24.1 The Bellman-Ford algorithm

- Solves the single-source shortest-paths problem in the case in which weights may be negative
- Given a graph $G = (V, E)$ with source $s$ and weight function $w$, it 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

Relax edges, progressively decreasing $v.d$ until we achieve the actual shortest-path weight $\delta(s, v)$

BELLMAN-FORD($G, w, s$)
1. INITIALIZE-SINGLE-SOURCE($G, s$)
2. for $i = 1$ to $|G.V| - 1$
3. for each edge $(u, v) \in G.E$
4. RELAX($u, v, w$)
5. for each edge $(u, v) \in G.E$
6. if $v.d > u.d + w(u, v)$
7. return FALSE
8. return TRUE
The algorithm makes \(|V| - 1\) passes over the edges of the graph.

Each pass is one iteration of the for loop of lines 2–4 and consists of relaxing each edge of the graph once.

After making \(|V| - 1\) passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value.

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 \(\theta(V)\) time, and the for loop of lines 5–7 takes \(O(E)\) time.
Lemma 24.2:

Let \( G = (V,E) \) be a weighted, directed graph with source \( s \) and weight function \( w: E \to \mathbb{R} \), and assume that \( G \) contains no negative-weight cycles that are reachable from \( s \).

Then, after the \( |V| - 1 \) iterations of the for loop of lines 2–4 of BELLMAN-FORD, we have \( v.d = \delta(s,v) \) for all vertices \( v \) that are reachable from \( s \).

**Proof:** We prove the lemma by appealing to the path-relaxation property.

Consider any vertex \( v \) that is reachable from \( s \), and let \( p = (v_0,v_1,...,v_k) \), where \( v_0 = s \) and \( v_k = v \), be any shortest path from \( s \) to \( v \).

Because shortest paths are simple, \( p \) has at most \( |V| - 1 \) edges, and so \( k \leq |V| - 1 \). Each of the \( |V| - 1 \) iterations of the for loop of lines 2–4 relaxes all \( |E| \) edges.

Among the edges relaxed in the \( i \)th iteration, for \( i = 1,2,...,k \), is \( (v_{i-1},v_i) \). By the path-relaxation property, therefore,

\[
v.d = v_k.d = \delta(s,v_k) = \delta(s,v)\]

\( \Box \)
**Corollary 24.3:**

Let $G = (V, E)$ be a weighted, directed graph

- with source $s$ and
- weight function $w: E \rightarrow \mathbb{R}$, and
- assume that $G$ contains no negative-weight cycles that are reachable from $s$.

Then, for each vertex $v \in V$, there is a path from $s$ to $v$ if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on $G$.

---

**Theorem 24.4** (Correctness of the B-F algorithm)

Let BELLMAN-FORD be run on

- a weighted, directed graph $G = (V, E)$
- with source $s$ and
- weight function $w: E \rightarrow \mathbb{R}$.

If $G$ contains no negative-weight cycles that are reachable from $s$, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$.

If $G$ does contain a negative-weight cycle reachable from $s$, then the algorithm returns FALSE.
24.3 Dijkstra’s algorithm

- Dijkstra’s 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 nonnegative.
- Therefore, we assume that \( w(u, v) \geq 0 \) for each edge \((u, v) \in E\).
- With a good implementation, the running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm.

- Dijkstra’s 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 \( u \).
- The following implementation uses a min-priority queue \( Q \) of vertices, keyed by their \( d \) values.
Dijkstra($G, w, s$)

1. INITIALIZE-SINGLE-SOURCE($G, s$)
2. $S \leftarrow \emptyset$
3. $Q \leftarrow G.V$
4. while $Q \neq \emptyset$
5. $u \leftarrow$ EXTRACT-MIN($Q$)
6. $S \leftarrow S \cup \{u\}$
7. for each vertex $v \in G.Adj[u]$
8. RELAX $(u, v, w)$

- Shaded edges indicate predecessor values
- Black vertices are in the set $S$
Maintain the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8

- Line 3 initializes the min-priority queue $Q$ to contain all the vertices in $V$; since $S = \emptyset$; at that time, the invariant is true after line 3
- Each time through the **while** loop, line 5 extracts a vertex $u$ from $Q = V - S$ and line 6 adds it to set $S$, thereby maintaining the invariant
- The first time through this loop, $u = s$
- Vertex $u$, therefore, has the smallest shortest-path estimate of any vertex in $V - S$

Then, lines 7–8 relax each edge $(u,v)$ leaving $u$, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to $v$ found so far by going through $u$

- Observe that the algorithm never inserts vertices into $Q$ after line 3 and that each vertex is extracted from $Q$ and added to $S$ exactly once, so that the **while** loop iterates exactly $|V|$ times
- Because Dijkstra’s algorithm always chooses the “lightest” or “closest” vertex in $V - S$ to add to set $S$, we say that it uses a greedy strategy
Theorem 24.6 (Correctness of Dijkstra’s algorithm)

Dijkstra’s algorithm, run on a weighted, directed graph \( G = (V,E) \) with non-negative weight function \( w \) and source \( s \), terminates with \( u.d = \delta(s,u) \) for all vertices \( u \in V \).

