All-Pairs Shortest Paths

Analysis of Algorithms
All-Pairs Shortest Paths

- If a path has no negative edge weights, then we could simply make \( N \) runs of Dijkstra’s algorithm, at a total runtime cost of \( N (M \log N) \)
- If there are negative edge weights then we could make \( N \) runs of the Bellman-Ford algorithm at a total runtime cost of \( N^2 M \)
All-Pairs Shortest Paths

- However, notice that a shortest path must contain within it shortest paths
- This is the *optimal sub-structure* property that is the hallmark of a dynamic programming approach
- So let’s try a dynamic programming method to solve the all-pairs shortest path problem
Floyd-Warshall Method

- The Floyd-Warshall algorithm uses the adjacency matrix representation of the graph, hence it is most suitable for dense graphs.
- For this algorithm, we shall define \( D^{(k)} \) to be the matrix whose \((i,j)\) entry is \( d_{i,j}^{(k)} \), where \( d_{i,j}^{(k)} \) is the length of the shortest path from vertex \( i \) to vertex \( j \) whose intermediate vertices all lie in the set \( \{1, 2, ..., k\} \).
  - Assume vertex IDs start at 1 (not 0)
Floyd-Warshall Method

• After we have constructed the matrix $D^{(k-1)}$, we will use it to construct the matrix $D^{(k)}$

• But what is the matrix $D^{(0)}$?
  – The entry $d_{i,j}^{(0)}$ is the length of the shortest path from vertex $i$ to vertex $j$ using no intermediate vertices (i.e., it is just the length of the single edge from vertex $i$ to vertex $j$; $\infty$ if no such edge exists)
  – Therefore, $D^{(0)}$ is simply the adjacency matrix $A$
Floyd-Warshall Method

• Next, we consider all the paths from vertex $i$ to vertex $j$ whose intermediate vertices all lie in the set \{1, 2, ..., $k$\}
  – There are two possibilities:
    (1) The shortest path from $i$ to $j$ does not go through vertex $k$
    (2) The shortest path from $i$ to $j$ does go through vertex $k$

• Calculate both possibilities and store the better (smaller) value
Floyd-Warshall Method

- If the path does not go through vertex $k$, then the shortest path is simply looked up from the previous matrix, $D^{(k-1)}$
  - This is $d_{i,j}^{(k-1)}$
- If the path does go through vertex $k$ then it must have gone from vertex $i$ to vertex $k$, and then proceeded on to vertex $j$, both of which are looked up from previous matrix, $D^{(k-1)}$
  - This is $d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$
- In summary, set $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
Example: Consider the weighted directed graph with the following adjacency matrix:

\[
D^{(0)} = \begin{pmatrix}
0 & \infty & 11 & 2 & 6 \\
1 & 0 & 4 & \infty & \infty \\
10 & \infty & 0 & \infty & \infty \\
\infty & 2 & 6 & 0 & 3 \\
\infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]

Let us see how to use this to compute \(D^{(1)}\) (that is, all paths that go through vertex 1)
To find the (2,4) entry of the $D^{(1)}$ matrix we have to consider the paths through the vertex 1.

Is there a path from 2 to 1, then from 1 to 4, that has a better value than the current path?

Yes, because $1 + 2 = 3 < \infty$, so entry (2,4) is updated to 3.
### Floyd-Warshall Method

Let \( d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \)

Entry \((2, 4)\) in \(D^{(1)}\) \(k = 1, k-1 = 0\), \(i = 2\), \(j = 4\):

\[
d_{2,4}^{(1)} = \min(\infty, d_{2,1}^{(0)} + d_{1,4}^{(0)})
\]

\[
d_{2,4}^{(1)} = \min(\infty, 1 + 2)
\]

\[
d_{2,4}^{(1)} = \min(\infty, 3)
\]

\[
d_{2,4}^{(1)} = 3
\]

\[
D^{(1)} = \begin{pmatrix}
0 & \infty & 11 & 2 & 6 \\
1 & 0 & 4 & \boxed{3} & \boxed{7} \\
10 & \infty & 0 & \boxed{12} & \boxed{16} \\
\infty & 2 & 6 & 0 & 3 \\
\infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]

