OS Assignment-3

Report

Group-2

Sarthak Nikumbh Sham (20CS30035)
Ritwik Ranjan Mallik (20CS10049)
Saptarshi De Chaudhury (20CS10080)
Ayush Kumar Dwivedi (20CS10084)

Optimization strategy for re-computing shortest paths

We have used two separate functions: ‘dijkstra’ and ‘optimised_dijkstra’ in “consumer.cpp” for computing the shortest paths in the case of unoptimized and optimized versions respectively.

In unoptimized case, we use simple multi-source dijkstra. Thus, even after updation of the graph, each consumer process again considers all sources (the previous ones as well as any new source added) and runs multi-source dijkstra for the whole graph from these sources. The time complexity of each consumer process every time it runs is O(|E|*log|V|), where |E| is the number of edges and |V| is the number of vertices in the graph after updation.

In optimized case, we do 2 optimizations: firstly, we re-use the already calculated shortest paths from the previous computations done by the consumer wherever possible. Secondly, we use a queue instead of a priority queue (multi-source bfs instead of multi-source dijkstra). Below is the explaination for the optimization:

If the consumer is running for the first time, just push all the sources in the queue and run a simple multi-source bfs which runs in O(|V| + |E|) time complexity, where |E| is the number of edges and |V| is the number of vertices in the graph.

Now, when the consumer has already run once, we check if new nodes have been added to the graph or not. If not added, we just exit from the consumer process as the output files do not need to be changed.

If the graph has been updated since last shortest paths computation, we run bfs from every newly added vertex in a modified way such that if we are running the bfs from say, node i, then we simply ignore any node in the graph having index > i.

The above logic is important as it helps making it appear that nodes are being added to the original graph one-by-one.

So now, our process is simple: For each newly added node, do bfs from each node one-by-one (say current index i) to all nodes in the graph having index <= i. This gives us the shortest path from this vertex to all nodes in the graph with index <= i. Now, find the source which is nearest to this vertex, say it is source s.

Now we prove that any node x in the graph with index < i, will have only two possibilities for its shortest path to any source: either the original path which was calculated previously, or the path from x to i (found from bfs from node i) concatenated with path from i to s.

Proof: First, we observe that any node whose path has been updated in this process, will always have node i in its path. As otherwise, it will imply that the shortest path to the node

has been re-computed using the same nodes as before and this path is smaller than the one calculated previously by the consumer, but it was itself the shortest path, hence contradiction.
 It is trivial to say that shortest path from x to any source will either have i in its path or not have it. If it does not have i in its path, from the above observation it means that it should remain the same as previous shortest path already calculated.
Now if the shortest path from x contains node i, then its length should always be equal to the length of the path: x->i (found from bfs from node i) concatenated with i->s. Because, path from i->s is the shortest path from node i to any source node and path from x->i is also the shortest path from x to i as it has been found from bfs from i. Therefore, the above path becomes the shortest path.

Therefore, we simply compare the path lengths of all nodes from 0 to i and update the path to the shorter one.

The above process runs in O(m*(|V| + |E|) time and since m is a small constant, we say our optimized algorithm runs in O(|V| + |E|) time.