Weighted Graphs

DSA, Fall 2022

Recall

A directed graph is a pair (V, E) consisting of:

- A finite set V of vertices
- A finite set E of edges of the form (u, v) connecting vertex u to vertex v

Edge (u, v) — written $u \to v$ — has a source u and target v

Undirected graphs: take edges to be pairs {u, v}

A path p from u to v — written $u \overset{p}{\rightsquigarrow} v$ — is a sequence $\langle v_0, \ldots, v_k \rangle$ such that
$v_0 = u$, $v_k = v$, and $(\forall i)\ v_{i-1} \to v_i$

Weighted Graphs

A directed weighted graph is a pair (G, w) where:

- G = (V, E) is a directed graph
- w : E → R is a weight function associating a weight with each edge

For undirected weighted graphs, take edges to be pairs {u, v}

Weight of path $p = \langle v_0, \ldots, v_k \rangle$ $\quad w(p) = \sum_{i \in \{1,\ldots,n\}} w(v_{i-1}, v_i)$

Directed unweighted graphs: directed weighted graphs with constant weight 1

Weighted Graph Representations

Generalize graph representations

Adjacency list

- attach to each vertex a list of connected vertices
- each connected vertex has an associated weight

Adjacency matrix

- matrix indexed by vertices
- entry at (i, j) is weight of edge from i to j (special value for *no edge*?)

Adjacency List Representation

```
type WGraph struct {
    vertices int
    edges []*Edge
}

type Edge struct {
    target int
    weight float32
    next *Edge
}
```

Adjacency Matrix Representation

```
type WGraph struct {
    vertices int
    edges [][]float32
}
```

Algorithms for Weighted Graphs

We look at two core problems

- Minimum Spanning Tree
- Single-Source Shortest Paths

Other interesting problems:

- All-Pairs Shortest Paths
- Maximum Flow / Minimum Cut
- Minimum Flow / Maximum Cut

Minimum Spanning Trees

Given a (connected) undirected graph

A spanning tree is a tree that connects all the vertices in the graph using a subset of the edges in the graph

If a graph is weighted, we can ask for a spanning tree with minimum total weight (total weight = sum of the weights of the edges that are part of the tree)

- Kruskal's algorithm
- Prim's algorithm

Greedy Algorithms

An approach to devising algorithms:

- at each step of the algorithm, make the choice that is the best in the moment

Doesn't always lead to globally optimal solutions, but sometimes it is

(Can characterize those problems that do admit greedy algorithm solutions — cf Chapter 17 on matroids)

Priority Queues

Prim's algorithm relies on a Priority Queue ADT:

- each item has a key (e.g., weight) and a value (e.g., vertex)
- operation INSERT(Q, k, v)
- operation EXTRACTMIN(Q)
- operation DECREASEKEY(Q, k, v, d)

Can be implemented using Sets

- EXTRACTMIN := MINIMUM + DELETE
- DECREASEKEY := SEARCH + DELETE + INSERT

Prim's Algorithm

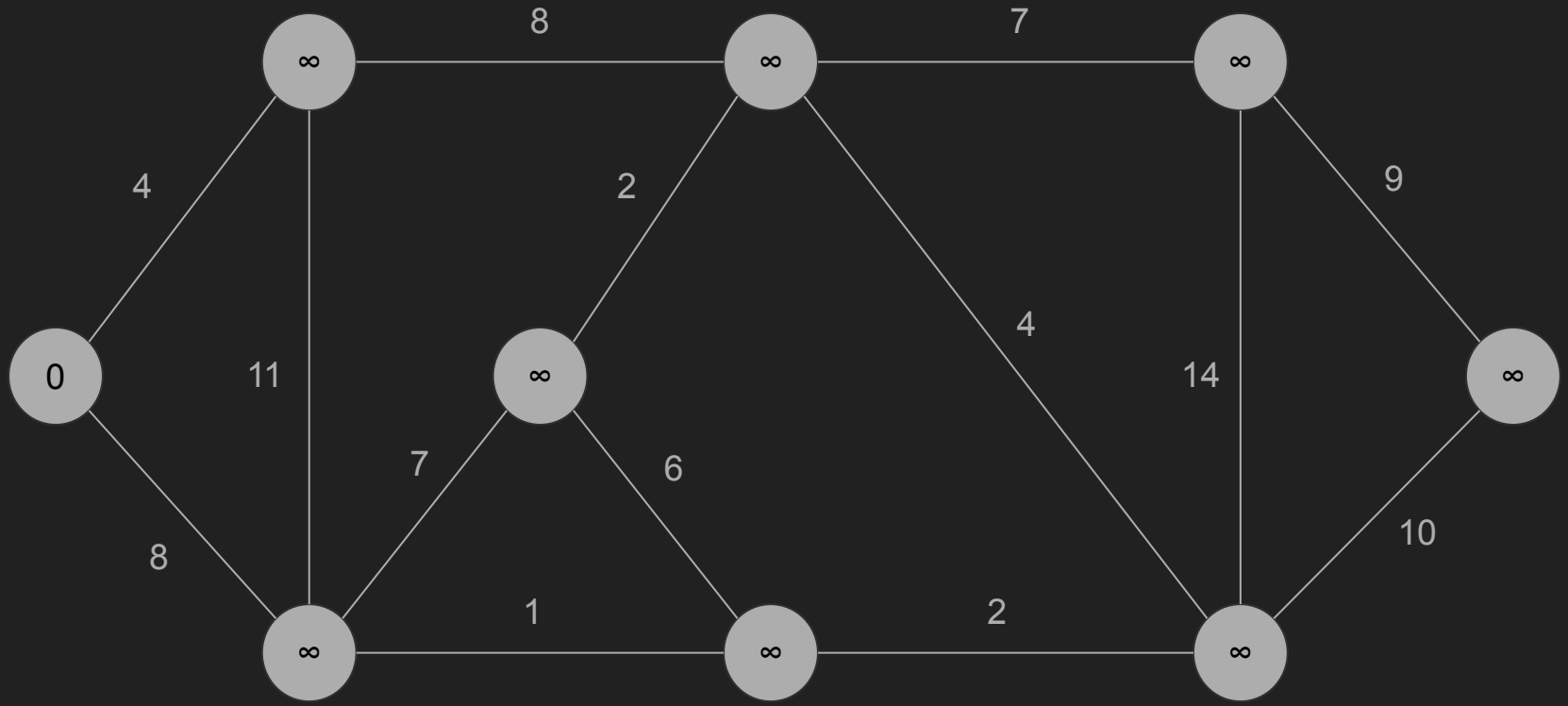
Given undirected weighted graph (G, w) and vertex r ∈ V

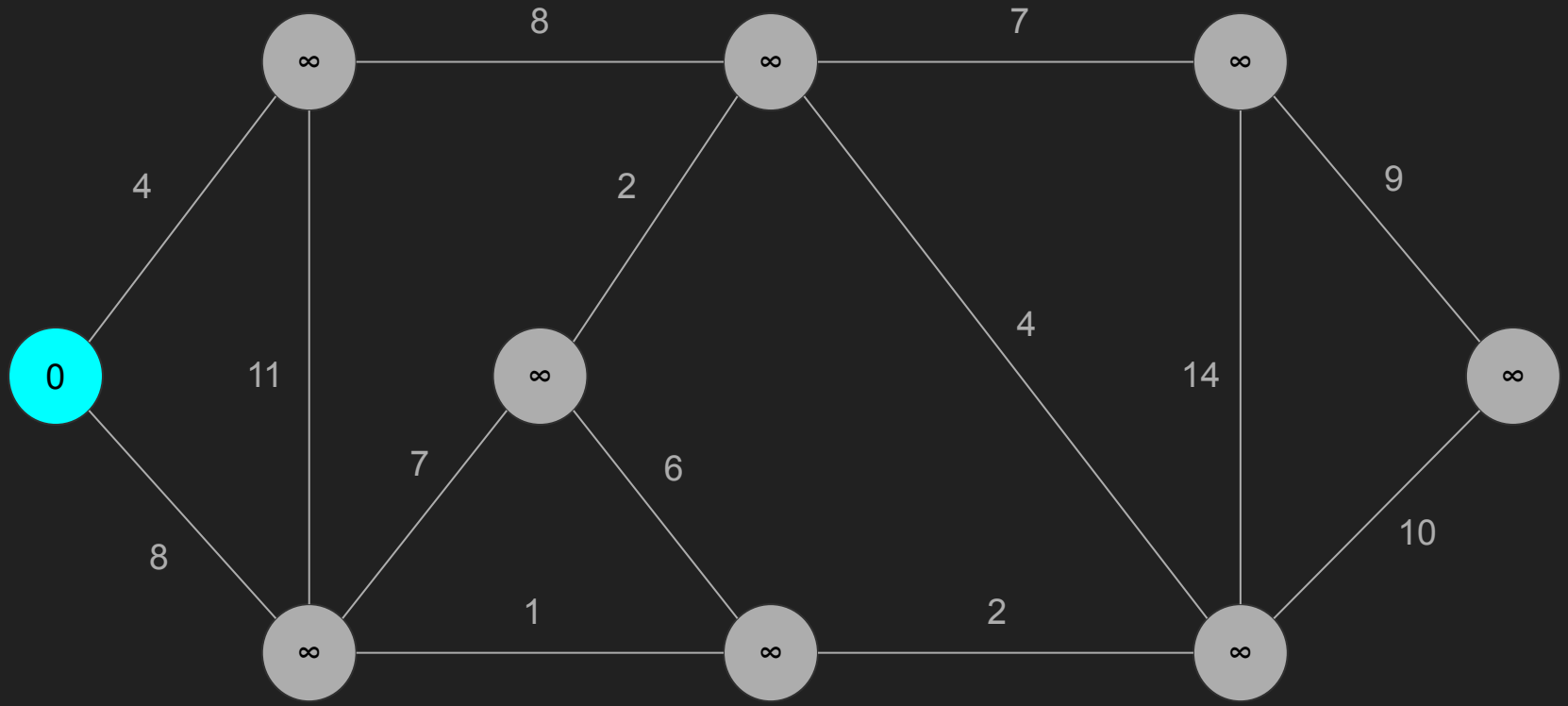
```
for v ∈ V
    key[v] ← ∞
    INSERT(Q, v)
key[r] ← 0
π[r] ← NIL
while Q ≠ ∅
    u ← EXTRACTMIN(Q)
    for v ∈ Adj[u]
        if v ∈ Q and w(u, v) < key[v]
            π[v] ← u
            key[v] ← w(u, v)
```

