Advanced Analysis of Algorithms

Dr. Qaiser Abbas
Department of Computer Science & IT,
University of Sargodha, Sargodha, 40100, Pakistan
qaiser.abbas@uos.edu.pk

Floyd-Warshall Algorithm (Background)

- For finding shortest paths between *all pairs* of vertices, run Bellman-Ford or Dijkstra's algorithm for each vertex in the graph. Thus, the run times for these strategies would be (for dense graphs where $|E| \approx |V|^2$):
  - **Bellman-Ford:**
    - $|V|\ O(VE) \approx O(V^4)$
  - **Dijkstra**
    - $|V|\ O(V^2 + E) \approx O(V^3)$
    - $|V|\ O(V \lg V + E) \approx O(V^2 \lg V + VE)$
- For dense graphs an often more efficient algorithm (with very low hidden constants) for finding all pairs shortest paths is the *Floyd-Warshall algorithm*.

Floyd-Warshall Algorithm

- The working of Floyd-Warshall algorithm is based on the property of *intermediate* vertices of a shortest path. An *intermediate* vertex for a path $p = <v_1, v_2, ..., v_j>$ is any vertex other than v_1 or v_j.
- If the vertices of a graph G are indexed by $\{1, 2, ..., n\}$, then consider a subset of vertices $\{1, 2, ..., k\}$. Assume p is a minimum weight path from vertex i to vertex j whose intermediate vertices are drawn from the subset $\{1, 2, ..., k\}$.

Floyd-Warshall Algorithm

- If we consider vertex k on the path, then either:
  - k is **not** an intermediate vertex of p (i.e., is not used in the minimum weight path)

    $\Rightarrow$ all intermediate vertices are in $\{1, 2, ..., k\text{-}1\}$
  - k is an intermediate vertex of p (i.e., is used in the minimum weight path)

    $\Rightarrow$ we can divide p at k giving two subpaths p_1 and p_2 giving $v_i \leadsto k \leadsto v_j$

Floyd-Warshall Algorithm

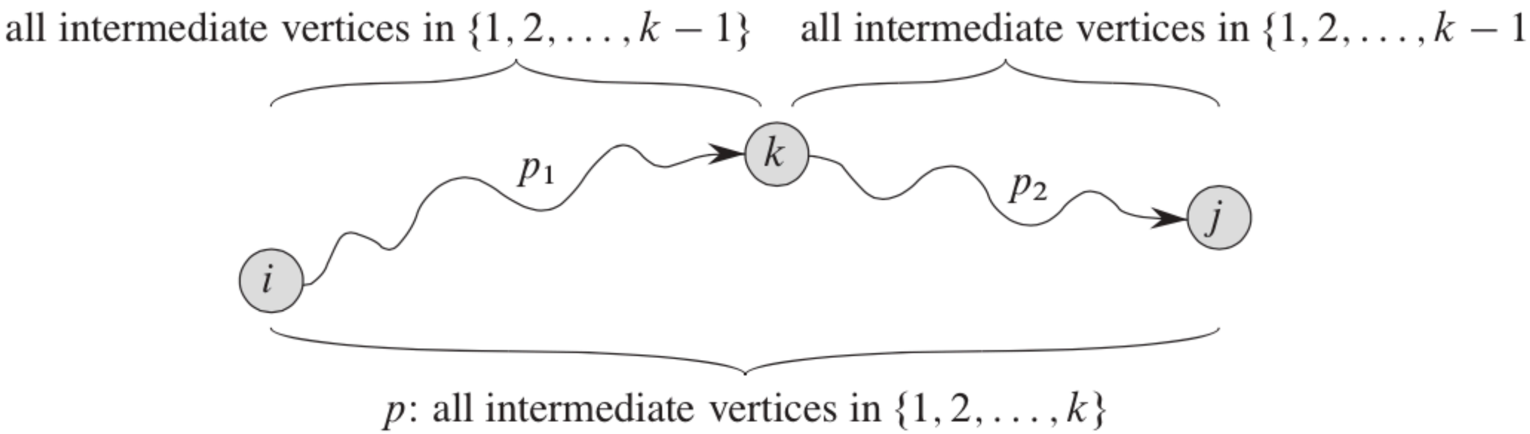


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

Floyd-Warshall Algorithm

- For D^0_{ij} matrix entries, if i=j, then D^0_{ij}= 0 and if i≠j, then D^0_{ij} = ∞ if there is no any edge.
- If a quantity $d^{(k)}_{ij}$ as the minimum weight of the path from vertex *i* to vertex *j* with intermediate vertices drawn from the set {1, 2, ..., *k*}, we have the following recursive solution

