COMS21102: All-Pairs Shortest Paths

Raphaël Clifford
clifford@cs.bris.ac.uk

Bristol University, Department of Computer Science
Bristol BS8 1UB, UK

23rd November 2006
What do we do if want the shortest path from every vertex to every other vertex?

- We could just run a single source shortest path algorithm for every vertex.
- Assuming we use Dijkstra’s shortest path algorithm, the total time would be $O(V^2 \log V + VE)$ using a Fibonacci heap.
- If negative weights are allowed we would use Bellman-Ford giving $O(V^2 E)$ time.
- We can do better.
All Pairs Shortest Paths - Dynamic Programming

- Input: Weighted directed graph $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$
- Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$

Represent the input graph as an $n \times n$ adjacency matrix $W$.

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

Define

$$d_{ij}^m = \text{shortest path from vertex } i \text{ to } j \text{ that uses at most } m \text{ edges}$$

We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$  \hspace{1cm} (1)$$

and for $m = 1, 2, \ldots, n - 1$

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + w_{kj}\}$$
Matrix Multiplication

The final shortest path distances $\delta(i, j)$ will be $d_{ij}^{(n-1)}$. How can we compute them efficiently? Let’s look at matrix multiplication and see how it can help us.

- Compute $C = A \cdot B$, where $C$, $A$, and $B$ are $n \times n$ matrices:

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]

The time required is $O(n^3)$ using a standard algorithm.

- What if we map “+” to “min” and “×” to “+”?

\[
c_{ij} = \min_{k} \{ a_{ik} + b_{kj} \}
\]
Matrix Multiplication

Therefore \( D^{(m)} = D^{(m-1)} \times W \)

To initialise \( D^0 = (d_{ij}^{(0)}) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} \)

- We have that \( D^{(1)} = D^{(0)} \cdot W = W^1 \)
- \( D^{(2)} = D^{(1)} \cdot W = W^2 \)
- \( D^{(3)} = D^{(2)} \cdot W = W^3 \)
- \ldots
- \( D^{(n-1)} = D^{(n-2)} \cdot W = W^{n-1} \)

Remember that the final result we are looking for is \( D^{(n-1)} \). The total time is \( \Theta(n \cdot n^3) = \Theta(n^4) \). This is no better than running Bellman-Ford \( n \) times!
First notice that $A^{n-1} = A^n = A^{n+1} = \ldots$. We can compute $A^n$ by repeated squaring.

- Compute $A^2$, $A^2 \cdot A^2 = A^4$, $A^4 \cdot A^4 = A^8$, \ldots, $A^{2^{\lceil \log(n-1) \rceil}}$
- Total number of “multiplications” is $O(\log n)$
- Therefore the total time is $O(n^3 \log n)$

Remember that $n = V$ so we have a time complexity of $O(V^3 \log V)$. To detect for negatively weighted cycles we check the diagonals for negative values in $O(n)$ additional time.
The Floyd-Warshall algorithm

Can we do any better than $O(V^3 \log V)$? Yes, by an even cleverer application of dynamic programming.

- The **Floyd-Warshall** algorithm takes $O(V^3)$ time and also allows negative weight edges (although not negatively weighted cycles)
- Define $c_{ij}^{(k)} = \text{weight of shortest path from } i \text{ to } j \text{ with all intermediate vertices coming from the set } \{1, \ldots, k\}$
- Therefore $\delta(i, j) = c_{ij}^{(n)}$ and $c_{ij}^{(0)} = w_{ij}$
The Floyd-Warshall recurrence

Consider all paths between vertices $i$ and $j$ whose intermediate vertices are in $\{1, \ldots, k\}$. Let $p$ be a minimum weight path from among them. We observe that

- If $k$ is not an intermediate vertex in $p$ then all intermediate vertices of path $p$ are in the set $\{1, \ldots, k-1\}$. Therefore, a shortest path from vertex $i$ to $j$ with all intermediate vertices in the set $\{1, \ldots, k-1\}$ is also a shortest path from from $i$ to $j$ with all intermediate vertices in $\{1, \ldots, k\}$.

- If $k$ is an intermediate vertex in $p$, then we consider the paths from $i$ to $k$ and $k$ to $j$ separately. Each separate path must be a shortest path with all intermediate vertices in $\{1, \ldots, k-1\}$

By observing the relationship between paths all of whose vertices are in $\{1, \ldots, k-1\}$ and paths all of whose vertices are in $\{1, \ldots, k\}$ we can set up a recurrence that is different from the one given before.
The Floyd-Warshall recurrence

We set up the recurrence as follows

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

This translates immediately to code

\begin{verbatim}
 n ← rows[W];
 D(0) ← W;
 for k ← 1 to n do
   for i ← 1 to n do
     for j ← 1 to n do
       d_{ij}^{(k)} ← \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)});
     end
   end
 end
 return D(n);
\end{verbatim}
The Floyd-Warshall example

On blackboard...
The Floyd-Warshall algorithm

In implementation you can leave off the superscripts

- Floyd-Warshall algorithm runs in \(O(n^3)\) time.
- It is simple to implement.
- Efficient in practice.
- Based on a clever observation that allowed dynamic programming to solve the all-pairs shortest path problem faster than was previously possible.

