UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Modelo de tres fases para el problema de asignación escolar en la Facultad de Ciencias, UNAM

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

MATEMÁTICO

P R E S E N T A :

Jorge Alejandro Amador Herrera

DIRECTORA DE TESIS:
Dra. Claudia Orquídea López Soto
Ciudad Universitaria, CD. MX., 2019

1. Datos del alumno
Amador
Herrera
Jorge Alejandro
56672964
Universidad Nacional Autónoma de México
Facultad de Ciencias
Matemáticas
310036069

2. Datos de la tutora
Dra.
Claudia Orquídea
López
Soto

3. Datos del sinodal 1
Dra.
Zaida Estefanía
Alarcón
Bernal

4. Datos del sinodal 2
Act.
Mauricio
Aguilar
González

5. Datos del sinodal 3
M. en I.
Adrián
Girard
Islas

6. Datos del sinodal 4
M. en I.O.
María del Carmen
Hernández
Ayuso

7. Datos del trabajo escrito.
Modelo de tres fases para el problema de asignación en la Facultad de Ciencias, UNAM
130 pp
2019

A mis padres

Agradecimientos

A mi familia, amigos y toda la gente que me ha apoyado en este viaje que es la vida.
A la Dra. Claudia Orquídea López Soto, por guiarme e impulsare a mejorar mi trabajo día con día. Agradezco también a la Dra. Zaida Estefanía Alarcón Bernal, al Act. Mauricio Aguilar González, al M. en I. Adrián Girard Islas y a la M. en I.O. María del Carmen Hernández Ayuso por sus valiosos comentarios que enriquecieron el contenido de esta tesis.
Finalmente, agradezco la Universidad Nacional Autónoma de México, que me dió todo a cambio de nada.

Índice general

Introducción

En cada inicio de un nuevo ciclo escolar, instituciones educativas de todo el mundo se enfrentan al problema de asignar recursos, personal, espacios y tiempo a las actividades académicas que se desarrollarán durante el periodo que empieza [1]. Para resolver estas dificultades, se aplican conocimientos adquiridos con la experiencia de ciclos anteriores así como ayuda de software para gestionar y organizar las asignaciones (especialmente cuando se trata de instituciones con miles de alumnos). Adicionalmente, en los últimos años se ha buscado modelar estas situaciones como problemas de optimización combinatoria, con el objetivo de utilizar herramientas matemáticas para auxiliar al personal administrativo en las labores de asignación de recursos. [2]

Específicamente, se han estudiado los problemas de asignación escolar en preparatoria y asignación escolar universitaria [3], en los cuales se busca encontrar soluciones óptimas para la asignación de eventos[1]-horas, eventos-maestros, eventos-exámenes, etc. La manera en que se definen matemáticamente es a partir de restricciones *fuertes* y *suaves*. Las primeras tienen que ver con la factibilidad de una asignación (e.g. que no se asignen dos eventos en un mismo salón a la misma hora o que no hayan dos maestros asignados al mismo evento), las segundas se relacionan con características deseables en la asignación (e.g. que los alumnos no tengan horas libres entre eventos). La función objetivo del problema será una medida de cuántas restricciones fuertes y suaves se violan en una solución particular. Por otro lado, las variables de decisión representan los cambios posibles en la asignación (horas en que se dan eventos, salones asignados, días de impartición, etc.).

Actualmente, se ha implementado este tipo de modelos en más de 50 instituciones de 11 países diferentes [4], además de que existen competencias internacionales en donde se presentan nuevas propuestas de modelos y algoritmos para solucionarlos [5]. El que se presenten nuevos modelos por país y por institución tiene que ver con que cada organismo académico tiene características particulares que deben de ser adaptadas al modelo general de optimización. Así, en estas preparatorias y universidades se han estudiado modelos creados con base en sus necesidades y características únicas.

Por otro lado, la búsqueda de nuevos y mejores algoritmos se relaciona con que, en general, el

[1] El concepto de *evento* engloba cualquier actividad académica que pueda ser asignada a un *espaciotiempo* definido, por ejemplo, materias, exámenes, conferencias, etc.

problema de asignación escolar es **NP-Duro**, por lo que usualmente se utilizan matheurísticas, metaheurísticas e hiperheurísticas para intentar solucionarlo.
En este trabajo se desarrolló una propuesta de modelo del problema de asignación escolar adaptado a la Facultad de Ciencias de la Universidad Nacional Autónoma de México, con el objetivo de generar una herramienta auxiliar para la asignación de recursos en esta institución (Figura 1). Este modelo se divide en tres fases:

1. Obtención y generación de datos de estudiantes.
2. Asignación evento-hora.
3. Asignación evento-maestro.

Figura 1: Fotografía de la Facultad de Ciencias de la Universidad Nacional Autónoma de México, cortesía de [6].

Se implementaron las tres fases en dos instancias artificiales del problema: una para un semestre par y otra para semestre impar. En ambos casos se utilizaron dos algoritmos distintos: una metaheurística híbrida genética con búsqueda local variable y una matheurística genética con búsqueda local variable y agrupación. Finalmente, se analizaron las soluciones obtenidas, especialmente en búsqueda de patrones que pudieran dar indicios de cómo crear mejores asignaciones en el problema real.

Organización de la tesis

A continuación se presenta un breve resumen del contenido de los capítulos del presente trabajo con la finalidad de guiar al lector.

§1 Problemas de asignación escolar

El modelo de asignación escolar para FCUNAM se construye a partir de modificaciones a problemas generales de asignación escolar. En esta sección se presentan brevemente la notación y teoría necesarias para definir los problemas generales de asignación de eventos y maestros, así como los conceptos de optimización combinatoria y teoría de la complejidad.

§2 Modelo de asignación escolar para la FCUNAM

Esta sección está dedicada a la descripción del modelo de asignación que se propone en este trabajo, se desarrolla cada una de las tres fases del mismo. Se presentan las funciones objetivo, variables de decisión, técnicas de recopilación y tratamiento de datos y las restricciones fuertes/suaves que se tomaron en cuenta.

§3 Metodologías utilizadas

El problema de asignación escolar pertenece a la clase de problemas **NP-Duros**, por lo que no es eficiente utilizar algoritmos exactos para intentar resolverlo. Sin embargo, mediante método heurísticos es posible encontrar soluciones cercanas al óptimo en tiempos accesibles computacionalmente. En esta sección se repasan algunas técnicas comunes de optimización, desde programación lineal hasta hiperheurísticas. Después, se presentan los dos métodos que se utilizaron para la obtención de resultados: el *nuevo algoritmo híbrido* y el *algoritmo genético con búsqueda local*.

§4 Resultados y discusión

En esta sección se presentan los resultados obtenidos en dos instancias artificiales, una de semestre impar y otra de semestre par. En cada caso se muestran los óptimos conseguidos, se discuten los patrones encontrados y se analizan las asignaciones obtenidas. Finalmente, se plantean las posibilidades para futuro trabajo de investigación e implementación real del modelo como herramienta auxiliar.

§5 Conclusiones y apéndices

Aquí se presentan las conclusiones del trabajo, que incluyen observaciones generales del modelo y sugerencias para su desarrollo en el futuro. En el primer apéndice se definen los diagramas de caja y se muestra un ejemplo. El segundo apéndice es una copia del código utilizado en este trabajo.

Capítulo 1

Problemas de asignación escolar

La asignación de recursos para tareas bajo restricciones predefinidas es un problema recurrente en el sector de transporte, eventos deportivos, distribución de empleados en empresas y en horarios escolares. En la actualidad se han desarrollado diversas técnicas computacionales en las áreas de investigación de operaciones, ciencias de la computación e inteligencia artificial con la finalidad de obtener soluciones satisfactorias a este tipo de problemas [7].
En particular, los problemas de *asignación escolar* se definen como aquellos que se originan en instituciones educativas. En [8] se dividen estos problemas en dos familias: *problemas de asignación universitaria* (PAU) y *problemas de asignación en preparatoria* (PAP).
En ambos casos, el problema de asignación se modela como un problema de optimización combinatoria, donde se busca una asignación óptima de los recursos.

1.1. Optimización combinatoria

Un problema de optimización consiste en encontrar los mejores valores de un conjunto de variables con base en una función objetivo y un conjunto de restricciones. Cuando dichas variables son discretas, se trata de un *problema de optimización combinatoria*. Formalmente, un problema $\Phi = (\Omega, f)$ de optimización combinatoria se define como [9]:

- Un vector de variables $\boldsymbol{x} = (x_1, \ldots, x_n)$.
- Un dominio D_i por cada variable.
- Un conjunto de restricciones para las variables.
- Una función objetivo a maximizar o minimizar $f : D_1 \times \cdots \times D_n \to \mathbb{R}$.

El conjunto $\Omega \equiv \{(x_1, \ldots, x_n) \mid x_i \in D_i$ y se satisfacen las restricciones$\}$ se conoce como espacio factible. Una solución óptima $\boldsymbol{x}^*$ en el espacio factible cumplirá $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega$ en un problema de minimización, y $f(\boldsymbol{x}^*) \geq f(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega$ en un problema de maximización.
En teoría, encontrar todas las soluciones en Ω permitiría compararlas para encontrar la solución óptima de Φ, sin embargo, en la práctica el espacio factible es tan grande que resulta poco eficiente o incluso inviable enumerar todos sus elementos (por ejemplo, en problemas de optimización de dinámicas moleculares, a una computadora de escritorio le tardaría millones de años encontrar todos los elementos de Ω [10]). Para cuantificar la dificultad de resolver un problema se utiliza la teoría de la complejidad.

1.2. Teoría de la complejidad

En [11] Alan Turing presenta la definición formal de algoritmo basado en lenguajes formales y máquinas de Turing. En su versión simplificada, esta definición es que un algoritmo A se compone de una serie finita de instrucciones que permiten al usuario resolver un problema Ψ [12]. Por ejemplo, en un problema Φ de optimización combinatoria, un algoritmo sería una serie de pasos para encontrar la solución óptima $\boldsymbol{x}^*$.
Existen diversas métricas para medir el rendimiento de un algoritmo A:

- Tiempo de ejecución: tiempo que le toma al algoritmo resolver el problema.
- Memoria: el espacio de memoria requerido para ejecutar el algoritmo.
- Calidad de la solución.
- Robustez: capacidad del algoritmo para adaptarse a cambios en los parámetros del problema.

Como el tiempo de ejecución depende de la computadora en la que se ejecutó A, es necesario introducir una métrica diferente que sea independiente del equipo en el que se trabaje. Dicha métrica C_A se conoce como la complejidad de A, y se define como el número de pasos que utiliza A para resolver un problema de tamaño n (con n variables). Mediante la notación de Landau se definen cotas para el comportamiento asintótico de C_A. En particular, la notación $\mathcal{O}$ de la *gran O* corresponde a una cota superior: A es de complejidad $\mathcal{O}(g(n))$ si $\exists M > 0, n_0 \in \mathbb{N}$ tales que $\forall n > n_0$ se tiene $C_A \leq Mg(n)$. En la Figura 1.1 se comparan diversos tipos de complejidades computacionales.

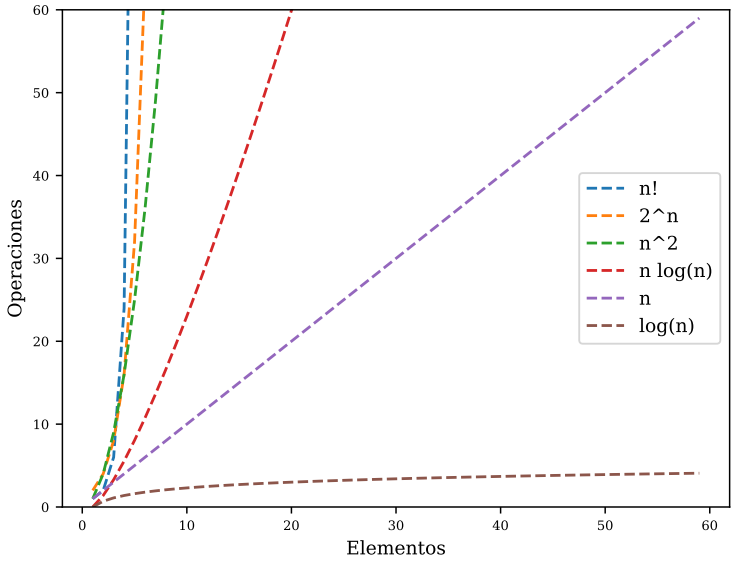


Figura 1.1: Ejemplos de gráficas de complejidades computacionales.

Un problema de decisión es aquel en el cual la solución es binaria y corresponde a una respuesta *sí* o *no*. Cada problema de optimización con función objetivo f corresponde a un problema de decisión: dado $k \in \mathbb{R}$, ¿existe una solución x_d tal que $f(x_d) = k$?.
Con estas definiciones, es común clasificar problemas en términos de la complejidad de los algoritmos que los resuelven [13]. Las clasificaciones básicas son:

- **P**: Si tiene un algoritmo con complejidad polinomial que lo resuelve.
- **NP**: Un problema de decisión es de esta clase si verificar que una solución es válida requiere un algoritmo con complejidad polinomial.
- **NP-Completo**: Un problema de decisión es de esta clase si es NP y cualquier otro problema NP puede ser reducido a él en un tiempo polinomial.
- **NP-Duro**: Un problema de optimización es NP-Duro si su problema de decisión asociado es NP-Completo.

De aquí se tiene que $P \subseteq NP$. De darse la inclusión opuesta ($NP \subseteq P$), se tendría que para todo problema NP siempre existe al menos un algoritmo polinomial que lo resuelve.

1.3. Tipos de problemas de asignación escolar

En [14] se definen los problemas de horarios como *la asignación, sujeta a restricciones, de un conjunto recursos en el espacio-tiempo, de tal manera que se satisfaga un número de objetivos.* En particular, los problemas de horarios escolares se dividen en tres clases [15]:

- **Horario de maestros**: Estos problemas se relacionan con la asignación semanal de clases en una escuela. El objetivo es asignar un conjunto de maestros al conjunto de grupos (de alumnos) en un cierto número de materias y horas, satisfaciendo ciertas restricciones dependiendo de la escuela. Por ejemplo, mientras que en una primaria cada maestro se asigna a un sólo grupo durante todo el día, en secundaria existen varios maestros, cada uno asignado a varios grupos y en diferentes materias. Otras restricciones tienen que ver con periodos de descanso para los maestros, disponibilidad de salones, clases a horas específicas, etc.

- **Horario de exámenes**: En estos problemas el objetivo principal es asignar un conjunto de exámenes a un conjunto de horas disponibles. Las restricciones principales tienen que ver con el número de alumnos que tomará cada examen, así como la cantidad de exámenes que cada alumno va a presentar.

- **Horario de eventos**: En estos problemas el propósito es la asignación de eventos (un evento es una clase, laboratorio, etc.) a horas y salones disponibles de tal manera que no existan conflictos entre los maestros, alumnos que toman los eventos y las características de cada salón.

Como se mencionó anteriormente, existen dos familias de problemas de asignación escolar: PAP y PAU. En cada familia se pueden plantear los tres tipos problemas de asignación, dependiendo de lo que se necesite, así como de la escuela en particular que se esté considerando.

1.3.1. Problema general de asignación universitaria de eventos

Los problemas de asignación universitaria pertenecen a la clase **NP-Duros**, además de que incluso encontrar soluciones factibles (aunque no sean óptimas) para problemas con un gran número de eventos puede ser poco eficiente[1] o incluso imposible en términos prácticos, pues la complejidad del problema es exponencial [17].
Para construir la función objetivo se consideran restricciones *fuertes* y *suaves*. Las restricciones fuertes son aquellas que no pueden ser violadas bajo ninguna circunstancia (las que definen Ω), mientras que existen instancias en las que es posible violar algunas restricciones suaves. Por ejemplo, una restricción fuerte es que un alumno no puede estar físicamente en dos clases al mismo

[1] En este trabajo se entiende *eficiente* como algoritmo polinomial, de acuerdo a la literatura [16]

tiempo. Por otro lado, una restricción suave puede ser evitar que un alumno tenga *horas libres* entre sus clases. La función objetivo se construye penalizando la violación de estos dos tipos de restricciones.
El objetivo del problema es encontrar una asignación en donde no se viole ninguna de las restricciones fuertes (factibilidad) y que minimice la penalización asociada a las restricciones suaves. Dicha asignación se hace de eventos a horas disponibles por día, como se puede ver esquemáticamente en la siguiente Figura:

Figura 1.2: Diagrama del problema de asignación universitaria de eventos a horas disponibles por día.

El modelo matemático descrito a continuación es una manera general en la que se plantea el problema de asignación universitaria de eventos [18], se describe únicamente para tomarlo como referencia y ejemplificar el modelado de asignaciones evento-hora. Se toma en cuenta una semana académica de 5 días con 9 horas disponibles por día. Además, se asume que cada salón tiene una capacidad específica de alumnos así como una característica particular (laboratorio, sala de cómputo, etc.). De esta manera, una solución factible asignará un evento a una hora disponible, de tal manera que se satisfagan las siguientes restricciones fuertes:

- En cada hora, un alumno sólo participa a lo más en un evento.
- Cada evento se asigna a un salón con capacidad y características adecuadas.
- En cada hora sólo se asigna un evento por salón.

Adicionalmente, se penaliza una asignación por cada ocurrencia de las siguientes restricciones suaves:

- Los eventos no deberían de ser asignados a la última hora del día.
- Ningún estudiante debería de tomar más de dos clases en horas consecutivas del día.
- Todo estudiante debería de tener al menos una clase al día.

Se toman en cuenta los siguientes conjuntos para representar la asignación como un problema de optimización combinatoria:

- $E = \{e_1, \ldots, e_n\}$ es un conjunto de n eventos.
- $R = \{r_1, \ldots, r_m\}$ es un conjunto de m salones.
- $T = \{t_1, \ldots, t_{45}\}$ es un conjunto de 45 horas.
- $S = \{s_1, \ldots, s_p\}$ es un conjunto de p alumnos
- $F = \{f_1, \ldots, f_q\}$ es un conjunto de q características de salones.

Se definen también las siguientes dos funciones:

- $G(r_k)$ son las características del salón r_k
- $C(r_k)$ es la capacidad de alumnos del salón r_k

Por último, se definen las variables de decisión del problema:

- $x_{ijkl} = \begin{cases} 1 & \text{si el alumno } s_i \text{ toma el evento } e_j \text{ a la hora } t_k \text{ en el salón } r_l, \\ 0 & \text{e.o.c.} \end{cases}$
- $y_{jlg} = \begin{cases} 1 & \text{si el evento } e_j \text{ se toma en el salón } r_l \text{ y requiere la característica } f_g, \\ 0 & \text{e.o.c.} \end{cases}$
- $z_{lg} = \begin{cases} 1 & \text{si el salón } r_l \text{ tiene la característica } f_g, \\ 0 & \text{e.o.c.} \end{cases}$

A partir de estas definiciones es posible describir las restricciones de manera formal. Las restricciones fuertes son:

- En cada hora, un alumno sólo participa a lo más en un evento.

$$\sum_{j=1}^{n}\sum_{l=1}^{m} x_{ijkl} \leq 1, \;\; i = 1, \ldots, p; \; k = 1, \ldots, 45.$$

- Cada evento se asigna a un salón con capacidad y características adecuadas.

$$\sum_{g=1}^{q} y_{jlg} \leq |G(r_l)|, \;\; j = 1, \ldots, n; \; l = 1, \ldots, m.$$

$$y_{jlg} \leq z_{lg}, \;\; j = 1, \ldots, n; \; l = 1, \ldots, m; \; g = 1, \ldots, q.$$

$$\sum_{i=1}^{p} x_{ijkl} \leq C(r_l), \;\; j = 1, \ldots, n; \; k = 1, \ldots, 45; \; l = 1, \ldots, m.$$

- En cada hora sólo se asigna un evento por salón.

$$\sum_{j=1}^{n} x_{ijkl} \leq 1, \;\; i = 1, \ldots, p; \; k = 1, \ldots, 45; \; l = 1, \ldots, m.$$

Para que una solución sea factible, es necesario que cada evento $e_1, \ldots, e_n$ sea asignado a exactamente un salón $r_1, \ldots, r_m$ y a exactamente una hora $t_1, \ldots, t_{45}$, de tal manera que las restricciones fuertes descritas anteriormente se satisfagan.
La formulación matemática de las restricciones suaves se hace mediante:

- Los eventos no deberían de ser asignados a la última hora del día.

$$\sum_{j=1}^{n}\sum_{l=1}^{m} x_{ijkl} = 0, \;\; i = 1, \ldots, p; \; k = 9, 18, \ldots, 45.$$

- Ningún estudiante debería de tomar más de dos clases en horas consecutivas del día.

$$\sum_{j=1}^{n}\sum_{l=1}^{m}\sum_{k=a}^{a+2} x_{ijkl} \leq 2, \;\; i = 1, \ldots, p; \; a = 1, 2, \ldots, 7, 10, 11, \ldots, 16, \ldots, 37, 38, \ldots, 43.$$

- Todo estudiante debería de tener al menos una clase al día.

$$\sum_{j=1}^{n}\sum_{l=1}^{m}\sum_{k=d+1}^{d+9} x_{ijkl} \geq 1, \;\; i = 1, \ldots, p; \; d = 0, 9, 18, 27, 36.$$

La solución óptima del problema se encuentra al minimizar el número de violaciones de restricciones suaves. Formalmente, la función objetivo será

$$f(\boldsymbol{w}) = \gamma F(\boldsymbol{w}) + S(\boldsymbol{w}), \tag{1.1}$$

en donde $\boldsymbol{w} = (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, las funciones F y S cuentan el número de violaciones a restricciones fuertes y suaves, respectivamente, y γ es una constante positiva que se selecciona de tal manera que violar una restricción fuerte añada más a la función objetivo que violar todas las restricciones suaves.
En este problema se asume que los maestros ya están asignados a cada evento. Sin embargo, en muchos casos es necesario hacer también esa asignación (maestro-evento) antes o después de construir los horarios que tendrán los eventos en la escuela.

1.3.2. Problema general de asignación escolar de maestros

Como en el problema de asignación escolar de eventos a horas, en el de maestro-evento se suelen tomar tanto restricciones fuertes, que tienen que ver con los contratos de maestros, las materias para las cuales están capacitados y la demanda de alumnos por semestre, como restricciones suaves, que se relacionan con exigencias particulares de cada escuela [19]. Por otro lado, no se toman en cuenta como factores variables aquellos relacionados con salones y alumnos, pues se hace la suposición de que el horario de los eventos ya ha sido elegido o se elegirá posteriormente.
Como en la sección anterior, se describe un modelo generalizado de asignación maestro-materia [20] con la finalidad de tener un punto de referencia respecto al modelo propuesto para la Facultad de Ciencias, UNAM. Primero, basada en ciertos parámetros, la administración escolar en [20] calcula la *carga de trabajo* de cada evento, medida en horas. Los parámetros usuales son:

- Número de créditos asociados al evento.
- Número de estudiantes inscritos.
- Composición del grupo (alumnos de último grado, de primer ingreso, etc.)
- Formato de enseñanza (clases especiales como tutorías o educación a distancia)
- Formas de evaluación, que incluyen exámenes, proyectos y reportes.
- Material extra, como salones de cómputo, uso de recursos y demás.

De esta manera, dos eventos que duran el mismo número de horas a la semana podrían tener una carga de trabajo (medida también en horas) completamente diferente. A partir de cierta antigüedad trabajando en la escuela, algunas universidades otorgan un *descuento por edad* en sus horas de trabajo, en estos casos se debe de tomar en cuenta el descuento particular de cada maestro al momento de calcular su carga de trabajo para el semestre. Adicionalmente, la carga se disminuye para un maestro que dará clases a dos o más grupos sobre la misma materia. Por ejemplo, en [20] se proponen decrementos en la carga de trabajo que aumentan mientras a más grupos enseña un

maestro la misma materia (e.g., un maestro que enseña Cálculo a tres grupos tendrá un mayor decremento en la carga de trabajo que uno que enseña Geometría a dos grupos).
Respecto a la elección de los maestros sobre las materias que quieren enseñar en cada ciclo escolar, cada institución suele tener un sistema particular [21]. Mientras que en algunas universidades los maestros proponen las materias que desean antes de que la administración construya los horarios, en otras se eligen todas las materias que se abrirán para el siguiente semestre y después cada maestro elige las de su preferencia. En algunas instituciones pequeñas dedicadas a la enseñanza de nivel maestría y doctorado, cada departamento tiene una reunión entre la administración y los profesores, en donde se determinan las materias que se impartirán [22].
Las restricciones fuertes de asignación maestro-evento que se consideran en [20] son:

- Cada evento debe de tener un sólo maestro.
- Los maestros deben de tener una carga de trabajo que corresponda a las horas por semestre de su contrato.

Por otro lado, se imponen dos restricciones suaves: la primera es que los maestros tengan cargas de trabajo similares, y la segunda es que impartan los eventos de su preferencia. Los datos necesarios para cuantificar esto son:

- $M = \{m_1, \ldots, m_n\}$ es un conjunto de n maestros.
- $E = \{e_1, \ldots, e_r\}$ es un conjunto de r eventos.
- $\boldsymbol{V}$ es el vector con las cargas de trabajo asociadas a los eventos.
- A es la matriz de aptitudes, donde $A_{ij} = 1$ si el maestro i es especialista en el evento j, y $A_{ij} = 0$ e.o.c.
- $\boldsymbol{a}$ es el vector con los descuentos por edad de cada maestro.
- $\boldsymbol{W}_{\min}$ y $\boldsymbol{W}_{\max}$ son las cotas del intervalo de horas que pueden ser asignadas a cada maestro, dependiendo de su contrato.
- P es la matriz de preferencias, en donde P_{ij} es la preferencia del maestro i para que se le asigne el evento j, y va del 1 al 3.

Las variables de decisión son

$$x_{ij} = \begin{cases} 1 & \text{si el maestro } i \text{ es asignado al evento } j \\ 0 & \text{e.o.c.} \end{cases} \tag{1.2}$$

Además, se utilizan dos variables extras. Un vector $\boldsymbol{W}$ para calcular la carga de trabajo de cada maestro dada una asignación particular y un escalar Z para calcular la carga de trabajo máxima posible para todos los maestros. De esta manera, la función objetivo es

$$\min_{\boldsymbol{x}} \ \alpha Z - \sum_{i=1}^{n} \sum_{j=1}^{r} P_{ij} x_{ij}. \tag{1.3}$$

Las restricciones se formulan como

$$\sum_{i=1}^{n} A_{ij} x_{ij} = 1 \ \ \forall j \in E, \tag{1.4}$$

$$\sum_{j=1}^{m} V_j x_{ij} = W_i + a_i \ \ \forall i \in M, \tag{1.5}$$

$$Z - (W_i + a_i) \geq 0 \ \ \forall i \in M, \tag{1.6}$$

$$W_{i\min} \leq W_i \leq W_{i\max} \ \ \forall i \in M, \tag{1.7}$$

$$x_{ij} \in \{0, 1\} \ \ \forall i, j. \tag{1.8}$$

Al minimizar la función objetivo Z, se asegura la igualdad entre maestros en términos de la carga de trabajo a la vez que maximiza el número de maestros que imparten eventos de su elección. La constante α se utiliza para determinar qué debería de tener un mayor peso, la igualdad entre cargas de trabajo o preferencias de eventos. La restricción (1.4) asegura que cada evento tiene un sólo maestro, la ecuación (1.5) calcula la carga de trabajo por maestro, la restricción (1.6) asegura que la carga máxima Z sea mayor que la carga de cualquier maestro, la ecuación (1.7) mantiene la carga del maestro i dentro de los límites de su contrato y la restricción (1.8) es el dominio de definición de las variables de decisión.

La teoría y modelos generales descritos en este capítulo son referencias a partir de las cuales se modeló el problema de horarios para la Facultad de Ciencias de la UNAM.

Capítulo 2

Modelo de asignación escolar para la FCUNAM

Aunque los problemas generales de asignación escolar han sido estudiados detalladamente con anterioridad y se han proporcionado distintos tipos de soluciones a los mismos [23], cada institución educativa tiene particularidades, lo que resulta en la necesidad de modelos específicos para cada una de ellas. Por ejemplo, en competencias internacionales de problemas de asignación escolar se ha trabajado con modelos para distintas preparatorias y universidades en más de 50 instituciones de 11 países diferentes [4], mientras que, de manera independiente, existen investigaciones para adaptar este problema a universidades en Irán [24], Rusia [25], USA [26], etc.
En este trabajo se desarrolló un modelo específico para resolver el problema de asignación escolar que surge en la creación de horarios semestrales en la Facultad de Ciencias de la UNAM (FCUNAM). Como cualquier otra institución educativa, la FCUNAM tiene particularidades que deben de ser consideradas para la creación de dicho modelo. Específicamente, al considerar sólo las licenciaturas en Física, Matemáticas y Actuaría, y sólo los dos primeros semestres de estas carreras, se tienen varias características importantes:

1. Los eventos tienen duraciones de 1 a 3 horas, y se dividen en: clases teóricas, laboratorios, de inglés y de cómputo. La facultad cuenta con salones con las características correspondientes a cada tipo de evento.

2. Los eventos en la facultad empiezan desde las 7:00 AM y terminan hasta las 10:00 PM.

3. Los semestres se dividen en pares e impares, dependiendo de su fecha inicio. Los que empiezan en febrero son pares, los que empiezan en agosto son impares.

4. Durante los primeros semestres, los alumnos de estas tres carreras toman tanto eventos compartidos (de tronco común) como eventos únicos de cada licenciatura, incluyendo tanto

eventos obligatorios como optativos.

5. Los eventos no están seriados, esto es, se permite que los alumnos inscriban eventos que no corresponden al semestre oficial que están cursando.

6. Si en el semestre inmediato anterior un alumno reprobó al menos un evento, el límite de eventos que puede inscribir al próximo semestre es de 6. En caso de haber aprobado todos los eventos o de obtener un permiso especial de la administración, este límite no aplica.

7. Los alumnos inscriben los eventos al inicio del semestre (mediante un sistema de recolección de firmas) basados en un horario que es publicado al final del semestre inmediato anterior en la página web de la facultad.

Además de las características propias de la FCUNAM, se considera el modelo administrativo utilizado actualmente para la generación de horarios. Dicho modelo se puede dividir en cinco fases consecutivas:

1. Se escogen los eventos que se impartirán al siguiente semestre.

2. Cada evento se asigna a un horario particular.

3. Se hace la asignación maestro-evento una vez que los maestros han elegido sus preferencias de eventos a impartir.

4. Se escogen los salones en los que se impartirá cada evento.

5. Los alumnos inscriben los eventos de su interés mediante el sistema de recolección de firmas.

Por los puntos descritos anteriormente, además de la cantidad de salones, eventos y alumnos inscritos, el problema de asignación escolar en la FCUNAM es particularmente complejo. A modo de comparación, en la siguiente Tabla se muestran las divisiones de problemas usuales de asignación escolar que, en [27], se catalogan como problemas *pequeños*, *medianos* y *grandes*, junto con el problema de la FCUNAM cuando sólo se consideran 2 semestres en el turno matutino (de 7:00 A.M. a 4 P.M.).

Parámetro	Pequeño	Mediano	Grande	FCUNAM
Eventos	100	400	400	116
Salones	5	10	10	118
Alumnos	80	200	400	1074
Tipos de salón (laboratorio, cómputo, etc.)	5	5	10	4
Número máximo de eventos por alumno	20	20	20	$6/\infty$
Número máximo de alumnos por evento	20	50	100	347

Tabla 2.1: Comparación de problemas de asignación usuales con el de la FCUNAM, datos correspondientes al semestre par.

Debido al tamaño tan grande del problema de asignación de la FCUNAM, se planteó un modelo simplificado que pudiera ser resuelto en un tiempo razonable a la vez que arrojara resultados significativos para la creación de horarios en esta institución educativa. En dicho modelo simplificado se consideraron solamente las carreras de Física, Matemáticas y Actuaría. Además, se tomaron en cuenta sólo los dos primeros semestres para el caso par y el primer semestre para el impar[1], así como un número menor de horas (turno matutino, de 7:00 A.M. a 4:00 P.M.), alumnos, eventos y salones que los estimados.
Por otro lado, se planteó este modelo en tres fases que buscan ser análogas al modelo administrativo actual de la FCUNAM. Estas fases son:

1. **Creación de instancia:**
   Se recopilan datos de la página web de la FCUNAM sobre los tipos y capacidades de cada salón. Además, se utiliza la información de semestres anteriores para hacer una estimación del número de eventos y alumnos que se esperan al siguiente semestre. De aquí se genera una instancia simplificada del problema de asignación con un conjunto artificial de alumnos y eventos esperados.

2. **Asignación evento-hora:**
   A partir de la instancia construida en el paso anterior, se resuelve el problema de asignación evento-hora tomando restricciones basadas en el conjunto artificial de alumnos esperados, así como en la capacidad de cada salón.

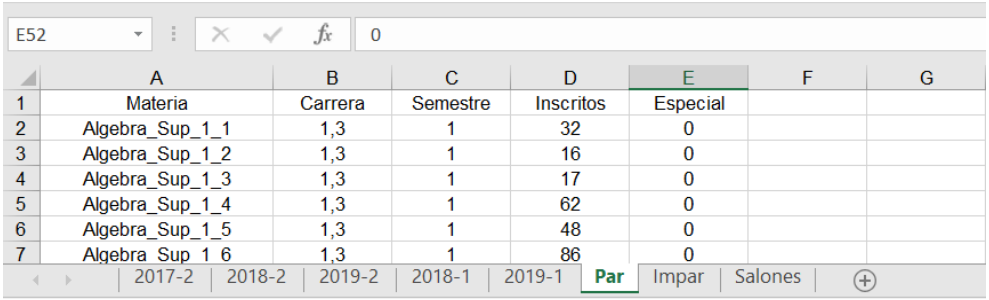
3. **Asignación evento-maestro:**
   Una vez que se ha obtenido un horario óptimo para los eventos del semestre, se realiza la asignación evento-maestro con restricciones basadas en las preferencias de cada maestro, su experiencia y una evaluación hecha por alumnos que tomaron clase con ellos anteriormente.

[1]Pues es en los primeros semestres en donde hay más conflictos entre eventos de tronco común y los de cada licenciatura por separado.

Al terminar las tres fases del modelo simplificado, se consigue un modelo de horario óptimo de eventos para el siguiente semestre en donde ya se han asignado también maestros y salones a cada evento. En las siguientes secciones se describe con detalle cada fase del modelo.

2.1. Creación de instancia

En general, en los modelos de una institución se crean instancias del problema de asignación tomando en cuenta que ya se conocen con exactitud los eventos que tomará cada alumno de la institución. Esto se debe, en parte, a su forma particular de inscripción. Por ejemplo, en muchos casos se trata de instituciones de preparatoria (en donde los alumnos no escogen eventos, sino que toman los que ya han sido establecidos para cada año de su trayectoria escolar) o de instituciones pequeñas de posgrado en donde los alumnos escogen los eventos de especialidad que tomarán desde que inician sus estudios. En cambio, en la FCUNAM no se conocen con exactitud los eventos que tomaran o que piensan tomar los alumnos al siguiente semestre. Sin embargo, esta información se puede estimar a partir de la demanda de eventos en semestres anteriores. Por ello, el primer paso en este modelo es la recolección de dicha información.
En la página web de la FCUNAM [28] se registran, para cinco semestres, los eventos que se impartieron, los salones que se utilizaron, la cantidad de alumnos inscritos a cada evento y el horario en el que se asignaron. Para la creación del modelo, primero se registraron tablas con información de los eventos en cada semestre (Figura 2.1). La base de datos incluye el nombre del evento, las carreras a las que va designado (con el código matemáticas=1, física=2, actuaría=3), el semestre al que pertenece el evento, el número de alumnos inscritos y el tipo de salón que utiliza (con códigos para diferenciar cada laboratorio de física, salón de cómputo y salón normal).
Finalmente, se obtiene un promedio de estos datos para crear las bases *par* e *impar*, que corresponden a los alumnos esperados por materia para cada tipo de semestre. Para este modelo simplificado, en los casos pares se registraron sólo eventos de los primeros dos semestres (para considerar alumnos que pasaron a segundo y que no acreditaron materias de primero), y para los impares sólo los de primer semestre (para considerar alumnos que acaban de ingresar a la FCUNAM).



E52 | 0

	A	B	C	D	E	F	G
1	Materia	Carrera	Semestre	Inscritos	Especial		
2	Algebra_Sup_1_1	1,3	1	32	0		
3	Algebra_Sup_1_2	1,3	1	16	0		
4	Algebra_Sup_1_3	1,3	1	17	0		
5	Algebra_Sup_1_4	1,3	1	62	0		
6	Algebra_Sup_1_5	1,3	1	48	0		
7	Algebra_Sup_1_6	1,3	1	86	0		

2017-2 | 2018-2 | 2019-2 | 2018-1 | 2019-1 | Par | Impar | Salones

Figura 2.1: Muestra de la base de datos utilizada en la primera fase del modelo.

En la base de datos final se hace una primera estimación del número total de alumnos esperados a_1 obteniendo el promedio de los alumnos totales en Cálculo 2 y Geometría Analítica 2 (Cálculo 1 y Geometría Analítica 1) para el semestre par (impar).
Después, se hace una estimación de los alumnos esperados por cada licenciatura promediando sobre los alumnos esperados en los eventos particulares de cada carrera. Los eventos que se utilizaron para el semestre impar se muestran en la siguiente tabla:

Alumnos de primer semestre			
Licenciatura	Matemáticas	Física	Actuaría
Evento 1	Geometría moderna 1	Álgebra para físicos	Teoría del seguro
Evento 2	-	Computación	Inglés 1
Evento 3	-	Física contemporánea	-

Tabla 2.2: Eventos utilizados para estimar la cantidad de alumnos de cada licenciatura en un semestre impar.

Los eventos utilizados para esta estimación en el semestre par se muestran a continuación:

Alumnos de segundo semestre			
Licenciatura	Matemáticas	Física	Actuaría
Evento 1	Optativas del nivel I-IV	Laboratorio de mecánica	Contabilidad
Evento 2	-	Mecánica vectorial	Inglés 2
Evento 3	-	-	Programación

Tabla 2.3: Evento utilizados para estimar la cantidad de alumnos de cada licenciatura en un semestre par.

