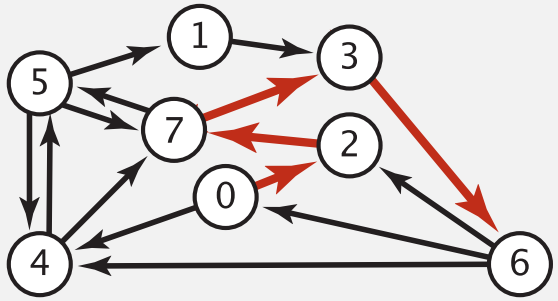
Shortest paths in an edge-weighted digraph

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

edge-weighted digraph

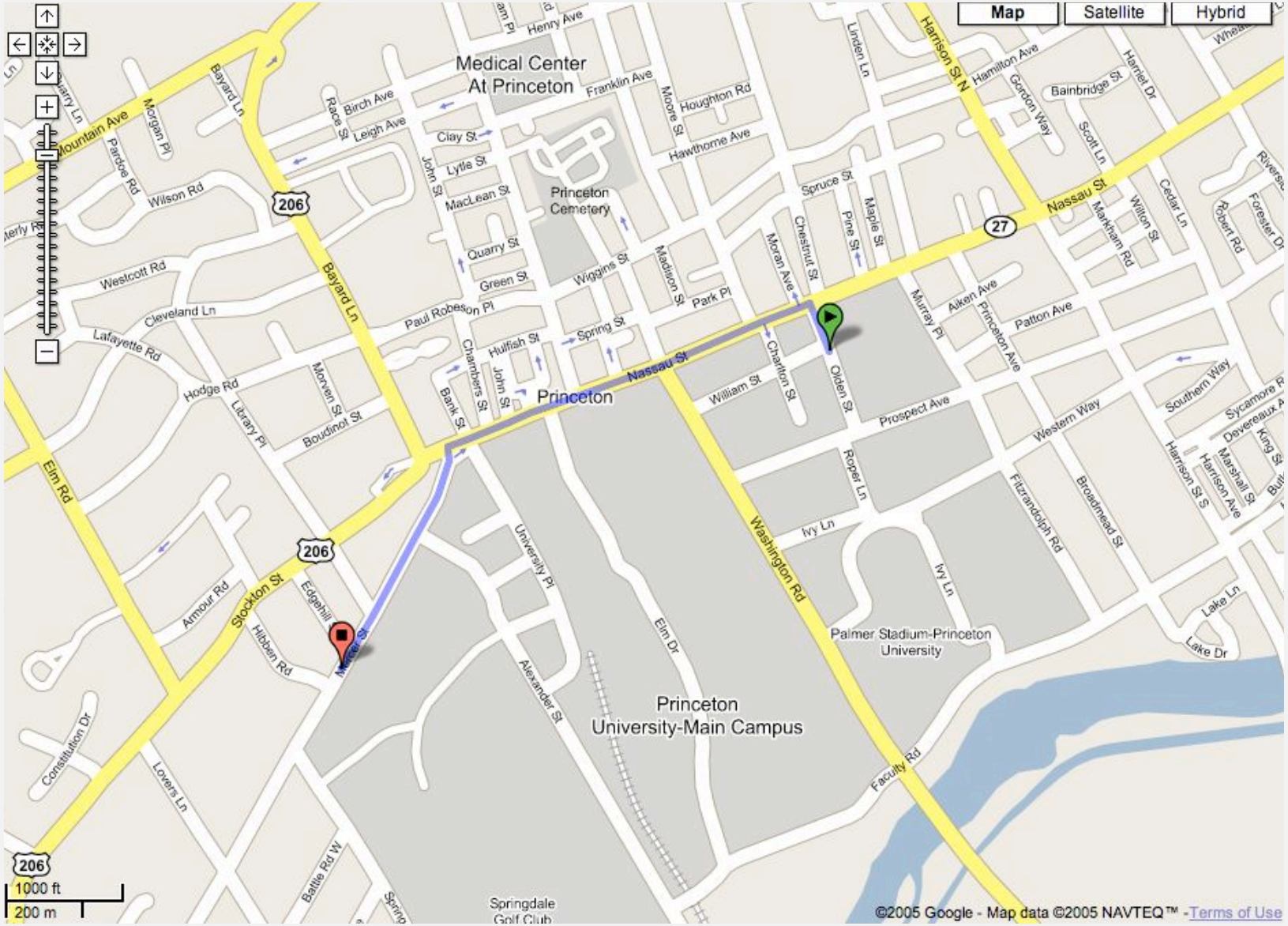
```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



shortest path from 0 to 6

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

Google maps


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

Car navigation

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

http://en.wikipedia.org/wiki/Seam_carving

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.
- Euclidean weights.

Cycles?

- No directed cycles.
- No "negative cycles."

Simplifying assumption. Shortest paths from s to each vertex v exist.

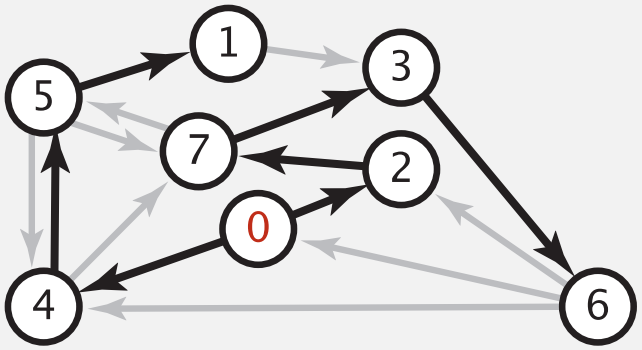
Data structures for single-source shortest paths

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

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



shortest-paths tree from 0

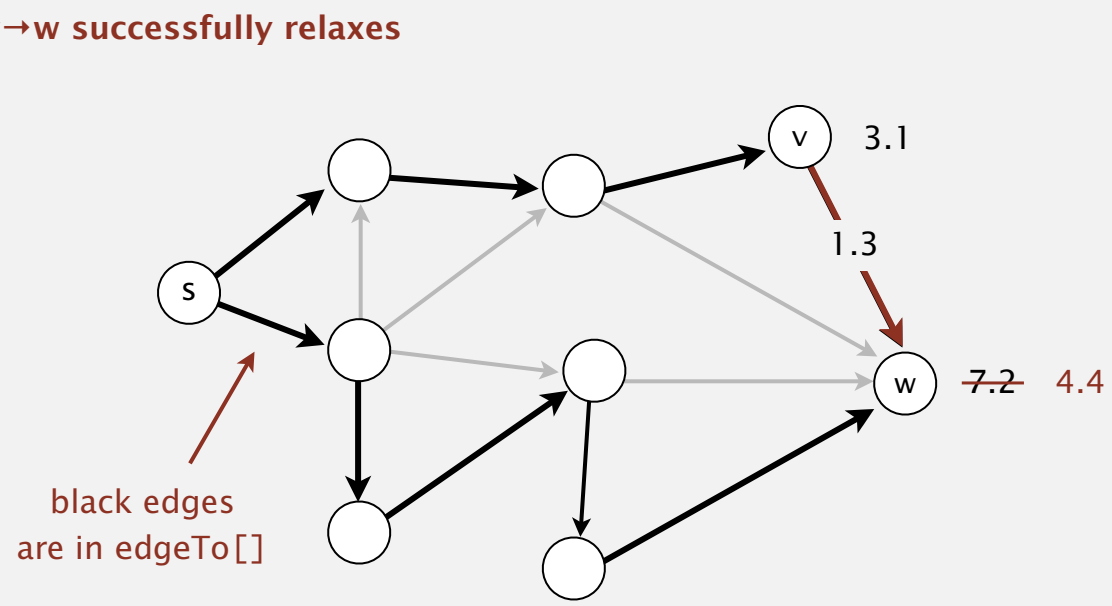
	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

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`→`w` gives shorter path to `w` through `v`,
  update both `distTo[w]` and `edgeTo[w]`.



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`→`w` gives shorter path to `w` through `v`,
  update both `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

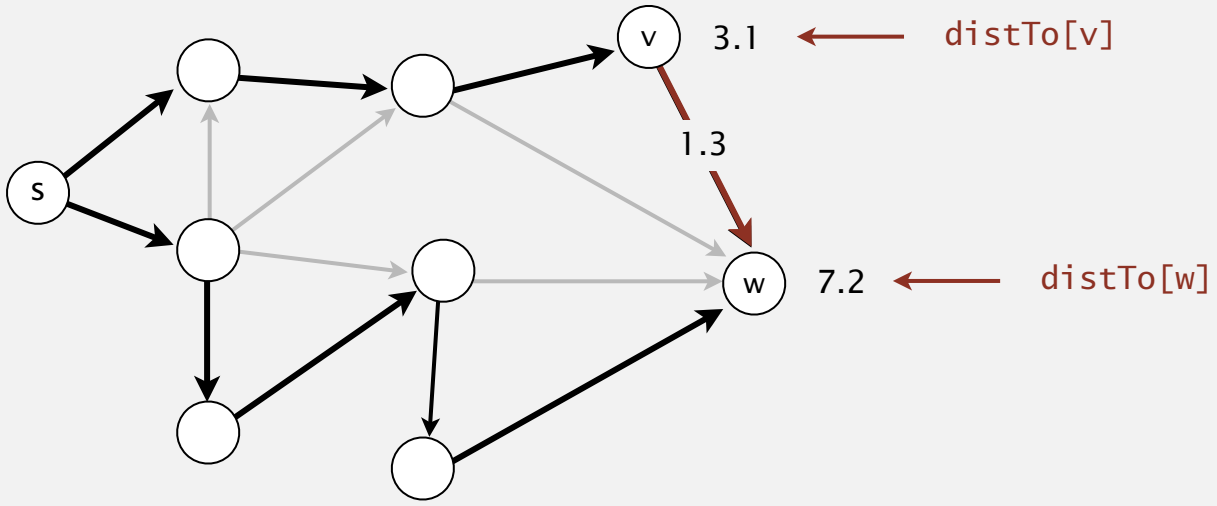
