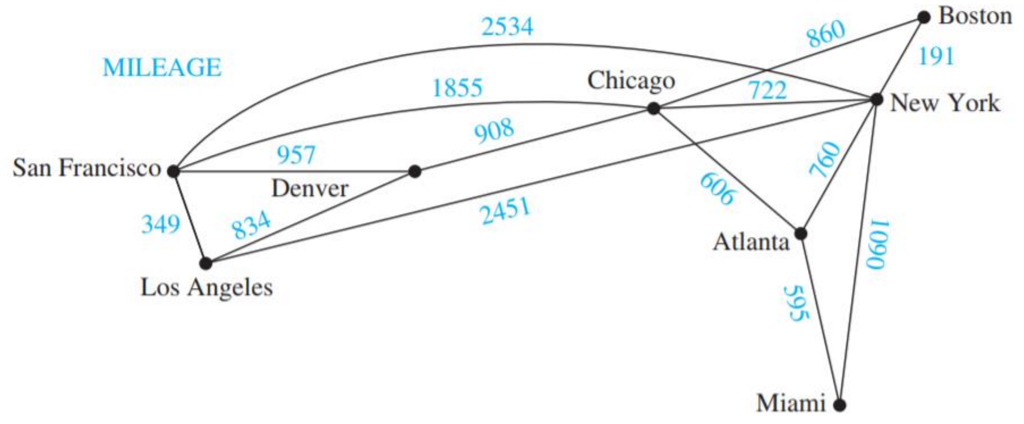
Single Source Shortest Path

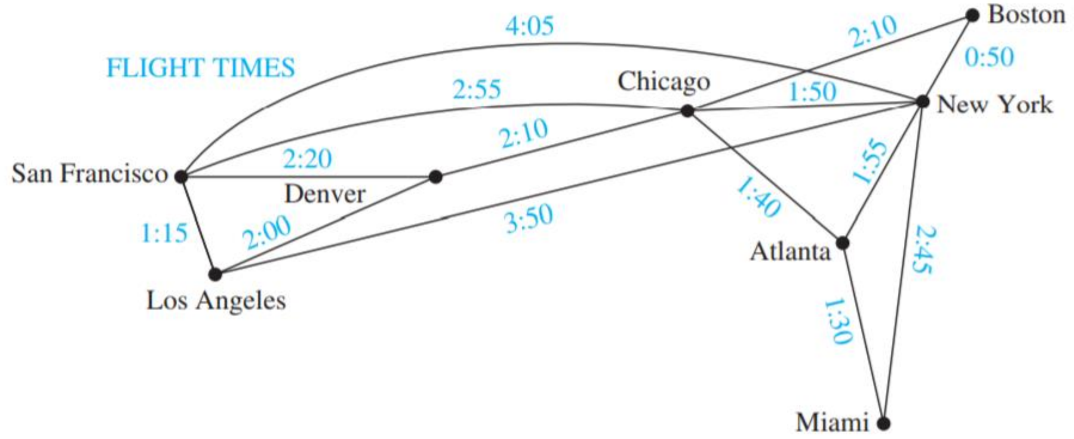
Shortest Path Problems

Weighted Graphs

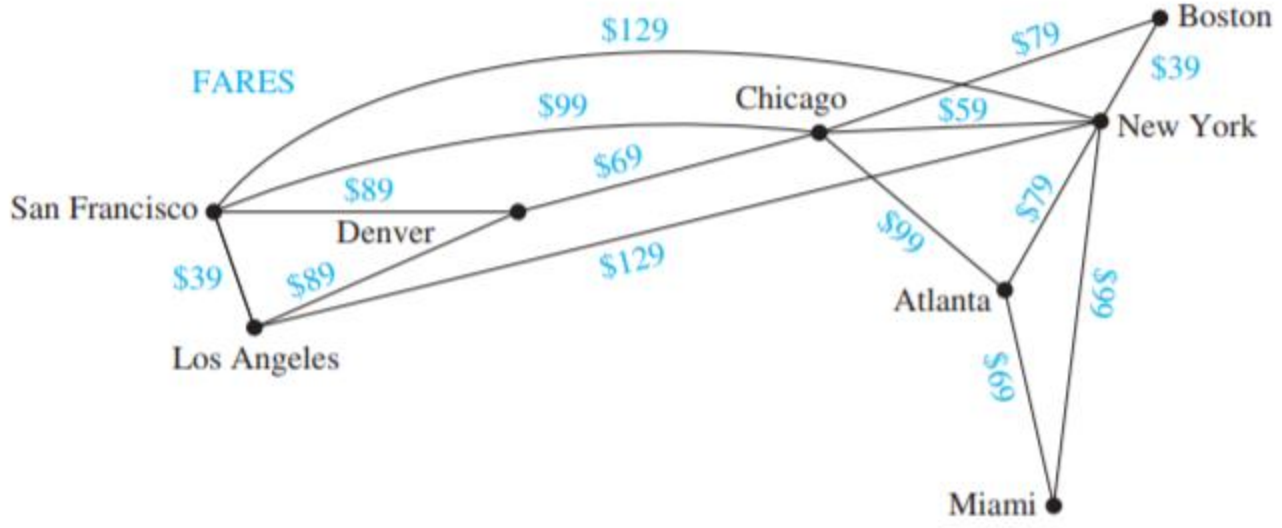
- Many problems can be modeled using graphs with **weights** assigned to their edges
- representing cities by vertices and flights by edges
- assigning distances between cities to the edges
- assigning flight times to edges
- assigning fares to the edges

Weighted Graphs





Weighted Graphs



Weighted Graphs

- Graphs that have a number assigned to each edge

Used to model:

- Communication networks
- Communication costs
- Response times of the computers over these lines
- Distance between computers

Shortest Path Algorithm

- Generalize distance to weighted setting
- Digraph G = (V,E) with weight function W: E → R (assigning real values to edges)
- Weight of path *p = v1 → v2 → … → vk* is

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

- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest Path Problems

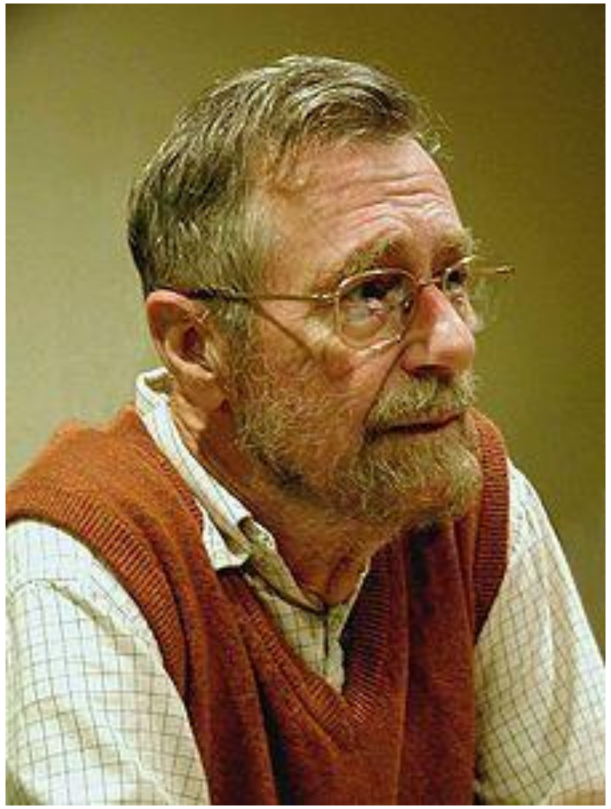
- **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. *The topic of this lecture.*
- **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
- **Unweighted shortest-paths** – BFS.

For this…

We’ll have Dijkstra’s Algorithm as an example.

Dijkstra’s Algorithm

Edsger Wybe Dijkstra



May 11, 1930 – August 6, 2002

Dutch computer scientist from Netherlands

Received the 1972 A. M. Turing Award

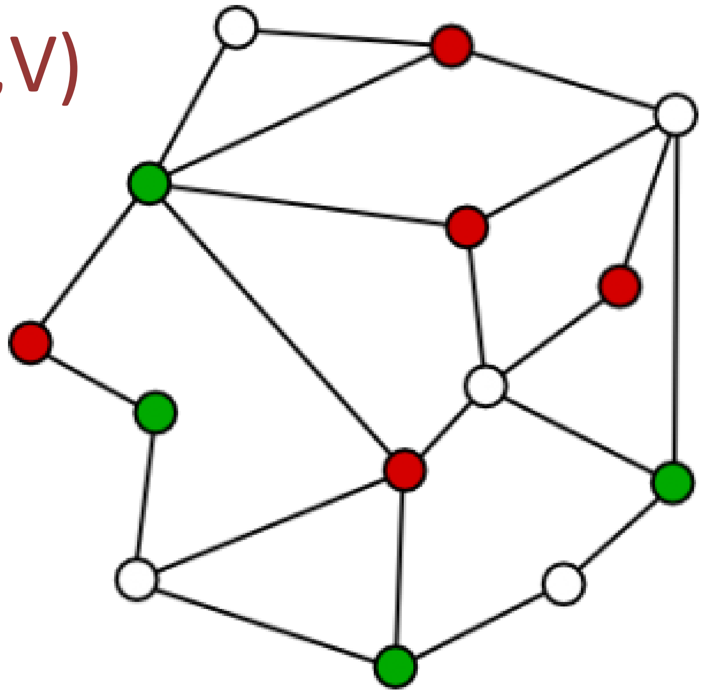
Known for his many essays on programming

Single-Source Single Path Problem

- The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.

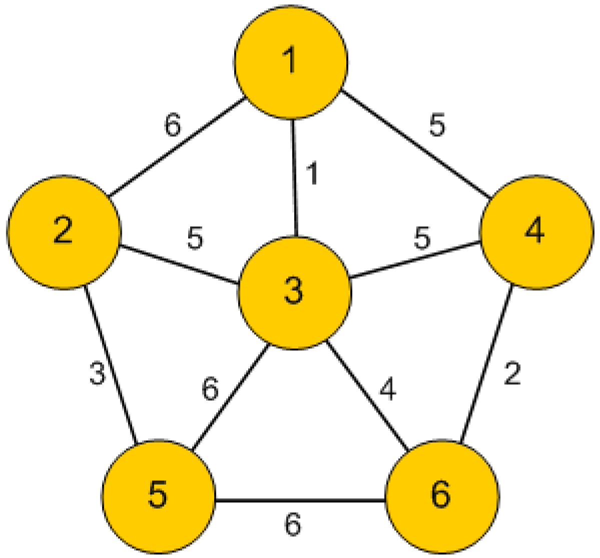
Weighted graph G = (E,V)

Source vertex s ∈ V to all vertices v ∈ V



Solution to Single-Source Single Path Problem

- Works on both **directed** and **undirected** graphs.
- All edges must have nonnegative weights.
- Graph must be connected

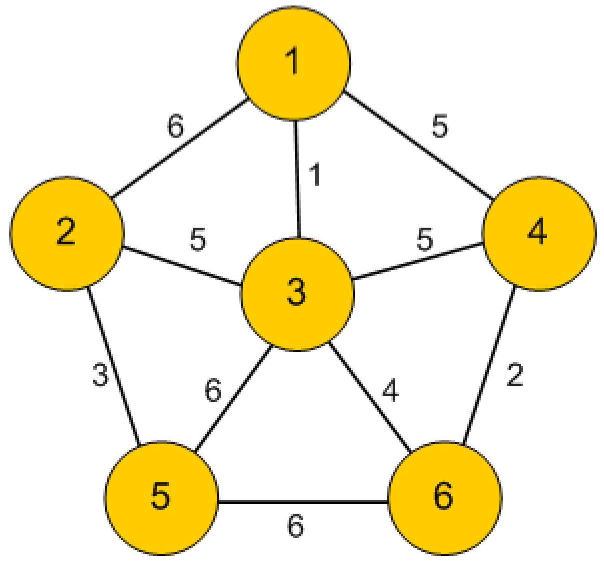


Output of Dijkstra’s Algorithm

Original algorithm outputs value of shortest path not the path itself

Value: $\delta(1,5) = 7$

Path: $\{1,3,5\}$



Implementation

Edge Relaxation

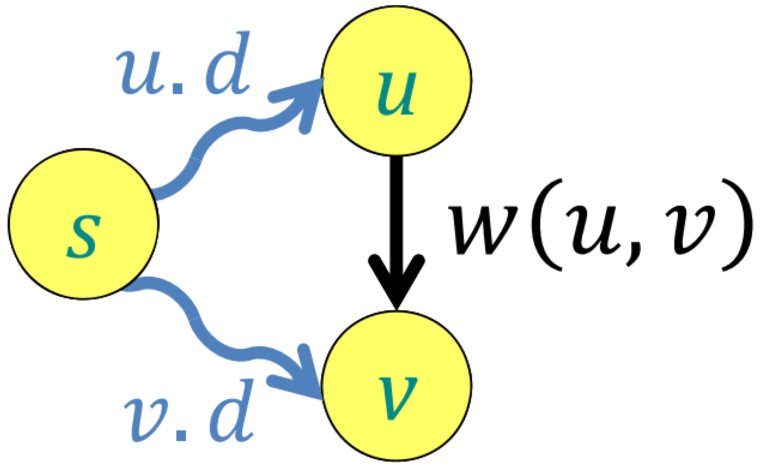
- Relaxing an edge, (a concept you can find in other shortest-path algorithms as well) is trying to lower the cost of getting to a vertex by using another vertex.

Edge Relaxation

Consider an edge (u,v)

If D[v] > D[u]+w(u,v) then

D[v]=D[u]+w[u,v]



Edge Relaxation

Maintain value D[u] for each vertex

- Each starts at infinity, and decreases as we find out about a shorter path from v to u (D[v] = 0)

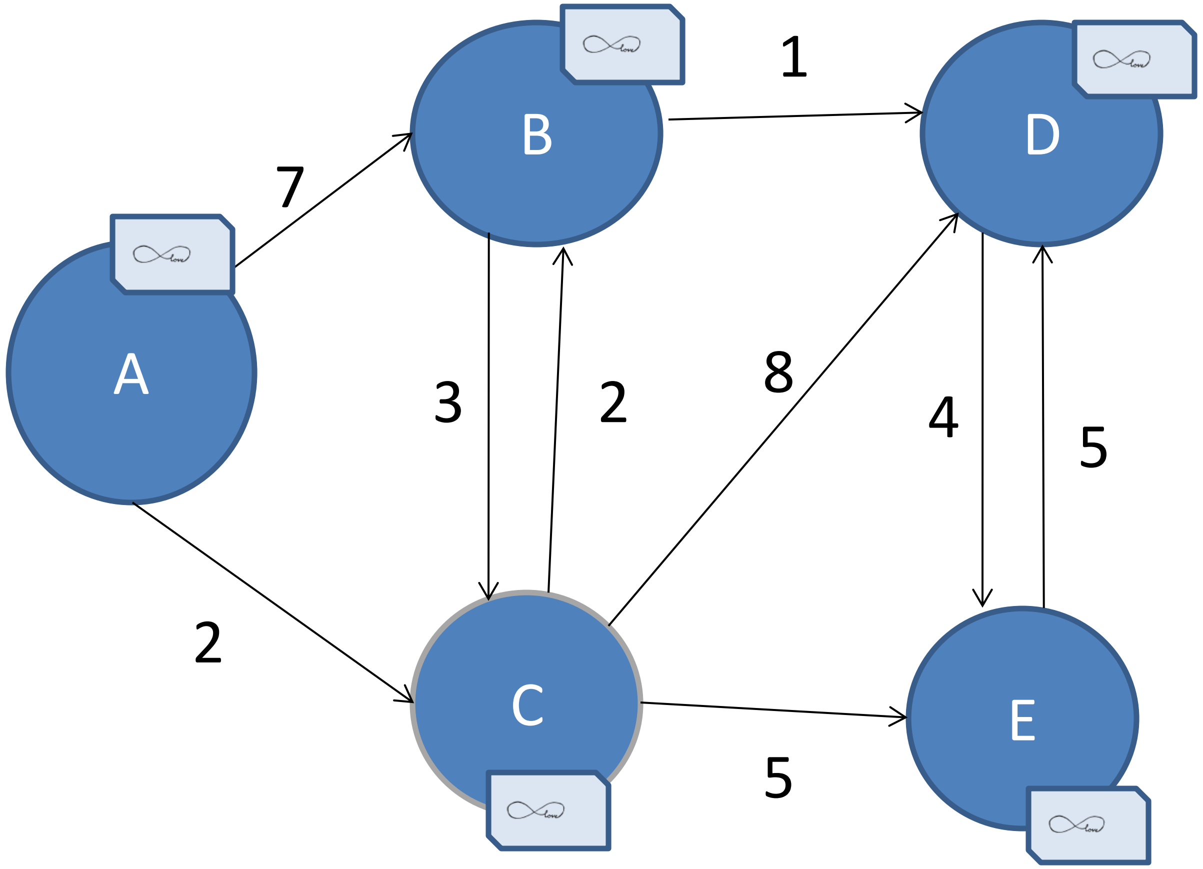
Maintain priority queue, Q, of vertices to be relaxed

- Use D[u] as key for each vertex
- Remove min vertex from Q, and relax its neighbors

Dijkstra’s

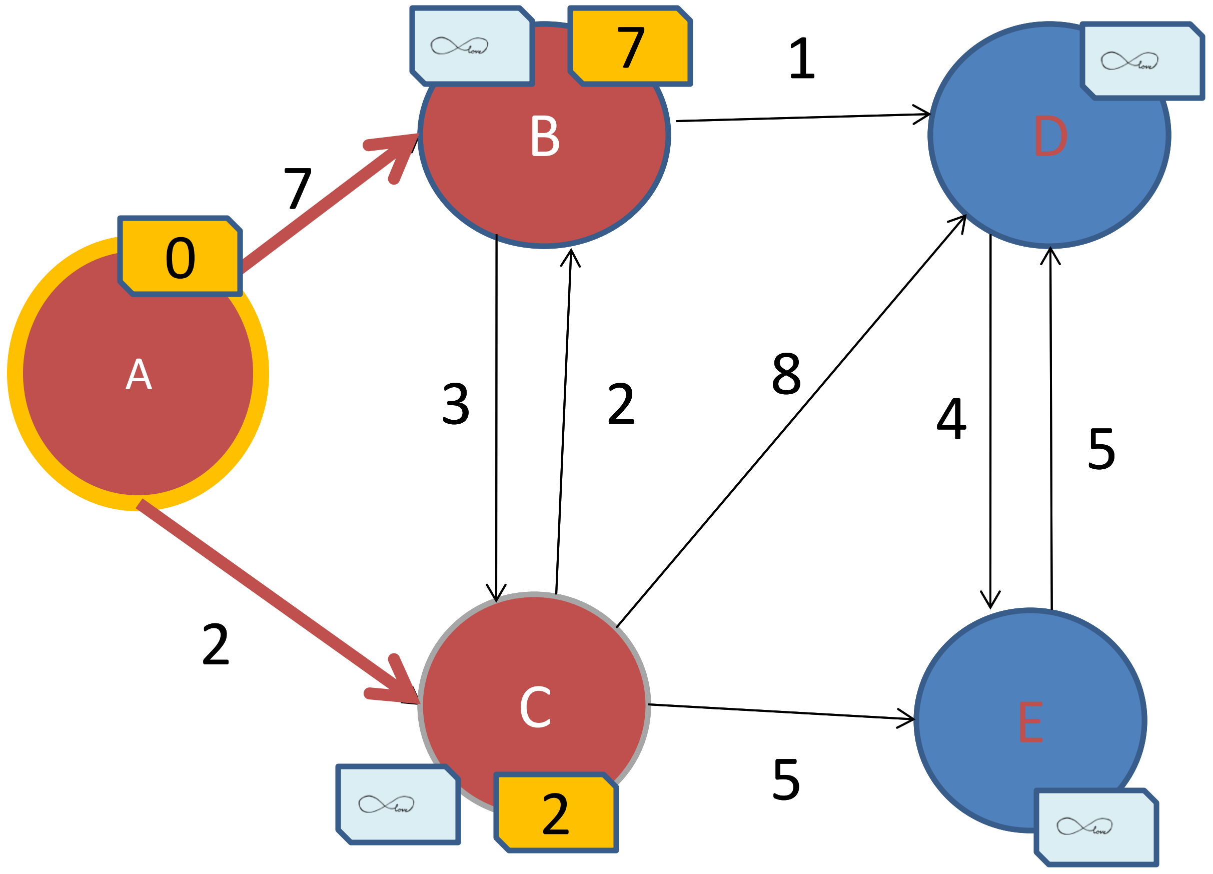
```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                      (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
      S←SU{u}                               (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                         then d[v] ←d[u] + w(u, v)          (set new value of shortest path)
                                                  (if desired, add traceback code)

return dist
```

Shortest Path



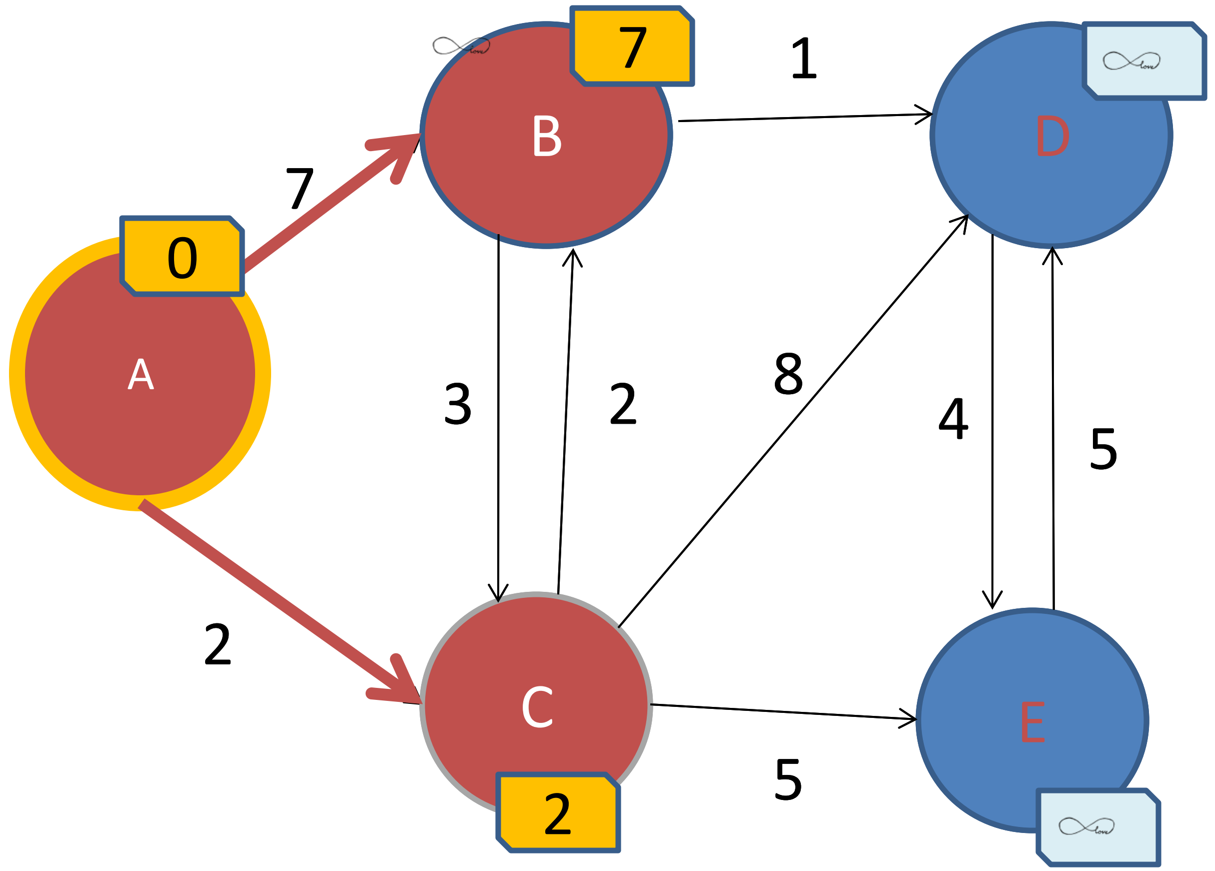
	A	B	C	D	E
	∞	∞	∞	∞	∞

Shortest Path



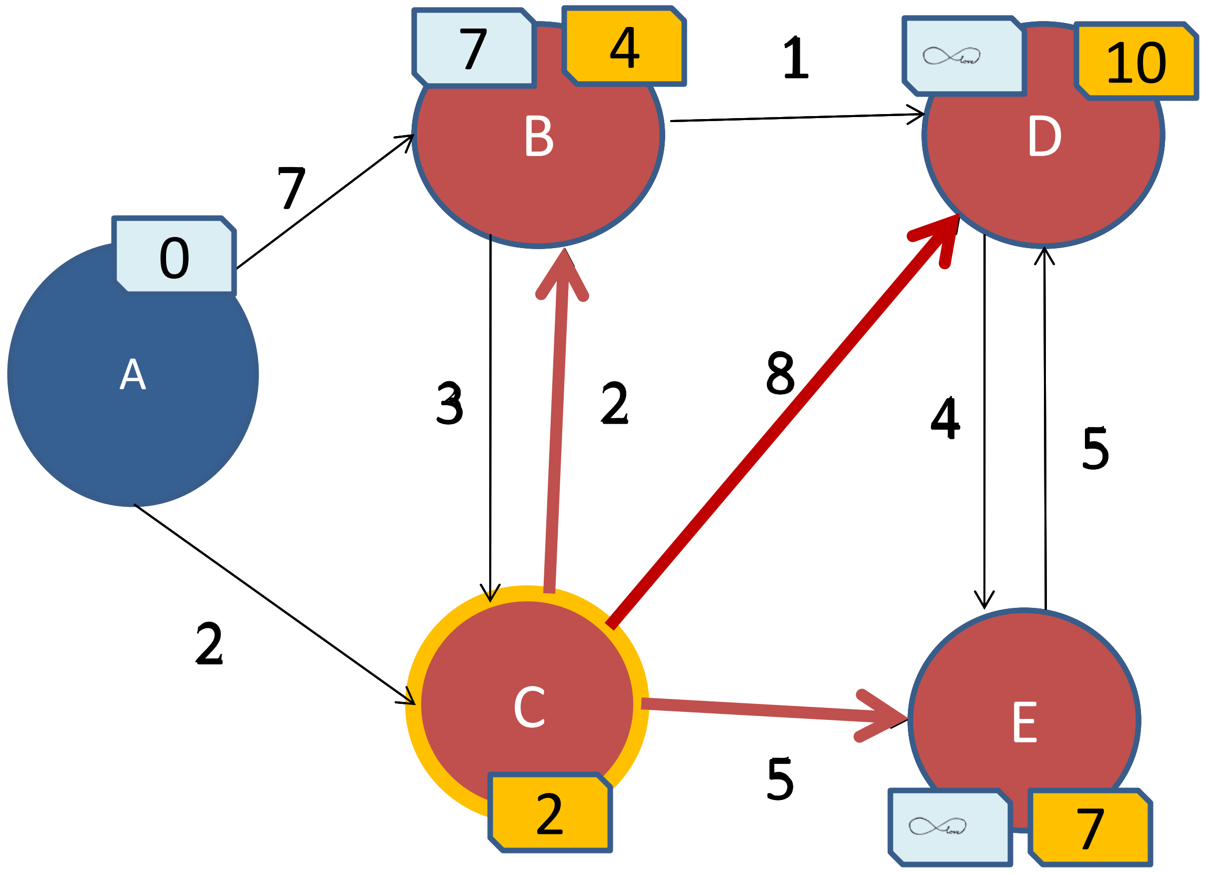
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞

Shortest Path



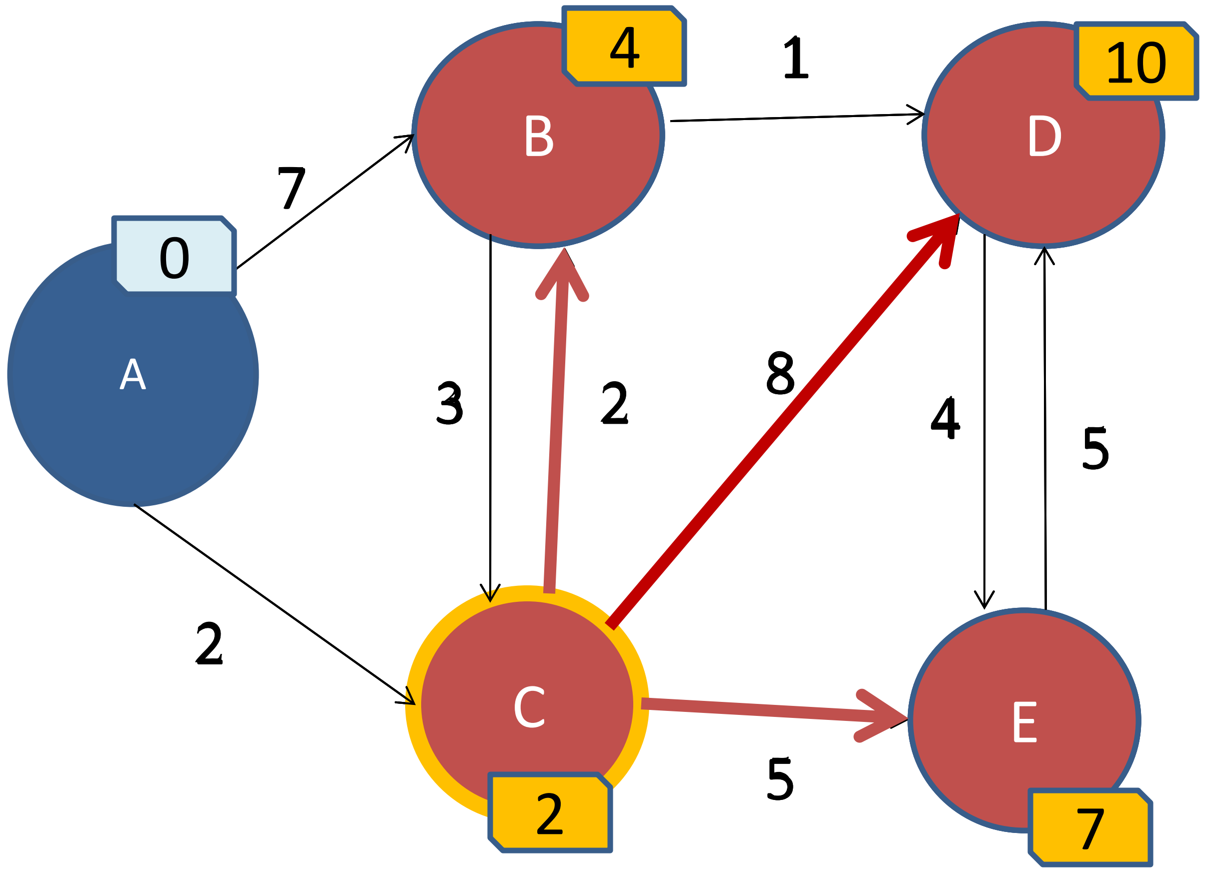
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞

Shortest Path



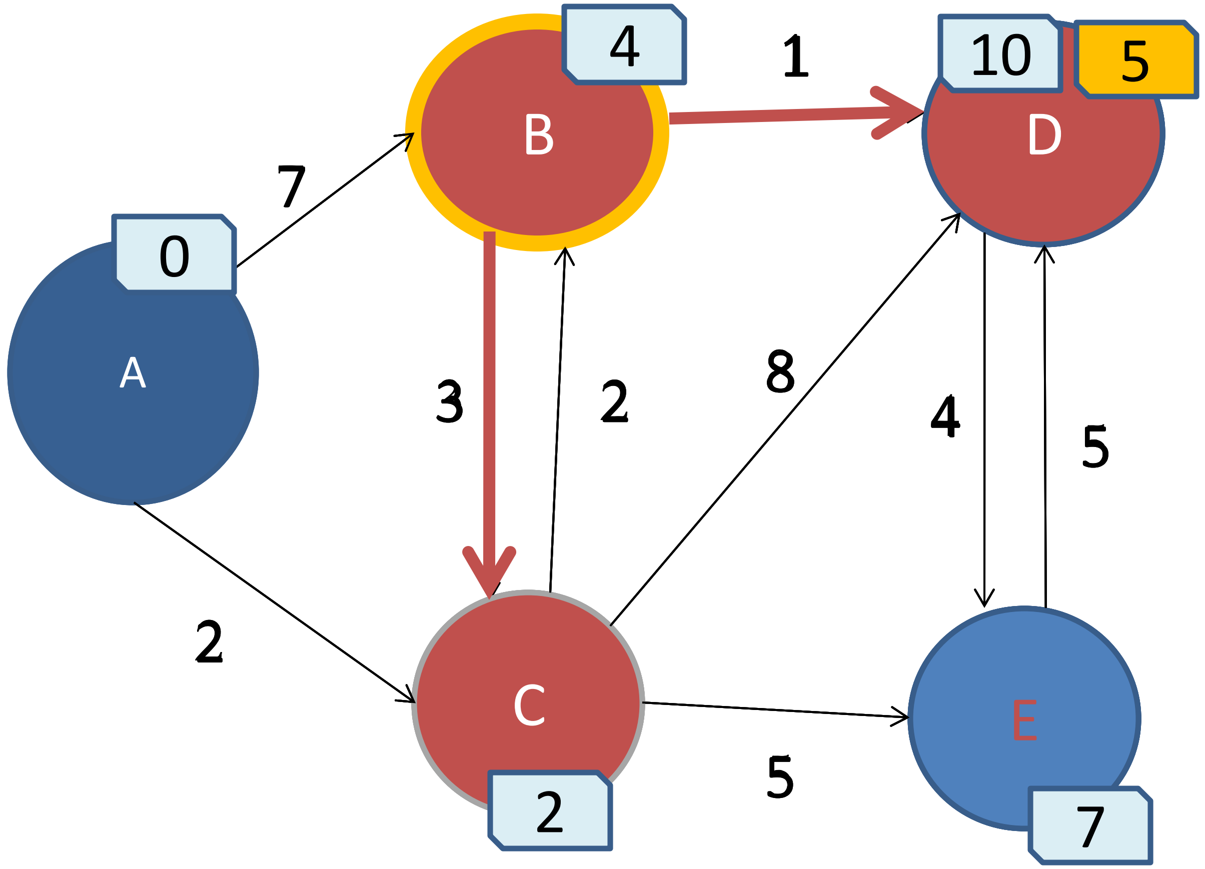
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C

Shortest Path



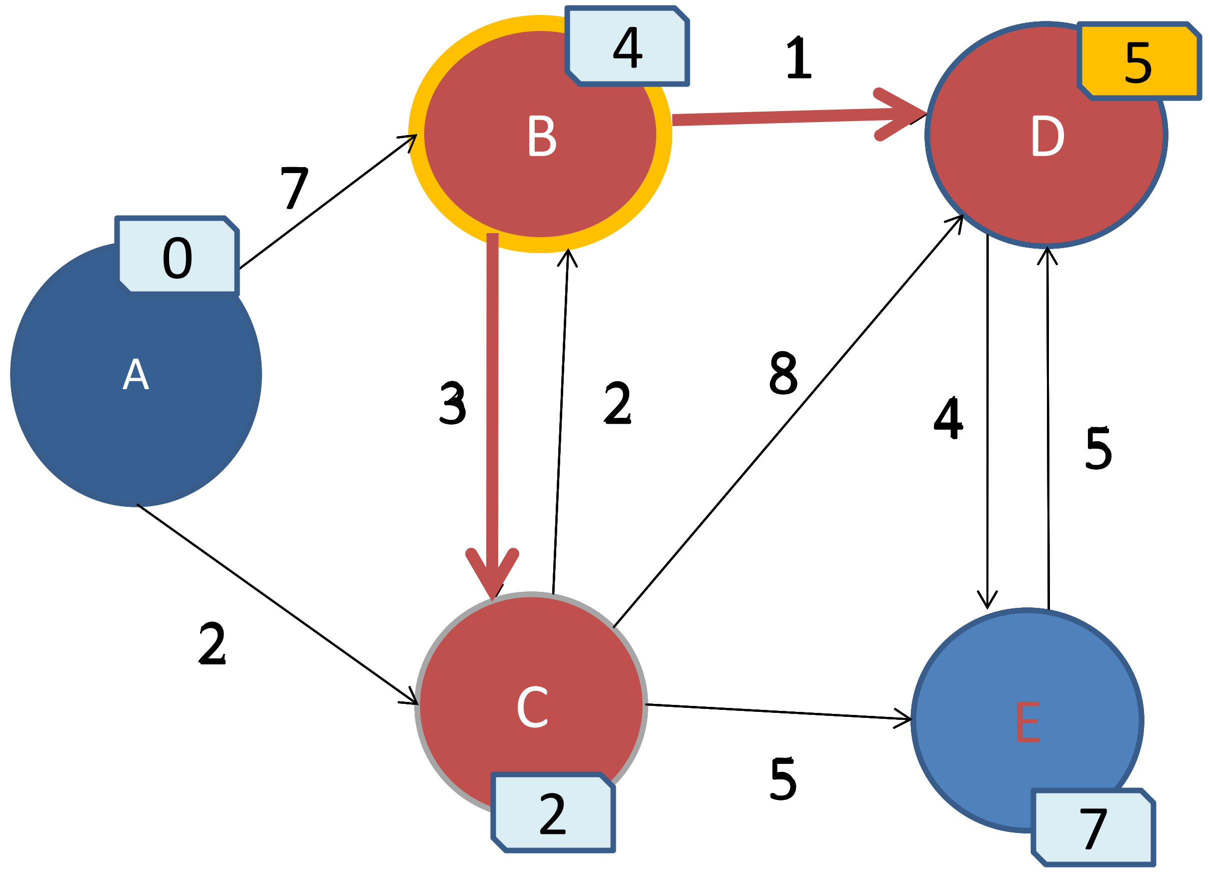
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C

Shortest Path



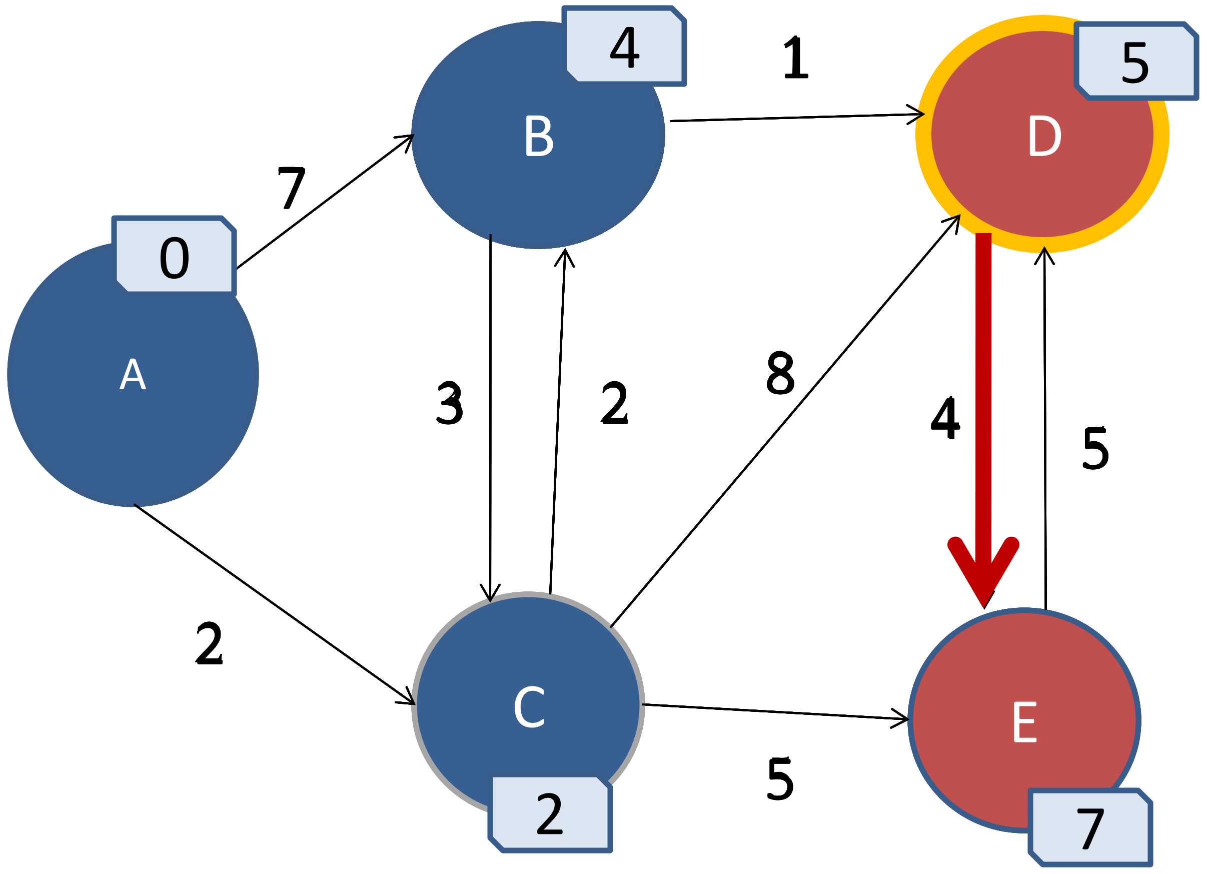
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C

Shortest Path



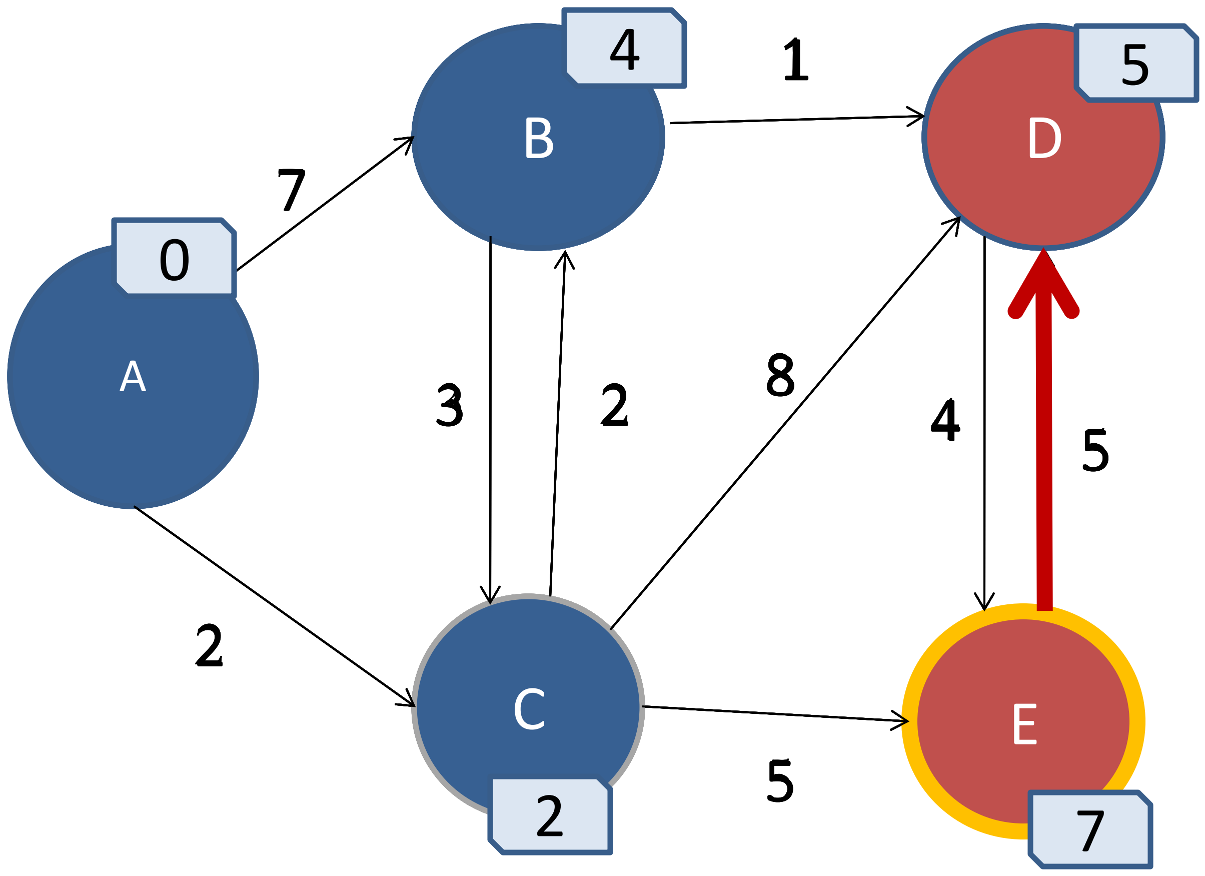
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C

Shortest Path



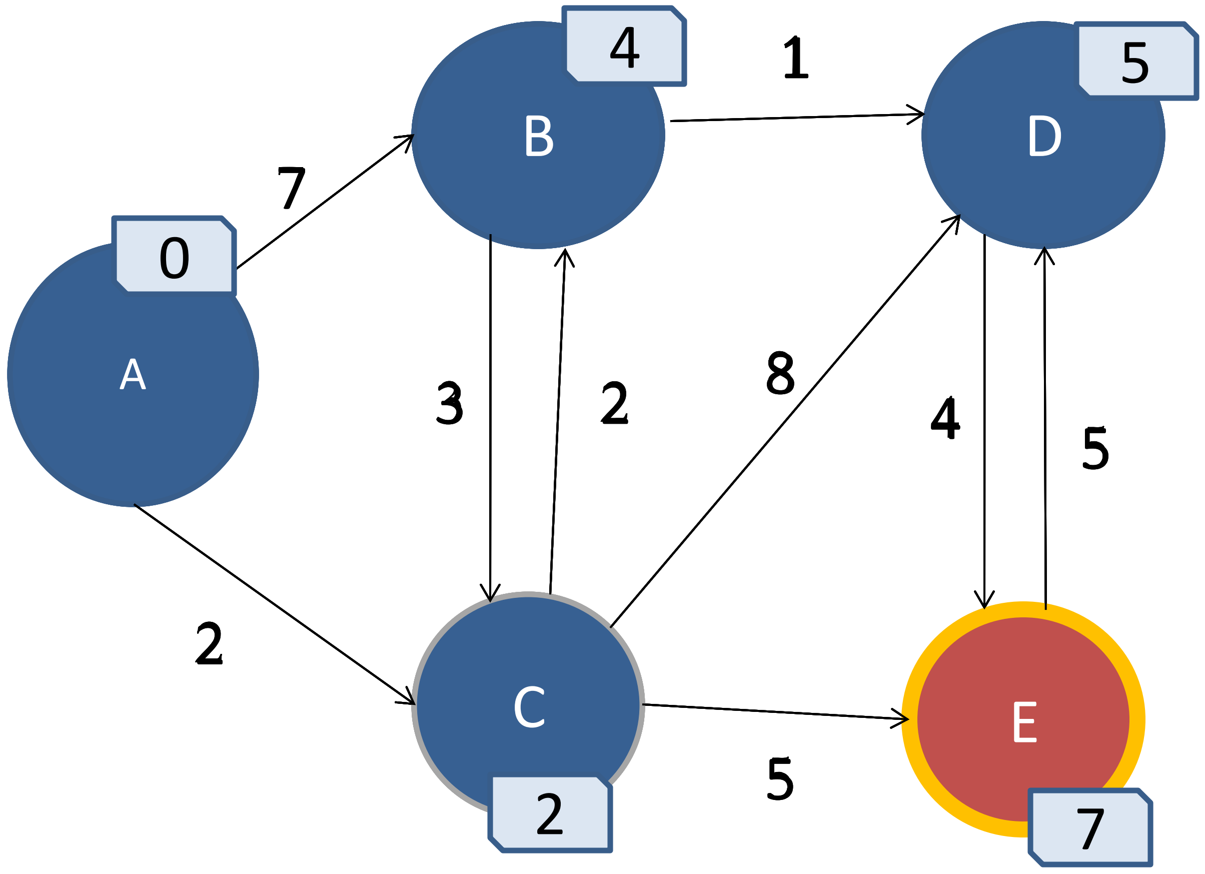
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C

Shortest Path



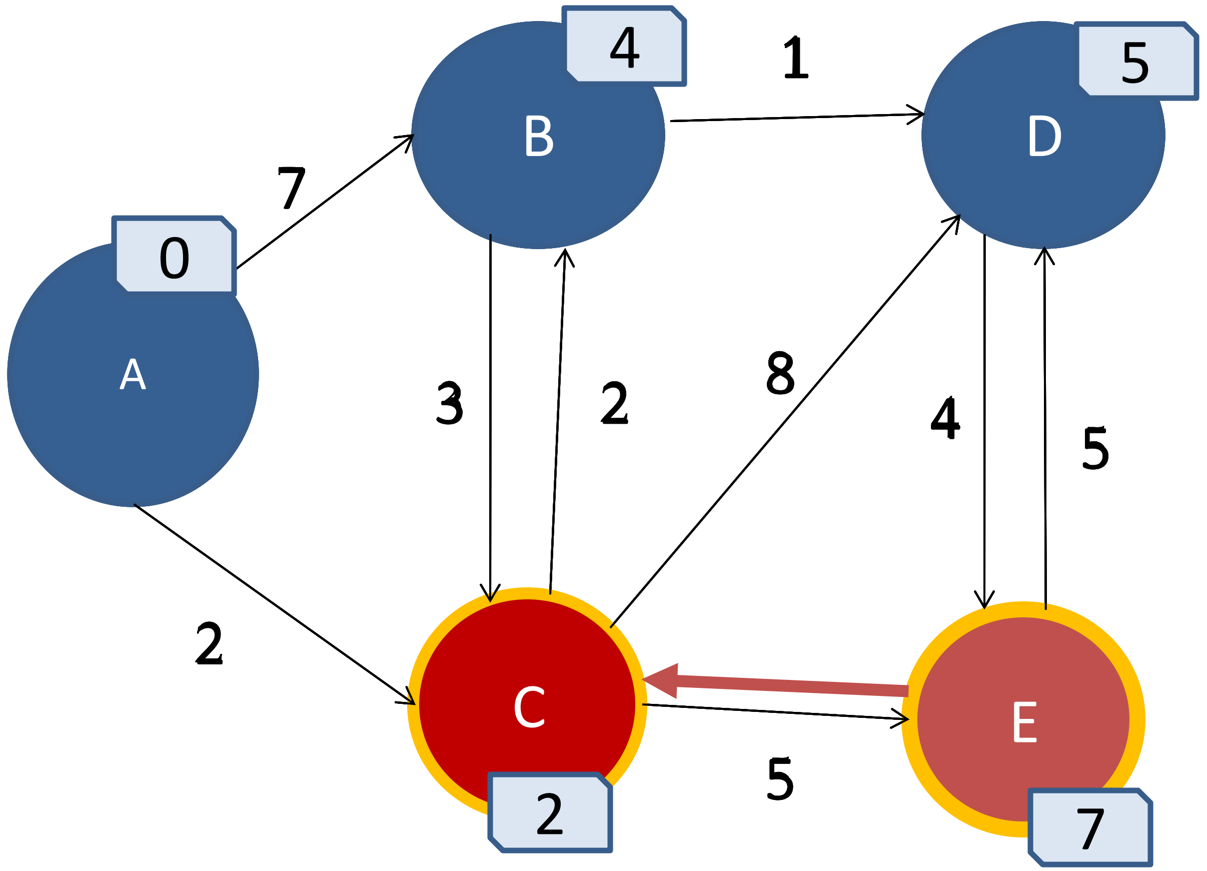
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Shortest Path



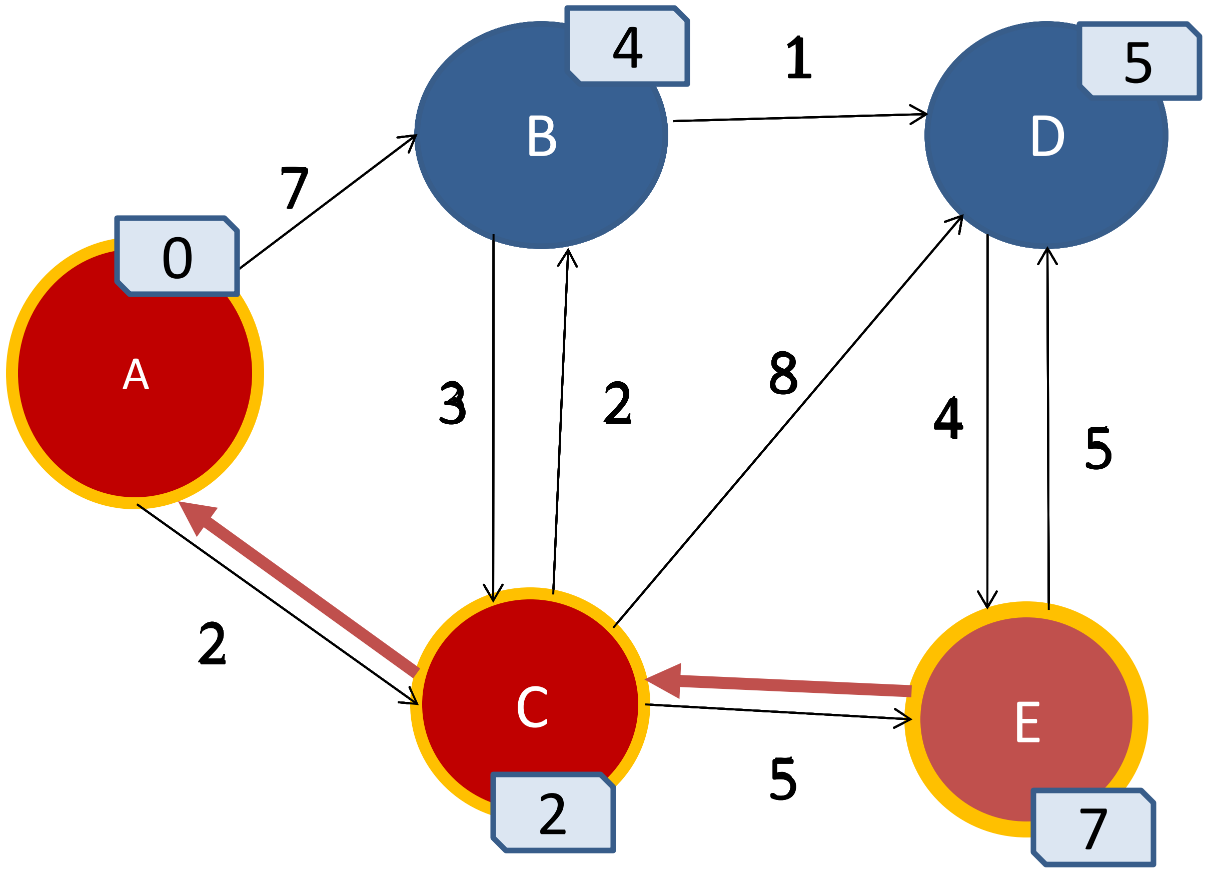
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Shortest Path



