

Práctica 2: Satisfacibilidad Lógica y Búsqueda Heurística

Heurística y Optimización

Grado de Ingeniería Informática, 2018 - 2019



Autores:

Juan José Garzón Luján (100363861)
Carlos García Cañibano (100363813)

Índice de contenidos

Parte 1:

páginas 3-7

Modelización del problema

páginas 3-4

Resolución y análisis

páginas 4-7

Parte 2:

páginas 7-15

Modelización del problema

páginas 7-8

Implementación del algoritmo

páginas 8-9

Funciones heurísticas

páginas 9-10

Casos de prueba y análisis

páginas 10-15

Conclusiones

página 15

Parte 1

Modelización del problema

A la hora de modelizar el problema de satisfacibilidad lógica necesitamos generar todas las cláusulas que dan lugar a la resolución de nuestro problema. Una vez tenemos todas las cláusulas en forma conjuntiva deberemos pasarlo a nuestro programa utilizando jacop.

Nuestro problema se compone de 4 restricciones:

- **Para cada elemento solo una posición verdadera:** Esta restricción restringe el número de apariciones de un elemento ya que no tiene sentido que elemento tenga varias posiciones. Para ello tenemos la siguiente cláusula en Forma Normal Conjuntiva:

$$(X_{ij} \vee X_{ij} \vee X_{ij} \dots) \forall i, j \text{ donde haya una casilla libre}$$

De ello se encarga nuestra función addClause, la cuál recibe un SatWrapper, una matriz bidimensional y el fichero. Así, esta función cada vez que es invocada recorre todo el fichero buscando las casillas libres y poniéndolas en una restricción para que solo se permite la aparición única por objeto a insertar en el mapa.

- **Para cada casilla solo se le puede asignar un elemento:** Esta restricción evita que los elementos se pisen unos a otros y así impidan la correcta colocación de los elementos. Para ello tenemos la siguiente cláusula en Forma Normal Conjuntiva:

$$(\neg A_{ij} \vee \neg S_{0ij} \vee \neg S_{1ij} \dots \vee \neg S_{kij}) \forall i, j \text{ donde haya una casilla libre, } k \text{ es el número de serpientes}$$

De ello se encarga nuestra función addSingleElementClause que irá haciendo esta cláusula con cada casilla libre en el libre. Para ello recibe un SatWrapper, las matrices de AI y serpientes y el fichero a leer.

- **Para cada fila solo una serpiente:** Esta restricción regula el número de serpientes, ya que el problema nos dice que solo puede haber una por fila. Para ello tenemos la siguiente cláusula en Forma Normal Conjuntiva:

$$(\neg S_{000} \vee \neg S_{10j}) \forall j \text{ que será una casilla libre de la línea}$$

Esta cláusula se debe generar con todas las combinaciones posibles entre serpientes y casillas vacías de esa fila. El primer número indica la serpiente. Los otros dos números irán cambiando respecto a la fila y columna de esa celda vacía. Esta cláusula es un ejemplo de que si en se puede poner una serpiente en la casilla 00 se creará una cláusula con otra serpiente en la siguiente casilla vacía de esa fila.

Este proceso lo hará nuestra función addSnakeRowClause que recibirá la matriz de las serpientes, un SatWrapper y el fichero a leer. Hará el procedimiento anteriormente explicado para generar todas las cláusulas.

- **No serpientes en la misma fila ni columna de AI:** Esta restricción impide que ninguna serpiente coincida en la misma fila ni columna de donde se coloca AI. Para ello tenemos las siguientes cláusulas en Forma Normal Conjuntiva:

$$(\neg A_{00} \vee \neg S_{00j}) \forall j \text{ que será una casilla libre de la fila.}$$

$(\neg A_{00} \vee \neg S_{0i0}) \forall i$ que será una casilla libre de la columna.

