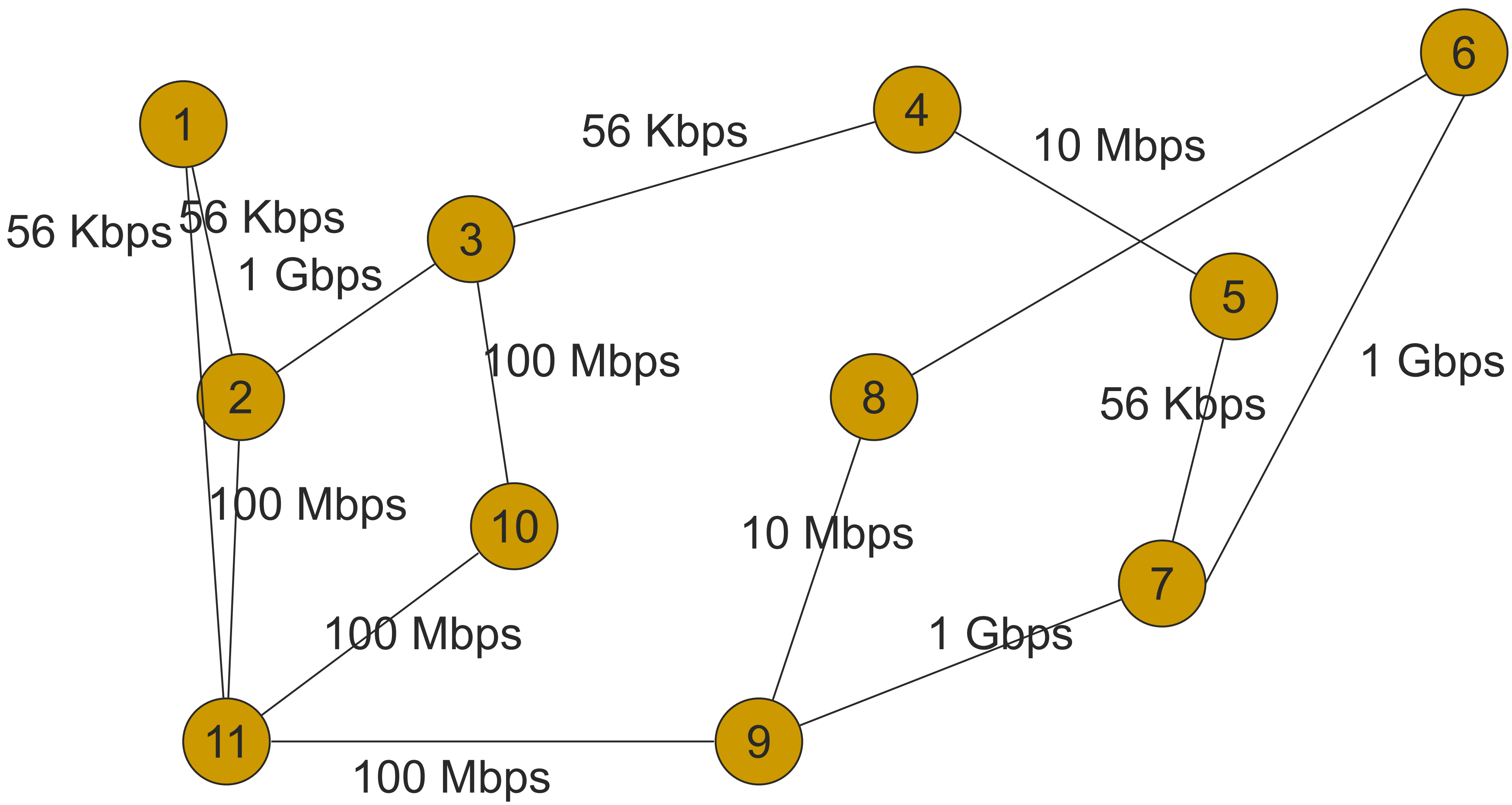
Shortest Paths

in edge-weighted graphs

Communication Speeds in a Computer Network

Find fastest way to route a data packet between two computers



shortest paths in edge weighted graphs

- Find the fastest way to travel across the country using a graph representing roads, with edge weights represented by
  - distances
  - travel times (to accommodate speed limits) between cities
  - Flight distances between airports

Shortest Path problems

- Single source shortest paths
- Find a shortest path between two given vertices
- All pair shortest paths

Single Source Shortest Path problems

- Undirected graphs with non-negative edge weights
- Directed graphs with non-negative edge weights
- Directed graphs with arbitrary weights

Single Source Shortest Paths

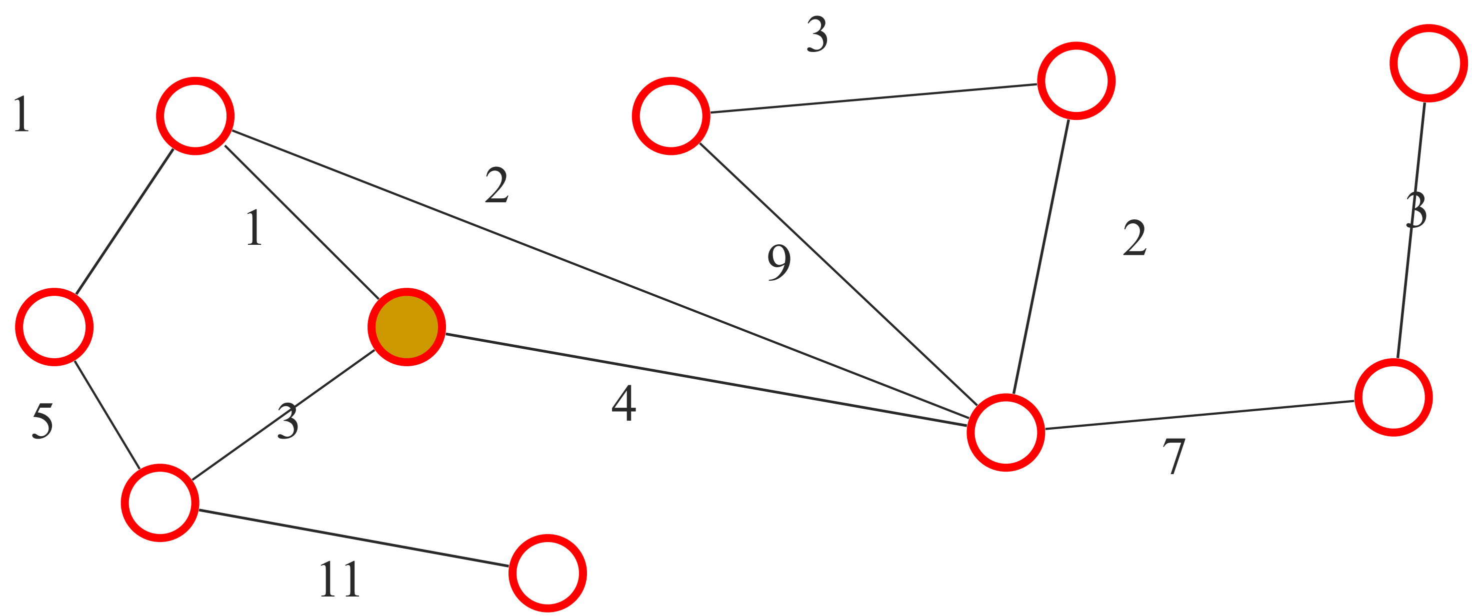
- If graph is not weighted (all edge-weights are unit-weight): BFS works
- Now assume: graph is edge-weighted
  - Every edge is associated a positive number
    - Possible weights: integers, real numbers, rational numbers
    - Edge-weights can represent: distance, connection cost, affinity

Single Source Shortest Paths

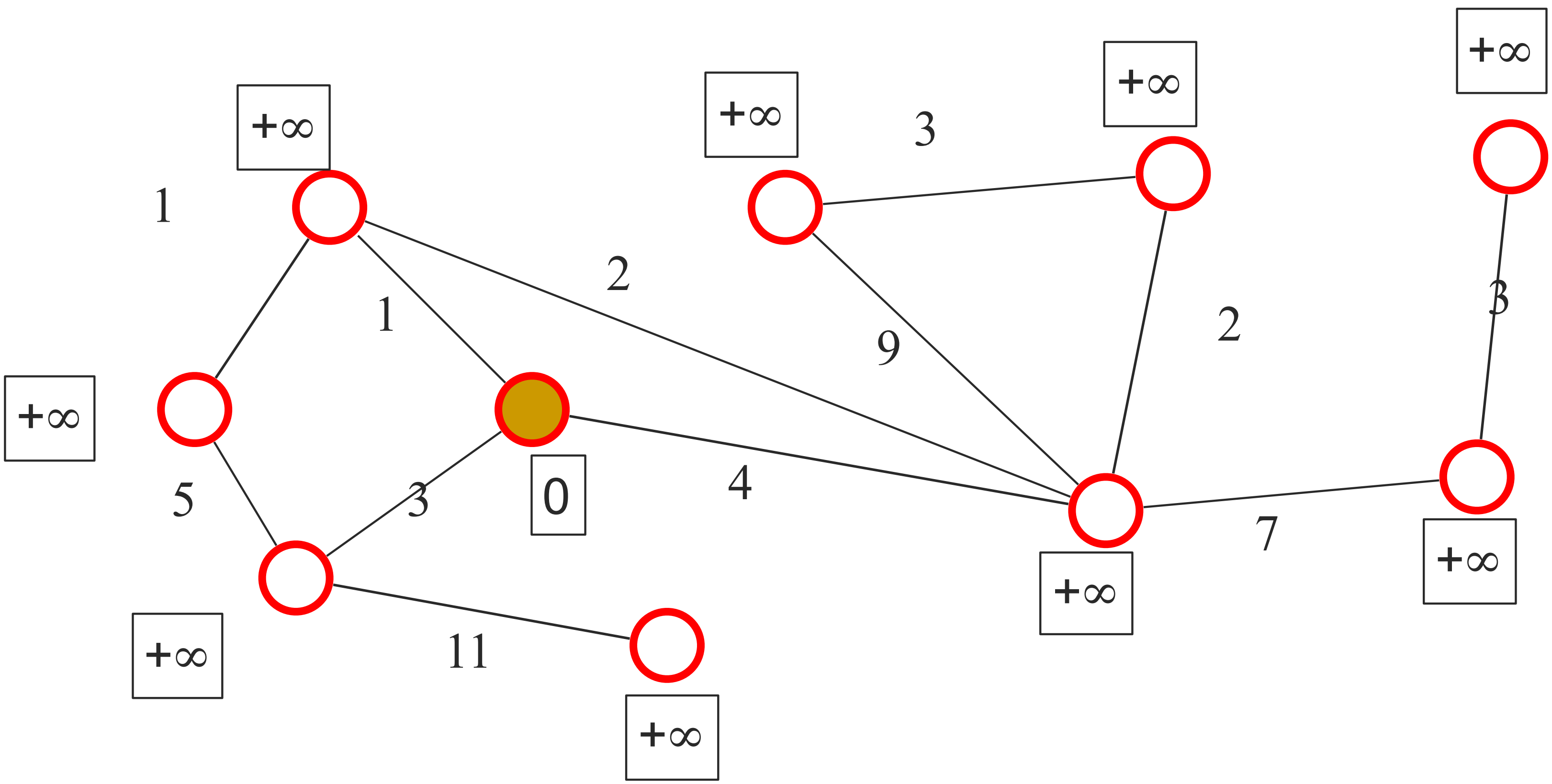
- Input: An edge-weighted undirected graph and a source node v with: for every edge e edge-weight w(e) > 0
- Output: All single-source shortest paths (end their weight) for v in G: for every node $w \neq v$ in G a shortest path from v to w.
  - Here, a *path* p from v to w consisting of edges e_1, e_2, … , e_n is shortest in G, if its length

    w(p) = $\sum_{i=0}^{k-1} w(e_i)$ is minimum (i.e, there is no path from
  - v to w in G that is shorter).

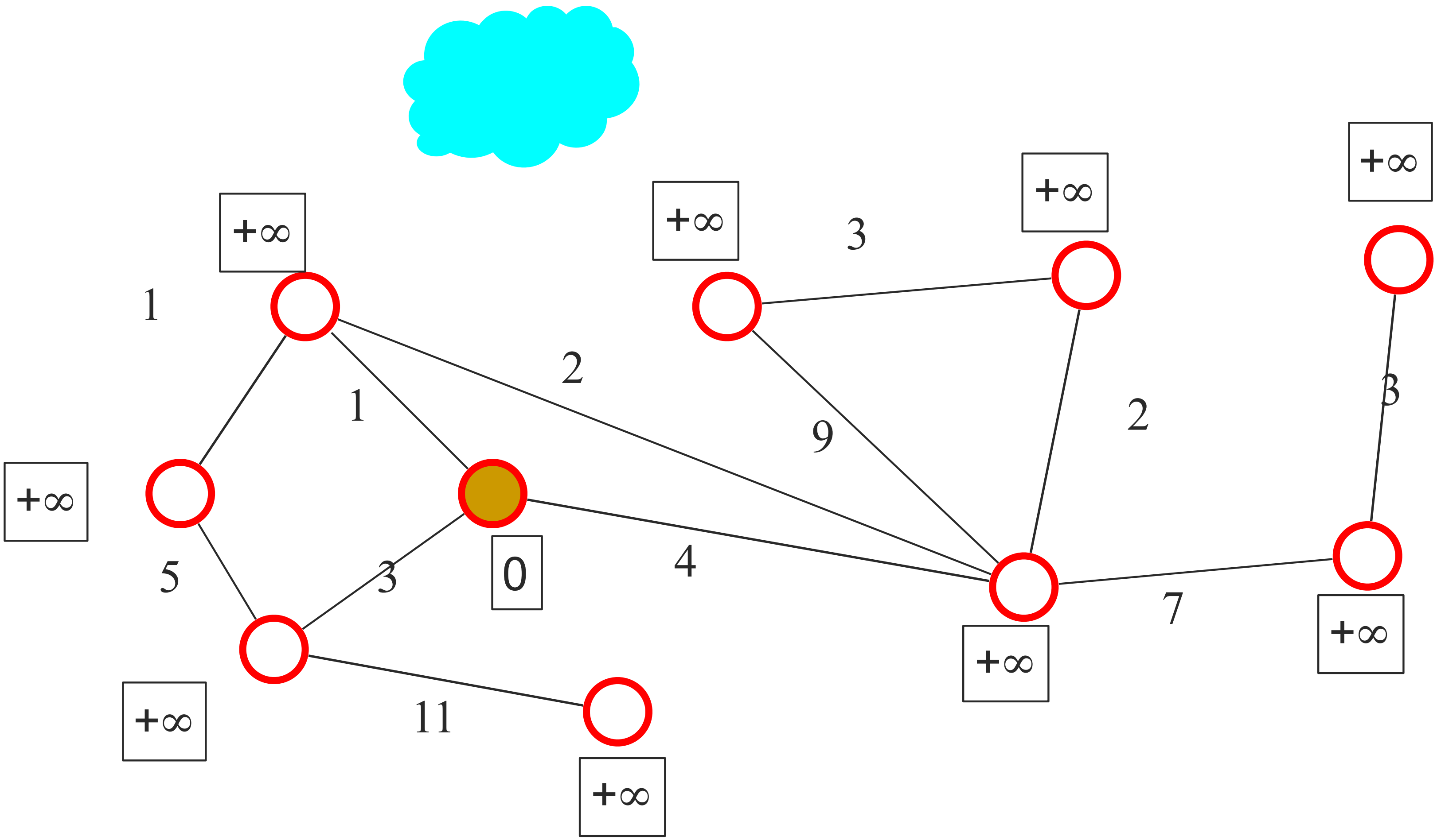
Dijkstra’s algorithm: a greedy algorithm



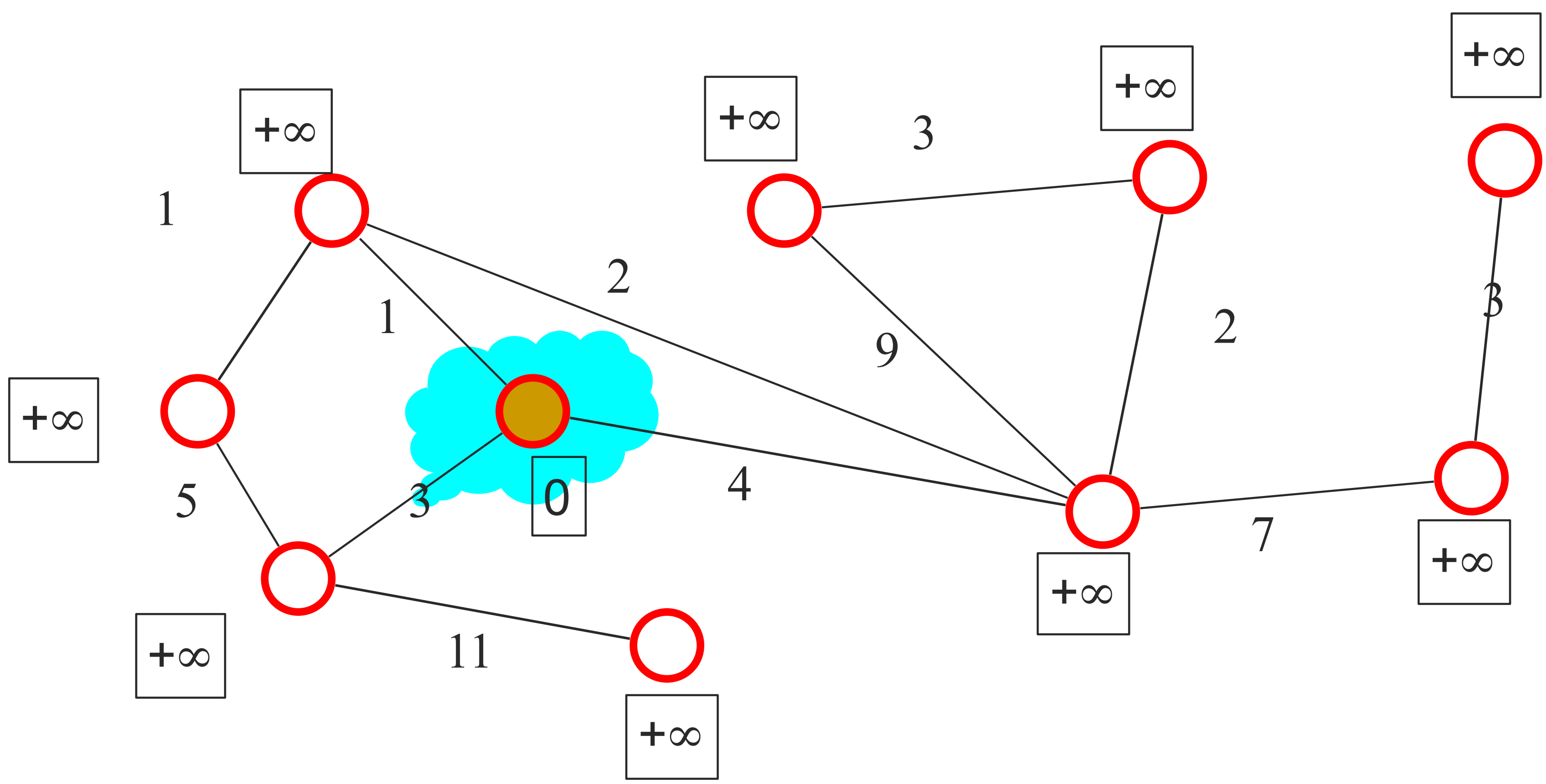
Dijkstra's algorithm: Initializing



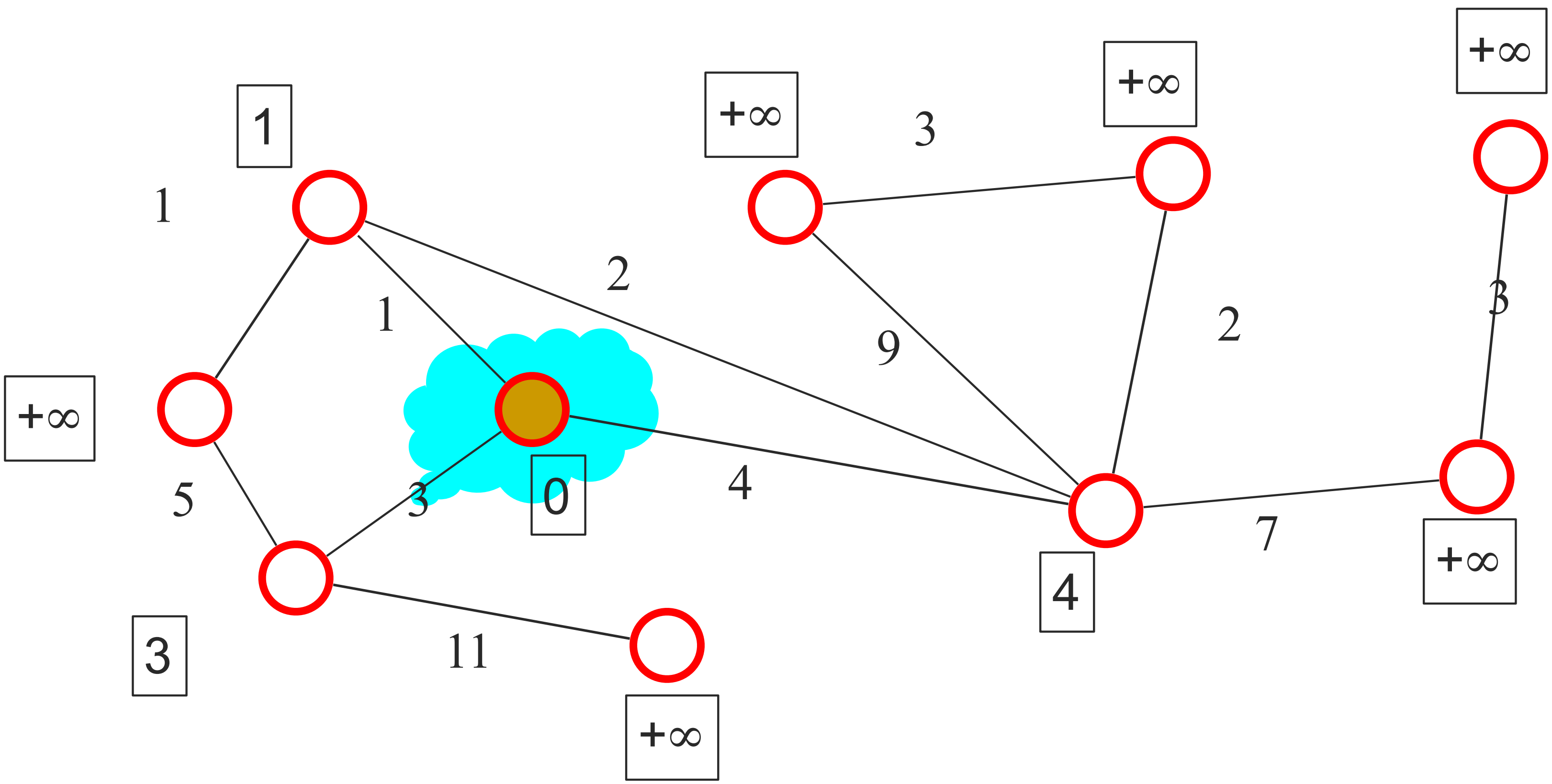
Dijkstra's algorithm: Initializing Cloud C (consisting of "solved" subgraph)



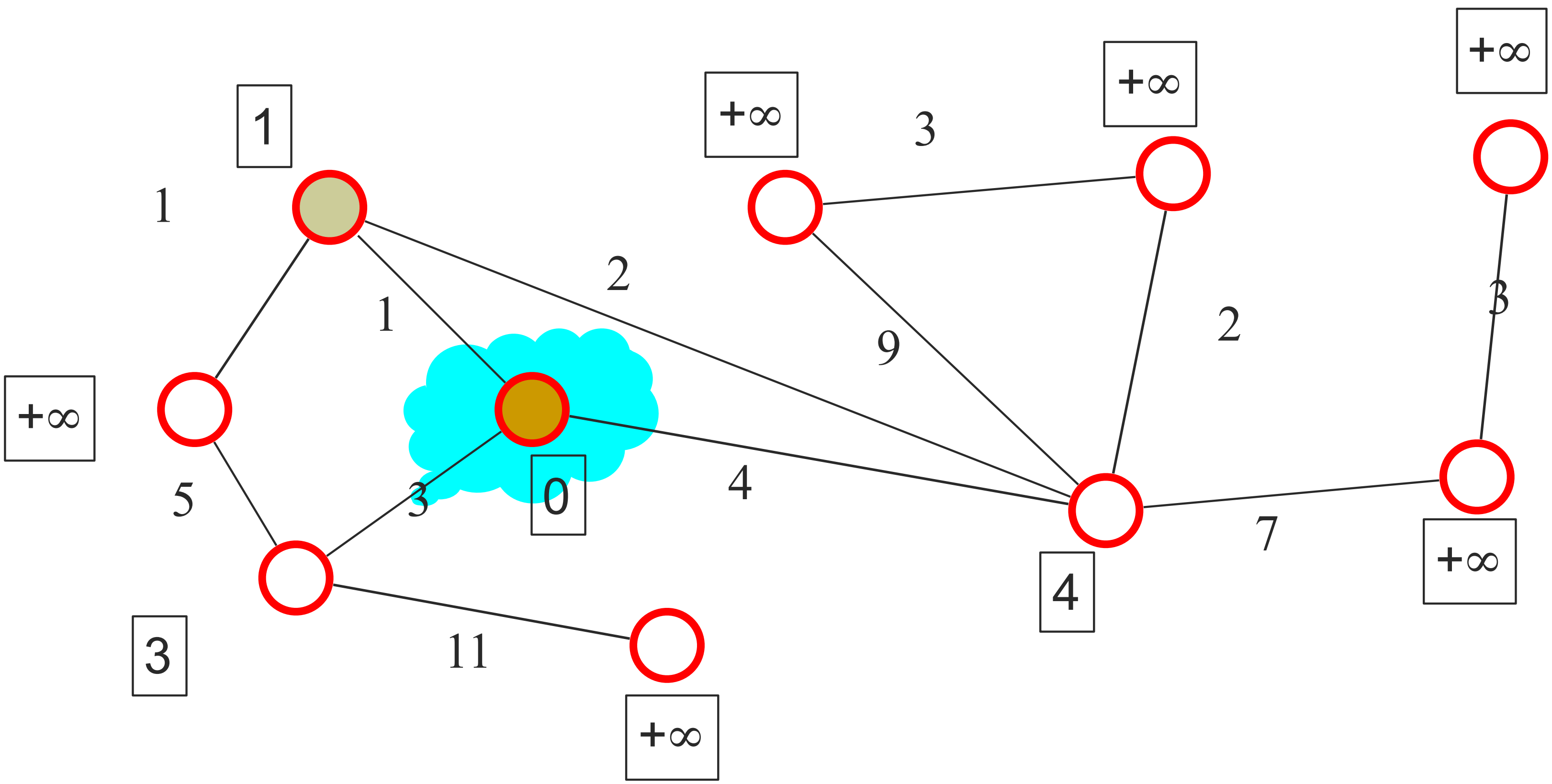
Dijkstra's algorithm: pull v into C



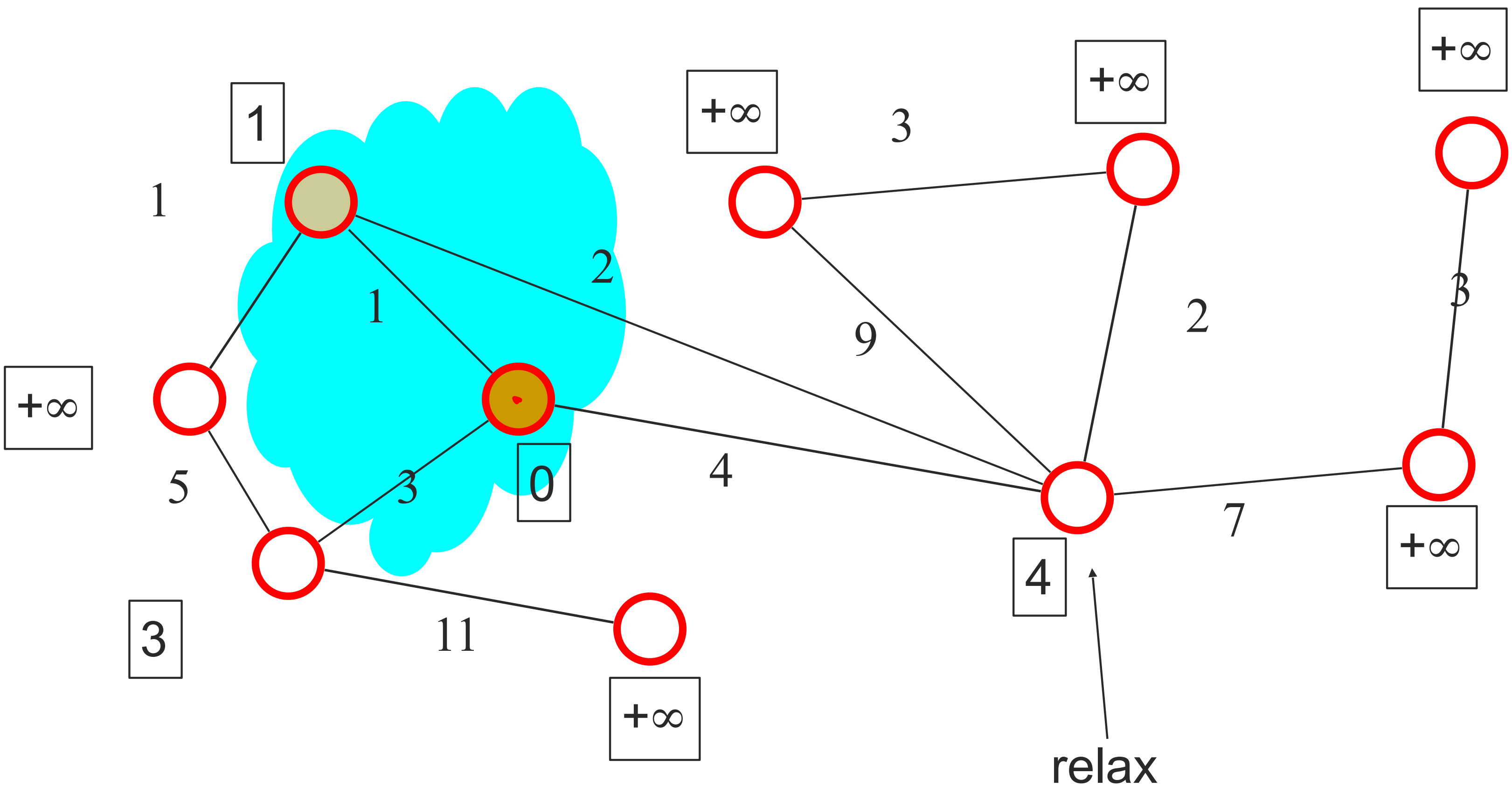
Dijkstra's algorithm: update *C's* neighborhood



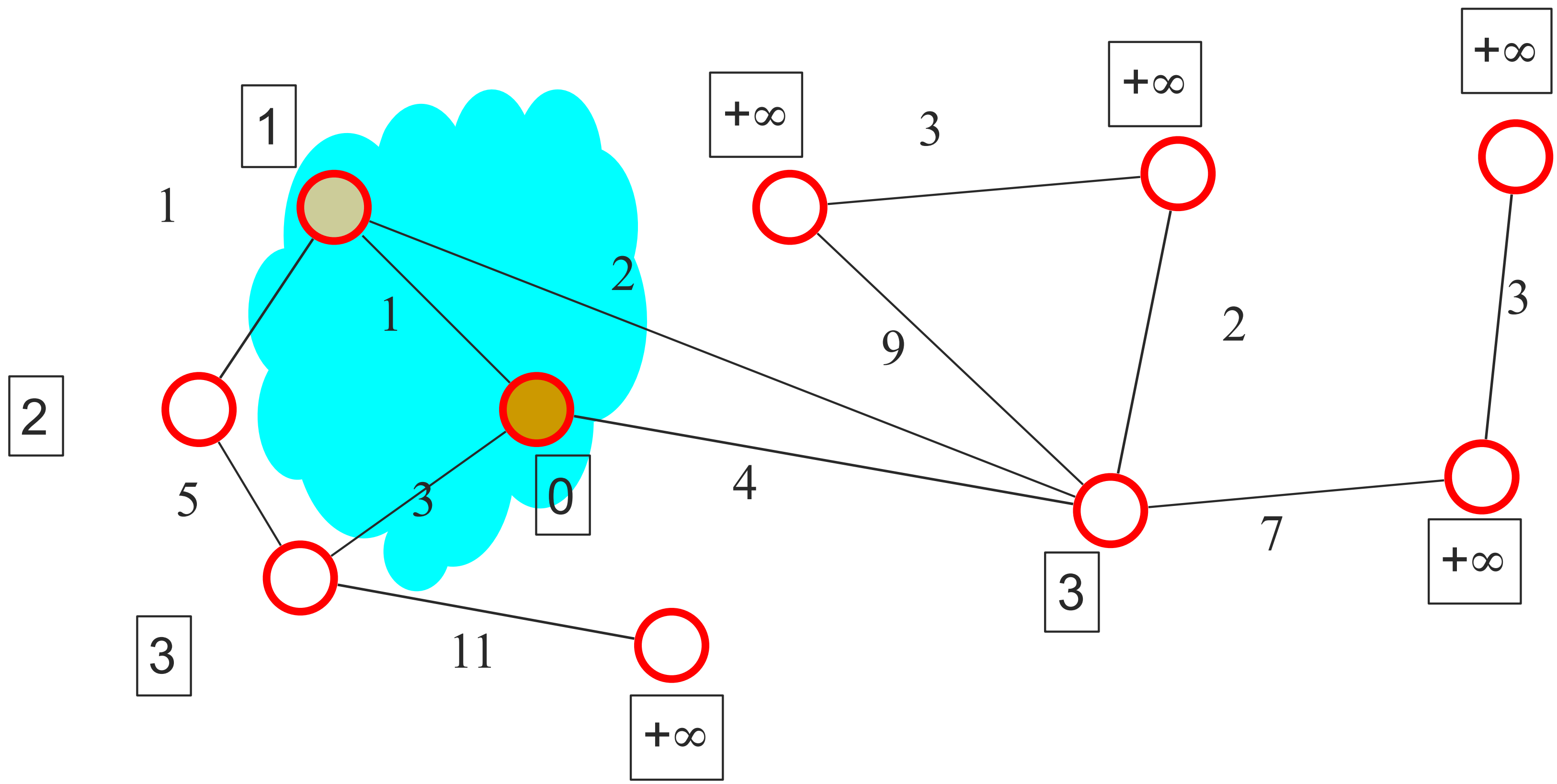
Dijkstra's algorithm: pick closest vertex u outside C



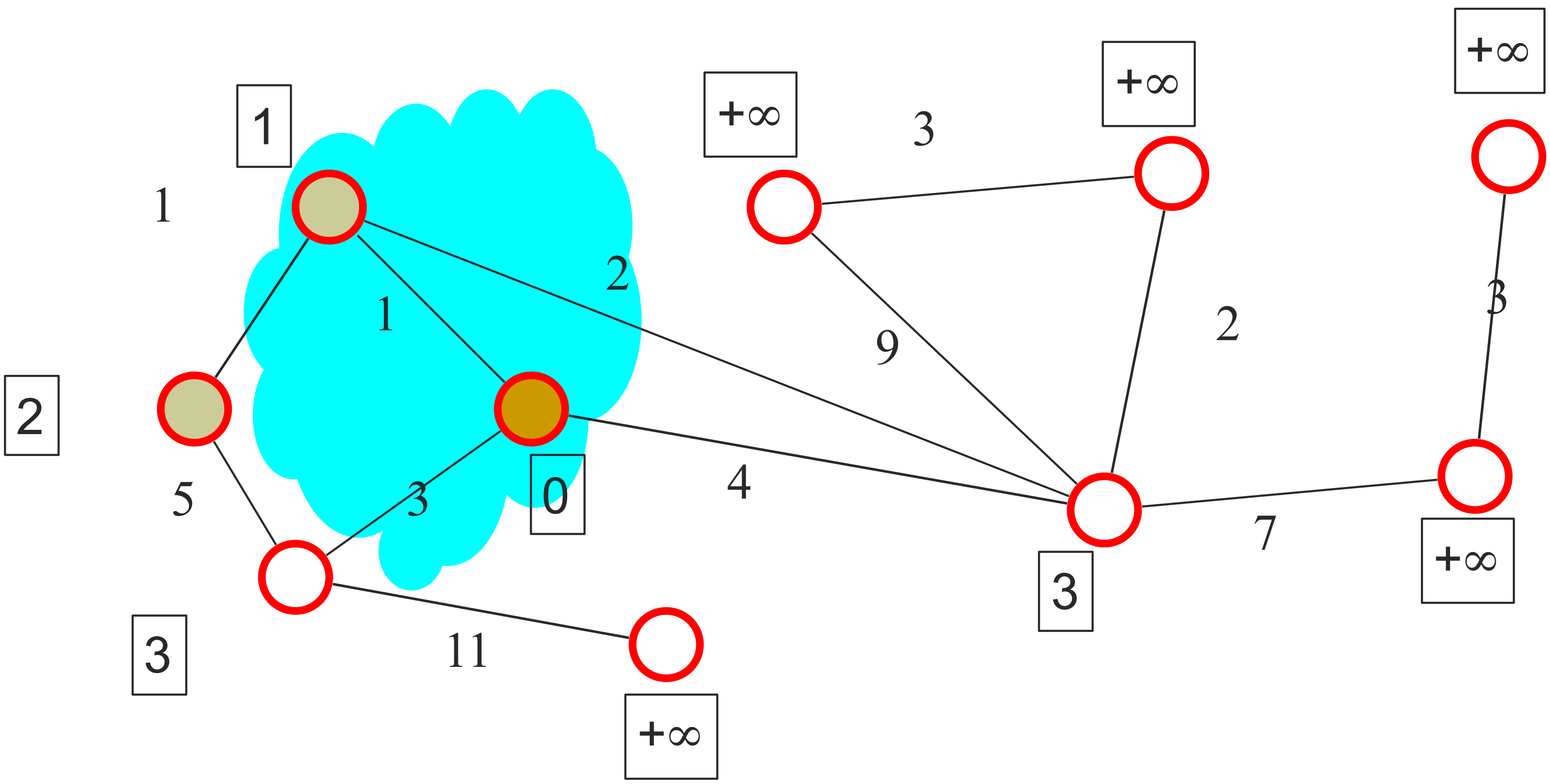
Dijkstra's algorithm: pull u into C



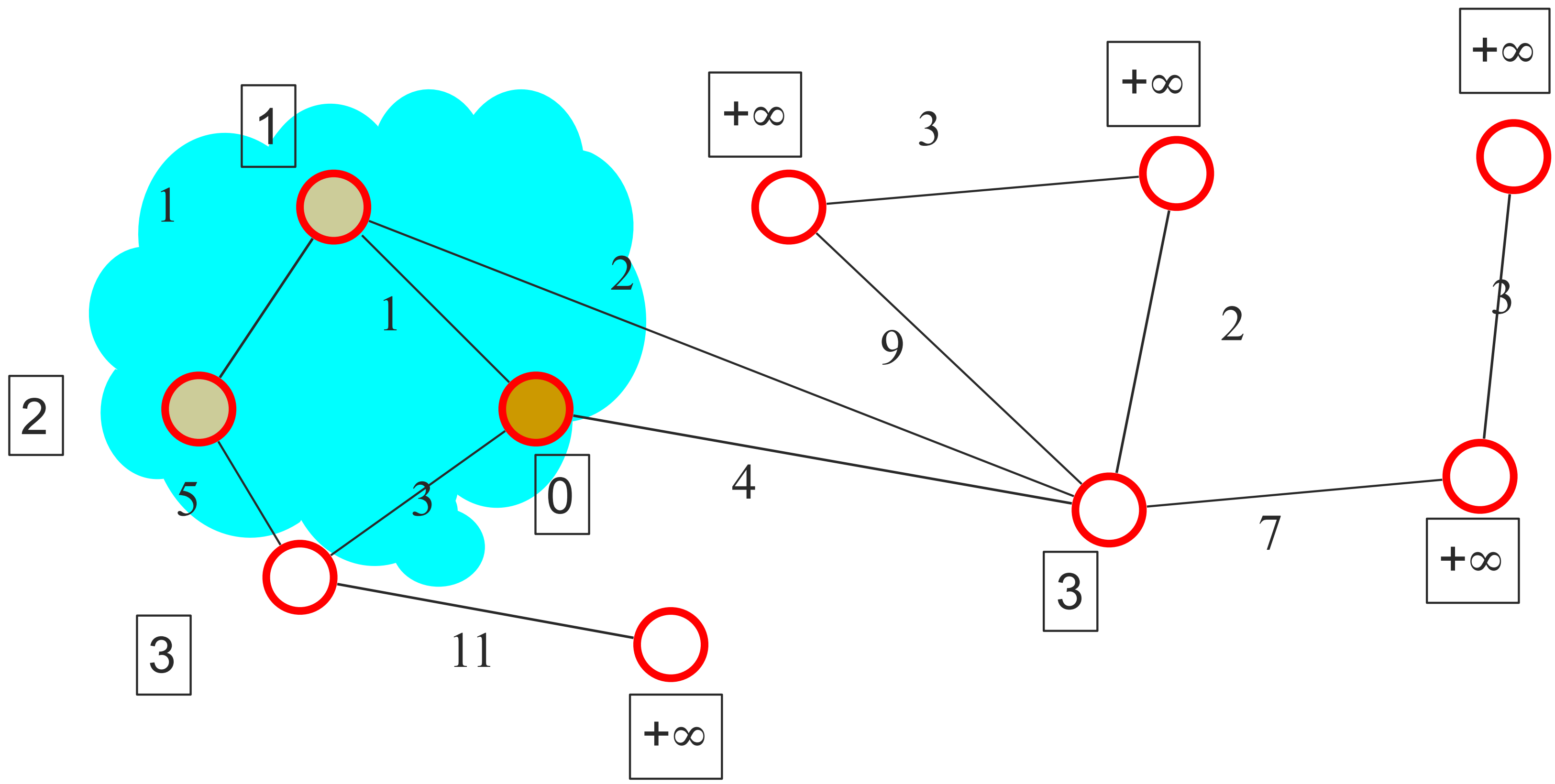
Dijkstra's algorithm: update *C's* neighborhood



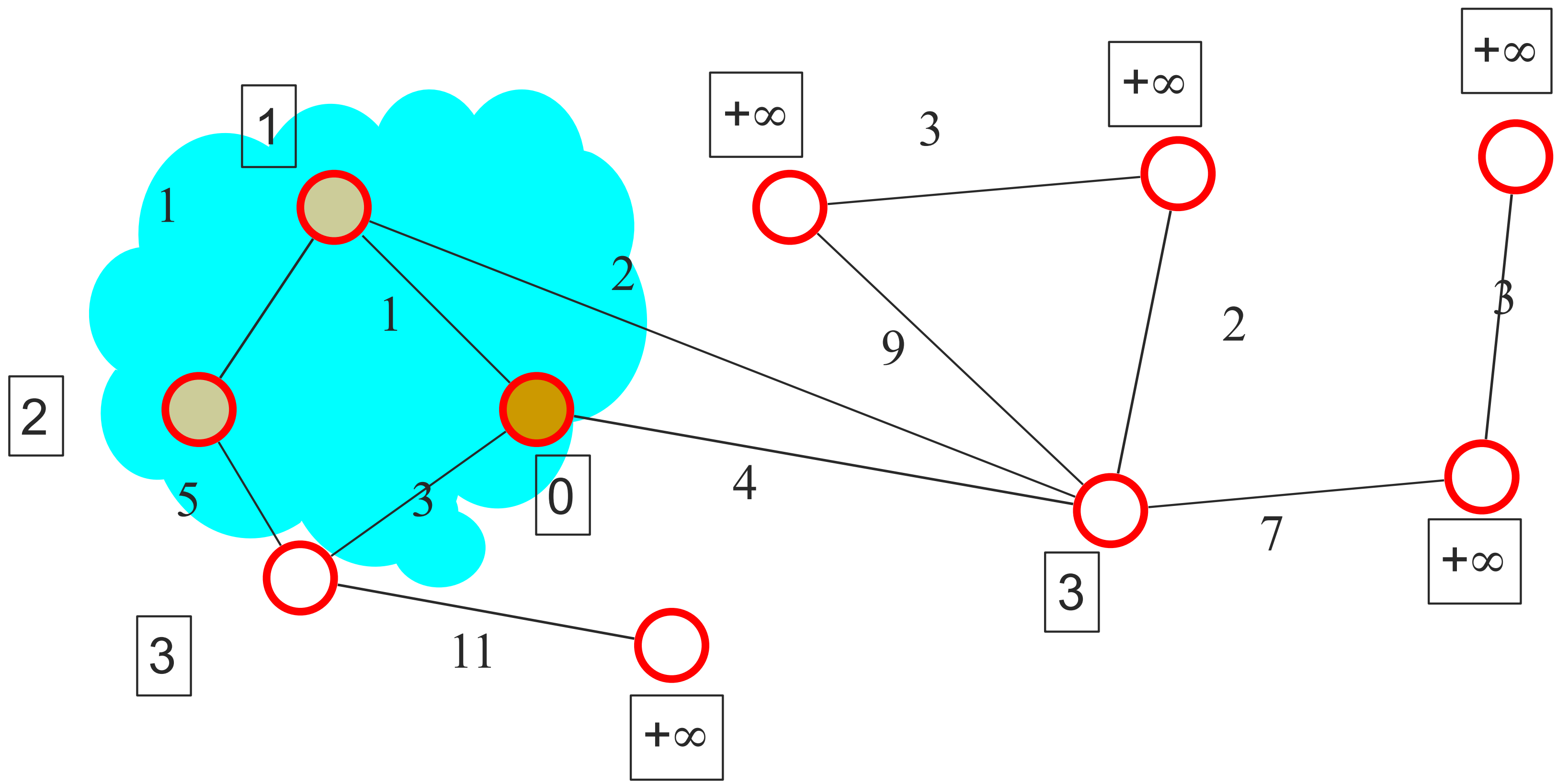
Dijkstra's algorithm: pick closest vertex u outside C



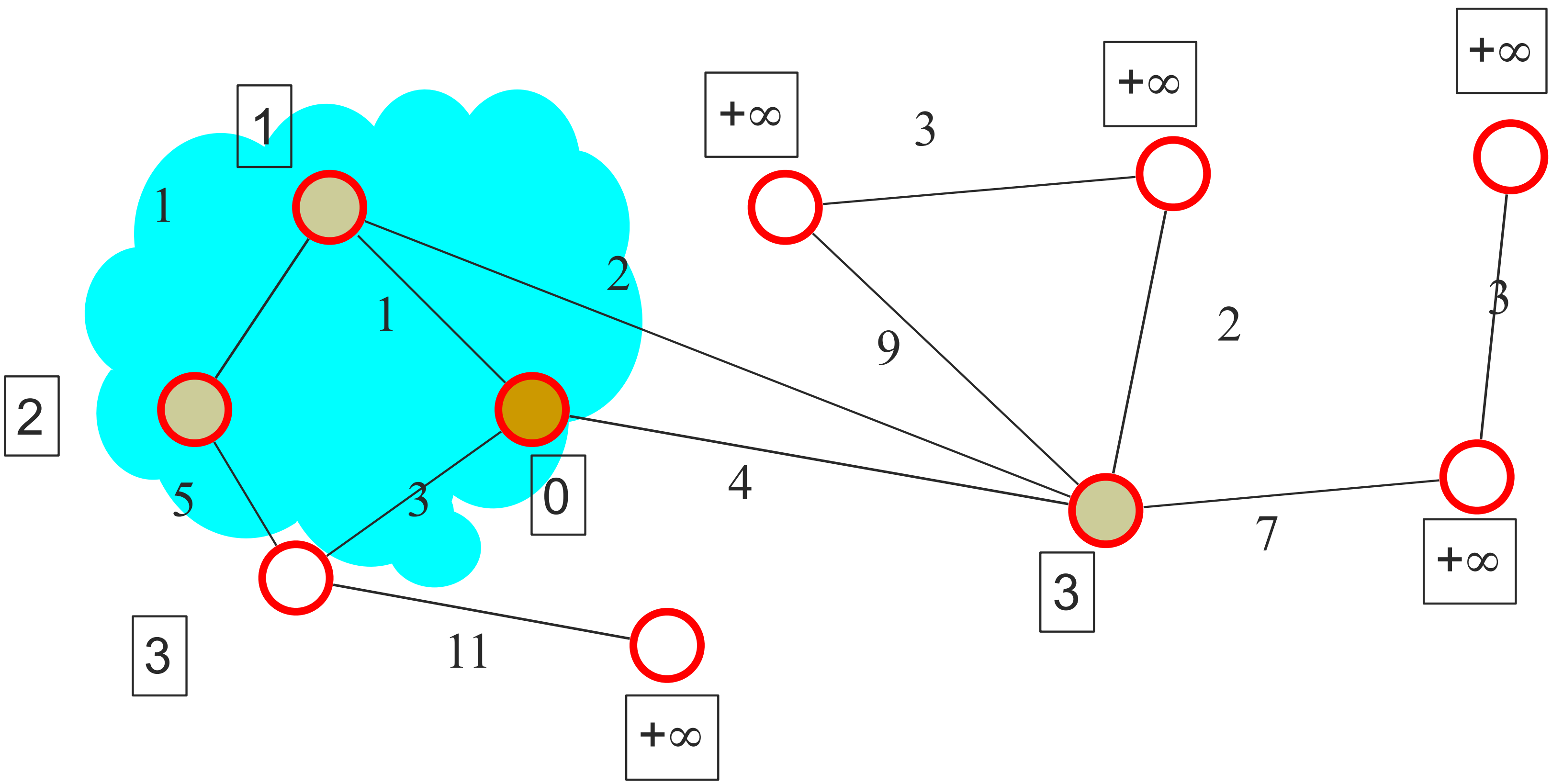
Dijkstra's algorithm: pull u into C



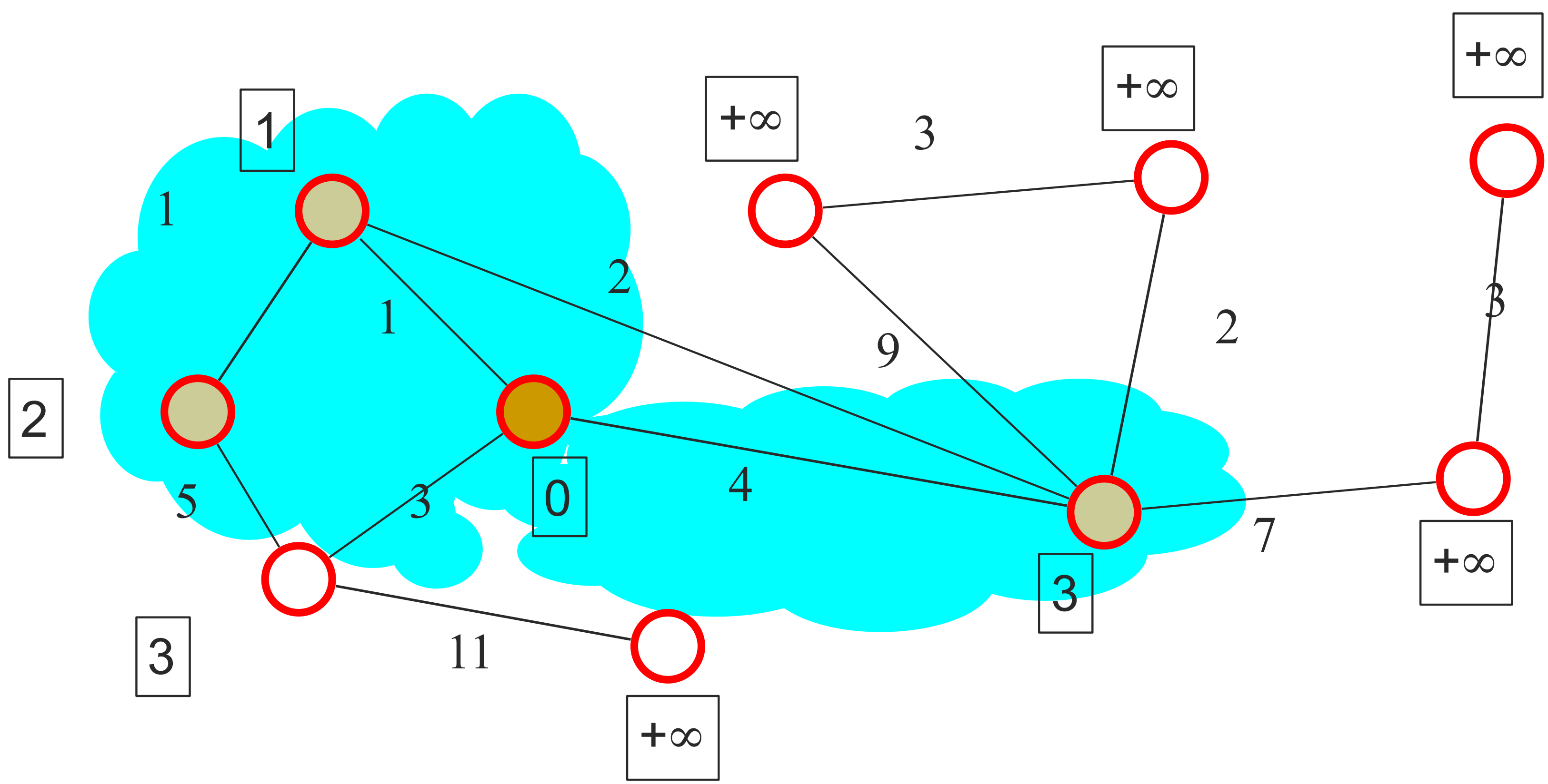
Dijkstra's algorithm: update *C's* neighborhood



