Shortest Paths

- Dijkstra’s algorithm
- implementation
- negative weights

References:
- Algorithms in Java, Chapter 21
  http://www.cs.princeton.edu/introalgsds/55dijkstra
Edsger W. Dijkstra: a few select quotes

The question of whether computers can think is like the question of whether submarines can swim.

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.

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.
Shortest paths in a weighted digraph
Shortest paths in a weighted digraph

Given a weighted digraph, find the shortest directed path from $s$ to $t$.

Note: weights are arbitrary numbers
- not necessarily distances
- need not satisfy the triangle inequality
- Ex: airline fares [stay tuned for others]
Versions

- source-target (s-t)
- single source
- all pairs.
- nonnegative edge weights
- arbitrary weights
- Euclidean weights.
Early history of shortest paths algorithms


Ford (1956). RAND, economics of transportation.


Applications

Shortest-paths is a broadly useful problem-solving model

- Maps
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Subroutine in advanced algorithms.
- Telemarketer operator scheduling.
- 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.

Dijkstra’s algorithm
- implementation
- negative weights
Single-source shortest-paths

**Given.** Weighted digraph, single source $s$.

**Distance** from $s$ to $v$: length of the shortest path from $s$ to $v$.

**Goal.** Find distance (and shortest path) from $s$ to every other vertex.

Shortest paths form a tree
Single-source shortest-paths: basic plan

**Goal:** Find distance (and shortest path) from \( s \) to **every** other vertex.

**Design pattern:**
- **ShortestPaths class** (WeightedDigraph client)
- instance variables: vertex-indexed arrays \( \text{dist[]} \) and \( \text{pred[]} \)
- client query methods return distance and path iterator

Note: Same pattern as Prim, DFS, BFS; BFS works when weights are all 1.
Edge relaxation

For all \( v \), \( \text{dist}[v] \) is the length of some path from \( s \) to \( v \).

Relaxation along edge \( e \) from \( v \) to \( w \).

- \( \text{dist}[v] \) is length of some path from \( s \) to \( v \)
- \( \text{dist}[w] \) is length of some path from \( s \) to \( w \)
- if \( v-w \) gives a shorter path to \( w \) through \( v \), update \( \text{dist}[w] \) and \( \text{pred}[w] \)

\[
\text{if } (\text{dist}[w] > \text{dist}[v] + e \text{.weight}()) \{
\text{dist}[w] = \text{dist}[v] + e \text{.weight}();
\text{pred}[w] = e;
\}
\]

Relaxation sets \( \text{dist}[w] \) to the length of a shorter path from \( s \) to \( w \) (if \( v-w \) gives one)
**Dijkstra's algorithm**

**S**: set of vertices for which the shortest path length from \( s \) is known.

**Invariant**: for \( v \) in \( S \), \( \text{dist}[v] \) is the length of the shortest path from \( s \) to \( v \).

Initialize \( S \) to \( s \), \( \text{dist}[s] \) to 0, \( \text{dist}[v] \) to \( \infty \) for all other \( v \)

Repeat until \( S \) contains all vertices connected to \( s \)

- find \( e \) with \( v \) in \( S \) and \( w \) in \( S' \) that minimizes \( \text{dist}[v] + e.\text{weight()} \)
- relax along that edge
- add \( w \) to \( S \)
Dijkstra's algorithm

S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, dist[v] is the length of the shortest path from s to v.

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s
• find e with v in S and w in S' that minimizes dist[v] + e.weight()
• relax along that edge
• add w to S
Dijkstra's algorithm proof of correctness

**S**: set of vertices for which the shortest path length from \( s \) is known.

**Invariant**: for \( v \) in \( S \), \( \text{dist}[v] \) is the length of the shortest path from \( s \) to \( v \).

**Pf.** (by induction on \( |S| \))

- Let \( w \) be next vertex added to \( S \).
- Let \( P^* \) be the \( s-w \) path through \( v \).
- Consider any other \( s-w \) path \( P \), and let \( x \) be first node on path outside \( S \).
- \( P \) is already longer than \( P^* \) as soon as it reaches \( x \) by greedy choice.
Shortest Path Tree

![Shortest Path Tree Diagrams at 25%, 50%, 75%, and 100%]

- **25%**
- **50%**
- **75%**
- **100%**
Dijkstra’s algorithm
implementation
negative weights
Weighted directed edge data type

```java
public class Edge implements Comparable<Edge> {
    public final int v, int w;
    public final double weight;

    public Edge(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 int compareTo(Edge that) {
        if (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else return 0;
    }
}
```

code is the same as for (undirected) WeightedGraph except from() and to() replace either() and other()
Identical to WeightedGraph but just one representation of each Edge.

```java
public class WeightedDigraph
{
    private int V;
    private SET<Edge>[] adj;

    public Graph(int V)
    {
        this.V = V;
        adj = (SET<Edge>[]) new SET[V];
        for (int v = 0; v < V; v++)
            adj[v] = new SET<Edge>();
    }

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

    public Iterable<Edge> adj(int v)
    { return adj[v]; } 
}
```
Dijkstra's algorithm: implementation approach

