Distances in Graphs

Distance Algorithms

- Calculating distances in graphs
  - Single source - single destination
  - Single source - all destinations
  - All sources - all destinations

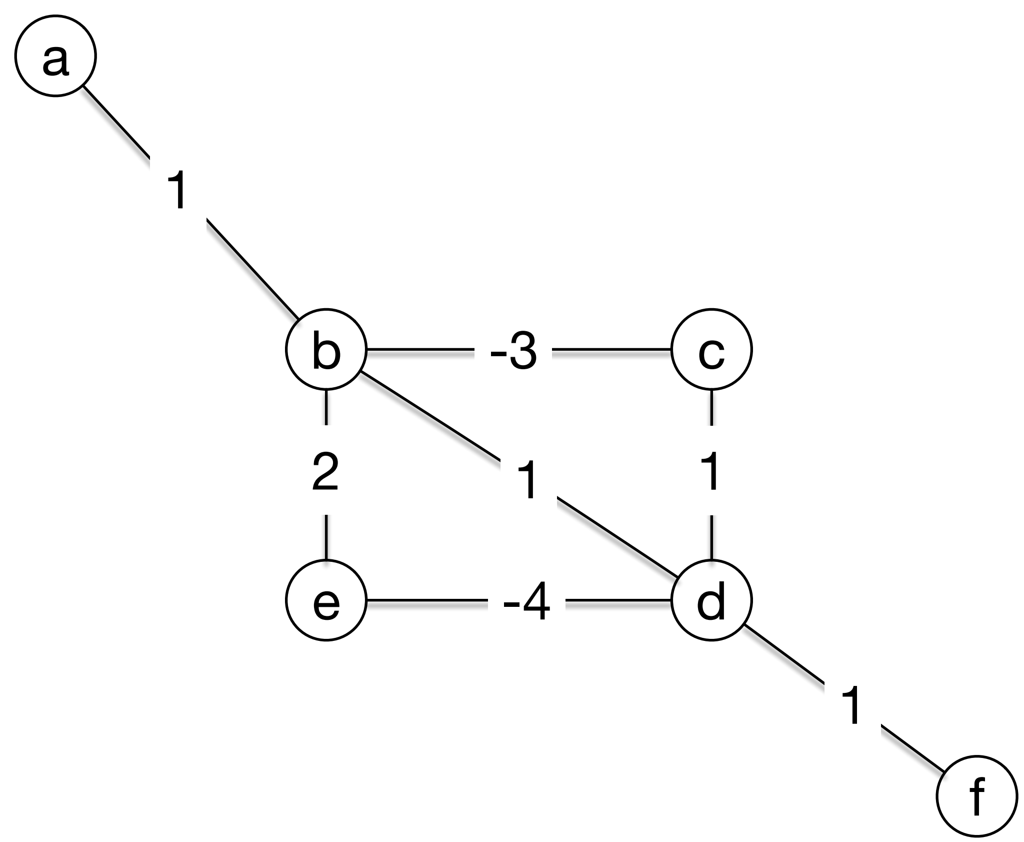
  - Directed Graphs
  - Undirected Graphs

Distance Algorithms

- Graph has only positive weights
- Graph can have negative weights
  - but not a negative cycle

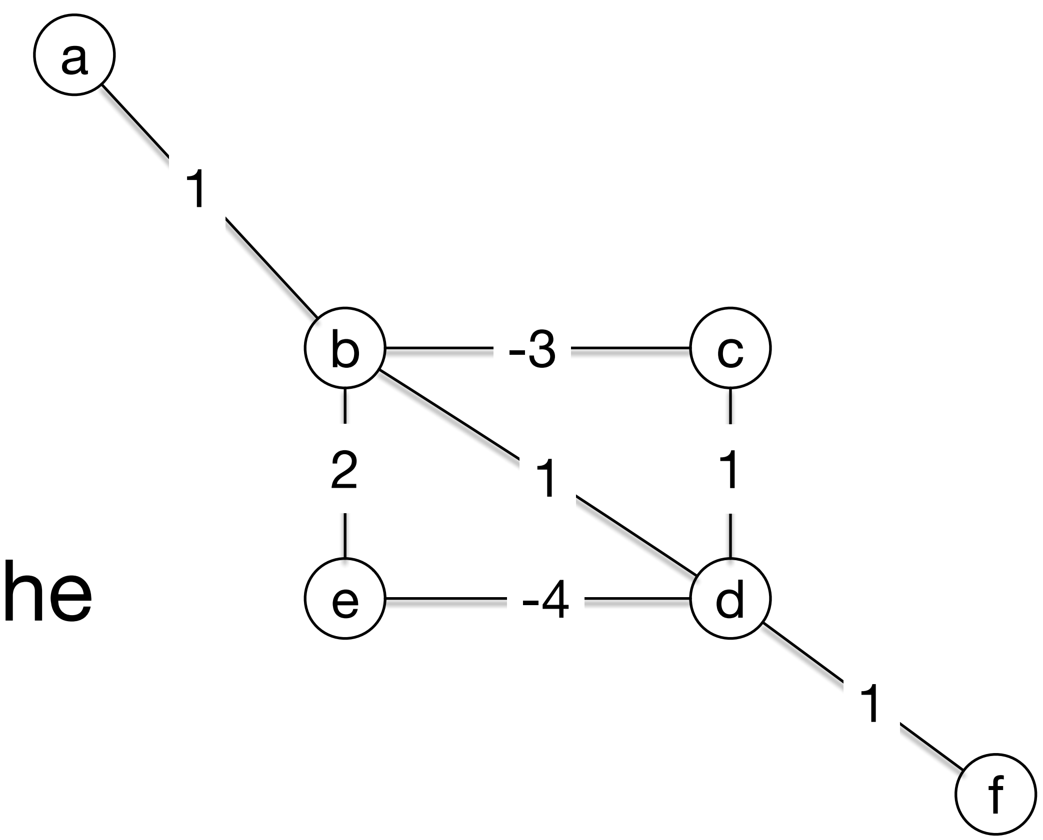
Distance Algorithms

- Negative cycle example:
  - What is the distance from a to f



Distance Algorithms

- a-b-d-f costs 3
- a-b-c-d-f costs 0
- a-b-c-d-e-d-e-b-d-f costs -2
- and a few more times around the cycle costs even less

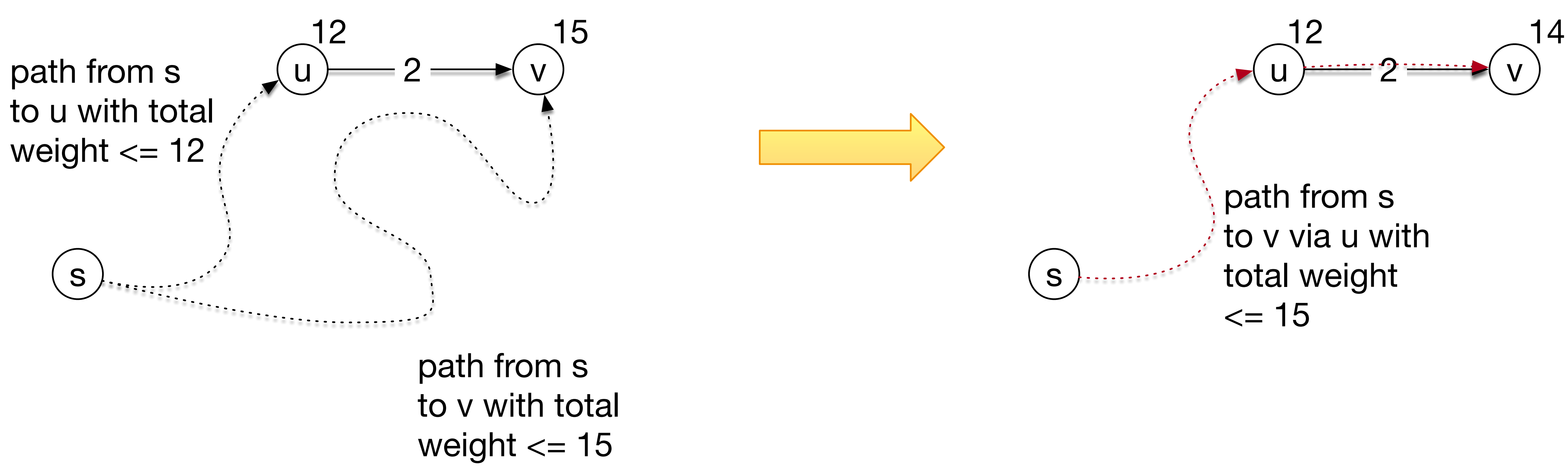


Distance Algorithms

- Single Source Algorithms:
- Relaxation
    - Fundamental approach to maintain estimates for distances
    - Assume for each vertex, we have an upper bound for the distance from the source
    - Relaxation then improves the distance bound towards the true value

Distance Algorithms

- Example



Distance Algorithms

- Relaxation:
  - $u.d$ distance bound for distance between s and u
  - Let $(u, v) \in E$. We relax along (u, v) by setting
    - $v.d \leftarrow \min\left(v.d, u.d + w(u, v)\right)$

Distance Algorithms

- Relaxation:
    - In addition, we can maintain a predecessor field at all nodes
    - Because we do not only want the distance, but also how we got there
    - When we relax, we set the predecessor field in v to u if we replace $v.d$ with $u.d + w(u, v)$ because the latter is smaller

Bellman-Ford

- Bellman-Ford Single Source Distance Algorithm
  - We initialize by:
    - Source s gets distance 0 and itself as predecessor
    - Every node u adjacent to s gets distance $w(s, u)$ and predecessor

Bellman-Ford

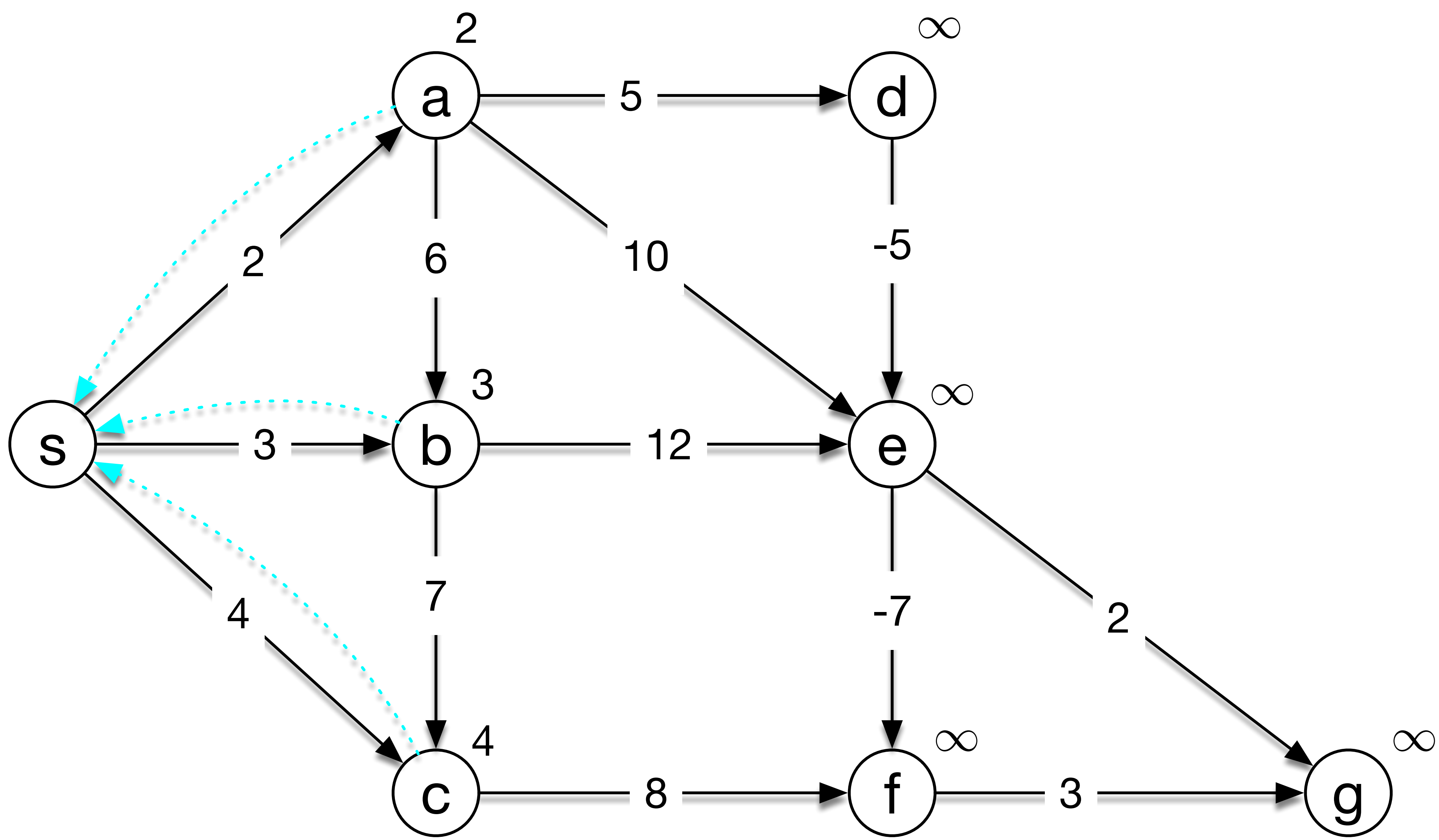
- Best path from source s to node u cannot have more than $|V| - 1$ edges in it
  - So, we relax with every edge a total of $|V| - 1$ times
  - If we can relax afterwards, something is fishy:
    - We have found evidence for a negative cycle

Bellman-Ford

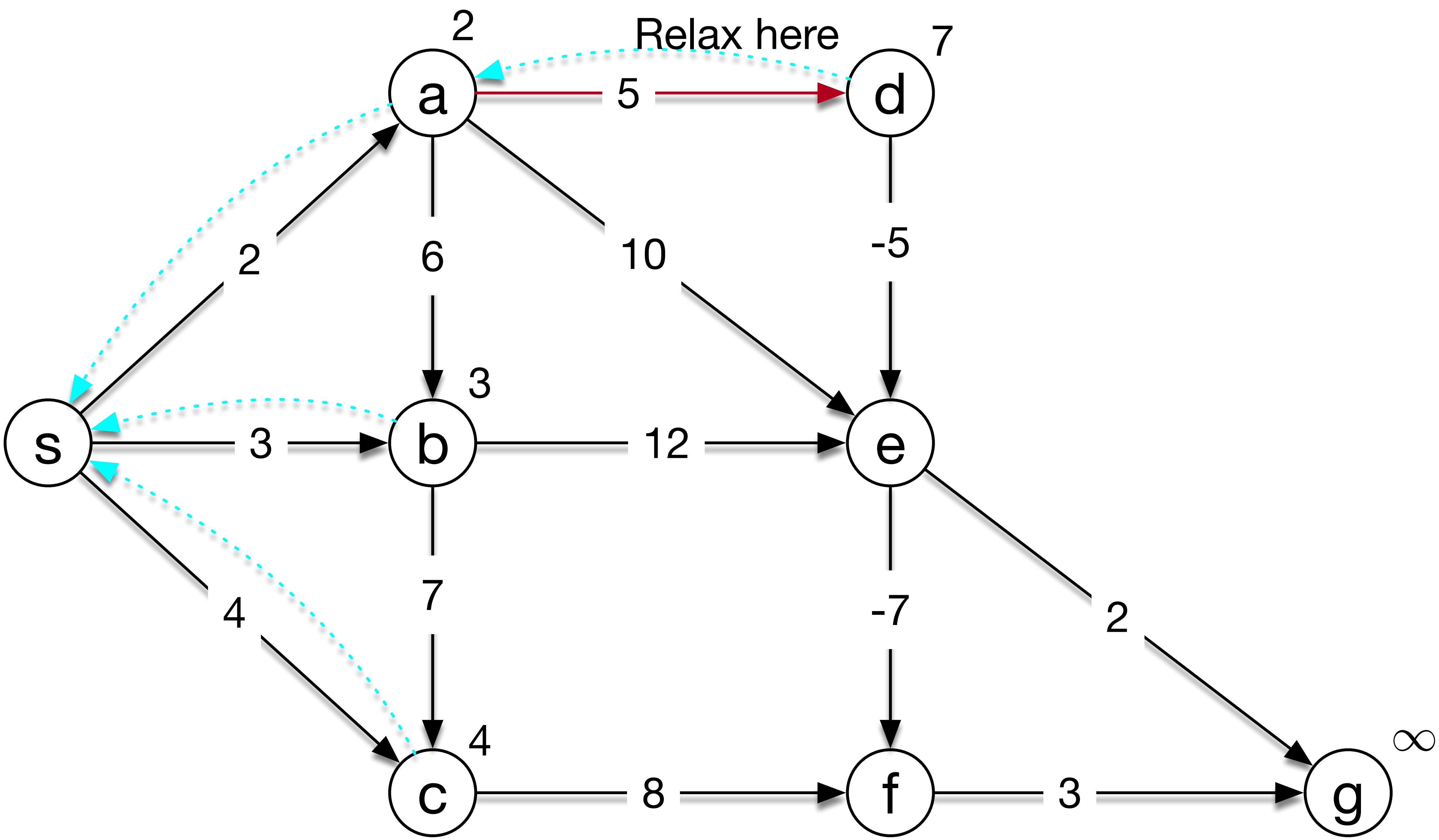
- def Bellman_Ford(s, V, E):
    - initialize(s,V,E)
    - for i in range(len(V)-1):
        - for (u,v) in E:
            - relax(u,v)
    - for (u,v) in E:
        - if relax(u,v) changes the distance in v:
            - return 'negative weight cycle detected'
    - return 'done'

