Universidad Carlos III

Curso Heurística y Optimización 2023/2024

Práctica 2

Satisfacción de Restricciones y Búsqueda Heurística

FECHA: 15/12/23 **ENTREGA:** Final

GRUPO: 84

Alumno 1

Nombre: -

NIU: -

Alumno 2

Nombre: -

NIU: -

Índice

Introducción

En este documento se describirán los modelos requeridos para cada parte de la práctica, estas descripciones incluirán explicaciones detalladas de todas las decisiones de diseño tomadas en cuanto a variables y restricciones, y cual es la utilidad de cada una de ellas.
Posteriormente, se analizarán los resultados obtenidos y cómo las restricciones planteadas limitan la solución del problema, y además se proporcionarán ejemplos adicionales para generar nuevas soluciones que permitan comprobar cómo las modificaciones afectan a la resolución.
Por último se incluirán las conclusiones y reflexiones finales respecto a la práctica, la dificultad de esta, y los conocimientos obtenidos en su desarrollo.

Descripción de los modelos implementados

1. Parte 1

1.1 Modelado

En el modelado de la parte 1 se pide implementar la asignación de los diferentes tipos de transportes (TSU, Transporte Sanitario Urgente, y TNU, Transporte Individual No Urgente) a las plazas de garaje dentro de los parkings.
Cualquiera de los transportes puede llevar instalado, o no, un congelador especial para transportar muestras y medicamentos. Estos vehículos deben estar aparcados en plazas especiales que disponen de conexión a la red eléctrica dentro de los parkings.

Un esquema de una posible configuración de un parking puede observarse en la *Tabla 1*.

1,1	1,2	1,3	1,4	1,5	1,6	>>
2,1	2,2	2,3	2,4	2,5	2,6	>>
3,1	3,2	3,3	3,4	3,5	3,6	>>
4,1	4,2	4,3	4,4	4,5	4,6	>>
5,1	5,2	5,3	5,4	5,5	5,6	>>

Tabla 1: Esquema de un *parking*.

Es relevante destacar que el enunciado planteado no solo indica las pautas generales para resolver el problema en cuestión, sino que especifica además un conjunto de restricciones que deben implementarse de manera precisa. Por tanto, para abordar eficientemente las complejidades asociadas a las posibles y diferentes ubicaciones de los vehículos en el estacionamiento, se pide emplear la librería *python-constraint*, conforme a las indicaciones y requerimientos específicos del contexto.
Las restricciones sirven son aquellas indicaciones que definen las distintas condiciones que deben cumplir los vehículos dentro del área de estacionamiento. Además de añadir un nivel adicional de complejidad al problema, son imperativas para asegurar un sistema de asignación de plazas que cumpla con los criterios establecidos de manera precisa y eficaz.

- Todo vehículo tiene que tener asignada una plaza y solo una.

$$\forall v \in V,\ \exists p \in P\colon\ asignado(v,p)\ \wedge\ \neg\, \exists p' \in P,\ p'\ \neq\ p\ :\ asignado(v,\ p')$$

  - V es el conjunto de todos los vehículos.
  - P es el conjunto de todas las plazas.
  - $asignado(v,p)$ es una función que devuelve verdadero si el vehículo v está asignado a la plaza p, y falso en caso contrario.

  Con respecto a la implementación en el programa, y su relación con el modelado pensado originalmente:
  El modelado de esta primera restricción, se basa en la creación de una estructura de datos que llamamos *P* (o parking) que representa todas las posibles posiciones de estacionamiento como tupla de coordenadas. Esto lo hacemos de forma que para el cumplimiento de esta primera restricción, para cada vehículo en la lista '*vehículos*', se agrega una única variable al problema. Por lo que para cada vehículo se agrega una variable al problema de restricciones.

- Dos vehículos distintos, como es natural, no pueden ocupar la misma plaza.

$$\forall v_1, v_2 \in V,\ \forall p \in P:\ (v_1 \neq v_2)\ \rightarrow\ \neg(asignado(v_1, p)\ \wedge\ asignado(v_2, p))$$

  - V es el conjunto de todos los vehículos.
  - P es el conjunto de todas las plazas.
  - $asignado(v, p)$ es una función que devuelve verdadero si el vehículo v está asignado a la plaza p, y falso en caso contrario.

  Con respecto a la implementación en el programa, y su relación con el modelado pensado originalmente:
  El modelado de esta segunda restricción. Dado que cada variable representa la posición de estacionamiento de un vehículo, usamos $AllDifferentConstraint$ con el objetivo de que se nos garantice que no haya superposición de ubicaciones entre vehículos distintos.

- Los vehículos provistos de congelador solo pueden ocupar plazas con conexión a la red eléctrica.

$$\forall v \in V_c, \forall p \in P_c:\ congelador(v)\ \rightarrow conexion_{eléctrica}(p)$$

  - V_c es el conjunto de vehículos provistos de congelador.
  - P_c es el conjunto de plazas con conexión a la red eléctrica.
  - $congelador(v)$ es una función que devuelve verdadero si el vehículo v está provisto de congelador, y falso en caso contrario.
  - $conexion_{eléctrica}(p)$ es una función que devuelve verdadero si la plaza p tiene conexión a la red eléctrica, y falso en caso contrario.

  Con respecto a la implementación en el programa, y su relación con el modelado pensado originalmente:
  El modelado de esta tercera restricción, se implementa mediante una verificación durante la construcción del problema de restricciones. Para cada vehículo en la lista, se examinan sus características, incluyendo la presencia de un congelador. Si el vehículo cuenta con un congelador, se añade una restricción al problema, indicando que su posición de estacionamiento debe estar dentro de un conjunto predeterminado de plazas con conexión eléctrica. Con este enfoque buscamos asegurar que los vehículos con congelador se asignen exclusivamente a ubicaciones que cumplen con el requisito de disponibilidad de conexión eléctrica, cumpliendo con las necesidades de este tipo de vehículos en el contexto del estacionamiento.

- Un vehículo de tipo TSU no puede tener aparcado por delante, en su misma fila, a ningún otro vehículo excepto si este es también de tipo TSU. Por ejemplo, si un TSU está aparcado en la plaza 2.3 no podrá haber aparcado un TNU en las plazas 2.4, 2.5, 2.6

$$\forall v_{TSU} \in V_{TSU}, \forall P_{TSU} \in P_{TSU}, \forall v_{TNU} \in V_{TNU}, \forall p_{TNU} \in P_{TNU}:$$

$$tipo(v_{TSU}) = TSU\ \wedge\ tipo(v_{TNU})\ \neq TSU\ \rightarrow fila(p_{TSU})\ = fila\,(p_{TNU})\ \wedge\ columna(p_{TSU}) < columna(p_{TNU})$$

  - V_{TSU} es el conjunto de vehículos de tipo TSU.
  - P_{TSU} es el conjunto de plazas ocupadas por vehículos de tipo TSU.
  - V_{TNU} es el conjunto de vehículos de tipo TNU.
  - P_{TNU} es el conjunto de plazas ocupadas por vehículos de tipo TNU.
  - $tipo\ (v)$ es una función que devuelve el tipo de un vehículo (TSU o TNU).
  - $fila(p)$ es una función que devuelve el número de fila de una plaza p.
  - $columna(p)$ es una función que devuelve el número de columna de una plaza p.