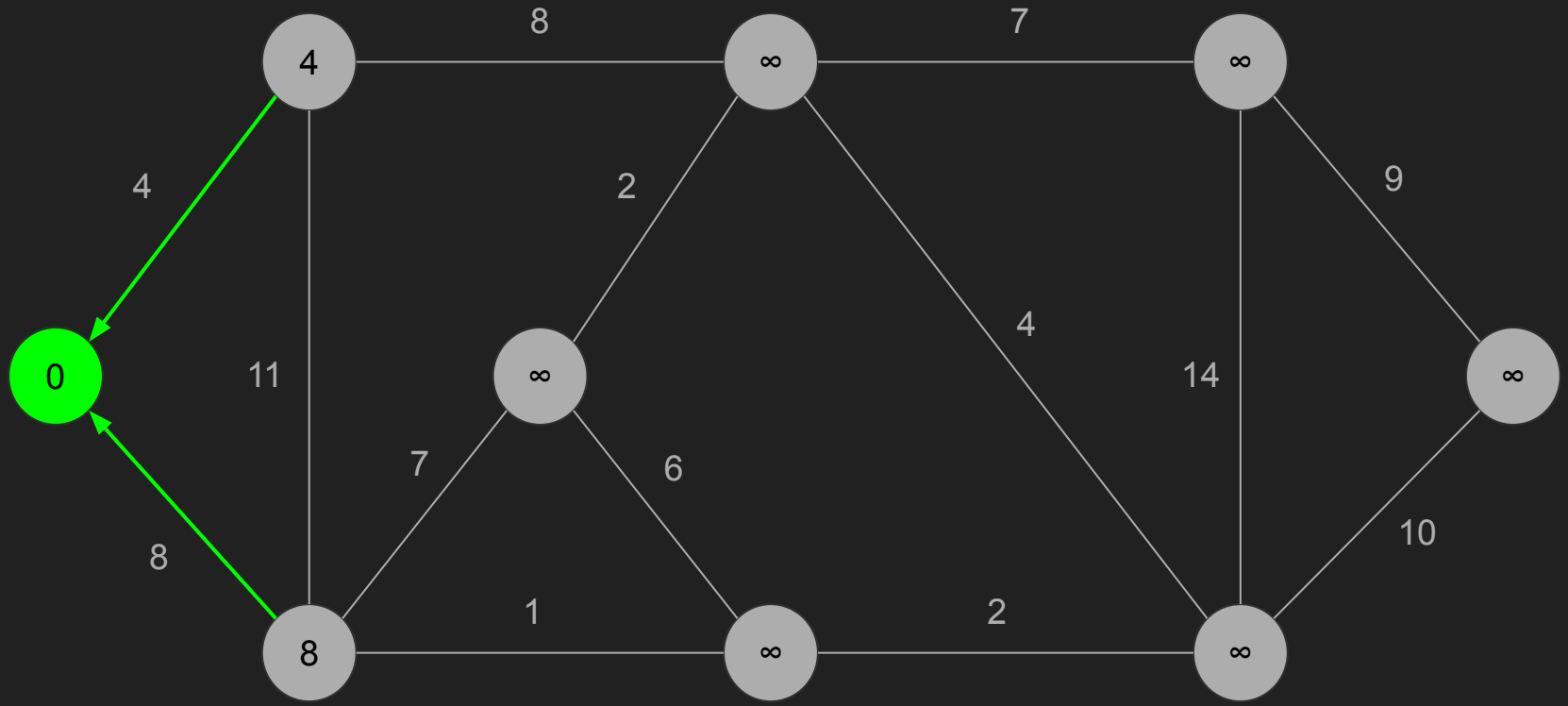
Prim's Algorithm

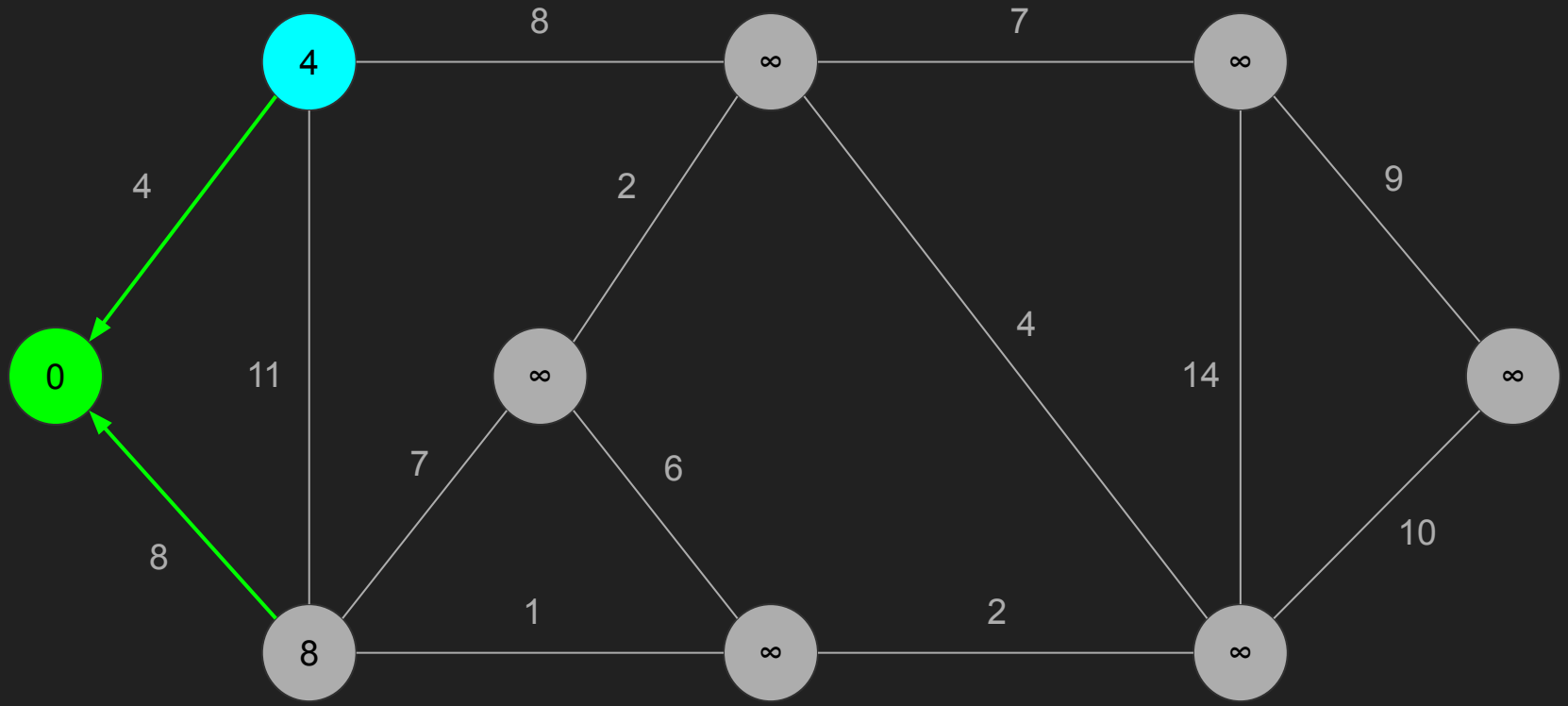
Given undirected weighted graph (G, w) and vertex r ∈ V

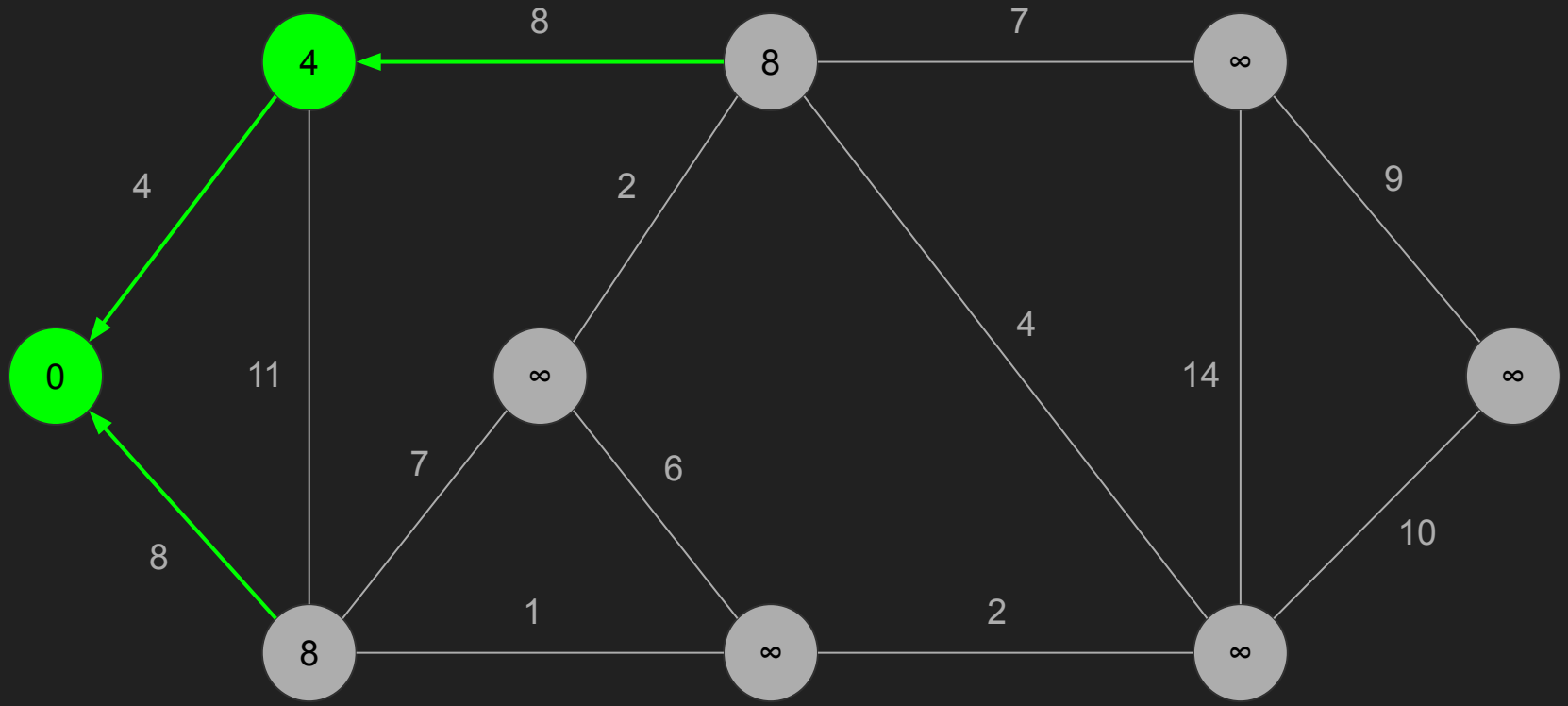
```
for v ∈ V
    key[v] ← ∞
    INSERT(Q, v)
key[r] ← 0  {DECREASEKEY}
π[r] ← NIL
while Q ≠ ∅
    u ← EXTRACTMIN(Q)
    for v ∈ Adj[u]
        if v ∈ Q and w(u, v) < key[v]
            π[v] ← u
            key[v] ← w(u, v) {DECREASEKEY}
```

