View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidad Carlos III de Madrid e-Archivo

Universidad Carlos III de Madrid

Escuela politécnica superior

Grado en Ingeniería Informática

Proyecto Fin de Grado

Resolución del juego Sokoban con técnicas de búsqueda

Autor: Rubén Majadas Sanz

Tutor: Carlos Linares López

Agradecimientos

Quiero aprovechar esta oportunidad para agradecer a todas las personas que el apoyo que me ha ayudado a llevar a cabo este proyecto.

A mi familia. En especial a mis padres, por todas las oportunidades que me han brindado y el apoyo incondicional recibido. A mi hermana, por su confianza y ayudarme a tomar siempre las decisiones acertadas. A mis abuelas por el cariño y la fe que siempre han tenido.

A Ana por haber compartido conmigo momentos únicos y ayudarme a ser mejor persona.

A todos mis amigos con los que gracias a ellos he conseguido pasar siempre buenos momentos. A Laura y Rafa, por crecer con ellos y ser mis confidentes. A Lorena y Ángel, porque aun estando lejos, son una gran ayuda.

A los compañeros de universidad que a lo largo de esta experiencia se han convertido en amigos.

A todos los profesores que he tenido a lo largo de mi formación y que han conseguido, además de educarme, transmitir una motivación para continuar investigando por mi propia cuenta. En especial a mi tutor, Carlos Linares López, por su tiempo y colaboración, su entusiasmo me ha ayudado a encontrar la motivación necesaria para llevar a cabo este proyecto.

Muchas gracias a todos.

Rubén Majadas Sanz

Resumen

La búsqueda es una rama principal de la Inteligencia Artificial (IA) que se encarga de encontrar solución a los problemas mediante su representación en diferentes estados. Los juegos siempre han servido como campo de pruebas de estas técnicas.

En esta ocasión se aplica la búsqueda de un solo agente para resolver los niveles del juego Sokoban de forma óptima. Esta tarea es realmente complicada por la gran cantidad de movimientos posibles en cada una de las situaciones del jugador. Además, las soluciones necesitan una larga sucesión de movimientos para encontrar la meta.

A estas dificultades se le suma la posibilidad de alcanzar posiciones que impiden que el problema se pueda resolver, lo que lo complica aún más. Todo esto provoca que sea prácticamente imposible resolver algún nivel con búsqueda.

Para abordar este problema se realiza un estudio exhaustivo de los objetivos que se quieren alcanzar y de las investigaciones anteriores. Esto nos ayuda a establecer todas las funcionalidades necesarias, a plantear las decisiones de diseño adecuadas y a implementar diferentes alternativas para encontrar el mejor agente.

Para solucionar este problema se diseñan diferentes técnicas, como localizar posiciones que impiden alcanzar la meta o eliminar los estados que se repitan a lo largo de la búsqueda, que ayudan a reducir la dimensión del problema.

A continuación, se implementan diferentes algoritmos de búsqueda y las correspondientes heurísticas con el objetivo de alcanzar la solución óptima de cada uno de los niveles.

Además de resolver los problemas durante este proyecto se realiza una comparativa entre los diferentes experimentos realizados con la finalidad de analizar la importancia de las heurísticas.

Una vez obtenidos todos los resultados se elige la mejor configuración encontrada y se analizan las ventajas e inconvenientes que se encuentran, con el objetivo de mejorar en futuros trabajos.

Abstract

Search is a central topic in Artificial Intelligence that is responsible for solving problems through their representation in different states. Games have always been used as a test of these techniques.

In this dissertation, search single agent is applied to optimally solve Sokoban game levels. This is a very complicated task due to the large number of possible moves the player has at every state. In addition, solutions are composed of long sequences of moves to find goal

These difficulties are compounded by the possibility of reaching positions that prevent the problem from being solved, which further complicates this problem. All this makes it virtually impossible to solve some of the levels.

To approach this problem, it is performed a comprehensive study of the objectives to be achieved as well as a study of other previous research. This helps us establish all necessary functionalities, to propose appropriate design decisions and to implement different alternatives to find the best agent.

Initially, different techniques are designed to solve this problem, such as locating reachable positions that prevent from achieving the goal or eliminating repeated states along the search, which helps reduce the dimension of the problem.

Then, different algorithms are implemented with the corresponding search heuristics in order to reach the optimal solution in each level.

In addition to solving the problems, in this project, different experiments are compared in order to analyze the importance of the heuristics.

Once all results have been obtained, the best model is chosen and the advantages and disadvantages are discussed to provide with future work lines.

Índice general

1. Introducción **1**

1.1. Descripción del problema 1

1.2. Motivación 2

1.3. Estructura del documento 3

2. Estado de la Cuestión **5**

2.1. Área de aplicación 5

2.1.1. Inteligencia artificial 5

2.1.2. Búsqueda heurística 6

2.1.3. Búsqueda no informada 8

2.1.4. Heurísticas 11

2.1.5. Búsqueda informada 16

2.1.6. Aplicaciones de la IA en juegos 20

2.2. Sokoban 21

2.3. Nomenclatura 23

2.4. Sokoban como problema de Búsqueda 24

2.5. Estudios previos 25

3. Objetivos del trabajo **29**

4. Desarrollo **31**

4.1. Análisis del sistema 31

4.1.1. Descripción de las características funcionales 31

4.1.2. Restricciones del sistema 32

4.1.3. Entorno operacional 33
4.1.4. Especificación de casos de uso 34
4.1.5. Especificación de requisitos 38
4.2. Diseño del sistema 46
4.2.1. Arquitectura del sistema 46
4.2.2. Descripción general del sistema 47
4.2.3. Decisiones de diseño 50
4.3. Agentes desarrollados 55
4.3.1. Búsqueda a ciegas 55
4.3.2. Búsqueda heurística 56
4.3.3. Otros experimentos 59

5. Resultados **63**
5.1. Experimentos base 63
5.2. Mejoras de los experimentos 67
5.2.1. Distancia Manhattan a la asignación de Kuhn 68
5.2.2. Manhattan con Kuhn 70
5.2.3. Manhattan a las asignaciones y a la meta próxima 72
5.3. Conclusiones de los resultados 74

6. Gestión del proyecto **77**
6.1. Descripción de las fases del proyecto 77
6.2. Planificación 78
6.3. Presupuesto 82
6.3.1. Costes estimados 82
6.3.2. Costes reales 84
6.4. Marco regulador 85

7. Conclusiones **87**
7.1. Conclusiones generales 87
7.2. Conclusiones referentes a los objetivos 89
7.3. Trabajos futuros 90

A. Resultados de los experimentos **93**

B. Summary **107**

Índice de figuras

2.1. Secuencia de nodos visitados en amplitud 9
2.2. Secuencia de nodos visitados en profundidad 10
2.3. Nomenclatura utilizada para representar el nivel 78. 23

4.1. Diagrama de casos de uso 35
4.2. Diagrama de Modelo-Vista-Controlador 46
4.3. Diagrama clases del sistema 48
4.4. Diagrama de flujo del sistema 50
4.5. Diagrama de flujo del algoritmo 51
4.6. Secuencia de acciones que dejan el nivel sin solución. 52
4.7. Ejemplo de deadlocks fijos 53
4.8. Ejemplo de deadlocks dinámicos 53
4.9. Ejemplo de cálculo de la distancia *Manhattan* 57
4.10. Ejemplo de cálculo del algoritmo de *Kuhn* 58
4.11. Ejemplo de cálculo de *Manhattan* a las asignaciones realizadas con *Kuhn* 59
4.12. Ejemplo de cálculo de *Manhattan* o *Kuhn* en función de la profundidad. 60

5.1. Comparativa de nodos expandidos de Manhattan frente a Kuhn . 66
5.2. Comparativa de tiempos de Manhattan frente a Kuhn 67
5.3. Comparativa de nodos expandidos de Manhattan frente a Kuhn . 69
5.4. Comparativa de tiempos de Manhattan frente a Kuhn 69
5.5. Comparativa de nodos expandidos de Manhattan frente a Kuhn . 71

5.6. Comparativa de tiempos de Manhattan frente a Kuhn 71
5.7. Comparativa de nodos expandidos de Manhattan frente a Kuhn . 73
5.8. Comparativa de tiempos de Manhattan frente a Kuhn 73

6.1. Planificación inicial del proyecto 81
6.2. Planificación real del proyecto 81

B.1. Sequence of actions that leave the level unsolvable. 112
B.2. Example of fixed deadlocks 112
B.3. Example of dynamic deadlocks 113
B.4. Comparative nodes expanded of Manhattan against Kuhn 115
B.5. Comparative times of Manhattan against Kuhn 116

Índice de tablas

2.1. Complejidades de los algoritmos de fuerza bruta 12
2.2. Comparativa de las complejidades temporal y espacial 20
2.3. Estadísticas de los *solver* de la página Sokobano 27

4.1. Características del equipo. 33
4.2. Caso de uso modelo 35
4.3. Caso de uso 001 36
4.4. Caso de uso 002 36
4.5. Caso de uso 003 36
4.6. Caso de uso 004 37
4.7. Caso de uso 005 37
4.8. Caso de uso 006 37
4.9. Requisito funcional de ejemplo 40
4.10. Requisito funcional 001 40
4.11. Requisito funcional 002 40
4.12. Requisito funcional 003 41
4.13. Requisito funcional 004 41
4.14. Requisito funcional 005 41
4.15. Requisito funcional 006 42
4.16. Requisito funcional 007 42
4.17. Requisito funcional 008 42
4.18. Requisito funcional 009 43
4.19. Requisito funcional 010 43

4.20. Requisito funcional 011 43
4.21. Requisito funcional 012 44
4.22. Requisito no funcional 001 44
4.23. Requisito no funcional 002 44
4.24. Requisito no funcional 003 45
4.25. Requisito no funcional 004 45
4.26. Requisito no funcional 005 45

5.1. Problemas resueltos con búsqueda 65
5.2. Niveles que no son resueltos por A* con distancia Manhattan . . . 65
5.3. Niveles que no son resueltos por A* con algoritmo de Kuhn 65
5.4. Rendimiento de Kuhn frente a Manhattan 66
5.5. Rendimiento del experimento frente a Kuhn 68
5.6. Rendimiento del experimento frente a Kuhn 70
5.7. Rendimiento del experimento frente a Kuhn 72

6.1. Definición de tiempos y tareas del desarrollo del proyecto. 79
6.2. Definición real de tiempos y tareas del desarrollo del proyecto. . . 80
6.3. Estimación de costes humanos. 82
6.4. Costes humanos añadiendo la Seguridad Social. 83
6.5. Presupuesto estimado de recursos hardware. 83
6.6. Resumen del presupuesto estimado. 83
6.7. Costes humanos reales. 84
6.8. Costes humanos reales añadiendo la Seguridad Social. 84
6.9. Presupuesto real de recursos hardware. 85
6.10. Resumen del presupuesto real. 85

A.1. Resultados con fuerza bruta (Amplitud) 1/3. 94
A.2. Resultados con fuerza bruta (Amplitud) 2/3 95
A.3. Resultados con fuerza bruta (Amplitud) 3/3 96
A.4. Resultados con A* y distancia Manhattan 1/3. 97
A.5. Resultados con A* y distancia Manhattan 2/3. 98

A.6. Resultados con A* y distancia Manhattan 3/3. 99
A.7. Resultados con A* y algoritmo de Kuhn 1/3. 100
A.8. Resultados con A* y algoritmo de Kuhn 2/3. 101
A.9. Resultados con A* y algoritmo de Kuhn 3/3. 102
A.10. Resultados con A* con el segundo experimento 1/3. 103
A.11. Resultados con A* con el segundo experimento 2/3. 104
A.12. Resultados con A* con el segundo experimento 3/3. 105

B.1. Performance of this experiment against Kuhn 116
B.2. Performance against Kuhn 117
B.3. Performance against Kuhn 118

Capítulo 1

Introducción

1.1. Descripción del problema

Sokoban es un videojuego de lógica creado por el japonés Hiroyuki Imabayashi en la década de los ochenta. El objetivo del juego es muy simple, se trata de transportar las cajas, en el menor número de movimientos, hasta ciertas posiciones de un almacén. La única forma de mover las cajas es empujándolas, y solamente se puede mover una caja en cada acción.

Al ser un juego muy intuitivo, tanto los movimientos como las reglas, permiten que cualquier usuario sea capaz de enfrentarse a estos niveles. La colección principal está formada por 90 niveles, en los que la dificultad va aumentando progresivamente. Además, algunos usuarios de la comunidad proporcionan los laberintos que ellos mismos desarrollan, lo que permite que siempre haya algún reto disponible.

El principal objetivo para cualquier jugador es resolver el laberinto, sin embargo, los jugadores más experimentados tratan de batir sus propias puntuaciones, utilizando como medida los empujones o los movimientos del jugador.

A pesar de su apariencia asequible, Sokoban es un verdadero reto de lógica. La dificultad de resolver cada nivel reside en la interacción entre las diferentes cajas de cada laberinto, los múltiples movimientos que pueden realizarse en cada turno y que algunos desplazamientos provocan que el nivel sea imposible de resolver.

En el área de la informática este problema se ha planteado como búsqueda en juegos de un solo agente. Los juegos que se han utilizado siempre para investigar dentro de este área han sido N-puzzle[1] o el cubo de Rubik[2], estos juegos tienen un pequeño factor de ramificación[3] y unas profundidades[4] moderadas a las que se alcanza la solución.

En el juego del Sokoban es más complicado, no solamente por tener un factor de ramificación elevado comparable al de otros juegos como el ajedrez. Además, para encontrar la solución se necesitan muchos empujones. Esta combinación genera un árbol de búsqueda tan grande que complica sustancialmente encontrar la solución óptima[5]. Al analizar la complejidad que tenía se determinó que pertenece a la clase de complejidad PSpace-Complete [1].

1.2. Motivación

La principal motivación para llevar a cabo este proyecto ha sido la oportunidad de desarrollar un programa que resuelva los niveles de este famoso juego. Esta motivación surge como entusiasta de los juegos de lógica y la frustración de quedarse bloqueado en multitud de ocasiones por diferentes motivos. También es una gran ocasión para investigar en la rama que más he disfrutado a lo largo de toda mi experiencia universitaria, la inteligencia artificial.

La gran dificultad que tiene el juego hace que sea un verdadero desafío este proyecto. Debido al enorme espacio de búsqueda, cualquier pequeña mejora que se logre puede suponer un gran avance en la investigación. Aún ni los programas más especializados en resolver este tipo de problemas son capaces de encontrar una solución a los niveles más complicados del juego. Por este motivo, descubrir nuevos métodos para reducir este gran espacio de búsqueda o recorrerlo más rápido, se convierte en un progreso significativo en los problemas de lógica de un solo agente.

[1] `https://en.wikipedia.org/wiki/Sliding_puzzle` visitado 19 Jul. 2016

[2] `https://en.wikipedia.org/wiki/Rubik%27s_Cube` visitado 19 Jul. 2016

[3] Posibles movimientos en cada situación actual

[4] Movimientos que se han realizado desde el inicio del juego

[5] Solución al problema ejecutando el menor número de movimientos de las cajas posible.

Además, el análisis y visualización de los resultados se pueden comprobar en el mismo entorno gráfico que se juega habitualmente.

Y, por último, la capacidad de encontrar situaciones que posteriormente se pueden aplicar en la realidad, como evitar situaciones irrevocables o la toma de decisiones en tiempo real.

1.3. Estructura del documento

A lo largo de esta memoria se explican detalladamente las diferentes fases del proyecto, desde el estudio del problema y desarrollo hasta las pruebas realizadas y el análisis y conclusiones de los resultados obtenidos. Para comprender mejor el contenido, el documento se estructura de la siguiente forma:

- Tras esta *introducción*, se realiza un análisis sobre el *estado de la cuestión* del juego. En primer lugar, se establece el área en el que se estudia este problema, seguido de las reglas del juego, y por último se exponen otros estudios realizados previamente.

- Después, se establecen los *objetivos* que se quieren alcanzar con la investigación en este proyecto.

- A continuación, se describe detalladamente el trabajo realizado. Para ello, en primer lugar se establece un *análisis y diseño del sistema*. Después, se profundiza en los algoritmos y heurísticas utilizadas para llevar a cabo la *implementación de los agentes*.

- Posteriormente, se explican las *pruebas* realizadas para el correcto funcionamiento del sistema y se exponen los resultados obtenidos con cada uno de los diseños aportados.

- Una vez descrito todo el contenido del proyecto, se incorpora en el documento la gestión del proyecto necesaria para llevar a cabo esta investigación. Para ello se describen las fases del proyecto, la planificación y el presupuesto.

Por último, se contemplan las conclusiones alcanzadas, tanto de los resultados como las obtenidas a lo largo del proyecto, además de establecer si los objetivos establecidos en el tercer apartado se han logrado satisfacer. Para terminar, se establecen las futuras lineas de investigación que se pueden seguir tras este estudio.

Capítulo 2

Estado de la Cuestión

En este apartado, se realizará una explicación del área en las que se aplica este proyecto, seguido de una breve descripción del juego, Sokoban, para poder mostrar las conclusiones y resultados obtenidos en las investigaciones realizadas hasta el momento.

2.1. Área de aplicación

En esta sección, se explica el **área** en el que se aplica el proyecto. En primer lugar, se trata la rama principal de la informática en la que se basa este proyecto, la **Inteligencia Artificial**. A continuación, se explican las **técnicas de búsqueda** en las que se fundamenta la investigación. Por último, se mencionan las **aplicaciones de la Inteligencia Artificial en los juegos**, que han servido de precedentes a lo largo de la historia.

2.1.1. Inteligencia artificial

A lo largo de la historia, el ser humano ha tenido siempre la necesidad de comprender su propia inteligencia y reproducirla en diferentes organismos. Esta necesidad, se puede constatar porque en algunos mitos y leyendas se trata de

otorgar a diferentes seres estas cualidades, como en *Galatea*[1], *Golem*[2] o la tan conocida obra de *Frankestein*. Todas estas obras, han sido precursoras de un nuevo género, la ciencia ficción, en el que la IA trata de convertir toda esa ficción en realidad, a pesar de encontrarse bastante lejos de alcanzar estos niveles de similitud con un humano. Por este motivo, en 1950, **Alan Turing** establece un método de medir la semejanza entre el comportamiento de una máquina y un humano [2]. Más adelante, recibiría el nombre de *Test de Turing*[3].

La Inteligencia Artificial (IA) es un campo de la informática en el que se investiga como aportar conocimiento y habilidades a las máquinas, de tal forma que tengan comportamientos similares a los humanos, fundamentalmente se basa en el razonamiento que somos capaces de aplicar. El objetivo es resolver los problemas que se plantean emulando el planteamiento que haría una persona sobre ellos. De esta forma aprovecha la velocidad de cómputo que tienen los ordenadores para adquirir nuevo conocimiento o resolver nuevos problemas [3].

La cantidad de problemas a los que se enfrentan, dentro esta área, es muy diversa, debido a que es multidisciplinar. Todas estas técnicas reúnen características de diferentes ámbitos como optimización matemática, estadística, lógica, filosofía o neurociencia [4].

2.1.2. Búsqueda heurística

La búsqueda es una técnica de la IA y de la optimización matemática que trata de resolver problemas, encontrando la mejor solución o una aproximación a ella. Se basa en expandir cada estado[4] con cada uno de sus posibles operadores[5] para encontrar la secuencia, desde el estado inicial hasta el estado meta, que sea solución del problema.

[1] `https://misterios.co/mitologia-pigmalion-y-galatea/` visitado 19 Jul. 2016

[2] `http://labohemia4.blogspot.com.es/2015/06/mitologia-el-golem.html` visitado 19 Jul. 2016

[3] `https://es.wikipedia.org/wiki/Test_de_Turing` visitado 19 Jul. 2016

[4] Representación de la situación de un problema

[5] Acciones que se puede ejecutar en cada estado

Se puede representar como un grafo[6], que es recorrido normalmente como un árbol. El objetivo es recorrer esta estructura, como indique el algoritmo escogido, para encontrar esta solución o estado final. El tamaño de este espacio de búsqueda[7] dependerá de dos atributos, el factor de ramificación[8], b, y la profundidad[9], d, a la que se encuentre la solución.

Por este motivo, el máximo espacio de búsqueda vendrá determinado siempre por la expresión:

$$b^d$$

En ocasiones la búsqueda heurística se convierte en el único método viable para resolver los problemas de optimización más complejos que se pueden encontrar constantemente en aplicaciones del mundo real.

Con el fin de poder generar una comparativa y verificar la utilidad entre los diferentes algoritmos, se analizan las siguientes características:

- **Completo**: Si existe una solución y se garantiza que la encuentra.
- **Admisible**: Si además de hallar la solución, se asegura que la solución es óptima.
- **Eficiencia**: Medida, tanto en tiempo como en memoria, del coste de resolver el problema.
- **Problemas**: Los inconvenientes que se encuentran al utilizar el algoritmo elegido.

Encontrar la solución dentro de esta gran estructura es una tarea demasiado compleja. Por este motivo, en muchas ocasiones es necesario guiar el recorrido para poder alcanzar mucho antes la solución.
Según el grado de información que se tiene al recorrer la estructura, la búsqueda se puede dividir en:

[6]Representación de estados como un conjunto finito de vértices unidos por arcos o aristas
[7]Conjunto de estados
[8]Número de estados, que se obtienen a partir de un estado padre
[9]Acciones que se han realizado desde el estado inicial

- **No informada**: Cuando no se dispone de ninguna información que se use como criterio para escoger el siguiente nodo a expandir. También se conoce como búsqueda a ciegas o por fuerza bruta.

- **Informada**: Cuando se conoce información total o parcial del dominio que ayuda a encontrar la solución del problema. También se conoce como búsqueda heurística

2.1.3. Búsqueda no informada

La búsqueda no informada o por fuerza bruta utiliza algoritmos que recorren completamente el espacio de búsqueda. Utilizan métodos generales, independientemente del problema. Esto se debe a que no se tiene ningún conocimiento del dominio y por lo tanto es imprescindible comprobar si entre todos los estados existe la solución. Como estos métodos son exhaustivos se requiere una gran cantidad de recursos, en cuanto se halla una solución dan por concluida la búsqueda, incluso si esta no es la mejor.

En el peor de los casos, estos algoritmos deben recorrer todo el espacio de búsqueda. Pero se garantiza la completitud de todos estos algoritmos, es decir, que si existe una solución la encuentra. Por otro lado, la admisibilidad, no se puede garantizar ya que depende del propio algoritmo.

Hay diferentes algoritmos aplicables a la búsqueda a ciegas, aunque todos ellos se basan prácticamente en dos estrategias fundamentales, amplitud o profundidad.

2.1.3.1. Primero en Amplitud

El algoritmo de primero en amplitud, o *breadth first search*, recorre el espacio de búsqueda por niveles de profundidad. Basándose en la idea intuitiva de recorrer una lista en el orden que se van generando los diferentes nodos. De esta forma no se expande ningún nodo con una profundidad mayor a la actual, hasta que no se han terminado de visitar todos los nodos a dicha profundidad. Así este

algoritmo además de garantizar que es completo, también es admisible. La solución que encuentra es óptima [5]. Se puede observar un ejemplo de ejecución de este algoritmo en el árbol de búsqueda mostrado en la figura 2.1.

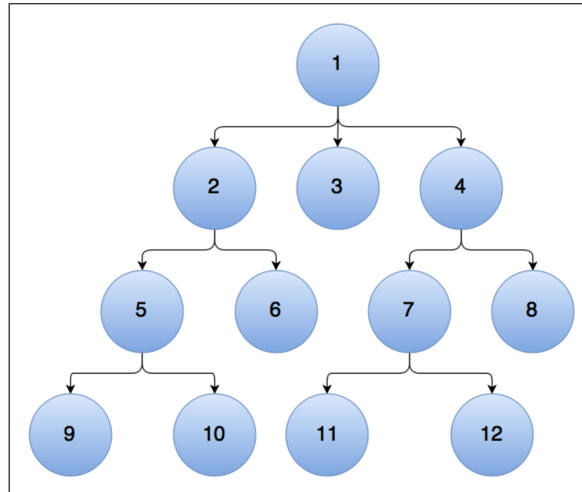


Figura 2.1: Secuencia de nodos visitados en amplitud

La estructura de datos utilizada para implementar este algoritmo es una cola en la que se van insertando los sucesores al final de la estructura y en cada paso del algoritmo se expande el que se encuentra en primera posición. Por lo tanto sigue una política **FIFO**[10], donde el primer elemento insertado en la estructura es el primero en salir.

Las características de este algoritmo, se pueden resumir en los siguientes apartados:

- **Completo**: Si el factor de ramificación es finito, encuentra la solución si existe.
- **Admisible**: Si el coste de todos los operadores es el mismo, entonces la solución hallada es óptima.
- **Eficiencia**: La complejidad es exponencial. Si el problema es pequeño, se puede asumir la ocupación de memoria. En cualquier otro caso, este consumo

[10]Siglas para representar *First In First Out*, El primer elemento insertado es el primero en eliminarse de la estructura.

resulta prohibitivo.

- **Problemas**: Tanto el tiempo como la memoria se consume exponencialmente, pero en estos casos, en los que la complejidad espacial es la misma que la temporal, se agota normalmente la memoria antes que el tiempo.