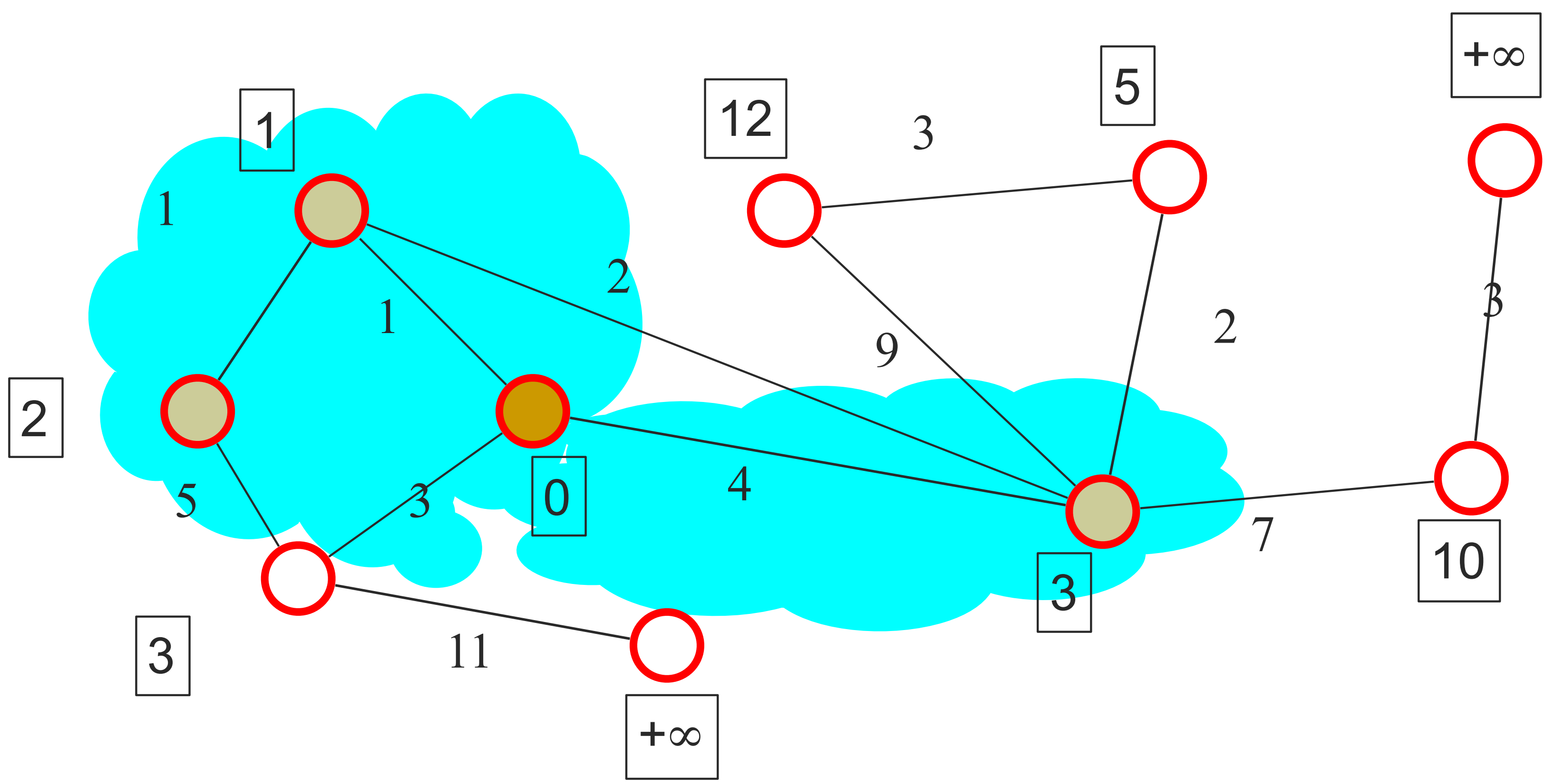
Dijkstra's algorithm: pick closest vertex u outside C



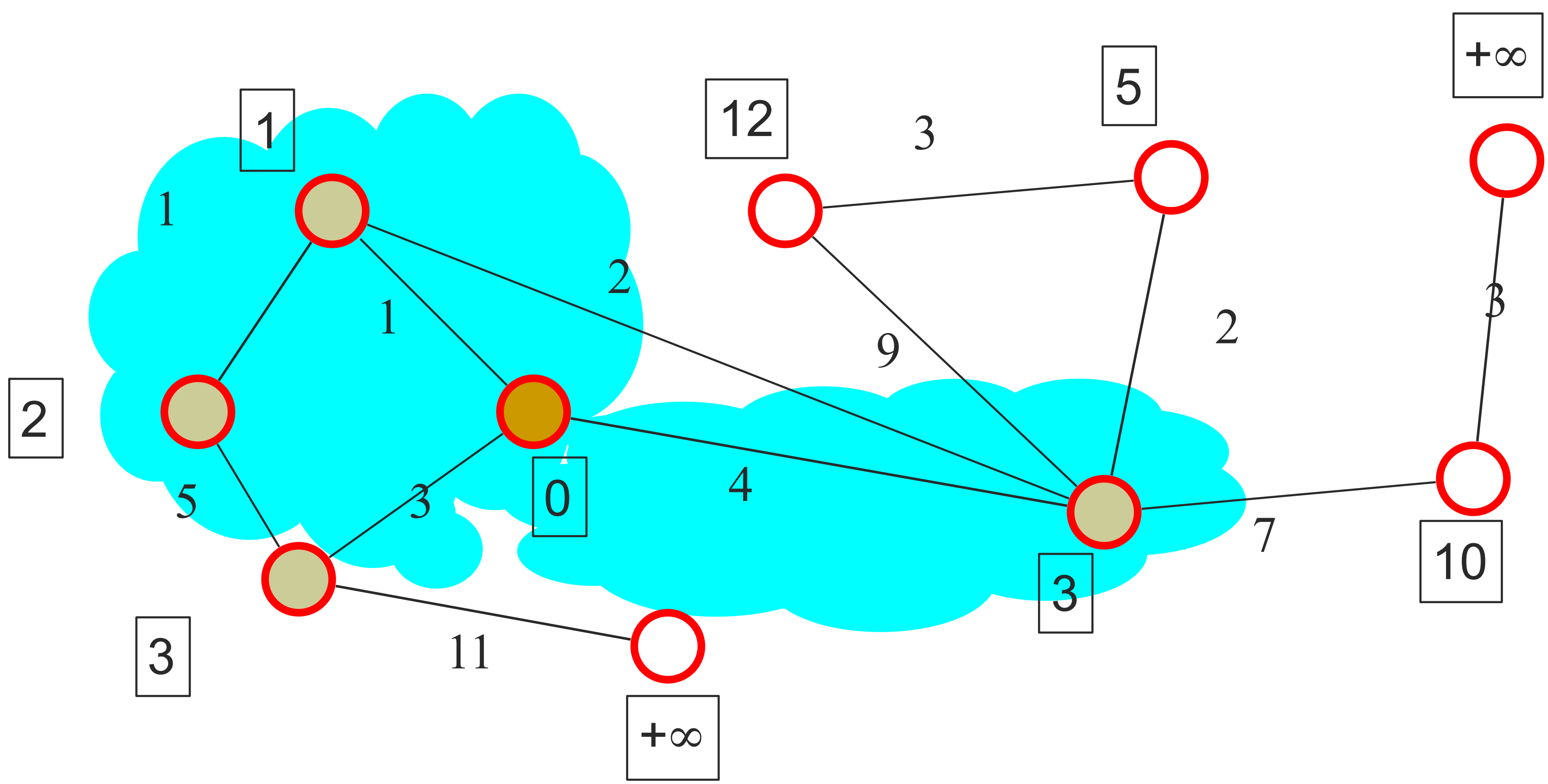
Dijkstra's algorithm: pull u into C



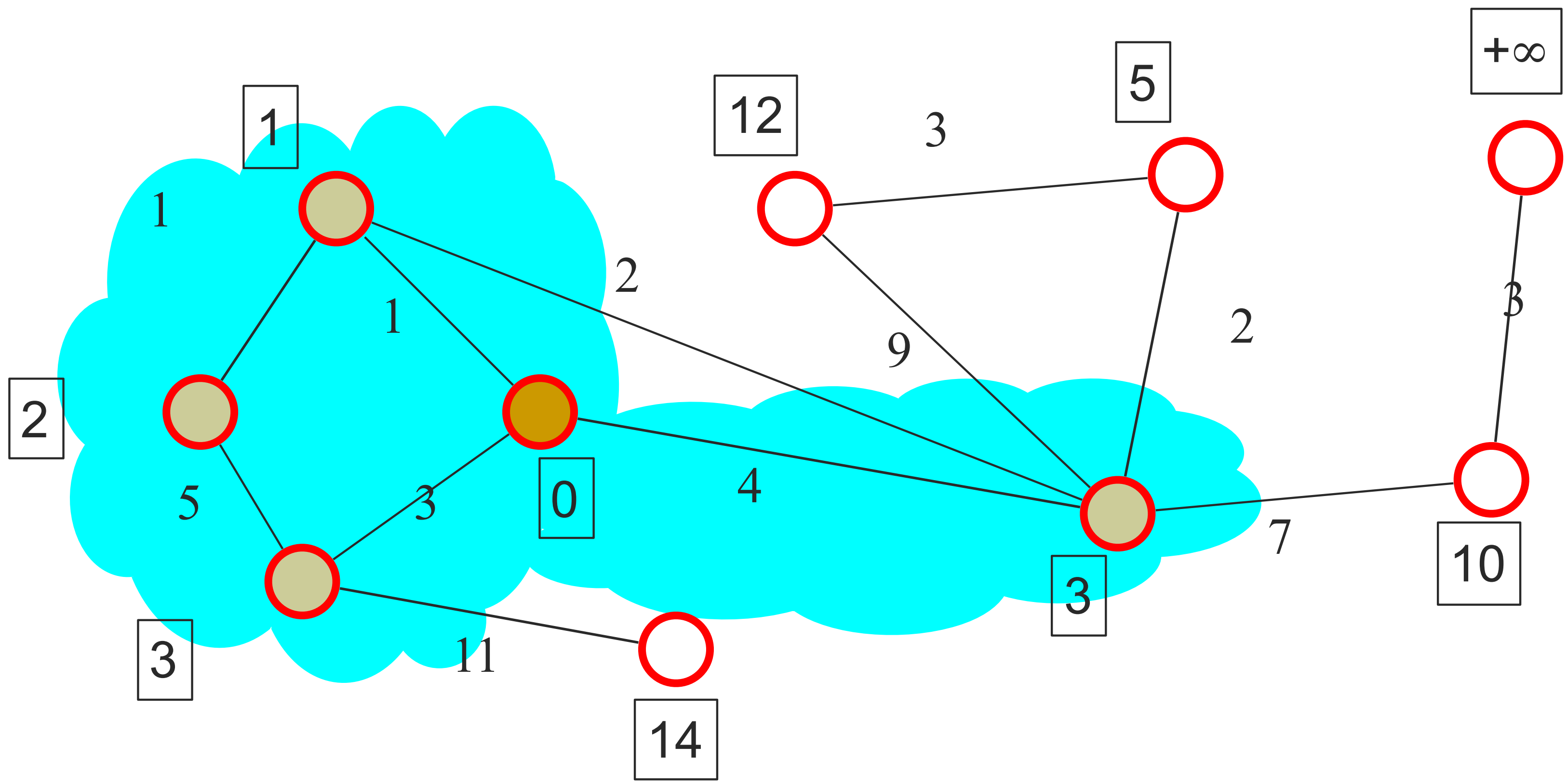
Dijkstra's algorithm: update *C's* neighborhood



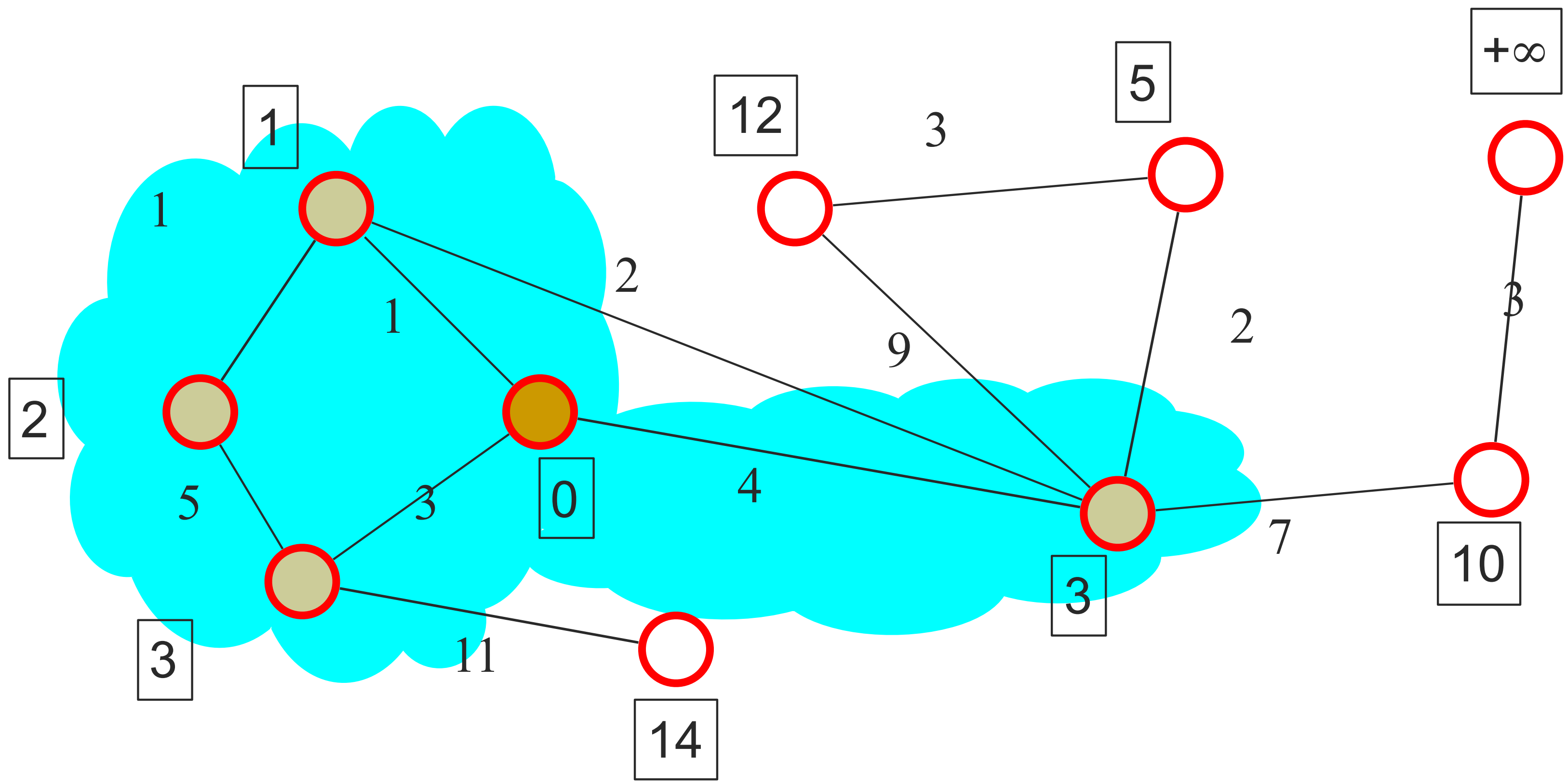
Dijkstra's algorithm: pick closest vertex u outside C



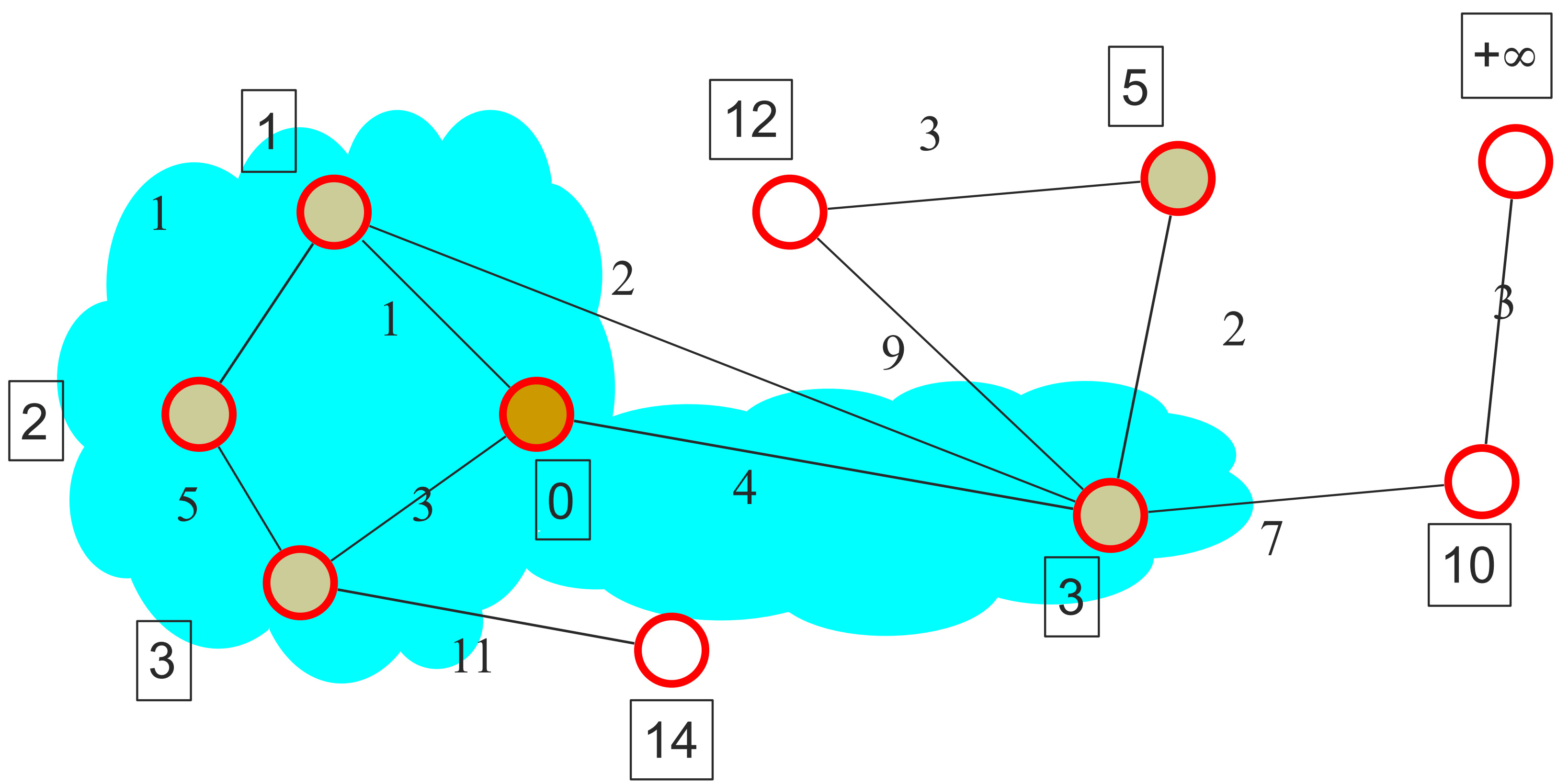
Dijkstra's algorithm: pull u into C



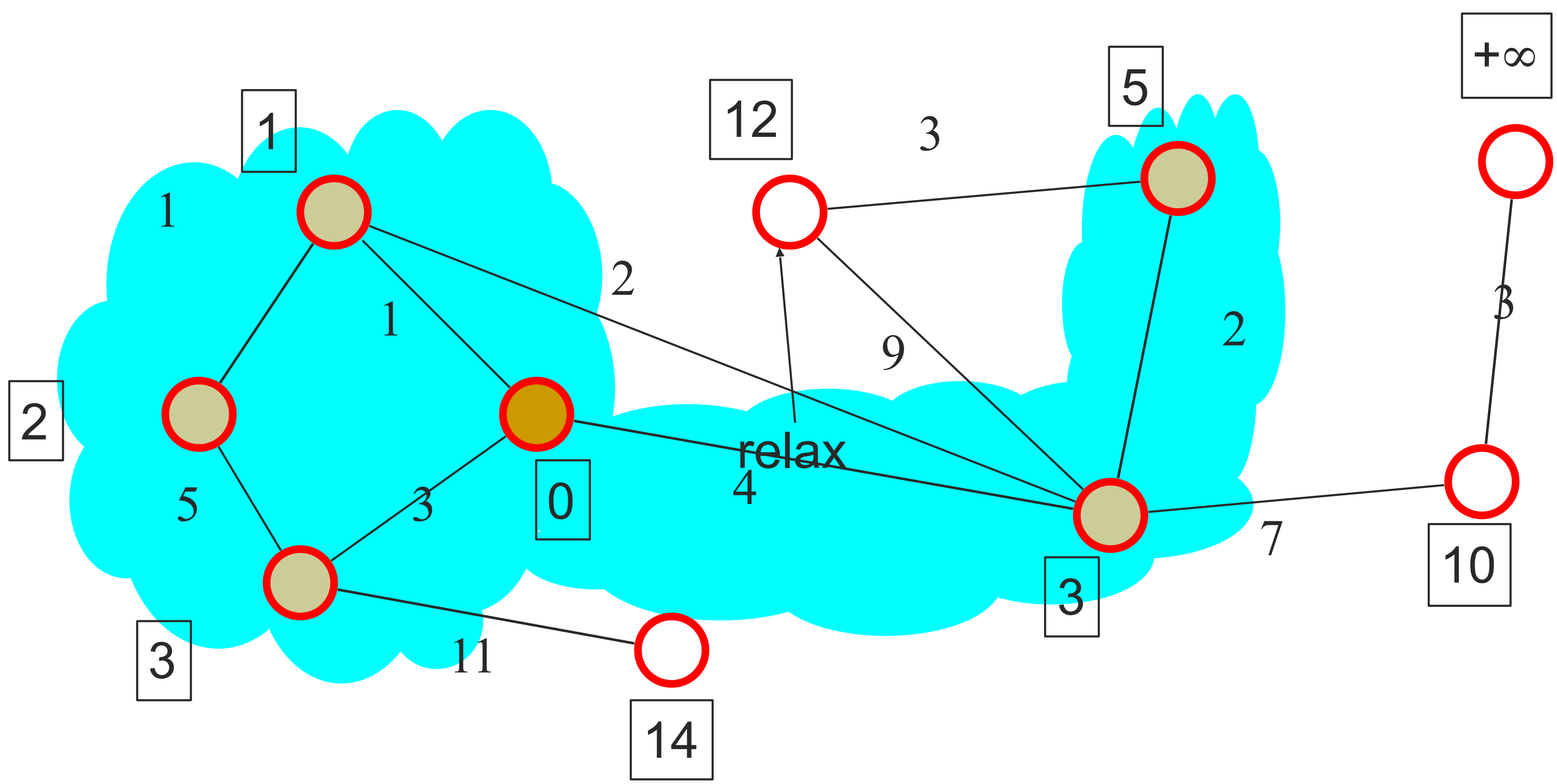
Dijkstra's algorithm: update *C's* neighborhood



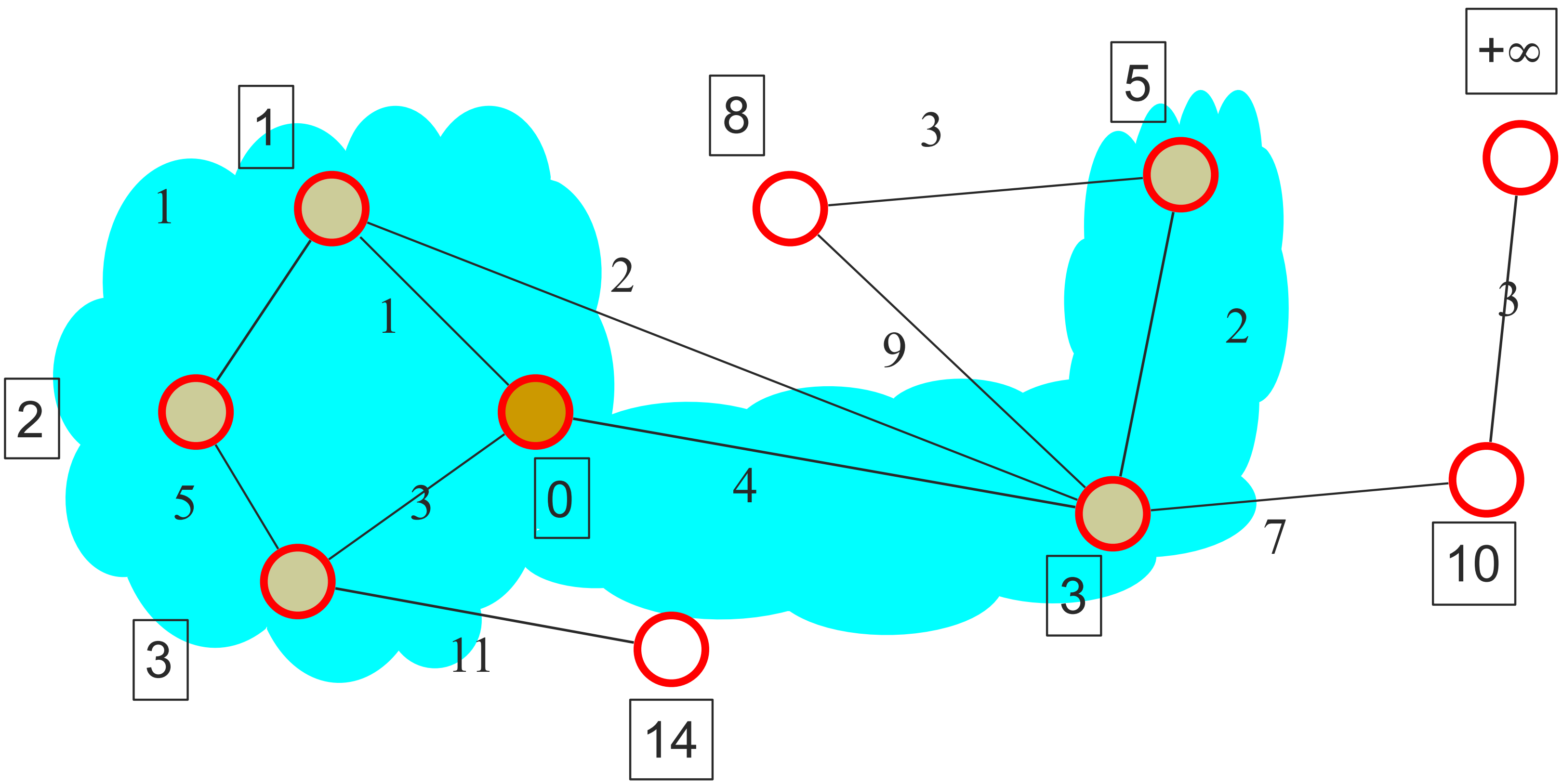
Dijkstra's algorithm: pick closest vertex u outside C



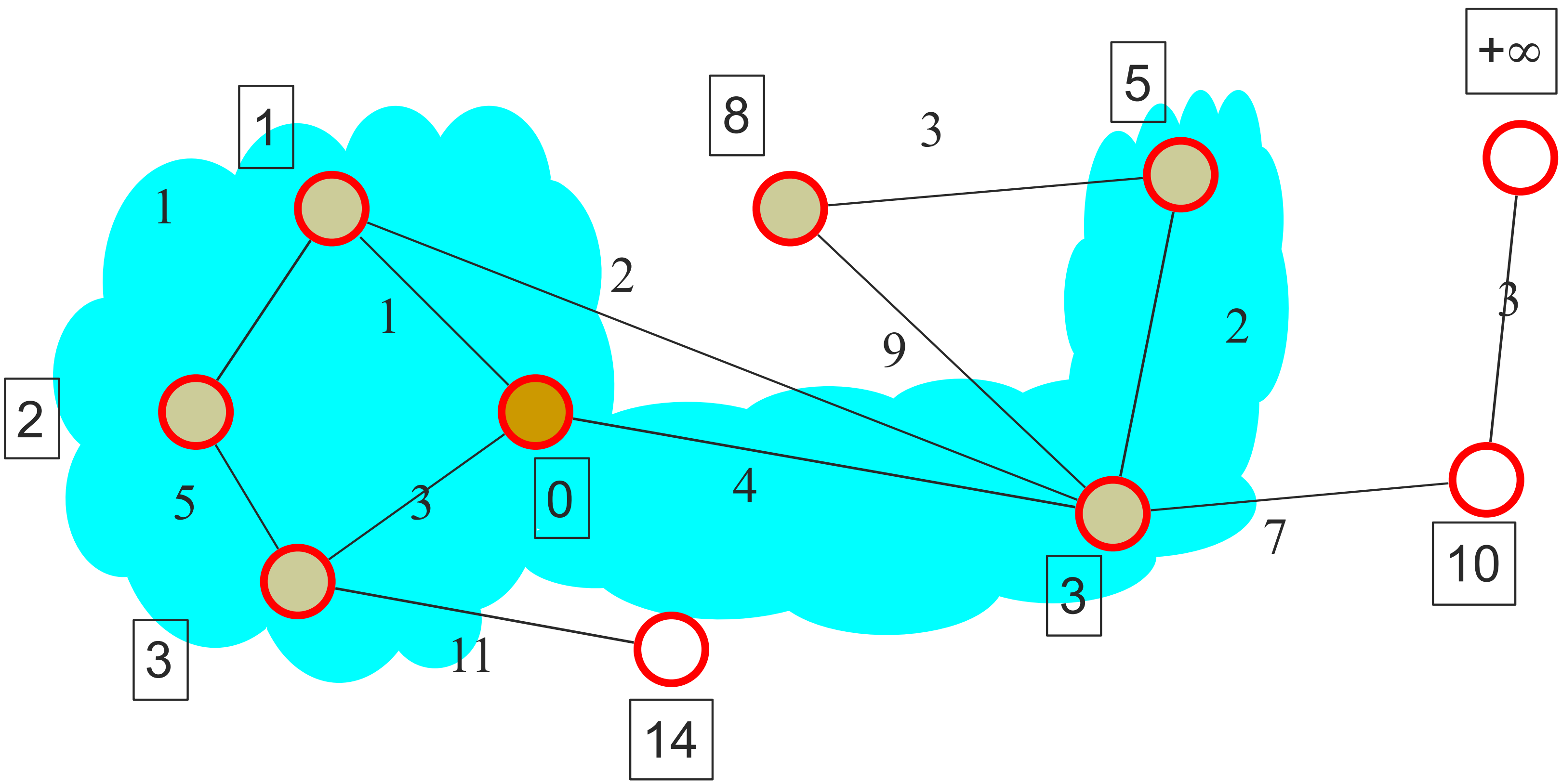
Dijkstra's algorithm: pull u into C



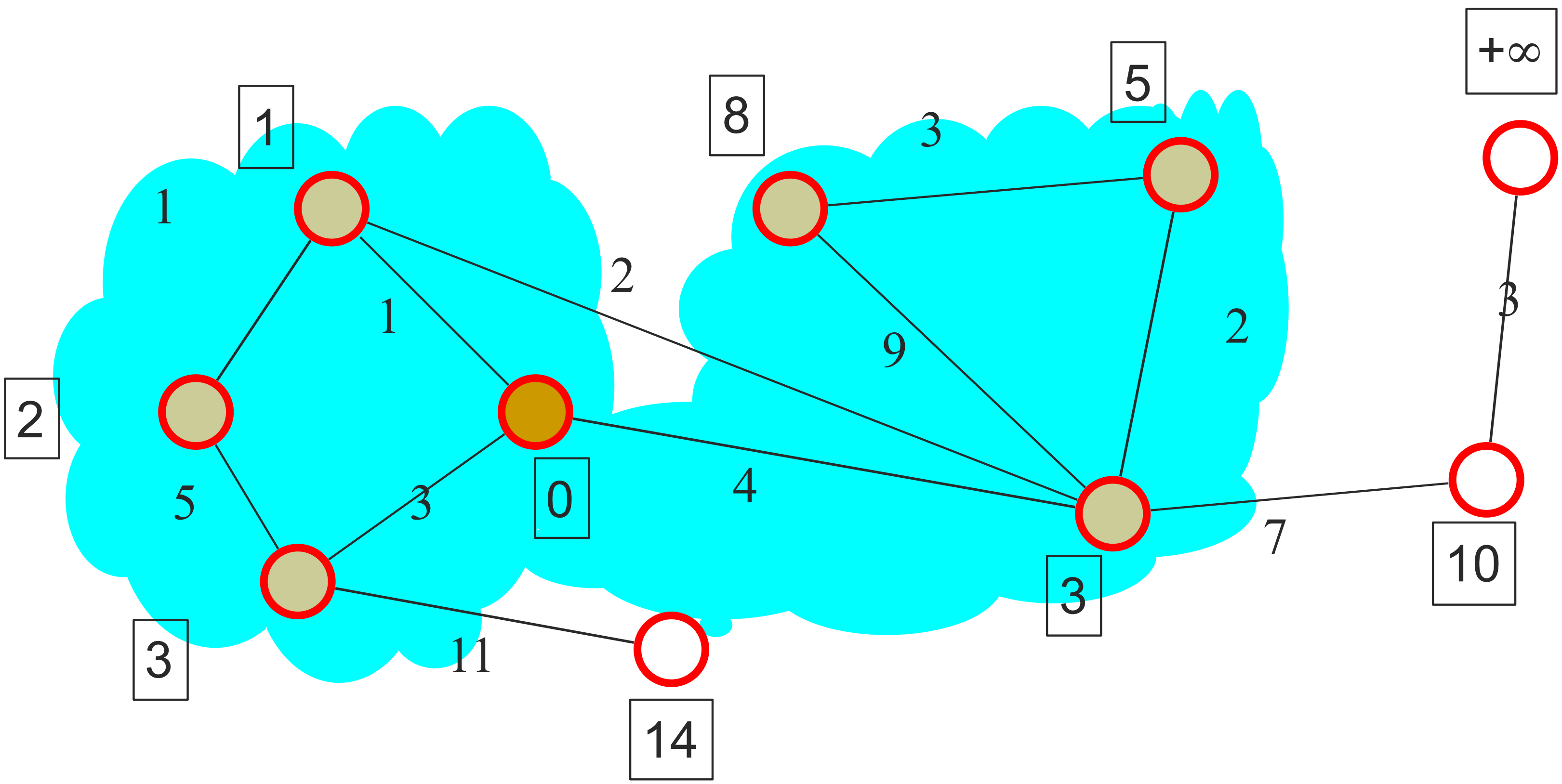
Dijkstra's algorithm: update *C's* neighborhood



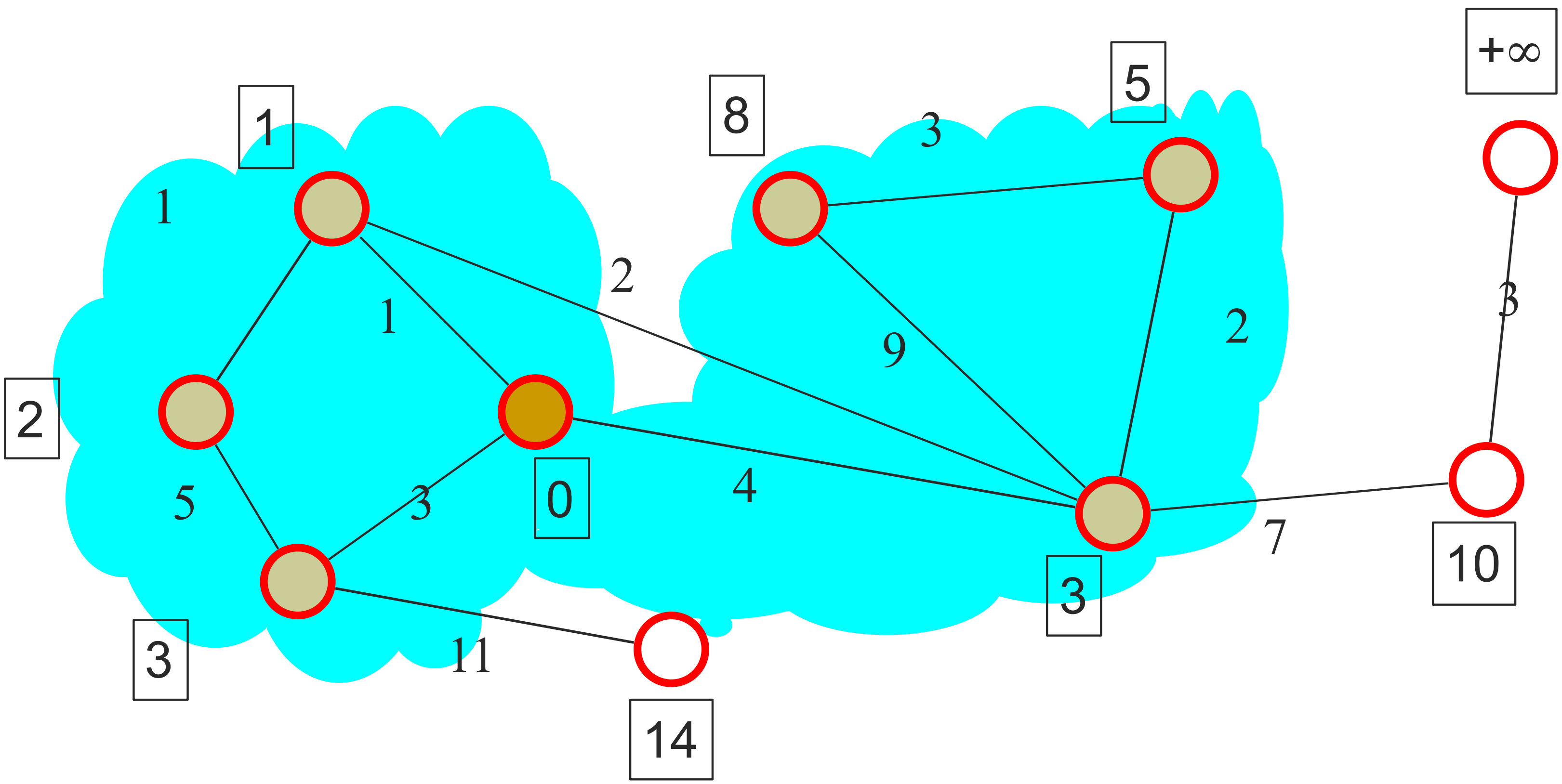
Dijkstra's algorithm: pick closest vertex u outside C



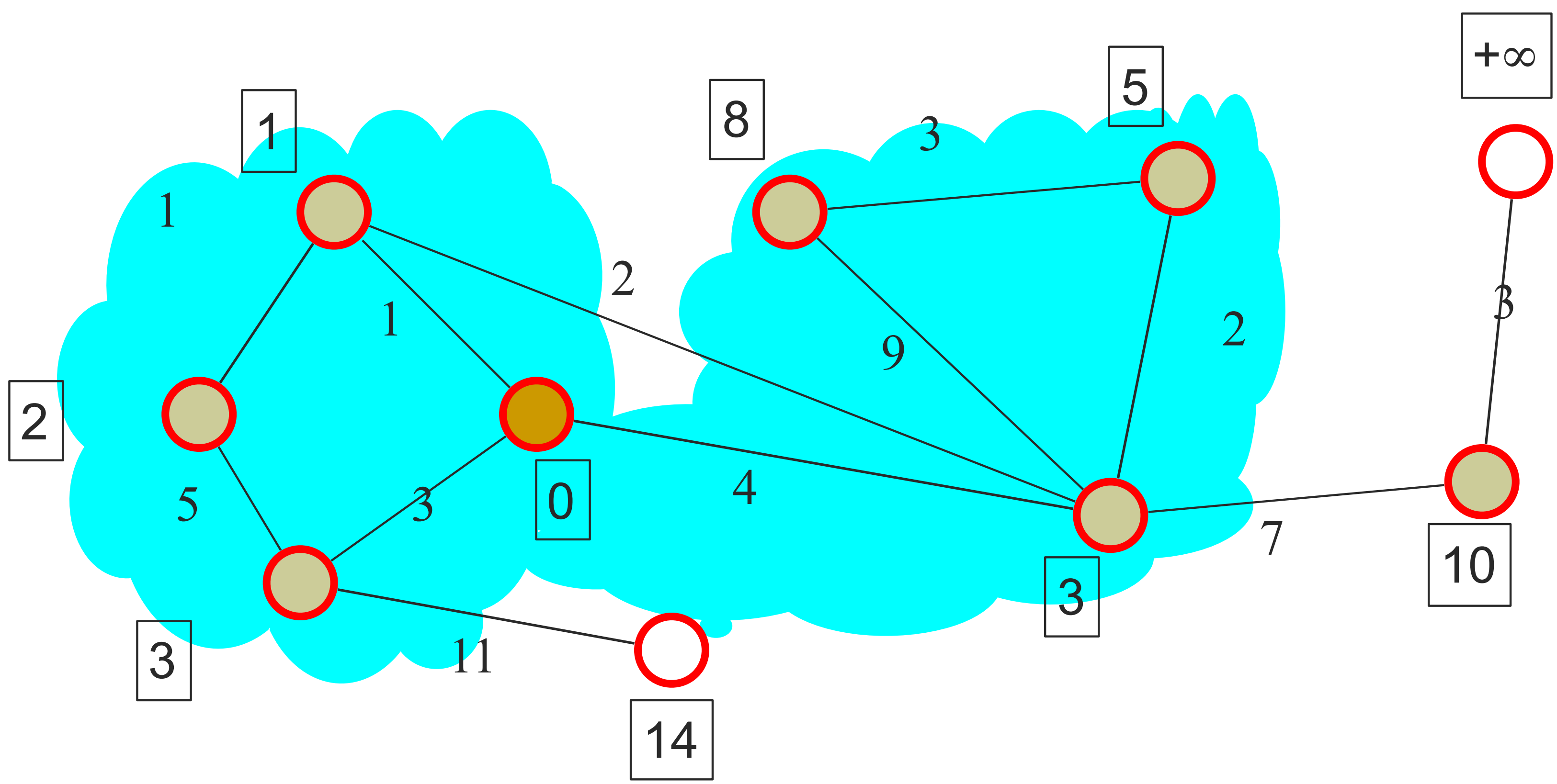
Dijkstra's algorithm: pull u into C



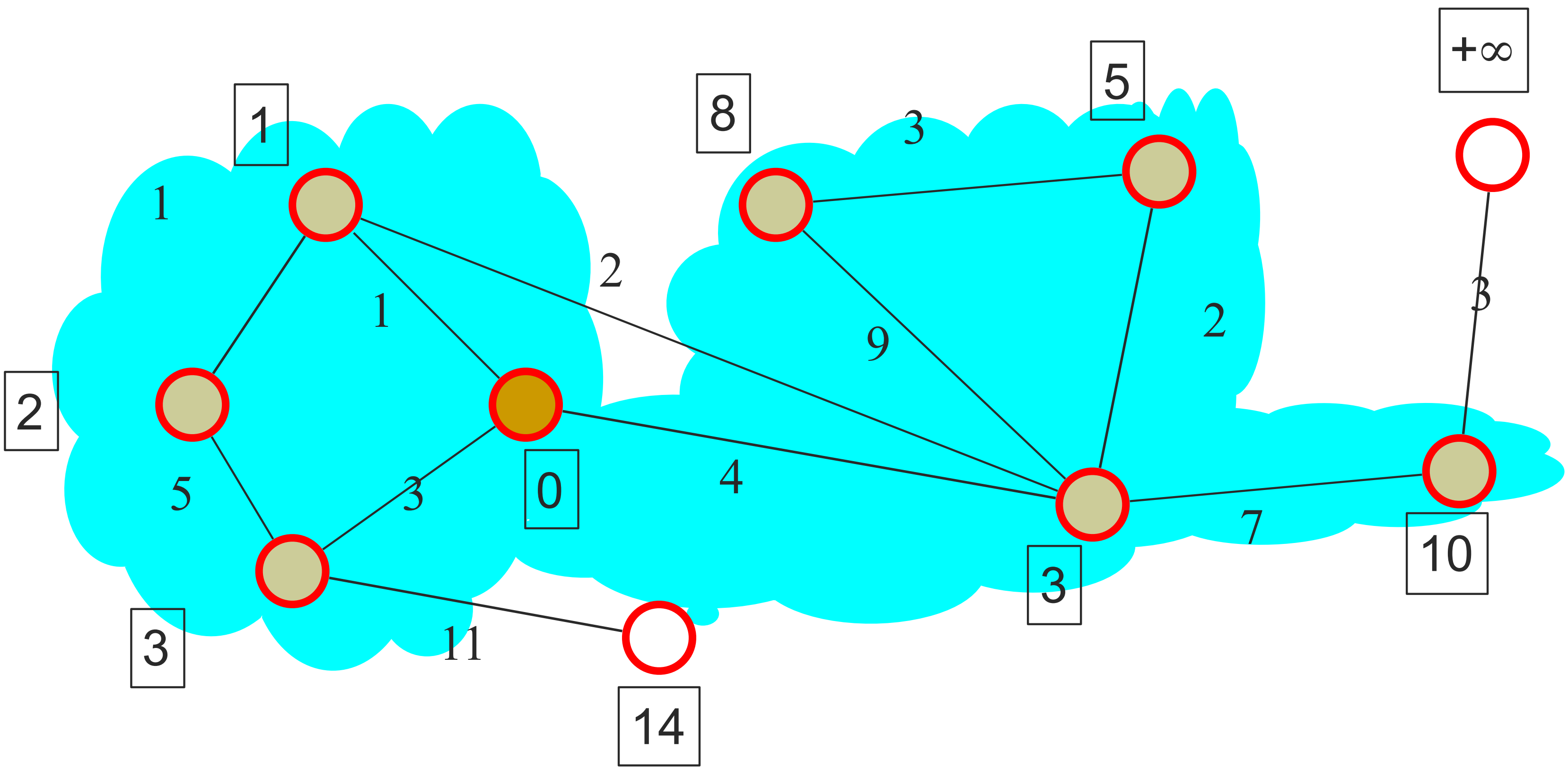
Dijkstra's algorithm: update *C's* neighborhood



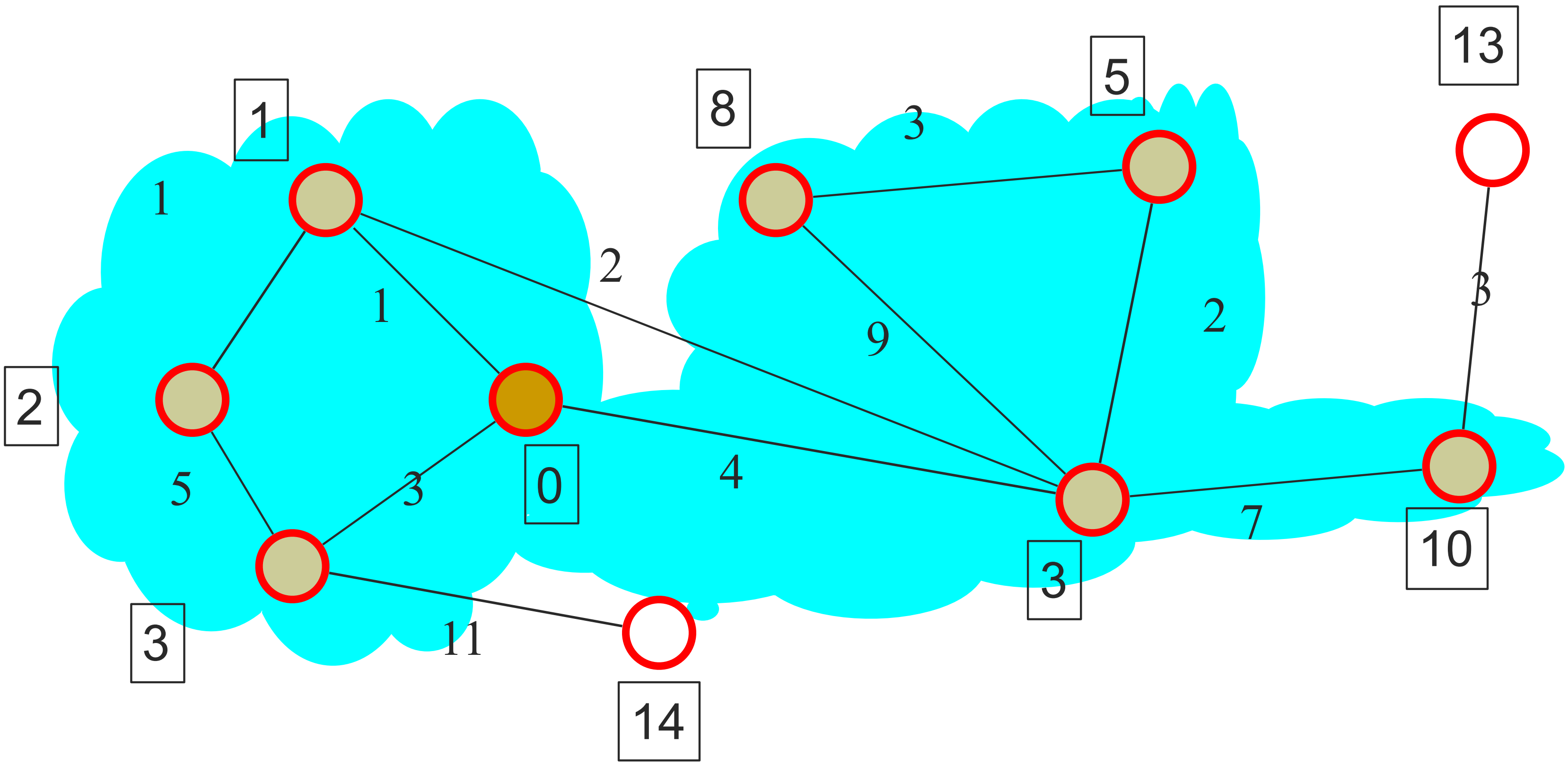
Dijkstra's algorithm: pick closest vertex u outside C



