Introduction to Algorithms

Chap 25

Shortest Paths III
• All-pairs shortest paths
• Matrix-multiplication algorithm
• Floyd-Warshall algorithm
• Johnson’s algorithm

Hsin-Lung Wu, CSIE, NTPU
Shortest paths

**Single-source shortest paths**

- Nonnegative edge weights
  - Dijkstra’s algorithm: \(O(E + V \log V)\)
- General
  - Bellman-Ford algorithm: \(O(VE)\)
- DAG
  - One pass of Bellman-Ford: \(O(V + E)\)
Shortest paths

**Single-source shortest paths**

- Nonnegative edge weights
  - Dijkstra’s algorithm: $O(E + V \log V)$
- General
  - Bellman-Ford: $O(VE)$
- DAG
  - One pass of Bellman-Ford: $O(V + E)$

**All-pairs shortest paths**

- Nonnegative edge weights
  - Dijkstra’s algorithm $|V|$ times: $O(VE + V^2 \log V)$
- General
  - Three algorithms today.
All-pairs shortest paths

**Input:** Digraph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, with edge-weight function $w : E \rightarrow \mathbb{R}$.

**Output:** $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$. 
All-pairs shortest paths

**Input:** Digraph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, with edge-weight function $w : E \to \mathbb{R}$.

**Output:** $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

**Idea:**
- Run Bellman-Ford once from each vertex.
- Time = $O(V^2E)$.
- Dense graph ($n^2$ edges) $\Rightarrow \Theta(n^4)$ time in the worst case.

*Good first try!*
Dynamic programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define

$$d_{ij}^{(m)} = \text{weight of a shortest path from } i \text{ to } j \text{ that uses at most } m \text{ edges}.$$

**Claim:** We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

and for $m = 1, 2, \ldots, n - 1$,

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}.$$
Proof of claim

\[ d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \} \]
Proof of claim

\[ d_{ij}(m) = \min_k \{ d_{ik}(m-1) + a_{kj} \} \]

Relaxation!

for \( k \leftarrow 1 \) to \( n \)

\begin{align*}
\text{do if } & d_{ij} > d_{ik} + a_{kj} \\
\text{then } & d_{ij} \leftarrow d_{ik} + a_{kj}
\end{align*}
Proof of claim

\[ d_{ij}(m) = \min_k \{d_{ik}(m-1) + a_{kj}\} \]

Relaxation!

for \( k \leftarrow 1 \) to \( n \)
do if \( d_{ij} > d_{ik} + a_{kj} \) then \( d_{ij} \leftarrow d_{ik} + a_{kj} \)

Note: No negative-weight cycles implies

\[ \delta(i, j) = d_{ij}(n-1) = d_{ij}(n) = d_{ij}(n+1) = \ldots \]
Matrix multiplication

Compute $C = A \cdot B$, where $C$, $A$, and $B$ are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$ 

Time = $\Theta(n^3)$ using the standard algorithm.
Matrix multiplication

Compute $C = A \cdot B$, where $C$, $A$, and $B$ are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Time $= \Theta(n^3)$ using the standard algorithm.

What if we map “+” $\rightarrow$ “min” and “·” $\rightarrow$ “+”?
Matrix multiplication

Compute $C = A \cdot B$, where $C$, $A$, and $B$ are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$ 

Time $= \Theta(n^3)$ using the standard algorithm.

What if we map “+” $\rightarrow$ “min” and “\cdot” $\rightarrow$ “+”?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\}.$$ 

Thus, $D^{(m)} = D^{(m-1)} \times A$.

Identity matrix $= I = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty \end{pmatrix} = D^0 = (d_{ij}^{(0)})$. 
Matrix multiplication (continued)

The \((\min, +)\) multiplication is **associative**, and with the real numbers, it forms an algebraic structure called a **closed semiring**.

Consequently, we can compute

\[
\begin{align*}
D^{(1)} &= D^{(0)} \cdot A = A^1 \\
D^{(2)} &= D^{(1)} \cdot A = A^2 \\
&\vdots \\
D^{(n-1)} &= D^{(n-2)} \cdot A = A^{n-1},
\end{align*}
\]

yielding \(D^{(n-1)} = (\delta(i, j))\).

