Universidad Carlos III

Heurística y Optimización

Curso 2024-25

Laboratorio 2: CSP y Búsqueda

Miembros:

Raul Armas 100495746@alumnos.uc3m.es

~~**Enrique Ruiz Romanos 100495896@alumnos.uc3m.es**~~

INTRODUCCIÓN

PARTE 1

El objetivo es asignar a cada avión, de manera eficiente y sin conflictos, un taller o parking en cada una de las franjas horarias disponibles durante un día de mantenimiento. Además, deben cumplirse varias restricciones importantes, como la capacidad limitada de los talleres, las restricciones en el tipo de tareas que pueden realizarse en cada taller, y la condición de que los aviones no se encuentren en celdas adyacentes al mismo tiempo para evitar colisiones. Usaremos CSP para resolver este problema

MODELO

Este es un problema de satisfacción de restricciones CSP, lo he definido como:

1. **Variables:**

   •Cada variable representa la ubicación del avión A_i en una franja horaria t , con el formato:

   $$\mathrm{Var}_i^t \in \{\mathrm{PRK}, \mathrm{STD}, \mathrm{SPC}\}$$

   •Por ejemplo, si hay 5 franjas horarias y 3 aviones, habrá 5*3 = 15 variables.

2. **Dominios:**

   •Cada variable puede tomar valores del conjunto de ubicaciones disponibles: talleres y parkings (PRK , STD , SPC).

   •El dominio son todas las coordenadas de los talleres, estas coordenadas están clasificadas según el tipo de taller

3. **Restricciones:**

Según los requisitos, las restricciones son:

- **<u>Restricción 1:</u>** Cada avión debe tener asignado exactamente un taller o parking en cada franja horaria.

Sea A el conjunto de aviones, T el conjunto de talleres y parkings, y F el conjunto de franjas horarias. La restricción se expresa como:

$$\forall a \in A, \forall f \in F, \exists! t \in T : \text{asignación}(a, f) = t$$

Esto significa que para cada avión “a” y cada franja “f” , debe existir un único “t” (taller o parking) al que el avión esté asignado.

- **Restricción 2:** Hasta 2 aviones por taller por franja horaria, y como máximo 1 avión JUMBO por taller por franja horaria.

Sea $J \subseteq A$ el conjunto de aviones JUMBO $\text{ocupación}(t, f)$ el conjunto de aviones asignados al taller t en la franja f. Entonces:

1. Límite de ocupación:

$$\forall t \in T, \forall f \in F : |\text{ocupación}(t, f)| \leq 2$$

2. Límite de aviones JUMBO:

$$\forall t \in T, \forall f \in F : |\text{ocupación}(t, f) \cap J| \leq 1$$

- **Restricción 3:** Los talleres especialistas (SPC) pueden hacer tareas estándar, pero no al revés.

Sea $T_{\text{STD}} \subseteq T$ el conjunto de talleres estánda $T_{\text{SPC}} \subseteq T$ el conjunto de talleres especialistas. Si $tarea(a)$ denota las tareas asignadas a un avión a :

1. **Un avión con una tarea estándar puede asignarse a T_{STD} o T_{SPC} :**

$$\forall a \in A, \text{tarea}(a) = \text{estándar} \implies \text{asignación}(a, f) \in T_{\text{STD}} \cup T_{\text{SPC}}$$

2. **Un avión con una tarea especialista sólo puede asignarse a T_{SPC} :**

$$\forall a \in A, \text{tarea}(a) = \text{especialista} \implies \text{asignación}(a, f) \in T_{\text{SPC}}$$

- **Restricción 4:** Las tareas de tipo 2 deben completarse antes que las de tipo 1. Si el avión tiene atributo T.