Al promediar sobre el número de alumnos esperados en los eventos de las tablas anteriores se

obtiene un estimado de los parámetros

$$p_i = \begin{cases} \text{número de matemáticos} & \text{si } i = 1, \\ \text{número de físicos} & \text{si } i = 2, \\ \text{número de actuarios} & \text{si } i = 3. \end{cases} \tag{2.1}$$

Con los datos de los alumnos esperados de cada licenciatura se hace un segundo estimado a_2 del número total de alumnos que se esperan ($a_2 = p_1 + p_2 + p_3$). El estimado final de alumnos totales se obtiene mediante

$$p = \text{máx}(a_1, a_2). \tag{2.2}$$

Se calcula el número total de alumnos de esta manera porque se encontró que en algunos semestres hay alumnos que cursan los eventos de tronco común (a_1) pero no inscriben algunos de los eventos correspondientes a su licenciatura (a_2) o viceversa[2]. Al tomar el máximo de ambas cantidades se puede crear una instancia del problema de asignación en donde se considere esta situación en donde no todos los alumnos inscriben las materias que corresponden a su semestre.
Para generar dicha instancia se definen dos estructuras. La primera es una matriz M de n por 6, en donde n es el número total de eventos. Cada renglón i de esta matriz corresponde a un evento, y tiene la forma

$$[M]_i = (\text{mat}_i, \text{fis}_i, \text{act}_i, \text{sem}_i, \text{esp}_i, \text{cod}_i), \tag{2.3}$$

en donde las variables de entrada se definen como

$$\begin{aligned} \text{mat}_i &= \begin{cases} 1 & \text{si los matemáticos pueden inscribir el evento } i, \\ 0 & \text{e.o.c.}, \end{cases} \\ \text{fis}_i &= \begin{cases} 1 & \text{si los físicos pueden inscribir el evento } i, \\ 0 & \text{e.o.c.}, \end{cases} \\ \text{act}_i &= \begin{cases} 1 & \text{si los actuarios pueden inscribir el evento } i, \\ 0 & \text{e.o.c.}, \end{cases} \\ \text{sem}_i &= \text{semestre en que se imparte } i, \\ \text{esp}_i &= \text{número de alumnos esperados para } i, \\ \text{cod}_i &= \text{código de repetición de } i. \end{aligned}$$

La última variable, el código de repetición, es una etiqueta a cada evento de diferente categoría. Es decir, todos los grupos de Cálculo 1 tienen un código, los grupos de Inglés 1 otro y así sucesivamente.

[2]Este fenómeno se encontró en la base de datos, en ningún semestre coincidieron el número de alumnos inscritos a eventos obligatorios y a eventos optativos.

La otra estructura es una matriz P que contiene la instancia de alumnos. Es una matriz de p por 8, en donde cada renglón corresponde a un alumno. En esta etapa se incrementan (o disminuyen) simultáneamente en pasos de una unidad los valores p_1, p_2, p_3 de tal manera que $p_1 + p_2 + p_3 = p$. Se hace este ajuste manual para hacer congruentes tanto el estimado total de alumnos como el de alumnos por licenciatura. Los primeros p_1 renglones de P corresponden a matemáticos, los siguientes p_2 a físicos y los últimos p_3 a actuarios. Un renglón i de esta matriz tiene la forma

$$[P]_i = (\text{ins}_i, \text{lic}_i, 0, 0, 0, 0, 0, 0). \tag{2.4}$$

Las dos variables se definen mediante

$$\text{ins}_i = \text{semestre al que está inscrito el alumno } i,$$
$$\text{lic}_i = \begin{cases} 1 & \text{si el alumno es de matemáticas,} \\ 2 & \text{si el alumno es de física,} \\ 3 & \text{si el alumno es de actuaría.} \end{cases}$$

Las otras 6 entradas representan los eventos inscritos por cada alumno, comienzan en 0 para después distribuir los eventos de la base de datos en estas entradas. El algoritmo para hacer dicha distribución empieza listando todos los eventos, comenzando por los del semestre *principal*[3] y después los del semestre de materias que no fueron aprobadas anteriormente. Para cada evento se obtienen las carreras a las que se puede impartir, de aquí se genera una lista de alumnos disponibles que pueden ser asignados a este evento. Para ello, se verifica también que los alumnos no hayan llegado ya al límite de 6 eventos, además de que no tengan asignado un evento con el mismo código de repetición (así se evita que alguien inscriba, por ejemplo, dos eventos de Cálculo 1 al mismo tiempo). Se va asignando el evento de manera ponderada a los alumnos de la lista hasta llegar a los esp_i alumnos que se esperaban para el evento. El algoritmo se muestra a continuación:

[3]Es decir, el semestre que corresponde oficialmente: 2do para par y 1ro para impar

Require: M, P.
Ensure: P con eventos distribuidos.
1: Crea lista L de n eventos ordenados por semestre *principal* y no *principal*.
2: **for** $i = 1$ to n **do**
3: $e \leftarrow$ evento i de L.
4: $x \leftarrow \text{esp}_e$
5: Crea lista A de alumnos que pueden tomar el evento e y que no han llegado al límite de 6 eventos.
6: **while** $x > 0$ **do**
7: $a \leftarrow$ alumno aleatorio de A
8: $w(a) \leftarrow$ número de eventos a los que ya está inscrito a
9: Muestrea q en unif$(0, 1)$
10: **if** $q < \text{pond}(w)$ **then**
11: Asigna evento e al alumno a.
12: $x \leftarrow x - 1$
13: **else**
14: Continúa
15: Escribe archivo con P.

Tabla 2.4: Algoritmo para generar instancia de alumnos artificiales.

Como se puede ver, para la asignación de eventos se pondera utilizando la función

$$\text{pond}(w) = -\frac{7}{50}w + 1. \tag{2.5}$$

Se construyó esta ponderación (Figura 2.2) con la ecuación usual de una recta $\Delta y = m\Delta x$, de tal manera que la probabilidad de asignación de un evento e a un alumno p decrece linealmente desde 1 cuando p no tiene ningún otro evento inscrito hasta 0.3 cuando p tiene otros 5 eventos inscritos. De esta manera, será más probable asignar un evento a los alumnos que tengan menos eventos registrados, con lo que se evita que hayan muchos alumnos con el máximo de 6 inscritos y otros con 1 o ninguno.

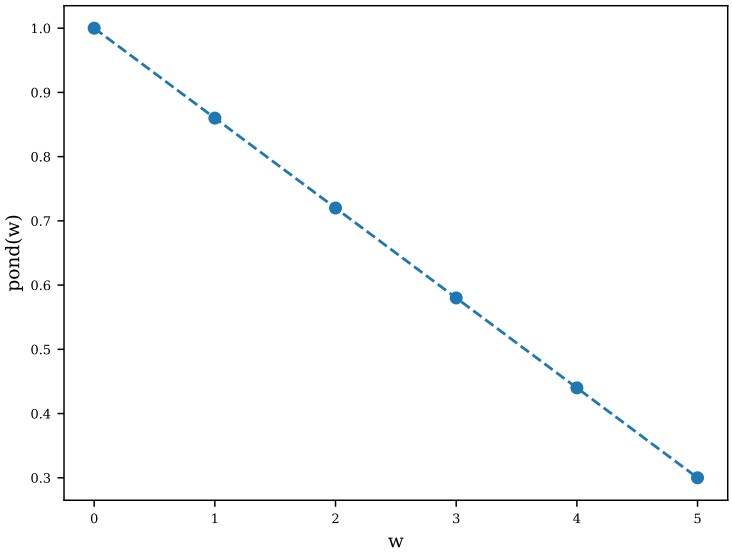


Figura 2.2: Función pond(w) para la asignación de eventos a alumnos.

Al terminar, se obtiene la matriz P actualizada que codifica la instancia de alumnos artificiales que tomarán los eventos de M. Con dicha instancia, ya se puede plantear de manera habitual el problema de asignación escolar para la FCUNAM, que es la siguiente fase del modelo.

2.2. Asignación evento-hora

Como se mencionó anteriormente, cada escuela tiene particularidades que deben de ser consideradas al plantear el problema de asignación en la misma. Por ejemplo, en [20] y [18] se trabaja con instituciones pequeñas o medianas[4], por lo que se utilizan vectores y matrices con entradas binarias y con dimensiones del número de alumnos, eventos, salones, etc. En cambio, en [29] se estudia el modelado de asignación para una preparatoria, y, debido al tamaño del problema, se propone utilizar vectores y matrices más pequeñas pero que contengan entradas en $\mathbb{Z}$ en lugar de $\mathbb{N}$. Así, incrementan las posibilidades en cada entrada de las variables del problema, pero se reduce el uso de memoria computacional.
En este trabajo se optó por seguir el mismo esquema y modelar el problema de asignación con variables que toman valores en los enteros. Para el *input* del algoritmo se utiliza la matriz P creada en el paso anterior además de dos matrices enteras adicionales. La primera es la matriz E de n

[4]Con base en la Tabla presentada en la sección anterior.

por 4, en donde n es el número total de eventos. Cada renglón i de E tiene la forma

$$[E]_i = (\text{tip}_i, \text{hrs}_i, \text{dia}_i, \text{esp}_i), \tag{2.6}$$

en donde

tip_i = código con el tipo de salón necesario (laboratorio, normal o cómputo) para el evento i,
hrs_i = número de horas que dura i por día,
dia_i = número de días que se imparte i a la semana,
esp_i = número de alumnos esperados para i.

La segunda matriz de entrada es S, con dimensiones de m por 2, en donde m es el número de salones disponibles. Sus renglones son del tipo

$$[S]_i = (\text{cap}_i, \text{tip}_i), \tag{2.7}$$

en donde

cap_i = capacidad del salón i,
tip_i = código con el tipo de salón que es i (laboratorio, normal o computo).

Con las tres matrices P,E y S queda definido por completo el *input* del algoritmo. Por otro lado, se tienen cuatro variables de decisión. La primera es el vector columna X^1 de dimensión n. En donde

$$X^1{}_i = \text{Salón asignado al evento } i, \tag{2.8}$$

es decir, en esta variable se guardan los datos que corresponden a la asignación de salones para cada evento. La segunda variable X^2 es una matriz en donde se guardan las horas a las que se imparte cada evento, sus dimensiones son de n por 9 y cada uno de sus renglones tiene la forma

$$[X^2]_i = (h_{i1}, h_{i2}, h_{i3}, h_{i4}, h_{i5}, h_{i6}, h_{i7}, h_{i8}, h_{i9}), \tag{2.9}$$

en donde h_{ij} es una variable binaria que vale 0 cuando el evento i no se imparte a la hora j, y 1 e.o.c. En total, hay nueve horas de clase, comenzando a las 7:00 A.M. y terminando a las 4:00 P.M.
La tercera variable X^3 corresponde a la información de los días en los que se imparte cada evento, se trata de una matriz de n por 5, con renglones del tipo

$$X^3{}_i = (d_{i1}, d_{i2}, d_{i3}, d_{i4}, d_{i5}), \tag{2.10}$$

en donde d_{ij} vale 0 cuando el evento i no se imparte en el día j, y 1 e.o.c. Se toma en cuenta una semana laboral usual en México (de 5 días: lunes a viernes) para definir esta matriz.

Por último, se requiere de una variable X^4 en donde se guarde el horario de cada alumno. Se define como un tensor de p por 5 por 9, en donde cada sub-matriz de 5 por 9 representa el horario de un alumno, y se define mediante

$$[X^4]_i = \begin{bmatrix} a_{i11} & a_{i12} & a_{i13} & \dots & a_{i19} \\ a_{i21} & a_{i22} & a_{i23} & \dots & a_{i29} \\ a_{i31} & a_{i32} & a_{i33} & \dots & a_{i39} \\ a_{i41} & a_{i42} & a_{i43} & \dots & a_{i49} \\ a_{i51} & a_{i52} & a_{i53} & \dots & a_{i59} \end{bmatrix} \tag{2.11}$$

en donde las entradas son

$$a_{ijk} = \begin{cases} 0 & \text{si el alumno } i \text{ no toma ningún evento a la hora } j \text{ en el día } k, \\ \text{e} & \text{si el alumno } i \text{ toma sólo el evento } e \text{ a la hora } j \text{ en el día } k, \\ \text{-l} & \text{si el alumno } i \text{ tiene } \textit{empalme} \text{ de } l \text{ eventos a la hora } j \text{ en el día } k. \end{cases} \tag{2.12}$$

De esta manera, una solución completa al problema de asignación de la FCUNAM es una lista

$$\boldsymbol{X} = [X^1, X^2, X^3, X^4]. \tag{2.13}$$

La ventaja de utilizar matrices enteras es que se puede codificar una gran cantidad de información en menos memoria a comparación del modelo de referencia descrito en el capítulo 1. Por ejemplo, en X^4 se puede saber no sólo a qué hora toman eventos los alumnos, sino exactamente qué evento tienen asignado y si tienen uno o más asignados a la misma hora. Si se hubiera utilizado un X^4 binario, tendría que haber sido un tensor X^4_{ijkl} de p por n por 5 por 9 para saber si el alumno i toma el evento j a la hora k en el día l.
La función objetivo se construye a partir de restricciones *suaves* y *fuertes*. Las fuertes son aquellas relacionadas con la factibilidad de la solución, es decir, con que el horario correspondiente cumpla ciertas restricciones básicas:

1. En cada hora, los alumnos sólo toman un evento (no se permiten *empalmes*).
2. Cada evento se asigna a un salón que tenga la capacidad y equipamiento necesario.
3. En cada hora, ningún salón puede ser asignado a más de un evento.
4. Los eventos se dan durante las horas necesarias (1,2 o 3) de manera continua, i.e., un evento que dure más de 1 hora no puede impartirse por partes en diferentes horas.

Se define la función $F(\boldsymbol{X})$ asociada a restricciones fuertes como

$$F(\boldsymbol{X}) = \sum_{i=1}^{4} F_i(\boldsymbol{X}), \tag{2.14}$$

en donde cada $F_i(\boldsymbol{X})$ cuenta el número de violaciones totales por semana de la restricción fuerte i en el horario $\boldsymbol{X}$.
Por otro lado, se tiene sólo una restricción suave, que es

1. Los alumnos deben de tener el menor número posible de *horas libres*.

Al imponer esta restricción suave se garantiza que un horario óptimo sea cómodo para los alumnos. Se define $S(\boldsymbol{X})$, que cuenta el número de horas libres por semana que tiene cada alumno en el horario $\boldsymbol{X}$. Así, la función objetivo es

$$f(\boldsymbol{X}) = \lambda F(\boldsymbol{X}) + S(\boldsymbol{X}), \tag{2.15}$$

en donde λ es una constante de penalización que asegura que una solución no factible no sea mejor que una factible. El problema de asignación es minimizar esta función, en donde el mejor horario posible es aquel para el que $f(\boldsymbol{X}) = 0$.
Una vez que se resuelve esta fase del modelo, se obtiene una solución $\boldsymbol{X}_{opt}$ que corresponde a un horario óptimo para los eventos del semestre. De acuerdo al esquema de la FCUNAM, se presenta dicho horario a los profesores para pasar a la siguiente asignación, la de evento-maestro.

2.3. Asignación evento-maestro

Así como el problema de asignación evento-hora requiere de la creación de una instancia de alumnos, para el problema de evento-maestro se necesita, en primer lugar, la generación de una base de datos relacionados con cada maestro adscrito a la institución educativa en cuestión. De nuevo, en cada artículo se utilizan diferentes parámetros dependiendo de las características particulares de cada escuela. En este trabajo se utilizó como principal referencia la investigación realizada en [20], debido, principalmente, a la profundidad del análisis que ahí se desarrolla, pues es un trabajo dedicado exclusivamente a la asignación evento-maestro. Sin embargo, existen modificaciones significativas en este modelo particular de 3 fases, mismas que se plantearon de acuerdo a las necesidades específicas de la FCUNAM.
La base de datos necesaria para esta fase tiene, para cada maestro, sus horas de trabajo semanales de acuerdo a su contrato, sus años de antigüedad, sus preferencias de eventos para enseñar (en orden descendente, del que más les interesa impartir al que menos les interesa), la lista de eventos en los cuales el maestro es experto y una evaluación hecha por alumnos que anteriormente cursaron eventos con él. Aunque actualmente no se utiliza este último parámetro para la asignación evento-maestro en esta escuela, se decidió incluirlo para explorar un espacio de soluciones más diverso.
A diferencia de la primera fase, para este problema no existen datos públicos que pudieran ser utilizados para generar una instancia artificial de maestros. Por ello, se creó una base completamente artificial generada de forma aleatoria. En la Figura (2.3) se aprecia una muestra de la misma.

	A	B	C	D	E	F	G	H
1	Maestro	Horas	Antiguedad	Evaluacion	Preferencias	Experto		
2	Galeote	5	0	10	2,1,3,54	2,1,3,54		
3	Alejandro	15	1	9	3,4,14,21	3,4,14,21		
4	Rebeca	6	3	10	5,3,1,55	5,3,1,55		
5	Misael	10	10	7	8,7,6,3	8,7,6,3		
6	Paola	10	20	2	1,2,3,4	1,2,3,4		
7	Pedro	15	7	8	1,2,8,9	1,2,8,9		

datos

Figura 2.3: Muestra de la base de datos artificial de maestros.

Aunque no se consideraron datos reales de profesores para la creación de la base de datos, sí se tomaron en cuenta el número posible de horas de enseñanza de acuerdo con datos de los semestres **2019-1** y **2019-2**. En la página web de la FCUNAM hay un registro de las materias que enseñaron los profesores en estos dos últimos semestres. De aquí se observó que, en general, existen 7 posibilidades de horas de trabajo docente al día[5]:

Posibilidad	Eventos de una hora	Eventos de dos horas	Horas totales por día
1	3	0	3
2	2	0	2
3	1	0	1
4	0	1	2
5	0	2	4
6	1	1	3

Tabla 2.5: Posibles horas de enseñanza en la FCUNAM.

La séptima posibilidad es enseñar un evento de laboratorio con duración de tres horas, dos veces a la semana, este caso se trata por separado más adelante. Con base en estos datos, para los rubros de preferencias, *experto* y horas de enseñanza semanales se utilizó la variable aleatoria (v.a.) m_1 distribuida uniformemente en el conjunto $\{1, 2, 3, 4, 5, 6\}$ para que a cada maestro artificial le correspondiera a una posibilidad de la Tabla 2.5. Para el rubro de antigüedad se utilizó la v.a. m_2 uniforme en $[1, 30] \cap \mathbb{Z}$. Finalmente, se usó la v.a. m_3 uniforme en $[1, 10] \cap \mathbb{Z}$ para la evaluación de cada maestro.

Así, dado un maestro, primero se muestrea un valor de m_1 para asignarle las horas por semana, después, se le asignan aleatoriamente cuatro eventos a sus preferencias y a *experto*[6] del tipo que

[5]Horas de enseñanza en la FCUNAM, sin tomar en cuenta actividades de investigación, enseñanza en posgrado, preparación de clases, etc.

[6]Se asume que todos los maestros van a querer impartir sólo eventos en los que son expertos.

sea necesario (de 1 o 2 horas, dependiendo del valor de m_1). Por último, se asignan los años de antigüedad y la evaluación muestreando m_2 y m_3, respectivamente. Se repite este proceso hasta generar el número l de maestros que se quieren en la base de datos (Tabla 2.6).

Require: l.
Ensure: Base artificial de l maestros.
1: Crea documento en formato xlsx.
2: Define las tres v.a. m_1, m_2 y m_3.
3: **for** $i = 1$ to l **do**
4: Muestrea un valor de m_1 y asigna horas de trabajo de i de acuerdo a dicho valor.
5: $EVE \leftarrow$ lista de eventos que pueden ser asignados a las preferencias del maestro i de acuerdo al valor de m_1.
6: Eventos preferidos y *experto* $\leftarrow$ asigna-preferencias(EVE,i)
7: Asigna años de antigüedad de i mediante un valor muestreado de m_2.
8: Asigna evaluación de i mediante un valor muestreado de m_3.
9: Escribe los datos en el renglón i del archivo xlsx.

Tabla 2.6: Algoritmo para generar instancia de maestros artificiales.

Cada maestro tendrá cuatro eventos en sus preferencias y en *experto*. Sin embargo, si se asignaran estos 4 eventos de EVE de manera completamente aleatoria, se podrían tener dos casos problemáticos: eventos que no fueron asignados a las preferencias de ningún maestro y eventos que aparezcan en las preferencias de un gran número de maestros. Para evitar estos dos posibles conflictos se utilizó la función asigna-preferencias(EVE,i). Su pseudocódigo es:

Require: EVE, i.
Ensure: Asignación de 4 eventos a las preferencias y a *experto* del maestro i.
1: $x \leftarrow 4$.
2: **while** $x > 0$ **do**
3: $e \leftarrow$ evento aleatorio de EVE.
4: **if** e no está en las preferencias de otro maestro **then**
5: Asigna e a las preferencias de i
6: $x \leftarrow x - 1$
7: **else**
8: ...

Tabla 2.7: Algoritmo de la función asigna-preferencias(EVE,l).

8:	...
9:	$w \leftarrow$ número de maestros que tienen a e en sus preferencias.
10:	Muestrea q en unif(0,1)
11:	**if** $q <$dist(w) **then**
12:	Asigna e a las preferencias de i
13:	$x \leftarrow x - 1$
14:	**else**
15:	Continúa

Tabla 2.7: Algoritmo de la función asigna-preferencias(EVE,l) (continuación).

La función dist: $[1, 19] \to \mathbb{R}$ que aparece en el paso 10 está definida como

$$\text{dist}(w) = -\frac{1}{19}w + \frac{20}{19}. \tag{2.16}$$

Se construyó dist(w) a partir de la ecuación de recta $\Delta y = \Delta x$, de tal manera que, si un evento e está en las preferencias de sólo un maestro, se asigne con probabilidad 1, si está en las preferencias de 20 maestros la probabilidad de asignación sea 0, y las otras probabilidades sean valores en la recta que une ambos puntos. Su gráfica se muestra a continuación:

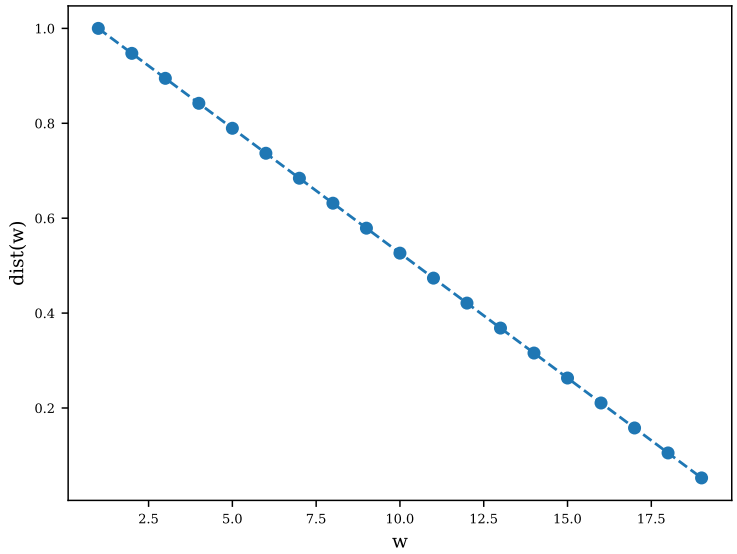


Figura 2.4: Función dist(w) para la asignación de eventos a las preferencias de los maestros.

Mediante dist(w) y asigna-preferencias(EVE,l) se limita el número máximo de maestros que

quieran impartir un evento a 19[7], además de que, mientras más maestros quieren impartir un evento e, es menos probable que ocurra una nueva asignación del mismo. De esta manera, se promueve la asignación de eventos que aún no estén en las preferencias de nadie. Con estos procedimientos se consigue una instancia artificial de maestros con sus respectivas preferencias de todos los eventos que se impartirán al siguiente semestre, con excepción de los laboratorios de Física. Para este último caso, se agregan manualmente maestros a la base pues, de acuerdo a la información recopilada en la página web de la FCUNAM, en general los docentes de laboratorio no imparten eventos extra además de las dos clases de laboratorio por semana. Estos maestros agregados manualmente tienen 6 horas de trabajo por semana, su antigüedad y evaluación también se generan de manera aleatoria. Una vez terminada la base de datos, se traslada su información a la matriz entera $\widetilde{M}$ de l por 7. Los renglones de $\widetilde{M}$ tienen la forma

$$[\widetilde{M}]_i = (\text{hrs}_i, \text{ant}_i, \text{eval}_i, \text{pref}_{i1}, \text{pref}_{i2}, \text{pref}_{i3}, \text{pref}_{i4}), \tag{2.17}$$

en donde

$$\begin{aligned} \text{hrs}_i &= \text{horas de contrato semanales del maestro } i, \\ \text{ant}_i &= \text{años de antigüedad de } i, \\ \text{eval}_i &= \text{evaluación de } i, \\ \text{pref}_{ij} &= \text{preferencia } j\text{-ésima de evento que } i \text{ desea impartir.} \end{aligned}$$

La matriz $\widetilde{M}$ es el primer dato de entrada para esta fase, el segundo es el vector columna H de longitud n, en donde

$$H_i = \text{número de horas que se imparte a la semana el evento } i. \tag{2.18}$$

Con las dos matrices $\widetilde{M}$ y H queda definido por completo el *input* del algoritmo. Además, se tendrá una variable de decisión: la matriz $\boldsymbol{Y}$ de l por 3[8], definida mediante

$$\boldsymbol{Y}_{ij} = \begin{cases} e & \text{si se ha asignado el evento } e \text{ al maestro } i, \\ 0 & \text{e.o.c.} \end{cases} \tag{2.19}$$

Análogamente a la fase dos, toda la información de una solución al problema de asignación evento-maestro queda codificada en $\boldsymbol{Y}$.
Para la función objetivo en este caso, se vuelven a considerar restricciones fuertes y suaves. Las fuertes son:

[7] El límite en 19 se escogió con base en el número de maestros *artificiales* utilizado en esta instancia.

[8] Aunque se permite que los maestros seleccionen 4 eventos preferidos, el número máximo de posibles eventos asignados a un maestro, de acuerdo con la Tabla 2.5, es de 3.

1. Los maestros imparten eventos en los cuales son expertos.
2. Los maestros imparten eventos de sus preferencias.
3. Ningún maestro trabaja más de lo establecido en sus horas semanales.

Se define la función $\widetilde{F}(\boldsymbol{Y})$ asociada a restricciones fuertes como

$$\widetilde{F}(\boldsymbol{Y}) = \sum_{i=1}^{3} \widetilde{F}_i(\boldsymbol{Y}), \tag{2.20}$$

en donde cada $\widetilde{F}_i(\boldsymbol{Y})$ cuenta el número de violaciones de la restricción fuerte i en la asignación $\boldsymbol{Y}$. Se tienen tres restricciones suaves, que son:

1. Los maestros imparten los eventos en los primeros órdenes de sus preferencias.
2. Los maestros elegidos para impartir eventos son aquellos con mejores evaluaciones.
3. Los maestros elegidos para impartir eventos son aquellos con más años de antigüedad.

Cada una de estas restricciones pone la prioridad de la solución en un rubro específico que se espera en los maestros elegidos para impartir eventos (satisfacción de los maestros, evaluación y experiencia, respectivamente). La función asociada a las restricciones suaves también se expresa como una suma:

$$\widetilde{S}(\boldsymbol{Y}) = \alpha\widetilde{S}_1(\boldsymbol{Y}) + \beta\widetilde{S}_2(\boldsymbol{Y}) + \gamma\widetilde{S}_3(\boldsymbol{Y}). \tag{2.21}$$

En este caso, cada función aporta la contribución total del cumplimiento de un rubro en específico. Por ejemplo, $\widetilde{S}_1(\boldsymbol{Y})$ mide la satisfacción total de preferencias. Cada maestro tiene 4 eventos que desearía impartir, de los cuales se le asignan, a lo más, 3. Estos eventos son los que se acomodan en orden descendente de preferencia en la matriz $\widetilde{M}$. Luego, para cada maestro, $\widetilde{S}_1(\boldsymbol{Y})$ suma un 5 si el maestro tiene asignado el evento que seleccionó en primer lugar, un 3 si tiene asignado el evento en segundo lugar, 1 para el tercero y 1 para el último. En caso de que no tenga asignado ningún evento de sus preferencias, no se suma nada. En total, si un maestro tiene asignados los dos primeros eventos que quería impartir y alguno de los últimos dos, contribuiría con $5+3+2=10$ a la función. Por otro lado, $\widetilde{S}_2(\boldsymbol{Y})$ mide la evaluación total asociada a la solución: suma la evaluación de todos los maestros que tienen al menos un evento asignado. Análogamente, $\widetilde{S}_3(\boldsymbol{Y})$ calcula la antigüedad total asociada a $\boldsymbol{Y}$ sumando la antigüedad ponderada a_{pond} de cada maestro que tiene al menos un evento asignado. Dicha ponderación se calcula mediante

$$a_{\text{pond}}(x) = \min(\lceil \frac{x}{2} \rceil, 10), \tag{2.22}$$

en donde $\lceil\ \rceil$ es la función techo, y x son los años de antigüedad. Se utiliza (2.22) para que la antigüedad de cada maestro se traduzca en una contribución del 1 al 10.
Finalmente, los coeficientes α, β y γ son variables entre 0 y 1 que se escogen de acuerdo al rubro que se quiera priorizar en la solución, y se impone la condición de normalización:

$$\alpha + \beta + \gamma = 1. \tag{2.23}$$

La función objetivo es

$$\widetilde{f}(\boldsymbol{Y}) = -\mu\widetilde{F}(\boldsymbol{Y}) + \widetilde{S}(\boldsymbol{Y}), \tag{2.24}$$

en donde μ es una constante de penalización que asegura que una solución no factible no sea mejor que una factible. El problema de asignación es maximizar esta función, y la mejor asignación posible depende de los valores de α, β y γ.
Al resolver esta fase se obtiene una solución $\boldsymbol{Y}_{opt}$ que corresponde a una asignación óptima de maestros-eventos. El óptimo total del modelo en 3 fases es la pareja $(\boldsymbol{X}_{opt},\boldsymbol{Y}_{opt})$ que corresponde a la optimización completa de las asignaciones involucradas en la creación de horarios en la FCUNAM.

Capítulo 3

Metodologías utilizadas

Como se mencionó en el primer capítulo, los problemas de asignación escolar pertenecen a una clase de problemas llamados *problemas de optimización combinatoria*, en donde el objetivo es encontrar una solución óptima dado un conjunto de restricciones. En los últimos años se han desarrollado una gran variedad de métodos para resolver este tipo de problemas [30], en los cuales se utilizan diversas técnicas matemáticas y computacionales.
En la siguiente sección se presentan algunas de estas metodologías con el objetivo de ofrecer un panorama general del área de optimización matemática, además de justificar la elección de algoritmos particulares que se utilizaron en este trabajo.

3.1. Técnicas de optimización [31]

La rama de optimización matemática es un subconjunto del área conocida como Investigación de Operaciones (IDO). De manera informal, la investigación de operaciones puede ser definida como:

> la aplicación de métodos científicos y matemáticos para el estudio y análisis de problemas que involucran sistemas complejos [32].

El rango de aplicaciones de la IDO incluye problemas de decisión como administración de cadenas de suministro, planificación de horarios en la industria aerocomercial, desarrollo de planes de logística para transporte de bienes, diseño de redes de telecomunicaciones, finanzas, etc. Estos problemas se describen en términos de un modelo matemático que involucra diversas variables y relaciones entre ellas. Las variables usualmente representan decisiones específicas, y el problema incluye una *función objetivo* con la cual se da una medida del impacto de las decisiones tomadas (a través del valor de las variables asociadas a las mismas). Así, la solución óptima del modelo representa el conjunto de las mejores decisiones.

3.1.1. Programación lineal

Un problema lineal es aquel en donde las restricciones y la función objetivo son lineales. En su forma canónica, puede ser expresado como

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.a} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}, \\ & \boldsymbol{x} \geq 0, \end{aligned} \tag{3.1}$$

en donde $\boldsymbol{x}$ es el vector de variables de decisión, $\boldsymbol{c}$ y $\boldsymbol{b}$ son vectores de coeficientes, $\boldsymbol{A}$ es una matriz de coeficientes y $[\ ]^T$ es el operador transpuesta.
Se ha demostrado que este tipo de problemas pueden ser resueltos en un tiempo polinomial al utilizar el método *elipsoidal* [33]. Otro de los métodos más utilizados en programación lineal, especialmente en los softwares comerciales, es el método *simplex* en su versión primal y dual [34], aunque la complejidad de este algoritmo es exponencial [35]. Todos estos algoritmos son exactos, es decir, siempre pueden encontrar la solución al problema de optimización lineal (cuando ésta existe).

3.1.2. Programación entera

Otra de las metodologías más comunes en IDO es la programación entera. Estos métodos se utilizan cuando se requiere que las variables del problema tomen valores exclusivamente en $\mathbb{Z}$. Cuando sólo se necesita que un subconjunto de las variables sean enteras, se utiliza el término *programación entera-mixta*.
Se puede encontrar una cota de la solución óptima de un problema entero mediante la técnica de relajación, que consiste en eliminar la restricción de que las variables pertenezcan únicamente a $\mathbb{Z}$ y resolver el problema de programación lineal resultante. Actualmente, la mayoría de los programas comerciales que resuelven problemas enteros utilizan relajaciones para después intentar acercar la cota encontrada al óptimo real del problema [36]. Adicionalmente, existen otras técnicas como la *descomposición en dos fases* y el algoritmo de *Branch-and-Price* [37].

3.1.3. Programación no lineal

En la programación no lineal se estudian problemas de optimización en los cuales la función objetivo o las restricciones tienen términos no lineales. Formalmente, se tienen un conjunto $X \subseteq \mathbb{R}^n$, funciones f, g_i y h_j que van de X a $\mathbb{R}$ para todo $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, p\}$, con $m, p \in \mathbb{N}$. Un

problema no lineal tiene la forma

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(x) \\ \text{s.a} \quad & g_i(x) \leq 0 \ \forall i, \\ & h_j(x) = 0 \ \forall j, \\ & x \in X. \end{aligned} \tag{3.2}$$

Dependiendo del tipo específico de funciones involucradas en el problema y de X, los problemas de este tipo pueden ser resueltos con algoritmos de *optimización convexa*, *programación cuadrática*, *programación separable*, etc. [38]

3.1.4. Heurísticas

Como se mencionó anteriormente, todo problema de programación lineal puede ser resuelto en tiempo polinomial con algoritmos exactos. Sin embargo, con problemas enteros y no lineales utilizar este tipo de algoritmos puede ser poco eficiente, situación que empeora mientras más grandes sean los espacios de búsqueda [39].
Las heurísticas son otro tipo de métodos de optimización que se basan en técnicas de aproximación para encontrar soluciones cada vez mejores de manera eficiente [40]. Es decir, el objetivo de una heurística es producir una solución aproximada del problema en un lapso de tiempo razonable. Aunque esta solución puede no ser el verdadero óptimo, la rapidez con la que se encuentra hace que estos algoritmos sean utilizados frecuentemente en problemas **NP-Duros**, especialmente cuando se trata de sistemas complejos que involucran un gran número de variables. En estos casos, las heurísticas suelen ser la única opción viable para encontrar soluciones aproximadas [41].
Entre los algoritmos heurísticos más utilizados se encuentran los algoritmos glotones, de búsqueda local, búsquedas estocásticas, recocido simulado, búsqueda tabú, algoritmos genéticos, etc.

3.1.5. Matheurísticas, Metaheurísticas e Hiperheurísticas

En la mayoría de las fuentes citadas en este trabajo se encontraron definiciones distintas e incluso conflictivas de *metaheurísticas* y *heurísticas*. Por ejemplo, mientras que en [42] todos los algoritmos discutidos se definen como heurísticos, en [43] esos mismos algoritmos se definen como metaheurísticos. Por otro lado, en [44] se hace una distinción, pues se menciona que, mientras las heurísticas tienen que adaptarse a las necesidades de cada problema, una metaheurística es más general y no requiere de la introducción de nuevas modificaciones significativas para uno u otro problema. Finalmente, en [45] se distinguen las heurísticas de las metaheurísticas de acuerdo a la

cantidad de parámetros que se tienen que ajustar en cada algoritmo para adaptarse a un problema en particular. En este trabajo se toman *heurística* y *metaheurística* como sinónimos.
Actualmente, también se utilizan algoritmos híbridos que combinan métodos exactos con heurísticas, a este tipo de metodologías se le conoce como matheurísticas. El objetivo de una matheurística es combinar la exactitud y control de los métodos clásicos con la rapidez de las heurísticas [46]. En [47] se describe la hibridación en términos de una estructura de maestro-esclavo, en donde la heurística toma el rol de maestro, actuando de manera global, y el algoritmo exacto actúa localmente como esclavo de la heurística.
Por último, las hiperheurísticas son métodos que buscan automatizar el proceso de selección, hibridación e incluso generación de heurísticas para la solución de problemas de optimización. Incorporan técnicas de *machine learning* e inteligencia artificial. Una de sus motivaciones es la creación de sistemas que puedan detectar de manera automática el mejor procedimiento para cada nuevo problema de optimización.

3.1.6. Preprocesamiento

El preprocesamiento consiste en modificar el problema de manera que sea más fácil de resolver. Usualmente funciona mediante la reducción del espacio de búsqueda a partir del procesamiento de la información del problema (restricciones y función objetivo). También existe el preprocesamiento a partir de soluciones iniciales, en este caso, se alteran dichas soluciones antes de iniciar el algoritmo principal (heurística o exacto) [48].

3.2. Metaheurísticas para el problema en la FCUNAM

Durante los últimos 20 años, el problema de asignación en universidad y preparatoria ha sido estudiado ampliamente por varios investigadores, por lo que se han propuesto diferentes metodologías para su solución. En [49] se describen las propuestas con heurísticas evolutivas, algoritmos multi-criterio, hiperheurísticas y métodos adaptativos. Otra propuesta consiste en convertir el problema a una gráfica no dirigida y resolverlo con algoritmos de coloramiento [50]. También se han utilizado técnicas de razonamiento basado en restricciones [51], en donde se van añadiendo las restricciones del horario secuencialmente. De las heurísticas utilizadas destacan también el algoritmo de la colonia de hormigas [52], recocido simulado [53] y búsqueda tabú [54].
Para este trabajo se utilizaron modificaciones de los dos métodos presentados en [55]. En dicha investigación (publicada en el año 2014) se estudia un algoritmo híbrido de búsqueda local con heurística genética, y se propone una modificación al mismo que consiste en añadir un preprocesamiento al problema utilizando agrupaciones, ambos métodos se comparan con otros de los algoritmos más utilizados en la actualidad y demuestran ser igual o más eficientes en la mayoría de

los casos. Por esta comparación, y por lo novedoso de la propuesta híbrida, fue que se escogieron estos dos métodos.

