Shortest Paths

1. Dijkstra’s algorithm.
2. The Bellman-Ford algorithm.
3. The Floyd-Warshall algorithm.

The Bellman-Ford algorithm

1958

1962

The Bellman-Ford algorithm

Procedure BELLMAN-FORD(G, s)

Inputs:

- G: a directed graph containing a set V of n vertices and a set E of m directed edges with arbitrary weights.
- s: a source vertex in V.

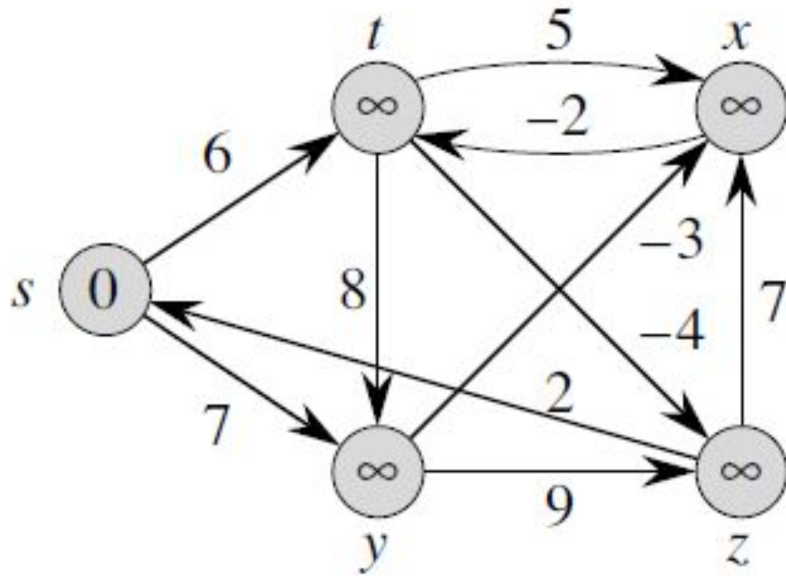
Result: Same as DIJKSTRA (page 94).

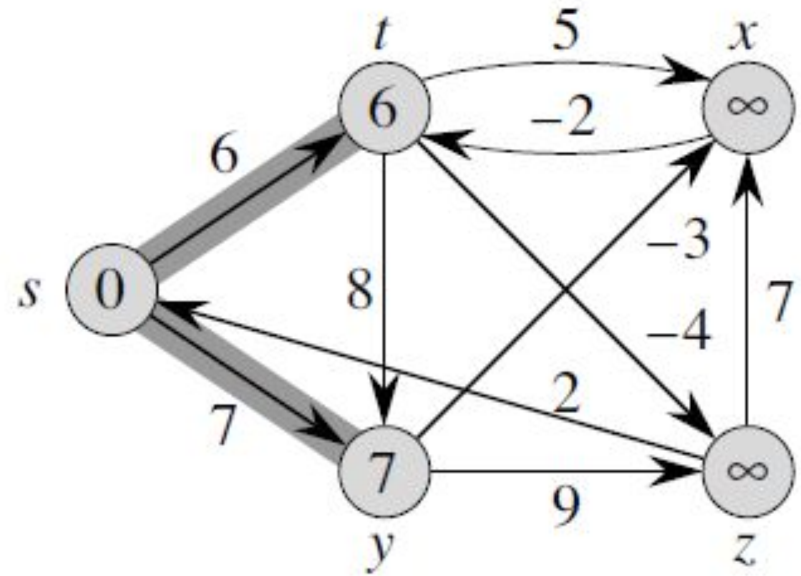
1. Set $shortest[v]$ to ∞ for each vertex v except s, set $shortest[s]$ to 0, and set $pred[v]$ to NULL for each vertex v.
2. For $i = 1$ to $n - 1$:

    A. For each edge (u, v) in E:

        i. Call RELAX(u, v).

