CSCI203

Algorithms and Data Structures

Graphs (II)

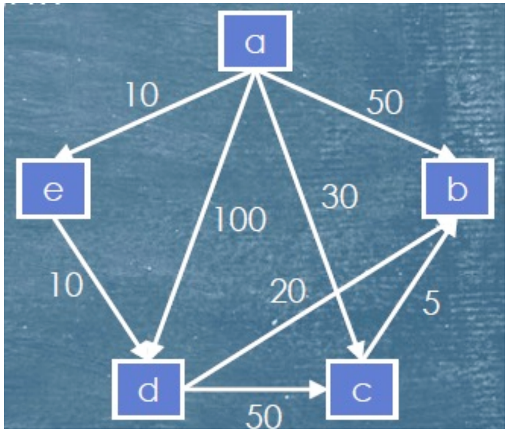
Lecturer: Dr. Fenghui Ren
Room 3.203
Email: fren@uow.edu.au

Weighted Graphs

- Frequently, we find that travelling along an edge in a graph has some associated cost (or profit) associated with it:
  - The distance along the edge;
  - The cost of petrol;
  - The time of travel;
  - Etc.
- We call these Weighted Graphs.
- We call the edge values weights

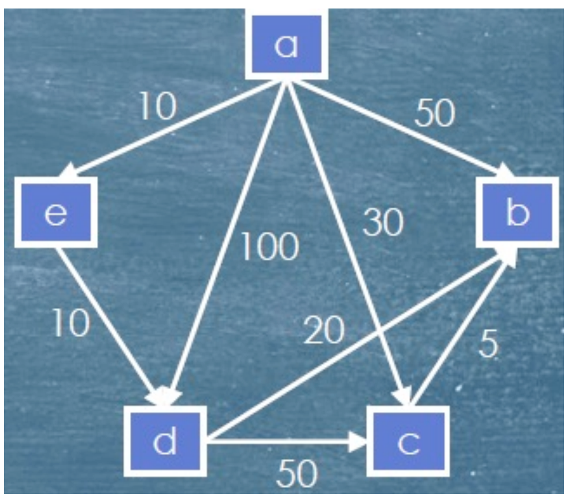
Definition

- We extend our previous graph definition as follows:
  - A weighted graph, G, consists of the ordered sequence, (V, E, W) where V and E are the vertices and edges and W are the edge weights.
- Consider the weighted graph shown to the right:
  - $V = \{a, b, c, d, e\}$
  - $E = \{(a, b), (a, c), (a, d), (a, e), (c, b), (d, b), (d, c), (e, d)\}$
- W is a function that maps edges to weights:
  - e.g $W((a, b)) = 50$.



Representation

- We can extend our adjacency matrix representation of a graph by replacing the zero-one existence value with the edge weight.



—	50	30	100	10
—	—	—	—	—
—	5	—	—	—
—	20	50	—	—
—	—	—	10	—

Representation

- In this example, "—" indicates that no edge exists.
- The actual value will depend on the nature of the weights:
  - E.g. if all weights are non-zero use 0.
  - We often use ∞ to represent missing edges.
- We can also use the adjacency list representation:
  - We just need to pair each edge with its corresponding weight.

Shortest Path

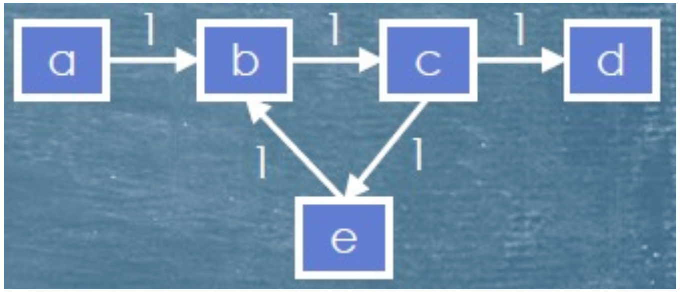
- A common problem associated with weighted graphs is finding the shortest path between vertices.
- There are several versions of this problem:
  - Single Source—All destinations;
  - Single Source—Single Destination;
  - All Sources—All Destinations;
  - All Sources—Single Destination.
- Each has applications in the real world.
- We will start by looking at the first of these types.

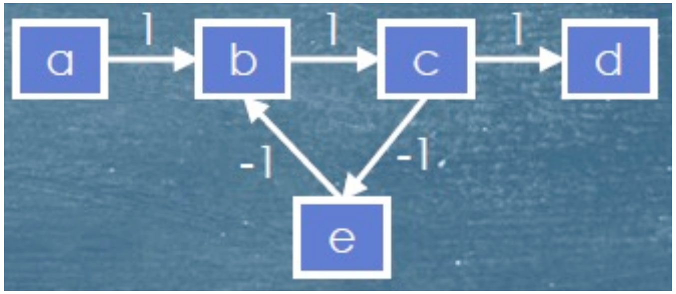
Single Source—All Destinations

- This problem is stated as follows:
- Starting at some source vertex, s, find the shortest path from s to each other reachable vertex in the graph.
- As we shall see later, the solution to this problem can be used as a basis for solving all of the other shortest path formulations.
- We will examine two algorithms for solving this problem:
  - Dijkstra;
  - Bellman Ford.
- Each has advantages in certain cases.

Negative Weights

- There is no a priori reason why the edge weights in a graph must be positive, but negative edge weights can cause problems.
- Consider the following graph:
  - Clearly the length of the shortest path from a to d is 3.
- But what if we change the weights?
  - Now what is the shortest path?
- The problem is that we now have a negative cost cycle.





What is a Path

- While it is obvious what a path is, we should define it formally.
- A path p from vertex v_1 to vertex v_k is an ordered sequence $(v_1, v_2, \ldots, v_{k-1}, v_k)$ where each edge $(v_i, v_{i+1}) \in E$.
- The weight of path p, $W(p)$ is the sum of the edge weights:
  - $W(p) = \sum_{i=1}^{k-1} W(v_i, v_{i+1})$

What doesn't work?

- We might be tempted to try using a technique we already know for traversing a graph, breadth first search, in our search for shortest paths.
- Unfortunately, this does not always work.
- The two definitions of shortest path:
  - Fewest edges;
  - Smallest weight;
- May not always coincide.

Dijkstra's Algorithm

- This algorithm works by dividing the vertices into two sets, S and C.
- At each iteration, S contains the set of nodes that have already been chosen.
  - This is the selected set.
- At each iteration, C contains the set of nodes that have not yet been chosen:
  - This is the candidate set.
- At each step we move the node which is cheapest to reach from C to S.