Proof: We use the following loop invariant:

- At the start of each iteration of the while loop of lines 4–8, \( v.d = \delta(s,v) \) for each vertex \( v \in S \).

It suffices to show for each vertex \( u \in V \), we have \( u.d = \delta(s,u) \) at the time when \( u \) is added to set \( S \). Once \( u.d = \delta(s,u) \), we rely on the upper-bound property to show that it holds thereafter.

Initialization: Initially, \( S = \emptyset \), and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration, \( u.d = \delta(s,u) \) for the vertex added to set \( S \). For the purpose of contradiction, let \( u \) be the first vertex for which \( u.d \neq \delta(s,u) \) when it is added to set \( S \). We shall focus our attention on the situation at the beginning of the iteration of the while loop in which \( u \) is added to \( S \) and derive the contradiction that \( u.d = \delta(s,u) \) at that time by examining a shortest path from \( s \) to \( u \). We must have \( u \neq s \) because \( s \) is the first vertex added to set \( S \) and \( s.d = \delta(s,s) = 0 \) at that time.
Because \( u \neq s \), we also have that \( S \neq \emptyset \) just before \( u \) is added to \( S \). There must be some path from \( s \) to \( u \), for otherwise \( u.d = \delta(s,u) = \infty \) by the no-path property, which would violate our assumption that \( u.d \neq \delta(s,u) \). Because there is at least one path, there is a shortest path \( p \) from \( s \) to \( u \). Prior to adding \( u \) to \( S \), path \( p \) connects a vertex in \( S \), namely \( s \), to a vertex in \( V - S \), namely \( u \). Let us consider the first vertex \( y \) along \( p \) such that \( y \in V - S \), and let \( x \in S \) be \( y \)'s predecessor along \( p \).

Thus, we can decompose path \( p \) into \( s \xrightarrow{p_1} x \xrightarrow{p_2} y \xrightarrow{p_2} u \). (Either path \( p_1 \) or \( p_2 \) may have no edges.)
We claim that \( y.d = \delta(s,y) \) when \( u \) is added to \( S \). To prove this claim, observe that \( x \in S \). Then, because we chose \( u \) as the first vertex for which \( u.d \neq \delta(s,u) \) when it is added to \( S \), we had \( x.d = \delta(s,x) \) when \( x \) was added to \( S \). Edge \((x,y)\) was relaxed at that time, and the claim follows from the convergence property.

We can now obtain a contradiction to prove that \( u.d = \delta(s,u) \). Because \( y \) appears before \( u \) on a shortest path from \( s \) to \( u \) and all edge weights are non-negative (notably those on path \( p_2 \)), we have \( \delta(s,y) \leq \delta(s,u) \), and thus

\[
y.d = \delta(s,y) \leq \delta(s,u) \leq u.d.
\]

But because both vertices \( u \) and \( y \) were in \( V - S \) when \( u \) was chosen in line 5, we have \( u.d \leq y.d \). Thus, the two inequalities above are in fact equalities, giving

\[
y.d = \delta(s,y) = \delta(s,u) = u.d.
\]

Consequently, \( u.d = \delta(s,u) \), which contradicts our choice of \( u \). We conclude that \( u.d = \delta(s,u) \) when \( u \) is added to \( S \), and that this equality is maintained at all times thereafter.

**Termination:** At termination, \( Q = \emptyset \) which, along with our invariant that \( Q = V - S \), implies that \( S = V \). Thus, \( u.d = \delta(s,u) \) for all vertices \( u \in V \).
25 All-Pairs Shortest Paths

- Consider the problem of finding shortest paths between all pairs of vertices in a graph.
- Given a weighted, directed graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbb{R}$ that maps edges to real-valued weights.
- We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from $u$ to $v$, where the weight of a path is the sum of the weights of its constituent edges.

We can solve the problem by running a single-source shortest-paths algorithm $|V|$ times.
- If all edge weights are nonnegative, we can use Dijkstra’s algorithm.
- If we use the linear-array implementation of the min-priority queue, the running time is $O(V^3 + VE) = O(V^3)$.
- Binary min-heap implementation of min-priority queue yields a running time of $O(VE \lg V)$, which is an improvement if the graph is sparse.
- Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \lg V + VE)$. 


