4.4 Shortest Paths

- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

Continental U.S. routes (August 2010)

Shortest outgoing routes on the Internet from Lumeta headquarters

http://www.continental.com/web/en-US/content/travel/routes

map by Lumeta Corporation, March 8, 2006
Shortest paths in a weighted digraph

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

edge-weighted digraph

Shortest path from 0 to 6

• Edge-weighted digraph API
• Shortest paths properties
• Dijkstra's algorithm
• Edge-weighted DAGs
• Negative weights

Shortest path variants

Which vertices?
- Source-sink: from one vertex to another.
- Single source: from one vertex to every other.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?
- Nonnegative weights.
- Arbitrary weights.
- Euclidean weights.

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

Simplifying assumption. There exists a shortest path from $s$ to each vertex $v$.

Shortest path applications

- Map routing.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Weighted directed edge API

public class DirectedEdge

DirectedEdge(int v, int w, double weight)

int from() vertex v
int to() vertex w
double weight() weight of this edge
String toString() string representation

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.

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;
}

d fish public int from()
{  return v;  }

d fish public int to()
{  return w;  }

d fish public double weight()
{  return weight; }

Edge-weighted digraph API

public class EdgeWeightedDigraph

EdgeWeightedDigraph(int V)

EdgeWeightedDigraph(In in)

void addEdge(DirectedEdge e)

Iterable<DirectedEdge> adj(int v)

int V()

int E()

Iterable<DirectedEdge> edges()

String toString()

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation
**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<Edge>[] 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 {
    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`?
    }

    public static double[] sp(EdgeWeightedDigraph G, int s) {
        shortest paths from `s` in graph `G`
        double[] distTo = new double[G.V()];
        boolean[] hasPathTo = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0;
        for (int k = 0; k < G.V(); k++)
            for (DirectedEdge e : G.adj(k))
                if (distTo[e.from()] + e.weight() < distTo[e.to()])
                    distTo[e.to()] = distTo[e.from()] + e.weight();
        return distTo;
    }
}
```

```
% java SP tinyEWD.txt 0
0 to 0 (0.00): 0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0 to 3 (0.99): 2->7 0.34  7->3 0.39
0 to 4 (0.38): 0 to 5 (0.73): 4->5 0.35
0 to 6 (1.51): 0 to 7 (0.60): 2->7 0.34
```

**Similar to edge-weighted undirected graph but only add edge to `v`'s adjacency list**

This page contains code examples for implementing an edge-weighted digraph in Java, and a class `SP` for finding single-source shortest paths. The code shows how to add edges and retrieve adjacent edges, and the `SP` class demonstrates how to calculate shortest paths from a given source vertex to all other vertices in the graph.
Data structures for single-source shortest paths

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

**Observation.** A shortest path 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 \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>( \text{edgeTo}[v] )</th>
<th>( \text{distTo}[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5-1</td>
<td>1.05</td>
</tr>
<tr>
<td>2</td>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>7-3</td>
<td>0.37</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>5</td>
<td>4-5</td>
<td>0.73</td>
</tr>
<tr>
<td>6</td>
<td>3-6</td>
<td>1.49</td>
</tr>
<tr>
<td>7</td>
<td>2-7</td>
<td>0.34</td>
</tr>
</tbody>
</table>

![Shortest path tree from 0](image)

**Shortest-paths optimality conditions**

**Proposition.** Let \( G \) be an edge-weighted digraph. Then \( \text{distTo}[v] \) are the shortest path distances from \( s \) iff:
- For each vertex \( v \), \( \text{distTo}[v] \) is the length of some path from \( s \) to \( v \).
- For each edge \( e = v \rightarrow w \), \( \text{distTo}[w] \leq \text{distTo}[v] + e \cdot \text{weight}() \).

**Pf.** (necessary)
- Suppose that \( \text{distTo}[w] > \text{distTo}[v] + e \cdot \text{weight}() \) for some edge \( e = v \rightarrow w \).
- Then, \( e \) gives a path from \( s \) to \( w \) through \( v \) of length less than \( \text{distTo}[w] \).
Shortest-paths optimality conditions

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

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

**Pf.** [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from $s$ to $w$.
- Then, $\text{distTo}[v_k] = \text{distTo}[v_{k-1}] + e_k.\text{weight}()$
- $\text{distTo}[v_{k-1}] = \text{distTo}[v_{k-2}] + e_{k-1}.\text{weight}()$
- $\ldots$
- $\text{distTo}[v_1] = \text{distTo}[v_0] + e_1.\text{weight}()$

- Collapsing these inequalities and eliminate $\text{distTo}[v_0] = \text{distTo}[s] = 0$:
  
  $\text{distTo}[w] = \text{distTo}[v_k] = e_k.\text{weight}() + e_{k-1}.\text{weight}() + \ldots + e_1.\text{weight}()$

- Thus, $\text{distTo}[w]$ is the weight of shortest path to $w$. $

Generic shortest-paths algorithm

**Generic algorithm (to compute 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 from $s$, assuming SPT exists.

**Pf sketch.**

- Throughout algorithm, $\text{distTo}[v]$ is the length of a simple path from $s$ to $v$ and $\text{edgeTo}[v]$ is last edge on path.
- Each successful relaxation decreases $\text{distTo}[v]$ for some $v$.
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times. $

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).
Edsger W. Dijkstra: select quotes

“Do only what only you can do.”

“In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”

“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.”

Dijkstra’s algorithm

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

Dijkstra’s algorithm visualization

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

-- Edsger Dijkstra
**Dijkstra's algorithm visualization**

**Shortest path trees**

- 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.

**Dijkstra's algorithm: correctness proof**

**Proposition.** Dijkstra’s algorithm computes SPT in any edge-weighted digraph with nonnegative weights.

**Pf.**

- Each edge \( e = v \rightarrow w \) is relaxed exactly once (when \( v \) 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.

**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]);
    }
}
```

