ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

MODELO DE PROGRAMACIÓN LINEAL ENTERA PARA LA GENERACIÓN DE HORARIOS DE CLASE EN LA UNIVERSIDAD

PROYECTO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERA MATEMÁTICA

MARÍA BELÉN HEREDIA GUZMÁN
mariabelenherediag@gmail.com

Director: LUIS MIGUEL TORRES CARVAJAL
luis.torres@epn.edu.ec

QUITO, OCTUBRE 2014

DECLARACIÓN

Yo MARÍA BELÉN HEREDIA GUZMÁN, declaro bajo juramento que el trabajo aquí escrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual, correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su reglamento y por la normatividad institucional vigente.

María Belén Heredia Guzmán

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por MARÍA BELÉN HEREDIA GUZMÁN, bajo mi supervisión.

Luis Miguel Torres Carvajal
Director del Proyecto

AGRADECIMIENTOS

En primer lugar, agradezco a mi familia por toda la paciencia y comprensión en los días grises, a mi padre por su sabiduría, consejos y enseñanzas que guían mi camino. A mis amigos por compartir tantos momentos gratos, por animarme y ser motivo de alegría. Finalmente, a Luis Miguel por la valiosa oportunidad de desarrollar este proyecto, y por ser un gran profesor e inspiración.

DEDICATORIA

A mi padre.

María Belén

Índice de contenido

Índice de figuras

Índice de tablas

Resumen

En el presente trabajo presentamos un modelo de programación lineal entera para la programación de clases en la universidad, teniendo en cuenta las restricciones impuestas por la disponibilidad de profesores y aulas, así como por los pénsums de los estudiantes. Proponemos una heurística constructiva y una heurística de mejora local para obtener soluciones factibles para el modelo, y presentamos los resultados computacionales para varias instancias entre ellas correspondientes, a la planificación de horarios en la Facultad de Ciencias de la Escuela Politécnica Nacional.

Abstract

In this work we present an integer programming model for scheduling lectures at a university, taking into account restrictions imposed by the availability of lecturers and classrooms, as well as by the students' curricula. We propose a constructive heuristic and an local improvement heuristic to provide feasible solutions for this model, and report computational results for various instances including real ones concerning the timetable planning at the Science Faculty of the Escuela Politécnica Nacional.

Capítulo 1

Introducción

El problema de planificación de horarios ha sido ampliamente estudiado en los campos de la optimización combinatoria y la investigación de operaciones por aproximadamente 50 años, debido a su gran aplicabilidad práctica. En efecto, problemas de este tipo se presentan en contextos tan diversos como la programación de horarios de clase [1, 2], horarios de enfermería [3, 4], y horarios de deportes [5, 6], entre otros. Un indicador del creciente interés en la automatización de la programación de horarios son las conferencias PATAT (Practice and Theory of Automated Timetabling), las que se llevan a cabo cada dos años con la finalidad de difundir nuevos modelos, algoritmos y aplicaciones en este campo.

El objetivo del presente trabajo es desarrollar e implementar un modelo de programación lineal entera para la planificación de horarios de clase en el contexto particular de la Facultad de Ciencias de la Escuela Politécnica Nacional. Debido a que la solución de este modelo resulta ser computacionalmente muy costosa para los solvers actuales, se investigan además heurísticas primales (de tipo constructivo y de mejora) para producir soluciones iniciales en un esquema de ramificación y corte.

Esta tesis está organizada en seis partes: en el Capítulo 1 se introducen conceptos teóricos y se resumen trabajos preliminares relevantes para los objetivos de la investigación, en el Capítulo 2 se presenta la formulación del problema como un modelo de programación lineal entera, en el Capítulo 3 se describen las heurísticas primales de solución, en el Capítulo 4 se detalla la implementación computacional del modelo, en el Capítulo 5 se discuten los resultados obtenidos en las pruebas computacionales y, finalmente, en el Capítulo 6 se presentan algunas conclusiones.

1.1 El problema de planificación de horarios de clase

El problema de horarios de clase consiste en la asignación, sujeta a restricciones, de un número de sesiones de clase y recursos asociados (generalmente profesores y aulas), a un número limitado de períodos, de tal forma que cada sesión sea dictada por un profesor en un aula específica y en un período determinado dentro del horizonte de planificación, que generalmente es una semana. Existen diversas formas de modelizar este problema, y durante las últimas décadas se han propuesto algoritmos de solución tanto de tipo exacto como heurístico [7, 8, 9].

La asignación de las sesiones de clase está sujeta a restricciones fuertes y débiles. Las restricciones fuertes son aquellas que deben ser respetadas en cualquier solución admisible, por ejemplo:

- Durante cada período deben existir los recursos suficientes (aulas) para las sesiones de clase que fueron programadas en él.
- Tanto profesores como alumnos no pueden asistir a más de una sesión de clase a la vez.

Las restricciones débiles son aquellas que son deseables pero que no son fundamentales. Estas restricciones permiten evaluar la calidad de la solución obtenida. En la práctica es usualmente imposible encontrar horarios que satisfagan todas las restricciones débiles [7, 10]. Entre las restricciones débiles más comunes se tienen:

- Preferencias de horario de profesores.
- Prohibición de cruces de horario entre sesiones de clase correspondientes a ciertos pares de cursos que muchos alumnos tomarán simultáneamente.
- Restricciones de precedencia entre sesiones de diferentes cursos.
- Compacidad del horario para profesores y alumnos.

Cuando no se consideran restricciones débiles, el problema de horarios consiste en encontrar algún horario factible que satisfaga las restricciones fuertes, y puede ser formulado como un problema de búsqueda. En caso de incluir restricciones débiles, el problema de horarios es un problema de optimización combinatoria, y las restricciones débiles se formulan como parte de la función objetivo [11].

En un artículo de revisión ampliamente difundido en el área, Schaerf [8] clasifica a los problemas de horarios de clase en tres categorías:

- *Horarios de clase en la escuela:* En este problema, los alumnos están agrupados en clases, y cada clase recibe un conjunto de cursos. Las restricciones a considerarse son evitar los cruces de horario de profesores y respetar la disponibilidad de aulas.

- *Horarios de clase en la universidad:* Dado un número de aulas y períodos, el problema consiste en programar un conjunto de sesiones de clase. La principal diferencia con el problema de horarios en la escuela es que los cursos de universidad pueden tener estudiantes en común de distintos niveles (e incluso de distintas carreras). Dos cursos están en conflicto si tienen estudiantes en común, se busca asignar las sesiones de los cursos a períodos, de tal forma que se eviten cruces de horarios de los profesores, se respete la disponibilidad de aulas y en lo posible las sesiones de los cursos en conflicto no sean programadas simultáneamente.

- *Horarios de exámenes en la universidad:* El problema además de considerar las restricciones presentes en el problema de horarios de clase en la universidad debe tomar en cuenta que existe un examen de cada materia, que se requieren períodos libres entre dos exámenes y que no deben programarse muchos exámenes consecutivos para un mismo grupo de estudiantes.

1.2 Modelos y métodos de solución

En la mayoría de los casos, el problema de horarios puede verse como la generalización de problemas clásicos de la optimización sobre grafos, que se conoce pertenecen a la clase de problemas NP-difíciles. En particular, el problema de horarios de clase ha sido abordado como un problema de coloración de un grafo; el mismo consiste en asignar colores a sus nodos, de tal forma que dos nodos adyacentes no tengan el mismo color. Para el caso de la programación de horarios, los nodos del grafo representan las sesiones de clase, existe una arista entre dos nodos si las sesiones correspondientes están en conflicto, y los colores representan los períodos disponibles para la programación. En ciertas aplicaciones específicas, el problema de horarios de clase ha sido reducido a problemas de flujo en redes, con la finalidad de aprovechar los algoritmos eficientes de solución disponibles para este caso [8, 10, 11, 12]. Sin embargo, en este tipo de problemas sólo se pueden considerar restricciones muy simples, insuficientes para modelizar la mayoría de situaciones reales [13].

Existen diferentes enfoques para resolver el problema de horarios de clase. A modo general, los mismos pueden clasificarse en métodos heurísticos y algoritmos exactos [7, 8].

1.2.1 Heurísticas para programación de horarios

Las heurísticas son métodos de solución para problemas de optimización que buscan encontrar buenas soluciones factibles en corto tiempo de cálculo. Existen dos tipos de heurísticas [14]:

- *Heurísticas constructivas:* Son aquellas que construyen soluciones factibles "desde cero". Ejemplos de heurísticas constructivas para el caso de problemas de horarios son los métodos secuenciales, que cuentan entre las primeras estrategias de solución propuestas. Conocidos también como heurísticas directas, consisten en ordenar las sesiones de clase de acuerdo a algún criterio y luego programar sucesivamente cada sesión en un período en el cual no cause conflictos.

  Un criterio común que se emplea para ordenar las sesiones es su *conflictividad*, la misma que se define a partir de algún indicador de la dificultad de programar una sesión; por ejemplo, el grado de restricción de la disponibilidad de horario del profesor a cargo [7]. Por ejemplo, el sistema SCHOLA descrito en [15] está definido por el siguiente conjunto de reglas:

  – Programar la sesión más "difícil", aquella que tiene muchas restricciones, al período más favorable en el que pueda ser asignada es decir, aquel en el que se maximicen o minimicen los criterios considerados en la función objetivo del problema.
  – Si existe una sesión que puede ser asignada solamente a un período, programarla.
  – Si al terminar el algoritmo una sesión no ha podido ser programada, desprogramar otra(s) hasta lograr programarla.

  El inconveniente del método secuencial anterior es su dependencia del ordenamiento inicial de las sesiones: para las sesiones asignadas al final quedan generalmente muy pocos períodos disponibles, lo que puede producir soluciones cuya calidad no sea tan buena como se desearía. Para evitar esto, las sesiones pueden ser reordenadas con un cierto componente de aletoriedad, con la idea de repetir el proceso varias veces y seleccionar la mejor solución [16].

- *Heurísticas de mejoramiento:* El objetivo de las heurísticas de mejoramiento es modificar una o varias soluciones factibles conocidas, para obtener una nueva solución factible cuyo valor en la función objetivo sea mejor. Un caso prominente de

las heurísticas de mejoramiento lo constituyen las metaheurísticas o heurísticas de búsqueda local, que son métodos generales, aplicables a cualquier problema de optimización combinatoria. Algunos ejemplos de metaheurísticas son los algoritmos genéticos y la búsqueda Tabú [7, 8, 10, 14].

Para el problema de horarios, los algoritmos genéticos constituyen el tipo de metaheurística más estudiado. Estos algoritmos son heurísticas inspirados en los mecanismos de mutación y selección natural propuestos en la teoría de la evolución de las especies de Darwin. Los algoritmos genéticos comienzan con un conjunto de soluciones iniciales, en el caso del problema de horarios con un conjunto de horarios factibles, conocido como población inicial. Cada individuo de esta población (es decir, cada solución) es codificado como un cromosoma. Durante el proceso de mutación, se emplean distintos "operadores de cruce", para construir nuevas soluciones a partir de las soluciones existentes. Por ejemplo, el operador *Crossover* intercambia información genética de dos "soluciones padres" para crear una nueva solución cuyo valor sea mejor que el de sus predecesores. La selección natural se modeliza mediante la eliminación periódica de las soluciones con los peores valores en la función objetivo. Iterativamente, se van mejorando las poblaciones hasta alcanzar algún criterio de parada [17, 18, 19].

La búsqueda Tabú es otra metaheurística que ha sido exitosamente aplicada en algunos problemas de optimización combinatoria. En este caso, se define algún criterio de vecindad entre las soluciones factibles de un problema. Luego el problema de optimización original $P\text{-}OPT$ es reformulado como el problema general de explorar un grafo $\mathcal{G}$, cuyos nodos son las soluciones factibles de P-OPT y donde dos nodos son adyacentes si las soluciones correspondientes son vecinas, hasta encontrar el nodo que minimiza la función objetivo de P-OPT. (Esto explica el término de "búsqueda local"). La estrategia más simple consiste en el "descenso directo" (en un problema de minimización): seleccionar iterativamente el vecino con el menor valor en la función objetivo y terminar cuando todos los vecinos tengan un valor más alto que el nodo actual. Generalmente, esta estrategia conduce a un "mínimo local", asociado a una solución de P-OPT cuyo valor puede no ser satisfactorio.

Para evitar el estancamiento en un mínimo local, el algoritmo de búsqueda Tabú mantiene una lista, conocida como lista tabú, con nodos o movimientos que están prohibidos de ser visitados (o revisitados) durante un cierto número de iteraciones. El mecanismo básico de la búsqueda Tabú consiste en explorar suce-

sivamente la vecindad de la solución actual y determinar aquella nueva solución que posea el menor valor en la función objetivo. Para evitar mínimos locales se exige además que la solución a explorar no esté contenida en la lista tabú. Cada vez que un nuevo nodo de $\mathcal{G}$ es visitado, la solución correspondiente ingresa a la lista tabú por un cierto número de iteraciones [20, 21, 22].

1.2.2 Métodos exactos

Estos métodos parten de la formulación del problema como un modelo de programación lineal entera y realizan una exploración exhaustiva del espacio de búsqueda empleando alguna variante del método de ramificación y acotación (branch-and-bound).

Un programa lineal es un problema de optimización que consiste en la maximización o minimización de una función lineal, también conocida como función objetivo, sujeta a restricciones lineales [23].

Los orígenes de la programación lineal se remontan a la primera mitad del siglo XX, con el del trabajo de Leonid Kantorovich sobre asignación óptima de recursos en 1939 [24], aunque sistemas de desigualdades lineales habían sido estudiados previamente por Joseph Fourier, Carl Friedrich Gauss y otros en el siglo XIX. Sin embargo, la amplia difusión de la programación lineal empezó con un hito histórico que ocurrió poco después de la Segunda Guerra Mundial: en 1947, George Dantzig, quien durante la guerra había trabajado en el Pentágono en la planificación de la logística de misiones militares para la Fuerza Aérea de EEUU, formula esta tarea como un programa lineal y desarrolla el método del simplex para automatizar su solución. Hasta en la actualidad, el algoritmo del simplex es uno de los métodos más ampliamente utilizados para resolver programas lineales [25].

El período posterior a la Segunda Guerra Mundial estuvo marcado por un creciente interés en la automatización y optimización de procesos industriales, apoyado por el permanente desarrollo de los computadores electrónicos. Nacieron los campos de la programación matemática y la investigación de operaciones, enfocados en el desarrollo y estudio de modelos matemáticos de optimización para un sinnúmero de aplicaciones prácticas [23].

La optimización combinatoria es una rama de la matemática aplicada que combina técnicas de combinatoria, programación lineal, y teoría de algoritmos para resolver problemas de optimización sobre estructuras discretas [26]. Muchos de los problemas de optimización combinatoria provienen de aplicaciones de la vida real que ocurren a diario; por ejemplo, enrutamiento de vehículos, diseño de redes de telecomunicaciones,

control de inventarios, calendarización de máquinas, asignación de tareas a empleados, entre otros. Algunos problemas de optimización combinatoria fueron estudiados ya hace un par de siglos: por ejemplo, Monge [27] consideró en 1784 el problema de transportar tierra desde un terreno a otro intentado que el desplazamiento a nivel microscópico sea mínimo [28]. Sin embargo, al igual que la programación lineal y entera, el campo de la optimización combinatoria experimenta un desarrollo acelerado en la segunda mitad del siglo XX.

En general, un programa lineal mixto puede formularse como se indica a continuación [14]:

Definición 1. *Sean* m, $n \in N$, $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, l, $u \in \hat{\mathbb{Q}}^n$, *con* $\hat{\mathbb{Q}} := \mathbb{Q} \cup \{\pm\infty\}$, *y* $I \subseteq N = \{1, ..., n\}$. *El problema*

$$\begin{aligned} \text{mín} \quad & c^T x \\ \text{s.t.} \quad & Ax = b, \\ & l \leq x \leq u, \\ & x_j \in \mathbb{Z} \text{ para todo } j \in I, \end{aligned}$$

es llamado un programa lineal mixto.

Se conoce a $c^T x$ *como la función objetivo del problema;* l *y* u *se denominan límites superior e inferior, respectivamente, de las variables de decisión* x.

Definición 2. *El programa anterior se conoce específicamente como:*

- *Programa lineal si* $I = \emptyset$.
- *Programa entero si* $I = N$.
- *Programa binario, si* $l = \mathbf{0}$, $u = \mathbf{1}$, $I = N$.

El algoritmo *branch-and-bound* (ramificación y acotación) es un método general para resolver problemas de optimización discretos que consiste en dividir sucesivamente el problema dado en subproblemas más pequeños, mediante el particionamiento del espacio de solución en dos o más regiones, y empleando un mecanismo que permita verificar a priori si es necesario explorar cada una de las mismas. El mecanismo generalmente utilizado consiste en acotar la función objetivo a optimizar sobre cada región del problema. Así, conforme la exploración avanza, algunas regiones pueden ser descartadas si el valor de sus cotas es peor que el valor objetivo de alguna solución factible [29, 30, 31].

Bertsimas y Tsitsiklis [31] describen el algoritmo de branch-and-bound de la siguiente forma:

Sea F el conjunto de soluciones factibles del problema

$$\begin{aligned} \text{mín} \quad & c^T x \\ s.t. \quad & x \in F. \end{aligned}$$

El conjunto F puede ser dividido en un número finito de conjuntos $F_1, ..., F_k$ y se puede resolver independientemente cada uno de los subproblemas:

$$\begin{aligned} \text{mín} \quad & c^T x \\ \text{s.t.} \quad & x \in F_i. \end{aligned}$$

Luego, comparamos las soluciones óptimas de los subproblemas y elegimos la mejor (cada uno de los subproblemas puede ser tan difícil como el problema original). De manera similar, cada uno de los subproblemas que no pueda ser resuelto directamente es dividido en otros subproblemas. Este paso del algoritmo es conocido como *branching* (o ramificación). Durante el transcurso del algoritmo se genera un árbol de exploración (ver Figura 1.1) donde cada nodo representa un subproblema. El nodo raíz del árbol corresponde al problema original y las hojas son los subproblemas resueltos.

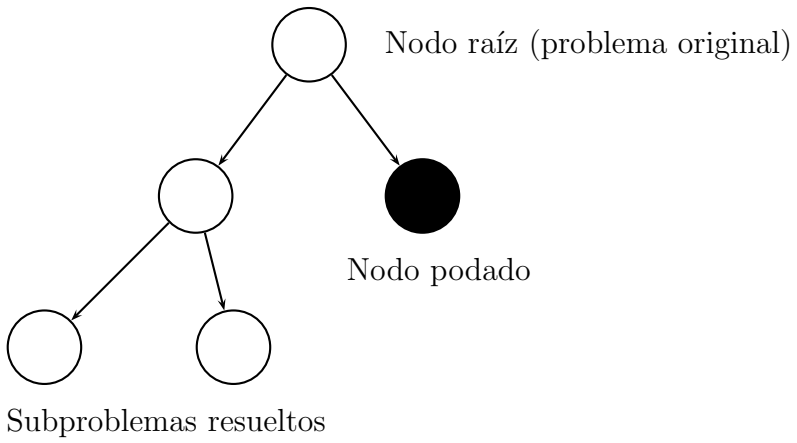

Figura 1.1: Árbol de exploración del algoritmo de branch-and-bound

La otra componente del algoritmo es el acotamiento o *bounding*. El mismo consiste en determinar cotas inferiores B_i para la función objetivo sobre los subespacios F_i. El valor B_i puede ser encontrado, para el caso de programas enteros, al resolver la relaja-

ción lineal del subproblema, la cual se obtiene al omitir las restricciones de integralidad de las variables.

En el transcurso del algoritmo, algunos subproblemas pueden ser resueltos hasta la optimimalidad, lo que permite obtener una o más soluciones factibles. Llamaremos *solución incumbente* a la mejor solución factible encontrada hasta una determinada iteración. Notar que su valor U es una cota superior válida para el óptimo del problema original. Si la cota inferior B_i para un subproblema F_i satisface que $B_i \geq U$, entonces el subproblema ya no requiere ser considerado, dado que cualquier solución factible de F_i no podrá mejorar la solución incumbente. El cálculo de cotas es importante porque puede servir para "podar" gran parte del árbol de búsqueda.

El algoritmo general de branch-and-bound tiene la siguiente estructura:

1. Calcular U, de ser posible, a partir de alguna solución factible obtenida heurísticamente o inicializar $U := \infty$.

2. Inicializar la cola de subproblemas por procesar $Q := \{F\}$ con el problema original.

3. Mientras $Q \neq \emptyset$ hacer:

4. Seleccionar un subproblema $F_i \in Q$ y retirarlo de la cola.

5. Si el subproblema es infactible, borrarlo y volver a 3; si no, calcular B_i (acotamiento).

6. Si $B_i \geq U$, borrar el subproblema (poda el árbol) y volver a 3.

7. Si $B_i < U$, de ser posible, obtener la solución óptima del subproblema o dividirlo en otros subproblemas, más pequeños $F_{i_1}, F_{i_2}, ..., F_{i_k}$ (ramificación).

8. Insertar $F_{i_1}, F_{i_2}, ..., F_{i_k}$ en Q y volver a 3.

Por ejemplo, en el caso de un programa entero, el paso de acotamiento puede consistir en resolver la relajación lineal de un subproblema. Si la solución óptima x^* de la relajación lineal no es entera, entonces escogemos una variable x_i tal que x_i^* no sea entera y creamos dos nuevos subproblemas, agregando, a cada uno de ellos una de las dos siguientes restricciones:

$$x_i \leq \lfloor x_i^* \rfloor \qquad o \qquad x_i \geq \lceil x_i^* \rceil.$$

Está claro que toda solución entera debe satisfacer una de estas dos restricciones.