Initialize $S$ to $s$, $dist[s]$ to 0, $dist[v]$ to $\infty$ for all other $v$
Repeat until $S$ contains all vertices connected to $s$
  • find $v$–$w$ with $v$ in $S$ and $w$ in $S'$ that minimizes $dist[v] + weight[v$–$w]$
  • relax along that edge
  • add $w$ to $S$

Idea 1 (easy): Try all edges

Total running time proportional to $VE$
Dijkstra's algorithm: implementation approach

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s
  • find v–w with v in S and w in S’ that minimizes dist[v] + weight[v–w]
    • relax along that edge
    • add w to S

Idea 2 (Dijkstra): maintain these invariants
  • for v in S, dist[v] is the length of the shortest path from s to v.
  • for w in S’, dist[w] minimizes dist[v] + weight[v–w].

Two implications
  • find v–w in V steps (smallest dist[] value among vertices in S’)
  • update dist[] in at most V steps (check neighbors of w)

Total running time proportional to V²
Dijkstra's algorithm implementation

Initialize S to s, dist[s] to 0, dist[v] to $\infty$ for all other v
Repeat until S contains all vertices connected to s

• find v–w with v in S and w in S’ that minimizes dist[v] + weight[v–w]
  • relax along that edge
  • add w to S

Idea 3 (modern implementations):
• for all v in S, dist[v] is the length of the shortest path from s to v.
• use a priority queue to find the edge to relax

Total running time proportional to $E \lg E$
Dijkstra's algorithm implementation

Q. What goes onto the priority queue?
A. Fringe vertices connected by a single edge to a vertex in S

Starting to look familiar?
Lazy implementation of Prim's MST algorithm

```java
public class LazyPrim {
    Edge[] pred = new Edge[G.V()];
    public LazyPrim(WeightedGraph G) {
        boolean[] marked = new boolean[G.V()];
        double[] dist = new double[G.V()];
        for (int v = 0; v < G.V(); v++)
            dist[v] = Double.POSITIVE_INFINITY;
        MinPQplus<Double, Integer> pq;
        pq = new MinPQplus<Double, Integer>();
        dist[s] = 0.0;
        pq.put(dist[s], s);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            if (marked[v]) continue;
            marked(v) = true;
            for (Edge e : G.adj(v)) {
                int w = e.other(v);
                if (!marked[w] && (dist[w] > e.weight())) {
                    dist[w] = e.weight();
                    pred[w] = e;
                    pq.insert(dist[w], w);
                }
            }
        }
    }
}
```
**Lazy implementation of Dijkstra's SPT algorithm**

```java
public class LazyDijkstra
{
    double[] dist = new double[G.V()];
    Edge[] pred = new Edge[G.V()];
    public LazyDijkstra(WeightedDigraph G, int s)
    {
        boolean[] marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            dist[v] = Double.POSITIVE_INFINITY;
        MinPQplus<Double, Integer> pq;
pq = new MinPQplus<Double, Integer>();
dist[s] = 0.0;
pq.put(dist[s], s);
while (!pq.isEmpty())
{
    int v = pq.delMin();
    if (marked[v]) continue;
    marked(v) = true;
    for (Edge e : G.adj(v))
    {
        int w = e.to();
        if (dist[w] > dist[v] + e.weight())
        {
            dist[w] = dist[v] + e.weight();
pred[w] = e;
pq.insert(dist[w], w);
        }
    }
}
}

code is **the same** as Prim's (!!)

except
• WeightedDigraph, not WeightedGraph
• weight is distance to \( s \), not to tree
• add client query for distances
```
Dijkstra’s algorithm example

**Dijkstra’s algorithm.** [Dijkstra 1957]
Start with vertex 0 and greedily grow tree $T$. At each step, add cheapest path ending in an edge that has exactly one endpoint in $T$. 

