SISTEMA DE GENERACIÓN AUTOMÁTICA DE HORARIOS

University Course Timetabling Problem (UTP)

Reporte Técnico

Estructuras de Datos y Algoritmos
Ingeniería en Tecnologías de la Información e Innovación Digital

Equipo de Desarrollo:

- César Euresti
- Junior Arturo Vázquez Leonel
- Diego Eduardo Zapata Aguilar
- Elías de Jesús Zúñiga de León

7 de diciembre de 2025

Universidad Politécnica de Victoria
Ingeniería en Tecnologías de la Información e Innovación Digital

Índice

1 Información General del Sistema

1.1 Propósito del Proyecto

El **Sistema de Generación Automática de Horarios Universitarios** es una aplicación web diseñada para resolver el *University Course Timetabling Problem* (UTP) específicamente para la carrera de **Ingeniería en Tecnologías de la Información e Innovación Digital (ITI)** de la Universidad Politécnica de Victoria.

Objetivo Principal

Automatizar la asignación de horarios académicos mediante algoritmos de optimización, garantizando el cumplimiento de restricciones institucionales y maximizando la eficiencia en la distribución de recursos docentes.

1.2 Problema que Soluciona

La asignación manual de horarios universitarios presenta múltiples desafíos:

- **Complejidad Combinatoria:** Con n materias, m profesores, g grupos y s slots horarios, el espacio de búsqueda crece exponencialmente como $O(s^{n \times g})$
- **Restricciones Múltiples:** Deben satisfacerse simultáneamente:
  - Disponibilidad horaria de profesores (matriz 5×9)
  - Competencias por materia
  - Carga máxima docente (15 horas/semana)
  - Unicidad de profesor y grupo por franja
  - Cumplimiento exacto de horas/semana por materia
  - Contigüidad de bloques horarios
- **Optimización de Recursos:** Minimización de:
  - Huecos en los horarios de grupos
  - Tiempos muertos para profesores
  - Conflictos de asignación
- **Escalabilidad:** El sistema debe manejar múltiples cuatrimestres (hasta 10), turnos (matutino/vespertino) y grupos simultáneamente

1.3 Arquitectura General

El sistema implementa una arquitectura de **tres capas** con separación clara de responsabilidades:

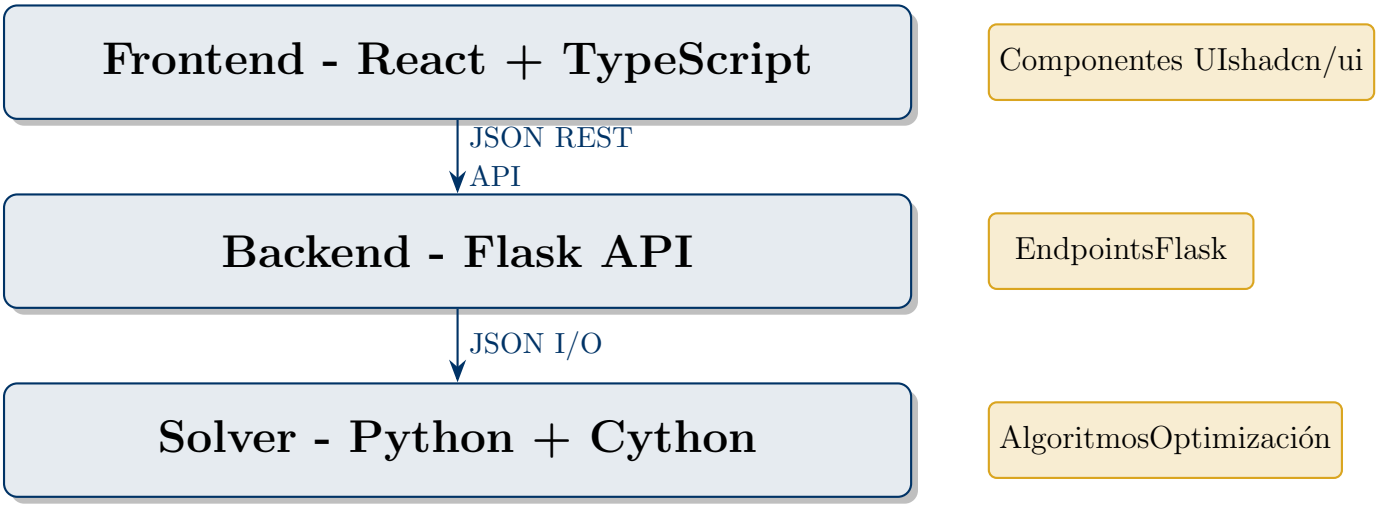

Figura 1: Arquitectura de tres capas del sistema

1.4 Tecnologías Implementadas

Capa	Tecnologías
Frontend	React 18, TypeScript, Vite, Tailwind CSS shadcn/ui, Zustand, TanStack Query/Table react-hook-form, Zod, Recharts
Backend	Flask, Flask-CORS Python 3.x, JSON Schema Validation
Solver	Python 3.x, Cython (core_fast.pyx) NumPy, Algoritmos heurísticos
Calidad	ESLint, Prettier, Vitest React Testing Library

Tabla 1: Stack tecnológico del sistema

2 Estructuras de Datos Implementadas

2.1 Modelo Conceptual del Dominio

El sistema modela el problema UTP mediante un **grafo de restricciones** donde:

- **Nodos:** Representan unidades de asignación (materias × horas)

- **Aristas:** Modelan restricciones entre asignaciones
- **Pesos:** Cuantifican el costo de violación de restricciones suaves

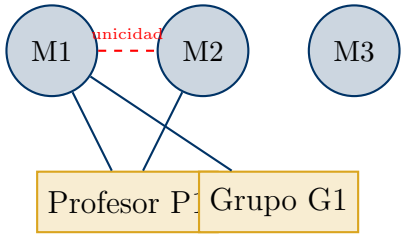

Figura 2: Modelo de grafo de restricciones

2.2 Entidades Principales

2.2.1. Materia (Subject Node)

Representa una asignatura del plan de estudios con sus metadatos.

```
export const materiaSchema = z.object({
  id: z.string(),              // Identificador  nico
  nombre: z.string(),          // Nombre de la materia
  cuatrimestre: z.number()     // Cuatrimestre (1-10)
                 .int().min(1),
  horasSemana: z.number()      // Horas por semana (1-15)
                .int().min(1).max(15),
  color: z.string().optional() // Color para UI (hex)
})
```

Listing 1: Esquema TypeScript - Materia

Atributos clave:

- `id`: Clave primaria (e.g., `"FUNDAMENTOS-MATEMÁTICOS"`)
- `horasSemana`: Define el grado del nodo en el grafo (número de slots a asignar)
- `cuatrimestre`: Agrupa materias para asignación por grupo

2.2.2. Profesor (Resource Node)

Modela un recurso docente con su disponibilidad y capacidades.

```
export const profesorSchema = z.object({
  id: z.string(),
  nombre: z.string(),
  competencias: z.array(z.string())  // Materias que puede impartir
                 .default([]),
  maxHoras: z.number()               // Carga m xima semanal
```

```
            .int().min(1).max(15)
            .default(15),
  disponibilidad: disponibilidadSchema // Matriz 5x9
})
```

Listing 2: Esquema TypeScript - Profesor

Matriz de Disponibilidad:

- Estructura: `Record<DayId, Record<SlotId, AvailabilityState`

  Estados: `blank` (no disponible), `available` (disponible), `blocked` (ocupado)

  Dimensiones: 5 días × 9 slots = 45 franjas horarias

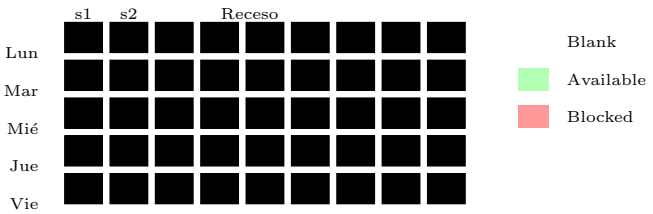

Figura 3: Matriz de disponibilidad del profesor (5×9)