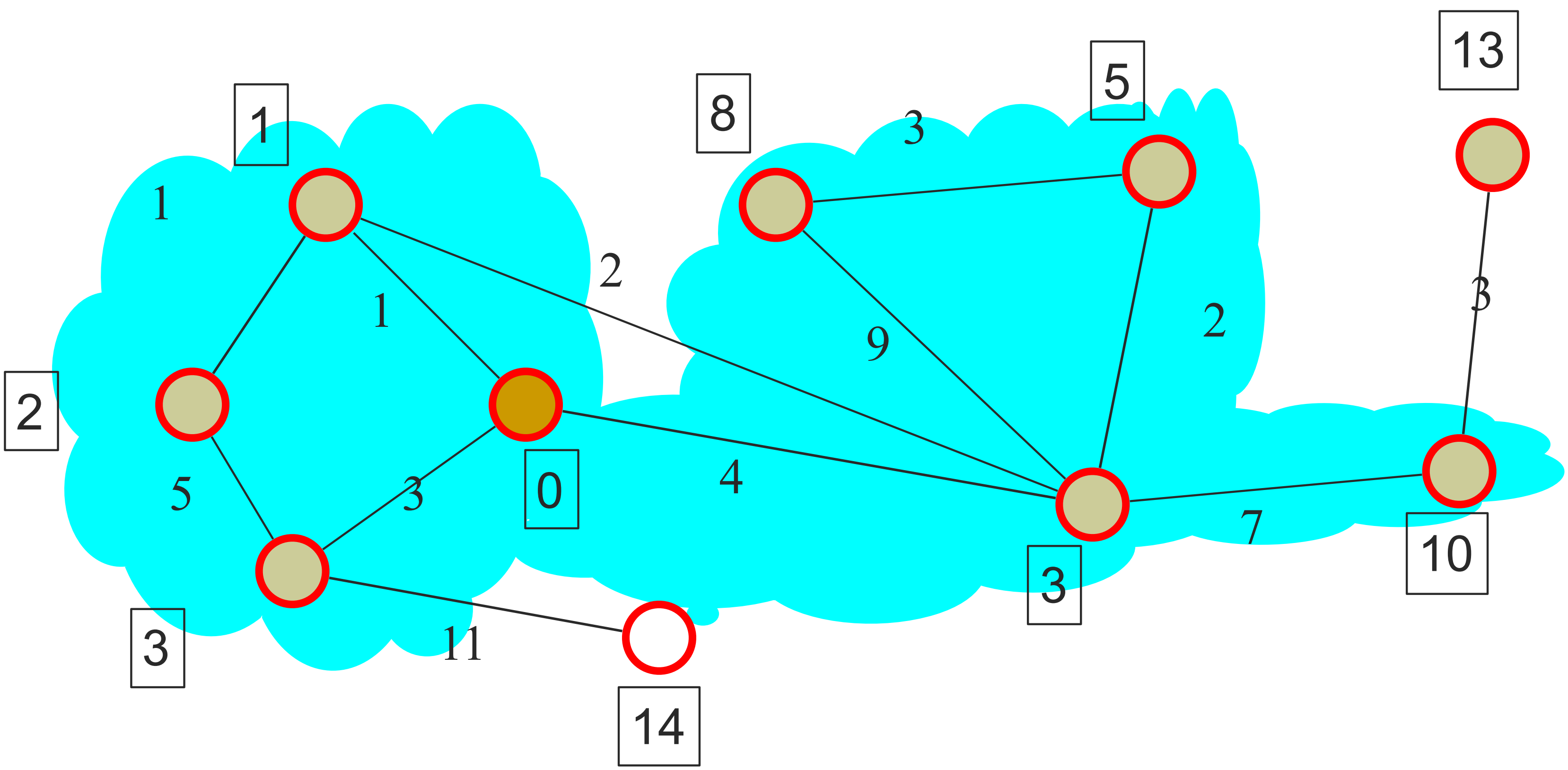
Dijkstra's algorithm: pull u into C



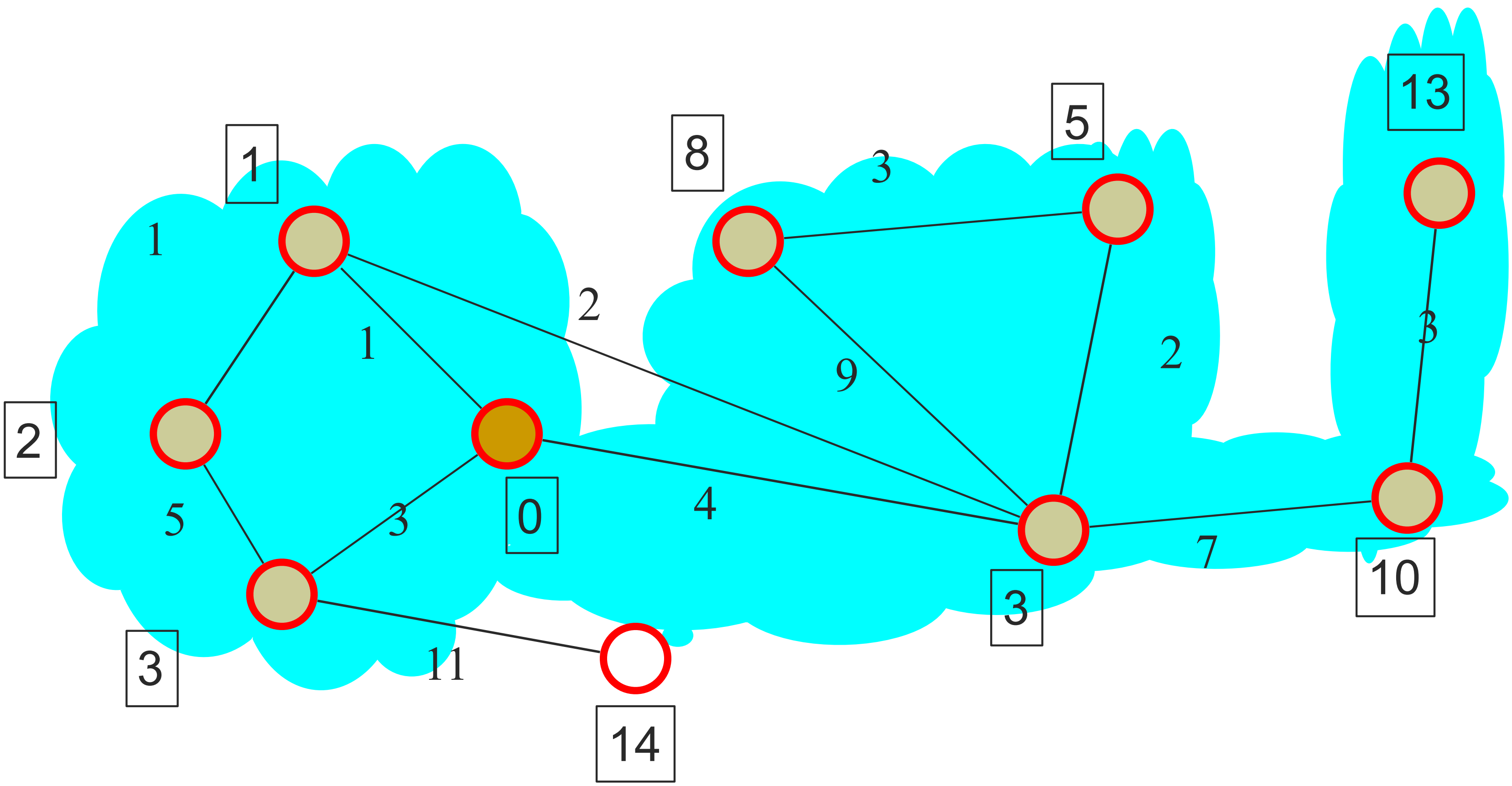
Dijkstra's algorithm: update *C's* neighborhood



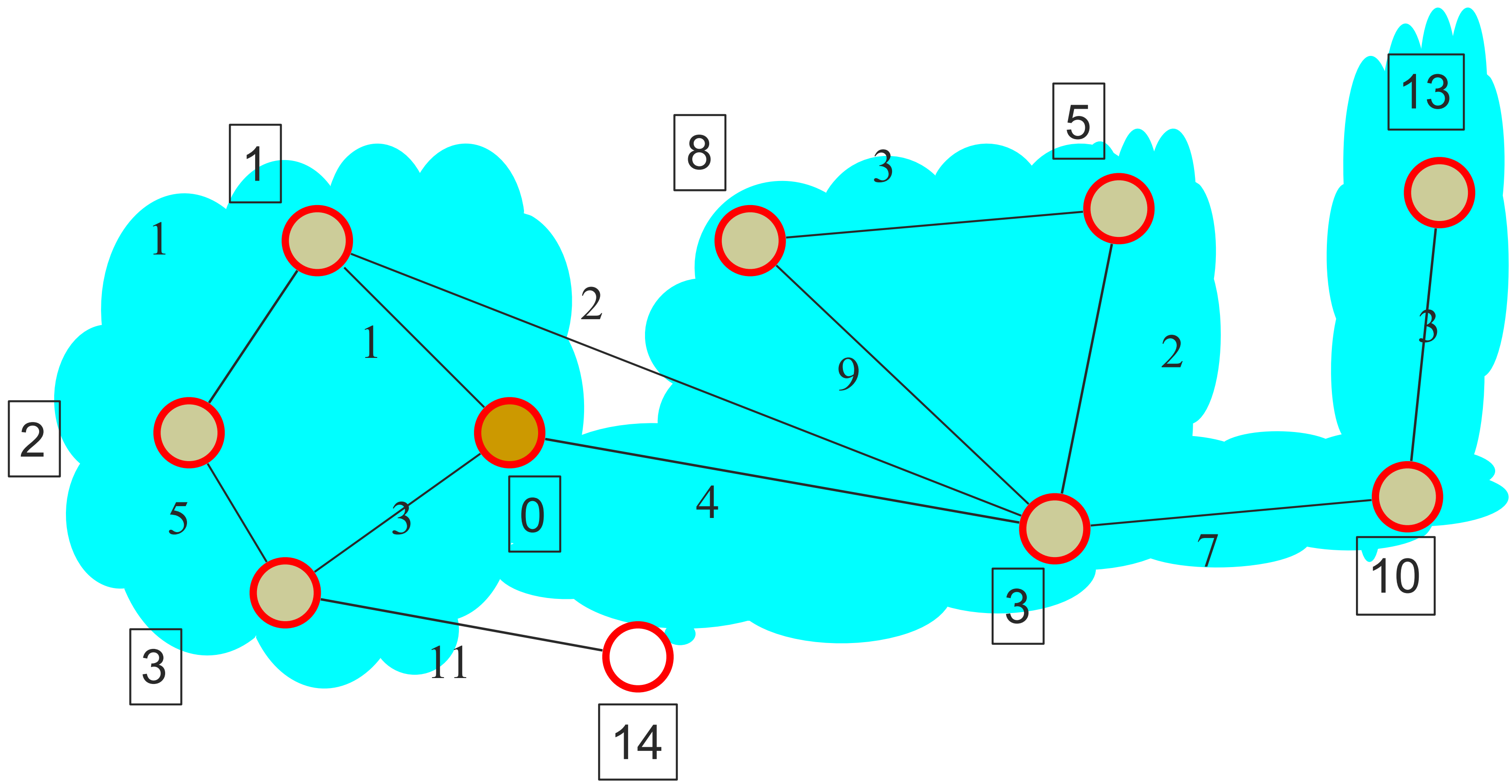
Dijkstra's algorithm: pick closest vertex u outside C



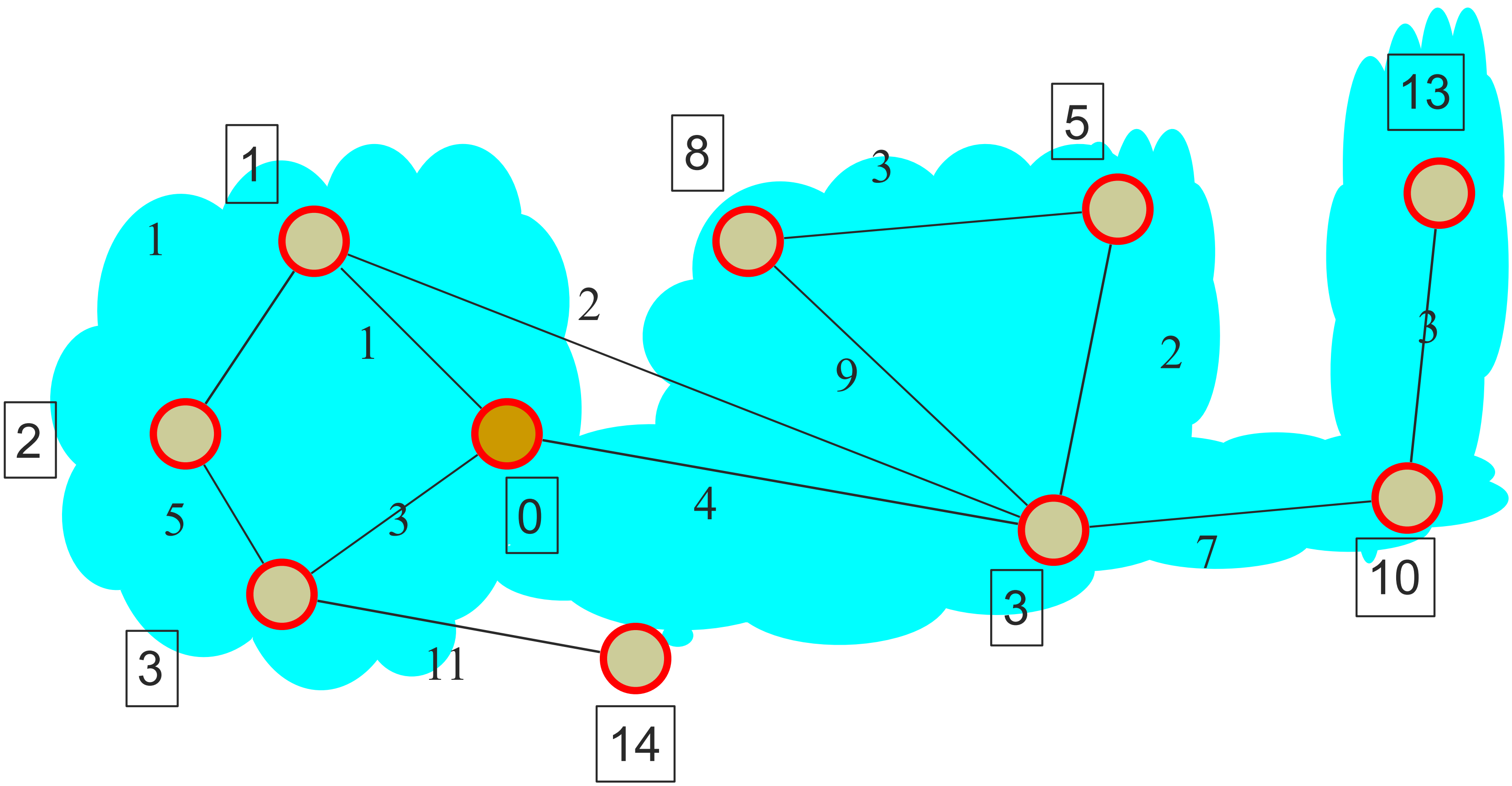
Dijkstra's algorithm: pull u into C



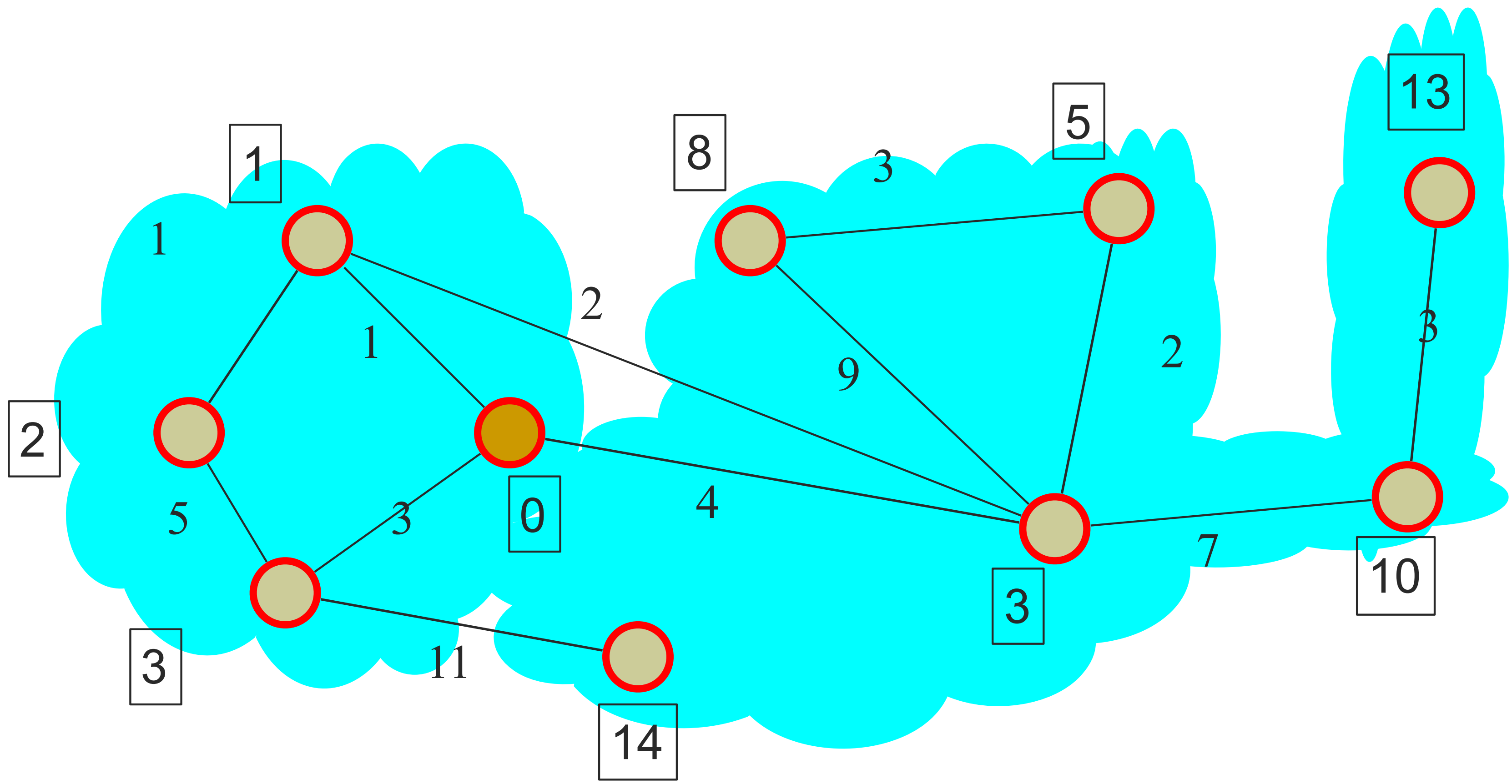
Dijkstra's algorithm: update *C's* neighborhood



Dijkstra's algorithm: pick closest vertex u outside C



Dijkstra's algorithm: pull u into C



When pulling a neighbour u of C into C

- The value associated with u denotes the length of a shortest path from v to u
- For any vertex x not in the cloud
  - the value associated with x denotes a shortest way from v to x without the use of other vertices outside of the cloud
  - $+\infty$ denotes that the vertex cannot be reached yet from x via cloud vertices only

Algorithm DijkstraShortestPaths(G,v)

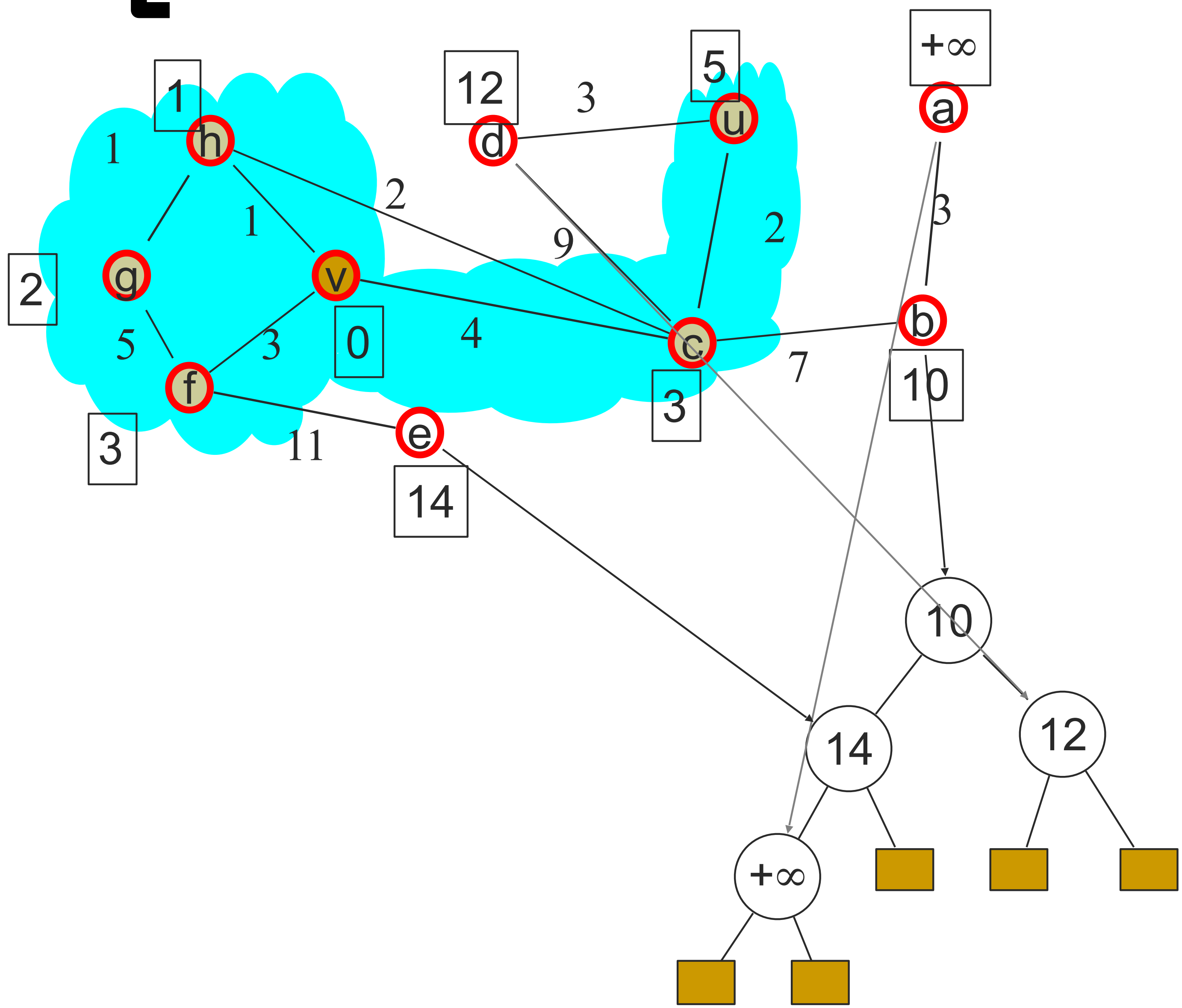
Input: A simple undirected graph G with nonnegative edge-weights, a distinguished vertex v in G

Output: A label D[u] for each vertex u in G such that D[u] is the distance from v to u in G.

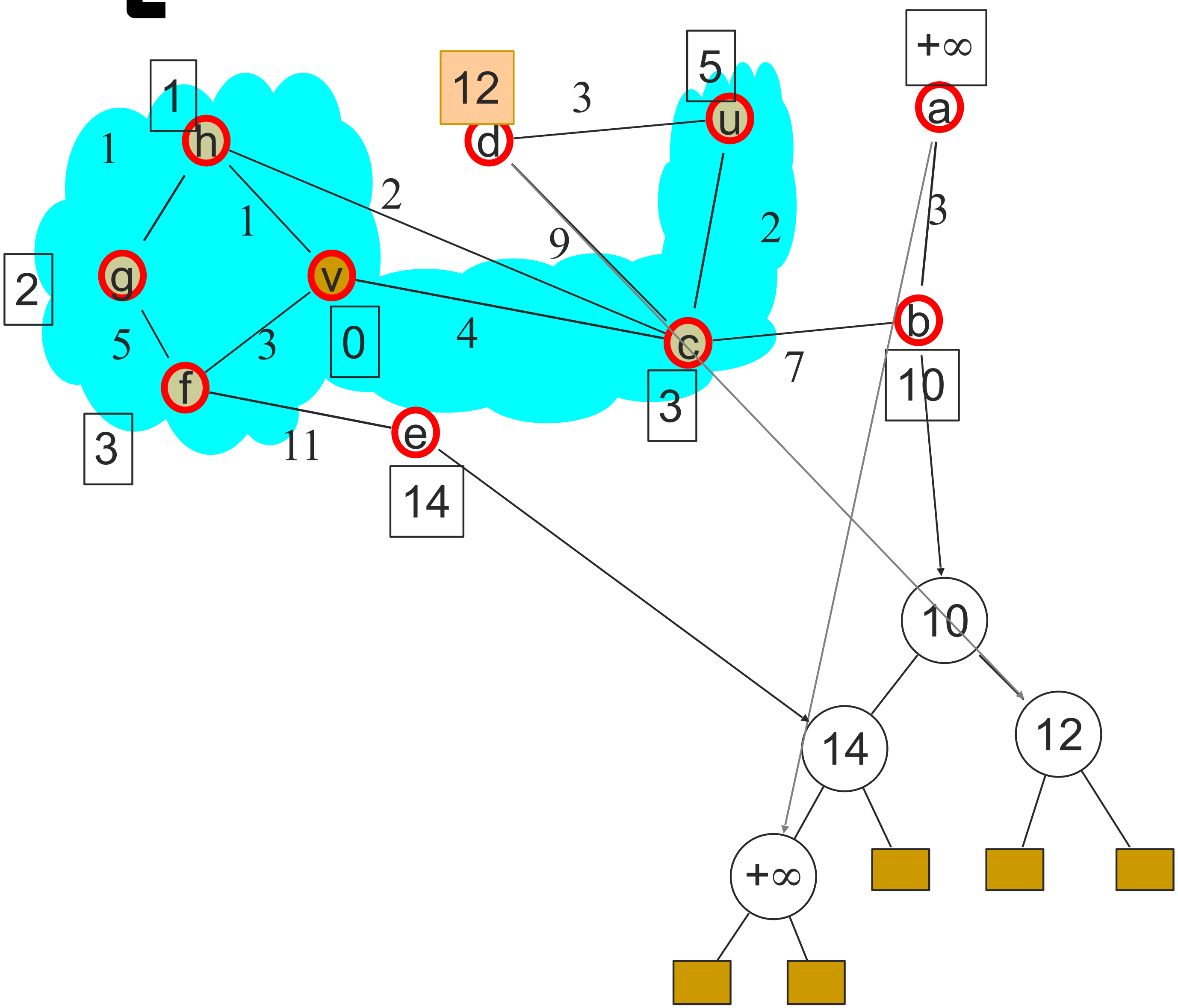
Algorithm DijkstraShortestPaths(G,v)

```
D[v]←0
for each vertex u≠v of G do
   D[u]←+∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
Relaxation   D[z]←D[u]+w((u,z))
             update z's key in Q to D[z]
return D
```

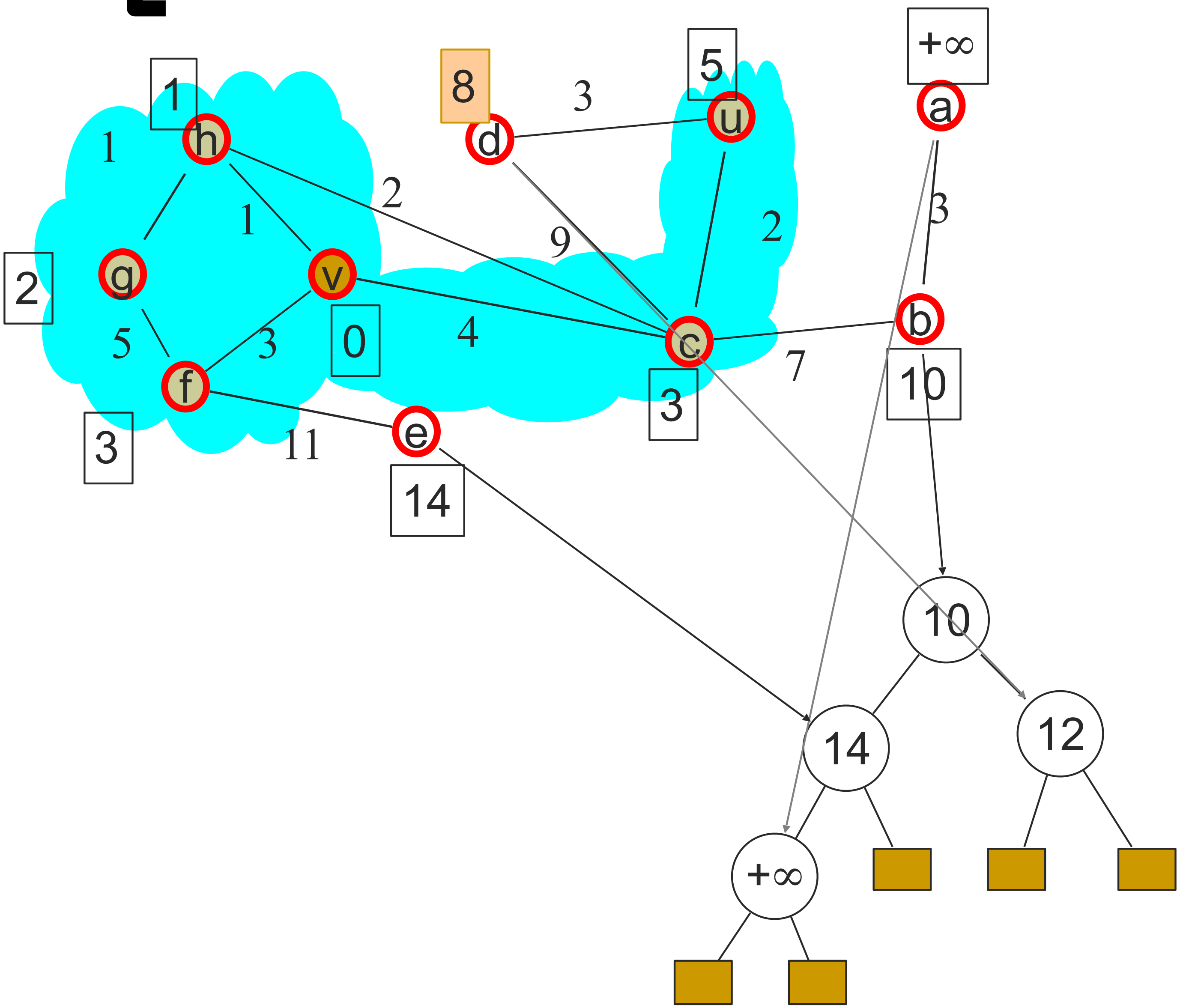
Dijkstra’s algorithm using heaps



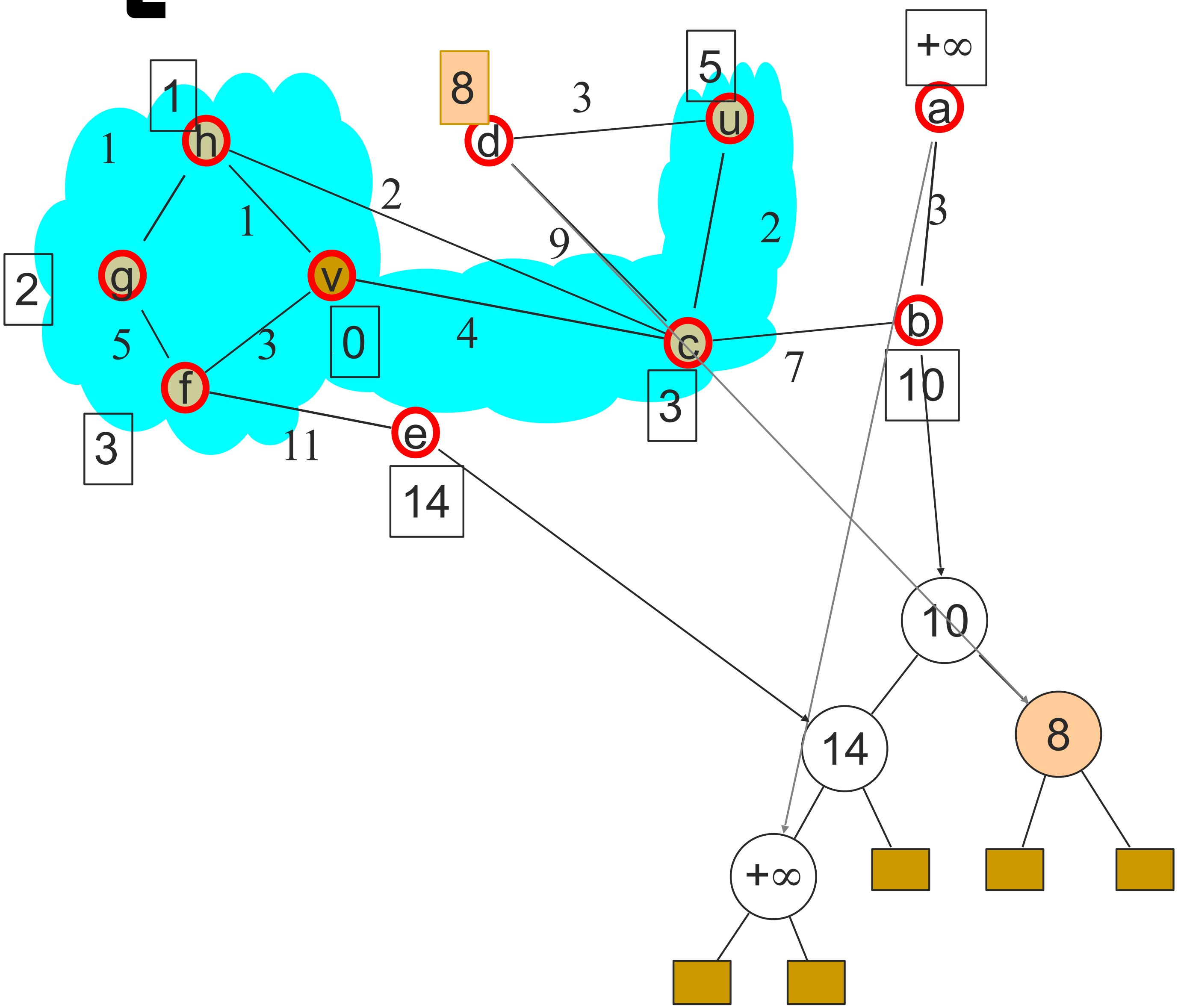
Dijkstra’s algorithm using heaps



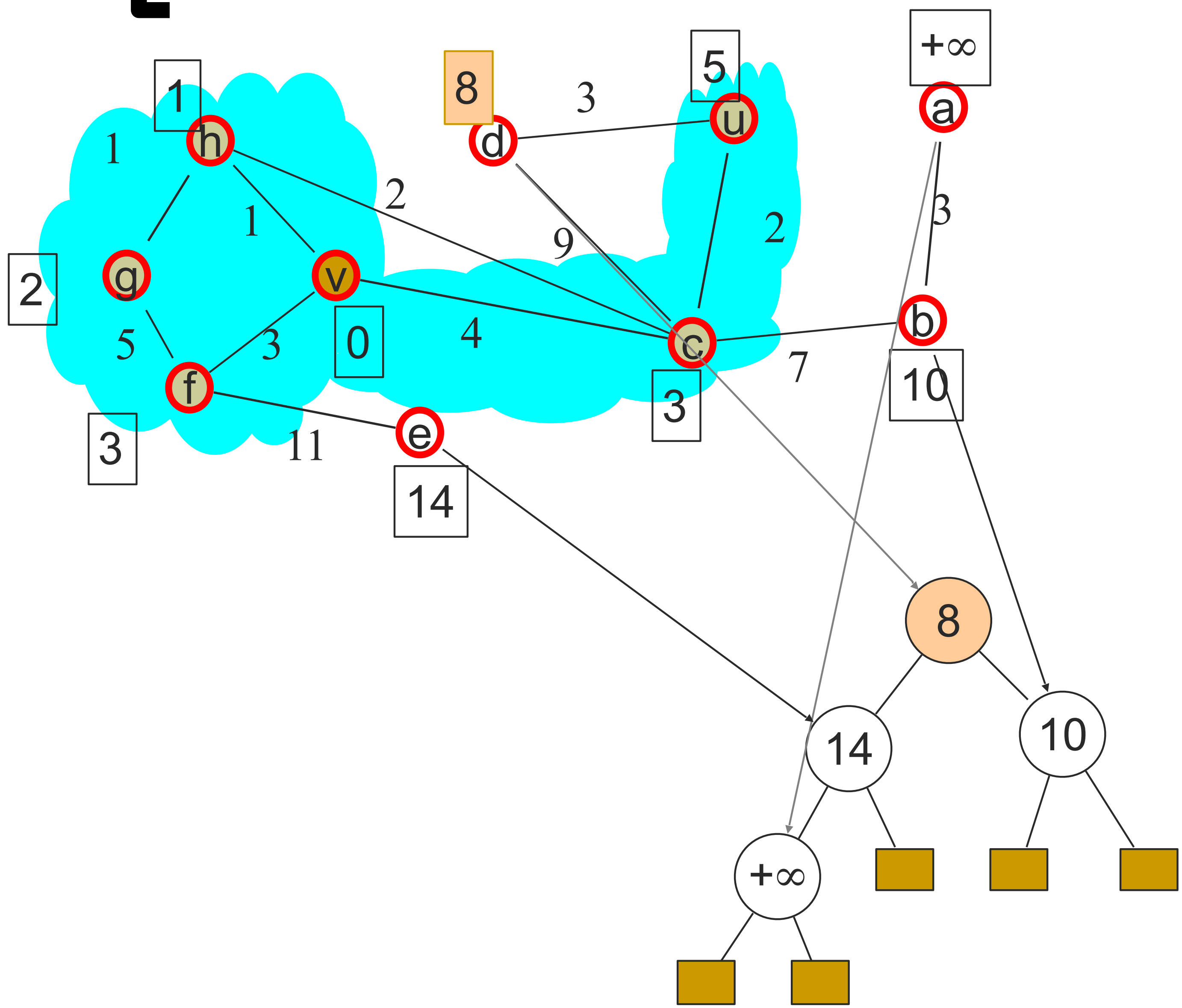
Dijkstra's algorithm using heaps



Dijkstra's algorithm using heaps



Dijkstra's algorithm using heaps



Running time

```
D[v]←0
for each vertex u≠v of G do
   D[u]←+∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
         D[z]←D[u]+w((u,z))
         update z's key in Q to D[z]
return D
```

Relaxation

Running time for $G=(V,E)$ with $|V|=n$ and $|E|=m$

- Insertion of vertices in priority queue Q
  - O(n) when using heap
- While loop:
  - Per iteration:
    - Remove vertex from Q O(log n)
    - Relaxation O(deg(u) log(n))
  - $\sum_{u \in G} (1 + \deg(u)) \log n$ is O((n+m) log n)
- Overall running time: O(m log n)

In real life applications

- Often the graphs are sparse
- Then O(m log n) may be O(n log n)

Correctness

Algorithm DijkstraShortestPaths(G,v)

```
D[v]←0
for each vertex u≠v of G do
   D[u]←+∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
Relaxation   D[z]←D[u]+w((u,z))
             update z's key in Q to D[z]
return D
```

Correctness of Dijkstra's algorithm

- To show: whenever u is pulled into cloud C, D[u] stores the length from a shortest path from u to v

- Definition: For vertices u and v in G, we denote with d(u,v) the length of a shortest path from u to v.

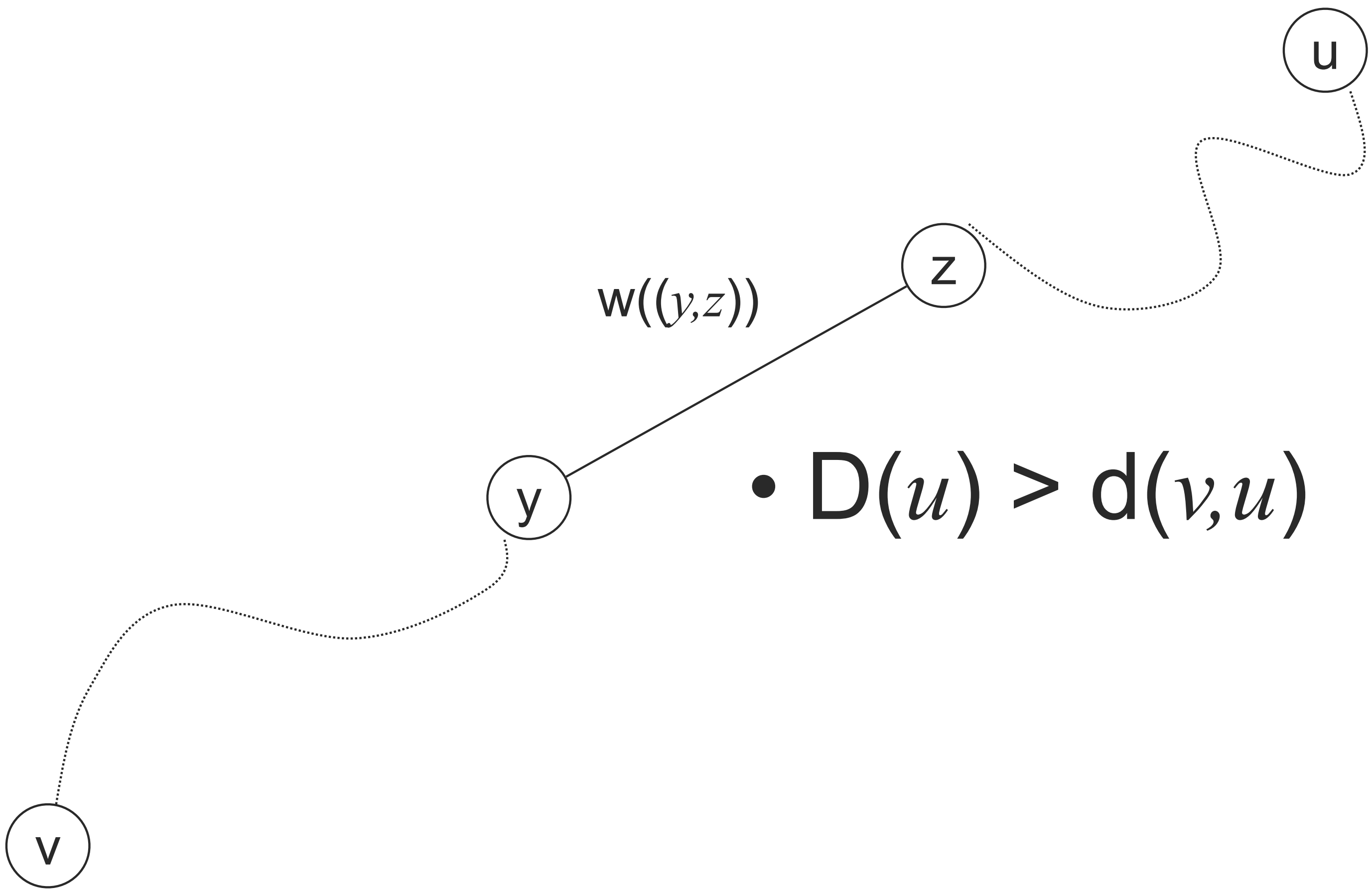
Whenever u is pulled into cloud C, D[u] stores the length from a shortest path from u to v

Proof. Assume: claim is wrong. Then:

there exists a vertex t that is pulled into cloud C and D[t] > d(v,t)

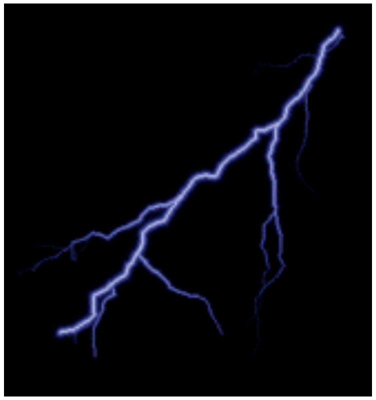
We define:

- u the first such vertex (currently) pulled into C
- P a shortest path in G from source v to vertex u
- y the last vertex that lies on P and is pulled "correctly" into C
- z the vertex closest to y that lies on P and is not in C



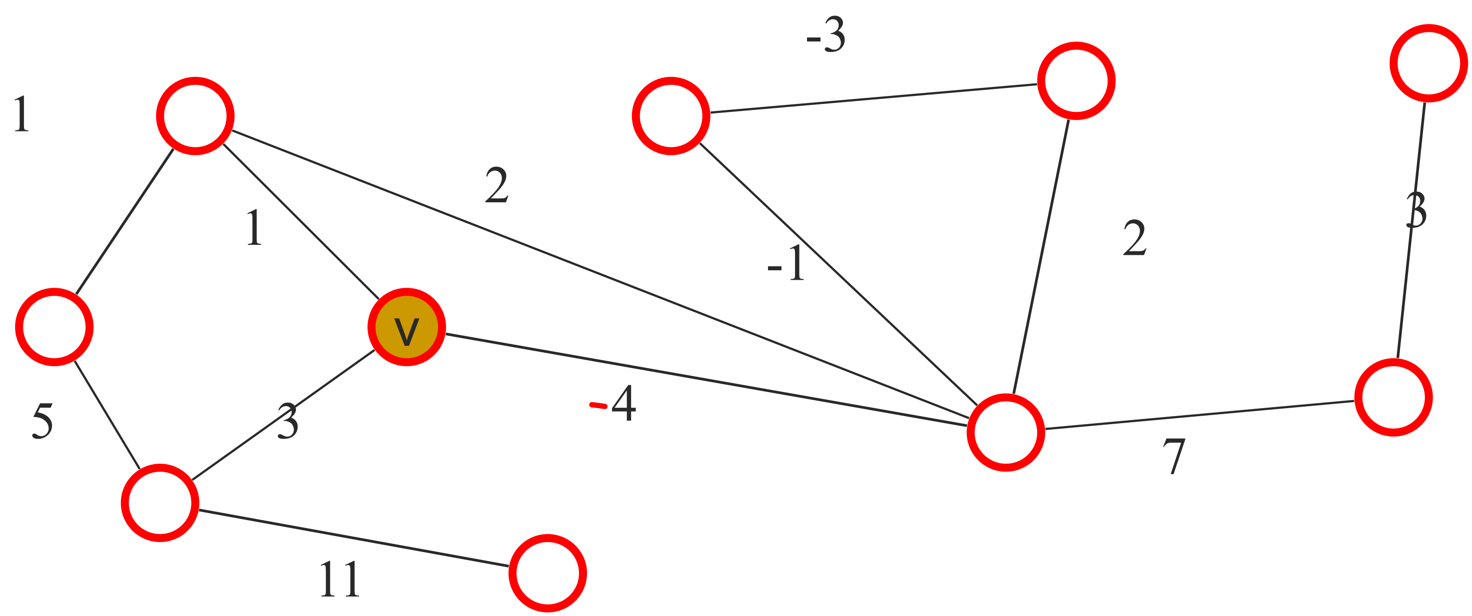
- $D(u) > d(v,u)$

- $D(u) > d(v,u)$
- $y \in C$, $D[y] = d(v,y)$
- $D(u) \leq D(z)$
- $D(z) = d(v,z)$

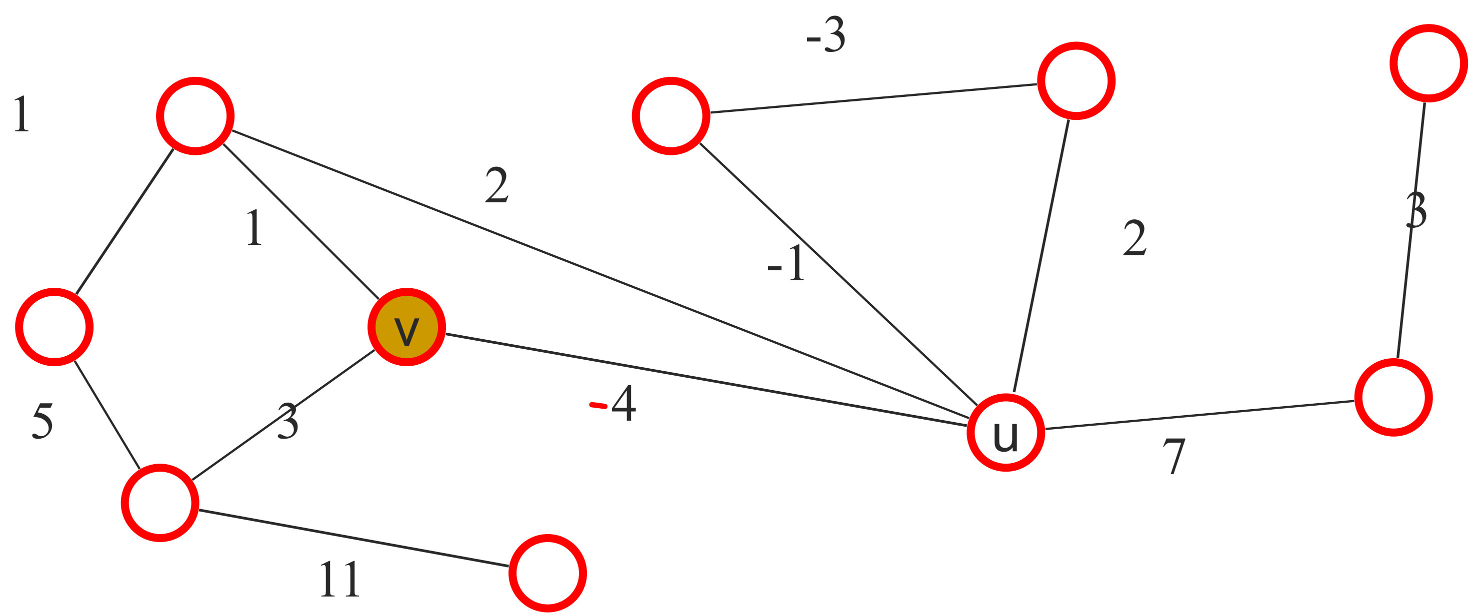


- $\underline{D(u)} \leq D(z) = d(v,z) \leq d(v,z) + d(z,u)$
  $= \underline{d(v,u)}$

Example (Input contains negative edges)

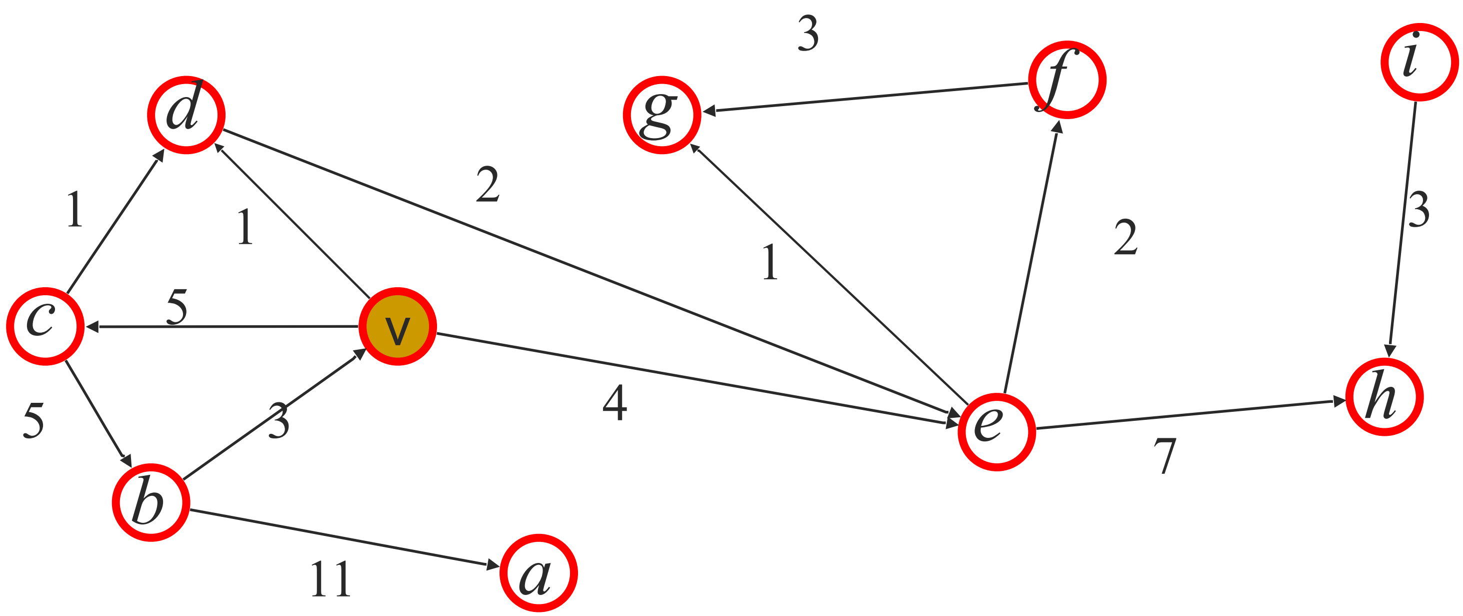


How far away is u from v?



