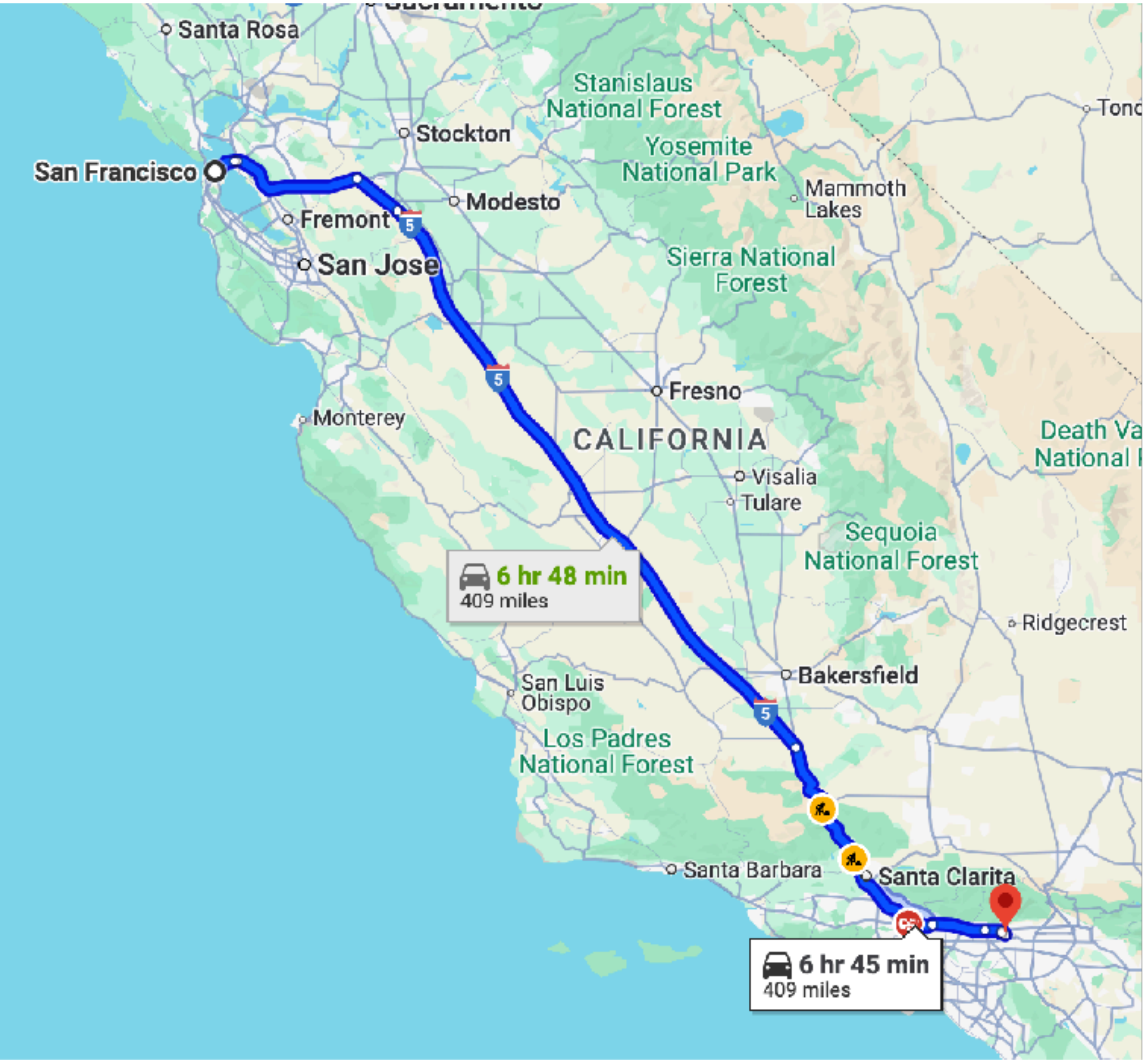
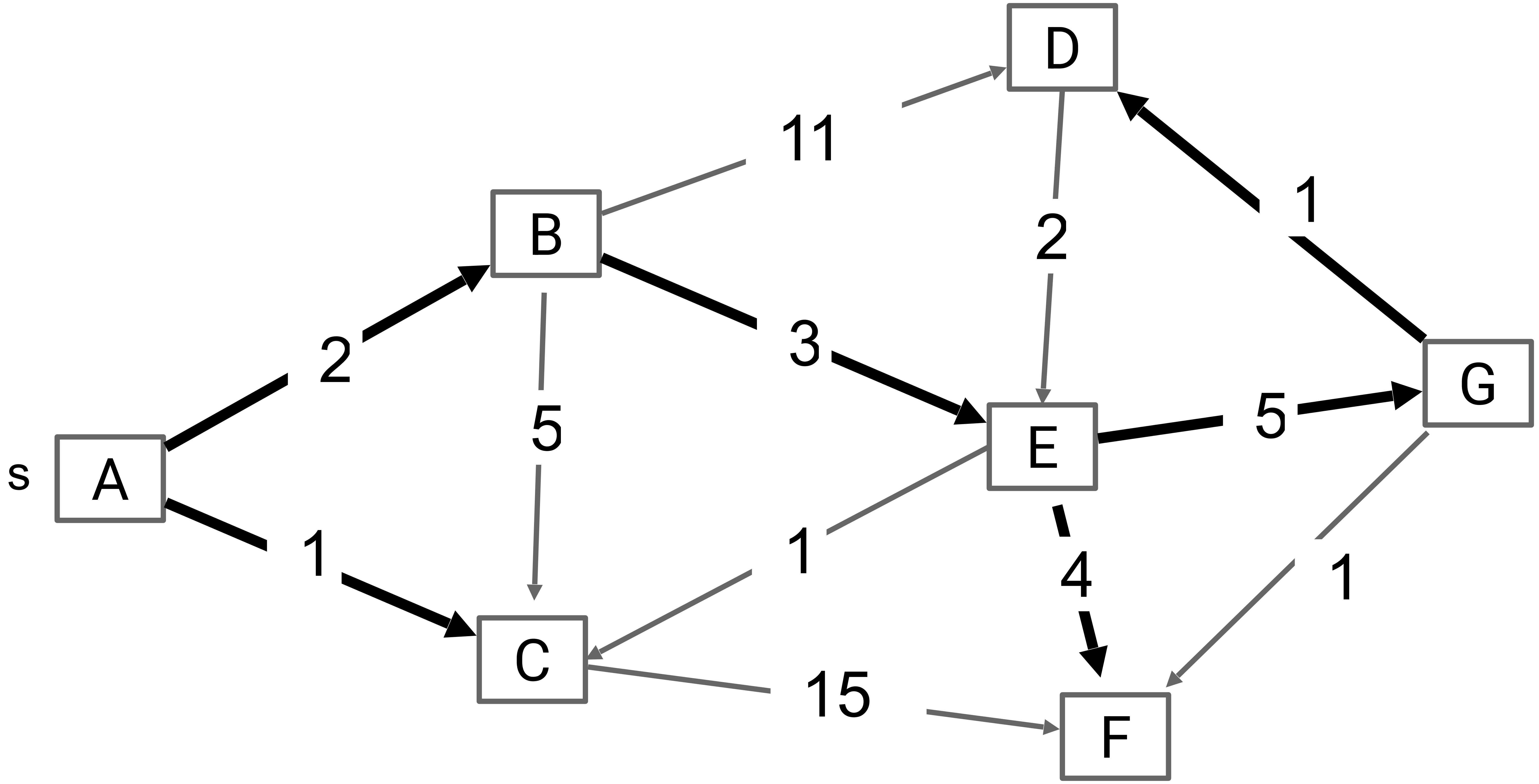
CS62 Class 23: Shortest paths

Graphs





Hashtable review

Insert the keys into both a separate chaining and open addressing (with linear probing) hashtable M = 5. Assume the hash function h(k) = k % M. Assume no resizing.

1, 12, 22, 32, 42

1) What is the length of the longest chain?

2) What is the average number of probes per insertion?

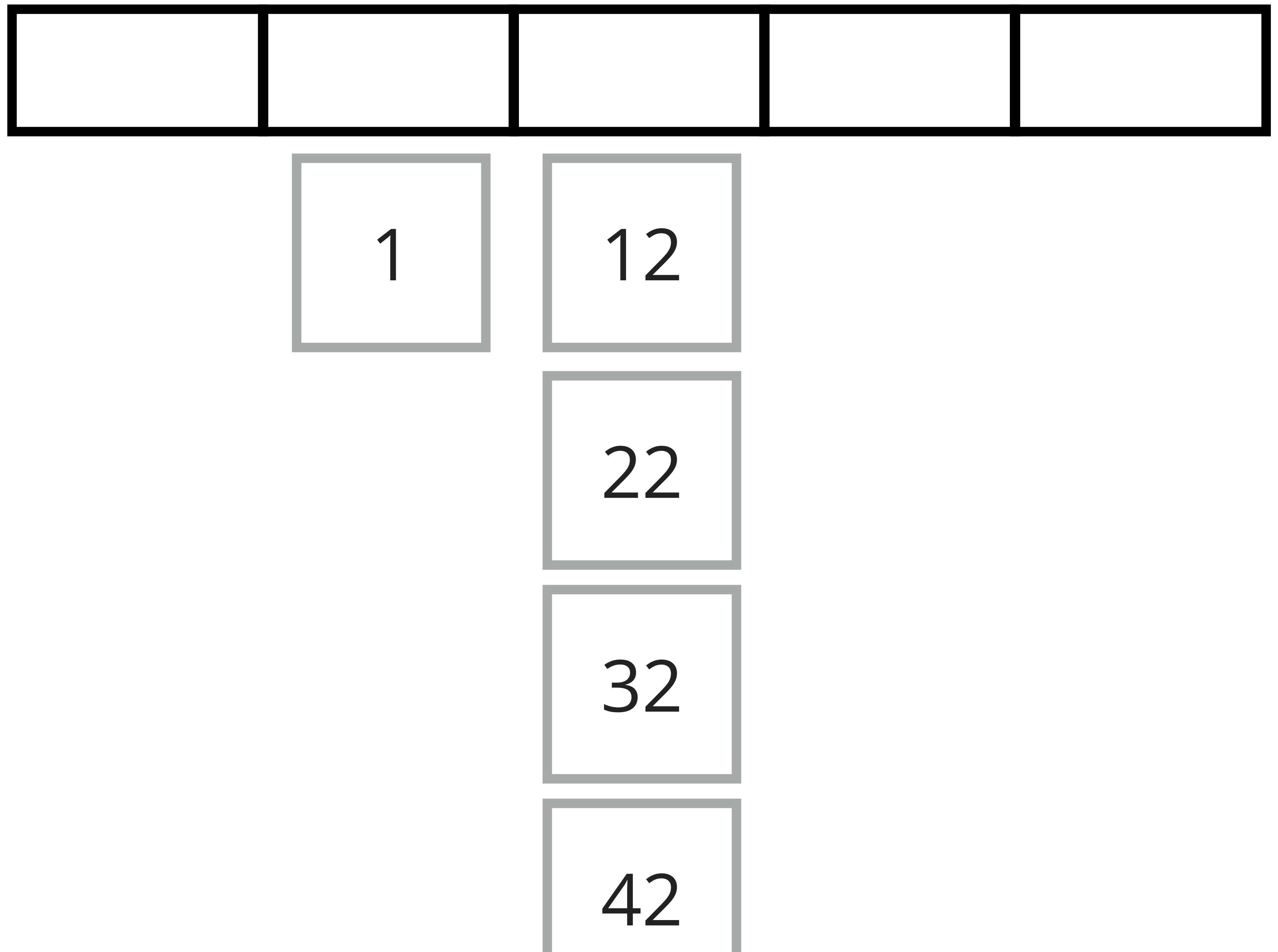
Hashtable review

Insert the keys into both a separate chaining and open addressing (with linear probing) hashtable M = 5. Assume the hash function h(k) = k % M. Assume no resizing.

1, 12, 22, 32, 42

1) What is the length of the longest chain? - **4**
2) What is the average number of probes per insertion? - **2.2**

separate chaining



open addressing

42	1	12	22	32

Probes: 1 (1) + 1 (12) + 2 (32) + 3 (42) + 4 (42) =
(1+1+2+3+4)/5 = 2.2

Graph Problems

Last time, saw two ways to find paths in a graph, DFS & BFS.

Give an example of a graph that would make the space efficiency bad for DFS or BFS.

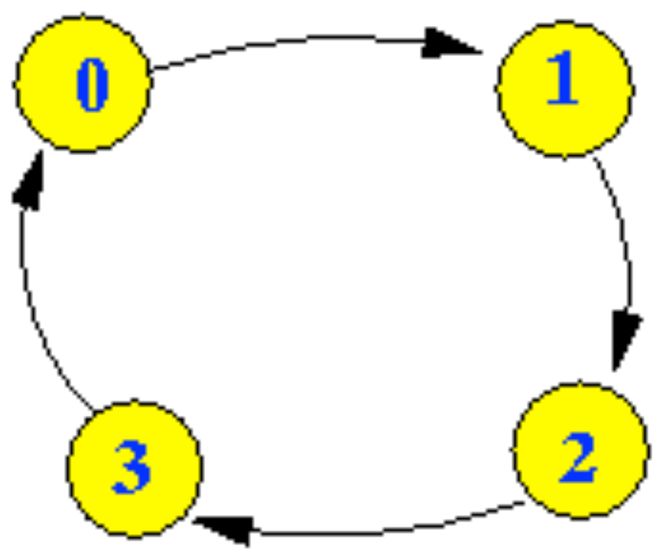
Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DFS	$O(V+E)$ time $\Theta(V)$ space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BFS	$O(V+E)$ time $\Theta(V)$ space

BFS vs. DFS for space efficiency

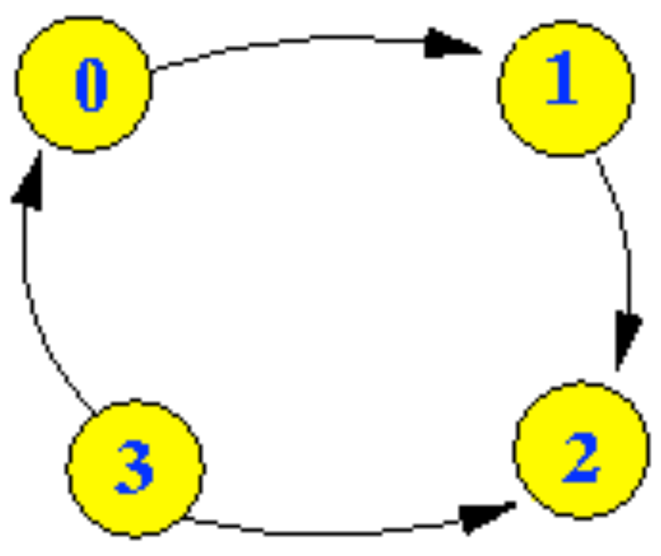
- DFS is worse for spindly graphs.
  - Call stack gets very deep.
  - Computer needs Θ(V) memory to remember recursive calls.
- BFS is worse for absurdly “bushy” graphs.
  - Queue gets very large. In worst case, queue will require Θ(V) memory.
  - Example: 1,000,000 vertices that are all connected. 999,999 will be enqueued at once.
- Note: In our implementations, we have to spend Θ(V) memory anyway to track distTo and edgeTo arrays.
  - Can optimize by storing distTo and edgeTo in a map instead of an array.

Strongly connected digraph algorithm

- A strongly connected digraph is a directed graph in which it is possible to reach any vertex starting from any other vertex by traversing edges.
- Pick a random starting vertex s.
- Run DFS/BFS starting at s.
  - If have not reached all vertices, return false.
- Reverse edges.
- Run DFS/BFS again on reversed graph.
  - If have not reached all vertices, return false.
  - Else return true.



Strongly connected



Not strongly connected

Agenda

- Edge-weighted graphs
- Shortest paths
- Dijkstra’s algorithm

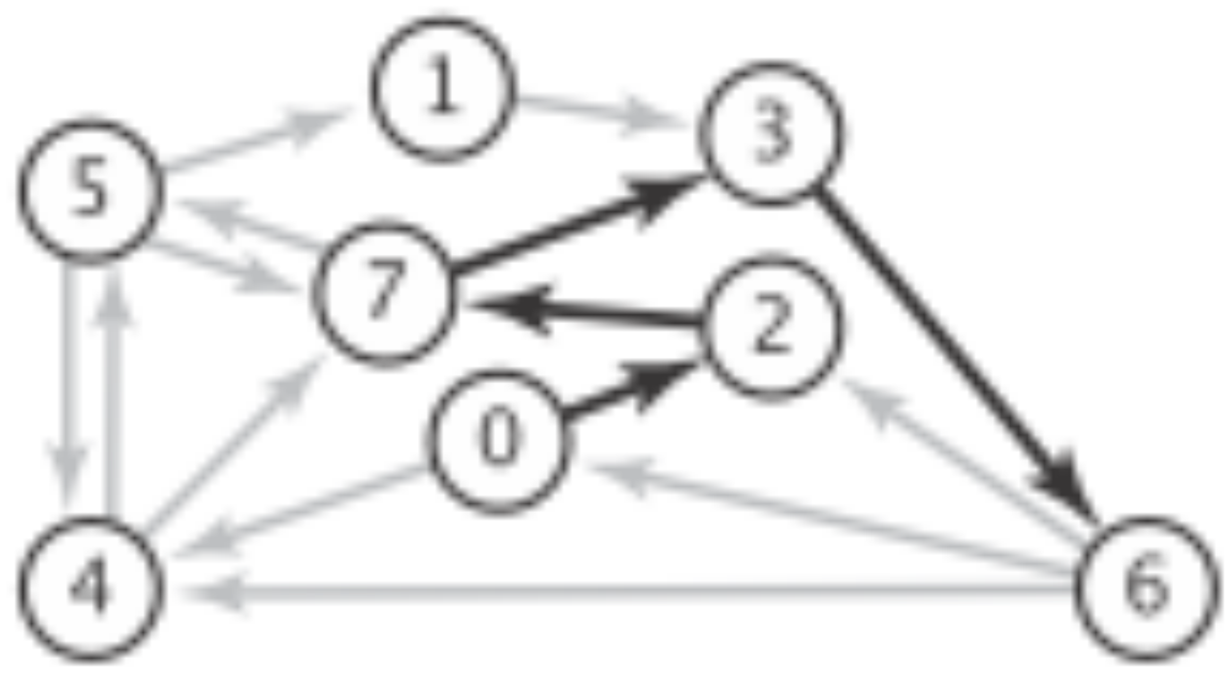
Edge-weighted graph

Edge-weighted graphs

- Edge-weighted digraph: a digraph where we associate weights/costs with each edge.
- Shortest path from vertex s to vertex t: a directed path from s to t with the property that no other such path has a lower weight (total weight sum of edges it consists of).
- Assumptions:
  - Not all vertices need to be reachable.
  - Weights are positive.
  - Shortest paths are not necessarily unique but they are simple.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Weighted directed edge API

only difference is we now have weights

- `public class DirectedEdge`
  - `DirectedEdge(int v, int w, double weight)`
    - Constructs a weighted edge from v to w `(v->w)` with the provided `weight`.
  - `int from()`
    - Returns vertex source of this edge.
  - `int to()`
    - Returns vertex destination of this edge.
  - `double weight()`
    - Returns weight of this edge.
  - `String toString()`
    - Returns the string representation of this edge.

Weighted directed edge in Java

```
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

   public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

   public int from() {
        return v;
    }

    public int to() {
        return w;
    }

    public double weight() {
        return weight;
    }
```

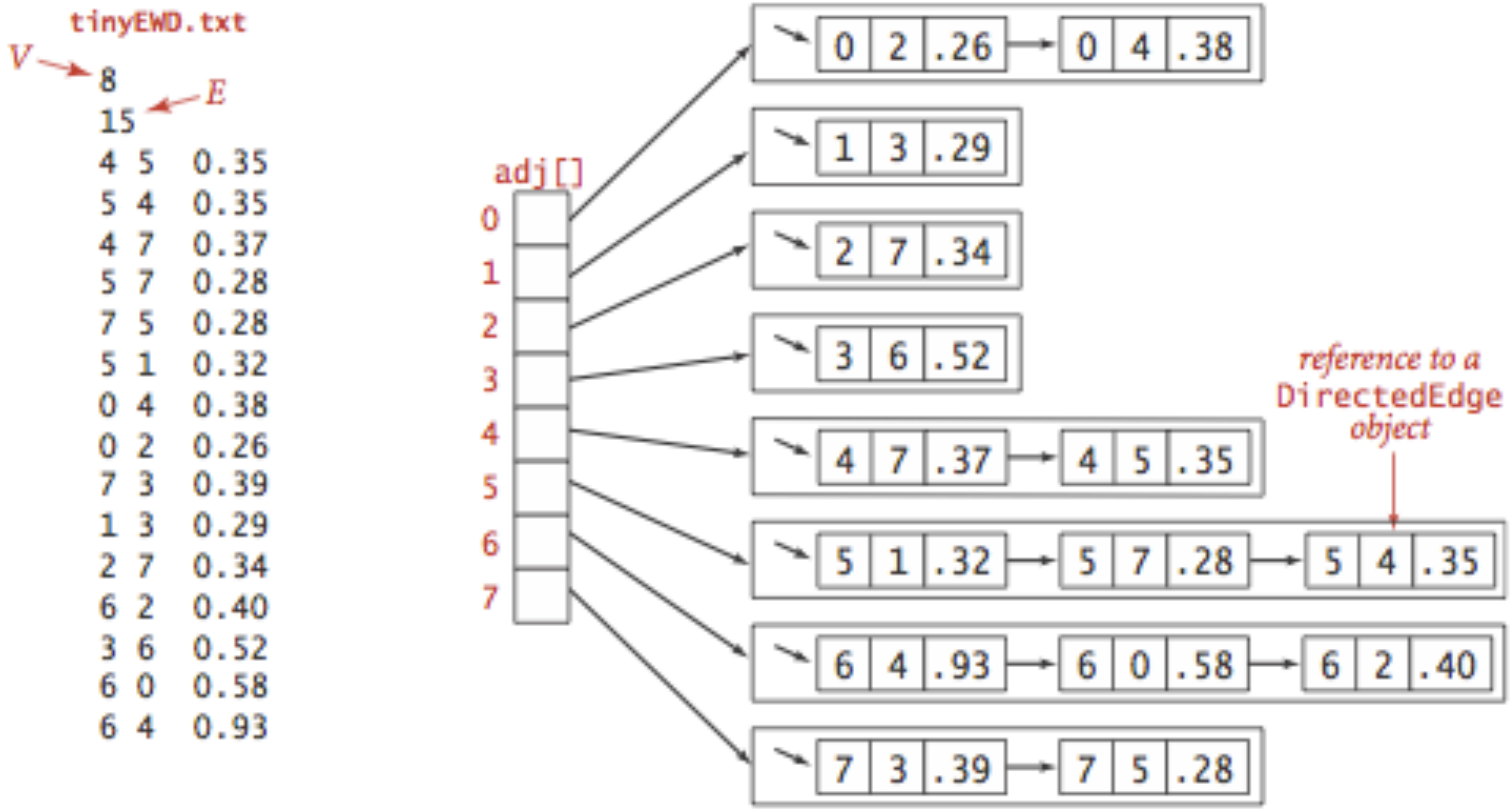
Edge-weighted digraph API

- `public class EdgeWeightedDigraph`
  - `EdgeWeightedDigraph(int v)`
    - Constructs an edge-weighted digraph with v vertices.
  - `void addEdge(DirectedEdge e)`
    - Add weighted directed edge e.
  - `Iterable<DirectedEdge> adj(int v)`
    - Returns edges adjacent from v.
  - `int V()`
    - Returns number of vertices.
  - `int E()`
    - Returns number of edges.
  - `Iterable<DirectedEdge> edges()`
    - Returns all edges.

only difference is edges are DirectedEdge objects instead of integers

Edge-weighted digraph adjacency list representation

- `public class EdgeWeightedDigraph`
- `EdgeWeightedDigraph(int v)`
  - Constructs an edge-weighted digraph with V vertices.
- `void addEdge(DirectedEdge e)`
  - Add weighted directed edge e.
- `Iterable<DirectedEdge> adj(int v)`
  - Returns edges adjacent from v.
- `int V()`
  - Returns number of vertices.
- `int E()`
  - Returns number of edges.
- `Iterable<DirectedEdge> edges()`
  - Returns all edges.



Edge-weighted digraph representation

Edge-weighted digraph in Java

```
public class EdgeWeightedDigraph {
    private final int V;                  // number of vertices in this digraph
    private int E;                        // number of edges in this digraph
    private SinglyLinkedList<DirectedEdge> adj[];
    // adj[v] = adjacency list for v

    public EdgeWeightedDigraph(int V) {
       this.V = V;
       this.E = 0;
       adj = new SinglyLinkedList<DirectedEdge>[V];
        for (int v = 0; v < V; v++)
            adj[v] = new SinglyLinkedList<DirectedEdge>();
    }
    public void addEdge(DirectedEdge e) {
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        E++;
    }

   public Iterable<DirectedEdge> adj(int v) {
       return adj[v];
    }
```

DirectedEdge instead of int

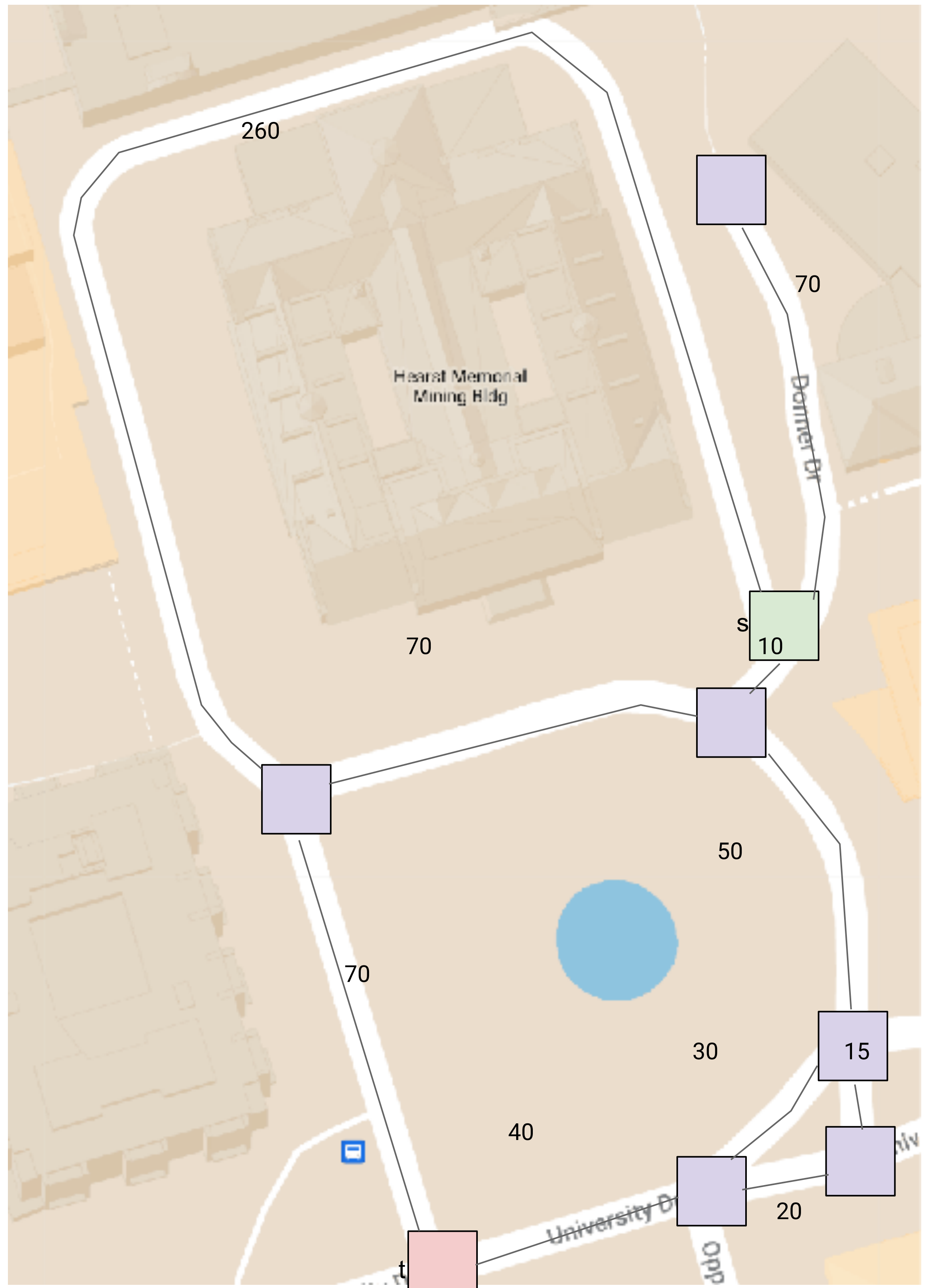
extract v & w with .from() and .to() getters

Shortest paths

BreadthFirstSearch for Google Maps

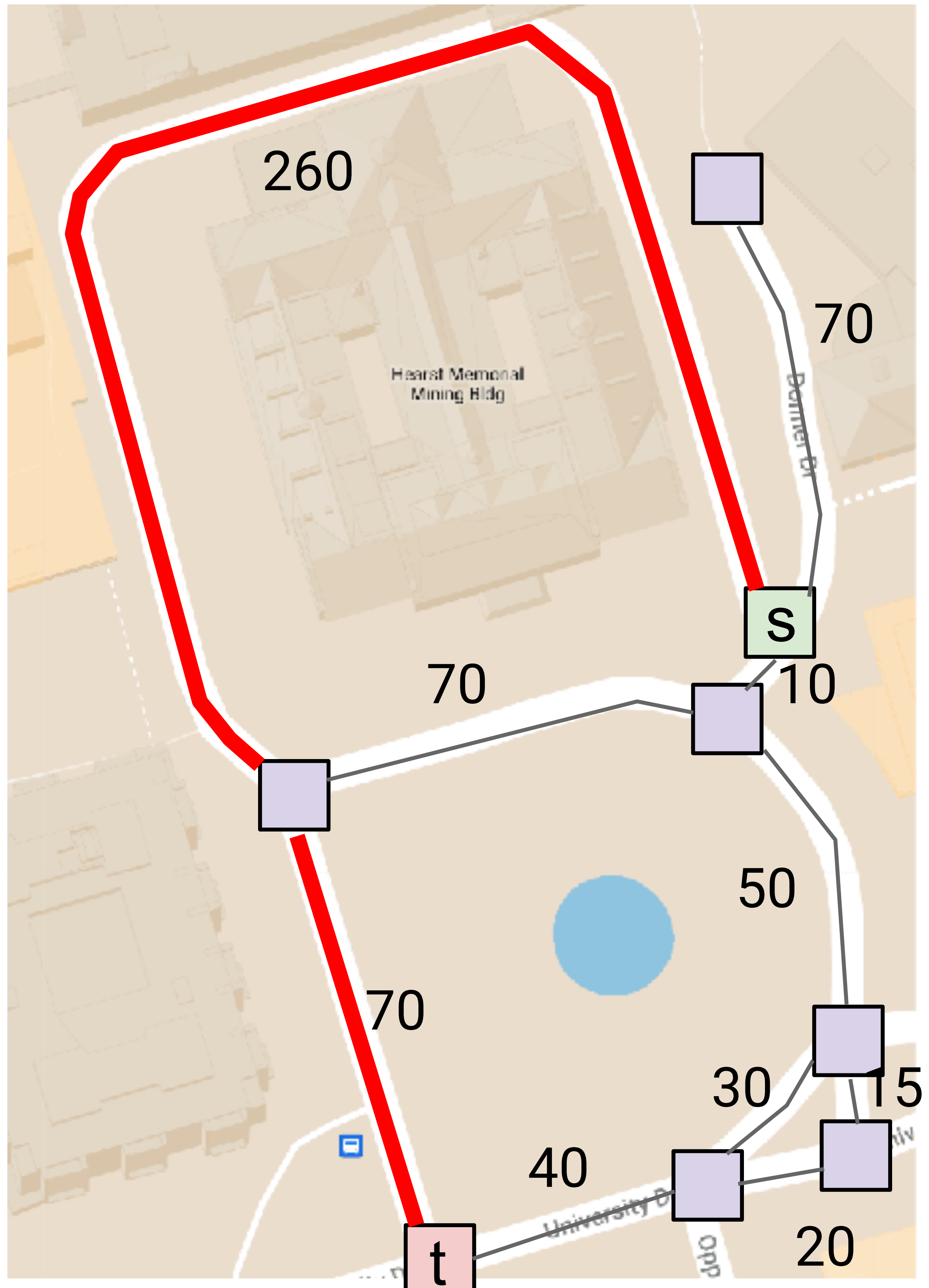
BFS would not be a good choice for a google maps style navigation application.

- The problem: BFS returns path with shortest number of edges, not necessarily the shortest path.
- That’s why we need an edge-weighted graph.

