Lecture 11
Shortest Paths

Jianchen Shan
Department of Computer Science
Hofstra University

Lecture Goals

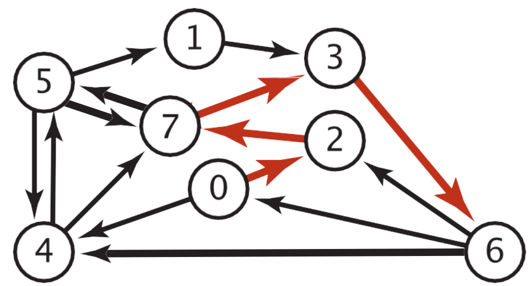
- In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- We introduce and analyze Dijkstra's algorithm for shortest-paths problems with nonnegative weights.
- Next, we consider an even faster algorithm for DAGs, which works even if the weights are negative.
- We conclude with the Bellman–Ford–Moore algorithm for edge-weighted digraphs with no negative cycles.
- We also consider applications ranging from content-aware fill to arbitrage.

Shortest Paths in an Edge-weighted Digraph

Given an edge-weighted digraph, find the shortest path from s to t.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



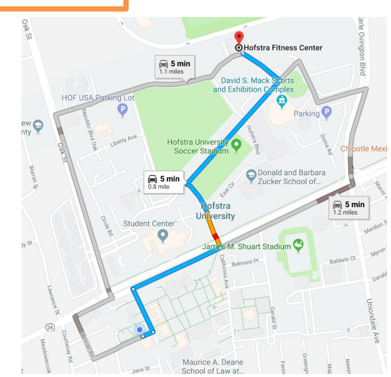
shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Can we use BFS?

Variants

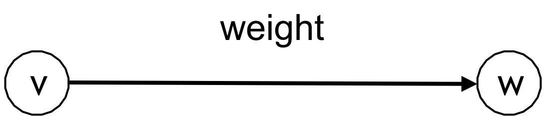
- **Which vertices?**
  - Single source: from one vertex s to every other vertex.
  - Source-sink: from one vertex s to another t.
  - All pairs: between all pairs of vertices.
- **Nonnegative weights?**
- **Cycles?**
  - Negative cycles.



Simplifying assumption: Each vertex is reachable from s.

Weighted Directed Edge API

```
public class DirectedEdge
---------------------------------------------------------------
    DirectedEdge(int v, int w, double weight)  //weighted edge v->w
  int from()                        // vertex v
  int to()                          // vertex w
  double weight()                   // the weight
  String toString()                 // string representation
```



Idiom for processing an edge e: int v = e.from(), w = e.to();

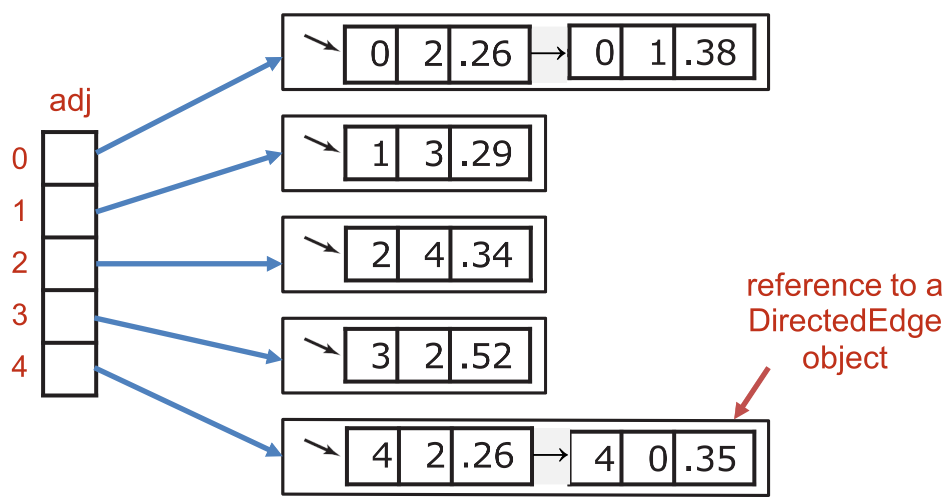
Weighted Edge: Java Implementation

```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from()
    { return v; }

    public int to()
    { return w; }

    public int weight()
    { return weight; }
}
```

Edge-Weighted Graph API

```
public class EdgeWeightedDigraph
```

EdgeWeightedDigraph(int V)	*// edge-weighted digraph with V vertices*
void addEdge(DirectedEdge e)	*// add weighted directed edge e*
Iterable <DirectedEdge> adj(int v)	*// edges pointing from v*
Iterable <DirectedEdge> edges()	*// all edges in this graph*
int V()	*// number of vertices*
int E()	*// number of edges*
String toString()	*// string representation*



Edge-Weighted Digraph: Adjacency-Lists Implementation

```
public class EdgeWeightedDigraph
{
    private final int V;
    private final List<DirectedEdge>[] adj;

    public EdgeWeightedDigraph (int V)
    {
        this.V = V;
        adj = (List<DirectedEdge>[]) new ArrayList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new ArrayList<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v)
    {   return adj[v];      }
}
```

add edge e = v->w to
only v's adjacency lists

Single-source Shortest Paths API

Goal. Find the shortest path from s to every other vertex.

```
public class SP
------------------------------------------------------------------------
               SP(EdgeWeightedGraph G, int s)  // shortest paths from s in graph G
        double distTo(int v)                   // length of shortest path from s to v
Iterable <DirectedEdge> pathTo(int v)          // shortest path from s to v
```

