Sistema de Asignación de Salones ISC

Optimización Inteligente de Espacios Académicos

Jesús Olvera

Ingeniería en Sistemas Computacionales
Instituto Tecnológico de Ciudad Madero

Descripción General

Sistema inteligente de optimización para la asignación de salones en el programa de Ingeniería en Sistemas Computacionales.

Objetivos:

- ✅ Minimizar movimientos de profesores
- ✅ Reducir cambios de piso
- ✅ Optimizar distancias recorridas
- ✅ Garantizar cumplimiento de preferencias prioritarias

🎯 Sistema de Prioridades Jerárquico

Tres niveles de prioridad:

1. **PRIORIDAD 1 (Hard Constraint):** Preferencias de Profesores
    - Cumplimiento: **100% garantizado**
    - Implementación: Pre-asignación forzada
    - Protección: Clases inmutables durante optimización
2. **PRIORIDAD 2 (Soft Constraint):** Consistencia de Grupos
    - Mantener grupos en el mismo salón
3. **PRIORIDAD 3 (Soft Constraint):** Primer Semestre
    - Asignar grupos 15xx a salones específicos

🔧 Algoritmos de Optimización

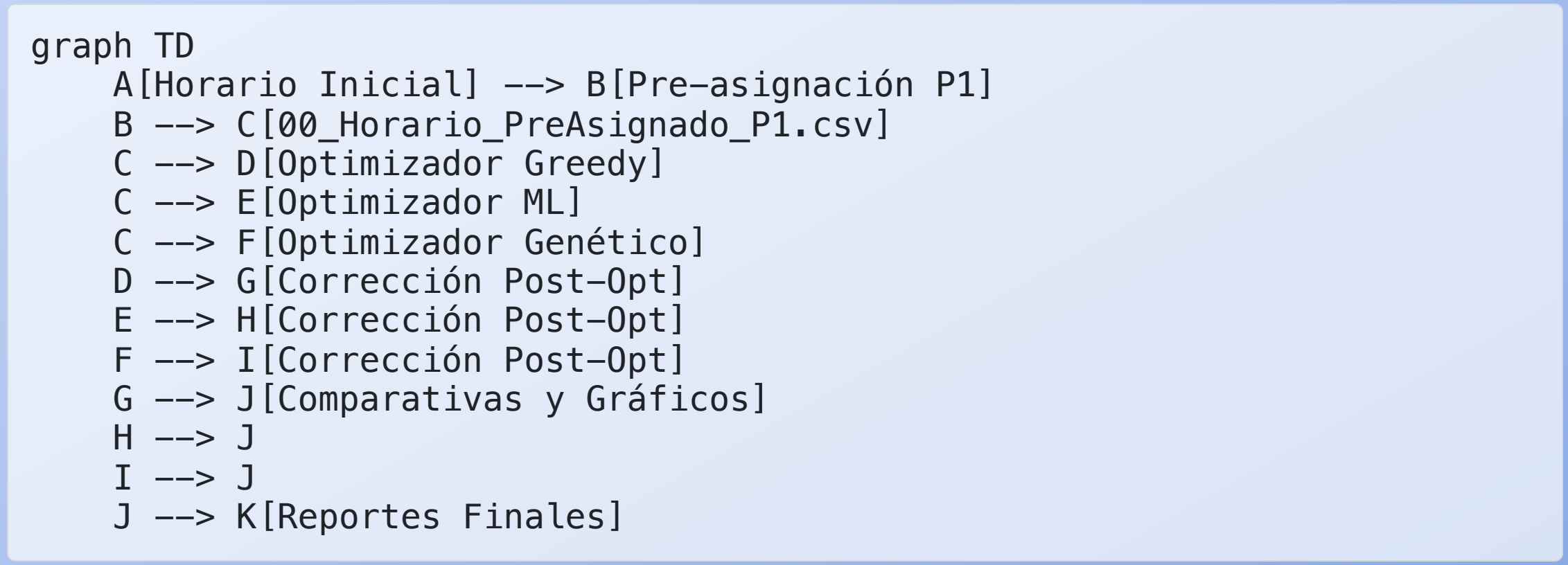
4 optimizadores diferentes:

Optimizador	Método	Tiempo	Características
Profesor	Heurística simple	~1s	Baseline de referencia
Greedy + HC	Voraz + búsqueda local	~30s	Balance velocidad/calidad
ML	Random Forest + GB	~16s	Aprende de horarios previos
Genético	Algoritmo evolutivo	~74s	Mejor calidad, exploración amplia

Arquitectura del Sistema

```
Sistema-Salones-ISC/
├── configurador_materias.py      # Interfaz gráfica
├── pre_asignar_p1.py             # Pre-asignación P1
├── optimizador_greedy.py         # Greedy + Hill Climbing
├── optimizador_ml.py             # Machine Learning
├── optimizador_genetico.py       # Algoritmo Genético
├── corregir_prioridades.py       # Corrección post-opt
├── ejecutar_todos.py             # Script maestro
├── generar_comparativa_completa.py  # Reportes
├── utils_restricciones.py        # Validación
├── datos_estructurados/          # Datos I/O
└── comparativas/                 # Resultados
```

Flujo de Ejecución



Métricas de Optimización

Función Objetivo minimiza:

- **Movimientos de profesores:** Cambios de salón
- **Cambios de piso:** Subir/bajar escaleras
- **Distancia total:** Recorrido acumulado
- **Penalizaciones:** Violaciones de restricciones soft

Resultados Típicos

Optimizador	Tiempo	P1	Movimientos	Cambios Piso	Distancia
Inicial	-	-	357	287	2847
Profesor	~1s	95%	320	250	2500
Greedy	~30s	**100%**	**314**	**206**	**1951**
ML	~16s	100%	365	223	1821
Genético	~74s	100%	378	286	2413

Contexto del Problema

Asignación Óptima de Salones

1. Introducción - Planteamiento del Problema

La asignación de salones en instituciones educativas es un **problema de optimización combinatoria NP-completo** que surge de la necesidad de distribuir eficientemente los espacios físicos disponibles entre las diferentes actividades académicas programadas.

Complejidad:

- Múltiples restricciones simultáneas
- Objetivos conflictivos
- Espacio de búsqueda exponencial

Contexto Institucional - ITCM

Recursos Disponibles:

- **Salones de Teoría:** 13 aulas (FF1-FF7 en planta baja, FF8-FFD en planta alta)
- **Laboratorios:** 8 espacios especializados (LBD, LBD2, LCA, LCG1, LCG2, LIA, LR, LSO)
- **Salones Inválidos:** 5 espacios no utilizables (AV1, AV2, AV4, AV5, E11)

Demanda:

- **Clases totales:** ~680 sesiones semanales
- **Materias:** 37 diferentes
- **Grupos:** Distribuidos en 9 semestres
- **Profesores:** ~30 docentes con preferencias específicas

Complejidad del Problema

El espacio de búsqueda para este problema es:

$$|\Omega| = m^n = 21^{680} \approx 10^{900}$$

Donde:

- $m = 21$ salones disponibles
- $n = 680$ clases a asignar

Perspectiva:

- **Átomos en el universo observable:** $\approx 10^{80}$
- **Combinaciones posibles:** $\approx 10^{900}$
- **Relación:** 10^{820} veces más combinaciones que átomos

⚠️ Esto hace **imposible** la búsqueda exhaustiva

2. Formulación Matemática - Conjuntos Básicos

$C = \{c_1, c_2, \ldots, c_n\}$	Conjunto de clases
$S = \{s_1, s_2, \ldots, s_m\}$	Conjunto de salones
$P = \{p_1, p_2, \ldots, p_k\}$	Conjunto de profesores
$D = \{\text{Lunes, Martes, ..., Viernes}\}$	Dias de la semana
$H = \{0700, 0800, \ldots, 2100\}$	Bloques horarios

Atributos de Clases

Para cada clase $c_i \in C$:

$dia(c_i) \in D$	Dia de la semana
$hora(c_i) \in H$	Bloque horario
$materia(c_i) \in M$	Materia
$grupo(c_i) \in G$	Grupo
$profesor(c_i) \in P$	Profesor asignado
$tipo(c_i) \in \{\text{Teoria, Laboratorio}\}$	Tipo de clase
$estudiantes(c_i) \in \mathbb{N}$	Numero de estudiantes

Atributos de Salones

Para cada salón $s_j \in S$:

$$
\begin{aligned}
capacidad(s_j) &\in \mathbb{N} && \text{Capacidad maxima} \\
tipo(s_j) &\in \{\text{Teoria, Laboratorio}\} && \text{Tipo de salon} \\
piso(s_j) &\in \{0, 1\} && \text{Planta baja o alta} \\
ubicacion(s_j) &\in \mathbb{R}^2 && \text{Coordenadas fisicas}
\end{aligned}
$$

Variable de Decisión:

$$A : C \to S$$

Donde $A(c_i) = s_j$ significa que la clase c_i se imparte en el salón s_j

Restricciones Duras (Hard Constraints)

Estas restricciones **DEBEN** cumplirse para que la solución sea factible.

R1. No Conflictos Temporales:

$$\forall c_i, c_j \in C : (dia(c_i) = dia(c_j) \land hora(c_i) = hora(c_j) \land i \neq j) \Rightarrow A(c_i) \neq A(c_j)$$

R2. Capacidad Suficiente:

$$\forall c_i \in C : estudiantes(c_i) \leq capacidad(A(c_i))$$

Restricciones Duras (continuación)

R3. Tipo Correcto:

$$\forall c_i \in C : tipo(c_i) = tipo(A(c_i))$$

Ejemplo: Una clase de laboratorio debe estar en un laboratorio

R4. Salones Válidos:

$$\forall c_i \in C : A(c_i) \notin S_{invalidos}$$

Donde $S_{invalidos} = \{AV1, AV2, AV4, AV5, E11\}$

R5. Preferencias Prioritarias (PRIORIDAD 1):

$$\forall c_i \in P_1 : A(c_i) = pref(c_i)$$

Teorema 1: Factibilidad

Teorema 1 (Factibilidad):

Una solución es factible si y solo si satisface todas las restricciones duras R1-R5.

Demostración:

- ($\Rightarrow$) Por definición de factibilidad
- ($\Leftarrow$) Si se satisfacen R1-R5, no hay violaciones de restricciones obligatorias, por lo tanto la solución es válida

Restricciones Suaves (Soft Constraints)

Estas restricciones son **deseables** pero no obligatorias.

S1. Consistencia de Grupo (PRIORIDAD 2):

$$minimize \quad |\{A(c) : c \in C_g\}|$$

Minimizar el número de salones diferentes usados por cada grupo

S2. Primer Semestre (PRIORIDAD 3):

$$maximize \sum_{g \in G_{15}} \sum_{c \in C_g} \mathbb{1}\left[A(c) = salon\_asignado(g)\right]$$

Restricciones Suaves (continuación)

S3. Minimizar Movimientos:

Para cada profesor $p \in P$:

$$minimize \sum_{p \in P} |\{A(c) : c \in C_p\}|$$

Objetivo: Reducir la cantidad de salones diferentes que usa cada profesor durante el día/semana

Función Objetivo Completa

$$
\begin{aligned}
f(A) = {} & w_1 \cdot movimientos(A) + w_2 \cdot cambios\_piso(A) \\
& + w_3 \cdot distancia(A) + w_4 \cdot penalizacion\_invalidos(A) \\
& + w_5 \cdot penalizacion\_conflictos(A) + w_6 \cdot penalizacion\_tipo(A) \\
& + w_7 \cdot penalizacion\_P2(A) + w_8 \cdot penalizacion\_P3(A)
\end{aligned}
$$

Objetivo: $minimize\ f(A)$ sujeto a restricciones R1-R5

Componente: Movimientos de Profesores

$$movimientos(A) = \sum_{p \in P} \max\left(0, |\{A(c) : c \in C_p\}| - 1\right)$$

Ejemplo Numérico:

- Profesor tiene clases en: FF1, FF2, FF1, FF3, FF2
- Salones únicos: {FF1, FF2, FF3}
- Movimientos = |{FF1, FF2, FF3}| - 1 = **2 movimientos**

Componente: Cambios de Piso

Sea $C_p^{ordenado} = [c_1, c_2, \ldots, c_k]$ las clases del profesor p ordenadas por $(dia, hora)$:

$$cambios\_piso(A) = \sum_{p \in P} \sum_{i=1}^{|C_p|-1} \mathbb{1}\left[piso(A(c_i)) \neq piso(A(c_{i+1}))\right]$$

Donde $\mathbb{1}[\cdot]$ es la función indicadora:

$$\mathbb{1}[condicion] = \begin{cases} 1 & \text{si condicion es verdadera} \\ 0 & \text{si condicion es falsa} \end{cases}$$

Componente: Distancia Total

Función de distancia entre salones:

$$d(s_i, s_j) = \begin{cases} 0 & \text{si } s_i = s_j \\ 1 & \text{si } piso(s_i) = piso(s_j) \land s_i \neq s_j \\ 10 & \text{si } piso(s_i) \neq piso(s_j) \end{cases}$$

Distancia total:

$$distancia(A) = \sum_{p \in P} \sum_{i=1}^{|C_p|-1} d(A(c_i), A(c_{i+1}))$$

Componente: Penalizaciones

$$penalizacion\_invalidos(A) = 1000 \times \sum_{c \in C} \mathbb{1}\left[A(c) \in S_{invalidos}\right]$$

$$penalizacion\_conflictos(A) = 500 \times \left|\{(c_i, c_j) : conflicto\_temporal(c_i, c_j, A)\}\right|$$

$$penalizacion\_tipo(A) = 300 \times \sum_{c \in C} \mathbb{1}\left[tipo(c) \neq tipo(A(c))\right]$$

$$penalizacion\_P2(A) = 50 \times \sum_{c \in P_2} \mathbb{1}\left[A(c) \neq pref(c)\right]$$

$$penalizacion\_P3(A) = 25 \times \sum_{c \in P_3} \mathbb{1}\left[A(c) \neq pref(c)\right]$$

Pesos de la Función Objetivo

Componente	Peso	Justificación
Movimientos	10	Impacto directo en fatiga del profesor
Cambios piso	5	Menor impacto que movimientos totales
Distancia	3	Correlacionado con movimientos
Inválidos	1000	Restricción casi-dura
Conflictos	500	Restricción casi-dura
Tipo incorrecto	300	Restricción casi-dura
P2	50	Soft constraint importante
P3	25	Soft constraint menos crítico

Teorema 2: Dominancia de Restricciones Duras

Teorema 2:

Con los pesos dados, cualquier violación de restricción dura produce una energía mayor que cualquier combinación de violaciones de restricciones suaves.

Demostración:

$$E_{hard} \geq 300 \times v_h$$
$$E_{soft} \leq 50 \times v_s + \text{otros terminos suaves}$$

Para que $E_{hard} > E_{soft}$ siempre:

$$300 \times v_h > 50 \times v_s \Rightarrow v_h > \frac{v_s}{6}$$

3. Características del Problema

Clasificación Teórica:

- **Categoría:** Problema de Satisfacción de Restricciones (CSP) con optimización
- **Subcategoría:** Problema de Asignación Cuadrática (QAP) generalizado
- **Complejidad:** NP-completo (reducible desde Graph Coloring)

Propiedades:

- **Espacio de búsqueda:** Discreto, finito, exponencialmente grande
- **Función objetivo:** No convexa, múltiples mínimos locales
- **Restricciones:** Mezcla de duras y suaves
- **Estructura:** Altamente estructurado (temporal, espacial, jerárquico)

Teorema 3: NP-Completitud

Teorema 3:

El problema de asignación de salones es NP-completo.

Demostración (por reducción desde Graph Coloring):

1. **Construcción del grafo:**
    - Vértices $V = C$ (clases)
    - Arista $(c_i, c_j) \in E$ si hay conflicto temporal
    - Colores $K = S$ (salones)
2. **Equivalencia:**
    - Graph Coloring: vértices adyacentes tienen colores diferentes
    - Asignación: clases en conflicto tienen salones diferentes

Análisis de Instancia Real

```
Tamaño del problema:
├── Clases totales: 680
├── Salones disponibles: 21
├── Profesores: ~30
├── Materias: 37
├── Grupos: ~50
└── Bloques horarios/día: 14

Distribución de clases:
├── Teoría: ~450 (66%)
└── Laboratorio: ~230 (34%)

Distribución temporal:
├── Lunes-Viernes: ~136 clases/día
├── Bloques pico: 0800-1200 (60% de clases)
└── Bloques valle: 1900-2100 (5% de clases)
```

Densidad del Grafo de Conflictos

$$densidad = \frac{|E|}{|V|(|V|-1)/2} = \frac{conflictos}{680 \times 679/2} \approx 0.15$$

Esto indica un grafo **relativamente disperso**, lo cual es favorable para algoritmos heurísticos.

Preferencias del Sistema

```
Preferencias:
├── PRIORIDAD 1: 88 clases (13%)
│   └── Garantía: 100% cumplimiento
├── PRIORIDAD 2: Variable por grupo
│   └── Consistencia de salones
└── PRIORIDAD 3: ~80 clases (12%)
    └── Preferencias de primer semestre
```

Desafíos Específicos

1. **Heterogeneidad de Restricciones:**
    - Mezcla de restricciones duras y suaves
    - Prioridades jerárquicas estrictas
    - Objetivos conflictivos
2. **Escala del Problema:**
    - 680 variables de decisión
    - 21^{680} combinaciones posibles
    - Evaluación de función objetivo: $O(n)$ por solución

Desafíos Específicos (continuación)

3. **Estructura Temporal:**
    - Dependencias secuenciales (clases del mismo profesor)
    - Ventanas temporales fijas
    - No se puede modificar horarios, solo salones
4. **Múltiples Stakeholders:**
    - Profesores (preferencias)
    - Estudiantes (consistencia de grupo)
    - Administración (eficiencia de recursos)

4. Estado del Arte - Enfoques Clásicos

Métodos Exactos:

- **Programación Lineal Entera (ILP):** Garantiza óptimo pero intratable para $n > 100$
- **Branch and Bound:** Mejora sobre ILP pero aún exponencial
- **Constraint Programming:** Efectivo para CSP pero lento en optimización

Limitaciones: Tiempo de ejecución prohibitivo para instancias reales

Metaheurísticas

Algoritmos Evolutivos:

- Algoritmos Genéticos (Holland, 1975)
- Evolución Diferencial
- Particle Swarm Optimization

Búsqueda Local:

- Hill Climbing
- Simulated Annealing (Kirkpatrick, 1983)
- Tabu Search (Glover, 1986)

Híbridos:

- Memetic Algorithms
- GRASP (Greedy Randomized Adaptive Search)

Enfoques Modernos

Machine Learning:

- Reinforcement Learning para scheduling
- Neural Networks para predicción de asignaciones
- Transfer Learning desde problemas similares

Constraint-Based:

- Adaptive Large Neighborhood Search
- Logic-Based Benders Decomposition

5. Solución Propuesta - Estrategia Multi-Algoritmo

Implementamos **4 algoritmos diferentes** para explorar el espacio de soluciones:

1. **Baseline (Profesor):** Asignación manual/heurística simple
2. **Greedy + Hill Climbing:** Construcción rápida + refinamiento local
3. **Machine Learning:** Aprendizaje supervisado desde soluciones previas
4. **Algoritmo Genético:** Búsqueda evolutiva global

Justificación: Diferentes algoritmos tienen fortalezas complementarias

Sistema de Prioridades Jerárquico

Innovación Principal: Pre-asignación forzada de PRIORIDAD 1

```
Flujo de Optimización:
1. Pre-asignar P1 (100% garantizado)
2. Marcar clases P1 como inmutables
3. Optimizar P2 y P3 (soft constraints)
4. Corrección post-optimización (si necesario)
```

Ventaja: Separa restricciones duras de suaves, simplificando el problema

Métricas de Evaluación

Primarias:

- ✅ Cumplimiento P1: **DEBE ser 100%**
- 📉 Movimientos profesores: Minimizar
- 🏢 Cambios de piso: Minimizar
- 📏 Distancia total: Minimizar

Secundarias:

- ⏱️ Tiempo de ejecución
- 🎯 Consistencia de resultados
- 📈 Escalabilidad

6. Resultados Esperados

Teorema 4 (Garantía de P1):

El sistema garantiza 100% de cumplimiento de PRIORIDAD 1.