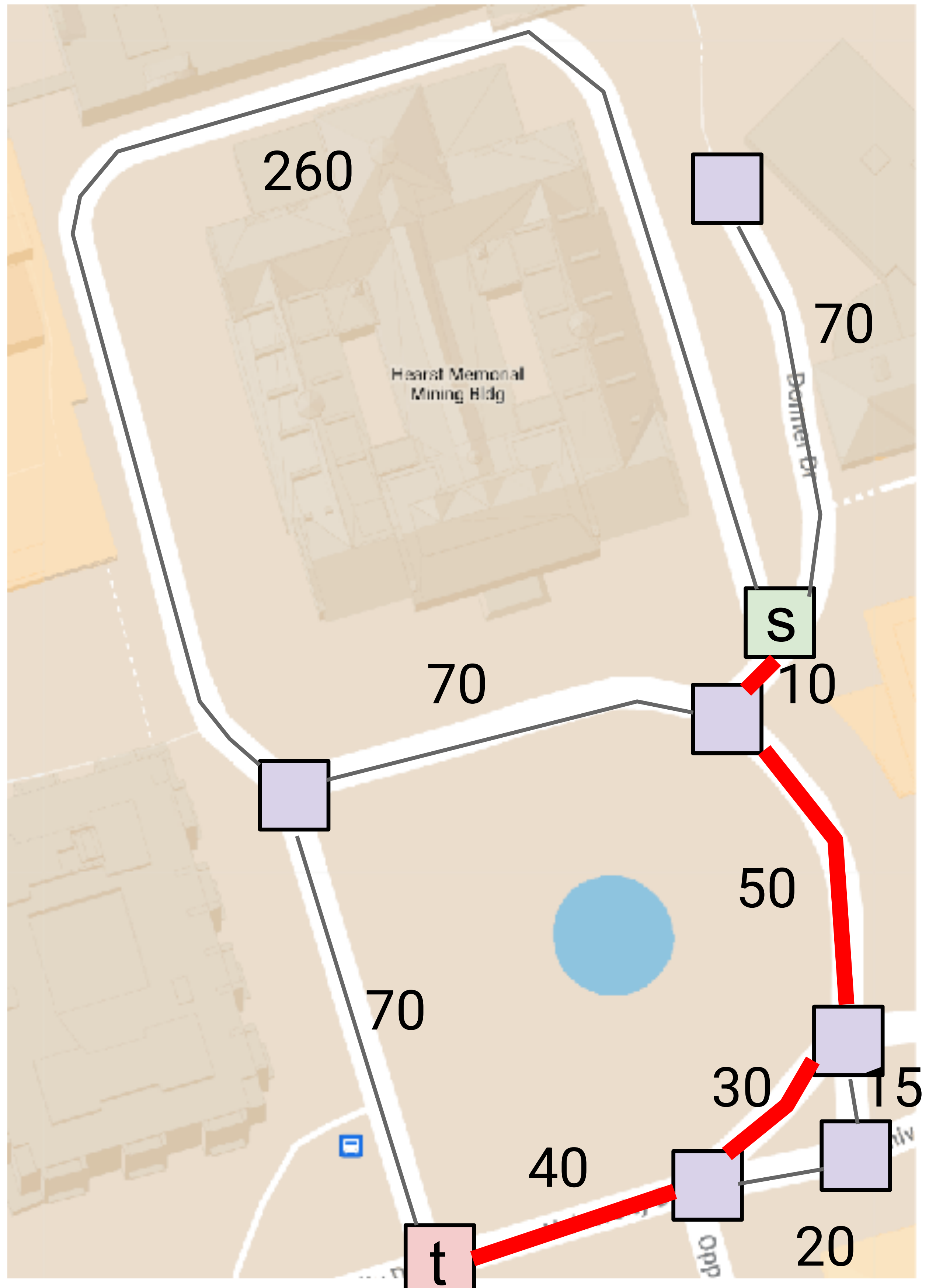


Goal: go from s (green) to t (red)

BFS answer



Correct shortest path



Shortest Path variants

- Single source: from one vertex `s` to every other vertex.
- Single sink: from every vertex to one vertex `t`.
- Source-sink: from one vertex `s` to another vertex `t`.
- All pairs: from every vertex to every other vertex.

- What version is there in Google Maps?

Shortest Paths Assumptions

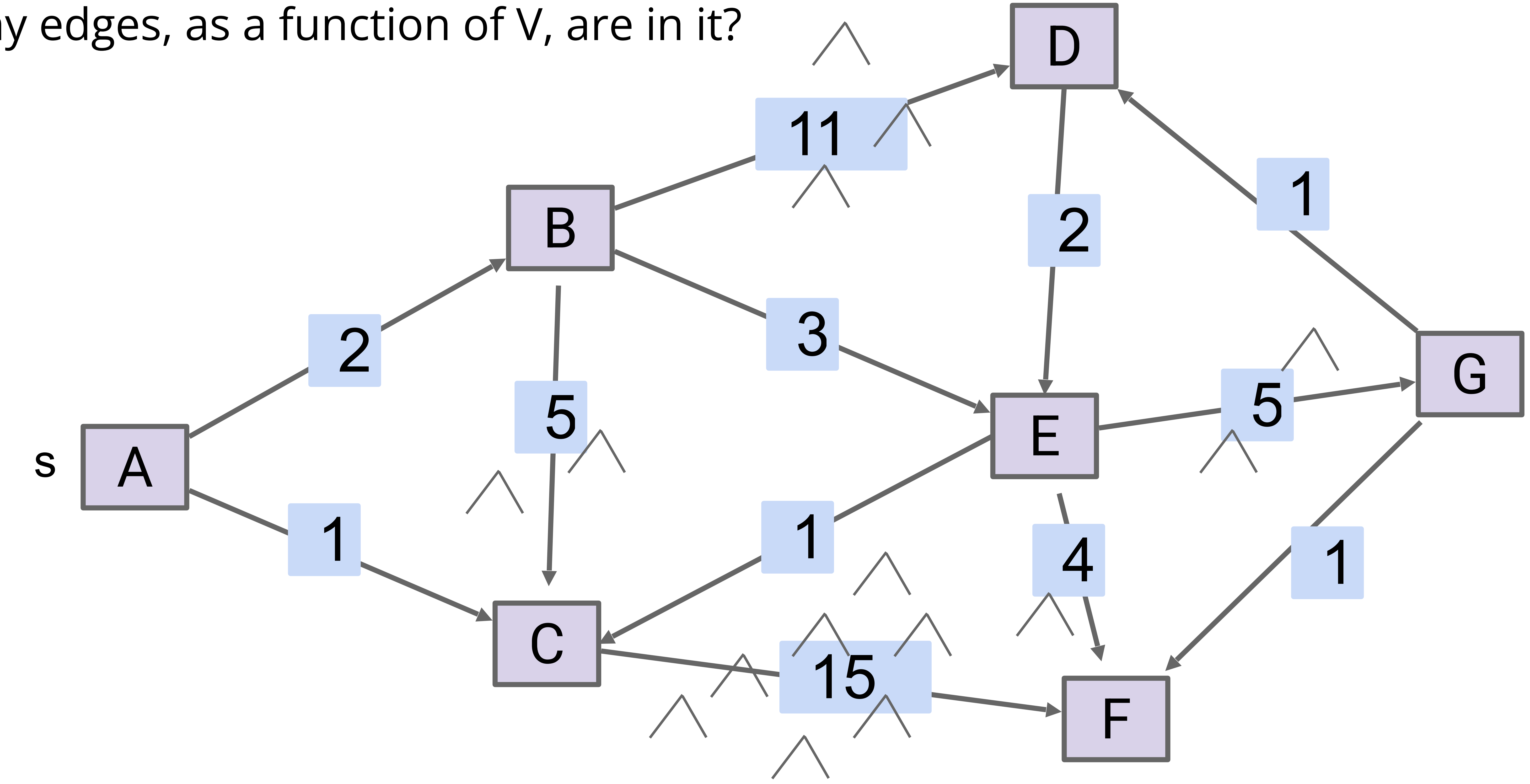
- Not all vertices need to be reachable.
  - We will assume so in this lecture.
- Weights are non-negative.
  - There are algorithms that can handle negative weights.
- Shortest paths are not necessarily unique but they are simple.

Worksheet time!

Find the shortest paths from source vertex s to every other vertex. (Single source shortest path)

What data structure does your path look like?

How many edges, as a function of V, are in it?



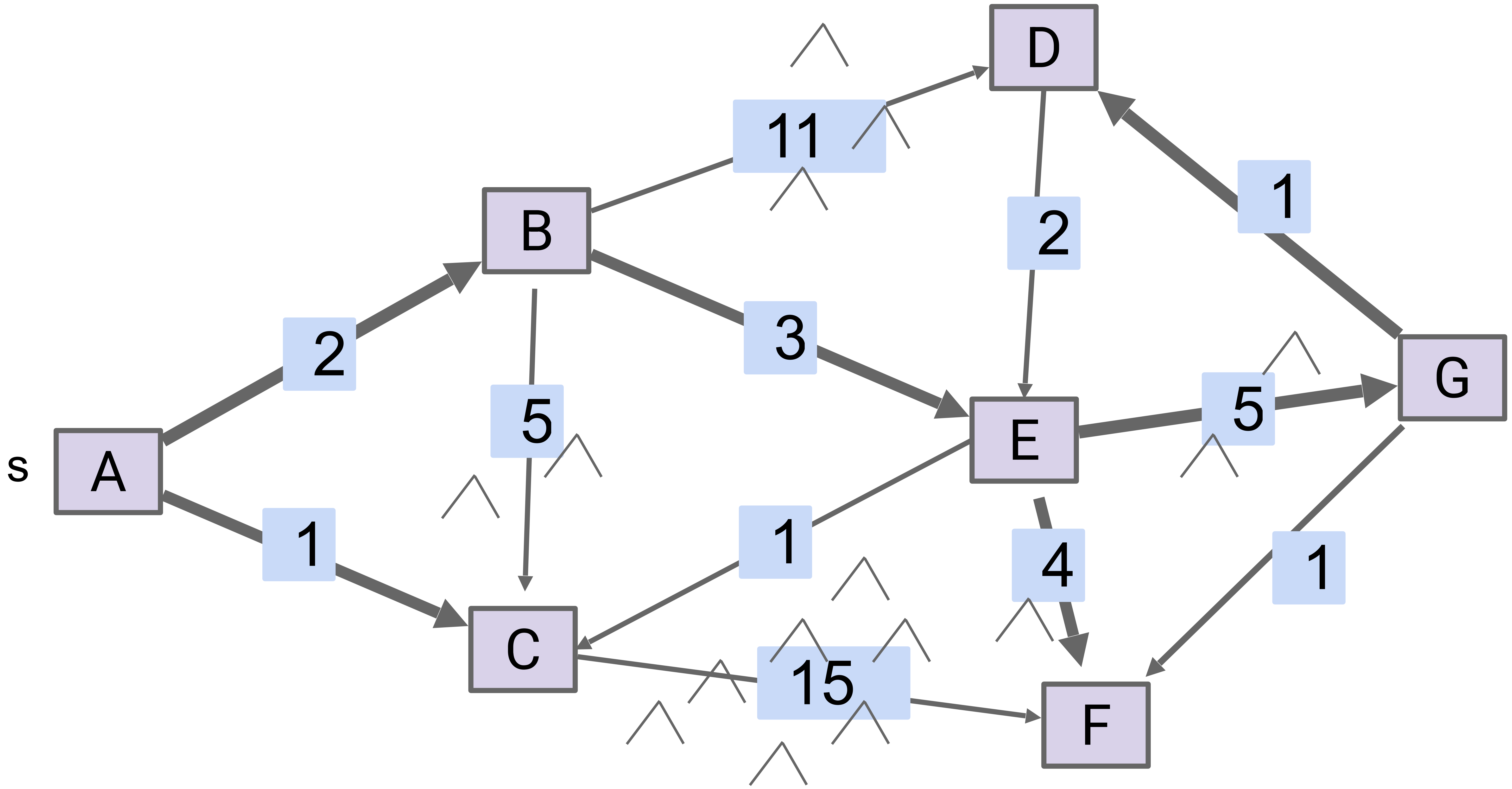
What algorithm did you as a human come up with?

Worksheet answers

Find the shortest paths from source vertex s to every other vertex. (Single source shortest path)

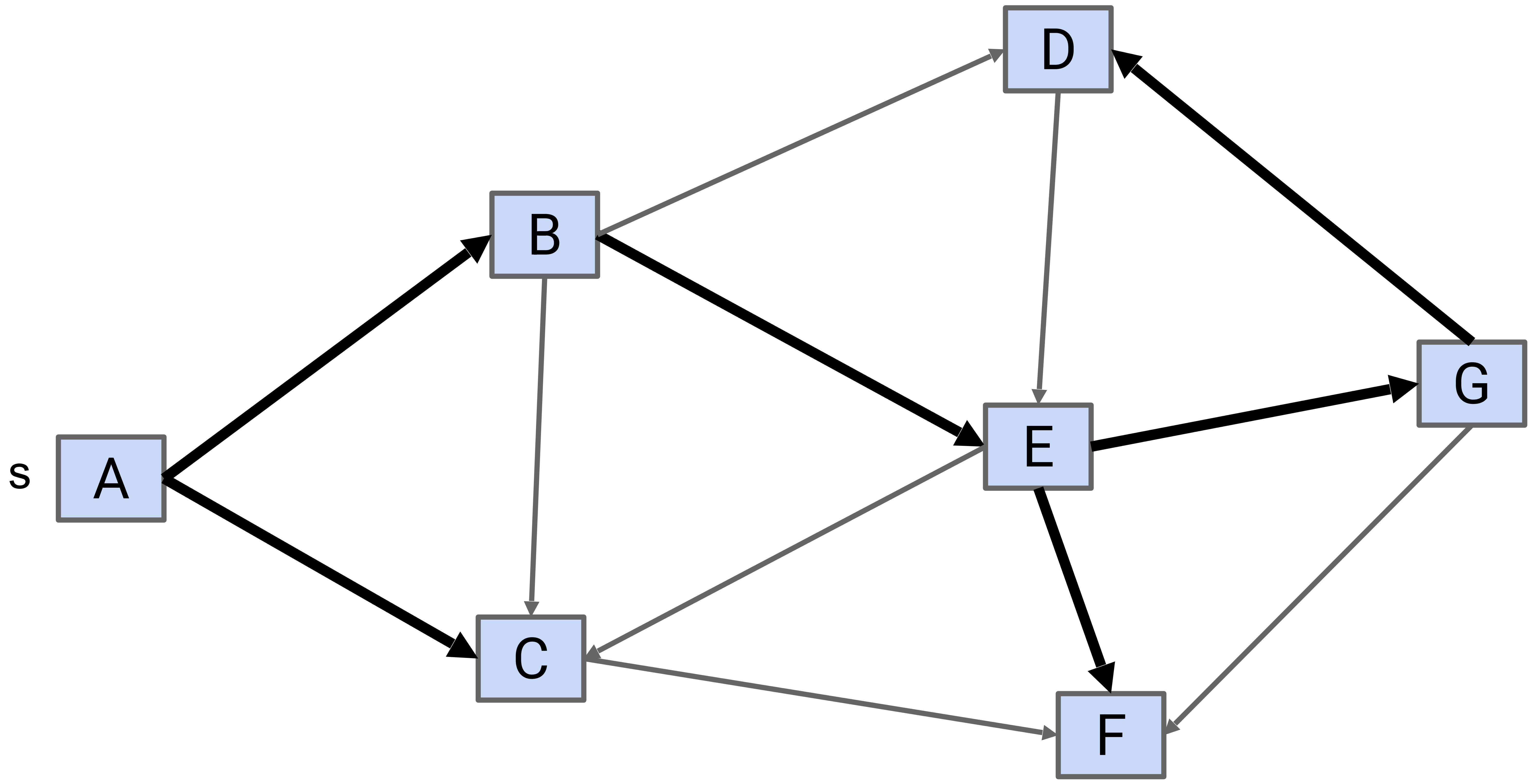
What data structure does your path look like? A tree

How many edges, as a function of V, are in it? E = V-1 (7 vertices, 6 edges)



SPT Edge Count

If G is a connected edge-weighted graph with V vertices and E edges, there are exactly *V-1* edges are in the **Shortest Paths Tree** (SPT) of G, assuming every vertex is reachable.



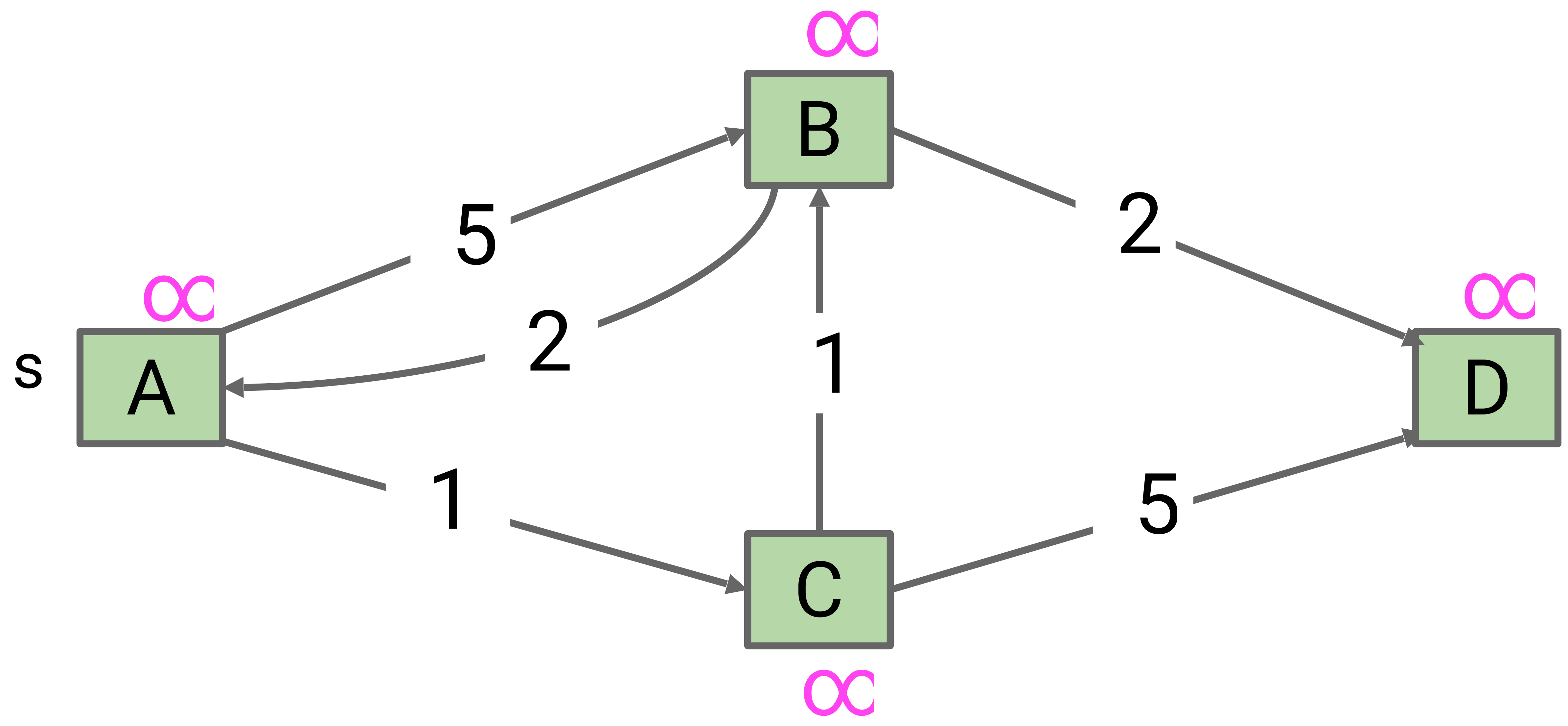
Dijkstra’s Algorithm (bad examples)

Creating an Algorithm

Let's create an algorithm for finding the shortest paths.

Will start with a bad algorithm and then successively improve it.

- Algorithm begins in state below. All vertices unmarked. All distances infinite. No edges in the SPT.



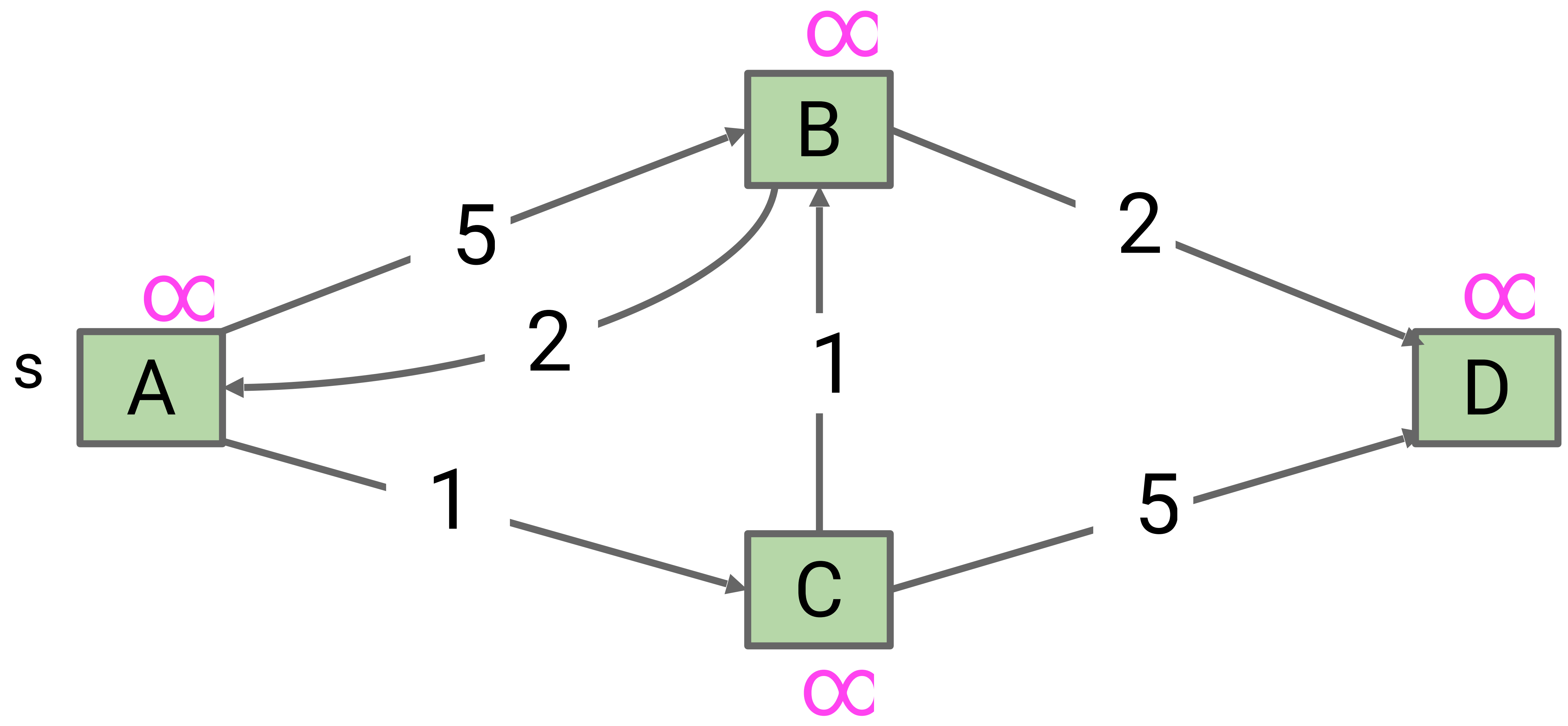
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

  Remove a vertex from the fringe and mark it.

  For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



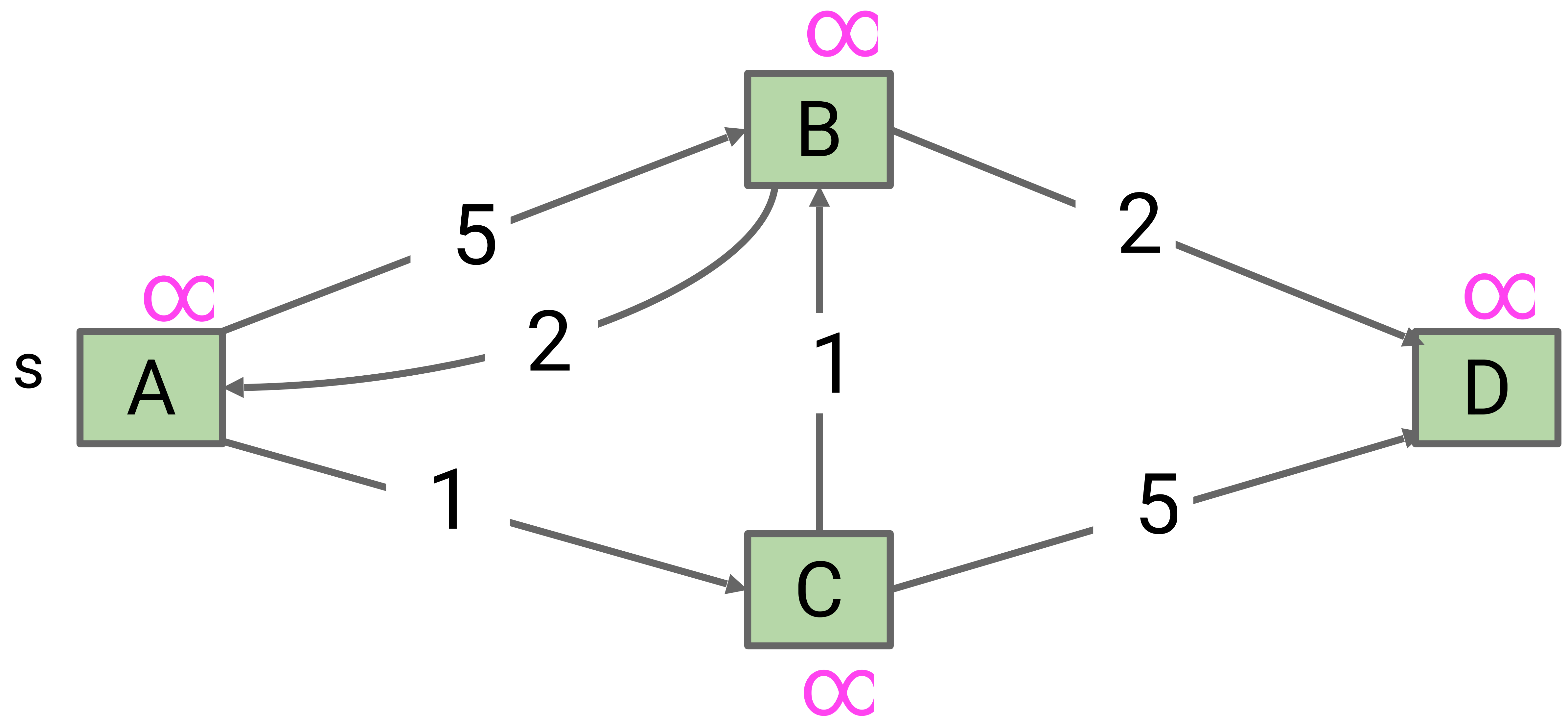
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [A]

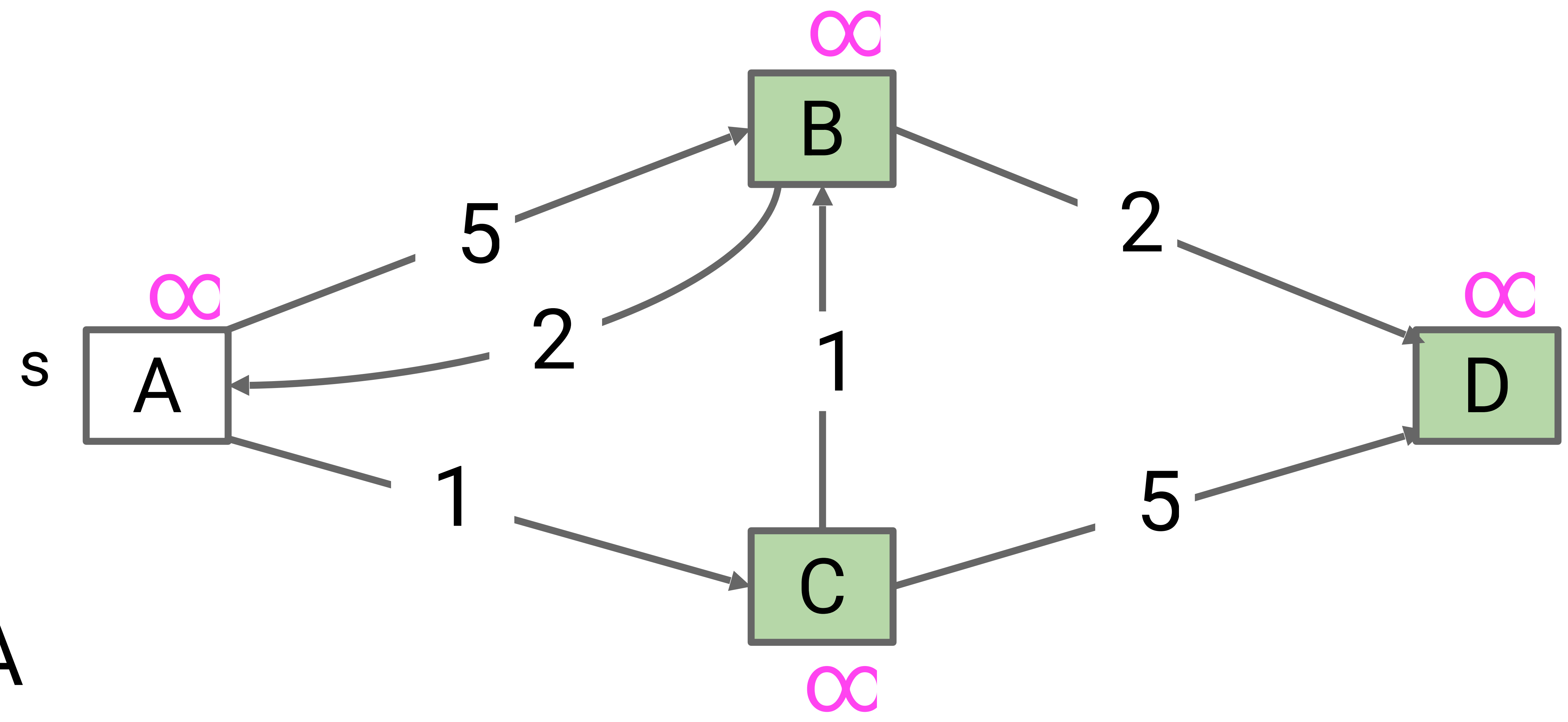
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~]
Removed vertex: A

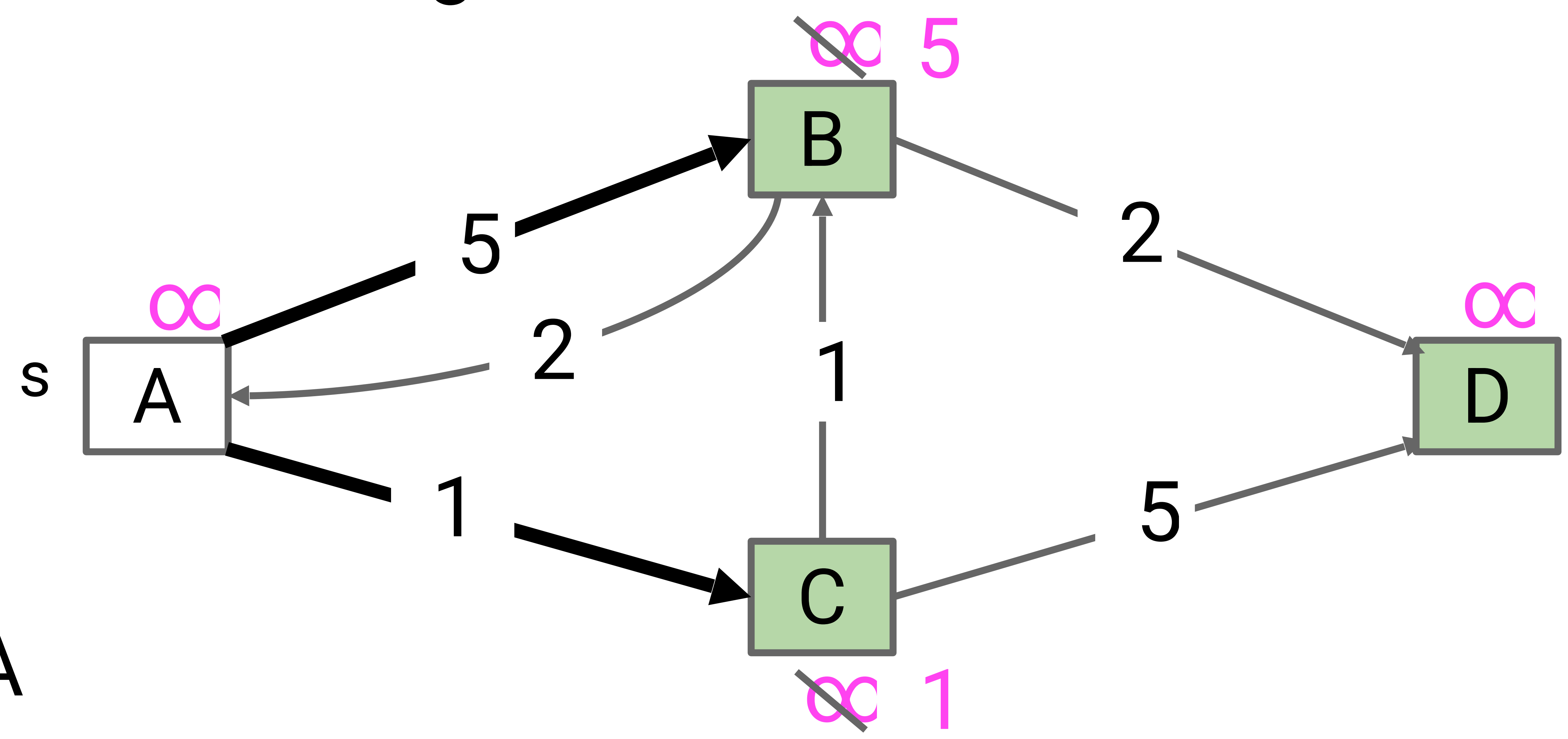
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A,~~ B, C]
Removed vertex: A

