Nivel 1 - Tamaño 10x19, 19 casillas. Es el nivel utilizado como ejemplo en los capítulos anteriores, ya representado en la Figura 1.1.
- Nivel 2 - Tamaño 18x22, 396 casillas. Figura 6.1
- Nivel 3 - Tamaño 31x28, 868 casillas. Figura 6.2.
- Nivel 4 - Tamaño 41x48, 1.968 casillas. Figura 6.3.

Además, se ha evaluado el comportamiento del agente inteligente utilizando los siguientes algoritmos: A*, D* Lite, RTA*, LRTA* y LRTA*(k) con $k = 1, 10$ y 100 .

¹Indicar que se ha modificado el número de ticks (turnos o instantes de tiempo) que puede durar una partida como máximo. Esto representa, a efectos prácticos, el número de acciones que tiene el agente inteligente para intentar terminar el juego. El predeterminado en GVGAI es de 2.000 ticks, y se ha aumentado hasta 10.000 para poder analizar el comportamiento de los algoritmos sin límite de acciones y comprobar que, eventualmente, todos los algoritmos implementados terminan la partida.

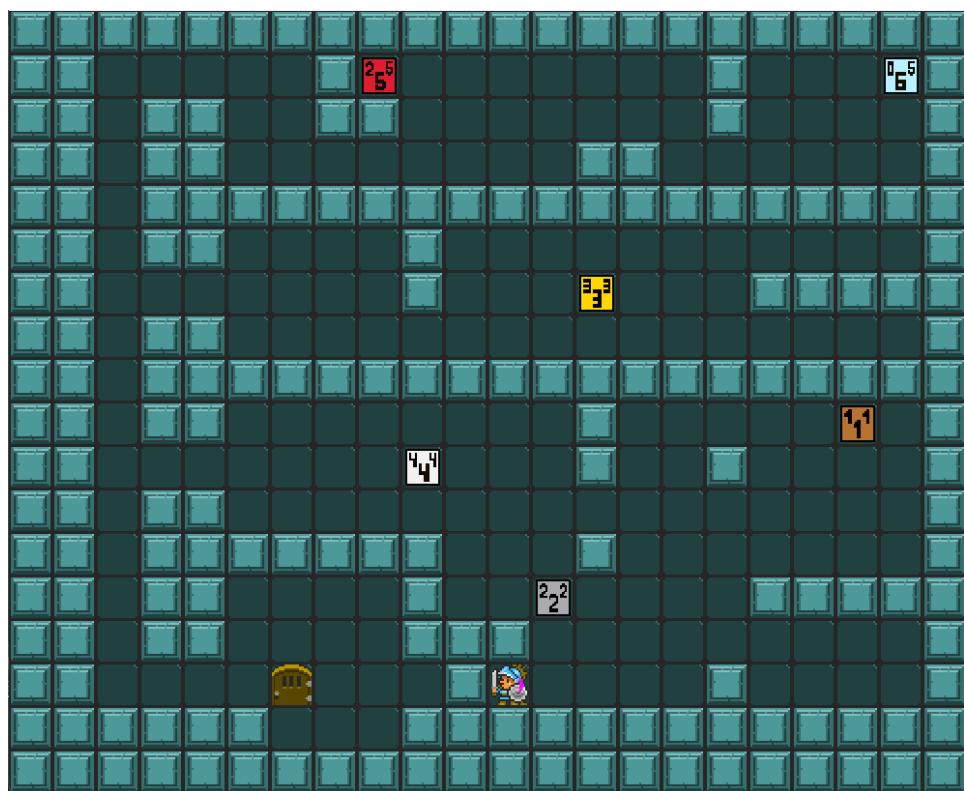


Figura 6.1: Mapa del nivel 2.

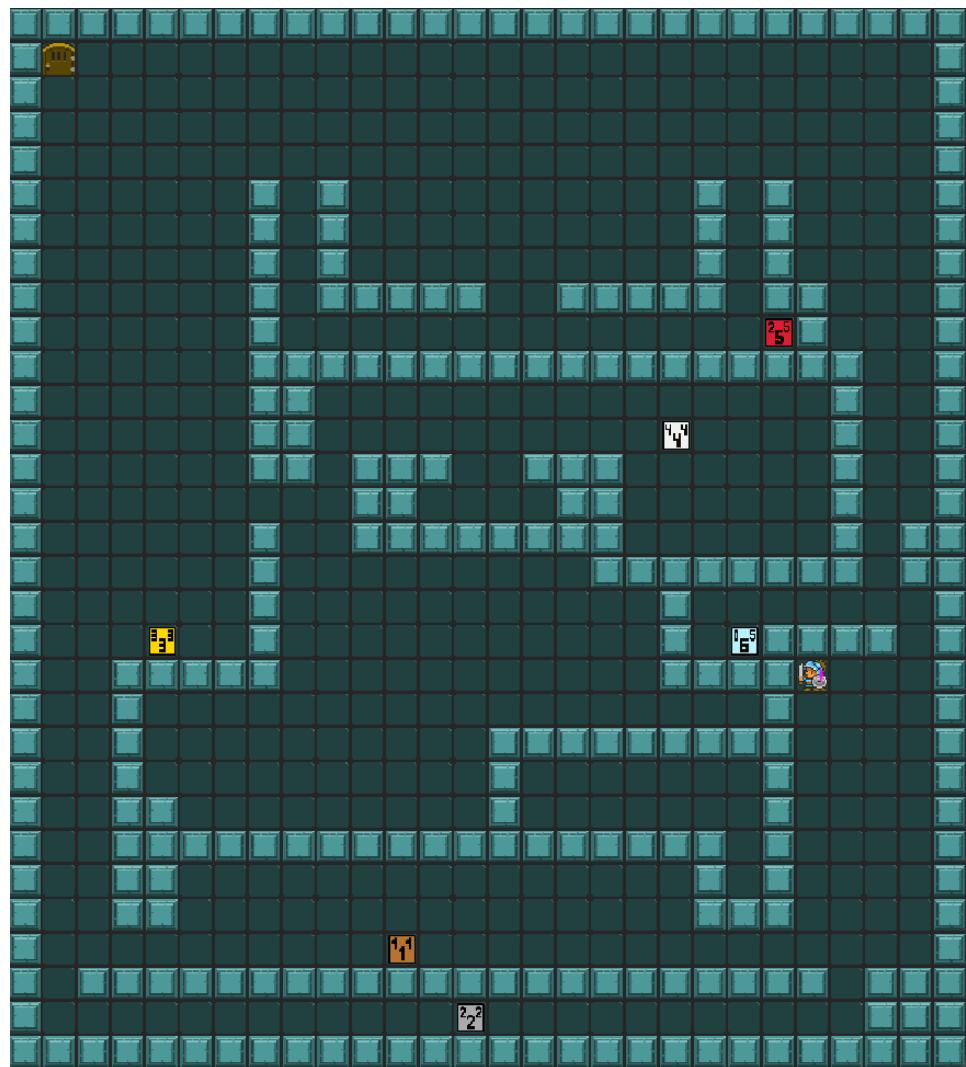


Figura 6.2: Mapa del nivel 3.

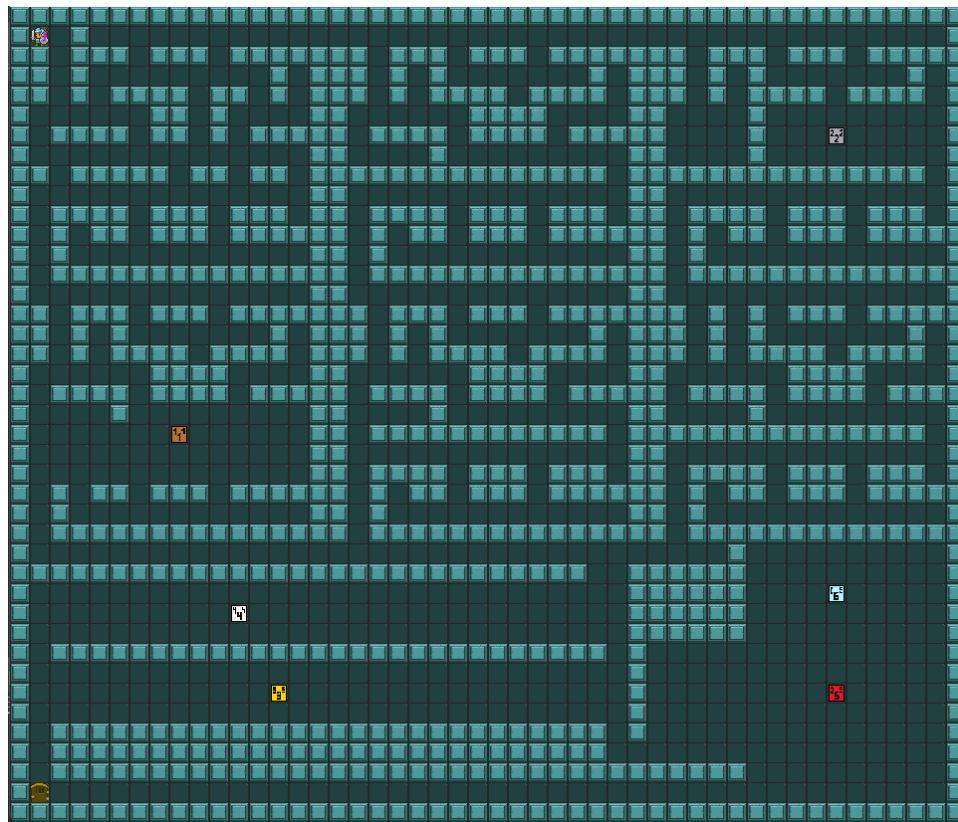


Figura 6.3: Mapa del nivel 4.

