1. Task Description

The task is to solve the shortest path problem on a dynamic graph with directed but unweighted edges. Firstly the test harness sends the initial graph. The time spent on loading, pre-processing or indexing the initial graph will not count into the total execution time.

Then the workload comes in batches. Each batch consists of three types of operations:

1. \( A \ u \ v \) -- add an edge from vertex \( u \) to \( v \).
2. \( D \ u \ v \) -- delete the edge from \( u \) to \( v \), if it exists.
3. \( Q \ u \ v \) -- query the distance of the shortest path from \( u \) to \( v \).

Our goal is to answer these queries correctly, and as quickly as possible.

2. Solution Overview

We have tried to improve the performance from the following aspects:

- Reduce the overall search space for each query: Bidirectional-BFS.
- Reduce the number of basic operations per query: Bit Compression and Optimizing program’s spatial locality.
- Develop parallelism: Build Delta Graph to support fully concurrent query execution within a batch.

3. Bidirectional-BFS

**Bidirectional-BFS instead of naive BFS:** To reduce the search space of each query, we search from both forward direction and backward direction.

**Decision-making of exploration direction:** At each iteration, we select the direction of smaller sum of degrees to explore first.

4. Edge List’s Bit Compression

**Compression of adjacent edge list:** offset field + state field

- Reduce space cost (however we do not care). In best case, \( M' = M/8 \) (for 64-bit integer, \( M/64 \)).
- Reduce query’s execution time cost. Because we also maintain the visited vertex set in this compression format during Bidirectional-BFS.

5. Fully Concurrent Query Execution Mechanism

**Delta Graph:** When processing a batch, we maintain a Delta Graph over all the A/D operations. The Delta Graph preserves not only updated edges in this batch, but also each edge’s A/D time stamp list in order.

For example, if the edge \( e(v_0, v_1) \) is deleted at time \( t_2 \), and added back at time \( t_4 \), then its time stamps are \((t_2, D)\) and \((t_4, A)\). Furthermore, if an edge already exists in the version of graph before this batch, we add \((t_0, A)\) to the head of its time stamp list. Otherwise, we add \((t_0, D)\). Finally its A/D time stamp list is \([[(t_0, A), (t_2, D), (t_4, A)]\).

6. Optimizing Spatial Locality

- Rearrange graph’s storage in memory: Neighbor vertices in each vertex’s adjacent edge list are arranged continuously in physical address. Improvement on memory locality can reduce the cache miss rate.
- Reorder graph’s bandwidth: Graphs with lower graph bandwidth have a better locality. When the Bit Compression strategy does not work well with sparse graphs (e.g. roadmap graph), we reorder the initial graph by Reverse Cuthill-McKee Algorithm to reduce the graph’s bandwidth.
- Warmup Cache: Before begin processing workload batches, we execute a batch of randomly generated queries to warm up cache.

7. Third Party Libraries

- Jemalloc 3.6.0
- Intel Threading Building Blocks 4.3