![Graphs showing the process of Dijkstra's algorithm](image)
Eager implementation of Dijkstra’s algorithm

Use indexed priority queue that supports
• contains: is there a key associated with value v in the priority queue?
• decrease key: decrease the key associated with value v

[more complicated data structure, see text]

Putative “benefit”: reduces PQ size guarantee from E to V
• no significant impact on time since \( \lg E < 2\lg V \)
• extra space not important for huge sparse graphs found in practice
  [ PQ size is far smaller than E or even V in practice]
• widely used, but practical utility is debatable (as for Prim’s)
Improvements to Dijkstra’s algorithm

Use a d-way heap (Johnson, 1970s)
• easy to implement
• reduces costs to $E d \log_d V$
• indistinguishable from linear for huge sparse graphs found in practice

Use a Fibonacci heap (Sleator-Tarjan, 1980s)
• very difficult to implement
• reduces worst-case costs (in theory) to $E + V \lg V$
• not quite linear (in theory)
• practical utility questionable

Find an algorithm that provides a linear worst-case guarantee?
[open problem]
Dijkstra's Algorithm: performance summary

Fringe implementation directly impacts performance

Best choice depends on sparsity of graph.
- 2,000 vertices, 1 million edges. heap 2-3x slower than array
- 100,000 vertices, 1 million edges. heap gives 500x speedup.
- 1 million vertices, 2 million edges. heap gives 10,000x speedup.

Bottom line.
- array implementation optimal for dense graphs
- binary heap far better 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:** All 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 from vertex which was discovered least recently.  
**Prim.** Take edge of minimum weight.  
**Dijkstra.** Take edge to vertex that is closest to $s$.  
...

*Gives simple algorithm for many graph-processing problems*

**Challenge:** express this insight in (re)usable Java code
Priority-first search: application example

Shortest s–t paths in Euclidean graphs (maps)
- Vertices are points in the plane.
- Edge weights are *Euclidean distances*.

A sublinear algorithm.
- Assume graph is already in memory.
- Start Dijkstra at s.
- Stop when you reach t.

Even better: exploit geometry
- For edge v–w, use weight \( d(v, w) + d(w, t) - d(v, t) \).
- Proof of correctness for Dijkstra still applies.
- In practice only \( O(V^{1/2}) \) vertices examined.
- Special case of \( A^* \) algorithm

[Practical map-processing programs precompute many of the paths.]
Dijkstra’s algorithm
implementation
negative weights
Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

• 1 oz. gold ⇒ $327.25.
• 1 oz. gold ⇒ £208.10 ⇒ $327.00. [208.10 × 1.5714]
• 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28. [455.2 × 0.6677 × 1.0752]

<table>
<thead>
<tr>
<th>Currency</th>
<th>£</th>
<th>Euro</th>
<th>¥</th>
<th>Franc</th>
<th>$</th>
<th>Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK Pound</td>
<td>1.0000</td>
<td>0.6853</td>
<td>0.005290</td>
<td>0.4569</td>
<td>0.6368</td>
<td>208.100</td>
</tr>
<tr>
<td>Euro</td>
<td>1.4599</td>
<td>1.0000</td>
<td>0.007721</td>
<td>0.6677</td>
<td>0.9303</td>
<td>304.028</td>
</tr>
<tr>
<td>Japanese Yen</td>
<td>189.050</td>
<td>129.520</td>
<td>1.0000</td>
<td>85.4694</td>
<td>120.400</td>
<td>39346.7</td>
</tr>
<tr>
<td>Swiss Franc</td>
<td>2.1904</td>
<td>1.4978</td>
<td>0.011574</td>
<td>1.0000</td>
<td>1.3941</td>
<td>455.200</td>
</tr>
<tr>
<td>US Dollar</td>
<td>1.5714</td>
<td>1.0752</td>
<td>0.008309</td>
<td>0.7182</td>
<td>1.0000</td>
<td>327.250</td>
</tr>
<tr>
<td>Gold (oz.)</td>
<td>0.004816</td>
<td>0.003295</td>
<td>0.0000255</td>
<td>0.002201</td>
<td>0.003065</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
Shortest paths application: Currency conversion

Graph formulation.
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes **product** of weights.
Reduce to shortest path problem by taking logs

- Let weight(v-w) = - lg (exchange rate from currency v to w)
- multiplication turns to addition
- Shortest path with costs c corresponds to best exchange sequence.

**Challenge.** Solve shortest path problem with negative weights.
Shortest paths with negative weights: failed attempts

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

Dijkstra selects vertex 3 immediately after 0. But shortest path from 0 to 3 is 0→1→2→3.

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

Adding 9 to each edge changes the shortest path because it adds 9 to each segment, wrong thing to do for paths with many segments.

**Bad news:** need a different algorithm.
Shortest paths with negative weights: **negative cycles**

**Negative cycle.** Directed cycle whose sum of edge weights is negative.

**Observations.**
- If negative cycle \( C \) on path from \( s \) to \( t \), then shortest path can be made arbitrarily negative by spinning around cycle.
- There exists a shortest \( s-t \) path that is simple.

**Worse news:** need a different **problem**
Shortest paths with negative weights

Problem 1. Does a given digraph contain a negative cycle?