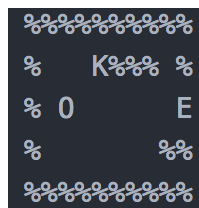
Esta clausula se deberá generar con cada casilla libre fijando una para Al y haciendo parejas con las posiciones de las distintas serpientes en las celdas vacías por fila y columna. Estos son ejemplos de donde Al puede ir en la casilla 00 y se hace pareja con una serpiente (en este caso la primera) en la siguiente posición libre de esa fila o columna perteneciente a las coordenadas de Al. Esto se repetirá con todas las serpientes y cambiando la posición de Al.

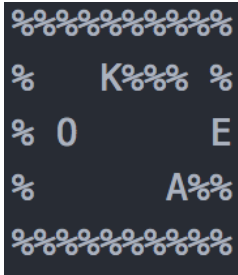
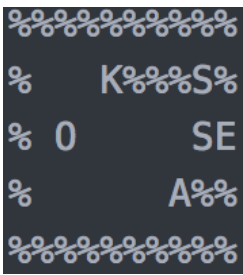
Esta restricción se lleva a cabo en addAlClause que recibe un SatWrapper, las matrices de Al y serpientes y el fichero a leer. Hará lo descrito anteriormente para que esta restricción se cumpla.

Resolución y análisis de las pruebas

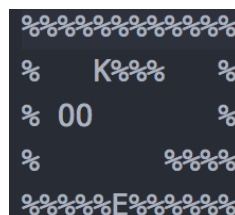
Probaremos si nuestra implementación hace lo que debe hacer para resolver de la manera adecuada los distintos mapas y así poder analizar los resultados.

MAPA 1



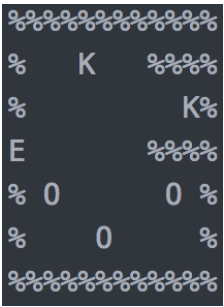
N.º Serpientes	0	2	3
Resultado			Por pantalla imprime: "Solution not found"

MAPA 2



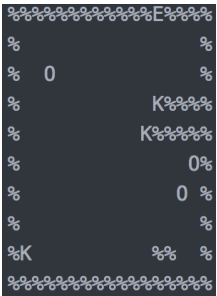
N.º Serpientes	1	2	3
Resultado			Por pantalla imprime: “Solution not found”

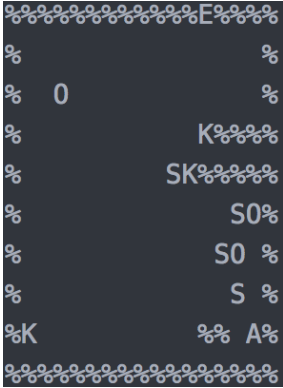
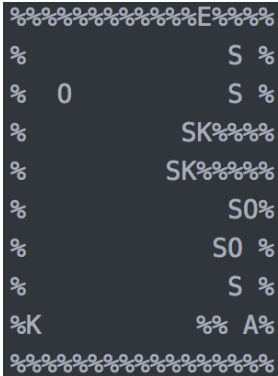
MAPA 3



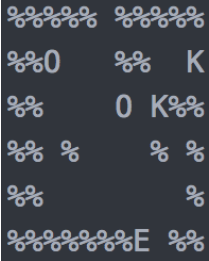
N.º Serpientes	2	4	6
Resultado			Por pantalla imprime: “Solution not found”

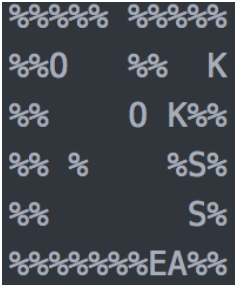
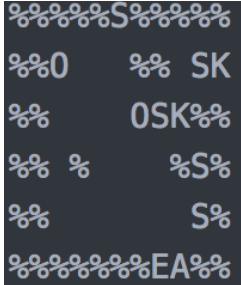
MAPA 4



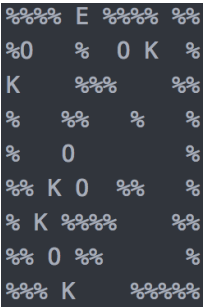
N.º Serpientes	4	7	8
Resultado			Por pantalla imprime: “Solution not found”

