Design and Analysis of Algorithms

Part 6

Paths in Graphs

Elias Koutsoupias
with thanks to Giulio Chiribella

Hilary Term 2021
How to find the optimal route?

In many applications, like GoogleMaps, it is important to find the shortest route from one location to another. More generally, going from one location to another could have a cost and it is important to find the route that minimizes the cost.

**Graph formulation:**

- locations $\rightarrow$ vertices
- connections between two locations $\rightarrow$ edges
- cost of going from one location to another $\rightarrow$ weight on the edge
- route $\rightarrow$ path
- cost of a route $\rightarrow$ sum of the weights of the edges on the path.
Shortest paths

Consider a directed graph $G = (V, E)$ with weight function (or length) $w : E \rightarrow \mathbb{R}_{\geq 0}$.

- The weight (or length) of a path $p = \langle v_0, v_1, \cdots, v_k \rangle$ is
  
  \[ w(p) := \sum_{i=1}^{k} w(v_{i-1}, v_i) \]

  I.e. $w(p)$ is the sum of edge weights on path $p$.

- The shortest-path weight (or length) from vertex $u$ to vertex $v$ is
  
  \[ \delta(u, v) := \left\{ \begin{array}{ll}
  \min \{ w(p) : p \text{ is a path from } u \text{ to } v \} & \text{if } \exists \text{ path from } u \text{ to } v \\
  \infty & \text{otherwise}
  \end{array} \right. \]

- A shortest path from $u$ to $v$ is a path $p$ such that $w(p) = \delta(u, v)$.
Consider the simplest case: \textit{all weights equal to 1.}

\[ \delta(u, v) = \text{minimum number of edges on a path from } u \text{ to } v, \text{ if such a path exists; otherwise, } \delta(u, v) = \infty. \]

Given a source vertex \( s \), \textbf{breadth-first search (BFS)} finds all the vertices reachable from \( s \) and, for each reachable vertex \( v \) it finds the shortest-path length \( \delta(s, v) \) and a shortest path from \( s \) to \( v \).

\textbf{Input:} A graph \( G = (V, E) \), either directed or undirected, and a source vertex \( s \in V \).

\textbf{Output:} For each \( v \in V \), an integer \( d[v] \) and a back pointer \( \pi[v] \), such that

1. \( d[v] = \delta(s, v) \)
2. \( \pi[v] = u \) is the predecessor of \( v \) on a shortest path from \( s \) to \( v \).
   If there is no path from \( s \) to \( v \), then \( \pi[v] = \text{NIL} \).
DFS (left) and BFS (right) on a tree, starting at vertex 1. The labels indicate the order by which the vertices are discovered.
**Intuitive idea**

**Idea**: Send a wave out from the source $s$.

- The wave first hits all vertices 1 edge away from $s$.
- From there, the wave will then hit all vertices 2 edges away from $s$.
- When the wave reaches $v$, we know the distance from $s$ to $v$.

The shortest path is constructed by keeping track of the vertices reached by the wave on the way to $v$.

**Data structure implementation**: the wave can be implemented by a first-in-first-out (FIFO) queue $Q$.

A vertex $v$ is put in the queue when it is reached by the wave, and it is removed by the queue after all its neighbours are reached.
A **FIFO queue** is an *abstract data structure* with three basic operations:

1. \texttt{ENQUEUE}(Q, x): Inserts \( x \) at the *end* of the queue \( Q \).
2. \texttt{DEQUEUE}(Q): Returns and removes the item at the *head* of the queue \( Q \).
3. \texttt{ISEMPTY}(Q): Returns whether or not the queue is empty.

In the pseudo code we will use the notation \( Q = \emptyset \) and \( Q \neq \emptyset \).

**Implementation:**

Linked list with an extra “END” pointer to the tail of the list.
The head of the queue corresponds to the head of the linked list.
This makes all operations \( O(1) \).
The BFS algorithm [CLRS 22.2]

BFS\((V, E, s)\)

