CSE 5311: Design and Analysis of Algorithms

All-Pairs Shortest Path Algorithms

Alex Dillhoff

University of Texas at Arlington

All-Pairs Shortest Paths

TychoLink is a telecommunications company looking to optimize its network for the fastest and most efficient data transfer possible.

The network consists of multiple routers, each connected by various types of links that differ in latency and bandwidth.

Ensure that data packets can travel from any router to any other router in the network using the path that offers the best balance between low latency and high bandwidth.

All-Pairs Shortest Paths

There are three objectives in total:

1. Determine the all-pairs shortest paths across the network, taking into account both latency and bandwidth.
2. Minimize the overall latency for data packet transmission across the network.
3. Maximize the effective bandwidth along the chosen paths to ensure high data transfer rates.

All-Pairs Shortest Paths

One solution is to run the Bellman-Ford algorithm for each router in the network to find the shortest paths to all other routers.

This results in a time complexity of $O(V^2E)$.

If the network is dense, then the number of edges $E = \Theta(V^2)$, which results in a time complexity of $O(V^4)$.

All-Pairs Shortest Paths

There is another solution, the **Floyd-Warshall algorithm**, which can find the shortest paths between all pairs of routers in the network in $O(V^3)$ time.

The algorithm is particularly useful when the network is dense, as it is more efficient than running the Bellman-Ford algorithm for each router.

Problem Representation

Given a network $G = (V, E)$ and a set of weights $W = (w_{ij})$ for each edge $(i, j) \in E$, the goal is to find the shortest path between all pairs of vertices in V.

The graph and weights will be represented as an adjacency matrix with entries w_{ij} for each edge $(i, j) \in E$.

Problem Representation

The weights are defined as

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge } (i,j) & \text{if } i \neq j, (i,j) \in E \\ \infty & \text{if } i \neq j, (i,j) \notin E. \end{cases}$$

The (i,j) entry of the output matrix is $\delta(i,j)$, the shortest-path weight from i to j.

A Naive Solution

A Naive Solution

To construct a dynamic programming solution, we need to establish that the problem has **optimal substructure**.

The shortest path structure was first discussed in the context of the Bellman-Ford algorithm, which is based on the principle of **relaxation**.

Recursive Solution

Step 2 is to state the recursive solution.

Recursive Solution

Step 2 is to state the recursive solution.

Let $l_{ij}^{(r)}$ be the minimum weight of any path $i \rightsquigarrow j$ that contains at most r edges.

For $r = 0$, the cost is either 0 if $i = j$ or ∞ otherwise.

Recursive Solution

For $r = 1$, try all possible predecessors k of j.

$$\begin{aligned} l_{ij}^{(r)} &= \min\left\{ l_{ij}^{(r-1)}, \min\{ l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n \} \right\} \\ &= \min\{ l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n \}. \end{aligned}$$

Recursive Solution

Either the solution comes from a path of length $r-1$ or from a path of length $r-1$ with an additional edge (k, j).

The shortest path weights $\delta(i, j)$ contain at most $n-1$ edges since the shortest path cannot contain a cycle.

This implies that $\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$

Bottom-Up Approach

Starting with a matrix $W = (w_{ij})$, where w_{ij} are the edge weights, the following approach computes a series of matrices $L^{(0)}, L^{(1)}, \ldots, L^{(n-1)}$, where $L^{(r)} = (l_{ij}^{(r)})$.

The final matrix $L^{(n-1)}$ contains the shortest path weights.

Bottom-Up Approach

```
def extend_shortest_paths(L, W):
    n = len(L)
    L_prime = [[float('inf') for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                L_prime[i][j] = min(L_prime[i][j], L[i][k] + W[k][j])
    return L_prime
```

Bottom-Up Approach

This function has the structure of matrix multiplication and has a running time of $O(n^3)$.

Since this must be repeated n times, the total running time is $O(n^4)$.

Bottom-Up Approach

This function has the structure of matrix multiplication and has a running time of $O(n^3)$.

Since this must be repeated n times, the total running time is $O(n^4)$.

The similarity to matrix multiplication is a **key insight** that will lead to a more efficient algorithm.

Bottom-Up Approach

Consider the primary statement in the loop:

$$l_{ij}^{(r)} = \min\{l_{ij}^{(r)}, l_{ik}^{(r-1)} + w_{kj}\}.$$

Bottom-Up Approach

If we swap $\min$ with $+$ and $\cdot$ with $\min$, we can rewrite this as a matrix multiplication:

$$l_{ij}^{(r)} = l_{ij}^{(r)} + l_{ik}^{(r-1)} \cdot w_{kj}.$$

To complete the transformation, we also need to swap the identity of $\min$, which is ∞, with the identity of $+$, which is 0.

Faster APSP

What's the point?

Faster APSP

What's the point?