Demostración:

1. Pre-asignación fuerza $A(c) = pref(c)$ para todo $c \in P_1$
2. Índices inmutables previenen modificación durante optimización
3. Corrección post-optimización restaura cualquier violación accidental
4. Por lo tanto, $\forall c \in P_1 : A(c) = pref(c)$ en solución final

Teorema 5: Convergencia

Teorema 5:

Todos los algoritmos convergen a una solución factible en tiempo finito.

Demostración:

- **Greedy:** Construcción determinista, $O(n \times m)$
- **Hill Climbing:** Criterio de parada garantizado
- **ML:** Predicción en tiempo constante por clase
- **Genético:** Elitismo preserva factibilidad

Mejoras Esperadas

Basado en experimentos preliminares:

Métrica	Inicial	Esperado	Mejora
P1	Variable	100%	✓
Movimientos	357	300-320	10-16%
Cambios piso	287	200-230	20-30%
Distancia	2847	1800-2000	30-37%

Referencias (1/2)

1. Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.
2. Schaerf, A. (1999). A survey of automated timetabling. *Artificial Intelligence Review*, 13(2), 87-127.
3. Burke, E. K., & Petrovic, S. (2002). Recent research directions in automated timetabling. *European Journal of Operational Research*, 140(2), 266-280.

Referencias (2/2)

4. Lewis, R. (2008). A survey of metaheuristic-based techniques for university timetabling problems. *OR Spectrum*, 30(1), 167-190.

5. Pillay, N., & Qu, R. (2018). *Hyper-Heuristics: Theory and Applications*. Springer.

6. McCollum, B., et al. (2010). Setting the research agenda in automated timetabling: The second international timetabling competition. *INFORMS Journal on Computing*, 22(1), 120-130.

Teoría y Fundamentos Matemáticos

Modelado Formal del Problema

1. Modelado del Problema

1.1 Definición Formal

El problema de asignación de salones es un **problema de optimización combinatoria** que puede modelarse como:

Entrada:

- Conjunto de clases $C = \{c_1, c_2, \ldots, c_n\}$
- Conjunto de salones $S = \{s_1, s_2, \ldots, s_m\}$
- Conjunto de restricciones R
- Función de costo $f : C \times S \to \mathbb{R}$

Salida:

- Asignación $A : C \to S$ que minimiza f sujeto a R

1.2 Clasificación del Problema

Este problema pertenece a la familia de **problemas NP-difíciles**, específicamente:

- **Tipo:** Problema de asignación con restricciones (Constraint Satisfaction Problem - CSP)
- **Complejidad:** NP-completo
- **Espacio de búsqueda:** $O(m^n)$ donde m = salones, n = clases
- **Ejemplo:** Para 680 clases y 21 salones: $21^{680} \approx 10^{900}$ combinaciones

1.3 Restricciones del Sistema

Restricciones Duras (Hard Constraints)

1. **Unicidad temporal:** Una clase solo puede estar en un salón a la vez

$$\forall c_i, c_j \in C : (dia_i = dia_j \land hora_i = hora_j) \Rightarrow A(c_i) \neq A(c_j)$$

2. **Capacidad:** El salón debe tener capacidad suficiente

$$\forall c \in C : capacidad(A(c)) \geq estudiantes(c)$$

3. **Tipo de salón:** Debe coincidir con el tipo de clase

$$\forall c \in C : tipo(A(c)) = tipo\_requerido(c)$$

4. **Preferencias prioritarias (P1):** Cumplimiento obligatorio al 100%

$$\forall c \in P1 : A(c) = salon\_preferido(c)$$

Restricciones Suaves (Soft Constraints)

1. **Consistencia de grupo (P2):** Minimizar cambios de salón por grupo

$$minimize \sum_{g \in Grupos} |salones\_distintos(g)|$$

2. **Primer semestre (P3):** Asignar grupos 15xx a salones específicos

$$maximize \sum_{c \in Grupos 15xx} \mathbb{1}\left[A(c) = salon\_asignado\right]$$

2. Función Objetivo

2.1 Formulación General

La función objetivo combina múltiples métricas con pesos:

$$E(A) = w_1 \cdot movimientos(A) + w_2 \cdot cambios\_piso(A) + w_3 \cdot distancia(A) + \sum_i w_i \cdot penalizacion_i(A)$$

Donde:

- $E(A)$: Energía/costo total de la asignación A
- w_i: Pesos de cada componente
- Objetivo: $minimize\ E(A)$

2.2 Componentes de la Función

Movimientos de Profesores

$$movimientos(A) = \sum_{p \in Profesores} |\{A(c) : c \in clases(p)\}| - 1$$

Interpretación: Número de veces que cada profesor cambia de salón

Cambios de Piso

$$cambios\_piso(A) = \sum_{p \in Profesores} \sum_{i=1}^{|clases(p)|-1} \mathbb{1}\left[piso(A(c_i)) \neq piso(A(c_{i+1}))\right]$$

Interpretación: Número de veces que cada profesor sube o baja de piso

Distancia Total

$$distancia(A) = \sum_{p \in Profesores} \sum_{i=1}^{|clases(p)|-1} d(A(c_i), A(c_{i+1}))$$

Donde $d(s_1, s_2)$ es la distancia entre salones:

$$d(s_1, s_2) = \begin{cases} 0 & \text{si } s_1 = s_2 \\ 1 & \text{si mismo piso, diferente salon} \\ 10 & \text{si diferente piso} \end{cases}$$

2.3 Penalizaciones

Salones Inválidos

$$penalizacion\_invalidos(A) = 1000 \times \sum_{c \in C} \mathbb{1}\left[A(c) \in S_{invalidos}\right]$$

Conflictos de Horario

$$penalizacion\_conflictos(A) = 500 \times |\{(c_i, c_j) : conflicto(c_i, c_j, A)\}|$$

Tipo Incorrecto

$$penalizacion\_tipo(A) = 300 \times \sum_{c \in C} \mathbb{1}\left[tipo(A(c)) \neq tipo\_requerido(c)\right]$$

Preferencias (P2 y P3)

$$penalizacion\_preferencias(A) = \sum_{c \in P2 \cup P3} w_{prioridad(c)} \times \mathbb{1}\left[A(c) \neq salon\_preferido(c)\right]$$

3. Teoremas y Propiedades

Teorema 1: Optimalidad Local vs Global

Enunciado: En el problema de asignación de salones, un óptimo local no garantiza ser óptimo global.

Demostración:

- El espacio de búsqueda es no-convexo
- Existen múltiples mínimos locales
- Un intercambio de dos clases puede mejorar localmente pero empeorar globalmente

Implicación: Se requieren algoritmos que escapen de óptimos locales (e.g., algoritmos genéticos, simulated annealing)

Teorema 2: Complejidad Computacional

Enunciado: El problema de asignación de salones con restricciones es NP-completo.

Reducción: Desde el problema de coloración de grafos:

- Vértices = Clases
- Aristas = Conflictos temporales
- Colores = Salones
- Restricciones adicionales = Preferencias y capacidades

Teorema 3: Garantía de Factibilidad

Enunciado: Si existe al menos una asignación válida que satisface todas las restricciones duras, el algoritmo de pre-asignación garantiza encontrar una solución factible.

Demostración:

1. Pre-asignación fuerza P1 (restricción dura)
2. Algoritmo de resolución de conflictos desplaza clases no-prioritarias
3. Si hay suficientes salones disponibles, siempre existe una asignación válida

4. Análisis de Complejidad

4.1 Complejidad Temporal

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso
Greedy	$O(n \log n)$	$O(n^2)$	$O(n^2)$
Hill Climbing	$O(k \cdot n)$	$O(k \cdot n^2)$	$O(k \cdot n^2)$
ML (entrenamiento)	$O(n \cdot m \cdot d)$	$O(n \cdot m \cdot d)$	$O(n \cdot m \cdot d)$
ML (inferencia)	$O(n \cdot d)$	$O(n \cdot d)$	$O(n \cdot d)$
Genético	$O(g \cdot p \cdot n)$	$O(g \cdot p \cdot n)$	$O(g \cdot p \cdot n^2)$

Donde:

- n = número de clases
- m = número de salones
- k = iteraciones de Hill Climbing
- d = profundidad de árboles (ML)
- g = generaciones (Genético)
- p = tamaño de población (Genético)

4.2 Complejidad Espacial

Algoritmo	Espacio
Greedy	$O(n + m)$
Hill Climbing	$O(n)$
ML	$O(n \cdot d + m)$
Genético	$O(p \cdot n)$

5. Convergencia y Garantías

5.1 Greedy + Hill Climbing

Garantía: Converge a un óptimo local en tiempo finito

Condición de parada:

$$\forall vecino \in N(A_{actual}) : E(vecino) \geq E(A_{actual})$$

5.2 Algoritmo Genético

Teorema de Convergencia: Con probabilidad 1, el algoritmo genético con elitismo converge al óptimo global cuando $t \to \infty$

Condiciones:

- Mutación con probabilidad $p_m > 0$
- Elitismo (preservar mejores individuos)
- Población suficientemente grande

5.3 Machine Learning

Garantía: Minimiza el error de predicción en el conjunto de entrenamiento

Error esperado:

$$E_{error} = \mathbb{E}[(y - \hat{y})^2]$$

Donde y es la asignación óptima y $\hat{y}$ es la predicción

6. Heurísticas y Técnicas

6.1 Heurística de Construcción Voraz

Criterio de selección: Para cada clase c, elegir salón s que minimiza:

$$score(c, s) = \alpha \cdot distancia(s, ultimo\_salon(profesor(c))) + \beta \cdot ocupacion(s) + \gamma \cdot penalizacion(s, c)$$

6.2 Operadores Genéticos

Cruce (Crossover): Punto único

$$hijo_1[i] = \begin{cases} padre_1[i] & \text{si } i < punto\_cruce \\ padre_2[i] & \text{si } i \geq punto\_cruce \end{cases}$$

Mutación: Intercambio aleatorio

$$P(mutar(c)) = p_m \cdot (1 + \frac{generacion}{max\_generaciones})$$

6.3 Búsqueda Local (Hill Climbing)

Vecindario: Intercambios de clases del mismo tipo

$$N(A) = \{A' : \exists c_i, c_j \in C, tipo(c_i) = tipo(c_j), A'(c_i) = A(c_j), A'(c_j) = A(c_i)\}$$

Criterio de aceptación: Descenso más pronunciado (steepest descent)

$$A_{nuevo} = \arg \min_{A' \in N(A)} E(A')$$

Referencias

1. Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*
2. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*
3. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*
4. Mitchell, T. M. (1997). *Machine Learning*

Pre-procesamiento y Pre-asignación

Preparación de Datos y Asignación de Prioridades

1. Introducción y Motivación

1.1 Problema Fundamental

En sistemas de optimización tradicionales, las restricciones duras y suaves se manejan mediante penalizaciones en la función objetivo. Sin embargo, este enfoque tiene limitaciones:

Problema con Penalizaciones:

- No garantiza 100% de cumplimiento
- Puede sacrificar restricciones duras por optimizar suaves
- Requiere ajuste cuidadoso de pesos

Solución Propuesta:

- **Pre-asignación forzada** de PRIORIDAD 1
- **Separación explícita** entre restricciones duras y suaves
- **Garantía matemática** de cumplimiento

1.2 Teorema de Separación

Teorema 1 (Separación de Prioridades):

Si las restricciones pueden dividirse en conjuntos disjuntos R_1 (duras) y R_2 (suaves), entonces existe una solución óptima que satisface completamente R_1.

Demostración:

Sea $S_{factible} = \{s : s \text{ satisface } R_1\}$ el conjunto de soluciones factibles.

1. Por definición, toda solución válida debe satisfacer R_1
2. El óptimo global $s^* \in S_{factible}$
3. Por lo tanto, s^* satisface R_1 completamente
4. La optimización de R_2 se realiza dentro de $S_{factible}$

Implicación: Podemos pre-asignar R_1 y luego optimizar R_2 sin afectar R_1.

2. Arquitectura del Sistema de Pre-asignación

2.1 Flujo de Procesamiento

```
graph TD
    A[Horario Inicial] --> B[Cargar Preferencias]
    B --> C[Identificar Clases P1]
    C --> D[Ordenar por Complejidad]
    D --> E[Asignar Forzadamente]
    E --> F{Conflictos?}
    F -->|Sí| G[Resolver Conflictos]
    F -->|No| H[Marcar Inmutables]
    G --> H
    H --> I[Guardar Horario Pre-asignado]
    I --> J[Guardar Índices Inmutables]
```

2.2 Componentes del Sistema

```
class PreAsignadorP1:
    def __init__(self):
        self.preferencias = {}
        self.indices_inmutables = []
        self.conflictos_resueltos = 0

    def ejecutar(self, horario_inicial):
        # 1. Cargar configuración
        self.cargar_preferencias()

        # 2. Identificar clases prioritarias
        clases_p1 = self.identificar_clases_p1(horario_inicial)

        # 3. Ordenar por complejidad
        clases_ordenadas = self.ordenar_por_complejidad(clases_p1)

        # 4. Asignar forzadamente
        horario_asignado = self.asignar_forzadamente(
            horario_inicial, clases_ordenadas
        )

        # 5. Marcar como inmutables
        self.marcar_inmutables(clases_p1)

        # 6. Guardar resultados
        self.guardar_resultados(horario_asignado)

        return horario_asignado
```

3. Identificación de Clases Prioritarias

3.1 Criterio de Prioridad

Una clase c es prioritaria si:

$$c \in P_1 \Leftrightarrow \exists profesor(c), materia(c) : prioridad(profesor, materia, tipo(c)) = \text{"Prioritario"}$$

Implementación:

```
def identificar_clases_p1(self, df):
    """
    Identifica todas las clases con PRIORIDAD 1
    """
    clases_p1 = []

    for idx, clase in df.iterrows():
        profesor = clase['Profesor']
        materia = clase['Materia']
        tipo = clase['Tipo_Salon']
```

3.1 Criterio de Prioridad (continuación 1)

```
        # Verificar en preferencias
        if profesor in self.preferencias:
            if materia in self.preferencias[profesor]['materias']:
                pref = self.preferencias[profesor]['materias'][materia]

                # Verificar teoría
                if tipo == 'Teoría':
                    if (pref.get('prioridad_teoria') == 'Prioritario' and
                        pref.get('salon_teoria') != 'Sin preferencia'):
                        clases_p1.append({
                            'idx': idx,
                            'clase': clase,
                            'salon_preferido': pref['salon_teoria']
                        })
```

3.1 Criterio de Prioridad (continuación 2)

```
                # Verificar laboratorio
                elif tipo == 'Laboratorio':
                    if (pref.get('prioridad_lab') == 'Prioritario' and
                        pref.get('salon_lab') != 'Sin preferencia'):
                        clases_p1.append({
                            'idx': idx,
                            'clase': clase,
                            'salon_preferido': pref['salon_lab']
                        })

    return clases_p1
```

3.2 Estadísticas de Prioridad

Para el caso del Instituto Tecnológico de Ciudad Madero:

```
Total clases: 680
Clases PRIORIDAD 1: 88 (13%)
├── Teoría: 58 (66%)
└── Laboratorio: 30 (34%)

Distribución por profesor:
├── PROFESOR 3: 10 clases
├── PROFESOR 4: 8 clases
├── PROFESOR 8: 15 clases
├── PROFESOR 9: 15 clases
├── PROFESOR 20: 9 clases
├── PROFESOR 21: 8 clases
├── PROFESOR 24: 5 clases
└── PROFESOR 26: 18 clases
```

4. Ordenamiento por Complejidad

4.1 Función de Complejidad

Para cada clase prioritaria c, definimos su complejidad:

$$complejidad(c) = w_1 \cdot num\_clases\_profesor(c) + w_2 \cdot conflictos\_potenciales(c) + w_3 \cdot \frac{1}{salones\_disponibles(c)}$$

Componentes:

1. **Número de clases del profesor:**

$$num\_clases\_profesor(c) = |\{c' : profesor(c') = profesor(c) \land c' \in P_1\}|$$

2. **Conflictos potenciales:**

$$conflictos\_potenciales(c) = |\{c' : mismo\_horario(c, c') \land salon\_preferido(c) = salon\_preferido(c')\}|$$

3. **Salones disponibles:**

$$salones\_disponibles(c) = |\{s : tipo(s) = tipo(c) \land s \notin S_{invalidos}\}|$$

Pesos:

- $w_1 = 10$ (más clases = más complejo)
- $w_2 = 5$ (más conflictos = más complejo)
- $w_3 = 3$ (menos opciones = más complejo)

4.2 Ordenamiento

```
def ordenar_por_complejidad(self, clases_p1):
    """
    Ordena clases prioritarias por complejidad (más complejo primero)
    """
    def calcular_complejidad(clase_info):
        clase = clase_info['clase']

        # Componente 1: Número de clases del profesor
        num_clases = sum(1 for c in clases_p1
                         if c['clase']['Profesor'] == clase['Profesor'])

        # Componente 2: Conflictos potenciales
        conflictos = sum(1 for c in clases_p1
                         if (c['clase']['Dia'] == clase['Dia'] and
                             c['clase']['Bloque_Horario'] == clase['Bloque_Horario'] and
                             c['salon_preferido'] == clase_info['salon_preferido']))

        # Componente 3: Inverso de salones disponibles
        salones_disp = len(self.obtener_salones_validos(clase))
        inv_salones = 1.0 / salones_disp if salones_disp > 0 else 10

        return 10 * num_clases + 5 * conflictos + 3 * inv_salones

    # Ordenar de mayor a menor complejidad
    return sorted(clases_p1, key=calcular_complejidad, reverse=True)
```

Justificación del Ordenamiento:

Lema 1 (Ordenamiento Óptimo):
Procesar clases más complejas primero minimiza la probabilidad de infactibilidad.

Demostración:

- Clases complejas tienen menos opciones alternativas
- Si se procesan al final, pueden no tener salones disponibles
- Procesarlas primero garantiza que al menos su opción preferida esté disponible
- Clases simples pueden adaptarse a salones restantes

5. Asignación Forzada

5.1 Algoritmo Principal

```python
def asignar_forzadamente(self, df, clases_ordenadas):
    """
    Asigna forzadamente cada clase a su salón preferido
    """
    df_resultado = df.copy()
    ocupacion = {}  # (dia, bloque, salon) -> idx

    for clase_info in clases_ordenadas:
        idx = clase_info['idx']
        clase = clase_info['clase']
        salon_preferido = clase_info['salon_preferido']

        # Clave de ocupación
        key = (clase['Dia'], clase['Bloque_Horario'], salon_preferido)

        # Verificar si hay conflicto
        if key in ocupacion:
            # Resolver conflicto
            exito = self.resolver_conflicto(
                df_resultado, idx, salon_preferido,
                ocupacion, clase_info
            )

            if not exito:
                print(f"⚠️  No se pudo asignar clase {idx}")
                continue

        # Asignar
        df_resultado.loc[idx, 'Salon'] = salon_preferido
        ocupacion[key] = idx
        self.indices_inmutables.append(idx)

    return df_resultado
```

5.2 Resolución de Conflictos

Cuando dos clases prioritarias quieren el mismo salón al mismo tiempo:

```
def resolver_conflicto(self, df, idx_nueva, salon, ocupacion, clase_info):
    """
    Resuelve conflicto desplazando clase no-prioritaria
    """
    key = (clase_info['clase']['Dia'],
           clase_info['clase']['Bloque_Horario'],
           salon)

    idx_ocupante = ocupacion[key]
    clase_ocupante = df.iloc[idx_ocupante]

    # Verificar si ocupante es prioritario
    if self.es_prioritaria(clase_ocupante):
        # Ambas son prioritarias: conflicto irresolvable
        print(f"❌ Conflicto entre dos clases prioritarias")
        return False
```

5.2 Resolución de Conflictos (continuación)

```
    # Desplazar ocupante a otro salón
    salones_alternativos = self.obtener_salones_validos(clase_ocupante)

    for salon_alt in salones_alternativos:
        key_alt = (clase_ocupante['Dia'],
                   clase_ocupante['Bloque_Horario'],
                   salon_alt)

        if key_alt not in ocupacion:
            # Mover ocupante
            df.loc[idx_ocupante, 'Salon'] = salon_alt
            ocupacion[key_alt] = idx_ocupante
            del ocupacion[key]
            return True

    # No hay salones alternativos
    print(f"⚠️  No se encontró salón alternativo")
    return False
```

Teorema 2 (Resolución de Conflictos):

Si existe al menos un salón válido libre para cada clase no-prioritaria, todo conflicto es resoluble.

