Informe de Solución: Planificación de Horarios

Jackson C. Vera Pineda - estudiante de 4^{to} año, Ciencia de la Computación, facultad de Matemática y Computación, Universidad de La Habana

Massiel Paz Otaño - estudiante de 4^{to} año, Ciencia de la Computación, facultad de Matemática y Computación, Universidad de La Habana

8 de febrero de 2025

1. Introducción

Este informe aborda el problema de la planificación de horarios, un problema clásico de optimización combinatoria con múltiples restricciones. El objetivo es diseñar un horario que minimice conflictos, maximice el uso eficiente de recursos y satisfaga las preferencias de los estudiantes y profesores. A continuación, se presenta la formalización del problema, los subproblemas clave, estrategias de solución, análisis de complejidad y demostraciones de correctitud.

2. Demostración de NP-Completitud

a) Pertenece a NP
Una solución propuesta (un horario con asignaciones de aulas, profesores y tiempos) puede verificarse en tiempo polinomial:

- Comprobar que no hay superposiciones de clases en el mismo aula.
- Verificar que los profesores no están asignados a dos clases simultáneamente.
- Asegurar que se respetan las preferencias de estudiantes y disponibilidades de recursos.

Estas verificaciones requieren tiempo proporcional al número de actividades y restricciones, por lo que **está en NP**.

b) Es NP-Hard: reducción desde un problema NP-Completo
Para demostrar que es NP-Hard, reducimos un problema ya conocido como NP-Completo al problema escolar. Usaremos Graph Coloring (coloración de grafos), que es NP-Completo.

- **Graph Coloring**: Dado un grafo $G = (V, E)$ y k colores, ¿pueden asignarse colores a los nodos de modo que nodos adyacentes no compartan el mismo color?
- **Reducción al problema escolar**:
  - **Nodos del grafo**: Representan clases o exámenes.
  - **Aristas del grafo**: Representan conflictos (ejemplo: dos clases que no pueden ocurrir al mismo tiempo).
  - **Colores**: Representan bloques de tiempo (horas/días) en el horario.

- **Condiciones adicionales**:
  - Aulas como çolores secundarios"(cada color de tiempo debe asignarse a un aula disponible).
  - Profesores como restricciones de color (un profesor no puede estar en dos clases simultáneas).

Si podemos resolver el problema escolar, automáticamente resolvemos Graph Coloring. Como Graph Coloring es NP-Completo, el problema escolar es NP-Hard.

Por tanto, dado que el problema de planificación de horarios (escolar) está en NP y es NP-Hard, podemos concluir que es NP-Completo

El problema se reduce al problema de coloración de grafos (Graph Coloring), que es NP-Completo.

Reducción

- Construir un grafo G donde cada nodo representa una clase c.
- Crear una arista entre dos nodos si comparten: mismo profesor, estudiantes en común (evitar superposición), y/o requisitos de aula especializada.
- Asignar colores (pares (r, t)) a los nodos

Por lo tanto, Si podemos resolver la planificación de horarios en tiempo polinomial, entonces podemos resolver Graph Coloring en tiempo polinomial, lo cual es imposible a menos que $P = NP$.

3. Formalización del Problema

El problema se modela como un problema de optimización combinatoria con restricciones duras y blandas. A continuación, se definen los conjuntos, variables, restricciones y función objetivo.

3.1. Conjuntos y Variables

- $C = \{c_1, c_2, ..., c_n\}$: Conjunto de cursos/clases.
- $P = \{p_1, p_2, ..., p_m\}$: Conjunto de profesores.
- $S = \{s_1, s_2, ..., s_k\}$: Conjunto de estudiantes.
- $R = \{r_1, r_2, ..., r_l\}$: Conjunto de aulas/recursos.
- $T = \{t_1, t_2, ..., t_q\}$: Conjunto de franjas horarias.
- $x_{c,r,t} \in \{0, 1\}$: Variable binaria que indica si la clase c se asigna al aula r en el tiempo t.
- $y_{p,c} \in \{0, 1\}$: Variable binaria que indica si el profesor p está asignado a la clase c.
- $z_{s,c_1,c_2,t} \in \{0, 1\}$ (solo para versión lineal): Variable auxiliar que indica conflicto para el estudiante s entre c_1 y c_2 en tiempo t.

3.2. Parámetros

- $E_s \subseteq C$: Clases inscritas por el estudiante $s \in S$.
- Capacidad$(r) \in \mathbb{N}$: Capacidad máxima del aula r.
- Estudiantes$(c) \in \mathbb{N}$: Número de estudiantes en la clase c.
- Preferencia$_{s,c,t} \in \{0, 1\}$: Preferencia del estudiante s por la clase c en tiempo t.
- $\alpha, \beta, \gamma \geq 0$: Pesos para conflictos, subutilización y preferencias respectivamente.