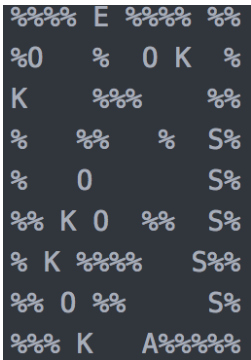
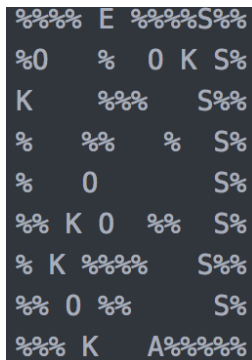
MAPA 5



N.º Serpientes	2	5	7
Resultado			Por pantalla imprime: “Solution not found”

MAPA 6



N.º Serpientes	5	8	9
Resultado			Por pantalla imprime: “Solution not found”

Podemos ver con estas diferentes pruebas que nuestra implementación se comporta de la manera adecuada en todos los escenarios posible. Nunca coloca a AI en la misma fila o columna que una serpiente , nunca hay 2 serpientes en una misma fila y en el caso de que se necesiten mas filas para colocar a las serpientes el programa no encuentra solución. Vemos también que nunca se pisan los objetos del mapa antes de colocar a AI y a las serpientes, todo está en el mismo lugar de antes.

Parte 2

Modelización del problema

Antes de proceder a la implementación del algoritmo para solucionar el problema, debemos definir que estructura de datos utilizaremos para representar los estados y que operaciones se pueden aplicar para poder avanzar. En primer lugar pasaremos a describir la estructura de datos utilizada:

State.h

- **ID:** Identificador único para cada estado. Nos servirá para resolver el camino una vez que el problema ha sido resuelto.
- **FATHER_ID:** Relación única entre un estado padre y el estado actual. Nos servirá para resolver el camino una vez que el problema ha sido resuelto.
- **AL_X_POS:** Posición actual en el eje x de AI dentro del mapa.
- **AL_Y_POS:** Posición actual en el eje y de AI dentro del mapa.
- **FATHER_X:** Posición inmediatamente posterior en el eje x de AI dentro del mapa. Nos servirá para realizar comprobaciones sobre, por ejemplo, la nueva posición de una roca.
- **FATHER_Y:** Posición inmediatamente posterior en el eje y de AI dentro del mapa. Nos servirá para realizar comprobaciones sobre, por ejemplo, la nueva posición de una roca.
- **STATE_KEYS:** Todas las llaves que todavía no han sido recogidas en el estado actual con sus respectivas posiciones. El número de llaves cambia según el estado, por eso las almacenamos. Además, tener sus posiciones, nos ayuda a identificarlas.

- **STATE_ROCKS:** Todas las rocas que hay presentes en el mapa con sus respectivas posiciones. La posición de las rocas cambia según el estado, por eso las almacenamos. Además, tener sus posiciones, nos ayuda a identificarlas.
- **F:** El valor para el estado actual de $f(n)$ siguiendo la fórmula $f(n) = g(n) + h(n)$. Nos servirá para ordenar los estados dentro de la lista ABIERTA.
- **G:** El valor acumulado para el estado actual de todos los costes asumidos para poder situarnos en la posición actual. Nos servirá para calcular $f(n)$ y para saber el coste total de la solución encontrada.
- **H:** El valor heurístico del estado basado en una de las tres funciones heurísticas utilizadas por el algoritmo. Guiará A* y nos permitirá encontrar soluciones admisibles y óptimas. También es necesario para calcular $f(n)$.

Ahora que la estructura de datos está definida, pasaremos a describir los diferentes operadores utilizados para intentar encontrar la solución dentro del mapa:

Actualizar posición:

Para que AI pueda conseguir su objetivo, es necesario que cambie su posición en el mapa. Para que esta acción se pueda realizar, no debe existir ninguna serpiente ni en la misma fila ni en la misma columna, tampoco deben ser excedidos los límites del mapa y la casilla objetivo no puede contener una serpiente o pared. Una vez realizada la actualización, los únicos valores de la estructura modificados, son las posiciones en el eje x e y dentro del mapa. (A no ser que se mueva una roca o se coja una llave, que será explicado más adelante)

Coger llave

Para que AI pueda salir del laberinto, necesita conseguir todas las llaves que se encuentran distribuidas en el mapa. Por este motivo, si la casilla objetivo del siguiente movimiento de AI contiene una llave, este la coge. “Coger una llave” significa actualizar la colección de llaves que almacena el estado actual de AI, borrando la encontrada.