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
        StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
        for (DirectedEdge e : sp.pathTo(v)) {
            StdOut.print(e + " ");
            StdOut.println();
        }
}
```

```
0 to 0 (0.00):
0 to 1 (1.05):  0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26):  0->2 0.26
0 to 3 (0.99):  0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38):  0->4 0.38
0 to 5 (0.73):  0->4 0.38  4->5 0.35
0 to 6 (1.51):  0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60):  0->2 0.26  2->7 0.34
```

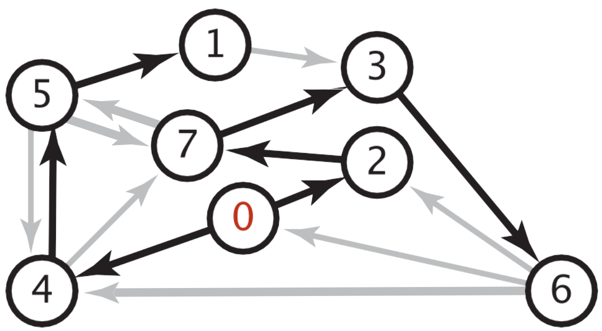
Data Structures for Single-source Shortest Paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists.

Consequence. Can represent the SPT with two vertex-indexed arrays:

- distTo[v] is length of shortest path from s to v.
- edgeTo[v] is last edge on shortest path from s to v.



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

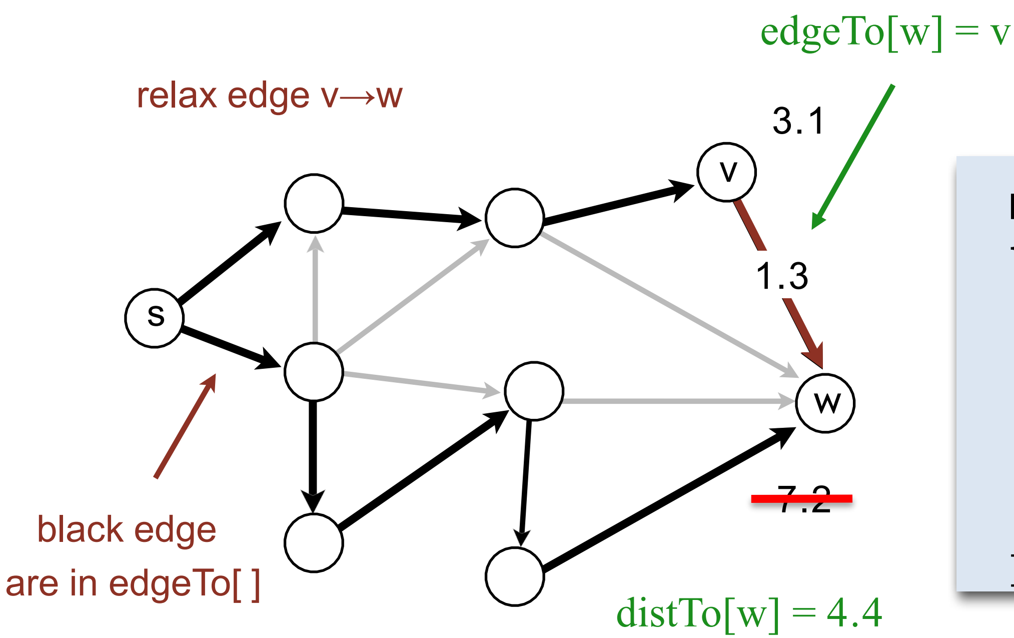
```
public double distTo(int v)
{ return distTo[v]; }
```

```
public Iterable<DirectedEdge> pathTo(int v)
{
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for  (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

Edge Relaxation

Relax edge e = v→w. (basic of building SPT)