En este capítulo, se estudian y comparan las características del agente desarrollado y cada uno de sus módulos. En primer lugar, en la Sección 6.1, se analiza la capacidad de cada uno de los módulos del agente inteligente para resolver el videojuego desarrollado. A continuación, en la Sección 6.2, se analiza la eficacia del agente para resolver el problema, y su evolución a lo largo de los múltiples mapas desarrollados. Seguidamente, en la Sección 6.3, se analizan las diferencias entre los algoritmos de BH utilizados, comparando sus resultados. Finalmente, en la Sección 6.4, se realiza una comparación preliminar con otras técnicas utilizadas en el desarrollo de agentes inteligentes, para tener una mayor perspectiva del rendimiento del agente inteligente desarrollado.

6.1. Estudio Cualitativo

En primer lugar, se estudia la capacidad que tiene cada uno de los módulos del agente inteligente para resolver el problema que se le ha asignado.

6.1.1. Módulo MiniZinc

Lo primero que se ha comprobado es que el módulo MiniZinc encuentra la solución óptima para este problema. En *El Mochilero*, el peso máximo que puede llevar el jugador es de 10 unidades, y hay un total de 6 recursos que se pueden obtener, cada uno con un coste y peso propios. Estos son:

- **Recurso 1:** $peso = 1$, $valor = 1$.
- **Recurso 2:** $peso = 2$, $valor = 2$.
- **Recurso 3:** $peso = 3$, $valor = 3$.
- **Recurso 4:** $peso = 4$, $valor = 4$.
- **Recurso 5:** $peso = 2$, $valor = 5$.
- **Recurso 6:** $peso = 0$, $valor = 5$.

Bajo estas condiciones, el conjunto de mayor valor que se puede tener es el compuesto por los recursos 1, 3, 4, 5 y 6, alcanzando un valor total de 18 puntos. En la Figura 6.4 se muestra la salida del módulo MiniZinc, que indica que se debe obtener este mismo conjunto. El conjunto óptimo obtenido no varía con el mapa y la distribución espacial de obstáculos y objetos, así que el el conjunto de objetos óptimos obtenido por el módulo MiniZinc siempre será el mismo si el conjunto de objetos no cambia su peso y/o valor, por lo que este módulo es capaz de resolver con éxito el problema de optimización que contiene el videojuego, el cual se mantendrá inmutable durante toda la ejecución del mismo.

```
'recurso1=1
recurso2=0
recurso3=1
recurso4=1
recurso5=1
recurso6=1
-----
=====
```

Figura 6.4: Resultado devuelto por el módulo MiniZinc. Si un recurso está igualado a 1, indica que debe pertenecer al conjunto final de objetos. Si está igualado a 0, indica que no.

6.1.2. Módulo PDDL

El siguiente análisis estudia la capacidad del módulo PDDL para encontrar un plan con el que obtener el conjunto óptimo anterior. Nuevamente, como el conjunto de objetos no varía con el mapa, así que el plan generado por el módulo PDDL siempre es el mismo. Para obtener el plan, las posibles acciones que tiene el resolutor son:

- **Coger un objeto.** Por el orden de precedencias fijado, para coger un objeto es necesario haber tenido en algún momento el objeto anterior.
- **Soltar.** Siempre se suelta el objeto de menor precedencia que se tiene en ese momento en la mochila.
- **Salir.** Debe ser siempre el final de todo plan, pues el juego termina cuando se llega a la salida.

Bajo estas condiciones, el plan obtenido por el módulo PDDL es el siguiente:

1. COGER RECURSO1
2. COGER RECURSO2
3. COGER RECURSO3
4. COGER RECURSO4
5. SOLTAR RECURSO1
6. SOLTAR RECURSO2
7. COGER RECURSO5
8. COGER RECURSO1
9. COGER RECURSO6
10. SALIR

Para verificar que el plan alcanza correctamente el conjunto de objetos óptimo y respeta las restricciones impuestas, en la Tabla 6.1 se simula dicho plan. Como se puede ver, este cumple dichas condiciones y termina con el conjunto de objetos indicado por el módulo MiniZinc, por lo que se puede afirmar que este módulo es capaz de resolver con éxito el problema de precedencias que hay en el juego, alcanzando así el conjunto de objetos deseado.

It.	Acción	Mochila	Peso	Mundo
0			0	R1, R2, R3, R4, R5, R6
1	COGER R1	R1	1	R2, R3, R4, R5, R6
2	COGER R2	R1, R2	3	R3, R4, R5, R6
3	COGER R3	R1, R2, R3	6	R4, R5, R6
4	COGER R4	R1, R2, R3, R4	10	R5, R6
5	SOLTAR R1	R2, R3, R4	9	R1, R5, R6
6	SOLTAR R2	R3, R4	7	R1, R2, R5, R6
7	COGER R5	R3, R4, R5	9	R1, R2, R6
8	COGER R1	R1, R3, R4, R5	10	R2, R6
9	COGER R6	R1, R3, R4, R5, R6	10	R2
10	SALIR			

Tabla 6.1: Simulación de ejecución del plan obtenido por el módulo PDDL. En la iteración 0 se muestra el estado inicial, y en las posteriores el estado resultante de ejecutar la acción correspondiente. Para cada acción, se muestra el estado que genera, indicando los recursos en la mundo y en la mochila, así como el peso que tiene en ese momento. Para indicar los recursos disponibles según las restricciones de precedencias, estos están destacados en negrita.

6.1.3. Módulo Heurístico

Finalmente, se ha comprobado que el agente inteligente es capaz de completar el plan generado por el módulo PDDL. Esto se cumple en todos los niveles creados, y con todos los algoritmos evaluados, de forma que el módulo heurístico siempre es capaz de encontrar cada uno de los objetos en el orden especificado, soltarlos cuando sea pertinente y llegar a la salida con éxito. De esta forma, el agente inteligente es capaz de conseguir, en todos los casos, superar el nivel con el conjunto de objetos de valor óptimo.

Cada algoritmo de BH tiene su forma característica de alcanzar el objetivo, por lo que es interesante estudiarlas. Para ello, se muestra a continuación el comportamiento de cada uno de los algoritmos en la misma situación. Se ha elegido, en concreto, el mapa 3 del videojuego, cuando el objetivo es obtener el recurso 6 (esta situación se muestra en la Figura 6.5). Dicha situación ha sido escogida porque permite ilustrar, de forma clara, la diferencia de comportamiento entre todos ellos.

6.1.3.1. Algoritmos de Búsqueda en Tiempo Real

Los algoritmos LRTA* y RTA*, al actualizar únicamente el valor heurístico de la casilla en la que se encuentran, terminan recorriendo muchas casillas que tienen un bajo valor heurístico pero no conducen hasta el objetivo, in-

cluso varias veces, para actualizar su valor heurístico a uno más cercano a su coste real. Esto se ve reflejado en el caso ilustrado en la Figura 6.5. En él, estos algoritmos recorren varias veces las zonas A y B antes de, eventualmente, avanzar en la búsqueda y obtener el recurso 6. Esto se debe a que la heurística que se está utilizando es la distancia Manhattan, lo cual hace que el algoritmo tiende a explorar dichas zonas (pues según la heurística están más cerca del objetivo), por encima de ir hacia la izquierda (pues según la heurística significa alejarse del objetivo), a pesar de que es el camino correcto. Concretamente, LRTA* explora un total de 8 veces la zona A y 4 veces la zona B. RTA*, como se explica en el Capítulo 4, escapa con mayor facilidad que LRTA* de los mínimos locales, lo cual se ve reflejado en este caso, ya que entra la mitad de las veces en estas zonas (4 en A y 1 en B). En el caso de LRTA*(k), con $k = 1$ se comporta exactamente igual que LRTA* (tal y como cabría esperar), pero conforme aumenta el valor de k se amplía el número de casillas cuya heurística se actualiza por ejecución y, en consecuencia, se disminuye esa reexploración de casillas. Con $k = 10$, entra 2 veces en A y 1 vez en B; y con $k = 100$, solo entra una vez en cada zona.