Bellman-Ford

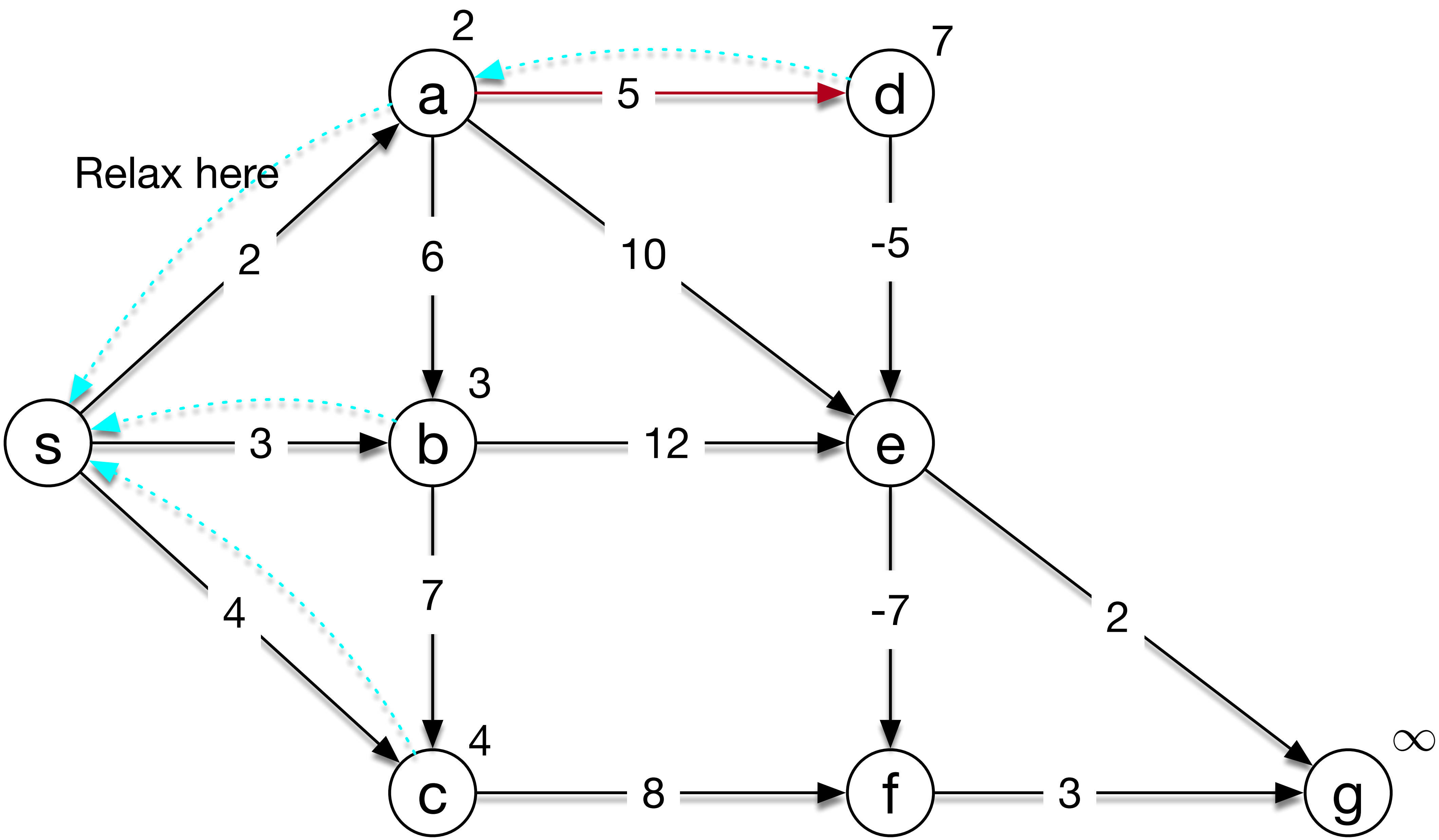
- Example: After initialization



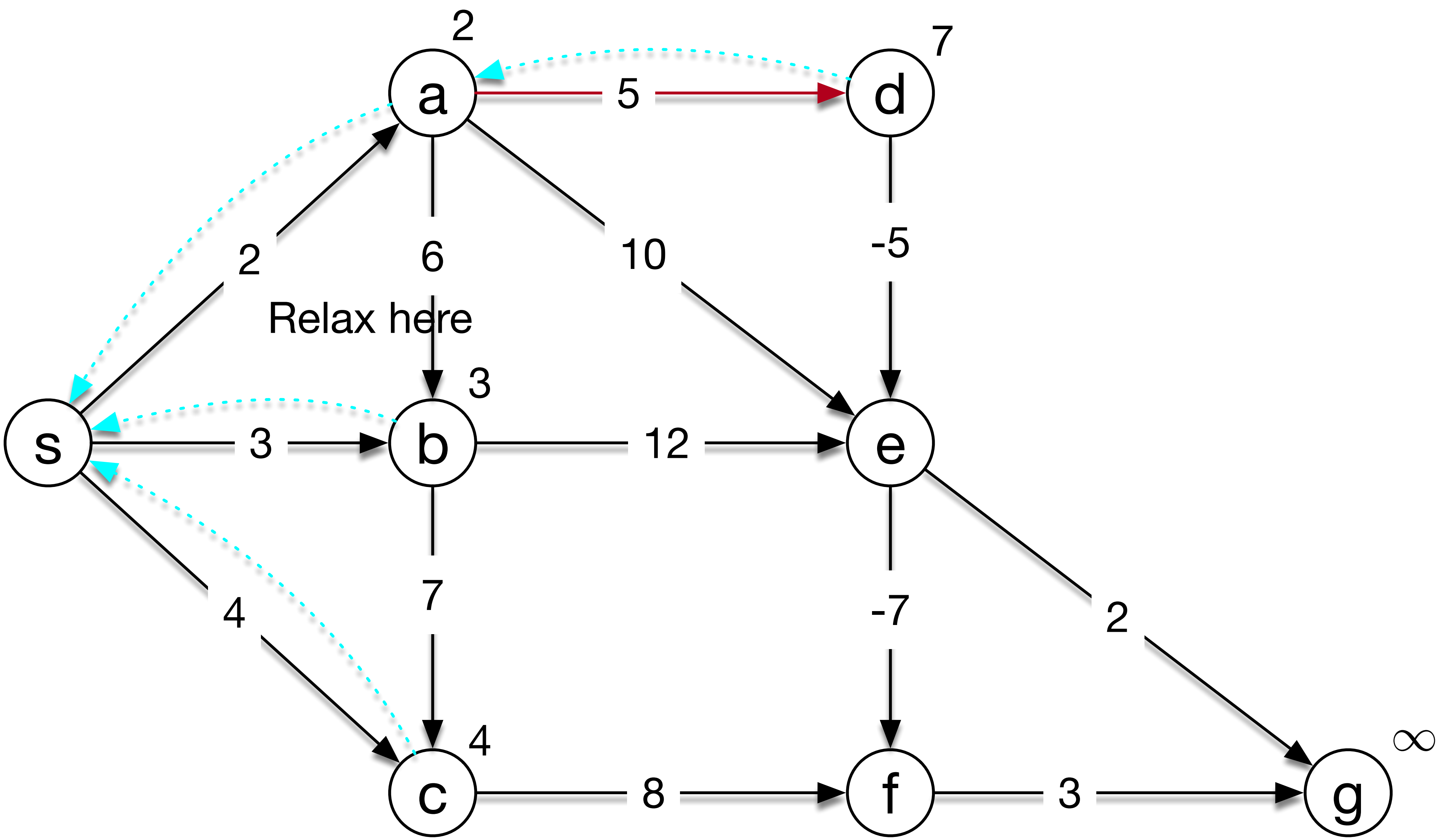
Bellman-Ford



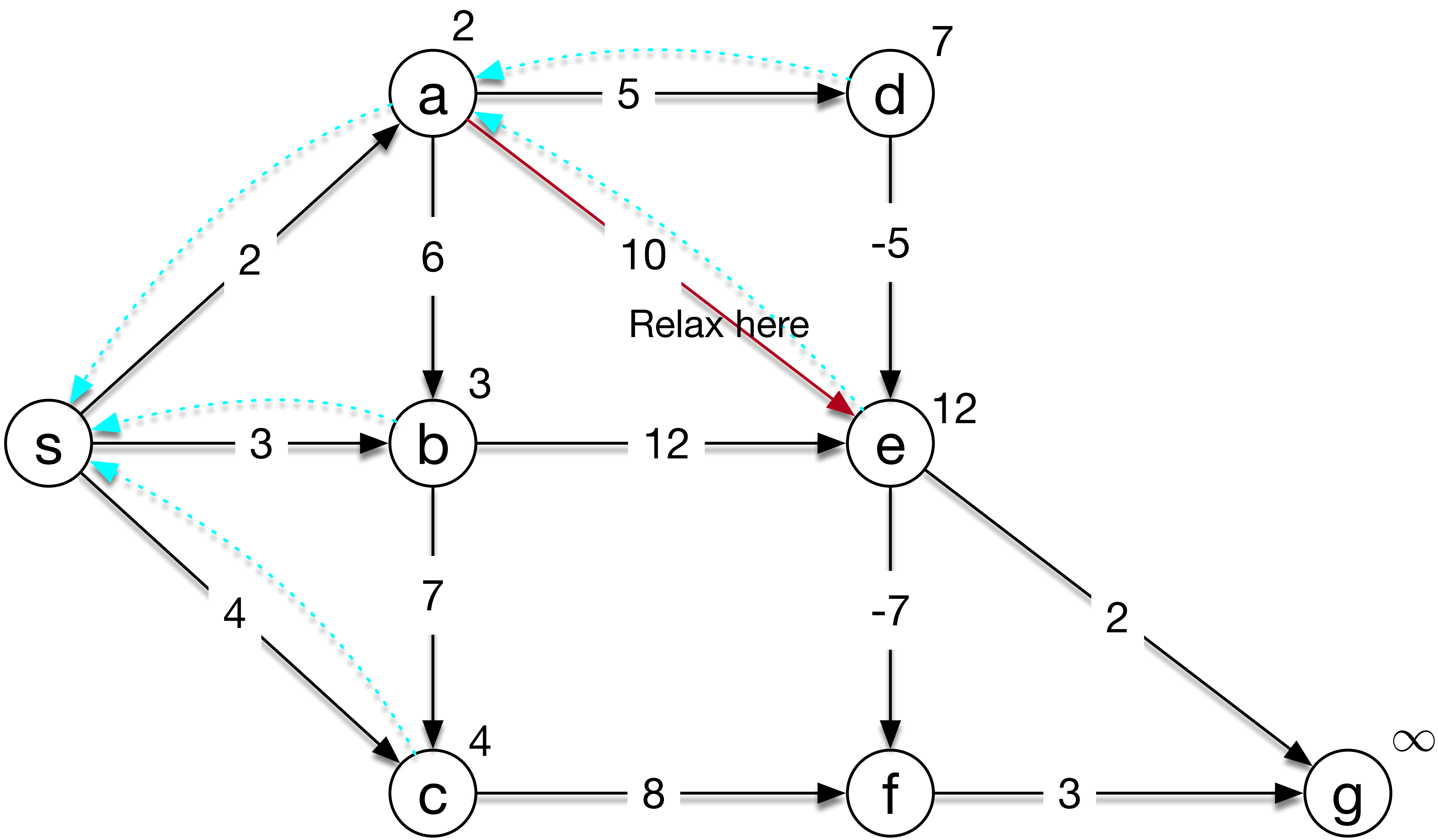
Bellman-Ford



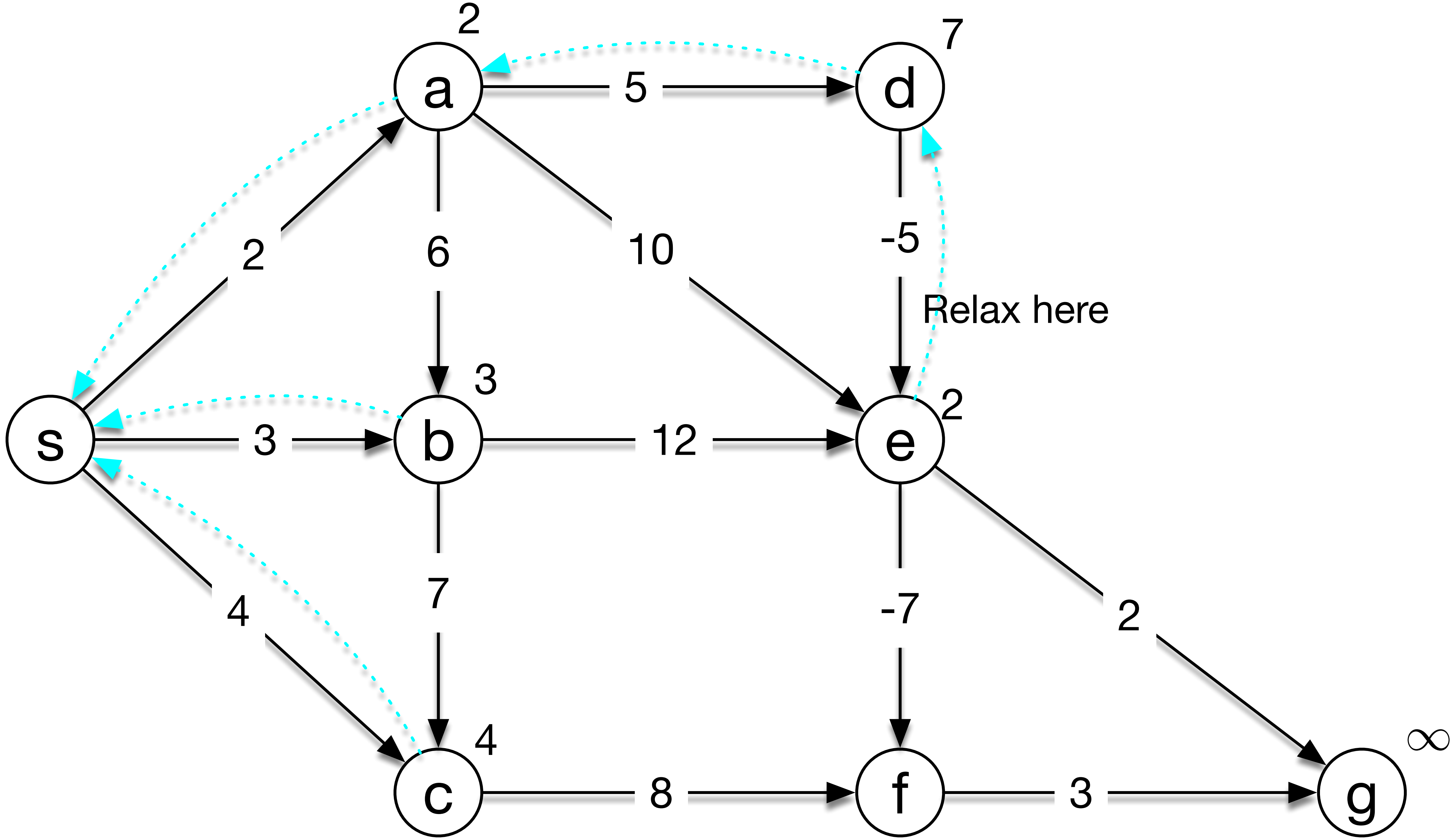
Bellman-Ford



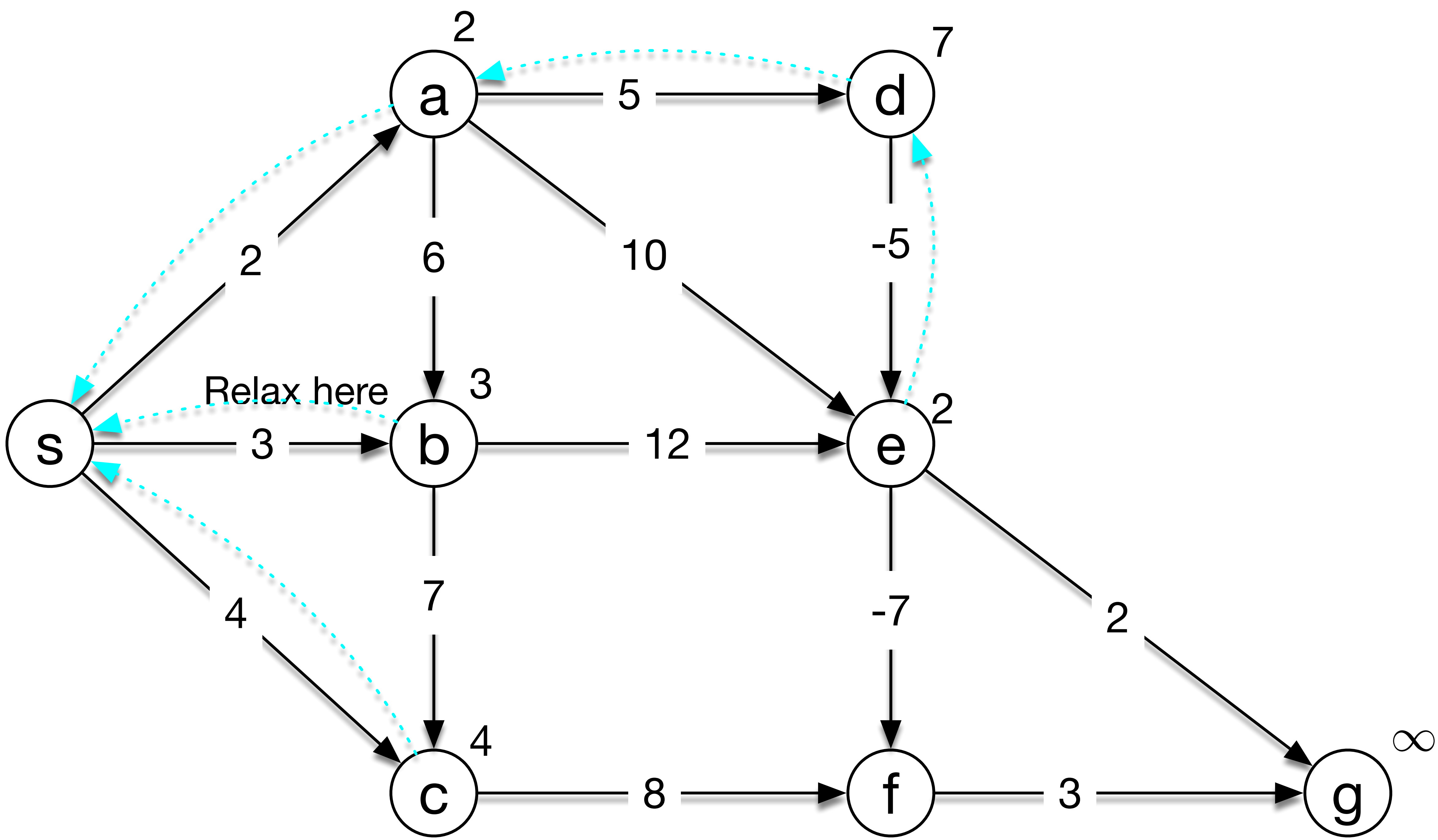
Bellman-Ford



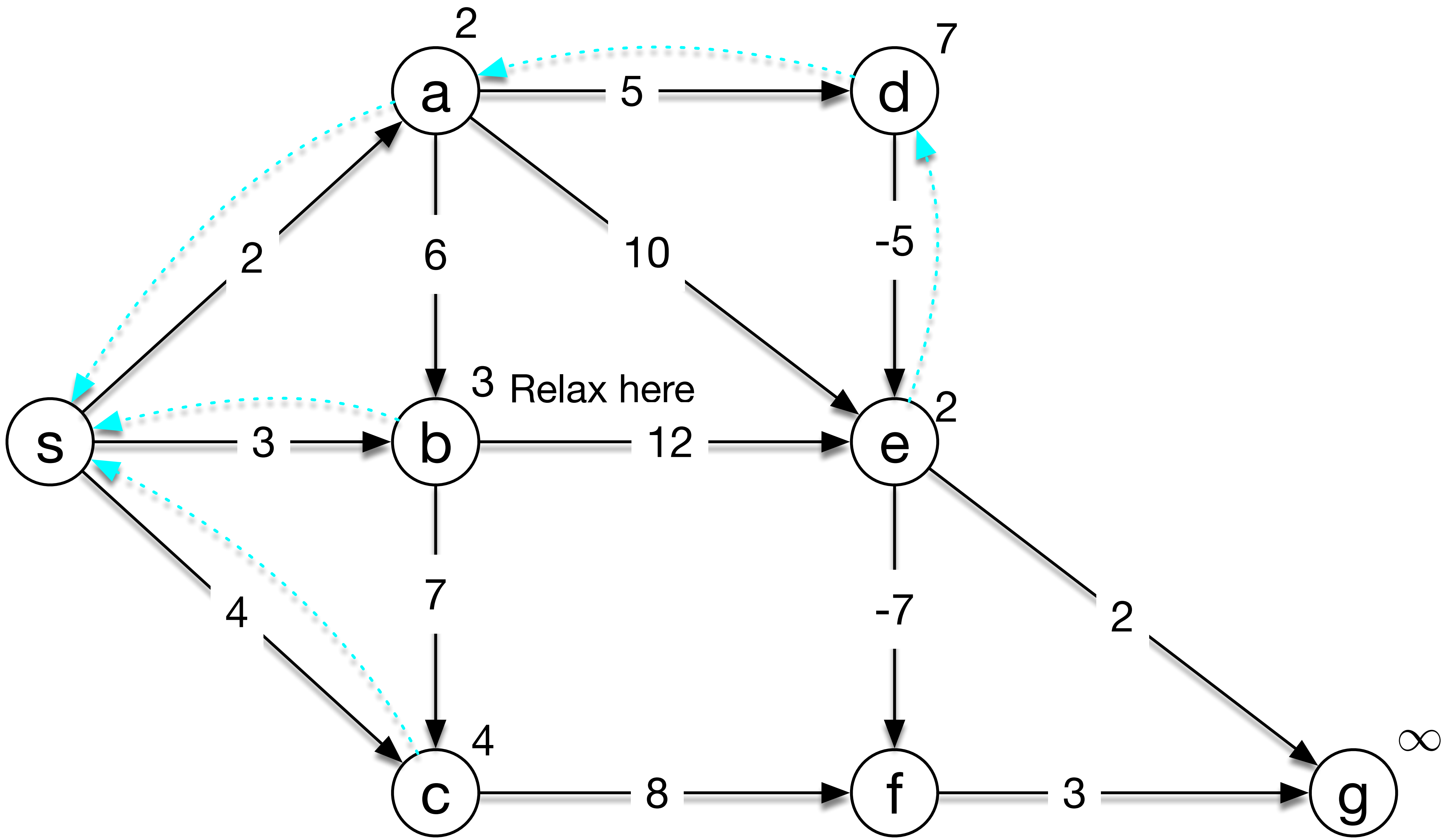
Bellman-Ford



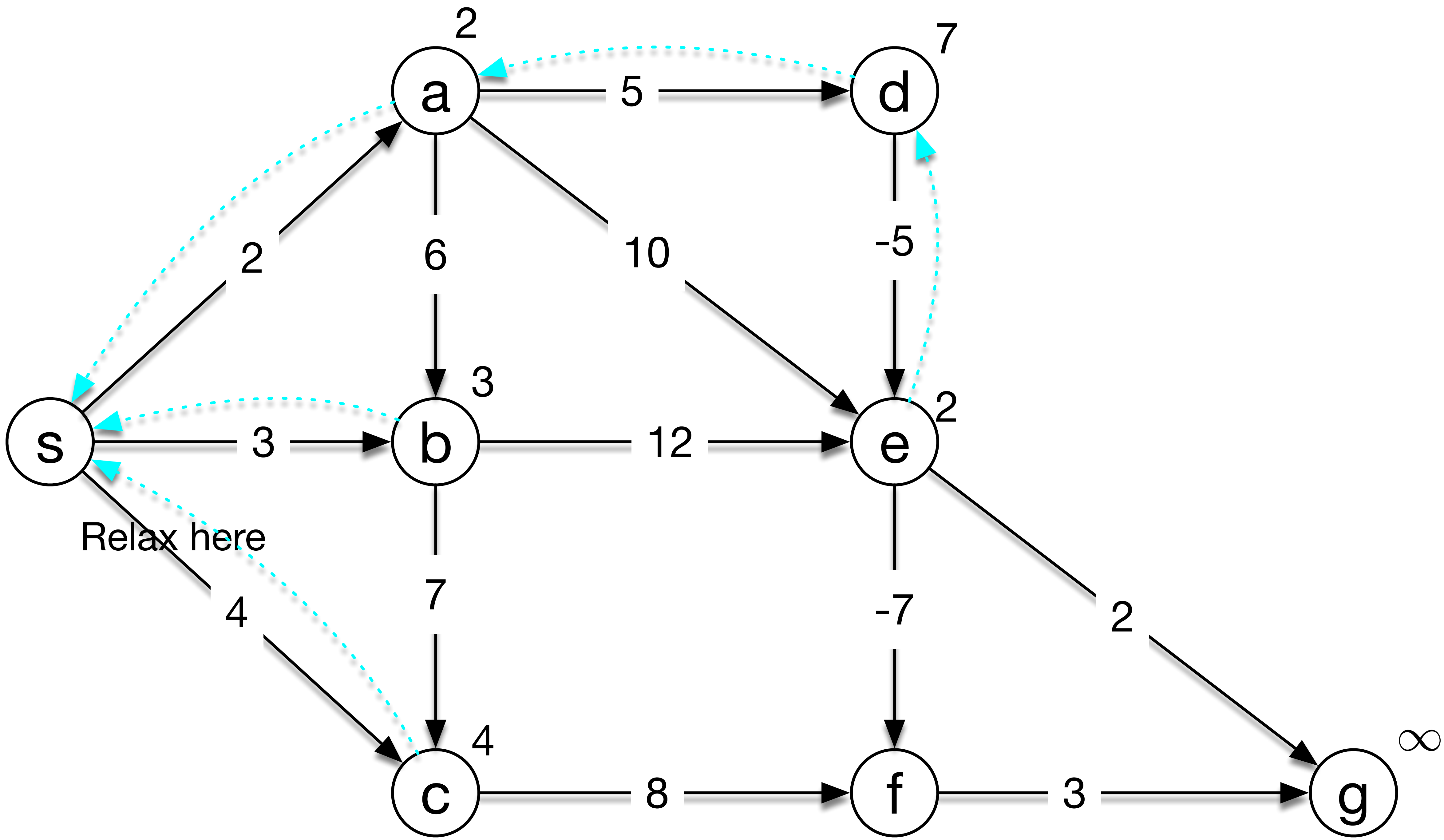
Bellman-Ford



