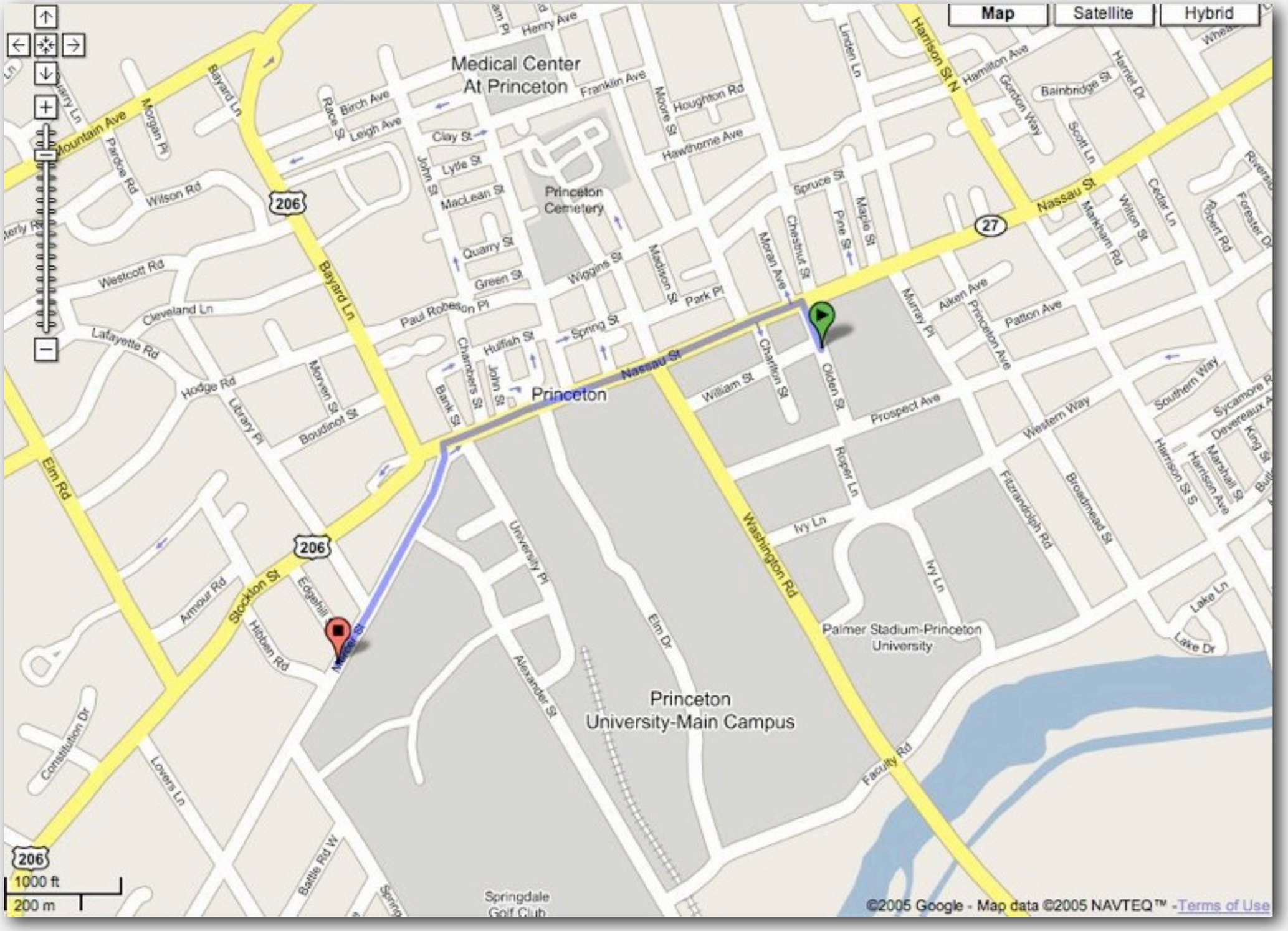
Algorithms and Data Structures, Fall 2011

Shortest paths

Rasmus Pagh

Based on slides by Kevin Wayne, Princeton

Google maps

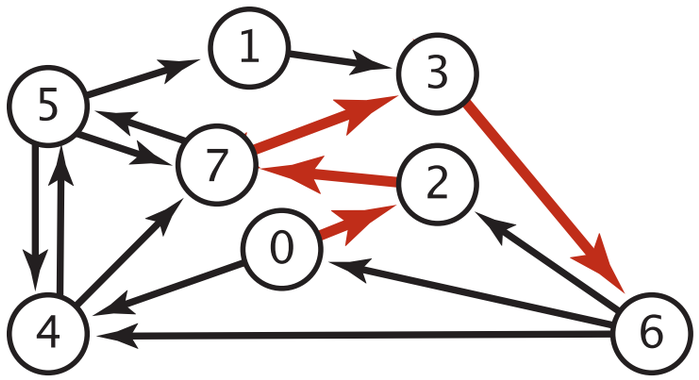

Map
Satellite
Hybrid
Medical Center
At Princeton
Princeton
Cemetery
Princeton
Princeton
University-Main Campus
Palmer Stadium-Princeton
University
Springdale
Golf Club
Mountain Ave
Bayard Ln
Nassau St
Harrison St N
Washington Rd
Stockton St
Elm Rd
Wilson Rd
Westcott Rd
Cleveland Ln
Lafayette Rd
Hodge Rd
Boudinot St
Paul Robeson Pl
Hulfish St
Spring St
Green St
Quarry St
MacLean St
Lytle St
Clay St
Birch Ave
Leigh Ave
Henry Ave
Franklin Ave
Houghton Rd
Hawthorne Ave
Moore St
Linden Ln
Hamilton Ave
Gordon Way
Bainbridge St
Harriet Dr
Scott Ln
Wiggins St
Madison St
Park Pl
Moran Ave
Chestnut St
Pine St
Spruce St
Maple St
Murray Pl
Aiken Ave
Princeton Ave
Patton Ave
Markham Rd
Witten St
Cedar Ln
Robert Rd
Forester Dr
Southern Way
Western Way
Prospect Ave
William St
Chambers St
Bank St
John St
Olden St
Charlton St
Roper Ln
Ivy Ln
Fitzrandolph Rd
Broadmead St
Harrison St S
Harrison Ave
Marshall St
King St
Lake Ln
Lake Dr
Faculty Rd
University Pl
Alexander St
Elm Dr
Edgehill St
Armour Rd
Hibben Rd
Constitution Dr
Lovers Ln
Battle Rd W
Morven St
Library Pl
Morgan Pl
Pardee Rd
Berry Ln
Merly Rd
Sherman Ave
Devereaux Ave
Sycamore Rd
Forbes Ave
Riverside
Wheat
206
27
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Shortest paths in a weighted digraph