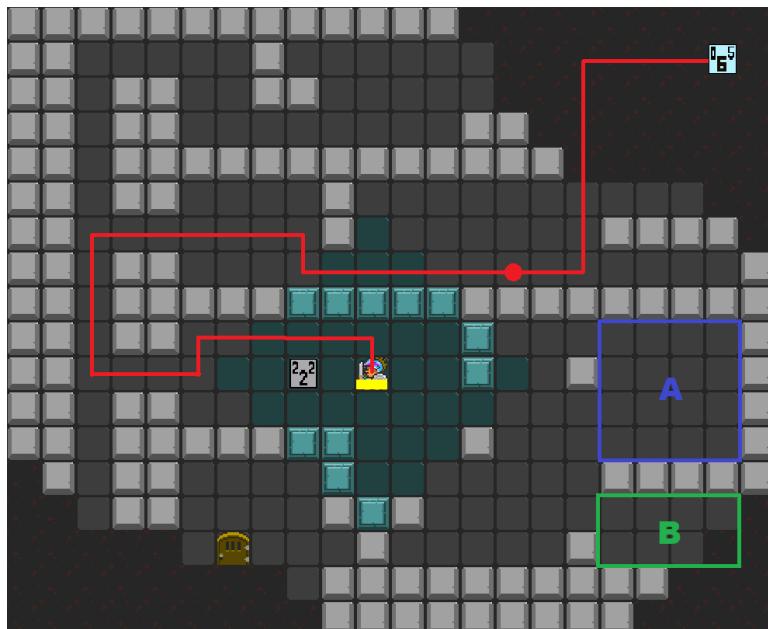
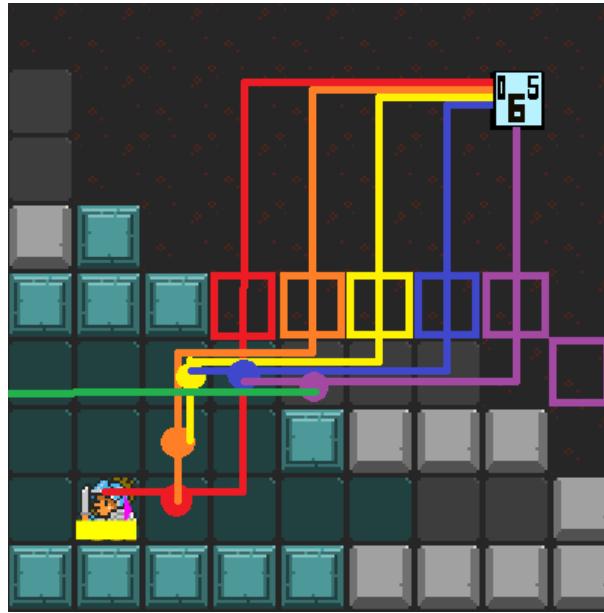
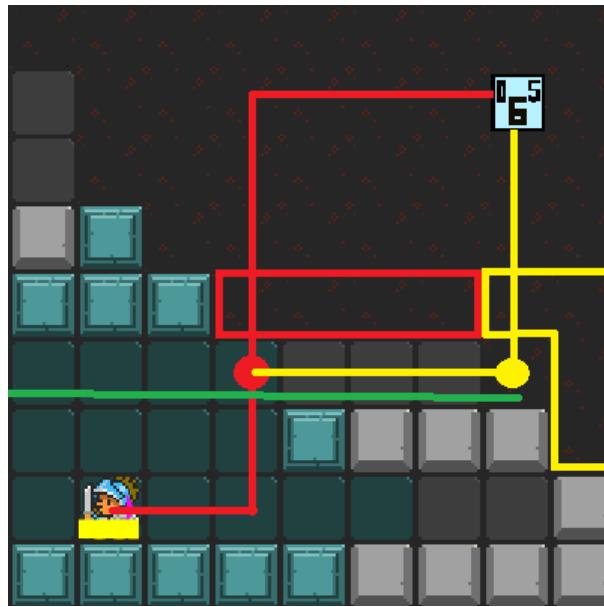


Figura 6.5: Situación sobre la que se ilustran los comportamientos de los diferentes algoritmos de BH. El objetivo actual del agente es obtener el recurso 6 (cuadrado celeste). Las casillas azuladas son las que se encuentran en la zona de visibilidad del jugador, las grisáceas son casillas ya exploradas con anterioridad y las negras son casillas sin explorar, cuyo contenido es desconocido. Se marcan un camino y un punto rojo, así como dos zonas, A y B, relevantes para la explicación de algunos algoritmos.



((a)) Replanificaciones realizadas por D* Lite. El orden en el que se han generado los planes es: (1) rojo, (2) naranja, (3) amarillo, (4) azul, (5) morado y (6) verde, siendo este último el plan que, finalmente, le hace salir del callejón sin salida en el que se encontraba.



((b)) Replanificaciones realizadas por A*. El orden en el que se han generado los planes es: (1) rojo, (2) amarillo y (3) verde, siendo este último el plan que, finalmente, le hace salir del callejón sin salida en el que se encontraba.

Figura 6.6: Replanificaciones realizadas por D* Lite y A* en la situación mostrada en la Figura 6.5. Cada color representa un plan. El círculo de su color indica el momento en el que se aborta (y por tanto se vuelve a planificar) y los rectángulos de su color representan los nuevos muros que ha descubierto durante la ejecución de dicho plan.

6.1.3.2. Algoritmos de Búsqueda Offline e Incremental

En la situación elegida, tanto D* Lite como A* encuentran, inicialmente, el mejor camino posible con la información que se tiene del mapa (camino rojo de la Figura 6.5), y el agente los sigue exactamente de la misma forma; no obstante, sus diferencias se aprecian a la hora replanificar cuando se encuentra un obstáculo en dicho camino. Para ilustrar este fenómeno, se muestra el comportamiento de estos algoritmos en una posición más avanzada del plan (punto rojo de la Figura 6.5), donde se empieza a obtener nueva información del mundo y cada algoritmo reaccionan de forma diferente a ello. En esta situación, nos encontramos ante un camino sin salida, por lo que eventualmente ambos algoritmos llegarán a la conclusión de que hay que salir de esa zona y buscar un camino alternativo.

En el caso de D* Lite, replanifica cada vez que recibe información nueva del entorno. Esto hace que el número de planificaciones tienda a ser alto pero, a cambio, reacciona inmediatamente a los cambios, recalculando inmediatamente el camino óptimo a seguir. Esto se refleja en la Figura 6.6(a), donde se muestra que D* Lite replanifica un total de 5 veces (una por casilla) hasta encontrar el camino alternativo. Esto se debe a que está explorando una zona nueva y, en cada paso, recibe información nueva con la que actualizar el plan; por otro lado, solo avanza 5 casillas por el camino sin salida antes de encontrar el camino alterno. En contraposición, A* solo replanifica cuando su camino no es continuable, lo que hace que tienda a replanificar menos veces que D* Lite, pero en ocasiones puede haber recorrido casillas inútilmente, e incluso verse obligado a volver sobre sus pasos. Esto se ilustra en la Figura 6.6(b), donde se muestra que A* solo replanifica 2 veces (las 2 ocasiones en las que se encuentra con un muro); pero en contra posición, ha avanzado 8 casillas por el camino sin salida antes de encontrar el camino alterno y se ve, además, obligado a volver sobre sus pasos.

6.2. Estudio Cuantitativo

Aunque se ha comprobado que el agente inteligente desarrollado resuelve satisfactoriamente el problema al que se enfrenta, es interesante hacer un estudio sobre su rendimiento. Esto es, principalmente, el tiempo total invertido en cada módulo y su evolución con el tamaño del mapa.²

²Es relevante especificar, de cara a los resultados temporales obtenidos, que el dispositivo utilizado es un portátil Lenovo Legion Y540-15IRH, con procesador Intel Core i7-9750HF (información completa sobre el dispositivo en <https://www.pcccomponentes.com/lenovo-legion-y540-15irh-intel-core-i7-9750hf-16gb-1tb-512gb-ssd-gtx1660ti-156>).

6.2.1. Módulos MiniZinc y PDDL

Estos dos módulos han sido aunados en una misma sección, pues su experimentación y resultados son similares. Como ya se ha explicado al inicio del capítulo, los objetos utilizados en los cuatro mapas no varían, por lo que, teóricamente, el tiempo de ejecución de ambos módulos debería mantenerse estático con el uso de diferentes algoritmos de BH o mapas. Para comprobar esto, se han hecho siete ejecuciones en cada mapa, cada uno con una configuración de algoritmo de BH diferente, y se han recogido los tiempos obtenidos. Los resultados se muestran en las Tablas 6.2 y 6.3.

Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	309.63	232.61	238.47	216.35
D* Lite	331.01	323.15	343.72	379.89
RTA*	249.77	343.72	352.34	239.36
LRTA*	331.47	228.39	333.99	231.51
LRTA*(1)	325.64	339.43	328.36	337.64
LRTA*(10)	260.84	272.89	335.54	340.32
LRTA*(100)	279.59	234.02	341.28	245.36
Promedio	298.28	282.03	324.81	284.35

Tabla 6.2: Tiempos de ejecución del módulo MiniZinc (en ms), utilizando diferentes mapas y configuraciones de BH en el agente inteligente.

Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	206.49	218.79	287.24	224.61
D* Lite	226.48	216.90	271.07	260.40
RTA*	273.74	271.07	207.49	220.89
LRTA*	248.11	222.90	206.64	227.68
LRTA*(1)	223.95	219.73	210.18	214.80
LRTA*(10)	257.85	225.74	208.89	219.96
LRTA*(100)	210.88	219.83	211.68	218.67
Promedio	235.36	227.85	229.03	226.71

Tabla 6.3: Tiempos de ejecución del módulo PDDL (en ms), utilizando diferentes mapas y configuraciones de BH en el agente inteligente.

Tal y como es de esperar, ni en la ejecución de PDDL ni en la de MiniZinc se altera el tiempo de ejecución utilizando un algoritmo de BH u otro, pues son módulos independiente. Además, en ninguno de los dos casos se aprecia una tendencia creciente en el tiempo de ejecución conforme aumenta el tamaño del mapa. Esto tiene sentido, pues solo estamos variando

el tamaño del mapa pero se está manteniendo el mismo conjunto de objetos. Por ello, ni la complejidad del problema a optimizar ni la del proceso de planificación se ven notablemente afectados por el tamaño del mapa. En contraposición, se va a modificar el número de objetos para ver cómo afecta al tiempo de ejecución de estos módulos. En las Tablas 6.4 y 6.5 se puede ver el análisis realizado, con 3 número de objetos diferentes y 10 ejecuciones en cada caso; y en la Figura 6.7 se muestran gráficamente los resultados de media y desviación típica obtenido.

Ejecución	6 Objetos	12 Objetos	24 Objetos
1	271.83	323.88	242.17
2	302.54	330.89	352.49
3	210.96	314.53	312.70
4	214.82	217.72	204.66
5	307.98	310.09	310.79
6	192.24	325.28	310.26
7	200.60	303.58	183.89
8	195.09	213.06	197.97
9	184.71	323.03	214.47
10	318.65	311.11	193.62
Promedio	239.94	297.32	252.30
Desviación Típica	53.87	43.97	62.67

Tabla 6.4: Tiempos de ejecución del módulo MiniZinc (en ms), utilizando diferentes números de objetos y haciendo 10 ejecuciones para cada caso.