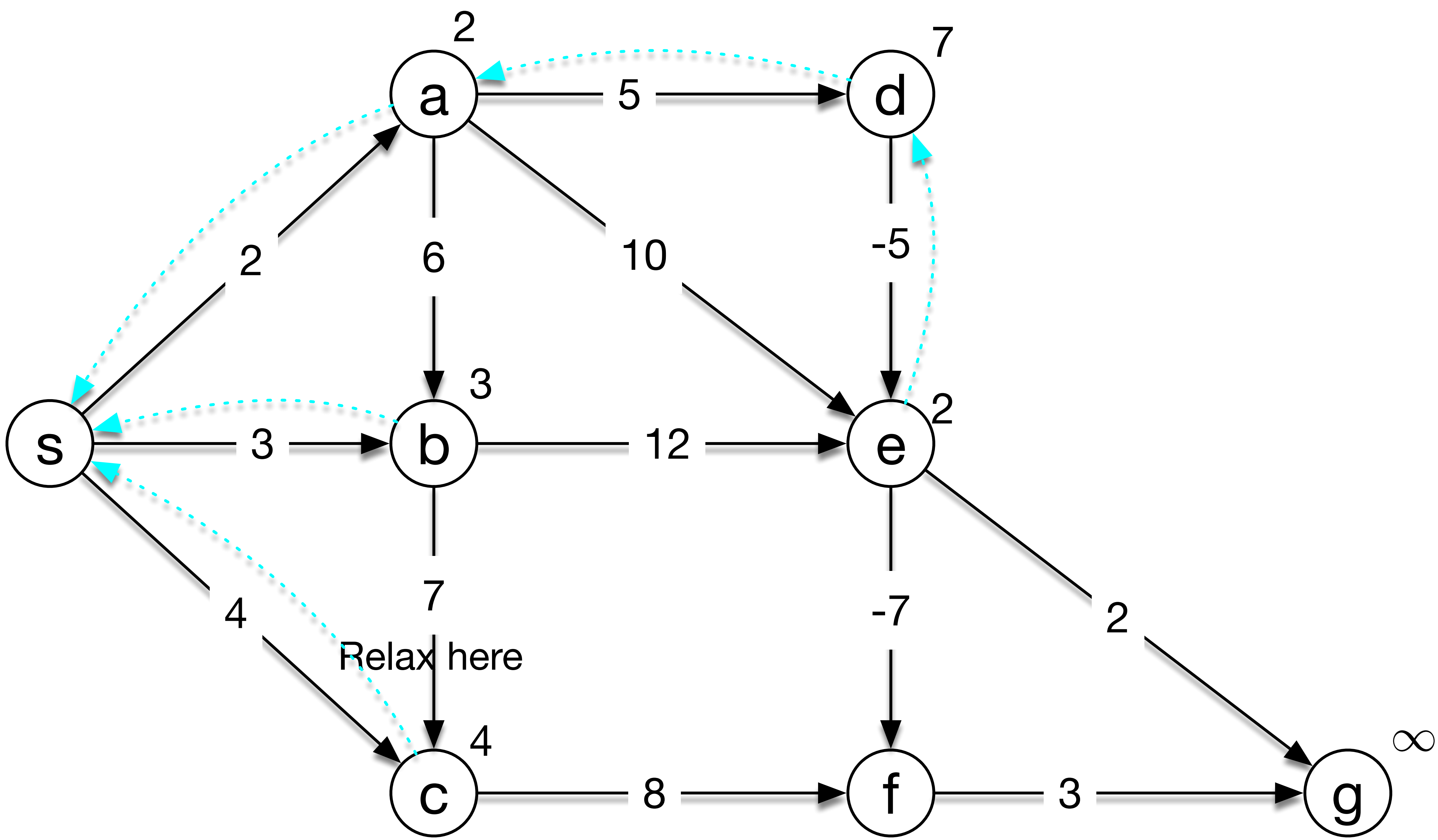
Bellman-Ford



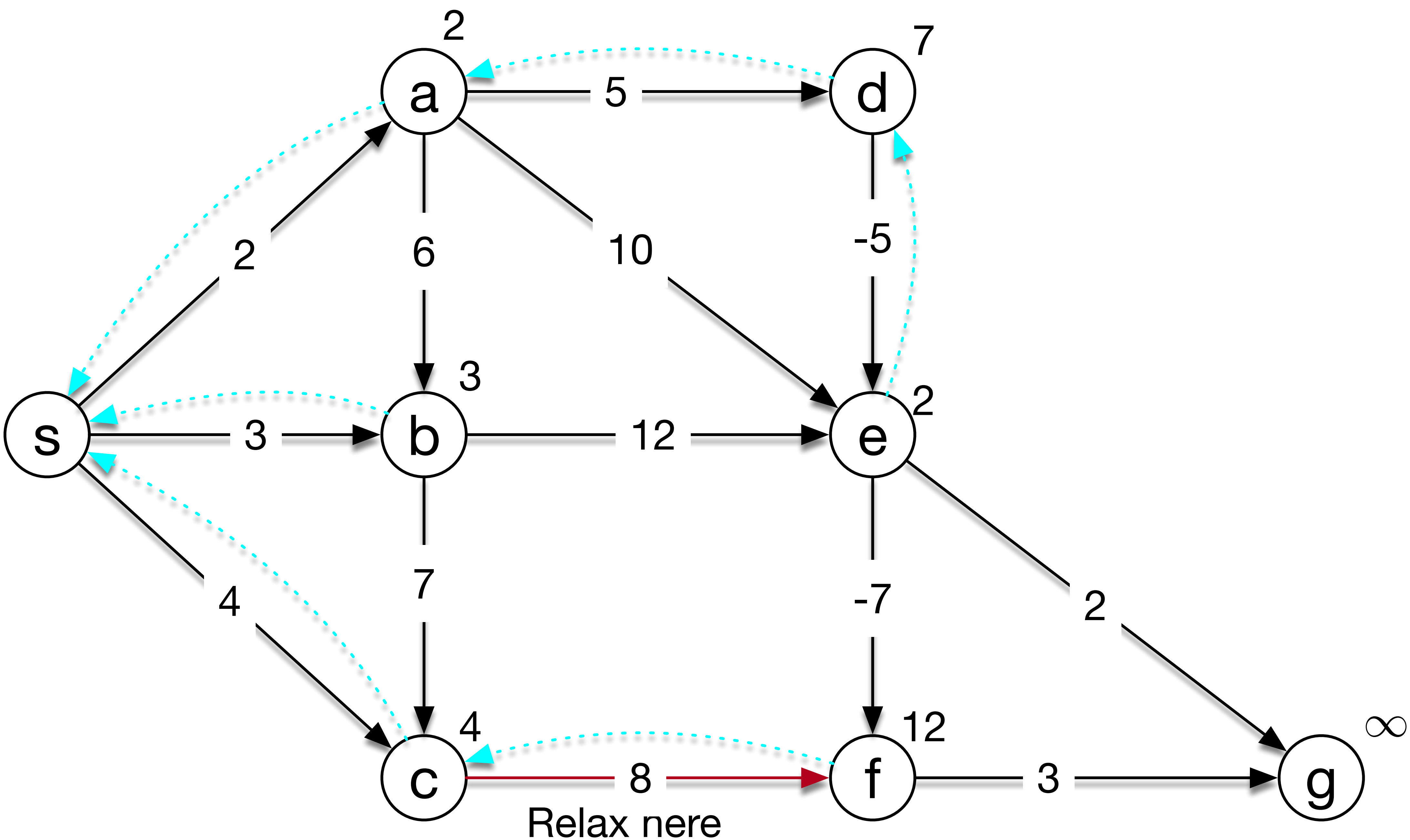
Bellman-Ford



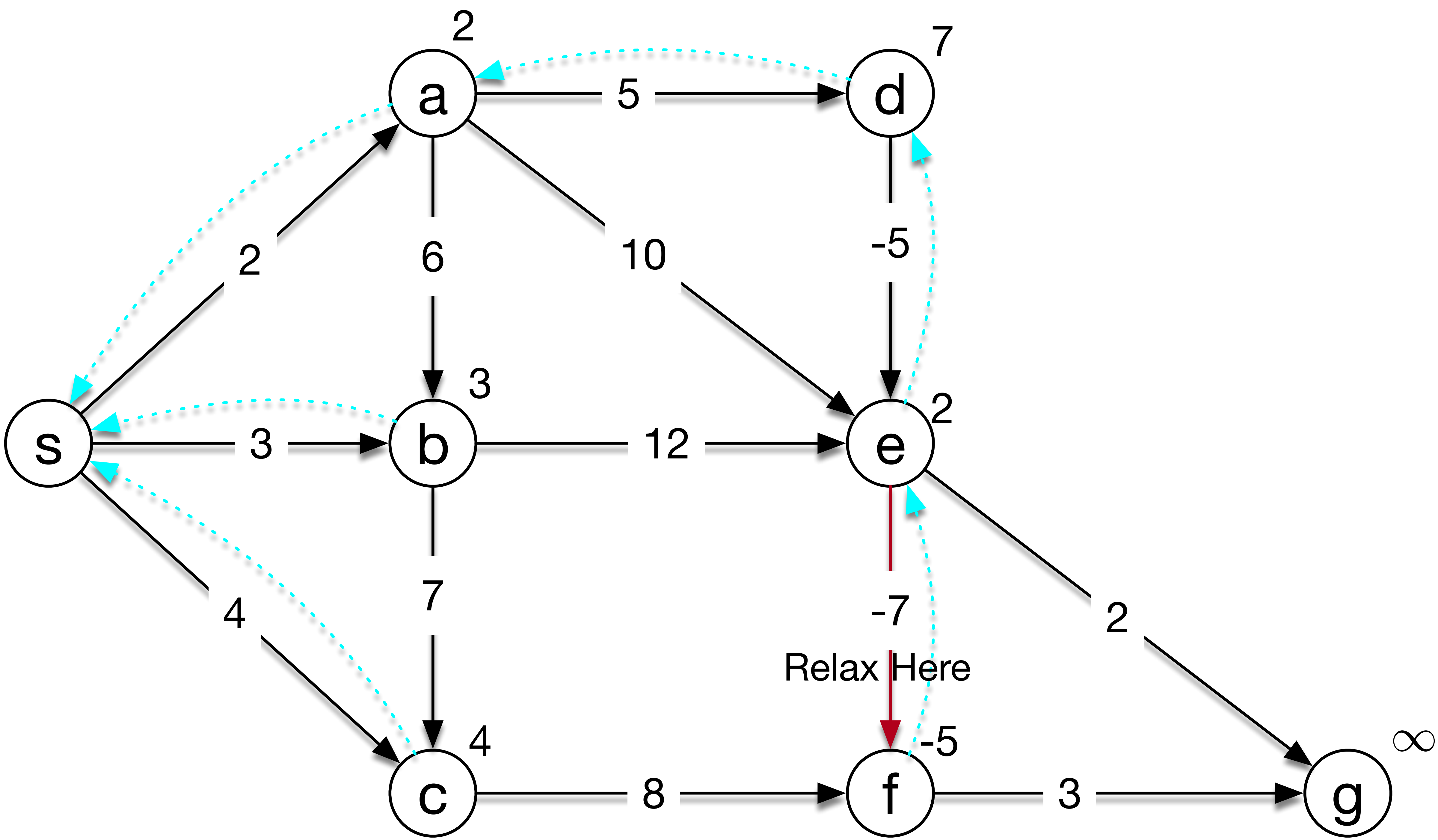
Bellman-Ford



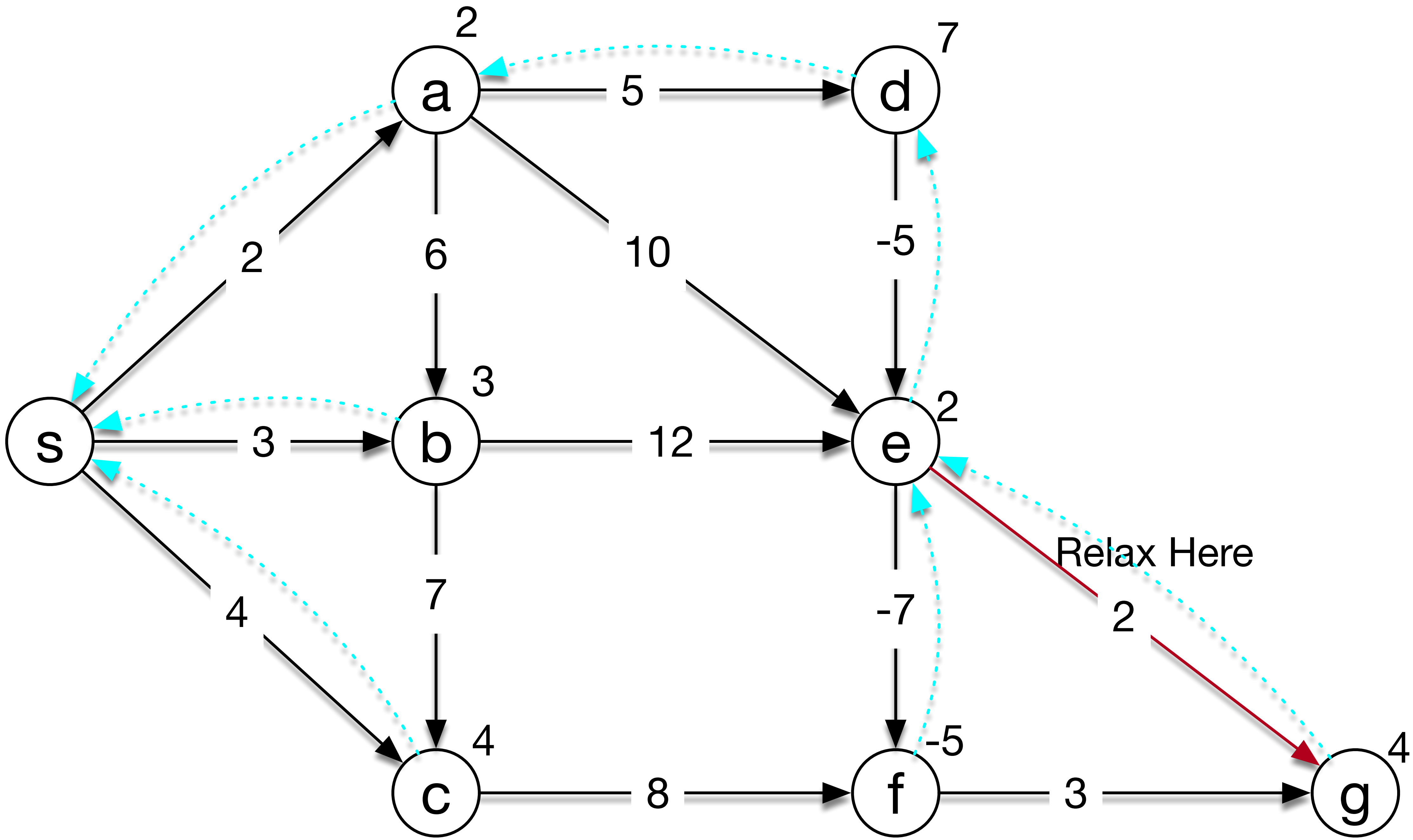
Bellman-Ford



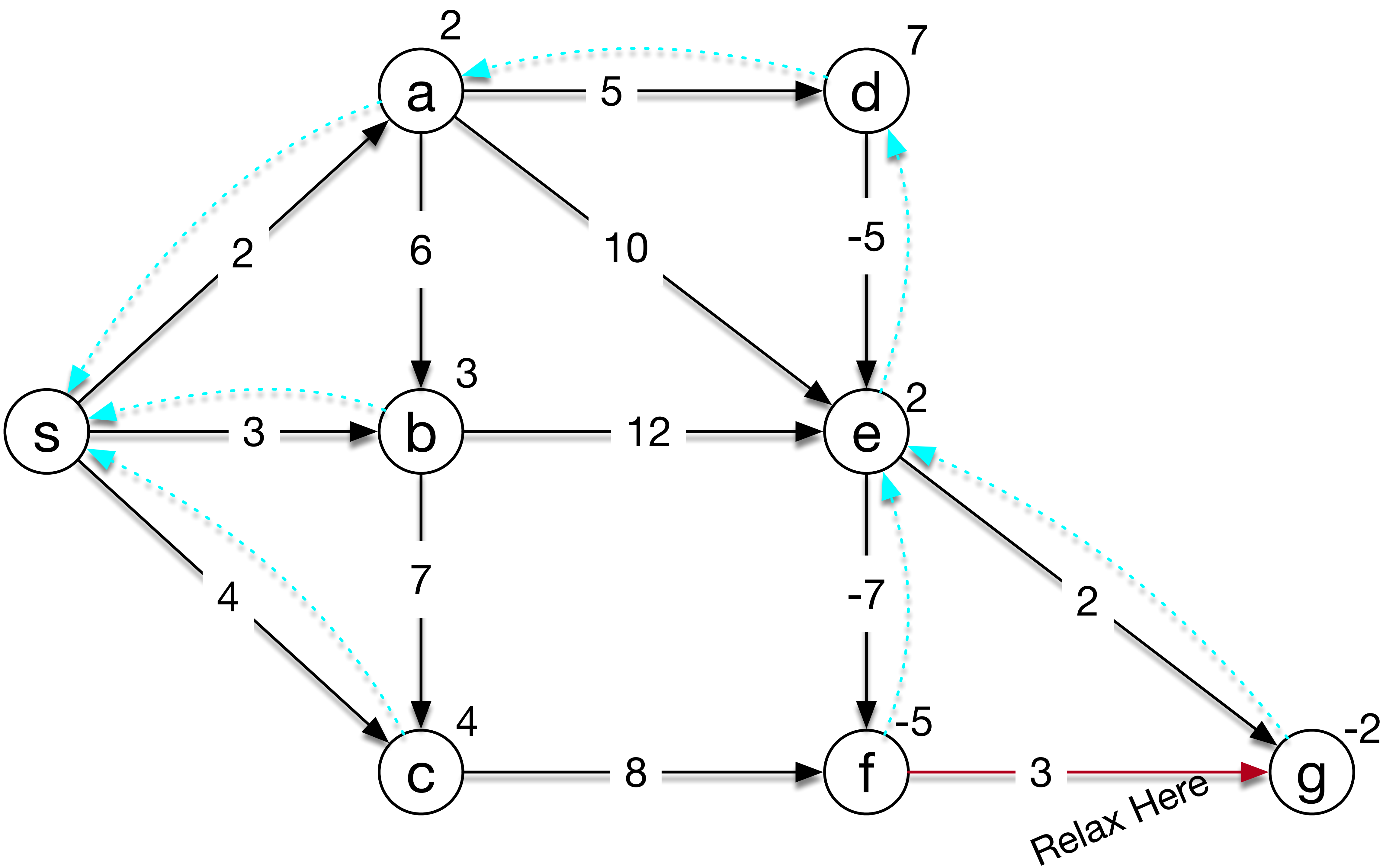
Bellman-Ford



Bellman-Ford



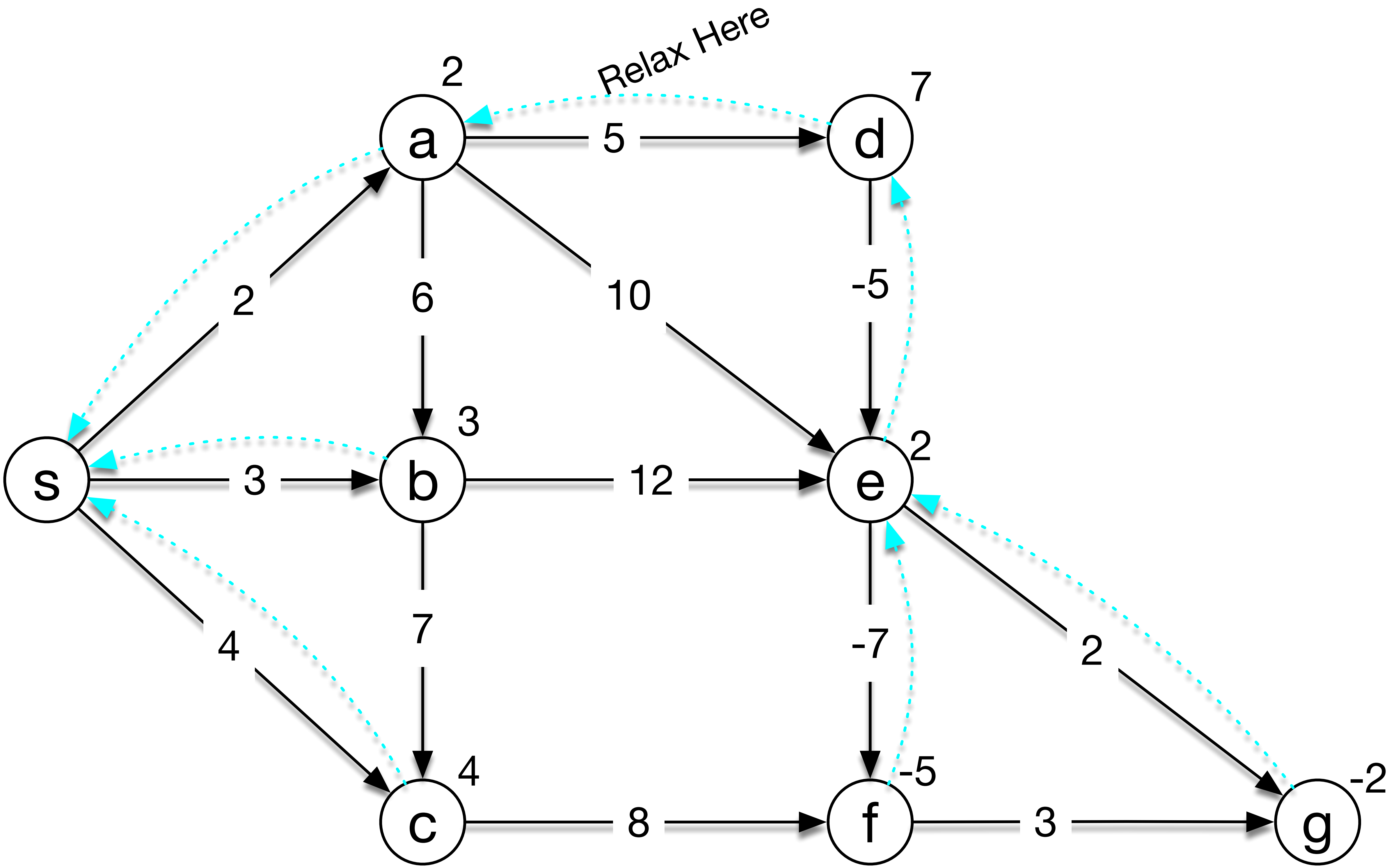
Bellman-Ford



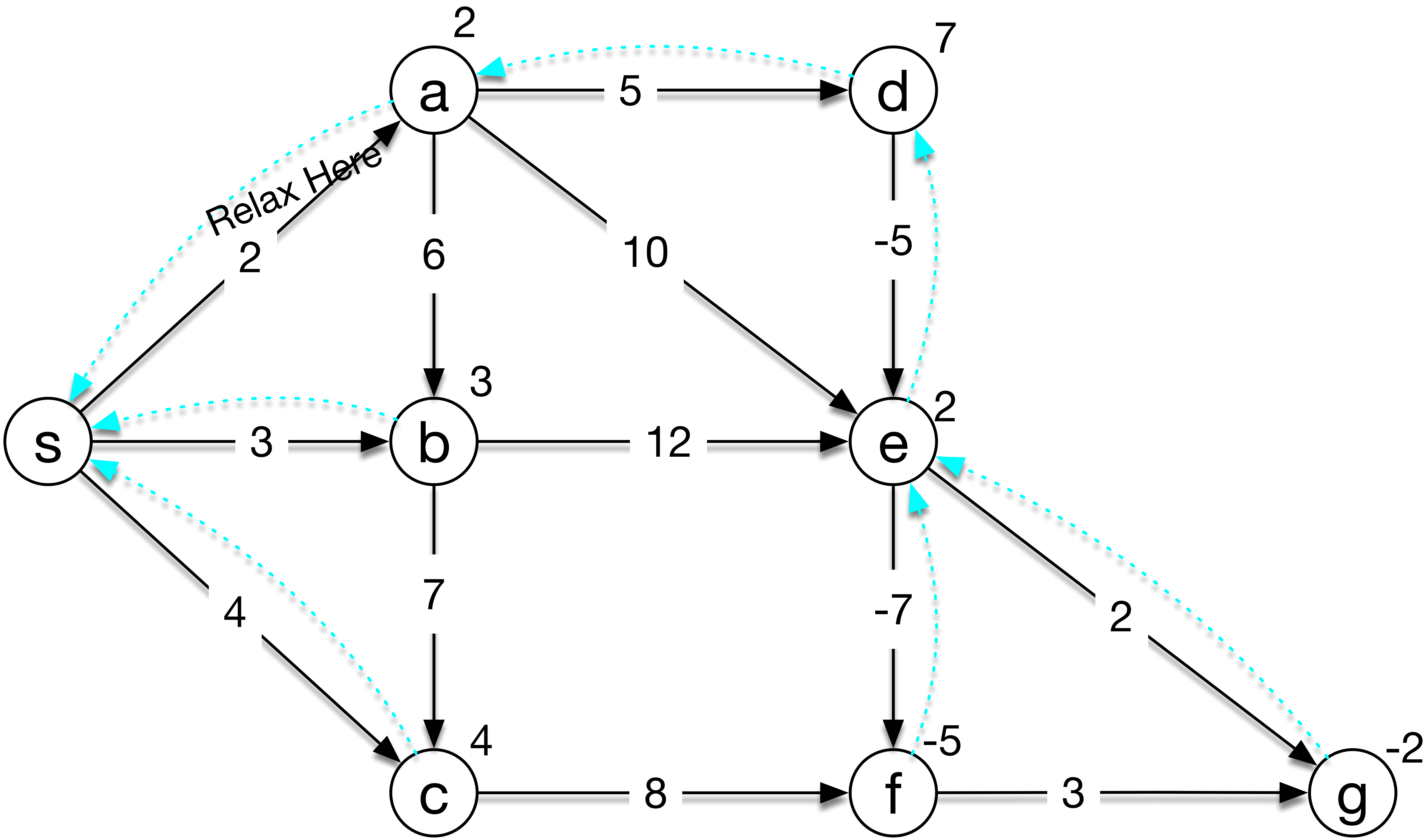
Bellman-Ford

- Now, we can start again

