Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Optimización de la planificación académica con Inteligencia Artificial

Academic scheduling optimization using Artificial Intelligence

David de León Rodríguez

La Laguna, 2 de julio de 2021

D. **José Andrés Moreno Pérez**, con N.I.F. 42.935.437-A, catedrático de universidad adscrito al departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

”*Optimización de la planificación académica con Inteligencia Artificial*”

ha sido realizada bajo su dirección por D. **David de León Rodríguez**, con N.I.F. 54112325H.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de julio de 2021

Agradecimientos

Gracias a las personas que me han acompañado en la carrera, mis amigos del instituto, por supuesto, mi familia, y al tiempo.

Licencia

Resumen

Este trabajo aborda la planificación académica del alumno en la enseñanza universitaria. Se plantea la elección de las asignaturas en las que el alumno se matricula y la elección de grupos de cada una para minimizar los conflictos en los horarios de las asignaturas elegidas. En particular, se presta atención al contexto del Grado en Ingeniería Informática dentro de la Universidad de La Laguna. Se propone una formulación del problema y la obtención de soluciones de alta calidad usando la metaheurística GRASP. Se diseña e implementa los algoritmos correspondientes para aportar soluciones de las versiones de optimización de una función objetivo que engloba los distintos tipos de conflictos y de optimización multi-objetivo al considerar de forma separada los distintos tipos de conflictos. La implementación desarrollada es probada con varias instancias diferentes para obtener resultados con los que se analiza su eficiencia y utilidad.

Palabras clave: Planificación académica, GRASP, metaheurística

Abstract

This work approaches an academic scheduling problem with the use of metaheuristics. The choice of subjects to be enrolled in and their group assignation is proposed to minimize conflicts in the student's time management. In particular this paper works in the context of the Computer Engineering degree in the University of La Laguna. A formulation of the problem is presented with the implementation of a GRASP metaheuristic to search for high quality solutions. The algorithms are designed and implemented first with one objective function that includes vaious types of conflicts in time tables, and then another with a multi-objetive approach considering the different objectives separately. The developed implementation is tested with various instances to render results of its efficiency and utility.

Keywords: Academic scheduling, metaheuristic, GRASP

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

El proceso de matriculación dentro de la Universidad de La Laguna destaca por la opinión prácticamente unánime de todo el alumnado: es complicado y muy poco claro. Este trabajo surge con la intención de aportar una propuesta para paliar este problema.

1.1. Planificación académica

Actualmente el proceso de matriculación consiste en que el alumno elija las asignaturas a matricularse de la lista de asignaturas que le quedan por superar, con posibilidad de elegir los turnos en caso de que el alumno esté en el grupo de alumnos asignados al primer turno. Este proceso tiene varios inconvenientes que se pueden achacar a una perspectiva enfocada en el sistema y no en los componentes que lo conforman, como son el alumnado y su tiempo.

Las problemáticas asociadas a los horarios se repiten cada curso, pudiendo evitarse en el mismo proceso de matriculación, en lugar de a posteriori teniendo que ir a secretaría y pasar un proceso burocrático para pedir cualquier cambio. Ejemplos de los problemas que se dan con el proceso de matriculación actual son el solapamiento de horarios, que afecta gravemente al rendimiento del alumno; la tardanza en la publicación de la asignación de grupos que afecta a la organización de las clases de prácticas, o jornadas que ocupan días enteros que igualmente hace que el rendimiento de los alumnos afectados baje.

En este trabajo se propone una solución a la problemática expuesta anteriormente. Haciendo uso de Inteligencia Artificial se automatiza el sistema para que el proceso de matriculación calcule por si mismo las posibilidades de configuraciones que tiene el alumno según las asignaturas en las que puede y/o quiere matricularse, en lugar de la elección de asignaturas por parte del alumno y después la asignación de grupos. Se propone crea un programa que asigne los grupos de las asignaturas en el acto y optimice los requerimientos de tiempo evitando problemas como los explicados en el anterior párrafo.

El enfoque de este TFG es hacia el alumno, trabajando en alto nivel sobre las estructuras que tienen los horarios, consideradas fijas, y la búsqueda de una asignación de grupos que le presente al alumno la menor cantidad de problemas. Las cuestiones de planificación de horarios a más bajo nivel, sobre la asignación de aulas y profesores a las distintas asignaturas y grupos y periodos de tiempo, se dejan en manos de la administración de la ULL que se encarga de ello en el presente. Queda claro así que las necesidades de tiempo que pueden tener los alumnos y los requerimientos para un buen aprendizaje son el centro de este proyecto, en lugar de la organización eficiente de los recursos disponibles.

De esta manera, los programas desarrollados trabajan con los horarios del curso 2020/2021 del Grado de Ingeniería Informática de la Universidad de La Laguna, y dada una lista proporcionada por el usuario de las asignaturas en las que piensa que puede o quiere matricularse, busca configuraciones de matrícula, con las asignaturas y su asignación de grupos, en las que haya la menor cantidad de conflictos de tiempo.

1.2. Estado del Arte

Una metaheurística es un procedimiento de alto nivel para generar una solución suficientemente buena a un problema dado bajo unos requerimientos determinados de tiempo y recursos, sin asegurar que dicha solución sea la solución óptima global para dicho problema [4]. Estas técnicas son útiles a la hora de resolver problemas cuando los recursos computacionales son limitados.

Los problemas de organización académica se consideran problemas no deterministas NP-duros, lo que significa que el tiempo computacional requerido para su resolución crece exponencialmente con el tamaño de la instancia [1]. Es por esta razón que las metaheurísticas se utilizan para abordar estos problemas, dadas su capacidad para modelar problemas reales de manera flexible y robusta, obteniendo resultados aceptables con tiempos y recursos limitados [9].

La organización o planificación académica comprende varios tipos de problemas diferentes. Se identifican 2 categorías de problemas que son la organización de exámenes y organización de cursos [2]. El problema que propone resolver este TFG caería dentro de la categoría de organización de cursos. Esta categoría se encarga de asignar recursos a periodos de tiempo, coordinándolos para obtener una solución con la menor cantidad de conflictos. A su vez, la organización de cursos se puede separar en otras dos categorías [5]: problemas que utilizan los datos de los alumnos ya matriculados para organizar las clases, aulas, alumnos y periodos de tiempos, éstos son propios de modelos de horarios de posgrados; y por otro lado, los problemas que manejan los cursos ofrecidos por la universidad, que se dedican a asignar recursos limitados, como aulas o personal, a intervalos de tiempo para realizar tareas como son las clases que se deben dar en una asignatura concreta. El problema de este trabajo caería en esta última categoría, por la que se organizan los horarios disponibles de la universidad para asignar las asignaturas y su asignación de grupos a la matrícula del usuario, asegurando la menor cantidad de conflictos con el uso de metaheurísticas.

La metaheurística implementada en este trabajo es un procedimiento de búsqueda adaptativa voraz aleatorizada, comúnmente conocido como GRASP (Greedy Randomized Adaptive Search Procedure). Se trata de una metaheurística que consiste en un método multi-arranque, en el que durante un número de iteraciones determinado se construye una solución diferente que luego se mejora con un procedimiento de búsqueda local. La primera fase de construcción tiene cualidades aleatorias, con lo que se consiguen resultados no-deterministas permitiendo obtener soluciones distintas en cada iteración del procedimiento [7].

El GRASP ha sido aplicado a varios tipos de problemas de optimización combinatoria [3] , entre ellos problemas de planificación de recursos y horarios, que incluyen ejemplos de planificación académica [8].

Además, en la literatura se han propuesto implementaciones multiobjetivo del GRASP y demás metaheurísticas [6]. Dada la naturaleza de este problema, que se explicará en los

siguientes apartado, es interesante poder incluir distintos objetivos a optimizar dentro de la implementación del procedimiento que se quiere desarrollar.

1.3. Contenido de la memoria

En este primer capítulo introductorio describimos la situación actual de la planificación de la actividad académica por parte del estudiante y repasamos el Estado del Arte sobre optimización y metaheurísticas y sus usos en el campo de organización temporal en centros educativos. En el capítulo 2 abordamos la formulación del problema en cuestión y los objetivos que las soluciones obtenidas deben suplir. El capítulo 3 y se explicarán los principales aspectos de la metaheurística GRASP aplicada. En el capítulo 4 se explicará la implementación de la técnica GRASP para el problema, las clases y métodos utilizados. El capítulo 5 analiza los resultados obtenidos y la comparación de las distintas alternativas contempladas. Y finalmente, en el capítulo 6 se expondrán las conclusiones y posibles líneas futuras de trabajo. El capítulo 7 incluye el resumen y conclusiones en inglés (*Summary and Conclusions*).

Capítulo 2

Especificación del problema

El problema abordado en este trabajo consiste en la planificación de la matrícula del estudiante en base a los horarios de las asignaturas elegidas. En la bibliografía sobre planificación académica en universidades u otros contextos académicos los problemas más abundantes tratan de la planificación de la docencia de cursos, de la elección de aulas a las asignaturas o de estudiantes a las aulas, tratando de asignar los recursos de la manera más eficiente posible. Sin embargo, nuestro problema no trabaja al nivel de la administración de la universidad, sino que trata de aprovechar los recursos y su organización ya disponible por el alumnado, es decir, usar la planificación ya ofrecida por la universidad para mejorar la planificación de la matrícula de cada estudiante.

2.1. Formalización del problema

Los datos con los que trabajamos, pues, consisten en la lista de posibles asignaturas que determina el estudiante - las asignaturas donde podría o querría matricularse el alumno - y los horarios de los posibles grupos de dichas asignaturas que obtenemos de la planificación ofrecida por la universidad. Cada asignatura tiene como características de interés el número de créditos, el cuatrimestre al que pertenece y los horarios de cada grupo de teoría, de problema y de laboratorio posible.

Parta establecer la notación matemática N denota el número total de asignaturas disponible y cada asignatura A_i, $i \in \{0, 1, ..., N-1\}$, se caracteriza por

$$A_i = (C_i, S_i, c_i, p_i, l_i)$$

donde:

- $i \in \{0, 1, ..., N-1\}$ es índice de la asignatura,
- C_i es el número de créditos de la asignatura i
- S_i es el cuatrimestre al que pertenece la asignatura i
- $c_i = (c_{i1}, ..., c_{iX_i})$ es el vector de los grupos de teoría de la asignatura i
- $p_i = (p_{i1}, ..., p_{iY_i})$ es el vector de los grupos de problemas de la asignatura i
- $l_i = (l_{i1}, ..., l_{iZ_i})$ es el vector de los grupos de laboratorio de la asignatura i

Las valores X_i, Y_i y Z_i corresponden, respectivamente, al número de grupos de teoría, el número de grupo de problemas y el número de grupo de laboratorio de la asignatura i.