**Input:** A directed or undirected graph \((V, E)\), and a source \(s \in V\).

**Output:** For each \(v \in V\), \(d[v]\) and \(\pi[v]\)

1. \(d[s] = 0; \pi[s] = nil\) \hspace{1cm} // Source can be reached with path of length 0
2. **for** each \(u \in V - \{s\}\) \hspace{1cm} // Mark all other nodes
3. \(d[u] = \infty\) \hspace{1cm} // as not reached yet
4. \(\pi[u] = nil\)
5. \(Q = \emptyset\) \hspace{1cm} // Initialise \(Q\)
6. **ENQUEUE**\((Q, s)\) \hspace{1cm} // to contain only source \(s\)
7. **while** \(Q \neq \emptyset\)
8. \(u = \text{DEQUEUE}(Q)\) \hspace{1cm} // Node \(u\) is finished
9. **for** each \(v \in Adj[u]\)
10. **if** \(d[v] = \infty\) \hspace{1cm} // If \(v\) is reached for the first time.
11. \(d[v] = d[u] + 1\) \hspace{1cm} // record shortest distance,
12. \(\pi[v] = u\) \hspace{1cm} // create backpointer,
13. **ENQUEUE**\((Q, v)\) \hspace{1cm} // and join queue of unfinished nodes
Running time

BFS takes $O(|V| + |E|)$ in total.

- $O(|V|)$ because of the initialization step and of the enqueue/dequeue operations (each vertex is enqueued/dequeued at most once)
- $O(|E|)$ because we examine the edge $(u, v)$ only when $u$ is dequeued. Hence every edge is examined at most once (or at most twice if the graph is undirected and we represent edges as unordered pairs $\{u, v\}$).
Example

Consider the following graph with source $s = A$.

First, the wave hits vertices $\{B, C, F\}$.
Second, it hits $\{E, D, G\}$.
Third, it hits $\{H\}$.
At the fourth step, the wave does not discover any new nodes.
The BFS tree

The **BFS tree** consists of the vertices reachable from \( s \) and has an edge between \( u \) and \( v \) iff \( u = \pi[v] \).

Explicitly, the BFS tree is the graph \( G_{\pi} := (V_{\pi}, E_{\pi}) \) with
\[
V_{\pi} := \{ u \in V : \pi[u] \neq \text{NIL} \} \cup \{ s \} \quad \text{and} \quad E_{\pi} := \{ (\pi[u], u) : u \in V_{\pi} \}.
\]

* note the usual abuse of notation: in math, trees are **undirected graphs**.

**Example (cont’d)**

![BFS Tree Diagram]

**Note:** \( d[v] \) is the level of \( v \) in the BFS tree.

**Note:** the BFS tree depends on the order in which adjacent nodes are explored, for example, if \( E \) came before \( D \), the edge \((E, H)\) would be in the tree instead of \((D, H)\).
**Lower bound on** \(d[v]\)

**Lemma 1**  If \(d[v] < \infty\), there exists a path of length \(d[v]\) from \(s\) to \(v\).

**Proof.** By induction on \(d[v]\).

*Base case.* If \(d[v] = 0\), then \(v\) must be the source \(s\). Hence, Lemma 1 trivially holds.

*Induction step.* Suppose that, for every \(v\) with \(d[v] \leq d_0\), there exists a path of length \(d[v]\) from \(s\) to \(v\). Suppose that \(v\) is such that \(d[v] = d_0 + 1\). Then, let \(u = \pi[v]\) be the predecessor of \(v\). By construction \(d[v] = d[u] + 1\), and therefore \(d[u] = d_0\).

Now, the induction hypothesis guarantees that there exists a path of length \(d[u]\) from \(s\) to \(u\). Adding the edge \((u, v)\) to this path, we obtain a path of length \(d[u] + 1 = d[v]\) from \(s\) to \(v\). \(\square\)

**Corollary 1 (lower bound on** \(d\)).**  \(d[v] \geq \delta(s, v)\) for every vertex \(v\).
Properties of the queue

**Lemma 2** If \( u \) is enqueued before \( v \), then \( d[u] \leq d[v] \). Furthermore, if \( Q = \langle v_1, \ldots, v_r \rangle \) is the queue at a given step of BFS, then

\[
d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1] + 1.
\]

**Proof.** By induction on the number of \texttt{ENQUEUE}/\texttt{DEQUEUE} operations.

*Base case.* For the base case, only vertex \( s \) is been enqueued, and Lemma 2 trivially holds.

*Induction step.* When a vertex \( u \) is dequeued, by induction, every vertex \( v \) in the queue has value \( d[v] \in \{d[u], d[u] + 1\} \). And when a neighbor \( v_r \) of \( u \) is enqueued, it gets a value \( d[v_r] = d[u] + 1 \).