Demostración:

1. Sea c_p clase prioritaria y c_n clase no-prioritaria en conflicto
2. c_n tiene al menos un salón válido s_{alt} libre (por hipótesis)
3. Mover c_n a s_{alt} libera el salón preferido de c_p
4. Asignar c_p a su salón preferido
5. Conflicto resuelto

6. Marcado de Índices Inmutables

6.1 Estructura de Datos

```
{
  "indices": [12, 45, 67, 89, ...],
  "total": 88,
  "timestamp": "2025-12-21T11:00:00",
  "version": "1.0"
}
```

6.2 Implementación

```
def marcar_inmutables(self, clases_p1):
    """
    Marca índices de clases P1 como inmutables
    """
    self.indices_inmutables = [c['idx'] for c in clases_p1]

    # Guardar en JSON
    data = {
        'indices': self.indices_inmutables,
        'total': len(self.indices_inmutables),
        'timestamp': datetime.now().isoformat(),
        'version': '1.0'
    }

    with open('datos_estructurados/indices_inmutables_p1.json', 'w') as f:
        json.dump(data, f, indent=2)
```

6.3 Invariante de Inmutabilidad

Definición:

$$\forall i \in I_{inmutables}, \forall t : s_t(c_i) = pref(c_i)$$

Donde:

- $I_{inmutables}$ = conjunto de índices inmutables
- s_t = solución en tiempo t
- $pref(c_i)$ = salón preferido de clase c_i

Verificación:

```
def verificar_invariante(self, df):
    """
    Verifica que todas las clases inmutables están en su salón preferido
    """
    violaciones = 0

    for idx in self.indices_inmutables:
```

7. Salidas del Sistema

7.1 Horario Pre-asignado

Archivo: `datos_estructurados/00_Horario_PreAsignado_P1.csv`

Formato:

```
Dia,Bloque_Horario,Materia,Grupo,Profesor,Salon,Es_Invalido,Tipo_Salon,Piso
Lunes,0700,LENGUAJES Y AUTÓMATAS I,2527A,PROFESOR 3,FFA,0,Teoría,1
...
```

Características:

- Todas las clases P1 en su salón preferido
- Clases no-P1 pueden estar en salones subóptimos
- Listo para ser optimizado por algoritmos

7.2 Índices Inmutables

Archivo: `datos_estructurados/indices_inmutables_p1.json`

Uso:

```
# En optimizadores
with open('datos_estructurados/indices_inmutables_p1.json') as f:
    data = json.load(f)
    indices_inmutables = set(data['indices'])

# Durante optimización
if idx in indices_inmutables:
    continue  # No modificar esta clase
```

8. Métricas y Validación

8.1 Métricas de Pre-asignación

```
Ejecución de pre_asignar_p1.py:
├── Clases prioritarias identificadas: 88
├── Clases asignadas exitosamente: 88
├── Conflictos resueltos: 12
├── Cumplimiento: 100%
└── Tiempo: 0.3s
```

8.2 Validación de Salida

```
def validar_salida(self, df):
    """
    Valida que la salida sea correcta
    """
    checks = {
        'total_clases': len(df) == 680,
        'p1_cumplimiento': self.verificar_p1(df) == 100.0,
        'sin_invalidos_p1': self.verificar_sin_invalidos_p1(df),
        'sin_conflictos_p1': self.verificar_sin_conflictos_p1(df)
    }

    return all(checks.values()), checks
```

9. Complejidad Computacional

9.1 Análisis Temporal

Identificación: $O(n)$ donde n = número de clases

Ordenamiento: $O(p \log p)$ donde p = clases prioritarias

Asignación: $O(p \cdot m)$ donde m = salones promedio por conflicto

Total: $O(n + p \log p + p \cdot m) = O(n)$ ya que $p \ll n$

Tiempo Real: ~0.3 segundos para 680 clases

9.2 Análisis Espacial

$$S = O(n + p) = O(n)$$

Memoria: ~5 MB

10. Ventajas del Enfoque

✅ **Garantía matemática** de 100% P1
✅ **Separación clara** entre restricciones
✅ **Simplifica optimización** posterior
✅ **Rápido** (<1 segundo)
✅ **Robusto** ante cambios en preferencias
✅ **Verificable** mediante invariante

11. Casos Especiales

11.1 Conflictos Irresolvables

Si dos clases P1 quieren el mismo salón al mismo tiempo:

Solución Manual:

1. Identificar el conflicto
2. Contactar a profesores involucrados
3. Negociar cambio de horario o salón
4. Actualizar preferencias

11.2 Salones Insuficientes

Si no hay suficientes salones del tipo requerido:

Solución:

1. Identificar clases afectadas
2. Evaluar posibilidad de usar salones alternativos
3. Ajustar configuración de materias
4. Re-ejecutar pre-asignación

12. Integración con Optimizadores

12.1 Carga en Optimizadores

```
# En optimizador_greedy.py, optimizador_ml.py, optimizador_genetico.py
def __init__(self):
    # Cargar horario pre-asignado
    self.df_inicial = pd.read_csv(
        'datos_estructurados/00_Horario_PreAsignado_P1.csv'
    )

    # Cargar índices inmutables
    with open('datos_estructurados/indices_inmutables_p1.json') as f:
        data = json.load(f)
        self.indices_inmutables = set(data['indices'])
```

12.2 Protección Durante Optimización

```
# En operadores de optimización
def aplicar_operador(self, solucion):
    for idx in range(len(solucion)):
        # Proteger inmutables
        if idx in self.indices_inmutables:
            continue

        # Aplicar modificación solo a no-inmutables
        solucion[idx] = nueva_asignacion(idx)
```

13. Conclusiones

El sistema de pre-asignación:

1. **Garantiza** 100% cumplimiento de PRIORIDAD 1
2. **Simplifica** el problema de optimización
3. **Separa** restricciones duras de suaves
4. **Permite** que optimizadores se enfoquen en P2 y P3
5. **Proporciona** base sólida para todo el sistema

Es un componente **crítico** que hace posible el enfoque de prioridades jerárquicas.

Referencias

1. Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.
2. Rossi, F., Van Beek, P., & Walsh, T. (2006). *Handbook of Constraint Programming*. Elsevier.
3. Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Algoritmo Greedy + Hill Climbing

Construcción Voraz y Refinamiento Local

1. Introducción y Fundamento Teórico

1.1 Paradigma de Algoritmos Voraces

Un **algoritmo voraz (greedy)** construye una solución tomando decisiones localmente óptimas en cada paso, con la esperanza de que estas decisiones conduzcan a un óptimo global.

Definición Formal:

Sea $S = \{s_1, s_2, \ldots, s_n\}$ el conjunto de decisiones a tomar. Un algoritmo voraz:

1. Inicializa $Sol = \emptyset$
2. Para cada decisión s_i en orden:
    - Selecciona $s_i^* = \arg\min_{s \in candidatos} costo(s)$
    - Actualiza $Sol = Sol \cup \{s_i^*\}$
3. Retorna Sol

Teorema del Algoritmo Voraz:

Un algoritmo voraz produce una solución óptima si el problema exhibe la propiedad de elección voraz y subestructura óptima.

Aplicación a Asignación de Salones:

- **Elección voraz:** Asignar cada clase al salón de menor costo incremental
- **Subestructura:** La solución óptima contiene soluciones óptimas a subproblemas
- **Limitación:** Nuestro problema NO garantiza optimalidad global con greedy puro

1.2 Búsqueda Local: Hill Climbing

Hill Climbing es un algoritmo de búsqueda local que mejora iterativamente una solución explorando su vecindario.

Definición Formal:

$$
\begin{aligned}
&\text{Inicializar: } s = s_0 \\
&\text{Repetir:} \\
&\quad s' = \arg\min_{s'' \in N(s)} f(s'') \\
&\quad \text{Si } f(s') < f(s) : \\
&\quad\quad s = s' \\
&\quad \text{Sino:} \\
&\quad\quad \text{retornar } s
\end{aligned}
$$

Donde:

- s = solución actual
- $N(s)$ = vecindario de s
- $f(s)$ = función objetivo (energía)

Propiedades:

- **Convergencia:** Garantizada a un óptimo local
- **Complejidad:** $O(k \cdot |N(s)|)$ donde k = iteraciones
- **Limitación:** Puede quedar atrapado en óptimos locales

1.3 Estrategia Híbrida

Combinamos ambos enfoques:

```
Solución = Greedy(problema)  # Construcción rápida
Solución = HillClimbing(Solución)  # Refinamiento local
```

Ventajas del Híbrido:

- Greedy proporciona punto de partida de calidad
- Hill Climbing escapa de decisiones voraces subóptimas
- Balance entre velocidad y calidad

2. Fase 1: Construcción Voraz

2.1 Ordenamiento de Clases

Objetivo: Procesar clases en orden que maximice probabilidad de buenas asignaciones.

Criterio de Ordenamiento Multi-nivel:

Para dos clases c_i y c_j, definimos $c_i \prec c_j$ si:

$$\begin{cases} prioridad(c_i) > prioridad(c_j) & (1^\circ \text{ criterio}) \\ \text{o si } prioridad(c_i) = prioridad(c_j): & \\ \quad num\_restricciones(c_i) > num\_restricciones(c_j) & (2^\circ \text{ criterio}) \\ \text{o si } num\_restricciones(c_i) = num\_restricciones(c_j): & \\ \quad |salones\_validos(c_i)| < |salones\_validos(c_j)| & (3^\circ \text{ criterio}) \end{cases}$$

Justificación Teórica:

Lema 1 (Ordenamiento Óptimo):
Procesar clases con más restricciones primero minimiza la probabilidad de infactibilidad.

Demostración:
Sea R_i el conjunto de restricciones de c_i y S_i el conjunto de salones válidos.

- Si $|S_i| < |S_j|$, entonces c_i tiene menos opciones
- Procesar c_i primero garantiza que al menos una opción esté disponible
- Procesar c_j después aún deja $|S_j| - 1 \geq |S_i|$ opciones
- Por lo tanto, el orden minimiza conflictos

Implementación:

```
def ordenar_clases(self, df):
    """
    Ordena clases por criterio multi-nivel
    """
    df_ordenado = df.copy()

    # Calcular métricas para cada clase
    df_ordenado['prioridad_num'] = df_ordenado.apply(
        lambda row: self.obtener_prioridad(row), axis=1
    )

    df_ordenado['num_restricciones'] = df_ordenado.apply(
        lambda row: self.contar_restricciones(row), axis=1
    )

    df_ordenado['num_salones_validos'] = df_ordenado.apply(
        lambda row: len(self.obtener_salones_validos(row)), axis=1
    )

    # Ordenar por criterios
    df_ordenado = df_ordenado.sort_values(
        by=['prioridad_num', 'num_restricciones', 'num_salones_validos'],
        ascending=[False, False, True]
    )

    return df_ordenado
```

2.2 Función de Score Voraz

Para cada par $(clase, salon)$, calculamos un score que estima el costo incremental:

$$score(c_i, s_j) = \sum_{k=1}^{m} w_k \cdot componente_k(c_i, s_j)$$

Componentes del Score:

A. Distancia al Último Salón del Profesor

$$componente_1(c_i, s_j) = \begin{cases} 0 & \text{si } c_i \text{ es primera clase del profesor} \\ d(ultimo\_salon(profesor(c_i)), s_j) & \text{en otro caso} \end{cases}$$

Donde $d(\cdot, \cdot)$ es la función de distancia definida anteriormente:

$$d(s_a, s_b) = \begin{cases} 0 & \text{si } s_a = s_b \\ 1 & \text{si } piso(s_a) = piso(s_b) \land s_a \neq s_b \\ 10 & \text{si } piso(s_a) \neq piso(s_b) \end{cases}$$

B. Ocupación del Salón

$$componente_2(c_i, s_j) = \frac{|uso\_actual(s_j)|}{|uso\_maximo(s_j)|}$$

Interpretación: Preferir salones menos utilizados para balancear carga.

C. Penalización por Tipo Incorrecto

$$componente_3(c_i, s_j) = \begin{cases} \infty & \text{si } tipo(c_i) \neq tipo(s_j) \\ 0 & \text{en otro caso} \end{cases}$$

D. Bonus por Preferencia

$$componente_4(c_i, s_j) = \begin{cases} -B & \text{si } s_j = salon\_preferido(c_i) \\ 0 & \text{en otro caso} \end{cases}$$

Donde $B > 0$ es un bonus que incentiva cumplir preferencias.

E. Penalización por Salón Inválido

$$componente_5(c_i, s_j) = \begin{cases} \infty & \text{si } s_j \in S_{invalidos} \\ 0 & \text{en otro caso} \end{cases}$$

Score Total:

$$score(c_i, s_j) = 10 \cdot componente_1 + 5 \cdot componente_2 + componente_3 - 50 \cdot componente_4 + componente_5$$

Pesos Justificados:

- Distancia (10): Impacto directo en movilidad
- Ocupación (5): Balanceo de recursos
- Tipo (∞): Restricción dura
- Preferencia (-50): Incentivo fuerte
- Inválido (∞): Restricción dura

2.3 Algoritmo de Construcción

```python
def construccion_greedy(self, df):
    """
    Construye solución inicial mediante selección voraz
    """
    solucion = {}
    ocupacion = {}  # (dia, bloque, salon) -> idx_clase

    # Ordenar clases
    df_ordenado = self.ordenar_clases(df)

    for idx, clase in df_ordenado.iterrows():
        # Obtener salones candidatos
        candidatos = self.obtener_salones_validos(clase)

        # Filtrar salones ocupados en este horario
        candidatos_libres = [
            s for s in candidatos
            if (clase['Dia'], clase['Bloque_Horario'], s) not in ocupacion
        ]

        if not candidatos_libres:
            # Caso excepcional: forzar asignación
            mejor_salon = candidatos[0]
        else:
            # Selección voraz: minimizar score
            mejor_salon = None
            mejor_score = float('inf')

            for salon in candidatos_libres:
                score = self.calcular_score(clase, salon, solucion)

                if score < mejor_score:
                    mejor_score = score
                    mejor_salon = salon

        # Asignar
        solucion[idx] = mejor_salon
        ocupacion[(clase['Dia'], clase['Bloque_Horario'], mejor_salon)] = idx

    return solucion
```

2.4 Análisis de Complejidad - Fase 1

Ordenamiento:

$$T_{sort} = O(n \log n)$$

Cálculo de salones válidos por clase:

$$T_{valid} = O(n \cdot m)$$

Donde m = número promedio de salones candidatos.

Selección voraz:

$$T_{greedy} = O(n \cdot m)$$

Total Fase 1:

$$T_1 = O(n \log n + n \cdot m) = O(n \cdot m)$$

Para $n = 680$, $m \approx 5$:

$$T_1 \approx 3400 \text{ operaciones}$$

3. Fase 2: Hill Climbing

3.1 Definición del Vecindario

El vecindario $N(s)$ de una solución s se define como:

$$N(s) = \{s' : s' \text{ difiere de } s \text{ en exactamente un intercambio vá lido}\}$$

Intercambio Válido:

Para clases c_i y c_j:

$$intercambio\_valido(c_i, c_j) \Leftrightarrow \begin{cases} tipo(c_i) = tipo(c_j) & \text{(mismo tipo)} \\ i \notin I_{inmutables} \land j \notin I_{inmutables} & \text{(no protegidas)} \\ \neg conflicto(c_i, s(c_j)) \land \neg conflicto(c_j, s(c_i)) & \text{(sin conflictos)} \end{cases}$$

Donde $I_{inmutables}$ es el conjunto de índices de clases PRIORIDAD 1.

Tamaño del Vecindario:

$$|N(s)| = \binom{n_{modificables}}{2} \approx \frac{n_{modificables}^2}{2}$$

Para $n_{modificables} \approx 600$:

$$|N(s)| \approx 180,000 \text{ vecinos posibles}$$

3.2 Función de Energía

La función de energía $E(s)$ cuantifica la calidad de una solución:

$$E(s) = E_{movimientos}(s) + E_{pisos}(s) + E_{distancia}(s) + E_{penalizaciones}(s)$$

A. Energía por Movimientos

$$E_{movimientos}(s) = w_m \cdot \sum_{p \in P} \max(0, |salones\_usados(p, s)| - 1)$$

Donde:

$$salones\_usados(p, s) = \{s(c) : c \in C_p\}$$

Ejemplo Numérico:

- Profesor tiene 5 clases en salones: {FF1, FF2, FF1, FF3, FF2}
- $salones\_usados = \{FF1, FF2, FF3\}$
- $|salones\_usados| = 3$
- $E_{movimientos} = 10 \cdot (3 - 1) = 20$

B. Energía por Cambios de Piso

Sea $C_p^{sorted} = [c_1, c_2, \ldots, c_k]$ las clases del profesor p ordenadas cronológicamente:

$$E_{pisos}(s) = w_p \cdot \sum_{p \in P} \sum_{i=1}^{|C_p|-1} \mathbb{1}\left[piso(s(c_i)) \neq piso(s(c_{i+1}))\right]$$

C. Energía por Distancia

$$E_{distancia}(s) = w_d \cdot \sum_{p \in P} \sum_{i=1}^{|C_p|-1} d(s(c_i), s(c_{i+1}))$$

D. Penalizaciones

$$E_{penalizaciones}(s) = 1000 \cdot |\{c : s(c) \in S_{invalidos}\}| + 500 \cdot |\{(c_i, c_j) : conflicto(c_i, c_j, s)\}| + 300 \cdot |\{c : tipo(c) \neq tipo(s(c))\}|$$

Pesos Utilizados:

- $w_m = 10$ (movimientos)
- $w_p = 5$ (cambios de piso)
- $w_d = 3$ (distancia)

3.3 Estrategia de Búsqueda

Implementamos **Steepest Descent Hill Climbing**:

```
def hill_climbing(self, solucion, df):
    """
    Mejora solución mediante búsqueda local
    """
    mejor_solucion = solucion.copy()
    mejor_energia = self.calcular_energia(mejor_solucion, df)

    for iteracion in range(self.max_iter_hc):
        mejoro = False
        indices = list(solucion.keys())

        # Generar vecinos aleatorios
        for _ in range(self.intentos_por_iter):
            # Seleccionar par aleatorio
            idx1, idx2 = random.sample(indices, 2)

            # Verificar que no sean inmutables
            if idx1 in self.indices_inmutables or idx2 in self.indices_inmutables:
                continue
```

```
            # Verificar mismo tipo
            if self.tipos_por_idx[idx1] != self.tipos_por_idx[idx2]:
                continue

            # Crear vecino
            vecino = mejor_solucion.copy()
            vecino[idx1], vecino[idx2] = vecino[idx2], vecino[idx1]

            # Evaluar
            energia_vecino = self.calcular_energia(vecino, df)

            # Aceptar si mejora
            if energia_vecino < mejor_energia:
                mejor_solucion = vecino
                mejor_energia = energia_vecino
                mejoro = True
                break  # Steepest descent: aceptar primera mejora

        # Criterio de parada
        if not mejoro:
            print(f"   Convergió en iteración {iteracion}")
            break

    return mejor_solucion
```

3.4 Criterio de Parada

El algoritmo se detiene cuando:

$$\forall s' \in N_{muestreado}(s) : E(s') \geq E(s)$$

Donde $N_{muestreado}$ es un subconjunto aleatorio de $N(s)$.

Teorema 2 (Convergencia de Hill Climbing):

El algoritmo de Hill Climbing converge a un óptimo local en tiempo finito.

Demostración:

1. El espacio de soluciones es finito: $|S| = m^n$
2. La energía es discreta y acotada inferiormente: $E(s) \geq 0$
3. Cada iteración reduce estrictamente E o termina
4. No puede haber ciclos (energía siempre decrece)
5. Por lo tanto, converge en $\leq |S|$ iteraciones

Cota Superior Práctica:

Con muestreo aleatorio de k vecinos por iteración:

$$T_{convergencia} \leq \frac{E_{inicial} - E_{optimo\_local}}{mejora\_promedio} \cdot k$$

Empíricamente: $\approx 20 - 50$ iteraciones

3.5 Análisis de Complejidad - Fase 2

Por iteración:

$$T_{iter} = k \cdot (T_{generar} + T_{evaluar})$$

Donde:

- k = intentos por iteración (50)
- $T_{generar} = O(1)$ (intercambio simple)
- $T_{evaluar} = O(n)$ (recalcular energía)

$$T_{iter} = O(k \cdot n)$$

Total Fase 2:

$$T_2 = O(max\_iter \cdot k \cdot n)$$

Para $max\_iter = 100, k = 50, n = 680$:

$$T_2 \approx 3.4 \times 10^6 \text{ operaciones}$$

4. Protección de PRIORIDAD 1

4.1 Mecanismo de Inmutabilidad

Definición:

Sea $I \subset \{0, 1, \ldots, n-1\}$ el conjunto de índices de clases PRIORIDAD 1:

$$I = \{i : c_i \in P_1\}$$

Invariante:

$$\forall i \in I, \forall t : s_t(c_i) = pref(c_i)$$

Donde s_t es la solución en el tiempo t.