Given an edge-weighted digraph, find the shortest (directed) 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
```

Optimization problems expressible as shortest paths

Many optimization problems can be expressed as a shortest path computation in a suitable weighted graph. (Course goal: Be able to do this!)

Examples (board):

1. Fastest route (no traffic lights), where a turn takes 5 seconds.
2. Cheapest route, considering gasoline use and road tolls.

Problem session:

Model this problem as **one** shortest path computation:

3. Find the shortest path from vertex s to a vertex v in the set S (e.g., S could be vertices with gasoline stations).

Shortest path applications

- Map routing.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest path variants

Which vertices?

- Source-sink: from one vertex to another.
- Single source: from one vertex to every other.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Arbitrary weights.

Cycles?

- No cycles.
- No "negative cycles."

Simplifying assumption. There exists a shortest path from s to each vertex v.

Single-source shortest paths API

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

	public class SP	
	SP(EdgeWeightedDigraph 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*
boolean	**hasPathTo(int v)**	*is there a path from s to v?*

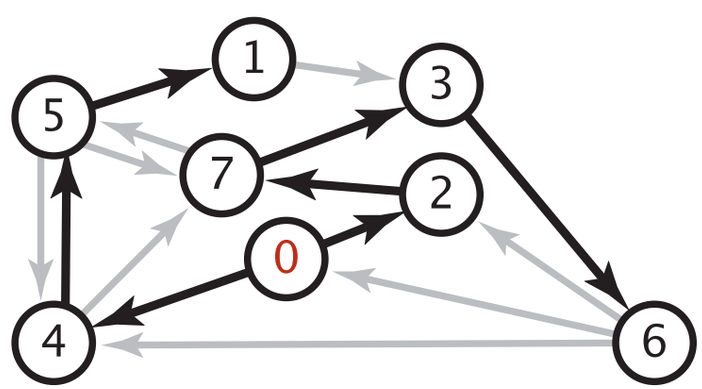
Data structures for single-source shortest paths

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

Observation. A shortest path tree (SPT) solution exists. Why?

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.



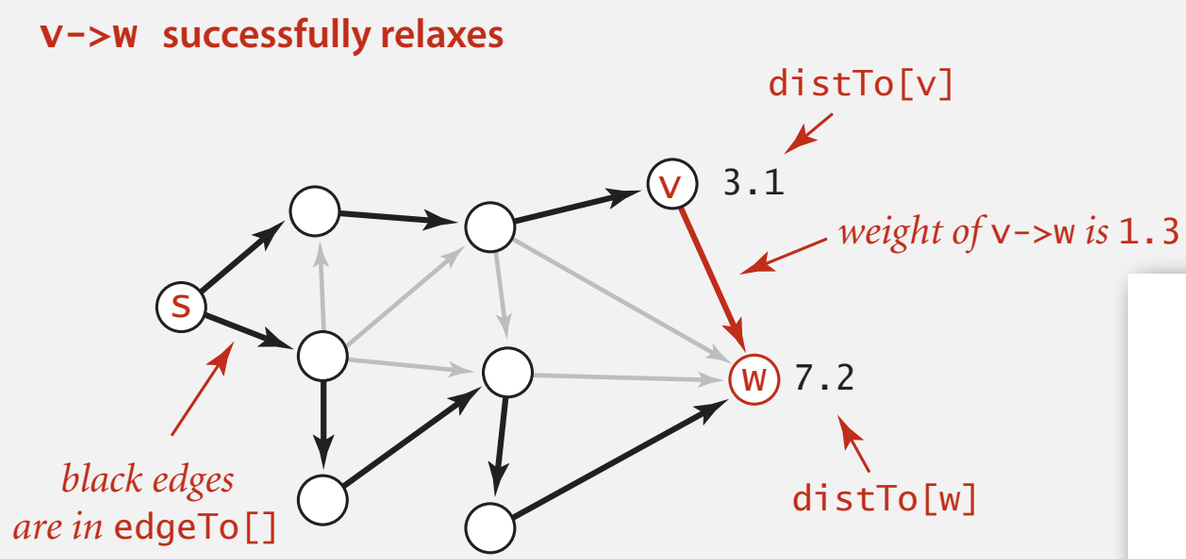
	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

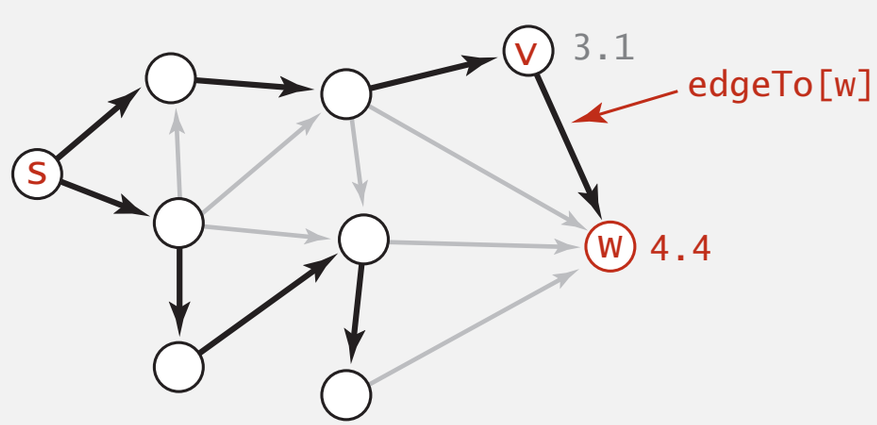
shortest path tree from 0

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `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 $e = v \rightarrow w$ 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

