4.4 **Shortest Paths**

- APIs
- properties
- *Bellman–Ford algorithm*
- *Dijkstra’s algorithm*
- *topological sort algorithm*
- negative weights
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 → 7 → 3 → 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 7
- Texture mapping.
- Robot navigation.
- Typesetting in TeX.
- 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 other vertex.
- Single sink: from every vertex to one vertex $t$.
- Source–sink: from one vertex $s$ to another $t$.
- All pairs: between all pairs of vertices.

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

Cycles?
- No directed cycles.
- No “negative cycles.”

Simplifying assumption. Each vertex is reachable from $s$. 
Shortest paths: quiz 1

Which variant in car GPS?

A. Single source: from one vertex \( s \) to every other vertex.

B. Single sink: from every vertex to one vertex \( t \).

C. Source–sink: from one vertex \( s \) to another \( t \).

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

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

    String toString()
    // string representation
}
```

**Idiom for processing an edge e:**

```java
int v = e.from(), w = e.to();
```
Weighted directed edge: implementation in Java

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;
    }
}
```

from() and to() replace either() and other()
Edge-weighted digraph API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class EdgeWeightedDigraph</code></td>
<td></td>
</tr>
<tr>
<td><code>EdgeWeightedDigraph(int V)</code></td>
<td>edge-weighted digraph with V vertices</td>
</tr>
<tr>
<td><code>EdgeWeightedDigraph(In in)</code></td>
<td>edge-weighted digraph from input stream</td>
</tr>
<tr>
<td><code>void addEdge(DirectedEdge e)</code></td>
<td>add weighted directed edge e</td>
</tr>
<tr>
<td><code>Iterable&lt;DirectedEdge&gt; adj(int v)</code></td>
<td>edges adjacent from v</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>number of edges</td>
</tr>
<tr>
<td><code>Iterable&lt;DirectedEdge&gt; edges()</code></td>
<td>all edges</td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>string representation</td>
</tr>
</tbody>
</table>

**Conventions.** Allow self-loops and parallel edges.
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V

E

adj

0  2  0.26  0  4  0.38

1  3  0.29

2  7  0.34

3  6  0.52

4  7  0.37  4  5  0.35

5  1  0.32  5  4  0.35  5  7  0.28

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
Edge-weighted digraph: adjacency-lists implementation in Java

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<Edge>[][]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<DirectedEdge>();
    }

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

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

add edge \( e = v \to w \) to only \( v \)'s adjacency list
Single-source shortest paths API

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

```java
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?
```
4.4 Shortest Paths

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

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

```
shortest-paths tree from 0
```

```
<table>
<thead>
<tr>
<th></th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>1.05</td>
<td>5-&gt;1 0.32</td>
</tr>
<tr>
<td>2</td>
<td>0.26</td>
<td>0-&gt;2 0.26</td>
</tr>
<tr>
<td>3</td>
<td>0.97</td>
<td>7-&gt;3 0.37</td>
</tr>
<tr>
<td>4</td>
<td>0.38</td>
<td>0-&gt;4 0.38</td>
</tr>
<tr>
<td>5</td>
<td>0.73</td>
<td>4-&gt;5 0.35</td>
</tr>
<tr>
<td>6</td>
<td>1.49</td>
<td>3-&gt;6 0.52</td>
</tr>
<tr>
<td>7</td>
<td>0.60</td>
<td>2-&gt;7 0.34</td>
</tr>
</tbody>
</table>
```
Data structures for single-source shortest paths

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

```java
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 $e = v \rightarrow w$.

- $\text{distTo}[v]$ 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 to $w$, update $\text{distTo}[w]$ and $\text{edgeTo}[w]$.
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 to $w$, update $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

```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;
    }
}
```
What are the values of \( \text{distTo}[v] \) and \( \text{distTo}[w] \) after relaxing \( v \rightarrow w \)?

A. 10.0 and 15.0
B. 10.0 and 17.0
C. 12.0 and 15.0
D. 12.0 and 17.0
Framework for shortest-paths algorithm

**Generic algorithm (to compute a 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.

**Key properties.**

- `distTo[v]` is the length of a simple path from `s` to `v`.
- `distTo[v]` cannot increase.
Framework for shortest-paths algorithm

**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 done:
- Relax any edge.

**Efficient implementations.**

- How to choose which edge to relax next?
- How many edge relaxations needed?

**Ex 1.** Dijkstra’s algorithm (non-negative weights).
**Ex 2.** Topological sort algorithm (no directed cycles).
**Ex 3.** Bellman–Ford algorithm (no negative cycles).
4.4 **Shortest Paths**

- APIs
- properties
- Bellman–Ford algorithm
- Dijkstra’s algorithm
- topological sort 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.

```java
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 demo

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

an edge-weighted digraph

<table>
<thead>
<tr>
<th>Edges</th>
<th>Weights</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>
Bellman–Ford algorithm demo

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

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

passes
4
7
10
13
SPT
**Bellman–Ford algorithm: correctness proof**

**Proposition.** Let \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = v \) be a shortest path from \( s \) to \( v \).

Then, after pass \( i \), \( \text{distTo}[v_i] = d^*(v_i) \).