3.3. Restricciones Duras

- **No superposición**:
$$\sum_{c \in C} x_{c,r,t} \leq 1 \quad \forall r \in R, t \in T$$
- **Disponibilidad de profesores**:
$$\sum_{c \in C} y_{p,c} \cdot x_{c,r,t} \leq 1 \quad \forall p \in P, t \in T$$
- **Aforo**:
$$x_{c,r,t} = 1 \implies \text{Capacidad}(r) \geq \text{Estudiantes}(c) \quad \forall c \in C, r \in R, t \in T$$

3.4. Restricciones Blandas y Función Objetivo

3.4.1. Versión Cuadrática (Original)

$$\begin{aligned}\text{Minimizar } \alpha \cdot & \underbrace{\sum_{s \in S} \sum_{\substack{c_1, c_2 \in E_s \\ c_1 < c_2}} \sum_{t \in T} \left(\sum_{r_1} x_{c_1,r_1,t} \right) \left(\sum_{r_2} x_{c_2,r_2,t} \right)}_{\text{Conflictos}} \\ & + \beta \cdot \underbrace{\sum_{c,r,t} x_{c,r,t} \cdot (\text{Capacidad}(r) - \text{Estudiantes}(c))}_{\text{Subutilización}} \\ & + \gamma \cdot \underbrace{\sum_{s,c \in E_s} \sum_{r,t} x_{c,r,t} \cdot (1 - \text{Preferencia}_{s,c,t})}_{\text{Preferencias insatisfechas}}\end{aligned}$$

3.4.2. Versión Linealizada

- **Restricciones adicionales**:
$$\begin{cases} z_{s,c_1,c_2,t} \geq \sum_{r_1} x_{c_1,r_1,t} + \sum_{r_2} x_{c_2,r_2,t} - 1 & \forall s, c_1 < c_2 \in E_s, t \\ z_{s,c_1,c_2,t} \leq \sum_{r} x_{c_1,r,t} & \forall s, c_1 < c_2 \in E_s, t \\ z_{s,c_1,c_2,t} \leq \sum_{r} x_{c_2,r,t} & \forall s, c_1 < c_2 \in E_s, t \end{cases}$$
- **Función Objetivo Lineal**:
$$\begin{aligned}\text{Minimizar } \alpha \cdot & \underbrace{\sum_{s, c_1 < c_2 \in E_s} \sum_{t} z_{s,c_1,c_2,t}}_{\text{Conflictos (lineal)}} \\ & + \beta \cdot \text{Subutilización} + \gamma \cdot \text{Preferencias insatisfechas}\end{aligned}$$

4. Subproblemas Clave y Estrategias de Solución

4.1. Subproblema 1: Asignación de Profesores sin Conflictos

- **Estrategia**: Modelar como un problema de emparejamiento en grafos bipartitos.
- **Formalización**: El problema se modela como un grafo bipartito ponderado $G = (T \cup C, E)$ donde:
  - T : Conjunto de profesores (nodos en una partición).
  - C : Conjunto de clases (nodos en la otra partición).
  - E : Aristas que conectan profesores t_i con clases c_j, con pesos w_{ij} que representa el costo de asignar c_j a t_i. Si un profesor no puede asignar una clase (por superposición de horarios o falta de disponibilidad), $w_{ij} = \infty$.

  Objetivo: Encontrar un **emparejamiento perfecto** (asignar cada clase a un profesor y viceversa) que minimice el costo total, garantizando que ningún profesor tenga clases superpuestas y viceversa.
- **Algoritmo**: Algoritmo húngaro.
  Pasos:
  1. **Reducción de filas y columnas**: Restar el mínimo de cada fila y luego de cada columna.
  2. **Cubrir ceros**: Usar el mínimo de líneas para cubrir todos los ceros en matriz reducida.
  3. **Ajustes de costos**:
     - Si el número de líneas es igual al de la matriz, se ha encontrado una asignación óptima.
     - De lo contrario, encontrar el elemento mínimo no cubierto, restárselo a las filas no cubiertas, y sumarlo a las columnas cubiertas. Volver al paso 2.
- **Pseudocódigo**:

```
Entrada: Matriz de costo n x n (cost).
Salida: Asignaciones óptimas (match).

1. Inicializar etiquetas para filas (u) y columnas (v)
2. Para cada fila i:
    - u[i] = min(cost[i][j]) para toda j.
3. Para cada columna j:
    - v[j] = min(cost[i][j] - u[i]) para toda i.
4. Usar BFS/DFS para encontrar emparejamientos perfectos.
5. Repetir hasta encontrar el emparejamiento.
```

- **Correctitud**:
  Este algoritmo garantiza una asignación óptima debido a:
  - **Teorema de Köning**: Establece que en grafos bipartitos, el tamaño del máximo apareamiento es igual al mínimo número de nodos necesarios para cubrir todas las aristas.

