Lecture 16 Single-Source Shortest Paths

MCS/CS 401 Computer Algorithms I — July 25, 2022
Instructor: Danko Adrovic

Lecture Outline

- Single-Source Shortest Paths
  - Shortest-path problem
  - Optimal substructure of a shortest path
  - Negative-weight edges
  - Cycles
  - Representation of shortest paths
  - Relaxation
  - The Bellman-Ford algorithm
  - Bellman-Ford algorithm time complexity
  - Dijkstra's algorithm
  - Dijkstras algorithm time complexity

1 Single-Source Shortest Paths

Practical motivation to study single-source shortest path problems comes from the need to determine the best possible route between two locations, like two cities. Imagine a road map of the USA on which distances between each pair of adjacent cities is recorded. The question is then, given such a map, what is the shortest route between, let's say, Chicago and Houston?

One way to approach this problem is to consider all possible routes from Chicago to Houston, determine which route is the shortest, and then select that route. It is easy to see that, on such a map, enumerating all possible routes would take a long time, as most of those enumerated routes would be impractical. For example, one would not want to select a route that passes through Los Angeles to get from Chicago to Houston.

In this lecture we will show how we can solve such a problem efficiently. Our main way to represent problems of these types is going to use weighted, directed graphs in which vertices represent locations and weighted edges the distances between adjacent locations.

1.1 Shortest-path problem

A shortest path problem consist of a weighted, directed graph $G = (V, E)$, with a weight function $w : E \mapsto \mathbb{R}$ mapping edges to real-valued weights. The **weight** $w(p)$ of a path $p =< v_0, v_1, \ldots v_k >$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

We define the **shortest-path weight** $\delta(u, v)$ from u to v by

$$\delta(u,v) = \begin{cases} min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & if\ there\ is\ a\ path\ from\ u\ to\ v \\ \infty & otherwise. \end{cases}$$

A **shortest-path** from vertex u to v is then defined as any path p with weight $w(p) = \delta(u, v)$.

1.2 Optimal substructure of a shortest path

Algorithms, which solve the shortest-paths problems, typically rely on the property that a shortest path between two vertices contains other shortest paths within it. Recall that the notion of optimal substructure is one of the key indicators that dynamic programming and the greedy method might be applicable.

Lemma 1 (Subpaths of shortest paths are shortest paths)**.** *Given a weighted graph $G = (V, E)$ with weight function $w : E \mapsto \mathbb{R}$, let $p =< v_0, v_1, \ldots v_k >$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} =< v_i, v_{i+1}, \ldots v_j >$ be a subpath of p from any vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.*

Proof. If we break-up the path p into components $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k\}$, then we have the combined weight $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Assume now that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k\}$ is a path from v_0 to v_k whose weight $w(p) = w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. □

1.3 Negative-weight edges

Some instances of the single-source shortest-paths problem may include edges whose weights are negative. Such instances can lead to two **problematic spots** when combined with cycles, which we call **negative cycle**: sum of edge weights on the cycle is negative, and **positive cycle**:sum of edge weights on the cycle is positive. The instance of the **zero cycle**, where the sum of edge weights on the cycle is zero, is not problematic, as explained below in section "Cycles".

If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value. If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path; we can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$. Positive-weight cycle has a similar effect (see section "Cycles" below), only we do not have a symbol for it. In particular, we will see that the **Bellman-Ford** algorithm can detect only the **negative-weight** cycle on some path from s to v, which we label $\delta(s, v) = -\infty$

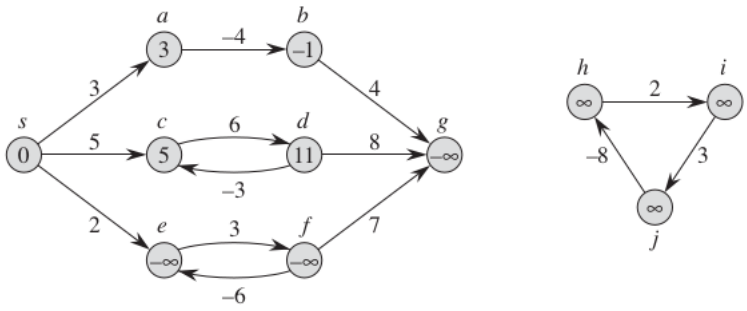


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

The effect of negative/positive-weight cycles is illustrated in the figure above: bottom path shows the problematic negative cycle. The middle path shows the problematic positive cycle. In both instances, the path is ill-defined as the cycling causes an indefinite, path-weight updating process.