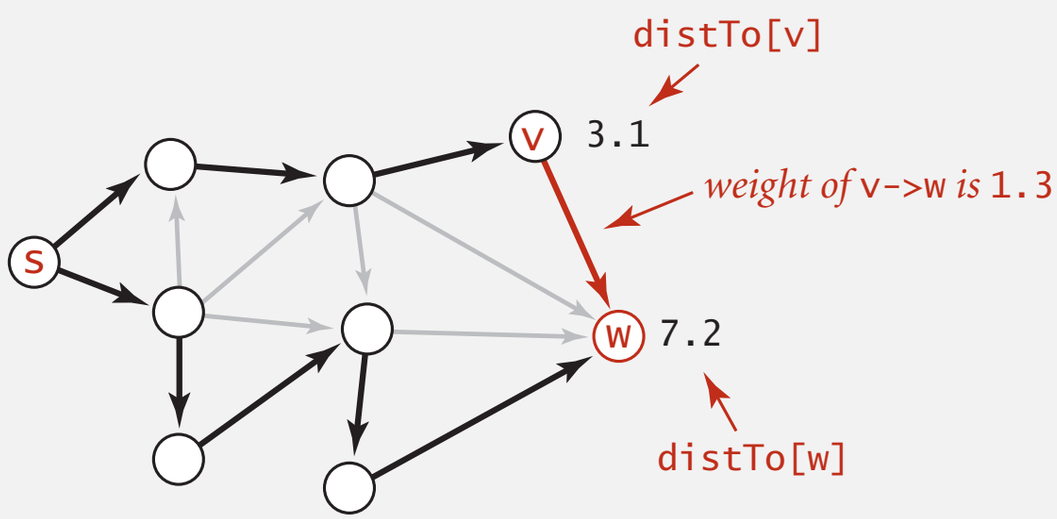
Proposition. Let G be an edge-weighted digraph.

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

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

Pf. $\Leftarrow$ [necessary]

- Suppose that `distTo[w] > distTo[v] + e.weight()` for some edge $e = v \rightarrow w$.
- 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:

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

Pf. $\Rightarrow$ [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,

$$
\begin{aligned}
\texttt{distTo[}v_k\texttt{]} &\le \texttt{distTo[}v_{k-1}\texttt{]} + e_k\texttt{.weight()} \\
\texttt{distTo[}v_{k-1}\texttt{]} &\le \texttt{distTo[}v_{k-2}\texttt{]} + e_{k-1}\texttt{.weight()} \\
&\ldots \\
\texttt{distTo[}v_1\texttt{]} &\le \texttt{distTo[}v_0\texttt{]} + e_1\texttt{.weight()}
\end{aligned}
$$

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

- Add inequalities; simplify; and substitute $\texttt{distTo[}v_0\texttt{]} = \texttt{distTo[s]} = 0$:

$$\texttt{distTo[w]} = \texttt{distTo[}v_k\texttt{]} \le \underbrace{e_k\texttt{.weight()} + e_{k-1}\texttt{.weight()} + \ldots + e_1\texttt{.weight()}}_{\text{weight of shortest path from s to w}}$$

weight of some path from s to w

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

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Proposition. Generic algorithm computes SPT from s. ← assuming SPT exists

Pf sketch.

- 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. ■

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

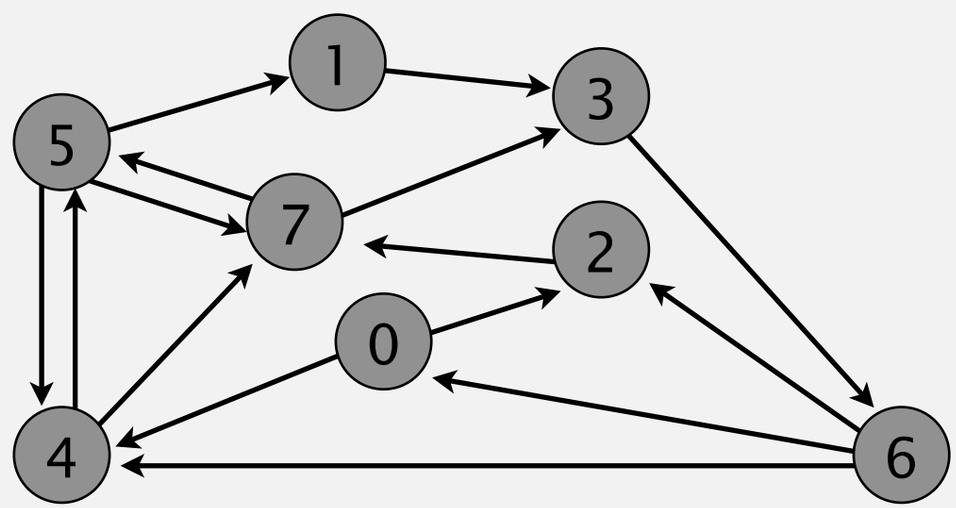
Ex 3. Bellman-Ford algorithm (no negative cycles).

Edsger W. Dijkstra: select quotes

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 incident from that vertex.

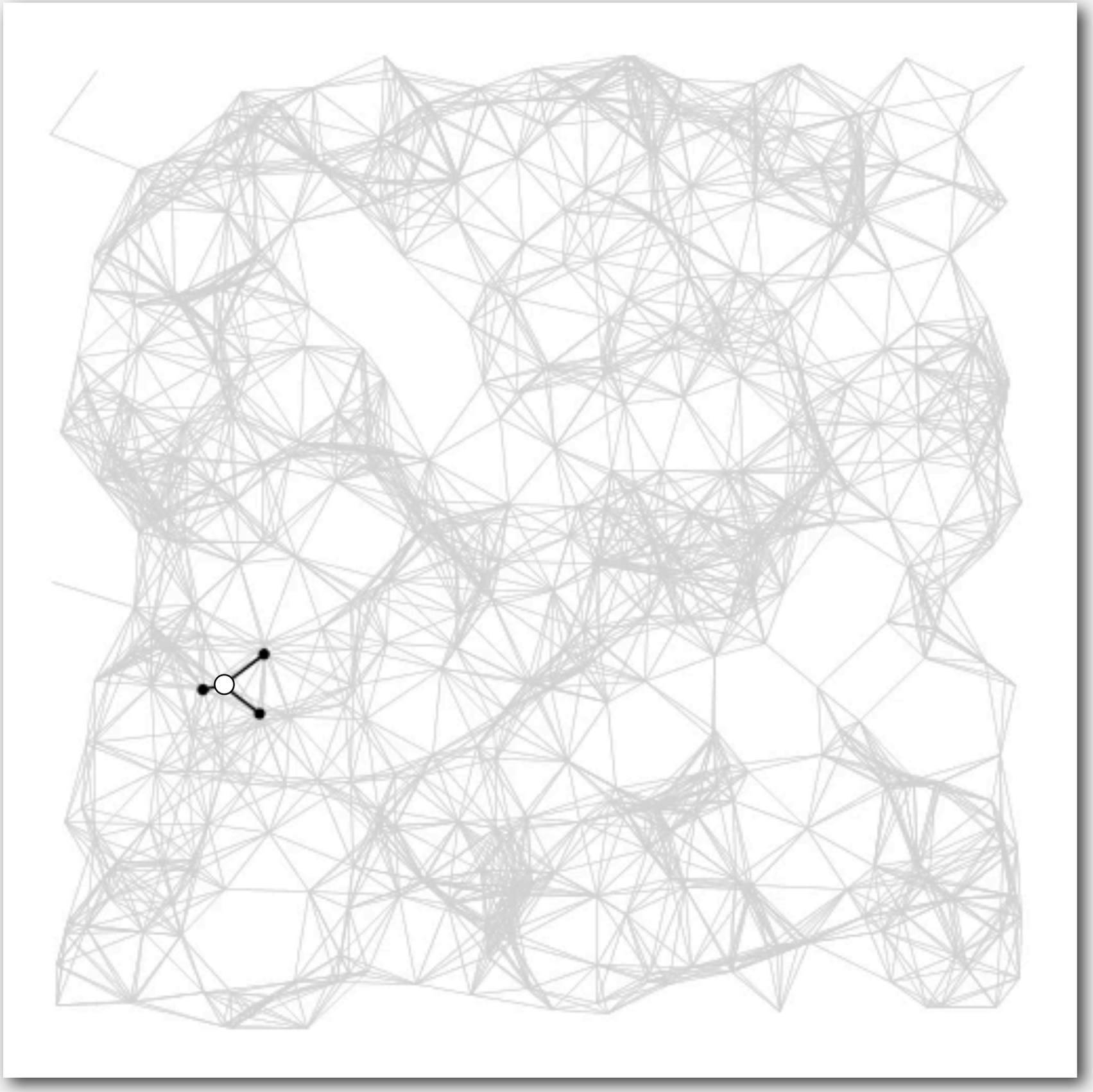
```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



