Práctica 2- Satisfacción de Restricciones y Búsqueda Heurística

Heurística y optimización- Curso 25/26

Nombre, apellidos y NIA	
Alumn@ 1:	Ana Grima Vázquez de Prada (100495785) 100495785@alumnos.uc3m.es Grupo 80 - Grado en Ingeniería Informática
Alumn@ 2:	Alejandra de los Santos Blanco (100495843) 100495843@alumnos.uc3m.es Grupo 80- Grado en Ingeniería Informática

Índice

Introducción..... 3
Parte 1: Satisfacción de restricciones (BINAIRO).....3
1.1 Modelización del problema.....3
1.1.1 Variables.....3
1.1.2 Función objetivo..... 3
1.1.3 Dominio..... 4
1.1.4 Restricciones.....4
1.2 Implementación del modelo..... 5
Parte 2: Algoritmos de búsqueda para caminos mínimos (DIMACS)..... 5
2.1 Modelización..... 6
2.1.1 Representación del grafo..... 6
2.1.2 Parámetros del modelo..... 6
2.1.3 Estado del problema..... 6
2.1.4 Sucesores..... 7
2.1.5 Función de coste acumulado..... 7
2.1.6 Objetivo..... 7
2.1.7 Función de evaluación para búsquedas informadas.....7
2.1.8 Criterio de terminación..... 7
2.2 Heurísticas..... 7
2.2.1 Heurística 0..... 7
2.2.2 Heurística 1..... 8
2.2.3 Heurística 2..... 8
2.3 Implementación del algoritmo..... 9
Parte 3: Análisis de resultados.....10
3.1 Análisis de resultados..... 10
3.1.1 Análisis de resultados y complejidad del problema de la Parte 1..... 10
3.1.2 Análisis de resultados y complejidad del problema de la Parte 2..... 11
3.2 Casos de pruebas..... 11
3.2.1 Pruebas para la Parte 1.....11
3.2.2 Pruebas para la Parte 2..... 12
Conclusiones de la práctica.....14

Introducción

El objetivo de esta práctica es aplicar técnicas de satisfacción de restricciones y búsqueda heurística a la resolución de problemas combinatorios. Por un lado, se aborda la modelización y resolución del pasatiempo BINAIRO mediante un enfoque CSP, definiendo formalmente variables, dominios y restricciones. Por otro lado, se estudia el problema del camino mínimo en grafos de gran escala del 9th DIMACS Shortest-Path Challenge, implementando y comparando algoritmos de búsqueda óptima informados y no informados. Finalmente, se analizan los resultados obtenidos, evaluando la eficiencia y corrección de las soluciones en función de las decisiones de modelado y del algoritmo empleado.

Parte 1: Satisfacción de restricciones (BINAIRO)

En esta primera parte se resuelve el pasatiempo BINAIRO mediante un enfoque de satisfacción de restricciones, construyendo un modelo formal que define de forma precisa las condiciones del tablero. Esta separación entre modelización y resolución permite al solver explorar únicamente configuraciones válidas y facilita la verificación de las soluciones obtenidas para distintas instancias.

1.1 Modelización del problema

El problema BINAIRO se formula como un problema de satisfacción de restricciones en el que cada casilla de un tablero $n \times n$ se modela como una variable sujeta a un conjunto de reglas lógicas. El objetivo de la modelización es definir formalmente las variables, sus dominios y las restricciones necesarias para garantizar la obtención de una solución válida.

1.1.1 Variables

Se define una variable por cada casilla del tablero:

$$x_{i,j} \; \forall (i,j) \in \{1,\ldots,n\} \times \{1,\ldots,n\}$$

Cada variable representa el símbolo contenido en la casilla de la fila i y columna j.
Los valores posibles son:

- X: disco negro
- O: disco blanco

Así, el dominio general del problema es el conjunto $\{X, O\}$
El dominio concreto de cada variable depende del valor inicial especificado en la entrada:

- Si la casilla es ".", entonces la variable puede tomar valores en el conjunto $\{X, O\}$
- Si la casilla está fija en la entrada, entonces el dominio queda restringido a dicho valor.

Esta decisión permite incorporar las casillas fijas directamente mediante la definición del dominio, sin necesidad de restricciones adicionales.
Modelar una variable por casilla permite capturar la estructura natural del tablero y evita restricciones globales innecesarias. Al ser un CSP puro, no es necesario introducir variables adicionales o estados auxiliares.