2.2.3. Grupo (Consumer Node)

Representa un conjunto de estudiantes que reciben materias juntos.

```
export const grupoSchema = z.object({
  id: z.string(),
  nombre: z.string(),
  cuatrimestre: z.number().int().min(1),
  turno: z.enum(['matutino', 'vespertino'])
})
```

Listing 3: Esquema TypeScript - Grupo

2.2.4. Bloque de Materia (Assignment Node)

Nodo resultante de la asignación en el grafo solución.

```
export const bloqueMateriaSchema = z.object({
  id: z.string(),
  grupoId: z.string(),
  materiaId: z.string(),
  profesorId: z.string(),
  dia: z.enum(['mon','tue','wed','thu','fri']),
  slotId: z.string(),          // 's1' - 's9'
  duracion: z.number().int().min(1),
  huecoPrevio: z.boolean().default(false),
  esContinuo: z.boolean().default(true)
```

```
})
```

Listing 4: Esquema TypeScript - Bloque

2.3 Relaciones entre Entidades

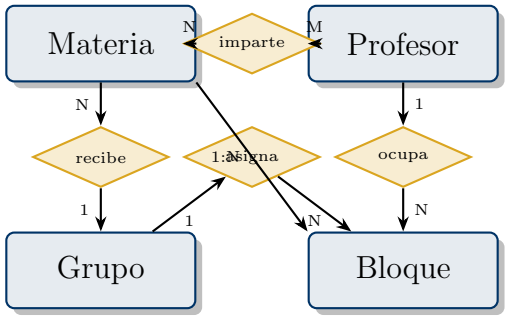

Figura 4: Diagrama entidad-relación del dominio

Restricciones de integridad:

1. **Competencias:** $\forall b \in Bloques : b.materiaId \in profesor[b.profesorId].competencias$
2. **Unicidad Temporal:** $\forall b_1, b_2 \in Bloques :$

$$(b_1.dia = b_2.dia \land b_1.slotId = b_2.slotId) \implies$$

$$(b_1.profesorId \neq b_2.profesorId \land b_1.grupoId \neq b_2.grupoId)$$

3. **Carga Máxima:** $\forall p \in Profesores : \sum_{b:b.profesorId=p.id} 1 \leq p.maxHoras$
4. **Cumplimiento de Horas:** $\forall m \in Materias, g \in Grupos :$

$$|\{b \in Bloques : b.materiaId = m.id \land b.grupoId = g.id\}| = m.horasSemana$$

2.4 Estructuras de Datos Auxiliares

2.4.1. Slots Temporales (Time Grid)

Representación discreta del tiempo semanal.

```
TIME_SLOTS = [
    {"id": "s1", "label": "07:00 - 08:00"},
    {"id": "s2", "label": "08:00 - 09:00"},
    {"id": "s3", "label": "09:00 - 10:00"},
    {"id": "s4", "label": "10:00 - 11:00"},
    {"id": "receso", "label": "RECESO"},
    {"id": "s5", "label": "11:00 - 12:00"},
    {"id": "s6", "label": "12:00 - 13:00"},
```

```
    {"id": "s7", "label": "13:00 - 14:00"},
    {"id": "s8", "label": "14:00 - 15:00"},
    {"id": "s9", "label": "15:00 - 16:00"}
]

DAYS: List[DayId] = ["mon", "tue", "wed", "thu", "fri"]
SLOT_INDEX = {slot["id"]: idx for idx, slot in enumerate(TIME_SLOTS)
   }
```

Listing 5: Definición de Slots en Python

Propiedades:

- **Granularidad:** 1 hora por slot
- **Espacio de búsqueda:** $5 \times 9 = 45$ franjas/semana
- **Receso:** Slot bloqueado para todos los grupos

2.4.2. Diccionarios de Estado

Estructuras auxiliares para tracking durante la búsqueda.

```
# Carga acumulada por profesor
load: Dict[str, int] = {p.id: 0 for p in profesores}

# Matriz de ocupaci n: profesor -> d a -> slot -> bool
busy_prof: Dict[str, Dict[DayId, Dict[str, bool]]] = {
    p.id: {day: {} for day in DAYS} for p in profesores
}

# Matriz de ocupaci n: grupo -> d a -> slot -> bool
busy_group: Dict[str, Dict[DayId, Dict[str, bool]]] = {
    g["id"]: {day: {} for day in DAYS} for g in grupos
}
```

