

Memoria

CSP y búsqueda heurística

Curso Heurística y Optimización
Grado en Ingeniería Informática
Curso 2021-22

<i>Carlos Chato Hidalgo</i>	<i>100429076</i>	<i>Grupo 82</i>	<i>100429076@alumnos.uc3m.es</i>
<i>Sergio Gil Nova</i>	<i>100429089</i>	<i>Grupo 82</i>	<i>100429089@alumnos.uc3m.es</i>

Índice

Índice	1
Introducción	2
Problema de satisfacción de restricciones	2
Descripción del modelo	2
Implementación del modelo	3
Análisis de los resultado obtenidos	4
Problema de búsqueda heurística	7
Descripción del modelo	7
Implementación del modelo	9
Análisis de los resultado obtenidos	11
Conclusión	13

Introducción

El presente documento pretende recoger nuestro desempeño en la resolución de la práctica sobre CSP y búsqueda heurística. A tal efecto, el documento se estructura en las siguientes secciones:

- Esta introducción
- Resolución del problema de satisfacción de restricciones
- Resolución del problema de búsqueda heurística
- Conclusión

Las dos secciones de resolución se estructuran, a su vez, en el planteamiento del problema y justificación de las decisiones tomadas en la implementación, y baterías de pruebas con numerosos casos diferentes.

Por último, en la conclusión extraemos los resultados más importantes de cada ejercicio y valoramos el grado de aptitud de nuestros algoritmos.

Problema de satisfacción de restricciones

Descripción del modelo

CSP = (X, D, C), donde

X Conjunto de variables
 $x_i \in N$ representan los contenedores

D = {d₁, d₂} Conjunto de celdas que pueden ocupar las variables
d₁ = { (m, n) }, posiciones que pueden ocupar los contenedores refrigerados
d₂ = { (m, n) }, posiciones que pueden ocupar los contenedores normales
m es la pila y n es la profundidad en ambos casos.

C Conjunto de restricciones

1. Dos contenedores no pueden ocupar la misma celda
 $R_{ij} = \{ (v_i, v_j) / v_i \in d_i, v_j \in d_j, v_i \neq v_j \}$
2. Un contenedor no puede flotar (o bien está en el fondo o bien tiene debajo otro contenedor)
 $R_{ij} = \{ (v_i, v_j) / v_i \in d_i \wedge v_j \in d_j \wedge v_i \neq v_j \wedge v_i[n] = \text{profundidad pila } i \vee (v_i[m] = v_j[m] \wedge v_j[n] - v_i[n] = 1) \}$
3. Los contenedores que van al puerto 2 deben estar en la base o tener debajo contenedores que van al puerto 2 (para evitar reordenar los contenedores).
 $R_{ij} = \{ (v_i, v_j) / v_i \in d_i \wedge v_j \in d_j \wedge v_i \neq v_j \wedge \text{puerto}_i \neq 2 \vee (v_i[n] = \text{profundidad pila } i \vee (v_i[m] = v_j[m] \wedge v_i[n] < v_j[n] \wedge \text{puerto}_j = 2)) \}$

Todas estas restricciones permiten al solucionador de *python-constraint* reducir al máximo el dominio de cada variable (las celdas que puede ocupar) por arco consistencia y camino consistencia.

Implementación del modelo

A continuación, pasamos a describir la implementación del problema, para la cual hemos utilizado Python y programación orientada a objetos.

En primer lugar, el programa debe invocarse de la siguiente forma:

```
python CSPStowage.py path mapa contenedores
```

Donde el path es el directorio donde están los archivos de entrada, y donde se guardan los de salida; mapa es el nombre del archivo con la configuración del barco; contenedores es el nombre del archivo con la lista de contenedores.

El problema que se resuelve es una instancia de la clase Problema, la cual aglutina todos los elementos de un CSP y posee un método que invoca al solver de *python-constraint*.

Las variables se definen como un vector de números naturales, que son los identificadores de los contenedores. Asimismo, los dominios se definen como un vector de pares (pila, profundidad) para los valores que puedan ocupar cada tipo de contenedor.

Por otro lado, las restricciones se definen como métodos de tipo booleano de la clase Problema. Cada método devuelve True o False en función de si los valores del dominio de cada variable cumplen o no con la restricción, lo cual permite al *solver* reducir el dominio de las variables cada vez que aplica arco consistencia y camino consistencia.