  Con respecto a la implementación en el programa, y su relación con el modelado pensado originalmente:
  El modelado de esta cuarta restricción, se implementa mediante un proceso de verificación entre los diferentes tipos de vehículos. Se examinan todas las combinaciones posibles de vehículos, y si se encuentra que un vehículo es de tipo TSU y el otro es de tipo TNU, se agrega una restricción al problema. Esta restricción asegura que el vehículo de tipo TSU no pueda tener otro vehículo de tipo TNU estacionado por delante de él en la misma fila del estacionamiento. Este enfoque se revela como una solución efectiva para modelar las condiciones específicas del estacionamiento, garantizando que se respete la restricción de colocación para los vehículos de tipo TSU.

- Por cuestiones de maniobrabilidad dentro del parking todo vehículo debe tener libre una plaza a izquierda o derecha (mirando en dirección a la salida). Por ejemplo, si un vehículo ocupa la plaza 3.3 no podrá tener aparcado un vehículo en la 2.3 y otro en la 4.3, al menos una de esas dos plazas deberá quedar libre.

$$\forall v \in V, \forall p \in P: \neg(asignado(v,p) \wedge (ocupado(p_{izquierda}) \vee ocupado(p_{derecha}))$$

  - V es el conjunto de todos los vehículos.
  - P es el conjunto de todas las plazas.
  - $asignado(v,p)$ es una función que devuelve verdadero si el vehículo v está asignado a la plaza p, y falso en caso contrario.
  - $ocupado(p)$ es una función que devuelve verdadero si la plaza p está ocupado por algún vehículo, y falso en caso contrario.
  - $p_{izquierda}$ y $p_{derecha}$ representan las plazas a la izquierda y derecha de la plaza p respectivamente.

  Con respecto a la implementación en el programa, y su relación con el modelado pensado originalmente:
  El modelado de esta quinta restricción, se basa en el correcto funcionamiento de la función "restricción de maniobrabilidad" que desempeña un papel crucial en la evaluación de la disposición de vehículos dentro del estacionamiento, garantizando condiciones óptimas de maniobrabilidad. Nuestra implementación contempla escenarios especiales para la primera y última fila, donde los vehículos tienen solo una dirección adyacente. En el caso general, verifica la presencia de vehículos en las posiciones laterales y, si se detecta alguna interferencia, la función devuelve "*False*", indicando una violación de la restricción de maniobrabilidad. En situaciones donde no hay conflictos, retorna "*True*", validando así la configuración del estacionamiento respecto a la mencionada restricción. Este enfoque contribuye significativamente a la planificación eficiente de la disposición de vehículos, mejorando la accesibilidad y movilidad dentro del estacionamiento.

1.2 Modo de resolución

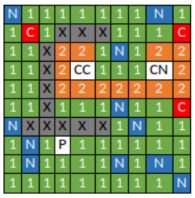
Se debe trabajar con la librería python constraint, por tanto en primer lugar se importan los módulos necesarios para resolver problemas de restricciones.
También se importa el módulo csv, que será necesario para escribir el archivo solución (de tipo csv).

Posteriormente hace falta procesar el archivo de entrada (de tipo txt) para extraer los datos del problema. Para ello se crea una función que devuelva esta información, "*procesar archivo"*, la función recibe como argumento una ruta, abre en modo lectura el archivo ubicado en la ruta especificada y lee su contenido línea por línea. La información se irá obteniendo tal y como se ha indicado en el enunciado, de forma que se irán leyendo el número de filas, de columnas, el ID, el tipo y si el vehículo dispone o no de congelador. Esta es una de las funciones más importantes de nuestro programa dado que es la encargada de identificar todo lo que hay en el estacionamiento, de procesarlo y de mandarlo correctamente al problema.

Pasamos ahora a la función de mayor importancia "*resolver problema*", dado que para el resto de funciones ya hemos explicado su implementación, asumimos que ya es conocido cuando lo mencionemos aquí. En esta función, se crea una instancia de la clase "*Problem*" y se define el conjunto de plazas de estacionamiento como una cuadrícula de filas y columnas. Luego, para cada vehículo, se agrega una variable al problema que representa la plaza de estacionamiento asignada a ese vehículo. Se aplican restricciones para garantizar que todas las plazas sean diferentes y, si un vehículo tiene la propiedad de congelación, se aplica una restricción adicional utilizando las plazas de conexión. Además, incorporamos dentro tal y como se nos pide las restricciones específicas del problema, como la función "*restricción aparcado por delante*", que evita que un TSU tenga otro vehículo aparcado por delante en la misma fila, y la función "*restricción maniobrabilidad*", que controla la maniobrabilidad de los vehículos en función de sus posiciones en la cuadrícula. Debido a la cantidad de vehículos que hemos decidido que reciba cada restricción es que está dentro de 1, 2 o 3 bucles "for". Finalmente, se obtienen las soluciones mediante "*getSolutions*" y se devuelven como resultado.

2. Parte 2

2.1 Modelado



1	Tiempo/Coste energético)
2	
N	Usuario no contagioso
C	Usuario contagioso
CC	Atención enf. Contagiosas
CN	Atención enf. no contagiosas
P	Parking
X	No transitable

Se nos plantea para la segunda parte un problema de planificación con búsqueda heurística, en el cuál:
Un servicio de urgencias incluye el traslado de pacientes a centros de tratamiento desde sus domicilios, utilizando una ambulancia. Hay dos tipos de usuarios: con enfermedades infectocontagiosas (marcados como C) y sin ellas (marcados como N). El mapa

esquemático incluye centros de atención (CC y CN), el parking (P), energía gastada (números) y posiciones no transitables (X).
El vehículo parte siempre del parking y regresa siempre a él. Puede recoger a múltiples pacientes en un solo trayecto y tiene 10 plazas, 2 de ellas para pacientes contagiosos. Los pacientes no contagiosos pueden ocupar estas plazas siempre que no viajen con pacientes contagiosos. El vehículo tiene 50 unidades de energía y recarga en el parking si es necesario.

Los pacientes contagiosos se recogen y dejan primero en cada trayecto. El vehículo se mueve entre celdas adyacentes horizontal o verticalmente. El tiempo/coste de tránsito por celdas no marcadas es de una unidad. La tarea es diseñar el traslado optimizando el uso de energía y cumpliendo las condiciones establecidas.

Entonces, empecemos con el modelado. Principalmente definimos una clase Estado, esto es fundamental dado que definirá nuestro "estado inicial" como punto de partida desde el cual el algoritmo de búsqueda comienza a explorar el espacio de estados en busca de una solución.