1.1.2 Función objetivo

El problema no requiere optimización, por lo que no se define una función objetivo: no se minimiza ni maximiza ninguna magnitud; solo se busca una solución factible.
Esta elección se ajusta a la naturaleza del problema BINAIRO, cuyo único propósito es encontrar una configuración válida.

1.1.3 Dominio

El dominio general del problema es: $D = \{X, O\}$
Por tanto: $x_{i,j} \in D;\ \forall i, j$

Para aquellas variables correspondientes a casillas fijadas en la entrada, el dominio queda reducido a un único valor, mientras que las casillas inicialmente vacías mantienen el dominio completo.
En aquellas casillas donde la entrada impone un valor fijo:

$$x_{i,j} = X \text{ o } x_{i,j} = O$$

Y únicamente las casillas marcadas con "." poseen un dominio libre:

$$x_{i,j} \in \{X, O\}\ si\ dato_{i,j} = .$$

El uso de un dominio binario maximiza la eficiencia del solver y simplifica la formulación de restricciones.

1.1.4 Restricciones

Las casillas pre-rellenadas no se contabilizan como restricciones, ya que se imponen directamente mediante la definición del dominio de las variables.

El juego BINAIRO impone cuatro grupos de restricciones fundamentales:

1. Igual número de símbolos en cada fila

Cada fila debe contener exactamente $\frac{n}{2}$ símbolos X y $\frac{n}{2}$ símbolos O.
Esta restricción se modela comprobando que el número de símbolos X en cada fila es igual a $\frac{n}{2}$. Dado que todas las variables solo pueden tomar valores en el conjunto $\{X, O\}$, esta condición implica que el número de símbolos O en la fila también es $\frac{n}{2}$.

$$\sum_{j=1}^{n} [x_{i,j} = X] = \frac{n}{2}\ \forall i$$

Esta restricción asegura que cada fila contiene la misma cantidad de discos blancos y negros.
Para que esta restricción (y la siguiente) sea satisfacible, el tamaño del tablero n debe ser un número par. En caso contrario, no sería posible distribuir exactamente $\frac{n}{2}$ símbolos X y $\frac{n}{2}$ símbolos O en una fila, o columna.

2. Igual número de símbolos en cada columna

De forma análoga, cada columna debe contener la misma cantidad de discos blancos y negros.
Esta restricción se modela comprobando que el número de símbolos X en cada columna es igual a $\frac{n}{2}$. Dado que todas las variables solo pueden tomar valores en el conjunto $\{X, O\}$, esta condición implica que el número de símbolos O en la columna también es $\frac{n}{2}$.

$$\sum_{i=1}^{n} [x_{i,j} = X] = \frac{n}{2}\ \forall j$$

Se garantiza el equilibrio vertical del tablero.

3. No tres iguales consecutivos en filas

En ninguna fila puede aparecer una secuencia de tres símbolos idénticos consecutivos.

$$\neg(x_{i,j} = x_{i,j+1} = x_{i,j+2})\ \forall i,\ j \in \{1,\dots, n-2\}$$

4. No tres iguales consecutivos en columnas

En ninguna columna puede aparecer una secuencia de tres símbolos idénticos consecutivos.

$$\neg(x_{i,j} = x_{i+1,j} = x_{i+2,j})\ \forall j,\ i \in \{1,\dots, n-2\}$$

1.2 Implementación del modelo

La implementación del modelo de satisfacción de restricciones se ha realizado utilizando la biblioteca python-constraint, que permite definir problemas CSP de forma declarativa a partir de variables, dominios y restricciones.

En primer lugar, el programa lee el fichero de entrada y valida que la instancia sea correcta: el tablero debe ser cuadrado, no vacío y de tamaño par, ya que las restricciones de equilibrio entre símbolos requieren que el número de columnas y filas sea divisible entre dos. Asimismo, se comprueba que los símbolos de entrada sean válidos (X, O o ".").
A continuación, se construye el problema de satisfacción de restricciones definiendo una variable por cada casilla del tablero. El dominio de cada variable se establece en función del valor inicial de la entrada: las carillas pre-rellenadas quedan restringidas a un único valor, mientras que las casillas vacías pueden tomar cualquiera de los dos símbolos permitidos. De este modo, los valores iniciales se incorporan directamente al modelo sin necesidad de restricciones adicionales.