1.4 Cycles

Can a shortest path contain a cycle? As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight. That is, if $p =< v_0, v_1, \ldots v_k >$ is a path and $c =< v_i, v_{i+1}, \ldots v_j >$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the path $p' =< v_0, v_1, \ldots v_i, v_{j+1}, v_{j+2}, ..., v_k >$ has weight $w(p') = w(p) - w(c) < w(p)$, and so p cannot be a shortest path from v_0 to v_k.

That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same. Thus, if there is a shortest path from a source vertex s to a destination vertex v that contains a 0-weight cycle, then there is another shortest path from s to v without this cycle. As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free. Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Therefore, we can restrict our attention to shortest paths of at most $|V| - 1$ edges.

1.5 Representation of shortest paths

We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. We represent shortest paths similarly to how we represented breadth-first trees form the previous sections. Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$ a **predecessor** $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this

lecture set the π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v.

During the execution of a shortest-path algorithm, the π values might not indicate shortest paths. In the same way as in the breadth-first search, we will be interested in the **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ induced by the π values, whose components we define as $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$ and $E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$.

The main reason for addressing the representation of shortest paths is that we can prove that the π values generated by the shortest-path algorithms have the property that, when they terminate, G_π is a *shortest path tree*; that is, G_π is a rooted tree containing a shortest path from the source s to every vertex that is reachable from s. These trees, whose properties we define below, are very much like the tree generated by the breadth-first search algorithm, which we termed breadth-first trees.

A **shortest-paths tree** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Note, however, that the shortest-paths are not necessarily unique, and neither are the shortest-path trees, as can be seen in the figure below.

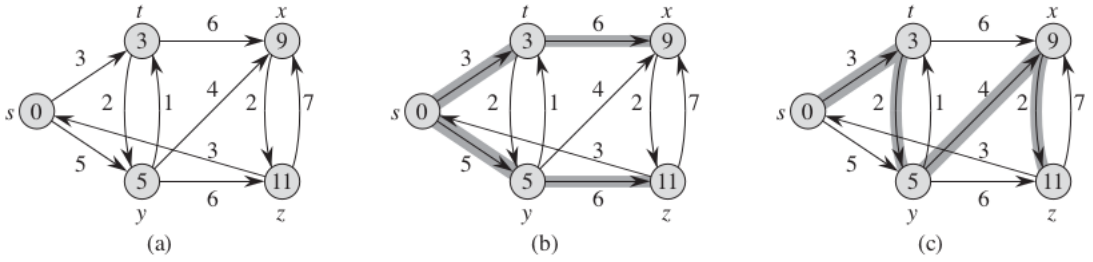


Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

1.6 Relaxation

The idea, or in this specific case technique, of **relaxation** is a very common one in algorithms dealing with *optimization*. For each vertex $v \in V$, we maintain an attribute $v.d$ which is an upper bound on the weight of a shortest path from a source s to vertex v. We call $v.d$ a **shortest-path estimate**, which are initialized along with predecessors by the following $\Theta(V)$-time procedure:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Once initialized, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$. A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The following code performs a relaxation step on edge (u, v) in time $O(1)$:

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

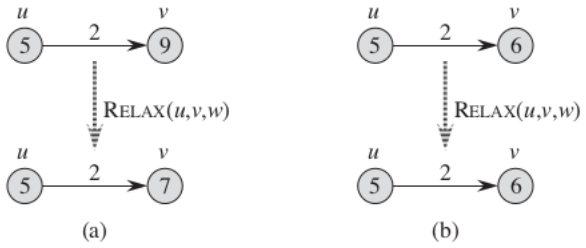


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

Think of the "relaxation" as if the algorithm had reached v in a prior step, now after investigating approach to v via new vertex u, a better route is detected, and so the value $v.d$ decreases to $u.d + w(u, v)$ value; this is what the figure (a) above describes. Figure (b) describes the situation where no change of $v.d$ occurs.

The following properties of shortest paths and relaxation play an instrumental role in the construction and correctness of the algorithms in this lecture. The Lemma/Corollary numbers refer to their location in our book.

Properties of shortest paths and relaxation

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

1.7 The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \mapsto \mathbb{R}$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

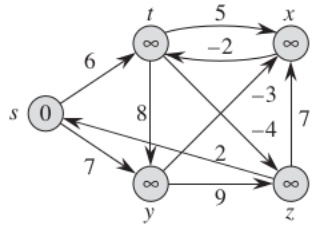
The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

As an example of the Bellman-Ford algorithm, consider the graph figure below as an input. The table below illustrates all the computed values (edge relaxations) during the running of the algorithm. For consistency, alphabetical order was used in the vertex selection - that is, we used this vertex to explore its outgoing edges. The algorithm performs $|V|-1 = 5-1 = 4$ passes in the **for** loop on line 2 in the algorithm.

