Práctica 2: Satisfacción de Restricciones y Búsqueda Heurística
Heurísticas y Optimización.
Grado de Ingeniería en Informática. Curso 2022-2023 ´
Planning and Learning Group
Departamento de Informatica

Alumnos:
Alejandro García Berrocal - 100451059
Rubén Vecino Garzón - 100428968

Github: https://github.com/AlejandroGB-9/practica-2-heuristica-gr84-451059-428968

Indice

Introducción

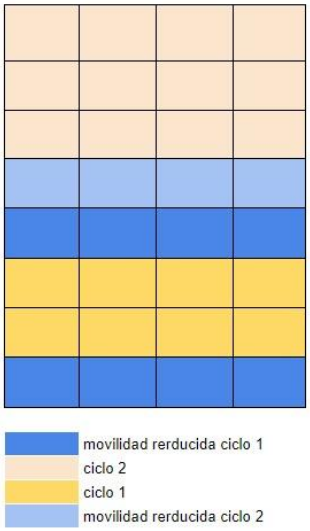
Se mostrarán las ideas que se han llegado a desarrollar para las dos partes y de qué manera se han modelizado estas. Se explicarán qué son las partes que hacen que sea un problema CSP y un algoritmo de búsqueda además de los estados, variables, dominios y restricciones que se han considerado en ambas partes.

Siguiendo se mostrará un análisis de los resultados y una breve conclusión sobre el proyecto y algunas mejoras que se podrían haber tenido en cuanto por nuestra parte o por el profesorado al presentar este proyecto.

Parte 1: Validación con Python constraint

Descripción del modelo.

Antes de empezar a definir los dominios y las variables, se ha decidido transformar el autobús en una matriz e invertirla. De esta manera se pasa de trabajar con una matriz de 4x8 a una de 8x4. Se ha de mencionar que para la matriz no se incluyen los espacios de pasillos, puertas u otros asientos que no pueda tomar un alumno pero si se toma en cuenta el pasillo en algunas de las restricciones que se mencionan más adelante.



Los elementos que se necesitan para definir un problema CSP son una de red de restricciones $R = \{X, D, C\}$ que consiste en un conjunto finito de valores $X = \{X_i\}_{i=1}^{n}$ que se definen sobre unos dominios que contienen los valores posibles a tomar por cada variable $D = \{D_i\}_{i=1}^{n}$ y un conjunto de restricciones aplicadas al problema $C = \{C_i\}_{i=1}^{n}$.

Según el caso de alumnos se puede dar un número de variables de 1 a 32, por lo que no se pueden definir un número fijo de variables. Por lo que n puede tomar los valores anteriormente mencionados.

Las variables pueden ser las siguientes:

$X = \{AL_{i[C1]}, AL_{i[C2]}, AL_{i[MRC1]}, AL_{i[MRC2]}\}$

Dadas estas variables se definen los dominios en relación a estas categorías. Esto es, C# representa el ciclo al que pertenece el alumno, mientras que si tiene MR delante implica que ese alumno adopta los asientos de movilidad reducida. Al ser de movilidad reducida se tienen que tener disponibles asientos destinados especialmente para estos alumnos según el ciclo al que pertenecen.