Sea $tareas_2(a)$ y $tareas_1(a)$ el número de tareas de tipo 2 y tipo 1 pendientes de un avión a , respectivamente. Sea $A_T \subseteq A$ el conjunto de aviones que tienen el atributo T . Entonces:

1. **Un avión con tareas de tipo 2 pendientes no puede realizar tareas de tipo 1:**
   - En cualquier franja horaria f , si $a \in A_T$ y tiene tareas de tipo 2 pendientes

$$\forall a \in A, \forall f \in F: tareas_2(a) > 0 \Rightarrow asignación(a, f) \in T_{SPC}$$

2. **Todas las tareas de tipo 2 deben completarse antes de empezar las de tipo 1:**

- Un avión con atributo T no puede iniciar tareas de tipo 1 hasta que todas sus tareas de tipo 2 hayan sido completadas en todas las franjas horarias:

$$\forall a \in A_T : \sum \text{tareas}_2(a, f) = 0 \implies \text{tareas}_1(a) \text{ comienza.}$$

- **<u>Restricción</u> 5:** Adyacencia libre para maniobrabilidad. Al menos un taller o parking adyacente a un avión debe estar vacío.

Sea $adyacente(t_1, t_2)$ una función que verifica si t_1 y t_2 son talleres o parkings adyacentes. Para cualquier taller o parking t , debe haber al menos un taller o parking adyacente vacío en cada franja horaria:

$$\forall t \in T, \forall f \in F, \text{ocupación}(t, f) \neq \emptyset \implies \exists t\prime \in T : \text{adyacente}(t, t\prime) \land \text{ocupación}(t\prime, f) = \emptyset$$

- **<u>Restricción 6:</u>** Los aviones JUMBO no pueden estar en talleres adyacentes en la misma franja horaria.

Si $a_1, a_2 \in J$ (dos aviones JUMBO), y $asignación(a_1, f)$ devuelve el taller o parking asignado al avión a en la franja f , entonces:

$$\forall a_1, a_2 \in J, \forall f \in F : \text{adyacente}(\text{asignación}(a_1, f), \text{asignación}(a_2, f)) \implies a_1 = a_2$$

DECISIONES DE DISEÑO

En el diseño de este código, se han tomado varias decisiones clave para garantizar modularidad, claridad y eficiencia, siempre respetando el uso de python-constraint como prerrequisito. A continuación, se detallan los aspectos más destacados:

Representación de entidades

Se decidió crear una clase llamada *Avion* para encapsular los atributos importantes de los aviones, como su ID, tipo, restricciones y tareas de mantenimiento. Esta decisión permite una organización más clara del código y facilita futuras modificaciones o extensiones, como agregar nuevos atributos o reglas específicas. Al tratar a cada avión como un objeto, se mejora la legibilidad y se simplifica la gestión en las restricciones.

Estructura del problema CSP

Para modelar las variables del CSP, se optó por asignar identificadores únicos a cada avión en cada franja horaria, siguiendo el formato "<*avion*>_<*franja*>". Esto facilita la identificación y manipulación de las variables. Además, se agruparon estas variables en diccionarios por franjas horarias (*dicc_var*) y por aviones (*dicc_por_avion*), lo que permite aplicar restricciones específicas de forma eficiente y organizada.

Modularidad en las restricciones

Se eligió encapsular cada restricción en funciones independientes, como *no_mas_de_2*, adyacentes y *tareas2_completadas*. Esto garantiza un diseño modular, donde cada restricción es fácilmente comprensible y ajustable. Por ejemplo, adyacentes no sólo valida la maniobrabilidad, sino que también optimiza las verificaciones utilizando conjuntos para agilizar las búsquedas y validaciones.

Tratamiento especial para los aviones JUMBO

Los aviones JUMBO tienen un manejo específico, como limitar su asignación a un taller por franja horaria y evitar posiciones adyacentes con otros JUMBO. Estas restricciones se implementaron de forma separada para cumplir con las restricciones especificas.

Optimización en el manejo de tareas

Para cumplir con la regla de que las tareas de tipo 2 deben completarse antes que las de tipo 1, se emplearon las funciones *tareas2_completadas* y *tareas1_completadas*. Estas validaciones utilizan bucles optimizados y operaciones con conjuntos para minimizar el tiempo de cálculo, lo que asegura un rendimiento aceptable incluso con múltiples aviones y franjas horarias.