Problem 2. Find the shortest simple path from s to t.

Bad news: Problem 2 is intractable
Good news: Can solve problem 1 in $O(VE)$ steps
Good news: Same algorithm solves problem 2 if no negative cycle

Bellman-Ford algorithm
- detects a negative cycle if any exist
- finds shortest simple path if no negative cycle exists
Edge relaxation

For all $v$, $dist[v]$ is the length of some path from $s$ to $v$.

Relaxation along edge $e$ from $v$ to $w$.
- $dist[v]$ is length of some path from $s$ to $v$
- $dist[w]$ is length of some path from $s$ to $w$
- if $v$-$w$ gives a shorter path to $w$ through $v$, update $dist[w]$ and $pred[w]$

```java
if (dist[w] > dist[v] + e.weight()){
    dist[w] = dist[v] + e.weight();
    pred[w] = e;
}
```

Relaxation sets $dist[w]$ to the length of a shorter path from $s$ to $w$ (if $v$-$w$ gives one)
Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

- Initialize \( \text{dist}[v] = \infty, \text{dist}[s] = 0 \).
- Repeat \( v \) times: relax each edge \( e \).

```java
for (int i = 1; i <= G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (Edge e : G.adj(v))
        {
            int w = e.to();
            if (dist[w] > dist[v] + e.weight())
            {
                dist[w] = dist[v] + e.weight();
                pred[w] = e;
            }
        }
```
Shortest paths with negative weights: **dynamic programming algorithm**

**Running time proportional to** $E \cdot V$

**Invariant.** At end of phase $i$, $\text{dist}[v] \leq$ length of any path from $s$ to $v$ using at most $i$ edges.

**Theorem.** If there are no negative cycles, upon termination $\text{dist}[v]$ is the length of the shortest path from $s$ to $v$.

And $\text{pred}[]$ gives the shortest paths.
Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Observation. If $\text{dist}[v]$ doesn't change during phase $i$, no need to relax any edge leaving $v$ in phase $i+1$.

FIFO implementation.
Maintain queue of vertices whose distance changed.

be careful to keep at most one copy of each vertex on queue

Running time.
• still could be proportional to EV in worst case
• much faster than that in practice
Initialize $\text{dist}[v] = \infty$ and $\text{marked}[v] = \text{false}$ for all vertices $v$.

```java
Queue<Integer> q = new Queue<Integer>();
marked[s] = true;
dist[s] = 0;
q.enqueue(s);
```

```java
while (!q.isEmpty())
{
    int v = q.dequeue();
    marked[v] = false;
    for (Edge e : G.adj(v))
    {
        int w = e.target();
        if (dist[w] > dist[v] + e.weight())
        {
            dist[w] = dist[v] + e.weight();
            pred[w] = e;
            if (!marked[w])
            {
                marked[w] = true;
                q.enqueue(w);
            }
        }
    }
}
```
## Single Source Shortest Paths Implementation: Cost Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Typical Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nonnegative Costs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dijkstra (classic)</td>
<td>$V^2$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Dijkstra (heap)</td>
<td>$E \log E$</td>
<td>$E$</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>$EV$</td>
<td>$EV$</td>
</tr>
<tr>
<td>Bellman-Ford-Moore</td>
<td>$EV$</td>
<td>$E$</td>
</tr>
</tbody>
</table>

**Remark 1.** Negative weights makes the problem harder.

**Remark 2.** Negative cycles makes the problem intractable.
Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?
- Ex: $1 \Rightarrow 1.3941$ Francs $\Rightarrow 0.9308$ Euros $\Rightarrow 1.00084$.
- Is there a negative cost cycle?
- Fastest algorithm is valuable!

\[-0.4793 + 0.5827 - 0.1046 < 0\]
Negative cycle detection

If there is a negative cycle reachable from \( s \).
Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.

Finding a negative cycle. If any vertex \( v \) is updated in phase \( v \), there exists a negative cycle, and we can trace back \( \text{pred}[v] \) to find it.
Negative cycle detection

**Goal.** Identify a negative cycle (reachable from any vertex).

**Solution.** Add 0-weight edge from artificial source $s$ to each vertex $v$. Run Bellman-Ford from vertex $s$. 
Shortest paths summary

Dijkstra’s algorithm
• easy and optimal for dense digraphs
• PQ/ST data type gives near optimal for sparse graphs

Priority-first search
• generalization of Dijkstra’s algorithm
• encompasses DFS, BFS, and Prim
• enables easy solution to many graph-processing problems

Negative weights
• arise in applications
• make problem intractable in presence of negative cycles (!)
• easy solution using old algorithms otherwise

Shortest-paths is a broadly useful problem-solving model