The all-pairs shortest path problem (APSP)

**Input:** a directed graph $G = (V, E)$ with edge weights

**Goal:** find a minimum weight (shortest) path between every pair of vertices in $V$ (sometimes we only want the cost of these paths)

**Solution 1:** run Dijkstra's algorithm $V$ times, once with each $v \in V$ as the source node (requires no negative-weight edges in $E$)

- If $G$ is dense – array implementation of priority $Q$
  - $O(V \cdot V^2) = O(V^3)$ time

- If $G$ is sparse – binary heap implementation of priority $Q$
  - $O(V \cdot ((V + E) \log V)) = O(V^2 \log V + VE \log V)$ time

**Solution 2:** run the Bellman-Ford algorithm $V$ times (negative edge weights allowed), once from each vertex.

- $O(V^2 E)$, which on a dense graph is $O(V^4)$

**Solution 3:** Use an algorithm designed for the APSP problem. (we will see a few)

E.g., Floyd-Warshall Algorithm introduces a *dynamic programming* technique allows negative-weight edges, but no negative-weight cycles uses adjacency matrix representation of $G = (V, E)$

---

**Overview of Dynamic Programming**

- Some recursive divide-and-conquer algorithms are inefficient—they solve the same sub-problems multiple times.
- Dynamic programming is a technique that can be used to cut down on the inefficiency (solve each sub-problem just once).
- Typically, applied to optimization problems

Two approaches to dynamic programming:

**Method 1:** Top-Down Recursive Approach (memo-ization)

- start with recursive divide-and-conquer algorithm
- keep "top-down" approach of original algorithm
- save solutions to sub-problems in a table (can be lots of storage)
- only recurse on a sub-problem if the solution not in table.

**Method 2:** Bottom-Up iterative approach

- start with recursive divide-and-conquer algorithm
- figure out the dependencies between the sub-problems (which solutions are needed for each sub-problem)
- re-write the algorithm so it solves the sub-problems in the correct order (so we won’t have to save as many solutions in the table).

---

**Slow-APSP Algorithm**

**Input:** Adjacency matrix $A$

**Output:** Shortest path matrix $D^{(n)}$

**Observation:** When $G$ contains no negative-weight cycles, all shortest paths consist of at most $n - 1$ edges

**Solution for $D$:**

- Define $D^{(i)}[i, j] = d^{(i)}_{ij}$ as the minimum weight of any path from vertex $i$ to vertex $j$, consisting of at most $k$ edges
- $D^{(0)} = A$, original adjacency matrix (only paths are single edges)
- $D^{(0)}$, the matrix we want to compute
- $D^{(0)}$'s elements are: $d^{(i)}_{ij} = \min(d^{(i-1)}_{ij}, d^{(i-1)}_{ik} + d^{(i-1)}_{kj})$

---

**Idea:**

Find all vertices reachable in two hops, $D^{(2)}$, save the matrix, and use it to find all vertices reachable in three hops, $D^{(3)}$, save the matrix, and use it to find all vertices reachable in four hops, $D^{(4)}$, and so on until we find $D^{(n)}$. This matrix will contain the shortest path between every pair of vertices in the graph.
Computing $D^{(m)}$ from $D^{(m-1)}$

Extend-Shortest Paths ($D$, $A$)
1. $n = \text{rows}[D]$
2. let $D' = (d_{ij}')$ be an n x n matrix
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. \[ d_{ij}' = \infty \]
6. for $k = 1$ to $n$
7. \[ d_{ij}' = \min (d_{ij}', d_{ik} + a_{kj}) \]
8. return $D'$

The above algorithm is very closely related to an algorithm for matrix multiplication.

Matrix multiplication

Matrix-Multiply ($A$, $B$)
1. $n = \text{rows}[A]$
2. let $C = $ be an n x n matrix
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. $c_{ij} = 0$
6. for $k = 1$ to $n$
7. $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8. return $C$

Recall how matrix multiplication works. This algorithm is structurally similar to ESP. Other features they have in common include running time and associativity of operations.

