1 Introduction

The Test Case Compression problem asks us to construct a set of pipelines that minimize total cost while satisfying two requirements:

1. Each flow must appear at least its required number of times.
2. For every vertex v and each of its preconditions $p \in P_v$, there must exist at least one pipeline where p precedes v.

The cost of a pipeline is defined as the sum of the costs of all distinct vertices it traverses, respecting location constraints.

Our solution is divided (as requested) into two parts:

- **U_sub**: select a minimum-cost subset of the original pipelines, using a greedy reduction strategy that removes expensive pipelines while ensuring that flow multiplicities and precondition coverage remain satisfied.
- **U_new**: construct additional pipelines from scratch by translating flow quotas into a degree-balanced graph, solving it with a cost-guided max-flow algorithm, and converting the result into pipelines via Eulerization and rebalancing.

This two-phase approach leverages the structure of the input while guaranteeing feasibility through a robust graph-based synthesis procedure.

2 Model and Formulation

The problem is modeled using three main entities: vertices, flows, and pipelines.

2.1 Vertices

Each vertex v is defined by:

- A cost $c_v \in \mathbb{N}$.
- A location tag $\ell_v \in \{0, 1, 2\}$, where 0 indicates a start-only vertex, 2 an end-only vertex, and 1 a mid vertex.
- A set of preconditions $P_v \subseteq V$. For every $p \in P_v$, there must exist at least one pipeline in which p precedes v.

2.2 Flows

A flow f is an ordered sequence of vertices:

$$f = (v_1, v_2, \ldots, v_k),$$

with the following properties:

- $s_f = v_1$ is the start vertex, $t_f = v_k$ is the end vertex.
- Each flow has a required multiplicity w_f, i.e. it must appear at least w_f times across all pipelines.
- The cost of the flow is the sum of the costs of its vertices, respecting location constraints.
- A flow has a location tag $\ell_f \in \{0, 1, 2\}$, where 0 indicates a start-only flow, 2 an end-only flow, and 1 a mid flow. This is defined only by s_f and t_f

2.3 Pipelines

A pipeline is a concatenation of flows $(f_1, f_2, \ldots, f_m)$ such that the endpoint of f_i coincides with the start of f_{i+1}.

The cost of a pipeline is defined as the sum of the costs of its distinct vertices, with shared junction vertices counted only once.

2.4 Constraints

The solution must output a set of pipelines $\mathcal{P}$ such that:

1. For every flow f, the total number of appearances of f in $\mathcal{P}$ is at least w_f.
2. For every vertex v and each $p \in P_v$, there exists at least one pipeline $\pi \in \mathcal{P}$ where p precedes v.

2.5 Objective

Minimize the total cost:

$$\text{Cost}(\mathcal{P}) = \sum_{\pi \in \mathcal{P}} \sum_{v \in \pi} c_v,$$

where each vertex cost c_v is counted once per pipeline in which it appears.

3 Subset Selection (U_sub)

The first part of the problem is to select a minimum-cost subset of the original pipelines provided in the input.

This is achieved by preprocessing the contribution of each pipeline and then applying a greedy reduction strategy.

3.1 Preprocessing

For every original pipeline i, we compute two types of information:

1. **Flow multiplicities:** the number of times each flow f occurs in pipeline i. This allows us to track how many appearances of each flow are covered if pipeline i is selected.
2. **Precondition coverage:** the set of pairs (p, v) such that both p and v appear in pipeline i, with p occurring before v. Each such pair contributes to satisfying the prerequisite requirement for vertex v.

This preprocessing maps each pipeline to the exact amount it contributes to covering quotas and prerequisites.

3.2 Greedy Reduction

Initially, we assume that all original pipelines are selected. We then attempt to remove pipelines one by one in order of descending cost (most expensive first).

A pipeline i can be safely removed if:

- For every flow f, the total coverage across the remaining pipelines is still at least w_f.
- For every precondition pair (p, v), there remains at least one pipeline that covers it.

If both conditions are satisfied, pipeline i is removed. Otherwise, it is kept. After the first pass, a second sweep is performed to detect any pipelines that have become removable after earlier deletions. The second sweep is a vestige from earlier versions, since coverage and slack only decrease, it cannot remove anything the first pass could not.

3.3 Outcome

The result of this phase is a reduced set of pipelines *U_sub* that guarantees feasibility of all flow multiplicities and prerequisite coverage, but at a lower cost compared to taking all the original pipelines. After extensive experimentation, more complex heuristics did not yield consistent gains, so we adopted the simplest approach that performed best overall.

4 New Pipeline Construction (U_new)

The second task of the problem requires constructing a new set of pipelines directly from the flows.

This is the main scoring part of the challenge, where the cost of the solution is evaluated.