Proposition. Let G be an edge-weighted digraph.

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`→`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:

- 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,

$$\begin{aligned} \texttt{distTo}[v_1] &\le \texttt{distTo}[v_0] + \texttt{e}_1\texttt{.weight()} \\ \texttt{distTo}[v_2] &\le \texttt{distTo}[v_1] + \texttt{e}_2\texttt{.weight()} \\ &\ldots \\ \texttt{distTo}[v_k] &\le \texttt{distTo}[v_{k-1}] + \texttt{e}_k\texttt{.weight()} \end{aligned}$$

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

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

$$\texttt{distTo}[w] = \texttt{distTo}[v_k] \le \underbrace{\texttt{e}_1\texttt{.weight()} + \texttt{e}_2\texttt{.weight()} + \ldots + \texttt{e}_k\texttt{.weight()}}_{\text{weight of shortest path from s to w}}$$

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

weight of some path from s 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 (if it exists) from `s`.

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?

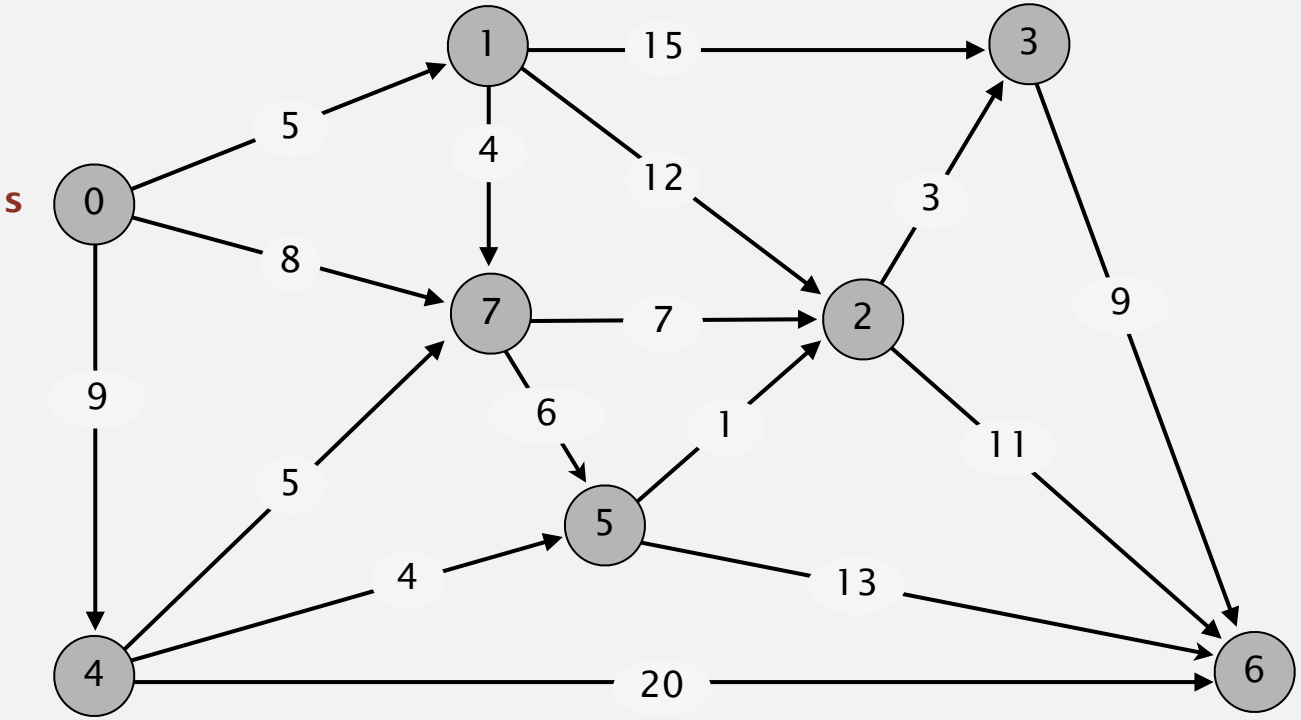
Ex 1. Dijkstra's algorithm (nonnegative weights).

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

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

Dijkstra's algorithm demo

- 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.

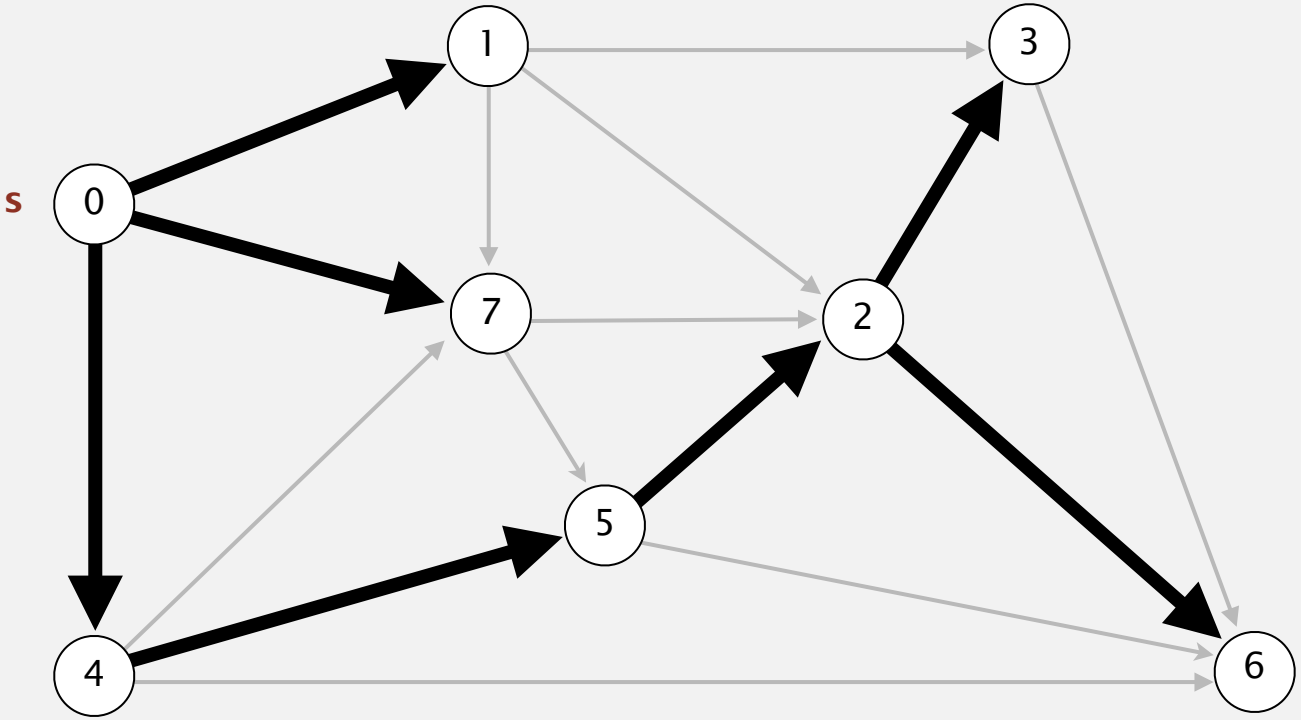


an edge-weighted digraph

