Floyd–Warshall Algorithm

All Pairs Shortest Path (APSP)

William Fiset

FW algorithm overview

In graph theory, the **Floyd-Warshall (FW)** algorithm is an **All-Pairs Shortest Path (APSP)** algorithm. This means it can find the shortest path between all pairs of nodes.

The time complexity to run FW is $O(V^3)$ which is **ideal for graphs no larger than a couple hundred nodes**.

Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	$O(V+E)$	$O((V+E)\log V)$	$O(VE)$	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

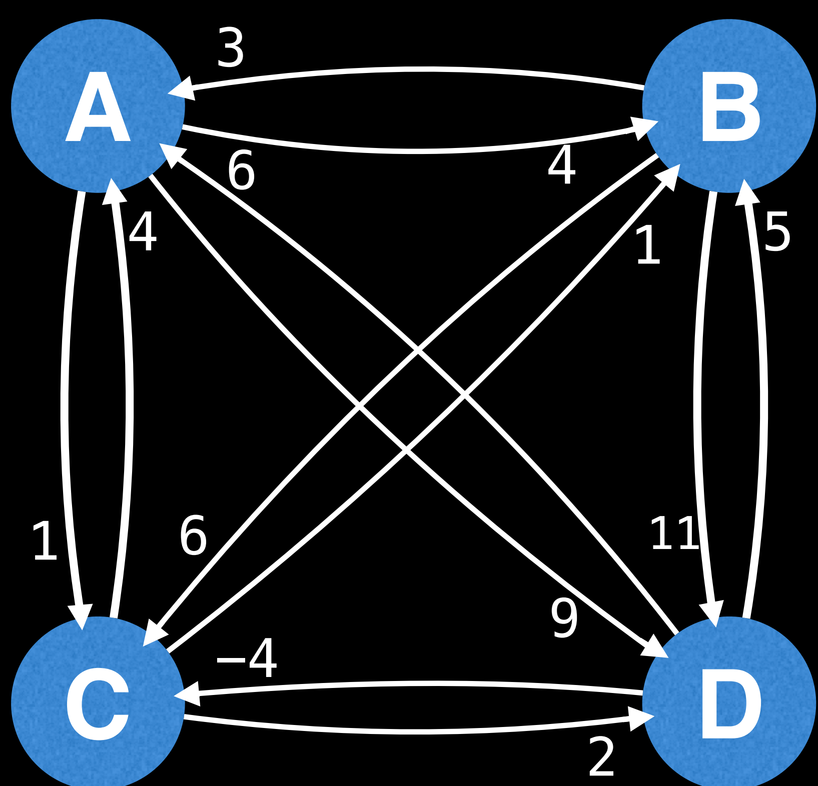
Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	O(V^3)
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

Graph setup

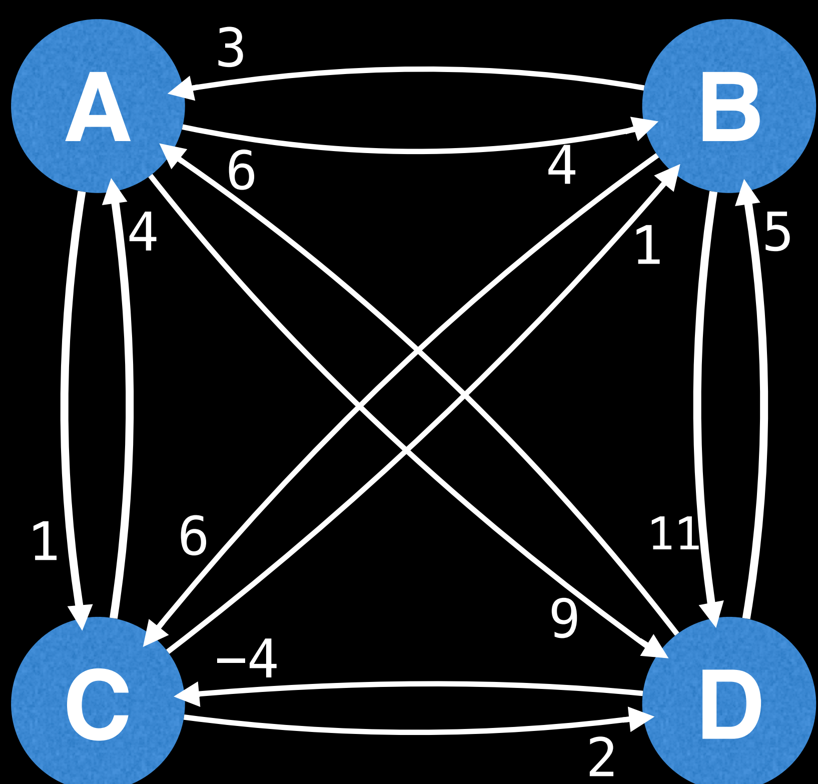
With FW, the optimal way to **represent our graph is with a 2D adjacency matrix *m*** where cell m[i][j] represents the edge weight of going from node i to node j.



$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & 0 & 4 & 1 & 9 \\
B & 3 & 0 & 6 & 11 \\
C & 4 & 1 & 0 & 2 \\
D & 6 & 5 & -4 & 0
\end{array}
$$

Graph setup

With FW, the optimal way to **represent our graph is with a 2D adjacency matrix *m*** where cell m[i][j] represents the edge weight of going from node i to node j.