El método branch-and-bound se desarrolla durante la década de 1950 y de acuerdo a [30] tiene 3 orígenes: En 1957, la noción de branch-and-bound es descrita por Harry Markowitz y Alan Mannespero en un trabajo publicado en Econometrica [32] pero el método no es detallado formalmente como un algoritmo. En 1958, Willard Eastman publica su tesis doctoral en Harvard [33], en la que el autor diseña algoritmos para resolver varios modelos, incluyendo el problema del agente viajero. En 1960, Alisa Land y Alison Doing [34] presentan el método como un algoritmo para implementar códigos computacionales exitosos para resolver problemas de programación entera. El nombre "branch-and-bound" fue acuñado por Jhon Litte, Katta Murty, Dura Sweeney y Carlin Karel en [35]. El trabajo realizado por Markowintz y Mannespero incluye la idea de mejorar las cotas inferiores obtenidas por la relajación lineal añadiendo inecuaciones lineales, también llamados planos cortantes. Los planos cortantes son sistemáticamente utilizados en 1958 en el algoritmo de programación entera de Ralph Gomory [36]. La combinación del algoritmo branch-and-bound y planos cortantes se convirtió a partir de mediados de la década de los noventa en el procedimiento más utilizado para resolver problemas de programación entera mixta. Este algoritmo recibe usualmente el nombre de "branch-and-cut" (ramificación y corte).

La programación entera es un marco flexible para modelar muchos problemas de optimización discreta. Para el caso particular del problema de horarios, en [1] se propone un método de ramificación y corte para el problema de programación de horarios de cursos de Udine. Las instancias de la Universidad de Udine se han establecido como un referente para probar el rendimiento de algoritmos de planeación de horarios, a partir de que fueron utilizadas en la *Segunda Competencia Internacional de Horarios* (Second International Timetabling Competition) celebrada el 2007. El conjunto comprende 14 instancias, cuyo tamaño varía de 30 a 131 cursos, lo que corresponde a entre 160 y 434 sesiones de clase, y entre 13 a 150 currículums (que representan preinscripciones de estudiantes en un conjunto de cursos).

Las restricciones fuertes del modelo formulado en [1] son las siguientes:

- Para cada curso se deben programar todas sus sesiones de clase.
- Dos cursos no pueden ser dictados en la misma aula simultáneamente.
- Dos cursos que son dictados por el mismo profesor o pertenecen a un mismo currículums no puede ser programados simultáneamente.
- No se pueden asignar sesiones de clase a períodos donde el profesor no estará disponible.

Las restricciones débiles del modelo son:

- Se penaliza el número de estudiantes sin asiento durante una clase.
- Para cada curso, se indica un número de *días de instrucción* en los que deben estar distribuidos sus sesiones de clase. Se penaliza si las sesiones son dictadas en un número de días inferior al indicado.
- Se penaliza la existencia de "horas huecas" durante un día en un currículum. Con esta restricción se busca obtener horarios compactos.
- En lo posible, se espera que las sesiones de clase de un curso sean dictadas en la misma aula.

Asociada a cada una de las restricciones débiles se definen variables auxiliares para aplicar las penalizaciones, cuyos valores son calculados a partir de las variables de asignación. El trabajo se centra en el desarrollo de planos cortantes, los cuales son obtenidos a partir de las restricciones de vinculación entre las variables de asignación y las variables auxiliares. Los planos cortantes *Tipo 1* se obtienen de la restricción que calcula las variables para la restricción de compacidad; y los planos cortantes *Tipo 2* se obtienen de la restricción fuerte que establece que para cada curso todas sus sesiones de clase deben ser dictadas una vez, así como la restricción que vincula las variables asociadas a los números de días de instrucción. En la Tabla 1.1 se resumen la descripción de las instancias de la universidad de Udine y las cotas inferiores obtenidas con el empleo de estos planos cortantes, comparadas con valores de otros trabajos previos.

En [2] se describe un modelo de programación lineal implementado en la Escuela de Economía y Administración de la Universidad Hannover en Alemania. Fueron programadas aproximadamente 150 sesiones de clase, tutoriales y seminarios para entre 5 y 650 estudiantes, dictados por 100 profesores. El problema de decisión formulado consiste en asignar cursos a períodos de tiempo y aulas de tal forma que se cumplan todas las restricciones fuertes y de ser posible, la mayor cantidad de restricciones débiles. El problema tiene las siguientes características: cada curso c es dictado en uno o varios grupos de enseñanza g. Cada curso tomado por un grupo g requiere uno o varios profesores, si se requiere un conjunto de profesores para dicho grupo, todos ellos deben ser asignados al mismo tiempo y a la misma aula. Los profesores indican su horario de preferencia, y pueden requerir descansos entre horas de clase. Además, establecen un número máximo de horas de clase por día y pueden sugerir días consecutivos de

Tabla 1.1: Descripción de las instancias del problema de horarios de la Universidad Udine. Cotas inferiores obtenidas con los planos cortantes propuestos en [1] y comparación con trabajos anteriores. *Fuente:* [1]

						Cotas Inferiores				**Cota Superior**
Instancia	**Aulas**	**Períodos**	**Cursos**	**Sesiones**	**Currículums**	**Tipo 1**	**Tipo 1 + 2**	**Burke et al. [37]**	**Lanch and Lübbecke [38]**	**Müller [39]**
comp01	6	30	30	160	14	4	4	5	4	5
comp02	16	25	82	283	70	0	0	6	8	51
comp03	16	25	72	251	68	0	0	43	23	84
comp04	18	25	79	286	57	0	0	2	27	37
comp05	9	36	54	152	139	99	95	183	101	330
comp06	18	25	108	361	70	0	0	6	7	48
comp07	20	25	131	434	77	0	0	0	0	20
comp08	18	25	86	324	61	0	0	2	34	41
comp09	18	25	76	279	75	0	0	0	40	109
comp10	18	25	115	370	67	0	0	0	4	16
comp11	5	45	30	162	13	0	0	0	0	0
comp12	11	36	88	218	150	0	0	5	32	333
comp13	19	25	82	308	66	1	3	0	37	66
comp14	17	25	85	275	60	0	0	0	41	59

enseñanza para descansar el resto de la semana. El modelo formulado tiene cerca de 100.000 variables binarias.

Las restricciones débiles del modelo son las siguientes: satisfacer la preferencia de los profesores, utilizar la capacidad adecuada de las aulas y penalizar el número de estudiantes sin asiento durante el desarrollo de las actividades de docencia. El modelo fue implementado en el Solver COIN-CBC [40], fue resuelto en 248.34 s. hasta la optimalidad, y se obtuvo un horario de buena calidad para el semestre de verano del año 2006.

En [41] se presentan varios enfoques para programar horarios post matriculación propuestos para la *Segunda Competencia Internacional de Horarios*: se exhibe un enfoque utilizando búsqueda local y técnicas de programación con restricciones (constraint programming) utilizando un método original de descomposición. En [42] se formulan dos modelos de programación lineal entera para un problema de programación de horarios para universidades cuyo objetivo es minimizar la asignación en períodos no deseados, balanceando la carga de trabajo diaria para cada grupo de alumnos.

Finalmente, en [11] se propone un modelo básico de planificación de horarios que exponemos a continuación con más detalle, por cuanto el mismo captura la idea fundamental del problema y sirve como punto de partida para otros modelos, incluyendo el que desarrollaremos en esta tesis.

Sean $C_1, ..., C_q$ los cursos a programar donde el curso C_i tiene k_i sesiones. Existen r conjuntos de cursos $S_1, ..., S_r$ que tienen estudiantes en común; por tanto, todas las sesiones de los cursos que pertenecen a S_l, $\forall l \in 1, ..., r$, deben ser programadas en períodos distintos. Sea p el número de períodos y l_k el número máximo de sesiones que pueden ser programadas en el período k, dado por el número de aulas disponibles en el período k. Por ultimo, sea d_{ik} un indicador del nivel de conveniencia para dictar una sesión del curso C_i en el período k, donde valores más altos indican una mayor conveniencia.

Las variables de decisión binarias están dadas por:

$$y_{ik} = \begin{cases} 1, & \text{si una sesión de } C_i \text{ es programada en el período } k, \\ 0, & \text{caso contrario.} \end{cases}$$

El modelo es el siguiente:

$$\text{máx} \sum_{i=1}^{q} \sum_{k=1}^{p} d_{ik} y_{ik}$$

$$s.t.$$

$$\sum_{k=1}^{p} y_{ik} = k_i, \qquad \forall i = 1, ..., q, \tag{1.1}$$

$$\sum_{i=1}^{q} y_{ik} \leq l_k, \qquad \forall k = 1, ..., p, \tag{1.2}$$

$$\sum_{C_i \in S_l} y_{ik} \leq 1, \qquad \forall l = 1, ..., r, \quad k = 1, ..., p, \tag{1.3}$$

$$y_{ik} \in \{0, 1\}, \qquad i = 1, ..., q, \quad k = 1, ..., p. \tag{1.4}$$

La función objetivo a maximizar mide el nivel de conveniencia del horario programado para los cursos. La restricción (1.1) especifica que deben programarse exactamente k_i sesiones para el curso C_i, $1 \leq i \leq q$; (1.2) controla la disponibilidad de aulas; (1.3) evita cruces entre cursos que tengan estudiantes en común; y (1.4) define las variables como binarias.

Capítulo 2

Modelo de programación lineal entera para el cálculo de horarios en la Facultad de Ciencias

2.1 Descripción del problema

En el presente trabajo abordamos la tarea de modelar el problema de horarios de clase considerando las particularidades de la Facultad de Ciencias de la Escuela Politécnica Nacional. La Facultad de Ciencias es la unidad académica encargada de llevar adelante las actividades de investigación y docencia en las áreas de la física, la matemática y la economía. Agrupa a los Departamentos de Física y Matemática; y gestiona las carreras de Física, Matemática, Ingeniería Matemática e Ingeniería en Ciencias Económicas y Financieras.

Como se señaló en el capítulo anterior, la planificación manual de horarios de clase es una tarea compleja y que requiere de mucho tiempo, debido, entre otras razones, a las diversas interacciones que existen al momento de programar las sesiones de clase, sobre todo si se tiene en cuenta que en el contexto universitario un mismo curso puede agrupar estudiantes de distintos niveles, pénsums y carreras. Por tanto, resulta muy conveniente la implementación de un modelo matemático de optimización para automatizar dicha tarea y alcanzar soluciones de buena calidad.

El horizonte temporal para la planificación de las actividades de docencia es de una semana, con seis días laborables y un determinado conjunto de períodos cada día. Un *período* es una hora de clase cuya duración en el caso de la Escuela Politécnica Nacional está establecida en 60 minutos. Un *curso* identifica a un grupo de estudiantes que re-

ciben una asignatura específica, impartida por un profesor en un horario determinado. Asociados a un curso se definen su número estimado de estudiantes, su configuración semanal (número de sesiones de clase, y número de períodos por cada sesión), el profesor a cargo del mismo y un código de grupo. En la Facultad de Ciencias, los cursos suelen identificarse por un nombre de asignatura y un código de grupo. Por ejemplo, usualmente se dictan tres cursos de Cálculo en una Variable, los que son identificados como GR1, GR2 y GR3, respectivamente. Una *sesión* de clase se refiere a cada una de las reuniones semanales en las cuales se imparten las actividades de docencia de un curso. Asociada a cada sesión están su duración (medida en períodos) y el tipo de aula requerido. Por ejemplo, el curso de Cálculo en una Variable generalmente es configurado para ser impartido en 3 sesiones semanales, cada una con una duración de 2 períodos. Además, al tratarse de una asignatura básica, es común que los grupos sean numerosos por lo que las sesiones deben ser programadas en aulas de gran capacidad. Las *materias* o asignaturas que un estudiante debe aprobar durante su carrera están organizadas en un pénsum. Un pénsum asigna a cada materia un nivel, y establece pre- y correquisitos entre materias. Por ejemplo, Cálculo Vectorial puede tomarse únicamente después de haber aprobado Cálculo en una Variable (prerrequisito); o Laboratorio de Óptica debe tomarse conjuntamente con Óptica (correquisito). Cada pénsum se identifica mediante un código. Cursos correspondientes a materias de un mismo nivel referencial no pueden ser programados en horarios simultáneos. Materias con nombres distintos en las diferentes carreras y pénsums pueden referirse al mismo contenido académico y ser impartidas en cursos comunes. Así, en un mismo curso pueden estar presentes estudiantes de la asignatura "Física"del pénsum 2012 de la carrera de Ingeniería Matemática, y estudiantes de la asignatura "Mecánica Newtoniana"del pénsum 2012 de la carrera de Física. Estas relaciones de equivalencia entre materias están dadas por tablas de equivalencia, empleadas por las autoridades a cargo del proceso de planificación.

Dentro del horizonte de planificación semanal, los profesores especifican su horario de disponibilidad para las actividades de docencia, el mismo que debe incluir 40 horas semanales para los profesores titulares con dedicación a tiempo completo, o 20 horas semanales para aquellos a medio tiempo. Por otra parte, ciertos profesores laboran exclusivamente bajo la modalidad de contrato de docencia y pueden tener horarios de disponibilidad más restringidos. Los profesores pueden especificar, dentro del horario de disponibilidad, su *preferencia* para dictar clases en ciertos períodos.

La planificación académica en la Facultad de Ciencias se realiza de forma semestral, e involucra las siguientes actividades:

1. Se definen las materias a ser ofertadas en el semestre en planificación.
2. Se proyecta el número de estudiantes esperado para cada materia.
3. Se establece el número de cursos a ofertarse por cada materia.
4. Se asignan profesores a los diferentes cursos, proceso que se conoce como *asignación académica*.
5. Se configuran los cursos, es decir, se establecen para cada curso el número de sesiones, la duración de cada sesión y el tipo de aula (pequeña, grande, laboratorio) requerido.
6. Se programan los horarios de clase.
7. Los estudiantes se inscriben en los cursos.

El modelo aquí presentado aborda el sexto paso de la planificación, la programación de los horarios de clase. La misma consiste en asignar cada sesión de clase a un período, un día de la semana, y un aula de clase determinados, de manera que se satisfagan todas las restricciones fuertes y la mayor cantidad posible de restricciones débiles del modelo.

Para la formulación del modelo se han considerado las siguientes restricciones fuertes:

- Ningún profesor puede asistir a más de una sesión de clase por período a la vez, y se debe respetar su disponibilidad de horario.
- No pueden existir cruces de horario entre sesiones de clase pertenecientes a cursos de un mismo nivel de un pénsum.
- Cada sesión de clase debe ser programada en un aula adecuada.
- Durante cada período deben existir las aulas suficientes para albergar a todas las sesiones programadas.
- Para cada curso, puede dictarse máximo una sesión por día.

Por otra parte, se han incluido las siguientes restricciones débiles:

- Satisfacer las preferencias de horario de los profesores.
- Penalizar cruces de horario entre cursos correspondientes a materias de niveles aledaños.

2.2 Especificación de parámetros

Definimos a continuación diferentes conceptos y parámetros requeridos para la formulación del modelo:

El conjunto K de días contiene los días en los que se pueden planificar las sesiones de clase. El conjunto H de períodos está conformado por los períodos considerados en el horizonte de planificación. En el caso de la Facultad de Ciencias se consideran 6 días de la semana (desde el lunes hasta el sábado) y 13 períodos diarios (desde las 7:00 hasta las 21:00).

El conjunto M de *materias* agrupa las asignaturas ofertadas en el semestre en planificación. El conjunto C de *cursos* incluye los cursos relacionados con estas materias. En el conjunto S de *sesiones* se enlistan las sesiones de clase a dictar. La duración de una sesión $i \in S$ (es decir, el número de períodos requeridos por la misma) se representa con δ_i. Denotaremos con S_c el conjunto de sesiones de clase correspondientes al curso $c \in C$. Sin pérdida de generalidad, asumiremos en adelante que los elementos de todos los conjuntos señalados han sido etiquetados con número enteros, empezando desde cero. Así, si al primer curso en C le corresponden las primeras tres sesiones en S, y cada sesión tiene una duración de 2 períodos, indicaremos con esto por $S_0 = \{0, 1, 2\}$, $\delta_0 = \delta_1 = \delta_2 = 2$.

Dadas dos materias m_1 e $m_2 \in M$,

$$p_{m_1 m_2} = \begin{cases} 1, & \text{si el pénsum de } m_1 \text{ es igual al pénsum de } m_2 \\ & \text{y existe una cadena de prerrequisitos de } m_1 \text{ a } m_2, \\ 0, & \text{caso contrario.} \end{cases}$$

El indicador de precedencia se usa para definir el concepto de *distancia* entre materias, el mismo que juega un papel fundamental en nuestro modelo de optimización. Si dos materias m_1, m_2 comparten un mismo pénsum, la distancia $d^M_{m_1 m_2}$ entre ambas es la diferencia en valor absoluto de los niveles a los que pertenecen, a excepción de que exista una cadena de prerrequisitos de m_1 hasta m_2. En este último caso, o si las materias no pertenecen a un mismo pénsum, su distancia se define como $+\infty$:

$$d^M_{m_1 m_2} = \begin{cases} |\text{nivel}(m_1) - \text{nivel}(m_2)|, & \text{si } p_{m_1 m_2} = 0 \text{ y el pénsum de } m_1 \\ & \text{es igual al pénsum de } m_2, \\ +\infty, & \text{caso contrario.} \end{cases}$$

En el modelo prohibimos cruces de horarios entre materias cuya distancia sea cero y penalizamos los cruces entre las demás materias según la distancia entre ellas (cruces entre materias de menor distancia reciben una mayor penalización). Si la distancia entre dos materias es $+\infty$, entonces los cruces entre ambas no son penalizados. Observar que de acuerdo a la definición anterior, esto puede ocurrir en dos casos:

- Si las dos materias no pertenecen a un mismo pénsum;
- si existe una cadena de prerrequisitos entre una materia y la otra, en cuyo caso ningún alumno puede cursarlas simultáneamente.

La matriz $D^M \in (\mathbb{Z} \cup \infty)^{M \times M}$ contiene las distancias entre todas las materias.

La matriz de distancia entre cursos D^C se calcula a partir de la distancia entre materias. Un aspecto a tomar en cuenta es que cada curso $c \in C$ puede estar asociado a diferentes materias en los distintos pénsums y carreras (cuyos alumnos reciben clases en forma compartida). La distancia $d^C_{c_1 c_2}$ entre dos cursos $c_1, c_2 \in C$ se define como el mínimo de las distancias entre las materias relacionadas, es decir, si M_c es el conjunto de materias asociadas a un curso $c \in C$, entonces:

$$d^C_{c_1 c_2} = \text{mín}\{d^M_{m_1 m_2} | m_1 \in M_{c_1} \text{ y } m_2 \in M_{c_2}\}.$$

De manera similar, se define a la distancia $d_{i\ell}$ entre dos sesiones $i, \ell \in S$ como la distancia entre los cursos a los que pertenecen. La matriz D contiene las distancias entre sesiones.

Dos sesiones son *conflictivas* si son dictadas por el mismo profesor o si su distancia es cero. La matriz $A \in \{0,1\}^{S \times S}$ de conflictos entre sesiones se define por:

$$a_{i\ell} = \begin{cases} 1, & \text{si } d_{i\ell} = 0 \text{ ó } i \text{ e } \ell \text{ son dictadas por el mismo profesor,} \\ 0, & \text{caso contrario.} \end{cases}$$

Una de las restricciones del modelo prohíbe la programación simultánea de sesiones de clase conflictivas, con lo que se garantiza el cumplimiento de las dos primeras restricciones fuertes citadas anteriormente.

Al programar cada sesión es necesario considerar la disponibilidad de horario del profesor que imparte el curso respectivo. Con esta finalidad, se define para cada $i \in S$ y cada día $k \in K$ el conjunto L_{ik} como el conjunto de períodos factibles para el inicio de i en el día k, es decir, aquellos en los que se puede programar el inicio de la sesión de

tal manera que el profesor que la dicta disponga de tiempo suficiente para su ejecución:

$$L_{ik} := \{j \in H | \text{ la sesión } i \text{ puede comenzar a ser dictada en el período } j \text{ el día } k\}.$$

Para cada $j \in L_{ik}$ se calcula el parámetro ω_{ijk} que representa el grado de cumplimiento de las preferencias de horario del profesor que dicta la sesión i, si la misma es programada para empezar en el período h del día k (valores altos indican un menor cumplimiento de las preferencias de horario). El parámetro ω_{ijk} se calcula como la suma de los valores de preferencia de horario indicados por el profesor, sobre los períodos en los que se dictaría la sesión i, si la misma empieza en el período h del día k.

Una sesión debe ser asignada a un aula de clase que cumpla con las condiciones necesarias para su desarrollo, como son capacidad y tipo (aula de clase o laboratorio de computación) adecuados. Para ello, se define el concepto de *aulas genéricas* R. Un aula genérica representa a un conjunto de aulas que comparten características similares, como la capacidad y la infraestructura disponible. El conjunto R contiene todas las aulas genéricas disponibles, y para cada $i \in S$ el conjunto $R_i \subseteq R$ es el conjunto de aulas genéricas en las que puede ser programada la sesión i, debido a que éstas cumplen con los requisitos de infraestructura necesarios.

Para controlar la disponibilidad de aulas se define el parámetro N_{rkj} para todo $r \in R$, $k \in K$ y $j \in H$ como el número de aulas del tipo especificado por el aula genérica r, disponibles el día k en el período j. La disponibilidad puede variar de un día a otro y entre los distintos períodos en un día, debido a que la infraestructura es compartida con otras facultades.