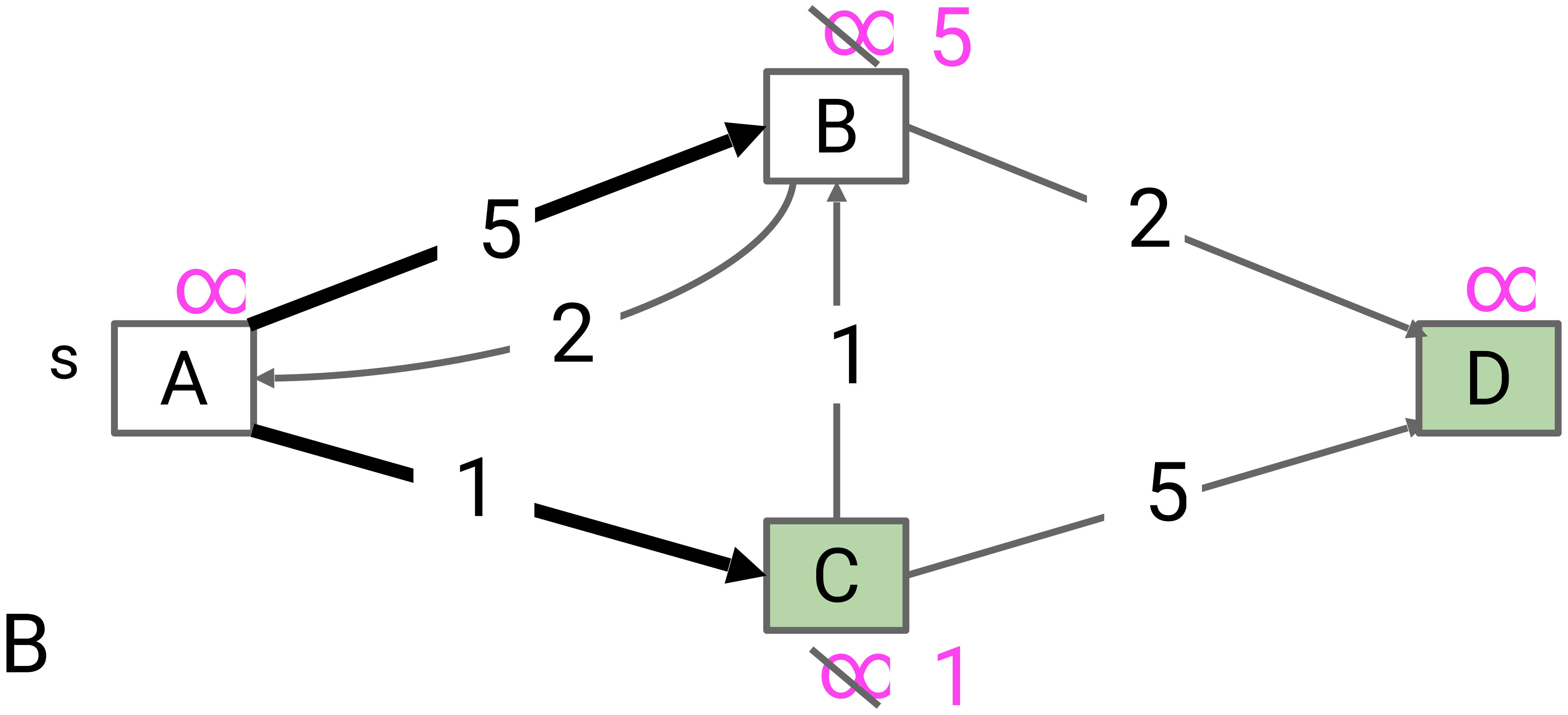
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, C]
Removed vertex: B

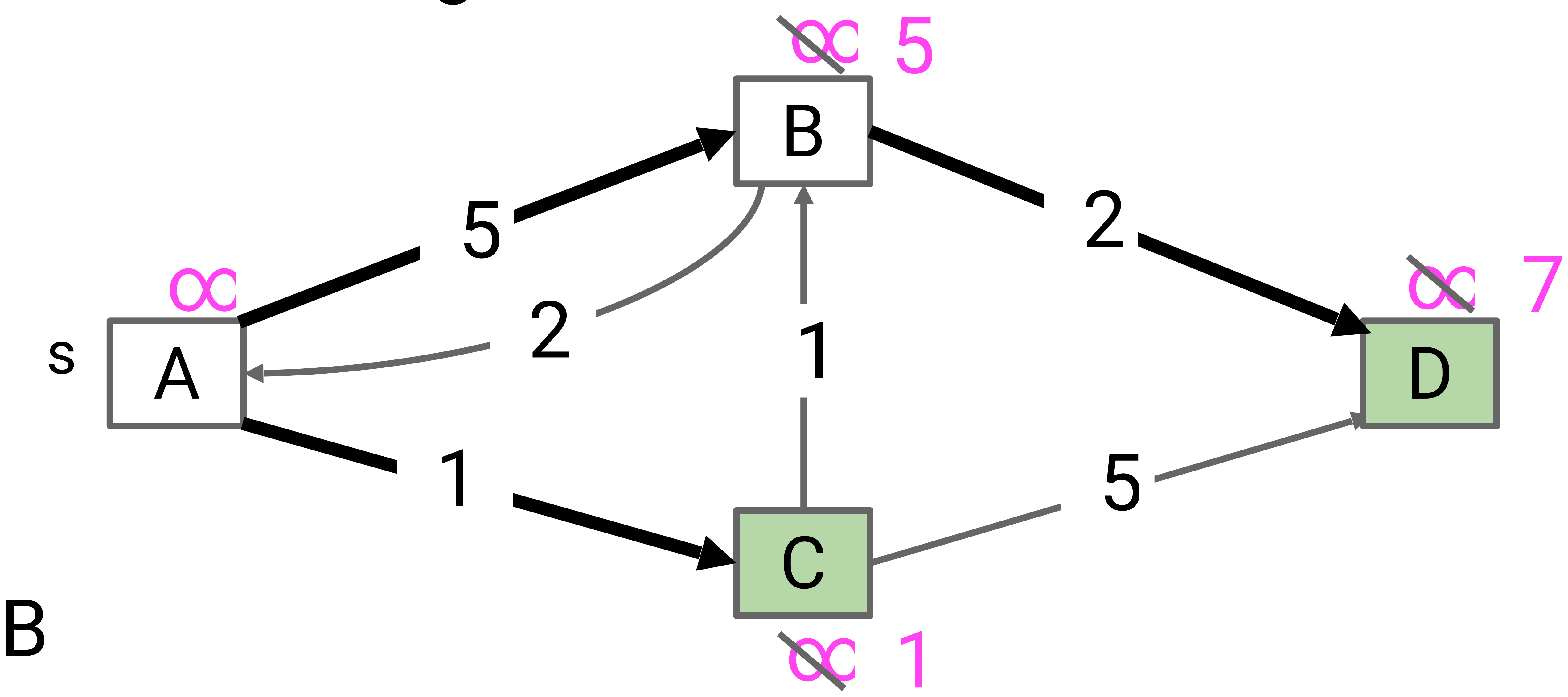
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

    Remove a vertex from the fringe and mark it.

    For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, C, D]
Removed vertex: B

The edge B→A is not added to SPT, because A is already part of the SPT.

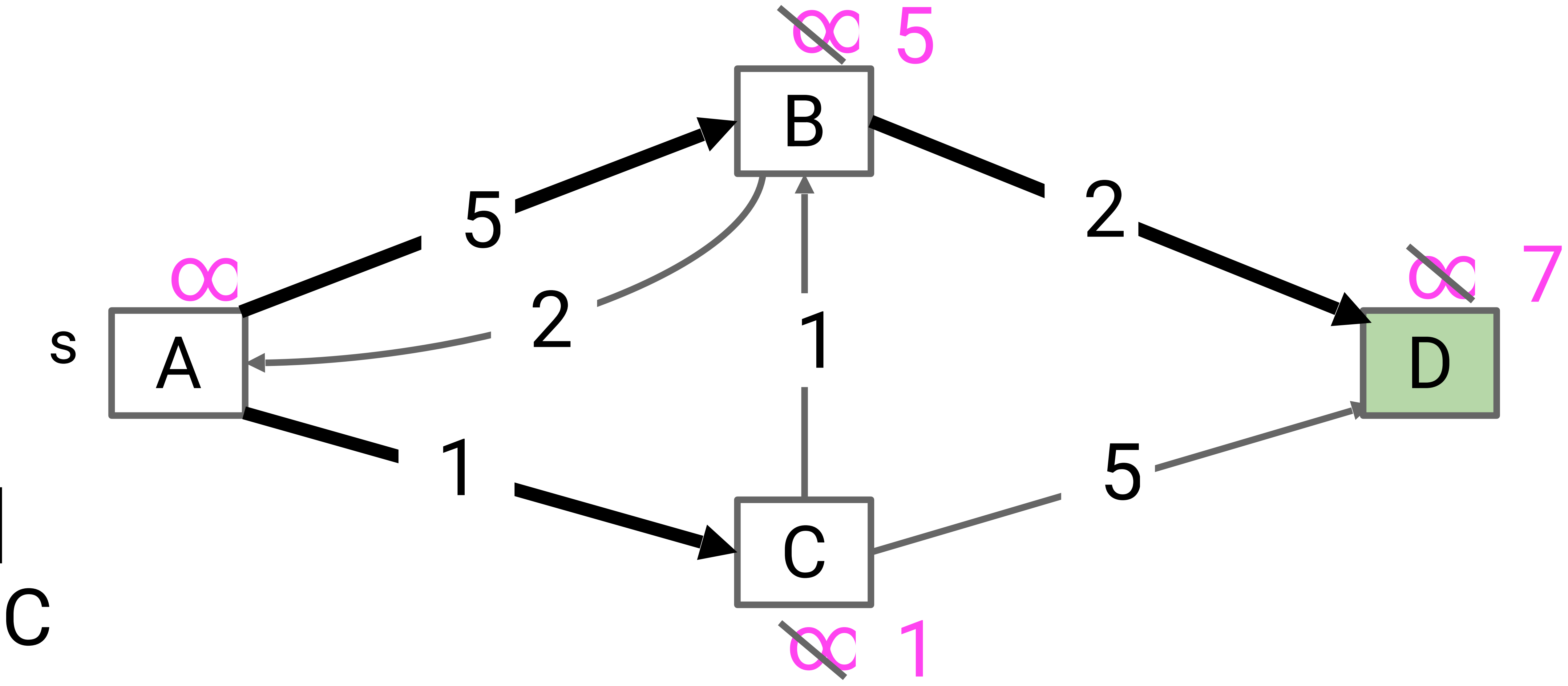
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, ~~C~~, D]
Removed vertex: C

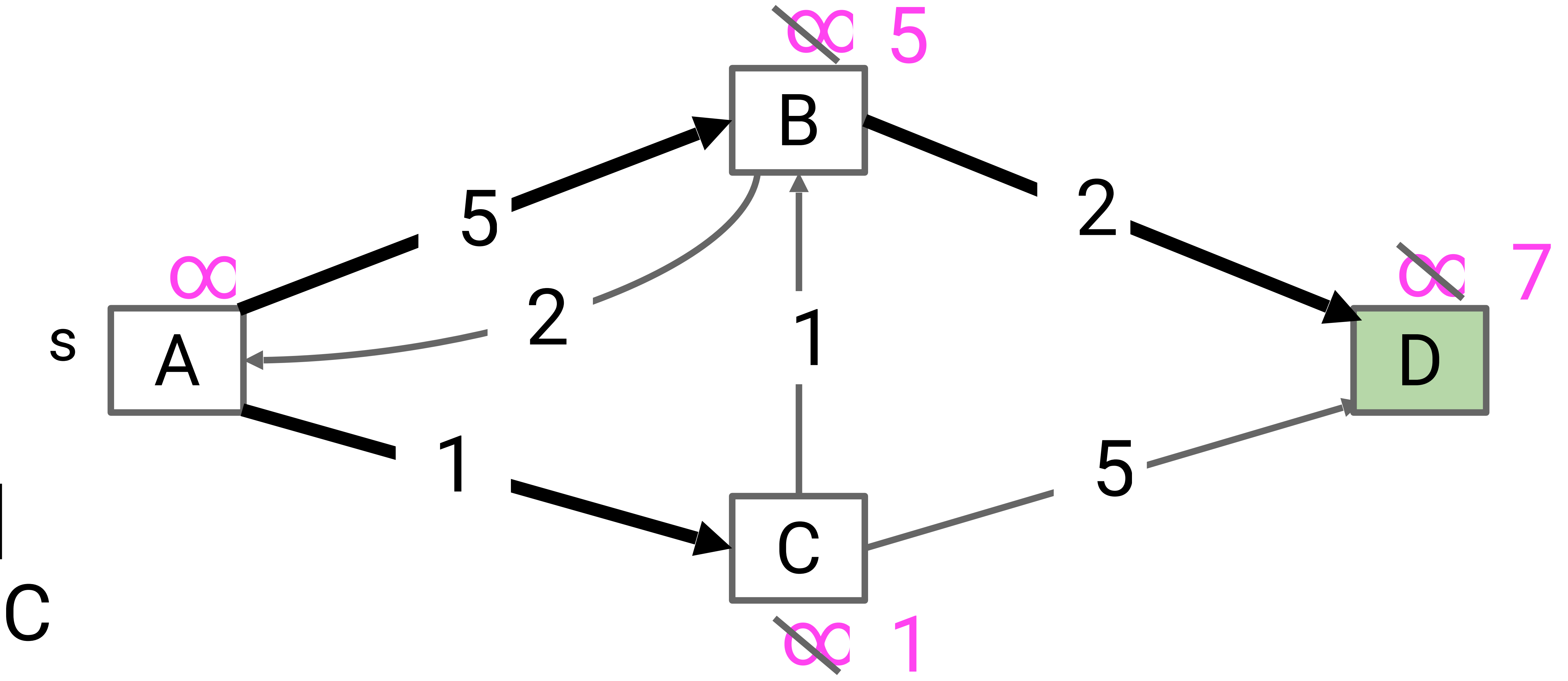
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, ~~C~~, D]
Removed vertex: C

Nothing happens.

C→B not added, B already in SPT.

C→D not added, D already in SPT.

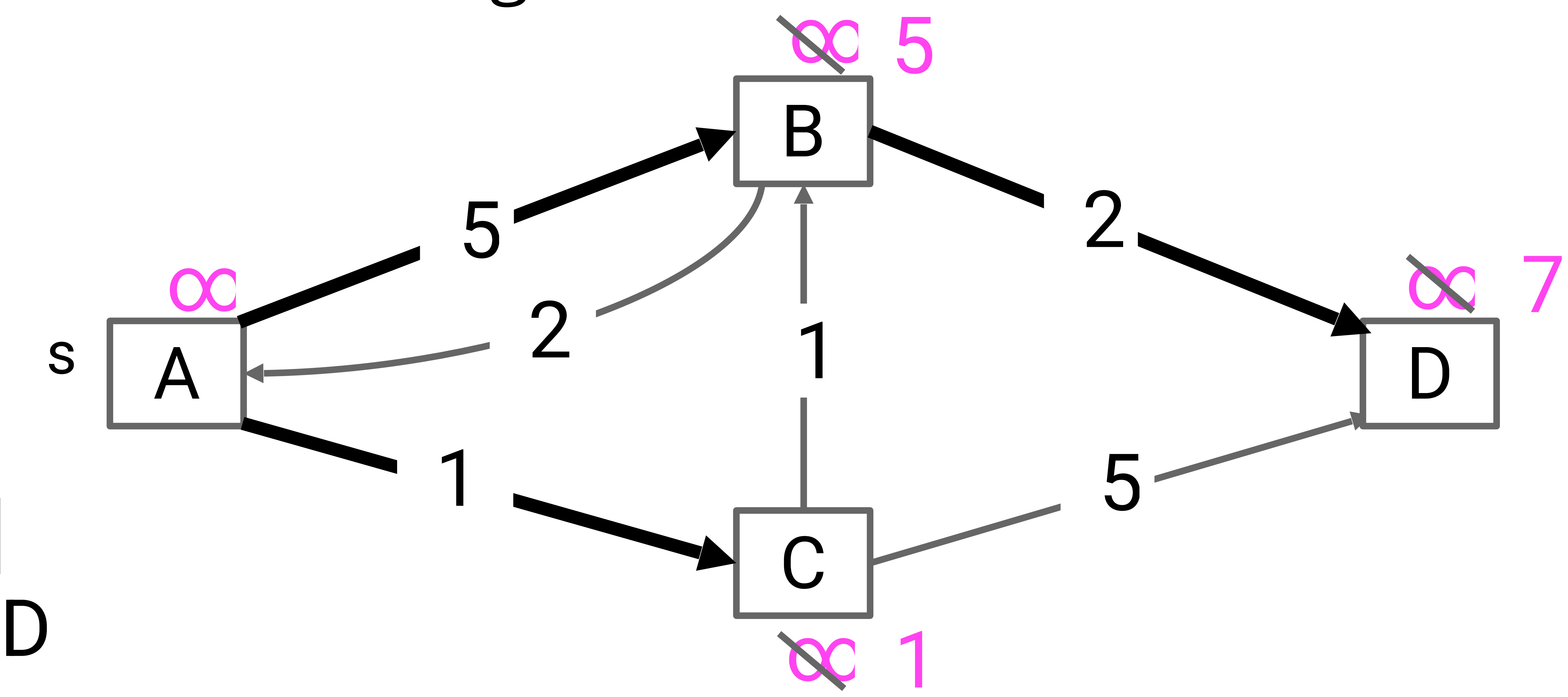
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, ~~C~~, ~~D~~]
Removed vertex: D

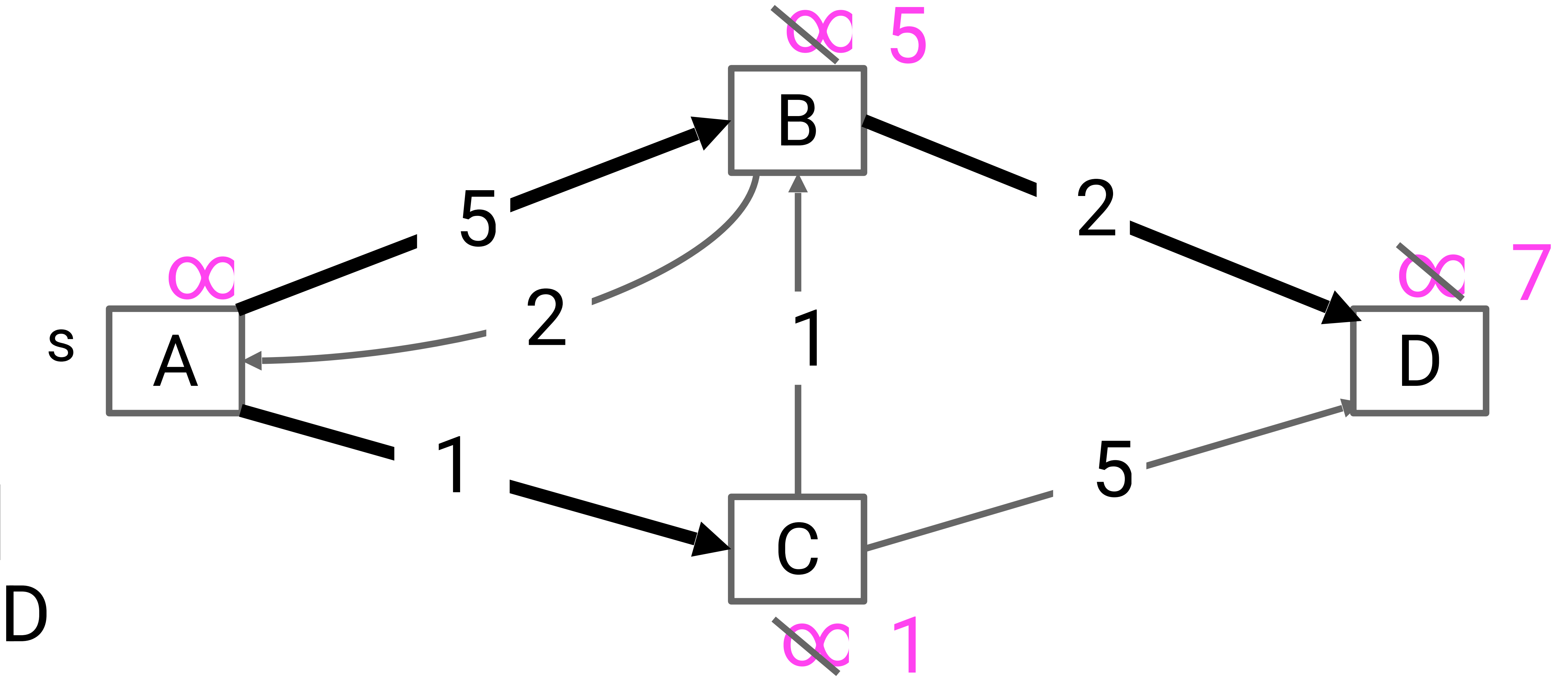
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A~~, ~~B~~, ~~C~~, ~~D~~]
Removed vertex: D

Nothing happens.

D has no neighbors (there are no edges going out of D).

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

    Remove a vertex from the fringe and mark it.

    For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

Takeaways:

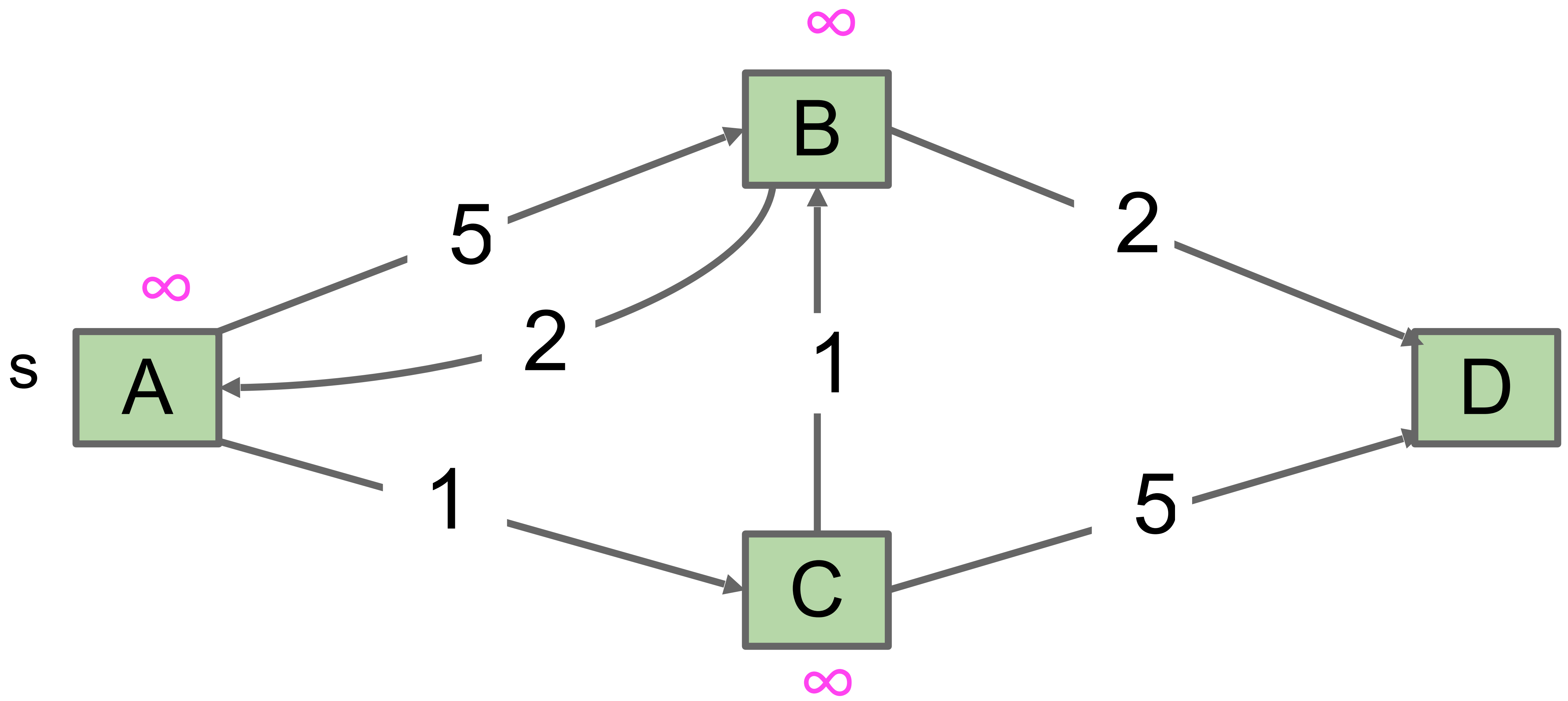
Algorithm #1 (BFS) visits:
every node 1 edge away,
then every node 2 edges away,
then every node 3 edges away, etc.

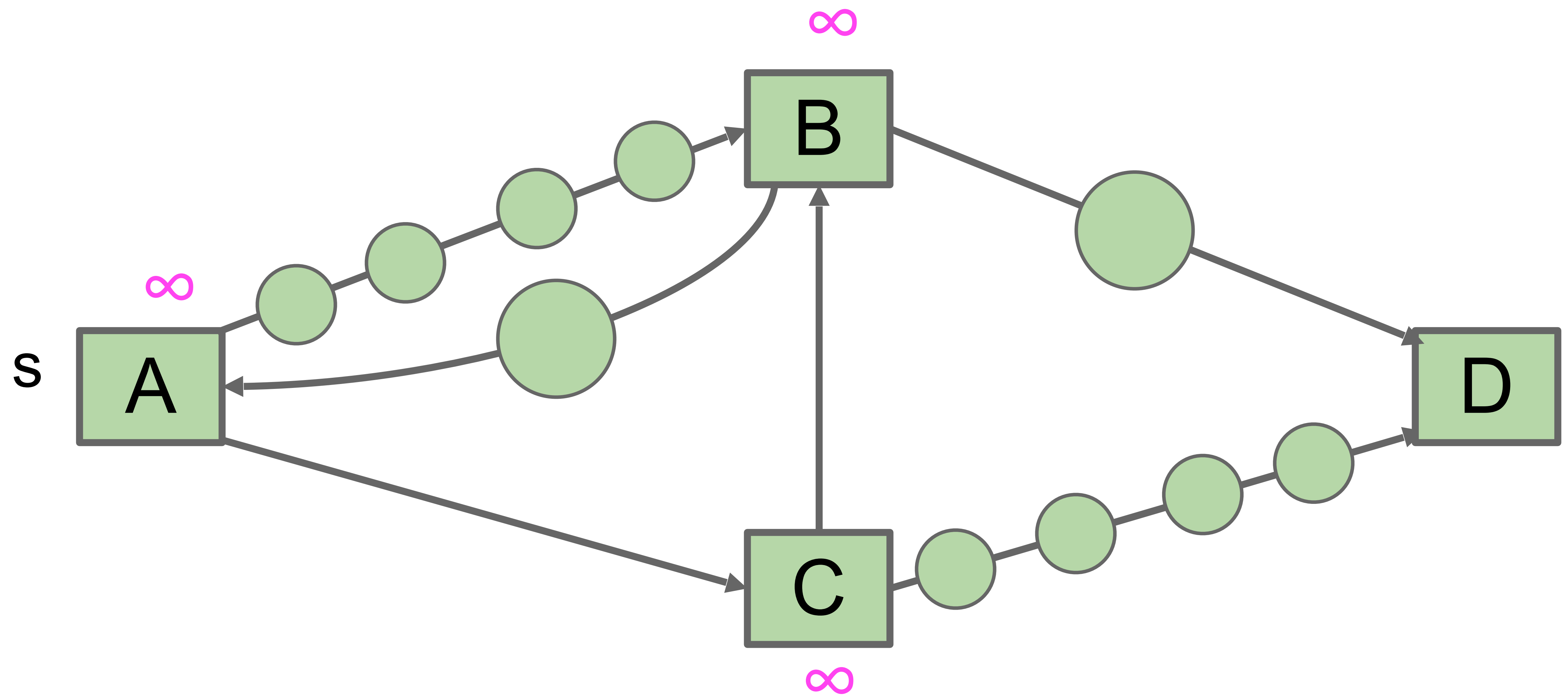
- This algorithm would work if all our edges were the same length.

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



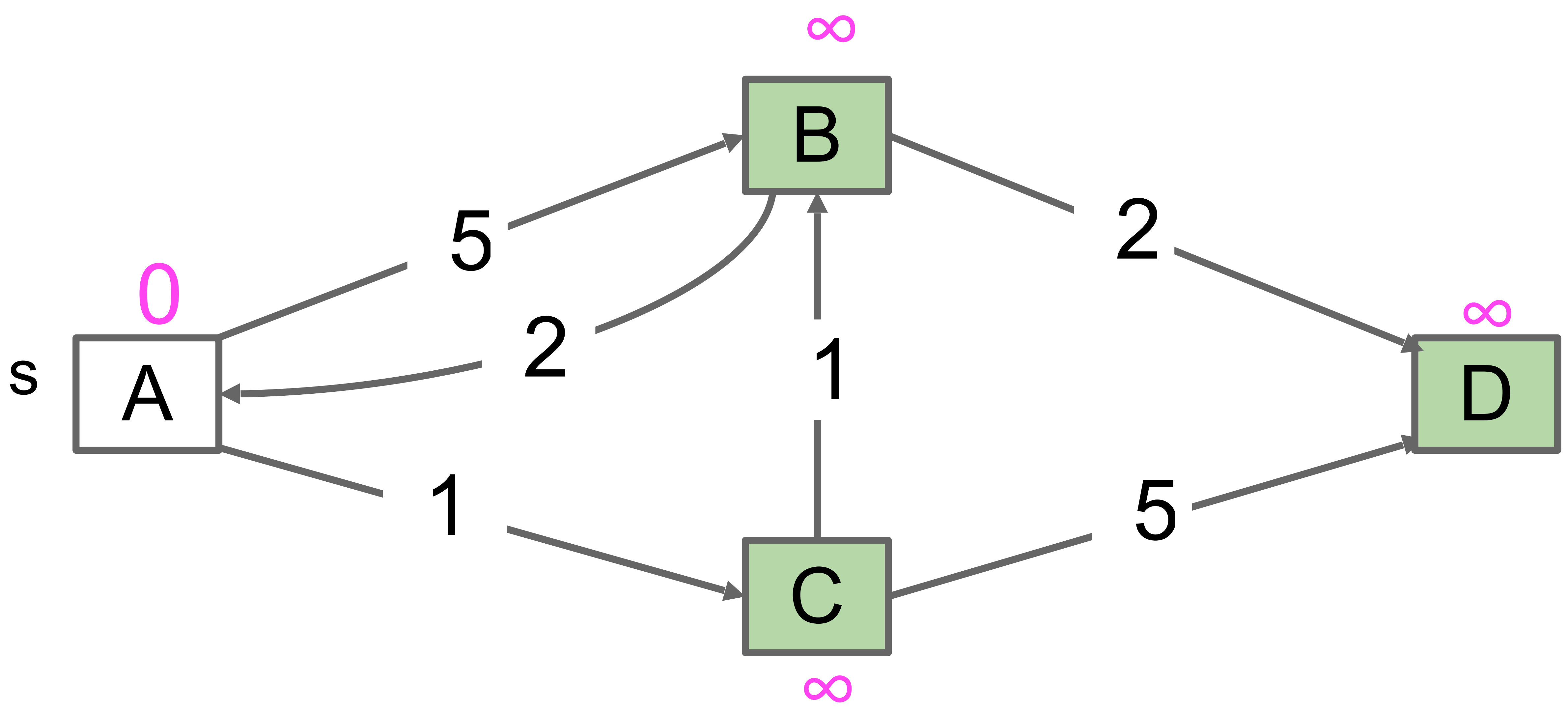


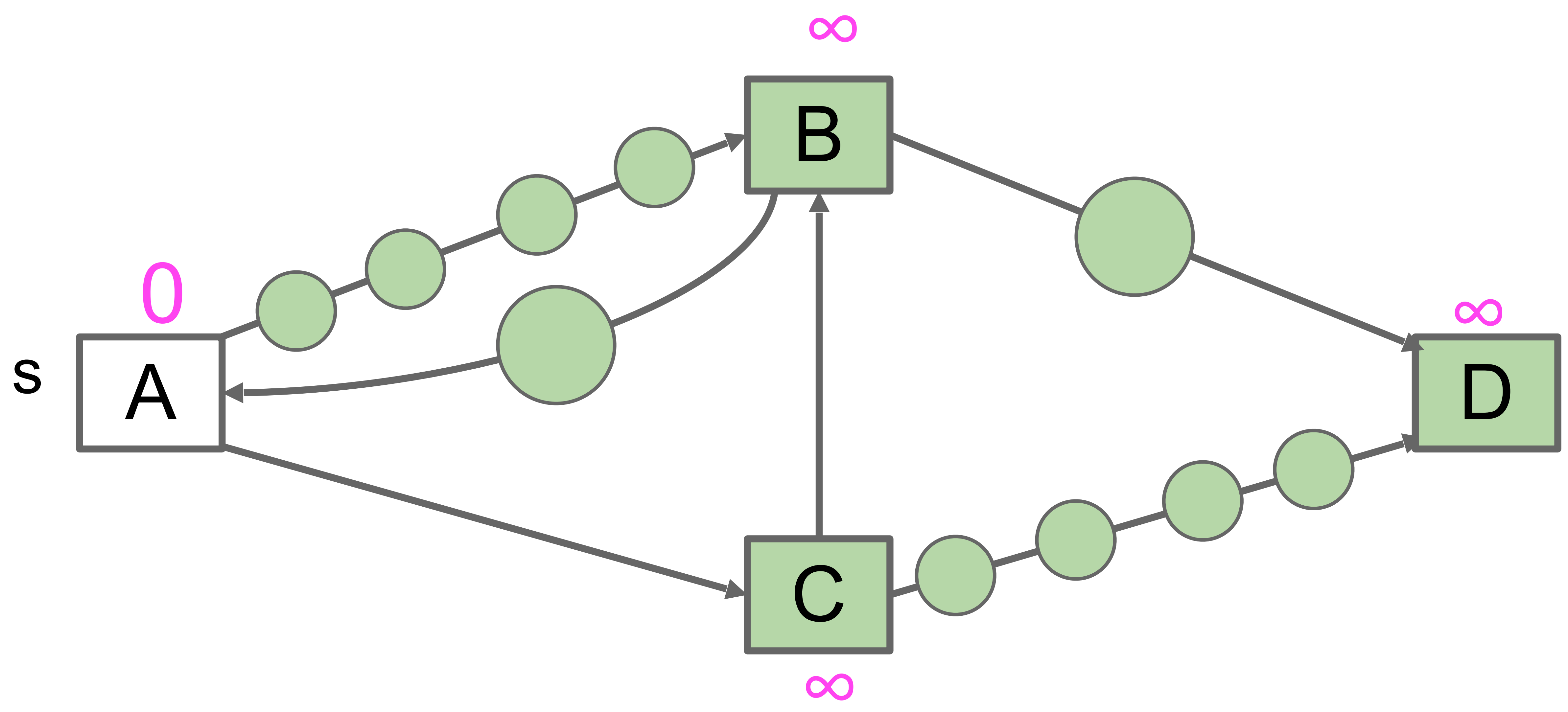
Order of visited nodes:

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



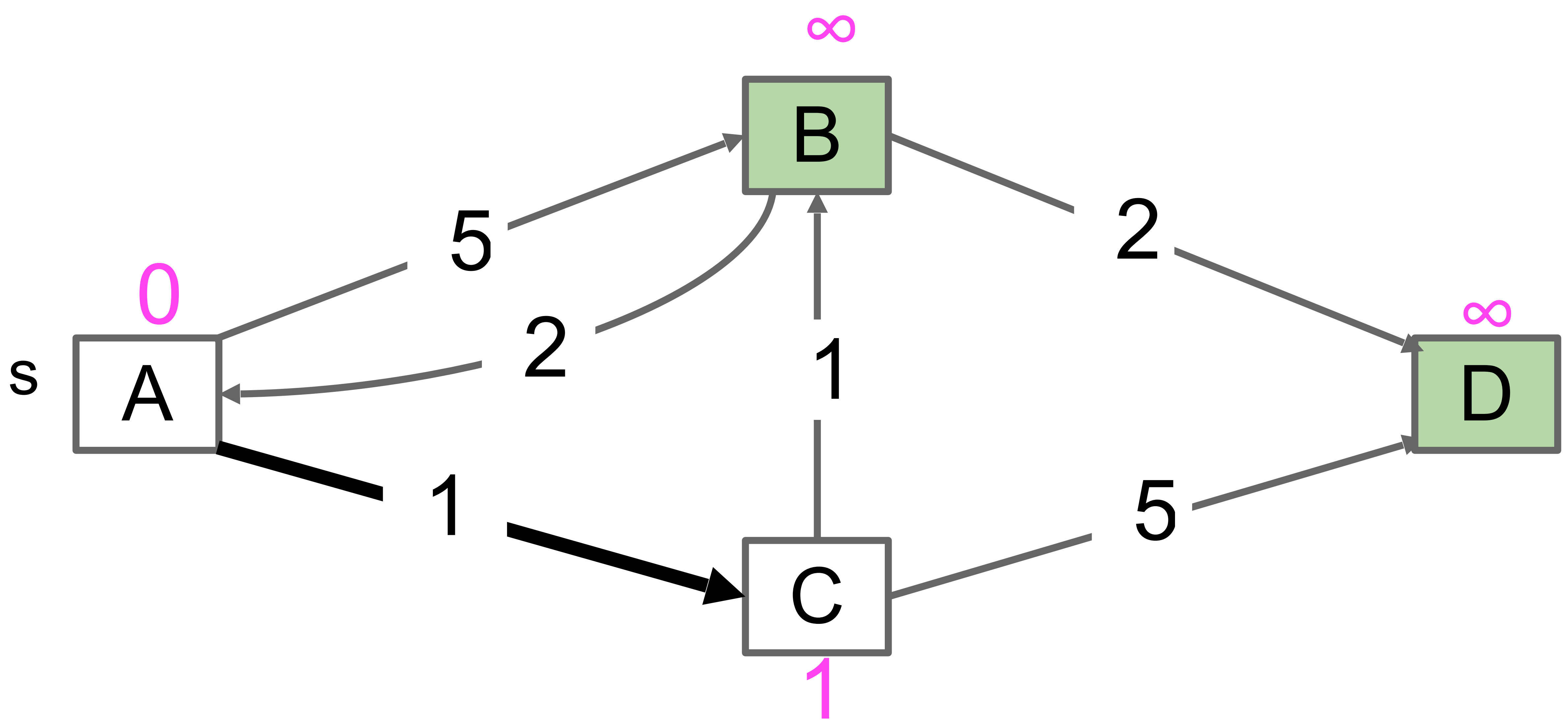


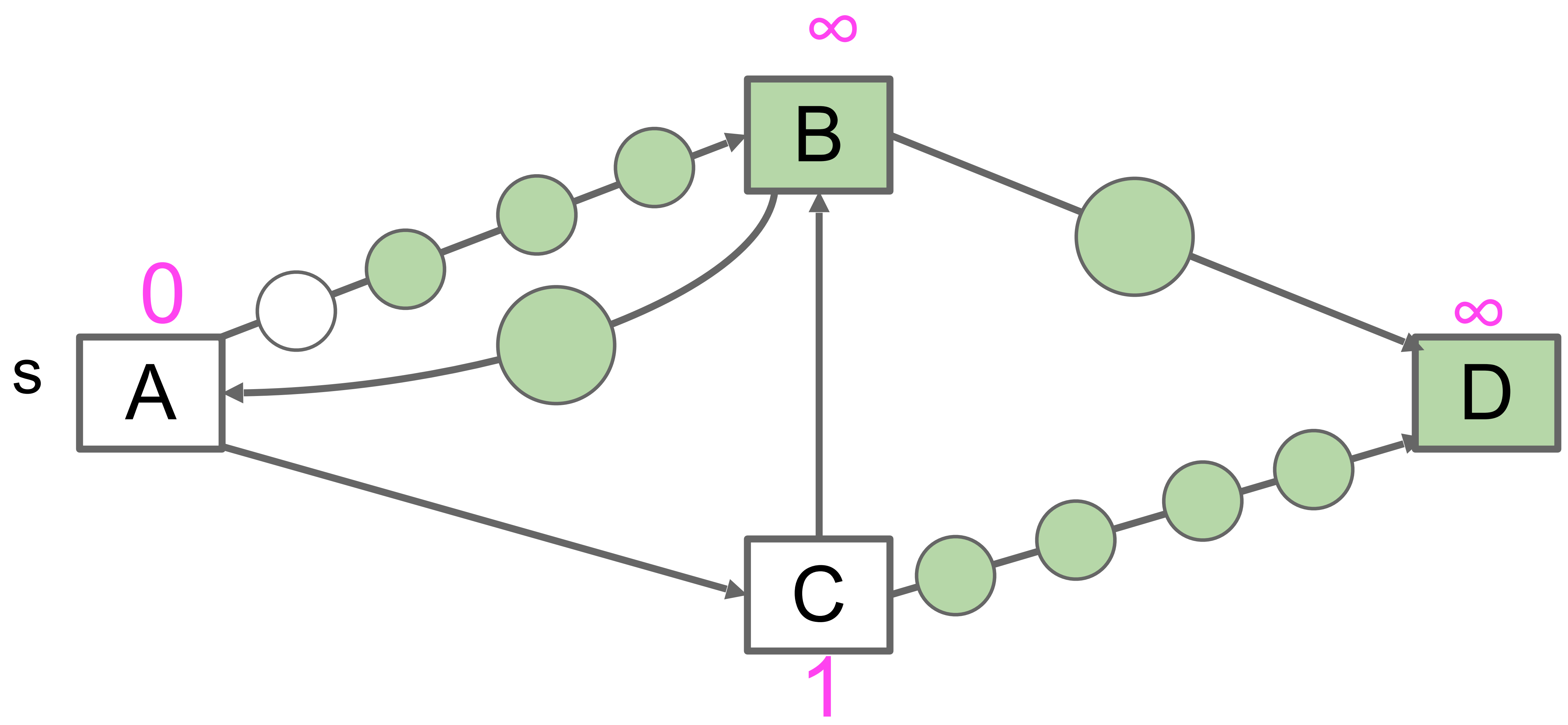
Order of visited nodes: A

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



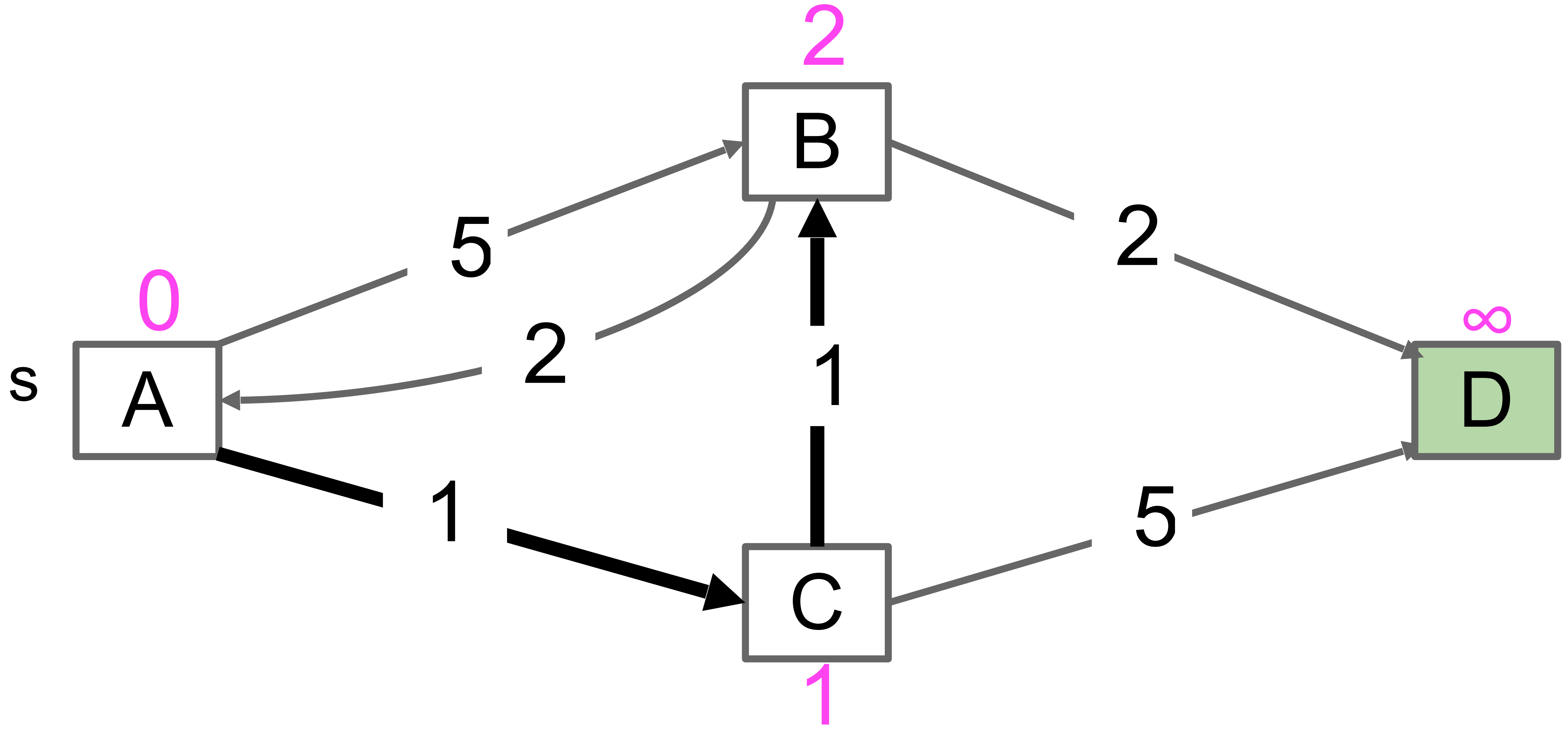


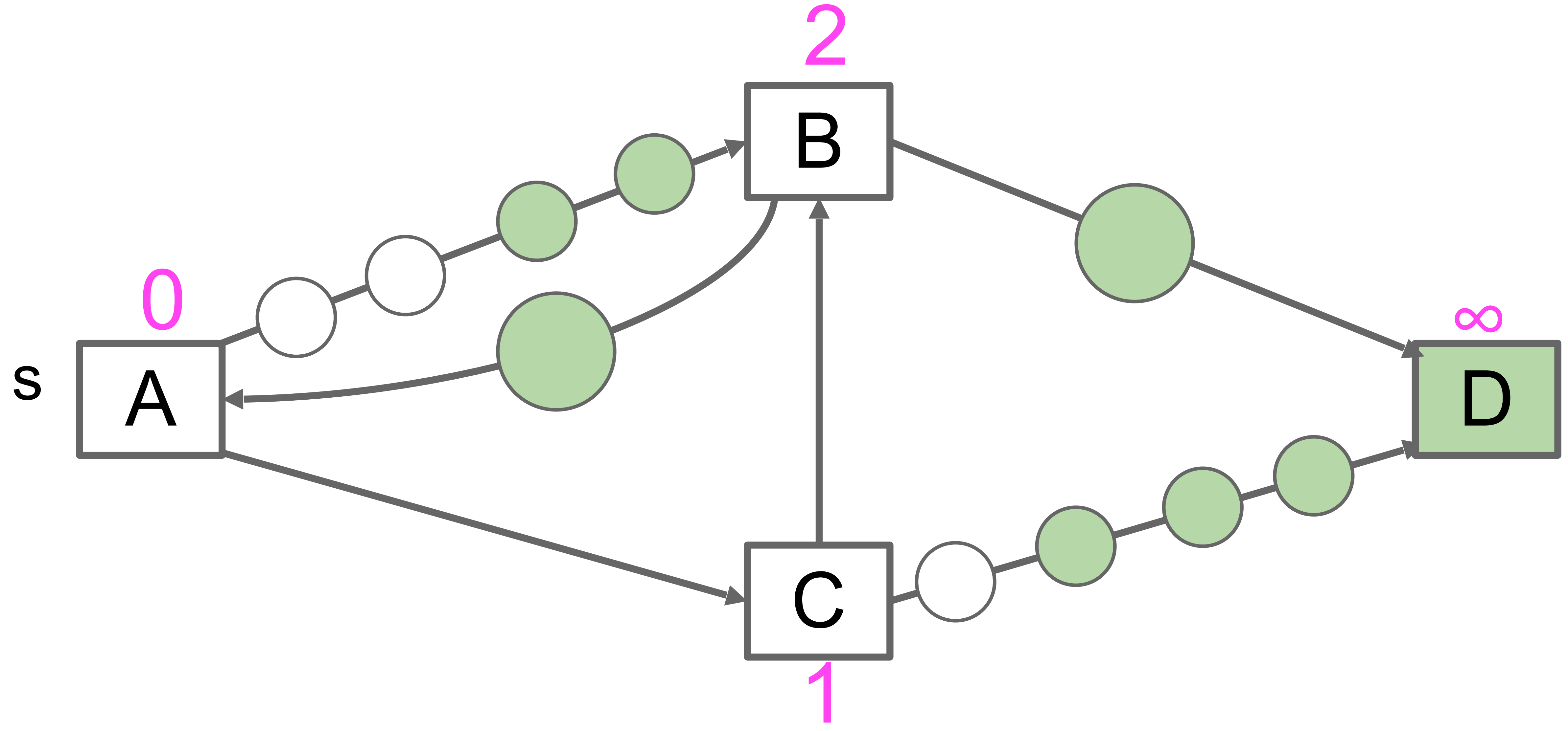
Order of visited nodes: AC

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



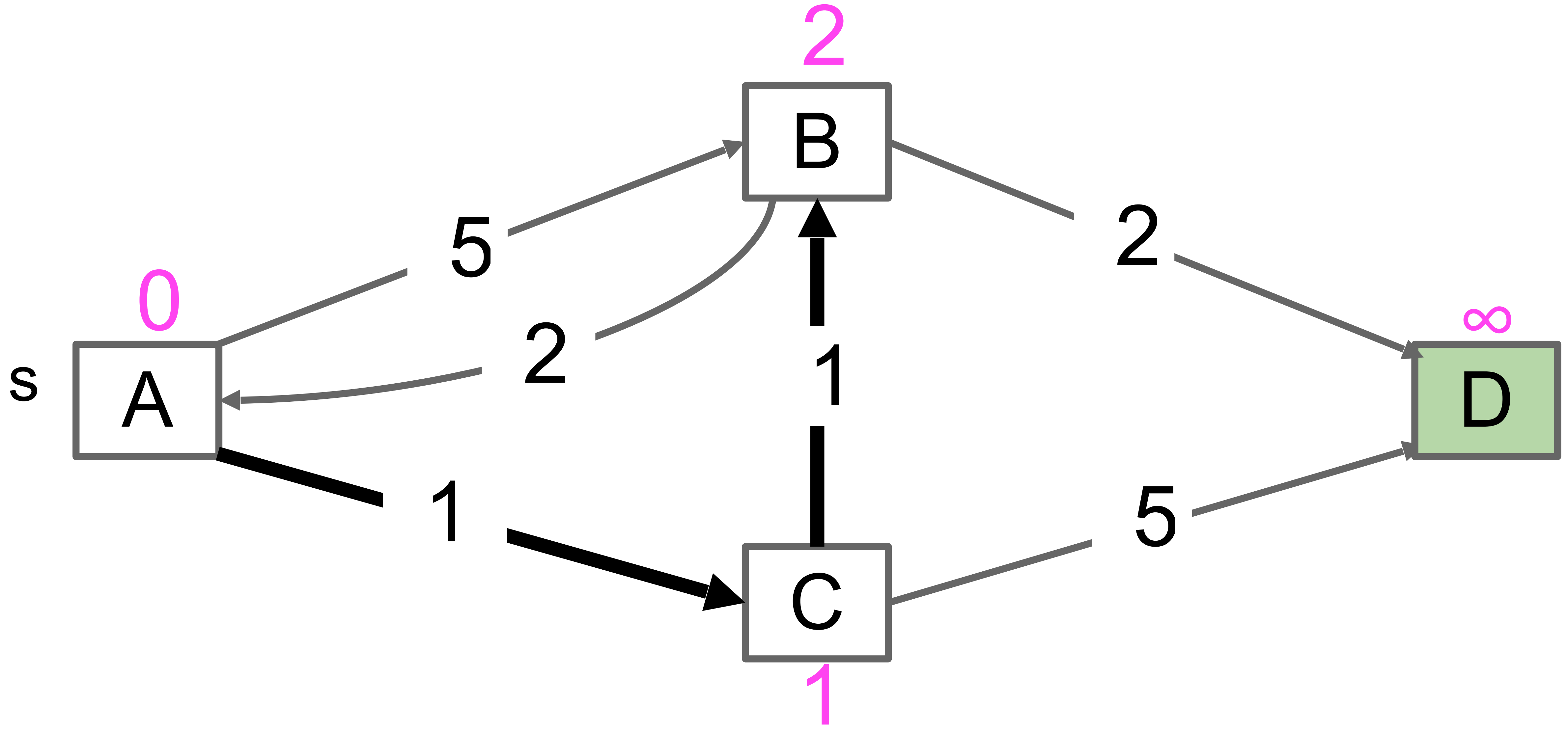


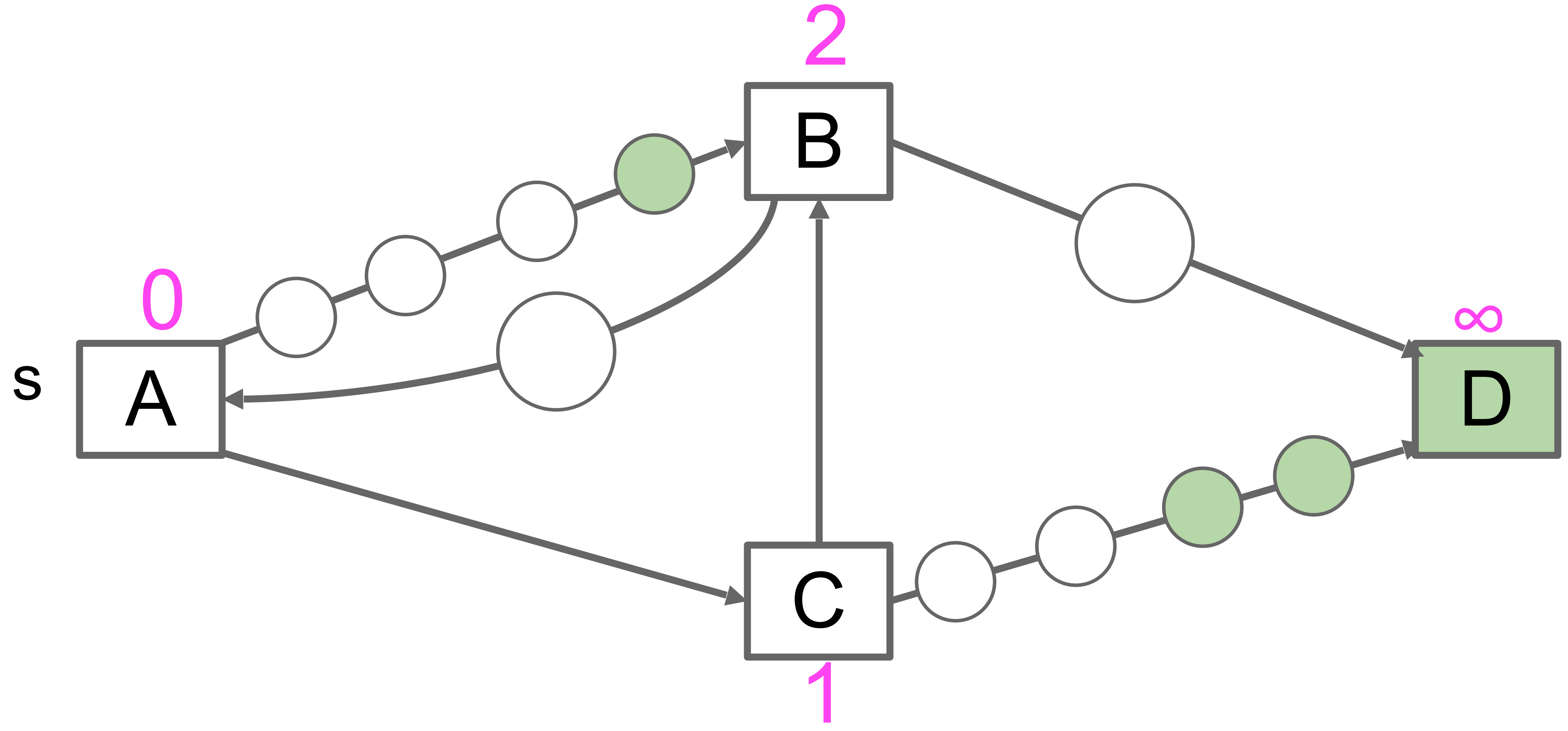
Order of visited nodes: ACB

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



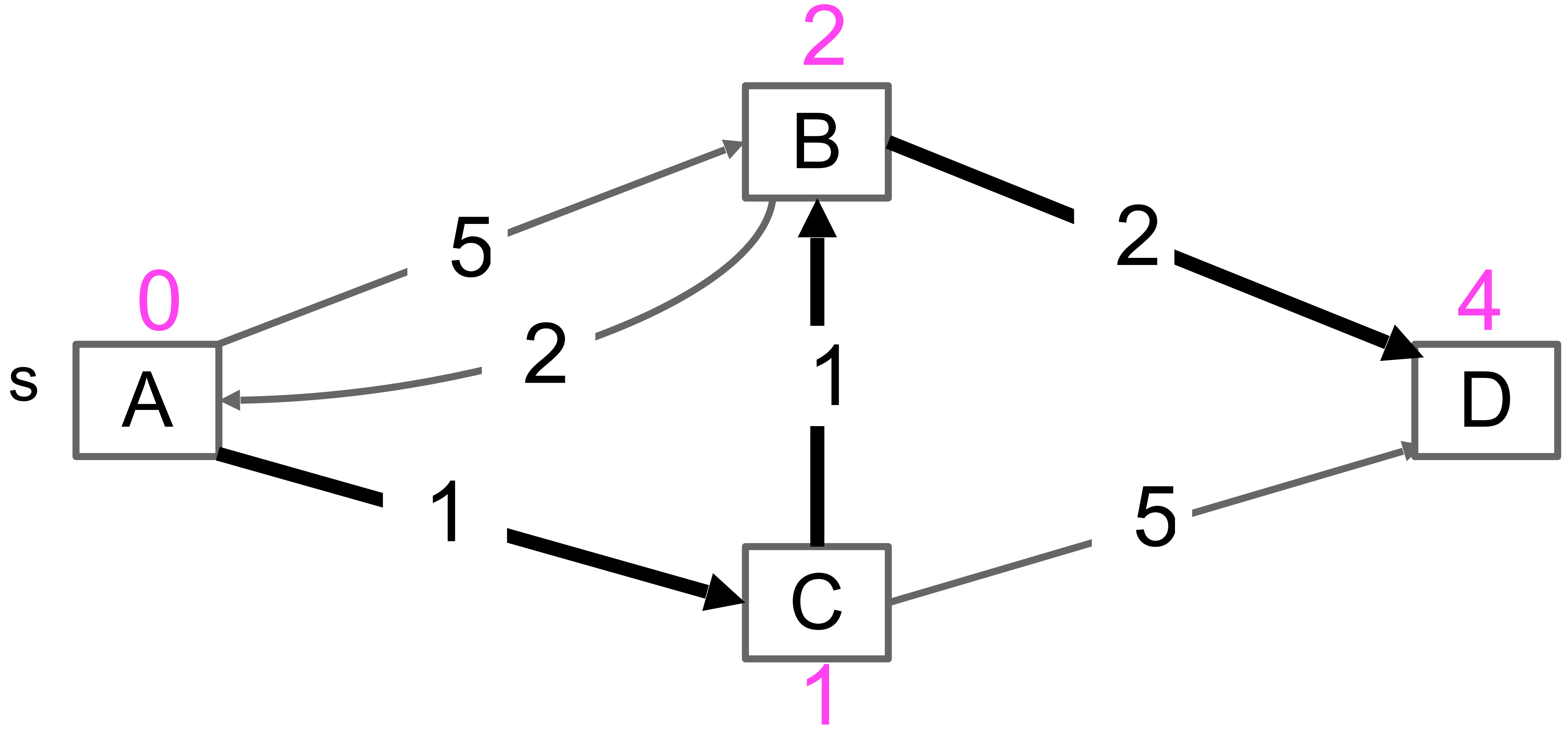


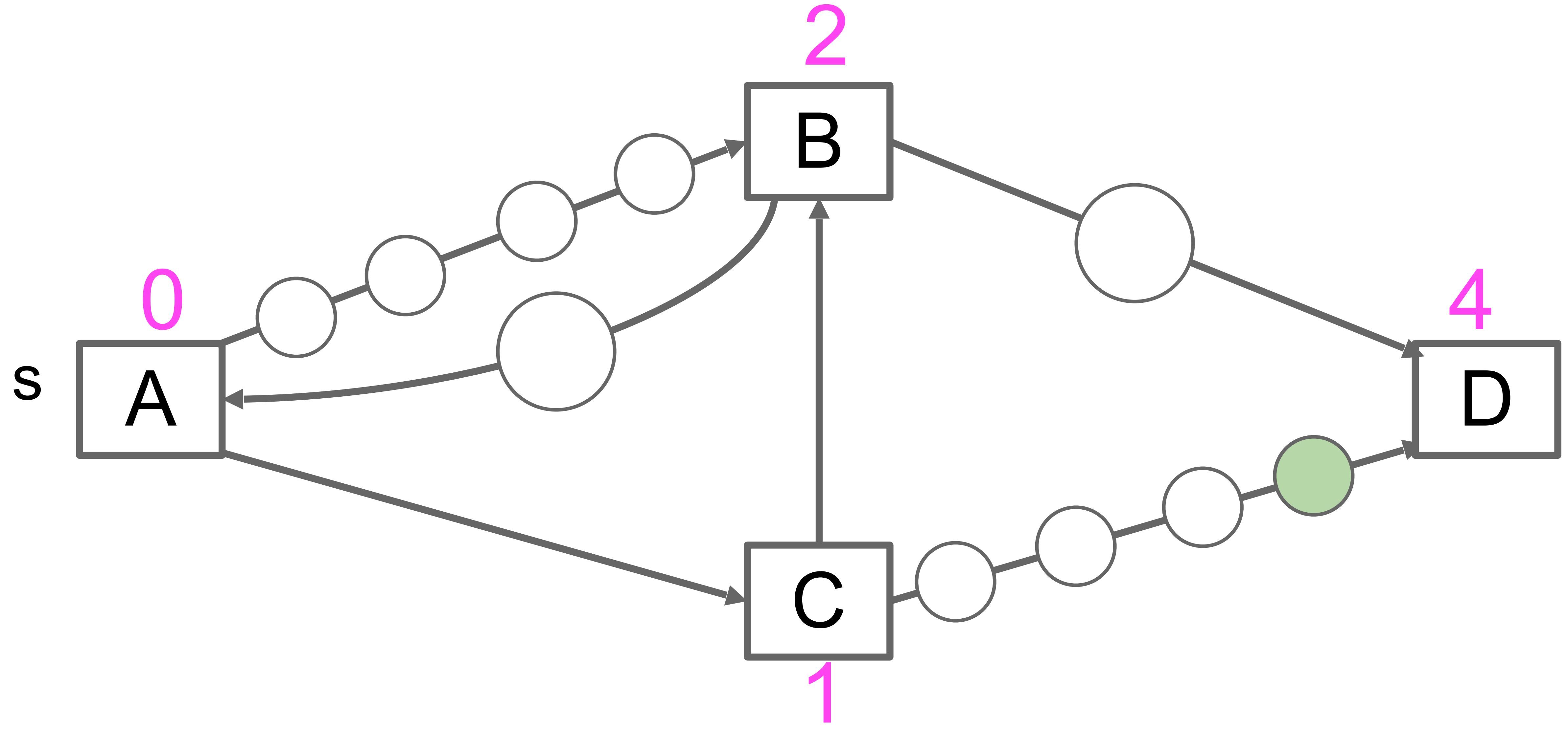
Order of visited nodes: ACB

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



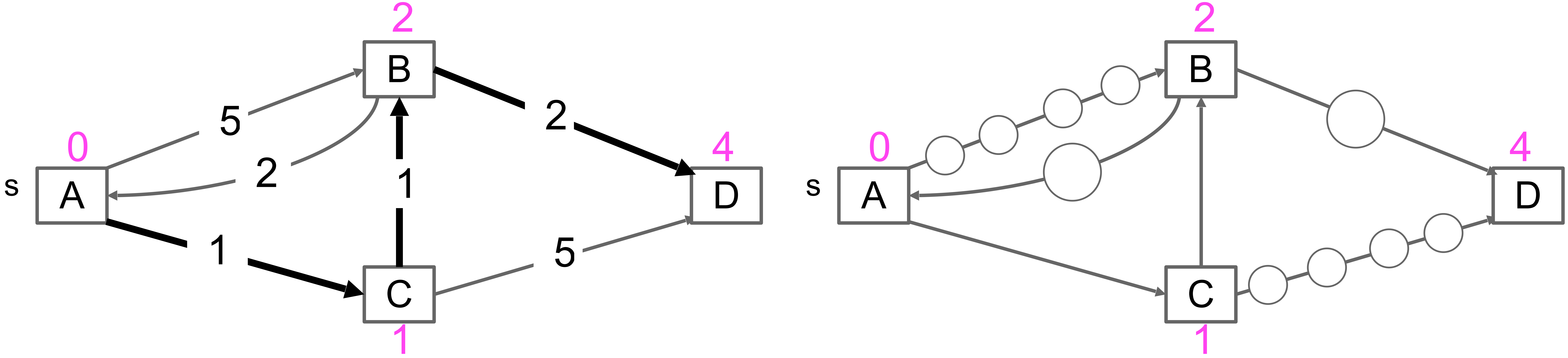


Order of visited nodes: ACBD

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



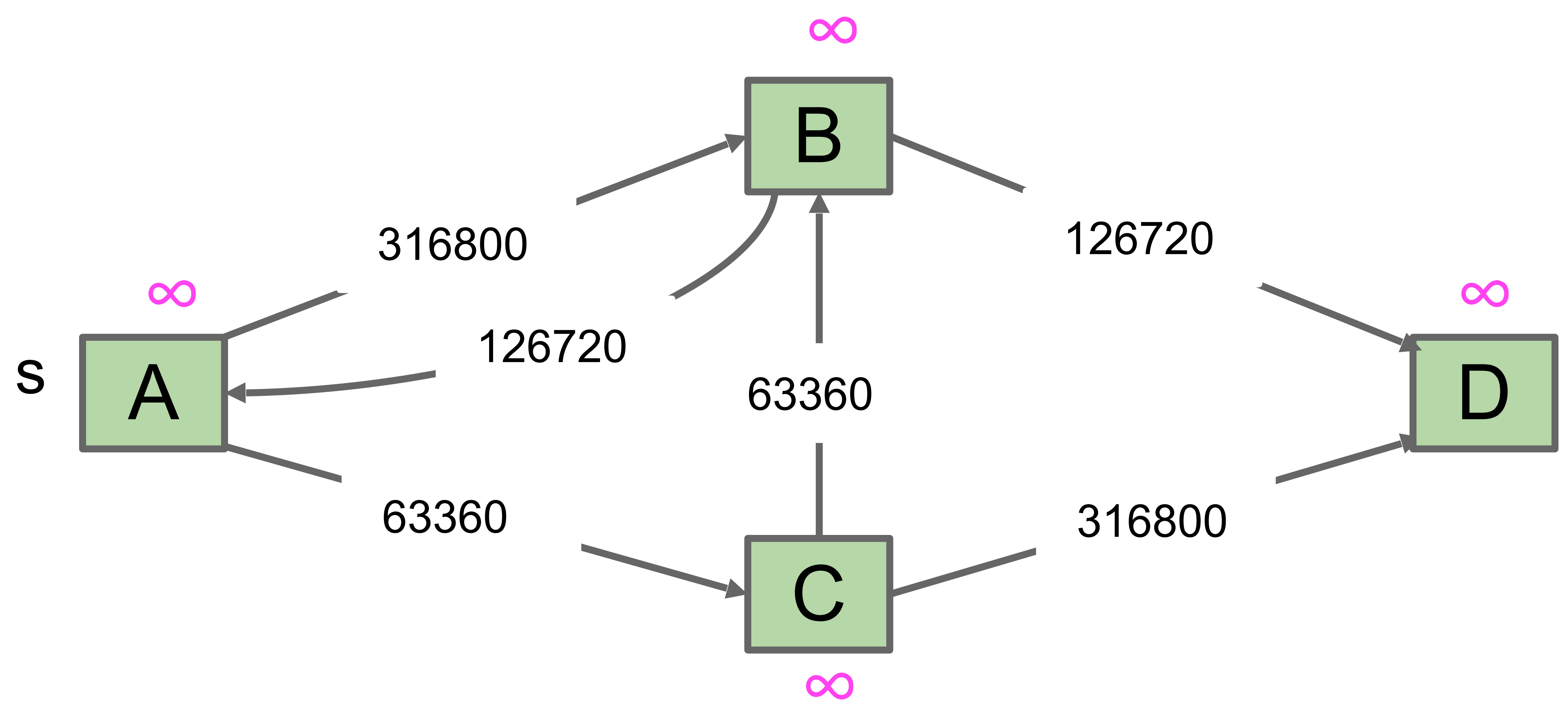
Order of visited nodes: ACBD

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

Takeaways:

- It works, but can be really slow. For example, consider the graph below.
- What if we measured in inches instead of miles? Or had fractional weights?



Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

Takeaways:

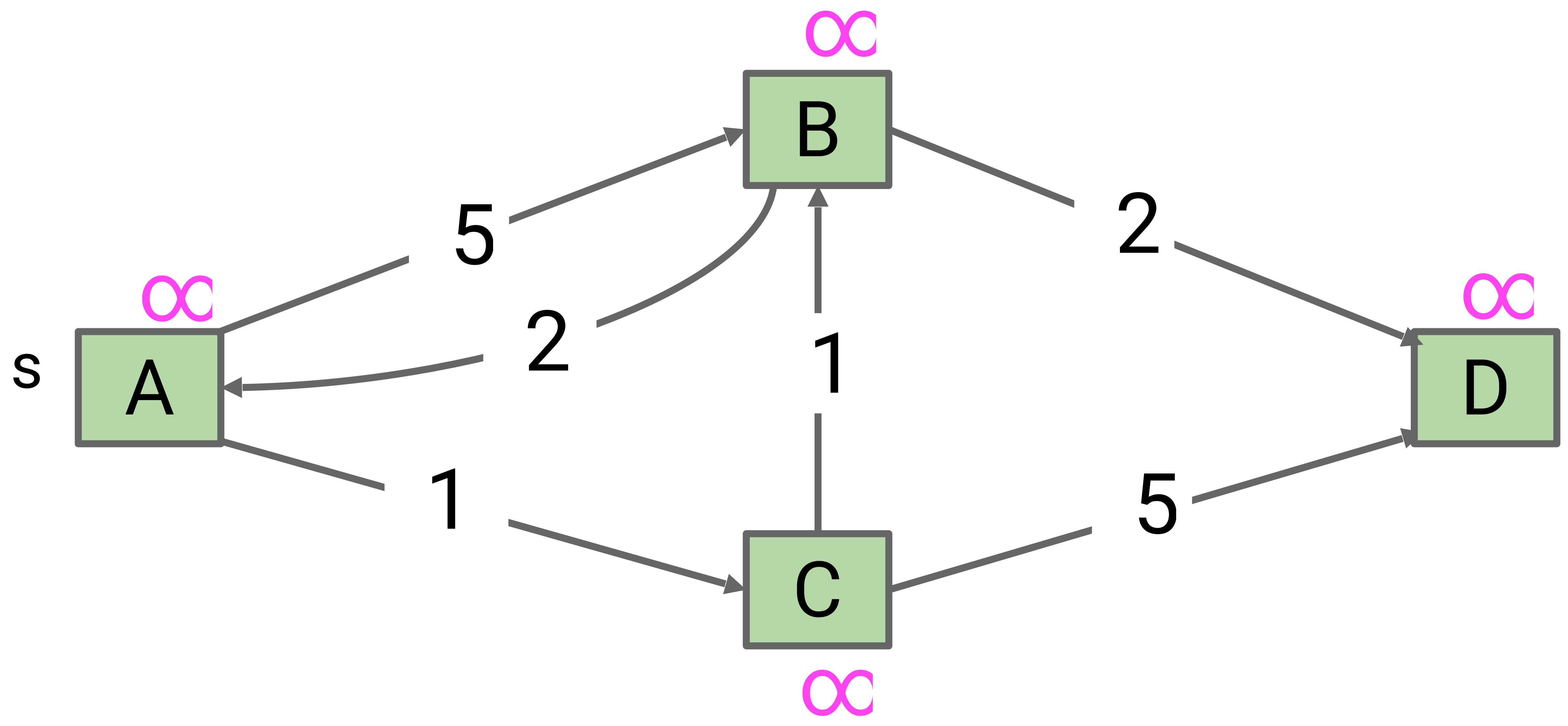
Algorithm #1 (BFS) visits:	Algorithm #2 (dummy nodes) visits:
every node 1 edge away,	every node distance 1 away,
then every node 2 edges away,	then every node distance 2 away,
then every node 3 edges away, etc.	then every node distance 3 away, etc.

- Algorithm #2 order is sometimes called **best-first** order.
- Let's try to visit the nodes in the same order as Algorithm #2 did, but without creating dummy nodes.

Bad Algorithm #3 (Best-First Search)

Bad algorithm #3: Perform best-first search.

- Similar to BFS, but we remove the closest edge from the fringe each time.
- We can use a **priority queue** to track the closest edge.



Bad Algorithm #3 (Best-First Search)

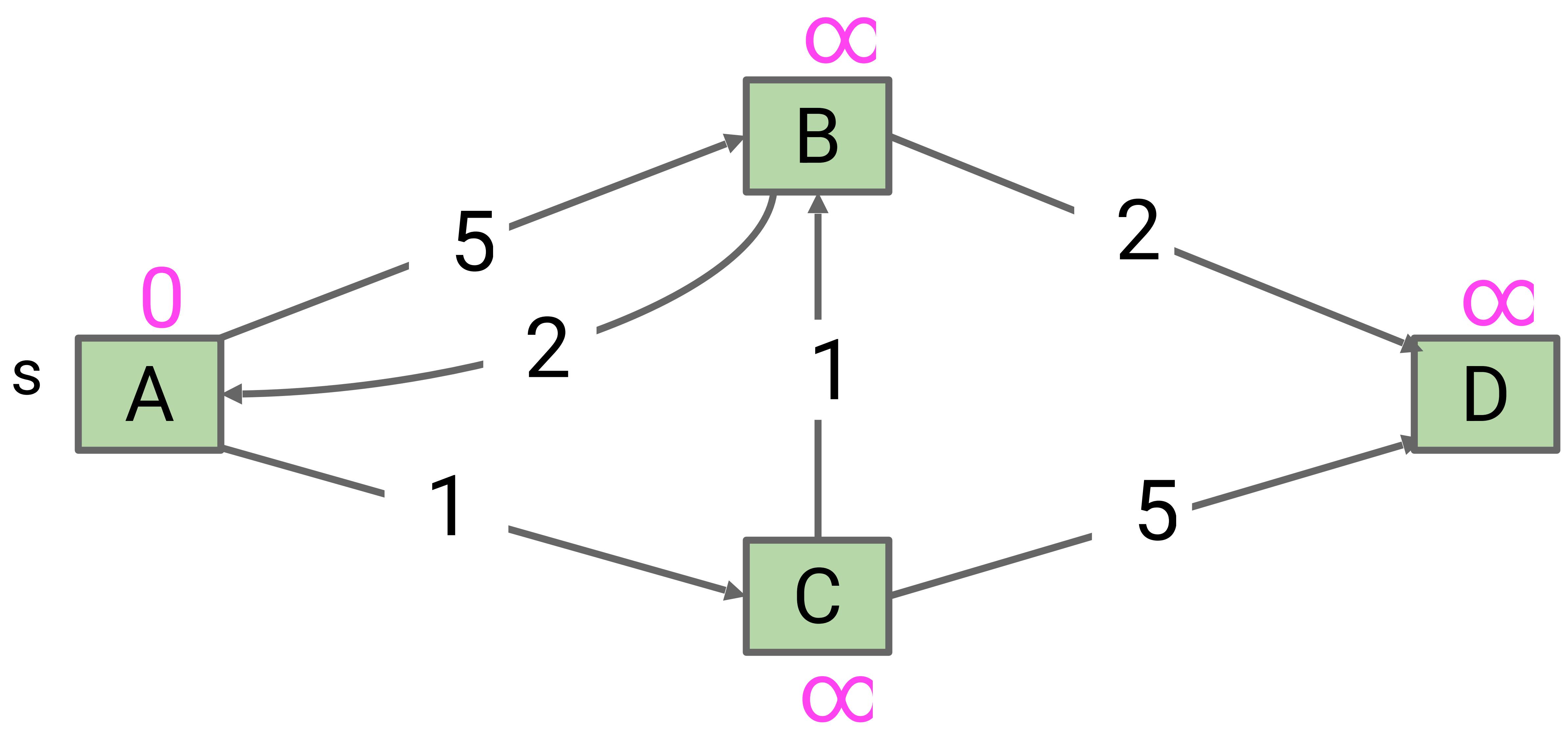
Add the start (A) to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

Only difference from Algorithm #1:
We added the word "closest".



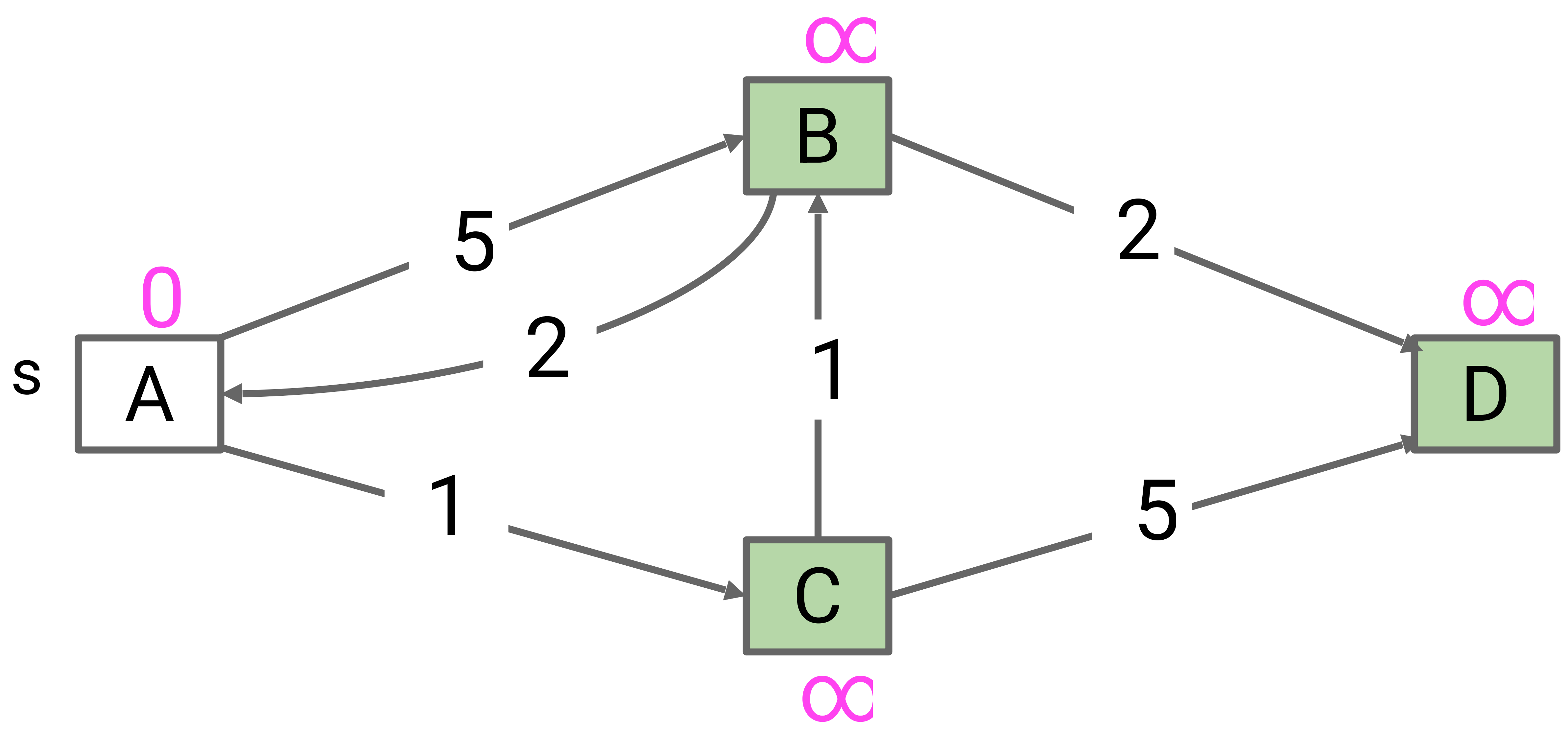
Fringe: [A=0]

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

> **Remove the closest vertex from the fringe and mark it.**
>
> For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



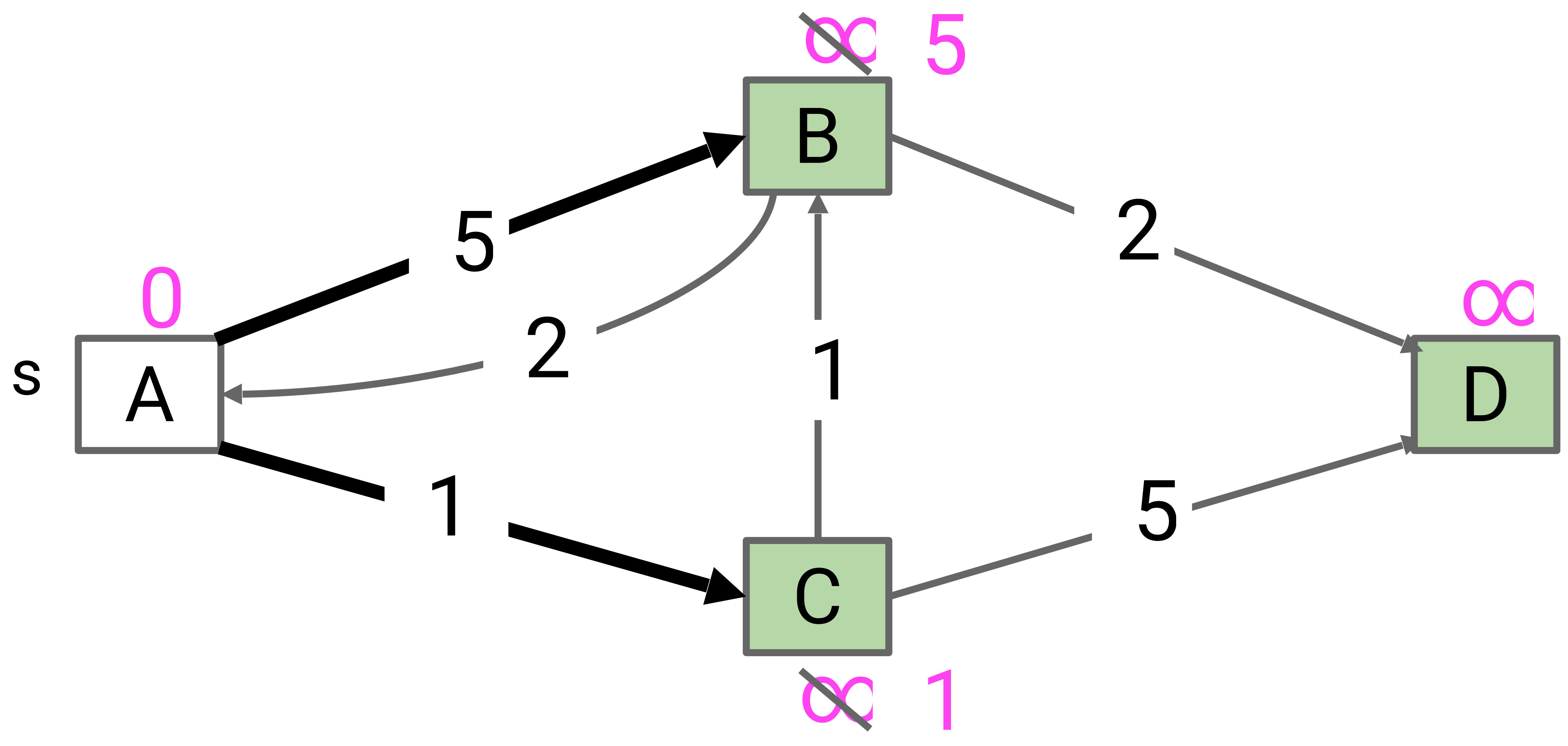
Fringe: [~~A=0~~]
Removed vertex: A

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



Fringe: [~~A=0,~~ C=1, B=5]
Removed vertex: A

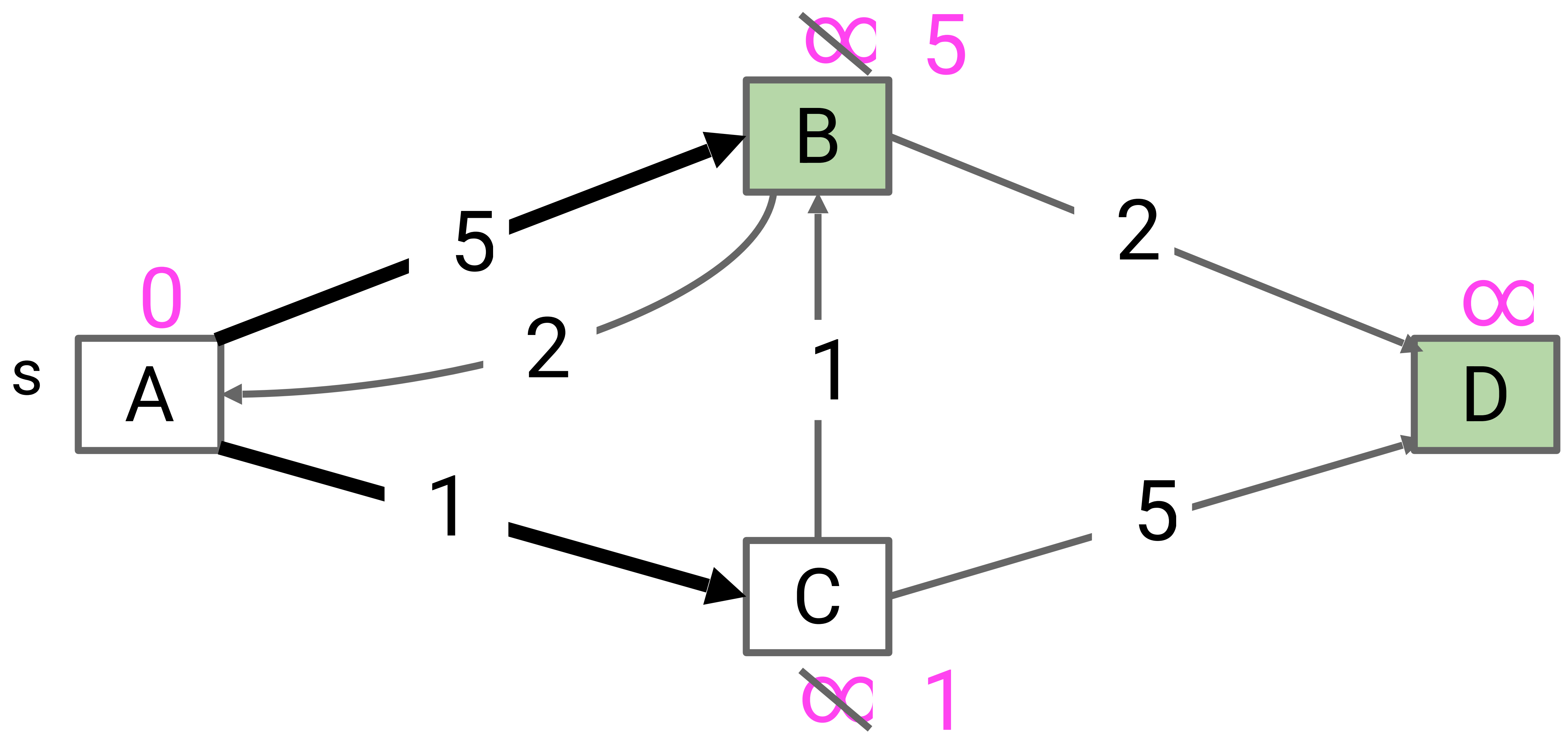
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

> **Remove the closest vertex from the fringe and mark it.**
>
> For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

In BFS, we removed B here, but in best-first, we're removing C because it's closer.



Fringe: [~~A=0~~, ~~C=1~~, B=5]
Removed vertex: C

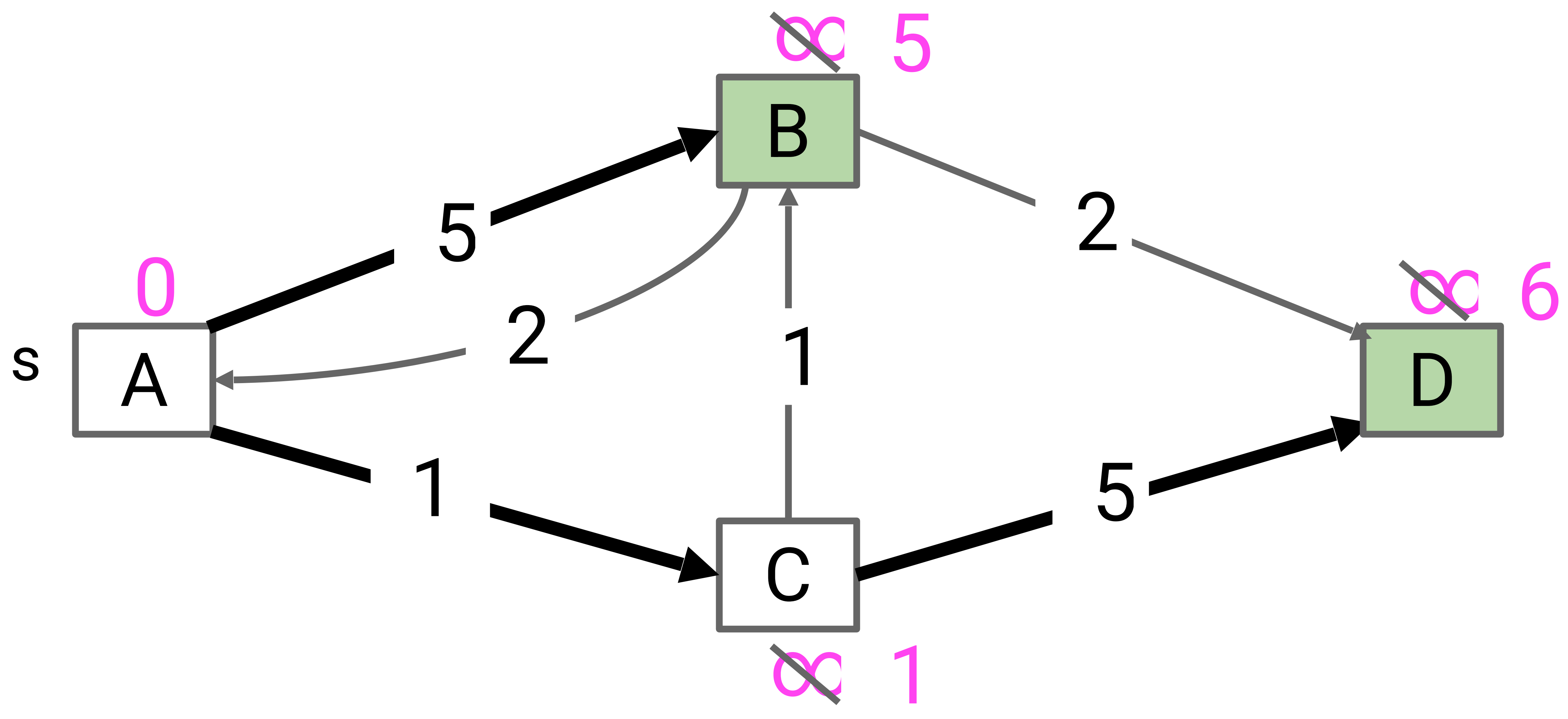
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

  Remove the closest vertex from the fringe and mark it.

  For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



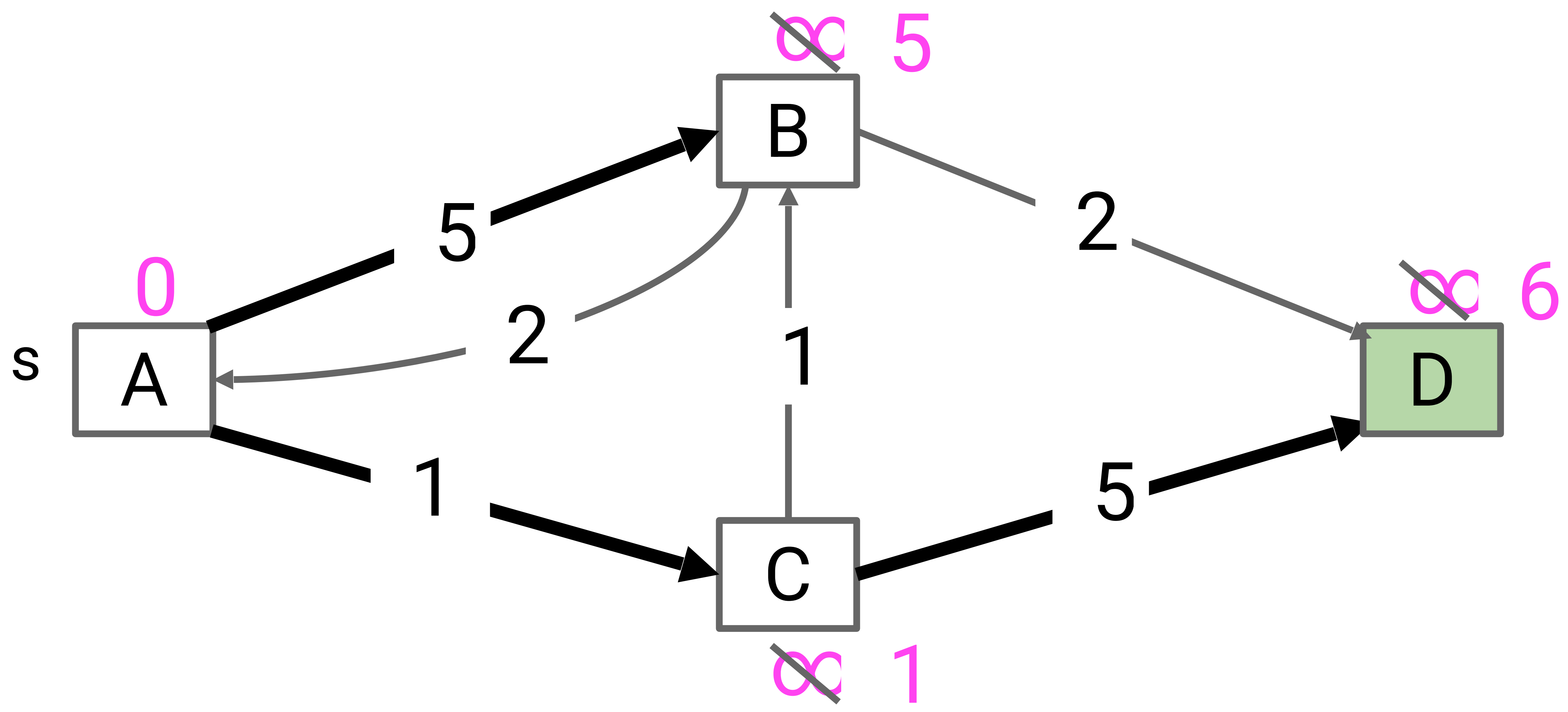
Fringe: [~~A=0, C=1,~~ B=5, D=6]
Removed vertex: C

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

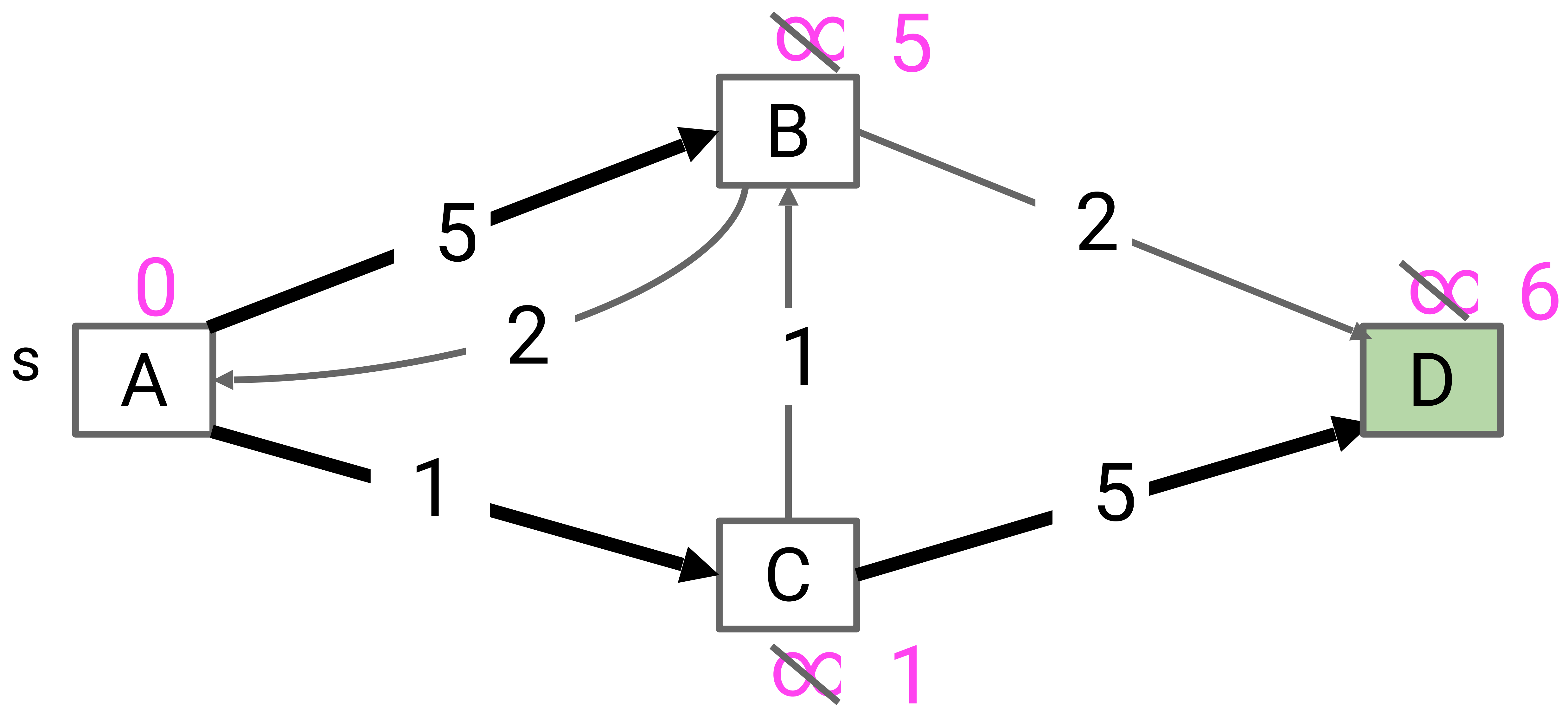
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**

The only outgoing edge is B→D.
D is already part of the SPT, so do nothing.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