Una vez definidas las variables y sus dominios, se añaden las restricciones del problema. Para cada fila y cada columna se impone una restricción que garantiza que el número de símbolos X es exactamente $\frac{n}{2}$; dado el dominio binario de las variables, esta condición asegura automáticamente el mismo número de símbolos O. Además, se añaden restricciones que impiden la aparición de tres símbolos idénticos consecutivos tanto en filas como en columnas. Estas restricciones se aplican a nivel de fila o columna completa y verifican internamente todas las subsecuencias relevantes.

Tras definir el modelo, el solver explora el espacio de soluciones y genera todas las asignaciones que satisfacen simultáneamente las restricciones. El programa muestra por pantalla el tablero inicial, el número total de soluciones encontradas y una de las soluciones válidas. Finalmente, se escriben en el fichero de salida tanto la instancia original como todas las soluciones obtenidas, utilizando un formato tabular uniforme que facilita su interpretación.

Esta implementación se corresponde con el modelo descrito en el apartado anterior y separa claramente la definición del problema de su resolución, aprovechando las capacidades del solver para gestionar de forma eficiente la búsqueda de soluciones factibles.

Parte 2: Algoritmos de búsqueda para caminos mínimos (DIMACS)

La segunda parte de la práctica aborda el problema del camino mínimo en grafos de gran escala del 9th DIMACS Shortest-Path Challenge, que requieren algoritmos de búsqueda eficientes debido a su tamaño y complejidad. En esta sección se define el marco formal del problema y los elementos clave del proceso de búsqueda, incluyendo la estructura del grafo y el funcionamiento de los algoritmos utilizados.

2.1 Modelización

Se ha diseñado un modelo algorítmico que permite calcular un camino óptimo entre dos vértices utilizando búsquedas informadas (A*) y no informadas (Dijkstra), garantizando siempre la optimalidad de la solución.
A continuación se describe la modelización formal del problema: estructuras principales, parámetros, representación del grafo y del estado, y reglas de transición.

2.1.1 Representación del grafo

El problema está definido sobre un grafo dirigido:

$$G = (V, A)$$

Donde:

- V es el conjunto de vértices, con $|V| \approx 2.5\text{x}10^5 - 3.2\text{x}10^5$ según el mapa utilizado.
- $A \subseteq V\text{x}V$ es el conjunto de arcos dirigidos, cada uno con un coste asociado en metros.

Cada arco se presenta como, con coste estrictamente positivo:

$$a = (u, v, c(u, v))$$

Donde:

- u es el vértice origen.
- v es el vértice destino.
- $c(u, v)$ es el coste, distancia real en metros.

Los ficheros DIMACS proporcionan:

- En el fichero .gr: $a\ u\ v\ c(u, v))$
- En el fichero .co: $v\ id\ lon\ lat$ donde lat y lon vienen escaladas $\text{x}10^6$. Las coordenadas se almacenan en diccionarios independientes y se emplean exclusivamente para el cálculos de heurísticas.

Este grafo se almacena internamente mediante una lista de adyacencia, donde cada vértice u contiene una lista de pares $(v, c(u, v))$. Esta estructura es óptima en memoria para grafos dispersos, permite iterar sucesores en tiempo proporcional al grado del vértice, y evita almacenamiento redundante.

2.1.2 Parámetros del modelo

Vértice origen: $s \in V$
Vértice destino: $t \in V$
Coste de los arcos:

- Para cada arco $(u, v) \in A$: $c(u, v) > 0$

Coordenadas geográficas:

- Para cada vértice v: $(lat(v), lon(v))$

2.1.3 Estado del problema

Un estado queda definido únicamente por el vértice en el que se encuentra el algoritmo:

$$estado = v \in V$$

Para completar la modelización necesaria en búsquedas óptimas, se asocia a cada estado un coste acumulado:

$$g(v) = coste\ del\ camino\ desde\ s\ hasta\ v$$

Este valor no se incluye explícitamente en la definición del estado, ya que el grafo es único, estático y muy grande; incorporarlo dentro del estado implicaría duplicar información y aumentar innecesariamente el consumo de memoria. El grafo se mantiene como estructura global y solo el vértice actual forma parte del estado.

