AI2615 算法设计与分析 2020-2021 春季学期

Lecture 5 – Graph Algorithms: Finding the Shortest Path

2021 年 3 月 26 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

We will continue studying graph algorithms. In this lecture, we will study various algorithms for finding the shortest path under various settings.

Definition 1. Given a (directed or undirected) graph $G = (V, E)$ and two vertices $u, v \in V$, the *distance* between u and v is the length of the shortest path between u and v.

1 Finding Shortest Path in Unweighted Graphs

Problem 2. Given a (directed or undirected) graph $G = (V, E)$ and a vertex $s \in V$, find the distances between s and all vertices in V.

In the last lecture, we have seen that the graph searching algorithm depth-first search (DFS) and how it is used to find the (strongly) connected components in a graph. Can we use it here to find the distance between two vertices?

Starting from a given vertex, the depth-first-search, as its name suggests, goes as "deep" as possible until it reaches a vertex all of whose neighbors have already been visited. In particular, when DFS goes from a vertex u to one of its neighbors v, DFS will only go to u's other neighbors after exploring all those unexplored vertices reachable from v.

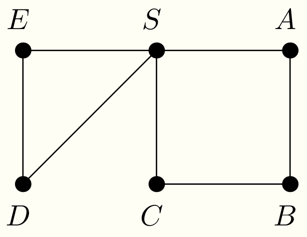

Figure 1: Find the distance between S and each of A, B, C, D, E.

Suppose we are using DFS for the graph in Figure 1 that starts from vertex S. DFS will find one of S's neighbors and explore it. Suppose A is explored. Instead of exploring other S's neighbors, C and E, DFS will first explore A's neighbor, B, and will then go to C. After observing that all of C's neighbors, S and B, have already been explored, DFS will then go backward to B. After seeing all B's neighbors, C and A, have already been explored, DFS will then go backward to A, and finally to S. Until then, it will explore S's other unexplored neighbors D and E. The search path for DFS is shown on the left-hand side of Figure 2.

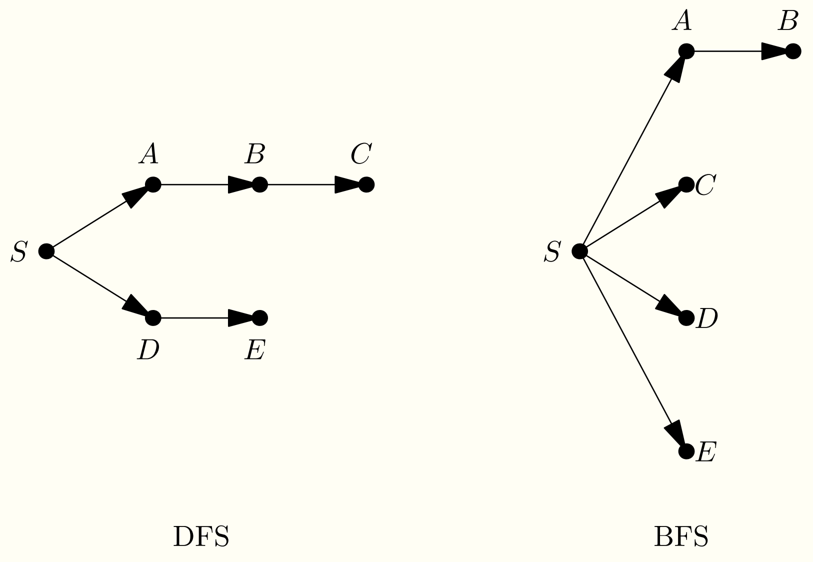

Figure 2: The search paths for DFS and BFS.

It is then clear to see that DFS does not search the vertices according to the distances. For example, vertex C has distance 1 from S, but it is only explored after two more vertices A and B. We need another graph searching algorithm that only goes further after *all of S's neighbors have been explored.* We require an algorithm that has search path shown on the right-hand side of Figure 2. This is what *breadth-first search* (BFS) does!