2.3 Formulación del modelo

El objetivo del modelo es asignar cada sesión a un período, un día y a un aula genérica, de tal forma que se satisfagan todas las restricciones fuertes y la mayor cantidad posible de restricciones débiles descritas en la sección anterior. Para ello, emplearemos las siguientes variables de decisión binarias:

$$x_{irkj} = \begin{cases} 1, & \text{si la sesión } i \in S \text{ es programada en el aula genérica } r \in R_i, \\ & \text{para empezar en el período } j \in L_{ik} \text{ en el día } k \in K, \\ 0, & \text{caso contrario.} \end{cases}$$

Emplearemos además variables auxiliares $y_{i\ell} \in \{0,1\}$ para indicar la existencia de cruces entre sesiones de clase:

$$y_{i\ell} + y_{\ell i} = \begin{cases} 1, & \text{si existe un cruce de horario entre las sesiones } i \text{ y } \ell, \\ 0, & \text{caso contrario.} \end{cases}$$

El modelo de programación lineal entera que proponemos para la planificación de horarios es el siguiente (PLAN-HOR):

$$\text{mín} \sum_{i\in S}\sum_{\ell\in S}\gamma(d_{i\ell})y_{i\ell} + \sum_{i\in S}\sum_{r\in R_i}\sum_{k\in K}\sum_{j\in L_{ik}}\omega_{ijk}x_{irkj}$$

s.r.

$$\sum_{r\in R_i}\sum_{k\in K}\sum_{j\in L_{ik}} x_{irkj} = 1, \qquad \forall i \in S, \tag{2.1}$$

$$\sum_{i\in S}\sum_{\substack{\ell\in S\\ \ell\neq i}} a_{i\ell}y_{i\ell} = 0, \tag{2.2}$$

$$\sum_{i\in S_c}\sum_{r\in R_i}\sum_{j\in L_{ik}} x_{irkj} \leq 1, \qquad \forall c \in C, \forall k \in K, \tag{2.3}$$

$$\sum_{i:r\in R_i}\sum_{\hat{j}\in\Omega_{ijk}} x_{irk\hat{j}} \leq N_{rkj}, \qquad \forall r \in R, \forall k \in K, \forall j \in H, \tag{2.4}$$

$$\sum_{r\in R_i} x_{irkj} + \sum_{r\in R_\ell}\sum_{\hat{j}\in\Gamma_{i\ell jk}} x_{\ell rk\hat{j}} - 1 \leq y_{i\ell}, \qquad \begin{array}{l}\forall i, \ell \in S, \ell \neq i, \\ \forall k \in K, \forall j \in L_{ik},\end{array} \tag{2.5}$$

$$\sum_{r\in R_i} x_{irkj} + \sum_{r\in R_\ell} x_{\ell rkj} - 1 \leq y_{i\ell}, \qquad \begin{array}{l}\forall i, \ell \in S\ \ell < i, \\ \forall k \in K, \forall j \in L_{ik} \cap L_{\ell k},\end{array} \tag{2.6}$$

$$x_{irkj} \in \{0,1\}, \qquad \forall i \in S, \forall r \in R_i, \forall k \in K, \forall j \in L_{ik}, \tag{2.7}$$

$$y_{i\ell} \in \{0,1\}, \qquad \forall i, \ell \in S. \tag{2.8}$$

La función objetivo mide las penalizaciones por cruces de horario de sesiones de clase que pertenecen a cursos de niveles aledaños, y por la violación de las preferencias de horario de los profesores. El peso asignado a un cruce de horario de dos sesiones $i, \ell \in S$ depende de la distancia $d_{i\ell}$ entre ambas y es igual a $\gamma(d_{i\ell})$, donde γ es una función decreciente. En la implementación computacional hemos utilizado

$$\gamma(d_{i\ell}) := \begin{cases} \frac{10}{d_{i\ell}}, & \text{si } 0 < d_{i\ell} < +\infty, \\ 0, & \text{caso contrario.} \end{cases}$$

Recordar que los cruces entre sesiones con distancia cero están prohibidos por las restricciones fuertes. Las penalizaciones por el incumplimiento de preferencia de horario de los profesores están en los parámetros ω_{ijk} descritos en la sección anterior.

Cada sesión debe ser dictada exactamente una vez a la semana, en algún aula adecuada, lo que se expresa en las restricciones (2.1). Las restricciones (2.3) establecen que para cada curso se dicte máximo una sesión de clase por día. La restricción (2.2) prohíbe la programación simultánea de sesiones de clase conflictivas. La familia de restricciones (2.4) refleja la disponibilidad del número de aulas tipo. El conjunto $\Omega_{ijk} := \{\hat{j} \in L_{ik} | j - \delta_i + 1 \leq \hat{j} \leq j\}$, utilizado en esta familia indica los períodos del día k en los que puede haber empezado la sesión i para que la misma se esté dictando durante el período j.

Las restricciones (2.5) y (2.6) vinculan los valores de las variables de asignación x_{irkj} con las variables auxiliares $y_{i\ell}$ para indicar la existencia de cruces entre sesiones de clase. Recordemos que $y_{i\ell}$ debe tomar el valor de 1 si existe un cruce de horario entre las sesiones i e ℓ. Notar que la existencia de cruces puede determinarse a partir de los valores de las variables de asignación x_{irkj}:

$$y_{i\ell} + y_{\ell i} = 1 \Leftrightarrow x_{ir_1jk} + x_{\ell r_2\ \hat{j}k} = 2,$$

para algún $r_1, r_2 \in R, k \in K, j \in L_{ik}$ y $\hat{j} \in \Gamma_{i\ell jk}$.

El conjunto $\Gamma_{i\ell jk} := \{\hat{j} \in L_{\ell k} | j + 1 \leq \hat{j} \leq j + \delta_i - 1\}$ indica los períodos del día k en los que puede empezar la sesión ℓ para que exista un cruce con la sesión i que empezó previamente en el período j. Cada una de las sumas que componen los lados izquierdos de las restricciones (2.5) y (2.6) están acotadas por 1 según se sigue de (2.1). Consideremos primero una restricción de la familia (2.5). Si una sesión i fue programada en el período $j \in L_{ik}$ del día k en alguna aula $r \in R_i$ entonces:

$$\sum_{r \in R_i} x_{irkj} = 1,$$

existen dos posibilidades para el valor de la segunda suma. En el caso de que la sesión $\ell \neq i$ no fue programada para iniciar en alguno de los períodos que pertenecen al conjunto $\Gamma_{i\ell jk}$, la suma vale cero y por tanto el lado izquierdo de la restricción se

evalúa a 0, de tal forma que se obtiene la restricción redundante:

$$y_{i\ell} \geq 0.$$

Por el contrario, si la sesión ℓ empieza en algún período $\hat{j} \in \Gamma_{i\ell jk}$ el lado izquierdo de (2.5) toma el valor de 1 y de esta forma el valor de la variable $y_{i\ell}$ es forzado a ser 1. Las restricciones (2.6) contabilizan los cruces cuando i y ℓ empiezan en el mismo período. La condición $\ell < i$ sirve para evitar contar este tipo de cruces por duplicado. Finalmente las restricciones (2.7) y (2.8) definen las variables de decisión como binarias.

El siguiente resultado establece la complejidad computacional del modelo:

Teorema 1. *PLAN-HOR es NP-difícil.*

Demostración. Probaremos que el problema de *k-coloración de grafos*, que se conoce es NP-completo [43], puede reducirse al problema de encontrar una solución factible para PLAN-HOR. El problema de k-coloración de grafos consiste en asignar a los nodos de un grafo no dirigido un máximo de k colores de tal forma que dos nodos adyacentes no sean coloreados del mismo color.

Considerar una instancia del problema de k-coloración, dada por un grafo no dirigido $G = (V, E)$ y un número natural $k \in \mathbb{N}$. Dicha instancia puede reducirse a una instancia del problema PLAN-HOR con las siguientes características: el horizonte de planificación es un día y k períodos. Los nodos representan sesiones de clase, es decir, existen $|V|$ sesiones, cada una de ellas dura un período y cada curso tiene una sola sesión de clase. Para cada arista $uv \in E$, las sesiones correspondientes a los nodos adyacentes $u, v \in V$ son considerados como sesiones conflictivas. Existen $|V|$ aulas del mismo tipo (suficientes para programar todas las sesiones simultáneamente).

Dada una solución factible de la instancia transformada ésta puede usarse para definir una solución de la instancia original, identificando el período asignado a cada sesión con un color para el nodo correspondiente. En efecto, se utilizan a lo más k colores porque las sesiones pueden ser planificadas a lo más en k períodos, y la condición de que dos nodos adyacentes no sean coloreados del mismo color se cumple dado que dos sesiones conflictivas no pueden ser programadas en el mismo período (recordar que si existe una arista entre dos nodos u y v las sesiones correspondientes son conflictivas). Por otra parte, dada una solución factible de la instancia original, ésta define una solución de la instancia transformada ya que, todas las sesiones son planificadas exactamente una vez en algún período k, porque todos los nodos son coloreados. Para cada curso se dicta máximo una sesión de clase por día porque cada curso tiene una

única sesión. La disponibilidad de aulas se respeta dado que existen tantas aulas como sesiones de clase. Finalmente, no se programan simultáneamente dos sesiones de clase conflictivas porque dos nodos adyacentes no son coloreados del mismo color.

De esta forma, se ha demostrado que un problema de k-coloración de grafos puede reducirse al problema de encontrar una solución factible para el modelo de planificación de horarios PLAN-HOR. Por tanto, el problema PLAN-HOR es NP-difícil. □

Capítulo 3

Heurísticas Primales

En este capítulo se describen heurísticas para la construcción de soluciones factibles para el modelo PLAN-HOR formulado en el capítulo anterior.

Al tratarse de un problema NP-difícil, el estudio de heurísticas primales es de gran interés, tanto como algoritmos de solución independientes, como también para producir soluciones iniciales en un esquema de branch-and-bound. Presentamos en las siguientes secciones dos heurísticas: una constructiva y una de mejora.

3.1 Heurística constructiva

Nuestra heurística constructiva genera una solución factible para el modelo *PLAN-HOR* empleando el principio de los métodos secuenciales descritos en el Capítulo 1, combinado con una estrategia de aleatorización. La misma comprende los siguientes pasos principales:

Las sesiones a programar son ordenadas aleatoriamente desde 1 hasta $|S|$. Para $i = 1, ..., |S|$, se programa la sesión i en un período que no cause alguna infactibilidad. Una infactibilidad es una violación a alguna de las restricciones fuertes del modelo *PLAN-HOR* como un cruce entre sesiones conflictivas (ver Capítulo 2), insuficientes aulas disponibles, entre otros. En caso de existir más de un período factible para i, se programa la sesión en el período más favorable, es decir, aquel que minimiza la función objetivo. Si una sesión no puede ser programada, se desprograman una a una las sesiones que fueron programadas antes que ella, hasta poder programarla. Se intenta luego programar las sesiones desprogramadas.

La heurística *solución inicial* descrita en el Algoritmo 1 construye una solución factible para el modelo *PLAN-HOR* empleando este esquema. Para procurar encontrar

una buena solución, se repite la ejecución de la heurística varias veces, cambiando aleatoriamente el orden inicial de las sesiones en cada caso, evaluando la calidad de la solución en la función objetivo del modelo, y eligiendo la mejor solución obtenida hasta alcanzar un criterio de parada. Como criterios de parada se consideraron el número de iteraciones en los que la solución no ha mejorado y un tiempo máximo de ejecución.

Algoritmo 1 Heurística constructiva para una solución inicial

```
lista_proceso ← ∅, lista_sesion ← ∅;
Ordenar aleatoriamente todas las sesiones e insertarlas en lista_sesion;
mientras lista_sesion ≠ ∅ hacer
    si lista_proceso contiene un ciclo entonces
        Reiniciar la ejecución de la heurística;
    fin si
    i ← lista_sesion.pop_front();
    Calcular matriz de disponibilidad de períodos PP_i (ver Algoritmo 2);
    Intentar programar i, determinar variables periodos_factibles y
    sesion_programada (ver Algoritmo 3);
    si sesion_programada == verdadero entonces
        lista_proceso.push_front(i)
    si no
        Depurar lista intentos_programar;
        si periodos_factibles == verdadero entonces                ▷ (Caso 1)
            Desprogramar ℓ ∈ lista_programada que utiliza un aula requerida por i,
            está programada en algún período factible para i
            y no pertenece a intentos_programar_i;
        si no                                                      ▷ (Caso 2)
            Desprogramar ℓ ∈ lista_programada que es conflictiva con i
            y que no pertenece a intentos_programar_i;
        fin si
    fin si
fin mientras
```

Para controlar las iteraciones, nuestro algoritmo hace uso de dos listas:

La *lista_sesion* almacena las sesiones que aún deben ser programadas, mientras que las sesiones ya programadas son almacenadas en la *lista_programada*. Un aspecto importante para garantizar la terminación del algoritmo consiste en evitar la formación de ciclos. Un *ciclo* ocurre si un conjunto de sesiones son programadas y desprogramadas

siguiendo un orden repetitivo. Por ejemplo, supongamos que la sesión 6 fue programada, luego las sesiones 10, 12 y 5; para programar la sesión 5 fue necesario desprogramar las sesiones 6, 10 y 12. Luego se las reprograma, y para programar la sesión 12 es necesario desprogramar la sesión 5. Al intentar programar nuevamente la sesión 5, se deben desprogramar otra vez las sesiones 6, 10 y 12 y el algoritmo entra así en un lazo infinito. Las sesiones fueron programadas en el siguiente orden:

$$6 \quad 10 \quad 12 \quad 5 \quad 6 \quad 10 \quad 12 \quad 5$$

El orden en el que son programadas las sesiones es almacenado en la *lista_proceso* y ésta se utiliza para identificar ciclos. Cuando un ciclo es detectado el algoritmo desprograma todas las sesiones y reinicia la ejecución con un nuevo orden aleatorio de las sesiones. Notar que la existencia de un ciclo está determinada por el orden original de las sesiones. En los experimentos computacionales realizados, los ciclos ocurrieron con muy poca frecuencia, por lo que la estrategia aquí señalada fue suficiente para obtener resultados satisfactorios.

Dada $i \in S$, se define la matriz $PP_i \in \mathbb{Z}^{H \times D}$ cuyos elementos se inicializan mediante $(PP_i)_{jk} := \omega_{ijk}$, cuando el período j del día k está dentro del horario de disponibilidad del profesor a cargo de la sesión i. Caso contrario, $(PP_i)_{jk} = -1$. La matriz se actualiza en cada paso de la heurística para determinar el período factible más favorable en el que una sesión puede ser programada. Esta matriz es fundamental para el cumplimiento de las restricciones (2.2) y (2.3) del modelo *PLAN-HOR* y a su vez, en lo posible, para minimizar la función objetivo.

Para obtener la matriz PP_i se requieren las siguientes funciones: La función *modificar1* modifica la matriz PP_i asignando el valor de -1 en los días y períodos en los que previamente fueron programadas sesiones que tienen conflicto con la sesión i. La función requiere los siguientes parámetros: la sesión i a ser programada, la matriz PP_i, una sesión ℓ previamente programada y conflictiva con i, el día k y período h en el que fue programada la sesión ℓ. La función modifica PP_i empleando el criterio siguiente; si la sesión ℓ fue programada en el período h, para programar una sesión i (conflictiva con ℓ) y no causar un cruce, la sesión i puede ser únicamente programada en los períodos anteriores a $h - \delta_i + 1$ y en los períodos posteriores a $h + \delta_\ell - 1$.

Para obtener una "buena" solución, al programar una sesión, se debe tomar en cuenta, en lo posible el cambio en la función objetivo del modelo *PLAN-HOR*. Es decir, debe ser asignada a un período donde no hayan sido programadas sesiones de cursos aledaños y satisfaga la preferencia de horario del profesor que la dicta. Para ello,

se modifica la matriz de preferencia PP_i en el caso de que una sesión ℓ de un curso aledaño haya sido previamente programada en el período, sumando en las entradas correspondientes el valor de $\gamma(d_{i\ell})$. Esta tarea se desarrolla en la función *modificar2*.

La función requiere los siguientes parámetros: La matriz PP_i, la sesión ℓ programada y no conflictiva con i, el día k y el período h en el que fue programada la sesión ℓ; además del conjunto L_{ik} de períodos en los que i puede iniciar el día k. El resultado es la matriz PP_i modificada con la información que permitirá programar, de ser posible, la sesión i en su período más favorable.

Adicionalmente, recordemos que dos sesiones i, ℓ de un mismo curso no pueden ser programadas el mismo día por la restricción (2.3) del modelo *PLAN-HOR*. Esto se consigue almacenando -1 en las entradas de PP_i correspondientes a todos los períodos del día en le que ℓ fue programada. En el cuadro Algoritmo 2 resumimos el procedimiento para la construcción de la matriz PP_i.

Algoritmo 2 Construcción de PP_i

Entrada: i, PP_i, $lista\_programada$;
Salida: PP_i;
  para todo $\ell \in lista\_programada$ **hacer**
    si $a_{i\ell} = 1$ **entonces**
      $h \leftarrow$ período en el ℓ está programada;
      $k \leftarrow$ día en el ℓ está programada;
      modificar1$(PP_i, \ell, k, h, L_{ik})$;
    si no
      modificar2$(PP_i, \ell, k, h, L_{ik})$;
    fin si
    si i, ℓ pertenecen al mismo curso **entonces**
      $(PP_i)_k \leftarrow -1$;
    fin si
  fin para

Para intentar mejorar la función objetivo del modelo, una sesión i debe ser programada en el período factible más favorable; es decir en un período h del día k tal que $(PP_i)_{hk}$ sea el valor más pequeño distinto de -1 (recordar que el valor de -1 es asignado a los períodos prohibidos). Para ello, se definen dos variables: $periodos\_factibles$ que toma el valor verdadero si y sólo si existen períodos factibles para programar i y la variable *aulas_disponibles* que es verdadera si existe un aula del tipo y capacidad

adecuada para programar la sesión i en su mejor período factible seleccionado. El Algoritmo 3 bosqueja la programación de una sesión i. Las variables auxiliares k_0, h_0 y r_0 almacenan información para determinar la combinación de aula, día y período que sea factible para programar la sesión y minimice la función objetivo. La programación de la sesión i es almacenada en dia_i, $hora_i$ y $aula_i$.

Algoritmo 3 Intentar programar una sesión i en su período factible más favorable

Entrada: Sesión i a programar;
Salida: dia_i, $hora_i$, $aula_i$, $periodos\_factibles$ y $sesion\_programada$;

```
min ← INFINITO;
periodos_factibles ← falso;
aulas_disponibles ← falso;
sesion_programada ← falso;
para todo k ∈ D hacer
    para todo h ∈ L_ik hacer
        si (PP_i)_hk < min y (PP_i)_hk ≠ −1 entonces
            periodos_factibles ← verdadero
            si existe r ∈ R_i disponible el día k período h entonces
                aulas_disponibles ← verdadero
            si no
                aulas_disponibles ← falso
            fin si
        fin si
        si aulas_disponibles = verdadero entonces
            min ← (PP_i)_hk;
            k_0 ← k;
            h_0 ← h;
            r_0 ← r;
        fin si
    fin para
fin para
si min < INFINITO entonces
    sesion_programada ← verdadero;
    dia_i ← k_0;
    hora_i ← h_0;
    aula_i ← r_0;
si no
    sesion_programada ← falso;
fin si
```

Si el lazo principal del Algoritmo 3 termina con el valor $min \neq INFINITO$ significa que la sesión i pudo ser programada, dado que existe una combinación de aula, día y período factibles para programarla. Entonces se procede a fijar $dia_i \leftarrow k_0$, $hora_i \leftarrow h_0$

y $aula_i \leftarrow r_0$ y se resta 1 de la cantidad de aulas tipo r_0 durante los períodos del día k_0 en los que se dicte la sesión i. Existen dos casos por los que una sesión i no puede ser programada:

1. (*Caso 1*) No hay un aula para programar i pese a que existen períodos factibles en los que puede ser programada. En este caso, para programar la sesión i es necesario desprogramar una o varias sesiones que fueron anteriormente programadas en alguna de las aulas requeridas por i durante los períodos factibles en los que podría ser programada. De no ser posible desprogramar una sesión que cumple simultáneamente las condiciones: compartir tipo aula y estar programada en alguno de los períodos factibles de la sesión i, se procede a desprogramar una sesión que cumple al menos la primera condición mencionada, ocupar el mismo tipo de aula, con la idea de que los períodos disponibles para la sesión i se ajusten posteriormente.

2. (*Caso 2*) No hay disponibilidad de horario para programarla; es decir, no existen períodos factibles. Para programar i es necesario desprogramar una o varias sesiones que evitan la programación de la sesión i. Para desprogramar una sesión se considera lo siguiente: En primer lugar, se intenta desprogramar únicamente sesiones en días nuevos. Un día nuevo es aquel en el que no están programadas otras sesiones del mismo curso que la sesión i. Se busca primero desprogramar una sesión conflictiva con i que fue programada en un día nuevo y es dictada por el mismo profesor ya que esto aumenta la posibilidad de programar la sesión i. Si no es posible desprogramar una sesión dictada por el mismo profesor, entonces se intenta desprogramar cualquier sesión conflictiva con i que sea dictada en un día nuevo. Si no es posible desprogramar ninguna sesión conflictiva en un día nuevo, entonces se busca desprogramar cualquier sesión conflictiva con i siempre y cuando no pertenezca al mismo curso que i. Por último si ni siquiera esto es posible, se desprograma cualquier sesión, aunque pertenezca al mismo curso que i ya que puede darse el caso que la duración de dos sesiones de un curso sean distintas y la programación de una de ellas sea únicamente posible en el día en el que fue asignada la otra.