This shows that

- \( d[v_r] \geq d[v] \) for every vertex \( v \) enqueued before \( v_r \),
- the values in the queue differ by at most 1.
Upper bound on $d[v]$

**Lemma 3 (upper bound on $d$).** If there exists a path of length $l$ from $s$ to $v$, then $d[v] \leq l$.

**Proof.** By induction on $l$. *Base cases.* For $l = 0$, $v = s$ and $d[s] = 0$. For $l = 1$, the path is an edge $(s, v)$, and therefore $d[v] = 1$.

*Induction step.* Suppose that Lemma 3 holds for paths of length $l \leq l_0$, and suppose that $\langle s, v_1, \ldots, v_{l_0}, v \rangle$ is a path of length $l_0 + 1$ from $s$ to $v$. By the induction hypothesis, $d[v_{l_0}] \leq l_0$. Now, there are three possibilities:

1. $v$ is enqueued before $v_{l_0}$ is enqueued. Then, Lemma 2 implies $d[v] \leq d[v_{l_0}] \leq l_0$.
2. $v$ is enqueued before $v_{l_0}$ is dequeued. Then, Lemma 2 implies $d[v] \leq d[v_{l_0}] + 1 \leq l_0 + 1$.
3. $v$ is enqueued after $v_{l_0}$ is dequeued. Then, $\pi[v] = v_{l_0}$ and $d[v] = d[v_{l_0}] + 1 = l_0 + 1$.

$\square$
Theorem 1. \( d[v] = \delta(s, v) \) for every vertex \( v \in V \).

Proof. Suppose that \( \delta(s, v) \) is finite. The theorem follows directly from the lemmas.

Suppose that \( \delta(s, v) \) is infinite. Then, Corollary 1 implies \( d[v] \geq \delta(s, v) = \infty \).
Correctness of BFS (cont’d)

**Theorem 2.** If $0 < d[v] < \infty$, then $\pi[v]$ is the predecessor of $v$ on a shortest path from $s$ to $v$.

**Proof.** By induction on $d[v]$. *Base case.* For $d[v] = 1$, $(s, v)$ is an edge in $E$ and, by construction, $\pi[v] = s$.

*Induction step.* Suppose that Theorem 2 holds for every vertex $v$ such that $d[v] \leq l_0$. For a vertex $w$ with $d[w] = l_0 + 1$, define $v_{l_0} := \pi[w]$. By construction, $d[v_{l_0}] = d[w] - 1 = (l_0 + 1) - 1 = l_0$. Hence, Lemma 1 implies that there exists a path $\langle s, v_1, \ldots, v_{l_0} \rangle$. Adding the edge $(v_{l_0}, w)$, we obtain the path $p = \langle s, v_1, \ldots, v_{l_0}, w \rangle$. The length of the path is $l_0 + 1 = d[w]$. Hence, Theorem 1 implies that $p$ is a shortest path from $s$ to $w$. The predecessor of $w$ in $p$ is $v_{l_0} = \pi[w]$. □
DFS vs BFS [DPV 4.2]

- DFS goes “as far as possible” (depth), while BFS discovers all the vertices at a certain distance before moving to further vertices. In terms of data structures, this is reflected in the fact that DFS uses (implicitly) a last-in-first-out queue (i.e. a stack), while BFS uses a first-in-first-out queue.

- Unlike DFS, BFS may not reach all vertices: BFS discovers only the vertices that are reachable from the source $s$.

- Both algorithms have linear running time $O(|V| + |E|)$. 
Dijkstra’s Algorithm [CLRS 24.3]

- Dijkstra’s algorithm solves the Single-Source Shortest Path Problem for *non-negative weights*.
- It is essentially a weighted version of breath-first search.
- The algorithm uses a *min-priority queue* $Q$ (instead of FIFO queue), with keys given by the *shortest-path weight estimates* $d[v]$.
- At termination,
  - $d[v]$ is equal to the distance from $s$ to $v$
  - the algorithm provides a back-pointer $\pi[v]$, such that $\pi[v]$ is the predecessor of $v$ on a shortest path from $s$ to $v$, if such path exists, or $\pi[v] = \text{NIL}$ otherwise.
Dijkstra\((V, E, w, s)\)