2.1.4 Sucesores

Los sucesores de un vértice u se definen como:

$$Succ(u) = \{v \mid (u,\ v) \in A\}$$

Cada transición tiene un coste:

$$coste\ (u \to v) = c(u,\ v)$$

La actualización del coste acumulado es:

$$g(v) = g(u) + c(u, v)$$

2.1.5 Función de coste acumulado

El coste acumulado $g(v)$ representa el coste total del camino más barato conocido desde el vértice origen u hasta un vértice v. El coste de un camino $u \rightsquigarrow v$ es:

$$g(v) = \sum_{(u,v) \in camino} c(u,\ v)$$

Este valor se actualiza durante la exploración del grafo.

2.1.6 Objetivo

El objetivo es encontrar un camino desde s hasta t tal que el coste total sea mínimo:

$$min\ g(t)$$

2.1.7 Función de evaluación para búsquedas informadas

Cuando se usa A*, cada nodo se evalúa mediante:

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

donde $h(v)$ es una heurística admisible, distancia estimada desde v hasta t.
Este aspecto se desarrollará más adelante en la sección 2.2 Heurísticas.

2.1.8 Criterio de terminación

El algoritmo termina cuando $v = t$ y se ha encontrado el camino con el menor valor de $g(t)$.

2.2 Heurísticas

Para resolver el problema del camino mínimo de forma eficiente, se implementan varias heurísticas con el objetivo de comparar su comportamiento y evaluar la reducción en el número de nodos expandidos respecto al algoritmo sin heurística (Dijkstra).
Todas las heurísticas descritas cumplen la condición de admisibilidad:

$$h(v) \leq distancia\ real(v,\ t)$$

Que garantiza que A* encuentra siempre el camino óptimo.
A continuación se describen las tres heurísticas utilizadas.

2.2.1 Heurística 0

$h_0(v) = 0 \rightarrow$ Sin heurística

Esta heurística asigna un valor nulo a todos los nodos:

$$h_0 = 0\ \forall v \in V$$

Interpretación:
Es equivalente al algoritmo Dijkstra, ya que la función de evaluación se convierte en:

$$f(v) = g(v) + 0 = g(v)$$

Características:

- Admisible.
- No informa en absoluto sobre la distancia restante.

- Máximo número de expansiones de entre todas las heurísticas probadas.
- Se utiliza como baseline (peor caso) para comparar las mejoras obtenidas con heurísticas más avanzadas.

2.2.2 Heurística 1

Distancia euclídea en el plano (proyección)

Se aproxima la distancia entre dos vértices considerando sus coordenadas geográficas como puntos en un plano cartesiano, ignorando curvatura y escalas.

Si cada vértice v tiene coordenadas: $(lat(v),\ lon(v))$

Entonces la distancia euclídea aproximada es:

$$h_1(v) = \sqrt{(lat(v) - lat(t))^2 + (lon(v) - lon(t))^2}$$

Dado que los valores viene multiplicados por 10^6, el resultado es proporcional pero admisible.

Interpretación:

- Proporciona una estimación de la cercanía geométrica entre dos puntos.
- Aunque no refleja la distancia real por carretera, siempre está por debajo de la distancia real, por lo que es admisible.
- Reduce significativamente el número de expansiones frente a Dijkstra.

Ventajas:

- Muy sencilla de calcular.
- Muy baja sobrecarga computacional.
- Gran mejora en metas lejanas en mapas densos.

2.2.3 Heurística 2

Distancia euclídea normalizada por factor métrico

Las coordenadas del fichero .co están en unidades “pseudo-cartesianas” (latitud y longitud $\mathrm{x}10^6$), y no son distancias reales.

Para obtener una heurística más informada, se normaliza la distancia euclídea mediante un factor que aproxima la conversión entre unidades del fichero y metros reales.

Definimos:

$$h_2 = k\,\mathrm{x}\sqrt{(lat(v) - lat(t))^2 + (lon(v) - (lon(t))^2}$$

Interpretación:

Esta heurística amplifica la escala respecto a h_1, pero manteniendo la condición:

$$h_2(v) \leq distancia\ real(v,\ t)$$

Lo que garantiza si admisibilidad mientras el factor k no exceda una relación razonable.

Ventajas:

- Más informativa que h_1.
- Reduce aún más las expansiones, especialmente en mapas grandes como BAY o NY.
- Sigue siendo computacionalmente muy simple.