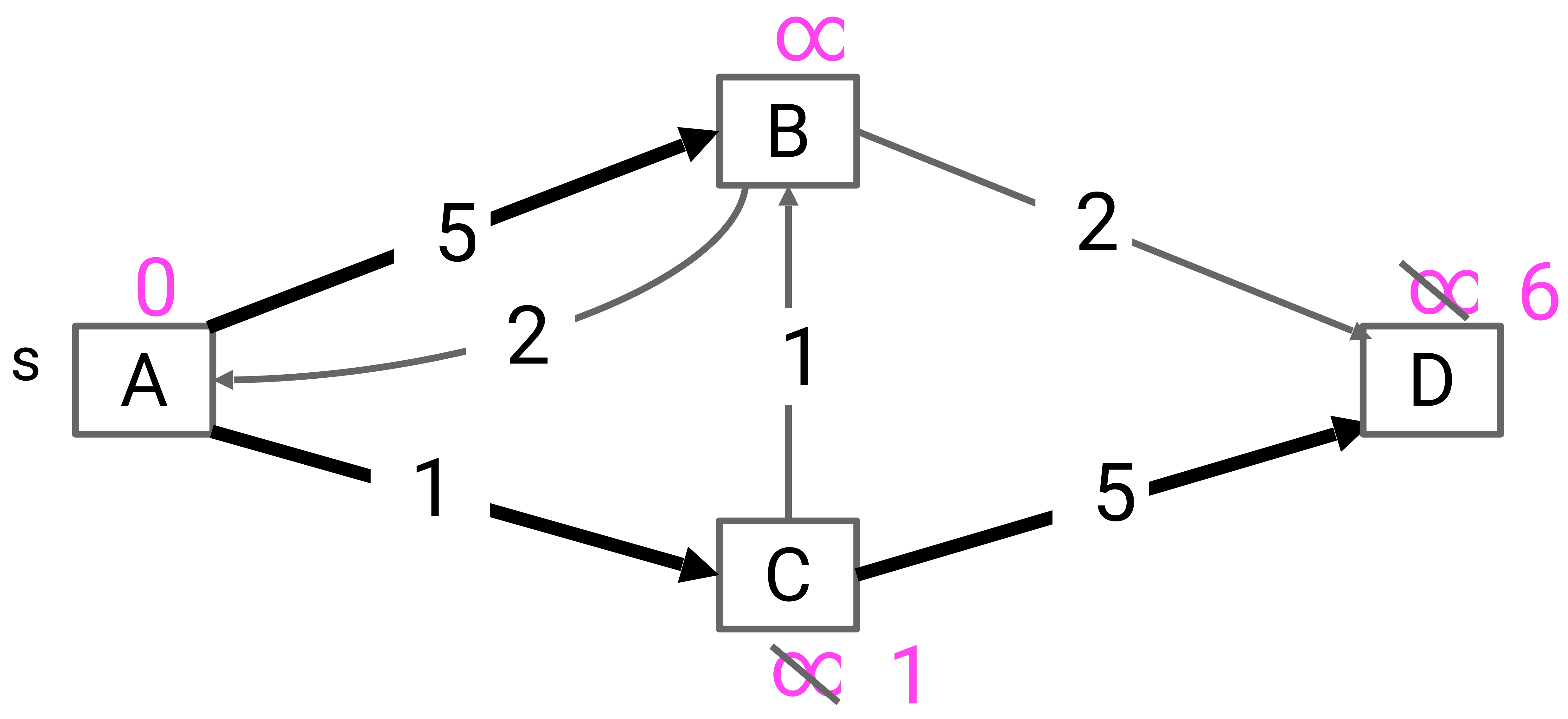
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

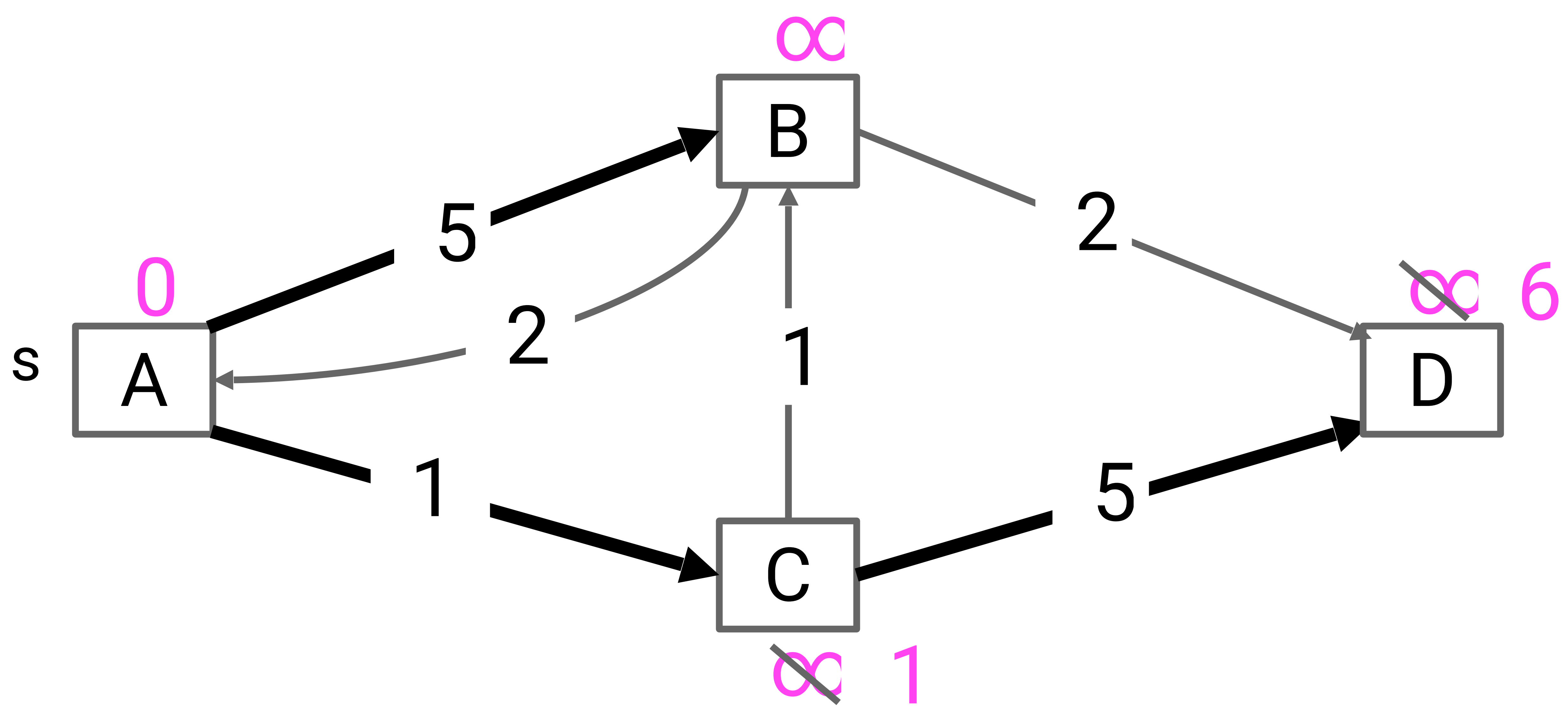
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

> Remove the closest vertex from the fringe and mark it.
>
> **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**

No outgoing edges from D, so do nothing.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

Bad Algorithm #3 (Best-First Search)

Bad algorithm #3: Perform best-first search.

- Similar to BFS, but we remove the closest edge from the fringe each time.
- We can use a priority queue to track the closest edge.

Takeaways:

- Pro: We visited the nodes in best-first order (same order as in Algorithm #2), without creating dummy nodes.
- Con: We got the wrong answer. Why?
- Let's revisit the step where things went wrong.

Bad Algorithm #3 (Best-First Search)

For each outgoing edge v→w: if w is not already part of SPT, add the edge, mark w, and add w to fringe.

C→B edge: B was in the SPT (via A→B), so we did nothing.

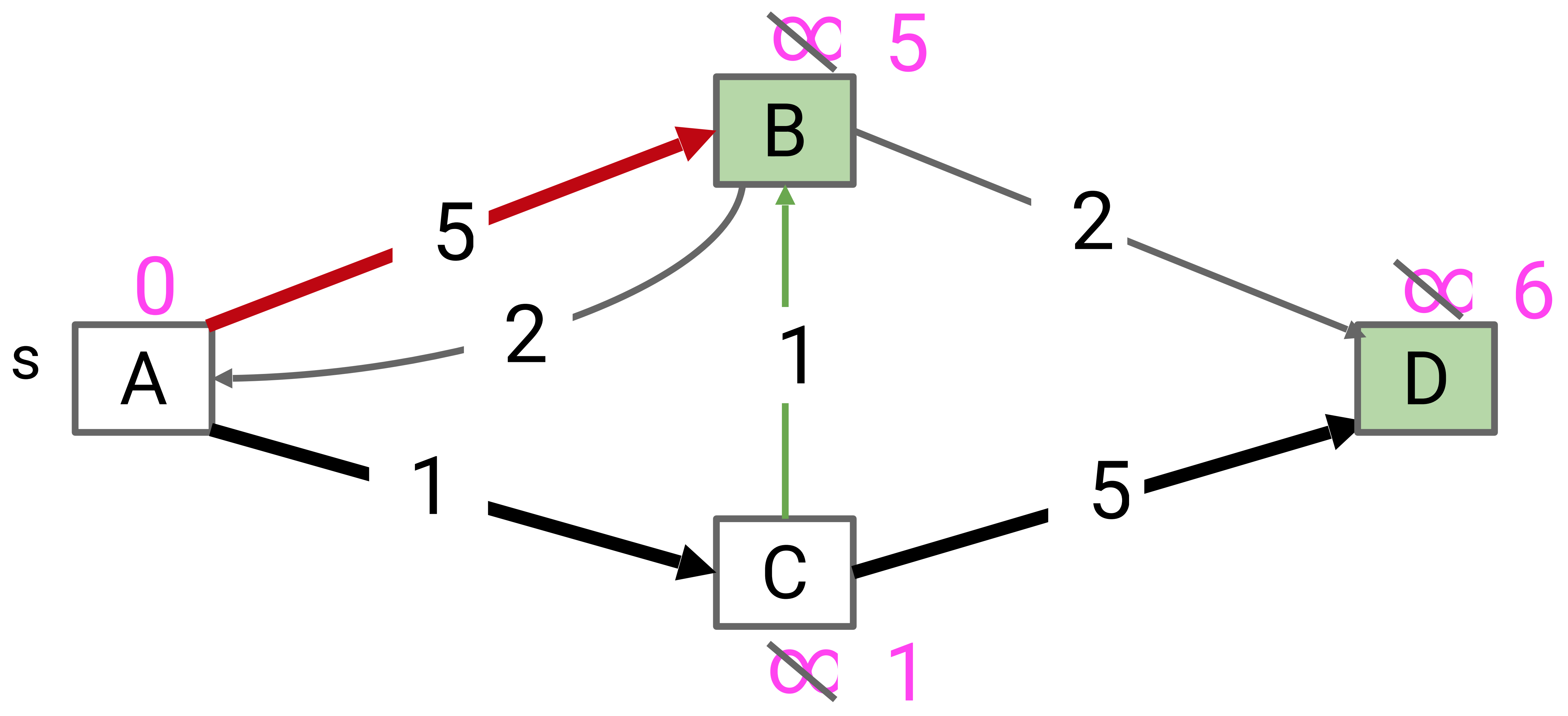
What should we have done here?

- We should have added edge C→B, and thrown out the old edge (A→B) to B. Why?

- The distance to B via C→B is 2.

This is better than the currently best known distance to B (5, via A→B).

Fringe: [~~A=0~~, ~~C=1~~, B=5, D=6]
Removed vertex: C

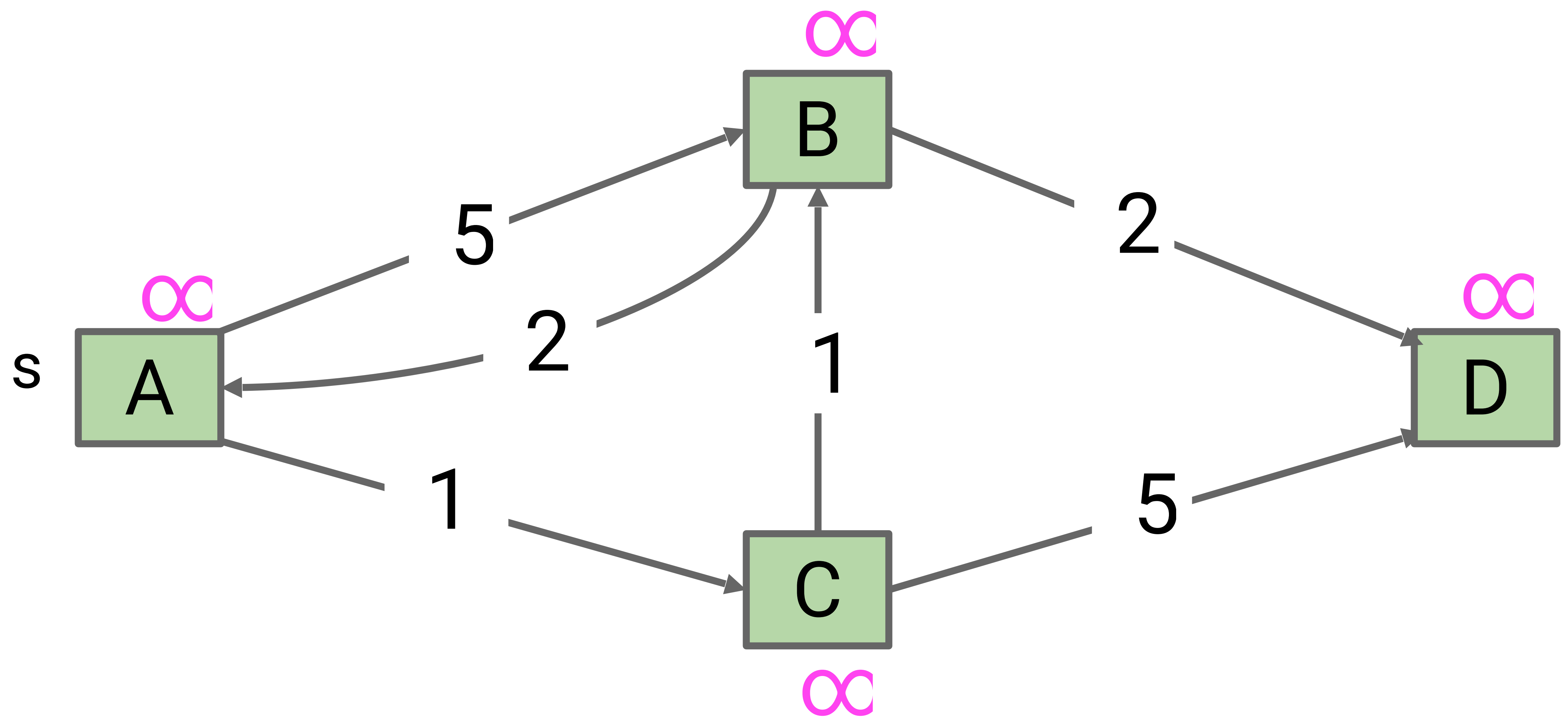


Finding a Shortest Paths Tree Algorithmically

Dijkstra's Algorithm:

- So far, we've added an edge v→w *if w is not already part of the SPT*.
- Instead, we should add an edge *if that edge yields better distance*.
- Use the priority queue to track best known distances.

We'll call this process "edge **relaxation**".



Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

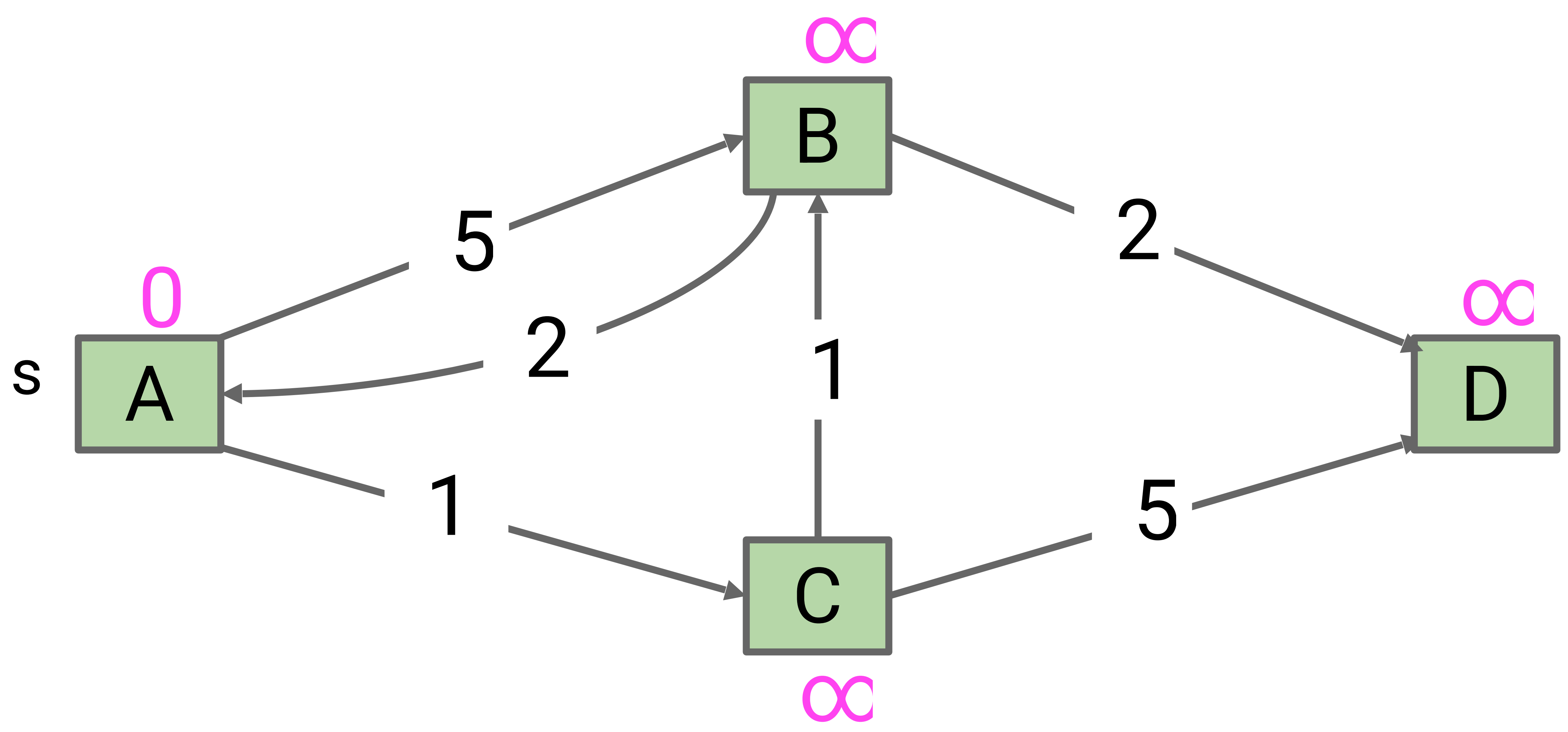
Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: $\boxed{\text{if the edge gives a better distance to w,}}$ add the edge, and update w in the fringe.

Key difference from Algorithm #3:
The condition for adding an edge.
(This used to say "if w not in SPT").

Extra bookkeeping: Instead of adding to the fringe as we go, we'll add all vertices to start.
This lets us track the best known distance to each vertex.

Fringe: [A=0, B=∞, C=∞, D=∞]



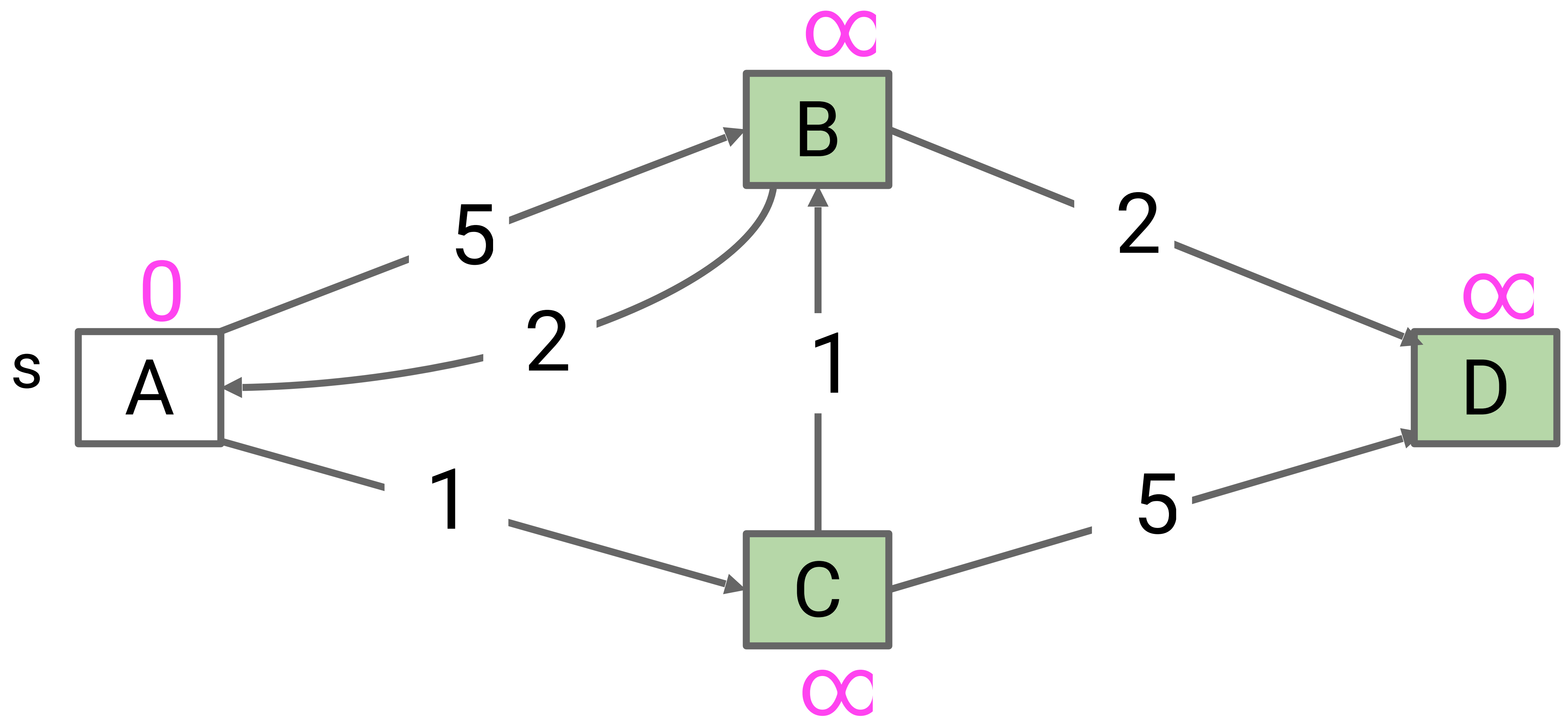
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



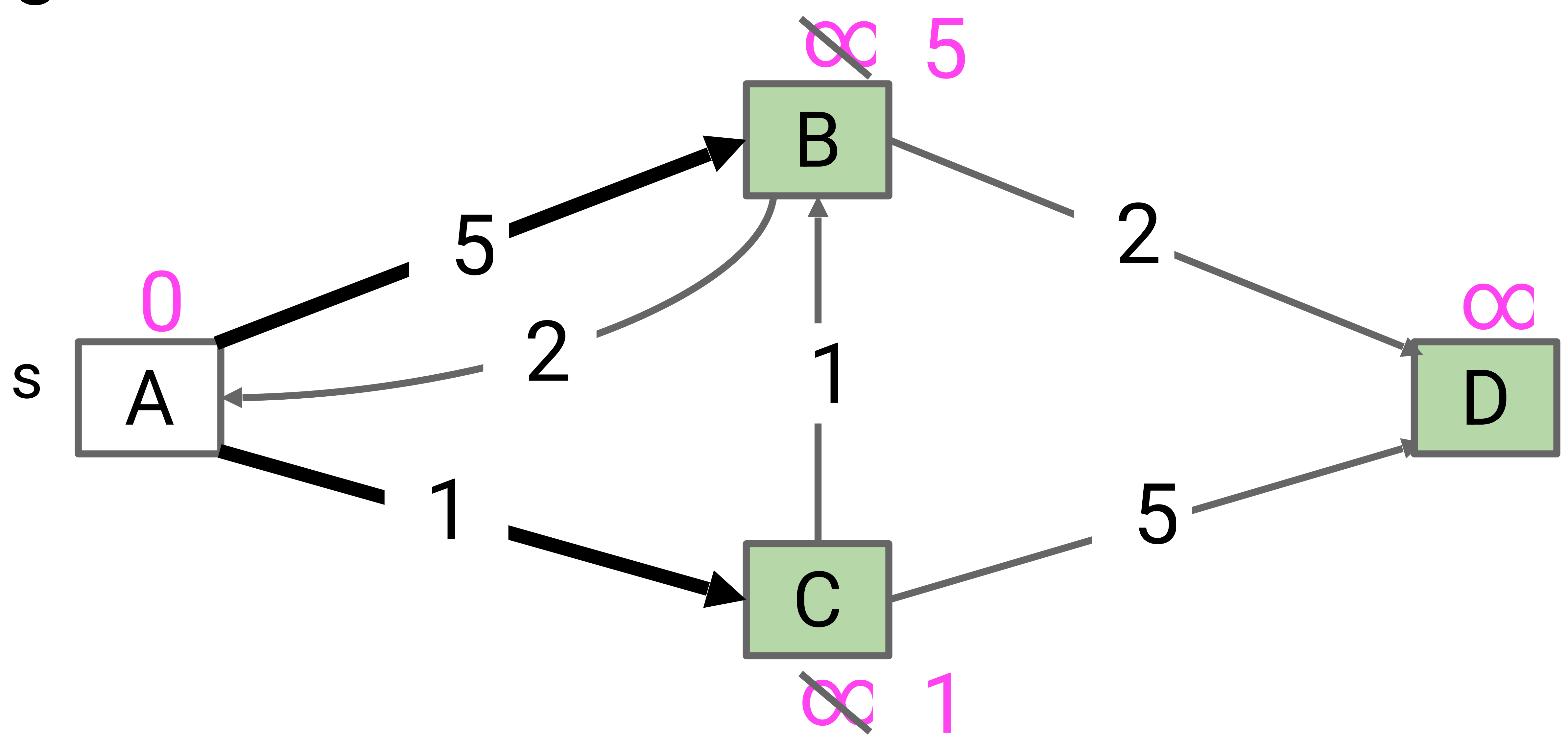
Fringe: [~~A=0~~, B=∞, C=∞, D=∞]
Removed vertex: A

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.**



Fringe: [~~A=0~~, C=1, B=5, D=∞]
Removed vertex: A

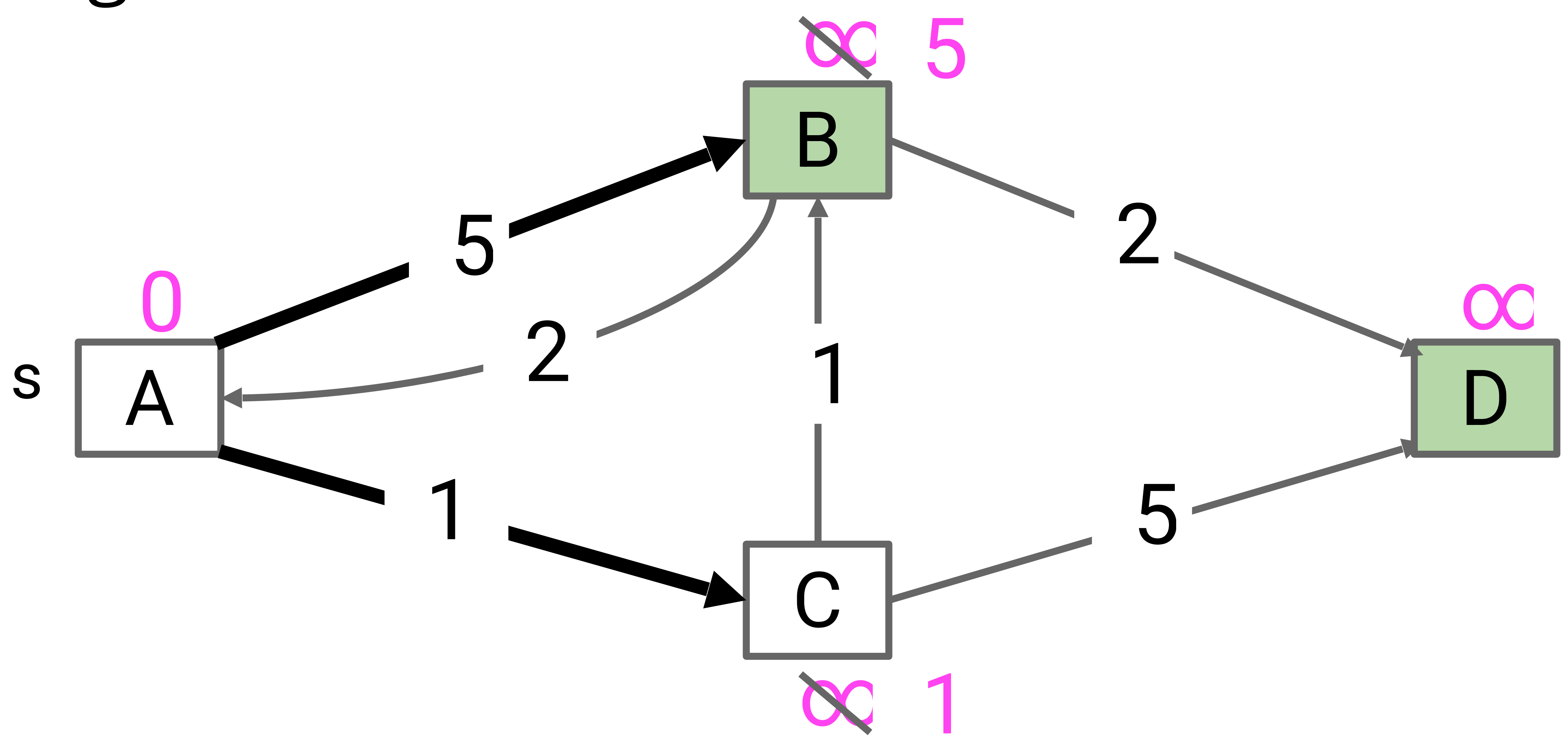
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, B=5, D=∞]
Removed vertex: C

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

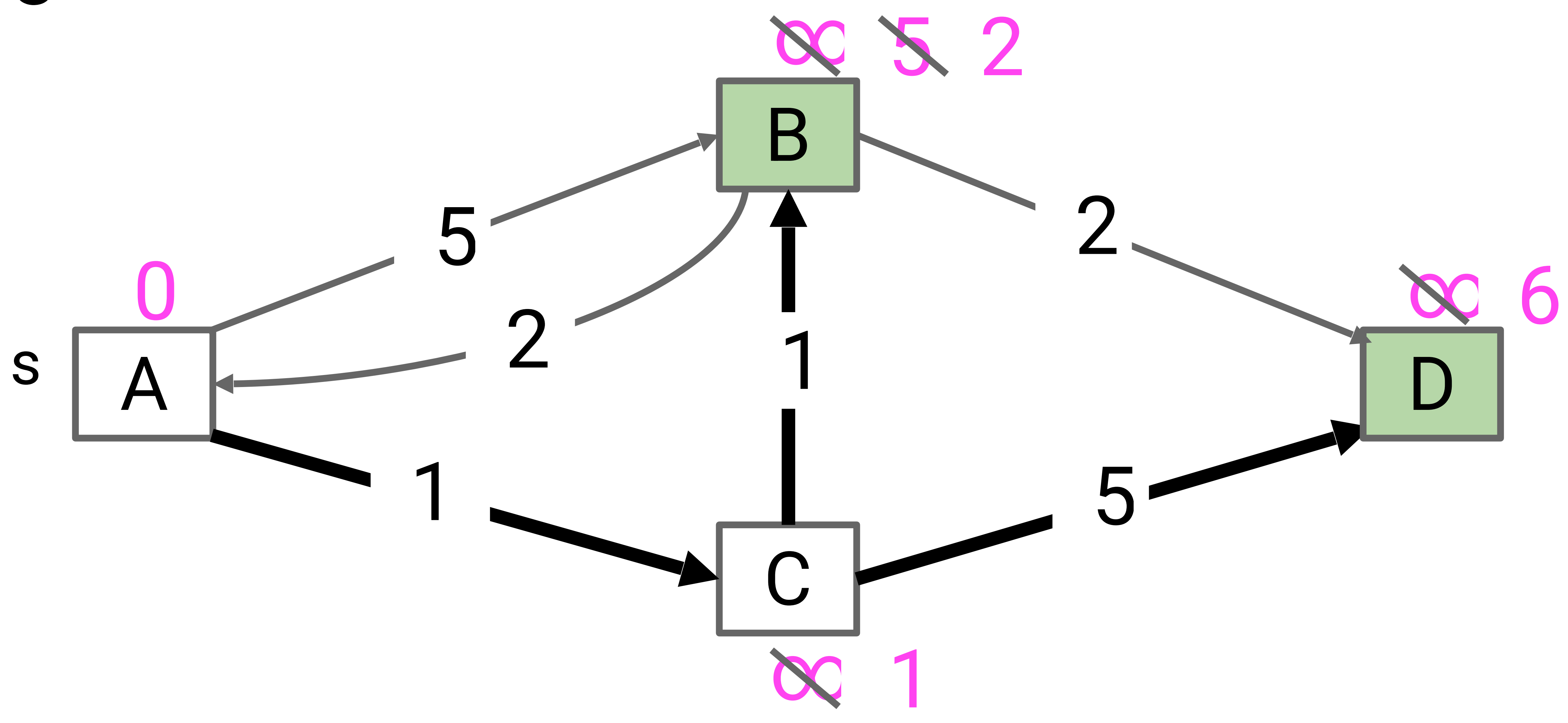
While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.

Improvement: We used C→B because the distance via C→B (2) is better than the distance via A→B (5).
This also means we throw out the old edge (A→B) to B.