- distTo[v] is length of shortest known path from s to v.
- distTo[w] is length of shortest known path from s to w.
- edgeTo[w] is last edge on shortest known path from s to w.
- If e = v→w gives shorter path to w through v, update distTo[w] and edgeTo[w].




```
private void relax(DirectedEdge e)
{
    int  v  = e.from(), w = e.to();
    if  (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Shortest-paths Optimality conditions

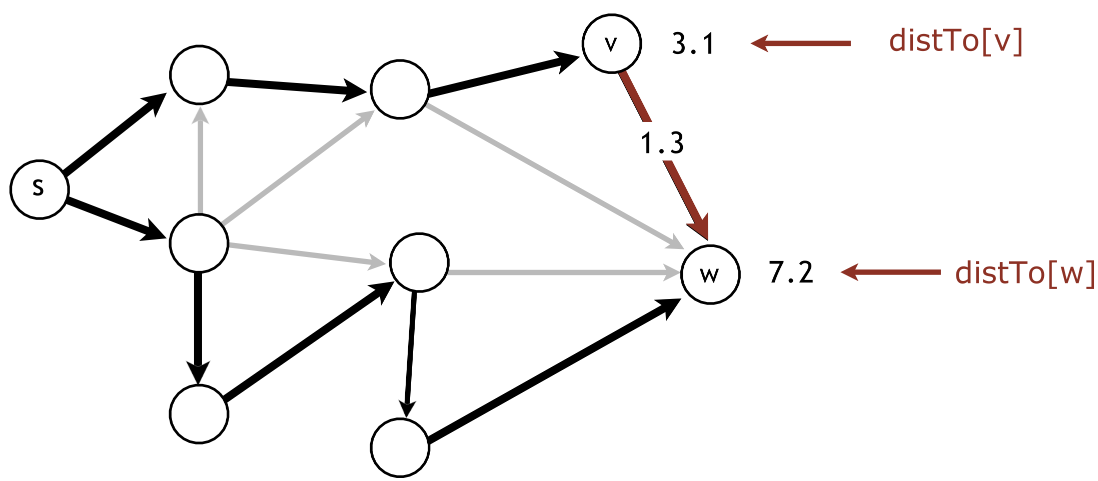
Proposition. Let G be an edge-weighted digraph.

Then distTo[] are the shortest path distances from s *iff*:

- distTo[s] = 0.
- For each vertex v, distTo[v] is the length of some path from s to v.
- For each edge e = v→w, distTo[w] ≤ distTo[v] + e.weight( ).

Pf. ⇒ [necessary]

- Suppose that distTo[w] > distTo[v] + e.weight() for some edge e = v→w.
- Then, e gives a path from s to w (through v) of length less than distTo[w].



Shortest-paths Optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then distTo[] are the shortest path distances from s *iff*:

- distTo[s] = 0.
- For each vertex v, distTo[v] is the length of some path from s to v.
- For each edge e = v→w, distTo[w] ≤ distTo[v] + e.weight( ).

Pf. ⟸ [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from s to w.
- Then,

$$\text{distTo}[v_1] \le \text{distTo}[v_0] + e_1.\text{weight}(\,)$$

$$\text{distTo}[v_2] \le \text{distTo}[v_1] + e_2.\text{weight}(\,)$$

$$\ldots\ldots$$

$$\text{distTo}[v_k] \le \text{distTo}[v_{k-1}] + e_k.\text{weight}(\,)$$

e_i = i^{th} edge on shortest path from s to w

- Add inequalities; simplify; and substitute distTo[v0] = distTo[s] = 0:

$$\text{distTo}[w] = \text{distTo}[v_k] \le \underline{e_1.\text{weight}(\,) + e_2.\text{weight}(\,) + \ldots + e_k.\text{weight}(\,)}$$

- Thus, distTo[w] is the weight of shortest path to w.

weight of shortest path from s to w

Generic Shortest-paths Algorithm

Generic algorithm (to compute SPT from s)

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat until done:

- Relax any edge.

Proposition. Generic algorithm computes SPT (if it exists) from s.

Pf.

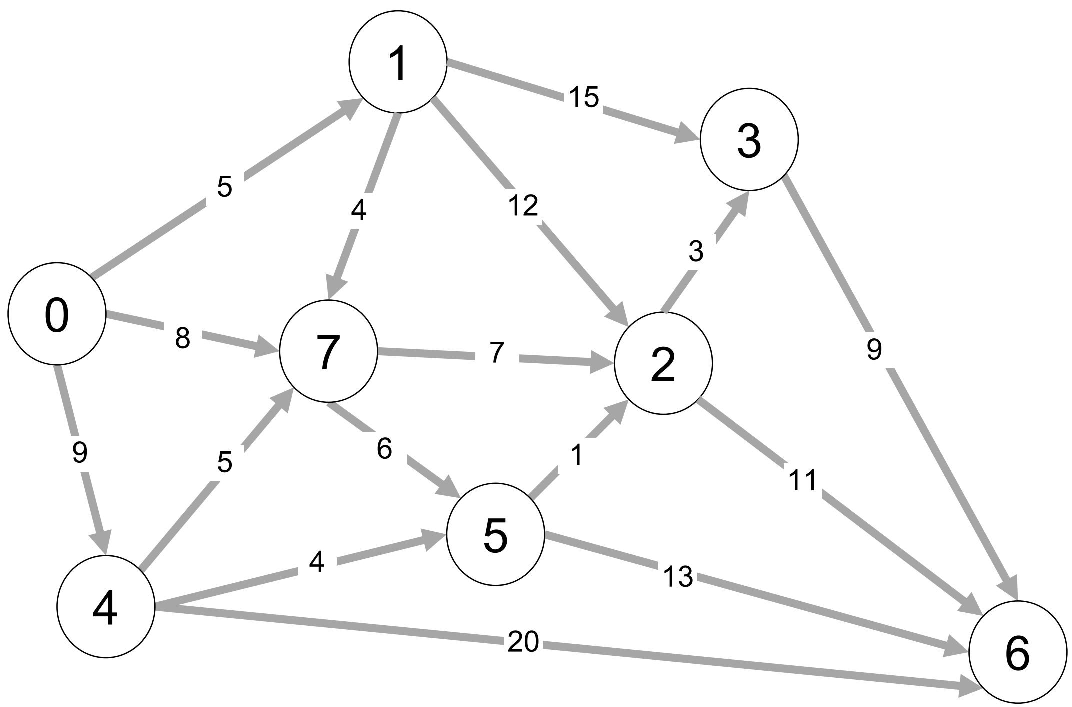
- Throughout algorithm, distTo[v] is the length of a simple path from s to v (and edgeTo[v] is last edge on path).
- Each successful relaxation decreases distTo[v] for some v.
- The entry distTo[v] can decrease at most a finite number of times.

Efficient implementations. How to choose which edge to relax?

- Ex 1. Dijkstra's algorithm. (nonnegative weights, directed cycles).
- Ex 2. Topological sort algorithm. (no directed cycles).
- Ex 3. Bellman–Ford algorithm. (no neigtive cycles).

Dijkstra's Algorithm

- Consider vertices in increasing order of distance from s
  - (non-tree vertex with the lowest distTo[ ] value).
- Add vertex to tree and relax all edges pointing from that vertex.



choose source vertex 0
relax all edges adjacent from 0
choose vertex 1
relax all edges adjacent from 1

choose vertex 7
relax all edges adjacent from 7
choose vertex 4
relax all edges adjacent from 4

choose vertex 5
relax all edges adjacent from 5
choose vertex 2
relax all edges adjacent from 2
choose vertex 3
relax all edges adjacent from 3
choose vertex 6
relax all edges adjacent from 6

v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	~~15~~	14
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	~~14~~	13	
6	~~∞~~	~~29~~	~~26~~	25
7	~~∞~~	8		

v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	~~7~~	4	
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

Dijkstra’s Algorithm: Correctness Proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge e = v→w is relaxed exactly once (when v is relaxed),
  - leaving distTo[w] ≤ distTo[v] + e.weight().
- Inequality holds until algorithm terminates because:
  - distTo[w] cannot increase ← distTo[ ] values are monotone decreasing
  - distTo[v] will not change ← we choose lowest distTo[ ] value at each step (and edge weights are nonnegative)
- Thus, upon termination, shortest-paths optimality conditions hold.

Dijkstra's Algorithm: Java Implementation

