4.4 **Shortest Paths**

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm

https://algs4.cs.princeton.edu
Shortest paths in an edge-weighted digraph

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

![Edge-weighted digraph](image)

edge-weighted digraph

\[
\begin{align*}
4 \rightarrow 5 & \quad 0.35 \\
5 \rightarrow 4 & \quad 0.35 \\
4 \rightarrow 7 & \quad 0.37 \\
5 \rightarrow 7 & \quad 0.28 \\
7 \rightarrow 5 & \quad 0.28 \\
5 \rightarrow 1 & \quad 0.32 \\
0 \rightarrow 4 & \quad 0.38 \\
0 \rightarrow 2 & \quad 0.26 \\
7 \rightarrow 3 & \quad 0.39 \\
1 \rightarrow 3 & \quad 0.29 \\
2 \rightarrow 7 & \quad 0.34 \\
6 \rightarrow 2 & \quad 0.40 \\
3 \rightarrow 6 & \quad 0.52 \\
6 \rightarrow 0 & \quad 0.58 \\
6 \rightarrow 4 & \quad 0.93
\end{align*}
\]

shortest path from 0 to 6

\[
0 \rightarrow 2 \rightarrow 7 \rightarrow 3 \rightarrow 6
\]

length of path = 1.51

\[
(0.26 + 0.34 + 0.39 + 0.52)
\]
Google maps
Shortest path applications

- PERT/CPM.
- Map routing.
- **Seam carving.** see Assignment 6
- Texture mapping.
- Robot navigation.
- Typesetting in \text{T\LaTeX}.
- Currency exchange.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Shortest path variants

Which vertices?
- Single source: from one vertex \( s \) to every vertex.
- Single destination: from every vertex to one vertex \( t \).
- Source–destination: from one vertex \( s \) to another vertex \( t \).
- All pairs: between all pairs of vertices.

Restrictions on edge weights?
- Non-negative weights.
  - Euclidean weights.
  - Arbitrary weights.

Directed cycles?
- Prohibit.
- Allow.

Simplifying assumption. Each vertex is reachable from \( s \).
Shortest paths: quiz 1

Which variant in car GPS? Hint: drivers sometimes make wrong turns.

A. Single source: from one vertex $s$ to every vertex.

B. Single destination: from every vertex to one vertex $t$.

C. Source–destination: from one vertex $s$ to another vertex $t$.

D. All pairs: between all pairs of vertices.
4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm
Data structures for single-source shortest paths

**Goal.** Find a shortest path from \( s \) to every vertex.

**Observation 1.** There exists a shortest path from \( s \) to \( v \) that is simple.

**Observation 2.** A shortest-paths tree (SPT) solution exists. Why?

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

- \( \text{distTo}[v] \) is length of a shortest path from \( s \) to \( v \).
- \( \text{edgeTo}[v] \) is last edge on a shortest path from \( s \) to \( v \).

\[
\begin{array}{ccc}
\text{distTo[]} & \text{edgeTo[]} \\
0 & \text{null} & \\
1 & 1.05 & 5->1 0.32 \\
2 & 0.26 & 0->2 0.26 \\
3 & 0.97 & 7->3 0.37 \\
4 & 0.38 & 0->4 0.38 \\
5 & 0.73 & 4->5 0.35 \\
6 & 1.49 & 3->6 0.52 \\
7 & 0.60 & 2->7 0.34 \\
\end{array}
\]
Edge relaxation

Relax edge \( e = v \rightarrow w \).

- \( \text{distTo}[v] \) is length of shortest known path from \( s \) to \( v \).
- \( \text{distTo}[w] \) is length of shortest known path from \( s \) to \( w \).
- \( \text{edgeTo}[w] \) is last edge on shortest known path from \( s \) to \( w \).
- If \( e = v \rightarrow w \) yields shorter path from \( s \) to \( w \), via \( v \), update \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \).

\[ \text{relax edge } e = v \rightarrow w \]
What are the values of $\text{distTo}[v]$ and $\text{distTo}[w]$ after relaxing $e = v \rightarrow w$?