**Input:** A directed or undirected graph \((V, E)\), \(s \in V\), \(w : E \rightarrow \mathbb{R}_{\geq 0}\).

**Output:** For each \(v \in V\), \(d[v]\) and \(\pi[v]\)

1. **for** each \(v \in V\)
2. \(d[v] = \infty\)
3. \(\pi[v] = \text{nil}\)
4. \(d[s] = 0\)
5. \(Q = \text{MAKE-QUEUE}(V)\) with \(d[v]\) as keys
6. **while** \(Q \neq \emptyset\)
7. \(u = \text{EXTRACT-MIN}(Q)\) \hspace{1cm} // Shortest route to \(u\) known
8. **for** each vertex \(v \in \text{Adj}[u]\)
9. \(\text{if } d[u] + w(u, v) < d[v]\) \hspace{1cm} // Shorter route to \(v\) via \(u\) discovered
10. \(d[v] = d[u] + w(u, v)\)
11. \(\pi[v] = u\)
12. \(\text{DECREASE-KEY}(Q, v, d[v])\)
Example

\[
\begin{array}{cccccc}
\text{n-th iteration} & \text{Init} & 1 & 2 & 3 & 4 & 5 \\
\text{EXTRACT-MIN}(Q) (source) \ d[A] & 0 & A & 0 & C & 0 & B \\
\d & 0 & 4 & 0 & 3 & 0 & 3 \\
\d[B] & \infty & 2 & \infty & 2 & \infty & 2 \\
\d[C] & \infty & \infty & 6 & \infty & 5 & \infty \\
\d[D] & \infty & \infty & 7 & \infty & 4 & \infty \\
\d[E] & \infty & \infty & 4 & \infty & 4 & \infty \\
Q \langle A, B, C, D, E \rangle & \langle C, B, D, E \rangle & \langle B, D, E \rangle & \langle E, D \rangle & \langle D \rangle & \langle \rangle \\
\end{array}
\]
Correctness

Loop invariants for the while loop:
I1. For all \( v \in V \), we have \( d[v] \geq \delta(s, v) \).
I2. For all \( v \in S := V \setminus Q \), we have \( d[v] = \delta(s, v) \).

Initialisation:
I1. Just before the start of the while loop, \( d[s] = 0 = \delta(s, s) \)
and \( d[v] = \infty \) for every other vertex \( v \in V, v \neq s \).
   Hence I1 holds.
I2. Before the start of the while loop \( S = \emptyset \), so I2 is vacuously true.

Termination: At the end, \( S = V \).
Hence, I2 implies that \( d[v] = \delta(s, v) \) for all \( v \in V \), as required.
Maintenance of I1: \( d[v] \geq \delta(s, v) \ \forall v \in V \)

Suppose that I1 holds before an iteration of the while loop. We have to show that I1 still holds after the iteration.

For a generic vertex \( v \in V \), there are two possibilities: either \( d[v] \) did not change after the iteration, or \( d[v] \) changed.

1. If \( d[v] \) did not change, then \( d[v] \geq \delta(s, v) \) by I1 before the iteration.

2. If \( d[v] \) changed, let \( u \) be the vertex selected by EXTRACT-MIN in this iteration of the loop. Then, we have the inequality

\[
\begin{align*}
  d[v] &= d[u] + w(u, v) \\
  &\geq \delta(s, u) + w(u, v) \quad \text{by I1 before the iteration} \\
  &\geq \delta(s, v),
\end{align*}
\]

where the last inequality holds because \( \delta(s, u) + w(u, v) \) is the length of a path from \( s \) to \( v \).

\[ \square \]

**Terminology:** \( \delta(s, v) \leq \delta(s, u) + w(u, v) \) is called the triangle inequality.
Suppose that I2 holds before an iteration of the **while** loop. We have to show that I2 still holds after the iteration.

For a generic vertex $v \in S$ there are three possibilities:

1. $v$ was in $S$ before the iteration of the **while** loop.
   Then, $d[v] = \delta(s, v)$ by I2 before the iteration.

2. $v$ has been added to $S$ in this iteration *and* there is no path from $s$ to $v$.
   Then, the $d$-value is $d[v] \geq \delta(s, v) = \infty$, namely $d[v] = \infty$.

3. $v$ has been added to $S$ in this iteration *and* there is a path from $s$ to $v$.
   Then, there is also a *shortest* path $s \xrightarrow{p} v$.

Just before $v$ is added to $S$, the path $p$ connects a vertex in $S$ (the source $s$) with a vertex outside $S$ (vertex $v$). Let $y$ be the first vertex along $p$ that is outside $S$, and let $x$ be the predecessor of $y$ (by definition, $x$ is in $S$).

...picture on the next slide.
Maintenance of I2: \( d[v] = \delta(s, v) \), \( \forall v \in S \) (cont’d)

Optimal substructure: since \( s \xrightarrow{p_1} x \xrightarrow{p_2} v \) is a shortest path to \( v \), 
\[ s \xrightarrow{p_1} x \xrightarrow{p_2} y \] 
is a shortest path to \( y \).