Suppose we maintain a sequence of vertices who have been visited but whose neighbors have not been checked. Starting from S, all its neighbors A, C, D, E are stored. Suppose A is the first vertex in the sequence. When we explore A's neighbor, B, we need to make sure that B is put *at the end* of the sequence, to ensure that B will be checked after C, D, E. This naturally suggests us to use a *queue* to maintain the sequence. Algorithm 1 demonstrates how BFS finds the distance from s to each vertex in the graph G. If a *stack* was used, B will be put *at the beginning* of the sequence after A is removed, and the algorithm becomes the DFS algorithm in the last lecture.

Theorem 3. *After executing Algorithm 1, for any $u \in V$, dist(u) is the distance between s and u.*

Proof. We will prove this by induction. For any $d \in \mathbb{Z}_{\geq 0}$, let $P(d)$ be the following statement.

$P(d)$: At some stage of the algorithm, we have

1. for each vertex u with distance at most d from s, $\text{dist}(u)$ equals to the distance between s and u;
2. each vertex w with distance more than d from s satisfies $\text{dist}(w) = \infty$;
3. Q contains exactly those vertices at distance d from s.

For the base step, $P(0)$ is clearly true: all the three conditions are satisfied after Line 3 of the algorithm.

For the inductive step, suppose $P(d)$ is true. We will show that $P(d+1)$ is true. Suppose we are at a stage where all the three conditions are satisfied for d. Let Q_d be the set of vertices in Q at current stage. We

Algorithm 1 Breadth-First Search

```
BFS($G = (V, E)$, $s \in V$)
 1: initialize dist($u$) $= \infty$ for each $u \in V$
 2: dist($s$) $\leftarrow 0$
 3: Queue $Q \leftarrow [s]$
 4: while $Q \neq \emptyset$
 5:     $u \leftarrow Q$.pop();
 6:     for each $v : (u, v) \in E$
 7:         if dist($v$) $= \infty$          // dist($v$) $= \infty$ implies $v$ has not been explored
 8:             dist($v$) $\leftarrow$dist($u$) $+ 1$
 9:             $Q$.push($v$)
10:         endif
11:     endfor
12: endwhile
```

show that the three conditions are satisfied after $|Q_d|$ iterations of the while-loop, and this will concludes $P(d+1)$.

In the next $|Q_d|$ iterations of the while-loop, exactly those unexplored vertices (with dist set to ∞) adjacent to vertices in Q_d are explored, and each of such vertices w satisfies $\text{dist}(w) = d+1$ by Line 7-10 (since each $u \in Q_d$ satisfies $\text{dist}(u) = d$ by Condition 1 and 3 of the induction hypothesis $P(d)$). For each vertex w with distance $d+1$ from s, we have $\text{dist}(w) = \infty$ before those $|Q_d|$ iterations (by Condition 2 of $P(d)$) and w must be adjacent to a vertex with distance d from s. By Condition 3 of $P(d)$, Q_d contains exactly those vertices at distance d, so $\text{dist}(w)$ will be set to $d+1$ in the next $|Q_d|$ iterations. This proves Condition 1 of $P(d+1)$. Those vertices with distance at least $d+2$ from s are not adjacent to any vertex in Q_d (since Q_d only contains vertices at distance d), so dist is set to ∞ for those vertices. This proves Condition 2 of $P(d+1)$. After $|Q_d|$ iterations, those vertices with distance d have been removed from Q. We have seen that all the vertices with distance $d+1$ are added to Q after $|Q_d|$ iterations, and no other vertex is added, so Condition 3 of $P(d+1)$ is also proved.

Since $P(d)$ is true for all $d \in \mathbb{Z}_{\geq 0}$, Condition 1 implies this theorem. $\square$

The running time for Algorithm 1 is $O(n+m)$, since each edge has been visited $O(1)$ times. To be exact, each edge has been visited once if G is directed, and twice if G is undirected.

2 Finding Shortest Path in Weighted Graphs with Non-negative Weights

Problem 4. Given a weighted (directed or undirected) graph $G = (V, E, \ell)$ such that $\ell(u, v) \geq 0$ for each $(u, v) \in E$ and a vertex $s \in V$, find the distances between s and all vertices in V.

A natural way to handle integer-weighted graphs is to break each edge into unit-length segments and introducing dummy vertices between those segments. This translates a weighted graph to an unweighted graph, and we can then use BFS to find the distances between s and all the other vertices. Figure 3 illustrates this technique.