Matrix multiplication is associative! We can leverage this to **repeatedly square** the matrix W to compute the shortest paths in fewer steps.

Faster APSP

We can get to this result in fewer steps by using **repeated squaring**.

$$
\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \cdot W, \\
L^{(4)} &= W^4 = W^2 \cdot W^2, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}
$$

Faster APSP

In total, we compute $O(\lg n)$ matrices, each of which takes $O(n^3)$ time to compute.

The total running time is $O(n^3 \lg n)$.

Faster APSP

```
def faster_all_pairs_shortest_paths(W):
    n = len(W)
    L = W
    m = 1
    while m < n - 1:
        L = extend_shortest_paths(L, L)
        m *= 2
    return L
```

Faster APSP

Since we know that $L^{(r)} = L^{(n-1)}$ for all $r \geq n-1$, we can stop the loop when $m \geq n-1$.

Faster APSP

Exercise: Run APSP on the following graph and show the resulting matrices at each step.

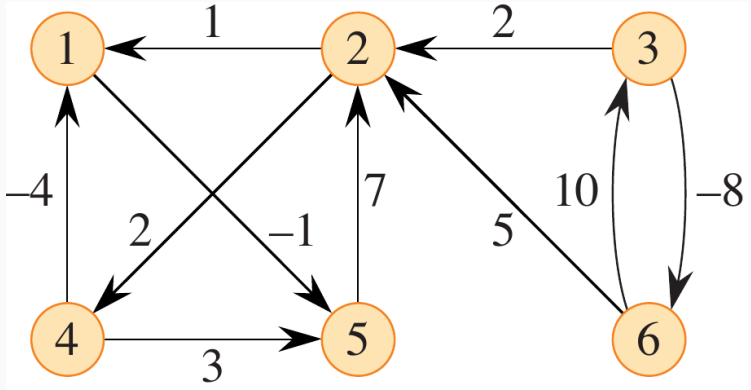

Figure 23.2 from CLRS.

The Floyd-Warshall Algorithm

Floyd-Warshall

Floyd-Warshall is a dynamic programming approach that reconsiders the structure of a shortest path.

It can handle negative weight edges, but it will fail to produce a result if a negative weight cycle exists.

Runs in $O(V^3)$ time.

Floyd-Warshall

Given a path $p = \langle v_1, v_2, \ldots, v_l \rangle$, an **intermediate vertex** is an vertex of p other than v_1 and v_l.

Then define $d_{ij}^{(k)}$ as the weight of the shortest path from i to j that uses only the vertices $\{1, 2, \ldots, k\}$ as intermediate vertices.

The vertices are arbitrarily numbered from 1 to n.

Floyd-Warshall

Consider a shortest path $i \rightsquigarrow j$ that uses intermediate vertices in the set $\{1, 2, \ldots, k\}$:

1. **Drop** k: If k is not an intermediate vertex, then the path is a shortest path from i to j that uses only the vertices $\{1, 2, \ldots, k-1\}$ as intermediate vertices.

Floyd-Warshall

Consider a shortest path $i \rightsquigarrow j$ that uses intermediate vertices in the set $\{1, 2, \ldots, k\}$:

2. **Split** k: If k is an intermediate vertex, split the path p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$.
   - In this case, p_1 is a shortest path from i to k that uses only the vertices $\{1, 2, \ldots, k-1\}$ as intermediate vertices, and p_2 is a shortest path from k to j that uses only the vertices $\{1, 2, \ldots, k-1\}$ as intermediate vertices.

Floyd-Warshall

Notice that, in either case, the set of intermediate vertices is reduced.

Recursive Solution

Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j that uses only the vertices $\{1, 2, \ldots, k\}$ as intermediate vertices.

The base case is $d_{ij}^{(0)} = w_{ij}$.

Recursive Solution

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1. \end{cases}$$

The goal is to compute $D^{(n)} = (d_{ij}^{(n)})$, since all intermediate vertices belong to the set $\{1, 2, \ldots, n\}$.

Bottom-Up Approach

With a recurrent solution in hand, a bottom-up approach can be used to compute the shortest path.

The **Floyd-Warshall** algorithm takes as input a weighted adjacency matrix $W = (w_{ij})$ and returns a matrix $D = (d_{ij})$.

Bottom-Up Approach

```
def floyd_warshall(W):
    n = len(W)
    D = W
    for k in range(n):
        D_prime = [[float('inf') for _ in range(n)] for _ in range(n)]
        for i in range(n):
            for j in range(n):
                D_prime[i][j] = min(D[i][j], D[i][k] + D[k][j])
        D = D_prime
    return D
```

Bottom-Up Approach

Based on the previous code, the runtime is similar to matrix multiplication.

The total running time is $O(n^3)$.

Constructing the Shortest Paths

How can we reconstruct the shortest paths?