3.3. Algoritmo genético con búsqueda local

El primer método es un algoritmo genético con búsqueda local (AGBL). En la primera parte de esta sección se explican las heurísticas genética y de búsqueda local variable por separado para después presentar la hibridación de ambos.

3.3.1. Heurísticas genéticas

Las heurísticas genéticas son algoritmos estocásticos que fueron desarrollados en un intento de imitar los mecanismos de selección natural y evolución. Así, los algoritmos genéticos operan sobre un conjunto de soluciones o *población* que va evolucionando con el tiempo de manera aleatoria pero estructurada. En primer lugar, cada miembro de la población tiene que tener asociado un genotipo, que es una codificación que lo caracteriza de manera única. Después, partiendo de una población inicial, se aplican una serie de operadores en la población para crear generaciones consecutivas que mejorarán con el tiempo. La manera en la que se mide dicha mejora es por medio de una *función de fitness*, que asocia un número a cada individuo y que se va maximizando en cada generación. Las restricciones del problema se suelen integrar a la función de fitness con un término de penalización. Existen tres operadores de reproducción que actuarán sobre la población n-ésima para producir la siguiente generación. El primero es un operador de selección, que se encarga de escoger ciertos individuos de la población que pasarán al *mating pool* para aparearse. En este trabajo se utilizó el operador de ruleta de casino, pues, de acuerdo con [56], este operador, además de sencillo, es efectivo para producir horarios óptimos en el problema de asignación escolar. La ruleta de casino funciona en dos fases:

1. Asocia una distribución de probabilidad a la población, en donde la probabilidad de cada individuo se relaciona con su valor de fitness.

2. Uno a uno, escoge los individuos que pasarán al *pool* utilizando dicha distribución probabilística.

Esta técnica es análoga a una ruleta de casino, en donde cada segmento de la rueda es proporcional al fitness de un individuo. En la Figura 3.1 se ve un diagrama de este operador en una población de cuatro individuos. El que tiene peor fitness tiene una probabilidad de 0.15 de ser escogido para el *mating pool*, mientras que el de mejor fitness tiene una probabilidad de 0.35.

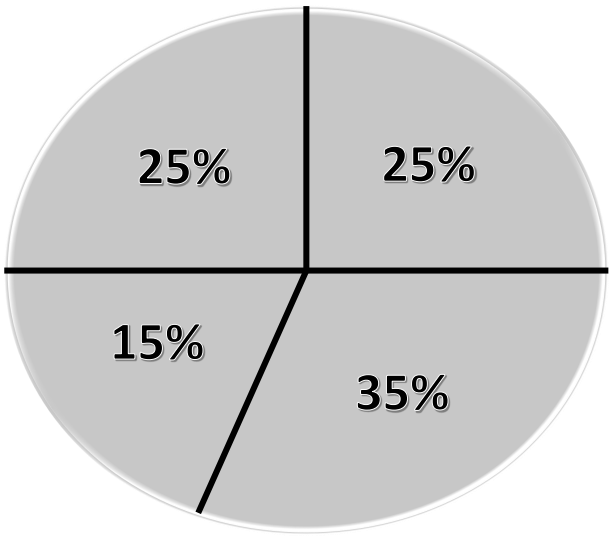


Figura 3.1: Diagrama del operador de ruleta de casino en una población con cuatro individuos.

Una vez que se ha determinado número de individuos en el *pool*, se aplica el operador de entrecruzamiento. En este proceso se seleccionan aleatoriamente un par de individuos del *mating pool* para aparearlos, propiciar la recombinación genética y así crear una o más soluciones hijas que formarán parte de la siguiente generación. Después, se utiliza un operador de mutación, en el cual se alteran, de manera aleatoria, los cromosomas de cada solución hija, propiciando la diversidad genética en la población. Por último, en algunas variaciones de la heurística genética se utiliza el *esquema elitista* [57], que consiste en la preservación del mejor cromosoma en cada población. El individuo con mejor fitness se copia automáticamente a la siguiente generación sin pasar por mutaciones. De esta manera, se asegura que en cada población se tendrá al menos un individuo con un fitness igual o mejor que en la población anterior. Se repiten estos pasos hasta que se cumpla un criterio de paro, que puede ser el número total de generaciones, valor de la función objetivo o tiempo de ejecución del programa. En la Tabla 3.1 se muestra el pseudocódigo de una heurística genética con criterio de paro sobre el número de generaciones.

Require: n número de generaciones, m probabilidad de mutación, p_1 tamaño de población, p_2 tamaño de *pool*.
Ensure: Resolver problema de optimización.
1: Crea población inicial P_0.
2: ...

Tabla 3.1: Algoritmo de la heurística genética con esquema de elitismo.

```
2: ...
3: Evalua el fitness de cada individuo.
4: X_m ← mejor individuo de la población.
5: for i = 1 to n do
6:    Aplica operador de selección para crear pool con p_2 participantes.
7:    P_k ← ∅.
8:    P_k ← P_k ⋃ {X_m}
9:    while |P_k| < p_1 do
10:      Selecciona parejas del pool y crea nuevos individuos con el operador de entrecruzamiento.
11:      Genera número aleatorio rnd
12:      if rnd < m then
13:         Muta a los nuevos individuos.
14:      else
15:         No mutar.
16:      Añade los nuevos individuos a P_k
17:   P_0 ← P_k
18:   X_m ← mejor individuo de la población.
```

Tabla 3.1: Algoritmo de la heurística genética con esquema de elitismo (continuación).

3.3.2. Búsqueda local variable

La heurística de búsqueda local parte de una solución candidata y, de manera iterada, se mueve a soluciones vecinas que mejoren el valor de la función objetivo. Para ello, se requiere de al menos una estructura de vecindad basada en cambios a las variables de decisión. Por ejemplo, cuando las variables son continuas, las vecindades pueden ser bolas abiertas en $\mathbb{R}$ (Figura 3.2). Entre las versiones más utilizadas del algoritmo de búsqueda local se encuentran:

- Búsqueda local aleatoria: Dada una solución x_0, escoge un vecino al azar y, si mejora la función objetivo, actualiza la solución, en caso contrario escoge otro vecino aleatorio.

- Escalada de montaña: Dada una solución x_0, busca de entre todos los vecinos al que tenga el mejor valor de función objetivo.

- Búsqueda local iterada: Es una modificación de escalada de montaña en donde, una vez encontrado un óptimo local, se hacen modificaciones al mismo para propiciar la exploración de otros sectores del espacio de búsqueda.

- Recocido simulado: Funciona de manera análoga a un sistema termodinámico. Se asocia una temperatura al problema y se define la probabilidad de dar un paso de búsqueda local con base en esta temperatura. Al ir calentando y enfriando el sistema se propicia la salida de mínimos locales.

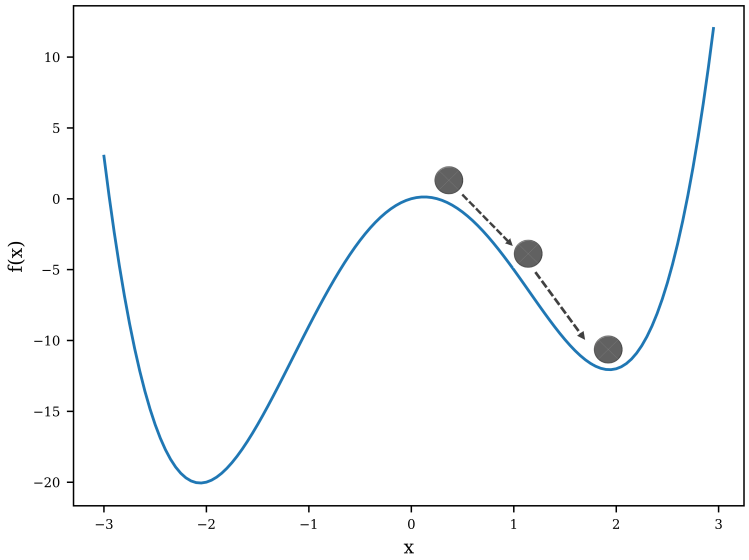


Figura 3.2: Ejemplo de un algoritmo de búsqueda local (escalada de montaña) para minimizar la función $f(x) = 2x - 8x^2 + x^4$. Las vecindades son intervalos abiertos en $\mathbb{R}$

En estas versiones de búsqueda local se utiliza la misma estructura de vecindad durante todo el algoritmo. Sin embargo, existen problemas multidimensionales en donde es útil considerar más de un tipo de vecindad. El método de búsqueda local variable es una escalada de montaña sobre una vecindad $V(\boldsymbol{X})$ compuesta de n sub-vecindades $V_i(\boldsymbol{X})$. Partiendo de una solución $\boldsymbol{X}_0$, busca entre los vecinos respecto a $V_1(\boldsymbol{X})$ al que tenga el mejor valor de función objetivo. Si ningún vecino tuvo una mejora, busca respecto a $V_2(\boldsymbol{X})$, si no encuentra vecinos con mejoras, busca respecto a $V_3(\boldsymbol{X})$, y así sucesivamente. En cuanto encuentra un vecino $\boldsymbol{X}_1$ con un mejor valor de función objetivo, reinicia la búsqueda partiendo de $\boldsymbol{X}_1$.
Por ejemplo, en una función objetivo con dominio de dos dimensiones (Figura 3.3), se puede definir $V_1(\boldsymbol{X})$ como la vecindad que considera únicamente intervalos abiertos en el eje x, y $V_2(\boldsymbol{X})$ como aquella que considera los intervalos abiertos en el eje y. El algoritmo de búsqueda local variable intentará encontrar mejoras a la solución, primero buscando sobre el eje de las ordenadas, y luego sobre el de las abscisas.

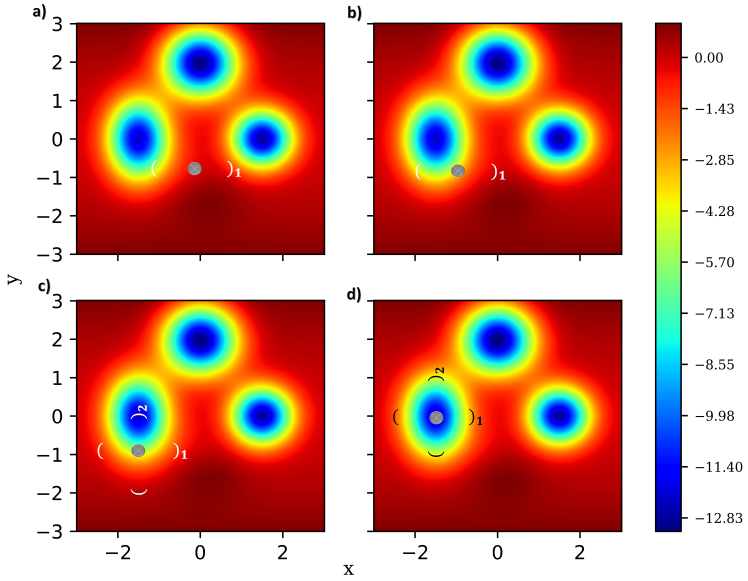


Figura 3.3: Ejemplo del algoritmo de búsqueda local variable para minimizar la función $f(x,y) = -12.5\exp\left(-\frac{(x+1.5)^2}{0.5} + \frac{y^2}{0.98}\right) - 12.5\exp\left(-\frac{(x-1.5)^2}{0.5} + \frac{y^2}{0.5}\right) - 13\exp\left(-\frac{x^2}{0.72} + \frac{(y-2)^2}{0.72}\right) - 0.1[3(1-x)^2\exp\left(-x^2-(y+1)^2\right) - 10(x/5) - x^3]$. Las imágenes denotan la secuencia del método, de a) a d). En a) y b) se aplica $V_1(\boldsymbol{X})$. En c) primero se usa $V_1(\boldsymbol{X})$, como no encuentra una mejor solución, se pasa a $V_2(\boldsymbol{X})$. Finalmente, en d) se utilizan $V_1(\boldsymbol{X})$ y $V_2(\boldsymbol{X})$, con ninguna de las dos vecindades se encuentra una mejor solución.

A continuación se presenta el pseudocódigo de la heurística de búsqueda local variable:

Require: n estructuras de vecindad $V_i(\boldsymbol{X})$, $\boldsymbol{X}_0$ solución inicial.
Ensure: Resolver problema de optimización.

```
1: i ← 1.
2: while i ≤ n do
3:    V ← número de vecinos en V_i(X)
4:    for j = 1 to V do
5:       if Vecino X_j mejora la solución then
6:          X_0 ← X_j
7:          ...
```

Tabla 3.2: Algoritmo de la heurística de búsqueda local variable.

```
7:          ...
8:          i ← 1
9:          Salir del ciclo for
10:       else
11:         Continuar
12:   i ← i + 1
```

Tabla 3.3: Algoritmo de la heurística de búsqueda local variable.

La heurística híbrida AGBL incorpora las heurísticas genética y de búsqueda local variable en un esquema de preprocesamiento (Figura 3.4). Mientras el algoritmo maestro es la heurística genética, la búsqueda local variable actúa en cada una de las soluciones hijas.

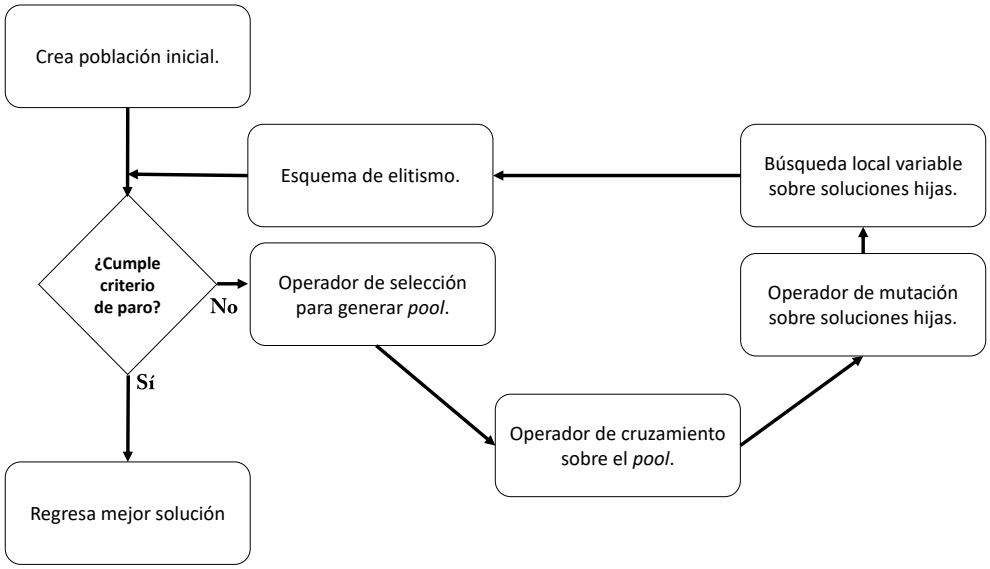


Figura 3.4: Diagrama de flujo de AGBL.

De esta manera, los integrantes de la nueva población pasan por un preprocesamiento que intenta mejorar su genética. El preprocesamiento de un individuo $\boldsymbol{X}_0$ se divide en dos fases. En la primera, que se conoce como *fase de construcción*, se aplica una búsqueda local variable que compara a los vecinos únicamente en términos de la componente *fuerte* de la función objetivo. En esta fase se busca encontrar o *construir* una solución que viole el menor número posible de restricciones fuertes y, en el mejor de los casos, que se encuentre en la zona factible del espacio de búsqueda. Al terminar la fase de construcción, se tiene un óptimo local $\boldsymbol{X}_1$ y empieza la segunda fase, llamada *fase de mejoramiento*, que consiste en otra búsqueda local variable pero comparando

vecinos en términos de la función objetivo completa. Al terminar la fase de mejoramiento se obtiene un óptimo local $\boldsymbol{X}_2$ que pasa a reemplazar al individuo $\boldsymbol{X}_0$. Se repite este procedimiento para cada miembro de la población.
Este es el funcionamiento general de AGBL, los detalles de estructura de cromosomas y tipos de vecindades dependen del problema específico en el que se utilice. En este trabajo se implementaron dos variantes de este algoritmo, una para la asignación evento-hora y otra para evento-maestro. Cabe mencionar que en ambas variaciones se introdujo un parámetro extra ω, que es la frecuencia con la que se aplica la búsqueda local variable a una población antes de pasar a la siguiente generación. Aunque en [18] se utiliza la búsqueda local en cada iteración de AGBL, aquí se introdujo dicho parámetro para tener más control sobre la velocidad de convergencia del algoritmo.

3.3.3. AGBL en la asignación evento-hora

Como se describió en el capitulo 2, en el modelo de asignación evento-hora las soluciones son del tipo

$$\boldsymbol{X} = [X^1, X^2, X^3, X^4], \tag{3.3}$$

en donde cada miembro de la lista codifica parte de la información del horario asociado a la solución. De manera resumida, se tiene:

- X^1 guarda la información de los salones asignados.
- X^2 guarda la información de las horas a las que se dan los eventos.
- X^3 guarda la información de los días en los que se dan los eventos.
- X^4 guarda la información del horario de cada alumno.

El primer paso para utilizar AGBL es poder generar una $\boldsymbol{X}$ inicial. Para ello, se hacen asignaciones aleatorias de salones a eventos (para crear X^1) y de las horas y días en que se imparte cada evento (para crear X^2 y X^3, respectivamente). Finalmente, se calcula X^4 de la información de los eventos que toma cada alumno y de los valores de X^1, X^2 y X^3.
Luego, para cada individuo se consideró que su genotipo es simplemente $\boldsymbol{X}$. Por otro lado, la función $g(\boldsymbol{X})$ de fitness fue

$$g(\boldsymbol{X}) = \frac{1}{f(\boldsymbol{X}) + 1}, \tag{3.4}$$

pues, como el objetivo es minimizar la función objetivo $f(\boldsymbol{X})$, este fitness aumentará mientras mejor sea una solución. Dada una población de tamaño p_1, la distribución $\rho(\boldsymbol{X})$ para crear el *mating pool* es

$$\rho(\boldsymbol{X}) = \frac{g(\boldsymbol{X})}{\sum_{i=1}^{p_1} g(\boldsymbol{X}_i)}, \tag{3.5}$$

en donde la suma se hace sobre todos los miembros $\boldsymbol{X}_i$ de la población.
Dado el *pool*, se van escogiendo aleatoriamente parejas de soluciones para aplicarles el operador de entrecruzamiento y crear nuevos individuos. En este caso se eligió que cada par $\boldsymbol{X}_1$, $\boldsymbol{X}_2$ generara dos nuevas soluciones $\boldsymbol{X}_{\text{h1}}$, $\boldsymbol{X}_{\text{h2}}$. El entrecruzamiento se realiza por etapas. En primer lugar, se desarrollan las estructuras X^1 de las soluciones hijas de la siguiente manera:

1. Se definen X^1_{h1} y X^1_{h2} con ceros en todas sus entradas.

2. Se genera un número aleatorio *rand*, si $rand < 0.5$, la solución X^1_{h1} hereda la entrada 1 de X^1_1 y X^1_{h2} la hereda de X^1_2. En caso contrario, X^1_{h1} la hereda de X^1_2 y X^1_{h2} de X^1_1.

3. Se repite el paso 2 para llenar todas las entradas de X^1_{h1} y X^1_{h2}.

De manera análoga, en las matrices X^2 y X^3 de las soluciones hijas se heredan renglones del padre o de la madre, de manera aleatoria (Figura 3.5). Finalmente, las estructuras X^4 se calculan a partir de las anteriores (los horarios de cada alumno sólo dependen de los horarios de los eventos).

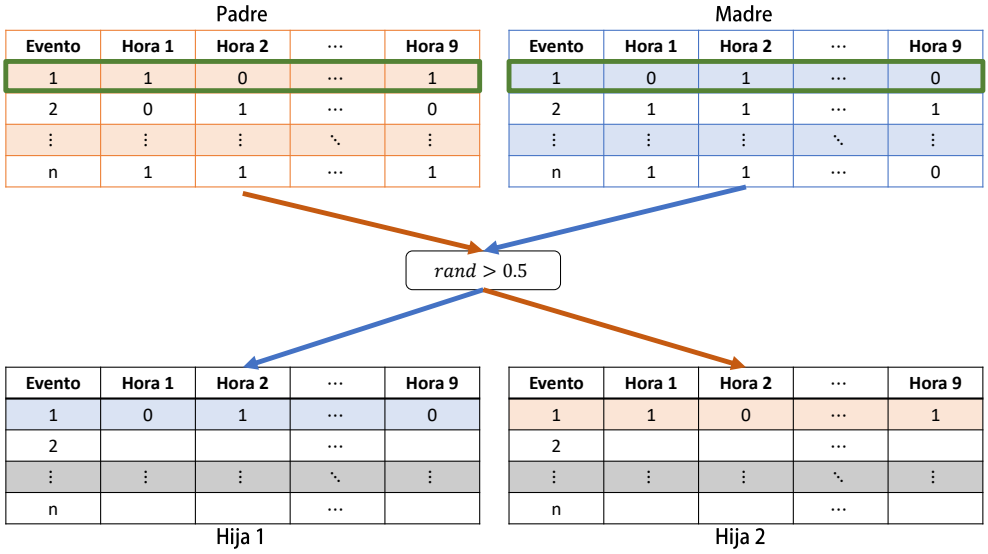


Figura 3.5: Esquema del operador de entrecruzamiento para construir dos X^2 hijas. Se selecciona el primer renglón en las soluciones padres. Dependiendo del valor de *rand*, estos primeros renglones se heredan a alguna de las soluciones hijas. Se repite este proceso para los n renglones.

Cada individuo de la nueva población tiene una probabilidad p_m de sufrir mutaciones. El operador de mutación Ω se definió como

$$\Omega(\boldsymbol{X}) = V_4' \circ V_3' \circ V_2' \circ V_1'(\boldsymbol{X}), \tag{3.6}$$

en donde la función $V_i^{'}$ escoge un vecino aleatorio respecto a la vecindad i, $\forall i \in \{1, 2, 3, 4\}$, y $\circ$ denota la composición de funciones.
La vecindad $V_1(\boldsymbol{X})$ consiste en todas las soluciones que son idénticas a $\boldsymbol{X}$ salvo que dos eventos tienen sus salones intercambiados (Figura 3.6).

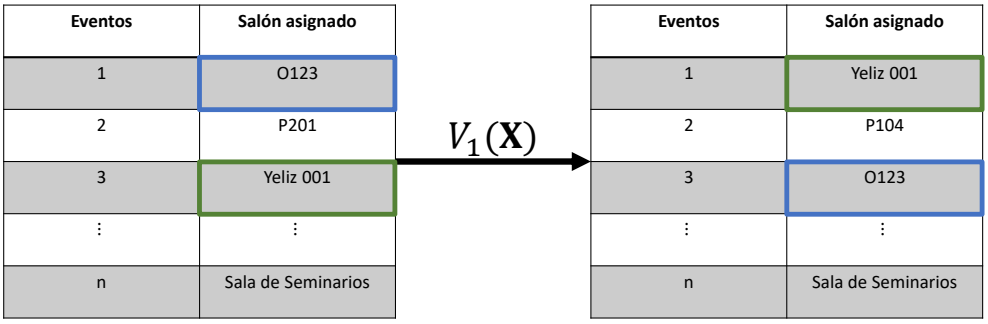


Figura 3.6: La primera vecindad intercambia el salón asignado a un par de eventos.

En la vecindad $V_2(\boldsymbol{X})$ se encuentran las soluciones en donde, dado un evento e, los días en los que se imparte son diferentes (Figura 3.7). Esta vecindad se convierte en la función identidad para eventos que se imparten toda la semana.

Eventos	Lunes	Martes	Miércoles	Jueves	Viernes
1	0	1	0	1	0
2	0	0	0	0	1
3	1	1	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮
n	1	0	0	1	0

$V_2(\mathbf{X})$

Eventos	Lunes	Martes	Miércoles	Jueves	Viernes
1	0	1	0	1	0
2	0	0	0	0	1
3	0	0	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮
n	1	0	0	1	0

Figura 3.7: La segunda vecindad cambia los días en que se imparte un evento.

De manera análoga, en $V_3(\boldsymbol{X})$ están los vecinos que, dado un evento e, cambian las horas de impartición del mismo (Figura 3.8). Estos cambios de horas se hacen en *bloques*, de manera que si un evento se imparte 2 horas o más, el cambio de horario se hace sobre todas las horas como un sólo bloque. Así se evita que este tipo de eventos se segmenten a lo largo del día.

Eventos	Hora 1	Hora 2	Hora 3	⋯	Hora 9
1	0	1	0	⋯	0
2	0	0	0	⋯	1
3	1	1	0	⋯	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	0	0	⋯	0

$V_3(\mathbf{X})$

Eventos	Hora 1	Hora 2	Hora 3	⋯	Hora 9
1	0	1	0	⋯	0
2	0	0	0	⋯	1
3	0	1	1	⋯	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	0	0	⋯	0

Figura 3.8: La tercera vecindad cambia las horas en que se imparte un evento.

Finalmente, la cuarta vecindad $V_4(\boldsymbol{X})$ selecciona dos eventos con duración de una hora e intercambia sus horarios de impartición (Figura 3.9). Cuando los dos eventos se dan a la misma hora, esta vecindad se vuelve la identidad.

Eventos	Hora 1	Hora 2	Hora 3	⋯	Hora 9
1	0	1	0	⋯	0
2	0	0	0	⋯	1
3	1	1	0	⋯	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	0	0	⋯	0

$V_4(\mathbf{X})$

Eventos	Hora 1	Hora 2	Hora 3	⋯	Hora 9
1	1	0	0	⋯	0
2	0	0	0	⋯	1
3	1	1	0	⋯	0
⋮	⋮	⋮	⋮	⋱	⋮
n	0	1	0	⋯	0

Figura 3.9: La cuarta vecindad intercambia el horario de un par de eventos.

Estas mismas cuatro vecindades se utilizaron para las fases de construcción y mejoramiento de la búsqueda local variable. Se definieron de esta manera para explorar el espacio de búsqueda respecto a los parámetros principales en la creación de un horario: días, horas y salones asignados.

3.3.4. AGBL en la asignación evento-maestro

En esta fase del modelo las soluciones están representadas por una sola matriz $\boldsymbol{Y}$ que contiene información de los eventos que impartirá cada maestro. Para la creación de soluciones iniciales, sólo se asignan aleatoriamente los eventos al conjunto de maestros.
De manera análoga a la fase anterior, el genotipo de cada solución se tomó como su matriz asociada $\boldsymbol{Y}$. Sin embargo, en este caso se desea maximizar la función objetivo $\widetilde{f}(\boldsymbol{Y})$, por lo que el fitness $\widetilde{g}(\boldsymbol{Y})$ se definió como

$$\widetilde{g}(\boldsymbol{Y}) = |\widetilde{f} - \widetilde{f}_{\text{peor}}| + 1, \tag{3.7}$$

en donde $\widetilde{f}_{\text{peor}}$ es el valor de $\widetilde{f}$ asociado a la peor solución en la población. Así, el fitness de una solución es proporcional a su *distancia* (medida en términos de $\widetilde{f}$) de la peor solución en la población. Se utilizó valor absoluto porque $\widetilde{f}$ puede tomar valores negativos. La distribución $\widetilde{\rho}$ para el *matching pool*, con una población de tamaño p_1, fue

$$\widetilde{\rho}(\boldsymbol{Y}) = \frac{\widetilde{g}(\boldsymbol{Y})}{\sum_{i=1}^{p_1} \widetilde{g}(\boldsymbol{Y})}, \tag{3.8}$$

en donde la suma se hace sobre todos los pobladores $\boldsymbol{Y}_i$.
Del *pool* se irán escogiendo aleatoriamente las parejas para crear la siguiente generación. Para esta fase también se eligió que cada pareja $\boldsymbol{Y}_1$, $\boldsymbol{Y}_2$ generara dos soluciones hijas $\boldsymbol{Y}_{h1}$ y $\boldsymbol{Y}_{h2}$. El operador de entrecruzamiento actúa de la siguiente manera:

1. Se definen $\boldsymbol{Y}_{h1}$ y $\boldsymbol{Y}_{h2}$ con ceros en todas sus entradas.
2. Determina el maestro maes_1 que imparte el evento e_1 en la solución $\boldsymbol{Y}_1$.
3. Determina el maestro maes_2 que imparte el evento e_1 en la solución $\boldsymbol{Y}_2$.
4. Genera un número aleatorio *rand*, si $rand < 0.5$, se asigna e_1 al maestro maes_1 en $\boldsymbol{Y}_{h1}$ y al maestro maes_2 en $\boldsymbol{Y}_{h2}$. En caso contrario, $\boldsymbol{Y}_{h1}$ hereda la asignación de maestro de $\boldsymbol{Y}_2$, y $\boldsymbol{Y}_{h2}$ de $\boldsymbol{Y}_1$.
5. Se repite este procedimiento para todos los eventos.

De esta manera, las soluciones hijas heredan las asignaciones de cada evento de alguno de sus padres (Figura 3.10). En este esquema de recombinación genética existe un caso particular que es problemático: cuando un maestro r ya tiene asignados los cuatro eventos disponibles y en la siguiente iteración de cruzamiento se intenta asignarle un quinto evento al mismo maestro de parte de alguno de los padres. En este caso, dicho evento se asigna de manera aleatoria a algún maestro que tenga espacio disponible.

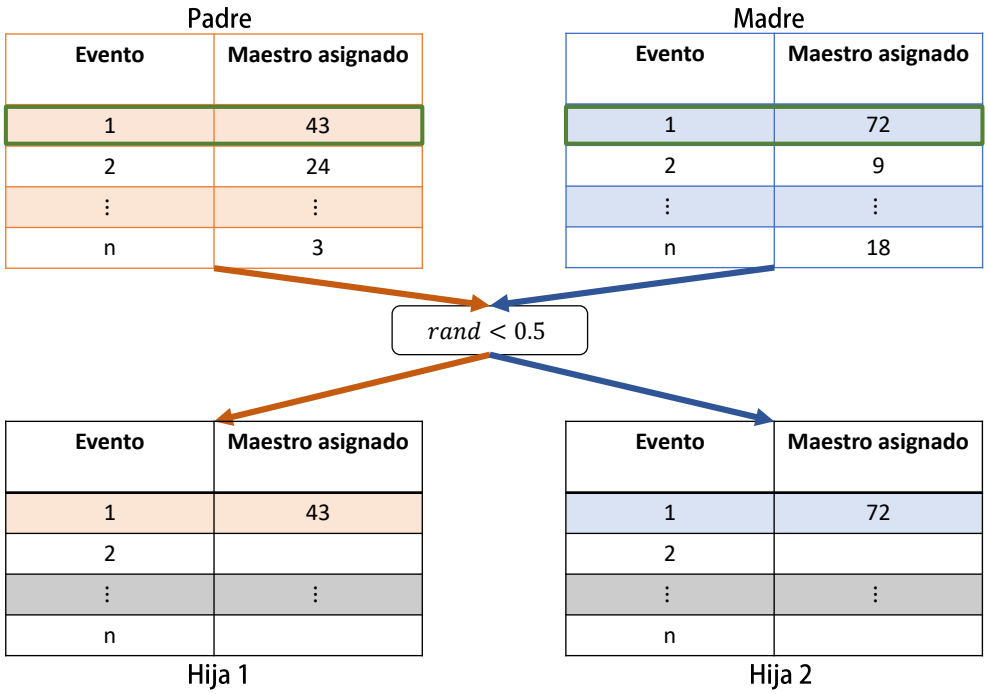


Figura 3.10: Esquema del operador de entrecruzamiento para construir dos Y hijas. Primero, se determina qué maestra imparte e_1 en cada Y padre. Dependiendo del valor de $rand$, cada hija hereda esta asignación de maestra del padre o de la madre. Se repite este proceso para todos los eventos.

En esta fase, el operador de mutación $\widetilde{\Omega}$ se definió como

$$\widetilde{\Omega}(\boldsymbol{Y}) = \text{sample}(\widetilde{V}_4', \widetilde{V}_3', \widetilde{V}_2', \widetilde{V}_1', \boldsymbol{Y}), \tag{3.9}$$

en donde la función $\widetilde{V}_i'$ escoge un vecino aleatorio respecto a la vecindad i, $\forall i \in \{1, 2, 3, 4\}$, y sample aplica aleatoriamente una de estas cuatro funciones a $\boldsymbol{Y}$. En la asignación evento-hora, cada una de las vecindades representaba una pequeña perturbación en $\boldsymbol{X}$, por lo que la mutación aplicaba las cuatro de manera consecutiva. Sin embargo, en este caso las vecindades (descritas a continuación) realizan cambios significativos en $\boldsymbol{Y}$, por lo que la mutación utiliza sólo alguna de las cuatro posibilidades.
La vecindad $\widetilde{V}_1$ consiste en intercambiar los eventos asignados a un par de maestros (Figura 3.11).

Maestros	Evento 1	Evento 2	Evento 3
1	0	0	0
2	23	45	0
3	29	0	0
⋮	⋮	⋮	⋮
l	12	0	0

$\tilde{V}_1(\boldsymbol{Y})$

Maestros	Evento 1	Evento 2	Evento 3
1	0	0	0
2	12	0	0
3	29	0	0
⋮	⋮	⋮	⋮
l	23	45	0

Figura 3.11: La primera vecindad intercambia los eventos asignados a un par de maestros.

En $\widetilde{V}_2$ se aplica una reflexión respecto al centro geométrico de los renglones de $\boldsymbol{Y}$ (Figura 3.12). Cada solución tiene un único vecino, y aplicar dos veces esta vecindad es equivalente al operador identidad.

Maestros	Evento 1	Evento 2	Evento 3
1	0	0	0
2	23	45	0
3	11	0	0
4	0	0	0
5	48	38	23

$\tilde{V}_2(\boldsymbol{Y})$

Maestros	Evento 1	Evento 2	Evento 3
1	48	38	23
2	0	0	0
3	11	0	0
4	23	45	0
5	0	0	0

Figura 3.12: La segunda vecindad es una simetría de reflexión respecto al centro de los renglones.

Cada renglón de la matriz $\boldsymbol{Y}$ contiene hasta tres eventos asignados a cada maestro. La vecindad $\widetilde{V}_3$ selecciona un renglón con al menos un evento asignado, toma el último evento e del renglón, de izquierda a derecha, y lo asigna a otro maestro que tenga a e entre sus preferencias (Figura 3.13).

Maestros	Evento 1	Evento 2	Evento 3	Pref.
1	48	38	53	48,38,12
2	1	37	0	37,4,3
⋮	14	62	0	16,35,86
$l-1$	4	0	0	4,23,53
l	12	0	0	54,24,8

$\tilde{V}_3(\boldsymbol{Y})$

Maestros	Evento 1	Evento 2	Evento 3	Pref.
1	48	38	0	48,38,12
2	1	37	0	37,4,3
⋮	14	62	0	16,35,86
$l-1$	4	53	0	4,23,53
l	12	0	0	54,24,8

Figura 3.13: La tercera vecindad cambia la asignación de un evento a algún maestro que lo quiera impartir.

En la tercera vecindad se promueve el movimiento de soluciones hacia regiones factibles del espacio de búsqueda, lo cual es favorable cuando operan las fases de construcción y mejoramiento de AGBL. Sin embargo, en la parte de mutación resulta conveniente perturbar las soluciones hacia el espacio no factible, de manera que haya más diversidad genética. Por ello, en $\widetilde{V}_4$ se selecciona un evento e y se asigna a algún maestro con espacio disponible (Figura 3.14).

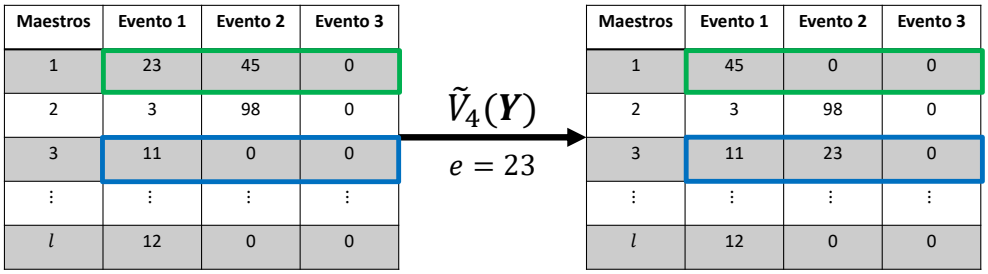


Figura 3.14: La cuarta vecindad cambia la asignación de un evento a algún maestro con espacio disponible.

3.4. Nuevo algoritmo híbrido

El segundo método utilizado en este trabajo fue el *nuevo algoritmo híbrido* (NAH), que es una matheurística que combina AGBL con un algoritmo exacto basado en el problema de agrupación. En la sección anterior se describió AGBL de manera general antes de presentar las versiones para la asignación en FCUNAM debido a que dicho algoritmo puede ser adaptado a diversos problemas de optimización. Sin embargo, NAH está construido específicamente para el problema de asignación evento-hora, por lo que en esta sección se describe directamente la adaptación para FCUNAM.
La idea principal de NAH es reducir el tamaño del problema de asignación evento-hora mediante la aglomeración de alumnos en distintos grupos. Al considerar a cada uno de estos grupos como un nuevo alumno individual, se reduce el número de alumnos que deben de ser tomados en cuenta en la optimización mediante AGBL. Los grupos se van fragmentando en subgrupos de menor tamaño hasta que se regresa a la instancia original con todos los alumnos. A continuación se describe este procedimiento de manera formal.
Sea U un conjunto cualquiera. Una partición de U es una familia τ de conjuntos $u \subseteq U$ tal que

$$\bigcup u_i = U,$$
$$u_i \cap u_j = \varnothing \qquad \forall i \neq j,$$

es decir, es una familia de subconjuntos de U que unidos resultan en U y son disjuntos por pares. Luego, el problema de agrupación consiste en encontrar una partición de un conjunto U de tal

manera que se satisfagan ciertas condiciones asociadas a un problema de optimización.
En este caso, se va a particionar el conjunto A de alumnos totales en la instancia del problema. Para ello, comenzando por el primer evento[1] e_1, se seleccionan todos los alumnos que están inscritos a él y se agrupan en un conjunto G_1, estos alumnos ya no pueden ser seleccionados para los siguientes conjuntos. Se pasa al segundo evento e_2 y se agrupan todos los alumnos que están inscritos a él en G_2, estos alumnos tampoco pueden ser seleccionados en las demás iteraciones. Se repite este procedimiento con los otros eventos hasta conseguir una partición $\tau = \{G_1, \ldots, G_k\}$ de A. El pseudocódigo de este procedimiento se muestra en la Tabla 3.4.