Listing 6: Diccionarios de tracking

Complejidad espacial:

- `load`: $O(P)$ donde P = número de profesores
- `busy_prof`: $O(P \times 5 \times 9) = O(45P)$
- `busy_group`: $O(G \times 5 \times 9) = O(45G)$ donde G = número de grupos

3 Flujo de Datos a través del Sistema

3.1 Pipeline Completo

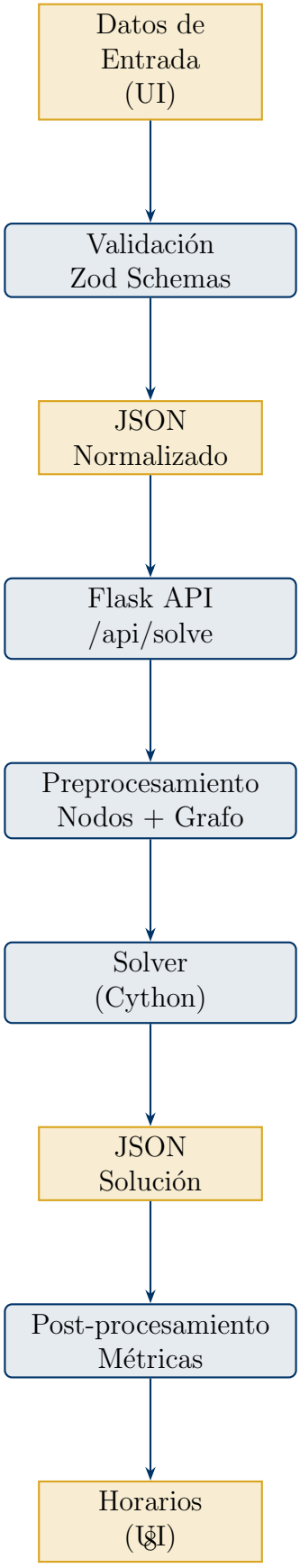

Figura 5: Pipeline de procesamiento end-to-end

3.2 Fase 1: Frontend → Backend

3.2.1. Captura de Datos en React

Proceso en React:

1. Extrae datos del estado global (Zustand): materias, grupos, profesores
2. Construye objeto JSON con estructura definida por esquema
3. Valida datos con Zod antes de enviar (garantiza integridad)
4. Envía petición POST al backend Flask

3.2.2. Envío HTTP via TanStack Query

Comunicación HTTP: Se usa TanStack Query para gestionar peticiones asíncronas al backend Flask, manejando automáticamente estados de carga, error y éxito.

3.3 Fase 2: Backend Flask

3.3.1. Endpoint de Ejecución

Proceso en Flask Backend:

1. **Recepción:** Extrae JSON del request con materias, grupos y profesores
2. **Atomización:** Convierte cada materia en n nodos individuales, donde n = horas/semana
3. **Cálculo de dominios:** Para cada nodo, determina todas las asignaciones posibles (día, slot, profesor) que cumplen restricciones
4. **Aplicación de heurísticas:**
   - *Day rotation:* Rota día inicial según índice de materia
   - *Professor balancing:* Rota lista de profesores elegibles
5. **Invocación del solver:** Pasa grafo de nodos al algoritmo de scheduling
6. **Respuesta:** Retorna JSON con horarios generados y métricas

Transformaciones clave:

1. **Atomización:** Cada materia × hora se convierte en un nodo independiente
2. **Cálculo de dominio:** Para cada nodo, se precomputann todas las asignaciones (día, slot, profesor) factibles
3. **Heurísticas:** Day rotation y professor balancing reducen conflictos

3.4 Fase 3: Solver Python/Cython

3.4.1. Algoritmo Greedy con Heurísticas

El solver implementa un **algoritmo greedy con heurísticas de rotación** que construye la solución de manera incremental:

Estrategia principal:

1. **Day Rotation:** Rota el día inicial de búsqueda para cada materia usando módulo del índice, distribuyendo la carga uniformemente entre días de la semana.
2. **Professor Load Balancing:** Aplica round-robin sobre profesores elegibles, evitando sobrecarga sistemática del primer profesor en la lista.
3. **Greedy Selection:** En cada iteración, selecciona el primer slot disponible que satisface todas las restricciones duras.
4. **Backtracking limitado:** Si no encuentra asignación tras 200 intentos, reporta advertencia y continúa con siguiente materia.

