Efficient Algorithms

Ekkapot Charoenwanit

Electrical and Computer Engineering (ECE)
TGGS
KMUTNB

Lecture 11: Graph Algorithms (Part III)

All-Pair Shortest Paths (APSP)

Shortest Path Problem

In a shortest path problem, we are given a ***weighted***, ***directed*** graph $G = (V, E, w)$ with a weight function $w: E \rightarrow \mathbb{R}$ that maps edges to ***real-valued*** weights.

The weight $w(p)$ of a path $p = < v_0, v_1, v_2, \dots, v_k >$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Shortest Path Problem

We define the ***shortest path weight*** from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p): u \rightsquigarrow v\} \\ \infty \end{cases}$$ **(Eq.1)**

A ***shortest path*** from u to v is then defined as ***any path*** p with weight $w(p) = \delta(u,v)$.

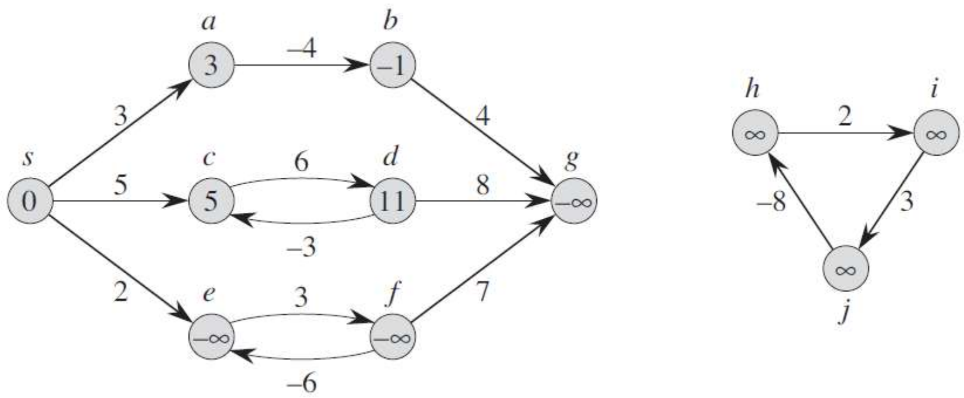
If there is no path from u to v, $\delta(u,v) = \infty$.

Even though there is a path u to v, a shortest path may not exist in the presence of ***a negative-weight cycle*** reachable from u.

Negative-Weight Cycle

Even though there is a path u to v, a shortest path may not exist in the presence of at least one ***negative-weight cycle*** reachable from u. Thus, $\delta(u, v) = -\infty$.

In the example below, a shortest path from s to f is ***undefined*** because we can always find a path with a smaller weight by traversing the negative-weight cycle $< e, f, e >$ as many times as we want before reaching f.



All-Pair Shortest Paths: Ad-hoc Solutions

If $G = (V, E)$ contains ***no negative-weight edges***, we can run ***Dijkstra's Algorithm*** for each $v \in V$ as ***source vertex***.

- The algorithm runs in $O((V + E)\log V) \cdot |V| = O((V^2 + VE)\log V)$ time.
- If G is ***dense***, the running time is $O(V^3 \log V)$.

If $G = (V, E)$ contains ***negative-weight edges,*** we can run **Bellman-Ford** for each $v \in V$ as ***source vertex***.

- The algorithm runs in $O(VE) \cdot |V| = O(V^2E)$ time.
- If G is ***dense***, the running time is $O(V^4)$.

We will see that we can achieve a ***better time complexity*** than that of these two ad-hoc solutions.

DP Solution

Assuming that a weighted, directed graph $G = (V, E, w)$ is represented by an ***adjacency matrix*** $W = (w_{ij})$, consider a ***shortest path*** p from a vertex i to a vertex j.

Suppose that p contains ***at most*** m edges, and if there are ***no negative-weight cycles*** on p, m is ***finite***.

If $i = j$, $w(p) = 0$ and p contains ***no edges***.

If $i \neq j$, we can break p into $i \leadsto k \rightarrow j$, where $p' = i \leadsto k$, and p' contains ***at most*** $m - 1$ edges.

