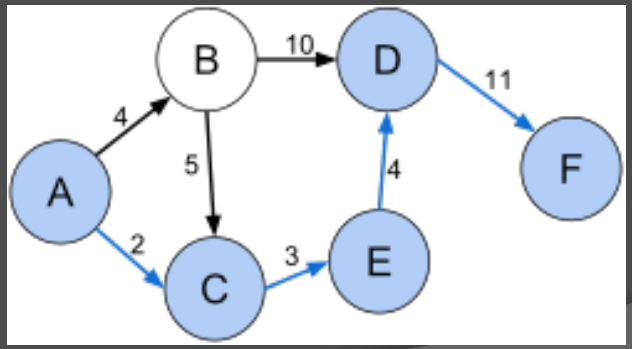
Advanced Computer Contest Preparation

Lecture 10

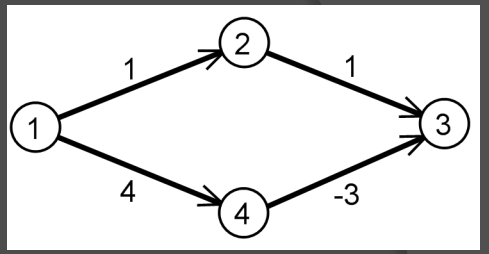
SSSP WITH NEGATIVE EDGES
ALL PAIRS SHORTEST PATH

Review: SSSP

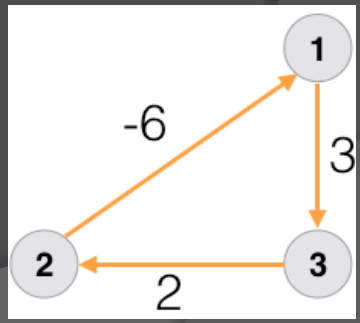
- Single Source Shortest Path
- A problem that requires us to find the distance from one node to any other node
- We know Dijkstra's algorithm to solve the single source shortest path problem



New Graph Terminology



- Negative Edge
  - An edge with a negative weight
- Negative Cycle
  - A cycle in which the sum of the weights of the cycle is negative
  - This cycle can be infinitely looped to lower cost
  - Shortest paths cannot be computed properly when negative cycles exist
- Undirected graphs cannot have negative edges since a negative cycle will always exist



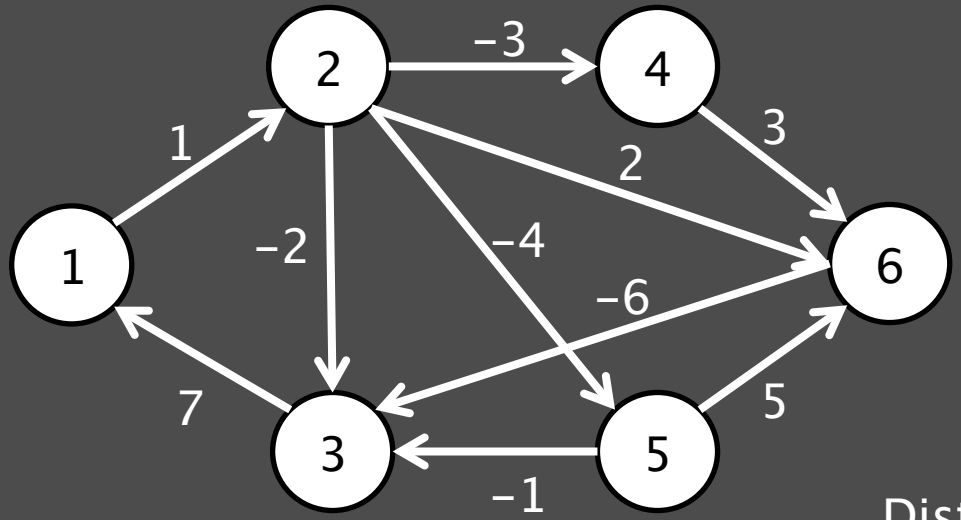
An Algorithm For Graphs with Negative Edges

- Limitation of Dijkstra's algorithm is that it cannot handle graphs with negative edges
- Bellman-Ford Algorithm can solve SSSP where the graph has negative edges
- Bellman-Ford Algorithm can also detect negative cycles

Bellman-Ford Algorithm

- **Principle of Relaxation**
  - Approximation of correct shortest distance that is gradually replaced with a more accurate, lower distance
  - Estimate is never less than the answer
  - Dijkstra's algorithm uses this principle when we try to update neighbors of the current node
- Example: `dist[nxt] = min(dist[nxt],dist[cur]+cost)`
- Bellman-Ford simply relaxes all edges $V-1$ times
- Order of edge relaxation does not matter