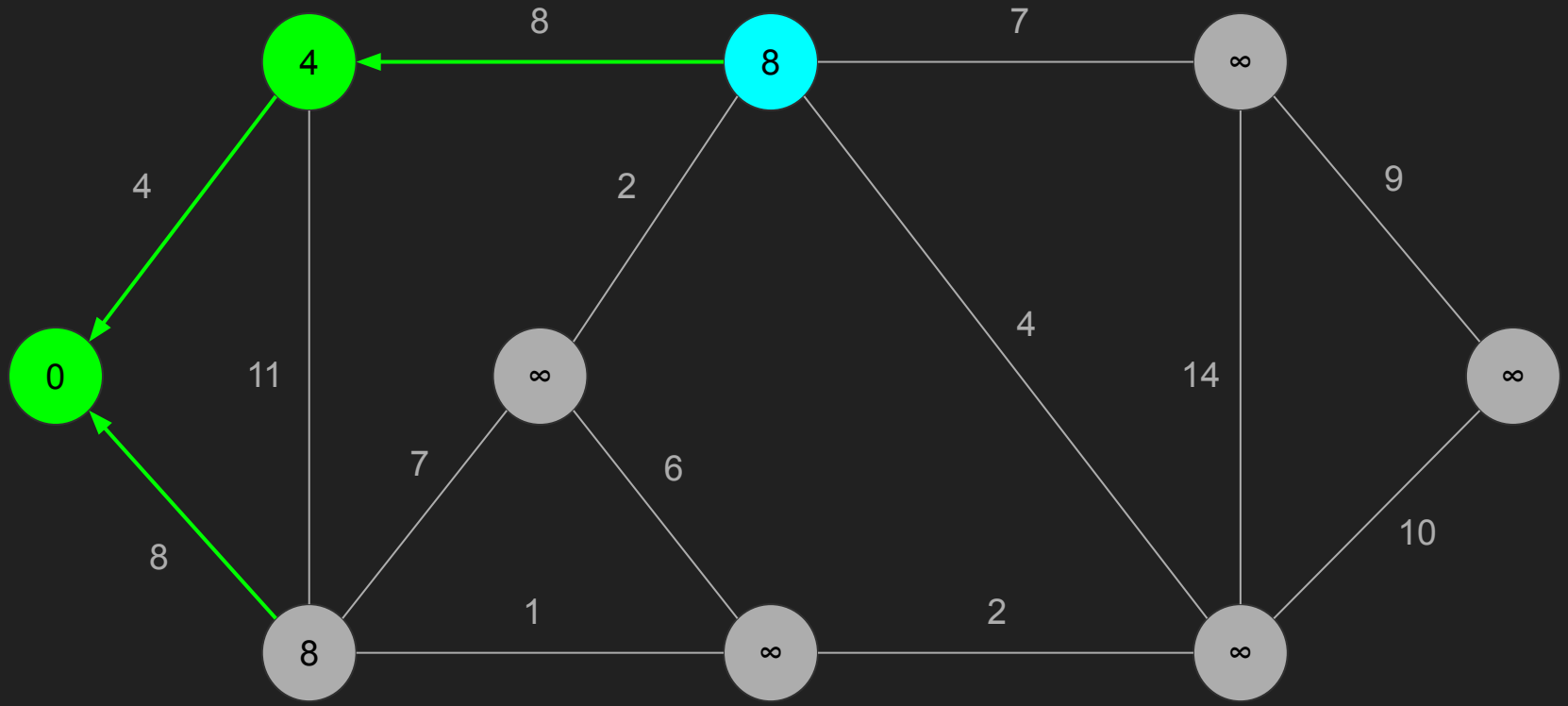

8
7
∞
∞
∞
4
2
9
0
11
∞
4
14
∞
7
6
8
10
1
2
∞
∞
∞

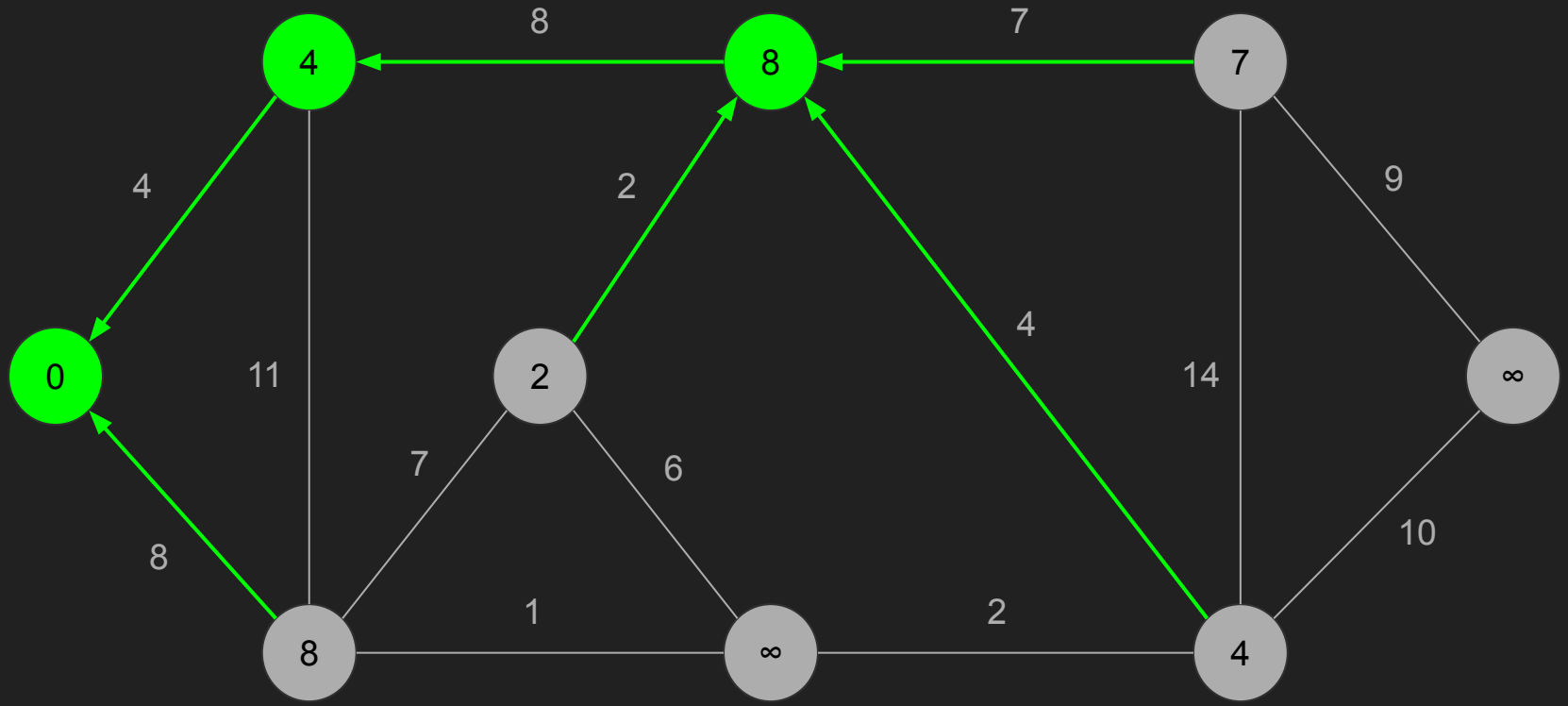

8
7
∞
∞
∞
4
2
9
0
11
∞
4
14
∞
7
6
8
10
1
2
∞
∞
∞

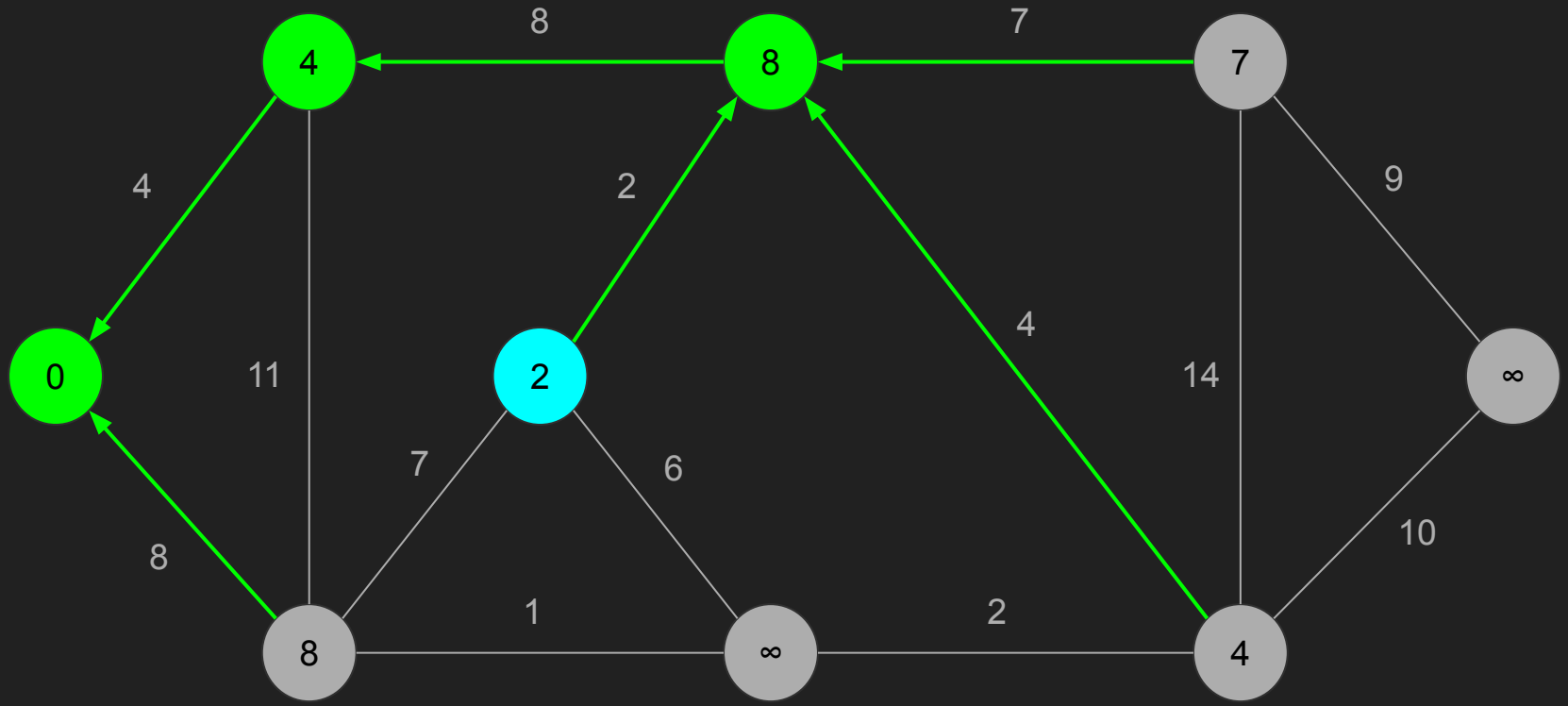

4
8
∞
7
∞
4
2
9
0
11
∞
4
14
∞
7
6
8
10
8
1
2
∞
∞

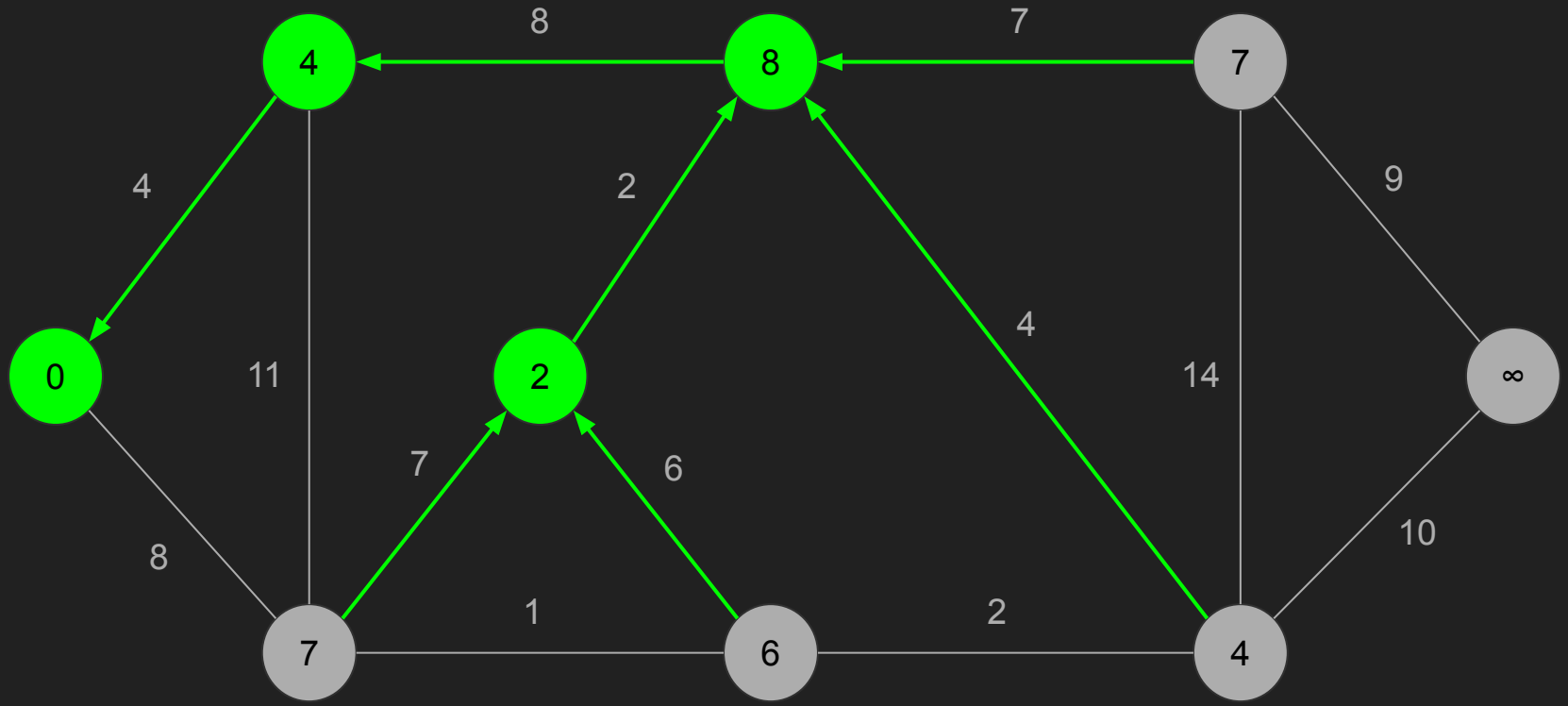

4
8
∞
7
∞
4
2
9
0
11
∞
4
14
∞
7
6
8
10
8
1
2
∞
∞

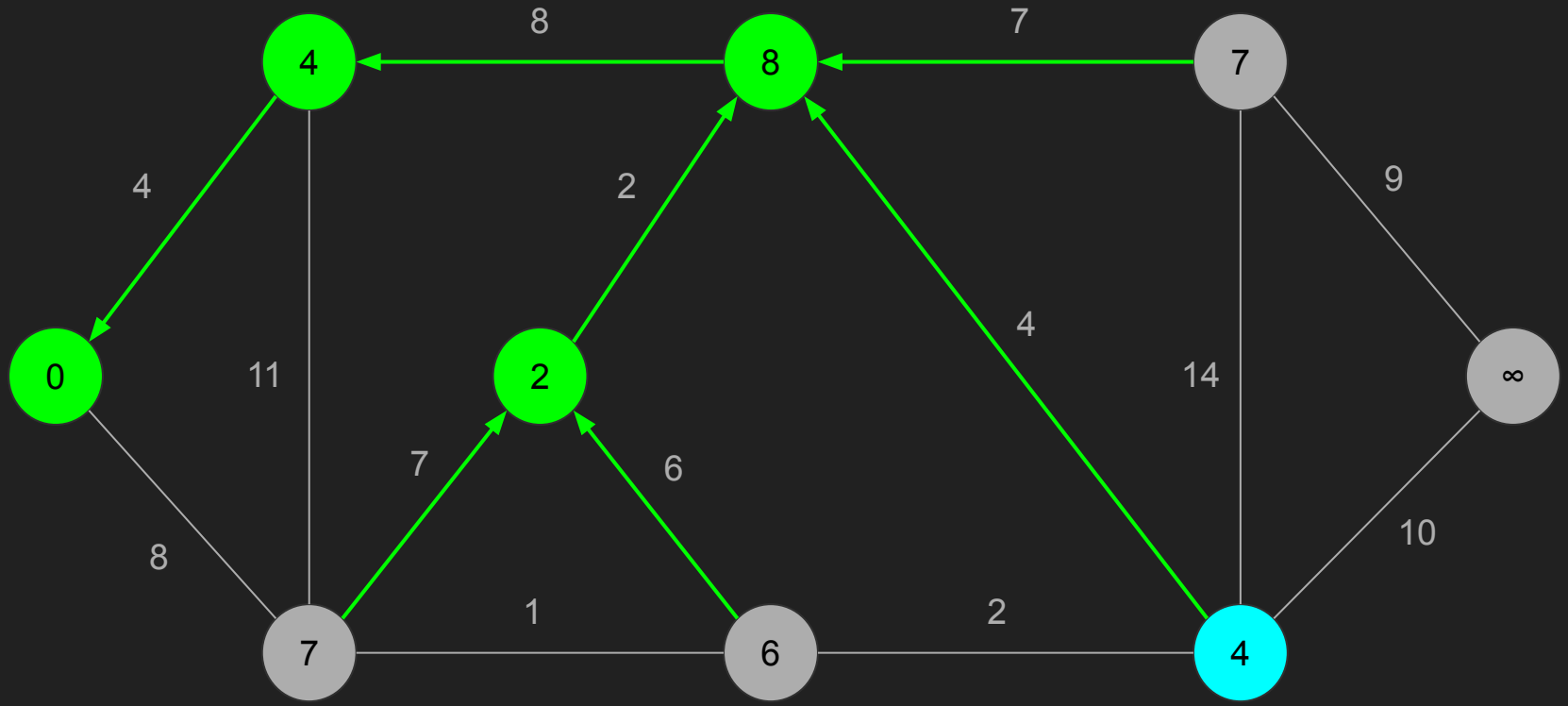

0
4
8
∞
∞
8
∞
4
8
11
2
4
14
9
7
6
10
8
1
2

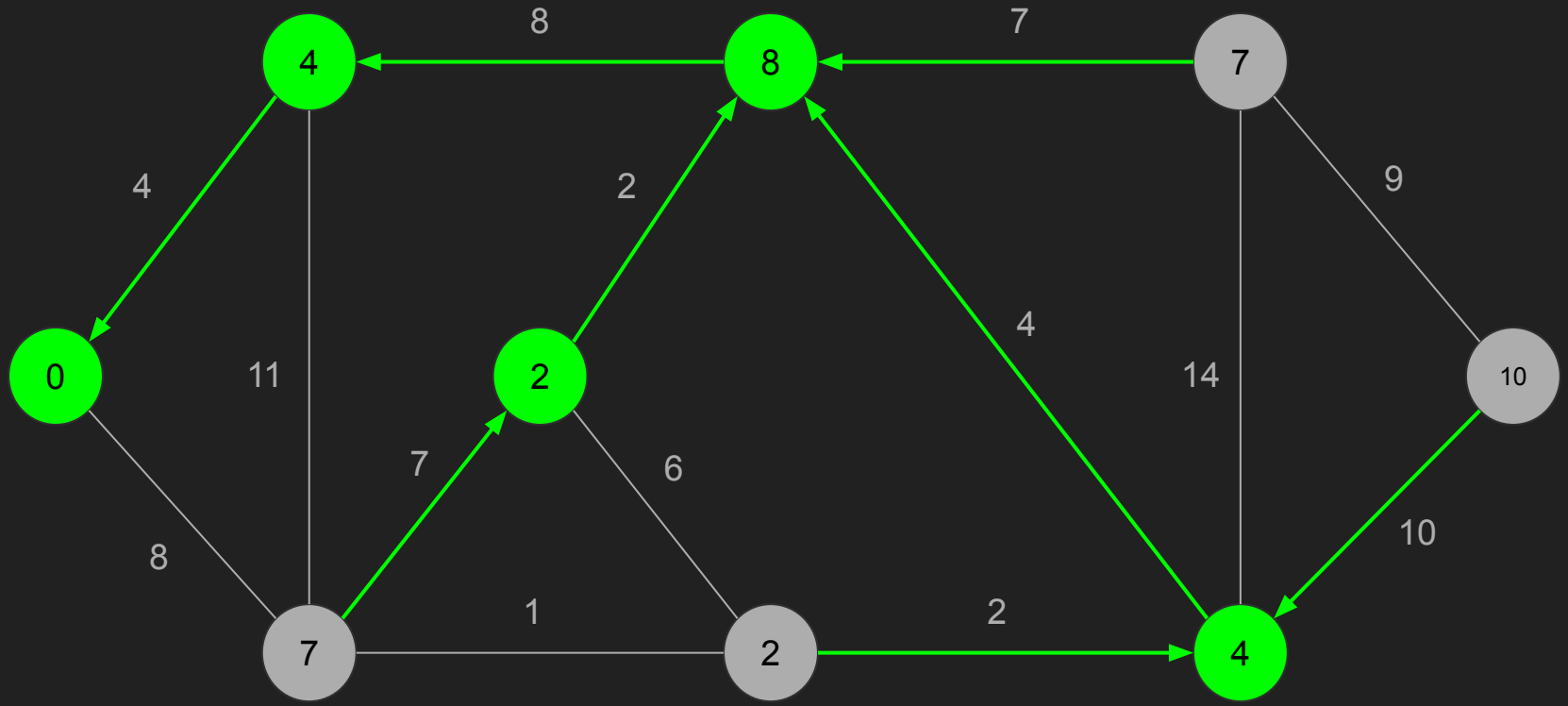

4
8
8
7
∞
4
2
9
0
11
∞
4
14
∞
7
6
8
10
8
1
2
∞
∞

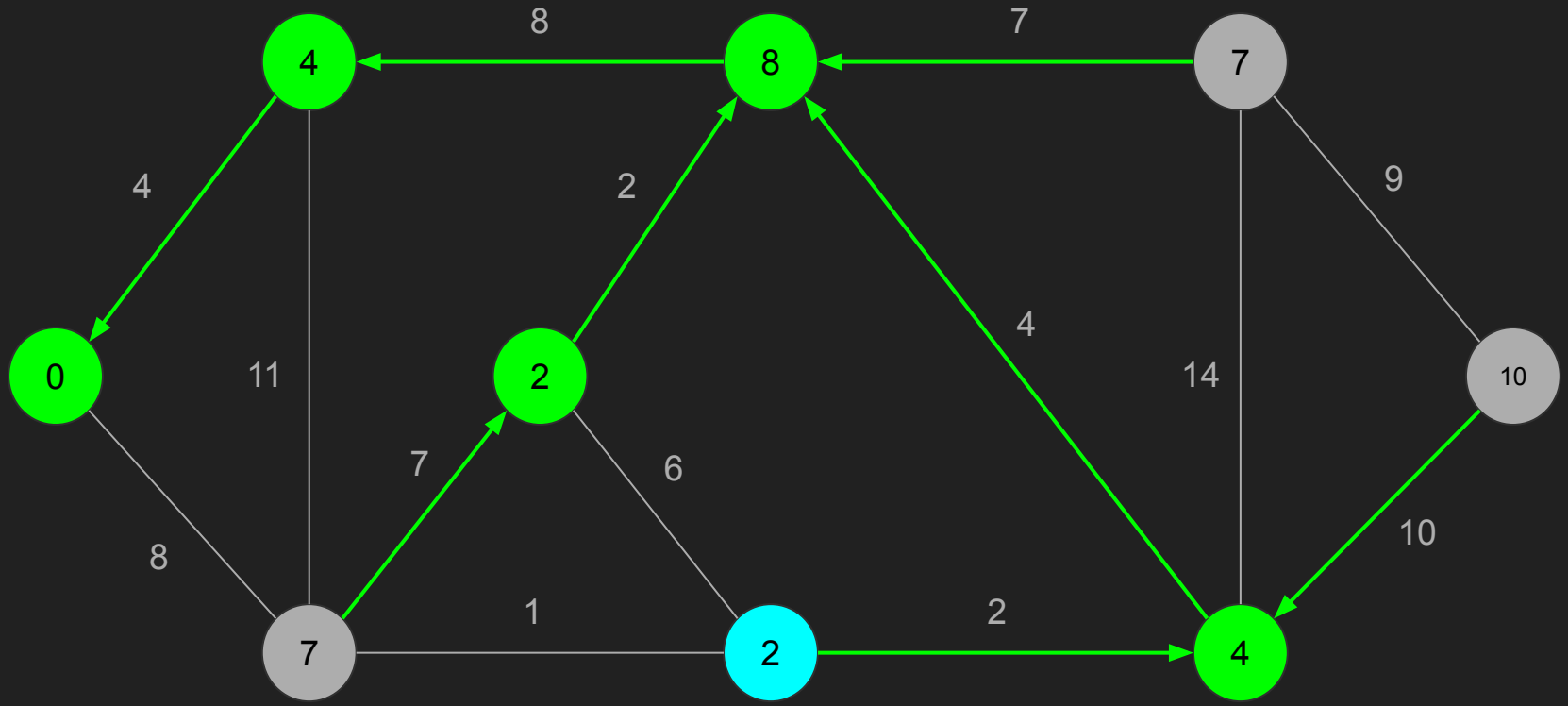

8
7
4
8
7
4
2
9
4
0
11
2
14
∞
7
6
10
8
1
2
8
∞
4

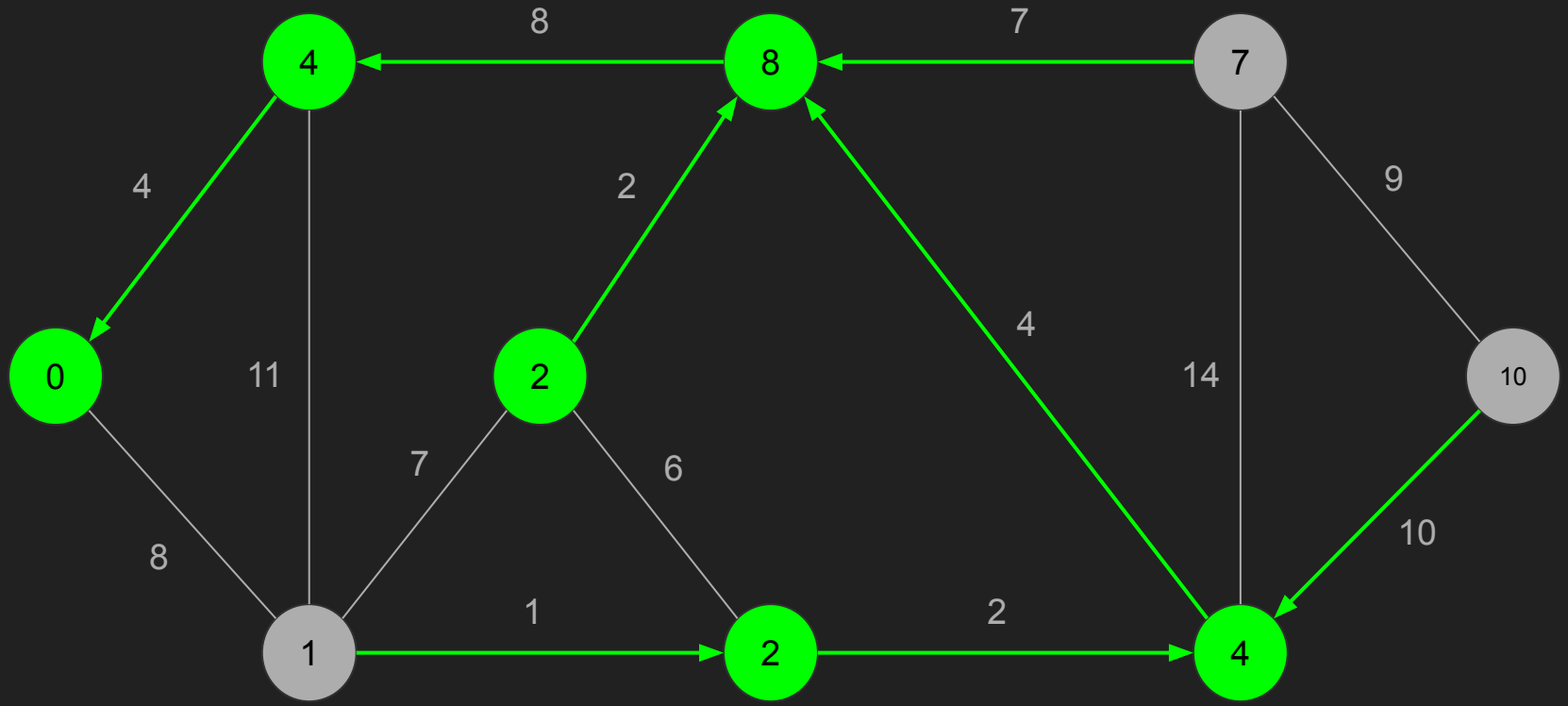

4
8
7
0
2
∞
8
∞
4
8
4
11
7
6
14
9
4
2
10
1
2

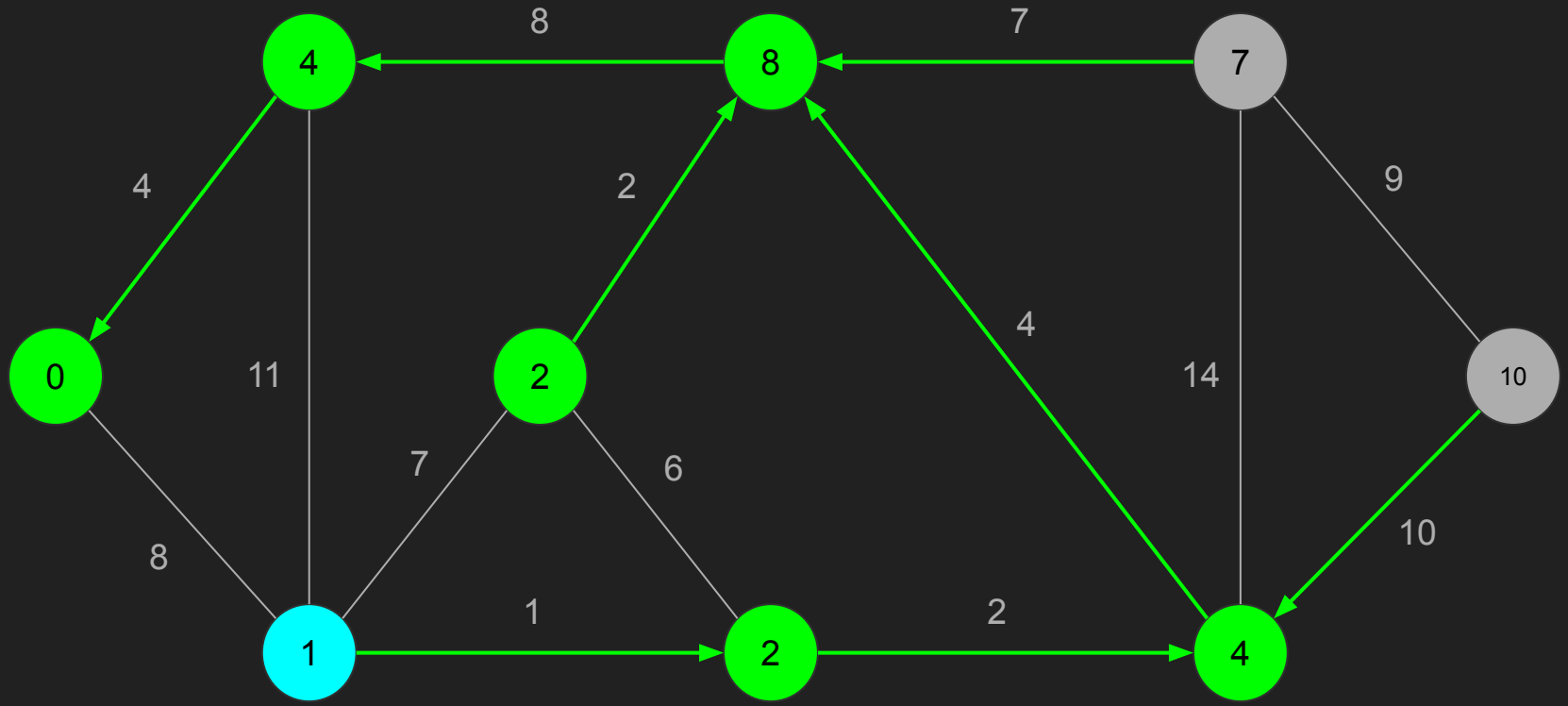

4
8
7
0
2
8
7
4
11
2
4
14
9
∞
7
6
8
10
1
2
7
6
4

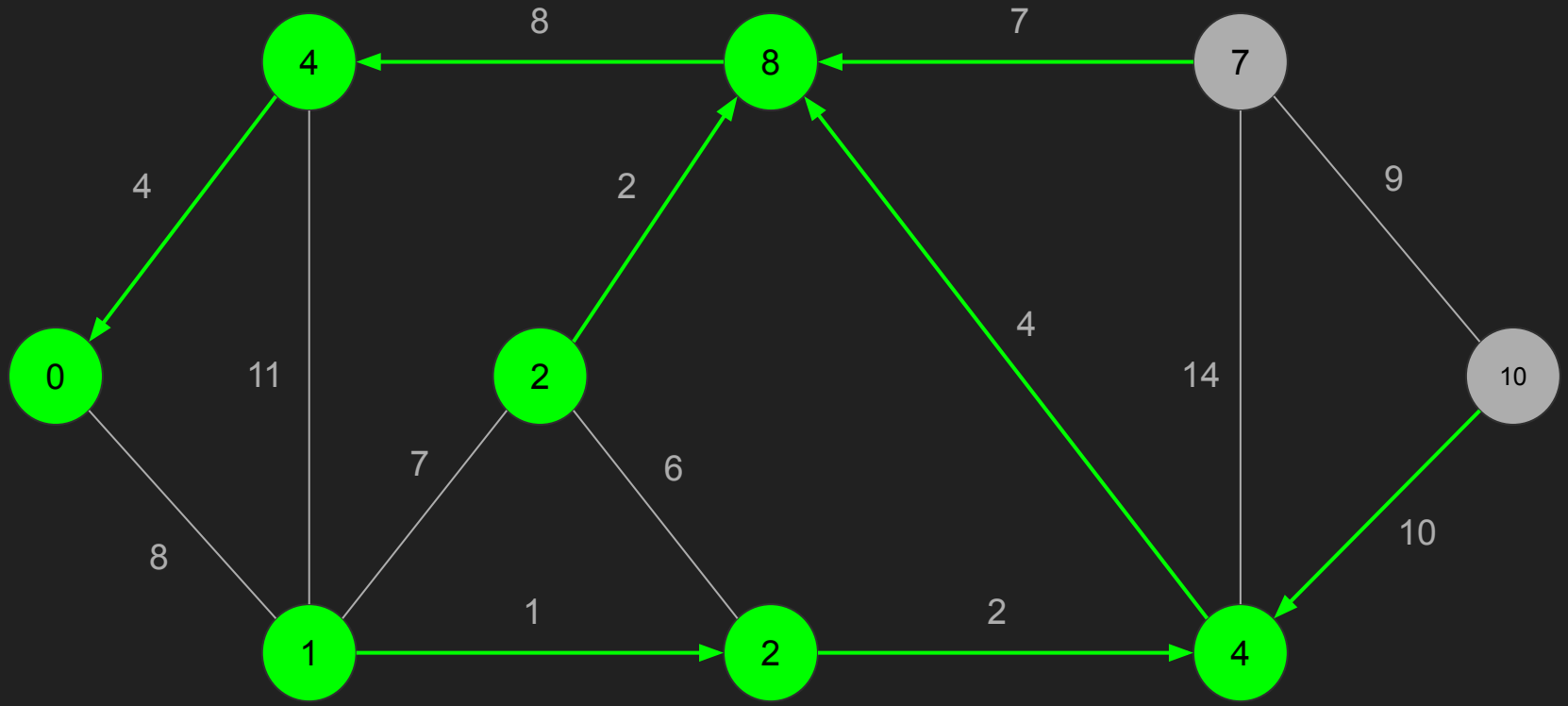

4
8
7
0
2
∞
7
6
4
8
4
11
7
2
14
9
8
1
2
10