La primera restricción se puede indicar directamente con la función *AllDifferentConstraint()* del módulo *constraint*, para asegurarnos de que no se asignan dos contenedores a la misma celda.

La segunda restricción evita que se coloquen contenedores flotando en el aire.

Para cada par de contenedores distintos (B, A), se mira si B está debajo de A. Primero mira si B está en la base, lo cual valida la restricción inmediatamente. Si B no estuviera en la base, mira que estén alineados, y que la profundidad de A sea **un nivel** superior a la de B. Que estén alineados da sentido a esta restricción (si no están en la misma pila no hay nada que chequear); que A esté un nivel por encima de B expresa la esencia de esta restricción: un contenedor encima de otro. Otras formulaciones darían por bueno la existencia de huecos entre contenedores, lo cual es inaceptable.

La última restricción permite ordenar los contenedores desde el principio para evitar recolocaciones en el puerto 1. Una descripción en alto nivel es «si un contenedor A va al puerto 2 y debajo tiene un contenedor B que va al puerto 2, se cumple la restricción». Sin embargo, una descripción más pragmática en pseudocódigo sería la siguiente:

```
Para cada contenedor A con destino el puerto 2:
```

```
    Para cada contenedor B distinto de A:
```

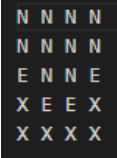
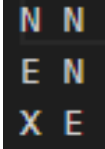
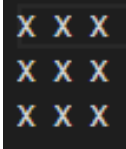
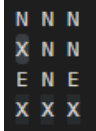
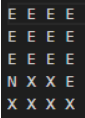
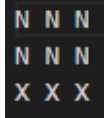
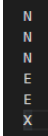
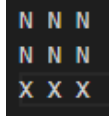
```
        Si A y B están en la misma columna Y A está encima de B Y B va al puerto 2:
```

A cumple la restricción
 Si A no ha cumplido la restricción, devuelve False
 La restricción se cumple, devuelve True

Análisis de los resultado obtenidos

Las pruebas se han realizado mezclando todos los mapas con todas las listas de contenedores, para tener una cobertura total de casos generales y especiales.

Para ello se ha hecho uso de ocho tipos diferentes de mapas y de tres listas de contenedores diferentes. Por cada mapa se han mirado los diferentes resultados, haciendo todas las combinaciones posibles con los diferentes contenedores de que se dispone.

 <p>Mapa 1</p>	 <p>Mapa 2</p>	 <p>Mapa 3</p>	 <p>Mapa 4</p>
 <p>Mapa 5</p>	 <p>Mapa 6</p>	 <p>Mapa 7</p>	 <p>Mapa 8</p>

 <p>Lista de contenedores 1</p>	 <p>Lista de contenedores 2</p>	 <p>Lista de contenedores 3</p>
--	--	--

- Mapa1-Contenedor1: (Comprobamos que el programa vaya bien en general)

```
Número de soluciones: 1368
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (2, 3), 5: (1, 3)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (2, 3), 5: (0, 2)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (1, 3), 5: (2, 3)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (1, 3), 5: (0, 2)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (0, 2), 5: (1, 3)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (0, 2), 5: (2, 3)}
{1: (3, 2), 2: (3, 1), 3: (3, 0), 4: (2, 3), 5: (0, 2)}
```

Obtenemos esa gran cantidad de soluciones debido a que tenemos un número de contenedores pequeño en comparación con todas las celdas disponibles. Por esta razón, el número de posibles combinaciones solución será alto, aun cumpliendo todas las condiciones expresadas anteriormente.

Podemos comprobar que las soluciones que se muestran en la captura son correctas, estableciéndose los contenedores que necesitan carga en las celdas de energía, además de que los contenedores que van al puerto 2 se establecen por debajo que los contenedores que van al puerto 1. Además, todas las soluciones son distintas, por lo que

podemos afirmar que no se producen colisiones en las soluciones dadas por el programa, cumpliendo con ello todas las restricciones.