Bellman-Ford – Diagram



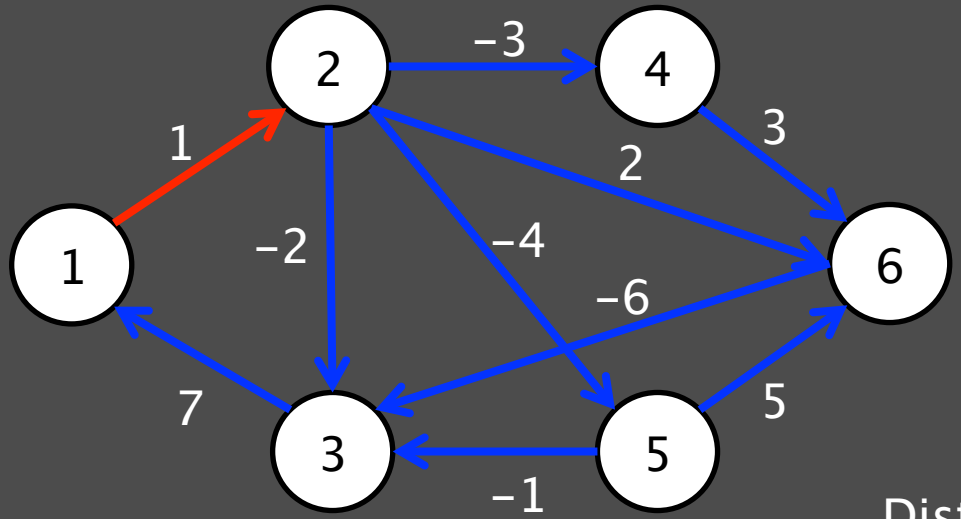
Before Iterations

Dist

1	2	3	4	5	6
0	∞	∞	∞	∞	∞

Bellman-Ford – Diagram

Red edges signify successful relaxations



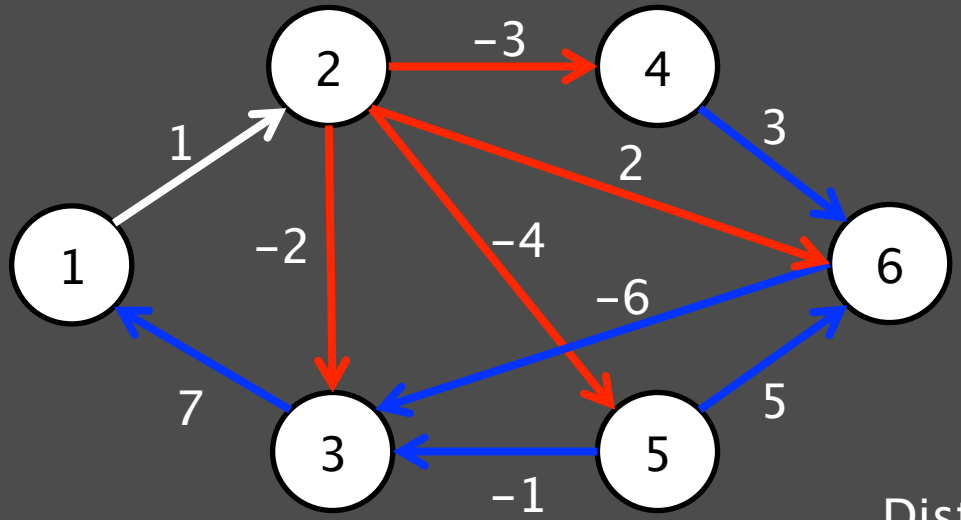
Iteration 1

Blue edges signify that we have already attempted to relax it in the current iteration

