Practical 1 Solution

Algorithms and Data Structures

Practical 1

Group groupnr
David van Munster (s1103638)

144

1 Explanation of the Algorithm

1.1 Initialization

The data is read using the standard input/output streams using the C++ standard library. Two graphs are used to store the stages and roads. Graph P contains both *pedestrian* and *both* roads, while graph B contains both *bus* and *both* roads. The graphs are represented as adjacency lists.

Algorithm 1 Initialization.

```
1:  function Initialize(Graph P, Graph B, stream Input)
2:      int S = Next(Input)
3:      int R = Next(Input)

4:      int s_0, s_1, T
5:      while input is available do
6:          s_0 = Next(Input)
7:          s_1 = Next(Input)
8:          T = Next(Input)

9:          if T == 0 then
10:             AddEdge(P, s_0, s_1, T)
11:         else if T == 1 then
12:             AddEdge(B, s_0, s_1, T)
13:         else if T == 2 then
14:             AddEdge(P, s_0, s_1, T)
15:             AddEdge(B, s_0, s_1, T)
```

1.2 Algorithm Logic

The algorithm maximizes roads to close by finding the Minimum Spanning Trees (MSTs) for both graphs and preserving only necessary roads. Roads of type *both* are prioritized since they serve dual purposes.

Two MSTs are generated using Kruskal's algorithm with a Union-Find data structure. Edge weights are assigned as 0 for *both* roads and 1 for *pedestrian* or *bus* roads, ensuring *both* roads are selected first. The MSTs guarantee connectivity with minimal edges for each mode.

The merged edge count sums all edges from MST_P and non-zero weight edges from MST_B, avoiding double-counting of *both* roads. The answer is $R - n$ where n is the merged count.

Algorithm 2 Main Algorithm

```
1: function ALGORITHM(int S, int R)
2:     // Initialize.
3:     P, B ← GRAPH(S)
4:     INITIALIZE(P, B)

5:     // Check connection.
6:     if P or B is disconnected then
7:         return -1

8:     // Generate MSTs.
9:     MST_P ← GENERATEMST(P)
10:    MST_B ← GENERATEMST(B)
11:    return R - GETMERGEDEDGECOUNT(MST_P, MST_B)
```

1.3 Returning the Answer

The algorithm terminates after generating both MSTs and computing the merged edge count. If any graph is disconnected, -1 is returned early. Otherwise, the number of roads to close is computed as $R - n$ where R is the total number of roads and n is the count of non-overlapping edges in both MSTs.

1.4 Optimizations

The algorithm terminates early in case of a disconnected graph, avoiding unnecessary computation of the MSTs.

Since each graph only contains two different types of weights, the sorting algorithm in the `generate_mst()` function can be simplified a lot. We can put each edge in one of two edge lists without having to do any comparisons. We can simply iterate through the first edge list with all the priority edges. This reduces the complexity from $O(R \log R)$ to $O(R)$.

2 Correctness Analysis

The correctness relies on the following properties:

MST: Kruskal's algorithm generates a MST by greedily selecting edges with a priority avoiding cycles. With weights 0 for *both* and 1 for specific roads, the algorithm minimizes the number of roads while keeping the stages connected.

Connectivity: An MST contains $S - 1$ edges and connects all vertices. If graph P or B is disconnected, no valid solution exists, correctly returning -1. When both are connected, their MSTs guarantee minimal connectivity for each mode.

Edges: Edges of type *both* appear in both MST_P and MST_B with weight 0 in the MST representation. The merged count adds all edges from MST_P and only non-zero edges from MST_B, correctly accounting for shared roads exactly once. The remaining $R-n$ roads can be closed without affecting connectivity.

3 Complexity Analysis

Let S be the number of stages and R be the number of roads.

Initialization: Reading input and constructing adjacency lists takes $O(R)$ time, as each road is added to at most two graphs.

Connectivity Check: Depth-first search on each graph visits all vertices and edges, requiring $O(S + R)$ time per graph, thus $O(S + R)$ total.

MST Generation: For each graph, sorting edges takes $O(R)$ time. Processing edges with Union-Find operations takes $O(R \log S)$ time, as each find and union operation has $O(\log S)$ complexity. Therefore, MST generation takes $O(R \log S)$ per graph.

Merged Count: Iterating over MST edges takes $O(S)$ since each MST has at most $S-1$ edges.

The overall complexity is $O(S + R + R \log S) = O(R \log S)$ when R is significant, dominated by the Union-Find operations in MST generation.

4 Reflection

Finding a suitable algorithm was in hindsight very doable, however during experimentation you get to experience the advantages and disadvantages of the different data structures, representations, etc. which were quite overwhelming to apply practically.

It was also needed to look at the problem with a different perspective. The initial idea of removing unneeded edges did not click until it was reformulated as 'finding the smallest trees'.