```
public   class   DijkstraSP
{
    private   DirectedEdge []   edgeTo;
    private   double []  distTo;
    private   MinPQ<Double>  pq;

    public   DijkstraSP(EdgeWeightedDigraph G,  int  s)
    {
        edgeTo  = new DirectedEdge[G.V()];
        distTo    = new double[G.V()];
        pq  = new MinPQ<Double>(G.V());

        for (int  v  = 0; v  <   G.V();  v++)
            distTo[v] = Double.POSITIVE_INFINITY;
            distTo[s] = 0.0;

        pq.insert(s,  0.0);
        while (!pq.isEmpty())
        {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

relax vertices in order
of distance from s

Dijkstra's Algorithm: Java Implementation

```
private   void   relax(DirectedEdge  e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w]    = distTo[v] + e.weight();
        edgeTo[w]   = e;
        if   (pq.contains(w))     pq.decreaseKey  (w,  distTo[w]);
        else                      pq.insert       (w,  distTo[w]);
    }
}
```

update PQ

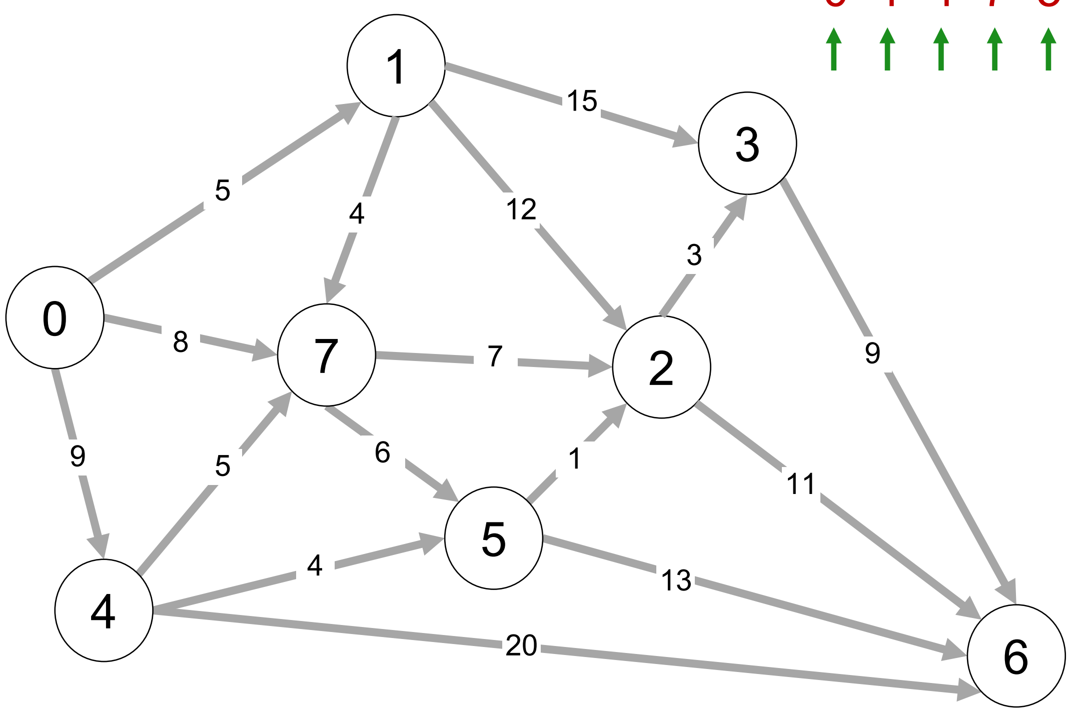
Shortest Paths in Edge-weighted DAG

Suppose that an edge-weighted digraph has no directed cycles.
Is it easier to find shortest paths than in a general digraph?

Yes!

- Consider vertices in topological order.
- Relax all edges pointing from that vertex

0 1 4 7 5 2 3 6



v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	~~15~~	14
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~29~~	~~26~~	25
7	~~∞~~	8		

v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	~~7~~	5
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	4		
6	~~-~~	~~4~~	~~5~~	2
7	~~-~~	0		

Shortest Paths in Edge-weighted DAG: Correctness Proof

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to E + V.

edge weights can be negative!

Pf.

- Each edge e = v→w is relaxed exactly once (when v is relaxed),
  - leaving distTo[w] ≤ distTo[v] + e.weight().
- Inequality holds until algorithm terminates because:
  - distTo[w] cannot increase
    distTo[ ] values are monotone decreasing
  - distTo[v] will not change
    because of topological order, no edge pointing to v will be relaxed after v is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold.