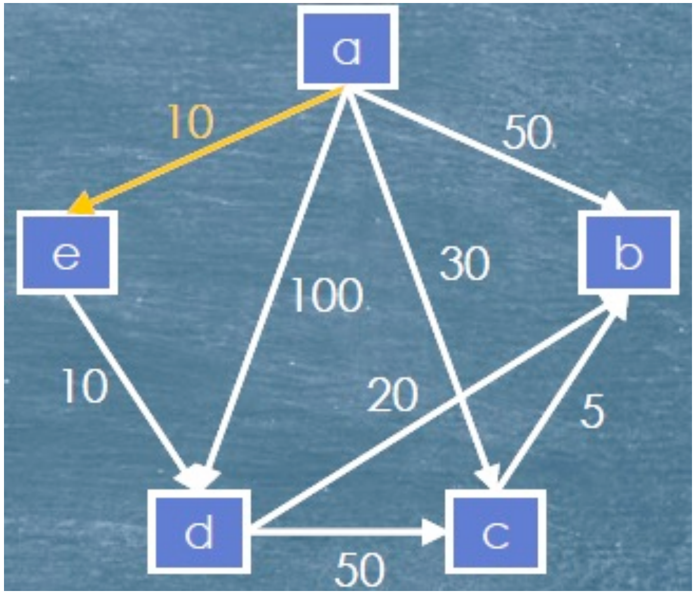
Dijkstra's Algorithm

- We also need a function D such that $D(c_i)$ is the shortest distance we have so far found from vertex s to vertex c_i in the candidate set C.
- Initially:
  - The selected set, S, just contains the start vertex;
  - The candidate set, C, contains all the other vertices;
  - The distance function, $D()$ has value 0 for vertex s and is infinite for all other vertices.
- We start by re-evaluating D for each vertex directly reachable from vertex s.

Dijkstra's Algorithm: an Example

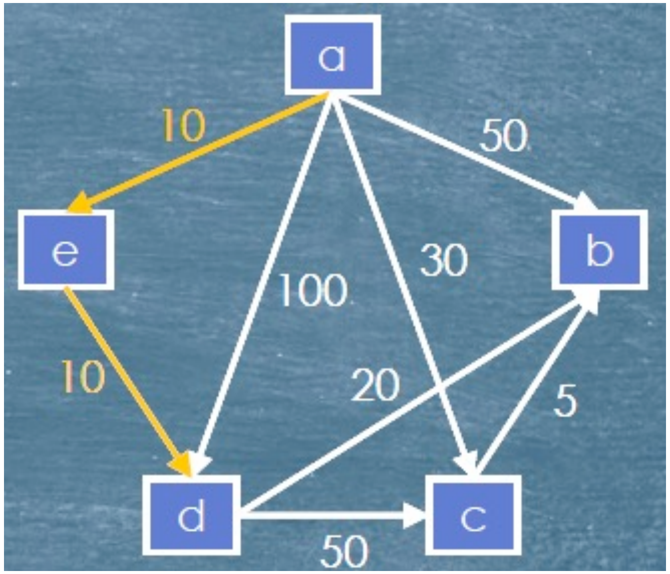
- Step 0:
- $S = \{a\}$
- $C = \{b, c, d, e\}$
- $D = \{50, 30, 100, 10\}$

- We now select the minimum value of D, $D(e) = 10$.



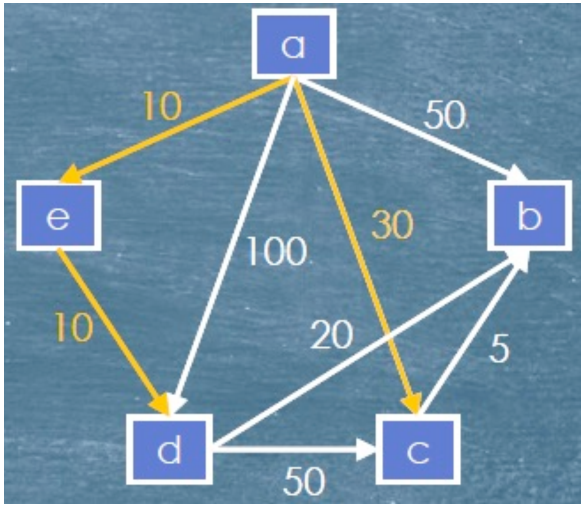
Dijkstra's Algorithm: an Example

- Step 1: move vertex e from C to S.
- $S = \{a, e\}$
- $C = \{b, c, d\}$
- We now update D by looking at vertices we can reach from vertex e.
- $D = (50, 30, 100 \rightarrow 20)$
- We now select the minimum value of D, $D(d) = 20$.



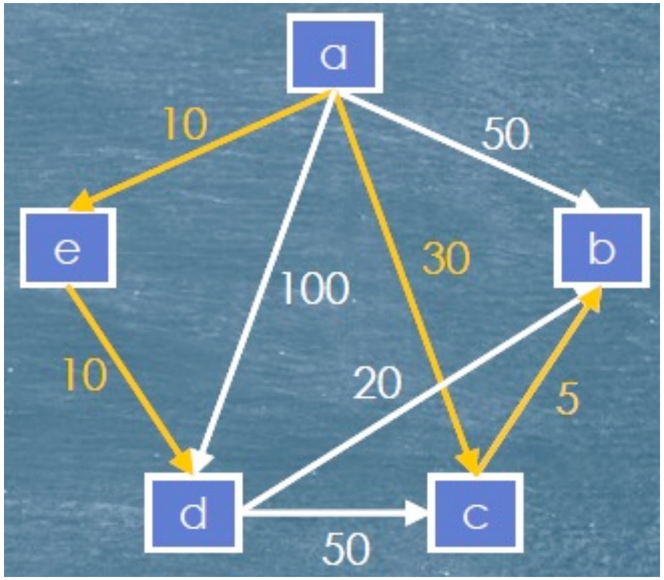
Dijkstra's Algorithm: an Example

- Step 2: move vertex d from C to S.
- $S = \{a, e, d\}$
- $C = \{b, c\}$
- We now update D by looking at vertices we can reach from vertex d.
- $D = (50 \rightarrow 40, 30)$
- We now select the minimum value of D, $D(c) = 30$.



