CSCE 411-502 Design and Analysis of Algorithms

Spring 2025

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MW 1:30 pm–3:00 pm

Senior Grader: William Kang
Phone: 979) 575-9987
Email: rkdvlfah1018@tamu.edu
Questions: via phone and email
and by appointments

Assignment #4 Solution

1. Another way to do topological sorting on directed acyclic graphs is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges. Develop an $O(n + m)$-time algorithm using this approach. Your algorithm should also be able to tell when the input graph has cycles.

Solution. First note that if every vertex has in-degree larger than 0, then there must be cycles so that topological sorting is impossible. To see this, start from an arbitrary vertex v and traverse the graph in the reversed directions of the edges. This traverse cannot be stopped because at each vertex of in-degree larger than 0, we can follow the reversed direction of an edge going into that vertex and go to another vertex. Therefore, this traversing must have repeated vertices. the part of the traverse that starts and ends at the same vertex will make a cycle.

Thus, we can topologically sort the input graph by repeatedly picking a vertex v of in-degree 0, removing v (and all edges going out from v) from the graph, and placing v in the output array $T[1..n]$ that is supposed to be topologically sorted. This makes sure that no edges going into v can go from right to left in the array T because the in-degree of v is 0 when v is considered (note the the vertex v may have in-degree larger than 0 in the original graph). This process will stop either because all vertices of the input graph are placed in the array T, so we get the graph topologically sorted, or because we could not find any vertex of in-degree 0, which, by the discussion in the previousl paragraph, imples that there are cycles in the graph so that topological sortig is impossible.

To implement this by an algorithm of time $O(n + m)$, we need to dynamically record the in-degree of each vertex and find vertices of in-degree 0 efficiently. We will use an array D[1..n] to record the in-degree of vertices so that D[v] is the in-degree of the vertex v. We will use a queue Q to collect vertices of in-degree 0. Note that we can add an element, delete an element, and test the emptyness of the queue Q in time $O(1)$.

The algorithm is given below.

```
Topological Sorting(G)
\\ D[1..n] stores the in-degree for each node
1. for (v = 1; v ≤ n; i++) D[v] = 0;
2. for (each arc[v, w]) D[w] = D[w] + 1;
\\ find vertexes of in-degree 0 and store them in the queue Q
3. Q = emptyset;
   for (each vertex v)
      (if D[v] == 0) EnQueue(Q, v);
\\ repeatedly place in-degree 0 vertices in the output T[1..n]
4. for (i = 1; i ≤ n; i++)
4.1   if (Q == emptyset) return("cycles, no topological sorting");
4.2   v = DeQueue(Q);
4.3   T[i] = v;
4.4   for (each arc [v, w])
         D[w] = D[w] - 1;
         if (D[w] == 0) EnQueue(Q, w);
\\ topological sorting is successful
5. return the array T[1..n].
```

The complexity analysis of the algorithm is relatively simple. Note that for the queue Q, each of the operations EnQueue, DeQueue, and testing if Q is empty takes time $O(1)$. Thus, step 1 take time $O(n)$, step 2 takes time $O(m)$, and step 3 takes time $O(n)$. For step 4, the analysis goes similar to that for DFS: the body of the inner for-loop at step 4.4 takes time $O(1)$ and is executed in total m times because the for-loop at step 4.4 examines each arc exactly once. Therefore, the total time taken by the inner for-loop of step 4.4 in the entire execution of the algorithm is $O(m)$. Without counting step 4.4, step 4 takes time $O(n)$ because each of the steps 4.1-4.3 takes time $O(1)$. As a result, step 4 in total takes time $O(n + m)$. Summarizing all these, we conclude that the algorithm takes time $O(n + m)$.

2. Develop a linear-time (i.e., *O(m + n)*-time) algorithm that solves the Single-Source Shortest Path problem for graphs whose edge weights are positive integers bounded by 5. (**Hint**. You can either modify Dijstras algorithm or consider using Breath-First-Search.)

Solution. To implement the idea of converting the given problem to BFS, for the given weighted graph G, we construct an unweighted graph U by replacing each edge e of weight $wt(e)$ in G with a simple path p_e of $wt(e)$ edges in U. Thus, to pass the edge e of weight $wt(e)$ in G, we pass the $wt(e)$ edges of the corresponding path p_e in U. It is easy to see that for any two vertices s and t in G, a path P_G from s to t in the weighted graph G is the shortest if and only if the corresponding path P_U from s to t in the unweighted graph U is the shortest (i.e., the one using the fewest edges), where P_U in the unweighted graph U is the path P_G in the weighted graph G with the edges in P_G being replaced by the corresponding paths in U. Therefore, we can apply BFS on the unweighted graph U, starting from the vertex s, which will

give a shortest path in U from s to each of the other vertices in U (see Homework #3, Question 1). From this, we can easily find a shortest path from s to each of the other vertices in G in the graph G, thus solve the single-source shortest path problem for the weighted graph G.