El problema entonces consiste en incluir o no cada asignatura en la matrícula del estudiante, y además en caso de incluirla, determinar la elección de grupo de teoría, de problemas y de laboratorio de cada asignatura incluida. Cada una de las posibilidades de lugar a una solución del problema y el propósito es elegir la solución óptima con la que se obtengan menos conflictos en los horarios del alumno.

2.2. Soluciones del problema

Las soluciones con las que trabajan los algoritmos implementados en este trabajo están representadas como matrices de $N \times 3$, siendo N el número de asignaturas. Las tres columnas representan los tres tipos de grupos: la primera columna el grupo de teoría, la segunda el de problemas, y la tercera el de laboratorio. La siguiente tabla muestra el esquema de una posible solución

A_0	x_0	y_0	z_0
A_1	x_1	y_1	z_1
A_2	x_2	y_2	z_2
...	...	...	...
A_{N-1}	x_{N-1}	y_{N-1}	z_{N-1}

Figura 2.1: Matriz de representación de la solución

Cada asignatura A_i estará identificado por su código según establece la universidad y cada casilla x_i, y_i y z_i puede tener cualquier valor, respectivamente, dentro de la lista c_i de grupos de teoría, p_i de grupos de problemas, y l_i de grupos de laboratorio disponibles en cada asignatura i. Para indicar si la asignatura representada en la fila i no está en matricula se pone un -1 en la casilla del grupo de teoría, de forma que si está incluida en la matricula entonces tendrá un grupo de teoría asignado, por lo cual dicha casilla tendrá un valor distinto de -1. Así mismo, para el caso de asignaturas que no tienen grupos de problemas o laboratorio se pone siempre un -1 en la casilla correspondiente.

Esta representación también puede ser explicada como una lista de ternas $S = (x_i, y_i, z_i), i = 0, 1, ...N - 1$ donde, para cada asignatura i, x_i es uno de sus grupos de teoría, y_i uno de sus grupos de problemas, y z_i uno de sus grupos de laboratorio.

Aparte de las cuestiones temporales de los horarios, una característica crítica de una matrícula es alcanzar un número mínimo de créditos; en otro caso una solución óptima sería no matricularse de ninguna asignatura. Concretamente, en la universidad de La Laguna la matrícula debe incluir el número mínimo de créditos para una matrícula completa, para una matrícula parcial, o para cumplir con las bases de las distintas becas. Por ello, hay que tener en cuenta que la matrícula tenga los créditos necesarios y que no prevalezcan completamente los conflictos en los horarios por encima estos u otros requerimientos administrativos de la matrícula.

Esta característica se tiene en cuenta como una restricción que determina si una solución es factible o no, obligando que para que una solución sea considerada factible debe cumplir el mínimo de créditos, o en caso de que la lista de asignaturas de entrada

en las que el alumno puede matricularse no tuviera ese número de créditos mínimo el alumno tiene que matriculase todas las asignaturas disponibles. Teniendo en cuenta el caso de estudiantes que cursan estudios oficiales de grado a la misma vez que trabajan, o que sea una segunda carrera, sacar el grado curso a curso no tiene tanta importancia como para un estudiante de nuevo ingreso. Una solución que no cumpla con el mínimo de créditos o que no incluya todas las asignaturas incluidas en la entrada puede ser de interés para estos estudiantes. Esto se debe a que las necesidades de tiempo de sus actividades fuera de la Universidad pueden ser más importantes que el tiempo que deben dedicar al grado o la cantidad de asignaturas que deben cursar cada año académico.

Para evaluar si el número de créditos matriculados alcanza el mínimo número de créditos exigidos administrativamente se usa la expresión

$$Cred(S) = \sum_{x_i \neq -1} c_i \geq min_C$$

donde $Cred(S)$ es el número de créditos matriculados y $minC$ el mínimo de créditos requerido.

Por otro lado, en este trabajo no se ha tenido en cuenta la limitación de la asignación de grupos por la que ésta se hace en función del turno asignado al alumno. Las técnicas implementadas asignan los grupos sin hacer distinción de turnos pudiendo el alumno, para una misma asignatura, ir a la clase de teoría en grupo de mañana y a su grupo de prácticas por la tarde. Los grupos solo se describen por el intervalo de tiempo que ocupan. Esto se debe a que, aunque el sistema tenga esa limitación, en la práctica es normal ver a alumnos acudiendo a clases en distintos turnos a los que se le asignaron, sobre todo de teoría y prácticas, por no haberse adecuado la asignación de grupos a las necesidades del alumno.

2.3. Objetivos

La minimización de conflictos en los horarios de los grupos de las asignaturas seleccionados en la matrícula son los que determinan la calidad de las soluciones. Los conflictos que se tienen en cuenta son los solapamientos entre asignaturas diferentes, las jornadas de más de 8 horas o con más de 6 horas consecutivas y las horas muertas. El tiempo semanal durante el que se presenta cada uno de estos cuatro conflictos se toma como una función objetivo a minimizar. Por lo tanto, las valoraciones de las soluciones que se utiliza para compararlas son estas cuatro funciones. Estos cálculos se realizan para cada día de la semana, siendo el valor correspondiente la suma de los tiempos de todos los días de la semana. Una matrícula sin conflictos en sus horarios tendría un valor igual a 0 en cada uno de los objetivos.

Por tanto, los objetivos a minimizar contempladas en este trabajo son los siguientes:

1. **El solapamiento de horarios**

   Evitar el solapamiento de horarios de diferentes asignaturas incluidas en la matrícula es el primero de los objetivos considerados. Este es el conflicto más crítico pues inhabilita al alumno de asistir a una de las asignaturas implicadas lo que puede suponer la pérdida de la evaluación continua.

2. **Jornadas de más de 8 horas**

En segundo lugar se considera el objetivo de evitar jornadas de más de 8 horas. Sobrepasar las 8 horas de clases del alumno provoca que tenga una parte excesiva de su día ocupado por clases y dificulta que pueda dedicar un tiempo razonable al estudio o a otras actividades.

3. **Más de 6 horas de clase seguidas**

   Además del número de horas diarias dedicadas a las clases, sobrepasar las 6 horas consecutivas de clase provoca una bajada notable en el rendimiento en las últimas horas de clase motivado por el cansancio. Evitar más de 6 horas de clase permite que el alumno mantenga un rendimiento apropiado durante toda la jornada de clases.

4. **Horas muertas**

   "Horas muertas"es el termino usual para referirse al intervalo tiempo entre dos asignaturas que supera la margen del descanso predeterminado por los horarios de la universidad. La universidad de La Laguna establece dos periodos de media hora de descanso en todos los horarios, uno por la mañana y otro por la tarde. Todo intervalo de tiempo libre entre dos asignaturas que sea mayor que esa media hora se considerará tiempo muerto. Evitar horas muertas supone una mayor eficiencia del horario, ocupando menos tiempo dentro del día a día del alumno.

Para expresar formalmente los sumatorios que dan lugar al cálculo de estas cuatro funciones objetivo usamos la función:

$$\chi(a) = \begin{cases} a & \text{si } a > 0 \\ 0 & \text{si } a \leq 0 \end{cases}$$

La valoración de la solución S a minimizar por el problema queda representado en las siguientes fórmulas, sumando el resultado de cada día de la semana para cada función objetivo:

$$\text{mín}\, F(S) = \text{mín}\{F_1(S), F_2(S), F_3(S), F_4(S)\}$$

Donde:

1. F_1 contabiliza el total de tiempo de solapamiento entre las asignaturas seleccionadas. Se obtiene de la siguiente suma

   $$F_1(S) = \sum \chi\left(\text{mín}\{Fin_i, Fin_j\} - \text{máx}\{Ini_i, Ini_j\}\right)$$

   Donde $[Ini_i, Fin_i]$ representa cada intervalo de clase de la asignatura i, y la suma está extendida a todos los días de la semana y a todas las horas de clase de los grupos de pares de asignaturas $\{i, j\}$ que no tienen un -1 en la solución S.

2. F_2 evalúa la extralimitación en las 8 horas de clase diaria.

   $$F_2(S) = \sum \chi(t_F - t_I - 8h)$$

   Donde t_F y t_I representan el instante final de la última clase y el inicial de la primera clase de cada día y la suma está extendida a todos los días de la semana.

3. F_3 evalúa los intervalos de tiempo que sobrepasan las 6 horas de clase. consecutiva.

   $$F_3(S) = \sum \chi(T_c - 6h)$$

   Donde T_c representa cualquier intervalo de consecutivo ocupado por clases. La suma está extendida a todos los días de la semana.

4. F_4 evalúa las horas de tiempos muertos entre clases consecutivas.

$$F_4(S) = \sum \chi(T_d > \frac{1}{2}h)$$

Donde T_d representa cualquier intervalo de tiempo muerto entre dos clases. La suma está extendida a todos los periodos de descanso entre asignaturas y todos los días de la semana.

Con la formulación anterior, el objetivo de este trabajo es encontrar una solución que minimice las cuatro funciones objetivo. Para aplicar las técnicas de optimización uniobjetivo se toma como función objetivo suma, siendo esta resultado de la suma de las problemáticas en los horarios.

$$F_{suma}(S) = F_1(S) + F_2(S) + F_3(S) + F_4(S)$$

Capítulo 3

Metaheurística GRASP

La metaheurística GRASP (*Greedy Randomized Adaptative Search Procedure* - Procedimiento de búsqueda voraz adaptativo y aleatorio) combina un proceso constructivo con una búsqueda local dentro de un esquema de arranque múltiple. GRASP es un método de búsqueda multi-arranque en el cual en cada iteración se construye una solución utilizando un algoritmo voraz para luego mejorar la solución obtenida con una búsqueda local u otro procedimiento de mejora de la solución. Utiliza la característica de multi-arranque para diversificar la búsqueda de la solución óptima, construyendo en cada repetición una solución inicial diferente, con cada cual se comenzaría a explorar el espacio de soluciones desde un punto inicial diferente, abarcando más zonas de dicho espacio de soluciones.

Algoritmo 1: GRASP

```
1 while StopCondition = false do ;                          // Arranque múltiple
2
3 |  S ← semiGreedyConstruction() ;                         // Fase constructiva
4 |  S ← localSearch(S) ;                                   // Fase de meojora
```

El pseudocódigo descrito en el algoritmo 1 representa la base de la implementación del GRASP. Durante un número determinado de iteraciones, se construye una solución con un procedimiento voraz aleatorizado y luego la solución generada se mejora con una búsqueda local o con cualquier otro método de mejora. El número de iteraciones del GRASP viene determinado por la condición de parada que estime el programador. En nuestro caso hemos decidido optar por determinar un número de iteraciones que debe ejecutarse el bucle, pudiendo también haber elegido el tiempo de ejecución como condición de parada o incluso el valor de la solución que se desea obtener.