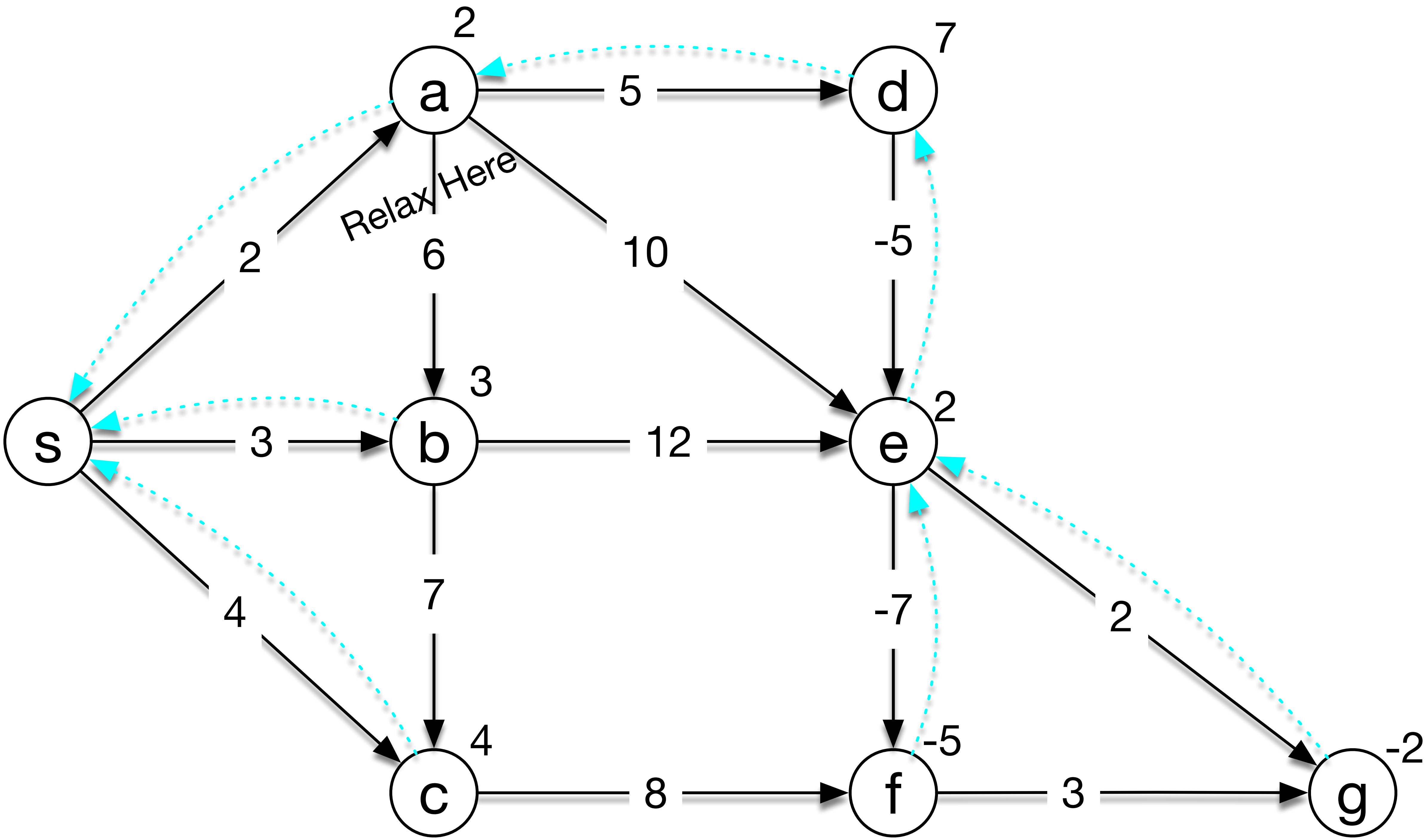
Bellman-Ford



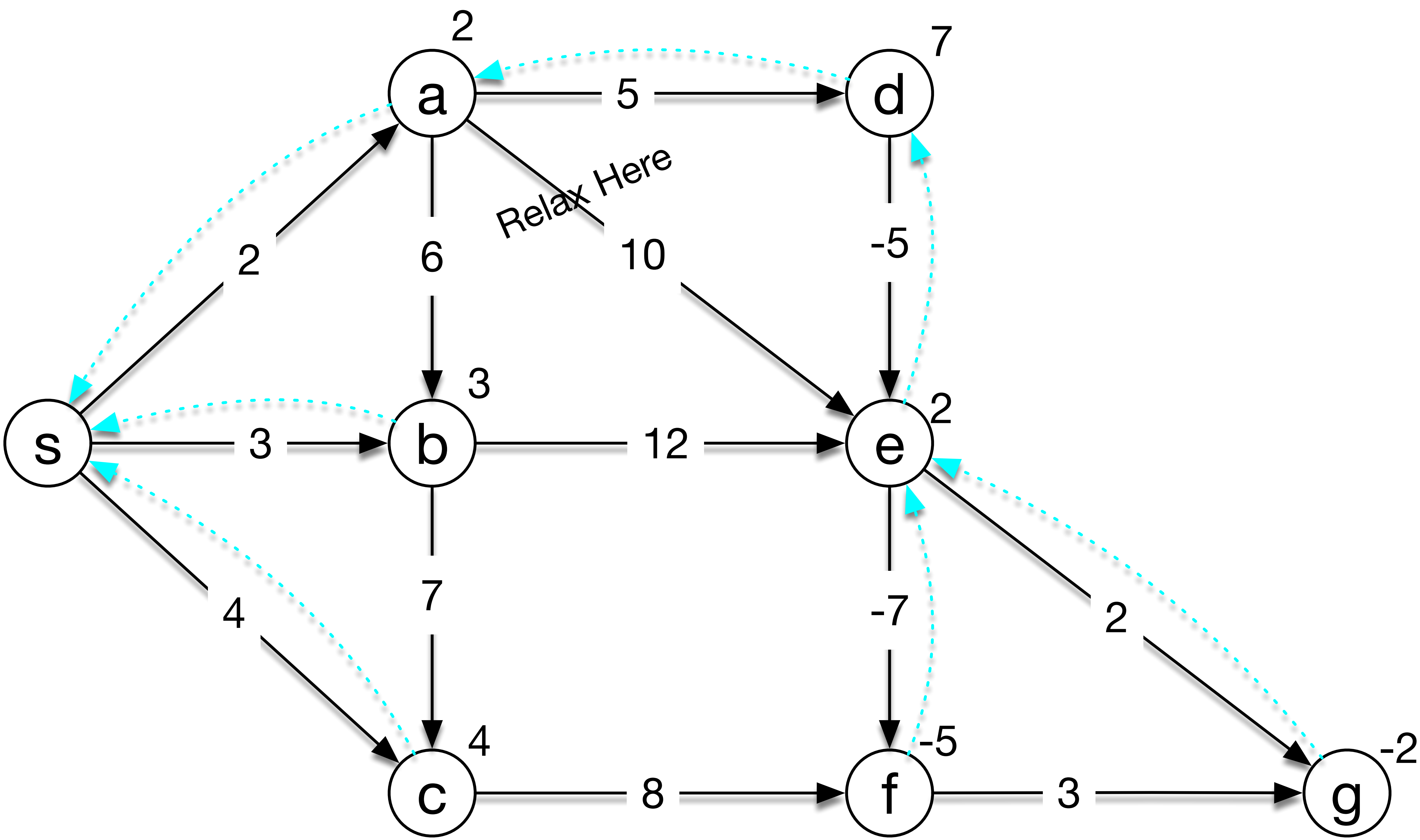
Bellman-Ford



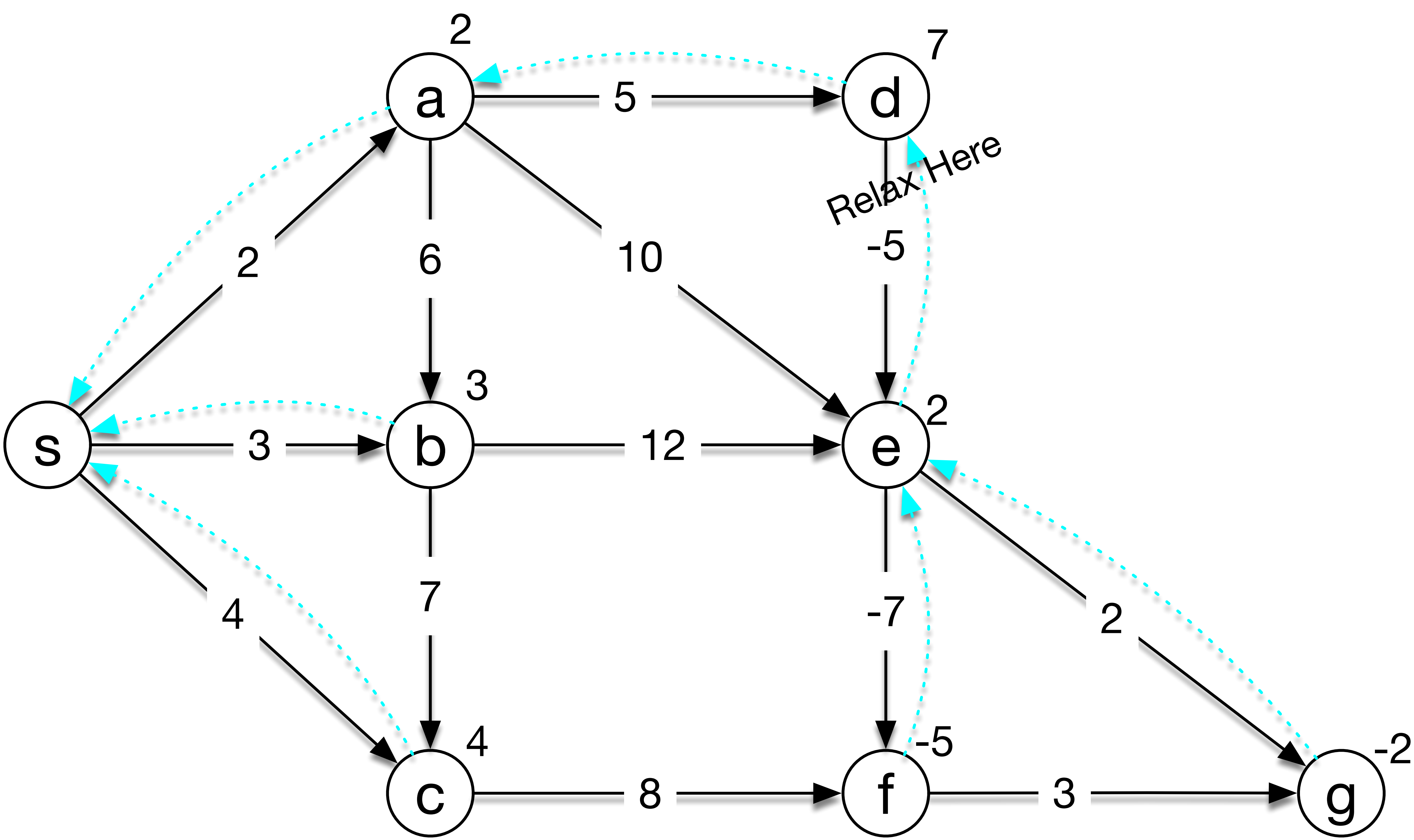
Bellman-Ford



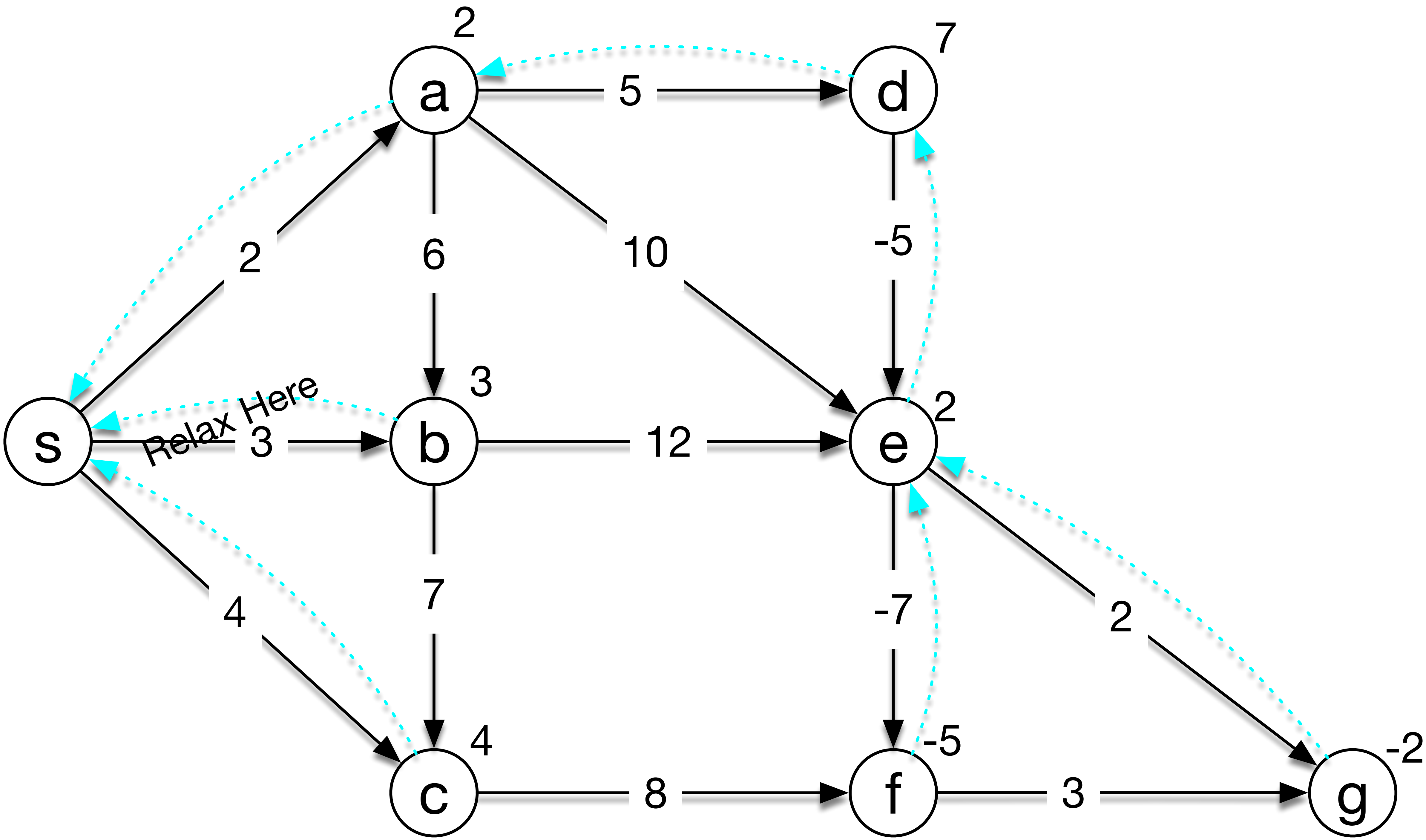
Bellman-Ford



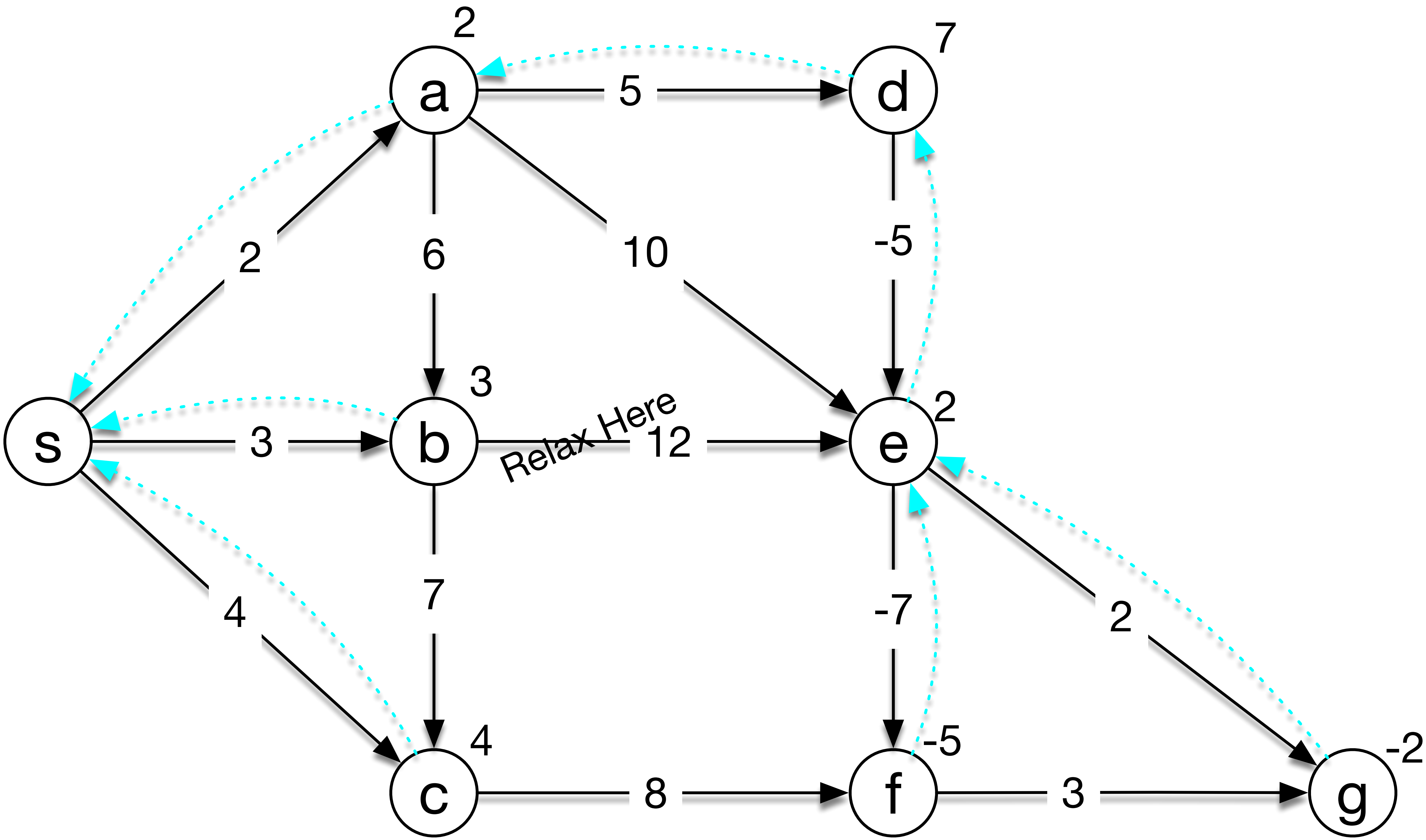
Bellman-Ford



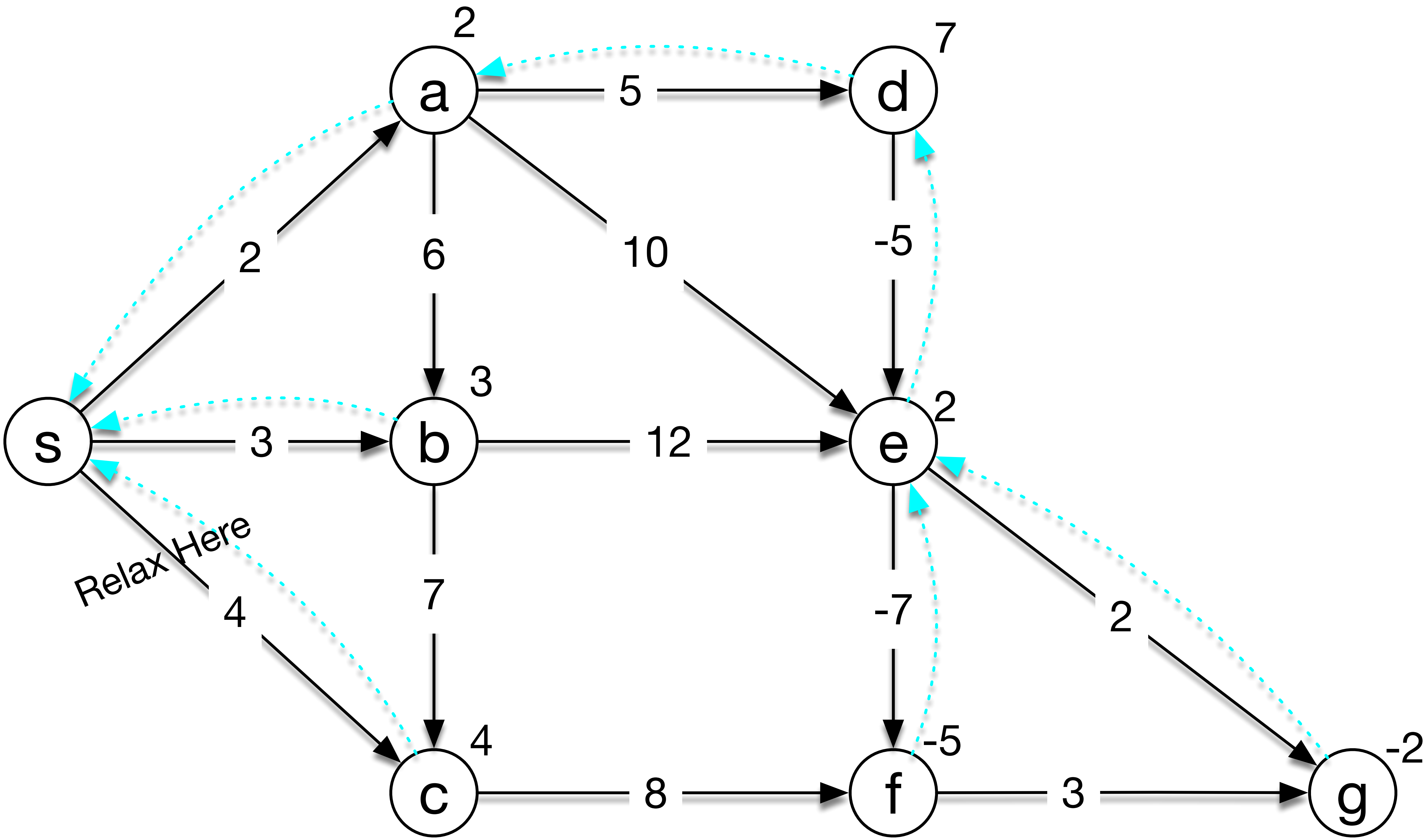
Bellman-Ford



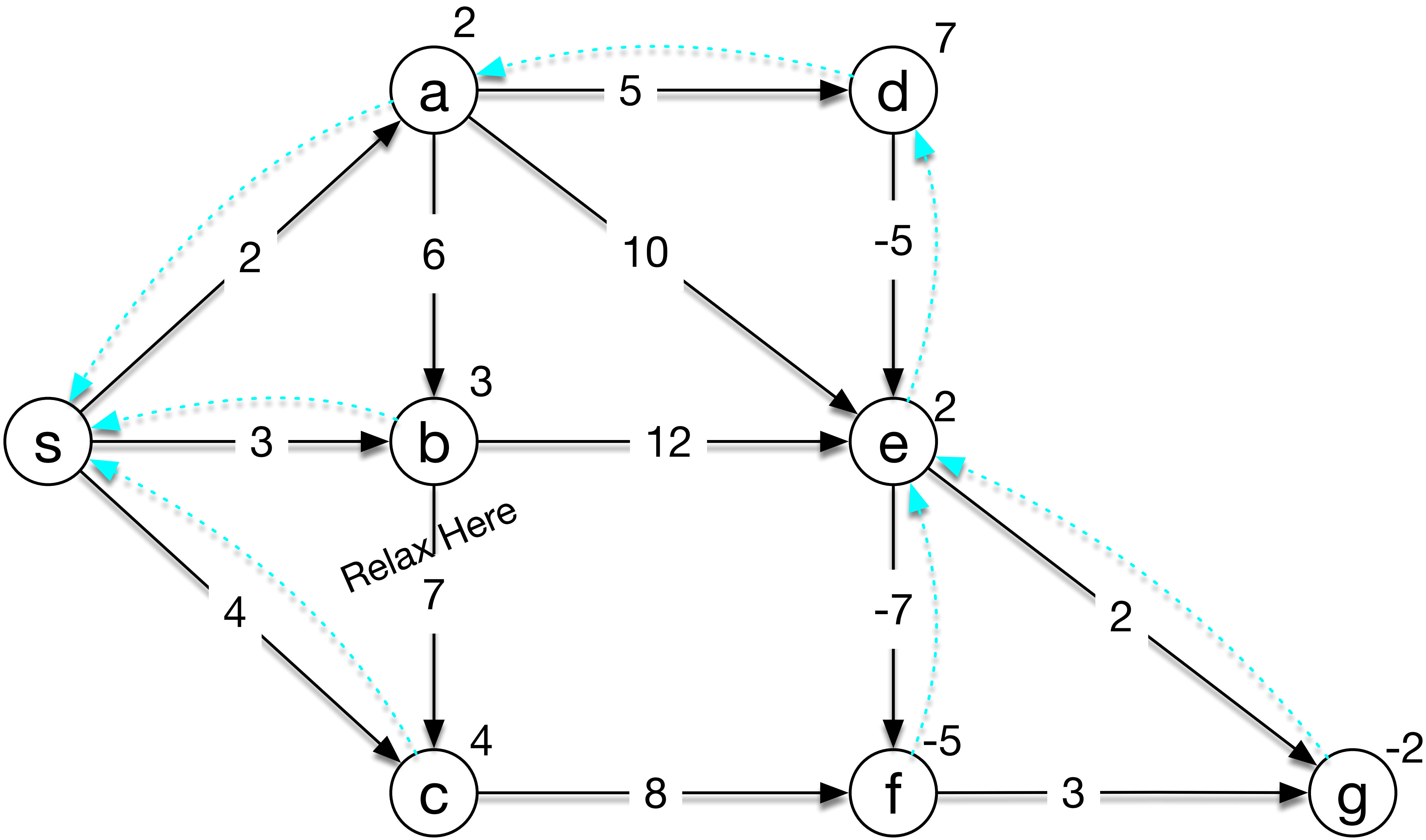
Bellman-Ford



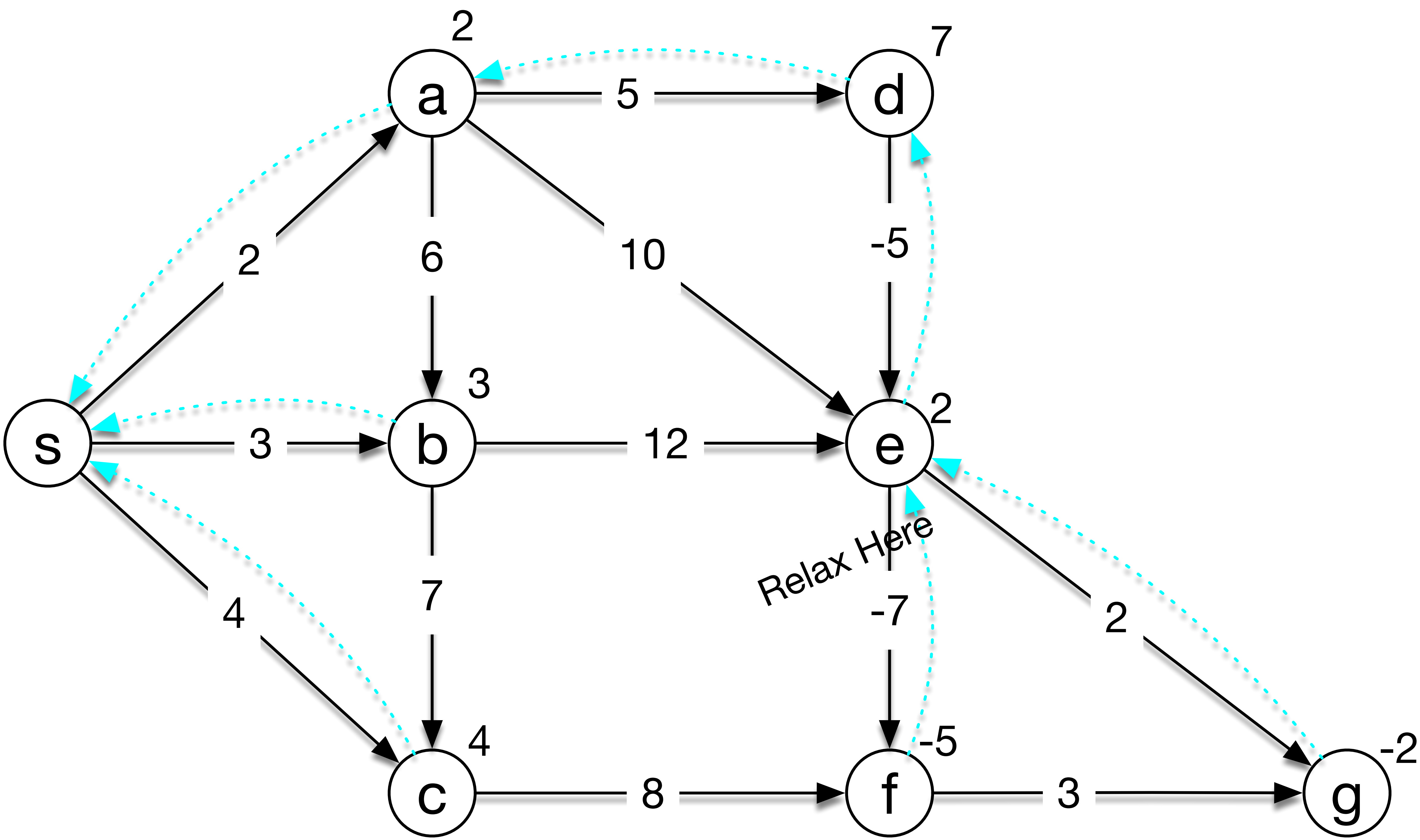