2.1.3.2. Primero en Profundidad

El algoritmo de primero en profundidad, o *depth-first search*, examina por completo el espacio de búsqueda siguiendo siempre un camino, hasta que encuentra una solución o el camino llega a su fin, en este caso regresa a un nivel superior, *backtracking*, y selecciona otro posible camino con el objetivo de alcanzar la meta. Este algoritmo no garantiza que si existe una solución es capaz de encontrarla, esto se debe al parámetro de profundidad máxima que explorar, por lo tanto, no es completo. Por otro lado, no se puede garantizar que la solución hallada sea óptima, puesto que se expanden nodos a niveles de profundidad superiores sin tener en cuenta que otro camino alcance la meta en menos pasos. Se puede observar un ejemplo de ejecución del algoritmo de profundidad en en el árbol de búsqueda mostrado en la figura 2.2.

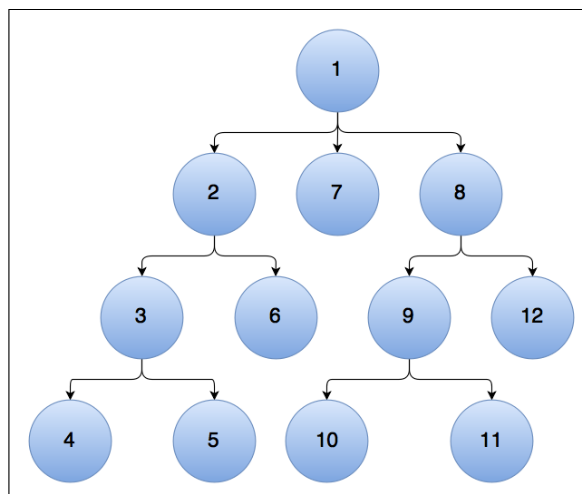


Figura 2.2: Secuencia de nodos visitados en profundidad

La estructura de datos que se utiliza para implementar este algoritmo es una pila en la que se apilan los nuevos elementos y a continuación se selecciona el nodo

a expandir que se encuentra en la cima de la estructura. Por lo tanto sigue una política **LIFO**[11], donde el primer elemento insertado en la estructura es el último en salir.

Las características de este algoritmo, se pueden resumir en los siguientes apartados:

- **Completo**: No es completo, no se puede establecer el parámetro de profundidad máxima.
- **Admisible**: No se garantiza que la solución encontrada sea la óptima.
- **Eficiencia**: Buena cuando las metas están alejadas del estado inicial.
- **Problemas**: Si no se establece un límite puede que un camino seleccionado no tenga fin o consuma todo el tiempo y memoria sin analizar otras ramas. Por otro lado, no garantiza que la solución encontrada sea óptima.

2.1.4. Heurísticas

Una heurística, dentro de la IA, es una estimación sobre el coste para alcanzar una solución del problema. Este valor se utiliza como una guía para seleccionar el siguiente estado a expandir, con el objetivo de que cada estado seleccionado se aproxime cada vez más a la meta.

Las heurísticas se utilizan cuando se dispone de información parcial del problema. Esto sirve para mejorar los resultados, tanto en tiempo como en memoria, que se obtienen al utilizar algoritmos por fuerza bruta, mostrados en la tabla 2.1, en los que no se utiliza ningún conocimiento sobre el dominio.

La calidad de la heurística viene dada por la precisión de las estimaciones que hace al entorno. Por lo que será mejor cuanta mayor sea la información que aporte. Sin embargo, la función heurística se evalúa en cada estado que se genera, por esta razón este cálculo tiene que ser lo más rápido posible. El objetivo se convierte en

[11] Siglas para representar *Last In First Out*, El primer elemento eliminado es el último insertado.

		Algoritmo	
		Amplitud	Profundidad
Complejidad	Temporal	O(b^d)	O(b^d)
	Espacial	O(b^d)	O(d)

Tabla 2.1: Complejidades de los algoritmos de fuerza bruta

encontrar un equilibrio entre la eficiencia y la precisión de las estimaciones que hace.

Una de las formas más comunes de obtener estas funciones heurísticas es mediante **relajación de restricciones**. Esta técnica consiste en eliminar una o varias restricciones del problema original y realizar una estimación óptima de los pasos necesarios para resolver el problema modificado. De esta forma se obtiene una estimación que permite realizar cálculos más veloces y unas estimaciones más precisas que si se aplicaran otras funciones generales, aplicables prácticamente a cualquier problema.

Una de las principales características de la función heurística es la admisibilidad, esto permite a los algoritmos que utilizan estas estimaciones encontrar las soluciones óptimas para el problema que se plantea. Esta cualidad se cumple siempre que los valores obtenidos por la función, nunca sobrestimen el esfuerzo real para llegar a la meta. Por lo tanto, una heurística, *h(n)*, es admisible si se cumple que para todos sus valores calculados nunca se supera el coste real, h^*, para alcanzar la meta desde el estado al que se aplica. Matemáticamente se representa de la siguiente forma:

$$h(n) \leq h^*(n), \forall n$$

Aunque la mayor parte de las heurísticas se obtienen por **relajación de restricciones**, no es el único método. Otras alternativas son:

- **Heurísticas clásicas**: Utilizar algoritmos más comunes, que se han utilizado para otros problemas.

- **Funciones probabilísticas**: Para establecer los mejores sucesores, basándose en técnicas de muestreo estadístico.

- **Añadiendo restricciones**: Se renuncia a la admisibilidad, pero puede ser una posibilidad para encontrar una solución aunque no se garantice que sea óptima.

- **Elaborar una función de evaluación**: A partir de algunos rasgos o atributos propios del problema se puede establecer una ponderación para obtener una heurística.

2.1.4.1. Distancia Manhattan

La distancia Manhattan es una heurística utilizada en multitud de ocasiones, tanto en estimación de rutas, sobre todo por ciudad, como en juegos de lógica.

Aunque se puede aplicar a cualquier número de dimensiones, normalmente se aplica solamente sobre un plano, dos dimensiones. Se basa en calcular la diferencia entre dos puntos utilizando las coordenadas en valor absoluto, de tal forma que calcular la distancia entre la meta, $q(x_2, y_2)$, y el inicio, $p(x_1, y_1)$, se representa matemáticamente del siguiente modo:

$$d(p, q) = |x_2 - x_1| + |y_2 - y_1|$$

Si dentro de un problema hay varias metas, basta con acumular el resultado de cada una de ellas, a la meta más cercana, con un sumatorio y se obtiene el valor heurístico deseado. No se realiza ninguna asignación de metas, porque una asignación errónea convertiría la heurística en no admisible.

Esta medida resulta interesante aplicarla en juegos de lógica en los que sólo se permiten los movimientos en horizontal y en vertical. Así se consigue una estimación que no supera el coste real en ningún momento y por lo tanto se convierte en una heurística admisible.

2.1.4.2. Asignación de metas

Otro método para guiar la búsqueda es realizar una asignación para cada una de las metas que se quieren alcanzar. Esta asignación tiene que ser en todo momento mínima para que se mantenga admisible, por lo que en muchas ocasiones es necesario volver a realizar estas adjudicaciones constantemente.

Para llevarlo a cabo se representan los puntos iniciales y las metas, como dos conjuntos de vértices, y el objetivo es encontrar la menor unión entre cada par de vértices de diferentes grupos. Dentro del campo de las matemáticas, existe un área encargada de estudiar las relaciones dentro de estos conjuntos, **teoría de grafos** y en especial, este asunto se considera que pertenece a los problemas de **grafos bipartitos**[12].

Esta tarea se puede realizar de diversas formas, desde una elección manual a otros métodos más elaborados como los siguientes:

Programación lineal

Es una técnica de optimización matemática en la que se trata de minimizar o maximizar una función lineal, llamada **función objetivo**, que satisfaga una serie de restricciones que se expresan mediante un sistema lineal de inecuaciones.

Dentro del campo de la búsqueda, es necesario que la función objetivo sea de minimización. Esto se debe a que la solución que se desea encontrar es óptima y por lo tanto esta función no tiene que sobrestimar el coste real de encontrar la solución.

Esta técnica requiere demasiado tiempo de cómputo por lo que no es adecuado utilizarla si se necesita recalcular la asignación de las metas en cada paso.

Cualquier problema modelado con programación lineal se ajusta al siguiente

[12] `https://en.wikipedia.org/wiki/Bipartite_graph` visitado 4 Ago. 2016

esquema:

$$\min z = c^T x_i$$

$$Ax_i \leq b$$

$$x_i \geq 0$$

Donde:

- La función objetivo puede ser de minimización o de maximización.
- La función objetivo queda representada por **z**.
- El vector de coeficientes, **c**, se encuentra traspuesto para poder multiplicar cada valor de este vector por la variable a la que se encuentre asignada, $\boldsymbol{x_i}$.
- Las restricciones se agrupan en la matriz de coeficientes, **A**.
- El vector de términos independientes, **b**, que contiene los valores constantes en función de las restricciones.
- Todas las variables tienen que ser mayor o igual a cero.

Algoritmo de Kuhn

El algoritmo de Kuhn, o método húngaro, es un algoritmo de optimización que resuelve problemas de asignación en tiempo polinómico $O(n^3)$ [6] [7]. Coincide con la mayor capacidad de cómputo que se puede realizar actualmente.

Este algoritmo es capaz de calcular este valor en un tiempo razonable y ofrece una mayor información del dominio que la *distancia manhattan*.

Se modela como una matriz de dimensiones NxM dónde se representa el coste de asignar cada N-ésimo operario con cada M-ésima tarea.

El algoritmo busca la menor asignación posible dentro de la matriz proporcionada. Para ello, esta matriz se comienza a resolver como un sistema de ecuaciones mediante *Gauss-Jordan*[13], consiguiendo situar a cero, por lo menos un

[13] `https://en.wikipedia.org/wiki/Gaussian_elimination` visitado 5 Ago. 2016

elemento de cada fila y cada columna [8]. Estos elementos se corresponden con la asignación realizada.

En caso de que dos asignaciones diferentes produzcan el mismo valor mínimo, se queda únicamente con la primera solución ya que lo que realmente interesa es la estimación realizada por el algoritmo y en ambos casos sería la misma.

Esta heurística se puede utilizar a pesar de que se necesiten recalcular las asignaciones en cada movimiento, siempre que el número de metas no sea muy elevado. Por el contrario, esta tarea conllevaría demasiado tiempo y es posible que se prefiera utilizar una heurística que se pueda calcular más rápidamente, aunque sea menos precisa.

2.1.5. Búsqueda informada

La búsqueda informada o heurística trata de alcanzar las mismas soluciones que los algoritmos de fuerza bruta, pero reduciendo considerablemente el número de nodos expandidos. Para conseguir este objetivo utilizan heurísticas que guían la exploración, de esta forma pretenden seleccionar los estados que lleven a la solución óptima más rápidamente.

Estos algoritmos centran sus esfuerzos en reducir considerablemente la complejidad de los algoritmos de fuerza bruta. Para ello basan su estrategia en dos principales técnicas: reducir el número de nodos expandidos, **tiempo**, y/o reducir el consumo de **memoria**.

Gracias a esto, es capaz de resolver un mayor número de problemas, más complejos, que no se lograban resolver por consumir completamente los recursos disponibles.

Hay que encontrar la heurística adecuada a cada problema, logrando alcanzar el equilibrio entre la precisión de las estimaciones y el tiempo de cálculo. Si por el contrario la heurística está mal informada se puede llegar a convertir en un algoritmo de búsqueda por fuerza bruta.

Hay diferentes algoritmos aplicables a la búsqueda informada, estos métodos estudian cómo encontrar la solución intentando reducir los recursos utilizados,

tiempo o memoria o ambos a la vez. A continuación, se muestran los dos algoritmos más populares dentro de la búsqueda informada.

A*

El algoritmo A*, pertenece al grupo de algoritmos del **mejor primero**. Fue presentado por primera vez en 1968 [9]. Este algoritmo garantiza encontrar la solución óptima entre dos estados si se cumplen unas determinadas condiciones.

Para seleccionar cual es el mejor estado a expandir, se utiliza una función de ordenación para cada uno de los estados, en los que se tiene en cuenta el coste de alcanzar ese estado y una estimación del esfuerzo restante para llegar a la meta.

El algoritmo selecciona en cada iteración el nodo más prometedor de los contenidos en la **lista abierta**[14] para expandirlo. El estado expandido se elimina de la lista abierta y se guarda en otra estructura similar, la **lista cerrada**[15]. A continuación, se comprueba si alguno de los nodos descubiertos es la solución y se finalizaría la búsqueda. En caso contrario los nuevos elementos se insertan ordenadamente en la lista abierta para que puedan ser explorados en sucesivas iteraciones. Este orden viene determinado por la función de ordenación del algoritmo, que representa como:

$$f(n) = g(n) + h(n)$$

Donde:

- g(n) representa el coste real acumulado desde estado inicial a este estado.
- h(n) representa la estimación para alcanzar la solución desde el estado actual.
- f(n) es la combinación de las otras funciones. El algoritmo utiliza este valor para insertar de forma ordenada los estados que se van generando.

El algoritmo selecciona el estado con menor valor de la función de ordenación, **f(n)**, en caso de que varíos estados compartan este valor, se selecciona el que tiene

[14] Estructura que almacena los estados descubiertos, pero todavía por explorar.

[15] Estructura que almacena los estados visitados por el algoritmo.

un valor mayor de **g(n)**, puesto que ya lleva mayor recorrido realizado y el valor heurístico, **h(n)**, es menor. Por lo tanto, requiere explorar menos estados ya que se encuentra más cerca de alcanzar la solución del problema.

Las características de este algoritmo, se pueden resumir en los siguientes apartados:

- **Completo**: El algoritmo A* garantiza que si existe una solución la encuentra.

- **Admisible**: La admisibilidad del algoritmo depende completamente de la heurística utilizada, si la heurística es admisible la solución que encuentra A* es óptima.

- **Eficiencia**: Depende de la heurística que se utilice. No tiene sentido dedicar más tiempo a calcular una buena heurística que a realizar la búsqueda equivalente. Hay que encontrar un equilibrio entre la información proporcionada y el tiempo de cálculo.

- **Problemas**: El consumo de memoria es exponencial, dado que tiene que almacenar todos los estados generados en la lista abierta.

IDA*

El algoritmo IDA*, *Iterative Deepening A**, fue descrito por Richard Korf en 1985. Es similar al algoritmo de **profundidad iterativa**, de fuerza bruta. En este algoritmo se establece un límite de profundidad, a partir del cual no se expande ningún nodo. Para ello se ejecuta el algoritmo de *primero en profundidad* hasta alcanzar este límite y una vez visitados todos los nodos que se encuentran en este rango de profundidad, se incrementa el límite siguiendo cualquier criterio. En el algoritmo IDA* este incremento viene dado por la función heurística empleada en el algoritmo **A*** [10].

$$f(n) = g(n) + h(n)$$

Donde:

- g(n) representa el coste real acumulado desde el estado inicial al estado n.
- h(n) representa la estimación para alcanzar la solución desde el estado n.
- f(n) una estimación del coste para llegar hasta una meta desde el estado inicial pasando por el estado n. El algoritmo utiliza este valor para insertar de forma ordenada los estados que se van generando.

Este algoritmo es capaz de encontrar el camino óptimo entre dos puntos, reduciendo el consumo de memoria exponencial que tenía el algoritmo A*. Para conseguirlo, basa su estrategia en la reexpansión de nodos. Esto puede provocar un alto coste temporal si el dominio de exploración tiene muchas transposiciones, como por ejemplo ocurre buscando el camino óptimo en una rejilla.

Las características de este algoritmo, se pueden resumir en los siguientes apartados:

- **Completo**: El algoritmo IDA* garantiza que si existe una solución la encuentra.
- **Admisible**: El algoritmo IDA* es admisible, si se utiliza una heurística admisible.
- **Eficiencia**: Mientras que su complejidad en tiempo es también exponencial, la complejidad espacial se convierte en lineal en la profundidad del árbol de búsqueda.
- **Problemas**: Es necesario volver a expandir nodos ya visitados porque no se almacenan. Esto provoca un incremento del tiempo de ejecución, en particular si hay muchas transposiciones.

Con el objetivo de analizar los diferentes algoritmos, a continuación, en la tabla 2.2, se muestra una comparativa de sus complejidades, en el peor de los casos, que se han explicado a lo largo de este capítulo.

		Algoritmo			
		Amplitud	Profundidad	A*	IDA*
Complejidad	Temporal	b^d	b^d	b^d	b^d
	Espacial	b^d	d	b^d	d

Tabla 2.2: Comparativa de las complejidades temporal y espacial

Aunque en la tabla aparezca que los algoritmos de fuerza bruta tengan la misma complejidad que la búsqueda informada, la complejidad temporal de los algoritmos informados es considerablemente menor que en la búsqueda no informada. Esto se debe a que, en la tabla, se contempla únicamente la complejidad en el peor caso. Si se utiliza una heurística que no aporta ninguna información estos algoritmos se convierten en una búsqueda a ciegas en el espacio de búsqueda del problema.

2.1.6. Aplicaciones de la IA en juegos

Los juegos siempre han sido utilizados como un área donde realizar las pruebas de diferentes técnicas y algoritmos desarrollados en el campo de la Inteligencia Artificial.

Los resultados obtenidos se pueden comparar empíricamente con el comportamiento humano y con otras técnicas de Inteligencia Artificial. Resulta sencillo implementar estos algoritmos en los juegos debido a que se encuentran delimitados por un conjunto de reglas y un objetivo claro a alcanzar.

Los primeros juegos sobre los que se implementaron técnicas de IA fueron los juegos de lógica, el principal **N-Puzzle** aunque hay muchos otros como el *sudoku* o juegos de tablero como las *damas* o el *ajedrez* [11].

Actualmente, las técnicas de la IA se utilizan para mejorar la jugabilidad de los videojuegos, creando agentes con comportamientos cada vez más similares a los humanos o haciendo que el contenido se genere de forma procedural[16].

[16]El contenido se genera en función de un algoritmo. No esta prediseñado.

Para lograr esta tarea los agentes diseñados suelen usar las siguientes técnicas.

- Hacer trampas, el agente puede conocer la información completa de todos los usuarios, incluidos los enemigos.
- Máquinas de estado. Para controlar el comportamiento del agente en función de las situaciones que ocurran.
- Lógica difusa. La percepción de una situación no es siempre la misma, por lo que se cubren diferentes opciones.

Algunas de estos proyectos han acabado teniendo éxito entre el público como *WATSON*[17] o *Robocup*[18].

2.2. Sokoban

Sokoban es un videojuego de lógica creado por el japonés Hiroyuki Imabayashi en la década de los ochenta. El objetivo del juego es muy simple, se trata de transportar las cajas en el menor número de movimientos hasta ciertas posiciones de un almacén.

Las reglas de este juego son muy sencillas.

- El personaje solo puede empujar las cajas, en ningún momento podrá tirar de ninguna de ellas.
- En cada turno, solamente podrá empujar una caja como máximo o desplazarse de un sitio a otro.

A pesar de tener estas reglas tan simples, es un verdadero reto de lógica. Porque a la posibilidad de mover distintas cajas en cada turno hay que sumarle los movimientos que hacen que el nivel se vuelva irresoluble. Estas posiciones se

[17]Sistema capaz de resolver preguntas a partir de una base de datos.
[18]Competición de fútbol de robots.

pueden encontrar tanto a lo largo del laberinto como la propia interacción con otras cajas, pueden provocar que el nivel no tenga solución a partir de ese punto. Además, el alto número de movimientos necesarios para colocar cada una de las cajas hace que sea un juego bastante complicado.

Para cualquier usuario, la meta principal es encontrar la solución de cada uno de los laberintos. Sin embargo, los jugadores más experimentados intentan superar sus propias puntuaciones intentando encontrar la mejor solución, normalmente tomando como medida los movimientos del jugador. Otra opción es minimizar el número de movimientos de las cajas. En multitud de ocasiones la solución óptima para ambas medidas se basa en el mismo recorrido.

Este famoso juego ha sido desarrollado en muchas plataformas y se han obtenido algunas variantes modificando algunas de las reglas del juego original[19].

- Cambiar el tipo de casillas. En lugar de utilizar los cuadrados, se utilizan hexágonos regulares (*Hexoban*) o triángulos equiláteros (*Trioban*).

- Utilizar varios personajes, que colaboran para alcanzar el mismo objetivo (*Multiban*).

- Modificar el objetivo del juego. A partir de esta premisa se han creado multitud de variantes. Añadiendo colores o números a las cajas y metas como en *Block-o-Mania* o *Sokolor*, de tal forma que cada caja tiene una meta asignada. Otra posibilidad se representa en *Cyberbox* cuyo objetivo es que el personaje avance de una posición a otra, es decir, que sea capaz de escapar del laberinto.

- Añadir nuevos elementos al juego, como teletransportes, agujeros, muros móviles o caminos en las que las cajas se pueden mover en un solo sentido. Algunos juegos que reunen está características son: *Push Crate*, *Sokonex*, *Xsok* y *Cyberbox*.

[19] `https://en.wikipedia.org/wiki/Sokoban#Sokoban_published_by_Thinking_Rabbit` visitado 8 Ago. 2016

- Incorporar la acción que permita tirar de las cajas además de empujarlas, como en *Pukoban*.
- Invertir el objetivo. A partir de una posición final en el juego original, se pide alcanzar una posición determinada. Para ello se tira de las cajas en lugar de empujarlas. Este modo esta incluido en *Sokoban YASC* y *Sokofan*.

2.3. Nomenclatura

Como los usuarios de la comunidad pueden aportar sus propias colecciones de mapas ha sido necesario establecer un estándar en cuanto a la representación. Siguiendo la nomenclatura utilizada en la mayoría de las investigaciones realizadas sobre este problema, se utilizarán los siguientes caracteres para representar las diferentes casillas que nos podemos encontrar en el juego:

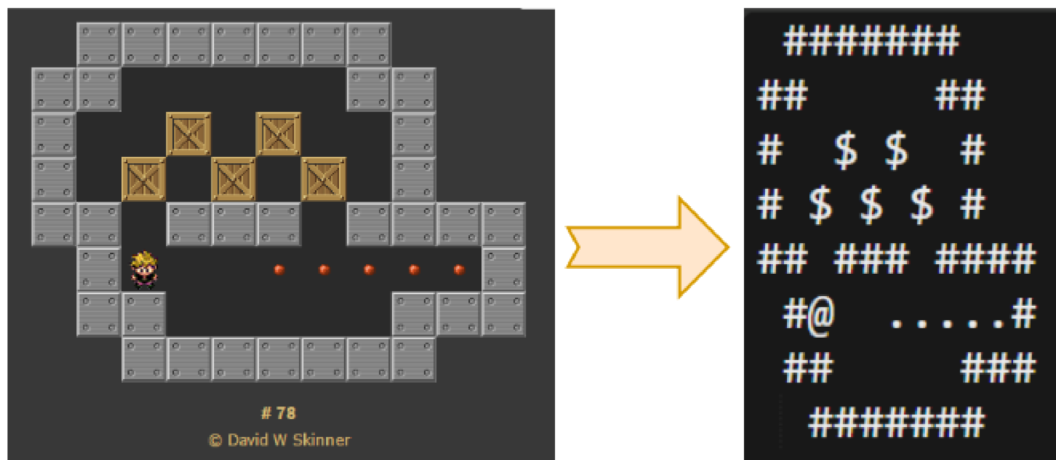


Figura 2.3: Nomenclatura utilizada para representar el nivel 78.

- Con el carácter en blanco, *espacio*, se representan las celdas vacías.
- Con @ se representa al jugador sobre una posición vacía.
- Con el símbolo # se establecen los muros del laberinto. Posiciones inaccesibles tanto para el jugador como las cajas.

- El carácter *$* representa una caja dentro del laberinto.
- Por último, aparece el *.* para simbolizar una meta.

Además de estos símbolos, se pueden encontrar también estos otros, cuando se sobreponen dos símbolos:

- El carácter *+* representa que el jugador se encuentra encima de una meta.
- Si se detecta el carácter *, una caja se encuentra en una meta.

2.4. Sokoban como problema de Búsqueda

Para modelar cualquier dominio como un problema de búsqueda heurística, es necesario definir las siguientes configuraciones:

- **Espacio de estados**: Estructura de datos necesaria para almacenar toda la información relevante de cada situación factible.
- **Conjunto de operadores**: Representa la colección de acciones que están permitidas dentro del problema.
- **Estado inicial**: Situación inicial del problema.
- **Estado meta**: Objetivo que se quiere alcanzar.

Para modelizar esta tarea la mayoría de los algoritmos se centran en los empujones de las cajas en lugar de los movimientos del jugador, con el objetivo de reducir la memoria utilizada. Desde el punto de vista de la búsqueda heurística el problema del Sokoban se puede modelar como:

- **Espacio de estados**: Se almacena la situación actual del mapa, se puede guardar el laberinto o simplemente las posiciones de las cajas y del jugador para ahorrar memoria. Además, se puede almacenar información secundaría como la asignación calculada o datos relativos a la estructura como nodo y nivel de profundidad, para facilitar los cálculos del algoritmo.

- **Conjunto de operadores**: Las acciones posibles en el juego original son cuatro. Simbolizan empujar una caja en cualquier dirección permitida. Teniendo en cuenta que solamente están permitidos los movimientos en vertical y horizontal.
- **Estado inicial**: Representación del nivel proporcionado por la colección de problemas, es decir, el problema a resolver.
- **Estado meta**: Aquel estado en el que todas las cajas se encuentren situadas en las casillas objetivo.

2.5. Estudios previos

Un solucionador de problemas, *solver*, es un programa que trata de alcanzar la solución al dominio en el que se aplica. En el juego del Sokoban los solucionadores que se proponen pueden ser de tres tipos diferentes:

- Encontrar la solución del laberinto, sin importar el número de movimientos realizados.
- Alcanzar la meta en el menor número de movimientos de las cajas.
- Encontrar la solución utilizando el mínimo número de movimientos del jugador.