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

- Optimal values (when *k* = *n*) in a matrix as

$$D^{(n)} = \left(d_{ij}^{(n)}\right) = \delta(i, j)$$

Floyd-Warshall Algorithm

- Different methods for constructing shortest paths in the Floyd- Warshall algorithm.
  - One way, is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix.
  - Alternatively, we can compute the predecessor matrix Π while the algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\Pi^{(0)}$, $\Pi^{(1)}$, … , $\Pi^{(n)}$,where $\Pi = \Pi^{(n)}$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set from {1,2,…k}

Floyd-Warshall Algorithm

- We can give a recursive formulation of $\pi_{ij}^{(k)}$ When k=0, a shortest path from i to j has no intermediate vertices at all. Thus,

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

- For k ≥ 1, if we take the path i → k → j, where k ≠ j, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set {1,2,…k}. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set {1,2,…k-1}. Formally, for k≥1
- .

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

Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$

$D^{(0)} = W$

$\Pi^{(0)} = \pi^{(0)}_{ij} = \text{NIL if } i=j \text{ or } w_{ij} = \infty$

$\qquad\qquad = i \quad \text{if } i \neq j \text{ and } w_{ij} < \infty$

for $k = 1$ to n

 let $D^{(k)} = (d^{(k)}_{ij})$ be a new nxn matrix

 for $i = 1$ to n

  for $j = 1$ to n

   $d^{k}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$

   if $d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$

    $\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}$

   else

    $\pi^{(k)}_{ij} = \pi^{(k-1)}_{kj}$

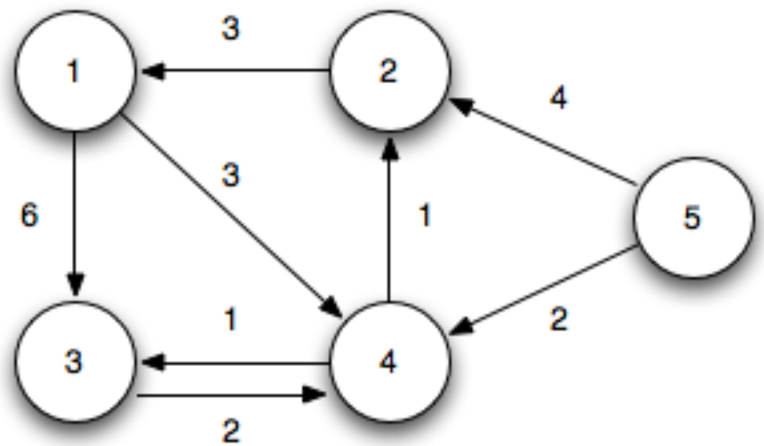
return $D^{(n)}$

Floyd-Warshall Algorithm

- Basically, the algorithm works by repeatedly exploring paths between every pair using each vertex as an intermediate vertex.
- Since Floyd-Warshall is simply three (tight) nested loops, the run time is clearly $O(V^3)$.

Floyd-Warshall Algorithm

- **Example:**



Floyd-Warshall Algorithm

- **Example:**
  – *Initialization*: (k = 0)

D

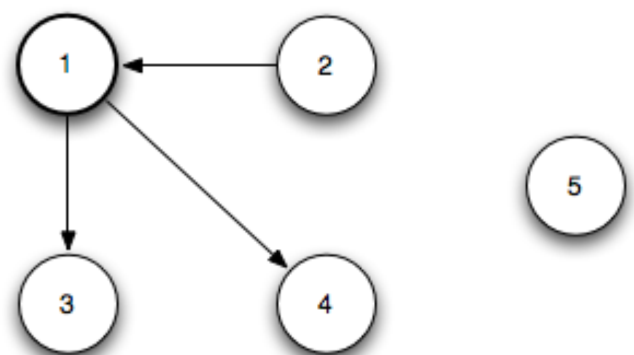
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Floyd-Warshall Algorithm