Bellman-Ford



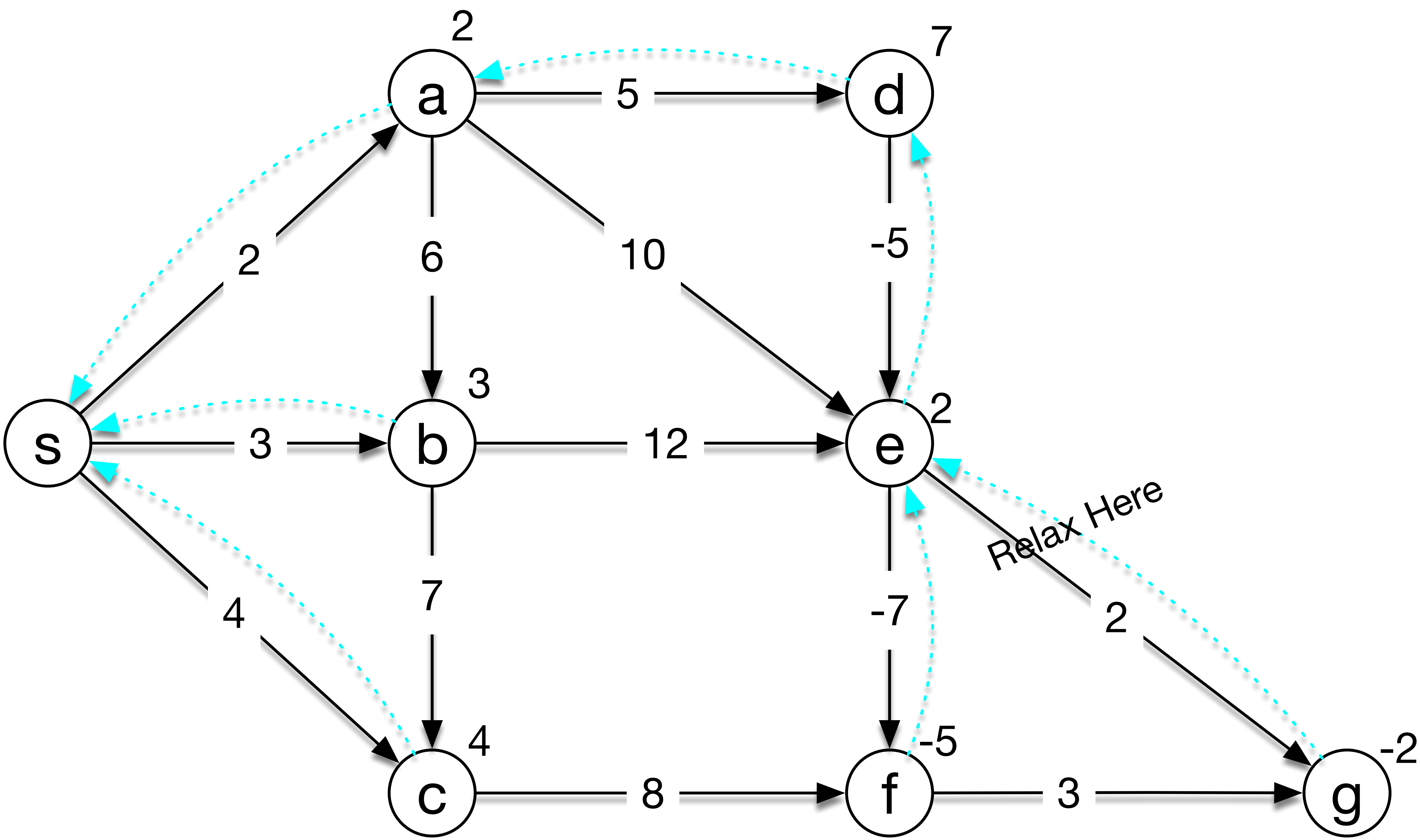
Bellman-Ford



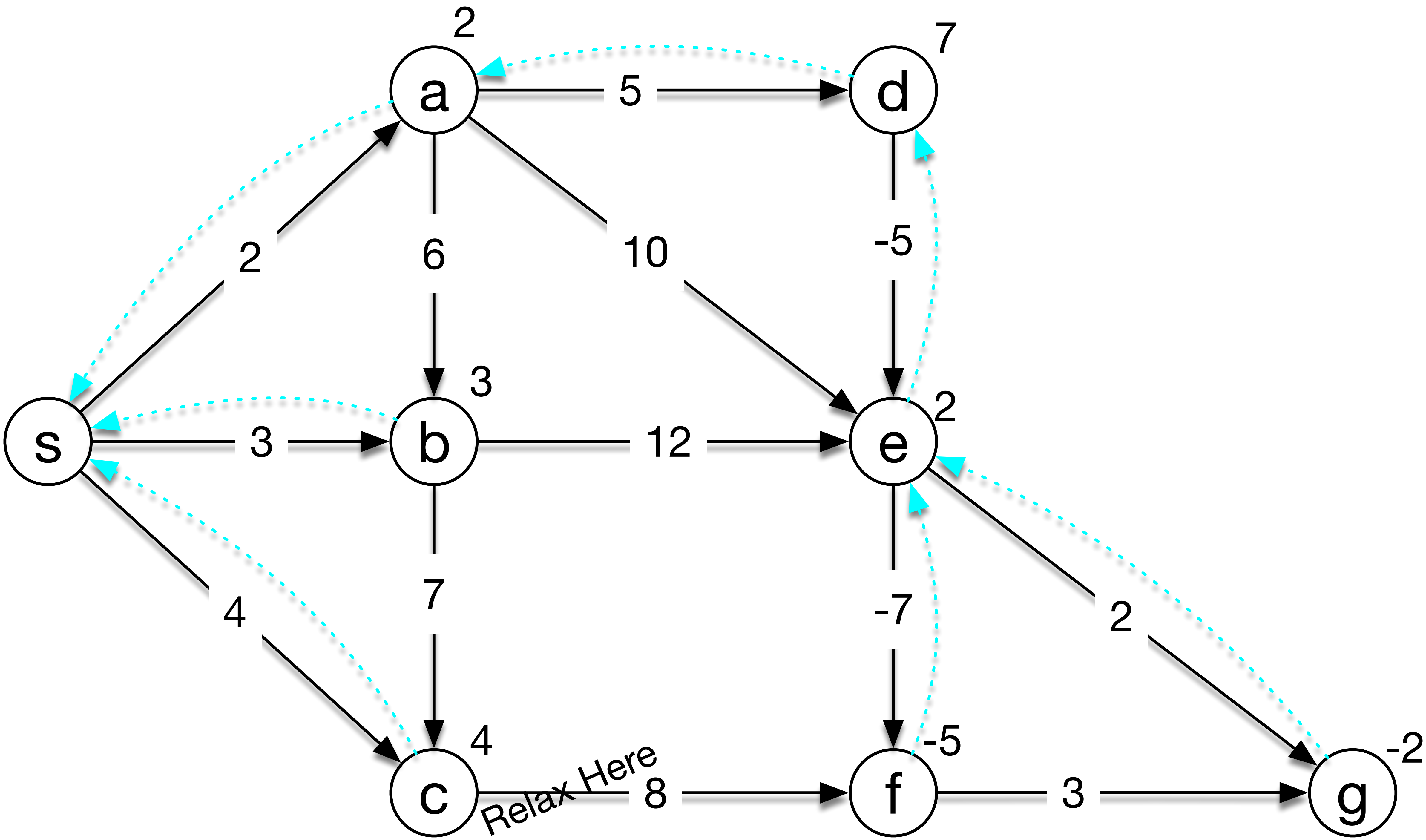
Bellman-Ford



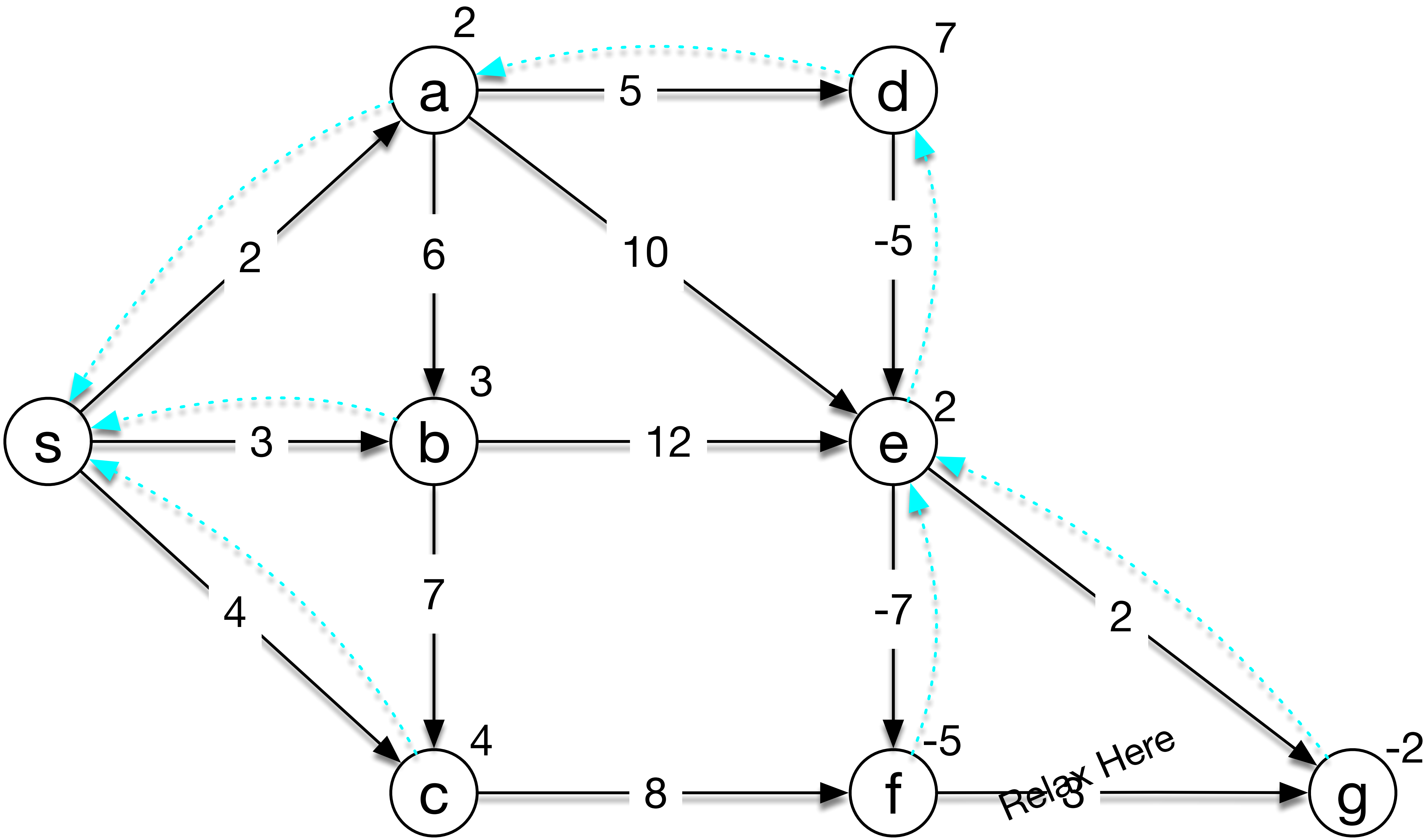
Bellman-Ford



Bellman-Ford



Bellman-Ford

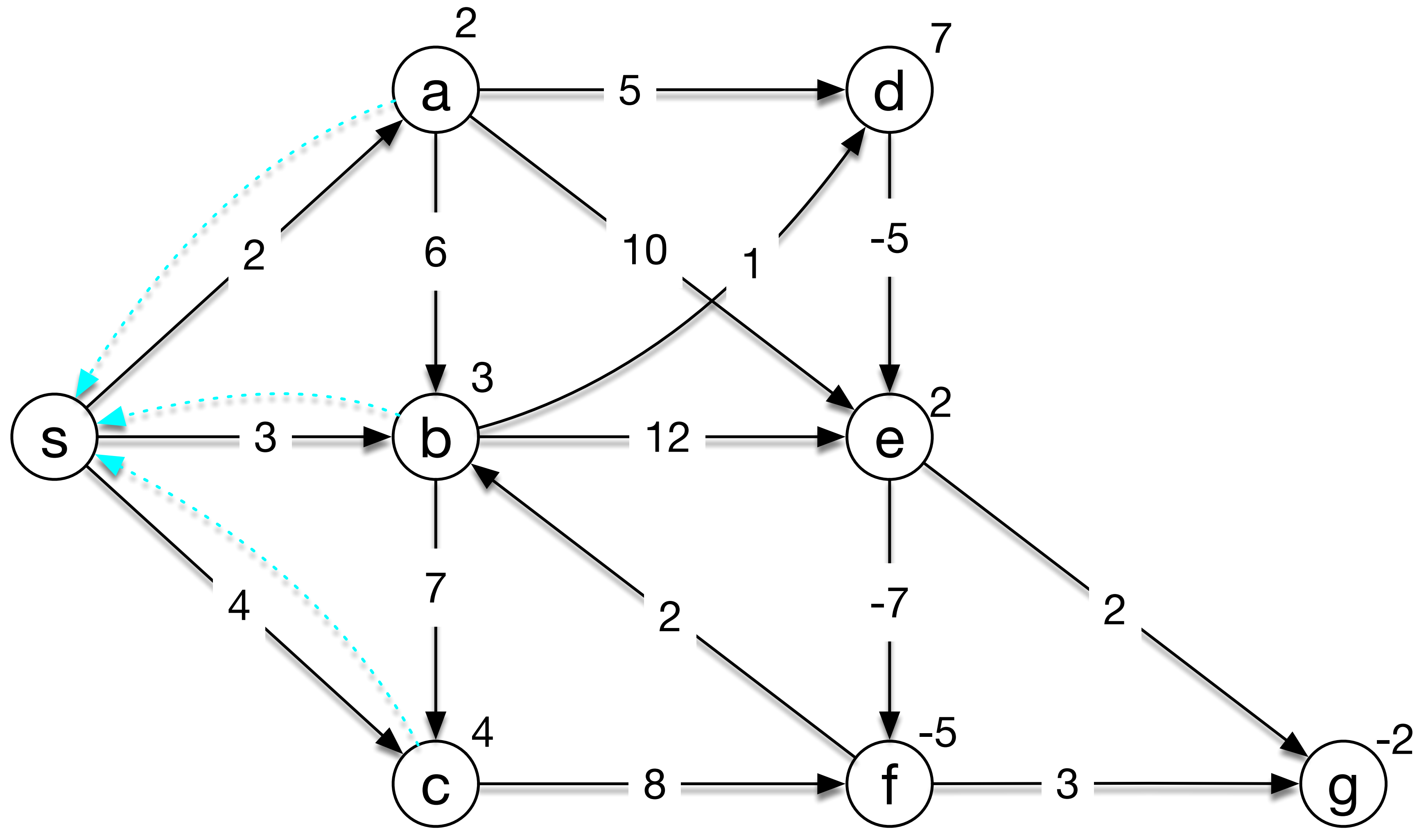


Bellman-Ford

- The second round, nothing changed
  - This was an easy example, after all
- We can therefore shortcut the rest, decide that there are no cycles, and finish

