

# TP2 - Gestión eficiente de recursos en sistemas ferroviarios

Candelaria Sutton, Dafydd Jenkins, Josefina Jahde

November 17, 2024

## Introducción

En este trabajo, abordamos el problema de la planificación y gestión óptima del material rodante para una empresa ferroviaria. El problema consiste en definir una asignación eficiente de material rodante a las distintas estaciones cabecera de una línea ferroviaria. Se busca minimizar el número total de unidades de material rodante necesarias para cubrir la demanda de la línea en cada horario, considerando que es posible la reutilización de unidades entre viajes.

Este problema pertenece a la categoría de problemas de circulación en redes y tiene gran relevancia en la optimización de recursos en operaciones logísticas y de transporte. En el contexto de la industria ferroviaria, una asignación ineficiente del material rodante (trenes) puede llevar a costos innecesarios y al uso ineficiente de los recursos disponibles, afectando tanto a la empresa como a los usuarios del servicio.

El objetivo de este trabajo es desarrollar un modelo que permita resolver el problema de circulación de trenes utilizando algoritmos de flujo de costo mínimo. Además, se implementará un set de experimentos para analizar el rendimiento del modelo en diferentes escenarios de demanda y costos.

En este informe se pueden encontrar las secciones:

- Sección 1: se describe el problema, las restricciones y las consideraciones del mismo.
- Sección 2: se describe la metodología, explicando cómo se modeló el problema y se implementó dicho modelo.
- Sección 3: detalla los experimentos realizados, presentando hipótesis, diseño de pruebas y resultados obtenidos.
- Sección 4: se presentan las conclusiones, discutiendo la efectividad del modelo y posibles extensiones para trabajos futuros.

## Sección 1: El problema

El problema consiste en definir una asignación eficiente de material rodante a las distintas estaciones cabecera de una línea ferroviaria. Se busca minimizar el número total de unidades de material rodante necesarias para cubrir la demanda de la línea en cada horario, considerando que es posible la reutilización de unidades entre viajes. Para cada viaje que se realiza en la línea durante un día, se tiene la demanda en cantidad de pasajeros y las estaciones y horarios de salida y llegada. Además se cuenta con restricciones: cantidad máxima de unidades que pueden circular por la línea y la capacidad máxima (en pasajeros) de cada unidad.

Se toma una versión simplificada con las siguientes consideraciones:

- Los trasposos de unidades ocurren únicamente en las cabeceras de la línea.
- Se pueden almacenar infinita cantidad de unidades de material rodante en las cabeceras.
- Todos los servicios tienen el mismo tipo de unidad rodante (en cuanto a cantidad de pasajeros).

## Sección 2: Modelado e implementación

Para la resolución del problema, se adaptó el modelo propuesto por Alexander Schrijver REFERENCIA!!!. Dicho modelo se realiza a partir del cronograma, la información sobre la demanda y las restricciones de los servicios que recorren una línea durante el día, usando un digrafo que modele una red espacio-tiempo. Se formula la decisión como un problema de Circulación sobre la red que se reduce a un problema de Flujo de Costo Mínimo y se resuelve utilizando algoritmos de flujo. Tanto para la construcción de la red como para los algoritmos de flujo se utilizó la librería *NetworkX* de Python.

### Creación del Grafo

La resolución comienza con la construcción del digrafo de la red. El mismo contiene un conjunto de vértices  $V$ , donde cada vértice representa un evento de llegada o salida a una estación. Dicho nodo tiene como etiqueta el nombre de la estación y el horario correspondiente y posee un imbalance igual a 0. El conjunto  $A$  de arcos del digrafo está compuesto por tres tipos:

- de tren:  $(i, j)$  conecta el evento de partida y el de arribo de un servicio, representando el viaje del servicio desde su estación origen a su estación destino. Estos arcos cuentan con una cota inferior  $l_{ij}$  de la cantidad de unidades necesarias para satisfacer la demanda y una cota superior  $u_{ij}$  de la cantidad máxima de unidades que pueden circular en un servicio. El costo es  $c_{ij} = 0$ .
- de traspaso:  $(i, j)$  conecta dos eventos consecutivos (en tiempo) dentro de una estación, indicando el pasaje de unidades entre eventos. Esta arista tiene cota inferior  $l_{ij} = 0$ ,  $u_{ij} = \infty$  y costo  $c_{ij} = 0$ .
- de trasnoche: dos de ellas, una para cada estación, conectan el último evento del día con el primero. Modela las condiciones de borde respecto que la cantidad de unidades al finalizar el día en una estación debe ser la necesaria para poder iniciar las operaciones al día siguiente.

Otros dos arcos conectan el último evento de una estación con el primero de la otra. Modela el traspaso de unidades al final de un día entre estaciones para la utilización al comienzo del siguiente.

