Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

1. All-Pairs Shortest Paths
    - Shortest Paths and Matrix Multiplication
    - The Floyd-Warshall Algorithm
    - Johnson's Algorithm for Sparse Graphs

The All-Pairs Shortest Paths Problem

- Consider a weighted, directed graph $G = (V, E)$, with a weight function $w : E \to \mathbb{R}$, 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 typically want the output in tabular form:

  The entry in u's row and v's column should be the weight of a shortest path from u to v.

All-Pairs via Exhaustive Single Pair

- We can solve an all-pairs shortest-paths problem by running a single source shortest-paths algorithm $|V|$ times, once for each vertex as the source.
  - 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 $\mathrm{O}\left(|V|^3 + |V||E|\right) = \mathrm{O}\left(|V|^3\right)$.
    - The binary min-heap implementation of the min-priority queue yields a running time of $\mathrm{O}\left(|V||E| \log |V|\right)$, which is an improvement if the graph is sparse.
  - If the graph has negative-weight edges, we must run the slower Bellman-Ford algorithm once from each vertex.

    The resulting running time is $\mathrm{O}\left(|V|^2|E|\right)$, which on a dense graph is $\mathrm{O}\left(|V|^4\right)$.

The Set Up and the Variables

- We mostly use an adjacency matrix representation.
- Assume that the vertices are numbered $1, 2, \ldots, |V|$.
- Then the input is an $n \times n$ matrix $W = (w_{ij})$, representing the edge weights of the n-vertex directed graph $G = (V, E)$,

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

- We allow negative-weight edges, but we assume for the time being that the input graph contains no negative-weight cycles.
- The tabular output of the all-pairs shortest-paths 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.
- 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.

Predecessor Matrix and Predecessor Subgraph

- To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a **predecessor matrix** $\Pi = (\pi_{ij})$, where:
  - π_{ij} is NIL if either $i = j$ or there is no path from i to j;
  - π_{ij} is the predecessor of j on some shortest path from i, otherwise.
- For each vertex $i \in V$, we define the **predecessor subgraph of** G **for** i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where:
  - $V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$;
  - $E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\}$.
- Just as the predecessor subgraph G_π is a shortest-paths tree for a given source vertex, the predecessor subgraph $G_{\pi,i}$ of G for i (induced by the i-th row of the Π matrix) should be a shortest-paths tree with root i.

Print All Pairs Shortest Paths Procedure

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

```
PRINTALLPAIRSSHORTESTPATH($\Pi, i, j$)
if $i == j$
    print $i$
elseif $\pi_{ij} ==$ NIL
    print “no path from” $i$ “to” $j$ “exists”
else PRINTALLPAIRSSHORTESTPATH($\Pi, i, \pi_{ij}$)
    print $j$
```

Conventions and Notation

- We assume that the input graph $G = (V, E)$ has n vertices, so that $n = |V|$.
- We use the convention of denoting matrices by uppercase letters, such as W, L or D, and their individual elements by subscripted lowercase letters, such as w_{ij}, ℓ_{ij} or d_{ij}.
- Some matrices will have parenthesized superscripts, e.g., $L^{(m)} = (\ell_{ij}^{(m)})$ or $D^{(m)} = (d_{ij}^{(m)})$, to indicate iterates.
- Finally, for a given $n \times n$ matrix A, we assume that the value of n is stored in the attribute A.rows.

Subsection 1

Shortest Paths and Matrix Multiplication

Dynamic Programming for All-Pairs Shortest Paths

- We present a dynamic programming algorithm for the all-pairs shortest paths problem on a directed graph $G = (V, E)$.
- Each major loop of the dynamic program will invoke an operation that is very similar to matrix multiplication, so that the algorithm will look like repeated matrix multiplication.
- We first develop a $\Theta(|V|^4)$-time algorithm for the all-pairs shortest paths problem and then improve its running time to $\Theta(|V|^3 \log |V|)$.
- Recall the steps for developing a dynamic programming algorithm:
    1. Characterize the structure of an optimal solution.
    2. Recursively define the value of an optimal solution.
    3. Compute the value of an optimal solution in a bottom-up fashion.
    4. Construct an optimal solution from computed information.
       This step will not be carried out in detail.