```
0→1   5.0
0→4   9.0
0→7   8.0
1→2  12.0
1→3  15.0
1→7   4.0
2→3   3.0
2→6  11.0
3→6   9.0
4→5   4.0
4→6  20.0
4→7   5.0
5→2   1.0
5→6  13.0
7→5   6.0
7→2   7.0
```

Dijkstra's algorithm demo

- 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.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

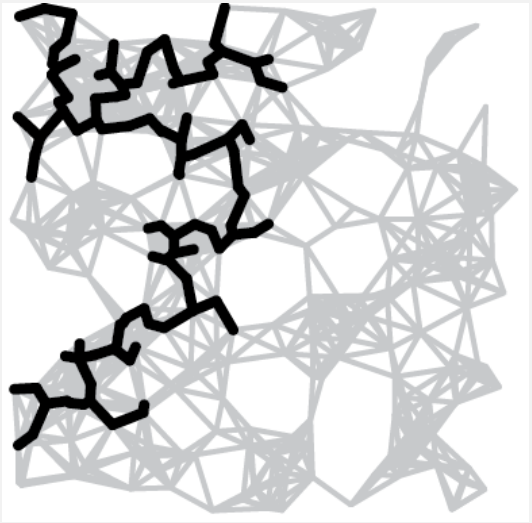
Computing spanning trees in graphs

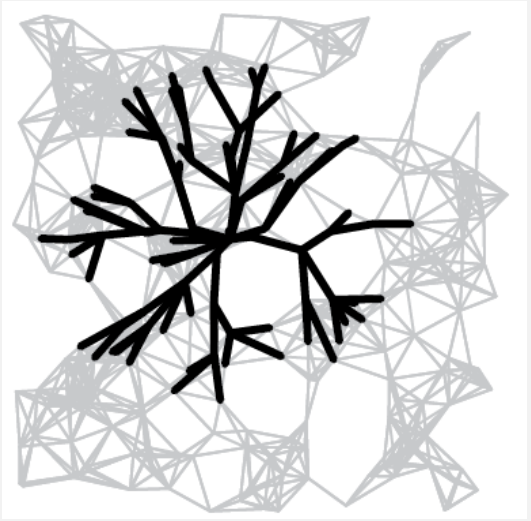
Dijkstra's algorithm seem familiar?

- 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.

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
unordered array	1	V	1	V^2
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	d $\log_d$ V	d $\log_d$ V	$\log_d$ V	E $\log_{E/V}$ V
Fibonacci heap (Fredman-Tarjan 1984)	1 †	log V †	1 †	E + V log V

† amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- d-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Acyclic edge-weighted digraphs

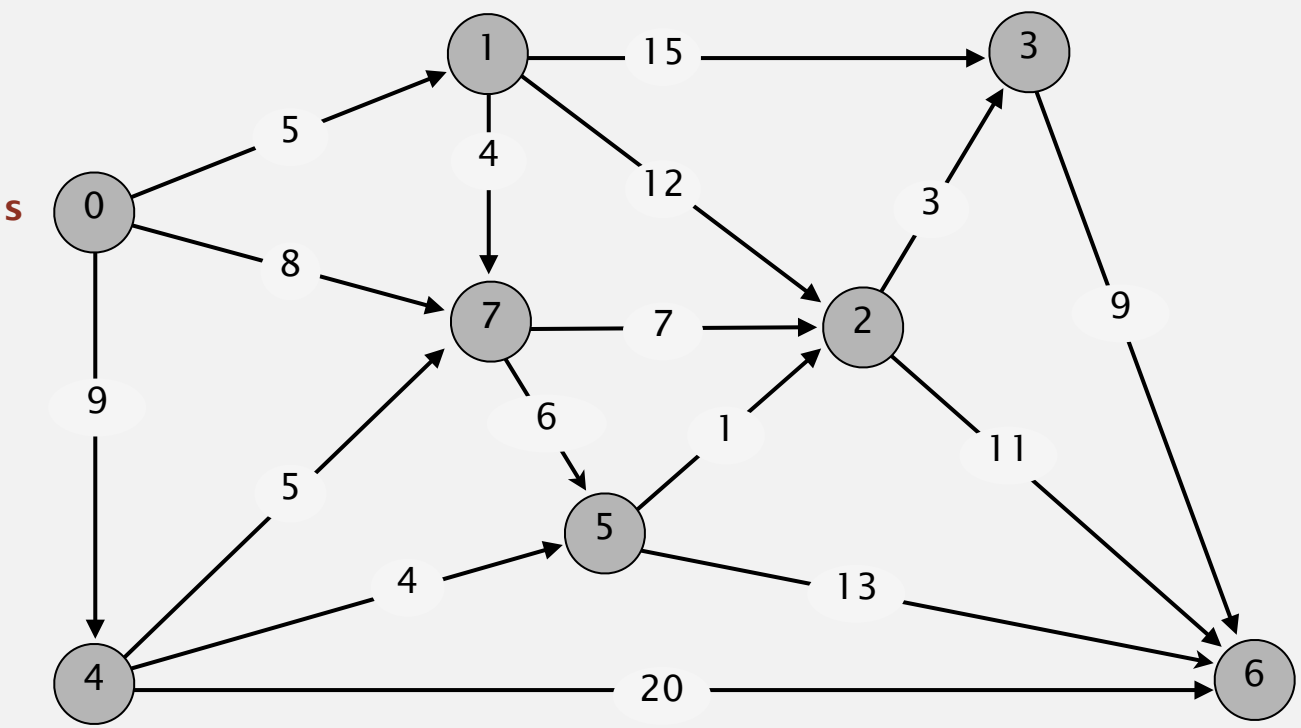
Q. Suppose that an edge-weighted digraph has no directed cycles.

Is it easier to find shortest paths than in a general digraph?

A. Yes!

Acyclic shortest paths demo

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

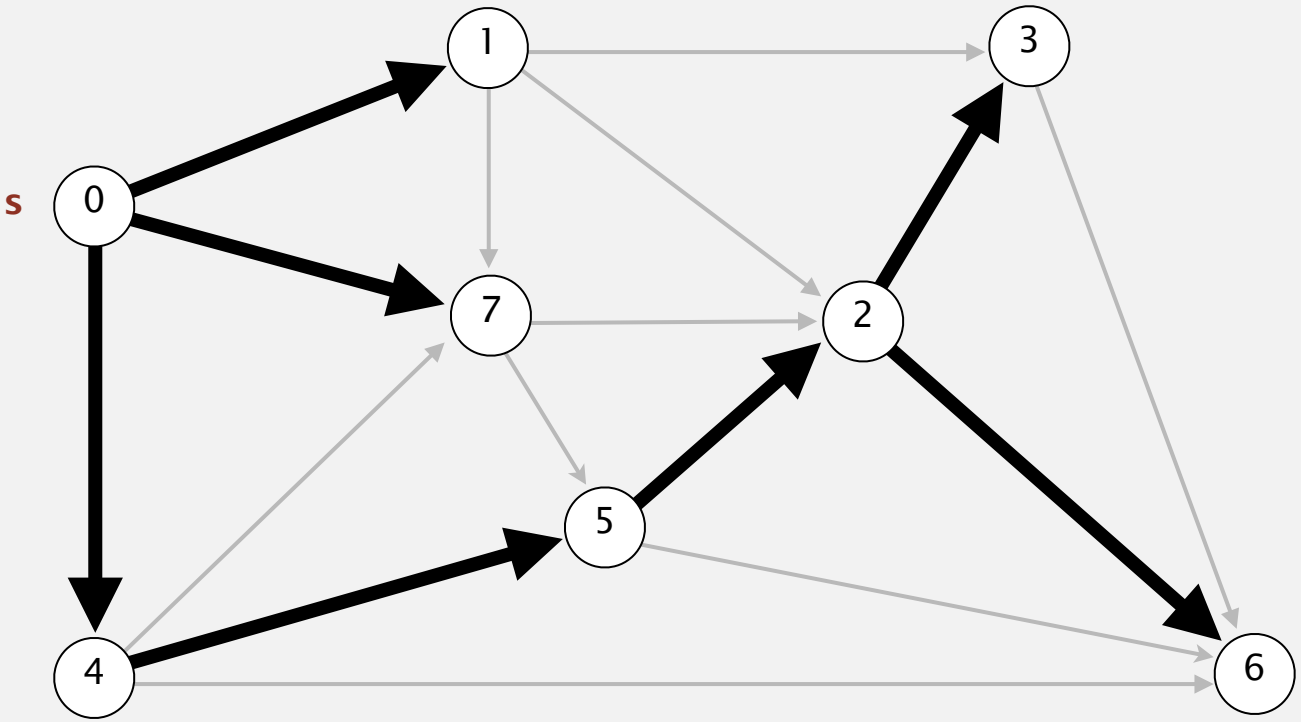


an edge-weighted DAG

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Acyclic shortest paths demo

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



0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Shortest paths in edge-weighted DAGs

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 DAGs

