All-pairs shortest paths

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

5 November 2013

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?
- For example, we might want to store these paths in a database for efficient access later.

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?
- For example, we might want to store these paths in a database for efficient access later.
- We could use Dijkstra (if the edge weights are non-negative) or Bellman-Ford, with each vertex in turn as the source, which would achieve complexity $O(VE + V^2 \log V)$ and $O(V^2 E)$ respectively.

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?
- For example, we might want to store these paths in a database for efficient access later.
- We could use Dijkstra (if the edge weights are non-negative) or Bellman-Ford, with each vertex in turn as the source, which would achieve complexity $O(VE + V^2 \log V)$ and $O(V^2 E)$ respectively.
- Can we do better?

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?
- For example, we might want to store these paths in a database for efficient access later.
- We could use Dijkstra (if the edge weights are non-negative) or Bellman-Ford, with each vertex in turn as the source, which would achieve complexity $O(VE + V^2 \log V)$ and $O(V^2 E)$ respectively.
- Can we do better?

Today: algorithms for general graphs with better runtimes than this.

- The Floyd-Warshall algorithm: time $O(V^3)$.
- Johnson's algorithm: time $O(VE + V^2 \log V)$.

All-pairs shortest paths

- We have seen two different ways of determining the shortest path from a vertex s to all other vertices.
- What if we want to determine the shortest paths between all pairs of vertices?
- For example, we might want to store these paths in a database for efficient access later.
- We could use Dijkstra (if the edge weights are non-negative) or Bellman-Ford, with each vertex in turn as the source, which would achieve complexity $O(VE + V^2 \log V)$ and $O(V^2 E)$ respectively.
- Can we do better?

Today: algorithms for general graphs with better runtimes than this.

- The Floyd-Warshall algorithm: time $O(V^3)$.
- Johnson's algorithm: time $O(VE + V^2 \log V)$.

Assume for simplicity that the input graph has no negative-weight cycles.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.
- For a path $p = p_1, \ldots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \ldots, p_{k-1}$.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.
- For a path $p = p_1, \dots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \dots, p_{k-1}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j such that the intermediate vertices are all in the set $\{1, \dots, k\}$.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.
- For a path $p = p_1, \dots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \dots, p_{k-1}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j such that the intermediate vertices are all in the set $\{1, \dots, k\}$.
- If there is no shortest path from i to j of this form, then $d_{ij}^{(k)} = \infty$.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.
- For a path $p = p_1, \dots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \dots, p_{k-1}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j such that the intermediate vertices are all in the set $\{1, \dots, k\}$.
- If there is no shortest path from i to j of this form, then $d_{ij}^{(k)} = \infty$.
- In the case $k = 0$, $d_{ij}^{(0)} = W_{ij}$.

All-pairs shortest paths

- In the Floyd-Warshall algorithm, we assume we are given access to a graph G with n vertices as a $n \times n$ adjacency matrix W. The weights of the edges in G are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the edge } i \to j & \text{if such an edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

- We use the optimal substructure property of shortest paths (the triangle inequality) to write down a dynamic programming recurrence.
- For a path $p = p_1, \dots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \dots, p_{k-1}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j such that the intermediate vertices are all in the set $\{1, \dots, k\}$.
- If there is no shortest path from i to j of this form, then $d_{ij}^{(k)} = \infty$.
- In the case $k = 0$, $d_{ij}^{(0)} = W_{ij}$.
- On the other hand, for $k = n$, $d_{ij}^{(n)} = \delta(i, j)$.

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \dots, k\}$.

Ashley Montanaro
ashley@cs.bris.ac.uk

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \dots, k\}$. Then observe that:

- If k is not an intermediate vertex of p, then p is also a minimum-weight path with all intermediate vertices in the set $\{1, \dots, k-1\}$.

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \ldots, k\}$. Then observe that:

- If k is not an intermediate vertex of p, then p is also a minimum-weight path with all intermediate vertices in the set $\{1, \ldots, k-1\}$.
- If k is an intermediate vertex of p, then we decompose p into a path p_1 between i and k, and a path p_2 between k and j.

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \dots, k\}$. Then observe that:

- If k is not an intermediate vertex of p, then p is also a minimum-weight path with all intermediate vertices in the set $\{1, \dots, k-1\}$.
- If k is an intermediate vertex of p, then we decompose p into a path p_1 between i and k, and a path p_2 between k and j.
- By the triangle inequality, p_1 is a shortest path from i to k. Further, it does not include k (as otherwise it would contain a cycle).