- Mapa1-contenedor2: (Comprobamos cómo funciona con mapa grande y un número de contenedores menor)

```
Número de soluciones: 48
{1: (3, 2), 2: (2, 3), 3: (1, 3)}
{1: (3, 2), 2: (2, 3), 3: (0, 2)}
{1: (3, 2), 2: (1, 3), 3: (2, 3)}
{1: (3, 2), 2: (1, 3), 3: (0, 2)}
{1: (3, 2), 2: (0, 2), 3: (1, 3)}
{1: (3, 2), 2: (0, 2), 3: (2, 3)}
```

En esta situación el número de soluciones es menor debido a que tenemos menos contenedores para las posibles combinaciones de soluciones posibles. Podemos apreciar cómo el único contenedor refrigerado se establece siempre en una casilla de energía, mientras que el resto de contenedores se pueden establecer en el resto de casillas disponibles dando igual que sea de carga o no, cumpliendo el resto de restricciones.

- Mapa1-contenedor3: (Comprobamos cómo el número de soluciones disminuye con menos contenedores)

```
Número de soluciones: 20
{1: (3, 2), 2: (3, 1)}
{1: (3, 2), 2: (2, 3)}
{1: (3, 2), 2: (1, 3)}
{1: (3, 2), 2: (0, 2)}
```

En este caso, tan solo tenemos dos contenedores no refrigerados por lo que se podrán colocar en cualquier celda, siempre y cuando cumplan el resto de condiciones. Al ser tan solo dos contenedores, el número de posibles combinaciones es bastante reducido.

- mapa2-contenedor1:

```
Número de soluciones: 12
{1: (1, 1), 2: (1, 0), 3: (0, 0), 4: (1, 2), 5: (0, 1)}
{1: (1, 1), 2: (1, 0), 3: (0, 0), 4: (0, 1), 5: (1, 2)}
{1: (1, 1), 2: (0, 0), 3: (1, 0), 4: (0, 1), 5: (1, 2)}
{1: (1, 1), 2: (0, 0), 3: (1, 0), 4: (1, 2), 5: (0, 1)}
```

En este caso el mapa dos tiene tan solo cinco casillas para poder colocar los contenedores, cabe destacar que el contenedor1 (archivo) tan solo tiene cinco contenedores por lo que habrá soluciones pero muy pocas, debido a que las combinaciones posibles son muy escasas. Podemos comprobar como siempre los contenedores de refrigerados se colocan en las celdas de energía y el resto siguiendo las restricciones.

- mapa3-contenedor1

```
Número de soluciones: 0
```

En este caso, dado que todas las celdas están prohibidas, el dominio de que disponen las variables es el conjunto vacío. Evidentemente, en esta situación no hay soluciones al problema porque no hay ninguna instanciación posible.

- mapa4-contenedor1

```
Número de soluciones: 0
```

Esta prueba es interesante porque el mapa 4 tiene dos celdas electrificadas y contenedor1 tiene dos contenedores refrigerados. Además, el mapa tiene suficientes celdas en conjunto para los contenedores. Ante esta situación, cabe preguntarse por qué el programa no arroja ninguna solución. La razón es que existe una celda de tipo X sobre una de las celdas electrificadas. Esta situación hace que esa celda sea inalcanzable y, por tanto, que exista déficit de celdas E, haciendo irresoluble el problema.

- Mapa5-Contenedor1:

```
Número de soluciones: 1284
{1: (3, 3), 2: (3, 2), 3: (3, 1), 4: (2, 2), 5: (1, 2)}
{1: (3, 3), 2: (3, 2), 3: (3, 1), 4: (2, 2), 5: (2, 1)}
```

En este caso queremos comprobar que los contenedores puedan establecerse en cualquier celda incluidas las de energía, siempre y cuando los contenedores refrigerados hayan ya ocupado su posición en dichas celdas. Como podemos comprobar esto se cumple tras realizar esta prueba.

El resto de mapas han sido elaborados para ver los siguientes resultados del programa, y seguir comprobando que funcionan correctamente en las siguientes pruebas, siendo el código general para diversos casos a afrontar.

Lo interesante a destacar de los restantes mapas son las soluciones de los mapas 7 y 8. El siete es un mapa de una sola columna, por lo que todos los contenedores deberán estar unos encima de otros, pero siempre estando los del puerto dos abajo y los del puerto encima, por lo que es un buen ejemplo para comprobar esta restricción.

```
Número de soluciones: 1
{1: (0, 2), 2: (0, 3), 3: (0, 4)}
```

Como podemos ver en la captura, para el caso de contenedor2, tan solo hay una solución disponible, siendo esta, la colocación primero del contenedor del puerto dos en una celda electrificada (aunque no sea refrigerado), y luego el resto de contenedores, colocando el refrigerado en la siguiente celda electrificada, por lo que el testeo de las restricciones sigue siendo correcto.