Debido a la formulación del problema por la que tenemos varias funciones objetivo independientes que minimizar, se han implementado dos versiones del GRASP. En la primera versión implementada se intenta minimizar la suma total de todos los objetivos, actuando como si hubiera una sola función objetivo. La segunda se trata de la versión multiobjetivo, en la que se tienen en cuenta cada objetivo independientemente de los demás en cada iteración de los métodos, por lo que se intenta mejorar un objetivo cada vez sin deteriorar los demás.

A continuación se describen los procesos de construcción y mejora del algoritmo GRASP y luego sus versiones multiobjetivo.

3.1. Construcción Semi Greedy

En el paso de construcción el algoritmo voraz añade iterativamente un elemento a la solución guiándose por una función de evaluación adaptativa. Esta escoge el elemento a añadir en cada iteración aleatoriamente de una lista de candidatos de alta calidad, que se calcula en función de los elementos elegidos en iteraciones previas.

Algoritmo 2: SemiGreedy

```
1 S ← ∅;
2 E ← A = {asignaturas disponibles};
3 while S no esté llena do
4 |   RCL ← { elementos de E de menor valor objetivo};
5 |   i* ← elemento de RCL escogido aleatoriamente;
6 |   S ← S ∪ {i*};
7 |   E ← E − {i*};
8 return S;
```

El procedimiento de construcción Semi Greedy descrito en el algoritmo 2 es el siguiente. En primer lugar, parte de una solución vacía *S* y el conjunto de todas las posibles asignaturas con grupos asignados a incluir *E*. A continuación comienza el bucle en el que rellena la solución. En cada iteración construye una lista restringida de candidatos RCL (*Restricted Candidate List*) que contiene un subconjunto de los elementos de *E* que sean de menor valor del objetivo, más tarde se especifica el mecanismo de selección de estos elementos. Acto seguido se escoge aleatoriamente un elemento de esta lista restringida de candidatos y se añade a la solución, actualizando el conjunto de elementos que se podrán añadir en la próxima iteración (*líneas 5, 6 y 7*).

Algoritmo 3: Crear Lista Restringida de Candidatos según calidad

```
1  L ← ∅ ; // Lista de candidatos
2  S ← SolucionIncompleta;
3  if S = ∅ then
4  |   L ← L ∪ {Opciones por defecto de las asignaturas};
5  else
6  |   // Para todas las asignaturas aún por matricular
7  |   for asignatura i ∉ S do
8  |   |   // Si no hay asignaturas matriculadas del mismo curso
9  |   |   if i.year ≠ j.year, ∀j ∈ S then
10 |   |   |   L ← L ∪ {i con opciones por defecto }
11 |   |   else
12 |   |   |   L ← L ∪ {i con grupos asignados en común con alguna asignatura j ∈ S
   |   |   |   que sea del mismo curso que i};
13 c_min = min{c_i : i ∈ L};
14 c_max = max{c_i : i ∈ L};
15 RCL ← {i ∈ L : i.value ≤ c_min + α(c_max − c_min)};
16 return RCL
```

El procedimiento de construcción de la lista restringida de candidatos (RCL) en cada iteración de la función de construcción Semi Greedy se puede observar en el pseudocódigo

del Algoritmo 3. El criterio de selección de los candidatos a incluir en la lista restringida se trata de, a nivel administrativo en la disposición de los horarios proporcionada por la universidad, las asignaturas de un mismo curso no se solapan cuando tienen grupos similares tanto de teoría, de problemas y de laboratorio. Teniendo esto en cuenta, en cada iteración de la función de construcción, el procedimiento de creación de la lista de candidatos devuelve la RCL con las asignaturas consideradas para incluir en la solución con una asignación de grupos similar a las asignaturas de su mismo curso que ya han sido añadidas a la solución en iteraciones anteriores (*línea 10*).

En caso de que aún no se hayan añadido asignaturas a la solución, o de que no hayan asignaturas en la solución del mismo curso que la asignatura que se está procesando, se añadirían a la lista de candidatos las opciones por defecto de dicha asignatura (*líneas 4 y 8*). Dichas opciones por defecto de una asignatura consisten en las primeras configuraciones de asignación de grupos dentro de un mismo turno según la cardinalidad. Por ejemplo, para la asignatura Álgebra, con 2 grupos de teoría, 4 de problemas y 9 de laboratorio, las primeras configuraciones para el turno de mañana serían el grupo 1 de teoría, los grupos 1 o 2 de problemas y los grupos del 1 al 4 de laboratorio, por lo que las opciones por defecto que se incluirán en la lista restringida de candidatos serían las de la tabla 3.1.

C	P	L	C	P	L
1	1	1	1	2	1
1	1	2	1	2	2
1	1	3	1	2	3
1	1	4	1	2	4

Tabla 3.1: Asignación de grupos por defecto para la asignatura Álgebra

Esta técnica de construcción de la lista restringida de candidatos se tuvo en cuenta dada la cantidad muy grande de opciones que podían añadirse a la lista. Se hicieron pruebas para comparar el funcionamiento de esta técnica con la opción más simple de añadir todos los elementos posibles a la lista de candidatos y luego filtrarla. Se observó que añadir todos los elementos posibles tardaba mucho más, debido a que debe calcular las funciones objetivos de cada solución resultante de añadir un elemento de la lista, por lo que se concluyó que esta técnica sería la utilizada. Las pruebas y sus resultados de la comparación entre estos dos métodos se encuentran en el apartado 5.

Tras la creación de los candidatos se filtra la lista obtenida según cardinalidad o calidad. Con el esquema de cardinalidad la lista restringida de candidatos final tendría los k mejores candidatos, mientras que según el esquema de calidad, el mostrado en el pseudocódigo, la lista que se devuelve tiene los mejores candidatos determinados por la siguiente fórmula (*líneas 11, 12 y 13*): sea c_{min} el candidato con menor valor de la lista y c_{max} el candidato con mayor valor, $RCL = \{i \in L : c_{min} \leq i.value \leq c_{min} + \alpha(c_{max} - c_{min})\}$. α es un valor prefijado que determina la voracidad del algoritmo. Si α es 0 se elegirá únicamente el mejor candidato, obteniendo un método greedy, mientras que si α es 1 la lista devuelta tendría todos los candidatos, dando lugar a un procedimiento aleatorio.

3.2. Búsqueda local

Una vez obtenida una solución en la fase constructiva, ésta se mejora con un procedimiento de búsqueda local. Los métodos de búsqueda local mejoran la solución reemplazándola iterativamente por otras soluciones vecinas a la solución actual hasta que no haya mejora posible entre los vecinos de la solución actual.

En nuestra implementación, los vecinos de una solución S ($n \in N(S)$) se obtienen de cambiar alguno de los elementos de la matriz de la solución. Los algoritmos de búsqueda local implementados en este trabajo analizan todas las soluciones resultantes de cambiar un solo elemento de la matriz de la solución actual.

Algoritmo 4: Búsqueda Local Primer Vecino

```
1  S ← Solucion;
2  MEJORA ← true;
3  while MEJORA = true do
4  |   MEJORA ← false;
5  |   for ∀n ∈ N(S) do
6  |   |   if n.value < S.value then
7  |   |   |   S ← n;
8  |   |   |   MEJORA ← true;
9  |   |   if MEJORA = true then
10 |   |   |   break;
```

En la versión de la búsqueda local por el primer vecino implementada según el pseudocódigo descrito en el algoritmo 4, en cada iteración, el algoritmo elige cambiar al primer vecino que encuentre que sea mejor que la solución actual. En la *línea 2* se inicializa un booleano que representa si se ha hecho una mejora a $true$ para entrar en el bucle, que para cuando no se haya encontrado una mejora en la iteración. Dentro del bucle se resetea el booleano de mejora (*línea 4*) y acto seguido se visitan todos los vecinos a la solución actual. Se para de visitar vecinos en cuanto se encuentre uno mejor, momento en el que se actualiza la solución y se cambia el booleano de mejora a $true$ (*líneas 8 y 9*) para salir del bucle que recorre los vecinos y pasar a la siguiente iteración del bucle de la búsqueda. En caso de que la solución actual sea un óptimo local, es decir, que ninguno de sus vecinos sea mejor, entonces el booleano de mejora no se habrá puesto a $true$, lo que hará parar el bucle del procedimiento de búsqueda local.

El algoritmo de búsqueda local según la estrategia de mejor vecino implementada siguiendo el pseudocódigo descrito en el algoritmo 5 cambia en cada iteración la solución actual al mejor de todos sus vecinos, a diferencia de la estrategia del primer vecino que cambiaba al primer vecino que encuentre que sea mejor que la solución actual. Como se puede ver, los pseudocódigos de las dos versiones son similares, con la diferencia que en lugar de actualizar la solución y pasar a la siguiente iteración al encontrar un vecino mejor, en esta versión se recorren todos los vecinos de la solución (*línea 6*), almacenando el mejor que se haya encontrado ($BEST$) y si dicho mejor vecino es mejor que la solución actual entonces ésta se actualiza y el booleano de mejora se vuelve a poner a $true$.

Algoritmo 5: Búsqueda Local Mejor Vecino

```
1  S ← Solucion;
2  BEST ← S;
3  MEJORA ← true;
4  while MEJORA = true do
5  |  MEJORA ← false;
6  |  for ∀n ∈ N(S) do
7  |  |  if n.value < BEST.value then
8  |  |  |  BEST ← n;
9  |  if BEST.value = S.value then
10 |  |  S ← BEST;
11 |  |  MEJORA ← true;
```

3.3. Versión Multiobjetivo

En principio el GRASP se implementó de la forma descrita anteriormente, en la que el valor que tenían en cuenta los códigos para guiar la búsqueda era la suma de todas las funciones objetivo consideradas en el problema. La posibilidad de optimizar cada objetivo de manera simultánea a través de una implementación multiobjetivo se propuso tras la revisión de la formulación del problema. La literatura de implementaciones de metaheurísticas sobre problemas de optimización combinatoria multiobjetivo es escasa comparada con la literatura disponible sobre implementaciones de optimización de un solo objetivo, por lo cual es de especial interés desarrollar más sobre esta faceta de la metaheurística GRASP.

La implementación anterior, en la que la función objetivo utilizada es la suma de todas las funciones objetivo, podría considerarse también como una implementación multiobjetivo ya que se siguen teniendo en cuenta varios objetivos a la vez. Se trataría pues de una versión combinada cuyo funcionamiento podría modificarse multiplicando cada función objetivo por un coeficiente que represente su relevancia.

Para la nueva implementación se decidió optar por una implementación del GRASP multiobjetivo combinado esta vez secuencial. Esta implementación consiste en que, tanto en la fase de construcción como en la de mejora, se tienen en cuenta todos los objetivos, a diferencia de una implementación pura, en la que en cada iteración del GRASP se tiene en cuenta solo un objetivo cada vez [6]. A diferencia del método implementado anteriormente, esta versión se fija en la mejora de un solo objetivo en cada iteración de la construcción y de mejora.