How can we actually construct the shortest paths? We currently only know the length of the paths, not the actual paths themselves.

- We make the predecessor matrix \(\Pi\)
- \(\Pi = (\pi_{ij})\) where \(\pi_{ij} = \text{NIL}\) if either \(i = j\) or there is no path from \(i\) to \(j\), and otherwise \(\pi_{ij}\) is the predecessor of \(j\) in some shortest path from \(i\) to \(j\)
- From the predecessor matrix we can find the vertices for a given shortest path
F-W application - transitive closure

Compute

\[ t_{ij} = \begin{cases} 
1 & \text{if there exists a path from } i \text{ to } j \\
0 & \text{otherwise} 
\end{cases} \]

We use Floyd-Warshall but with \((\lor, \land)\) instead of \((\min, +)\). So we have

\[ t_{ij}^{(k)} = t_{ij}^{(k-1)} \lor (t_{ik}^{(k-1)} \land t_{kj}^{(k-1)}) \]

instead of

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

Total time is \(\Theta(n^3)\) as before
Johnson’s algorithm for sparse graphs

Johnson’s algorithm solves the all-pairs shortest path problem in $O(V^2 \log V + VE)$ time using a quite different approach to the previous algorithms we have discussed. For sparse graphs (i.e. where $E$ is $o(V^2)$) this is asymptotically the fastest method we have seen.

- The basic trick is to reweight the graph so that all the edge weights are nonnegative
- We can then run Dijkstra’s algorithm, once from each vertex
- Using the fastest implementation of Dijkstra’s algorithm, with the Fibonacci-heap priority queue, this gives us the time complexity of $O(V^2 \log V + VE)$
Any new weight function $\hat{w}$ must satisfy the following criteria

- For all pairs of vertices $u, v \in V$, a path $p$ is a shortest path from $u$ to $v$ using weight function $w$ if and only if $p$ is also a shortest path from $u$ to $v$ using weight function $\hat{w}$.
- All weights $\hat{w}(u, v)$ are non-negative.
Graph reweighting

Theorem

Given a function \( h : V \to \mathbb{R} \), reweight each edge \((u, v) \in E\) by \( w_h(u, v) = w(u, v) + h(u) - h(v) \). Then, for any two vertices, all paths between them are reweighted by the same amount.

Proof.

Let \( p = v_0 \to v_1 \to \ldots \to v_k \) be a path in \( G \). We have

\[
\begin{align*}
    w_h(p) &= \sum_{i=1}^{k} (w_h(v_{i-1}, v_i)) \\
    &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
    &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
    &= w(p) + h(v_0) - h(v_k)
\end{align*}
\]
Corollary

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

- Main idea: Find a function \( h : V \to \mathbb{R} \) such that \( w_h(u, v) \geq 0 \) for all \( (u, v) \in E \). Then, run Dijkstra’s algorithm from each vertex on the reweighted graph.
- \( w_h(u, v) \geq 0 \) if and only if \( h(v) - h(u) \leq w(u, v) \)
Johnson’s algorithm

1. Find a function \( h : V \rightarrow \mathbb{R} \) such that \( w_h(u, v) \geq 0 \) for all \((u, v) \in E\) by using Bellman-Ford to solve the difference constraints \( h(v) - h(u) \leq w(u, v) \), or determine that \( G \) has a negatively weighted cycle

\[
\text{Time} = O(VE)
\]

2. Run Dijkstra’s algorithm from each vertex \( u \in V \) using the weight function \( w_h \) to compute \( \delta_h(u, v) \) for all \( v \in V \)

\[
\text{Time} = O(V^2 \log V + E \log V)
\]

3. For each \((u, v) \in V \times V\) compute \( \delta(u, v) = \delta_h(u, v) - h(u) + h(v) \)

\[
\text{Time} = O(V^2)
\]

Total time is \( O(V^2 \log V + VE) \)
Conclusions

- We have looked at the All-Pairs Shortest Path problem in graphs that may contain edges with negative weights.
- We have seen three main methods.
  1. The first uses matrix multiplication to solve the problem in $O(V^3 \log V)$ time.
  2. Floyd-Warshall uses a cleverer application of dynamic programming to reduce this to $O(V^3)$ time and is simple to implement.
  3. Finally, Johnson’s algorithm uses the solution to a set of difference constraints to reweight all the edges to have positive weights. This allows us to run Dijkstra’s algorithm from each vertex giving a final time complexity of $O(V^2 \log V + VE)$. If the graph is sparse then this is the fastest of the three algorithms we have seen.
Further Reading

- **Introduction to Algorithms**
  - Chapter 25 – All-Pairs Shortest Paths

- **Algorithms**
  S. Dasgupta, C.H. Papadimitriou and U.V. Vazirani
  [http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/](http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/)
  - Chapter 6, Section 6.6 – Brief discussion of Floyd-Warshall