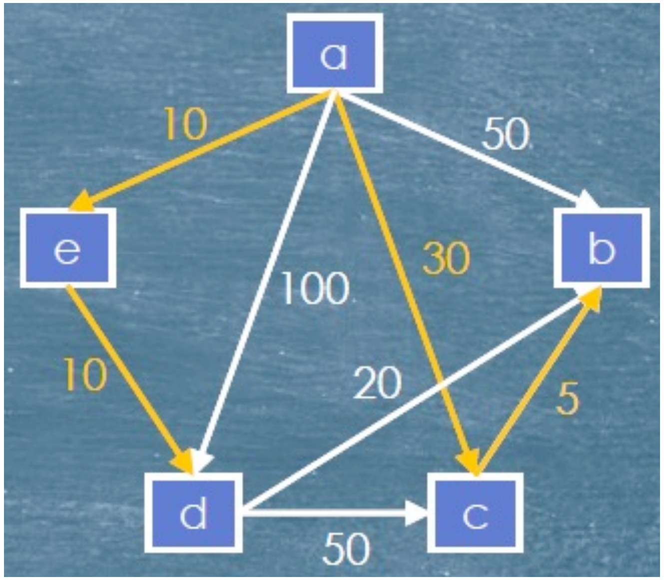
Dijkstra's Algorithm: an Example

- Step 3: move vertex c from C to S.
- $S = \{a, e, d, c\}$
- $C = \{b\}$
- We now update D by looking at vertices we can reach from vertex c.
- $D = (40 \rightarrow 35)$
- We now select the minimum value of D, $D(b) = 35$.



Dijkstra's Algorithm: an Example

- Step 4: move vertex b from C to S.
- $S = \{a, e, d, c, b\}$
- $C = \{\}$
- We now have no remaining candidate vertices; we have finished.
- $W(a) = 0, W(e) = 10, W(d) = 20, W(c) = 30, W(b) = 35.$



Dijkstra's Algorithm: Pseudocode

```
Procedure Dijkstra(G: array[1..n, 1..n]): array [2..n]
   D: array[2..n]
   C: set = {2, 3, …, n}
   for i = 2 to n do
       D[i] = G[1, i]
   od
   repeat
       v = the index of the minimum D[v] not yet selected
       remove v from C // and implicitly add v to S
       for each u ∈ C do
          D[u] = min(D[u], D[v] + G[v, u])
       rof
   until C contains no reachable nodes
   return D
end Dijkstra
```

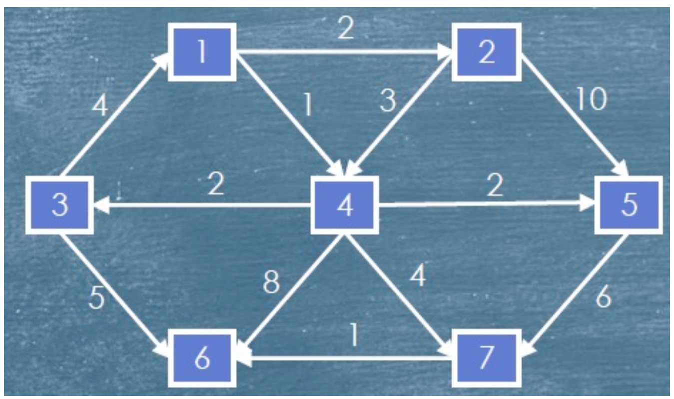
Recording Paths

- Like the basic DFS, Dijkstra's algorithm does not record the shortest path to each vertex, just its total weight.
- Also, like DFS, we can use a parent record, p, to keep track of how we reach each vertex.
- This entails a couple of minor changes to the algorithm…

Dijkstra's Algorithm: Path Recording

```
Procedure Dijkstra(G: array[1..n, 1..n]): array [2..n]
   D: array[2..n], P: array[2..n]
   C: set = {2, 3, …, n}, S: set = {}
   for i = 2 to n do
        D[i] = G[1, i]
        P[i]=1
   od
   repeat
        v = the index of the minimum D[v] not yet selected
        move v from C to S
        for each u ∈ C do
           D[u] = min(D[u], D[v] + G[v, u])
           p[u] = v
        rof
   until C contains no reachable nodes
   return D
end Dijkstra
```

A Larger Example

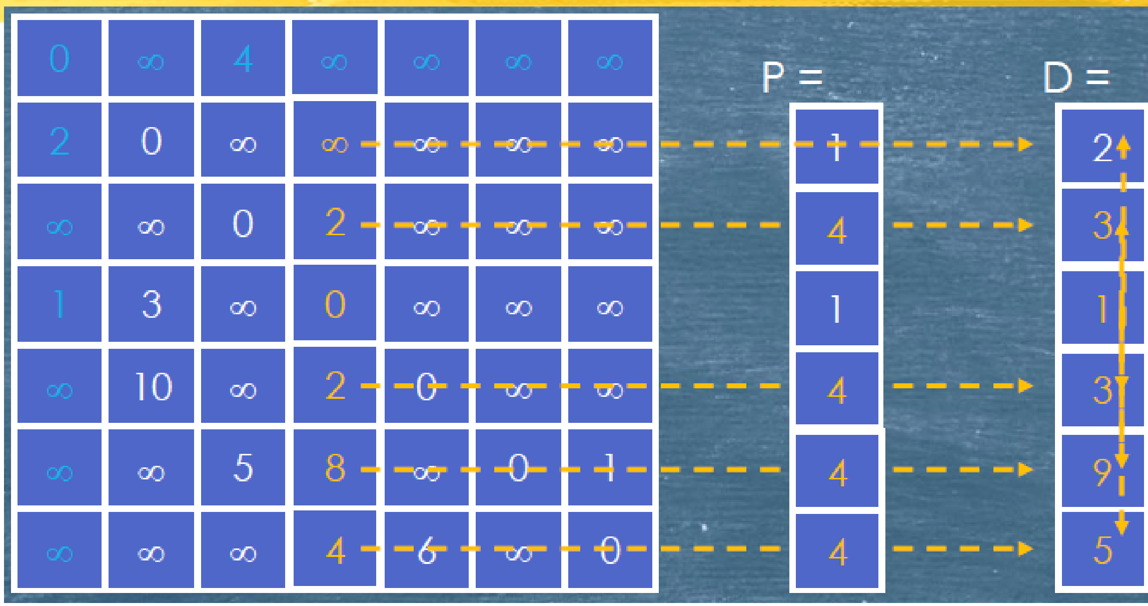


From

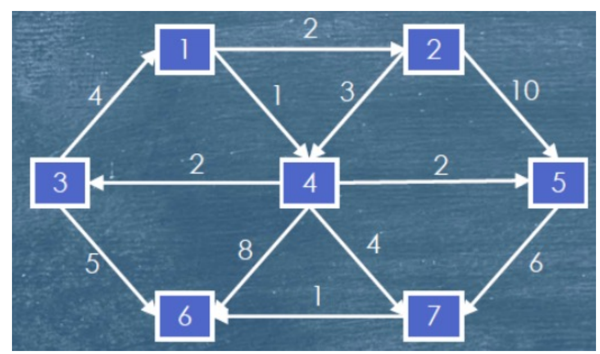
	0	∞	4	∞	∞	∞	∞
	2	0	∞	∞	∞	∞	∞
	∞	∞	0	2	∞	∞	∞
To	1	3	∞	0	∞	∞	∞
	∞	10	∞	2	0	∞	∞
	∞	∞	5	8	∞	0	1
	∞	∞	∞	4	6	∞	0