Los dos últimos tipos de *solver* garantizan que, si encuentran la solución, esta es óptima. Para conseguirlo, es necesario que los algoritmos utilizados para recorrer el espacio de búsqueda empleen una heurística admisible.

Al ser un dominio tan complicado de resolver, no existen muchas investigaciones al respecto. Pocas de ellas llegan a resolver ni siquiera un solo nivel de la colección original.

Por esta razón cabe destacar la tesis doctoral de Andreas Junghanss, que aplicando una gran cantidad de técnicas sofisticadas para el tratamiento de los

datos y utilizando la búsqueda en profundidad iterativa con A*, IDA* [12], consiguió resolver 54 de los 90 problemas que forman la colección original. A este *solver* le puso el nombre de *Rolling Stone*. Inicialmente no resolvía ningún nivel [13] pero a medida que iba incorporando, junto a Jonathan Schaeffer, nuevas herramientas como *macros*[20], *tablas hash*[21] para eliminar ramas repetidas en el espacio de búsqueda [14] y *bases de datos de patrones*[22] consiguió alcanzar esta cifra [15]. Todas estas herramientas le permitían disminuir considerablemente el espacio de búsqueda y alcanzar la solución del problema.

Este impresionante trabajo ha servido de base para cualquier investigación para obtener solucionadores del Sokoban. Tanto es así, que, basándose en la idea de la base de datos de patrones, en 2002 se realiza una investigación [16] que tiene como resultado solamente 10 problemas solucionados. Aunque la mayor parte del proyecto se basa en técnicas de preprocesado de datos que en los propios métodos utilizados en las bases de datos de patrones.

En el 2011, la investigación de Timo Virkkala, concluyo que 24 problemas de la colección original se pueden resolver sin usar el algoritmo de estimación del límite inferior [17] utilizado en el proyecto *Rolling Stone*. De todas formas, acaba diciendo que la búsqueda heuíristica no es la mejor forma de crear un *solver* porque la forma de pensar en cada nivel es diferente, ya sea dividiendo el problema en subproblemas más sencillos o encontrando los túneles y cruces en los que puede quedarse el jugador bloqueado.

Aunque no es una publicación científica, un grupo de autores han fundado una wiki, *Sokoban Wiki*, con mucha información acerca de este juego. Dentro se puede encontrar una gran cantidad de estadísticas de diferentes solucionadores incorporados en la página. La parte negativa es que no se ofrecen mucho detalle sobre cómo conseguir esos resultados, pero son capaces de resolver 71 problemas,

[20] Agrupamiento de varios movimientos ,que se realizan consecutivos, en uno solo.

[21] Estructura de datos para almacenar ordenadamente las claves generadas por un algoritmo. Esto permite revisar eficientemente si hay algún estado repetido.

[22] Almacena la solución de algunos subproblemas, entonces el estado más similar que aparezca en la base de datos se utiliza como valor heurístico.

JSoko y 86 problemas *Takaken* como se pueden ver en las estadísticas de la propia página[23]. En la tabla 2.3 se muestra un resumen de los niveles que lograban resolver, los diferentes *solver* de la página *Sokobano*, de la colección original.

		Solver			
Colección	**Niveles**	**BoxSearch**	**Takaken**	**YASS**	**JSoko**
Original (XSokoban)	90	*42*	*86*	*79*	*71*

Tabla 2.3: Estadísticas de los *solver* de la página Sokobano

[23]http://www.sokobano.de/wiki/index.php?title=Solver_Statistics

Capítulo 3

Objetivos del trabajo

El objetivo de este Trabajo Fin de Grado se basa principalmente en la investigación sobre un método de resolución de partidas de un juego puzzle con un solo jugador, Sokoban.

El primer objetivo será estudiar este juego de lógica, y plantearlo como un problema de Inteligencia Artificial, fundamentalmente basado en búsqueda heurística, aunque se estudiarán otras ramas que ayuden a reducir la complejidad del problema y alcanzar mejores resultados.

Por lo tanto, el objetivo primordial se basará en encontrar una solución óptima a cualquier mapa del juego, para ello se estudiará la complejidad del problema y las decisiones necesarias para poder alcanzar la meta de cada uno de los laberintos de la colección seleccionada.

Para la implementación de todas estas técnicas o diferentes métodos de encontrar la solución a los niveles, así como también las pruebas que sean necesarias realizar, para garantizar el perfecto funcionamiento del proyecto, se utilizará como lenguaje de programación Python, con el fin de conocer un lenguaje nuevo, no utilizado a lo largo del Grado.

Por último, se implementará un agente inteligente[1] que ejecuta las acciones necesarias para llegar a la meta, basándose en toda la investigación realizada durante el proyecto.

[1] https://en.wikipedia.org/wiki/Intelligent_agent

Para dejar claro los objetivos de la investigación se establecen los siguientes puntos:

1. Estudiar el estado de la cuestión en la resolución del juego del Sokoban.
2. Diseñar un programa que juegue a Sokoban, encontrando la solución óptima a los diferentes laberintos de la colección de niveles proporcionados.
3. Se implementarán diferentes algoritmos de búsqueda y heurísticas.
4. Se hará una evaluación empírica de diferentes combinaciones de algoritmos de búsqueda y heurísticas.
5. Se estudiarán los motivos por las que el agente elaborado es capaz, o no, de hallar las soluciones de los problemas.
6. La implementación se realizará en Python, con el objetivo de aprender un nuevo lenguaje de programación.

Capítulo 4

Desarrollo

En este capítulo se describe el sistema desarrollado para solucionar el problema propuesto en el primer capítulo. Para ello se detalla todo lo relevante a lo largo de las diferentes fases de construcción del proyecto, desde el *análisis del sistema* hasta el *diseño* del mismo. Por último, se explican con detalle los *agentes creados* en esta investigación.

4.1. Análisis del sistema

En este punto se ofrece una descripción de los problemas que resuelve el sistema, con qué sistemas interactúa y quiénes son los usuarios que van a usar el sistema. Para ello, se plasma mediante la **especificación de requisitos**, tanto **funcionales** como **no funcionales**, los **actores**, el **entorno** donde se desarrolla y por último la modelización de los **casos de uso** detectados.

4.1.1. Descripción de las características funcionales

El sistema desarrollado en esta investigación debe ser capaz de encontrar, a partir de un mapa, la secuencia de movimientos necesarios para alcanzar la solución. Las características funcionales del sistema representan el comportamiento del sistema. Estas características son las siguientes:

- Tiene que leer el mapa, que representa la situación inicial del problema.

- Debe almacenar los datos en las estructuras de datos necesarias para poder acceder de forma eficiente a la información necesaria.
- Tiene que ser capaz de generar estados, para utilizar la búsqueda heurística.
- El programa tiene que calcular las posiciones en las que, si se coloca una caja, el problema no tiene solución, *deadlock*.
- El programa puede realizar la asignación mínima de objetivos a las metas.
- Se debe de localizar las cajas que el jugador puede mover en cada turno, objetivos accesibles.
- Debe generar los estados sucesores a partir de cualquier estado actual.
- Podrá realizar el movimiento de cajas por el mapa, siempre que se cumplan las reglas del juego.
- El programa permitirá utilizar algoritmos de búsqueda, tanto informada como a ciegas.
- Los algoritmos podrán utilizar una o varias heurísticas como orientación durante la búsqueda.
- El sistema deberá comprobar si un estado es la meta.
- Debe devolver los resultados estadísticos para poder comparar entre diferentes ejecuciones.

4.1.2. Restricciones del sistema

En este apartado se establecen los criterios que el sistema debe verificar mediante los requisitos no funcionales.

- El lenguaje de programación del proyecto será Python, en su versión 3.5.
- El sistema es portable, debido a que solo es necesario instalar un intérprete de Python en el equipo que se desee utilizar.

- Se establece como límite de tiempo para resolver cada nivel, un tiempo máximo de cinco minutos.
- La estabilidad y disponibilidad del sistema depende del equipo en el que se instale el intérprete.

4.1.3. Entorno operacional

En este punto se define el entorno tecnológico necesario para cubrir los requisitos del sistema.

En primer lugar, se establecen las características del portátil utilizado en la tabla 4.1.

Características del equipo	
Procesador	Intel® Core™i3-5010U de 64-bit Dual-core a 2,1 GHz Niveles de caché: - Nivel 1: 128 KB - Nivel 2: 512 KB - Nivel 3: 3072 KB
Memoria RAM	6 GB de memoria principal
Disco duro	500 GB

Tabla 4.1: Características del equipo.

El lenguaje de programación utilizado para llevar a cabo la implementación de este proyecto es **Python**.

El sistema operativo utilizado ha sido una distribución de Linux, concretamente una imagen de Ubuntu 15.10. Dentro de este sistema operativo se ha utilizado la herramienta *open-source Project Jupyter*[1] con el componente*Ipython*[2].

Para elaborar la documentación del proyecto se ha utilizado LaTeX.

[1] `https://es.wikipedia.org/wiki/SciPy`

[2] Consola interactiva, que incluye un intérprete de Python

4.1.4. Especificación de casos de uso

A lo largo de este apartado se exponen los actores que participan en este proyecto y los diagramas y tablas que representan los casos de uso.

4.1.4.1. Descripción de los actores

Los actores son los elementos externos necesarios que interactúan con el sistema. En este proyecto solo es necesario el rol de **usuario**, cualquier persona puede interactuar con el programa y todos ellos tienen las mismas funcionalidades disponibles.

Además, como el código es *open-source*, cualquier usuario puede actuar como **desarrollador** si desea realizar cualquier modificación sobre el código inicial.

4.1.4.2. Descripción de los atributos de los casos de uso

Para la realización de la descripción textual de los distintos casos de uso, se han seleccionado una serie de atributos que describen cada uno de los casos de uso. A continuación se realiza una descripción del significado de cada uno de los atributos utilizados para la descripción de los casos de uso. Se muestra un ejemplo en la tabla 4.2.

- Código: Referencia abreviada y única del caso de uso, se formula mediante CU seguido de un - y de tres dígitos. Por ejemplo, CU-001.
- Nombre: Título del caso de uso.
- Actores: Elemento externo que interactúa con el sistema. El caso de uso representa una funcionalidad demandada por un actor.
- Descripción: Se realiza una descripción básica de la funcionalidad o funcionalidades del caso de uso.
- Precondiciones y poscondiciones: Se realiza una descripción de las condiciones que deben cumplirse para poder realizar una operación, y el estado en el que queda el sistema tras realizar esta operación.

- Escenario: Se realiza una descripción básica de las acciones que se ejecutaran paso a paso en el caso de uso.

Caso de uso	
Código	Valor
Nombre	Valor
Actores	Valor
Precondiciones	Valor
Postcondiciones	Valor
Escenario	Valor

Tabla 4.2: Caso de uso modelo

4.1.4.3. Descripción textual de los casos de uso

En este apartado se exponen los casos de uso analizados a lo largo del proyecto. Para ello se utilizará una tabla para cada uno de ellos, con los campos explicados en el apartado anterior. Pero antes se establece un diagrama con todos los actores y sus interacciones con el sistema. [Figura 4.1]

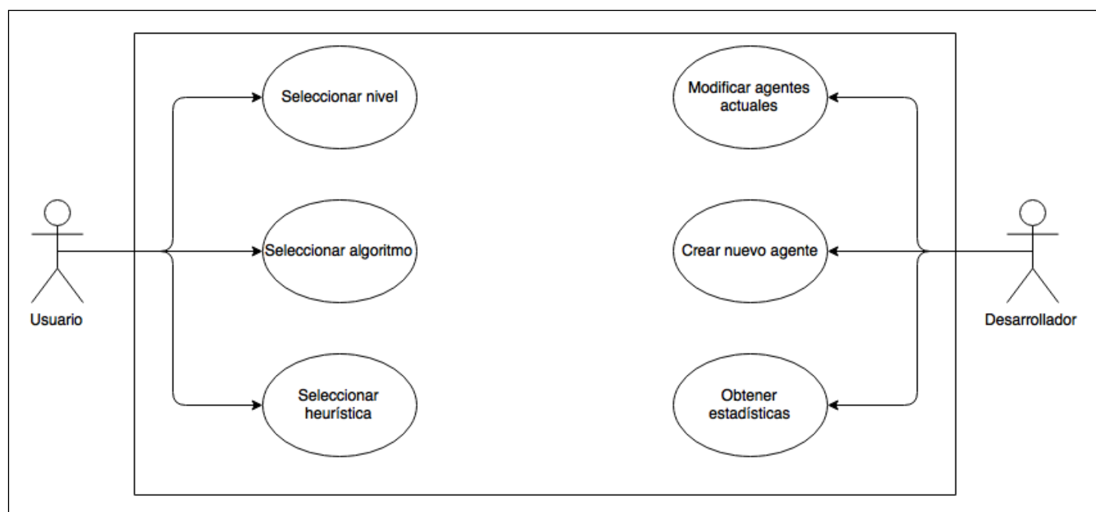


Figura 4.1: Diagrama de casos de uso

Caso de uso	
Código	CU-001
Nombre	Seleccionar nivel
Actores	Usuario
Precondiciones	El intérprete tiene que estar abierto.
Postcondiciones	El programa comienza a ejecutarse.
Escenario	Se selecciona la colección de problemas que quiere utilizar y el nivel concreto.

Tabla 4.3: Caso de uso 001

Caso de uso	
Código	CU-002
Nombre	Seleccionar algoritmo
Actores	Usuario
Precondiciones	Se tiene que modificar antes de ejecutar el programa.
Postcondiciones	El programa utiliza el algoritmo indicado como guía.
Escenario	Modificar el algoritmo que el programa tiene que utilizar.

Tabla 4.4: Caso de uso 002

Caso de uso	
Código	CU-003
Nombre	Seleccionar heurística
Actores	Usuario
Precondiciones	Se tiene que modificar antes de ejecutar el programa.
Postcondiciones	El programa utiliza la heurística elegida.
Escenario	Modificar el fichero con la heurística a utilizar.

Tabla 4.5: Caso de uso 003

Caso de uso	
Código	CU-004
Nombre	Obtener estadísticas
Actores	Desarrollador
Precondiciones	El programa tiene que haber terminado la ejecución.
Postcondiciones	Se almacenan los resultados en un fichero externo.
Escenario	El desarrollador puede consultar las estadísticas tras la ejecución.

Tabla 4.6: Caso de uso 004

Caso de uso	
Código	CU-005
Nombre	Crear nuevo agente
Actores	Desarrollador
Precondiciones	Antes de iniciar la ejecución del agente.
Postcondiciones	El nuevo experimento se podrá utilizar otras veces.
Escenario	El desarrollador crea un nuevo experimento.

Tabla 4.7: Caso de uso 005

Caso de uso	
Código	CU-006
Nombre	Modificar los agentes ya creados
Actores	Desarrollador
Precondiciones	Antes de iniciar la ejecución del agente.
Postcondiciones	Se puede ejecutar el experimento modificado.
Escenario	El desarrollador puede modificar los agentes existentes.

Tabla 4.8: Caso de uso 006

4.1.5. Especificación de requisitos

Para la realización de la descripción textual de los distintos requisitos que han sido identificados, se han seleccionado una serie de atributos que describen cada uno de ellos. A continuación se realiza una descripción del significado de cada uno de los atributos utilizados para su descripción, como en el ejemplo de la tabla 4.9:

- Código: Identificación unívoca abreviada del requisito, se construye mediante el código del requisito seguido de un - y de tres dígitos. Los requisitos serán divididos en funcionales y no funcionales y sus códigos son RF para los requisitos funcionales y RNF para los requisitos no funcionales. Por ejemplo, RF-001.
- Nombre: Título del requisito.
- Descripción: Explicación breve de la funcionalidad o restricción que representa el requisito.
- Fuente: Indica a través de que fuente ha sido identificado el requisito. Normalmente este valor se corresponderá con uno o varios códigos de los casos de uso.
- Necesidad: Determina el grado de importancia del requisito. Los valores que puede tomar este atributo son los siguientes:
  - Esencial: El requisito tiene que ser implementado.
  - Deseable: Es preferible implementar el requisito, pero no es obligatorio.
  - Opcional: El requisito se podrá implementar, pero no es importante ni obligatorio.
- Prioridad: Define la urgencia del requisito, de forma que permita definir el orden en el cual serán incluido en el proceso de diseño y el orden de implementación. Los valores que puede tomar este atributo son los siguientes:

- Alta: El requisito debe ser implementado en las fases iniciales del proyecto.
- Media: Se debe implementar después de los requisitos con prioridad alta.
- Baja: El requisito debe ser implementados en las fases finales del desarrollo. Estos requisitos no influirán en el correcto funcionamiento del sistema.

- Estabilidad: Define la estabilidad del requisito durante la vida útil del software. Un requisito estable tiene menos probabilidad de cambiar a lo largo del proyecto. Los valores que puede tomar este atributo son los siguientes:
  - Estable: El requisito no puede variar durante el ciclo de vida del sistema.
  - Inestable: El requisito puede variar a lo largo del ciclo de vida del sistema.
- Verificabilidad: Define el grado para constatar un requisito, es decir, indica en qué grado es posible comprobar que el requisito se ha incorporado en el sistema desarrollado. Los valores que puede tomar este atributo son los siguientes:
  - Alta: Se puede verificar que el requisito ha sido implementado en el sistema. Este tipo de requisitos se corresponden con las funcionalidades básicas del sistema.
  - Media: Se puede verificar que el requisito ha sido implementado en el sistema. Pero requiere de una comprobación compleja o del código fuente del sistema.
  - Baja: Es difícil verificar si el requisito ha sido implementado en el sistema o en algunos casos no es posible.

Requisito del Sistema			
Código	RF-XXX	**Fuente**	Valor
Nombre	Valor		
Descripción	Valor		
Necesidad	Valor	**Prioridad**	Valor
Estabilidad	Valor	**Verificabilidad**	Valor

Tabla 4.9: Requisito funcional de ejemplo

4.1.5.1. Descripción textual de los requisitos

En este apartado se establecen los requisitos, tanto funcionales como no funcionales, siguiendo la metodología explicada en el apartado anterior.

Requisito del Sistema			
Código	RF-001	**Fuente**	CU-001
Nombre	Leer mapa		
Descripción	El sistema debe ser capaz de recibir un nivel o laberinto.		
Necesidad	Esencial	**Prioridad**	Media
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.10: Requisito funcional 001

Requisito del Sistema			
Código	RF-002	**Fuente**	CU-001
Nombre	Almacenar datos en estructuras		
Descripción	El sistema debe almacenar la información en estructuras de datos para acceder a ellos de forma eficiente.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.11: Requisito funcional 002

Requisito del Sistema			
Código	RF-003	**Fuente**	CU-001
Nombre	Calcular *deadlocks*		
Descripción	El sistema debe realizar un cálculo inicial de las posiciones del laberinto inaccesibles.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.12: Requisito funcional 003

Requisito del Sistema			
Código	RF-004	**Fuente**	CU-002
Nombre	Generar estados		
Descripción	El sistema debe ser capaz de representar las situaciones actuales del nivel en estados.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.13: Requisito funcional 004

Requisito del Sistema			
Código	RF-005	**Fuente**	CU-002
Nombre	Realizar asignaciones		
Descripción	El programa podrá realizar la asignación de cajas a sus metas.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.14: Requisito funcional 005

Requisito del Sistema			
Código	RF-006	**Fuente**	CU-002
Nombre	Localizar posibles movimientos		
Descripción	En cada turno del programa, el jugador deberá localizar las cajas que puede mover en dicho turno.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.15: Requisito funcional 006

Requisito del Sistema			
Código	RF-007	**Fuente**	CU-002
Nombre	Generar estados sucesores		
Descripción	El sistema tiene que ser capaz de generar los sucesores a partir del estado actual.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.16: Requisito funcional 007

Requisito del Sistema			
Código	RF-008	**Fuente**	CU-002
Nombre	Mover cajas		
Descripción	El jugador podrá mover cajas siempre que cumpla las reglas del juego.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.17: Requisito funcional 008

Requisito del Sistema			
Código	RF-009	**Fuente**	CU-002, CU-005, CU-006
Nombre	Usar algoritmos de búsqueda		
Descripción	El sistema permitirá utilizar algoritmos de búsqueda, tanto informada como a ciegas.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.18: Requisito funcional 009

Requisito del Sistema			
Código	RF-010	**Fuente**	CU-003, CU-005, CU-006
Nombre	Usar heurísticas		
Descripción	El sistema permitirá guiar la búsqueda con heurísticas.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.19: Requisito funcional 010

Requisito del Sistema			
Código	RF-011	**Fuente**	CU-002
Nombre	Comprobar meta		
Descripción	El sistema comprobará si el estado actual es el estado final.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.20: Requisito funcional 011

Requisito del Sistema			
Código	RF-012	**Fuente**	CU-004
Nombre	Obtener resultados		
Descripción	El sistema debe devolver varías estadísticas para comparar entre diferentes experimentos. Las estadísticas devueltas serán: número de cajas, número de nodos expandidos, profundidad alcanzada, tiempo y si se soluciona el nivel.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.21: Requisito funcional 012

Requisito del Sistema			
Código	RNF-001	**Fuente**	CU-005, CU-006
Nombre	Lenguaje de programación		
Descripción	El sistema se implementara en Python, en su versión 3.5.		
Necesidad	Deseable	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.22: Requisito no funcional 001

Requisito del Sistema			
Código	RNF-002	**Fuente**	CU-005, CU-006
Nombre	Portabilidad del sistema		
Descripción	El sistema será portable, solo es necesario disponer de un intérprete de Python.		
Necesidad	Deseable	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Media

Tabla 4.23: Requisito no funcional 002

Requisito del Sistema			
Código	RNF-003	**Fuente**	CU-005, CU-006
Nombre	Límite de tiempo		
Descripción	Se establece un límite de tiempo para alcanzar la solución.		
Necesidad	Esencial	**Prioridad**	Alta
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.24: Requisito no funcional 003

Requisito del Sistema			
Código	RNF-004	**Fuente**	CU-005, CU-006
Nombre	Estabilidad		
Descripción	La estabilidad del proyecto depende de los equipos donde se instale el intérprete.		
Necesidad	Deseable	**Prioridad**	Baja
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.25: Requisito no funcional 004

Requisito del Sistema			
Código	RNF-005	**Fuente**	CU-005, CU-006
Nombre	Disponibilidad		
Descripción	La disponibilidad del proyecto depende de los equipos donde se instale el intérprete.		
Necesidad	Deseable	**Prioridad**	Baja
Estabilidad	Estable	**Verificabilidad**	Alta

Tabla 4.26: Requisito no funcional 005

4.2. Diseño del sistema

A lo largo de este apartado se abordan las decisiones de diseño tomadas para la elaboración del sistema. En primer lugar se trata la *arquitectura* elegida y a continuación se explican detalladamente cada uno de sus *componentes* o subsistemas que lo forman.

4.2.1. Arquitectura del sistema

La arquitectura elegida para el sistema se basa en el **Modelo-Vista-Controlador (MVC)**, como se muestra en la figura 4.2. Por lo tanto, el sistema queda dividido en tres componentes claramente definidos.

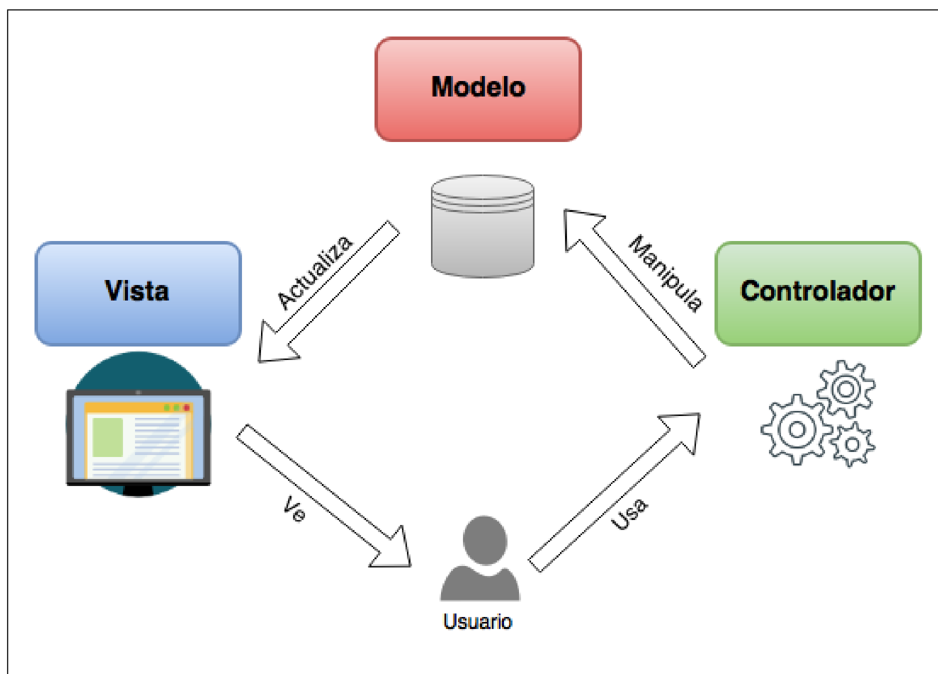


Figura 4.2: Diagrama de Modelo-Vista-Controlador

A continuación, se exponen cada uno de los componentes de esta arquitectura:

- **Modelo**: Es la parte que **almacena la información**, así como las reglas para transformar dicha información. En esta aplicación, está formado por

las estructuras que se crean cuando se proporciona un nivel y los métodos que modifican estas estructuras.

Esta parte es independiente de la vista y del controlador.

- **Vista**: Es la parte que presenta la información del modelo al usuario, la **interfaz**. Puede acceder a los datos del modelo, para consultarlos, pero no puede modificarlos.