- **Example:**
  - *Iteration 1*: (k = 1) Shorter paths from 2 $\rightsquigarrow$ 3 and 2 $\rightsquigarrow$ 4 are found through vertex 1



D

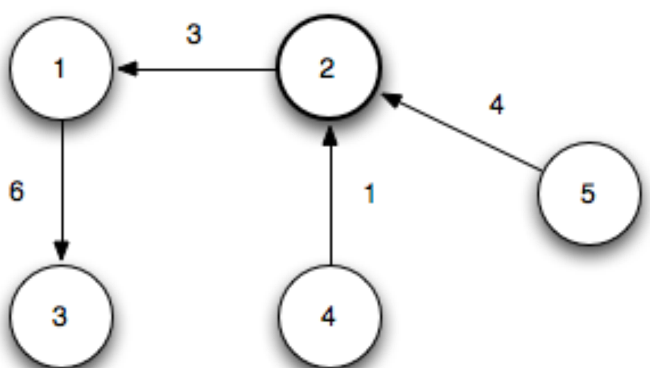
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Floyd-Warshall Algorithm

- **Example:**
  - *Iteration 2*: (*k* = 2) Shorter paths from 4 $\rightsquigarrow$ 1, 5 $\rightsquigarrow$ 1, and 5 $\rightsquigarrow$ 3 are found through vertex 2



D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Floyd-Warshall Algorithm

- **Example:**
  – *Iteration 3*: (k = 3) No shorter paths are found through vertex 3

D

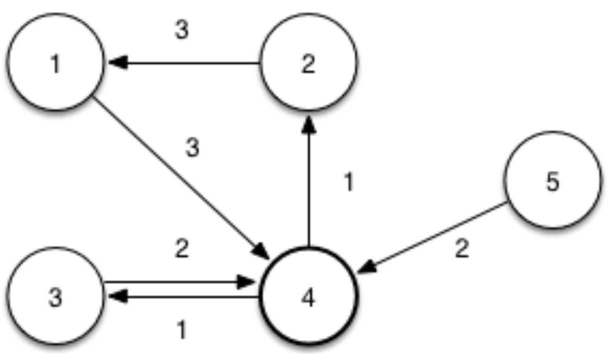
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Floyd-Warshall Algorithm

- **Example:**
  - *Iteration 4*: (k = 4) Shorter paths from 1 ↝ 2, 1 ↝ 3, 2 ↝ 3, 3 ↝ 1, 3 ↝ 2, 5 ↝ 1, 5 ↝ 2, 5 ↝ 3 are found through vertex 4



D

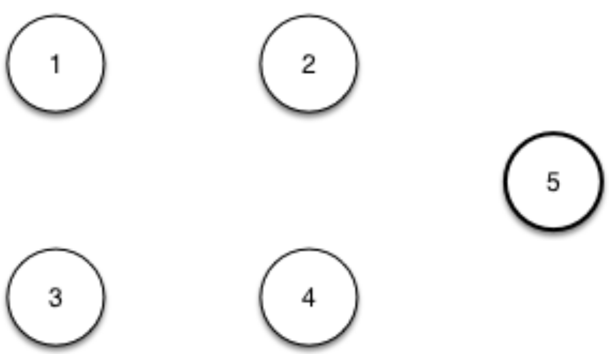
	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	2	**4**	**4**	5	/

Floyd-Warshall Algorithm

- **Example:**
  - *Iteration 5*: (k = 5) No shorter paths are found through vertex 5



D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

Floyd-Warshall Algorithm

- **Example:**
  – The final shortest paths for all pairs is given by

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

Transitive Closure

- Floyd-Warshall can be used to determine whether or not a graph has *transitive closure*, i.e., whether or not there are paths between all vertices.
  - Assign all edges in the graph to have weight = 1
  - Run Floyd-Warshall
  - Check if *all* $d_{ij} < n$
- This procedure can implement a slightly more efficient algorithm through the use of logical operators rather than min() and +.