Para programar una sesión, en algunos casos, se requiere desprogramar una o varias sesiones como se indicó anteriormente. Si una sesión es desprogramada se deben: restituir el aula tipo que utiliza y habilitar su disponibilidad de horario. Si una sesión ℓ fue desprogramada para programar una sesión i, ésta es almacenada en la lista

$intentos\_programar_i$ junto con el número de iteración en el que fue desprogramada con la finalidad de evitar desprogramar la misma sesión varias veces e intentar otras posibilidades y con esto evitar la formación de ciclos. Tras un número suficientemente grande de iteraciones, la sesión ℓ es eliminada de dicha lista para aumentar las alternativas de programar la sesión i. A este proceso lo denominamos *depurar* periódicamente la lista $intentos\_programar_i$.

3.2 Heurística de mejora local

La heurística de mejora local tiene como objetivo, como su nombre lo indica, mejorar la calidad de la solución inicial obtenida por la heurística constructiva. La idea fundamental consiste en seleccionar, en cada iteración, dos sesiones que son reprogramadas en un par de períodos nuevos, con la finalidad de mejorar el valor de la función objetivo del modelo *PLAN-HOR*, asegurando siempre que la nueva solución obtenida sea factible. Para ello, diseñamos dos criterios de selección e implementamos aquel para el cual obtuvimos, durante pruebas computacionales preliminares, una mejor relación entre ahorro en la función objetivo y tiempo de ejecución (número de iteraciones).

La *primera estrategia* consiste en calcular el aporte de cada sesión en la función objetivo. El *aporte* en la función objetivo de una sesión i que fue programada el día $k \in K$, en el período $h \in H$ y en el aula tipo $r \in R_i$ está dado por:

$$\omega_{ihk} + \sum_{\ell \in S} \gamma(d_{i\ell}) y_{i\ell}, \tag{3.1}$$

donde, como se indicó en el capítulo anterior,

$$y_{i\ell} = \begin{cases} 1, & \text{si existe cruce de horario entre las sesiones } i \text{ y } \ell, \\ 0, & \text{caso contrario.} \end{cases}$$

La estrategia requiere especificar como parámetro el número Q de pares de sesiones a reprogramar. Las sesiones son ordenadas de forma descendente por su aporte en la función objetivo, las primeras Q sesiones son almacenadas en la *lista_qsesiones*, y se itera sobre esta lista. Para cada sesión $i \in lista\_qsesiones$, se selecciona la otra sesión ℓ a ser reprogramada como aquella cuyo valor de cruce con la sesión sea el mayor. El siguiente cuadro detalla el funcionamiento de la heurística de mejora local con esta estrategia:

Algoritmo 4 Heurística de mejora local con estrategia1

mientras $lista\_qsesiones \neq \emptyset$ **hacer**

Retirar la primera sesión i de $lista\_qsesiones$;

Determinar la sesión ℓ para la cual $\gamma(d_{i\ell})y_{i\ell}$ sea máximo y distinto de cero;

Desprogramar las dos sesiones y determinar las combinaciones factibles de período, día y aula en las podrían ser reprogramadas;

Seleccionar aquella combinación que minimice la función objetivo;

fin mientras

La *segunda estrategia* de selección también depende de un parámetro Q y se basa en calcular para cada par de sesiones su *peso conjunto* en la función objetivo, de forma parecida al cálculo del aporte de una sesión presentado en la ecuación (3.1). El peso conjunto del par de sesiones i e ℓ es:

$$\omega_{ih_1k_1} + \omega_{\ell h_2k_2} + \sum_{\hat{\ell} \in S} \gamma(d_{i\hat{\ell}})y_{i\hat{\ell}} + \sum_{\substack{\hat{\ell} \in S \\ \hat{\ell} \neq i}} \gamma(d_{\ell\hat{\ell}})y_{\ell\hat{\ell}}, \tag{3.2}$$

donde h_1, h_2 son los períodos de los días k_1, k_2 en los que fueron inicialmente programadas las sesiones i e ℓ, respectivamente.

Los pares de sesiones son ordenados de forma descendente por el peso conjunto en la función objetivo y se seleccionan los Q primeros. Se procede entonces como en el caso anterior: cada par de sesiones es desprogramado y reprogramado en el par de períodos donde el aporte a la función objetivo sea mínimo.

Capítulo 4

Implementación Computacional

El modelo fue programado en el lenguaje C++ utilizando la biblioteca para resolver problemas de programación enteros mixtos SCIP. SCIP (Solving Constraint Integer Programs) es un entorno para resolver programas con restricciones y programas enteros mixtos empleando el esquema de branch-and-cut desarrollado desde el año 2001 en el *Konrad-Zuse-Zentrum für Informationstechnik* Berlin (ZIB), en Alemania [44]. SCIP es de acceso libre para propósitos académicos, y está orientado al control total del proceso de solución y acceso detallado de la información del solver [29, 45].

4.1 Modelo de datos

4.1.1 Tablas

Para almacenar los datos requeridos por nuestro modelo, emplearemos las siguientes tablas con sus respectivos campos:

- Tabla `carreras`:

id	nombre
autonumérico	texto

id	nombre
0	Física
1	Ingeniería en Ciencias Económicas y Financieras
2	Ingeniería Matemática
3	Matemática

La tabla `carreras` detalla las carreras a ser consideradas en la planificación. Contiene dos campos: un identificador numérico único que sirve como clave primaria *id*, y un campo de texto de *nombre* de la carrera. Siempre que sea pertinente, los campos *id* y *nombre* se emplearán en las otras tablas del modelo con el mismo significado. La Facultad de Ciencias oferta cuatro carreras: Física, Ingeniería en Ciencias Económicas y Financieras, Ingeniería Matemática y Matemática.

- Tabla `pensum`:

id	nombre	id_carrera
autonumérico	texto	numérico

La tabla `pensum` indica los pénsums disponibles para las distintas carreras dentro de la planificación.

- Tabla `materias`:

id	codigo	nombre	id_pensum	nivel
autonumérico	texto	texto	numérico	numérico

La tabla `materias` describe las diferentes materias a ser planificadas. Cada materia tiene un nombre descriptivo y un código. Además, cada materia está asociada a un pénsum y a un nivel específico.

- Tabla `dias`:
  La tabla `dias` lista los días de la semana a ser considerados dentro de la planificación. Para el caso de la programación académica en la Facultad, estos días son:

id	nombre
autonumérico	texto

id	nombre
0	Lunes
1	Martes
2	Miércoles
3	Jueves
4	Viernes
5	Sábado

- Tabla `horas`:
  La tabla `horas` almacena información acerca de los períodos disponibles para cada día de la planificación. En el caso de la Facultad de Ciencias, estos períodos son:

id	nombre
autonumérico	texto

id	nombre
0	7:00-8:00
1	8:00-9:00
2	9:00-10:00
3	10:00-11:00
4	11:00-12:00
5	12:00-13:00
6	13:00-14:00
7	14:00-15:00
8	15:00-16:00
9	16:00-17:00
10	17:00-18:00
11	18:00-19:00
12	19:00-20:00
13	20:00-21:00

- Tabla `profesores`:

id	nombre
autonumérico	texto

La tabla `profesores` detalla la lista de profesores que dictarán algún curso en la Facultad de Ciencias, durante el período académico en planificación. Para cada profesor se almacenan sus nombres y un identificador numérico único.

- Tabla `valores_de_preferencia`:
  La tabla `valores_de_preferencia` almacena los niveles admisibles de preferencia o disponibilidad de horario. Para cada nivel se especifica un identificador numérico y un nombre.

id	nombre
autonumérico	texto

En las pruebas computacionales realizadas en la Facultad de Ciencias, hemos trabajado siempre con los siguientes valores:

id	nombre
-1	No puede
0	Preferible
1	Disponible

- Tabla `preferencias_de_horarios`:

id	id_profesor	id_dia	id_hora	valor
autonumérico	numérico	numérico	numérico	numérico

La tabla `preferencias_de_horarios` indica el nivel de preferencia de cada profesor para dictar clases en un día y período específicos.

- Tabla `cursos`:

id	id_profesor	num_estudiantes	grupo
autonumérico	numérico	numérico	texto

La tabla `cursos` especifica los cursos que serán dictados durante el semestre a planificar, con su respectivo profesor, el número de estudiantes que lo cursará (también conocido como cupo) y el identificador de grupo. Este último es un código empleado en la Facultad de Ciencias para diferenciar entre los distintos paralelos de una misma materia. (Por ejemplo, Álgebra Lineal GR1 y Álgebra Lineal GR2).

- Tabla `tipos_aulas`:

id	nombre
autonumérico	texto

La tabla `tipos_aulas` detalla los tipos de aula disponibles para programar las actividades de docencia; por ejemplo, aulas de clase normales, laboratorios de computación, aulas magnas, entre otros.

- Tabla `aulas_genericas`:

id	id_tipo	capacidad
autonumérico	numérico	numérico

En el modelo se asignará cada sesión a un aula genérica adecuada. Un aula genérica representa un conjunto de aulas con características similares en cuanto al tipo de aula y la capacidad.

- Tabla `disponibilidad_aulas_genericas`:

id	id_aula_generica	id_dia	id_hora	número
autonumérico	numérico	numérico	numérico	numérico

La tabla `disponibilidad_aulas_genericas` detalla el número de aulas genéricas disponibles un día y período específicos.

- Tabla `sesiones`:

id	id_curso	duracion	tipo_aula	id_dia	id_hora	id_aula_generica
autonumérico	numérico	numérico	numérico	numérico	numérico	numérico

La tabla `sesiones` detalla información acerca de cada una de las sesiones individuales de un curso: el identificador del curso al que pertenece, su duración (número de períodos) y el tipo de aula que requiere. En los campos *id_dia*, *id_hora* e *id_aula_generica* se almacenará posteriormente la solución obtenida por el modelo de planificación de horarios.

- Tabla `relacion_cursos_materias`:

id_curso	**id_materia**
numérico	numérico

La tabla `relacion_cursos_materias` establece las relaciones (de tipo "muchos a muchos") entre los cursos y las materias. Dado que existen diversos pénsums y carreras en la Facultad de Ciencias; diferentes materias pueden ser dictadas en un mismo curso compartido, es decir, un curso puede contener estudiantes de diferentes pénsums y carreras. Por otra parte, una misma materia puede ser dictada en varios cursos, si el número de estudiantes esperado es suficientemente grande.

- Tabla `prerrequisitos`:

id	id_materia1	id_materia2
autonumérico	numérico	numérico

La tabla `prerrequisitos` detalla las relaciones de precedencia entre materias establecidas en un pénsum. Para inscribirse en la *materia2*, un estudiante requiere haber aprobado anteriormente la *materia1*.

4.1.2 Relaciones

Las relaciones entre las tablas descritas en la sección anterior se muestran en la siguiente figura:

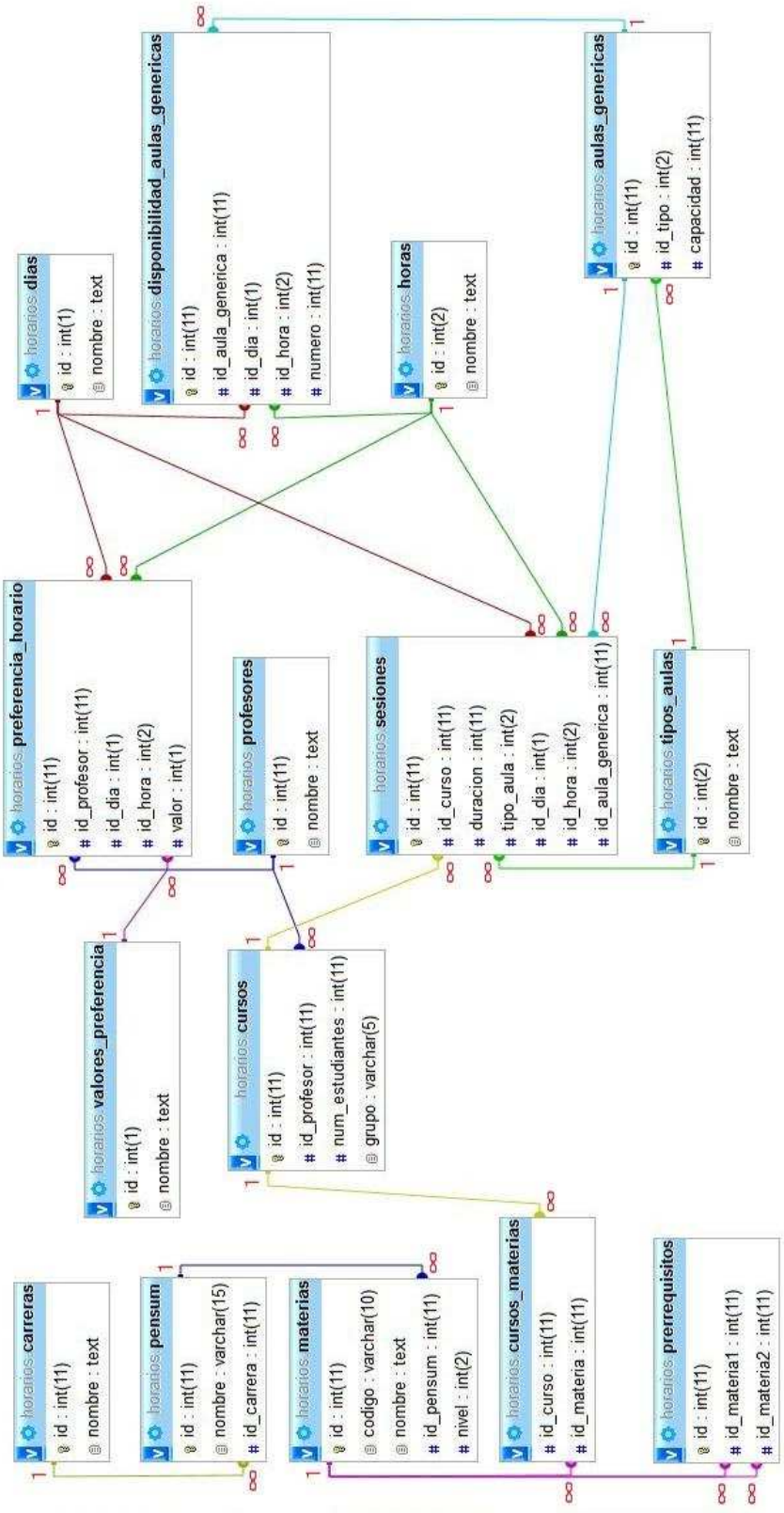

Figura 4.1: Relaciones.

4.1.3 Implementación de las tablas en C++

Las tablas fueron implementada como clases en C++. Por ejemplo, cada objeto de la clase *profesor* representa un registro en la tabla `profesores` y un objeto de la clase *profesores*, un vector de profesores, representa a la tabla en sí.

Examinaremos, a manera de ejemplo, la implementación de algunas de las tablas principales del modelo. El archivo de encabezado `profesores.hpp` contiene la declaración de las clases profesor y profesores.

- `profesores.hpp`

```
#ifndef PROFESORES_HPP_INCLUDED
#define PROFESORES_HPP_INCLUDED

#include<string>
#include<vector>

using namespace std;

class profesor {
  public:
  unsigned id;
  string nombre;
};

class profesores : public vector<profesor> {
public:
  void lectura(string nom_archivo);
  void escritura (ostream& os);
};

#endif // PROFESORES_HPP_INCLUDED
```

Las variables miembro de la clase `profesor` corresponden a campos de la tabla `profesores`. Se han declarado como variables públicas el *id* o código único de tipo entero sin signo, y el *nombre*, de tipo cadena de caracteres. La clase profesores implementa la tabla profesores como un vector de objetos de tipo profesor. En esta clase se han definido funciones para la lectura y escritura de la tabla. La tabla es leída desde un archivo de texto cuya primera línea indica el número total de registros, seguidos de los datos de la tabla, un registro por cada fila, separados por espacios. A continuación se listan las definiciones de las funciones de la clase profesores:

- profesores.cpp

```
#include "profesores.hpp"

#include <fstream>
#include <iostream>
#include <sstream>

using namespace std;

void profesores::lectura(string nom_archivo)
{
   // Lectura de datos desde archivo
   ifstream f(nom_archivo.c_str());
   string linea;
   if (f.is_open()) // este "if" es para controlar posibles ↩
      errores de apertura
   {
    unsigned int l=0; // registra el núm. de línea actual
    while (! f.eof() ) // continuar lectura hasta llegar al ↩
       final del archivo...
    {
      getline (f,linea); // leer una línea
      if(linea[0]=='%') continue; // saltar comentarios
      // Poner la línea en un bufer temporal para procesarla
      stringstream buf;
      buf.str("");
      buf << linea;
      if(l==0) {
          // Leyendo la primera línea...
          unsigned n;
          buf >> n;
          resize(n);
          l++;
      }
      else {
          profesores& esta_tabla= *this;
          buf >> esta_tabla[l-1].id >> esta_tabla[l-1].nombre;
          l++;
      }
    }
    f.close();
   }
```

```
}

void profesores::escritura (ostream& os)
{
    profesores& esta_tabla= *this;
    for(unsigned i=0; i < size(); i++)
       os << esta_tabla[i].id << "   "
          << esta_tabla[i].nombre << endl;
}
```

La función de `lectura` requiere el nombre del archivo que contiene los datos de la tabla profesores, si no existen errores al abrir el archivo se procede a leer cada una de las líneas de la tabla y almacenar la información en la clase profesores. Toda línea cuyo primer caracter sea % es considerada como comentario y la primera línea (sin comentario) contiene el número de registros de la tabla profesores. La función de `escritura` requiere un objeto de clase la `ostream` (es decir, un flujo de salida, como `cout`) y presenta en forma de tabla cada registro de la clase profesores.

Por ejemplo, la tabla profesores de la instancia 1 se almacenaría como el siguiente archivo de texto:

%Tabla de profesores	
%num_registros:	
3	
%id	nombre
0	Profesor1
1	Profesor2
2	Profesor3

El código asociado a la declaración de la tabla cursos se encuentra en el archivo `cursos.hpp`

- cursos.hpp

```
#ifndef CURSOS_HPP_INCLUDED
#define CURSOS_HPP_INCLUDED

#include<string>
#include<vector>

using namespace std;
```

```
class curso {
  public:
  unsigned id;
  unsigned id_profesor;
  unsigned num_estudiantes;
  string grupo;
};

class cursos : public vector<curso> {
public:
  void lectura(string nom_archivo);
  void escritura (ostream& os);
};

#endif // CURSOS_HPP_INCLUDED
```

La clase curso contiene cuatro variables miembros, que representan a los cuatro campos de la tabla cursos: se han declarado como variables públicas sin signo el *id* o código único, el *id_profesor* que debe tener un objeto asociado de la clase profesor, la variable *num_estudiantes* que almacena el cupo del curso y la variable *grupo* que almacena el identificador del curso. Las tres primeras variables son de tipo entero sin signo, mientras que la última es una cadena de caracteres. Las funciones de lectura y escritura son similares a aquellas de la tabla *profesores* descritas anteriormente.

El código asociado a la declaración de la tabla `preferencia_horario` se encuentra en el archivo `preferencia_horario.hpp`. Además de las funciones de lectura y escritura en la clase se define la función *consultar_preferencia*. La función `consultar_preferencia(unsigned id_profesor, unsigned id_dia, unsigned id_hora)` permite consultar la preferencia del profesor cuyo id es `id_profesor`, el día `id_dia` en el período `id_hora`.

- preferencia_horario.hpp

```
#ifndef PREFERENCIAS_HORARIOS_HPP_INCLUDED
#define PREFERENCIAS_HORARIOS_HPP_INCLUDED

#include <string>
#include <vector>
```

```
using namespace std;

class preferencia_horario {
  public:
  unsigned id;
  unsigned id_profesor;
  unsigned id_dia;
  unsigned id_hora;
  int valor;
};

class preferencias_horarios : public vector<preferencia_horario↩
   > {
public:
  void lectura(string nom_archivo);
  void escritura (ostream& os);
  int consultar_preferencia(unsigned id_profesor, unsigned ↩
     id_dia, unsigned id_hora) const;
};
```

Cada profesor indica su preferencia de horario empleando el formulario (implementado en una hoja de cálculo de Microsoft Excel) indicado en la Figura 4.2. Estos datos son copiados como hojas de un libro de Excel. Cada uno de los períodos de la semana es coloreado con verde si el profesor prefiere dictar algún curso en dicho período, con amarillo si dictar clases en ese período no le incomoda, y con rojo si por algún motivo no le es posible dictar un curso en ese período.

Para transformar los colores de las celdas de la hoja de cálculo en los valores requeridos por la tabla `preferencias_horarios` se implementó una macro en Excel. El color verde es codificado como 0, el color amarillo es codificado como 1 y el color rojo es codificado como -1. Esta macro está descrita en el Algoritmo 5. La variable j (columna) representa los días de la semana. La variable i (fila) representa los períodos.

Una vez transformados los colores en valores de preferencia, una segunda macro se encarga de extraer estos valores de todas las hojas del libro y escribir en un archivo de texto la tabla de preferencia_de_horario correspondiente.