Mover roca

En ocasiones, AI necesita mover una roca para descubrir un camino mejor, evitar una serpiente, coger una llave, encontrar la salida, o cualquier combinación de las acciones descritas anteriormente. Para que una roca pueda ser movida, tiene que ser seguro para AI (al mover la roca, no puede significar la muerte de AI) y la futura posición de la misma tiene que estar contenida en los límites del mapa. “Mover una roca” significa actualizar la colección de rocas de manera que identificamos la roca encontrada y cambiamos su posición actual dentro de la colección, con la nueva posición a la que será movida.

Implementación del algoritmo

En esta sección se clarificarán algunos detalles acerca del código desarrollado que pudiesen no quedar claros o necesitar de una explicación adicional a la proporcionada con los comentarios.

El código está diseñado desde un punto de vista modular. Al utilizar muchas funciones para realizar pequeñas tareas nos aseguramos de que el código es mantenible, reutilizable y más legible. Muchas de las funciones necesitan bastantes parámetros para ser invocadas, pero evitan la repetición de los mismos fragmentos de código una y otra vez. Un ejemplo de esto sería la función **apply_movement**. En esta función se ha ideado

un sistema de suma de coordenadas que en tiempo de uso permite parametrizar completamente todos los posibles movimientos de AI.

También cabe destacar la utilización de un lenguaje de desempeño potente como C++. El paso por referencia de los atributos nos permite realizar un programa menos pesado en memoria y presumiblemente rápido.

El último detalle a destacar en este apartado serían las definiciones customizadas de operadores que hemos diseñado para la clase **state**. De esta manera las funciones de ordenación, utilizarán el criterio exclusivo de menor valor de $f(n)$ para ordenar sus componentes. También se ha definido nuevamente el operador de igualdad entre estos objetos. Hemos seguido un criterio de comparación entre todos los elementos relevantes que definen un estado concreto como si se tratase de una “fotografía en el tiempo”. Se comparan las posiciones, llaves y rocas una a una para comprobar que dos estados sean exactamente iguales.

Gracias a este enfoque, nos ha sido realmente sencillo implementar la versión adicional con movimientos en diagonal. El código está bien estructurado y las modificaciones añadidas se realizan de manera sencilla. Solo hemos tenido que añadir las combinaciones de sumas y restas en los ejes x e y para obtener todos los movimientos y aumentar el empuje de las rocas en su respectiva función **move_rock**. ($x+1, x-1, y+1, y-1, x+1;y+1, x+1;y-1, x-1;y+1, x-1;y-1$)

Funciones heurísticas

En esta sección describiremos las diferentes funciones heurísticas utilizadas para resolver el problemas. Explicaremos hasta que punto son informadas y en qué casos de uso pueden llegar a ser más efectivas unas u otras. Sin embargo, en el último punto de la memoria, haremos un análisis de resultados con diferentes mapas y diferentes heurísticas.

- Distancia de Manhattan entre la posición actual de AI, la posición de todas las llaves que quedan por recoger y la distancia a la casilla de salida:

$$\sum^{N^{\circ} llaves} (Distancia\ de\ Manhattan\ llave\ K) + (Distancia\ de\ Manhattan\ salida\ E)$$

Es una heurística admisible y, pensamos, altamente informada. La distancia de Manhattan representa de forma exacta el mapa y los movimientos que AI puede realizar sobre él. (Descartando los diagonales...) Además, recoger las llaves siempre es una prioridad y hasta que no se consigan todas, AI no encontrará “atractiva” la salida. Pensamos que identifica bien los objetivos del problema y se ajusta a la mecánica del mismo.

