4.4 **Shortest Paths**

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

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

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

length of path = 1.51

0 → 2 → 7 → 3 → 6

(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 $\text{TEX}$.
- 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 $s$. 

we assume this in today’s lecture (except as noted)
Shortest paths: quiz 1

Which variant in car GPS?

A. Single source: from one vertex \( s \) to every other vertex.
B. Single destination: from every vertex to one vertex \( t \).
C. Source–destination: from one vertex \( s \) to another \( t \).
D. All pairs: between all pairs of vertices.
4.4 **Shortest Paths**

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

https://algs4.cs.princeton.edu
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 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 \( v \).
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 $\text{edgeTo}[w]$. 

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
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.

Key properties.

- $\text{distTo}[v]$ is the length of a simple path from $s$ to $v$.
- $\text{distTo}[v]$ does not increase.

no repeated vertices
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.

Efficient implementations.
- Which edge to relax next?
- How many edge relaxations needed?

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
  - seam carving
Weighted directed edge API

```java
public class DirectedEdge
{
    DirectedEdge(int v, int w, double weight) // weighted edge v→w
    {
        int from() // vertex v
        {
            int v = e.from(), w = e.to();
            if (distTo[w] > distTo[v] + e.weight())
            {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
            }
        }
        int to() // vertex w
        {
            int v = e.from(), w = e.to();
            if (distTo[w] > distTo[v] + e.weight())
            {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
            }
        }
        double weight() // weight of this edge
        {
            int v = e.from(), w = e.to();
            if (distTo[w] > distTo[v] + e.weight())
            {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
            }
        }
    }
}
```

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

V

E

0 2 0.26
1 3 0.29
2 7 0.34
3 6 0.52
4 7 0.37
5 1 0.32
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
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<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 = \( v \rightarrow w \) to only \( v \)'s adjacency list
# Single-source shortest paths API

**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>( \text{SP}(\text{EdgeWeightedDigraph } G, \text{ int } s) )</td>
<td>shortest paths from ( s ) in digraph ( G )</td>
</tr>
<tr>
<td>double ( \text{distTo}(\text{int } v) )</td>
<td>length of shortest path from ( s ) to ( v )</td>
</tr>
<tr>
<td>Iterable ( &lt;\text{DirectedEdge}&gt; ) ( \text{pathTo}(\text{int } v) )</td>
<td>shortest path from ( s ) to ( v )</td>
</tr>
<tr>
<td>boolean ( \text{hasPathTo}(\text{int } v) )</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
- seam carving

https://algs4.cs.princeton.edu
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);
```

**Running time.** Order of growth is \( E \times V \) in both best- and worst-case.
Bellman–Ford algorithm demo

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

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>
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 \( i \) ]

- Inductive hypothesis: after pass \( i \), \( \text{distTo}[v_i] = d^*(v_i) \).
- Since \( \text{distTo}[v_{i+1}] \) is the length of some path from \( s \) to \( v_{i+1} \), we must have \( \text{distTo}[v_{i+1}] \geq d^*(v_{i+1}) \).
- Immediately after relaxing edge \( v_i \rightarrow v_{i+1} \) 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.
\[
\Rightarrow \leq V - 1 \text{ passes suffice.}
\]
Bellman–Ford algorithm: practical improvement

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

**Queue-based implementation of Bellman–Ford.** 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 on typical inputs.
**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 with $E \times V$ running time.

![Graph diagram]

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 graph with negative cycle](image)

\[
\text{length of negative cycle} = 1 + 2 + 3 + (-8) = -2
\]

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

\[
0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \cdots \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 5
\]
4.4 Shortest Paths

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

$\Phi', \Box', \in \mathcal{N}, \rho \subseteq S \leftarrow \Box \leftarrow (3 = T) \lor M \land 2 = T \leftarrow \Rightarrow \lor (\forall \Phi' \subseteq M), (\forall \Theta' \subseteq M), (\forall, \Phi, \forall) \Phi'(\forall, \forall \leftarrow 1 \land 1) \Theta' \subseteq M'$
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 incident from that vertex.

\[
\begin{align*}
0 \rightarrow 1 & \quad 5.0 \\
0 \rightarrow 4 & \quad 9.0 \\
0 \rightarrow 7 & \quad 8.0 \\
1 \rightarrow 2 & \quad 12.0 \\
1 \rightarrow 3 & \quad 15.0 \\
1 \rightarrow 7 & \quad 4.0 \\
2 \rightarrow 3 & \quad 3.0 \\
2 \rightarrow 6 & \quad 11.0 \\
3 \rightarrow 6 & \quad 9.0 \\
4 \rightarrow 5 & \quad 4.0 \\
4 \rightarrow 6 & \quad 20.0 \\
4 \rightarrow 7 & \quad 5.0 \\
5 \rightarrow 2 & \quad 1.0 \\
5 \rightarrow 6 & \quad 13.0 \\
7 \rightarrow 5 & \quad 6.0 \\
7 \rightarrow 2 & \quad 7.0
\end{align*}
\]

an edge-weighted digraph
Dijkstra’s algorithm demo

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest \texttt{distTo[]} value).
- Add vertex to tree and relax all edges incident from that vertex.
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 \( P \):
Dijkstra’s algorithm: correctness proof

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

Corollary. Dijkstra’s algorithm computes shortest path distances.

Pf. Upon termination, $T$ contains all vertices (reachable from $s$).
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)

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]);
    }
}
```
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.

```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?
    }
    ...
}
```

for Dijkstra's algorithm:
- index = vertex
- key = distance from s
Goal. Implement `DECREASE-KEY` operation 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 array `qp[]` that maps from the vertex to the binary heap node index.

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq[]</td>
<td>-</td>
<td></td>
<td>v₃</td>
<td>v₅</td>
<td>v₇</td>
<td>v₂</td>
<td></td>
<td>v₀</td>
<td>v₄</td>
</tr>
<tr>
<td>qp[]</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>keys[]</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>0.0</td>
<td>6.0</td>
<td>8.0</td>
<td>4.0</td>
<td>2.0</td>
<td>-</td>
</tr>
</tbody>
</table>
```

vertex 2 has priority 3.0 and is at heap index 4

decrease key of vertex v₂
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 seems 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.
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
Algorithm for shortest paths

Select algorithm based on properties of edge-weighted graph.
- Negative weights (but no “negative cycles”): Bellman–Ford.
- Non-negative weights: Dijkstra.
- DAG: topological sort.

In practice. Algorithm with better worst-case running time is (usually) fastest.

<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

https://algs4.cs.princeton.edu
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=vlFCV2spKtg
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 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 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 sinks?