```
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

Content-aware resizing

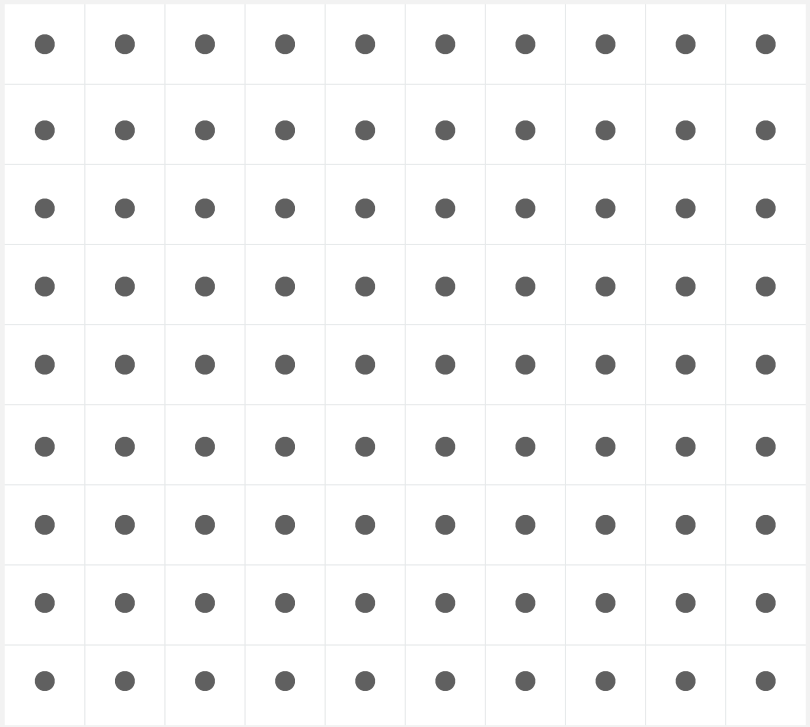
Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.

In the wild. Photoshop CS 5, Imagemagick, GIMP, ...

Content-aware resizing

To find vertical seam:

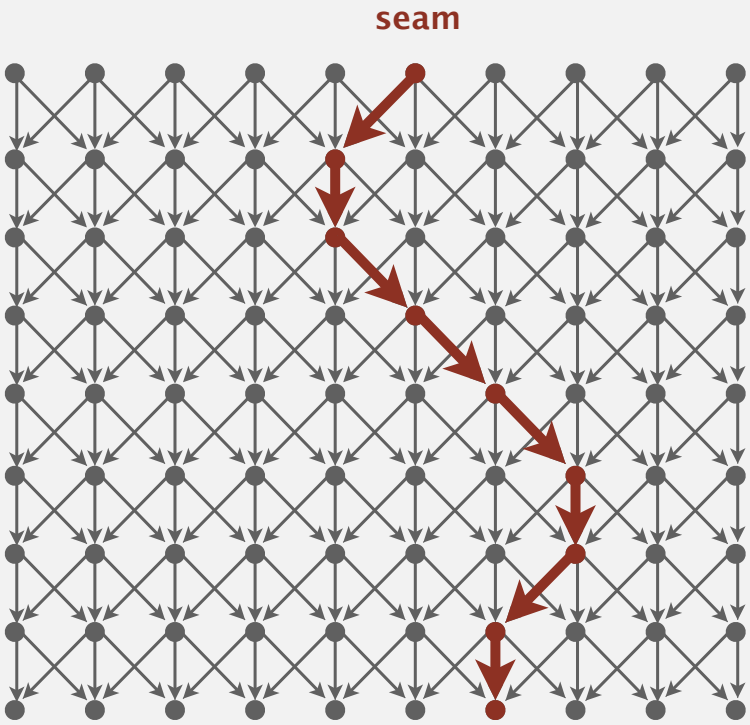
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To find vertical seam:

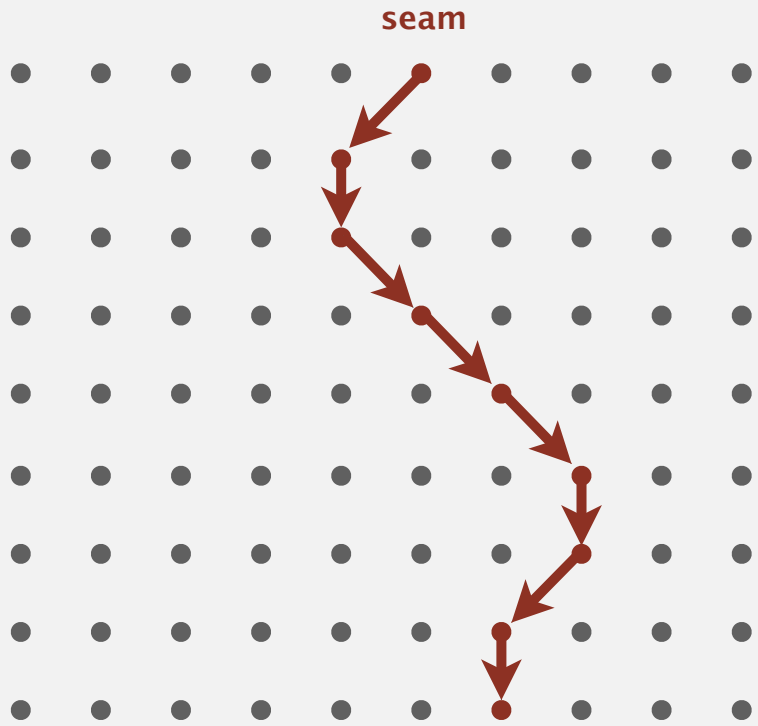
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To remove vertical seam:

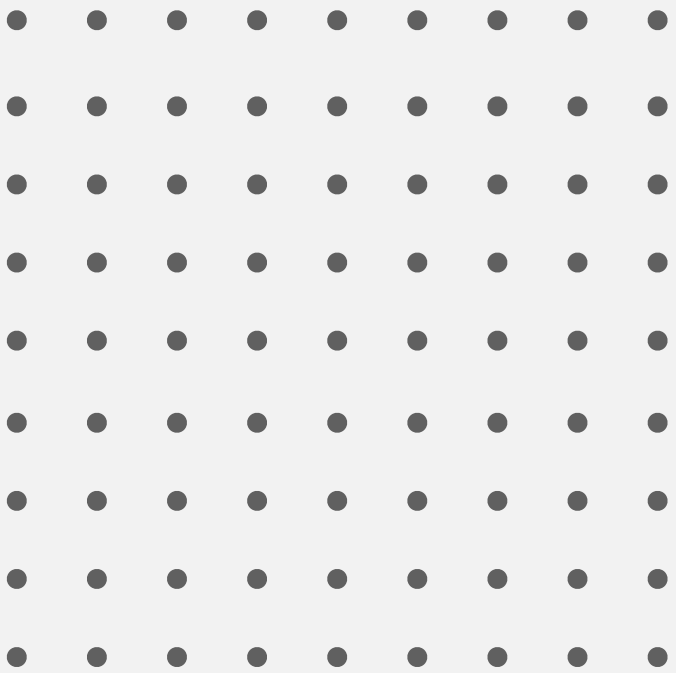
- Delete pixels on seam (one in each row).



Content-aware resizing

To remove vertical seam:

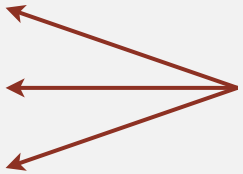
- Delete pixels on seam (one in each row).



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.



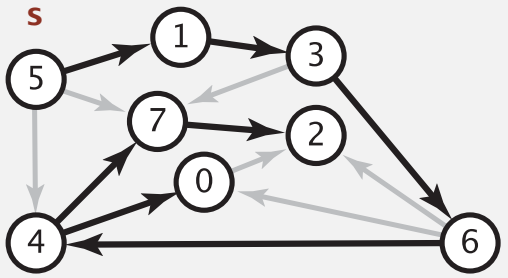
equivalent: reverse sense of equality in `relax()`