A. 10.0 and 15.0
B. 10.0 and 17.0
C. 12.0 and 15.0
D. 12.0 and 17.0
Generic algorithm (to compute a SPT from s)

For each vertex v: \( \text{distTo}[v] = \infty \).
For each vertex v: \( \text{edgeTo}[v] = \text{null} \).
\( \text{distTo}[s] = 0 \).
Repeat until \( \text{distTo}[v] \) values converge:
  - Relax any edge.

Key properties. Throughout the generic algorithm,
  • \( \text{distTo}[v] \) is either infinity or the length of a (simple) path from \( s \) to \( v \).
  • \( \text{distTo}[v] \) does not increase.
Generic algorithm (to compute a SPT from s)

For each vertex v: \( \text{distTo}[v] = \infty. \)
For each vertex v: \( \text{edgeTo}[v] = \text{null}. \)
\( \text{distTo}[s] = 0. \)
Repeat until \( \text{distTo}[v] \) values converge:
  - Relax any edge.

Efficient implementations.

- Which edge to relax next?
- How many edge relaxations needed to guarantee convergence?

Ex 1. Bellman–Ford algorithm.
Ex 2. Dijkstra’s algorithm.
Ex 3. Topological sort algorithm.
4.4 Shortest Paths

- properties
- APIs
- Bellman-Ford algorithm
- Dijkstra’s algorithm
Weighted directed edge API

```java
public class DirectedEdge {
    DirectedEdge(int v, int w, double weight) {
        weighted edge v→w
    }

    int from() {
        vertex v
    }

    int to() {
        vertex w
    }

    double weight() {
        weight of this edge
    }
}
```

Relaxing an edge \( e = v \rightarrow w \).

```java
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;
    }
}
```
Weighted directed edge: implementation in Java

**API.** Similar to Edge for undirected graphs, but a bit simpler.

```java
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 double weight() {
        return weight;
    }
}
```
### Edge-weighted digraph API

**API.** Same as `EdgeWeightedGraph` except with `DirectedEdge` objects.

```java
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 incident from v
    int V() // number of vertices

    // ...
}
```
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

8
15
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

Bag objects

reference to a DirectedEdge object

adj

0
1
2
3
4
5
6
7
Edge-weighted digraph: adjacency-lists implementation in Java

**Implementation.** Almost identical to EdgeWeightedGraph.

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

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

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

    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }
}
```
**Goal.** Find the shortest path from \( s \) to every other vertex.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`public class SP</td>
<td></td>
</tr>
<tr>
<td><code>SP(EdgeWeightedDigraph G, int s)</code></td>
<td>shortest paths from ( s ) in digraph ( G )</td>
</tr>
<tr>
<td><code>double distTo(int v)</code></td>
<td>length of shortest path from ( s ) to ( v )</td>
</tr>
<tr>
<td><code>Iterable&lt;DirectedEdge&gt; pathTo(int v)</code></td>
<td>shortest path from ( s ) to ( v )</td>
</tr>
<tr>
<td><code>boolean hasPathTo(int v)</code></td>
<td>is there a path from ( s ) to ( v )?</td>
</tr>
</tbody>
</table>
4.4 Shortest Paths

- properties
- APIs
  - Bellman–Ford algorithm
  - Dijkstra’s algorithm
Bellman–Ford algorithm

For each vertex $v$: $\text{distTo}[v] = \infty$.
For each vertex $v$: $\text{edgeTo}[v] = \text{null}$.
$\text{distTo}[s] = 0$.
Repeat $V-1$ times:
  - Relax each edge.