$D = \{C1,\ C2,\ MRC1,\ MRC2\}$

$C1: \{0 - 3,\ 0 - 3\},\ C2: \{4 - 7\},\ MRC1: \{0\ o\ 3,\ 0 - 3\},\ MRC2: \{4,\ 0 - 3\}$

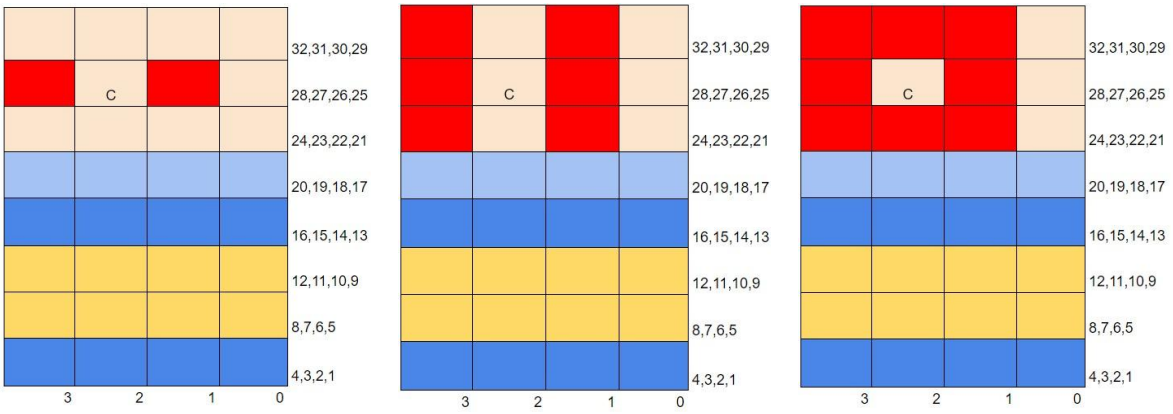
La definición de estos dominios se hace en relación a la fila y columna, $\{fila,\ columna\}$ donde se sitúe el asiento en el autobús. En caso de duda véase la representación al principio de la parte 1.

Ahora se consideran las restricciones del problema en base al enunciado. Estas restricciones validarán que un alumno pueda tomar asiento respecto a otro, sea común, conflictivo, de movilidad reducida o un hermano.

Dado un asiento del autobús $a_{i[s]} \in D_{i[s]}$ para un alumno i con un dominio $[s = \{fila,\ columna\}]$ y asiento del autobús $a_{i'[s']} \in D_{i'[s']}$ para otro alumno i', se aplica la restricción $R_x = \{(a_{i[s]}, a_{i'[s']})\}$. La restricción determinará aquellos valores o asientos que puedan ocupar dichos alumnos. Las restricciones definidas son:

Para los alumnos conflictivos con otros conflictivos o de movilidad reducida:

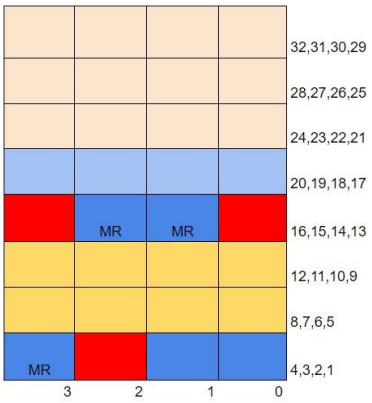
1. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) > 1$
2. $absolute(AL_{1[s]fila} - AL_{2[s]fila}) > 1$
3. $absolute(AL_{1[s]fila} - AL_{2[s]fila}) == 1 \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) > 1$



Para los alumnos con movilidad reducida con otro de movilidad reducida o común:

1. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) == 1 \wedge$
$((AL_{1[s]columna} == 1 \wedge AL_{2[s]columna} == 2) \vee (AL_{1[s]columna} == 2 \wedge AL_{2[s]columna} == 1))$

2. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) > 1$
3. $absolute(AL_{1[s]fila} - AL_{2[s]fila}) > 0$

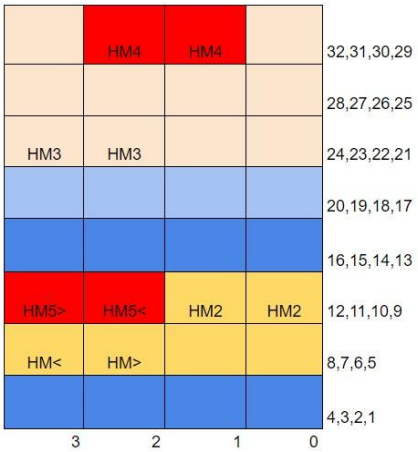


Para los alumnos que son hermanos del mismo ciclo y no son de movilidad reducida ninguno:

1. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) == 1 \wedge$
$((AL_{1[s]columna} == 1 \wedge AL_{2[s]columna} != 2) \vee (AL_{1[s]columna} == 2 \wedge AL_{2[s]columna} != 1))$
2. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) == 1 \wedge$
$((AL_{2[s]columna} == 1 \wedge AL_{1[s]columna} != 2) \vee (AL_{2[s]columna} == 2 \wedge AL_{1[s]columna} != 1))$

Para los alumnos que son hermanos de distinto ciclo y no son de movilidad reducida ninguno:

1. $AL_{1[s]fila} == AL_{2[s]fila} \wedge absolute(AL_{1[s]columna} - AL_{2[s]columna}) == 1 \wedge$
$((AL_{1[s]columna} != 1 \wedge AL_{1[s]columna} != 2) \wedge (AL_{2[s]columna} == 1 \wedge AL_{2[s]columna} == 2))$



Análisis de los resultados

Análisis archivo 1 (alumnos01.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 8 alumnos, estos son 2 hermanos de distintos ciclos, 2 conflictivos de 2 ciclos distintos , 1 conflictivo y de movilidad reducida y los demás comunes.

El número de posibles soluciones son de 628992. Se trabaja con todas las restricciones del problema excepto la de los hermanos que son del mismo ciclo.

Output: {'1XC': 8, '2XX': 17, '3XX': 7, '4RC': 18, '5XX': 10, '6XX': 15, '7XC': 28, '8RX': 20}

Análisis archivo 2 (alumnos02.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 10 alumnos, estos son 6 alumnos de movilidad reducida para el ciclo 1 y 2 de movilidad reducida para el ciclo 2, los demás alumnos son comunes.

El número de posibles soluciones son 0 debido a que es un caso imposible . El número de alumnos de movilidad reducida excede el número de asientos disponibles destinados a estos alumnos para este ciclo.

Análisis archivo 3 (alumnos03.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 7 alumnos, estos son 4 alumnos de movilidad reducida del ciclo 2 y los demás alumnos son comunes.

El número de posibles soluciones son 0 debido a que es un caso imposible . Al igual que en el anterior, el número de alumnos de movilidad reducida excede el número de asientos disponibles destinados a estos alumnos para este ciclo.

Análisis archivo 4 (alumnos04.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 9 alumnos, estos son 6 hermanos ambos de movilidad reducida, conflictivos del ciclo 2 y 2 del mismo ciclo. Por otro lado, hay uno más conflictivo del ciclo 2 y los demás comunes.

El número de posibles soluciones son de 4138560. Se trabaja con todas las restricciones del problema.

Output: {'1XX': 14, '2XC': 31, '3RX': 1, '4RX': 3, '5XX': 16, '6XC': 17, '7XX': 6, '8XC': 18, '9XX': 5}

Análisis archivo 5 (alumnos05.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 7 alumnos, estos son 2 hermanos y uno de movilidad reducida.

El número de posibles soluciones son de 2081520. Se trabaja con todas las restricciones del problema.

Output: {'1XX': 27, '2XC': 22, '3XX': 5, '4RX': 16, '5XX': 30, '6XX': 10, '7XX': 6}

Análisis archivo 6 (alumnos06.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 11 alumnos, estos son 6 hermanos 2 conflictivos, 2 de distinto ciclo y uno que es de movilidad reducida pero el otro es común. Por otro lado, 1 de movilidad reducida y el resto comunes.

Se trabajan con todas las restricciones del problema excepto la de los hermanos que son del mismo ciclo.

Análisis archivo 7 (alumnos07.txt)

En esta configuración, se estudia un caso en el que nos encontramos con 8 alumnos, estos son 2 hermanos de distintos ciclos y uno conflictivo, 3 conflictivos y 1 conflictivo y de movilidad reducida.

Se trabajan con todas las restricciones del problema excepto la de los hermanos que son del mismo ciclo.

