Algoritmos de Generación de Horarios Universitarios

Backend del Sistema de Scheduling

5 de diciembre de 2025

Índice

1. Introducción

Este documento describe los algoritmos implementados en el backend del generador de horarios universitarios. El sistema utiliza dos algoritmos principales: **Greedy** (voraz) para generación rápida y **Backtracking** (retroceso) para búsqueda exhaustiva con garantías de completitud.

2. Estructuras de Datos

2.1. Nodo (Node)

Representa una **hora-clase** que debe ser asignada. Cada materia con N horas semanales genera N nodos independientes.

Campo	Tipo	Descripción
id	string	Identificador único: "grupo-materia-unidad"
grupo_id	string	ID del grupo estudiantil
materia_id	string	ID de la materia
unit_index	int	Índice de la hora (0, 1, 2...)
possible_assignments	list	Lista de tuplas (día, slot, profesor)
assigned_day	int	Día asignado (-1 si no asignado)
assigned_slot	int	Slot asignado (-1 si no asignado)
assigned_prof	int	Índice del profesor asignado

Cuadro 1: Estructura del Nodo

Ejemplo: Una materia "Cálculo" con 4 horas semanales para el grupo "ISC-1A" genera 4 nodos:

- ISC-1A-calculo-0
- ISC-1A-calculo-1
- ISC-1A-calculo-2
- ISC-1A-calculo-3

2.2. Matrices de Estado

El sistema utiliza matrices NumPy para verificaciones en tiempo $O(1)$:

Matriz	Dimensiones	Propósito
prof_schedule	$[P \times 5 \times 9]$	1 si profesor ocupado, 0 si libre
group_schedule	$[G \times 5 \times 9]$	1 si grupo ocupado, 0 si libre
prof_load	$[P]$	Carga actual de cada profesor
prof_max_load	$[P]$	Carga máxima permitida
group_materia_day_count	$[G \times M \times 5]$	Horas por materia/grupo/día
group_materia_day_slots	$[G \times M \times 5]$	Bitmask de slots ocupados

Cuadro 2: Matrices de Estado (P = profesores, G = grupos, M = materias)

3. Algoritmo Greedy (Voraz)

3.1. Definición

El algoritmo **Greedy** es una estrategia de resolución de problemas que toma decisiones **localmente óptimas** en cada paso, con la esperanza de que estas decisiones conduzcan a una solución global aceptable.

3.2. Complejidad

$$O(N \times A) \tag{1}$$

Donde N = número de nodos y A = asignaciones posibles por nodo.

3.3. Funcionamiento Paso a Paso

1. **Ordenamiento inicial:** El sistema recibe una lista de "nodos" (horas-clase) que deben ser asignadas. Cada nodo representa una hora específica de una materia para un grupo determinado.

2. **Iteración secuencial:** El algoritmo recorre cada nodo en orden. Para cada nodo, tiene una lista de "asignaciones posibles" que son combinaciones válidas de (día, horario, profesor).

3. **Selección de la primera opción válida:** Para cada nodo, el algoritmo prueba las asignaciones posibles una por una. La primera que pase todas las validaciones se aplica inmediatamente.

4. **Sin retroceso:** Una vez que se asigna un nodo, esa decisión es **permanente**. El algoritmo nunca vuelve atrás para cambiar una asignación anterior.

5. **Manejo de fallos:** Si ninguna asignación es válida para un nodo, se deja sin asignar y se genera una advertencia.

3.4. Pseudocódigo

```
funcion solve_greedy():
    Para i desde 0 hasta cantidad_de_nodos:
        nodo = nodos[i]
        grupo_idx = obtener_indice_grupo(nodo.grupo_id)
        materia_idx = obtener_indice_materia(nodo.materia_id)

        asignado = Falso

        Para cada (dia, slot, profesor) en nodo.asignaciones_posibles:
            Si es_valido(nodo, grupo_idx, materia_idx, dia, slot,
   profesor):
                aplicar_movimiento(nodo, grupo_idx, materia_idx, dia,
   slot, profesor)
                asignado = Verdadero
                Salir del bucle interno

        Si NO asignado:
            Registrar advertencia

    Retornar Verdadero
```

Listing 1: Algoritmo Greedy

3.5. Ejemplo Ilustrativo

Supongamos que tenemos que asignar 3 horas de "Base de Datos" al grupo ISC-3A:

Nodo	**Opciones Disponibles**	**Decisión**
BD-hora1	(Lun-s1-Prof.López), (Lun-s2-Prof.García)	Lun-s1-Prof.López ✓
BD-hora2	(Lun-s2-Prof.López), (Mar-s1-Prof.López)	Lun-s2-Prof.López ✓
BD-hora3	(Mar-s1-Prof.López), (Mié-s1-Prof.López)	Mar-s1-Prof.López ✓

Cuadro 3: Ejemplo de asignación Greedy

3.6. Ventajas y Desventajas

Ventajas	**Desventajas**
Muy rápido (milisegundos)	No garantiza solución óptima
Determinista	Puede quedarse atascado
Bajo consumo de memoria	No explora alternativas
Siempre termina	Sensible al orden de entrada

Cuadro 4: Análisis del algoritmo Greedy

4. Algoritmo Backtracking (Retroceso)

4.1. Definición

El algoritmo **Backtracking** es una técnica de búsqueda exhaustiva que explora el espacio de soluciones de forma sistemática. A diferencia del Greedy, el Backtracking **puede deshacer decisiones** cuando detecta que un camino no lleva a una solución válida.