Since \( x \) was in \( S \) before the iteration of the while loop, 
we have \( d[x] = \delta(s, x) \) by I2 before the iteration. 
Moreover, when \( x \) was extracted from \( Q \), 
the \( d \)-value of \( y \) must have been updated by lines 10-11 of the code. 
...continue on the next slide.
The convergence property

Lemma 1 (Convergence property).

Suppose \( s \xrightarrow{p_1} x \rightarrow y \) is a shortest path.
If \( d[x] = \delta(s, x) \) before lines 10-11 are executed on vertex \( y \),
then \( d[y] = \delta(s, y) \) after lines 10-11 are executed on vertex \( y \).

After executing lines 10 and 11, we have

\[
\begin{align*}
    d[y] & \leq d[x] + w(x, y) \quad \text{by lines 10-11} \\
    &= \delta(s, x) + w(x, y) \quad \text{by hypothesis} \\
    &= \delta(s, y) \quad \text{because } s \xrightarrow{p_1} u \rightarrow v \text{ is a shortest path}
\end{align*}
\]

By I1, we have \( d[y] \geq \delta(s, y) \).
Hence \( d[y] = \delta(s, y) \). \qed
Maintenance of I2: \( d[v] = \delta(s, v), \ \forall v \in S \) (cont’d)

The Convergence Property implies \( d[y] = \delta(s, y) \) before \( v \) is extracted from \( Q \).

Since \( y \) and \( v \) were both in \( Q \) when \( v \) was chosen by \textsc{Extract-Min}, we have \( d[v] \leq d[y] = \delta(s, y) \leq \delta(s, v) \).

In conclusion, we obtained \( d[v] \leq \delta(s, v) \).

Since \( d[v] \geq \delta(s, v) \) by I1, we obtained \( d[v] = \delta(s, v) \).

This proves the validity of I2 after the iteration of the \textbf{while} loop. \( \square \)
Constructing shortest paths

**Theorem.** For $v \neq s$ and $d[v] < \infty$, $\pi[v]$ is the predecessor of $v$ on a shortest path from $s$ to $v$.

**Proof.** Let $u := \pi[v]$, and let $s \xrightarrow{p} u$ be a shortest path from $s$ to $u$. At termination, we have

\[
\delta(s, v) = d[v] \quad \text{by I2}
\]
\[
= d[u] + w(u, v) \quad \text{because } u = \pi[v]
\]
\[
= \delta(s, u) + w(u, v) \quad \text{by I2} \tag{1}
\]

Hence $s \xrightarrow{p} u \rightarrow v$ is a shortest path from $s$ to $v$. \qed
Running time

The running time of Dijkstra’s algorithm depends on the implementation of the min-priority queue. For a min-heap implementation, one has

1. **MAKE-QUEUE**: $O(|V|)$
2. **EXTRACT-MIN**: $O(\log |V|)$
3. **DECREASE-KEY**: $O(\log |V|)$

**MAKE-QUEUE** is executed once.

**EXTRACT-MIN** is executed $|V|$ times.

Since each vertex $v \in V$ is added to $S$ exactly once, each edge in $\text{Adj}[v]$ is examined in the for loop exactly once.

Thus there are a total of $|E|$ iterations of the for loop, and hence a total of at most $|E|$ **DECREASE-KEY** operations.

**Total running time**:

$$O(|V|) + O(|V| \log |V|) + O(|E|) + O(|E| \log |V|)$$

$$= O((|V| + |E|) \log |V|)$$
So far, we assumed that the weight function $w : E \to \mathbb{R}$ was non-negative, namely that $w(u, v) \geq 0$ for every edge $(u, v) \in E$.

How about graphs where some of the weights are negative?

**Problem: negative-weight cycles**

$\langle B, C, B \rangle$ is a negative-weight cycle: $w(B, C) + w(C, B) = -2$.

There is no shortest path from source $A$ in this graph!
**General graphs: Bellman-Ford Algorithm [CLRS 24.1]**

**Bellman-Ford** $(V, E, w, s)$

**Input:** A directed or undirected graph $(V, E)$, $s \in V$, $w : E \rightarrow \mathbb{R}$.

**Output:** FALSE, if there exist a negative-weight cycle reachable from $s$, otherwise TRUE, and for each $v \in V$, $d[v]$ and $\pi[v]$.

