ESTRUCTURAS DE DATOS Y ALGORITMOS USADOS:

En primera instancia voy a explicar la razón por la que hemos elegido las estructuras de datos mas significativas en nuestro programa.
En una gran cantidad de casos hemos usado HashMaps. Los casos mas 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 mas 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.

Las demás estructuras de datos son poco relevantes por lo que respecta la implementación de las funciones principales pues en muchos casos la ArrayList nos ha permitido almacenar muchos contenidos, los cuales no tenían un interés especial al acceder, o simplemente las accederíamos todas iterativamente (cómo es el caso de classes_seleccionadas donde cada asignación guarda las Classes que el horario le ha seleccionado).

Por lo que al algoritmo se refiere, hemos usado la técnica del Forward checking para optimizar nuestro backtracking. 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 la 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 la asignaciones aún cuentan por lo menos con suficientes

posibilidades (Classe) 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 mas 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 sólo 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.