Shortest Paths

Brief History

**Shimbel (1955).** Information networks.

**Ford (1956).** RAND, economics of transportation.


**Dantzig (1958).** Simplex method for linear programming.

**Bellman (1958).** Dynamic programming.

**Moore (1959).** Routing long-distance telephone calls for Bell Labs.

**Dijkstra (1959).** Simpler and faster version of Ford’s algorithm.

Typeset in Adobe’s PostScript font.

Dijkstra's Algorithm

**Assumptions.**
- Digraph \( G \).
- Single source \( s \).
- Edge weights \( c(v, w) \) are nonnegative.

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

**Valid weights.** For all vertices \( v \), \( \pi(v) \) is length of some path from \( s \) to \( v \).

**Edge relaxation.**
- Consider edge \( e = v \rightarrow w \).
- If current path from \( s \) to \( v \) plus edge \( v \rightarrow w \) is shorter than current path to \( w \), then update current path to \( w \).

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

Dijkstra's Algorithm

Maintain set of weights \( \pi(v) \) and a set of explored vertices \( S \) for which \( \pi(v) \) is the length shortest \( s - v \) path.
- Initialize: \( S = \{ s \} \), \( \pi(s) = 0 \).
- Repeatedly choose unexplored node \( w \) which minimizes:

\[
\pi(w) = \min_{v \in S} \pi(v) + c(v, w)
\]

- set \( \text{pred}[w] = v \)
- add \( w \) to \( S \), and set \( \pi(w) = \pi(v) + c(v, w) \)
Dijkstra’s Algorithm

Dijkstra’s algorithm. Maintain set of weights $\pi(v)$ and a set of explored vertices $S$ for which $\pi(v)$ is the length of shortest $s-v$ path.

- Initialize: $S = \{s\}$, $\pi(s) = 0$.
- Repeatedly choose unexplored node $w$ which minimizes:
  \[
  \pi(w) = \min_{v \in S} (\pi(v) + c(v,w))
  \]
  - set $\text{pred}[w] = v$
  - add $w$ to $S$, and set $\pi(w) = \pi(v) + c(v,w)$

Dijkstra’s Algorithm: Proof of Correctness

Invariant. For each vertex $v$ in $S$, $\pi(v)$ is the length of shortest $s-v$ path.

Pf. (by induction on $|S|$)
- Let $w$ be next vertex added to $S$.
- $\pi(w) = \pi(v) + c(v,w)$ is length of some $s-v$ path.
- Consider any $s-v$ path $P$, and let $x$ be first node on path outside $S$.
- $P$ is already too long as soon as it reaches $x$ by greedy choice.
Dijkstra’s Algorithm: Implementation

Critical step. Choose unexplored node $w$ which minimizes:

$$\pi(w) = \min_{v \in V \setminus S} \pi(v) + c(v, w)$$

Brute force implementation. Test all edges $\Rightarrow O(EV)$ time.

Efficient implementation. Maintain a priority queue of unexplored vertices, prioritized by $\pi(w)$.

Q. How to maintain $\pi$?
A. When exploring $v$, for each edge $v \rightarrow w$ leaving $v$, update

$$\pi(w) = \min \{ \pi(w), \pi(v) + c(v, w) \}.$$
Dijkstra’s Algorithm: Java Implementation

```java
public Dijkstra(WeightedDigraph G, int s) {
    pi = new double[G.V()];
    pred = new Edge[G.V()];
    for (int v = 0; v < G.V(); v++) pi[v] = INFINITY;

    IndexMinPQ<Double> pq = new IndexMinPQ<Double>(G.V());
    pi[s] = 0.0;
    pq.insert(s, pi[s]);

    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (Edge e : G.adj(v)) {
            int w = e.target;
            if (pi[w] > pi[v] + e.weight) {
                pi[w] = pi[v] + e.weight;
                pred[w] = e;
                if (pq.contains(w)) pq.decrease(w, pi[w]);
                else pq.insert(w, pi[w]);
            }
        }
    }
}
```

Indexed Priority Queue

Indexed PQ: Array Implementation

- Maintain vertex indexed array `keys[i]`.
- Insert key: change `keys[i]`.
- Decrease key: change `keys[i]`.
- Delete min: scan through `keys[i]` for each item `i`.
- Maintain a boolean array `marked[i]` to mark items in the PQ.