1. for each $v \in V$
   2. $d[v] = \infty$; $\pi[v] = nil$
   3. $d[s] = 0$
   4. for $i = 1$ to $|V| - 1$
      // correctly computes the distances
   5. for each edge $(u, v) \in E$
      // if there is no negative-weight cycle reachable from $s$
   6. if $d[u] + w(u, v) < d[v]$
      // cycle reachable from $s$
   7. $d[v] = d[u] + w(u, v)$; $\pi[v] = u$
   8. for each edge $(u, v) \in E$
      // checks for negative-weight cycles
   9. if $d[u] + w(u, v) < d[v]$
      // reachable from $s$
   10. return FALSE
   11. else return TRUE
Initialisation takes $\Theta(|V|)$.

Each of the $|V| - 1$ iterations of lines 5–7 takes $\Theta(|E|)$.

The for loop of lines 8-11 takes $O(|E|)$.

In total, the running time is $\Theta(|V||E|)$. 
Correctness: the case of no negative-weight cycles

If no negative-weight cycle is reachable from \( s \), then the shortest paths are well-defined.

**Invariant of the for loop 4-7.** Before the \( i \)-th iteration of the loop,

\[ I_1 \quad d[v] \geq \delta(s, v), \forall v \in V \]

\[ I_2 \quad \text{for every shortest path } \langle v_0, v_1, \ldots, v_k \rangle \text{ from } v_0 = s \text{ to } v_k, \]
\[ d[v_j] = \delta(s, v_j), \forall j < i \]

**Initialisation.** Before the first iteration, \( d[s] = 0 = \delta(s, s) \), and
\( d[v] = \infty \geq \delta(s, v) \) for every \( v \in V, v \neq s \). Hence, \( I_1 \) holds.
Moreover, \( d[v_0] = d[s] = 0 = \delta(s, s) \). Hence, \( I_2 \) holds.

**Termination.** Termination occurs for \( i = |V| \).
Then, \( I_2 \) implies \( d[v_j] = \delta(s, v_j) \) for all \( j < |V| \). Since there are \( |V| \) vertices in the graph, one has \( k < |V| \), and therefore \( d[v_j] = \delta(s, v_j) \) for all \( j < k \). Hence, if there is a path from \( s \) to \( v \), \( d[v] = \delta(s, v) \). If there is no path, \( \delta(s, v) = \infty \) and \( d[v] \geq \delta(s, v) \) by \( I_1 \), whence \( d[v] = \delta(s, v) \).
Fact: I1 is maintained also by the internal for loop (lines 5-7).

Indeed, consider the iteration of the internal for loop on the edge \((u, v)\)

- Only the \(d\)-value of \(v\) can change in this iteration. For all the other \(d\)-values, I1 still holds.
- If \(v\) did not change, then I1 still holds.
- If \(d[v]\) changed, then \(d[v] = d[u] + w(u, v)\).

Hence, one has

\[
d[v] = d[u] + w(u, v) \\
\geq \delta(s, u) + w(u, v) \quad \text{by I1 before the iteration} \\
\geq \delta(s, v).
\]

Since I1 is maintained by the internal for loop 5-7, it is also maintained by the external for loop 4-7.
**Maintenance of I2:** \( d[v_j] = \delta(s, v_j), \forall j < i \)

Assume that I2 holds before the \( i \)-th iteration of the **for** loop 4-7.

**Fact:** For every \( v \in V \), the \( d \)-value \( d[v] \) does not increase after each iteration of the internal **for** loop 5-7 (and therefore it does not increase after each iteration of the external **for** loop 4-7)

For every \( j < i \), the new \( d \)-value \( d'[v_j] \) is no larger than the old \( d \)-value \( d[v_j] = \delta(s, v_j) \). By I1, this implies \( d'[v_j] = \delta(s, v_j) \).

For \( j = i \), consider the iteration of the internal **for** loop on the edge \((v_{i-1}, v_i)\). After this iteration, one has

\[
\begin{align*}
    d[v_i] &\leq d[v_{i-1}] + w(v_{i-1}, v_i) \\
           &= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \quad \text{because I2 holds for all } j < i \\
           &= \delta(s, v_i) \quad \text{because } \langle s, v_1, \ldots, v_k \rangle \text{ is a shortest path,} \\
           &\text{whence } \langle s, v_1, \ldots, v_i \rangle \text{ is a shortest path}
\end{align*}
\]

Since the \( d \)-values are non-increasing, this bound continues to hold after other iterations of the internal **for** loop. Hence, \( d[v_i] = \delta(s, v_i) \).
Correctness: the case of negative-weight cycles

**Theorem.** If there is a negative-weight cycle reachable from $s$, then the **for** loop 5-7 will return FALSE.

**Proof.** Suppose that there is a negative weight-cycle reachable from $s$, say $c = (v_1, v_2, \ldots, v_n, v_{n+1})$ with $v_{n+1} = v_1$, and $w(c) := \sum_{i=1}^{n} w(v_i, v_{i+1}) < 0$.