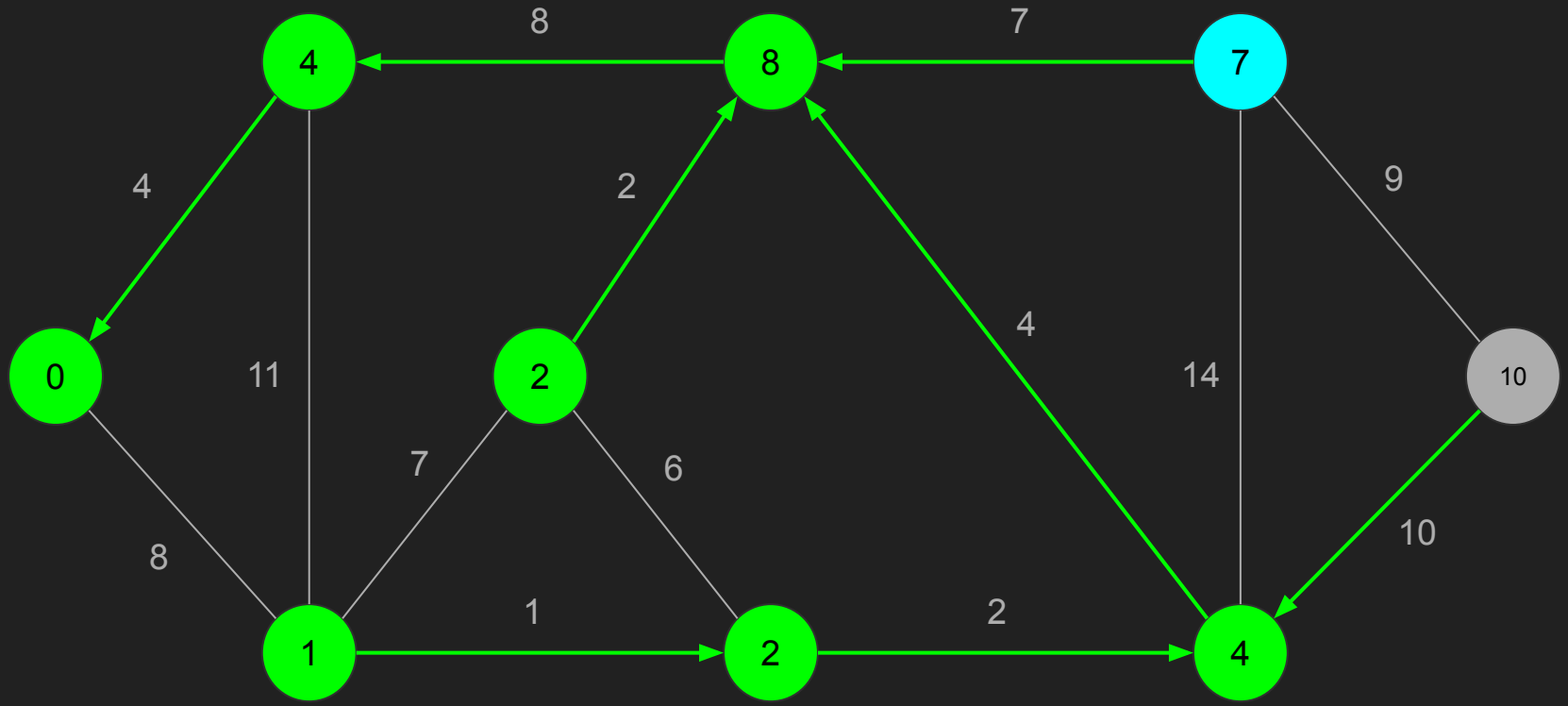

4
8
7
0
2
10
7
2
4
8
11
7
6
14
9
4
10
8
1
2
7
2
4

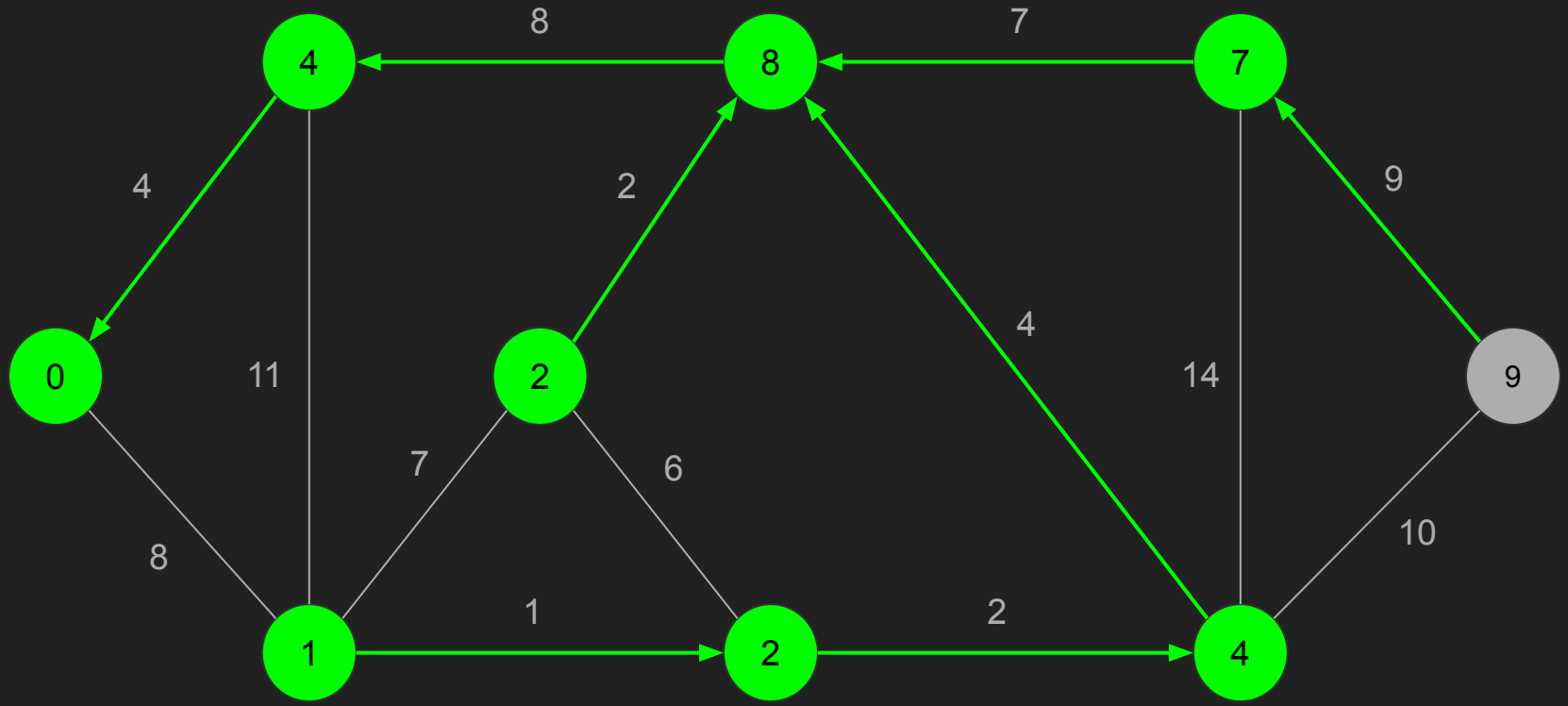

4
8
8
7
7
4
2
9
0
11
2
4
14
10
7
6
8
10
1
2
7
2
4

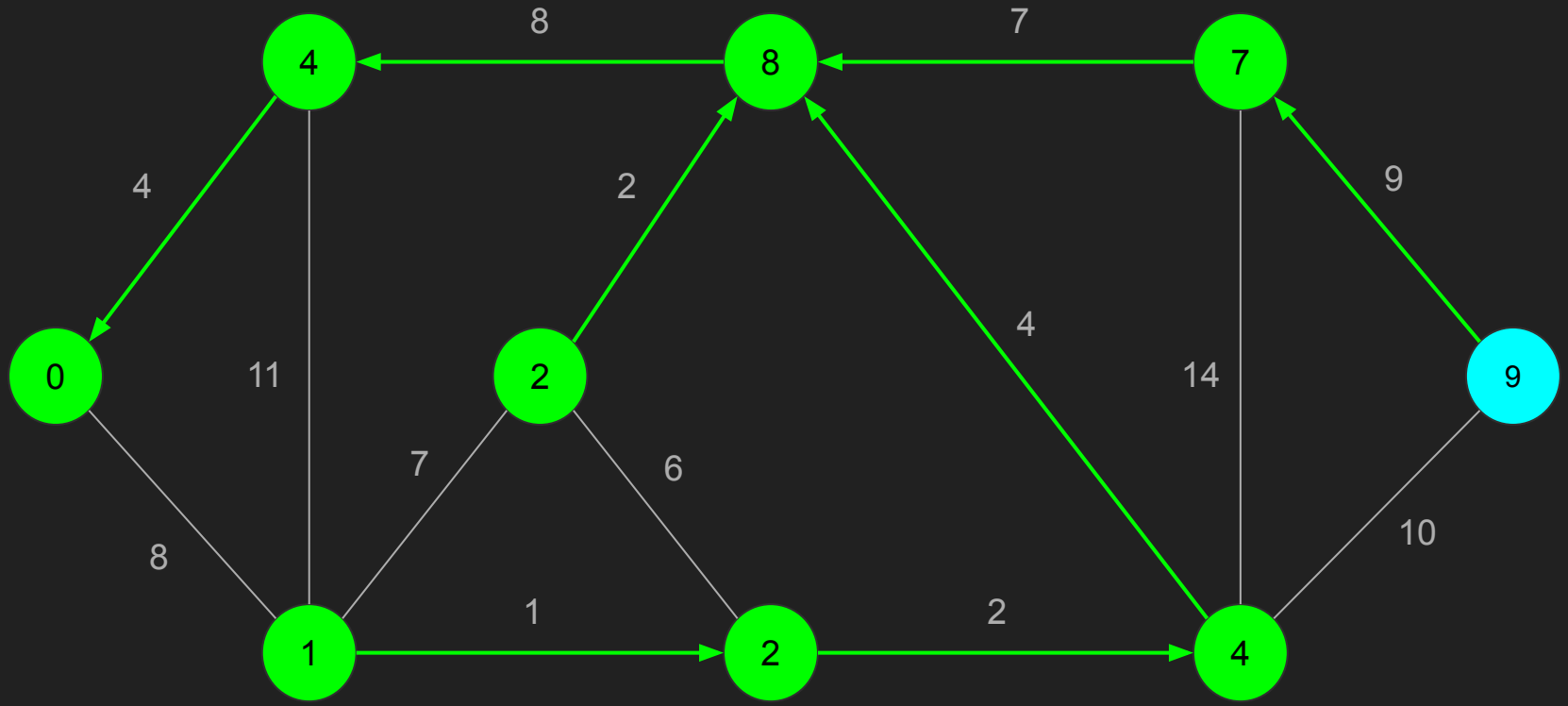

4
8
7
0
2
7
10
1
2
4
8
11
7
6
4
14
9
4
2
8
1
2
10

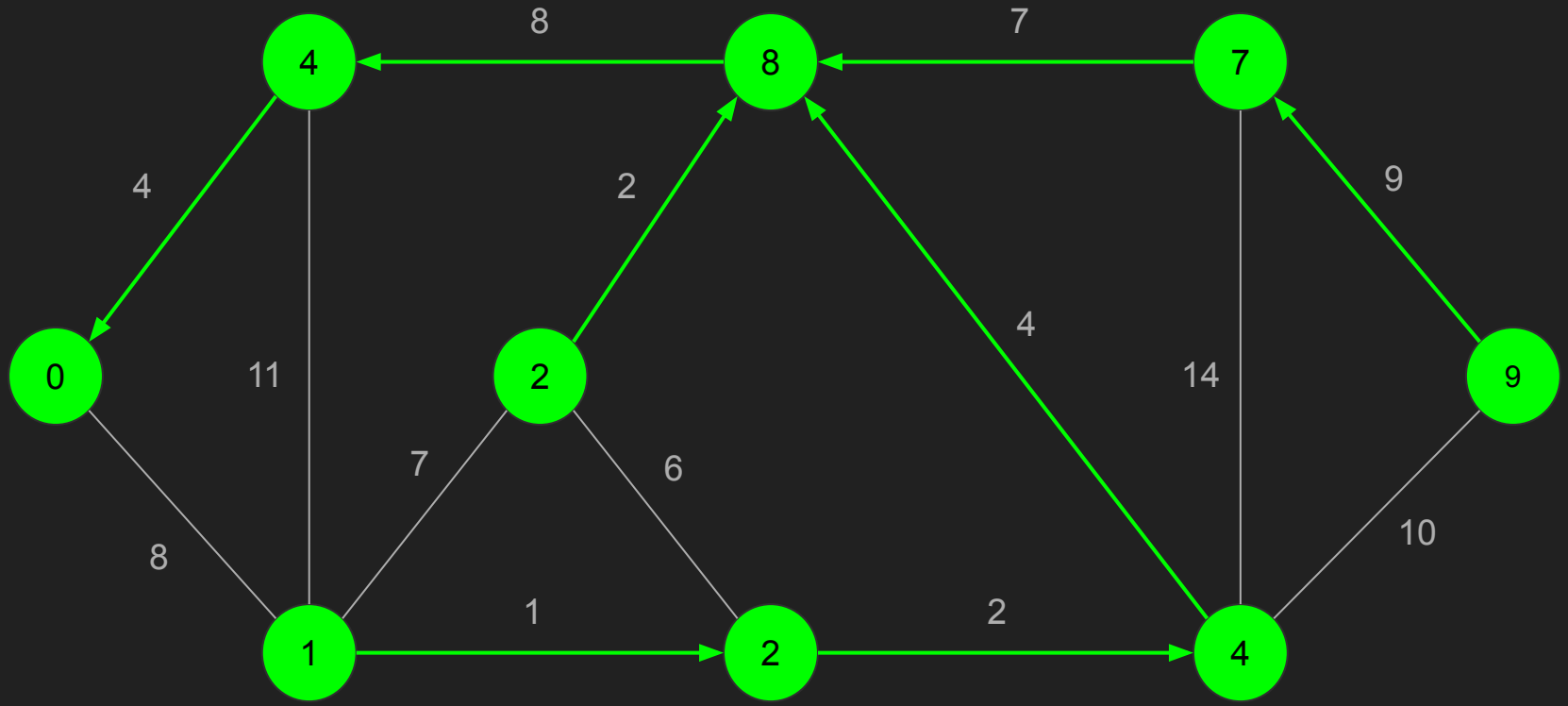

4
8
7
0
2
7
10
1
2
4
8
11
7
6
4
14
9
4
2
8
1
2
10


4
8
7
0
2
7
10
1
2
4
8
11
7
6
4
14
9
10
1
2
4


4
8
7
0
2
7
10
1
2
4
8
11
7
6
4
14
9
10
1
2
4


4
8
7
0
2
9
1
2
4
8
4
11
2
4
14
9
7
6
8
1
2
10


8
7
4
8
7
4
2
9
4
0
11
2
14
9
7
6
8
10
1
2
1
2
4


4
8
7
0
2
9
1
2
4
8
4
11
2
4
14
9
7
6
8
10
1
2

Correctness

Loop invariant: the vertices not in Q and associated π edges are part of a minimum spanning tree

Key step: the edge selected at every step of the loop maintains the invariant

Theorem 23.1 — Let (G, w) be a connected undirected weighted graph. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V^G-S) be a cut that respects A, and let u be an edge crossing the cut with minimum weight among all edges crossing the cut. Then edge u is safe for A.

Running Time Analysis

Given a good priority queue implementation

- e.g., AVL trees with minimum/maximum pointers
- INSERT : Θ(log n)
- EXTRACTMIN: Θ(log n)
- DECREASEKEY: Θ(log n)

Add every vertex to the priority queue

Loop through every vertex (EXTRACTMIN) and every edge (DECREASEKEY)

Θ(V log V + V log V + E log V) ~ Θ(E log V) if graph is connected

Shortest Paths

Given a directed weighted graph (G, w)

Shortest distance between u and v in G is:

$$\partial(u,v) = \min \{ w(p) : u \overset{p}{\rightsquigarrow} v \} \quad \text{if there is a path from u to v}$$
$$= \infty \quad \text{otherwise}$$

A path $u \overset{p}{\rightsquigarrow} v$ is a shortest path if $w(p) = \partial(u, v)$

Question: how do you compute shortests paths efficiently?