The Bellman-Ford algorithm

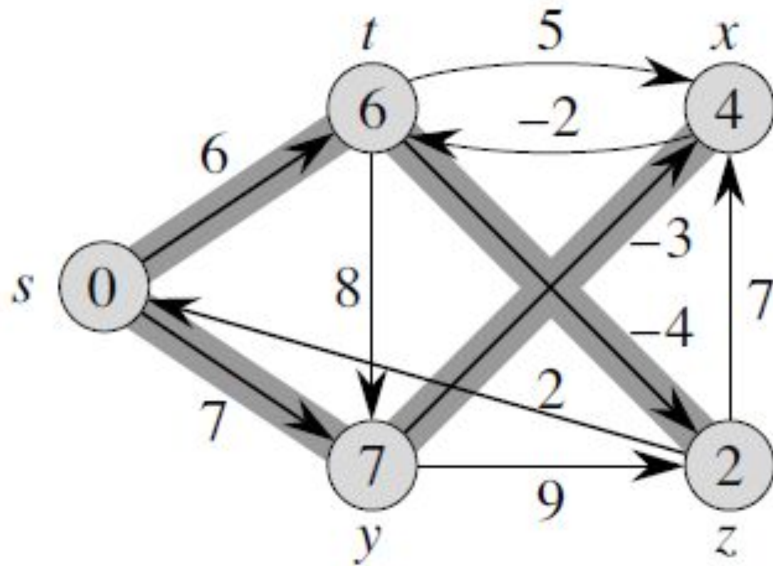




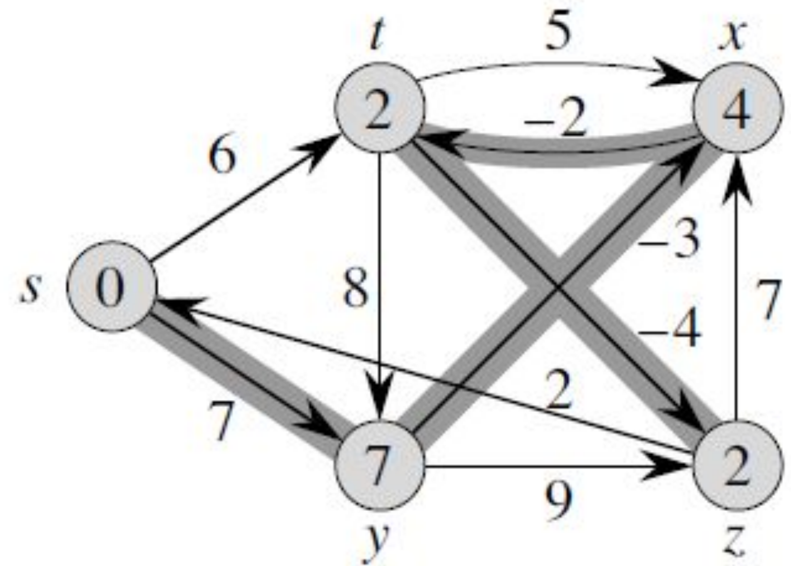
the fixed order: (t; x); (t; y); (t; z); (x; t); (y; x); (y; z); (z; x); (z; s); (s; t); (s; y)

Part (a) shows the situation just before the first pass.

The Bellman-Ford algorithm



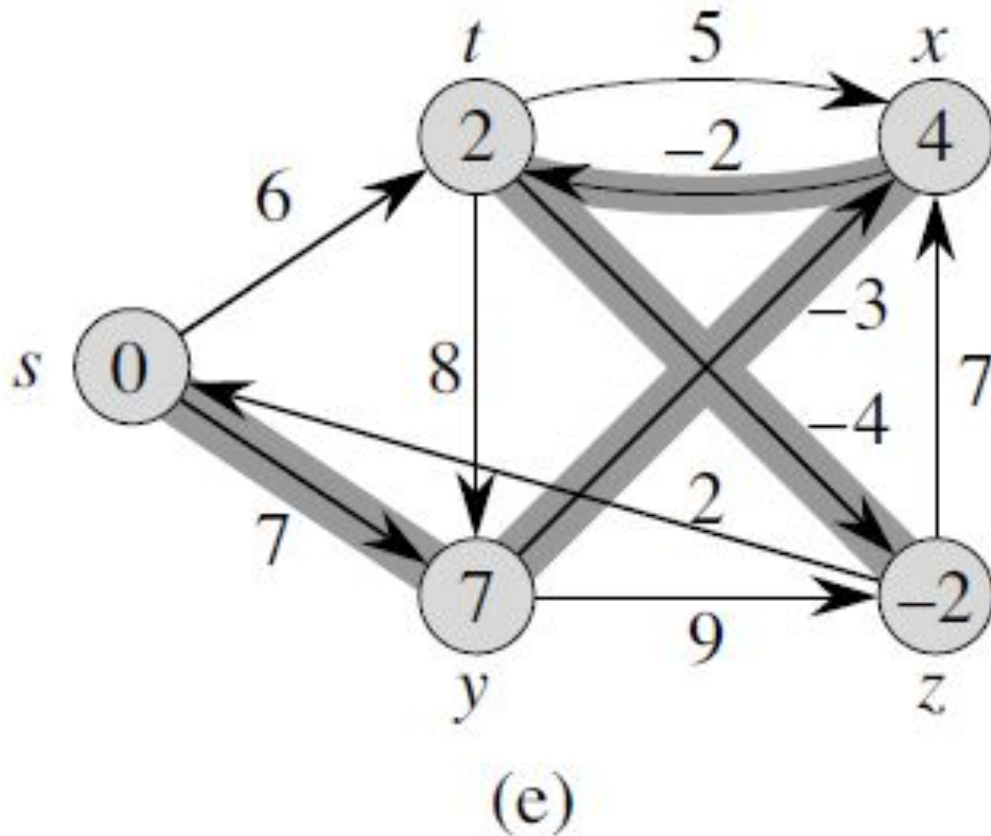
(c)