The Structure of a Shortest Path

- We characterize the structure of an optimal solution.
- For the all-pairs shortest paths problem on a graph $G = (V, E)$, we have proven that all subpaths of a shortest path are shortest paths.
- Suppose we represent the graph by an adjacency matrix $W = (w_{ij})$.
- Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most m edges.

  Assuming that there are no negative-weight cycles, m is finite.
  - If $i = j$, then p has weight 0 and no edges.
  - If $i \neq j$, then we decompose p into $i \overset{p'}{\rightsquigarrow} k \to j$, where p' contains at most $m - 1$ edges. We proved that p' is a shortest path from i to k. So

$$\delta(i, j) = \delta(i, k) + w_{kj}.$$

A Recursive Solution to All-Pairs Shortest-Paths

- Let $\ell_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
  - When $m = 0$, there is a shortest path from i to j with no edges if and only if $i = j$. Thus, $\ell_{ij}^{(0)} = \begin{cases} 0, & \text{if } i = j \\ 1, & \text{if } i \neq j \end{cases}$.
  - For $m \geq 1$, we compute $\ell_{ij}^{(m)}$ as the minimum of $\ell_{ij}^{(m-1)}$ (the weight of a shortest path from i to j consisting of at most $m-1$ edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j.
    Thus, we recursively define

$$\begin{aligned} \ell_{ij}^{(m)} &= \min\left(\ell_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\{\ell_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1 \leq k \leq n}\{\ell_{ik}^{(m-1)} + w_{kj}\}. \end{aligned}$$

The latter equality follows since $w_{jj} = 0$, for all j.

The Shortest Path Weights $\delta(i, j)$

- If the graph contains no negative-weight cycles, then, for every pair of vertices i and j for which $\delta(i, j) < \infty$, there is a shortest path from i to j that is simple and, thus, contains at most $n - 1$ edges.
- A path from vertex i to vertex j with more than $n - 1$ edges cannot have lower weight than a shortest path from i to j.
- The actual shortest-path weights are therefore given by

$$\delta(i, j) = \ell_{ij}^{(n-1)} = \ell_{ij}^{(n)} = \ell_{ij}^{(n+1)} = \cdots .$$

Extending Shortest Paths

- With input $W = (w_{ij})$, we compute $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.
- The final matrix $L^{(n-1)}$ contains the actual shortest-path weights.
- Observe that $\ell_{ij}^{(1)} = w_{ij}$, for all vertices $i, j \in V$, whence $L^{(1)} = W$.
- The following procedure, given $L^{(m-1)}$ and W, returns $L^{(m)}$, i.e., it extends the shortest paths computed so far by one more edge.

ExtendShortestPaths(L, W)

```
n = L.rows
let $L' = (\ell'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ to $n$
    for $j = 1$ to $n$
        $\ell'_{ij} = \infty$
        for $k = 1$ to $n$
            $\ell'_{ij} = \min(\ell'_{ij}, \ell_{ik} + w_{kj})$
return $L'$
```

Computing the Shortest-Path Weights Bottom Up

- For all-pairs shortest paths, we compute the shortest-path weights by extending shortest paths edge by edge.

  Letting $A \cdot B$ denote $\textsc{ExtendShortestPaths}(A, B)$, we compute the sequence of $n-1$ matrices

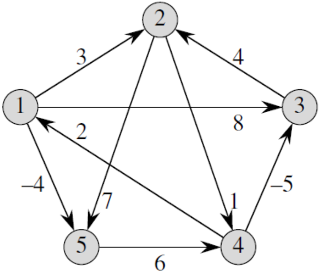
  $$L^{(1)} = L^{(0)}W = W,\ L^{(2)} = L^{(1)}W = W^2, \ldots, L^{(n-1)} = L^{(n-2)}W = W^{n-1}.$$
- The matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights.
- The following procedure computes this sequence in $\Theta(n^4)$ time.