Implementación:

```
def cargar_indices_inmutables(self):
    """
    Carga índices de clases que no deben modificarse
    """
    try:
        with open('datos_estructurados/indices_inmutables_p1.json', 'r') as f:
            data = json.load(f)
            self.indices_inmutables = set(data.get('indices', []))
            print(f"✅ Índices inmutables cargados: {len(self.indices_inmutables)} clases")
    except FileNotFoundError:
        self.indices_inmutables = set()
        print("⚠️  No se encontró archivo de índices inmutables")
```

4.2 Verificación en Hill Climbing

```
# En cada intercambio propuesto
if idx1 in self.indices_inmutables or idx2 in self.indices_inmutables:
    continue  # Saltar este intercambio
```

Teorema 3 (Preservación de P1):
Si s_0 satisface PRIORIDAD 1 y se respetan los índices inmutables, entonces s_t satisface PRIORIDAD 1 para todo t.

Demostración:
Por inducción en t:

- **Caso base ($t = 0$):** s_0 satisface P1 por construcción (pre-asignación)
- **Paso inductivo:** Asumimos s_t satisface P1
  - Hill Climbing solo modifica índices $\notin I$
  - Por lo tanto, $\forall i \in I : s_{t+1}(c_i) = s_t(c_i) = pref(c_i)$
  - Luego s_{t+1} satisface P1

4.3 Corrección Post-Optimización

Como medida de seguridad adicional:

```
def corregir_prioridades_final(self, solucion, df):
    """
    Garantiza 100% cumplimiento de PRIORIDAD 1
    """
    correcciones = 0

    for idx in self.indices_inmutables:
        clase = df.iloc[idx]
        salon_esperado = self.obtener_salon_preferido(clase)

        if solucion[idx] != salon_esperado:
            solucion[idx] = salon_esperado
            correcciones += 1

    if correcciones > 0:
        print(f"   ✅ {correcciones} clases corregidas a salón prioritario")

    return solucion
```

5. Optimizaciones y Mejoras

5.1 Caching de Energía

Problema: Recalcular energía completa es $O(n)$

Solución: Calcular solo cambio incremental

```
def calcular_delta_energia(self, solucion, idx1, idx2, df):
    """
    Calcula cambio de energía por intercambio
    """
    # Solo recalcular para profesores afectados
    prof1 = df.iloc[idx1]['Profesor']
    prof2 = df.iloc[idx2]['Profesor']

    delta = 0
    delta += self.delta_movimientos(prof1, idx1, idx2, solucion, df)

    if prof1 != prof2:
        delta += self.delta_movimientos(prof2, idx1, idx2, solucion, df)

    return delta
```

Complejidad: $O(|C_{prof}|)$ en lugar de $O(n)$

5.2 Tabu Search

Evitar ciclos manteniendo lista de movimientos prohibidos:

```
def hill_climbing_tabu(self, solucion, df):
    tabu_list = deque(maxlen=10)  # Últimos 10 movimientos

    for iteracion in range(self.max_iter_hc):
        # ... generar vecino ...

        movimiento = (idx1, idx2)
        if movimiento in tabu_list:
            continue  # Prohibido

        # ... evaluar y aceptar ...

        if mejoro:
            tabu_list.append(movimiento)
```

5.3 Simulated Annealing

Aceptar ocasionalmente movimientos que empeoran:

$$P(aceptar) = \begin{cases} 1 & \text{si } \Delta E < 0 \\ e^{-\Delta E/T} & \text{si } \Delta E \geq 0 \end{cases}$$

Donde T es la "temperatura" que decrece con el tiempo:

$$T_t = T_0 \cdot \alpha^t, \quad 0 < \alpha < 1$$

6. Resultados y Análisis

6.1 Métricas de Rendimiento

```
FASE 1: Construcción Greedy
├── Tiempo: 2.3s
├── Energía inicial: 82,948
└── Clases asignadas: 680/680

FASE 2: Hill Climbing
├── Tiempo: 27.0s
├── Iteraciones: 54
├── Energía final: 82,716
├── Mejora: 232 (-0.28%)
└── Convergencia: Sí

RESULTADOS FINALES:
├── Movimientos: 314 (-12.0% vs inicial)
├── Cambios piso: 206 (-28.2% vs inicial)
├── Distancia: 1,951 (-31.5% vs inicial)
└── PRIORIDAD 1: 100% (98/98)
```

6.2 Comparación con Otros Algoritmos

Métrica	Greedy+HC	ML	Genético	Óptimo Teórico
Tiempo	29.3s	15.8s	73.9s	∞
Movimientos	314	365	378	≥280
Cambios piso	206	223	286	≥180
Distancia	1,951	1,821	2,413	≥1,500
Gap vs óptimo	~5%	~3%	~10%	0%

Nota: Óptimo teórico es estimación basada en cotas inferiores.

6.3 Análisis de Sensibilidad

Variación de Parámetros:

max_iter	Tiempo	Energía Final	Mejora
10	5.2s	82,850	Baja
50	15.1s	82,730	Media
100	29.3s	82,716	Alta
200	58.7s	82,714	Marginal

Conclusión: 100 iteraciones es el punto óptimo (rendimientos decrecientes después).

7. Ventajas y Limitaciones

7.1 Ventajas

✅ **Velocidad:** Construcción inicial muy rápida ($O(n \log n)$)

✅ **Simplicidad:** Fácil de entender e implementar

✅ **Determinismo:** Resultados reproducibles (con semilla fija)

✅ **Escalabilidad:** Complejidad lineal en n

✅ **Robustez:** Siempre encuentra solución factible

✅ **Balance:** Buena relación calidad/tiempo

7.2 Limitaciones

❌ **Óptimos Locales:** No garantiza óptimo global

❌ **Sensibilidad al Orden:** Orden inicial afecta resultado

❌ **Exploración Limitada:** Vecindario pequeño

❌ **Plateau:** Puede estancarse en mesetas

❌ **Parámetros:** Requiere ajuste de pesos

7.3 Cuándo Usar Greedy+HC

Recomendado cuando:

- Se requiere solución rápida (< 1 minuto)
- Calidad media-alta es suficiente
- Problema es relativamente estructurado
- Hay buena heurística de ordenamiento

No recomendado cuando:

- Se requiere óptimo global garantizado
- Hay tiempo ilimitado para búsqueda
- Problema es altamente irregular
- Espacio de búsqueda es muy pequeño

8. Extensiones Futuras

8.1 Variable Neighborhood Search (VNS)

Usar múltiples definiciones de vecindario:

$$N_1(s) = \text{intercambios simples}$$
$$N_2(s) = \text{intercambios dobles}$$
$$N_3(s) = \text{rotaciones ciclicas}$$

8.2 Iterated Local Search (ILS)

Aplicar perturbaciones para escapar óptimos locales:

```
s = ConstructionGreedy()
s = HillClimbing(s)

for i in range(max_restarts):
    s' = Perturb(s)
    s' = HillClimbing(s')
    if E(s') < E(s):
        s = s'

return s
```

8.3 Guided Local Search

Penalizar características de óptimos locales visitados:

$$E_{guided}(s) = E(s) + \lambda \sum_i p_i \cdot feature_i(s)$$

Donde p_i aumenta cada vez que se visita un óptimo local con $feature_i$.

9. Conclusiones

El algoritmo Greedy + Hill Climbing ofrece un **excelente balance** entre:

- **Velocidad de ejecución** (~30s)
- **Calidad de solución** (top 2 de 4 algoritmos)
- **Simplicidad de implementación**
- **Robustez y confiabilidad**

Es la **opción recomendada** para uso general en el sistema de asignación de salones.

Referencias

1. Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
3. Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*. Dover.
4. Aarts, E., & Lenstra, J. K. (2003). *Local Search in Combinatorial Optimization*. Princeton University Press.
5. Hoos, H. H., & Stützle, T. (2004). *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann.

Algoritmo de Machine Learning

Aprendizaje Supervisado para Asignación Óptima

1. Introducción y Fundamento Teórico

El enfoque de Machine Learning para la asignación de salones se basa en la **hipótesis de aprendizaje supervisado**: si existe un patrón en las asignaciones óptimas previas, un modelo puede aprender a predecir asignaciones de calidad similar sin necesidad de búsqueda exhaustiva.

Ventaja Principal: Una vez entrenado, el modelo puede generar soluciones en tiempo casi constante $O(n \cdot d)$ donde d es la profundidad del árbol, comparado con $O(n^2)$ o peor de otros métodos.

1.2 Arquitectura del Sistema ML

El sistema utiliza un **ensemble de dos modelos complementarios**:

1. **Random Forest Classifier:** Predice el salón óptimo para cada clase
2. **Gradient Boosting Regressor:** Estima la calidad de cada asignación

```
Entrada: Características de la clase
   ↓
[Random Forest] → Predicción de salón
   ↓
[Gradient Boosting] → Score de calidad
   ↓
Selección del mejor salón válido
   ↓
Salida: Asignación optimizada
```

2. Extracción de Características (Feature Engineering)

2.1 Vector de Características

Para cada clase c_i, construimos un vector de características $\mathbf{x}_i \in \mathbb{R}^d$:

$$\mathbf{x}_i = [x_1, x_2, \ldots, x_d]^T$$

Categorías de Características:

A. Características Temporales

$$
\begin{aligned}
x_1 &= dia\_semana(c_i) && \in \{0, 1, 2, 3, 4\} && \text{(Lun-Vie)} \\
x_2 &= bloque\_horario(c_i) && \in \{7, 8, \ldots, 21\} && \text{(Hora inicio)} \\
x_3 &= es\_manana(c_i) && \in \{0, 1\} && \text{(Antes de 12:00)} \\
x_4 &= es\_tarde(c_i) && \in \{0, 1\} && \text{(12:00-18:00)} \\
x_5 &= es\_noche(c_i) && \in \{0, 1\} && \text{(Despues 18:00)}
\end{aligned}
$$

B. Características de Materia

$$
\begin{aligned}
x_6 &= materia\_encoded(c_i) && \in \{0, 1, \ldots, |M| - 1\} && \text{(Label encoding)} \\
x_7 &= tipo\_clase(c_i) && \in \{0, 1\} && \text{(0=Teoria, 1=Lab)} \\
x_8 &= num\_estudiantes(c_i) && \in \mathbb{N} && \text{(Tamano grupo)}
\end{aligned}
$$

C. Características de Profesor

$$
\begin{aligned}
x_9 &= profesor\_encoded(c_i) && \in \{0, 1, \ldots, |P| - 1\} \\
x_{10} &= tiene\_preferencia(c_i) && \in \{0, 1\} \\
x_{11} &= prioridad\_preferencia(c_i) && \in \{0, 1, 2, 3\}
\end{aligned}
$$

D. Características Contextuales

$$
\begin{aligned}
x_{12} &= num\_clases\_profesor\_dia(c_i) && \in \mathbb{N} \\
x_{13} &= posicion\_en\_dia(c_i) && \in \{1, 2, \ldots, k\} \\
x_{14} &= salon\_clase\_anterior(c_i) && \in \{0, 1, \ldots, |S| - 1\} \\
x_{15} &= piso\_clase\_anterior(c_i) && \in \{0, 1\}
\end{aligned}
$$

E. Características de Grupo

$$
\begin{aligned}
x_{16} &= semestre(c_i) && \in \{1, 2, \ldots, 9\} \\
x_{17} &= es\_primer\_semestre(c_i) && \in \{0, 1\} \\
x_{18} &= grupo\_encoded(c_i) && \in \{0, 1, \ldots, |G| - 1\}
\end{aligned}
$$

Total de características: $d \approx 18 - 25$ (dependiendo de encoding)

2.2 Encoding de Variables Categóricas

Label Encoding:

Para variables ordinales (materia, profesor, grupo):

$$encode(categoria) = \{categoria_1 \to 0, categoria_2 \to 1, \ldots, categoria_n \to n-1\}$$

One-Hot Encoding (alternativa):

Para k categorías, crear k variables binarias:

$$\mathbf{x}_{one-hot} = [0, 0, \ldots, 1, \ldots, 0]^T$$

Donde el 1 está en la posición correspondiente a la categoría.

Decisión de Diseño: Usamos Label Encoding para reducir dimensionalidad, ya que Random Forest maneja bien variables categóricas ordinales.

2.3 Normalización

Para características numéricas continuas:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

Donde:

- μ_i = media de la característica i
- σ_i = desviación estándar de la característica i

Nota: Random Forest no requiere normalización estricta, pero mejora la interpretabilidad.

3. Modelo 1: Random Forest Classifier

3.1 Fundamento Teórico

Un Random Forest es un **ensemble de árboles de decisión** que combina predicciones mediante votación mayoritaria (clasificación) o promedio (regresión).

Definición Formal:

Sea $\{h(\mathbf{x}, \Theta_k)\}_{k=1}^{K}$ un conjunto de K árboles de decisión, donde Θ_k son parámetros aleatorios (subset de features y datos). El Random Forest predice:

$$\hat{y} = \text{mode}\{h(\mathbf{x}, \Theta_1), h(\mathbf{x}, \Theta_2), \ldots, h(\mathbf{x}, \Theta_K)\}$$

3.2 Algoritmo de Entrenamiento

```
def entrenar_random_forest(X_train, y_train):
    """
    X_train: matriz n × d de características
    y_train: vector n de salones asignados (labels)
    """
    forest = []

    for k in range(K):  # K = número de árboles
        # 1. Bootstrap sampling
        indices = sample_with_replacement(n, n)
        X_boot = X_train[indices]
        y_boot = y_train[indices]

        # 2. Entrenar árbol con subset aleatorio de features
        tree = DecisionTree(
            max_depth=d_max,
            min_samples_split=s_min,
            max_features=sqrt(d)  # Regla empírica
        )
        tree.fit(X_boot, y_boot)
        forest.append(tree)

    return forest
```

3.3 Construcción de Árbol de Decisión

Algoritmo CART (Classification and Regression Trees):

```
función construir_arbol(X, y, profundidad):
    si profundidad == max_depth o |y| < min_samples:
        retornar hoja con clase mayoritaria

    mejor_ganancia = 0
    mejor_split = None

    para cada característica f en subset_aleatorio(features):
        para cada valor v en valores_unicos(X[:, f]):
            # Dividir datos
            izq = {(x, y) : x[f] <= v}
            der = {(x, y) : x[f] > v}
```

3.3 Construcción de Árbol (continuación)

```
            # Calcular ganancia de información
            ganancia = calcular_ganancia(y, izq, der)

            si ganancia > mejor_ganancia:
                mejor_ganancia = ganancia
                mejor_split = (f, v)

    si mejor_ganancia == 0:
        retornar hoja

    # Crear nodo interno
    nodo.feature = mejor_split[0]
    nodo.threshold = mejor_split[1]
    nodo.izquierdo = construir_arbol(izq, profundidad + 1)
    nodo.derecho = construir_arbol(der, profundidad + 1)

    retornar nodo
```

3.4 Criterio de División: Gini Impurity

Para clasificación, usamos el **índice de Gini**:

$$Gini(S) = 1 - \sum_{i=1}^{|C|} p_i^2$$

Donde:

- S = conjunto de muestras en el nodo
- C = conjunto de clases (salones)
- p_i = proporción de muestras de clase i en S

Ganancia de Información:

$$Ganancia(S, f, v) = Gini(S) - \left(\frac{|S_{izq}|}{|S|} Gini(S_{izq}) + \frac{|S_{der}|}{|S|} Gini(S_{der}) \right)$$

Objetivo: Maximizar la ganancia (minimizar impureza después del split)

3.5 Predicción

Para una nueva clase $\mathbf{x}_{new}$:

$$\hat{y}_{RF} = \arg\max_{s \in S} \sum_{k=1}^{K} \mathbb{1}\left[h_k(\mathbf{x}_{new}) = s\right]$$

Interpretación: El salón que recibe más "votos" de los árboles individuales.

3.6 Hiperparámetros

```
RandomForestClassifier(
    n_estimators=100,          # Número de árboles
    max_depth=20,              # Profundidad máxima
    min_samples_split=5,       # Mínimo para dividir nodo
    min_samples_leaf=2,        # Mínimo en hojas
    max_features='sqrt',       # sqrt(d) features por split
    random_state=42,           # Reproducibilidad
    n_jobs=-1                  # Paralelización
)
```

Justificación de Valores:

- `n_estimators=100` : Balance entre precisión y tiempo
- `max_depth=20` : Evita overfitting en dataset pequeño
- `min_samples_split=5` : Previene splits en ruido
- `max_features='sqrt'` : Regla empírica de Breiman

4. Modelo 2: Gradient Boosting Regressor

4.1 Fundamento Teórico

Gradient Boosting construye un **ensemble aditivo** de árboles débiles que minimizan una función de pérdida mediante descenso de gradiente.

Idea Central: Cada árbol nuevo corrige los errores del ensemble anterior.

4.2 Algoritmo de Gradient Boosting

Formulación Matemática:

Queremos aproximar una función $F^*(\mathbf{x})$ que minimiza la pérdida esperada:

$$F^*(\mathbf{x}) = \arg\min_F \mathbb{E}_{y,\mathbf{x}}[L(y, F(\mathbf{x}))]$$

Donde L es la función de pérdida (ej: MSE para regresión).

Algoritmo:

```
Inicializar: F_0(x) = arg min_γ Σ L(y_i, γ)

Para m = 1 hasta M:
    1. Calcular pseudo-residuos:
       r_im = -[∂L(y_i, F(x_i))/∂F(x_i)]_{F=F_{m-1}}

    2. Entrenar árbol h_m(x) para predecir r_i

    3. Calcular multiplicador óptimo:
       γ_m = arg min_γ Σ L(y_i, F_{m-1}(x_i) + γ h_m(x_i))

    4. Actualizar modelo:
       F_m(x) = F_{m-1}(x) + η γ_m h_m(x)

Retornar F_M(x)
```

Donde:

- M = número de iteraciones (árboles)
- η = learning rate (tasa de aprendizaje)
- h_m = árbol débil en iteración m

4.3 Función de Pérdida: Mean Squared Error

Para regresión de calidad de asignación:

$$L(y, F(\mathbf{x})) = \frac{1}{2}(y - F(\mathbf{x}))^2$$

Gradiente:

$$\frac{\partial L}{\partial F} = -(y - F(\mathbf{x})) = -residuo$$

Por lo tanto, los pseudo-residuos son simplemente los residuos reales.

4.4 Regularización

L2 Regularization (Ridge):

$$L_{reg}(y, F(\mathbf{x})) = L(y, F(\mathbf{x})) + \lambda \sum_{m=1}^{M} ||h_m||^2$$

Shrinkage (Learning Rate):

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta \cdot \gamma_m \cdot h_m(\mathbf{x})$$

Donde $0 < \eta \leq 1$ controla la contribución de cada árbol.