Riesgo controlado:

Usar un k demasiado alto podría violar la admisibilidad, pero un valor pequeño mantiene la optimalidad. Para esta práctica se ha optado por elegir un $k = 1.2$, un valor significativamente inferior a la conversión real entre microgrados y metros, con el objetivo de mantener la admisibilidad de la heurística.

2.2.4 Comparación formal de heurísticas

Heurística	Admisibilidad	Consistencia	Expansiones	Comentario
h_0	Trivial	Consistente	Máximas	Equivalente a Dijkstra
h_1	Sí	Sí	Menores que h_0	Usa geometría real aproximada
h_2	Sí si $k \leq k_{max}$	Usualmente sí	Mínimas	Heurística más informativa

2.3 Implementación del algoritmo

La resolución del problema del camino mínimo se ha implementado mediante una versión general del algoritmo A*, que permite reproducir tanto una búsqueda no informada (Dijkstra) como búsquedas informadas, en función de la heurística utilizada. Esta decisión evita duplicación de código y garantiza una implementación homogénea para todos los experimentos realizados. El algoritmo opera sobre un grafo dirigido almacenado en memoria mediante listas de adyacencia y utiliza estructuras explícitas para gestionar los nodos pendientes de expansión y los ya explorados.

El proceso de búsqueda mantiene dos estructuras principales: una lista abierta, que contiene los vértices pendientes de expansión ordenados según su función de evaluación, y una lista cerrada, que almacena los vértices ya expandidos para evitar reexploraciones innecesarias. La lista abierta se gestiona como una colección de pares (vértice,$f(v)$), donde $f(v) = g(v) + h(v)$, seleccionándose en cada iteración el vértice con menor valor de evaluación. La lista cerrada se implementa como un conjunto, lo que permite comprobar de forma eficiente si un vértice ya ha sido expandido.

Para cada vértice extraído de la lista abierta, el algoritmo genera todos sus sucesores a partir de la lista de adyacencia del grafo. El coste acumulado $g(v)$ se actualiza sumando el coste del arco correspondiente y se almacena en una estructura asociativa que mantiene el mejor coste conocido para cada vértice. Cuando se detecta un camino de menor coste hacia un sucesor, se actualiza tanto su coste como el puntero al vértice padre, permitiendo reconstruir posteriormente el camino óptimo desde el destino hasta el origen.

El algoritmo finaliza cuando el vértice destino es extraído de la lista abierta, garantizando que el coste obtenido es mínimo siempre que la heurística utilizada sea admisible. En caso de que la lista abierta se vacíe sin alcanzar el destino, se concluye que no existe un camino entre los vértices indicados. Como resultado del proceso, se devuelve el camino óptimo reconstruido, el coste total del mismo y el número de vértices expandidos durante la búsqueda, métrica utilizada posteriormente en el análisis comparativo de rendimiento.

La implementación de Dijkstra se obtiene como un caso particular del algoritmo A*, utilizando una heurística nula $h(v) = 0$ para todos los vértices. De este modo, la función de

evaluación coincide con el coste acumulado $g(v)$, reproduciendo exactamente el comportamiento de una búsqueda no informada. Las variantes informadas se obtienen sustituyendo esta heurística por estimaciones basadas en la distancia geométrica entre vértices, manteniendo en todos los casos la optimalidad de la solución y permitiendo analizar el impacto de la heurística en el número de expansiones y el tiempo de ejecución.

Parte 3: Análisis de resultados

3.1 Análisis de resultados

En este apartado analizaremos todos los resultados obtenidos en las dos partes.

3.1.1 Análisis de resultados y complejidad del problema de la Parte 1

El crecimiento del tiempo de ejecución en la Parte 1 viene determinado por el tamaño del tablero y, principalmente, por el número de configuraciones posibles que deben explorarse para encontrar soluciones válidas. A medida que aumenta el tamaño del tablero, el espacio de búsqueda crece de forma exponencial, lo que incrementa significativamente el tiempo necesario para generar y comprobar soluciones. El algoritmo es capaz de resolver tableros de tamaño moderado, mientras que instancias de mayor tamaño se vuelven computacionalmente costosas debido al elevado número de combinaciones posibles.

Variables:
El problema BINAIRO se modela como un Problema de Satisfacción de Restricciones (CSP), donde se define una variable por cada casilla del tablero.