**Pf.** [ by induction on \( i \) ]
- Suppose \( \text{distTo}[v_i] = d^*(v_i) \) after pass \( i \).
- 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 \( v_i \rightarrow v_{i+1} \) in pass \( i+1 \), we have
  \[
  \text{distTo}[v_{i+1}] \leq \text{distTo}[v_i] + \text{weight}(v_i, v_{i+1})
  = d^*(v_i) + \text{weight}(v_i, v_{i+1})
  = d^*(v_{i+1}).
  \]
- 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.

\[ \Rightarrow \leq V - 1 \text{ passes}. \]
Bellman–Ford algorithm: practical improvement

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

FIFO implementation. Maintain queue of vertices whose \( \text{distTo[]} \) values needs updating.

Impact.
- In the worst case, the running time is still proportional to \( E \times V \).
- But much faster in practice.
4.4 Shortest Paths

- APIs
- properties
- Bellman–Ford algorithm
- Dijkstra’s algorithm
- topological sort algorithm
“Do only what only you can do.”

“The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.”

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”

“APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.”

\begin{equation}
\Phi' \sigma', \in N \rho S \xleftarrow{\ {}^0 \{} \leftarrow (3 = T \vee N \wedge 2 = T \rightarrow + (\forall \Phi' \in M), (\forall \Theta' \subseteq M), (\forall \tau, \Phi, \tau) \Phi'(\tau, \tau \leftarrow \tau \cdot 1 \cdot 1) \Theta' \subseteq M'
\end{equation}
Dijkstra’s algorithm demo

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

![Graph Image]

an edge-weighted digraph
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 adjacent from that vertex.

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>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 visualization
Dijkstra’s algorithm visualization
**Dijkstra’s algorithm: correctness proof**

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

**Pf.** [by induction on $|T|$]

- Let $w$ be next vertex added to $T$.
- Let $P$ be the $s \rightarrow w$ path of length $\text{distTo}[w]$.
- Consider any other $s \rightarrow w$ path $P'$.
- Let $x \rightarrow y$ be first edge in $P'$ that leaves $T$.
- $P'$ is no shorter than $P$:

\[
\text{length}(P) = \text{distTo}[w] \leq \text{distTo}[y] \leq \text{distTo}[x] + \text{weight}(x, y) \leq \text{length}(P').
\]
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);
        }
    }
}
```
Dijkstra’s algorithm: Java implementation

```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.decreaseKey(w, distTo[w]);
        else pq.insert (w, distTo[w]);
    }
}
```
What is the order of growth of the running time of Dijkstra’s algorithm when using a binary heap for the priority queue?

A. \( V + E \)
B. \( V \log E \)
C. \( E \log V \)
D. \( E \log E \)
Dijkstra’s algorithm: which priority queue?

Depends on PQ implementation: \( V \text{ INSERT}, V \text{ DELETE-MIN}, \leq E \text{ DECREASE-KEY}. \)

<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>

\( ^\dagger \) amortized

Bottom line.
- Array implementation optimal for complete graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
Priority-first search

Dijkstra’s algorithm seem familiar?

- Prim’s algorithm is essentially the same algorithm.
- Both in same family of algorithms.

Main distinction: rule used to choose next vertex for the tree.

- Prim: Closest vertex to the tree (via an undirected edge).
- Dijkstra: Closest vertex to the source (via a directed path).

Note: DFS and BFS are also in same family.
**Goal.** Given a digraph $G$, let $p(e)$ be the probability that edge $e$ succeeds. Find a path from $s$ to $t$ that maximizes the probability of success (assuming edge failures are independent).

![Diagram with probabilities]

**Probability Calculation:**

Probability = $0.5 \times 0.9 \times 0.9 \times 0.8 = 0.324$
4.4 **Shortest Paths**

- APIs
- properties
- Bellman–Ford algorithm
- Dijkstra’s algorithm
- topological sort algorithm
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!
Topological sort algorithm demo

- Consider vertices in topological order.
- Relax all edges adjacent 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
Topological sort algorithm demo

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

shortest-paths tree from vertex s

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>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>
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);
    }
}
What is the order of the running time of the topological sort algorithm for computing a SPT in an edge-weighted DAG?

A. \( V \)

B. \( E \)

C. \( V + E \)

D. \( V \log E \)
Topological sort algorithm: correctness proof

**Invariant.** For each vertex \( v \) in \( T \), \( \text{distTo}[v] = d^*(v) \).

**Pf.** [ by induction on \( |T| \) ]

- Let \( w \) be next vertex (in topological order) added to \( T \).
- Let \( P \) be the \( s \rightarrow w \) path of length \( \text{distTo}[w] \).
- Consider any other \( s \rightarrow w \) path \( P' \).
- \( P' \) must be a path to a vertex in \( T \) plus one extra edge, say \( x \rightarrow w \). Why?
- \( P' \) is no shorter than \( P \):

\[
\text{length}(P) = \text{distTo}[w] \\
\leq \text{distTo}[x] + \text{weight}(x, w) \\
= d^*(x) + \text{weight}(x, w) \\
\leq \text{length}(P') \ \blacksquare
\]
Content-aware resizing

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

[Image: Content-aware resizing tool]

http://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, ...
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.
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).
**SHORTEST PATH VARIANTS IN A DIGRAPH**

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

**Q2.** How to model multiple sources and sinks?

[Diagram of a directed graph with labeled vertices and edges]
**Challenge.** Given an edge-weighted DAG, find the longest path from $s$ to every other vertex.

**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  

**longest path from 5 to 0**

$(0.32 + 0.29 + 0.52 + 0.93 + 0.38 = 2.44)$
Algorithm 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>
<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