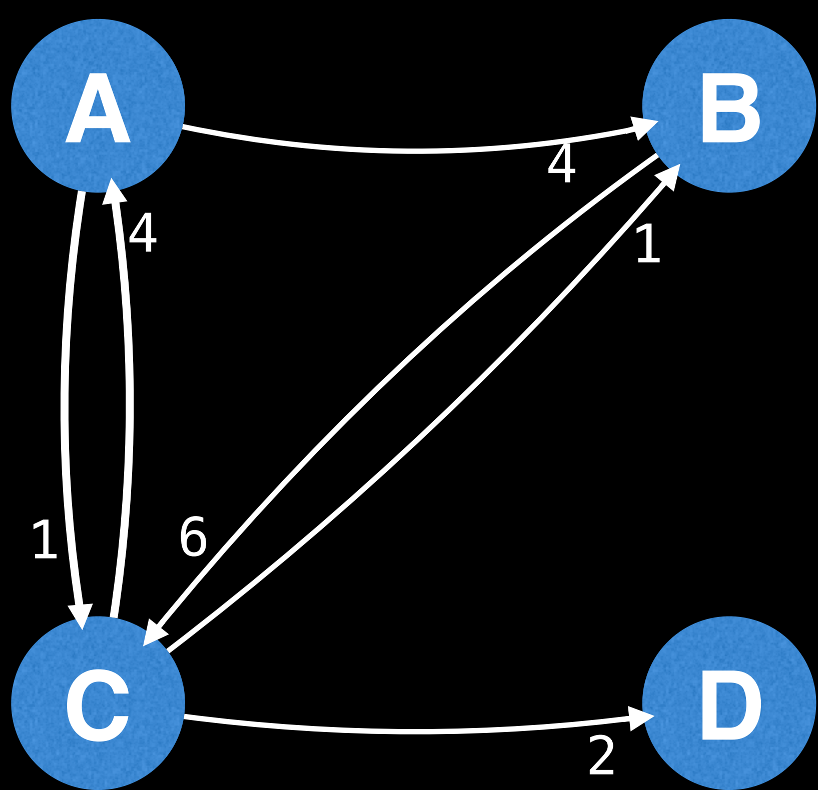


$$
\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}
\begin{array}{c}
\begin{array}{cccc} A & B & C & D \end{array} \\
\begin{bmatrix}
0 & 4 & 1 & 9 \\
3 & 0 & 6 & 11 \\
4 & 1 & 0 & 2 \\
6 & 5 & -4 & 0
\end{bmatrix}
\end{array}
$$

NOTE: In the graph above, it is assumed that **the distance from a node to itself is zero**. This is why the diagonal is all zeros.

Graph setup

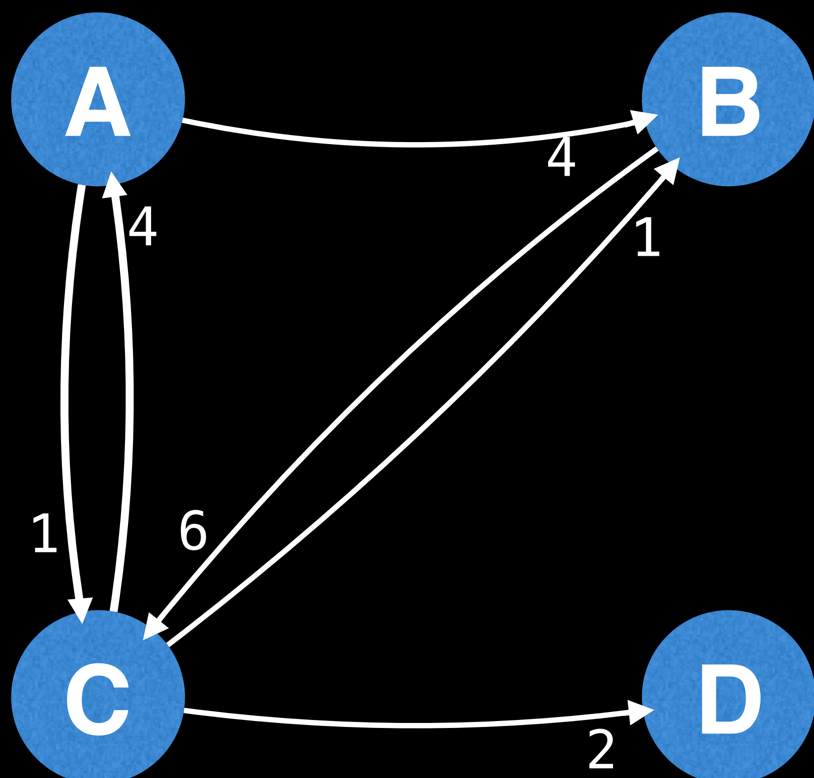
If there is no edge from node i to node j then set the edge value for m[i][j] to be positive infinity.



$$
\begin{array}{c|cccc|}
 & A & B & C & D \\
\hline
A & 0 & 4 & 1 & \infty \\
B & \infty & 0 & 6 & \infty \\
C & 4 & 1 & 0 & 2 \\
D & \infty & \infty & \infty & 0 \\
\end{array}
$$

Graph setup

If there is no edge from node i to node j then set the edge value for m[i][j] to be positive infinity.

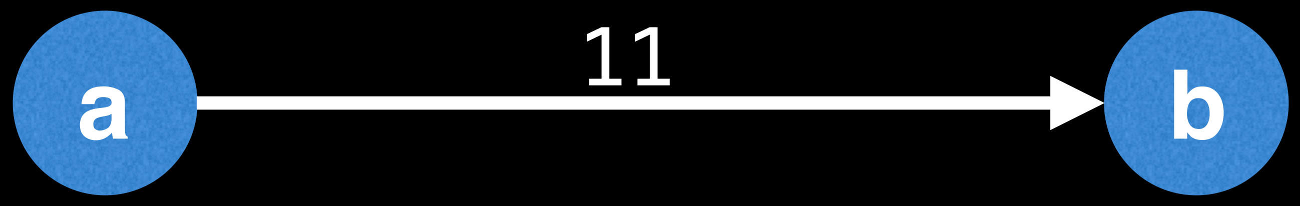


$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & 0 & 4 & 1 & \infty \\
B & \infty & 0 & 6 & \infty \\
C & 4 & 1 & 0 & 2 \\
D & \infty & \infty & \infty & 0
\end{array}
$$

IMPORTANT: If your programming language does not support a special constant for $+\infty$ such that $\infty + \infty = \infty$ and $x + \infty = \infty$ then **avoid using $2^{31}-1$ as infinity**! This will cause integer overflow; prefer to use a large constant such as 10^7 instead.