- **Optimalidad**: Cada iteración reduce el costo total potencial hasta alcanzar un apareamiento perfecto con costo mínimo.

- **Complejidad**:
  La complejidad es $O(n^3)$, donde n es el número de nodos (clases o profesores, el que sea mayor). Esto se debe a las operaciones de reducción y ajuste iterativo en la matriz de costos.

- **Implementación en Python**:
  La biblioteca *scipy* ofrece una implementación eficiente del algoritmo.

```
import numpy as np
from scipy.optimize import linear_sum_assignment
import time

# Matriz de costos (filas: profesores, columnas: clases)
# cost[i][j] = costo de asignar la clase j al profesor i
cost = np.array([
    [3, 1, 2],
    [2, 4, 3],
    [3, 2, 1]
])

# Medir el tiempo de inicio
start_time = time.time()

# Aplicar el Algoritmo Húngaro
row_ind, col_ind = linear_sum_assignment(cost)

# Medir el tiempo de fin
end_time = time.time()

# Calcular el tiempo de ejecución
execution_time = end_time - start_time

# Asignaciones óptimas
for teacher, class_idx in zip(row_ind, col_ind):
    print(f"Profesor {teacher} -> Clase {class_idx} (Costo: {cost[teacher][clas

# Costo total
print("Costo total:", cost[row_ind, col_ind].sum())

# Imprimir el tiempo de ejecución
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")
```

Salida:

```
Profesor 0 -> Clase 1 (Costo: 1)
Profesor 1 -> Clase 0 (Costo: 2)
```

```
Profesor 2 -> Clase 2 (Costo: 1)
Costo total: 4
Tiempo de ejecución: 0.000000 segundos
```

4.2. Subproblema 2: Programación de Clases en Aulas y Horarios

- **Formalización**:
  El problema se modela como un grafo $G = (V, E)$, donde:
  - **Nodos** V: Representan clases c_i.
  - **Aristas** E: Conectan dos clases c_i y c_j si comparten un conflicto (mismo profesor, mismo grupo de estudiantes, uso de un aula en el mismo horario, etc.).
  - **Colores**: Cada çolorrepresenta una combinación única de **aula** r y **horario** t.
  - **Restricciones duras**:
    - Ningún profesor puede estar en dos lugares al mismo tiempo.
    - Ningún aula puede albergar dos clases simultáneamente.
    - Capacidad máxima del aula respetada.

  Objetivo: Asignar a cada clase c_i un par (r, t) tal que 2 nodos adyacentes (2 clases con conflictos) tengan el mismo color (combinació aula-horario).

- **Estrategia**:
  El problema se reduce a **coloración de grafos**, donde los colores son recursos (aulas y horarios). La heurística Greedy con ordenamiento por grado de conflicto garantiza una solución factible:
  1. **Ordenar las clases** por grado descendente (número de conflictos).
  2. **Asignar colores secuencialmente**, eligiendo el 1er color disponible que no viole las restricciones.

- **Algoritmo**: Heurística greedy con ordenamiento por grado de conflicto.
  Pasos:
  1. **Construir el grafo de conflictos**.
  2. **Ordenar nodos** de mayor a menor grado.
  3. **Asignar colores**: para cada clase c_i seleccionar el çolor"(r, t) más temprano posible (horario) que no esté asignado a sus vecinos en G.

- **Pseudogcódigo**:

```
Entrada: Grafo de conflictos G = (V, E)
Salida: Asignación de colores (aulas y horarios)

1. Ordenar V en orden descendente por grado.
2. Inicializar colores_disponibles = lista de combinaciones (r, t).
3. Para cada clase c_i en V:
   a. Para cada color en colores_disponibles:
      i. Si el color no está asignado a vecinos de c_i:
         - Asignar color a c_i.
```

```
        - Marcar color como usado.
        - Romper el bucle.
    4. Si una clase no puede ser asignada: Retornar "No hay solución".
```