Operation of Slow-APSP Algorithm

Slow-APSP ($A$)
1. $n = \text{rows}[A]$
2. $D^{(1)} = A$
3. for $m = 2$ to $n - 1$
4. $D^{(m)} = \text{Extend-Shortest-Paths}(D^{(m-1)}, A)$
5. return $D^{(n-1)}$

Extend-Shortest Paths ($D$, $A$)
1. $n = \text{rows}[D]$
2. let $D' = (d_{ij}')$ be an n x n matrix
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. \[ d_{ij}' = \infty \]
6. for $k = 1$ to $n$
7. \[ d_{ij}' = \min (d_{ij}', d_{ik} + a_{kj}) \]
8. return $D'$

Operation of Slow-APSP Algorithm

Slow Dynamic-Programming Solution

Slow-APSP ($A$)
1. $n = \text{rows}[A]$
2. $D^{(1)} = A$
3. for $m = 2$ to $n - 1$
4. $D^{(m)} = \text{Extend-Shortest-Paths}(D^{(m-1)}, A)$
5. return $D^{(n-1)}$
Operation of Slow-APSP Algorithm

\[
D^{(1)} = \begin{bmatrix}
0 & 1 & -3 & -4 & \infty \\
3 & 0 & -4 & -1 & 1 \\
7 & 4 & 0 & 5 & \infty \\
8 & 5 & 1 & 0 & 6 \\
2 & -1 & -5 & -2 & 0
\end{bmatrix}
\]

Running Time of Slow-APSP

Lines 3 & 4 of Slow-APSP: \(|V| - 2\) iterations
For each iteration there is a call to Extend-Shortest-Paths
Lines 3 – 7 of ESP: \(|V|^3\) time for triply-nested for loops
Overall running time = \(((|V| - 2) \cdot |V|^3) = O(V^4)\)
Hence, the name...

However, the running time of Slow-APSP can be improved if we remember that we are not interested in all the \(D^{(n-1)}\) matrices: we only need matrix \(D^{(n-1)}\). Can we compute this faster?

Faster-APSP

Recall that, in the absence of negative-weight cycles, \(D^{(m)} = D^{(n-1)}\) for all \(m \geq n - 1\). Therefore, we can compute \(D^{(n-1)}\) with only \([\lg(n-1)]\) calls to ESP by computing the sequence

\[
D^{(1)} = A, \\
D^{(2)} = A \cdot A, \\
D^{(4)} = A^2 \cdot A^2, \\
\vdots \\
D^{([\lg(n-1)] \cdot 2^{[\lg(n-1)]} \cdot 1)} = A^{[\lg(n-1)]} \cdot A^{[\lg(n-1)]} \cdot 1
\]

Since \(2^{[\lg(n-1)]} = n - 1\), the final product is equal to \(D^{(n-1)}\)

The faster APSP algorithm uses the technique of repeated squaring.

Faster Dynamic-Programming Solution

Faster-APSP (A)
1. \(n = \text{rows}[A]\)
2. \(D^{(1)} = A\)
3. \(m = 1\)
4. while \(m < n - 1\) do
5. \(D^{(2m)} = \text{Extend-Shortest-Paths}(D^{(m)}, D^{(m)})\)
6. \(m = 2m\)
5. return \(D^{(m)}\)

Extend-Shortest Paths (D, A)
1. \(n = \text{row}[D]\)
2. let \(D' = (d_{ij}')\) be an \(n \times n\) matrix
3. for \(i = 1\) to \(n\) do
4. \(\text{for } j = 1\) to \(n\) do
5. \(d_{ij}' = \infty\)
6. \(\text{for } k = 1\) to \(n\) do
7. \(d_{ij} = \min(d_{ij}', d_{ik} + a_{kj})\)
8. return \(D'\)