Ejecución	6 Objetos	12 Objetos	24 Objetos
1	167.83	191.04	202.61
2	189.96	199.49	197.13
3	188.83	196.33	188.91
4	194.70	193.17	189.37
5	181.80	185.58	199.35
6	197.61	192.89	192.27
7	197.39	193.54	204.64
8	184.89	211.14	186.16
9	189.81	239.29	192.53
10	199.68	195.89	189.89
Promedio	189.25	199.84	194.29
Desviación Típica	9.48	15.37	6.29
Tamaño del Plan	10	30	68

Tabla 6.5: Tiempos de ejecución del módulo PDDL (en ms), utilizando diferentes números de objetos y haciendo 10 ejecuciones para cada caso.

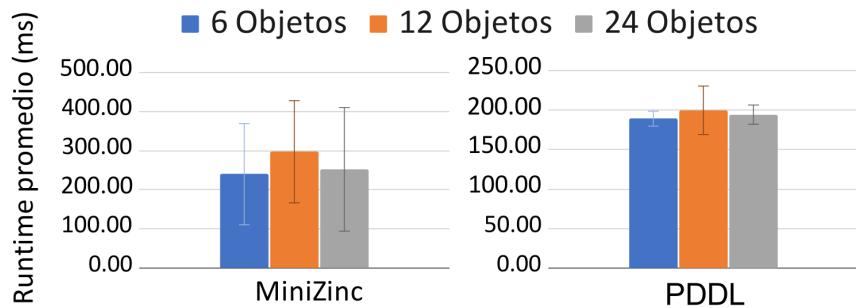


Figura 6.7: Media y desviación típica de los tiempos de ejecución de MiniZinc y PDDL, resultantes de las ejecuciones mostradas en las Tablas 6.4 y 6.5

Estos resultados no reflejan ninguna tendencia concreta, por lo que el tiempo de ejecución de estos módulos no está viéndose afectado por el número de objetos, al menos perceptiblemente. Esto puede deberse principalmente a dos motivos:

- Aumentar el número de objetos no aumenta notoriamente la dificultad del problema, pues los algoritmos se encuentran con, a grandes rasgos, el mismo problema pero más largo. Aun así, aunque la diferencia fuera mínima, se debería poder apreciar un aumento en el tiempo de ejecución tras cuadriplicar el número de objetos (en el caso de PDDL, se ha llegado a generar un plan 7 veces más grande).
- Como tanto MiniZinc como PDDL se ejecutan mediante llamadas al sistema operativo (a diferencia de los algoritmos de BH, que han sido integrados en Java), es probable que el grosor del tiempo de ejecución sea invertido en las propias peticiones y sus correspondientes tiempos de espera. Esto también explicaría los altos valores de desviación típica obtenidos, consecuencia de la irregularidad que llegan a tener los tiempos de espera en las llamadas al sistema.

6.2.2. Módulo Heurístico

Para este módulo se han estudiado, en cada mapa, múltiples parámetros con los que poder analizar en detalle las características diferenciadoras de cada uno de los algoritmos de BH.³ Al igual que en los otros módulos, a continuación se analiza el tiempo de ejecución, en la Tabla 6.6 se puede observar que el módulo heurístico sí se ve afectado con el tamaño del mapa, habiendo en todos los algoritmos una clara tendencia a aumentar el tiempo de ejecución conforme aumenta el tamaño. Esto se puede deber, principalmente, a dos factores: que el número de veces que se planifica un camino

³Todos los datos de esta sección se encuentran agrupados en la Tabla 6.12

esté aumentando con el tamaño del mapa, o que lo esté haciendo el tiempo que se tarda por plan. Ambos factores se estudian a continuación.

Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	10.15	33.03	47.09	44.91
D* Lite	21.70	68.44	142.11	914.86
RTA*	29.20	69.81	99.26	137.53
LRTA*	36.61	126.67	237.94	90.97
LRTA*(1)	48.10	172.90	311.99	149.45
LRTA*(10)	45.84	86.08	179.05	243.44
LRTA*(100)	50.56	96.49	129.24	254.39

Tabla 6.6: Tiempos de ejecución (en ms) de cada uno de los algoritmos de BH en los mapas analizados. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo que menos tiempo ha invertido en ejecutar las acciones calculadas).

6.2.2.1. Número de planificaciones

Número de planificaciones				
Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	19	23	21	29
D* Lite	38	81	92	192
RTA*	197	697	1554	2656
LRTA*	303	1737	4458	1700
LRTA*(1)	303	1737	4458	1700
LRTA*(10)	211	551	1694	2408
LRTA*(100)	211	527	992	2016

Tabla 6.7: Número de veces que planifican cada uno de los algoritmos de BH en cada uno de los mapas. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo que menos veces ha planificado).

En la Tabla 6.7 se puede observar cómo, efectivamente, el número de veces que es necesario planificar ($nPlans$) tiende a aumentar con el tamaño del mapa. En el caso de A* y D* Lite, es por ser un mundo parcialmente observable: conforme más grande es el mapa, mayor es el número de casillas que se desconocen, y más veces habrá que replanificar utilizando la nueva información obtenida. Para A*, este aumento no es tan rápido, ya que solo vuelve a planificar cuando no se puede seguir el plan, y por tanto ha adquirido mucha información entre una ejecución y otra. Para D* Lite,

se replanifica cada vez que se adquiere información nueva, por lo que es lógico que el número de veces que se planifica sea notablemente más alto. En el caso de los algoritmos de BTR, la explicación es simple: como estos algoritmos planifican en cada movimiento, si aumenta el tamaño del mapa, generalmente aumenta de movimientos necesarios para resolver el problema, por lo que será mayor el número de veces que se planifique.

6.2.2.2. Tiempo por plan

Tiempo medio de ejecución por plan (ms)				
Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	0.53	1.44	2.24	1.55
D* Lite	0.57	0.84	1.54	4.76
RTA*	0.15	0.10	0.06	0.05
LRTA*	0.12	0.07	0.05	0.05
LRTA*(1)	0.16	0.10	0.07	0.09
LRTA*(10)	0.22	0.16	0.11	0.10
LRTA*(100)	0.24	0.18	0.13	0.13

Tabla 6.8: Tiempo medio que tarda en planificar cada uno de los algoritmos de BH en cada uno de los mapas. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo que menos tiempo ha invertido por plan).

En la Tabla 6.8 se puede ver que el tiempo empleado por cada plan no tiende a aumentar en los algoritmos de BTR. Esto demuestra que se cumple la propiedad diferenciadora de este tipo de algoritmos, pues han sido diseñados para que sus ejecuciones siempre tengan un tiempo bajo y constante. En el caso de A* y D* Lite, el tiempo dedicado a cada plan sí que sigue una tendencia creciente con el tamaño. La explicación es que estos algoritmos, en cada ejecución, encuentran un plan para ir desde el nodo actual hasta el nodo objetivo. Por ello, a mayor sea el tamaño del mapa, mayor el espacio de búsqueda, y por tanto más nodos será necesario expandir hasta encontrar el plan. Para confirmar esto, se estudia a continuación la evolución del número de expansiones realizadas por cada algoritmo con el tamaño del mapa.

6.2.2.3. Número de expansiones

En la Tabla 6.9 se muestran las expansiones realizadas por cada algoritmo ($nExpan$). Como se puede ver, este valor tiende a aumentar con el tamaño del mapa en todos los algoritmos, lo cual es esperable ya que

los objetos se encuentran más alejados y hay más nodos que explorar. En consecuencia hay que hacer búsquedas más profundas y más anchas para encontrar los caminos hacia los objetivos.

Número de expansiones realizadas				
Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	660	3026	4503	5807
D* Lite	1742	5859	9159	29710
RTA*	197	697	1554	2656
LRTA*	303	1737	4458	1700
LRTA*(1)	303	1737	4458	1700
LRTA*(10)	548	1643	5478	7567
LRTA*(100)	718	3342	4882	14369

Tabla 6.9: Número de veces que expande nodos cada uno de los algoritmos de BH en cada uno de los mapas. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo que menos expansiones ha realizado).

Es curioso destacar que en RTA* y LRTA*, este valor ha de corresponder siempre al de número de planificaciones realizados. Esto se debe a que estos algoritmos siempre hacen una única expansión por plan, correspondiente al nodo en el que se encuentran. A LRTA*(1) le ocurre lo mismo porque, a efectos prácticos, se comporta exactamente igual que LRTA*; de hecho, se puede observar a lo largo de todas las tablas de datos cómo los resultados de LRTA*(1) son idénticos a los de LRTA*, a excepción del tiempo de ejecución que es siempre algo mayor, pues aunque el comportamiento es el mismo, el diseño no lo es, y el de LRTA*(k) con $k = 1$ resulta algo menos eficiente que el de LRTA*.

6.2.2.4. Memoria necesaria

Adicionalmente, vamos a estudiar el número máximo de nodos en memoria que llega a tener cada algoritmo (\maxMem). Esta característica es importante tenerla en cuenta ya que repercute, directamente, en la memoria que necesita el algoritmo para su ejecución. En la Tabla 6.10 se puede apreciar cómo este valor sigue también una clara tendencia de aumento con el tamaño del mapa. En el caso de A* y D* Lite se debe a que, en el proceso de planificación, tienen que mantener todo el árbol de búsqueda en memoria. En el caso de los algoritmos de BTR, se debe a que almacenan en memoria todos los nodos por los que pasan, para poder mantener actualizado el valor heurístico. Por ello, el valor es mayor conforme aumenta el número de nodos que se exploran.