- Distancia Euclídea entre la posición actual de AI, la posición de todas las llaves que quedan por recoger y la distancia a la casilla de salida:

$$\sum^{N^{\circ} llaves} (Distancia\ Euclídea\ llave\ K) + (Distancia\ de\ Euclídea\ salida\ E)$$

Es una heurística admisible. Sin embargo, pensamos que está menos informada que la anterior para los movimientos fila/columna. En el caso de los movimientos diagonales, se ajusta mejor al problema y presumiblemente generará menos nodos para conseguir la solución óptima. (Lo comprobaremos más adelante...) Para cualquier caso, es funcional y es también informada.

- Distancia Euclídea entre la posición actual de AI, la distancia más cercana de entre todas las llaves que quedan por recoger y la distancia a la casilla de salida:

$$\min(Distancia\ Euclídea\ llave\ K) + (Distancia\ Euclídea\ salida\ E)$$

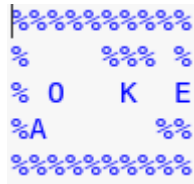
Es una heurística admisible. Sin embargo, aunque no parezca intuitivo, no siempre es la mejor idea ir a por la llave más cercana si hay otra acción que interviene positivamente en la resolución del mapa o existe un camino por el cual recoger todas las llaves que no implique recoger la más cercana primero. No obstante, nos servirá para comparar resultados y descubrir como, en ocasiones, el diseño de funciones heurísticas requiere de un pensamiento analítico poco convencional.

Descritas todas las heurísticas que utilizaremos para nuestro problema, pasaremos a la sección de pruebas donde documentaremos y explicaremos todos los resultados obtenidos con multitud de mapas.

Casos de prueba y análisis en función del mapa/heurística

Para la realización de esta parte utilizaremos los mapas añadidos como ejemplo en Aula Global y dos adicionales de nuestra propia creación.

lab1.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(3, 1) → (3, 2) → (3, 3) → (2, 3) → (2, 4) → (2, 5) → (2, 6) → (2, 7) → (2, 8) → (2, 9)	(3, 1) → (3, 2) → (3, 3) → (3, 4) → (3, 5) → (3, 6) → (2, 6) → (2, 7) → (2, 8) → (2, 9)	(3, 1) → (3, 2) → (3, 3) → (3, 4) → (3, 5) → (3, 6) → (2, 6) → (2, 7) → (2, 8) → (2, 9)
Nodos expandidos	24	28	28
Coste total	18	18	18
Tiempo tomado	0.00468 segundos	0.005098 segundos	0.005172 segundos
Longitud del camino	9	9	9

En este primer caso la solución encontrada es óptima y en el menor número de movimientos. Como suponíamos, al no estar habilitados los movimientos en diagonal para esta versión, la primera heurística está más informada y se ajusta mucho mejor al caso del problema. La ejecución es más rápida y se expanden menos nodos. A medida que la complejidad del mapa aumente, descubriremos si estas características iniciales se acentúan.

lab2.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(1, 1) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (1, 4) → (2, 4) → (2, 5) → (3, 5) → (4, 5)	(1, 1) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (1, 4) → (2, 4) → (3, 4) → (3, 5) → (4, 5)	(1, 1) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (1, 4) → (2, 4) → (3, 4) → (3, 5) → (4, 5)
Nodos expandidos	50	68	68
Coste total	20	20	20
Tiempo tomado	0.005554 segundos	0.01365 segundos	0.011406 segundos
Longitud del camino	9	9	9

Para este caso se repiten los mismo patrones descritos en la primera prueba. La solución es óptima y en el menor número de movimientos posibles. Parece que el comportamiento de las diferentes heurísticas sigue siendo el mismo. La primera es la más informada y, de momento, la más óptima de cara a resolver el problema.