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
```

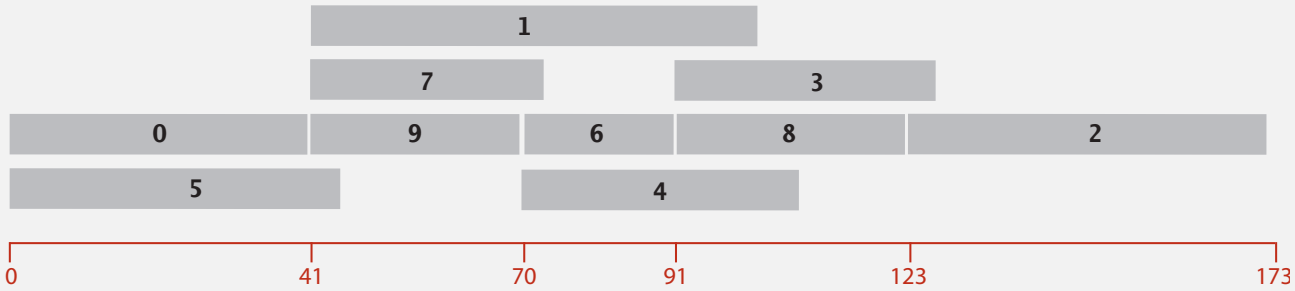


Key point. Topological sort algorithm works even with negative weights.

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



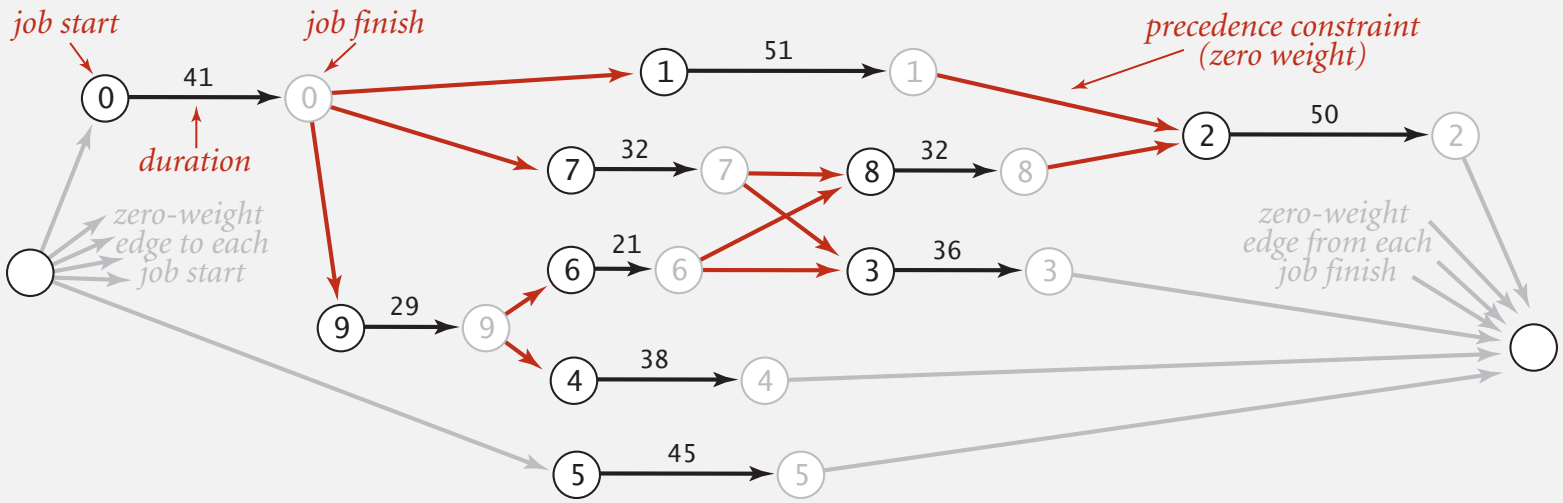
Parallel job scheduling solution

Critical path method

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

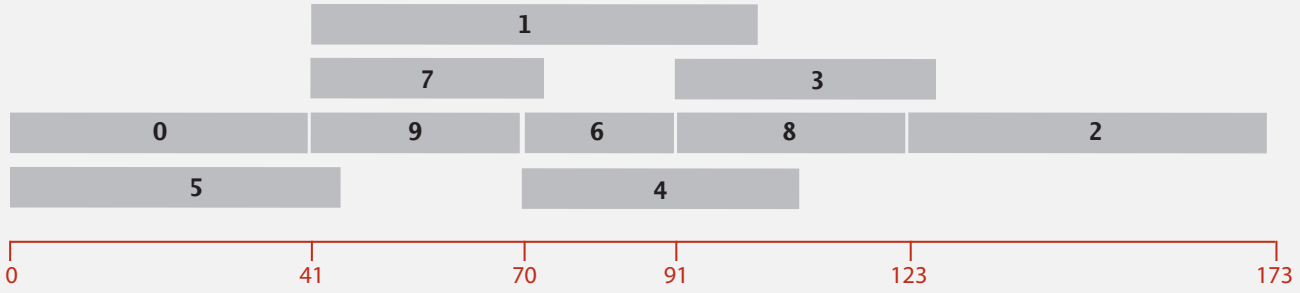
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

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	

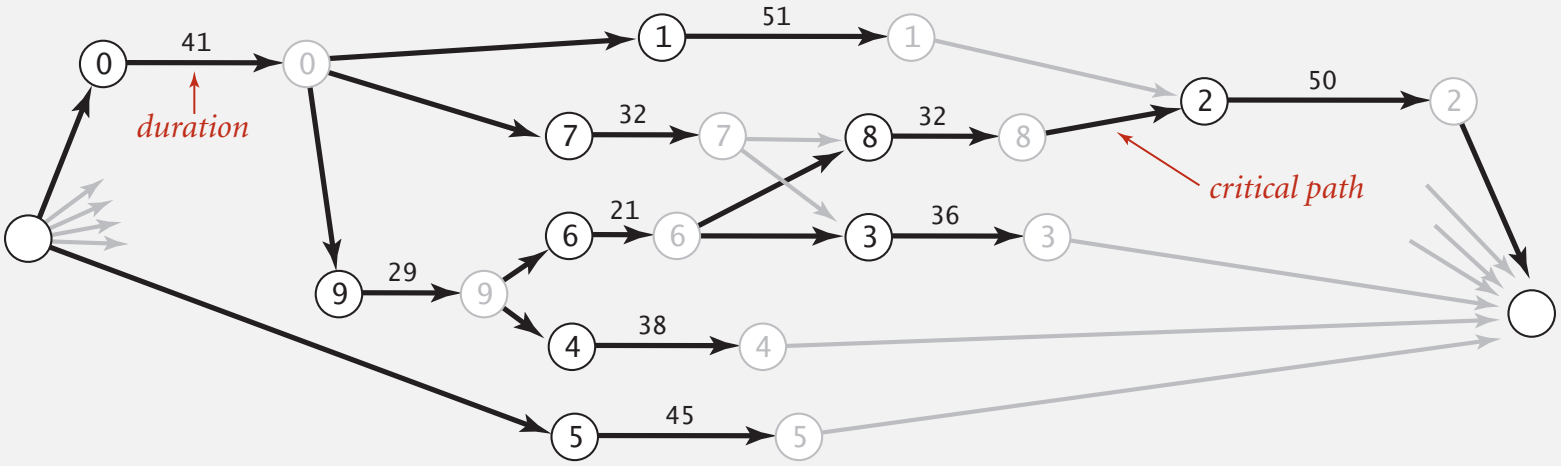


Critical path method

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

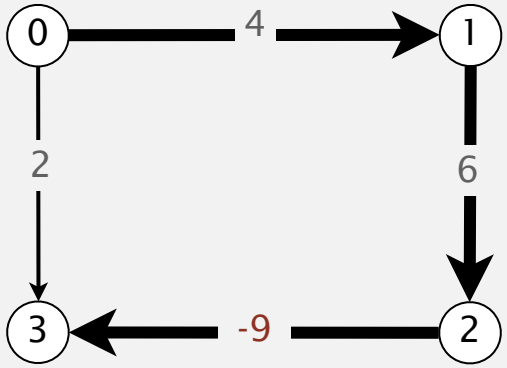


Parallel job scheduling solution



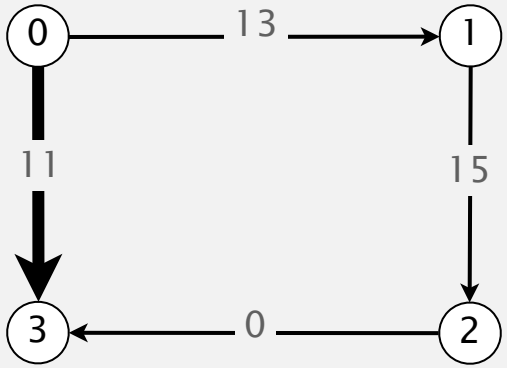
Shortest paths with negative weights: failed attempts

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.

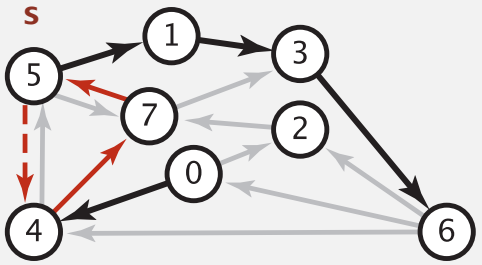
Conclusion. Need a different algorithm.

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

Bellman-Ford algorithm

Bellman-Ford algorithm

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

Repeat V times:

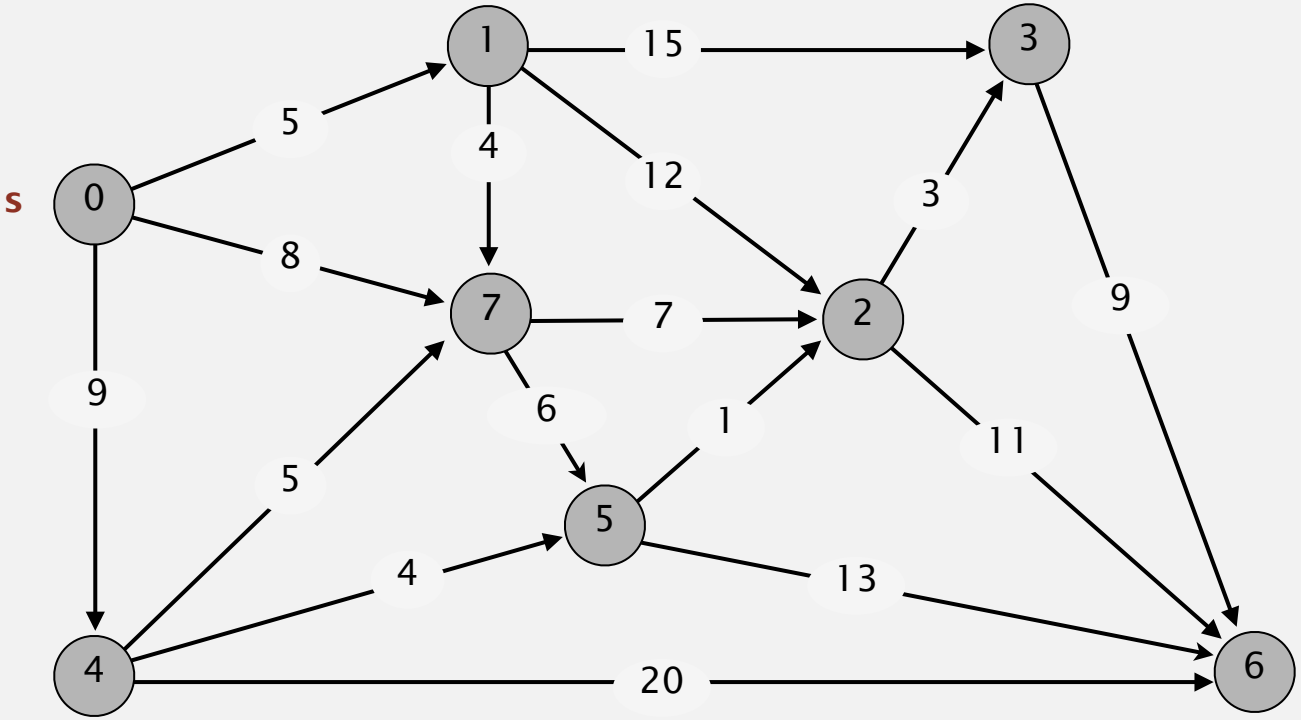
- **Relax each edge.**

```
for (int i = 0; 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 demo

Repeat V times: relax all E edges.