Shortest Paths in Edge-weighted DAG: Java Implementation

```
public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP (EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
            distTo[s] = 0.0;

        Topological topological = new Topological(G);
        for (int v : topological.order())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}
```

topological order

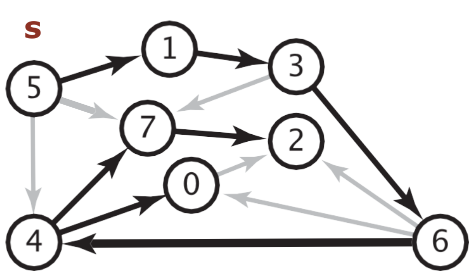
Longest Paths in Edge-weighted DAG

Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.

equivalent: reverse sense of equality in relax()

longest paths input		shortest paths input	
5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93

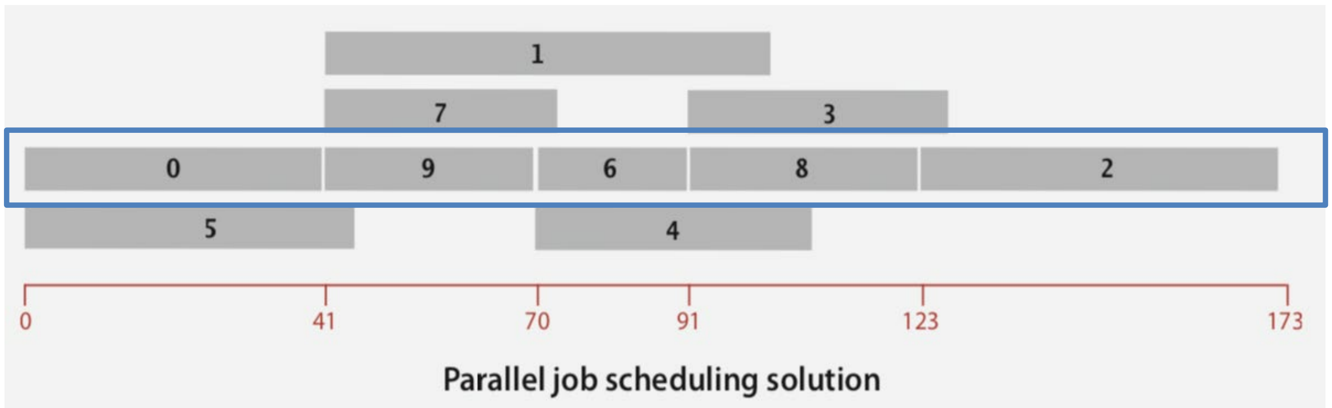


Key point. Topological sort algorithm works even with negative weights.

Longest Paths in Edge-weighted DAG: Application

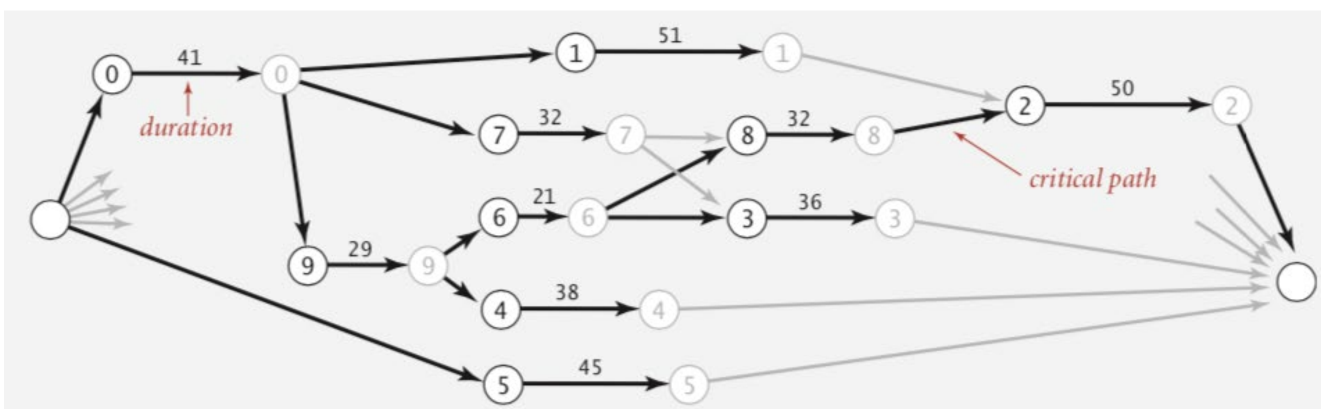
Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6



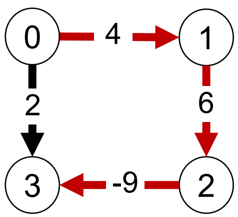
Parallel job scheduling solution

Use **longest path** **from the source to schedule each job.**

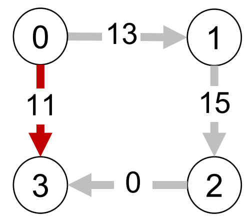


Shortest Paths with Negative weights

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

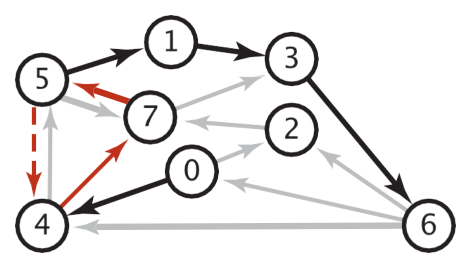


Adding 9 to each edge weight changes the shortest path from 0→1→2→3 to 0→3.