Dist

1	2	3	4	5	6
0	1	∞	∞	∞	∞

Bellman-Ford – Diagram

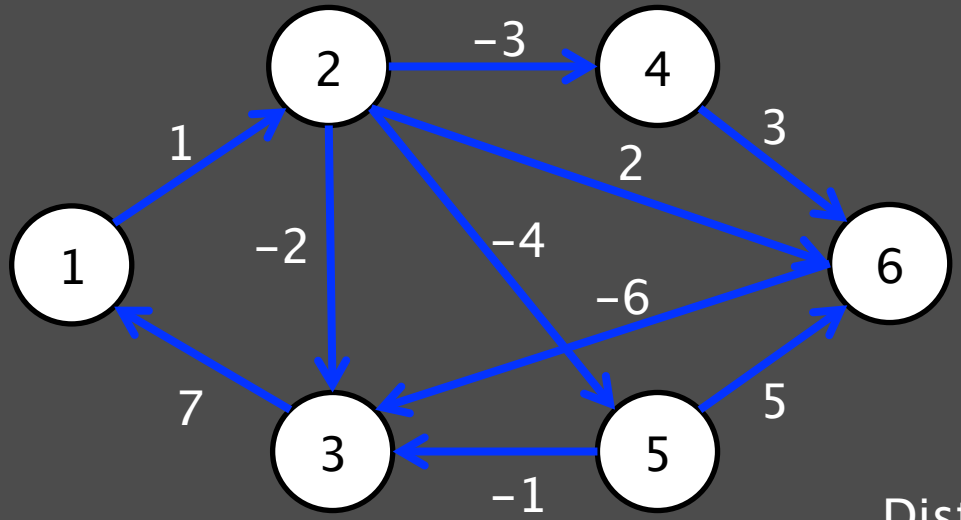


Iteration 2

Dist

1	2	3	4	5	6
0	1	−1	−2	−3	3

Bellman-Ford – Diagram

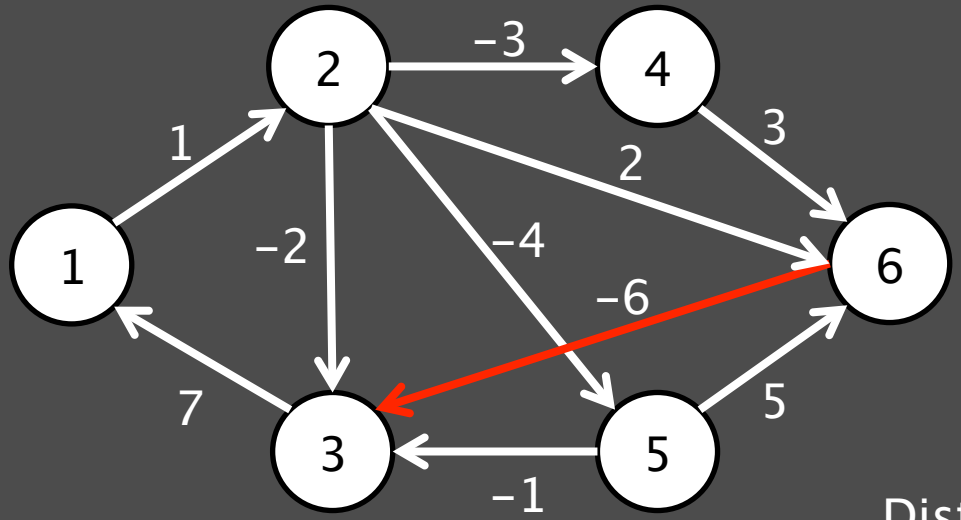


Iteration 2

Dist

1	2	3	4	5	6
0	1	−1	−2	−3	3

Bellman-Ford – Diagram

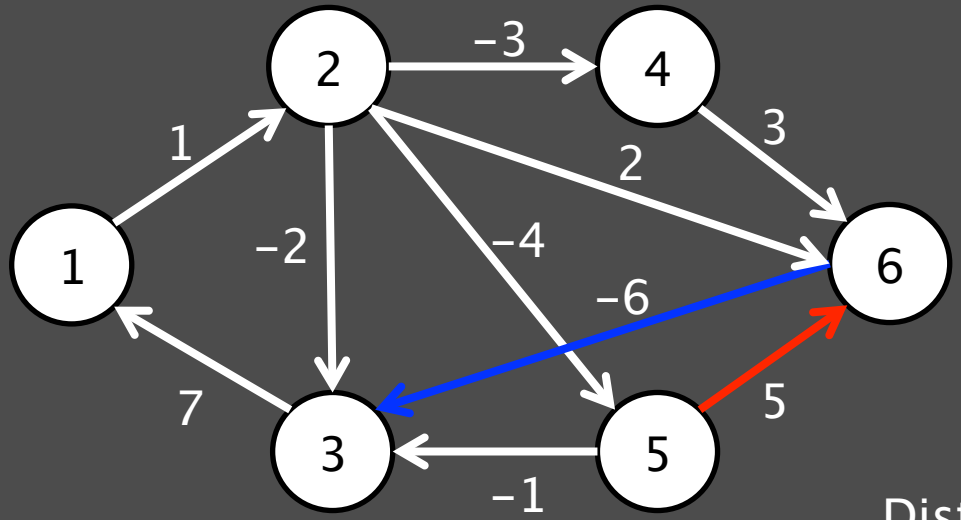


Iteration 3

Dist

1	2	3	4	5	6
0	1	−3	−2	−3	3

Bellman-Ford – Diagram

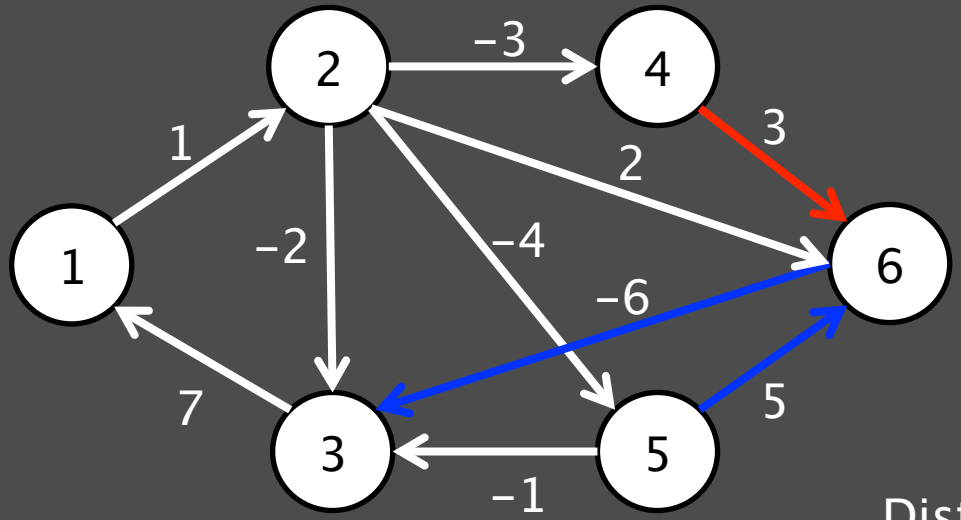


Iteration 3

Dist

1	2	3	4	5	6
0	1	−3	−2	−3	2

Bellman-Ford – Diagram

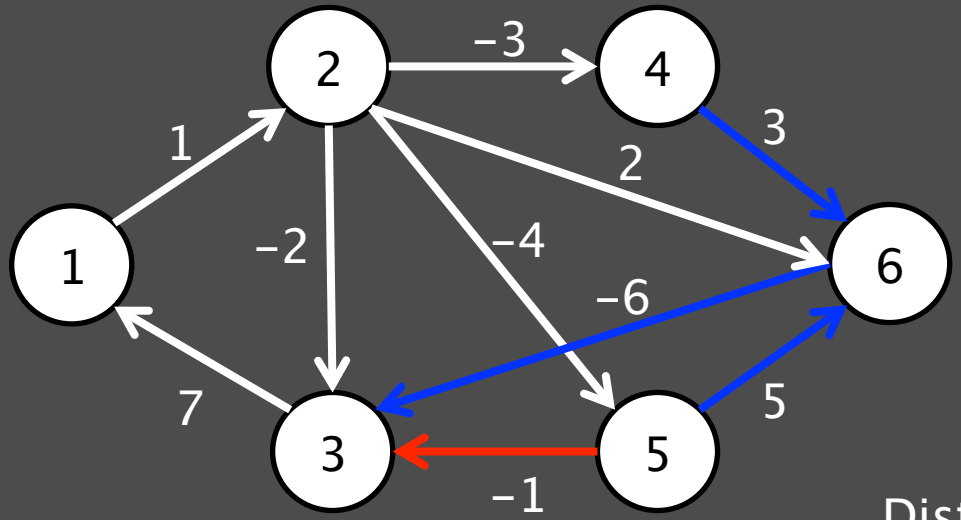


Iteration 3

Dist

1	2	3	4	5	6
0	1	−3	−2	−3	1

Bellman-Ford – Diagram

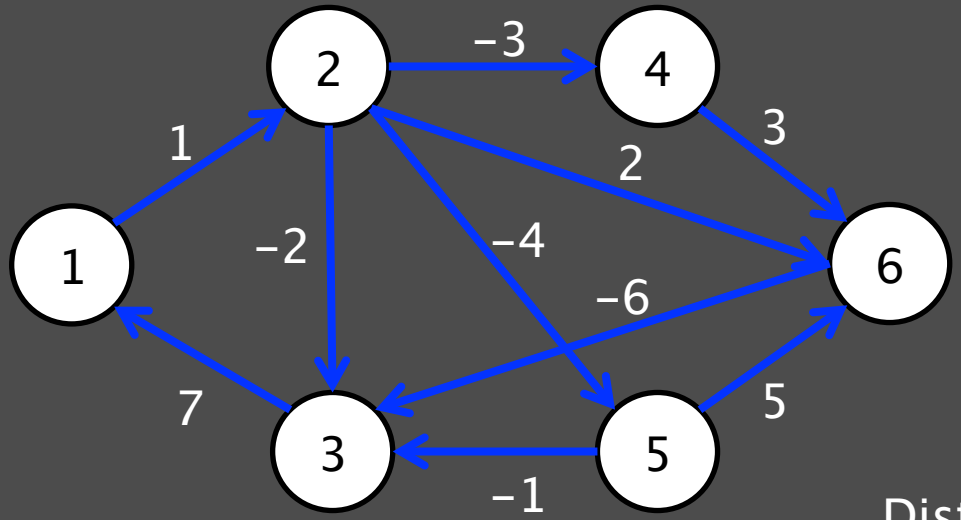


Iteration 3

Dist

1	2	3	4	5	6
0	1	−4	−2	−3	1

Bellman-Ford – Diagram

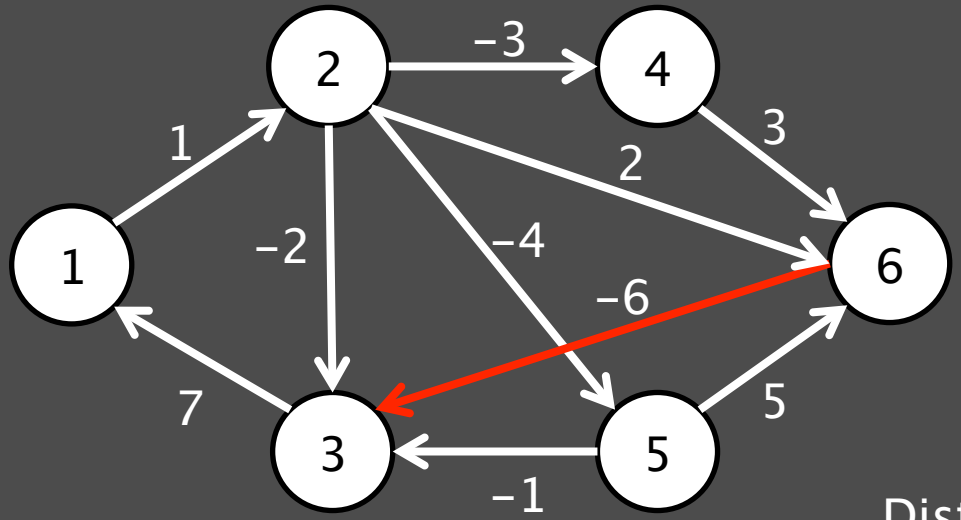


Iteration 3

Dist

1	2	3	4	5	6
0	1	−4	−2	−3	1

Bellman-Ford – Diagram

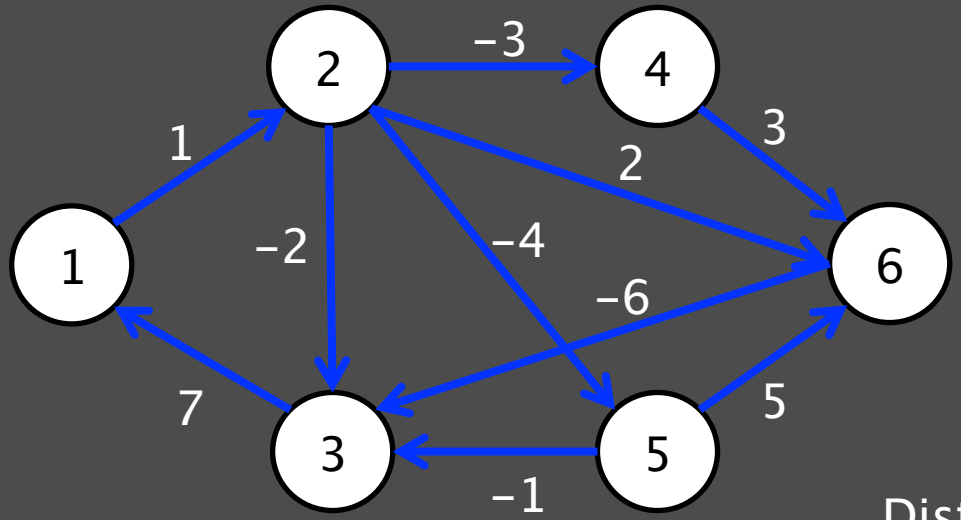


Iteration 4

Dist

1	2	3	4	5	6
0	1	−5	−2	−3	1

Bellman-Ford – Diagram

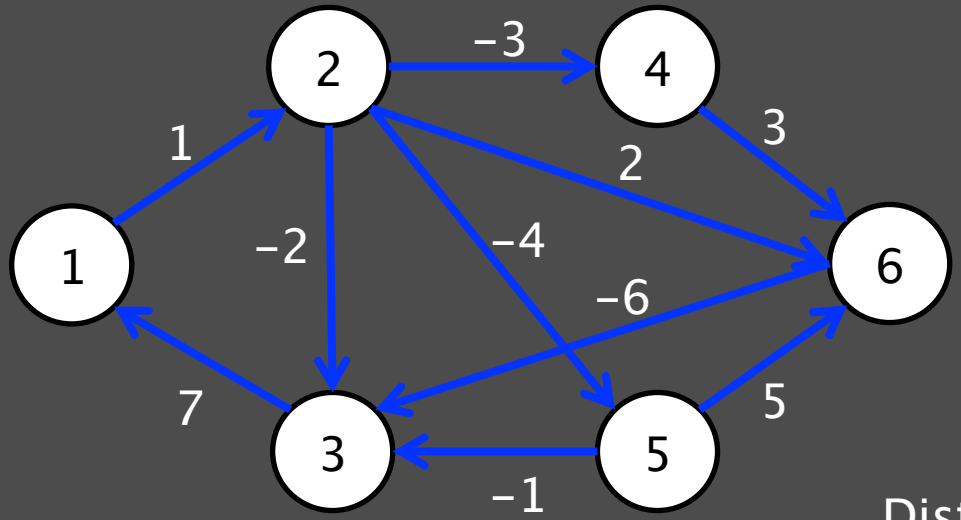


Iteration 4

Dist

1	2	3	4	5	6
0	1	−5	−2	−3	1

Bellman-Ford – Diagram



Iteration 5

We have made *V-1* iterations, so we are done.

Dist

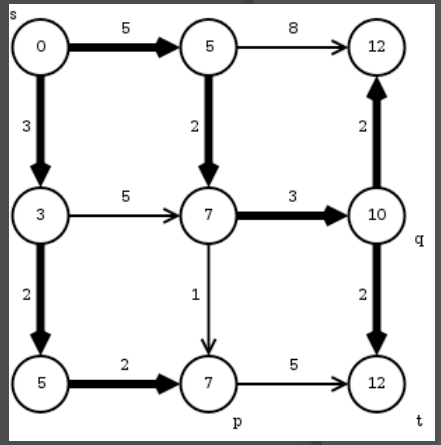
1	2	3	4	5	6
0	1	−5	−2	−3	1

Bellman-Ford Algorithm – Proof

- Proof by induction, assuming no negative cycles
- Before any iterations, there is at least 1 correct distance value, the distance to the source (which is 0)
- Suppose N iterations have occurred and $N+1$ nodes have the correct shortest distances
- If $N = V-1$, we are done since all V nodes have the shortest distance

Bellman-Ford – Proof (continued)

- Otherwise, divide the nodes into 2 sets, one containing $N+1$ nodes that have correct answers and the other set contains the rest