La clase `preferencia_horarios` contiene cinco variable miembros: las variables públicas de tipo entero sin signo `id`, `id_profesor` que proviene de la tabla

Profesor:	Profesor 1					
Dedicación:	Tiempo completo					
Horario	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado
07:00 - 08:00						
08:00 - 09:00						
09:00 - 10:00						
10:00 - 11:00						
11:00 - 12:00						
12:00 - 13:00						
13:00 - 14:00						
14:00 - 15:00						
15:00 - 16:00						
16:00 - 17:00						
17:00 - 18:00						
18:00 - 19:00						
19:00 - 20:00						
20:00 - 21:00						

Figura 4.2: Ejemplo de Horario Preferencia

Algoritmo 5 color_preferecia

para $j = columna\_inicial$ hasta $columna\_final$ **hacer**
  para $i = fila\_inicial$ hasta $fila\_final$ **hacer**
    si $color.celda(i, j) = verde$ **entonces**
      $valor.celda(i, j) = 0$;
    si no
      si $color.celda(i, j) = amarillo$ **entonces**
        $valor.celda(i, j) = 1$
      si no
        si $color.celda(i, j) = rojo$ **entonces**
          $valor.celda(i, j) = -1$
        fin si
      fin si
    fin si
  fin para
fin para

`profesores`, `id_dia` que proviene de la tabla `dias` e `id_hora` que proviene de la tabla `horas`, y la variable pública de tipo entero `valor`. A continuación se detalla la función `consultar_preferencia` que se encuentra en `consultar_preferencia.cpp`:

```
int preferencias_horarios::consultar_preferencia(unsigned ↩
    id_profesor, unsigned id_dia, unsigned id_hora) const
{
     const preferencias_horarios& esta_tabla= *this;
     for(unsigned i=0; i<esta_tabla.size(); i++)
     {
         if((esta_tabla[i].id_profesor==id_profesor)
```

```
            && (esta_tabla[i].id_dia==id_dia)
            && (esta_tabla[i].id_hora==id_hora))
            return esta_tabla[i].valor;
    }
    assert(1==0);
    return 0;
}
```

La función recorre secuencialmente los objetos de tipo preferencia_horario almacenadas en el vector preferencias_horarios hasta identificar aquel cuyas variables `id_profesor`, `id_dia`, `id_hora` coincidan con los valores de los parámetros correspondientes y retorna el valor de preferencia asignado.

La estructura de todas las demás tablas, así como sus funciones de lectura y escritura, está implementada de manera similar. Adicionalmente, para la tabla *disponibilidad_aulas_genericas* se implementa la función `unsigned consultar_num_aula(unsigned id_aula, unsigned id_dia, unsigned id_hora)` que permite consultar el número de aulas disponibles de tipo *id_aula*, el día *id_dia* en el período *id_hora*.

4.2 Funciones y rutinas

La Figura 4.3 ilustra el funcionamiento global del programa. Describiremos a continuación con más detalle las funciones y rutinas que componen el código, agrupándolos en cuatro categorías: lectura de datos y preprocesamiento, implementación de las heurísticas primales, implementación del modelo de programación entera en SCIP, y salida de resultados.

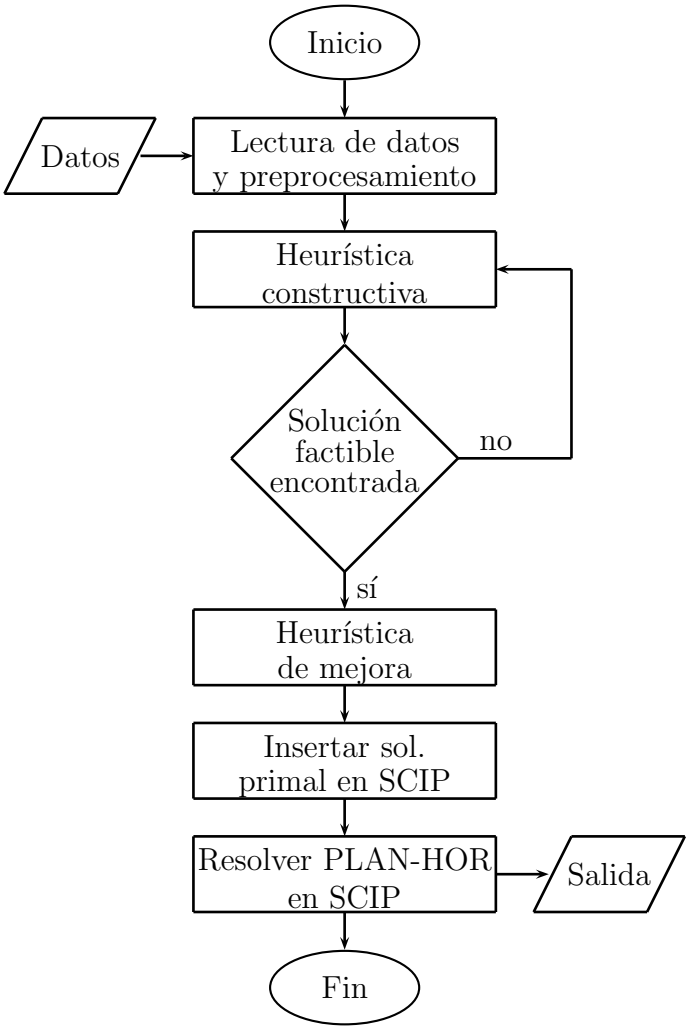

Figura 4.3: Funcionamiento global del programa.

4.2.1 Lectura de datos y preprocesamiento

Para la lectura de las tablas del modelo de datos se emplean las funciones miembro de las clases correspondientes (ver sección 4.3.1). Adicionalmente, las siguientes funciones sirven para preparar los datos requeridos por el modelo *PLAN_HOR* definidos en la Sección 2.2:

- `void precedenciamaterias(unsigned num_materias,`
  `const prerrequisitos & tabla_prerrequisito,`
  `vector<vector<unsigned>> & P)`

  Descripción: La función crea la matriz de precedencia de materias P tal que $p_{m_1 m_2} = 1$ si el pénsum de la materia m_1 es igual al pénsum de m_2 y existe una

cadena de prerrequisitos de m_1 a m_2, y $p_{m_1m_2} = 0$ caso contrario. La matriz P es retornada como parámetro de referencia.

- `void distanciamaterias(const vector<vector<unsigned>> & PR,`
  `const materias & tabla_materia,`
  `vector<vector<unsigned>> & DM)`

  Descripción: La función calcula la matriz de distancia entre materias $DM = (d^M_{m_1m_2})$ tal que $d^M_{m_1m_2}$ es la diferencia en valor absoluto de los niveles a los que pertenecen m_1 y m_2, si las materias comparten el mismo pénsum y no existe una cadena de prerrequisitos de m_1 a m_2. Si cualquiera de estas condiciones no se cumple, $d^M_{m_1m_2} = INFINITO$.

- `void Mcursoi(unsigned i,`
  `const relaciones_cursos_materias & tabla_relacion_curso_materia,`
  `vector<unsigned> & Mi)`

  Descripción: La función almacena las materias asociadas al curso i en el vector M_i.

- `void distanciacursos(const vector<vector<unsigned>> & DM,`
  `const cursos & tabla_curso,`
  `const materias & tabla_materia,`
  `const relaciones_cursos_materias & tabla_relacion_curso_materia,`
  `vector<vector<unsigned>> & DC)`

  Descripción: La función calcula la matriz de distancia entre cursos $DC = (d^C_{c_1c_2})$, donde $d^C_{c_1c_2}$ se define como el mínimo de las distancias entre las materias relacionadas; es decir, $d^C_{c_1c_2} := \text{mín}\{d_{m_1m_2} | m_1 \in M_{c_1} \text{ y } m_2 \in M_{c_2}\}$.

- `void distanciasesiones(const vector<vector<unsigned>> & DC,`
  `const sesiones & tabla_sesion,`
  `vector<vector<unsigned>> & DS,`
  `vector<vector<double>> & DS1)`

  Descripción: La función calcula la matriz de distancia entre sesiones $DS = (d_{i\ell})$ donde $d_{i\ell}$ es la distancia de los cursos a los que pertenecen las sesiones i y ℓ. Esta

función también retorna la matriz a utilizarse para aplicar penalizaciones en la función objetivo $DS1 = (ds_{i\ell})$ tal que $ds_{i\ell} := \gamma(d_{i\ell})$.

- `void conflictossesiones(const vector<vector<unsigned>> & DS,`
  `const sesiones & tabla_sesion,`
  `const cursos & tabla_curso,`
  `vector<vector<unsigned>> & A)`

  Descripción: La función calcula la matriz de conflictos entre sesiones $A = (a_{i\ell})$, donde $a_{i\ell}$ es igual a 1 si $d_{i\ell} = 0$ o i e ℓ son dictadas por el mismo profesor.

- `void inicio_horario(const sesiones & tabla_sesion,`
  `const cursos & tabla_curso,`
  `unsigned num_horas,`
  `unsigned num_dias,`
  `const preferencias_horarios & tabla_preferencia_horario,`
  `vector<vector<vector<unsigned>>> & L,`
  `vector<vector<vector<int>>> & SPN)`

  Descripción: La función calcula las estructuras L y SPN. La estructura L es empleada para señalar los períodos de un día en el que puede ser programado el comienzo de una sesión, y la estructura SPN señala el nivel de preferencia correspondiente. El vector $L[i][k]$ contiene como componentes los períodos en los que puede iniciar la sesión i en el día k, mientras que el valor de cada componente correspondiente del vector $SPN[i][k]$ representa la satisfacción o desagrado del profesor que dicta la sesión i para dictar clases en ese período. Por ejemplo:

  $L[2][3] = (0, 2, 4)$: Indica que sesión 2 puede ser dictada el día 3 comenzando en el período 0, 2 ó 6.
  $SPN[2][3] = (0, 1, 0)$: El valor de preferencia para que la sesión 2 sea dictada el día 3 empezando en el período 0 es 0; si su período de comienzo es 2, el valor es 1; y si su período de inicio es 4, el valor es 0. Esta función es presentada con más detalle en el Algoritmo 6.

- `void conjunto_tipo_aula(const sesiones & tabla_sesion,`
  `const cursos & tabla_curso,`
  `const aulas & tabla_aula,`

```
                    vector<vector<unsigned>> & R)
```

Descripción: La función construye la estructura R, que almacena en $R[i]$ un vector con las aulas genéricas en las que puede ser programada la sesión i.

- `void conjunto_aula_sesiones(unsigned num_aulas,`
  `const vector<vector<unsigned>> & R,`
  `vector<vector<unsigned>> & RS)`

  Descripción: La función almacena en la r-ésima coordenada de la estructura RS las sesiones que pueden ser programadas en el aula genérica r.

- `void curso_sesiones(const cursos & tabla_curso,`
  `const sesiones & tabla_sesion,`
  `vector<vector<unsigned>> & SC)`

  Descripción: La función almacena en la c-ésima coordenada de la estructura SC las sesiones que pertenecen al curso c.

- `void estructura_variable(const vector<vector<unsigned>> & R,`
  `const vector<vector<vector<unsigned>>> & L,`
  `vector<vector<vector<vector<unsigned>>>> & J)`

  Descripción: La función construye la estructura J empleada para determinar qué variables serán implementadas en el modelo de programación entera en *SCIP*. Por motivos de eficiencia computacional, las variables de decisión x_{irkj} son implementadas en *SCIP* únicamente para aquellas combinaciones de sesión, aula genérica, día y período que pueden ser utilizadas realmente en alguna solución factible. Para determinar estas combinaciones (y para poder acceder a las variables correspondientes), se almacena para cada $i \in S$, $k \in K$ y $0 \leq r \leq R[i].size()$ un vector $J[i][r][k]$ que contiene los períodos en los que i puede iniciar en el día k, suponiendo que es programada en el aula genérica $R[i][r]$.

Algoritmo 6 inicio horario

Entrada: tabla_sesion, tabla_curso, tabla_hora, tabla_dia, tabla_preferencia_horario

Salida: L, SPN

```
para todo i ∈ S hacer
    para todo k ∈ K hacer
        para todo h ∈ H hacer
            s ← 0
                ▷ No permitir la programación de la sesión i si el inicio es muy tarde
            si h + tabla_sesion[i].duracion > tabla_hora.size() entonces
                continue;
            fin si
                    ▷ Tomar en cuenta la disponibilidad de horario de los profesores
            puede_dictar ← verdadero
            para l = h hasta h + tabla_sesion[i].duracion − 1 hacer
                p ← consultar_preferencia(tabla_curso[tabla_sesion[i].id_curso].id_profesor,k,l);
                si p = −1 entonces
                    puede_dictar ← falso
                    break;
                si no
                    s ← s + p;
                fin si
                si puede_dictar = verdadero entonces
                    L[i][k].push_back(h);
                    SPN[i][k].push_back(s);
                fin si
            fin para
        fin para
    fin para
fin para
```

4.2.2 Heurísticas Primales

Una vez que han sido calculadas las matrices y estructuras necesarias, se procede a construir una solución inicial para insertarla en el solver SCIP. Aquí se describen algunas de las funciones principales para obtener una solución primal con la *heurística constructiva* y la *heurística de mejora* presentadas en la Capítulo 3:

- `void lista_sesion_orden_aleatorio(const sesiones & tabla_sesion,`
  `list<unsigned> & lista)`

  Descripción: La función retorna las sesiones ordenadas de forma aleatoria en *lista*.

- `bool existe_ciclo(const list <unsigned> & lista)`

  Descripción: La función retorna el valor de *verdadero* si en la lista se detecta un ciclo. Como se definió anteriormente, un ciclo ocurre si un conjunto de sesiones son programadas y desprogramadas siguiendo un mismo orden. Si un ciclo es identificado en la *lista_proceso*, que es la lista que almacena el orden de programación de las sesiones, la ejecución de la heurística se reinicia. La idea principal de la función consiste en tomar los últimos $n \in \{6, ..., lista.size()/2\}$ elementos de la lista y dividirlos en dos partes de igual tamaño y se los almacena dos listas ($lista_1$ y $lista_2$), que se comparan entre sí. Si son iguales, existe un ciclo, caso contrario se procede sucesivamente hasta $n = lista.size()/2$.

- `void modificar_SPN1(unsigned i,`
  `unsigned l,`
  `unsigned k,`
  `unsigned h,`
  `sesiones tabla_sesion,`
  `vector<vector<vector<unsigned>>> L,`
  `vector<vector<vector<int>>> & SPN)`

  Descripción: La función modifica la matriz $SPN[i]$ asignando el valor de -1 en el día y períodos respectivos en los que previamente fueron programada la sesión ℓ que tienen conflicto con la sesión i.

- `void modificar1_SPN1(unsigned i,`
  `unsigned l,`
  `unsigned k,`
  `unsigned j,`
  `const sesiones & tabla_sesion,`
  `const & vector<vector<vector<unsigned >>> L,`
  `const vector<vector<double>> & DS1,`

vector<vector<vector<int>>> & SPN)

Descripción: La función modifica la matriz $SPN[i]$ asignando el valor de $\gamma(d_{i\ell})$ en el día y períodos en los que previamente fueron programada la sesión ℓ que no tiene conflicto con la sesión i.

- void heuristica_sol_inicial(ostream& os,

  const sesiones & tabla_sesion,

  const dias & tabla_dia,

  const horas & tabla_hora,

  const aulas & tabla_aula,

  const disponibilidad_aulas_genericas & tabla_disponibilidad_aula_generica,

  const cursos & tabla_curso,

  unsigned num_profesores,

  const vector<vector<vector<int>>> & SPN,

  const vector<vector<unsigned>> & AS,

  const vector<vector<vector<unsigned>>> & L,

  const vector<vector<unsigned>> & R,

  const vector<vector<double>> & DS1,

  vector<unsigned> & v_dia,

  vector<unsigned> & v_hora,

  vector<unsigned> & v_aula)

  Descripción: La función construye una solución factible para el modelo *PLAN-HOR* y ésta es almacenada en los vectores *v_dia*, *v_hora* y *v_aula*. La i-ésima coordenada de los tres vectores almacena el día, período y aula genérica, respectivamente, en el que es programada la sesión i. El Algoritmo 1 presentado en el Capítulo 3 describe la función con mayor detalle.

- void soluciones(const sesiones & tabla_sesion,

  const dias & tabla_dia,

  const horas & tabla_hora,

  const aulas & tabla_aula,

```
const disponibilidad_aulas_genericas & tabla_disponibilidad_aula_generica,
const cursos & tabla_curso,
const vector<vector<vector<int>>> & SPN,
const vector<vector<unsigned >> & AS,
const vector<vector<vector<unsigned>>> & L,
const vector<vector<unsigned>> & R,
const vector<vector<double>> & DS1,
const preferencias_horarios & tabla_preferencia_horario,
ostream & os,
unsigned num_prof,
vector<unsigned> & v_dia,
vector<unsigned> & v_hora,
vector<unsigned> & v_aula)
```

Descripción: La función calcula varios horarios llamando a `heuristica_sol_inicial`, y cambiando aleatoriamente el orden de las sesiones en cada ejecución. Se devuelve la mejor solución obtenida hasta alcanzar alguno de los siguientes criterios de parada: 500 iteraciones en las que la solución no ha mejorado, o un tiempo máximo de ejecución fijado en una hora.

- ```
  void heuristica_mejora_local(unsigned Q,
  const sesiones & tabla_sesion,
  const cursos & tabla_curso,
  const preferencias_horarios & tabla_preferencia_horario,
  const dias & tabla_dia,
  const horas & tabla_hora,
  const vector<vector<double>> & DS1,
  const aulas & tabla_aula,
  const disponibilidad_aula_genericas & disponibilidad_aula_generica,
  const vector<vector<vector<int>>> & SPN,
  const vector<vector<unsigned >> & AS,
  ```
  ```

`const vector<vector<vector<unsigned>>> & L,`

`const vector<vector<unsigned>> & R,`

`unsigned num_prof,`

`ostream & os,`

`vector<unsigned> & v_dia,`

`vector<unsigned> & v_hora,`

`vector<unsigned> & v_aula)`

Descripción: La función reprograma pares de sesiones con la finalidad de mejorar la solución obtenida por la heurística constructiva. El parámetro Q es el número de pares de sesiones a reprogramar. El Algoritmo 4 presentado en el Capítulo 3 bosqueja la operación de esta función.

- `void matriz_cruce_sesiones(const sesiones & tabla_sesion,`
  `const vector<unsigned> & v_dia,`
  `const vector<unsigned> & v_hora,`
  `vector<vector<unsigned>> & M)`

  Descripción: La función calcula la matriz de cruces entre sesiones M. La matriz $M \in \{0,1\}^{S \times S}$ contiene elementos de la forma:

$$m_{i\ell} = \begin{cases} 1, & \text{si existe un cruce de horario entre las sesiones } i \text{ y } \ell, \\ 0, & \text{caso contrario.} \end{cases}$$

  Se puede observar que representan a las variables $y_{i\ell}$ del modelo *PLAN-HOR*.

4.2.3 Implementación del modelo en el solver SCIP

En esta sección se presentan las funciones relacionadas con la implementación del modelo de programación lineal entera en el solver SCIP.

- `ProblemaMIP()`

  Descripción: Se crea el objeto solver, se cargan los parámetros de configuración y se crea un problema de programación lineal entera vacio.

- `~ProblemaMIP()`

  Descripción: El destructor libera la memoria asignada dinámicamente a las variables y restricciones del problema, y a otras estructuras auxiliares.

- add_vars(const vector<vector<double>> & DS1,

  const vector<vector<vector<int>>> & SPN,

  const vector<vector<vector<vector<unsigned>>>> & J)

  Descripción: Construye las variables del problema. Por ejemplo, el código para crear las variables x_{irkj}, una vez que la estructura para el almacenamiento de las mismas ha sido correctamente dimensionada, es el siguiente:

```
for(unsigned int i=0; i<_x.size(); i++)
 for(unsigned int r1=0; r1<_x[i].size(); r1++)
  for(unsigned int k=0; k<_x[i][r1].size(); k++)
   for(unsigned int j1=0; j1<_x[i][r1][k].size(); j1++)
   {
   //Fijar un nombre para la variable
   SCIP_VAR * var;
   namebuf.str("");
   namebuf << "x$" <<i<<"_"<<R[i][r1]<<"_"<<k<<"_"<<J[i][r1][k↩
      ][j1];
   //Recordar r1 no representa un aula sino un índice válido ↩
      del vector de aulas R[i].
   //Recordar j1 no representa un período sino un índice válido↩
       del vector J[i][r1][k].
   //Crear un objeto SCIP_VAR para la nueva variable binaria.
   SCIP_CALL_EXC(SCIPcreateVar(_scip, & var, namebuf.str().↩
      c_str(), 0.0, 1.0, SPN[i][k][j1], SCIP_VARTYPE_BINARY, ↩
      TRUE, FALSE, NULL, NULL, NULL, NULL, NULL) );
   //Notar que SPN[i][k][j1] es el coeficiente de costo de x[i↩
      ][r1][k][j1]
    //Agregar la variable al problema.
    SCIP_CALL_EXC( SCIPaddVar(_scip, var) );
    // Guardar el puntero al objeto SCIP_VAR, para tener acceso↩
        a la variable posteriormente.
    _x[i][r1][k][j1] = var;
    }
```

- add_cons(unsigned num_dias,

  unsigned num_horas,

  const aulas & tabla_aula,

  const disponibilidad_aulas_genericas & disponibilidad_aulas_genericas,

  const sesiones & tabla_sesion,