$\textsc{SlowAllPairsShortestPaths}(W)$

```
1. $n = W.\text{rows}$
2. $L^{(1)} = W$
3. for $m = 2$ to $n - 1$
4.     let $L^{(m)}$ be a new $n \times n$ matrix
5.     $L^{(m)} = \textsc{ExtendShortestPaths}(L^{(m-1)}, W)$
6. return $L^{(n-1)}$
```

An Example

- A graph and the matrices $L^{(m)}$ computed by the procedure SlowAllPairsShortestPaths.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Improving the Running Time

- We are interested in $L^{(n-1)}$; not in all $L^{(m)}$.
- Without negative-weight cycles, $L^{(m)} = L^{(n-1)}$, for all $m \geq n-1$.
- Since the EXTENDSHORTESTPATHS operation ("$\cdot$") is associative, we can compute $L^{(n-1)}$ with only $\lceil \log (n-1) \rceil$ matrix products by computing:

$$\begin{array}{rcl} L^{(1)} & = & W; \\ L^{(2)} & = & W^2 = W \cdot W; \\ L^{(4)} & = & W^4 = W^2 \cdot W^2; \\ & \vdots & \\ L(2^{\lceil \log (n-1) \rceil}) & = & W^{2^{\lceil \log (n-1) \rceil}} = W^{2^{\lceil \log (n-1) \rceil}-1} \cdot W^{2^{\lceil \log (n-1) \rceil}-1}. \end{array}$$

- Since $2^{\lceil \log (n-1) \rceil} \geq n-1$, the product $L^{(2^{\lceil \log (n-1) \rceil})}$ is equal to $L^{(n-1)}$.

Computing the Sequence of Matrices

```
FasterAllPairsShortestPaths($W$)
$n = W$.rows
$L^{(1)} = W$
$m = 1$
while $m < n - 1$
    let $L^{(2m)}$ be a new $n \times n$ matrix
    $L^{(2m)} =$ ExtendShortestPaths($L^{(m)}, L^{(m)}$)
    $m = 2m$
return $L^{(m)}$
```

Correctness and Time Requirements

- In each iteration of the while loop of Lines 4-7, we compute $L^{(2m)} = (L^{(m)})^2$, starting with $m = 1$.
- At the end of each iteration, we double the value of m.
- The final iteration computes $L^{(n-1)}$ by actually computing $L^{(2m)}$, for some $n - 1 \leq 2m < 2n - 2$, whence $L^{(2m)} = L^{(n-1)}$.
- The next time the test in Line 4 is performed, m has been doubled,

  So $m \geq n - 1$ and the "while" test fails.

  The procedure returns the last matrix it computed.
- The running time of FASTERALLPAIRSSHORTESTPATHS is $\Theta(n^3 \log n)$, since each of the $\lceil \log(n-1) \rceil$ matrix products takes $\Theta(n^3)$ time.

Subsection 2

The Floyd-Warshall Algorithm

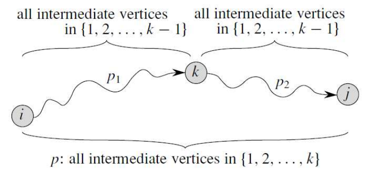
Idea Behind the Floyd-Warshall Algorithm

- In the **Floyd-Warshall algorithm**, we again characterize the structure of a shortest path.
- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ is any vertex in the set $\{v_2, v_3, \ldots, v_{\ell-1}\}$.
- The Floyd-Warshall algorithm relies on the following observation:
  - 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 among them (p is simple).
  - The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. This relationship depends on whether or not k is an intermediate vertex of path p.

The Two Cases Considered by Floyd-Warshall

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$.

  Thus, a shortest path from vertex i to vertex 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 $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$. Then p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.



  Note that 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

- We define a recursive formulation of shortest path estimates.
- 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 $\{1, 2, \ldots, k\}$.
- When $k = 0$, a path from vertex i to vertex 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 $d_{ij}^{(0)} = w_{ij}$.
- Define $d_{ij}^{(k)}$ recursively by

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

- Because for any path, all intermediate vertices are in $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives $d_{ij}^{(n)} = \delta(i, j)$, for all $i, j \in V$.