Johnson's Algorithm

- Floyd-Warshall is efficient for dense graphs, if the graph is sparse then an alternative all pairs shortest path strategy known as *Johnson's algorithm* can be used.
- This algorithm uses Bellman-Ford to detect any negative weight cycles and then *reweighting* the edges to allow Dijkstra's algorithm to find the shortest paths. Has running time $O(V^2 \lg V + VE)$.
- The problem is to find all pairs shortest paths in a given weighted directed Graph and weights may be negative.

Johnson's Algorithm

- If we apply Dijkstra's Single Source shortest path algorithm O(Vlog V) for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V*VLogV) time.
- So, Dijkstra's SSSP seems to be a better option than Floyd Warshell O(V^3), but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge.
- The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.

Johnson's Algorithm

- **How to transform a given graph to a graph with all non-negative weight edges?**
- Adding weight to all edges. Unfortunately, this doesn't work.
- In a weighted graph, assume that the shortest path from a source 's' to a destination 't' is correctly calculated using a shortest path algorithm. Is the following statement true?
  - If we increase weight of every edge by 1, the shortest path always remains same.

    (A) Yes

    (B) No
  - **Answer: (B)** (Explanation is on next slide)

Johnson's Algorithm

- **Explanation:** See the following counterexample.
- There are 4 edges s→a, a→b, b→t and s→t of wights 1, 1, 1 and 4 respectively. The shortest path from s to t is s-a, a-b, b-t. If we increase weight of every edge by 1, the shortest path changes to s-t.

```
      1          1         1
s-----a-----b-----t
 \                       /
   \                   /
     \_________/
            4
```

- So, If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.

Johnson's Algorithm

- The idea of Johnson's algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be h[u].
- We reweight edges using vertex weights. For example, for an edge (u, v) of weight w(u, v), the new weight becomes w(u, v) + h[u] – h[v].
- The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative.

Johnson's Algorithm

- How do we calculate h[] values?
  - Bellman-Ford algorithm is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are h[] values.

Johnson's Algorithm

- **Theory of Algorithm**

  1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G’.

  2) Run Bellman-Ford algorithm on G’ with s as source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.

  3) Reweight the edges of original graph. For each edge (u, v), assign the new weight as “original weight + h[u] – h[v]”.

  4) Remove the added vertex s and run Dijkstra’s algorithm for every vertex.

Johnson's Algorithm

JOHNSON(G, w)

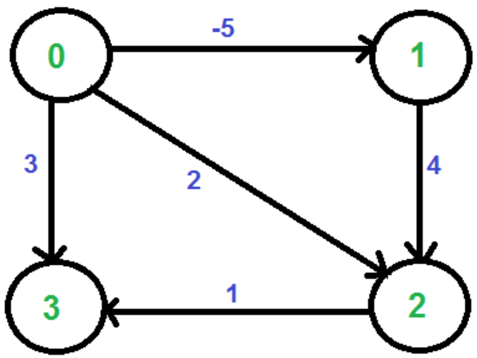
compute G', where $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
    $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD(G', w, s) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
        set $h(v)$ to the value of $\delta(s, v)$
            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

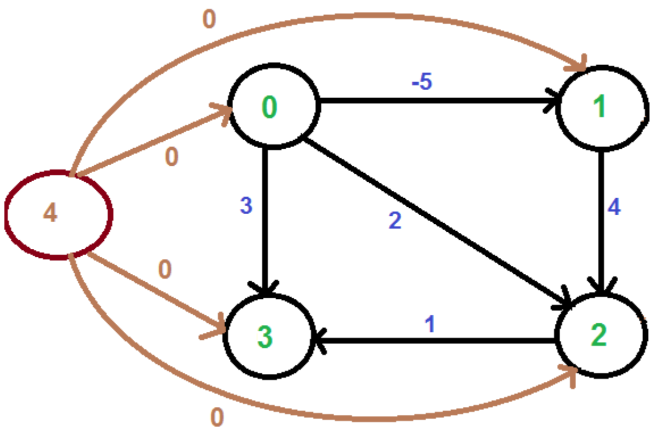
Johnson's Algorithm

- **How does the transformation ensure nonnegative weight edges?**
- The following property is always true about h[] values as they are shortest distances.
  - h[v] <= h[u] + w(u, v) The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v).
  - The new weights are w(u, v) + h[u] - h[v]. The value of the new weights must be nonnegative because of the inequality "h[v] <= h[u] + w(u, v)".