Máximo número de nodos en memoria				
Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	202	625	2005	2517
D* Lite	329	846	1702	4847
RTA*	66	130	348	582
LRTA*	64	176	387	340
LRTA*(1)	64	176	387	340
LRTA*(10)	87	126	336	604
LRTA*(100)	87	126	357	595

Tabla 6.10: Número máximo de nodos en memoria que almacena cada uno de los algoritmos de BH en cada uno de los mapas. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo que menos ha llegado a almacenar en memoria).

6.2.2.5. Número de Ticks

Número de ticks empleados				
Algoritmo	Mapa 1	Mapa 2	Mapa 3	Mapa 4
A*	216	399	408	667
D* Lite	196	364	408	511
RTA*	287	984	1998	3484
LRTA*	428	2424	5825	2140
LRTA*(1)	428	2424	5825	2140
LRTA*(10)	303	772	2235	3059
LRTA*(100)	303	741	1272	2568

Tabla 6.11: Número de ticks que emplea para terminar el juego con cada uno de los algoritmos de BH en cada uno de los mapas. En negrita se resalta el mejor resultado obtenido para cada mapa (este es, el algoritmo con el que menos ticks se ha tardado en resolver el problema).

Por último, el número de ticks (turnos o instantes de tiempo) que se emplean para resolver el problema es una característica fundamental a estudiar en este caso, pues nos indica cuánto tiempo, a ojos del juego, tarda el agente inteligente en resolver el problema. En la Tabla 6.11 se muestra el número de ticks que tarda el agente inteligente en completar el juego con cada uno de los algoritmos de BH. Este valor también tiende a aumentar con el tamaño del mapa. Es lógico pues, mientras más grande sea el mapa, lo más probable es que sea necesario realizar un número mayor de acciones para superar el juego. Además, hay otros 2 factores que influyen en este valor:

- A mayor tamaño de mapa, más casillas inicialmente desconocidas, por lo que es más probable que el plan diseñado no sea válido y haya que replanificar, perdiendo ticks en el proceso. Esto solo afecta a los algoritmos A* y D* Lite, pues los algoritmos de BTR nunca planifican más allá de los nodos colindantes al actual, los cuales que siempre se conocen.
- A mayor tamaño de mapa, más nodos hay que explorar hasta encontrar un camino. Esto no afecta a los algoritmos A* y D* Lite porque hacen una exploración en tiempo de ejecución (en una sola ejecución se calcula una ruta completa). Los algoritmos de BTR, no obstante, hacen la exploración en tiempo real (la ruta se encuentra a lo largo de múltiples ejecuciones), consumiendo ticks en el proceso de exploración.

6.3. Estudio Comparativo de los algoritmos de Búsqueda Heurística

Finalmente, se van a comparar las propiedades de los diferentes algoritmos de BH utilizados, con apoyo en los resultados obtenidos, para concluir cuáles son los fuertes y las debilidades de cada uno. Para ello, primero se van a exponer las diferencias entre los algoritmos de BTR, pues son sencillas de explicar y sus repercusiones son directamente apreciable en las tablas anteriormente mostradas; y tras ello se realizará un análisis comparativo general de todos los algoritmos.

6.3.1. Comparación entre los Algoritmos en Tiempo Real

Entre RTA* y LRTA*, como consecuencia de sus propiedades diferenciadoras explicadas en el Capítulo 4, RTA* suele encontrar la solución en menos ticks. Esta diferencia repercute directamente en *maxMem*, *nExpan* y *nPlans*, haciendo que RTA* consiga, generalmente, mejores resultados en todos los sentidos. Cada una de estas diferencias se explican a continuación:

- **nPlans.** Estos algoritmos planifican tras cada tick, por lo que un menor número de ticks conlleva un menor número de planificaciones.
- **nExpan.** De la misma manera, cada vez que se planifica solo se expande una única vez, por lo que un menor número de planificaciones implica un menor número de expansiones (de hecho, en estos algoritmos, estos dos valores siempre coinciden).
- **maxMem.** Estos algoritmos solo almacenan en memoria los nodos por los que han pasado. Por ello, si la solución es alcanzada en menos ticks, generalmente conllevará que se hayan explorado menos nodos y, por tanto, se hayan almacenado menos.

No obstante, aunque generalmente RTA* encuentra generalmente la solución de forma más rápida, su mejora no es absoluta, y puede haber situaciones en las que LRTA* consiga mejores resultados. En este caso, ocurre en el mapa 4. Estos algoritmos se diseñan para que los ticks sean lo más rápidos posibles, como se puede ver en la Tabla 6.8. Con LRTA*(*k*), en cambio, se puede apreciar que, conforme aumenta el valor de *k*, también aumenta el tiempo de ejecución por tick. Esto se debe a que pueden expandir más nodos en cada ejecución, lo que repercute en un mayor número total de nodos expandidos, pero a cambio es capaz de encontrar el camino en menos intentos. Además, el número de ticks está estrechamente relacionado con el número de planificaciones (pues planifican cada vez que se cambia de casilla), por lo que también tiende a disminuir este valor.

6.3.2. Análisis Comparativo General

- **Número de ticks.** El algoritmo que completa el juego en menos ticks siempre es D* Lite (Tabla 6.11). Esto tiene sentido, ya es un algoritmo que reconfigura el plan cada vez que recibe información nueva, adaptándose lo antes posible a los cambios. A diferencia de él, con A* sólo se replanifica cuando su plan no se puede continuar. Y obviamente, los algoritmos que más ticks invierten son los de tiempo real, pues están diseñados para invertir múltiples ticks en el proceso de búsqueda.
- **Número de planificaciones.** El algoritmo que menos veces planifica es A* (Tabla 6.7). D* Lite lo hace muchas más veces por la razón anteriormente explicada (cada vez que recibe información nueva, vuelve a crear un plan). Aun así, los algoritmos que más veces tienen que planificar son los de tiempo real, ya que lo hacen cada vez que el agente inteligente cambia de casilla.
- **Número de expansiones.** Los algoritmos que menos nodos expande son RTA* y por LRTA* (Tabla 6.9). Esto se debe a que tienen acotado a 1 el número de nodos que pueden expandir por tick. Por esto mismo, referente a LRTA*(k), conforme k aumenta, crece el número de nodos expandidos, hasta que, con un valor lo suficientemente alto, se llegan a expandir más nodos que en A*. D* Lite es el algoritmo que más nodos expande, con diferencia. Esto se debe a que da más importancia al valor heurístico, $h(n)$, lo cual repercute en que la búsqueda inicial sea menos eficiente a la hora de encontrar un camino en pos de una mayor exploración que acelere las posteriores replanificaciones. El problema es que, cada vez que se cambia de objetivo (un total de 8 veces a lo largo de la partida) la información que se tiene sobre los nodos deja de ser útil, por lo que es necesario repetir el proceso. Además, en este juego en concreto, al tener que “dar muchas vueltas cogiendo objetos”, la mayoría del mapa se descubre relativamente rápido, por lo que las capacidades de la búsqueda incremental se diluyen rápidamente. Todo esto hace que, finalmente, D* Lite obtenga este resultado tan negativo.
- **Tiempo de ejecución.** En la misma línea, como consecuencia del alto número de veces que se expande, D* Lite es el algoritmo que más tiempo de ejecución consume (Tabla 6.6). Relativo a los algoritmos de BTR, el bajo número de expansiones que realizan no repercute en un tiempo de ejecución pequeño, pues, aunque consumen poco tiempo por ejecución, en contrapartida obtienen un número de planificaciones del orden de cientos de veces mayor que en A*. En consecuencia, este último es, en todos los casos, el algoritmo con menor tiempo de ejecución.

	Algoritmo	nExpan	maxMem	nPlans	Runtime (ms)	Ticks
Mapa 1	A*	660	202	19	10.15	216
	D* Lite	1742	329	38	21.70	196
	RTA*	197	66	197	29.20	287
	LRTA*	303	64	303	36.61	428
	LRTA*(1)	303	64	303	48.10	428
	LRTA*(10)	548	87	211	45.84	303
	LRTA*(100)	718	87	211	50.56	303
Mapa 2	A*	3026	625	23	33.03	399
	D* Lite	5859	846	81	68.44	364
	RTA*	697	130	697	69.81	984
	LRTA*	1737	176	1737	126.67	2424
	LRTA*(1)	1737	176	1737	172.90	2424
	LRTA*(10)	1643	126	551	86.08	772
	LRTA*(100)	3342	126	527	96.49	741
Mapa 3	A*	4503	2005	21	47.09	408
	D* Lite	9159	1702	92	142.11	408
	RTA*	1554	348	1554	99.26	1998
	LRTA*	4458	387	4458	237.94	5825
	LRTA*(1)	4458	387	4458	311.99	5825
	LRTA*(10)	5478	336	1694	179.05	2235
	LRTA*(100)	4882	357	992	129.24	1272
Mapa 4	A*	5807	2517	29	44.91	667
	D* Lite	29710	4847	192	914.86	511
	RTA*	2656	582	2656	137.53	3484
	LRTA*	1700	340	1700	90.97	2140
	LRTA*(1)	1700	340	1700	149.45	2140
	LRTA*(10)	7567	604	2408	243.44	3059
	LRTA*(100)	14369	595	2016	254.39	2568

Tabla 6.12: Datos extraídos experimentalmente sobre los algoritmos de BH utilizados, que agrupa toda la información mostrada a lo largo de los apartados anteriores. Esto es, para cada algoritmo: el número de expansiones que ha realizado (*nExpan*), el número de nodos que ha tenido, como máximo, almacenados en memoria (*maxMem*), el número de veces que ha planificado (*nPlans*), el tiempo de ejecución empleado en planificar (Runtime) y el número de ticks que ha empleado el agente inteligente en completar el nivel (Ticks). En negrita se resaltan los mejores resultados obtenidos, en cada mapa, para cada parámetro (esto es, en todos los casos, el menor valor obtenido).