  Con esta parte interactúa el usuario, indicando los cambios que hay que realizar en el modelo al controlador. Se notifica cuando el modelo se actualiza, para que los datos que consulta el usuario estén actualizados.

- **Controlador**: Reacciona a las peticiones del usuario y ejecuta las tareas necesarias para modificar el modelo.

  Esta parte es la encargada de relacionar la vista con el modelo.

4.2.2. Descripción general del sistema

Para estudiar mejor el diseño de la aplicación, se realiza un diseño de las diferentes clases necesarias para el desarrollo del proyecto. El diseño definitivo se muestra en la figura 4.3.

A continuación, se explican detalladamente los atributos y los métodos o funciones que forman cada una de ellas.

- **Main**: Es la clase principal del programa. Se encarga de almacenar la lectura del fichero en diferentes estructuras para mejorar el rendimiento. Además, se utiliza como el hilo principal para establecer la conexión entre las diferentes clases.

- **Position**: Esta clase sirve de representación de cada una de las celdas del laberinto, como un par de coordenadas, x e y. Además implementa el método **expand_node_mov()** que se encarga de devolver una lista con las posiciones adyacentes a una caja a las que se puede llegar desde una determinada

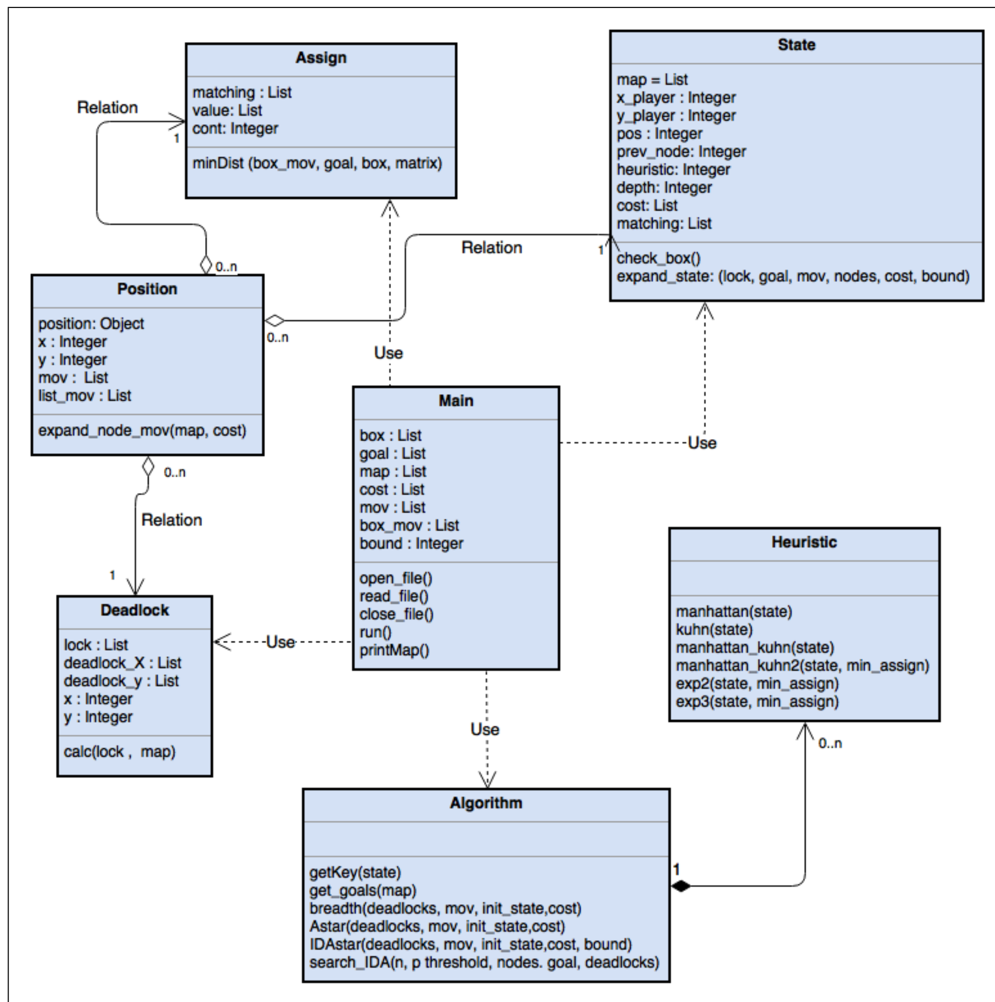


Figura 4.3: Diagrama clases del sistema

posición. Esta tarea es interesante para saber qué cajas puede mover el jugador en cada turno.

- **Deadlock**: Con esta clase representan las posiciones que cuando son ocupadas pueden hacer que el nivel se quede sin solución, esto es, que son *deadlock*. A lo largo de esta clase se desarrolla la función **calc()** que se encarga de encontrar todas estas posiciones dentro del laberinto.

- **Assign**: Se encarga de encontrar la asignación mínima de cada caja a cada meta. La función utilizada para realizar esta asignación es el **método húngaro** o **algoritmo de kuhn**. Este algoritmo es capaz de detectar las

asignaciones que no son factibles, entonces se detecta que el problema no tiene solución.

- **State**: Con esta clase representan cada una de las situaciones que se pueden encontrar durante la búsqueda. A partir de un estado inicial, utilizando la función **expand_state()** implementada en esta clase, se generan los estados sucesores, para acabar encontrando la solución del problema.

- **Algorithm**: Dentro de esta clase, se implementan los diferentes algoritmos de la búsqueda. En este caso sólo se han desarrollado el algoritmo de fuerza bruta, **amplitud**, y utilizando heurísticas: **A*** e **IDA***. Se encarga de invocar, si es necesario, a la clase heurística para obtener un valor estimado sobre la distancia restante a la meta.

- **Heuristic**: En esta clase se implementan las diferentes heurísticas creadas para aportar información sobre el dominio del problema. La creación de nuevas heurísticas genera nuevos experimentos sin necesidad de cambiar el algoritmo de búsqueda.

Para comprender mejor el funcionamiento del programa se ha desarrollado el diagrama de flujo 4.4.

Como la verdadera tarea de cómputo, dentro de un algoritmo de búsqueda, es el propio algoritmo, se realiza el diagrama de flujo 4.5, con el funcionamiento y las tareas que realiza cualquier algoritmo para recorrer el espacio de búsqueda.

Donde se puede observar que, en cada iteración del algoritmo, se selecciona el estado actual, donde la primera iteración se corresponde con el laberinto del fichero. A continuación, se obtiene la posición donde se encuentra el jugador y se calculan las posiciones accesibles que son adyacentes a alguna caja, es decir las posiciones en las que el jugador puede empujar una caja para alcanzar un estado sucesor, eliminando aquellos que nos lleven a un problema sin solución.

A continuación, se comprueba si el estado actual se corresponde con la meta. En caso negativo, se selecciona el siguiente estado actual, eso varía en función

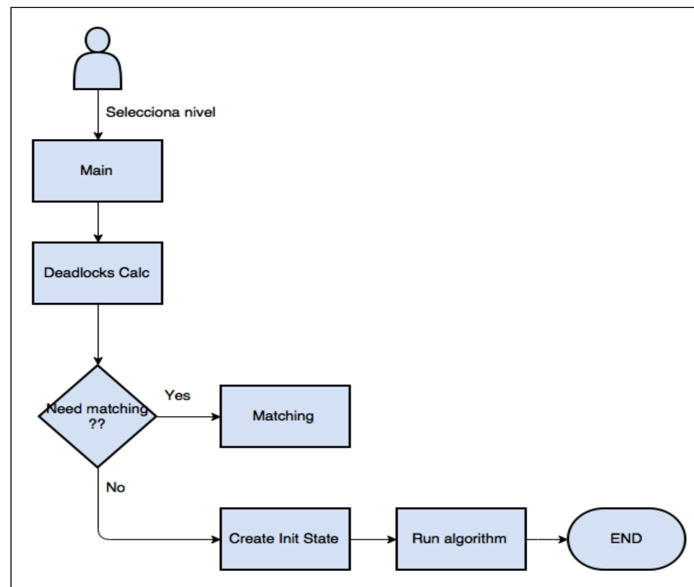


Figura 4.4: Diagrama de flujo del sistema

del algoritmo de búsqueda utilizado y se ejecuta otra iteración del algoritmo. Esta tarea se realiza constantemente hasta encontrar una solución o hasta que se alcanza el límite de tiempo establecido.

4.2.3. Decisiones de diseño

Como se ha observado en publicaciones mencionadas y en lo comentado en anteriores apartados, el principal problema de resolver el juego del Sokoban, mediante búsqueda heurística, es el **consumo exponencial de memoria**. Ante esta situación se elaboran una serie de técnicas, con el principal objetivo de reducir esta complejidad, que se discuten a continuación.

Colección de problemas

La colección de problemas original, dispone de tableros demasiado grandes por lo que se ha decidido optar por una colección con niveles de menores dimensiones. Aunque siguen manteniendo la dificultad que se encuentra al poder quedar bloqueado con algún movimiento erróneo.

La colección elegida es **Microban**, creada por *David W. Skinner*. Esta de-

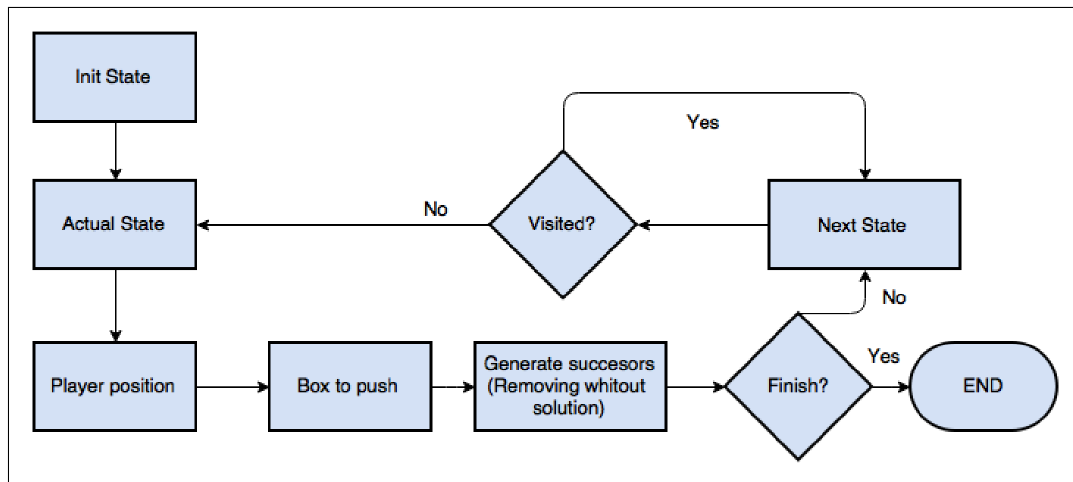


Figura 4.5: Diagrama de flujo del algoritmo

cisión se basa en que, aunque las dimensiones son menores, los niveles mantienen su complejidad y se han utilizado como batería de pruebas en otras investigaciones. La colección consta de 155 problemas lo que cubre un amplio abanico de problemas a resolver. De esta forma se pueden medir la calidad del proyecto y los resultados obtenidos.

Movimientos vs empujones

A partir de cómo se representa un estado, tratando este problema como búsqueda, se pueden encontrar dos posibilidades. Representar cada estado como un movimiento del jugador o por el contrario que cada estado se corresponda con el empujón de una caja.

Si se representa cada estado como un empujón de alguna caja, se reduce considerablemente el número de estados para representar la misma situación, en el otro caso se necesitarían varios estados para alcanzar la misma posición del juego.

Por este motivo la elección tomada, es representar cada estado como el empujón de una caja para conseguir reducir el tamaño del árbol de búsqueda. Las soluciones que se pueden alcanzar siguen siendo óptimas en el número de empujones y, además, en la mayoría de los niveles, esta solución también es óptima en

el número de movimientos, aunque esto no se puede garantizar.

cálculo de deadlocks

Una vez que se juega a Sokoban, nos damos cuenta de que, si colocamos las cajas en algunas posiciones, el problema puede dejar de tener solución. A estas posiciones son conocidas como *deadlocks*. En la imagen 4.6 se muestra como una secuencia de acciones puede dejar un nivel sin solución.

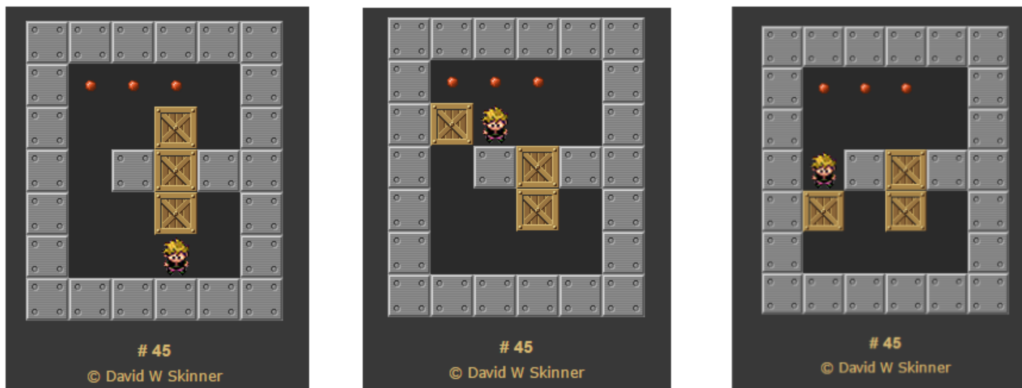


Figura 4.6: Secuencia de acciones que dejan el nivel sin solución.

Estas posiciones se clasifican, en este trabajo, en dos tipos: **fijos** y **dinámicos**.

Los *deadlocks fijos* dependen únicamente del mapa, independientemente de los movimientos que se realicen a lo largo de la partida. Estas posiciones se localizan en las esquinas del laberinto y en las casillas adyacentes a las paredes sin permitir que una caja colocada ahí se separe del muro. Para realizar esta tarea se hace un precálculo y se mantiene a lo largo de toda la ejecución. En la imagen 4.7, se puede observar en rojo los *deadlocks fijos* encontrados.

Por el contrario, los *deadlocks dinámicos* se originan con la interacción entre los diferentes elementos del nivel. Se provocan al mover las cajas y que, al chocar con otras cajas o algún muro, estas no puedan moverse. Esta tarea se necesita calcular en cada estado que se intente expandir, para no crear sucesores sin solución. En la figura 4.8 se muestran algunos ejemplos.

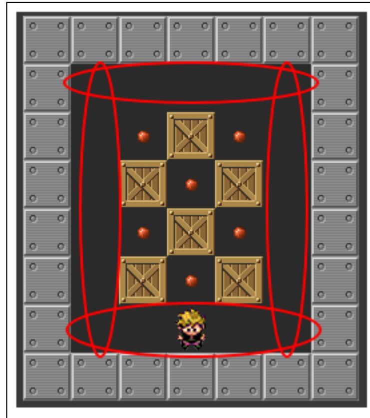


Figura 4.7: Ejemplo de deadlocks fijos

Figura 4.8: Ejemplo de deadlocks dinámicos

Con este sistema de detección no se puede garantizar que se encuentran todos los *deadlocks* del laberinto, pero se puede garantizar que todas las posiciones localizadas lo son.

Realizar asignaciones

Otra forma de detectar las ramas del árbol que no tienen solución es la tarea de asignación de cajas. A cada caja le debe corresponder una meta, si no somos capaces de garantizar que haya una asignación directa para cada objetivo, este camino no tiene solución. Por lo tanto, es inútil seguir profundizando por ese camino a pesar de que la heurística estime que es el mejor camino posible.

El algoritmo para realizar la asignación elegido es el *método húngaro o algoritmo de Kuhn*.

Tratamiento de estados repetidos

Otra forma de reducir el número de estados generados es comprobar si un estado ya ha sido expandido y por lo tanto eliminarlo para no visitar sus sucesores dos veces, puesto que va a generar los mismos sucesores que el estado original.

En multitud de ocasiones, diferentes secuencias de acciones son capaces de generar el mismo estado. Estas posiciones se denominan transposiciones.

Para detectar los estados repetidos, se hacen comparaciones únicamente con los estados en la lista abierta, puesto que compararlos con todos los estados explorados supone un alto coste computacional. Para ello, se ha modificado la función de comparación de los estados. La nueva funcionalidad localiza en primer lugar, si las cajas se encuentran en las mismas posiciones, independientemente del orden. A continuación, se comprueba si ambas posiciones del jugador se encuentran accesibles, es decir, si se puede mover desde una posición a la otra sin necesidad de realizar ningún empujón. En este caso, se detecta un estado idéntico que haría que nuestro sistema pudiera caer en bucles y ralentizaría considerablemente la búsqueda.

Esta mejora es significativa, puesto que reduce considerablemente el número de estados y de esta forma el programa es capaz de resolver un mayor número de problemas.

Heurísticas

Las heurísticas sirven para guiar la búsqueda, por este motivo es interesante disponer de una heurística bien informada. Con el objetivo de explorar un número menor de nodos y encontrar antes la solución.

Por otro lado, es necesario encontrar un equilibrio entre lo que se tarda en calcular este valor y la información que aporta. Si este coste es demasiado elevado, igual resulta más interesante que el algoritmo explore más nodos.

En todo momento el objetivo principal de esta investigación es resolver los niveles de forma óptima, sin embargo, se valorará la posibilidad de utilizar heurísticas no admisibles si así somos capaces de resolver un número mayor de problemas.

Todos estos experimentos realizados se exponen detalladamente en el siguiente apartado.

Orden de selección de los estados

El orden de selección de estados viene dado por el propio algoritmo. En el caso de los algoritmos informados, este criterio se modifica en función de los valores heurísticos que posean cada uno de los estados.

En el algoritmo A* se selecciona el estado con el menor valor de f(n) de la lista abierta. En caso de que haya un empate, se selecciona aquel estado que está a una mayor profundidad, debido a que le quedan menos estados para alcanzar una solución. Si también se encuentran empatados en el valor de profundidad, entonces se puede optar por usar diferentes técnicas como decidir aleatoriamente o usar otra función heurística como desempate.

4.3. Agentes desarrollados

En este apartado se explican los agentes desarrollados para solucionar el problema. Se mencionan todas las ideas de diseño que han ido surgiendo durante la realización del proyecto, independientemente de los resultados obtenidos.

El siguiente apartado se encarga de analizar los resultados obtenidos y se mide la calidad de las soluciones, así como las explicaciones necesarias para esclarecer el éxito o fracaso de cada experimento.

4.3.1. Búsqueda a ciegas

El primer algoritmo diseñado es el algoritmo de **primero en amplitud**. Un algoritmo de fuerza bruta que a pesar de su complejidad es capaz de encontrar la solución óptima del problema si dispone de la memoria suficiente. Puesto que no vale para resolver el problema óptimamente, se ha desestimado la idea de utilizar el algoritmo de *primero en profundidad*.

Para realizar esta tarea, simplemente se ha implementado una cola en la que se van insertando los estados generados al final de la estructura, y se selecciona el primer elemento de la estructura. Este algoritmo sigue la política, **FIFO** (First In First Out), primer elemento en insertarse en la estructura es el primero en eliminarse.

A priori esta ejecución tiene que establecer el menor número de problemas resueltos de toda la investigación. En las siguientes ejecuciones, estos resultados deberán mejorar porque utilizamos la búsqueda para reducir el número de estados expandidos.

4.3.2. Búsqueda heurística

A continuación, se exponen las implementaciones de los diferentes experimentos que utilizan la búsqueda heurística.

Algoritmos

Los algoritmos utilizados dentro de esta sección son: A* e IDA*. El primer objetivo es saber qué algoritmo es capaz de responder mejor ante este dominio. Para ello se realizará una comparativa entre estos algoritmos utilizando la misma función heurística.

Heurísticas

Como sólo es necesario cambiar la heurística para crear un nuevo experimento, se combinan los algoritmos mencionados anteriormente con las siguientes heurísticas.

Distancia Manhattan
Esta estimación se basa en calcular la suma de la distancia mínima entre cada caja y cualquier meta. Sin tener en cuenta que otra caja se encuentre a menor distancia de la misma meta.

$$h(n) = d(A) + d(B) + d(C)$$

$$d(A) = \text{mín}\{1; 0; 1\} = 0$$

$$d(B) = \text{mín}\{2; 1; 0\} = 0$$

$$d(C) = \text{mín}\{3; 2; 1\} = 1$$

$$h(n) = 0 + 0 + 1 = 1$$

Figura 4.9: Ejemplo de cálculo de la distancia *Manhattan*

Este cálculo es bastante rápido porque se dispone de las posiciones de las cajas y las metas. Y sólo es necesario realizar la resta en valor absoluto de las coordenadas. En la imagen 4.9 se muestra un ejemplo de cómo se calcula la distancia *Manhattan*.

Es admisible, porque nunca sobrestima el valor real necesario para alcanzar la meta. No es una heurística muy informada debido a que la estimación que se obtiene, $h(n)$, puede estar muy lejos del valor real, $h^*(n)$.

A pesar de esto, este algoritmo no tiene en cuenta que varias cajas calculen la mínima distancia a una meta errónea o a una meta ya asignada.

Algoritmo de Kuhn

Esta heurística se encarga de realizar una asignación de las cajas sobre las metas. Para que este algoritmo devuelva un valor es necesario encontrar una asignación válida, que a cada caja le asigne una meta. Este algoritmo es capaz de detectar estados que no tienen solución lo que nos ahorra expandir algunos nodos que no conducen al objetivo.

Para realizar esta tarea, se calculan los movimientos de las cajas necesarios para alcanzar cada una de las metas, creando una matriz con estos valores. Una vez

formada la matriz, se utiliza la reducción de Gauss para resolver el sistema de ecuaciones que garantiza una asignación de mínimo coste. En caso de que existan varias asignaciones mínimas, el algoritmo devolverá solamente una, debido a que el valor que retorna la función nunca sobrestima el coste real de alcanzar la solución y por lo tanto se convierte en una heurística admisible.

$$h(n) = d(A) + d(B) + d(C)$$
$$d(A) = \min\{1; 0; 1\} = 1$$
$$d(B) = \min\{2; 1; 0\} = 1$$
$$d(C) = \min\{3; 2; 1\} = 1$$
$$h(n) = 1 + 1 + 1 = 3$$

Figura 4.10: Ejemplo de cálculo del algoritmo de *Kuhn*

Al comparar este ejemplo con el de la distancia *Manhattan*, en la imagen 4.10, se puede observar que el valor devuelto por *Kuhn* es mayor y por lo tanto la estimación es más precisa.

Este algoritmo es mucho más complejo que el anterior. En primer lugar, es necesario realizar un cálculo de cada una de las cajas a cada meta. Una vez que tengamos estos valores el algoritmo se encarga de realizar la asignación mínima. Este algoritmo cuenta con una complejidad O (n^3).

Esta heurística está mucho mejor informada que la de Manhattan. Esto se debe a que se asigna una meta a cada caja, sin embargo, con la distancia Manhattan se pueden asignar varias cajas a la misma meta, lo que resulta imposible. Además, tiene en cuenta los obstáculos que se pueden encontrar en el laberinto.

El verdadero problema que esto conlleva, es el tiempo de ejecución necesario para realizar estos cálculos. Además, en el problema del Sokoban, como no se sabe si la asignación es correcta, es necesario reasignar en cada nuevo estado las metas,

para garantizar la admisibilidad de esta heurística.

4.3.3. Otros experimentos

Con la intención de mejorar los resultados obtenidos se han investigado otras heurísticas que reduzcan el espacio de búsqueda. Estos nuevos experimentos se han implementado, intentando solucionar los problemas encontrados en los anteriores.

A continuación, se describen las heurísticas alternativas ensayadas.

Aplicar la distancia Manhattan a la asignación realizada con Kuhn

Con el objetivo de mejorar el rendimiento, este experimento se basa en realizar una asignación inicial, estableciendo el valor obtenido como límite superior, a partir del cual será necesario realizar una nueva asignación. A continuación, se calcula la distancia Manhattan entre las cajas y las metas de la asignación realizada. Cuando la profundidad de los nuevos estados excede el límite inicial es necesario volver a calcular la asignación.

De esta forma no es necesario realizar la asignación en cada iteración y se consigue un cálculo muy rápido hasta la profundidad límite establecida.

Suponiendo que la asignación inicial es:

$$A \rightarrow Izquierda$$

$$B \rightarrow Centro$$

$$C \rightarrow Derecha$$

$$d(A) = 1 \quad d(B) = 1 \quad d(C) = 1$$

$$h(n) = d(A) + d(B) + d(C) = 1 + 1 + 1 = 3$$

Figura 4.11: Ejemplo de cálculo de *Manhattan* a las asignaciones realizadas con *Kuhn*

El principal problema que encontramos es que la heurística deja de ser ad-

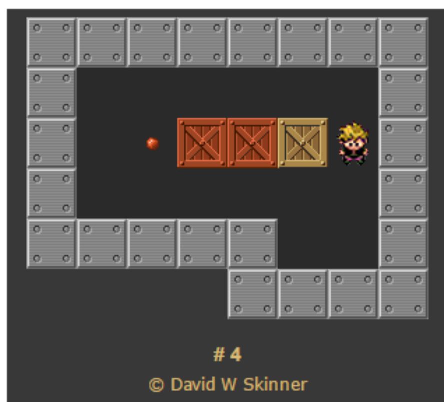
misible, porque si la asignación realizada al inicio del problema no es la correcta la estimación puede sobrestimar el coste de la solución óptima, de esta forma no se puede garantizar la optimalidad de la solución encontrada. Si por el contrario es capaz de solucionar un mayor número de problemas, con soluciones cercanas a las óptimas, puede considerarse un progreso en la investigación.

