CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MWF 3:50 pm - 5:00 pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: vaibhavbajaj@tamu.edu
Office Hours: T,TR 2:00 pm - 3:00 pm

Solutions to Assignment #3

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

Solutions. Denote by $[u, v]$ the directed edge from vertex u to vertex v. Let Q be a First-In-First-Out queue that keeps all the vertices with degree 0. The output of the algorithm is given in an array A, in the topologically sorted order.

```
TopSort2(G) \\ G is a directed graph.
1.  for (each vertex u) indeg[u] = 0;
2.  for (each vertex u)
      for (each edge [u,v]) indeg[u]++;
3.  for (each vertex u)
      if (indeg[u] = 0) Q ← u;
4.  p = −1;
5.  while (Q is not empty)
5.1   u ← Q;  p++;  A[p] = u;
5.2   for (each edge [u,v])
5.2.1    indeg[v]--;
5.2.2    if (indeg[v] == 0) Q ← v;
6.  if (p < n − 1) return('G has cycles');
6.1.  else return(A[0..n − 1]).
```

The array A maintains a partial topological order as vertices of in-dgree 0 are added in. Each vertex u is pushed into the queue Q at most once, either in step 3 if its indegree is 0 in the original graph G, or in step 5.2.2 when its indegree becomes 0 from 1. Therefore, the `while`-loop in step 5 takes time $O(|V| + |E|)$. It is easy to verify that all other steps of the algorithm take time $O(|V| + |E|)$. Therefore, the time complexity of the algorithm is $O(|V| + |E|)$.

The correctness of the algorithm is easy to see. When we place a vertex u in the array $A[p]$, the indegree of u is 0, which means that all coming-in edges to u are from array elements $A[q]$ with $q < p$. Thus, if the algorithm returns at step 6.1, where all vertices of G are placed in the array A, then there will be no edge in G that goes from $A[q]$ to $A[p]$ with $p < q$, i.e., the array A gives a topologically sorted order for the vertices of the graph G.

On the other hand, if the algorithm returns at step 6, then the graph G has a subgraph G' in which no vertices have indegree 0. Then the subgraph G' must have cycles. This can be seen as follows. Start from any vertex in the subgraph G' and traverse G' following the reversed edge directions. Since each vertex in G' has indegree larger than 0, this traversing can never stop.

Thus, the traversing must eventually repeat vertices, i.e., the subgraph G' must have cycles. Thus, if the algorithm returns at step 6, then the graph G must have cycles.

2. Let G be a directed graph with strongly connected components C_1, C_2, ..., C_k. The *component graph* G^c for G is a directed graph of k vertices w_1, w_2, ..., w_k such that there is an edge from w_i to w_j in G^c if and only if there is an edge from some vertex in C_i to some vertex in C_j. Develop an $O(|V|+|E|)$-time algorithm that on a given directed graph $G = (V, E)$ produces the component graph G^c for G. Make sure that there is at most one edge between two vertices in the component graph G^c.

Solutions. Denote by $[u, v]$ the edge from vertex u to vertex v. Use the algorithm given in class to construct all strongly connected components C_1, C_2, ..., C_k of G, which takes time $O(|V| + |E|)$. Assume that the result is stored in an array $Scc[0..n-1]$ with $1 \le Scc[v] \le k$ so that a vertex v in G is in the strongly connected component C_i if and only if $Scc[v] = i$. Let $CG[1..k]$ be an adjacency list for the component graph G^c, which, obviously contains at most $|V|$ vertices and $|E|$ edges.

```
SCC-CG(G) \\ G is a directed graph.
1.  use the SCC algorithm given in class to construct the array Scc[0..n − 1];
2.  E^c = ∅;
3.  for (each edge [u, v] in G)
3.1    if (Scc[u] ≠ Scc[v] )
3.2       add an edge [Scc[u], Scc[v]] to E^c;
4.  sort the edges in E^c then remove repetitive edges;
5.  for (each edge [x, y] in E^c)
5.1   add y to the linked list CG[x];
6.  return CG[1..k].
```

By the SCC algorithm given in the class, the strongly connected components of the graph G can be constructed in time $O(|V| + |E|)$ and named by the integers 1, 2, ..., k, respectively. Thus, for each vertex v in G, $Scc[v]$ takes a value between 1 and k. Therefore, if an edge $[u, v]$ in G has $Scc[u] = x$ and $Scc[v] = y$, with $x \neq y$, then we add an edge $[x, y]$ in step 3.2 to the set E^c. Steps 1-3 and 5-6 obviously take time $O(|V| + |E|)$. We need to give some explnations to step 4 that sorts the edges in E^c. Since each edge $[x, y]$ in E^c satisfies $1 \le x, y \le k$, we can use the linear-time sorting algorithm RadixSort to sort the edges in E^c in time $O(|E|+k) = O(|V|+|E|)$ (note that after step 3, E^c contains $|E|$ edges). Once the edges in E^c get sorted, we can easily go through the sorted edges in E^c and remove repetitive edges. Thus, step 4 of the algorithm takes time $O(|V| + |E|)$. Summarizing the above discussion, we conclude that the algorithm in total takes time $O(|V| + |E|)$. □

3. Design algorithms for Max(H), Insert(H, a), and Delete(H, i), where the set H is stored in a max-heap, a is the element to be inserted into the heap H, and i is the index of the element in the heap H to be deleted. Analyze the complexity of your algorithms.