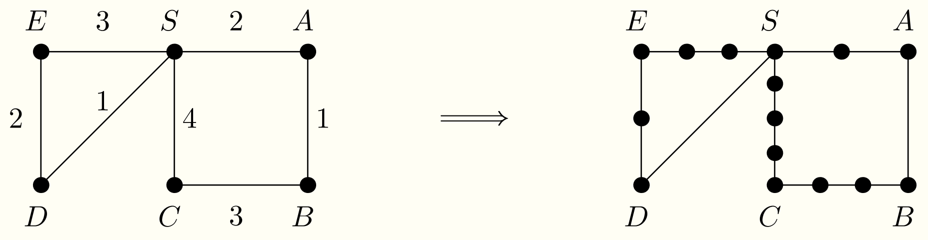

Figure 3: Breaking edges into unit-length segments.

However, the time complexity for this technique depends on the weights of the edges, as the weights of the edges affect the size of the graph constructed. This technique can be very inefficient if edges have large weights. For example, the graph in Figure 4 (only look at the graph on the left-hand side at this moment) contains only three vertices, but the BFS algorithm will have to process a graph with 350 vertices! In this example, for the first 99 iterations of the BFS, we are exploring those dummy vertices between S and A and the first 99 vertices of those 199 vertices between S and B. There is nothing interesting happening during these iterations. Suppose one while-loop iteration of the BFS algorithm takes one unit of time. The first "interesting" event happens at time $T = 100$ when we reach vertex A. Can we "take a nap" when the BFS algorithm is performing the first 99 iterations and setup an "alert clock" that rings at $T = 100$?

When we start from S, we can setup an alert clock for each of S's neighbors at a time that is equal to the length of the corresponding edge. Thus, A is set to ring at $T = 100$ and B is set to ring at $T = 200$. We find the next time when an alert clock rings. This corresponds to $T = 100$ where the alert clock for A rings. We then know that the distance between S and A is 100, as the alert clock for A is the first ringing one, meaning that the BFS algorithm reaches A before any other vertices. After we have reached A, we update the alert clock for each of A's neighbors. Therefore, the alert clock for B is set to ring at $T = 150$. Figure 4 illustrates the above.

In general, when we are at a given vertex u, we setup or update the alert clock for each of u's neighbors. The alert clock for each of u's neighbors v is only an *upper bound* for the distance between v and s, as we may later find a shortcut to v and update v's alert clock. However, if an alert clock for u rings, we know that the current time is the distance between u and s. This alert clock technique can be viewed as a

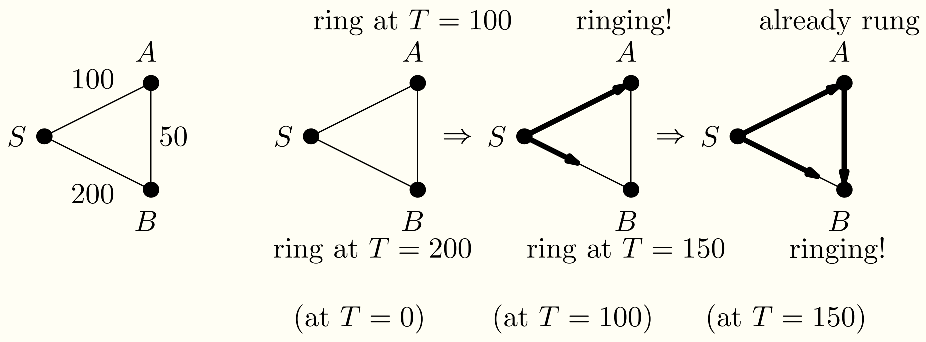

Figure 4: Alert clock technique: At time $T = 0$, A is set to ring at $T = 100$ and B is set to ring at $T = 200$; at time $T = 100$, the BFS reaches A, A rings, and B is set to ring at $T = 150$; at time $T = 150$, B rings.

speedup version of BFS. Below we informally describe our alert clock technique.

1. set the alert clock at s to $T = 0$ and let it ring;
2. **while** there is an alert clock that has not rung;
3.   let u be the vertex whose alert clock will ring next;
4.   **for** each of u's neighbor v whose alert clock has not rung:
5.   **if** v's ringing time is later than u's ringing time plus $\ell(u, v)$, update v's ringing time;
6. terminate the algorithm when all the alert clocks have rung, and the distance between each vertex u and s is u's ringing time.