	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Shortest Path

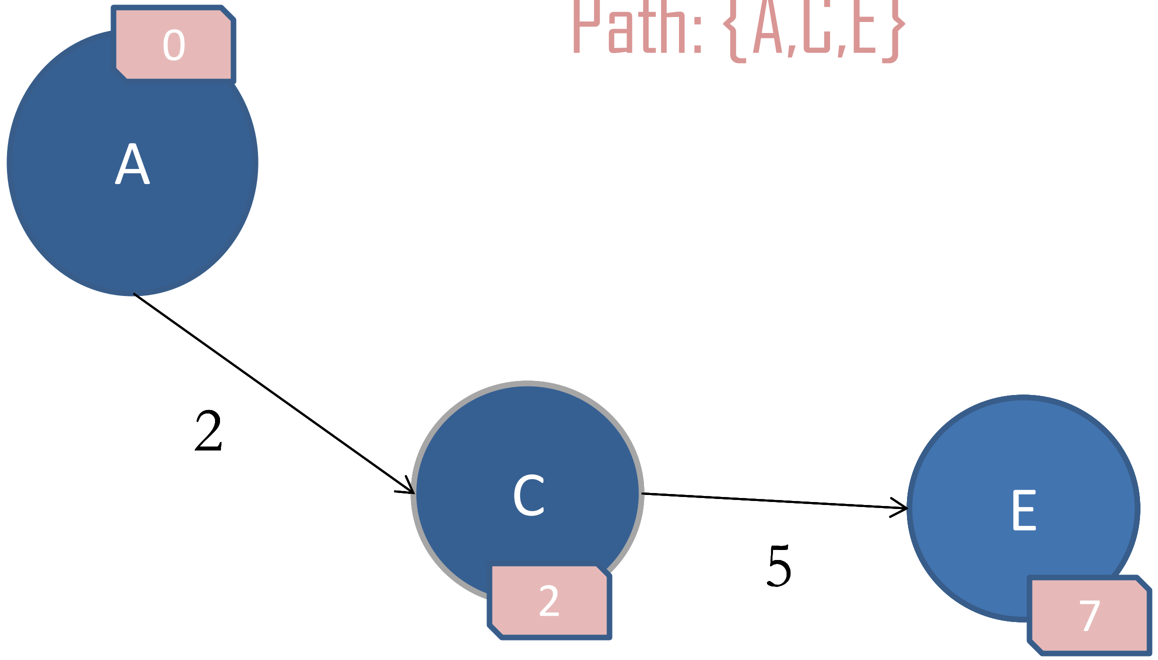


	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

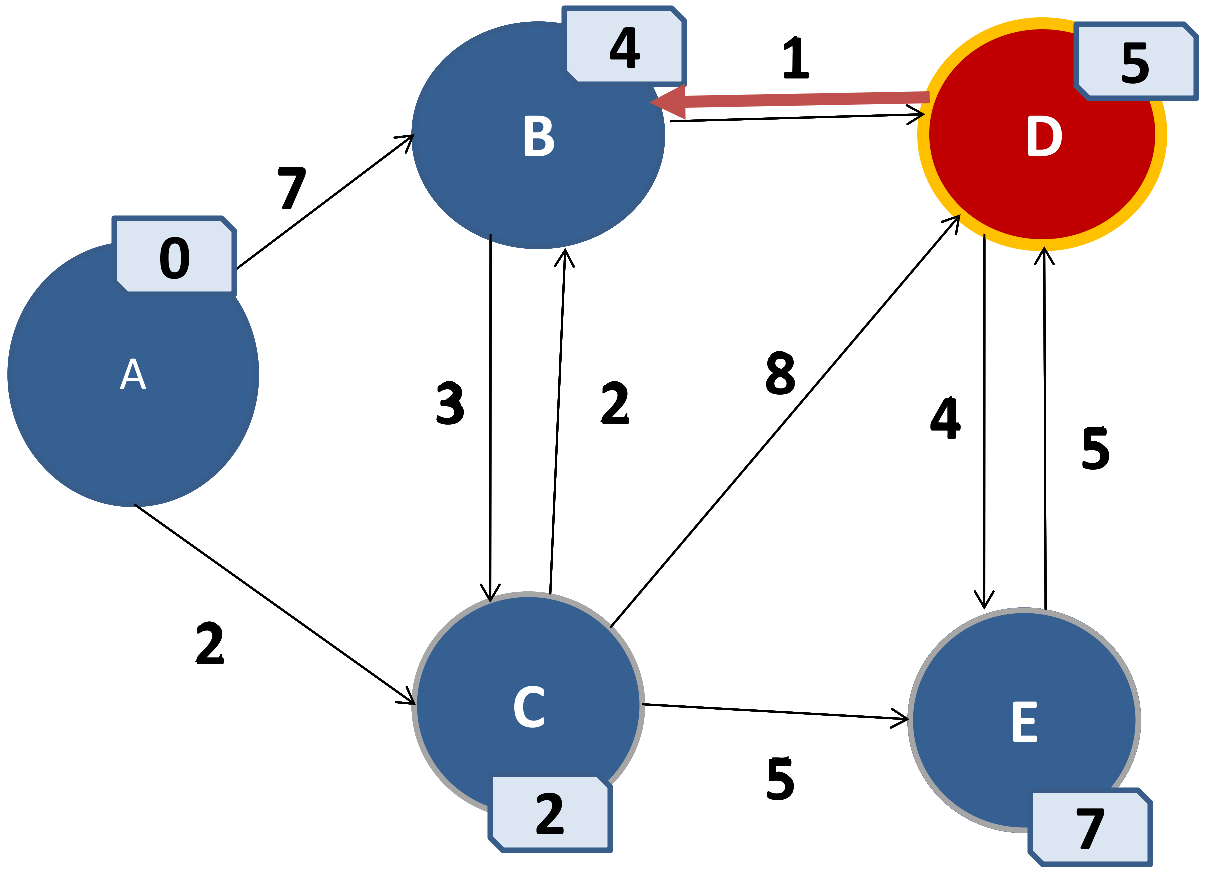
Solution

Value: $\delta(A,E) = 7$

Path: {A,C,E}

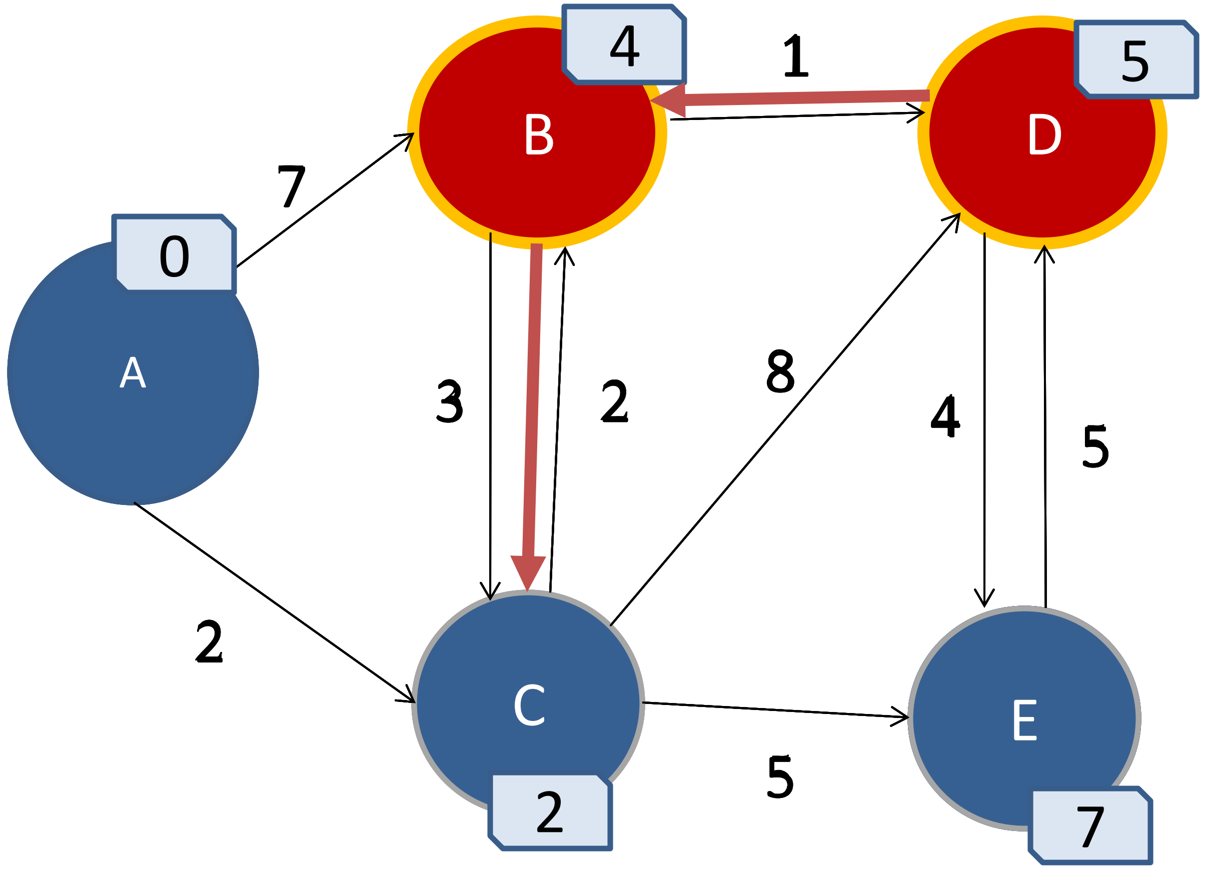


Shortest Path



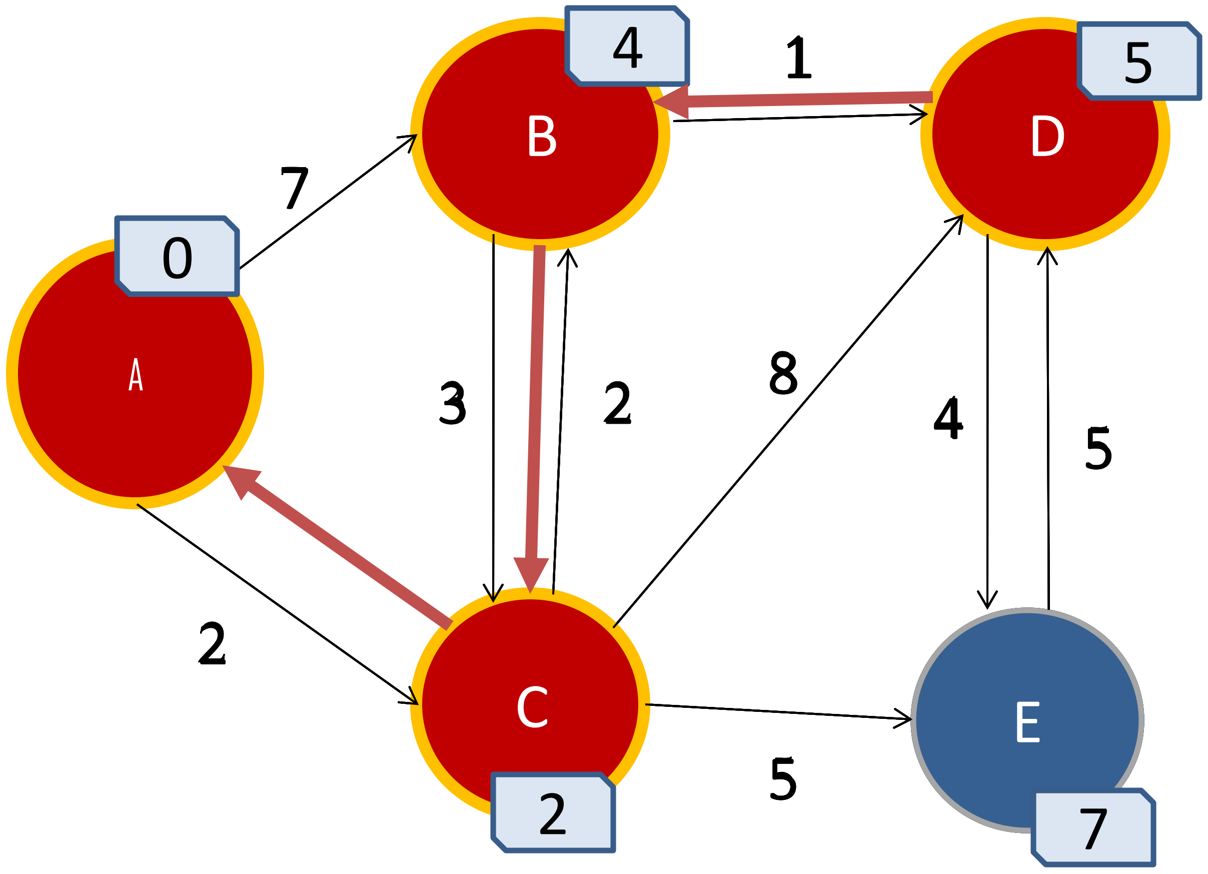
	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Shortest Path



	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Shortest Path



	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	7 A	2 A	∞	∞
C	0	4 C	2 A	10 C	7 C
B	0	4 C	2 A	5 B	7 C
D	0	4 C	2 A	5 B	7 C
E	0	4 C	2 A	5 B	7 C

Edge Relaxation

Looking for the minDistance is O(V)

```
          dist[s] ←0                            (distance to source v      ero)
O(V)      for  all v ∈ V–{s}
               do  dist[v] ←∞                   (set all other      ces to infinity)
          S←∅                                   (S, the set    isited vertices is initially empty)
          Q←V                                   (Q, th   queue initially contains all vertices)
  O(V)    while Q ≠∅                            (w   e the queue is not empty)
          do   u ← mindistance(Q,dist)          (select the element of Q with the min. distance)
               S←SU{u}                          (add u to list of visited vertices)
               for all v ∈ neighbors[u]
                    do  if   dist[v] > dist[u] + w(u, v)       (if new shortest path found)
                         then      d[v] ←d[u] + w(u, v)        (set new value of shortest path)
                                                (if desired, add traceback code)

          return dist
```

Edge Relaxation

```
dist[s] ← 0                                   (distance to source vertex is zero)
O(V)  for all v ∈ V–{s}
          do dist[v] ← ∞                      (set all other distances to infinity)
S ← ∅                                         (S, the set of visited vertices is initially empty)
Q ← V                                         (Q, the queue initially contains all vertices)
O(V)  while Q ≠ ∅                             (while the queue is not empty)
      do u ← mindistance(Q,dist)              (select the element of Q with the min. distance)
         S ← S∪{u}                            (add u to list of visited vertices)
         for all v ∈ neighbors[u]
             do if dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                   then d[v] ← d[u] + w(u, v)         (set new value of shortest path)
                                              (if desired, add traceback code)
      return dist
```

Continue looping until queue is empty

Edge Relaxation

Looking for the minDistance is O(V)

```
       dist[s] ←0                                  (distance to source v        ero)
O(V)   for  all v ∈ V–{s}
              do  dist[v] ←∞                       (set all other        ces to infinity)
       S←∅                                         (S, the set      isited vertices is initially empty)
       Q←V                                         (Q, the    ueue initially contains all vertices)
O(V)   while Q ≠∅                                  (w   le the queue is not empty)
       do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
            S←SU{u}                                (add u to list of visited vertices)
            for all v ∈ neighbors[u]
                do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                         then     d[v] ←d[u] + w(u, v)        (set new value of shortest path)
                                                    (if desired, add traceback code)
       return dist
```

Edge Relaxation

```
      dist[s] ← 0                                (distance to source vertex is zero)
O(V)  for all v ∈ V–{s}
          do dist[v] ← ∞                         (set all other distances to infinity)
      S ← Ø                                      (S, the set of visited vertices is initially empty)
      Q ← V                                      (Q, the queue initially contains all vertices)
O(V)  while Q ≠ Ø                  O(V)          (while the queue is not empty)
      do  u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
          S ← S∪{u}                              (add u to list of visited vertices)
          for all v ∈ neighbors[u]
O(E)          do if  dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                 then    d[v] ← d[u] + w(u, v)            (set new value of shortest path)
                                                 (if desired, add traceback code)
      return dist
```

Total time O(|V^2| + |E|) = O(|V^2|)

```
        dist[s] ← 0                                  (distance to source vertex is zero)
O(V)    for  all v ∈ V–{s}
              do  dist[v] ← ∞                        (set all other distances to infinity)
        S ← Ø                                        (S, the set of visited vertices is initially empty)
        Q ← V                                        (Q, the queue initially contains all vertices)
        while Q ≠ Ø                                  (while the queue is not empty)
        do   u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
             S ← S∪{u}                               (add u to list of visited vertices)
             for all v ∈ neighbors[u]
                  do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                          then     d[v] ← d[u] + w(u, v)        (set new value of shortest path)
                                                         (if desired, add traceback code)
        return dist
```

```
          dist[s] ← 0                                      (distance to source vertex is zero)
O(V)      for all v ∈ V–{s}
              do dist[v] ← ∞                               (set all other distances to infinity)
          S ← Ø                                            (S, the set of visited vertices is initially empty)
          Q ← V                                            (Q, the queue initially contains all vertices)
O(V)      while Q ≠ Ø                                      (while the queue is not empty)
          do  u ← mindistance(Q,dist)                      (select the element of Q with the min. distance)
              S ← S U {u}                                  (add u to list of visited vertices)
              for all v ∈ neighbors[u]
                  do if  dist[v] > dist[u] + w(u, v)           (if new shortest path found)
                         then  d[v] ← d[u] + w(u, v)           (set new value of shortest path)
                                                       (if desired, add traceback code)
          return dist
```

Looking for the minDistance is O(log(V)) using removeMin()

```
       dist[s] ← 0                                      (distance to source v        ero)
O(V)   for  all v ∈ V−{s}
             do  dist[v] ← ∞                            (set all other d        ces to infinity)
       S ← ∅                                            (S, the set      isited vertices is initially empty)
       Q ← V                                            (Q, the    ueue initially contains all vertices)
       while Q ≠ ∅                                      (wh  le the queue is not empty)
O(V)   do  u ← mindistance(Q,dist)                      (select the element of Q with the min. distance)
           S ← S∪{u}                                    (add u to list of visited vertices)
           for all v ∈ neighbors[u]
               do  if   dist[v] > dist[u] + w(u, v)            (if new shortest path found)
                   then      d[v] ← d[u] + w(u, v)             (set new value of shortest path)
                                                   (if desired, add traceback code)

       return dist
```

Edge Relaxation

```
        dist[s] $\leftarrow 0$                                (distance to source vertex is zero)
O(V)    for  all v $\in V-\{s\}$
              do  dist[v] $\leftarrow \infty$                 (set all other distances to infinity)
        S$\leftarrow\emptyset$                                (S, the set of visited vertices is initially empty)
        Q$\leftarrow$V                                        (Q, the queue initially contains all vertices)
O(V)    while Q $\neq\emptyset$                  O(logV)      (while the queue is not empty)
        do   u $\leftarrow$ mindistance(Q,dist)               (select the element of Q with the min. distance)
             S$\leftarrow$S$\cup$\{u\}                        (add u to list of visited vertices)
             for all v $\in$ neighbors[u]
O(E)               do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                          then d[v] $\leftarrow$d[u] + w(u, v)   (set new value of shortest path)
                                                  (if desired, add traceback code)
        return dist
```

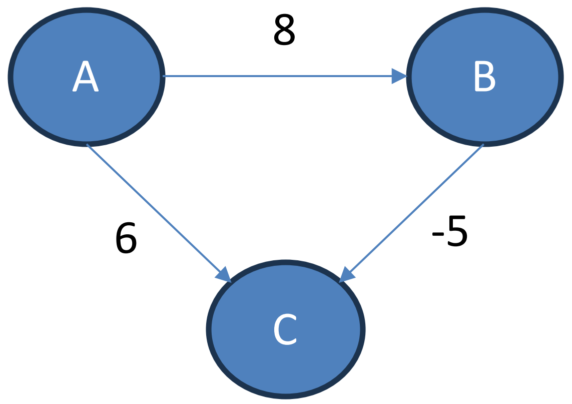
Edge Relaxation

```
      dist[s] ← 0                                  (distance to source vertex is zero)
O(V)  for all v ∈ V–{s}
          do dist[v] ← ∞                           (set all other distances to infinity)
      S ← ∅                                        (S, the set of visited vertices is initially empty)
      Q ← V                                        (Q, the queue initially contains all vertices)
O(V)  while Q ≠ ∅                    O(logV)       (while the queue is not empty)
      do   u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
           S ← S∪{u}                               (add u to list of visited vertices)
           for all v ∈ neighbors[u]
O(E)           do if  dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                      then d[v] ← d[u] + w(u, v)         (set new value of shortest path)
                                  O(logV)          (if desired, add traceback code)

      return dist
```

Total time O(VlogV + ElogV) = O((V+E) logV)

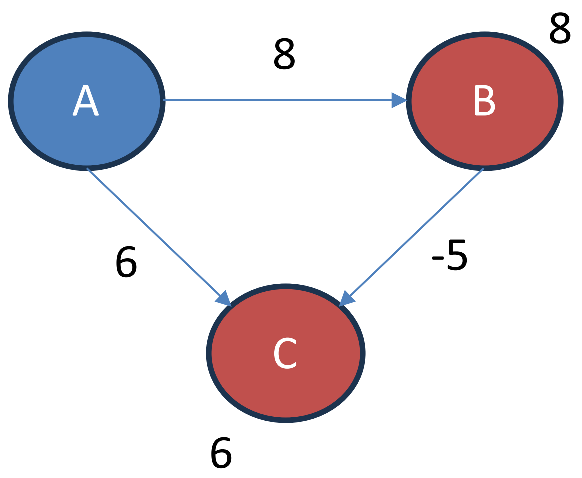
Bellman-Ford

Why can’t Dijkstra handle negative weights?

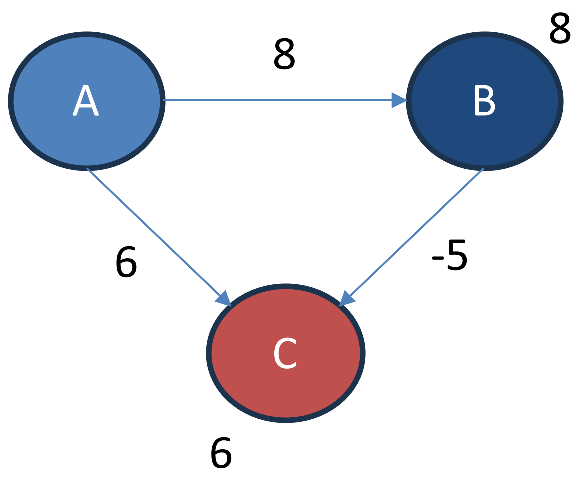


Mark A as visited

Why can’t Dijkstra handle negative weights?

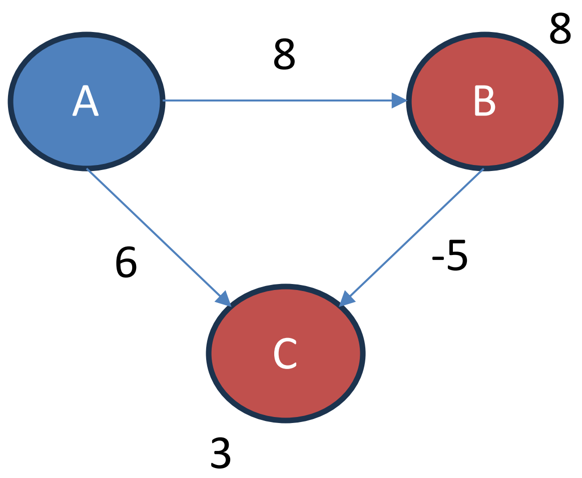


Why can’t Dijkstra handle negative weights?



Mark C as Visited

Why can’t Dijkstra handle negative weights?



Mark B as Visited

8+(-5)= 3

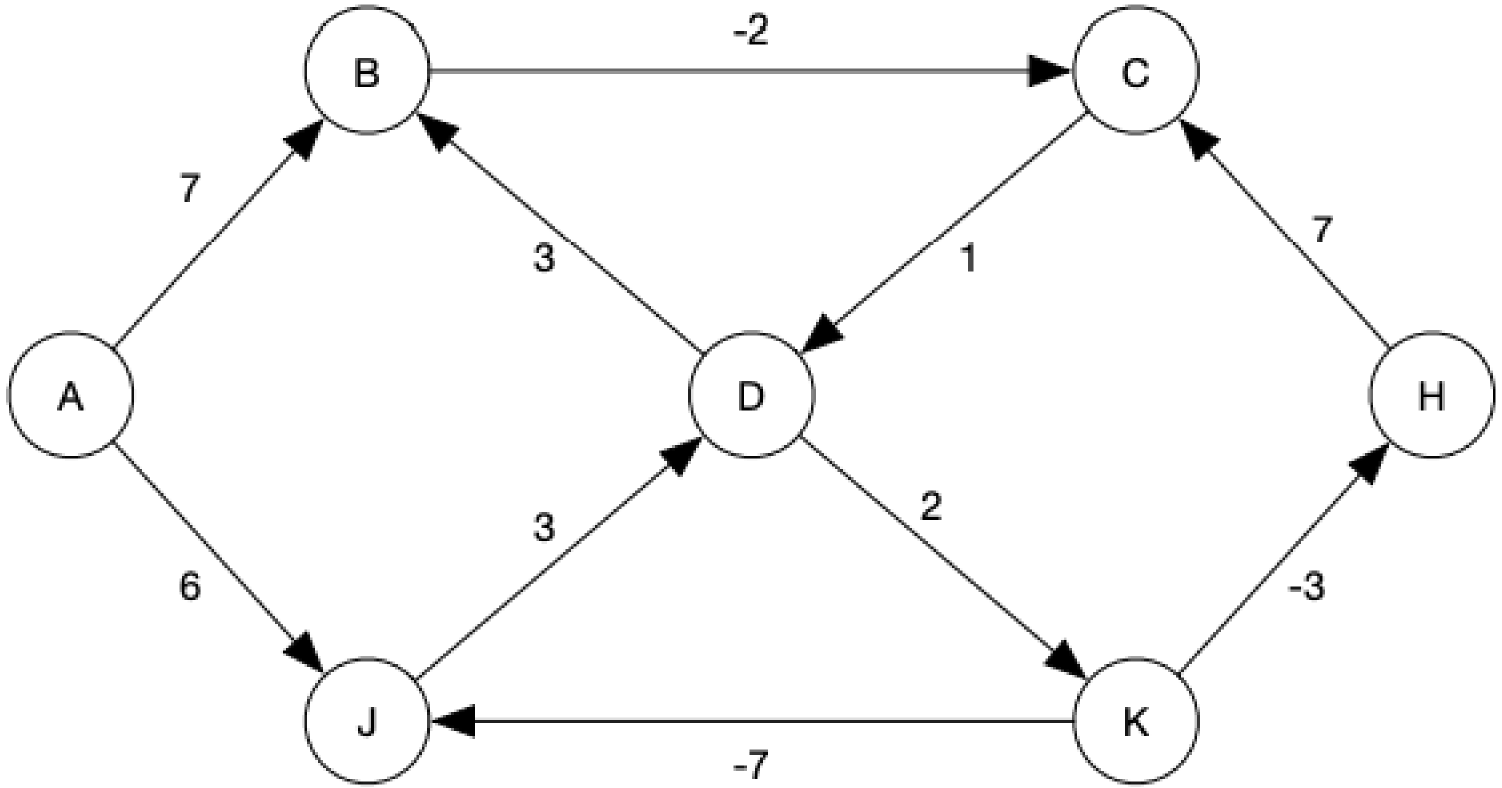
Should be a shorter but will not be reconsidered anymore

Bellman-Ford

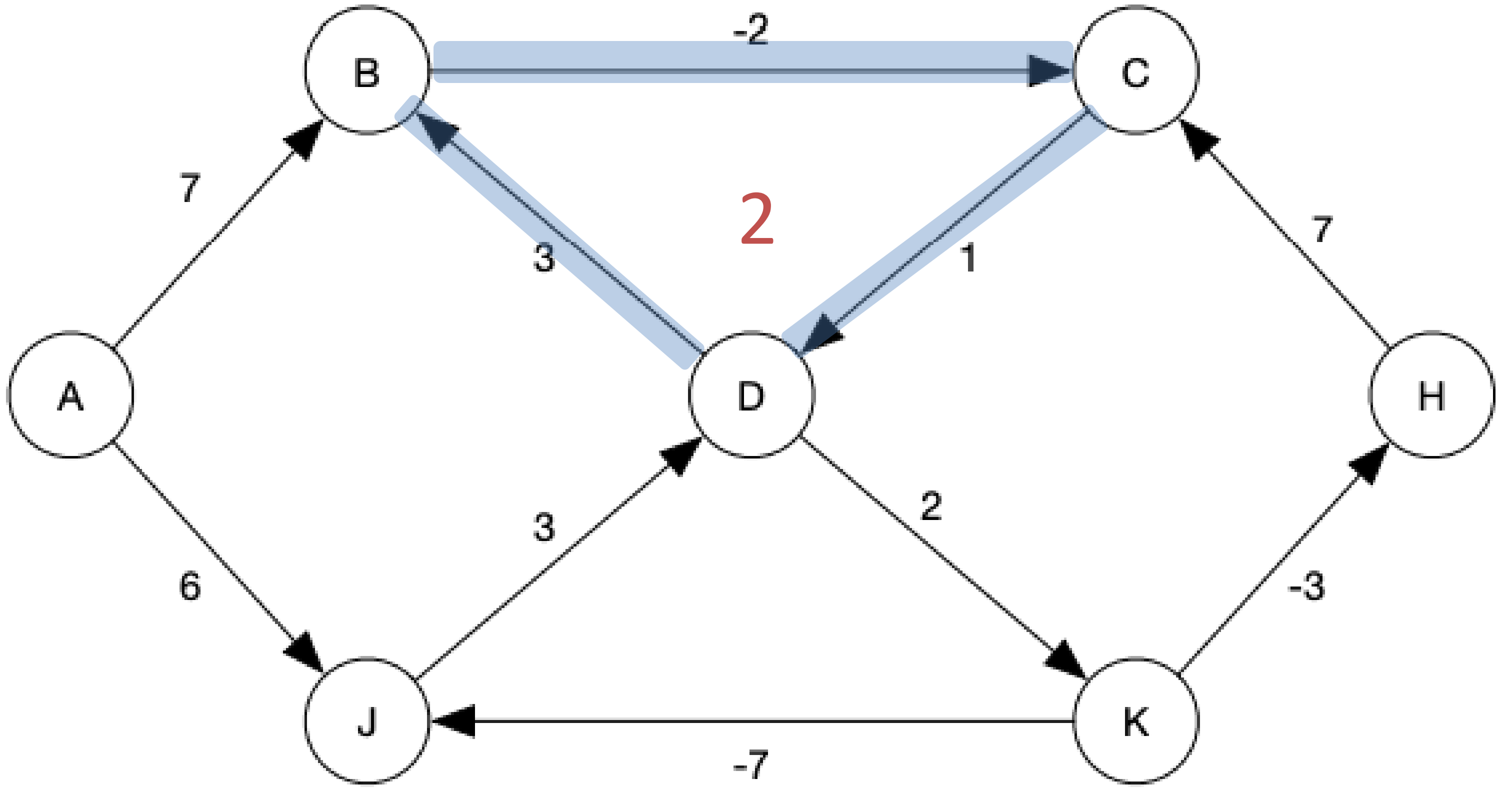
- Given some graph, G = (V, E), and some starting node S ∈ V, the BellmanFord algorithm will find the shortest paths (or paths with minimum weight) from S to all other nodes in V. Note that G must not contain any negative weight cycles.

What is a negative cycle?

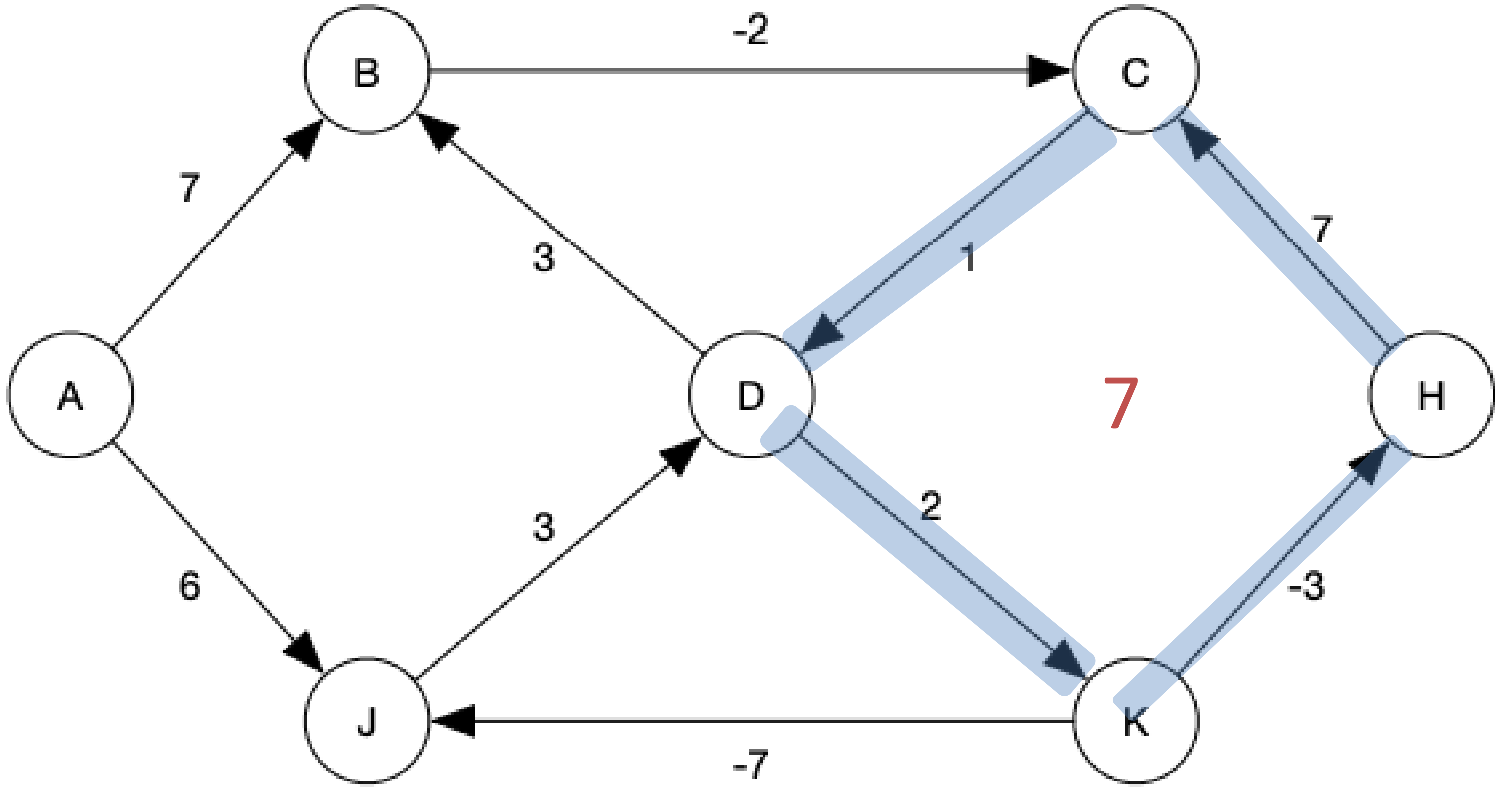
What is a negative cycle?