Conclusión de los análisis

Se ha comprobado que funciona de manera correcta las restricciones desarrolladas para estos casos a probar. Además durante la selección de casos a probar nos hemos dado cuenta que ha medida de que se añadían más alumnos al problema el número de soluciones incrementaban exponencialmente dando a tiempos de ejecución muy altos. Claro está, que estas soluciones y tiempos de ejecución varían según las características de los alumnos del problema. Si nos encontramos con menos alumnos comunes, las soluciones decaen al igual que los tiempos de ejecución, de la misma manera pasaría al contrario.

Por otro lado, se ve que de las restricciones que más restringen es la de conflictivos ya que se compara a nivel de conflictivos y movilidad reducida.

Por razones de última hora, el código empezó a funcionar de distinta manera por lo que solo los tests del 01 al 05 se incluirán en el *.sh,* se ha de mencionar que antes funcionaban correctamente y daban una solución sólida.

Parte 2: Planificación con Búsqueda Heurística

Clase Nodo.

Se ha visto necesario el crear otro archivo *.py* para crear una clase nodo. Esta clase nodo tiene la información relevante para crear e identificar un nodo. La información que recoge esta clase es el estado del nodo, su padre y los costes g, h y f. Esta clase es de importancia para hacer uso del coste e identificar el padre del nodo en funciones auxiliares que se han creado.

Algoritmo A estrella.

Para este algoritmo se ha de definir los estados inicial y final. El estado inicial es una lista vacía sin ningún alumno introducido a lo que llamaremos cola. El estado final será aquel que tenga una longitud igual a los alumnos del fichero de entrada o a considerar en el problema.

Para representar los estados se ha optado por hacerlo a través de una lista que contenga los alumnos y cada nodo cuando se expanda añadirá un alumno que no tenga ya dentro del estado.

El algoritmo comienza con una lista como nodo inicial con costes ceros. Por otro lado se crean las listas abierta y cerrada para almacenar los hijos creados y los nodos expandidos en la lista que corresponda. Inicialmente el nodo con el que se comienza, cola vacía, se añade a la lista abierta.

Para empezar a expandir nodos y obtener el nodo meta es necesario crear un bucle *while* con las condiciones de que la lista abierta no esté vacía y que no se haya alcanzado el nodo meta.

Se tomará el primer nodo de la lista abierta y ordenada para expandirlo. Si el nodo ya estuviese expandido, es decir, ya estaría en la lista cerrada se continuaría al siguiente nodo. En caso contrario, el nodo se añade a la lista cerrada con los demás nodos expandidos y se obtendrían sus hijos los cuales se añadirían a la lista abierta.

Una vez se encontrase el nodo meta, este nodo se buscaría en la lista abierta y se devolvería el nodo, su coste total y el total de nodos expandidos.

Funciones auxiliares al algoritmo.

Este apartado será en referencia a las funciones auxiliares que se han necesitado para el desarrollo del algoritmo A*. El total de funciones auxiliares creadas han sido de 9.

La primera función ***orderListAlumnos()*** tiene la funcionalidad de ordenar los alumnos según su asiento de menor a mayor. Esto es ya que se mencionaba que la lista de los alumnos debía estar ordenada bajo este criterio y así utilizarse en el algoritmo.

La segunda función ***orderOpenList()*** tiene la función de ordenar la lista abierta de los nodos expandidos según el casto de menor a mayor. Para ello se crea una lista para introducir los costes de los nodos de la lista abierta. Se hace un *sort()* de esa lista y una vez ordenados de menor a mayor se buscan aquellos nodos que tengan ese mismo coste. Esos nodos se añaden a una nueva lista abierta. No es necesario preocuparse por el orden de aquellos nodos con mismo coste ya que se acabarán expandiendo uno detrás de otro. Una vez se introduzcan los nodos en orden ascendente se devolverá como la lista abierta a utilizar.