- Shortest paths tree is the set of edges used to travel with minimal distance from source to any other node
- There must exist some node w in the second set whose parent in the shortest paths tree is in the first set
- When the edge (u,w) is relaxed, $dist(w)$ will be correct
- Therefore, in the $N+1$–th iteration, there are at least $N+2$ nodes with correct distances



Bellman-Ford – Negative Cycles

- Proof of detection of negative cycles
- Let $v_0, v_1, \ldots, v_{n-1}$ be the nodes in a negative cycle
- Their entries in the distance array are $d_0, d_1, \ldots, d_{n-1}$ and weight of the edge between v_i and v_{i+1} is w_i
  - Weight of edge between v_{n-1} and v_0 is w_{n-1}
- Because the cycle is negative, $w_0+w_1+\ldots+w_{n-1} < 0$
- We will show by contradiction that relaxation is always possible

Bellman-Ford – Negative Cycles (Continued)

- Assume that there exists no i such that $d_i + w_i < d_{i+1}$
- That means that for each i, $d_i + w_i \geq d_{i+1}$
- If we add all n equations, together, we get:

$(d_0+d_1+\ldots+d_{n-1}) + (w_0+w_1+\ldots+w_{n-1}) \geq (d_0+d_1+\ldots+d_{n-1})$

- Cancel out $(d_0+d_1+\ldots+d_{n-1})$ from both sides
- We are left with $(w_0+w_1+\ldots+w_{n-1}) \geq 0$
- This contradicts with the fact that in a negative cycle, $w_0+w_1+\ldots+w_{n-1} < 0$
- Therefore, it is always possible to relax an edge

Bellman-Ford – Negative Cycles

- We have proved that it is always possible to relax an edge if there is a negative cycle
- The maximum number of edges of a path in a graph without a negative cycle is $V-1$
  - All other nodes are intermediate nodes
- Therefore, if we can still relax an edge after $V-1$ iterations, a negative cycle exists