Combinar varías heurísticas en una

Otra opción para crear nuevas heurísticas ha sido combinarlas. Para ello como en el apartado anterior, se establece un límite inicial calculado con el algoritmo de Kuhn.

Con esta idea, se han creado dos experimentos nuevos. Ambos ejecutan como heurística la distancia Manhattan hasta que los estados alcanzan la profundidad límite.

Una vez alcanzado este valor, el primero de ellos, ejecuta el algoritmo de Kuhn en las sucesivas iteraciones, con lo que se garantiza la optimalidad de la solución. En la figura 4.12 se muestra el cálculo.



Manteniendo la misma asignación
que el ejemplo anterior.
Si $Profundidad \leq Limite$:

$$d(A) = 1 \quad d(B) = 1 \quad d(C) = 1$$
$$h(n) = d(A) + d(B) + d(C) = 1 + 1 + 1 = 3$$

Sino, se reasignan las metas:

$$d(A) = 0 \quad d(B) = 2 \quad d(C) = 1$$
$$h(n) = d(A) + d(B) + d(C) = 0 + 2 + 1 = 3$$

Figura 4.12: Ejemplo de cálculo de *Manhattan* o *Kuhn* en función de la profundidad.

Por otro lado, el otro experimento, en busca de un mayor rendimiento se continúa usando la distancia Manhattan, pero en este caso a las asignaciones realizadas con Kuhn. Como se ha explicado en el apartado anterior, aplicar la

distancia Manhattan a unas asignaciones realizadas, no garantiza que se encuentre una solución óptima, es decir, la heurística pierde la admisibilidad.

Capítulo 5

Resultados

En este capítulo se analizan los resultados obtenidos por cada uno de los agentes descritos en el capítulo anterior. Para medir la calidad del agente creado, se analizan los niveles que es capaz de resolver y los que no, así como una comparativa entre las diferentes ejecuciones.

Cabe destacar que, sin las decisiones de diseño tomadas en el capítulo anterior, para reducir el tamaño de búsqueda del problema, sería imposible resolver ni siquiera un solo nivel, ni de esta ni de ninguna otra colección.

Las tablas con los resultados obtenidos, de los mejores experimentos, se encuentran en el apéndice A de este mismo documento.

5.1. Experimentos base

Debido a la complejidad del problema, es probable que se agoten los recursos computacionales si se tarda mucho en encontrar la solución.

Por este motivo es necesario establecer límites en tiempo o memoria de las ejecuciones del programa. Normalmente, otras investigaciones establecen media hora como tiempo máximo para resolver cada nivel. Para este proyecto se ha decidido utilizar únicamente cinco minutos. En caso de que el número de problemas resueltos sea bajo se ampliaría este límite.

Para establecer una comparativa de los agentes implementados, se toma

como base el algoritmo de fuerza bruta. En este caso, del total de 155 problemas que posee la colección, se resuelven 45 problemas. El número de estados expandidos es muy elevado y la profundidad máxima alcanzada para encontrar la solución es de 27 niveles. Esto hace que para cualquier problema sea prácticamente imposible encontrar una solución mediante fuerza bruta debido a que la mayoría de los niveles requieren muchos más empujones para alcanzar la meta.

Con el objetivo de resolver una mayor cantidad de problemas, se utilizan los algoritmos de búsqueda heurística. Estos algoritmos son A* e IDA*. Para ello, se decide utilizar una heurística poco informada, distancia Manhattan, pero que sea eficiente en obtener estas estimaciones.

El primer experimento realizado con el algoritmo A* utilizando la distancia Manhattan, es capaz de resolver de forma óptima 120 niveles, esto supone un gran avance ya que es capaz de resolver más de la mitad de los problemas de la colección. Además, se reduce alrededor de un 85 % el número de estados expandidos en los niveles que se resolvían por fuerza bruta. También se experimentó con el algoritmo IDA* pero en este caso solo hemos sido capaces de resolver 72 laberintos que en comparación con A* son bastantes menos.

A continuación, se decide realizar otro experimento con el algoritmo de Kuhn como heurística. Esta heurística es mucho más informada, aunque realizar este cálculo es bastante costoso. De esta forma podemos observar cómo se comportan ambas heurísticas.

En este caso ambos algoritmos mejoran los resultados obtenidos en el primer experimento, lo que nos permite decir que una heurística mejor informada mejora considerablemente la búsqueda realizada en este dominio. El algoritmo IDA* mejora en trece niveles y con A* el programa es capaz de resolver ocho niveles más respecto al uso de la distancia de Manhattan, consiguiendo unos resultados de 85 y 128 problemas resueltos óptimamente. A pesar de mejorar más con la búsqueda en profundidad iterativa, el algoritmo A* es capaz de resolver niveles mucho más complejos. A continuación se presenta la tabla 5.1 con el resumen de los resultados obtenidos en estos dos primeros experimentos.

	A*	IDA*
Manhattan	120	72
Kuhn	128	85

Tabla 5.1: Problemas resueltos con búsqueda

Esto se debe a que el algoritmo de profundidad iterativa consigue reducir la complejidad espacial a base de re-expandir nodos cuando llega al límite calculado. Con ello, se utiliza más tiempo para re-expandir nodos ya visitados, lo que hace que este algoritmo alcance en muchos más problemas el límite establecido. En caso de que este límite de tiempo se ampliará, se podría estudiar este algoritmo como hace Andreas Junghanss en su tesis doctoral [12], puesto que, en estas circunstancias, el algoritmo A* lograría agotar la memoria disponible.

Ante estos resultados, la investigación se centra a partir de este momento alrededor del algoritmo A*. Esta decisión se debe a que es capaz de resolver un mayor número de problemas, reduce en un 85 % el número de nodos expandidos respecto a la búsqueda por fuerza bruta y garantiza que la solución encontrada es óptima si la heurística es admisible.

A continuación, en las tablas 5.2 y 5.3 se exponen los niveles que no han sido capaces de resolver el algoritmo A* con las diferentes heurísticas.

A* con Manhattan													
36	78	83	93	98	99	105	106	108	109	111	112	113	114
115	117	118	121	122	123	126	133	134	137	138	139	140	141
143	144	145	146	150	151	153							

Tabla 5.2: Niveles que no son resueltos por A* con distancia Manhattan

A* con Kuhn													
36	93	98	99	105	108	109	111	112	113	114	117	122	123
126	134	137	138	139	140	141	143	144	145	146	150	153	

Tabla 5.3: Niveles que no son resueltos por A* con algoritmo de Kuhn

Analizando otros aspectos estadísticos de las ejecuciones, se puede observar que el número de nodos expandidos disminuye alrededor de un tercio. Por otro lado, el tiempo de ejecución aumenta poco menos de un tercio, lo que justifica este aumento de niveles solucionados. Estas diferencias se muestran en la tabla 5.4.

Rendimiento de Kuhn frente a Manhattan	
Nodos expandidos	Se reduce un 36,7 %
Tiempo	Se incrementa un 32,4 %

Tabla 5.4: Rendimiento de Kuhn frente a Manhattan

En las figuras 5.1 y 5.2 se presentan las gráficas, tanto en nodos expandidos como en tiempo, en las que se comparan ambos experimentos.

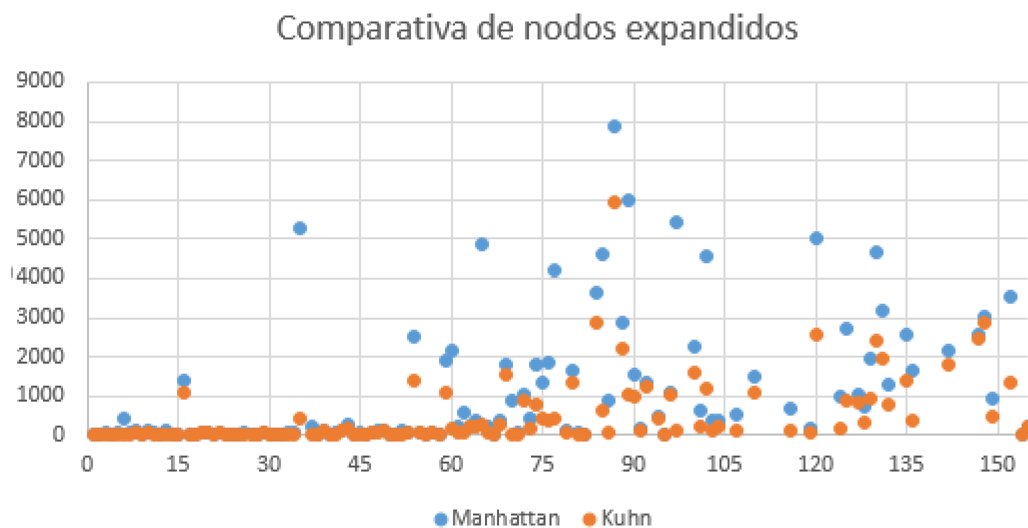


Figura 5.1: Comparativa de nodos expandidos de Manhattan frente a Kuhn

Gracias a los resultados obtenidos, se demuestra que una heurística más informada es capaz de ayudar en la búsqueda, siempre que el tiempo que se tarda en calcular esta estimación no sea superior a la mejora que produce. Por este motivo, se intenta encontrar una heurística que tenga un equilibrio entre la información que proporciona y el coste del cálculo, para lograr que el mayor número de problemas sean resueltos.

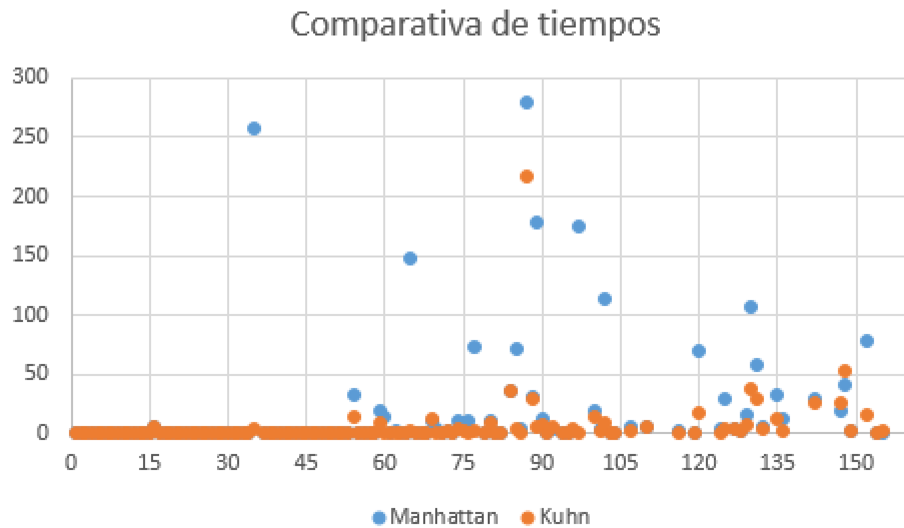


Figura 5.2: Comparativa de tiempos de Manhattan frente a Kuhn

En definitiva, utilizando algoritmos de búsqueda informada, se ha conseguido reducir el número necesario de nodos expandidos para alcanzar la solución óptima entorno a un 85 %. Esto logra aumentar el número de problemas resueltos desde los 44 iniciales hasta la suma de 120, simplemente utilizando el algoritmo de mejor el primero A* con una heurística poco informada como la distancia Manhattan.

Además, utilizando una heurística más informada como el algoritmo de Kuhn, se logra resolver 8 problemas nuevos que han sido imposibles de resolverse antes. De esta forma se puede puede demostrar cómo resolver el problema del Sokoban, mediante el uso de la IA, concretamente de la búsqueda heurística.

5.2. Mejoras de los experimentos

Ante los resultados anteriores, cabe una posibilidad de mejorar estos mismos si se encuentra una heurística que aporte información y no conlleve un alto coste computacional. Con esta finalidad, se han planteado los siguientes experimentos, si bien con ellos se renuncia a la optimalidad de las soluciones encontradas.

5.2.1. Distancia Manhattan a la asignación de Kuhn

Dentro de este experimento se calcula inicialmente una asignación y un límite inicial con el algoritmo de Kuhn. Como este cálculo es bastante costoso, se ha optado por no realizar esta tarea en cada nodo.

En este experimento, como se ha explicado en la sección 4.3.3, página 56, se realiza el cálculo de la distancia Manhattan a las asignaciones calculadas con el algoritmo de Kuhn. Este cálculo tiene el mismo coste computacional que la distancia Manhattan óptima, pero aporta una mayor información ya que a cada caja le corresponde una meta.

Por el contrario, esta heurística pierde la admisibilidad, debido a que, si la asignación no es correcta, la información aportada es errónea y el algoritmo no logra alcanzar la solución óptima. Esto se puede solucionar, basta con reasignar las metas calculadas al inicio para que el valor calculado nunca sobrestime el coste real.

La heurística cambia completamente cuando se llega al límite inicial. La principal intención es que a medida que nos acercamos a la meta podemos necesitar una mayor información para encontrar la meta. Por este motivo a partir de este límite inicial la heurística utilizada es el método húngaro y nos encargamos de recalcular la asignación de los objetivos.

Rendimiento frente a Kuhn	
Problemas resueltos	**124** (4 menos)
Nodos expandidos	Se incrementa un 31,5 %
Tiempo	Se incrementa un 19,8 %

Tabla 5.5: Rendimiento del experimento frente a Kuhn

De esta forma somos capaces de mejorar la velocidad de exploración de nodos, ya que se visita un alrededor de un 30 % pero el tiempo sólo aumenta un 20 %.

Al igual que en el anterior apartado, en las figuras 5.3 y 5.4 se muestran las comparativas de este experimento observando los nodos expandidos y el tiempo utilizado para resolver los niveles, con el mejor experimento hasta el momento, el algoritmo de Kuhn.

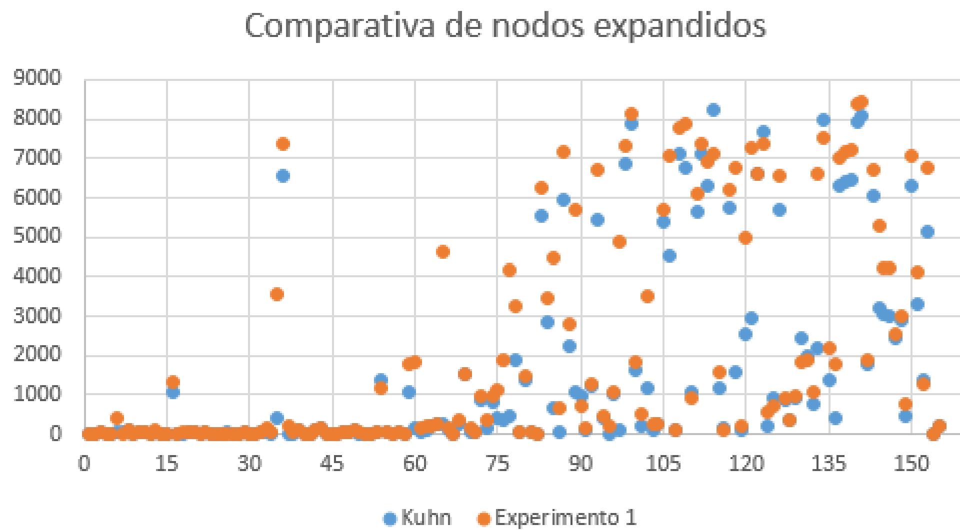


Figura 5.3: Comparativa de nodos expandidos de Manhattan frente a Kuhn

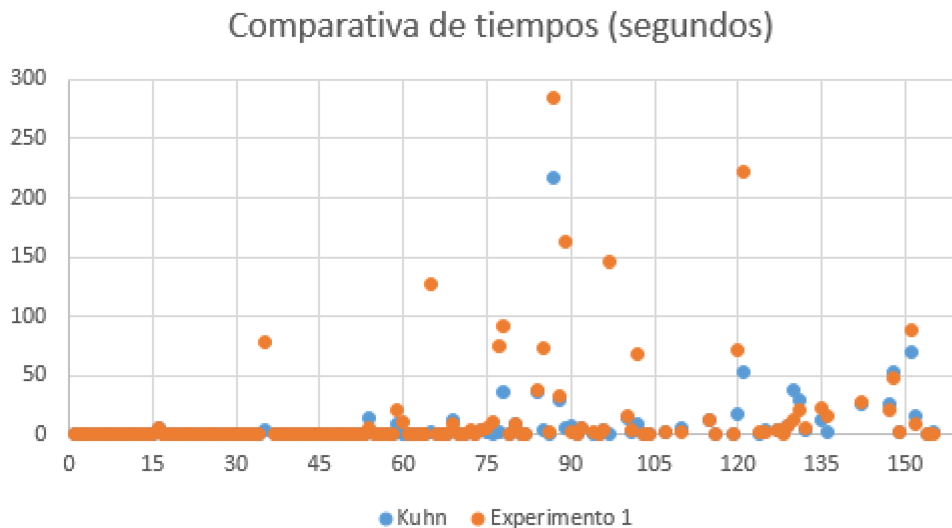


Figura 5.4: Comparativa de tiempos de Manhattan frente a Kuhn

Los resultados obtenidos son peores, debido a que se ha renunciado a la

admisibilidad del algoritmo con la intención de resolver un mayor número de problemas y el resultado ha sido contrario al esperado. Empeora en los principales objetivos que tratamos de resolver, en primer lugar, resuelve cuatro problemas menos, y está peor informada por lo que es necesario aumentar considerablemente los estados visitados para alcanzar una solución.

Por todos estos motivos, este experimento queda desestimado para el progreso del proyecto.

5.2.2. Manhattan con Kuhn

En este experimento, se ha optado por mantener el algoritmo de Kuhn como heurística principal, hasta que se alcanza el limite inicial. A partir de este momento, buscando una mayor velocidad se comienza a utilizar la distancia Manhattan. De esta forma se busca encontrar la reducción de nodos que nos proporciona Kuhn y una vez que se acerque a la meta expandir nodos rápidamente para alcanzar la meta, debido a que se supone que las cajas se encuentran cerca de su correspondiente asignación.

Esta decisión se ha tomado con el objetivo de mejorar el rendimiento sin renunciar a la optimalidad del algoritmo. Esta heurística se mantiene admisible, porque las dos heurísticas que se utilizan lo son y en ningún momento se excede el coste real para alcanzar la solución.

A continuación, se establece un resumen con los resultados obtenidos en este experimento:

Rendimiento frente a Kuhn	
Problemas resueltos	**127** (1 menos)
Nodos expandidos	Se incrementa un 5 %
Tiempo	Se incrementa un 2 %

Tabla 5.6: Rendimiento del experimento frente a Kuhn

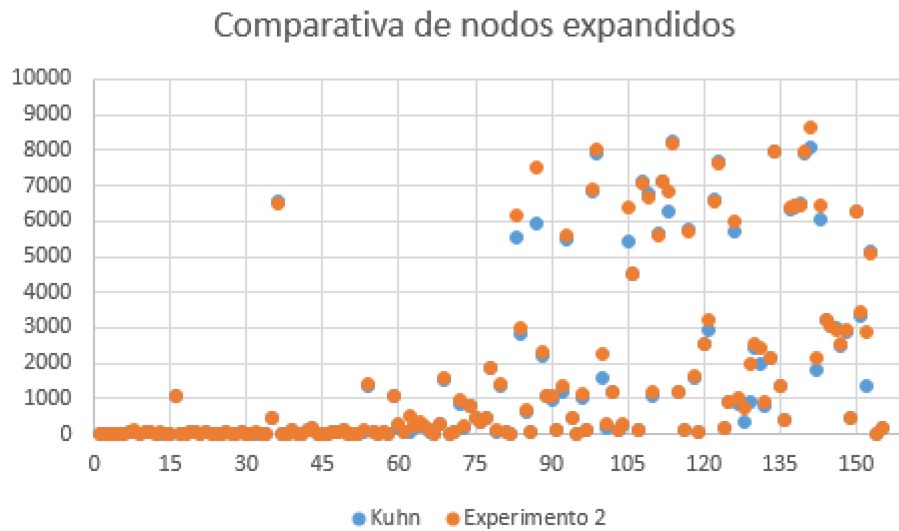


Figura 5.5: Comparativa de nodos expandidos de Manhattan frente a Kuhn

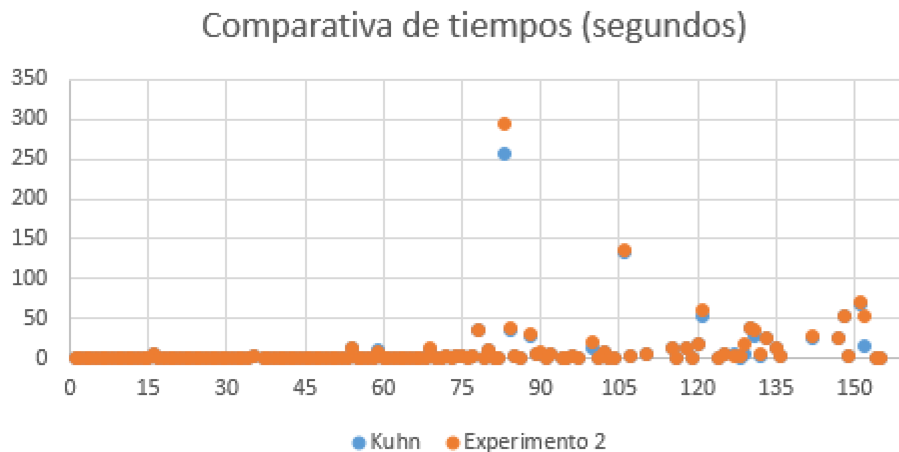


Figura 5.6: Comparativa de tiempos de Manhattan frente a Kuhn

Analizando los resultados obtenidos, en las figuras 5.5 y 5.6, nos damos cuenta de que este experimento es capaz de mejorar la velocidad de los nodos explorados por el algoritmo. El único nivel que no es capaz de resolver comparado con el algoritmo de Kuhn es el 87, este nivel supera el límite establecido de los cinco minutos. Mientras en el experimento base, era capaz de resolverlo en casi cuatro minutos por lo que la diferencia no es tan significativa.

En la mayoría de los niveles se alcanza la solución con el mismo número de

estados explorados, lo que significa que no ha sido necesario utilizar la distancia Manhattan dentro de esta heurística.

Por todos estos motivos este experimento resulta muy interesante y una alternativa al método húngaro a partir del límite, lo que puede ser de gran ayuda para resolver algunos niveles del juego.

5.2.3. Manhattan a las asignaciones y a la meta próxima

Como se ha visto en el primer experimento adicional, utilizar la distancia Manhattan a las asignaciones realizadas por Kuhn puede provocar que la estimación sea incorrecta si esta asignación no se corresponde con la solución. A pesar de renunciar a la admisibilidad, se continúa buscando una heurística sencilla de calcular que reduzca el tiempo en el que el algoritmo consigue alcanzar la solución.

En este experimento, se trata de utilizar la distancia Manhattan a las asignaciones del algoritmo de Kuhn y una vez alcanzado el límite, siguiendo la misma hipótesis que en el experimento anterior, se cambia por la distancia Manhattan óptima. De esta forma, se quiere conseguir una alta velocidad de exploración de nodos y ser capaces de hallar una solución, aunque no sea óptima.

Los resultados obtenidos en este experimento se muestran en la tabla 5.7.

Rendimiento frente a Kuhn	
Problemas resueltos	**123** (5 menos)
Nodos expandidos	Se incrementa un 36,2 %
Tiempo	Se incrementa un 22,2 %

Tabla 5.7: Rendimiento del experimento frente a Kuhn

De esta forma somos capaces de mejorar la velocidad de exploración de nodos, ya que se visita un alrededor de un 35 % pero el tiempo sólo aumenta un 20 %.