Computing Shortests Paths

Breadth-first search computes shortest paths in unweighted graphs

- all shortests path starting from a given source vertex
- d[u] = distance from source vertex to u
- π[u] = predecessor of u in the shortest path from s to u
- generalize the idea

Technique: relaxation

- maintain an estimate d[u]
- d[u] : upper bound on the the weight of shortest path from s to u
- refine that estimate during processing

Dijkstra's Algorithm

Also a greedy algorithm

Maintain a set S of vertices whose shortest paths from s have been computed

Repeatedly select a vertex $u \in V - S$ with minimum shortest path estimate

- add to S
- relax all edges from u

Use a Priority Queue to select vertex with minimum shortest path estimate

- key for priority queue: d[u]

Dijkstra's Algorithm

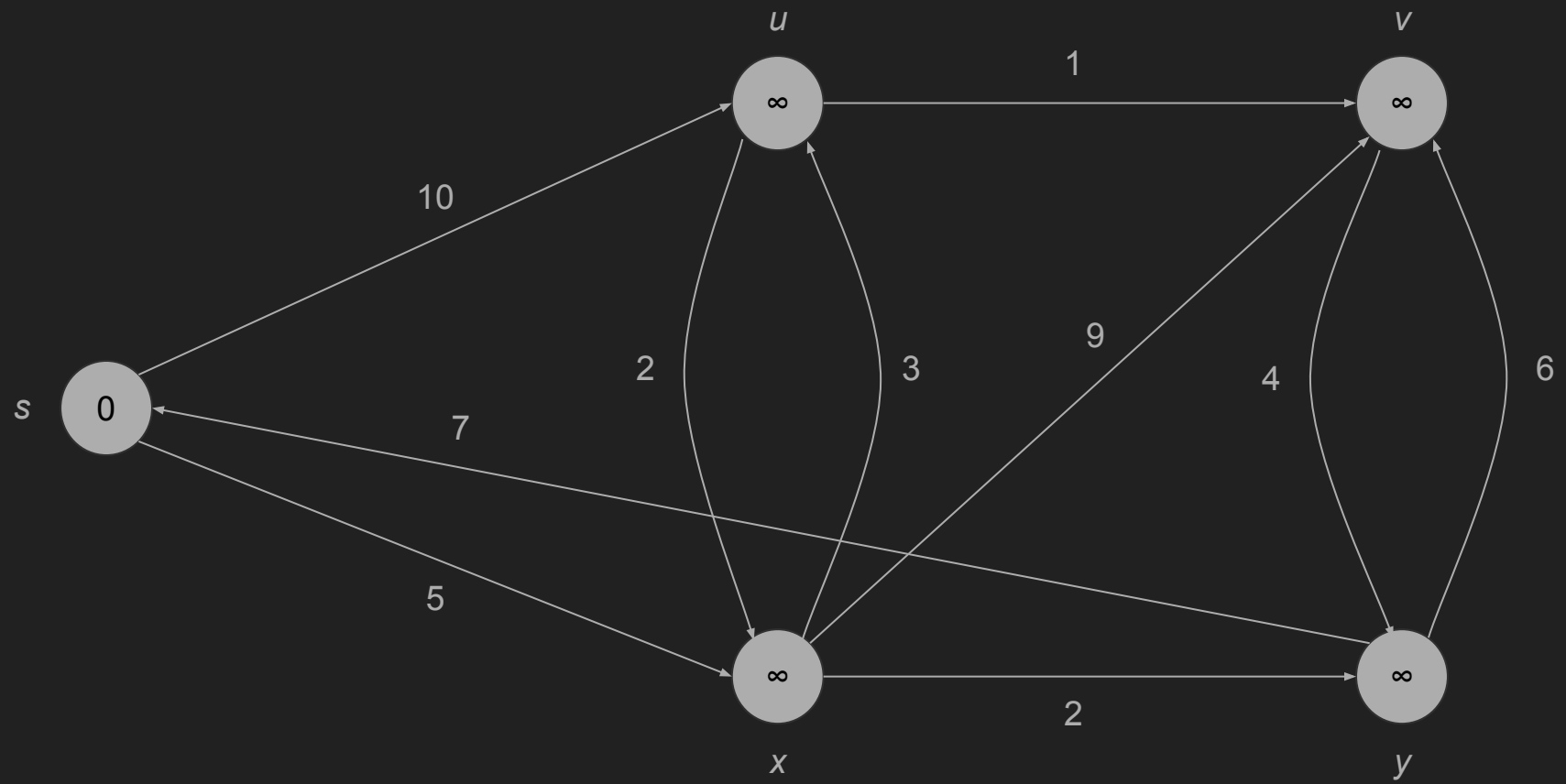
Given directed weighted graph (G, w) and vertex s ∈ V

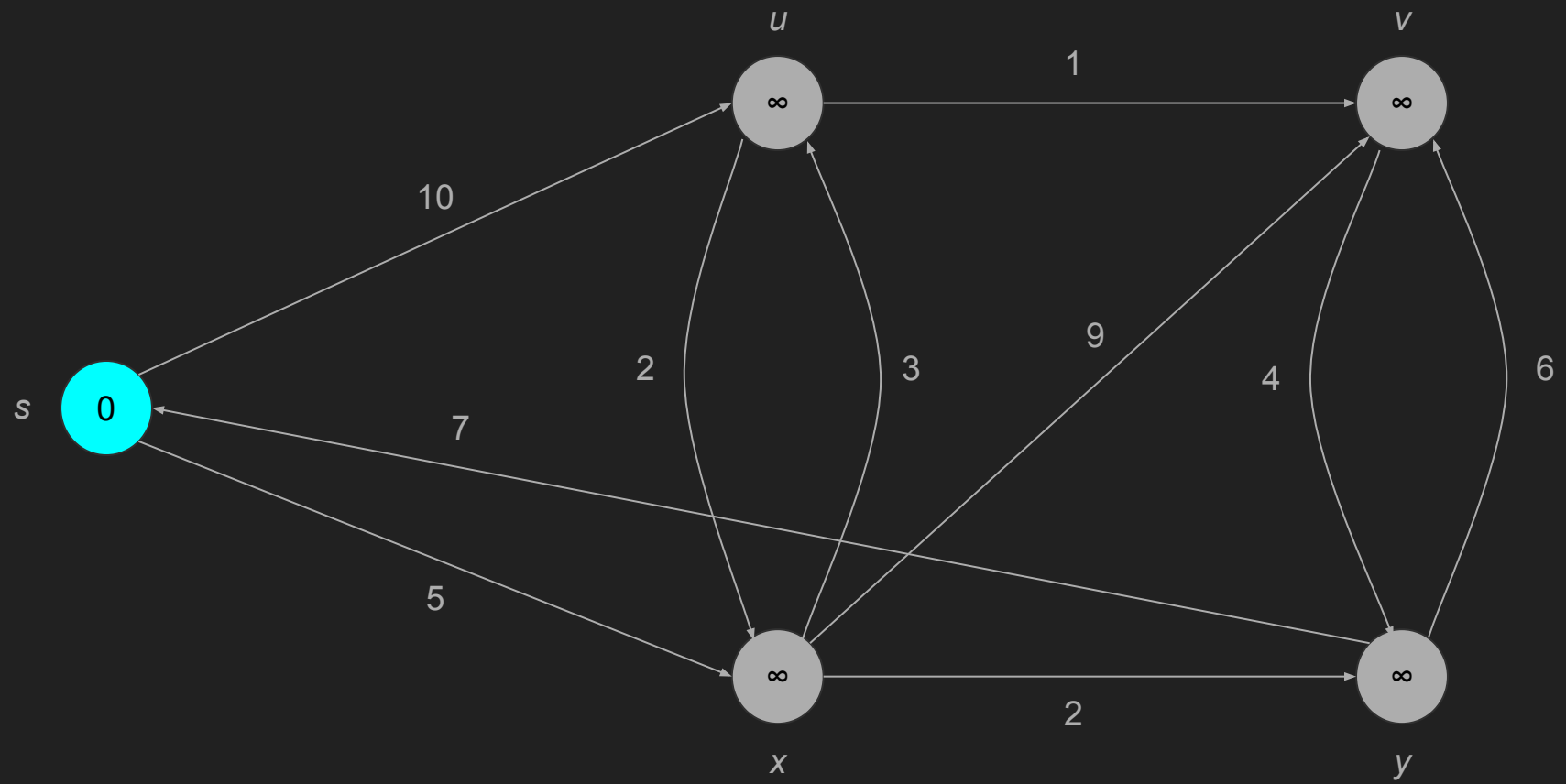
```
for v ∈ V
    d[v] ← ∞
    π[v] ← NIL
    INSERT(Q, v)
d[s] ← 0
while Q ≠ ∅
    u ← EXTRACTMIN(Q)
    for v ∈ Adj[u]
        RELAX(u, v, w)
```

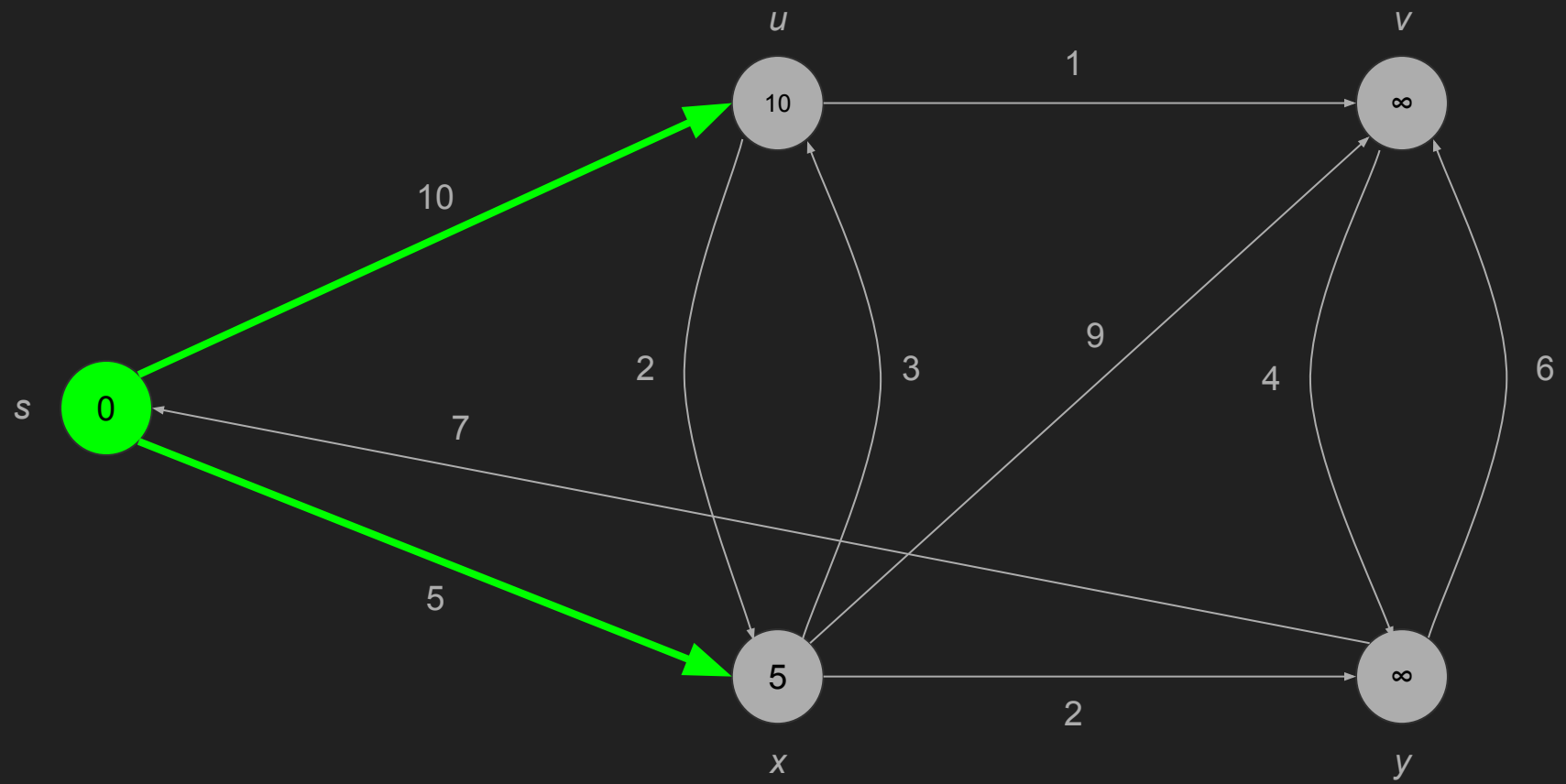
Dijkstra's Algorithm

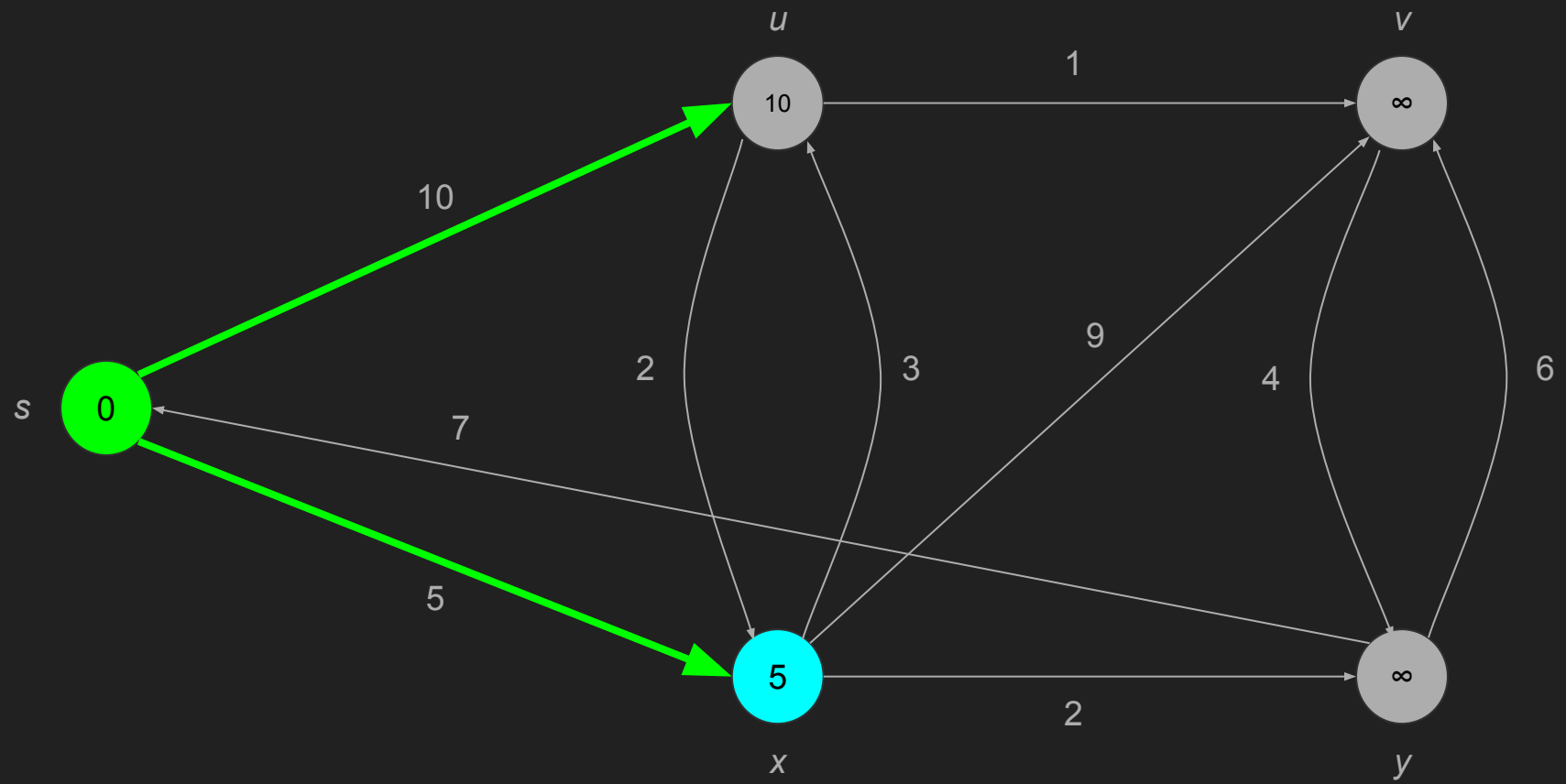
Given directed weighted graph (G, w) and vertex s ∈ V