```
const vector<vector<vector<unsigned>>> & L,
const vector<vector<vector<vector<unsigned>>>> & J,
const vector<vector<unsigned>> & AS,
const vector<vector<unsigned>> & R,
const vector<vector<unsigned>> & SC,
const vecto<vector<unsigned>> & RS)
```

Descripción: Construye las restricciones del problema. Por ejemplo, el código para crear la restricción (2.2) (ver Capítulo 2), la cual evita cruces de materias conflictivas es el siguiente:

```
ostringstream namebuf;

SCIP_CONS * cons;
namebuf.str("");
//Nombre de la restricción:
namebuf<<"sesiones_conflictivas";
//Crear el objeto tipo SCIP_CONS para almacenar una restricción↩
    del tipo == 0.
SCIP_CALL_EXC(SCIPcreateConsLinear(_scip, & cons, namebuf.str()↩
   .c_str(), 0, NULL, NULL, 0, 0, TRUE, TRUE, TRUE, TRUE, TRUE,↩
    FALSE, FALSE, FALSE, FALSE, FALSE) );

//Fijar los coeficientes.
    for(unsigned i=0; i<_y.size(); i++)
     for(unsigned j=0; j<_y[i].size(); j++)
         {
     if(i==j)
     continue;
     if(AS[i][j]==1)
     SCIP_CALL_EXC( SCIPaddCoefLinear(_scip, cons, _y[i][j], 1)↩
         );
     }
    //Agregar la restricción al problema.
    SCIP_CALL_EXC( SCIPaddCons(_scip, cons) );
   //Guardar el puntero al objeto SCIP_CONS, para tener acceso ↩
      a la restricción posteriormente.
    _asignatura_profesor=cons;
```

- `solucion_inicial_SCIP(const vector<unsigned> & v_dia,`

```
const vector<unsigned> & v_hora,
const vector<unsigned> & v_aula,
const vector<vector<vector<int>>> & SPN,
const vector<vector<unsigned>> & M,
SCIP* scip,
SCIP_SOL* solucion,
vector<vector<vector<vector<SCIP_VAR >>>> & x,
vector<vector<SCIP_VAR *>> & y,
vector<vector<unsigned>> R)
```

Descripción: Es llamada por la función siguiente para exportar la solución obtenida por las heurísticas (almacenada en los vectores v_dia, v_hora y v_aula) a una solución en el formato requerido por *SCIP* (en los vectores x y y).

- `solve(const vector<vector<vector<unsigned>>> & L,`
  `const vector<vector<unsigned>> & R,`
  `const vector<unsigned> & v_dia,`
  `const vector<unsigned> & v_hora,`
  `const vector<unsigned> & v_aula,`
  `const vector<vector<unsigned>> & M;)`

  Descripción: Añade la solución inicial obtenida por las heurísticas y almacenada en los vectores *v_dia*, *v_hora* y *v_aula*. Luego invoca las funciones pertinentes de SCIP y resuelve el problema de programación lineal entera.

  El siguiente es el código para la implementación de la función *solve*:

```
// Crear el objeto para almacenar la solución.
SCIP_SOL *solucion;

SCIP_CALL_EXC( SCIPcreateSol(_scip, &solucion, NULL) );
//Exportar solución de heurística al formato SCIP.
solucion_inicial_SCIP(v_dia, v_hora, v_aula, L, M, _scip, ↩
   solucion, _x,  _y, R);

//Añadir la solución inicial al modelo.
SCIP_CALL_EXC( SCIPaddSolFree(_scip, &solucion, &almacenar))↩
   ;
```