Validación de entrada y procesamiento inicial

Se ha creado la función lector para separar la lógica de lectura del archivo de entrada y garantizar que los datos se procesen correctamente. Además, al estar las ubicaciones de talleres y parkings en conjuntos (*set*), las operaciones de búsqueda son más rápidas, especialmente en restricciones como *adyacentes* y *tareas2_completadas*.

Formateo de resultados

Para presentar las soluciones de manera clara y legible, se ha decidido implementar la función *format_solution*. Esta función organiza las salidas para que sea sencillo interpretar cómo se asignan los aviones a talleres o parkings en cada franja horaria.

Eficiencia en la aplicación de restricciones

Se ha priorizado la eficiencia en varias áreas del diseño:

1. Se han utilizado estructuras de datos como diccionarios y conjuntos para reducir la complejidad computacional en la validación de restricciones.
2. Se han definido las restricciones de forma específica, aplicándose únicamente a las franjas y aviones relevantes.
3. Se han procesado las soluciones de forma incremental, obteniendo una solución inicial antes de explorar otras posibles soluciones.

Reflexión general

El diseño del código está pensado para ser modular, claro y eficiente. La representación de entidades, la separación de restricciones y el uso de estructuras de datos adecuadas permiten cumplir con los requisitos del problema y mantener la flexibilidad para futuros ajustes. Aunque python-constraint es un prerrequisito, el diseño garantiza que su uso esté bien integrado y que el modelo sea comprensible y funcional.

ANÁLISIS DE RESULTADOS

El algoritmo basado en **CSP** consigue todas las soluciones correctamente cuando los inputs son pequeños, ya que el espacio de búsqueda es manejable. Sin embargo, cuando el tamaño del input aumenta, como en mapas más grandes o con más aviones, el tiempo de ejecución crece significativamente. Esto ocurre porque el número de posibles combinaciones de variables y restricciones se incrementa exponencialmente, lo que hace que el algoritmo tarde más en encontrar una solución.

PARTE 2

El problema de rodaje de aviones se puede modelar como un problema de búsqueda en el que se busca minimizar el makespan, es decir, el tiempo total que tardan todos los aviones en llegar a sus respectivas pistas de despegue, por supuesto evitando choques etc…

MODELO

Las distintas casillas del mapa representan un Nodo en nuestro código, estos nodos son distintos según quien los haya generado, es decir, según el camino que se esté midiendo, una misma coordenada podría tener un predecesor distinto, un peso distinto o un coste heurístico distinto.
El problema se ha modelado como un algoritmo A* multiagente, donde elegimos el próximo desplazamiento restringiendo los nodos ya ocupados por otros aviones en ese mismo instante.
Nuestro principal objetivo es minimizar el makespan, (el número de movimientos y esperas)
El espacio de estados del problema lo he definido por las combinaciones posibles de:

- Las posiciones (x, y) de los aviones en el mapa.
- El tiempo en que un avión llega a una posición.
- Las celdas ocupadas por otros aviones y los obstáculos en el mapa.

DECISIONES DE DISEÑO

- **Clase *Context*:**

La clase Context está pensada para manejar toda la información que necesitamos para correr A*. La idea es agrupar el mapa, los obstáculos y los nodos reservados en un solo objeto, lo que hace que pasar todos estos datos a las funciones sea mucho más sencillo. En vez de tener que pasar un montón de parámetros, con solo pasar este objeto conseguimos acceso a todo lo necesario. De este modo, además de que el código se vuelve más limpio y más fácil de gestionar, evitamos errores por pasar mal los datos.

- **Clase *Nodo*:**

La clase Nodo es clave en A*. Cada nodo tiene las coordenadas del avión en un momento dado, junto con el coste acumulado (g), la heurística estimada (h) y la suma de ambos (f), que es lo que usamos para decidir qué nodo procesar primero. También tiene un parent para poder reconstruir el camino una vez lleguemos al objetivo. Si no tuviéramos esta clase, sería mucho más complicado gestionar y comparar nodos, lo que haría que el algoritmo fuera menos eficiente.