- Moreover, p' is a shortest path from i to k by the ***Optimal Substructure Lemma*** (See ***Lecture 10***).

DP Solution: Recurrence Formulation

Let ${l_{ij}}^{(m)}$ be the ***minimum weight*** of any path from vertex i to vertex j that contains ***at most*** m edges.

Base Case: when $m = 0$, we have $\quad {l_{ij}}^{(0)} = \begin{cases} 0 & if\ i = j \\ \infty & if\ i \neq j \end{cases}$

That is, there is only a shortest path if and only if $i = j$.

Recursive Case: we ***exhaustively*** search for the minimum among all possible predecessors k of j.

$$
\begin{aligned}
{l_{ij}}^{(m)} &= \min({l_{ij}}^{(m-1)}, \min_{1 \le k \le n}({l_{ik}}^{(m-1)} + w(k,j))) \\
&= \min_{1 \le k \le n}({l_{ik}}^{(m-1)} + w(k,j)) \quad [w_{jj} = 0 \text{ for all } j] \qquad \textit{(Eq.2)}
\end{aligned}
$$

DP Solution

Observation: If G contains ***no negative-weight cycles***, then for every pair of vertices i and j for which $\delta(i,j) < \infty$, there is a ***shortest path*** from i to j that is ***simple*** with ***at most*** $n-1$ edges.

This ***observation*** implies the following equalities:

$$\delta(i,j) = l_{ij}{}^{(n-1)} = l_{ij}{}^{(n)} = l_{ij}{}^{(n+1)} = \cdots$$

DP Solution: Bottom-Up Solution

Taking as input the adjacency matrix $W = (w_{ij})$, we compute a series of matrices $L^{(1)}$, $L^{(2)}$, $L^{(3)}$,…, $L^{(n-1)}$, where $L^{(m)} = (l_{ij}{}^{(m)})$ for $i = 1,2,3 \ldots, n-1$.

The ***final matrix*** $L^{(n-1)}$ contains the ***actual shortest path weights*** for every pair of vertices.

In essence, the algorithm on the right, given $L^{(m-1)}$and W as input, produces $L^{(m)}$ as output, effectively extending the shortest paths computed so far by ***one more edge:***

$L^{(0)}$ and W and produces $L^{(1)}$,

$L^{(1)}$ and W and produces $L^{(2)}$,

….

$L^{(n-2)}$ and W and produces $L^{(n-1)}$

EXTEND-SHORTEST-PATHS(L, W)

$n = L.rows$

let $L' = (l'_{ij})$ be a new $n \times n$ matrix

for $i = 1$ **to** n

  for $j = 1$ **to** n

    $l'_{ij} = \infty$

    for $k = 1$ **to** n

      $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$

return L'

DP Solution: Bottom-Up Solution

Since the algorithm consists of ***triply nested*** for loops, each of which executes for exactly n times, the running time is $\Theta(n^3)$.

EXTEND-SHORTEST-PATHS(L, W)

```
n = L.rows
let L′ = (l′_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        l′_ij = ∞
        for k = 1 to n
            l′_ij = min(l′_ij, l_ik + w_kj)
return L′
```

DP Solution: Matrix Multiplication View

We will show that the DP solution we have just arrived at has a close relation to ***matrix multiplication***.

Recall that, to compute $C = A \cdot B$ of two $n \times n$ matrices, we compute the following:

$c_{ij} = \sum_{i=1}^{n} a_{ik} \cdot b_{kj}$ for $i, j = 1,2, \dots, n$ ***(Eq.3)***

Observe that if we make the following ***symbolic substitutions in Eq.2***, we obtain ***Eq.3***.

$$
\begin{aligned}
l^{(m-1)} &\rightarrow a \\
w &\rightarrow b \\
l^{(m)} &\rightarrow c \\
min &\rightarrow + \\
+ &\rightarrow \cdot
\end{aligned}
$$

DP Solution: Matrix Multiplication View

Thus, if we apply these changes to $Extend - Shortest - Paths$ and replace ∞ with 0, we obtain the following matrix multiplication algorithm we are familiar with. (See ***Lecture 6***)

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

DP Solution: Matrix Multiplication View

Let's return to the ***APSP*** Problem.