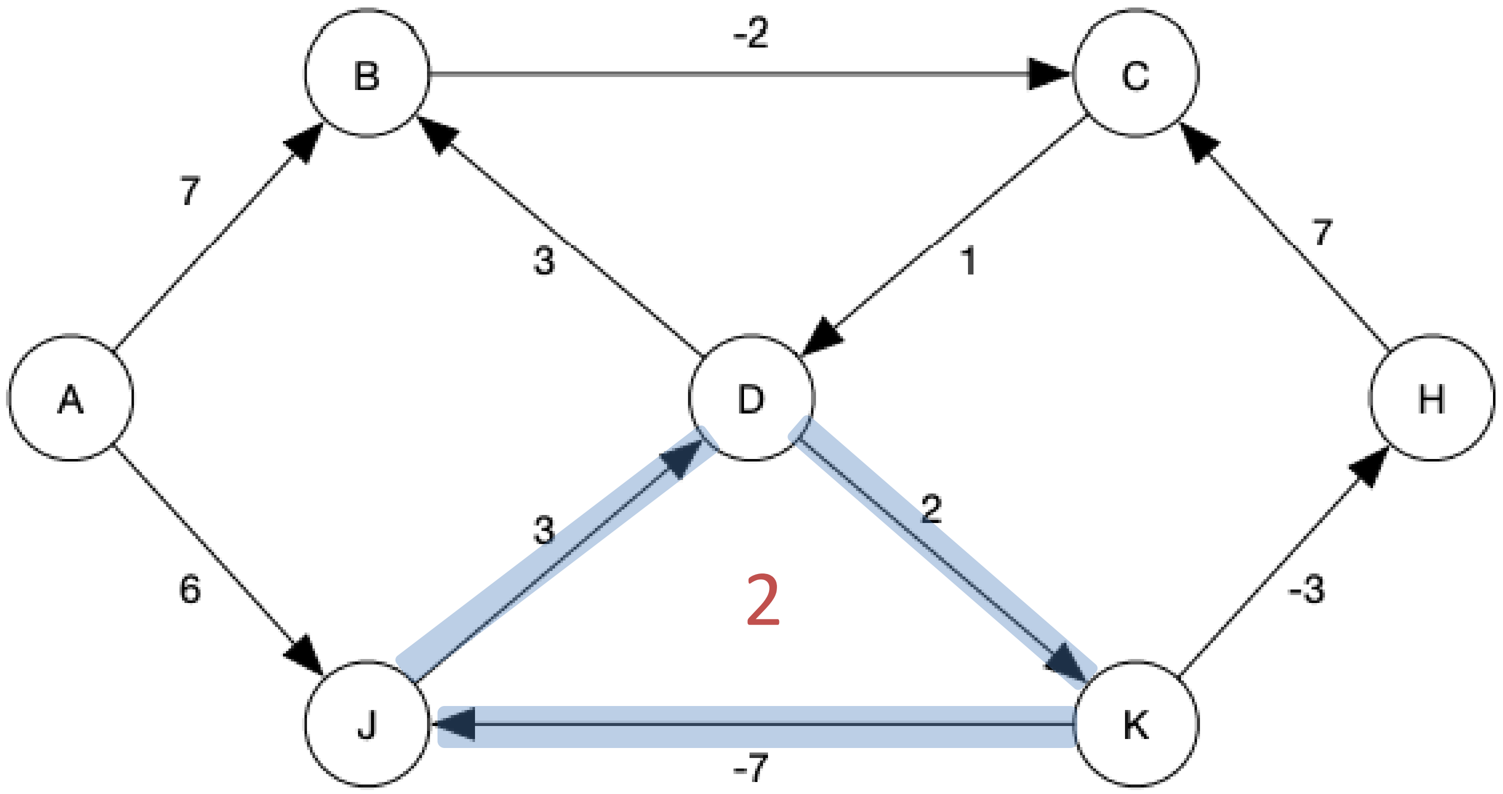
What is a negative cycle?



What is a negative cycle?



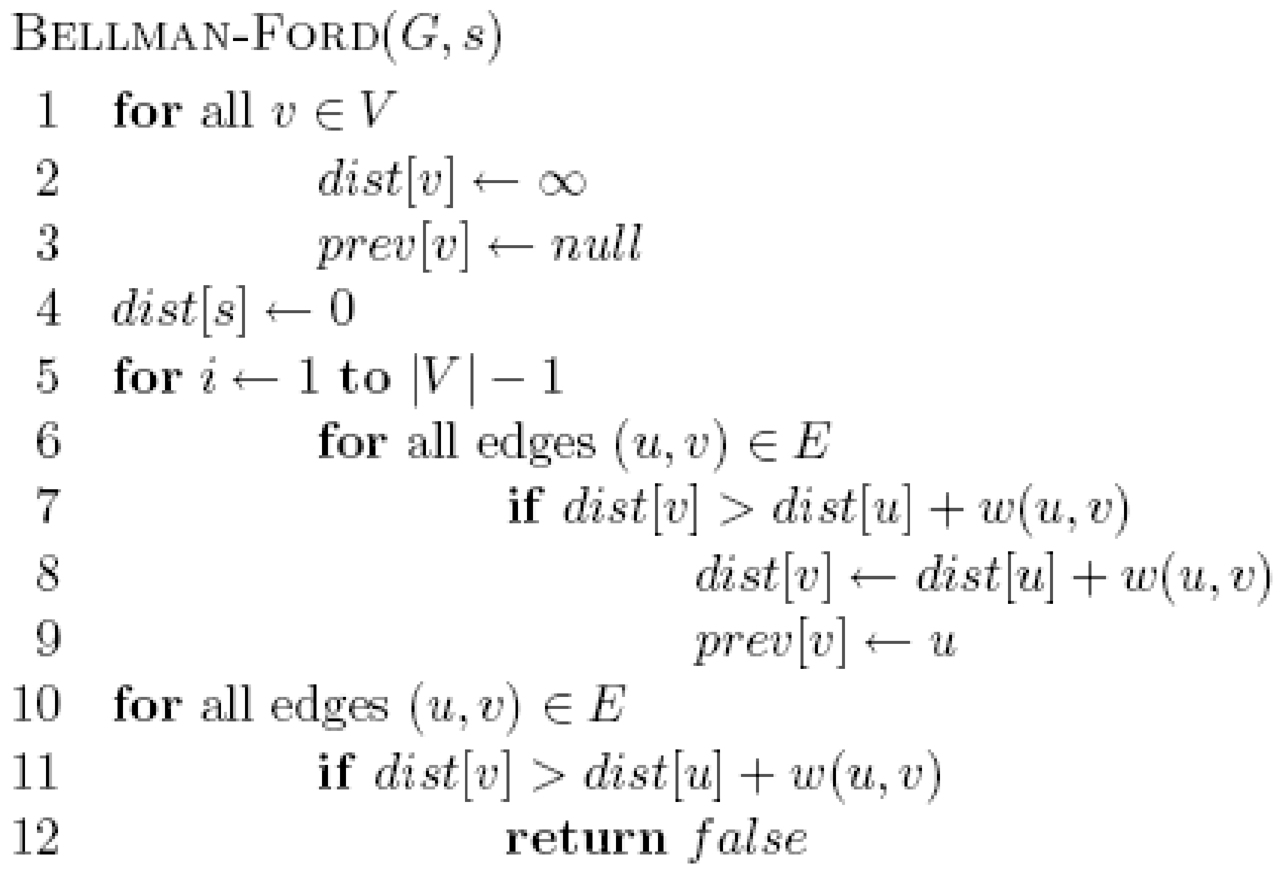
What is a negative cycle?



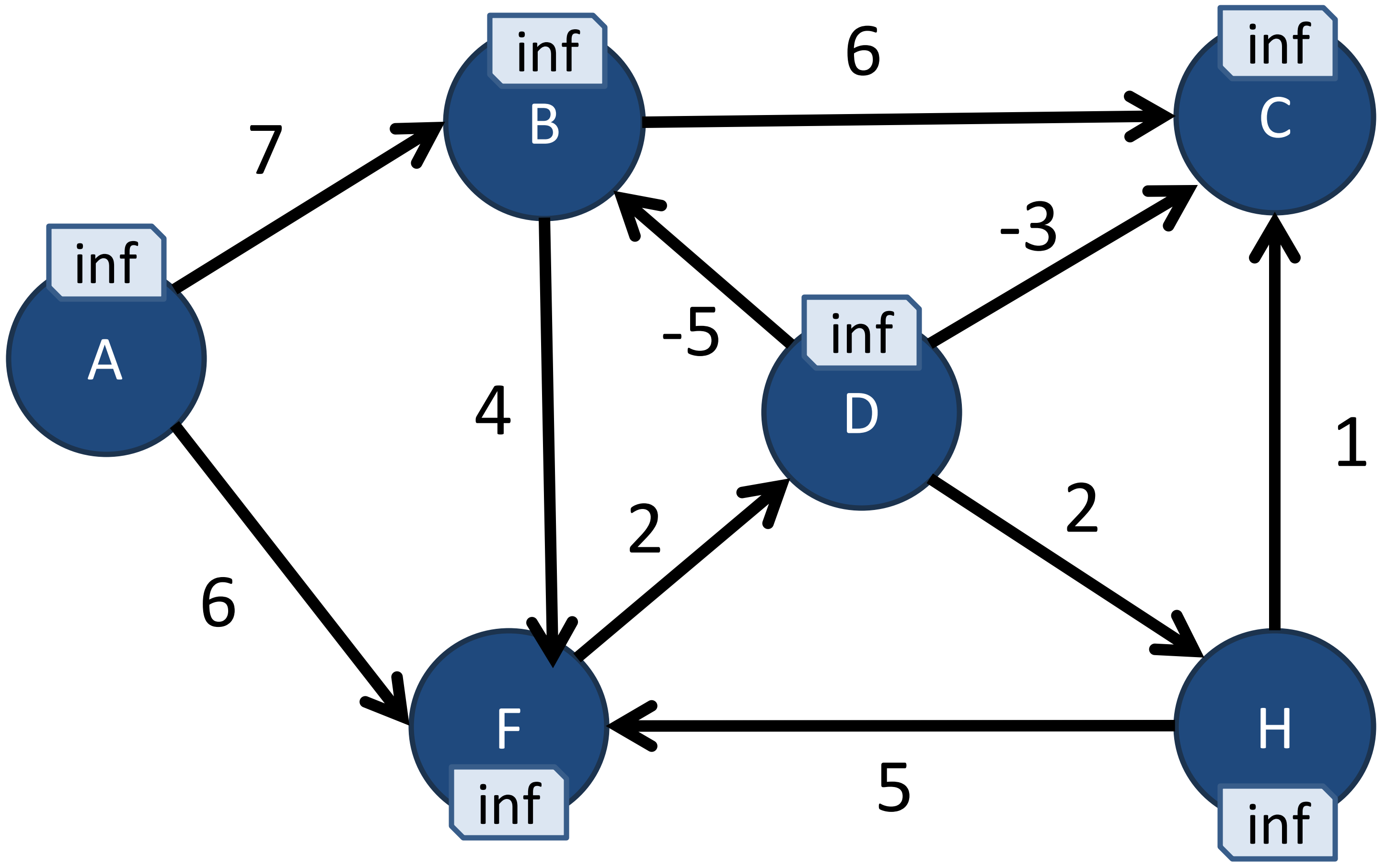
No Well-Defined Shortest Path

- Because you can keep lowering the total cost indefinitely, there is **no "shortest" path** to nodes reachable through the negative cycle — the cost can be made arbitrarily negative.

Bellman-Ford algorithm

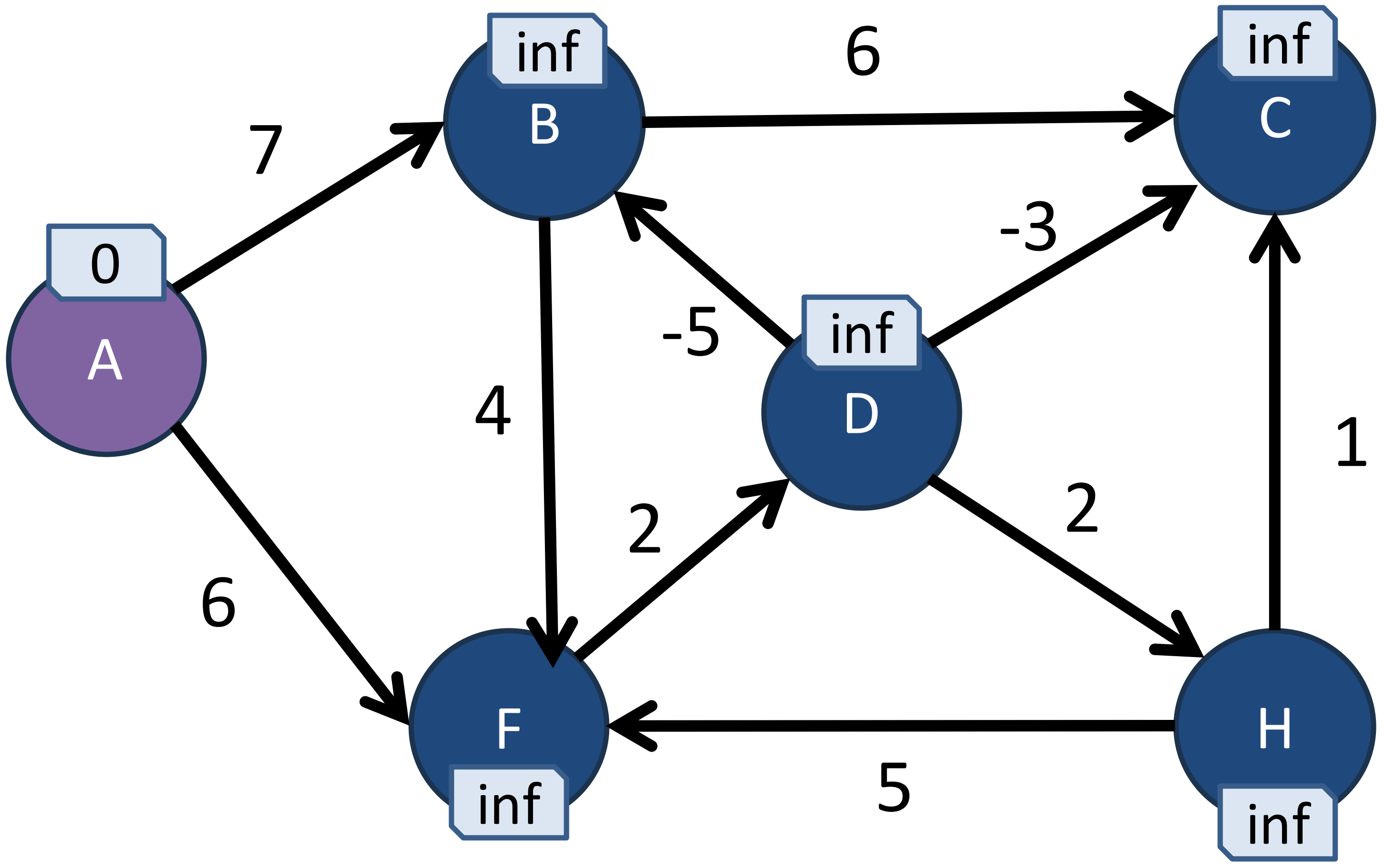


```
BELLMAN-FORD($G, s$)
for all $v \in V$
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ to $|V| - 1$
        for all edges $(u, v) \in E$
                if $dist[v] > dist[u] + w(u, v)$
                        $dist[v] \leftarrow dist[u] + w(u, v)$
                        $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
                return $false$
```



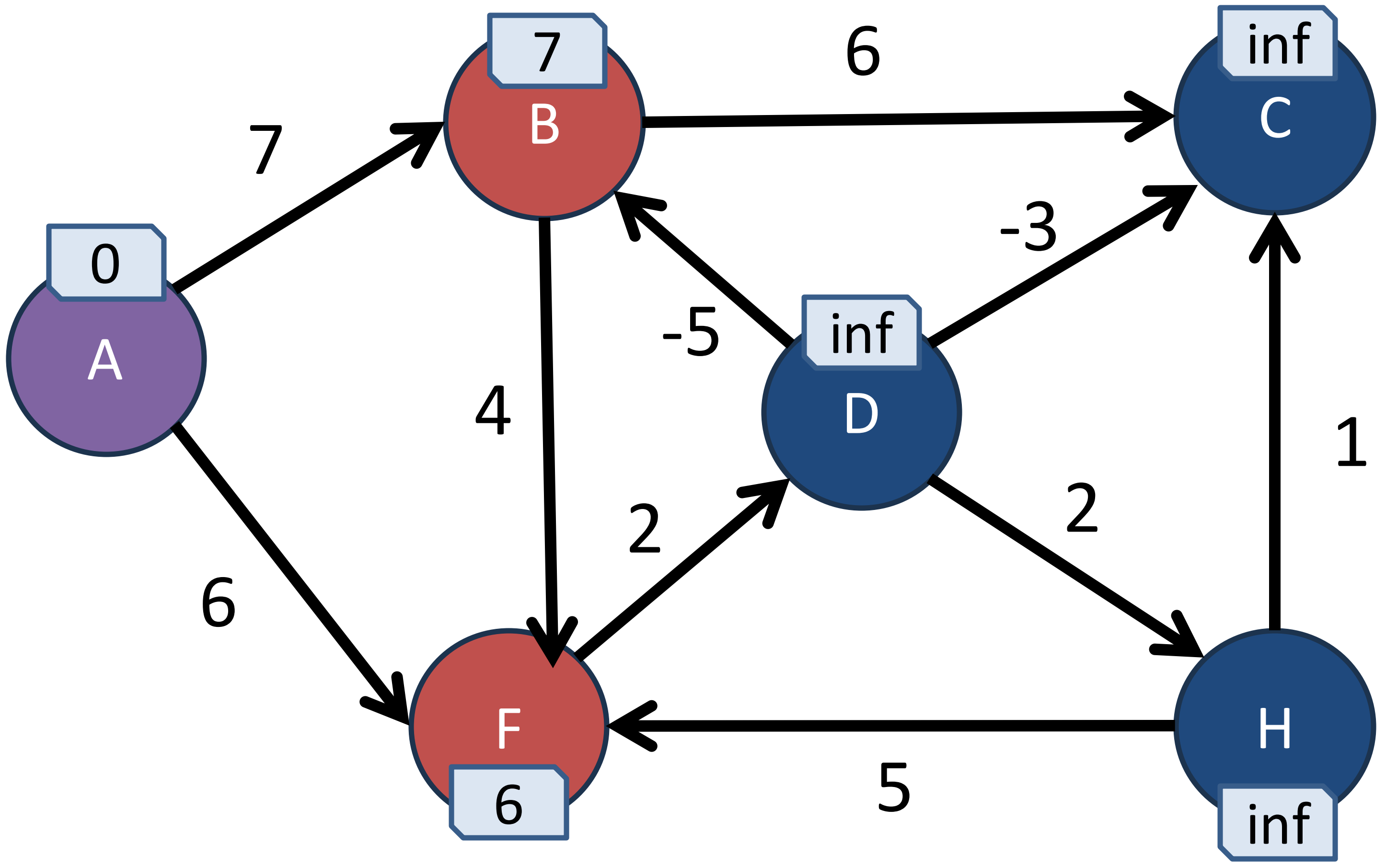
	Cost	Previous
A		
B		
C		
D		
F		
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



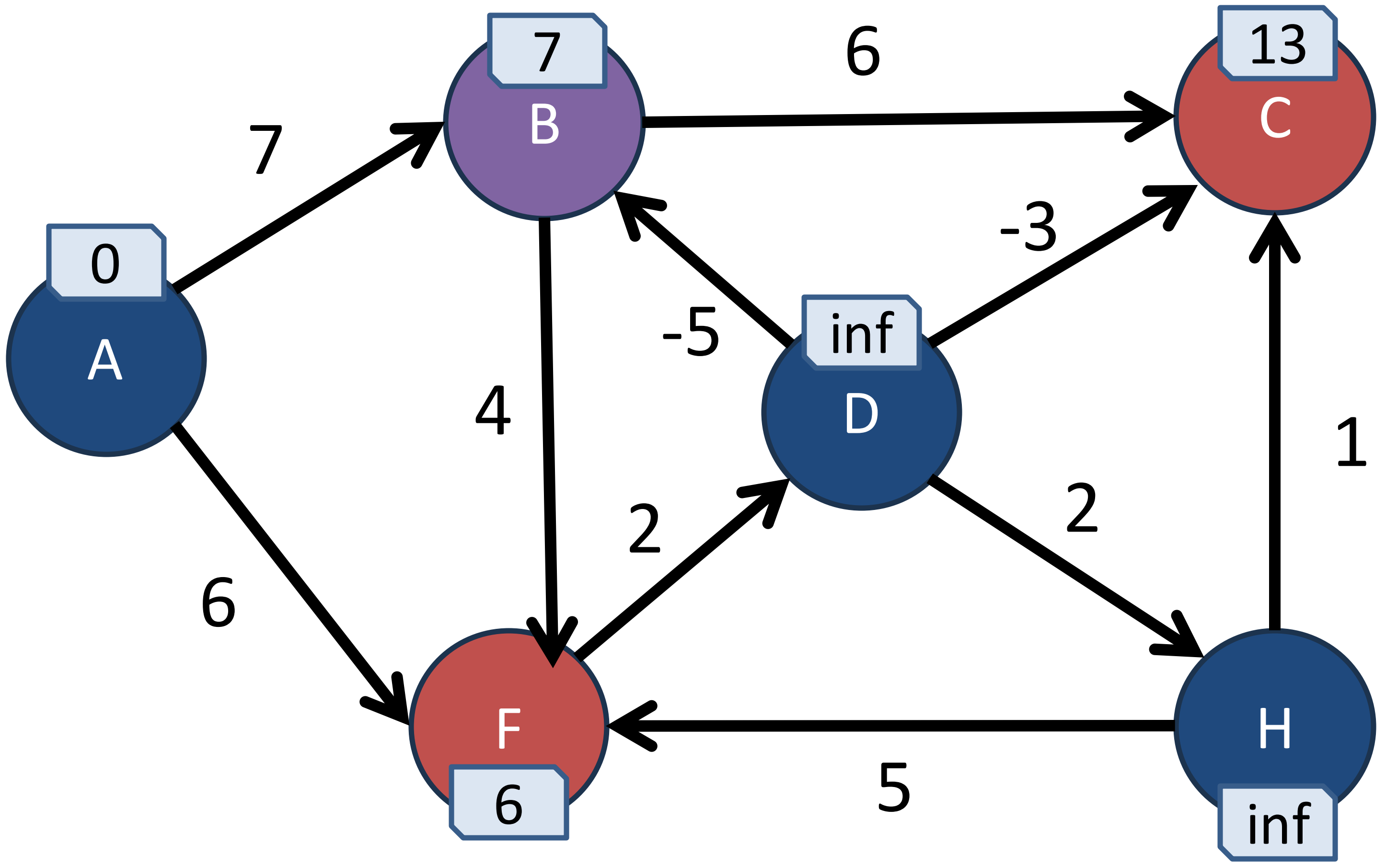
	Cost	Previous
A	**0**	
B		
C		
D		
F		
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



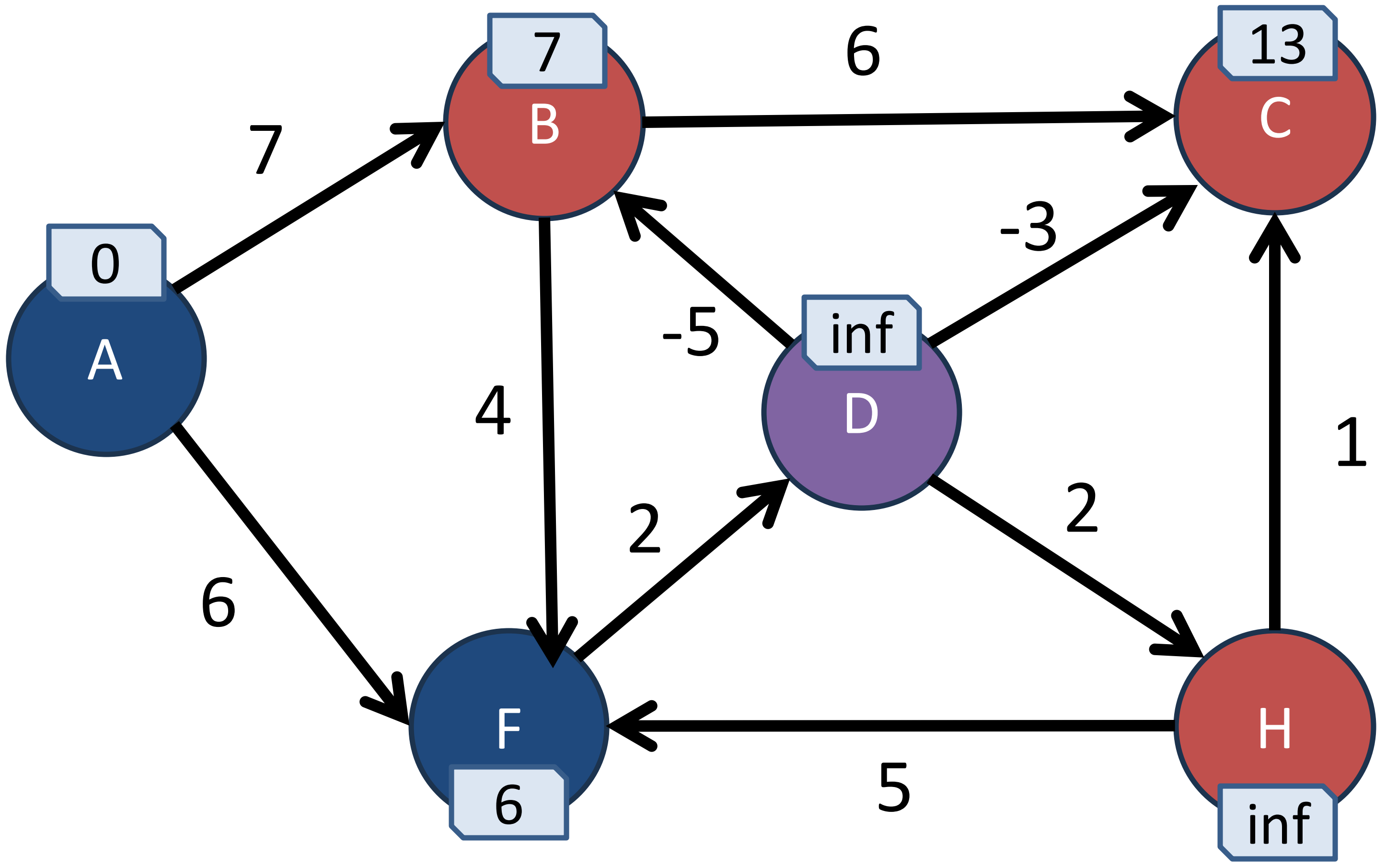
	Cost	Previous
A	0	
B	7	A
C		
D		
F	6	A
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



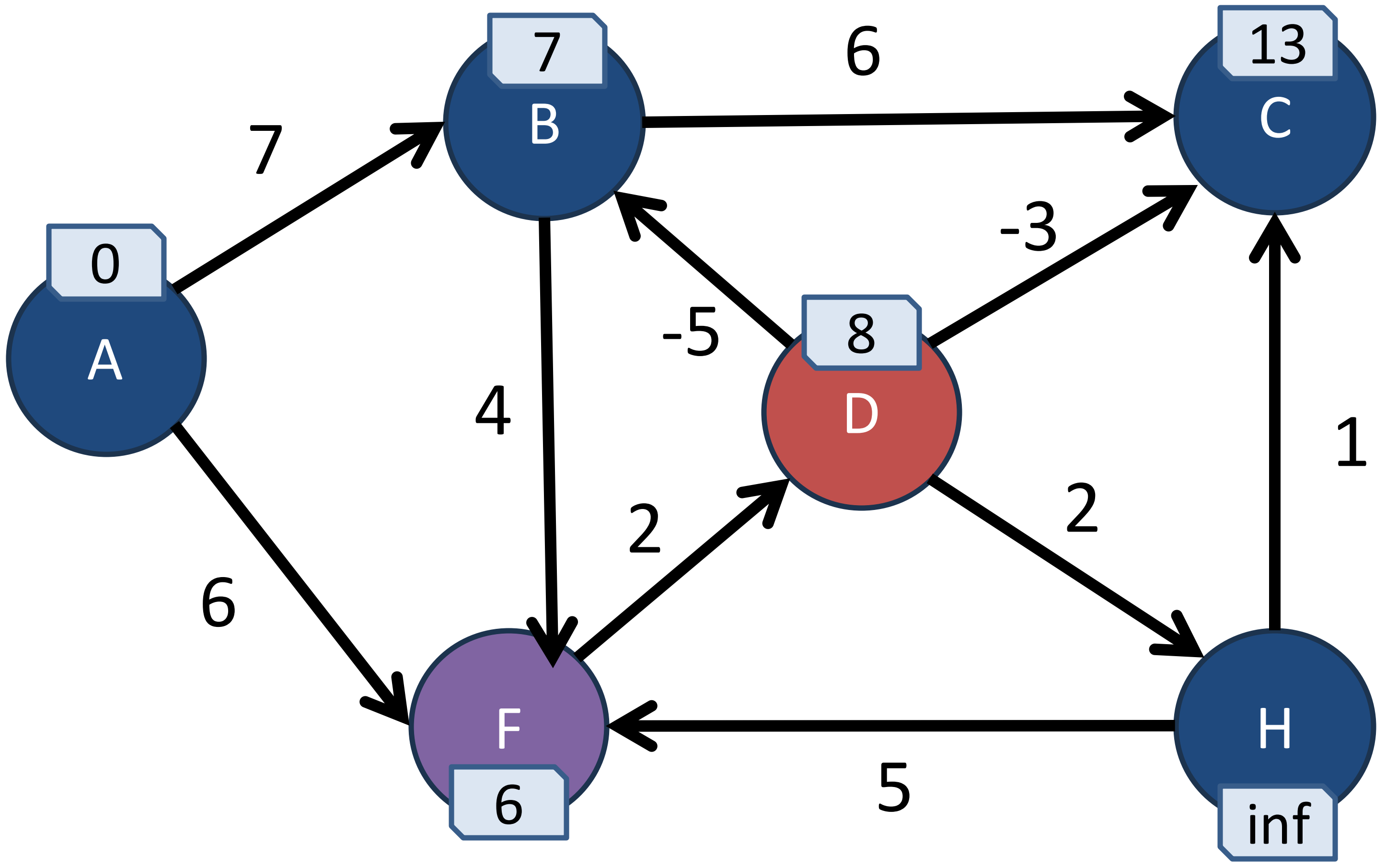
	Cost	Previous
A	0	
B	7	A
C	13	B
D		
F	6	A
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