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \dots, k\}$. Then observe that:

- If k is not an intermediate vertex of p, then p is also a minimum-weight path with all intermediate vertices in the set $\{1, \dots, k-1\}$.
- If k is an intermediate vertex of p, then we decompose p into a path p_1 between i and k, and a path p_2 between k and j.
- By the triangle inequality, p_1 is a shortest path from i to k. Further, it does not include k (as otherwise it would contain a cycle).
- The same reasoning shows that p_2 is a shortest path from k to j.

A dynamic-programming recurrence

Let p be a shortest (i.e. minimum-weight) path from i to j with all intermediate vertices in the set $\{1, \ldots, k\}$. Then observe that:

- If k is not an intermediate vertex of p, then p is also a minimum-weight path with all intermediate vertices in the set $\{1, \ldots, k-1\}$.
- If k is an intermediate vertex of p, then we decompose p into a path p_1 between i and k, and a path p_2 between k and j.
- By the triangle inequality, p_1 is a shortest path from i to k. Further, it does not include k (as otherwise it would contain a cycle).
- The same reasoning shows that p_2 is a shortest path from k to j.

We therefore have the following recurrence for $d_{ij}^{(k)}$:

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

The Floyd-Warshall algorithm

Based on the above recurrence, we can give the following bottom-up algorithm for computing $d_{ij}^{(n)}$ for all pairs i, j.

The Floyd-Warshall algorithm

Based on the above recurrence, we can give the following bottom-up algorithm for computing $d_{ij}^{(n)}$ for all pairs i, j.

FloydWarshall(W)

1. $d^{(0)} \leftarrow W$
2. for $k = 1$ to n
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6. return $d^{(n)}$.

The Floyd-Warshall algorithm

Based on the above recurrence, we can give the following bottom-up algorithm for computing $d_{ij}^{(n)}$ for all pairs i, j.

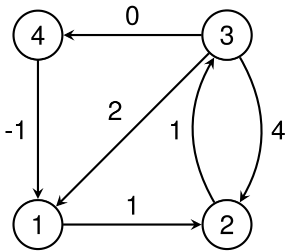
FloydWarshall(W)

1. $d^{(0)} \leftarrow W$
2. for $k = 1$ to n
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6. return $d^{(n)}$.

- The time complexity is clearly $O(n^3)$ and the algorithm is very simple.
- Correctness follows from the argument on the previous slide.

Example

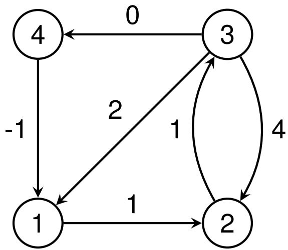
Consider the following graph and its corresponding adjacency matrix:



$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

Example

Consider the following graph and its corresponding adjacency matrix:

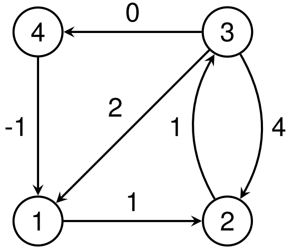


$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(1)} = \begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 3 & 0 & 0 \\ -1 & 0 & \infty & 0 \end{pmatrix}$$

Example

Consider the following graph and its corresponding adjacency matrix:

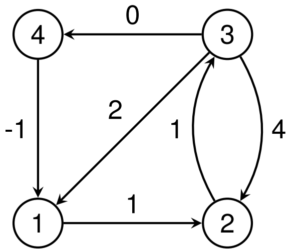


$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(1)} = \begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 3 & 0 & 0 \\ -1 & 0 & \infty & 0 \end{pmatrix}, \quad d^{(2)} = \begin{pmatrix} 0 & 1 & 2 & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 3 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}$$

Example

Consider the following graph and its corresponding adjacency matrix:

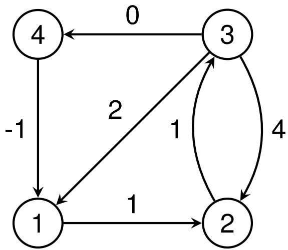


$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(3)} = \begin{pmatrix} 0 & 1 & 2 & {\color{blue}2} \\ {\color{blue}3} & 0 & 1 & {\color{blue}1} \\ 2 & 3 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}$$

Example

Consider the following graph and its corresponding adjacency matrix:



$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(3)} = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 3 & 0 & 1 & 1 \\ 2 & 3 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}, \quad d^{(4)} = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}.$$

Constructing the shortest paths

- We would like to construct a predecessor matrix Π such that Π_{ij} is the predecessor vertex of j in a shortest path from i to j.