SSSP(G, s)
1. for (each edge e = [v, w] of weight wt(v, w) in G)
   create a path of wt(v,w) edges from v to w in U;
2. apply BFS on the graph U, starting from s to construct the array dad_U[1..N] that represents the BFS-tree T in U;
3. for (each vertex v in U that is also a vertex in G)
   if (dad_U[v] == Nil) dad_G[v] = Nil;
   else t = dad_U(v);
   while (t is not in G) t = dad_U[t];
   dad_G[v] = t;
4. return the array dad_G[1..n].

Step 3 above is to construct the array $\text{dad}_G[1..n]$ that represents the shortest path tree for the weighted graph G based on the array $\text{dad}_U[1..N]$ constructed by BFS in step 2 that represents the shortest path tree for the unweighted graph U, where for each vertex v in U that is also in G, we trace the array $\text{dad}_U[1..N]$ to find the nearest ancestor w of v that is in G, from the construction of the graph U, we know that $[w, v]$ is an edge in G. By setting $\text{dad}_G[v] = w$, we add the edge $[w, v]$ in the shortest path tree for the graph G.

As discussed above, this algorithm solves the single source shortest path problem correctly. To analyze the complexity of the algorithm, note that by the given condition, the weight $wt(v, w)$ of an edge $[v, w]$ in G is a positive integer not larger than 5. Thus, each edge in the graph G is replaced by a path of at most 5 edges (and at most 4 new vertices) in the graph U. If the graph G has n vertices and m edges, then the graph U has $N \leq n + 4m$ vertices and $M \leq 5m$ edges. Therefore, the BFS on the unweighted graph U in step 2 of the algorithm SSSP(G, s) takes time $O(N + M) = O((n + 4m) + 5m) = O(n + m)$. All other steps of the algorithn SSSP(G, s) obviously take time $O(n+m)$. In conclusion, the algorithm SSSP solves the single source shortest path problem in time $O(n + m)$.

We also give a brief description on how the given problem is solved by a modified Dijkstra's algorithm. The key observation is that because of the condition that edge weights are positive integers not larger than 5, we can prove that at any moment, the dist-values of the fringes must be among 5 consecutive integers. Therefore, we can keep all the fringes in 5 sets, in terms of their dist-values. Based on this representation, the operations of finding a fringe of the minimum dist-value, adding a new fringe to the fringe set, and changing the dist-value of a fringe all can be done in time $O(1)$. Now if you look into Dijkstra's algorithm with the above operations in time $O(1)$, you can easily derive that Dijkstra's algorithm take time $O(n + m)$.

3. Let G be a weighted graph. Let P be a path in G. The *bandwidth* of the path P is defined to be the minimum edge weight over all edges on the path P. Develop an algorithm that solves the following problem: given a weighted graph G and two vertices s and t, find a path from s to t whose bandwidth is the largest over all paths from s to t in G. (**Hint**. Consider the approach of Dijkstra.)

Solution. We solve the problem based on the approach of Dijkstra. Thus, we start from the source vertex s, and grow a tree by repeatedly adding to next best fringe to the tree. However, to measure the quality of a path, we use the bandwidth of the path, instead of the length of the path. Note that when we extend a path P by attaching a new edge e to one of the ends of P, the bandwidth of the new path is equal to the minimum of the bandwidth of P and the weight of the edge e.

```
MaxBandwidth(G, s, t)
\\ construct a maximum bandwidth path from s to t in the graph G
1. for(v = 1; v ≤ n; i ++)
       status[v] = unseen;  wt[v] = 0;  dad[v] = Nil;
2. status[s] = in-tree;
3. for (each edge [s,w])
       status[w] = fringe;  wt[w] = weight(s,w);  dad[w] = s;
4. while (there are fringes)
4.1    pick the fringe v with the largest wt-value;  status[v] = in-tree;
4.2    for (each edge [v,w])
           if (status[w] == unseen)
               status[w] = fringe;  dad[w] = v;
               wt[w] = min{wt[v], weight[v,w]};
           else if (status[w] == fringe) & (wt[w] < min{(wt[v], weight[v,w]})
               dad[w] = v;  wt[w] = min{(wt[v], weight[v,w]}
5. return the arrays dad[1..n] and wt[1..n].
```

As we discussed in class, a straightforward implementation of the above algorithm will give an $O(n^2)$-time algorithm. If we use a heap H for the fringes (H should be a max-heap so that step 4.1 for finding a fringe of the largest wt-value can be done in time $O(\log n)$), then the algorithm runs in time $O(m \log n)$.

The proof for the correctness of the algorithm is very similar to that for the original Dijkstra's algorithm for the shortest path problem. Thus, we give a brief description for the proof. We prove by induction on the number of in-tree vertices that when a vertex v becomes in-tree, then the path in the tree from s to v is a maximum bandwidth path from s to v. Now suppose that for the latest in-tree vertex v, the claim is not true, and that there is a path P'_v from s to v whose bandwidth is larger than that of the path P_v in the tree from s to v. We traverse the path P'_v, starting from s until we encounter the first fringe w. Then the partial path of P'_v from s to w would have a bandwidth larger than that of P_v, but this would lead to the conclusion that we picked v as the next in-tree vertex but $wt[v]$ is smaller than $wt[w]$ for the fringe w, contradicting step 4.1.

Students are encouraged to write a more complete proof for the correctness of the above algorithm.