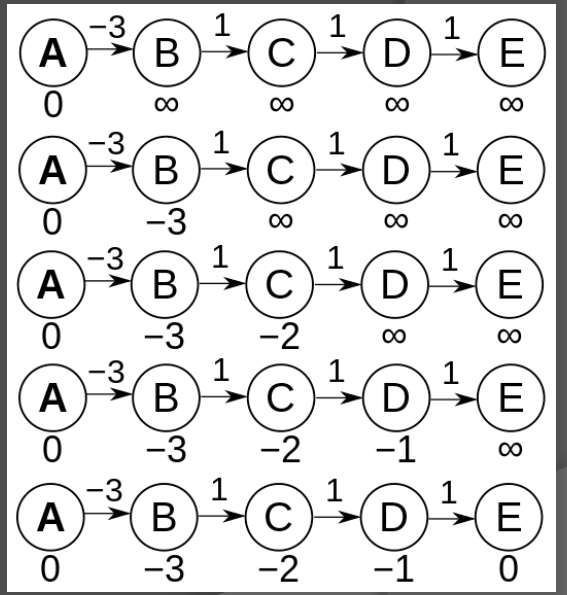
Bellman-Ford Algorithm

```
//Initialize all dists to infinity
dist(source) = 0;
//compute the shortest paths
for i from 1 to V-1
    for each edge e
        if (dist(e.n1) + e.cost < dist(e.n2))
            dist(e.n2) = dist(e.n1) + e.cost;
              //can use a parent array to find paths

//check for negative cycles
for each edge e
       if (dist(e.n1) + e.cost < dist(e.n2))
              //negative cycle found
```

Bellman-Ford Algorithm - Analysis

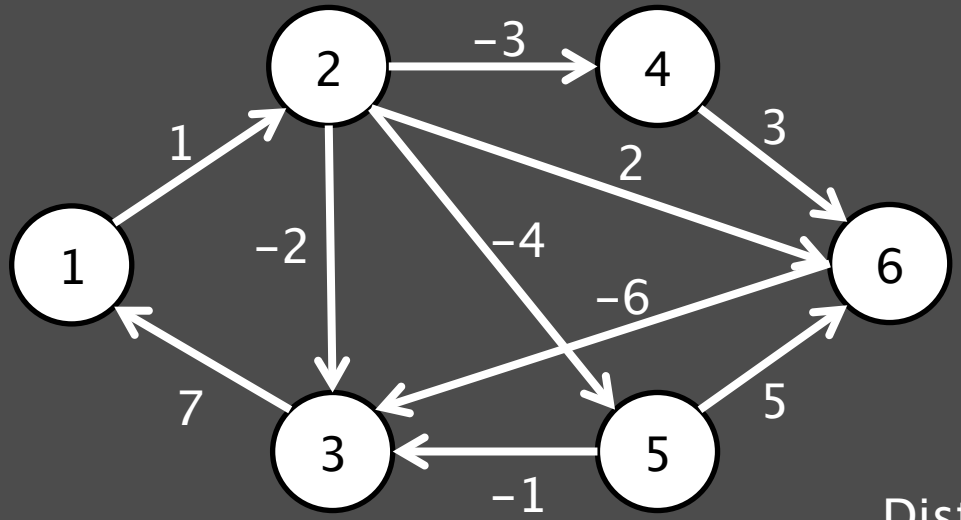
- Time Complexity: $O(VE)$
- Slower than Dijkstra's algorithm, but more versatile (due to capability to handle negative edges)
- Can make average case run faster if we terminate if no changes to distance array are made in an iteration



A "Faster" Shortest Path Algorithm

- The Shortest Path Faster Algorithm (SPFA) can also solve SSSP with negative edges
- SPFA is an improvement of the Bellman-Ford Algorithm
- Instead of relaxing all edges $V-1$ times, we relax outgoing edges from nodes (like Dijkstra)
- Similar to Dijkstra except:
  - If we update distance to a node, push it into a regular queue
  - We can visit nodes more than once
  - We choose the next node to visit based on the first node in the queue