Recall that we compute the shortest-path weights by extending shortest paths ***edge by edge***.

Letting $A \cdot B$ denote the matrix "***product***" returned by $Extend - Shortest - Paths(A, B)$, we can compute the sequence of $n - 1$ matrices as follows:

$$L^{(1)} = L^{(0)} \cdot W = W$$
$$L^{(2)} = L^{(1)} \cdot W = W^2$$
$$L^{(3)} = L^{(2)} \cdot W = W^3$$

$$L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$$

Recall that the ***final matrix*** $L^{(n-1)}$ contains the ***actual shortest path weights*** for every pair of vertices.

DP Solution: Matrix Multiplication View

We arrive at the following ***SLOW*** algorithm to compute all-pair shortest paths.

The algorithm starts by initializing $L^{(1)} = W$ and extends the shortest paths computed so far ***by one edge at a time***.

- $L^{(n-1)}$ returned by the algorithm contains the actual shortest paths.

The algorithm runs in $\Theta(n^4)$ time because there are $n - 2 = \Theta(n)$ iterations, each of which runs in $\Theta(n^3)$ time.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
for m = 2 to n - 1
    let L^(m) be a new n × n matrix
    L^(m) = EXTEND-SHORTEST-PATHS(L^(m-1), W)
return L^(n-1)
```

DP Solution: Matrix Multiplication View

We can apply ***the repeated squaring technique*** to reduce the number of matrix multiplications to $\lceil \log_2(n-1) \rceil$ as follows:

$$L^{(1)} = W$$

$$L^{(2)} = W^2 = W \cdot W$$

$$L^{(4)} = W^4 = W^2 \cdot W^2$$

$$L^{(8)} = W^8 = W^4 \cdot W^4$$

$$L^{(2^{\lceil \log_2(n-1) \rceil})} = W^{2^{\lceil \log_2(n-1) \rceil}} = W^{2^{\lceil \log_2(n-1) \rceil - 1}} \cdot W^{2^{\lceil \log_2(n-1) \rceil - 1}}$$

Since $2^{\lceil \log_2(n-1) \rceil} \geq n-1$, it is always guaranteed that

$$L^{(2^{\lceil \log_2(n-1) \rceil})} = L^{(n-1)}$$

DP Solution: Matrix Multiplication View

The following ***faster*** algorithm can achieve a better time complexity of $\Theta(n^3 \log n)$ using the repeated squaring technique:

- There are $\lceil \log_2(n-1) \rceil$ matrix multiplications, each of which takes $\Theta(n^3)$ time.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
m = 1
while m < n - 1
    let L^(2m) be a new n × n matrix
    L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
    m = 2m
return L^(m)
```

The Floyd-Warshall Algorithm

In the ***Floyd-Warshall*** algorithm, a different way of characterizing the structure of shortest paths is used.

The ***Floyd-Warshall*** algorithm considers the ***intermediate vertices*** of a shortest path, where an intermediate vertex of a ***simple path*** $p = < v_1, v_2, \ldots, v_l >$ is any vertex of p other than v_1 or v_l.

- That is, it can be any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The Floyd-Warshall Algorithm

For any pair of vertices $i, j \in V$, consider all paths from i to j with ***all intermediate vertices*** drawn from the set $\{1,2,\ldots,k\}$ and let p be a ***minimum weight-path*** among all of them.

Case I: If k is ***not*** an intermediate vertex of path p, then all intermediate vertices of p are in the set $\{1,2,\ldots,k-1\}$.

- A shortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k\}$.

Case II: If k is an intermediate vertex of path p, then we decompose p into $i \rightsquigarrow k \rightsquigarrow$j: $p_1 = i \rightsquigarrow k$ and $p_2 = k \rightsquigarrow j$.

- By the ***Optimal Substructure Property***, p_1 is a shortest path from i to k with all the intermediate vertices in the set $\{1,2,\ldots,k\}$.
- In fact, we can make ***a stronger statement***, because k is not an intermediate vertex of p_1, all intermediate vertices of p_1 are in the set $\{1,2,\ldots,k-1\}$.
- Therefore, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$.
- Similarly, p_2 is a shortest path from k to j with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$.

