CS 170 Project Reflection

Team: nAiVe_aLGoRiThM

Members: Sunay Dagli, John Lee, Wilson Nguyen

Algorithm

Our algorithm involved two different algorithms for small and medium/large inputs.

Small Input Algorithm:
For inputs with relatively small amounts of vertices and edges, using a naive approach was feasible in terms of output time.

This approach involved first traversing through all nodes that were in the current Djikstra's shortest path, removing one, running Djikstra's shortest path and comparing the output to the minimum, and removing the one that led to the minimum decrease in shortest path. The rationale behind doing nodes first was because we thought this was most efficient in removing both nodes and vertices, since removing nodes in turn removes vertices without using the vertice budget.

After removing nodes, we had two versions of the naive algorithm that had a similar structure to that of the node removal. It first checks the Djikstra's shortest path, removes the edges that were present in that initial shortest path and checks the Djikstra shortest path weight, and removes the edge that led to the minimum decrease in shortest path weight. However, we noticed in certain edge cases that sometimes the edge to be removed when k > 1 was not in the shortest path. As a result, we created two copies of the algorithm, the only difference was that the edges to be removed were all the edges, not just those in the shortest path. Then the algorithm returns the c and the k values from the max score from the results from both of these algorithms.

This was a good approach because it was the most naive approach, which led to us understanding the problem deeper for more intricate solutions. This did lead to above average results on the leaderboard. Since the inputs were smaller, the time to output the results was manageable, but it was extremely time consuming for medium/large inputs as it ran for O(E^k logV).

Medium/Large Input Algorithm:
For medium and large inputs, our naive method took around 4 minutes to run for each large input and 1 minute for each medium input, which would total to 5 min/input * 300 inputs = 1500 min = 25 hours of computation. We decided that this was infeasible and thus we came up with a greedy approximation algorithm.

The inspiration for this algorithm came from manually trying the problem on very small inputs (V = 8, E = 12). While we were doing so, we incremented `k` by [1, 2, … , k] in each iteration to see how we can design general heuristics in which we decide how to take out certain edges for large inputs. We realized that the edge that we are taking out in each step always includes at least one edge from the current shortest path.

Thus, for this approach, we first took out `c` vertices that have the most impact on maximizing the shortest path, then ran Dijkstra's algorithm to find the shortest path from `s` to `t` to determine which edge within the shortest path should be taken out. For each iteration, we calculated the heuristic value as the increase between the edge weight of a certain edge and the shortest cycle from the two end points of the edge. This heuristic determines how removing the given edge will increase the shortest path given that the new shortest path follows all edges from its old shortest path and instead goes for the shortest cycle to account for the removed edge.

Such an approach is greedy as we look at one edge per iteration and not multiple edges which overall may maximize the shortest path. However, it proved to be a good approximation as we were able to place in the top 25% of all outputs within a very fast runtime of O(k E^2 logV), which took less than 10 seconds to compute per input on our machines (detailed below).

Attempted Approaches

Reverse Algorithm

The first approach that we came up with was to solve the problem in reverse. We first find the longest path from `s` to `t` (without cycles), since this would be the optimal result for the maximum shortest path. To fit the constraints for `k` and `c`, we then add a certain amount of edges and vertices to our maximum shortest path in a manner so that the only path connecting `s` to `t` would be the longest path. However, in the worst case where `k` is smaller than the number of min-cut edges, we add edges that minimally reduce the length of our longest path until the `k` constraint is satisfied.

However this approach simply failed because finding the longest path (without cycles) is a NP-Hard problem itself, which is reduced from the Hamiltonian Path problem. Solving an NP-Hard problem to solve another NP-Hard problem was not the best idea.

Resources Used

The only computational resource we used was our local machines (detailed below) for this project. The runtime of our algorithms were relatively fast, and thus we were able to generate all outputs within 40 minutes collaboratively - with all three machines doing 1/3 of the outputs, and

90 minutes individually (on machine 1). The difference between collaborative times and individual time is due to the processing speed of our different computers.

The specs of our machines are as follows:

Machine 1)

Processor: 2.6 GHz 6-Core Intel Core i7
Memory: 16GB 2400 MHz DDR4

Machine 2)

Processor: 3.1 GHz 4-Core Intel Core i7
Memory: 16GB 2400 MHz DDR4

Machine 3)

Processor: 3.1 GHz 4-Core Intel Core i7
Memory: 16GB 2400 MHz DDR4