At start

- $G =$



$C = \{2,3,4,5,6,7\}$
$S = \{1\}$
$v = 4$
$D(v) = 1$

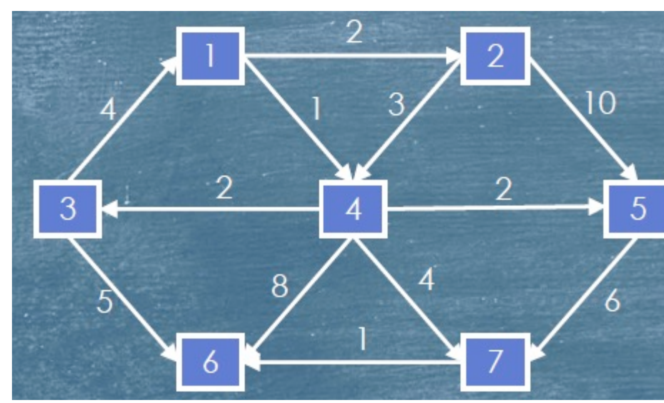


After step 1

- $G =$

$$\begin{bmatrix} 0 & \infty & 4 & \infty & \infty & \infty & \infty \\ 2 & 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 2 & \infty & \infty & \infty \\ 1 & 3 & \infty & 0 & \infty & \infty & \infty \\ \infty & 10 & \infty & 2 & 0 & \infty & \infty \\ \infty & \infty & 5 & 8 & \infty & 0 & 1 \\ \infty & \infty & \infty & 4 & 6 & \infty & 0 \end{bmatrix} \quad P = \begin{bmatrix} 1 \\ 4 \\ 1 \\ 4 \\ 4 \\ 4 \end{bmatrix} \quad D = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 3 \\ 9 \\ 5 \end{bmatrix}$$

$C = \{2, 3, 5, 6, 7\}$
$S = \{1,4\}$
$v = 2$
$D(v) = 2$



After step 2

- $G =$

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

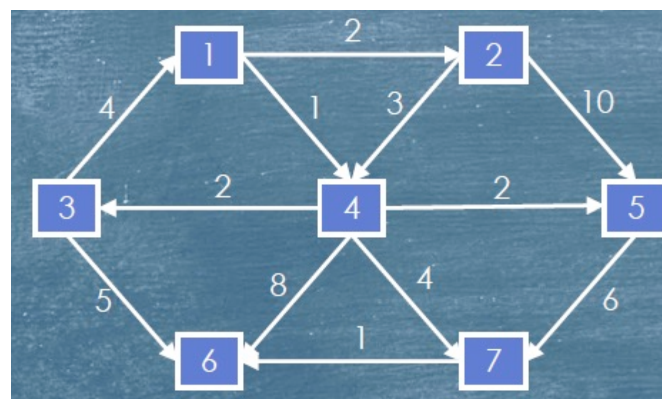
P =

1
4
1
4
3
4

D =

2
3
1
3
8
5

$$C = \{3, 5, 6, 7\}$$
$$S = \{1,4,2\}$$
$$v = 3$$
$$D(v) = 3$$



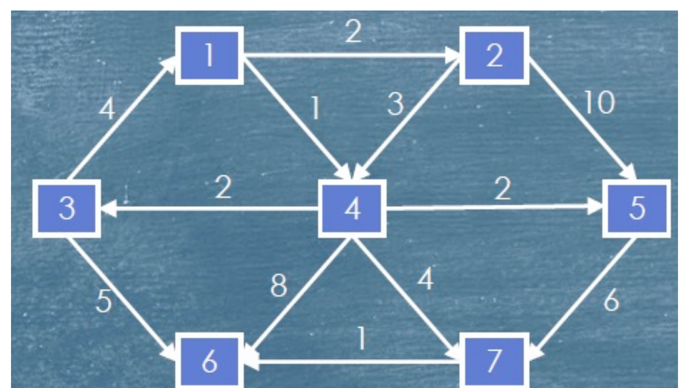
After step 3

- $G =$

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

$$P = \begin{bmatrix} 1 \\ 4 \\ 1 \\ 4 \\ 3 \\ 4 \end{bmatrix} \qquad D = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 3 \\ 8 \\ 5 \end{bmatrix}$$

$C = \{5, 6, 7\}$
$S = \{1,4,2,3\}$
$v = 5$
$D(v) = 3$



After step 4

- $G =$

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

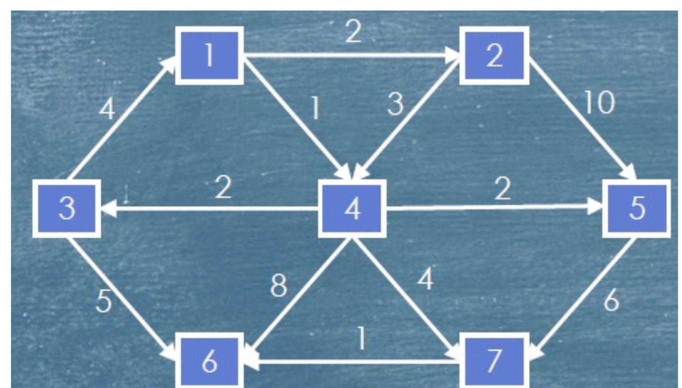
P =

1
4
1
4
7
4

D =

2
3
1
3
6
5

$C \ = \ \{6,7\}$
$S \ = \ \{1,4,2,3,5\}$
$v \ = 7$
$D(v) \ =5$



After step 5

- $G =$

0	∞	4	∞	∞	∞	∞
2	0	∞	∞	∞	∞	∞
∞	∞	0	2	∞	∞	∞
1	3	∞	0	∞	∞	∞
∞	10	∞	2	0	∞	∞
∞	∞	5	8	∞	0	1
∞	∞	∞	4	6	∞	0