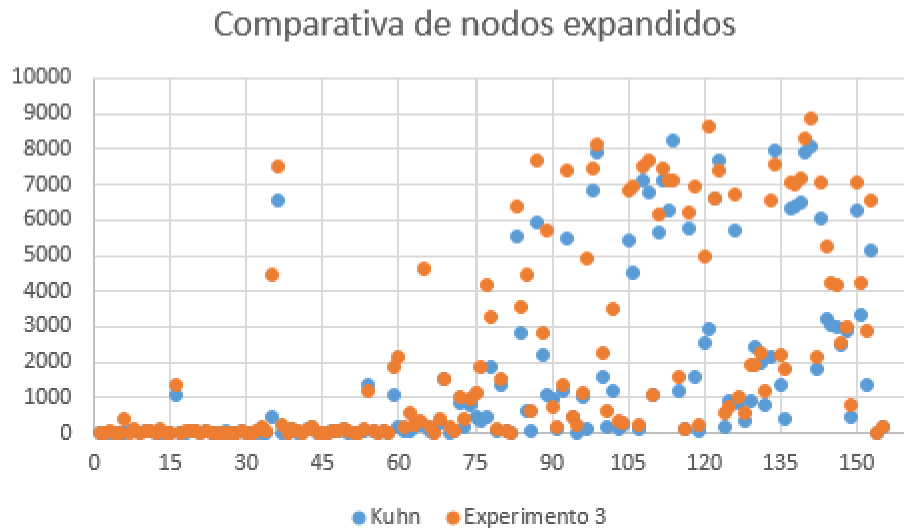


Figura 5.7: Comparativa de nodos expandidos de Manhattan frente a Kuhn

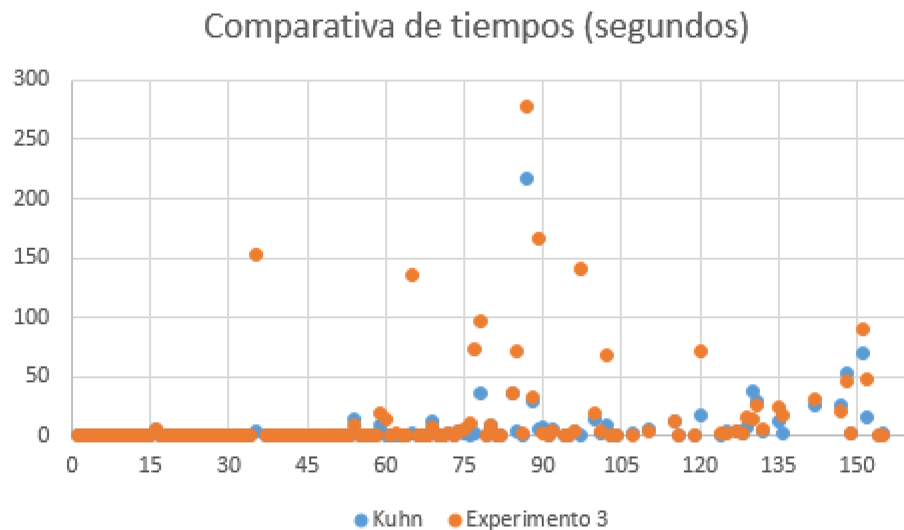


Figura 5.8: Comparativa de tiempos de Manhattan frente a Kuhn

En las figuras 5.7 y 5.8 se comparan los resultados obtenidos frente al mejor experimento, el algoritmo de Kuhn. Estos resultados son peores, como en el primer experimento de mejora, se ha renunciado a la admisibilidad del algoritmo con la intención de resolver un mayor número de problemas. Además, se empeora en los principales objetivos que tratamos de resolver, en primer lugar resuelve un menor número de problemas y se aumenta considerablemente el número de nodos

necesarios para encontrar la solución.

Este experimento se ha desestimado por no mejorar los resultados obtenidos hasta el momento y no encontrar ninguna característica que pueda ayudar al progreso de este trabajo.

5.3. Conclusiones de los resultados

Estudiando cada uno de los experimentos realizados se puede destacar que el problema del Sokoban se puede plantear como una tarea de búsqueda heurística, debido a que con una heurística poco informada se mejora de 44 niveles resueltos a 120.

Se ha demostrado la gran dificultad de encontrar una heurística que obtenga buenos resultados. El principal motivo es que en algunos mapas es necesario alejarse de la meta para alcanzar la solución, esto convierte la búsqueda a bajos niveles de profundidad en una búsqueda por fuerza bruta.

Utilizando una heurística más informada como el algoritmo de Kuhn hemos sido capaces de resolver de forma óptima 8 niveles más. Aunque no parezca un salto tan grande como la mejora que habíamos tenido antes, estos resultados son más interesantes todavía porque el algoritmo es capaz de resolver niveles mucho más complicados que de ninguna otra forma era capaz de hacerlo.

Con el objetivo de resolver un mayor número de problemas se ha intentado buscar el equilibrio de la heurística entre la información aportada y la velocidad de expansión de los estados. Esta tarea resulta complicada porque ni renunciando a la admisibilidad hemos sido capaces de mejorar los resultados obtenidos con el algoritmo de Kuhn. Sin embargo, el segundo experimento obtiene unos resultados similares al algoritmo de Kuhn lo que puede resultar interesante para determinados niveles.

La idea principal que se ha consolidado tras esta investigación es que una heurística más informada es capaz de resolver un mayor número de problemas, como se ha demostrado. Además, el algoritmo elegido para la búsqueda de mejor el

primero (A*) se podría sustituir por el algoritmo de profundidad iterativa (IDA*) en caso de que el consumo de memoria limitará la búsqueda, para ello se podría ampliar el límite de tiempo establecido.

Capítulo 6

Gestión del proyecto

En este capítulo se describe toda la planificación del proyecto para que todo el desarrollo se lleve a cabo con éxito. Para ello, en primer lugar **se enuncian las diferentes fases localizadas**, a continuación se realiza una **planificación** del tiempo necesario para llevar a cabo cada una de las fases junto con el **presupuesto** necesario y por último se establece el marco legal sobre el que se rige esta investigación.

6.1. Descripción de las fases del proyecto

Para lograr desarrollar este proyecto es necesario realizar una buena planificación. Para ello, se divide en las diferentes tareas que se exponen a continuación:

1. **Definición del problema**. Para comenzar, se enuncia el problema y los objetivos que se quieren alcanzar.

2. **Análisis y toma de requisitos**.En esta fase se estudia el problema y se obtiene una especificación detallada del sistema que se va a construir.

3. **Diseño**. Se configura la estructura de la aplicación y cada una de las clases que intervienen en el proyecto.

4. **Implementación del sistema**. Se desarrolla el sistema siguiendo los parámetros establecidos en el análisis y diseño del sistema.

5. **Cálculo de *deadlocks***. Se implementa el cálculo de *deadlocks* como medida para reducir el espacio de búsqueda del problema.

6. **Pruebas del sistema**. Se comprueban las funcionalidades básicas del sistema y se corrigen fallos que se detecten.

7. **Implementación de agentes**. Se desarrollan las diferentes configuraciones establecidas en el diseño, tanto los algoritmos de búsqueda como las heurísticas que emplean.

8. **Pruebas de agentes**. Se comprueba que los algoritmos y heurísticas implementadas funcionan correctamente y se basan en los conocimientos teóricos presentados.

9. **Experimentación y resultados**. Se configuran diferentes experimentos y se miden los resultados obtenidos, con el objetivo de encontrar el mejor resultado posible.

10. **Documentación**. En esta fase se escribe la documentación necesaria para poder llevar a cabo el proyecto.

11. **Presentación**. Durante esta fase se prepara la presentación del proyecto realizado.

6.2. Planificación

La planificación estimada para este proyecto se ha representado con la tabla 6.1, en la que se indica los periodos de tiempo en los que se pretende atender cada una de las fases.

Además, en la figura 6.1, se presenta un diagrama de Gantt para representar gráficamente la planificación estimada del proyecto.

A pesar de que la estimación realizada trata de ser totalmente realista, es imposible predecir algunos de los problemas que pueden surgir durante el desarrollo del proyecto. Por este motivo, una vez completado el proyecto, se calcula

Tarea	Duración	Comienzo	Fin
Definición del problema	1 semana	30/10/2015	06/11/2015
Análisis y toma de requisitos	3 semanas	09/11/2015	26/11/2015
Diseño	2 semanas	27/11/2015	09/12/2015
Implementación del sistema	6 semanas	10/12/2015	21/01/2016
Cálculo de *deadlocks*	1 semana	22/01/2016	27/01/2016
Pruebas del sistema	3 semanas	28/01/2016	14/02/2016
Implementación de agentes	8 semanas	15/02/2016	17/04/2016
Pruebas de agentes	3 semanas	01/03/2016	01/05/2016
Experimentación y resultados	4 semanas	02/05/2016	2/06/2016
Documentación	10 semanas	03/06/2016	12/08/2016
Presentación	2 semanas	13/08/2016	31/08/2016

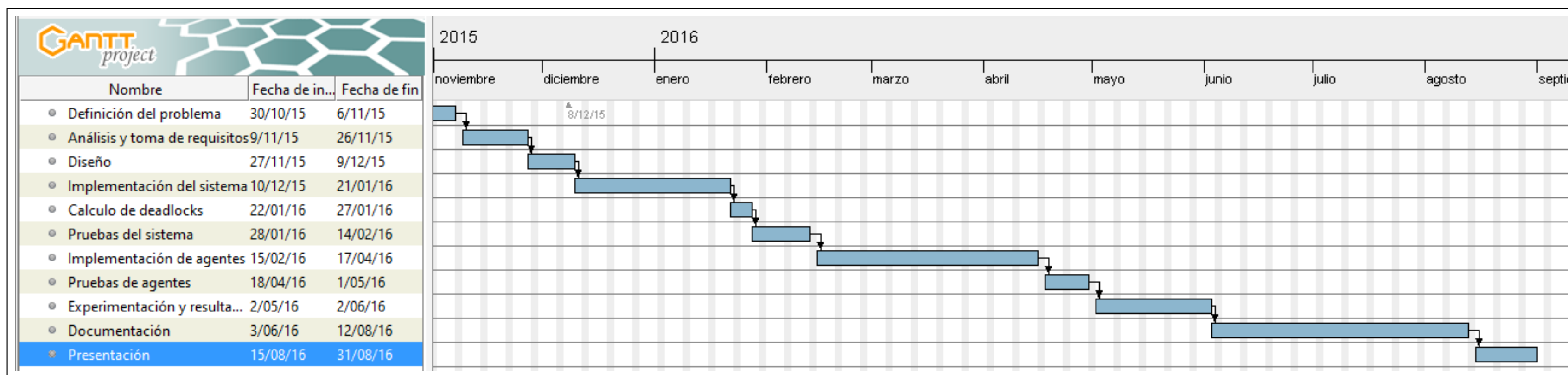
Tabla 6.1: Definición de tiempos y tareas del desarrollo del proyecto.

la dedicación que ha sido necesaria para cada una de las etapas del proyecto. La planificación real se presenta en la tabla 6.2.

Como se ha hecho también en el caso anterior, se genera la representación gráfica 6.2, utilizando un diagrama de Gantt para el coste real del proyecto.

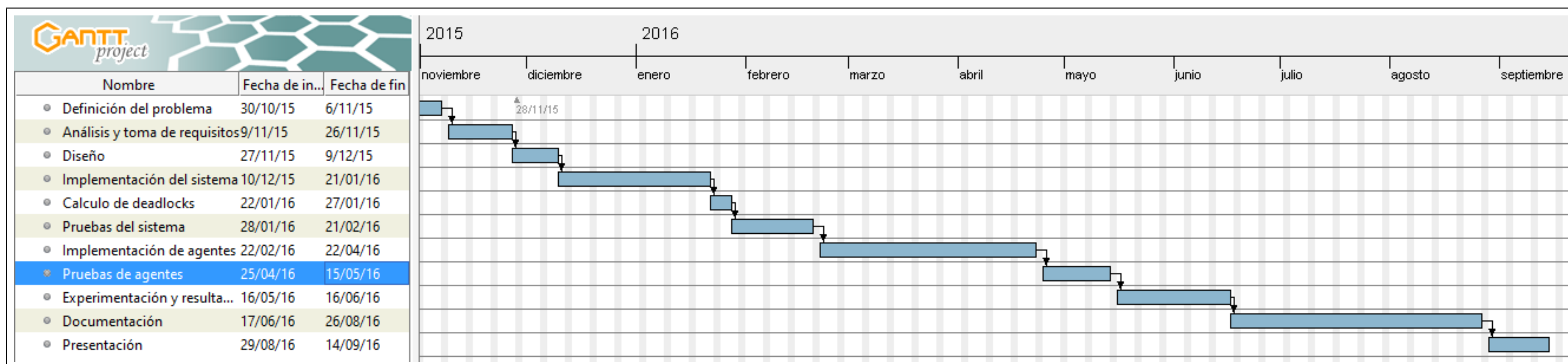
Tarea	Duración	Comienzo	Fin
Definición del problema	1 semana	30/10/2015	06/11/2015
Análisis y toma de requisitos	3 semanas	09/11/2015	26/11/2015
Diseño	2 semanas	27/11/2015	09/12/2015
Implementación del sistema	6 semanas	10/12/2015	21/01/2016
Cálculo de deadlocks	1 semana	22/01/2016	27/01/2016
Pruebas del sistema	4 semanas	28/01/2016	19/02/2016
Implementación de agentes	8 semanas	22/02/2016	22/04/2016
Pruebas de agentes	4 semanas	25/04/2016	13/05/2016
Experimentación y resultados	4 semanas	16/05/2016	16/06/2016
Documentación	10 semanas	17/06/2016	26/08/2016
Presentación	2 semanas	29/08/2016	14/09/2016

Tabla 6.2: Definición real de tiempos y tareas del desarrollo del proyecto.



Nombre	Fecha de in...	Fecha de fin
Definición del problema	30/10/15	6/11/15
Análisis y toma de requisitos	9/11/15	26/11/15
Diseño	27/11/15	9/12/15
Implementación del sistema	10/12/15	21/01/16
Calculo de deadlocks	22/01/16	27/01/16
Pruebas del sistema	28/01/16	14/02/16
Implementación de agentes	15/02/16	17/04/16
Pruebas de agentes	18/04/16	1/05/16
Experimentación y resulta...	2/05/16	2/06/16
Documentación	3/06/16	12/08/16
Presentación	15/08/16	31/08/16

Figura 6.1: Planificación inicial del proyecto



Nombre	Fecha de in...	Fecha de fin
Definición del problema	30/10/15	6/11/15
Análisis y toma de requisitos	9/11/15	26/11/15
Diseño	27/11/15	9/12/15
Implementación del sistema	10/12/15	21/01/16
Calculo de deadlocks	22/01/16	27/01/16
Pruebas del sistema	28/01/16	21/02/16
Implementación de agentes	22/02/16	22/04/16
Pruebas de agentes	25/04/16	15/05/16
Experimentación y resulta...	16/05/16	16/06/16
Documentación	17/06/16	26/08/16
Presentación	29/08/16	14/09/16

Figura 6.2: Planificación real del proyecto

6.3. Presupuesto

Dentro de este apartado se realiza un análisis económico de los recursos necesarios para llevar a cabo este proyecto.

Para comprender mejor el gasto de cada uno de los recursos se detalla en diferentes secciones: **humanos**, ***hardware*** y ***software***.

6.3.1. Costes estimados

A continuación se presenta la previsión de costes del proyecto.

Recursos humanos

Los recursos humanos engloban cada uno de los empleados necesarios para llevar a cabo este proyecto.

A lo largo de este proyecto se ha considerado necesario disponer de un **desarrollador** y un **jefe de proyecto o supervisor**. En la tabla 6.3 se exponen las horas estimadas junto al coste en euros de cada una de ellas para cada uno de los empleados necesarios.

Puesto	Horas estimadas	Coste (€/hora)	Coste total
Desarrollador	1075	17 €	18.275 €
Jefe de proyecto	65	24 €	1.560 €
Total			**19.835 €**

Tabla 6.3: Estimación de costes humanos.

Además del salario que hay que abonar a cada empleado, en la tabla 6.4 se tiene en cuenta la cotización a la Seguridad Social (SS) que se corresponde con un 30 % añadido sobre el sueldo de cada empleado. Este porcentaje se divide en las siguientes áreas: un 23,6 % de Seguridad Social, un 5,5 % en desempleo, un 0,6 % para formación profesional y un 0,2 % para el fondo de garantía salarial Fogasa.

Puesto	Coste total	Coste total con SS
Desarrollador	18.275 €	23.757,5 €
Jefe de proyecto	1.560 €	2.028 €
Total	19.835 €	**25.785,5 €**

Tabla 6.4: Costes humanos añadiendo la Seguridad Social.

Recursos hardware y software

A continuación, en la tabla 6.5, se establecen los recursos hardware necesarios para llevar a cabo el proyecto. Tanto la vida útil como el uso de cada recurso se expresa en meses.

Concepto	Coste	Vida útil	Uso	Coste proyecto
Ordenador	599 €	36	10	166,39 €
Total				**166,39 €**

Tabla 6.5: Presupuesto estimado de recursos hardware.

Los recursos software se desprecian dentro de estos cálculos, puesto que se ha utilizado software libre y gratuito a lo largo de todo el proyecto.

Resumen

En la tabla 6.6 se muestra un resumen con cada uno de los gastos.

Categoría	Coste total
Recursos humanos	25.785,5 €
Recursos Hardware	166,39 €
Recursos Software	0 €
Total	**25.951,89 €**

Tabla 6.6: Resumen del presupuesto estimado.

6.3.2. Costes reales

Como la planificación real difiere ligeramente de la estimada, es necesario realizar un nuevo presupuesto, teniendo en cuenta estos sobrecostes.

Recursos humanos

En cuanto a los recursos humanos utilizados, será necesario disponer del desarrollador dos semanas más. Los nuevos resultados se muestran en las tablas 6.7 y 6.8.

Puesto	Horas estimadas	Coste (€/hora)	Coste total
Desarrollador	1125	17 €	19.125 €
Jefe de proyecto	65	24 €	1.560 €
Total			**20.685 €**

Tabla 6.7: Costes humanos reales.

Puesto	Coste total	Coste total con SS
Desarrollador	19.125 €	24.862,5 €
Jefe de proyecto	1.560 €	2.028 €
Total	20.685 €	**26.890,5 €**

Tabla 6.8: Costes humanos reales añadiendo la Seguridad Social.

Recursos hardware y software

Teniendo en cuenta el mismo criterio anterior, es necesario disponer del equipo utilizado dos semanas más lo que modifica levemente el coste en recursos hardware. Los nuevos costes *hadware* se muestran en la tabla 6.9.

Los recursos *software* se continúan manteniendo despreciables gracias a la utilización de software libre y gratuito.

Concepto	Coste	Vida útil	Uso	Coste proyecto
Ordenador	599 €	36	11	183,03 €
Total				**183,03 €**

Tabla 6.9: Presupuesto real de recursos hardware.

Resumen

Por último, se presenta, en la tabla 6.10, un desglose de cada gasto y el resultado del presupuesto total del proyecto.

Categoría	Coste total
Recursos humanos	26.890,5 €
Recursos Hardware	183,03 €
Recursos Software	0 €
Total	**27.073,53 €**

Tabla 6.10: Resumen del presupuesto real.

6.4. Marco regulador

Los únicos aspectos legales que se deben tener en cuenta dentro de este proyecto son los derechos de autor.

Las colecciones de Sokoban son publicadas con el objetivo de que los diferentes tipos de usuarios puedan incorporar a su juego nuevos niveles y retos. De todas formas, a lo largo de este proyecto se especifica claramente la autoría del conjunto utilizado para la investigación.

Por otro lado, las estadísticas obtenidas de las investigaciones realizadas en la web *Sokobano.de*[1] permiten la divulgación de los resultados siempre que se cumplan las siguientes condiciones:

[1] `http://sokobano.de/wiki/index.php?title=Solvers` visitado 28 Ago. 2016

1. No se pueden vender ni podrán ser utilizados con ninguna actividad comercial.

2. Las divulgaciones tienen que reproducir este contenido en la documentación.

Capítulo 7

Conclusiones

En este capítulo se realiza una descripción de las conclusiones obtenidas tras la realización de este trabajo. En primer lugar, se presentan las conclusiones generales obtenidas tras la realización de este trabajo; a continuación se presentan las conclusiones referentes a los objetivos presentados al inicio de este documento; y por último se presentan los trabajos futuros que podrán realizarse a partir del trabajo presentado en este documento.

7.1. Conclusiones generales

Una vez terminado el proyecto, las conclusiones obtenidas son positivas. Se han logrado cumplir todos los objetivos marcados y se han obtenido unos resultados incluso mejores de los esperados al comienzo de esta investigación.

Este proyecto comenzó con la idea de desarrollar un agente que fuera capaz de encontrar la solución óptima de unos pocos niveles sencillos del juego. Este objetivo se ha superado con creces cuando se han conseguido resolver 128 niveles de los 155 totales de la colección Microban.

Para realizar esta tarea no es suficiente aplicar las mejores técnicas de búsqueda o alcanzar una heurística demasiado informada, es necesario realizar una serie de cálculos que reduzcan el gran espacio de búsqueda resultante. Las técnicas sobre el tratamiento de los datos han conseguido reducir enormemente

el espacio de búsqueda. Las decisiones de diseño que más han ayudado a reducir el tamaño del grafo son: la optimización de los **empujones frente a los movimientos** del jugador, el cálculo de posiciones inaccesibles, ***deadlocks*** y el **tratamiento de los estados repetidos**. Otras técnicas utilizadas han servido para ayudar a lograr mejores resultados, pero han tenido un menor impacto como la decisión de diseño de la estructura de datos utilizada para la lista abierta o qué estado hay que seleccionar en caso de que tengan un mismo valor heurístico.

De esta forma se ha logrado demostrar que el problema del Sokoban se puede abordar desde el punto de vista de la búsqueda heurística. El algoritmo con el que mejores resultados se han obtenido ha sido el algoritmo de mejor el primero, A*. Esto se debe a que el consumo de memoria en estos problemas nunca alcanza el total del que se dispone. En ese caso se debería de utilizar el algoritmo de profundidad iterativa, IDA*, que consigue reducir la complejidad espacial volviendo a visitar estados ya explorados. Por esta misma razón, en el límite temporal de cinco minutos establecido, el algoritmo IDA* es capaz de resolver un menor número de niveles y por lo tanto se selecciona A* para realizar la comparativa de resultados con las diferentes heurísticas con el objetivo de alcanzar los mejores resultados.

También se demuestra la importancia de la información que aportan las heurísticas dentro del campo de la búsqueda. A partir de la búsqueda por fuerza bruta, en la que somos capaces de resolver 45 problemas sin utilizar ninguna guía se consiguen alcanzar 120 niveles resueltos de forma óptima, solamente con la aplicación de una heurística poco informada, la distancia Manhattan. A pesar de la dificultad para encontrar heurísticas informadas dentro de este dominio, se decide utilizar la asignación mínima calculada por el algoritmo de Kuhn en cada uno de los estados, gracias a esto se lograron resolver 8 niveles más y se reduce considerablemente el número de estados necesarios para alcanzar la solución óptima.

Como ejecutar el algoritmo de Kuhn en cada uno de los nodos es demasiado costoso se ha intentado alcanzar un equilibrio en la heurística utilizada. Para ello, se ha intentado encontrar una heurística que aporte información, pero calcularla

sea mucho más sencillo. Se han realizado tres experimentos diferentes intentando combinar varias heurísticas, pero ninguno de estas pruebas mejora los resultados obtenidos. Gracias a esto, se demuestra que en el problema del Sokoban es mucho más importante la información que aporta una heurística que el tiempo que se invierta en calcularla. También se demuestra la dificultad de encontrar una heurística puesto que en muchos niveles hay que hacer movimientos alejándose de las metas.

Las principales limitaciones que se han tenido a lo largo de este proyecto han sido sobre todo por motivos *hardware*. La limitación de memoria que se posee, nos obliga a establecer un tiempo máximo de ejecución y algunos niveles no se resuelven porque necesitaría más tiempo para alcanzar la meta.

Otra limitación, aunque menos importante, que se ha encontrado es el lenguaje de programación. *Python* es un lenguaje interpretado. Esto hace que sea mucho más lento que otros compilados como *C* y al ser una tarea que requiere tantos cálculos, no permite encontrar la solución en algunos laberintos.

7.2. Conclusiones referentes a los objetivos

En este apartado se analizan los objetivos marcados en el capítulo 3 de este documento y se evalúa cada uno de los puntos analizando si se han cumplido.

1. **Estudiar el estado de la cuestión en la resolución del juego del Sokoban**. Este criterio es básico para poder desarrollar el proyecto, ha sido necesario investigar acerca de los diferentes algoritmos y heurísticas con la finalidad de alcanzar el mejor agente posible. Además, es imprescindible consultar investigaciones anteriores para desarrollar nuevas alternativas a los diseños ya existentes.

2. **Diseñar un programa que juegue a Sokoban, encontrando la solución óptima a los diferentes laberintos de la colección de niveles proporcionados**. Este objetivo se cumple, además la cantidad de niveles

resueltos de forma óptima es muy superior al esperado inicialmente, a pesar de establecer un límite de tiempo menor que el utilizado en otras investigaciones.

3. **Se implementarán diferentes algoritmos de búsqueda y heurísticas**. Este criterio también se cumple, además es muy importante para poder realizar una comparativa de los diferentes agentes diseñados.

4. **Se hará una evaluación empírica de diferentes combinaciones de algoritmos de búsqueda y heurísticas**. La evaluación empírica se ha realizado con éxito lo que ayuda a analizar los resultados y obtener unas buenas conclusiones acerca del proyecto.

5. **Se estudiarán los motivos por las que el agente elaborado es capaz, o no, de hallar las soluciones de los problemas**. Se comparan los diferentes agentes y se elige el mejor diseño posible para alcanzar los mejores resultados.

6. **La implementación se realizará en Python, con el objetivo de aprender un nuevo lenguaje de programación.** El proyecto entero, se ha implementado en este lenguaje de programación. A pesar de no tener ninguna noción anterior de este lenguaje, esta decisión ha permitido obtener una nueva perspectiva y abordar el problema con un lenguaje desconocido.

7.3. Trabajos futuros

Una vez finalizado el proyecto se establecen unas futuras líneas de investigación que podrían ayudar a la mejora del programa y a obtener unos mejores resultados.

- **Mejora de los algoritmos**. Mejorar los algoritmos y la representación de los estados para que el consumo de memoria disminuya. De esta forma se pueden generar muchos más estados.

- **Seleccionar algoritmos y heurísticas adecuadas a cada nivel**. Esto permitiría mayor versatilidad y alcanzar las soluciones en función del laberinto, basándose en otros problemas similares.
- **Añadir nuevas técnicas que reduzcan el espacio de búsqueda**. Se ha observado que estas técnicas son imprescindibles para conseguir resolver niveles, por lo que se pueden añadir algunas de las mencionadas en otras investigaciones o cualquier otra que tenga esa misma intención.
- **Sokoban Cooperativo**. Utilizar varios jugadores dentro del mismo laberinto, para ello se debería utilizar algoritmos que tienen en cuenta a varios agentes como A* cooperativo o el algoritmo LRA*.