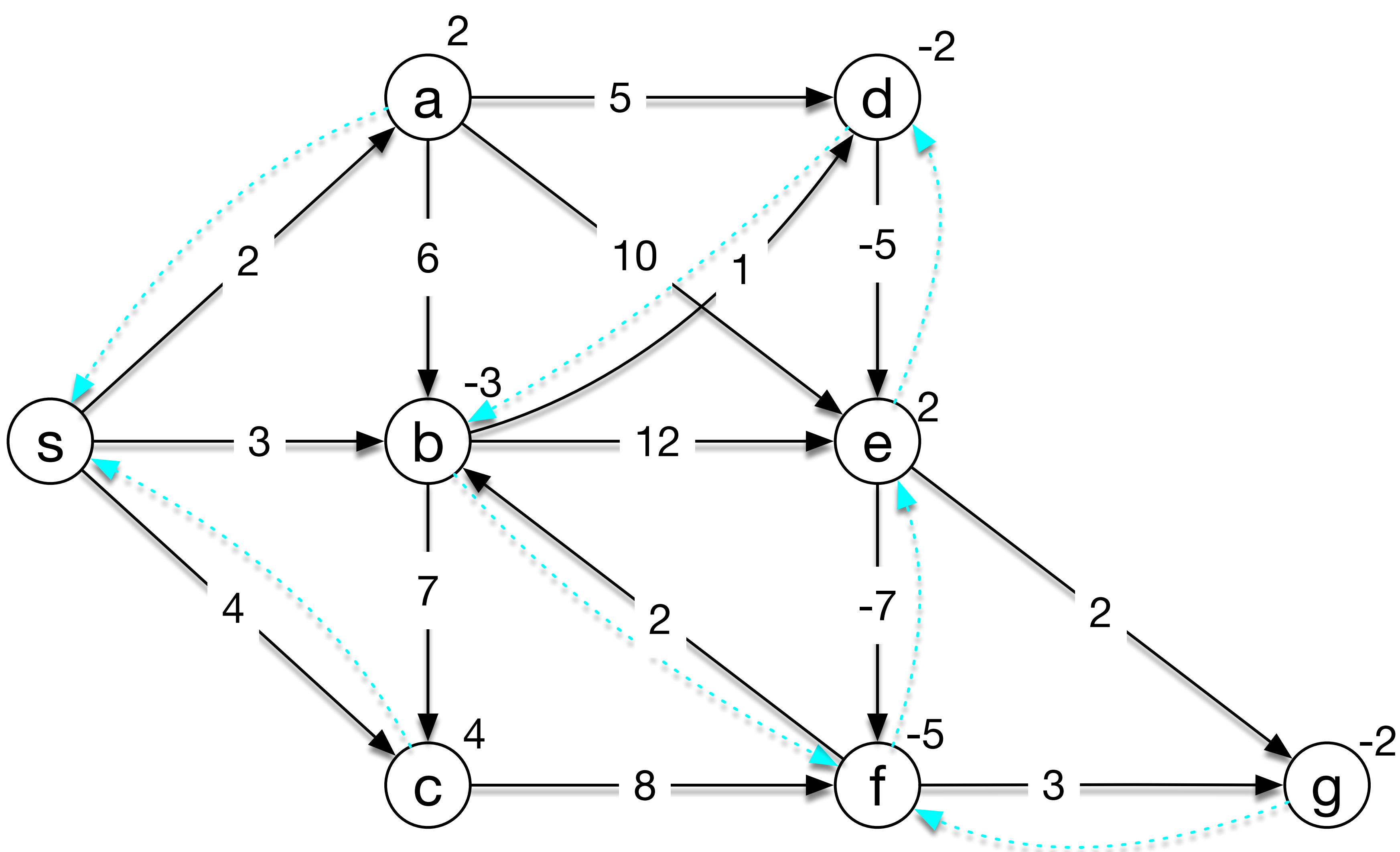
Bellman-Ford

- Now, let's create a negative weight cycle b-d-e-f-b



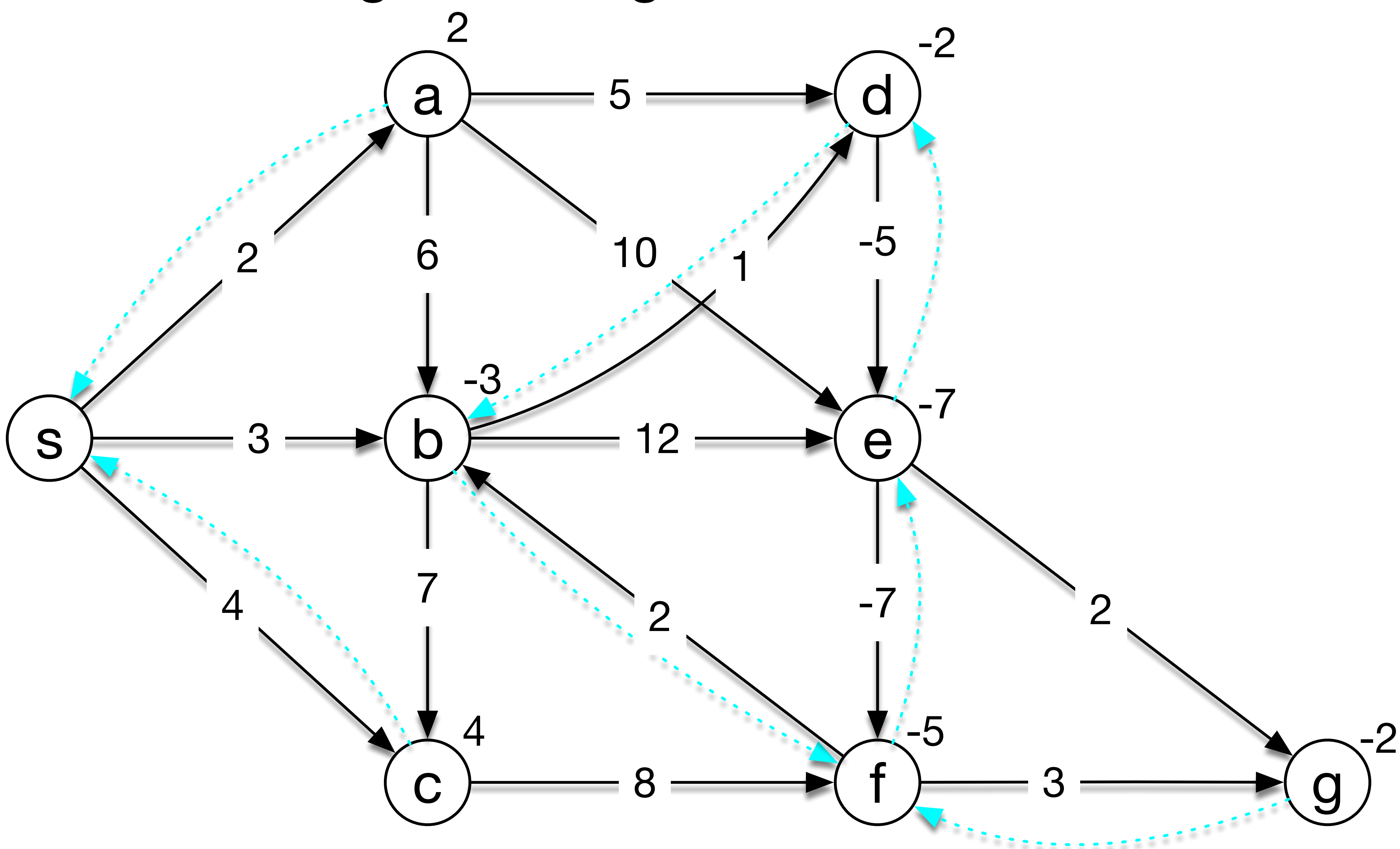
Bellman-Ford

- The first time around, we would get something like this
    - Your mileage will vary because of choices in the ordering of edges



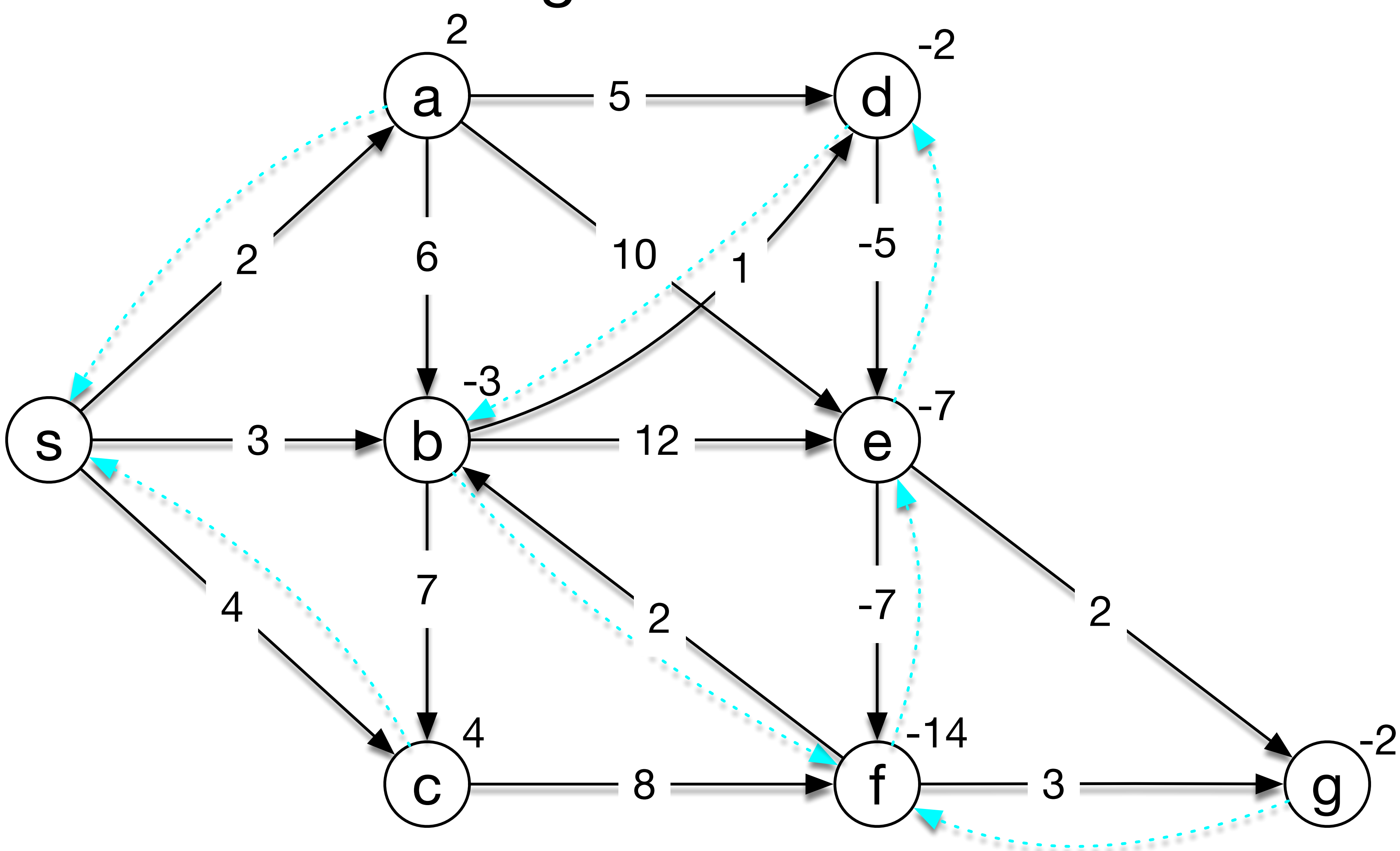
Bellman-Ford

- If we relax along d-e, we get



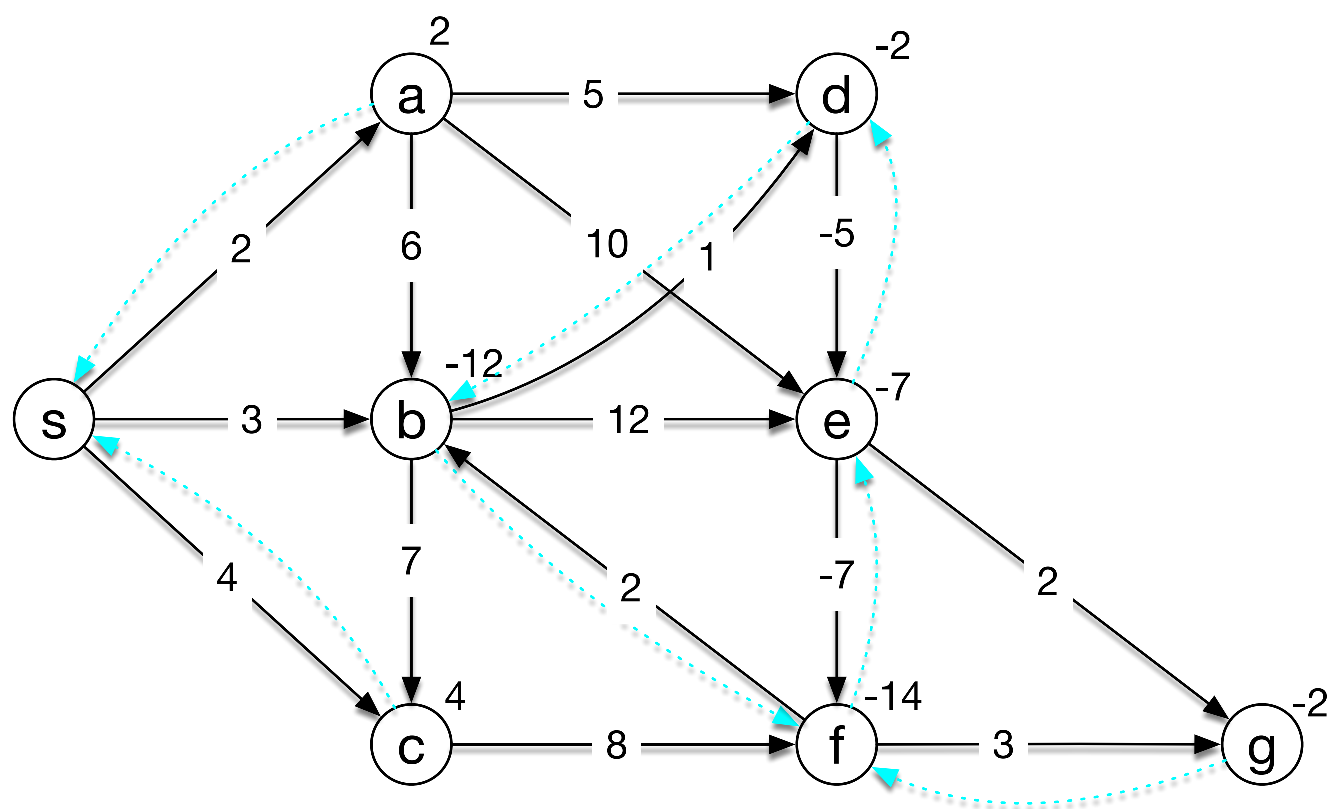
Bellman-Ford

- Then we can relax along e-f



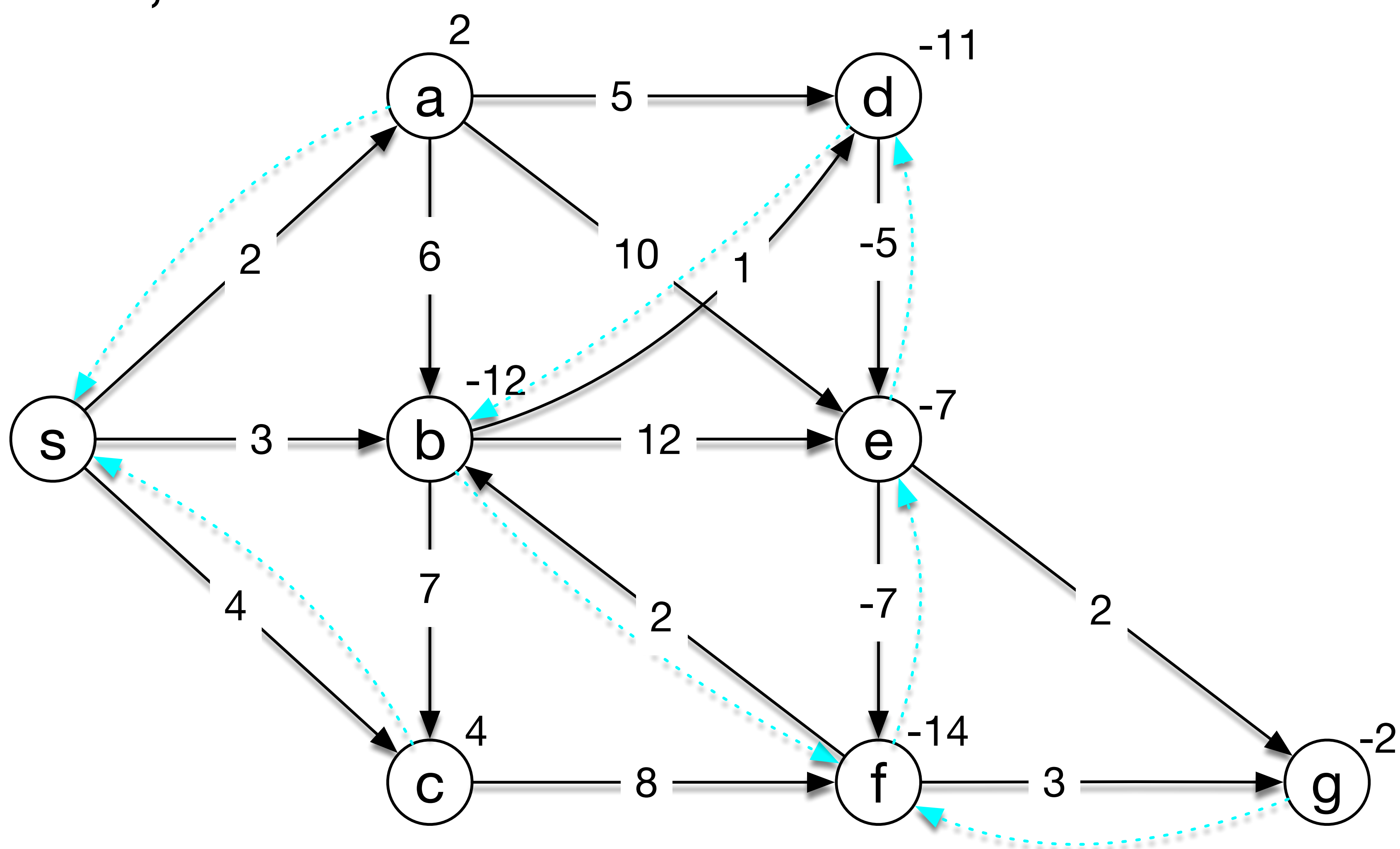
Bellman-Ford

- Then along f-b



Bellman-Ford

- After we relax with (b,d), we see that the predecessor graph no longer reaches s. We also can see that every time we relax with the edges in the graph, we lower distances,



DAG's

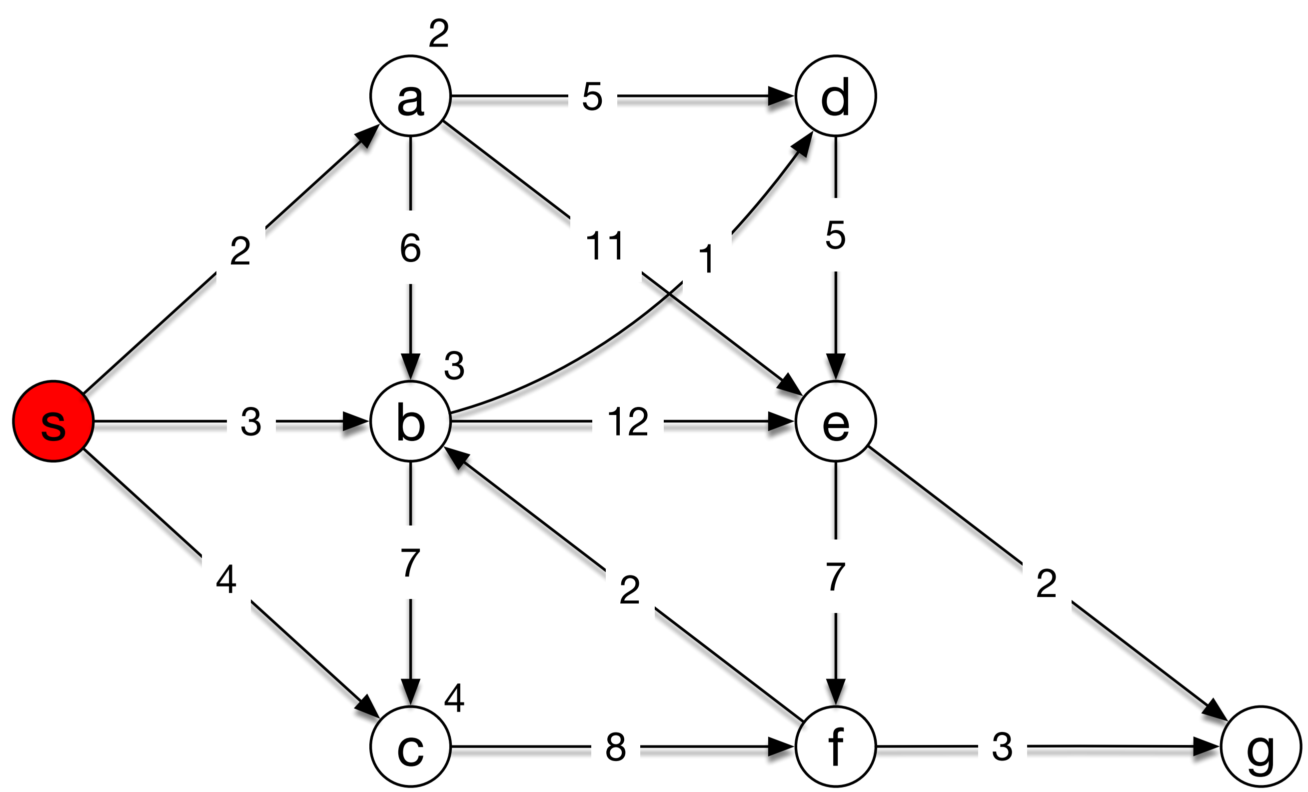
- If we have a directed acyclic graph:
  - We can use topological sort of vertices: $U = [u_1, u_2, u_3, \ldots, u_n]$
  - We then execute
    - for u in U:
      - for v in u.adjacency:
        - relax with (u,v)
  - This is a very fast algorithm for DAG's only

Dijkstra's Algorithm

- Single source all destinations algorithms that only assumes that weights are all positive
- Also uses a Greedy strategy
- Builds a subset S of nodes for which the exact distance is known

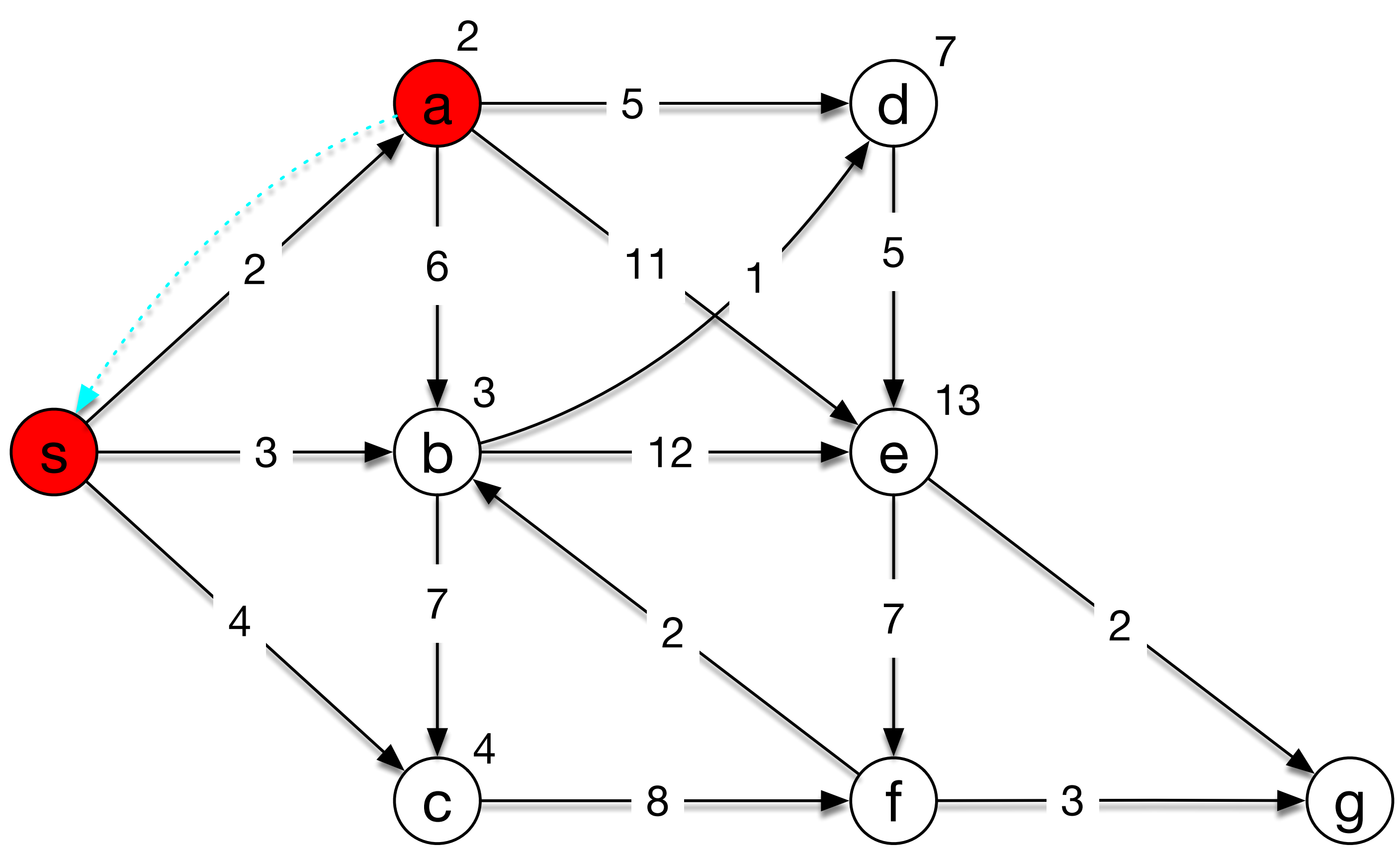
Dijkstra's Algorithm

- Example:
  - Dijkstra starts with just the source in S
  - We initialize all nodes (but do not write infinities here)