(d)

Parts (b) through (e) show the situation after each successive pass.

The Bellman-Ford algorithm



(e)

The *shortest* and *pred* values in part (e) are the final values.

The Bellman-Ford algorithm

Consider a shortest path from the source **s** to any vertex **v**.
If we relax the edges, in order, along a shortest path from s to v, then shortest [v] and pred[v] are correct.
If there are not negative-weight cycles on the graph, then there is always a shortest path from s to v that does not contain a cycle.

The Bellman-Ford algorithm

Every acyclic path must contain at most **n - 1** edges. If a path contains **n** edges then it must visit some vertex twice, which would make a cycle.

The first time that step 2A relaxes all edges, it must relax the first edge on this shortest path. The second time that step 2A relaxes all edges, it must relax the second edge on the shortest path, and so on. After the (n – 1) time, all edges on the shortest path have been **relaxed**, in order, and therefore shortest [v] and pred [v] are correct.

The Bellman-Ford algorithm

The graph contains a negative-weight cycle and we have already run the BELLMAN-FORD procedure on it.
=>around and around a negative-weight cycle
=>getting a lower-weight path each time around

That means that there is at least one edge (u; v) on the cycle for which shortest[v] will decrease if you relax it again.

The Bellman-Ford algorithm

How to find a negative-weight cycle, if one exists, after running BELLMAN-FORD?

Go through the edges once again.
If we find an edge (u; v) for which
shortest [u]+weight(u; v) < shortest[v], then

- vertex **v** is either on a negative-weight cycle or
- is reachable from one.

Procedure FIND-NEGATIVE-WEIGHT-CYCLE(G)

Input: G: a directed graph containing a set V of n vertices and a set E of m directed edges with arbitrary weights on which the BELLMAN-FORD procedure has already been run.

Output: Either a list of vertices in a negative-weight cycle, in order, or an empty list if the graph has no negative-weight cycles.

1. Go through all edges to find any edge (u, v) such that $shortest[u] + weight(u, v) < shortest[v]$.
2. If no such edge exists, then return an empty list.
3. Otherwise (there is some edge (u, v) for which $shortest[u] + weight(u, v) < shortest[v]$), do the following:
   - A. Let *visited* be a new array with one element for each vertex. Set all elements of *visited* to FALSE.
   - B. Set x to v.
   - C. While $visited[x]$ is FALSE, do the following:
     - i. Set $visited[x]$ to TRUE.
     - ii. Set x to $pred[x]$
   - D. At this point, we know that x is a vertex on a negative-weight cycle. Set v to $pred[x]$.
   - E. Create a list *cycle* of vertices initially containing just x.
   - F. While v is not x, do the following:
     - i. Insert vertex v at the beginning of *cycle*.
     - ii. Set v to $pred[v]$.
   - G. Return *cycle*.

The Bellman-Ford algorithm

Procedure BELLMAN-FORD(G, s)

Inputs:

- G: a directed graph containing a set V of n vertices and a set E of m directed edges with arbitrary weights.
- s: a source vertex in V.

Result: Same as DIJKSTRA (page 94).

1. Set *shortest*$[v]$ to ∞ for each vertex v except s, set *shortest*$[s]$ to 0, and set *pred*$[v]$ to NULL for each vertex v.
2. For $i = 1$ to $n - 1$:
   A. For each edge (u, v) in E:
      i. Call RELAX(u, v).