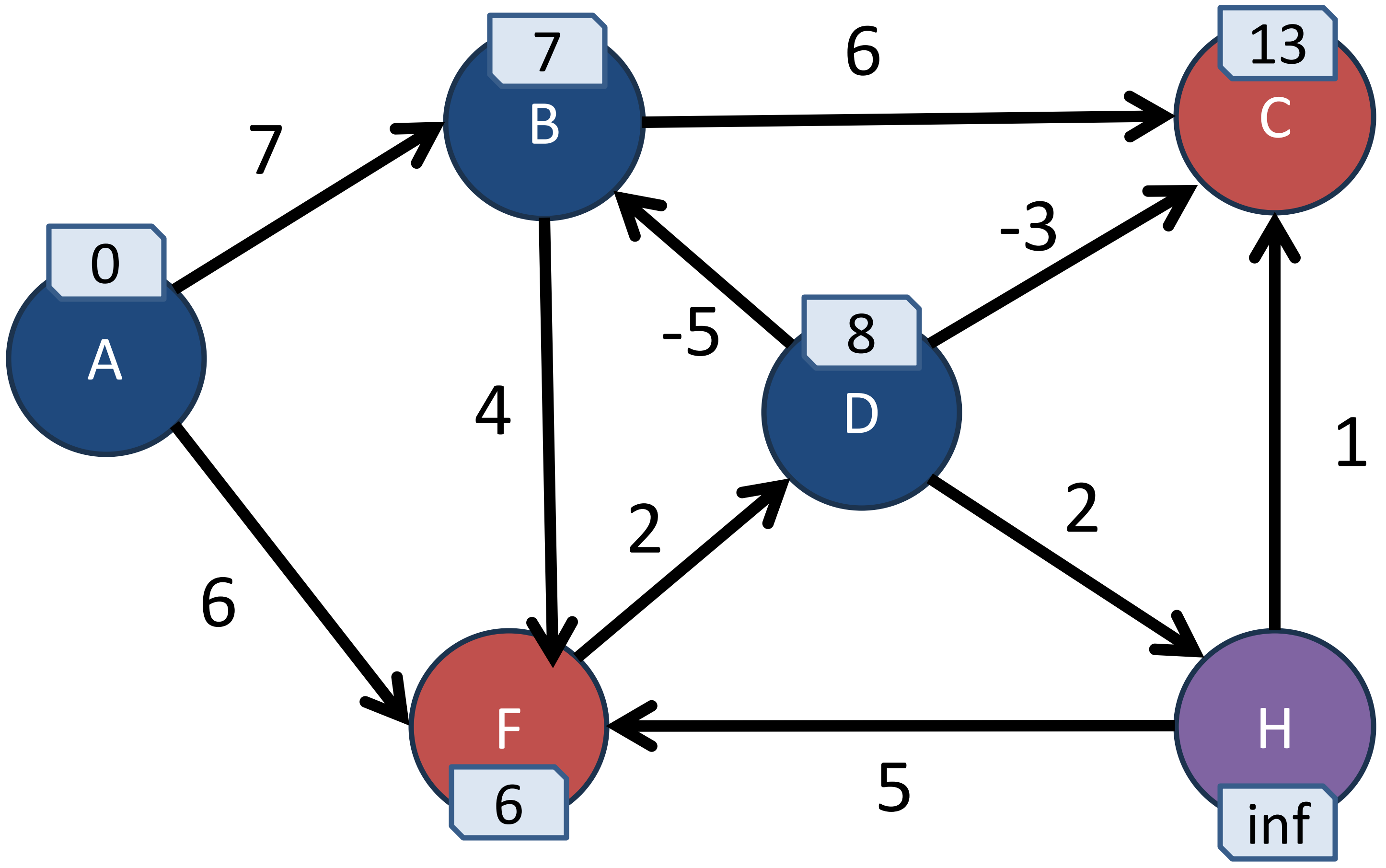
	Cost	Previous
A	0	
B	7	A
C	13	B
D		
F	6	A
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



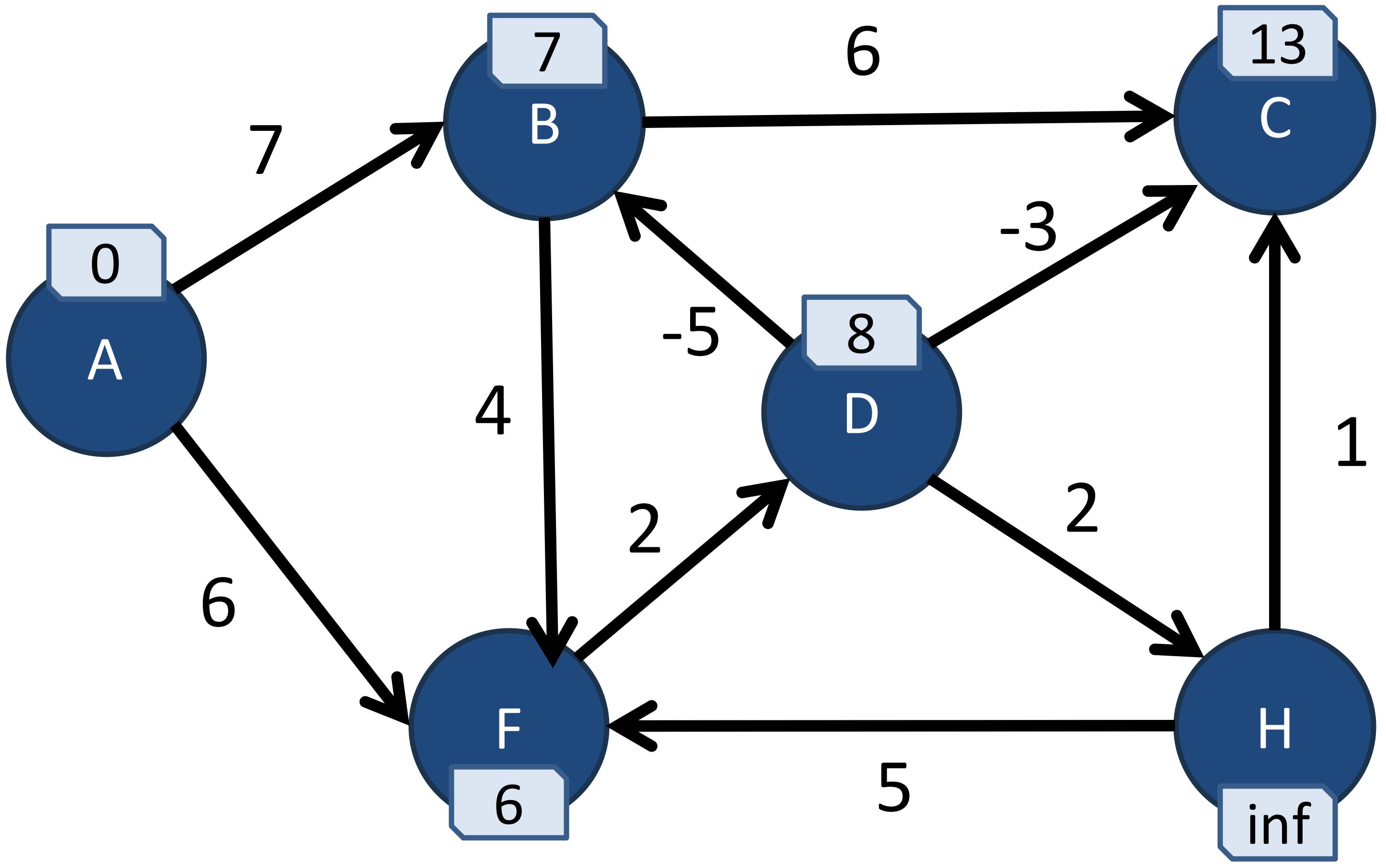
	Cost	Previous
A	0	
B	7	A
C	13	B
D	8	F
F	6	A
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	7	A
C	13	B
D	8	F
F	6	A
H		

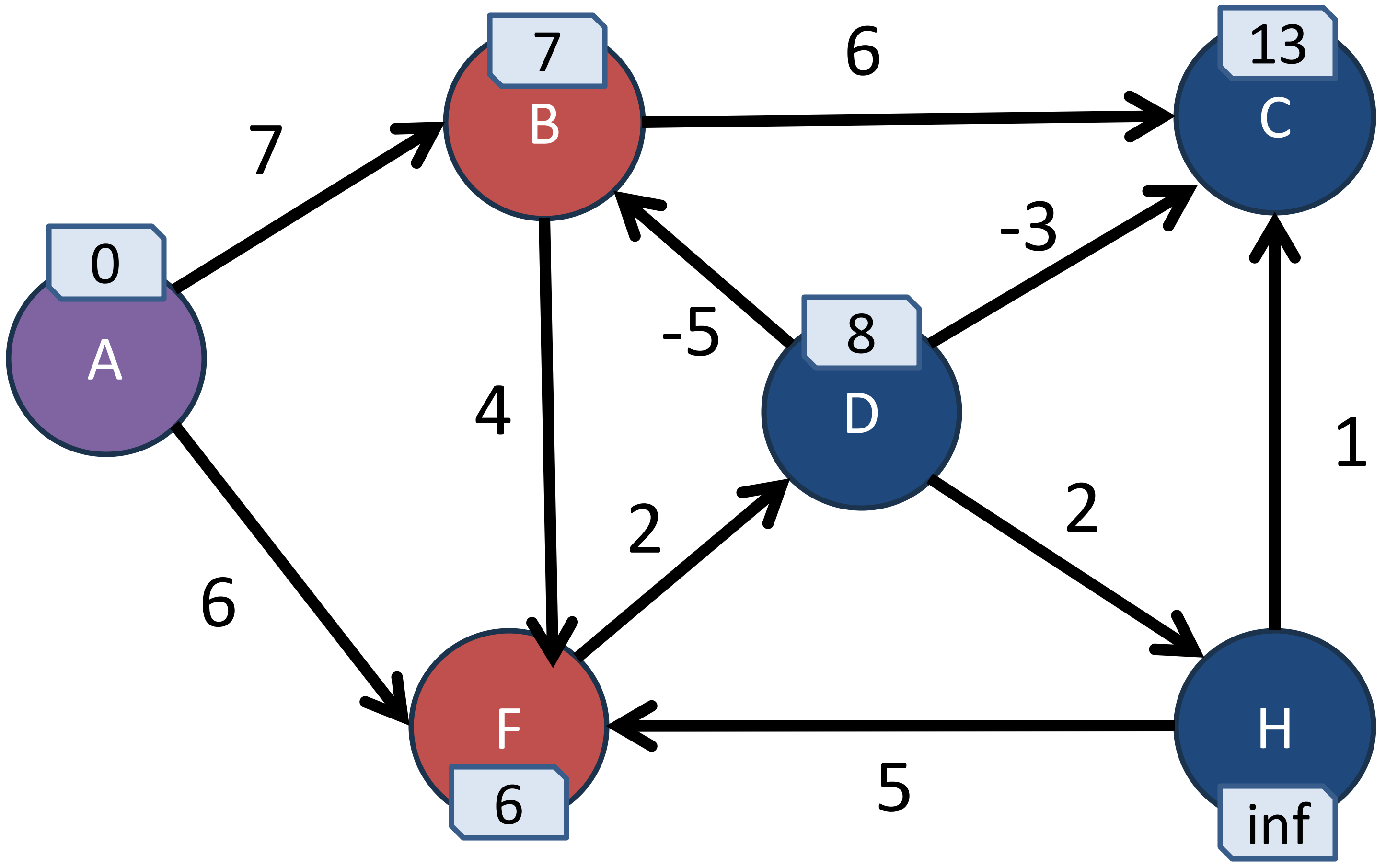
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	7	A
C	13	B
D	8	F
F	6	A
H		

Second Iteration

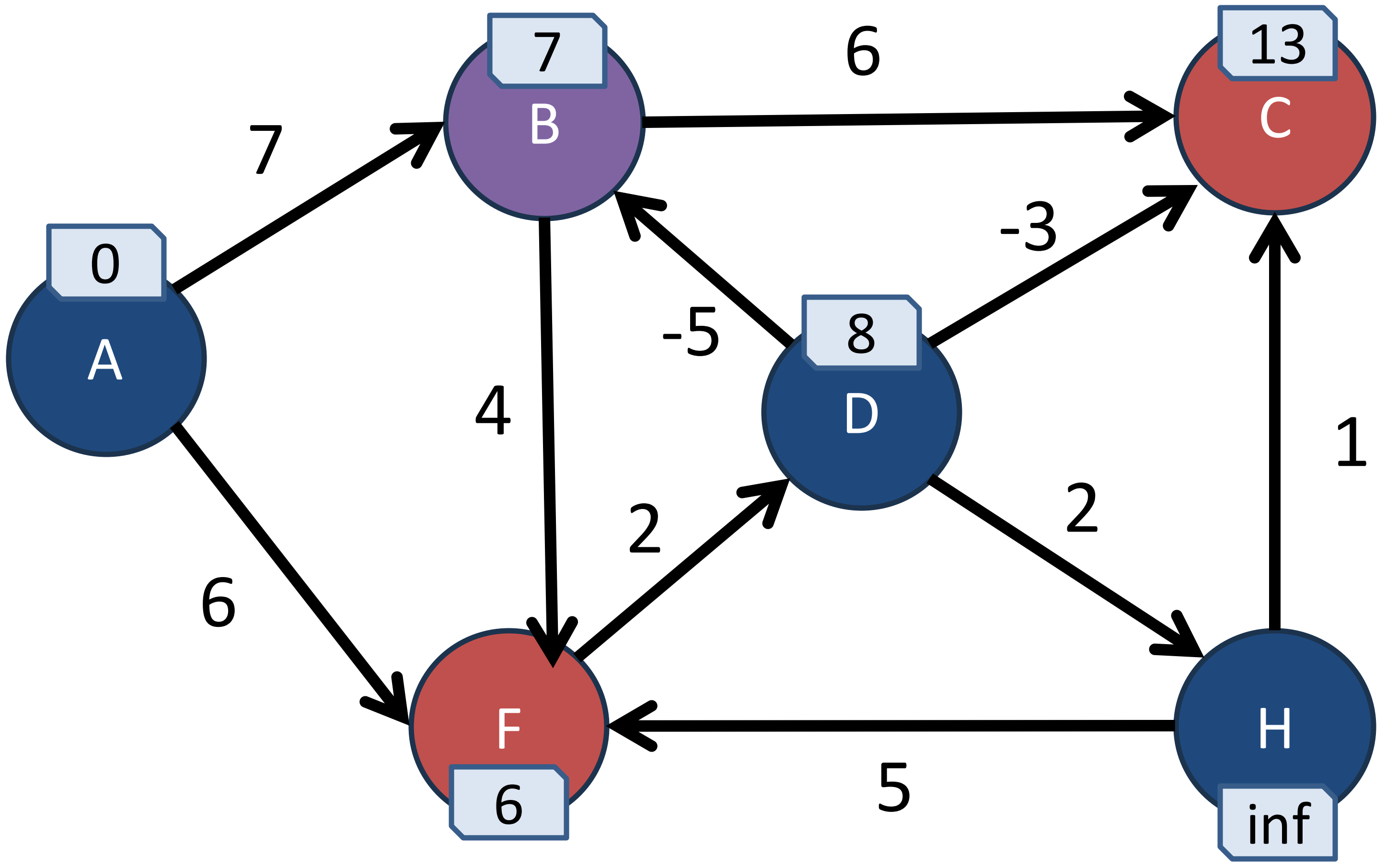
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	7	A
C	13	B
D	8	F
F	6	A
H		

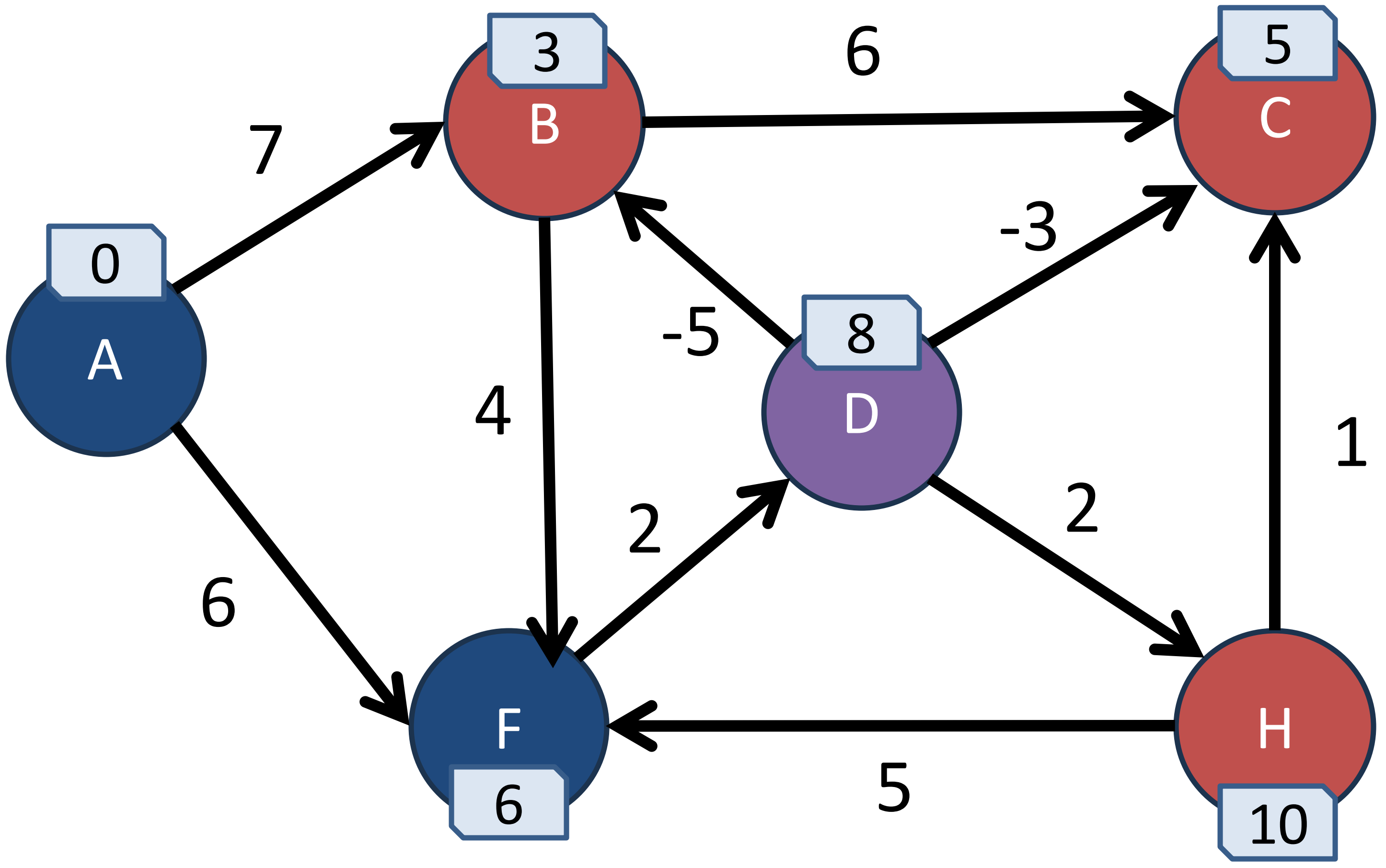
Second Iteration

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



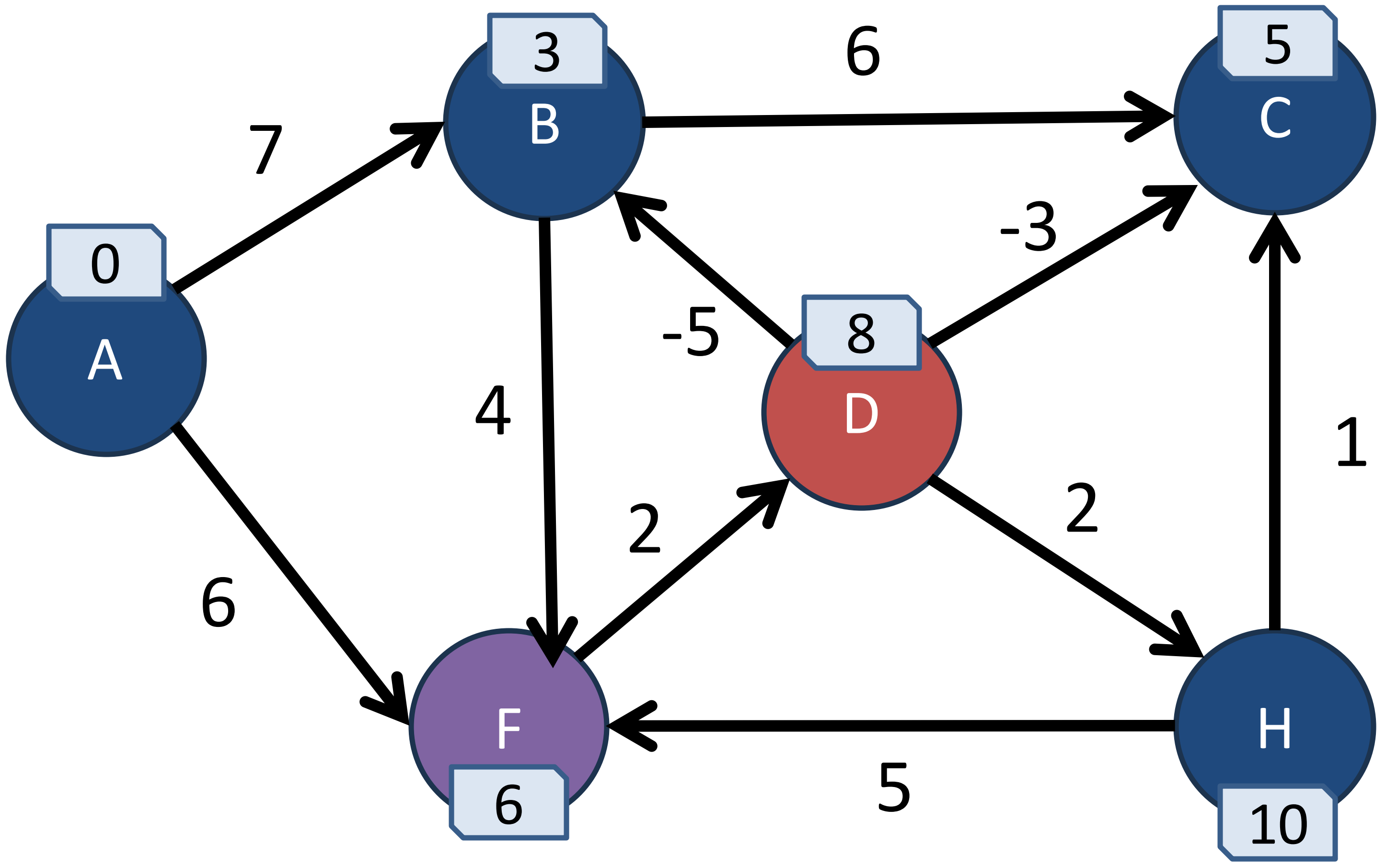
	Cost	Previous
A	0	
B	7	A
C	13	B
D	8	F
F	6	A
H		

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



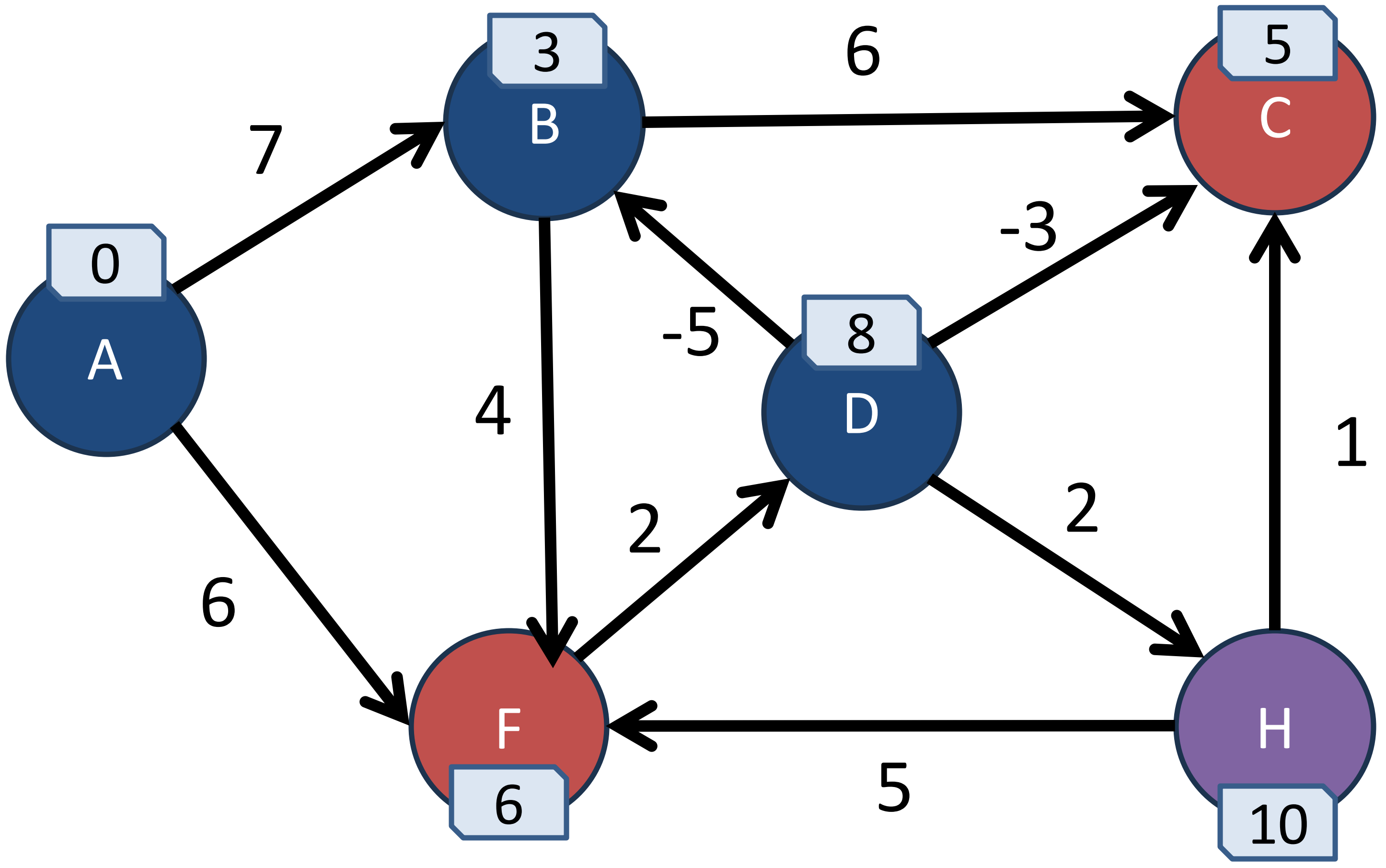
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



