# CSCE-629 Analysis of Algorithms

**Spring 2019**

**Instructor:** Dr. Jianer Chen
**Office:** HRBB 315C
**Phone:** 845-4259
**Email:** `chen@cse.tamu.edu`
**Office Hours:** T,Th 11:00 am–12:30 pm

**Teaching Assistant:** Qin Huang
**Office:** HRBB 315A
**Phone:** 402-6216
**Email:** `huangqin@email.tamu.edu`
**Office Hours:** MWF 3:00 pm–4:00 pm

## Solutions to Assignment #3

**(Prepared with TA Qin Huang)**

**1.** 1. Suppose that we have a sequence of MakeSet-Find-Union operations in which no Find appears before any Union. What is the computational time for this sequence?

*Solutions.* Suppose that the length of the sequence of MakeSet-Find-Union operations is $m$ and that the number of elements in the set $S$ is $n$. We claim that the computational time for this sequence is $O(m + n)$.

Each of the MakeSet and Union operations takes constant time. Thus, the total time for all MakeSet and Union operations is bounded by $O(m)$.

For the Find operations, we follow the same idea as presented in class, but define boundary/internal nodes differently. A node on a Find-path in a tree $T$ is a *boundary node* if the node is a child of the root of $T$, and is an *internal node* if it is not a child of the root. Since there are at most $m$ Find operations in the sequence, the total number of boundary nodes is bounded by $m$. Moreover, we claim that for any element $a$ in the set $S$, there is at most one internal node labeled $a$. To see this, note that if a node $v$ labeled $a$ is an internal node on a Find-path, then the node $v$ will become a child of the root $r$ of the tree after the Find operation. Since no Union appears after any Find, the tree root $r$ remains as a tree root for the rest of the computation, thus, the node $v$ labeled $a$ will remian as a child of the root $r$ and cannot be an internal node. As a consequence, there are at most $n$ internal nodes, each is labeled by a different element in the set $S$. Therefore, the total number of boundary nodes and internal nodes, which is equal to the sum of the lengths of Find-paths in the sequence, is bounded by $m + n$. As shown in class, this means that the total time spent on all the Find operations in the sequence is bounded by $O(m+n)$. Thus, the total computational time for the sequence is also bounded by $O(m+n)$. □

**2.** (Question 24.1-5, Textbook, p. 655) Let $G = (V, E)$ be a weighted directed graph. Develop an $O(nm)$-time algorithm that finds the value $\delta(v)$ for every vertex $v$, which is defined as:

$$\delta(v) = \min_{w \in V}\{\text{the length of the shortest path in G from } w \text{ to } v\}.$$

*Solutions.* The algorithm is a modification of Bellman-Ford algorithm, as given in Algorithm 1, where $wt(*, *)$ is the weight function for edges in the graph $G$.

**Algorithm 1. Pseudocode for Problem 2**
Input: $G = (V, E)$

```
1. for (each vertex $v \in V$) $D[v] = 0$;
2. for ($i = 1$ to $|V| - 1$)
2.1    for (each edge $[u, v] \in E$)
2.2        if ($D[u] + wt(u, v) < D[v]$)  $D[v] = D[u] + wt(u, v)$;
3. for (each edge $(u, v) \in E$)
       if ($D[u] + wt(u, v) < D[v]$)  return ("negative cycles exist.");
4. return $D[\cdot]$;
```

Assume the graph $G$ contains no negative cycles. For any two vertices $u$ and $w$ in the graph $G$, denote by $dist[u, w]$ the distantce from $u$ to $w$, i.e., the weight of the shortest path from $u$ to $w$. We claim that in the output $D[\cdot]$ of the algorithm, we have $D[v] = \delta(v)$ for every vertex $v$.

First observe that we cannot have $D[v] < \delta(v)$ for any vertex $v$. This can be proved by induction on the number of times step 2.2 of the algorithm is executed. Before the first execution of step 2.2, we have (because of step 1) $D[w] = 0 \geq \delta(w)$ for every vertex $w$. Assume that before an execution $D[v] = D[u] + wt(u, v)$ of step 2.2 on an edge $[u, v]$, we have $D[w] < \delta(w)$ for every vertex $w$. Then the execution of $D[v] = D[u] + wt(u, v)$ gives

$$D[v] = D[u] + wt(u, v) \leq \delta(u) + wt(u, v) \leq \delta(v),$$

where the inequality $D[u] \leq \delta(u)$ comes from the inductive hypothesis, while the inequality $\delta(u) + wt(u, v) \leq \delta(v)$ is because the path (from some vertex) of weight $\delta(u)$ to the vertex $u$ plus the edge $[u, v]$ gives a path to the vertex $v$ of weight $\delta(u) + wt(u, v)$. Therefore, $D[w] < \delta(w)$ remains true for every vertex $w$ (including the vertex $v$) after the execution of $D[v] = D[u] + wt(u, v)$ in step 2.2 so the induction goes through. This proves our claim.