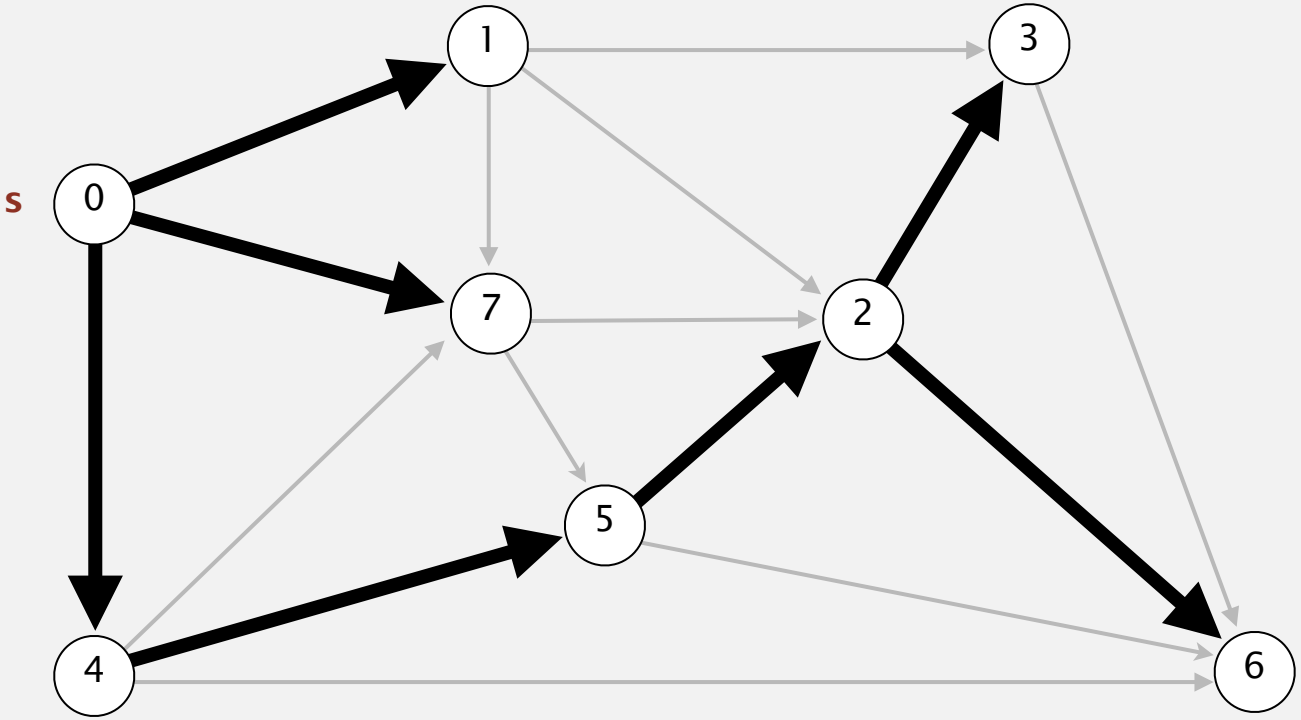


an edge-weighted digraph

```
0→1   5.0
0→4   9.0
0→7   8.0
1→2  12.0
1→3  15.0
1→7   4.0
2→3   3.0
2→6  11.0
3→6   9.0
4→5   4.0
4→6  20.0
4→7   5.0
5→2   1.0
5→6  13.0
7→5   6.0
7→2   7.0
```

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



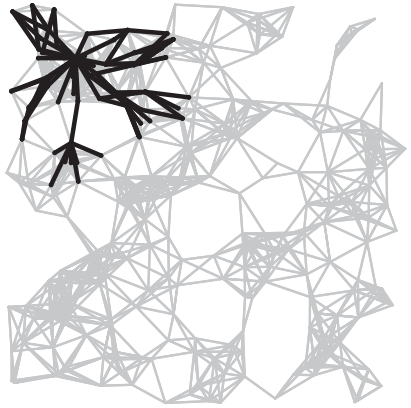
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

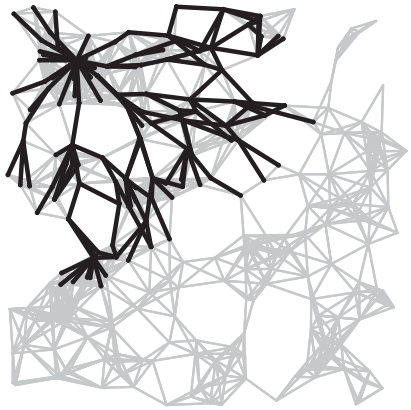
Bellman-Ford algorithm visualization

passes

4



7



10

13

SPT

Bellman-Ford algorithm: analysis

Bellman-Ford algorithm

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

Repeat V times:

- **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 pass i, found shortest path containing at most i edges.

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.

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

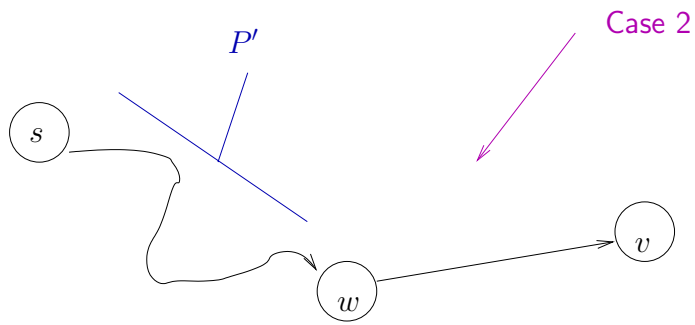
Shortest-paths is a broadly useful problem-solving model.

The Bellman-Ford Algorithm

Optimal Substructure (Formal)

Lemma: Let $G = (V, E)$ be a directed graph with edge lengths c_e and source vertex s.
[G might or might not have a negative cycle]
For every $v \in V$, $i \in \{1, 2, \ldots\}$, let P = shortest s-v path with at most i edges. (Cycles are permitted.)

Case 1: If P has $\leq (i-1)$ edges, it is a shortest s-v path with $\leq (i-1)$ edges.
Case 2: If P has i edges with last hop (w, v), then P' is a shortest s-w path with $\leq (i-1)$ edges.



Proof of Optimal Substructure

Case 1: By (obvious) contradiction.

Case 2: If Q (from s to w, $\leq (i-1)$ edges) is shorter than P' then $Q + (w, v)$ (from s to v, $\leq i$ edges) is shorter than $P' + (w, v)$ $(= P)$ which contradicts the optimality of P. QED!

The Recurrence

Notation: Let $L_{i,v}$ = minimum length of a s-v path with $\leq i$ edges.

- With cycles allowed
- Defined as $+\infty$ if no s-v paths with $\leq i$ edges

Recurrence: For every $v \in V$, $i \in \{1, 2, \ldots\}$

$$L_{i,v} = \min \left\{ \begin{array}{ll} L_{(i-1),v} & \text{Case 1} \\ \min_{(u,v)\in E}\{L_{(i-1),w} + c_{wv}\} & \text{Case 2} \end{array} \right\}$$

Correctness: Brute-force search from the only (1+in-deg(v)) candidates (by the optimal substructure lemma).

If No Negative Cycles

Now: Suppose input graph G has no negative cycles.

$\Rightarrow$ Shortest paths do not have cycles

[removing a cycle only decreases length]

$\Rightarrow$ Have $\leq (n-1)$ edges

Point: If G has no negative cycle, only need to solve subproblems up to $i = n-1$.

Subproblems: Compute $L_{i,v}$ for all $i \in \{0, 1, \ldots, n-1\}$ and all $v \in V$.

The Bellman-Ford Algorithm

Let A = 2-D array (indexed by i and v)

Base case: $A[0, s] = 0$; $A[0, v] = +\infty$ for all $v \neq s$.

For $i = 1, 2, \ldots, n-1$

For each $v \in V$

$$A[i, v] = \min \left\{ \begin{array}{l} A[i-1, v] \\ \min_{(w,v) \in E} \{A[i-1, w] + c_{wv}\} \end{array} \right\}$$

As discussed: If G has no negative cycle, then algorithm is correct [with final answers = $A[n-1, v]$'s]

Stopping Early

Note: Suppose for some $j < n-1$, $A[j,v] = A[j-1,v]$ for all vertices v.

$\Rightarrow$ For all v, all future $A[i,v]$'s will be the same

$\Rightarrow$ Can safely halt (since $A[n-1,v]$'s $=$ correct shortest-path distances)

From Bellman-Ford to Internet Routing

Note: The Bellman-Ford algorithm is intuitively "distributed".

Toward a routing protocol:
(1) Switch from source-driven to destination driven

[Just reverse all directions in the Bellman-Ford algorithm]

- Every vertex v stores shortest-path distance from v to destination t and the first hop of a shortest path

[For all relevant destinations t]

("Distance vector protocols")

Handling Asynchrony

(2) Can't assume all $A[i, v]$'s get computed before all $A[i-1, v]$'s

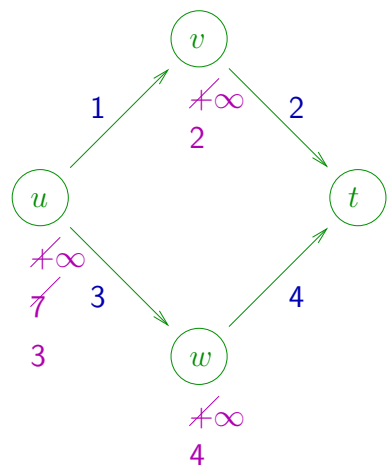
Fix: Switch from "pull-based" to "push-based": As soon as $A[i, v] < A[i-1, v]$, v notifies all of its neighbors.