Time = \(\Theta(n \cdot n^3) = \Theta(n^4)\). No better than \(n \times B-F\).
Improved matrix multiplication algorithm

Repeated squaring: \( A^{2k} = A^k \times A^k \).

Compute \( A^2, A^4, \ldots, A^{2^{\lceil \lg(n-1) \rceil}} \).

\( O(\lg n) \) squarings

Note: \( A^{n-1} = A^n = A^{n+1} = \ldots \).

Time = \( \Theta(n^3 \lg n) \).

To detect negative-weight cycles, check the diagonal for negative values in \( O(n) \) additional time.
Floyd-Warshall algorithm

Also dynamic programming, but faster!

Define $c_{ij}^{(k)} = \text{weight of a shortest path from } i \text{ to } j \text{ with intermediate vertices belonging to the set } \{1, 2, \ldots, k\}.$

Thus, $\delta(i, j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.
Floyd-Warshall recurrence

\[ c_{ij}^{(k)} = \min_k \{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \} \]

intermediate vertices in \( \{1, 2, \ldots, k\} \)
Pseudocode for Floyd-Warshall

for \( k \leftarrow 1 \) to \( n \)
    do for \( i \leftarrow 1 \) to \( n \)
        do for \( j \leftarrow 1 \) to \( n \)
            do if \( c_{ij} > c_{ik} + c_{kj} \)
                then \( c_{ij} \leftarrow c_{ik} + c_{kj} \)

Notes:

• Okay to omit superscripts, since extra relaxations can’t hurt.
• Runs in \( \Theta(n^3) \) time.
• Simple to code.
• Efficient in practice.
Transitive closure of a directed graph

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

**Idea:** Use Floyd-Warshall, but with $(\lor, \land)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \lor (t_{ik}^{(k-1)} \land t_{kj}^{(k-1)}).$$

Time $= \Theta(n^3)$. 
Graph reweighting

**Theorem.** Given a function \( h : V \rightarrow \mathbb{R} \), *reweight* each edge \((u, v) \in E\) by \( w_h(u, v) = w(u, v) + h(u) - h(v) \). Then, for any two vertices, all paths between them are reweighted by the same amount.
Graph reweighting

Theorem. Given a function $h : V \to \mathbb{R}$, reweight each edge $(u, v) \in E$ by $w_h(u, v) = w(u, v) + h(u) - h(v)$. Then, for any two vertices, all paths between them are reweighted by the same amount.

Proof. Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path in $G$. We have

$$w_h(p) = \sum_{i=1}^{k-1} w_h(v_i, v_{i+1})$$

$$= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}))$$

$$= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k)$$

$$= w(p) + h(v_1) - h(v_k).$$

Same amount!
Shortest paths in reweighted graphs

**Corollary.** $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$. 
Shortest paths in reweighted graphs

Corollary. $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$.

**Idea:** Find a function $h : V \rightarrow \mathbb{R}$ such that $w_h(u, v) \geq 0$ for all $(u, v) \in E$. Then, run Dijkstra’s algorithm from each vertex on the reweighted graph.

**Note:** $w_h(u, v) \geq 0$ iff $h(v) - h(u) \leq w(u, v)$. 
Johnson’s algorithm

1. Find a function \( h : V \rightarrow \mathbb{R} \) such that \( w_h(u, v) \geq 0 \) for all \( (u, v) \in E \) by using Bellman-Ford to solve the difference constraints \( h(v) - h(u) \leq w(u, v) \), or determine that a negative-weight cycle exists.
   - Time = \( O(VE) \).

2. Run Dijkstra’s algorithm using \( w_h \) from each vertex \( u \in V \) to compute \( \delta_h(u, v) \) for all \( v \in V \).
   - Time = \( O(VE + V^2 \lg V) \).

3. For each \( (u, v) \in V \times V \), compute
   \[
   \delta(u, v) = \delta_h(u, v) - h(u) + h(v) .
   \]
   - Time = \( O(V^2) \).

Total time = \( O(VE + V^2 \lg V) \).