- Nuestra clase "*Estado*" se ha diseñado para representar los nodos utilizados en el algoritmo A*. Cada instancia de la clase encapsula la información esencial del estado del sistema en un momento dado durante la búsqueda de la solución. Los atributos principales incluyen la referencia al nodo padre, la posición del vehículo (identificada por fila y columna), el tipo de vehículo, la variación de energía acumulada, las listas de pacientes a recoger (tanto contagiosos como no contagiosos), el número de asientos disponibles para ambos tipos de pacientes, el parking, y las ubicaciones de los centros de atención para ambos tipos de pacientes.
  Además, se calculan y almacenan los costes asociados al nodo, que son fundamentales para el funcionamiento del algoritmo A*. El coste de la función de evaluación (f(x)) se desglosa en el costo acumulado desde el nodo inicial hasta el nodo actual (g(x)), el costo heurístico que estima la distancia al objetivo (h(x)), y la suma de ambos. La flexibilidad de la clase se refleja en la capacidad para inicializar los atributos con valores proporcionados o heredados del nodo padre, facilitando así la manipulación y creación eficiente de instancias de la clase durante la búsqueda de la solución óptima.

Ahora, pasamos a definir los operadores. Estos son fundamentales, ya que define cómo se puede pasar de un estado a otro en un espacio de búsqueda. Estos por tanto permiten una transición entre estados, una exploración del espacio de estados, y tendrán asociado un costo de transición.

- Nosotras consideramos las diferentes formas en la que los operadores podrían afectar al problema, llegando a la conclusión de que la función más afectada por ellos sería la de "*generar sucesores*" que es la encargada de generar los estados sucesores a partir del estado actual, considerando las posibles acciones que el vehículo puede realizar (o sea, los operadores que le afectan) en el problema de traslado de pacientes. Estas acciones están definidas por los operadores que determinan cómo el vehículo se puede mover en el mapa, recoger pacientes y dejar paciente, entre otras.
  En el código, los operadores están implementados como funciones como "*sucesores P*", "*sucesores N*", "*sucesores C*", "*sucesores CC*", "*sucesores CN*", y "*sucesores enteros*", cada uno encargado de calcular los cambios en el estado resultante después de realizar una acción específica (también operadores de movimiento). Los operadores afectan variables como la energía, la lista de pacientes por recoger, la ocupación de plazas no contagiosas y contagiosas, entre otras. Una breve explicación de cómo funcionan:

  1. "*recoger tipo n*": determina si se puede recoger un paciente no contagioso y devuelve un par de valores booleanos indicando si se puede recoger y si se debe asignar a un asiento contagioso.
  2. "*recoger tipo c*": determina si se puede recoger un paciente contagioso y devuelve un valor booleano indicando si se puede recoger.
  3. "*parada cn*": verifica la disponibilidad de asientos para pacientes contagiosos y no contagiosos en una parada y devuelve un par de valores booleanos indicando si se pueden asignar asientos a pacientes contagiosos y no contagiosos.
  4. "*sucesores P*": genera sucesores cuando se encuentra un estado con un parking en la casilla.
  5. "*sucesores N*": genera sucesores cuando se encuentra un paciente no contagioso en la casilla, dado un operador y una dirección de movimiento.
  6. "*sucesores C*": genera sucesores cuando se encuentra un paciente contagioso en la casilla, dado un operador y una dirección de movimiento.
  7. "*sucesores CC*": genera sucesores cuando se encuentra un centro de atención de pacientes contagiosos en la casilla.
  8. "*sucesores CN*": genera sucesores cuando se encuentra un centro de atención de pacientes no contagiosos en la casilla.
  9. "*sucesores enteros*": genera sucesores cuando se encuentra un número en la casilla.

La función principal "*generar sucesores*" utiliza estos operadores para crear nuevos estados sucesores en direcciones específicas (izquierda, derecha, arriba y abajo). Luego, los estados se ordenan según su coste total ("coste fx") para que el algoritmo de búsqueda explore primero aquellos con menor costo.

Definimos a continuación lo que es nuestro estado objetivo. Este es claramente aquel en el que todos los pacientes han sido transportados a sus respectivos centros de atención, y la ambulancia ha vuelto al parking acabando ahí su trabajo, por eso lo definimos con la meta en nuestro algoritmo A*.

Pasamos a la función de costo. El costo de una acción se determina por el tiempo (costo energético) asociado con el movimiento de una celda a otra.

- La función de costo es un componente clave que cuantifica el "costo" de las acciones realizadas por el vehículo en el traslado de pacientes. La función de costo se calcula sumando el costo actual de energía (medido en unidades de energía) en un estado particular, que se representa mediante el atributo "coste fx" de la clase "Estado".
  La función de costo se compone de dos términos principales: el costo acumulado de g(x) y el costo heurístico h(x). El primero, g(x), mide la distancia acumulada recorrida por el vehículo desde el estado inicial hasta el estado actual y se almacena en "coste gx". El segundo, h(x), representa una estimación heurística del costo restante para alcanzar el objetivo desde el estado actual, y se almacena en "coste hx".
  La suma de g(x) y h(x) da como resultado el costo total f(x) = g(x) + h(x), que es utilizado para ordenar los estados sucesores y determinar el próximo estado a explorar durante el proceso de búsqueda.
  Para nosotras, el costo de g(x) refleja el consumo de energía acumulado por el vehículo a medida que se desplaza entre las ubicaciones del mapa. Mientras que el costo heurístico h(x), que lo explicaremos en la sección de heurísticas, incorpora consideraciones adicionales.

Hablamos ahora de la función sucesores. Dada una configuración actual, la función sucesor generaría todas las acciones posibles y los estados resultantes después de aplicar esas acciones (operadores). Ahora repetiremos un poco lo que hablamos en el apartado de operadores, pero centrándonos más en cómo trabaja la función, no en su relación con los operadores.

- Para nosotras la función "generar sucesores" es fundamental en el contexto del algoritmo de búsqueda utilizado para resolver el problema de traslado de pacientes. Su objetivo principal es generar los estados sucesores a partir de un estado dado, considerando las acciones posibles que el vehículo puede realizar en el mapa. Procedemos a realizar una explicación detallada de cómo decidimos implementar la función.
  En primer lugar, se inicia una lista vacía llamada "sucesores" para almacenar los estados sucesores. Luego, se define una función interna llamada "agregar sucesor" que se encarga de evaluar la posibilidad de mover el vehículo en una dirección específica y agregar un estado sucesor si la acción es válida.
  Dentro de esta función, calculamos las nuevas coordenadas del vehículo según la dirección de movimiento especificada. Se verifica la validez del movimiento asegurándose de que las nuevas coordenadas estén dentro de los límites del mapa y que la celda correspondiente no sea un obstáculo ('X').
  Una vez validado el movimiento, se identifica el tipo de celda en la posición actual del vehículo en el mapa. Luego, se invocan funciones específicas para generar sucesores según el tipo de celda, sin mencionar detalles de implementación. Estas funciones se encargan de actualizar el estado del sistema en función de la acción realizada.
  Finalmente, la lista de sucesores se ordena en función de su costo total, lo que facilita la exploración del espacio de estados durante el proceso de búsqueda. La función devuelve la lista de sucesores generados, la cual está ordenada por su costo total.