Computing Shortest-Path Weights Bottom Up

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

FloydWarshall(W)

```
$n = W.\text{rows}$
$D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

Running Time

- 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 Θ-notation is small.
- Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Constructing a Shortest Path

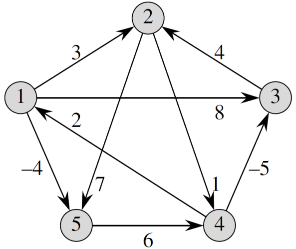
- There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm.
  - One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix.
    Given Π, the PRINTALLPAIRSSHORTESTPATH procedure will print the vertices on a given shortest path.
  - Alternatively, we can compute the predecessor matrix Π while the algorithm computes the matrices $D^{(k)}$.
    Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

Constructing a Shortest Path (The Recursive Formulas)

- When $k = 0$, a shortest path from i to j has no intermediate vertices at all.
$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & \text{if } i = j \text{ or } w_{ij} = \infty \\ i, & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}.$$
- For $k \geq 1$:
    - If we take the path $i \leq k \leq j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.
    - Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

Formally, for $k \geq 1$,
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)}, & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}.$$

Example



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Transitive Closure of a Directed Graph

- Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, we define the **transitive closure** of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}.$$

- One way to compute G^* in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm.
  - If there is a path from vertex i to vertex j, we get $d_{ij} < n$.
  - Otherwise, we get $d_{ij} = \infty$.

Transitive Closure: Alternative Way

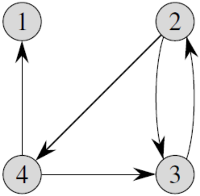
- For $i, j, k = 1, 2, \ldots, n$, define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise.
- Then $G^* = (V, E^*)$ has (i, j) in E^* if and only if $t_{ij}^{(n)} = 1$.
- A recursive definition of $t_{ij}^{(k)}$ is:
  - For $k = 0$,
  $$t_{ij}^{(0)} = \begin{cases} 0, & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1, & \text{if } i = j \text{ or } (i, j) \in E \end{cases}.$$
  - For $k \geq 1$,
  $$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Computing the Transitive Closure

TransitiveClosure(G)

```
$n = |G.V|$
let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
for $i = 1$ to $n$
    for $j = 1$ to $n$
        if $i == j$ or $(i, j) \in G.E$
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for $k = 1$ to $n$
    let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
return $T^{(n)}$
```

Example



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Subsection 3

Johnson's Algorithm for Sparse Graphs

Goal of Johnson's Algorithm

- **Johnson's algorithm** finds shortest paths between all pairs in $O\left(|V|^2 \log |V| + |V||E|\right)$ time.
- For sparse graphs, it is asymptotically faster than either repeated squaring of matrices or the Floyd-Warshall algorithm.
- The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative weight cycle.
- Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm.

Johnson's Algorithm and Reweighting

- Johnson's algorithm uses the technique of **reweighting**:
  - If all edge weights w in a graph $G = (V, E)$ are nonnegative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex.
    With an efficient implementation of a min-priority queue, the running time of this all-pairs algorithm is $\mathrm{O}\left(|V|^2 \log |V| + |V||E|\right)$.
  - If G has negative-weight edges but no negative-weight cycles, we compute a new set of nonnegative edge weights $\widehat{w}$ that allows us to use the same method, which must satisfy:
    1. For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\widehat{w}$.
    2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is nonnegative.

    We can preprocess G to determine the new weight function $\widehat{w}$ in $\mathrm{O}(|V||E|)$ time.

Preserving Shortest Paths by Reweighting

- We use:
    - δ for shortest-path weights derived from w;
    - $\widehat{\delta}$ for shortest-path weights derived from $\widehat{w}$.

Lemma (Reweighting does not Change Shortest Paths)

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v).$$

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex v_0 to vertex v_k. Then p is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function $\widehat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $w(p) = \widehat{\delta}(v_0, v_k)$.

Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\widehat{w}$.

Preserving Shortest Paths by Reweighting (Proof)

- We start by showing that $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$.

$$\begin{array}{rcl} \widehat{w}(p) & = & \sum_{i=1}^k \widehat{w}(v_{i-1}, v_i) \\ & = & \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ & = & \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ & = & w(p) + h(v_0) - h(v_k). \end{array}$$

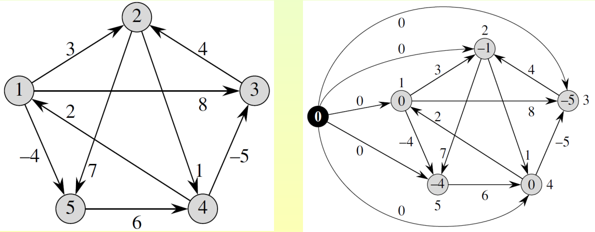
  Any path p from v_0 to v_k has $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$.

  $h(v_0)$ and $h(v_k)$ do not depend on the path. So, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using $\widehat{w}$. Thus, $w(p) = \delta(v_0, v_k)$ iff $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.
- Finally, we show that G has a negative-weight cycle using w if and only if G has a negative-weight cycle using $\widehat{w}$.

  Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. We have $\widehat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c)$. Thus c has negative weight using w if and only if it has negative weight using $\widehat{w}$.

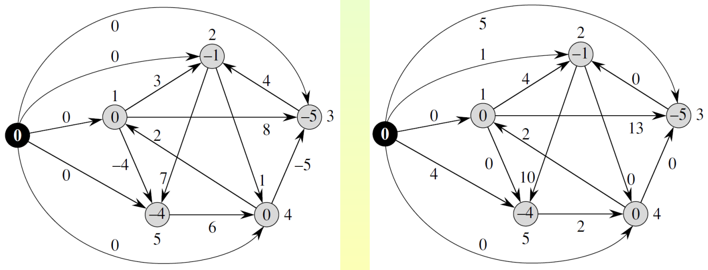
Producing Nonnegative Weights by Reweighting I

- Next, we ensure $\widehat{w}(u,v)$ is nonnegative, for all edges $(u,v) \in E$.
- Given a weighted, directed graph $G = (V,E)$ with weight function $w : E \to \mathbb{R}$, we construct $G' = (V', E')$, where $V' = V \cup \{s\}$, for some new vertex $s \notin V$, and $E' = E \cup \{(s,v) : v \in V\}$.
- We extend the weight function w so that $w(s,v) = 0$, for all $v \in V$.
    - Note that because s has no edges that enter it, no shortest paths in G', other than those with source s, contain s.
    - Moreover, G' has no negative-weight cycles if and only if G has no negative-weight cycles.



Producing Nonnegative Weights by Reweighting II

- Now suppose that G and G' have no negative-weight cycles.
- Let us define $h(v) = \delta(s, v)$, for all $v \in V'$.
- By the Triangle Inequality, we have $h(v) \leq h(u) + w(u, v)$, for all edges $(u, v) \in E'$.
- Thus, if we define the new weights $\widehat{w}$ by reweighting according to $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$, we have $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, as was our goal.



Johnson's Procedure

- Assume the edges are stored in adjacency lists.
- Output is $D = (d_{ij})$, where $d_{ij} = \widehat{\delta}(i,j)$, or "negative-weight cycle".