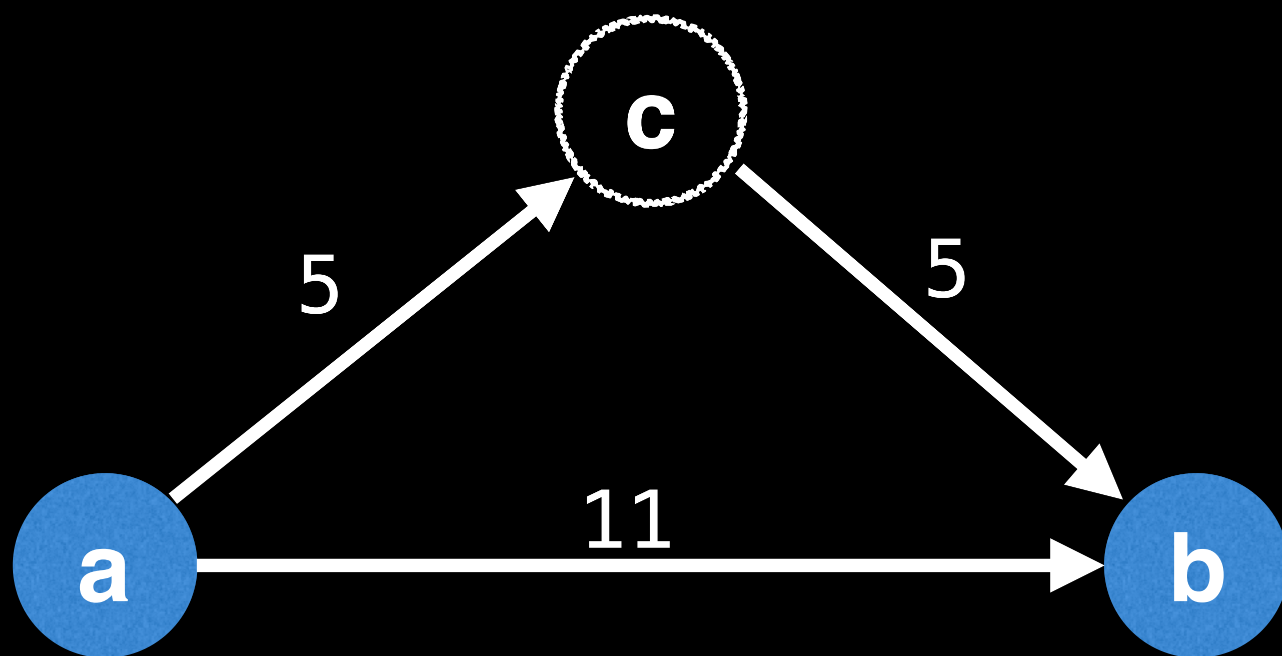
The main idea behind the Floyd–Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.

The main idea behind the Floyd-Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.



Suppose our adjacency matrix tells us that the distance from **a** to **b** is: **m[a][b] = 11**

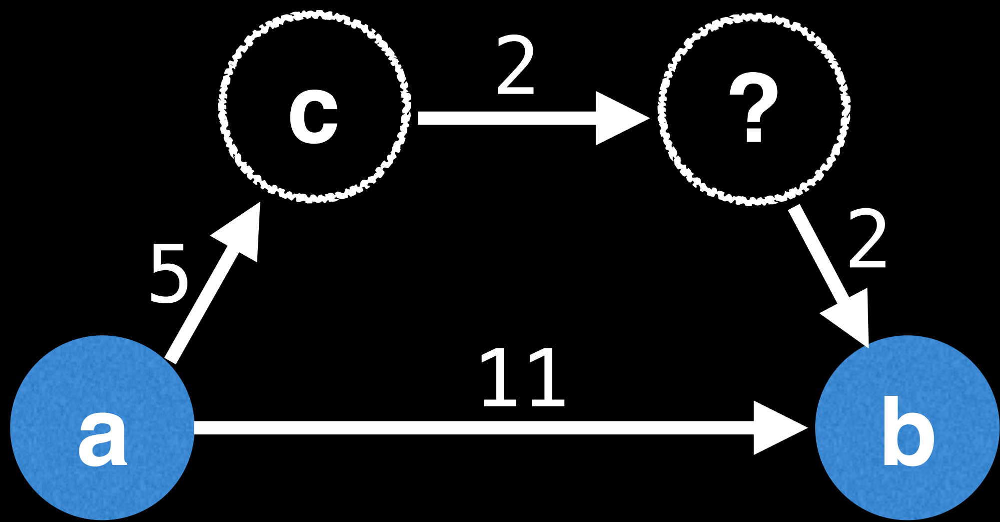
The main idea behind the Floyd–Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.



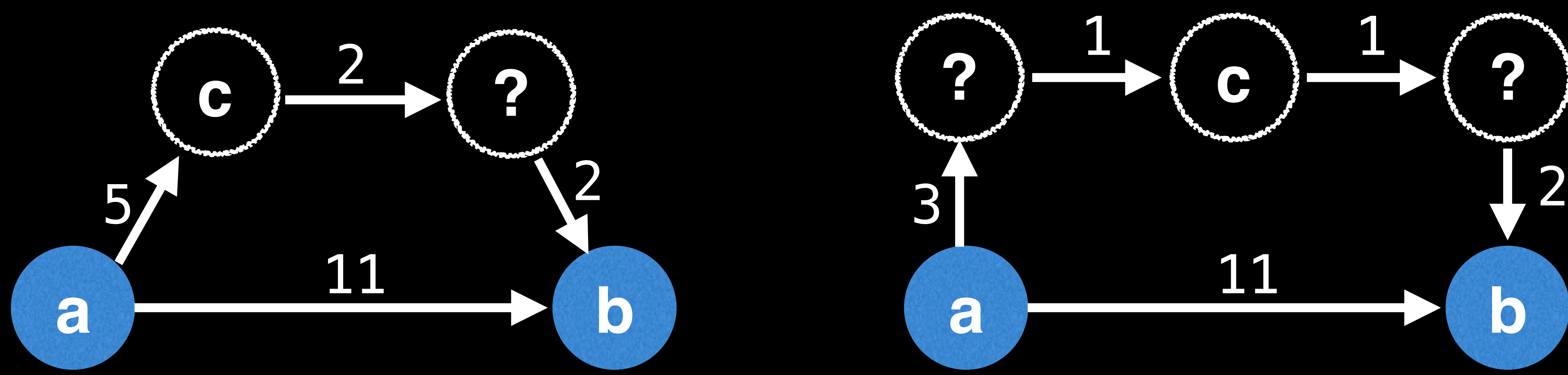
Suppose there exists a third node, **c**. If **m[a][c] + m[c][b] < m[a][b]** then it’s better to route through **c**!

The goal of Floyd–Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.