Constructing the Shortest Paths

How can we reconstruct the shortest paths?

By maintaining a matrix P that stores the predecessor of each vertex along the shortest path.

Constructing the Shortest Paths

Let $P^{(k)} = (p_{ij}^{(k)})$ for $k = 0, 1, \ldots, n$ be the matrix of predecessors.

Each entry $p_{ij}^{(k)}$ is defined recursively.

Constructing the Shortest Paths

The base case is

$$p_{ij}^{(0)} = \begin{cases} i & \text{if } i \neq j \text{ and } w_{ij} < \infty, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Constructing the Shortest Paths

The recursive case is

$$p_{ij}^{(k)} = \begin{cases} p_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ p_{ij}^{(k-1)} & \text{otherwise.} \end{cases}$$

Constructing the Shortest Paths

The recursive case is split into two parts.

1. If the shortest path from i to j has k as an intermediate vertex, then it is $i \rightsquigarrow k \rightsquigarrow j$ where $k \neq j$.

   - Choose j's predecessor to be the predecessor of j on a shortest path from k to j with all intermediate vertices less than k: $p_{ij}^{(k)} = p_{kj}^{(k-1)}$.

Constructing the Shortest Paths

2. k is not an intermediate vertex. Keep the same predecessor as the shortest path from i to j with all intermediate vertices less than k: $p_{ij}^{(k)} = p_{ij}^{(k-1)}$.

Floyd-Warshall

Exercise: Run Floyd-Warshall on the following graph and show the resulting matrices at each step.

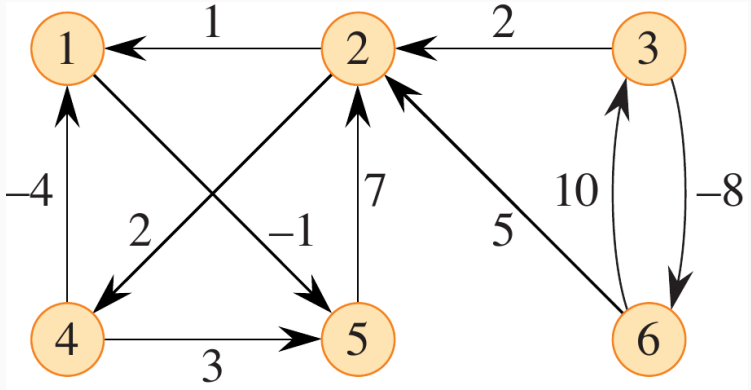

Figure 23.2 from CLRS.

Transitive Closure of a Graph

The algorithms presented above successfully solve the all-pairs shortest paths problem.

What if we simply wanted to determine if a path exists between for all pairs of vertices?

Transitive Closure of a Graph

The algorithms presented above successfully solve the all-pairs shortest paths problem.

What if we simply wanted to determine if a path exists between for all pairs of vertices?

The answer to this question lies in the **transitive closure** of a graph.

Transitive Closure of a Graph

Let $G = (V, E)$ be a directed graph with a vertex set $V = \{1, 2, \ldots, n\}$.

The **transitive closure** of G is a graph $G^* = (V, E^*)$ such that $(i, j) \in E^*$ if there is a path from i to j in G.

Transitive Closure of a Graph

One solution to this problem is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm.

- If $d_{ij} < n$, then there is a path from i to j in G.
- If $d_{ij} = \infty$, then there is no path from i to j in G.

Transitive Closure of a Graph

First, substitute the $\min$ and $+$ operations with $\vee$ (OR) and $\wedge$ (AND), respectively.

This will allow us to determine if a path exists between two vertices.

Transitive Closure of a Graph

Just like Floyd-Warshall, we will maintain a series of matrices $T^{(0)}, T^{(1)}, \ldots, T^{(n-1)}$, where $T^{(r)} = (t_{ij}^{(r)})$.

The final matrix $T^{(n)}$ contains the transitive closure of the graph.

Transitive Closure of a Graph

The values are defined as

$$t_{ij}^{(r)} = \begin{cases} 1 & \text{if } r = 0 \text{ and } (i,j) \in E, \\ 1 & \text{if } r > 0 \text{ and } (t_{ij}^{(r-1)} = 1 \vee (t_{ik}^{(r-1)} \wedge t_{kj}^{(r-1)})), \\ 0 & \text{otherwise.} \end{cases}$$

Transitive Closure of a Graph

```
def transitive_closure(W):
    n = len(W)
    T = W
    for k in range(n):
        T_prime = [[0 for _ in range(n)] for _ in range(n)]
        for i in range(n):
            for j in range(n):
                T_prime[i][j] = T[i][j] or (T[i][k] and T[k][j])
        T = T_prime
    return T
```

Transitive Closure of a Graph

This algorithm has a running time of $\Theta(n^3)$ while using simpler operations compared to the Floyd-Warshall algorithm.