- **Clase *MinHeap*:**

La clase MinHeap es una implementación de un heap mínimo, que necesitamos para que A* funcione bien. El heap nos permite siempre acceder al nodo con el coste más bajo (f) de manera eficiente, lo que es esencial para que A* expanda los nodos más prometedores primero. Sin el heap, tendríamos que hacer una búsqueda lineal en cada paso para encontrar el siguiente nodo a procesar, lo cual sería mucho más lento y menos eficiente.

- **Código Modular:**

El código está dividido en varias funciones que se encargan de una cosa cada una, lo cual hace que sea más fácil de entender y de modificar. Cada función tiene un objetivo claro, como calcular la heurística, expandir un nodo o reconstruir el camino. Esta modularización también facilita el testeo y la depuración, ya que podemos comprobar que cada parte funciona por separado sin tener que preocuparnos de todo el sistema al mismo tiempo.

- **Heurísticas:**

    - **Manhattan:** Es una forma de calcular la distancia que solo tiene en cuenta los movimientos horizontales y verticales. Es rápida y sencilla, pero no siempre captura la complejidad real del problema, ya que no toma en cuenta obstáculos o tiempos de espera.

    - **Euclídes:** Esta heurística calcula la distancia en línea recta entre dos puntos. Es más precisa que Manhattan, pero puede ser menos útil en un entorno de cuadrícula, ya que los aviones no se pueden mover en direcciones diagonales.

    - **Heurística Propia:** Esta es la que considero más útil para este problema. Toma la distancia de Manhattan, pero la ajusta con penalizaciones por los obstáculos que rodean al nodo a explorar y al objetivo (ya que son pasos extra al tener que esquivar). Esto hace que la heurística sea más realista, ya que no solo

tiene en cuenta la distancia directa, sino también las restricciones que surgen cuando los aviones deben evitar obstáculos.

- **Código Multiagente:**

En este tipo de problemas con múltiples aviones, uno de los desafíos es asegurarse de que no haya colisiones. Para hacerlo, utilizamos un conjunto de reserved donde guardamos las celdas que ya están ocupadas o las rutas que ya han sido tomadas. Además de almacenar las celdas ocupadas, guardamos los bordes entre celdas adyacentes, lo que nos permite garantizar que los aviones no se crucen en los caminos. De esta manera, conseguimos que los aviones no colisionen mientras siguen su ruta.

ANÁLISIS DE RESULTADOS

He notado que la heurística de **Manhattan** genera más tiempo de ejecución (makespan) porque no tiene en cuenta los tiempos de espera, que es algo clave cuando hay aviones que tienen que esperar para evitar colisiones. En cambio, mi heurística propia es más precisa, ya que penaliza el tiempo de espera, lo que hace que el cálculo sea más realista.
Al comparar el rendimiento de las heurísticas, el tiempo de ejecución no es lineal. Por ejemplo, cuando el mapa tiene 20 aviones y un tamaño de 20x20 casillas, el algoritmo tarda 0.078 segundos y expande 4362 nodos. Sin embargo, en un mapa de 52x36 con 36 aviones, el tiempo de ejecución se incrementa a 0.69 segundos, lo que muestra que el tiempo crece significativamente con el tamaño del mapa y el número de aviones.

CONCLUSIONES

A pesar de que este trabajo estaba pensado para hacerlo entre dos personas, lo he realizado yo solo por ciertas complicaciones. Este proyecto me ha permitido entender mucho mejor cómo funcionan las heurísticas y cómo optimizar el código para hacerlo más eficiente. Además, este proyecto me ha dado una comprensión más profunda sobre cómo se pueden usar técnicas de búsqueda heurística para resolver problemas complejos en entornos reales. Ha sido un reto, pero también una gran oportunidad para mejorar mis habilidades en programación y en optimización de algoritmos.