Y para acabar de explicar las decisiones de modelado, hablamos de las heurísticas implementadas. Hemos implementado tres, y estas tienen como objetivo proporcionar estimaciones del costo restante desde un estado dado hasta el objetivo final.

- Heurística 1 (Dijkstra): Esta heurística asigna un valor de 0, ya que implementa el algoritmo de Dijkstra. No se considera un componente heurístico en el sentido tradicional, sino que calcula la distancia real desde el estado actual hasta el objetivo. En este caso, la ambulancia busca la ruta más corta sin tomar en cuenta la cantidad de pacientes restantes.
- Heurística 2: Esta heurística tiene un valor de 0 si no quedan pacientes por recoger, indicando que no hay costo adicional si ya se han recogido todos los pacientes. Si aún quedan pacientes, la heurística asigna un valor igual a la suma de la cantidad de pacientes no contagiosos y contagiosos restantes. Esta heurística penaliza ligeramente los estados con pacientes pendientes.
- Heurística 3: Esta heurística calcula una estimación de la distancia total recorrida para recoger a todos los pacientes. Utiliza el algoritmo de paciente más cercano, donde se elige el paciente más cercano en cada paso. La distancia total incluye la distancia entre pacientes y el regreso al punto de inicio.

- Heurística 4: Esta heurística presenta una heurística personalizada que se basa en la distancia mínima a todos los pacientes. Utiliza una estrategia similar a la del paciente más cercano, pero selecciona en cada paso el paciente que está a la distancia mínima actual. La distancia total incluye el regreso al punto de inicio.

En resumen, estas heurísticas son fundamentales para guiar el algoritmo de búsqueda hacia soluciones más eficientes, proporcionando estimaciones del costo futuro desde estados intermedios hasta el objetivo final. Cada heurística adopta un enfoque diferente para evaluar el progreso hacia el objetivo en función de la información disponible en el estado actual del sistema.

2.1 Modo de resolución

A grandes rasgos, en este programa queda dividido en:

1. Lectura de Mapa desde CSV:
   El bloque inicial del programa se encarga de importar las bibliotecas necesarias, como "*sys*", "*time*", y "*csv*". La función "*cargar mapa desde csv*" toma la ruta de un archivo CSV como entrada y procesa su contenido, convirtiendo los valores en enteros y eliminando comillas, si es necesario. Este proceso facilita la representación del mapa en una estructura de datos manipulable.
2. Volcado de Resultados:
   Las funciones "*write statistics*" y "*write solution*" se centran en la salida del programa. "write statistics" es responsable de registrar estadísticas clave, como el tiempo total de ejecución, el costo total, la longitud del plan y el número de nodos expandidos, en un archivo '.stat'. Por otro lado, "*write solution*" guarda los pasos de la solución en un archivo '.*output*'.
3. Función de Ordenación de Lista en Lista:
   La función "fusionar órdenes" es utilizada para fusionar dos listas ordenadas de nodos de acuerdo con su costo total ("*coste fx*"). Esto es crucial para mantener la lista de nodos abiertos ordenada durante la ejecución del algoritmo A*.
4. Clase Estado:
   La clase "*Estado*" modela un nodo en el espacio de búsqueda del algoritmo A*. Cada instancia representa un estado posible del sistema, con atributos que incluyen la posición de la ambulancia, la cantidad de pacientes, el costo acumulado, y funciones para calcular los costos asociados con las heurísticas implementadas.
5. Generar Sucesores:
   La función "*generar sucesores*" es esencial para la expansión del espacio de búsqueda. Para un estado dado, determina los estados sucesores posibles considerando las acciones de la ambulancia, como recoger pacientes o realizar paradas en centros.
6. Acciones del Vehículo:
   Las funciones como "*recoger tipo n*", "*recoger tipo c*", y otras definen las acciones específicas que puede realizar la ambulancia, como recoger pacientes no contagiosos, contagiosos, y hacer paradas en centros de atención. Estas funciones son vitales para la generación de sucesores. Es importante destacar que nosotras hemos decidido implementar la recogida de pacientes de forma que lo N tienen prioridad sobre los C a lo hora de recogida, pero los C tienen prioridad sobre los N a la hora de dejada, o sea que evaluamos qué tipo de pacientes ya hay en la ambulancia antes de permitir recoger a unos u otros. Mencionamos esto debido a que analizando resultados con algunos compañeros notamos una diferencia en los costes y vimos que era por esto.
7. Algoritmo A*:
   La función "*a estrella*" implementa el algoritmo de búsqueda A*, que guía la exploración del espacio de estados hacia la solución óptima. Utiliza una lista de nodos abiertos ordenada por costo total ("*coste fx*") y una lista de nodos cerrados para evitar la redundancia en la exploración.
8. Programa Principal:
   La función "*main*" sirve como punto de entrada del programa. Lee la ruta del archivo CSV y la heurística desde la línea de comandos, carga el mapa, define el estado inicial y llama a la función A* para encontrar la solución. Las estadísticas y la solución se escriben en los archivos correspondientes.

Cosas a destacar adicionales, con el objetivo de optimizar el código para que compile de forma más rápida y para una resolución más rápida de problemas:

- Función hash es una función matemática que toma una entrada (o "*mensaje*") y devuelve un valor hash, que es generalmente una cadena de caracteres o un número. El propósito principal de una función hash es mapear datos de longitud variable a datos de longitud fija. Estas funciones son ampliamente utilizadas en la informática para diversas aplicaciones. Esta función hash la utilizamos para asignar de manera eficiente las claves a ubicaciones específicas en una tabla hash. Esto facilita la búsqueda, inserción y eliminación de elementos, ya que el valor hash se utiliza como índice para acceder a la ubicación correspondiente en la estructura de datos.

Esto lo usamos en el algoritmo A*, donde el conjunto "*closed list*" se utiliza para almacenar los estados que ya han sido expandidos, evitando así expandir el mismo estado más de una vez y previniendo bucles infinitos. El uso de un conjunto en lugar de una lista para "*closed list*" tiene ventajas en términos de eficiencia, especialmente en operaciones de búsqueda y verificación de pertenencia.
Para nuestro código, "*closed list*" es un conjunto que almacena los estados ya expandidos, lo que significa que no se volverán a expandir. Esto mejora la eficiencia del algoritmo A* al evitar redundancias en la expansión de estados y reducir la complejidad temporal del algoritmo. La función hash se utiliza implícitamente en las operaciones de agregar y verificar pertenencia, contribuyendo así a la eficiencia del conjunto "*closed list*" en la implementación del algoritmo A*.