SPFA – Diagram



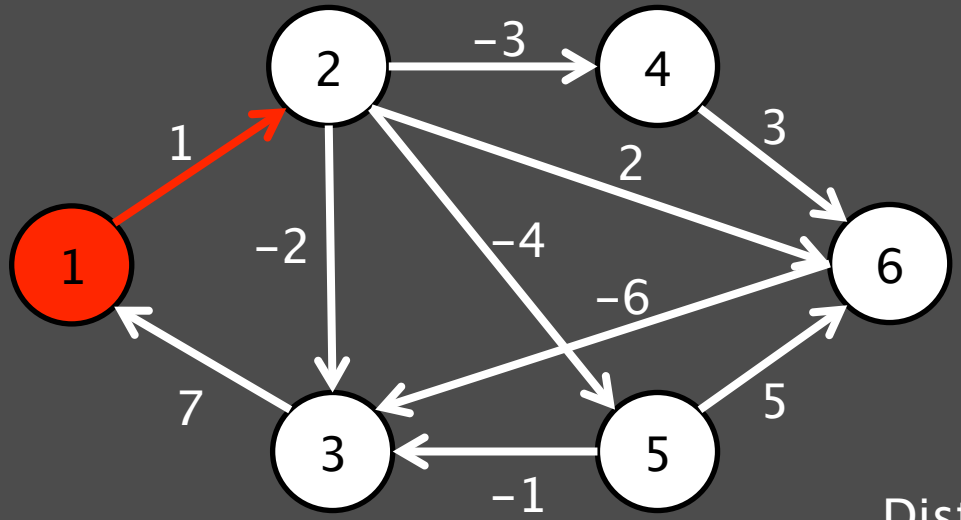
Queue:
{1}

Dist

1	2	3	4	5	6
0	∞	∞	∞	∞	∞

SPFA – Diagram

A red node signifies the current node, red edges signify successful relaxations

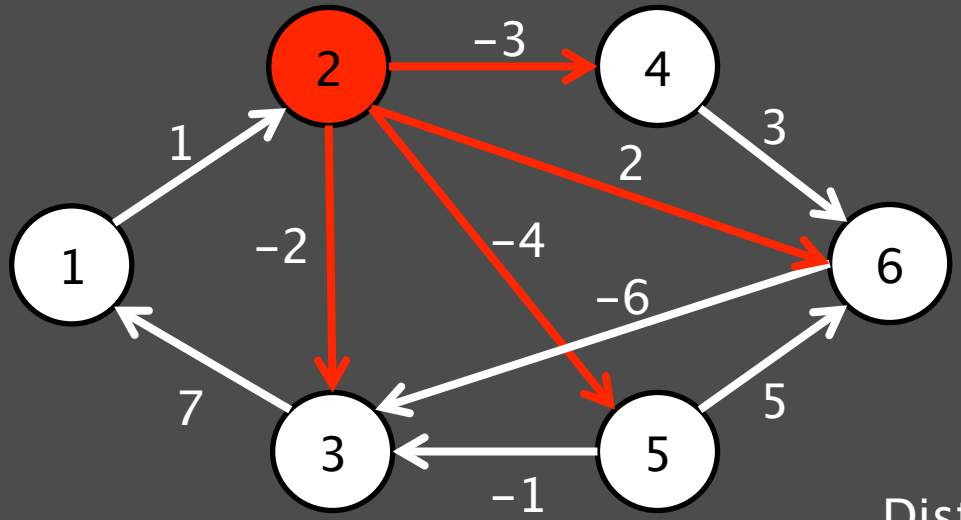


Queue:
{2}

Dist

1	2	3	4	5	6
0	1	∞	∞	∞	∞

SPFA – Diagram

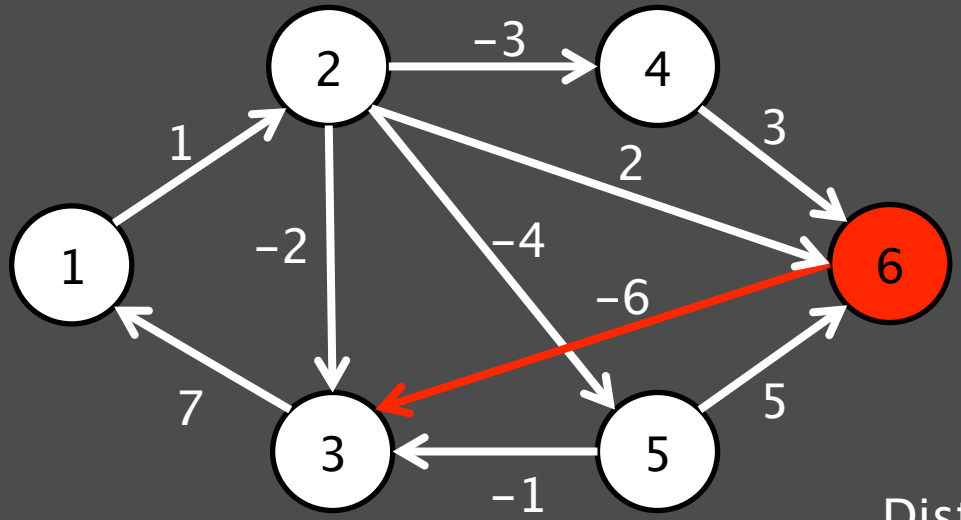


Queue:
`{6,5,3,4}`

Dist

1	2	3	4	5	6
0	1	−1	−2	−3	3

SPFA – Diagram

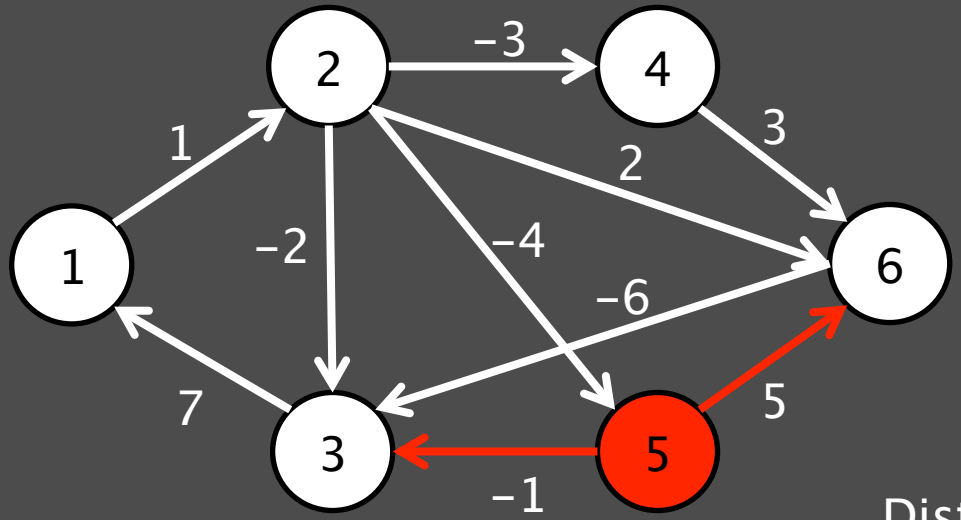


Queue:
`{5,3,4}`

Dist

1	2	3	4	5	6
0	1	−3	−2	−3	3

SPFA – Diagram

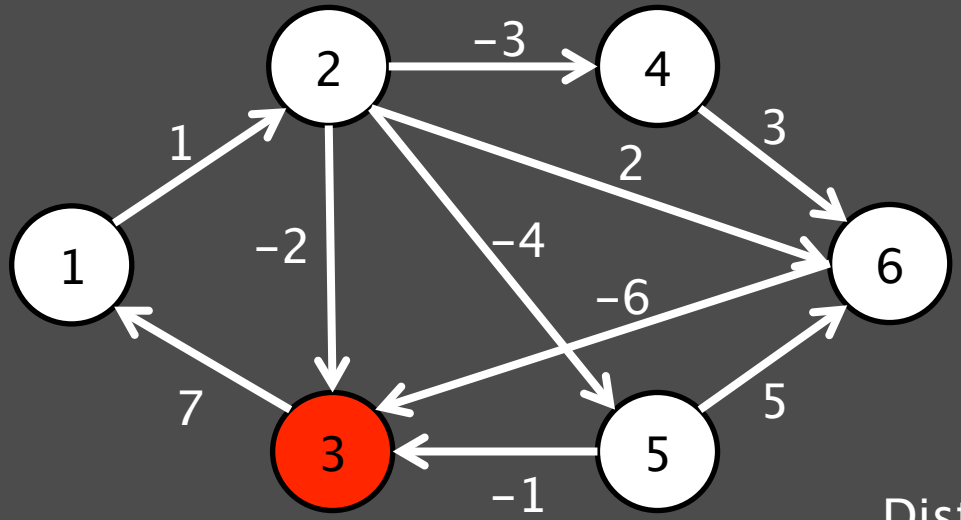


Queue:
`{3,4,6}`

Dist

1	2	3	4	5	6
0	1	−4	−2	−3	2

SPFA – Diagram

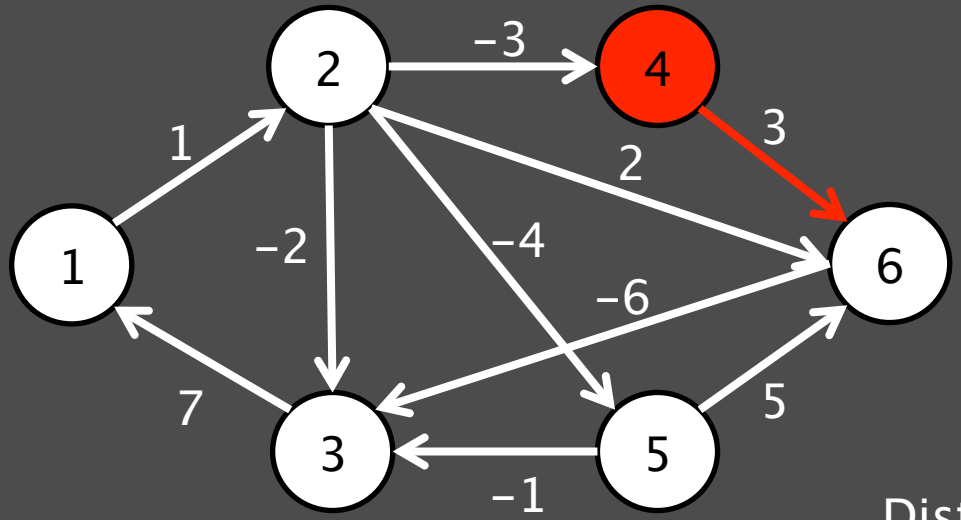


Queue:
`{4,6}`

Dist

1	2	3	4	5	6
0	1	−4	−2	−3	2

SPFA – Diagram

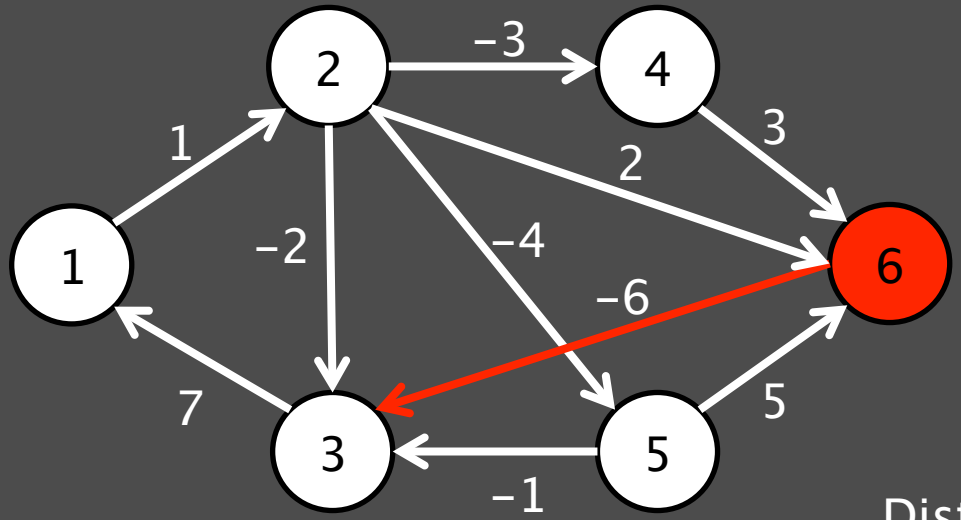


Queue:
{6}

Dist

1	2	3	4	5	6
0	1	−4	−2	−3	1

SPFA – Diagram

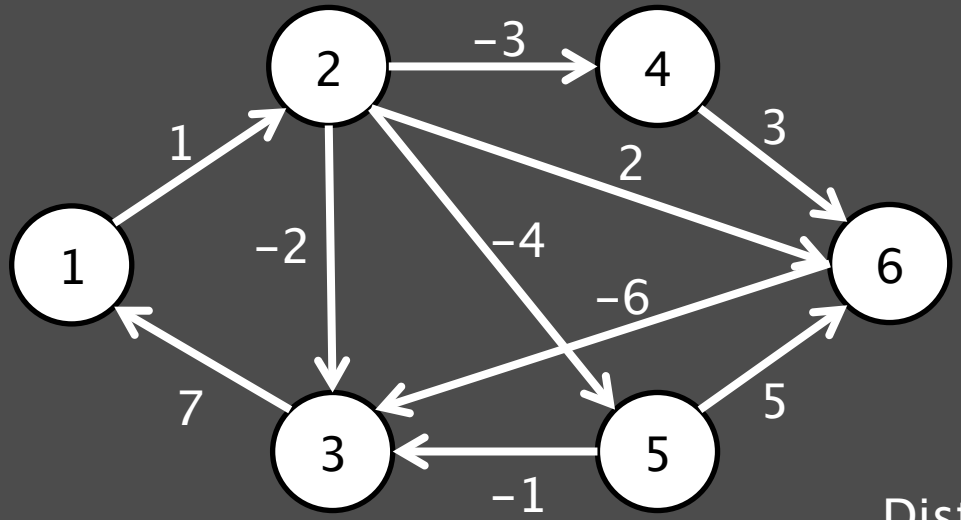


Queue:
{3}

Dist

1	2	3	4	5	6
0	1	−5	−2	−3	1