6.4. Comparación con otras técnicas

Con el objetivo de obtener una perspectiva básica acerca del rendimiento del agente inteligente desarrollado, es interesante comparar su eficacia, a la hora de resolver el videojuego implementado, con la de otras técnicas comúnmente utilizadas en agentes inteligentes. En esta línea, y a modo de análisis comparativo preliminar, se han implementado también dos nuevos agentes que utilizan, respectivamente, una implementación de Monte Carlo Tree Search (MCTS) [36] y un modelo de Aprendizaje por Refuerzo (AR) [37], técnicas que han sido ya utilizadas, con buenos resultados, en el desarrollo de agentes para videojuegos [38, 39, 40]. En estos dos métodos, a diferencia de los ya vistos en este trabajo, la eficacia a la hora de resolver el problema depende críticamente del tiempo computacional que se les proporcione; esto se debe a la naturaleza de dichas técnicas, pues ambas se basan en un proceso iterativo que aumenta la eficacia del agente conforme más iteraciones se realice (en el caso de MCTS, se puede construir un mayor árbol de estados; en el caso de AR, se puede realizar un mayor entrenamiento).⁴

Bajo este pretexto, sabiendo que en este TFG se utiliza un ordenador de gama media y que este estudio es un análisis preliminar con afán comparativo, se ha creado un nuevo nivel, más simple, que mantiene las características del videojuego pero reduce enormemente el espacio de estados respecto a los ya utilizados (Figura 6.8). Esto tiene un objetivo doble: reducir el tiempo de procesamiento por acción y acortar el número de acciones necesarias para terminar una partida o simulación, disminuyendo así considerablemente el tiempo computacional empleado por estos algoritmos.



Figura 6.8: Mapa del nuevo nivel simple creado. Su tamaño es de 5x8 (40 casillas). Se ha reducido al mínimo la búsqueda de recursos, para que los algoritmos puedan alcanzarlos fácilmente y se centren en resolver el problema de precedencias y optimización. Para que el problema de búsqueda no desaparezca completamente, se ha ubicado la salida relativamente lejos, de forma que no sea trivial llegar a ella.

⁴Para mayor claridad, el funcionamiento de ambas técnicas es introducido, de forma general, en las próximas secciones, correspondientes a la experimentación realizada con cada una de ellas.

Para poder comparar los resultados, primero se ha ejecutado el agente inteligente desarrollado con los diferentes algoritmos de BH implementados, obteniendo así su rendimiento en este nuevo mapa. Los resultados se muestran en la Tabla 6.13.

Resultados obtenidos por el agente inteligente desarrollado

Algoritmo	Ticks	Pts	Recursos
A*	75	18	1, 3, 4, 5 y 6
D* Lite	56	18	1, 3, 4, 5 y 6
RTA*	55	18	1, 3, 4, 5 y 6
LRTA*	52	18	1, 3, 4, 5 y 6
LRTA*(1)	52	18	1, 3, 4, 5 y 6
LRTA*(10)	55	18	1, 3, 4, 5 y 6
LRTA*(100)	55	18	1, 3, 4, 5 y 6

Tabla 6.13: Resultados obtenidos por el agente inteligente desarrollado en el nivel simple creado, con cada uno de los algoritmos de BH implementados. Se muestra el número de ticks empleados, la puntuación (Pts) obtenida y el subconjunto de recursos obtenido en cada ejecución.

6.4.1. Monte Carlo Tree Search

MCTS [36] es una familia de algoritmos de toma de decisiones cuya metodología general consiste en construir un árbol de búsqueda mediante la toma de decisiones aleatorias. Esto se consigue a través del proceso iterativo mostrado en la Figura 6.9. En él, para cada iteración se realizan los cuatro pasos explicados a continuación:

1. **Selección.** Se selecciona un nodo concreto del árbol. El criterio utilizado a la hora de seleccionar un nodo es la llamada política del árbol, y es propia de cada algoritmo de MCTS concreto.
2. **Expansión.** Se añade al árbol un nodo hijo suyo, seleccionado aleatoriamente.
3. **Simulación.** Partiendo de dicho nodo, realiza una secuencia de acciones, generalmente seleccionadas de forma aleatoria, hasta que se alcanza un nodo terminal, que finaliza la partida una puntuación concreta.
4. **Retropropagación.** Se actualiza el valor que tienen el nodo seleccionado y sus antecesores. La forma de valorar un nodo n suele ser mediante la puntuación media, que es una división $Q(n)/N(n)$ donde $N(n)$ es el número de iteraciones en las que se ha visitado el nodo n y

$Q(n)$ es el valor acumulado de las puntuaciones alcanzadas en dichas iteraciones. Por ejemplo, si se ha pasado por cierto nodo en 5 iteraciones y las recompensas obtenidas en ellas han sido 1, 4, 0, 2 y 2 puntos, su valor será de $9/5 = 4.5$.

Un agente inteligente que utiliza MCTS realiza este procedimiento antes de cada acción y, tras finalizar las iteraciones, selecciona la acción que lleve al nodo, hijo del actual, que mayor valor haya obtenido.

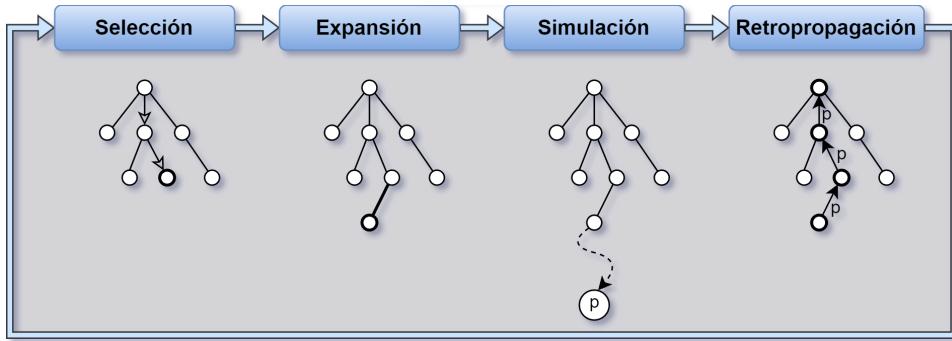


Figura 6.9: Esquema representativo del proceso iterativo de MCTS. Consta de un bucle en el que, en cada iteración se realizan cuatro pasos: selección, donde las flechas representan el proceso de selección desde el nodo raíz hasta el nodo elegido, que está resaltado; expansión, donde un nuevo nodo, también resaltado, es añadido al árbol; simulación, representada por la flecha discontinua, hasta alcanzar un nodo terminal que obtiene una puntuación p ; y retropropagación, donde las flechas representan el proceso de propagación de la puntuación p , y los valores de los nodos resaltados son actualizados.

De cara a este estudio preliminar se ha utilizado una implementación, ya proporcionada por el entorno GVGAI, del algoritmo MCTS *Upper Confidence Bounds for Trees* (UTC) [36], uno de los algoritmos más utilizados de esta familia.⁵ En él, la política empleada consiste en, estando en un nodo n , seleccionar el nodo hijo n' que maximice la siguiente expresión:

$$UCT = \bar{X}' + C \sqrt{\frac{2 \ln N(n)}{N(n')}}$$

donde $N(n)$ es el número de veces que el nodo n ha sido visitado, \bar{X}' es el valor del nodo n' ($Q(n')/N(n')$), normalizado en el rango $[0, 1]$ y $C > 0$ es una constante.

⁵La implementación proporcionada tiene el procesamiento originalmente limitado a un tiempo de ejecución concreto. Esto ha sido modificado para que el límite sea un número concreto de iteraciones, de forma que los resultados no sean dependientes de la máquina concreta en la que se ejecute el algoritmo.

A la hora de seleccionar el número de iteraciones a realizar, se ha utilizado como referencia el artículo *Ordinal Monte Carlo Tree Search* [41], en el que se experimenta con MCTS en diferentes videojuegos implementados en GVGAI. En él se puede ver que este algoritmo, con 10.000 iteraciones por ejecución, obtiene buenos resultados en todos los juegos probados, por lo que este es el valor utilizado para *El Mochilero*.

Bajo estas condiciones, se han realizado 10 ejecuciones del algoritmo (Tabla 6.14) y se observa que el agente siempre llega satisfactoriamente a la salida, con 29.3 ticks de media, y obteniendo siempre los recursos 1, 2, 3, 4 y 6. Este conjunto no proporciona la mejor puntuación posible, por lo que se puede concluir que este algoritmo obtiene peores resultados que el agente desarrollado, al no ser capaz de alcanzar el conjunto óptimo de recursos.

Resultados obtenidos utilizando MCTS

Ejecución	Ticks	Pts	Recursos
1	25	15	1, 2, 3, 4 y 6
2	33	15	1, 2, 3, 4 y 6
3	32	15	1, 2, 3, 4 y 6
4	26	15	1, 2, 3, 4 y 6
5	40	15	1, 2, 3, 4 y 6
6	28	15	1, 2, 3, 4 y 6
7	26	15	1, 2, 3, 4 y 6
8	27	15	1, 2, 3, 4 y 6
9	29	15	1, 2, 3, 4 y 6
10	27	15	1, 2, 3, 4 y 6
Promedio	29.3	15	

Tabla 6.14: Resultados obtenidos utilizando MCTS en el nivel simple creado, a lo largo de 10 ejecuciones diferentes. Se muestra el número de ticks empleados, la puntuación (Pts) obtenida y el subconjunto de recursos obtenido en cada ejecución.