- La función "*ordenar lista*" toma dos listas ordenadas, "*lista a*" y "*lista b*", y las combina en una sola lista ordenada, asegurando que el orden se mantenga. La función utiliza dos índices ("*index a*" y "*index b*") para recorrer ambas listas de manera simultánea. Compara los elementos actuales en las posiciones de los índices y agrega el elemento más pequeño (en términos de "*coste fx*") a la lista resultante, avanzando el índice correspondiente. Este proceso continúa hasta que una de las listas se agota.
  Luego, la función agrega los elementos restantes de ambas listas a la lista resultante, si los hay. El uso de esta función en el contexto del algoritmo A* es clave para mantener la lista "open list" ordenada por el valor de la función de coste total ("*coste fx*"). La lista "*open list*" contiene los estados pendientes de expansión, y al mantenerla ordenada, se selecciona eficientemente el estado con el costo total mínimo en cada iteración del bucle principal del algoritmo A*.
  Esta implementación mejora la eficiencia del algoritmo A* al garantizar que los estados en "*open list*" estén ordenados, lo que facilita la selección del próximo estado a expandir. La complejidad temporal de las operaciones de inserción y eliminación en "*open list*" se reduce, contribuyendo así a una ejecución más eficiente del algoritmo A*.

- Implementamos una función que nos indicase si una heurística era admisible o no.

Análisis de resultados

1. Análisis de resultados del Modelado 1. Estudio de la complejidad, variables y restricciones.

El presente apartado de la memoria se adentra en el análisis detallado de los resultados obtenidos a través del Modelado 1, centrándose en un estudio de la complejidad inherente al problema abordado. La comprensión profunda de las variables y restricciones aplicadas durante el proceso de modelado es esencial para evaluar la eficacia del programa.

Durante este análisis, exploraremos la complejidad computacional del modelo. Además, examinaremos en detalle las variables que influyen en la resolución del problema, buscando insights que permitan optimizar y perfeccionar el rendimiento del modelo.La evaluación de las restricciones introducidas en el modelado es otro componente fundamental de este apartado. Nos sumergiremos en el impacto de estas restricciones en la solución final, analizando cómo afectan la viabilidad y la eficiencia del proceso de resolución. Este enfoque crítico nos permitirá identificar áreas de mejora, refinar estrategias y, en última instancia, fortalecer la utilidad de nuestro modelo en el contexto del problema abordado.

Planteamos a continuación diferentes problemas a resolver y cómo los realiza el programa.

1. Problema 1

```
3x3
PE:(1,1)(1,2)(1,3)
1-TSU-C
2-TNU-C
3-TNU-C
```

Con este ejemplo de problema buscamos una fácil observación del comportamiento del programa. Este deberá mostrarnos únicamente 2 soluciones dado que estamos ante tres plazas eléctricas acompañadas de tres vehículos necesitados de conexión, y al ser uno de ellos de tipo TSU y los otros TNU, por la restricción 4 únicamente se puede cumplir aquellas posiciones en las que los vehículos TNU están intercambiados entre ellos en las plazas (1,1) y (1,2) dado que siempre deben tener delante al TSU

Así pues las dos posibles combinaciones se ven de esta forma:

3-TNU-C	2-TNU-C	1-TSU-C
-	-	-
-	-	-

2-TNU-C	3-TNU-C	1-TSU-C
-	-	-
-	-	-

Entonces, podemos observar que los únicos vehículos intercambiables que eran los TNU-C, se intercambiaron para las soluciones y el TSU-C no dio paso a otras soluciones.

En cuanto a la velocidad de ejecución del problema, observamos que el tiempo de ejecución fue de 0,0275 segundos, más adelantes cuando ya hayamos evaluado un par de soluciones más hablaremos del rendimiento del programa.

2. Problema 2

```
2x2
PE:(1,1)(1,2)
1-TSU-C
2-TNU-X
```

Con este ejemplo buscamos que se observe fácilmente el cumplimiento de la restricción 5, dado que para este ejemplo planteamos dos plazas con conexión eléctrica pero únicamente un vehículo eléctrico. Este planteamiento provocará, siguiendo las indicaciones del enunciado, que el vehículo TNU-X pueda ocupar las plazas (1,1)/(2,1)/(2,2) dado que nunca se podrá colocar delante del TSU-C.

Así pues las tres posibles combinaciones se ven de esta forma:

2-TNU-X	1-TSU-C
-	-

-	1-TSU-C
2-TNU-X	-

1-TSU-C	-
-	2-TNU-X

Tal y como habíamos anticipado, en ninguna circunstancia un vehículo se colocará debajo de otro, y además no mencionamos que este caso es uno de los más importantes a destacar desde nuestro punto de vista dado que estamos haciendo la evaluación de la restricción 5 para los vehículos en la fila 1 y en la última fila únicamente, que al final creemos que eran las dos condiciones de la restricción 5 más complicadas de ver, sin embargo son excepciones que deben considerarse de manera obligatoria.
En cuanto a la velocidad de ejecución del problema, observamos que el tiempo de ejecución fue de 0,0113 segundos, más adelantes cuando ya hayamos evaluado un par de soluciones más hablaremos del rendimiento del programa.

3. Problema 3

```
5x4
PE:(1,1)(1,2)(1,3)(2,1)
1-TNU-C
2-TSU-C
3-TNU-C
4-TSU-C
```

Con este ejemplo buscamos que se observe con facilidad que el problema no tiene solución.
Entonces, si observamos detenidamente el ejemplo vemos cómo para este ejemplo estamos planteando 4 plazas eléctricas y 4 vehículos con necesidad de plaza eléctrica, sin embargo dos de las plazas eléctricas están colocadas de forma que una está debajo de otra.
Esto provoca que a la hora de colocar los vehículos no exista combinación legal posible, dado que para siempre se colocará un vehículo debajo de otro que violaría la restricción 5, no permitiendo la maniobrabilidad del vehículo de la plaza (1,1).

En cuanto a la velocidad de ejecución del problema, observamos que el tiempo de ejecución fue de 0,0114 segundos, más adelantes cuando ya hayamos evaluado un par de soluciones más hablaremos del rendimiento del programa.

4. Problema 4

```
5x4
PE:(1,1)(1,2)(3,1)
1-TNU-C
2-TSU-X
3-TNU-X
4-TNU-C
```

Con este ejemplo, ya no vamos a poder observar fácilmente los resultados, como para poder comprobar que todos son correctos. Esto es debido a que como observamos estamos definiendo un problema en el que el parking es de 20 plazas, con 3 eléctricas pero con 4 vehículos y solo 2 ellos con necesidad de conexión, esto supone que las combinaciones de posibles soluciones sea mayor.
Entonces, lo que vamos a hacer es coger unos cuantos de ellos y ver que las restricciones se cumplen.