Dijkstra's algorithm, shown in Algorithm 2, implements this alert clock technique.

Algorithm 2 Dijkstra's algorithm

```
DIJKSTRA($G = (V, E, \ell)$, $s \in V$)
1:  initialize dist($u$) = $\infty$ for each $u \in V$ and initialize pre($u$) = NIL
2:  dist($s$) $\leftarrow$ 0
3:  $Q = [(s, \text{dist}(s))]$
4:  while $Q \neq \emptyset$
5:      $u \leftarrow Q$.popmin()  // $Q$.popmin() find and remove an element with smallest dist($u$)
6:      for each $v : (u, v) \in E$
7:          if dist($v$) > dist($u$) + $\ell(u, v)$
8:              dist($v$) = dist($u$) + $\ell(u, v)$
9:              pre($v$) = $u$
10:             $Q$.modify(($v$, dist($v$)))
11:         endif
12:     endfor
13: endwhile
```

The array “pre” in Algorithm 2 is used to keep track to the shortest path. In particular, $\text{pre}(u)$ is the vertex right before u in a shortest path from s to u.

To help understanding the correctness of Dijkstra’s algorithm, the algorithm maintains a set R containing those vertices whose distances from s are already determined. In other words, R contains those vertices whose alert clocks have rung. Initially, R only contains vertex s, and at the end of the algorithm, R contains all the vertices (if the graph is (strongly) connected). At each intermediate step, Dijkstra’s algorithm is essentially finding a vertex $u \in V \setminus R$ that has minimum distance from s (the next ringing alert clock) and adding this vertex to R, then it updates “dist” for all u’s neighbors that is in $V \setminus R$.

Theorem 5. *After executing Algorithm 2, for any $u \in V$, $\text{dist}(u)$ is the distance between s and u.*

Exercise 6. Prove Theorem 5.

At Line 5, instead of popping the first element in Q, we need to find an element with smallest $\text{dist}(u)$. Storing Q as a queue is no longer helpful. In fact, the running time for Dijkstra’s algorithm depends on the data structure we used for Q.

Let T_{pop} be the time requires to execute Line 5, and let T_{mod} be the time requires to execute Line 10. The overall time complexity is given by $O(nT_{\text{pop}} + mT_{\text{mod}})$. If we use a naïve array implementation, we have $T_{\text{pop}} = O(n)$ and $T_{\text{mod}} = O(1)$, which gives overall time complexity $O(n^2)$. If we use a *heap*, we have $T_{\text{pop}} = O(\log n)$ and $T_{\text{mod}} = O(\log n)$, which gives overall time complexity $O((m+n)\log n)$. Therefore, if the graph is *sparse*, a heap is more preferable; if the graph is *dense*, an array is more preferable.

We can generalize the standard heap to a d-ary heap: instead of using a binary tree as it is in the standard heap, we use a d-ary tree. It is easy to verify that $T_{\text{pop}} = O(d\log n/\log d)$ and $T_{\text{mod}} = O(\log n/\log d)$. The optimal choice is $d \approx m/n$, which is the average degree of the graph. This gives overall time complexity $O(m\log n/\log(m/n))$. For sparse graphs with $m = O(n)$, this is $O(n\log n)$, which is as good as if a heap was used. For dense graphs with $m = \Omega(n^2)$, this is $O(n^2)$, which is as good as if an array was used. For graphs with intermediate density $m = \Theta(n^{1+\delta})$, this is $O(m)$, which is optimal.

A known better data structure for Dijkstra’s algorithm is a *Fibonacci heap*. It can achieve $T_{\text{pop}} = O(\log n)$, and it can achieve an *amortized cost* for T_{mod} with $T_{\text{mod}} = O(1)$. This means that the modify operations can take various amount of time, but the *average* time complexity is $O(1)$.