Respecto al mapa número ocho, hay que comentar que al ser un mapa sin celdas electrificadas, las combinaciones con las listas de contenedores uno y dos no tienen solución, (algo lógico al no poder conectar los contenedores refrigerados a una celda con electricidad). Por lo tanto, con este mapa podemos concluir que las restricciones tomadas y elaboradas como se expresa anteriormente han resultado correctas para diversos escenarios de prueba.

Problema de búsqueda heurística

Descripción del modelo

Espacio de estados:

$$((P_i, L_i) (B)) \forall i \in \{1..n\}$$

Donde:

n representa el número de contenedores.

Los pares (P_i, L_i) representan cada contenedor (se identifica cada uno de ellos por su posición dentro de la tupla).

$P_i \in \{0, 1, 2, 3\}$ Es el lugar donde se encuentra el contenedor i -ésimo en el estado

actual, puede estar en puerto origen (0), puerto 1 (1), puerto 2 (2) o en el propio barco (3).

$L_i = (n_i, m_i)$ ó \emptyset_i Indica la posición del contenedor dentro del barco en caso de que esté dentro de él. En caso de que no esté en el barco, este parámetro será "vacío para el contenedor i -ésimo", lo cual representamos con el símbolo \emptyset_i .

$B: K \rightarrow V$ Es una función biyectiva que relaciona parámetros del barco con sus valores correspondientes, que van cambiando en cada estado:

$$K = \{ (pila, nivel)_i \forall i \in \{1..n\text{úm. celdas}\} \} \cup \{ \text{"puerto"} \}$$

Es el dominio de esta función, que comprende las celdas del barco en forma de pares ordenados (stack, depth) y el valor "**puerto**", que representa la posición del barco en cada estado.

$$V = \{ (T, C)_i \forall i \in \{1..n\text{úm. celdas}\} \} \cup \{ \text{puerto} \}$$

Es la imagen de esta función, que relaciona las celdas del barco con pares de valores: $T \in \{\text{«N»}, \text{«E»}\}$ es el tipo de la celda y $C \in \{T, \perp\}$ dice si está ocupada o no. Además, el valor **puerto** puede ser el puerto origen (0), puerto 1 (1) o puerto 2 (2).

Cabe destacar que el resto de información estática para resolver el problema está disponible en un vector auxiliar, guardando así en los estados la mínima información necesaria para que sean unos estados informados y eficientes. Dicho vector auxiliar, **vcont**, se define así:

$$((tipo, destino)_i \forall i \in \{1..n\text{úm. contenedores}\})$$

Donde cada tupla representa la información estática del contenedor correspondiente, que se compone del tipo del contenedor y su puerto de destino.

$$tipo \in \{\text{«R»}, \text{«S»}\}$$

$$destino \in \{1, 2\}$$

Estado inicial:

$((0, \emptyset)_i (B)) \forall i \in \{1..n\}$

B: $K \rightarrow V$

$K = \{ (pila, nivel)_i \mid \forall i \in \{1..n\} \} \cup \{ \text{"puerto"} \}$

$V = \{ (T, \perp)_i \mid \forall i \in \{1..n\} \} \cup \{0\}$, T puede ser N o E en función del mapa de entrada

Estado final:

Es implícito, se alcanza cuando todos los contenedores están en su puerto correspondiente. Para que el algoritmo sepa cuándo ha encontrado un estado final, debe comprobar que se cumple esta condición:

$$P_i = vcont[i][destino] \quad \forall i \in \{1..n\}$$

Operadores:

Operador	Explicación	Precondiciones	Resultado
cargar(X, C)	Carga contenedor X en celda C	$P_X = P_B,$ $L_X = \emptyset_X,$ $(vcont[X][tipo] \neq R \vee B[X][0] = E)$	$P_X = 3,$ $L_X = C$
descargar(X, S)	Descarga contenedor X en puerto S	$P_B = S,$ $P_X = 3,$ $L_{encima(X)} = \emptyset$	$P_X = S,$ $L_X = \emptyset$
navegar()	Mueve el barco al siguiente puerto ¹	$P_B < 2$	$P_B = P_B + 1$

Notas:

- El puerto en el que está el barco, B["puerto"], se sustituye por P_B por legibilidad.
- La tercera precondición para cargar se cumple cuando el contenedor X es normal o, si es refrigerado, si la celda C es electrificada.
- La tercera precondición para descargar establece que encima de X no debe haber otro contenedor. En caso de que X esté en la superficie, esta condición se satisface automáticamente.