Por lo tanto, el GRASP multiobjetivo implementado considera simultáneamente los objetivos en la fase de construcción y en la fase de mejora. Para ello, primero en la construcción Semigreedy, en lugar de crear la lista restringida de candidatos basándose en el valor de la suma de todas las funciones objetivos, el Semigreedy tiene en cuenta un objetivo diferente en cada iteración elegido al azar entre los cinco disponibles (solapamiento, jornadas de 8 horas, jornadas continuas de 6 horas, horas muertas y mínimo de créditos), añadiendo a la lista de candidatos aquellos que aporten mejor calidad según la función objetivo elegida en cada iteración.

3.4. Semigreedy Multiobjetivo

La diferencia entre los pseudocódigos de la versión de optimización un objetivo y la versión multiobjetivo sería cambiar las líneas del pseudocódigo en las que se crea la lista restringida de candidatos para que en lugar de utilizar el valor de la suma de objetivos c_i del elemento i de la lista de candidatos, utiliza el valor c_{ij} de elemento i de la lista y de la función objetivo elegida j. Por lo tanto, en cada iteración en la que crea la lista de candidatos y añade uno de sus elementos a la solución, tiene en cuenta un objetivo diferente. Para ello al principio de cada iteración del SemiGreedy se escoge el objetivo que se tendrá en cuenta para la creación de la lista de candidatos, y esta se crea a partir de los valores de dichos objetivos en la solución resultado de añadir cada elemento de la lista.

Algoritmo 6: SemiGreedy Multiobjetivo

```
1  S ← ∅;
2  E ← A = {asignaturas disponibles};
3  O ← 0;
4  while S no esté llena do
5  |  if O > Número de objetivos then
6  |  |  O ← 0
7  |  RCL ← { elementos de E de menor valor objetivo O };
8  |  i* ← elemento de RCL escogido aleatoriamente;
9  |  S ← S ∪ {i*};
10 |  E ← E − {i*};
11 |  O ← O + 1;
12 return S;
```

3.5. Búsqueda Local Multiobjetivo

Dado que en la fase de construcción se tienen en consideración todos los objetivos, optimizando la función objetivo elegida al crear la lista de candidatos en cada iteración; en la fase de mejora debemos observar todas las funciones objetivo de igual manera. Debido a que ahora tenemos varias funciones objetivos que optimizar en lugar de solo un valor debemos cambiar el concepto de comparación de soluciones por el que determinamos que una solución es mejor que otra. En la versión de un solo objetivo solo comparábamos un valor, la suma de todos los valores objetivos. Ahora en la versión multiobjetivo calculamos si una solución x domina a otra solución y, esto es, si todas las funciones objetivos de x tienen valores iguales o mejores que las de y, y además que una de las funciones objetivo de x sea mejor que la de y (*línea 7*).

A diferencia de la versión de un solo objetivo en la que se iteraba hasta que no se encontrara ninguna solución vecina mejor que la actual, en la versión multiobjetivo iteramos sobre las serie de objetivos que deseamos tener en cuenta, de forma que el bucle de la búsqueda pare cuando no se hayan encontrado soluciones vecinas mejores que la solución actual para ninguno de los objetivos. Para ello, pasamos a considerar el siguiente objetivo cuando no se haya encontrado un vecino mejor respecto al objetivo actual (*línea 15*), habiendo cambiado la solución actual a dicho vecino anteriormente (*línea 8*). Y en

caso de que se haya encontrado una solución vecina mejor reseteamos el objetivo que se tiene en cuenta (*línea 13*). De esta manera, si no se ha encontrado ninguna mejora para ninguno de los objetivos se cumplirá la condición de parada del bucle pues el identificador del objetivo a tener en cuenta será mayor que el número de objetivos que tenemos.

Algoritmo 7: Búsqueda Local Primer Vecino Multi-Objetivo

```
1  S ← Solucion;
2  NObj ← Número de objetivos;
3  i ← 0;
4  while i < NObj do
5  |  MEJORA ← false;
6  |  for ∀n ∈ N(S) do
7  |  |  if ∀n.o_i ≤ S.o_i ∧ ∃n.o_i < S.o_i, i = 0, ..., k then
8  |  |  |  S ← n;
9  |  |  |  MEJORA ← true;
10 |  |  if MEJORA = true then
11 |  |  |  break;
12 |  if MEJORA = true then
13 |  |  i ← 0
14 |  else
15 |  |  i ← i + 1
```

Capítulo 4

Estructuras de datos

En este capitulo se incluye una descripción de los ficheros con los datos y la estructura de clases usadas en la implementación de los algoritmos descritos en el capítulo 3 cuyos resultados son mostrados y analizados en el capítulo 6. Los archivos con el código de las clases que se explicarán a continuación se encuentran en un repositorio en GitHub: `https://github.com/alu0100965667/TFG_DavidDeLeon`.

4.1. Ficheros *.subject*

La información de las asignaturas con las que trabajan los algoritmos se almacena en ficheros `.subject` con un formato determinado que representa toda la información necesaria de una asignatura. Al inicio de la ejecución del programa, la clase `Instance` lee los ficheros `.subject` de un directorio indicado creando los objetos `SubjectInfo` correspondientes con la información leída. El formato de los ficheros `.subject` es el mostrado en el ejemplo de la figura 4.1 que correspondería a la asignatura Fundamentos de Ingeniería del Software.

Tal y como se muestra en la figura 4.1 los datos se organizan por filas con el contenido indicado en la primera columna con el ejemplo concreto que aparece en la de la derecha. En la primera fila se incluye el nombre de la asignatura seguido por el código correspondiente. A continuación se tiene el número de créditos, el curso del que forma parte según el plan de estudios y el cuatrimestre en el que está programada. Seguidamente aparece el carácter de la asignatura (troncal, obligatoria u optativa) y el itinerario del que forma parte. Si no pertenece a ningún itinerario en la fila correspondiente aparece un 0. Finalmente, precedidos de una fila con las letras C, P y L aparecen los grupos correspondientes de teoría, problemas y laboratorio con los horarios de impartición semanal. En primer lugar aparece el número del grupo (empezando desde 0), la hora de comienzo y el día de la semana de cada sesión.

En el ejemplo representado en la figura 4.1, la asignatura tiene dos grupos de teoría 4 de problemas y 6 de laboratorio. El primer grupo de teoría (grupo 0) tiene clases los lunes a las 12:00 y a las 13:00 y el segundo grupo de teoría tiene clases los lunes a las 18:00 y los martes a la misma hora. Las filas que representan las horas que ocupa la asignatura deben de corresponder con intervalos de una hora (55 minutos en teoría), teniendo que poner varias filas para un mismo grupo, una por cada hora de clase en la semana.

Nombre	`Fundamentos de Ingeniería del Software`
ID	`139262024`
nº créditos	`6`
Curso	`1`
Cuatrimestre	`1`
Optativa	`false`
Itinerario	`0`
C (Teoría)	`C`
Grupo Inicio Día	`0 1200 1`
...	`0 1300 1`
...	`1 1800 1`
...	`1 1800 2`
P (Problemas)	`P`
...	`0 1300 0`
...	`1 1300 3`
...	`2 1900 1`
...	`3 1900 3`
L (Laboratorio)	`L`
...	`0 930 3`
...	`1 930 2`
...	`2 930 1`
...	`3 1530 3`
...	`4 1530 2`
...	`5 1530 1`

Figura 4.1: Formato de los ficheros `.Subject` de las asignaturas

4.2. Estructura de clases

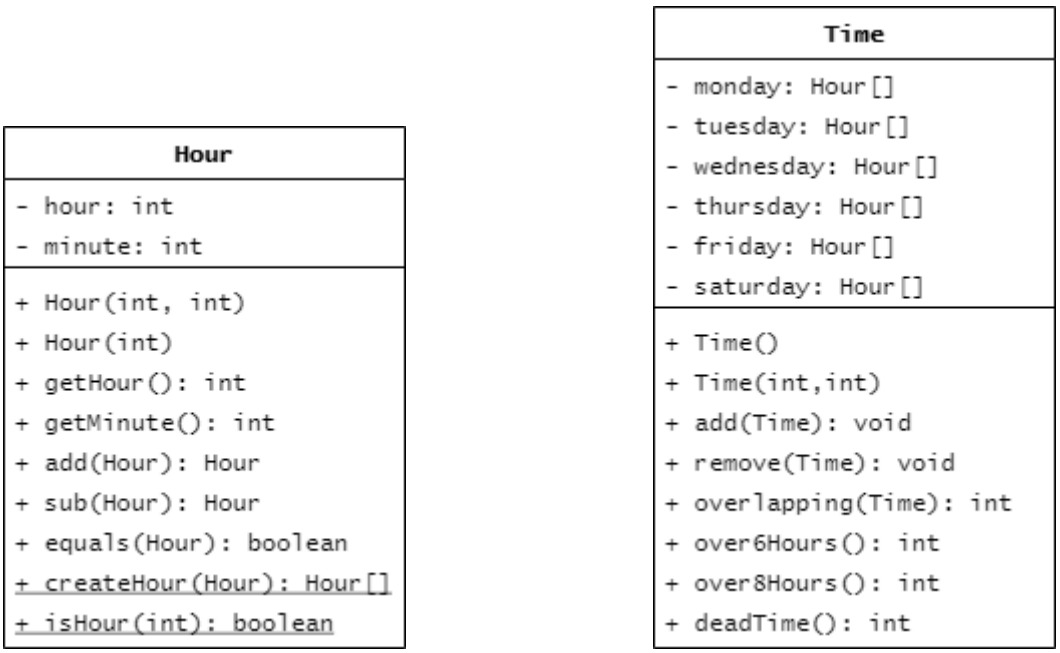
Los algoritmos implementados trabajan con las siguiente estructura de clases.

La clase `Hour`

Tipo de datos que representa las unidades de tiempo que maneja el algoritmo. Almacena la hora y el minuto con sus atributos, y la constante `Interval` determina el intervalo en minutos que representa el tipo de dato, en nuestro caso intervalos de cinco minutos. La clase también contiene métodos para poder operar con las horas pudiendo sumar, restar y comparar (figura 4.2a).

La clase `Time`

Tipo de datos que almacena la información de un horario. Tiene un atributo para cada día de la semana en el que almacena objetos `Hour`, con lo que representa el tiempo que ocupa una actividad dentro de un horario. Contiene métodos para poder añadir y quitar horarios (figura 4.2b). Dado que los objetivos de las soluciones se calculan sobre los horarios, están implementadas en esta clase las siguientes funciones:



(a) Clase `Hour` (b) Clase `Time`

Figura 4.2: Clases `Hour` y `Time`

1. **Función para evaluar los solapamientos** - `overlapping(Time)`:

   Con esta función, a un objeto `Time` se le pasa otro objeto `Time` y calcula los intervalos de tiempo que tienen en común, devolviendo el número de intervalos comunes que se han encontrado. Con esta función se evita uno de los mayores problemas en los horarios de los estudiantes como es el solapamiento de asignaturas.