SPFA – Diagram



Queue:
{ }

The queue is empty, so we are done.

Dist

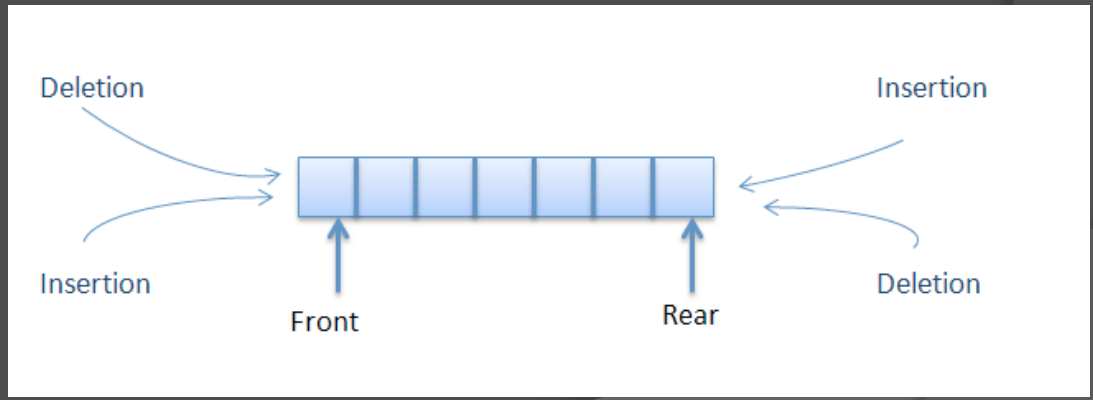
1	2	3	4	5	6
0	1	−5	−2	−3	1

SPFA – Pseudocode

```
//Initialize all dists to infinity
dist(source) = 0;
queue Q;
Q.push(source);
while Q is not empty{
        Node nd = Q.front(); Q.pop();
        for each neighbor u{
                if dist(nd) + cost(nd,u) < dist(u){
                        dist(u) = dist(nd) + cost(nd,u)
                        if (u not in Q){ //use bool array
                                Q.push (u);
                        }
                }
        }
}
```

SPFA – Optimizations

- There are two optimizations that deal with inserting into the queue
- Need a double-ended queue (deque) for these optimizations
- Small Label First
- Large Label Last



SPFA – Small Label First

- Suppose we insert node nd
- If $dist(nd) < dist(front(deque))$, push nd to the front, otherwise, push nd to the back