v	distTo[v]	edgeTo[v]
0	0.00	-
1		
2		
3		
4		
5		
6		
7		

Dijkstra's algorithm visualization

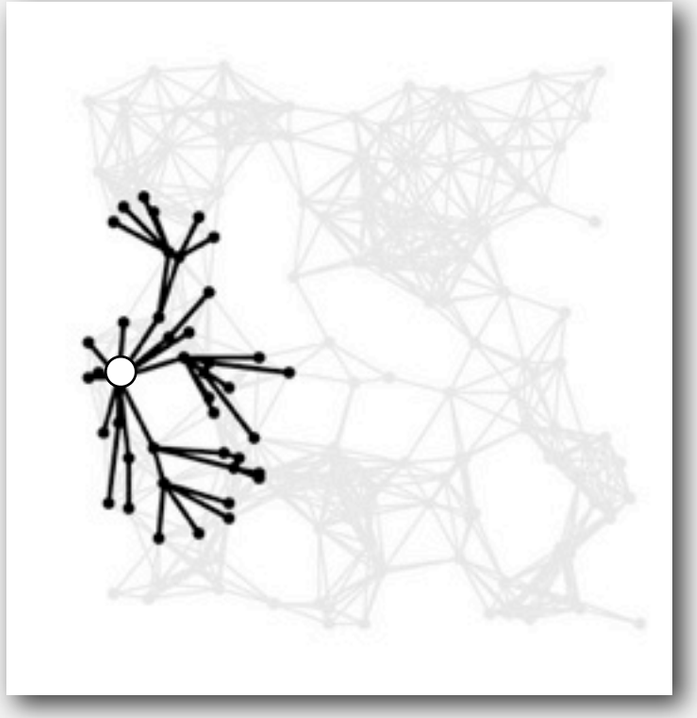
Dijkstra's algorithm visualization



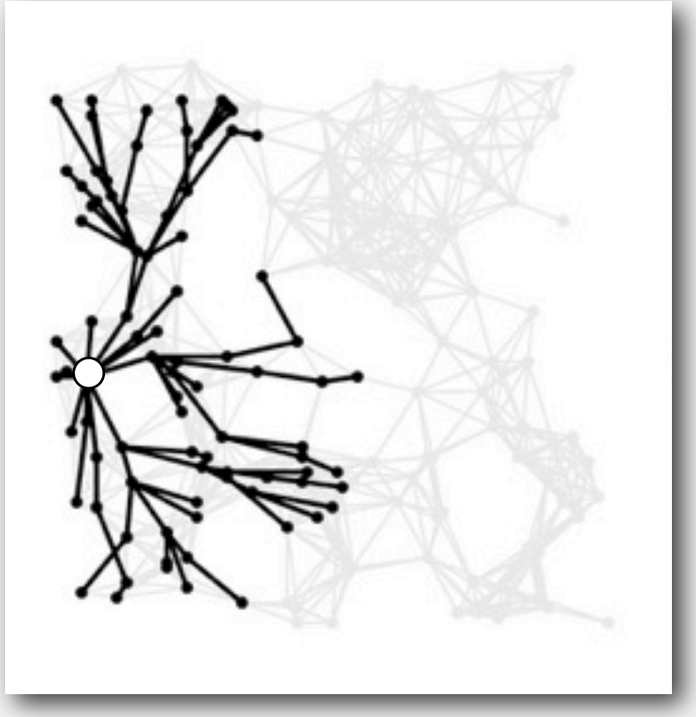
Shortest path trees

- 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 incident from that vertex.

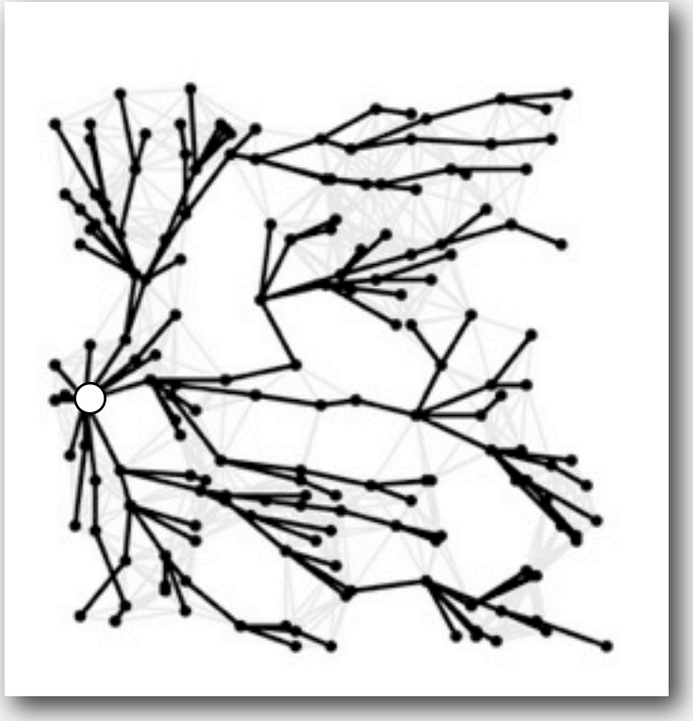
25%



50%



100%



Dijkstra's algorithm: correctness proof

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

Pf.

- Each edge $e = v \rightarrow w$ 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 ← edge weights are nonnegative and we choose lowest `distTo[]` value at each step
- Thus, upon termination, shortest-paths optimality conditions hold. ▪

Dijkstra's algorithm: Java implementation

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

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

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V^2
binary heap	log V	log V	log V	E log V
Fibonacci heap	1 †	log V †	1 †	E + V log V

† amortized

Bottom line.

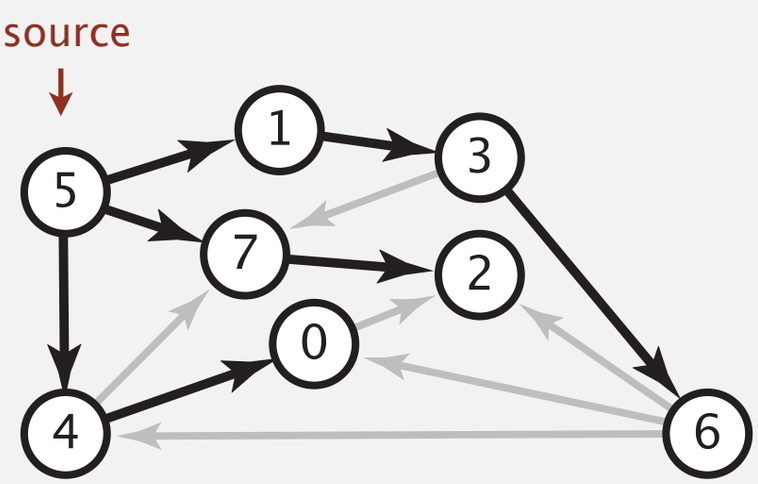
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- Fibonacci heap best in theory, but not worth implementing.

