ESTRUCTURAS DE DATOS Y ALGORITMOS USADOS:

ESTRUCTURAS DE DATOS:

1. ESTRUCTURAS BÁSICAS PARA EL ALGORITMO:

En primera instancia voy a explicar la razón por la que hemos elegido las estructuras de datos más significativas en nuestro programa.
En una gran cantidad de casos hemos usado HashMaps. Los casos más relevantes es la estructura de datos que mantiene el plan de estudios con todas sus asignaturas. El motivo de nuestra elección no es otro que poder mantener un tiempo de acceso constante en los casos en el que lo considerábamos de gran importancia. De esta manera, por ejemplo, a la hora de asignar correquisitos a las asignaturas, podemos buscarlo en el map por su id. Además usamos una implementación de map en el que no tenga en cuenta el orden, pues en nuestro caso no es de ninguna relevancia. Usaremos esta misma estructura para almacenar por ejemplo, nuestro conjunto de Asignaciones.

La otra estructura de datos que puede que sea la más relevante es la que le permite a cada Asignación mantener su conjunto con todas las posibilidades que tiene (Classes). En este caso teníamos claro que necesitábamos también el uso de maps ya que necesitábamos hacer muchos accesos muy concretos a la hora de hacer la poda del forward checking. Por este motivo decidimos usar la estructura de datos:
Map, Map<DiaSetmana, ArrayList<Classe>>> possibles_classes;

De esta manera mantenemos todas las Classes ordenadas por id_aula. Una vez en el contexto de una aula, tendría las 5 opciones que representan los 5 días de la semana. Finalmente en una aula, en un día concreto, decidimos usar una ArrayList para almacenar propiamente las posibilidades ya que el numero que puede contener es muy limitado (de hecho constante), por lo que hacer un recorrido allí tiene un coste asimptóticamente negligible. De esta manera, para mantener por ejemplo, la restricción de ocupación solamente tendremos que hacer 2 accesos para seleccionar el aula y el día, y después hacer un recorrido de tiempo constante para eliminar las que ya no son posibles. De la misma manera procedemos con las demás restricciones, ahorrándonos hacer accesos innecesarios.

Otro estructura importante, será una linked-list que mantenemos dentro de la clase Horario. Dentro de ella, cada vez que el horario haga una elección, añadiremos la clase allí. En un principio habíamos puesto estas estructuras de datos en cada una de las propias asignaciones, cosa que la final resulto ser muy ineficiente. De este modo, lo modificamos para mantener una sola estructura centralizada.

2. ESTRUCTURAS PARA MANTENER LAS RESTRICCIONES:

Mantenemos dos diccionarios importantes dentro de nuestra clase Plan de Estudio. Ellas son las de las restricciones modificables, pues las que no lo son están incrustadas directamente en el código de cada una de las asignaciones en particular.

Tenemos pues dos estructuras diferentes, una de las cuales mantendrá nuestra conjunto total de restricciones modificables para dicho plan de estudio. Por otro lado, la otra, nos representará el subconjunto de el diccionario anterior, que usaremos para nuestro algoritmo. De esta manera somos capaces de diferenciar entre las que consideramos "activas" y las que simplemente existen en nuestro plan de estudios.
Hemos usado un diccionario y no por ejemplo una lista por un motivo muy sencillo. Queremos dotar a la interfaz de la capacidad de eliminar elementos de ambos diccionarios para dar flexibilidad al usuario. Por ese motivo, la forma más eficiente es usarlos, y a partir de una clave, poder encontrar en tiempo constante la restricción que queremos eliminar.

Solo una de estas estructuras estará replicada en la clase Horario. Hemos añadido en el constructor de este, el diccionario de las restricciones activas, pues este lo necesitará para hacer la correspondiente poda.

Las demás restricciones como la de ocupación, subgrupo o la de correquisito esta codificada dentro de cada una de las asignaciones en forma de clase Restricción por lo que no hace falta una estructura de datos que las contenga todas.

ALGORITMO DE FORWARD CHECKING:

Por lo que al algoritmo se refiere, hemos usado la técnica del Forward checking para optimizar nuestro backtracking.

Aun así, antes de empezar el proceso del algoritmo, hemos realizado lo que hemos nombrado preprocesado. En este preprocesado, gracias a que el horari contiene un map con todas las restricciones modificables, haremos una primera poda en la que eliminaremos todas la posibilidades de todas las asignaciones que incumplan alguna de ellas. El motivo de no hacer este proceso durante el algoritmo es que en ningún caso, las Clases eliminadas en esta parte pueden ser recuperables. La decisión de no usarlas es irreversible independientemente si el algoritmo acaba o no.
Una vez finalizada la poda inicial, ahora sí, empezaremos el proceso que constituye el propio algoritmo de forward checking.