Heurísticas: (aquí solo las definimos, más adelante explicamos cada una)

Heurística 1: Cuenta los contenedores mal colocados, es decir, aquellos que no están en su destino.

Conjunto de contenedores descolocados: $M = \{ (P_i, L_i) \mid \forall i \in \{1..n\} / vcont[i][destino] \neq P_i \}$
 $h_1(N) = |M|$

Heurística 2: Cuenta los contenedores que el barco se deja por el camino.

Contenedores dejados atrás: $M = \{ (P_i, L_i) \mid \forall i \in \{1..n\} / P_i < vcont[i][destino], P_i \neq P_B \}$

¹ El barco no puede retroceder según esta parte del enunciado: "Además, hay que considerar los viajes del barco, que puede navegar del puerto inicial al puerto 1, y del puerto 1 al puerto 2". Por tanto, consideramos que solo puede avanzar.

$$h_2(N) = |M|$$

Implementación del modelo

A continuación se procede a describir la implementación del algoritmo, la cual se ha realizado en python. Este es un lenguaje orientado a objetos, por lo que nuestra resolución está basada en este paradigma.

En primer lugar, el programa debe invocarse de la siguiente forma:

```
python ASTARStowage.py path mapa contenedores heurística
```

Donde el path es el directorio donde están los archivos de entrada, y donde se guardan los de salida; mapa es el nombre del archivo con la configuración del barco; contenedores es el nombre del archivo con la lista de contenedores; y heurística es el nombre de la heurística que se quiere utilizar (puede tomar los valores 1 o 2).

El módulo fuerte del programa es la clase Problema, la cual se encarga de realizar la resolución del problema de búsqueda.

Dentro de esta clase, nos encontramos todos los métodos necesarios para la resolución. La clase tiene como parámetros la información de los contenedores, la heurística a utilizar y las listas abierta y cerrada.

El método más importante de clase es el de A*, el cual es el encargado de ejecutar todo el algoritmo para dar una respuesta al problema planteado. La resolución se realiza junto con la ayuda de otros métodos que describiremos a continuación.²

La programación de este método es bastante básica. Lo que realizamos es ejecutar el bucle principal hasta que la lista de abiertos (nodos por abrir) se vacíe o hasta que el problema encuentre la solución. Dentro del bucle se realizan las siguientes operaciones:

- *es_meta()*: Este método se encarga de realizar la función de evaluación para saber si el estado que actual es meta o no. Esto es necesario debido a que el estado final es implícito, por lo que necesitamos de una función de evaluación para ello. Además, realizamos este chequeo antes de expandir los nodos, por lo que al sacarlo de la lista de nodos lo examinamos, y en caso de ser el final, devolvemos la solución al problema. En caso contrario, se prosigue con el algoritmo generando los hijos de dicho nodo.
- *get_children()* : Este método se encarga de aplicar todos los operadores definidos previamente al estado actual. Con esto conseguimos que nos retorne una lista con todos los posibles hijos de ese estado. Además, cabe destacar que cada operador será el encargado de establecer los nuevos costes a cada nodo hijo generado para hacer posteriormente una recolocación de todos los nodos en la lista de abiertos. Destacar que antes de aplicar los operadores, se aplican unas precondiciones para que los nodos generados sean físicamente posibles, ya que los nodos que no tienen una lógica física se ignoran.

² Destacamos que este análisis es muy básico, donde no describimos muchas parte del código, la idea de esta sección es que el lector tenga una ligera idea sobre lo que hace el algoritmo, en caso de que quiera profundizar en sobre el funcionamiento del mismo, puede leer todos los comentarios del código donde está todo explicado paso a paso y con detalle, sobre la función de cada elemento.