Así pues seleccionamos 8 posibles combinaciones que se ven de esta forma:

4-TNU-C	-	-	-
-	-	-	-
1-TNU-C	-	-	3-TNU-X
2-TSU-X	-	-	-
-	-	-	-

4-TNU-C	1-TNU-C	-	-
-	-	-	-
-	-	-	-
-	-	-	2-TSU-X
-	3-TNU-X	-	-

-	1-TNU-C	-	-
-	-	-	2-TSU-X
4-TNU-C	-	3-TNU-X	-
-	-	-	-
-	-	-	-

4-TNU-C	1-TNU-C	-	-
-	-	3-TNU-X	-
-	-	-	-
-	-	-	-
-	-	2-TSU-X	-

-	1-TNU-C	-	-
-	-	3-TNU-X	-
4-TNU-C	-	-	-
2-TSU-X	-	-	-
-	-	-	-

4-TNU-C	-	-	-
-	-	-	-
1-TNU-C	-	-	2-TSU-X
-	-	-	-
-	-	3-TNU-X	-

4-TNU-C	1-TNU-C	-	-
-	-	-	2-TSU-X
3-TNU-X	-	-	-
-	-	-	-
-	-	-	-

-	1-TNU-C	-	3-TNU-X
-	-	-	-
4-TNU-C	-	-	-
-	-	-	-
-	-	2-TSU-X	-

Con este ejemplo, podemos observar estas 8 posibles distribuciones seleccionadas aleatoriamente, y por ejemplo en la imagen 1 podemos ver cómo se cumple a la perfección la restricción 5 también para los vehículos en filas intermedias, dejando para ambas una plaza arriba o abajo.
También observamos que en ningún momento se coloca un TNU delante de un TSU. Y dado que las plazas con conexión son más que el número de vehículos que necesitan conexión se darán situaciones en las que la plaza con conexión quede ocupada por un vehículo sin necesidad de conexión, lo cuál es correcto.

En cuanto a la velocidad de ejecución del problema, observamos que el tiempo de ejecución fue de 0,0978 segundos, más adelantes cuando ya hayamos evaluado un par de soluciones más hablaremos del rendimiento del programa.

5. Problema 5

```
5x6
PE:(1,1)(1,2)(2,1)(4,1)(5,1)(5,2)
1-TSU-C
2-TNU-X
3-TNU-X
4-TNU-C
5-TSU-X
6-TNU-X
7-TNU-C
8-TSU-C
```

Con este ejemplo tampoco vamos a poder evaluar las soluciones de forma cuidadosa dado que el número de soluciones como era de esperarse es muy alto. Añadimos este ejemplo dado que es el que se nos planteó en el enunciado de la práctica, y imaginamos que puede ayudar en la corrección de la misma.
Haciendo un análisis externo del posible número de soluciones que nos puede dar observamos que al igual que en el ejemplo anterior nos encontramos ante un número elevado de plazas con conexión, 6 plazas, frente a un número no tan alto de vehículos que necesitan conexión, 4 vehículos.

Esto provocará más posibles combinaciones dado que entra la opción de que los vehículos sin necesidad de conexión eléctrica también puedan ocupar esas plazas. Hablaremos más detenidamente de esto en la parte de evaluación de incrementos de soluciones.

En cuanto a la velocidad de ejecución del problema, observamos que el tiempo de ejecución fue de 357,62 segundos, más adelante hablamos del rendimiento del programa.

Evaluación del incremento de soluciones en función del problema

Para este apartado lo que vamos a evaluar es cómo y porqué se produce un incremento en el número de soluciones aun sin haber realizado un cambio muy brusco en los datos del input. Aunque es fácil de ver esto se debe a que estamos ampliando el rango de combinaciones siempre que:

1. Ampliamos el rango de plazas de parking.
2. Ampliamos el número de plazas de parking con conexión y reducimos el número de vehículos con congelador, dado que esto provoca que los vehículos sin congelador también puedan ocupar esas plazas.
3. La falta de vehículos TSU, dado que estos limitan el movimiento de los TNU, que siguiendo la restricción 4 sabemos que un TNU no puede colocarse delante de un TSU.

Y estas son algunas de las observaciones que concluimos, luego para nuestros problemas planteados vemos que:

1. Problema 1: 2 soluciones. Tiene sentido dado que solo tenemos 3 plazas con conexión y 3 vehículos con congelador, además dos de ellos (TNU) quedan limitados por el TSU que siempre se coloca delante de ellos en las 2 distribuciones.
2. Problema 2: 3 soluciones. Aunque tiene menos vehículos incrementa las combinaciones debido a que estamos frente a un parking con dos plazas de conexión y un vehículo con conexión, sin embargo no se incrementa demasiado dado que las dimensiones del parking no es grande entonces no pueden haber muchas combinaciones cumpliendo la restricción 5.
3. Problema 3: Sin solución
4. Problema 4: 1334. La misma lógica aplicada para el problema 2, solo que ahora las dimensiones del parking sí que son mayores.
5. Problema 5: 2175288. La misma lógica del problema 4, solo que ahora aumentamos las plazas con conexión.

Evaluación del incremento del tiempo en función del problema

Los resultados de apartado después de observar el motivo de incremento de soluciones cobran mucho sentido, entonces pasamos a dejarlos únicamente comentados destacando que el motivo es tanto de la implementación del problema como de la cantidad de combinaciones que se producen por el problema.

1. Problema 1: 0,0275
2. Problema 2: 0,0113
3. Problema 3: 0,0114
4. Problema 4: 0,0978
5. Problema 5: 357,62

2. Análisis de resultados del Modelado 2. Estudio de la complejidad, variables y restricciones.

En este apartado de la memoria, vamos a evaluar el correcto funcionamiento del código implementado, para ello hemos decidido llevar a cabo un par de pruebas con ejemplos específicos que arrojen la información que nos interesa, y de igual manera hacemos esto para comparar las heurísticas implementadas.

1. Problema 1

```
CN;1;CC;C;N;P
```

Este mapa, es un mapa sencillo, por la implementación que hemos realizado nosotras, primero deberá recoger a los N, luego a los C, dejar a los C y luego dejar a los N. Todo esto obviamente gastando la respectiva energía que le corresponde por casilla y empezando y acabando en el parking.
Aplicando las heurísticas diseñadas por nosotros, se nos plantean los siguientes resultados:

```
Heurística y costes admisibles
Tiempo total: 0.0006072521209716797
Coste Total: 10
Longitud del plan: 11
Nodos expandidos: 42
(1,6):P:50
(1,5):N:49
(1,4):C:48
(1,3):CC:47
(1,2):1:46
(1,1):CN:45
(1,2):1:44
(1,3):CC:43
(1,4):1:42
(1,5):N:41
(1,6):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.0005342960357666016
Coste Total: 10
Longitud del plan: 11
Nodos expandidos: 42
(1,6):P:50
(1,5):N:49
(1,4):C:48
(1,3):CC:47
(1,2):1:46
(1,1):CN:45
(1,2):1:44
(1,3):CC:43
(1,4):1:42
(1,5):N:41
(1,6):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.0010693073272705078
Coste Total: 10
Longitud del plan: 11
Nodos expandidos: 42
(1,6):P:50
(1,5):N:49
(1,4):C:48
(1,3):CC:47
(1,2):1:46
(1,1):CN:45
(1,2):1:44
(1,3):CC:43
(1,4):1:42
(1,5):N:41
(1,6):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.0005156993865966797
Coste Total: 10
Longitud del plan: 11
Nodos expandidos: 42
(1,6):P:50
(1,5):N:49
(1,4):C:48
(1,3):CC:47
(1,2):1:46
(1,1):CN:45
(1,2):1:44
(1,3):CC:43
(1,4):1:42
(1,5):N:41
(1,6):P:50
```

Análisis de resultados:
Procedemos a hacer un análisis de los resultados obtenidos, y tal y como era de esperarse para un ejemplo tan reducido un cambio en las heurísticas no provocará un gran cambio en la ejecución del programa. Por eso mejor pasaremos a evaluar mapas de dimensiones mayores en los que los cambios son más notorios.