• If the graph has negative-weight edges, we cannot use Dijkstra’s algorithm
• Instead, we must run the slower Bellman-Ford algorithm once from each vertex
• The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$
• We shall see how to do better

• APSP = all-pairs shortest-paths (APSP) problem

• Unlike the single-source algorithms, most of the following algorithms use an adjacency-matrix representation
• For convenience, we assume that the vertices are numbered $1, 2, \ldots, |V|$, so that the input is an $n \times n$ matrix $W$ representing the edge weights of an $n$-vertex directed graph $G = (V, E)$
• I.e., $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \text{ and } (i, j) \in E \\ -\lambda & \text{if } i \neq j \text{ and } (i, j) \notin E \\ \end{cases}$$
We allow negative-weight edges, but assume that the input graph contains no negative-weight cycles.

The tabular output of the APSP algorithms is an $n \times n$ matrix $D = (d_{ij})$, where entry $d_{ij}$ contains the weight of a shortest path from vertex $i$ to vertex $j$.

That is, if we let $\delta(i,j)$ denote the shortest-path weight from vertex $i$ to vertex $j$, then $d_{ij} = \delta(i,j)$ at termination.

To solve the APSP problem on an input adjacency matrix, we need to compute also a predecessor matrix $\Pi = (\pi_{ij})$,

where $\pi_{ij}$ is NIL if either $i = j$ or there is no path from $i$ to $j$, and otherwise $\pi_{ij}$ is the predecessor of $j$ on some shortest path from $i$.

Just as the predecessor subgraph $G_{\pi}$ is a shortest-paths tree for a given source vertex, the subgraph induced by the $i$th row of the $\Pi$ matrix should be a shortest-paths tree with root $i$. 
• For each vertex $i \in V$, define the predecessor subgraph of $G$ for $i$ as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{ j \in V : \pi_{ij} \neq \text{NIL} \} \cup \{i\}$$

and

$$E_{\pi,i} = \{ (\pi_{ij}, j) : j \in V_{\pi,i} - \{i\} \}$$

• If $G_{\pi,i}$ is a shortest-paths tree, then the following procedure prints a shortest path from vertex $i$ to vertex $j$

```
PRINT-ALL-PAIRS-SHORTEST-PATH(\Pi, i, j)
1. if $i = j$
2. print $i$
3. elseif $\pi_{ij} = \text{NIL}$
4. print “no path from” $i$ “to” $j$ “exists”
5. else PRINT-ALL-PAIRS-SHORTEST-PATH(\Pi, i, \pi_{ij})
6. print $j$
```
25.2 The Floyd-Warshall algorithm

- We use a dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph \( G = (V, E) \).
- The resulting algorithm runs in \( \Theta(V^3) \) time.
- As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles.
- After studying the resulting algorithm, we look at a similar method for finding the transitive closure of a directed graph.

The structure of a shortest path

- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path.
- An intermediate vertex of a simple path \( p = \{v_1, v_2, \ldots, v_l\} \) is any vertex of \( p \) other than \( v_1 \) or \( v_l \).
- I.e., any vertex in the set \( \{v_2, v_3, \ldots, v_{l-1}\} \).
- Under our assumption that the vertices of \( G \) are \( V = \{1, 2, \ldots, n\} \), let us consider a subset \( \{1, 2, \ldots, k\} \) of vertices for some \( k \).
For any pair of vertices \( i, j \in V \), consider all paths from \( i \) to \( j \) whose intermediate vertices are all drawn from \( \{1, 2, \ldots, k\} \), and let \( p \) be a minimum-weight path from among them.

The Floyd-Warshall algorithm exploits a relationship between path \( p \) and shortest paths from \( i \) to \( j \) with all intermediate vertices in the set \( \{1, 2, \ldots, k - 1\} \).

The relationship depends on whether or not \( k \) is an intermediate vertex of path \( p \).

If \( k \) is not an intermediate vertex of \( p \), then all intermediate vertices of \( p \) are in the set \( \{1, 2, \ldots, k - 1\} \). Thus, a shortest path from \( i \) to \( j \) with all intermediate vertices in the set \( \{1, 2, \ldots, k - 1\} \) is also a shortest path from \( i \) to \( j \) with all intermediate vertices in the set \( \{1, 2, \ldots, k\} \).