2. **Función para evaluar el exceso jornada de más de 8 horas** - `over8Hours()`:

   De forma similar a la función anterior, esta función calcula si el alumno tiene una jornada con pausas de más de ocho horas. La diferencia entre esta función y la anterior es que la de jornada de seis jornadas seguidas sin descanso, mientras que esta lo que intenta evitar es que el alumno tenga que estar en la universidad más de ocho horas, para que así sus estudios no limiten otras actividades en el día a día.

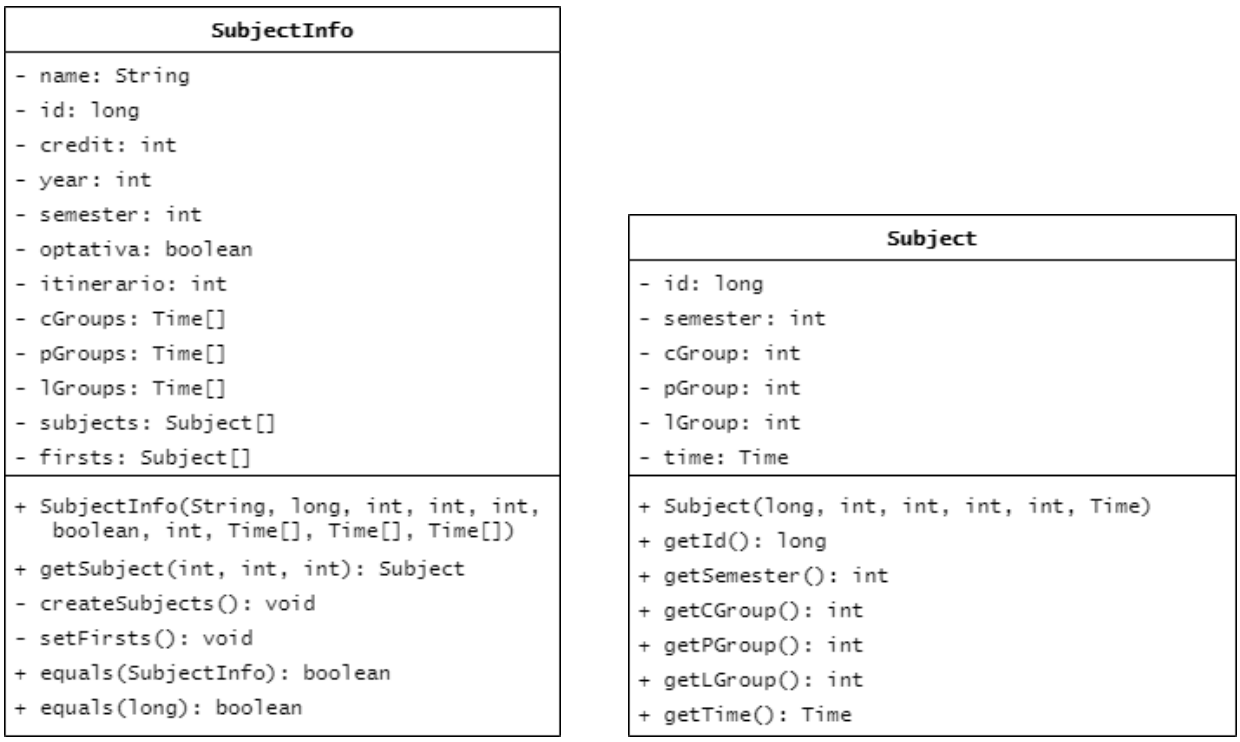
3. **Función para evaluar el exceso de jornada continua de más de 6 horas** - `over6Hours()`:

   Esta función calcula si en alguno de los días del objeto `Time` hay un intervalo de tiempo sin pausas de demás de seis horas, devolviendo la cantidad de tiempo que se sobrepasa de dicho límite. Con esta función se evitaría que la jornada pueda agotar al alumno al no tener ningún descanso en tal intervalo de tiempo.

4. **Función para evaluar los tiempos muertos** - `deadTime()`:

   Esta función busca intervalos de tiempo desocupados que sean mayores que el descanso que se incluye en los horarios normalmente. Analiza los horarios buscando intervalos libre entre clase y clase que sean mayores de 30 minutos que es el tiempo de descanso que tienen todos los horario de la carrera. Está función además de este propósito principal de evitar tiempos muertos, también asegura que se disminuya la elección de grupos de diferentes turnos.

Dado que el objetivo que concierne al número de créditos matriculados no tiene que ver con horarios, éste se encuentra implementado en la clase `Solution`.



(a) Clase `SubjectInfo` (b) Clase `Subject`

Figura 4.3: Clases `SubjectInfo` y `Subject`

La clase `SubjectInfo`

Este tipo de objeto almacena toda la información de una asignatura. Entre sus atributos se encuentran los metadatos compartidos por toda asignatura (nombre, código, curso y cuatrimestre al que pertenece, etc.) y los horarios para cada grupo de teoría, de problemas y de laboratorio que tiene la asignatura. Además tiene un método con el que produce todos los objetos `Subject` para cada configuración de grupos que puede tener la asignatura, que son los objetos que manejan los algoritmos en su ejecución.

La clase `Subject`

Los objetos asignatura (`Subject`) almacenan el identificador de la asignatura que representan junto con los índices de grupos de teoría, problemas y laboratorio elegidos y el horario que resulta de tener dicha configuración de grupos. Esta clase se usa para obtener el horario de una asignatura con una asignación de grupos determinada y utilizarlo para cambiar dicha asignatura en la solución y para calcular las características de dicha solución.

La clase `Solution`

Cada objeto de esta clase contiene toda la información de una solución determinada y los métodos para operar con ella. Como atributos tiene, para cada cuatrimestre, una matriz de representación de la solución, explicada en el apartado de formulación del

problema (capítulo 2), esto es un array de pares de un objeto `SubjectInfo` y tres números para los grupos de teoría, problemas y laboratorio. Además, también tiene para cada cuatrimestre un booleano que representa si la solución tiene o no alguna solución vecina con mejor valor (`best1C` y `best2C`) para que ayude en la ejecución del algoritmo de mejora.

La clase `Solution` descrita en la figura 4.4a tiene también métodos para cada una de las funciones objetivo que devuelven los valores del estado actual de la solución, utilizando los métodos de la clase `Time` para calcular los valores. A su vez tiene un método para el cálculo independiente de las funciones objetivos de cada cuatrimestre. Además tiene la implementación encargada de comprobar que se cumple el mínimo número de créditos `getValueCredits()`. Este método devuelve la diferencia entre el mínimo de créditos requeridos para una matrícula completa y el número de créditos matriculados en la solución. En caso de que el número de créditos matriculados supere al mínimo requerido devuelve un 0 para evitar obtener soluciones con un valor objetivo negativo.

Los métodos disponibles, aparte de constructores y los métodos de obtención (*getters*) y de establecimiento (*setters*), son los siguientes:

- `reset()`: función que desmatricula todas las asignaturas de la solución para que quede vacía, lista para ser tratada por uno de los algoritmos.
- `getNeighborFirstBest()`: función que devuelve el primer vecino que encuentre que sea mejor que la solución actual. En esta función, las soluciones vecinas a una solución dada son aquellas resultantes de cambiar alguno de los elementos de las matrices de la solución dada.
- `getNeighborBest()`: función que devuelve el mejor de los vecinos de la solución actual. Recorre todos los vecinos resultantes de cambiar uno de los elementos de la solución actual y devuelve aquel que tenga el mejor valor de función objetivo.
- `getNeighborMO()`: versión de la búsqueda local para la implementación de un GRASP multiobjetivo que optimiza cada función objetivo por separado.
- `dominates(Solution)`: método que devuelve *true* si la solución pasada por parámetro domina a la actual. Esto es que las funciones objetivos de la solución parámetro de valores iguales o mejores que la actual y que por lo menos uno de los objetivos sea mejor. Tiene además una versión para tener en cuenta la mejora de un objetivo concreto pasado por parámetro.

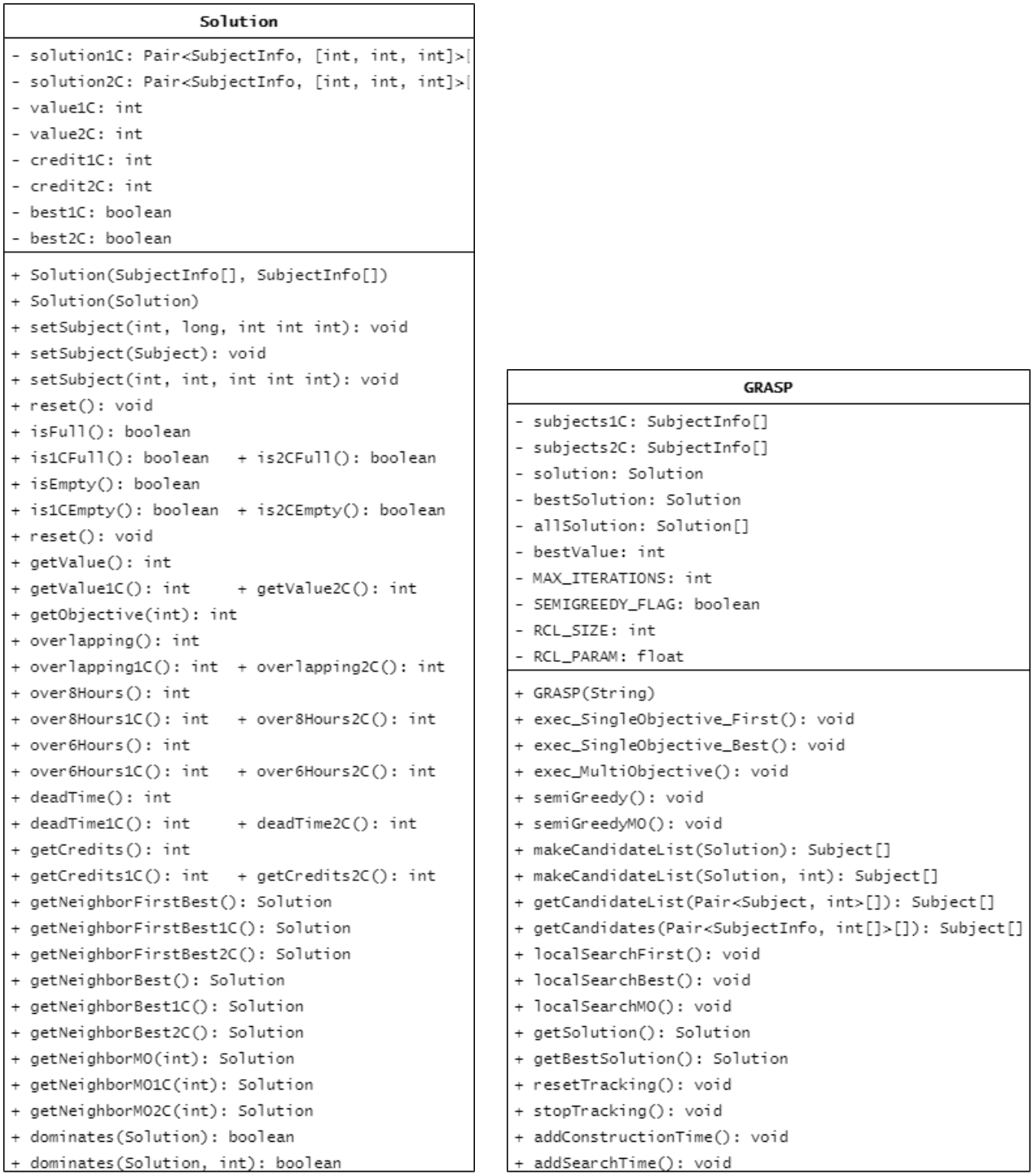
De esta forma, la clase `Solution` queda expuesta como el almacenaje de la representación de la solución y los métodos necesarios para la obtención de vecinos y el reseteo para reiniciar la ejecución de los algoritmos.