Entonces, primeramente el algoritmo empezará, de forma indistinta, por la primera asignación que tenga guardada (que puede ser cualquiera). Entonces cargará todas sus posibilidades en aquel momento factibles y empieza a iterar sobre estas.
Por eso, seleccionará la primera y procede a hacer el forward checking con las demás. El forward checking lo tiene implementado cada clase asignación, simplemente llamando a funciones implementadas en las subclases que hemos creado de restricción.
Entonces, la clase horario (donde se implementa el algoritmo) llamará para todas las asignaciones, que hagan forward checking y les pasará la clase que acaba de seleccionar para que la usen para hacer la poda. Pero es que además, el forward checking no tan solo hace la poda sino que además iremos recopilando todas las clases que hemos podado. El motivo es que

una vez hemos seleccionado una clase, tendremos que comprobar si esa elección aún nos mantiene con posibilidades de generar un horario correcto. En este caso es trivial. Dado que al final, deberemos tener en nuestro horario todas las asignaciones, debemos comprobar que todas las asignaciones aún cuentan por lo menos con suficientes posibilidades (Clase) como para satisfacer el numero de clases que tiene que tener en una semana. Entonces, comprobamos si hay alguna vacía. Si no es así, nos encontramos en dos casos.
En el caso de por ejemplo haber elegido (M1 10 de 8 a 10 en el aula A5001) pero que M1 10 tenga más de una clase en una semana. Por lo tanto necesitamos seguir iterando sobre esta asignación para elegir otra. En este caso hacemos una llamada a la función recursiva pero con el mismo índice. El problema que nos encontramos, fue que para que eso funcionara, en la fase de forward checking también deberíamos añadir en la asignación que estamos tratando, una estructura de datos con las clases que hemos ido eligiendo y eliminar así de las posibilidades restantes la que acabamos de elegir. De esta manera cuando llamemos recursivamente, volverá a iterar y elegir otra opción sobre las demás posibilidades que no hemos elegido aún y son factibles (y así hasta asignar todo el numero de clases que M1 10 tiene que tener en una semana).

En el segundo caso, ya hemos acabado con esa asignación y por lo tanto, simplemente hacemos la llamada recursiva aumentado el índice en 1 y por eso, en la siguiente recursión, comprobando la siguiente asignación.

Todo esto en el caso que sea una Clase válida. Si no fuese así, tenemos que hacer el backtracking. Eso lo hace muy fácil el hecho que la llamada de forward checking no solo pode, sino que además nos retorne lo que ha eliminado. Entonces, dadas todas las Classes que hemos eliminado, tan solo tenemos que volverlas a añadir a su asignación correspondiente (motivo por el cual es necesario que cada clase mantenga la información necesaria para hacer dicha reconstrucción, el id de asignatura y del grupo) y borrar de la asignación actual la Clase seleccionada. Y entonces lo que hará será seguir iterando sobre su conjunto de Classes restante.
Finalmente, tenemos como casos base si llega ha tener un índice mayor al numero de asignaciones, significará que ya tenemos un horario correcto, y en caso contrario y llega al final de código, significará que no existe un horario posible.

ESTRUCTURA Y JERARQUÍA DE LAS CLASSES RESTRICCIONES:

En el algoritmo hemos hecho dos grandes diferenciaciones de las restricciones es función de si las considerábamos modificables o eran innegociables para cada una de las asignaciones. Entonces hemos creado toda una jerarquía en la que de la clase Restricción, cuelgan la clase de las Restricciones Flexibles y las que no los son. Tanto las modificables como las que no lo son compartirán por razones obvias un gran numero de funciones respectivamente, además al implementarlas en forma de interfaz nos asegurábamos que en el código, pudiésemos tratarlas de la misma forma (en función de si son o no modificables) sin preocuparnos de que caso específico se trata.

Este hecho ha sido especialmente importante, en el caso de las restricciones modificables. El motivo es que de esta manera, en la interfaz gráfica, en el menú en el que le damos al usuario la posibilidad de modificarlas, de mostrarlas, eliminarlas... Podemos hacer una implementación genérica y entonces

hemos podido añadir posteriormente tantas restricciones como hemos podido sin tener que cambiar nada.