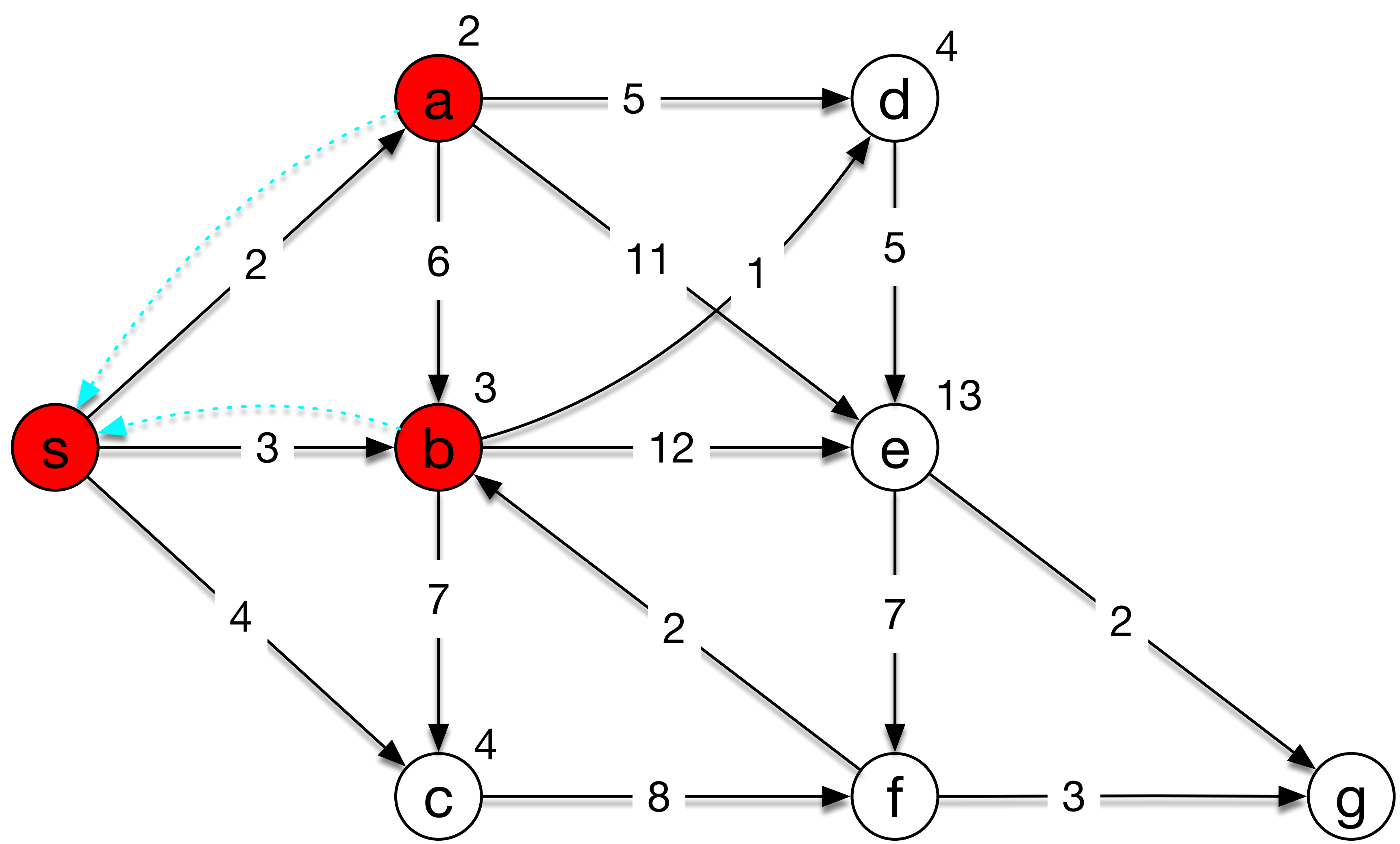
Dijkstra's Algorithm

- Dijkstra then adds the node u with the smallest distance to S and then uses the edges adjacent to u to relax
- Add a and relax (a,b), (a,d), and (a,e)



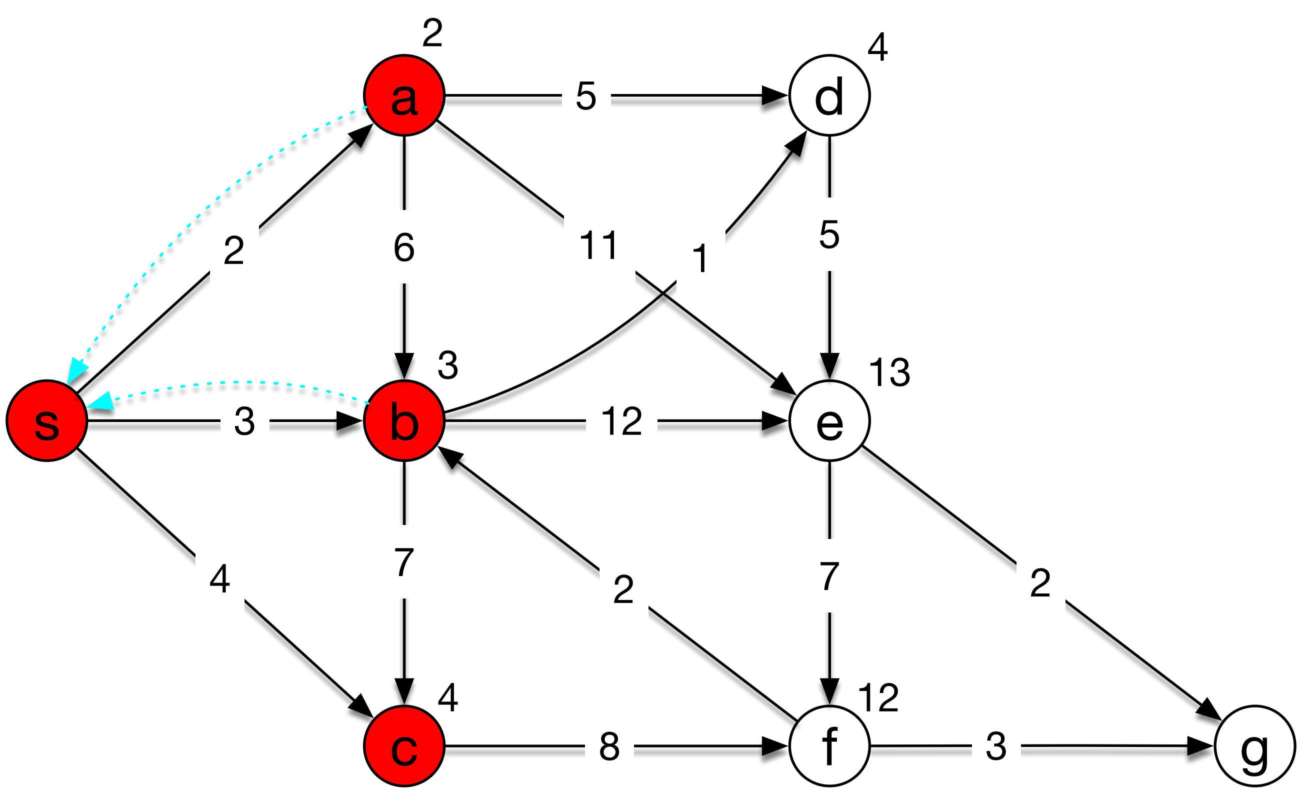
Dijkstra's Algorithm

- The current distances from s are 2, 3, 4, 7, and 13
- We pick b and relax along its outgoing edges



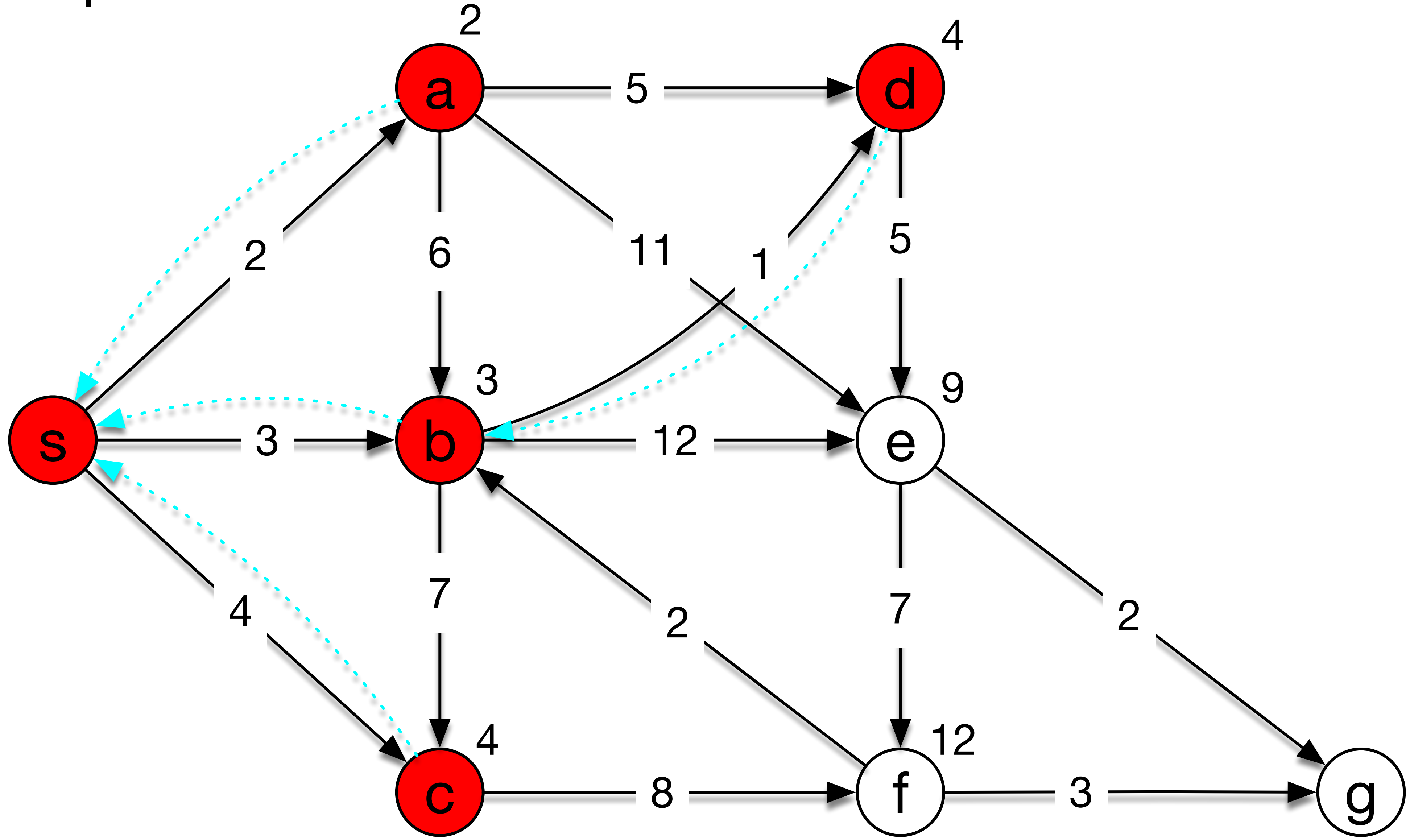
Dijkstra's Algorithm

- Now c and d is the lowest distance node, pick c



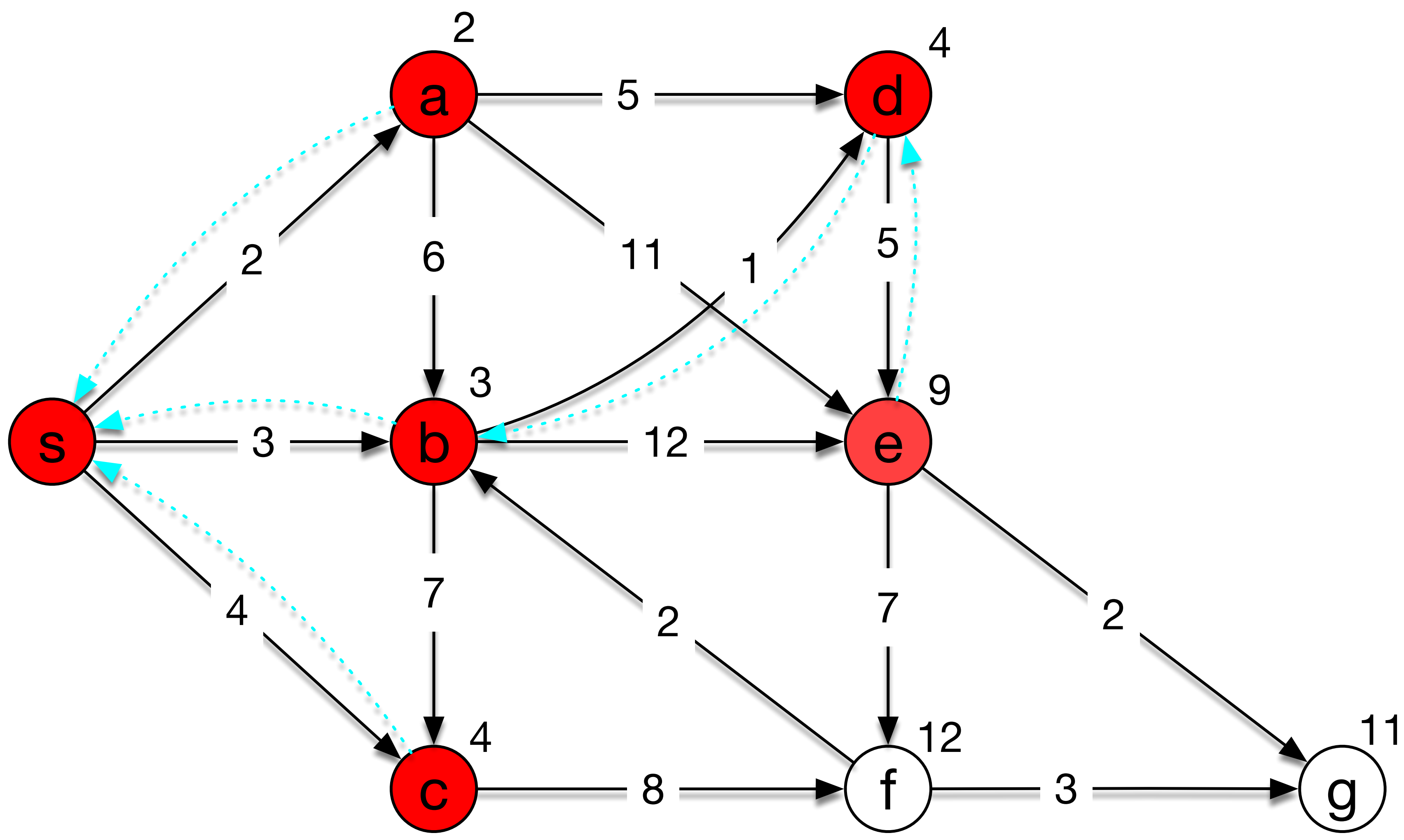
Dijkstra's Algorithm

- Now pick d



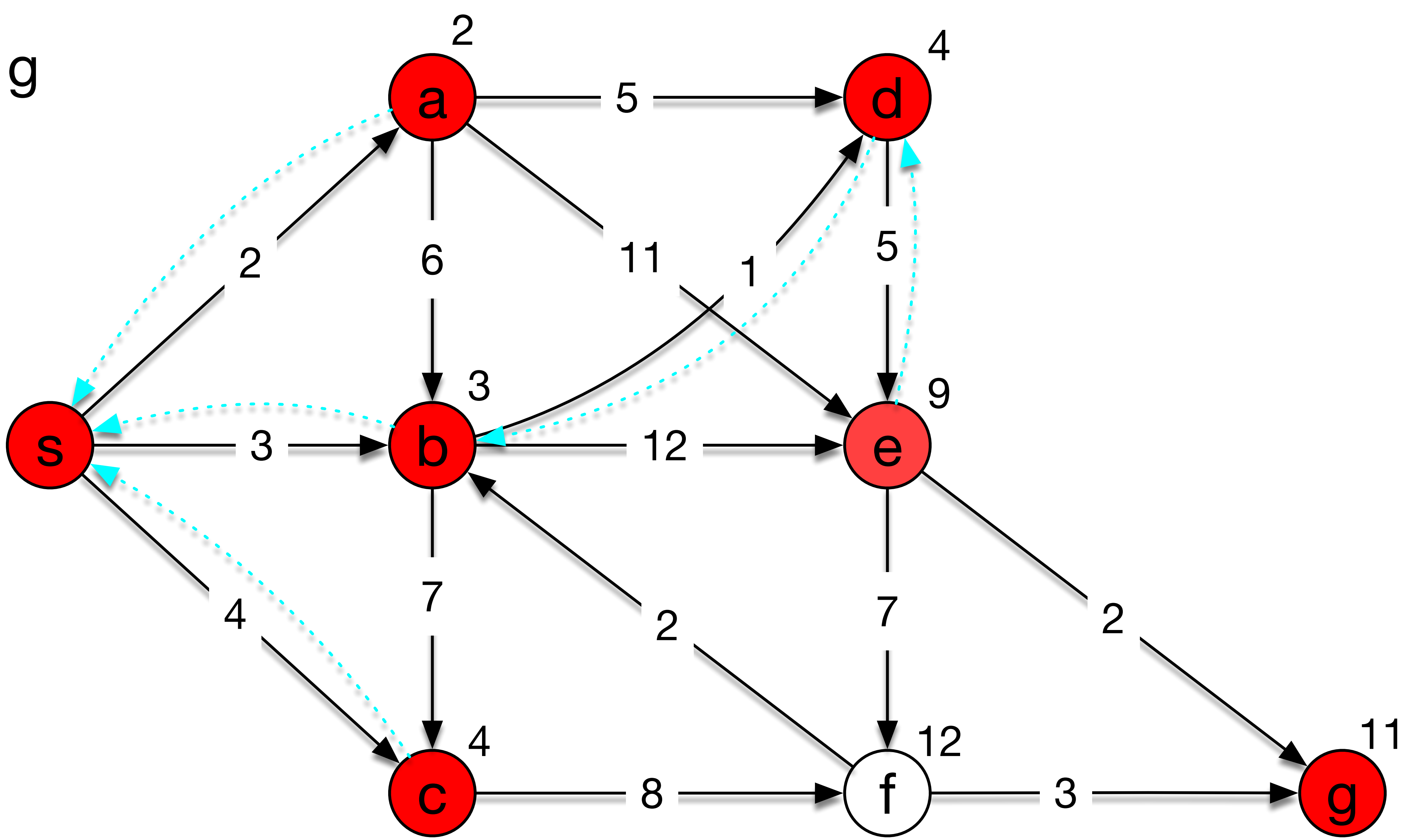
Dijkstra's Algorithm

- Then e



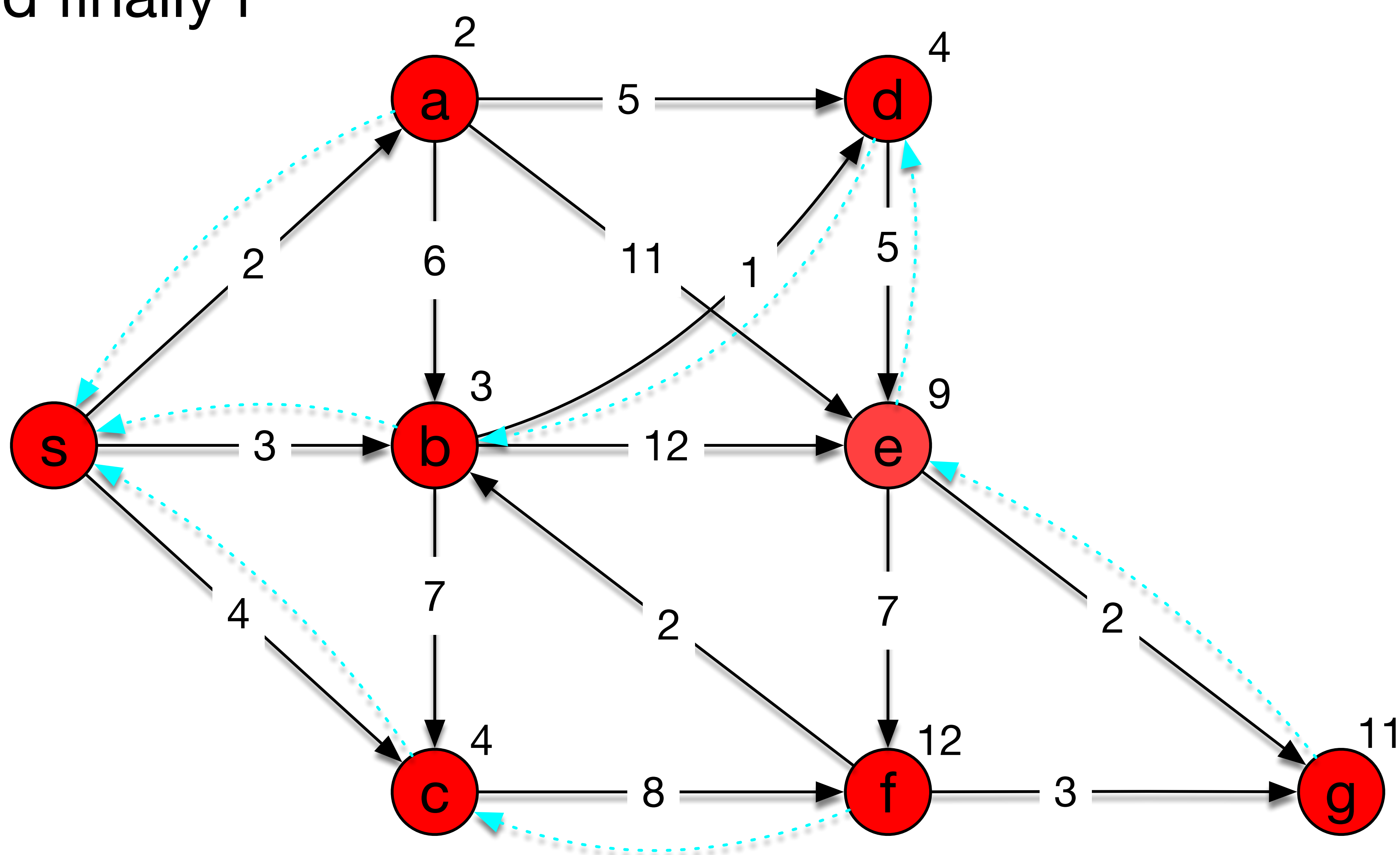
Dijkstra's Algorithm

- g



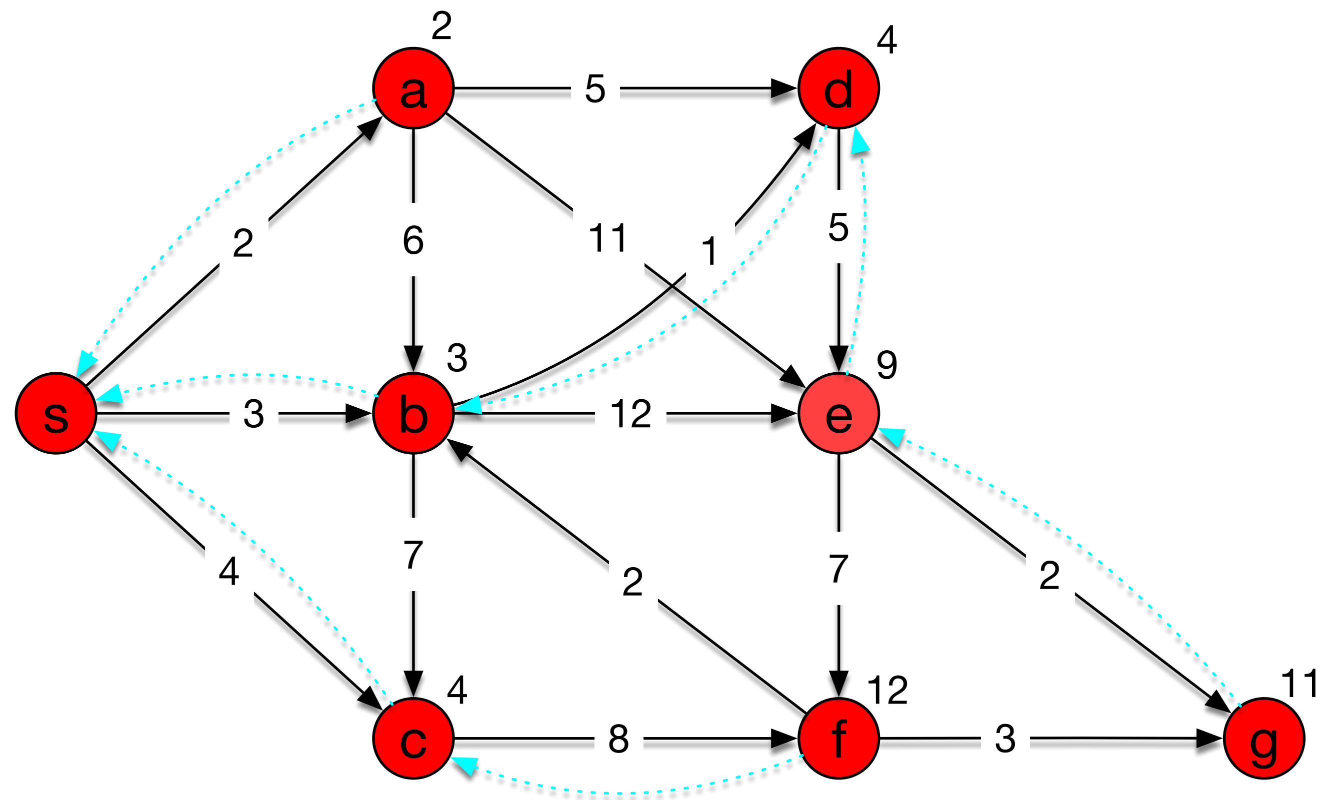
Dijkstra's Algorithm

- And finally f



Dijkstra's Algorithm

- To find the best way from s to g, we just start in g and follow the predecessor link