Constructing the shortest paths

- We would like to construct a predecessor matrix Π such that Π_{ij} is the predecessor vertex of j in a shortest path from i to j.
- We can do this in a similar way to computing the distance matrix. We define a sequence of matrices $\Pi^{(0)}, \dots, \Pi^{(n)}$ such that $\Pi_{ij}^{(k)}$ is the predecessor of j in a shortest path from i to j only using vertices in the set $\{1, \dots, k\}$.

Constructing the shortest paths

- We would like to construct a predecessor matrix Π such that Π_{ij} is the predecessor vertex of j in a shortest path from i to j.
- We can do this in a similar way to computing the distance matrix. We define a sequence of matrices $\Pi^{(0)}, \dots, \Pi^{(n)}$ such that $\Pi_{ij}^{(k)}$ is the predecessor of j in a shortest path from i to j only using vertices in the set $\{1, \dots, k\}$.
- Then, for $k = 0$,

$$\Pi_{ij}^{(0)} = \begin{cases} \text{nil} & \text{if } i = j \text{ or } W_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } W_{ij} \neq \infty. \end{cases}$$

Constructing the shortest paths

- We would like to construct a predecessor matrix Π such that Π_{ij} is the predecessor vertex of j in a shortest path from i to j.
- We can do this in a similar way to computing the distance matrix. We define a sequence of matrices $\Pi^{(0)}, \dots, \Pi^{(n)}$ such that $\Pi_{ij}^{(k)}$ is the predecessor of j in a shortest path from i to j only using vertices in the set $\{1, \dots, k\}$.
- Then, for $k = 0$,

$$\Pi_{ij}^{(0)} = \begin{cases} \text{nil} & \text{if } i = j \text{ or } W_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } W_{ij} \neq \infty. \end{cases}$$

- For $k \geq 1$, we have essentially the same recurrence as for $d^{(k)}$. Formally,

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

The Floyd-Warshall algorithm with predecessors

FloydWarshall(W)

$d^{(0)} \leftarrow W$
for $k = 1$ to n
    for $i = 1$ to n
        for $j = 1$ to n
            if $d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
                $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$
                $\Pi_{ij}^{(k)} \leftarrow \Pi_{ij}^{(k-1)}$
            else
                $d_{ij}^{(k)} \leftarrow d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
                $\Pi_{ij}^{(k)} \leftarrow \Pi_{kj}^{(k-1)}$
return $d^{(n)}$.

Ashley Montanaro
ashley@cs.bris.ac.uk

University of BRISTOL

Johnson's algorithm

- For sparse graphs with non-negative weight edges, running Dijkstra with each vertex in turn as the source is more efficient than the Floyd-Warshall algorithm.

Johnson's algorithm

- For sparse graphs with non-negative weight edges, running Dijkstra with each vertex in turn as the source is more efficient than the Floyd-Warshall algorithm.
- Johnson's algorithm uses Dijkstra's algorithm to solve the all-pairs shortest paths problem for graphs which may have negative-weight edges. It is based around the idea of first reweighting G so that all the weights are non-negative, then using Dijkstra.

Johnson's algorithm

- For sparse graphs with non-negative weight edges, running Dijkstra with each vertex in turn as the source is more efficient than the Floyd-Warshall algorithm.

- Johnson's algorithm uses Dijkstra's algorithm to solve the all-pairs shortest paths problem for graphs which may have negative-weight edges. It is based around the idea of first reweighting G so that all the weights are non-negative, then using Dijkstra.

- For sparse graphs, its complexity $O(VE + V^2 \log V)$ (the same as Dijkstra) is faster than the Floyd-Warshall algorithm.

Johnson's algorithm

- For sparse graphs with non-negative weight edges, running Dijkstra with each vertex in turn as the source is more efficient than the Floyd-Warshall algorithm.
- Johnson's algorithm uses Dijkstra's algorithm to solve the all-pairs shortest paths problem for graphs which may have negative-weight edges. It is based around the idea of first reweighting G so that all the weights are non-negative, then using Dijkstra.
- For sparse graphs, its complexity $O(VE + V^2 \log V)$ (the same as Dijkstra) is faster than the Floyd-Warshall algorithm.
- We assume that we are given G as an adjacency list, and have access to a weight function $w(u, v)$ which tells us the weight of the edge $u \to v$.