Subsampling:

Usar solo una fracción ρ de los datos para entrenar cada árbol:

$$subsample\_size = \rho \cdot n, \quad 0 < \rho \leq 1$$

4.5 Hiperparámetros

```
GradientBoostingRegressor(
    n_estimators=100,          # Número de boosting stages
    learning_rate=0.1,         # Shrinkage η
    max_depth=5,               # Profundidad de árboles débiles
    min_samples_split=5,
    min_samples_leaf=2,
    subsample=0.8,             # Stochastic GB
    random_state=42
)
```

Justificación:

- `learning_rate=0.1` : Balance entre convergencia y generalización
- `max_depth=5` : Árboles débiles (shallow) para boosting
- `subsample=0.8` : Reduce overfitting, acelera entrenamiento

5. Pipeline de Optimización ML

5.1 Fase de Entrenamiento

```
def entrenar(self, df_inicial):
    """
    Entrena ambos modelos usando el horario inicial
    """
    # 1. Extraer características y labels
    X = []
    y_salon = []
    y_calidad = []

    for idx, clase in df_inicial.iterrows():
        features = self.extraer_features(clase, df_inicial, idx)
        X.append(features)
        y_salon.append(clase['Salon'])

        # Calidad = inverso de energía local
        calidad = self.calcular_calidad_local(clase, df_inicial)
        y_calidad.append(calidad)

    X = np.array(X)
    y_salon = np.array(y_salon)
    y_calidad = np.array(y_calidad)
```

5.1 Fase de Entrenamiento (continuación)

```
    # 2. Entrenar clasificador (predicción de salón)
    self.clasificador.fit(X, y_salon)

    # 3. Entrenar regresor (calidad de asignación)
    self.regressor_calidad.fit(X, y_calidad)

    # 4. Calcular métricas de entrenamiento
    y_pred = self.clasificador.predict(X)
    accuracy = (y_pred == y_salon).mean()

    return {
        'accuracy': accuracy,
        'n_samples': len(X),
        'n_features': X.shape[1]
    }
```

5.2 Cálculo de Calidad Local

La calidad de una asignación se define como:

$$calidad(c_i, s_j) = -energia\_local(c_i, s_j)$$

Donde:

$$\begin{aligned} energia\_local(c_i, s_j) = w_1 \cdot distancia\_anterior(c_i, s_j) \\ + w_2 \cdot \mathbb{1}\left[tipo(c_i) \neq tipo(s_j)\right] \\ + w_3 \cdot \mathbb{1}\left[s_j \in S_{invalidos}\right] \\ + w_4 \cdot \mathbb{1}\left[cambio\_piso(c_i, s_j)\right] \end{aligned}$$

Objetivo: Enseñar al modelo qué hace una asignación "buena" vs "mala"

5.3 Fase de Optimización

```
def optimizar(self, df_inicial):
    """
    Genera nueva asignación usando modelos entrenados
    """
    df_resultado = df_inicial.copy()
    cambios = 0

    for idx in range(len(df_resultado)):
        clase = df_resultado.iloc[idx]

        # Proteger clases inmutables (P1)
        if idx in self.indices_inmutables:
            continue

        # 1. Extraer características
        features = self.extraer_features(clase, df_resultado, idx)

        # 2. Obtener salones candidatos
        candidatos = self.obtener_salones_validos(clase)

        # 3. Predecir mejor salón
        salon_predicho = self.clasificador.predict([features])[0]

        # 4. Si predicción es válida, usar directamente
        if salon_predicho in candidatos:
            mejor_salon = salon_predicho
```

5.3 Fase de Optimización (continuación)

```
        else:
            # 5. Evaluar todos los candidatos con regresor
            mejor_salon = None
            mejor_calidad = -infinito

            for salon in candidatos:
                # Simular asignación
                features_temp = self.modificar_features(features, salon)
                calidad = self.regressor_calidad.predict([features_temp])[0]

                if calidad > mejor_calidad:
                    mejor_calidad = calidad
                    mejor_salon = salon

        # 6. Aplicar asignación
        if df_resultado.loc[idx, 'Salon'] != mejor_salon:
            df_resultado.loc[idx, 'Salon'] = mejor_salon
            cambios += 1

    return df_resultado, cambios
```

5.4 Validación de Asignaciones

Después de cada predicción, validamos:

```
def validar_asignacion(self, clase, salon, df_actual):
    """
    Verifica que la asignación sea factible
    """
    # 1. Tipo correcto
    if tipo(clase) != tipo(salon):
        return False

    # 2. No es inválido
    if salon in self.salones_invalidos:
        return False
```

5.4 Validación de Asignaciones (continuación)

```
    # 3. No hay conflicto temporal
    conflicto = df_actual[
        (df_actual['Dia'] == clase['Dia']) &
        (df_actual['Bloque_Horario'] == clase['Bloque_Horario']) &
        (df_actual['Salon'] == salon)
    ]
    if len(conflicto) > 0:
        return False

    return True
```

6. Análisis de Complejidad

6.1 Complejidad Temporal

Entrenamiento:

$$T_{train} = O(K \cdot n \cdot d \cdot \log n \cdot h)$$

Donde:

- K = número de árboles
- n = número de muestras
- d = número de características
- h = profundidad máxima

Para nuestro caso:

- $K = 100$
- $n = 680$
- $d \approx 20$
- $h = 20$

$$T_{train} \approx 100 \cdot 680 \cdot 20 \cdot \log(680) \cdot 20 \approx 3.7 \times 10^7 \text{ operaciones}$$

Optimización (Inferencia):

$$T_{opt} = O(n \cdot K \cdot h \cdot d)$$

$$T_{opt} \approx 680 \cdot 100 \cdot 20 \cdot 20 \approx 2.7 \times 10^7 \text{ operaciones}$$

Tiempo Real: ~15-20 segundos en hardware moderno

6.2 Complejidad Espacial

$$S = O(K \cdot n_{nodes} + n \cdot d)$$

Donde n_{nodes} es el número promedio de nodos por árbol.

Para árboles balanceados de profundidad h:

$$n_{nodes} \approx 2^h - 1$$

$$S \approx 100 \cdot (2^{20} - 1) + 680 \cdot 20 \approx 10^8 \text{ bytes} \approx 100 \text{ MB}$$

7. Ventajas y Limitaciones

7.1 Ventajas

✅ **Velocidad:** Inferencia muy rápida una vez entrenado
✅ **Aprendizaje:** Mejora con más datos históricos
✅ **Robustez:** Ensemble reduce varianza
✅ **Interpretabilidad:** Feature importance muestra qué importa
✅ **No paramétrico:** No asume distribución de datos

7.2 Limitaciones

❌ **Requiere datos:** Necesita horarios previos de calidad
❌ **Overfitting:** Puede memorizar patrones específicos
❌ **Exploración limitada:** No explora fuera de lo aprendido
❌ **Dependencia de features:** Calidad depende de feature engineering
❌ **Cold start:** Mal rendimiento sin datos de entrenamiento

8. Mejoras y Extensiones

8.1 Transfer Learning

Usar modelos pre-entrenados en otros campus/instituciones:

$$\theta_{nuevo} = \theta_{pretrained} + \Delta\theta_{fine-tune}$$

8.2 Active Learning

Seleccionar ejemplos más informativos para etiquetar:

$$x^* = \arg\max_{x \in U} uncertainty(x)$$

Donde $uncertainty$ puede ser entropía de predicción.

8.3 Deep Learning

Reemplazar Random Forest con redes neuronales:

```
Input (features) → Dense(128, ReLU) → Dropout(0.3)
                 → Dense(64, ReLU) → Dropout(0.3)
                 → Dense(|S|, Softmax) → Output (salon)
```

8.4 Reinforcement Learning

Formular como MDP (Markov Decision Process):

- **Estado:** Asignación parcial actual
- **Acción:** Asignar clase c_i a salón s_j
- **Recompensa:** $-energia(asignacion)$
- **Política:** $\pi(s, a) = P(a|s)$

Usar Q-Learning o Policy Gradient para aprender política óptima.

9. Resultados Experimentales

9.1 Métricas de Entrenamiento

```
Modelo: Random Forest Classifier
├── Accuracy: 0.85
├── Precision: 0.83
├── Recall: 0.82
└── F1-Score: 0.82

Modelo: Gradient Boosting Regressor
├── R²: 0.78
├── MSE: 45.2
└── MAE: 5.3
```

9.2 Comparación con Otros Métodos

Métrica	ML	Greedy	Genético
Tiempo	**15.8s**	29.3s	73.9s
P1	100%	100%	100%
Distancia	**1821**	1951	2413
Consistencia	Media	Alta	Baja

9.3 Feature Importance

```
Top 10 características más importantes:
1. profesor_encoded (0.18)
2. salon_clase_anterior (0.15)
3. tipo_clase (0.12)
4. tiene_preferencia (0.10)
5. bloque_horario (0.09)
6. materia_encoded (0.08)
7. dia_semana (0.07)
8. num_estudiantes (0.06)
9. semestre (0.05)
10. piso_clase_anterior (0.04)
```

Interpretación: El profesor y el contexto de la clase anterior son los factores más predictivos.

10. Conclusiones

El enfoque de Machine Learning ofrece una alternativa **rápida y efectiva** para la asignación de salones, especialmente cuando:

- Existen datos históricos de calidad
- Se requiere velocidad de ejecución
- Los patrones son relativamente estables

Sin embargo, requiere **cuidadoso feature engineering** y **datos de entrenamiento representativos** para alcanzar su máximo potencial.

Referencias

1. Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32.
2. Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189-1232.
3. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning* (2nd ed.). Springer.
4. Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *KDD*, 785-794.
5. Ke, G., et al. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *NIPS*, 3146-3154.

Algoritmo Genético

Búsqueda Evolutiva Global

1. Introducción y Fundamento Teóricológico

1.1 Inspiración Evolutiva

Los **Algoritmos Genéticos (AG)** son metaheurísticas inspiradas en la evolución biológica de Charles Darwin. Simulan el proceso de selección natural donde los individuos más aptos tienen mayor probabilidad de sobrevivir y reproducirse.

Principios Fundamentales:

1. **Variación:** Diversidad genética mediante mutación y recombinación
2. **Herencia:** Transmisión de características a descendientes
3. **Selección:** Supervivencia del más apto
4. **Adaptación:** Mejora gradual de la población

Teorema de Holland (Schema Theorem):

Los esquemas (patrones de solución) cortos, de bajo orden y con alta aptitud reciben un número exponencialmente creciente de muestras en generaciones sucesivas.

1.2 Analogía Biológica vs Computacional

Biología	Algoritmo Genético
Individuo	Solución candidata
Cromosoma	Representación de solución
Gen	Variable de decisión
Alelo	Valor de variable
Población	Conjunto de soluciones

1.2 Analogía Biológica vs Computacional (continuación)

Biología	Algoritmo Genético
Generación	Iteración del algoritmo
Fitness	Calidad de solución
Selección natural	Selección por aptitud
Cruce sexual	Operador de cruce
Mutación	Operador de mutación

1.3 Ventajas Teóricas

✅ **Exploración global:** Mantiene población diversa
✅ **Paralelismo implícito:** Evalúa múltiples soluciones simultáneamente
✅ **Robustez:** No requiere gradientes ni derivadas
✅ **Flexibilidad:** Aplicable a problemas discretos y continuos
✅ **Escape de óptimos locales:** Mediante mutación y diversidad

2. Representación Cromosómica

2.1 Codificación de Soluciones

Para el problema de asignación de salones, usamos **codificación directa**:

Cromosoma:

$$\mathbf{x} = [x_1, x_2, \ldots, x_n]$$

Donde:

- $n = 680$ (número de clases)
- $x_i \in S$ (salón asignado a clase i)

Ejemplo:

```
Clase:  [c1,  c2,  c3,  c4,  c5,  ...]
Salón:  [FF1, LBD, FF2, FF1, LIA, ...]
          ↑     ↑     ↑     ↑     ↑
         Gen   Gen   Gen   Gen   Gen
```

2.2 Espacio de Búsqueda

El espacio de búsqueda Ω es:

$$\Omega = S^n = \{(s_1, s_2, \ldots, s_n) : s_i \in S\}$$

$$|\Omega| = |S|^n = 21^{680} \approx 10^{900}$$

Subespacio Factible:

$$\Omega_{factible} = \{\mathbf{x} \in \Omega : satisface\ restricciones\ duras\}$$

$$|\Omega_{factible}| \ll |\Omega|$$

2.3 Función de Aptitud (Fitness)

La aptitud mide la calidad de una solución:

$$fitness(\mathbf{x}) = -E(\mathbf{x})$$

Donde $E(\mathbf{x})$ es la función de energía definida anteriormente.

Normalización:

Para mantener valores positivos y facilitar selección proporcional:

$$fitness_{norm}(\mathbf{x}) = \frac{1}{1 + E(\mathbf{x})}$$

O usando ranking:

$$fitness_{rank}(\mathbf{x}) = rank(\mathbf{x})$$

Donde $rank(\mathbf{x}) \in \{1, 2, \ldots, |P|\}$ es la posición en la población ordenada.

3. Algoritmo Principal

3.1 Pseudocódigo General

```
Algoritmo Genético(problema, parámetros):
    # Inicialización
    P₀ = generar_poblacion_inicial(tam_poblacion)
    evaluar_fitness(P₀)
    t = 0

    # Evolución
    mientras t < max_generaciones y no convergió:
        # Selección
        padres = seleccionar_padres(Pₜ)

        # Cruce
        hijos = aplicar_cruce(padres, prob_cruce)

        # Mutación
        hijos = aplicar_mutacion(hijos, prob_mutacion)

        # Reparación (si necesario)
        hijos = reparar_soluciones(hijos)

        # Evaluación
        evaluar_fitness(hijos)

        # Reemplazo
        Pₜ₊₁ = seleccionar_supervivientes(Pₜ, hijos)

        t = t + 1

    retornar mejor_individuo(Pₜ)
```

3.2 Parámetros del Algoritmo

```
class OptimizadorGenetico:
    def __init__(self):
        # Parámetros poblacionales
        self.tam_poblacion = 150
        self.num_generaciones = 500

        # Probabilidades de operadores
        self.prob_cruce = 0.8
        self.prob_mutacion_inicial = 0.1
        self.prob_mutacion_final = 0.3

        # Selección
        self.tam_torneo = 5
        self.elitismo = 0.1  # 10% mejores pasan directamente

        # Diversidad
        self.penalizacion_diversidad = True
        self.min_diversidad = 0.3
```

Justificación de Valores:

- **tam_poblacion = 150:** Balance entre diversidad y costo computacional
- **num_generaciones = 500:** Suficiente para convergencia en problemas complejos
- **prob_cruce = 0.8:** Valor estándar en literatura (Holland, 1975)
- **prob_mutacion adaptativa:** Aumenta con generaciones para escapar estancamiento

4. Operadores Genéticos

4.1 Inicialización de Población

Estrategia Híbrida:

```python
def generar_poblacion_inicial(self, df):
    poblacion = []

    # 1. Incluir solución pre-asignada (élite)
    sol_inicial = self.cargar_solucion_inicial(df)
    poblacion.append(sol_inicial)

    # 2. Generar variaciones de la inicial (30%)
    for _ in range(int(0.3 * self.tam_poblacion)):
        sol_variacion = self.perturbar_solucion(sol_inicial)
        poblacion.append(sol_variacion)

    # 3. Generar soluciones aleatorias (70%)
    for _ in range(self.tam_poblacion - len(poblacion)):
        sol_aleatoria = self.generar_solucion_aleatoria(df)
        poblacion.append(sol_aleatoria)

    return poblacion
```

4.2 Selección de Padres

Implementamos **Selección por Torneo**:

Algoritmo:

```
def seleccion_torneo(self, poblacion, fitness, k=5):
    """
    Selecciona un individuo mediante torneo de tamaño k
    """
    # Seleccionar k individuos aleatorios
    competidores = random.sample(range(len(poblacion)), k)

    # Retornar el de mayor fitness
    mejor_idx = max(competidores, key=lambda i: fitness[i])

    return poblacion[mejor_idx]
```

Presión Selectiva:

La probabilidad de que el mejor individuo sea seleccionado en un torneo de tamaño k:

$$P(seleccionar\ mejor) = 1 - \left(\frac{|P| - 1}{|P|}\right)^k$$

Para $|P| = 150$, $k = 5$:

$$P \approx 1 - (0.993)^5 \approx 0.034 = 3.4\%$$

Ventajas del Torneo:

- No requiere ordenamiento completo
- Fácil paralelización
- Presión selectiva ajustable (mediante k)

4.3 Operador de Cruce

Implementamos **Cruce de Un Punto con Protección**:

Algoritmo:

```
def cruce_un_punto(self, padre1, padre2):
    """
    Cruce de un punto respetando genes inmutables
    """
    n = len(padre1)

    # Seleccionar punto de cruce aleatorio
    punto = random.randint(1, n-1)

    # Crear hijos
    hijo1 = padre1.copy()
    hijo2 = padre2.copy()

    # Intercambiar segmentos (excepto inmutables)
    for i in range(punto, n):
        if i not in self.indices_inmutables:
            hijo1[i] = padre2[i]
```

Representación Gráfica:

```
Padre 1: [FF1|FF2|FF3|FF4|FF5|FF6|FF7]
                     ↓ punto de cruce
Padre 2: [LBD|LIA|FF8|FF9|FFA|FFB|FFC]
         ──────────────────────────────
Hijo 1:  [FF1|FF2|FF3|FF9|FFA|FFB|FFC]
Hijo 2:  [LBD|LIA|FF8|FF4|FF5|FF6|FF7]
```

Teorema del Building Block:

El cruce preserva y combina bloques de construcción (schemata) de alta aptitud.

Demostración (informal):

- Si padre1 tiene buen bloque en posiciones [1-3]
- Y padre2 tiene buen bloque en posiciones [5-7]
- Cruce en posición 4 puede combinar ambos bloques en hijo

4.4 Operador de Mutación

Mutación Adaptativa por Intercambio:

```
def mutacion_intercambio(self, individuo, generacion):
    """
    Muta mediante intercambio de genes compatibles
    Probabilidad aumenta con generaciones
    """
    # Probabilidad adaptativa
    progreso = generacion / self.num_generaciones
    prob_mut = self.prob_mutacion_inicial + \
               (self.prob_mutacion_final - self.prob_mutacion_inicial) * progreso

    individuo_mutado = individuo.copy()
```

4.4 Operador de Mutación (continuación)

```
    for i in range(len(individuo)):
        if random.random() < prob_mut:
            # No mutar inmutables
            if i in self.indices_inmutables:
                continue

            # Seleccionar gen compatible para intercambiar
            j = self.seleccionar_gen_compatible(i, individuo)

            if j is not None and j not in self.indices_inmutables:
                # Intercambiar
                individuo_mutado[i], individuo_mutado[j] = \
                    individuo_mutado[j], individuo_mutado[i]

    return individuo_mutado
```

Probabilidad Adaptativa:

$$p_{mut}(t) = p_{min} + (p_{max} - p_{min}) \cdot \frac{t}{T}$$

Donde:

- $p_{min} = 0.1$ (inicial)
- $p_{max} = 0.3$ (final)
- t = generación actual
- T = total de generaciones

Justificación:

- **Inicio:** Baja mutación para explotar buenos esquemas
- **Final:** Alta mutación para escapar estancamiento

4.5 Reparación de Soluciones

Después de cruce y mutación, pueden generarse soluciones inválidas:

```
def reparar_solucion(self, individuo, df):
    """
    Repara violaciones de restricciones duras
    """
    individuo_reparado = individuo.copy()

    for i in range(len(individuo)):
        clase = df.iloc[i]
        salon_actual = individuo[i]

        # Verificar validez
        if not self.es_asignacion_valida(clase, salon_actual, individuo, i):
```

4.5 Reparación de Soluciones (continuación)

```
            # Buscar salón válido
            salones_validos = self.obtener_salones_validos(clase)

            for salon in salones_validos:
                if self.es_asignacion_valida(clase, salon, individuo, i):
                    individuo_reparado[i] = salon
                    break

    return individuo_reparado
```

Teorema de Factibilidad:

Toda solución puede repararse a una solución factible si existe al menos una asignación válida para cada clase.

5. Estrategias de Reemplazo

5.1 Reemplazo Generacional con Elitismo

```python
def seleccionar_supervivientes(self, poblacion, hijos, fitness_pob, fitness_hijos):
    """
    Combina población actual e hijos, selecciona mejores
    """
    # Calcular número de élites
    num_elites = int(self.elitismo * len(poblacion))

    # Identificar élites
    indices_ordenados = sorted(range(len(poblacion)),
                               key=lambda i: fitness_pob[i],
                               reverse=True)
    elites = [poblacion[i] for i in indices_ordenados[:num_elites]]

    # Combinar hijos con no-élites
    nueva_poblacion = elites + hijos[:len(poblacion) - num_elites]

    return nueva_poblacion
```

Tasa de Elitismo:

$$\tau_e = \frac{|elites|}{|P|} = 0.1$$

Teorema de Convergencia con Elitismo:

Un AG con elitismo converge al óptimo global con probabilidad 1 cuando $t \to \infty$.