Require: A, *input* del problema de asignación.
Ensure: Partición τ.
1: $\tau \leftarrow \varnothing$
2: $i \leftarrow 1$
3: $j \leftarrow 1$
4: **while** $A \neq \varnothing$ **do**
5: $A(e_i) \leftarrow$ lista de alumnos que toman e_i
6: **if** $A \cap A(e_i) \neq \varnothing$ **then**
7: $G_j \leftarrow A \cap A(e_i)$
8: $\tau \leftarrow \tau \cup \{G_j\}$
9: $A \leftarrow A - A(e_i)$
10: $i \leftarrow i + 1$
11: $j \leftarrow j + 1$
12: **else**
13: $i \leftarrow i + 1$

Tabla 3.4: Algoritmo para crear la partición τ.

Luego, dado un alumno $a \in A$, se define $E(a)$ como el conjunto de eventos que toma a. Análogamente, se define

$$E(G_i) = \bigcup_{j}^{|G_i|} \{E(a_j) | a_j \in G_j\}, \tag{3.10}$$

es decir, $E(G_i)$ es el conjunto de eventos tomados por al menos un alumno en G_i. Se conoce $|E(G_i)|$ como el orden de G_i. En NAH se van a imponer cotas superiores MAX al orden de los conjuntos G_i, cuando existe un conjunto G_j cuyo orden supera dicha cota, se particiona en una familia de subgrupos $G_1^j, \ldots, G_l^j$ utilizando el mismo procedimiento con el que se generó τ, pero tomando en cuenta sólo aquellos eventos que no toman todos los alumnos en G_j (si se tomara en cuenta

[1]En este caso, los eventos se toman en el orden en que fueron introducidos a la base de datos

un evento con esa característica, el subgrupo resultante seria de nuevo G_j). El pseudocódigo para generar estos subgrupos se muestra a continuación:

Require: G_j, *input* del problema de asignación.
Ensure: Subgrupos $G_1^j, \ldots, G_l^j$.
1: $i \leftarrow 1$
2: $k \leftarrow 1$
3: **while** $G_j \neq \varnothing$ **do**
4: $e_i \leftarrow$ i-ésimo evento en $E(G_j)$
5: $G_j(e_i) \leftarrow$ lista de alumnos en G_j que toman e_i
6: **if** $G_j \neq G_j(e_i)$ **and** $G_j \cap G_j(e_i) \neq \varnothing$ **then**
7: $G_k^j \leftarrow G_j \cap G_j(e_i)$
8: $G_j \leftarrow G_j - G_j(e_i)$
9: $i \leftarrow i + 1$
10: $k \leftarrow k + 1$
11: **else**
12: $i \leftarrow i + 1$

Tabla 3.5: Algoritmo para reducir el orden de un grupo.

Una vez particionado G_j, se calcula el orden de cada uno de los subgrupos G_k^j. Si existe algún subgrupo cuyo orden sigue superando MAX, se repite el procedimiento de fragmentación en el subgrupo. Se sigue de esta manera hasta conseguir una partición τ' de A en donde el orden de todos los conjuntos esté debajo de la cota impuesta. Cabe mencionar que, mientras no existe una cota superior para el valor de MAX, la cota inferior c_{MAX} es el número máximo de eventos que toman los alumnos. Cuando MAX toma ese valor, la partición serán conjuntos unitarios, uno por cada alumno[2].
El diagrama de flujo de NAH se muestra en la Figura 3.15, se empieza con una cota superior MAX y se crea una partición $\tau' = \{G_1, ..., G_s\}$ que cumpla con la misma. Después, se crea una instancia del problema de asignación evento-hora en FCUNAM considerando que cada G_i es un sólo *alumno-grupal* que está inscrito a todos los eventos en $E(G_i)$. Se aplica AGBL a esta nueva instancia de alumnos-grupales y, al terminar, se hace $MAX = \max(MAX - 1, c_{\text{MAX}})$, se genera una partición con esta nueva cota y se vuelve a aplicar AGBL. Este procedimiento se repite hasta que se cumpla un criterio de paro, que en este trabajo se tomó como el tiempo de ejecución del programa. Cuando MAX ha llegado a su cota inferior, la instancia regresa a ser la original con todos los alumnos y NAH converge a $AGBL$.

[2]A menos que hayan alumnos que tomen exactamente los mismos eventos, en tal caso, habrán grupos no unitarios.

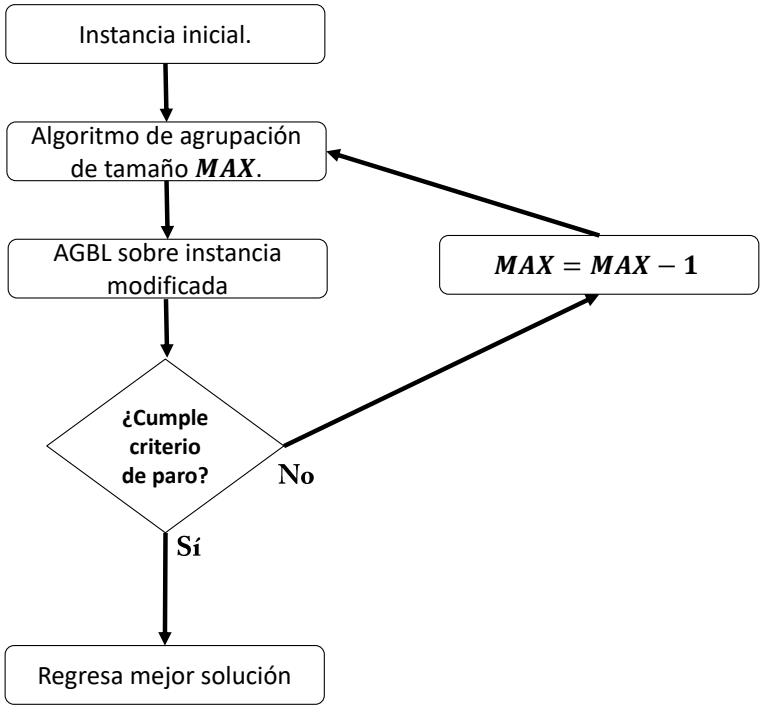


Figura 3.15: Diagrama de flujo de NAH.

Como la generación de grupos en NAH se hace a partir del conjunto de alumnos, y en la asignación evento-maestro ya no se toma en cuenta dicho conjunto, este segundo método sólo se implemento en la asignación evento-hora.

Capítulo 4

Resultados y discusión

Se utilizó el modelo propuesto en dos circunstancias distintas: para alumnos de primer ingreso en un semestre impar y para alumnos de segundo semestre en un semestre par. Primero, se realizó la recopilación y construcción de datos estadísticos como se describe en el capítulo 2 para la creación de las dos instancias artificiales. Después, se aplicaron AGBL y NAH en las asignaciones evento-hora y, finalmente, se utilizó AGBL para la optimización de las fases evento-maestro.
Todos los algoritmos fueron programados en Python[1] y ejecutados en una computadora con procesador de cuatro núcleos de 3.1 GHz cada uno.

4.1. Semestre impar

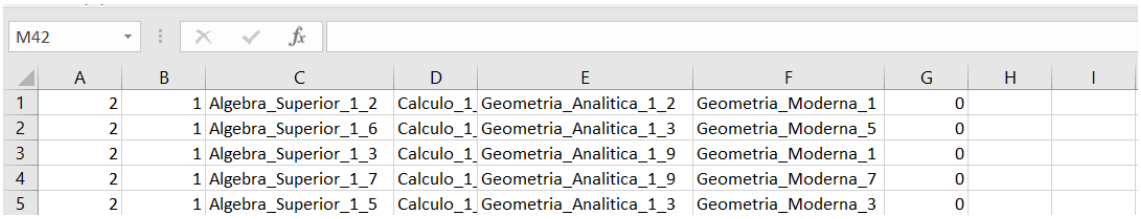
4.1.1. Alumnos artificiales

De la base de datos obtenida para el semestre impar, se calculó un número esperado de 271 matemáticos, 264 físicos y 277 actuarios que ingresarían a primer semestre. Esto es un total de 812 alumnos. Combinado con el número de eventos (75), salones (118) y horas por día (9, contando de las 7:00 A.M. a las 4:00 P.M.), se obtuvo una instancia extremadamente compleja, de acuerdo con la Tabla 2.1. Con el objetivo de tener un ejemplo en donde se obtuvieran horarios que representen soluciones reales para la FCUNAM pero que pudiera ser resuelto en un tiempo accesible[2], se optó por modificar la instancia a una con 120 alumnos, 40 por cada carrera. Por otro lado, se mantuvo el número de eventos en 75 y sólo se modificó el número de alumnos inscritos en cada uno para adecuarlo a los 120 alumnos (Figura 4.1). Respecto a los salones, se consideraron sólo los primeros pisos de los edificios O, P, Yelizcalli y el Nuevo Edificio, además de que en los otros edificios

[1] El código se encuentra en el segundo apéndice

[2] Al intentar resolver la instancia con 812 alumnos, AGBL tardó aproximadamente una semana en pasar de una generación a otra.

(Departamento de Matemáticas y Tlahuizcalpan) se omitieron la mitad de los salones disponibles, excepto para los laboratorios de física y salones de cómputo que se utilizan para clases de actuaría. Esto redujo el número de salones a 69. Por último, la capacidad de cada salón se redujo en 60 % para ajustarla a la instancia con menos alumnos.
A partir de estos datos se creó la instancia de alumnos artificiales mediante los procedimientos descritos en el capítulo 2.



M42

	A	B	C	D	E	F	G	H	I
1	2	1	Algebra_Superior_1_2	Calculo_1_	Geometria_Analitica_1_2	Geometria_Moderna_1	0		
2	2	1	Algebra_Superior_1_6	Calculo_1_	Geometria_Analitica_1_3	Geometria_Moderna_5	0		
3	2	1	Algebra_Superior_1_3	Calculo_1_	Geometria_Analitica_1_9	Geometria_Moderna_1	0		
4	2	1	Algebra_Superior_1_7	Calculo_1_	Geometria_Analitica_1_9	Geometria_Moderna_7	0		
5	2	1	Algebra_Superior_1_5	Calculo_1_	Geometria_Analitica_1_3	Geometria_Moderna_3	0		

Figura 4.1: Muestra del conjunto de alumnos artificiales en el semestre impar, cada renglón corresponde a un alumno, y muestra las materias que tiene inscritas.

4.1.2. Asignación evento-hora

En esta asignación la función objetivo es

$$f(\boldsymbol{X}) = \lambda F(\boldsymbol{X}) + S(\boldsymbol{X}), \tag{4.1}$$

en donde, como se mencionó anteriormente, λ es una constante de penalización. En el peor de los casos de la restricción suave, un alumno tendrá diario una clase de 7:00 a 8:00 A.M. y otra de 3:00 a 4:00 P.M., lo que suma un total de 7 horas libres al día y 35 a la semana. Si esto pasara para los 120 alumnos, se tendrían 4200 horas libres en total. Esta es una cota superior del valor máximo de $S(\boldsymbol{X})$. Así, para asegurar que ninguna solución no factible sea mejor que una factible, se fijó

$$\lambda = 4200. \tag{4.2}$$

Luego, en AGBL se impuso como criterio de paro el tiempo de ejecución, que se fijó en 800 minutos, por lo que no se tomó en cuenta un número máximo de generaciones. Los otros parámetros utilizados se muestran en la siguiente tabla:

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
t	Tiempo de paro (min)	800

Tabla 4.1: Parámetros utilizados en AGBL.

Por otro lado, el método NAH tiene a AGBL como rutina en cada iteración, por lo que en este caso sí se especificó un número máximo de 3000 generaciones para dicha subrutina, mientras que el criterio de paro del algoritmo completo fue el tiempo de ejecución, que se fijó en 800 minutos. La lista con los otros parámetros se muestra a continuación:

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
n_{gen}	Número de generaciones	3000
MAX	Cota superior del orden de cada alumno-grupal	24
t	Tiempo de paro (min)	800

Tabla 4.2: Parámetros utilizados en NAH.

En la Figura 4.2 se pueden ver dos distribuciones del número de alumnos inscritos a i eventos, con $i \in [1, 24] \cap \mathbb{Z}$. La distribución azul es respecto al conjunto original de alumnos, hay 80 que toman 5 eventos (son los actuarios y físicos) y 40 que toman 4 eventos (matemáticos). Por otro lado, la distribución naranja corresponde al primer conjunto de alumnos-grupales. Mediante el algoritmo de agrupación se redujo el número de alumnos en la instancia de 120 a 26. Además, como cada alumno-grupal toma los eventos de todos alumnos reales que lo constituyen, en la segunda distribución hay alumnos que inscriben hasta 24 eventos. Se tomó $MAX = 24$ para que NAH empezara con esta partición inicial de A.

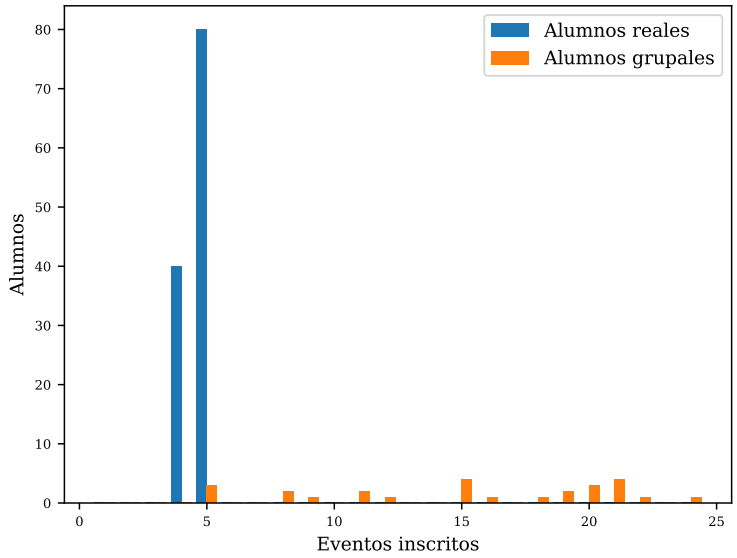


Figura 4.2: Distribución del número de alumnos que inscriben i eventos. Se muestran la distribución original (azul) y la del primer conjunto de alumnos-grupales (naranja).

Primero, se hicieron comparaciones de las fases de construcción y mejoramiento para un individuo de la primera generación en ambos métodos.

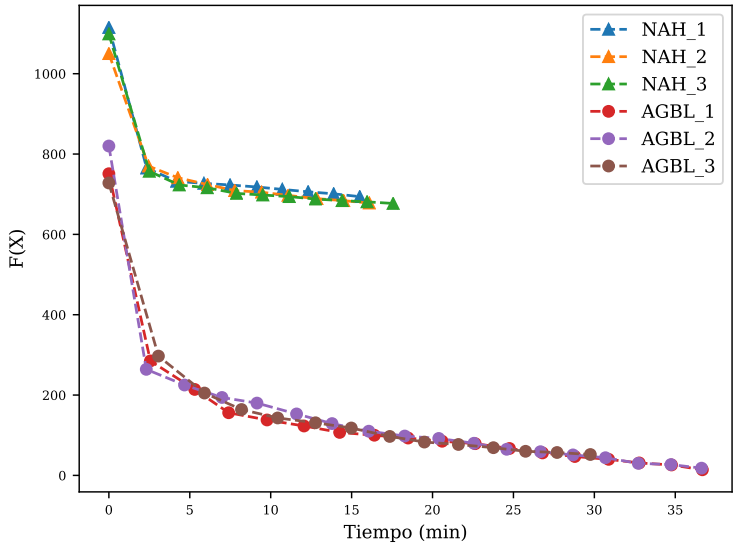


Figura 4.3: Gráficas de NAH y AGBL durante la fase de construcción de la búsqueda local.

En la Figura 4.3 se pueden ver las gráficas de tres corridas en NAH y tres en AGBL para la fase de construcción. En estas gráficas no se mide la función objetivo completa $f(\boldsymbol{X})$, sino sólo $F(\boldsymbol{X})$, que es la parte asociada a las restricciones fuertes. Se puede apreciar que las soluciones dadas por $AGBL$ son mejores que las de NAH en todas las ejecuciones. Esto era de esperarse pues, en la primera iteración de NAH, hay alumnos que toman de 15 a 24 eventos, y en el lapso matutino de 9 horas es imposible generar un horario en donde no hayan empalmes en los eventos de dichos alumnos (los empalmes son una restricción fuerte).
Inmediatamente después de la fase de construcción sigue la fase de mejoramiento, en la Figura 4.4 se pueden ver las continuaciones de las 6 gráficas anteriores al pasar a esta segunda fase. Como ahora se miden también las restricciones suaves, el eje y corresponde a los valores de $f(\boldsymbol{X})$.

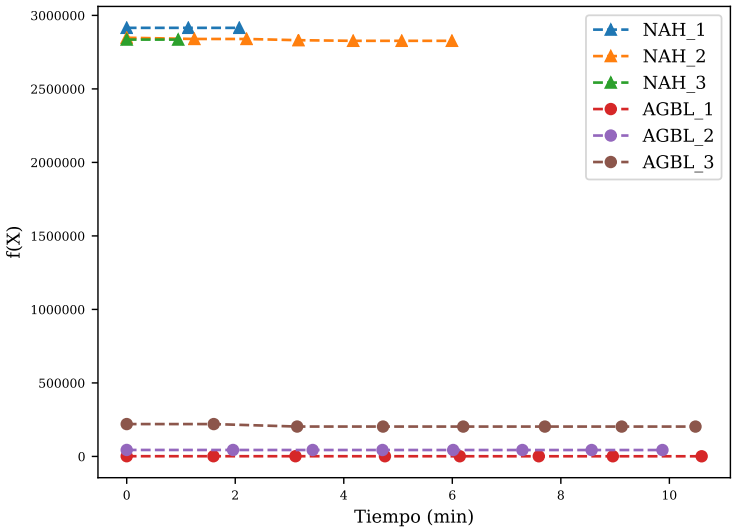


Figura 4.4: Gráficas de NAH y AGBL durante la fase de mejoramiento de la búsqueda local.

En general, en la fase de mejoramiento el estancamiento en un óptimo local ocurre de manera más rápida que en la fase de construcción. De las seis ejecuciones se puede estimar el tiempo de ejecución de la búsqueda local variable en ambos métodos, como se muestra en la Tabla 4.3.

Fase	Tiempo promedio en NAH (min)	Tiempo promedio en AGBL (min)
Construcción	16	35
Mejoramiento	3	10
Búsqueda completa	19	45

Tabla 4.3: Tiempos de ejecución de la búsqueda local variable.

Como se mencionó anteriormente, mientras NAH avanza y se va reduciendo la cota MAX, los alumnos-grupales se irán fragmentando en subgrupos hasta llegar a la instancia original y a AGBL. Por esta razón, las diferencias de tiempo de la búsqueda local en ambos métodos se irá haciendo cada vez menos significativa.
Sin embargo, completar este proceso tarda alrededor de 30 minutos por individuo, de acuerdo con la tabla anterior. Considerando que hay 8 soluciones en la población, pasar de una generación a otra tardaría aproximadamente 8×30 min $= 240$ min $= 4$ horas al utilizar cómputo en serie. Con la finalidad de reducir significativamente este tiempo de ejecución, se implementó cómputo en paralelo: la búsqueda local sobre los individuos de la población se divide equitativamente entre los procesadores disponibles (Figuras 4.5 y 4.6).

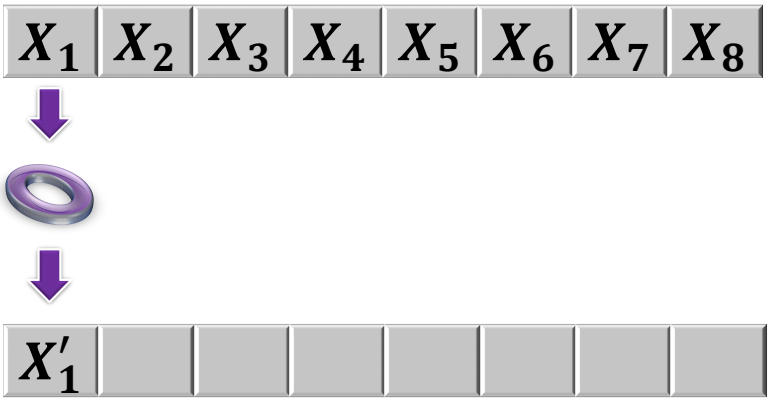


Figura 4.5: En el cómputo en serie, un procesador va haciendo la búsqueda local sobre los individuos, de uno en uno.

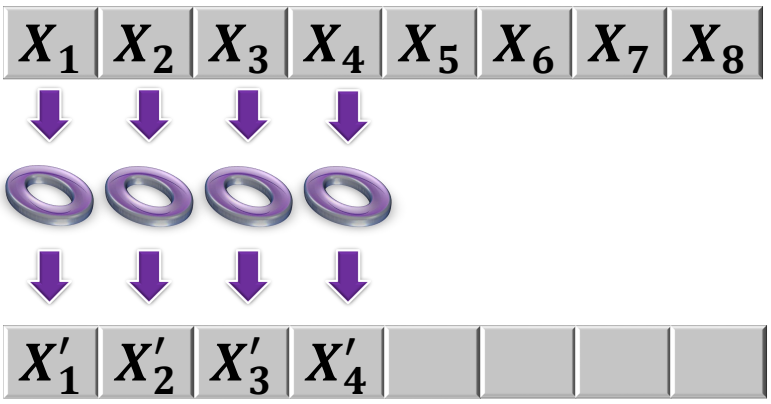


Figura 4.6: En el cómputo en paralelo, se hace la búsqueda local sobre varios individuos al mismo tiempo. Cada procesador se encarga de uno de ellos.

La computadora que se utilizó cuenta con 4 núcleos, de manera que cada uno realizó la búsqueda local sobre dos integrantes de la población. Así, el tiempo aproximado de este proceso se redujo de 4 horas a 2×30 min $= 60$ min $= 1$ hora. Después de implementar el cómputo en paralelo, se hicieron tres corridas completas de cada algoritmo.

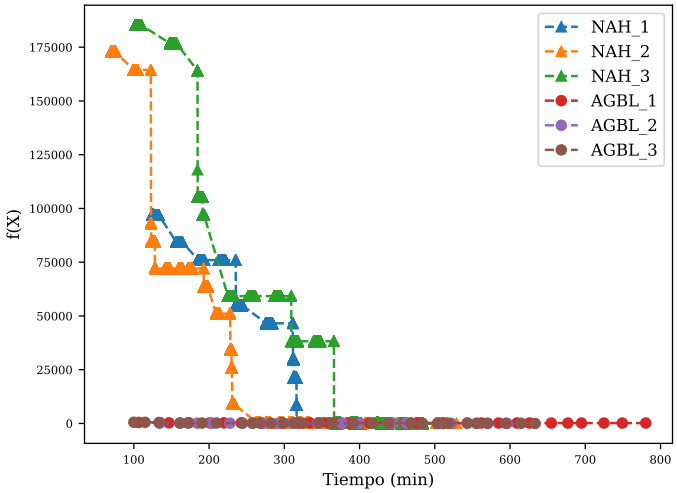


Figura 4.7: Evolución de las soluciones en NAH y AGBL en el semestre impar.

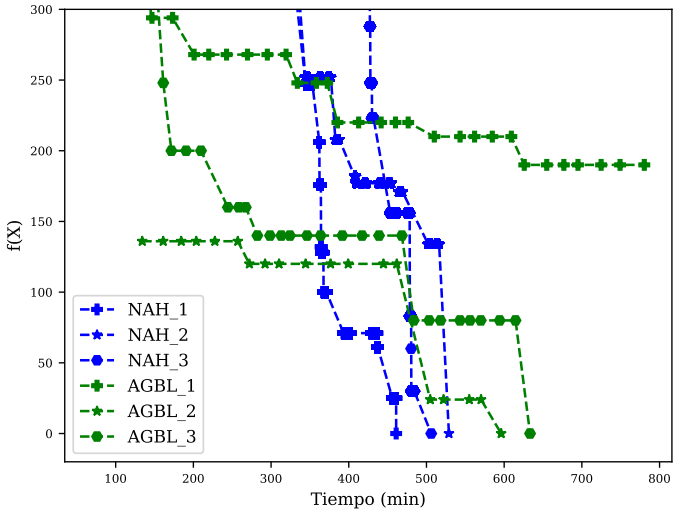


Figura 4.8: Evolución de las soluciones en NAH y AGBL en el semestre impar, *zoom* sobre el eje y.

Las primeras soluciones dadas por AGBL están mucho más cerca del óptimo buscado ($f = 0$) que las de NAH. Sin embargo, conforme pasa el tiempo y los alumnos-grupales se van fragmentando, el algoritmo de NAH supera a AGBL, como se aprecia en la Figura 4.8, que contiene las mismas gráficas pero con un *zoom* sobre el eje y, además de que muestra a todas las corridas de NAH de un sólo color y las de AGBL de otro para evidenciar que, en todos los casos, NAH llegó primero a una $\boldsymbol{X}_{opt}$. Aunque las gráficas anteriores son útiles para conocer la calidad de las soluciones encontradas en términos de la función objetivo, es necesario decodificar la información de las listas $\boldsymbol{X}$ asociadas a estas soluciones para poder visualizar y manipular los horarios que resultan de las mismas. Una opción es convertir cada $\boldsymbol{X}$ en un archivo que contiene los horarios de cada evento (Figura 4.9), otra es utilizar horarios de colores (Figura 4.10).

M27

	A	B	C	D	E	F	G	H	I	J	K	L
1		**Materia**	**Inicia**	**Termina**	**Lu**	**Ma**	**Mi**	**Ju**	**Vi**	**Salon**		
2	**0**	AlgSup 1	9	10	1	1	1	1	1	4		
3	**1**	AlgSup 1	9	10	1	1	1	1	1	30		
4	**2**	AlgSup 1	12	13	1	1	1	1	1	29		

Figura 4.9: Ejemplo de la traducción de una solución $\boldsymbol{X}$ a una base de datos de horarios. En este caso se trata de una solución inicial aleatoria $\boldsymbol{X}_0$.

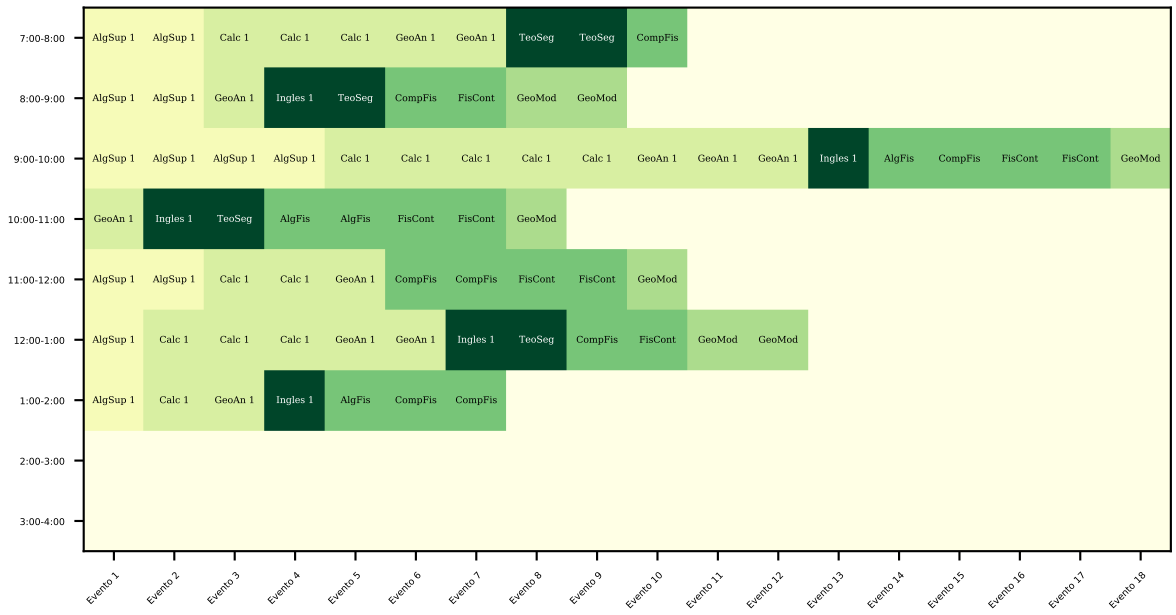


Figura 4.10: Ejemplo de la traducción de una solución $\boldsymbol{X}$ a un horario de colores. Éste es el asociado a la $\boldsymbol{X}_0$ de la Figura anterior.

Siguiendo este mismo formato, se consiguieron los siguientes horarios de color asociados a tres de las soluciones óptimas ($f(\boldsymbol{X}) = 0$) encontradas:

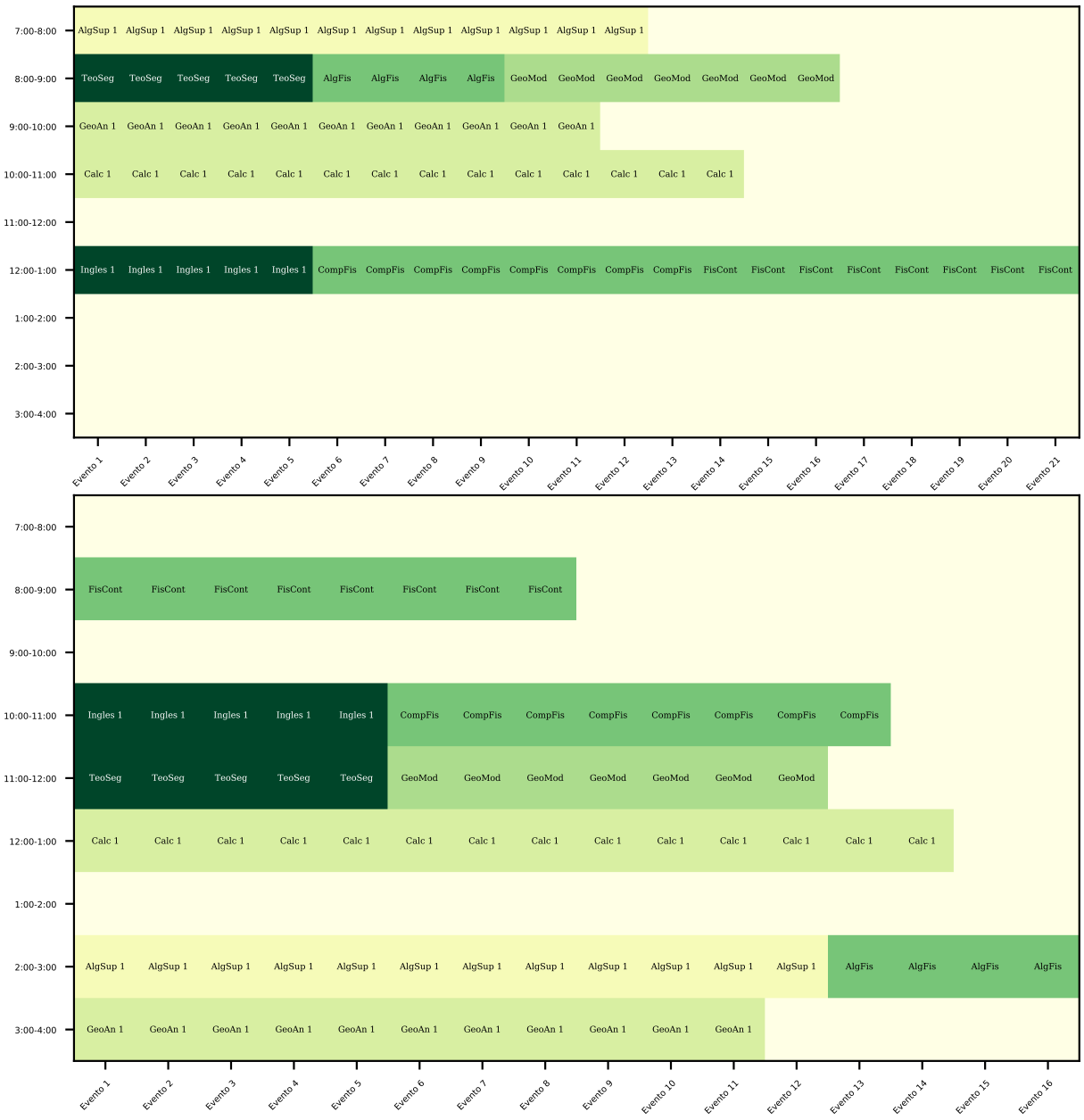


Figura 4.11: Horarios de color asociados a tres soluciones óptimas.

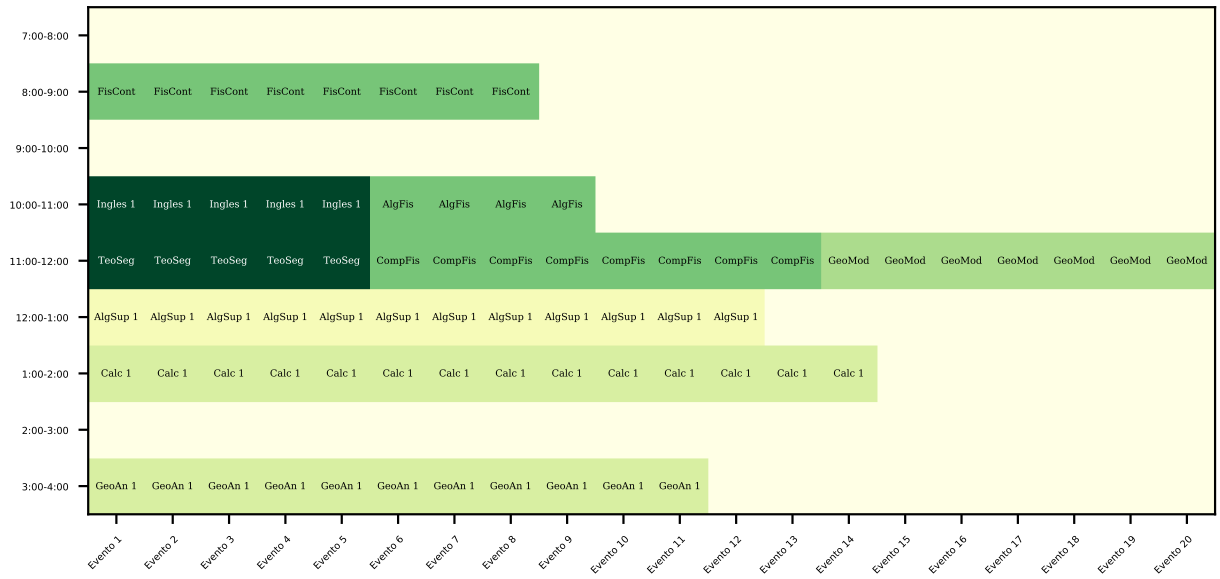


Figura 4.11: Horarios de color asociados a tres soluciones óptimas (continuación).

En los tres casos se presenta un patrón de aglomeración en el que todos los grupos disponibles de un tipo de evento se imparten a la misma hora, en FCUNAM este esquema de asignación se conoce como *horarios de bandas*. Cada banda está asociada a un tipo de evento: de físicos, matemáticos, actuarios y los de tronco común. Con lo cual, en los horarios óptimos, todos los alumnos toman los eventos de tronco común a la misma hora, y en las otras horas disponibles se distribuyen en los eventos específicos de sus licenciaturas. La aparición de los esquemas de bandas en las soluciones encontradas (además del hecho de que dichos esquemas han sido implementados anteriormente en FCUNAM) demuestra la utilidad de este modelo que, aunque bastante simplificado, es capaz de sugerir patrones para la creación de horarios óptimos. Respecto a los días de impartición, no se encontraron patrones sobresalientes, especialmente porque casi todos los eventos en esta instancia se imparten los cinco días de la semana.

De igual manera, en la asignación de salones no se encontraron comportamientos específicos en los horarios óptimos. Esto fue debido principalmente a que en esta instancia reducida de alumnos todos los salones son ideales para cada evento (salvo los laboratorios de física y salones de cómputo, que fueron asignados satisfactoriamente en todas las soluciones). Además, esto se esperaba desde la recopilación de datos de la página web de FCUNAM, pues se vio que, salvo *casos de aglomeración*, se cuenta con los edificios y capacidades necesarias para llevar a cabo la impartición de eventos. Los casos de aglomeración observados consisten en que, cuando hay al menos dos grupos A y B que corresponden al mismo evento, existe una tendencia en la que el grupo A está altamente saturado (en un caso el número de alumnos inscritos llegó a 347) mientras que los grupos restantes tienen pocos alumnos inscritos (hubieron casos con menos de 5 alumnos). En esta instancia artificial se

descartaron los fenómenos de aglomeración en todo momento (obtención de promedios, construcción de alumnos, etc.), por lo que el asignar un evento a salones grandes (e.g. las Aulas Magnas) o a salones pequeños tuvo un impacto nulo en la función objetivo.

4.1.3. Asignación evento-maestro

Ya que en el primer semestre de la licenciatura de Física no se llevan laboratorios, la construcción de la base artificial de maestros en este caso se llevó a cabo únicamente con los algoritmos descritos en el capítulo 2, sin tener que agregar manualmente maestros de laboratorio. Como se mencionó anteriormente, hubieron un total de 75 eventos. Luego, con la finalidad de tener un espacio de búsqueda diverso, se creó una base artificial con un total de 150 maestros. Al asignar aleatoriamente sus preferencias y el rubro *experto*, se obtuvo la distribución que se muestra en la Figura 4.12, en donde se puede observar que, para todo evento, hay al menos cuatro maestros que lo quieren impartir.