- El número total de variables es n^2, donde n es el tamaño de la cuadrícula (el tablero es n x n).
- Cada variable $x_{i,j}$ representa el símbolo, X ó O, contenido en la casilla de la fila i y columna j.
- Las casillas que están pre-rellenadas en la entrada no requieren variables o restricciones adicionales, sino que se manejan restringiendo el dominio de la variable a un único valor, X ó O. Las casillas vacías "." tienen el dominio completo: $\{X, O\}$.

Restricciones:
El modelo formal para el juego BINAIRO impone cuatro grupos fundamentales de restricciones. El número total de restricciones definidas es de $4n$. Estas se diferencian en cuatro grupos, con n restricciones en cada grupo:

- Igual número de símbolos en cada fila (n restricciones): se comprueba que cada fila contenga exactamente $\frac{n}{2}$ símbolos X y $\frac{n}{2}$ símbolos O. La implementación utiliza una restricción por cada fila para asegurar el equilibrio de X y O.
- Igual número de símbolos en cada columna (n restricciones): de manera análoga, se garantiza que cada columna contenga $\frac{n}{2}$ símbolos X y $\frac{n}{2}$ símbolos O. La implementación utiliza una restricción por cada columna para asegurar el equilibrio vertical.
- No tres símbolos iguales consecutivos en filas (n restricciones): se prohíbe la aparición de una secuencia de tres símbolos idénticos consecutivos en cualquier fila. La implementación añade una restricción de "consecutivos" por cada fila.
- No tres símbolos iguales consecutivos en columnas (n restricciones): se prohíbe la aparición de una secuencia de tres símbolos idénticos consecutivos en cualquier

columna. La implementación añade una restricción de “consecutivos” por cada columna.

Estas cifras (n^2 variables y $4n$ restricciones) son la base para el análisis de complejidad. Se debe observar que la complejidad del problema crece cuadráticamente con el número de variables, mientras que las restricciones crecen solo linealmente. Esta relación explica el crecimiento del coste computacional al aumentar el tamaño del tablero, tal como se demostró en la Prueba-5 (tablero 8x8), donde el solver requirió un mayor tiempo de resolución para encontrar múltiples soluciones.

3.1.2 Análisis de resultados y complejidad del problema de la Parte 2

En la Parte 2, el tiempo de ejecución depende fundamentalmente del número de vértices y arcos del grafo y, especialmente, del número de nodos expandidos durante la búsqueda. Las estructuras de datos empleadas crecen linealmente con el número de vértices: la lista ABIERTA, la lista CERRADA y los diccionarios de costes y predecesores tienen tamaño $O(|v|)$, mientras que el grafo se almacena una única vez mediante listas de adyacencia y coordenadas, con coste $O(|v| + |a|)$. En este contexto, el uso de heurísticas en A* permite reducir el número de nodos expandidos y, por tanto, el tiempo de ejecución respecto a Dijkstra, especialmente en instancias con múltiples ramas laterales.

Se compararon tres heurísticas: h_0 (Dijkstra), h_1 (distancia euclídea) y h_2, definida como una versión escalada de h_1 con $k = 1.2$. En todas las pruebas, las tres heurísticas devolvieron el mismo coste óptimo, validando la correcta implementación del algoritmo y la ausencia de sobreestimación en h_2.

En grafos simples, las diferencias en número de expansiones son reducidas, sin embargo, en instancias con múltiples ramas laterales, h_2 muestra una mejora clara en rendimiento. En la prueba de mayor tamaño, h_2 reduce el número de expansiones de 30 (h_0) y 26 (h_1) a 23, manteniendo el coste óptimo. Esto se debe a que el aumento controlado del peso heurístico penaliza los nodos que se alejan geométricamente del objetivo, reduciendo la exploración innecesaria.

Por tanto, h_2 se presenta como la heurística más eficiente en términos de expansiones, sin comprometer la optimalidad de la solución.

3.2 Casos de pruebas

3.2.1 Pruebas para la Parte 1

Prueba-1: tablero completo válido (6x6)
Se evaluó una instancia 6x6 completamente asignada y válida. El solver devuelve una única solución coincidente con la entrada, validando el correcto tratamiento de instancias ya resueltas.
Prueba-2: tablero completo inválido (6x6)
Se evaluó una instancia 6x6 completa que viola la restricción 2 (no tres iguales consecutivos en filas). El solver determina correctamente que no existen soluciones, validando la correcta implementación de dicha restricción.
Prueba-3: contradicción por casillas fijas (6x6)