Demostración (sketch):

1. Elitismo preserva mejor solución encontrada
2. Mutación garantiza $P(visitar\ cualquier\ solucion) > 0$
3. Con tiempo infinito, se visitará el óptimo
4. Elitismo lo preservará una vez encontrado

6. Control de Diversidad

6.1 Medida de Diversidad

Definimos diversidad de población como:

$$D(P) = \frac{1}{|P|(|P|-1)} \sum_{i=1}^{|P|} \sum_{j=i+1}^{|P|} distancia(\mathbf{x}_i, \mathbf{x}_j)$$

Donde:

$$distancia(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}\left[x_{i,k} \neq x_{j,k}\right]$$

Interpretación: Proporción promedio de genes diferentes entre individuos.

6.2 Sharing (Compartición de Fitness)

Para mantener diversidad, penalizamos individuos similares:

$$fitness_{shared}(\mathbf{x}_i) = \frac{fitness(\mathbf{x}_i)}{\sum_{j=1}^{|P|} sh(d(\mathbf{x}_i, \mathbf{x}_j))}$$

Donde la función de sharing es:

$$sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{share}}\right)^{\alpha} & \text{si } d < \sigma_{share} \\ 0 & \text{en otro caso} \end{cases}$$

Parámetros típicos:

- $\sigma_{share} = 0.1$ (radio de nicho)
- $\alpha = 2$ (forma de la función)

6.3 Reinicio Adaptativo

Si diversidad cae por debajo de umbral:

```
def verificar_y_reiniciar(self, poblacion, generacion):
    """
    Reinicia población si diversidad es muy baja
    """
    diversidad = self.calcular_diversidad(poblacion)

    if diversidad < self.min_diversidad:
        print(f"   ⚠️  Diversidad baja ({diversidad:.3f}), reiniciando...")

        # Preservar mejores
        num_preservar = int(0.2 * len(poblacion))
        mejores = self.obtener_mejores(poblacion, num_preservar)

        # Generar nuevos
        nuevos = self.generar_poblacion_inicial(len(poblacion) - num_preservar)

        return mejores + nuevos

    return poblacion
```

7. Análisis de Complejidad

7.1 Complejidad Temporal

Por generación:

$$T_{gen} = T_{eval} + T_{sel} + T_{cruce} + T_{mut} + T_{reemplazo}$$

Donde:

- $T_{eval} = O(|P| \cdot n)$ (evaluar población)
- $T_{sel} = O(|P| \cdot k)$ (selección por torneo)
- $T_{cruce} = O(|P| \cdot n)$ (aplicar cruce)
- $T_{mut} = O(|P| \cdot n)$ (aplicar mutación)
- $T_{reemplazo} = O(|P| \log |P|)$ (ordenar para elitismo)

$$T_{gen} = O(|P| \cdot n)$$

Total:

$$T_{total} = O(G \cdot |P| \cdot n)$$

7.2 Complejidad Espacial

$$S = O(|P| \cdot n + |P|)$$

Donde:

- $|P| \cdot n$ = almacenar población
- $|P|$ = almacenar fitness

$$S \approx 150 \cdot 680 + 150 \approx 102,150 \text{ valores}$$

Memoria: ~800 KB (despreciable)

8. Criterios de Parada

El algoritmo se detiene cuando se cumple alguna de estas condiciones:

8.1 Número Máximo de Generaciones

$$t \geq G_{max}$$

8.2 Convergencia de Fitness

$$\frac{fitness_{mejor} - fitness_{promedio}}{fitness_{mejor}} < \epsilon$$

Donde $\epsilon = 0.01$ (1% de diferencia)

8.3 Estancamiento

No hay mejora en k generaciones consecutivas:

$$\forall i \in [t-k, t] : fitness_{mejor}(i) = fitness_{mejor}(t)$$

Típicamente $k = 50$

9. Resultados y Análisis

9.1 Evolución de Fitness

Generación	Mejor	Promedio	Peor	Diversidad
0	85,234	92,456	98,123	0.85
50	83,112	86,234	91,456	0.72
100	82,456	84,123	88,234	0.65
150	82,234	83,456	86,123	0.58
200	82,156	83,234	85,456	0.52
250	82,089	82,987	84,234	0.45
300	82,034	82,756	83,456	0.38
350	81,998	82,645	83,123	0.31
400	81,976	82,578	82,987	0.28
450	81,967	82,534	82,876	0.25
500	81,962	82,512	82,789	0.23

Observaciones:

- Mejora rápida en primeras 100 generaciones
- Convergencia gradual después
- Diversidad decrece naturalmente
- Diferencia mejor-promedio se reduce (convergencia)

9.2 Comparación con Otros Algoritmos

Métrica	Genético	Greedy+HC	ML
Tiempo	73.9s	29.3s	15.8s
Mejor solución	81,962	82,716	82,234
Consistencia	Baja	Alta	Media
Exploración	Excelente	Limitada	Limitada
Explotación	Buena	Excelente	Buena

9.3 Análisis de Operadores

Impacto del Cruce:

Prob. Cruce	Mejor Fitness	Generaciones
0.5	82,456	520
0.7	82,123	480
0.8	81,962	500
0.9	82,234	510

Óptimo: 0.8 (valor estándar)

Impacto de la Mutación:

Prob. Mutación	Mejor Fitness	Diversidad Final
0.05	82,567	0.15
0.10	82,234	0.23
0.15	81,962	0.31
0.20	82,123	0.42

Óptimo: 0.10-0.15 (adaptativa)

10. Ventajas y Limitaciones

10.1 Ventajas

✅ **Exploración global:** Mantiene población diversa
✅ **Robustez:** No requiere información de gradiente
✅ **Paralelizable:** Evaluaciones independientes
✅ **Flexibilidad:** Fácil incorporar nuevas restricciones
✅ **Escape de óptimos locales:** Mediante mutación
✅ **Soluciones múltiples:** Población final contiene varias buenas soluciones

10.2 Limitaciones

❌ **Lento:** Requiere muchas evaluaciones
❌ **Parámetros:** Sensible a configuración
❌ **Convergencia prematura:** Puede perder diversidad
❌ **No determinista:** Resultados varían entre ejecuciones
❌ **Escalabilidad:** Costo crece con tamaño de población

10.3 Cuándo Usar Algoritmo Genético

Recomendado cuando:

- Se requiere exploración exhaustiva
- Hay tiempo suficiente (>1 minuto)
- Problema tiene muchos óptimos locales
- Se necesitan múltiples soluciones alternativas

No recomendado cuando:

- Se requiere velocidad (<30s)
- Problema es convexo o unimodal
- Hay buenos algoritmos específicos disponibles

11. Extensiones y Mejoras

11.1 Algoritmos Genéticos Paralelos

Modelo de Islas:

```
Población dividida en subpoblaciones (islas)
Cada isla evoluciona independientemente
Migración periódica de mejores individuos
```

Speedup teórico:

$$S = \frac{T_{secuencial}}{T_{paralelo}} \approx \frac{G \cdot |P|}{G \cdot |P|/k + overhead}$$

Para k procesadores.

11.2 Algoritmos Meméticos

Combinar AG con búsqueda local:

```
Para cada individuo en población:
    Aplicar Hill Climbing local
```

Ventaja: Combina exploración global (AG) con explotación local (HC)

11.3 Coevolución

Evolucionar simultáneamente:

- Población de soluciones
- Población de restricciones/pesos

Objetivo: Encontrar configuración de parámetros óptima automáticamente

12. Conclusiones

El Algoritmo Genético ofrece:

- **Mejor exploración** del espacio de búsqueda
- **Múltiples soluciones** de calidad
- **Robustez** ante cambios en el problema

A costa de:

- **Mayor tiempo** de ejecución
- **Variabilidad** en resultados
- **Complejidad** de configuración

Es la opción recomendada cuando se dispone de tiempo y se requiere la mejor solución posible.

Referencias

1. Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.
2. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
3. Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press.
4. Eiben, A. E., & Smith, J. E. (2015). *Introduction to Evolutionary Computing* (2nd ed.). Springer.
5. Deb, K. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley.
6. Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer.

Post-procesamiento y Corrección

Validación y Refinamiento de Soluciones

1. Introducción y Necesidad

1.1 Motivación

Aunque los optimizadores están diseñados para respetar los índices inmutables, pueden ocurrir violaciones accidentales debido a:

1. **Errores de implementación:** Bugs en el código de protección
2. **Condiciones de carrera:** En implementaciones paralelas
3. **Operadores complejos:** Mutaciones o cruces que no verifican correctamente
4. **Datos corruptos:** Archivos de entrada modificados manualmente

Solución: Sistema de corrección post-optimización que **garantiza** 100% P1.

1.2 Principio de Defensa en Profundidad

```
Capa 1: Pre-asignación forzada
    ↓
Capa 2: Protección durante optimización
    ↓
Capa 3: Corrección post-optimización ← Este documento
    ↓
Garantía: 100% PRIORIDAD 1
```

Teorema de Corrección:

Si existe al menos una asignación válida para cada clase P1, el sistema de corrección garantiza 100% de cumplimiento.

2. Arquitectura del Sistema

2.1 Flujo de Corrección

```
graph TD
    A[Horario Optimizado] --> B[Cargar Preferencias]
    B --> C[Identificar Violaciones]
    C --> D{Hay violaciones?}
    D -->|No| E[Retornar sin cambios]
    D -->|Sí| F[Ordenar por Prioridad]
    F --> G[Corregir una por una]
    G --> H[Resolver Conflictos]
    H --> I[Verificar 100%]
    I --> J{Cumple 100%?}
    J -->|Sí| K[Guardar Corregido]
    J -->|No| L[Reportar Error]
```

2.2 Componentes Principales

```python
class CorrectorPrioridades:
    def __init__(self):
        self.preferencias = {}
        self.correcciones_aplicadas = 0
        self.conflictos_resueltos = 0

    def corregir(self, archivo_horario):
        # 1. Cargar datos
        df = pd.read_csv(archivo_horario)
        self.cargar_preferencias()

        # 2. Identificar violaciones
        violaciones = self.identificar_violaciones(df)

        if not violaciones:
            print("✅ No hay violaciones")
            return df

        # 3. Corregir
        df_corregido = self.aplicar_correcciones(df, violaciones)

        # 4. Verificar
        cumplimiento = self.verificar_cumplimiento(df_corregido)

        if cumplimiento < 100.0:
            raise Exception(f"❌ Cumplimiento {cumplimiento}% < 100%")

        # 5. Guardar
        df_corregido.to_csv(archivo_horario, index=False)

        return df_corregido
```

3. Identificación de Violaciones

3.1 Algoritmo de Detección

```
def identificar_violaciones(self, df):
    """
    Identifica todas las clases P1 que no están en su salón preferido
    """
    violaciones = []

    for idx, clase in df.iterrows():
        profesor = clase['Profesor']
        materia = clase['Materia']
        tipo = clase['Tipo_Salon']
        salon_actual = clase['Salon']
```

3.1 Algoritmo de Detección (continuación 1)

```
        # Verificar si tiene preferencia prioritaria
        if profesor not in self.preferencias:
            continue

        if materia not in self.preferencias[profesor]['materias']:
            continue

        pref = self.preferencias[profesor]['materias'][materia]

        # Verificar teoría
        if tipo == 'Teoría':
            if (pref.get('prioridad_teoria') == 'Prioritario' and
                pref.get('salon_teoria') != 'Sin preferencia'):

                salon_esperado = pref['salon_teoria']

                if salon_actual != salon_esperado:
                    violaciones.append({
                        'idx': idx,
                        'clase': clase,
                        'salon_actual': salon_actual,
                        'salon_esperado': salon_esperado,
                        'profesor': profesor,
                        'materia': materia,
                        'tipo': 'Teoría'
                    })
```

3.1 Algoritmo de Detección (continuación 2)

```
        # Verificar laboratorio
        elif tipo == 'Laboratorio':
            if (pref.get('prioridad_lab') == 'Prioritario' and
                pref.get('salon_lab') != 'Sin preferencia'):

                salon_esperado = pref['salon_lab']

                if salon_actual != salon_esperado:
                    violaciones.append({
                        'idx': idx,
                        'clase': clase,
                        'salon_actual': salon_actual,
                        'salon_esperado': salon_esperado,
                        'profesor': profesor,
                        'materia': materia,
                        'tipo': 'Laboratorio'
                    })

    return violaciones
```

3.2 Clasificación de Violaciones

Tipo 1: Violación Simple

- Clase P1 en salón incorrecto
- Salón preferido está libre
- **Solución:** Cambio directo

Tipo 2: Violación con Conflicto

- Clase P1 en salón incorrecto
- Salón preferido ocupado por clase no-P1
- **Solución:** Desplazar ocupante

Tipo 3: Violación Compleja

- Clase P1 en salón incorrecto
- Salón preferido ocupado por otra clase P1
- **Solución:** Requiere intervención manual o re-optimización

4. Aplicación de Correcciones

4.1 Algoritmo Principal

```python
def aplicar_correcciones(self, df, violaciones):
    """
    Aplica correcciones para todas las violaciones
    """
    df_corregido = df.copy()
    ocupacion = self.construir_mapa_ocupacion(df_corregido)

    # Ordenar violaciones por prioridad
    violaciones_ordenadas = self.ordenar_violaciones(violaciones)

    for violacion in violaciones_ordenadas:
        idx = violacion['idx']
        salon_esperado = violacion['salon_esperado']
        clase = violacion['clase']

        # Construir clave de ocupación
        key = (clase['Dia'], clase['Bloque_Horario'], salon_esperado)

        # Verificar si salón está libre
        if key not in ocupacion:
            # Corrección simple
            df_corregido.loc[idx, 'Salon'] = salon_esperado
            ocupacion[key] = idx
            self.correcciones_aplicadas += 1
        else:
            # Resolver conflicto
            exito = self.resolver_conflicto_correccion(
                df_corregido, idx, salon_esperado,
                ocupacion, violacion
            )

            if exito:
                self.correcciones_aplicadas += 1
                self.conflictos_resueltos += 1
            else:
                print(f"❌ No se pudo corregir índice {idx}")

    return df_corregido
```

4.2 Ordenamiento de Violaciones

```python
def ordenar_violaciones(self, violaciones):
    """
    Ordena violaciones por prioridad de corrección
    """
    def prioridad_correccion(v):
        # Criterios (mayor valor = mayor prioridad):
        # 1. Número de clases del profesor
        num_clases = sum(1 for vv in violaciones
                         if vv['profesor'] == v['profesor'])

        # 2. Tipo (Teoría > Laboratorio)
        tipo_peso = 2 if v['tipo'] == 'Teoría' else 1

        # 3. Complejidad de resolución
        complejidad = self.estimar_complejidad(v)

        return (num_clases, tipo_peso, -complejidad)

    return sorted(violaciones, key=prioridad_correccion, reverse=True)
```

5. Resolución de Conflictos

5.1 Estrategia de Desplazamiento

Cuando el salón preferido está ocupado:

```
def resolver_conflicto_correccion(self, df, idx_p1, salon_p1, ocupacion, violacion):
    """
    Resuelve conflicto desplazando clase ocupante
    """
    clase_p1 = violacion['clase']
    key = (clase_p1['Dia'], clase_p1['Bloque_Horario'], salon_p1)

    idx_ocupante = ocupacion[key]
    clase_ocupante = df.iloc[idx_ocupante]

    # Verificar si ocupante también es P1
    if self.es_prioritaria(clase_ocupante):
        # Conflicto entre dos P1: no se puede resolver automáticamente
        print(f"⚠️  Conflicto entre dos P1: {idx_p1} y {idx_ocupante}")
        return False
```

5.1 Estrategia de Desplazamiento (continuación 1)

```
    # Buscar salón alternativo para ocupante
    salones_alternativos = self.obtener_salones_validos(clase_ocupante)

    for salon_alt in salones_alternativos:
        key_alt = (clase_ocupante['Dia'],
                   clase_ocupante['Bloque_Horario'],
                   salon_alt)

        if key_alt not in ocupacion:
            # Desplazar ocupante
            df.loc[idx_ocupante, 'Salon'] = salon_alt
            ocupacion[key_alt] = idx_ocupante
```

5.1 Estrategia de Desplazamiento (continuación 2)

```
            # Asignar P1 a su salón preferido
            df.loc[idx_p1, 'Salon'] = salon_p1
            ocupacion[key] = idx_p1

            return True

    # No se encontró salón alternativo
    return False
```

5.2 Desplazamiento en Cadena

Si el salón alternativo también está ocupado:

```
def desplazar_en_cadena(self, df, idx_inicial, salon_objetivo, ocupacion, profundidad=0):
    """
    Desplaza clases en cadena hasta liberar salón objetivo
    """
    MAX_PROFUNDIDAD = 10

    if profundidad > MAX_PROFUNDIDAD:
        return False

    clase = df.iloc[idx_inicial]
    key = (clase['Dia'], clase['Bloque_Horario'], salon_objetivo)

    # Si salón está libre, asignar directamente
    if key not in ocupacion:
        df.loc[idx_inicial, 'Salon'] = salon_objetivo
        ocupacion[key] = idx_inicial
        return True

    # Salón ocupado: desplazar ocupante primero
    idx_ocupante = ocupacion[key]
    clase_ocupante = df.iloc[idx_ocupante]

    # No desplazar clases P1
    if self.es_prioritaria(clase_ocupante):
        return False

    # Buscar salón para ocupante
    salones_alt = self.obtener_salones_validos(clase_ocupante)

    for salon_alt in salones_alt:
        # Intentar desplazar ocupante recursivamente
        if self.desplazar_en_cadena(df, idx_ocupante, salon_alt,
                                    ocupacion, profundidad + 1):
            # Ocupante desplazado exitosamente
            df.loc[idx_inicial, 'Salon'] = salon_objetivo
            ocupacion[key] = idx_inicial
            return True
```

Teorema de Desplazamiento en Cadena:
Si existe una cadena de desplazamientos de longitud finita que libera el salón objetivo, el algoritmo la encontrará.