2. Problema 2

```
CN;P;C;C
1;N;CC;C
1;1;1;C
N;N;N;X
```

Este mapa es un poco más complejo que el anterior, en este ya consideramos un número mayor de contagiosos y no contagiosos. Al igual que para el anterior se espera una ejecución en la que recoge a los no contagiosos primero, pero si en el camino se encuentra a algún contagioso lo recoge y lo deja primero y luego deja a los no contagiosos.
Dejo aquí debajo los datos arrojados tras la ejecución de cada heurística, van de la 1 a la 4, respectivamente:

```
Heurística y costes admisibles
Tiempo total: 9.700552225112915
Coste Total: 18
Longitud del plan: 19
Nodos expandidos: 8811
(1,2):P:50
(2,2):N:49
(2,1):1:48
(3,1):1:47
(4,1):N:46
(4,2):N:45
(4,3):N:44
(3,3):1:43
(3,4):C:42
(2,4):C:41
(2,3):CC:40
(2,4):1:39
(1,4):C:38
(1,3):C:37
(2,3):CC:36
(2,2):N:35
(2,1):1:34
(1,1):CN:33
(1,2):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.6230731010437012
Coste Total: 18
Longitud del plan: 19
Nodos expandidos: 3042
(1,2):P:50
(2,2):N:49
(3,2):1:48
(4,2):N:47
(4,1):N:46
(4,2):N:45
(4,3):N:44
(3,3):1:43
(3,4):C:42
(2,4):C:41
(2,3):CC:40
(1,3):C:39
(1,4):C:38
(1,3):1:37
(2,3):CC:36
(2,2):N:35
(2,1):1:34
(1,1):CN:33
(1,2):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.05735492706298828
Coste Total: 18
Longitud del plan: 19
Nodos expandidos: 752
(1,2):P:50
(2,2):N:49
(3,2):1:48
(4,2):N:47
(4,1):N:46
(4,2):N:45
(4,3):N:44
(3,3):1:43
(3,4):C:42
(2,4):C:41
(2,3):CC:40
(1,3):C:39
(1,4):C:38
(1,3):1:37
(2,3):CC:36
(2,2):N:35
(2,1):1:34
(1,1):CN:33
(1,2):P:50
```

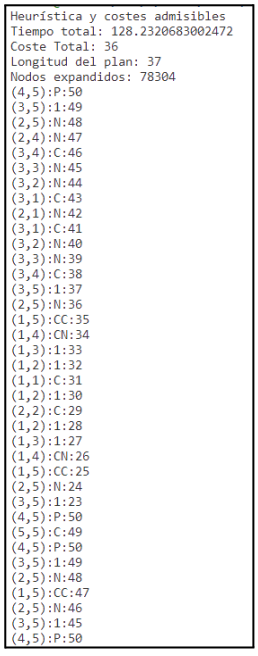
```
Heurística y costes admisibles
Tiempo total: 0.05670881271362305
Coste Total: 18
Longitud del plan: 19
Nodos expandidos: 747
(1,2):P:50
(2,2):N:49
(3,2):1:48
(4,2):N:47
(4,1):N:46
(4,2):N:45
(4,3):N:44
(3,3):1:43
(3,4):C:42
(2,4):C:41
(2,3):CC:40
(1,3):C:39
(1,4):C:38
(1,3):1:37
(2,3):CC:36
(2,2):N:35
(2,1):1:34
(1,1):CN:33
(1,2):P:50
```

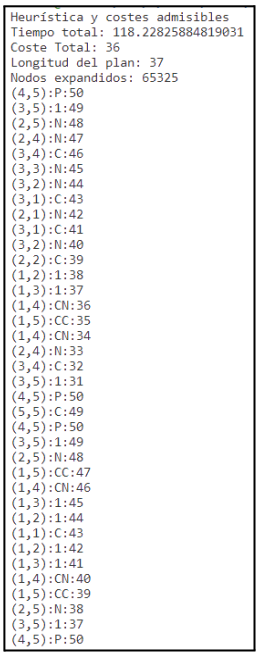
Análisis de resultados:
Podemos observar que para este mapa si que se observa mejora a medidas que utilizamos mejores heurísticas, o simplemente heurísticas diferentes a las de fuerza bruta. Esto como ya sabemos teóricamente es debido a que cambiamos el valor de los costes.
Así pues, observamos que mientras por fuerza bruta expandimos cerca de 9 mil nodos, cuando usamos la heurística 3 o 4 únicamente expandimos hasta un máximo de 752 nodos, lo cual es un porcentaje de mejora muy alto.
Y pasa exactamente lo mismo con el tiempo de ejecución, que obviamente lo relacionamos directamente con la reducción de nodos generados.
En cuanto a la longitud del plan, seguramente por como está planteada nuestra implementación puede ser un poco mayor dado que nosotras vamos y volvemos valor y volvemos, obviamente siempre respetando la cantidad de asientos disponibles y todo lo demás. Por que por ejemplo si lo hubiesemos implementado de la otra forma, no haría falta ir a por todos los N, y si pasa por una C y su CC, si no que puede recoger a los C directamente porque al expandir nodos los generas, dejarlos y ya ir a por los N y dejarlos en su CN, y pues para este caso sería un camino más corto y por consiguiente el óptimo para este ejercicio. Aunque este es el óptimo para como lo tenemos planteado nosotras claro.

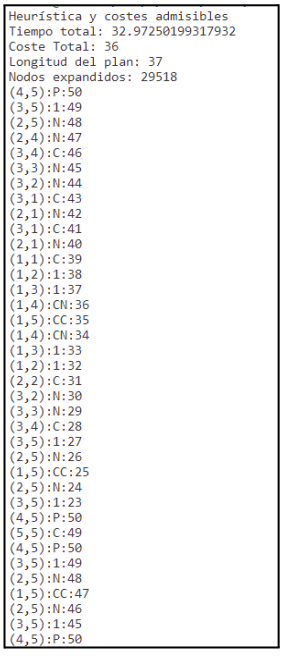
3. Problema 3

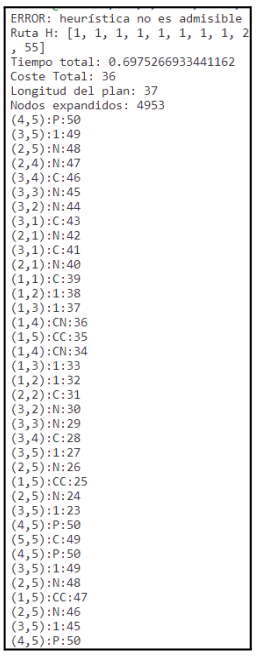
```
C;1;1;CN;CC
N;C;X;N;N;C
C;N;N;C;1;N
1;1;X;X;P;C
X;X;X;1;C;N
```