**Fact:** we have the inequality

$$\sum_{i=1}^{n} \left[ d[v_i] + w(v_i, v_{i+1}) \right] = \left( \sum_{i=1}^{n} d[v_i] \right) + w(c)$$

$$< \sum_{i=1}^{n} d[v_i]$$

$$= \sum_{i=1}^{n} d[v_{i+1}] .$$

Since the l.h.s. is strictly smaller than the r.h.s., there must exist a vertex $v_i$ such that $d[v_i] + w(v_i, v_{i+1}) < d[v_{i+1}]$.

Hence, the **for** loop of lines 8-11 will return FALSE upon inspecting the edge $(v_i, v_{i+1})$. \qed
Shortest paths in DAGs [CLRS 24.2]

Idea: Find a topological sort of $V$. The shortest path from $s$ to $v$ can only contain intermediate vertices that come before $v$ in the topological sort.

Input: A DAG $G = (V, E)$ with weight $w : E \to \mathbb{R}$ and a source vertex $s$

Output: For each $v \in V$, length $d[v]$ of shortest-path from $s$ to $v$

1. $\text{TOPOLOGICAL-SORT}(V, E)$
2. for each $v \in V$
3. \hspace{1em} $d[v] = \infty$
4. \hspace{1em} $\pi[v] = \text{nil}$
5. \hspace{1em} $d[s] = 0$
6. for each $u \in V \text{ in topological order}$
7. \hspace{1em} for each $v \in \text{Adj}[u]$
8. \hspace{2em} if $d[v] > d[u] + w(u, v)$
9. \hspace{2em} \hspace{1em} $d[v] = d[u] + w(u, v)$
10. \hspace{2em} \hspace{1em} $\pi[v] = u$
Running time

1. \textsc{Topological-sort}(V, E) takes $O(|V| + |E|)$ time
2. initialization (lines 2-5) takes $\Theta(|V|)$ time
3. line 6 is run one time on each vertex, resulting into a $O(|V|)$ time
4. the \textbf{for} loop of line 7 runs $O(|E|)$ time

\textbf{Total running time:} $O(|V| + |E|)$
Correctness

Invariant of the for loop of lines 6-10.

I1 \[ d[v] \geq \delta(s, v) \text{ for all } v \in V \]

I2 \[ \text{Let } (v_1, v_2, \ldots, v_n) \text{ be the topological sort.} \]

Before the for loop is executed on vertex \( v_k \),

one has \( d[v_i] = \delta(s, v_i) \) for every \( i \leq k \).

Initialisation:

□ \[ d[s] = \delta(s, s). \text{ For } v \neq s, d[v] = \infty \geq \delta(s, v). \text{ Hence, I1 holds.} \]

□ \[ \text{if } v_1 = s, d[v_1] = 0 = \delta(s, s) \text{ before the loop starts.} \]

Hence, I2 holds for \( v_1 = s \)

□ \[ \text{if } v_1 \neq s, \text{ then there cannot be a path from } s \text{ to } v_1. \]

Hence, \( \delta(s, v_1) = \infty = d[v_1] \) and I2 holds for \( v_1 \neq s \).

Termination. At termination, I2 yields \( d[v] = \delta(s, v) \) for every vertex \( v \in V \).

Maintenance of I1: proof as usual (cf. proof in Bellman-Ford algorithm).
Maintenance of I2: \( d[v_i] = \delta(s, v_i) \) for every \( i \leq k \).

When the \textbf{for} loop is executed on vertex \( v_k \), only vertices in \( Adj[v_k] \) are affected. That is, only vertices \( v_j \) with \( j > k \) are affected (because of topological sort). Hence, we only need to consider vertex \( v_{k+1} \).

Let \( v_j \) be the predecessor of \( v_{k+1} \) on a shortest path from \( s \) to \( v_{k+1} \). By construction, \( j < k + 1 \), and therefore \( d[v_j] = \delta(s, v_j) \) by I2 before iteration.