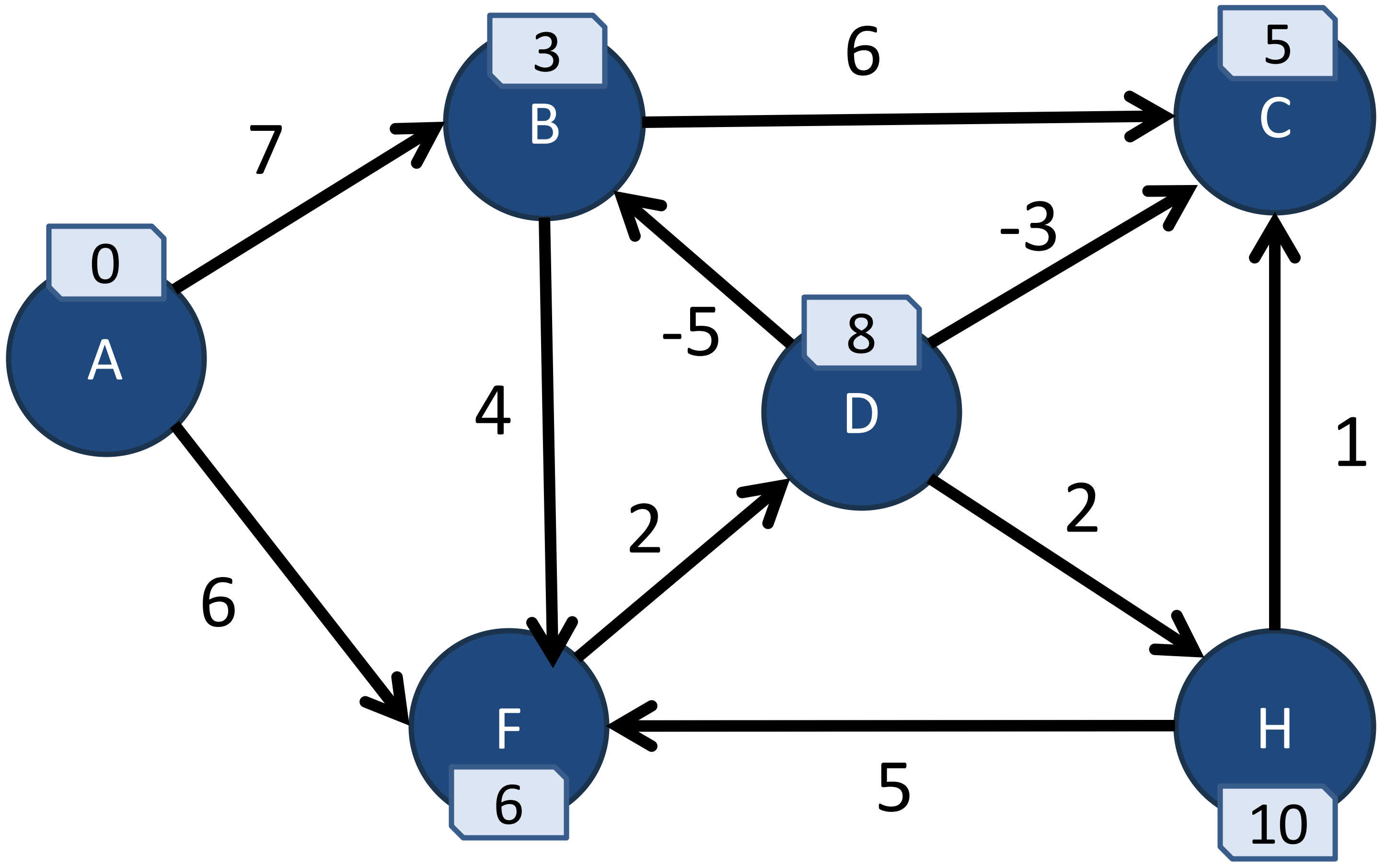
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

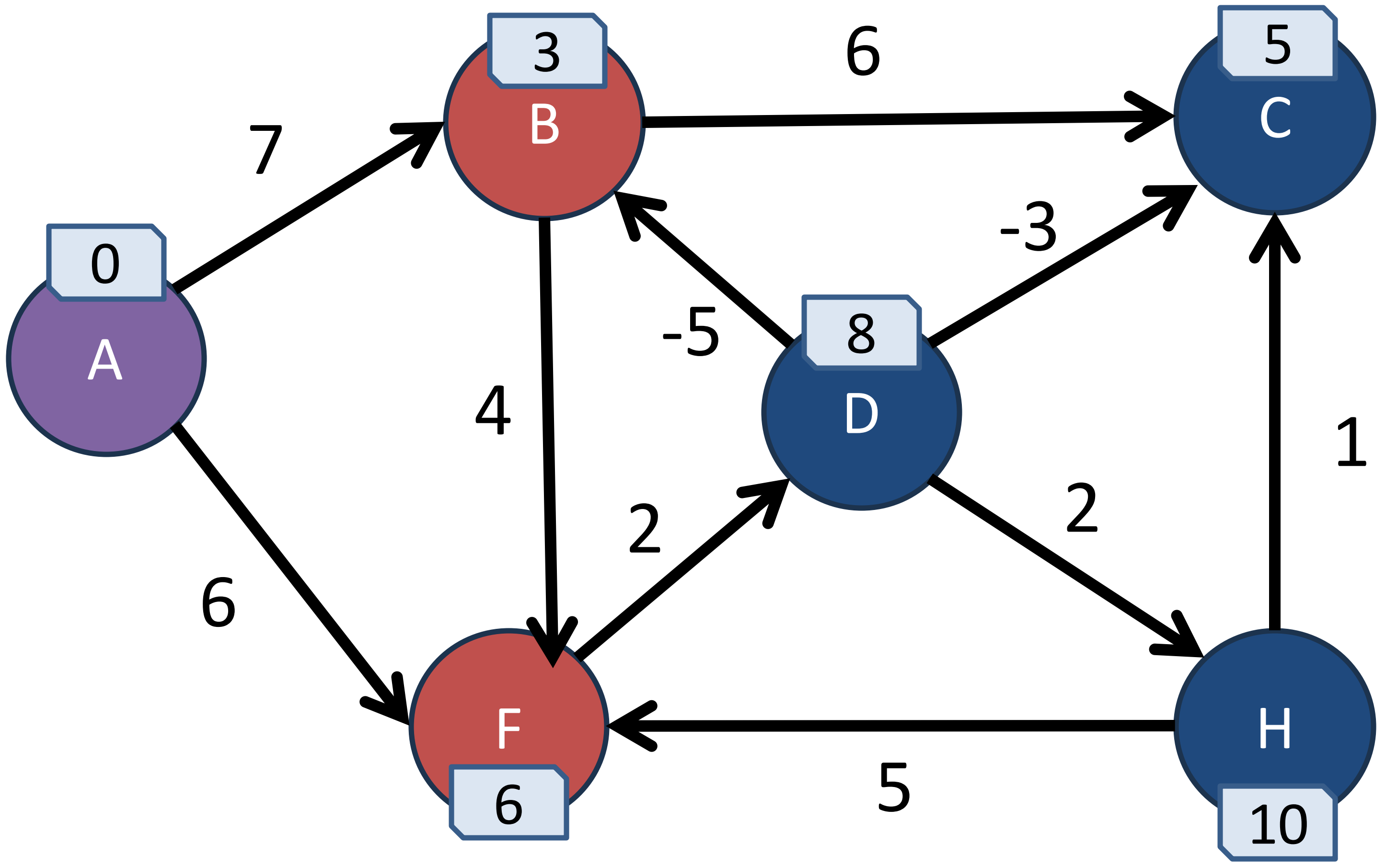
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

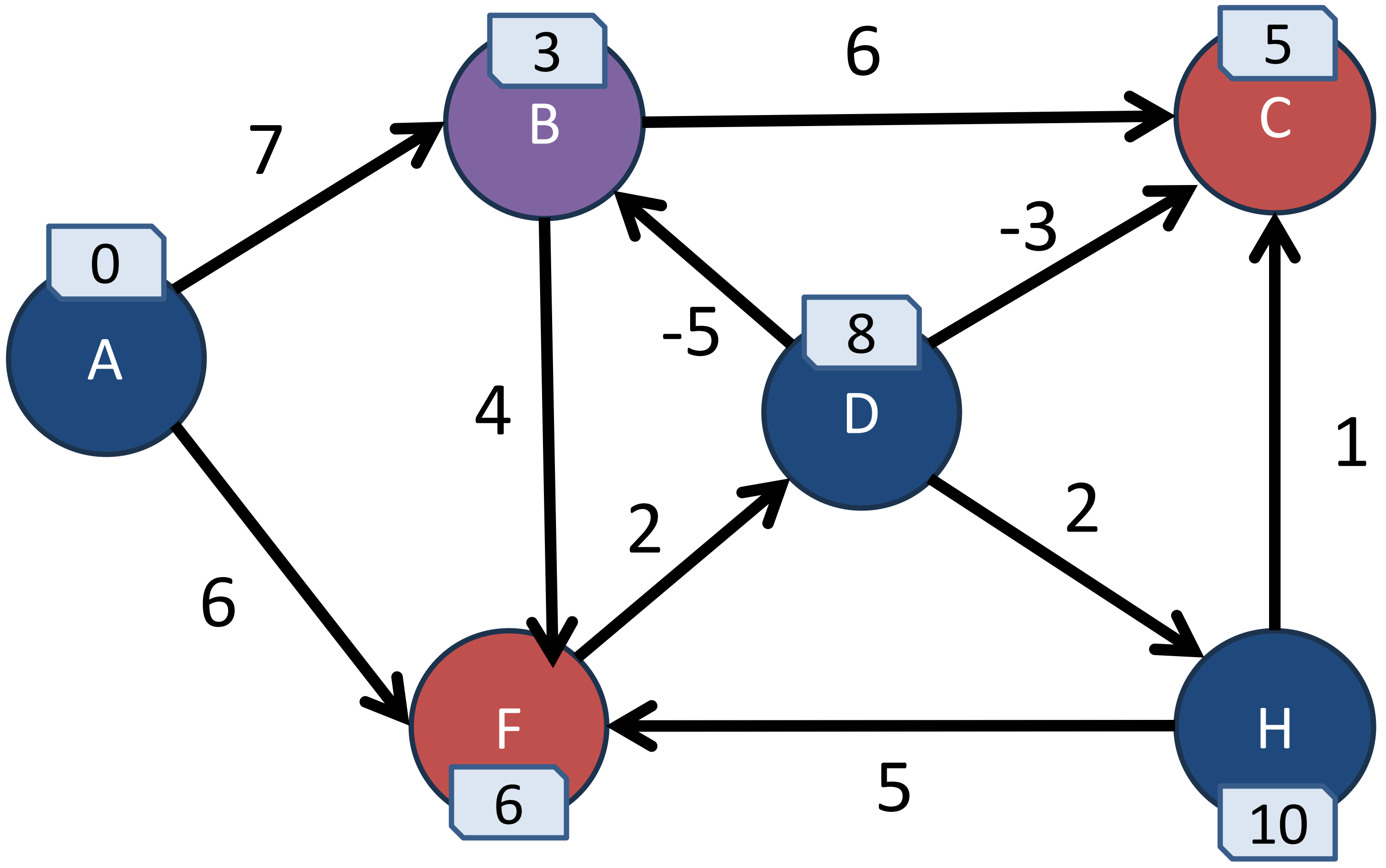
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC

Third Iteration



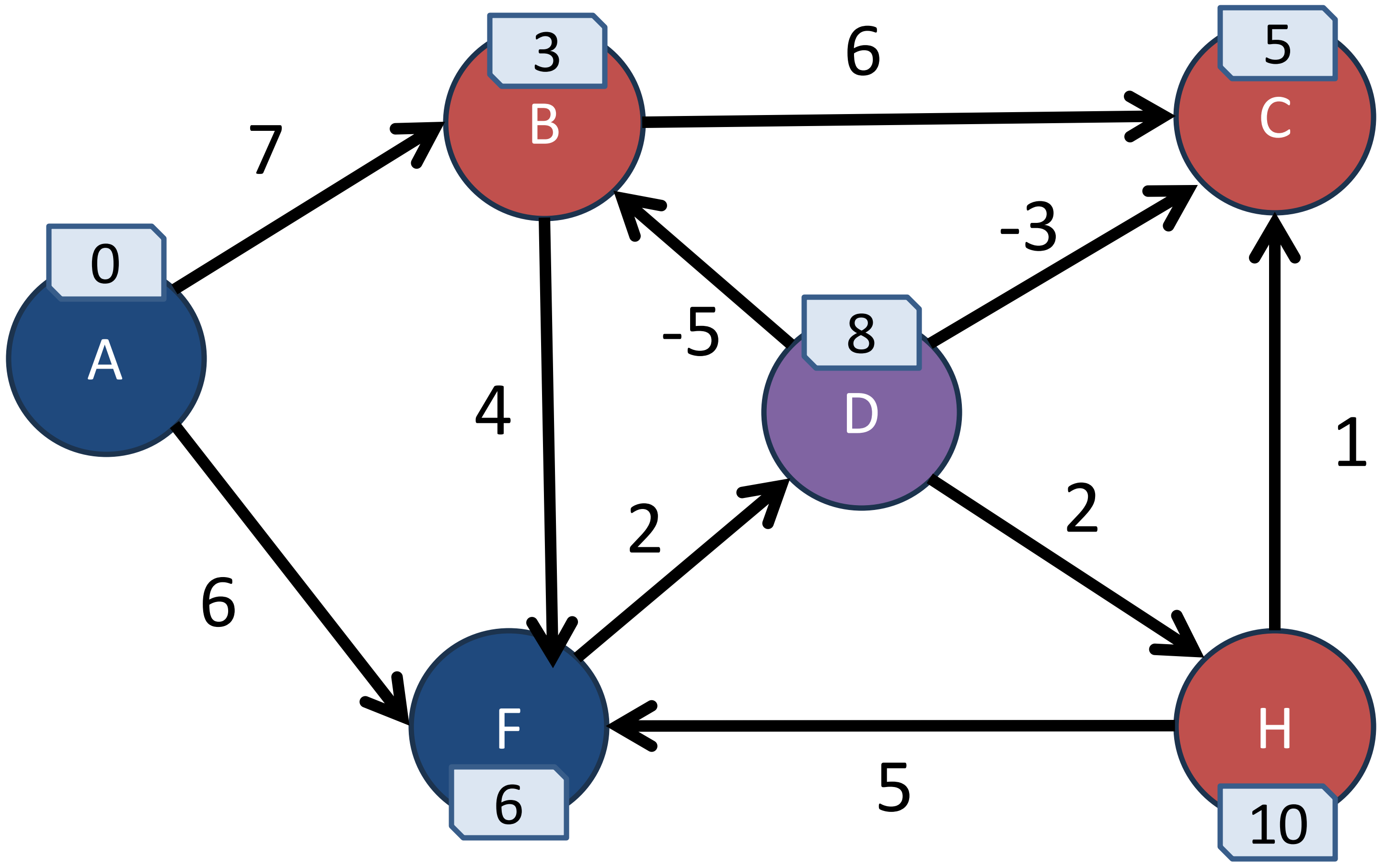
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



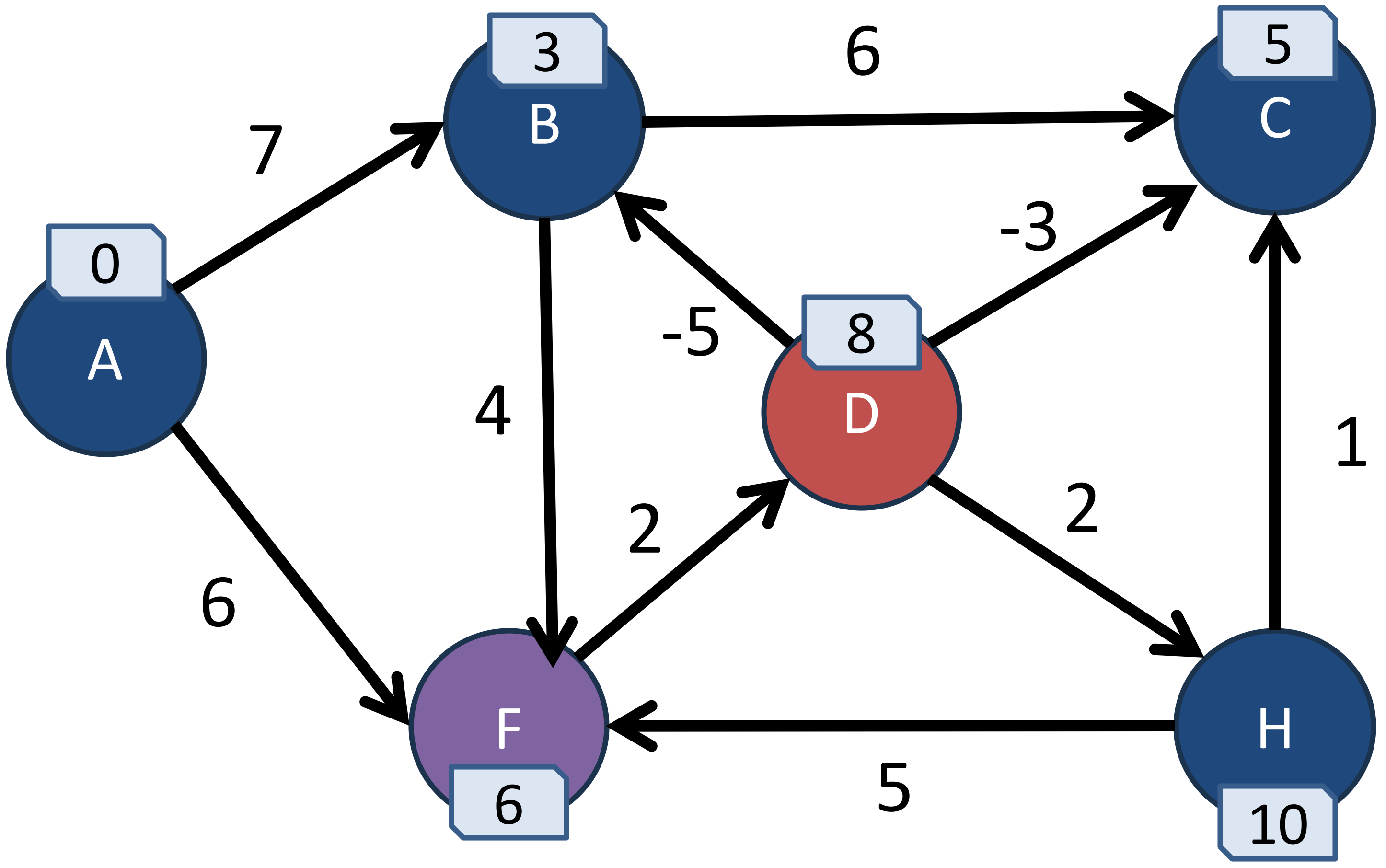
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



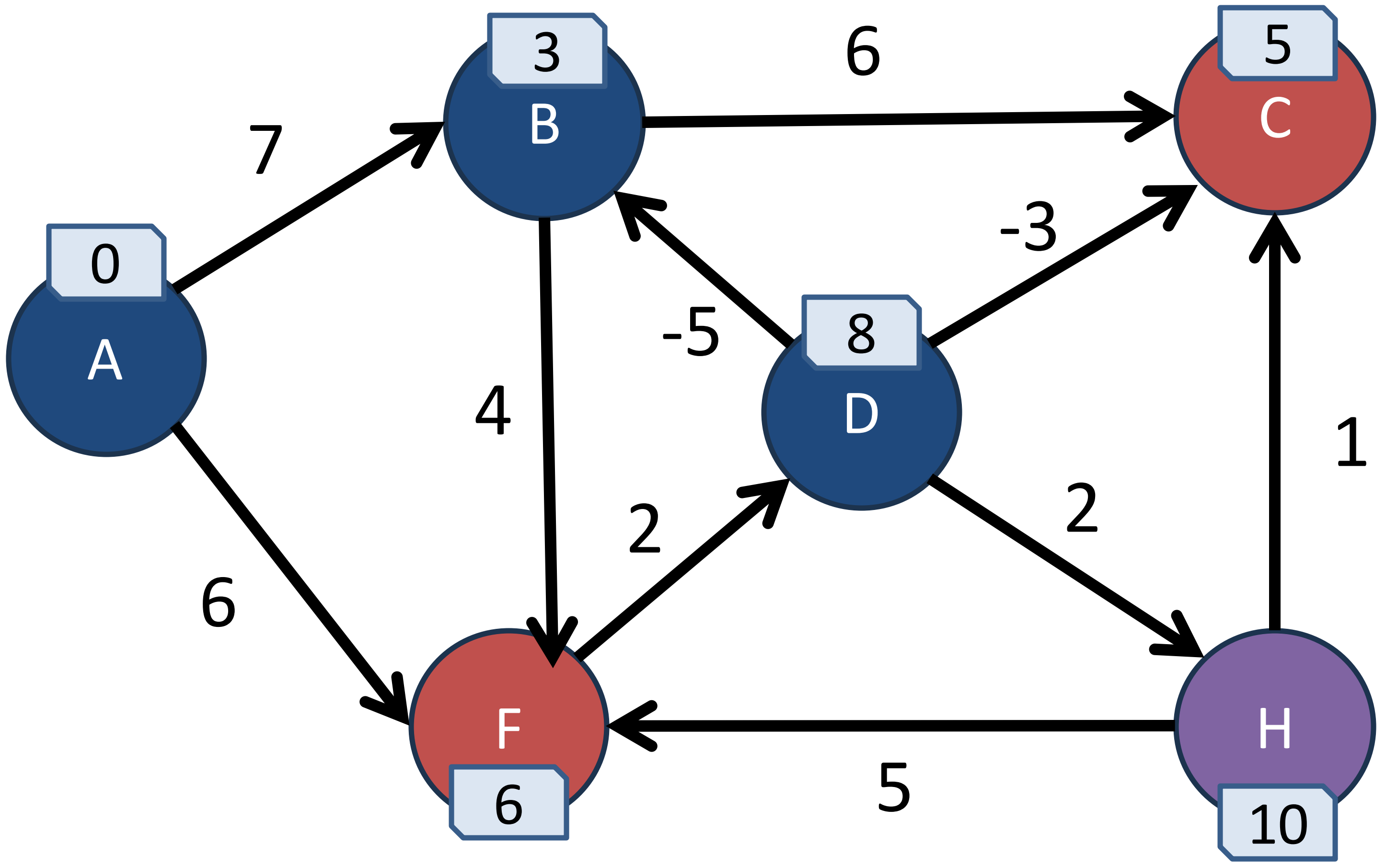
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



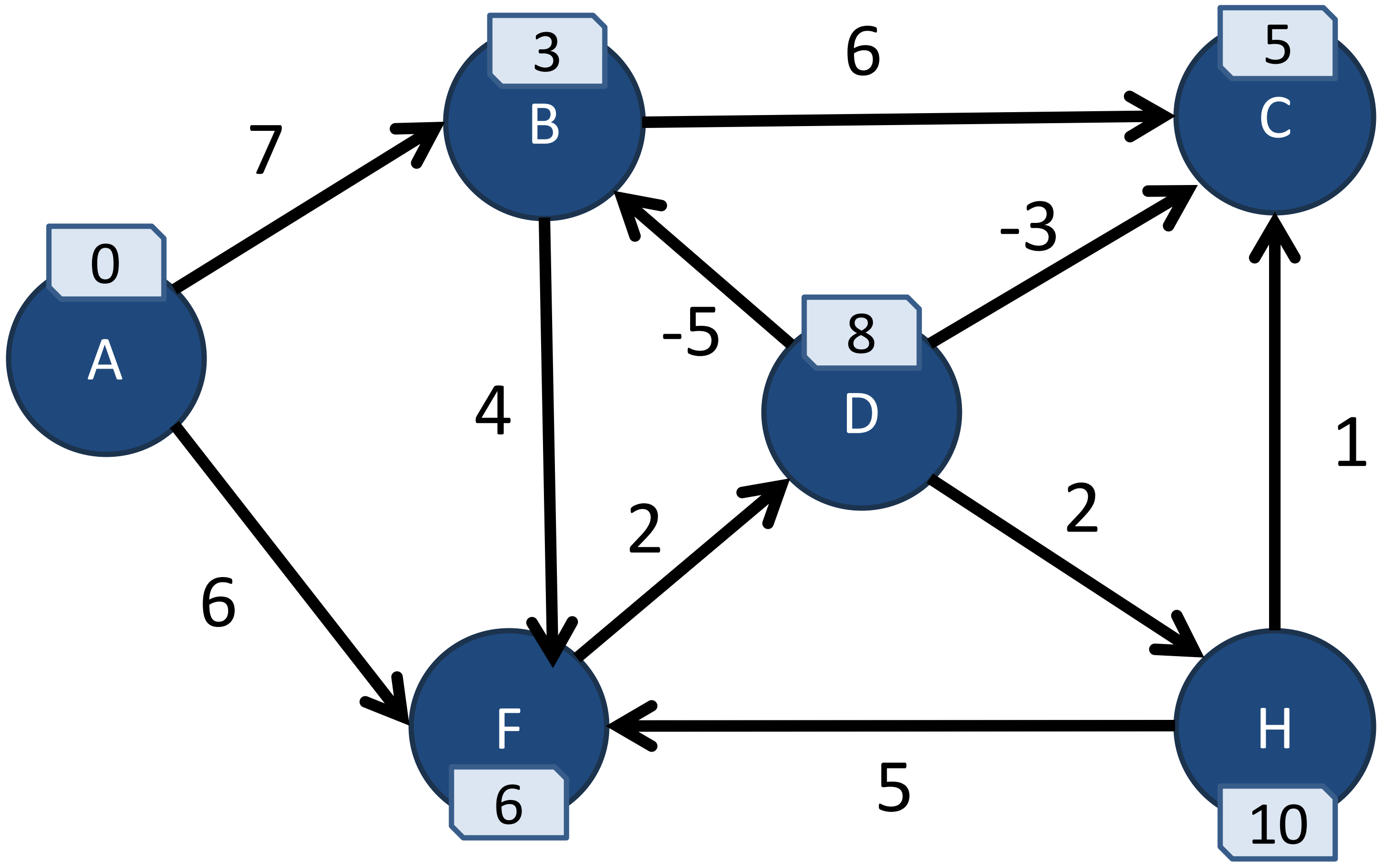
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



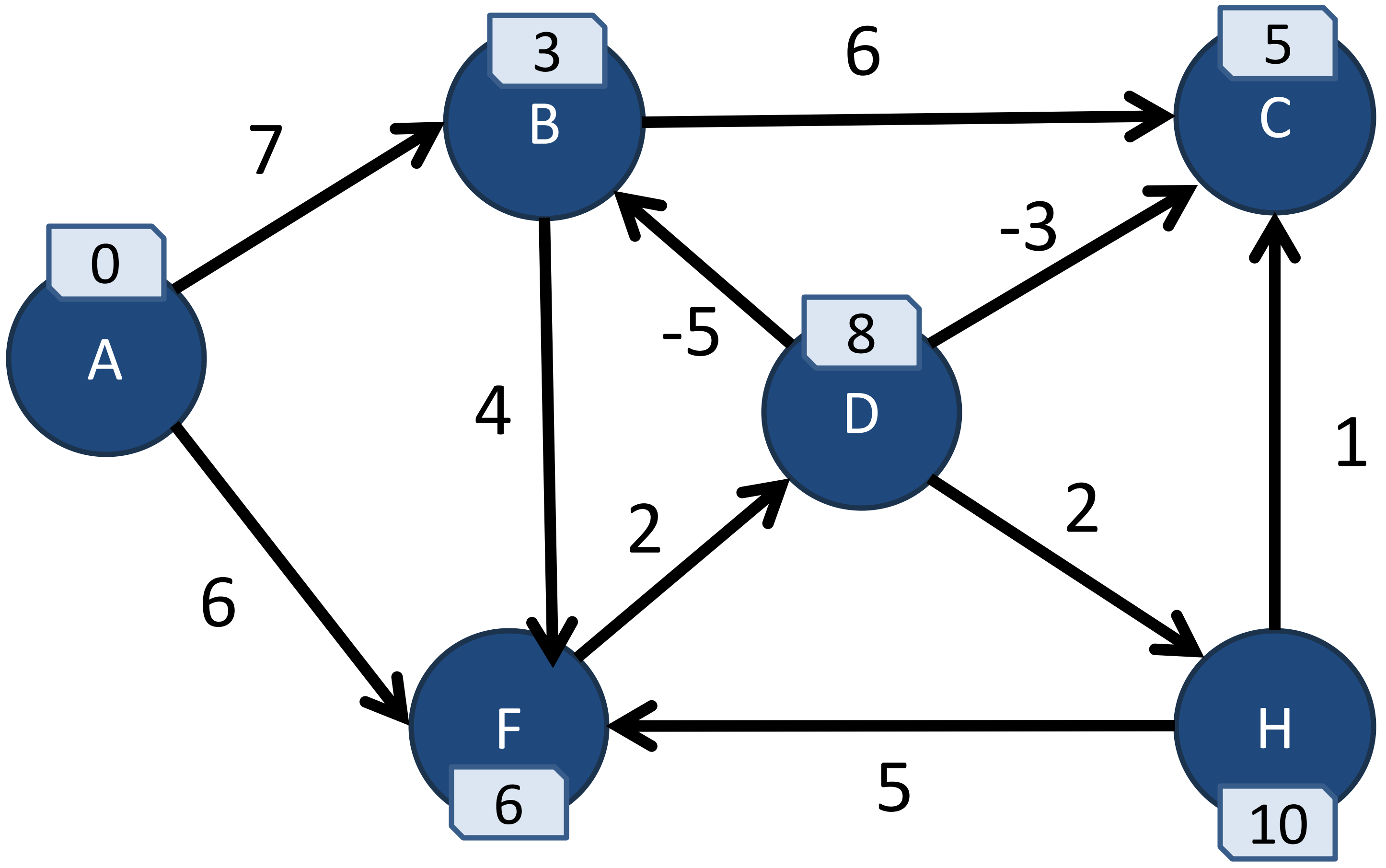
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