- *comprobación de duplicados*: Se comprueba, para cada hijo, que no esté en la lista cerrada, para evitar expandir nodos ya explorados. Además, se mira si ese hijo está en la lista abierta. En caso de que esté, nos quedamos con el que tenga menor coste acumulado y descartamos al otro. De esta forma evitamos expandir un nodo para el cual tenemos un camino mejor.

En caso de que la lista abierta se vacíe, esto nos indica que el problema planteado no tiene solución, ya que se han abierto todos los nodos posibles. Por tanto, se avisa al usuario de que su problema no tiene solución (como A* es completo, es seguro que si existiera una solución la habríamos encontrado).

Ahora hablemos un poco sobre el método *get_children()*. Este método está basado en los operadores que hemos comentado anteriormente en la formalización del modelo. Pero queríamos destacar las siguientes funciones que se han usado para evaluar las precondiciones:

- *celdas_posibles()*: Este método se encarga de pasar una lista con las celdas posibles que puede ocupar ese contenedor dentro del barco, con esto el método de cargar, sabe dónde colocar cada contenedor dentro del barco. Para generar esta lista se respetan las restricciones de la gravedad y de que los contenedores refrigerados deben ir en celdas electrificadas.
- *no_hay_nadie_encima()* : La función de este método está asociada con la descarga. Se encarga de informar a la precondición de si hay un contenedor encima del que queremos quitar. En caso de que haya uno encima, no podemos quitar este contenedor, porque primero tendremos que quitar el que está encima.

Pasamos a describir brevemente los operadores utilizados, estos son los encargados de generar los nuevos nodos hijos:

- *cargar()*: Este método realiza la carga de un contenedor en una celda del barco. La creación del nodo se realiza creando una copia en profundidad del nodo “padre”, añadiendo las nuevas configuraciones, es decir estar en el barco y los costes asociados tras realizar la operación.
- *descargar()*: Este método lo que hace es retirar un contenedor del barco, cambia los parámetros precisos, y añade los costes asociados. Este operador tan solo genera un hijo, debido a que solo descarga un contenedor.
- *navegar()*: Este método lo que hace es mover el barco de un puerto a otro, cambiando tan solo la tupla referida al barco, por lo que tan solo generará un nodo.³

Ahora pasamos a comentar las heurísticas implementadas para resolver el problema:

- **Heurística 1**: Para el cálculo de este valor contamos todos los contenedores que no están en su puerto de destino. Esta heurística hace un símil con una de las heurísticas más famosas del 8-puzzle. El símil se haría con las fichas que no se encuentran en su lugar, por lo que se puede ver nuestro problema actual como un problema de colocación, donde nos interesa saber cuántos contenedores no están bien colocados. Esta relajación de restricciones y el hecho de que en la meta la

³ Destacamos que como expresa el enunciado el barco tan solo podrá navegar del puerto inicial, al puerto uno y del uno al puerto dos, sin tener la posibilidad de ir marcha atrás, por lo que tan solo podrá hacer ese tipo de recorrido.

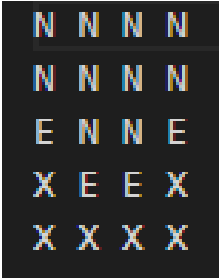
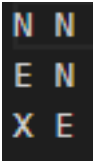
heurística vale 0 hace que la heurística 1 sea admisible, pues nunca va a sobreestimar el coste real hasta la meta.

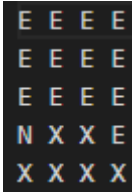
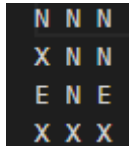
- **Heurística 2:** Esta heurística aplica una penalización de 50 unidades por cada contenedor mal colocado que se haya dejado el barco. El objetivo es que el algoritmo premie aquellos estados en los que el barco no deja ningún contenedor en un puerto incorrecto. En caso de que algún contenedor se quede atrás, el coste hasta la solución es infinito, porque retroceder no está contemplado en los operadores. Por tanto, esta heurística es admisible porque nunca sobreestima el coste que tomaría llegar a la solución.