### Operation Array Dijkstra

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>1</td>
<td>× V</td>
</tr>
<tr>
<td>delete-min</td>
<td>V</td>
<td>× V</td>
</tr>
<tr>
<td>decrease-key</td>
<td>1</td>
<td>× E</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>× V</td>
</tr>
<tr>
<td>contains</td>
<td>1</td>
<td>× V</td>
</tr>
<tr>
<td>total</td>
<td>V^2</td>
<td></td>
</tr>
</tbody>
</table>

Indexed PQ: Binary Heap Implementation

- Assume items are named 0 to N-1.
- Store priorities in a binary heap.

```
    30
     / 
    23  17
   /   / 
  20   8  0
 /  \
 5 2 30
```

How to decrease key of item `i`? Bubble it up.

How to know which heap node to bubble up? Maintains an extra array `qp[i]` that stores the heap index of item `i`.

Indexed PQ

- Assume items are named 0 to N-1.
- Insert, delete min, test if empty.
- Decrease key, contains.

[ST-like ops]
Dijkstra’s Algorithm: Priority Queue Choice

The choice of priority queue matters in Dijkstra’s implementation.
- Array: \(\Theta(V^2)\).
- Binary heap: \(O(E \log V)\).
- Fibonacci heap: \(O(E + V \log V)\).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Binary heap</th>
<th>Fib heap</th>
<th>Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>(V)</td>
<td>(\log V)</td>
<td>(1^+)</td>
<td>(\times V)</td>
</tr>
<tr>
<td>delete-min</td>
<td>(V)</td>
<td>(\log V)</td>
<td>(\log V^+)</td>
<td>(\times V)</td>
</tr>
<tr>
<td>decrease-key</td>
<td>(V)</td>
<td>(\log V)</td>
<td>(1^+)</td>
<td>(\times E)</td>
</tr>
<tr>
<td>is-empty</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(\times V)</td>
</tr>
<tr>
<td>contains</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(\times V)</td>
</tr>
<tr>
<td>total</td>
<td>(V^2)</td>
<td>(E \log V)</td>
<td>(E + V \log V)</td>
<td></td>
</tr>
</tbody>
</table>

† amortized

Best choice depends on sparsity of graph.
- 2,000 vertices, 1 million edges. **Heap**: 2-3x slower.
- 100,000 vertices, 1 million edges. **Heap**: 500x faster.
- 1 million vertices, 2 million edges. **Heap**: 10,000x faster.

Bottom line.
- Array implementation optimal for dense graphs.
- Binary heap far better for sparse graphs.
- Fibonacci heap best in theory, but not in practice.
Priority First Search

Priority first search. Maintain a set of explored vertices $S$, and grow $S$ by exploring edges with exactly one endpoint leaving $S$.

DFS. Edge from vertex which was discovered most recently.
BFS. Edge from vertex which was discovered least recently.
Prim. Edge of minimum weight.
Dijkstra. Edge to vertex which is closest to $s$.

Bellman-Ford-Moore

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.

Application: Currency Conversion

Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

- 1 oz. gold $\Rightarrow$ $327.25.$
- 1 oz. gold £208.10 $\Rightarrow$ $327.00.$
- 1 oz. gold 455.2 Francs $\Rightarrow$ 304.39 Euros $\Rightarrow$ $327.28.$

<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>1.3929</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>
Application: Currency Conversion

Graph formulation.
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.

$$\begin{array}{c|c|c|c}
\text{Vertex} & \text{Currency} & \text{Weight} \\
\hline
1 & 2.1904 & 0.003065 \\
2 & 0.6677 & 0.7182 \\
3 & 1.0752 & 208.100 \\
4 & 455.2 & 2.1904 \\
5 & 0.6677 & 0.7182 \\
6 & 1.0752 & 208.100 \\
7 & 129.520 & 0.003065 \\
8 & 29 & 0.003065 \\
\end{array}$$

Application: Currency Conversion

Reduction to shortest path problem.
- Let $\gamma(v, w)$ be exchange rate from currency $v$ to $w$.
- Let $c(v, w) = -\log \gamma(v, w)$.
- 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. Can fail if 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 can fail.