```
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;
    }
}
```

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

**Running time.** Algorithm takes $\Theta(EV)$ time and uses $\Theta(V)$ extra space.
Bellman–Ford algorithm demo

Repeat $V - 1$ times: relax all $E$ edges.
Bellman–Ford algorithm demo

Repeat $V - 1$ times: relax all $E$ edges.

shortest-paths tree from vertex $s$
Bellman–Ford algorithm: correctness proof

**Proposition.** Let \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k = v \) be any path from \( s \) to \( v \) containing \( k \) edges. Then, after pass \( k \), \( \text{distTo}[v_k] \leq \text{weight}(e_1) + \text{weight}(e_2) + \ldots + \text{weight}(e_k) \).

\[
\begin{array}{c}
\text{\( s \)} \\
\downarrow \\
 v_0 \\
\downarrow \\
 e_1 \\
\downarrow \\
 v_1 \\
\downarrow \\
 e_2 \\
\downarrow \\
 v_2 \\
\vdots \\
\downarrow \\
 e_k \\
\downarrow \\
 v_k \\
\downarrow \\
 v
\end{array}
\]

**Pf.** [ by induction on number of passes \( i \) ]

- **Base case:** initially, \( \text{distTo}[v_0] \leq 0 \).
- **Inductive hypothesis:** after pass \( i \), \( \text{distTo}[v_i] \leq \text{weight}(e_1) + \text{weight}(e_2) + \ldots + \text{weight}(e_i) \).
- **This inequality continues to hold because** \( \text{distTo}[v_i] \) **cannot increase.**
- **Immediately after relaxing edge** \( e_{i+1} \) **in pass** \( i+1 \), **we have**
  \[
  \text{distTo}[v_{i+1}] \leq \text{distTo}[v_i] + \text{weight}(e_{i+1}) \quad \text{\( \leq \text{weight}(e_1) + \text{weight}(e_2) + \ldots + \text{weight}(e_i) + \text{weight}(e_{i+1}). \) inductive hypothesis}
  \]
- **This inequality continues to hold because** \( \text{distTo}[v_{i+1}] \) **does not increase. \( \blacksquare \)**
Bellman–Ford algorithm: correctness proof

**Proposition.** Let \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k = v \) be any path from \( s \) to \( v \) containing \( k \) edges. Then, after pass \( k \), \( \text{distTo}[v_k] \leq \text{weight}(e_1) + \text{weight}(e_2) + \cdots + \text{weight}(e_k) \).

**Corollary.** Bellman–Ford computes shortest path distances.

**Pf.** [apply Proposition to a shortest path from \( s \) to \( v \)]

- There exists a shortest path \( P^* \) from \( s \) to \( v \) with \( k \leq V - 1 \) edges.
- From Proposition, \( \text{distTo}[v] \leq \text{length}(P^*) \).
- Since \( \text{distTo}[v] \) is the length of some path from \( s \) to \( v \), \( \text{distTo}[v] = \text{length}(P^*) \).
Bellman–Ford algorithm: practical improvement

**Observation.** If $\text{distTo}[v]$ does not change during pass $i$, not necessary to relax any edges incident from $v$ in pass $i + 1$.

**Queue-based implementation of Bellman–Ford.**

- Perform *vertex* relaxations.  
  
  ![relax all edges incident from v](image)

- Maintain *queue* of vertices whose $\text{distTo}[]$ values changed since it was last relaxed.

  ![must ensure each vertex is on queue at most once](image)

**Impact.**

- In the worst case, the running time is still $\Theta(EV)$.
- But much faster in practice on typical inputs.
What is the running time of the queue-based version of Bellman–Ford in the best case, as a function of $E$ and $V$?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(VE)$
Problem. Given a digraph $G$ with positive edge weights and vertex $s$, find a longest simple path from $s$ to every other vertex.

Goal. Design algorithm that takes $\Theta(EV)$ time in the worst case.

Longest simple path from 0 to 4: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