La tercera función es una de las más relevantes en el algoritmo de A*, esta función se llama ***nodeChildren()***. Como su nombre indica esta función es la que se encarga de crear los nodos hijos del nodo expandido. Dentro de esta función se escogerá un alumno de la lista de alumnos, si este alumno ya se encuentra dentro del nodo expandido o el alumno que se va a introducir es de movilidad reducida e irá detrás de otro alumno de la misma condición, el alumno no se introducirá para formar un estado hijo. Dado otro caso distinto a este, se formarán todos los nodos hijos posibles partiendo del nodo expandido y se clasificarán como un nodo. Esto es, su nodo padre será el nodo expandido, el estado del nodo será el nodo padre más el alumno introducido a la cola y se calcularán sus respectivos costes con unas funciones auxiliares que se explicarán más adelante. Por último, se introducen estos nodos hijos dentro de la lista abierta, la cual se ordenará según los costes con la función anterior.

La cuarta función, ***calculateNodeCosts()*** se encarga de poner el coste del nodo hijo proveniente de la función anterior. Para los costes g y h se necesita el uso de dos funciones que se dedican a calcular los estos dos costes del nodo. Por otro lado, el coste f se calcula una vez se obtengan los dos costes anteriores.

La quinta función ***calculateHCost()*** se encarga de calcular el valor de la heurística según dos heurísticas definidas. La primera heurística definida que se ha decidido implementar es el número de asientos restantes hasta alcanzar el nodo meta. Para esto se resta el número de alumnos de la lista y el número de alumnos que ya se encuentran en la cola. Si resultado de la diferencia es 0 significaría que se ha encontrado el nodo meta y los valores que se devolverían sería el coste de la heurística de la iteración y se devolvería *True* o *False* según se haya encontrado el nodo meta o no. La segunda heurística no se ha podido implementar.

La función sexta ***calculateGCost()*** calcula el coste g del nodo. Para ello primero se ha de saber si encontramos alumnos conflictivos en la cola ya que la resolución del cálculo es totalmente distinta. En caso de no haber alumnos conflictivos primero se comprobará si es

el primer alumno de la cola, lo que indica que el padre del nodo es una lista vacía, y si es el primero se le añade el coste correspondiente según el tipo de alumno. Si no es el primero de la cola, se comprobará el alumno que se encuentra enfrente del que se va a introducir. El caso más importante para este caso es que el alumno que se encuentre enfrente sea de movilidad reducida. Esto es porque se ha decidido que los alumnos con movilidad reducida no generan coste sino los alumnos que los ayudan a subir, siendo 3 como se mencionaba en el enunciado de la práctica al subir los dos alumnos simultáneamente.

Ahora, dado el caso que haya alumnos conflictivos en la cola se hace uso de otra función para calcular este coste. Esta función se llama ***calculateAllCost().*** Como su nombre indica calcula el coste del todo el nodo, es decir, de todos los alumnos de la cola. Esto es porque un alumno conflictivo influye sobre la mayoría de los alumnos de la cola ya que retrasa a los que tenga enfrente y detrás suya y los que tengan asientos mayores al suyo. Hay casos muy particulares dada esta condición y es por eso que se necesita de esta función que funciona en base a múltiples condicionantes dado el caso encontrado en un alumno. Todos los casos se aprecian más en el código fuente comentados. Poniendo un ejemplo pongamos que tenemos un alumno común (‘XX’) y que enfrente tenga a un alumno de movilidad reducida (‘XR’) que no sólo eso sino también es conflictivo (‘CR’), como este no genera coste como se ha mencionado anteriormente es necesario saber el alumno que se encuentra enfrente de este y se da el caso que también es conflictivo (‘CX’). Por otro lado, cogemos el alumno, si hay, detrás del que hemos partido y resulta que también es conflictivo, esto nos deja con esta sección de la cola, ‘CX ,CR ,XX, CX’, el coste de este alumno será de 3 (movilidad reducida) * 2 (conflictivo enfrente de movilidad reducida) * 2 (conflictivo movilidad reducida) * 2 (conflictivo detrás del alumno) = 24.