La clase `GRASP`

La clase `GRASP` descrita en la figura 4.4b implementa los pseudocódigos explicados en el apartado 3 que componen el algoritmo GRASP. Contiene métodos para la fase de construcción (`semiGreedy()`), la creación de la lista restringida de candidatos, y la búsqueda local o fase de mejora. También tiene una implementación multiobjetivo para cada uno de dichos métodos. Además tiene métodos para calcular el tiempo de ejecución del algoritmo y de las fases de construcción y mejora para el estudio estadístico que se hará tras el desarrollo del código (`resetTracking()`, `stopTracking()`...).

La clase `Instance`

Esta clase no se encarga del almacenaje de información, sino de la lectura de los archivos *.subject*. Su única función implementada es el constructor, con el que lee todas las asignaturas de ficheros *.subject* colocados en un directorio determinado y crea los objetos `SubjectInfo` correspondientes para el funcionamiento de los algoritmos



(a) Clase `Solution` (b) Clase `GRASP`

Figura 4.4: Clases `Solution` y `GRASP`

Capítulo 5

Resultados y comparación

En este capítulo mostraremos los experimentos realizados para comparar distintas opciones y los resultados obtenidos. Tanto los archivos de los programas, como las instancias utilizadas y los datos recogidos de las ejecuciones de los algoritmos se encuentran en un repositorio en GitHub: `https://github.com/alu0100965667/TFG_DavidDeLeon`.

Los programas han sido implementados en el lenguaje de programación JAVA con la versión 11. Se han ejecutado en un ordenador personal con procesador Inter Core i3-4030U con 1.90GHz y 4GB de memoria RAM.

5.1. Instancias

Para probar los algoritmos implementados se han utilizado varios tipos instancias o entradas distintas. Todos los datos han sido extraídos de la información disponible sobre el grado en Ingeniería Informática de la Universidad de La Laguna que se imparte en la Escuela Superior de Ingeniería y Tecnología. En el capítulo 2 se especificó que la entrada del problema es una lista de asignaturas en las que el alumno puede y/o quiere matricularse. La información de cada una de la serie de asignaturas que el alumno quiera tener en cuenta está en los ficheros `.subject` que se han creado a partir de los horarios y demás información proporcionada por el centro. Los ficheros `.subject` se sitúan en un directorio cuya ruta se le pasa como parámetro a los constructores del programa y éste crea los objetos necesarios para la ejecución de los algoritmos.

Las instancias creadas se tratan de casos que pueden darse en la realidad universitaria, puesto que el enfoque y objetivo de este trabajo es el análisis de las técnicas y los algoritmos implementados, para aportar una solución técnica realista a problemas a los que se enfrentan alumnos y alumnas de la comunidad universitaria al formalizar su matrícula. Observando las características de la realidad universitaria de los estudios de Ingeniería Informática en la Universidad de La Laguna diferenciamos tres tipos de situaciones y de las instancias correspondientes:

Tipo 1. El tipo de situaciones más frecuente son aquellas en las que el alumno se matricula únicamente en todas las asignaturas de un mismo curso, sin mezclar curso, que abarca a aquellos estudiantes que sacan la carrera año a año. El centro elabora los horarios de cada curso con el propósito ineludible de que estos alumnos puedan realizar la matrícula directamente sin que se presente ninguno de los conflictos que estamos considerando en sus horarios.

Tipo 2. Por otro lado, es relativamente frecuente el caso de alumnos que no consi-

guen superar alguna asignatura en uno de los cursos. Estas situaciones dan lugar a instancias con más asignaturas ya que se deben incluir las no superadas de cursos anteriores además de todas las del curso en el que se encuentre el alumno. Este tipo de instancias pueden ser dividida en otras categorías según cuántos cursos diferentes abarcan las asignaturas de la instancia. También se puede diferenciar distinto grado de dificultad de la instancia según la cantidad o proporción de asignaturas de cada curso que se incluye entre las asignaturas consideradas para un mismo cuatrimestre. En principio sería más complicada aquella instancia que tenga una proporción más equilibrada entre las asignaturas de los distintos cursos que las instancias correspondientes a los alumnos que tienen la mayoría de asignaturas de un curso y unas pocas de cursos anteriores.

Tipo 3. Por último, se encuentra el caso de estudiantes que, generalmente por motivos de trabajo, no desean matricularse en el número de asignaturas o créditos que requiere la universidad para una matrícula completa. En estos casos los programas y algoritmos trabajarán con menos asignaturas según dicte el usuario. Aún así la suma del número de créditos de las asignaturas a tener en cuenta debe de ser igual o superior al mínimo requerido para una matrícula parcial dándose un margen más amplio en el número de créditos matriculados y en la selección de las asignaturas a tener en cuenta.

En primer lugar, para las instancias de tipo 1, en las que las asignaturas posibles son todas del mismo curso, el algoritmo en todas sus variantes probadas llega por lo menos una vez a una solución con valor 0, es decir, que no tiene ningún conflicto de los que se tienen en cuenta en este trabajo. Estos resultados eran de esperar pues los horarios y grupos de las asignaturas para un mismo año están dispuestos de tal manera que no supongan ningún problema para el alumnado matriculado. Por lo que se puede concluir que los problemas como el solapamiento de horarios están resueltos desde el nivel administrativo de la Universidad en el caso de que el estudiante solo tenga matriculadas asignaturas de un mismo año.

Por esta razón el enfoque en el estudio computacional de los algoritmos estará en las instancias del segundo tipo en las que se mezclan asignaturas de distintos años, pudiendo con éstas medir la dificultad de las instancias para comparar las distintas implementaciones del GRASP que se han desarrollado. Además se utilizarán principalmente casos o asignaturas que incluyan los dos primeros años debido a que a partir del tercer año, dentro del Grado en Ingeniería Informática en la Universidad de La Laguna que está siendo usado para probar los algoritmos, las asignaturas son parte de itinerarios distintos, por lo que hay menos cantidad de grupos en cada asignaturas al estar los alumnos repartidos entre los cinco itinerarios disponibles en el momento de desarrollo de este trabajo. Por lo tanto, en busca de instancias que comprometan lo más posible los métodos implementados se considerarán las asignaturas con mayor variedad de grupos, que en el caso de este grado son las correspondientes a los dos primeros años.

5.2. Mejora en la creación de la RCL

En primer lugar se han analizado los resultados de la mejora del método implementado para la creación de la lista restringida de candidatos (RCL). En esta mejora se añaden elementos a la lista según los cursos y la asignación de grupos de las asignaturas incluidas

en la solución en iteraciones anteriores. Este método se ha comparado con aquel en el que la lista restringida contiene todas las asignaciones de grupos posibles de las asignaturas que aún no estén en la solución. Con esta comparación se demuestra que el método implementado es más eficiente que un método más general, obteniendo mejores resultados tanto en la calidad de las soluciones como en el tiempo de ejecución.

Los resultados de la tabla 5.1 se han obtenido ejecutando 50 iteraciones del algoritmo constructivo SemiGreedy con el método implementado y sin él para varias instancias de distinto tipo y dificultad. Se ha sacado de los datos adquiridos el promedio del tiempo de ejecución y de la suma de los valores objetivo obtenidos en cada iteración. La tabla muestra la media del tiempo de ejecución medida en segundos y calidad de las soluciones obtenidas por los métodos de creación de la lista restringida de candidatos con y sin la mejora.

Las instancias utilizadas en esta comparación son:

Instancia 1. Las asignaturas de segundo año sin ninguna otra asignatura adicional.

Instancia 2. Las asignaturas de segundo año junto con asignaturas del primer año.

Instancia 3. Las asignaturas de tercer año con tres opciones de asignaturas de los dos años anteriores en cada cuatrimestre.

	T. ejecución		Valor total	
	Sin mejora	Con mejora	Sin mejora	Con mejora
Instancia 1	68	4	332	44
Instancia 2	228	8	426	108
Instancia 3	72	7	357	104

Tabla 5.1: Comparación de la mejora en la generación de la RCL

Se puede observar la gran diferencia en el rendimiento de los dos métodos. La diferencia en el tiempo de ejecución promedio se debe a que a la hora de filtrar la RCL se debe calcular el valor que aporta cada elemento considerado para ser incluido. El método implementado reduce la cantidad de elementos a tener en cuenta, por lo que reduce la cantidad de operaciones a ejecutar para obtener el valor de cada uno de los elementos considerados a incluir en la lista. Por otro lado, la gran variedad de elementos que se consideran para incluir en la RCL si no se utiliza el método implementado es lo que produce soluciones de peor calidad. El método implementado utiliza las características consideradas a nivel administrativo en la creación de los horarios. Un método que añada todas las opciones a la lista añadiría también asignaciones de grupos tan dispares que la solución se iría construyendo sin ninguna dirección o sentido.

5.3. Parámetros del SemiGreedy

Los segundos experimentos y comparaciones se han hecho sobre los parámetros de la fase de construcción SemiGreedy que afectan al nivel de aleatoriedad del método. Se han implementado dos formas de la fase de construcción: una basada en calidad, en la que la lista restringida de candidatos se filtra según la calidad que sus elementos aportan a la solución, y otra basada en cardinalidad, en la que la lista se filtra según la cantidad de elementos que queremos que tenga.

Para las siguientes gráficas se ha utilizado los datos de 50 iteraciones para cada valor del parámetro a estudiar. La instancia utilizada consiste en las asignaturas del segundo año con asignaturas del año anterior en cada cuatrimestre para añadir variedad, formando parte por lo cual del segundo tipo de instancias.