Johnson's Algorithm

- **Example:**
  - Let us consider the following graph.

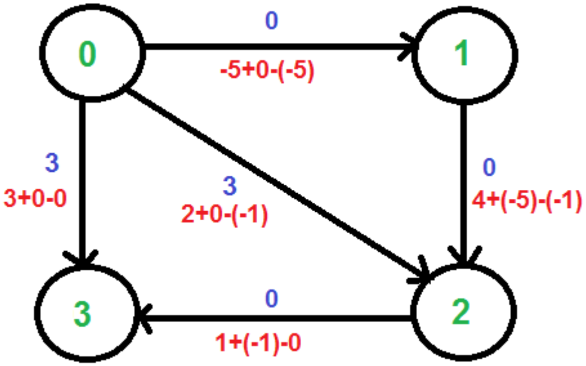




  - We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.

Johnson's Algorithm

- We calculate the shortest distances from 4 to all other vertices (0,1,2,3) using Bellman-Ford algorithm as h[] = {0, -5, -1, 0}.. Then Remove the source vertex 4 and reweight the edges using formula. w(u, v) = w(u, v) + h[u] - h[v].



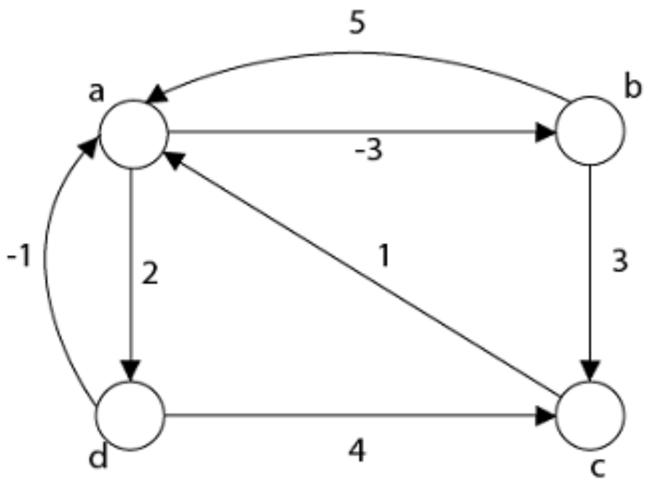
- Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.

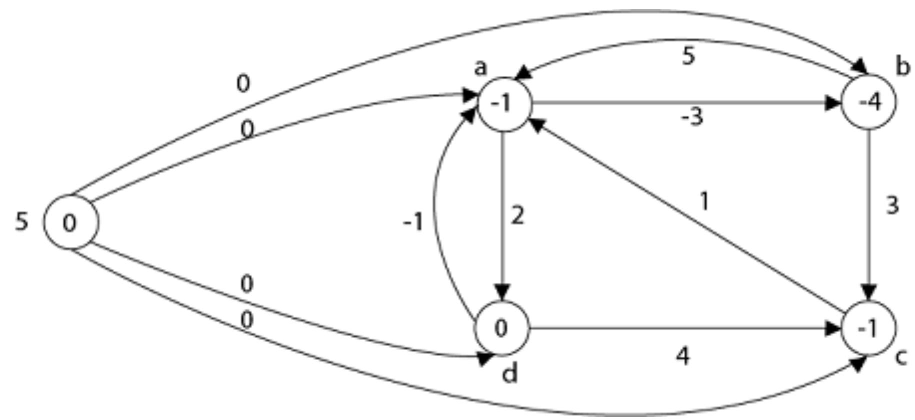
Johnson's Algorithm

- **Time Complexity:** The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times.
- Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VLogV). So overall time complexity is $O(V^2 \log V + VE)$.
- The time complexity of Johnson's algorithm becomes same as Floyd Warshell when the graphs is complete (For a complete graph $E = O(V^2)$. But for sparse graphs, the algorithm performs much better than Floyd Warshell.

Example Run (Read it Yourself)

- **Step1:** Take any source vertex's' outside the graph and make distance from's' to every vertex '0'.