Inicialmente, todos estos arcos tienen  $l_{ij} = 0$ ,  $u_{ij} = \infty$  y  $c_{ij} = 1$ . Además, estos arcos son los utilizados para obtener la cantidad de unidades de material rodantes necesarias en la línea.

La construcción del grafo se realiza con el método `create_graph()` que, al igual que todos los demás métodos de esta implementación, se encuentra implementado en el archivo `railway_service.py`.

### Entrada del Algoritmo

El algoritmo recibe un diccionario `data` con la siguiente estructura:

- `data["services"]`: Diccionario que contiene la información de los servicios (horarios de salida y llegada, demanda).
- `data["rs_info"]["capacity"]`: Capacidad del material rodante (RS).
- `data["rs_info"]["max_rs"]`: Máximo número de unidades de RS disponibles.
- `data["stations"]`: Lista de estaciones, en este caso, se consideran las estaciones `a` y `b`.

### Proceso del Algoritmo

1. **Inicialización:** Se crea un grafo dirigido  $G$  utilizando *NetworkX* y se define un diccionario `nodos_estacion` para almacenar los tiempos de salida y llegada de cada estación.
2. **Creación de Nodos y Aristas de Servicios:** Para cada servicio en `data["services"]`, se extraen los horarios de salida y llegada. Se calculan los RS necesarios dividiendo la demanda del servicio por la capacidad de los RS. Se crean nodos para los tiempos de salida y llegada del servicio y se añade una arista dirigida entre ellos, con:
  - **Peso (weight):** Inicialmente 0, ya que no hay costo asociado al servicio.

- **Cota mínima (lower\_bound):** Igual al número de RS necesarios para el servicio.
  - **Cota máxima (upper\_bound):** Igual al máximo número de RS que pueden circular.
3. **Creación de Aristas Internas en las Estaciones:** Para cada estación, se ordenan los tiempos de salida y llegada. Se crean aristas dirigidas entre tiempos consecutivos dentro de la misma estación para modelar el flujo continuo de trenes, con:
    - **Peso (weight):** 0, ya que no hay costo asociado al traspaso de unidades entre servicios de una misma estación.
    - **Cota mínima (lower\_bound):** 0.
    - **Cota máxima (upper\_bound):** Se fija en un valor alto (1e9).
  4. **Creación de Aristas Cíclicas:** Para garantizar la circulación continua del material rodante, se crean aristas entre el último y el primer nodo de cada estación. Estas aristas tienen un costo (**weight**) de 1, representando el costo adicional de mover unidades de RS entre servicios.
  5. **Aristas de Conexión entre Estaciones:** Se agregan aristas de conexión entre las estaciones **a** y **b** para modelar los trasposos posibles entre estaciones al final del día. Estas aristas también tienen un costo (**weight**) de 1 para representar el costo de los trasposos nocturnos.

### Salida del Algoritmo

El método devuelve:

- **G:** El grafo dirigido que modela el problema de circulación, con nodos y aristas etiquetados con las demandas y capacidades correspondientes.
- **nodos\_estacion:** Un diccionario con los tiempos de salida y llegada de cada estación, ordenados de menor a mayor.

### Resolución del problema de Circulación mediante reducción al problema de Flujo de Costo Mínimo

Para la reducción del problema de Circulación a un problema de Flujo de Costo Mínimo, se toma el grafo construido y se construye uno nuevo con las siguientes modificaciones:

- Para cada nodo  $i$  el imbalance  $b_i$  es:

$$b_i = \sum_{j \in N^-(i)} l_{ji} - \sum_{j \in N^+(i)} l_{ij}$$

- Para cada arco, su nueva capacidad  $\hat{u}_{ij}$  será  $\hat{u}_{ij} = u_{ij} - l_{ij}$

Sea  $\hat{x}^*$  la solución del problema de flujo de costo mínimo definido anteriormente,  $x_{ij}^* = l_{ij} + \hat{x}_{ij}^*$  es una solución óptima del problema de circulación original.

La transformación anteriormente descrita se implementa en el método `solve_circulacion()`.

### Entrada del Algoritmo

El algoritmo recibe como parámetro el grafo **G** del problema de Circulación.

### Proceso del Algoritmo

1. **Inicialización:** Se crea un digrafo **H** utilizando el método `transform_graph()` que realiza la transformación del grafo según se describió anteriormente. Este método define **H** como digrafo, le agrega los nodos de **G** y las aristas del mismo pero con la capacidad correspondiente según la transformación. Luego modifica los imbalances de los nodos según la transformación, recorriendo todo arco y sumándole al nodo de salida el  $l_{ij}$  y restándoselo al nodo de entrada.