Se evaluó una instancia 6x6 parcialmente asignada con una contradicción inicial. El solver detecta correctamente que no existen soluciones, validando la imposición de casillas fijas y la detección temprana de inconsistencias.
Prueba-4: tamaño mínimo (2x2)
Se evaluó una instancia mínima 2x2 sin casillas fijas. El solver devuelve las dos soluciones válidas, confirmando el correcto funcionamiento del modelo en el tamaño mínimo permitido.
Prueba-5: tablero con casillas pre-rellenadas (8x8)
Se evaluó una instancia 8x8 con un número intermedio de casillas fijadas. El solver encontró múltiples soluciones con un mayor tiempo de resolución, ilustrando el crecimiento del coste computacional y validando el comportamiento del modelo en instancias de mayor tamaño.
Prueba-6: tamaño impar no permitido (3x3)
Se evaluó una instancia 3x3 inválida por ser de tamaño impar. El programa detecta correctamente la invalidez durante la validación de la entrada y rechaza la instancia antes de construir el modelo.
Prueba-7: símbolo inválido en la entrada
Se evaluó una instancia 6x6 con un símbolo no permitido en la entrada. El programa detecta correctamente el error y rechaza la instancia durante la validación.
Prueba-8: fichero de entrada vacío
Se evaluó un fichero de entrada vacío. El programa detecta correctamente la ausencia de contenido y rechaza la instancia durante la validación.
Prueba-9: tablero con filas de distinta longitud
Se evaluó una instancia con filas de distinta longitud. El programa detecta correctamente que el tablero no es cuadrado y rechaza la entrada durante la validación.
Prueba-10: contradicción por exceso de símbolos en una fila (2x2)
Se evaluó una instancia 2x2 parcialmente asignada con una contradicción en una fila. El solver detecta correctamente que no existen soluciones, validando las restricciones de equilibrio.

3.2.2 Pruebas para la Parte 2

Para la resolución de las pruebas 1–6 se crean ficheros .gr y .co de tamaño reducido, independientes de la página web proporcionada en el enunciado. Estos ficheros representan mapas pequeños y controlados, basados en el mismo formato que los archivos originales de dicha página, pero adaptados para que la solución y el camino óptimos sean conocidos de antemano. Esta decisión permite validar de forma precisa y verificable el correcto funcionamiento del programa, asegurando que los algoritmos implementados seleccionan siempre el camino de menor coste.

Además, las pruebas se automatizaron mediante un script bash “script.sh” que ejecuta el programa con distintos pares origen–destino y distintos mapas, permitiendo comparar de forma sistemática los resultados obtenidos. El comando usado es:
"C:\Program Files\Git\bin\bash.exe" "./script.sh"

Prueba	**Descripción**	**H0**	**H1**	**H2**
1: múltiples caminos con	Se evaluó un grafo con varias rutas posibles entre origen y destino. Los	# vertices: 5 # arcos : 6	# vertices: 5 # arcos : 6	# vertices: 5 # arcos : 6