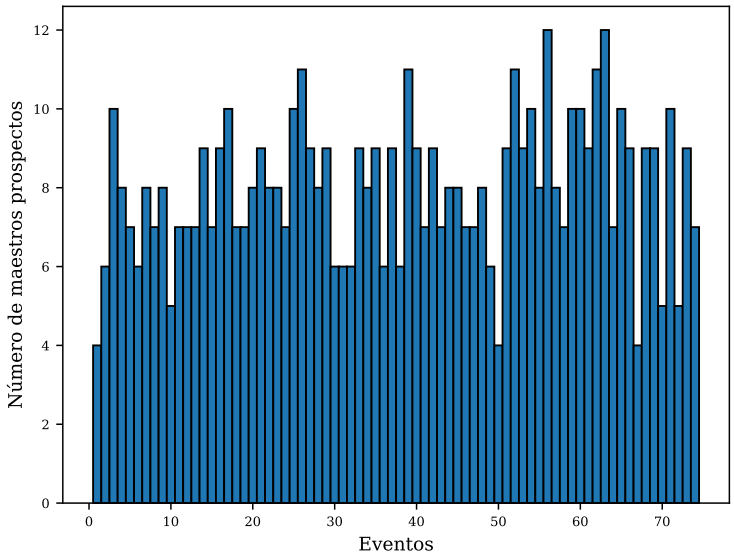


Figura 4.12: Distribución del número de maestros que quieren impartir cada evento.

Con estos datos artificiales se pudo proceder a la asignación evento-maestro. La función objetivo es

$$\widetilde{f}(\boldsymbol{Y}) = -\mu\widetilde{F}(\boldsymbol{Y}) + \widetilde{S}(\boldsymbol{Y}), \tag{4.3}$$

en donde μ es la constante de penalización. Para que ninguna solución no factible sea mejor que una factible, esta constante tiene que contrarrestar (por el signo de menos) el mejor caso posible de

$\widetilde{S}$. Luego, suponiendo que a todos los maestros se les asignaran sus tres primeros eventos preferidos y que todos tuvieran 30 años de experiencia y una evaluación de 10, se tendría contribución total de $150 \times 30 = 4500$ a las restricciones suaves. Esta es una cota superior del valor máximo de $\widetilde{S}$, por lo que se tomó

$$\mu = 4500. \tag{4.4}$$

En esta fase sólo se pudo utilizar AGBL. De manera análoga al caso evento-hora, se impuso como criterio de paro el tiempo de ejecución, que se fijó en 360 minutos. Todos los demás parámetros fueron idénticos a los de la fase anterior (Tabla 4.4).

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
t	Tiempo de paro (min)	360

Tabla 4.4: Parámetros utilizados en AGBL para la asignación evento-maestro.

En la asignación pasada se hicieron tres corridas de cada algoritmo porque, como se buscaba compararlos entre ellos, era necesario tener varias mediciones de cada uno con la finalidad de minimizar el *ruido estadístico* en los datos que resulta de la naturaleza estocástica de cada metodología. Como aquí sólo se utilizó AGBL, bastó con una ejecución del algoritmo.
Por otro lado, la función de restricciones suaves en esta asignación es una combinación lineal de los tres rubros que se pueden tomar en cuenta en la optimización: la satisfacción de los maestros (α), su evaluación (β) y su experiencia (γ). Así, aunque no se pudieron comparar los dos métodos, sí se pudieron comparar los tipos de soluciones que se obtienen al variar estos parámetros. Específicamente, se analizaron los 3 casos en donde se da prioridad a sólo uno de los rubros, además de un cuarto caso de equilibrio en donde todos se consideran igual de importantes (Tabla 4.5).

Caso	α	β	γ	Rubro priorizado
1	1	0	0	Satisfacción
2	0	1	0	Evaluación
3	0	0	1	Experiencia
4	1/3	1/3	1/3	Equilibrio

Tabla 4.5: Casos que se analizaron en la asignación evento-maestro.

Por los resultados obtenidos en la fase anterior, en esta asignación también se implementó la técnica de cómputo en paralelo. Se hizo una corrida de AGBL por cada caso.

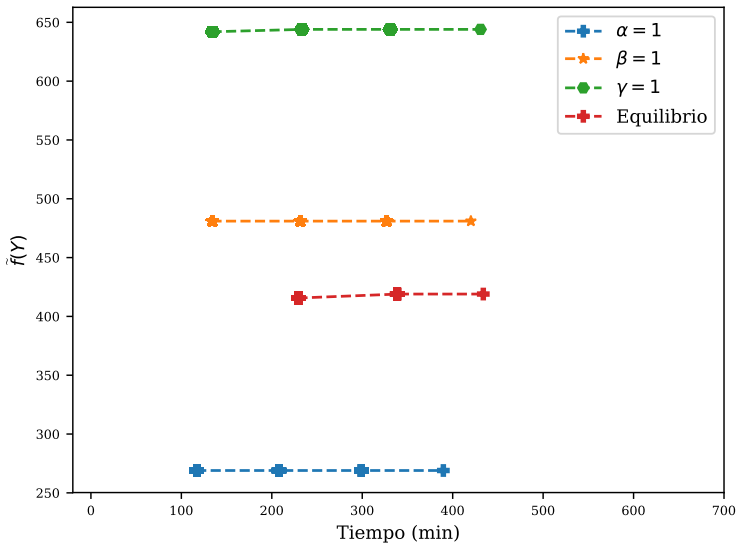


Figura 4.13: Evolución de las soluciones en AGBL, asignación evento-maestro.

Aunque pareciera que hay pocos puntos por gráfica, de hecho se trata de conjuntos con 1000 puntos y un punto final. Cada conjunto de puntos corresponde a AGBL pasando por 1000 generaciones (Figura 4.14) antes de aplicar la búsqueda local variable.

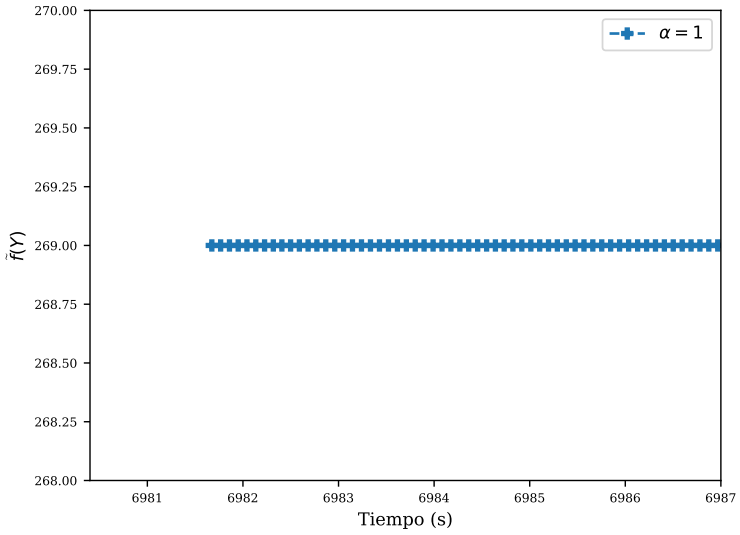


Figura 4.14: Al hacer un *zoom* en la gráfica anterior, se pueden apreciar los clústers de puntos, que corresponden a las 1000 generaciones de AGBL antes de aplicar la búsqueda local variable.

Las dos gráficas anteriores sólo ilustran el comportamiento de AGBL al intentar maximizar cada caso, no son una comparación de los óptimos obtenidos, pues, el que algunas gráficas alcancen valores mayores de $\widetilde{f}$, no significa que sus poblaciones asociadas hayan sido mejores que en los otros casos. Esto se debe a que, al variar los parámetros de $\widetilde{S}$, se tienen óptimos distintos en $\widetilde{f}$. Además, a diferencia de la asignación evento-hora, en donde se sabía que la mejor solución posible era $f = 0$, aquí no se puede saber si las soluciones obtenidas en cada caso coincidieron con el mejor valor posible o si son óptimos locales, ya que esto depende tanto de los parámetros de $\widetilde{S}$ como de los maestros disponibles (e.g. en una base de maestros con sólo 5 años de experiencia se tendrá un valor máximo posible distinto que con otra base de maestros con 50 años de experiencia).
En cambio, sí se pueden comparar las cuatro soluciones en términos de qué tanto se maximizaron los rubros de satisfacción, evaluación y experiencia en cada caso. Dada una solución $\boldsymbol{Y}$, se dice que un maestro m está satisfecho si se le asignaron su primera o segunda opción de evento. Luego, una manera de medir la *satisfacción* asociada a $\boldsymbol{Y}$ es calculando el porcentaje de maestros satisfechos respecto al número total de maestros que tienen asignado al menos un evento. En la Figura 4.15 se muestra este porcentaje para cada caso.

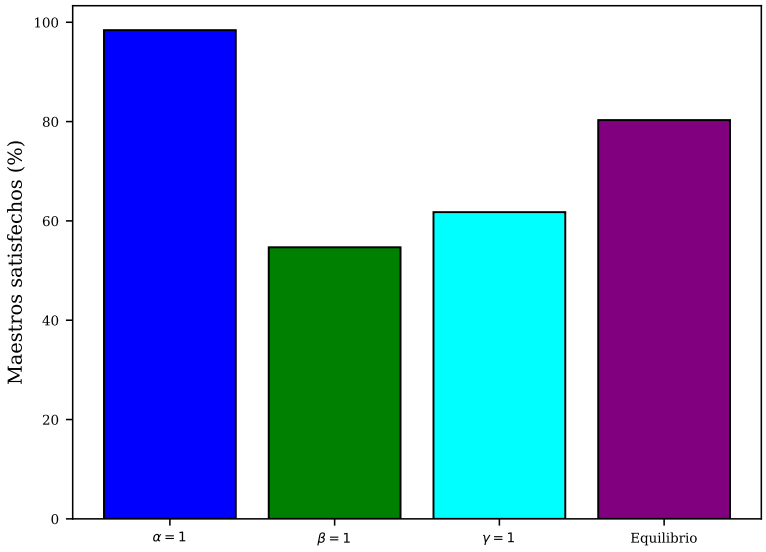


Figura 4.15: Porcentaje de maestros a los que se les asignó su primera o segunda opción de evento.

Los resultados exactos se muestran en la Tabla 4.6. Como era de esperarse, en el caso $\alpha = 1$ se alcanzó casi un 100 % de maestros satisfechos, mientras que en $\beta = 1$ y $\gamma = 1$ el porcentaje se mantuvo debajo del 63 %. Por otro lado, en la solución de equilibrio se alcanzó el 80.3 % de maestros satisfechos, lo que tiene sentido pues, en este caso, se le dio parte de la prioridad a la satisfacción.

Caso	Maestros satisfechos (%)
$\alpha = 1$	98.4
$\beta = 1$	54.5
$\gamma = 1$	62.6
Equilibrio	80.3

Tabla 4.6: Porcentaje de maestros satisfechos en cada solución.

Para medir la *calificación* asociada a una solución, se calculó el promedio de la calificación de todos los maestros con al menos un evento asignado (Tabla 4.7). Adicionalmente, en la Figura 4.16 se pueden observar los diagramas de caja de cada solución[3].

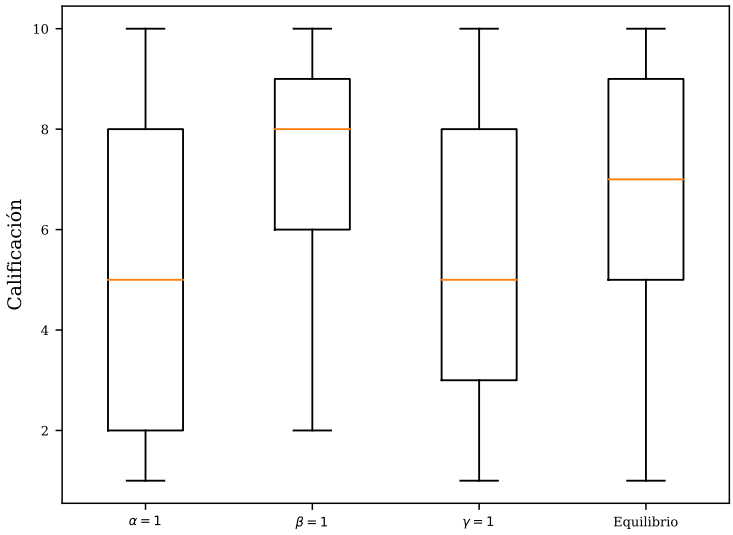


Figura 4.16: Diagramas de caja de calificaciones de maestros con al menos un evento asignado.

Nuevamente, los resultados coincidieron con lo que se esperaba. Aunque en todas las soluciones hay maestros con calificación de 10, de acuerdo con los diagramas de cajas, en el caso $\beta = 1$ y en el de equilibrio las calificaciones de los maestros se distribuyen principalmente alrededor de valores más grandes que 5. Esto se confirma con la tabla de promedios, en donde la calificación más alta la tuvo $\beta = 1$, seguida por la solución de equilibrio.

[3]En el primer apéndice se ofrece una breve definición de los diagramas de caja.

Caso	Calificación promedio
$\alpha = 1$	4.8
$\beta = 1$	7.5
$\gamma = 1$	5.2
Equilibrio	6.7

Tabla 4.7: Promedio de calificación en cada solución.

Análogamente, para la *experiencia* asociada a una solución se calculó el promedio de años de experiencia de todos los maestros con al menos un evento asignado (Tabla 4.8). Además de esto, se volvieron a graficar los diagramas de cajas (Figura 4.17).

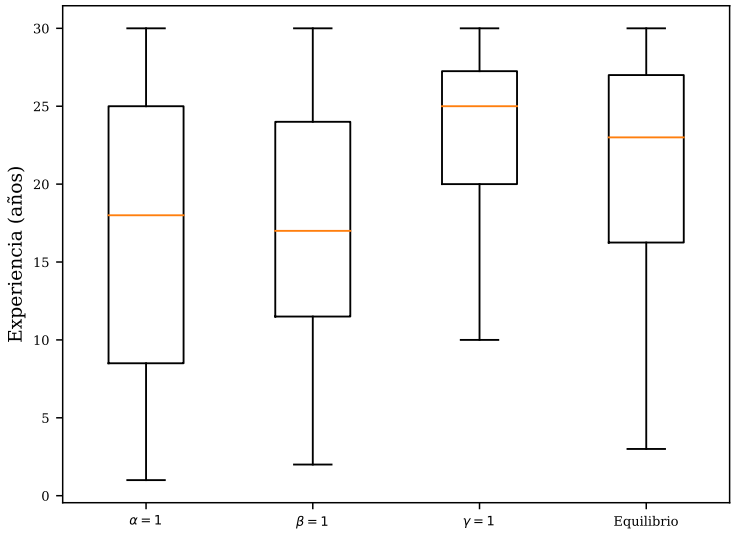


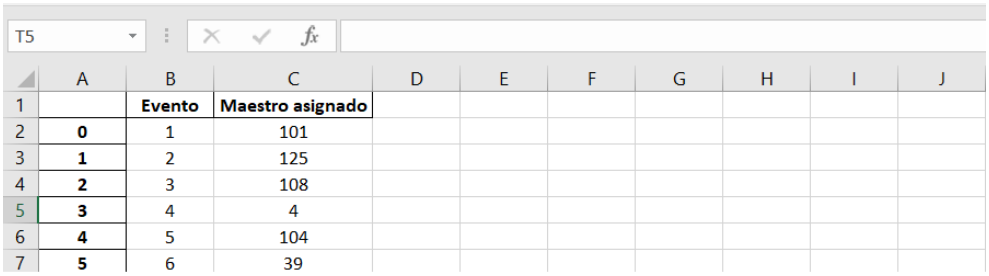
Figura 4.17: Diagramas de caja de años de experiencia de maestros con al menos un evento asignado.

Al igual que en el caso anterior, los diagramas de caja coinciden con los resultados que se esperaban. Tanto en la solución de equilibrio como en $\gamma = 1$ los años de experiencia de los maestros con al menos una materia asignada se distribuyen principalmente alrededor de 24 años, mientras que en $\alpha = 1$ y $\beta = 1$ lo hacen alrededor de 17 años. Además, $\gamma = 1$ tuvo el promedio más alto de años de experiencia, seguido por el promedio de la solución de equilibrio.

Caso	Experiencia promedio (años)
$\alpha = 1$	16.4
$\beta = 1$	16.8
$\gamma = 1$	23.4
Equilibrio	21.5

Tabla 4.8: Promedio de años de experiencia en cada solución.

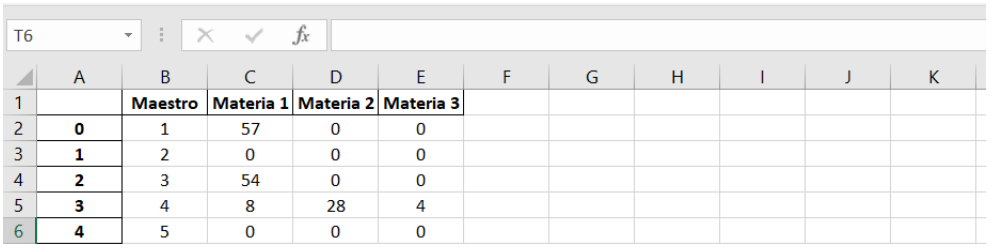
Como en la asignación pasada, resulta útil traducir la información de una $\boldsymbol{Y}$ a una base de datos usual que pueda ser manipulada de manera sencilla. Para ello, se convierte $\boldsymbol{Y}$ en un archivo (Figura 4.18) que contiene, para cada evento, el maestro que lo impartirá.



	A	B	C
1		Evento	Maestro asignado
2	0	1	101
3	1	2	125
4	2	3	108
5	3	4	4
6	4	5	104
7	5	6	39

Figura 4.18: Ejemplo de la traducción de una solución $\boldsymbol{Y}$ a una base de datos de la asignación por evento. En este caso se trata del óptimo para $\alpha = 1$.

Otra posible representación consiste en una base de datos que contiene, para cada maestro, los eventos que le fueron asignados (Figura 4.19). En este caso, un renglón con sólo 0's significa que al maestro correspondiente no se le asignó ningún evento.



	A	B	C	D	E
1		Maestro	Materia 1	Materia 2	Materia 3
2	0	1	57	0	0
3	1	2	0	0	0
4	2	3	54	0	0
5	3	4	8	28	4
6	4	5	0	0	0

Figura 4.19: Ejemplo de la traducción de una solución $\boldsymbol{Y}$ a una base de datos de las asignaciones de cada maestro. En este caso se trata del óptimo para $\alpha = 1$.

Como ya se analizaron las diferencias entre soluciones a través de las gráficas de satisfacción, experiencia y calificación, y como en este caso no se toman en cuenta diferentes horas, días, ni

licenciaturas, no fue necesario recurrir a la representación de colores, como en la fase pasada. Finalmente, combinando los óptimos $\boldsymbol{X}_{opt}$ de la fase anterior con los $\boldsymbol{Y}_{opt}$ de la asignación evento-maestro se construyen las soluciones completas $(\boldsymbol{X}_{opt}, \boldsymbol{Y}_{opt})$ para la asignación en FCUNAM durante el semestre impar.

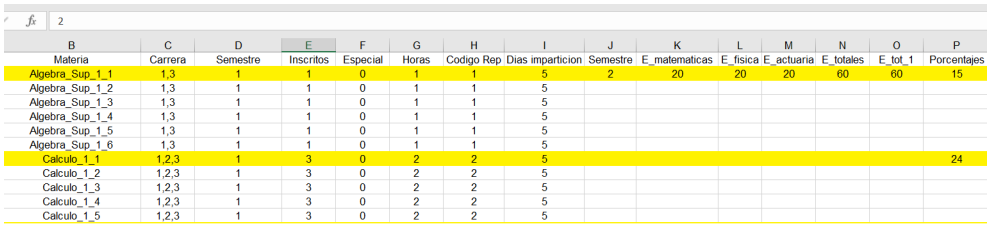
4.2. Semestre par

4.2.1. Alumnos artificiales

En este caso, se consideraron los dos primeros semestres debido a que los alumnos que pasaron a segundo semestre pueden estar volviendo a cursar materias que reprobaron en el ciclo anterior. De la base de datos se estimaron 352 matemáticos, 339 físicos y 362 actuarios, sumando un total de 1053 alumnos. Como en el semestre impar, esta instancia no puede ser resuelta en un tiempo accesible. Aquí también se modificó, de tal manera que hubieran 60 alumnos en total, 20 por licenciatura. Se escogió este número más pequeño porque, al considerar dos semestres al mismo tiempo, el número de eventos subió a un total de 117. Este número de eventos se mantuvo, y se modificaron los alumnos esperados por evento así como los datos de salones como en el caso de semestre impar.
En el caso anterior, de alumnos de primer ingreso, se pudieron saber con exactitud los eventos que llevó cada alumno pues, al entrar a la FCUNAM, sólo se toman eventos obligatorios de primer semestre. Así, al modificar la instancia con un menor número de alumnos, sólo se modificaron los alumnos esperados por evento para sumar los 120 de esa instancia. Sin embargo, para el semestre par no existe esta información pues, en general, se pueden dar tres posibilidades:

- Alumnos que inscriben sólo los eventos obligatorios/optativos de segundo semestre.
- Alumnos que inscriben sólo eventos que *recursan* de primer semestre.
- Alumnos que inscriben tanto eventos de segundo semestre como de *recursamiento*.

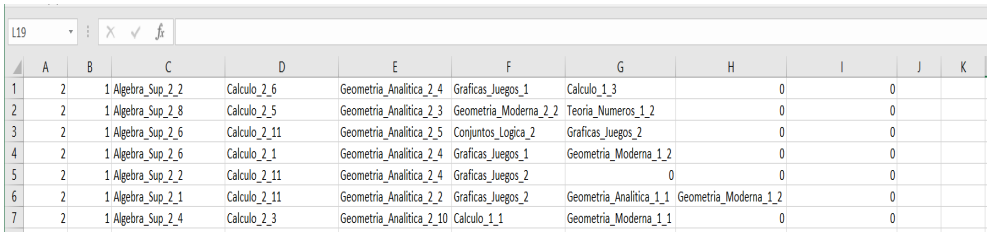
Por esta razón, la modificación de alumnos esperados por evento se hizo de acuerdo a porcentajes. De la base de datos se calculó, para cada evento, el porcentaje de alumnos (actuarios, físicos y/o matemáticos, según el caso) que lo inscribieron. Luego, en la instancia reducida, se seleccionó el número de alumnos esperados por evento de acuerdo al número reducido de alumnos pero conservando los porcentajes reales. De esta manera, se construyó una instancia reducida que conservara ciertos aspectos de la complejidad (eventos optativos, de *recursamiento* y obligatorios, todos al mismo tiempo) en el semestre par.



Materia	Carrera	Semestre	Inscritos	Especial	Horas	Codigo Rep	Dias imparticion	Semestre	E_matematicas	E_fisica	E_actuaria	E_totales	E_tot_1	Porcentajes
Algebra_Sup_1_1	1,3	1	1	0	1	1	5	2	20	20	20	60	60	15
Algebra_Sup_1_2	1,3	1	1	0	1	1	5							
Algebra_Sup_1_3	1,3	1	1	0	1	1	5							
Algebra_Sup_1_4	1,3	1	1	0	1	1	5							
Algebra_Sup_1_5	1,3	1	1	0	1	1	5							
Algebra_Sup_1_6	1,3	1	1	0	1	1	5							
Calculo_1_1	1,2,3	1	3	0	2	2	5							24
Calculo_1_2	1,2,3	1	3	0	2	2	5							
Calculo_1_3	1,2,3	1	3	0	2	2	5							
Calculo_1_4	1,2,3	1	3	0	2	2	5							
Calculo_1_5	1,2,3	1	3	0	2	2	5							

Figura 4.20: Mediante conservación de porcentajes, se determina el número de alumnos esperados por evento en la instancia reducida del semestre par.

Al utilizar este método con porcentajes ocurrieron situaciones en donde el número de alumnos esperados para algún evento era un número fraccionario, en estos casos se realizaron ajustes manuales de este dato con las funciones *techo* y *piso*. Finalmente, de la instancia reducida se distribuyeron los eventos en el conjunto de 60 alumnos artificiales, siguiendo los procedimientos descritos en el segundo capítulo. Como se puede ver en la Figura 4.21, se consiguió reproducir la existencia de alumnos que cursan a la par eventos de primer y segundo semestre.



2	1	Algebra_Sup_2_2	Calculo_2_6	Geometria_Analitica_2_4	Graficas_Juegos_1	Calculo_1_3	0	0	
2	1	Algebra_Sup_2_8	Calculo_2_5	Geometria_Analitica_2_3	Geometria_Moderna_2_2	Teoria_Numeros_1_2	0	0	
2	1	Algebra_Sup_2_6	Calculo_2_11	Geometria_Analitica_2_5	Conjuntos_Logica_2	Graficas_Juegos_2	0	0	
2	1	Algebra_Sup_2_6	Calculo_2_1	Geometria_Analitica_2_4	Graficas_Juegos_1	Geometria_Moderna_1_2	0	0	
2	1	Algebra_Sup_2_2	Calculo_2_11	Geometria_Analitica_2_4	Graficas_Juegos_2	0	0	0	
2	1	Algebra_Sup_2_1	Calculo_2_11	Geometria_Analitica_2_2	Graficas_Juegos_2	Geometria_Analitica_1_1	Geometria_Moderna_1_2	0	
2	1	Algebra_Sup_2_4	Calculo_2_3	Geometria_Analitica_2_10	Calculo_1_1	Geometria_Moderna_1_1	0	0	

Figura 4.21: Muestra del conjunto de alumnos artificiales en el semestre par, cada renglón corresponde a un alumno, y muestra las materias que tiene inscritas.

4.2.2. Asignación evento-hora

La función objetivo es

$$f(\boldsymbol{X}) = \lambda F(\boldsymbol{X}) + S(\boldsymbol{X}), \tag{4.5}$$

en donde λ es la constante de penalización que, en el semestre impar, se fijó en $\lambda = 4200$. En este caso se tienen sólo 60 alumnos, por lo que, en el peor de los casos donde cada uno tiene 7 horas libres por día, habría un total de $7 \times 5 \times 60 = 2100$ violaciones de la restricción suave. Esta es una cota superior de $S(\boldsymbol{X})$ $\forall \boldsymbol{X}$, por lo que se fijó

$$\lambda = 2100. \tag{4.6}$$

En este caso también se fijó un tiempo de paro en ambos algoritmos. La tabla de parámetros de AGBL se muestra a continuación:

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
t	Tiempo de paro (min)	800

Tabla 4.9: Parámetros utilizados en AGBL.

En NAH se utilizaron:

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
n_{gen}	Número de generaciones	3000
MAX	Cota superior del orden de cada alumno-grupal	16
t	Tiempo de paro (min)	800

Tabla 4.10: Parámetros utilizados en NAH.

En la Figura 4.22 se pueden ver las distribuciones análogas a las del semestre impar. Es importante mencionar que, al construir la instancia artificial de alumnos utilizando los datos obtenidos en la fase anterior, se dieron casos de eventos en los cuales faltaban estudiantes para llenar el requerimiento de alumnos esperados. Una posible explicación es que, en un semestre par, los alumnos que cursan eventos de segundo semestre y/o recursan los de primero no son sólo los que pasaron oficialmente a segundo, sino también aquellos cuya inscripción oficial es a cuarto, sexto y octavo semestre. Con la información recopilada de la página web de FCUNAM no se puede conocer este número excedente de alumnos inscritos que no pertenecen al segundo semestre oficial, por lo que no se puede eliminar al momento de crear el conjunto artificial de alumnos. Para resolver este problema sin aumentar el número de alumnos artificiales de segundo semestre, se incrementó el límite de eventos de 6 a 7. Por ello, en la distribución original hay alumnos que toman desde 3 hasta 7 eventos.
Con el algoritmo de agrupación se redujo la instancia de 60 a 35 alumnos. La reducción no fue tan significativa (comparada con la del semestre impar) debido a que, al haber un menor número de alumnos esperados por evento, habrá un mayor número de alumnos-grupales, cada uno compuesto de pocos alumnos reales. En la segunda distribución hay alumnos que inscriben hasta 16 eventos, por lo que te tomó $MAX = 16$.

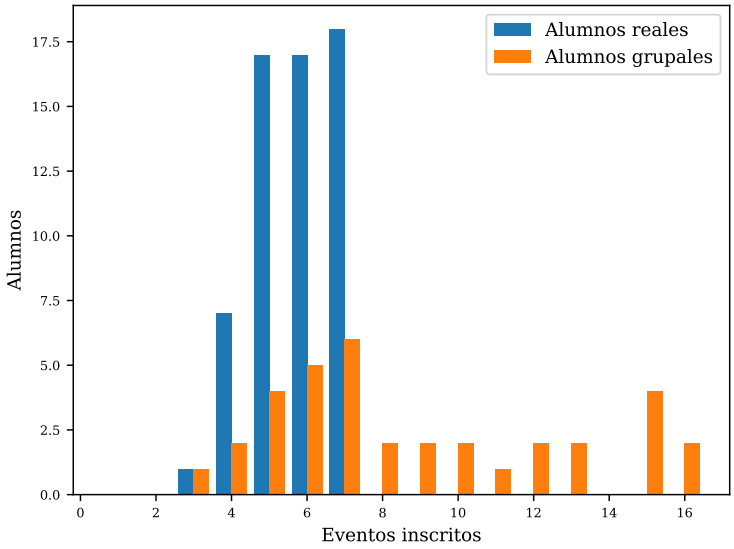


Figura 4.22: Distribución del número de alumnos que inscriben i eventos. Se muestran la distribución original (azul) y la del primer conjunto de alumnos-grupales (naranja).

En este semestre también se compararon las fases de construcción y mejoramiento en un individuo de la primera generación en ambos métodos. Se hicieron tres ejecuciones de cada algoritmo.

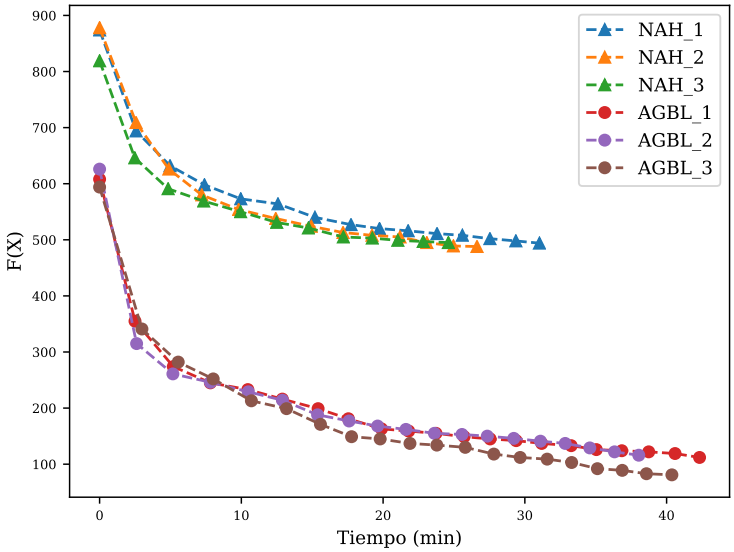


Figura 4.23: Gráficas de NAH y AGBL durante la fase de construcción de la búsqueda local.

En la fase de construcción (Figura 4.24) se puede ver que las soluciones dadas por AGBL son mejores que las de NAH en todas las iteraciones. Esto se debe a que es más probable violar la restricción fuerte del empalme de eventos cuando se tienen alumnos-grupales.

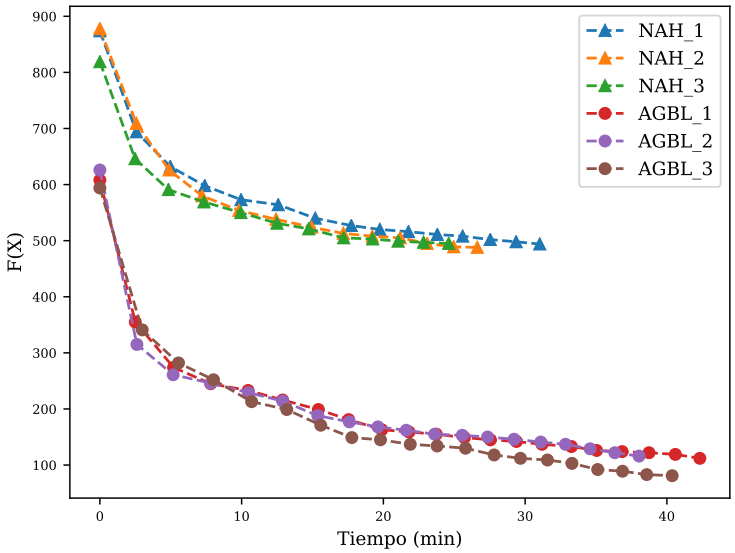


Figura 4.24: Gráficas de NAH y AGBL durante la fase de construcción de la búsqueda local.

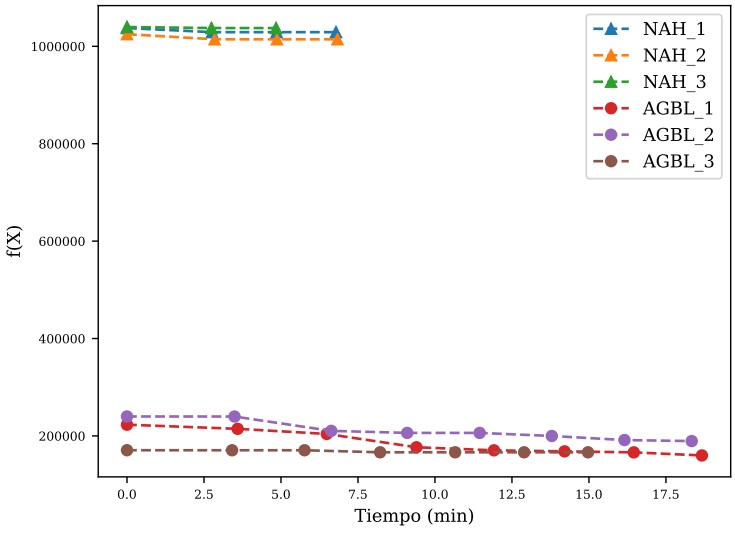


Figura 4.25: Gráficas de NAH y AGBL durante la fase de mejoramiento de la búsqueda local.

La continuación de las seis gráficas en la fase de mejoramiento se puede ver en la Figura 4.25. En todos estos casos, la fase de mejoramiento fue más rápida que la de construcción. Al calcular los promedios de tiempos de ejecución en cada fase se consiguen los datos de la Tabla 4.11. En este caso, pasar de una generación a otra mediante cómputo en serie tardaría alrededor de 8×40 min $= 320$ min ≈ 5 horas. Con cómputo en paralelo, en la computadora con 4 procesadores, esto se redujo a 40×2 min $= 80$ min.

Fase	Tiempo promedio en NAH (min)	Tiempo promedio en AGBL (min)
Construcción	27	40
Mejoramiento	6	17
Búsqueda completa	33	57

Tabla 4.11: Tiempos de ejecución de la búsqueda local variable.

Se hicieron tres corridas completas de cada método, los resultados obtenidos se muestran en la siguiente figura:

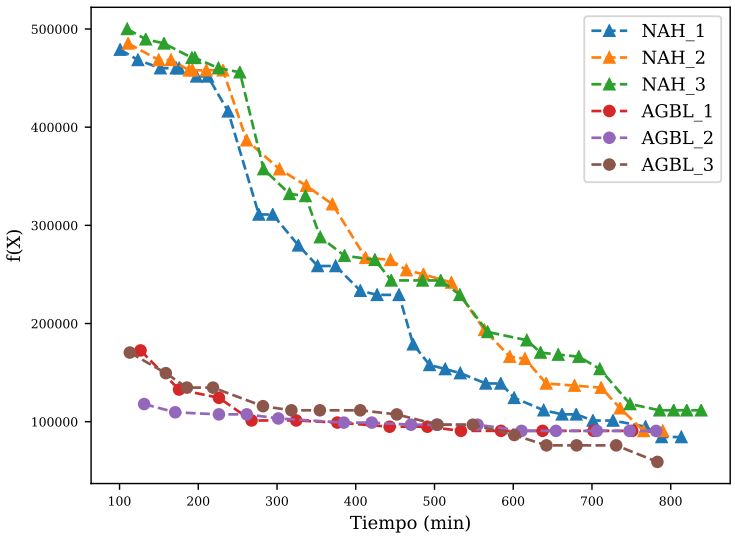


Figura 4.26: Evolución de las soluciones en NAH y AGBL en el semestre par.

De nuevo, las soluciones de NAH comienzan con valores mucho mayores de $f(\boldsymbol{X})$ que las de AGBL. A pesar de esto, se puede ver que en los seis casos se llegó a valores similares de la función objetivo, lo que puede indicar que el óptimo de la misma se encuentra alrededor de estos valores. De ser así, significaría que esta instancia no tiene solución factible, lo que, de hecho, es bastante

común en los problemas de asignación escolar más complejos [58].
En todos los casos, ambos métodos convergieron a óptimos cercanos entre ellos en los 800 minutos de ejecución. El que en esta instancia no haya habido una diferencia significativa entre la velocidad y calidad de soluciones de los dos algoritmos puede deberse al número tan reducido de alumnos que se utilizó (60), pues, como se mencionó anteriormente, esto provocó que hubiera un gran número de alumnos-grupales, cada uno compuesto por pocos alumnos reales. Estos alumnos-grupales tan pequeños convergen rápidamente al conjunto original, lo que, a su vez, significa que NAH se transforma rápidamente en AGBL. De aquí se infiere que no existen grandes diferencias entre utilizar NAH o AGBL cuando el número de alumnos es reducido, esto se corrobora en el artículo donde se introduce NAH pues, al compararlo con AGBL en problemas muy pequeños, la diferencia de tiempo entre ambos para llegar al óptimo es de segundos [55].
Las mejores soluciones tuvieron un valor de función objetivo de $90,579$, 84,435 y 59,058, respectivamente. A pesar de que en ninguna solución se alcanzó la factibilidad de la función objetivo, en los horarios de colores correspondientes apareció de nuevo el esquema de horarios de bandas (Figuras 4.27, 4.28 y 4.29), especialmente para los eventos de tronco común (los Cálculos, Geometrías y Álgebras). Esto sugiere que, incluso cuando se agrega complejidad al problema en FCUNAM, los esquemas en bandas son una buena solución para crear horarios en donde se evite que los alumnos tengan tanto eventos empalmados como horas libres.