Does Dijkstra’s algorithm work?

Directed graphs with positive edge weights



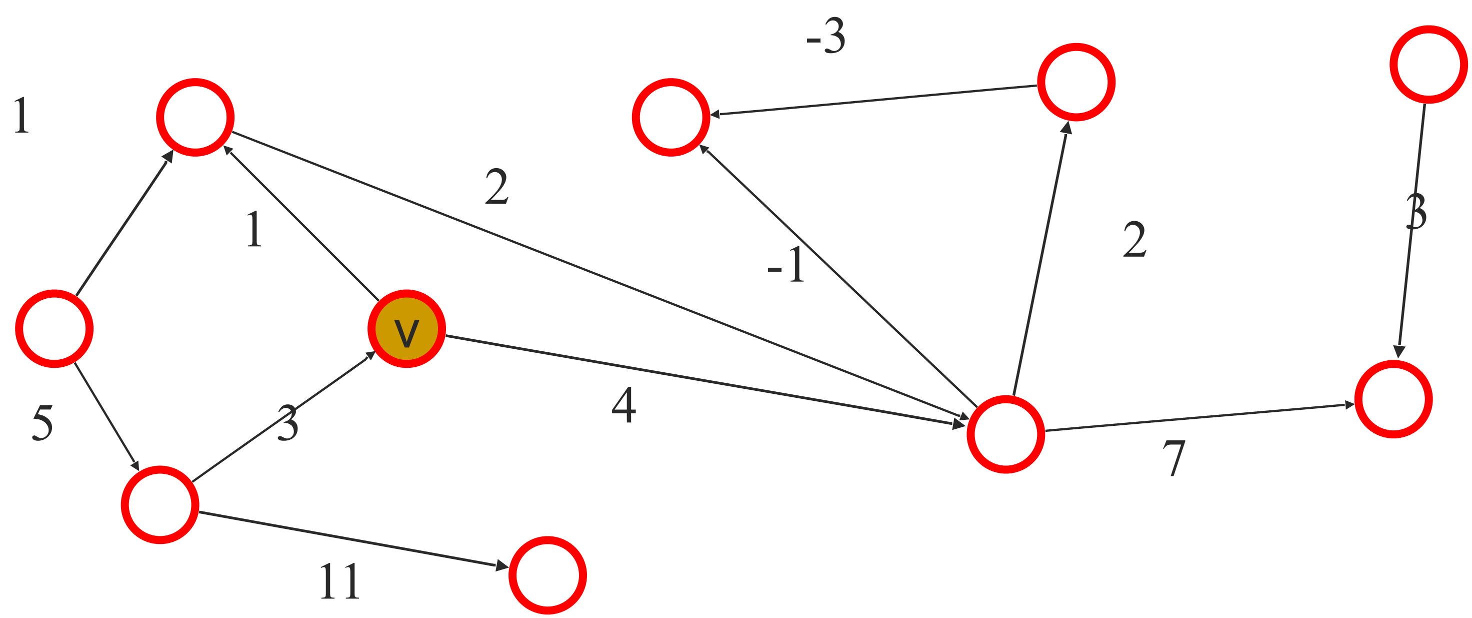
Does Dijkstra's algorithm work?

Single source shortest paths for directed graphs with positive edge

- Dijkstra's algorithm works without changes except here edges are directed, that is $(a,b) \neq (b,a)$
- The big-oh worst case running time remains the same

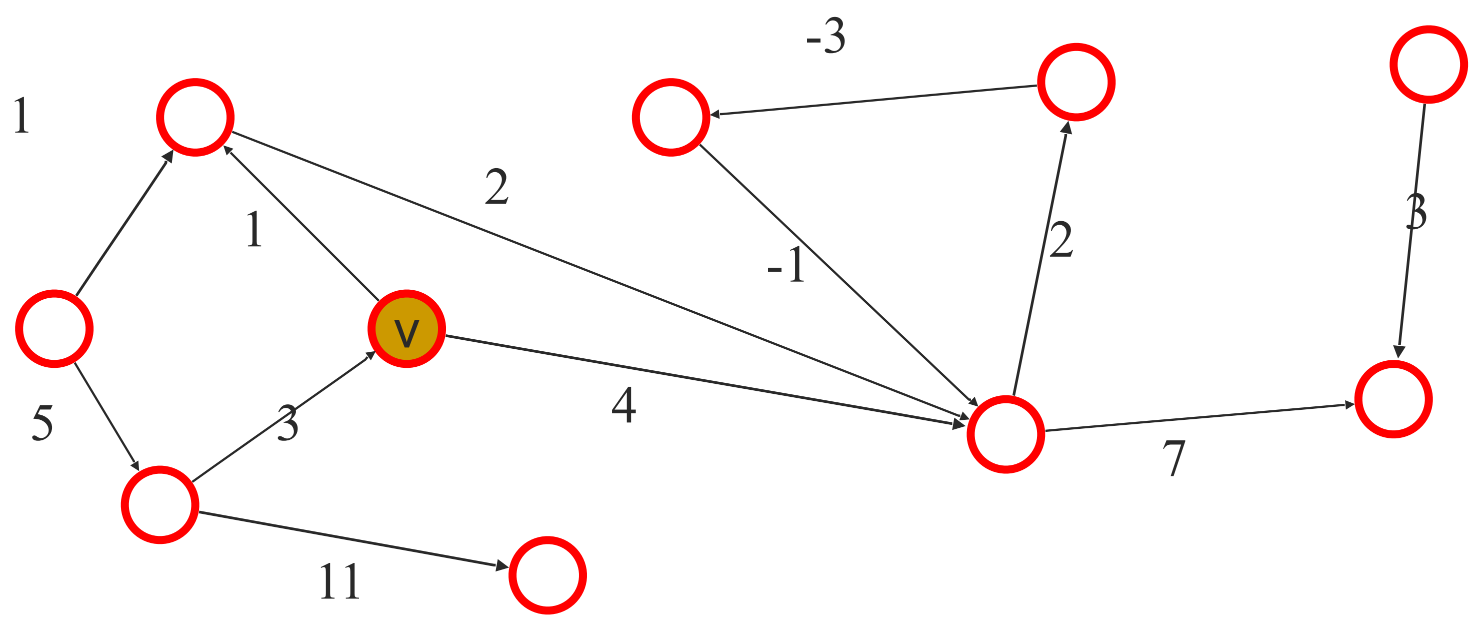
Shortest paths in graphs containing *negative* edges

- Not possible for undirected graphs
- What about directed graphs?



Shortest paths in directed graphs with negative edge weights

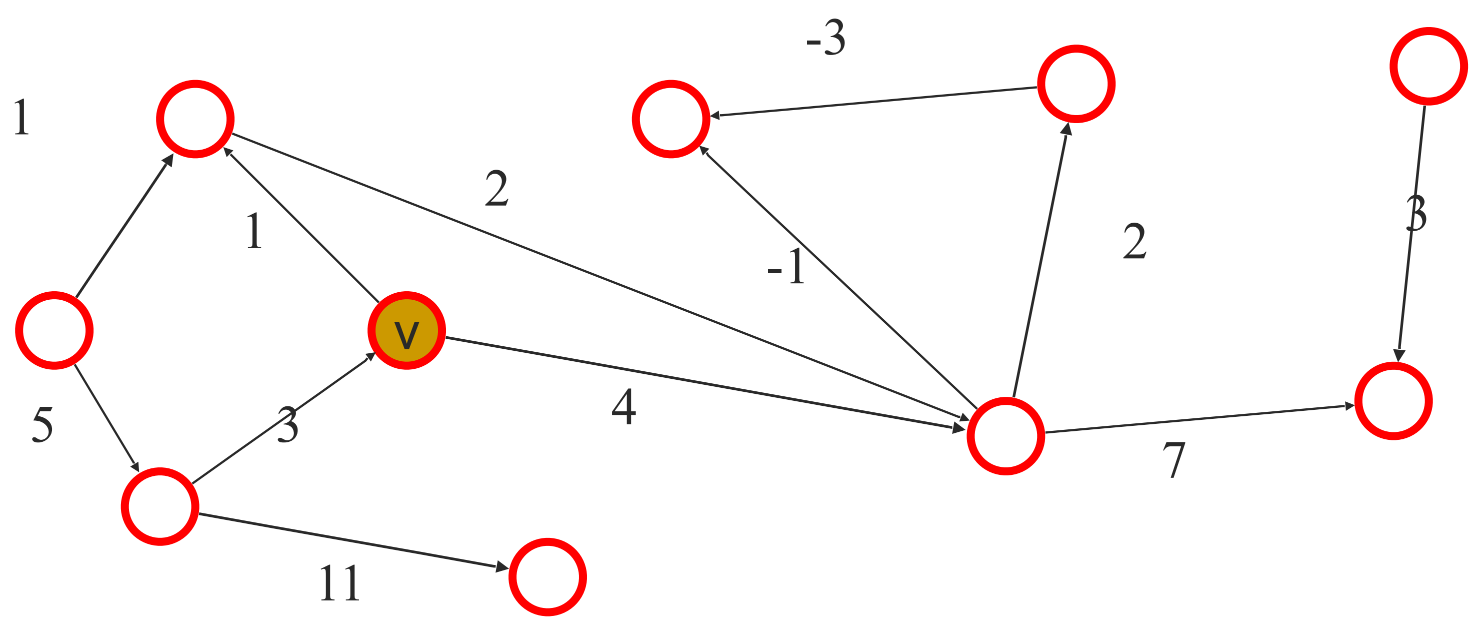
- Another example



Does Dijkstra's algorithm work?

Single Source shortest Paths in directed Graphs

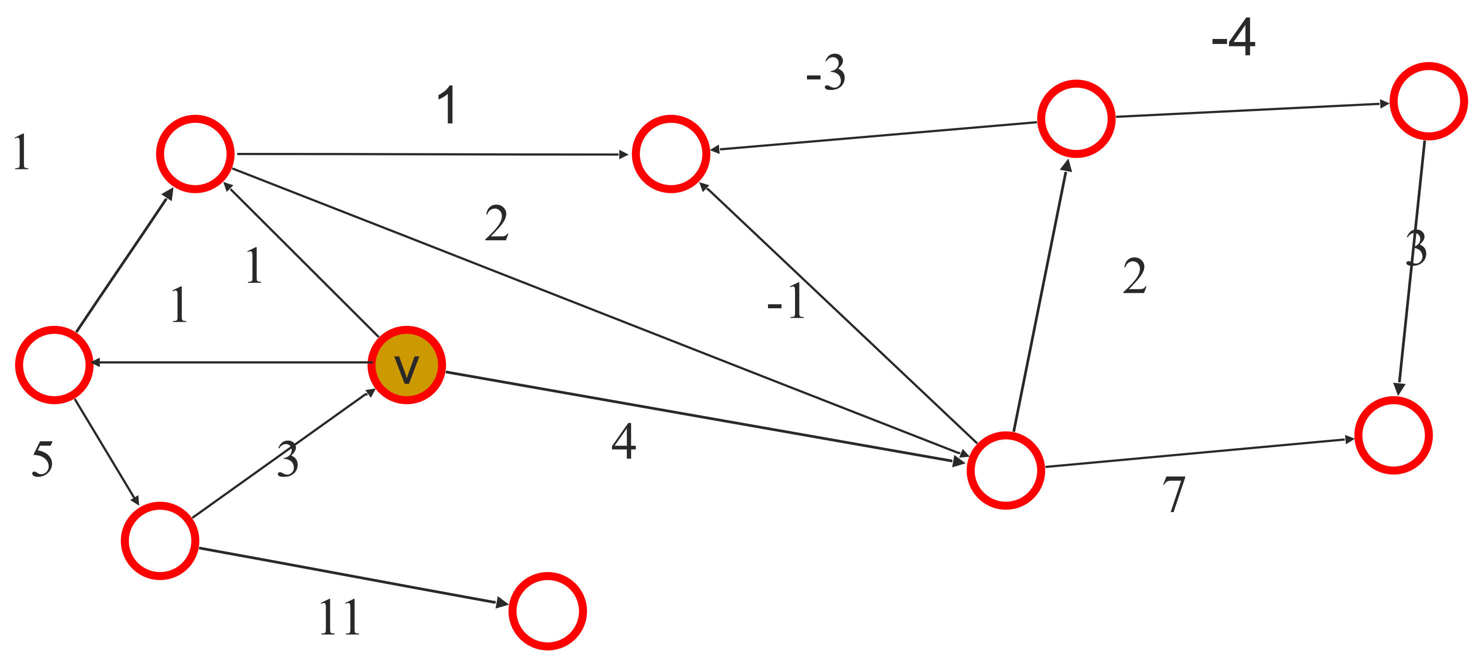
- If G does not contain any negative-weight cycles: does Dijkstra's algorithm work?



Negative edges and negative-weight cycles

- If G is directed, compute single-source shortest path problem using Bellman-Ford shortest path algorithm
- Negative-weight cycles are discovered

Bellman-Ford Algorithm for single source shortest paths in directed graphs



Algorithm Bellman-Ford(G,v)

Input: A simple undirected graph G with nonnegative edge-weights, a distinguished vertex v in G

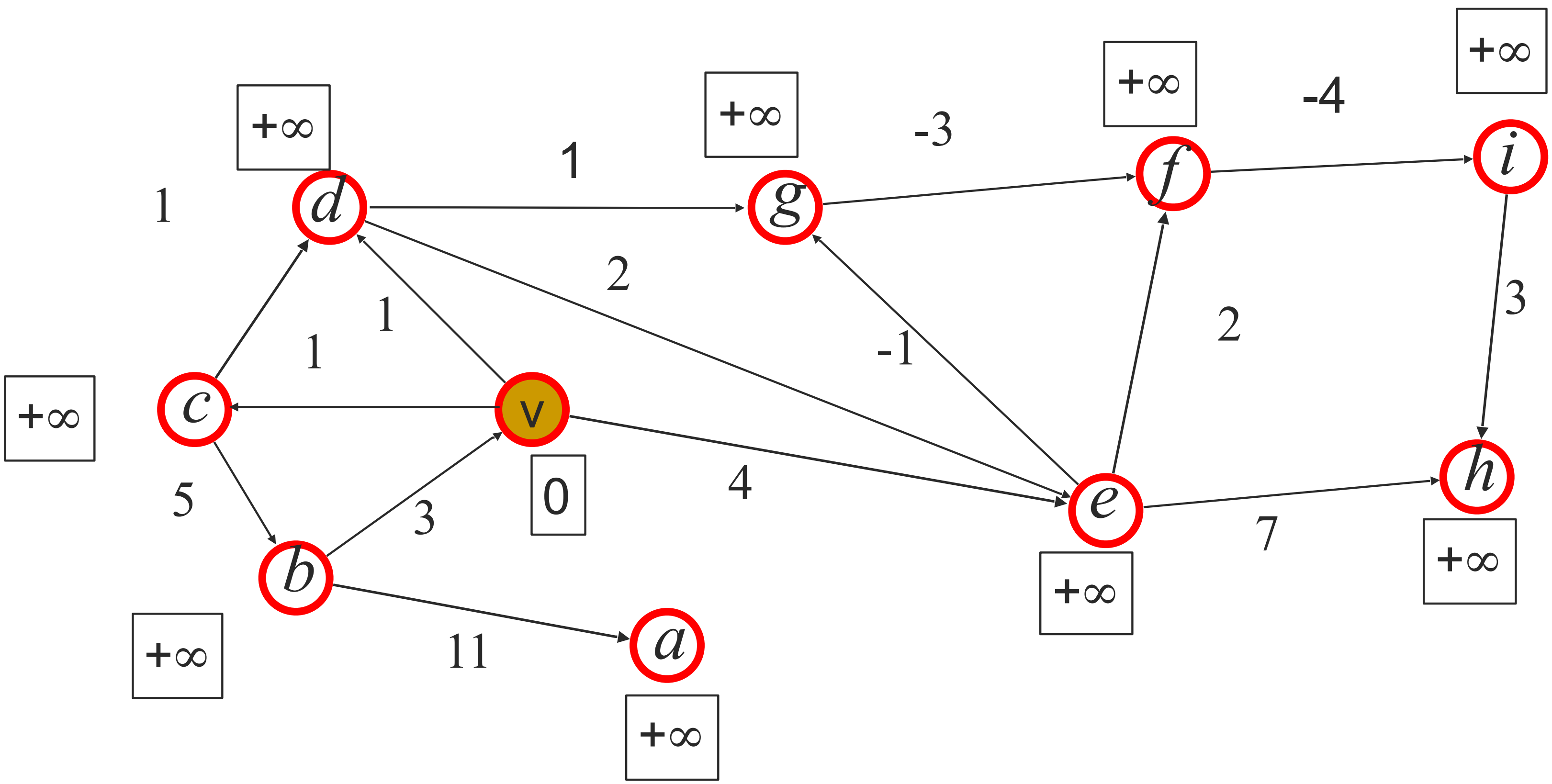
Output: A label D[u] for each vertex u in G such that D[u] is the distance from v to u in G.

Algorithm Bellman-Ford(G,v)

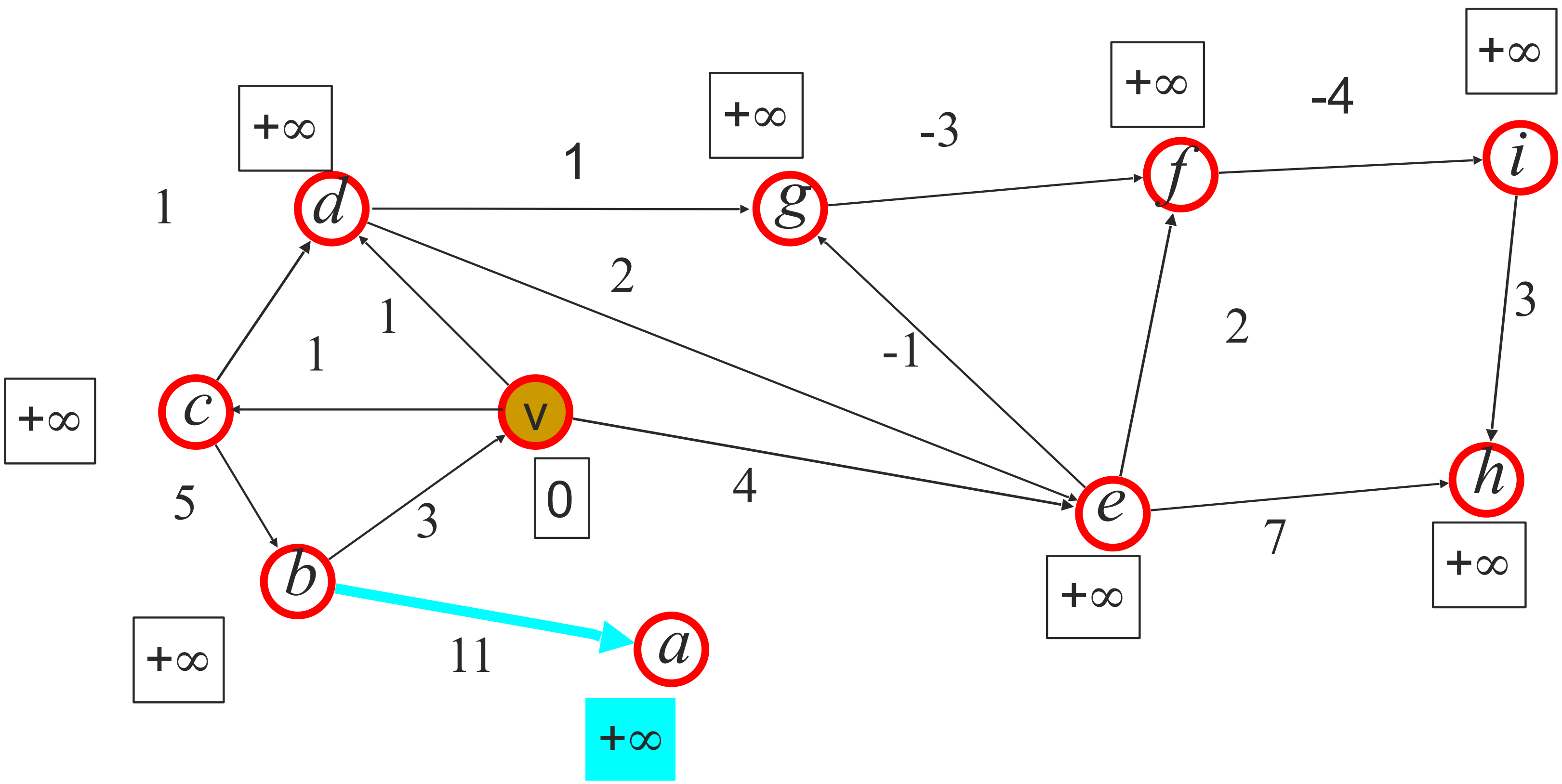
performs n-1 times a relaxation of every edge in the graph

```
D[v]←0
for each vertex u ≠ v of G do
   D[u]←+∞
for i ← 1 to n-1 do
   for each edge (u,z) in G do
      if D[u]+w((u,z)) < D[z] then
            D[z]←D[u]+w((u,z))
if there are no edges left with potential
   relaxation operations then
   return D
else
   return "G contains a negative cycle"
```