Solutions. A heap H is a complete binary tree with possibly some of its rightmost leaves missing. Thus, a heap of n nodes has a height bounded by $\log n$. Moreover, because of its special structure, a heap H of n elements can be represented by an array $A[1... \max]$ and an index n so that $A[1...n]$ holds the n elements of H. If we place the elements in the heap level by level, starting from the root, and following the order from left to right, into the array $A[1...n]$,

then it is not difficult to verify that for an element $A[i]$ in the heap, $A[2i]$ and $A[2i+1]$ are the two children of $A[i]$, and $A[\lfloor i/2 \rfloor]$ is the father of $A[i]$.

A max-heap $A[1...n]$ is a heap that satisfies the condition that the value of each element in the heap is not smaller than the values of its children, i.e., $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$ for all $1 \leq i \leq \lfloor n/2 \rfloor$. It is easy to verify that for all i, the value $A[i]$ is not smaller than that of any of its descendants. In particular, the root $A[1]$ has the maximum value in the heap.

The algorithms for Max(H), Insert(H, a), and Delete(H, i) are given as follows. For more detailed discussions on heaps, the students are referred to the textbook, Chapter 6.

```
Max(A[1.. max], n) \\ find the maximum in the max-heap A[1..n]
1.  return (A[1]);

Insert(A[1.. max], n, a) \\ insert a new element a in the max-heap A[1..n]
1.  n = n + 1;  A[n] = a;  i = n;
2.  while (A[⌊i/2⌋] < A[i] and i ≥ 2)
2.1    swap A[⌊i/2⌋] and A[i];
2.2    i = ⌊i/2⌋;

Delete(A[1.. max], n, i) \\ delete the element A[i] in the max-heap A[1..n]
1.  A[i] = A[n];  n = n − 1;
2.  if (i ≥ 2 and A[i] > A[⌊i/2⌋])
2.1   while (i ≥ 2 and A[i] > A[⌊i/2⌋]) \\ pushing up
2.1.1    swap A[i] and A[⌊i/2⌋];  i = ⌊i/2⌋;
3.  else
3.1   while (2i ≤ n and (A[i] < A[2i] or A[i] < A[2i + 1])) \\ pushing down
3.1.1    if (2i = n or A[2i] > A[2i + 1])
            swap A[i] and A[2i];  i = 2i;
3.1.2    else
            swap A[i] and A[2i + 1];  i = 2i + 1;
```

It is easy to see that the algorithm `Min`(A, n) takes time $O(1)$. Each of the algorithms `Insert`$(A, n; a)$ and `Delete`$(A, n; i)$ starts from a node in the heap, and traverses in a path between the root and a leaf of the heap. Since a heap of n elements has its height bounded by $\log n$, each of the algorithms `Insert`$(A, n; a)$ and `Delete`$(A, n; i)$ takes time $O(\log n)$. □

4. Consider an extended version of the Shortest-Path problem. Suppose that you want to traverse from city s to city t. In addition, for some reason, you also need to pass through cities x, y, and z (in any order) during your trip. Your objective is to minimize the cost of the trip. The problem can be formulated as a graph problem as follows: Given a positively weighted directed graph G and five vertices s, t, x, y, z, find a path from s to t that contains the vertices x, y, z such that the path is the shortest over all paths from s to t that contain x, y, z, assuming that these paths are allowed to contain repeated vertices and edges. Develop an $O(m \log n)$-time algorithm for this problem. (**Hint.** In this question, you can assume Dijkstra's Shortest-Path algorithm.)

Solution. The desired path will have three additional "stops" during the trip:

$$s \to w_1 \to w_2 \to w_3 \to t,$$

where (w_1, w_2, w_3) is a proper ordering of the vertices x, y, z. Therefore, there are two questions that need to be answered: (1) how do we find the correct ordering for the vertices x, y, z in the trip, to make the trip short? and (2) when the ordering for x, y, z is given, how do we find the shortest path from s to t using that ordering?