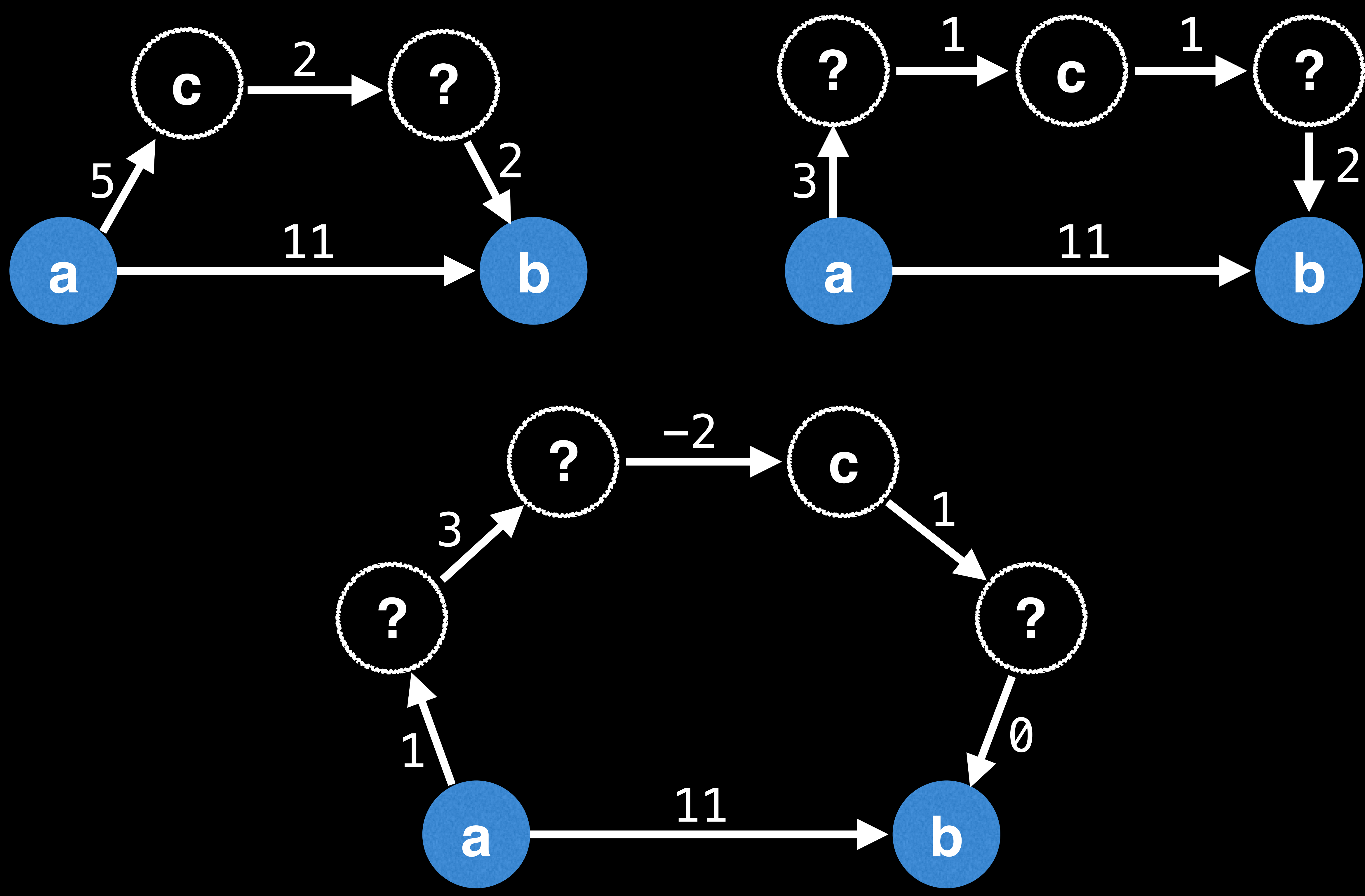
The goal of Floyd–Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.



The goal of Floyd–Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.



The goal of Floyd-Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.



The Memo Table

Let ‘dp’ (short for Dynamic Programming) be a 3D matrix of size n x n x n that acts as a memo table.

dp[k][i][j] = shortest path from i to j routing through nodes {0,1,…,k-1,k}

Start with k = 0, then k = 1, then k = 2, … This gradually builds up the optimal solution routing through 0, then all optimal solutions routing through 0 and 1, then all optimal solutions routing through 0, 1, 2, etc… up until n-1 which stores to APSP solution.

Specifically dp[n-1] is the 2D matrix solution we’re after.

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

dp[k][i][j] = m[i][j] if k = 0

otherwise:

dp[k][i][j] = **min**(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])

Reuse the best distance from i to j with values routing through nodes {0,1,…,k-1}

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

dp[k][i][j] = m[i][j] if k = 0

otherwise:

$$dp[k][i][j] = \mathbf{min}(dp[k-1][i][j],\ \underbrace{dp[k-1][i][k]+dp[k-1][k][j]}_{})$$

Find the best distance from i to j through node k reusing best solutions from {0,1,…,k-1}

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

The right side of the min function in english essentially says: “go from i to k” and then “go from k to j”

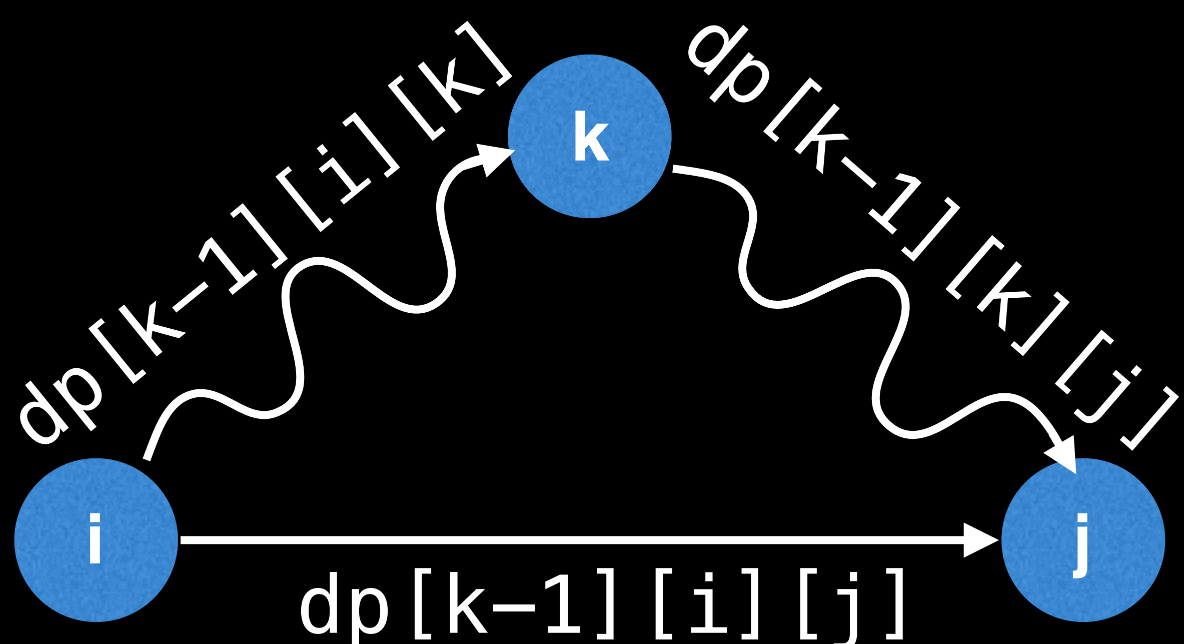
In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

dp[k][i][j] = m[i][j] if k = 0

otherwise:

dp[k][i][j] = **min**(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])

Visually this looks like:



Currently we’re using **$O(V^3)$** memory since our memo table ‘dp’ has one dimension for each of k, i and j.