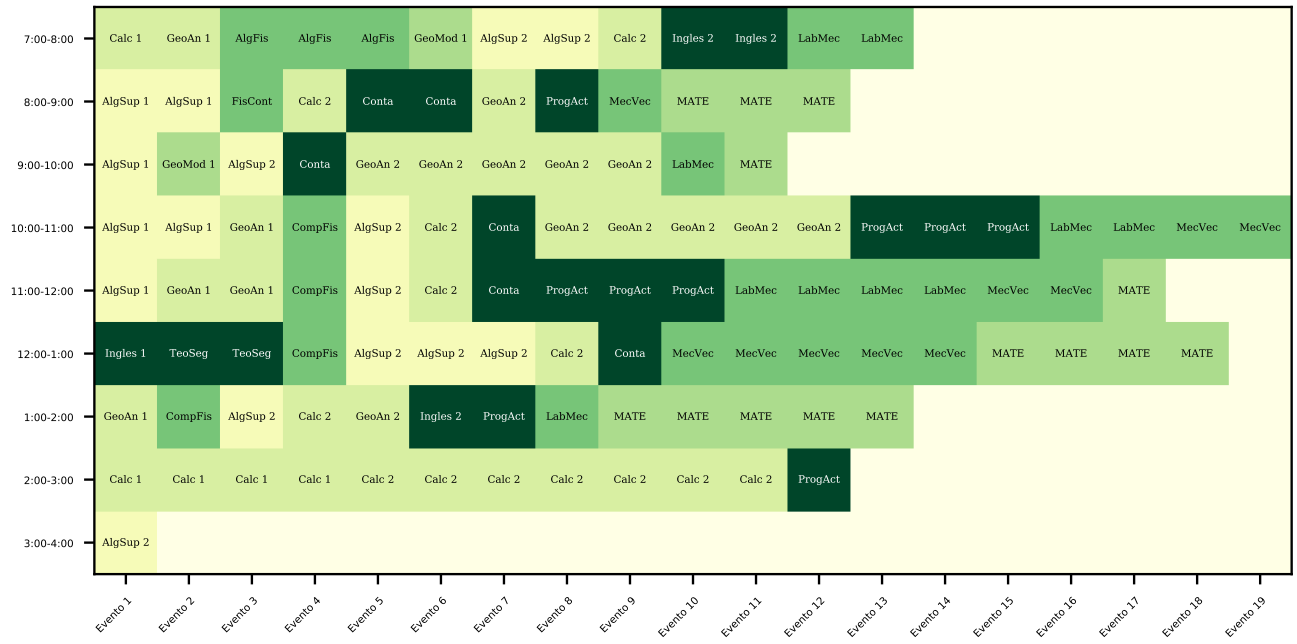


Figura 4.27: Horario de color para la solución con $f(\boldsymbol{X}) = 90,579$.

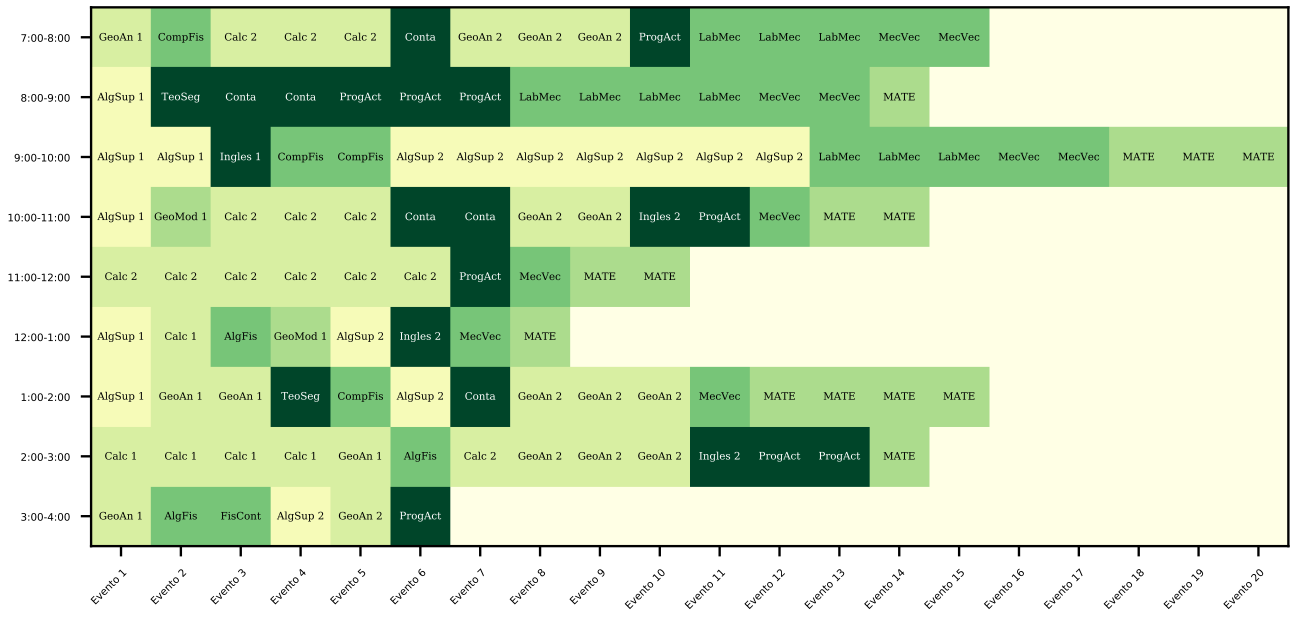


Figura 4.28: Horario de color para la solución con $f(\boldsymbol{X}) = 84,435$.

Figura 4.29: Horario de color para la solución con $f(\boldsymbol{X}) = 59,058$.

El promedio de f de las tres mejores soluciones es $78,024$. Aunque este valor está alejado del

óptimo ideal $f = 0$, y ni siquiera se encuentra en la zona de factibilidad, un horario asociado a este valor no es tan poco efectivo como se esperaría intuitivamente. Suponiendo que se hubieran violado el mayor número posible de restricciones suaves (2100), significaría que hay un total de $\frac{78024-2100}{2100} \approx 36$ violaciones de restricciones fuertes en dicho horario. Como se cumplen los requerimientos de capacidad y tipo de salones para cada evento, estas 36 violaciones tienen que corresponder a empalmes de materias, los cuales se cuentan por día. Esto significa que hay, aproximadamente, $\frac{36}{5} \approx 7$ empalmes diarios. Tomando en cuenta que hay 60 alumnos y 117 eventos, un horario con 7 empalmes diarios no está tan alejado de ser viable.

Por otro lado, la ausencia de bandas tan bien definidas como en el semestre impar puede deberse al *ruido* en la información relacionado con alumnos de semestres posteriores, mismo por el que se tuvo que aumentar de manera artificial el límite de eventos de 6 a 7. Otra posible introducción de ruido en los datos es que se consideraron todas las materias optativas de matemáticas del nivel I a IV, cuando, en realidad, puede que un alumno de segundo semestre no inscriba todos los eventos de esa categoría, sino sólo aquellos para los que se considere a sí mismo con el conocimiento suficiente. Finalmente, los resultados pudieron haber sido afectados por el número tan reducido de alumnos, lo que abre la posibilidad a repetir la instancia par con más alumnos (y con mayor poder de cómputo) para confirmar que los horarios de bandas siguen siendo el patrón dominante en los óptimos de semestres pares.

Respecto a los días de impartición y salones asignados, se obtuvieron resultados completamente análogos al semestre impar. En la sección correspondiente se discutieron las posibles razones de estos resultados, por lo que no se repiten aquí.

4.2.3. Asignación evento-maestro

Considerando los dos primeros semestres, se tuvieron 116 eventos en total. De manera análoga al semestre impar, y con la finalidad de tener un espacio de búsqueda más diverso, se construyó una base artificial con $l = 2n$, es decir, con 232 maestros. De estos 232, 217 fueron generados con los algoritmos descritos en el capítulo dos, mientras que los 15 restantes fueron agregados manualmente (con rubros aleatorios) y corresponden a maestros que quieren impartir los eventos de Laboratorio de Mecánica, que es el evento de laboratorio que tienen que tomar los estudiantes de la licenciatura de Física en el segundo semestre. En la distribución de maestros prospectos (Figura 4.30) se puede ver que, para cada evento, hay al menos 2 maestros que quieren impartirlo. Luego, en la función objetivo

$$\widetilde{f}(\boldsymbol{Y}) = -\mu\widetilde{F}(\boldsymbol{Y}) + \widetilde{S}(\boldsymbol{Y}), \tag{4.7}$$

se tuvo que elegir nuevamente un valor para la constante μ de penalización. En el mejor de los casos, en donde todos los maestros tienen 30 años de experiencias, calificación de 10 y a todos se les asignaron sus primeros tres eventos preferidos, la contribución total a $\widetilde{S}$ sería de $232 \times 30 = 6960$.

Como esto es una cota al valor máximo de $\widetilde{S}$, se tomó

$$\mu = 6960. \tag{4.8}$$

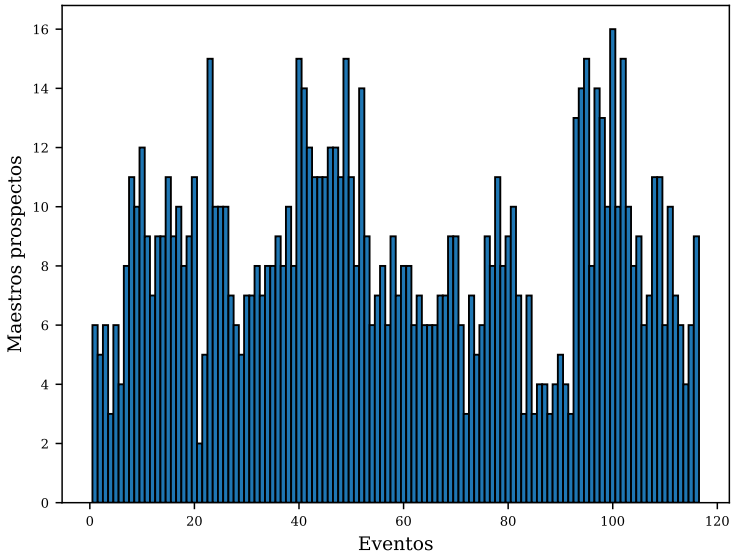


Figura 4.30: Distribución del número de maestros que quieren impartir cada evento.

Los parámetros de AGBL (Tabla 4.12) fueron los mismos que en el semestre impar.

Parámetro	Descripción	Valor
p_1	Tamaño de población	8
p_2	Tamaño del *pool*	4
p_m	Probabilidad de mutación	0.4
ω	Frecuencia de búsqueda local	1000
t	Tiempo de paro (min)	360

Tabla 4.12: Parámetros utilizados en AGBL para la asignación evento-maestro.

En este caso, se volvieron a analizar los cuatro casos discutidos anteriormente: $\alpha = 1$, $\beta = 1$, $\gamma = 1$ y el caso de equilibrio. Los resultados se muestran en la siguiente Figura:

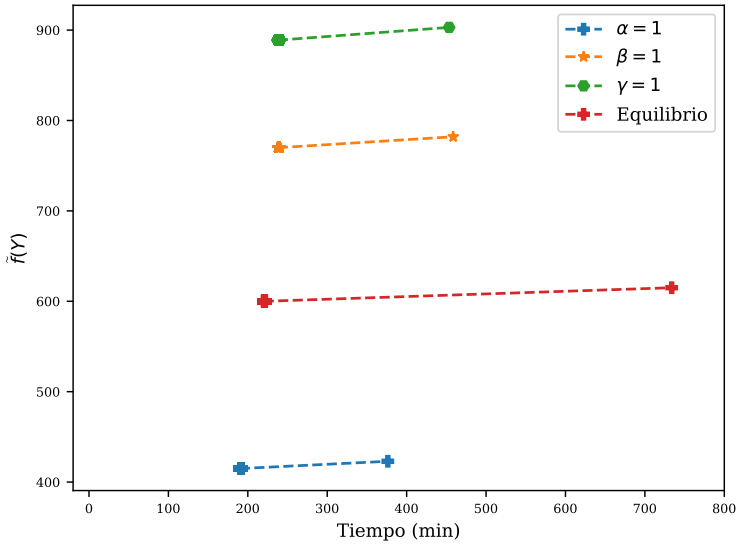


Figura 4.31: Evolución de las soluciones en AGBL, asignación evento-maestro.

Como era de esperarse, por la complejidad extra en esta instancia de semestre par, en todos los casos se obtuvo sólo un clúster de puntos más el punto final. Además, en la solución de equilibrio, AGBL empezó la búsqueda local variable antes del límite de tiempo $t = 360$ min., por lo que el programa siguió hasta que acabó con este proceso, lo que ocurrió en $t = 733.6$ min. Por otro lado, en estas gráficas (y en sus análogas del semestre impar) se puede observar que, en la asignación evento-maestro, la parte genética de AGBL no es tan efectiva para encontrar mejores soluciones como lo es su parte de búsqueda local variable.
La primera comparación entre soluciones se hizo en términos de su satisfacción. Como se observa en la Figura 4.32 y en la Tabla 4.13, en el caso $\alpha = 1$ se tuvo un mucho mayor porcentaje de maestros satisfechos, respecto a las otras soluciones. Esto coincide con lo que se esperaba.

Caso	Maestros satisfechos (%)
$\alpha = 1$	91.7
$\beta = 1$	51.8
$\gamma = 1$	46.2
Equilibrio	62.7

Tabla 4.13: Porcentaje de maestros satisfechos en cada solución.

La solución que tuvo el segundo mayor porcentaje de maestros satisfechos fue la de equilibrio, lo que también se esperaba de acuerdo a la teoría.

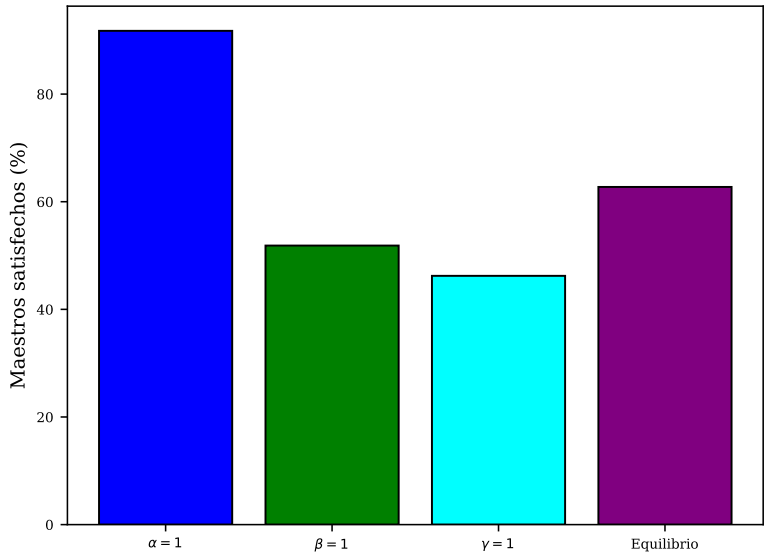


Figura 4.32: Porcentaje de maestros a los que se les asignó su primera o segunda opción de evento.

En el rubro de calificación, la solución $\beta = 1$ fue la que tuvo los mejores maestros, seguida del caso de equilibrio (Figura 4.33 y Tabla 4.33).

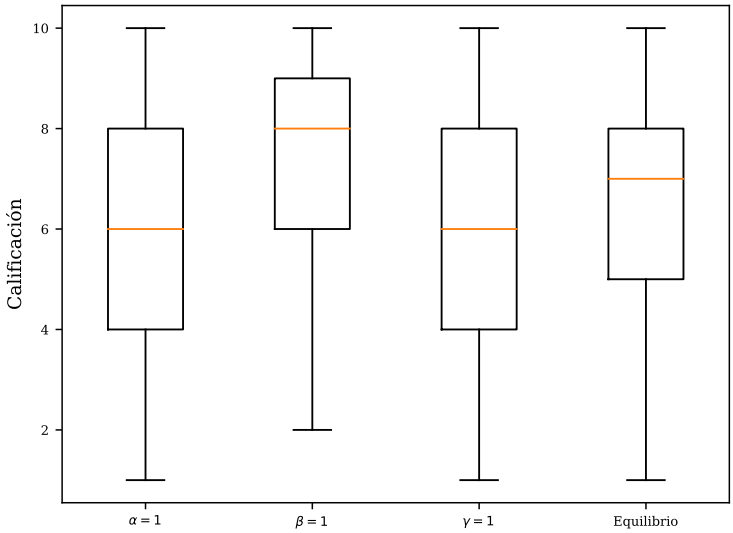


Figura 4.33: Diagramas de caja de calificaciones de maestros con al menos un evento asignado.

Caso	Calificación promedio
$\alpha = 1$	5.9
$\beta = 1$	7.2
$\gamma = 1$	5.8
Equilibrio	6.5

Tabla 4.14: Promedio de calificación en cada solución.

Finalmente, en el rubro de años de experiencia también se obtuvieron resultados que coincidieron con lo que se esperaba (Figura 4.34 y Tabla 4.34), pues la solución $\gamma = 1$ tuvo la mayor experiencia promedio, seguida por la solución de equilibrio.

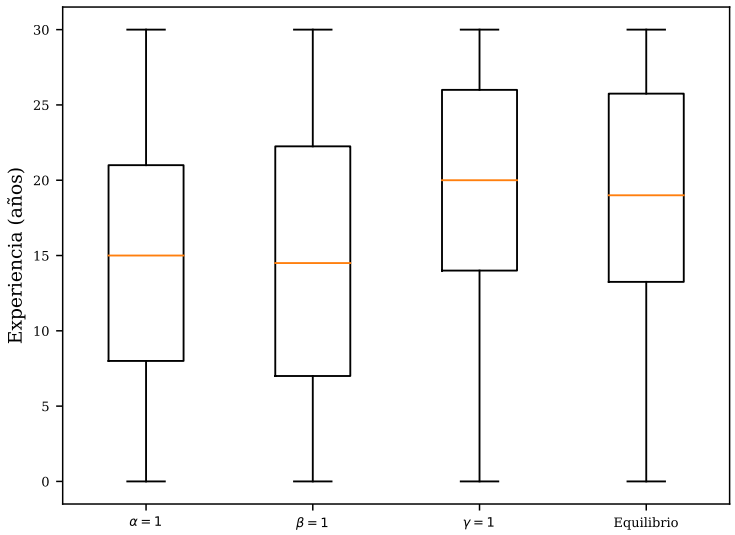


Figura 4.34: Diagramas de caja de los años de experiencia de maestros con al menos un evento asignado.

Caso	Experiencia promedio (años)
$\alpha = 1$	14.8
$\beta = 1$	14.9
$\gamma = 1$	19.3
Equilibrio	18.5

Tabla 4.15: Promedio de años de experiencia en cada solución.

A pesar de que en la asignación evento-maestro no existe una referencia de comparación con FCUNAM (i.e. un análogo a los horarios de banda en la asignación evento-hora), de los resultados obtenidos, especialmente de las comparaciones entre soluciones, se puede ver que, tanto para el semestre par como el impar, esta fase del modelo cumple con el objetivo de facilitar la tarea de asignar eventos a maestros de tal manera que se maximicen los rubros que se deseen.
Las soluciones completas del problema de asignación en FCUNAM para el semestre par son las tuplas $(\boldsymbol{X}_{opt}, \boldsymbol{Y}_{opt})$.

4.3. Trabajo futuro e implementación real

El modelo propuesto y desarrollado en esta investigación no pretende automatizar por completo ni resolver en su totalidad el problema de asignación en FCUNAM, sino funcionar como una herramienta auxiliar que, combinada con el conocimiento y experiencia del personal administrativo de esta institución, permita crear asignaciones efectivas de eventos-horas y eventos-maestros. En este sentido, los esquemas de bandas que aparecieron en las soluciones óptimas pueden funcionar como modelo guía para la creación de horarios reales. Asimismo, la asignación de evento-maestro también puede ser asistida por este modelo, pues, como se observó en ambos semestres, se requieren de asignaciones diferentes de acuerdo al rubro (α, β, γ) al que se le quiera dar prioridad.
Dicho esto, el presente trabajo es un primer intento de crear dicha herramienta auxiliar, por lo que existen varias modificaciones que se podrían implementar al mismo, además de estudios extra que serían convenientes analizar.
En primer lugar, el equipo de cómputo con el que se obtuvieron resultados es bastante limitado en comparación con equipos actuales utilizados específicamente para cómputo científico, por lo que un primer paso consistiría en utilizar un equipo con una mayor capacidad e intentar resolver una instancia con horario completo (de 7:00 A.M. a 10:00 P.M.) y más semestres simultáneos (e.g. los primeros 4) con sus alumnos correspondientes. En una instancia de ese tamaño se recomendaría utilizar NAH pues, de acuerdo a los resultados del semestre impar, este algoritmo es más efectivo que AGBL cuando se resuelven asignaciones con un gran número de alumnos.
Además de esto, en los dos semestres se observó que la asignación de salones a eventos llegaba a un óptimo en las primeras generaciones (al asignar los laboratorios y salones de cómputo correspondientes), por lo que después perdía sentido seguir buscando mejoras en las soluciones con la vecindad V_1, encargada de intercambiar salones. Esto se debe, principalmente, a que se cuenta con la capacidad y tipos de salones necesarios para la impartición de clases en FCUNAM, además de que no se consideraron fenómenos de aglomeración.
Sin embargo, cabría la posibilidad de separar la asignación de evento-salón en una fase independiente, y así convertir este modelo de 3 fases en uno con 4. De esta manera se podría evitar la búsqueda extra con V_1, y así mejorar la asignación evento-hora. Adicionalmente, el tener una fase

independiente de evento-salón permitiría incluir nuevas restricciones específicas de esa asignación. Por ejemplo, en [59] se propone como restricción suave de evento-salón que los alumnos recorran la menor distancia total posible al cambiar de salón entre sus diferentes clases. Dichas distancias totales podrían ser calculadas fácilmente al contar con la instancia de alumnos y el horario óptimo de la fase evento-horarios, además de las distancias fijas entre los distintos edificios de FCUNAM, mismas que podrían obtenerse utilizando software de visión por satélite (Figura 4.35).

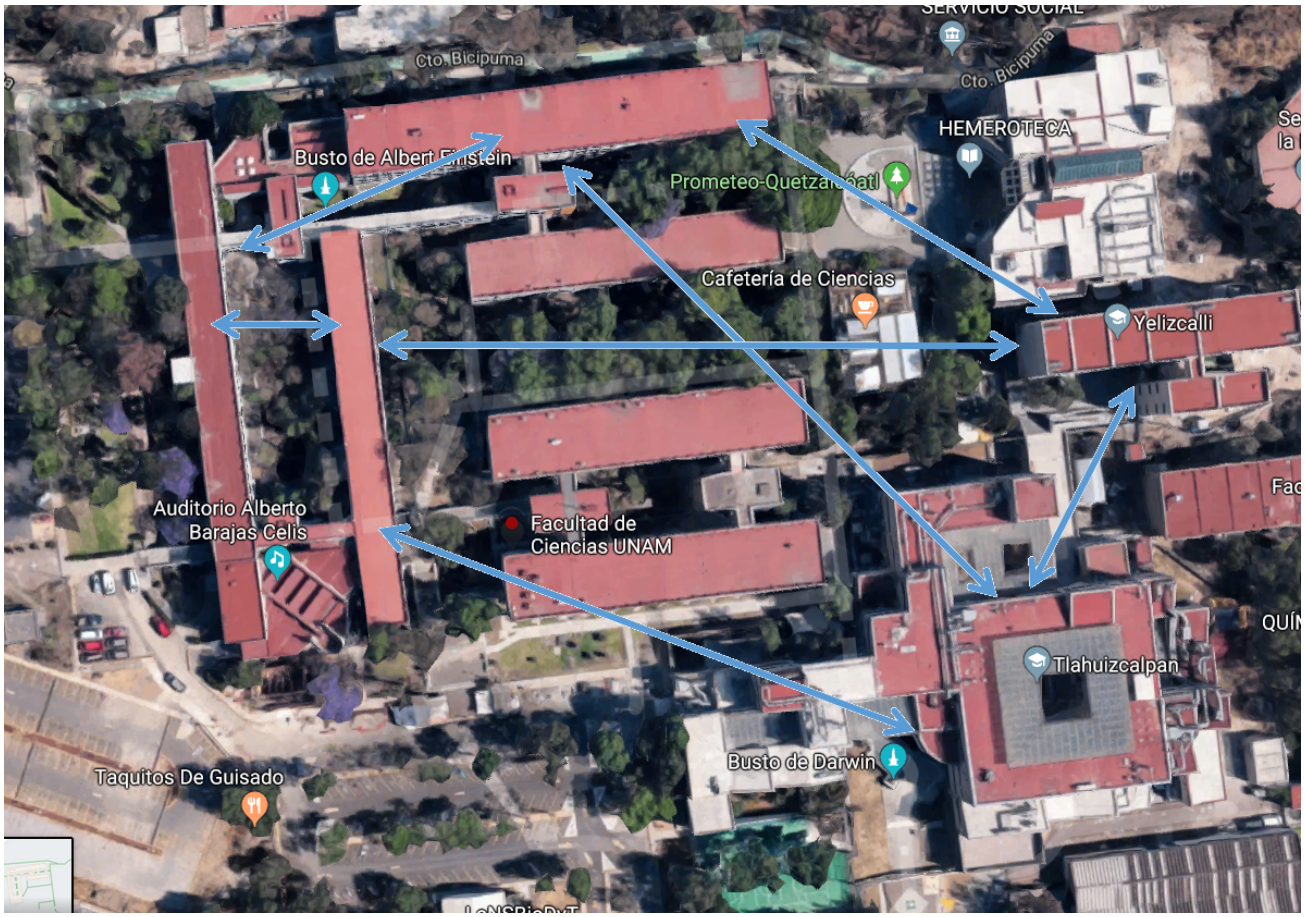


Figura 4.35: Una forma de calcular las distancias entre los edificios de FCUNAM es mediante software de visión por satélite. Fotografía cortesía de [60]

Una vez calculadas estas distancias, se introducirían a la fase independiente de evento-salón en forma de una matriz que represente la gráfica K_6 con pesos, en donde cada vértice es un edificio de FCUNAM y el peso de una arista es la distancia entre sus vértices (edificios) extremos (Figura 4.36).

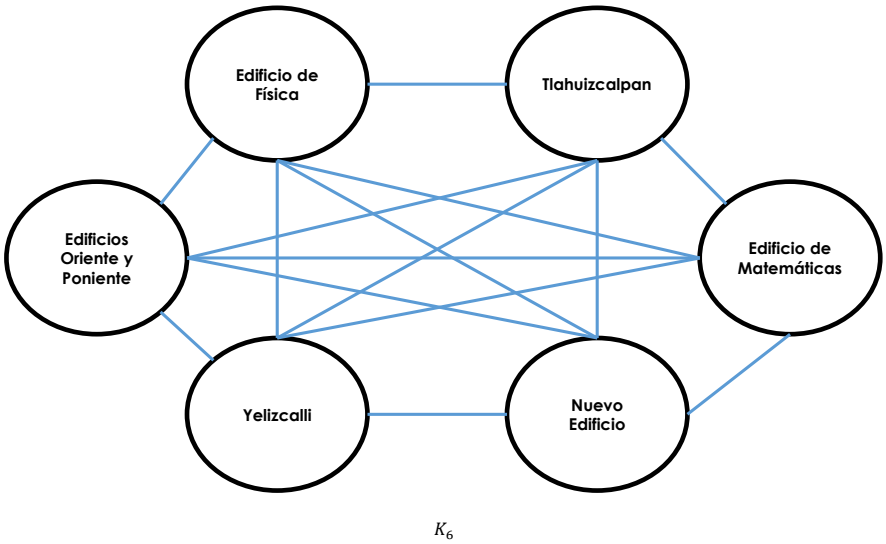


Figura 4.36: Gráfica isomorfa a K_6 asociada a los edificios en FCUNAM, no se muestran los pesos para evitar el amontonamiento de información.

Después de introducir estas modificaciones, se podría probar el modelo resultante de 4 fases en una instancia real reducida. Para ello, se sustituiría la primer fase (recopilación de datos y estadística) con la participación de alumnos reales, que formarían parte de una instancia de prueba para el siguiente semestre de acuerdo a los eventos que planean inscribir (Tabla 4.16). Finalmente, se podría modificar esta instancia real para simular los fenómenos de aglomeración y ver qué tanto afectan a los horarios óptimos.

Fase	Descripción
1	Encuesta de eventos a inscribir, realizada a p alumnos.
2	Asignación evento-hora
3	Asignación evento-salón
4	Asignación evento-maestro

Tabla 4.16: Esquema de modelo en 4 fases en una instancia real reducida.

Estos son sólo algunos de los distintos análisis técnicos que se pueden formular a partir del modelo aquí presentado. Por otro lado, también se puede avanzar en la instauración de estas metodologías como herramienta auxiliar para el personal administrativo en FCUNAM. Para lograr este objetivo, se tendría que crear una interfaz gráfica de usuario (GUI por las siglas en inglés de *Graphic User Interface*) amigable con el usuario (Figuras 4.37 y 4.38).

Figura 4.37: Esquema de una GUI que permitiría traducir este trabajo en una herramienta auxiliar para el personal administrativo en FCUNAM. Fase de evento-hora.

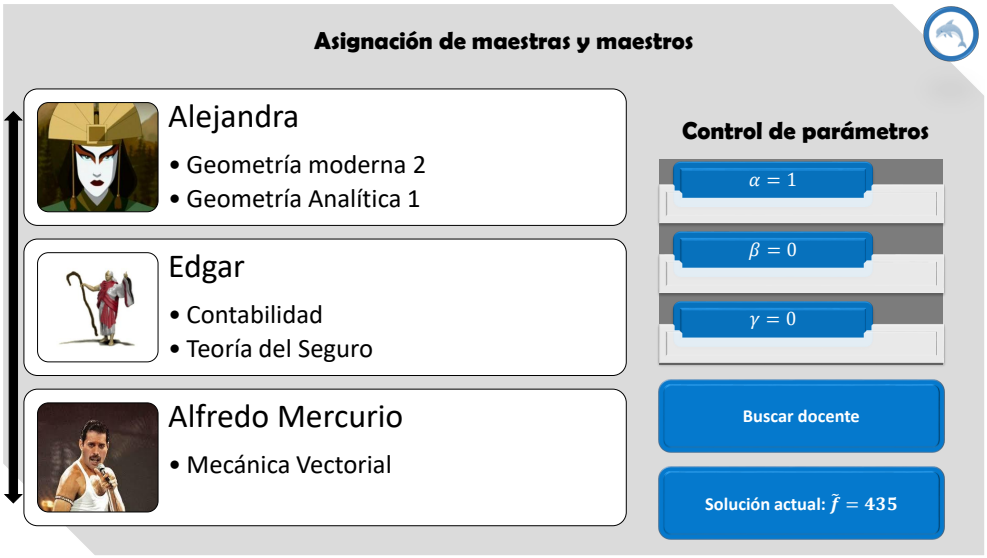


Figura 4.38: Esquema de una GUI que permitiría traducir este trabajo en una herramienta auxiliar para el personal administrativo en FCUNAM. Fase de evento-maestro. Imágenes de maestros son cortesía de [61], [62] y [63]

Capítulo 5

Conclusiones

En este trabajo se presentó la propuesta de un modelo del problema de asignación escolar en la Facultad de Ciencias de la Universidad Nacional Autónoma de México (FCUNAM). Se desarrolló el mismo en tres fases clave de la asignación escolar: obtención/generación de datos de estudiantes, creación de horarios para cada evento y designación de maestros que los imparten. Para estudiar la utilidad de dicho modelo, se implementó en dos instancias artificiales, una para un semestre impar y otra para un semestre par. En cada caso se utilizó la metaheurística AGBL así como la matheurística NAH para resolverlo.

De los resultados obtenidos, se pudieron comprobar los beneficios de utilizar un modelo matemático como herramienta auxiliar para la creación de horarios en esta institución educativa. Específicamente, se observó que, a pesar de trabajar con modelos simplificados del problema real, las soluciones obtenidas pueden dar indicios de cómo crear mejores horarios. El ejemplo más claro de esto fue que, en las soluciones óptimas de horarios de eventos, se encontraron *esquemas de bandas*, los cuales ya han sido implementados con anterioridad en FCUNAM. También se comprobó que mientras más alumnos se consideren en el problema, resulta más eficiente utilizar NAH que AGBL, pues, en todas las ejecuciones del semestre impar, NAH llego a mejores soluciones y en menos tiempo que AGBL. Después, se consideraron varios caminos de investigación futura, así como los pasos necesarios para lograr la instauración de estas metodologías como una herramienta auxiliar para el personal administrativo de FCUNAM.

Finalmente, en este trabajo se comprobó que, aún utilizando los métodos matemáticos más recientes y trabajando con instancias muy reducidas, el problema de asignación escolar en FCUNAM es particularmente difícil de resolver, no sólo porque el problema en sí mismo es **NP-Duro**, sino también por el gran número de variables involucradas en este caso particular.

Apéndice A

Diagramas de caja

Sea $S \subseteq \mathbb{R}$ un conjunto finito y ordenado. Un diagrama de caja con bigotes (o sólo diagrama de caja) es una gráfica que muestra, mediante cinco cantidades, un resumen estadístico de los datos en S [64]. Las cinco cantidades son:

1. La mediana: Si $|S|$ es impar, la mediana $\widetilde{s}$ es el número de *en medio* en S. Si $|S|$ es par, $\widetilde{s}$ es el promedio de los dos números que se encuentran en medio.

2. El mínimo: Es el número $m \in S$ tal que $m \leq x \ \forall x \in S$.

3. El máximo: Es el número $M \in S$ tal que $M \geq x \ \forall x \in S$.

4. El cuartil inferior: Sea $S_1 = \{x \in S | x \leq \widetilde{s}\}$. El primer cuartil Q_1 es la mediana de S_1.

5. El cuartil superior: Sea $S_3 = \{x \in S | x \geq \widetilde{s}\}$. El tercer cuartil Q_3 es la mediana de S_3.

Así, un diagrama de caja y bigotes (figura A.1) consiste en una caja delimitada por Q_1 y Q_3, bigotes que van de los cuartiles a m y M, y una línea dentro de la caja, a la altura de $\widetilde{s}$.

Figura A.1: Diagrama general de caja y bigotes.

Por ejemplo, sea $S = \{25, 28, 29, 29, 30, 34, 35, 35, 37, 38\}$. Como tiene cardinalidad par, la mediana es

$$\widetilde{s} = \frac{30 + 34}{2} = 32. \tag{A.1}$$

Luego, $S_1 = \{25, 28, 29, 29, 30\}$ y $S_2 = \{34, 35, 35, 37, 38\}$, por lo que

$$Q_1 = 29,$$
$$Q_3 = 35.$$

Finalmente, se tiene que

$$m = 25,$$
$$M = 38.$$

El diagrama de caja se muestra a continuación:

Figura A.2: Diagrama de caja y bigotes asociado a S.

