COL 351

TUTORIAL SHEET 8

1. **[KT-Chapter6]** Suppose you are given a directed graph $G = (V, E)$ with length l_e on edges (which could be negative, but no negative cycles), and a sink vertex t. Assume you are also given finite values $d(v)$ for all the vertices $v \in V$. Someone claims that for each node $v \in V$, the quantity $d(v)$ is the cost of the minimum-cost path from node v to t. (i) Give a linear time algorithm which verifies whether the claim is correct, (ii) Assuming that all the distances $d(v)$ are correct, and that all $d(v)$ values are finite, you now need to compute distances to a different sink vertex t'. Give an $O(m \log n)$ time algorithm for computing these distances $d'(v)$ for all the vertices $v \in V$.

   Solution: (i) First of all, we must have $d(v) \leq d(w) + l(v, w)$ for every edge (v, w) — indeed, this says that one way of going from v to t is first go to w and then go to t. Assume this condition holds for every edge e. The first observation is that $d(v)$ is at most the length of shortest path from v to t. We can show this as follows: consider the shortest path P from v to t and then add up the above inequality for all edges in this path.

   Now, consider the edges which lie on a shortest path from any of the vertices to t. On such an edge e, if the values $d()$ are indeed correct, then we must have $d(v) = d(w) + l(v, w)$)(why ?). So, we consider all edges for which equality holds – such edges must form a connected graph. Now show that if P is a path in this connected graph from v to t, then $d(v)$ is equal to the length of this path (again, by adding up the equations for every edge). And so, from the previous paragraph, it follows that $d(v)$ is equal to the length of the shortest path from v to t.

   (ii) We would like to run Dijkstra because Dijkstra takes $O(m \log n)$ time. But, we need all edge lengths to be non-negative. For this, we define a new length of edge $e = (v, w)$ as $l'_e = l_e + d(w) - d(v)$. As noted above, $l'_e \geq 0$. Also, argue that for any vertex v, a shortest path with respect to l_e is also a shortest path with respect to l'_e and vice versa.

2. **[KT-Chapter6]** Suppose we are given a directed graph $G = (V, E)$, with costs on edges – the costs may be positive or negative, but every cycle in the graph has positive cost. We are also given two nodes v and w in the graph G. Give an efficient algorithm to compute the number of shortest $v - w$ paths in G (the algorithm should NOT list the paths; it should just output the number of such paths).

   Solution: We modify Bellman Ford algorithm. We build a table $S[i, u]$, which stores the length of the shortest path from u to w which uses at most i edges. Further, we have a table $T[i, u]$ which stores the number of paths using i edges from u to w whose cost is $S[i, u]$ (i.e., the shortest path using i edges). Now, suppose the out-neighbours

of a vertex u are $v_1, \ldots, v_k$. Then,

$$S[i+1, u] = \min_{r=1}^{k}(l_{(u,v_r)} + S[i, v_r]).$$

Now, let $v_{s_1}, \ldots, v_{s_l}$ be the neighbours of u which achieve the minimum above, i.e., for which $S[i+1, u] = l_{(u,v_r)} + S[i, v_r]$. Then, we update

$$T[i+1, u] = T[i, v_{s_1}] + \ldots + T[i, v_{s_l}].$$

Show how to initialize the tables.

3. You are given a directed graph G where all edge lengths are positive except for one edge. Given a source vertex s, give $O(m \log n)$ time algorithm for finding a shortest path from a vertex s to a vertex t. Now assume there are a constant number of edges in G which have negative weights (rest have positive weights). Give an $O(m \log n)$ time algorithm to find a shortest path from s to t. **Solution:** Let R denote the edges $e_1, \ldots, e_k$ with negative length, and let $e_i = (u_i, v_i)$. Let H be the graph obtained by removing the edges in R. Let $D[v]$ denote the shortest path from s to v in H. Also, let $D_i[v]$ denote the shortest path from v_i to v in H. Computing these take $O(km \log n)$ time, which is $O(m \log n)$ since k is a constant.