Es análogo a resolver un laberinto: si llegas a un callejón sin salida, retrocedes y pruebas otro camino.

4.2. Complejidad

$$O(A^N) \quad \text{(peor caso teórico)} \tag{2}$$

En la práctica es mucho menor gracias a las podas agresivas de las restricciones.

4.3. Funcionamiento Paso a Paso

1. **Estructura recursiva:** El algoritmo es una función que se llama a sí misma. Cada llamada intenta asignar un nodo específico (identificado por su índice).

2. **Caso base:** Si el índice del nodo es igual al total de nodos, significa que todos fueron asignados exitosamente. ¡Se encontró una solución completa!

3. **Exploración de opciones:** Para el nodo actual, el algoritmo prueba cada asignación posible (día, slot, profesor) una por una.

4. **Validación:** Antes de aplicar una asignación, verifica que cumpla todas las restricciones.

5. **Aplicar y recurrir:** Si la asignación es válida, la aplica y llama recursivamente a `backtrack`($nodo\_idx + 1$).

6. **Retroceso:** Si la llamada recursiva retorna `False`, el algoritmo **deshace** la asignación y prueba la siguiente opción.

7. **Agotamiento de opciones:** Si ninguna opción funciona, retorna `False`, causando que el nodo anterior también intente otra opción (efecto cascada).

4.4. Operaciones Apply y Undo

El backtracking requiere poder **revertir** cada decisión:

4.4.1. Aplicar Movimiento (apply_move)

```
prof_schedule[profesor][dia][slot] = 1       // Marcar ocupado
group_schedule[grupo][dia][slot] = 1         // Marcar ocupado
prof_load[profesor] += 1                     // Incrementar carga
group_materia_day_count[grupo][materia][dia] += 1
group_materia_day_slots[grupo][materia][dia] |= (1 << slot)  // Bitmask
```

4.4.2. Deshacer Movimiento (undo_move)

```
prof_schedule[profesor][dia][slot] = 0       // Liberar
group_schedule[grupo][dia][slot] = 0         // Liberar
prof_load[profesor] -= 1                     // Decrementar
group_materia_day_count[grupo][materia][dia] -= 1
group_materia_day_slots[grupo][materia][dia] &= ~(1 << slot)  // Limpiar
    bit
```

4.5. Control de Tiempo

Dado que el Backtracking puede tardar horas en casos complejos, implementamos un sistema de límite de tiempo:

1. **Contador de llamadas:** Se incrementa `call_count` en cada llamada recursiva.
2. **Verificación periódica:** Cada 1000 llamadas, se compara el tiempo transcurrido contra el límite.
3. **Bandera de abort:** Si se excede el tiempo, `time_limit_reached = True`.
4. **Propagación inmediata:** Todas las funciones verifican esta bandera y retornan inmediatamente.
5. **Mejor solución parcial:** Se guarda la mejor solución encontrada hasta el momento.

4.6. Pseudocódigo Completo

```
funcion backtrack(nodo_idx):
    call_count += 1

    // CONTROL DE TIEMPO
    Si call_count % 1000 == 0:
        Si time_limit_reached:
            Retornar Falso
        Si tiempo_actual() - tiempo_inicio > limite_tiempo:
            Imprimir "Tiempo limite alcanzado"
            time_limit_reached = Verdadero
            Retornar Falso

    // GUARDAR MEJOR SOLUCION PARCIAL
    Si nodo_idx > max_assigned_count:
        max_assigned_count = nodo_idx
        best_assignments = copiar_estado_actual()
        Imprimir "Nueva mejor solucion: {nodo_idx}/{total}"

    // CASO BASE: SOLUCION COMPLETA
    Si nodo_idx >= cantidad_de_nodos:
        Retornar Verdadero

    // VERIFICACION DE ABORT
    Si time_limit_reached:
        Retornar Falso