Apéndice B

Código utilizado en este trabajo

```
#!/usr/bin/env python
import math
import numpy as np
import multiprocessing as mp
import time
import collections
import matplotlib.pyplot as plt
import pandas as pd
import xlsxwriter


#FUNCIONES AUXILIARES
#######################################
#######################################
#######################################

#Funcion auxiliar para intersectar materias de G.
def interG(G,alumnos):
    numu = np.size(G)
    intro = set(alumnos[int(G[0])])
    for a in range(1,numu):
        new = set(alumnos[int(G[a])])
        intro = intro.intersection(new)
    return intro

#Funcion auxiliar para saber horas de inicio y final de una materia.
def intervalo_local(arreglo):
    horas = [i for i, e in enumerate(arreglo) if e!=0]
    inicia = horas[0]
    termina = horas[-1]
    return inicia, termina

#Funcion auxiliar para saber horas de inicio y final de las materias.
def intervalo(mat_horario):
    n_mat = np.size(mat_horario[:,0])
    mat_inter = np.zeros((n_mat,2))
```

```
    for a in range(n_mat):
        materia = mat_horario[a,:]
        inicia, termina = intervalo_local(materia)
        mat_inter[a,0] = inicia+7
        mat_inter[a,1] = termina+8
    return mat_inter

#Funcion auxiliar para mutar un hijo.
def mutacion(hijo,proba_m,alumnos):
    ran = np.random.uniform()
    if ran<proba_m:
#        muta_1 = vecindad_select(hijo,alumnos,1)
        muta_1 = vecindad_select(hijo,alumnos,2)
        muta_2 = vecindad_select(muta_1,alumnos,3)
        mutado = vecindad_select(muta_2,alumnos,4)
    else:
        mutado = hijo
    return mutado

#Funcion auxiliar para crossover del metodo genetico.
def cruza(ma,pa,alumnos):
    mats_ma = ma[0]
    mats_pa = pa[0]
    math_ma = ma[1]
    matd_ma = ma[2]
    math_pa = pa[1]
    matd_pa = pa[2]
    n_mate = np.size(math_ma[:,0])
    n_horas = np.size(math_ma[0,:])
    n_dias = np.size(matd_ma[0,:])
    math1 = np.zeros((n_mate,n_horas))
    math2 = np.zeros((n_mate,n_horas))
    matd1 = np.zeros((n_mate,n_dias))
    matd2 = np.zeros((n_mate,n_dias))
    mats1 = np.zeros(n_mate)
    mats2 = np.zeros(n_mate)
    for a in range(n_mate):
        ran = np.random.uniform()
        if ran<0.5:
            math1[a,:] = np.array(math_ma[a,:])
            math2[a,:] = np.array(math_pa[a,:])
        else:
            math1[a,:] = np.array(math_pa[a,:])
            math2[a,:] = np.array(math_ma[a,:])
    for b in range(n_mate):
        ron = np.random.uniform()
        if ron<0.5:
            matd1[b,:] = np.array(matd_ma[b,:])
            matd2[b,:] = np.array(matd_pa[b,:])
        else:
            matd1[b,:] = np.array(matd_pa[b,:])
            matd2[b,:] = np.array(matd_ma[b,:])
    for c in range(n_mate):
        ren = np.random.uniform()
        if ren<0.5:
```

```
            mats1[c] = mats_ma[c]
            mats2[c] = mats_pa[c]
        else:
            mats1[c] = mats_pa[c]
            mats2[c] = mats_ma[c]
    alumno1 = genera_alumnoshoras(math1,matd1,alumnos)
    alumno2 = genera_alumnoshoras(math2,matd2,alumnos)
    hijo1 = [mats1,math1,matd1,alumno1]
    hijo2 = [mats2,math2,matd2,alumno2]
    return hijo1, hijo2

#Funcion auxiliar para ruleta del metodo genetico.
def matchpool(poblacion,poblacion_f,n_pool):
    tamano = np.size(poblacion_f)
    f_modi = [1.0/(poblacion_f[j]+1) for j in range(tamano)]
    normaliza = np.sum(f_modi)
    probas = [f_modi[i]/normaliza for i in range(tamano)]
    index = [i for i in range(tamano)]
    index_pool = np.random.choice(index,size=n_pool,replace=False,p=probas)
    pool = []
    for a in range(n_pool):
        who = int(index_pool[a])
        seleccion = poblacion[who]
        pool.append(seleccion)
    return pool

#Funcion auxiliar para acomodar una lista de soluciones en orden de fitness.
def acomodar(poblacion,hard_coef,necesario_mat,salones):
    tamano = len(poblacion)
    fit = [objetivo(poblacion[i], hard_coef,
    necesario_mat,salones) for i in range(tamano)]
    orden = np.argsort(fit)
    new_fit = np.sort(fit)
    new_pop = []
    for a in range(tamano):
        index = int(orden[a])
        elemento = poblacion[index]
        new_pop.append(elemento)
    return new_pop, new_fit

#Funcion auxiliar para mostrar las clases en lugar de los salones en alumnos_horas.
def intercambia_info(alumnos_horas,mat_salon):
    n_alumnos = np.size(alumnos_horas[:,0,0])
    n_horas = np.size(alumnos_horas[0,:,0])
    nuevo = np.zeros((n_alumnos,n_horas,5))
    for x in range(n_alumnos):
        for y in range(n_horas):
            for z in range(5):
                salon = int(alumnos_horas[x,y,z])
                if salon<0:
                    nuevo[x,y,z] = int(salon)
                elif salon==0:
                    continue
                else:
                    clase = [i+1 for i, e in enumerate(mat_salon) if e==salon]
```

```
                    nuevo[x,y,z] = clase[0]
    return nuevo

#Funcion auxiliar para sumar arreglos.
def suma_detector(arreglo1,arreglo2):
    tamano = np.size(arreglo1)
    arreglo = np.zeros(tamano)
    for a in range(tamano):
        value1 = arreglo1[a]
        value2 = arreglo2[a]
        if value1!=0 and value2!=0:
            if value1<0:
                arreglo[a] = value1-1
            else:
                arreglo[a] = -1
        else:
            arreglo[a] = value1+value2
    return arreglo

#Funcion auxiliar para busqueda de entradas seguidas en una lista.
def buscar_continuo(lista,elementos):
    contador = 0
    tamano = np.size(lista)
    ubicaciones = np.zeros(elementos)

    for a in range(tamano):
        maybe = lista[a]
        if maybe==0:
            continue
        else:
            ubicaciones[0] = a
            break

    for c in range(elementos-1):
        for b in range(int(ubicaciones[c])+1,tamano):
            maybo = lista[b]
            if maybo==0:
                continue
            else:
                ubicaciones[c+1] = b
                break

    distancias = [ubicaciones[i+1]-ubicaciones[i]-1 for i in range(elementos-1)]
    distancia = np.sum(distancias)
    return distancia

#Funcion auxiliar para medir el numero de horas libres que tiene un alumno en un dia.
def contador_horas(alumno):
    tamano = np.size(alumno)
    contador = 0
    ubicaciones = []
    for a in range(tamano):
        maybe = alumno[a]
        if maybe==0:
            continue
```

```
        else:
            ubicaciones.append(a)
    distancias = [ubicaciones[i+1]-ubicaciones[i]-1
    for i in range(np.size(ubicaciones)-1)]
    distancia = np.sum(distancias)
    return distancia

#Funcion auxiliar para calcular el numero de alumnos que tiene cada materia.
def contador_inscritos(alumnos,n_materias):
    inscritos = np.zeros(n_materias)
    alumnado = np.size(alumnos[:,0])
    materiado = np.size(alumnos[0,:])
    for a in range(alumnado):
        for b in range(materiado):
            materia = int(alumnos[a,b])
            if materia==0:
                break
            else:
                inscritos[materia-1] = inscritos[materia-1] + 1
    return inscritos

#######################################
#MATERIAS-HORARIOS
#######################################

#CARGA DE DATOS
def generador1(archivo,alumnosnpy):
    materias_exc = (pd.read_excel(archivo, sheet_name='Materias')).as_matrix()
    salones_exc = (pd.read_excel(archivo, sheet_name='Salones')).as_matrix()
    alumnos = np.load(alumnosnpy)
    n_salones = len(salones_exc[:,0])  #Salones.
    n_materias = len(materias_exc[:,0])  #Materias.
    n_alumnos = len(alumnos[:,0]) #Alumnos.
    n_dias = 5 #Dias.
    inscritos = contador_inscritos(alumnos,n_materias)
    necesario_mat = np.zeros((n_materias,4))
    for g in range(n_materias):
        necesario_mat[g,0] = materias_exc[g,5]
        necesario_mat[g,1] = materias_exc[g,6]
        necesario_mat[g,2] = materias_exc[g,8]
        necesario_mat[g,3] = inscritos[g]
    salones = np.zeros((n_salones,2))
    for a in range(n_salones):
        salones[a,0] = salones_exc[a,2]
        salones[a,1] = salones_exc[a,3]
    alumnos = np.load(alumnosnpy)
    return necesario_mat, salones, alumnos

#FUNCION OBJETIVO
#####################################
#HARD CONSTRAINTS.
#####################################

#En cada hora, los alumnos solo toman una clase (no hay empalme).
def hard_1(alumnos_horas):
```

```
    contador = 0
    n_alumnos = np.size(alumnos_horas[:,0,0])
    n_horas = np.size(alumnos_horas[0,:,0])
    for x in range(n_alumnos):
        for y in range(n_horas):
            for z in range(5):
                value = alumnos_horas[x,y,z]
                if value<0:
                    contador = contador + abs(value)
                else:
                    continue
    return contador

#Cada materia se asigna a un salon adecuado en tipo y en capacidad.
def hard_2(mat_salon_2,salones,necesario_mat):
    contador = 0
    num_m = np.size(mat_salon_2)
    for a in range(num_m):
        alumni = int(necesario_mat[a,3])
        saloni = int(mat_salon_2[a])-1
        saloni_tipo = salones[saloni,1]
        materia_tipo = necesario_mat[a,0]
        capacidad = salones[saloni,0]
        check_tipo = (materia_tipo==saloni_tipo)
        check_capacidad = (alumni<=capacidad)
        contador = contador+(1-check_capacidad)+(1-check_tipo)
    return contador

#En cada hora solo se asigna una materia por salon.
def hard_3(mat_hora,mat_dia):
    contador = 0
    num_d = 5
    num_h = np.size(mat_hora[0,:])
    for d in range(num_d):
        mat_dadas = [i for i, e in enumerate(mat_dia[:,d]) if e!=0]
        for h in range(num_h):
            lista = [mat_hora[j,h] for j in mat_dadas]
            check = [item for item, count
            in collections.Counter(lista).items() if count > 1 and item!=0]
            if len(check)>0:
                contador = contador + 1
                continue
            else:
                continue
    return contador

#Las materias se dan durante las horas y dias necesarios.
#Por construccion de las soluciones, esta condicion se cumple por defecto.


##########################################################
#SOFT CONSTRAINS
##########################################################

#Los alumnos deben de tener el menor numero de horas libres posibles.
```

```
def soft_1(alumnos_horas):
    contador = 0
    num_d = 5
    num_a = np.size(alumnos_horas[:,0,0])
    for a in range(num_a):
        for b in range(num_d):
            lista = alumnos_horas[a,:,b]
            horas_libres = contador_horas(lista)
            contador = contador + horas_libres
    return contador


#########################################################
#OBJECTIVE FUNCTION
#########################################################
def objetivo(solu, hard_coef,necesario_mat,salones):
    alumnos_horas = solu[3]
    mat_salon = solu[1]
    mat_salon_2 = solu[0]
    mat_dia = solu[2]
    hardo_1 = hard_1(alumnos_horas)
    hardo_2 = hard_2(mat_salon_2,salones,necesario_mat)
    hardo_3 = hard_3(mat_salon, mat_dia)
    soft = soft_1(alumnos_horas)
    hard = hardo_1+hardo_2+hardo_3
    value = hard_coef*hard + soft
    return value

#########################################################################################
#########################################################################################
#########################################################################################




#GENERAR SOLUCION INICIAL
#########################################################################################

#Genera la estructura de mat_salon
def inicial_matsalon(necesario_mat,salones):
    n_materias = np.size(necesario_mat[:,0])
    n_salones = np.size(salones[:,0])
    materia_salon = np.zeros(n_materias)
    for a in range(n_materias):
        asigna = np.random.randint(n_salones)+1
        materia_salon[a] = int(asigna)
    return materia_salon

#Genera la estructura de mat_horarios
def inicial_mathorario(necesario_mat,n_horas,materia_salon):
    n_materias = np.size(necesario_mat[:,0])
    materia_horas = np.zeros((n_materias,n_horas))
    for m in range(n_materias):
        continuas = int(necesario_mat[m,1])
```

```
        asigna = np.random.choice(n_horas-2)
        for i in range(continuas):
            toca = asigna+i
            salon = materia_salon[m]
            materia_horas[m,toca] = salon
    return materia_horas

#Genera la estructura de mat_dias
def inicial_matdias(necesario_mat):
    n_materias = np.size(necesario_mat[:,0])
    materia_dias = np.zeros((n_materias,5))
    for m in range(n_materias):
        dias = int(necesario_mat[m,2])
        if dias==5:
            materia_dias[m,:] = [1,1,1,1,1]
        else:
            dias_toca = np.random.choice(5,dias,replace=False)
            for d in range(dias):
                dio = int(dias_toca[d])
                materia_dias[m,dio] = 1
    return materia_dias

#A partir de las estructuras anteriores, genera alumnos_horas.
def genera_alumnoshoras(mat_horarios,mat_dias,alumnos):
    n_alumnos = np.size(alumnos[:,0])
    n_horas = np.size(mat_horarios[0,:])
    alumnos_horas= np.zeros((n_alumnos,n_horas,5))
    n_materias = np.size(mat_horarios[:,0])
    for a in range(n_alumnos):
        lleva = [alumnos[a,:][i] for i, e in enumerate(alumnos[a,:]) if e!=0]
        for l in range(np.size(lleva)):
            materia = int(lleva[l])-1
            horario_materia = mat_horarios[materia]
            dias_materia = [i for i, e in enumerate(mat_dias[materia,:]) if e!=0]
            for g in range(np.size(dias_materia)):
                day = int(dias_materia[g])
                alumnos_horas[a,:,day] =
                suma_detector(alumnos_horas[a,:,day],horario_materia)
    return alumnos_horas

#Genera una sola solucion inicial
def sol_inicial(necesario_mat,salones,alumnos,n_horas):
    n_alumnos = np.size(alumnos[0,:])
    primero = inicial_matsalon(necesario_mat,salones)
    segundo = inicial_mathorario(necesario_mat,n_horas,primero)
    tercero = inicial_matdias(necesario_mat)
    cuarto = genera_alumnoshoras(segundo,tercero,alumnos)
    solucion = [primero,segundo,tercero,cuarto]
    return solucion

#
################################################################################
################################################################################
################################################################################
```

```
#Metodos de VNS
##############################################################################
##############################################################################
##############################################################################

#Primera vecindad, intercambia los salones de dos materias.
def vecindad_1(x,alumnos):
    num_m = np.size(x[0])
    num_h = np.size(x[1][0,:])
    mat_salon = x[0]
    mat_horario = x[1]
    mat_salon_new = np.array(mat_salon)
    intercambio = np.random.choice(num_m,2,replace=False)
    entry1 = int(intercambio[0])
    entry2 = int(intercambio[1])
    mat_salon_new[entry1] = mat_salon[entry2]
    mat_salon_new[entry2] = mat_salon[entry1]
    mat_horario_new = np.array(mat_horario)
    for m in range(num_h):
        check = mat_horario_new[entry1,m]
        if check!=0:
            mat_horario_new[entry1,m] = mat_salon_new[entry1]
        else:
            continue
    for n in range(num_h):
        checko = mat_horario_new[entry2,n]
        if checko!=0:
            mat_horario_new[entry2,n] = mat_salon_new[entry2]
        else:
            continue
    matdia_new = np.array(x[2])
    alumno_new = genera_alumnoshoras(mat_horario_new,matdia_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,matdia_new,alumno_new]
    return vecino

#Segunda vecindad. Para una materia que se da menos de 5 dias, intercambia dos entradas.
def vecindad_2(x,alumnos):
    mat_dias = x[2]
    num_m = np.size(mat_dias[:,0])
    sumus = [sum(mat_dias[i,:]) for i in range(num_m)]
    mat_dias_new = np.array(mat_dias)
    lista = [i for i in range(num_m) if sumus[i]<5]
    intercambio = np.random.choice(lista)
    materio = int(intercambio)
    lista_ceros = [i for i in range(5) if mat_dias[materio,i]==0]
    lista_unos = [i for i in range(5) if mat_dias[materio,i]==1]
    entry1 = int(np.random.choice(lista_ceros))
    entry2 = int(np.random.choice(lista_unos))
    mat_dias_new[materio,entry1] = mat_dias[materio,entry2]
    mat_dias_new[materio,entry2] = mat_dias[materio,entry1]
    mat_horario_new = np.array(x[1])
    mat_salon_new = np.array(x[0])
```

```
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Tercera vecindad. Para una materia intercambia aleatoriamente su centro de horario.
def vecindad_3(x,alumnos):
    mat_salon_new = np.array(x[0])
    mat_dias_new = np.array(x[2])
    mat_horario = x[1]
    num_m = np.size(mat_horario[:,0])
    num_h = np.size(mat_horario[0,:])
    mat_horario_new = np.array(mat_horario)
    materio = int(np.random.choice(num_m))
    salon_materio = x[0][materio]
    horas_dadas = [i for i in range(num_h) if mat_horario[materio,i]!=0]
    continuas = np.size(horas_dadas)
    posibles_centros = [r for r in range(num_h+1-continuas)]
    nuevo_centro = int(np.random.choice(posibles_centros))
    nuevo_horario = np.zeros(num_h)
    for g in range(continuas):
        nuevo_horario[nuevo_centro+g] = salon_materio
    mat_horario_new[materio,:] = nuevo_horario
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Cuarta vecindad. Para dos materias de una hora, intercambia sus horarios.
def vecindad_4(x,alumnos):
    mat_salon_new = np.array(x[0])
    mat_dias_new = np.array(x[2])
    mat_horario = x[1]
    num_m = np.size(mat_horario[:,0])
    num_h = np.size(mat_horario[0,:])
    mat_horario_new = np.array(mat_horario)
    condition = False
    mat1 = int
    mat2 = int
    while condition==False:
        par = np.random.choice(num_m,2,replace=False)
        mat1 = int(par[0])
        mat2 = int(par[1])
        horas1 = np.sum([1 for u in range(num_h) if mat_horario[mat1,u]!=0])
        horas2 = np.sum([1 for u in range(num_h) if mat_horario[mat2,u]!=0])
        if horas1==1 and horas2==1:
            condition = True
        else:
            continue
    salo1 = x[0][mat1]
    salo2 = x[0][mat2]
    cuando1 = [i for i in range(num_h) if mat_horario[mat1,i]!=0][0]
    cuando2 = [i for i in range(num_h) if mat_horario[mat2,i]!=0][0]
    mat_horario_new[mat1,cuando1] = 0
    mat_horario_new[mat1,cuando2] = salo1
    mat_horario_new[mat2,cuando2] = 0
    mat_horario_new[mat2,cuando1] = salo2
```

```
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Selecciona entre todas las vecindades definidas y regresa un vecino correspondiente.
def vecindad_select(inicial,alumnos,n):
    if n==1:
        value = vecindad_1(inicial,alumnos)
    elif n==2:
        value = vecindad_2(inicial,alumnos)
    elif n==3:
        value = vecindad_3(inicial,alumnos)
    else:
        value = vecindad_4(inicial,alumnos)
    return value

#Metodo descendente aleatorio con opcion para hacerlo secuencial.
def rna(xo,stop,hard_coef,salones,necesario_mat,alumnos, seque=None):
    if seque==None:
        intentos = 0
        x_mejor = xo
        f_mejor = objetivo(x_mejor, hard_coef,necesario_mat,salones)
        while intentos<stop:
            metodo = np.random.choice(4)+1
            x_prueba = vecindad_select(x_mejor,alumnos,metodo)
            f_prueba = objetivo(x_prueba, hard_coef,necesario_mat,salones)
            if f_prueba<f_mejor:
                x_mejor = x_prueba
                f_mejor = f_prueba
                intentos = 0
            elif f_prueba==f_mejor:
                x_mejor = x_prueba
                f_mejor = f_prueba
                intentos = intentos+1
            else:
                intentos = intentos+1
    else:
        metodo = seque
        intentos = 0
        x_mejor = xo
        f_mejor = objetivo(x_mejor, hard_coef,necesario_mat,salones)
        while intentos<stop:
            x_prueba = vecindad_select(x_mejor,alumnos,metodo)
            f_prueba = objetivo(x_prueba, hard_coef,necesario_mat,salones)
            if f_prueba<f_mejor:
                x_mejor = x_prueba
                f_mejor = f_prueba
                intentos = 0
            elif f_prueba==f_mejor:
                x_mejor = x_prueba
                f_mejor = f_prueba
                intentos = intentos+1
            else:
                intentos = intentos+1
    return x_mejor
```

```
#Metodo VNS (por defecto es no secuencial, se necesita activar aparte)
def vns(xo,pasos,stop,hard_coef,necesario_mat,salones,alumnos,seque=None):
    if seque==None:
        x_mejor = xo
        f_mejor = objetivo(x_mejor, hard_coef,necesario_mat,salones)
        for i in range(pasos):
            k = 1
            while k<=4:
                x_vecino = vecindad_select(x_mejor,alumnos,k)
                x_prueba = rna(x_vecino,stop,hard_coef,salones,necesario_mat,alumnos)
                f_prueba = objetivo(x_prueba, hard_coef,necesario_mat,salones)
                if f_prueba==f_mejor:
                    x_mejor = x_prueba
                    f_mejor = f_prueba
                    k = k+1
                elif f_prueba<f_mejor:
                    x_mejor = x_prueba
                    f_mejor = f_prueba
                    k = 1
                else:
                    k = k+1
    else:
        x_mejor = xo
        f_mejor = objetivo(x_mejor, hard_coef,necesario_mat,salones)
        for i in range(pasos):
            k = 1
            while k<=4:
                x_vecino = vecindad_select(x_mejor,alumnos,k)
                x_prueba = rna(x_vecino,stop,hard_coef,salones,necesario_mat,alumnos,k)
                f_prueba = objetivo(x_prueba, hard_coef,necesario_mat,salones)
                if f_prueba==f_mejor:
                    x_mejor = x_prueba
                    f_mejor = f_prueba
                    k = k+1
                elif f_prueba<f_mejor:
                    x_mejor = x_prueba
                    f_mejor = f_prueba
                    k = 1
                else:
                    k = k+1

    return x_mejor
##############################################################################
##############################################################################


#Primera vecindad gals, intercambia los salones de dos materias.
def vegals_1(x,alumnos,a,b):
    num_m = np.size(x[0])
    num_h = np.size(x[1][0,:])
    mat_salon = x[0]
    mat_horario = x[1]
    mat_salon_new = np.array(mat_salon)
    intercambio = [a,b]
```

```
    entry1 = int(intercambio[0])
    entry2 = int(intercambio[1])
    mat_salon_new[entry1] = mat_salon[entry2]
    mat_salon_new[entry2] = mat_salon[entry1]
    mat_horario_new = np.array(mat_horario)
    for m in range(num_h):
        check = mat_horario_new[entry1,m]
        if check!=0:
            mat_horario_new[entry1,m] = mat_salon_new[entry1]
        else:
            continue
    for n in range(num_h):
        checko = mat_horario_new[entry2,n]
        if checko!=0:
            mat_horario_new[entry2,n] = mat_salon_new[entry2]
        else:
            continue
    matdia_new = np.array(x[2])
    alumno_new = genera_alumnoshoras(mat_horario_new,matdia_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,matdia_new,alumno_new]
    return vecino

#Segunda vecindad gals. Para una materia
#que se da menos de 5 dias, intercambia dos entradas.
def vegals_2(x,alumnos,e,a,b):
    mat_dias = x[2]
    num_m = np.size(mat_dias[:,0])
    sumus = [sum(mat_dias[i,:]) for i in range(num_m)]
    mat_dias_new = np.array(mat_dias)
    intercambio = e
    materio = int(intercambio)
    entry1 = a
    entry2 = b
    mat_dias_new[materio,entry1] = mat_dias[materio,entry2]
    mat_dias_new[materio,entry2] = mat_dias[materio,entry1]
    mat_horario_new = np.array(x[1])
    mat_salon_new = np.array(x[0])
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Tercera vecindad. Para una materia intercambia aleatoriamente su centro de horario.
def vegals_3(x,alumnos,e,a,continuas):
    mat_salon_new = np.array(x[0])
    mat_dias_new = np.array(x[2])
    mat_horario = x[1]
    num_m = np.size(mat_horario[:,0])
    num_h = np.size(mat_horario[0,:])
    mat_horario_new = np.array(mat_horario)
    materio = e
    salon_materio = x[0][materio]
    nuevo_centro = a
    nuevo_horario = np.zeros(num_h)
    for g in range(continuas):
        nuevo_horario[nuevo_centro+g] = salon_materio
```

```
    mat_horario_new[materio,:] = nuevo_horario
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Cuarta vecindad. Para dos materias de una hora, intercambia sus horarios.
def vegals_4(x,alumnos,a,b):
    mat_salon_new = np.array(x[0])
    mat_dias_new = np.array(x[2])
    mat_horario = x[1]
    num_m = np.size(mat_horario[:,0])
    num_h = np.size(mat_horario[0,:])
    mat_horario_new = np.array(mat_horario)
    condition = False
    mat1 = a
    mat2 = b
    salo1 = x[0][mat1]
    salo2 = x[0][mat2]
    cuando1 = [i for i in range(num_h) if mat_horario[mat1,i]!=0][0]
    cuando2 = [i for i in range(num_h) if mat_horario[mat2,i]!=0][0]
    mat_horario_new[mat1,cuando1] = 0
    mat_horario_new[mat1,cuando2] = salo1
    mat_horario_new[mat2,cuando2] = 0
    mat_horario_new[mat2,cuando1] = salo2
    alumno_new = genera_alumnoshoras(mat_horario_new,mat_dias_new,alumnos)
    vecino = [mat_salon_new,mat_horario_new,mat_dias_new,alumno_new]
    return vecino

#Selecciona entre todas las vecindades definidas y regresa un vecino correspondiente.
def vegals_select(inicial,alumnos,n,a,b,c=None):
    if n==1:
        if a==b:
            value = inicial
        else:
            value = vegals_1(inicial,alumnos,a,b)
    elif n==2:
        value = vegals_2(inicial,alumnos,a,b,c)
    elif n==3:
        value = vegals_3(inicial,alumnos,a,b,c)
    else:
        if a==b:
            value = inicial
        else:
            value = vegals_4(inicial,alumnos,a,b)
    return value

#Funcion para delta evaluar una solucion.
def f_delta(solu,hard_coef,necesario_mat,salones):
    alumnos_horas = solu[3]
    mat_salon = solu[1]
    mat_salon_2 = solu[0]
    mat_dia = solu[2]
    hardo_1 = hard_1(alumnos_horas)
    hardo_2 = hard_2(mat_salon_2,salones,necesario_mat)
    hardo_3 = hard_3(mat_salon, mat_dia)
```

```
    hard = hardo_1+hardo_2+hardo_3
    value = hard
    return value

#Dado una solucion inicial y un evento, evalua todos los vecinos posibles.
def vegals(xo,hard_coef,alumnos,necesario_mat,mat_salones,e,f_delto):
    mat_horario = xo[1]
    num_h = np.size(mat_horario[0,:])
    mat_dias = xo[2]
    n_materias = np.size(mat_horario[:,0])
    xn = xo
    fn = f_delto(xn,hard_coef,necesario_mat,mat_salones)
    find = False
    marker = 0
    #Vecindad 1
    for b in range(e+1,n_materias):
        xp = vegals_select(xn,alumnos,1,e,b)
        fp = f_delto(xp,hard_coef,necesario_mat,mat_salones)
        if fp<fn:
            xn = xp
            fn = fp
            find = True
            marker = 1
            break
    #Vecindad 2
    if find==False:
        lista_ceros = [i for i in range(5) if mat_dias[e,i]==0]
        lista_unos = [i for i in range(5) if mat_dias[e,i]==1]
        for c in lista_unos:
            for d in lista_ceros:
                xp = vegals_select(xn,alumnos,2,e,int(c),int(d))
                fp = f_delto(xp,hard_coef,necesario_mat,mat_salones)
                if fp<fn:
                    xn = xp
                    fn = fp
                    find = True
                    marker = 1
                    break
            if find==False:
                continue
            else:
                break
    #Vecindad 3
    if find==False:
        horas_dadas = [i for i in range(num_h) if mat_horario[e,i]!=0]
        continuas = np.size(horas_dadas)
        posibles_centros = [r for r in range(num_h+1-continuas)]
        for f in range(np.size(posibles_centros)):
            centro_p = int(posibles_centros[f])
            xp = vegals_select(xn,alumnos,3,e,centro_p,continuas)
            fp = f_delto(xp,hard_coef,necesario_mat,mat_salones)
            if fp<fn:
                xn = xp
                fn = fp
                find = True
```

```
                marker = 1
                break
    #Vecindad 4
    if find==False:
        horas1 = np.sum([1 for u in range(num_h) if mat_horario[e,u]!=0])
        if horas1==1:
            for g in range(e+1,n_materias):
                horas2 = np.sum([1 for u in range(num_h) if mat_horario[g,u]!=0])
                if horas2==1:
                    xp = vegals_select(xn,alumnos,4,e,g)
                    fp = f_delto(xp,hard_coef,necesario_mat,mat_salones)
                    if fp<fn:
                        xn = xp
                        fn = fp
                        find = True
                        marker = 1
                        break

    return [xn,fn,marker]

#Seccion de busqueda local para GALS de construccion.
def gals_cons(xo,hard_coef,necesario_mat,salones,alumnos,namae):
    xn = xo
    fn = f_delta(xn,hard_coef,necesario_mat,salones)
    n_materias = np.size(necesario_mat[:,0])
    i = 0
    while i<(n_materias-1):
        if fn==0:
            i = n_materias
        else:
            buscar = vegals(xn,hard_coef,alumnos,necesario_mat,salones,i,f_delta)
            if buscar[2]==1:
                xn = buscar[0]
                fn = buscar[1]
                i = 0
            else:
                i = i+1
    return xn

#Seccion de busqueda local para GALS de mejoramiento.
def gals_impro(xo,hard_coef,necesario_mat,salones,alumnos):
    xn = xo
    fn = objetivo(xn,hard_coef,necesario_mat,salones)
    n_materias = np.size(necesario_mat[:,0])
    i = 0
    while i<(n_materias-1):
        if fn==0:
            i = n_materias
        else:
            buscar = vegals(xn,hard_coef,alumnos,necesario_mat,salones,i,objetivo)
            if buscar[2]==1:
                xn = buscar[0]
                fn = buscar[1]
                i = 0
            else:
```

```
                i = i+1
    return xn

#Busqueda local para GALS.
def gals_local(shared,hard_coef,necesario_mat,salones,alumnos,z,j):
    zeta = z
    size_all = len(shared)
    while z*j+zeta>size_all:
        zeta = zeta-1
    if zeta==0:
        pass
    else:
        for i in range(zeta):
            entry = i+j*z
            loco = shared[entry]
            construye = gals_cons(loco,hard_coef,necesario_mat,salones,alumnos)
            mejora = gals_impro(construye,hard_coef,necesario_mat,salones,alumnos)
            shared[entry] = mejora

#Para hacer GALS en paralelo sobre una poblacion.
def gals_poblacion(poblacion,hard_coef,necesario_mat,salones,alumnos):
    num_cores = mp.cpu_count()
    pop = len(poblacion)
    manager = mp.Manager()
    shared = manager.list()
    for i in range(pop):
        loki = poblacion[i]
        shared.append(loki)
    processes = []
    z = int(math.ceil((1.0*pop)/(num_cores)))
    for j in range(num_cores):
        pj = mp.Process(target=gals_local, args=(shared,
        hard_coef,necesario_mat,salones,alumnos,z,j))
        processes.append(pj)
        pj.start()
    for j in range(num_cores):
        processes[j].join()

    return shared

#GALS (genetic algorithm with local search) (version nuestra)
#################################################################################
def gals(hard_coef,necesario_mat,salones,alumnos,n_horas,
n_generaciones,n_poblacion,n_pool,proba_m,local_pasos,local_stop):
    poblacion = []
    for a in range(n_poblacion):
        solu = sol_inicial(necesario_mat,salones,alumnos,n_horas)
        solu = vns(solu,local_pasos,
        local_stop,hard_coef,necesario_mat,salones,alumnos,1)
        poblacion.append(solu)
    poblacion, poblacion_f = acomodar(poblacion,hard_coef,necesario_mat,salones)
    mejor = [poblacion[0],poblacion_f[0]]
    for r in range(n_generaciones):
        padres = matchpool(poblacion,poblacion_f,n_pool)
        hijos = []
```

```
        for b in range(int(n_poblacion/2.0)):
            pareja = np.random.choice(n_pool,size=2,replace=False)
            mama = padres[int(pareja[0])]
            papa = padres[int(pareja[1])]
            hijo1, hijo2 = cruza(mama,papa,alumnos)
            hijo1 = mutacion(hijo1,proba_m,alumnos)
            hijo2 = mutacion(hijo2,proba_m,alumnos)
            hijo1 = vns(hijo1,local_pasos,local_stop,hard_coef,
            necesario_mat,salones,alumnos,1)
            hijo2 = vns(hijo2,local_pasos,local_stop,hard_coef,
            necesario_mat,salones,alumnos,1)
            hijos.append(hijo1)
            hijos.append(hijo2)
        poblacion, poblacion_f = acomodar(hijos,hard_coef,necesario_mat,salones)
        mejor_local = [poblacion[0],poblacion_f[0]]
        if mejor_local[1]<mejor[1]:
            mejor = mejor_local
        else:
            continue
    return mejor

#GALS (genetic algorithm with local search) (version paper)
#############################################################################
def gals_paper(hard_coef,necesario_mat,salones,alumnos,
n_horas,n_generaciones,n_poblacion,n_pool,proba_m,
pob_inicial=None,timo=None, numo=None,alum_global=None):
    if timo==None:
        if pob_inicial==None:
            print('Empieza_nha')
            starto = time.time()
            poblacion = []
            for a in range(n_poblacion):
                solu = sol_inicial(necesario_mat,salones,alumnos,n_horas)
                poblacion.append(solu)
            poblacion = gals_poblacion(poblacion,
            hard_coef,necesario_mat,salones,alumnos)
            poblacion, poblacion_f = acomodar(poblacion,hard_coef,necesario_mat,salones)
            mejor = [poblacion[0],poblacion_f[0]]
            #SEGUIMIENTO
            tops = []
            tops.append([(time.time()-starto)/60,mejor[1]])
            SORA = 0
            vuelta = regreso(mejor[0],alum_global)
            alumno_excel(vuelta,0,'matematico_0_0_nha')
            alumno_excel(vuelta,41,'fisico_0_0_nha')
            alumno_excel(vuelta,81,'actuario_0_0_nha')
            horario_excel(vuelta,'horario_0_0_nha')
            #SEGUIMIENTO
            for r in range(n_generaciones):
                SORA = SORA+1
                padres = matchpool(poblacion,poblacion_f,n_pool)
                hijos = []
                for b in range(int(n_poblacion/2.0)):
                    pareja = np.random.choice(n_pool,size=2,replace=False)
                    mama = padres[int(pareja[0])]
```

```
                papa = padres[int(pareja[1])]
                hijo1, hijo2 = cruza(mama,papa,alumnos)
                hijo1 = mutacion(hijo1,proba_m,alumnos)
                hijo2 = mutacion(hijo2,proba_m,alumnos)
                hijos.append(hijo1)
                hijos.append(hijo2)
            if SORA%1000==0:
                hijos = gals_poblacion(hijos,hard_coef,
                necesario_mat,salones,alumnos)
            else:
                pass
            poblacion, poblacion_f = acomodar(hijos,hard_coef,necesario_mat,salones)
            mejor_local = [poblacion[0],poblacion_f[0]]
            if mejor_local[1]<mejor[1]:
                mejor = mejor_local
                poblacion[0] = mejor[0]
            else:
                poblacion[0] = mejor[0]
                poblacion_f[0] = mejor[1]
            #SEGUIMIENTO
            tops.append([(time.time()-starto)/60,mejor[1]])
            if SORA%200==0:
                vuelta = regreso(mejor[0],alum_global)
                alumno_excel(vuelta,0,'matematico_0_'+str(SORA)+'_nha')
                alumno_excel(vuelta,41,'fisico_0_'+str(SORA)+'_nha')
                alumno_excel(vuelta,81,'actuario_0_'+str(SORA)+'_nha')
                horario_excel(vuelta,'horario_0_'+str(SORA)+'_nha')
            else:
                pass
            #SEGUIMIENTO
        tamano = len(tops)
        grafo = np.zeros((2,tamano))
        for ja in range(tamano):
            grafo[0,ja] = tops[ja][0]
            grafo[1,ja] = tops[ja][1]
        np.save('nha_0.npy',grafo)
        return poblacion
    else:
        print('Empieza_Grupo_'+str(numo))
        starto = time.time()
        poblacion = np.array(pob_inicial)
        poblacion, poblacion_f = acomodar(poblacion,hard_coef,necesario_mat,salones)
        mejor = [poblacion[0],poblacion_f[0]]
        #SEGUIMIENTO
        tops = []
        tops.append([(time.time()-starto)/60,mejor[1]])
        SORA = 0
        #SEGUIMIENTO
        for r in range(n_generaciones):
            SORA = SORA+1
            padres = matchpool(poblacion,poblacion_f,n_pool)
            hijos = []
            for b in range(int(n_poblacion/2.0)):
                pareja = np.random.choice(n_pool,size=2,replace=False)
                mama = padres[int(pareja[0])]
```

```
                papa = padres[int(pareja[1])]
                hijo1, hijo2 = cruza(mama,papa,alumnos)
                hijo1 = mutacion(hijo1,proba_m,alumnos)
                hijo2 = mutacion(hijo2,proba_m,alumnos)
                hijos.append(hijo1)
                hijos.append(hijo2)
            if SORA%1000==0:
                hijos = gals_poblacion(hijos,
                hard_coef,necesario_mat,salones,alumnos)
            else:
                pass
            poblacion, poblacion_f = acomodar(hijos,hard_coef,necesario_mat,salones)
            mejor_local = [poblacion[0],poblacion_f[0]]
            if mejor_local[1]<mejor[1]:
                mejor = mejor_local
                poblacion[0] = mejor[0]
            else:
                poblacion[0] = mejor[0]
                poblacion_f[0] = mejor[1]
            #SEGUIMIENTO
            tops.append([(time.time()-starto)/60,mejor[1]])
            if SORA%200==0:
                vuelta = regreso(mejor[0],alum_global)
                alumno_excel(vuelta,0,'matematico_'+str(numo)+'_'+str(SORA)+'_nha')
                alumno_excel(vuelta,41,'fisico_'+str(numo)+'_'+str(SORA)+'_nha')
                alumno_excel(vuelta,81,'actuario_'+str(numo)+'_'+str(SORA)+'_nha')
                horario_excel(vuelta,'horario_'+str(numo)+'_'+str(SORA)+'_nha')
            else:
                pass
            #SEGUIMIENTO
        tamano = len(tops)
        grafo = np.zeros((2,tamano))
        for ja in range(tamano):
            grafo[0,ja] = tops[ja][0]
            grafo[1,ja] = tops[ja][1]
        np.save('nha_'+str(numo)+'.npy',grafo)
        return poblacion
else:
    starto = time.time()
    poblacion = []
    print('Empieza_gals')
    for a in range(n_poblacion):
        solu = sol_inicial(necesario_mat,salones,alumnos,n_horas)
        poblacion.append(solu)
    poblacion = gals_poblacion(poblacion,hard_coef,necesario_mat,salones,alumnos)
    poblacion, poblacion_f = acomodar(poblacion,hard_coef,necesario_mat,salones)
    mejor = [poblacion[0],poblacion_f[0]]
    #SEGUIMIENTO
    tops = []
    tops.append([(time.time()-starto)/60,mejor[1]])
    SORA = 0
    alumno_excel(mejor[0],0,'matematico_0_gals')
    alumno_excel(mejor[0],41,'fisico_0_gals')
    alumno_excel(mejor[0],81,'actuario_0_gals')
    horario_excel(mejor[0],'horario_0_gals')
```

```
        #SEGUIMIENTO
        while (time.time()-starto)/60<timo and mejor[1]>0:# and SORA<=n_generaciones:
            SORA = SORA+1
#            print('Generacion'+str(SORA),mejor[1])
            padres = matchpool(poblacion,poblacion_f,n_pool)
            hijos = []
            for b in range(int(n_poblacion/2.0)):
                pareja = np.random.choice(n_pool,size=2,replace=False)
                mama = padres[int(pareja[0])]
                papa = padres[int(pareja[1])]
                hijo1, hijo2 = cruza(mama,papa,alumnos)
                hijo1 = mutacion(hijo1,proba_m,alumnos)
                hijo2 = mutacion(hijo2,proba_m,alumnos)
                hijos.append(hijo1)
                hijos.append(hijo2)
            if SORA%1000==0:
                hijos = gals_poblacion(hijos,hard_coef,necesario_mat,salones,alumnos)
            else:
                pass
            poblacion, poblacion_f = acomodar(hijos,hard_coef,necesario_mat,salones)
            mejor_local = [poblacion[0],poblacion_f[0]]
            if mejor_local[1]<mejor[1]:
                mejor = mejor_local
                poblacion[0] = mejor[0]
            else:
                poblacion[0] = mejor[0]
                poblacion_f[0] = mejor[1]
            #SEGUIMIENTO
            tops.append([(time.time()-starto)/60,mejor[1]])
            if SORA%200==0:
                alumno_excel(mejor[0],0,'matematico_'+str(SORA)+'_gals')
                alumno_excel(mejor[0],41,'fisico_'+str(SORA)+'_gals')
                alumno_excel(mejor[0],81,'actuario_'+str(SORA)+'_gals')
                horario_excel(mejor[0],'horario_'+str(SORA)+'_gals')
            else:
                pass
            #SEGUIMIENTO
        tamano = len(tops)
        grafo = np.zeros((2,tamano))
        for ja in range(tamano):
            grafo[0,ja] = tops[ja][0]
            grafo[1,ja] = tops[ja][1]
        np.save('gals.npy',grafo)
        return poblacion
################################################################################
################################################################################



#GALS con agrupacion
################################################################################


#Elimina un numero de un nparray.
def remover(x,n):
```

```
    index = np.argwhere(x==n)
    y = np.delete(x,index)
    return y

#Dado un grupo G de alumnos, regresa su asociado E(G).
def grupoE(alumnos,G):
    E = np.array([])
    for a in G:
        checa = alumnos[a,:]
        E = np.union1d(E,checa)
    E = remover(E,0)
    return E

#Dado un evento e, crea el grupo G asociado G(e).
def grupoG(disponibles,alumnos,e):
    G = []
    dispo_nuevos = np.array(disponibles)
    for a in disponibles:
        checa = alumnos[a,:]
        if (e+1) in checa:
            G.append(a)
            dispo_nuevos = remover(dispo_nuevos,a)
    return G, dispo_nuevos

#Crea la particion inicial.
def particion_inicial(alumnos):
    n_alumnos = np.size(alumnos[:,0])
    alumnos_dispo = [i for i  in range(n_alumnos)]
    grupos = []
    E_grupos = []
    e = 0
    while np.size(alumnos_dispo)>0:
        nuevo_G, alumnos_dispo = grupoG(alumnos_dispo,alumnos,e)
        if len(nuevo_G)==0:
            e = e+1
        else:
            nuevo_EG = grupoE(alumnos,nuevo_G)
            grupos.append(nuevo_G)
            E_grupos.append(nuevo_EG)
            e = e+1
    return grupos, E_grupos

#Particiona un grupo G
def reduceG(alumnos,G):
    n_alumnos = np.size(G)
    G_padre = np.array(G)
    G_partido = []
    E_partido = []
    inter = interG(G_padre,alumnos)
    e = 0
    if np.size(G)>1:
        while np.size(G_padre)>1:
            if (e+1) not in inter:
                G_hijo, G_padre = grupoG(G_padre,alumnos,e)
                if np.size(G_hijo)>0:
```

```
                    nuevo_EG = grupoE(alumnos,G_hijo)
                    G_partido.append(G_hijo)
                    E_partido.append(nuevo_EG)
                    inter = interG(G_padre,alumnos)
                    e = e+1
                else:
                    e = e+1
            else:
                e = e+1
        G_ultimo = [G_padre[0]]
        G_partido.append(G_ultimo)
        E_partido.append(grupoE(alumnos,G_ultimo))
    else:
        G_ultimo = [G_padre[0]]
        G_partido.append(G_ultimo)
        E_partido.append(grupoE(alumnos,G_ultimo))
    return G_partido, E_partido

#Dado un grupo G lo particiona hasta que los sub E tengan el tamano deseado (check+nga).
def agrupar(alumnos,G,E,maximo):
    G_n = G
    E_n = E
    tamano_n = np.size(G_n)
    index = 0
    while index<tamano_n:
        order = np.size(E_n[index])
        if order<=maximo:
            index = index+1
        else:
            G_k, E_k = reduceG(alumnos,G_n[index])
            del G_n[index]
            del E_n[index]
            tamanio = np.size(G_k)
            tamano_n = tamano_n-1+tamanio
            for j in range(tamanio):
                G_n.insert(index+j,G_k[j])
                E_n.insert(index+j,E_k[j])
    return G_n, E_n

#New grouping algorithm. VERSION VIEJA
#def agrupar(alumnos,G,E,maximo):
#    n_grupos = len(G)
#    G_f = []
#    E_f = []
#    for i in range(n_grupos):
#        G_n, E_n = checkG(alumnos,G[i], E[i], maximo)
#        tamanio = np.size(G_n)
#        for j in range(tamanio):
#            G_f.append(G_n[j])
#            E_f.append(E_n[j])
#    return G_f, E_f

#Crea una instancia modificada de alumnos dada una agrupacion G.
def modificado(alumnos,grupos,maximo):
    n_grupos = np.size(grupos[0])
```

```
    g_alumnos = np.zeros((n_grupos,maximo))
    for a in range(n_grupos):
        materios = np.size(grupos[1][a])
        for b in range(materios):
            g_alumnos[a,b] = grupos[1][a][b]
    return g_alumnos

#Regresa solucion con g_alumnos a la instancia original.
def regreso(x,alumnos):
    xn_0 = np.array(x[0])
    xn_1 = np.array(x[1])
    xn_2 = np.array(x[2])
    xn_3 = genera_alumnoshoras(xn_1,xn_2,alumnos)
    xn = [xn_0,xn_1,xn_2,xn_3]
    return xn

#New hybrid algorithm.
def nha(hard_coef,necesario_mat,salones,alumnos,n_horas,
n_generaciones,n_poblacion,n_pool,proba_m,timo):
    start_time = time.time()
    maximo = 20#Primera cota es numero total de materias
    max_f = 6  #Ultima cota es numero de materias maximas por alumno
    grupo_0 = particion_inicial(alumnos)
    G_0, E_0 = grupo_0[0], grupo_0[1]
    grupos = agrupar(alumnos,G_0, E_0,maximo)
    g_alumnos = modificado(alumnos,grupos,maximo)
    poblacion = gals_paper(hard_coef,necesario_mat,salones,g_alumnos,
    n_horas,n_generaciones,n_poblacion,n_pool,proba_m,None,None,None,alumnos)
    x_mejor = poblacion[0]
    f_mejor = objetivo(x_mejor,hard_coef,necesario_mat,salones)
    numo = 1
    while f_mejor>0 and float(time.time()-start_time)/60<timo:
        if maximo>max_f:
            maximo = maximo-1
            G_n, E_n = grupos[0], grupos[1]
            grupos = agrupar(alumnos,G_n,E_n,maximo)
            g_alumnos = modificado(alumnos,grupos,maximo)
        else:
            maximo = 0
            g_alumnos = alumnos
            print(np.size(alumnos[:,0]))
        poblacion = gals_paper(hard_coef,necesario_mat,salones,g_alumnos,
        n_horas,n_generaciones,n_poblacion,n_pool,proba_m,poblacion,None,numo,alumnos)
        x_local = poblacion[0]
        f_local = objetivo(x_local,hard_coef,necesario_mat,salones)
        numo = numo + 1
        if f_local<=f_mejor:
            x_mejor = x_local
            f_mejor = f_local
    x_mejor = regreso(x_mejor,alumnos)
    return x_mejor


#############################################################################
#CONVERTIDOR DE DATOS A EXCELL
```

```
##############################################################################

#Dada una solucion, transfrma info de un alumno  a un horario en excell.
def alumno_excel(x,n,namae):
    alumnos_horas = np.array(x[3])
    mat_salon = np.array(x[0])
    horario = intercambia_info(alumnos_horas,mat_salon)[n,:,:]
    shapo = np.shape(horario)
    datos = np.zeros((shapo[0],shapo[1]+1))
    for x in range(shapo[1]):
        datos[:,x+1] = horario[:,x]
    for y in range(shapo[0]):
        datos[y,0] = 7+y
    df = pd.DataFrame(datos,columns=['hora','lu','ma','mi','ju','vi'])
    writer = pd.ExcelWriter(namae+'.xlsx', engine='xlsxwriter')
    df.to_excel(writer, sheet_name='Sheet1')
    writer.save()

#Dada una solucion, escribe un excel con los horarios de cada materia.
def horario_excel(x,namae):
    materias_exc = (pd.read_excel('entrada_horarios.xlsx',
    sheet_name='Materias')).as_matrix()
    mat_salon = x[0]
    mat_horario = x[1]
    mat_horas = intervalo(mat_horario)
    mat_dias = x[2]
    n_materias = np.size(mat_dias[:,0])
    salones = np.zeros((n_materias,1))
    lista = []
    for t in range(n_materias):
        value1 = (materias_exc[t,1],mat_horas[t,0],mat_horas[t,1],mat_dias[t,0])
        value2 = (mat_dias[t,1],mat_dias[t,2],mat_dias[t,3],mat_dias[t,4],mat_salon[t])
        value = value1+value2
        lista.append(value)
    labels = ['Materia','Inicia','Termina','Lu','Ma','Mi','Ju','Vi','Salon']
    df = pd.DataFrame.from_records(lista,columns=labels)
    writer = pd.ExcelWriter(namae+'.xlsx', engine='xlsxwriter')
    df.to_excel(writer, sheet_name='Sheet1')
    writer.save()

##############################################################################
##############################################################################
##############################################################################
##############################################################################
##############################################################################
#ASIGNACION MAESTROS-MATERIAS
##############################################################################
##############################################################################
##############################################################################
##############################################################################
##############################################################################
##############################################################################


###################
```

```
#AUXILIARES
###################

#Muta a un individuo
def xmen(inicial,datos):
    n = np.random.randint(1,5)
    n_maestros = np.size(inicial[:,0])
    n_materias = np.size(datos[0])
    if n==1:
        maistros = np.random.choice(n_maestros,2,replace=False)
        a, b = int(maistros[0]), int(maistros[1])
        value = vecindad21(inicial,a,b)
    elif n==2:
        value = vecindad22(inicial)
    elif n==3:
        maistros = np.random.choice(n_maestros,2,replace=False)
        a, b = int(maistros[0]), int(maistros[1])
        if 0 in inicial[b,:] and np.sum(inicial[a,:])>0:
            value = vecindad23(inicial,a,b)
        else:
            value = inicial
    else:
        b = int(np.random.choice(n_maestros))
        e = int(np.random.choice(n_materias))
        if 0 in inicial[b,:]:
            value = vecindad24(inicial,e,b)
        else:
            value = inicial
    return value

#Funcion auxiliar para mutar un hijo.
def mutacion2(hijo,proba_m,datos):
    ran = np.random.uniform()
    if ran<proba_m:
        mutado = xmen(hijo,datos)
    else:
        mutado = hijo
    return mutado

#Funcion auxiliar para crossover del metodo genetico.
def cruza2(ma,pa,n_materias):
    n_maestros = np.size(ma[:,0])
    n_max = np.size(ma[0,:])
    hijo1 = np.zeros((n_maestros,n_max))
    hijo2 = np.zeros((n_maestros,n_max))
    for a in range(1,n_materias+1):
        ma_who = encontrador(ma,a)
        pa_who = encontrador(pa,a)
        ran = np.random.uniform()
        if ran<0.5:
            hijo1 = asignador(hijo1,a,ma_who)
            hijo2 = asignador(hijo2,a,pa_who)
        else:
            hijo1 = asignador(hijo1,a,pa_who)
            hijo2 = asignador(hijo2,a,ma_who)
```

```
    return hijo1, hijo2

#Funcion auxiliar para ruleta del metodo genetico.
def matchpool2(poblacion,poblacion_f,n_pool):
    tamano = np.size(poblacion_f)
    worst = poblacion_f[0]
    f_modi = [abs(poblacion_f[j]-worst)+1 for j in range(tamano)]
    normaliza = np.sum(f_modi)
    probas = [f_modi[i]/normaliza for i in range(tamano)]
    index = [i for i in range(tamano)]
    index_pool = np.random.choice(index,size=n_pool,replace=False,p=probas)
    pool = []
    for a in range(n_pool):
        who = int(index_pool[a])
        seleccion = poblacion[who]
        pool.append(seleccion)
    return pool

#Funcion auxiliar para acomodar una lista de soluciones en orden de fitness.
def acomodar2(poblacion,datos,alfa,beta,gama):
    tamano = len(poblacion)
    fit = [objetivo2(poblacion[i],datos,alfa,beta,gama) for i in range(tamano)]
    orden = np.argsort(fit)
    new_fit = np.sort(fit)
    new_pop = []
    for a in range(tamano):
        index = int(orden[a])
        elemento = poblacion[index]
        new_pop.append(elemento)
    return new_pop, new_fit

#Auxiliar para, dada una materia, encontrar a quien le corresponde.
def encontrador(solu,e):
    n_maestros = np.size(solu[:,0])
    donde = int
    for i in range(n_maestros):
        now = solu[i,:]
        if e in now:
            donde = i
            break
        else:
            continue
    return donde

#Auxiliar para encontrar posicion de valor en array.
def prefiero(lista,numero):
    value = float
    try:
        value = np.where(lista==numero)[0][0]
    except IndexError:
        value = -1
    return value

#Auxiliar para longitud de lista/int
def lengo(lista):
```

```
    val = 0
    try:
        val = len(lista)
    except TypeError:
        val = 1
    return val

#Auxiliar para asignar materia a un maestro
def asignador(solu,e,a1):
    solo = np.array(solu)
    tamano = np.size(solu[0,:])
    index = int
    if 0 in solu[a1,:]:
        for t in range(tamano):
            prueba = solu[a1,t]
            if prueba==0:
                index = t
                break
            else:
                continue
        solo[a1,index] = e
    else:
        solo = materiador(solo,e)
    return solo

#Auxiliar para asignar materia de forma aleatoria
def materiador(solu,azar):
    solo = np.array(solu)
    n_maestros = np.size(solo[:,0])
    check = True
    while check==True:
        maestro = np.random.randint(n_maestros)
        if 0 in solo[maestro,:]:
            for k in range(3):
                if solo[maestro,k]==0:
                    solo[maestro,k] = azar
                    check = False
                    break
                else:
                    continue
        else:
            continue
    return solo

#Auxiliar para convertir datos str a numeros.
def stringer(lista):
    new = np.zeros(4)
    tamanio = len(lista)
    commas_pos = []
    for i in range(tamanio):
        test = lista[i]
        if test==',':
            commas_pos.append(i)
    uno = lista[0]
    for x1 in range(1,commas_pos[0]):
```

```
        uno = uno + lista[x1]
    new[0] = int(uno)

    dos = lista[commas_pos[0]+1]
    for x2 in range(commas_pos[0]+2,commas_pos[1]):
        dos = dos + lista[x2]
    new[1] = int(dos)

    tres = lista[commas_pos[1]+1]
    for x3 in range(commas_pos[1]+2,commas_pos[2]):
        tres = tres + lista[x3]
    new[2] = int(tres)

    cuatro = lista[commas_pos[2]+1]
    for x4 in range(commas_pos[2]+2,len(lista)):
        cuatro = cuatro + lista[x4]
    new[3] = int(cuatro)

    return new

######################
#CARGA DE DATOS
def generador2(archivo1,archivo2):
    materias_exc = (pd.read_excel(archivo1, sheet_name='Materias')).as_matrix()
    maestros_exc = (pd.read_excel(archivo2, sheet_name='datos')).as_matrix()
    n_materias = len(materias_exc[:,0])
    n_maestros = len(maestros_exc[:,0])
    maestro_dato = np.zeros((n_maestros,4))
    materia_horas = np.zeros(n_materias)
    maestro_experto = np.zeros((n_maestros,4))
    maestro_pref = np.zeros((n_maestros,4))
    for g in range(n_materias):
        value = materias_exc[g,6]*materias_exc[g,8]
        materia_horas[g] = value
    for t in range(n_maestros):
        maestro_dato[t,0] = maestros_exc[t,1]
        maestro_dato[t,1] = maestros_exc[t,2]
        maestro_dato[t,2] = maestros_exc[t,3]
        maestro_dato[t,3] = maestros_exc[t,4]
        maestro_experto[t,:] = stringer(maestros_exc[t,5])
        maestro_pref[t,:] = stringer(maestros_exc[t,6])
    return materia_horas, maestro_dato, maestro_experto, maestro_pref


#FUNCION OBJETIVO

#HARD CONSTRAINT. Revisa que los maestros den clases
#que saben y quieren, y que no se pasen de horas
def hard2(maes_materia, maes_datos):
    hard21 = 0
    hard22 = 0
    hard23 = 0
    mat_hr = maes_datos[0]
    for t in range(np.size(maes_materia[:,0])):
        favoritas = maes_datos[3][t]
```

```
        experto = maes_datos[2][t]
        materios = maes_materia[t,:]
        hr_tra = maes_datos[1][t,0]
        hr_mat = 0
        for a in range(len(materios)):
            materia = materios[a]
            if materia==0:
                hard23 = hard23 + (hr_mat>hr_tra)
                break
            else:
                hr_mat = hr_mat + mat_hr[int(materia)-1]
                hard21 = hard21 + 1-(materia in favoritas)
                hard22 = hard22 + 1-(materia in experto)
    return -hard21-hard22-hard23


#SOFT CONSTRAINTS. #Mide preferencias, experiencia y evaluacion
def soft2(maes_materia, maes_datos, alfa, beta, gama):
    felicidad = 0
    calificacion = 0
    experiencia = 0
    for t in range(np.size(maes_materia[:,0])):
        favoritas = maes_datos[3][t]
        materios = maes_materia[t,:]
        if materios[0]==0:
            continue
        else:
            calificacion = calificacion + maes_datos[1][t,3]
            experiencia = experiencia + min(int(maes_datos[1][t,2]/2.0),10)
            for a in range(len(materios)):
                materia = materios[a]
                if materia==0:
                    break
                else:
                    val = prefiero(favoritas,materia)
                    if val<0:
                        continue
                    else:
                        felicidad = felicidad + 4 - val
    return alfa*felicidad+beta*calificacion+gama*experiencia


#Funcion objetivo
def objetivo2(solucion, datos, alfa, beta, gama):
    soft = soft2(solucion,datos,alfa,beta,gama)
    n_maestros = np.size(solucion[:,0])
    cota = 10*n_maestros*(alfa+beta+gama)
    hard = hard2(solucion,datos)
    return soft+cota*hard

##########################
#GENERA SOLUCION INICIAL
##########################
def inicial2(datos):
    n_maestros = np.size(datos[2][:,0])
```

```
    n_materias = np.size(datos[0])
    materios = [i+1 for i in range(n_materias)]
    np.random.shuffle(materios)
    solu = np.zeros((n_maestros,3))
    for j in range(n_materias):
        azar = materios[j]
        solu = materiador(solu,azar)
    return solu

#################################################################

#Vecindad 1. Dada una solucion, intercambia las materias de dos maestros.
def vecindad21(solu,a,b):
    vecino = np.array(solu)
    vecino[b,:] = solu[a,:]
    vecino[a,:] = solu[b,:]
    return vecino

#Vecindad 2. Dada una solucion, pone todas las asignaciones de cabeza
def vecindad22(solu):
    lx = np.size(solu[:,0])
    ly = np.size(solu[0,:])
    vecino = np.zeros((lx,ly))
    for i in range(np.size(vecino[:,0])):
        vecino[i,:] = solu[-1-i,:]
    return vecino

#Vecindad 3. Dada una solucion, le da la
#ultima materia de alguien a otro maestro que le  guste.
def vecindad23(solu,a,b):
    vecino = np.array(solu)
    maestro_a = solu[a,:]
    maestro_b = solu[b,:]
    tamano = np.size(maestro_a)
    materia = 0
    index_a = int
    index_b = int
    for t in range(tamano):
        prueba = maestro_a[-1-t]
        if prueba==0:
            continue
        else:
            materia = prueba
            index_a = -1-t
            break
    for y in range(tamano):
        pruebo = maestro_b[y]
        if pruebo==0:
            index_b = y
            break
        else:
            continue
    vecino[a,index_a] = 0
    vecino[b,index_b] = solu[a,index_a]
    return vecino
```

```
#Vecindad 4. Dada una solucion, escoge una materia y la asigna a otro profesor
def vecindad24(solu,e,b):
    vecino = np.array(solu)
    a = encontrador(solu,e)
    if a==b:
        nada=3
    else:
        maestro_a = solu[a,:]
        maestro_b = solu[b,:]
        tamano = np.size(maestro_a)
        index_a = int
        index_b = int
        for t in range(tamano):
            prueba = maestro_a[t]
            if prueba!=e:
                continue
            else:
                index_a = t
                break
        for y in range(tamano):
            pruebo = maestro_b[y]
            if pruebo==0:
                index_b = y
                break
            else:
                continue
        a_primo = np.zeros(tamano)
        for u in range(tamano):
            if u<index_a:
                a_primo[u] = maestro_a[u]
            else:
                try:
                    a_primo[u] = maestro_a[u+1]
                except IndexError:
                    continue
        vecino[a,:] = a_primo
        vecino[b,index_b] = e
    return vecino


#Selecciona entre todas las vecindades definidas y regresa un vecino correspondiente.
def vegals_select2(solu,n,a=None,b=None):
    if n==1:
        value = vecindad21(solu,a,b)
    elif n==2:
        value = vecindad22(solu)
    elif n==3:
        value = vecindad23(solu,a,b)
    else:
        value = vecindad24(solu,a,b)
    return value

#Funcion para delta evaluar una solucion.
```

```
def f_delta2(solu,datos,alfa,beta,gama):
    hard = hard2(solu,datos)
    value = hard
    return value

#Dado una solucion inicial y un evento, evalua todos los vecinos posibles.
def vegals2(xo,datos,e,alfa,beta,gama,funcion):
    n_maestros = np.size(xo[:,0])
    n_materias = np.size(datos[0])
    xn = xo
    fn = funcion(xn,datos,alfa,beta,gama)
    marker = 0
    find = False
    #Vecindad 1
    for b in range(e+1,n_maestros):
        xp = vegals_select2(xn,1,e,b)
        fp = funcion(xp,datos,alfa,beta,gama)
        if fp>fn:
            xn = xp
            fn = fp
            marker = 1
            find = True
            break
    #Vecindad 2
    if find==False:
        xp = vegals_select2(xn,2)
        fp = funcion(xp,datos,alfa,beta,gama)
        if fp>fn:
            xn = xp
            fn = fp
            marker = 1
            find = True
    #Vecindad 3
    if find==False:
        if np.sum(xo[e,:])>0:
            for c in range(e):
                vecino = xo[c,:]
                if 0 in vecino:
                    xp = vegals_select2(xn,3,e,c)
                    fp = funcion(xp,datos,alfa,beta,gama)
                    if fp>fn:
                        xn = xp
                        fn = fp
                        marker = 1
                        find = True
                        break
    if find==False:
        if np.sum(xo[e,:])>0:
            for c in range(e+1,n_maestros):
                vecino = xo[c,:]
                if 0 in vecino:
                    xp = vegals_select2(xn,3,e,c)
                    fp = funcion(xp,datos,alfa,beta,gama)
                    if fp>fn:
                        xn = xp
```

```
                        fn = fp
                        marker = 1
                        find = True
                        break
    #Vecindad 4
    if find==False:
        for d in range(n_materias):
            for g in range(n_maestros):
                vecino = xn[g,:]
                if 0 in vecino:
                    xp = vegals_select2(xn,4,d+1,g)
                    fp = funcion(xp,datos,alfa,beta,gama)
                    if fp>fn:
                        xn = xp
                        fn = fp
                        marker = 1
                        find=True
                        break
            if find==False:
                continue
            else:
                break
    return [xn,fn,marker]

#Seccion de busqueda local para GALS de construccion.
def gals_cons2(xo,datos,alfa,beta,gama):
    xn = xo
    fn = f_delta2(xn,datos,alfa,beta,gama)
    n_maestros = np.size(xo[:,0])
    i = 0
    while i<(n_maestros):
        if fn==0:
            i = n_maestros
        else:
            buscar = vegals2(xn,datos,i,alfa,beta,gama,f_delta2)
            if buscar[2]==1:
                xn = buscar[0]
                fn = buscar[1]
                i = 0
            else:
                i = i+1
    return xn

#Seccion de busqueda local para GALS de mejoramiento.
def gals_impro2(xo,datos,alfa,beta,gama):
    xn = xo
    fn = objetivo2(xn,datos,alfa,beta,gama)
    n_maestros = np.size(xo[:,0])
    i = 0
    while i<(n_maestros):
        buscar = vegals2(xn,datos,i,alfa,beta,gama,objetivo2)
        if buscar[2]==1:
            xn = buscar[0]
            fn = buscar[1]
            i = 0
```

```
        else:
            i = i+1
    return xn

#Busqueda local para GALS.
def gals_local2(shared,datos,alfa,beta,gama,z,j):
    zeta = z
    size_all = len(shared)
    while z*j+zeta>size_all:
        zeta = zeta-1
    if zeta==0:
        pass
    else:
        for i in range(zeta):
            entry = i+j*z
            loco = shared[entry]
            construye = gals_cons2(loco,datos,alfa,beta,gama)
            mejora = gals_impro2(construye,datos,alfa,beta,gama)
            shared[entry] = mejora

#Para hacer GALS en paralelo sobre una poblacion.
def gals_poblacion2(poblacion,datos,alfa,beta,gama):
    num_cores = mp.cpu_count()
    pop = len(poblacion)
    manager = mp.Manager()
    shared = manager.list()
    for i in range(pop):
        loki = poblacion[i]
        shared.append(loki)
    processes = []
    z = int(math.ceil((1.0*pop)/(num_cores)))
    for j in range(num_cores):
        pj = mp.Process(target=gals_local2, args=(shared,datos,alfa,beta,gama,z,j))
        processes.append(pj)
        pj.start()
    for j in range(num_cores):
        processes[j].join()

    return shared


#GALS2
#################################################################################
def gals_paper2(datos,alfa,beta,
gama,n_generaciones,n_poblacion,n_pool,proba_m,timo,SEGUIR):
    starto = time.time()
    print('Acaba de empezar '+str(SEGUIR))
    n_materias = np.size(datos[0])
    poblacion = []
    for a in range(n_poblacion):
        solu = inicial2(datos)
        poblacion.append(solu)
    poblacion = gals_poblacion2(poblacion,datos,alfa,beta,gama)
    poblacion, poblacion_f = acomodar2(poblacion,datos,alfa,beta,gama)
    mejor = [poblacion[-1],poblacion_f[-1]]
```

```
    ##################################
    tops = []
    tops.append([(time.time()-starto)/60,mejor[1]])
    materia_excel(mejor[0],'asignacion_maestro_'+'inicial'+'_'+str(SEGUIR))
    SORA = 0
    ##################################
    while (time.time()-starto)/60<timo:
        SORA = SORA+1
        padres = matchpool2(poblacion,poblacion_f,n_pool)
        hijos = []
        for b in range(int(n_poblacion/2.0)):
            pareja = np.random.choice(n_pool,size=2,replace=False)
            mama = padres[int(pareja[0])]
            papa = padres[int(pareja[1])]
            hijo1, hijo2 = cruza2(mama,papa,n_materias)
            hijo1 = mutacion2(hijo1,proba_m,datos)
            hijo2 = mutacion2(hijo2,proba_m,datos)
            hijos.append(hijo1)
            hijos.append(hijo2)
        ##################################
        if SORA%1000==0:
            hijos = gals_poblacion2(hijos,datos,alfa,beta,gama)
        else:
            pass
        ##################################
        poblacion, poblacion_f = acomodar2(hijos,datos,alfa,beta,gama)
        mejor_local = [poblacion[-1],poblacion_f[-1]]
        if mejor_local[1]>mejor[1]:
            mejor = mejor_local
        else:
            poblacion[-1] = mejor[0]
            poblacion_f[-1] = mejor[1]
        tops.append([(time.time()-starto)/60,mejor[1]])
        ##################################
        #SEGUIMIENTO
        if SORA%200==0:
            materia_excel(mejor[0],'asignacion_maestro_'+str(SORA)+'_'+str(SEGUIR))
        else:
            pass
        #SEGUIMIENTO
        ##################################
    tamano = len(tops)
    grafo = np.zeros((2,tamano))
    for ja in range(tamano):
        grafo[0,ja] = tops[ja][0]
        grafo[1,ja] = tops[ja][1]
    np.save('gals2_'+str(SEGUIR)+'.npy',grafo)
    print('Acaba de terminar '+str(SEGUIR))
    return poblacion
#############################################################################
#############################################################################


#############################################################################
#CONVERTIDOR DE DATOS A EXCELL
```

```
##############################################################################

#Dada una solucion, escribe un excel con las asignaciones de cada materia.
def materia_excel(x,namae):
    maestros_exc = (pd.read_excel('entrada_maestros.xlsx',
    sheet_name='datos')).as_matrix()
    n_maestros = np.size(x[:,0])
    lista = []
    for t in range(n_maestros):
        value = (maestros_exc[t,0],x[t,0],x[t,1],x[t,2])
        lista.append(value)
    labels = ['Maestro','Materia 1','Materia 2','Materia 3']
    df = pd.DataFrame.from_records(lista,columns=labels)
    writer = pd.ExcelWriter(namae+'.xlsx', engine='xlsxwriter')
    df.to_excel(writer, sheet_name='Sheet1')
    writer.save()
```