Para este mapa igual que en el anterior evaluamos una mayor cantidad de pacientes tanto contagiosos como no contagiosos. En este caso al tratarse de un mapa de mayores dimensiones esperamos que el tiempo de ejecución sea mayor.
Dejo aquí debajo los datos arrojados tras la ejecución de cada heurística, van de la 1 a la 4, respectivamente:









Análisis de resultados:
Tal y como habíamos mencionado en la breve descripción del problema el tiempo de ejecución es mucho mayor para la primera heurística porque es por fuerza bruta y para las restantes menos la 4, porque siguen sin estar lo suficientemente informadas, es por ello que si bien es verdad que el tiempo de ejecución con la heurística 3 se reduce mucho, si hubiésemos definido una mejor heurística se haría en mucho menos tiempo.
Ahora me gustaría destacar cómo para este ejemplo la heurística 4 deja de ser admisible. Esto provoca que la compilación sea tan rápida como se muestra en la imagen. Esta falta de admisibilidad puede deberse a cómo está estructurado el mapa, dado que observamos que para mayores dimensiones la admisibilidad de la heurística 4 dejaba de ser existente y por eso diseñamos otras nuevas y decidimos dejar esta para poder mencionarla en la memoria y porque además sí que es admisible para ciertos ejemplos.

4. Problema 4

```
N;N;N;N;C;CC;CN
N;X;X;X;X;X;X
N;X;X;X;X;X;X
N;X;X;X;X;X;X
N;X;X;X;X;X;X
N;X;X;X;X;X;X
N;X;X;X;X;X;X
P;X;X;X;X;X;X
```

Hemos elegido este mapa, dado que nos pareció interesante ver el comportamiento tan especial para este caso.Debido a la interpretación que tomamos nosotras del enunciado consideramos que la recogida de los usuarios no contagiosos tenía prioridad, y por tanto en el caso de que el vehículo tuviese ocupadas las 10 plazas y se cruzase con un contagiosos obviamente este no iba poder ser recogido y por tanto no lo íbamos a dejar en su centro antes que a un no contagioso. Y hablando con nuestros compañeros nos dimos cuenta de eso, que bueno realmente es una segunda interpretación del enunciado pero queríamos dejarlo claro.

Dejo aquí debajo los datos arrojados tras la ejecución de cada heurística, van de la 1 a la 4, respectivamente:

```
Heurística y costes admisibles
Tiempo total: 0.017083406448364258
Coste Total: 28
Longitud del plan: 29
Nodos expandidos: 795
(8,1):P:50
(7,1):N:49
(6,1):N:48
(5,1):N:47
(4,1):N:46
(3,1):N:45
(2,1):N:44
(1,1):N:43
(1,2):N:42
(1,3):N:41
(1,4):N:40
(1,5):C:39
(1,6):CC:38
(1,7):CN:37
(1,6):CC:36
(1,5):C:35
(1,6):CC:34
(1,5):1:33
(1,4):N:32
(1,3):N:31
(1,2):N:30
(1,1):N:29
(2,1):N:28
(3,1):N:27
(4,1):N:26
(5,1):N:25
(6,1):N:24
(7,1):N:23
(8,1):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.0135419368743896 48
Coste Total: 28
Longitud del plan: 29
Nodos expandidos: 683
(8,1):P:50
(7,1):N:49
(6,1):N:48
(5,1):N:47
(4,1):N:46
(3,1):N:45
(2,1):N:44
(1,1):N:43
(1,2):N:42
(1,3):N:41
(1,4):N:40
(1,5):C:39
(1,6):CC:38
(1,7):CN:37
(1,6):CC:36
(1,5):C:35
(1,6):CC:34
(1,5):1:33
(1,4):N:32
(1,3):N:31
(1,2):N:30
(1,1):N:29
(2,1):N:28
(3,1):N:27
(4,1):N:26
(5,1):N:25
(6,1):N:24
(7,1):N:23
(8,1):P:50
```

```
Heurística y costes admisibles
Tiempo total: 0.011178493499755 86
Coste Total: 28
Longitud del plan: 29
Nodos expandidos: 540
(8,1):P:50
(7,1):N:49
(6,1):N:48
(5,1):N:47
(4,1):N:46
(3,1):N:45
(2,1):N:44
(1,1):N:43
(1,2):N:42
(1,3):N:41
(1,4):N:40
(1,5):C:39
(1,6):CC:38
(1,7):CN:37
(1,6):CC:36
(1,5):C:35
(1,6):CC:34
(1,5):1:33
(1,4):N:32
(1,3):N:31
(1,2):N:30
(1,1):N:29
(2,1):N:28
(3,1):N:27
(4,1):N:26
(5,1):N:25
(6,1):N:24
(7,1):N:23
(8,1):P:50
```

```
ERROR: heurística no es admisible
Ruta H: [1, 1, 1, 1, 1, 1, 1, 1, 1,
Tiempo total: 0.00973200798034668
Coste Total: 28
Longitud del plan: 29
Nodos expandidos: 446
(8,1):P:50
(7,1):N:49
(6,1):N:48
(5,1):N:47
(4,1):N:46
(3,1):N:45
(2,1):N:44
(1,1):N:43
(1,2):N:42
(1,3):N:41
(1,4):N:40
(1,5):C:39
(1,6):CC:38
(1,7):CN:37
(1,6):CC:36
(1,5):C:35
(1,6):CC:34
(1,5):1:33
(1,4):N:32
(1,3):N:31
(1,2):N:30
(1,1):N:29
(2,1):N:28
(3,1):N:27
(4,1):N:26
(5,1):N:25
(6,1):N:24
(7,1):N:23
(8,1):P:50
```

Análisis de resultados:
De este únicamente queríamos destacar lo que hemos mencionado al inicio. Si nos adentramos en la evaluación de los resultados, pues aún siendo un problema grande podemos ver que se ejecuta a gran velocidad debido a que la cantidad de nodos a expandir es bastante reducida porque toda la zona centro hacia la derecha está bloqueada por obstáculos.

Conclusiones

Como conclusiones de la práctica, consideramos que nos ha permitido entender mucho mejor la parte del temario relacionado con los algoritmos de búsqueda y la heurística dado que la implementación a nuestro parecer nos exigía un mínimo de conocimiento sobre lo que estábamos haciendo además de que a medida que avanzabamos teníamos que pensar en formas de optimizar el código y cosas por el estilo.

Si que es verdad que nos hubiera gustado disponer de más recursos a nivel práctico antes de empezar, porque la parte 1 se nos complicó debido a la falta de conocimiento sobre la herramienta, además también de no entender mucho el temario.

Finalmente, nos ha parecido una práctica compleja pero retadora.

PRÁCTICA 2. SATISFACCIÓN DE RESTRICCIONES Y BÚSQUEDA HEURÍSTICA

Heurística y Optimización

Grado de Ingeniería en Informática. Curso 2023-2024