Implementation	T_{pop}	T_{mod}	Overall
Array	$O(n)$	$O(1)$	$O(n^2)$
Heap (standard)	$O(\log n)$	$O(\log n)$	$O((m+n)\log n)$
d-ary heap	$O\left(\frac{d\log n}{\log d}\right)$	$O\left(\frac{\log n}{\log d}\right)$	$O\left((nd+m)\frac{\log n}{\log d}\right)$
Fibonacci heap	$O(\log n)$	$O(1)$ (amortized)	$O(n\log n + m)$

Table 1: The time complexity for Dijkstra’s algorithm with different data structures for Q

3 Finding Shortest Path in General Weighted Graphs

In this section, we consider weighted graphs that can contain edges with negative weights. Suppose a graph contains a negatively-weighted cycle (a cycle such that the sum of the weights of all its edges is negative). If the cycle is reachable from s and a vertex u can be reached from the cycle, the distance between s and u is $-\infty$, as we can keep looping around the cycle to reduce the distance travelled. To circumvent this, we can define our problem in the following two different ways.

1. Define the distance between s and u to be the length of the shortest *simple path* from s to u. A simple path is a path that does not visit any vertex more than once.
2. If a graph contains a negatively-weighted cycle, the algorithm needs to decide it; otherwise, the algorithm needs to find the distance from s to all vertices in the graph.

The first problem is known to be NP-hard. We will focus on the second problem.
In Sect. 3.1, we consider this problem in general; in Sect. 3.2, we consider the special case where the graph is a directed acyclic graph.

3.1 General Graphs

Problem 7. Given a weighted (directed or undirected) graph $G = (V, E, \ell)$ and a vertex $s \in V$, decide the following:

- whether or not the graph contains a negatively-weighted cycle;
- if not, find the distances between s and all vertices.

Dijkstra's algorithm does not work under this setting. Consider the counterexample shown in Figure 5. Dijkstra's algorithm will find A in the first while-loop iteration. Then A will be removed from Q and its distance from S will be finalized to 10. However, we know that the shortest S-A path is $S \to B \to A$ which has length 8.

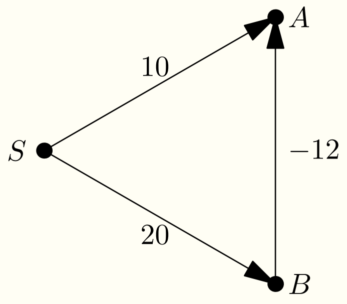

Figure 5: A counterexample showing that Dijkstra's algorithm fails for weighted graphs with possibly negatively-weighted edges.

If we check Algorithm 2 carefully, the part making it fail for general weighted graph is Line 5, where u is removed from Q and the distance from s to u is finalized to $\text{dist}(u)$. Such a removal is unsafe under the presence of edges with negative weights: it is possible that there is a vertex w whose distance from s is more than that of u, but there is a negatively-weighted edge from w to u which makes the path from s to u via w shorter. On the other hand, the update part at Line 7-8 is still functional for general weighted graphs. Notice that $\text{dist}(v)$ is an upper bound for the distance between s and v in the course of the algorithm. If we realize that the distance from s to u is at most $\text{dist}(u)$ and we have $\text{dist}(v) > \text{dist}(u) + \ell(u, v)$, it is certainly safe to update the upper bound for the distance between s and v to $\text{dist}(u) + \ell(u, v)$.

Observing those, one natural idea to fix Dijkstra's algorithm is to perform the update step for a sufficient large number of times for each edge, until we are ready to conclude the distances from s to all the other vertices. That is, we choose a large number T, and repeat the following for T times:

- For each $(u, v) \in E$, if $\text{dist}(v) > \text{dist}(u) + \ell(u, v)$, update $\text{dist}(v) \leftarrow \text{dist}(u) + \ell(u, v)$.

It remains to decide a safe value of T.

A first observation is that, *once $\text{dist}(u)$ correctly reflects the distance from s to u, it will remain being correct throughout the rest of the algorithm.* This is because $\text{dist}(u)$ is always an upper bound for the distance from s to u that can only be decreased throughout the algorithm. Once $\text{dist}(u)$ decreases to the correct distance, it cannot be further decreased while still being an upper bound.