The Floyd-Warshall Algorithm

Let ${d_{ij}}^{(k)}$ be the weight of a shortest path from i to j for which all the intermediate vertices are in the set $\{1,2,\ldots,k\}$.

${d_{ij}}^{(0)}$ is a shortest path from i to j with no intermediate vertices at all so such a path contains ***at most one edge***.

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

(Eq.4)

Because, for any path, all intermediate vertices are in the set $\{1,2,\ldots,n\}$, the matrix $D^{(n)} = ({d_{ij}}^{(n)})$ is the ***final solution*** where $\delta(i,j) = {d_{ij}}^{(n)}$ for all $i,j \in V$.

The Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm consists of ***three nested for loops***, each of which executes ***exactly*** n iterations.

Because each execution of ***line 7*** takes $\Theta(1)$ time, the total running time is $n^3 \cdot \Theta(1) = \Theta(n^3)$.

FLOYD-WARSHALL(W)

1 $n = W.rows$
2 $D^{(0)} = W$
3 **for** $k = 1$ **to** n
4   let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
5   **for** $i = 1$ **to** n
6   **for** $j = 1$ **to** n
7             $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8 **return** $D^{(n)}$

Transitive Closure of a Directed Graph

Given a ***weighted***, ***directed*** graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, we can determine whether G contains a path from i to j for all pair of vertices $i, j \in V$.

We define the ***transitive closure*** of G as $G^* = (V, E^*)$, where $E^* = \{(i, j): \text{there is a path from } i \text{ to } j \text{ in } G\}$.

The most obvious way to compute the ***transitive closure*** of a graph is to assign a weight of 1 to each edge of E and run ***the Floyd-Warshall algorithm*** on the graph.

- If there is a path from i *to* j, $d_{ij} < n$.
- Otherwise, $d_{ij} = \infty$.
- The algorithm takes $\Theta(n^3)$.

Transitive Closure of a Directed Graph

Another method for computing the ***transitive closure*** is to rely on the ***matrix multiplication view***:

$$min \quad \rightarrow \quad \vee$$

$$+ \quad \rightarrow \quad \wedge$$

Let ${t_{ij}}^{(k)}$ to be

- 1 if there is a path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k\}$
- 0 otherwise

Thus, we can formulate a recurrence solution as follows:

Base Case:

$${t_{ij}}^{(0)} = \begin{cases} 0 & if\ i \neq j\ \wedge (i,j) \notin E \\ 1 & if\ i = j\ \vee\ (i,j) \in E \end{cases} \qquad \textbf{(Eq.5)}$$

Recursive Case:

$${t_{ij}}^{(k)} = {t_{ij}}^{(k-1)} \vee ({t_{ik}}^{(k-1)} \wedge\ {t_{kj}}^{(k-1)}) \qquad for\ k \geq 1 \qquad \textbf{(Eq.6)}$$

Transitive Closure of a Directed Graph

As in the Floyd-Warshall algorithm, we compute the sequence of matrices $T^{(k)} = ({t_{ij}}^{(k)})$ for $k = 1,2, \ldots n - 1$.

The algorithm takes $\Theta(n^3)$ time.

- There are ***three nested for loops***, each of which executes ***exactly*** n iterations.
- Therefore, there are n^3 iterations in total.
- Each iteration takes $\Theta(1)$ time (***line 12***)

```
TRANSITIVE-CLOSURE(G)
1  n = |G.V|
2  let T^(0) = (t_ij^(0)) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          if i == j or (i, j) ∈ G.E
6              t_ij^(0) = 1
7          else t_ij^(0) = 0
8  for k = 1 to n
9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10     for i = 1 to n
11         for j = 1 to n
12             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13 return T^(n)
```

Johnson’s Algorithm

Johnson’s Algorithm computes all-pair shortest paths for all the vertices.

- Suitable for ***sparse graphs***
- Asymptotically better than the ***Floyd-Warshall*** algorithm
- Either returns a matrix of shortest paths for all pairs of vertices or reports that the graph contains a ***negative-weight cycle***
- Uses both ***Dijkstra’s algorithm*** and the ***Bellman-Ford*** algorithm ***as subroutines***
- Uses the ***reweighting technique***

Johnson's Algorithm: Reweighting

If all weights w in a graph $G = (V, E, w)$ are ***non-negative***, we can find shortest paths between all pairs of vertices by running ***Dijkstra's algorithm*** on ***each vertex*** in the graph.