Conclusion.
Need a different algorithm.

Re-weighting. Add a constant to every edge weight doesn't work.

- A negative cycle is a directed cycle whose sum of edge weights is negative.
- A SPT exists iff no negative cycles, assuming all vertices reachable from s



negative cycle (-0.66 + 0.37 + 0.28)
5->4->7->5

shortest path from 0 to 6
0->4->7->5->4->7->5...->1->3->6

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

Bellman-Ford Algorithm

Bellman–Ford algorithm

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat V-1 times:

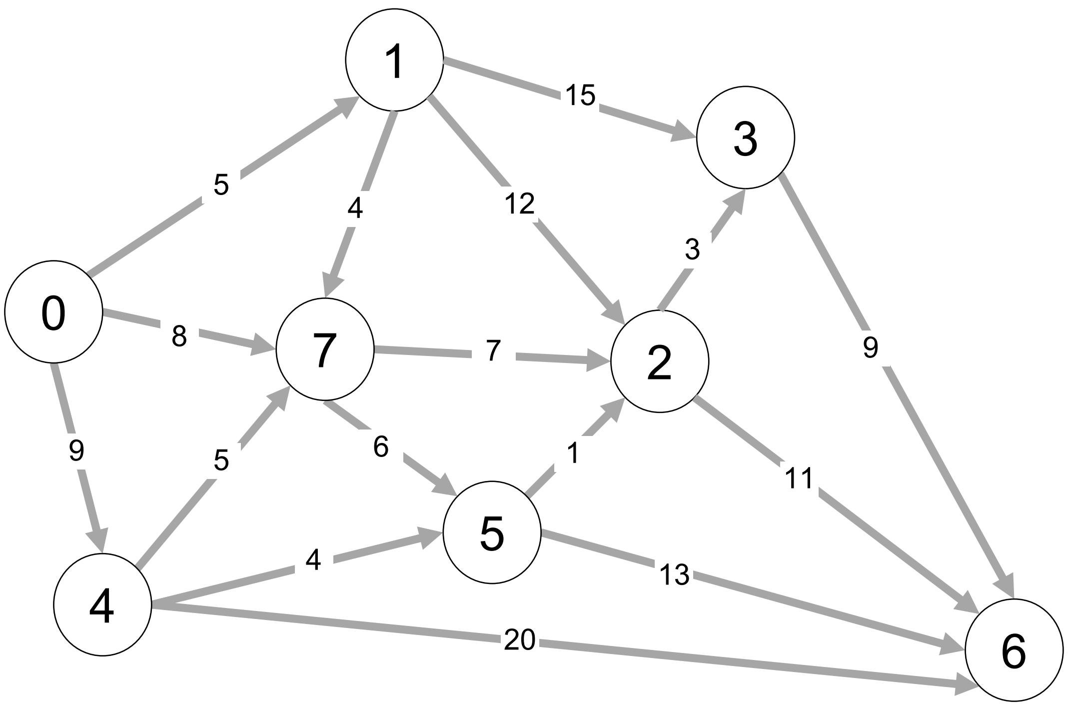
- Relax each edge.

```
for (int i = 1; i < G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

pass i (relax each edge)

Bellman-Ford Algorithm

Repeat V − 1 times: relax all E edges.



v	distTo[]			
0	~~∞~~	0		
1	~~∞~~	5		
2	~~∞~~	~~17~~	14	
3	~~∞~~	~~20~~	17	
4	~~∞~~	9		
5	~~∞~~	13		
6	~~∞~~	~~28~~	~~26~~	25
7	~~∞~~	8		

v	edgeTo[]			
0	-			
1	~~-~~	0		
2	~~-~~	~~1~~	5	
3	~~-~~	~~1~~	2	
4	~~-~~	0		
5	~~-~~	4		
6	~~-~~	~~2~~	~~5~~	2
7	~~-~~	0		

pass 1 pass 2 pass 3 (no further changes) pass 4-7 (no further changes)

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→2 7→5

Bellman–Ford Algorithm: Correctness Proof

- Proposition. Let $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k = v$ be a shortest path from s to v. Then, after pass i, $\text{distTo}[v_i] = d^*(v_i)$.

length of shortest path from s to v_i

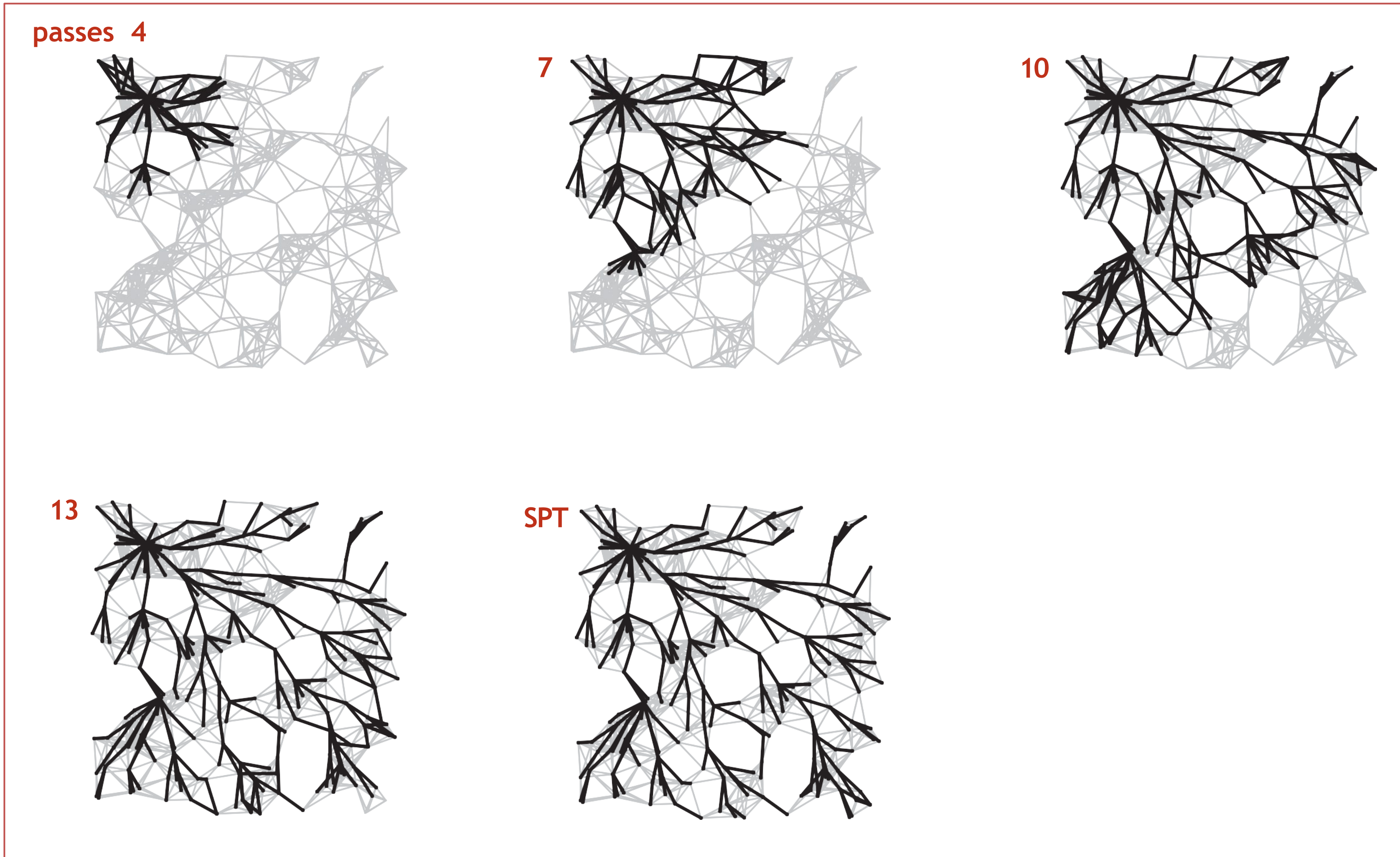
- Pf. [by induction on i]
  - Inductive hypothesis: after pass i, $\text{distTo}[v_i] = d^*(v_i)$.
