4.4 Shortest Paths

- APIs
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- 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->4 0.93

shortest path from 0 to 6

0.26 + 0.34 + 0.39 + 0.52 = 1.11

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Texture mapping.
- Robot navigation.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGR, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- 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?
- Nonnegative weights.
- Euclidean weights.
- Arbitrary weights.

Cycles?
- No directed cycles.
- No "negative cycles."

Simplifying assumption. Each vertex is reachable from \(s\).

Weighted directed edge API

```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 int weight()
    {
        return weight;
    }

    public String toString()
    {
        return String.valueOf(v) + "->" + String.valueOf(w); // Assuming from() and to() return strings
    }
}
```

Idiom for processing an edge \(e\): \(\text{int } v = e\text{-from()}, \text{ w } = e\text{-to();}\)

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 int weight()
    {
        return weight;
    }

    public String toString()
    {
        return String.valueOf(v) + "->" + String.valueOf(w); // Assuming from() and to() return strings
    }
}
```
**Edge-weighted digraph API**

```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<DirectedEdge>();
    }

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

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

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

**Edge-weighted digraph: adjacency-lists implementation in Java**

Same as `EdgeWeightedGraph` except replace `Graph` with `Digraph`.

```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<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        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.

```java
public class SP
{
    public double distTo(int v)
    { return shortest path length from s to v; }
    public Iterable<DirectedEdge> pathTo(int v)
    { return shortest path from s to v; }
    public boolean hasPathTo(int v)
    { is there a path from s to v? }
}
```
4.4 Shortest Paths

- APIs
- Shortest-paths properties
- Dijkstra’s algorithm
- Edge-weighted DAGs
- Negative weights

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

**Example**

- \( v \rightarrow w \) successfully relaxes
- Black edges are in \( \text{edgeTo}[] \)
**Edge relaxation**

Relax edge $e = v \to w$.

- $\text{distTo}[v]$ 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 $e = v \to w$ gives shorter path to $w$ through $v$, update both $\text{distTo}[w]$ 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;
    }
}
```

**Shortest-paths optimality conditions**

**Proposition.** Let $G$ be an edge-weighted digraph. Then $\text{distTo}[\cdot]$ are the shortest path distances from $s$ iff:

- $\text{distTo}[s] = 0$.
- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \to w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

**Pf.** $\Rightarrow$ [sufficient]

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \to w$.
- Then, $\text{distTo}[v_1] \leq \text{distTo}[v_0] + e_1.\text{weight}()$
- $\text{distTo}[v_2] \leq \text{distTo}[v_1] + e_2.\text{weight}()$
- $\text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}()$
- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:
  - $\text{distTo}[w] = \text{distTo}[v_0] \leq e_1.\text{weight}() + e_2.\text{weight}() + \ldots + e_k.\text{weight}()$
- Thus, $\text{distTo}[w]$ is the weight of shortest path to $w$. $\blacksquare$

**Generic shortest-paths algorithm**

**General algorithm (to compute a SPT from $s$)**

- Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ 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.**

- $\text{distTo}[v]$ is always the length of a simple path from $s$ to $v$.
- Each successful relaxation decreases $\text{distTo}[v]$ for some $v$.
- $\text{distTo}[v]$ can decrease at most a finite number of times. $\blacksquare$
Generic shortest-paths algorithm

Generic algorithm (to compute a 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).

Shortest paths: quiz 1

Let \( e = v \rightarrow w \) be an edge with weight 17.0. Suppose that \( \text{distTo}[v] = \infty \) and \( \text{distTo}[w] = 15.0 \). Which is the value of \( \text{distTo}[w] \) after calling \( \text{relax}(e) \)?

A. The program will throw a \text{java.lang.RuntimeException}.
B. 15.0
C. 17.0
D. + ∞
E. I don’t know.

Edsger W. Dijkstra: select quotes

“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.”

Edsger W. Dijkstra
Turing award 1972

http://catpad.net/michael/apl
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.

Dijkstra’s algorithm visualization

shortest-paths tree from vertex \( s \)

\[
\begin{array}{c|ccc}
\text{v} & \text{distTo[\cdot]} & \text{edgeTo[\cdot]} \\
\hline
0 & 0.0 & - \\
1 & 5.0 & 0 \rightarrow 1 \\
2 & 14.0 & 5 \rightarrow 2 \\
3 & 17.0 & 2 \rightarrow 3 \\
4 & 9.0 & 0 \rightarrow 4 \\
5 & 13.0 & 4 \rightarrow 5 \\
6 & 25.0 & 2 \rightarrow 6 \\
7 & 8.0 & 0 \rightarrow 7 \\
\end{array}
\]
Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.
- Each edge $e = v \rightarrow w$ is relaxed exactly once (when vertex $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e\text{.weight}()$.
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$ cannot increase
  - $\text{distTo}[v]$ will not change

Thus, upon termination, shortest-paths optimality conditions hold. $$\blacksquare$$
Computing a spanning tree in a graph

Dijkstra's algorithm seem familiar?
- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a spanning tree.

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 this family of algorithms.

4.4 Shortest Paths
- APIs
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights
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 adjacent from that vertex.

Shortest paths: quiz 3

What is the order of growth of the running time of the topological sort algorithm for computing shortest paths in an edge-weighted DAG?