2. **Resolución del problema de Flujo de Costo Mínimo:** Para el grafo  $H$  se resuelve el problema de Flujo de Costo Mínimo usando el método `min_cost_flow()` de `NetworkX`. Este método devuelve un diccionario que contiene para cada arco saliente de cada nodo de  $H$  el flujo que circula por dicho arco. Dicho diccionario se almacena en `circulacion`.
3. **Transformación del Flujo de Costo Mínimo a la Circulación:** Se recorren los nodos  $u$  del flujo de los que salen arcos. Para cada uno de sus vecinos  $v$ , se modifica `circulacion` sumándole a la arista  $u \rightarrow v$  el  $l_{ij}$ .

### Salida del Algoritmo

Se retorna el diccionario `circulacion` que contiene, para cada arco saliente de cada nodo, el flujo que circula por dicho arco.

### Generación de la respuesta final (cantidad de unidades necesarias)

Para calcular la cantidad total de unidades necesarias, se suma el flujo circulante por las aristas de trasnoche. Para ello se utiliza el método implementado `get_cost()`.

### Entrada del Algoritmo

El algoritmo recibe como parámetros

- el grafo  $G$  del problema de Circulación.
- el diccionario `circulación` resultante de `solve_circulacion()`.
- el diccionario `nodos_estacion` resultante de `create_graph()` (para obtener las etiquetas del primer y último nodo de cada estación).

### Proceso del Algoritmo

1. **Cálculo del costo resultante:** Se inicializa el costo resultante `cost` inicialmente igual a 0. Sean las estaciones  $x, y$  se recorren las posibles combinaciones  $(a, b)$  para  $a, b \in \{x, y\}$ . Para cada combinación se define `begin` como el primer nodo de  $a$ , y `end` como el último nodo de  $b$ . Finalmente, se suma a `cost` el flujo circulante por el arco `end`  $\rightarrow$  `begin`.

### Salida del Algoritmo

Se retorna `cost`, el costo resultante de sumar el flujo circulante por las aristas de trasnoche.

### Otros métodos

Se implementaron otros métodos para facilitar la implementación de los algoritmos, la experimentación y el uso por parte de los usuarios. A continuación se describe la funcionalidad que cumple cada uno y el modo de uso esperado para aquellos usuarios que desearan realizar sus propias experimentaciones.

- `load_instance()`: recibe la ruta un archivo json, a partir del cual generar una instancia de cronograma para resolver el problema.
- `show_graph()`: recibe un grafo  $G$  y el diccionario `nodos_estacion` generados por el método `create_graph()` y una instancia de cronograma `data`. Muestra el grafo ubicando los nodos de cada estación por columnas. Muestra en los arcos la cota mínima  $l$  y capacidad  $u$ .
- `show_flow()`: recibe un grafo  $G$  y el diccionario `nodos_estacion` generados por el método `create_graph()`, una instancia de cronograma `data` y el flujo `flow` resultante del método `solve_circulacion()`. Muestra el flujo en el grafo  $G$ .
- `modelo_circulacion()`: recibe una instancia de cronograma `data`. Resuelve el problema de circulación para la instancia haciendo uso de los métodos `create_graph()`, `solve_circulacion()` y `get_cost()`, y devuelve el la cantidad de unidades necesarias para resolver el problema.

- **modelo\_empresa()**: recibe una instancia de cronograma **data**. Resuelve el problema utilizando el método de la empresa y devuelve la cantidad de unidades necesarias. Para ello,
  1. Se inicializan contadores para el stock y la cantidad de unidades nuevas de cada estación en 0.
  2. Se almacenan los eventos de **data** como tuplas y se las ordena según el horario.
  3. Para cada evento, si es una llegada, se suman las unidades recibidas (iguales a lo mínimo necesario para cubrir la demanda) al stock de la estación. Si es una salida, se trata de cubrir la demanda con el stock disponible. Si este es insuficiente, el remanente se obtiene incrementando la cantidad de unidades nuevas.
  4. Se devuelve la cantidad de unidades nuevas para cada estación.

Los algoritmos desarrollados e implementados se testearon con la instancia **toy\_instance** propuesta por la cátedra y otras instancias que representan distintos casos bordes. Dichas instancias se encuentran en la carpeta **test** y son:

- **catch\_and\_shoot**: caso en el que cada salida de una estación se puede cubrir con stock existente en la misma proveniente de una llegada previa.
- **descordinados**: caso en el que cada salida necesitas generar unidades nuevas, es decir, no hay forma de optimizar la cantidad de unidades mediante el traspaso de las mismas entre eventos.
- **one\_sends**: caso en el que solo una estación envía unidades.

## Sección 3: Experimentación

Se realizó una serie de experimentos