To obtain a second observation, we consider the number of edges contained in the shortest path between s and a vertex u. At the beginning of the algorithm, $\text{dist}(s)$ is correctly set to 0. Consider a neighbor u of s such that the length of the shortest path from s to u contains only one edge (in this case, the path only contains the edge (s, u)). After one iteration of updating all edges, $\text{dist}(u)$ is correctly set to the actual distance from s to u, which is $\ell(s, u)$, and it will remain being correct by our first observation. Next, consider a vertex v such that the length of the shortest path from s to v contains two edges: $\{(s, u), (u, v)\}$. In this case, v may or may not be a neighbor of s. We know that there exists some vertex u such that the shortest path from s to u contains only one edge $\{(s, u)\}$, and $s \to u \to v$ is a shortest s-v path, although we do not know which vertex is u. After one iteration of updating all edges, since the shortest path from s to u contains one edge, $\text{dist}(u)$ is set correctly, and we know that the actual distance between s and v is $\text{dist}(u) + \ell(u, v)$ since $s \to u \to v$ is the shortest path. After two iterations, $\text{dist}(v)$ will be updated to $\min\{\text{dist}(w) + \ell(w, v) \mid w : (w, v) \in E\}$ if this is smaller than current value $\text{dist}(v)$. In particular, $\text{dist}(u) + \ell(u, v)$ is one of the candidates for $\text{dist}(v)$, and we have $\text{dist}(v) \leq \text{dist}(u) + \ell(u, v)$. Since we have seen that $\text{dist}(u) + \ell(u, v)$ is the actual distance and we know that $\text{dist}(v)$ is always an upper bound for this distance, we know that the distance for v is correctly set to $\text{dist}(u) + \ell(u, v)$. Notice that in the above reasoning, there are other updates at the remaining part of the graph, and we do not need to care about them.

In general, we state our second observation: *after t iterations, for any vertex u such that the shortest path between s and u contains at most t edges, $\text{dist}(u)$ correctly reflects the distance between s and u.*

Let us suppose that the graph does not contain a negatively-weighted cycle at this moment. Then the shortest path between s and any vertex u can contain at most $n-1$ edges. Otherwise, the path will have to visit a vertex twice, and we can find the shortcut with the edges between the two visits removed. We have conclude that $T = n-1$ iterations are sufficient by our second observation. This gives us Bellman-Ford algorithm, shown in Algorithm 3.

Algorithm 3 Bellman-Ford algorithm

```
BELLMAN-FORD($G = (V, E, \ell)$, $s \in V$)
1: initialize dist($u$) to $\infty$ for each $u \in V$
2: dist($s$) $\leftarrow 0$
3: for $t = 1, \dots, n-1$:
4:     for each edge $(u, v) \in E$
5:         if dist($v$) > dist($u$) + $\ell(u, v)$: update dist($v$) $\leftarrow$ dist($u$) + $\ell(u, v)$
6:     endfor
7: endfor
```

Theorem 8. *Suppose $G = (V, E, \ell)$ contains no negatively-weighted cycle. After executing Algorithm 3, for each $u \in V$, $dist(u)$ is exactly the distance between s and u.*

Proof. We will prove the following two observations.

- Observation 1: once dist(u) correctly reflects the distance from s to u, it will remain being correct throughout the rest of the algorithm.
- Observation 2: after t iterations of the for-loop at Line 3, for any vertex u such that the shortest path between s and u contains at most t edges, dist(u) correctly reflects the distance between s and u.

Observation 1 becomes apparent once noticing that dist(u) is always an upper bound for the distance between s and u and that the value of dist(u) can only decrease throughout the algorithm. Observation 2 will be proved by induction.

The base step $t = 0$ is trivial: Line 2 correctly set the value for dist(s) before the for-loop starts.

For the inductive step, suppose Observation 2 is true for some $t \in \mathbb{Z}_{\geq 0}$. After $t+1$ iterations, by the induction hypothesis, dist(u) has already been correctly set for any vertex u such that the shortest path between s and u contains at most t edges (it has already been correctly set after the first t iterations). Therefore, we only worry about those vertex u such that the path between s and u contains $t+1$ edges. Let $s \to v_1 \to \cdots \to v_t \to u$ be a shortest path from s to u. Then $s \to v_1 \to \cdots \to v_t$ must be a shortest path between s and v_t, which contains t edges. We know by the induction hypothesis that, by the end of the first t iterations, dist(v_t) equals to the distance from s to v_t, and it will be correct throughout the $(t+1)$-th iteration (by Observation 1). In addition, the distance between s and u is exactly dist(v_t) + $\ell(v_t, u)$. On the other hand, at the $(t+1)$-th iteration, the value dist(u) will be updated by taking a minimum