The loop of step 2 iterates n - 1 times.

The loop of step 2A iterates m times, once per edge. The total running time is Θ(nm).

The Bellman-Ford algorithm

To find whether a negative-weight cycle exists taking O(m) time.

If there is a negative-weight cycle, it can contain at most n edges, and so the time to trace it out is O(n).

The Bellman-Ford algorithm

Negative-weight cycles relate to ***arbitrage opportunities*** in currency trading.

BITCOIN | BITCOIN ACCEPTANCE | BITCOIN BUSINESS | BITCOIN EDUCATION | BITCOIN SERVICE | NEWS | FINANCIAL | SERVICES

Purse.io Offers Plenty of Bitcoin Arbitrage Opportunities

By **JP Buntinx** - September 13, 2015 785 0

The Bellman-Ford algorithm

n currencies $c_1; c_2; c_3; \ldots ; c_n$,
all the exchange rates between pairs of currencies

with 1 unit of currency $\mathbf{c_i}$ we can buy $\mathbf{r_{ij}}$ units of currency $\mathbf{c_j}$.
$\mathbf{r_{ij}}$ is the exchange rate between currencies $\mathbf{c_i}$ and $\mathbf{c_j}$.
both i and j range from 1 to n

!!!$\mathbf{r_{ii}} = 1$ for each currency $\mathbf{c_i}$

The Bellman-Ford algorithm

An arbitrage opportunity would correspond to a sequence of k currencies

$$< c_{j1};\ c_{j2};\ c_{j3};\ :::\ ;c_{jk} >$$

such that when you multiply out the exchange rates, you get a product strictly greater than 1:

$$r_{j1,j2} \cdot r_{j2,j3} \cdot \ \ldots \cdot \ r_{jk-1,jk} \ \cdot \ r_{jk,j1} > 1$$

The Bellman-Ford algorithm

lg(x * y) = lg x + lg y

$$r_{j1,j2} \cdot r_{j2,j3} \cdot \ldots \cdot r_{jk-1,jk} \cdot r_{jk,j1} > 1$$

$$lgr_{j1,j2} + lgr_{j2,j3} + \ldots + lgr_{jk-1,jk} + lgr_{jk,j1} > 0$$

Or, negating both sides of this inequality

$$(-lgr_{j1,j2}) + (-lgr_{j2,j3}) + \ldots + (-lgr_{jk-1,jk}) + (-lgr_{jk,j1}) < 0$$

The Bellman-Ford algorithm

To find an arbitrage opportunity, if one exists,

- construct a directed graph with one vertex $\mathbf{v}_i$ for each currency $\mathbf{c}_i$.
- For each pair of currencies $\mathbf{c}_i$ and $\mathbf{c}_j$, create directed edges $(\mathbf{v}_i; \mathbf{v}_j)$ and $(\mathbf{v}_j; \mathbf{v}_i)$ with weights $\mathbf{-lg\ r}_{ij}$ and $\mathbf{-lg\ r}_{ji}$, respectively.

The Bellman-Ford algorithm

- Add a new vertex **s** with a 0-weight edge (**s; v_i**) to each of the vertices v_1 through v_n.
- Run the Bellman-Ford algorithm on this graph with **s** as the source vertex, and
- use the result to determine whether it contains a negative-weight cycle.

If it does, then the vertices on that cycle correspond to the currencies in an arbitrage opportunity.

The Floyd-Warshall algorithm

The classic example of all-pairs shortest paths is the table of a road atlas giving distances between several cities.
You find the row for one city, you find the column for the other city, and the distance between them lies at the intersection of the row and column.

The Floyd-Warshall algorithm

There is one problem with this example: it's not all-pairs.

If it were all pairs, the table would have one row and one column for every intersection, not for just every city.

The number of rows and columns for just the one country would be in the millions.

The Floyd-Warshall algorithm

What would be a rightful application of all-pairs shortest paths?
Finding the **diameter** of a network: the longest of all shortest paths.
For example, suppose that a directed graph represents a communication network, and the weight of an edge gives the time it takes for a message to traverse a communication link.
Then the diameter gives you the longest possible transit time for a message in the network.

The Floyd-Warshall algorithm

Using the Floyd-Warshall algorithm, we can solve the all-pairs problem in $\Theta(n^3)$ time.

For the Floyd-Warshall algorithm the graph can have negative-weight edges but no negative-weight cycles.