Análisis de los resultado obtenidos

El problema propuesto tiene complejidad factorial en el número de contenedores, por lo que los tiempos de ejecución escalan rápidamente a medida que consideramos listas más grandes. Por tanto, hemos probado las mismas configuraciones de bahías que con el CSP, pero con listas de tan solo 2 y 4 contenedores. Cabe destacar que esta explosión factorial se debe a que las precondiciones de los operadores son las mínimas necesarias para asegurarnos de que no se poda ningún estado físicamente posible. Por tanto, se generan muchos estados que no alcanzan ninguna solución, los cuales se pueden ignorar utilizando nuestras heurísticas, como veremos a continuación.

Las pruebas con 2 contenedores arrojan un resultado satisfactorio en menos de 1 segundo, dando la secuencia mínima para resolver el problema. Esta secuencia la comprobamos fácilmente debido al pequeño número de contenedores. Por otro lado, las pruebas con 4 contenedores son bastante más lentas, del orden de minutos. Sin embargo, se sigue encontrando la solución cuando existe.

Mapa	contenedores1	contenedores 2	contenedores 3
	h1: Tiempo total: 1.8729116916656494 Coste total: 7130 Longitud del plan: 11 Nodos expandidos: 1142 h2: Tiempo total: 0.4630272388458252 Coste total: 7130 Longitud del plan: 11 Nodos expandidos: 401	h1: Tiempo total: 7.433981418609619 Coste total: 7130 Longitud del plan: 11 Nodos expandidos: 2054 h2: Tiempo total: 2.203679084777832 Coste total: 7143 Longitud del plan: 13 Nodos expandidos: 864	h1: Tiempo total: 0.005999326705932617 Coste total: 3559 Longitud del plan: 6 Nodos expandidos: 48 h2: Tiempo total: 0.004000186920166016 Coste total: 3559 Longitud del plan: 6 Nodos expandidos: 33
	h1: Tiempo total: 0.021999359130859375 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf	h1: Tiempo total: 0.5199990272521973 Coste total: 7137 Longitud del plan: 13 Nodos expandidos: 699	h1: Tiempo total: 0.0009996891021728516 Coste total: 3553 Longitud del plan: 6 Nodos expandidos: 22

	h2: Tiempo total: 0.02399897575378418 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf	h2: Tiempo total: 0.2969973087310791 Coste total: 7165 Longitud del plan: 15 Nodos expandidos: 477	h2: Tiempo total: 0.0010001659393310547 Coste total: 3553 Longitud del plan: 6 Nodos expandidos: 18
X X X X X X X X X	h1: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf h2: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf	h1: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf h2: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf	h1: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf h2: Tiempo total: 0.0 Coste total: inf Longitud del plan: 0 Nodos expandidos: inf
	h1: Tiempo total: 2.1140713691711426 Coste total: 7115 Longitud del plan: 11 Nodos expandidos: 1211 h2: Tiempo total: 0.5979993343353271 Coste total: 7115 Longitud del plan: 11 Nodos expandidos: 459	h1: Tiempo total: 7.470236778259277 Coste total: 7130 Longitud del plan: 11 Nodos expandidos: 2054 h2: Tiempo total: 2.166133403778076 Coste total: 7143 Longitud del plan: 13 Nodos expandidos: 864	h1: Tiempo total: 0.007000446319580078 Coste total: 3559 Longitud del plan: 6 Nodos expandidos: 48 h2: Tiempo total: 0.004000186920166016 Coste total: 3559 Longitud del plan: 6 Nodos expandidos: 33
	h1: Tiempo total: 0.022998809814453125 Coste total: inf Longitud del plan: 2 Nodos expandidos: inf h2: Tiempo total: 0.02399897575378418 Coste total: inf Longitud del plan: 2 Nodos expandidos: inf	h1: Tiempo total: 4.205048322677612 Coste total: 7115 Longitud del plan: 11 Nodos expandidos: 1615 h2: Tiempo total: 1.1560335159301758 Coste total: 7134 Longitud del plan: 13 Nodos expandidos: 654	h1: Tiempo total: 0.005994081497192383 Coste total: 3556 Longitud del plan: 6 Nodos expandidos: 44 h2: Tiempo total: 0.002999544143676758 Coste total: 3556 Longitud del plan: 6 Nodos expandidos: 31

Las pruebas han sido seleccionadas y elaboradas con ejemplos muy parecidos a los realizados en las pruebas de CSP. Además cabe destacar que , todas estas pruebas se han realizado con la intención de comprobar cómo se comporta el algoritmo ante diversas situaciones, tanto normales como anormales, dejando para ellos mapas en los que el número de soluciones es muy elevado, como mapas más reducidos, donde encontrar la solución no es tan trivial a simple vista, y el número de soluciones es mucho menor.