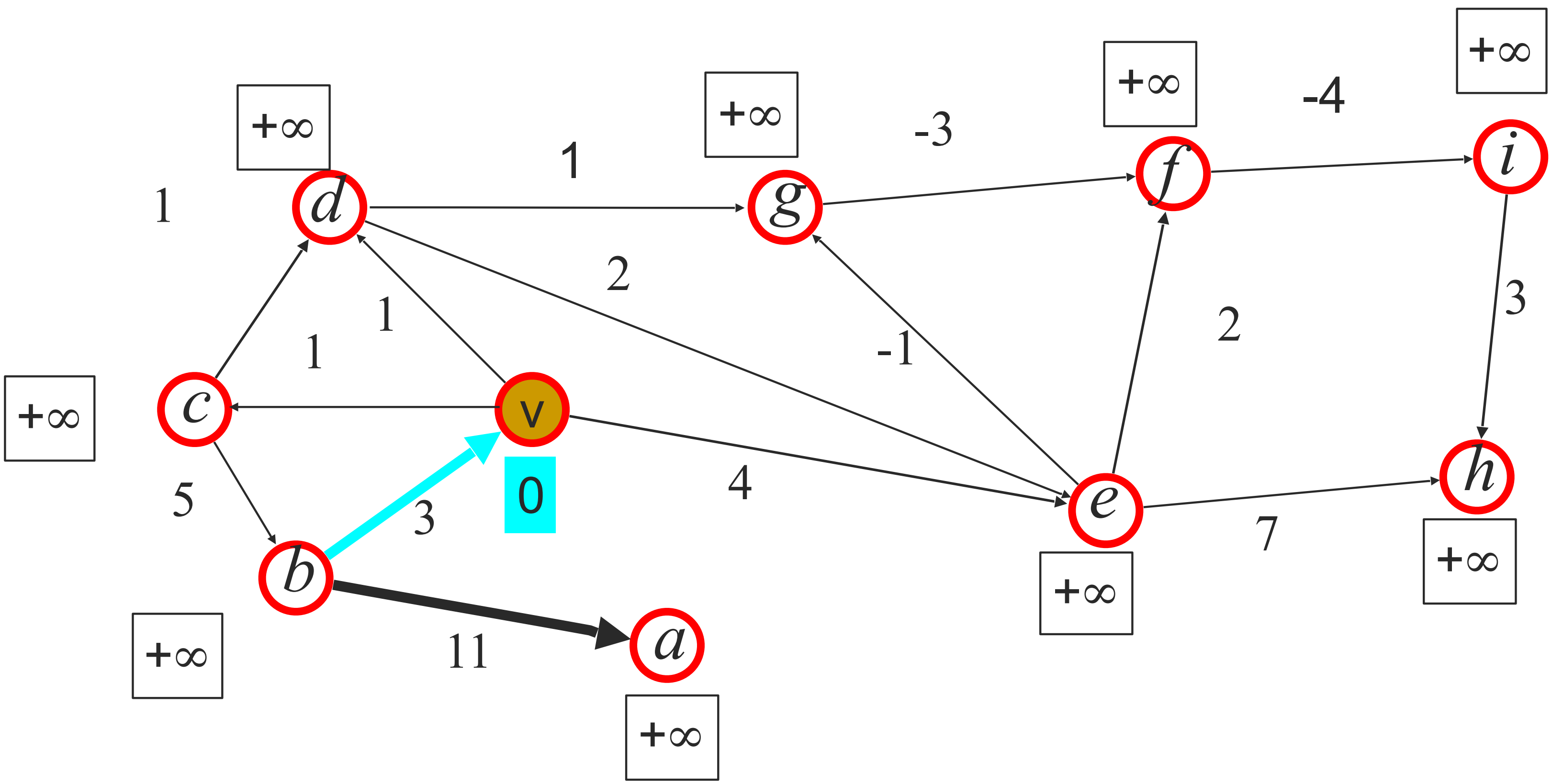
Initialize …



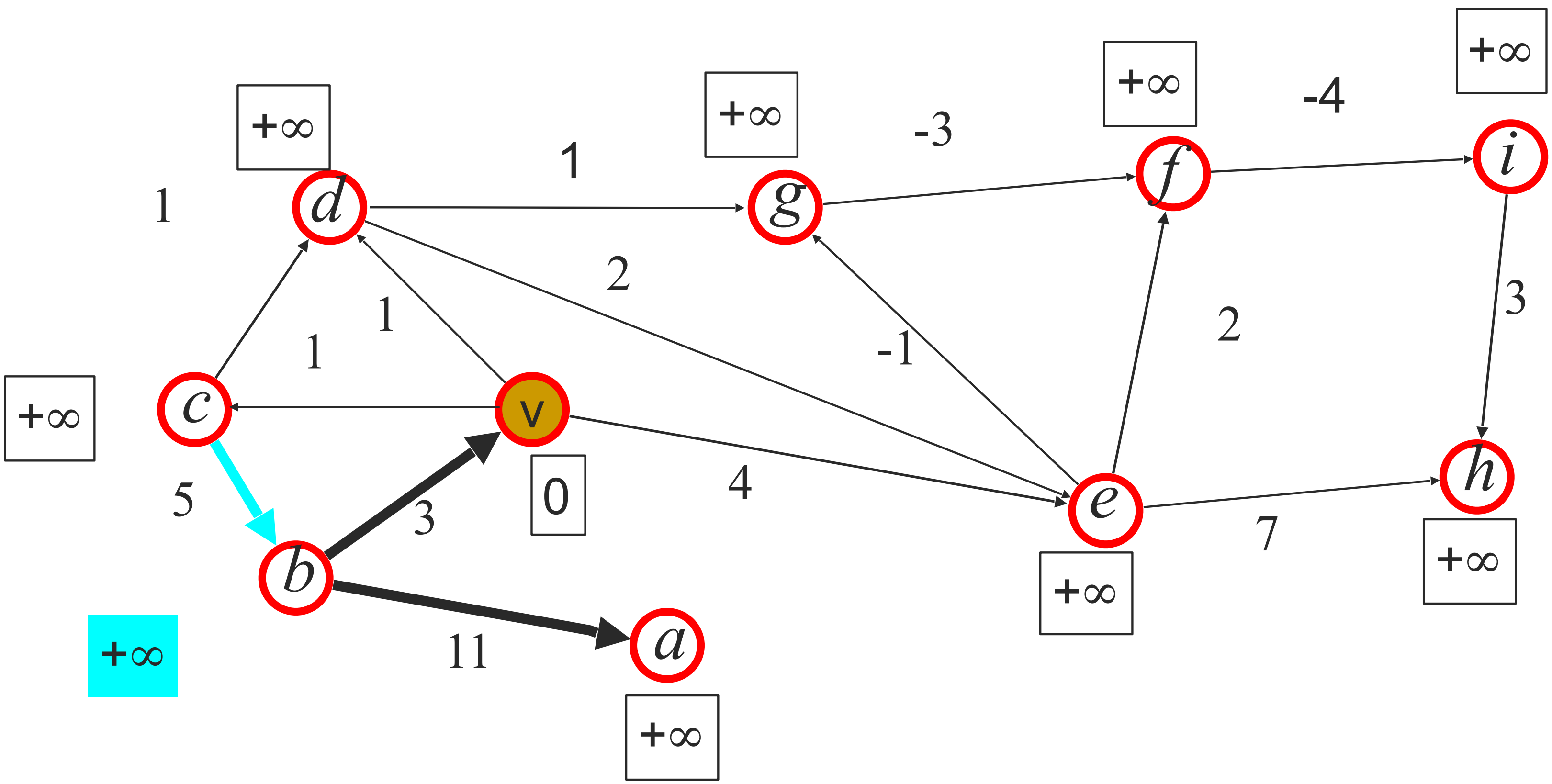
i=1



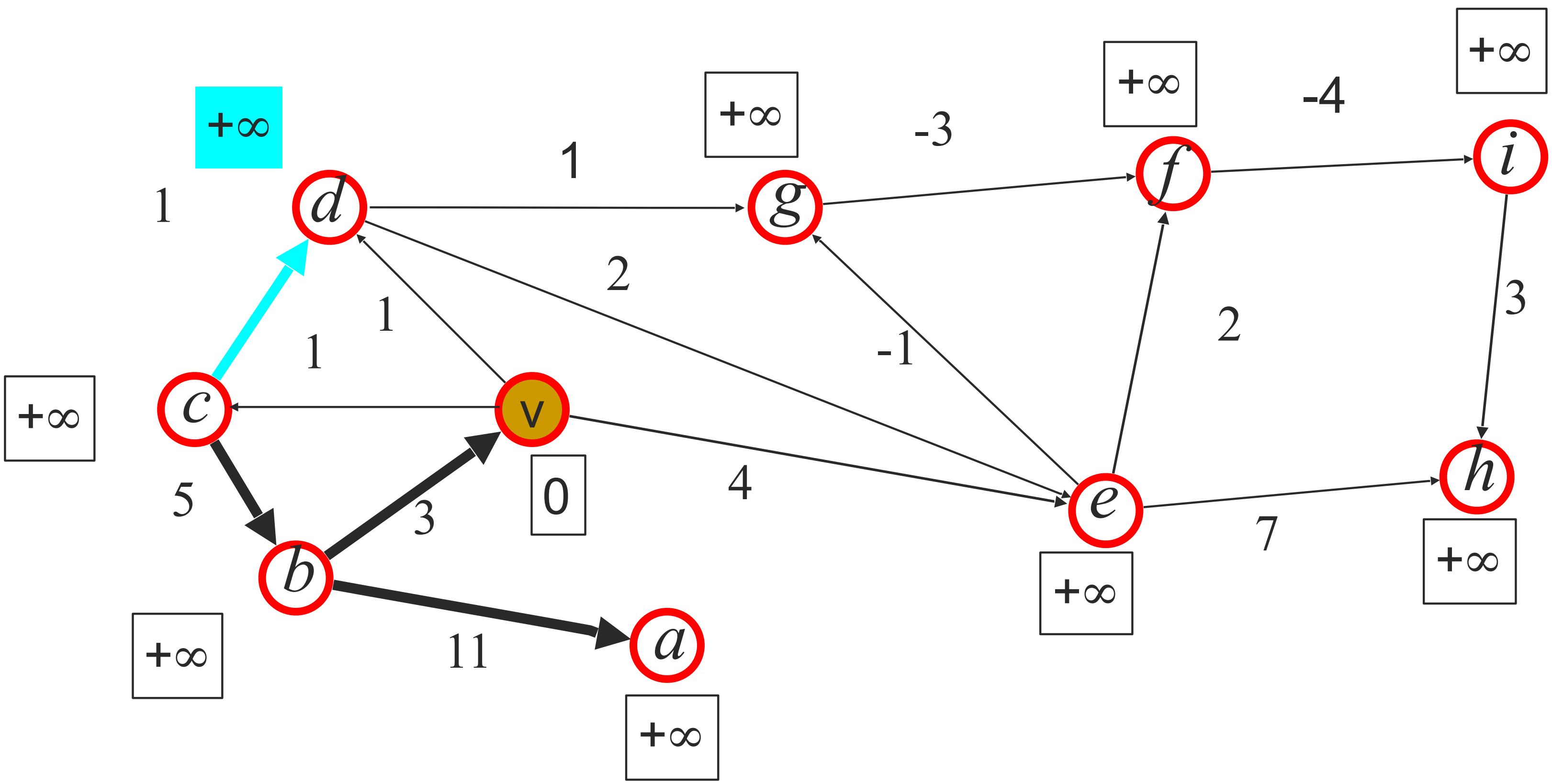
i=1



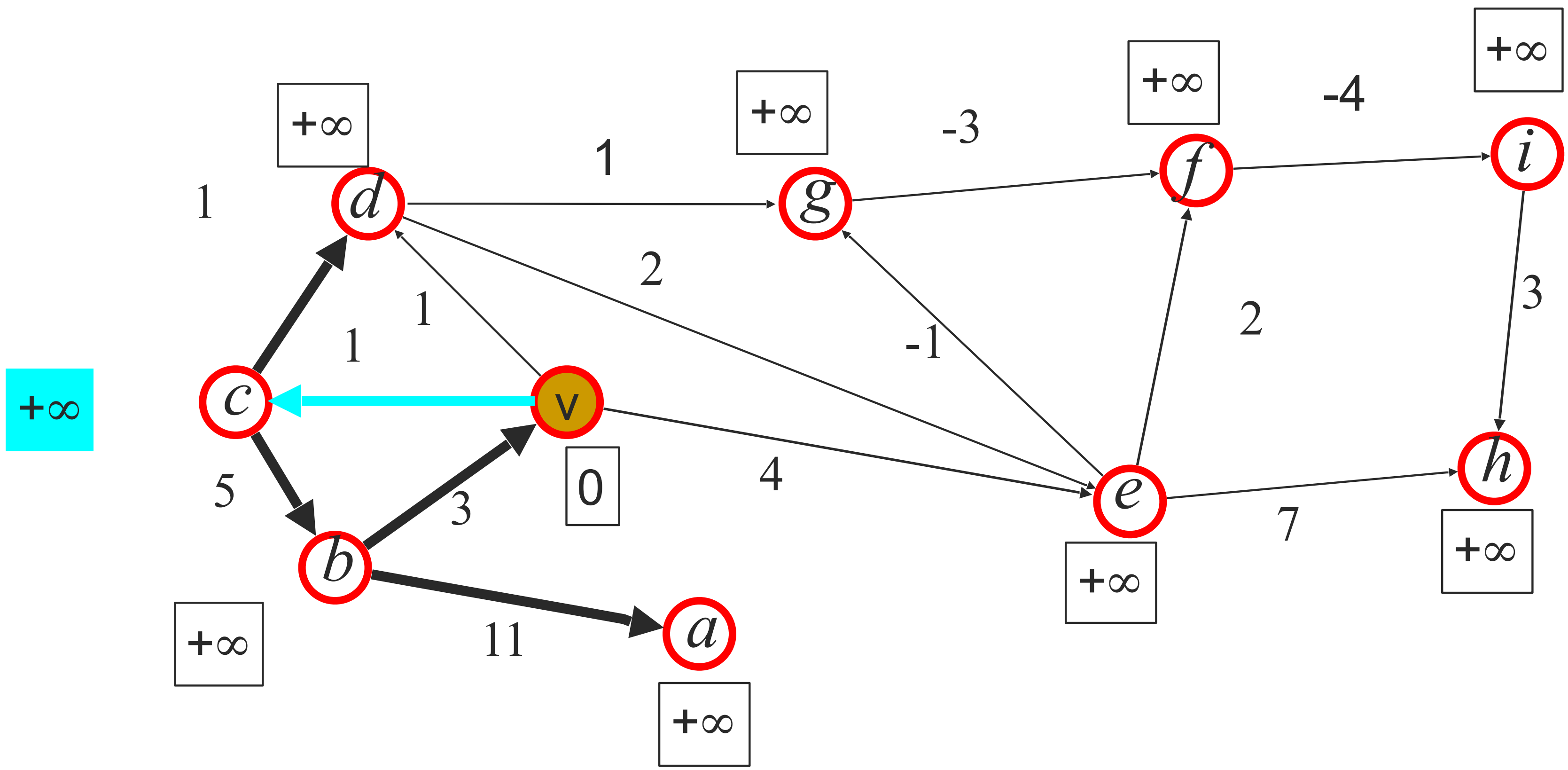
i=1



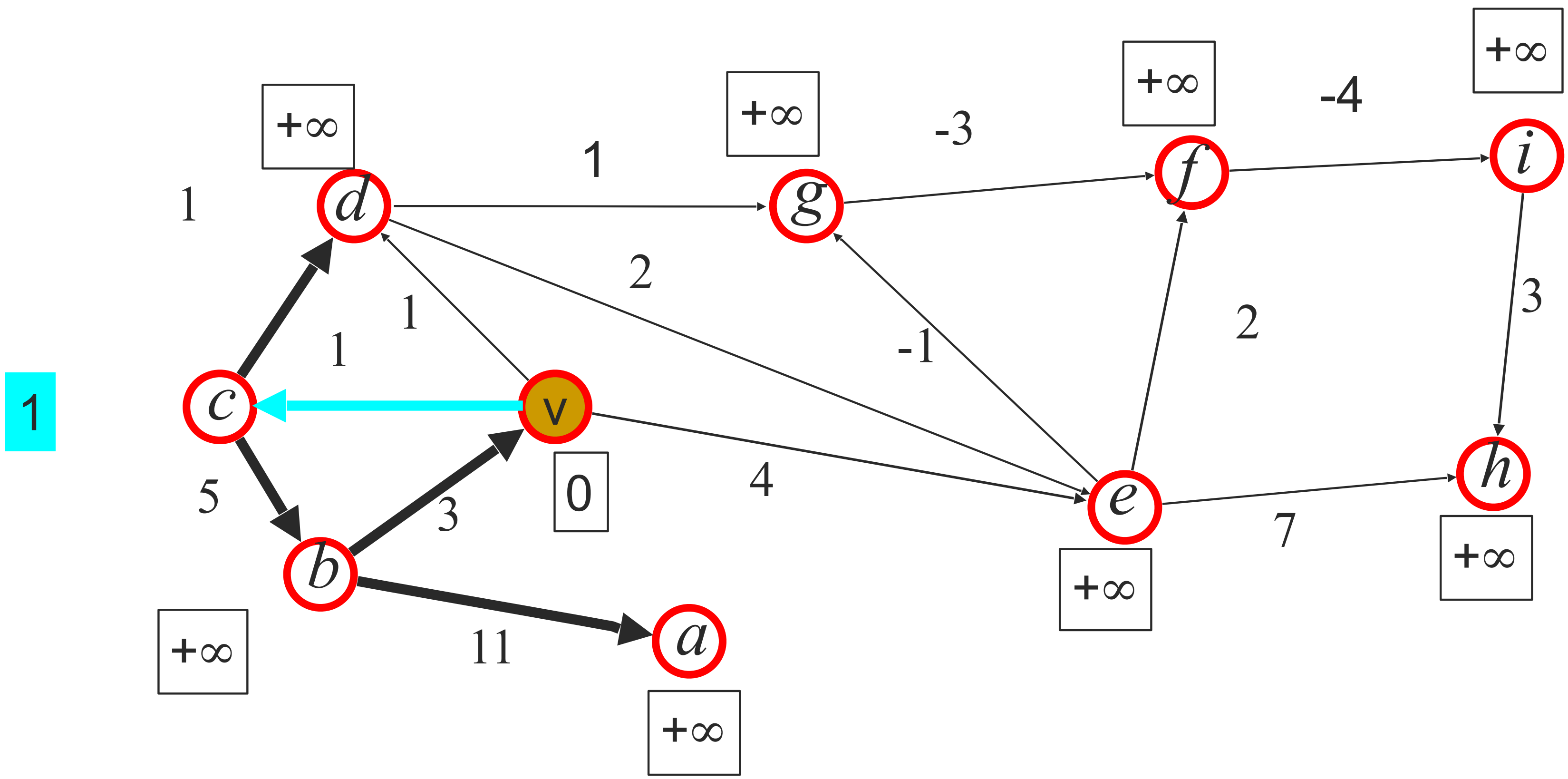
i=1



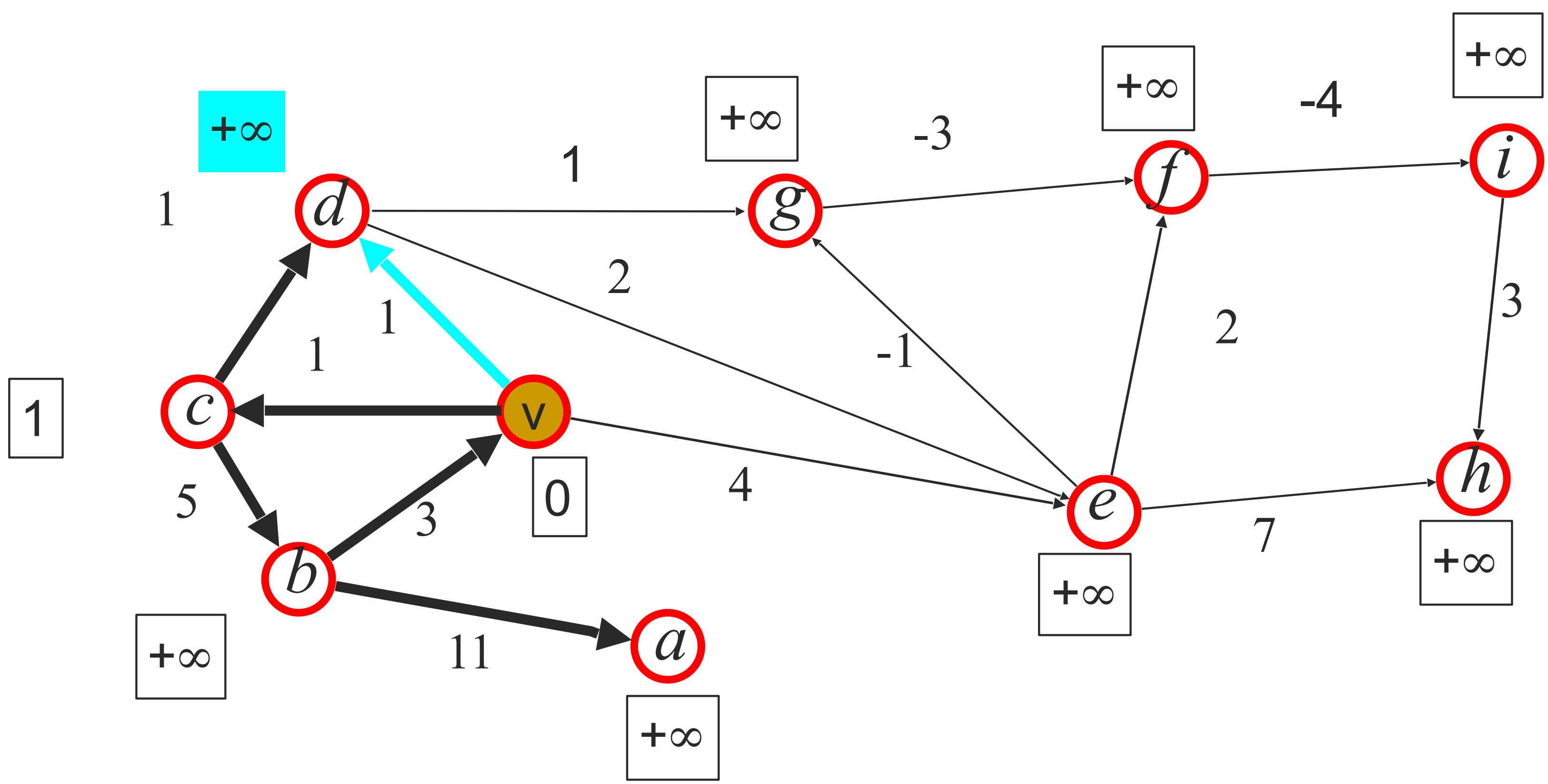
i=1



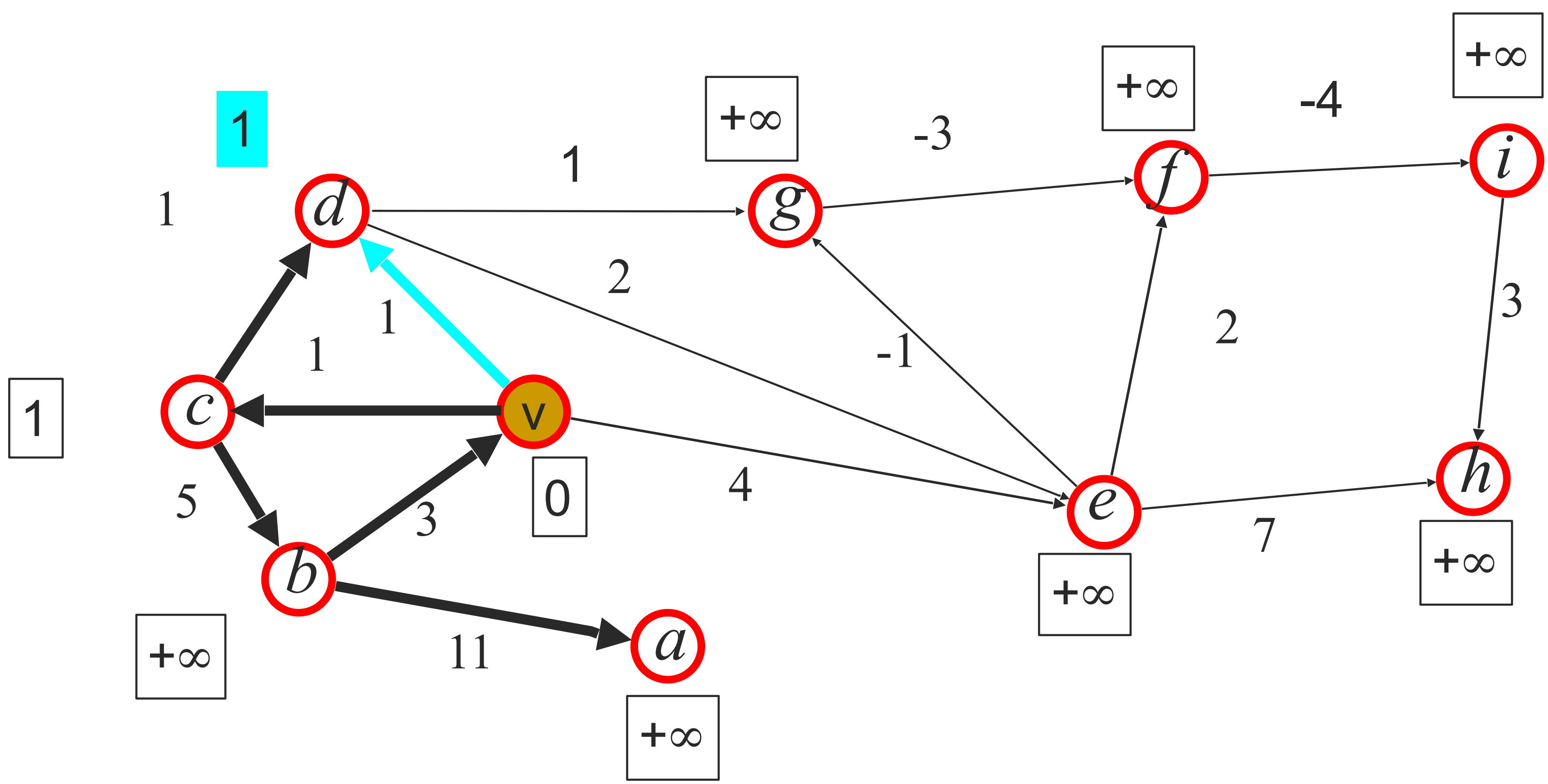
i=1



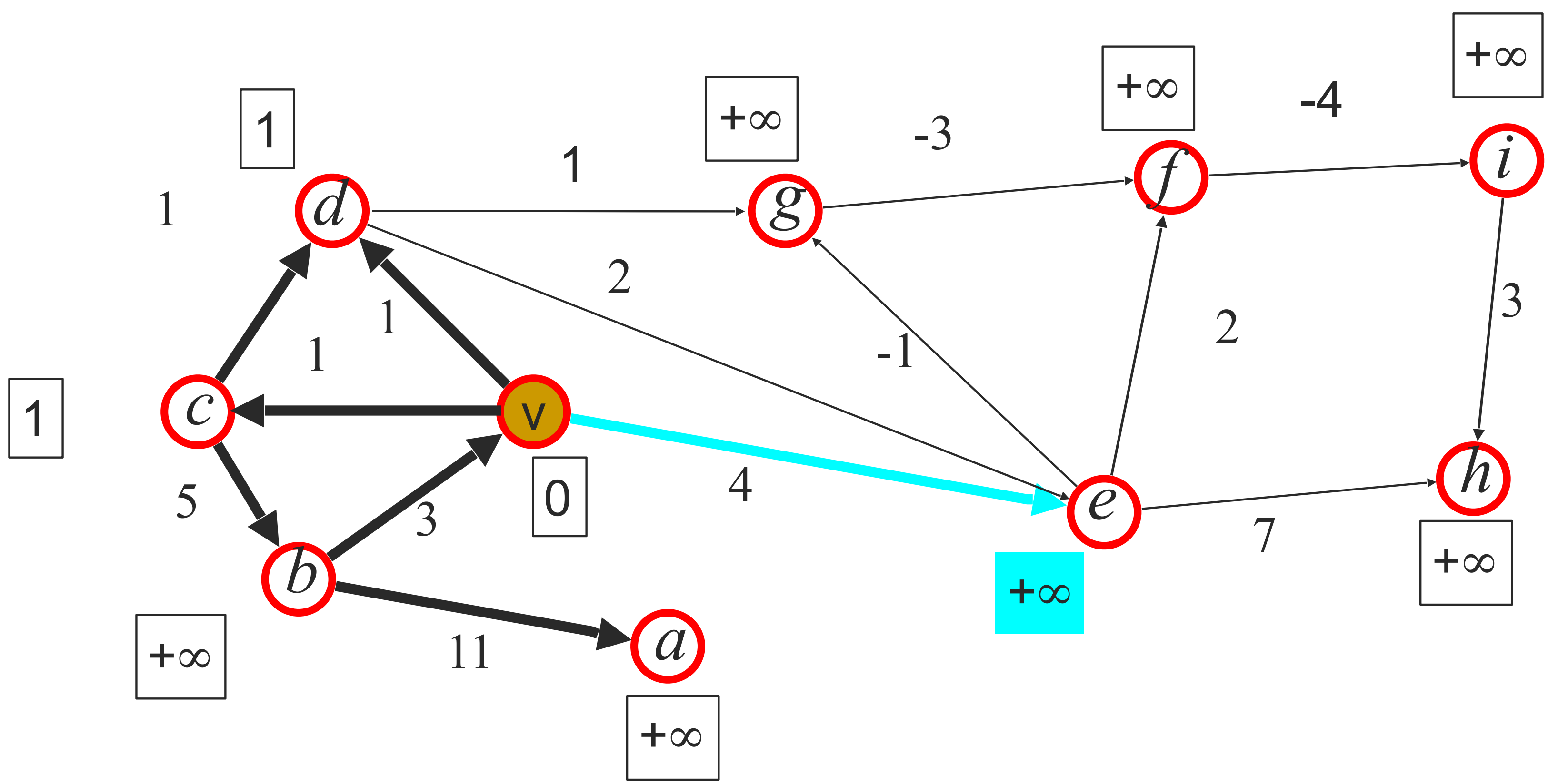
i=1



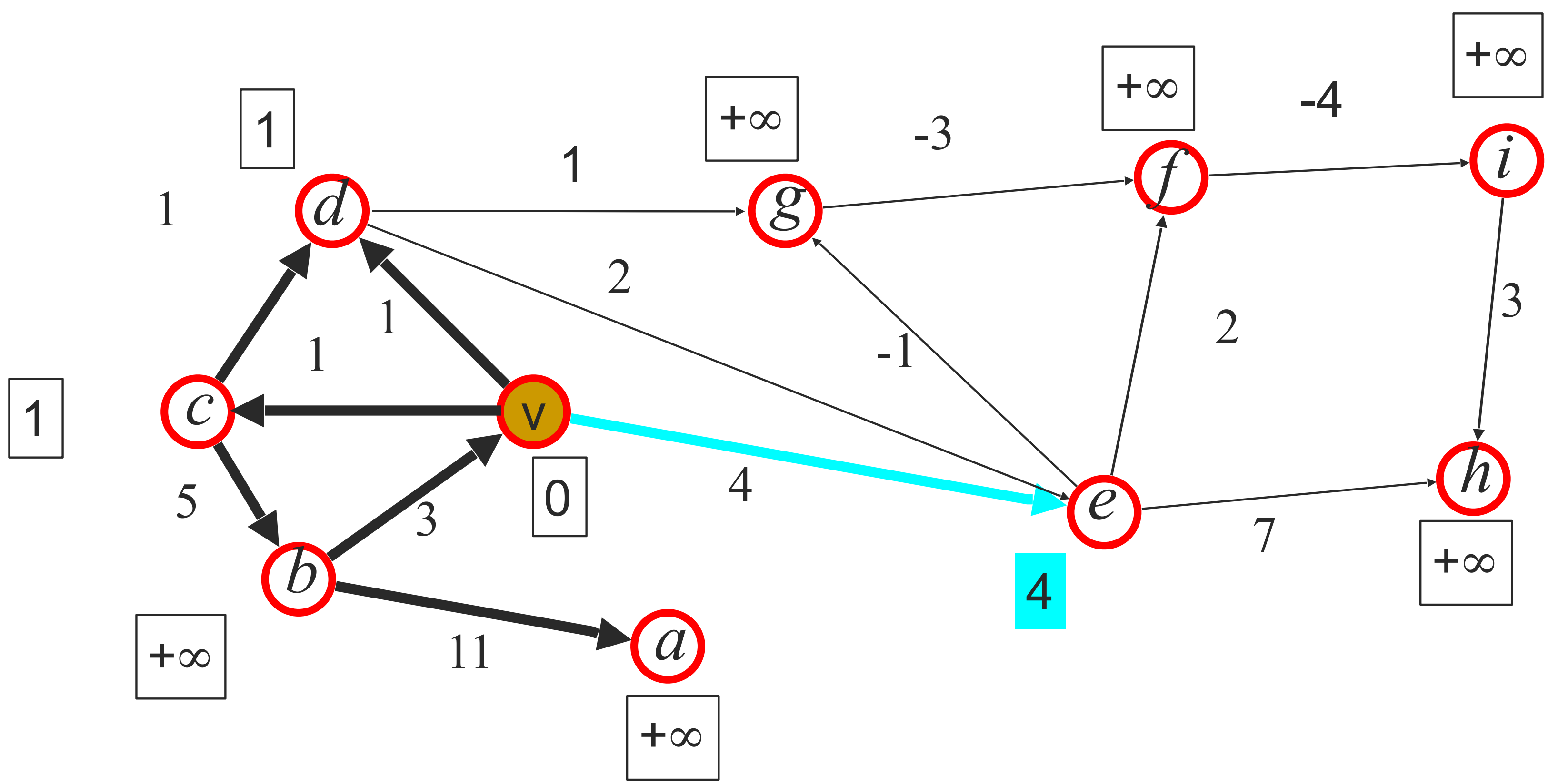
i=1



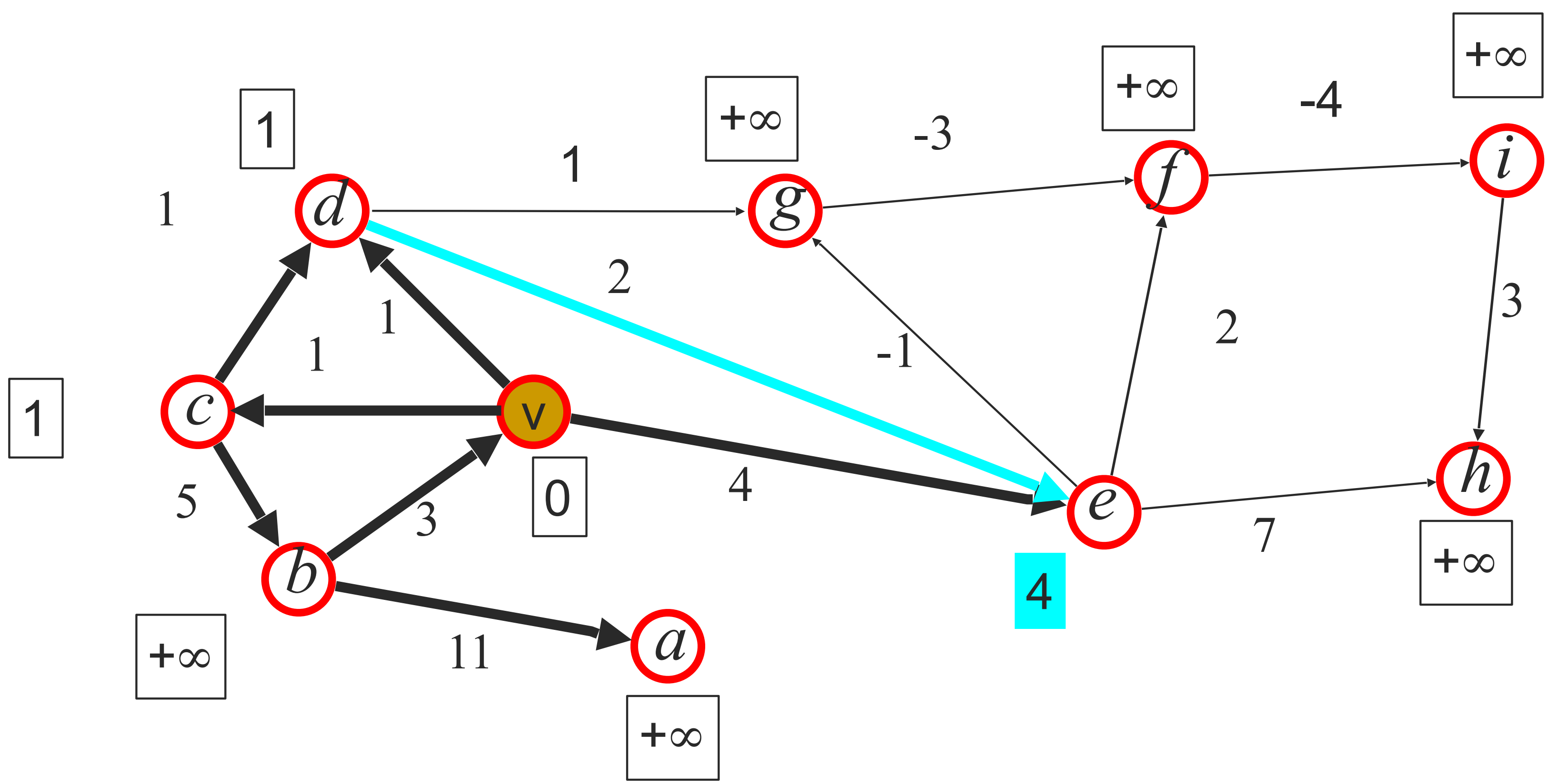
i=1



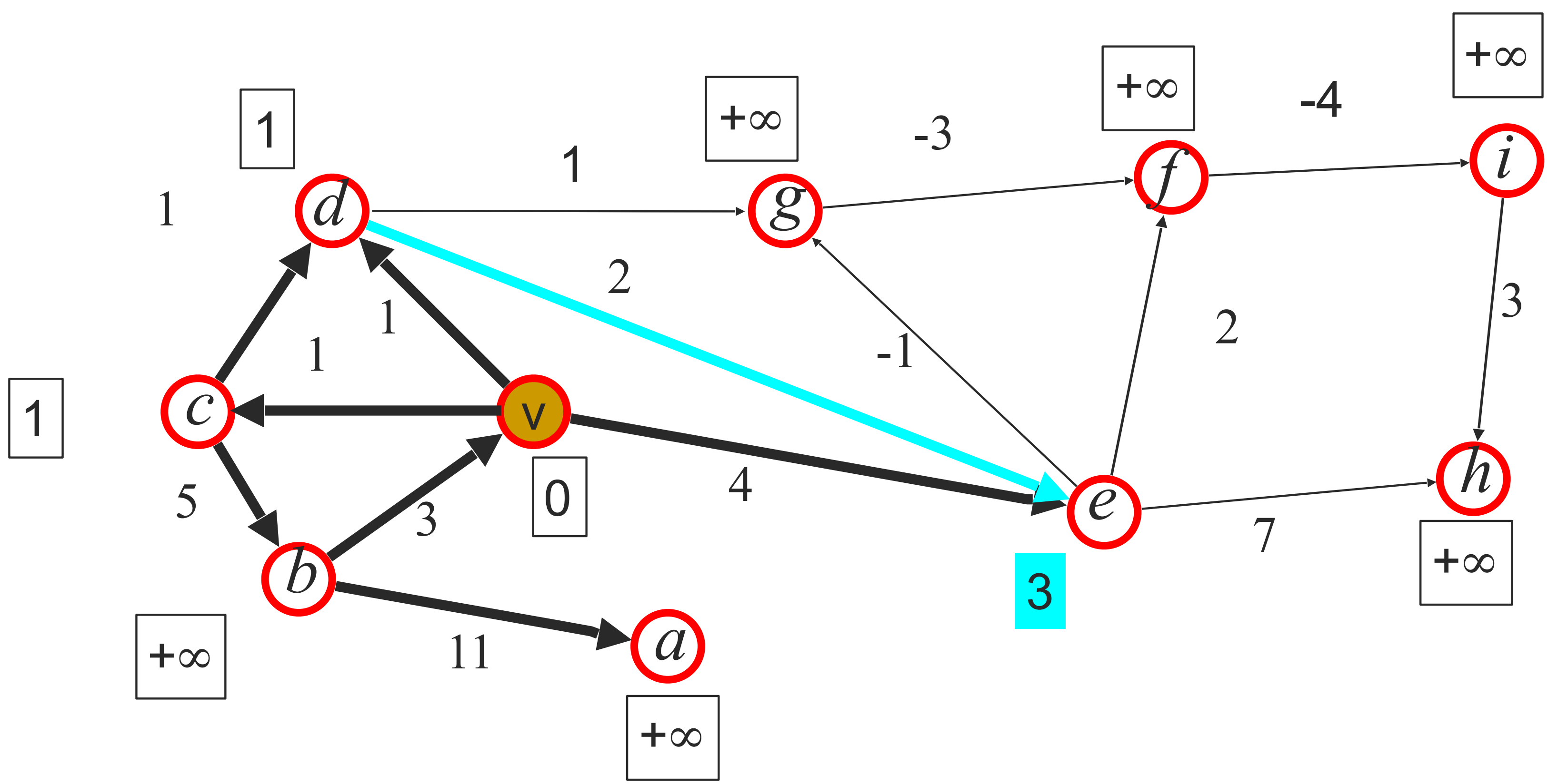
i=1



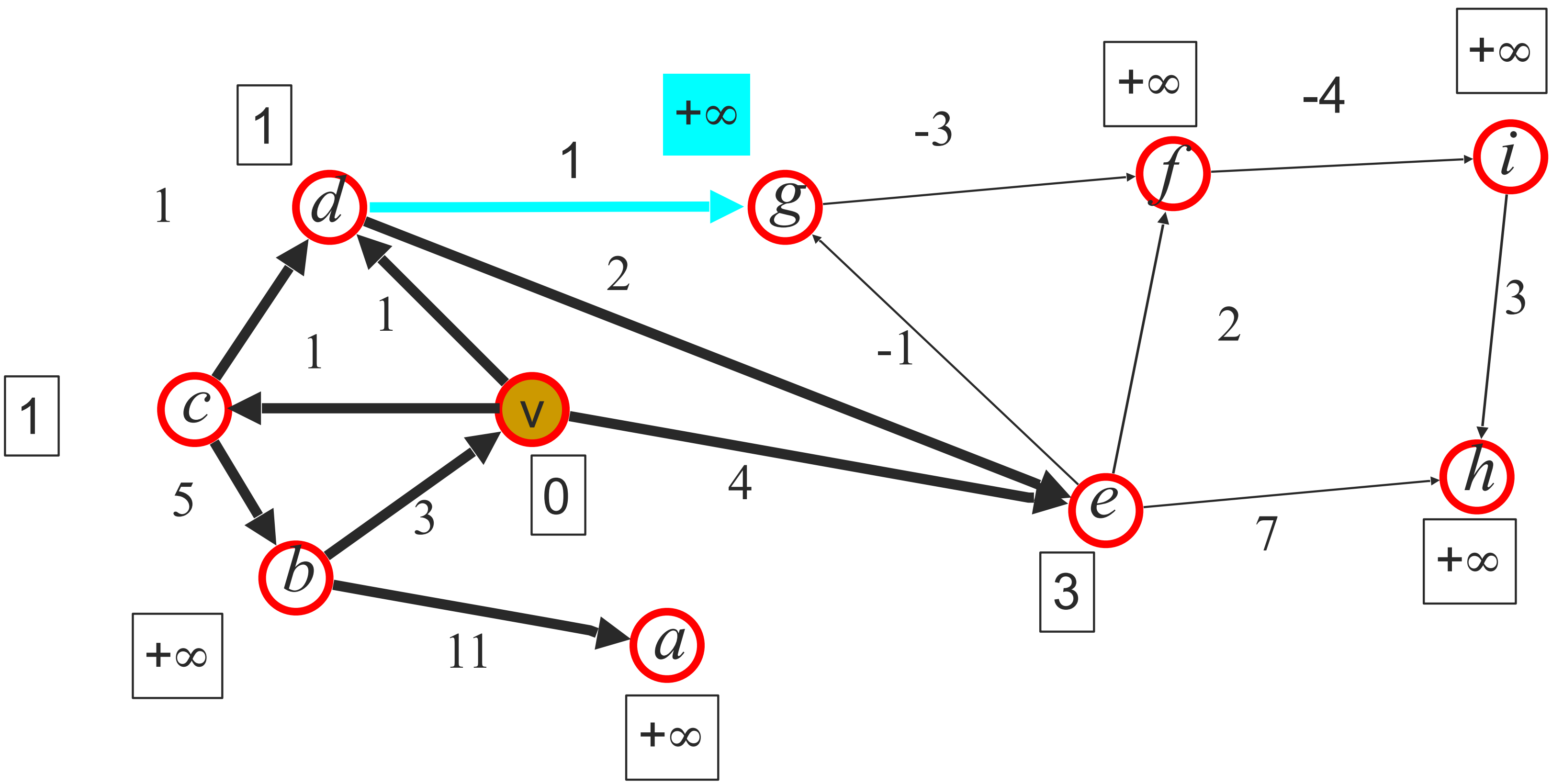
i=1



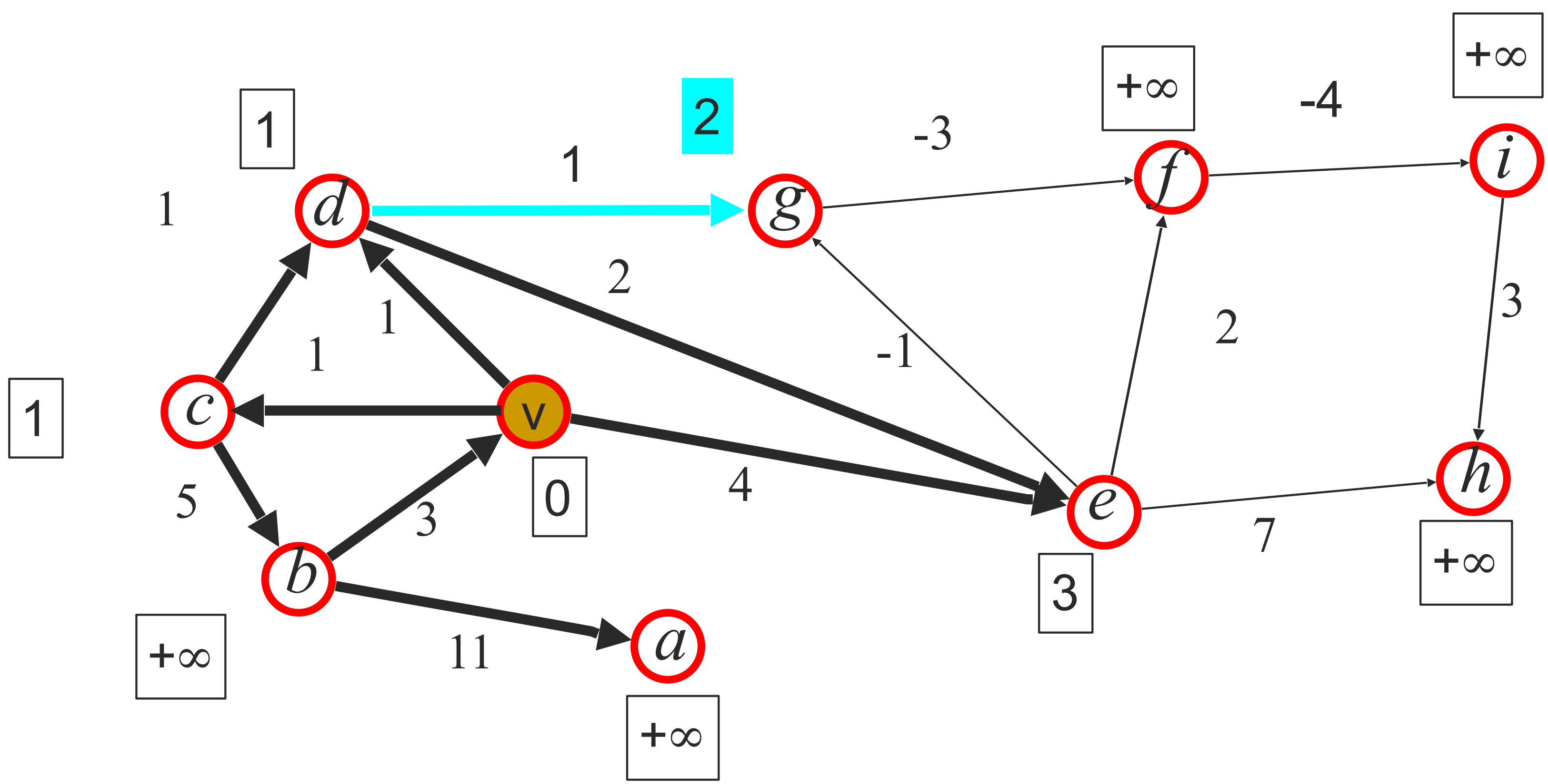
i=1



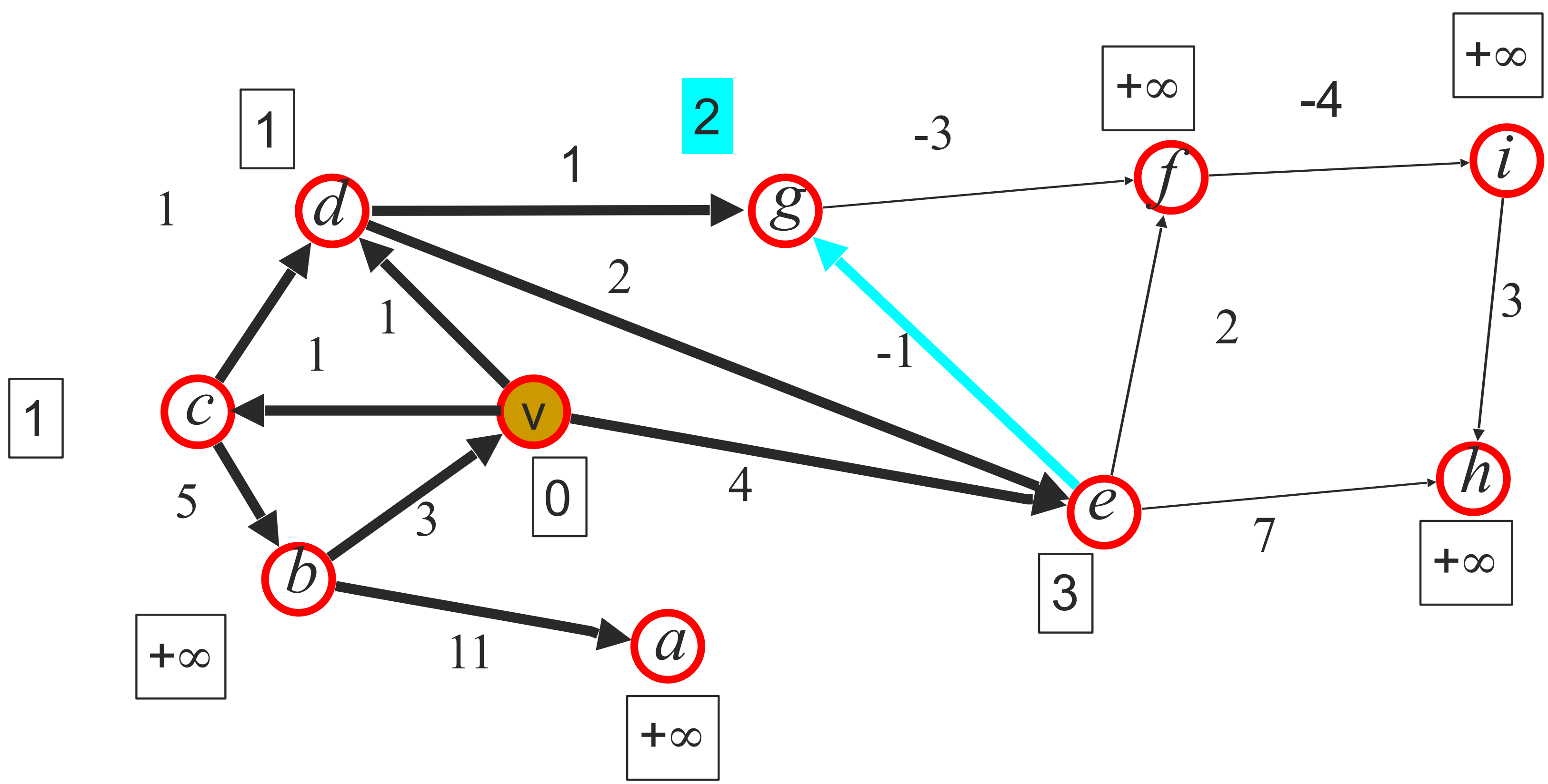
i=1



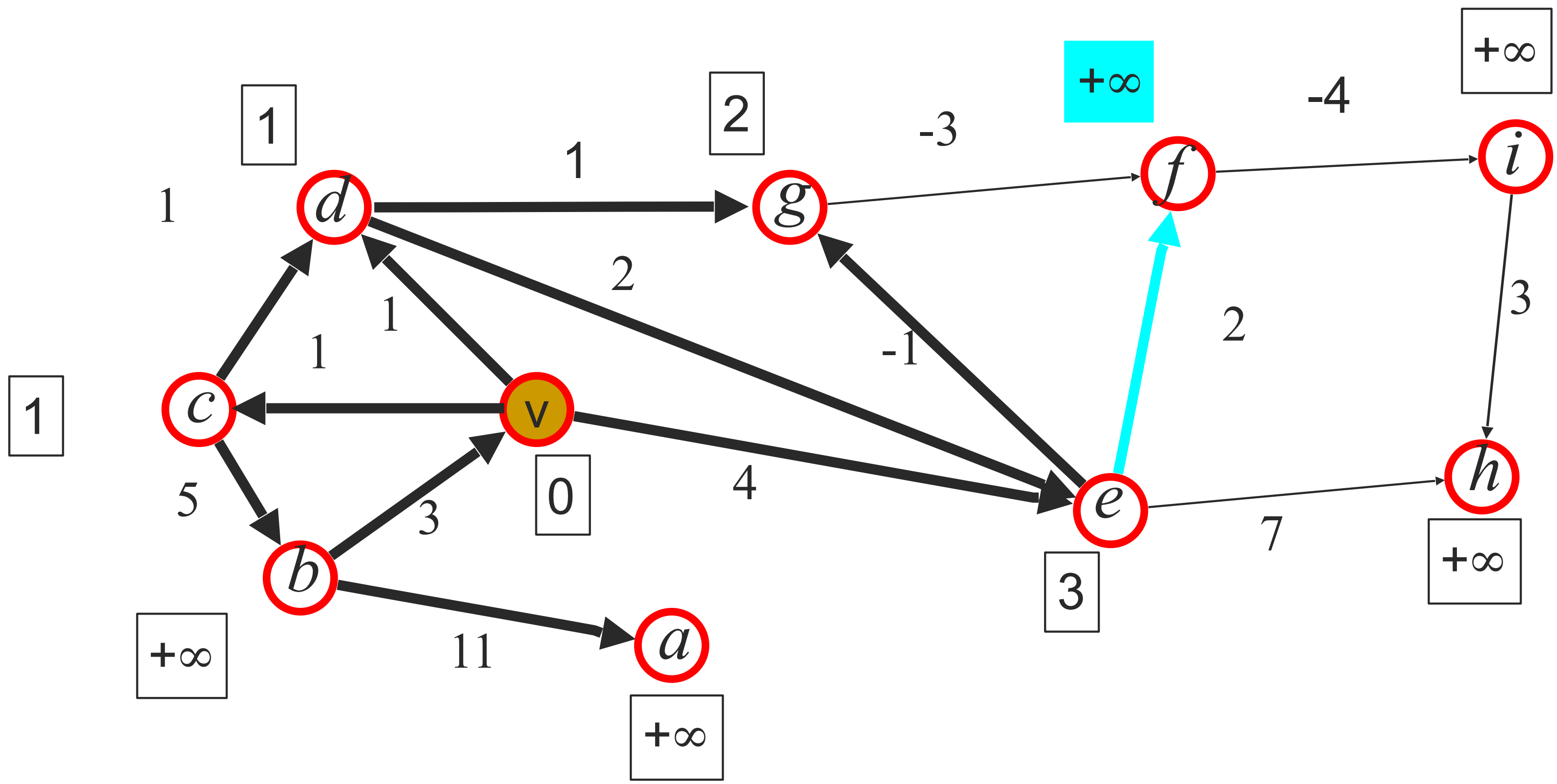
i=1



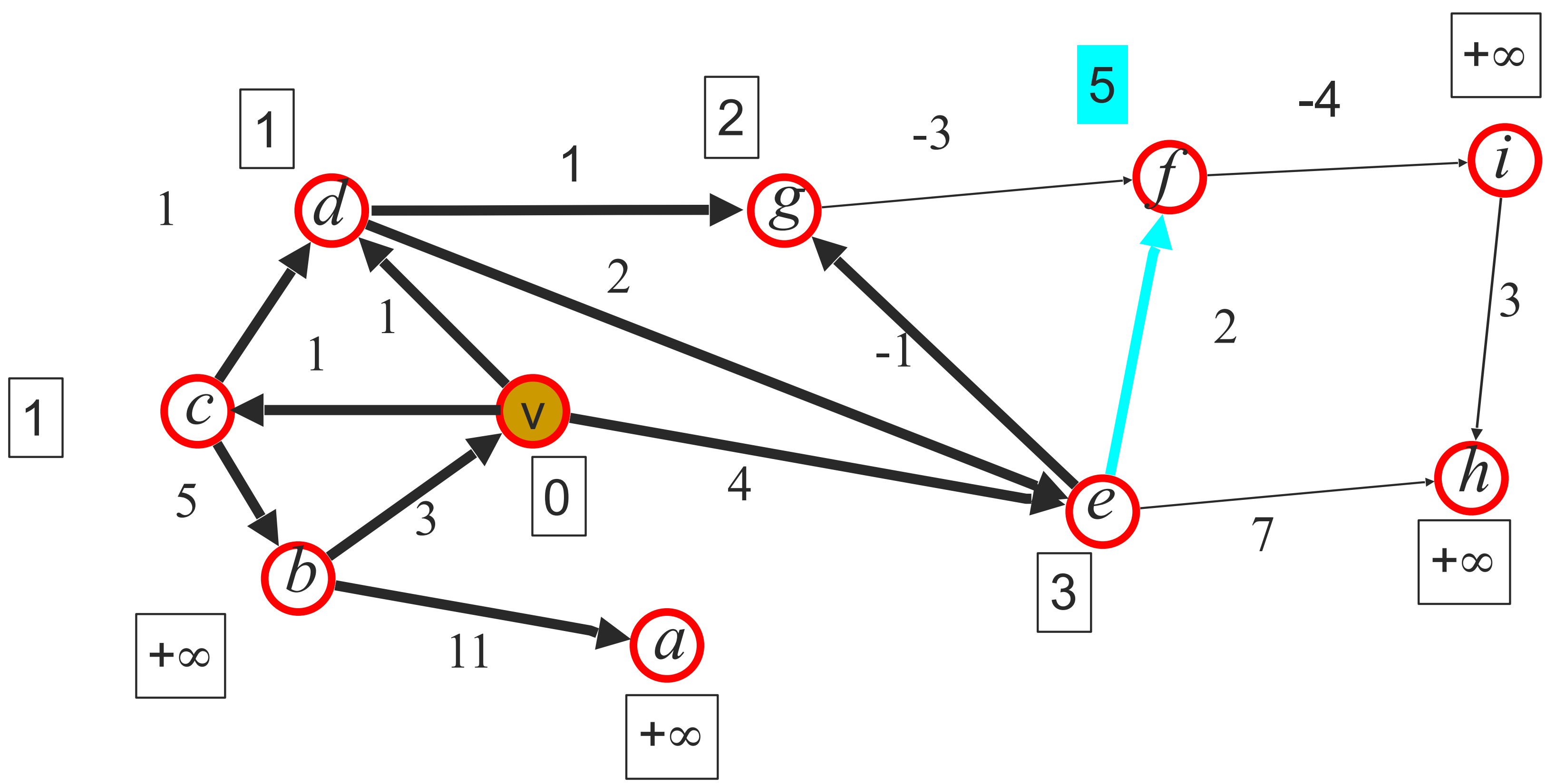
i=1



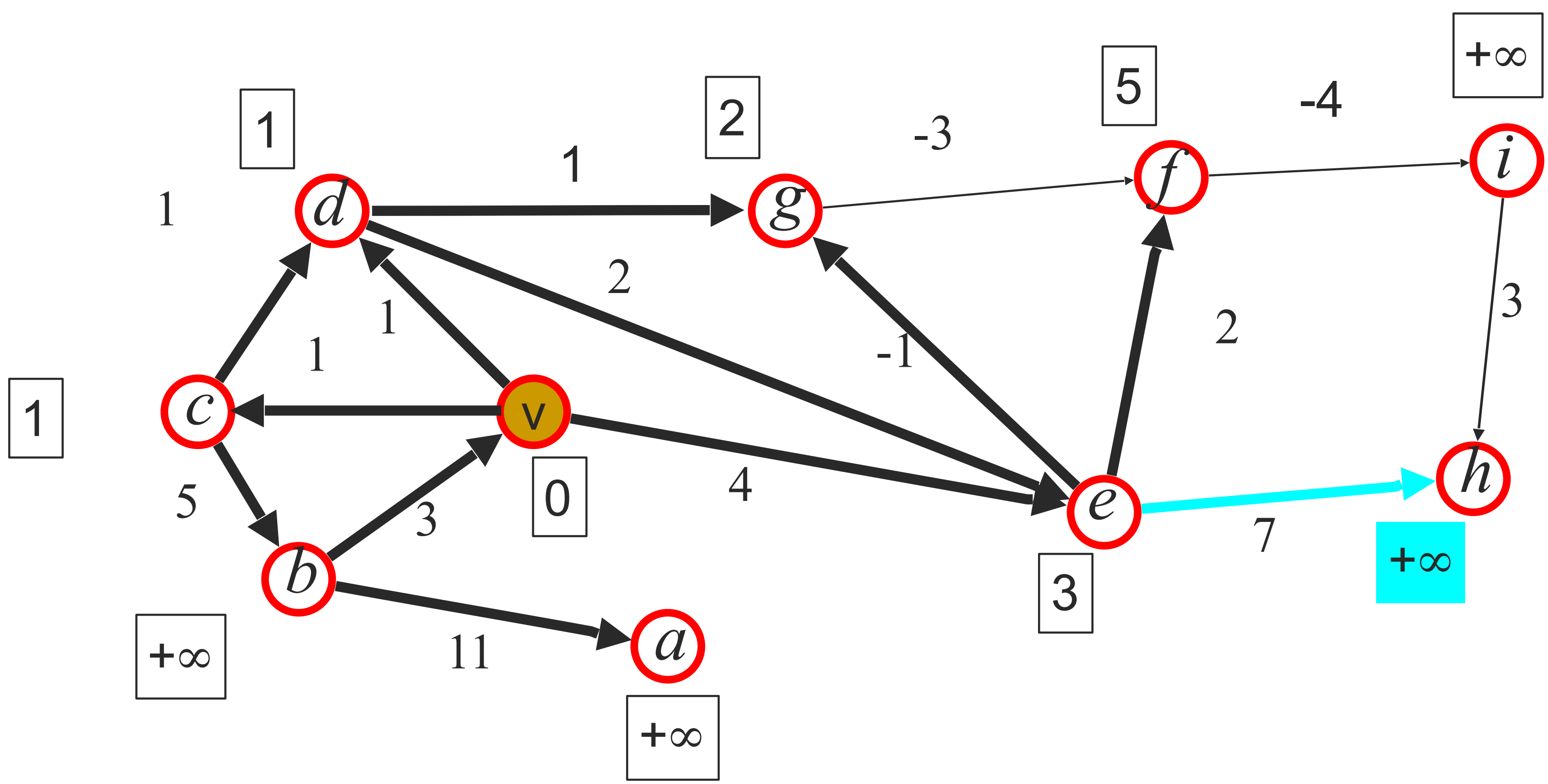
i=1



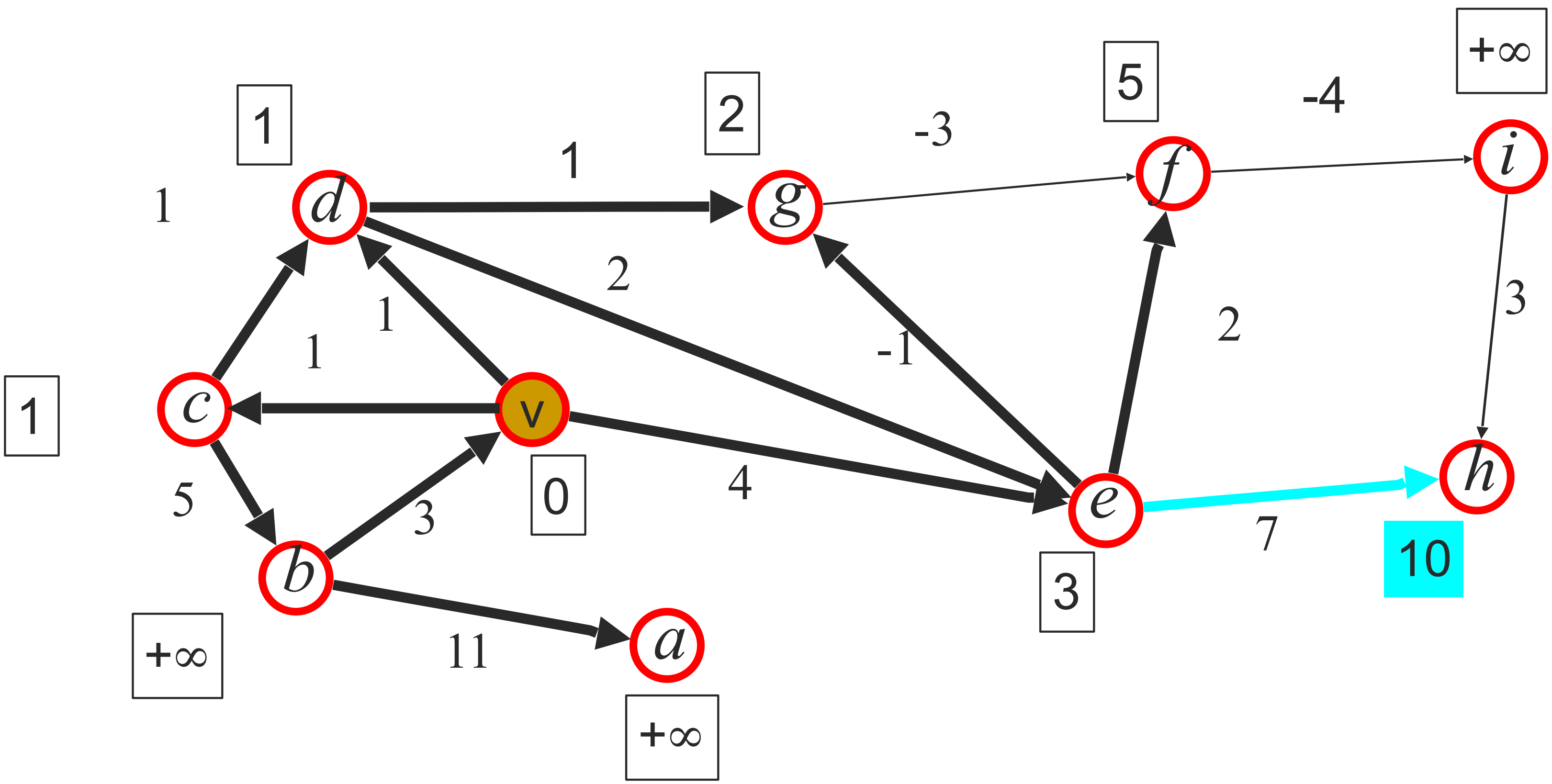
i=1



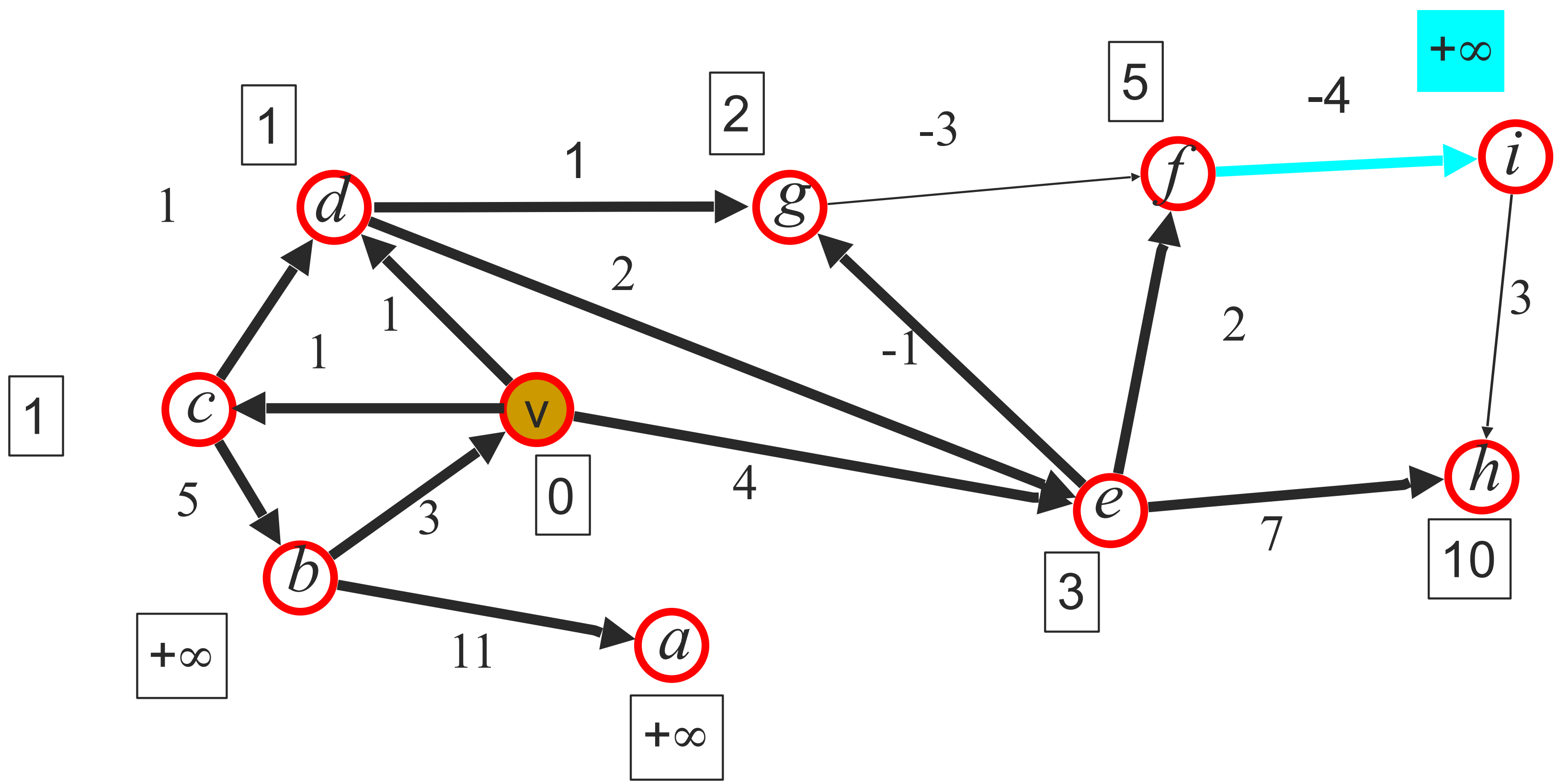
i=1



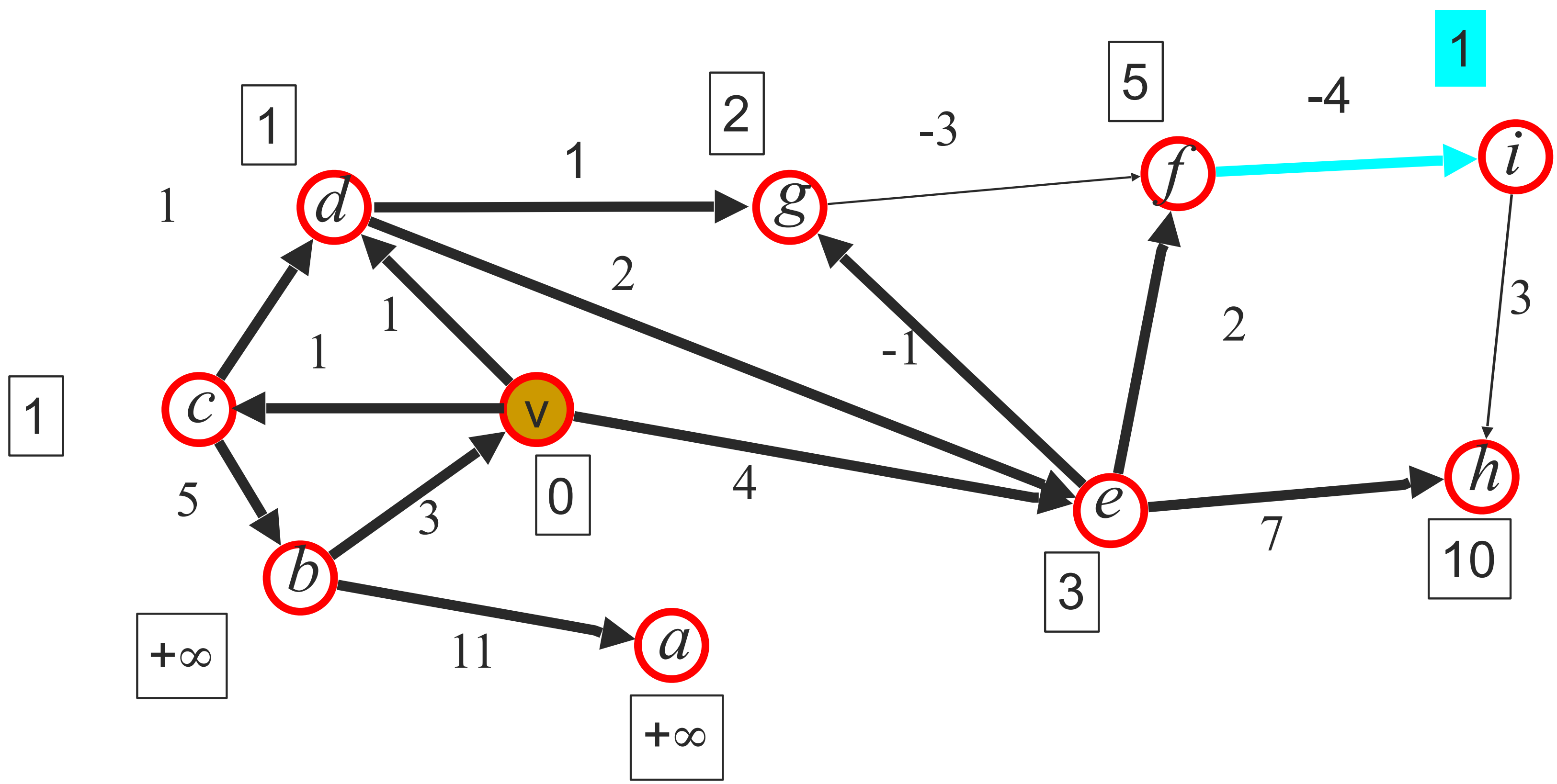
i=1



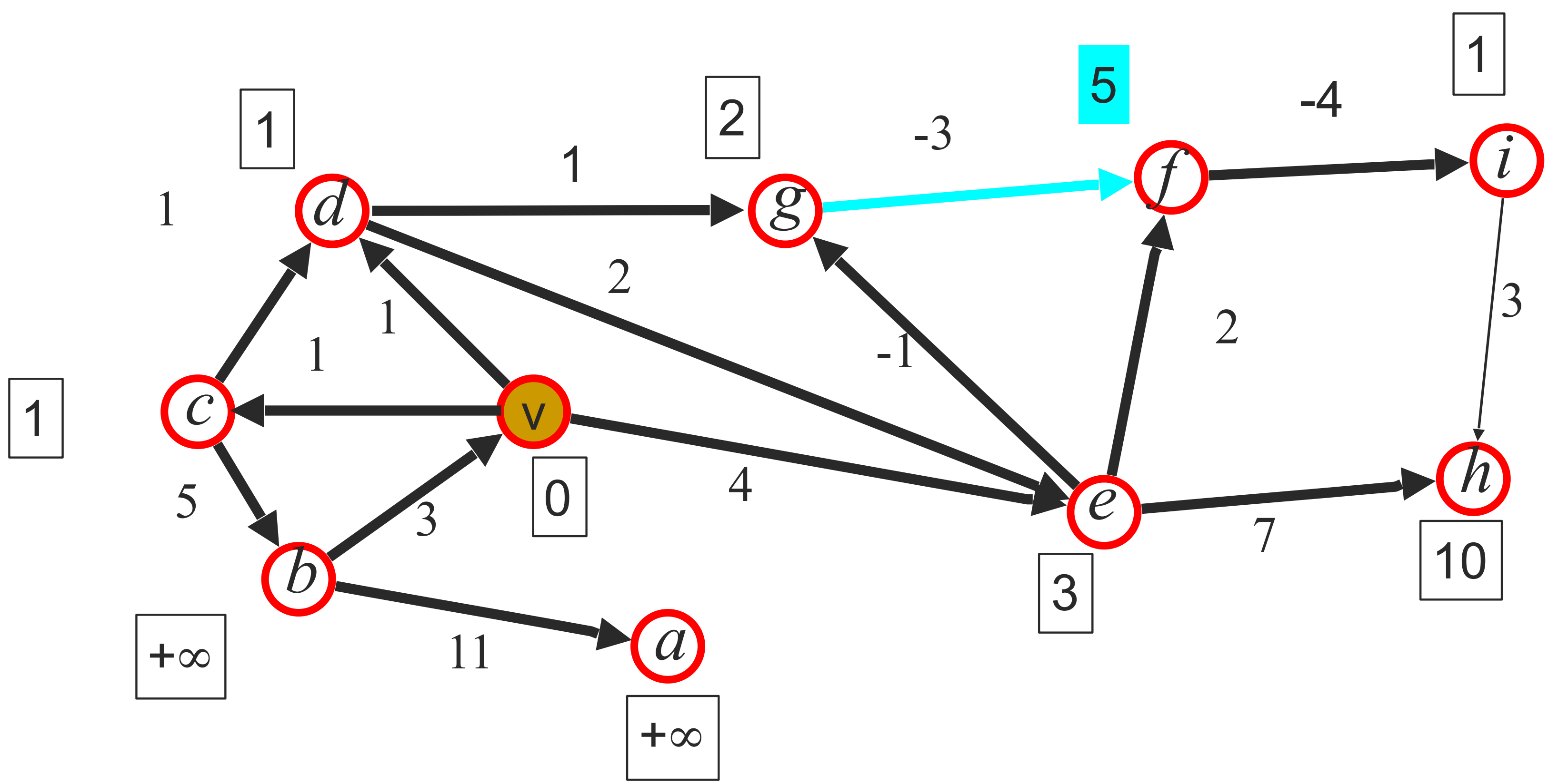
i=1



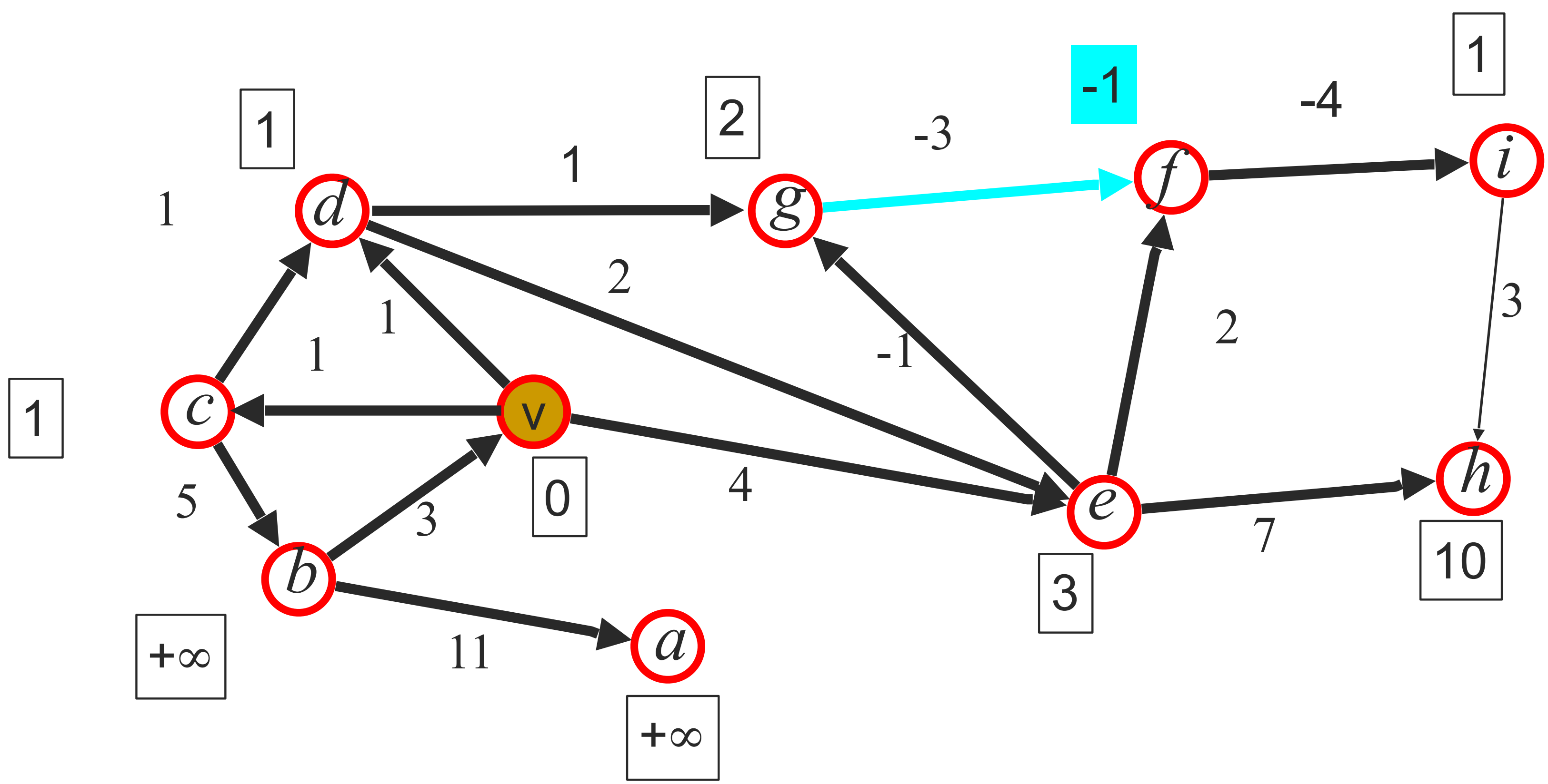
i=1



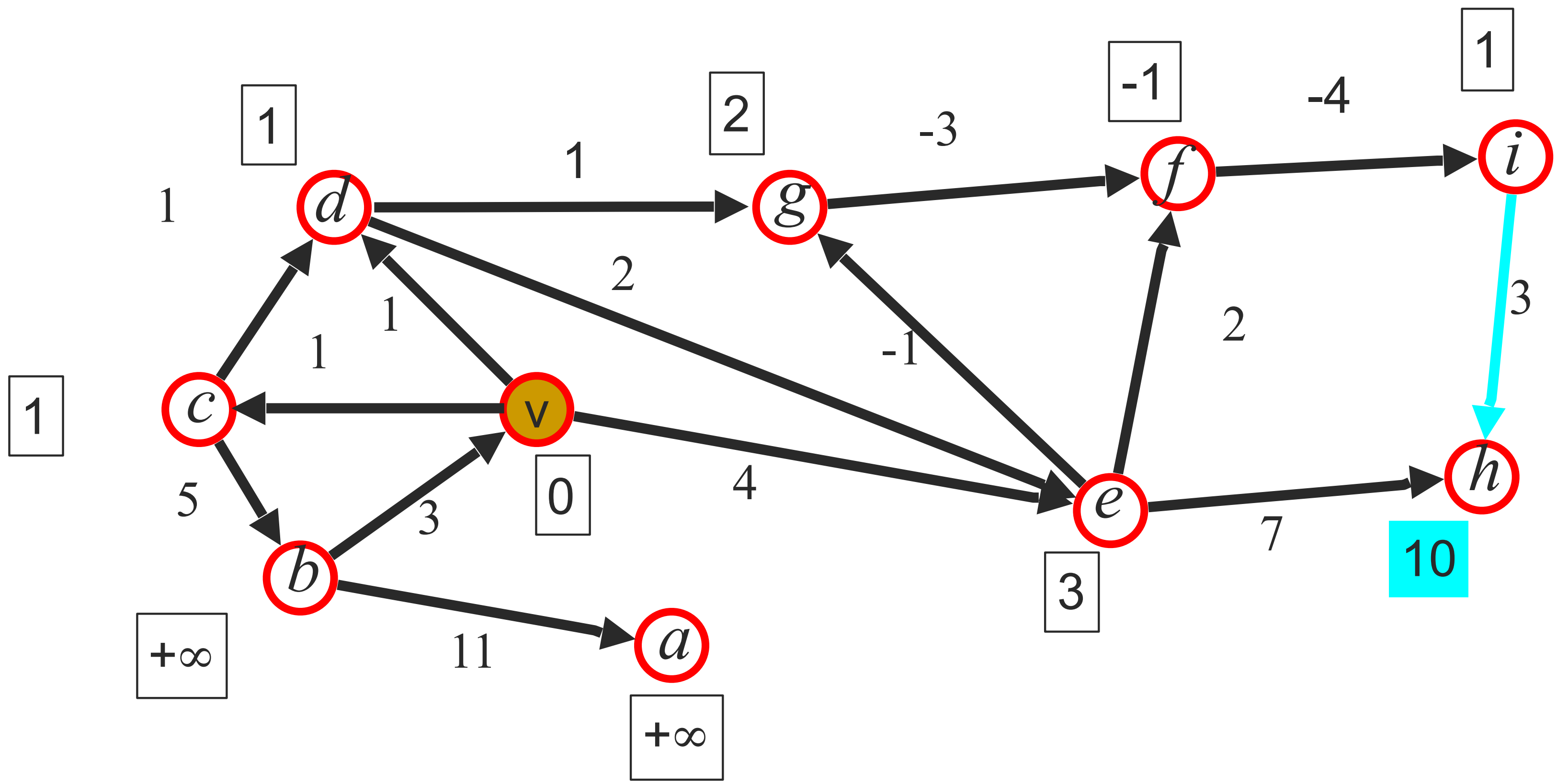
i=1



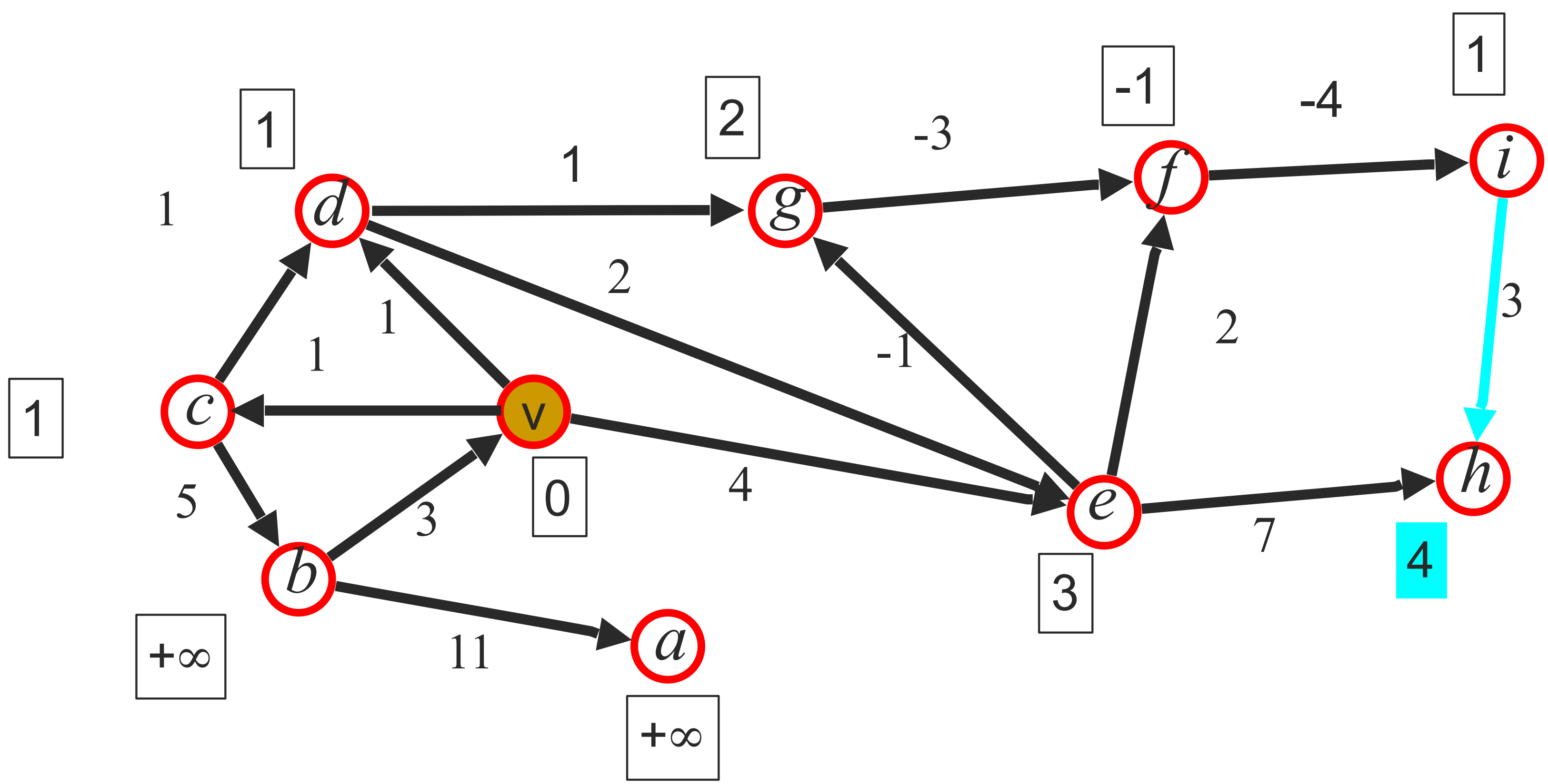
i=1