La octava función ***searchGoalNode()*** busca el nodo meta una vez se envíe la señal *True* de que se ha encontrado dicho nodo. Se buscará el nodo por la lista abierta este nodo meta al haberse expandido un nodo que resultase en ese como hijo. Para encontrarlo, se busca el estado que tenga una longitud igual al número de alumnos del autobús.

La última función y ajena al algoritmo es ***transformGoalNode(),*** su función es transformar la cola de alumnos obtenida como resultado del algoritmo en un diccionario cuyas claves serán los alumnos de la cola en el orden obtenido y su asiento en el bus como valor de la clave.

Análisis de los resultados

Los test cases se han realizado en base a distintas longitudes del nodo que se obtendría de la parte 1. Se ha de comentar que unos de los puntos para calcular el coste no se ha cumplido debido a la complejidad de este. Este punto es el que hace referencia a que un conflicto hará que se retrasen todos aquellos alumnos cuyo asiento sea mayor al de este. En otras palabras, que se coloquen detrás de el alumno conflictivo.

Por otro lado, se ha de dar cuenta que la longitud del plan no está implementada por lo que se mostrará un mensaje que dice eso. Además se ha de comentar que los costes se creen estar totalmente bien en muchos de los casos ,pero un posible ejemplo que sea complejo puede que los costes no sean los adecuados por lo que se ha comentado anteriormente.

En cuanto a las heurísticas sólo se han implementado para uno ya que no hemos podido desarrollar otra.

Finalmente, para los únicos casos que se han llegado a desarrollar ha de decirse que ha cumplido con las restricciones de que no puedan estar dos alumnos de movilidad reducida uno detrás de otro o al final.

Conclusiones

El trabajo ha sido de muchísima ayuda para poder entender CSP y aprender a modelarlo partiendo de los conocimientos obtenidos durante el curso ante un problema como el presentado. Por otro lado, aprender a implementar un algoritmo de búsqueda ha sido un tanto enrevesado al tener que incluir una clase nodo. Pero se ha de decir que el determinar una heurística ha sido de lo más complejo, tanto que no se ha podido desarrollar una segunda.

En cuanto a la parte 1 se ha mencionado que ha sido divertido y lo es, pero nos hemos encontrado el problema con asignar las restricciones que es lo que más dificultad ha tenido porque al tener cualquier mínimo error podría variar considerablemente.

Para la segunda parte más de lo mismo, lo más difícil era la heurística y el poder determinar los costes. El expandir un nodo ha sido de lo más sencillo.

Es un trabajo muy divertido de hacer en cuanto a las dos partes. Hubiera sido mucho mejor el poder haber tenido mucho más tiempo para desarrollarlo debido a los horarios y trabajos de las otras muchas asignaturas.

Nuestro código no es el mejor y se podría llegar a optimizar mucho más. Esto es ahorrarse ciertos bucles, listas u otras operaciones que se consideraron necesarias al implementarse. Pero debido al poco tiempo que se ha tenido para realizar ambas partes no se han podido aplicar mejoras a ambos códigos puesto que algunos están por terminar.

Como último punto, estaría muy bien que se pudiese dar un poco más de práctica y explicaciones en cuanto al funcionamiento de las librerías de python-constraint ya que al principio costaba un poco, pero más importante es la segunda parte. Esta parte no se ha llegado a explicar lo suficiente en las clases magistrales por lo que ha dificultado mucho el saber cómo desarrollar el algoritmo y las heurísticas.

Práctica 2: Satisfacción de Restricciones y Búsqueda Heurística
Heurísticas y Optimización.
Grado de Ingeniería en Informática. Curso 2022-2023 ´
Planning and Learning Group
Departamento de Informatica

Alumnos:
Alejandro García Berrocal - 100451059
Rubén Vecino Garzón - 100428968

Github: https://github.com/AlejandroGB-9/practica-2-heuristica-gr84-451059-428968