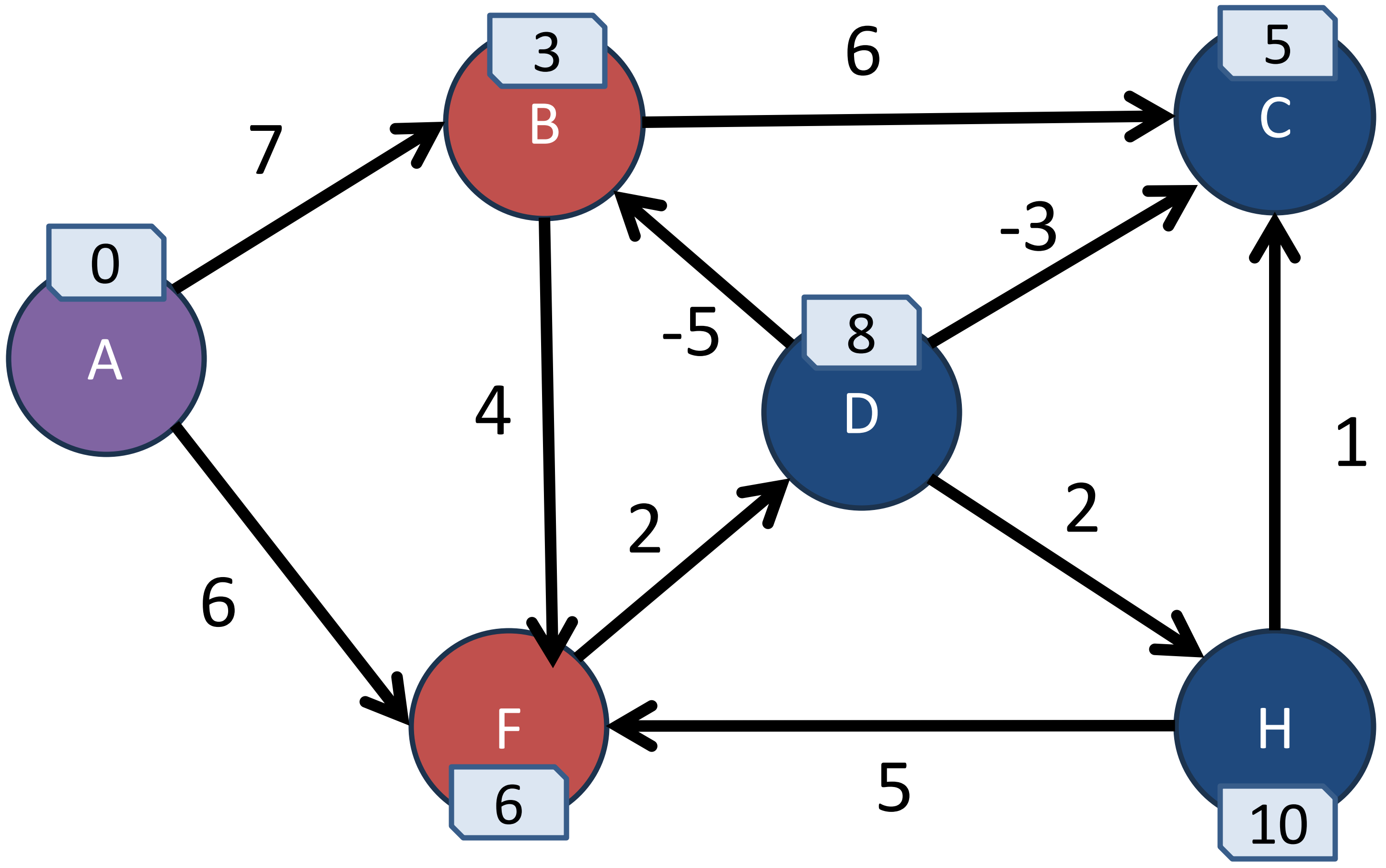
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	**0**	
B	**3**	**D**
C	**5**	**D**
D	**8**	**F**
F	**6**	**A**
H	**10**	**D**

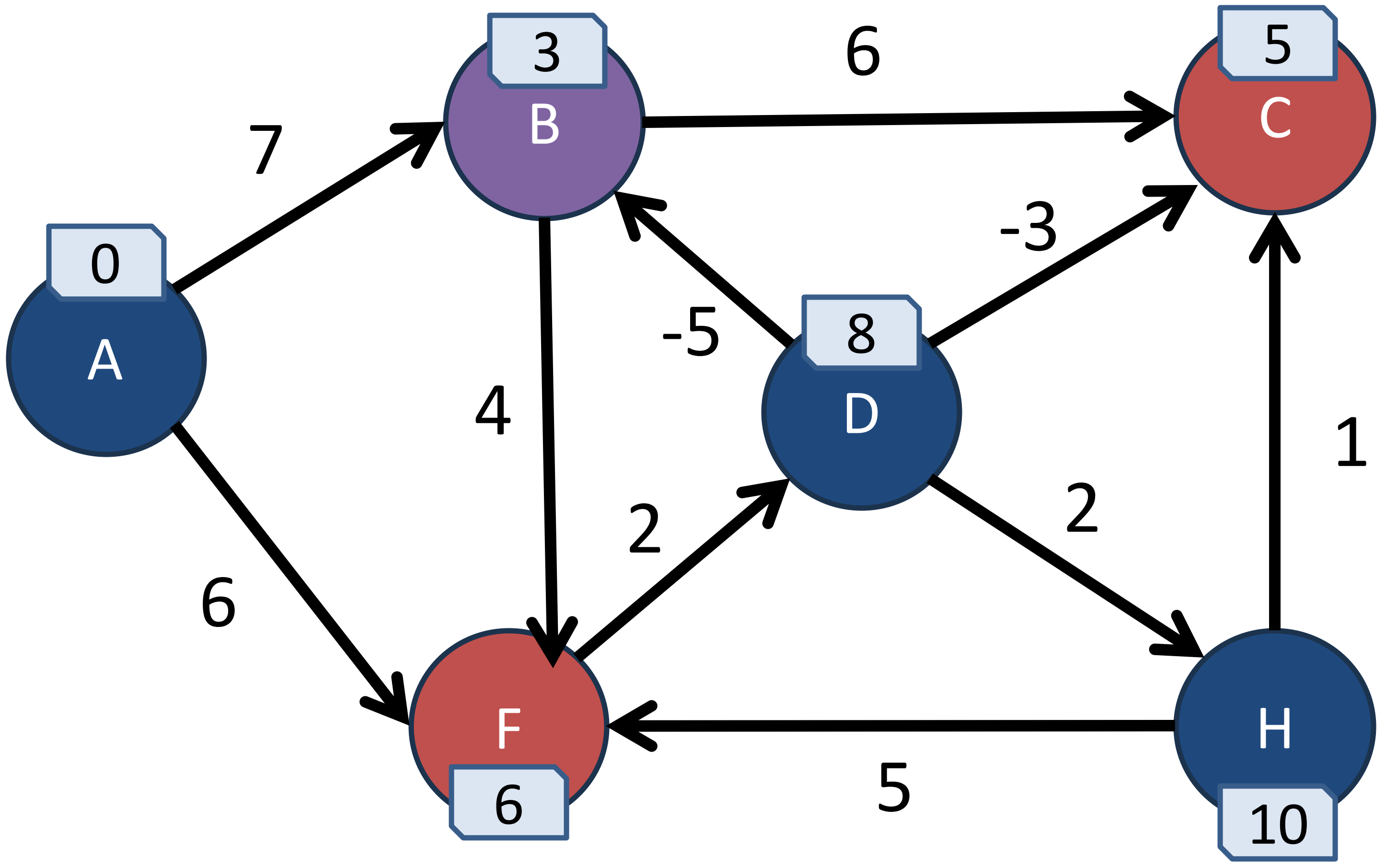
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC

Check for negative cycles



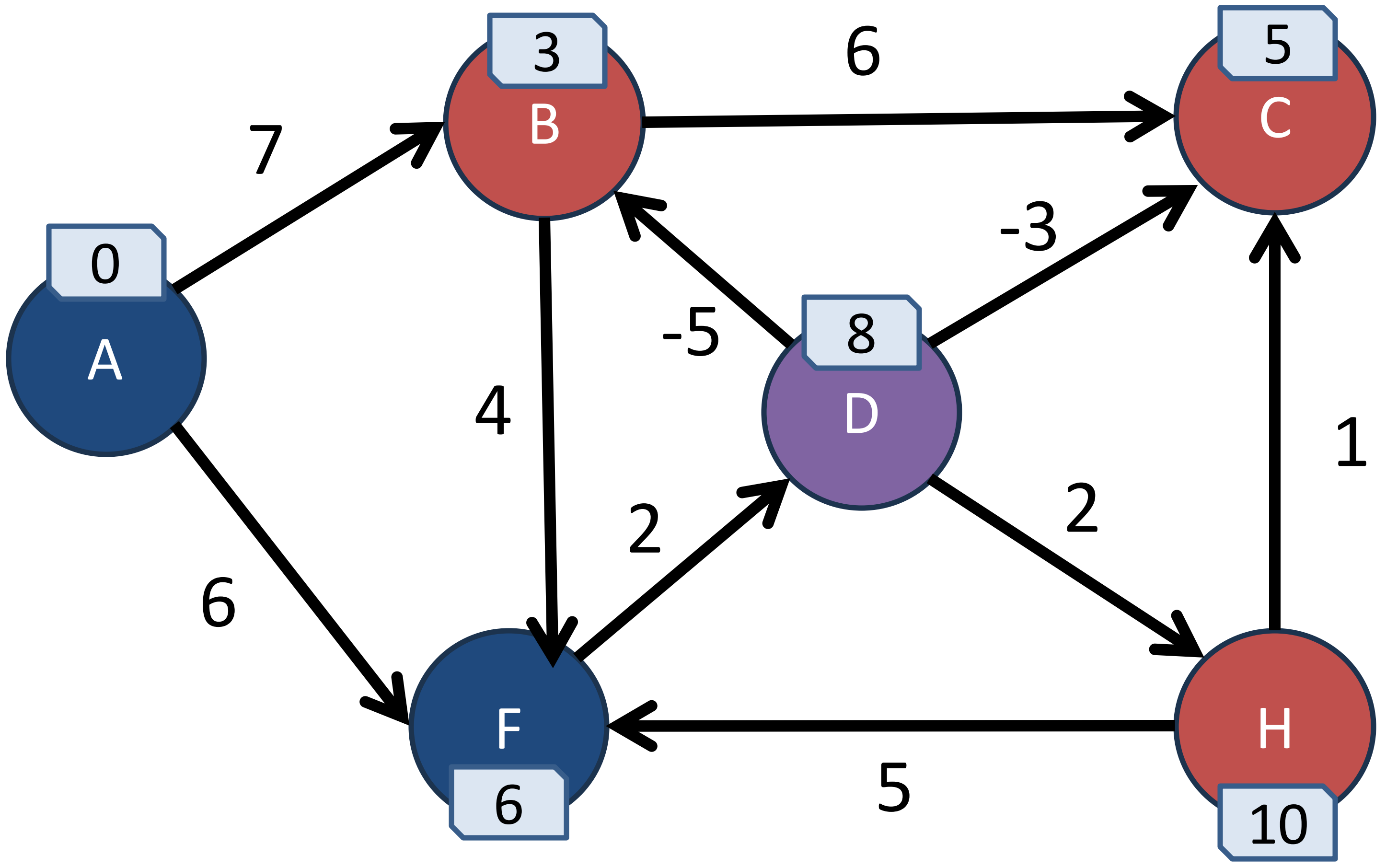
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



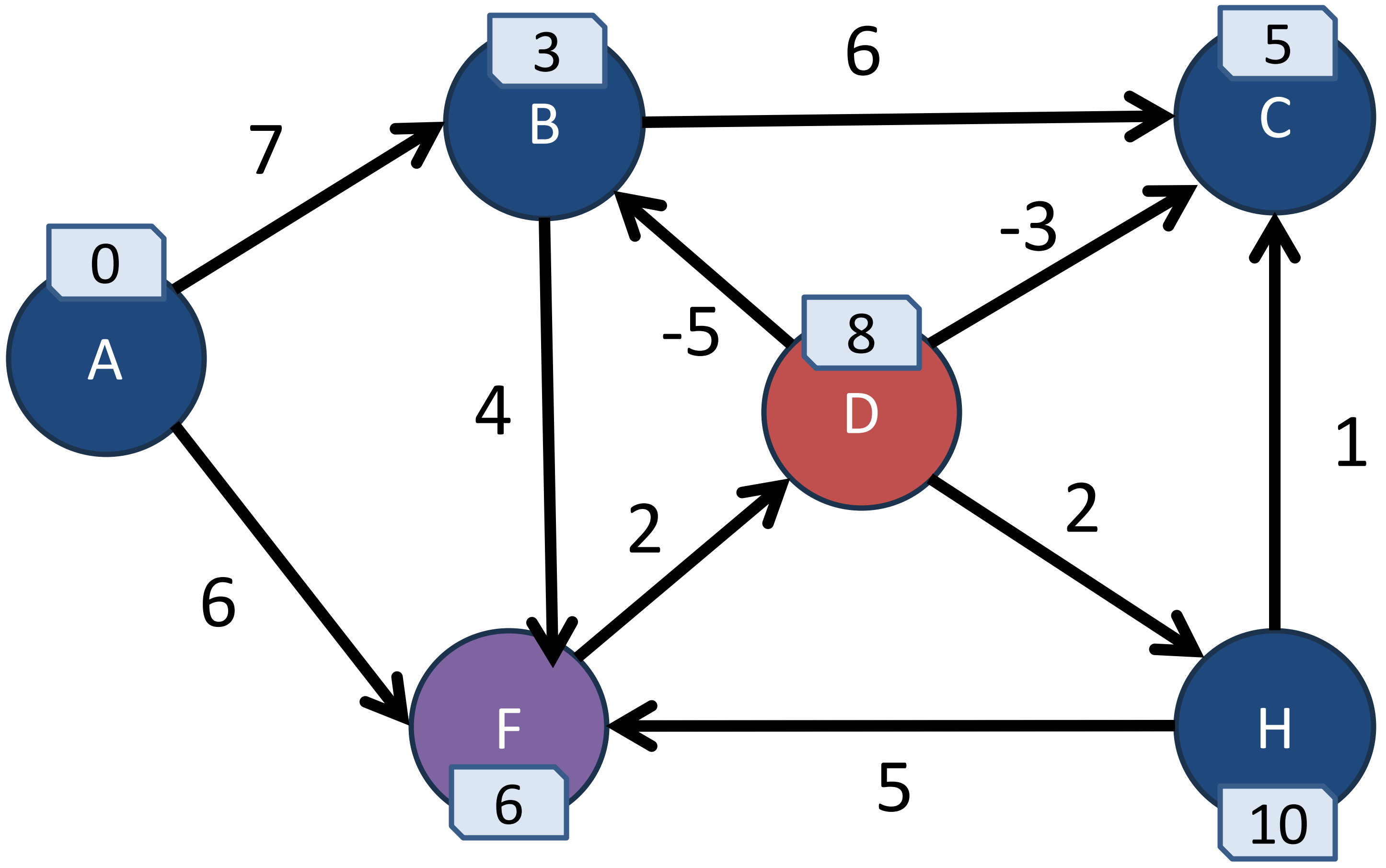
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



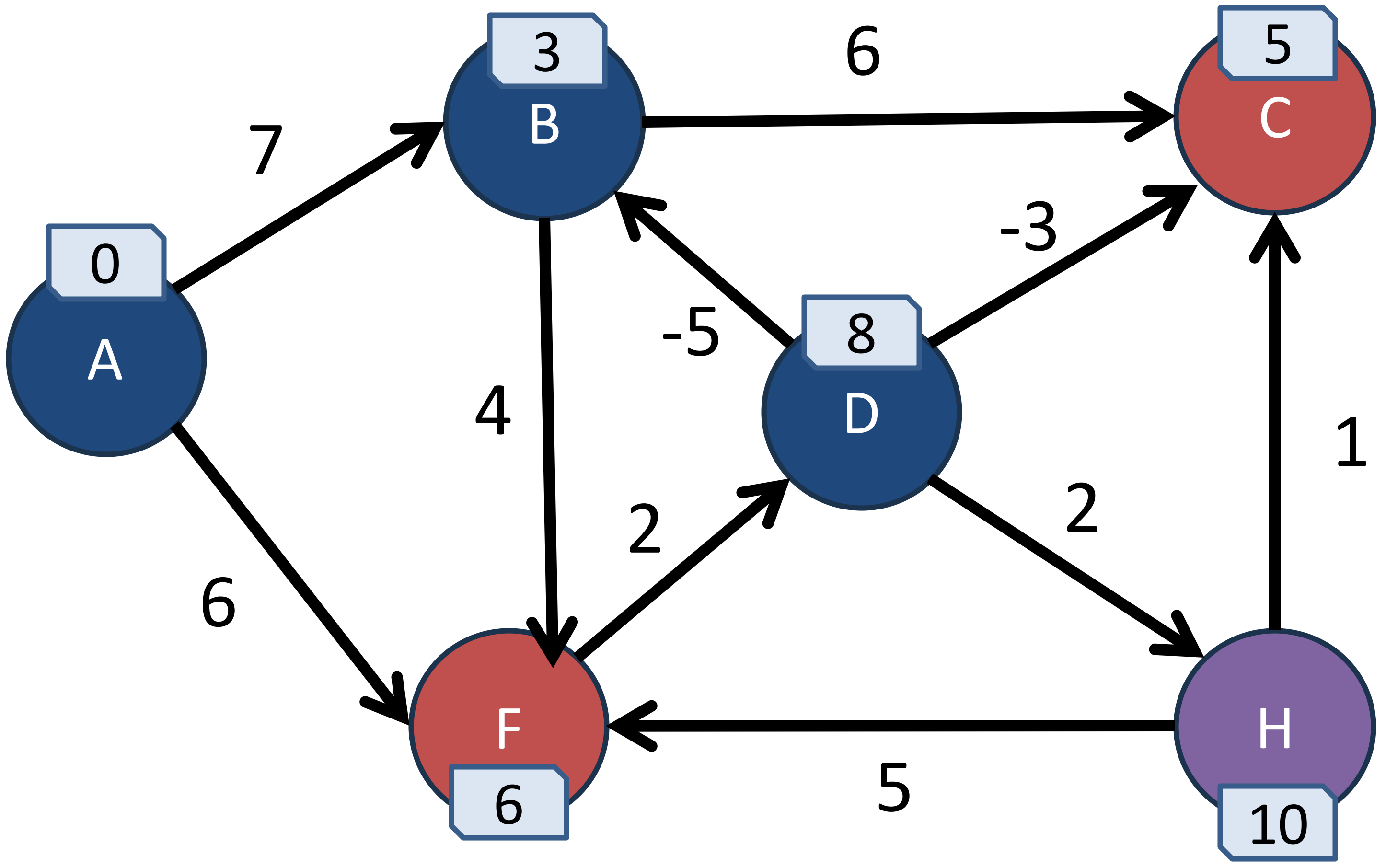
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



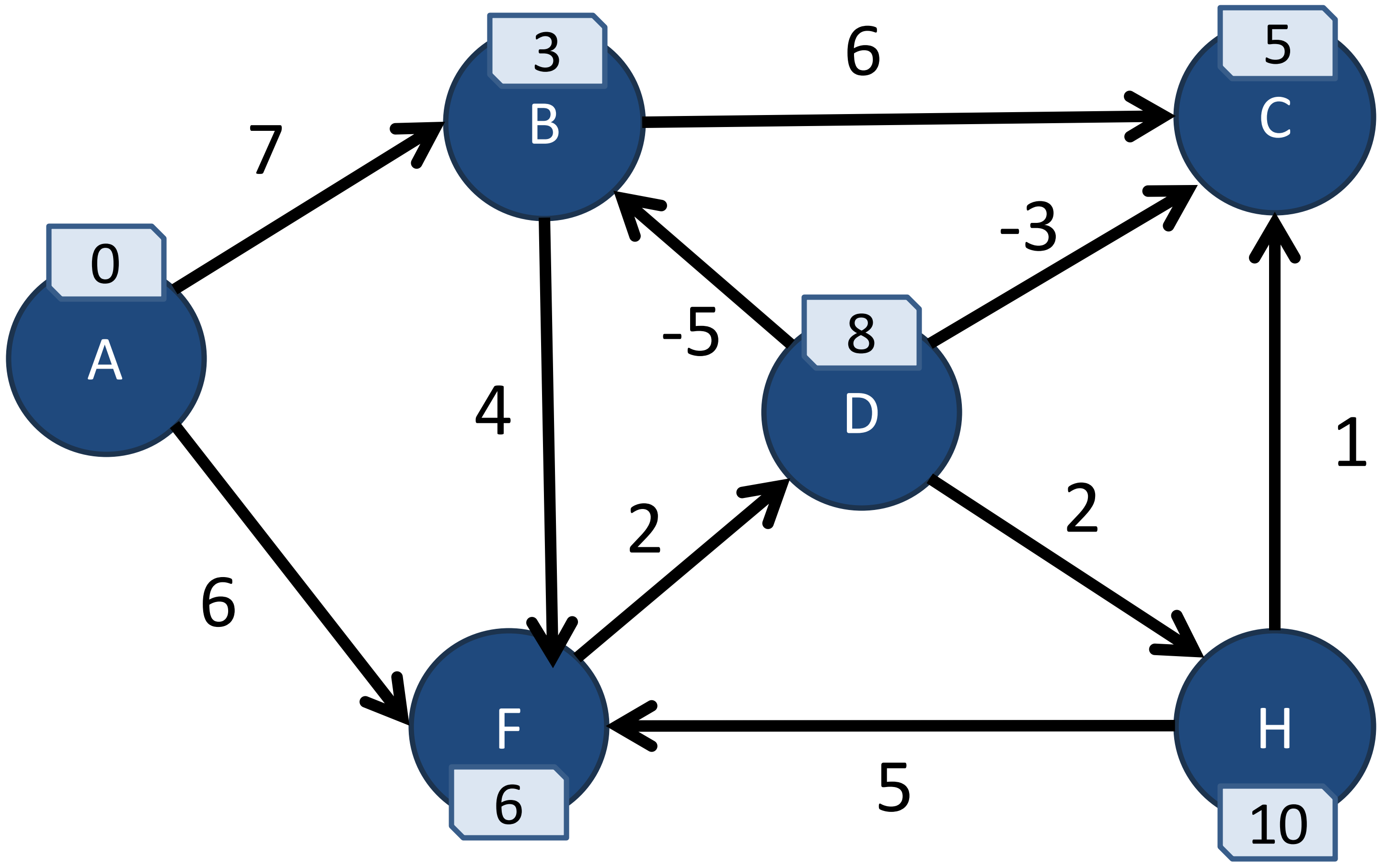
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

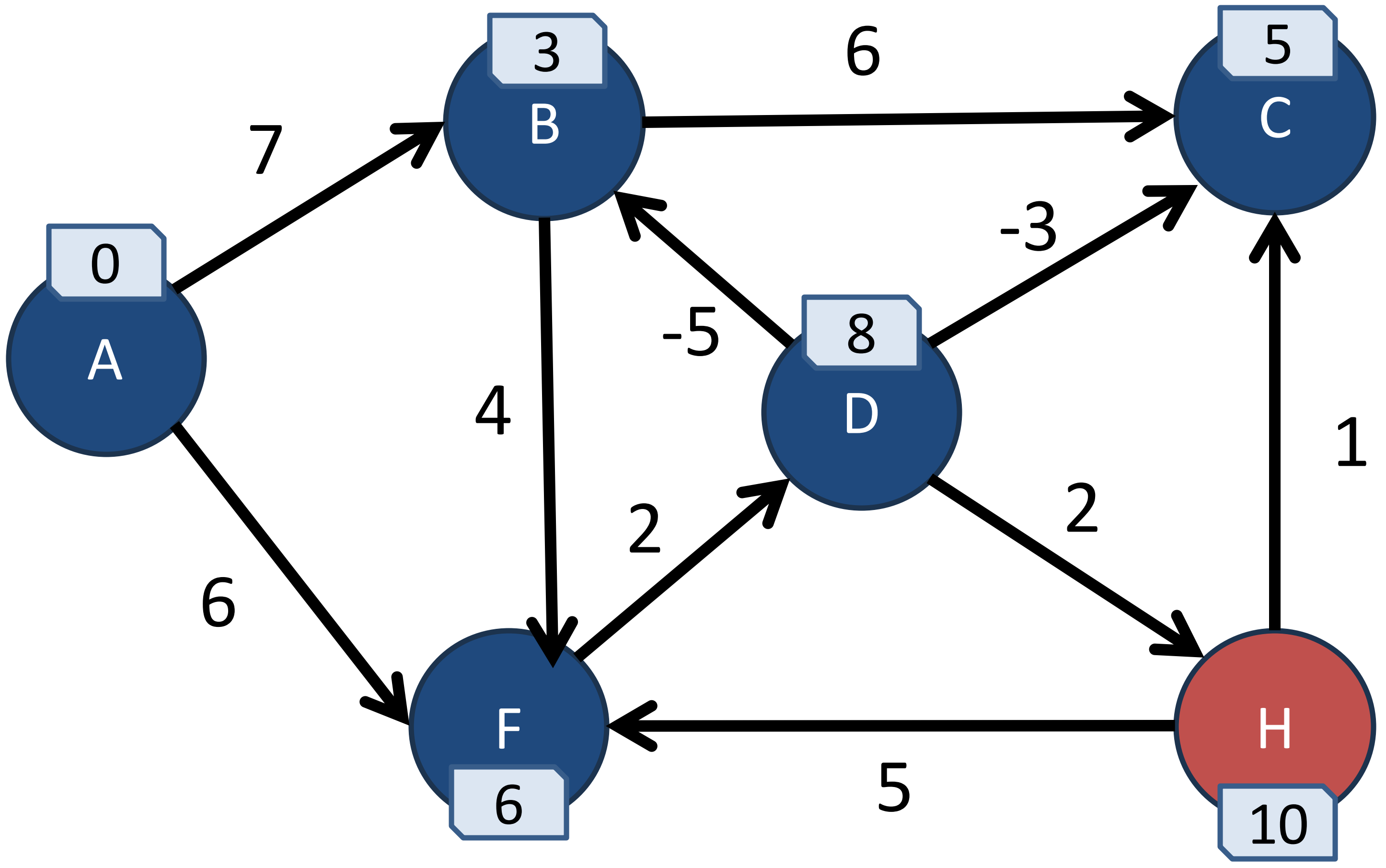
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

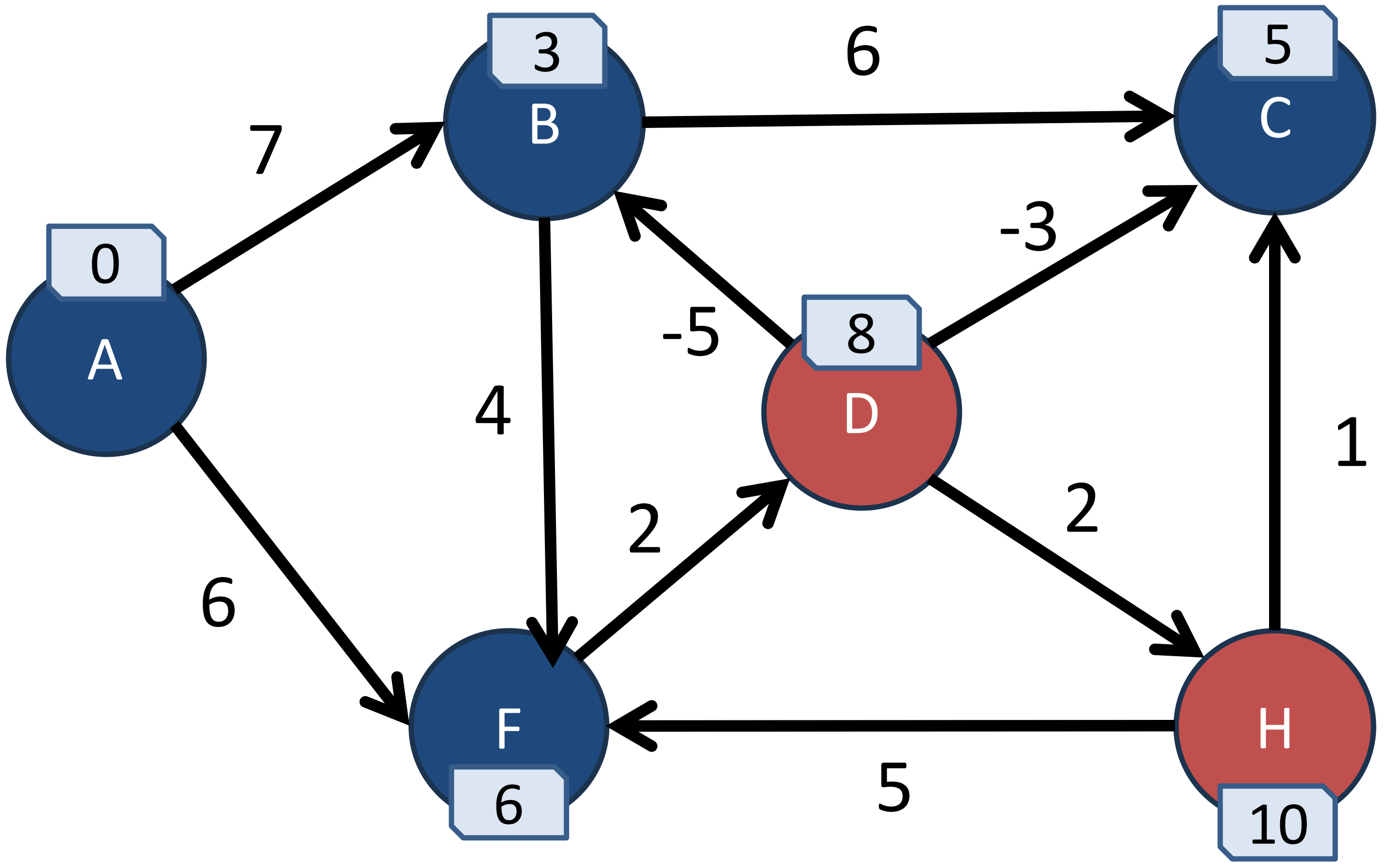
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC

Let’s trace the path



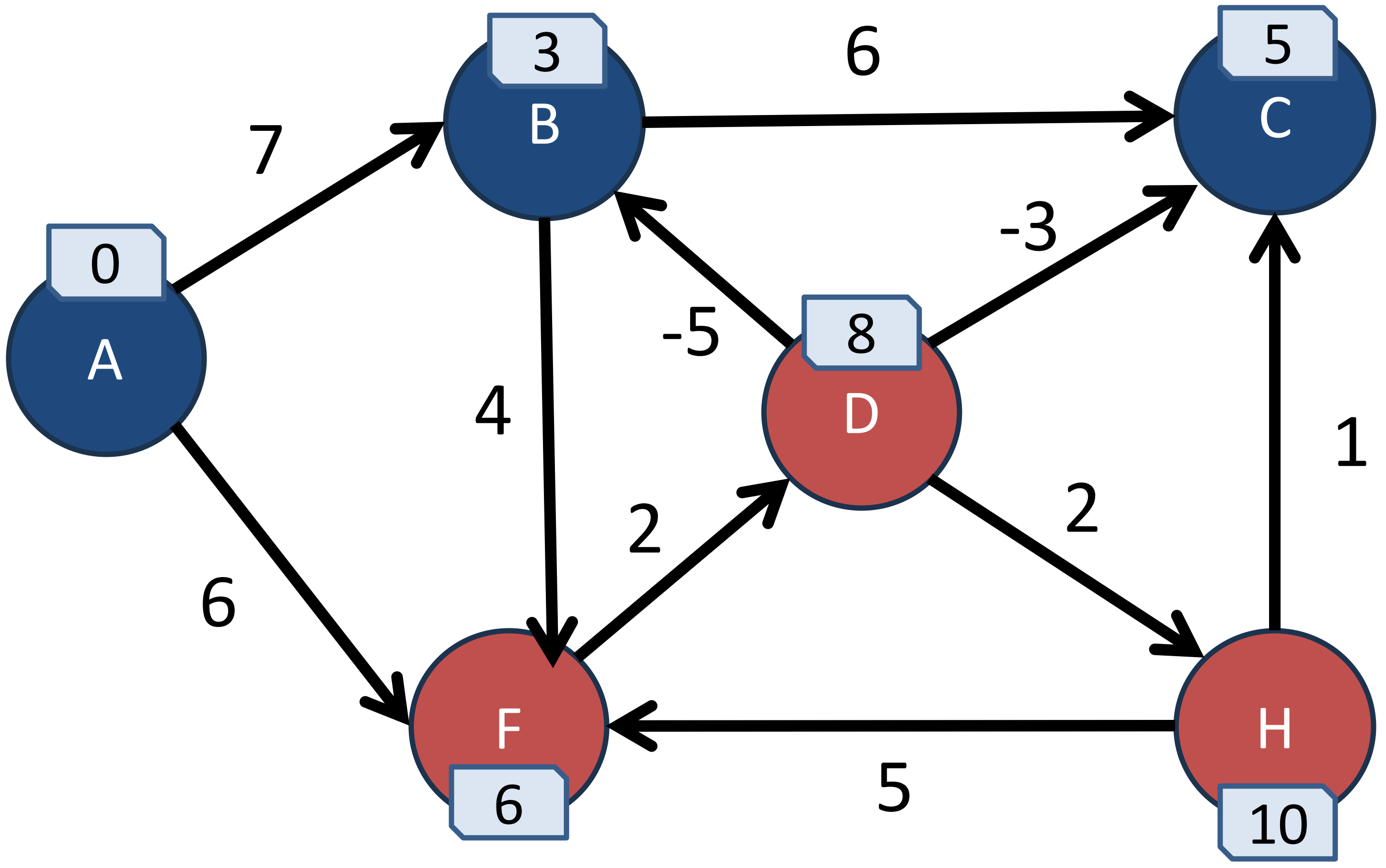
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



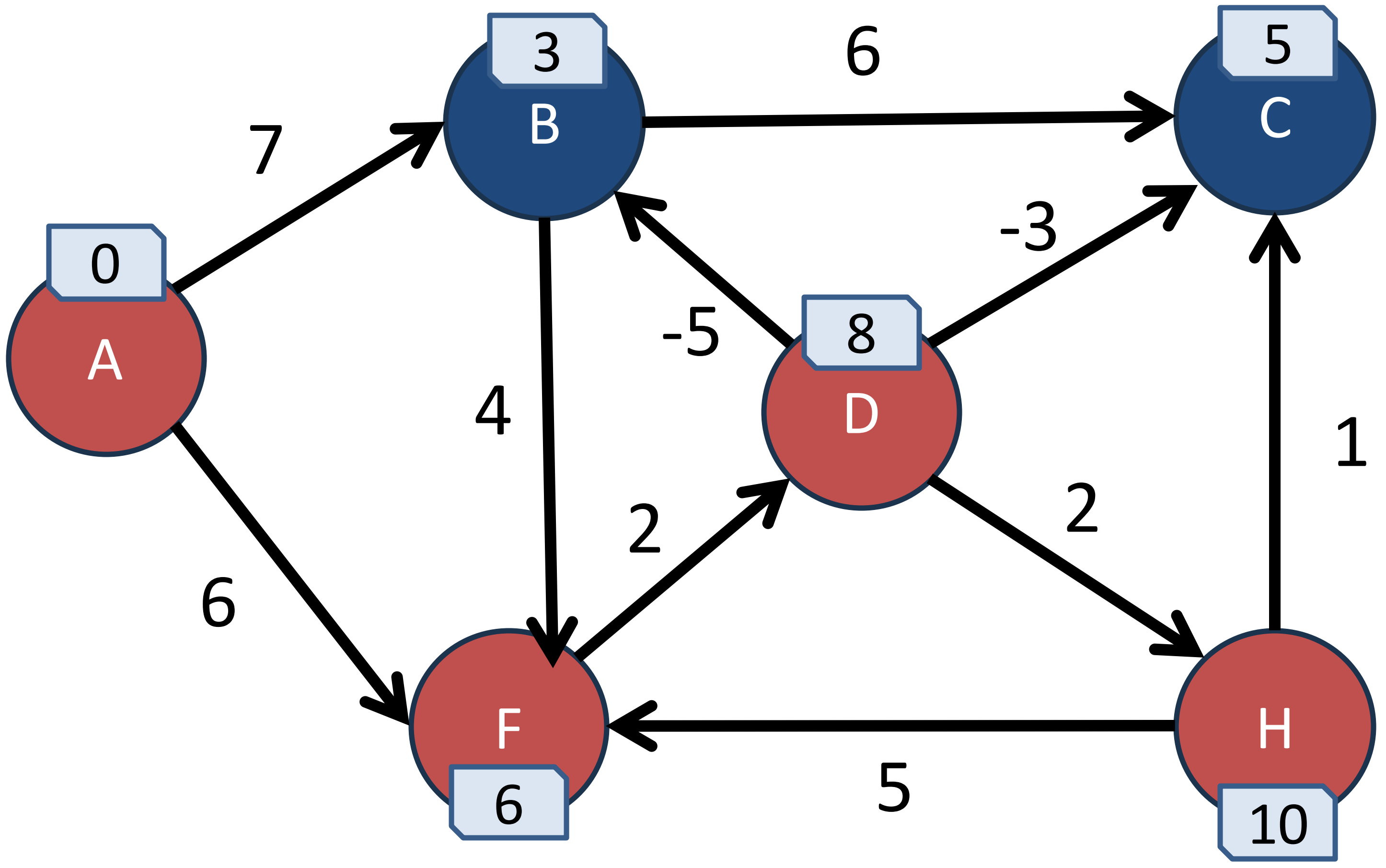
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



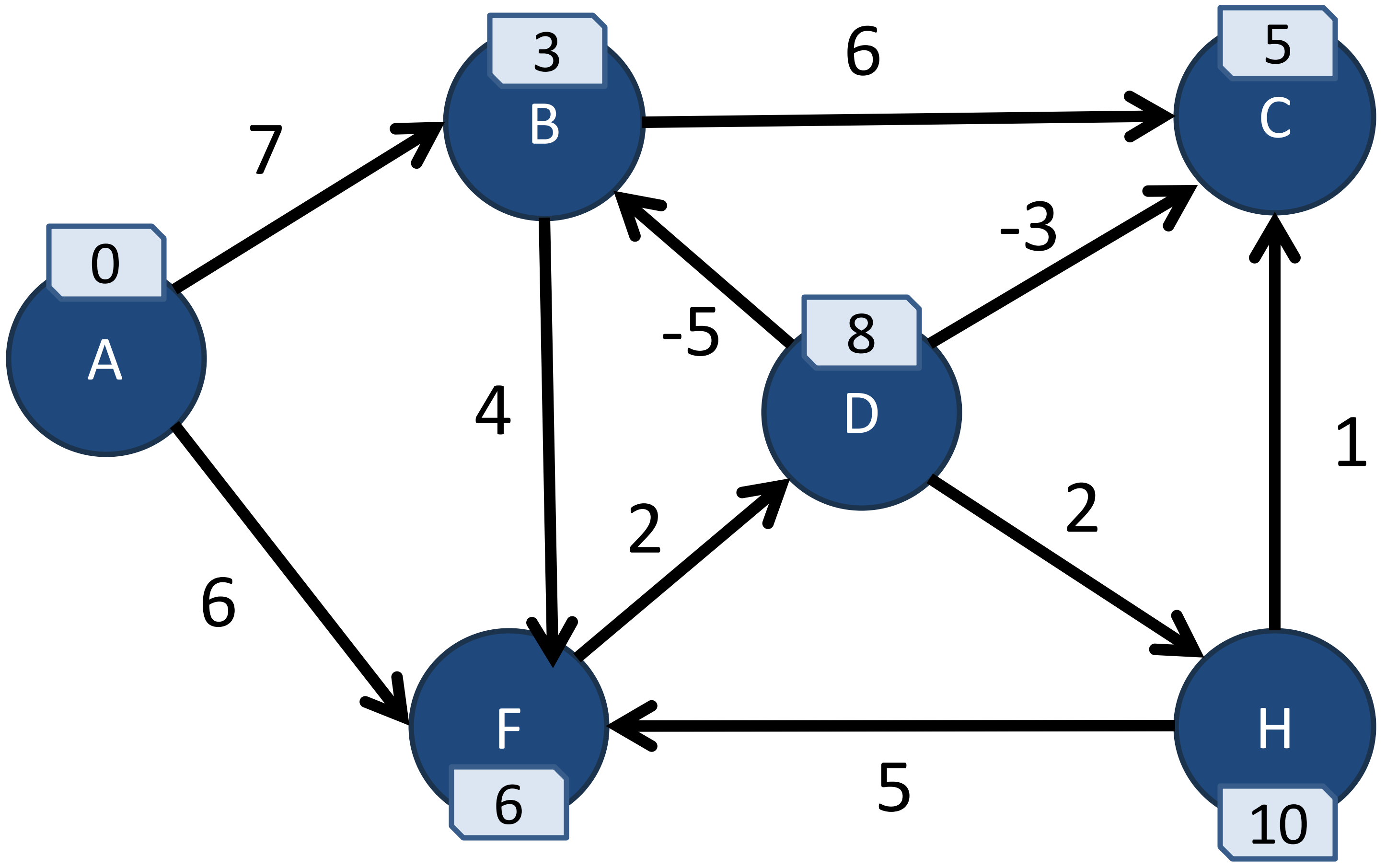
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



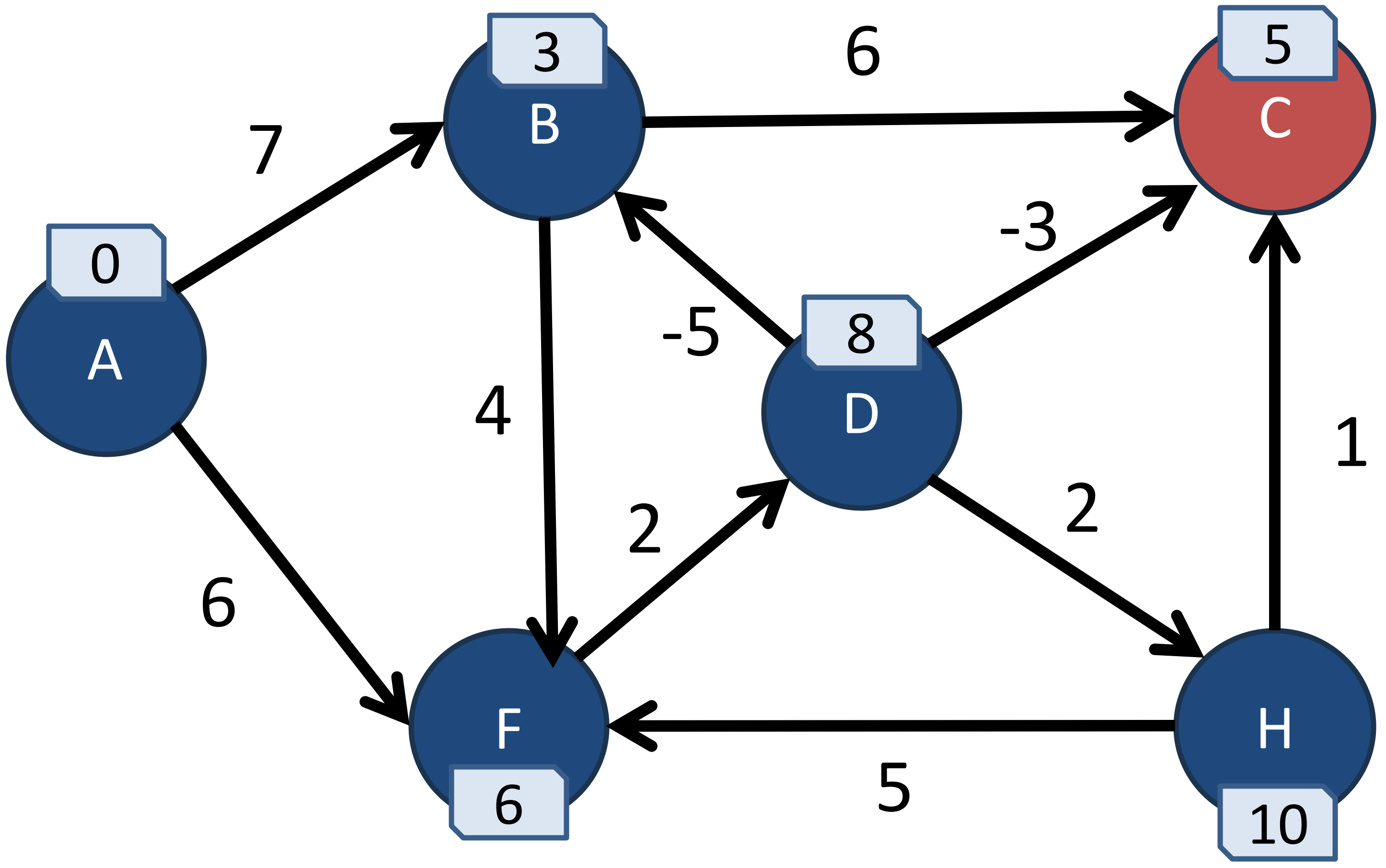
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



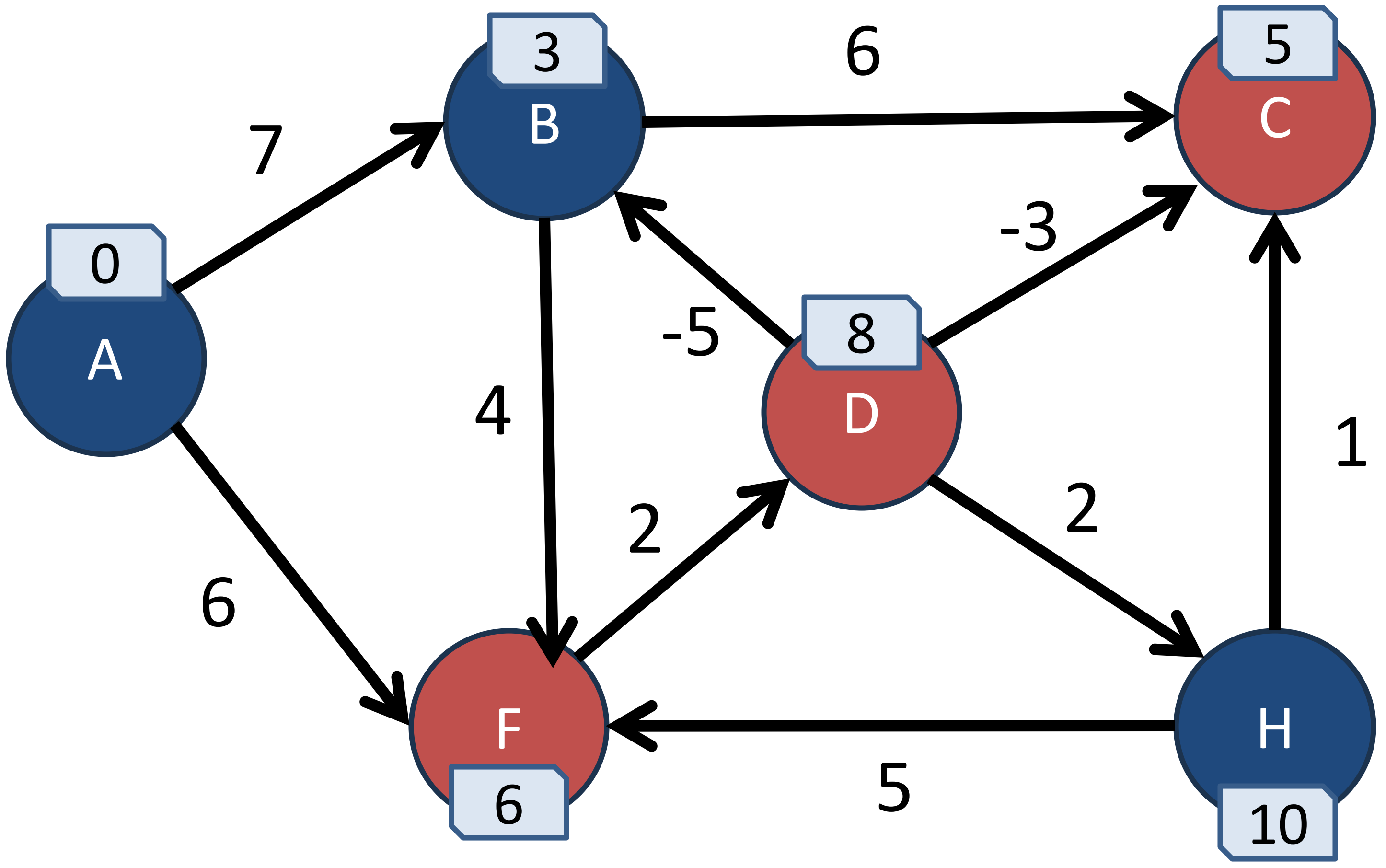
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



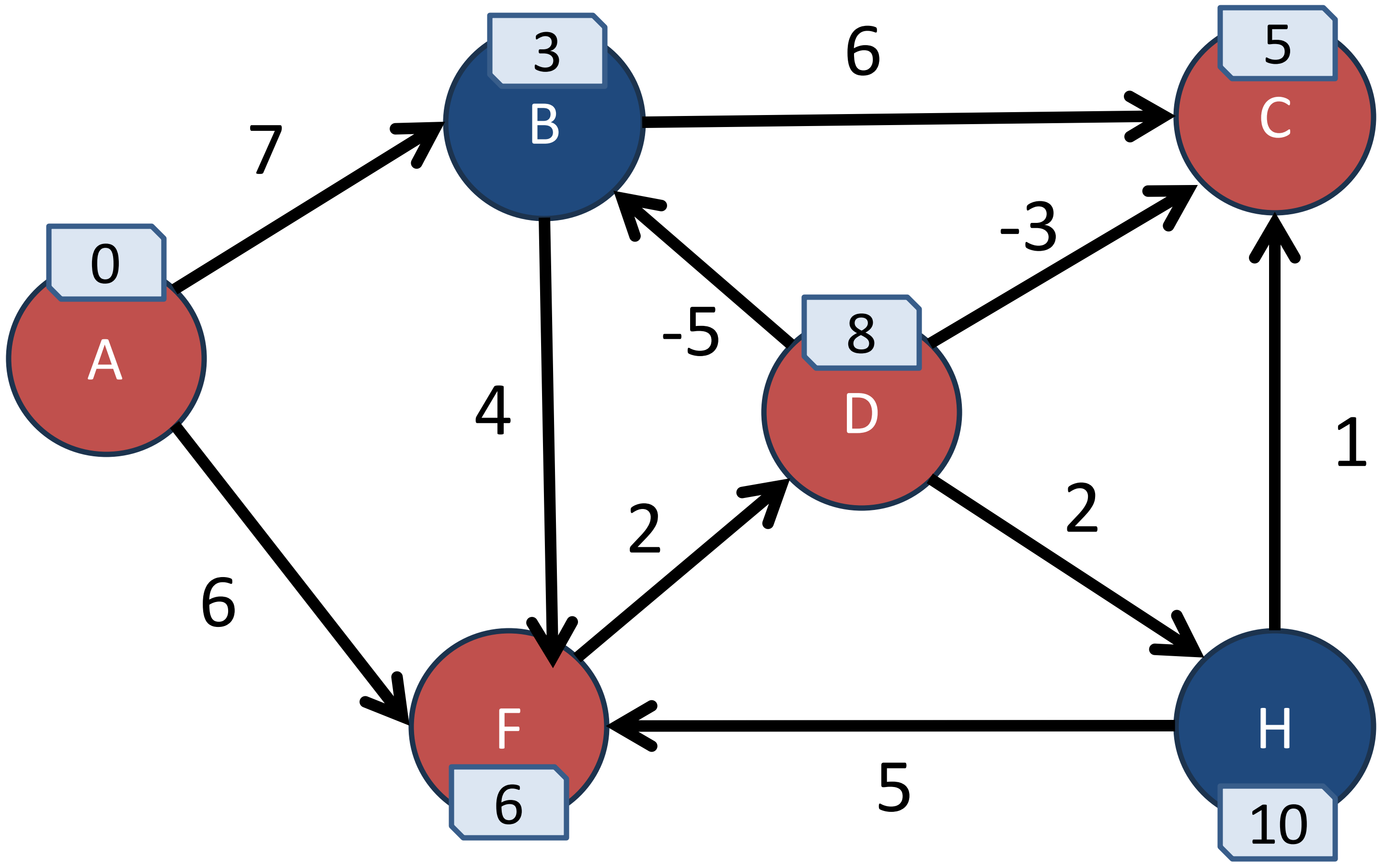
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

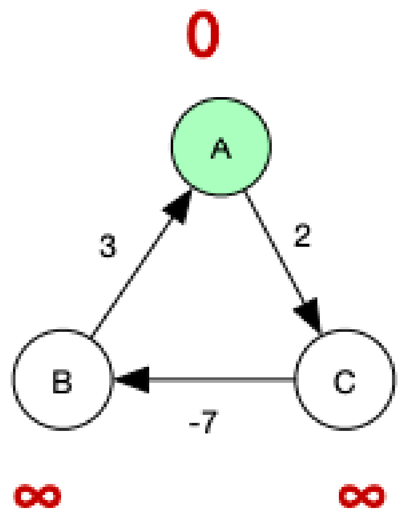
AB	AF	BC	BF	DB	DC	DH	FD	HF	HC



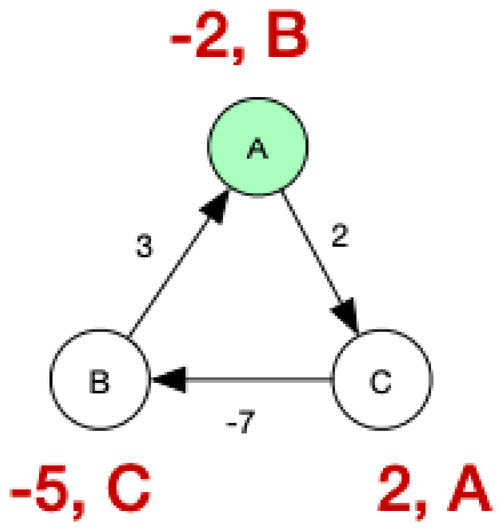
	Cost	Previous
A	0	
B	3	D
C	5	D
D	8	F
F	6	A
H	10	D

AB	AF	BC	BF	DB	DC	DH	FD	HF	HC

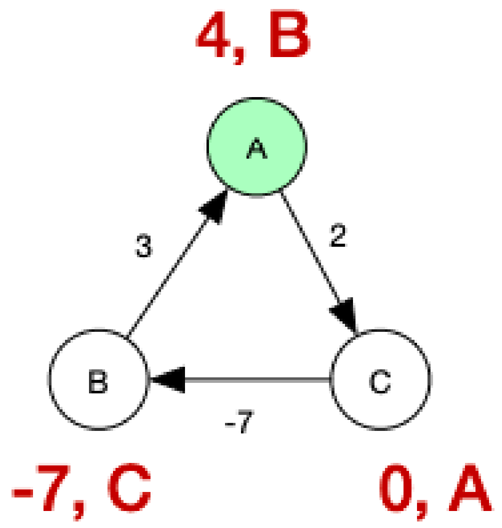
How does bellman-ford detect negative cycles?



initial state



after 1 iteration



after 2 iterations

$4 > -7 + 3$

Why this works

- If all shortest paths were found in V−1 iterations, no edge should need further relaxation.
- If further relaxation is possible, it means a **path can be made cheaper** — which is only possible if there's a **cycle with a net negative weight**.

Bellman-Ford algorithm

```
BELLMAN-FORD($G, s$)
for all $v \in V$
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ to $|V| - 1$
        for all edges $(u, v) \in E$
                if $dist[v] > dist[u] + w(u, v)$
                        $dist[v] \leftarrow dist[u] + w(u, v)$
                        $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
                return $false$
```

O(V)

Initialize all the distances

Bellman-Ford algorithm

```
BELLMAN-FORD$(G, s)$
1   for all $v \in V$
2           $dist[v] \leftarrow \infty$
3           $prev[v] \leftarrow null$
4   $dist[s] \leftarrow 0$
5   for $i \leftarrow 1$ to $|V| - 1$
6           for all edges $(u, v) \in E$
7                   if $dist[v] > dist[u] + w(u, v)$
8                           $dist[v] \leftarrow dist[u] + w(u, v)$
9                           $prev[v] \leftarrow u$
10  for all edges $(u, v) \in E$
11          if $dist[v] > dist[u] + w(u, v)$
12                  return $false$
```

O(V) (lines 1–3)

O(V) (line 5)

O(E) (line 6)

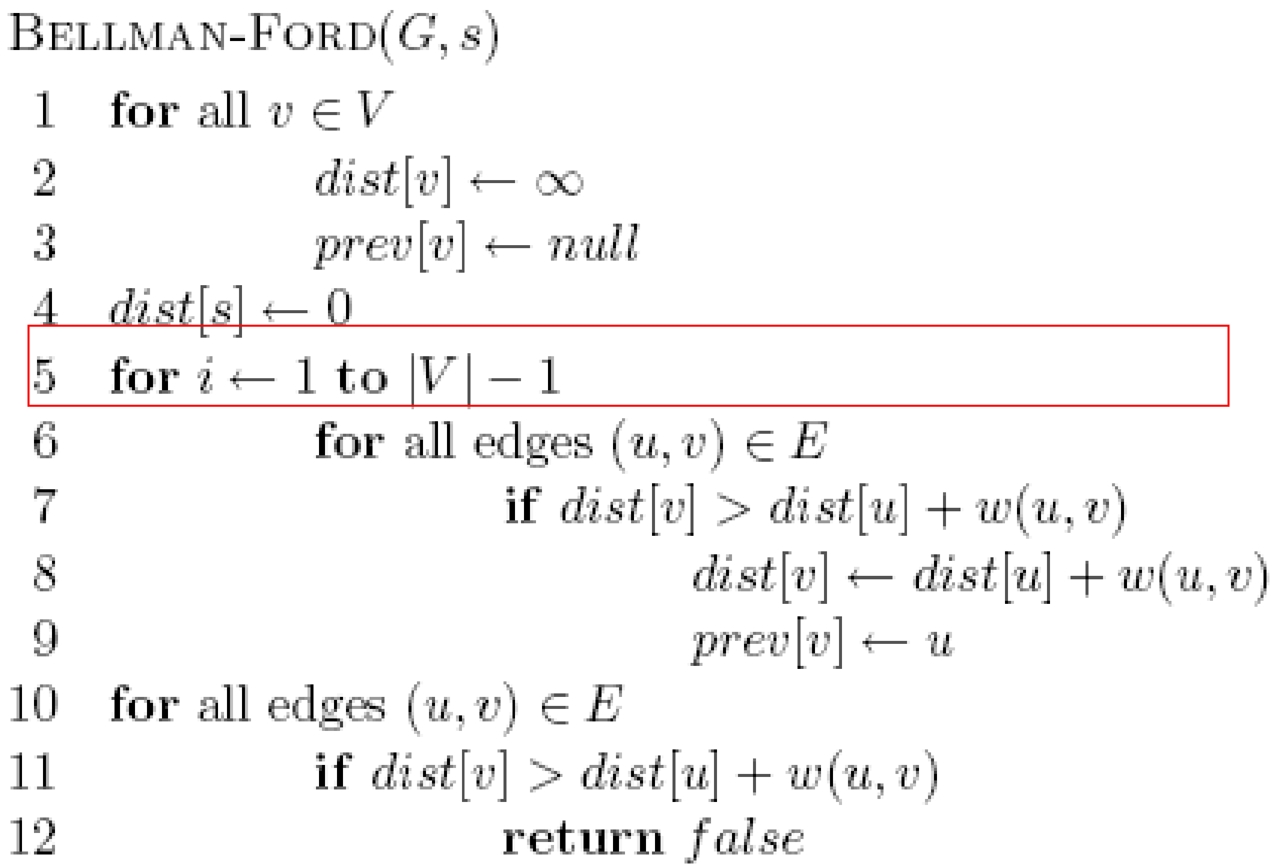
O(VE) (lines 5–9)

iterate over all edges/vertices and apply update rule

Bellman-Ford algorithm

O(V)

O(VE)



```
BELLMAN-FORD$(G, s)$
for all $v \in V$
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ to $|V| - 1$
        for all edges $(u, v) \in E$
                if $dist[v] > dist[u] + w(u, v)$
                        $dist[v] \leftarrow dist[u] + w(u, v)$
                        $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
                return $false$
```

Bellman-Ford algorithm

```
BELLMAN-FORD($G, s$)
for all $v \in V$
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ to $|V| - 1$
        for all edges $(u, v) \in E$
                if $dist[v] > dist[u] + w(u, v)$
                        $dist[v] \leftarrow dist[u] + w(u, v)$
                        $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
                return $false$
```

O(V)

O(VE)

O(V)

check for negative cycles