The Floyd-Warshall algorithm illustrates a clever algorithmic technique called “dynamic programming”.

The Floyd-Warshall algorithm

The Floyd-Warshall algorithm relies on one property of shortest paths.

If a shortest path, call it p, from vertex u to vertex v goes from vertex u to vertex x to vertex y to vertex v, then the portion of p that is between x and y is itself a shortest path from x to y.

That is, any subpath of a shortest path is itself a shortest path.

The Floyd-Warshall algorithm

the vertices are numbered from 1 to n
Vertex numbers become important.

shortest[u; v; x] is the weight of a shortest path from vertex u to vertex v in which each intermediate vertex – a vertex on the path other than u and v – is numbered from 1 to x

(u, v, and x as integers in the range 1 to n that represent vertices)

The Floyd-Warshall algorithm

Let's consider two vertices **u** and **v**.

Pick a number **x** in the range from **1** to **n**.

Consider all paths from **u** to **v** in which all intermediate vertices are numbered at most **x**.
Of all these paths, let path **p** be one with <u>minimum weight</u>.
Path **p** either contains vertex **x** or it does not.

The Floyd-Warshall algorithm

There are two possibilities:
First possibility: x is not an intermediate vertex in path p. Then all intermediate vertices of path p are numbered at most x - 1. What does this mean?
Shortest[u; v; x] equals shortest[u; v; x – 1].
Second possibility: x appears as an intermediate vertex in path p. Then
shortest[u; v; x] equals shortest[u; x; x – 1] +
+ shortest[x; v; x – 1].

The Floyd-Warshall algorithm

adjacency-matrix representation

The entry for edge (u; v) holds the weight of the edge, with a weight of ∞ indicating that the edge is absent. Shortest[u; v; 0] denotes the weight of a shortest path from u to v with all intermediate vertices numbered at most 0, such a path has no intermediate vertices.

The Floyd-Warshall algorithm

computes shortest[u; v; x] values => x=1
computes shortest[u; v; x] values => x=2
computes shortest[u; v; x] values => x=2
…
computes shortest[u; v; x] values => x=n

pred[u; v; x] as the predecessor of vertex v on a shortest path from vertex u in which all intermediate vertices are numbered at most x

Procedure FLOYD-WARSHALL(G)

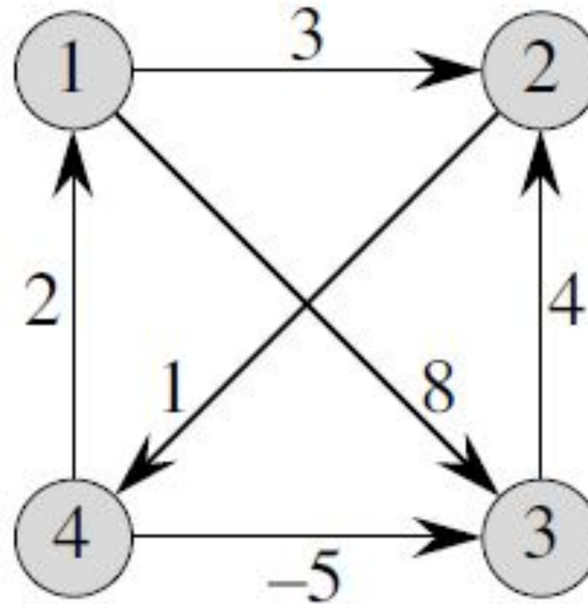
Input: G: a graph represented by a weighted adjacency matrix W with n rows and n columns (one row and one column per vertex). The entry in row u and column v, denoted w_{uv}, is the weight of edge (u, v) if this edge is present in G, and it is ∞ otherwise.

Output: For each pair of vertices u and v, the value of $shortest[u, v, n]$ contains the weight of a shortest path from u to v, and $pred[u, v, n]$ is the predecessor vertex of v on a shortest path from u.