The entries that have changed from \(D^{(0)}\) are shown boxed.

\[
D^{(0)} = \begin{pmatrix}
0 & \infty & 11 & 2 & 6 \\
1 & 0 & 4 & \infty & \infty \\
10 & \infty & 0 & \infty & \infty \\
\infty & 2 & 6 & 0 & 3 \\
\infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]
Floyd-Warshall Method

\( D^{(2)} \) with change boxed (only one change):

\[
D^{(2)} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & \infty & 11 & 2 & 6 \\
2 & 1 & 0 & 4 & 3 & 7 \\
3 & 10 & \infty & 0 & 12 & 16 \\
4 & 3 & 2 & 6 & 0 & 3 \\
5 & \infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]

To do this quickly in your “head”, note that since we are interested in location (4, 1) of \( D^{(2)} \), we step over 2 on row 4 of \( D^{(1)} \) and step down 2 of column 1 of \( D^{(1)} \) and add those values together to see if it is less than the current value in \( D^{(2)} \).
Floyd-Warshall Method

D(3) with changes boxed (two changes):

\[
D^{(3)} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & \infty & 11 & 2 & 6 \\
2 & 1 & 0 & 4 & 3 & 7 \\
3 & 10 & \infty & 0 & 12 & 16 \\
4 & 3 & 2 & 6 & 0 & 3 \\
5 & 16 & \infty & 6 & \text{18} & 0
\end{pmatrix}
\]

To update row 5, column 1:

Step down 3

\[
D^{(2)} = \begin{pmatrix}
10 & \infty & 0 & 12 & 16 \\
3 & 2 & 6 & 0 & 3 \\
\infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]

To update row 5, column 4:

Step down 3

\[
D^{(2)} = \begin{pmatrix}
10 & \infty & 0 & \text{12} & 16 \\
3 & 2 & 6 & 0 & 3 \\
\infty & \infty & 6 & \infty & 0
\end{pmatrix}
\]
Floyd-Warshall Method

\[ D^{(4)} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
0 & 4 & 8 & 2 & 5 \\
1 & 0 & 4 & 3 & 6 \\
10 & 14 & 0 & 12 & 15 \\
3 & 2 & 6 & 0 & 3 \\
16 & 20 & 6 & 18 & 0
\end{pmatrix} \]

\[ D^{(5)} = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
0 & 4 & 8 & 2 & 5 \\
1 & 0 & 4 & 3 & 6 \\
10 & 14 & 0 & 12 & 15 \\
3 & 2 & 6 & 0 & 3 \\
16 & 20 & 6 & 18 & 0
\end{pmatrix} \]

No changes from \( D^{(4)} \)
The overall algorithm is simply a matter of running through a loop $N$ times, with each entry being the minimum of the two possibilities. Therefore the overall runtime of the algorithm is proportional to $N^3$. **Works with negative weight edges!**

$$D^{(0)} \leftarrow A$$
for $k \leftarrow 1$ to $N$
  for $i \leftarrow 1$ to $N$
    for $j \leftarrow 1$ to $N$
      $$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

At the end of the procedure we have the matrix $D^{(N)}$ whose $(i,j)$ entry contains the length of the shortest path from $i$ to $j$, all of whose vertices lie in $\{1, 2, \ldots, N\}$. In other words, the shortest path in total.
Floyd-Warshall Method

- The actual paths are found by constructing a sequence of arrays $\Pi^{(k)}$ whose $(i,j)$ entry contains the predecessor of $j$ on the path from $i$ to $j$
- As the entries are updated, the predecessors will change
  - If the matrix entry is not changed then the predecessor entry does not change
  - If the matrix entry does change because the path originally from $i$ to $j$ becomes re-routed through the vertex $k$, then the predecessor entry becomes $k$
\( \Pi_{ij}^{(k)} \) = the predecessor of vertex \( j \) on the shortest path from vertex \( i \) to vertex \( j \) with all intermediate vertices in the set \( \{1, 2, ..., k\} \)

\[ k = 0: \Pi_{ij}^{(0)} = \text{NIL} \quad \text{if} \quad i = j \quad \text{or} \quad w_{ij} = \infty \]
\[ i \quad \text{if} \quad i \neq j \quad \text{and} \quad w_{ij} < \infty \]

\[ k \geq 1: \Pi_{ij}^{(k)} = \Pi_{ij}^{(k-1)} \quad \text{if} \quad d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \]
\[ k \quad \text{if} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \]