Fact: Algorithm guaranteed to converge eventually. (Assuming no negative cycles)

[Reason: Updates strictly decrease sum of shortest-path estimates]

$\Rightarrow$ RIP, RIP2 Internet routing protocols very close to this algorithm [see RFC 1058]

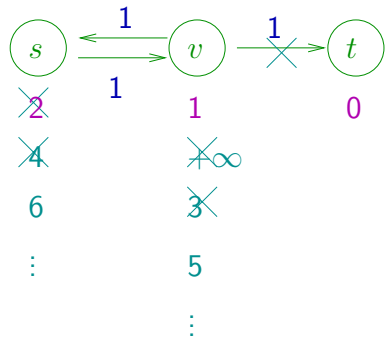
Example:



Handling Failures

Problem: Convergence guaranteed only for static networks (not true in practice).

Counting to Infinity:



Fix: Each V maintains entire shortest path to t, not just the next hop.

"Path vector protocol" "Border Gateway Protocol (BGP)"

Con: More space required.

Pro#1: More robust to failures.

Pro#2: Permits more sophisticated route selection (e.g., if you care about intermediate stops).

All-Pairs Shortest Paths (APSP)

Quiz

Question: How many invocations of a single-source shortest-path subroutine are needed to solve the all-pairs shortest path problem? [$n = \#$ of vertices]

A) 1

B) $n-1$

C) n

D) n^2

Running time (nonnegative edge costs):

$$n\cdot \text{ Dijkstra} = O(nm\log n) = \begin{array}{l} O(n^2\log n) \text{ if } m = \Theta(n) \\ O(n^3\log n) \text{ if } m = \Theta(n^2) \end{array}$$

Running time (general edge costs):

$$n\cdot \text{ Bellman-Ford} = O(n^2m) = \begin{array}{l} O(n^3) \text{ if } m = \Theta(n) \\ O(n^4) \text{ if } m = \Theta(n^2) \end{array}$$

Motivation

Floyd-Warshall algorithm: $O(n^3)$ algorithm for APSP.

- Works even with graphs with negative edge lengths.

Thus: (1) At least as good as n Bellman-Fords, better in dense graphs.

(2) In graphs with nonnegative edge costs, competitive with n Dijkstra's in dense graphs.

Important special case: Transitive closure of a binary (i.e., all-pairs reachability) relation.

Open question: Solve APSP significantly faster than $O(n^3)$ in dense graphs?

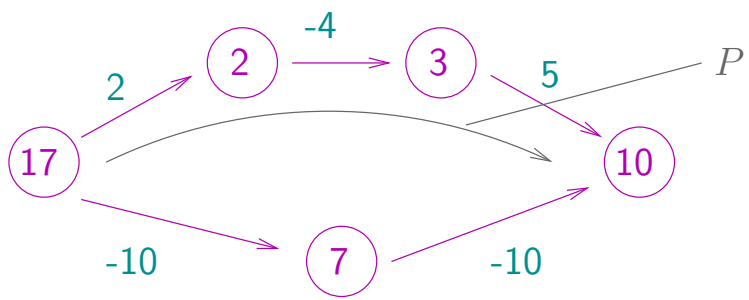
Optimal Substructure

Recall: Can be tricky to define ordering on subproblems in graph problems.

Key idea: Order the vertices $V = \{1, 2, \ldots, n\}$ arbitrarily. Let $V^{(k)} = \{1, 2, \ldots, k\}$.

Lemma: Suppose G has no negative cycle. Fix source $i \in V$, destination $j \in V$, and $k \in \{1, 2, \ldots, n\}$. Let $P =$ shortest (cycle-free) i-j path with all internal nodes in $V^{(k)}$.

Example: $[i = 17, j = 10, k = 5]$



Optimal Substructure (con'd)

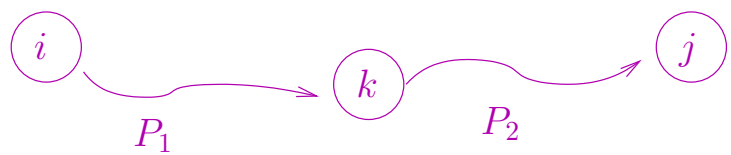
Optimal substructure lemma: Suppose G has no negative cost cycle. Let P be a shortest (cycle-free) i-j path with all internal nodes in $V^{(k)}$. Then:

Case 1: If k not internal to P, then P is a shortest (cycle-free) i-j path with all internal vertices in $V^{(k-1)}$.

Case 2: If k is internal to P, then:

P_1 = shortest (cycle-free) i-k path with all internal nodes in $V^{(k-1)}$ and

P_2 = shortest (cycle-free) k-j path with all internal nodes in $V^{(k-1)}$



Proof: Similar to Bellman-Ford opt substructure (you check!)

Problem Definition

Input: Directed graph $G = (V, E)$ with edge costs c_e for each edge $e \in E$, [No distinguished source vertex.]

Goal: Either

(A) Compute the length of a shortest $u \rightarrow v$ path for all pairs of vertices $u, v \in V$

OR

(B) Correctly report that G contains a negative cycle.

Quiz

Setup: Let A = 3-D array (indexed by i, j, k).

Intent: $A[i, j, k]$ = length of a shortest i-j path with all internal nodes in $\{1, 2, \ldots, k\}$ (or $+\infty$ if no such paths)

Question: What is $A[i, j, 0]$ if

(1) $i = j$ (2) $(i, j) \in E$ (3) $i \neq j$ and $(i, j) \notin E$

A) 0, 0, and $+\infty$

B) 0, c_{ij}, and c_{ij}

C) 0, c_{ij}, and $+\infty$

D) $+\infty$, c_{ij}, and $+\infty$

The Floyd-Warshall Algorithm

Let A = 3-D array (indexed by i, j, k)

Base cases: For all $i, j \in V$:

$$A[i,j,0] = \left\{ \begin{array}{l} 0 \text{ if } i = j \\ c_{ij} \text{ if } (i,j) \in E \\ +\infty \text{ if } i \neq j \text{ and } (i,j) \notin E \end{array} \right\}$$

For $k = 1$ to n

  For $i = 1$ to n

    For $j = 1$ to n

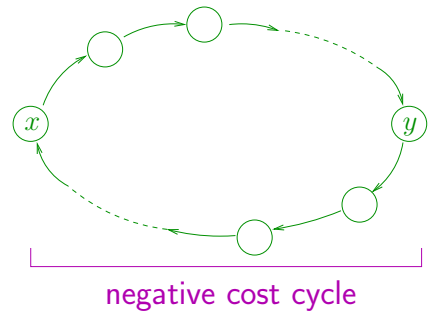
$$A[i,j,k] = \min \left\{ \begin{array}{ll} A[i,j,k-1] & \text{Case 1} \\ A[i,k,k-1] + A[k,j,k-1] & \text{Case 2} \end{array} \right\}$$

Correctness: From optimal substructure + induction, as usual.

Running time: $O(1)$ per subproblem, $O(n^3)$ overall.

Odds and Ends

Question #1: What if input graph G has a negative cycle?



Answer: Will have $A[i, i, n] < 0$ for at least one $i \in V$ at end of algorithm.

Question #2: How to reconstruct a shortest i-j path?

Answer: In addition to A, have Floyd-Warshall compute $B[i, j] =$ max label of an internal node on a shortest i-j path for all $i, j \in V$.

[Reset $B[i, j] = k$ if 2nd case of recurrence used to compute $A[i, j, k]$]

$\Rightarrow$ Can use the $B[i, j]$'s to recursively reconstruct shortest paths!