### Dijkstra’s algorithm: which priority queue?

Depends on PQ implementation: \( I \) insert, \( V \) delete-min, \( E \) 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>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>( d \log V )</td>
<td>( d \log V )</td>
<td>( d \log V )</td>
<td>( E \log d \log V )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>1</td>
<td>( \log V )</td>
<td>1</td>
<td>( E + V \log V )</td>
</tr>
</tbody>
</table>

Bottom line:
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- d-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 set of explored vertices \( S \).
- Grow \( S \) by exploring edges with exactly one endpoint leaving \( S \).

**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 \( S \).

**Challenge.** Express this insight in reusable Java code.
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!

Shortest paths in edge-weighted DAGs

Topological sort algorithm.
• Consider vertices in topologically order.
• Relax all edges incident from vertex.

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

Pf.
• Each edge $e = v \rightarrow w$ is relaxed exactly once (when $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.weight()$.
• 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.
Formulate as a shortest paths problem in edge-weighted DAGs.
• Negate all weights.
• Find shortest paths.
• Negate weights in result.

Key point. Topological sort algorithm works even with negative edge weights.

Critical path method

**CPM.** To solve a parallel job-scheduling problem, create acyclic edge-weighted digraph:
• 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)

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time while respecting the constraints.
Deadlines. Add extra constraints to the parallel job-scheduling problem. Ex. “Job 2 must start no later than 12 time units after job 4 starts.”

Consequences.
- Corresponding shortest-paths problem has cycles (and negative weights).
- Possibility of infeasible problem (negative cycles).

Negative cycles

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

Proposition. A SPT exists iff no negative cycles.

Assuming all vertices reachable from s
Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to \( E \cdot V \).

**Pf idea.** After phase \( i \), found shortest path containing at most \( i \) edges.

```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);
```

Bellman-Ford algorithm

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

**FIFO implementation.** Maintain queue of vertices whose \( \text{distTo}[v] \) changed.

**Overall effect.**
- The running time is still proportional to \( E \cdot V \) in worst case.
- But much faster than that in practice.

Bellman-Ford algorithm trace

```java
public class BellmanFordSP
{
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private int[] onQ;
    private Queue<Integer> queue;

    public BellmanFordSP(EdgeWeightedDigraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        onQ = new int[G.V()];
        queue = new Queue<Integer>();

        for (int v = 0; v < V; v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        queue.enqueue(s);

        while (!queue.isEmpty())
        {
            int v = queue.dequeue();
            onQ[v] = false;
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

**Private void relax(DirectedEdge e)**

```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 (!onQ[w])
        {
            queue.enqueue(w);
            onQ[w] = true;
        }
    }
}
```
Bellman-Ford algorithm visualization

**Finding a negative cycle**

**Negative cycle.** Add two method to the API for shortest-path:

- `boolean hasNegativeCycle()` — is there a negative cycle?
- `Iterable <DirectedEdge> negativeCycle()` — negative cycle reachable from `s`

**Finding a negative cycle**

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

**Finding a negative cycle**

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

**Remark 1.** Directed cycles make the problem harder.
**Remark 2.** Negative weights make the problem harder.
**Remark 3.** Negative cycles makes the problem intractable.

**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>Dynamic programming</td>
<td>No negative cycles</td>
<td>$E V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td></td>
<td>$E + V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

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>Currency</th>
<th>USD</th>
<th>EUR</th>
<th>GBP</th>
<th>CHF</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1.00</td>
<td>0.741</td>
<td>0.657</td>
<td>1.061</td>
<td>1.011</td>
</tr>
<tr>
<td>EUR</td>
<td>1.350</td>
<td>1.00</td>
<td>0.888</td>
<td>1.433</td>
<td>1.366</td>
</tr>
<tr>
<td>GBP</td>
<td>1.521</td>
<td>1.126</td>
<td>1.00</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>1.00</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>1.00</td>
</tr>
</tbody>
</table>

Ex. $1,000 ⇒ 741 Euros ⇒ 1,012.206 Canadian dollars ⇒ $1,007.14497.

Challenge. Express as a negative cycle detection problem.

Model as a negative cycle detection problem by taking logs.
- Let weight of edge $v \rightarrow w$ be $-\ln$ (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!