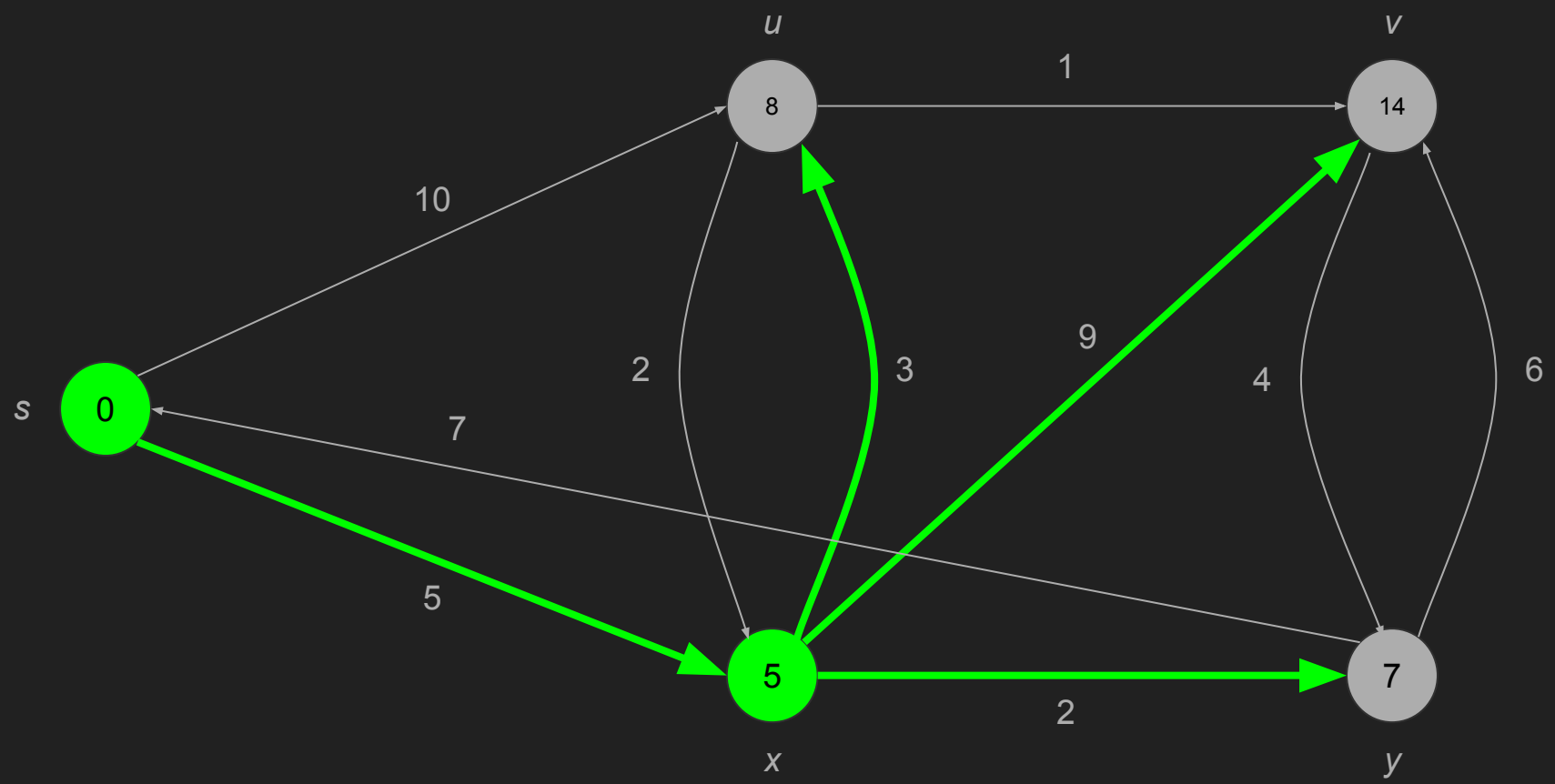
```
for v ∈ V
    d[v] ← ∞
    π[v] ← NIL
    INSERT(Q, v)
d[s] ← 0
while Q ≠ ∅
    u ← EXTRACTMIN(Q)
    for v ∈ Adj[u]
        if d[v] > d[u] + w(u, v)
            d[v] ← d[u] + w(u,v)
            π[v] ← u
```