Our approach relies on three ideas: (i) using a greedy subset of original pipelines to seed prerequisite coverage, (ii) translating remaining quotas into a graph where only flow endpoints matter, and (iii) completing this graph into an Eulerian multigraph with low-cost edges, so that Euler tours directly yield feasible pipelines.

4.1 Greedy Seeding of Prerequisites

Before building U_new, we compute a small subset U_sub of the original pipelines.

The purpose of U_sub is not cost minimization, but ensuring that some prerequisite pairs (p, v) are already satisfied.

Pipelines are considered in order of increasing cost, and we keep those that provide new prerequisite coverage.

Since prerequisites are relatively few in the datasets, this greedy strategy is sufficient to guarantee coverage while keeping the cost small.

4.2 Graph Model of Flow Requirements

The construction of U_new reduces to building a multigraph whose Euler tours represent pipelines.

The key intuition is that for each flow f, only its start vertex s_f and end vertex t_f matter: a pipeline containing f simply requires traversing the arc $(s_f \to t_f)$.

If flow f must appear w_f times in total, we insert w_f copies of the arc $(s_f \to t_f)$ into the graph.

To handle start and end location constraints, we add a sentinel node N, connected with separator arcs of identifier -1 so that pipelines can open or close correctly.

The resulting graph compactly represents all remaining requirements.

4.3 Cost-Guided Max-Flow Completion

The raw requirement graph may not be Eulerian: the in-degree and out-degree of vertices can differ.

To fix this, we add auxiliary arcs that rebalance degrees, turning the graph into one where an Eulerian decomposition exists.

This is achieved with a max-flow formulation:

- Vertices with deficit in in-degree are connected to a super-source, and those with surplus are connected to a super-sink.
- Candidate balancing arcs ($u \to v$) are generated from the set of flows, sorted by their cost.
- We add these arcs in three stages: first the cheapest third, then the medium-cost third, and finally the most expensive ones.
- After each stage, Dinic's algorithm is run with an ε-scaling reduced-cost filter, which biases augmenting paths toward low-cost arcs.

This heuristic ensures that cheap balancing arcs are used preferentially, while still guaranteeing feasibility.

By construction, the output graph is Eulerian: every vertex has balanced in/out degree, so it can be decomposed into trails.

4.4 Eulerization and Pipeline Extraction

Once the graph is Eulerian, we compute Euler tours starting from the sentinel node N.

Each Euler tour corresponds to a valid pipeline: traversing an arc ($s_f \to t_f$) means inserting flow f, and separator arcs with identifier -1 split the tour into distinct pipelines.

This guarantees that all flow quotas are met and every prerequisite pair is respected.

4.5 Handling Long Pipelines

If any constructed pipeline exceeds the allowed maximum length (1000 flows), we split it around a cut vertex v.

To ensure both halves remain valid, we connect the prefix of the pipeline to a valid end using a short backward path, and the suffix to a valid start using a short forward path.

These connectors are chosen by BFS over the flow graph, preferring cheapest arcs first.

4.6 Correctness Check and Late Prerequisites

Finally, we verify that all flow quotas and prerequisites are satisfied.

In rare cases, adding prerequisite-covering pipelines from U_sub can disturb Eulerian balance, making it impossible to decompose the graph directly.

If this happens, we postpone those prerequisites: first we construct an Eulerian solution covering the quotas, then we explicitly add the missing prerequisite pipelines afterwards. This guarantees correctness without breaking the Eulerian structure.

4.7 Outcome

The result of this phase, U_new, is a set of pipelines that achieves significantly lower total cost compared to relying only on the original pipelines.

It is worth noting that the prerequisite coverage problem is essentially NP-complete, so in principle much stronger solutions could be designed.

However, in practice this does not seem worthwhile, since the datasets contain relatively few prerequisites and a simple greedy strategy is sufficient.

On the flow balancing side, if we replaced our ε-scaled Dinic approach with a full Min-Cost Max-Flow algorithm, we would obtain an optimal completion of the requirement graph. Unfortunately, such an approach did not run within the time limits, which motivated the use of heuristics and staged cost-guided augmentations instead. This compromise ensures feasibility while keeping the runtime practical.

5 Final Comments

I truly enjoyed working on this problem. I felt it was highly motivating because it models situations that could be useful in real-world applications, which made the challenge especially interesting for me. I am proud of having reduced the core difficulty to finding an Eulerian cycle, and then approaching it through a Min-Cost Max-Flow formulation. This gave me the chance to apply both algorithmic knowledge and problem modeling skills in a meaningful way.

I also appreciated the opportunity to experiment with heuristics for improving Dinic's algorithm, even if not all attempts were successful. Although I would have liked to dedicate more time — the contest coincided with the chilean holidays, which took away a couple of working days — I still consider this a very rewarding experience.

Overall, it was a great challenge that pushed me to think creatively, and I am grateful for the opportunity to solve it.