```

```
    // OBTENER NODO ACTUAL
    nodo = nodos[nodo_idx]
    grupo_idx = obtener_indice_grupo(nodo.grupo_id)
    materia_idx = obtener_indice_materia(nodo.materia_id)

    // EXPLORAR TODAS LAS OPCIONES
    Para cada (dia, slot, profesor) en nodo.asignaciones_posibles:

        Si time_limit_reached:
            Retornar Falso

        Si es_valido(nodo, grupo_idx, materia_idx, dia, slot, profesor):

            // PASO 1: Aplicar la asignacion
            aplicar_movimiento(nodo, grupo_idx, materia_idx, dia, slot,
    profesor)

            // PASO 2: Recurrir al siguiente nodo
            Si backtrack(nodo_idx + 1):
                Retornar Verdadero

            // PASO 3: Verificar abort
            Si time_limit_reached:
                deshacer_movimiento(...)
                Retornar Falso

            // PASO 4: Deshacer (BACKTRACK)
            deshacer_movimiento(nodo, grupo_idx, materia_idx, dia, slot,
     profesor)

    // NINGUNA OPCION FUNCIONO
    Retornar Falso
```

Listing 2: Algoritmo Backtracking

4.7. Ejemplo: El Poder del Retroceso

Consideremos un escenario donde el Greedy fallaría:
Escenario: 2 materias (A y B), 1 profesor común, solo 2 slots disponibles.

- Materia A: 1 hora, puede ir en slot 1 o slot 2
- Materia B: 1 hora, SOLO puede ir en slot 1

Greedy (falla):

1. Asigna A en slot 1 (primera opción válida)
2. Intenta asignar B... ¡slot 1 ocupado!
3. **Resultado:** B queda sin asignar ×

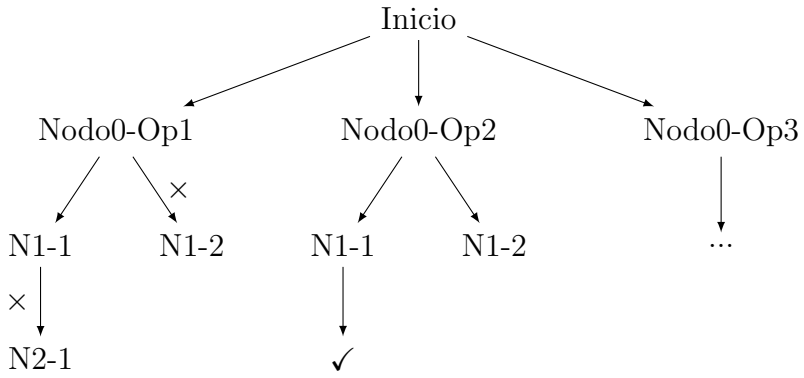
Backtracking (éxito):

1. Asigna A en slot 1
2. Intenta asignar B en slot 1... falla

3. **RETROCEDE:** Deshace A de slot 1
4. Asigna A en slot 2
5. Asigna B en slot 1... ¡éxito!
6. **Resultado:** Ambas materias asignadas ✓

4.8. Árbol de Decisiones

El backtracking puede visualizarse como un árbol donde cada nivel representa un nodo por asignar:



El algoritmo recorre el árbol en **profundidad primero** (DFS), retrocediendo cuando encuentra un callejón sin salida.

4.9. Ventajas y Desventajas

Ventajas	**Desventajas**
Encuentra solución si existe	Puede ser muy lento
Explora todo el espacio de búsqueda	Mayor consumo de memoria
Devuelve mejor solución parcial	Complejidad exponencial teórica
Configurable con límite de tiempo	

Cuadro 5: Análisis del algoritmo Backtracking

5. Comparación de Algoritmos

Criterio	Greedy	Backtracking
Tiempo disponible	Poco (segundos)	Más (minutos)
Importancia de completar	Baja	Alta
Complejidad del problema	Simple	Compleja
Restricciones	Pocas/relajadas	Muchas/estrictas
Uso recomendado	Vista previa rápida	Generación final

Cuadro 6: Comparación: ¿Cuándo usar cada algoritmo?

6. Restricciones Implementadas

Ambos algoritmos validan las siguientes restricciones antes de cada asignación:

1. **Profesor no ocupado:** $\texttt{prof\_schedule}[p][d][s] = 0$
2. **Grupo no ocupado:** $\texttt{group\_schedule}[g][d][s] = 0$
3. **Carga máxima de profesor:** $\texttt{prof\_load}[p] < \texttt{prof\_max\_load}[p]$
4. **Consistencia de profesor:** El mismo profesor debe dar todas las horas de una materia al mismo grupo
5. **Máximo 2 horas/día por materia:** $\texttt{group\_materia\_day\_count}[g][m][d] < 2$
6. **Contigüidad de bloques:** Si hay 2 horas de una materia en un día, deben ser consecutivas
7. **Máximo gap de 1 hora:** No puede haber huecos mayores a 1 slot entre clases
8. **Máximo 7 horas/día por grupo:** Previene días excesivamente largos

6.1. Validación de Contigüidad

Usa un **bitmask** para tracking eficiente:

```
Si count > 0:
    mask = group_materia_day_slots[grupo][materia][dia]
    adj_mask = (1 << (slot - 1)) | (1 << (slot + 1))
    Si (mask & adj_mask) == 0:
        Retornar Falso  // No es contiguo
```