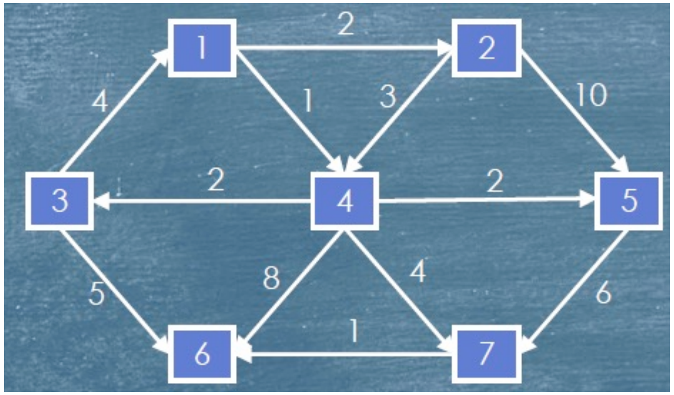
P =

1
4
1
4
7
4

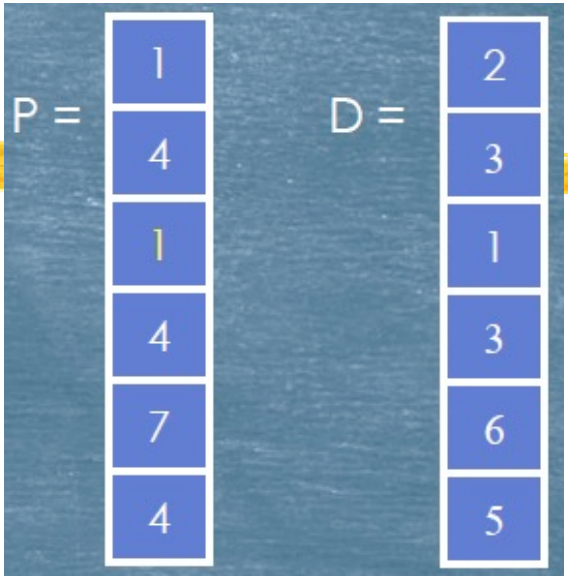
D =

2
3
1
3
6
5

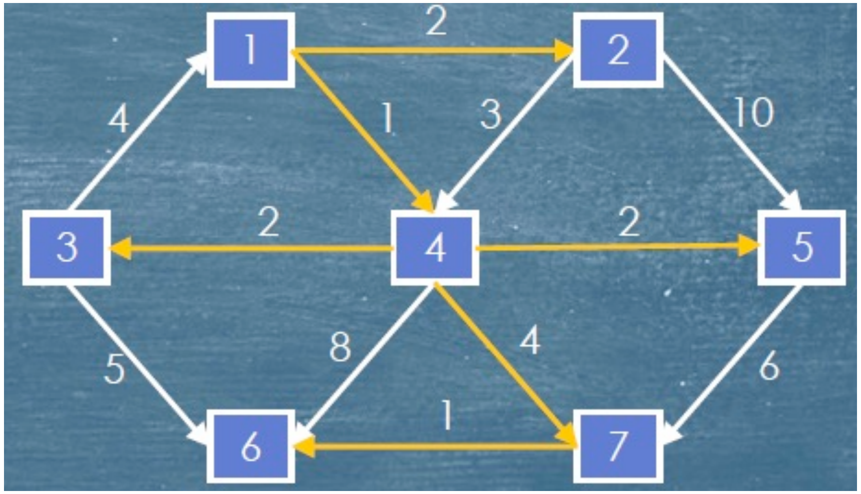
$C = \{6\}$
$S = \{1,4,2,3,5,7\}$
$v = 6$
$D(v) = 6$



- After step 6



- Paths from vertex 1:
  - To vertex 2
    - Path = (1, 2); W = 2
  - To vertex 3
    - Path = (1, 4, 3); W=3
  - To vertex 4
    - Path = (1, 4); W=1
  - To vertex 5
    - Path = (1, 4, 5); W=3
  - To vertex 6
    - Path = (1, 4, 7, 6); W = 6
  - To vertex 7
    - Path = (1, 4, 7); W = 5



Analysis

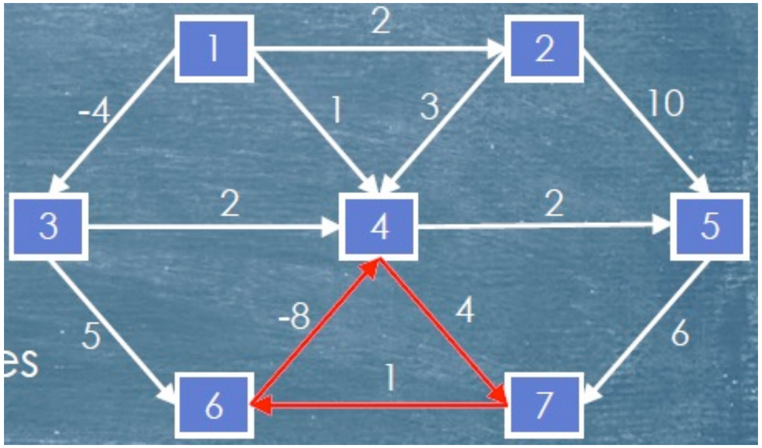
- The complexity of Dijkstra's Algorithm is $\Theta(V * \log V + E)$
  - How?
- Usually, $E > V$, in fact, in the worst case…
  - …$E \in \Theta(V^2)$—for a complete graph.
- There is one disadvantage:
  - The algorithm only works if all edge weights are non-negative.
- If we have negative edge weights, and especially negative edge cycles, we need a different algorithm.