  - Since $\text{distTo}[v_{i+1}]$ is the length of some path from s to v_{i+1}, we must have $\text{distTo}[v_{i+1}] \geq d^*(v_{i+1})$.
  - Immediately after relaxing edge $v_i \to v_{i+1}$ in pass i+1, we have

$$\begin{aligned} \text{distTo}[v_{i+1}] &\leq \text{distTo}[v_i] + \text{weight}(v_i, v_i+1) \\ &= d^*(v_i) + \text{weight}(v_i, v_{i+1}) \\ &= d^*(v_{i+1}). \end{aligned}$$

  - Thus, at the end of pass i+1, $\text{distTo}[v_{i+1}] = d^*(v_{i+1})$.
- Corollary. Bellman–Ford computes shortest path distances.
- Pf. There exists a shortest path from s to v with at most $V - 1$ edges.

$\Rightarrow \leq V - 1$ passes.

Bellman-Ford Algorithm Visualization

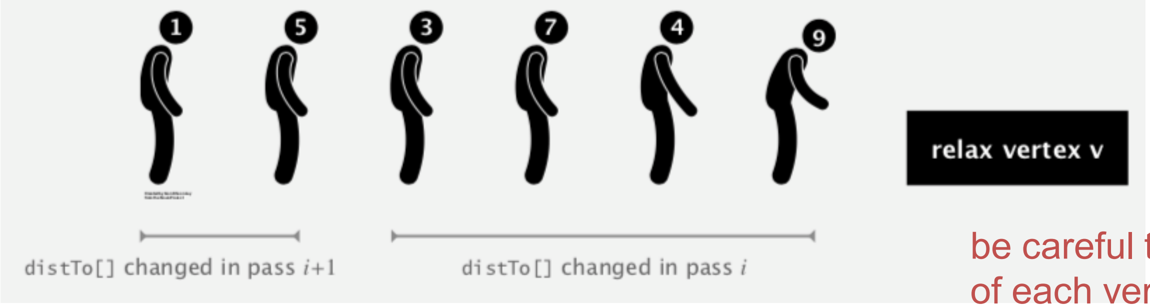


Bellman-Ford Algorithm Analysis

Observation. If distTo[v] does not change during pass i, no need to relax any edge pointing from v in pass i + 1.

Queue-based implementation of Bellman–Ford. Maintain queue of vertices whose distTo[ ] values needs updating.

In the worst case, the running time is still proportional to $E \times V$. But much faster in practice.



be careful to keep at most one copy of each vertex on queue

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating distTo[ ] and edgeTo[ ] entries of vertices in the cycle.

Proposition. If any vertex v is updated in phase V, there exists a negative cycle (and can trace back edgeTo[v] entries to find it).

Finding a negative cycle

Negative Cycle Application: Arbitrage Detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Ex. $1,000 $\Rightarrow$ 741 Euros $\Rightarrow$ 1,012.206 Canadian dollars $\Rightarrow$ $1,007.14497.

$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$