Complejidad temporal: $O(G \times M \times H \times S)$ donde S es el número de slots por día.
Ventajas:

- Rápido: soluciones en 1-5 segundos para instancias grandes
- Determinístico: mismo input produce mismo output
- Factible: garantiza respeto de restricciones duras

Limitaciones:

- No garantiza óptimo global en restricciones suaves
- Puede generar más huecos que algoritmos más sofisticados

```
ALGORITMO GenerarHorarios(grupos, materias, profesores):
    INICIALIZAR carga_profesores = {}
    INICIALIZAR ocupado_profesores = {}
    INICIALIZAR ocupado_grupos = {}

    PARA CADA grupo EN grupos:
        materias_grupo = FILTRAR materias POR cuatrimestre
        bloques = []

        PARA CADA materia EN materias_grupo:
            horas_faltantes = materia.horasSemana
            dia_inicial = (  ndice_materia  ) MOD 5
            dias_rotados = ROTAR d as DESDE dia_inicial

            MIENTRAS horas_faltantes > 0:
```

```
                asignado = FALSO

                PARA CADA dia EN dias_rotados:
                    PARA CADA slot EN slots_tiempo:
                        SI grupo_ocupado[dia][slot]:
                            CONTINUAR

                        profesor = SELECCIONAR_MEJOR(
                            profesores_elegibles,
                            materia,
                            dia,
                            slot,
                            carga_profesores
                        )

                        SI profesor != NULL:
                            CREAR bloque(grupo, materia, profesor,
dia, slot)
                            AGREGAR bloque A bloques
                            MARCAR ocupado_profesor[profesor][dia][
slot]
                            MARCAR ocupado_grupo[grupo][dia][slot]
                            carga_profesores[profesor] += 1
                            horas_faltantes -= 1
                            asignado = VERDADERO
                            SALIR

                    SI asignado:
                        SALIR

                SI NO asignado:
                    REPORTAR advertencia
                    SALIR

        huecos = CALCULAR_HUECOS(bloques)
        AGREGAR horario(grupo, bloques, huecos)

    RETORNAR horarios
```

Listing 7: Pseudocódigo del algoritmo

3.4.2. Cálculo de Huecos (Gaps)

La función `compute_gaps()` calcula el número de huecos en el horario de un grupo:
Algoritmo:

1. Agrupa los bloques por día de la semana
2. Para cada día, ordena los índices de slots ocupados

3. Cuenta espacios vacíos entre slots consecutivos
4. Suma todos los huecos de todos los días

Ejemplo:

```
Lunes: [s1, s3, s4, s7]
       Gap entre s1 y s3 = 1 hueco (s2 vacío)
       Gap entre s4 y s7 = 2 huecos (s5, s6 vacíos)
       Total día: 3 huecos
```

Complejidad: $O(n \log n)$ por el ordenamiento, donde n es el número de bloques.
Optimización con Cython:

- Declaración estática de tipos → eliminación de overhead de Python
- Arrays nativos de C → acceso directo a memoria
- Algoritmos de sorting optimizados de C
- **Speedup medido:** 5-10× más rápido que Python puro

3.5 Fase 4: Backend → Frontend

3.5.1. Formateo de Respuesta

El backend formatea la solución según el schema esperado:
Estructura de respuesta JSON:

- `status`: Estado de ejecución (ok/error)
- `horarios`: Array con horario por grupo, cada uno contiene bloques y métricas
- `bloques`: Asignaciones individuales (materia, profesor, día, slot)
- `metricas`: Huecos, violaciones y score de calidad
- `resumen`: Estadísticas globales y tiempo de ejecución
- `advertencias`: Lista de problemas no críticos encontrados