The Bellman-Ford Algorithm

- Independently invented by both Bellman and Ford.
- Works with graphs that have negative edge weights.
- Identifies negative cycles and vertices with a negative cycle on their path.
- Finds correct path and path length for all other vertices.
- Let us look at an example graph…

A Graph with a negative cycle

- Consider the graph shown:
- Because the edges between vertices 4, 7 and 6 form a cycle whose total weight is -3, we can reduce the cost of any vertex with a path through any of these vertices as much as we like.
- Note that some vertices, namely 2 and 3, still have well defined minimum costs of 2 and -4 respectively.
- All other vertices have undefined minimum cost



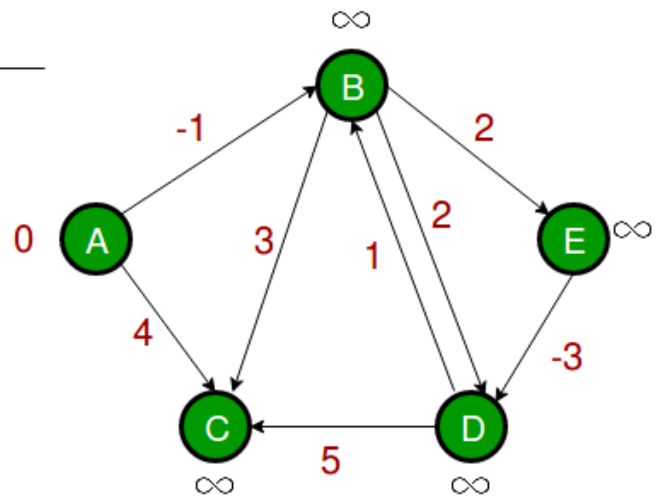
The Algorithm

- **Input:** Graph $G = (V, E, W)$ and a source vertex s
- **Output:** Shortest distance to all vertices from s. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.
  1. Initialization - $D[v] = \infty, v \in V, except\ V[s] = 0$
  2. Calculates shortest distances. Do following |V|-1 times
     - Do following for each edge u-v in E
       - If $D[v] > D[u] + W((u, v))$, then update $D[v]$
         $D[v] = D[u] + W((u, v))$
  3. Reports if there is a negative weight cycle in graph. Do following for each edge u-v in E
     - If $D[v] > D[u] + W((u, v))$, then "Graph contains negative weight cycle"

An Example

- Initialization

A	B	C	D	E
0	∞	∞	∞	∞

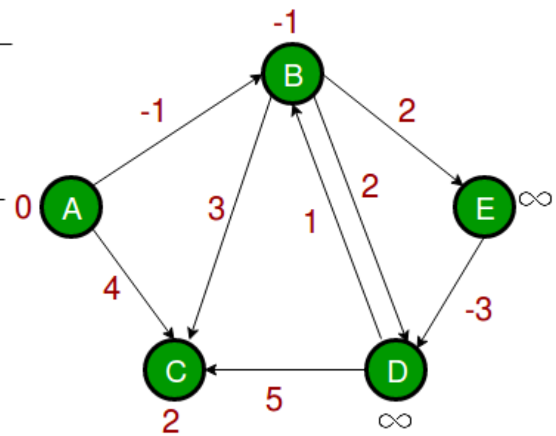


- Let all edges are processed in the following order:
  - (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

An Example

- Step 1 – Iteration #1

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



(B,E):

$D(E) > D(B) + W(B,E)$?

(D,B):

$D(B) > D(D) + W(D,B)$?

(B,D):

$D(D) > D(B) + W(B,D)$?

(A,B):

$D(B) > D(A) + W(A,B)$? Yes

$D(B) = D(A) + W(A,B) = -1$

(A,C):

$D(C) > D(A) + W(A,C)$? *Yes*

$D(C) = D(A) + W(A,C) = 4$

(D,C):

$D(C) > D(D) + W(D,C)$?

(B,C):

$D(C) > D(B) + W(B,C)$? *yes*

$D(C) = D(B) + W(B,C) = 2$

(E,D):

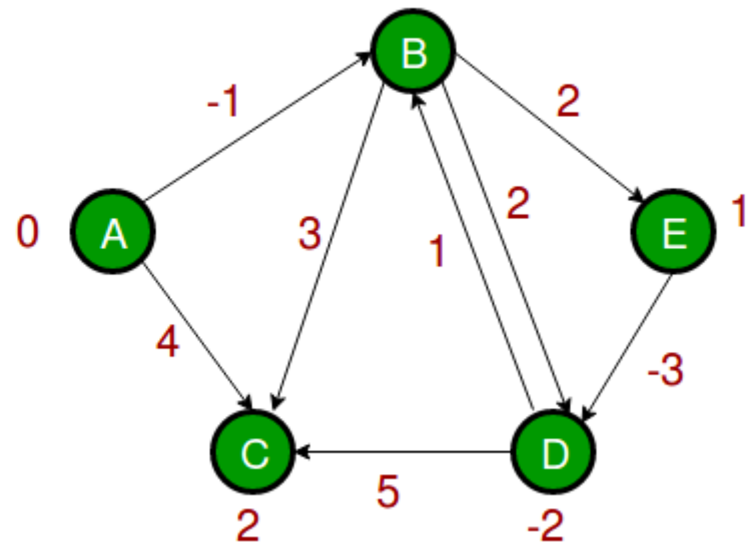
$D(D) > D(E) + W(E,D)$?

- Let all edges are processed in the following order:
  - (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

An Example

- Step 1 – Iteration #2

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



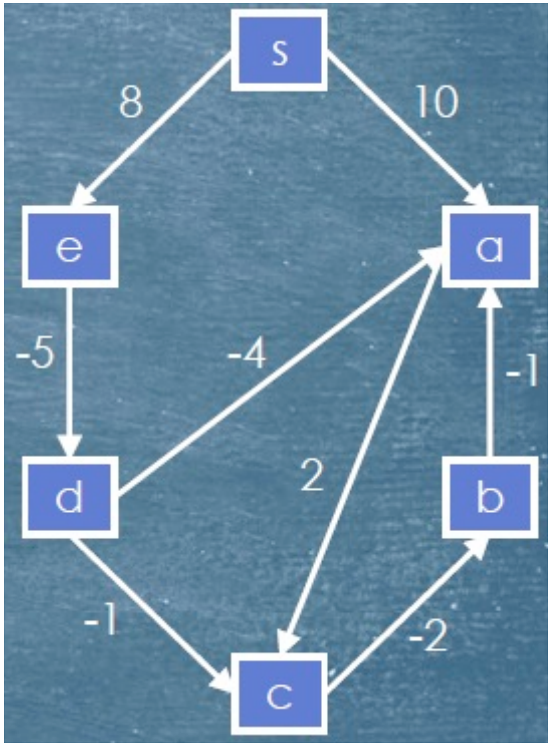
- Let all edges are processed in the following order:
  - (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

Different from Dijkstra's

- Unlike Dijkstra's algorithm, in which we update only the most promising (next lowest cost) vertex at each iteration, Bellman-Ford updates every vertex at each iteration.
- This means that each iteration of Bellman-Ford involves more work than the corresponding iteration of Dijkstra