Acyclic edge-weighted digraphs

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

A. Yes!

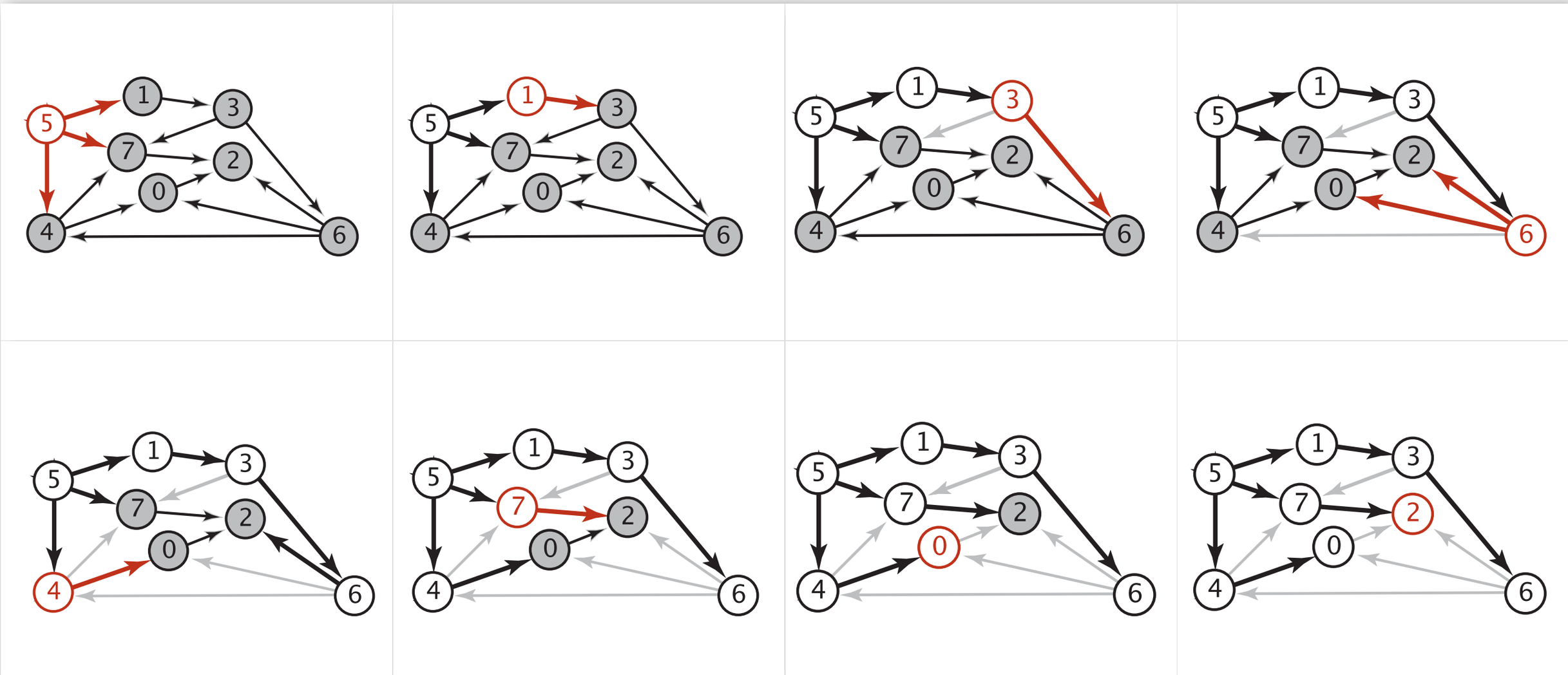
```
5->4  0.35
4->7  0.37
5->7  0.28
5->1  0.32
4->0  0.38
0->2  0.26
3->7  0.39
1->3  0.29
7->2  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



Shortest paths in edge-weighted DAGs

Topological sort algorithm.

- Consider vertices in topologically order.
- Relax all edges incident from vertex.



topological order: 5 1 3 6 4 7 0 2

Shortest paths in edge-weighted DAGs

Topological sort algorithm.

- Consider vertices in topologically order.
- Relax all edges incident from vertex.

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

Pf.

- Each edge $e = v \rightarrow w$ 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. ■

Longest paths in edge-weighted DAGs

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

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

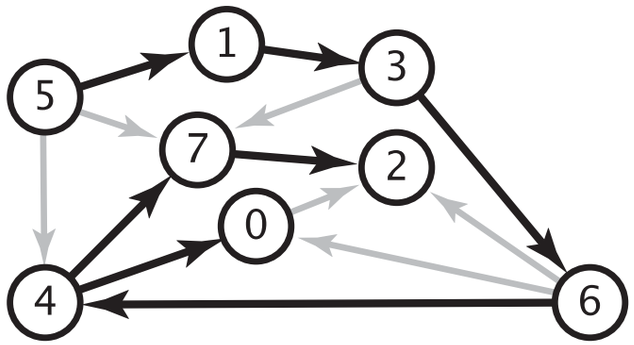
equivalent: reverse sense of equality in **`relax()`**

longest paths input

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

shortest paths input

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

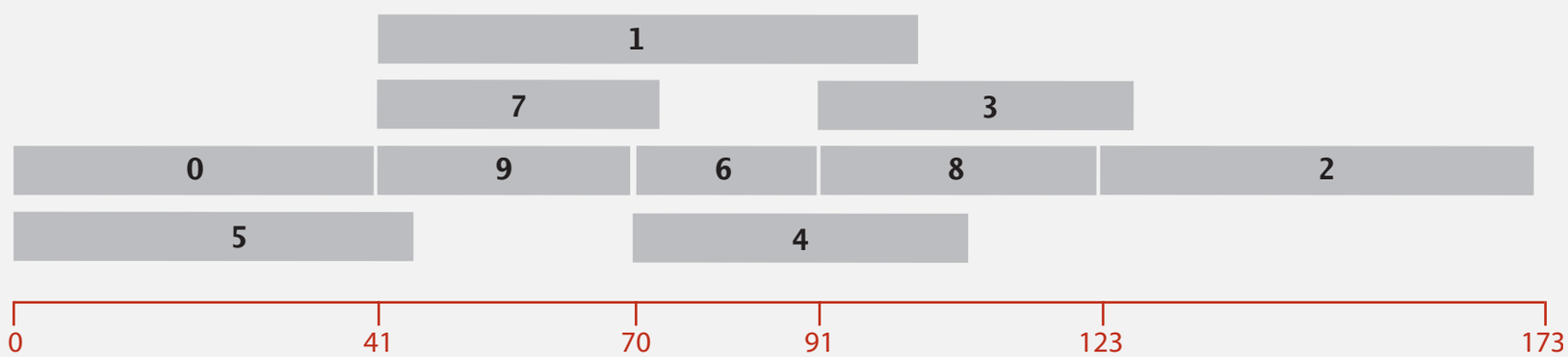


Key point. Topological sort algorithm works even with negative edge weights.

Longest paths in edge-weighted DAGs: 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	