**Floyd-Warshall Method**

- **Initialization**
  - \( k = 0: \Pi_{ij}^{(0)} = \text{NIL} \) if \( i = j \) or \( w_{ij} = \infty \)
  - \( i \) if \( i \neq j \) and \( w_{ij} < \infty \)

- **Update**
  - \( k \geq 1: \Pi_{ij}^{(k)} = \Pi_{ij}^{(k-1)} \) if \( d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \)
  - \( k \) if \( d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \)

\( \Pi_{ij} \) is the predecessor of vertex \( j \) on the shortest path from vertex \( i \) to vertex \( j \) with all intermediate vertices in the set \( \{1, 2, ..., k\} \).
To print the shortest path, look across the “source” row $i$ of $\Pi^{(N)}$ until $\Pi_{i,j}^{(N)} = i$

**Example:** Find the shortest path from 1 to 4 (look only at row 1 of $\Pi^{(N)}$):

- $\Pi_{1,4}^{(5)} = 5$
- $\Pi_{1,5}^{(5)} = 1$

Stop because $\Pi_{i,j}^{(N)} = i$

Print: 1,5,4  cost = 2
Transitive Closure of a Graph

The transitive closure problem asks: Is there a path from vertex \( i \) to vertex \( j \) for all vertex pairs \((i, j)\)?

1. Assign a weight of 1 to each edge in the original graph and along the diagonal \((i == j)\), otherwise assign a weight of \( \infty \) if there is no edge
2. Run the Floyd-Warshall algorithm
3. If there is a path from \( i \) to \( j \), then \( d_{ij} < \infty \), if there is no path, then \( d_{ij} = \infty \)
Transitive Closure of a Graph

• Alternatively, to save time and space, perform step 1 as shown in previous slide (using weight of 0 instead of $\infty$ for no edge), but for step 2 substitute logical OR for min and logical AND for +

\[ t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ OR } (t_{ik}^{(k-1)} \text{ AND } t_{kj}^{(k-1)}) \]

• Compute $T^k$ in order of increasing $k$

• This is faster than the standard Floyd-Warshall because logical operations on bits can be performed faster than arithmetic operations on integers
Transitive Closure of a Graph

\[ T(0) = \begin{pmatrix}
1 & 1 & 0 & 0 & 1 \\
2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 \\
4 & 0 & 0 & 1 & 1 \\
\end{pmatrix} \]

\[ T(1) = \begin{pmatrix}
1 & 1 & 0 & 0 & 1 \\
2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 \\
4 & 0 & 1 & 1 & 1 \\
\end{pmatrix} \]

\[ T(2) = \begin{pmatrix}
1 & 1 & 0 & 0 & 1 \\
2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 \\
4 & 0 & 1 & 1 & 1 \\
\end{pmatrix} \]

\[ T(3) = \begin{pmatrix}
1 & 1 & 0 & 0 & 1 \\
2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 \\
4 & 0 & 1 & 1 & 1 \\
\end{pmatrix} \]

\[ T(4) = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 \\
4 & 0 & 1 & 1 & 1 \\
\end{pmatrix} \]

Vertex 1 can get to every other vertex
Edge Weights and Shortest Path

Question: Suppose you have a graph $G'$ with edge weights obtained by adding a constant value to every edge in graph $G$, where $G$ might have negative weights.

When is the shortest path from a source $s$ to a destination $t$ guaranteed to be the same in both graphs?

a. When $G$ has no negatively weighted cycle
b. When all edge weights of $G$ are non-negative
c. When all paths from $s$ to $t$ (all $s$-$t$ paths) in $G$ have the same number of edges
d. Always
e. Never
Johnson’s Algorithm for Sparse Graphs

Idea: Use Bellman-Ford and Dijkstra!