Fringe: [~~A=0~~, ~~C=1~~, B=2, D=6]
Removed vertex: C



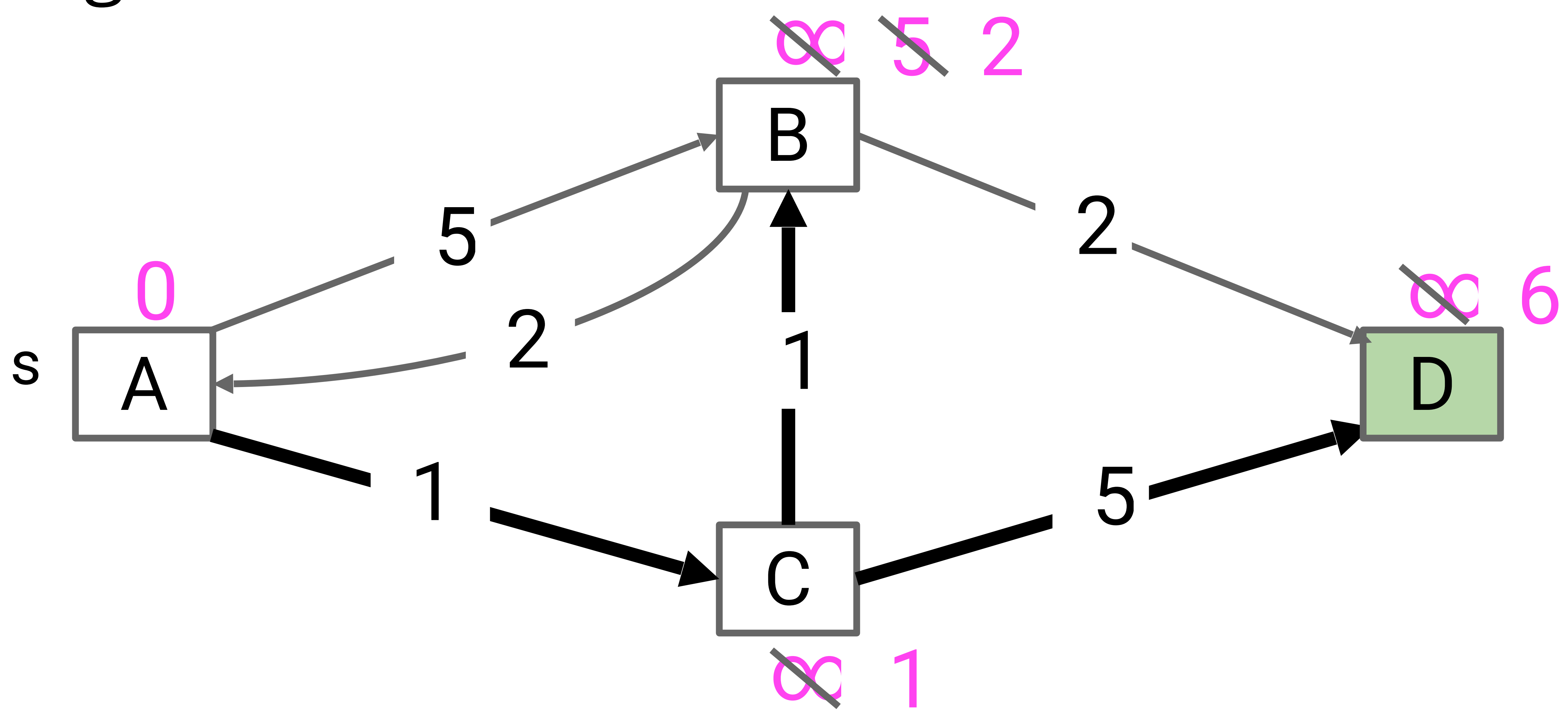
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=2~~, D=6]
Removed vertex: B

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

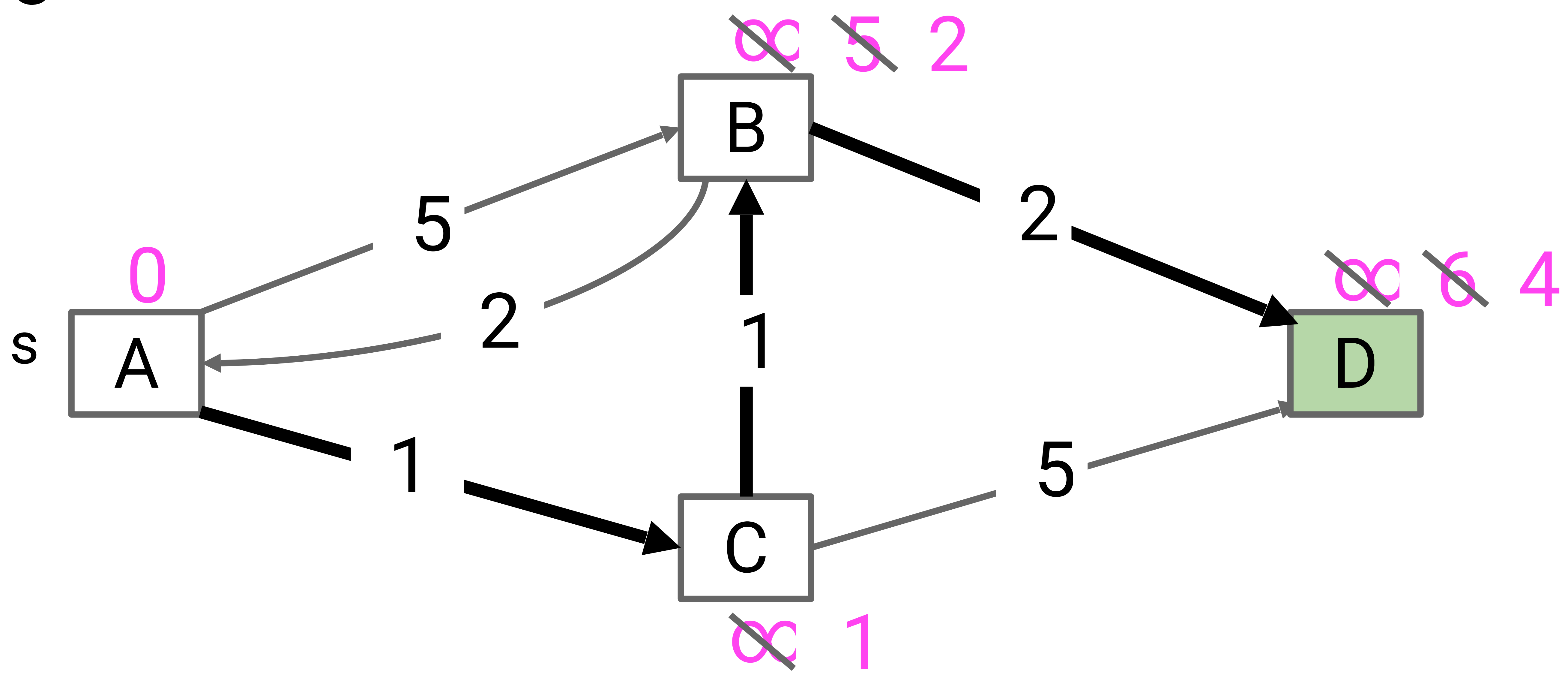
While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.

B→A (total=4) is not better than the best known way to A (0).

B→D (total=4) is better than the best known way to D (6, via C→D). So, we'll update the path to D.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=2~~, D=4]

Removed vertex: B

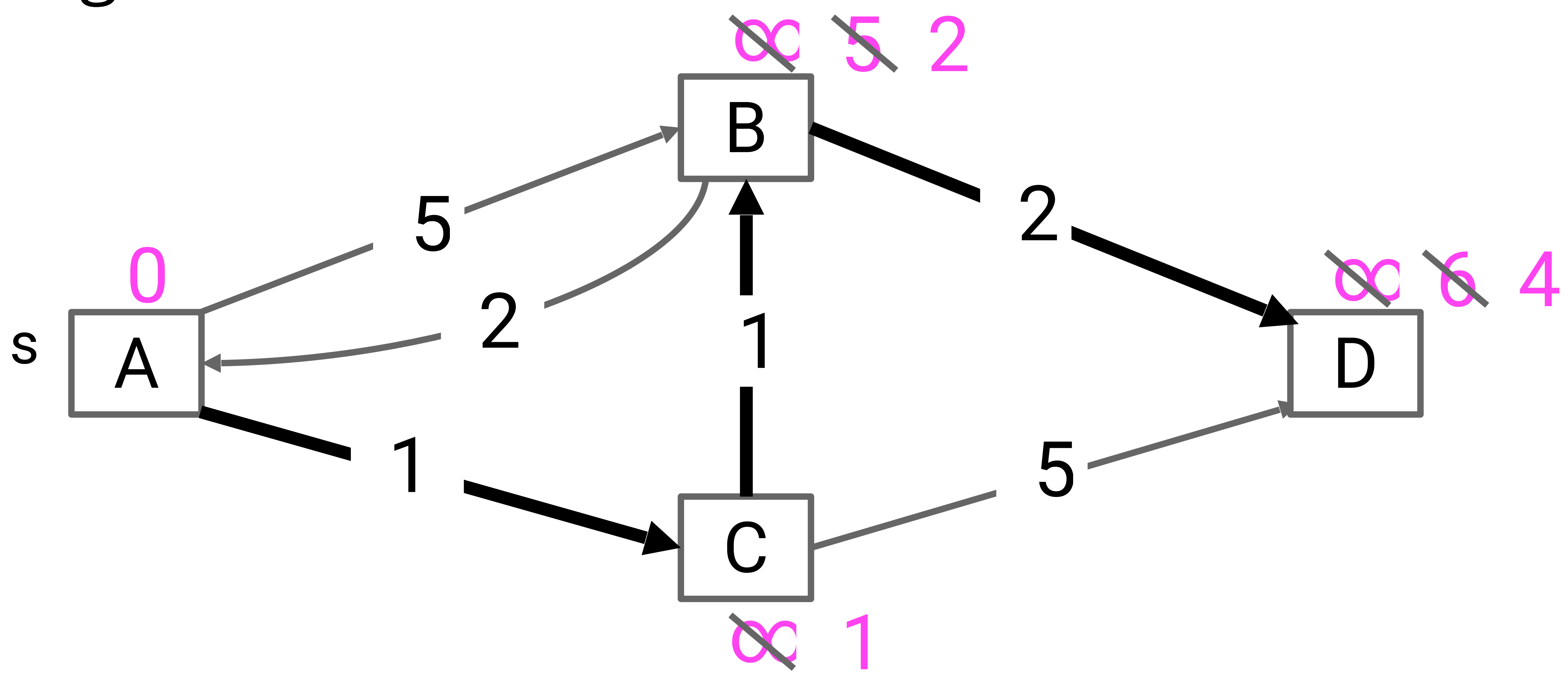
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=2~~, ~~D=4~~]
Removed vertex: D

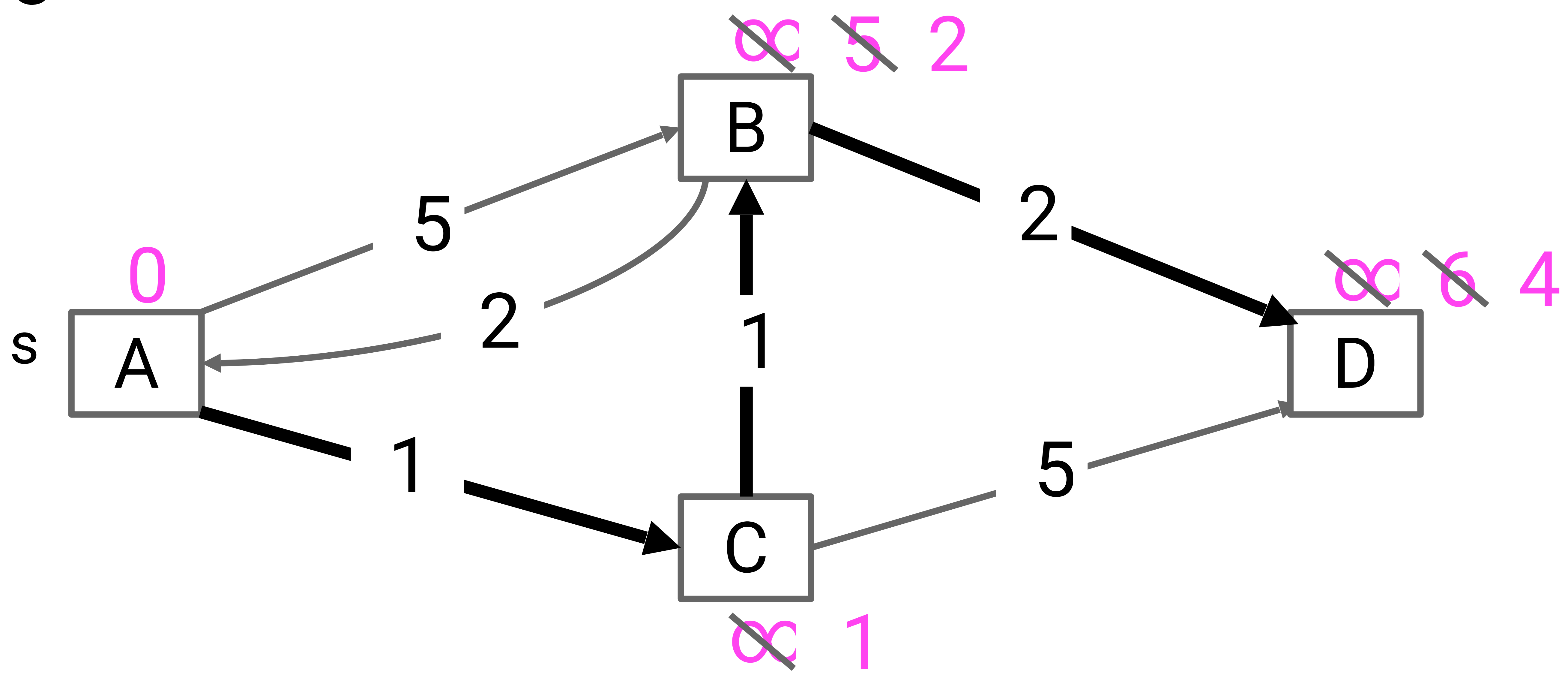
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.**

No outgoing edges from D, so do nothing.



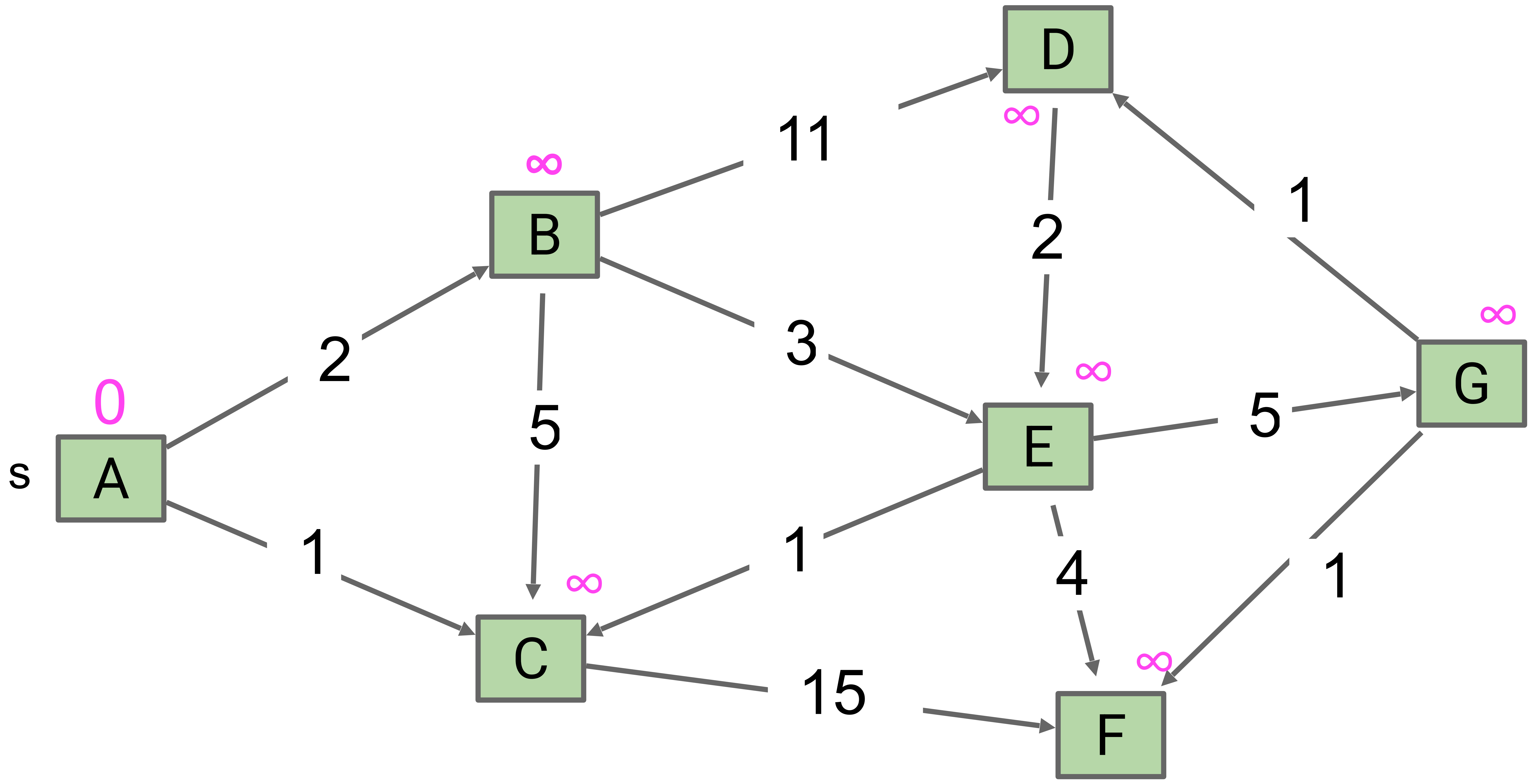
Fringe: [~~A=0~~, ~~C=1~~, ~~B=2~~, ~~D=4~~]
Removed vertex: D

Dijkstra’s Algorithm

Dijkstra’s Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

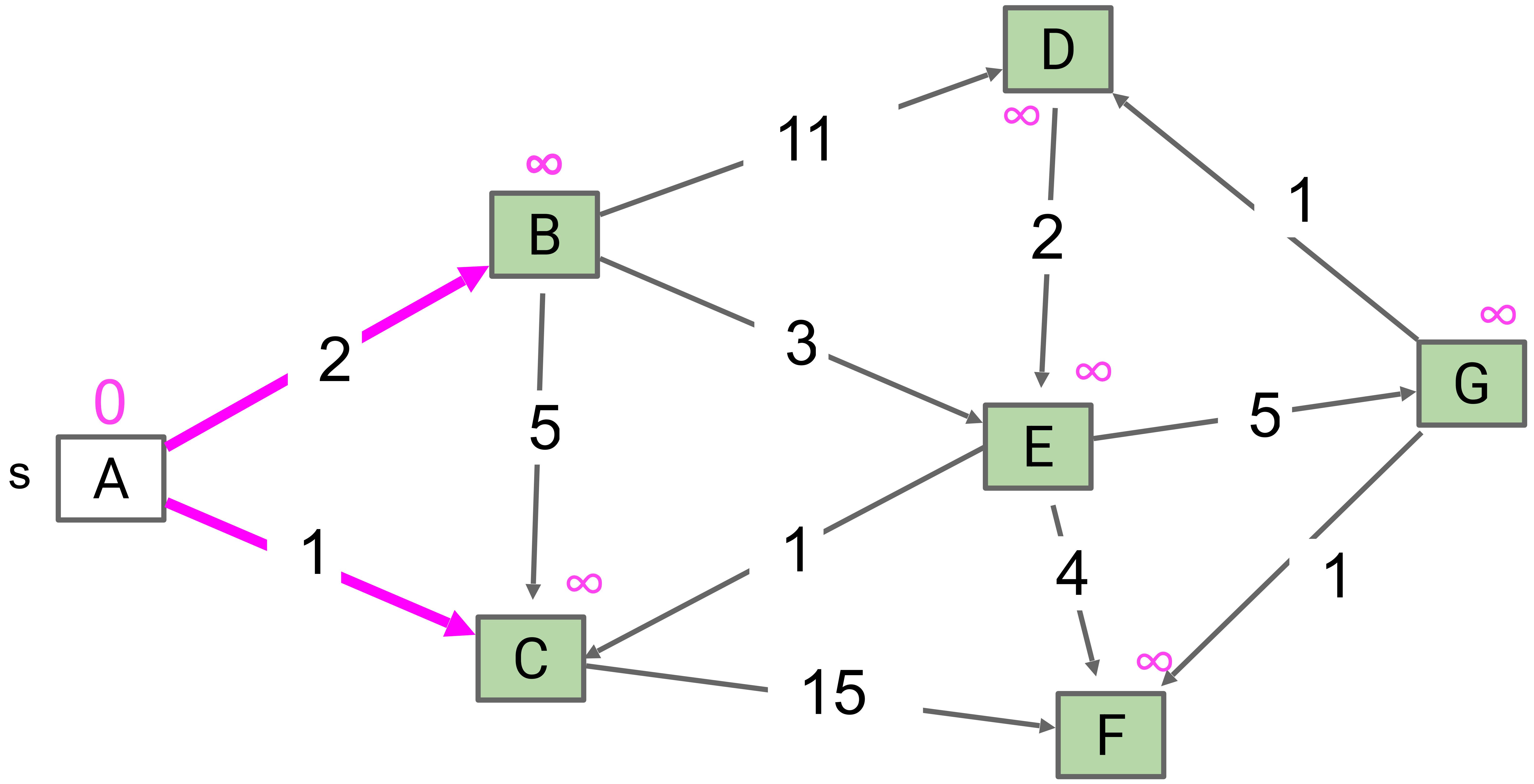


Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-
G	∞	-



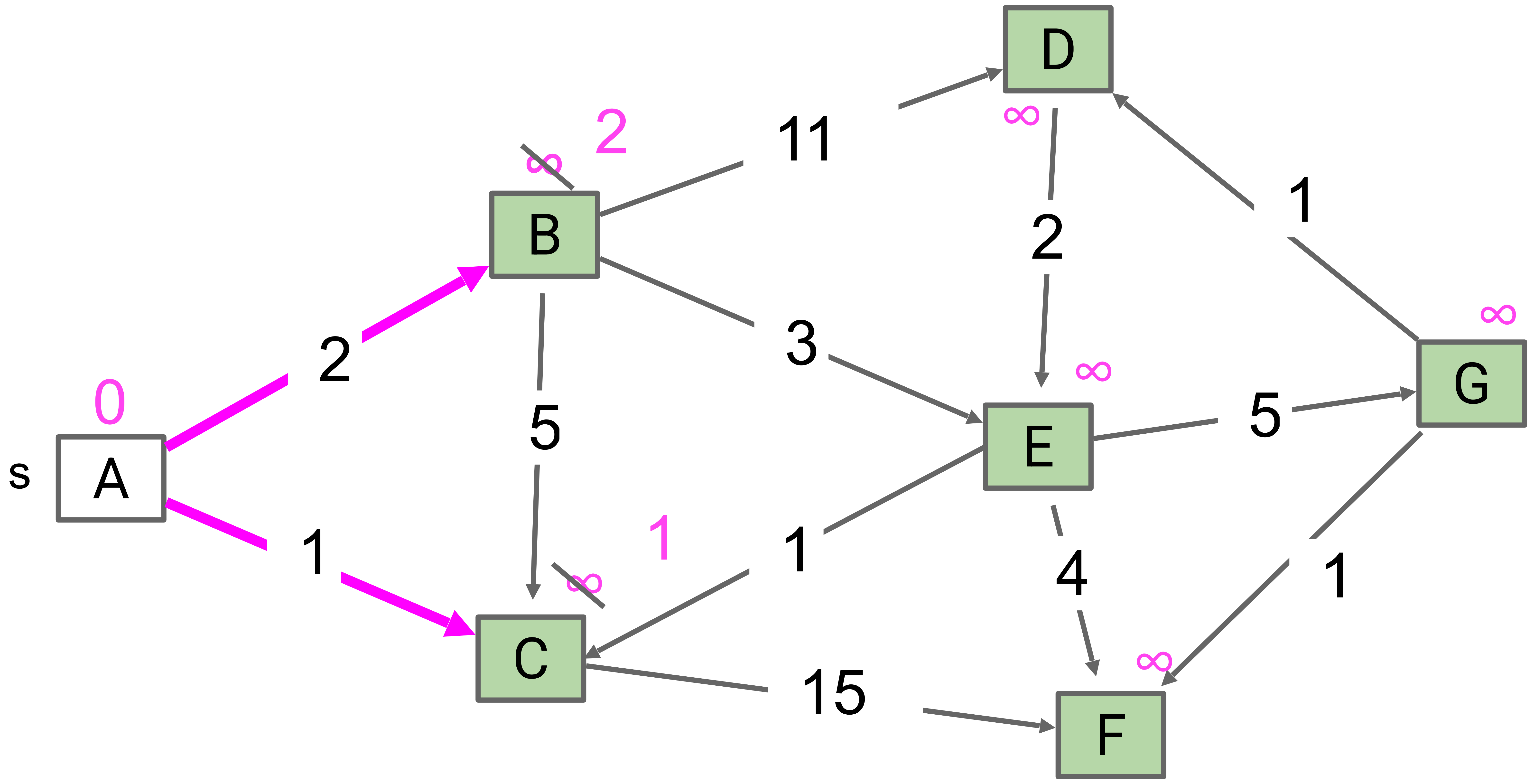
Fringe: [(B: ∞), (C: ∞), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra’s Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	∞	-
G	∞	-



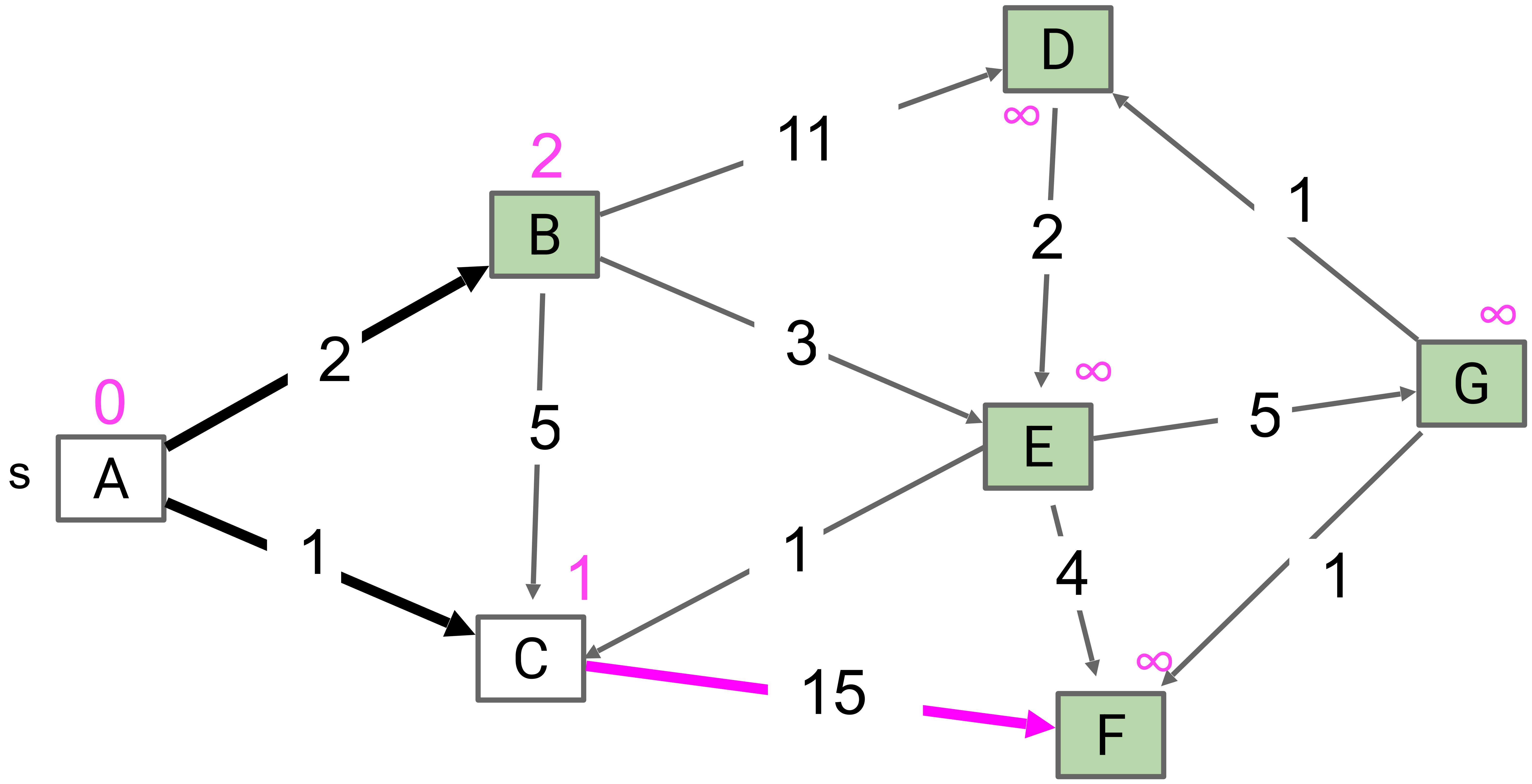
Fringe: [(C: 1), (B: 2)], (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	∞	-
G	∞	-



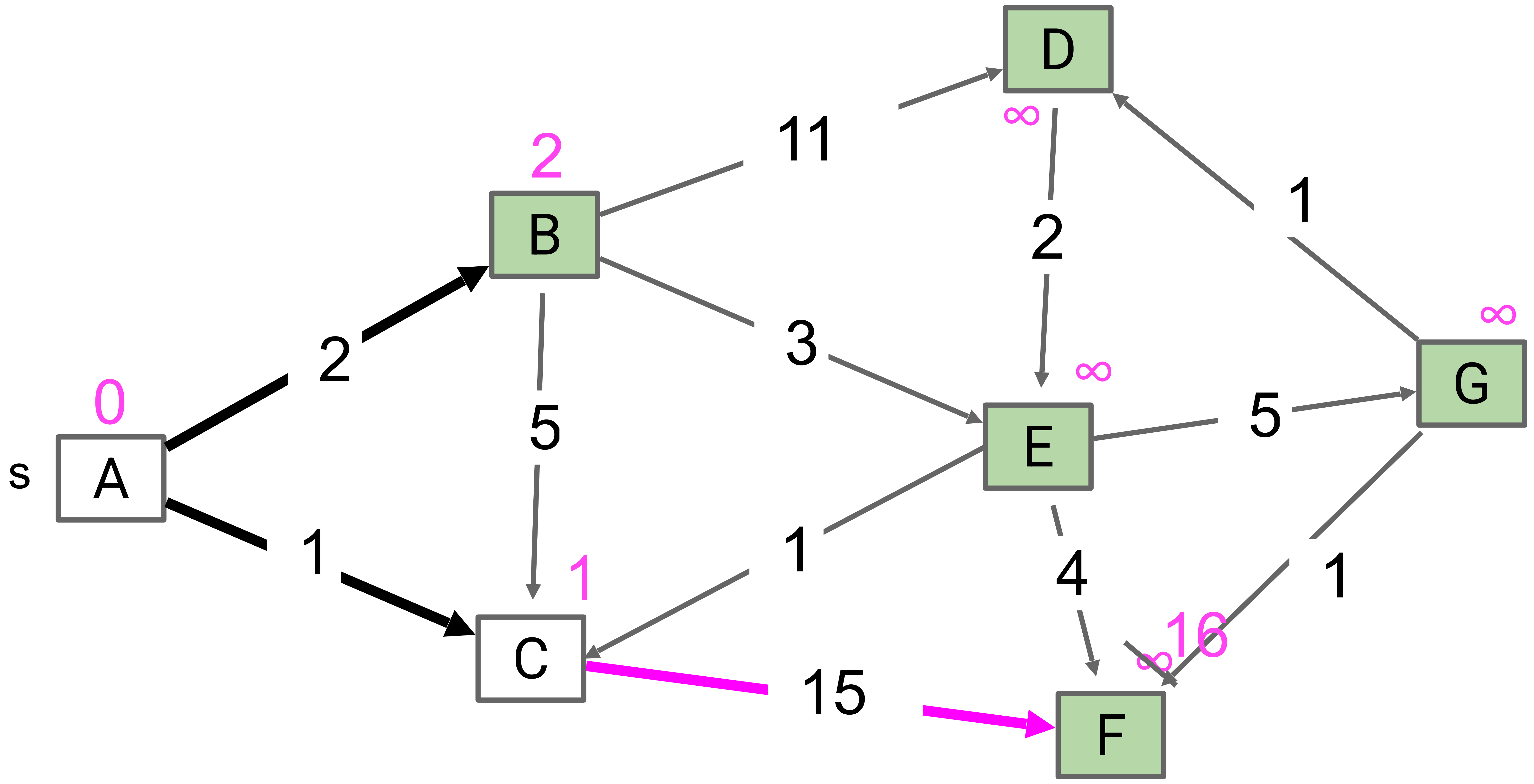
Fringe: [(B: 2), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	16	C
G	∞	-