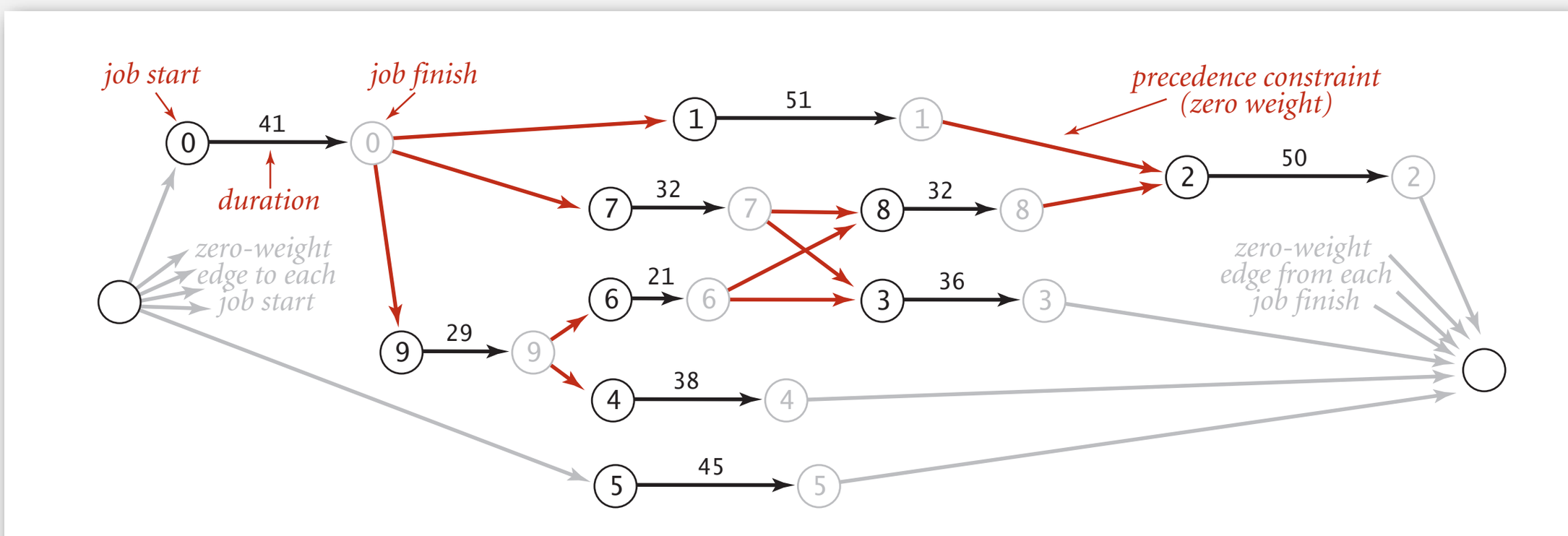
Parallel job scheduling solution

Critical path method

CPM. To solve a parallel job-scheduling problem, create acyclic edge-weighted digraph:

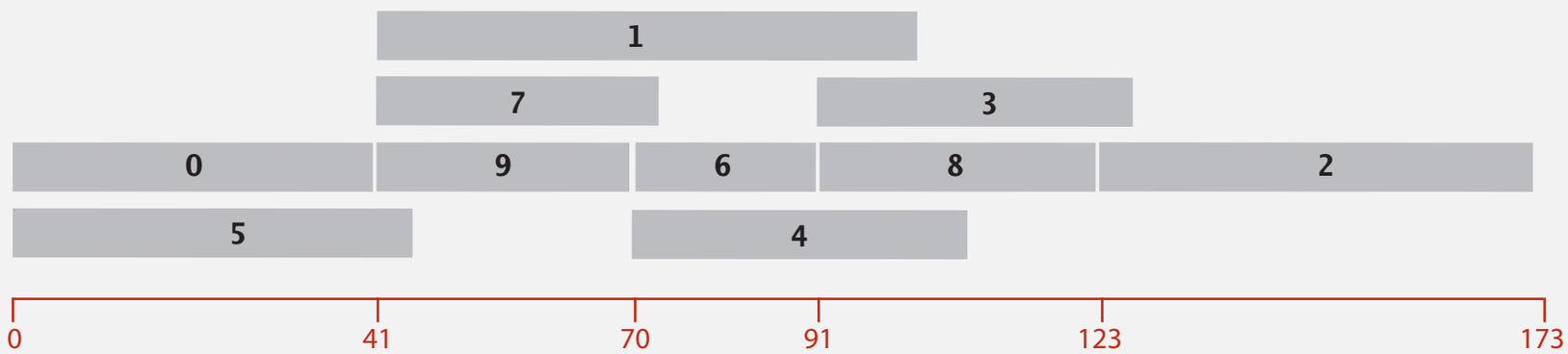
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)

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	

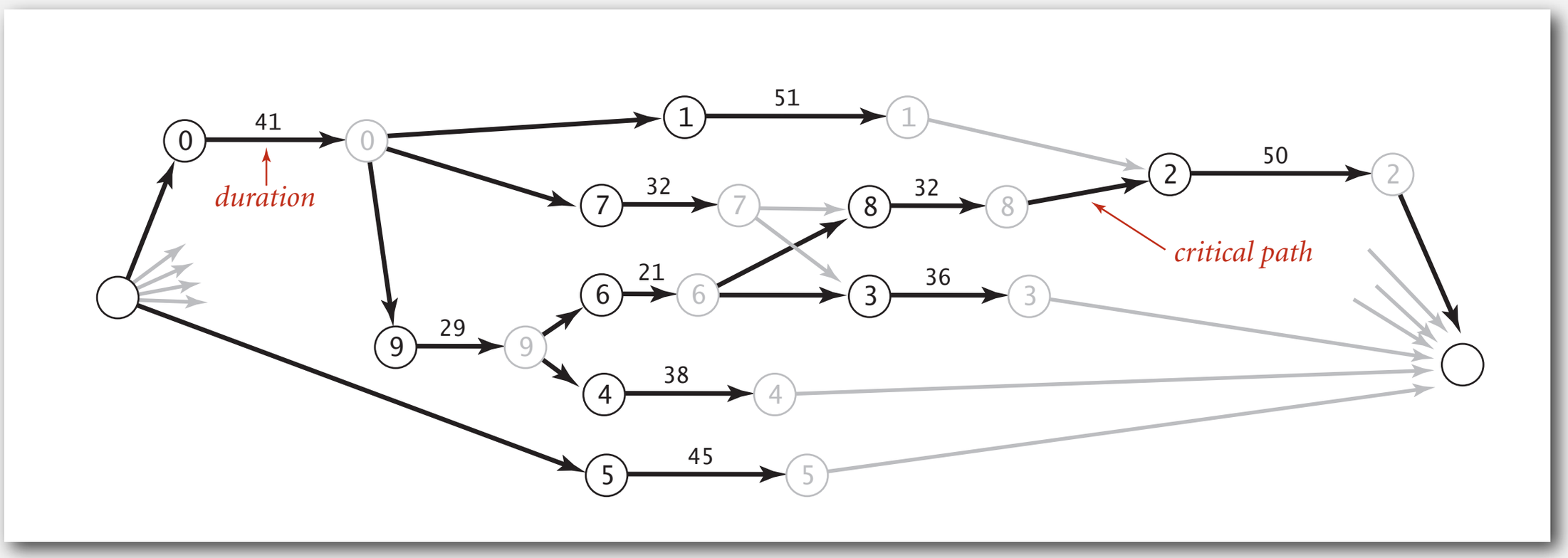


Critical path method

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

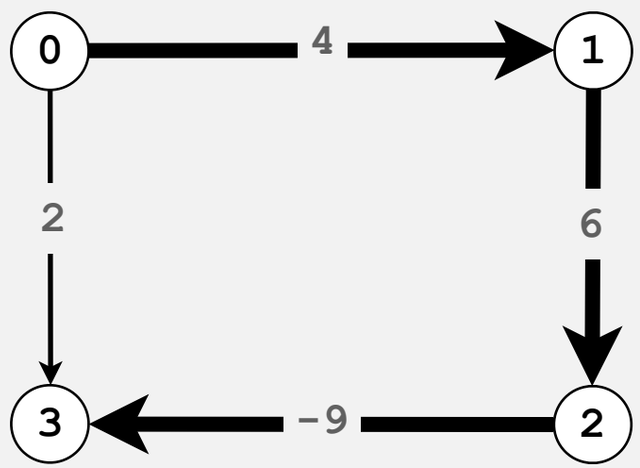


Parallel job scheduling solution



Shortest paths with negative weights: failed attempts

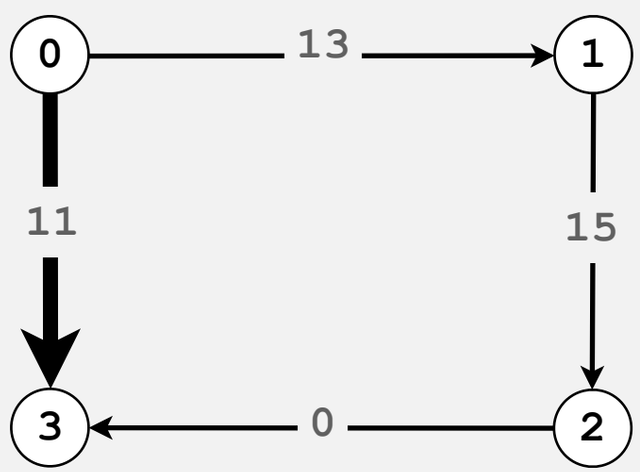
Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.

But shortest path from 0 to 3 is 0→1→2→3.

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



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

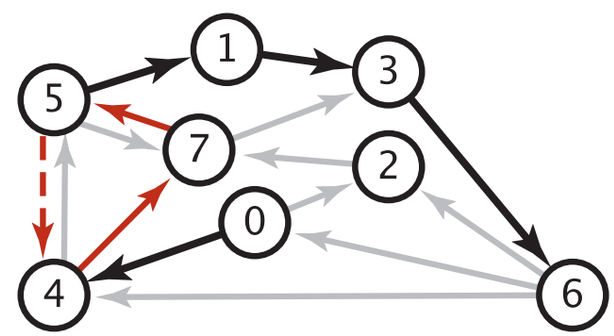
Bad news. Need a different algorithm.

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.

digraph

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



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

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Shortest paths with negative weights: dynamic programming algorithm

Dynamic programming algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat V 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);
```