Apéndice A

Resultados de los experimentos

A lo largo de este apartado se exponen los resultados obtenidos en las tres mejores alternativas de diseño que se han explicado a lo largo de este documento. Además, en primer lugar se establecen los resultados obtenidos por fuerza bruta, con el algoritmo de amplitud, con el objetivo de compararlos y analizar la mejora que se produce.

Una vez establecidos los resultados del experimento de búsqueda a ciegas se colocan los resultados de estas tres mejoras utilizando búsqueda heurística. En primer lugar, se exhiben los resultados de utilizar el algoritmo A* con una heurística muy básica, la distancia Manhattan. A continuación se sitúan los resultados con una heurística mucho más informada como es el algoritmo de Kuhn. Por último se expone el mejor experimento de los que utilizan la combinación de heurísticas en busca del equilibrio y hallar unos mejores resultados.

De todos estos experimentos expuestos, el que mejor resultados ha obtenido ha sido el algoritmo A* con el algoritmo de asignación de Kuhn, como se ha explicado en el capítulo de resultados.

Amplitud					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
1	2	315	8	0.05377483367919922	SI
2	3	10	3	0.0022306442260742188	SI
3	2	1348	13	0.2117900848388672	SI
4	3	5397	7	2.1168782711029053	SI
5	4	16933	6	12.493218421936035	SI
6	3	331308	9999	300.00130796432495	NO
7	6	94785	6	97.9404764175415	SI
8	2	665266	9999	300.0008306503296	NO
9	2	19	10	0.0025730133056640625	SI
10	3	517735	9999	300.000497341156	NO
11	2	353787	16	186.67210984230042	SI
12	2	37	11	0.0065381526947021484	SI
13	3	435368	9999	300.0012264251709	NO
14	2	410	10	0.06400156021118164	SI
15	2	569	12	0.08707761764526367	SI
16	3	337855	9999	300.0006191730499	NO
17	3	606	9	0.13163328170776367	SI
18	2	71804	13	20.322726249694824	SI
19	2	626831	9999	300.00069642066956	NO
20	2	138109	16	41.30935478210449	SI
21	2	27	5	0.004067659378051758	SI
22	2	347732	9999	300.00132393836975	NO
23	2	15942	10	3.4466562271118164	SI
24	2	485	9	0.07977700233459473	SI
25	3	413	7	0.08781552314758301	SI
26	3	3355	10	0.661658763885498	SI
27	2	3880	10	0.6719670295715332	SI
28	2	316	9	0.05852031707763672	SI
29	2	380278	9999	300.001624584198	NO
30	3	161	5	0.03828167915344238	SI
31	3	264	6	0.05762672424316406	SI
32	3	2892	9	0.5399820804595947	SI
33	3	65653	10	20.40826964378357	SI
34	4	57368	8	28.798877477645874	SI
35	5	306181	9999	300.13879466056824	NO
36	5	301459	9999	300.00106406211853	NO
37	3	434969	9999	300.0009093284607	NO
38	3	1131	8	0.28450655937194824	SI
39	2	653604	27	291.59192538261414	SI
40	2	764	7	0.15238189697265625	SI
41	3	129066	13	43.698604345321655	SI
42	3	437872	9999	300.00106406211853	NO
43	3	344630	9999	300.0014503002167	NO
44	1	1	1	0.00012111663818359375	SI
45	3	45458	11	10.17253589630127	SI
46	2	269	8	0.050248146057128906	SI
47	2	475040	9999	300.0001480579376	NO
48	3	328014	9999	300.00076127052307	NO
49	3	403878	9999	300.00094413757324	NO
50	2	390236	9999	300.00026273727417	NO
51	2	213	8	0.0555722713470459	SI
52	4	44658	8	18.82137179374695	SI

Tabla A.1: Resultados con fuerza bruta (Amplitud) 1/3.

Amplitud					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
53	4	89684	12	33.60710310935974	SI
54	4	329765	9999	300.00037384033203	NO
55	2	591979	9999	300.00118613243103	NO
56	2	28	6	0.004773139953613281	SI
57	2	546628	9999	300.0002820491791	NO
58	3	457	11	0.09006595611572266	SI
59	3	280441	9999	300.00017070770264	NO
60	4	505322	9999	300.0003523826599	NO
61	4	429309	9999	300.0017988681793	NO
62	4	339905	9999	300.00132155418396	NO
63	2	748384	9999	300.00013875961304	NO
64	4	309336	9999	300.1448953151703	NO
65	4	284643	9999	300.00056052207947	NO
66	3	243105	9999	300.00164556503296	NO
67	3	583	8	0.1142432689666748	SI
68	3	527569	9999	300.00017952919006	NO
69	3	346666	9999	300.0009255409241	NO
70	4	375603	9999	300.00043869018555	NO
71	2	337773	9999	300.0009620189667	NO
72	3	462174	9999	300.00051760673523	NO
73	3	373086	9999	300.0011386871338	NO
74	4	424305	9999	300.00077199935913	NO
75	4	428809	9999	300.0001344680786	NO
76	3	360851	9999	300.00105023384094	NO
77	4	448022	9999	300.0006411075592	NO
78	5	291465	9999	300.1434414386749	NO
79	3	130769	18	50.0881609916687	SI
80	4	382684	9999	300.0008625984192	NO
81	3	11645	12	2.9203903675079346	SI
82	3	530	14	0.12549328804016113	SI
83	4	357689	9999	300.0001268386841	NO
84	3	352724	9999	300.0008454322815	NO
85	3	423291	9999	300.00124073028564	NO
86	4	386678	9999	300.00117444992065	NO
87	4	345552	9999	300.0001163482666	NO
88	3	323140	9999	300.0008623600006	NO
89	4	405987	9999	300.00008726119995	NO
90	4	259297	9999	300.00095415115356	NO
91	4	214566	14	70.99551510810852	SI
92	3	473565	9999	300.00049328804016	NO
93	8	102266	9999	720.0211138725281	NO
94	3	574506	9999	300.0002806186676	NO
95	8	157505	9999	476.2632944583893	NO
96	3	439233	9999	300.00063848495483	NO
97	5	349699	9999	300.09906816482544	NO
98	5	391791	9999	300.22003412246704	NO
99	4	236306	9999	300.00107073783875	NO
100	4	439437	9999	300.0001595020294	NO
101	4	253762	9999	300.347336769104	NO
102	4	264479	9999	300.0001952648163	NO
103	4	363757	9999	300.001341342926	NO
104	3	426988	9999	300.00132966041565	NO

Tabla A.2: Resultados con fuerza bruta (Amplitud) 2/3

Amplitud					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
105	8	198778	9999	300.0007781982422	NO
106	5	265076	9999	300.00194692611694	NO
107	11	213098	9999	1071.9044325351715	NO
108	4	327538	9999	300.00131583213806	NO
109	5	254924	9999	300.30223321914673	NO
110	4	339065	14	139.75397658348083	SI
111	6	244145	9999	300.1924202442169	NO
112	5	302981	9999	300.00118041038513	NO
113	4	256149	9999	300.00086283683777	NO
114	6	304363	9999	300.45181107521057	NO
115	5	389363	9999	300.0003182888031	NO
116	5	343480	9999	300.00017285346985	NO
117	5	238548	9999	300.0012741088867	NO
118	4	370791	9999	300.00106143951416	NO
119	3	377927	9999	300.00043845176697	NO
120	4	511238	9999	300.0000944137573	NO
121	5	342454	9999	300.00046014785767	NO
122	5	297754	9999	300.0001971721649	NO
123	5	288158	9999	300.0007526874542	NO
124	3	484678	9999	300.00050258636475	NO
125	4	324685	9999	300.0001275539398	NO
126	7	256331	9999	479.53654074668884	NO
127	4	383839	9999	300.00054240226746	NO
128	4	448363	9999	300.00036239624023	NO
129	5	394979	9999	300.00064516067505	NO
130	4	328292	9999	300.00091576576233	NO
131	4	348993	9999	300.0007336139679	NO
132	4	417882	9999	300.00118827819824	NO
133	5	271868	9999	302.5569987297058	NO
134	4	296801	9999	300.0004036426544	NO
135	4	310694	9999	300.00058484077454	NO
136	4	370570	9999	300.00114583969116	NO
137	4	311237	9999	300.00043749809265	NO
138	5	318194	9999	300.2618088722229	NO
139	6	267795	9999	300.000839471817	NO
140	4	272159	9999	300.0008554458618	NO
141	6	597163	9999	300.00089502334595	NO
142	4	256829	9999	300.0011510848999	NO
143	6	250380	9999	461.54025077819824	NO
144	16	66455	9999	876.836786031723	NO
145	12	70235	9999	961.5977082252502	NO
146	12	44410	9999	2885.1914846897125	NO
147	3	299949	9999	300.0003035068512	NO
148	4	282814	9999	300.00026965141296	NO
149	4	426448	9999	300.0002887248993	NO
150	5	258359	9999	300.00057721138	NO
151	4	342172	9999	300.0014908313751	NO
152	4	243447	9999	300.0005958080292	NO
153	10	253049	9999	2322.9824476242065	NO
154	1	5	2	0.02653360366821289	SI
155	11	275169	9999	300.00194120407104	NO

Tabla A.3: Resultados con fuerza bruta (Amplitud) 3/3

A* con distancia Manhattan					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
1	2	17	8	0.0037429332733154297	SI
2	3	4	3	0.001214742660522461	SI
3	2	59	13	0.01509857177734375	SI
4	3	37	7	0.02454543113708496	SI
5	4	47	6	0.0393984317779541	SI
6	3	414	29	0.6187596321105957	SI
7	6	78	6	0.11334395408630371	SI
8	2	106	32	0.035402536392211914	SI
9	2	16	10	0.002450227737426758	SI
10	3	114	21	0.06803512573242188	SI
11	2	70	16	0.040102243423461914	SI
12	2	19	11	0.003865480422973633	SI
13	3	135	21	0.09544062614440918	SI
14	2	15	10	0.003462076187133789	SI
15	2	25	12	0.0053157806396484375	SI
16	3	1381	39	5.850705862045288	SI
17	3	14	9	0.0033807754516601562	SI
18	2	41	13	0.01840353012084961	SI
19	2	56	20	0.020145893096923828	SI
20	2	61	16	0.025992155075073242	SI
21	2	8	5	0.0015442371368408203	SI
22	2	67	15	0.05475640296936035	SI
23	2	21	10	0.007382631301879883	SI
24	2	32	9	0.008333921432495117	SI
25	3	21	7	0.017205238342285156	SI
26	3	55	10	0.02165675163269043	SI
27	2	31	10	0.00908207893371582	SI
28	2	18	9	0.00542449951171875	SI
29	2	77	22	0.057268381118774414	SI
30	3	11	5	0.003705739974975586	SI
31	3	28	6	0.009425163269042969	SI
32	3	39	9	0.015125036239624023	SI
33	3	55	10	0.027065515518188477	SI
34	4	90	8	0.10580182075500488	SI
35	5	5260	31	257.43166518211365	SI
36	5	7430	9999	300.0125961303711	NO
37	3	230	23	0.2519209384918213	SI
38	3	22	8	0.008017301559448242	SI
39	2	103	27	0.042716026306152344	SI
40	2	30	7	0.018319129943847656	SI
41	3	42	13	0.019913673400878906	SI
42	3	134	15	0.10310482978820801	SI
43	3	295	22	0.40004611015319824	SI
44	1	2	1	0.00014257431030273438	SI
45	3	63	11	0.024178266525268555	SI
46	2	18	8	0.004901409149169922	SI
47	2	83	22	0.04175090789794922	SI
48	3	116	14	0.09881281852722168	SI
49	3	137	21	0.10039567947387695	SI
50	2	40	17	0.02016448974609375	SI
51	2	15	8	0.004439353942871094	SI
52	4	97	8	0.07378053665161133	SI

Tabla A.4: Resultados con A* y distancia Manhattan 1/3.

A* con distancia Manhattan					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
53	4	91	12	0.048456668853759766	SI
54	4	2529	30	32.26441693305969	SI
55	2	81	27	0.03390240669250488	SI
56	2	10	6	0.0030405521392822266	SI
57	2	82	23	0.03792595863342285	SI
58	3	26	11	0.009260416030883789	SI
59	3	1904	50	17.93927240371704	SI
60	4	2177	44	13.951658964157104	SI
61	4	216	21	0.2895784378051758	SI
62	4	568	30	1.413830280303955	SI
63	2	203	50	0.09388208389282227	SI
64	4	357	30	0.5394260883331299	SI
65	4	4859	41	148.30123162269592	SI
66	3	203	15	0.2630882263183594	SI
67	3	29	8	0.008682012557983398	SI
68	3	385	28	0.45408010482788086	SI
69	3	1804	37	10.708430528640747	SI
70	4	869	26	2.735652208328247	SI
71	2	92	21	0.08008575439453125	SI
72	3	1047	40	2.376127243041992	SI
73	3	436	25	0.6856245994567871	SI
74	4	1808	34	10.400797605514526	SI
75	4	1326	34	7.996534585952759	SI
76	3	1877	56	10.804222583770752	SI
77	4	4191	55	72.50675415992737	SI
78	5	5894	9999	300.01823925971985	NO
79	3	106	18	0.07086014747619629	SI
80	4	1642	38	10.360068559646606	SI
81	3	78	12	0.038854122161865234	SI
82	3	34	14	0.018828868865966797	SI
83	4	6450	9999	300.01995372772217	NO
84	3	3640	68	35.16344237327576	SI
85	3	4611	51	71.52585887908936	SI
86	4	901	25	3.249978542327881	SI
87	4	7887	53	280.16649293899536	SI
88	3	2859	63	31.360047578811646	SI
89	4	5965	35	178.71731662750244	SI
90	4	1552	16	11.215238571166992	SI
91	4	181	14	0.10861873626708984	SI
92	3	1349	48	3.76092529296875	SI
93	8	6419	9999	300.0034475326538	NO
94	3	462	29	0.5616669654846191	SI
95	8	9	8	0.011618852615356445	SI
96	3	1105	37	2.6259186267852783	SI
97	5	5432	41	174.97335481643677	SI
98	5	7471	9999	300.00857377052307	NO
99	4	8336	9999	300.01892018318176	NO
100	4	2257	52	18.255237579345703	SI
101	4	639	15	3.3414788246154785	SI
102	4	4578	44	114.23878479003906	SI
103	4	349	12	0.4353010654449463	SI
104	3	355	27	0.5594537258148193	SI

Tabla A.5: Resultados con A* y distancia Manhattan 2/3.

A* con distancia Manhattan					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
105	8	6863	9999	300.0304136276245	NO
106	5	7202	9999	300.0251224040985	NO
107	11	543	10	5.657880067825317	SI
108	4	8062	9999	300.0084309577942	NO
109	5	8113	9999	300.00454545021057	NO
110	4	1488	14	5.745381593704224	SI
111	6	6333	9999	300.00552129745483	NO
112	5	7268	9999	300.0116128921509	NO
113	4	6856	9999	300.00996470451355	NO
114	6	7057	9999	300.0297439098358	NO
115	5	7138	9999	300.0215492248535	NO
116	5	684	14	1.8289291858673096	SI
117	5	6790	9999	300.01932740211487	NO
118	4	6961	9999	300.01425194740295	NO
119	3	183	18	0.2808246612548828	SI
120	4	4998	64	69.32617568969727	SI
121	5	8672	9999	300.0003774166107	NO
122	5	6658	9999	300.0246407985687	NO
123	5	7847	9999	300.00881457328796	NO
124	3	972	39	2.827332019805908	SI
125	4	2743	38	28.243235111236572	SI
126	7	6430	9999	300.0158705711365	NO
127	4	1046	32	3.605336904525757	SI
128	4	739	19	1.5312108993530273	SI
129	5	1974	22	15.911618709564209	SI
130	4	4685	36	106.77422261238098	SI
131	4	3173	31	58.53294539451599	SI
132	4	1305	37	5.531883239746094	SI
133	5	7098	9999	300.0138680934906	NO
134	4	7614	9999	300.01479625701904	NO
135	4	2589	36	32.04837942123413	SI
136	4	1671	25	12.45874810218811	SI
137	4	6946	9999	300.00946855545044	NO
138	5	7475	9999	300.0150234699249	NO
139	6	7312	9999	300.0118715763092	NO
140	4	8270	9999	300.0049247741699	NO
141	6	9044	9999	300.03148317337036	NO
142	4	2141	20	29.462268352508545	SI
143	6	6820	9999	300.0252277851105	NO
144	16	4567	9999	300.07477712631226	NO
145	12	5383	9999	300.0116732120514	NO
146	12	4453	9999	300.01803708076477	NO
147	3	2566	50	17.97459888458252	SI
148	4	3025	49	40.179367542266846	SI
149	4	912	35	2.0436618328094482	SI
150	5	7460	9999	300.00524520874023	NO
151	4	7381	9999	300.01626324653625	NO
152	4	3556	35	78.22096538543701	SI
153	10	6624	9999	300.0079038143158	NO
154	1	3	2	0.0035071372985839844	SI
155	11	200	175	0.479766845703125	SI

Tabla A.6: Resultados con A* y distancia Manhattan 3/3.

A* con algoritmo de Kuhn					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
1	2	14	8	0.008434534072875977	SI
2	3	4	3	0.0035333633422851562	SI
3	2	32	13	0.020739316940307617	SI
4	3	10	7	0.02206730842590332	SI
5	4	13	6	0.09144878387451172	SI
6	3	31	29	0.04842495918273926	SI
7	6	39	6	0.47931694984436035	SI
8	2	80	32	0.06229209899902344	SI
9	2	13	10	0.004895925521850586	SI
10	3	58	21	0.07723855972290039	SI
11	2	40	16	0.0702521800994873	SI
12	2	18	11	0.010829925537109375	SI
13	3	31	21	0.07551693916320801	SI
14	2	14	10	0.007433891296386719	SI
15	2	26	12	0.018635034561157227	SI
16	3	1090	39	5.920077562332153	SI
17	3	15	9	0.017709732055664062	SI
18	2	17	13	0.03013467788696289	SI
19	2	56	20	0.07135629653930664	SI
20	2	44	16	0.05076336860656738	SI
21	2	7	5	0.004072904586791992	SI
22	2	54	15	0.08853673934936523	SI
23	2	18	10	0.021969079971313477	SI
24	2	25	9	0.025328397750854492	SI
25	3	11	7	0.033246517181396484	SI
26	3	40	10	0.06690239906311035	SI
27	2	19	10	0.018611669540405273	SI
28	2	18	9	0.022011518478393555	SI
29	2	75	22	0.11890006065368652	SI
30	3	6	5	0.010360240936279297	SI
31	3	19	6	0.04329967498779297	SI
32	3	31	9	0.05431008338928223	SI
33	3	31	10	0.06207084655761719	SI
34	4	17	8	0.07925868034362793	SI
35	5	428	31	3.6546225547790527	SI
36	5	6556	9999	300.0114872455597	NO
37	3	24	23	0.05169248580932617	SI
38	3	19	8	0.022576332092285156	SI
39	2	102	27	0.11072802543640137	SI
40	2	12	7	0.016582489013671875	SI
41	3	19	13	0.035001277923583984	SI
42	3	126	15	0.2283647060394287	SI
43	3	178	22	0.4620034694671631	SI
44	1	2	1	0.00022029876708984375	SI
45	3	21	11	0.018937349319458008	SI
46	2	14	8	0.010896921157836914	SI
47	2	78	22	0.08871150016784668	SI
48	3	68	14	0.13712286949157715	SI
49	3	106	21	0.15823936462402344	SI
50	2	25	17	0.034807682037353516	SI
51	2	9	8	0.007452249526977539	SI
52	4	17	8	0.05500674247741699	SI

Tabla A.7: Resultados con A* y algoritmo de Kuhn 1/3.

A* con algoritmo de Kuhn					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
53	4	49	12	0.08209013938903809	SI
54	4	1384	30	13.498355150222778	SI
55	2	70	27	0.0725715160369873	SI
56	2	8	6	0.005888223648071289	SI
57	2	77	23	0.07581472396850586	SI
58	3	15	11	0.01355886459350586	SI
59	3	1067	50	9.008092641830444	SI
60	4	151	44	0.31271934509277344	SI
61	4	73	21	0.17747044563293457	SI
62	4	83	30	0.16540074348449707	SI
63	2	201	50	0.20806288719177246	SI
64	4	244	30	0.6109335422515869	SI
65	4	249	41	0.9643502235412598	SI
66	3	82	15	0.2899484634399414	SI
67	3	22	8	0.023418188095092773	SI
68	3	279	28	0.5689115524291992	SI
69	3	1527	37	11.325254440307617	SI
70	4	32	26	0.09154009819030762	SI
71	2	36	21	0.046883344650268555	SI
72	3	864	40	2.6068384647369385	SI
73	3	179	25	0.492978572845459	SI
74	4	796	34	3.672002077102661	SI
75	4	431	34	1.6420257091522217	SI
76	3	356	56	0.8263866901397705	SI
77	4	437	55	1.521777868270874	SI
78	5	1862	33	36.01975917816162	SI
79	3	58	18	0.07270097732543945	SI
80	4	1361	38	9.022711753845215	SI
81	3	38	12	0.05713534355163574	SI
82	3	24	14	0.023046255111694336	SI
83	4	5536	47	256.46968388557434	SI
84	3	2852	68	34.84189558029175	SI
85	3	652	51	2.8098080158233643	SI
86	4	61	25	0.15108299255371094	SI
87	4	5957	53	216.97016835212708	SI
88	3	2229	63	28.115923166275024	SI
89	4	1057	35	4.8698647022247314	SI
90	4	979	16	6.36525297164917	SI
91	4	115	14	0.15804100036621094	SI
92	3	1216	48	4.614457845687866	SI
93	8	5462	9999	300.0157792568207	NO
94	3	433	29	0.9082484245300293	SI
95	8	9	8	0.17404937744140625	SI
96	3	1041	37	3.622424840927124	SI
97	5	98	41	0.406757116317749	SI
98	5	6859	9999	300.0094611644745	NO
99	4	7902	9999	300.00360584259033	NO
100	4	1609	52	13.21697211265564	SI
101	4	201	15	1.0936384201049805	SI
102	4	1183	44	8.66225266456604	SI
103	4	119	12	0.3514120578765869	SI
104	3	233	27	0.6293125152587891	SI

Tabla A.8: Resultados con A* y algoritmo de Kuhn 2/3.

A* con algoritmo de Kuhn					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
105	8	5404	9999	300.02132511138916	NO
106	5	4509	50	132.3384280204773	SI
107	11	96	10	2.2174220085144043	SI
108	4	7128	9999	300.02999353408813	NO
109	5	6770	9999	300.0099666118622	NO
110	4	1068	14	4.523442268371582	SI
111	6	5641	9999	300.0066101551056	NO
112	5	7100	9999	300.03127884864807	NO
113	4	6295	9999	300.0084857940674	NO
114	6	8223	9999	300.0063781738281	NO
115	5	1185	29	11.57192349433899	SI
116	5	132	14	0.5460281372070312	SI
117	5	5751	9999	300.02835154533386	NO
118	4	1592	44	12.177318096160889	SI
119	3	82	18	0.2036113739013672	SI
120	4	2553	64	17.397937059402466	SI
121	5	2956	47	53.0362184047699	SI
122	5	6592	9999	300.02578949928284	NO
123	5	7696	9999	300.0099802017212	NO
124	3	191	39	0.4607105255126953	SI
125	4	900	38	4.194238901138306	SI
126	7	5706	9999	300.04639768600464	NO
127	4	858	32	4.042237758636475	SI
128	4	337	19	1.111509084701538	SI
129	5	910	22	6.355167627334595	SI
130	4	2429	36	36.681166887283325	SI
131	4	1973	31	29.027302980422974	SI
132	4	770	37	3.4955198764801025	SI
133	5	2166	39	25.297510862350464	SI
134	4	7981	9999	300.01867294311523	NO
135	4	1383	36	12.762071132659912	SI
136	4	393	25	1.6724627017974854	SI
137	4	6319	9999	300.0122241973877	NO
138	5	6386	9999	300.0043590068817	NO
139	6	6474	9999	300.0218415260315	NO
140	4	7911	9999	300.003009557724	NO
141	6	8069	9999	300.0169630050659	NO
142	4	1801	20	24.74454641342163	SI
143	6	6051	9999	300.0010416507721	NO
144	16	3222	9999	300.03745102882385	NO
145	12	3062	9999	300.0283913612366	NO
146	12	2978	9999	300.00816011428833	NO
147	3	2465	50	24.97734308242798	SI
148	4	2891	49	52.250070571899414	SI
149	4	463	35	1.4826714992523193	SI
150	5	6288	9999	300.0099952220917	NO
151	4	3320	50	69.28039193153381	SI
152	4	1368	35	14.523436546325684	SI
153	10	5115	9999	300.02870321273804	NO
154	1	3	2	0.004143714904785156	SI
155	11	197	175	1.044851541519165	SI

Tabla A.9: Resultados con A* y algoritmo de Kuhn 3/3.