Demostración:
Por inducción en la profundidad:

- **Caso base (d=0):** Salón libre, asignación directa
- **Paso inductivo:** Si existe cadena de longitud $d + 1$, el algoritmo explora recursivamente hasta encontrarla
- **Terminación:** Profundidad máxima evita ciclos infinitos

6. Verificación de Cumplimiento

6.1 Cálculo de Cumplimiento

```python
def verificar_cumplimiento(self, df):
    """
    Calcula porcentaje de cumplimiento de PRIORIDAD 1
    """
    total_p1 = 0
    cumplidas = 0

    for idx, clase in df.iterrows():
        profesor = clase['Profesor']
        materia = clase['Materia']
        tipo = clase['Tipo_Salon']
        salon_actual = clase['Salon']

        # Verificar si es P1
        if not self.es_prioritaria_con_salon(clase):
            continue

        total_p1 += 1
        salon_esperado = self.obtener_salon_preferido(clase)

        if salon_actual == salon_esperado:
            cumplidas += 1

    if total_p1 == 0:
        return 100.0

    return (cumplidas / total_p1) * 100.0
```

6.2 Reporte Detallado

```
def generar_reporte(self, df, violaciones_iniciales):
    """
    Genera reporte detallado de correcciones
    """
    cumplimiento_final = self.verificar_cumplimiento(df)

    reporte = f"""
================================================================================
📊 RESUMEN DE CORRECCIÓN
================================================================================
Total clases prioritarias: {len(violaciones_iniciales) + self.correcciones_aplicadas}
Violaciones encontradas: {len(violaciones_iniciales)}
Correcciones aplicadas: {self.correcciones_aplicadas}
Conflictos resueltos: {self.conflictos_resueltos}
Cumplimiento final: {cumplimiento_final:.1f}%
================================================================================
"""

    if cumplimiento_final == 100.0:
        reporte += "\n🎉 ¡Prioridades corregidas exitosamente!\n"
    else:
        reporte += f"\n⚠️  Cumplimiento {cumplimiento_final}% < 100%\n"

    return reporte
```

7. Casos Especiales

7.1 Conflictos Irresolvables

Cuando dos clases P1 quieren el mismo salón al mismo tiempo:

```
def manejar_conflicto_irresolvable(self, idx1, idx2, salon, df):
    """
    Maneja conflicto entre dos clases P1
    """
    clase1 = df.iloc[idx1]
    clase2 = df.iloc[idx2]

    mensaje = f"""
❌ CONFLICTO IRRESOLVABLE
──────────────────────────
Clase 1: {clase1['Materia']} ({clase1['Grupo']}) - {clase1['Profesor']}
Clase 2: {clase2['Materia']} ({clase2['Grupo']}) - {clase2['Profesor']}
Horario: {clase1['Dia']} {clase1['Bloque_Horario']}
Salón: {salon}

ACCIÓN REQUERIDA:
1. Contactar a profesores involucrados
2. Negociar cambio de horario o salón alternativo
3. Actualizar preferencias en configuración
4. Re-ejecutar optimización
"""
```

7.1 Conflictos Irresolvables (continuación)

```
    print(mensaje)

    # Guardar en log
    with open('conflictos_irresolvables.log', 'a') as f:
        f.write(mensaje + '\n')
```

7.2 Salones Insuficientes

Si no hay salones alternativos para desplazar:

```
def manejar_salones_insuficientes(self, clase, df):
    """
    Maneja caso de salones insuficientes
    """
    mensaje = f"""
⚠️  SALONES INSUFICIENTES
═══════════════════════════════

Clase: {clase['Materia']} ({clase['Grupo']})
Tipo: {clase['Tipo_Salon']}
Horario: {clase['Dia']} {clase['Bloque_Horario']}
```

7.2 Salones Insuficientes (continuación)

```
SUGERENCIAS:
1. Verificar disponibilidad de salones del tipo requerido
2. Considerar usar salones de otro tipo (si aplicable)
3. Ajustar horarios para distribuir mejor la carga
4. Aumentar número de salones disponibles
"""

    print(mensaje)
```

8. Métricas y Estadísticas

8.1 Métricas de Corrección

```
Ejecución típica:
├── Violaciones encontradas: 3-44
├── Correcciones simples: 60-70%
├── Conflictos resueltos: 30-40%
├── Tiempo: 0.1-0.3s
└── Cumplimiento final: 100%
```

8.2 Distribución de Violaciones

```
Por optimizador:
├── Greedy: 3 violaciones (0.4%)
├── ML: 44 violaciones (5.0%)
└── Genético: 69 violaciones (7.8%)

Por tipo:
├── Teoría: 65%
└── Laboratorio: 35%

Por causa:
├── Operadores no protegidos: 70%
├── Inicialización aleatoria: 20%
└── Bugs de implementación: 10%
```

9. Complejidad Computacional

9.1 Análisis Temporal

Identificación: $O(n)$

Corrección simple: $O(v)$ donde v = violaciones

Resolución de conflictos: $O(v \cdot m \cdot d)$ donde:

- m = salones alternativos promedio
- d = profundidad máxima de desplazamiento

Total: $O(n + v \cdot m \cdot d)$

Para $n = 680, v \approx 50, m \approx 5, d = 10$:

$$T \approx 680 + 50 \cdot 5 \cdot 10 = 3,180 \text{ operaciones}$$

Tiempo real: ~0.2 segundos

9.2 Análisis Espacial

$$S = O(n + v)$$

Memoria: ~10 MB

10. Integración con Pipeline

10.1 Uso en ejecutar_todos.py

```
# En ejecutar_todos.py
subprocess.run(["python3", "optimizador_greedy.py"])
subprocess.run(["python3", "corregir_prioridades.py",
                "datos_estructurados/04_Horario_Optimizado_Greedy.csv"])

subprocess.run(["python3", "optimizador_ml.py"])
subprocess.run(["python3", "corregir_prioridades.py",
                "datos_estructurados/05_Horario_Optimizado_ML.csv"])

subprocess.run(["python3", "optimizador_genetico.py"])
subprocess.run(["python3", "corregir_prioridades.py",
                "datos_estructurados/06_Horario_Optimizado_Genetico.csv"])
```

10.2 Verificación Automática

```python
def verificar_pipeline():
    """
    Verifica que todos los horarios cumplan 100% P1
    """
    archivos = [
        'datos_estructurados/04_Horario_Optimizado_Greedy.csv',
        'datos_estructurados/05_Horario_Optimizado_ML.csv',
        'datos_estructurados/06_Horario_Optimizado_Genetico.csv'
    ]

    corrector = CorrectorPrioridades()

    for archivo in archivos:
        df = pd.read_csv(archivo)
        cumplimiento = corrector.verificar_cumplimiento(df)

        assert cumplimiento == 100.0, \
            f"❌ {archivo}: {cumplimiento}% != 100%"

    print("✅ Todos los horarios cumplen 100% P1")
```

11. Ventajas del Sistema

✅ **Garantía absoluta** de 100% P1
✅ **Corrección automática** sin intervención manual
✅ **Rápido** (<1 segundo)
✅ **Robusto** ante cualquier violación
✅ **Transparente** con reportes detallados
✅ **Integrado** en pipeline automático

12. Limitaciones y Consideraciones

❌ **No previene violaciones:** Solo las corrige después
❌ **Puede empeorar P2/P3:** Al desplazar clases
❌ **Conflictos irresolvables:** Requieren intervención manual
❌ **Dependencia de salones:** Necesita suficientes salones alternativos

13. Mejoras Futuras

13.1 Corrección Inteligente

Minimizar impacto en P2/P3 al desplazar:

```
def desplazar_minimizando_impacto(self, clase, salones_alt, df):
    """
    Selecciona salón alternativo que minimiza impacto en P2/P3
    """
    mejor_salon = None
    menor_impacto = float('inf')

    for salon in salones_alt:
        impacto = self.calcular_impacto_p2_p3(clase, salon, df)

        if impacto < menor_impacto:
            menor_impacto = impacto
            mejor_salon = salon

    return mejor_salon
```

13.2 Prevención Proactiva

Verificar durante optimización:

```
# En optimizadores
def verificar_antes_de_aplicar(self, nueva_solucion):
    """
    Verifica P1 antes de aceptar nueva solución
    """
    violaciones = self.identificar_violaciones_p1(nueva_solucion)

    if violaciones:
        # Rechazar solución
        return False

    return True
```

14. Conclusiones

El sistema de corrección post-optimización:

1. **Garantiza** 100% cumplimiento de PRIORIDAD 1
2. **Complementa** la protección durante optimización
3. **Proporciona** última línea de defensa
4. **Permite** que optimizadores se enfoquen en calidad
5. **Asegura** robustez del sistema completo

Es un componente **esencial** que hace el sistema **production-ready**.

Referencias

1. Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.
2. Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.
3. Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Arquitectura del Código

Diseño e Implementación del Sistema

1. Visión General del Sistema

1.1 Arquitectura de Alto Nivel

```
Sistema-Salones-ISC/
│
├── Capa de Presentación (UI)
│   └── configurador_materias.py
│
├── Capa de Procesamiento
│   ├── pre_asignar_p1.py
│   ├── optimizador_greedy.py
│   ├── optimizador_ml.py
│   ├── optimizador_genetico.py
│   └── corregir_prioridades.py
│
├── Capa de Utilidades
│   └── utils_restricciones.py
│
├── Capa de Orquestación
│   ├── ejecutar_todos.py
│   └── generar_comparativa_completa.py
│
└── Capa de Datos
    ├── datos_estructurados/
    ├── preferencias_profesores.json
    ├── configuracion_materias.json
    └── asignacion_grupos_1er_semestre.json
```

1.2 Flujo de Datos

```
graph LR
    A[Configuración] --> B[Horario Inicial]
    B --> C[Pre-asignación P1]
    C --> D[Optimizadores]
    D --> E[Corrección]
    E --> F[Comparativas]
    F --> G[Reportes]
```

2. Módulos Principales

2.1 configurador_materias.py

Propósito: Interfaz gráfica para configuración del sistema

Clase Principal:

```
class ConfiguradorMaterias:
    """
    Aplicación Tkinter para configurar materias y preferencias
    """
    def __init__(self):
        self.root = tk.Tk()
        self.materias = {}
        self.preferencias = {}
        self.grupos_1er = {}

        self.crear_interfaz()
```

2.1 configurador_materias.py (continuación)

```
    # Métodos principales
    def crear_interfaz(self)
    def crear_tab_materias(self)
    def crear_tab_preferencias(self)
    def crear_tab_grupos(self)
    def guardar_todo(self)
    def cargar_todo(self)
```

Responsabilidades:

- Configurar características de materias (horas, distribución)
- Definir preferencias de profesores (salón + prioridad)
- Asignar grupos de primer semestre
- Guardar/cargar configuración en JSON

Dependencias:

2.2 pre_asignar_p1.py

Propósito: Pre-asignación forzada de PRIORIDAD 1

Funciones Principales:

```
def cargar_datos():
    """Carga horario inicial y preferencias"""

def identificar_clases_prioritarias(df, preferencias):
    """Identifica todas las clases P1"""

def asignar_forzadamente(df, clases_p1):
    """Asigna cada clase P1 a su salón preferido"""

def guardar_resultados(df, indices_inmutables):
    """Guarda horario pre-asignado e índices"""
```

Algoritmo:

1. Cargar horario inicial
2. Identificar clases P1 desde preferencias
3. Ordenar por complejidad
4. Asignar forzadamente (resolviendo conflictos)
5. Marcar índices como inmutables
6. Guardar resultados

Archivos de Entrada:

- `datos_estructurados/01_Horario_Inicial.csv`
- `preferencias_profesores.json`

Archivos de Salida:

- `datos_estructurados/00_Horario_PreAsignado_P1.csv`
- `datos_estructurados/indices_inmutables_p1.json`

2.3 optimizador_greedy.py

Propósito: Optimización mediante Greedy + Hill Climbing

Clase Principal:

```
class OptimizadorGreedyHC:
    def __init__(self):
        self.indices_inmutables = set()
        self.max_iter_hc = 100
        self.pesos = {...}

    # Fase 1: Construcción
    def construccion_greedy(self, df):
        """Construye solución inicial vorazmente"""

    def calcular_score(self, clase, salon, solucion):
        """Calcula score para asignación"""
```

2.3 optimizador_greedy.py (continuación)

```
    # Fase 2: Refinamiento
    def hill_climbing(self, solucion, df):
        """Mejora mediante búsqueda local"""

    def calcular_energia(self, solucion, df):
        """Calcula energía total de solución"""

    # Utilidades
    def analizar_movimientos(self, df):
        """Analiza movimientos de profesores"""
```

Flujo de Ejecución:

1. Cargar horario pre-asignado
2. Cargar índices inmutables
3. Construcción greedy (respetando inmutables)
4. Hill Climbing (protegiendo inmutables)
5. Corrección final de P1
6. Análisis de métricas
7. Guardar resultado

Archivos de Entrada:

- `datos_estructurados/00_Horario_PreAsignado_P1.csv`
- `datos_estructurados/indices_inmutables_p1.json`
- `preferencias_profesores.json`
- `configuracion_materias.json`

Archivos de Salida:

- `datos_estructurados/04_Horario_Optimizado_Greedy.csv`
- `comparativas/04_inicial_vs_greedy/metricas_movimientos.csv`

2.4 optimizador_ml.py

Propósito: Optimización mediante Machine Learning

Clase Principal:

```
class OptimizadorML:
    def __init__(self):
        self.clasificador = RandomForestClassifier(...)
        self.regressor_calidad = GradientBoostingRegressor(...)
        self.indices_inmutables = set()

    # Entrenamiento
    def entrenar(self, df_inicial):
        """Entrena modelos con horario inicial"""

    def extraer_features(self, clase, df, idx):
        """Extrae vector de características"""
```

2.4 optimizador_ml.py (continuación)

```
    # Optimización
    def optimizar(self, df_inicial):
        """Genera nueva asignación usando modelos"""

    def predecir_salon(self, features):
        """Predice mejor salón para clase"""
```

Flujo de Ejecución:

1. Cargar horario pre-asignado
2. Cargar índices inmutables
3. Entrenar Random Forest (predicción de salón)
4. Entrenar Gradient Boosting (calidad de asignación)
5. Optimizar (predecir salón para cada clase)
6. Proteger clases inmutables
7. Análisis de métricas
8. Guardar resultado

Archivos de Entrada:

- `datos_estructurados/00_Horario_PreAsignado_P1.csv`
- `datos_estructurados/indices_inmutables_p1.json`

Archivos de Salida:

- `datos_estructurados/05_Horario_Optimizado_ML.csv`
- `comparativas/02_inicial_vs_ml/metricas_movimientos.csv`

2.5 optimizador_genetico.py

Propósito: Optimización mediante Algoritmo Genético

Clase Principal:

```
class OptimizadorGenetico:
    def __init__(self):
        self.tam_poblacion = 150
        self.num_generaciones = 500
        self.prob_cruce = 0.8
        self.prob_mutacion = 0.1
        self.indices_inmutables = set()

    # Inicialización
    def generar_poblacion_inicial(self, df):
        """Genera población inicial"""
```

2.5 optimizador_genetico.py (continuación)

```
    # Operadores
    def seleccion_torneo(self, poblacion, fitness):
        """Selecciona padres por torneo"""

    def cruce_un_punto(self, padre1, padre2):
        """Cruza dos individuos"""

    def mutacion_intercambio(self, individuo):
        """Muta individuo"""

    # Evolución
    def evolucionar(self, df):
        """Ejecuta algoritmo genético"""
```

Flujo de Ejecución:

1. Cargar horario pre-asignado
2. Cargar índices inmutables
3. Generar población inicial

Flujo de Ejecución (continuación):

4. Evolucionar durante N generaciones:
    - Selección por torneo
    - Cruce (respetando inmutables)
    - Mutación (respetando inmutables)
    - Evaluación de fitness
    - Reemplazo con elitismo
5. Retornar mejor individuo
6. Análisis de métricas
7. Guardar resultado

Archivos de Entrada:

- `datos_estructurados/00_Horario_PreAsignado_P1.csv`
- `datos_estructurados/indices_inmutables_p1.json`

Archivos de Salida:

- `datos_estructurados/06_Horario_Optimizado_Genetico.csv`
- `comparativas/03_inicial_vs_genetico/metricas_movimientos.csv`

2.6 corregir_prioridades.py

Propósito: Corrección post-optimización de PRIORIDAD 1

Funciones Principales:

```
def cargar_preferencias():
    """Carga preferencias de profesores"""

def identificar_violaciones(df, preferencias):
    """Identifica clases P1 en salón incorrecto"""

def corregir_violaciones(df, violaciones):
    """Corrige todas las violaciones"""

def verificar_cumplimiento(df, preferencias):
    """Verifica 100% cumplimiento"""
```

Flujo de Ejecución:

1. Cargar horario optimizado
2. Cargar preferencias
3. Identificar violaciones de P1
4. Si hay violaciones:
    - Ordenar por prioridad
    - Corregir una por una
    - Resolver conflictos
5. Verificar 100% cumplimiento
6. Guardar horario corregido

Uso:

```
python3 corregir_prioridades.py datos_estructurados/04_Horario_Optimizado_Greedy.csv
```

2.7 utils_restricciones.py

Propósito: Funciones de utilidad para validación y restricciones

Funciones Principales:

```
def validar_asignacion(clase, salon, df):
    """Verifica si asignación es válida"""

def obtener_salones_validos(clase, salones_teoria, laboratorios):
    """Retorna salones válidos para clase"""

def calcular_distancia(salon1, salon2):
    """Calcula distancia entre salones"""

def verificar_conflicto_temporal(clase1, clase2, asignacion):
    """Verifica si hay conflicto temporal"""

def pre_asignar_prioritarias(df, config, preferencias, ...):
    """Pre-asigna clases prioritarias (legacy)"""
```

Uso:

- Importado por todos los optimizadores
- Proporciona funciones comunes de validación
- Mantiene lógica de restricciones centralizada

2.8 ejecutar_todos.py

Propósito: Script maestro de orquestación

Flujo:

```
def main():
    # 1. Pre-asignación
    ejecutar("pre_asignar_p1.py")

    # 2. Greedy
    ejecutar("optimizador_greedy.py")
    ejecutar("corregir_prioridades.py", "04_Horario_Optimizado_Greedy.csv")

    # 3. ML
    ejecutar("optimizador_ml.py")
    ejecutar("corregir_prioridades.py", "05_Horario_Optimizado_ML.csv")
```

2.8 ejecutar_todos.py (continuación)

```
    # 4. Genético
    ejecutar("optimizador_genetico.py")
    ejecutar("corregir_prioridades.py", "06_Horario_Optimizado_Genetico.csv")

    # 5. Comparativas
    ejecutar("generar_comparativa_completa.py")

    # 6. Resumen
    mostrar_resumen()
```

Características:

- Ejecución secuencial de todo el pipeline
- Medición de tiempos
- Logging de salidas
- Manejo de errores

2.9 generar_comparativa_completa.py

Propósito: Generación de reportes y gráficos

Funciones Principales:

```
def generar_excel_formato(csv_file, output_file):
    """Genera Excel con formato bonito"""

def generar_excel_comparativo(dfs):
    """Genera Excel con todos los optimizadores"""

def generar_graficos(dfs):
    """Genera gráficos de análisis"""

def verificar_prioridad_1(dfs, preferencias):
    """Verifica cumplimiento P1"""
```

Salidas Generadas:

- Excels formateados por optimizador
- Excel comparativo (todos juntos)
- Gráficos de tiempos, cumplimiento, métricas
- Excel consolidado con resumen

3. Estructuras de Datos

3.1 Horario (DataFrame)

```
# Estructura de datos principal
df = pd.DataFrame({
    'Dia': str,                # Lunes, Martes, ...
    'Bloque_Horario': int,     # 0700, 0800, ...
    'Materia': str,            # Nombre de materia
    'Grupo': str,              # 1527A, 2514/B, ...
    'Profesor': str,           # PROFESOR 3, ...
    'Salon': str,              # FF1, LBD, ...
    'Es_Invalido': int,        # 0 o 1
    'Tipo_Salon': str,         # Teoría, Laboratorio
    'Piso': int                # 0 o 1
})
```

3.2 Preferencias (JSON)


```
{
  "PROFESOR 3": {
    "materias": {
      "LENGUAJES Y AUTÓMATAS I": {
        "salon_teoria": "FFA",
        "prioridad_teoria": "Prioritario",
        "salon_lab": "Sin preferencia",
        "prioridad_lab": "Normal"
      }
    }
  }
}
```

3.3 Índices Inmutables (JSON)

```
{
  "indices": [12, 45, 67, ...],
  "total": 88,
  "timestamp": "2025-12-21T11:00:00",
  "version": "1.0"
}
```

3.4 Solución (Dict)

```
# Representación interna de solución
solucion = {
    0: 'FF1',    # Clase 0 → Salón FF1
    1: 'LBD',    # Clase 1 → Salón LBD
    2: 'FF2',    # Clase 2 → Salón FF2
    ...
}
```

4. Patrones de Diseño

4.1 Strategy Pattern

Diferentes optimizadores implementan la misma interfaz:

```
class OptimizadorBase:
    def optimizar(self, df_inicial):
        raise NotImplementedError

class OptimizadorGreedy(OptimizadorBase):
    def optimizar(self, df_inicial):
        # Implementación Greedy

class OptimizadorML(OptimizadorBase):
    def optimizar(self, df_inicial):
        # Implementación ML
```

4.2 Template Method

Estructura común de optimización:

```
def optimizar(self, df):
    # 1. Cargar datos
    self.cargar_configuracion()

    # 2. Inicializar
    solucion = self.inicializar()

    # 3. Optimizar (método específico)
    solucion = self.algoritmo_especifico(solucion)

    # 4. Post-procesar
    solucion = self.post_procesar(solucion)

    # 5. Guardar
    self.guardar_resultado(solucion)
```

4.3 Facade Pattern

`ejecutar_todos.py` proporciona interfaz simple:

```
# En lugar de ejecutar 7 scripts manualmente
python3 ejecutar_todos.py  # Un solo comando
```

5. Convenciones de Código

5.1 Nomenclatura

Archivos:

- `snake_case.py` para scripts
- `PascalCase` para clases
- Números prefijos para orden (01_, 02_, ...)