- The running time of this part is $O((V^2 + VE)\log V)$ with a ***min-priority Q***.

If G has ***negative-weight edges*** but ***no negative-weight cycles***, we simply compute the new set of edge weights so that we can apply ***Dijkstra's algorithm***.

- The new set of edge weights $\widehat{w}$ must satisfy the following ***two properties***:
  - (***P1***) 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 $\widehat{w}$
  - (***P2***) For all edges (u, v), the new weight $\widehat{w}(u, v)$ is ***non-negative***

Johnson's Algorithm: Reweighting

***Lemma**: (P1: Reweighting does not change shortest paths)*

Given a ***weighted***, ***directed*** graph $G = (V, E, w)$ with a weight function $w: E \rightarrow \mathbb{R}$, let $h: V \rightarrow \mathbb{R}$ be a function mapping vertices to real numbers, for each edge $(u, v) \in E$, we define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v). \qquad \textit{(Eq.7)}$$

Let $p = < v_0, v_1, v_2, \dots, v_k >$ be any path from $v_0$to v_k.

Then, p is a shortest path from $v_0$to $v_k$if and only if p is also a shortest path with weight function $\widehat{w}$.

(***Claim I***) That is, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \hat{\delta}(v_0, v_k)$. [$\hat{\delta}$ denotes shortest-path weights derived from weight function $\widehat{w}$.]

(***Claim II***) Further more, G has a negative-weight cycle using w if and only if G has a negative-weight cycle using $\widehat{w}$.

Johnson's Algorithm: Reweighting

<u>Proof</u>:

(Claim I)

We will start by showing that $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$.

Since

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

Since $h(v_0)$ and $h(v_k)$ are ***path-independent***, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using weight function $\widehat{w}$.

Thus, we have that $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \hat{\delta}(v_0, v_k)$. ■

Johnson's Algorithm: Reweighting

Proof:

(Claim II)

We show that G has a negative-weight cycle using w if and only if G has a negative-weight cycle using $\widehat{w}$.

Consider any cycle $c = < v_0, v_1, v_2, \ldots, v_k >$, where $v_0 = v_k$.

$$\widehat{w}(c) = w(c) + h(v_0) - h(v_k) \qquad [\textbf{\textit{Eq.7}}]$$

$$\widehat{w}(c) = w(c) \qquad [v_0 = v_k \rightarrow h(v_0) = h(v_k)]$$

Therefore, G has a negative-weight cycle using w if and only if G has a negative-weight cycle using $\widehat{w}$. ■

Johnson’s Algorithm: Reweighting

***Lemma*: (P2: Reweighting ensures non-negativity)**

Given a ***weighted***, ***directed*** graph $G = (V, E, w)$ with a weight function $w: E \rightarrow \mathbb{R}$, let $h: V \rightarrow \mathbb{R}$ be a function mapping vertices to real numbers. we define

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

For all edges (u, v) , the new weight $\widehat{w}(u, v)$ is ***non-negative.***

***Proof*:**

We construct a new graph $G' = (V', E')$, where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and $E' = E \cup \{(s, v): v \in V\}$.

We extend the weight function w so that $w(s, v) = 0$ for all $v \in V$.

Observation*:** G' has ***no negative-weight cycles if and only if G has ***no negative-weight cycles***.

Johnson's Algorithm: Reweighting

***Proof*: (Continued)**

Suppose that G and G' have ***no negative-weight cycles***.

Define $h(v) = \delta(s, v)$ for all $v \in V'$.