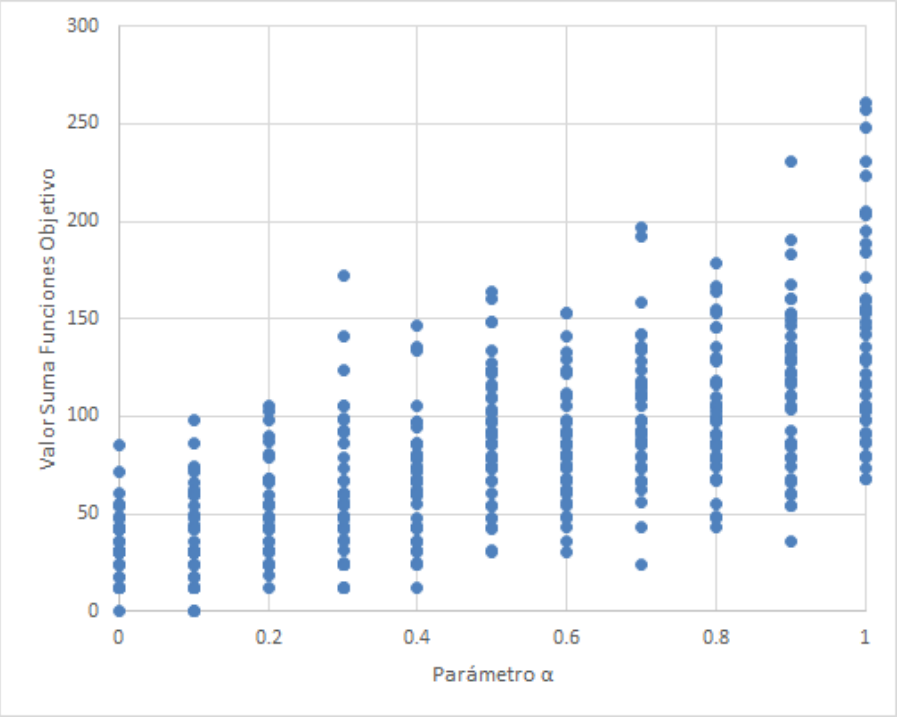


Figura 5.1: Valores de la construcción SemiGreedy con distinto parámetro α

En primer lugar observamos los resultados para la construcción basada en la calidad de los elementos de la lista de candidatos en la figura 5.1. Podemos observar que la variedad de la calidad de las distintas soluciones que devuelve el método crece al mismo tiempo que el valor del parámetro α aumenta. Esto se debe a que el parámetro α determina el nivel de aleatoriedad: con un valor 0 solo dejaría en la lista aquellos candidatos con el mejor valor, obteniendo así un método completamente voraz, y con 1 dejaría en la lista todos los candidatos tenidos en cuenta, con lo que obtenemos un algoritmo aleatorio.

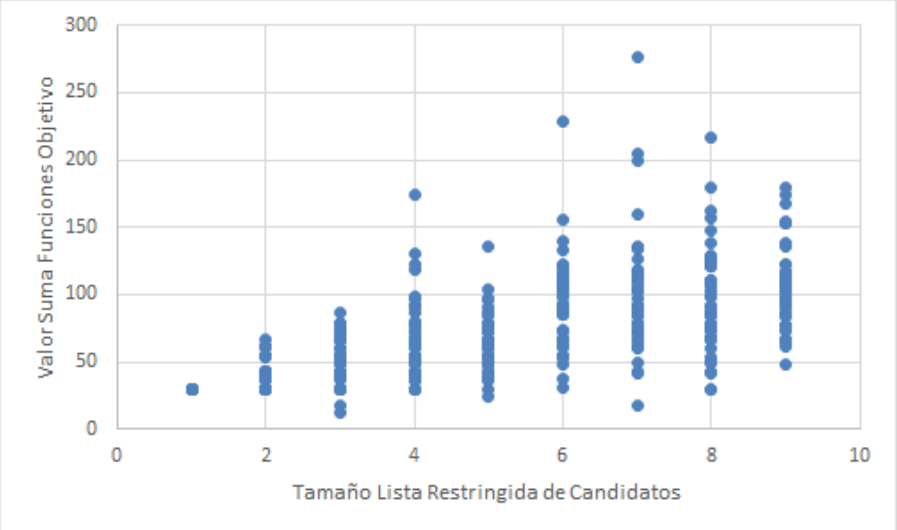


Figura 5.2: Valores de la construcción SemiGreedy con distinto tamaño de RCL

Para la implementación del SemiGreedy según la cardinalidad de los elementos de la lista de candidatos obtenemos la gráfica de la figura 5.2. En esta implementación el parámetro que podemos variar es el tamaño de la lista de candidatos. Como podemos observar una lista de candidatos de tamaño 1 devuelve soluciones con el mismo valor. De hecho, todas esas soluciones son la misma ya que los programas ordenan siempre los elementos de la lista con valores iguales de la misma manera. Por lo que una lista de tamaño un implica obtener siempre la misma solución.

5.4. Comparación de la versión objetivo

Por último se ha comparado los resultados de las dos versiones implementadas del GRASP: la versión multiobjetivo y la que utiliza como única función objetivo la suma de todas las funciones objetivo. Para ello se ha utilizado una instancia con asignaturas del tercer curso con asignaturas adicionales de los dos curso anteriores para cada cuatrimestre.

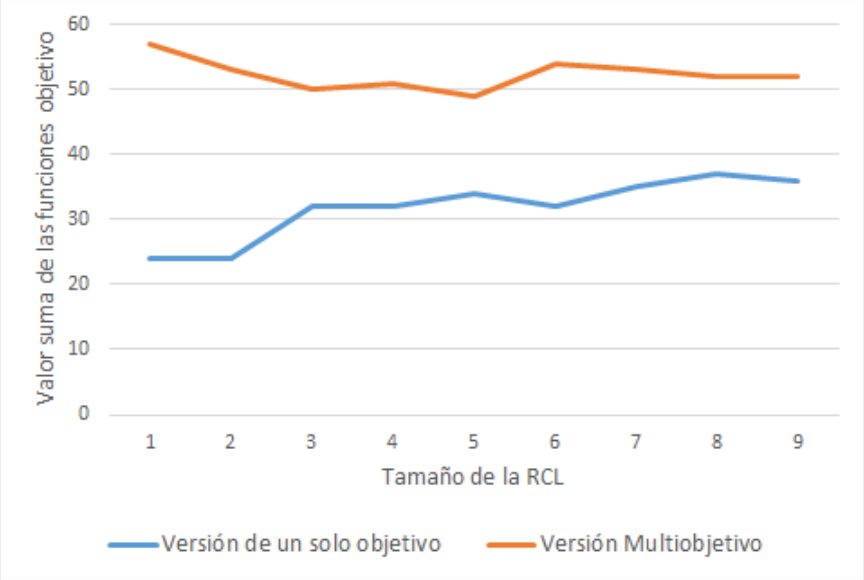


Figura 5.3: Comparación de los valores promedio de las dos versiones implementadas

Podemos observar que la versión multiobjetivo devuelve soluciones de peor calidad que la versión que combina las funciones objetivo. Esto se puede deber a las restricciones que supone la versión multiobjetivo que busca soluciones vecinas que dominen a la solución actual según un objetivo. Esto significa que no se permite el deterioro de ninguna de las funciones objetivo en la búsqueda de un vecino al que moverse, lo que implica que este movimiento está más restringido. La versión que utiliza la suma de todas las funciones objetivo puede moverse a soluciones que deterioren una de las funciones objetivo, pero que la mejora de los demás objetivos supere este deterioro y por lo cual se pueda mover a mayor cantidad de vecinos. Por lo cual, la versión de un solo objetivo puede abarcar más del espacio de soluciones que la versión multiobjetivo.

Las gráficas 5.4 y 5.5 muestran los valores de 300 iteraciones del GRASP de un solo objetivo para cada valor de aleatoriedad de la fase de construcción. Se observa la diferencia con las gráficas 5.1 y 5.2 de los datos de la fase de construcción. La mejora que lleva a cabo la fase de búsqueda local es observable en los peores valores que llega a devolver solo la fase de construcción y una iteración completa del GRASP. Mientras que

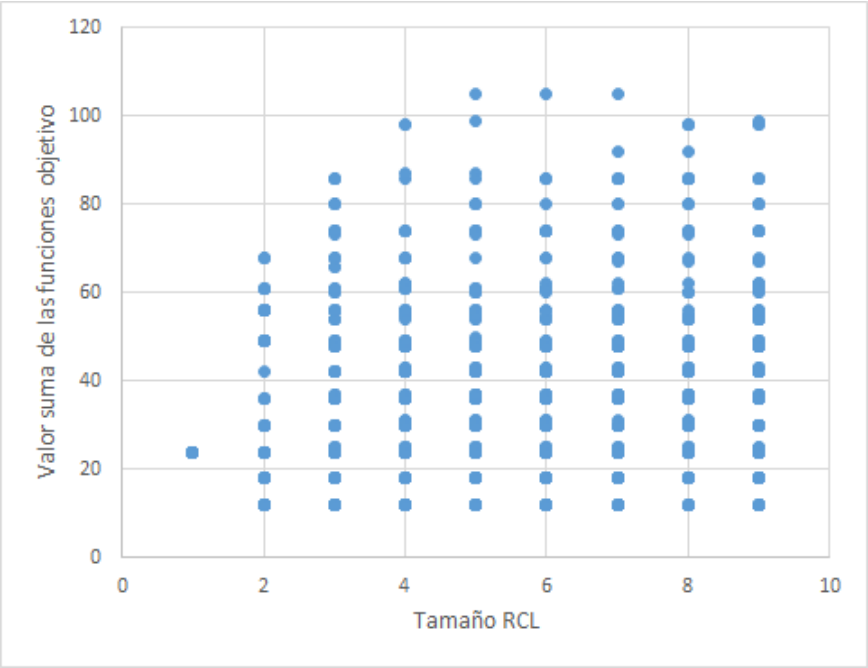


Figura 5.4: Valores de las ejecuciones de la versión de un solo objetivo primer vecino

con la fase de construcción observamos soluciones que llegan a valores de 200, con la fase de mejora la gran mayoría de soluciones no supera el 100 de valor de las funciones objetivo.

Además se observa cómo el parámetro α compromete más la calidad de las soluciones devueltas dado que si α es igual a 1 deja entrar a todas las opciones a la lista de candidatos. Mientras que el tamaño de la lista de candidatos se mantiene siempre el mismo, reduciendo la aleatoriedad de las soluciones construidas de forma más restrictiva.