over a set of values including $\mathrm{dist}(v_t)+\ell(v_t,u)$. Therefore, $\mathrm{dist}(u)\le\mathrm{dist}(v_t)+\ell(v_t,u)$. Since $\mathrm{dist}(u)$ is always an upper bound to the distance from s to u and $\mathrm{dist}(v_t)+\ell(v_t,u)$ is the actual distance, we know $\mathrm{dist}(u)=\mathrm{dist}(v_t)+\ell(v_t,u)$ and $\mathrm{dist}(u)$ correctly reflects the distance from s to u. The inductive step is concluded.
Since the graph contains no negatively-weighted cycle, the maximum number of edges in a shortest path is $n-1$. Therefore, all those values of $\mathrm{dist}(u)$ are correctly set after $n-1$ iterations. □

The running time for Bellman-Ford algorithm is obviously $O(mn)$.
It remains to consider the case where the graph contains a negatively-weighted cycle. We only need to implement the Bellman-Ford algorithm again (i.e., to have another $n-1$ iterations of updates) and see if $\mathrm{dist}(u)$ has been changed during this. If the graph does not contains a negatively-weighted cycle, we know that another implementation of the Bellman-Ford algorithm will not change $\mathrm{dist}(u)$ for any vertex u (by the second observation, all the distances are corrected set after the first implementation of the Bellman-Ford algorithm; by the first observation, each $\mathrm{dist}(u)$ remains being correct). If the graph contains a negatively-weighted cycle, for a vertex u with distance $-\infty$ from s, the shortest path between s and u contains infinitely many of edges. Therefore, our second observation does not applied. In fact, we will soon see that the updates will go on indefinitely.
Let $u_0\to u_1\to\cdots\to u_{k-1}\to u_0$ be a cycle with negative weight. After any iteration of the update, we must have $\mathrm{dist}(u_{i+1})>\mathrm{dist}(u_i)+\ell(u_i,u_{i+1})$ for some $i=0,1,\ldots,k-1$. (We suppose the indices are in $\mathbb{Z}_k$, so that $(k-1)+1=0$.) Otherwise, if we have $\mathrm{dist}(u_{i+1})\le\mathrm{dist}(u_i)+\ell(u_i,u_{i+1})$ for all $i=0,1,\ldots,k-1$, we have the following by adding all those k inequalities:

$$\sum_{i=0}^{k-1}\mathrm{dist}(u_i)\le\sum_{i=0}^{k-1}\mathrm{dist}(u_i)+\sum_{i=0}^{k-1}\ell(u_i,u_{i+1}),$$

which implies

$$\sum_{i=0}^{k-1}\ell(u_i,u_{i+1})\ge 0,$$

contradicting to that the cycle has a negative weight. Therefore, we have proved that there is always an update possible after any iteration.

3.2 Directed Acyclic Graphs

A directed acyclic graph contains no cycle, so we do not need to worry about negatively-weighted cycles.

Problem 9. Given a weighted directed acyclic graph $G=(V,E,\ell)$ and a vertex s, find the distances between s and other vertices.

It turns out that we can adapt Bellman-Ford algorithm for directed acyclic graphs with time complexity $O(n+m)$, which is significantly less than $O(nm)$ for Bellman-Ford algorithm on general graphs. The adapted algorithm only updates each edge (u, v) once, provided that a correct order of the updating is given.

Definition 10. Given a set of element S, a *partial order* $\leq$ assigns each ordered pair of elements (a, b) three possible outcomes: $a \leq b$, $b \leq a$ or that a and b are incomparable, such that it satisfies the following three axioms:

- $a \leq a$ for any $a \in S$;
- If $a \leq b$ and $b \leq a$, then $a = b$;
- If $a \leq b$ and $b \leq c$, then $a \leq c$.

Naturally, for all the vertices in a directed acyclic graph $G = (V, E)$, we can define a partial order $\leq$ such that, for two vertices u and v, we have $u \leq v$ if there is a path from u to v, $v \leq u$ if there is a path from v to u, and u, v are incomparable otherwise. This is a valid partial order: it is easy to check the first and the third axioms; the second axiom follows from that the graph contains no cycle.
A partial order can always be generalized to a *total order* that *agrees with the partial order*.