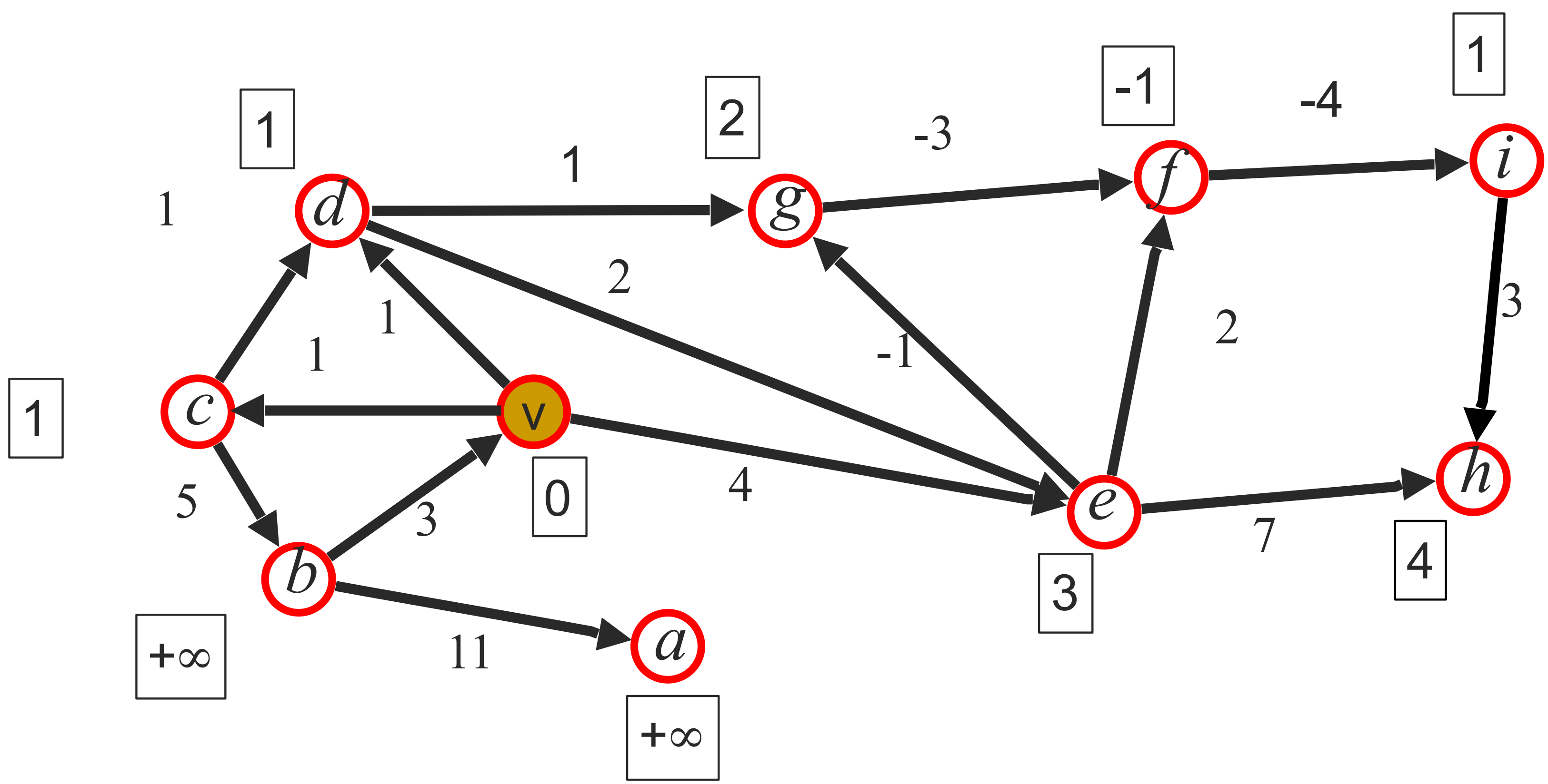
i=1



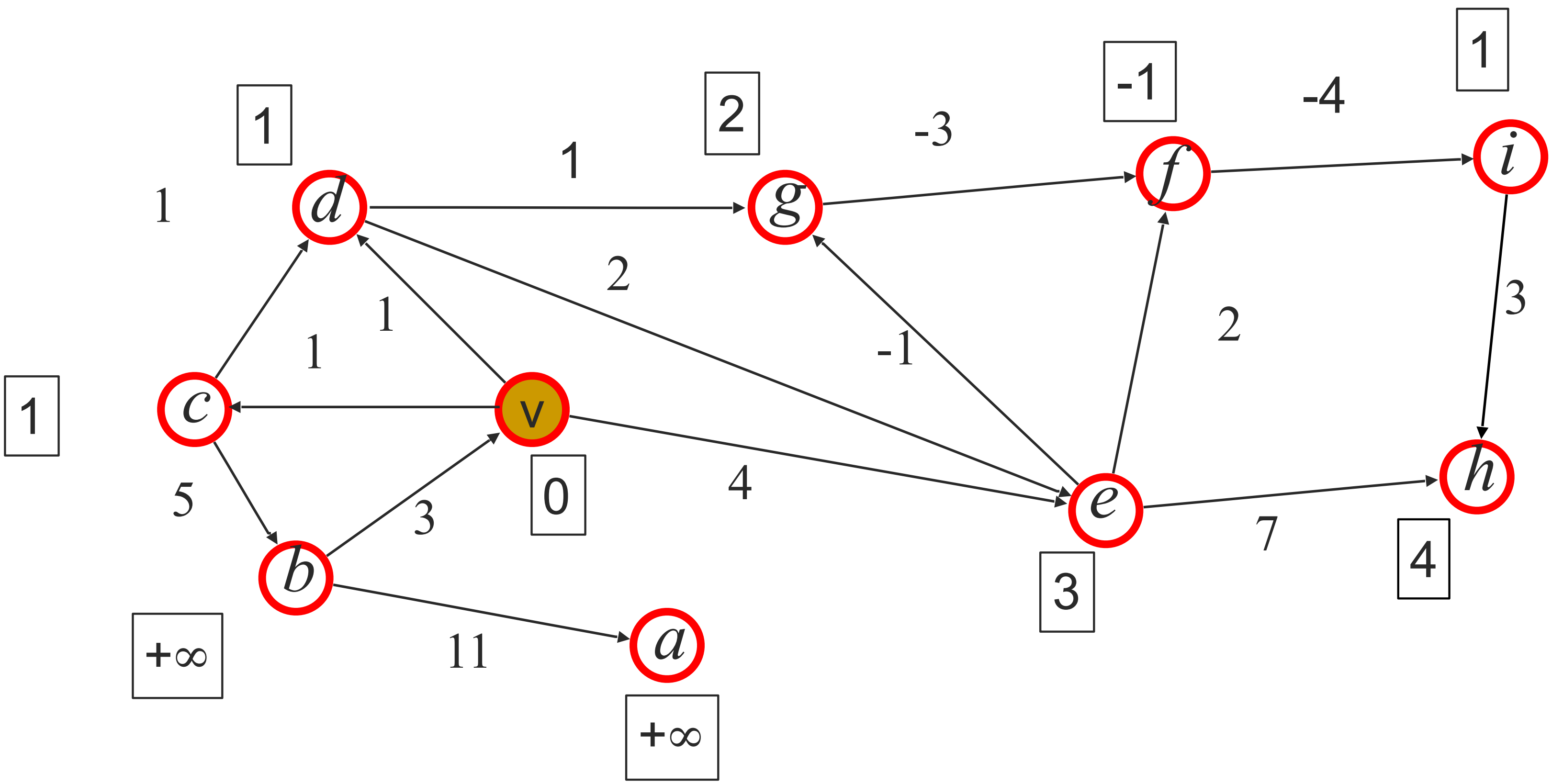
i=1



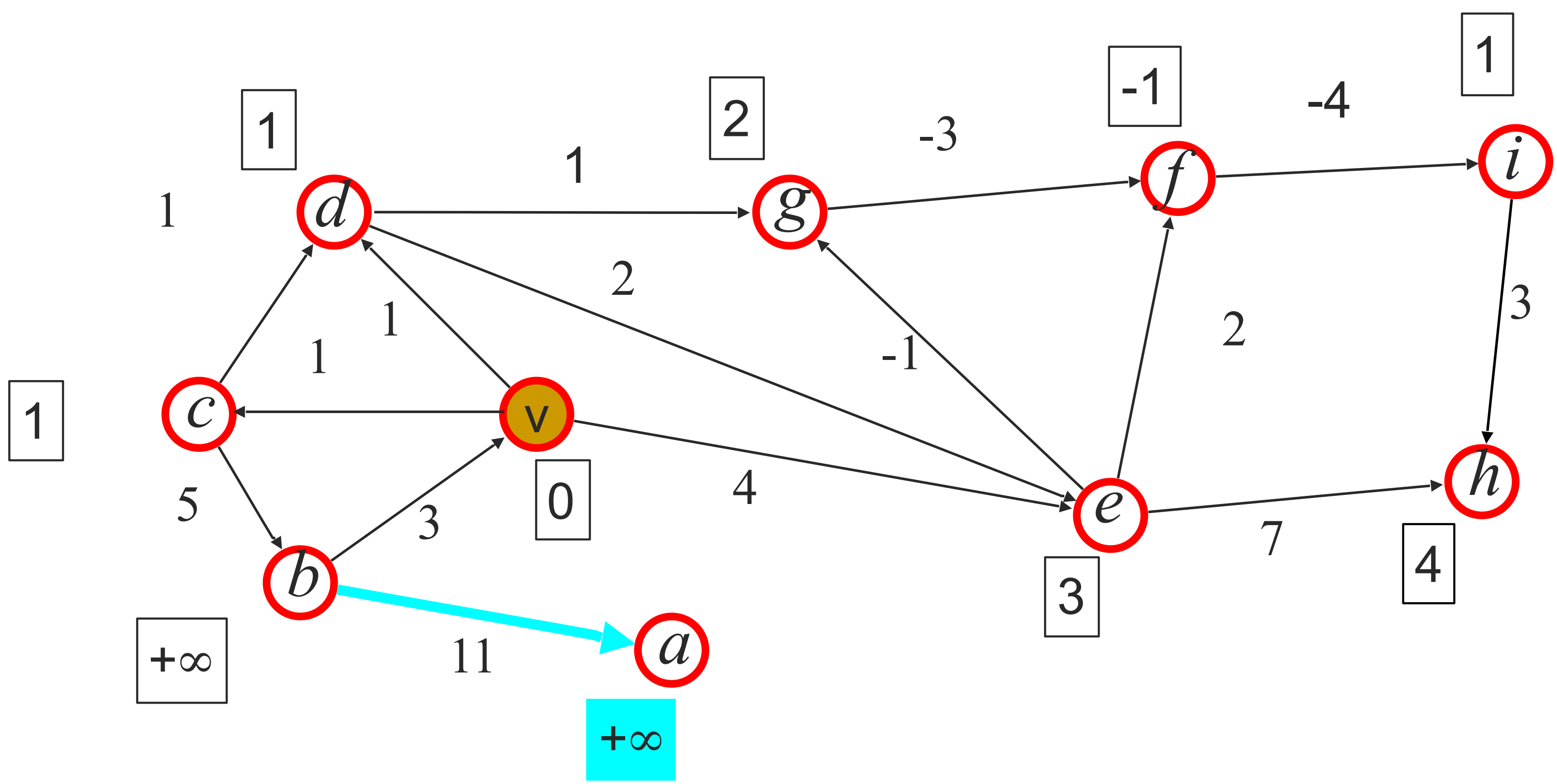
i=1



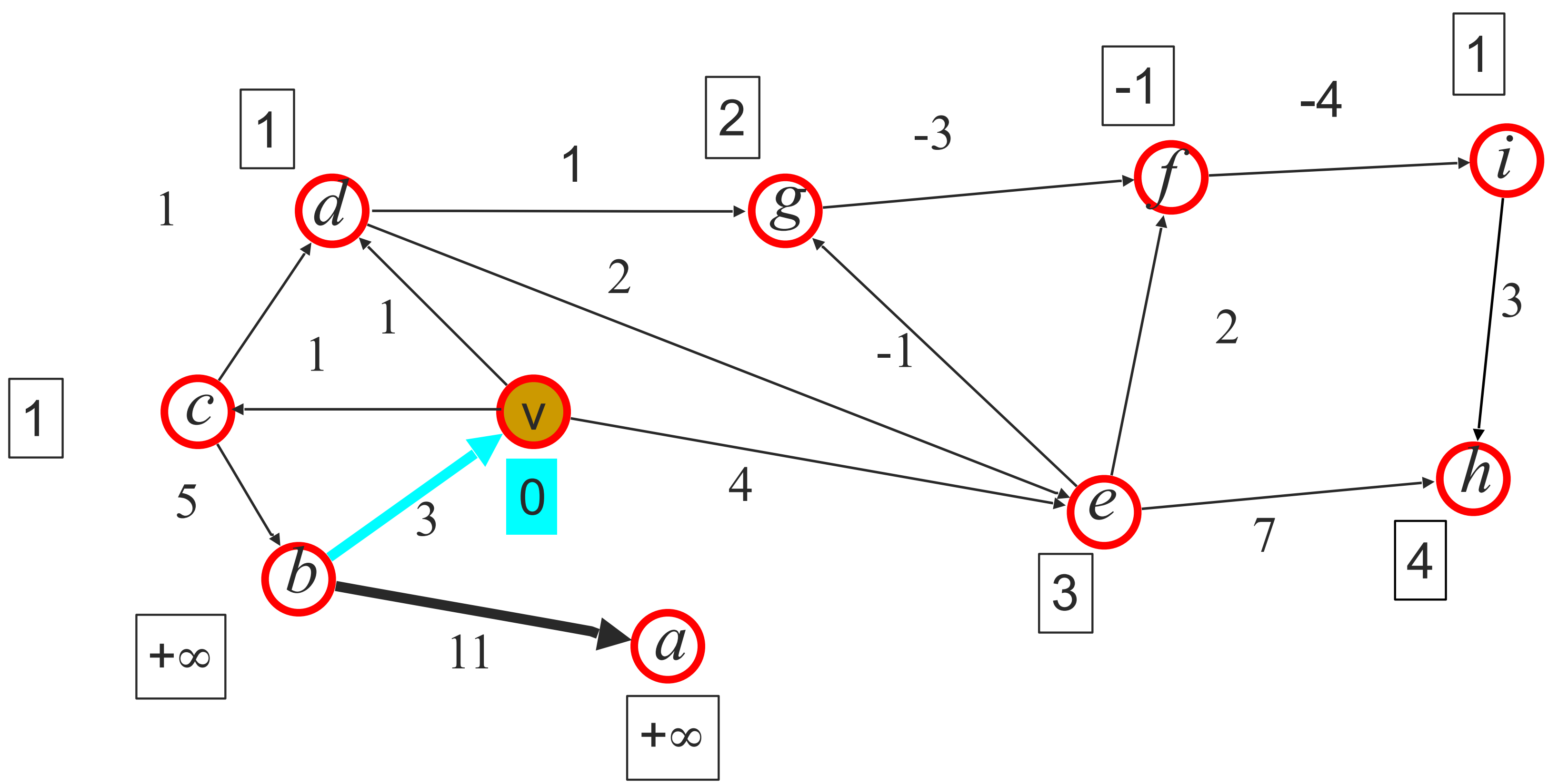
i=1



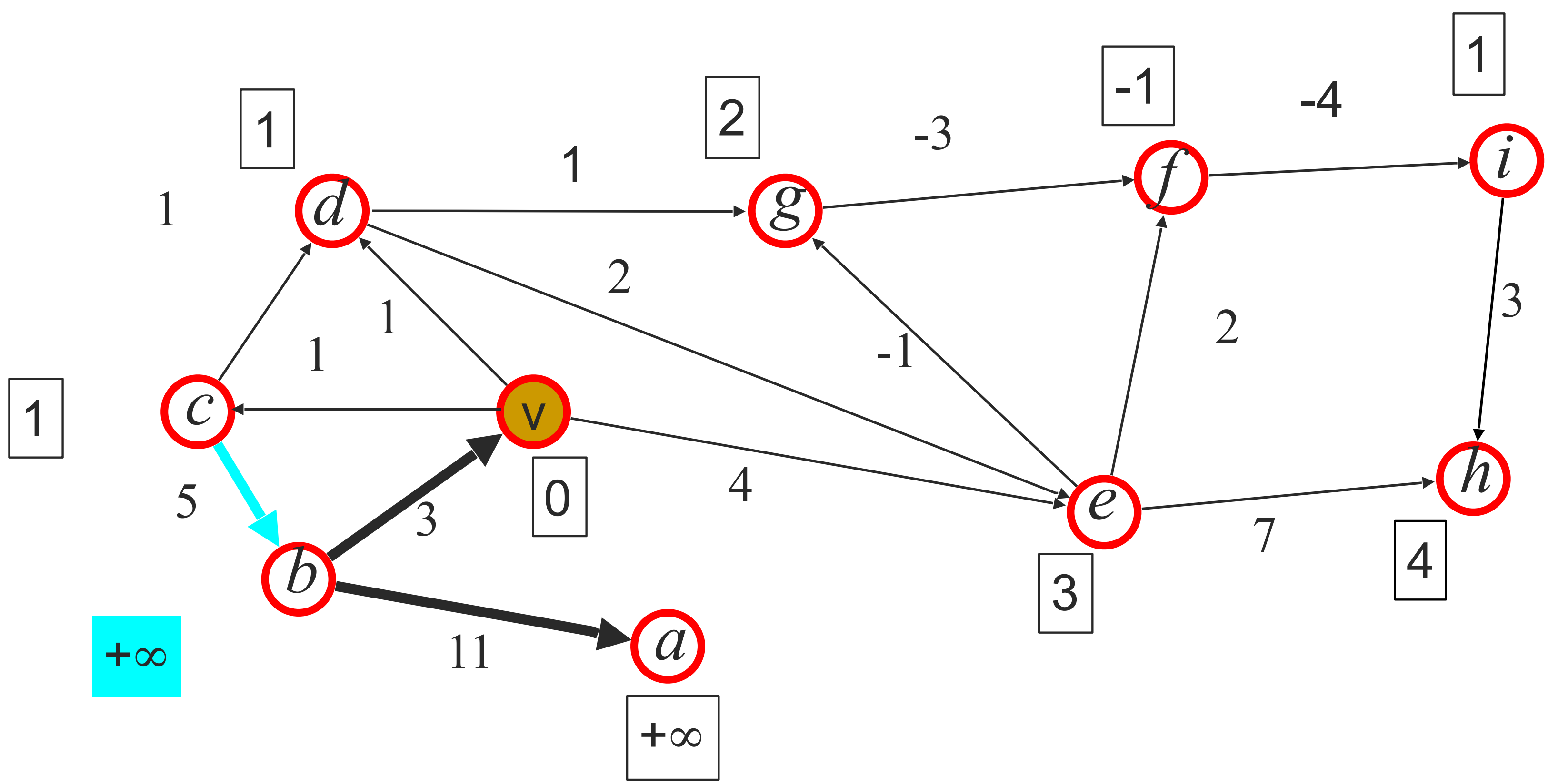
i=2



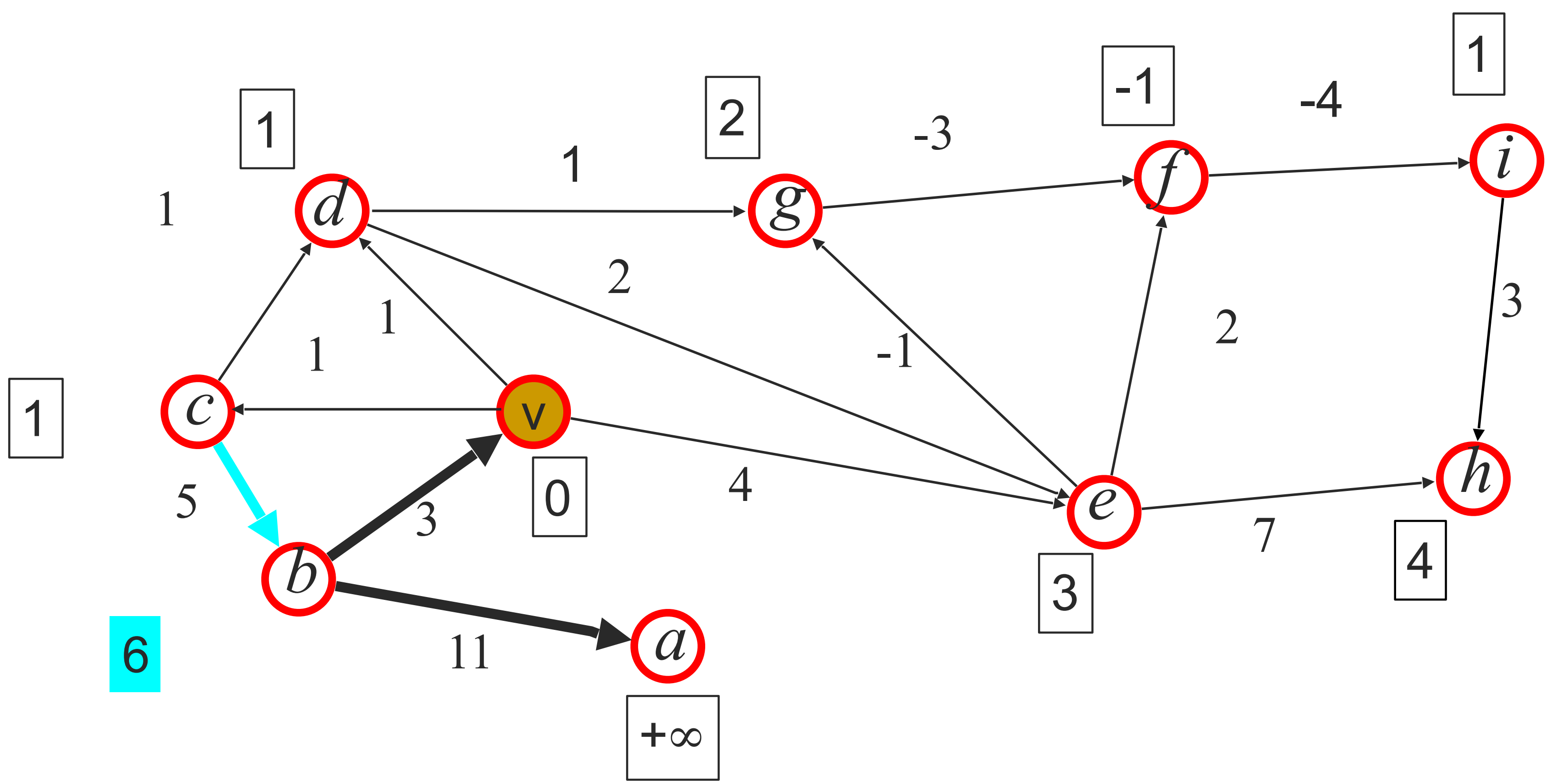
i=2



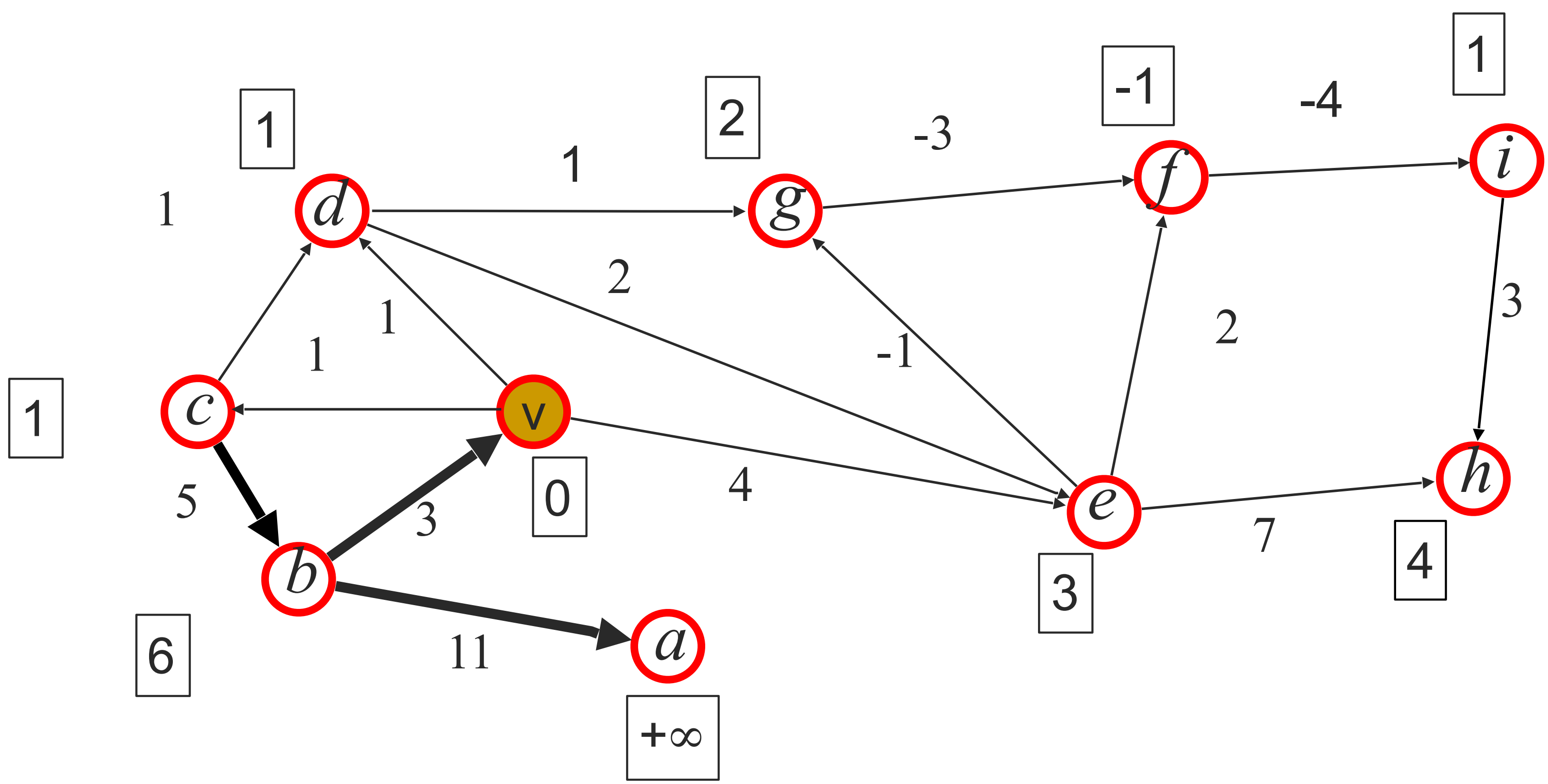
i=2



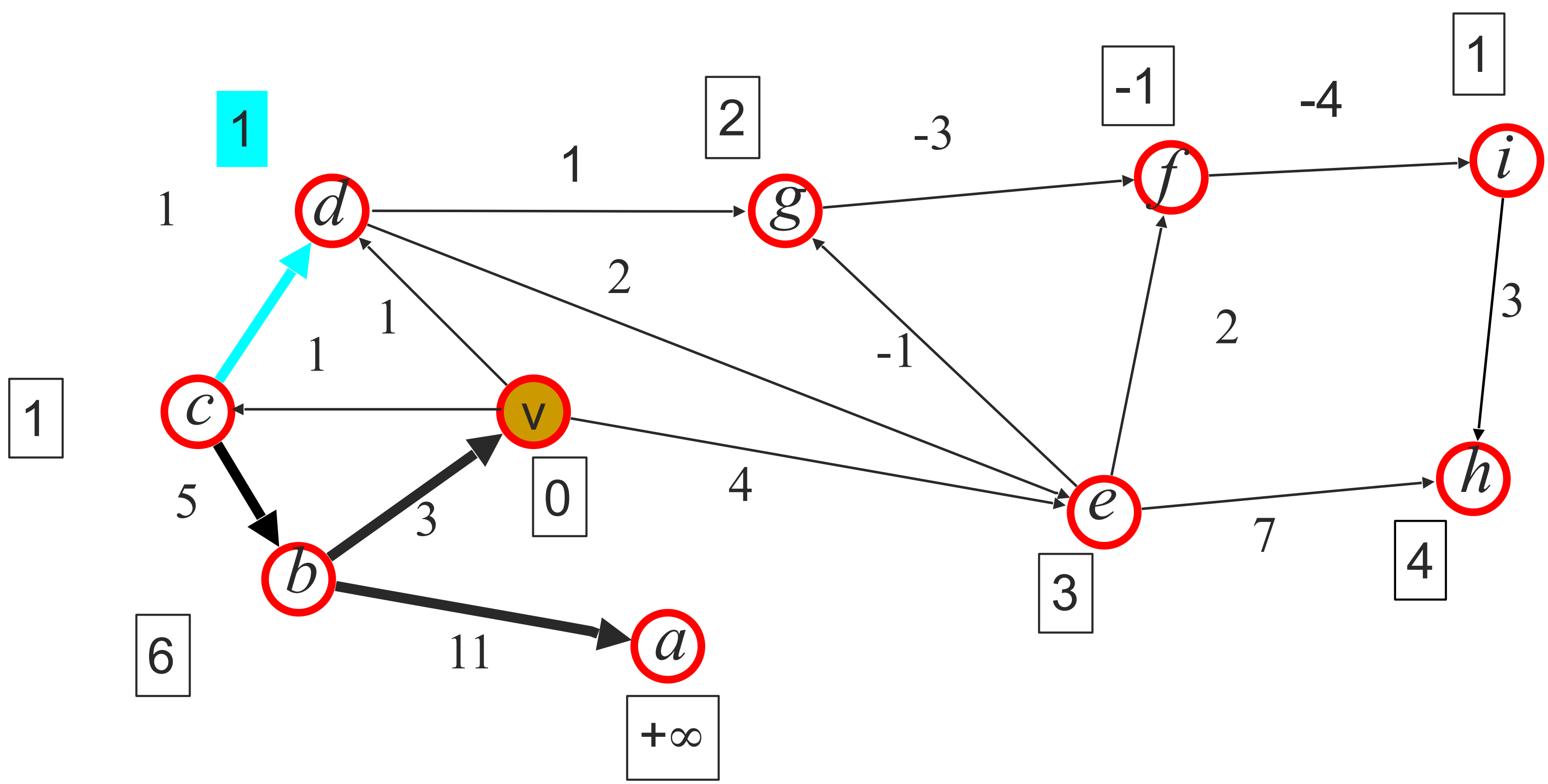
i=2



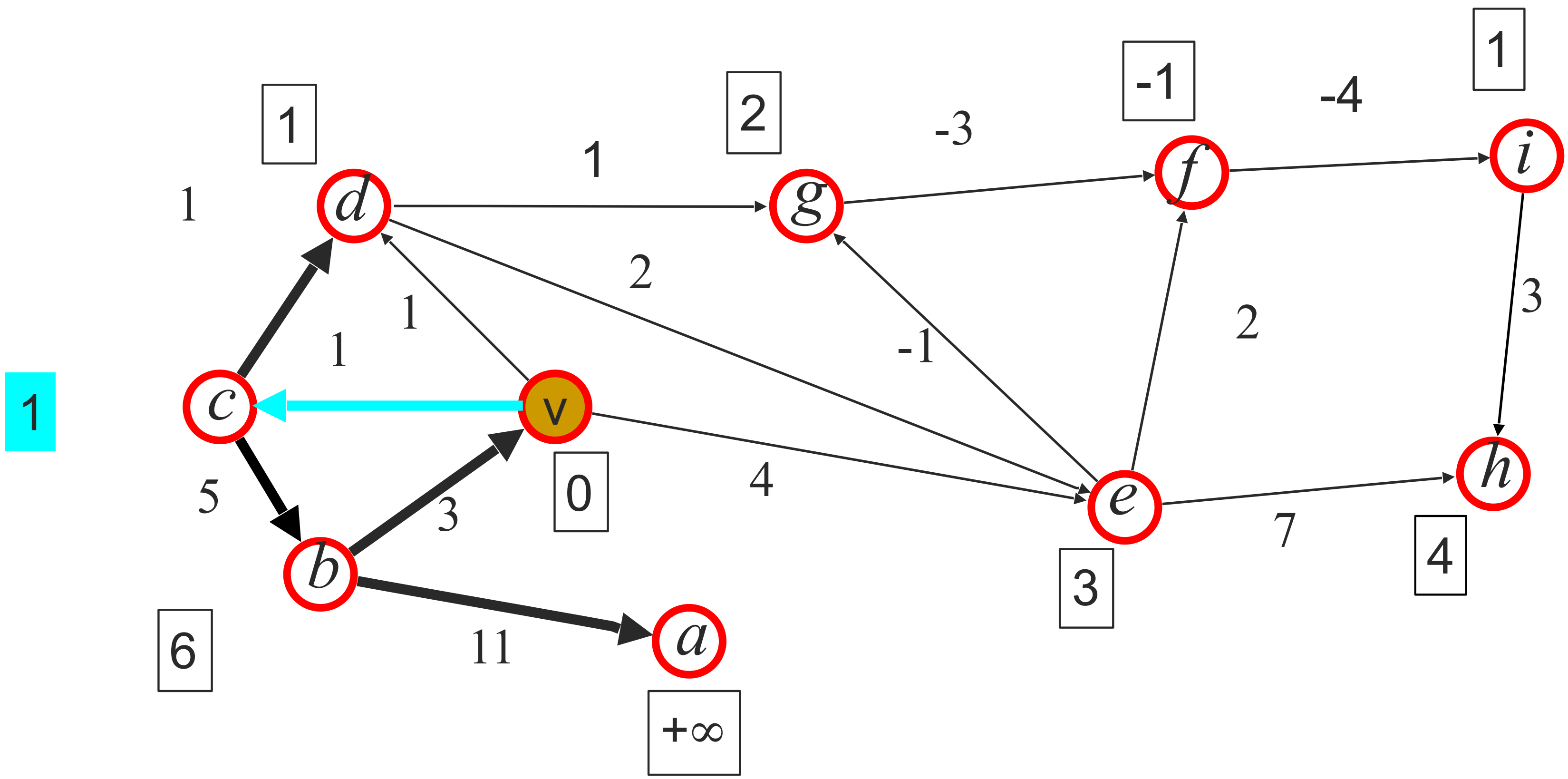
i=2



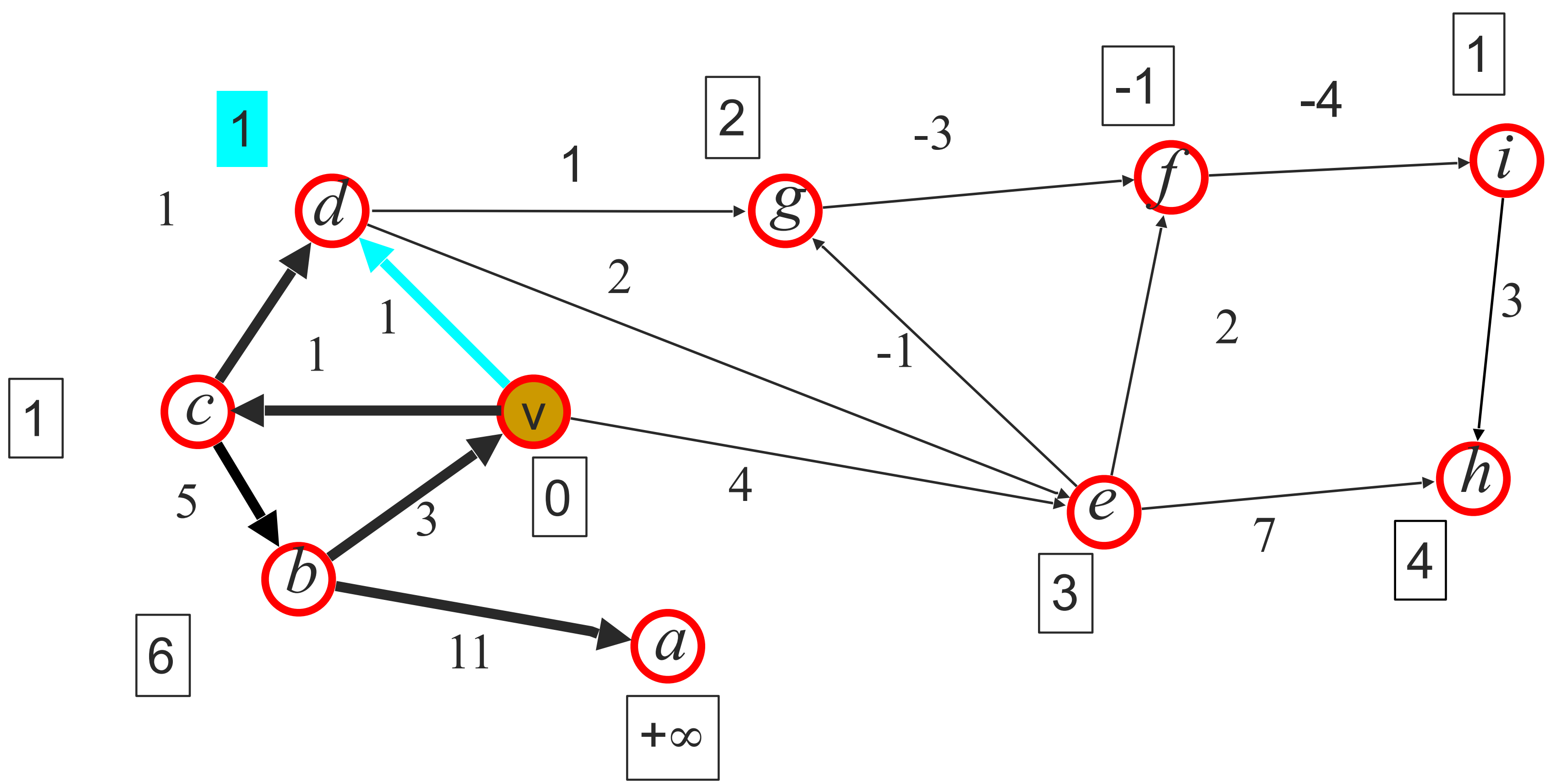
i=2



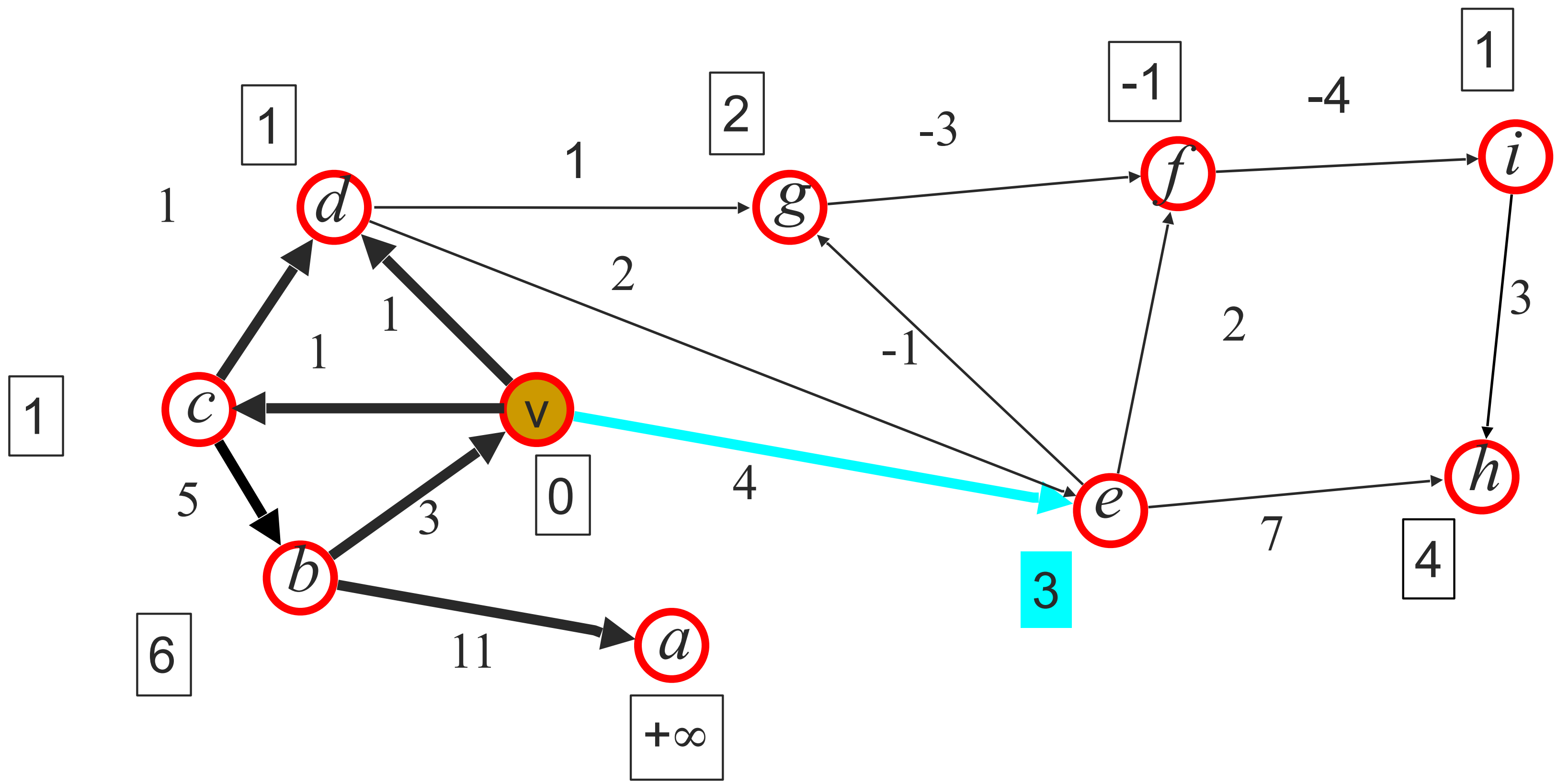
i=2



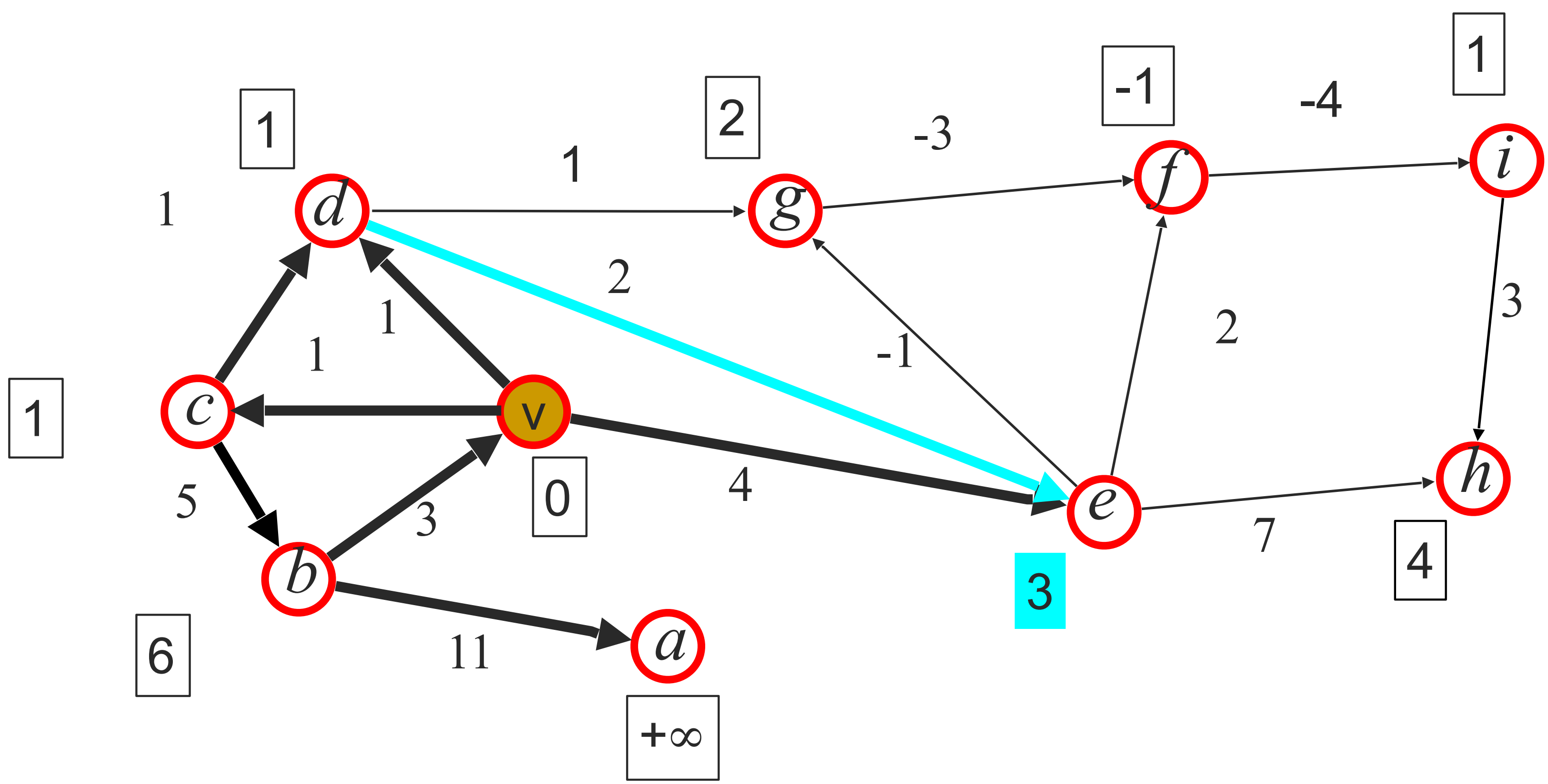
i=2



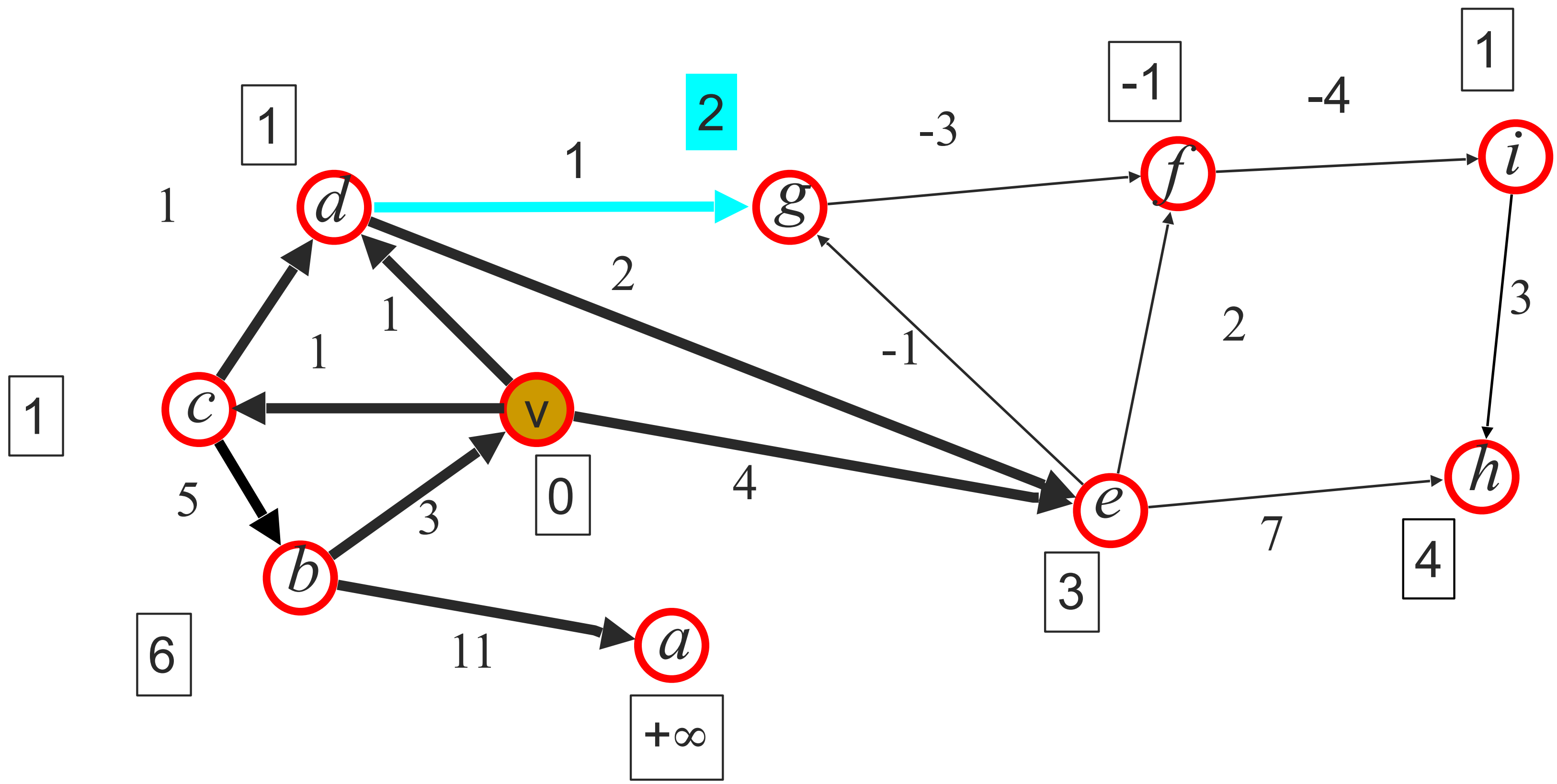
i=2



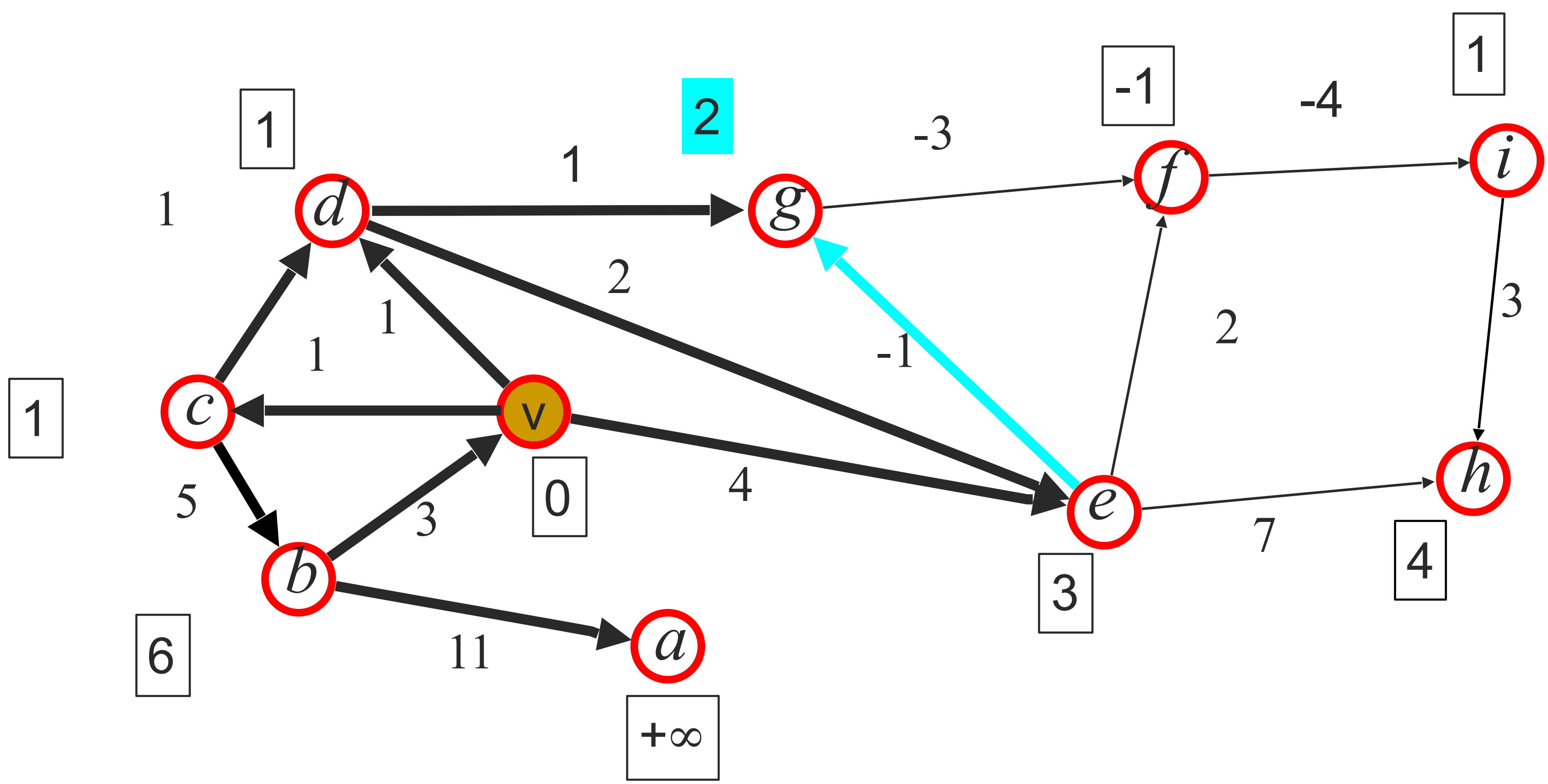
i=2



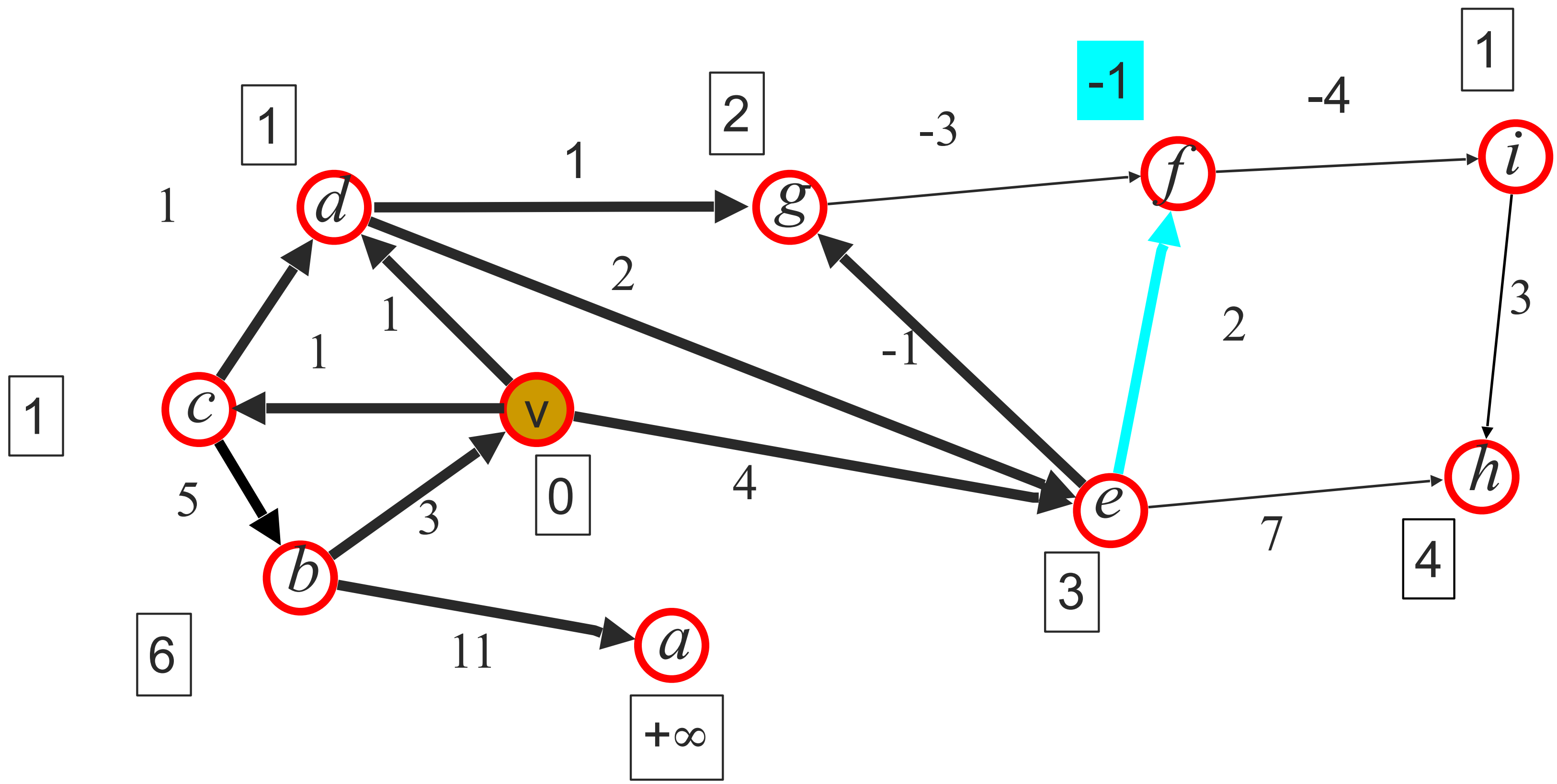
i=2



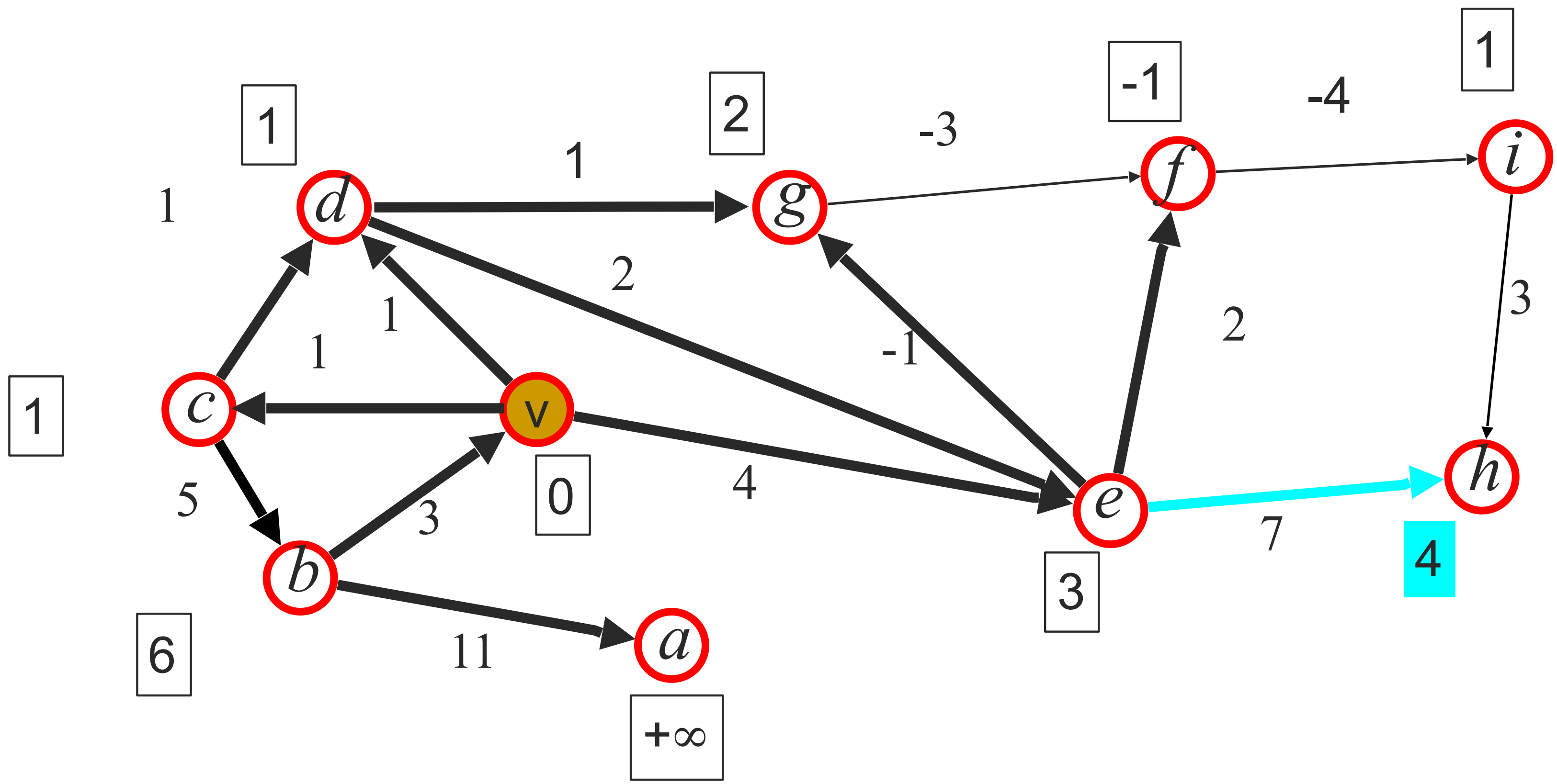
i=2



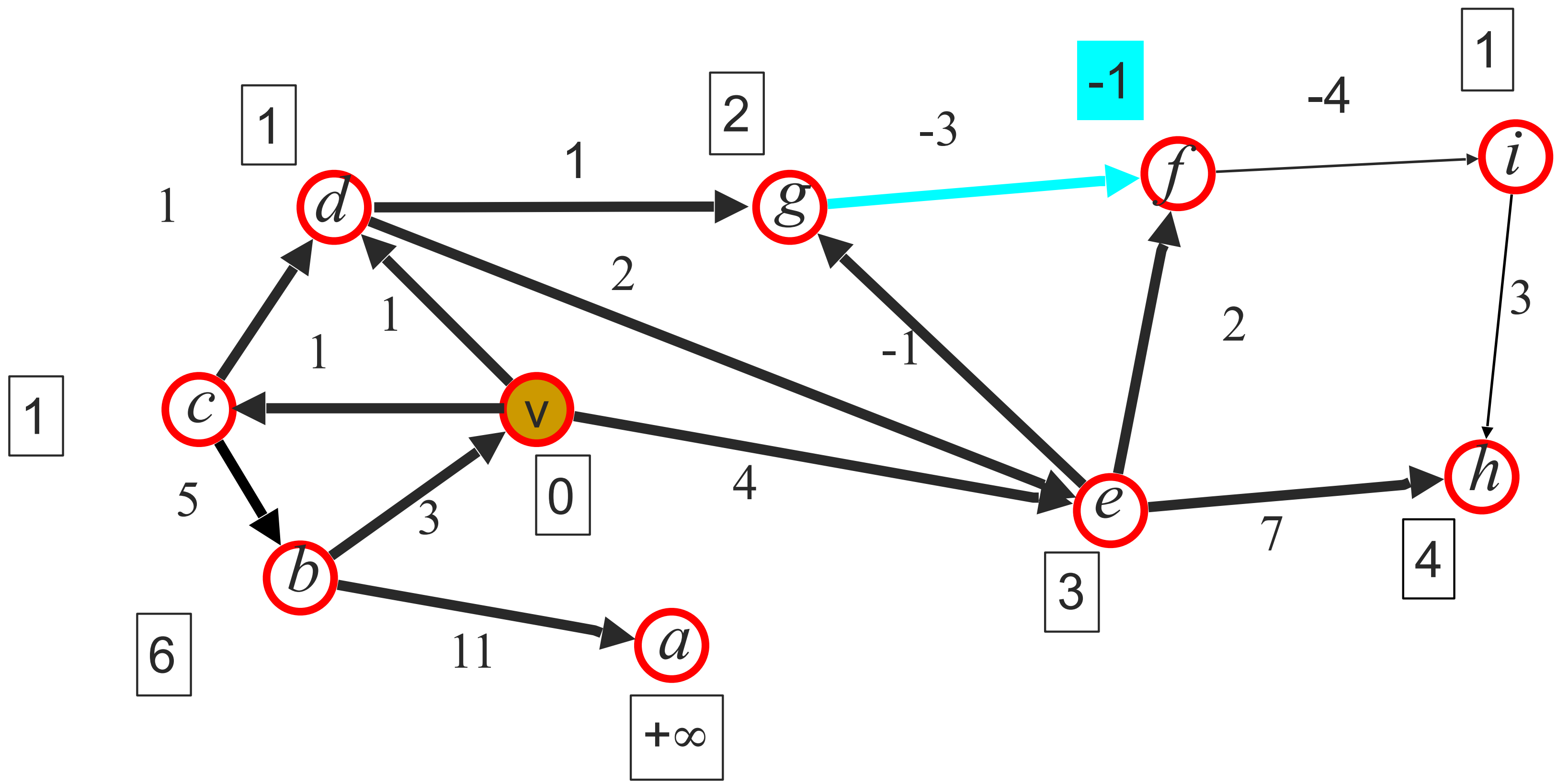
i=2



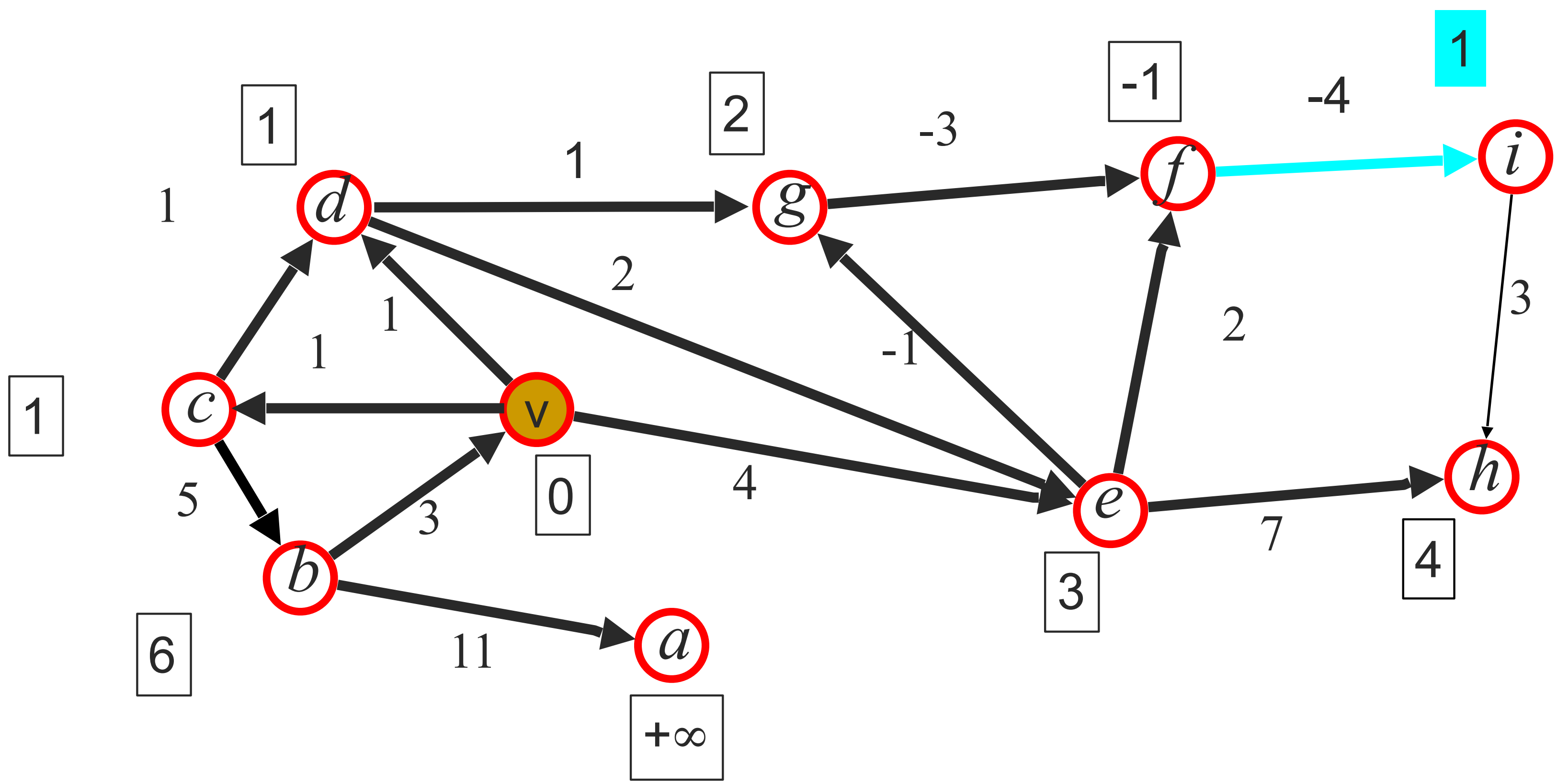
i=2



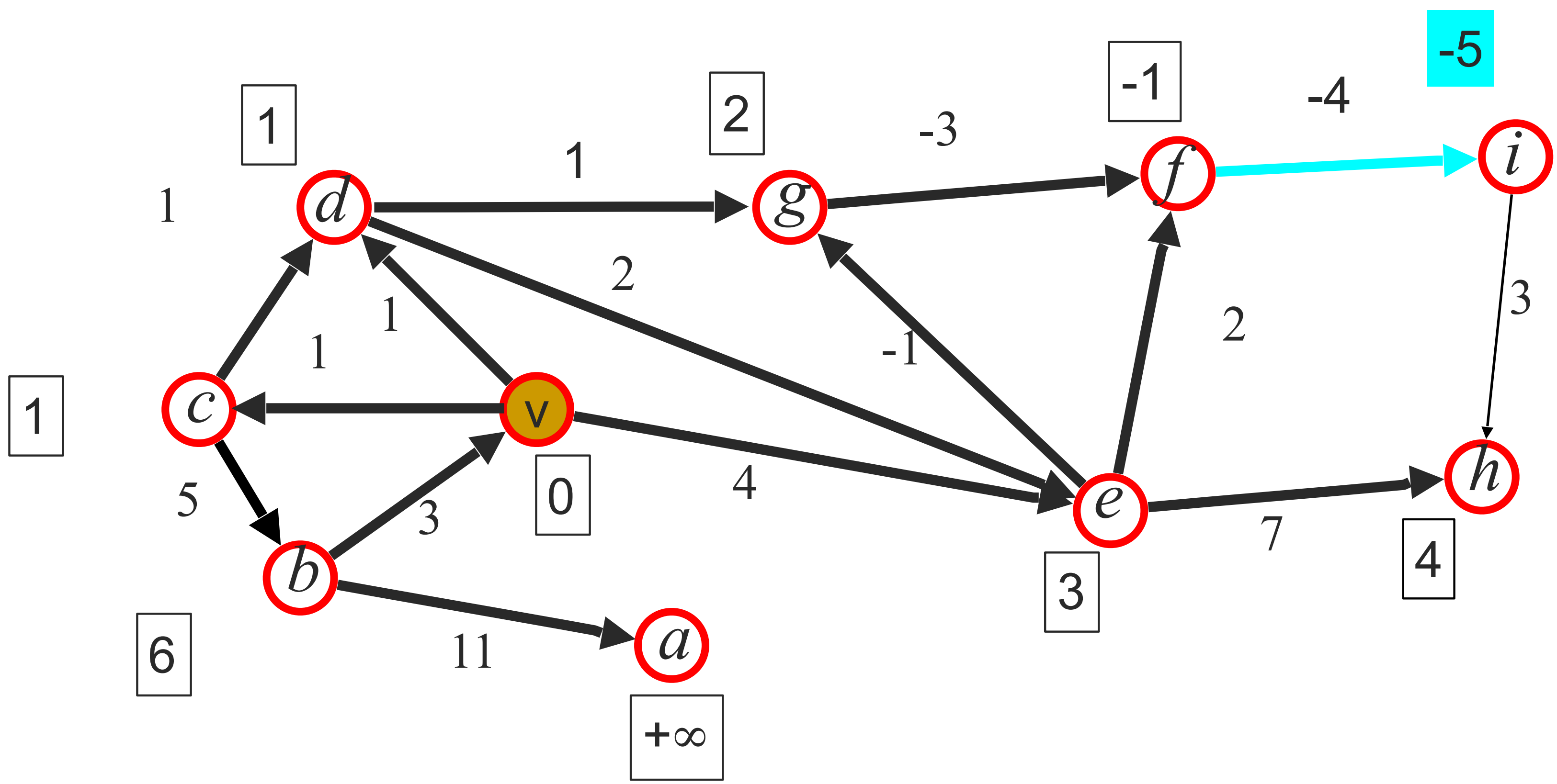
i=2



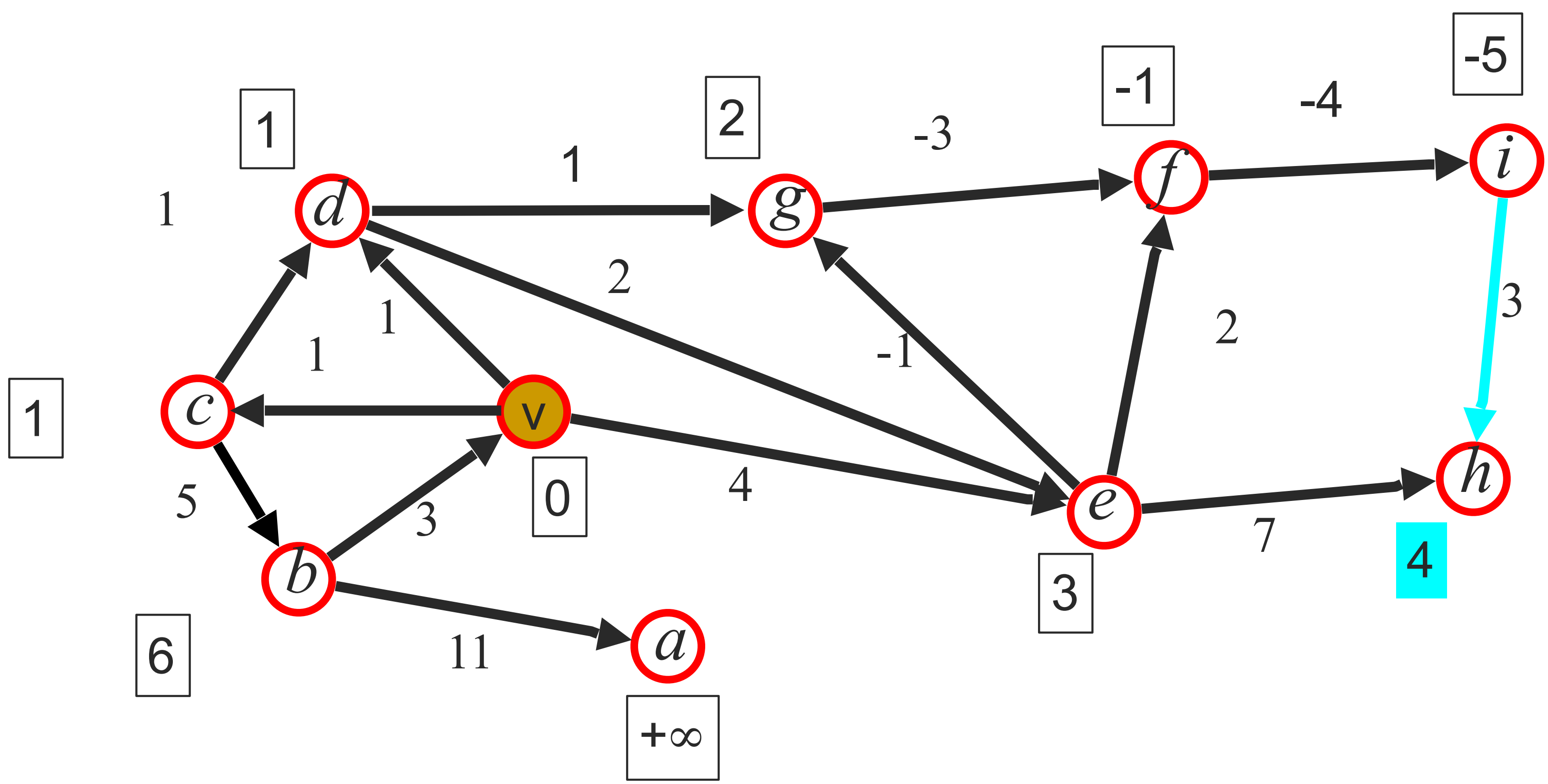
i=2



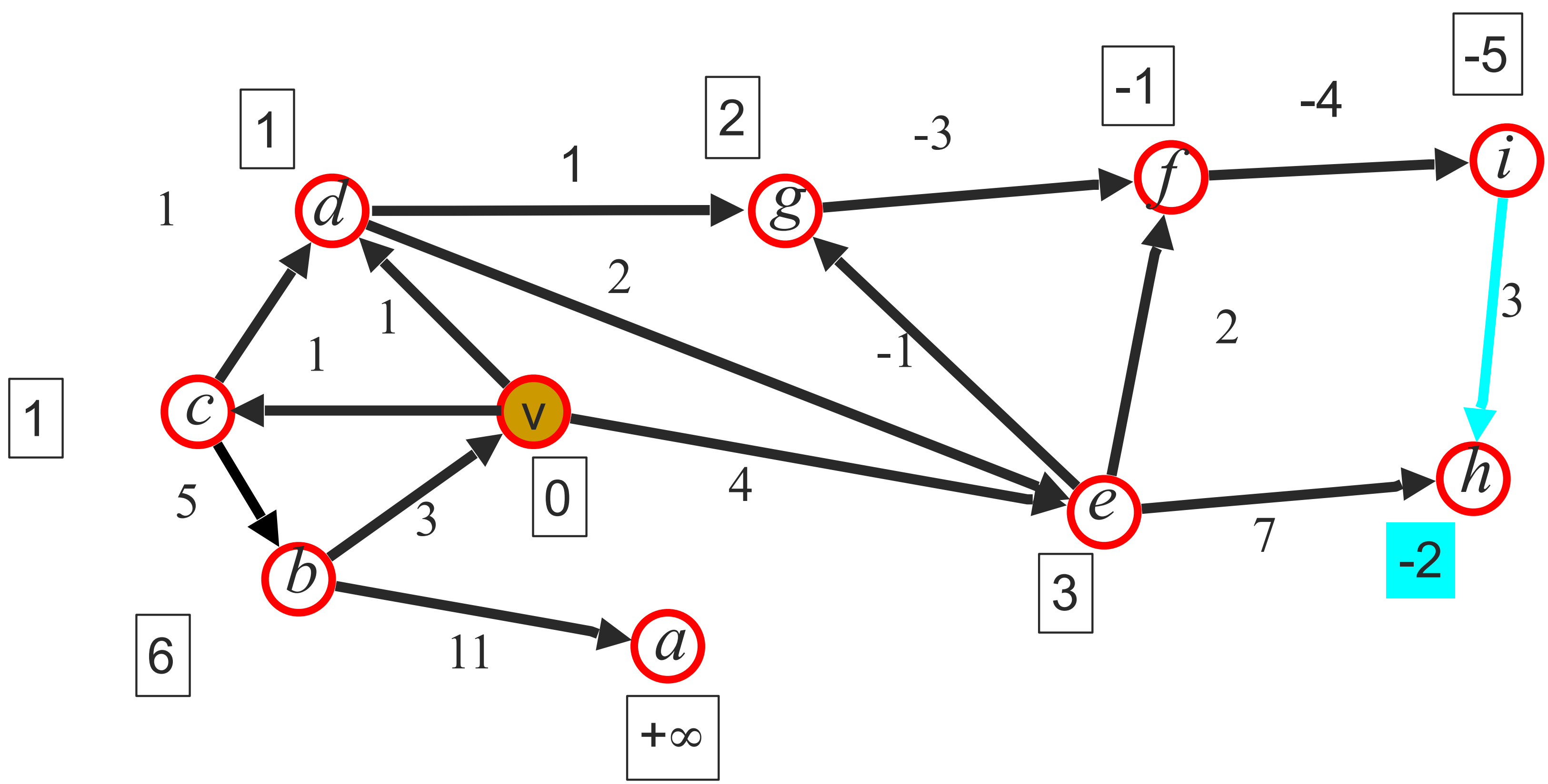
i=2



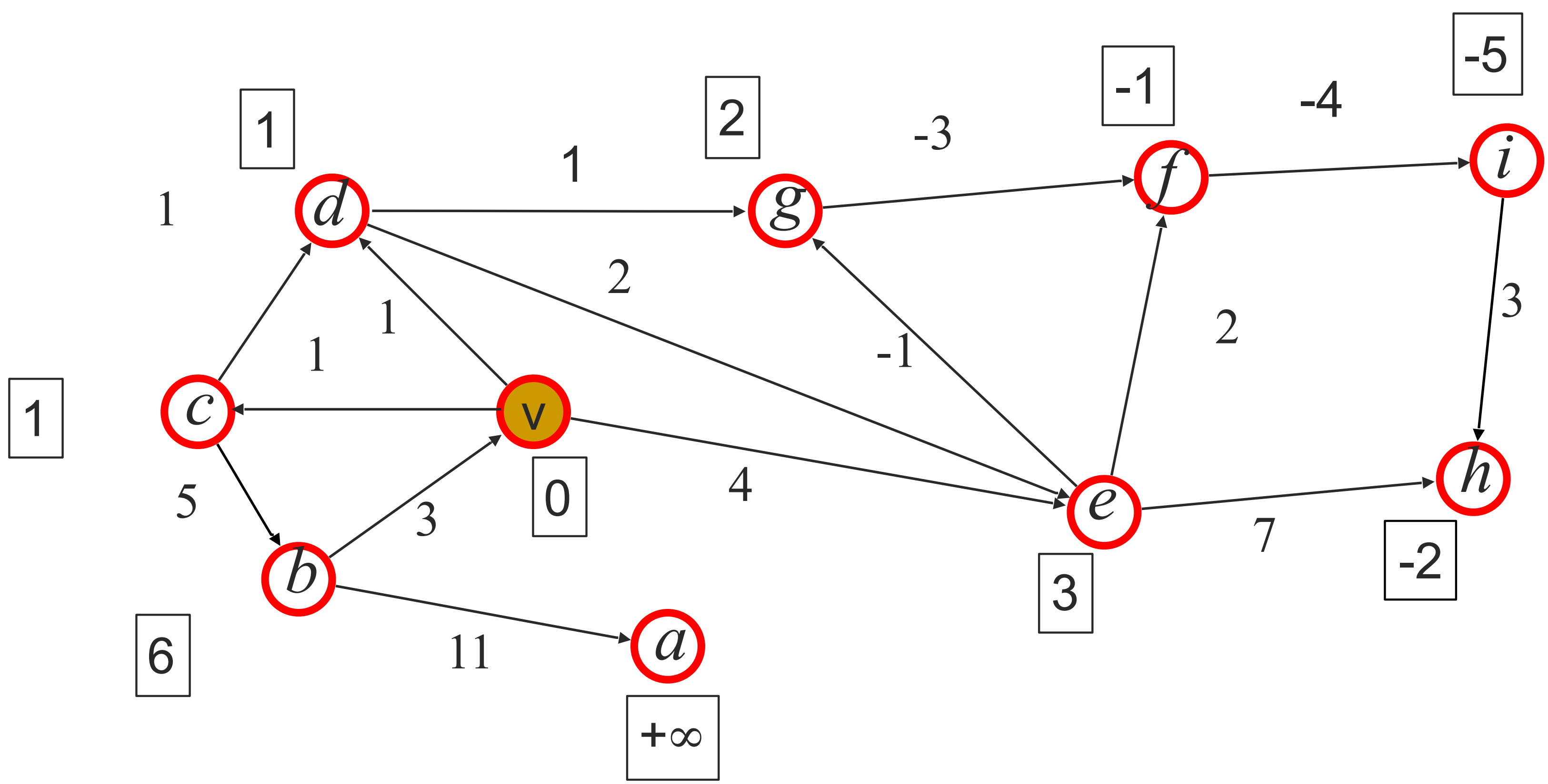
i=2



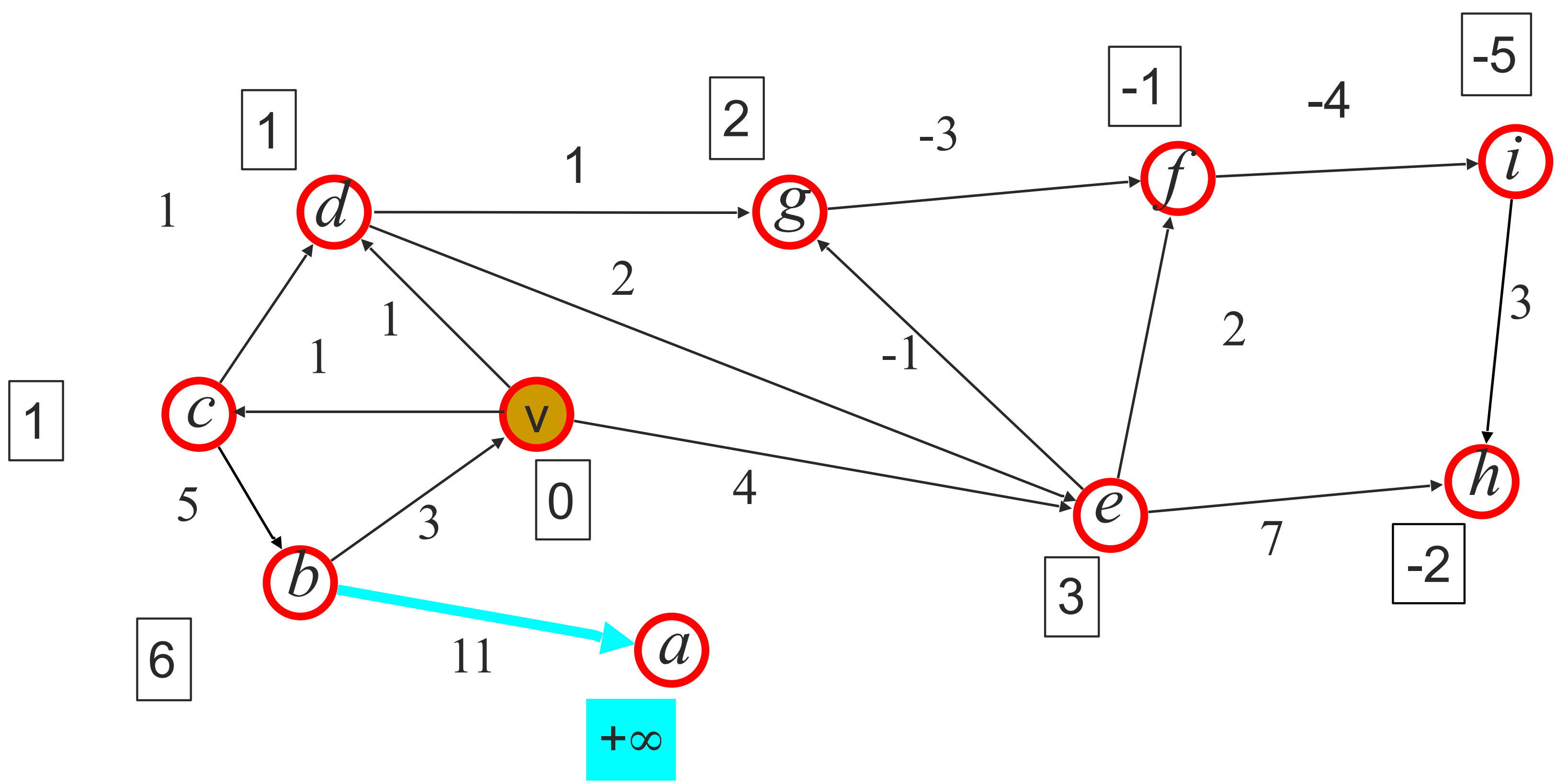
i=2



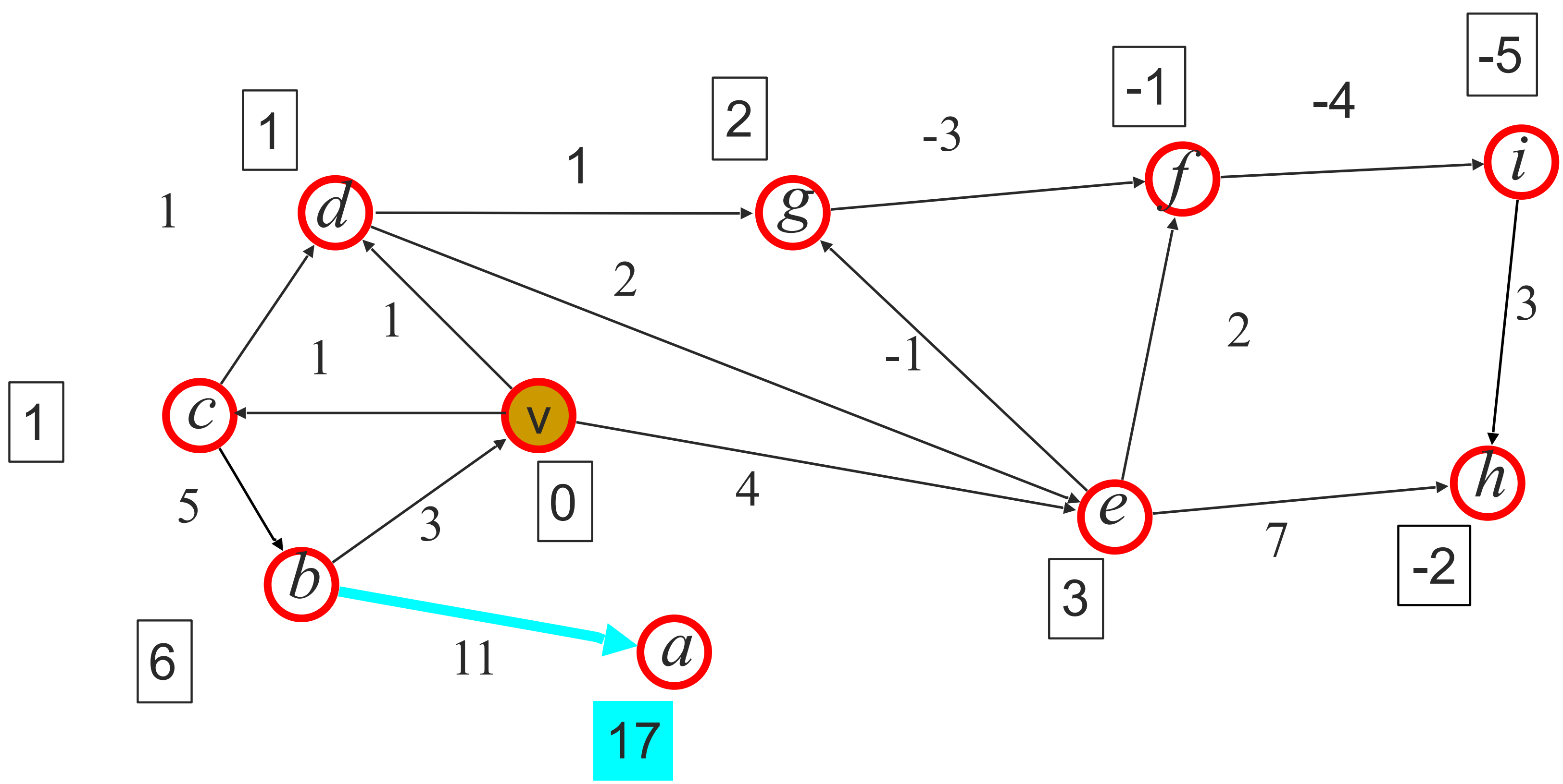
i=2