Experimento 2					
Nivel	Número de cajas	Nodos expandidos	Profundidad alcanzada	Tiempo	¿Solucionado?
1	2	14	8	0.0067179203033447266	SI
2	3	4	3	0.002875804901123047	SI
3	2	32	13	0.018507003784179688	SI
4	3	14	7	0.016468048095703125	SI
5	4	15	6	0.056761980056762695	SI
6	3	31	29	0.04712390899658203	SI
7	6	39	6	0.43332910537719727	SI
8	2	106	32	0.03965616226196289	SI
9	2	14	10	0.0038423538208007812	SI
10	3	66	21	0.07388567924499512	SI
11	2	40	16	0.047638654708862305	SI
12	2	18	11	0.01010751724243164	SI
13	3	33	21	0.06689333915710449	SI
14	2	15	10	0.0040318965911865234	SI
15	2	25	12	0.005904197692871094	SI
16	3	1095	39	5.937855958938599	SI
17	3	15	9	0.008010149002075195	SI
18	2	18	13	0.0247952938079834	SI
19	2	56	20	0.03988766670227051	SI
20	2	61	16	0.042504072189331055	SI
21	2	7	5	0.003027677536010742	SI
22	2	54	15	0.08166646957397461	SI
23	2	18	10	0.015512943267822266	SI
24	2	25	9	0.026601791381835938	SI
25	3	11	7	0.027530193328857422	SI
26	3	43	10	0.03726077079772949	SI
27	2	19	10	0.012229204177856445	SI
28	2	18	9	0.006910562515258789	SI
29	2	77	22	0.09432578086853027	SI
30	3	6	5	0.008201360702514648	SI
31	3	23	6	0.019199848175048828	SI
32	3	39	11	0.02898693084716797	SI
33	3	31	10	0.06268668174743652	SI
34	4	30	8	0.06448054313659668	SI
35	5	433	31	3.6654341220855713	SI
36	5	6508	9999	300.0148365497589	NO
37	3	24	23	0.03758549690246582	SI
38	3	19	8	0.03164172172546387	SI
39	2	103	27	0.06258130073547363	SI
40	2	12	7	0.010680437088012695	SI
41	3	19	13	0.03630828857421875	SI
42	3	133	15	0.13683748245239258	SI
43	3	184	22	0.43215441703796387	SI
44	1	2	1	0.0002269744873046875	SI
45	3	21	11	0.025922775268554688	SI
46	2	18	8	0.009781837463378906	SI
47	2	79	22	0.06691193580627441	SI
48	3	78	14	0.12989449501037598	SI
49	3	107	21	0.16788005828857422	SI
50	2	27	17	0.03917670249938965	SI
51	2	12	8	0.004825592041015625	SI
52	4	23	8	0.056558847427368164	SI

Tabla A.10: Resultados con A* con el segundo experimento 1/3.

Experimento 2					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
53	4	91	14	0.0998849868774414	SI
54	4	1392	30	13.570412874221802	SI
55	2	81	27	0.062125444412231445	SI
56	2	10	6	0.003249645233154297	SI
57	2	82	23	0.03748130798339844	SI
58	3	21	11	0.010078191757202148	SI
59	3	1091	50	8.92931342124939	SI
60	4	300	44	0.4704577922821045	SI
61	4	88	21	0.1804203987121582	SI
62	4	518	30	1.2172877788543701	SI
63	2	203	50	0.09633231163024902	SI
64	4	357	30	0.8473446369171143	SI
65	4	249	41	0.9171216487884521	SI
66	3	110	15	0.14075064659118652	SI
67	3	24	8	0.02359914779663086	SI
68	3	308	28	0.5439369678497314	SI
69	3	1605	37	11.953489780426025	SI
70	4	32	26	0.08191156387329102	SI
71	2	36	21	0.05992603302001953	SI
72	3	951	40	2.609591245651245	SI
73	3	202	25	0.49593043327331543	SI
74	4	796	34	3.7185399532318115	SI
75	4	442	34	1.6697373390197754	SI
76	3	356	56	0.832115888595581	SI
77	4	437	55	1.5492124557495117	SI
78	5	1862	33	36.36634087562561	SI
79	3	106	18	0.06461381912231445	SI
80	4	1399	38	9.235090017318726	SI
81	3	68	12	0.04449200630187988	SI
82	3	27	14	0.02132272720336914	SI
83	4	6185	47	295.4396297931671	SI
84	3	3006	68	37.44927191734314	SI
85	3	660	51	2.838948965072632	SI
86	4	61	25	0.14787054061889648	SI
87	4	7488	9999	300.0026400089264	NO
88	3	2315	63	30.153833866119385	SI
89	4	1057	35	5.009212255477905	SI
90	4	1068	16	6.507360935211182	SI
91	4	117	14	0.16022968292236328	SI
92	3	1343	48	4.393462419509888	SI
93	8	5607	9999	300.0237467288971	NO
94	3	464	31	0.6515257358551025	SI
95	8	9	8	0.15954113006591797	SI
96	3	1105	37	2.736973524093628	SI
97	5	98	41	0.39895057678222656	SI
98	5	6907	9999	300.0019676685333	NO
99	4	8020	9999	300.0112018585205	NO
100	4	2257	56	20.45092487335205	SI
101	4	280	15	1.298213005065918	SI
102	4	1191	44	8.70063328742981	SI
103	4	128	12	0.296062707901001	SI
104	3	291	27	0.49396824836730957	SI

Tabla A.11: Resultados con A* con el segundo experimento 2/3.

Experimento 2					
Nivel	**Número de cajas**	**Nodos expandidos**	**Profundidad alcanzada**	**Tiempo**	**¿Solucionado?**
105	8	6359	9999	300.04280734062195	NO
106	5	4514	50	135.80188488960266	SI
107	11	120	10	2.1092488765716553	SI
108	4	7065	9999	300.02032256126404	NO
109	5	6684	9999	300.02042412757874	NO
110	4	1188	14	4.7207677364349365	SI
111	6	5586	9999	300.0084924697876	NO
112	5	7105	9999	300.0012629032135	NO
113	4	6833	9999	300.0224058628082	NO
114	6	8204	9999	300.0094406604767	NO
115	5	1201	29	11.861238718032837	SI
116	5	133	14	0.54402756690979	SI
117	5	5712	9999	300.01035237312317	NO
118	4	1625	44	12.710602760314941	SI
119	3	82	18	0.21573209762573242	SI
120	4	2553	64	17.471702575683594	SI
121	5	3239	47	59.47453331947327	SI
122	5	6580	9999	300.0250012874603	NO
123	5	7642	9999	300.0353193283081	NO
124	3	193	39	0.48508667945861816	SI
125	4	900	38	4.247798919677734	SI
126	7	5997	9999	300.01860094070435	NO
127	4	1046	32	3.754856824874878	SI
128	4	725	19	1.72379732131958	SI
129	5	1962	22	16.73691725730896	SI
130	4	2548	36	38.81625580787659	SI
131	4	2404	31	34.468674182891846	SI
132	4	885	37	4.161886930465698	SI
133	5	2166	39	25.560125827789307	SI
134	4	7991	9999	300.02317214012146	NO
135	4	1383	36	12.826777935028076	SI
136	4	393	25	1.688828468322754	SI
137	4	6388	9999	300.01447224617004	NO
138	5	6464	9999	300.0133767127991	NO
139	6	6422	9999	300.0338935852051	NO
140	4	7959	9999	300.0174798965454	NO
141	6	8619	9999	300.00012040138245	NO
142	4	2141	20	28.996750354766846	SI
143	6	6456	9999	300.01507449150085	NO
144	16	3206	9999	300.0456202030182	NO
145	12	3054	9999	300.0859248638153	NO
146	12	2951	9999	300.00734543800354	NO
147	3	2566	50	24.789737462997437	SI
148	4	2930	49	52.87408256530762	SI
149	4	464	35	1.4791040420532227	SI
150	5	6278	9999	300.0126886367798	NO
151	4	3444	50	71.5256199836731	SI
152	4	2860	37	52.150002241134644	SI
153	10	5103	9999	300.02201986312866	NO
154	1	3	2	0.004260063171386719	SI
155	11	200	175	0.7114651203155518	SI

Tabla A.12: Resultados con A* con el segundo experimento 3/3.

Apéndice B

Summary

Introduction

Sokoban is a puzzle video game created by Hiroyuki Imabayashi in the eighties. The objective of this game is very simple, boxes must be transported, with the fewest moves, to some positions of a warehouse. The only way to move this boxes is pushing, and can only move a box in every action.

That simplicity allows any user may face these levels. The collection of problems consists of 90 levels, in which the difficulty increases progressively. In addition, some users of the community provide the labyrinths that they develop, allowing always been a challenge available.

The main aim for any player is to solve the maze, however, the more experienced players try to beat your own scores, using as a measure pushes or moves of the player.

Although it seems affordable, Sokoban is a real challenge of logic. The difficulty of solving each level is the interaction between the different boxes of each maze, multiple moves you can perform each turn and causes some pushes the level is impossible to solve.

In the area of computer science this problem has proposed as a single agent search. The games that have always been used to investigate, within this area, have

been N-puzzle or the Rubik's cube, these games have a small branching factor[1] and depths[2] moderate to which the solution is reached.

In the game of Sokoban it is more complicated, not only for having a high branching factor comparable to other games like chess. In addition, to find the solution needed much pushing. This combination creates a search tree so large that substantially complicates finding the optimal solution[3]. When analyzing the complexity was determined that belongs to the complexity class PSPACE-Complete [1].

[1]Possible movements in each current situation

[2]Movements have been made since the beginning of the game

[3]Solution to the problem by running the fewest possible movements of boxes.

Objectives

The objective of this dissertation is mainly based on research on a method of solving a puzzle games with a single agent game, Sokoban.

The first objective will be to study this logic game and make an issue of Artificial Intelligence, mainly based on heuristic search, although other branches to help reduce the complexity of the problem and achieve better results will be studied.

Therefore, the primary objective will be based on finding an optimal solution to any game map, for this the complexity of the problem will be studied and necessary to achieve the goal of each of the labyrinths of the selected collection decisions.

To implement all these techniques or different methods of finding the solution to levels as well as the necessary tests performed, to ensure the smooth operation of the project, Python is used as the programming language in order to learn a new language, not used throughout the degree.

Finally, an intelligent agent that runs needed to reach the goal actions, based on all the research conducted during this project will be implemented.

To clarify the research objectives are established the following points:

1. Study the state of the art in solving Sokoban game.

2. Design a program to play Sokoban, finding the optimal solution to different mazes collection of levels provided.

3. Different search algorithms and heuristics will be implemented.

4. An empirical evaluation of different combinations of search algorithms and heuristics will be performed.

5. The reasons why the agent produced is able, or not, to find the solutions of the problems will be studied.

6. The implementation will be performed with Python, in order to learn a new programming language.

Design decisions

The main problem to solve the game Sokoban, using heuristic search, is the exponential memory consumption. In this situation a number of techniques are developed, with the main objective to reduce this complexity, which are discussed below.

Collection mazes

The collection original problem has too large boards so has decided to opt for a collection of smaller levels. While still maintaining the difficulty encountered in power be locked with a wrong move.

The selected collection is **Microban**, created by *David W. Skinner*. This decision is based on that, although the dimensions are smaller, maintain their complexity levels and have been used as a test suite in other investigations. The collection consists of 155 problems which covers a wide range of problems to solve. This way you can measure the quality of the project and the results obtained.

Moves vs pushes

If each state is represented as a push any box, the number of states is significantly reduced to represent the same situation, in the other case several states would be required to achieve the same position in the game.

For this reason the choice made, each state is represented as the push of a box to get reduce the size of the search tree. The solutions can be achieved are still optimal in the number of pushes and also in most levels, this solution is also optimal in the number of movements, although this can not be guaranteed.

Finding deadlocks

Once play Sokoban, we realize that if we put the boxes in some positions, the problem can not be solved. These positions are known as *deadlocks*. Image B.1

is shown as a sequence of actions can leave a level without solution.

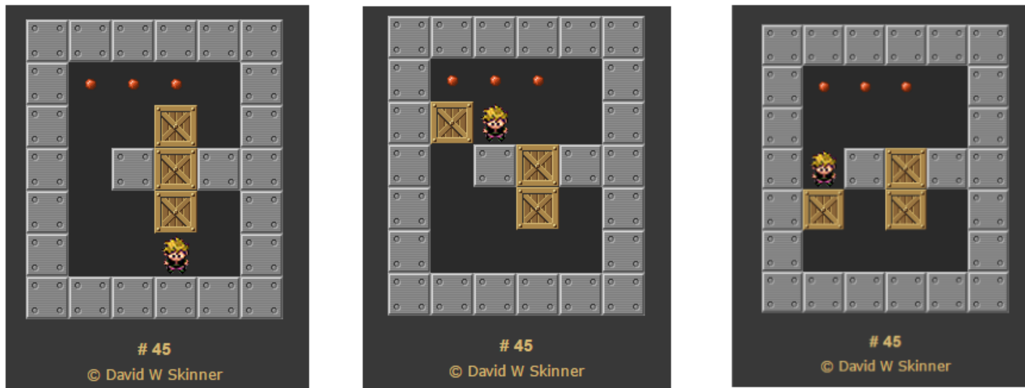


Figura B.1: Sequence of actions that leave the level unsolvable.

These positions are classified in this work, in two types: fixed and dynamic.

Fixed deadlocks depend only on the map, regardless of the movements that take place throughout the game. In the picture B.2 can be seen in red fixed deadlocks found.

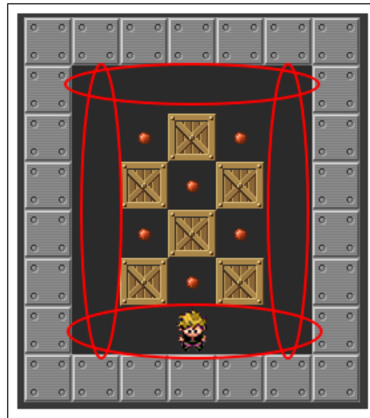


Figura B.2: Example of fixed deadlocks

Dynamic deadlocks are caused to move the boxes and when hitting a wall or other boxes, they can not move. This task must be calculated in each state who try to expand to avoid creating successors unsolvable. In the picture B.3 shows some examples.

With this detection system can not guarantee that all deadlocks labyrinth are located, but it can ensure that all positions found are deadlocks.

Figura B.3: Example of dynamic deadlocks

Matching

Another way to detect the tree branches that have no solution is the task of assigning boxes. Each box you must match a goal, if we are not able to ensure that there is a direct allocation for each objective, this road has no solution. The chosen algorithm for the allocation is Kuhn or Hungarian algorithm.

Locate repeated states

Another way to reduce the number of generated states is to check whether a state has been expanded and therefore eliminate their successors not to visit twice, since it will generate the same successors to the original state.

On many occasions, different sequences of actions are capable of generating the same state. These positions are called transpositions.

This improvement is significant, since it considerably reduces the number of states and thus the program is able to solve more problems.

Results

In this chapter the results obtained by each of the developed agents are analyzed. The quality of the agent is determined by the number of labyrinths can solve. In addition, other parameters such as the number of nodes explored and the time it takes to solve are compared.

Notably, without the design decisions taken in the previous chapter, to reduce the size of the problem search, it would be impossible to solve even a single level, or this or any other collection.

Due to the complexity of the problem, it is likely that resources are exhausted if it takes long to find the solution. For this reason, it is necessary to set limits on time or memory of the executions of the program. Normally, other research set half hour maximum time to solve each level. For this project it was decided to use only five minutes. If the number of solved problems is under this limit would be extended.

Initial Experiments

To establish a comparison of implemented agents, the basis is the brute force algorithm. In this case, the total of 155 collection has problems, 45 problems are solved. The number of expanded states is very high and the maximum depth to find the solution is 27 levels. This makes for any problem is virtually impossible to find a solution by brute force because most levels require a lot more jostling to reach the goal.

The first experiment with the A* algorithm using the Manhattan distance, is able to solve optimally 120 levels, this is a breakthrough as it is capable of resolving more than half of the collection problems. In addition, about 85 % reduces the number of states expanded at levels that were solved by brute force. He also experimented with the algorithm IDA* but in this case we have only been able to solve 72 mazes compared with A* are quite less.

Then it was decided to perform another experiment with Kuhn as heuristics.

This heuristic is much more informed although this calculation is quite expensive. In this way we can see how both behave heuristics.

In this case both algorithms improve the results obtained in the first experiment, which allows us to say that a better informed heuristic greatly improves search in this domain.

Given these results, the research focuses from this time around the A* algorithm. This decision is because it is able to solve more problems, reduced by 85 % the number of nodes expanded regarding brute force search and ensures that the solution found is optimal if the heuristic is admissible. In graphs B.4 and B.5 are displayed, both expanded nodes and time, in which both experiments are compared.

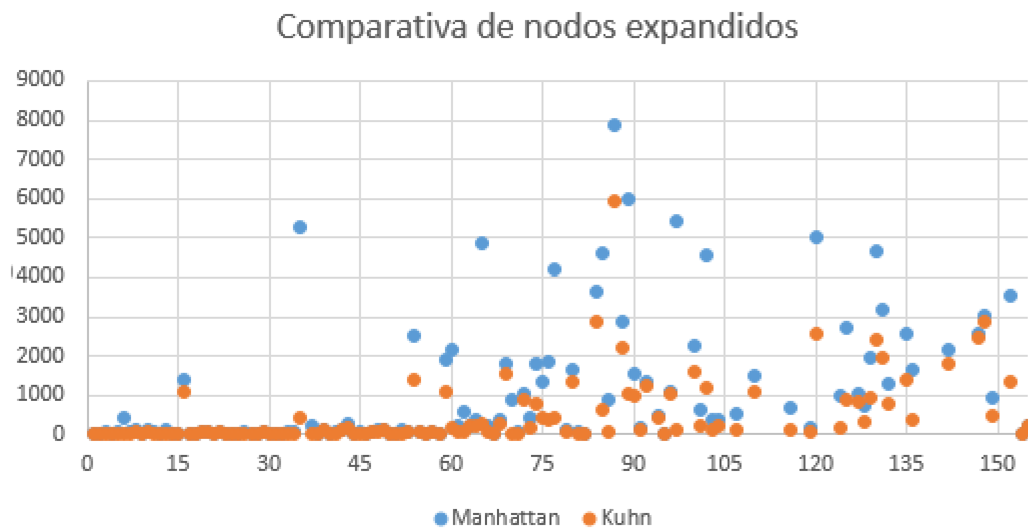


Figura B.4: Comparative nodes expanded of Manhattan against Kuhn

In this way it can be demonstrated how to solve the problem of Sokoban, using AI, specifically heuristic search.

Manhattan distance to the assignment of Kuhn

In this experiment, computing the Manhattan distance to assignments Kuhn algorithm. This calculation has the same computational cost optimal Manhattan distance but, provides greater information since each box corresponds to a goal.

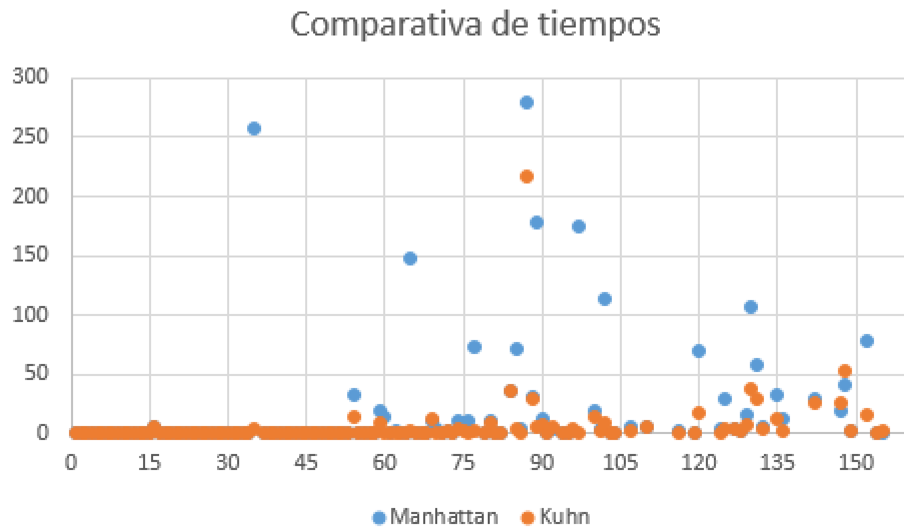


Figura B.5: Comparative times of Manhattan against Kuhn

In table B.1 a general comparison of the results is performed, against the best found so far experiment, Kuhn.

Performance against Kuhn	
Problems solved	**124** (4 less)
Expanded nodes	Increases 31,5 %
Time	Increases 19,8 %

Tabla B.1: Performance of this experiment against Kuhn

This experiment, worsens in the main objectives we are trying to solve, first solve four problems less and is worse off informed so it is necessary to significantly increase the states explored to reach a solution.

For all these reasons, this experiment is rejected for the progress of the project.

Manhattan and Kuhn

In this experiment, it has opted to keep the algorithm Kuhn as main heuristics, until the initial limit is reached. From this moment, looking for a faster start

to use the Manhattan distance.

In table B.2 a summary of the results obtained in this experiment is as follows:

Performance against Kuhn	
Problems solved	**127** (1 less)
Expanded nodes	Increases 5 %
Time	Increases 2 %

Tabla B.2: Performance against Kuhn

This experiment is able to improve the speed of nodes explored by the algorithm. In most levels the solution with the same number of explored states is reached, meaning it has not been necessary to use the Manhattan distance within this heuristic.

For all these reasons this experiment is very interesting and an alternative to Hungarian method from the limit, which can be of great help to solve some levels of the game.

Manhattan to assignments and to the nearest goal

In this experiment, it is to use the Manhattan distance assignment algorithm Kuhn and once the limit is reached, following the same assumptions as in the previous experiment, is changed by the Manhattan distance optimal. In this way, we want to achieve high speed exploring nodes and be able to find a solution although not optimal.

The results obtained in this experiment are shown in table B.3.

Performance against Kuhn	
Problems solveed	**123** (5 less)
Expanded nodes	Increases 36,2 %
Time	Increases 22,2 %

Tabla B.3: Performance against Kuhn

This experiment has been dismissed for failing to improve the results obtained so far and not finding any feature that can help the progress of this work.

Conclusions

Once the project is completed, the conclusions are positive. All objectives have been achieved and the results obtained are even better than expected at the beginning of the investigation.

This project began with the idea of developing an agent which is able to find the optimal solution of a few simple levels of the game. This objective has been exceeded when 128 levels of 155 total Microban collection have been resolved.

Techniques on data processing have managed to greatly reduce the search space. If not, it could not solve any level.

The algorithm with the best results have been obtained is A*. This occurs because the memory consumption in these problems never reaches the total that are available. In that case you should use the iterative algorithm depth, IDA* that can reduce the spatial complexity revisiting previously explored states. For this reason, the time limit of five minutes set, the algorithm IDA* is able to solve fewer levels and therefore is selected A*.

In this way it has been possible to show that the problem of Sokoban can be approached from the perspective of the heuristic search.

The significance of the information provided by the heuristics in the search field is also shown. From brute force search, in which we are able to solve 45 problems without using any guide are achieved reach 120 levels optimally solved only with the application of a heuristic poorly informed, Manhattan distance. Although the difficulty in finding heuristics informed at this domain, it is decided to use the minimum allocation calculated by the algorithm Kuhn in each of the states, due to this eight more levels are solved and the number of states required is considerably reduced to achieve the optimal solution.

Kuhn running algorithm in each of the nodes is too expensive, for this reason, we have tried to achieve a balance in the heuristics used. To this end, it has tried to find a heuristic to provide information, but calculate it much easier. Three different experiments were performed, in which attempts to combine several heuristics, but

neither these experiments improve the results obtained. It also demonstrates the difficulty of finding good heuristics in this domain.

The main limitations have been especially hardware reasons. The limitation of memory has forces us to set a maximum execution time and some levels are not resolved because need more time to reach the goal.

A further limitation, although less important, which has been found is the programming language. *Python* is an interpreted language. This makes it much slower than other compiled as *C* and being a task that requires many calculations, is not able to find the solution in some mazes.

Future work

Once the project is completed, a future research established that could help improve the program and to obtain better results.

- **Improve algorithms**. Improve the algorithms and representation of states to decrease memory consumption.
- **Select algorithms and heuristics appropriate to each level**. This will allow greater versatility and reach solutions based on the maze, based on similar problems.
- **Add new techniques that reduce the search space**. These techniques are indispensable for solving levels, so you can add some of those mentioned in other investigations or any other that has the same purpose.
- **Cooperative sokoban**. Using multiple players within the same maze, for this purpose should use algorithms that take into account several agents as A* algorithm cooperative or LRA*.

Bibliografía

[1] J. C. Culberson, "Sokoban is pspace-complete," Tech. Rep. 97-02, Department of Computing Science, University of Alberta, April 1997.

[2] A. Turing, "Computing machinery and intelligence," *Mind*, 1950.

[3] N. J. Nilsson, *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann, first ed., 1998.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, third ed., 2009.

[5] E. F. Moore, "The shortest path through a maze," in *Proceedings of an International Symposium on the Theory of Switching*, (Cambridge, Massachusetts, 2–5 April 1957), pp. 285–292, Cambridge: Harvard University Press, 1959.

[6] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[7] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society of Industrial and Applied Mathematics*, pp. 32–38, March 1957.

[8] H. W. Kuhn, "Variants of the hungarian method for assignment problems," *Naval Research Logistics Quarterly*, vol. 3, pp. 253–258, 1956.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[10] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[11] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[12] A. Junghanns, *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.

[13] A. Junghanns and J. Schaeffer, "Sokoban: A challenging single-agent search problem," in *Workshop on Using Games as an Experimental Testbed for AI Research*, Proceedings IJCAI-97, August 1997.

[14] A. Junghanns and J. Schaeffer, "Single-agent search in the presence of deadlock," in *Proceedings of AAAI-98*, (Madison WI, USA), pp. 419–424, July 1998.

[15] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1–2, pp. 219–251, 2001.

[16] A. Botea, M. Müller, and J. Schaeffer, "Using abstraction for planning in sokoban," in *Proceedings of the 3rd International Conference on Computers and Games*, pp. 285–292, Springer, 2002.

[17] T. Virkkala, "Solving sokoban," Master's thesis, University of Helsinki, 2011.