14 Single-Source and All-Pair Shortest Path Problem

Single-Source Shortest Path. We discuss the Bellman-Ford algorithm for single-source shortest path problem in graphs with negative edges.

Suppose we are given a graph $G = (V, E, w)$ with n nodes, m edges and edge weights w might be negative. Suppose the given source node is s. Let us also suppose the graph does not have negative weighted cycles.

We can view the Bellman-Ford algorithm as a dynamic programming algorithm as follows. Define $OPT[t, j]$ to be the shortest path distance from s to j using at most t edges. The following claim shows that $OPT[n-1, j]$ will be the real shortest path distance from s to j if there is no negative cycle.

Claim 7 *If the input graph G does not have negative cycle, then there exists a shortest path from s to j using at most $(n-1)$ edges.*

Proof: Suppose for contraction that a shortest path with the least number of edges has to use at least n edges. Then this path has to visit at least $n+1$ nodes, i.e., has to visit some node twice. Let's call this node i. Then the edges in the path between the first and second visits to node i is a cycle. If this cycle has weight ≥ 0, then by skipping this cycle, the path has no less total weight and uses strictly less number of edges, contradicting to our assumption that our path is the shortest path with the least number of edges. If this cycle has weight < 0, then this contradicts to the assumption that the graph has no negative weighted cycles. □

As boundary condition, we set $OPT[0, j] = 0$ for $j = s$, and $OPT[0, j] = +\infty$ for j not equal to s. We can also easily write the following recursion for $t > 0$.

$$OPT[t, j] = \min_{i:(i,j)\in E} \{OPT([t-1, i] + w_{ij}, OPT[t-1, j]\}$$

This dynamic programming takes time $O(nm)$, and space $O(m)$ (indeed, we only need to keep $OPT[t-1, \cdot]$ and $OPT[t, \cdot]$ when computing $OPT[t, \cdot]$).

A final note is that we can also detect whether we can reach a negative weighted cycle from s by computing $OPT[n, \cdot]$. If $OPT[n, \cdot]$ becomes even better than $OPT[n-1, \cdot]$, then we know that there must be a negative cycle. (And if there is a negative cycle reachable from s, $OPT[n, \cdot]$ must become better than $OPT[n-1]$ since we can loop in the negative cycle).

All-Pair Shortest Path. In the APSP problem, we are given a directed graph $G = (V, E, w)$ where w is the edge weights. The goal is to compute the shortest path distance for all pairs of nodes in V.

One straightforward solution for the APSP problem is to run the algorithms for Single Source Shortest Path (SSSP) problem n times, each time for a different source in V. If we do this with Dijkstra's algorithm, it takes $O(n^2 logn + nm)$ time. If we do this with Bellman-Ford algorithm, the time complexity is $O(n^2 m)$; which is worse than Dijkstra's algorithm, however can tolerate negative weighted edges.

Now let us try to improve the n-time Bellman-Ford algorithm, still using Dynamic Programming techniques.

Let us define $OPT[z][i][j]$ to be the shortest path distance from i and j using at most 2^z edges. Note that when $2^z \geq n-1$, $OPT[z][i][j]$ becomes the final solution (i.e. the shortest path distances with edge length no restrictions). This is because from the previous discussion, we know the existence of shortest paths with at most $(n-1)$ edges for graphs with no negative weighted cycles.

As boundary conditions, when $z = 0$, this means we have to reach from i to j by a single edge (or with no edge). Therefore, $OPT[0][i][i] = 0$, $OPT[0][i][j] = w_{i,j}$ if there is an edge (i, j), and $OPT[0][i][j] = \infty$ if there is no edge (i, j).

Now, let us consider how to compute $OPT[z][i][j]$ for $z \geq 1$. Consider any path for i to j using at most 2^z edges. There must be a node in this path, namely k, so that it takes at most 2^{z-1} edges to reach k from i (along this path), and it takes at most 2^{z-1} edges to reach j from k (along the path). Therefore, the first segment of the path becomes the subproblem $OPT[z-1][i][k]$, and the second segment of the path becomes the subproblem $OPT[z-1][k][j]$. In total, we would like to find a k so that $OPT[z-1][i][k] + OPT[z-1][k][j]$ is minimized. Therefore, we come up with the following recursion

$$OPT[z][i][j] = \min_k \{OPT[z-1][i][k] + OPT[z-1][k][j]\}.$$

Note that there are $O(n^2 \log n)$ subproblems, and it takes $O(n)$ time to solve each subproblem. Therefore, the time complexity of the whole algorithm is $O(n^3 \log n)$.

The second algorithm we introduce is the Floyd-Warshall algorithm. The algorithm is very similar in its pseudocode as the one above. However, the underlying idea is very different. Let us assume all nodes have distinct labels in $\{1, 2, \ldots, n\}$. Now for each path from i to j, let us define the max label of the path to be the maximum label of the intermediate nodes in the path, i.e. not counting the two endpoints i and j. For example, the max label of the path $3 \to 5 \to 4 \to 1 \to 6$ is 5.

Now let us define subproblems according to max label of paths, as follows. Let $OPT[k][i][j]$ to be the shortest path distance from i to j, among all paths with max label at most k. Since all paths have max label at most n, $OPT[n][i][j]$ will be the final solution.

As boundary conditions, $k = 0$ means that no intermediate nodes are allowed. Therefore $OPT[0][i][i] = 0, OPT[0][i][j] = w_{ij}$ if there is an edge (i, j), and $OPT[0][i][j] = \infty$ if there is no edge (i, j).

Now let us consider how to compute $OPT[k][i][j]$ for $k > 0$. Consider a path from i to j with max label at most k. There are two possibilities.

1. The max label of the path is strictly less than k. Then the max label of the path is at most $k-1$. The shortest path distance among all paths those belong to this case should be $OPT[k-1][i][j]$.

2. The max label of the path equals to k. We first note that k cannot appear more than once in the path, as that would introduce a loop. The weight of the loop is nonnegative as we do not allow negative weighted cycles. Therefore we can always skip the loop to decrease the number of occurrences of k in the path, while the path distance does not increase. Therefore,

let us identify the unique occurrence of k in the path, and divide the path into 2 segments. The first one is from i to k, and the second one is from k to j. It is easy to see that both segments have max label at most $k-1$ since there is no more k. Therefore, the first segment reduces to the subproblem $OPT[k-1][i][k]$, and the second segment reduces to the subproblem $OPT[k-1][k][j]$. The shortest path distance among all paths those belong to this case is $OPT[k-1][i][k] + OPT[k-1][k][j]$.

In total, we get the recursion as follows.

$$OPT[k][i][j] = \min\{OPT[k-1][i][j], OPT[k-1][i][k] + OPT[k-1][k][j]\}$$

The runtime of the algorithm is $O(n^3)$.