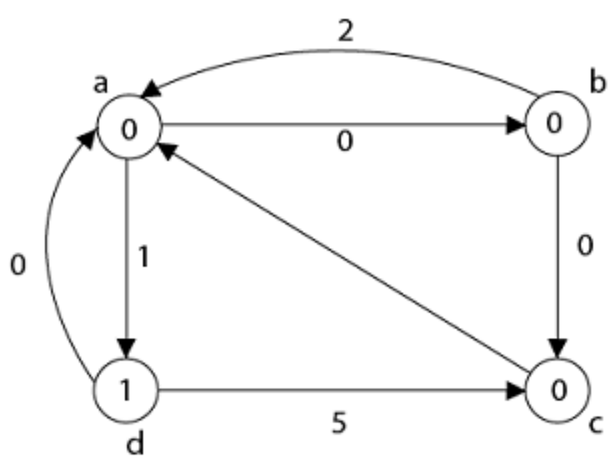
- **Step2:** Apply Bellman-Ford Algorithm and calculate minimum weight on each vertex.

Example Run (Read it Yourself)

- **Step3:**
  - $w(a, b) = w(a, b) + h(a) - h(b) = -3 + (-1) - (-4) = 0$
  - $w(b, a) = w(b, a) + h(b) - h(a) = 5 + (-4) - (-1) = 2$
  - $w(b, c) = w(b, c) + h(b) - h(c) = 3 + (-4) - (-1) = 0$
  - $w(c, a) = w(c, a) + h(c) - h(a) = 1 + (-1) - (-1) = 1$
  - $w(d, c) = w(d, c) + h(d) - h(c) = 4 + 0 - (-1) = 5$
  - $w(d, a) = w(d, a) + h(d) - h(a) = -1 + 0 - (-1) = 0$
  - $w(a, d) = w(a, d) + h(a) - h(d) = 2 + (-1) - 0 = 1$

Example Run (Read it Yourself)

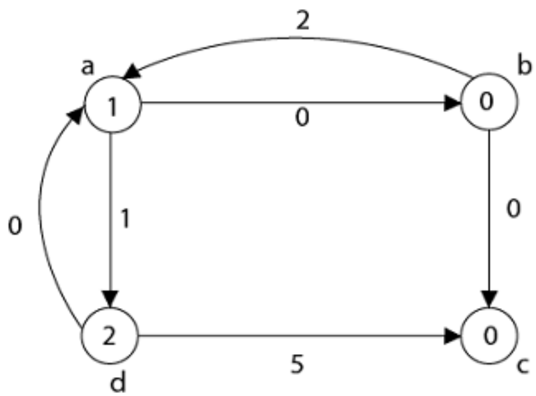
- **Step 4:** Now all edge weights are positive and now we can apply Dijkstra's Algorithm on each vertex and make a matrix corresponds to each vertex in a graph
- **Case 1:** 'a' as a source vertex



a, a	0
a, b	0
a, c	0
a, d	1

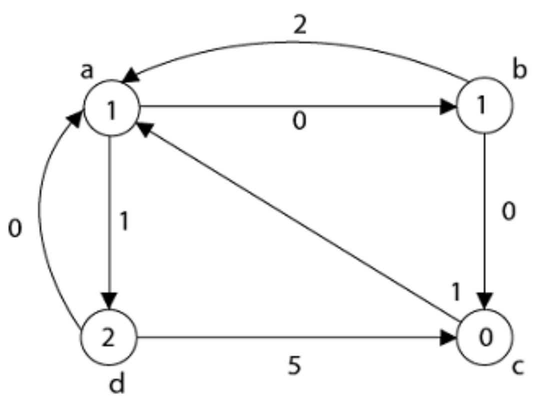
Example Run (Read it Yourself)