lab3.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(1, 7) → (1, 6) → (1, 5) → (1, 4) → (2, 4) → (3, 4) → (3, 5) → (3, 6) → (4, 6) → (4, 7) → (5, 7) → (5, 8) → (5, 9) → (5, 8) → (5, 7) → (4, 7) → (4, 6) → (4, 5) → (4, 4) → (4, 3) → (4, 2) → (4, 1) → (3, 1) → (3, 0)	(1, 7) → (1, 6) → (1, 5) → (1, 4) → (2, 4) → (3, 4) → (3, 5) → (3, 6) → (4, 6) → (4, 7) → (5, 7) → (5, 8) → (5, 9) → (5, 8) → (5, 7) → (4, 7) → (4, 6) → (4, 5) → (4, 4) → (4, 3) → (4, 2) → (4, 1) → (3, 1) → (3, 0)	(1, 7) → (1, 6) → (2, 6) → (3, 6) → (4, 6) → (4, 7) → (5, 7) → (5, 8) → (5, 9) → (5, 8) → (5, 7) → (4, 7) → (4, 6) → (4, 5) → (4, 4) → (3, 4) → (2, 4) → (1, 4) → (1, 3) → (1, 2) → (2, 2) → (2, 1) → (3, 1) → (3, 0)
Nodos expandidos	791	934	982
Coste total	48	48	48
Tiempo tomado	0.201587 segundos	0.202069 segundos	0.181708 segundos
Longitud del camino	23	23	23

En este caso también se encuentra la solución óptima y más corta. Sin embargo, el tiempo de ejecución resulta despreciable teniendo en cuenta las diferencias entre heurísticas. En términos de consumo de memoria, la primera sigue siendo la que menos nodos expande y sigue respetando el enunciado que mencionamos en la primera prueba. Parece cumplir dichas propiedades.

lab4.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(8, 1) → (8, 2) → (8, 3) → (8, 4) → (8, 5) → (8, 6) → (7, 6) → (7, 5) → (6, 5) → (6, 4) → (6, 3) → (6, 4) → (5, 4) → (5, 5) → (4, 5) → (3, 5) → (3, 4) → (2, 4) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (0, 12)	(8, 1) → (8, 2) → (8, 3) → (8, 4) → (8, 5) → (8, 6) → (7, 6) → (7, 5) → (6, 5) → (6, 4) → (6, 3) → (6, 5) → (5, 3) → (5, 4) → (5, 5) → (4, 5) → (3, 5) → (3, 4) → (2, 4) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (0, 12)	(8, 1) → (8, 2) → (8, 3) → (8, 4) → (8, 5) → (8, 6) → (7, 6) → (7, 5) → (6, 5) → (6, 4) → (6, 3) → (6, 5) → (5, 3) → (5, 4) → (5, 5) → (4, 5) → (3, 5) → (3, 4) → (2, 4) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (0, 12)
Nodos expandidos	2470	4398	6365
Coste total	60	60	60
Tiempo tomado	5.96191 segundos	14.789 segundos	22.7363 segundos
Longitud del camino	27	27	27

En este caso también se encuentra la solución óptima más corta. Sin embargo, los tiempos de ejecución y las diferencias de memoria se incrementan significativamente a medida que el mapa es más complejo. La primera heurística es la que mejor representa el problema y por este motivo consume aproximadamente $\frac{1}{3}$ de la memoria y el tiempo que las demás utilizan para encontrar la solución.

lab5.map

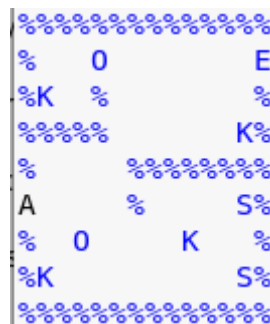


	Heurística 1	Heurística 2	Heurística 3
Camino	(5, 7) → (4, 7) → (4, 6) → (4, 5) → (3, 5) → (3, 6) → (2, 6) → (2, 7) →	(5, 7) → (4, 7) → (4, 6) → (4, 5) → (3, 5) → (3, 6) → (2, 6) → (2, 7) →	(5, 7) → (4, 7) → (4, 6) → (4, 5) → (3, 5) → (3, 6) → (2, 6) → (2, 7) →