Now we prove that $D[v]$ cannot be larger than $\delta(v)$ for any vertex $v$. Let $w$ be a vertex such that $dist[w, v] = \delta(v)$ (the existence of $w$ is ensured because the graph $G$ has no negative cycles). Let $P = \langle w = w_0, w_1, \ldots, w_{k-1}, w_k = v\rangle$ be a shortest path of weight $\delta(v)$ from $w$ to $v$ in the graph $G$. We prove that after $k$ executions of the **for**-loop in step 2 (i.e., steps 2.1-2.2), $D[v]$ is equal to the weight $\delta(v)$ of the path $P$. The proof is by induction on the length $k$ of $P$.

If $k = 0$, then $w = v$ and the weight $\delta(v)$ of the path $P$ is 0. Since $D[v] = 0$ by step 1 of the algorithm, after 0 executions of the **for**-loop in step 2, we have $D[v] = \delta(v) = 0$. Thus, the induction holds true for the initial case $k = 0$.

Now suppose that $k > 0$ and the length of the path $P$ is at least 1. consider the vertex $w_{k-1}$ in the path $P$. Note that we must hvae $\delta(w_{k-1}) = \delta(v) - wt(w_{k-1}, v)$. In fact, the existence of the path $\langle w = w_0, \ldots, w_{k-1}\rangle$ of weight $\delta(v) - wt(w_{k-1}, v)$ shows that $\delta(w_{k-1}) \leq \delta(v) - wt(w_{k-1}, v)$. On the other hand, if $\delta(w_{k-1}) < \delta(v) - wt(w_{k-1}, v)$, then the path of weight $\delta(w_{k-1})$, which is from some vertex $w'$ to vertex $w_{k-1}$, plus the edge $[w_{k-1}, v]$ would give a path to $v$ whose weight is strictly smaller than $\delta(v)$, contradiction the definition of $\delta(v)$. Thus, the path $\langle w = w_0, \ldots, w_{k-1}\rangle$ is a path of weight $\delta(w_{k-1})$ to the vertex $w_{k-1}$ that has length $k-1$. By the inductive hypothesis, after the $(k-1)$st executions of the **for**-loop in step 2, we have $D[w_{k-1}] = \delta(w_{k-1})$. Therefore, in the $k$th execution of the **for**-loop in step 2 when the edge $[w_{k-1}, v]$ is examined in step 2.1, step 2.2 will set $D[v] = D[w_{k-1}] + wt(w_{k-1}, v) = \delta(w_{k-1}) + wt(w_{k-1}, v) = \delta(v)$. After this, $D[v]$ will stay as $\delta(v)$ because by what we have proved above, $D[v]$ cannot be smaller than $\delta(v)$, so step 2.2 cannot be applied to further decrease $D[v]$.

Because there are no negative cycles, for every vertex $v$, there must be a simple shortest path of weight $\delta(v)$ from some vertex to $v$. The path has a length at most $|V| - 1$. Thus, $|V| - 1$

executions of the **for**-loop in step 2 guarantees the correctness of $D[v]$ for all vertices $v$. For step 3, observe that the above analysis shows that if there are no negative cycles, then $D[v] = \delta(v)$ for all vertices $v$ after step 2. Therefore, there cannot be any further improvement on $D[v]$ for any vertex $v$. Thus, any further improvement would imply the exitence of negative cycles. Note that if there are negative cycles, then the value $\delta(v)$ is not well-defined for some vertices $v$ (e.g., the vertices on a negative cycle).

The time complexity of the algorithm is obviously $O(mn)$. □

**3.** Assume that multiple edges are allowed in a weigted graph $G$, which is given in an adjacency list $G[n]$. Therefore, for each vertex $v$, the linked list $G[v]$ for $v$ may contain elements of the forms $[u, w_1]$ and $[u, w_2]$, standing for two edges from $v$ to $u$, with weights $w_1$ and $w_2$, respectively. Now the goal is that for each pair of vertices in the graph $G$, we want to remove all multiple edges except the one with the largest weight. Design a linear-time algorithm for this problem.