Definition 11. Given a set of element S, a *total order* $\leq$ assigns each ordered pair of elements (a, b) two possible outcomes: $a \leq b$ or $b \leq a$, such that it satisfies the following three axioms:

- $a \leq a$ for any $a \in S$;
- If $a \leq b$ and $b \leq a$, then $a = b$;
- If $a \leq b$ and $b \leq c$, then $a \leq c$.

Definition 12. Given a set of elements S, a partial order $\leq_p$ and a total order $\leq_t$, the total order $\leq_t$ *agrees with the partial order* $\leq_p$ if $a \leq_p b$ implies $a \leq_t b$ for any $a, b \in S$.

In fact, a set equipped with a partial order can be visualized by a directed acyclic graph, where it is not always possible to compare two elements. A set equipped with a total order can be visualized by an array sorted by the given total order.
Given a directed acyclic graph $G = (V, E)$ (weighted or unweighted), the order in which the vertices are explored by the BFS algorithm forms a total order that agrees with the partial order defined by G. See Algorithm 4.
After we have sorted the vertices by the total order, the adapted Bellman-Ford algorithm is shown in Algorithm 5.
The correctness of Algorithm 5 can be proved similarly as it is in the previous section. In particular, the first observation is still true. The second observation is adapted to the followings: *for each vertex v_i from s, dist(v_i) correctly reflects the distance between s and v_i after the $(i-1)$-th iteration.* This can be again proved by induction.

Algorithm 4 Find a total order for a directed acyclic graph

Input: a directed acyclic graph $G = (V, E)$

Output: a total order described by the array ind[$\cdot$]

```
1: $c \leftarrow n - 1$
2: initialize ind[$u$] $\leftarrow \infty$ for each $u \in V$
3: **while** there exists $u \in V$ with ind[$u$] $= \infty$:
4:     EXPLORE($v$)        // defined below
5: **endwhile**
```

EXPLORE(u):

```
1: **for** each $v : (u, v) \in E$:
2:     EXPLORE($v$)
3: **endfor**
4: ind[$u$] $\leftarrow c$
5: $c \leftarrow c - 1$
```

Algorithm 5 Bellman-Ford algorithm adapted for directed acyclic graph

Input: a weighted directed acyclic graph $G = (V, E, \ell)$ with vertices sorted by a total order, a vertex $s \in V$

Output: the distances from s to all vertices

```
1: initialize dist($u$) $= \infty$ for all $u \in V$
2: dist($s$) $\leftarrow 0$
3: **for** $i = 0, 1, \ldots, n - 1$
4:     let $v_i$ be the $i$-th vertex in the order
5:     **for** each $u : (v_i, u) \in E$:
6:         **if** dist($u$) $>$ dist($v_i$) $+ \ell(v_i, u)$, update dist($u$) $\leftarrow$ dist($v_i$) $+ \ell(v_i, u)$
7:     **endfor**
8: **endfor**
```

For the base step, if $v_0 = s$, we have $\text{dist}(v_0) = 0$ which is correct. Otherwise, we know that there is no path from s to v_0 (otherwise v_0 should be placed after s, and should not be the first element), and $\text{dist}(v_0) = \infty$ which correctly reflects the distance.

For the inductive step, suppose we are after the i-th iteration and consider vertex v_{i+1}. Let $s \to w_1 \to \cdots \to w_k \to v_{i+1}$ be the shortest path from s to v_{i+1}. Since v_{i+1} is reachable from w_k, w_k must be placed before v_{i+1}, and $w_k = v_j$ for some $j = 1, \ldots, i$. By the induction hypothesis and the first observation, $\text{dist}(w_k)$ correctly reflects the distance between s and $w_k = v_j$. At j-th iteration, $\text{dist}(v_{i+1})$ has already been updated to $\text{dist}(v_{i+1}) = \text{dist}(w_k) + \ell(w_k, v_{i+1})$. Since $j \leq i$, by the time the i-th iteration is over, $\text{dist}(v_{i+1})$ correctly reflects the distance. This proves the inductive step. We conclude the correctness of Algorithm 5.

It is easy to check that the time complexity for Algorithm 5 is $O(m+n)$.