distinta optimalidad	algoritmos seleccionan correctamente el camino de menor coste, validando la obtención de la solución óptima.	Solución óptima encontrada con coste 8 Tiempo de ejecución: 0.0000 segundos # expansiones : 5 (226244.34 nodes/sec)	Solución óptima encontrada con coste 8 Tiempo de ejecución: 0.0000 segundos # expansiones : 5 (118483.41 nodes/sec)	Solución óptima encontrada con coste 8 Tiempo de ejecución: 0.0001 segundos # expansiones : 3 (55452.87 nodes/sec)
2: atajo vs camino largo	Se evaluó un grafo en el que existen dos rutas entre el origen y el destino: un camino más largo con mayor coste y un atajo con menor coste. Los algoritmos identifican correctamente el atajo como la solución óptima, validando la comparación de costes y la selección del camino de menor coste total.	# vertices: 6 # arcos : 6 Solución óptima encontrada con coste 5 Tiempo de ejecución: 0.0000 segundos # expansiones : 6 (255319.15 nodes/sec)	# vertices: 6 # arcos : 6 Solución óptima encontrada con coste 5 Tiempo de ejecución: 0.0001 segundos # expansiones : 6 (76530.61 nodes/sec)	# vertices: 6 # arcos : 6 Solución óptima encontrada con coste 5 Tiempo de ejecución: 0.0001 segundos # expansiones : 6 (87847.73 nodes/sec)
3: origen y destino coinciden	Se evaluó una instancia en la que el origen coincide con el destino. Los algoritmos devuelven correctamente la solución trivial con coste cero, validando el tratamiento del caso base.	# vertices: 3 # arcos : 3 Solución óptima encontrada con coste 0 Tiempo de ejecución: 0.0000 segundos # expansiones : 1 (58139.53 nodes/sec)	# vertices: 3 # arcos : 3 Solución óptima encontrada con coste 0 Tiempo de ejecución: 0.0000 segundos # expansiones : 1 (74074.07 nodes/sec)	# vertices: 3 # arcos : 3 Solución óptima encontrada con coste 0 Tiempo de ejecución: 0.0000 segundos # expansiones : 1 (56497.18 nodes/sec)
4: menos expansiones	Se comparó el comportamiento de los algoritmos de búsqueda observando el número de nodos expandidos. Los resultados muestran que el uso de una heurística informada permite reducir significativamente las expansiones necesarias para alcanzar la solución, manteniendo la optimalidad del camino encontrado.	# vertices: 30 # arcos : 38 Solución óptima encontrada con coste 60 Tiempo de ejecución: 0.0001 segundos # expansiones : 30 (500000.00 nodes/sec)	# vertices: 30 # arcos : 38 Solución óptima encontrada con coste 60 Tiempo de ejecución: 0.0001 segundos # expansiones : 26 (259222.33 nodes/sec)	# vertices: 30 # arcos : 38 Solución óptima encontrada con coste 60 Tiempo de ejecución: 0.0001 segundos # expansiones : 16 (205391.53 nodes/sec)
5: vértice inexistente	Se evaluó una instancia con un vértice inexistente. El programa detecta correctamente la entrada inválida y finaliza la ejecución mostrando un mensaje de error. Elegimos los archivos de la prueba 1 eligiendo como vértice de destino 7.	Error: alguno de los vértices no existe en el mapa		

6: camino no encontrado	Se evaluó una instancia con un camino inexistente. El programa detecta correctamente la entrada inválida y finaliza la ejecución mostrando un mensaje de error. Elegimos los archivos de la prueba 6 eligiendo como vértice 1-4	# vertices: 4 # arcos : 2 No se ha encontrado solución. Y en el fichero de salida: “No se ha encontrado camino”		
7: mismo resultado	Se evaluó que el resultado obtenido por el programa es el mismo que el resultado del ejemplo proporcionado en el enunciado. Se realizó con el mapa USA-road-d.BAY de la web, con los mismos parámetros establecidos.	# vertices: 321270 # arcos : 800172 Solución óptima encontrada con coste 10216 Tiempo de ejecución: 0.0001 segundos # expansiones : 5 (37735.85 nodes/sec)	# vertices: 321270 # arcos : 800172 Solución óptima encontrada con coste 10216 Tiempo de ejecución: 0.0118 segundos # expansiones : 4 (339.77 nodes/sec)	# vertices: 321270 # arcos : 800172 Solución óptima encontrada con coste 10216 Tiempo de ejecución: 0.0001 segundos # expansiones : 4 (78740.16 nodes/sec)

Conclusiones de la práctica

En esta práctica se han aplicado técnicas de satisfacción de restricciones y algoritmos de búsqueda para resolver problemas con un elevado componente combinatorio. En la primera parte, el problema BINAIRO se formuló como un CSP, permitiendo analizar cómo la definición de variables y restricciones influye directamente en la complejidad del problema y en el tiempo de resolución a medida que aumenta el tamaño del tablero.

En la segunda parte, se implementaron algoritmos de búsqueda de caminos mínimos, comparando una búsqueda no informada (Dijkstra) con una búsqueda informada (A*). Los resultados obtenidos muestran que ambos algoritmos encuentran soluciones óptimas, pero que el uso de heurísticas admisibles reduce significativamente el número de nodos expandidos, especialmente en grafos de gran tamaño, mejorando la eficiencia del proceso de búsqueda.

En conjunto, la práctica ha permitido comprender la importancia de una correcta modelización del problema, así como el impacto de las restricciones y las heurísticas en el rendimiento de los algoritmos. Los experimentos realizados confirman que la combinación de modelos bien definidos y técnicas de búsqueda informadas constituye una herramienta eficaz para abordar problemas complejos de forma eficiente.

Práctica 1- Programación lineal

Heurística y optimización- Curso 25/26