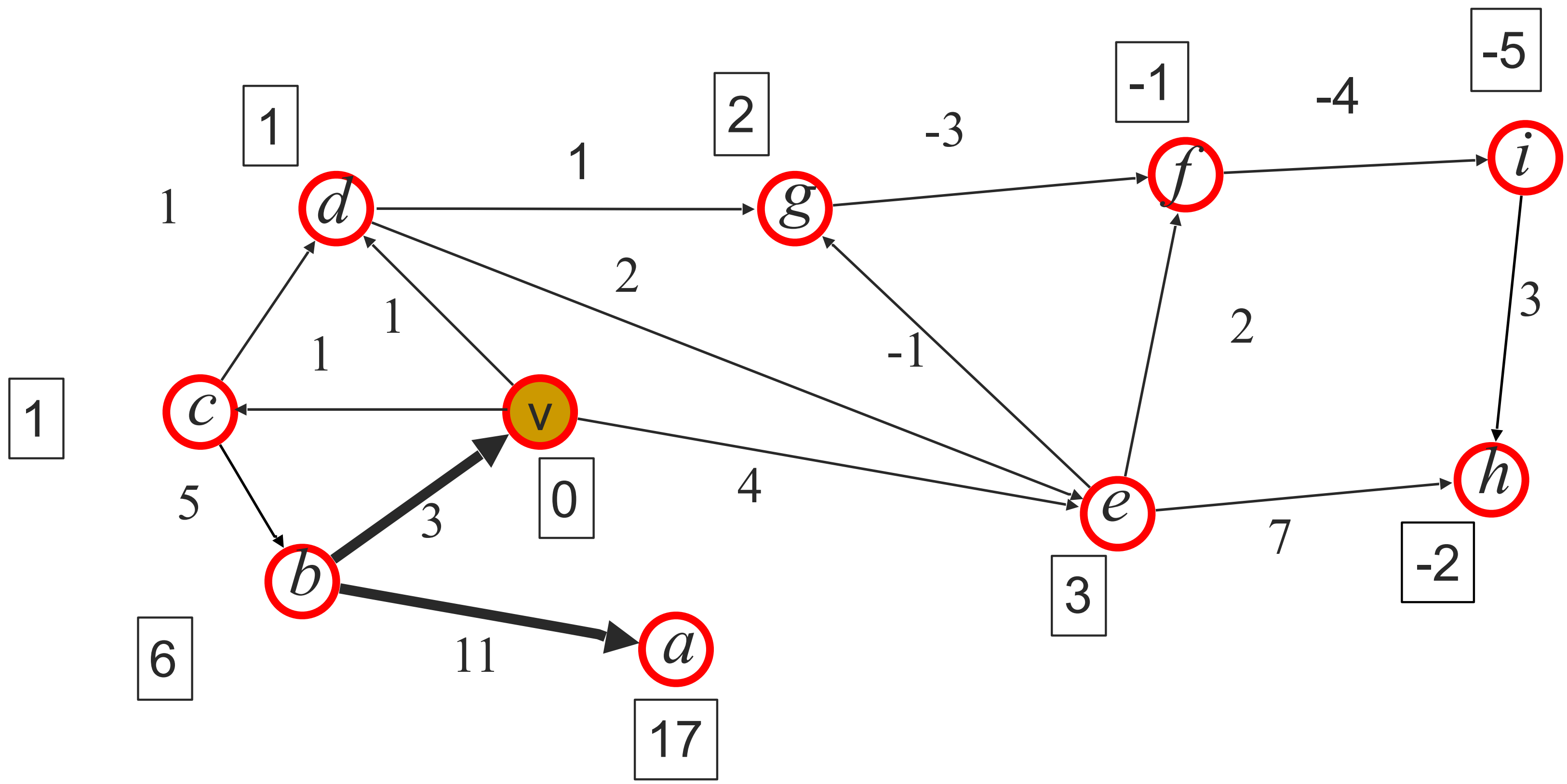
i=3



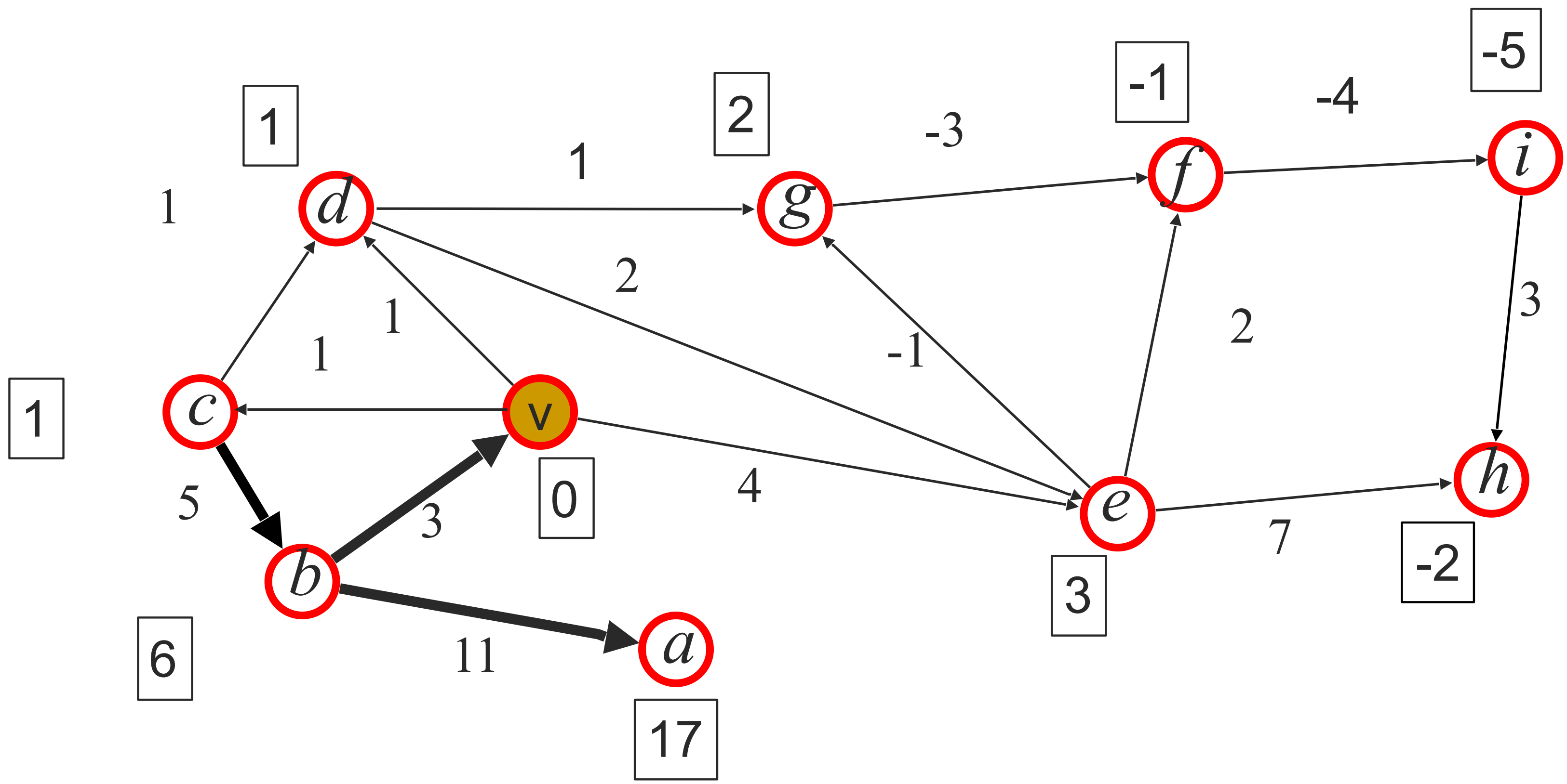
i=3



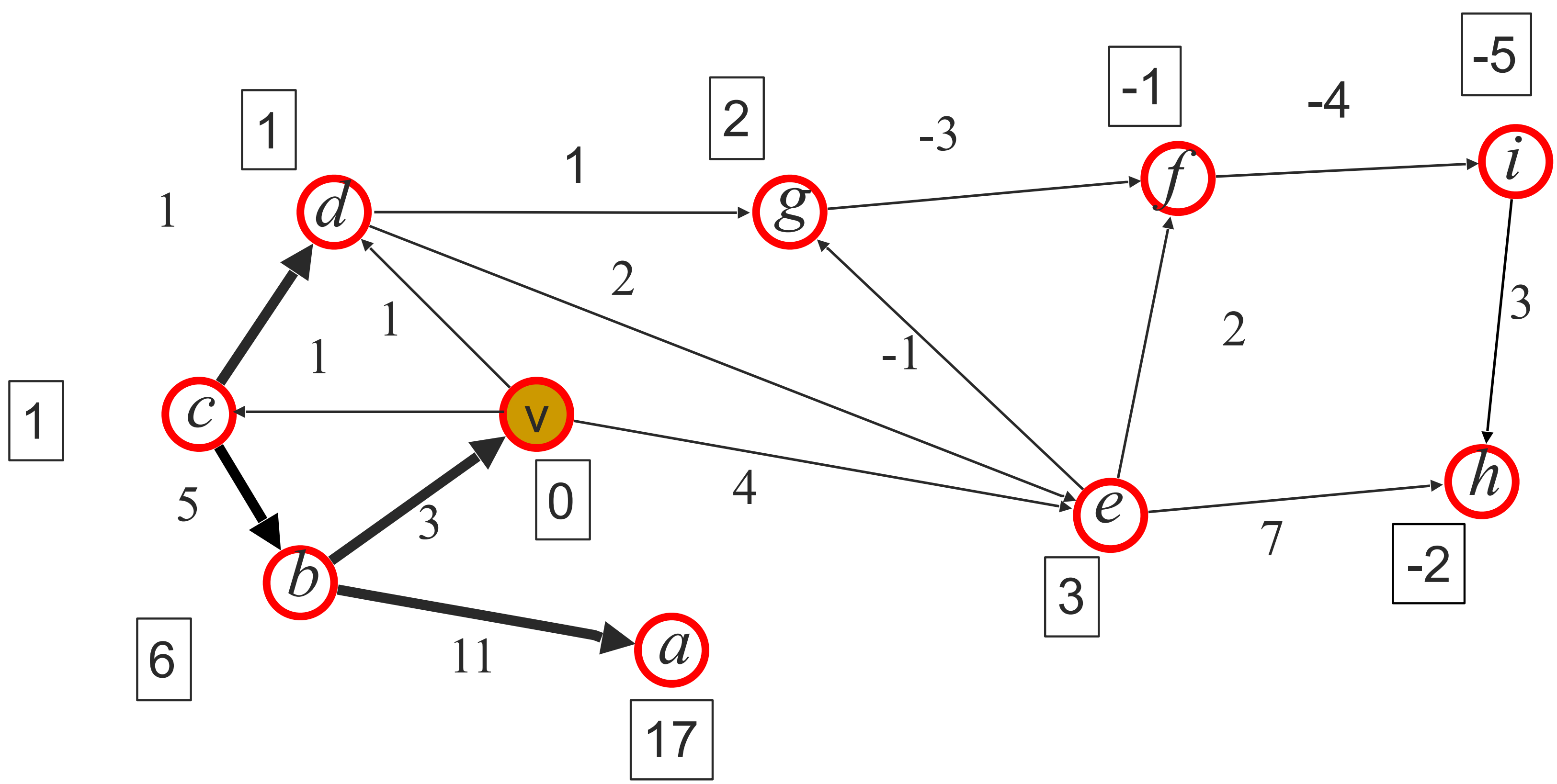
i=3



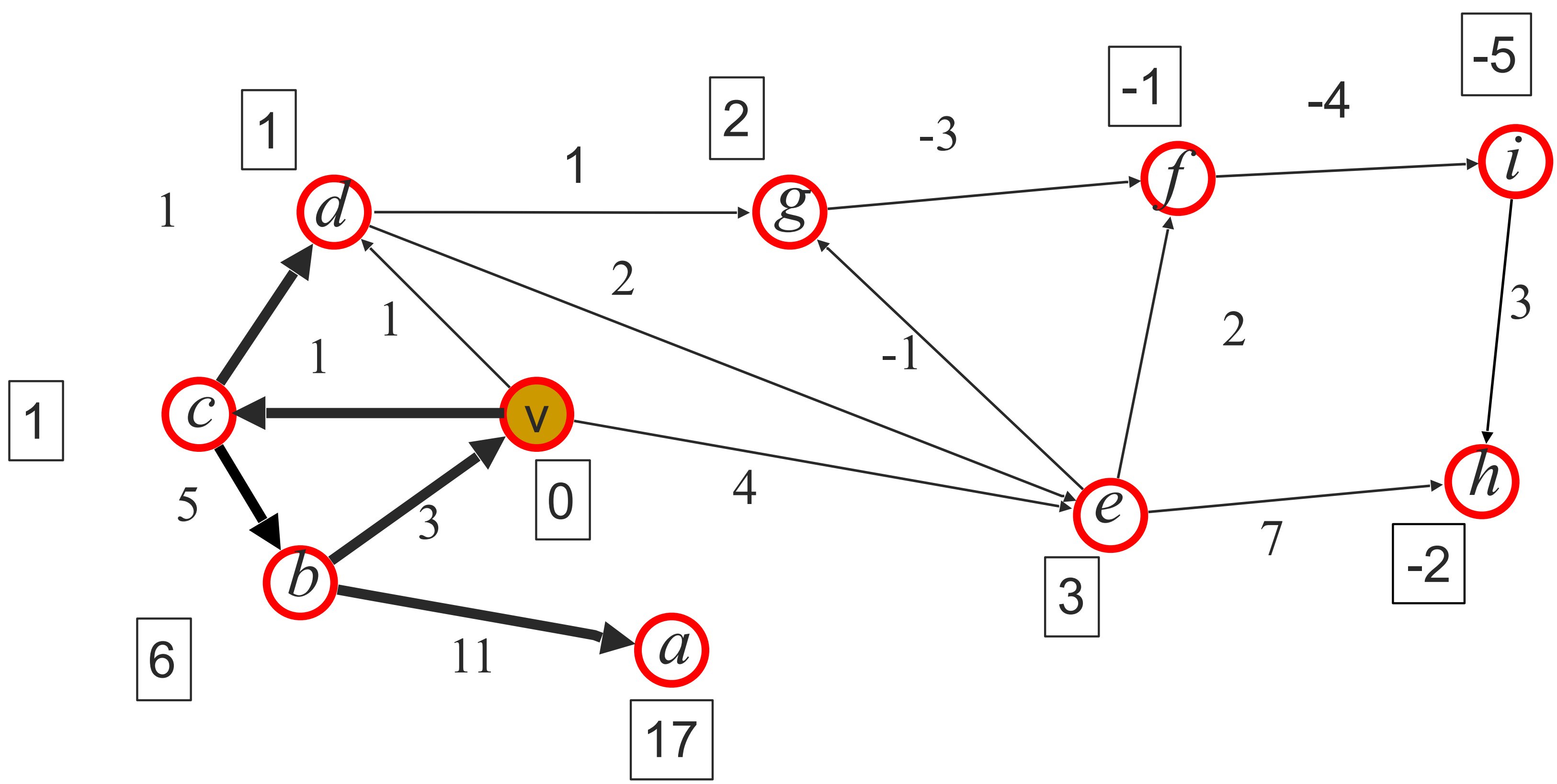
i=3



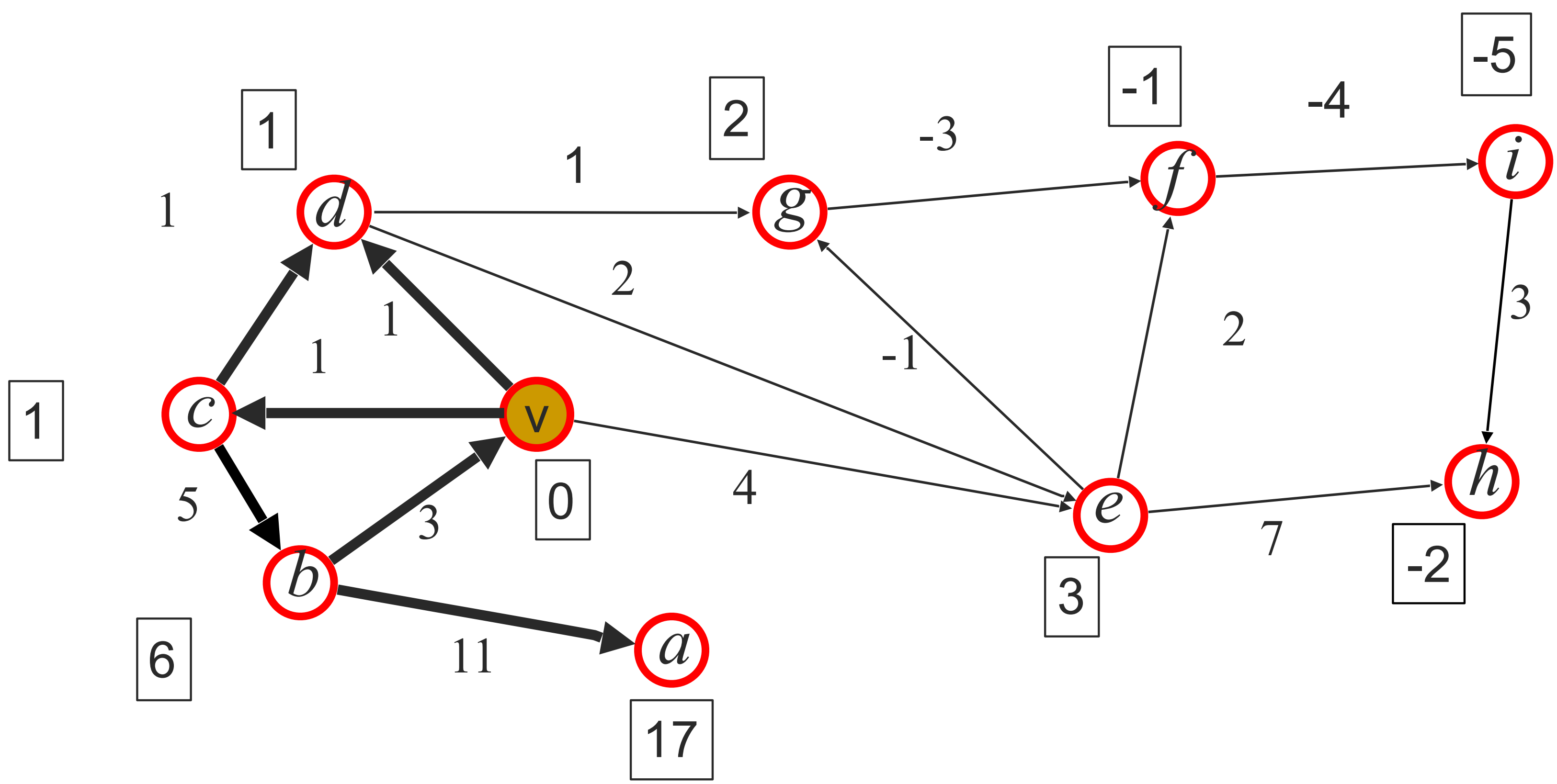
i=3



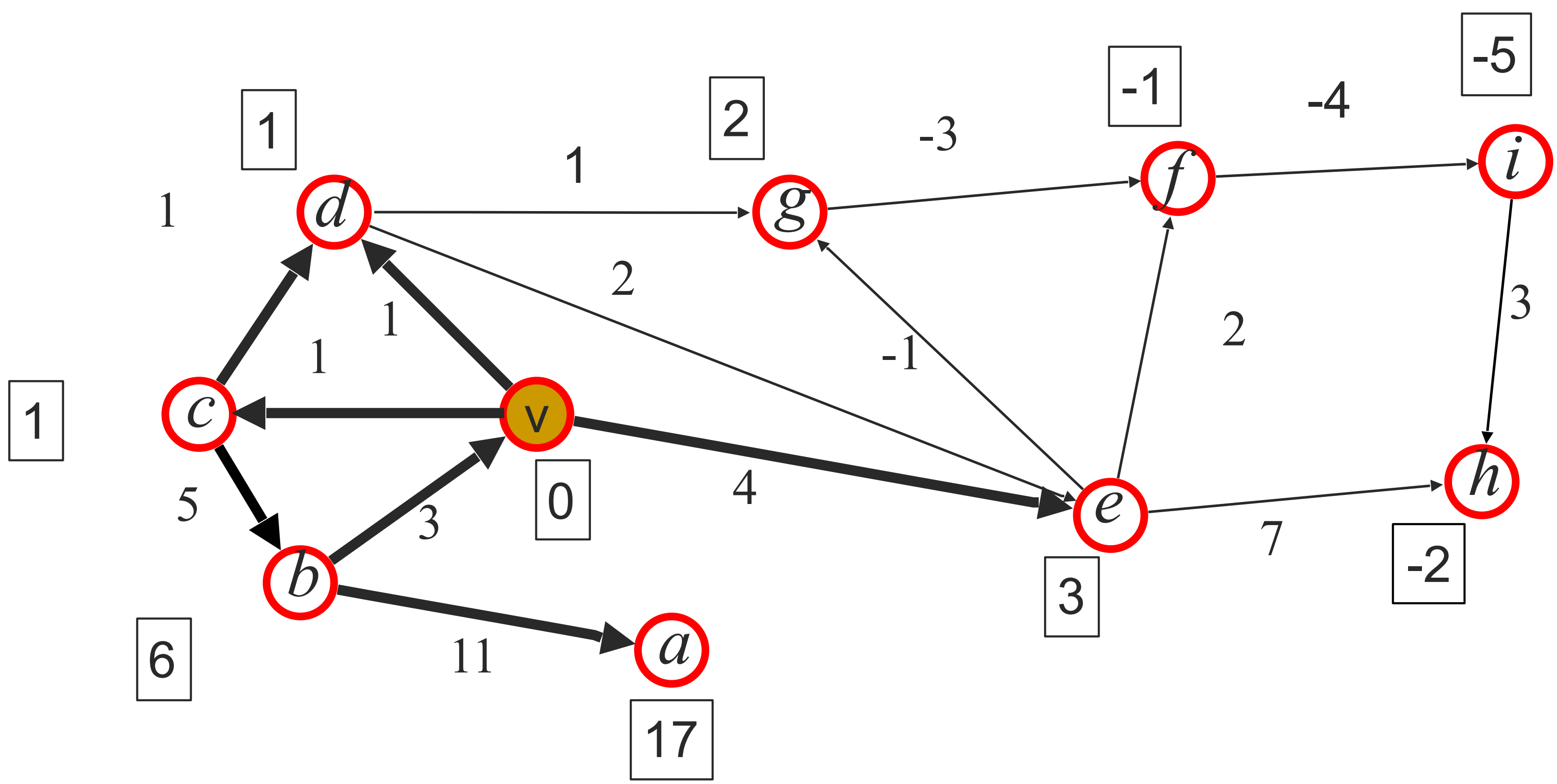
i=3



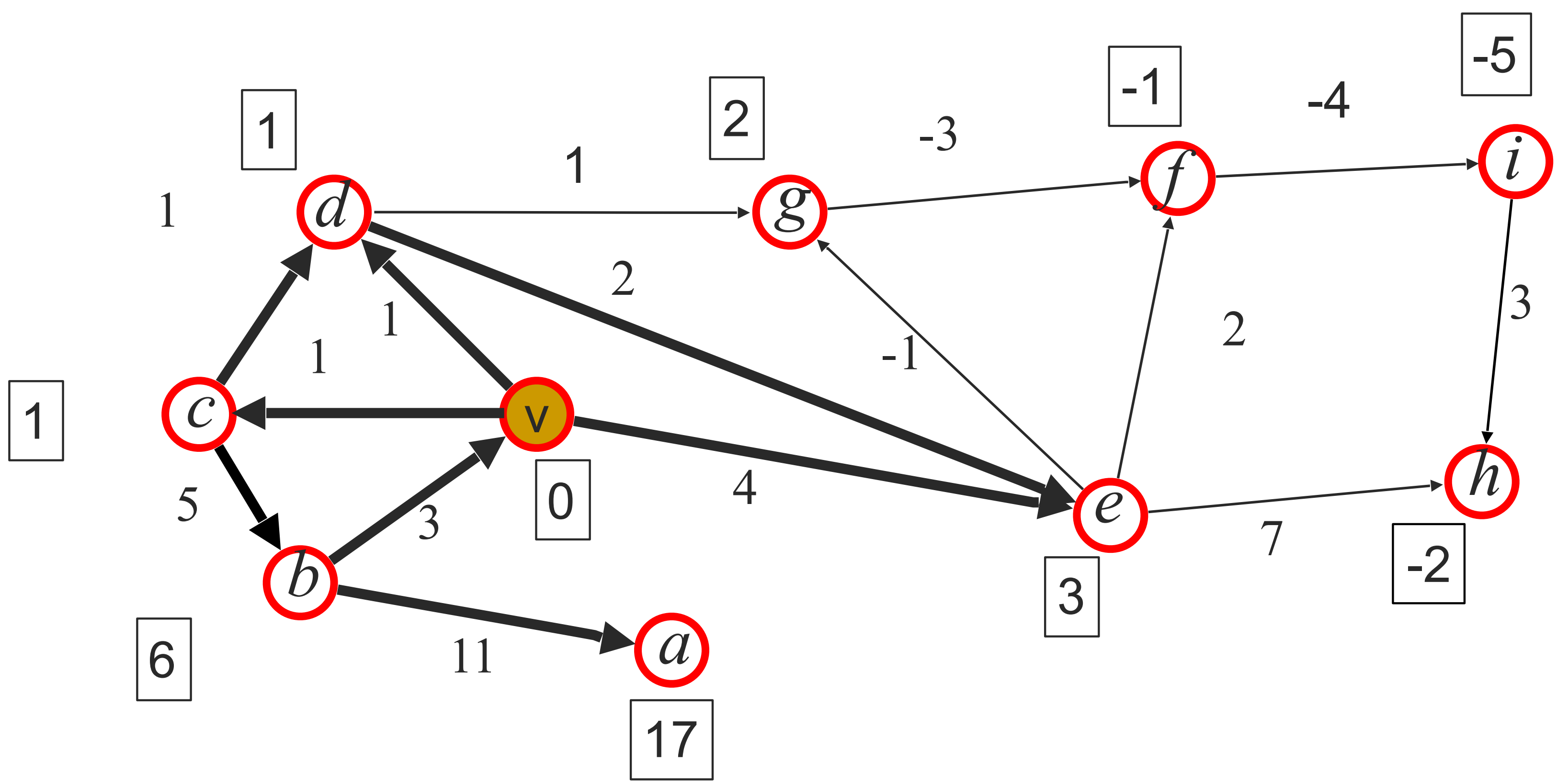
i=3



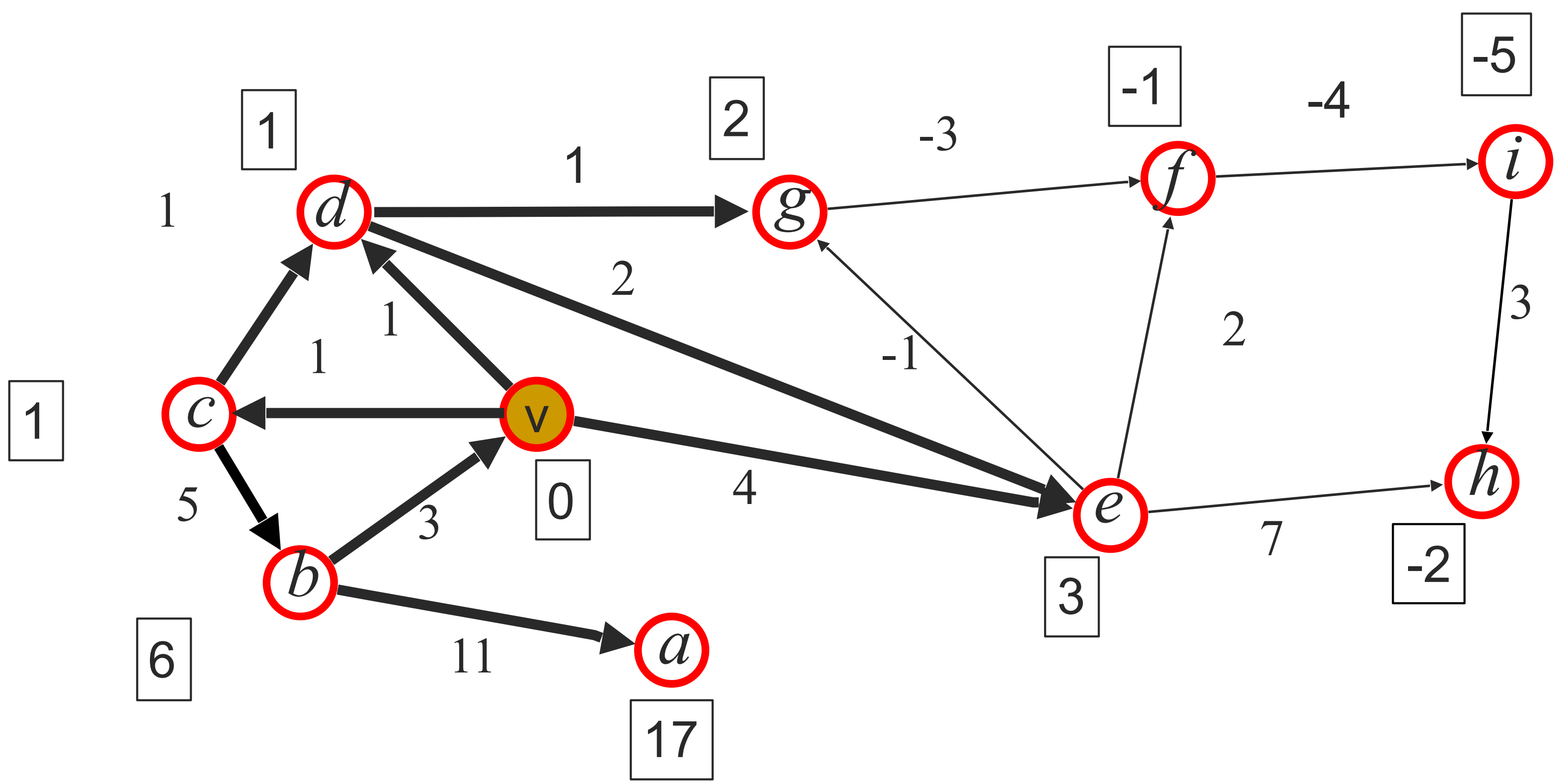
i=3



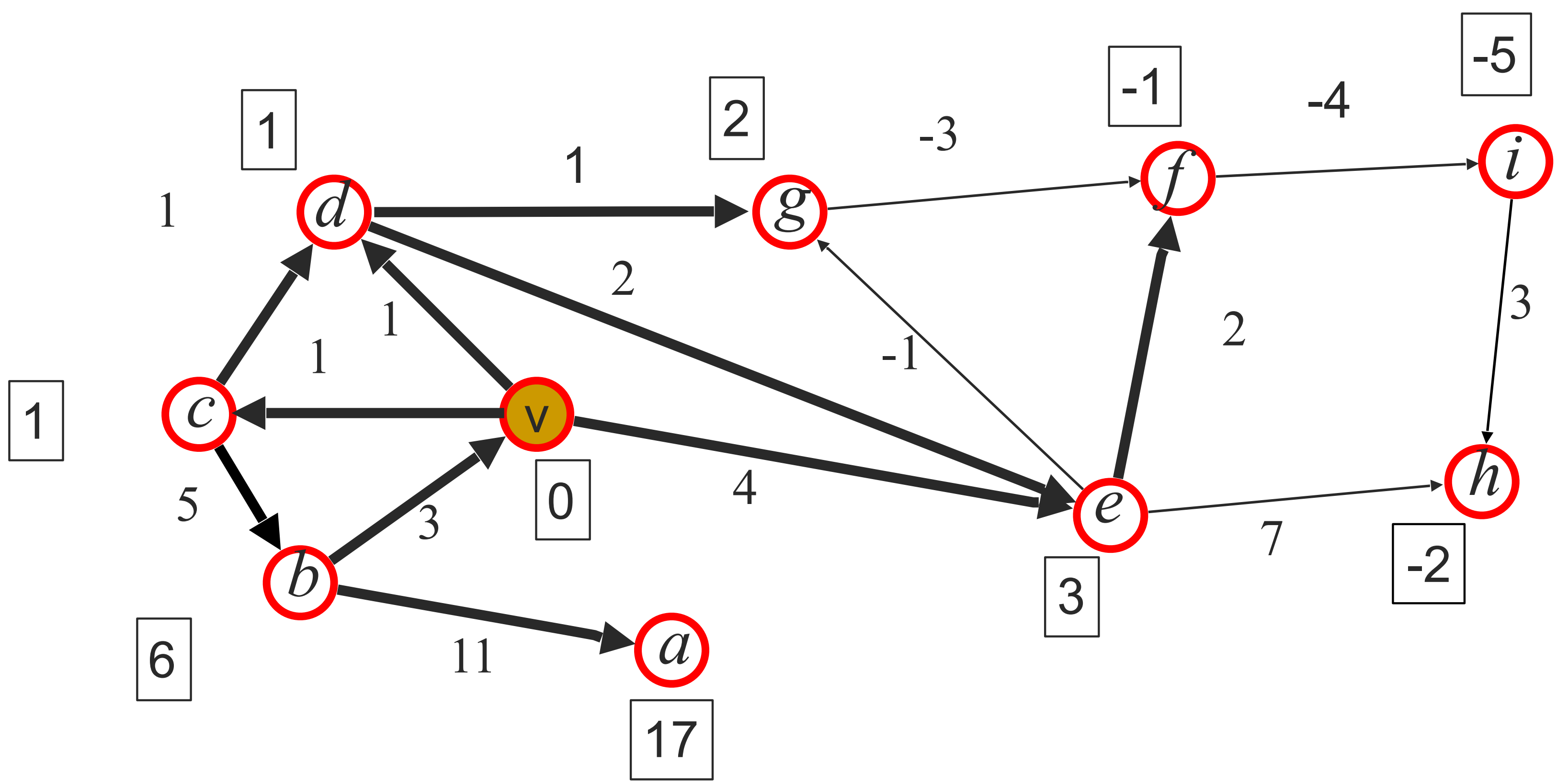
i=3



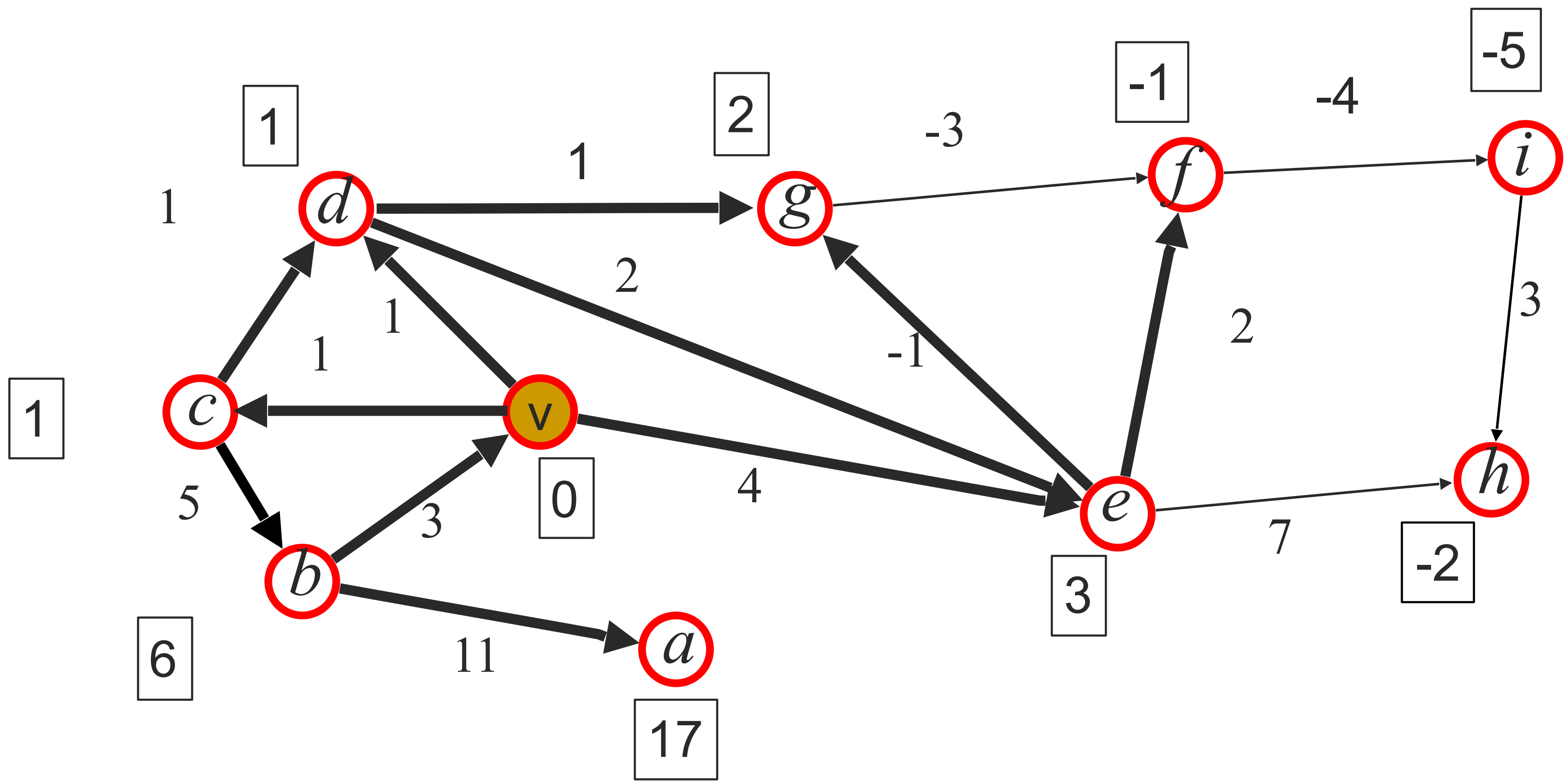
i=3



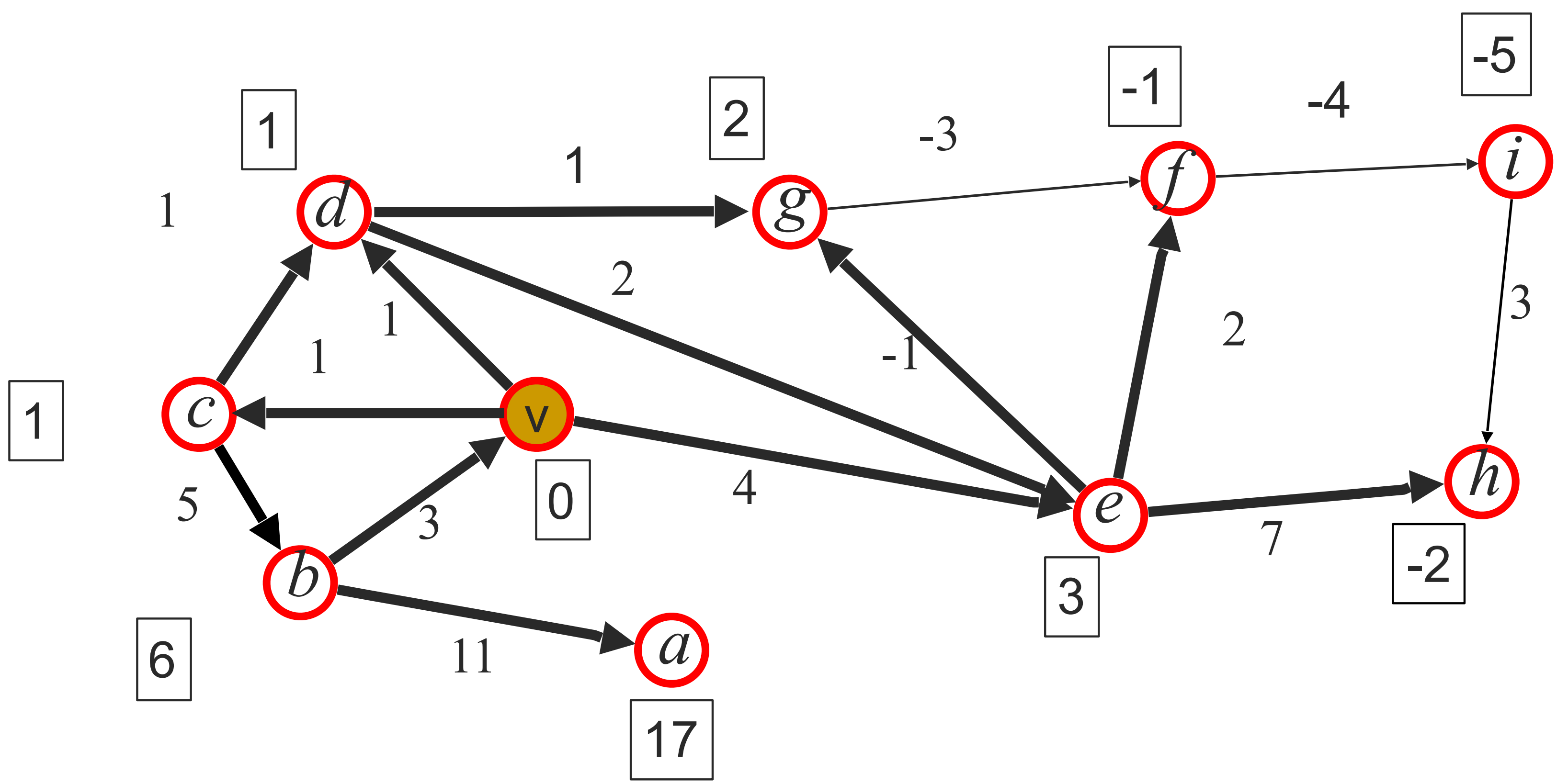
i=3



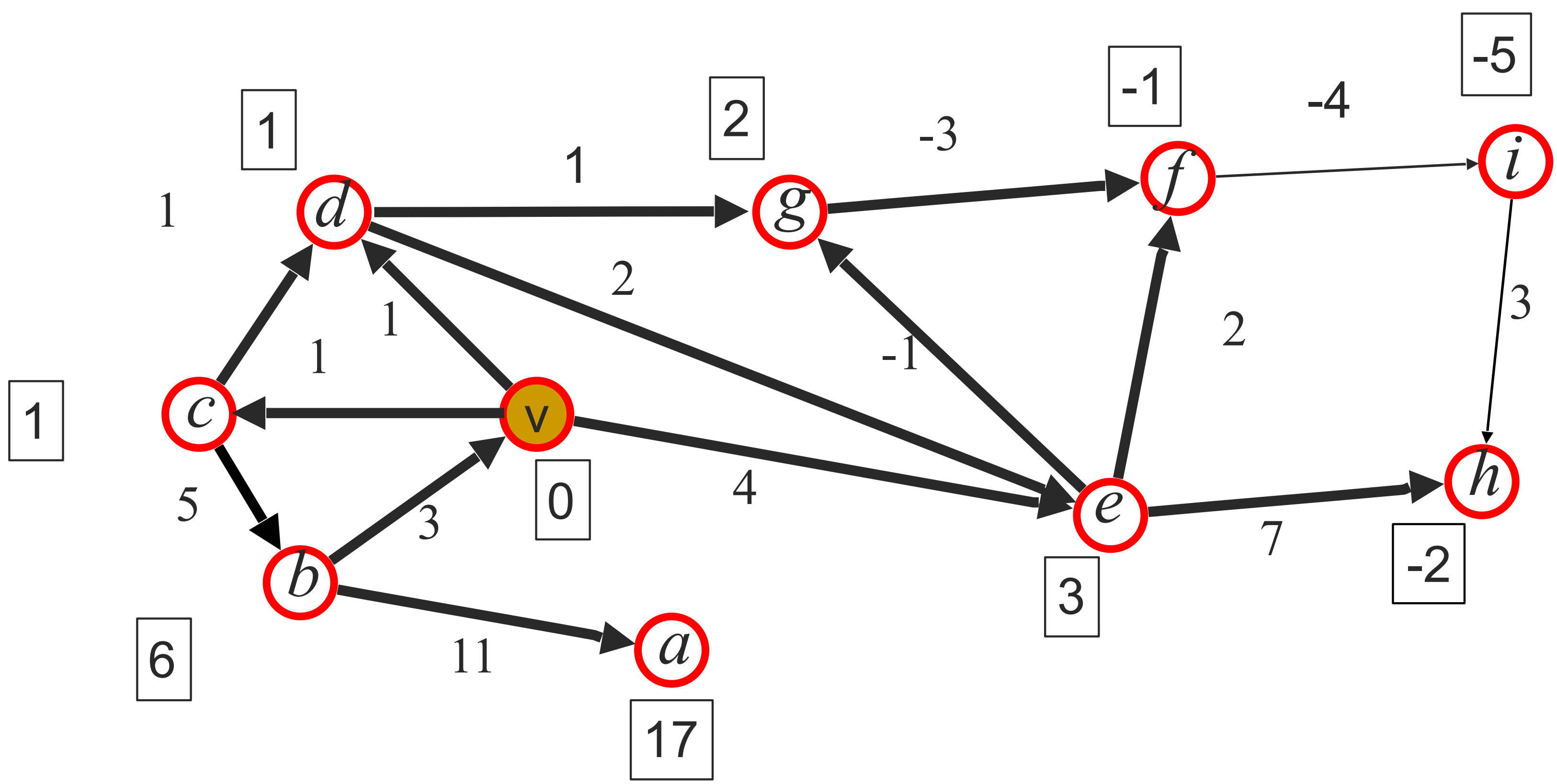
i=3



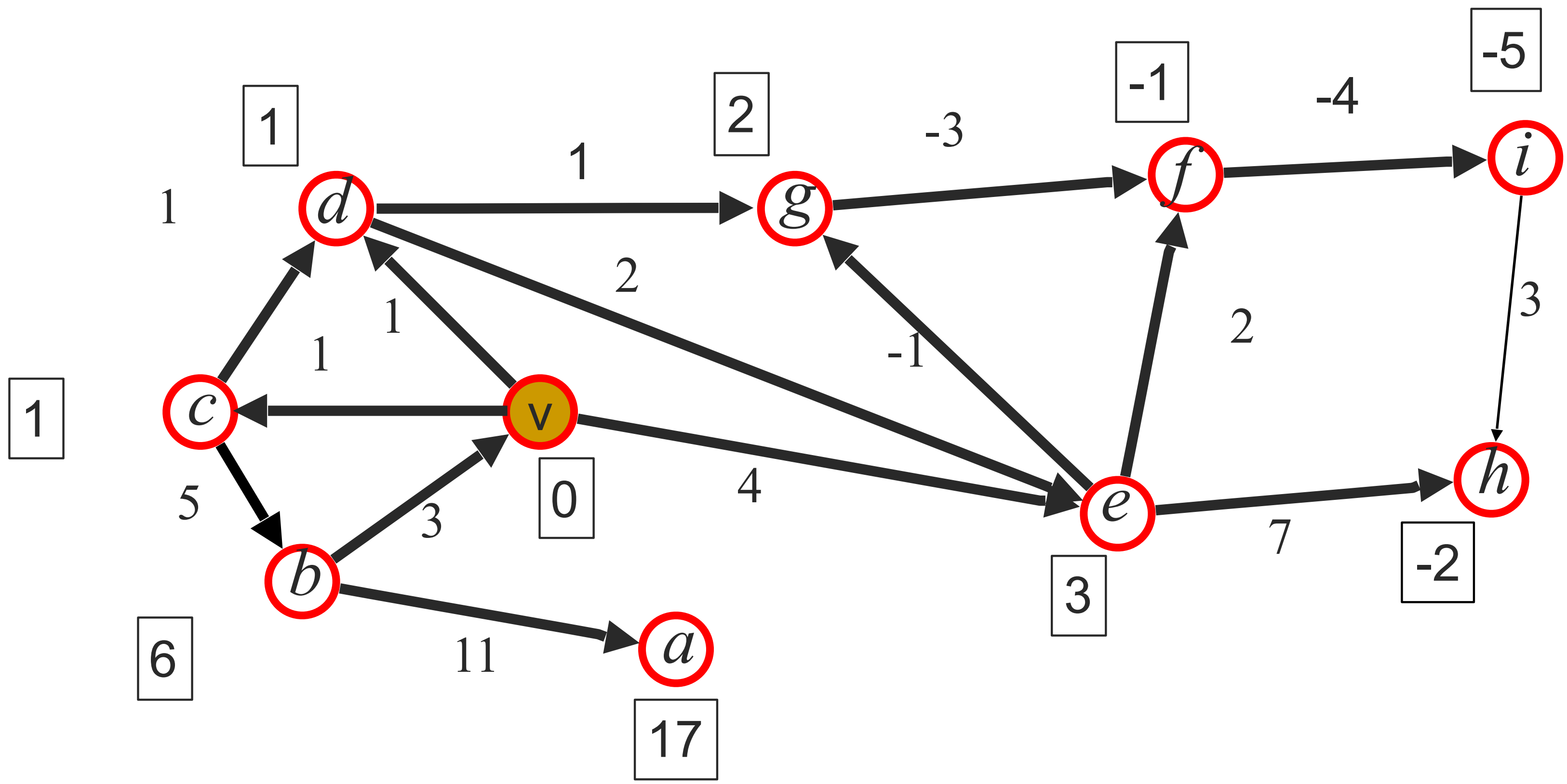
i=3



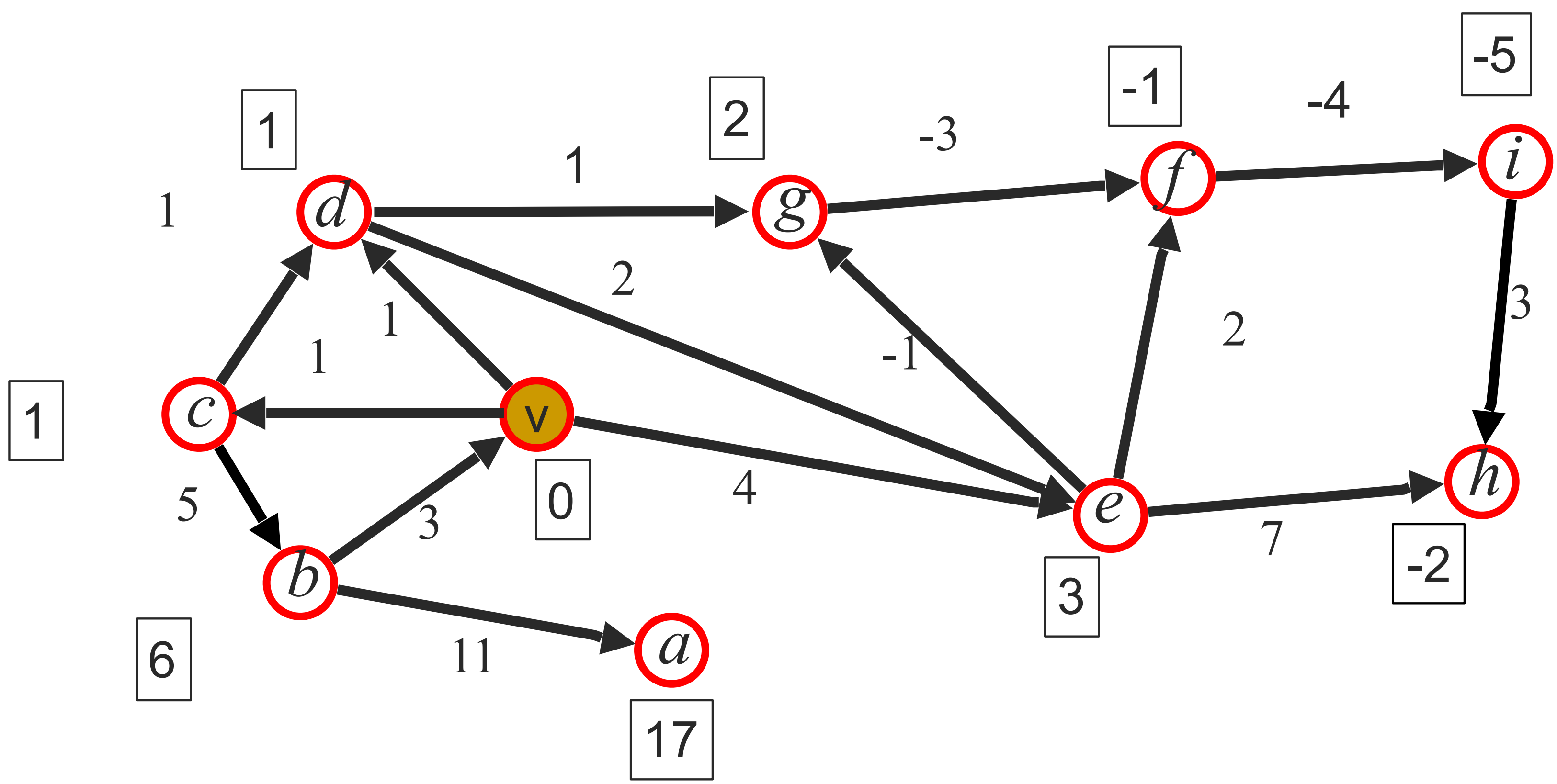
i=3



i=3



i=3



Algorithm continues until i=n-1

- In this example: $i = 9$, no more changes starting at $i = 4$

Running Time

Algorithm Bellman-Ford(G,v)

performs n-1 times a relaxation of every edge in the graph

```
D[v]←0
for each vertex u ≠ v of G do
   D[u]←+∞
for i ← 1 to n-1 do
   for each edge (u,z) in G do
      if D[u]+w((u,z)) < D[z] then
            D[z]←D[u]+w((u,z))
if there are no edges left with potential
   relaxation operations then
   return D
else
   return "G contains a negative cycle"
```

Running time of Bellman-Ford algorithm

- O(nm)

Shortest Paths in directed acyclic graphs

- Can we do faster than Bellman-Ford?
- Great final exam question ☺

All-pairs shortest paths

- For graphs with nonnegative edges
  - Run Dijkstra for each vertex (as a source).
  - n times $O(m \log n)$ is: $O(n\, m \log n)$

- For digraphs with negative edges
  - Run Bellman-Ford for each vertex (as a source).
  - n times $O(n\, m)$ is: $O(n^2\, m)$

- Use programming Dynamic Programming

Dynamic Programming: the three steps

1. Characterize the structure of an optimal solution (**optimal substructure**)
2. Define the value of an optimal solution recursively in terms of the optimal solutions to subproblems (**overlapping subproblems**)
3. Construct an optimal solution with help of a look-up table

Before computing All-Shortest Paths

- Dynamic programming approach to solving the Longest Common Subsequence Problem

Longest Common Subsequence

Similarity of two strings (e.g. words, texts, biological sequences, …)

Longest Common Subsequence
Input: Two strings s and t
Output: Find the longest common subsequence that appears in both s and t.