All-pairs shortest paths

Suprakash Datta

March 11, 2011
Single source shortest paths:

Shortest paths from a given node to all nodes.

Dijkstra’s algorithm:

- Complexity (using heaps): $O((V + E) \log V)$.
- Complexity (using Fibonacci heaps): $O(V \log V + E)$.
- Cannot handle negative edges

Single-source shortest paths with negative edges:

Bellman-Ford Algorithm

- may be covered later, covered in CSE3101
- Complexity: $\Theta(VE)$
Compute shortest paths between all pairs of nodes.

Possible approaches:

1. Run Dijkstra from each node
   - Complexity (using heaps): $O((V^2 + VE) \log V)$
   - Complexity (using Fibonacci heaps): $O(V^2 \log V + VE)$.
   - Cannot handle negative edges

2. Run Bellman-Ford from each node
   - Complexity: $O(V^2 E)$ – recall that $E$ may be $\Theta(V^2)$
   - Simple code, works with negative edges

3. Floyd-Warshall algorithm
   - Complexity: $O(V^3)$
   - Simple code, works with negative edges
INPUT: a directed graph $G = (V, E)$.

- Nodes are numbered $1 \ldots n$.
- Each edge $\langle i, j \rangle$ has a real-valued weight $w_{ij}$ \(^1\)
- Assume that all cycles have non-negative cost.

OUTPUT: A matrix $D = [d_{ij}]$, $d_{ij} =$ the (cost of the) shortest path from $i$ to $j$

\(^1\)formally $w : E \rightarrow \mathbb{R}$. 
The Floyd Warshall algorithm - Intuition

Definition

\( c^{(k)}_{ij} \): the cost of the shortest path from \( i \) to \( j \), with the intermediate nodes being restricted to the set \( \{1, 2, \ldots, k\} \).

- Therefore

\[
\begin{align*}
  c^{(0)}_{ij} &= 0 \text{ if } i = j \quad (1) \\
  &= w_{ij} \text{ if } i \neq j \text{ and } (i, j) \in E, \quad (2) \\
  &= \infty \text{ otherwise} \quad (3)
\end{align*}
\]

- Therefore

\[
c^{(k)}_{ij} = \min[c^{(k-1)}_{ij}, c^{(k-1)}_{ik} + c^{(k-1)}_{kj}]
\]

- The (cost of the) shortest path from \( i \) to \( j \) is \( d_{ij} = c^{(n)}_{ij} \).
The Floyd Warshall algorithm - Intuition

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

**Figure:** Two possibilities for node \( k \)
In the expression $c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$, there is nothing that says the intermediate nodes in $c_{ik}^{(k-1)}$ are disjoint from those in $c_{kj}^{(k-1)}$. What if there is overlap?

**Figure: Scenario 1**
In the expression $c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$, there is nothing that says the intermediate nodes in $c_{ik}^{(k-1)}$ are disjoint from those in $c_{kj}^{(k-1)}$. What if there is overlap?

**Figure:** Scenario 2
The Floyd Warshall algorithm - steps

Input: \( W = [w_{ij}] \)
Output: \( D = [d_{ij}] \)

\( \text{FLOYD-WARSHALL}(W) \)

1. \( n \leftarrow \text{rows}(W) \)
2. \( D^{(0)} \leftarrow W \)
3. \( \text{for } k \leftarrow 1 \text{ to } n \)
4. \( \text{do } D^{(k)} \leftarrow D^{(k-1)} \)
5. \( \text{for } i \leftarrow 1 \text{ to } n \)
6. \( \text{do for } j \leftarrow 1 \text{ to } n \)
7. \( \text{do if } D^{(k)}[i,j] > D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \)
8. \( \text{then } D^{(k)}[i,j] \leftarrow D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \)
9. \( \text{return } D^{(n)} \)

Compare with: \( c_{ij}^{(k)} = \min[c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}] \).
The Floyd Warshall algorithm - simplified

FLOYD-WARSHALL(W)
1  \( n \leftarrow \text{rows}(W) \)
2  \( D \leftarrow W \)
3  \textbf{for} \( k \leftarrow 1 \) \textbf{to} \( n \)
4    \textbf{do} \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \)
5      \textbf{do} \textbf{for} \( j \leftarrow 1 \) \textbf{to} \( n \)
6        \textbf{do if} \( D[i, j] > D[i, k] + D[k, j] \)
7          \textbf{then} \( D[i, j] \leftarrow D[i, k] + D[k, j] \)
8  \textbf{return} \( D \)

Notes

- \textbf{recall}: \( w_{ij} = 0 \), if \( i = j \), weight of edge \( \langle i, j \rangle \) if it exists, otherwise \( \infty \)

- the pseudocode assumes the use of \( \infty \) – use a suitable guard in your code. For today, all edges are non-negative; so -1 can be a guard value.
The Floyd Warshall algorithm - computing paths

Maintain a matrix called \textit{via}
\begin{verbatim}
FLOYD-WARSHALL(W)
    1  n ← rows(W)
    2  D ← W
    3  via ← nil  //initialize all entries
    4  for k ← 1 to n
    5      do for i ← 1 to n
    6           do for j ← 1 to n
    7              do if \(D[i, j] > D[i, k] + D[k, j]\)
    8                  then \(D[i, j] ← D[i, k] + D[k, j]\)
    9                      via[i, j] ← k
    10  return D, via
\end{verbatim}
The Floyd Warshall algorithm - printing paths

\text{PrintFWPath}(i, j)
1 \quad \text{if } \text{via}[i, j] = \text{nil}
2 \quad \text{then } \text{print}(i, j) \quad \text{If you want to}
3 \quad \text{else } \text{PrintFWPath}(i, \text{via}[i, j])
4 \quad \text{PrintFWPath}(\text{via}[i, j], j)

store the path (instead of print) insert into a queue instead.
Example of **dynamic programming**

1. When applicable, often provides very efficient solutions to optimization problems
2. Many contest problems require this technique
3. Covered in detail in CSE 3101 (Design and Analysis of Algorithms).

The efficiency arises from the clever formulation – the more obvious dynamic programming formulation yields a $\Theta(V^4)$-time algorithm.

Proofs of correctness are skipped here.