SPFA – Small Label First

```
//instead of queue Q, we have deque D
//replace "Q.push(u);" with this
if D is not empty and dist(u) < dist(D.front())
      D.push_front(u);
else
      D.push_back(u);
```

SPFA – Large Label Last

- Keep a variable that stores the average $dist(nd)$ for all nd in the deque
  - Update this variable when we push or pop elements
- As long as the first element of the deque is greater than the average, pop the first element and push it to the back

SPFA – Large Label Last

```
//add this after adding to D
while dist(D.front()) > avg{
      D.push_back(D.front());
      D.pop_front();
}

//before adding u to D
avg = (avg*size(D) + dist(u))/(size(D)+1);

//after removing u from D
avg = (avg*(size(D)+1) - dist(u))/size(D);
```

SPFA – Proof of Correctness

- Proof that the algorithm never computes wrong answers
- Lemma: If $dist(w) > dist(u) + cost(u,w)$, then u is in the queue
  - Proof by induction
  - Before any looping, the only node that can be relaxed is the source
  - During looping, a node u is popped and used to relax its neighbors
  - Immediately after that iteration, u cannot cause any more relaxations and is not in the queue

SPFA – Proof (continued)

- Lemma: If $dist(w) > dist(u) + cost(u,w)$, then u is in the queue
  - If u is relaxed, then other nodes might be able to cause relaxation
  - If there exists nodes x and w such that $dist(x) > dist(w) + cost(w,x)$…
    - Before the current loop iteration, then w is in the queue
    - During the current loop iteration but not before, then either $dist(x)$ increased (not possible), or $dist(w)$ decreased, meaning that it was relaxed
      - If w was relaxed, it now is or already was in the queue

SPFA – Termination Proof

- Corollary: The algorithm terminates iff no relaxations are possible
  - Nodes are always removed from queue, but only pushed upon successful relaxations
  - If there are no more relaxations, the queue is empty and the algorithm terminates

SPFA – Negative Cycles

- Therefore, if a negative cycle exists, the algorithm will not terminate on its own
  - Nodes will always be able to be relaxed
- From Bellman-Ford, if a graph does not contain a negative cycle, a node will be relaxed at most $V-1$ times
- Therefore, if any node is relaxed (pushed into the queue) more than $V-1$ times, a negative cycle exists

SPFA – Negative Cycle Pseudocode

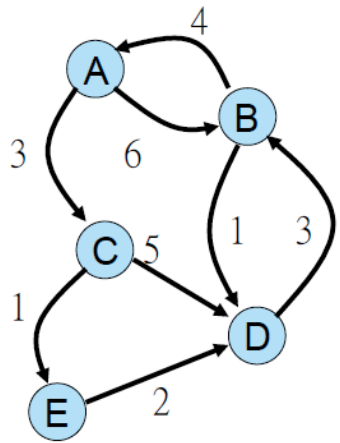
```
if dist(nd) + cost(nd,u) < dist(u){
      cnt(u)++;
      if (cnt(u) >= V){
            //negative cycle found
            terminate;
      }
      //relaxation
}
```

SPFA - Analysis

- Average Time Complexity: $O(E)$
- Worst Case Time Complexity: $O(VE)$
- Often slower than Dijkstra
- Faster than Bellman-Ford

All Pairs Shortest Path

- A problem that requires us to find the shortest distance from any one node to any other node



	A	B	C	D	E
A	0	6	3	6	4
B	4	0	7	10	8
C	12	6	0	3	1
D	7	3	10	0	11
E	9	5	12	2	0

↑

Resulting Adjacency Matrix Containing Distances

Using Existing Algorithms

- We can use Dijkstra's algorithm or SPFA starting from every node in order to get distances from any one node to any other node
- Runtime is same as the original algorithms, but multiplied by $O(N)$
- Disadvantages
  - Requires relatively lengthy code
  - Algorithm must be placed in a method
  - Arrays, variables need to be reset (if they are global)

Floyd-Warshall Algorithm

- A simple graph algorithm that computes the minimum cost from any two nodes
- Code is *very* short
- Works on graphs with negative edges
- Can detect negative cycles

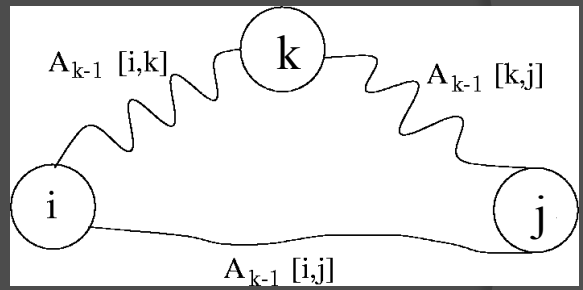
Floyd-Warshall Algorithm

```
for (each node k)
    for (each node i)
        for (each node j)
            dist(i,j) = min(dist(i,j),dist(i,k)+dist(k,j));
```

Note:
First loop must have `k` as its variable

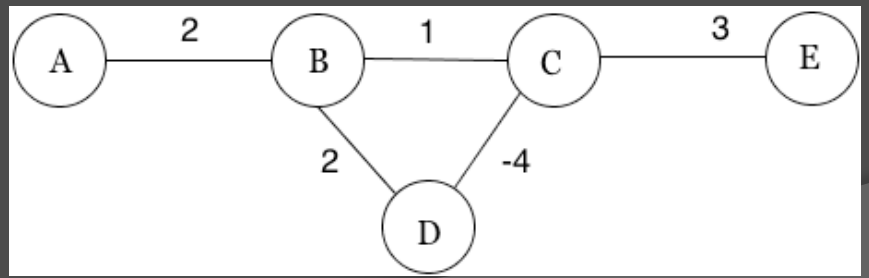
Floyd-Warshall Algorithm – Proof

- Let *shortestPath(i,j,k)* be the length of the shortest path from *i* to *j* using only nodes *1* to *k* as intermediate nodes
- *shortestPath(i,j,0)* is simply equal to cost of the edge between *i* and *j*
- Otherwise, *shortestPath(i,j,k+1)* is either
  - *shortestPath(i,j,k)* or
  - *shortestPath(i,k+1,k) + shortestPath(k+1,j,k)*
  - Note that for both possibilities, the third parameter is equal to *k*
- In the end, we will have computed all values of *shortestPath(i,j,V)*
  - The shortest path from *i* to *j* using any node as intermediate nodes



Floyd-Warshall Algorithm – Negative Cycles

- As stated before, the Floyd–Warshall algorithm can detect negative cycles
- For each node, check if the distance from that node to itself is negative
  - If any of these distances are negative, a negative cycle exists



Floyd-Warshall Algorithm - Analysis

- Runtime is $O(N^3)$
- Typically used on adjacency matrices
  - Memory does not matter since runtime is the bottleneck of the algorithm
  - Remember to initialize the adjacency matrix to infinity before getting graph and running algorithm
- Slower than running SSSP algorithms from every node, but code is significantly shorter

THANK YOU!