Bibliografía

[1] Manar Hosny and Shameem Fatima. A survey of genetic algorithms for the university timetabling problem. *International Proceedings of Computer Science and Information Technology*, 13, 2011.

[2] Hamed Babaei, Jaber Karimpour, and Amin Hadidi. A survey of approaches for university course timetabling problem. *Computers & Industrial Engineering*, 86:43–59, 2015.

[3] Michael W Carter and Gilbert Laporte. Recent developments in practical course timetabling. In *international conference on the practice and theory of automated timetabling*, pages 3–19. Springer, 1997.

[4] Gerhard Post, Luca Di Gaspero, Jeffrey H Kingston, Barry McCollum, and Andrea Schaerf. The third international timetabling competition. *Annals of Operations Research*, 239(1):69–75, 2016.

[5] Gerhard Post, Luca Di Gaspero, Jeffrey H Kingston, Barry McCollum, and Andrea Schaerf. The third international timetabling competition. *Annals of Operations Research*, 239(1):69–75, 2016.

[6] Fotografía de la facultad de ciencias, unam. `https://ernestomataplata.me/divulgacion/facultad-de-ciencias-unam/`.

[7] Temel Öncan. A survey of the generalized assignment problem and its applications. *INFOR: Information Systems and Operational Research*, 45(3):123–141, 2007.

[8] A Sima Uyar, Ender Ozcan, and Neil Urquhart. *Automated Scheduling and Planning: From Theory to Practice*, volume 505. Springer, 2013.

[9] William Cook, László Lovász, Paul D Seymour, et al. *Combinatorial optimization: papers from the DIMACS Special Year*, volume 20. American Mathematical Soc., 1995.

[10] Alejandro Amador. Metodologías de muestreo acelerado en dinámicas moleculares. Tesis no publicada, 2018.

[11] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 2009.

[12] Thomas A Sudkamp and Alan Cotterman. *Languages and machines: an introduction to the theory of computer science*, volume 2. Addison-Wesley Reading, Mass., 1988.

[13] Thomas A Sudkamp and Alan Cotterman. *Languages and machines: an introduction to the theory of computer science*, volume 2. Addison-Wesley Reading, Mass., 1988.

[14] Anthony Wren. Scheduling, timetabling and rostering—a special relationship? In *International Conference on the Practice and Theory of Automated Timetabling*, pages 46–75. Springer, 1995.

[15] Andrea Schaerf. A survey of automated timetabling. *Artificial intelligence review*, 13(2):87–127, 1999.

[16] Donald F Stanat and David F McAllister. *Discrete mathematics in computer science*. Number 510 S7. 1977.

[17] David Abramson and J Abela. A parallel genetic algorithm for solving the school timetabling problem. 1991.

[18] Victor A Bardadym. Computer-aided school and university timetabling: The new wave. In *international conference on the practice and theory of automated timetabling*, pages 22–45. Springer, 1995.

[19] Michael W Carter and Gilbert Laporte. Recent developments in practical course timetabling. In *international conference on the practice and theory of automated timetabling*, pages 3–19. Springer, 1997.

[20] Inga Lilja Eiríksdóttir et al. *Optimization model for assigning teachers to classes*. PhD thesis, 2016.

[21] Aldy Gunawan and Kien Ming Ng. Solving the teacher assignment problem by two metaheuristics. *International Journal of Information and Management Sciences*, 22(1):73, 2011.

[22] Victor A Bardadym. Computer-aided school and university timetabling: The new wave. In *international conference on the practice and theory of automated timetabling*, pages 22–45. Springer, 1995.

[23] George HG Fonseca and Haroldo G Santos. Variable neighborhood search based algorithms for high school timetabling. *Computers & Operations Research*, 52:203–208, 2014.

[24] Alireza Rashidi Komijan and Mehrdad Nouri Koupaei. A mathematical model for university course scheduling: a case study. *International Journal of Technical Research and Applications*, 3(19):20–5, 2015.

[25] Yuri Kochetov, Polina Obuhovskaya, and Mikhail Paschenko. Local search heuristics for the teacher/class timetabling problem. In *PATAT 2006—Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling. Masaryk University: Brno, Czech Republic*, pages 454–457, 2006.

[26] Keith Murray, Tomáš Müller, and Hana Rudová. Modeling and solution of a complex university course timetabling problem. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 189–209. Springer, 2006.

[27] Rakesh P Badoni, DK Gupta, and Pallavi Mishra. A new hybrid algorithm for university course timetabling problem using events based on groupings of students. *Computers & industrial engineering*, 78:12–25, 2014.

[28] Facultad de ciencias, unam. `http://www.fciencias.unam.mx/`.

[29] Grigorios N Beligiannis, C Moschopoulos, and Spiridon D Likothanassis. A genetic algorithm approach to school timetabling. *Journal of the Operational Research Society*, 60(1):23–42, 2009.

[30] Jan A Snyman. Practical mathematical optimization. 2005.

[31] Wayne L Winston and Jeffrey B Goldberg. *Operations research: applications and algorithms*, volume 3. Thomson Brooks/Cole Belmont, 2004.

[32] Operations research and analytics. `https://www.informs.org/About-INFORMS/What-is-Operations-Research`.

[33] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[34] George B Dantzig and Mukund N Thapa. *Linear programming 1: introduction*. Springer Science & Business Media, 2006.

[35] Donald Goldfarb. On the complexity of the simplex method. In *Advances in optimization and numerical analysis*, pages 25–38. Springer, 1994.

[36] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, 20, 2012.

[37] George L Nemhauser and Laurence A Wolsey. Integer programming and combinatorial optimization. *Wiley, Chichester. GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, 20:8–12, 1988.

[38] Mokhtar S Bazaraa, Hanif D Sherali, and Chitharanjan M Shetty. *Nonlinear programming: theory and algorithms*. John Wiley & Sons, 2013.

[39] Colin R Reeves. *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science*, volume 15. Mc Graw-Hill, 1995.

[40] Natallia Kokash. An introduction to heuristic algorithms. *Department of Informatics and Telecommunications*, pages 1–8, 2005.

[41] Colin R Reeves. *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science*, volume 15. Mc Graw-Hill, 1995.

[42] Heiner Müller-Merbach. Heuristics and their design: a survey. *European Journal of Operational Research*, 8(1):1–23, 1981.

[43] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[44] Ibrahim H Osman and James P Kelly. Meta-heuristics: an overview. In *Meta-heuristics*, pages 1–21. Springer, 1996.

[45] Wei King Tiong San Nah Sze. A comparison between heuristic and meta-heuristic methods for solving the multiple traveling salesman problem. 2007.

[46] Marco A Boschetti, Vittorio Maniezzo, Matteo Roffilli, and Antonio Bolufé Röhler. Matheuristics: Optimization, simulation and control. In *International Workshop on Hybrid Metaheuristics*, pages 171–177. Springer, 2009.

[47] M Caserta and S Voß. Matheuristics: Hybridizing metaheuristics and mathematical programming, 2010.

[48] Olaf Schenk, Andreas Wächter, and Michael Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *Computational Optimization and Applications*, 36(2-3):321–341, 2007.

[49] Sanja Petrovic and EK Burke. University timetabling. ch. 45 in the handbook of scheduling: Algorithms, models, and performance analysis (eds. j. leung), chapman hall, 2004.

[50] NK Cauvery. Timetable scheduling using graph coloring. *International Journal of P2P Network Trends and Technology*, 1(2):57–62, 2011.

[51] Teddy Wijaya and Ruli Manurung. Solving university timetabling as a constraint satisfaction problem with genetic algorithm. In *Proceedings of the international conference on advanced computer science and information systems (ICACSIS 2009), Depok*, 2009.

[52] Olivia Rossi-Doria, Michael Sampels, Mauro Birattari, Marco Chiarandini, Marco Dorigo, Luca M Gambardella, Joshua Knowles, Max Manfrin, Monaldo Mastrolilli, Ben Paechter, et al. A comparison of the performance of different metaheuristics on the timetabling problem. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 329–351. Springer, 2002.

[53] Salwani Abdullah, Khalid Shaker, Barry McCollum, and Paul McMullan. Dual sequence simulated annealing with round-robin approach for university course timetabling. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 1–10. Springer, 2010.

[54] Zhipeng Lü and Jin-Kao Hao. Adaptive tabu search for course timetabling. *European Journal of Operational Research*, 200(1):235–244, 2010.

[55] Rakesh P Badoni, DK Gupta, and Pallavi Mishra. A new hybrid algorithm for university course timetabling problem using events based on groupings of students. *Computers & industrial engineering*, 78:12–25, 2014.

[56] Grigorios N Beligiannis, C Moschopoulos, and Spiridon D Likothanassis. A genetic algorithm approach to school timetabling. *Journal of the Operational Research Society*, 60(1):23–42, 2009.

[57] Dinabandhu Bhandari, CA Murthy, and Sankar K Pal. Genetic algorithm with elitist model and its convergence. *International journal of pattern recognition and artificial intelligence*, 10(06):731–747, 1996.

[58] Pupong Pongcharoen, Weena Promtet, Pisal Yenradee, and Christian Hicks. Stochastic optimisation timetabling tool for university course scheduling. *International Journal of Production Economics*, 112(2):903–918, 2008.

[59] Keith Murray, Tomáš Müller, and Hana Rudová. Modeling and solution of a complex university course timetabling problem. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 189–209. Springer, 2006.

[60] Google maps. https://www.google.com/maps/place/Facultad+de+Ciencias+UNAM/@19.3240451,-99.1823727,1108m/data=!3m1!1e3!4m5!3m4!1s0x85ce000fdd96288f:0x1096af9b5b03d38d!8m2!3d19.3240451!4d-99.1801787.

[61] *Fandom* de avatar. https://avatar.fandom.com/es/wiki/Kyoshi.

[62] Video en youtube: "12 hours of age of empires priest conversion (ayoyoyo wololo)". https://www.youtube.com/watch?v=7CIGb1Ti06k.

[63] Página oficial de freddie mercury. http://www.freddiemercury.com/es.

[64] William M Bolstad and James M Curran. *Introduction to Bayesian statistics*. John Wiley & Sons, 2016.