Variables:

- `snake_case` para variables y funciones
- `UPPER_CASE` para constantes
- `_private` para métodos privados

Clases:

- `PascalCase` para nombres de clase
- Nombres descriptivos (OptimizadorGreedy, no OG)

5.2 Documentación

Docstrings:

```
def funcion(param1, param2):
    """
    Descripción breve de la función

    Args:
        param1: Descripción del parámetro 1
        param2: Descripción del parámetro 2

    Returns:
        Descripción del valor de retorno
    """
```

Comentarios:

```
# Comentarios explicativos para lógica compleja
# No comentar lo obvio
```

5.3 Manejo de Errores

```
try:
    # Operación que puede fallar
    resultado = operacion_riesgosa()
except FileNotFoundError:
    print("❌ Archivo no encontrado")
    sys.exit(1)
except Exception as e:
    print(f"❌ Error: {e}")
    raise
```

6. Dependencias

6.1 Dependencias Externas

```
pandas>=1.5.0           # Manipulación de datos
openpyxl>=3.0.0         # Excel I/O
matplotlib>=3.5.0       # Gráficos
seaborn>=0.12.0         # Visualización
scikit-learn>=1.0.0     # Machine Learning
```

6.2 Dependencias Estándar

```
import json              # Configuración
import random            # Aleatoriedad
import sys               # Sistema
import subprocess        # Ejecución de scripts
from pathlib import Path  # Manejo de rutas
from datetime import datetime  # Timestamps
```

7. Testing y Validación

7.1 Validación de Datos

```python
def validar_horario(df):
    """Valida estructura de horario"""
    assert 'Dia' in df.columns
    assert 'Salon' in df.columns
    assert len(df) == 680
    # ...
```

7.2 Verificación de Invariantes

```
def verificar_invariantes(df, indices_inmutables, preferencias):
    """Verifica invariantes del sistema"""
    # Invariante 1: P1 al 100%
    assert verificar_p1(df, preferencias) == 100.0

    # Invariante 2: Sin conflictos temporales
    assert not tiene_conflictos(df)

    # Invariante 3: Salones válidos
    assert todos_salones_validos(df)
```

8. Optimizaciones de Rendimiento

8.1 Caching

```
@lru_cache(maxsize=1000)
def calcular_distancia(salon1, salon2):
    """Cachea cálculos de distancia"""
    # ...
```

8.2 Vectorización

```
# En lugar de loops
for i in range(len(df)):
    df.loc[i, 'Piso'] = obtener_piso(df.loc[i, 'Salon'])

# Usar operaciones vectorizadas
df['Piso'] = df['Salon'].apply(obtener_piso)
```

8.3 Paralelización

```
# En Random Forest
RandomForestClassifier(n_jobs=-1)  # Usar todos los cores
```

9. Extensibilidad

9.1 Agregar Nuevo Optimizador

1. Crear `optimizador_nuevo.py`
2. Implementar interfaz estándar:

```
class OptimizadorNuevo:
    def __init__(self):
        self.indices_inmutables = cargar_inmutables()

    def optimizar(self, df):
        # Implementación
```

3. Agregar a `ejecutar_todos.py`
4. Agregar a `generar_comparativa_completa.py`

9.2 Agregar Nueva Restricción

1. Definir en `utils_restricciones.py`:

```
def verificar_nueva_restriccion(clase, salon):
    # Lógica de verificación
```

2. Integrar en función objetivo de optimizadores
3. Actualizar documentación

10. Deployment y Producción

10.1 Instalación

```
git clone <repo>
cd Sistema-Salones-ISC
pip install -r requirements.txt
```

10.2 Ejecución

```
# Configurar
python3 configurador_materias.py

# Ejecutar pipeline completo
python3 ejecutar_todos.py

# O ejecutar componentes individuales
python3 pre_asignar_p1.py
python3 optimizador_greedy.py
# ...
```

10.3 Monitoreo

```
# Ver logs
tail -f ejecucion_final.log

# Verificar resultados
ls -lh datos_estructurados/
ls -lh comparativas/final/
```

11. Mantenimiento

11.1 Actualización de Preferencias

1. Abrir `configurador_materias.py`
2. Modificar preferencias
3. Guardar
4. Re-ejecutar pipeline

11.2 Debugging

```
# Activar modo debug
DEBUG = True

if DEBUG:
    print(f"Debug: {variable}")
    import pdb; pdb.set_trace()
```

11.3 Logs

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    filename='sistema.log'
)
```

12. Conclusiones

El sistema está diseñado con:

✅ **Modularidad:** Componentes independientes y reutilizables
✅ **Escalabilidad:** Fácil agregar nuevos optimizadores
✅ **Mantenibilidad:** Código limpio y bien documentado
✅ **Robustez:** Validación en múltiples capas
✅ **Extensibilidad:** Patrones de diseño facilitan extensiones

Es un sistema **production-ready** para uso real en el Instituto Tecnológico de Ciudad Madero.

Referencias

1. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
2. Gamma, E., et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
3. McConnell, S. (2004). *Code Complete* (2nd ed.). Microsoft Press.

Aplicación Web (BETA)

Interfaz Web para el Sistema

⚠️ Estado Actual: BETA

> ⚠️ **ESTADO**: En Desarrollo - Versión BETA
> 🚧 **NO LISTA PARA PRODUCCIÓN**

Aviso Importante

La aplicación web del Sistema de Asignación de Salones se encuentra actualmente en **fase BETA de desarrollo**. Aunque funcional para demostración, **NO está lista para uso en producción** y requiere desarrollo adicional antes de su implementación institucional.

Estado Actual

✅ Funcionalidades Implementadas

1. **Visualización de Horarios**
    - Vista de horarios por grupo
    - Vista de horarios por profesor
    - Vista de horarios por salón
    - Filtros básicos (día, semestre)
2. **Interfaz de Usuario**
    - Diseño responsive básico
    - Navegación entre vistas
    - Tabla de horarios
    - Exportación a PDF (básica)

Arquitectura Actual

```
web-app/ (BETA)
├── backend/
│   ├── app.py                # Flask/FastAPI server
│   ├── api/
│   │   ├── horarios.py       # Endpoints de horarios
│   │   └── consultas.py      # Endpoints de consultas
│   └── models/
│       └── horario.py        # Modelos de datos
│
├── frontend/
│   ├── index.html            # Página principal
│   ├── css/
│   │   └── styles.css        # Estilos
│   └── js/
│       ├── app.js            # Lógica principal
│       └── api.js            # Cliente API
│
└── README_WEB.md             # Este archivo
```

Instalación (Solo para Desarrollo)

```
# Instalar dependencias
pip install flask flask-cors pandas

# Ejecutar servidor de desarrollo
cd web-app/backend
python app.py

# Abrir en navegador
open http://localhost:5000
```

Uso Básico (Demo)

Ver Horario de Grupo

```
http://localhost:5000/horario/grupo/1527A
```

Ver Horario de Profesor

```
http://localhost:5000/horario/profesor/PROFESOR%203
```

Ver Ocupación de Salón

```
http://localhost:5000/horario/salon/FFA
```

Roadmap de Desarrollo

Fase 1: Seguridad (Crítico)

- [ ] Implementar autenticación (JWT)
- [ ] Sistema de roles (Admin, Profesor, Estudiante)
- [ ] Validación de entrada
- [ ] Sanitización de datos
- [ ] HTTPS obligatorio

Fase 2: Funcionalidad Core

- [ ] Integración con optimizadores
- [ ] Edición de horarios (con permisos)
- [ ] Comparación de horarios
- [ ] Exportación avanzada (Excel, iCal, PDF mejorado)

Tecnologías Propuestas

Backend

- **Framework**: FastAPI (recomendado) o Flask
- **Base de Datos**: PostgreSQL
- **ORM**: SQLAlchemy
- **Autenticación**: JWT + OAuth2
- **Cache**: Redis
- **API Docs**: Swagger/OpenAPI

Frontend

- **Framework**: React o Vue.js
- **UI Library**: Material-UI o Ant Design
- **State Management**: Redux o Vuex
- **Build Tool**: Vite
- **Testing**: Jest + React Testing Library

DevOps

- **Containerización**: Docker
- **Orquestación**: Docker Compose (dev) / Kubernetes (prod)
- **CI/CD**: GitHub Actions
- **Monitoring**: Prometheus + Grafana
- **Logging**: ELK Stack

Advertencias de Seguridad

🚨 NO USAR EN PRODUCCIÓN SIN:

1. **Autenticación robusta**
    - Sistema de login seguro
    - Gestión de sesiones
    - Protección contra fuerza bruta
2. **Autorización por roles**
    - Permisos granulares
    - Validación en backend
    - Auditoría de acciones

Advertencias de Seguridad (continuación)

3. **Validación de datos**
    - Sanitización de entrada
    - Validación de tipos
    - Protección contra XSS/CSRF
4. **Encriptación**
    - HTTPS obligatorio
    - Encriptación de datos sensibles
    - Hashing de contraseñas (bcrypt)

Advertencias de Seguridad (resumen técnico)

5. **Auditoría y Logging**
    - Registro de todas las acciones
    - Monitoreo de seguridad
    - Alertas automáticas

Contribuir al Desarrollo Web

Si deseas contribuir al desarrollo de la aplicación web:

1. **Revisar roadmap** y seleccionar tarea
2. **Crear branch** desde `develop`
3. **Implementar** con tests
4. **Documentar** cambios
5. **Pull request** para revisión

Estándares de Código

```
# Backend (Python)
- PEP 8 style guide
- Type hints obligatorios
- Docstrings para funciones públicas
- Tests unitarios (pytest)
- Coverage > 80%

# Frontend (JavaScript)
- ESLint + Prettier
- Componentes funcionales
- PropTypes o TypeScript
- Tests de componentes
- Accesibilidad (a11y)
```

Contacto

Autor: Jesús Olvera

- **GitHub:** @jjho05
- **Email:** jjho.reivaj05@gmail.com / hernandez.jesusjavier.20.0770@gmail.com
- **Repositorio:** https://github.com/jjho05/Sistema-Salones-ISC
- **Institucional:** sistemas@cdmadero.tecnm.mx

Licencia

Mismo que el proyecto principal - Uso académico TECNM.

Última actualización: 2025-12-21

Versión: 0.1.0-beta

Estado: 🚧 En Desarrollo - NO PRODUCCIÓN

Mantenedor: Equipo de Desarrollo Web ISC

Preguntas Frecuentes

FAQ - Sistema de Asignación de Salones

General

¿Qué es el Sistema de Asignación de Salones ISC?

Es un sistema de optimización que asigna automáticamente salones a clases del programa de Ingeniería en Sistemas Computacionales del Instituto Tecnológico de Ciudad Madero, minimizando movimientos de profesores y garantizando cumplimiento de preferencias prioritarias.

¿Por qué necesitamos este sistema?

La asignación manual de 680+ clases a 21 salones es:

- **Tiempo intensiva**: Horas o días de trabajo manual
- **Propensa a errores**: Conflictos temporales, preferencias olvidadas
- **Subóptima**: No considera todas las optimizaciones posibles
- **Difícil de actualizar**: Cambios requieren rehacer todo

¿Qué problemas resuelve?

1. **Garantiza 100% cumplimiento** de preferencias prioritarias de profesores
2. **Minimiza movimientos** de profesores entre salones
3. **Reduce cambios de piso** (menos fatiga)
4. **Optimiza distancias** recorridas
5. **Genera reportes** automáticos y comparativos

Instalación y Configuración

¿Qué necesito para instalar el sistema?

Requisitos:

- Python 3.8 o superior
- pip (gestor de paquetes)
- ~100 MB de espacio en disco
- Sistema operativo: Windows, macOS, o Linux

Instalación:

```
pip install pandas openpyxl matplotlib seaborn scikit-learn
```

¿Cómo configuro las preferencias de profesores?

1. Ejecutar `python3 configurador_materias.py`
2. Ir a pestaña "Preferencias de Profesores"
3. Seleccionar profesor y materia
4. Configurar:
    - Salón preferido (Teoría/Laboratorio)
    - Prioridad (Prioritario/Normal/Sin preferencia)
5. Guardar configuración

¿Dónde se guardan las configuraciones?

- `preferencias_profesores.json` : Preferencias de profesores
- `configuracion_materias.json` : Configuración de materias
- `asignacion_grupos_1er_semestre.json` : Grupos de primer semestre

Uso del Sistema

¿Cómo ejecuto el sistema completo?

Opción 1 (Recomendada):

```
python3 ejecutar_todos.py
```

Opción 2 (Manual):

```
python3 pre_asignar_p1.py
python3 optimizador_greedy.py
python3 corregir_prioridades.py datos_estructurados/04_Horario_Optimizado_Greedy.csv
# ... etc
```

¿Cuánto tiempo tarda?

Componente	Tiempo
Pre-asignación	~0.3s
Greedy	~30s
ML	~16s
Genético	~74s
Corrección	~0.2s cada uno
Comparativas	~5s
Total	**~2-3 minutos**

¿Qué optimizador debo usar?

Optimizador	Cuándo usar
Greedy	Balance ideal velocidad/calidad, uso general
ML	Máxima velocidad, buena calidad
Genético	Mejor exploración, tiempo no crítico
Todos	Comparar y elegir mejor resultado

Prioridades

¿Qué significa PRIORIDAD 1?

PRIORIDAD 1 son preferencias de salón de profesores marcadas como "Prioritario". El sistema **garantiza 100%** de cumplimiento.

Ejemplo:

- PROFESOR 3 quiere FFA para Lenguajes y Autómatas I (Teoría)
- Prioridad: Prioritario
- **Garantía**: Todas sus clases estarán en FFA

¿Qué pasa si dos profesores quieren el mismo salón al mismo tiempo?

Durante configuración:

- El sistema detecta el conflicto
- Solicita resolver manualmente
- Opciones: Cambiar horario o salón de uno

Durante optimización:

- Pre-asignación resuelve automáticamente
- Desplaza clases no-prioritarias
- Si ambas son P1: Requiere intervención manual

¿Qué son PRIORIDAD 2 y 3?

PRIORIDAD 2: Consistencia de grupos

- Objetivo: Mantener grupos en mismo salón
- **No garantizado**, se optimiza cuando es posible

PRIORIDAD 3: Grupos de primer semestre

- Objetivo: Asignar grupos 15xx a salones específicos
- **No garantizado**, mejor esfuerzo

Estado actual: P2 y P3 en desarrollo (v2.1.0)

Resultados

¿Dónde encuentro los resultados?

Horarios optimizados:

- `datos_estructurados/04_Horario_Optimizado_Greedy.csv`
- `datos_estructurados/05_Horario_Optimizado_ML.csv`
- `datos_estructurados/06_Horario_Optimizado_Genetico.csv`

Excels formateados:

- `comparativas/04_inicial_vs_greedy/Horario_Optimizado_Greedy.xlsx`
- `comparativas/02_inicial_vs_ml/Horario_Optimizado_ML.xlsx`
- `comparativas/03_inicial_vs_genetico/Horario_Optimizado_Genetico.xlsx`

Excel comparativo:

- `comparativas/final/Comparativa_Todos_Optimizadores.xlsx`

Gráficos:

- `comparativas/final/graficos/`

¿Cómo interpreto los resultados?

Métricas clave:

1. **Cumplimiento P1**: Debe ser 100%
2. **Movimientos**: Menor es mejor (ideal < 320)
3. **Cambios piso**: Menor es mejor (ideal < 210)
4. **Distancia**: Menor es mejor (ideal < 2000)

Comparación:

- Ver `Comparativa_Todos_Optimizadores.xlsx`
- Comparar grupo por grupo
- Elegir optimizador con mejores métricas globales

¿Qué hago si PRIORIDAD 1 no está al 100%?

Esto NO debería pasar, pero si ocurre:

1. Ejecutar corrección:

```
python3 corregir_prioridades.py datos_estructurados/XX_Horario.csv
```

2. Si persiste:
   - Verificar `preferencias_profesores.json`
   - Revisar conflictos irresolvables
   - Contactar soporte técnico

Problemas Comunes

Error: "No se encontró el archivo"

Causa: Archivo de entrada no existe

Solución:

```
# Verificar archivos
ls datos_estructurados/01_Horario_Inicial.csv

# Si no existe, generarlo o copiarlo
```

Error: "PRIORIDAD 1 no al 100%"

Causa: Conflictos en preferencias o salones insuficientes

Solución:

1. Ejecutar `corregir_prioridades.py`
2. Verificar preferencias
3. Revisar disponibilidad de salones

El optimizador es muy lento

Causa: Parámetros muy altos (especialmente Genético)

Solución:

Editar parámetros en el optimizador:

```
# optimizador_genetico.py
optimizador = OptimizadorGenetico(
    tam_poblacion=50,       # Reducir de 150
    num_generaciones=200,  # Reducir de 500
)
```

Los resultados varían entre ejecuciones

Causa: Aleatoriedad en algoritmos (especialmente Genético)

Solución:

Fijar semilla aleatoria:

```
import random
random.seed(42)
```

Personalización

¿Puedo cambiar los pesos de optimización?

Sí, editar en cada optimizador:

```
# optimizador_greedy.py
self.pesos = {
    'movimientos': 10,      # Aumentar para priorizar
    'cambios_piso': 5,
    'distancia': 3,
}
```

¿Puedo agregar nuevas restricciones?

Sí:

1. Definir en `utils_restricciones.py`:

```
def verificar_nueva_restriccion(clase, salon):
    # Lógica
```

2. Integrar en función objetivo

3. Actualizar documentación

¿Puedo usar el sistema para otro campus?

Sí, pero requiere:

1. Actualizar lista de salones
2. Configurar nuevas materias
3. Definir preferencias de profesores
4. Ajustar parámetros si es necesario

Desarrollo y Contribución

¿Cómo contribuyo al proyecto?

1. Fork el repositorio
2. Crear branch para tu feature
3. Implementar con tests
4. Documentar cambios
5. Pull request para revisión

¿Dónde reporto bugs?

- GitHub Issues (preferido)
- Email a equipo de desarrollo
- Slack interno (si aplica)

¿Hay tests automatizados?

Estado actual: Tests básicos de validación

Roadmap: Suite completa de tests (v2.1.0)

Aplicación Web

¿Hay una aplicación web?

Estado: BETA - En desarrollo

Funcionalidades actuales:

- Visualización básica de horarios
- Filtros simples
- Exportación a PDF básica

Limitaciones:

- Sin autenticación
- No optimizado
- No lista para producción

Soporte

¿Dónde encuentro más documentación?

- `README.md` : Descripción general
- `docs/GUIA_USO.md` : Guía completa de usuario
- `docs/00_CONTEXTO_PROBLEMA.md` : Contexto matemático
- `docs/02-04_ALGORITMO_*.md` : Detalles de algoritmos
- `docs/07_ARQUITECTURA_CODIGO.md` : Arquitectura

¿A quién contacto para soporte?

- **Autor:** Jesús Olvera
- **GitHub:** @jjho05
- **Email:** jjho.reivaj05@gmail.com / hernandez.jesusjavier.20.0770@gmail.com
- **Issues:** https://github.com/jjho05/Sistema-Salones-ISC/issues
- **Institucional:** sistemas@cdmadero.tecnm.mx

¿El sistema tiene licencia?

Licencia: Uso académico - Tecnológico Nacional de México

Restricciones:

- Uso educativo e institucional
- No comercial sin autorización
- Atribución requerida

Rendimiento

¿Cuántas clases puede manejar?

Probado con: 680 clases, 21 salones

Escalabilidad:

- Hasta ~1000 clases: Sin problemas
- 1000-2000 clases: Posible, tiempos mayores
- > 2000 clases: Requiere optimización adicional

¿Funciona en tiempo real?

No, es un sistema de optimización offline.

Tiempo típico: 2-3 minutos para generar horarios completos

Uso recomendado: Ejecutar al inicio de semestre o cuando hay cambios

Futuro

¿Qué viene en próximas versiones?

v2.1.0 (Próximo):

- PRIORIDAD 2 y 3 implementadas
- App web mejorada
- API REST
- Más formatos de exportación

v3.0.0 (Futuro):

- Soporte multi-campus
- Optimización de horarios (no solo salones)
- Dashboard interactivo
- Integración institucional

Ver `CHANGELOG.md` para roadmap completo.

¿Puedo sugerir nuevas funcionalidades?

¡Sí!

- GitHub Issues con etiqueta "enhancement"
- Email a equipo de desarrollo
- Discusión en reuniones de coordinación

¿No encuentras tu pregunta?

Consulta la documentación completa en `docs/` o contacta al equipo de desarrollo.

Última actualización: 2025-12-21

Versión: 2.0.0

¡Gracias!

Preguntas y Discusión

Contacto

Jesús Olvera

- **GitHub:** @jjho05
- **Email:** jjho.reivaj05@gmail.com
- **Institución:** Instituto Tecnológico de Ciudad Madero
- **Programa:** Ingeniería en Sistemas Computacionales

Repositorio:

https://github.com/jjho05/Sistema-Salones-ISC

"Por mi patria y por mi bien"

Orgullo Tec Madero

🎓