*Solutions.* The difficulty here is that multiple edges between two vertices $v$ and $w$ may not appear consecutively in the linked list $G[v]$ (and $G[w]$) so it is not easy to identify the one with the largest weight and remove all others. We cannot afford to sort the linked list $G[v]$ for each vertex $v$ – sorting each list takes time at least $O(n)$, which would require time $O(n^2)$ to make all linked lists sorted.

Instead, we can use a single sorting to make multiple edges between any two vertices appear consecutively, as follows. Since the graph $G$ is given by the adnacency list $G[n]$, the vertices of $G$ are named by the integers 0, 1, …, $n-1$. By scanning the linked list $G[v]$ for each vertex $v$ in $G$, we get all edges incident to $v$, which are given by triples of the form $(v, u, w_{vu})$, standing for the edge between $v$ and $u$ of weight $w_{vu}$. Therefore, at the end of this process, we get a collection $C$ of triples of the form $(v, u, w_{vu})$, where each triple is for an edge in the graph $G$. This process obviously takes time $O(n + m)$, where $n$ and $m$ are the number of vertices and the number of edges in the graph $G$, respectively.

Then we sort the triples in $C$ using their first two components, i.e., $(v_1, u_1, w') \leq (v_2, u_2, w'')$ if either $v_1 < v_2$ or $v_1 = v_2$ and $u_1 \leq u_2$. Note that the first two components of the triples are vertex names, thus are integers between 0 and $n - 1$. Thus, these triples can be regarded as $m$ 2-digit numbers in which each digit takes its value from one of the $n$ integers betweem 0 and $n-1$. Therefore, we can use Radix-Sort to sort them in time $O(m+n)$ (See Lemma 8.3 in page 198 of the textbook). Let $L$ be the list of the sorted triples obtained in this process.

It is easy to see that in the list $L$, multiple edges between two vertice appear consecutively. Thus, now it is trivial to go through the list $L$ and for each pair $(v, u)$ of vertices, remove all edges but the one with the largest weight. Let the resulting list be $L'$. It is also trivial to use the list $L'$ to make a new adjacency list for a graph $G'$, which is the graph $G$ with all multiple edges removed except the one with the largest weight. Also, this can be done in time $O(n + m)$ in a straightforward way.

Putting all these together gives a linear-time algorithm that constructs the desired graph. □

**4.** Modify the QuickSort algorithm so that the pivot is selected using the linear-time Median-Finding algorithm. Prove that this modified QuickSort algorithm takes time $O(n \log n)$ in the worst case. Discuss why this algorithm is not used in practice.

*Solutions.*

Denote the set of $n$ elements by $S$. The algorithm is as follows:

1. Use the Median-Finding algorithm to calculate the median $m$ of $S$;

2. Partition $S$ into two subsets $S_l = \{x : x < m, x \in S\}, S_r = \{x, x > m, x \in S\}$;
3. Recursively sort $S_l$ and $S_r$;
4. Concatenate the two sorted lists $S_l$ and $S_r$.

Because we assume a linear-time Median-Finding algorithm to find the median $m$ of the set $S$ in step 1, and step 2 and step 4 obviously take time $O(n)$, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, which, as we know, has a solution $T(n) = O(n \log n)$. Thus, this modified QuickSort algorithm takes time $O(n \log n)$ in the worst case, which is better than the standard QuickSort algorithm in the worst-case scenario analysis.

However, this version of QuickSort, which asymptotically is optimal in the worst-case, is not a popular one in practical computing, for the following reasons. Although the Median-Finding algorithm runs in linear time, the hidden constant in its complexity $O(n)$ is rather large. This leads to a large hidden constant in the complexity $O(n \log n)$ of the modified QuickSort algorithm. As a consequence, compared to other $O(n \log n)$-time sorting algorithms (such as MergeSort and HeapSort), the modified QuickSort algorithm turns out to be less efficient. On the other hand, the standard QuickSort algorithm, though taking time $O(n^2)$ in the worst case, only achieves its worst-case complexity for very rare instances in practice. In particular, the pivot selection process in the standard QuickSort (in particular for the versions that use certain randomness in pivot selection) in most cases picks a pivot that, although not the perfect median in general, splits the input into two parts with somehow balanced sizes. This, in most cases, makes the standard QuickSort run in time $O(n \log n)$ with the hidden constant in $O(n \log n)$ much smaller than the one for the modified QuickSort. Thus, if QuickSort is the best approach, then the standard QuickSort in most cases performs better than the modified QuickSort.

You can think of other reasons by your own understanding. □