Claim

For any edge $u \to v$, define

$$\widehat{w}(u,v) := w(u,v) + h(u) - h(v),$$

where h is an arbitrary function mapping vertices to real numbers. Then any path $p = v_0, \dots, v_k$ is a shortest path from v_0 to v_k with respect to the weight function $\widehat{w}$ if and only if it is a shortest path with respect to the weight function w.

Claim

For any edge $u \to v$, define

$$\widehat{w}(u,v) := w(u,v) + h(u) - h(v),$$

where h is an arbitrary function mapping vertices to real numbers. Then any path $p = v_0, \dots, v_k$ is a shortest path from v_0 to v_k with respect to the weight function $\widehat{w}$ if and only if it is a shortest path with respect to the weight function w.

Proof

The total weights of p under $\widehat{w}$ and w are closely related:

$$\sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \quad = \quad \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)$$

Claim

For any edge $u \to v$, define

$$\widehat{w}(u,v) := w(u,v) + h(u) - h(v),$$

where h is an arbitrary function mapping vertices to real numbers. Then any path $p = v_0, \dots, v_k$ is a shortest path from v_0 to v_k with respect to the weight function $\widehat{w}$ if and only if it is a shortest path with respect to the weight function w.

Proof

The total weights of p under $\widehat{w}$ and w are closely related:

$$\begin{aligned} \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\ &= h(v_0) - h(v_k) + \sum_{i=1}^{k} w(v_{i-1}, v_i) \qquad \dots \end{aligned}$$

Claim

For any edge $u \leftarrow v$, define

$$\widehat{w}(u, v) := w(u, v) + h(u) - h(v),$$

where h is an arbitrary function mapping vertices to real numbers. Then any path $p = v_0, \dots, v_k$ is a shortest path from v_0 to v_k with respect to the weight function $\widehat{w}$ if and only if it is a shortest path with respect to the weight function w.

Proof

- So the weight of p under $\widehat{w}$ only differs from its weight under w by an additive term which does not depend on p.

Claim

For any edge $u \leftarrow v$, define

$$\widehat{w}(u,v) := w(u,v) + h(u) - h(v),$$

where h is an arbitrary function mapping vertices to real numbers. Then any path $p = v_0, \dots, v_k$ is a shortest path from v_0 to v_k with respect to the weight function $\widehat{w}$ if and only if it is a shortest path with respect to the weight function w.

Proof

- So the weight of p under $\widehat{w}$ only differs from its weight under w by an additive term which does not depend on p.
- So p is a shortest path with respect to $\widehat{w}$ if and only if it is a shortest path with respect to w.

□

Negative-weight cycles

Claim

A graph has a negative-weight cycle under weight function $\widehat{w}$ if and only if if has one under weight function w.

Proof

- Let $c = v_0, \dots, v_k$, where $v_0 = v_k$, be any cycle.

Negative-weight cycles

Claim

A graph has a negative-weight cycle under weight function $\widehat{w}$ if and only if if has one under weight function w.

Proof

- Let $c = v_0, \ldots, v_k$, where $v_0 = v_k$, be any cycle.
- As $v_0 = v_k$, $h(v_0) = h(v_k)$, so the weight of c under $\widehat{w}$ is the same as its weight under w.

Negative-weight cycles

Claim

A graph has a negative-weight cycle under weight function $\widehat{w}$ if and only if if has one under weight function w.

Proof

- Let $c = v_0, \ldots, v_k$, where $v_0 = v_k$, be any cycle.
- As $v_0 = v_k$, $h(v_0) = h(v_k)$, so the weight of c under $\widehat{w}$ is the same as its weight under w.
- So c is negative-weight under $\widehat{w}$ if and only if it is negative-weight under w.

$\square$

Reweighting

- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.

Reweighting

- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.
- This cannot create a new negative-weight cycle if there was not one there already.

Reweighting

- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.
- This cannot create a new negative-weight cycle if there was not one there already.
- We then define $h(v) = \delta(s, v)$ for all vertices v in G.

Reweighting

- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.
- This cannot create a new negative-weight cycle if there was not one there already.
- We then define $h(v) = \delta(s, v)$ for all vertices v in G.
- Now observe that $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for all edges $u \to v$ by the triangle inequality, so $h(v) - h(u) \leq w(u, v)$.