A. \( V \)
B. \( E \)
C. \( V + E \)
D. \( V \log E \)
E. I don't know.
Shortest paths in edge-weighted DAGs

**Proposition.** Topological sort algorithm computes the SPT in any edge-weighted DAG.

**Pf.**
- Each edge $e = v \rightarrow w$ is relaxed exactly once (when vertex $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e\cdot \text{weight}()$.
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$ cannot increase
  - $\text{distTo}[v]$ will not change
    - $\text{distTo}[v]$ is relaxed exactly once when vertex $v$ is relaxed
    - $\text{distTo}[w]$ values are monotonically decreasing because of topological order, no vertex pointing to $v$ will be relaxed after $v$ is relaxed

Thus, upon termination, shortest-paths optimality conditions hold. 

---

Content-aware resizing

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

---

http://www.youtube.com/watch?v=vIFCV2spKtg

---

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 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?

**Longest Path in a DAG**

Challenge. Given an edge-weighted DAG, find the longest path from \( s \) to \( v \).

Warning. Problem in digraphs is **NP-COMPLETE**.

### 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.

```plaintext
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.
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).

---

**4.4 **SHORTEST PATHS

- APIs
- shortest-paths properties
- Dijkstra’s algorithm
- edge-weighted DAGs
- negative weights

---

Shortest paths with negative weights: failed attempts

**Dijkstra.** Doesn’t work with negative edge weights.

**Re-weighting.** Add a constant to every edge weight doesn’t work.

**Conclusion.** Need a different algorithm.
**Negative cycles**

A **negative cycle** is a directed cycle whose sum of edge weights is negative.

**Proposition.** A SPT exists iff no negative cycles.

**Bellman–Ford algorithm**

**Bellman–Ford algorithm**

- **Initialize** distTo[s] = 0 and distTo[v] = \(\infty\) for all other vertices.
- **Repeat** V times:
  - Relax each edge.

**Bellman–Ford algorithm demo**

- **Repeat** V times: relax all E edges.

**Bellman–Ford algorithm demo**

- **Repeat** V times: relax all E edges.

**Proposition.** A SPT exists iff no negative cycles.

---

**Bellman–Ford algorithm**

**Bellman–Ford algorithm**

- **Initialize** distTo[s] = 0 and distTo[v] = \(\infty\) for all other vertices.
- **Repeat** V times:
  - Relax each edge.

**Bellman–Ford algorithm demo**

- **Repeat** V times: relax all E edges.

**Bellman–Ford algorithm demo**

- **Repeat** V times: relax all E edges.
Bellman–Ford algorithm: visualization

Bellman–Ford algorithm: analysis

Proposition. Bellman–Ford 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 to each vertex $v$ for which the shortest path from $s$ to $v$ contains $i$ edges (or fewer).

Bellman–Ford algorithm: practical improvement

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

FIFO implementation. Maintain queue of vertices whose $\text{distTo}[]$ changed.

Overall effect.
- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Single source shortest-paths implementation: cost summary

<table>
<thead>
<tr>
<th>algorithm</th>
<th>restriction</th>
<th>typical case</th>
<th>worst case</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>topological sort</td>
<td>no directed cycles</td>
<td>$E + V$</td>
<td>$E + V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Dijkstra (binary heap)</td>
<td>no negative weights</td>
<td>$E \log V$</td>
<td>$E \log V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Bellman–Ford</td>
<td>no negative cycles</td>
<td>$E V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Bellman–Ford (queue–based)</td>
<td>no negative cycles</td>
<td>$E + V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

Remark 1. Directed cycles make the problem harder.
Remark 2. Negative weights make the problem harder.
Remark 3. Negative cycles makes the problem intractable.
Finding a negative cycle

**Negative cycle.** Add two method to the API for SP.

```java
boolean hasNegativeCycle()
```

```java
Iterable<DirectedEdge> negativeCycle()
```

**Observation.** If there is a negative cycle, Bellman–Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.

**Proposition.** If Bellman–Ford updates any vertex `v` in pass `v`, there exists a negative cycle (and can trace `edgeTo[v]` entries back to find one).

**In practice.** Check for negative cycles more frequently.

---

**Negative cycle application: arbitrage detection**

**Problem.** Given table of exchange rates, is there an arbitrage opportunity?

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>GBP</th>
<th>CHF</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUR</td>
<td>1.350</td>
<td>1</td>
<td>1.888</td>
<td>1.433</td>
<td>1.366</td>
</tr>
<tr>
<td>GBP</td>
<td>1.521</td>
<td>1.126</td>
<td>1</td>
<td>1.614</td>
<td>1.538</td>
</tr>
<tr>
<td>CHF</td>
<td>0.943</td>
<td>0.698</td>
<td>0.620</td>
<td>1</td>
<td>0.953</td>
</tr>
<tr>
<td>CAD</td>
<td>0.995</td>
<td>0.732</td>
<td>0.650</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>

**Ex.** $1,000 \Rightarrow 741$ Euros $\Rightarrow 1,012.206$ Canadian dollars $\Rightarrow 1,007.14497$.

$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$
Negative cycle application: arbitrage detection

Currency exchange graph.
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is \( > 1 \).

\[
0.741 \times 1.366 \times 0.995 = 1.00714497
\]

Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logarithms.
- Set weight of edge \( v \rightarrow w \) to \(-\ln\) (exchange rate from currency \( v \) to \( w \)).
- Multiplication turns to addition; \( > 1 \) turns to \(< 0 \).
- Find a directed cycle whose sum of edge weights is \(< 0 \) (negative cycle).

 '\]

Remark. Fastest algorithm is extraordinarily valuable!

Shortest paths summary

Nonnegative weights.
- Arises in many application.
- Dijkstra’s algorithm is nearly linear-time.

Acyclic edge-weighted digraphs.
- Arise in some applications.
- Topological sort algorithm is linear time.
- Edge weights can be negative.

Negative weights and negative cycles.
- Arise in some applications.
- Bellman–Ford is quadratic in worst case.
- 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.