```
JOHNSON($G, w$)
compute $G'$, where $G'.V = G.V \cup \{s\}$, $G'.E = G.E \cup \{(s,v) : v \in G.V\}$,
and $w(s,v) = 0$, for all $v \in G.V$
if BELLMANFORD($G', w, s$) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
    set $h(v)$ to the value of $\delta(s,v)$ computed by the Bellman-Ford algorithm
for each edge $(u,v) \in G'.E$
    $\widehat{w}(u,v) = w(u,v) + h(u) - h(v)$
let $D = (d_{uv})$ be a new $n \times n$ matrix
for each vertex $u \in G.V$
    run DIJKSTRA($G, \widehat{w}, u$) to compute $\widehat{\delta}(u,v)$, for all $v \in G.V$
    for each vertex $v \in G.V$
        $d_{uv} = \widehat{\delta}(u,v) + h(v) - h(u)$
return $D$
```

How JOHNSON Works

- Line 1 produces G'.
- Line 2 runs Bellman-Ford on G' with weight function w and source s.
  - If G', hence G, contains a negative-weight cycle, Line 3 reports this.
  - Lines 4-12 assume that G' contains no negative-weight cycles.
    - Lines 4-5 set $h(v)$ to the shortest-path weight $\delta(s,v)$, computed by the Bellman-Ford algorithm, for all $v \in V'$.
    - Lines 6-7 compute the new weights $\widehat{w}$.
    - For each pair of vertices $u,v \in V$, the for loop of Lines 9-12 computes the shortest-path weight $\widehat{\delta}(u,v)$ by calling Dijkstra's algorithm once from each vertex in V.
    - Line 12 stores in d_{uv} the correct shortest-path weight $\delta(u,v)$.
- Finally, Line 13 returns the completed D matrix.
- If we implement the min-priority queue in Dijkstra's algorithm efficiently, JOHNSON runs in $\mathrm{O}\left(|V|^2\log|V| + |V||E|\right)$ time.
- Even a simpler minheap implementation yields $\mathrm{O}(|V||E|\log|V|)$, still asymptotically faster than Floyd-Warshall, if the graph is sparse.

Illustrating JOHNSON