

# Project Report - Entregas Caracol Lda

## 1. Solution Description

The problem consists of calculating the number of distinct paths between all pairs of intersections  $(A, B)$  in a one-way map (Directed Acyclic Graph – DAG) and assigning truck routes based on this value.

The solution, implemented in C++, is fully iterative and avoids exponential/sub-exponential approaches. The steps are:

### 1. Topological Sort (Kahn's Algorithm)

- Linearizes the graph to ensure that when visiting a node  $u$ , all predecessors have already been processed.
- Complexity:  $O(N + E)$ .

### 2. Batched Dynamic Programming

- Source nodes are processed in blocks of up to 128 vertices (BATCH\_SIZE).
- For each batch, a serialized array `batch_counts[u * BATCH_SIZE + k]` accumulates the number of paths between source  $(start\_base + k)$  and node  $u$ .
- A `batch_reachable` vector and a token mechanism avoid clearing the entire matrix between batches; only the used slices are re-initialized (Lazy Reset).
- Propagation follows the topological order, summing counts modulo  $M$  and marking destinations as reachable.

### 3. Truck Assignment and Output

- For each reachable pair  $(S, T)$ , we calculate  $truck(S, T) = 1 + (\text{paths}(S, T) \bmod M)$ .
- Routes for trucks in  $[m_1, m_2]$  are stored in vectors of pairs, later sorted lexicographically.

## High-Level Pseudocode

```
KahnTopoOrder(G):  
    compute in_degree  
    queue q = vertices with in_degree 0  
    topo = []  
    while q not empty:  
        u = pop(q); topo.append(u)  
        for v in adj[u]:  
            if --in_degree[v] == 0: push(q, v)  
    return topo
```

```
Solve(G):  
    topo = KahnTopoOrder(G)  
    for each batch B of sources in topo order:
```

```

init_batch_slices(B)
for u in topo starting at first element of B:
    if u not touched by B: continue
    record deliveries for reachable pairs (source in B, u)
    propagate counts from u to each v in adj[u]
sort and print routes per truck

```

## 2. Theoretical Analysis

### Time Complexity

Topological sorting is  $O(N + E)$ . Batched propagation executes, for each source, a traversal over the reachable subgraph: on average it is  $O(N + E)$  per batch, resulting in a global complexity of

$$O(N \times (N + E))$$

In the dense extreme ( $E \approx N^2$ ) this behaves as  $O(N^3)$ ; in sparse graphs it approaches  $O(N^2)$ . Empirical tests confirm that the solution is suitable for expected limits and avoids exponential/sub-exponential growth.

### Space Complexity

- **Graph (Adjacency List):**  $O(N + E)$ .
- **Auxiliary Structures (DP):** Vectors `batch_counts`, `batch_reachable`, `visit_token`, `topo_order` occupy  $O(N \times \text{BATCH\_SIZE})$ ; with `BATCH_SIZE = 128` this remains linear in practice.
- **Route Storage (Output):** In the worst case, all pairs  $(A, B)$  are valid. There are  $N(N - 1)/2$  possible pairs. Thus, the space to store output is  $O(N^2)$ .

Total Space Complexity:  $O(N^2 + E)$ .

## 3. Experimental Evaluation

### Experimental Results

Timings were measured on a MacBook Pro (macOS, Apple M-series CPU) running `./project` on DAGs generated by `python3 gen_test.py N` (standard density 0.3). Each instance is fed directly into the binary, and time is obtained with `time.perf_counter`. Full values are in `timings.csv`.

$N$	Time (s)	Ratio ( $T_N/T_{N/2}$ )
100	0.006	-
200	0.016	2.6x
400	0.055	3.4x

$N$	Time (s)	Ratio ( $T_N/T_{N/2}$ )
800	0.224	4.1x
1600	0.964	4.3x
3200	4.320	4.5x

**Analysis:** As  $N$  doubles, time grows between 3x–4.5x, reflecting the  $O(N \times (N + E))$  term. Generated graphs become dense as  $N$  grows, so  $E \propto N^2$  and behavior approaches  $O(N^3)$ , as predicted. Still, the solution easily processes instances up to  $N = 3200$ , comfortably meeting the time limit.

Automated tests include official examples, edge cases (minimal graph, disconnected components, multiple instances in same file, modular  $M$  scenarios), and larger random graphs, ensuring robustness and output correctness.