Equivalently, random selection of vertices (and their outgoing edges) can be used, without affecting the final result, although intermediately computed values may be different.

In this specific case, algorithm returns TRUE as there are no *negative cycles*. The lines $5-8$ in the Bellman-Ford algorithm check for the negative cycles.



Bellman-Ford Algorithm Illustration						
vertex selected	s	t	x	y	z	pass
-	0	∞	∞	∞	∞	-
s	0	6	∞	7	∞	1
t	0	6	11	7	2	1
x	0	6	11	7	2	1
y	0	6	4	7	2	1
z	0	6	4	7	2	1
s	0	6	4	7	2	2
t	0	6	4	7	2	2
x	0	2	4	7	2	2
y	0	2	4	7	2	2
z	0	2	4	7	2	2
s	0	2	4	7	2	3
t	0	2	4	7	-2	3
x	0	2	4	7	-2	3
y	0	2	4	7	-2	3
z	0	2	4	7	-2	3
s	0	2	4	7	2	4
t	0	2	4	7	-2	4
x	0	2	4	7	-2	4
y	0	2	4	7	-2	4
z	0	2	4	7	-2	4

The final graph picture looks like the picture in the original graph above, only with infinity symbols ∞ replaced with the values in the last row of the table.

1.8 Bellman-Ford algorithm time complexity

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization on line 1 takes time $\Theta(V)$, each of the $|V| - 1$ passes over the edges in lines $2 - 4$ takes time $\Theta(E)$, and the **for** loop on lines $5 - 7$ takes $O(E)$ time.

1.9 Dijkstra's algorithm

Dijkstras algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are non-negative. Therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstras algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstras algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

It is important to note that Dijkstra's algorithm is based on the **breadth-first search**, as was Prim's algorithm from the previous lecture. Recalling this fact will help us understand the algorithm better. From its definition, we can see that Dijkstra's algorithm is a **greedy** algorithm as it extracts edges that are of minimal weight.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

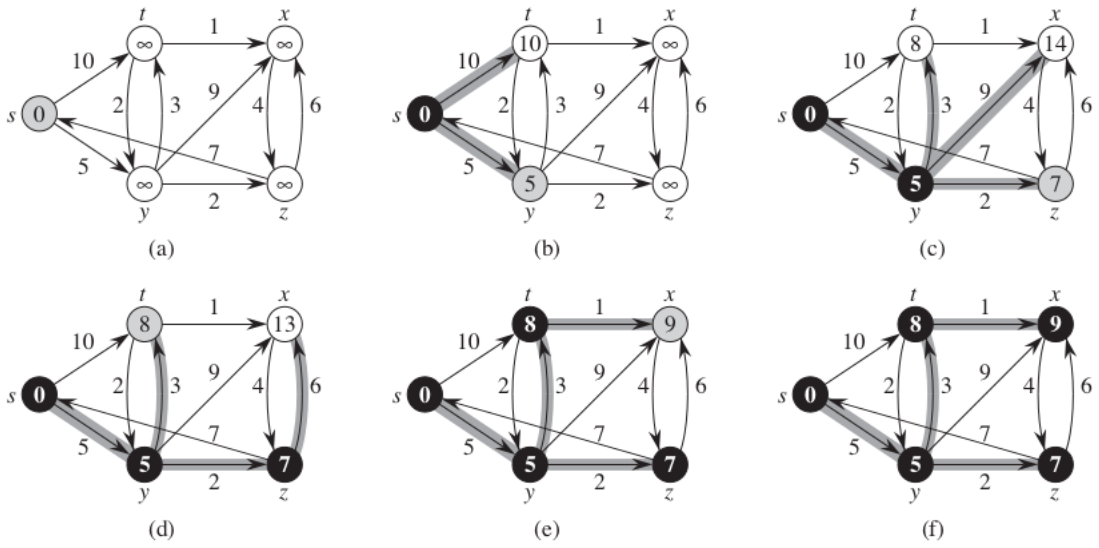


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

1.10 Dijkstra's algorithm time complexity

The running time of the algorithm depends on the data structure used to implement the min-priority queue. When binary min-heap is used, the running time of the algorithm is $O((E + V)lg(V))$.