Another Example

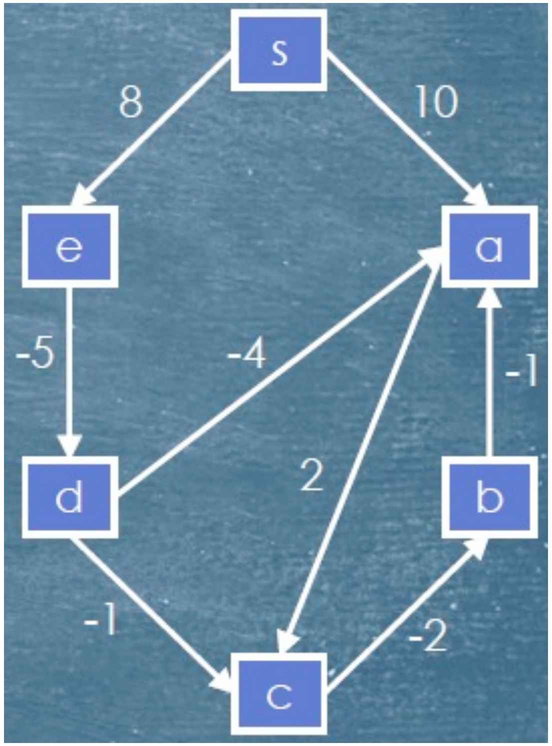
- Consider the following graph:
- It has 6 vertices so we will run through the main loop 5 times.
- Order of edges
  - $(u, v), u = s, a, b, c, d, e$



Initialization

Set initial values of D

s	a	b	c	d	e
0	∞	∞	∞	∞	∞

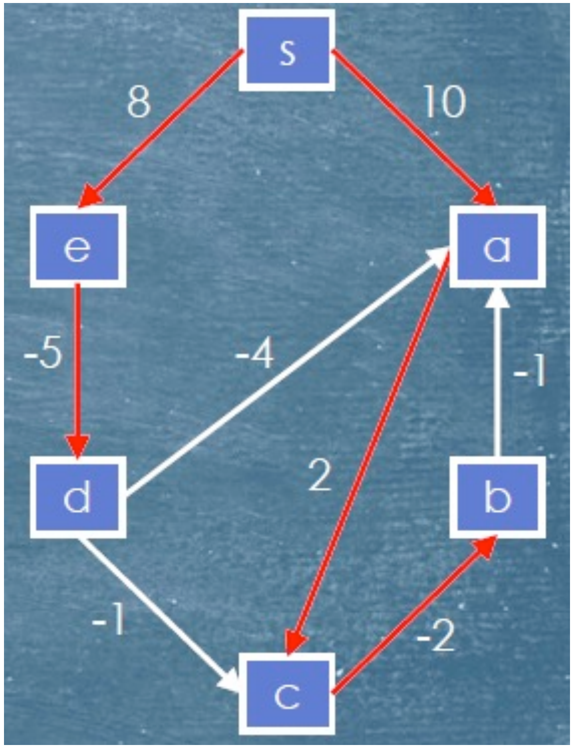


Step 2 - iteration #1

```
(s,a),(s,e) – update D[a],D[e]
(a,c) – update D[c]
(b,a) – no update
(c,b)- update D[b]
(d,a),(d,c) – no update
(e,d) – update D[d]
```

s	a	b	c	d	e
0	∞	∞	∞	∞	∞

s	a	b	c	d	e
0	10	10	12	3	8

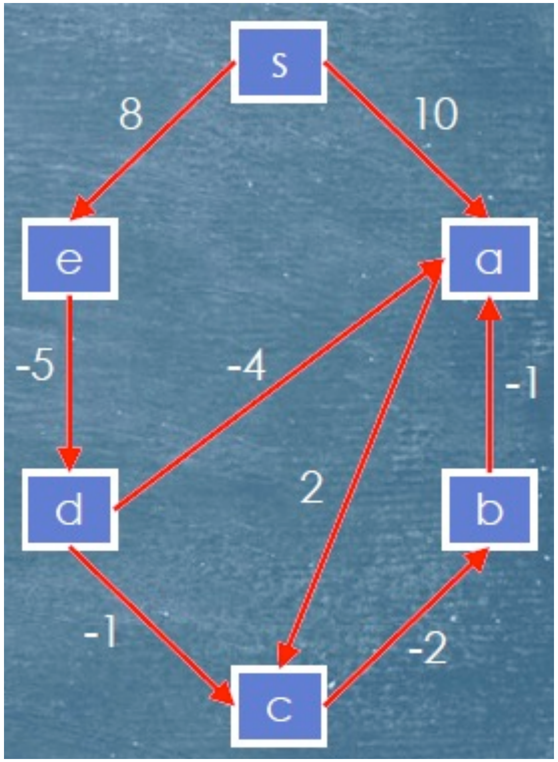


Step 2 - iteration #2

```
(s,a),(s,e) – no update
(a,c) – no update
(b,a) – no update
(c,b)- no update
(d,a),(d,c) – update D[a],D[c]
(e,d) – no update
```

s	a	b	c	d	e
0	10	10	12	3	8

s	a	b	c	d	e
0	−1	10	2	3	8

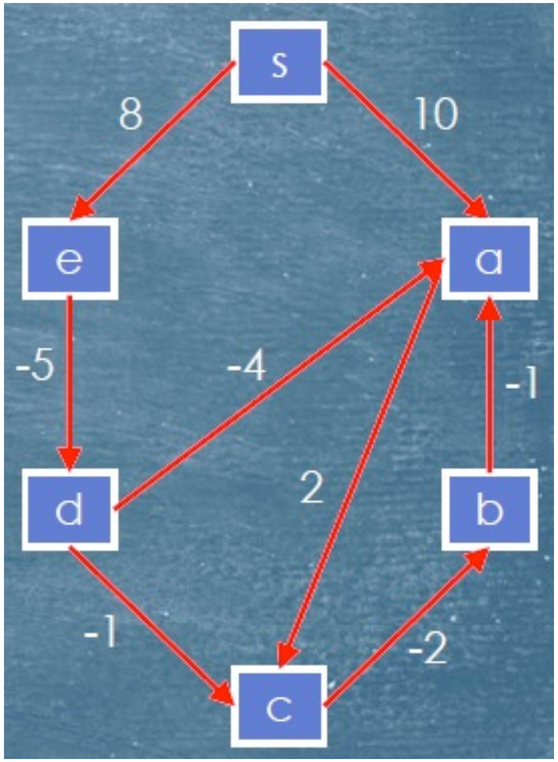


Step 2 - iteration #3

```
(s,a),(s,e) – no update
(a,c) – update D[c]
(b,a) – no update
(c,b)- update D[b]
(d,a),(d,c) – no update
(e,d) – no update
```