- **Case 2:** 'b' as a source vertex



b, a	2
b, b	0
b, c	0
b, d	2

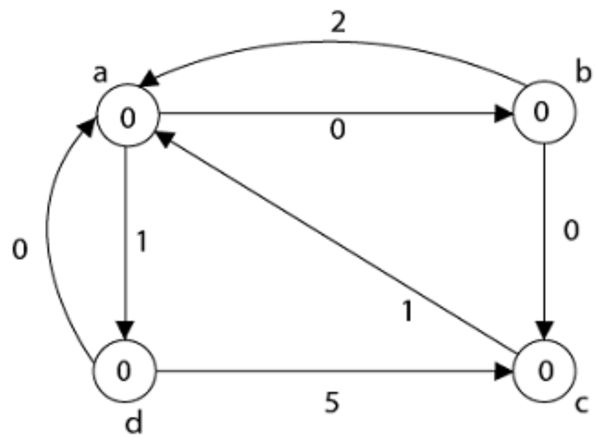
- **Case 3:** 'c' as a source vertex



c, a	1
c, b	1
c, c	0
c, d	2

Example Run (Read it Yourself)

- **Case4:**'d' as source vertex

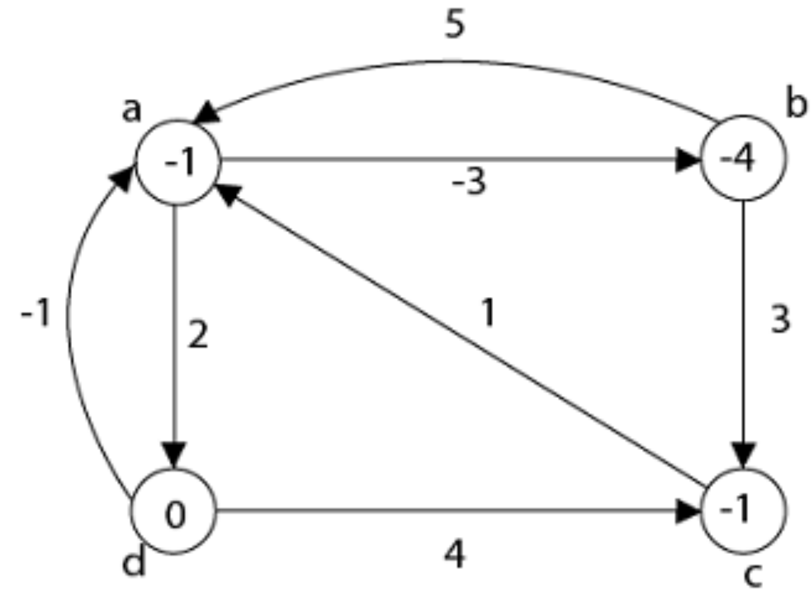


d, a	0
d, b	0
d, c	0
d, d	0

	a	**b**	**c**	**d**
a	0	0	0	1
b	1	0	0	2
c	1	1	0	2
d	0	0	0	0

Example Run (Read it Yourself)

- **Step5:**
- $d_{uv} \leftarrow \delta(u, v) + h(v) - h(u)$
  d (a, a) = 0 + (-1) - (-1) = 0
  d (a, b) = 0 + (-4) - (-1) = -3
  d (a, c) = 0 + (-1) - (-1) = 0
  d (a, d) = 1 + (0) - (-1) = 2
  d (b, a) = 1 + (-1) - (-4) = 4
  d (b, b) = 0 + (-4) - (-4) = 0
  d (c, a) = 1 + (-1) - (-1) = 1
  d (c, b) = 1 + (-4) - (-1) = -2
  d (c, c) = 0
  d (c, d) = 2 + (0) - (-1) = 3
  d (d, a) = 0 + (-1) - (0) = -1
  d (d, b) = 0 + (-4) - (0) = -4
  d (d, c) = 0 + (-1) - (0) = -1
  d (d, d) = 0



	a	b	c	d
a	0	-3	0	2
b	4	0	3	6
c	1	-2	0	3
d	-1	-4	-1	0

Homework #6

25.2-2

Show how to compute the transitive closure using the technique of Section 25.1.

25.2-4

As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

FLOYD-WARSHALL$'(W)$

```
n = W.rows
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d_ij = min(d_ij, d_ik + d_kj)
return D
```

Homework #6

25.2-6
How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

25.2-8
Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

25.3-4
Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min_{(u,v)\in E}\{w(u,v)\}$, just define $\hat{w}(u,v) = w(u,v) - w^*$ for all edges $(u,v) \in E$. What is wrong with the professor's method of reweighting?

25.3-6
Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He claims that instead we can just use $G' = G$ and let s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.