Notice that we will be looping over k starting from 0, then 1, 2… and so fourth. The important thing to note here is that previous result builds off the last since we need state k-1 to compute state k. With that being said, it is possible to **compute the solution for k in-place** saving us a dimension of memory and reducing the space complexity to **$O(V^2)$**!

Currently we're using **$O(V^3)$** memory since our memo table 'dp' has one dimension for each of k, i and j.

Notice that we will be looping over k starting from 0, then 1, 2… and so fourth. The important thing to note here is that previous result builds off the last since we need state k−1 to compute state k. With that being said, it is possible to **compute the solution for k in-place** saving us a dimension of memory and reducing the space complexity to **$O(V^2)$**!

The new recurrence relation is:

$$\boxed{\begin{array}{c} dp[i][j] = m[i][j] \text{ if } k = 0 \\ \text{otherwise:} \\ dp[i][j] = \min(dp[i][j],\ dp[i][k]+dp[k][j]) \end{array}}$$

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

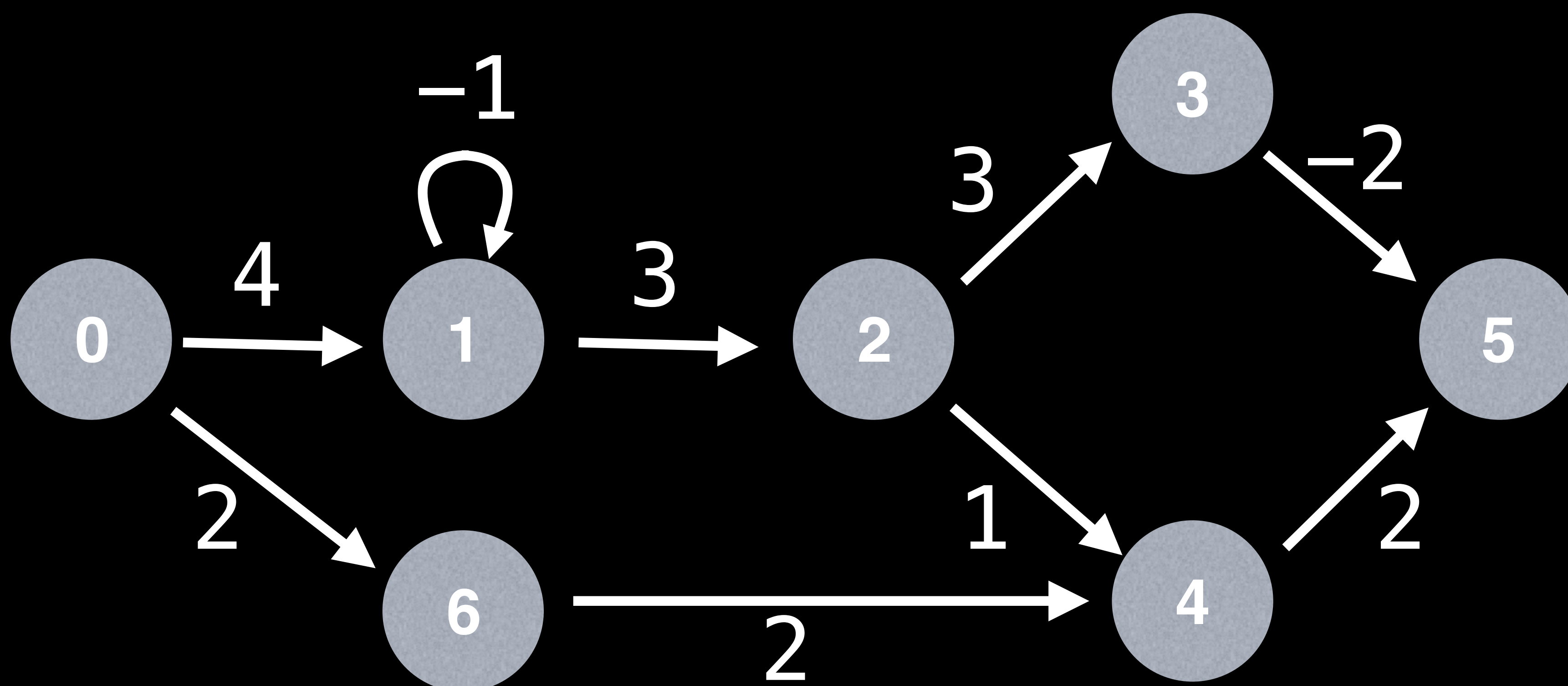
```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

Negative Cycles

What do we mean by a negative cycle?

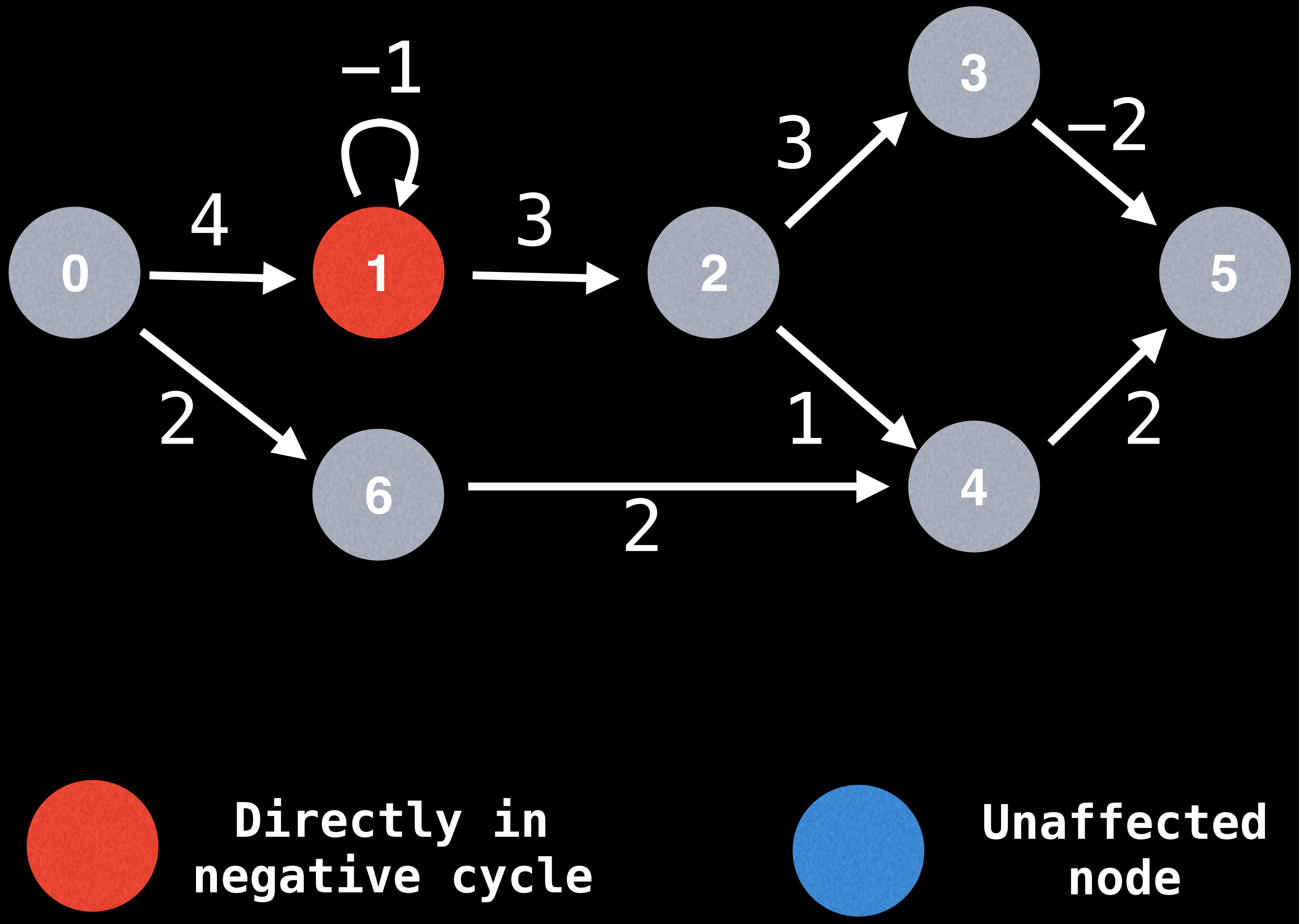
Negative Cycles

Negative cycles can manifest themselves in many ways…



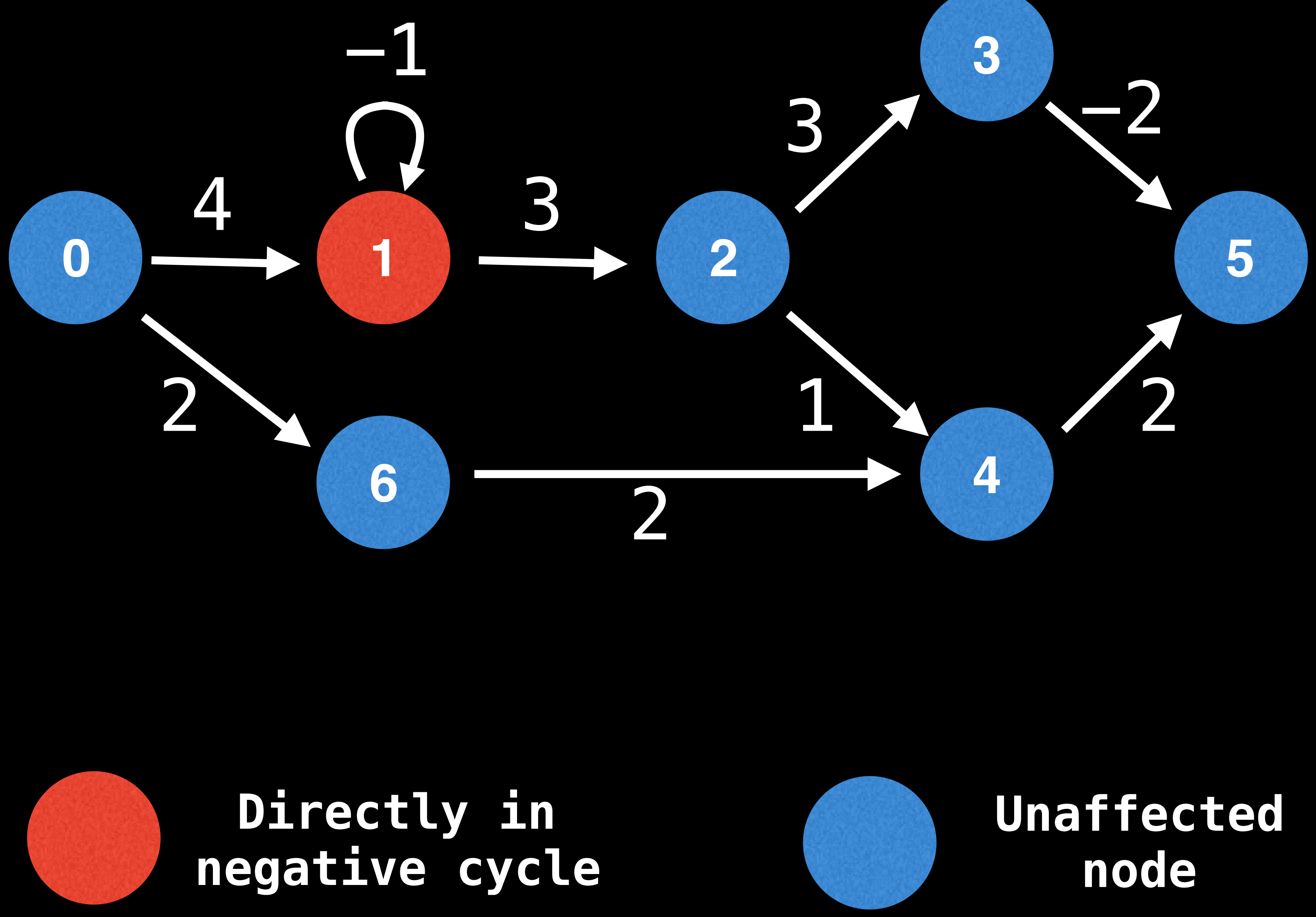
Negative Cycles

Negative cycles can manifest themselves in many ways…



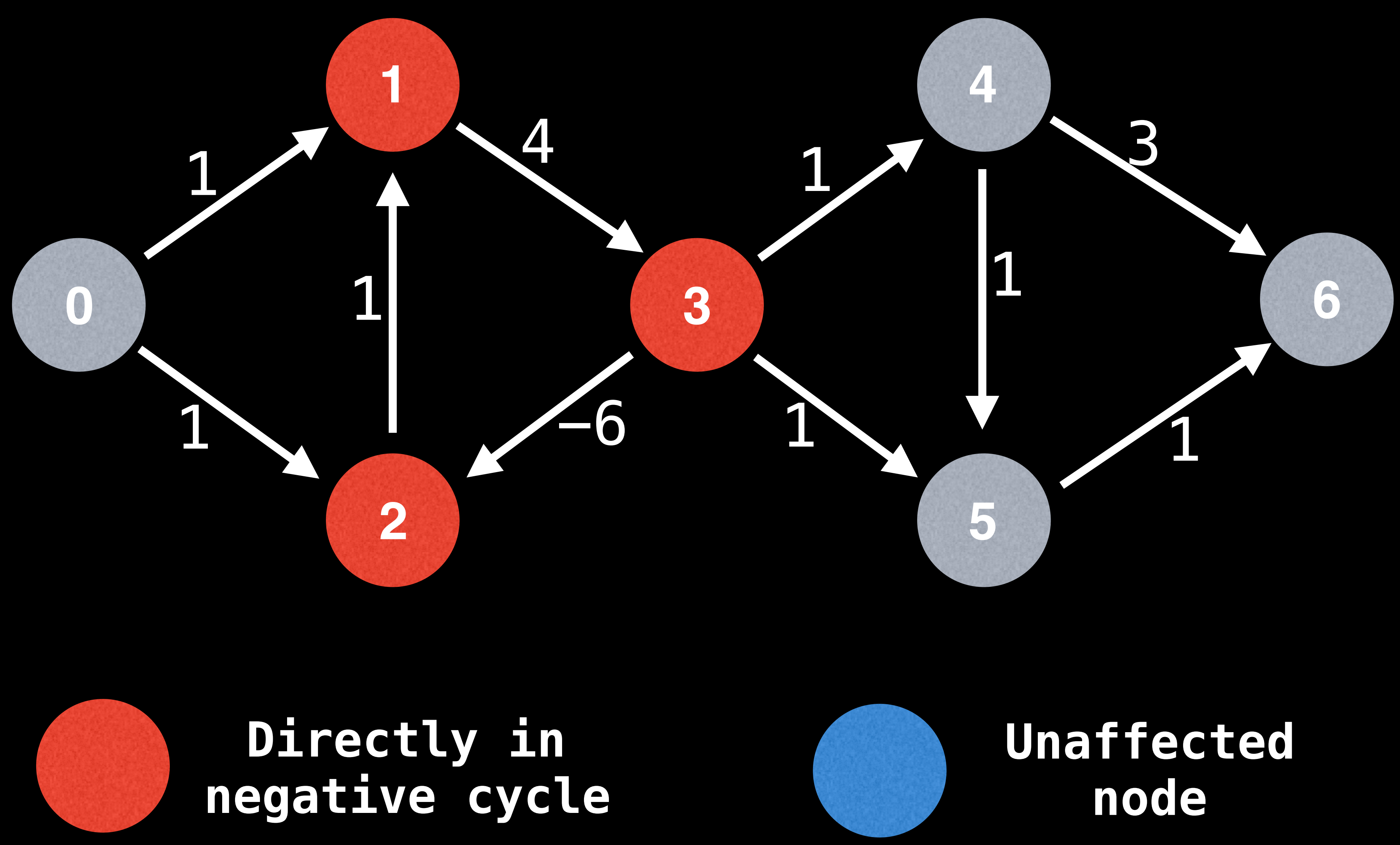
Negative Cycles

Negative cycles can manifest themselves in many ways…



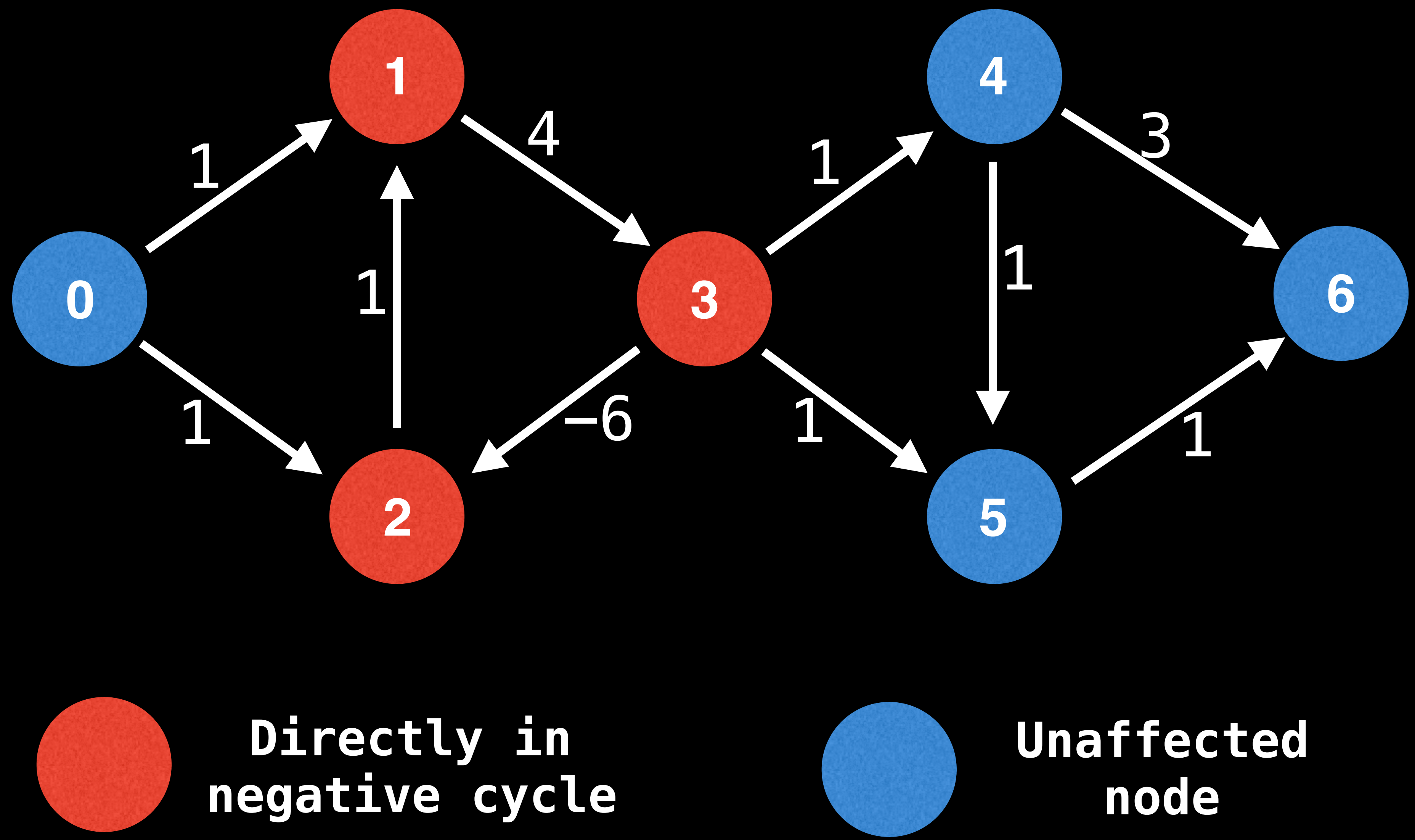