s	a	b	c	d	e
0	−1	10	2	3	8

s	a	b	c	d	e
0	−1	−1	1	3	8

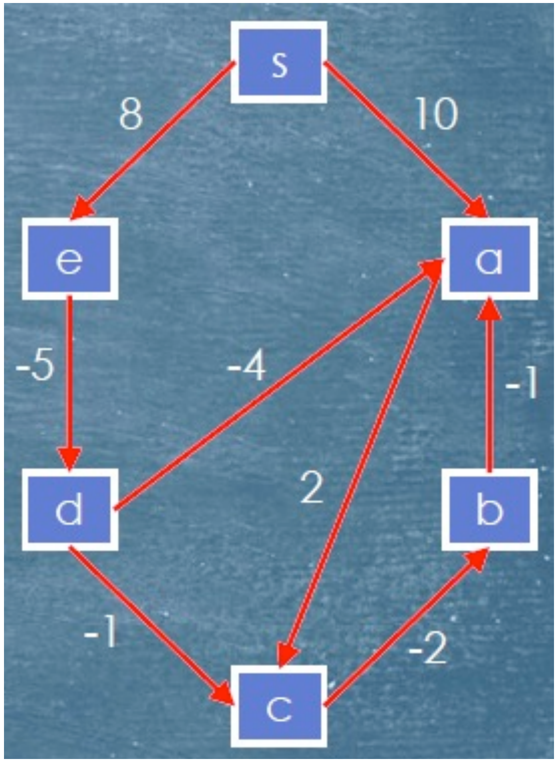


Step 2 - iteration #4

```
(s,a),(s,e) – no update
(a,c) – no update
(b,a) – update D[a]
(c,b)- no update
(d,a),(d,c) – no update
(e,d) – no update
```

s	a	b	c	d	e
0	−1	−1	1	3	8

s	a	b	c	d	e
0	−2	−1	1	3	8

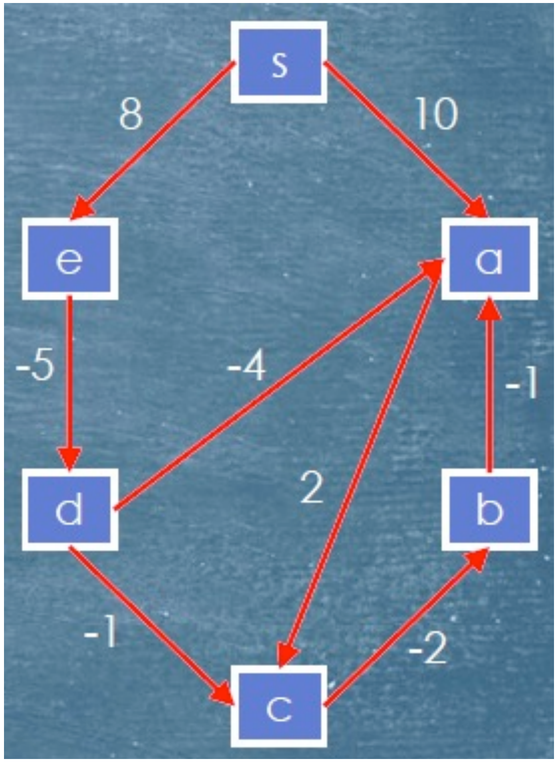


Step 2 - iteration #5

```
(s,a),(s,e) – no update
(a,c) – update D[c]
(b,a) – no update
(c,b)- update D[b]
(d,a),(d,c) – no update
(e,d) – no update
```

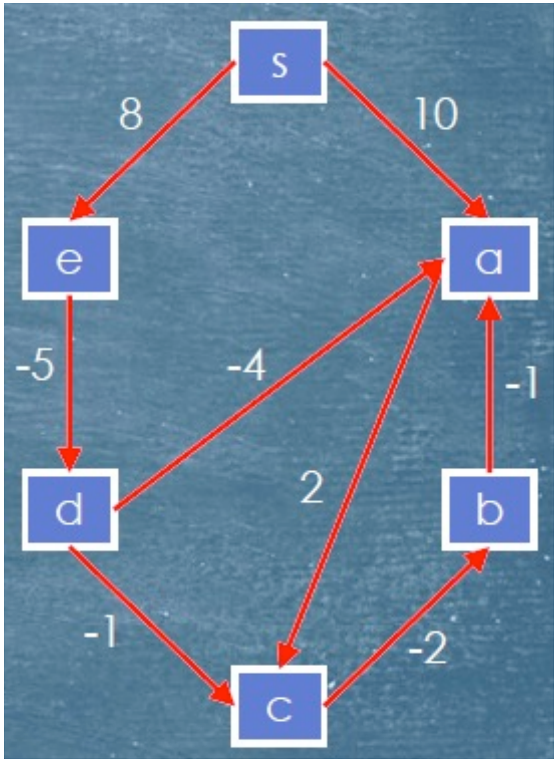
s	a	b	c	d	e
0	−2	−1	1	3	8

s	a	b	c	d	e
0	−2	−2	0	3	8



Step 3 - Check Negative Weight Cycle

```
(s,a) – D[a]>D[s]+w(s,a)?
(s,e) – D[e]>D[s]+w(s,e)?
(a,c) – D[c]>D[a]+w(a,c)?
(b,a) – D[a]>D[b]+w(b,a)? yes
(c,b) – D[b]>D[c]+w(c,b)?
(d,a) – D[a]>D[d]+w[d,a)?
(d,c) – D[c]>D[d]+w(d,c)?
(e,d) – D[d]>D[e]+w(e,d)?
```



The graph contains a negative cost cycle that involves vertex a.

s	a	b	c	d	e
0	−2	−2	0	3	8

Analysis

- Bellman-Ford performs the major loop $|V - 1|$ times.
- Inside this loop it checks every edge; $|E|$ operations.
- Finally, it does another $|E|$ checks for potential cycles.
- Overall, Bellman-Ford has $\Theta(|V| \times |E|)$ complexity.

Notes on Bellman-Ford Algorithm

- Bellman-Ford algorithm can handle directed and undirected graphs with non-negative weights.
- However, it can only handle directed graphs with negative weights, as long as we don't have negative cycles.
- When the graph has a negative cycle, Bellman-Ford algorithm can detect the cycle, but won't be able to find the shortest paths in this case.

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 9.3

- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 24.1 and 24.3