6.4.2. Aprendizaje por Refuerzo

AR [37] es una rama del aprendizaje automático en la que se entrena a un agente inteligente para que aprenda una política (esta es, una estrategia de juego) con el objetivo de maximizar la recompensa total obtenida. Para ello, es necesario que tras cada acción el entorno proporcione, además del nuevo estado del mundo, la recompensa obtenida r , que se define como "la recompensa por pasar del estado anterior s' al nuevo estado s mediante la acción a ". En este contexto, el entrenamiento es un proceso iterativo en el que, en cada iteración, el agente utiliza la nueva información obtenida del entorno (s, r) para actualizar su política, tal y como se muestra en la Figura

6.10. La forma en la que se actualiza y se define dicha política son las que definen un agente de AR concreto. Una vez entrenado, el agente inteligente juega utilizando la política aprendida.

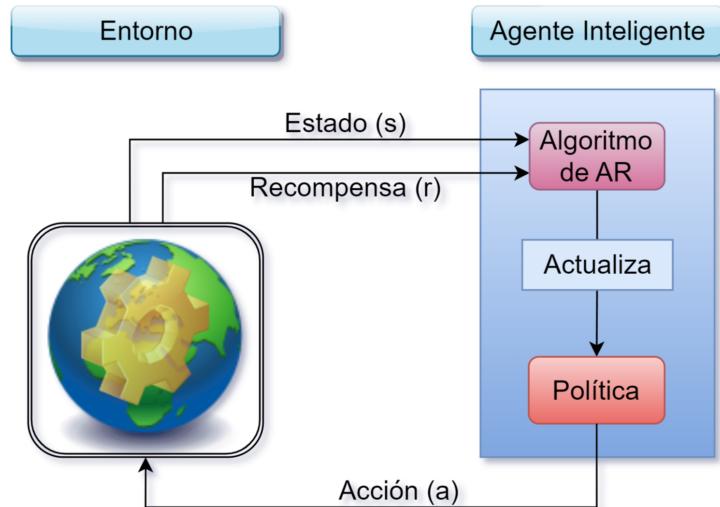


Figura 6.10: Modelo general de entrenamiento de un agente mediante AR. En cada iteración, el agente inteligente recibe del entorno el estado s y la recompensa r y la utiliza para actualizar su política, mediante la cual determina, dado el estado s , la mejor acción a a realizar.

De cara a este estudio preliminar, se ha decidido utilizar Deep Q-Learning (DQL) [42], pues se ha comprobado que obtiene buenos resultados en videojuegos de este tipo [43]. La estructura básica de un agente inteligente en este subgrupo del AR se muestra en la Figura 6.11. Esta se cimenta sobre el uso de la función $Q(s, a)$, que es una estimación de la recompensa final que potencialmente se alcanzará si, dado un estado s , se realiza la acción a . La política en estos casos suele ser, dado un estado s , elegir la acción que maximice $Q(s, a)$. En estos algoritmos, el objetivo es maximizar la calidad de las estimaciones de Q , para lo que se entrena una red neuronal a lo largo de las iteraciones.

Para la implementación de técnicas de AR en GVGAI, se han empleado OpenAI Gym [14], API para el desarrollo agentes de AR; y el proyecto GVGAI_GYM [44], que hace de interfaz entre el entorno GVGAI y dicha API. Tras instalarlas, e introducir en GVGAI_GYM el videojuego creado y las interacciones necesarias para su funcionamiento, la puesta en marcha de la nueva interfaz está finalizada. Por último se ha seleccionado, como agente a utilizar, un modelo que ya ha demostrado obtener buenos resultados en el videojuego *Breakout* de la Atari [45].

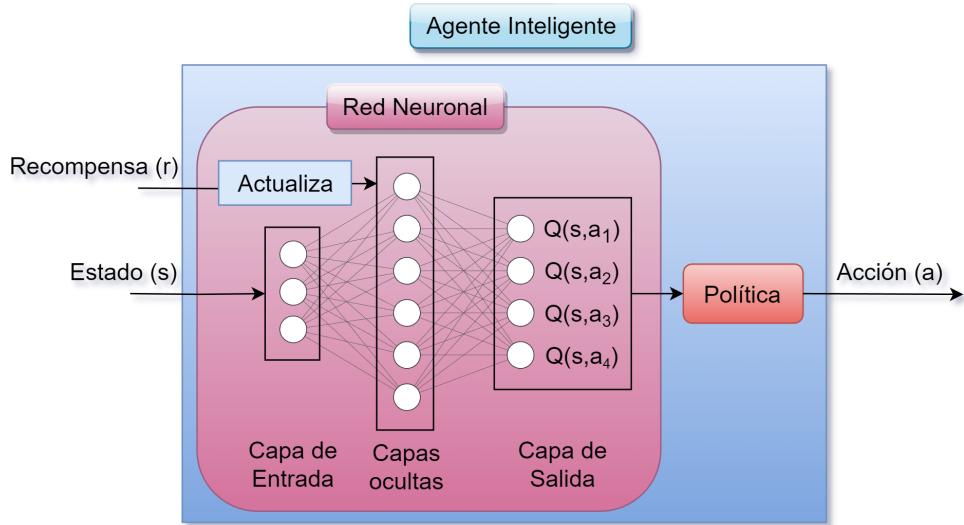


Figura 6.11: Estructura básica de un agente inteligente que implementa DQL. Consta de una red neuronal que, dado un estado s , genera los valores $Q(s, a_i)$ para todo i . Estos son utilizados como parte de la política del agente para seleccionar la próxima acción a . En el entrenamiento, la recompensa r se utiliza para actualizar los pesos de la red neuronal.

Tras más de 100.000 iteraciones de entrenamiento y un total de 67 partidas (a lo largo de las iteraciones, cada vez que se finaliza una partida se inicia una nueva), se ha comprobado que el agente inteligente consigue obtener el conjunto óptimo de recursos (1, 3, 4, 5 y 6) y alcanzar la salida en 857 ticks. Estos resultados son mejores que los obtenidos por MCTS, y demuestran gran potencial, pues el agente ha aprendido a obtener el conjunto óptimo de objetos. No obstante, su rendimiento sigue siendo inferior al agente inteligente desarrollado, pues emplea del orden de 10 veces más ticks que dicho agente inteligente con cualquiera de los algoritmos de BH implementados.

Capítulo 7

Conclusiones y Trabajos Futuros

RR, PA y BH son tres ramas clásicas dentro de la IA. No obstante, su uso de forma híbrida y coordinada no es tan común (véase un análisis bibliográfico del uso de ellas de forma aislada y simultánea en la Sección 1.2). En este TFG se ha presentado un agente inteligente que, gracias a la hibridación de estas técnicas, es capaz de resolver eficazmente un problema complejo. Dicho problema ha sido modelado como un videojuego, que presenta un laberinto con visibilidad parcial y una serie de objetos, con un peso y valor concretos, sobre los cuales se define un orden de precedencia de forma que, para que el jugador pueda obtener un objeto concreto, es necesario que haya obtenido ciertos objetos con anterioridad. A esto se suma la existencia de una restricción sobre el peso máximo de los objetos que puede portar el jugador. En este contexto, el objetivo del juego es salir del laberinto con el conjunto de objetos óptimo, en el menor tiempo posible. Para jugar eficazmente, el agente inteligente desarrollado divide el problema en tres partes diferenciadas, resueltas secuencialmente mediante una proceso de Optimización → Planificación → Búsqueda: primero se calcula el conjunto de objetos objetivo (mediante RR); a continuación, se obtiene el plan (secuencia de acciones) para obtener dicho objetivo (mediante PA); y, finalmente, se alcanzan secuencialmente los sucesivos objetivos parciales que componen el plan a cumplir (mediante BH).

Para este TFG, se ha realizado un proceso de documentación e investigación con el objetivo de ampliar el conocimiento sobre cada una de las ramas exploradas (RR, PA y BH) y sus herramientas existentes, así como entornos de desarrollo de videojuegos y agentes inteligentes (Capítulos 3 y 4). Además, se han desarrollado nuevas competencias en el uso de MiniZinc, PDDL y GVGAI, teniendo que afrontar nuevos retos como la creación de un videojuego en VGDL (Sección 5.2.1), la implementación de interaccio-

nes en GVGAI (Sección 5.2.1.4) y la comunicación con MiniZinc y PDDL desde Java, con el correspondiente procesamiento automático de datos que requiere dicha comunicación (Sección 5.2.2.2). Asimismo, se ha explorado la implementación de algoritmos de BH en nuevos contextos (Sección 5.2.2.3), tanto algoritmos introducidos en el Grado en Ingeniería Informática (A^* , RTA^* , $LRTA^*$), como algoritmos no estudiados con anterioridad (D^* Lite y $LRTA^*(k)$). Todo el trabajo desarrollado se pueden consultar en el siguiente repositorio de GitHub: <https://github.com/Corkiray/TFG>.

Los objetivos planteados en este TFG se han cumplido de forma satisfactoria: el videojuego ha sido desarrollado con éxito, con las características inicialmente propuestas (hay una visión parcial del laberinto, los objetos tienen pesos y valores concretos, hay un orden de precedencia entre ellos, y el jugador tiene un peso máximo que puede almacenar); el agente ha sido implementado acorde a las características deseadas, utilizando RR, PA y BH para resolver el videojuego creado; por último, se ha realizado una amplia validación experimental con la que se ha comprobado que dicho agente juega eficazmente, siendo capaz de, en todos los mapas creados, obtener el conjunto óptimo de objetos y salir del laberinto; obteniendo un rendimiento superior a otras técnicas como Monte Carlo Tree Search (MCTS) o Deep Q-Network (DQN).