Negative Cycles

Negative cycles can manifest themselves in many ways…



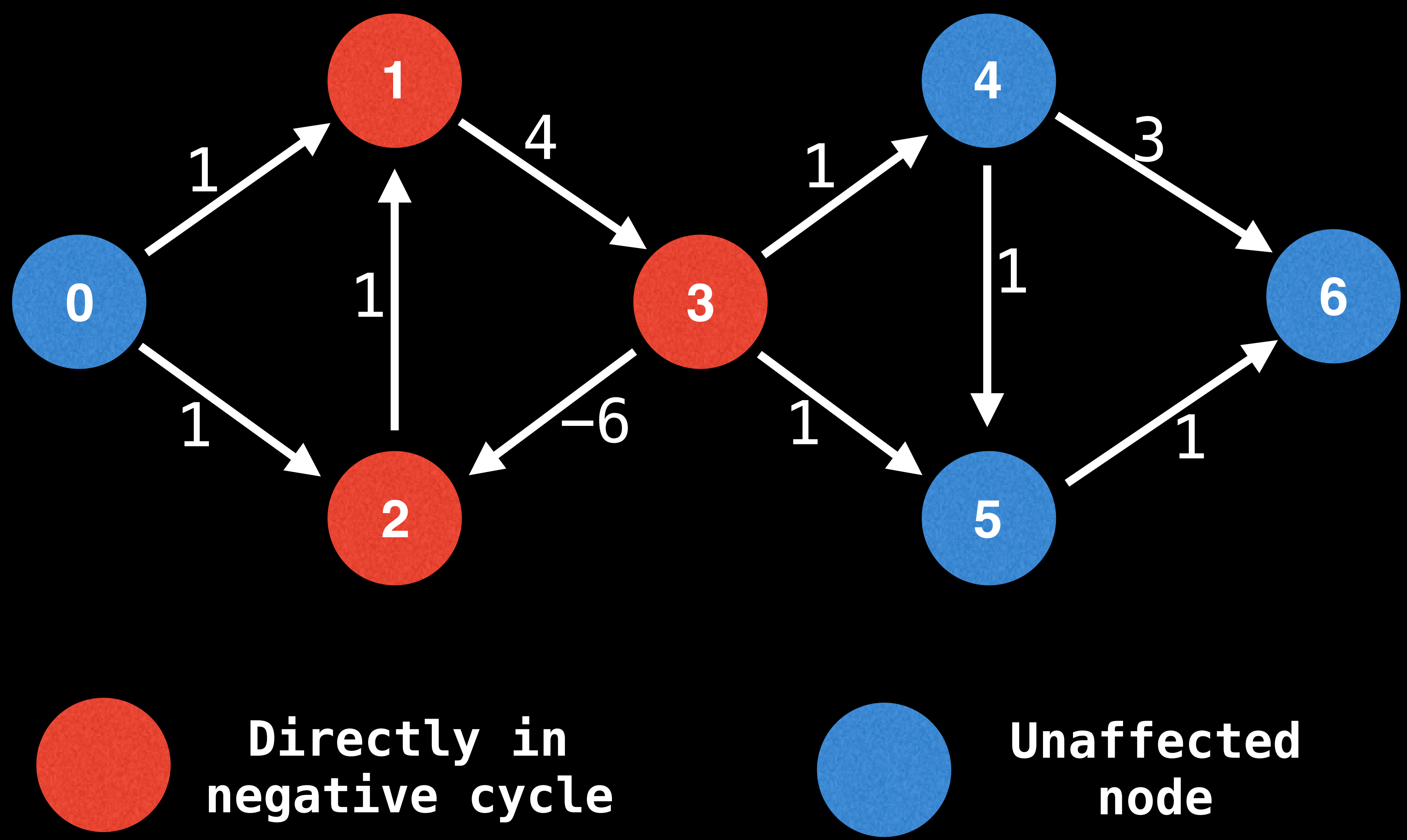
Negative Cycles

Negative cycles can manifest themselves in many ways…



Negative Cycles

The important thing to ask ourselves is does the optimal path from node i to node j go through a **red** node? If so the path is affected by the negative cycle and is compromised.



```
function propagateNegativeCycles(dp, n):

  # Execute FW APSP algorithm a second time but
  # this time if the distance can be improved
  # set the optimal distance to be -∞.
  # Every edge (i, j) marked with -∞ is either
  # part of or reaches into a negative cycle.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = -∞
          next[i][j] = -1
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Reconstructs the shortest path between nodes
# ’start’ and ‘end’. You must run the
# floydWarshall solver before calling this method.
# Returns null if path if affected by negative cycle.
function reconstructPath(start, end):
  path = []
  # Check if there exists a path between
  # the start and the end node.
  if dp[start][end] == +∞: return path

  at := start
  # Reconstruct path from next matrix
  for(;at != end; at = next[at][end]):
    if at == -1: return null
    path.add(at)

  if next[at][end] == -1: return null
  path.add(end)
  return path
```

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description

Next Video: Floyd–Warshall source code

Floyd–Warshall Algorithm source code

William Fiset

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description