- **Correctitud**:
  La heurística garantiza una solución válida debido a:
  - **Teorema de Welsh-Powell**: Si se ordenan los nodos por grado descendente, el número de colores necesarios es $\leq \Delta + 1$, donde Δ es el grado máximo del grafo.
  - **Evita conflictos**: Al asignar colores en orden de prioridad, se minimiza el riesgo de superposiciones.
- **Complejidad**:
  La complejidad es $O(n^2)$, donde n es el número de clases:
  - **Ordenamiento**: $O(n \log n)$
  - **Asignación de colores**: Para cada clase c_i, verificar los colores de sus vecinos ($O(n)$ por clase), en total, $O(n^2)$
- **Implementacón en Python**:
  Biblioteca útil: *networkx* para manejar grafos y coloreado (aunque aquí se implementa una versión simplificada).

```
import time

def schedule_classes(classes, conflicts):
    # Construir grafo de conflictos
    graph = {c: set() for c in classes}
    for c1, c2 in conflicts:
        graph[c1].add(c2)
        graph[c2].add(c1)

    # Ordenar clases por grado de conflicto descendente
    sorted_classes = sorted(classes, key=lambda x: len(graph[x]), reverse=True)

    # Asignar colores (combinaciones aula-horario)
    color_assignment = {}
    available_colors = [(r, t) for r in aulas for t in horarios]
    # Ejemplo: aulas y horarios predefinidos

    for c in sorted_classes:
        used_colors = {color_assignment[vecino] for vecino in graph[c]
        if vecino in color_assignment}
        for color in available_colors:
            if color not in used_colors:
                color_assignment[c] = color
                break
        if c not in color_assignment:
            return None  # No hay solución
```

```
    return color_assignment

# Ejemplo de uso
classes = ["C1", "C2", "C3", "C4"]
conflicts = [("C1", "C2"), ("C1", "C3"), ("C2", "C4")]
aulas = ["A1", "A2"]
horarios = ["T1", "T2"]

# Medir el tiempo de inicio
start_time = time.time()

schedule = schedule_classes(classes, conflicts)

# Medir el tiempo de fin
end_time = time.time()

# Calcular el tiempo de ejecución
execution_time = end_time - start_time

if schedule:
    for clase, (aula, hora) in schedule.items():
        print(f"{clase} -> Aula: {aula}, Hora: {hora}")
else:
    print("No se encontró solución.")

# Imprimir el tiempo de ejecución
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")
```

Entrada:

- *classes*: Lista de clases
- *conflicts*: Pares de clases que no pueden compartir aula u horario.
- *aulas y horarios*: Recursos disponibles (predefinidos).

Salida: Asignación de aulas y horarios para cada clase, o *None* si no hay solución.

```
C1 -> Aula: A1, Hora: T1
C2 -> Aula: A1, Hora: T2
C3 -> Aula: A1, Hora: T2
C4 -> Aula: A1, Hora: T1
Tiempo de ejecución: 0.000000 segundos
```

- **Limitaciones y mejoras**
  - **Optimalidad**: La heurística no garantiza el mínimo número de colores, pero es eficiente.
  - **Extensión**: Para restricciones adicionales (capacidad del aula), filtrar colores válidos en el paso 3a.

4.3. Subproblema 3: Manejo de Preferencias de Estudiantes

- **Formalización**: Optimizar restricciones blandas.
- **Estrategia**: Programación lineal entera (ILP) con relajación Lagrangiana.
- **Complejidad**: NP-Hard, pero relajaciones aproximadas en $O(n^3)$.

4.4. Subproblema 4: Condiciones Aleatorias (Eventos Imprevistos)

- **Formalización**: Replanificar ante cancelaciones de clases.
- **Estrategia**: Algoritmos online con ventanas deslizantes.
- **Análisis competitivo**: Cota de rendimiento frente al óptimo offline.

5. Cotas Mínimas y Análisis de Complejidad

- **Cota inferior para el problema general**: cualquier algoritmo exacto requiere $\Omega(2^n)$ en el peor caso (por la reducción a Graph Coloring).
- **Algoritmos de aproximación**: No se puede aproximar mejor que $O(n^{1-\epsilon})$ a menos que $P = NP$.

6. Estrategias de Solución Parcial/Total

- **Algoritmos Greedy**: Asignar primero las clases con mayor restricciones.
- **Algoritmos de Aproximación**: ILP con relajación lineal.
- **Metaheurísticas**: Algoritmos genéticos o simulated annealing.

7. Conclusión

Este informe ha presentado un enfoque estructurado para resolver el problema de planificación de horarios, demostrando su naturaleza NP-Hard y proponiendo estrategias para abordar subproblemas con análisis de complejidad y correctitud. Se recomienda implementar soluciones parciales usando enfoques híbridos o metaheurísticas para instancias grandes.