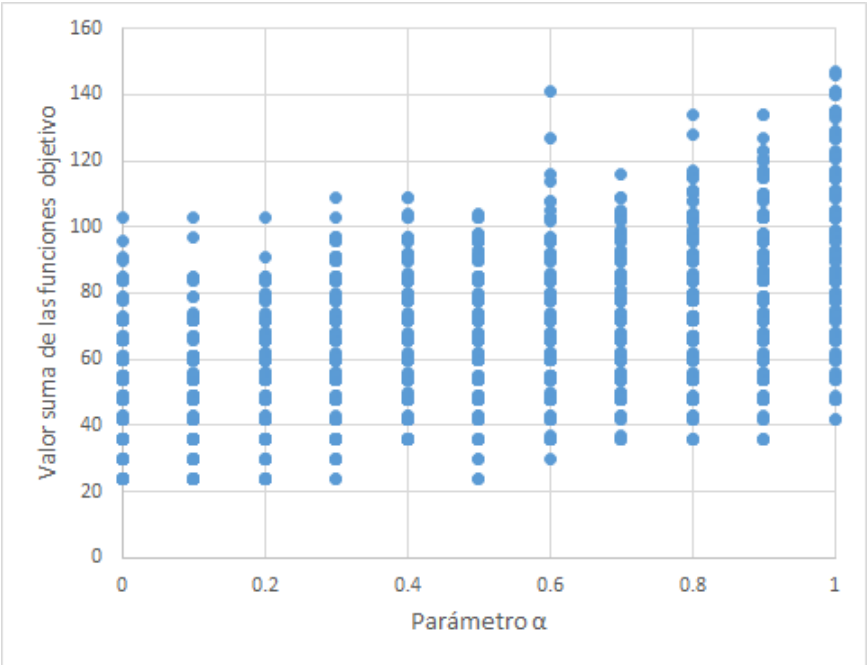


Figura 5.5: Valores de las ejecuciones de la versión de un solo objetivo mejor vecino

La gráfica 5.6 muestra los valores de las soluciones devueltas por la versión multiobjetivo. La diferencia con las gráficas anteriores 5.4 y 5.5 está en lo uniforme de la distribución de sus resultados. Esta diferencia es el resultado de tener un espacio más restringido en el que se realiza la búsqueda local. Las soluciones óptimas localmente son más frecuentes que en las versiones de un solo objetivo. Este suceso es consecuencia de que las características que describen un vecino mejor son más restrictivas en la versión multiobjetivo, por lo cual una misma solución puede tener más cantidad de vecinos mejores en la versión de un solo objetivo que en la multiobjetivo. Esto provoca que la búsqueda local se estanque en una mayor cantidad de soluciones, dando lugar a un gráfico uniforme a diferencia de una que denote una tendencia más específica.

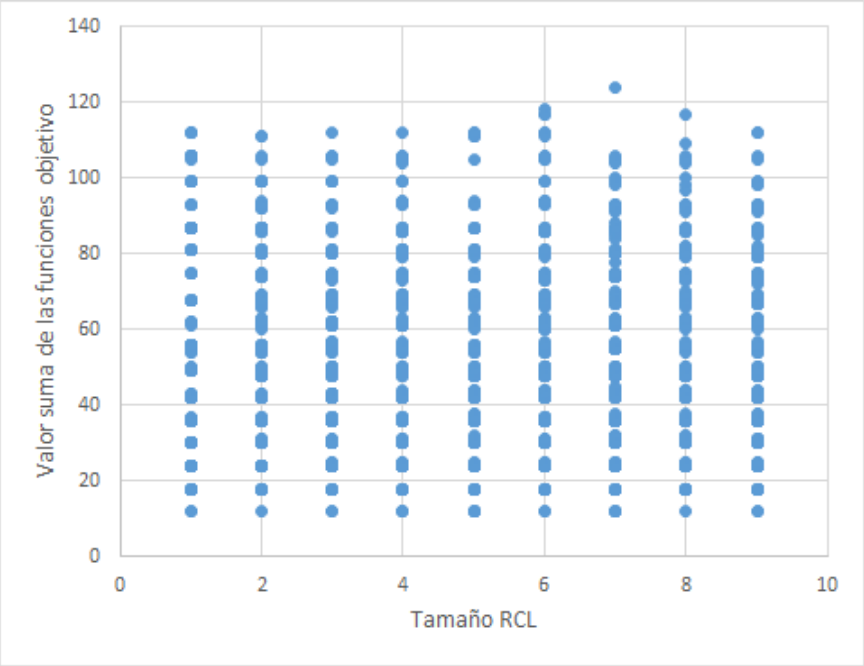


Figura 5.6: Valores de las ejecuciones de la versión multiobjetivo

Capítulo 6

Conclusiones y líneas futuras

El objetivo de este trabajo ha sido proponer una solución técnica a un problema que sigue persistiendo dentro de la comunidad universitaria. En primer lugar se ha estudiado la literatura sobre problemas similares y la implementación de metaheurísticas para su resolución. Luego se ha planteado el problema real de conflictos en los horarios del alumnado como un problema de optimización combinatoria para que pueda ser tratado por algoritmos que implementen una metaheurística. Tras la formulación del problema se ha descrito el funcionamiento de la metaheurística GRASP utilizada, las estructuras de datos necesaria para su implementación, y los resultados y comparación de su rendimiento.

El código trabaja con los horarios del Grado de Ingeniería Informática. En el inicio de la planificación de este trabajo se consideró implementar los algoritmos de forma que trabajasen con los horarios de todas las carreras de la universidad, o por lo menos un subgrupo de ellas, pero los modelos de horarios de la ULL difieren entre las carreras. El ejemplo más claro es la diferencia entre los horarios del Grado de Ingeniería Informática y los de Ingeniería Mecánica e Ingeniería Electrónica. Mientras que las clases de Informática tienen horarios fijados durante todo el cuatrimestre, en Mecánica y Electrónica varias clases pueden ser mensuales o quincenales. Además objetivos de este trabajo, como que la jornada no supere las 8 horas al día, son inevitables en Mecánica y Electrónica, en las que no hay turnos diferenciados y los días con clase desde por la mañana hasta por la tarde noche son la norma. Por lo tanto la implementación general a toda la Universidad de La Laguna sería de especial interés, aunque conllevaría a una administración más centralizada para la organización de los horarios de cada grado, o de una implementación mucho más complicada que abarque todos los tipos distintos de grupos que puedan tener todas las asignaturas de todos los grados de la ULL, así como las dificultados que vienen en los horarios de cada grado de forma predeterminada.

El desarrollo de la idea de este trabajo se podría continua con la implementación de otras metaheurísticas. Inicialmente se pretendía implementar varias metaheurísticas en el marco de este trabajo, entre ellas la búsqueda Tabú y la búsqueda por entornos variables (VNS). Estos dos ejemplos serían relativamente sencillos de implementar llevando a cabo cambios en las búsquedas locales ya implementadas. Se decidió implementar la versión multiobjetivo del GRASP en su lugar por la naturaleza y formulación del problema tratado y, además, el interés en el enfoque alternativo que supone una implementación multiobjetivo, de la que no hay tantos ejemplos en la literatura como de metaheurísticas fijadas en un solo objetivo.

Siguiendo el enfoque que tiene este trabajo hacia el alumno, queriéndose diferenciar de aquellos trabajos enfocados en recursos materiales sobre las personas involucradas, una línea futura en el desarrollo e integración de esta idea es introducir al alumno en la metaheurística. Para ello se podría ofrecer una opción para que el alumno pueda introducir intervalos de tiempo de, por ejemplo, sus actividades extraescolares, que el algoritmo debe tener en cuenta para evitar ocuparlos. También se podría dar la opción que el alumno pueda priorizar o suprimir objetivos según le convengan. Otras cuestiones que se podrían incluir son otras funciones objetivo que representen conflictos en los horarios que no se hayan tenido en cuenta en este trabajo.

En general, el futuro de este trabajo reside en su integración real en el proceso de matriculación en la universidad. Para ello es necesario de una coordinación de la administración de la universidad a todos los niveles. Desde la creación de los horarios de cada grado, pasando por los servicios TIC ocupados del proceso de matriculación. Se necesita de un cambio de paradigma en el que se atienda a las razones de las caídas del rendimiento académico de los alumnos y alumnas, y se intenten solucionar todos aquellos conflictos que se deban a características o fallos del sistema presente de la universidad.

Capítulo 7

Summary and Conclusions

The goal of this final project is to come up with a solution to problems and incoveniences that are shared by a large part of college students, and more specifically implementing a solution catered to the specifications of the Computer Science degree at the University of La Laguna. In this paper we first presented the issues that are common regarding time management in the process of college tuition and group assignment. Then we translated those issues into a combinatorial optimization problem listing four objective functions, each one representing a different issue: overlapping subjects, long and continued periods of class time with no breaks, and long periods of idle time between classes. The implemented techniques were then explained through their pseudocodes, and the data structures used in the implementations were shown. Finally the two versions of the implemented GRASP heuristics were compared and future paths to follow in the development of the idea of this final project were proposed.

Some future paths to be followed after the conclusion of this project are the addition of new objective functions that represent new time conflicts in the student experience that haven't been covered in this paper. As well as enabling the student to prioritize or suppress objective functions depending on their needs. Also, a way for the student to put the periods of time that they want to avoid occupying could be implemented. All of these paths follow the work's focus on the student experience, prioritizing the student's needs since the formulation of the problem.

The future of this project is determined by its integration in the tuition procedure, for which it is necessary the coordination between all levels of the administration inside the University of La Laguna. From the services in charge of the creation of the timetables for every subject, year and degree, to the IT services in charge of implementing the web services that hold the process of tuition.

Bibliografía

[1] Victor A. Bardadym. “Computer-aided school and university timetabling: The new wave”. En: *international conference on the practice and theory of automated timetabling*. Springer. 1995, págs. 22-45.

[2] Arindam Chaudhuri y Kajal De. “Fuzzy genetic heuristic for university course timetable problem”. En: *Int. J. Advance. Soft Comput. Appl* 2.1 (2010), págs. 100-121.

[3] Paola Festa y Mauricio G.C. Resende. “An annotated bibliography of GRASP–Part II: Applications”. En: *International Transactions in Operational Research* 16.2 (2009), págs. 131-172.

[4] Michel Gendreau y Jean-Yves Potvin. *Handbook of Metaheuristics*. 2nd. Springer Publishing Company, Incorporated, 2010. isbn: 1441916636.

[5] Zhipeng Lü y Jin-Kao Hao. “Adaptive tabu search for course timetabling”. En: *European journal of operational research* 200.1 (2010), págs. 235-244.

[6] Rafael Martí y col. “Multiobjective GRASP with path relinking”. En: *European Journal of Operational Research* 240.1 (2015), págs. 54-71.

[7] Mauricio G.C. Resende y Celso C. Ribeiro. *Optimization by GRASP*. Springer, 2016.

[8] L.I.D. Rivera. “Evaluation of parallel implementations of heuristics for the course scheduling problem”. En: *Master’s thesis, Instituto Tecnologico y de Estudios Superiores de Monterrey, Monterrey, Mexico* (1998).

[9] Chong Keat Teoh, Antoni Wibowo y Mohd Salihin Ngadiman. “Review of state of the art for metaheuristic techniques in Academic Scheduling Problems”. En: *Artificial Intelligence Review* 44.1 (2015), págs. 1-21.