phase i (relax each edge)

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After phase i, found shortest path containing at most i edges.

Bellman-Ford algorithm

Observation. If `distTo[v]` does not change during phase i, no need to relax any edge incident from v in phase $i+1$.

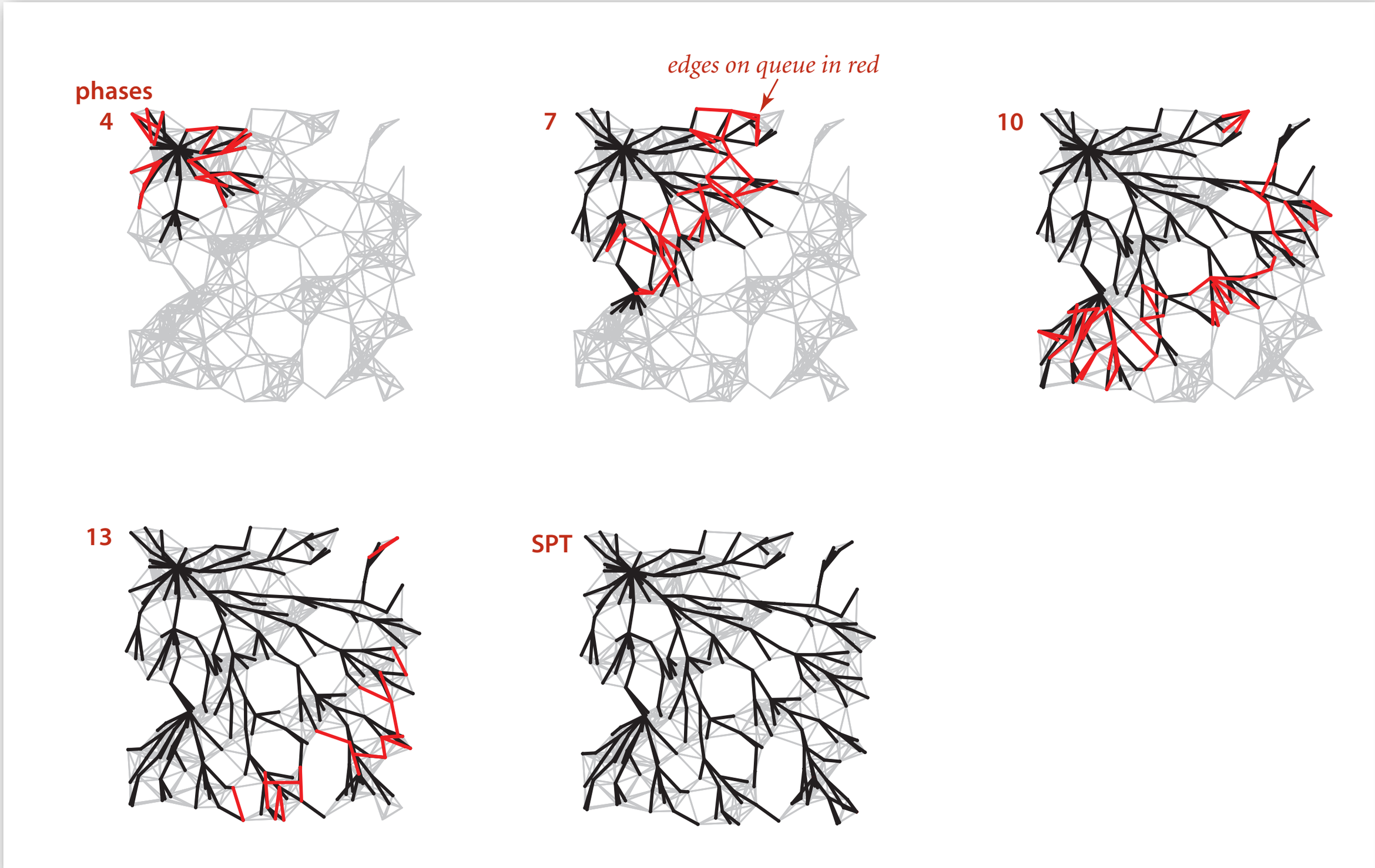
FIFO implementation. Maintain queue of vertices whose `distTo[]` changed.

be careful to keep at most one copy of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Bellman-Ford algorithm visualization



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
dynamic programming	no negative cycles	E V	E V	V
Bellman-Ford		E + V	E V	V

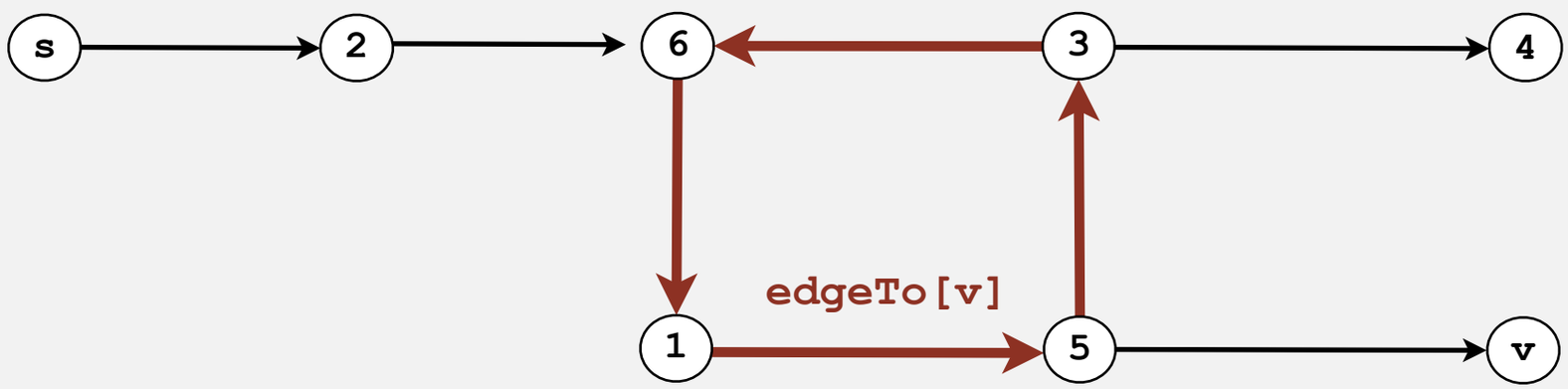
Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

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).

In practice. Check for negative cycles more frequently.