$h(v) \leq h(h) + w(u, v)$ for all $(u, v) \in E'$ [***Triangle Inequality***] (See ***Lecture 10***)

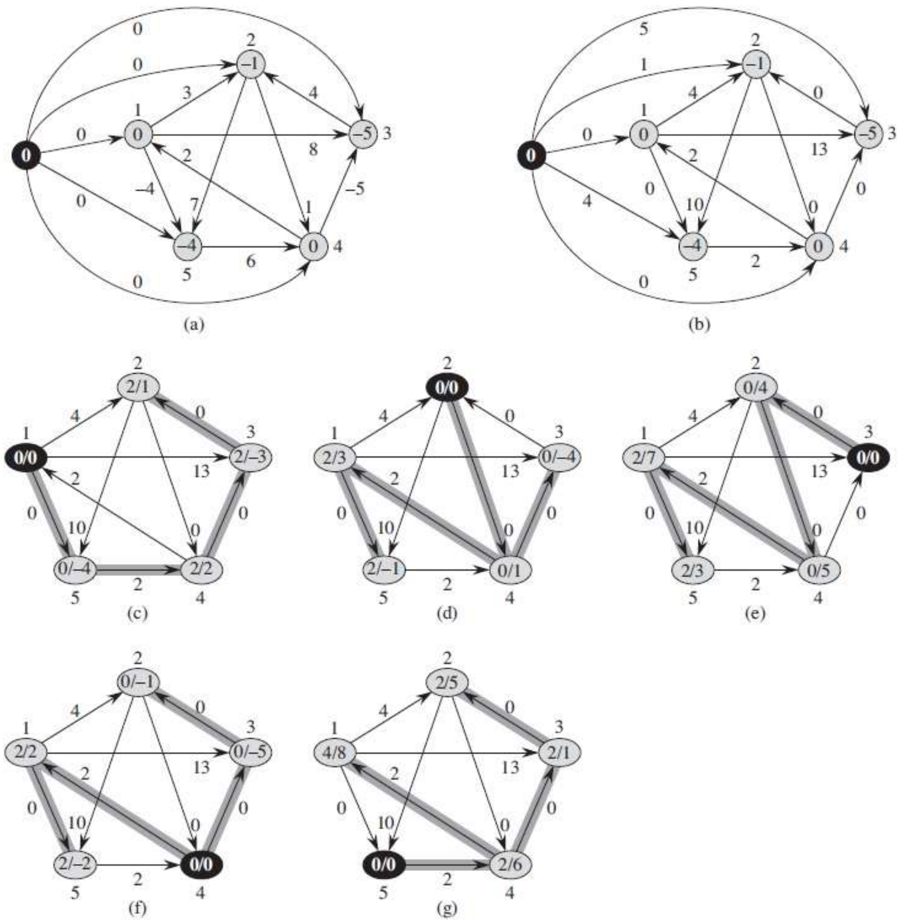
$h(v) - h(v) \leq h(h) + w(u, v) - h(v)$

$0 \leq w(u, v) + h(u) - h(v)$

$0 \leq \hat{w}(u, v)$ [***Eq.7***]

For all edges (u, v) , the new weight $\hat{w}(u, v)$ is ***non-negative.*** ■

Johnson’s Algorithm: Example



Johnson's Algorithm

JOHNSON(G, w)

compute G', where $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
    $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD(G', w, s) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
        set $h(v)$ to the value of $\delta(s, v)$
            computed by the Bellman-Ford algorithm
    for each edge $(u, v) \in G'.E$
        $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
    let $D = (d_{uv})$ be a new $n \times n$ matrix
    for each vertex $u \in G.V$
        run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
        for each vertex $v \in G.V$
            $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
    return D

Johnson's Algorithm: Analysis

The running time of Johnson's algorithm is determined by the for loop of ***lines 9-12***.

Running Dijkstra's algorithm V times costs $\mathrm{O}((V^2 + VE) \log V)$.

If G is sparse,

then $E = O(V)$ so the running time becomes $\mathrm{O}((V^2 + V \cdot \mathrm{V}) \log V) = O(V^2 \log V)$,

which is asymptotically faster than the ***Floyd-Warshall*** algorithm, which takes $\Theta(V^3)$ time.

Summary

In this lecture, we have covered the topic of all-pair shortest path problems:

- All-Pair Shortest Path Problems
  - Naive Dynamic Programming View
  - Matrix Multiplication View
  - Floyd-Warshall Algorithm
  - Transitive Closure of Directed Graphs
  - Johnson’s Algorithm