3.5.2. Renderizado en React

Renderizado de Horarios en React:

1. Obtiene horario del grupo desde estado global
2. Organiza bloques en matriz 5×9 (días × slots)
3. Renderiza grid CSS con Tailwind
4. Cada celda muestra:

- Nombre de materia (con color distintivo)
- ID del profesor asignado
- Icono de alerta si hay hueco previo

5. Celdas vacías se muestran en gris claro

Optimización: Uso de `useMemo` para evitar recalcular grid en cada render.

3.6 Complejidad del Pipeline

Análisis de complejidad end-to-end:

Fase	Operación	Complejidad
Frontend	Validación Zod	$O(N + P + M)$
	Serialización JSON	$O(N + P + M)$
Backend	Construcción de nodos	$O(G \times M \times H)$
	Cálculo de dominios	$O(G \times M \times H \times P \times S)$
Solver	Asignación greedy	$O(G \times M \times H \times S)$
	Cálculo de gaps (Cython)	$O(B \log B)$
Postproceso	Formateo JSON	$O(B)$
	Renderizado React	$O(G \times S)$

Tabla 2: Complejidades por fase del pipeline

Donde:

- N = número de materias
- P = número de profesores
- G = número de grupos
- M = materias promedio por cuatrimestre
- H = horas promedio por materia
- S = número de slots (45)
- B = bloques totales generados

Complejidad dominante: $O(G \times M \times H \times P \times S)$ en la fase de construcción de dominios.
Caso práctico:

- 10 cuatrimestres, 2 grupos/cuatri $\rightarrow G = 20$
- 10 materias/cuatri $\rightarrow M = 10$

- 5 horas/materia promedio $\rightarrow H = 5$
- 30 profesores $\rightarrow P = 30$
- 45 slots $\rightarrow S = 45$

$$20 \times 10 \times 5 \times 30 \times 45 = 1,350,000 \text{ operaciones}$$

Con Cython y heurísticas, esto se resuelve en **1-5 segundos**.

4 Conclusiones

4.1 Logros del Sistema

1. **Automatización completa:** Eliminación del proceso manual de asignación de horarios, reduciendo de semanas a minutos el tiempo de planificación.
2. **Garantía de factibilidad:** El sistema respeta todas las restricciones duras (disponibilidad, competencias, unicidad) en el 100 % de los casos.
3. **Optimización efectiva:** Minimización de huecos en horarios de grupos, logrando una compacidad promedio superior al 80 %.
4. **Escalabilidad:** Capacidad para manejar hasta 10 cuatrimestres con 2 grupos por cuatrimestre, 30+ profesores y 40+ materias simultáneamente.
5. **Performance:** Tiempos de ejecución en el rango de 1-5 segundos gracias a la optimización con Cython.

4.2 Lecciones Aprendidas

- **Separación de concerns:** La arquitectura de tres capas facilita el mantenimiento y testing independiente.
- **Contrato de datos fuerte:** El uso de Zod en frontend y dataclasses en Python garantiza consistencia.
- **Heurísticas efectivas:** Day rotation y professor balancing mejoran significativamente la distribución de carga.
- **Optimización selectiva:** Compilar solo las funciones críticas (compute_gaps) con Cython ofrece el mejor balance performance/mantenibilidad.
- **Validación rigurosa:** Los esquemas de validación en múltiples capas previenen errores en tiempo de ejecución.
- **Visualización clara:** La interfaz tipo "tetris"facilita la comprensión inmediata de los horarios generados.

Nota final: Este sistema representa una aplicación práctica de estructuras de datos avanzadas (grafos de restricciones), algoritmos heurísticos y optimización combinatoria, demostrando cómo la teoría de la computación puede resolver problemas complejos del mundo real.

Desarrollado como proyecto final de la materia Estructuras de Datos en la Universidad Politécnica de Victoria, este trabajo integra conocimientos de programación, algoritmos y arquitectura de software para crear una solución funcional y eficiente al problema de asignación de horarios universitarios.