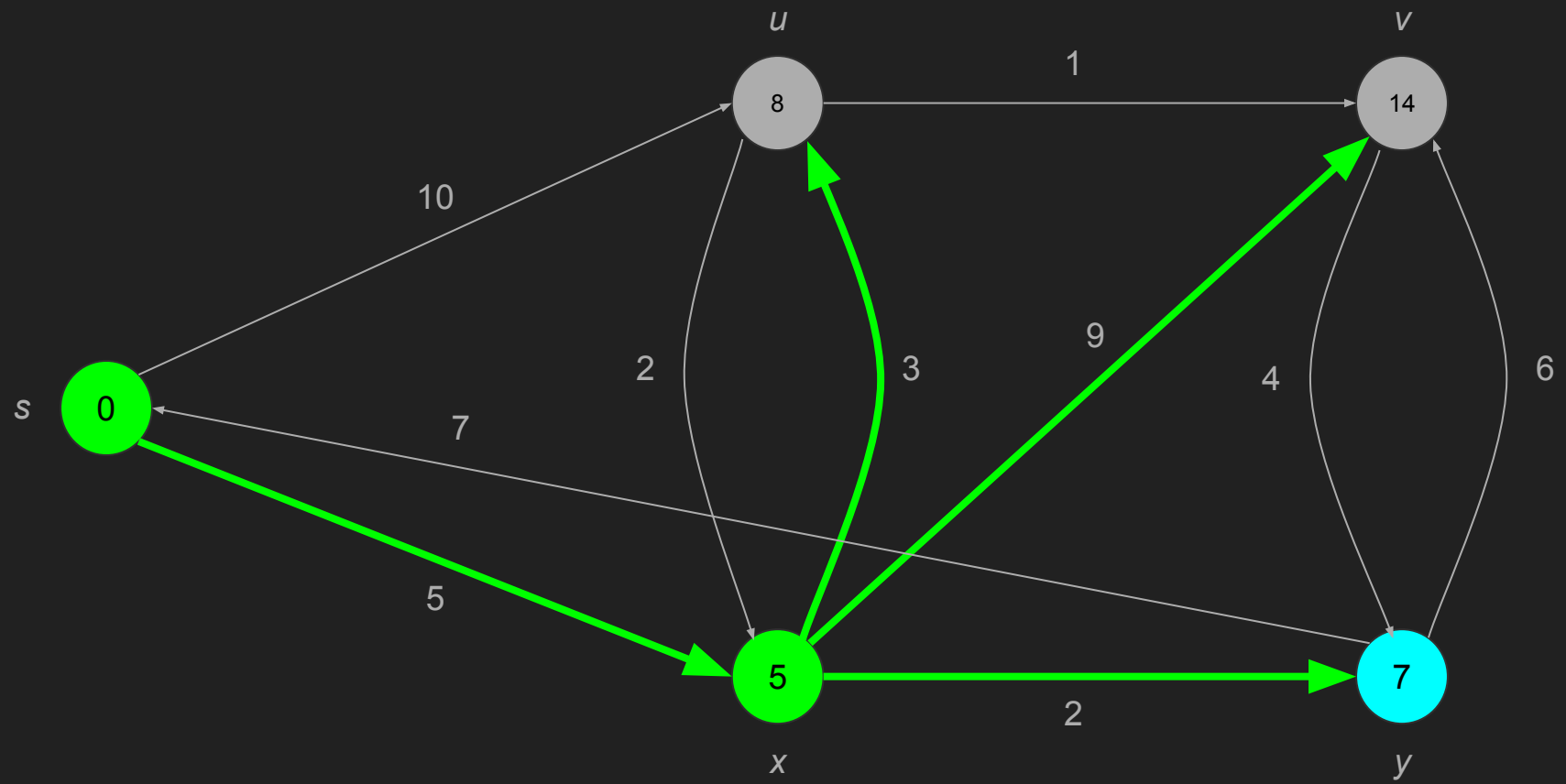

u
v
1
∞
∞
10
2
3
9
4
6
s
0
7
5
∞
∞
2
x
y

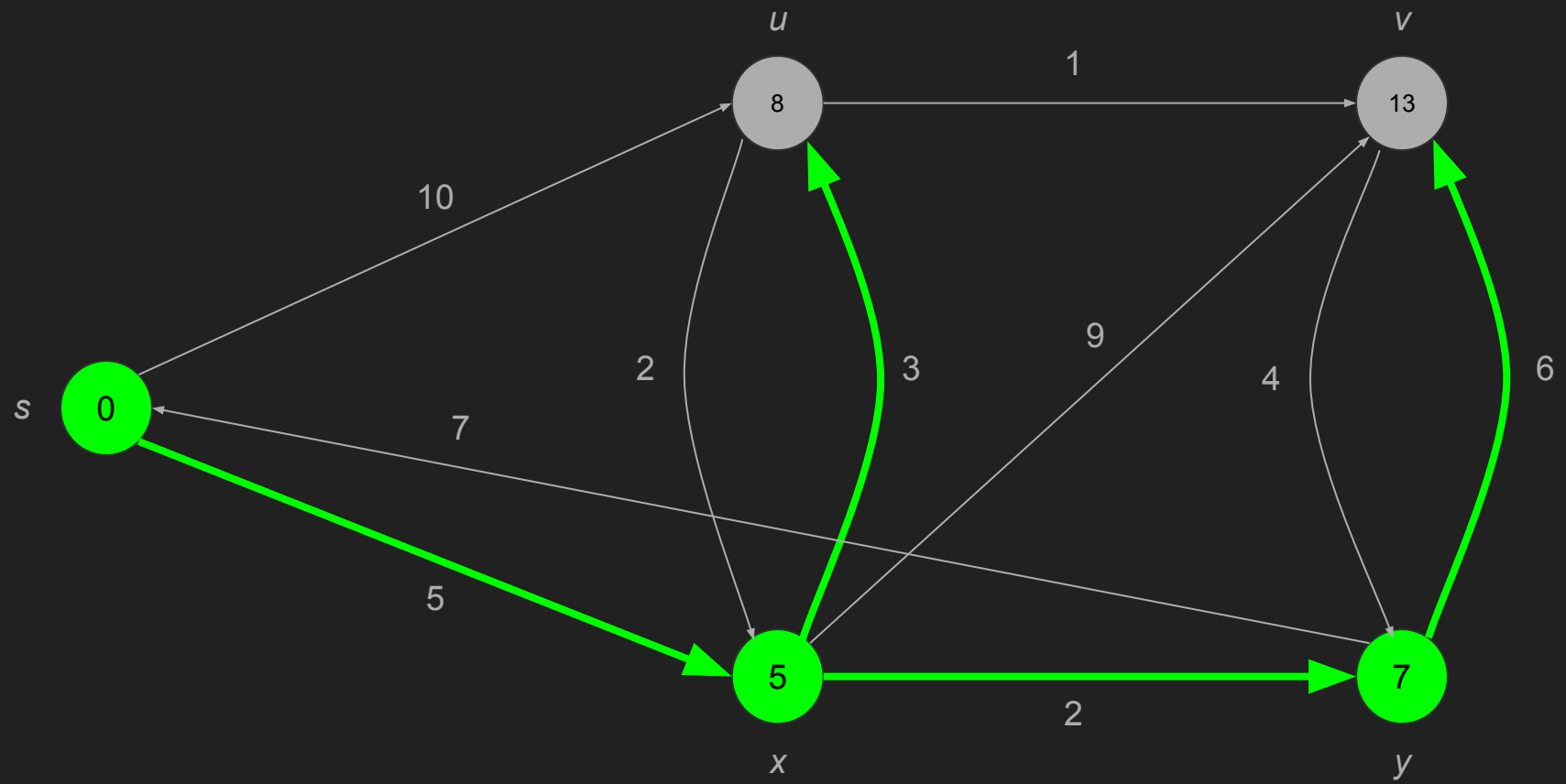

u
v
1
∞
∞
10
2
3
9
4
6
s
0
7
5
∞
∞
2
x
y

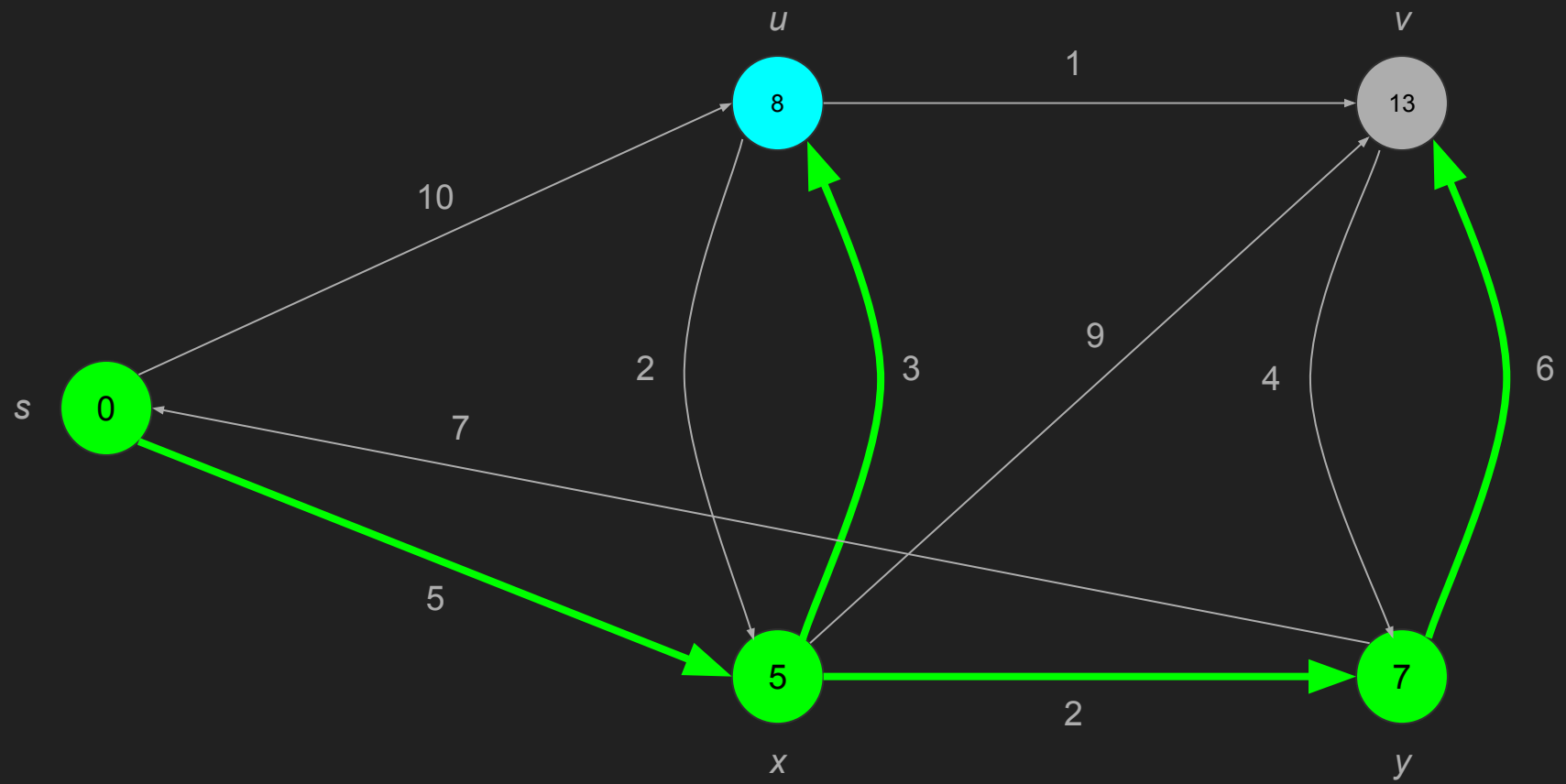

u
v
10
1
∞
10
s
0
2
3
9
4
6
7
5
5
∞
2
x
y

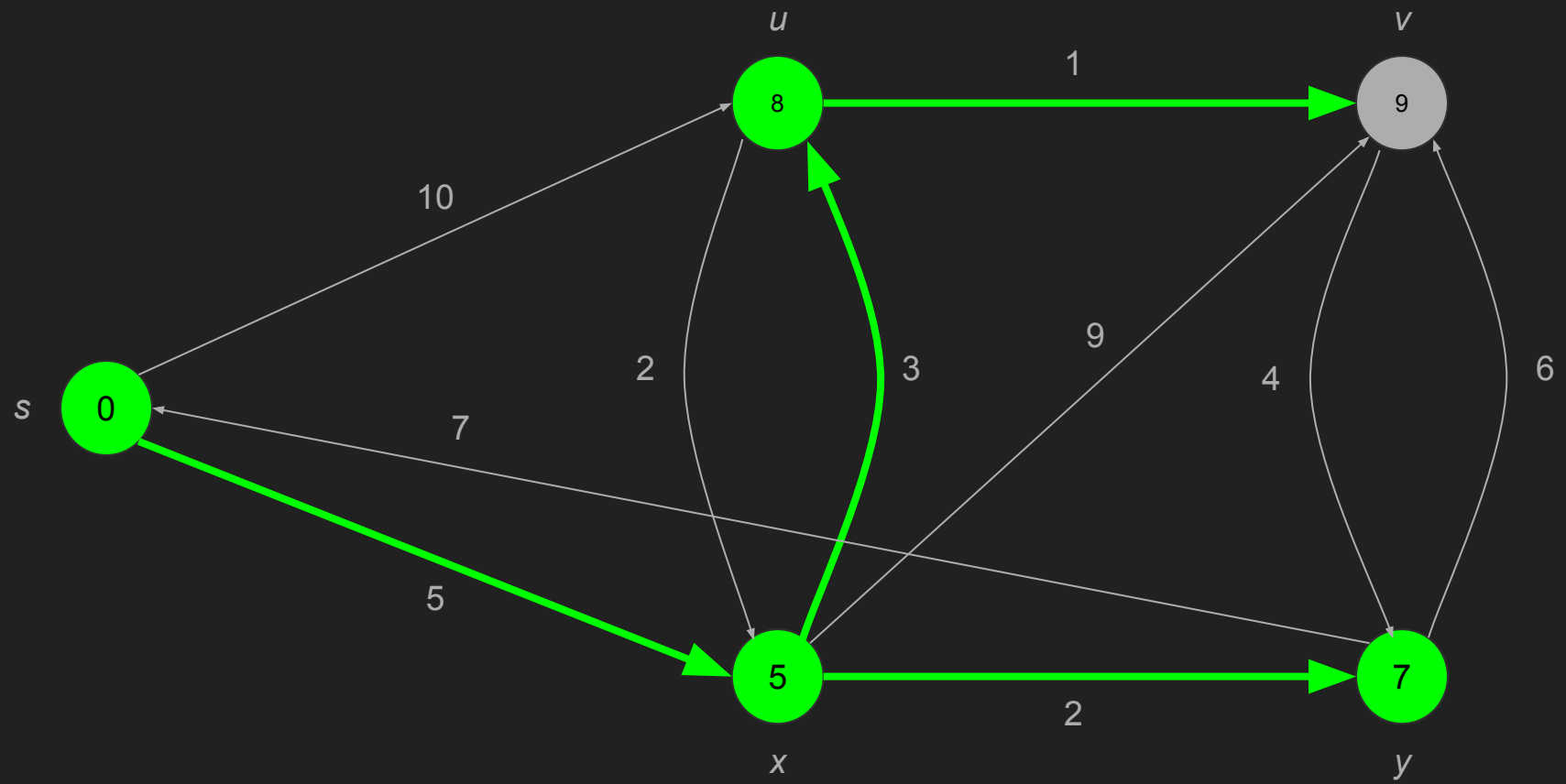

u
v
10
1
∞
10
s
0
2
3
9
4
6
7
5
5
∞
2
x
y

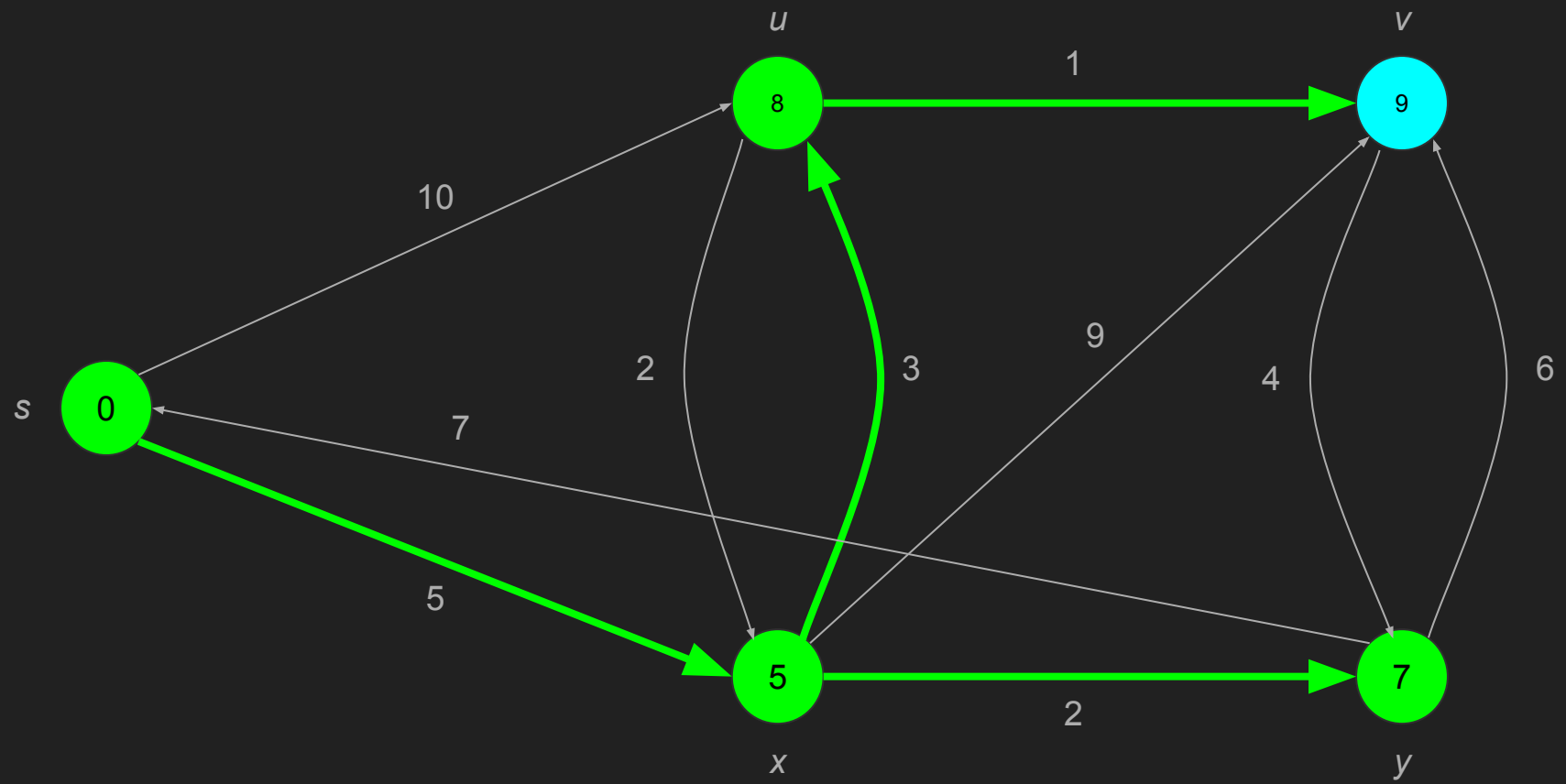

u
v
1
8
14
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y

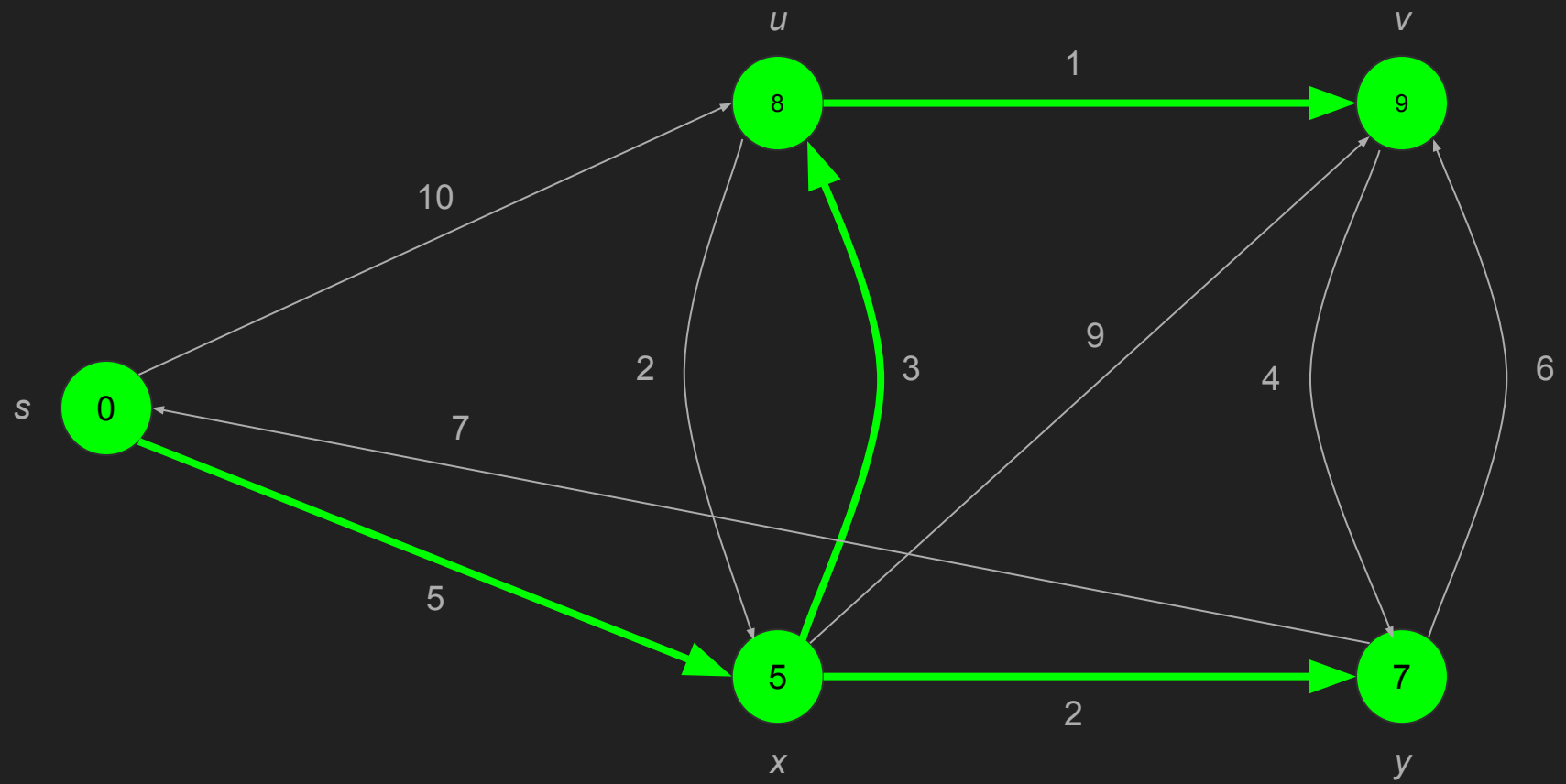

u
v
1
8
14
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y


u
v
1
8
13
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y


u
v
8
13
1
10
s
0
2
3
9
4
6
7
5
5
7
2
x
y


u
v
1
8
9
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y


u
v
1
8
9
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y


u
v
1
8
9
10
2
3
9
4
6
s
0
7
5
5
7
2
x
y

Correctness

Loop invariant: At the start of each iteration of the while loop

$$d[v] = \partial(s, v)$$

for every vertex $v \in S$

The argument is by contradiction — see textbook — Theorem 24.6)

Running Time Analysis

Similar to Prim's Algorithm:

Add every vertex to priority queue

Loop through every vertex (ExtractMin) and every edge (DecreaseKey)

$\Theta(V \log V + V \log V + E \log V) \sim \Theta(E \log V)$ if graph is connected

Can do better with Fibonacci heaps — $\Theta(V \log V + E)$