Mediante la experimentación, se ha analizado el comportamiento de los diferentes algoritmos de BH implementados, extrayendo las siguientes conclusiones:

- D^* Lite es el que encuentra el camino más corto de todos, siendo siempre el algoritmo que menos ticks tarda en resolver el videojuego (véase Sección 6.3.2). En promedio, emplea un 13% menos ticks que A^* (el segundo que menos ticks emplea) y hasta un 86% menos que $LRTA^*$ (el que más emplea de todos). Esto se debe a que es un algoritmo que reacciona inmediatamente a los cambios de forma que, cada vez que recibe información nueva, recalcula, al completo, un plan óptimo a seguir.
- A^* es el algoritmo que menos tiempo total de ejecución requiere (véase Sección 6.3.2). Es, en promedio, un 60% más rápido que RTA^* (el segundo más rápido) y un 88% más rápido que D^* Lite (el más lento de todos). Esto se debe, principalmente, a que es, con diferencia, el algoritmo que menos veces replanifica. En el caso de D^* Lite, al contrario, a pesar de utilizar menos ticks que el resto de algoritmos, el elevado número de replanificaciones repercute enormemente en el tiempo total de ejecución, consiguiendo este resultado tan negativo.
- Los algoritmos de BTR cumplen con el papel para el que fueron di-

señados: requieren poco tiempo de ejecución por plan.¹ RTA* y LRTA* son, de media, 15 veces más rápidos que A* por plan, y 20 veces más rápidos que D* Lite (véase Sección 6.2.2.2). De estos, RTA* es el más recomendable si se quiere minimizar el tiempo por plan, pues la diferencia de tiempo con LRTA* es de aproximadamente un 10% pero encuentran las soluciones en, de media, un 38% menos pasos.

- LRTA*(k) permite ajustar con el balance “tiempo por plan – número de ticks”, mediante el valor de k (véase Secciones 6.2.2.2 y 6.2.2.5). A mayor valor, mayor es el tiempo por ejecución, pero a cambio se acelera la convergencia hacia el camino óptimo, encontrando en menos ticks el camino deseado. En promedio, se ha obtenido que con $k = 1$ los resultados son similares a LRTA*; con $k = 10$, aumenta un 50% el tiempo por plan, y se reduce un 40% el número de ticks empleados; y con $k = 100$, aumenta un 70% el tiempo por plan y se reduce un 55% el número de ticks. Si se quiere un algoritmo de rápido procesamiento por tick, pero se tiene cierto margen de tiempo para procesar, con LRTA*(k), se puede adaptar el tiempo de ejecución al tiempo de procesamiento disponible para disminuir el número de ticks empleados.

En resumen, este agente es capaz de resolver eficazmente el videojuego creado, utilizando cualquiera de los diferentes algoritmos de BH planteados. Además, cada uno de ellos tiene unas características particulares que lo pueden hacer más adecuado para una situación u otra. El uso de uno en concreto dependerá de las virtudes que el usuario quiera priorizar: A*, cuando el mayor condicionante es el tiempo de procesado; D* Lite, cuando se quiere priorizar el camino más corto; RTA* si se quiere minimizar el tiempo de reacción; y LRTA*(k) si se desea hacer un balance entre tiempo por plan vs número de ticks empleados para resolver el problema.

De cara a trabajos futuros, en base a este TFG se pueden crear nuevos agentes inteligentes que trabajen con diferentes formas de interacción entre las ramas integradas, utilizando para ello las interfaces desarrolladas. También se puede adaptar el sistema para operar en el mundo que se desee, definiendo los archivos de modelo, dominio y configuración para resolver el problema o videojuego correspondiente. Para ello, también sería interesante ampliar la generalización del sistema, añadiendo una mayor automatización y simplicidad al proceso de creación de estos archivos. En un futuro, este sistema, ampliado y con decenas de juegos adaptados y múltiples agentes integrados, podría ser utilizado como un sólido framework para el desarrollo y experimentación con agentes inteligentes cuyo núcleo sea la hibridación de diferentes ramas de la IA.

¹ A diferencia del punto anterior, en este punto se está hablando del tiempo de ejecución medio por plan, es decir, el tiempo total de ejecución del algoritmo, dividido entre el número de veces que ha planificado.

Bibliografía

- [1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 4 ed., 2020.
- [2] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier Science Inc., 2006.
- [3] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.
- [4] M. Ghallab, D. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.
- [5] S. Edelkamp and S. Schrödl, *Heuristic Search: theory and applications*. Morgan Kaufmann Publishers Inc., 2011.
- [6] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, *et al.*, “Manifesto for agile software development.” <https://agilemanifesto.org>, 2001. Accedido: 2023-07-03.
- [7] “¿Qué es el desarrollo de Agile?” <https://learn.microsoft.com/es-es/devops/plan/what-is-agile-development>. Accedido: 2023-07-02.
- [8] “Flujo de trabajo de Gitflow: Atlassian Git Tutorial.” <https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>. Accedido: 2023-06-15.
- [9] “A constraint modelling and solving toolchain.” <https://constraintmodelling.org/>. Accedido: 2023-06-15.
- [10] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, “PDDL - the planning domain definition language”, *Technical Report, Tech. Rep.*, 1998.
- [11] J. Hoffmann, “FF: The fast-forward planning system”, *AI magazine*, vol. 22, no. 3, pp. 57–62, 2001.

- [12] J. Hoffmann, “The Metric-FF planning system: Translating “ignoring delete lists” to numeric state variables”, *Journal of artificial intelligence research*, vol. 20, pp. 291–341, 2003.
- [13] C. Bamford and J. Potter, “Griddly.” <http://griddly.com>, 2023. Accedido: 2023-06-13.
- [14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym”, 2016. arXiv preprint arXiv:1606.01540.
- [15] “Pygame.” <https://www.pygame.org/>, 2023. Accedido: 2023-06-13.
- [16] M. Chevalier-Boisvert, L. Willems, and S. Pal, “Minimalistic gridworld environment for gymnasium.” <https://github.com/Farama-Foundation/Minigrid>, 2018. Accedido: 2023-06-13.
- [17] C. Beattie, T. Köppe, E. A. Duéñez-Guzmán, and J. Z. Leibo, “Deepmind lab2d.” <https://github.com/deepmind/lab2d>, 2020. Accedido: 2023-06-15.
- [18] GAIGResearch, “Tabletop games framework.” <https://github.com/GAIGResearch/TabletopGames>, 2023. Accedido: 2023-06-15.
- [19] GAIGResearch, “Stratega.” <https://github.com/GAIGResearch/Stratega>, 2022. Accedido: 2023-06-15.
- [20] GAIGResearch, “GVGAI.” <https://github.com/GAIGResearch/GVGAI>, 2021. Accedido: 2023-06-15.
- [21] V. N. Vasilev, “Desarrollo de una arquitectura reactiva y deliberativa usando planificación en el entorno de juegos GVGAI”, *Trabajo Fin de Grado. Universidad de Granada*, 2020.
- [22] “MiniZinc software.” <https://www.minizinc.org>, 2023. Accedido: 2023-06-16.
- [23] “Basic modelling in MiniZinc.” <https://www.minizinc.org/doc-2.7.5/en/modelling.html#basic-structure-of-a-model>, 2020. Accedido: 2023-06-16.
- [24] “PDDL domain.” <https://planning.wiki/ref/pddl/domain>, 2023. Accedido: 2023-06-17.
- [25] “PDDL problem.” <https://planning.wiki/ref/pddl/problem>, 2023. Accedido: 2023-06-17.
- [26] C. Muise, “Planning.Domains.” <http://planning.domains>, 2015. Accedido: 2023-06-17.

- [27] E. Scala, P. Haslum, S. Thiébaux, and M. Ramirez, “Interval-based relaxation for general numeric planning”, in *Proceeding of the European Conference on Artificial Intelligence*, pp. 655–663, 2016.
- [28] M. Helmert, “The fast downward planning system”, *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.
- [29] N. J. Nilsson, *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.
- [30] S. Koenig and M. Likhachev, “Incremental A*”, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 1539–1546, 2001.
- [31] A. Stentz *et al.*, “The focussed D* algorithm for real-time replanning”, in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1652–1659, 1995.
- [32] S. Koenig and M. Likhachev, “Fast replanning for navigation in unknown terrain”, *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.
- [33] R. E. Korf, “Real-time heuristic search”, *Artificial intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990.
- [34] C. Hernández and P. Meseguer, “LRTA*(k)”, in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1238–1243, 2005.
- [35] T. Schaul, “A video game description language for model-based or interactive learning”, in *Proceedings of the Conference on Computational Intelligence in Games*, pp. 193–200, 2013.
- [36] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods”, *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.
- [37] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [38] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, “Mastering the game of go with deep neural networks and tree search”, *Nature*, vol. 529, pp. 484–489, 2016.

- [39] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play”, *Science*, vol. 362, pp. 1140–1144, 2018.
- [40] I. Szita, G. Chaslot, and P. Spronck, “Monte-carlo tree search in settlers of catan”, in *Advances in Computer Games*, (Berlin, Heidelberg), pp. 21–32, Springer Berlin Heidelberg, 2010.
- [41] T. Joppen and J. Fürnkranz, “Ordinal Monte Carlo Tree Search”, in *Monte Carlo Search International Workshop*, pp. 39–55, 2020.
- [42] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters”, in *Proceedings of the Association for the Advancement of Artificial Intelligence conference on artificial intelligence*, vol. 32, 2018.
- [43] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning”, *arXiv preprint arXiv:1312.5602*, 2013.
- [44] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, “Deep reinforcement learning for general video game ai”, in *Proceedings of the Conference on Computational Intelligence and Games*, pp. 1–8, 2018.
- [45] J. Chapman and M. Lechner, “Deep Q-Learning for Atari Breakout .” https://keras.io/examples/r1/deep_q_network_breakout/. Accedido: 2023-07-15.