We consider question (2) first. Assume (w_1, w_2, w_3) is an ordering of x, y, z. We claim that the following path P is the shortest among all s-t paths that pass through the vertices x, y, z in the ordering (w_1, w_2, w_3): the path P starts at s, follows a shortest path P_0 from s to w_1, then a shortest path P_1 from w_1 to w_2, then a shortest path P_2 from w_2 to w_3, and then a shortest path P_3 from w_3 to t. To see why this claim holds, let P' be any s-t path that passes through the vertices x, y, z in the ordering (w_1, w_2, w_3). Let P'_0, P'_1, P'_2, and P'_3 be the subpaths of P' that are, respectively, from s to w_1, from w_1 to w_2, from w_2 to w_3, and from w_3 to t. Since P_0, P_1, P_2, and P_3 are the shortest paths for those vertex pairs, respectively, we must have $\text{length}(P_i) \leq \text{length}(P'_i)$, for $i = 0, 1, 2, 3$. Therefore,

$$\begin{aligned} \text{length}(P) &= \text{length}(P_0) + \text{length}(P_1) + \text{length}(P_2) + \text{length}(P_3) \\ &\leq \text{length}(P'_0) + \text{length}(P'_1) + \text{length}(P'_2) + \text{length}(P'_3) = \text{length}(P'). \end{aligned}$$

That is, P is the shortest such path. As a conclusion, when an ordering (w_1, w_2, w_3) of x, y, z is given, we can call Dijkstra's algorithm (four times) to construct the shortest paths that are, respectively, from s to w_1, from w_1 to w_2, from w_2 to w_3, and from w_3 to t. The concatenation of these paths gives an s-t path that is the shortest among all s-t paths that pass through the vertices x, y, z in the ordering (w_1, w_2, w_3).

Now we get to question (1): how do we determine the best ordering of x, y, z? Currently there is no better way than exhaustive searching. Fortunately, there are in total only $3! = 6$ different orderings for x, y, z. Thus, we can try all of them. This gives us the following algorithm for the given problem:

```
Multi-Stop-Path(G, s, t, x, y, z) \\ G is a positively weighted graph.
1.  P = ∅;  wt(P) = +∞;
2.  for (each permutation (w1, w2, w3) of x, y, z)
2.1    call Dijkstra's algorithm to construct the shortest paths P0, P1, P2, and P3 that are,
       respectively, from s to w1, from w1 to w2, from w2 to w3, and from w3 to t;
2.2    Let P' be the concatenation of P0, P1,P2, P3;
2.3    if (the length wt(P') of P' is smallest than wt(P))
       then P = P';  wt(P) = wt(P');
3.  return (the path P).
```

On each permutation of x, y, z, the `for`-loop in the algorithm calls Dijkstra's algorithm four times to construct the shortest s-t path following that permutation. Since there are $3! = 6$ permutations of x, y, z, the algorithm basically calls Dijkstra's algorithm 24 times to find the shortest s-t path that contains x, y, z. Thus, its running time is of the same order as Dijkstra's algorithm, which is $O(m \log n)$. □

Remark 1. A natural question on the above algorithm is: what if there is no s-t path that contains all vertices x, y, and z? This can be handled as follows. We know that Dijkstra's algorithm can properly handle the issue when there is no path from s to t: in this case, when Dijkstra is running out of fringes, the vertex t still remains "unseen". Thus, we can assume that when there is no s-t path in the input graph, Dijkstra's algorithm will report so. Now for two vertices v_1 and v_2, if there is no path from v_1 to v_2, then we define the weight of the path from v_1 to v_2 as $+\infty$. As a consequence, for a given ordering (w_1, w_2, w_3) of x, y, z, if one of the s-w_1 path P_0, the w_1-w_2 path P_1, the w_2-w_3 path P_2, and the w_3-t path P_3 does not exist, then the concatenation P' of P_0, P_1, P_2, P_3 constructed in step 2.2 will have weight $+\infty$, which will not replace the current path P. In particular, if there is no s-t path that contains x, y, z, at all, then for all orderings of x, y, z, the path P' constructed in step 2.2 has weight $+\infty$ and will not

replace the path P. Thus, the path P remains as $\emptyset$ with weight $+\infty$, and will be returned in step 3, which is interpreted as a report that there is no s-t path contains the vertices x, y, z.

Remark 2. An algorithm with 24 calls to Dijkstra's algorithm seems a bit too much, in particular in practice. In fact, it is rather easy to reduce the number of calls to Dijkstra's algorithm. For example, very simple observations will show that you can solve the problem based on the same idea but using only 12 calls to Dijkstra for directed graphs, and only 9 calls to Dijkstra for undirected graphs (HOW?) In fact, I think I can further improve this so that I can solve the problem using only 4 calls to Dijkstra for directed graphs and using only 3 calls to Dijkstra for undirected graphs. Can you do this, or can you do even better?