- g <— e <— d <—b<—s

Dijkstra's Algorithm

```
def dijkstra(s, V, E):
   initialize(s,V,E)
   S = [ ]
   pq = priorityqueue(V)
   while pq:
      u = pq.pop
      S.append(u)
      for v in u.adjacent:
          relax(u,v) #changes possibly v.d
                     #and therefore pq
```

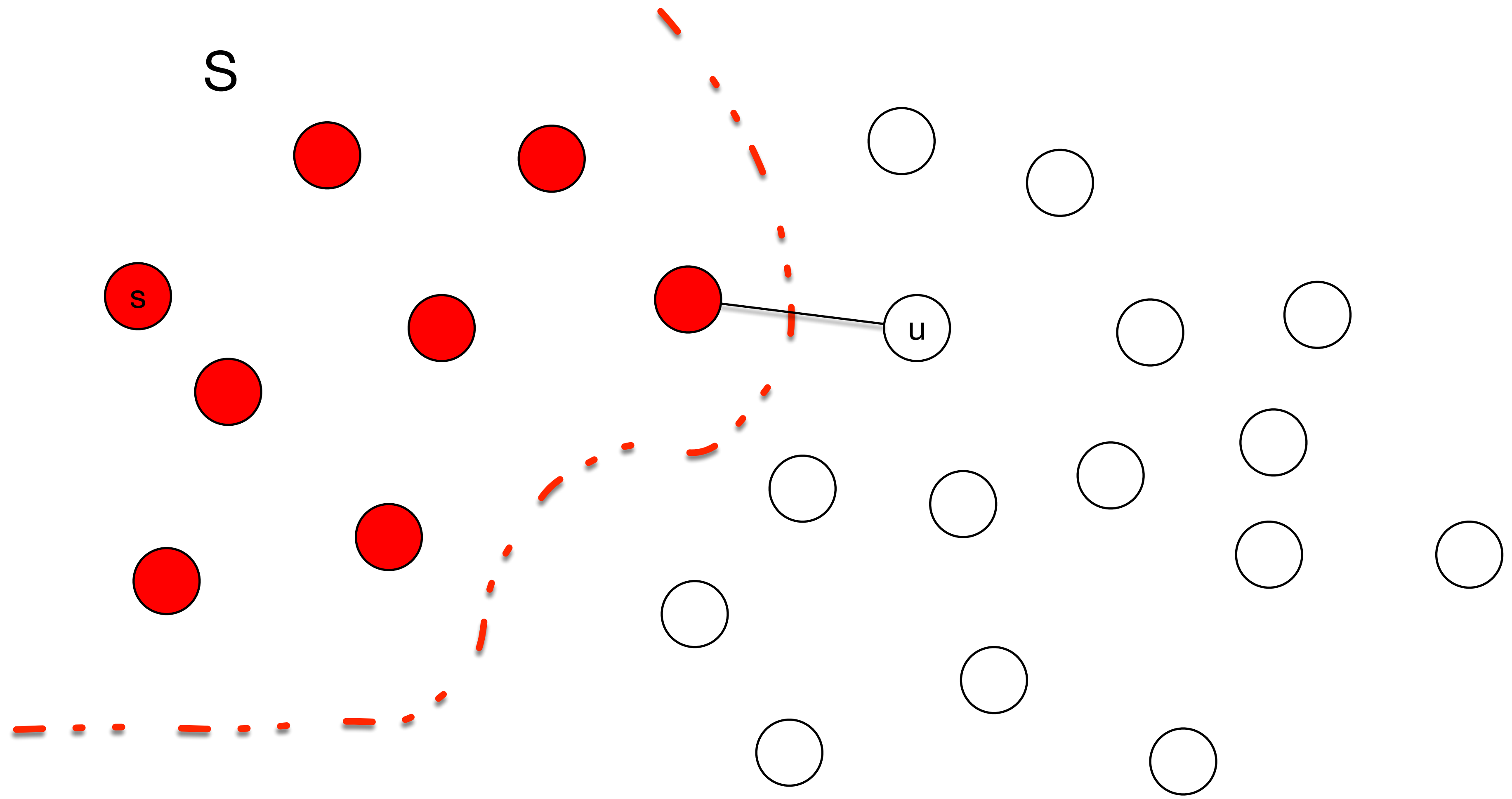
Dijkstra's Algorithm

- Correctness of Dijkstra's algorithm
  - Loop invariant for while loop:
    - All nodes in S have the correct distance from s

Dijkstra's Algorithm

- Initially, the loop invariant is vacuously true
- Now we need to show that each iteration of the while loop leaves the invariant valid
  - Assume that this is not true
  - And that we went wrong when u was selected and put into S

Dijkstra's Algorithm

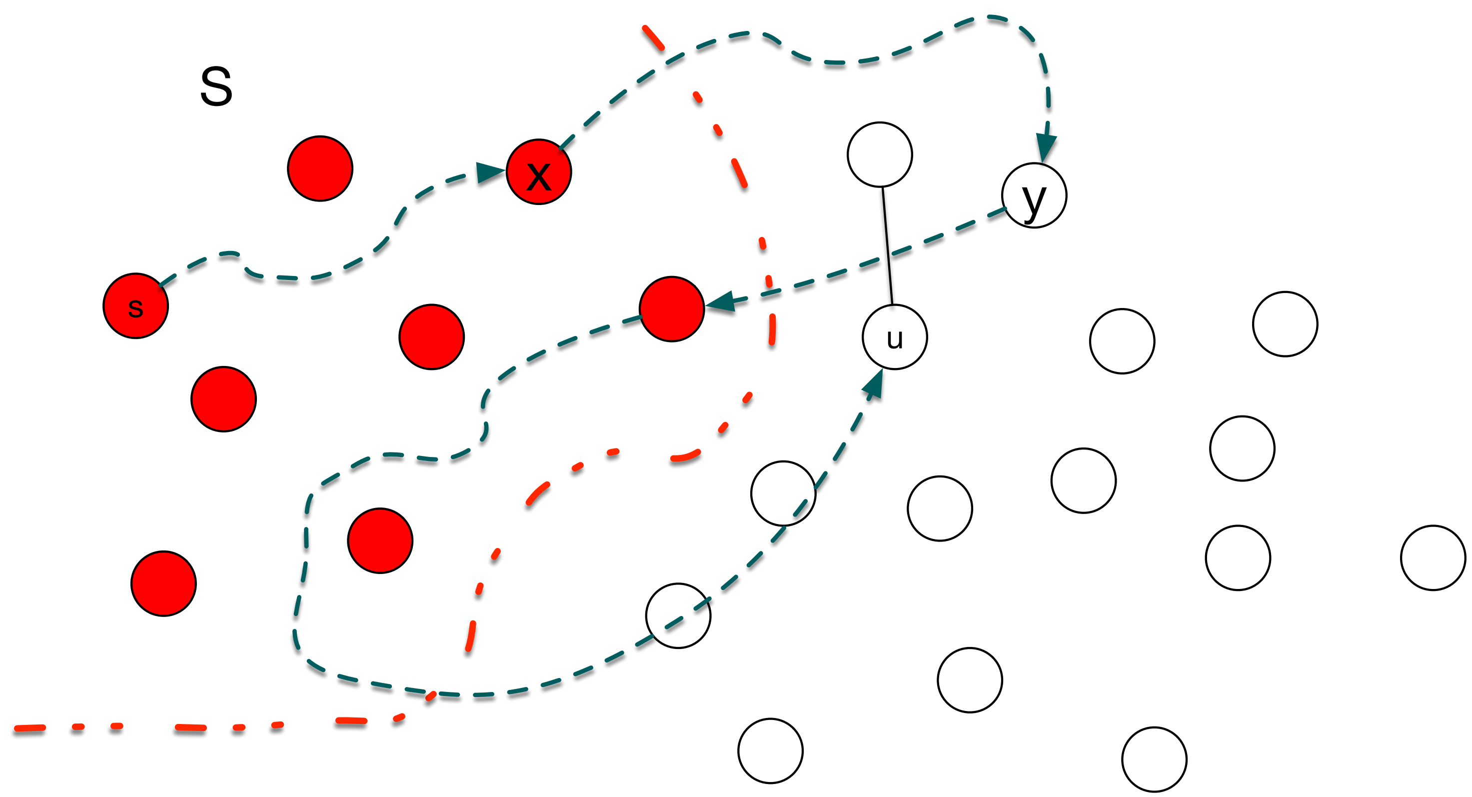


Dijkstra's Algorithm

- We can exclude the case that $s = u$ so $S \neq \emptyset$
- We can also exclude the case that there is no path between s and u
  - Because then the distance field in u would never be updated and stay at infinity, which is the correct value

Dijkstra's Algorithm

- So, there is a path from s to u that leads from S to outside of S.
- Let x be the last vertex on this path so that the part from s to x is completely in S and y the first node not in S



Dijkstra's Algorithm

- By the invariant, $x \in S$ has the correct distance field with $x.d = \delta(s, x)$
- Because $x \in S$, we relaxed along the edge (x, y)
  - Because x and y are on a shortest path from s to u, the shortest path from s to y also goes through x
  - Therefore, $y.d = x.d + w(x, y) = \delta(x, s) + w(x, y)$ at this time
  - But then this value can no longer change, so it is still true

Dijkstra's Algorithm

- Now, we selected u before y
  - This implies $u.d \leq y.d$
  - But because y is on the shortest path to u
  - $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d \leq u.d$
  - Which implies that $\delta(s, y) = \delta(s, u)$ and $y = u$ (because all weights are positive)
  - But we have already seen that $y.d = u.d$ was set correctly
- This contradiction proves correctness

Dijkstra's Algorithm

- Dijkstra's algorithm runs in time dependent on the implementation of the priority queue
- We update the priority queue potentially for each edge analyzed, which is all of them
  - Easiest implementation uses time quadratic in the number of vertices
- This gives a total runtime quadratic in the number of vertices because $|E| = O(|V|^2)$

Floyd's and Warshal's Algorithm

- Dynamic programming approach to the all sources - all destinations shortest path problem
- What is the shortest path from u to v going through nodes in $\{v_1, v_2, \ldots, v_k\}$ as intermediaries?
  - Call its length $\delta_k(u, v)$
  - Two cases: v_k is not on the shortest path:
    - $\delta_k(u, v) = \delta_{k-1}(u, v)$
  - Or: v_k is on the shortest path
    - $\delta_k(u, v) = \delta_{k-1}(u, u_k) + \delta_{k-1}(u_k, v)$

Floyd's and Warshal's Algorithm

- Start out with
  $$\delta_0(u,v) = \begin{cases} w(u,v) & \text{if there is an edge between } u \text{ and } v \\ \infty & \text{if there is no edge} \end{cases}$$
- Calculate
  $$\delta_k(u,v) = \min(\delta_{k-1}(u,v), \delta_{k-1}(u,u_k) + \delta_{k-1}(u_k,v))$$
- If $k = |V|$, then $\delta_k(u,v) = \delta(u,v)$
- To reconstruct shortest path, give way points.
  - If u_k is an intermediate node, then put u_k down for the connection (u,v)

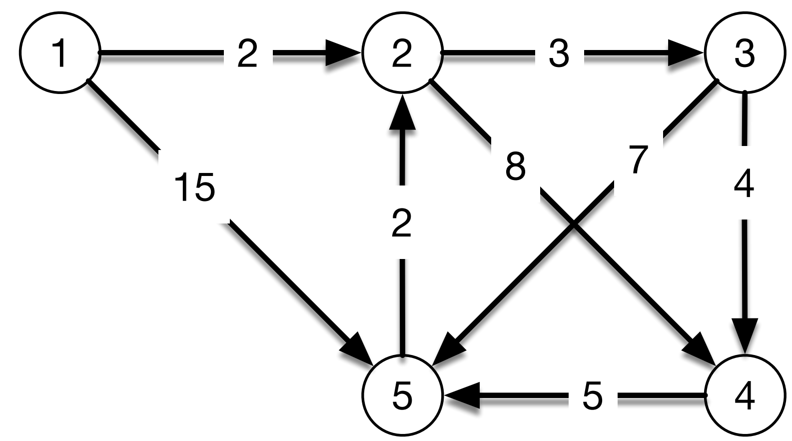
Floyd's and Warshal's Algorithm

- We can use a single matrix D for the distance calculation
- Initially, D contains the weights and Π only nones

```
D = W
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



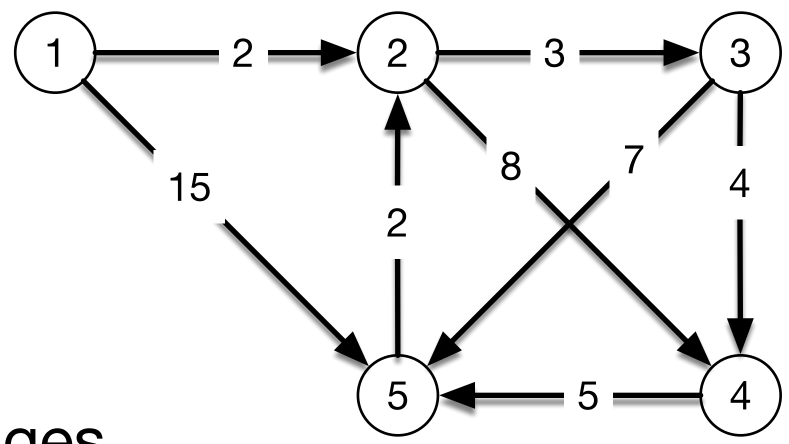
Set-up

	1	2	3	4	5
1	0	2			15
2		0	3	8	
3			0	4	7
4				0	5
5		2			0

	1	2	3	4	5
1					
2					
3					
4					
5					

Floyd's and Warshal's Algorithm

- Example:



k=1: no changes

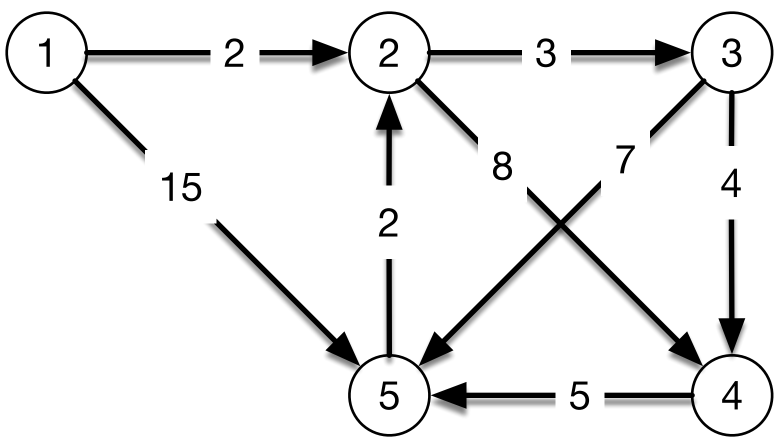
	1	2	3	4	5
1	0	2			15
2		0	3	8	
3			0	4	7
4				0	5
5		2			0

	1	2	3	4	5
1					
2					
3					
4					
5					

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



k=2

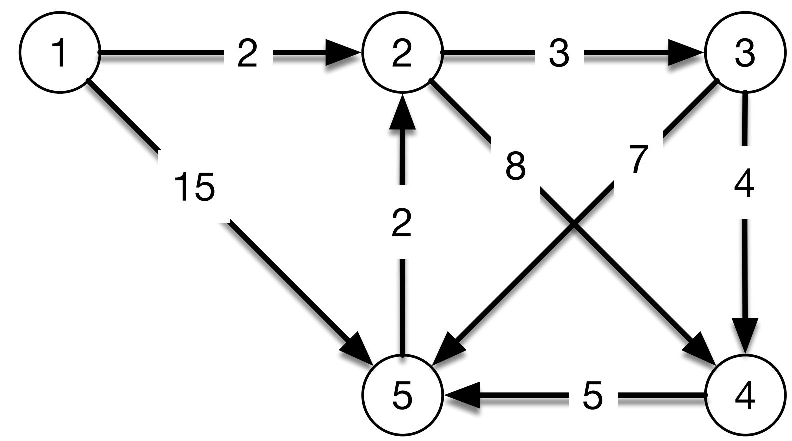
	1	2	3	4	5
1	0	2	5	10	15
2		0	3	8	
3			0	4	7
4				0	5
5		2	5	10	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



k=3

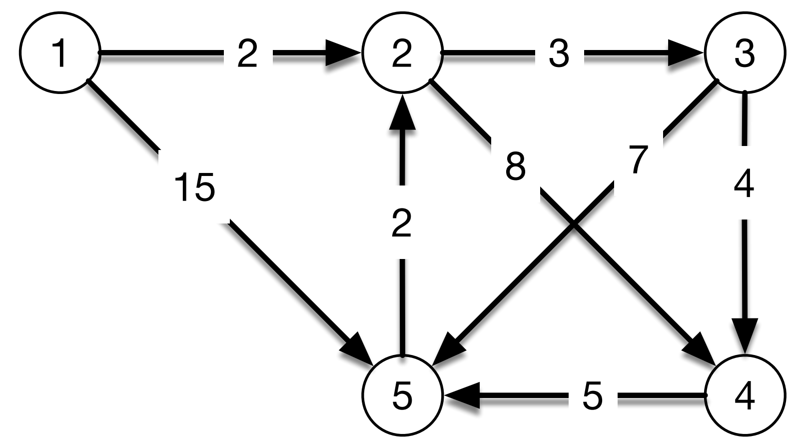
	1	2	3	4	5
1	0	2	5	9	15
2		0	3	7	10
3			0	4	7
4				0	5
5		2	5	10	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



k=4

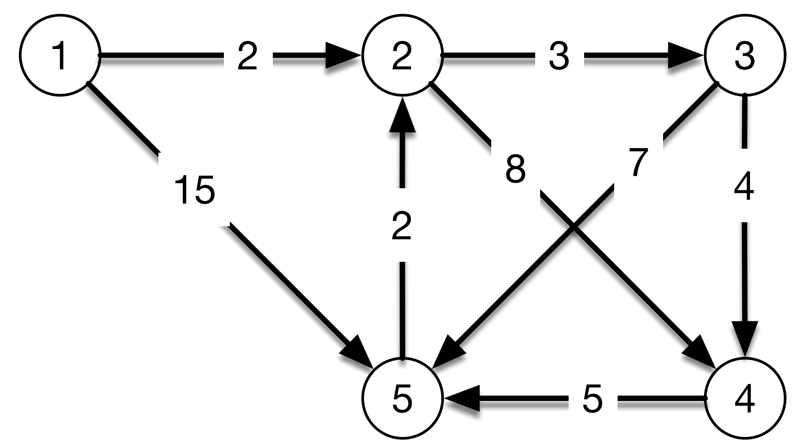
	1	2	3	4	5
1	0	2	5	9	12
2		0	3	7	10
3			0	4	7
4				0	5
5		2	5	10	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



k=5

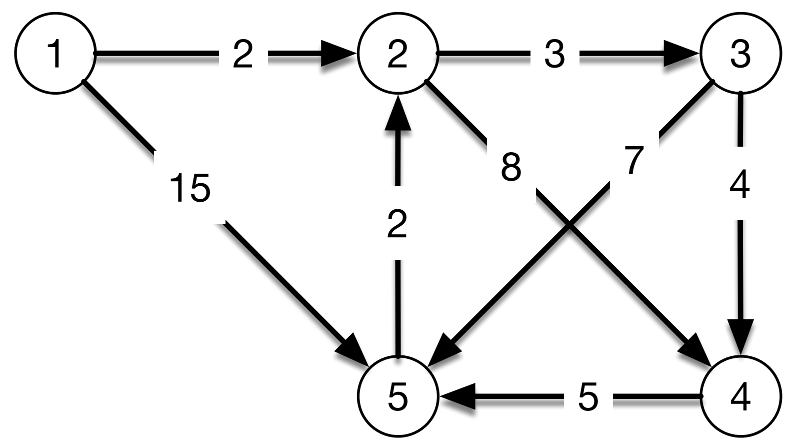
	1	2	3	4	5
1	0	2	5	9	12
2		0	3	7	10
3		9	0	4	7
4		7	10	0	5
5		2	5	9	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



Going from 1 to 5:
D matrix: 12
Π matrix: go through 3
1 to 3: go through 2
3 to 5: direct

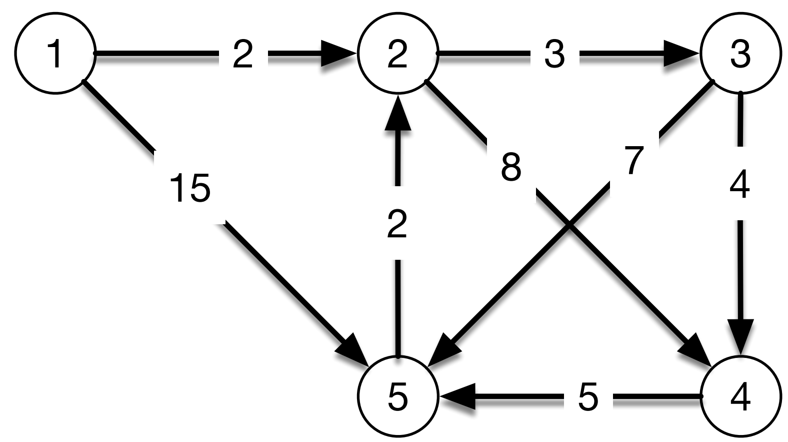
	1	2	3	4	5
1	0	2	5	9	12
2		0	3	7	10
3		9	0	4	7
4		7	10	0	5
5		2	5	9	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
            d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```

Floyd's and Warshal's Algorithm

- Example:



Going from 5 to 4:
D matrix: 9
Π matrix: go through 2
5 to 2: direct
2 to 4: go through 3
2 to 3: direct
3 to 4: direct
route is 5-2-3-4

	1	2	3	4	5
1	0	2	5	9	12
2		0	3	7	10
3		9	0	4	7
4		7	10	0	5
5		2	5	9	0

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

```
for k in {1 … n}:
    for i in {1 … n}:
        for j in {1 … n}:
             d[i,j] = min(d[i,j], d[i,k]+d[k,j]
```