After the \textbf{for} loop 6-10 was executed on \( v_j \), one had

\[
\begin{align*}
    d[v_{k+1}] &\leq d[v_j] + w(v_j, v_{k+1}) \\
    &= \delta(s, v_j) + w(v_j, v_{k+1}) \quad \text{by the invariant before iteration} \\
    &= \delta(s, v_{k+1}).
\end{align*}
\]

Since further iterations of the \textbf{for} loop can not increase the \( d \)-value, and since I1 holds, we conclude that \( d[v_{k+1}] = \delta(s, v_{k+1}) \) after the \textbf{for} loop is executed on \( v_k \).
All-pairs shortest paths [CLRS 25]

**All-pairs shortest paths**

**Input:** A directed graph \((V, E)\) with general weights \(w : E \rightarrow \mathbb{R}\), and with **no negative-weight cycles**.

**Task:** For each pair of vertices \(u\) and \(v\), find shortest path from \(u\) to \(v\).

**Using Single-source algorithms**

- Using Bellman-Ford algorithm for each vertex, leads to a \(O(|V|^2|E|)\) algorithm, which is \(O(|V|^3)\) for a sparse graph \(|E| = O(|V|)\) and \(O(|V|^4)\) for a dense graph \(|E| = O(|V|^2)\).
- If there are no negative-weight edges, using Dijkstra’s algorithm leads to a \(O((|V| + |E|) |V| \log |V|)\) algorithm.

**Using dynamic programming**

- \(O(|V|^3)\) with the Floyd-Warshall algorithm (next slides).
The Floyd-Warshall algorithm [CLRS 25.2]

Suppose the vertex-set is \( \{1, \ldots, n\} \) and let

\[
d[i, j; k] = \begin{cases} 
\text{length of shortest path from } i \text{ to } j, \text{ all of whose} \\
\text{intermediate nodes are in the interval } [1, k] 
\end{cases}
\]

Initially

\[
d[i, j; 0] = \begin{cases} 
w(i, j) & \text{if } (i, j) \in E \\
\infty & \text{otherwise} 
\end{cases}
\]
Suppose that we have calculated \( d[i, j; k] \) for every \( i \) and \( j \), and we want to compute \( d[i, j; k + 1] \).

**Optimal substructure:** if \( s \xrightarrow{p_1} u \xrightarrow{p_2} v \) is a shortest path, then \( s \xrightarrow{p_1} u \) and \( u \xrightarrow{p_2} v \) are shortest paths.

**Fact:** since there are no negative cycles (by assumption), a shortest path from \( i \) to \( j \) that uses intermediate vertices in \([1, k]\) goes through \( k \) at most once.

Hence, we have

\[
d[i, j; k + 1] = \min \left\{ d[i, j; k], \; d[i, k + 1; k] + d[k + 1, j; k] \right\}
\]
Pseudocode

**FLOYD-WARSHALL**($V, E, w$)

1. **for** $i = 1$ **to** $|V|$
2. **for** $j = 1$ **to** $|V|$
3. \hspace{1cm} $d[i, j; 0] = \infty$
4. **for** each edge $(i, j) \in E$
5. \hspace{1cm} $d[i, j; 0] = w(i, j)$
6. **for** $k = 0$ **to** $|V| - 1$
7. **for** $i = 1$ **to** $|V|$
8. \hspace{1cm} **for** $j = 1$ **to** $|V|$
9. \hspace{2cm} $d[i, j; k + 1] = \min \{d[i, j; k], d[i, k + 1; k] + d[k + 1, j; k]\}$

**Running time** is $O(|V|^3)$.  

---

DAA 2020-21  
6. Paths in Graphs – 43 / 44
Summary of Shortest Paths Algorithms

- **Breadth-First-Search** solves the single-source shortest path problem for directed or undirected graphs with unit weight, i.e. where $w(e) = 1$ for all $e \in E$ in $O(|V| + |E|)$.

- **Dijkstra’s Algorithm** solves the single-source shortest-path problem for directed or undirected graphs with non-negative weights in $O((|V| + |E|) \log |V|)$.

- The **Bellman-Ford Algorithm** solves the single-source shortest-path problem for directed or undirected graphs with arbitrary weights in $O(|V||E|)$. In addition, it checks for negative-weight cycles.

- **Shortest Paths in DAGs** solves the single-source shortest path problem with arbitrary weights in the case of a directed acyclic graph in $\Theta(|V| + |E|)$.

- The **Floyd-Warshall Algorithm** solves the all-pairs shortest path problem for directed or undirected graphs with no negative-weight cycles in $O(|V|^3)$. 