Faster-APSP Algorithm

\[
D^{(1)} = A = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

Operation of Faster-APSP Algorithm

\[
D^{(1)} = A = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

Operation of Faster-APSP Algorithm

\[
D^{(1)} = A = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

Operation of Faster-APSP Algorithm

\[
D^{(1)} = A = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

Operation of Faster-APSP Algorithm

\[
D^{(1)} = A = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & 3 & 8 & -4 & \infty \\
\infty & 0 & 0 & 7 & 1 \\
\infty & 4 & 0 & \infty & \infty \\
\infty & \infty & 0 & 0 & 6 \\
2 & \infty & -5 & \infty & 0
\end{bmatrix}
\]
Running Time of Faster-APSP

Lines 4 – 6 of Faster-APSP: \(\lceil \lg(V-1) \rceil\) iterations
For each iteration there is a call to Extend-Shortest-Paths
Lines 3 – 7 of ESP: \(|V|^3\) time for triply-nested for loops
Overall running time = \(\lceil \lg(V-1) \rceil \cdot |V|^3\) = \(O(|V|^3 \lg V)\)

Floyd-Warshall APSP Algorithm

Idea:
Find all vertices reachable using intermediate nodes in the range 1...1
\((D^{(1)})\), save the matrix, and use it to find all vertices reachable using intermediate vertices in the range 1...2 \((D^{(2)})\), save the matrix, and use it to find all vertices reachable using intermediate vertices in the range 1...3 \((D^{(3)})\), and so on until we find \(D^{(n)}\). This matrix will contain the shortest path between every pair of vertices in the graph.

Floyd-Warshall APSP Algorithm

Input: Adjacency matrix \(A\)
Output: Shortest path matrix \(D^{(n)}\)
Relies on the Optimal Substructure Property:
All sub-paths of a shortest path are shortest paths.
Solution for \(D\):
Define \(D^{(k)}[i, j] = d^{(k)}_{ij}\) as the minimum weight of any path from vertex \(i\) to vertex \(j\), such that all intermediate vertices are in \(\{1, 2, 3, ..., k\}\)
\(D^{(0)} = A\), original adjacency matrix (only paths are single edges)
\(D^{(n)}\) the matrix we want to compute
\(D^{(n)}\)’s elements are: \(D^{(n)}[i, j] = d^{(n)}_{ij} = \min(d^{(n-1)}_{ij}, d^{(n-1)}_{ik} + d^{(n-1)}_{kj})\)
The only intermediate nodes on these paths are in the set of vertices \(\{1, 2, 3, ..., k-1\}\)

Recursive Solution for \(D^{(n)}\)

\(D^{(n)}[i, j] = d^{(n)}_{ij} = \min(d^{(n-1)}_{ij}, d^{(n-1)}_{ik} + d^{(n-1)}_{kj})\)

Floyd-Warshall APSP Algorithm

Use adjacency matrix \(A\) for \(G = (V, E)\):
\(A[i, j] = a_{ij} = \begin{cases} w(i, j) & \text{if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \end{cases}\)

Floyd-Warshall-APSP(A)
1. \(n = \text{rows}[A]\)
2. \(D^{(0)} = A\)
3. for \(k = 1\) to \(n\) do
4. for \(i = 1\) to \(n\) do
5. for \(j = 1\) to \(n\) do
6. \(d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})\)
7. return \(D^{(n)}\)
Operation of FW-APSP Algorithm

FW-APSP Algorithm

FW-APSP Algorithm

FW-APSP Algorithm

FW-APSP Algorithm
## Running Time of FW-APSP

| Lines 3 – 6: $|V|^3$ time for triply-nested for loops |
|-----------------------------------------------|
| **Overall running time** = $\Theta(|V|^3)$ |

Additionally, the code is tight, with no elaborate data structures and so the constant hidden in the $\Theta$-notation is small.