**Bellman–Ford algorithm: negative weights**

**Remark.** The Bellman–Ford algorithm works even if some weights are negative, provided there are no negative cycles.

**Negative cycle.** A directed cycle whose length is negative.

![Diagram of a directed cycle with negative weights](image)

**length of negative cycle** = 1 + 2 + 3 + -8 = -2

**Negative cycles and shortest paths.** Length of path can be made arbitrarily negative by using negative cycle.

0 → 1 → 2 → 3 → 4 → ... → 1 → 2 → 3 → 4 → 1 → 2 → 5
4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm
“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

-- Edsger Dijkstra
Dijkstra’s algorithm

For each vertex v: \( \text{distTo}[v] = \infty \).
For each vertex v: \( \text{edgeTo}[v] = \text{null} \).

\( T = \emptyset \).
\( \text{distTo}[s] = 0 \).
Repeat until all vertices are marked:
- Select unmarked vertex v with the smallest \( \text{distTo}[v] \) value.
- Mark v.
- Relax each edge incident from v.

Key difference with Bellman–Ford. Each edge gets relaxed exactly once!
Dijkstra’s algorithm demo

Repeat until all vertices are marked:

- Select unmarked vertex $v$ with the smallest $\text{distTo}[]$ value.
- Mark $v$ and relax all edges incident from $v$.

an edge-weighted digraph

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→1</td>
<td>5.0</td>
</tr>
<tr>
<td>0→4</td>
<td>9.0</td>
</tr>
<tr>
<td>0→7</td>
<td>8.0</td>
</tr>
<tr>
<td>1→2</td>
<td>12.0</td>
</tr>
<tr>
<td>1→3</td>
<td>15.0</td>
</tr>
<tr>
<td>1→7</td>
<td>4.0</td>
</tr>
<tr>
<td>2→3</td>
<td>3.0</td>
</tr>
<tr>
<td>2→6</td>
<td>11.0</td>
</tr>
<tr>
<td>3→6</td>
<td>9.0</td>
</tr>
<tr>
<td>4→5</td>
<td>4.0</td>
</tr>
<tr>
<td>4→6</td>
<td>20.0</td>
</tr>
<tr>
<td>4→7</td>
<td>5.0</td>
</tr>
<tr>
<td>5→2</td>
<td>1.0</td>
</tr>
<tr>
<td>5→6</td>
<td>13.0</td>
</tr>
<tr>
<td>7→5</td>
<td>6.0</td>
</tr>
<tr>
<td>7→2</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm demo

Repeat until all vertices are marked:
• Select unmarked vertex $v$ with the smallest $\text{distTo}[]$ value.
• Mark $v$ and relax all edges incident from $v$.

<table>
<thead>
<tr>
<th>v</th>
<th>$\text{distTo}[]$</th>
<th>$\text{edgeTo}[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

shortest-paths tree from vertex $s$
Dijkstra’s algorithm: correctness proof

**Invariant.** For each marked vertex \( v \): \( \text{distTo}[v] = d^*(v) \).

**Pf.** [by induction on number of marked vertices]

- Let \( v \) be next vertex marked.
- Let \( P \) be the path from \( s \) to \( v \) of length \( \text{distTo}[v] \).
- Consider any other path \( P' \) from \( s \) to \( v \).
- Let \( x \rightarrow y \) be first edge in \( P' \) with \( x \) marked and \( y \) unmarked.
- \( P' \) is no shorter than \( P \):

\[
\text{length}(P) = \text{distTo}[v] \\
\leq \text{distTo}[y] \\
\leq \text{distTo}[x] + \text{weight}(x, y) \\
= d^*(x) + \text{weight}(x, y) \\
\leq \text{length}(P')
\]

Dijkstra chose \( v \) instead of \( y \)

relax vertex \( x \)

induction

weights are non-negative

by construction

length of shortest path from \( s \) to \( v \)
Dijkstra’s algorithm: correctness proof

**Invariant.** For each marked vertex $v$: $\text{distTo}[v] = d^*(v)$.

**Corollary 1.** Dijkstra’s algorithm computes shortest path distances.

**Corollary 2.** Dijkstra’s algorithm relaxes vertices in increasing order of distance from $s$. 

length of shortest path from $s$ to $v$

generalizes level-order traversal and breadth-first search
Dijkstra’s algorithm: Java implementation