If \( k \) is an intermediate vertex of path \( p \), then we decompose \( p \) into

\[
\begin{align*}
& p_1 \quad p_2 \\
& i \rightarrow k \rightarrow j
\end{align*}
\]

By Lemma 24.1, \( p_1 \) is a shortest path from \( i \) to \( k \) with all intermediate vertices in the set \( \{1, 2, \ldots, k\} \).
• In fact, we can make a slightly stronger statement.
• Because vertex $k$ is not an intermediate vertex of path $p_1$, all intermediate vertices of $p_1$ are in the set $\{1, 2, \ldots, k - 1\}$.
• Therefore, $p_1$ is a shortest path from $i$ to $k$ with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$.
• Similarly, $p_2$ is a shortest path from vertex $k$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$.
A recursive solution to the all-pairs shortest-paths problem

- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex $i$ to vertex $j$ for which all intermediate vertices are in the set \{1,2,...,k\}
- When $k = 0$, a path from $i$ to $j$ with no intermediate vertex numbered higher than 0 has no intermediate vertices at all
- Such a path has at most one edge, and hence
  $$d_{ij}^{(0)} = w_{ij}$$

Let us define recursively
  $$d_{ij}^{(k)} = \begin{cases} 
    w_{ij} & \text{if } k = 0 \\
    \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & \text{if } k \geq 1 
  \end{cases}$$

- Because for any path, all intermediate vertices are in the set \{1,2,...,n\}, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer:
  $$d_{ij}^{(n)} = \delta(i,j)$$
  for all $i,j \in V$
Computing the shortest-path weights bottom up

- Based on the prev. recurrence, we can use the following bottom-up procedure to compute the values $d_{ij}^{(k)}$ in order of increasing values of $k$

- Its input is an $n \times n$ matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

- The procedure returns the matrix $D^{(n)}$ of shortest-path weights

---

**FLOYD-WARSHALL** ($W$)

1. $n \leftarrow W.rows$
2. $D^{(0)} \leftarrow W$
3. for $k \leftarrow 1$ to $n$
4. let $D^{(k)} = \left( d_{ij}^{(k)} \right)$ be a new $n \times n$ matrix
5. for $i \leftarrow 1$ to $n$
6. for $j \leftarrow 1$ to $n$
7. $d_{ij}^{(k)} \leftarrow \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)$
8. return $D^{(n)}$
The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3–7.

Because each execution of line 7 takes \( O(1) \) time, the algorithm runs in time \( \Theta(n^3) \).

The code is tight, with no elaborate data structures, and so the constant hidden in the \( \Theta \)-notation is small.

Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.
Transitive closure of a directed graph

- Given a directed $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, we wish to determine whether $G$ contains a path from $i$ to $j$ for all vertex pairs $i, j \in V$.
- We define the **transitive closure** of $G$ as the graph $G^* = (V, E^*)$, where
  \[ E^* = \{ (i, j) : \text{there is a path from } i \text{ to } j \text{ in } G \} \]
- One way to compute it is to assign a weight of 1 to each edge of $E$ and run the F-W algorithm.
- If there is a path from $i$ to $j$, we get $d_{ij} < n$. Otherwise, $d_{ij} = \infty$.

- Another, similar way to compute the transitive closure of $G$ in $\Theta(n^3)$ time that can save time and space in practice.
- This method substitutes the logical operations $\lor$ and $\land$ for the arithmetic operations $\min$ and $+$ in the Floyd-Warshall algorithm.
- For $i, j, k = 1, 2; \ldots, n$, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph $G$ from vertex $i$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise.
- We construct the transitive closure $G^* = (V, E^*)$ by putting edge $(i, j)$ into $E^*$ if and only if $t_{ij}^{(n)} = 1$. 
A recursive definition of $t_{ij}^{(k)}$ is

$$t_{ij}^{(0)} = \begin{cases} 
0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\
1 & \text{if } i = j \text{ or } (i,j) \in E 
\end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \lor (t_{ik}^{(k-1)} \land t_{kj}^{(k-1)})$$

As in the Floyd-Warshall algorithm, we compute the matrices $T^{(k)} = (t_{ij}^{(k)})$ in order of increasing $k$.
TRANSITIVE-CLOSURE($G$)

1. $n \leftarrow |G.V|$

2. let $T^{(0)} = \begin{pmatrix} t^{(0)}_{ij} \end{pmatrix}$ be a new $n \times n$ matrix

3. for $i \leftarrow 1$ to $n$

4. for $j \leftarrow 1$ to $n$

5. if $i = j$ or $(i,j) \in G.E$

6. $t^{(0)}_{ij} \leftarrow 1$

7. else $t^{(0)}_{ij} \leftarrow 0$

8. for $k \leftarrow 1$ to $n$

9. let $T^{(k)} = \begin{pmatrix} t^{(k)}_{ij} \end{pmatrix}$ be a new $n \times n$ matrix

10. for $i \leftarrow 1$ to $n$

11. for $j \leftarrow 1$ to $n$

12. $t^{(k)}_{ij} = t^{(k-1)}_{ij} \lor \left( t^{(k-1)}_{ik} \land t^{(k-1)}_{kj} \right)$

13. return $T^{(n)}$
• The TRANSITIVE-CLOSURE procedure, like the F-W algorithm, runs in $\Theta(n^3)$ time
• On some computers logical operations on single-bit values execute faster than arithmetic operations on integer words of data
• Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the F-W algorithm’s by a factor corresponding to the size of a word of computer storage