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

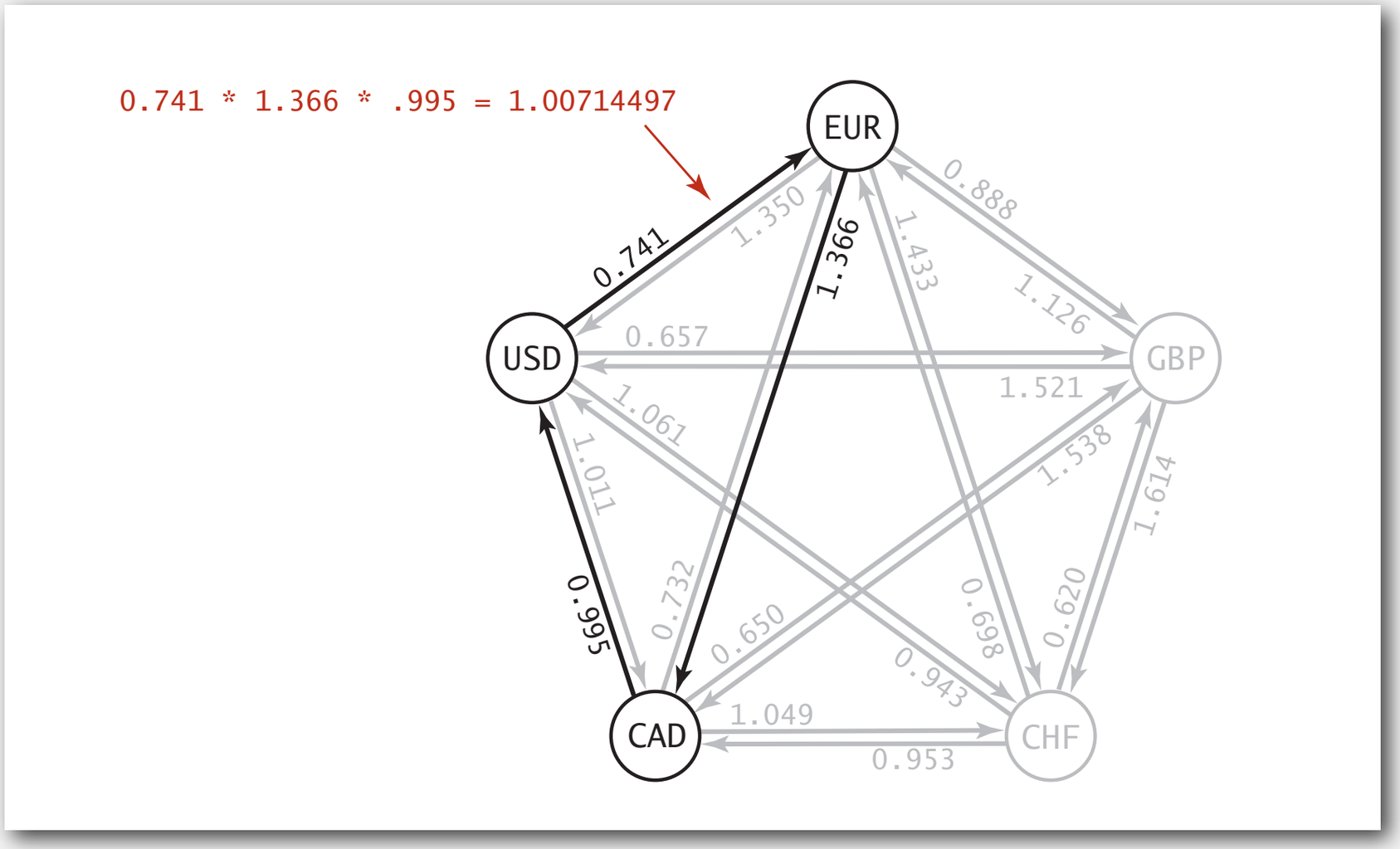
Ex. \$1,000 $\Rightarrow$ 741 Euros $\Rightarrow$ 1,012.206 Canadian dollars $\Rightarrow$ \$1,007.14497.

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

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.

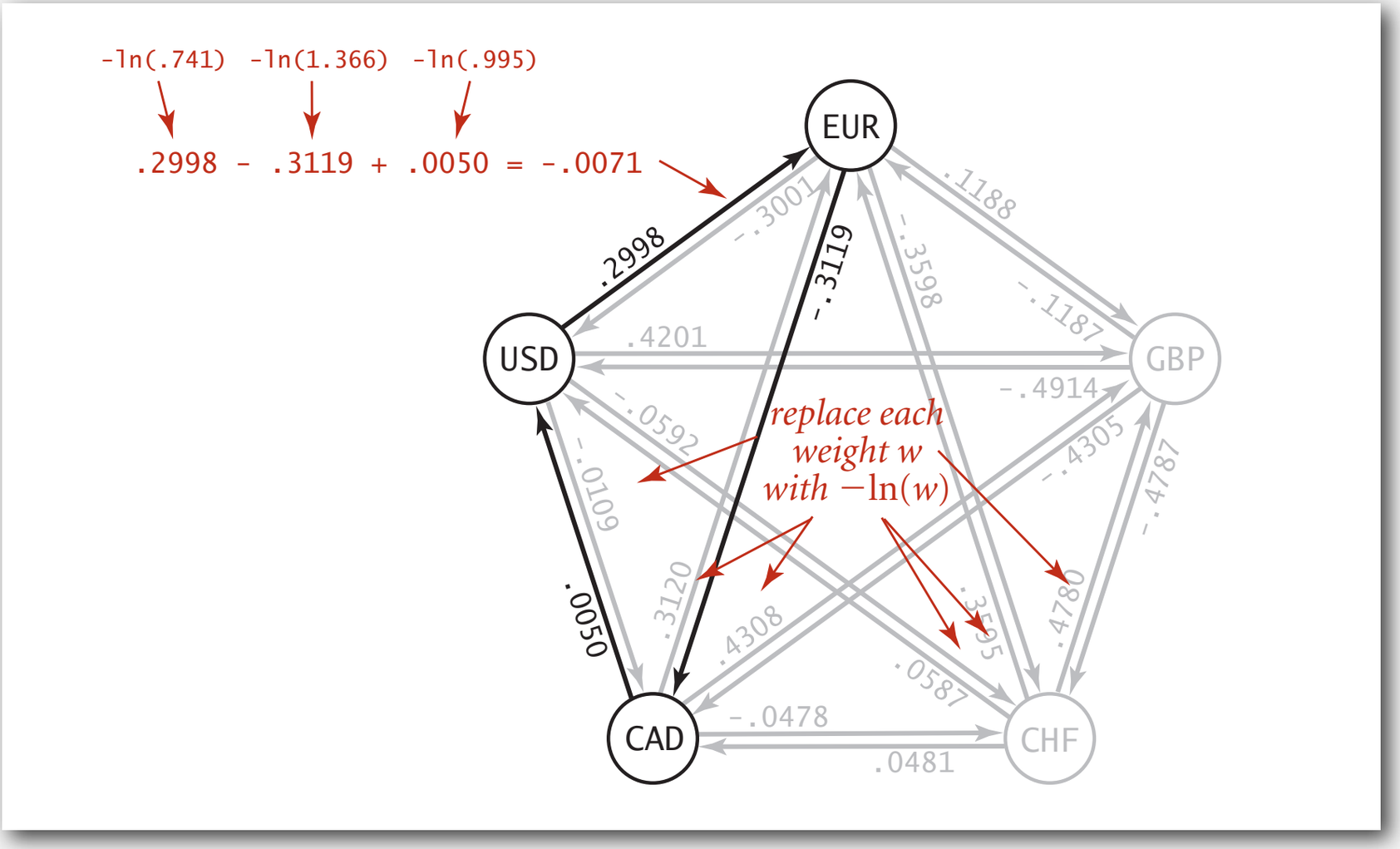


Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ 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).



Remark. Fastest algorithm is extraordinarily valuable!

Conclusion

Weighted graphs can be used to model a large variety of computational problems.

Shortest paths algorithms can sometimes be used as building blocks to solve optimization problems.

Efficiency of shortest paths depends on characteristics of the graph: Negative weights? Cycles? Negative cycles?

Releted course goals:

- **Choose** among and make use of the most important algorithms and data structures in libraries, based on knowledge of their complexity.
- **Design** algorithms for ad hoc problems by using and combining known algorithms and data structures.