All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted graph G.

We can make n calls to Dijkstra’s algorithm – $O(n(n+m)\log n)$ time.

From A one may reach C faster by reaching by traveling from A to B and then from B to C.

Alternatively ... Use dynamic programming:
The Floyd’s Algorithm

O(n^3) time.

Floyd’s Shortest Path Algorithm

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Shortest path between vertex i and vertex j without passing through any other vertex is weight(edge(i,j)).
  - Let it be $D^0(i,j)$.
- Idea #3: If $D^k(i,j)$ is the shortest path between vertex i and vertex j using vertices numbered 1, 2, ..., k, as intermediate vertices, then $D^k(i,j)$ is the solution to the shortest path problem between vertex i and vertex j.
Floyd’s Shortest Path Algorithm
Assume that we have $D_{k-1}(i,j)$ for all $i$ and $j$. We wish to compute $D_k(i,j)$. What are the possibilities? Choose between
(1) a path through vertex $k$.
Path Length: $D_k(i,j) = D_{k-1}(i,k) + D_{k-1}(k,j)$.
(2) Skip vertex $k$ altogether.
Path Length: $D_k(i,j) = D_{k-1}(i,j)$.

Floyd’s Shortest Path Algorithm

$D^0_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (v_i,v_j) \text{ is an edge} \\ \text{otherwise} & \end{cases}$

$D^k_{i,j} = \min \{D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j} \}$

Example:
Floyd’s Shortest Path Algorithm

Example:

![Graph](image)

\[ D^1 = \begin{bmatrix} 0 & 8 & 1 & \infty \\ 8 & 0 & 4 & 2 \\ 1 & 4 & 0 & 11 \\ \infty & 2 & 1 & 0 \end{bmatrix} \]

Floyd’s Shortest Path Algorithm

Example:

![Graph](image)

\[ D^2 = \begin{bmatrix} 0 & 8 & 1 & 10 \\ 8 & 0 & 4 & 9 \\ 1 & 4 & 0 & 11 \\ 10 & 2 & 1 & 0 \end{bmatrix} \]

Floyd’s Shortest Path Algorithm

Example:

![Graph](image)

\[ D^3 = \begin{bmatrix} 0 & 3 & 1 & 5 \\ 7 & 0 & 4 & 5 \\ 3 & 4 & 0 & 11 \\ 1 & 5 & 1 & 0 \end{bmatrix} \]
Floyd’s Shortest Path Algorithm

Example:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 8 & 3 & 4 \\
3 & 1 & 4 & 2 \\
4 & 1 & 8 & 1 \\
\end{array}
\]

\[
D^4
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 1 & 0 & 1 \\
0 & 2 & 1 & 0 \\
\end{array}
\]

Algorithm

\text{AllPair\(G\) \{assumes vertices 1,\ldots,n\}}

\begin{algorithmic}
  \ForAll{vertex pairs \((i,j)\)}
    \If{\(i=j\)}
      \State \(d^0(i,j) \leftarrow 0\)
    \ElseIf{\((i,j)\) is an edge in \(G\)}
      \State \(d^0(i,j) \leftarrow \text{weight of edge } (i,j)\)
    \Else
      \State \(d^0(i,j) \leftarrow +\infty\)
    \EndIf
  \EndFor

  \For{i \leftarrow 1 \text{ to } n}
    \For{k \leftarrow 1 \text{ to } n}
      \State \(d^4[i,j] \leftarrow \min(d^4[i,k] + d^4[k,j])\)
    \EndFor
  \EndFor

\end{algorithmic}
Floyd’s Shortest Path Algorithm

Observation:
\[ d^k[i,k] = d^{k-1}[i,k] \]
\[ d^k[k,j] = d^{k-1}[k,j] \]

**Algorithm**

```
Algorithm AllPair(G) {assumes vertices 1,...,n}
for all vertex pairs (i,j)
  if i = j
    d[i,j] ← 0
  else if (i,j) is an edge in G
    d[i,j] ← weight of edge (i,j)
  else d[i,j] ← +∞
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      if (d[i,k]+d[k,j] < d[i,j])
        path[i,j] ← vertex k
        d[i,j] ← d[i,k]+d[k,j]
```
Floyd’s Shortest Path Algorithm

- Can be applied to Directed Graphs
- Can accept –ve edges as long as there are no –ve cycles.