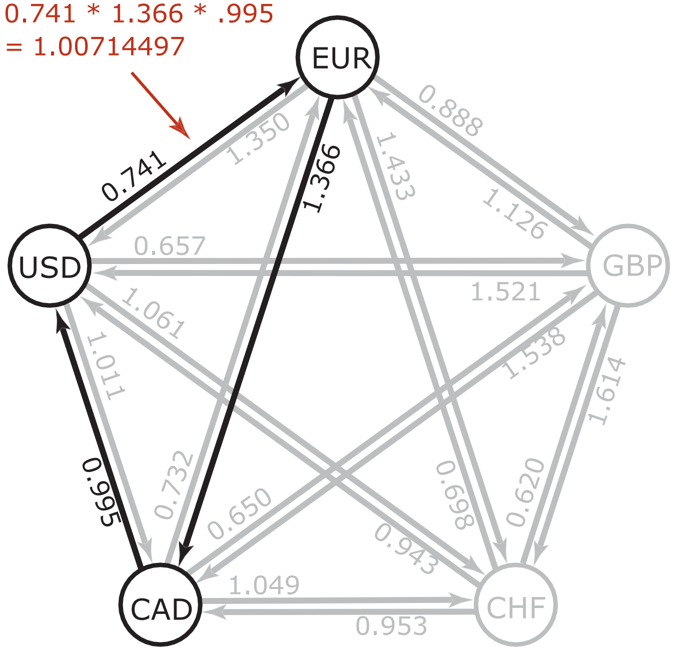
Negative Cycle Application: Arbitrage Detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.

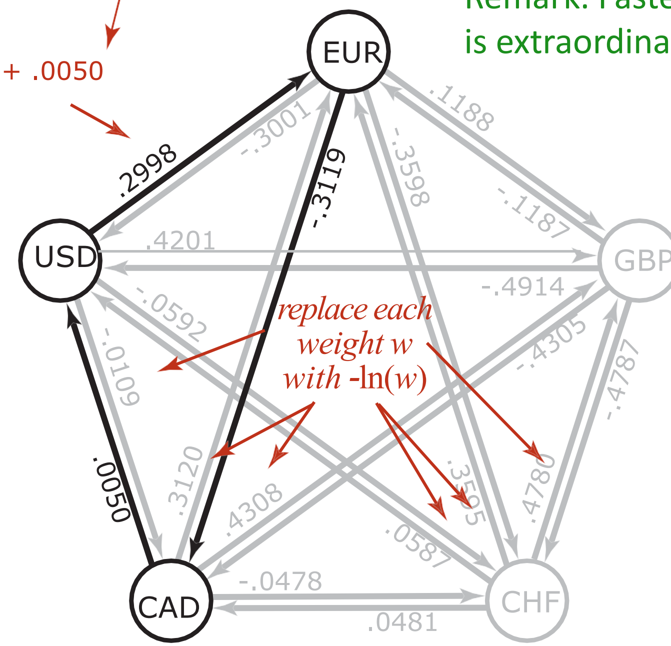
Challenge. Express as a negative cycle detection problem.

0.741 * 1.366 * .995
= 1.00714497



-ln(.741) -ln(1.366) -ln(.995)

.2998 - .3119 + .0050
= -.0071



Remark. Fastest algorithm is extraordinarily valuable!

Model as a negative cycle detection problem by taking logs.

- Let weight of edge v→w be - ln (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0.
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).

Single Source Shortest-paths Implementation: Cost Summary

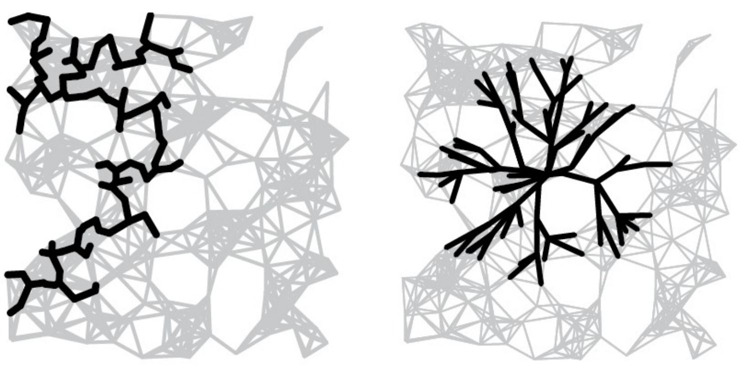
algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	E V	E V	V
Bellman-Ford (queue-based)		E + V	E V	V

- Remark 1. Directed cycles make the problem harder.
- Remark 2. Negative weights make the problem harder.
- Remark 3. Negative cycles makes the problem intractable.

Backup Slides

Dijkstra's Algorithm Analysis

- Dijkstra's algorithm seem familiar?
  - Prim's algorithm is essentially the same algorithm.
  - Both are in a family of algorithms that compute a graph's spanning tree.
- Main distinction: Rule used to choose next vertex for the tree.
  - Prim's: Closest vertex to the tree (via an undirected edge).
  - Dijkstra's: Closest vertex to the source (via a directed path).
- Note: DFS and BFS are also in this family of algorithms.



Computing spanning trees in graphs

$O(E \log V)$

operation	frequency	time per op
Insert	E	log V
delete min	E	log V
decrease key	E	log V