CSCI 470: Single-Source Shortest Paths, Dijkstra's algorithm, Bellman-Ford algorithm

Vijay Chaudhary

Nov 06, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. Shortest-paths problems

2. Dijkstra's algorithm

3. Bellman-Ford algorithm

Shortest-paths problems

Shortest-paths problems

- In a ***shortest-paths problem***, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights.
- The ***weight*** $w(p)$ of path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the weights of its constituent edges:

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

Shortest-paths problems

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

$$\delta(u,v) = \begin{cases} min\{w(p) : u \rightsquigarrow v\} & \exists \text{ a path } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

- A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.
- Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity than accumulates linearly along a path and that we would want to minimize.

Shortest Paths: Figure

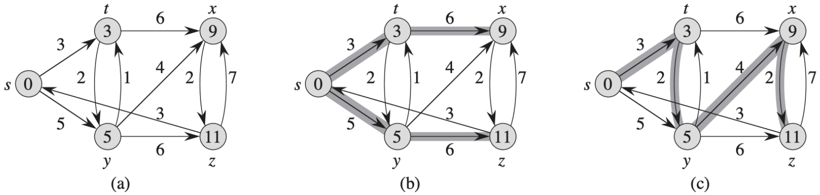


Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Negative-weight edges

- 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 $\delta(s, v) = -\infty$.

Figure: negative-weight edges

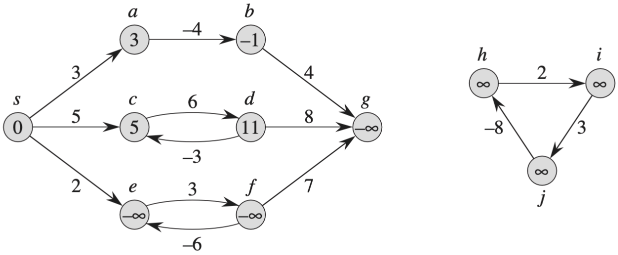


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path 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.

Cycles

- Can a shortest path contain a cycle?

Cycles

- Can a shortest path contain a cycle?
- They cannot contain negative-weight cycles as we saw earlier. How about positive-weight cycles?

Cycles

- Can a shortest path contain a cycle?
- They cannot contain negative-weight cycles as we saw earlier. How about positive-weight cycles?
- It cannot 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.

Cycles

- Can a shortest path contain a cycle?
- They cannot contain negative-weight cycles as we saw earlier. How about positive-weight cycles?
- It cannot 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.
- How about 0-weight cycles?

Cycles

- Can a shortest path contain a cycle?
- They cannot contain negative-weight cycles as we saw earlier. How about positive-weight cycles?
- It cannot 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.
- How about 0-weight cycles?
- We can always remove a 0-weight cycle to produce an alternative path whose path weight will still be the same.

Cycles: II

- 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. Thus, we can restrict our attention to shortest paths of at most $|V| - 1$ edges.

Relaxation

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

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

Relax

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Relaxation: Figure

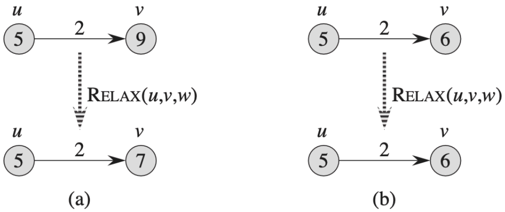


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation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

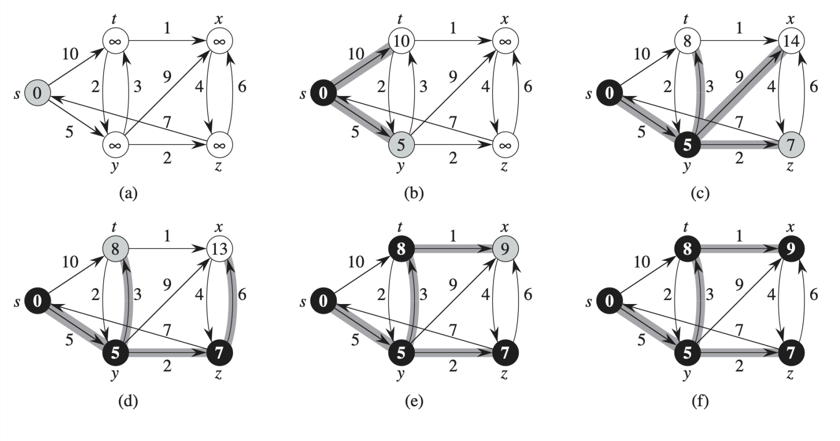
Dijkstra’s algorithm

Dijkstra's algorithm

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
> $u =$ EXTRACT-MIN(Q)
> $S = S \cup \{u\}$
> **for** each vertex $v \in G.Adj[u]$
> > RELAX(u, v, w)

Dijkstra: Figure



Runtime analysis

- There are three priority-queue operations involved here: INSERT (line 3), EXTRACT-MIN (line 5), DECREASE-KEY (implicit in RELAX, which is called in line 8).
- The algorithm calls both INSERT and EXTRACT-MIN once per vertex.
- Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the **for** loop of lines 7-8 exactly once during the course of algorithm.
- Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus DECREASE-KEY gets called at most $|E|$ times overall.

Runtime analysis: II

If we implement priorty-queue as an array, without a min-heap counterpart.

- Consider a case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$.
- We simply store $v.d$ in the vth entry of an array.
- Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN takes $O(V)$ time.
- This amounts to $O(V^2 + E) = O(V^2)$.

Runtime analysis: III

If we implement priorty-queue with a binary min-heap.

- The time to build the binary min-heap is $O(V)$.
- Each EXTRACT-MIN takes $O(\lg(V))$ time, and there are $|V|$ such operations.
- Each DECREASE-KEY takes $O(\lg(V))$ time, and there are $|E|$ such operations.
- The total running time is therefore $O(V \lg V + E \lg V) = O((V + E) \lg V)$.

Bellman-Ford algorithm

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 $w : E \to \mathbb{R}$, 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 $\delta(s, v)$.

Bellman-Ford algorithm: pseudocode

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

Bellman-Ford: illustration

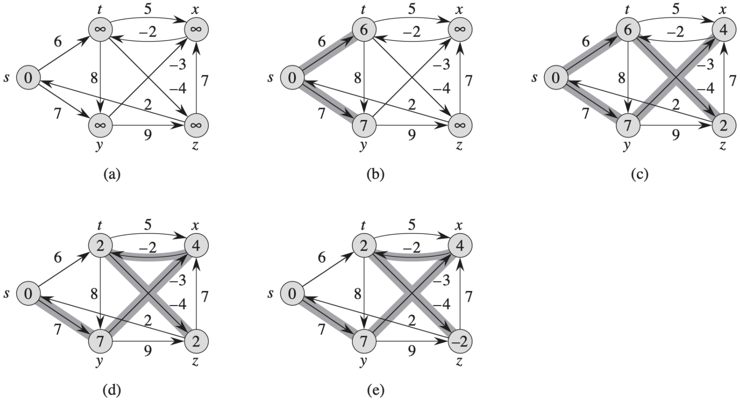


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Bellman-Ford runtime

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