Reweighting

- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.
- This cannot create a new negative-weight cycle if there was not one there already.
- We then define $h(v) = \delta(s, v)$ for all vertices v in G.
- Now observe that $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for all edges $u \to v$ by the triangle inequality, so $h(v) - h(u) \leq w(u, v)$.
- So, if we reweight according to the function h,

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$$

for all edges $u \to v$.

Reweighting

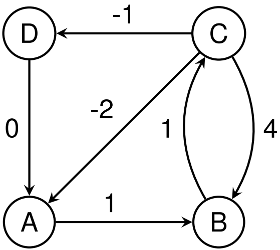
- Given a graph G, to define our new weight function, we add a new vertex s which has an edge of weight 0 to all other vertices in G.
- This cannot create a new negative-weight cycle if there was not one there already.
- We then define $h(v) = \delta(s, v)$ for all vertices v in G.
- Now observe that $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for all edges $u \to v$ by the triangle inequality, so $h(v) - h(u) \leq w(u, v)$.
- So, if we reweight according to the function h,

  $$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$$

  for all edges $u \to v$.
- Then, if $\widehat{\delta}(u, v)$ is the weight of a shortest path from u to v with weight function $\widehat{w}$, $\delta(u, v) = \widehat{\delta}(u, v) + h(v) - h(u)$.

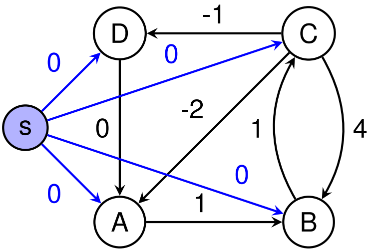
Example

Imagine we want to reweight the following graph:



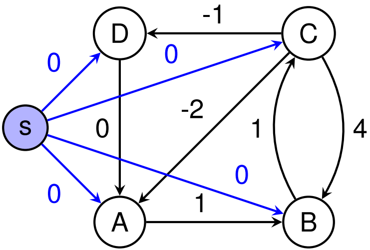
Example

Imagine we want to reweight the following graph:



Example

Imagine we want to reweight the following graph:

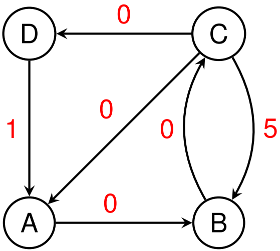


- Using Bellman-Ford, we compute

$$h(A) = -2, \quad h(B) = -1, \quad h(C) = 0, \quad h(D) = -1.$$

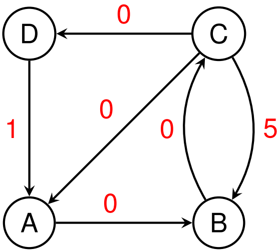
Example

Reweighting according to *h* gives the following graph:



Example

Reweighting according to *h* gives the following graph:



- For each pair of vertices u, v, $\delta(u,v) = \widehat{\delta}(u,v) + h(v) - h(u)$.
- For example, $\delta(C,A) = 0 - 2 - 0 = -2$ as expected.

Johnson's algorithm

From the above discussion, we can write down the following algorithm.

Johnson(G)

1. form a new graph G' by adding s to G, as defined above

Johnson's algorithm

From the above discussion, we can write down the following algorithm.

Johnson(G)

1. form a new graph G' by adding s to G, as defined above
2. compute $\delta(s, v)$ for all $v \in G$ using BellmanFord

Johnson's algorithm

From the above discussion, we can write down the following algorithm.

Johnson(G)

1. form a new graph G' by adding s to G, as defined above
2. compute $\delta(s, v)$ for all $v \in G$ using BellmanFord
3. for each edge $u \to v$ in G
4. $\qquad \widehat{w}(u, v) \leftarrow w(u, v) + \delta(s, u) - \delta(s, v)$

Johnson's algorithm

From the above discussion, we can write down the following algorithm.

Johnson(G)

1. form a new graph G' by adding s to G, as defined above
2. compute $\delta(s, v)$ for all $v \in G$ using BellmanFord
3. for each edge $u \to v$ in G
4. $\quad\quad \widehat{w}(u, v) \leftarrow w(u, v) + \delta(s, u) - \delta(s, v)$
5. for each vertex $u \in G$
6. $\quad\quad$ compute $\widehat{\delta}(u, v)$ for all v using Dijkstra
7. $\quad\quad$ for each vertex $v \in G$
8. $\quad\quad\quad\quad d_{uv} \leftarrow \widehat{\delta}(u, v) + \delta(s, v) - \delta(s, u)$
9. return d

Summary of all-pairs shortest paths algorithms

We have now seen two different algorithms for this problem.