	(2, 8) → (2, 9) → (2, 10) → (2, 11) → (2, 12) → (2, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (1, 7) → (0, 7)	(2, 8) → (2, 9) → (2, 10) → (2, 11) → (2, 12) → (2, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (1, 7) → (0, 7)	(2, 8) → (2, 9) → (2, 10) → (2, 11) → (2, 12) → (2, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (1, 7) → (0, 7)
Nodos expandidos	161	183	184
Coste total	50	50	50
Tiempo tomado	0.017644 segundos	0.01764 segundos	0.014701 segundos
Longitud del camino	23	23	23

En este mapa observamos que la presencia de rocas y serpientes interviene en la dificultad del algoritmo para encontrar la solución, pero no es determinante. La localidad de los objetivos y la amplitud del mapa es un factor de peso superior. El comportamiento de las heurísticas sigue los mismos patrones descritos anteriormente.

lab6.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(5, 0) → (5, 1) → (5, 2) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (7, 1) → (6, 1) → (6, 2) → (6, 3) → (6, 4) → (6, 5) → (6, 6) → (6, 7) → (6, 8) → (6, 9) → (6, 8) → (6, 7) → (6, 6) → (6, 5) → (5, 5) → (4, 5) → (3, 5) → (2, 5) → (1, 5) → (1, 4) → (1, 3) → (2, 3) → (2, 2) → (2, 1) → (2, 2) → (2, 3) → (1, 3) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (2, 12) → (3, 12) → (2, 12) → (1, 12) → (1, 13)	(5, 0) → (5, 1) → (5, 2) → (5, 3) → (6, 3) → (6, 2) → (6, 1) → (7, 1) → (6, 1) → (6, 2) → (6, 3) → (6, 4) → (6, 5) → (6, 6) → (6, 7) → (6, 8) → (6, 9) → (6, 8) → (6, 7) → (6, 6) → (6, 5) → (5, 5) → (4, 5) → (3, 5) → (2, 5) → (1, 5) → (1, 4) → (1, 3) → (2, 3) → (2, 2) → (2, 1) → (2, 2) → (2, 3) → (1, 3) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (2, 12) → (3, 12) → (2, 12) → (1, 12) → (1, 13)	(5, 0) → (5, 1) → (5, 2) → (5, 3) → (6, 3) → (6, 2) → (7, 2) → (7, 1) → (6, 1) → (6, 2) → (6, 3) → (6, 4) → (6, 5) → (6, 6) → (6, 7) → (6, 8) → (6, 9) → (6, 8) → (6, 7) → (6, 6) → (6, 5) → (5, 5) → (4, 5) → (3, 5) → (2, 5) → (1, 5) → (1, 4) → (1, 3) → (2, 3) → (2, 2) → (2, 1) → (2, 2) → (2, 3) → (1, 3) → (1, 4) → (1, 5) → (1, 6) → (1, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (2, 12) → (3, 12) → (2, 12) → (1, 12) → (1, 13)
Nodos expandidos	8480	11509	18508

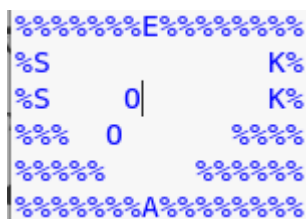
Coste total	100	100	100
Tiempo tomado	20.695 segundos	46.4869 segundos	55.4384 segundos
Longitud del camino	47	47	47

Mapa complejo que ensalza los datos obtenidos por el laboratorio 4. La primera heurística sigue siendo la mejor para realizar el cálculo en términos de memoria y tiempo. El conocimiento del problema queda mejor expresado en la primera que en ninguna otra.