1. Let *shortest* and *pred* be new $n \times n \times (n + 1)$ arrays.
2. For each u and v from 1 to n:
   A. Set $shortest[u, v, 0]$ to w_{uv}.
   B. If (u, v) is an edge in G, then set $pred[u, v, 0]$ to u. Otherwise, set $pred[u, v, 0]$ to NULL.
3. For $x = 1$ to n:
   A. For $u = 1$ to n:
      i. For $v = 1$ to n:
         a. If $shortest[u, v, x] < shortest[u, x, x - 1] + shortest[x, v, x - 1]$, then set $shortest[u, v, x]$ to $shortest[u, x, x - 1] + shortest[x, v, x - 1]$ and set $pred[u, v, x]$ to $pred[x, v, x - 1]$.
         b. Otherwise, set $shortest[u, v, x]$ to $shortest[u, v, x - 1]$ and set $pred[u, v, x]$ to $pred[u, v, x - 1]$.
4. Return the *shortest* and *pred* arrays.

The Floyd-Warshall algorithm

- example



$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix}$$

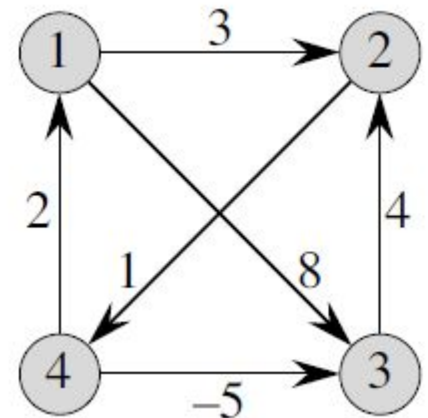
shortest[2; 4; 0] is 1, because we can get from vertex 2 to vertex 4 directly, with no intermediate vertices, by taking edge (2; 4) with weight 1

The Floyd-Warshall algorithm

- pred[u; v; 0] values

$$\begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}$$

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix}$$

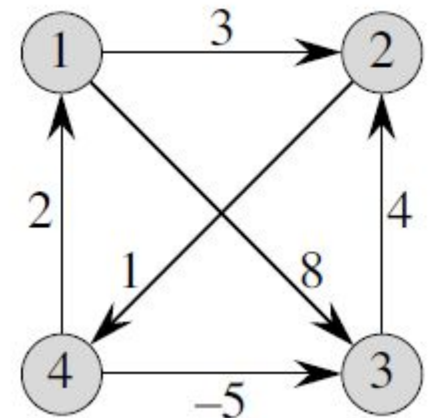


The Floyd-Warshall algorithm

- After running the loop for x=1

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}$$

3. For $x = 1$ to n:
   A. For $u = 1$ to n:
      i. For $v = 1$ to n:
         a. If $shortest[u, v, x] < shortest[u, x, x-1] + shortest[x, v, x-1]$, then set $shortest[u, v, x]$ to $shortest[u, x, x-1] + shortest[x, v, x-1]$ and set $pred[u, v, x]$ to $pred[x, v, x-1]$.
         b. Otherwise, set $shortest[u, v, x]$ to $shortest[u, v, x-1]$ and set $pred[u, v, x]$ to $pred[u, v, x-1]$.



The Floyd-Warshall algorithm

- After running the loop for x=2

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}$$

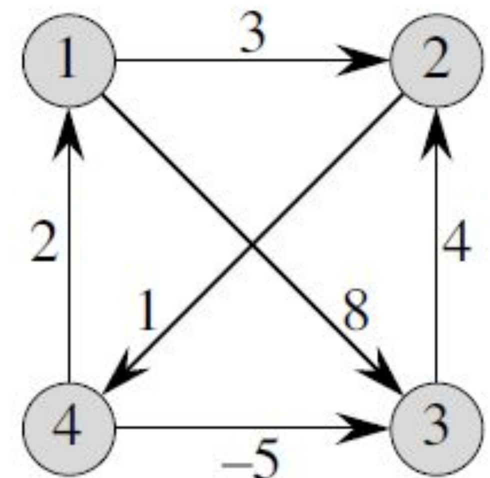
- After running the loop for x=3

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}$$

The Floyd-Warshall algorithm

- shortest[u; v; 4] and pred[u; v; 4] values

$$\begin{pmatrix} 0 & 3 & -1 & 4 \\ 3 & 0 & -4 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} \text{NULL} & 1 & 4 & 2 \\ 4 & \text{NULL} & 4 & 2 \\ 4 & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}$$



The Floyd-Warshall algorithm

dynamic programming

This technique applies only when:

1. we are trying to find an optimal solution to a problem,
2. we can break an instance of the problem into instances of one or more subproblems,
3. we use solutions to the subproblem(s) to solve the original problem, and
4. if we use a solution to a subproblem within an optimal solution to the original problem, then the subproblem solution we use must be optimal for the subproblem.