4.4 Shortest Paths

- APIs
- properties
- Bellman–Ford algorithm
- Dijkstra’s algorithm
- seam carving
Shortest paths in an edge-weighted digraph

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

**Exercise:** find the shortest path from 0 to 6 in the above digraph
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

**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 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 $s$. 

we assume this throughout today’s lecture (even though some algorithms can handle negative weights)
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
  - seam carving
Weighted directed edge API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class DirectedEdge()</td>
<td>weighted edge $v \rightarrow w$</td>
</tr>
<tr>
<td>int from()</td>
<td>vertex $v$</td>
</tr>
<tr>
<td>int to()</td>
<td>vertex $w$</td>
</tr>
<tr>
<td>double weight()</td>
<td>weight of this edge</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation</td>
</tr>
</tbody>
</table>

Idiom for processing an edge $e$: `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;
    }
}
```
## Edge-weighted digraph API

```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 adjacent from v

    int V()  // number of vertices

    int E()  // number of edges

    Iterable<DirectedEdge> edges()  // all edges
```

**Conventions.** Allow self-loops and parallel edges.
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

V
E

adj

Bag objects

reference to a DirectedEdge object
Almost identical to `EdgeWeightedGraph`.

```java
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<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];
    }
}
```
**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><code>public class SP</code></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

- APIs
- properties
- 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 $v$.
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 $v$.

**Computing path to specific vertex.**

```java
public Iterable<DirectedEdge> pathTo(int v)
{
    // Shortest-Paths Tree has already been computed and stored as
    // arrays distTo[] and edgeTo[]
    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 $\text{edgeTo}[w]$. 

![Diagram](https://via.placeholder.com/150)
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 $v$: $\text{distTo}[v] = \infty$.

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.
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

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

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

for (int i = 1; i < G.V(); i++)
  for (int v = 0; v < G.V(); v++)
    for (DirectedEdge e : G.adj(v))
      relax(e);
Bellman–Ford algorithm demo

Repeat $V - 1$ 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 $V - 1$ 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.

\[
\Rightarrow \quad \leq V-1 \text{ passes.}
\]
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 $i + 1$.

**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.
4.4 Shortest Paths

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

https://algs4.cs.princeton.edu
“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.”
"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

-- Edsger Dijkstra
Dijkstra’s algorithm demo

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

**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 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 \( s \to w \) path of length \( \text{distTo}[w] \).
- Consider any other \( s \to w \) path \( P' \).
- Let \( x \to y \) 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 chose \( w \) instead of \( y \)
- relax vertex \( x \)
- induction
- weights are non-negative

by construction

length of shortest \( s \to v \) path
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);
        }
    }
}
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]);
    }
}
```
update PQ
Indexed priority queue [see Section 2.4 of textbook for details]

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 indexed PQ with indices 0, 1, ..., n – 1
    }

    void insert(int i, Key key) {
        // associate key with index i
    }

    int delMin() {
        // remove a minimal key and return its associated index
    }

    void decreaseKey(int i, Key key) {
        // decrease the key associated with index i
    }

    boolean contains(int i) {
        // is i an index on the priority queue?
    }

    boolean isEmpty() {
        // is the priority queue empty?
    }

    int size() {
        // number of keys in the priority queue
    }
}
```
What is the order of growth of the running time of Dijkstra’s algorithm in the worst case when using a binary heap for the priority queue?

A. $V + E$

B. $V \log V$

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**

**Insight.** Four of our graph-search methods are the same algorithm!
- Maintain a tree of explored vertices $T$.
- Grow $T$ by exploring edges with exactly one endpoint leaving $T$.

**DFS.** Take edge from vertex which was discovered most recently.

**BFS.** Take edge from vertex which was discovered least recently.

**Prim.** Take edge of minimum weight.

**Dijkstra.** Take edge to vertex that is closest to $s$.

Each algorithm results in a tree of paths from the source node:
- DFS tree
- BFS tree
- Minimal Spanning Tree
- Shortest-Paths Tree.
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>$EV$</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
Design principle: pick algorithm based on known properties of input

Arbitrary graph (with no negative cycles)? Bellman-Ford.
Graph with no negative weights? Dijkstra.
DAG? Relax vertices in topological order.

Most specialized algorithm is usually (but not always) the 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>
4.4 **Shortest Paths**

- APIs
- properties
- 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.

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, ...
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.
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).