```
        // Invocar a rutinas de SCIP para resolver el modelo (↩
           iniciar branch-and-cut)
   SCIP_CALL_EXC( SCIPsolve(_scip) );
```

4.2.4 Salida de soluciones

- `show_sol(const sesiones & tabla_sesion,`

  `const cursos & tabla_curso,`

  `const dias & tabla_dia,`

  `const materias & tabla_materia,`

  `const horas & tabla_hora,`

  `const relaciones_cursos_materias & tabla_relacion_curso_materia,`

  `const vector<vector<vector<unsigned>>> & L,`

  `ostream & salida1,`

  `ostream & salida2,`

  `ostream & salida3,`

  `ostream & salida4,`

  `const vector<unsigned> & v_aulas,`

  `const vector<unsigned> & v_dias,`

  `const vector<unsigned> & v_horas,`

  `const vector<vector<vector<int>>> & SPN,`

  `const vector<vector<double>> & DS1,`

  `const vector<vector<unsigned>> & R)`

  Descripción: Esta función presenta el resultado obtenido por el solver. El resultado es organizado en cuatro archivos (salida1, salida2, salida3 y salida4). En el archivo `salida1` se presentan el valor de las variables (x_{irkj} y y_{ij}) y de la función objetivo. En el archivo `salida2` se presenta el horario, en el que se señala el profesor, el grupo, el cupo, los días y respectivos períodos en las que será dictado el curso junto con el aula genérica en la que será dictada cada sesión. En el archivo `salida3` se cuentan los valores de penalización de cada sesión programada y en el archivo `salida4` se presentan los estadísticos del proceso de branch-and-bound como: el tiempo, el número total de LPs resueltos, entre otros.

El horario solución presentado en el archivo `salida2`; tiene la forma mostrada en el siguiente ejemplo:

Materia(s):
NOMBRE_1 GR1
NOMBRE_2 GR1
Profesor: PROFESOR1
Numero estimado de estudiantes: 26
Horario:

martes	17 - 19	(aula capacidad 20)
miércoles	9 - 11	(laboratorio capacidad 20)

Capítulo 5

Resultados Computacionales

En este capítulo se reportan los resultados de las pruebas computacionales realizadas sobre el modelo de programación entera y las heurísticas de solución. La implementación fue probada en un computador con procesador dual Xeon E5504 con 13,6GB de RAM utilizando el sistema operativo Linux Centos 6.5. El modelo fue implementado utilizando la interface C++ de SCIP 3.0.1.

5.1 Instancias

Se emplearon 22 instancias para probar el correcto funcionamiento y el desempeño del modelo, tres de ellas reales correspondientes a la planificación académica de los semestres 2013-A, 2013-B y 2014-A de la Facultad de Ciencias de la Escuela Politécnica Nacional. En la Tabla 5.1 se detallan las características de cada instancia.

Las instancias 1 a 9 son instancias ficticias diseñadas principalmente para probar el modelo, su correcta formulación y su desempeño al aumentar paulatinamente el tamaño del problema. La instancia 20 pertenece a la planificación del semestre 2013-B, considerando las 4 carreras (Física, Ingeniería en Ciencias Económicas y Financieras, Matemática e Ing. Matemática) y sus respectivos pénsums (pénsum 2010 y 2012). Las instancias 10 a 19, son restricciones de la instancia real 20. La instancia 10 contiene una sola carrera, Matemática, con el pénsum 2010. La instancia 11 considera la carrera de Ingeniería en Ciencias Económicas y Financieras, con el pénsum 2010. La instancia 12 tiene una carrera, Ingeniería en Ciencias Económicas y Financieras, con los pénsums 2010 y 2012. La instancia 13 corresponde a la carrera de Física con los pénsums 2010 y 2012. La instancia 14 toma en cuenta dos carreras, Matemática e Ingeniería Matemática con los pénsums 2010 y 2012. La instancia 15 tiene 4 carreras, Física, Ingeniería en

Tabla 5.1: Características de las instancias empleadas en las pruebas computacionales. Las instancias en negrilla corresponden a datos reales de la planificación académica en la Facultad de Ciencias.

Instancia	Aulas	Carreras	Cursos	Días	Horas	Materias	Pénsums	Prerrequisitos	Profesores	Sesiones	Variables	Restricciones
1	5	1	6	5	4	8	2	4	3	16	538	3.714
2	4	1	7	6	4	10	2	4	4	18	592	4.148
3	4	1	8	6	4	16	2	7	5	19	605	4.507
4	4	1	10	5	8	14	2	7	6	23	989	10.026
5	5	2	11	5	10	21	3	9	6	26	1.561	18.105
6	4	2	12	6	14	18	3	11	11	33	2.784	68.474
7	4	2	18	6	14	26	3	16	14	45	4.632	125.382
8	5	2	20	6	12	46	3	28	11	50	5.841	127.469
9	5	1	26	6	14	29	3	14	30	66	7.780	182.619
10	14	1	32	6	14	26	1	13	58	87	18.152	225.134
11	8	1	45	6	14	41	1	40	58	108	18.270	233.089
12	8	1	45	6	14	47	2	36	58	108	18.270	233.089
13	9	1	37	6	14	37	2	26	58	98	21.447	334.512
14	8	2	43	6	14	63	4	28	58	113	21.930	371.197
15	14	4	40	6	14	59	4	48	58	104	21.945	300.679
16	14	1	37	6	14	31	1	27	58	98	26.774	334.932
17	8	3	61	6	14	100	6	65	58	156	39.789	747.881
18	14	4	98	6	14	128	4	105	58	241	92.449	1.501.199
19	14	4	98	6	14	127	8	81	58	241	92.449	1.501.199
20	**14**	**4**	**98**	**6**	**14**	**187**	**8**	**153**	**58**	**241**	**92.449**	**1.501.199**
21	**15**	**4**	**107**	**6**	**14**	**157**	**8**	**135**	**64**	**260**	**101.981**	**1.746.631**
22	**15**	**4**	**104**	**6**	**14**	**166**	**11**	**110**	**69**	**253**	**126.921**	**2.325.177**

Ciencias Económicas y Financieras, Matemática e Ing. Matemática, con el pénsum 2012 y los niveles 1 al 3. La instancia 16 tiene una carrera, Física, con el pénsum 2010. La instancia 17 tiene 3 carreras: Física, Matemática e Ingeniería Matemática los pénsums 2010 y 2012. La instancia 18 tiene 4 carreras, Física, Ingeniería en Ciencias Económicas y Financieras, Matemática e Ing. Matemática considerando únicamente el pénsum 2010. La instancia 19 abarca la planificación de 4 carreras, Física, Ingeniería en Ciencias Económicas y Financieras, Matemática e Ing. Matemática, con el pénsum 2010 para los niveles 5 al 8 y el pénsum 2012 para los niveles 1 al 4. La instancia 22 corresponde a la información del semestre 2013-A (4 carreras y 2 pénsums).

Finalmente, la instancia 21 es una instancia real cuyos datos provienen de la planificación del semestre 2014-A y comprende las 4 carreras (Física, Ingeniería en Ciencias Económicas y Financieras, Matemática e Ing. Matemática) con sus respectivos pénsums 2010 y 2012.

5.2 Configuración de parámetros de SCIP

En las pruebas computacionales se empleó el solver de programas enteros SCIP y se registraron para cada instancia los siguientes datos:

- El *valor* de la función objetivo.
- El *tiempo* de ejecución requerido por el algoritmo.
- La *cota dual* z_D que es la mejor cota inferior válida encontrada para el óptimo del problema.
- La *cota primal* z_P que es el valor de la mejor solución entera factible encontrada.
- La *brecha de integralidad* (gap), definida de la siguiente forma:

$$\frac{z_P - z_D}{\text{mín}(|z_P|, |z_D|)}. \tag{5.1}$$

Entre las diversas opciones de configuración de SCIP se encuentran la activación y desactivación de 30 heurísticas primales. Las heurísticas son las siguientes: Active Constraint Diving, Clique Primal, Coefficient Diving, Crossover, DINS, Feasibility Pump, Fix-and-Infer, Fractional Diving, Guided Diving, Integer Diving, Integer Shifting, Line Search Diving, Local Branching, Mutation, Objective Pseudo Cost Diving, Octane, Oneopt, Pseudo Cost Diving, RENS, RINS, Root Sol Diving Rounding, Shift

Tabla 5.2: Selección de heurística primales de SCIP. Número de instancias en las que la activación de una heurística mejora el tiempo requerido para alcanzar la solución óptima, o la brecha de optimalidad luego de una hora de cálculo.

Heurística	**Mejora tiempo**	**Mejora gap**
Coefficient Diving	1	0
Feasibility Pump	1	0
Line Search Diving	0	9
Pseudo Cost Diving	0	1
Rounding	1	0
SCIP todas heurísticas	2	2
Shift and Propagate	0	1

and Propagate, Shifting, Simple Rounding, Trivial, Try Sol, Twoopt, Vbounds, Vector Len Diving y Zi Rounding [14]. No siempre es conveniente la activación simultánea de todas las heurísticas, debido a que el tiempo de cálculo requerido por éstas puede empeorar el desempeño del algoritmo de branch-and-bound. Por ello, se realizaron pruebas preliminares para determinar cómo configurar las heurísticas primales de SCIP para el problema de generación de horarios. Las pruebas fueron realizadas sobre las 22 instancias con un tiempo límite de 3600s (una hora) por instancia, con el objetivo de seleccionar la mejor heurística y utilizar posteriormente únicamente esta heurística en la *configuración* de SCIP para las demás pruebas. En la Tabla 5.2 se indica para algunas heurísticas el número de instancias en las que la activación de la heurística mejora el tiempo de ejecución, en los casos en donde fue posible encontrar la solución óptima; y el número de instancias en las que mejora el gap, cuando no fue posible encontrar la solución óptima en el tiempo límite. Las heurísticas que no se muestran en la tabla son aquellas que no mejoraron el tiempo ni el gap para ninguna instancia.

En base a estos resultados, se escogió activar la heurística *Line Search Diving*, debido a que la misma mejora el gap en la mayor cantidad de instancias entre ellas, todas las instancias de mayor tamaño. La heurística *Line Search Diving* es una heurística primal de buceo.

La idea principal de las *heurísticas de buceo* es acotar o fijar variables de una solución fraccional y resolver la relajación lineal iterativamente hasta encontrar una solución entera factible. Para variables binarias, acotar y fijar variables son procesos equivalentes. El proceso de buceo se detiene si ocurre alguno de los tres siguientes casos: el problema es infactible, el óptimo de la relajación lineal LP es peor que el de alguna solución factible conocida para el programa entero, o se alcanza un criterio de parada. Este último puede incluir un tiempo límite para el funcionamiento de la heurística [14].

Existen distintas estrategias para seleccionar la variable a ser acotada o fijada dentro de una heurística de buceo. Una de ellas es *line search diving*. En el Algoritmo 7 se bosqueja el funcionamiento de esta heurística de buceo.

Algoritmo 7 Line Search Diving.

Entrada: $\bar{x}$ el óptimo de la relajación lineal LP del nodo actual del árbol de branch-and-bound,
$\dot{x}$ el óptimo de la relajación lineal en el nodo raíz,
s una línea que conecta $\dot{x}$ con $\bar{x}$,
$\bar{I} \leftarrow$ el conjunto de índices de todas las variables fraccionales de $\bar{x}$,
$c_{max} \leftarrow \infty$ si no se ha encontrado alguna solución factible, ó $c^T\tilde{x}$ con $\tilde{x}$ una solución factible,
$time \leftarrow 0$;
mientras $\bar{I} \neq \emptyset$ y $c^T\overline{x} < c_{max}$ y $time < time\_limit$ **hacer**
    seleccionar una variable x_j tal que s interseca primero el hiperplano $x_j = k$, con $k \in \{\lfloor \bar{x_j} \rfloor, \lceil \bar{x_j} \rceil\}$
    Acotar x_j con k ($x_j \leq \lfloor \bar{x_j} \rfloor$ ó $x_j \geq \lceil \bar{x_j} \rceil$ según sea el caso);
    Agregar restricción nueva al LP;
    si el LP modificado tiene una solución factible **entonces**
        $\bar{x} \leftarrow$ solución óptima del LP modificado;
    si no
        Parar
    fin si
    $\bar{I} \leftarrow$ conjunto de índices de todas las variables fraccionales de $\bar{x}$
    $time \leftarrow tiempo\_transcurrido$
fin mientras

5.3 Desempeño computacional

5.3.1 Comparación SCIP vs SCIP+heurística

Uno de los objetivos del presente trabajo es estudiar el desempeño práctico de las heurística constructivas y de mejora presentadas en el Capítulo 3, cuando son empleadas para inicializar un esquema de branch-and-bound. Con esta finalidad, se realizaron pruebas sobre las 22 instancias descritas en la sección 5.1. Cada instancia fue resuelta una vez empleado *SCIP* "desde cero" y otra vez, empleando SCIP con información de

una solución inicial obtenida por nuestras heurísticas (en adelante, llamaremos *SCIP + heurísticas* a esta configuración). Para todas las instancias se especificó un tiempo límite de 3600s (una hora) y la configuración mencionada en la sección anterior.

En el Capítulo 3 se describió la heurística constructiva. En cada iteración principal de la misma, las sesiones son ordenadas aleatoriamente y consideradas secuencialmente: cada sesión i es programada en su período factible más favorable. Cuando no es posible programar una sesión se desprograman sucesivamente las sesiones que fueron programadas antes que ella. Los criterios de parada establecidos son un número de 500 iteraciones sin mejora en la solución, o un tiempo límite de una hora.

La mejor solución encontrada por la heurística constructiva es refinada por la heurística de mejora. Se establece el número Q de pares de sesiones a ser desprogramados y reprogramados en una combinación de período y día factibles que minimicen la función objetivo. Para escoger los pares de sesiones se emplearon las dos estrategias de selección descritas en el Capítulo 3 .

En primer lugar, se determinó experimentalmente un valor adecuado para el parámetro Q. Para ello, se obtuvo una solución inicial con la heurística constructiva y se procedió a mejorarla considerando la primera estrategia de la heurística de mejora y valores de $Q \in \{|S|, 2|S|, 3|S|\}$. Los resultados se reportan en la Tabla 5.3. En la tabla se muestran el valor de la solución inicial y el porcentaje de mejora después de $|S|$, $2|S|$ y $3|S|$ iteraciones de la heurística de mejora, respectivamente. En base a estos resultados, se decidió fijar $Q = |S|$, porque se puede observar que la mejora después de $|S|$ iteraciones es marginal: se registra un cambio máximo del 3 % para la instancia 10, mientras que para las restantes la mejora es menor al 2 %. Las Figuras 5.1, 5.2 y 5.3 ilustran la evolución del valor de la solución a lo largo del tiempo para las instancias 20, 21 y 22, respectivamente.

Finalmente, se realizaron pruebas para escoger una de las dos estrategias de selección de sesiones en la heurística de mejora. En estas pruebas se midió el tiempo de ejecución y el valor de la función objetivo, cuando la heurística era aplicada sobre una solución inicial obtenida por la heurística constructiva. Los resultados se indican en la Tabla 5.4. La *estrategia 1* obtiene la mejor solución en 10 instancias, mientras que la otra estrategia produce mejores soluciones en 3 instancias. Adicionalmente, el tiempo requerido para la ejecución de la heurística de mejora local con la estrategia 1 es en promedio menor que el que toma la otra estrategia. Por este motivo, se seleccionó la estrategia 1 para las pruebas principales.

Las pruebas principales consistieron en comparar el desempeño de SCIP y SCIP + heurísticas sobre las 22 instancias descritas en la Sección 5.1. En cada caso, se registró el

Tabla 5.3: Heurística de mejora: Selección del parámetro Q.

Instancia	**Solución inicial**	$Q = \|S\|$		$Q = 2\|S\|$		$Q = 3\|S\|$	
		Solución	%	Solución	%	Solución	%
1	35,00	27,00	23 %	27,00	0 %	27,00	0 %
2	17,00	17,00	0 %	17,00	0 %	17,00	0 %
3	50,00	50,00	0 %	50,00	0 %	50,00	0 %
4	4,00	4,00	0 %	4,00	0 %	4,00	0 %
5	15,00	15,00	0 %	15,00	0 %	15,00	0 %
6	0,00	0,00	0 %	0,00	0 %	0,00	0 %
7	0,00	0,00	0 %	0,00	0 %	0,00	0 %
8	64,67	64,67	0 %	64,67	0 %	64,67	0 %
9	88,33	87,00	2 %	87,00	0 %	87,00	0 %
10	234,64	224,21	4 %	224,21	0 %	224,21	0 %
11	288,02	281,19	2 %	281,19	0 %	281,19	0 %
12	183,00	173,17	5 %	173,17	0 %	173,17	0 %
13	172,33	139,33	19 %	139,33	0 %	139,33	0 %
14	313,17	296,33	5 %	296,33	0 %	296,33	0 %
15	156,00	112,00	28 %	112,00	0 %	112,00	0 %
16	245,36	195,60	20 %	193,60	1 %	193,60	0 %
17	463,67	404,67	13 %	395,67	2 %	395,67	0 %
18	1.040,50	888,88	15 %	881,71	1 %	881,71	0 %
19	488,67	413,83	15 %	412,17	0 %	412,17	0 %
20	1.060,33	943,21	11 %	922,88	2 %	917,88	1 %
21	895,33	706,00	21 %	706,00	0 %	706,00	0 %
22	844,79	678,50	20 %	658,33	3 %	654,76	1 %

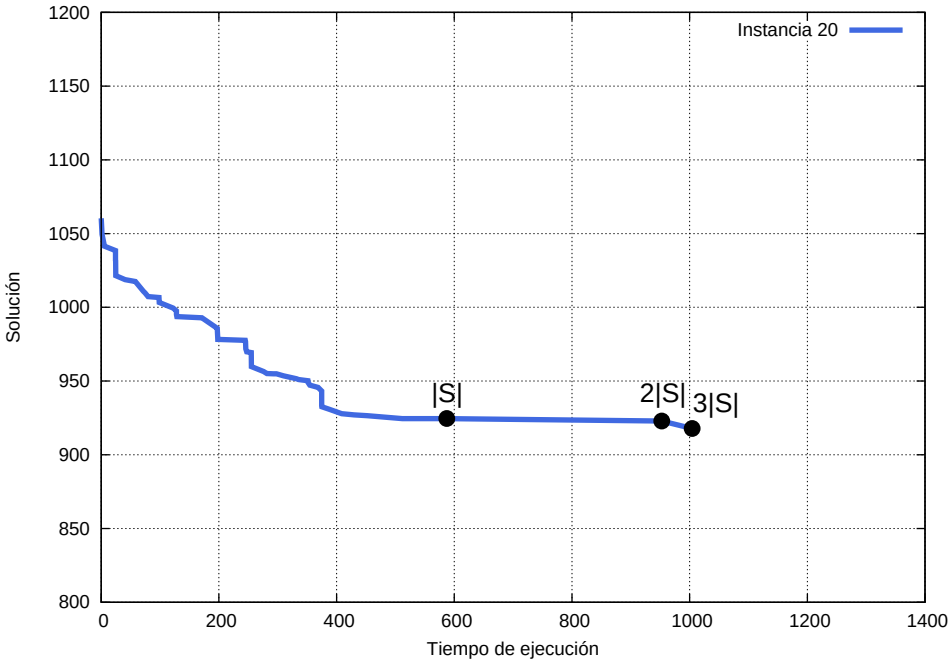

Figura 5.1: Instancia 20: Evolución de la calidad de la solución de la heurística de mejora respecto al tiempo de ejecución.

valor de la mejor cota dual y la mejor cota primal encontradas, además del tiempo requerido por el algoritmo, cuando fue posible encontrar la solución óptima dentro del

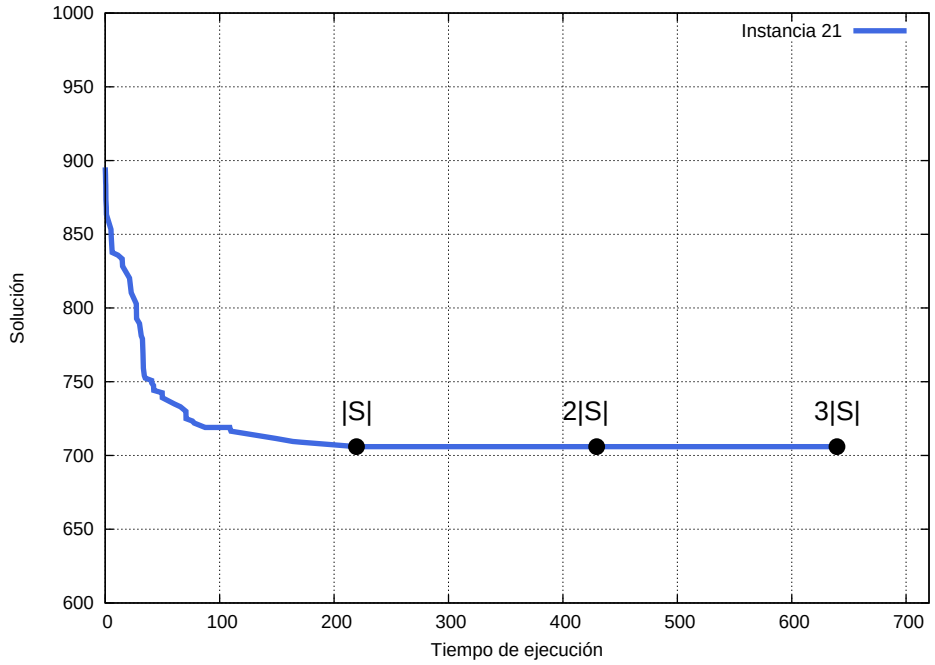

Figura 5.2: Instancia 21: Evolución de la calidad de la solución de la heurística de mejora respecto al tiempo de ejecución.

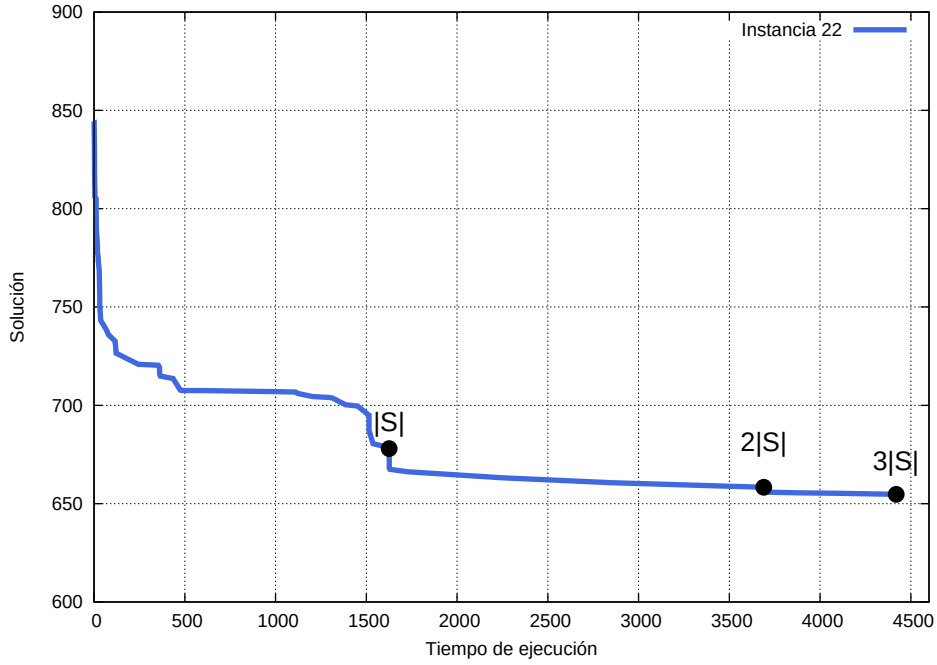

Figura 5.3: Instancia 22: Evolución de la calidad de la solución de la heurística de mejora respecto al tiempo de ejecución.

Tabla 5.4: Comparación entre las estrategias de selección en la heurística de mejora

Instancia	**Solución Inicial**	**Solución Estrategia 1**	**Tiempo (s)**	**Solución Estrategia 2**	**Tiempo (s)**
1	35,00	27,00	0,00	27,00	0,00
2	17,00	-	0,01	-	0,00
3	50,00	-	0,01	-	0,00
4	4,00	-	0,07	-	0,07
5	15,00	-	0,05	-	0,05
6	0,00	-	3,37	-	3,26
7	0,00	-	13,26	-	7,50
8	64,67	-	2,13	-	2,17
9	88,33	87,00	13,86	**86,00**	12,51
10	234,64	**224,21**	182,14	224,81	83,14
11	288,02	281,19	73,79	**280,86**	181,36
12	183,00	173,17	73,91	173,17	160,40
13	172,33	**139,33**	123,15	139,83	73,97
14	313,17	**296,33**	32,64	297,67	10,54
15	156,00	**112,00**	105,24	113,00	88,74
16	245,36	195,60	372,85	**193,93**	409,43
17	463,67	**404,67**	37,83	407,50	19,95
18	1040,50	**888,88**	453,12	918,31	727,43
19	488,67	**413,83**	489,16	424,17	693,82
20	1.060,33	**943,21**	455,85	963,07	887,12
21	895,33	**706,00**	249,91	738,00	456,02
22	844,79	**678,50**	2.902,19	752,21	2.042,01
Tiempo Total			**5.584,58**		5.859,53

límite de tiempo. Cuando esto no fue posible, se registró la brecha de optimalidad (gap) alcanzada al agotar el tiempo límite. Los resultados obtenidos están presentados en la Tabla 5.5. *SCIP+heurísticas* encuentra al menos una solución factible para todas las instancias, mientras que *SCIP* no encuentra ninguna solución para las 5 instancias de mayor tamaño, incluyendo las 3 instancias reales. En las instancias en que ambos procedimientos encontraron alguna solución, en la mayoría de los casos las soluciones primales encontradas por *SCIP+heurísticas* son mejores.

La Tabla 5.6 resume algunos indicadores importantes para la comparación. *SCIP+heurísticas* encuentra al menos una solución factible para todas las instancias mientras que *SCIP* sólo lo consigue para 17 instancias. En 6 instancias, *SCIP+heurísticas* encuentra la solución óptima dentro del límite de tiempo, mientras que *SCIP* lo consigue para 5 instancias. Si consideramos las 16 instancias donde ninguno de los algoritmos encuentra el óptimo, puede verse que la mejor solución primal obtenida por *SCIP+heurísticas* dentro del límite de tiempo supera a la de *SCIP* para las 15 instancias. Para la instancia 5, ambas obtienen soluciones del mismo valor. Finalmente, de las 5 instancias donde ambos algoritmos encuentran la solución óptima, en cuatro instancias el tiempo requerido por *SCIP+heurísticas* es menor. *SCIP* alcanza un mejor tiempo sólo para la instancia 1, la más pequeña de de todas.

Tabla 5.5: Desempeño computacional de SCIP vs SCIP+heurísticas.

	SCIP				**SCIP+heurísticas**					
Instancia	**Cota Primal**	**Cota Dual**	**Gap**	**Tiempo**	**Solución Inicial**		**Cota Primal**	**Cota Dual**	**Gap**	**Tiempo**
					Heurística Constructiva	**Heurística Mejora**				
1	27,00	27,00	0 %	196,99	35,00	27,00	27,00	27,00	0 %	205,69
2	11,00	11,00	0 %	10,80	17,00	17,00	11,00	11,00	0 %	6,91
3	50,00	30,67	63 %	-	50,00	50,00	50,00	32,67	48 %	-
4	4,00	4,00	0 %	38,52	4,00	4,00	4,00	4,00	0 %	0,74
5	12,00	8,50	41 %	-	15,00	15,00	12,00	12,00	0 %	1.378,55
6	0,00	0,00	0 %	28,62	0,00	0,00	0,00	0,00	0 %	0,90
7	0,00	0,00	0 %	105,93	0,00	0,00	0,00	0,00	0 %	5,33
8	69,00	30,26	128 %	-	64,67	64,67	64,67	30,74	110 %	-
9	102,17	1,00	10117 %	-	88,33	87,00	87,00	1,00	8600 %	-
10	238,29	17,11	1293 %	-	234,64	224,21	224,21	17,11	1210 %	-
11	293,26	127,83	129 %	-	288,02	281,19	281,19	122,83	129 %	-
12	197,83	101,17	96 %	-	183,00	173,17	173,17	101,17	71 %	-
13	206,67	11,00	1779 %	-	172,33	139,33	139,33	11,00	1167 %	-
14	306,67	6,00	5011 %	-	313,17	296,33	296,33	6,00	4839 %	-
15	166,00	31,00	435 %	-	156,00	112,00	112,00	31,00	261 %	-
16	366,43	11,00	3231 %	-	245,36	195,60	195,60	11,00	1678 %	-
17	536,83	11,00	4780 %	-	463,67	404,67	404,67	11,00	3579 %	-
18	Sin solución	64,72	-	-	1.040,50	888,88	888,88	63,94	1290 %	-
19	Sin solución	84,33	-	-	488,67	413,83	413,83	91,17	354 %	-
20	Sin solución	60,92	-	-	1.060,33	943,21	943,21	59,83	1476 %	-
21	Sin solución	59,00	-	-	895,33	706,00	706,00	59,00	1097 %	-
22	Sin solución	2,00	-	-	844,79	678,50	678,50	2,00	33825 %	-

Tabla 5.6: Comparación SCIP vs SCIP+heurísticas. Indicadores globales.

Criterio	**SCIP**	**SCIP+heurísticas**
Solución encontrada	17	22
Solución óptima	5	6
Mejor solución primal (de 16)	0	15
Mejor tiempo (de 5)	1	4

5.3.2 Comparación SCIP, Gurobi y Heurísticas

Adicionalmente, realizamos pruebas sobre las 22 instancias anteriormente descritas utilizando el solver Gurobi y SCIP con configuraciones predeterminadas y un límite de tiempo establecido en una hora y las heurísticas (constructiva y de mejora local) como algoritmos de solución independientes. Los resultados obtenidos se presentan en la Tabla 5.7.

Tanto *Gurobi* como las *heurísticas* encontraron una solución factible para todas las instancias, mientras que *SCIP* no encontró una solución para 5 instancias, dos de ellas instancias reales. *Gurobi* encontró la mejor solución en 11 instancias, mientras que las *heurísticas* encontraron una mejor solución en 6 instancias, entre ellas todas las instancias reales. En las 5 instancias restantes ambos procedimientos encontraron la misma solución. En los dos solvers, la brecha de optimalidad aún es muy grande. Esto se debe probablemente a la falta de buenas cotas duales. Si observamos las restricciones (2.5) y (2.6) corresponden a la linearización de una restricción cuadrática. De hecho, existe un cruce entre las sesiones i y ℓ si y solo si:

$$y_{i\ell} = x_{ir_1jk} \cdot x_{\ell r_2\ \hat{j}k},$$

para algún $r_1, r_2 \in R, k \in K, j \in L_{ik}$ y $\hat{j} \in \Gamma_{i\ell jk}$.

En general, se conoce que este tipo de linearización de restricciones cuadráticas produce relajaciones débiles [46]. En trabajo futuro se aborda la tarea de reducir la brecha de optimalidad.

5.3.3 Análisis de instancias individuales

Analizaremos a continuación con mayor detalle el proceso de solución sobre cinco instancias seleccionadas (5, 9, 17, 20 y 21) que tienen características muy distintas entre sí (ver Sección 5.1). En las Tablas 5.8, 5.9, 5.10, 5.11 y 5.12 se reporta el comportamiento de la heurística constructiva. Se indican el tiempo, número de iteración y valor de la solución para cada vez que la mejor solución es actualizada. La solución obtenida al

Tabla 5.7: Comparación Gurobi, SCIP y Heurísticas

	Gurobi				SCIP				Heurísticas	
Instancia	**Cota Primal**	**Cota Dual**	**Gap**	**Tiempo**	**Cota Primal**	**Cota Dual**	**Gap**	**Tiempo**	**Sol.**	**Tiempo**
1	27,00	27,00	0 %	0,00	27,00	27,00	0 %	178,12	27,00	7,61
2	11,00	11,00	0 %	0,00	11,00	11,00	0 %	6,87	17,00	0,46
3	50,00	50,00	0 %	500,00	50,00	28,75	74 %	-	50,00	0,49
4	4,00	4,00	0 %	0,00	4,00	4,00	0 %	26,90	4,00	0,39
5	12,00	11,00	9 %	-	12,00	8,17	47 %	-	15,00	0,91
6	0,00	0,00	0 %	0,00	0,00	0,00	0 %	10,99	0,00	4,2
7	0,00	0,00	0 %	0,00	0,00	0,00	0 %	28,56	0,00	15,75
8	45,00	34,70	30 %	-	76,33	30,58	150 %	-	64,67	8,23
9	81,17	2,20	3589 %	-	119,50	1,00	11850 %	-	87,00	23,43
10	274,45	27,00	916 %	-	263,64	17,11	1441 %	-	224,21	251,96
11	293,17	147,77	98 %	-	292,26	124,50	135 %	-	281,19	367,92
12	143,00	108,16	32 %	-	206,33	103,67	99 %	-	173,17	251,04
13	77,83	14,00	456 %	-	238,00	11,00	2064 %	-	139,33	239,47
14	269,17	22,00	1123 %	-	375,17	6,00	6153 %	-	296,33	178,84
15	48,00	31,00	55 %	-	197,00	31,00	535 %	-	112,00	270,28
16	154,26	15,11	921 %	-	399,45	11,00	3531 %	-	195,60	544,57
17	332,67	21,00	1484 %	-	Sin solución	11,00	-	-	404,67	828,27
18	1.797,83	168,35	968 %	-	Sin solución	53,50	-	-	888,88	4.053,12
19	217,00	124,33	75 %	-	Sin solución	84,33	-	-	413,83	4.089,16
20	1.309,69	167,53	682 %	-	Sin solución	53,83	-	-	943,21	4055,85
21	937,67	72,21	1198 %	-	Sin solución	59,00	-	-	706,00	3.849,91
22	1.935,57	9,67	19923 %	-	2269,85	2,00	113393 %	-	678,50	6502,19

terminar la heurística constructiva tiene un valor de 15 para la instancia 5 reportando una mejora del 64 % respecto a la primera solución encontrada; para la instancia 9 su valor es de 88,33, con 42 % de mejora; para la instancia 17 es 463,67 y 30 % de mejora; para la instancia 20 es 1.060,33 y 14 % de mejora; y para la instancia 21 es 895,33 y 19 % de mejora. En la heurística constructiva se fijaron dos criterios de parada: un tiempo de ejecución máximo de una hora, y un máximo de 500 iteraciones consecutivas sin encontrar una nueva mejor solución. Para las instancias 20 y 21 se alcanzó el tiempo límite de ejecución. En las demás instancias, la heurística terminó por el otro criterio.

Tabla 5.8: Instancia 5: Comportamiento de la heurística constructiva.

Tiempo	**No. iteración**	**Solución**	**% Mejora**
0,00	1	42,00	-
0,01	4	23,00	45 %
0,01	12	20,00	52 %
0,15	182	19,00	55 %
0,45	556	15,00	64 %

Tabla 5.9: Instancia 9: Comportamiento de la heurística constructiva.

Tiempo	**No. iteración**	**Solución**	**% Mejora**
0,00	1	152,83	-
0,02	2	149,00	3 %
0,04	3	146,67	4 %
0,06	4	128,50	16 %
0,23	12	124,17	19 %
0,25	13	117,83	23 %
0,55	28	111,33	27 %
0,64	32	110,50	28 %
1,00	50	108,83	29 %
2,39	117	97,33	36 %
9,57	463	88,33	42 %

En las Tablas 5.13, 5.14, 5.15 y 5.16 se presentan el valor de la solución obtenida, el tiempo, el número de iteración y el porcentaje de mejora para cada ocasión en que la solución es alterada por la heurística de mejora (consideramos a cada reprogramación de un par de sesiones como una iteración). Para la **instancia 5** la heurística no mejoró la solución inicial obtenida. La solución inicial de la **instancia 9** es mejorada de 88,33 a 87 alcanzando una mejora del 2 %; para la **instancia 17** de 463,67 a 404,67 alcanzando una mejora del 13 %; en la **instancia 20** es mejorada de 1060,33 a 943,21 alcanzando

Tabla 5.10: Instancia 17: Comportamiento de la heurística constructiva.

Tiempo	**No. iteración**	**Solución**	**% Mejora**
0,00	1	658,17	-
2,13	2	616,17	6 %
3,23	3	564,33	14 %
20,29	14	557,83	15 %
39,27	25	531,83	19 %
65,96	41	515,33	22 %
83,02	50	501,17	24 %
252,28	148	473,33	28 %
790,44	462	463,67	30 %

Tabla 5.11: Instancia 20: Comportamiento de la heurística constructiva.

Tiempo	**No. iteración**	**Solución**	**% Mejora**
0,00	1	1.229,40	-
48,17	5	1.227,43	0 %
61,22	7	1.223,98	0 %
85,99	10	1.181,88	4 %
402,28	34	1.167,07	5 %
467,83	39	1.134,57	8 %
594,20	50	1.119,57	9 %
1.062,04	83	1.060,33	14 %
3.600,00	83	1.060,33	14 %

Tabla 5.12: Instancia 21: Comportamiento de la heurística constructiva.

Tiempo	**No. iteración**	**Solución**	**% Mejora**
0,00	1	1.107,33	-
480,39	2	990,50	11 %
486,84	3	962,83	13 %
2.491,22	7	895,33	19 %
3.600,00	7	895,33	19 %

una mejora del 11 %; y en la **instancia 21** de 895,33 a 706, alcanzando una mejora del 21 %.

Tabla 5.13: Instancia 9: Comportamiento de la heurística mejora.

Tiempo	**No. iteración**	**Solución**	**% mejora**
0,00	0	88,33	-
0,74	3	87,00	2 %

Tabla 5.14: Instancia 17: Comportamiento de la heurística mejora.

Tiempo	**No. iteración**	**Solución**	**% mejora**
0.00	0	463.67	-
0.67	9	462.00	0 %
1.14	16	454.00	2 %
1.29	17	448.50	3 %
1.82	22	441.50	5 %
3.45	28	436.33	6 %
5.32	30	431.33	7 %
6.86	37	429.67	7 %
7.68	40	422.67	9 %
10.67	54	421.83	9 %
13.40	62	418.83	10 %
16.43	70	416.33	10 %
17.19	72	415.50	10 %
17.56	74	410.00	12 %
25.23	93	408.00	12 %
28.03	98	406.67	12 %
29.71	101	404.67	13 %

Tabla 5.15: Instancia 20: Comportamiento de la heurística mejora.

Tiempo	Solución	Iteración	% Mejora	Tiempo	Solución	Iteración	% Mejora
0,00	0	1.060,33	-	197,28	126	985,71	7 %
1,35	7	1.049,50	1 %	198,07	127	978,21	8 %
5,46	15	1.041,50	2 %	244,68	144	977,64	8 %
24,22	16	1.038,50	2 %	245,59	145	971,81	8 %
24,42	17	1.031,50	3 %	247,35	147	969,81	9 %
24,73	19	1.021,50	4 %	255,09	151	969,31	9 %
40,62	33	1.018,81	4 %	276,05	172	956,48	10 %
57,98	39	1.017,48	4 %	281,45	174	955,14	10 %
71,52	51	1.010,90	5 %	292,70	178	954,88	10 %
74,63	56	1.009,74	5 %	297,72	179	954,88	10 %
79,60	69	1.007,24	5 %	309,64	181	953,55	10 %
98,04	84	1.006,64	5 %	331,86	196	951,55	10 %
122,19	92	999,48	6 %	335,50	197	951,05	10 %
127,74	100	997,64	6 %	351,27	208	950,21	10 %
127,88	102	995,05	6 %	354,50	209	947,21	11 %
171,21	114	992,88	6 %	368,60	219	945,71	11 %
197,19	125	985,71	7 %	374,54	227	943,21	11 %

Tabla 5.16: Instancia 21: Comportamiento de la heurística mejora.

Tiempo	Solución	Iteración	% Mejora	Tiempo	Solución	Iteración	% Mejora
0,00	0	895,33	-	32,51	46	779,17	13 %
0,24	2	890,33	1 %	33,04	48	769,17	14 %
0,37	3	885,33	1 %	33,33	49	759,17	15 %
0,45	4	883,33	1 %	34,15	50	754,17	16 %
0,62	6	873,33	2 %	34,97	52	752,50	16 %
0,88	7	868,33	3 %	40,59	63	750,83	16 %
4,99	15	853,33	5 %	42,22	64	747,50	17 %
5,27	17	848,33	5 %	49,82	88	742,50	17 %
6,10	19	837,67	6 %	60,31	89	735,00	18 %
14,79	25	833,33	7 %	65,71	95	733,00	18 %
15,17	26	828,33	7 %	76,23	98	723,33	19 %
21,25	30	820,33	8 %	77,75	102	722,00	19 %
22,78	32	810,33	9 %	87,34	125	719,00	20 %
27,16	38	802,83	10 %	108,92	150	719,00	20 %
27,37	40	792,83	11 %	149,39	162	711,50	21 %
29,93	44	789,50	12 %	163,83	179	709,50	21 %
31,48	45	781,17	13 %	219,55	232	706,00	21 %

En la Tabla 5.17 se presentan el número de variables y restricciones del modelo antes y después del presolving con *SCIP+heurísticas*. El presolving es una forma de

transformar la instancia de un problema en una instancia equivalente que se espera sea más fácil de resolver. El *presolving* tiene como objetivo reducir el tamaño del modelo removiendo información como restricciones redundantes o fijando variables. Además, la relajación lineal es ajustada aprovechando la integralidad; es decir, ajustando las cotas de las variables y obteniendo información como implicaciones lógicas para utilizarla posteriormente en la generación de planos cortantes [44, 29]. Para la instancia 5 en *SCIP+heurísticas* el número de variables y restricciones es reducido a 976 variables y 1.005 restricciones, mientras que en *SCIP* es reducido a 1.105 variables y 7.034 restricciones de 1.561 variables y 18.105 restricciones. Para las otras instancias analizadas en esta sección no se presentan cambios en la reducción del número de variables y restricciones entre los dos procedimientos.

Antes del *presolving* la **instancia 5** contiene 1.561 variables y 18.105 restricciones, es decir, aproximadamente el 40 % de variables y 90 % de restricciones es reducido. La **instancia 9** contiene 7.780 variables y 182.619 restricciones antes del *presolving*, aproximadamente el 25 % de variables y 45 % de restricciones es reducido. En la **instancia 17**, el número de variables varía de 39.789 a 17.771 y el número de restricciones cambia de 747.881 a 162.960 restricciones, es decir, aproximadamente el 60 % de variables y el 80 % de restricciones es reducido. En la **instancia 20**, el número de variables es reducido de 92.449 a 41.847 y el número de restricciones baja de 1.501.199 a 335.638, aproximadamente el 60 % de variables y 80 % de restricciones es reducido. Finalmente, en la **instancia 21**, el número de variables cambia de 101.981 a 35.769 y el de restricciones de 1.746.631 a 235.394, aproximadamente el 65 % de variables y 85 % de restricciones es reducido. Esto indica que el impacto del presolving es particularmente alto en las instancias más grandes, sobre todo en las instancias reales.

Tabla 5.17: SCIP: Número de variables y restricciones después del presolving. Número y tipo de planos cortantes encontrados en nodo raíz.

Instancia	No. vars original	No. restr original	No. vars	No. restr	Tipo knapsack	Tipo setppc	Tipo logicor
2	1.561	18.105	976	1.005	3.738	2.773	0
9	7.780	182.619	5.859	100.031	68.067	24.473	7.491
17	39.789	747.881	17.771	162.960	103.423	55.621	3.916
20	92.449	1.501.199	41.847	335.638	251.907	80.338	3.393
21	101.981	1.746.631	35.769	235.394	140.126	95.628	0

Por último, en las Figuras 5.4, 5.5 y 5.6 se presentan la evolución de la cota dual y la cota primal a lo largo del tiempo para los dos algoritmos *SCIP* y *SCIP+heurística*, en las instancias 5, 9 y 20 respectivamente. La línea roja indica el desarrollo de la cota dual y la línea verde el de la cota primal. Para la **instancia 5** *SCIP+heurística* encuentra la solución óptima en 1.378,55s y *SCIP* no termina de resolver la instancia en

el tiempo límite. *SCIP+heurística* encuentra una mejor solución que *SCIP* en el tiempo límite establecido para la instancia 9, sin que ninguno de los dos métodos alcance una cota dual razonable, *SCIP* no encuentra ninguna solución factible para la instancia real 20 y ninguno de los algoritmos encuentra una cota dual aceptable. Para las otras dos instancias, el comportamiento es similar al de la instancia 20.

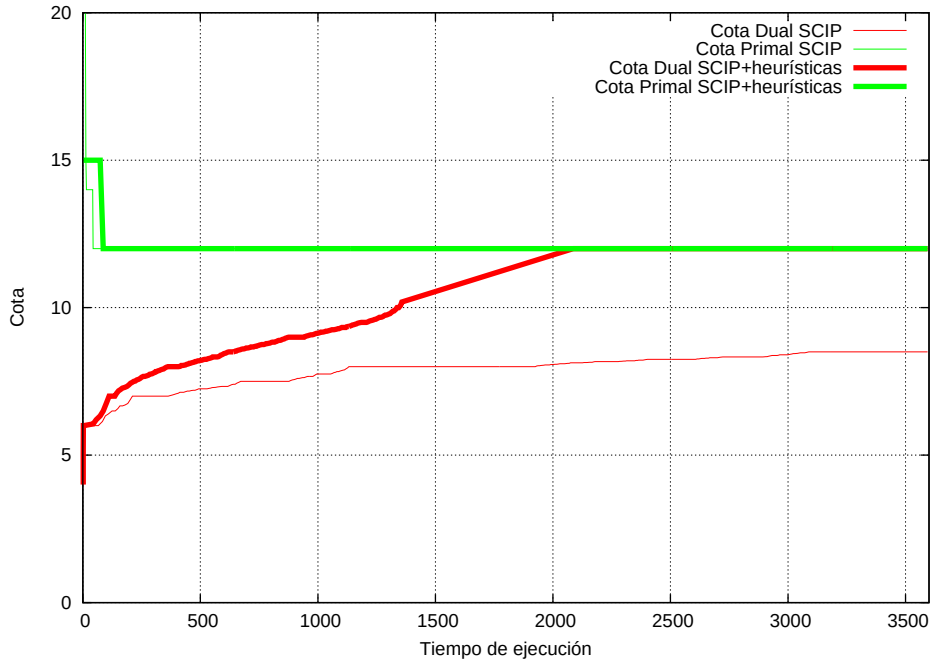

Figura 5.4: Instancia 5: Comportamiento de las cotas dual y primal de SCIP vs SCIP+heurísticas.

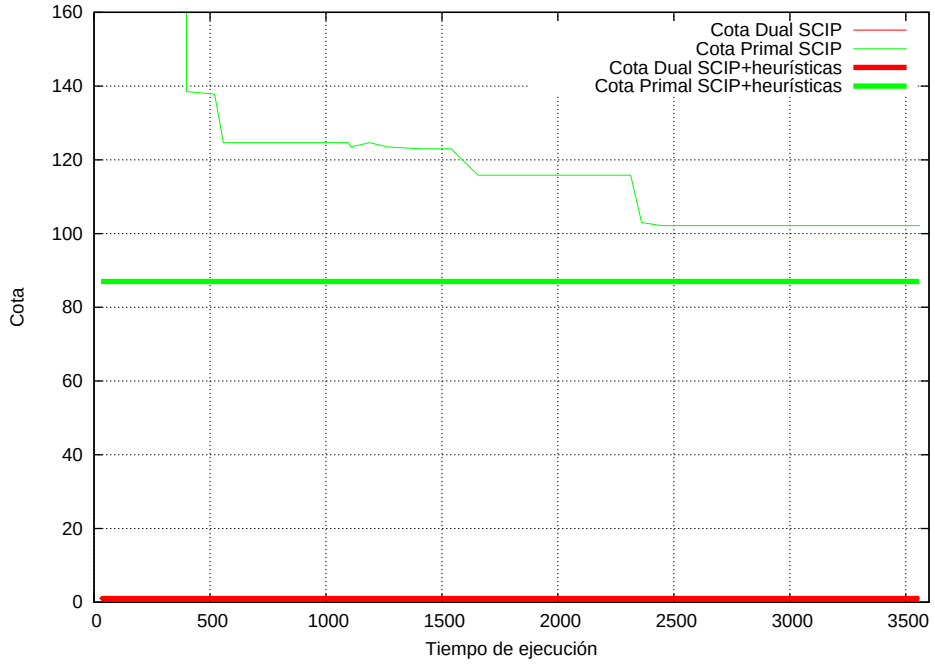

Figura 5.5: Instancia 9: Comportamiento de las cotas dual y primal de SCIP vs SCIP+heurísticas.

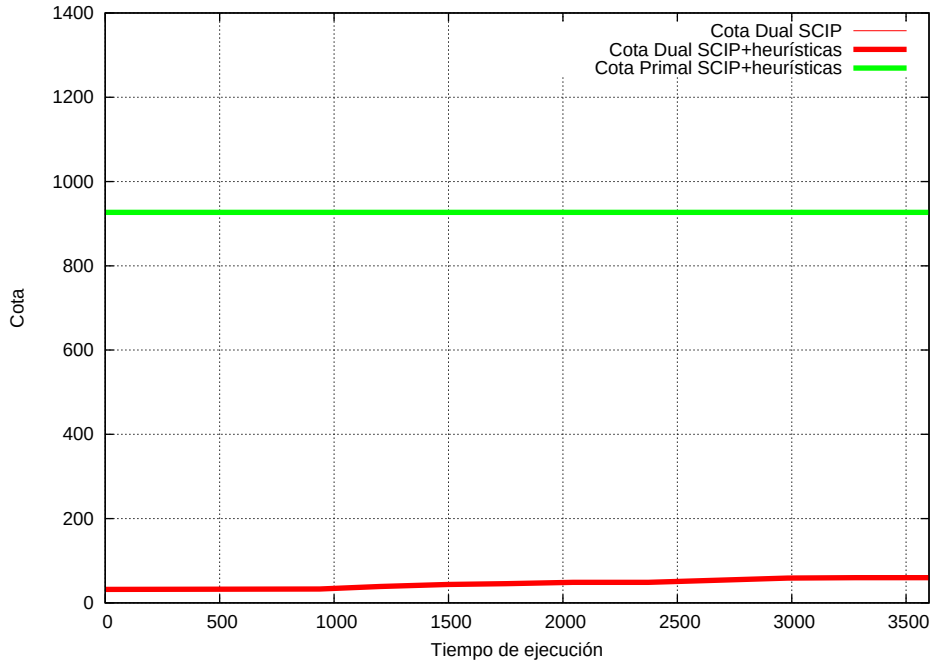

Figura 5.6: Instancia 20: Comportamiento de las cotas dual y primal de SCIP vs SCIP+heurísticas.

Capítulo 6

Conclusiones

La planificación de horarios es una tarea compleja que requiere de mucho tiempo y esfuerzo cuando es realizada manualmente. En caso de la Facultad de Ciencias se requería aproximadamente del trabajo de una persona dedicada una semana para la elaboración de un horario, y éste posiblemente poseía errores (difíciles de detectar) debido a la complejidad del problema. La implementación de un modelo de optimización permite la construcción de varios horarios en cuestión de horas, con la seguridad de que los horarios obtenidos satisfacen todas las restricciones fuertes del problema. Además, se permite incorporar factores que anteriormente no eran tomados en cuenta como la preferencia de horarios de los profesores y evitar cruces entre cursos pertenecientes a niveles aledaños. Adicionalmente, es posible integrar nuevas restricciones como por ejemplo; distribución semanal de sesiones de curso (por ejemplo; para evitar que sesiones de clase de un mismo curso sean dictadas en días consecutivos con la finalidad de mejorar el aprendizaje); establecer un límite de horas de clase de profesores y alumnos; y obtener un horario compacto tanto para alumnos como para profesores. La compacidad consiste en reducir el número de horas "'huecas" en el horario de un profesor o alumno.

Debido a la alta complejidad computacional del problema, los solvers actuales no pueden encontrar soluciones óptimas para instancias reales (en un tiempo de cálculo razonable). Por ello, se implementaron heurísticas primales que permitieron encontrar soluciones factibles para todas las instancias. Los horarios obtenidos fueron satisfactorios en la práctica y no pudieron ser mejorados por el solver SCIP dentro del tiempo límite establecido. Las brechas de optimalidad son aún grandes (en promedio del 3921 % para las instancias de mayor tamaño, es decir, desde la 10 hasta la 22). Posiblemente, esto se debe a que las cotas duales encontradas son todavía muy débiles.

Entre las diversas alternativas para reducir la brecha de optimalidad que podrían ser abordados en trabajos futuros, se encuentran la investigación de diversas estrategias para mejorar la cota dual, como por ejemplo: planos cortantes más específicos para el problema, o relajación lagrangiana y técnicas de descomposición. Adicionalmente, sería conveniente estudiar la posibilidad de paralelizar el algoritmo de solución para aumentar la efectividad en la exploración del espacio de búsqueda. Finalmente, se pueden incorporar en el modelo nuevas restricciones como las mencionadas arriba.

Referencias

[1] E. Burke, J. Marecek, A. Parkers, and H. Rudová, "A branch-and-cut procedure for the Udine Course Timetabling problem," *Annals of Operations Research*, vol. 194, no. 1, pp. 71–87, 2012.

[2] K. Schimmelpfeng and S. Helber, "Application of a real-world university-course timetabling model solved by integer programming," *OR Spectrum*, vol. 29, pp. 783–803, 2006.

[3] E. Burke, P. D. Causmaecker, G. V. Berghe, and H. V. Landeghem, "The State of the Art of Nurse Rostering," *Journal of Scheduling*, vol. 7, no. 6, pp. 441–499, 2004.

[4] S. Petrovic and G. V. Berghe, "A comparison of two approaches to nurse rostering problems," *Annals of Operations Research*, vol. 194, no. 1, pp. 365–384, 2012.

[5] D. Recalde, R. Torres, and P. Vaca, "Scheduling the professional Ecuadorian football league by integer programming," *Computers and Operations Research*, vol. 40, no. 10, pp. 2478–2484, 2013.

[6] C. Ribeiro, "Sports scheduling: Problems and applications," *International Transactions in Operational Research*, vol. 19, pp. 201–226, 2012.

[7] E. Burke and S. Petrovic, "Recent Search Directions in Automated Timetabling," *European Journal of Operational Research*, vol. 140, no. 2, pp. 266–280, 2002.

[8] A. Shaerf, "A Survey of Automated Timetabling," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 87–127, 1999.

[9] E. Burke, K. Jackson, J. Kingston, and R. Weare, "Automated University Timetabling: The State of the Art," *The Computer Journal*, vol. 40, no. 9, pp. 565–571, 1997.

[10] R. Qu, E. Burke, B. McCollum, L. Merlot, and S. Lee, "A Survey of Search Methodologies and Automated System Development for Examination Timetabling," *Journal of Scheduling*, vol. 12, no. 1, pp. 55–89, 2009.

[11] D. Werra, "An Introduction to Timetabling," *European Journal of Operational Research*, vol. 19, no. 2, pp. 151–162, 1985.

[12] R. Lewis, "A survey of metaheuristic-based techniques for University Timetabling problems," *OR Spectrum*, vol. 30, no. 1, pp. 167–190, 2008.

[13] B. McCollum, "A Perspective on Bridging the Gap Between Theory and Practice in University Timetabling," *Practice and Theory of Automated Timetabling VI*, vol. 3867, pp. 3–23, 2007.

[14] T. Berthold, *Primal Heuristics for Mixed Integer Programs*. Berlin: Technische Universität Berlin, 2007.

[15] W. Junginger, "Timetabling in Germany - A Survey," *Interfaces*, vol. 16, pp. 66–74, 1986.

[16] E. Burke, J. P. Newall, and R. F. Weare, "A Simple Heuristically Guided Search for the Timetable Problem," *Proceedings of the International ICSC Symposium on Engineering of Intelligent Systems, Univ. of La Laguna*, pp. 574–579, 1998.

[17] S. Jat and S. Yang, "A hybrid genetic algorithm and tabu search approach for post enrolment course timetabling," *Journal of Scheduling*, vol. 14, no. 6, pp. 617–637, 2011.

[18] T. Yigit, "Constraint-Based School Timetabling Using Genetic Algorithms," *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, vol. 4733, pp. 848–855, 2007.

[19] S. Yang and S. Jat, "Genetic algorithms with guided and local search strategies for university course timetabling," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, pp. 93–106, 2011.

[20] S. Abdullah, S. Ahmadi, E. Burke *et al.*, "A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem," *Journal of the Operational Research Society*, vol. 58, no. 11, pp. 1494–1502, 2006.

[21] G. Kendall and N. Hussin, "A Tabu Search Hyper-heuristic Approach to the Examination Timetabling Problem at the MARA University of Technology," *Practice and Theory of Automated Timetabling V*, pp. 270–293, 2005.

[22] T. Pais and P. Amaral, "Managing the tabu list length using a fuzzy inference system: an application to examination timetabling," *Annals of Operations Research*, vol. 194, pp. 341–363, 2012.

[23] G. Dantzing and M. Thapa, *Linear Programming.* Estados Unidos: Springer Series in Operations Research, 1997.

[24] L. Kantorovich, "A new method of solving some classes of extremal problems," *Doklady Akad Sci USSR*, vol. 28, pp. 211–214, 1940.

[25] R. Bixby, "A Brief History of Linear and Mixed-Integer Programming Computation," *Documenta Mathematica*, pp. 107–121, 2012.

[26] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver, *Combinatorial Optimization.* New York: Wiley Series in Discrete Mathematics and Optimization, 1998.

[27] G. Monge, "Mémoire sur la théorie des déblais et des remblais," *Histoire de l'Académie Royale des Sciences de Paris, avec les Mémoires de Mathématique et de Physique pour la même année*, pp. 666–704, 1784.

[28] A. Schrijver, *Theory of Linear Integer Programming.* Wiley Series in Discrete Mathematics and Optimization, 1986.

[29] T. Achterberg, *Constraint Integer Programming.* Berlin: Technischen Universität Berlin, 2007. [Online]. Available: \url{http://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/1112}

[30] W. Cook, "Markowitz and Manne + Eastman + Land and Doig = Branch and Bound," *Documenta Mathematica*, pp. 227–238, 2012.

[31] D. Bertsimas and N. Tsitsiklis, *Introduction to Linear Optimization.* Athena Scientific Series in Optimization and Neural Computation, 1997, no. 6.

[32] H. Markowitz and A. Mannespero, "On the solution of discrete programming problems," *Econometrica*, vol. 25, pp. 84–110, 1956.

[33] W. Eastman, "Linear programming with pattern constraints," Ph.D. dissertation, Department of Economics, Harvard University, Massachusetts, USA, 1958.

[34] A. Land and A. Doing, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, pp. 497–520, 1960.

[35] J. Little, K. Murty, D. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem," *Operations Research*, vol. 11, pp. 972–989, 1963.

[36] R. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, vol. 64, pp. 275–278, 1958.

[37] E. Burke, J. Marecek, A. Parkes, and P. Rudová, "Decomposition, reformulation, and diving in timetabling," *Computers and Operations Research*, vol. 37, no. 1, pp. 582–597, 2010.

[38] G. Lach and M. Lübbecke, "Curriculum based course timetabling: New solutions to Udine benchmark instances," *Annals of Operations Research*, vol. 194, no. 1, pp. 255–272, 2012.

[39] T. Müller, "ITC-2007 solver description: a hybrid approach," *Annals of Operation Research*, vol. 127, no. 1, pp. 429–446, 2009.

[40] COIN-CBC, "Coin-or branch and cut," https://projects.coin-or.org/Cbc.

[41] H. Cambazard, E. Hebrard, B. O'Sullivan, and A. Papadopoulos, "Local search and constraint programming for the post enrolment-based course timetabling problem," *Annals of Operations Research*, vol. 194, no. 1, pp. 111–135, 2012.

[42] A. Saldana, C. Olivia, and L. Predenas, "Modelos de Programación Entera para un Problema de Programación de Horarios para Universidad," *Revista Chilena de Ingeniería*, vol. 15, no. 3, pp. 245–259, 2007.

[43] R. M. Karp, "Reductibility among combinatorial problems," *Complexity of Computer Computations, Plenum Press*, pp. 85–103, 1972.

[44] T. Achterberg, *SCIP a Framework to Integrate Constraint and Mixed Integer Programming*, 2004.

[45] T. Achterberg, T. Berthold, T. Koch, and K. Wolter, "Constraint Integer Programming: A New Approach to Integrate CP and MIP," *Integration of AI and*

OR Techniques in Constraint Programming for Combinatorial Optimization Problems, vol. 5015, pp. 6–20, 2008.

[46] F. Glover, "Improved linear integer programming formulations of nonlinear integer problems," *Management Science*, vol. 22, pp. 455–460, 1975.