- Both algorithms work for graphs which may have negative-weight edges.

Summary of all-pairs shortest paths algorithms

We have now seen two different algorithms for this problem.

- Both algorithms work for graphs which may have negative-weight edges.
- The Floyd-Warshall algorithm runs in time $O(V^3)$ and is based on ideas from dynamic programming.

Summary of all-pairs shortest paths algorithms

We have now seen two different algorithms for this problem.

- Both algorithms work for graphs which may have negative-weight edges.
- The Floyd-Warshall algorithm runs in time $O(V^3)$ and is based on ideas from dynamic programming.
- Johnson's algorithm is based on reweighting edges in the graph and running Dijkstra's algorithm.

Summary of all-pairs shortest paths algorithms

We have now seen two different algorithms for this problem.

- Both algorithms work for graphs which may have negative-weight edges.
- The Floyd-Warshall algorithm runs in time $O(V^3)$ and is based on ideas from dynamic programming.
- Johnson's algorithm is based on reweighting edges in the graph and running Dijkstra's algorithm.
- The runtime of Johnson's algorithm is dominated by the complexity of running Dijkstra's algorithm once for each vertex, which is $O(VE + V^2 \log V)$ if implemented using a Fibonacci heap, and $O(VE \log V)$ if implemented using a binary heap.

Summary of all-pairs shortest paths algorithms

We have now seen two different algorithms for this problem.

- Both algorithms work for graphs which may have negative-weight edges.
- The Floyd-Warshall algorithm runs in time $O(V^3)$ and is based on ideas from dynamic programming.
- Johnson's algorithm is based on reweighting edges in the graph and running Dijkstra's algorithm.
- The runtime of Johnson's algorithm is dominated by the complexity of running Dijkstra's algorithm once for each vertex, which is $O(VE + V^2 \log V)$ if implemented using a Fibonacci heap, and $O(VE \log V)$ if implemented using a binary heap.
- This can be significantly smaller than the runtime of the Floyd-Warshall algorithm if the input graph is sparse.

Shortest path algorithms: the summary

To compute single-source shortest paths in a directed graph G which is...

- unweighted: use breadth-first search in time $O(V + E)$;
- weighted with non-negative weights: use Dijkstra's algorithm in time $O(E + V \log V)$;
- weighted with negative weights: use Bellman-Ford in time $O(VE)$.

Shortest path algorithms: the summary

To compute single-source shortest paths in a directed graph G which is...

- unweighted: use breadth-first search in time $O(V + E)$;
- weighted with non-negative weights: use Dijkstra's algorithm in time $O(E + V \log V)$;
- weighted with negative weights: use Bellman-Ford in time $O(VE)$.

To compute all-pairs shortest paths in a directed graph G which is...

- unweighted: use breadth-first search from each vertex in time $O(VE + V^2)$;
- weighted with non-negative weights: use Dijkstra's algorithm from each vertex in time $O(VE + V^2 \log V)$;
- weighted with negative weights: use Johnson's algorithm in time $O(VE + V^2 \log V)$.

Further Reading

- Introduction to Algorithms
  T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Chapter 25 – All-Pairs Shortest Paths

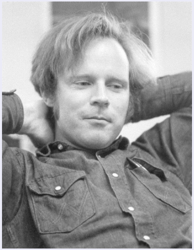
- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`
  - Lecture 20 – All-pairs shortest paths

Biographical notes

The Floyd-Warshall algorithm was invented independently by Floyd and Warshall (and also Bernard Roy).

Robert W. Floyd (1936–2001)

- American computer scientist who did major work on compilers and initiated the field of programming language semantics.
- He completed his first degree (in liberal arts) at the age of 17 and won the Turing Award in 1978.
- Had his middle name legally changed to "W".



Pic: IEEE

Biographical notes

Stephen Warshall (1935–2006)

- Another American computer scientist whose other work included operating systems and compiler design.
- Supposedly he and a colleague bet a bottle of rum on who could first prove correctness of his algorithm.
- Warshall found his proof overnight and won the bet (and the rum).

Donald B. Johnson (d. 1994)

- Yet another American computer scientist. Founded the computer science department at Dartmouth College and invented the *d*-ary heap.