```java
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);
            }
    }
}
```

- **PQ** that supports decreasing the key (stay tuned)
- **PQ** contains the unmarked vertices with finite `distTo[]` values
- Relax vertices in order of distance from `s`
Dijkstra’s algorithm: Java implementation

When relaxing an edge, also update PQ:

- Found first path from $s$ to $w$: add $w$ to PQ.
- Found better path from $s$ to $w$: decrease key of $w$ in PQ.

```java
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.insert(w, distTo[w]);
        else pq.decreaseKey(w, distTo[w]);
    }
}
```

Q. How to implement DECREASE-KEY operation in a priority queue?
Indexed priority queue (Section 2.4)

Associate an index between 0 and $n - 1$ with each key in a priority queue.
- Insert a key associated with a given index.
- Delete a minimum key and return associated index.
- Decrease the key associated with a given index.

for Dijkstra's algorithm:
\[ n = V, \]
\[ \text{index} = \text{vertex}, \]
\[ \text{key} = \text{distance from } s \]

```java
public class IndexMinPQ<Key extends Comparable<Key>> {
    IndexMinPQ(int n)
    // create PQ with indices 0, 1, ..., n - 1
    void insert(int i, Key key)
    // associate key with index i
    int delMin()
    // remove min key and return associated index
    void decreaseKey(int i, Key key)
    // decrease the key associated with index i
    boolean isEmpty()
    // is the priority queue empty?
    :
    :
}
```
**Goal.** Implement **DECREASE-KEY** operation in a binary heap.
DECREASE-KEY IN A BINARY HEAP

**Goal.** Implement DECREASE-KEY operation in a binary heap.

**Solution.**
- Find vertex in heap. How?
- Change priority of vertex and call `swim()` to restore heap invariant.

**Extra data structure.** Maintain an inverse array `qp[]` that maps from the vertex to the binary heap node index.

```
0 1 2 3 4 5 6 7 8
pq[]  -  v3  v5  v7  v2  v0  v4  v6  v1
qp[]  5  8  4  1  6  2  4  3  -
keys[] 1.0 2.0 3.0 0.0 6.0 8.0 4.0 2.0 -
```

Vertex 2 has priority 3.0 and is at heap index 4.
Dijkstra’s algorithm: which priority queue?

Number of PQ operations: $V \text{ INSERT}, V \text{ DELETE-MIN}, \leq E \text{ DECREASE-KEY}$.

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>INSERT</th>
<th>DELETE-MIN</th>
<th>DECREASE-KEY</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>$V$</td>
<td>1</td>
<td>$V^2$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$E \log V$</td>
</tr>
<tr>
<td>d-way heap</td>
<td>$\log_d V$</td>
<td>$d \log_d V$</td>
<td>$\log_d V$</td>
<td>$E \log_{E/V} V$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$1^\dagger$</td>
<td>$\log V^\dagger$</td>
<td>$1^\dagger$</td>
<td>$E + V \log V$</td>
</tr>
</tbody>
</table>

’amortized

**Bottom line.**

- Array implementation optimal for complete digraphs.
- Binary heap much faster for sparse digraphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
Observation. Prim and Dijkstra are essentially the same algorithm.

- Prim: Choose next vertex that is closest to any vertex in the tree (via an undirected edge).
- Dijkstra: Choose next vertex that is closest to the source vertex (via a directed path).
Algorithms for shortest paths

Variations on a theme: vertex relaxations.

- **Bellman–Ford**: relax all vertices; repeat $V - 1$ times.
- **Dijkstra**: relax vertices in order of distance from $s$.
- **Topological sort**: relax vertices in topological order.

### Table: Algorithms for Shortest Paths

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case running time</th>
<th>negative weights †</th>
<th>directed cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman–Ford</td>
<td>$EV$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$E \log V$</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>topological sort</td>
<td>$E$</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

† no negative cycles

see Section 4.4 and next lecture
Which shortest paths algorithm to use?

Select algorithm based on properties of edge-weighted digraph.

- Negative weights (but no “negative cycles”): Bellman–Ford.
- Non-negative weights: Dijkstra.
- DAG: topological sort.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case running time</th>
<th>negative weights †</th>
<th>directed cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman–Ford</td>
<td>$E V$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$E \log V$</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>topological sort</td>
<td>$E$</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

† no negative cycles
4.4 **Shortest Paths**

- properties
- APIs
- Bellman–Ford algorithm
- Dijkstra’s algorithm
- **seam carving**
Content-aware resizing

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

https://www.youtube.com/watch?v=vIFCV2spKtg
Content-aware resizing

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

*In the wild.* Photoshop, Imagemagick, GIMP, ...
To find vertical seam:

- Grid graph: 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.
To find vertical seam:

- Grid graph: 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).
**Shortest Path Variants in a Digraph**

Q1. How to model vertex weights (along with edge weights)?

Q2. How to model multiple sources and destinations?