Una vez realizadas las pruebas, podemos pasar al análisis comparativo de las dos heurísticas.

La heurística 1 cuenta el número de contenedores que todavía no están en su destino. Permite estimar el coste de la solución relajando la restricción de que el barco debe estar en un puerto determinado para descargar. A pesar de su diseño, nos hemos dado cuenta de que tiene muchas llanuras. Por ejemplo, estando en el puerto 0, todas las configuraciones posibles tienen el mismo valor para la heurística ya que independientemente de la configuración del barco, todos los contenedores están “mal colocados” todavía. Esta característica hace que nuestra heurística 1 sea poco informada y por tanto que arroje peores resultados.

La heurística 2 cuenta el número de contenedores que el barco se deja atrás. Esto se nos ocurrió porque observamos que en muchos estados el barco avanzaba pero quedaban contenedores tirados atrás. El efecto es que, al ir el barco en vacío o con pocos contenedores, esos estados se posicionan los primeros en la lista abierta, malgastando tiempo y recursos que podrían aprovechar los estados bien encaminados. Por tanto, nuestra heurística indica al algoritmo que los estados “malos” tienen un coste muy grande (infinito en teoría) hasta la solución, lo que hace que se coloquen al final de la lista beneficiando así a los estados más prometedores.

La heurística 2 presenta un grado de mejora sustancial, ya que en la tabla podemos comprobar que esta heurística mejora el tiempo medio de ejecución en un 295% (respecto a la heurística 1). En cuanto a nodos expandidos, la heurística 2 ofrece una mejora del 132% respecto a la heurística 1.

Conclusión

Esta práctica nos ha permitido comparar dos formas diferentes de buscar la solución a un problema logístico del mundo real. Por un lado, el CSP utiliza fuerza bruta para encontrar todas las posibles configuraciones de contenedores dentro del barco, por lo que no es especialmente eficiente. Por otro lado, el algoritmo A* realiza búsqueda informada, ya que se puede guiar por los costes de sus acciones y puede estimar qué estado está más cerca de la solución, para encontrar un resultado óptimo.

La resolución de este problema como un CSP nos ha aportado conocimientos sobre una nueva forma de resolver problemas, que es la satisfacción de restricciones. A pesar del tiempo que tarda este algoritmo (por utilizar fuerza bruta), entendemos que puede ser muy conveniente en problemas con muchas restricciones y dominios pequeños, ya que la solución se puede encontrar en muy poco tiempo al tener que hacer pocas iteraciones.

Además, la utilización de A* nos ha permitido asentar nuestros conocimientos sobre este algoritmo, ya que cuando lo conocimos en Inteligencia Artificial nos parecía demasiado complejo como para entenderlo. El hecho de utilizar una lista de nodos expandidos, otra de nodos pendientes de expandir y un método que genera hijos en base a precondiciones ahora parece tan sencillo que nos sentimos capaces de utilizar A* en muchos problemas cotidianos. El único factor que debemos mejorar es nuestro enfoque para plantear heurísticas, lo cual solo depende de la práctica y el aprendizaje continuo en esta materia.

Por último cabe destacar que nuestros conocimientos en algoritmos de búsqueda se han incrementado notablemente, debido a que esta práctica, te obliga a tener un amplio enfoque sobre todos los algoritmos posibles que hay, incluso más allá del A* que es el que nos ha tocado programar. Esto lo expresamos, debido a que durante el proceso del desarrollo del algoritmo, llegamos a pensar sobre el uso de algoritmos de búsqueda avariciosos, en algún momento determinado si se encontraba cerca de la meta, para optimizar la posible solución.

Consiguiendo con ello examinar de forma rápida una rama del algoritmo que parece exitosa, y en caso de que no encontrará solución hacer backtracking y seguir con la búsqueda.

En conclusión, esta práctica nos ha abierto el enfoque a nuevas formas de resolución de problemas, tanto de satisfacción de restricciones como de búsqueda. Ahora nos vemos capaces de resolver problemas que traten de estas materias.