1. Compute G’ (original G with a new “super source” called s).
2. Run Bellman-Ford once to be sure there is no negative weight cycle. If no negative weight cycle, find shortest path from s to each vertex using Bellman-Ford to get the “h” (cost) value for each vertex.
3. Make all edges non-negative using \( \hat{w}(u,v) = w(u,v) + h(u) - h(v) \)
4. Run Dijkstra from each vertex to get \( \delta' \)
5. Convert back to pre-Dijkstra weight using \( \delta(u,v) = \delta'(u,v) - h(u) + h(v) \)

Total runtime is proportional to \( NM + N(M \log N) \)
\( \approx N^2 \log N \) as long as the graph is sparse
Johnson’s Algorithm for Sparse Graphs

- Example: Suppose that for two vertices, \( u \) and \( v \), the shortest path from \( s \) to \( u \) has a cost (total weight) of 0 (\( h(u) = 0 \)), the shortest path from \( s \) to \( v \) has a cost of -3 (\( h(v) = -3 \)), and the weight of edge \((u,v)\) is -2
- Given the original edge weight \( w(u,v) \), the new weight will be \( \hat{w}(u,v) = w(u,v) + h(u) - h(v) \)
- This new weight is guaranteed to be non-negative (why?)

\[ \hat{w}(u,v) = -2 + 0 - (-3) = 1 \]
Johnson’s Algorithm for Sparse Graphs

- Re-weighting using the vertex costs $h(v)$ adds the same amount to every path from source $s$ to destination $t$
- Thus, re-weighting leaves the shortest path unchanged
- After re-weighting, all edge weights will be non-negative, so Dijkstra can be used!
- Question: If an $s$-$t$ path $P$ has total cost $C$ with the original edge weights, what is $P$’s total cost after re-weighting edges?
  a. $C$
  b. $C + h(s) + h(t)$
  c. $C + h(s) - h(t)$
  d. $C - h(s) + h(t)$
Step 1: Create $G'$ by adding a new “super” vertex, $s$, and add an edge with weight 0 from $s$ to each vertex.

Step 2: Compute shortest path $h(v)$ from $s$ to every other vertex using Bellman-Ford.
Johnson’s Algorithm for Sparse Graphs

- **Step 3:** Re-weight each edge using 
  \[ \hat{w}(u,v) = w(u,v) + h(u) - h(v) \]

![Diagram showing re-weighting process](image)
Johnson’s Algorithm for Sparse Graphs

- Step 4: Run Dijkstra from each vertex to get $\delta'$
- Step 5: Convert back to pre-weight distance using $\delta(u,v) = \delta'(u,v) - h(u) + h(v)$

G' with new weights

“Fake” Shortest costs $\delta'(u,v)$

“Real” Shortest costs $\delta(u,v)$
Johnson’s Algorithm for Sparse Graphs

$$\delta(u, v) = \delta'(u, v) - h(u) + h(v)$$
Johnson’s Algorithm for Sparse Graphs

\[ \delta(u,v) = \delta'(u,v) - h(u) + h(v) \]

G’ with new weights  
“Fake” Shortest costs \( \delta'(u,v) \)  
“Real” Shortest costs \( \delta(u,v) \)
Johnson’s Algorithm for Sparse Graphs

\[ \delta(u, v) = \delta'(u, v) - h(u) + h(v) \]
Johnson’s Algorithm for Sparse Graphs

Overall runtime is proportional to \( N^2 \log N \) for sparse graphs, which is much better than Floyd-Warshall, as long as \( M \approx N \).

\[
\delta(u, v) = \delta'(u, v) - h(u) + h(v)
\]
# Summary of All-Pairs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Sparse</th>
<th>Dense</th>
<th>Negative Weights?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>$N^2 \log N$</td>
<td>$N^3 \log N$</td>
<td>No</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>$N^3$</td>
<td>$N^4$</td>
<td>Yes</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>$N^3$</td>
<td>$N^3$</td>
<td>Yes</td>
</tr>
<tr>
<td>Johnson</td>
<td>$N^2 \log N$</td>
<td>$N^3 \log N$</td>
<td>Yes</td>
</tr>
</tbody>
</table>