Adding 9 to each edge changes the shortest path.

Shortest Paths: Negative Cost Cycles

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

Observation. If negative cycle $C$ on path from $s$ to $t$, then shortest path can be made arbitrarily negative by spinning around cycle; otherwise, there exists a shortest $s$–$t$ path that is simple.
Dynamic Programming

Dynamic programming algorithm.
- Initialize $\pi[v] = \infty$, $\pi[s] = 0$.
- Repeat $V$ times: relax each edge $e$.

```
for (int i = 1; i <= V; i++) {
    for (int v = 0; v < G.V(); v++) {
        for (Edge e : G.adj(v)) {
            int w = e.target;
            if (\pi[w] > \pi[v] + e.weight) {
                \pi[w] = \pi[v] + e.weight;
                pred[w] = v;
            }
        }
    }
}
```

Bellman-Ford-Moore

Observation. If $\pi[v]$ 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 $\Omega(EV)$ in worst case, but much faster in practice.

Dynamic Programming: Analysis

Running time. $\Theta(EV)$.

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

Theorem. Assuming no negative cycles, upon termination $\pi[v]$ is the length of the shortest path from from $s$ to $v$, and $\text{pred}[v]$ are the shortest paths.

Review: Edge Relaxation

Valid weights. For all vertices $v$, $\pi(v)$ is length of some path from $s$ to $v$.

Edge relaxation.
- Consider edge $e = v \rightarrow w$.
- If current path from $s$ to $v$ plus edge $v \rightarrow w$ is better than current path to $w$, then update current path to $w$.

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

Dynamic Programming: Analysis

Running time. $\Theta(EV)$.

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

Theorem. Assuming no negative cycles, upon termination $\pi[v]$ is the length of the shortest path from from $s$ to $v$, and $\text{pred}[v]$ are the shortest paths.
Arbitrage

Is there an arbitrage opportunity in currency graph?

- Ex: $1 \rightarrow 1.3941\text{ Francs} \rightarrow 0.9308\text{ Euros} \rightarrow 0.6677\text{ Francs} < 0$

-0.4793 + 0.5827 - 0.1046 < 0

-0.4793

-lg(0.6677) = 0.5827

-0.1046

Bellman-Ford-Moore Algorithm

Initialize $\pi[v] = \infty$ and $\text{marked}[v] = \text{false}$ for all vertices $v$.

```java
Queue<Integer> q = new Queue<Integer>();
marked[s] = true;
\pi[s] = 0;
q.enqueue(s);
while (!q.isEmpty(v)) {
    int v = q.dequeue();
    for (Edge e : G.adj(v)) {
        \text{int } w = e.target;
        if (\pi[w] > \pi[v] + e.weight) {
            \pi[w] = \pi[v] + e.weight;
            pred[w] = v;
            \text{if (!marked[w])} {
                \text{marked[w] = true;
                q.enqueue(w);
            }
        }
    }
}
```

Single Source Shortest Paths Implementation: Cost Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra (classic)</td>
<td>$V^2$</td>
<td>$V^2$</td>
<td>E</td>
</tr>
<tr>
<td>Dijkstra (heap)</td>
<td>$E \log V$</td>
<td>$E$</td>
<td>E</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>$E V$</td>
<td>$E V$</td>
<td>E</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>$E V$</td>
<td>$E V$</td>
<td>E</td>
</tr>
</tbody>
</table>

Remark 1. Negative weights makes the problem harder.
Remark 2. Negative cycles makes the problem intractable.

Arbitrage

If negative cycle reachable from $s$, Bellman-Ford-Moore gets stuck in infinite loop, updating vertices in a 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 Path in a DAG

#### Shortest/Longest Path in DAG

**Shortest path in DAG algorithm.**
- Consider vertices \( v \) in topological order:
  - relax each edge \( v \rightarrow w \)

**Theorem.** Algorithm computes shortest path in linear time (even if negative edge weights).