Pruebas adicionales para la versión diagonal

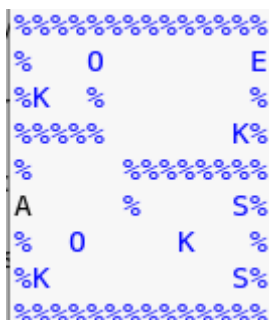
Para estas últimas pruebas utilizaremos los dos últimos mapas de diseño propio y analizaremos los resultados con el mismo formato que en pruebas anteriores.

lab5.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(5, 7) → (4, 6) → (3, 5) → (2, 6) → (2, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (1, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (0, 7)	(5, 7) → (4, 6) → (3, 5) → (2, 6) → (2, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (1, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (0, 7)	(5, 7) → (4, 6) → (3, 5) → (2, 6) → (2, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (1, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (0, 7)
Nodos expandidos	602	925	1339
Coste total	42	42	42
Tiempo tomado	0.314214 segundos	0.608162 segundos	0.77845 segundos
Longitud del camino	19	19	19

lab6.map



	Heurística 1	Heurística 2	Heurística 3
Camino	(5, 0) → (4, 1) → (5, 2) → (6, 3) → (6, 2) → (7, 1) → (6, 2) → (5, 3) → (4, 4) → (5, 5) → (6, 6) → (6, 7) → (6, 8) → (6, 9) → (6, 8) → (6, 7) → (6, 6) → (5, 5) → (4, 5) → (3, 6) → (2, 6) → (1, 5) → (1, 4) → (2, 3) → (2, 2) → (2, 1) → (1, 2) → (2, 3) → (1, 4) → (2, 5) → (2, 6) → (3, 7) → (3, 8) → (3, 9) → (3, 10) → (3, 11) → (3, 12) → (2, 12) → (1, 13)	(5, 0) → (4, 1) → (5, 2) → (6, 3) → (6, 2) → (7, 1) → (6, 2) → (5, 3) → (4, 4) → (5, 5) → (6, 6) → (6, 7) → (6, 8) → (6, 9) → (6, 8) → (6, 7) → (6, 6) → (5, 5) → (4, 4) → (3, 5) → (2, 5) → (1, 5) → (1, 4) → (2, 3) → (2, 2) → (2, 1) → (2, 2) → (2, 3) → (1, 4) → (2, 5) → (2, 6) → (2, 7) → (2, 8) → (2, 9) → (2, 10) → (2, 11) → (3, 12) → (2, 12) → (1, 13)	(5, 7) → (4, 6) → (3, 5) → (2, 6) → (2, 7) → (1, 8) → (1, 9) → (1, 10) → (1, 11) → (1, 12) → (1, 13) → (2, 14) → (1, 14) → (1, 13) → (1, 12) → (1, 11) → (1, 10) → (1, 9) → (1, 8) → (0, 7)
Nodos expandidos	7340	12595	33114
Coste total	80	80	80
Tiempo tomado	32.9884 segundos	124.891 segundos	342.71 segundos
Longitud del camino	38	38	38

Para nuestra sorpresa, en la versión diagonal, la heurística que mejor se sigue adaptando al problema es la primera. Parece ser que la distancia de Manhattan (según nuestra codificación de las funciones) es la que mejor representa las normas del problema y hace que nuestro algoritmo genere muchos menos nodos y tarde mucho menos tiempo. Esperábamos un comportamiento contrario en la versión diagonal, pero no ha sido así. Queda demostrado que la diversidad de caminos que puede seguir AI para alcanzar una casilla calculada en línea recta, hace que esté más confuso.

La complejidad de los movimientos de las rocas, es decir, las posibilidades de movimiento que tienen cada una, complica enormemente el problema. También lo hace la amplitud que pueda tener el mapa. Pensamos que este último factor es el que más incrementa la dificultad de resolución por unidad utilizada.

Conclusiones

Esta práctica nos ha ayudado a comprender mejor la materia estudiada a lo largo del curso. La dificultad para idear modelos en la primera parte y la diversidad que tiene la técnica para plantearse ha hecho que nos demos cuenta de lo complejo que puede llegar a ser. Para la parte de búsqueda informada hemos aprendido que no siempre lo evidente es lo correcto y que las conclusiones humanas que podemos extraer para intentar comunicárselas a un ordenador, a su vez son el resultado de muchas consideraciones en conjunto que estimamos obvias por estar acostumbrados a ellas. La modelización de problemas y su planteamiento para que puedan ser computables es una tarea compleja que requiere de pensamiento analítico abstracto.