Transitive Closure

- Transitive Relation:
  A relation $R$ on a set $A$ is called transitive if and only if for any $a$, $b$, and $c$ in $A$, whenever $<a, b> \in R$, and $<b, c> \in R$, $<a, c> \in R$

The transitive closure provides reachability information about a graph.
Computing the Transitive Closure

- We can perform BFS starting at each vertex – O(n(n+m))
- Alternatively ... Use dynamic programming: Warshall's Algorithm

Floyd-Warshall Transitive Closure

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, ..., k, as intermediate vertices:
  - Uses only vertices numbered 1, ..., k
  - (add this edge if it's not already in)

Warshall's Transitive Closure

Example:

```
1  2
0 1 0 1
0 0 0 0
1 0 1 0
```
Warshall’s Transitive Closure

Example:

\[
\begin{array}{c|c|c|c|c}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 1 \\
4 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Warshall’s Transitive Closure

Example:

\[
\begin{array}{c|c|c|c|c}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 1 \\
4 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Warshall’s Transitive Closure

Example:

\[
\begin{array}{c|c|c|c|c}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 1 \\
4 & 1 & 1 & 0 & 0 \\
\end{array}
\]
Warshall’s Transitive Closure

Example:

\[
\begin{pmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0
\end{pmatrix}
\]

Warshall’s Algorithm

- Warshall’s algorithm numbers the vertices of \( G \) as \( v_1, \ldots, v_n \) and computes a series of graphs \( G_0, \ldots, G_n \)
  - \( G_0 = G \)
  - \( G_k \) has an edge \((v_i, v_j)\) if \( G \) has a path from \( v_i \) to \( v_j \) with intermediate vertices in the set \( \{v_1, \ldots, v_k\} \)
- We have that \( G_n = G^* \)
- In phase \( k \), graph \( G_k \) is computed from \( G_{k-1} \)
- Running time: \( O(n^3) \), assuming \( \text{isAdjacent} \) is \( O(1) \) (e.g., adjacency matrix)

Algorithm FloydWarshall(G)

- \( G \) is a graph
- \( G^* \) is the transitive closure of \( G \)
- \( V \) is the set of vertices of \( G \)
- \( n \) is the number of vertices of \( G \)

Input: \( G \)

Output: \( G^* \)

\[
\begin{align*}
\text{for } & \quad i = 1 \text{ to } n \\
& \text{for } \quad j = 1 \text{ to } n \\
& \text{for } \quad k = 1 \text{ to } n
\end{align*}
\]

if \( G \) has a path from \( v_i \) to \( v_j \) with intermediate vertices in the set \( \{v_1, \ldots, v_k\} \)

\[ G_k \text{.areAdjacent}(v_i, v_j) = \text{true} \]

return \( G_n \)
Directed Graph With –ve weight

DAG-based Algorithm

 Works even with negative-weight edges
 Uses topological order
 Doesn’t use any fancy data structures
 Is much faster than Dijkstra’s algorithm
 Running time: O(n+m)

```plaintext
Algorithm DagDistances(G, s):
  for all v ∈ G.vertices:
    v.parent ← null
    if v = s:
      v.distance ← 0
    else:
      v.distance ← ∞

  Perform a topological sort of the vertices
  for u ← 1 to n:
    in topological order:
      for each e ∈ G.outEdges(u):
        w ← G.opposite(u, e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(w):
          w.distance ← r
          w.parent ← u
```

DAG-based Algorithm

 Works even with negative-weight edges
 Uses topological order
 Doesn’t use any fancy data structures
 Is much faster than Dijkstra’s algorithm
 Running time: O(n+m)