   Now suppose P is a shortest path from s to t in G and say it contains the edges $e_{i_1}, e_{i_2}, \ldots, e_{i_r}$ from E' in this order as go from s to t. Then it is easy to check that the length of the path P is equal to

$$D[u_{i_1}] + c_{e_{i_1}} + D_{i_1}[u_{i_2}] + l_{e_{i_2}} + D_{i_2}[u_{i_3}] + \cdots + D_{i_r}[t].$$

   Thus, we if knew the edges $e_{i_1}, e_{i_2}, \ldots, e_{i_r}$, then we can find the length of P in $O(1)$ time. So, we just try all such choices of $e_{i_1}, \ldots, e_{i_r}$ – there are at most $2^k \cdot k!$ such choices (there are 2^k subsets of edges in E' and then we try all orderings of this subset), which is a constant. So, in $O(m \log n)$ time, we can find the length of the path P.

4. Describe an efficient algorithm to find the second minimum shortest path between vertices u and v in a weighted graph without negative weights. The second minimum weight path must differ from the shortest path by at least one edge and may have the same weight as the shortest path. **Solution:** We first run Dijkstra and computer shortest path from s to every vertex v – store the shortest path length in an array $D[v]$. Let $\mathsf{pred}(v)$ be the predecessor of v (on the shortest path from s to v) – again, this can be computed while running Dijkstra's algorithm. Let $s = v_1, v_2, \ldots, v_n$ be the order in which the vertices are visited by Dijkstra's algorithm (so $\mathsf{pred}(v)$ appears before v in this ordering).

   We will compute the second shortest path length $D'[v]$ for all v in this order as well. $D'[s]$ is ∞. Now suppose we have calculated $D'[v]$ for $v_1, \ldots, v_{i-1}$ and now want to compute $D'[v_i]$. Now two cases arise: (i) the predecessor of v_i in the second shortest path remains $\mathsf{pred}(v_i)$, or (ii) it is some other vertex $w \neq \mathsf{pred}(v_i)$.

   In the first case, $D'[v_i]$ is equal to $D'[u] + w_{(u,v)}$, where u denotes $\mathsf{pred}(v_i)$ (observe that we have already computed $D'[u]$). In the second case, $D'[v_i] = D[w] + l_{(w,v)}$.

Thus, $D'[v_i]$ is set to the minimum of these two quantities. The running time of this algorithm: first we run Dijktra's algorithm, and after that computing $D'[v_i]$ takes $O(degree(v_i))$ time. Therefore, the overall running time is dominated by the the time to run Dijkstra's algorithm.

5. Given a directed acyclic graph that has maximal path length k, design an efficient algorithm that partitions the vertices into $k+1$ sets such that there is no path between any pair of vertices in a set. **Solution:** Let $D[v]$ denote the maximum path length of a path ending at v. We can compute this as follows: let $v_1, \ldots, v_n$ be a topological sort ordering of the vertices. Clearly, $D[v_1] = 0$ (in fact, initialize it to 0 for all vertices). Now, we will compute this quantity in this order. When we come to v_i, the longest path ending at v_i must have the second last vertex as some v_j such that $(v_{,j}\,, v_i)$ is an edge. It follows that $j < i$, and so, we already know $D[v_j]$. Therefore, we can set

$$D[v_i] = \max_{v_j:(v_j,v_i)\in E} D[v_j] + w_{(v_j,v_i)}.$$

It is easy to show by induction that this computes the length (i.e., number of edges) in the longest path ending at each vertex. So $D[v]$ lies in the range $\{0, 1, \ldots, k\}$. Also the time to compute $D[v_i]$ is proportional to the degree of v_i, and so, the overall time take to compute $D[v]$ for all the vertices is $O(n + m)$.

Let S_i be the set of vertices v for which $D[v] = i$. We claim that there cannot be a path between two vertices belonging to the same set S_i. Suppose not. Suppose $u, v \in S_i$ for some i and there is a path P from u to v. Composing this path with a path of length i ending at u (because $u \in S_i$), we get a walk of length $i + |P| > i$ ending at v. Since G does not have a cycle, each walk must be a path, and so there is a path of length larger than i ending at v. But this contradicts the fact that $D[v] = i$. Thus, the $k + 1$ desired sets are $S_0, S_1, \ldots, S_k$.