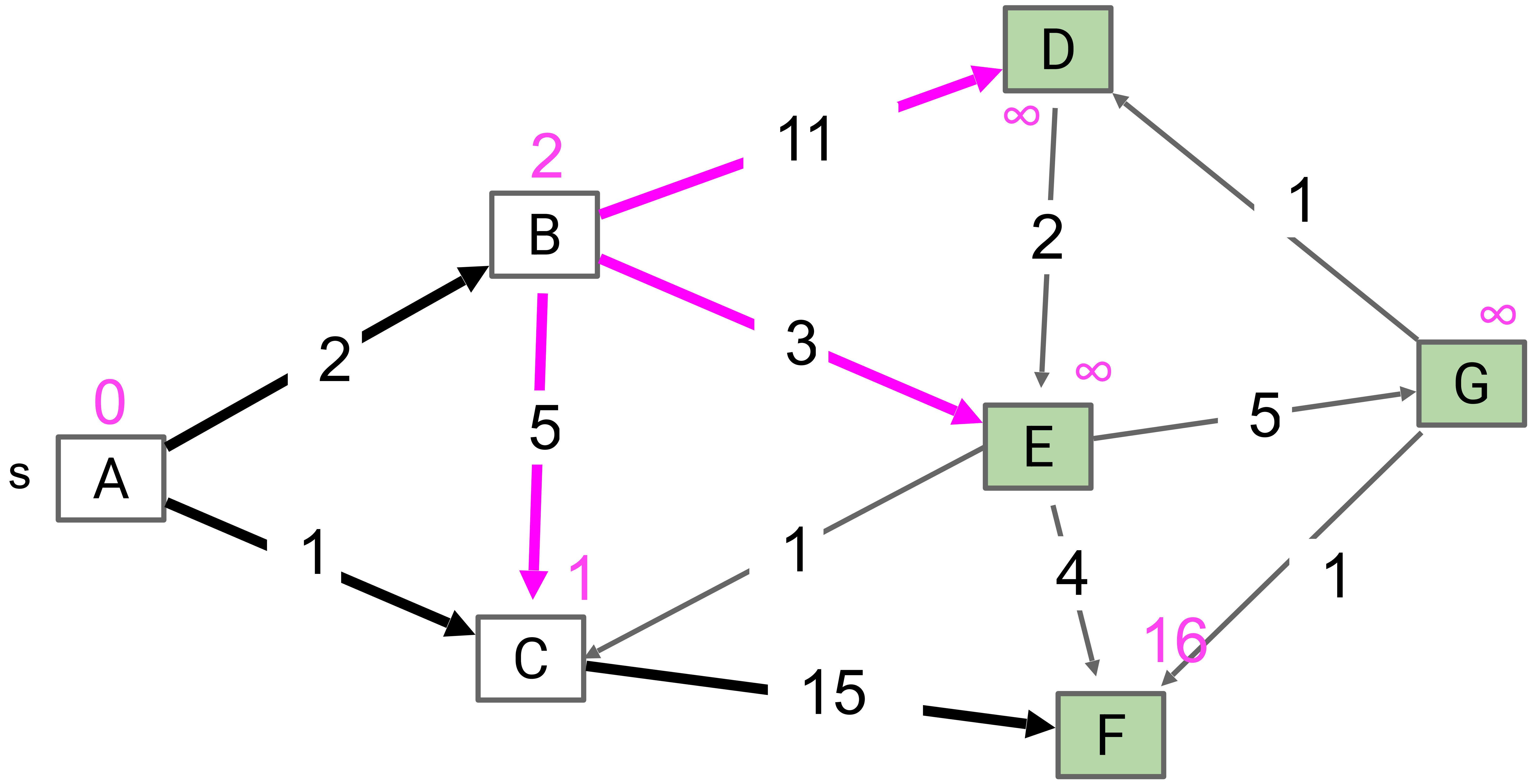

```
Fringe: [(B: 2), (F: 16), (D: ∞), (E: ∞), (G: ∞)]
```

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	16	C
G	∞	-



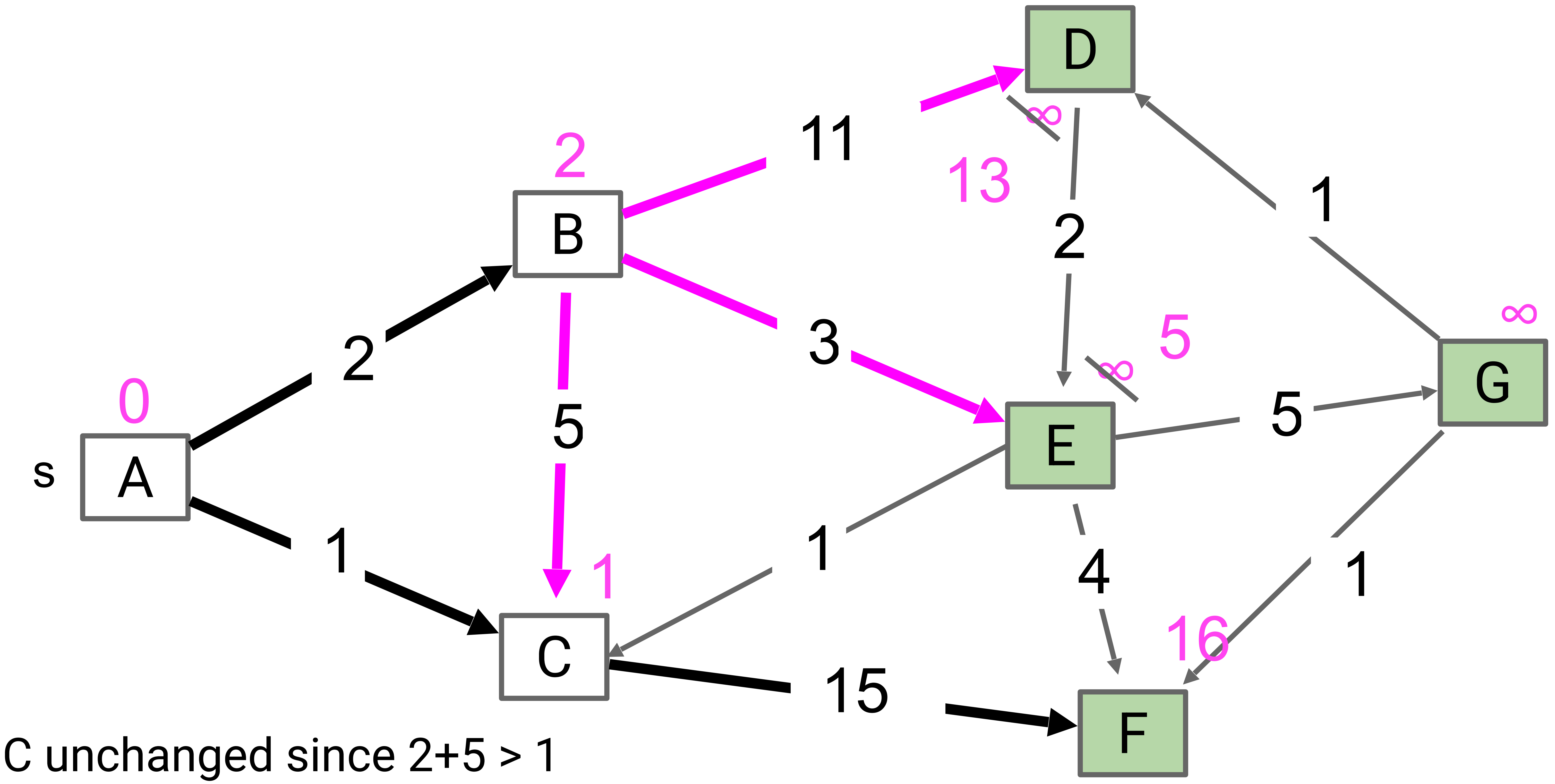
Fringe: [(F: 16), (D: ∞), (E: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



Vertex C unchanged since 2+5 > 1

`Fringe: [(E: 5), (D: 13), (F: 16), (G: ∞)]`

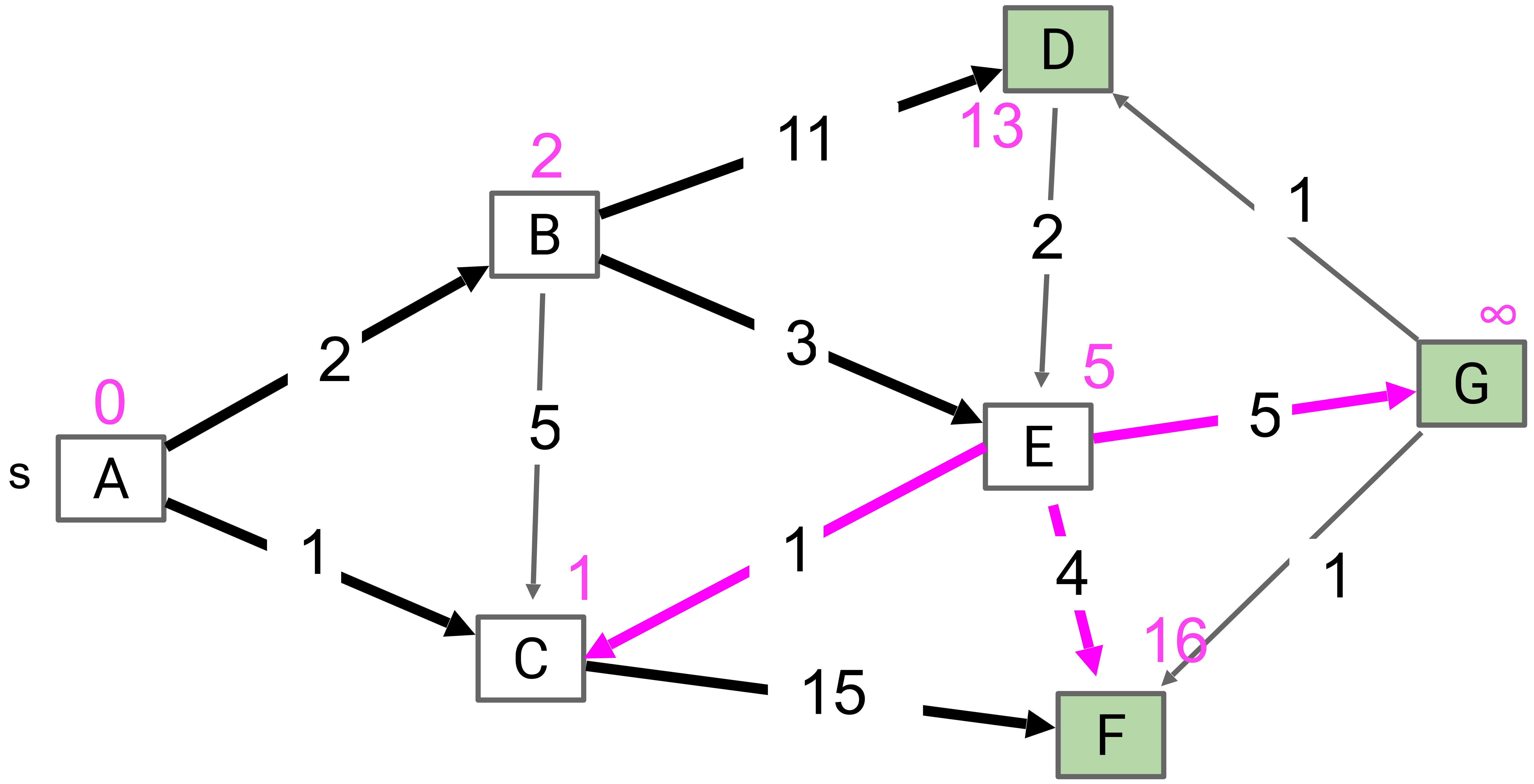
Which vertex is removed next?

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



Fringe: [(D: 13), (F: 16), (G: ∞)]

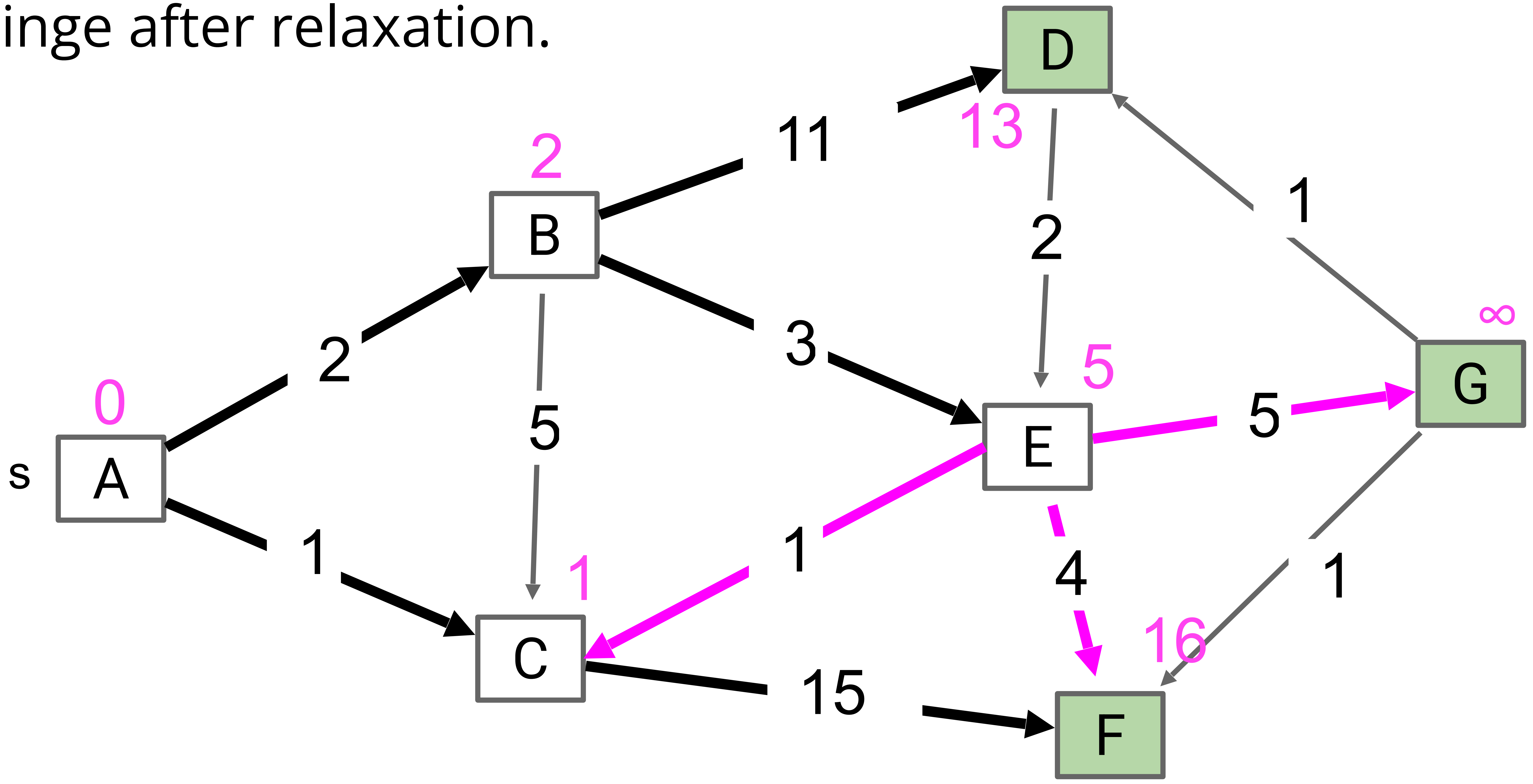
Worksheet time!

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

- Show distTo, edgeTo, and fringe after relaxation.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



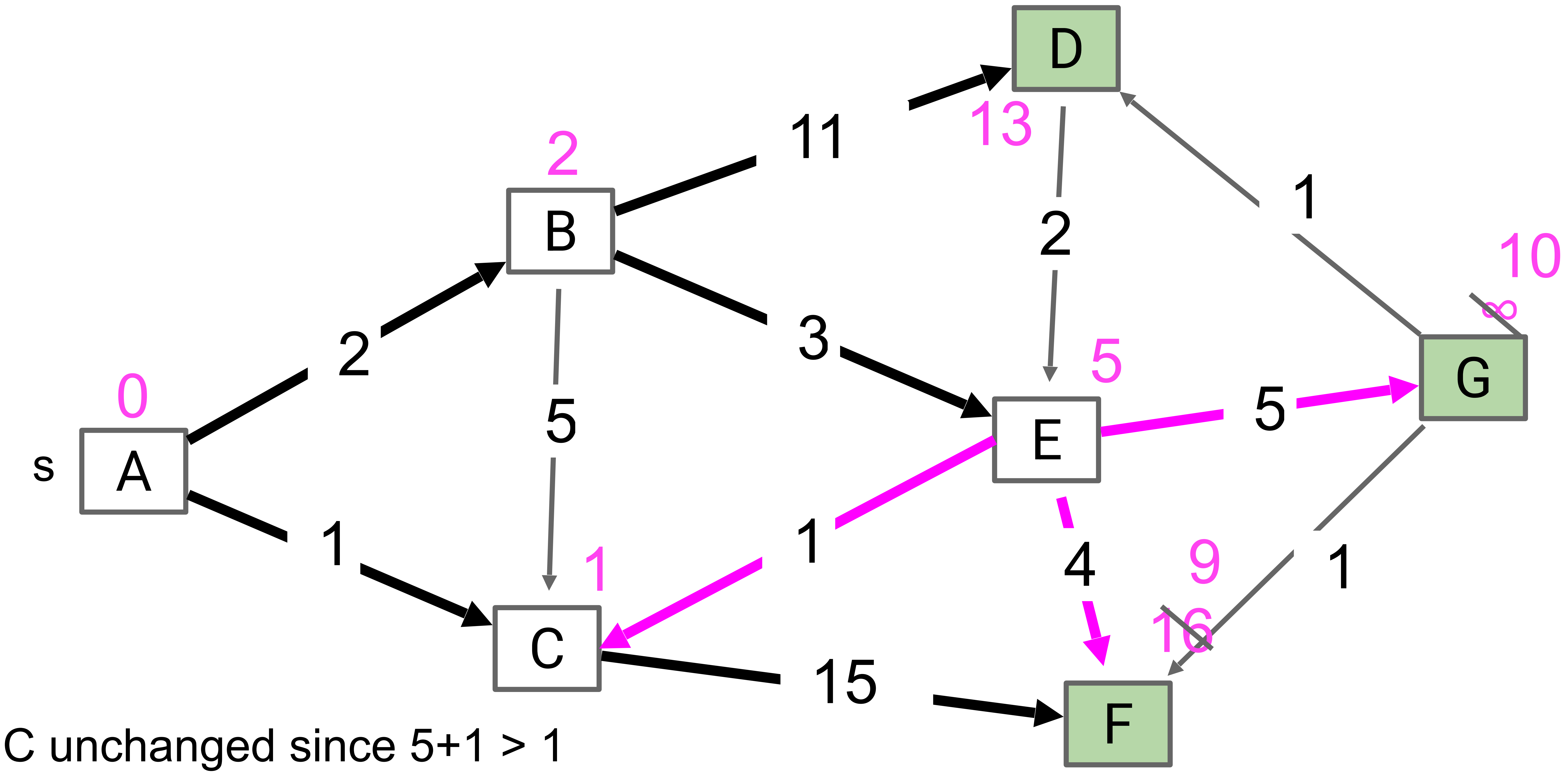
`Fringe: [(D: 13), (F: 16), (G: ∞)]`

Worksheet answers

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	9	E
G	10	E



Vertex C unchanged since 5+1 > 1

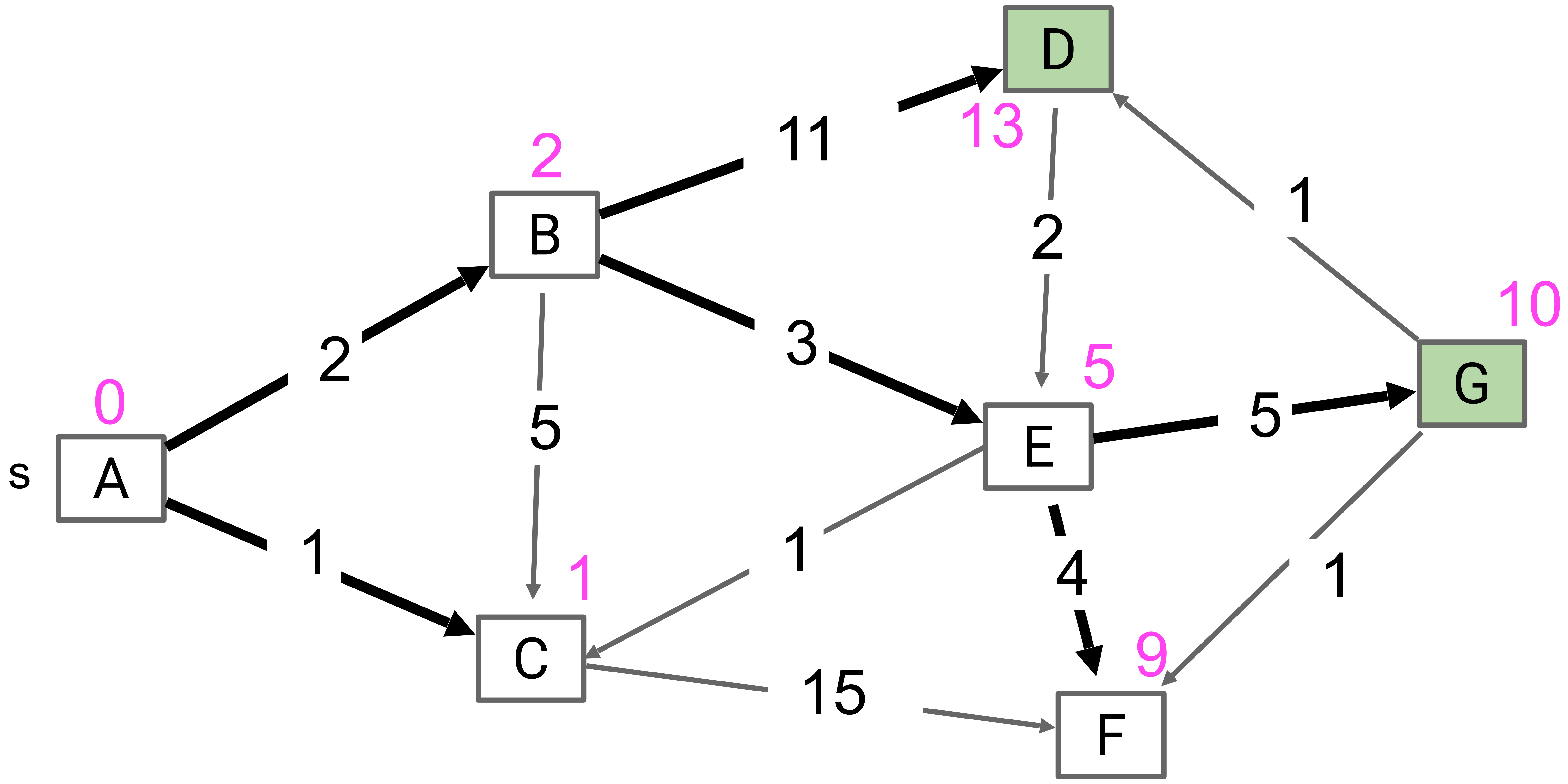
Fringe: [(F: 9), (G: 10), (D: 13)]

Dijkstra’s Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	9	E
G	10	E



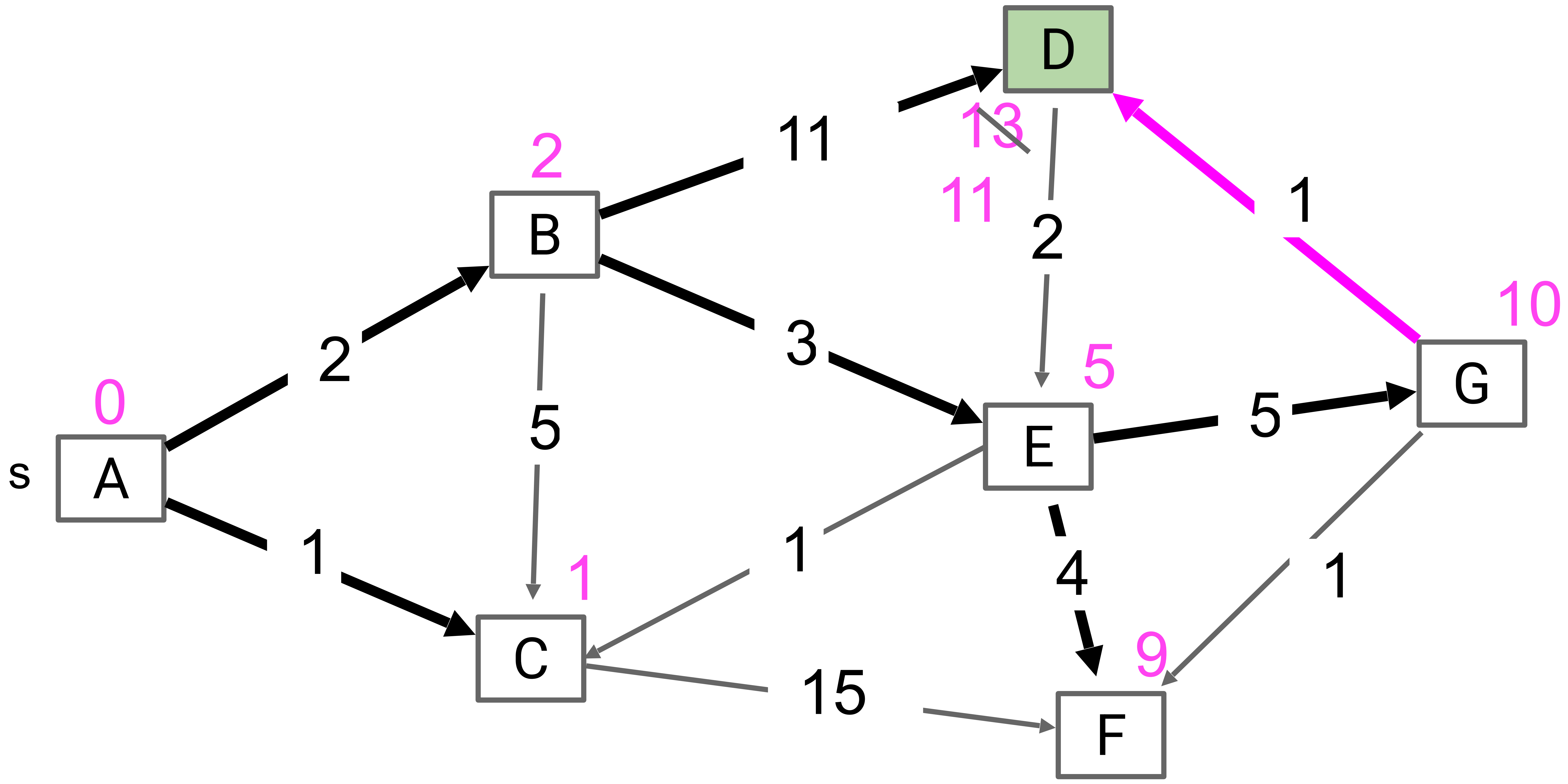
`Fringe: [(G: 10), (D: 13)]`

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



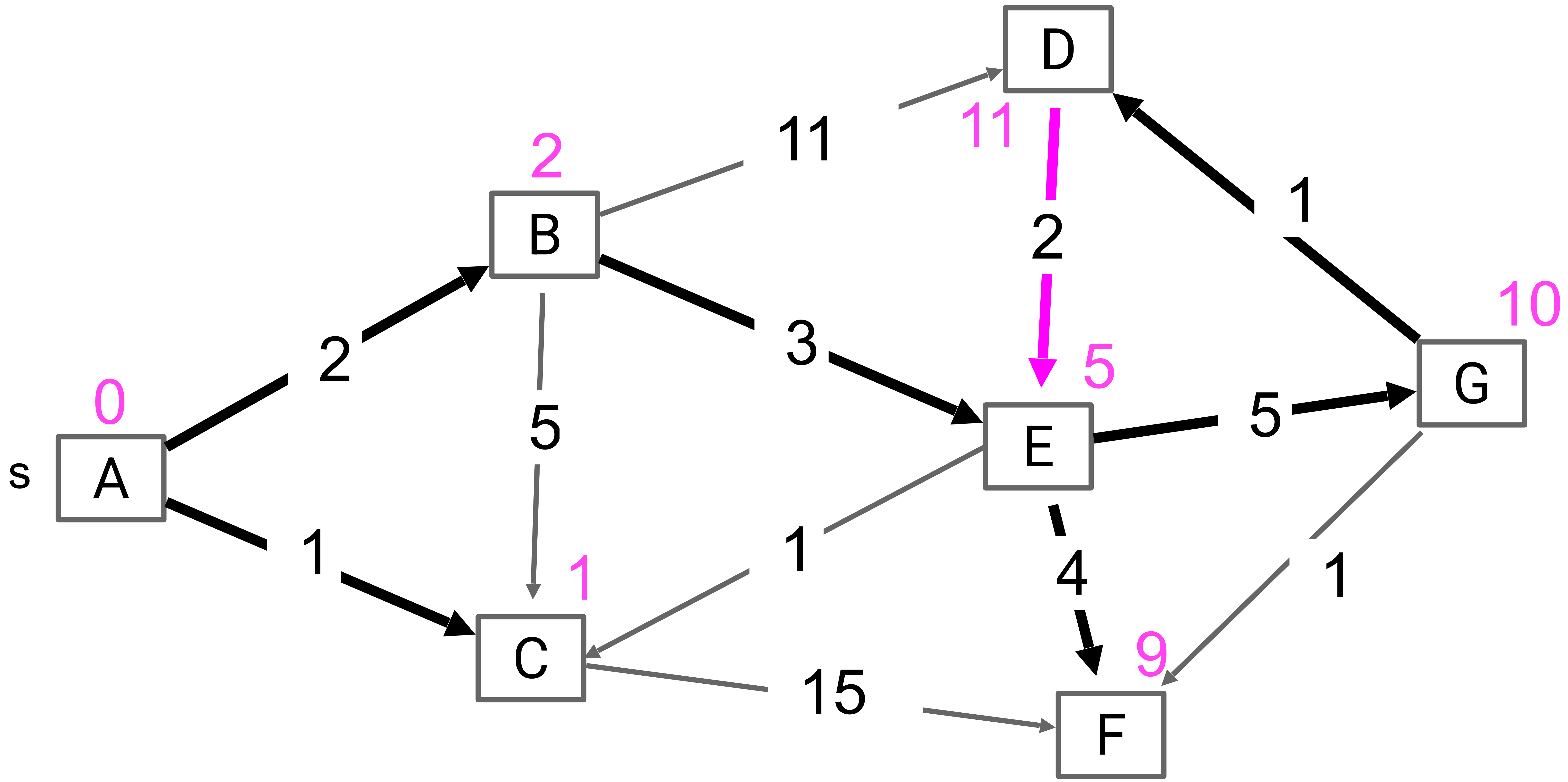
Fringe: [(D: 11)]

Dijkstra's Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



Fringe: [ ]

Vertex E unchanged since 11 + 2 > 5

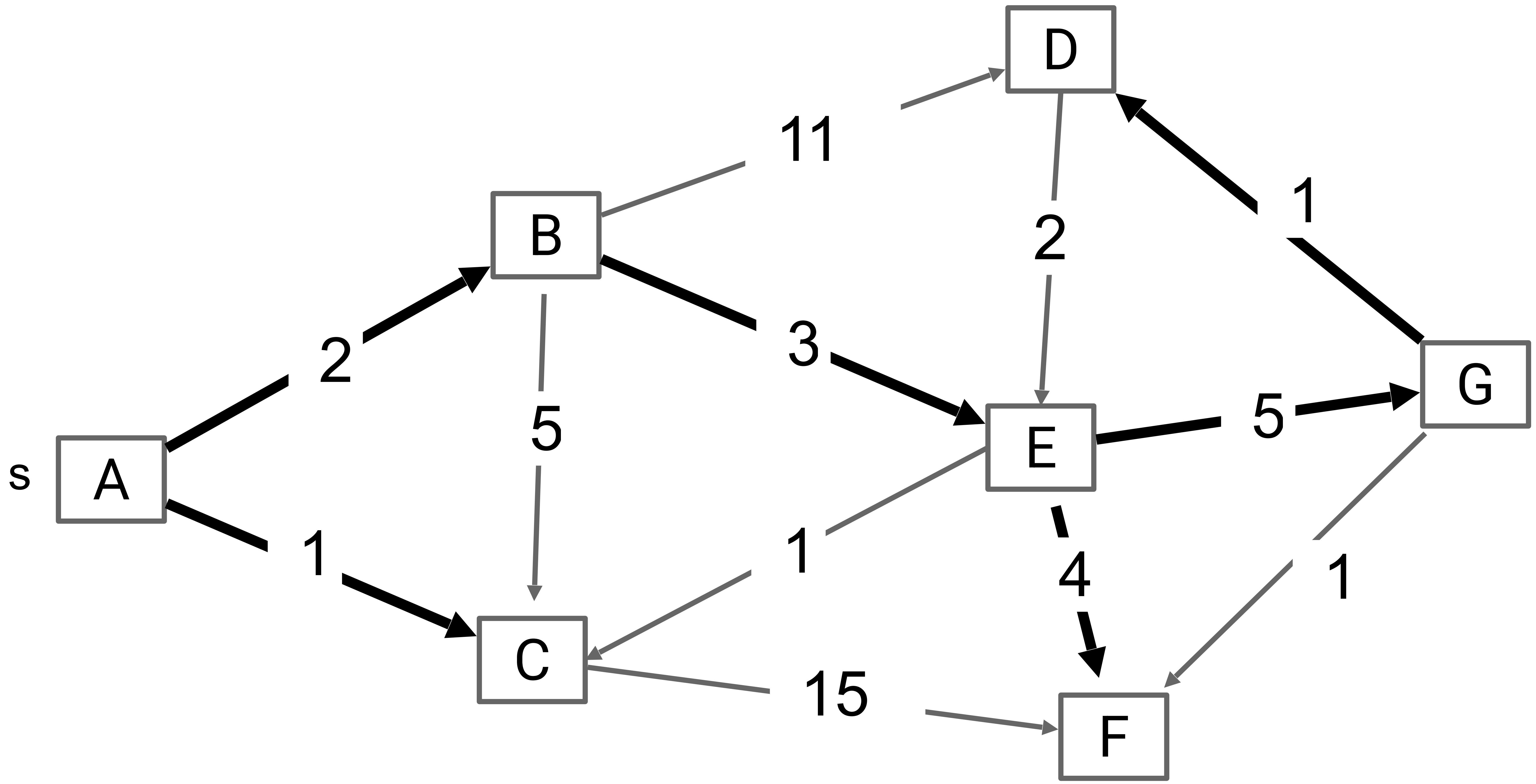
Note: If non-negative weights, **impossible for any inactive vertex** (white, not on fringe) **to be improved**!

Dijkstra’s Demo

Insert all vertices into fringe PQ (priority queue), storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

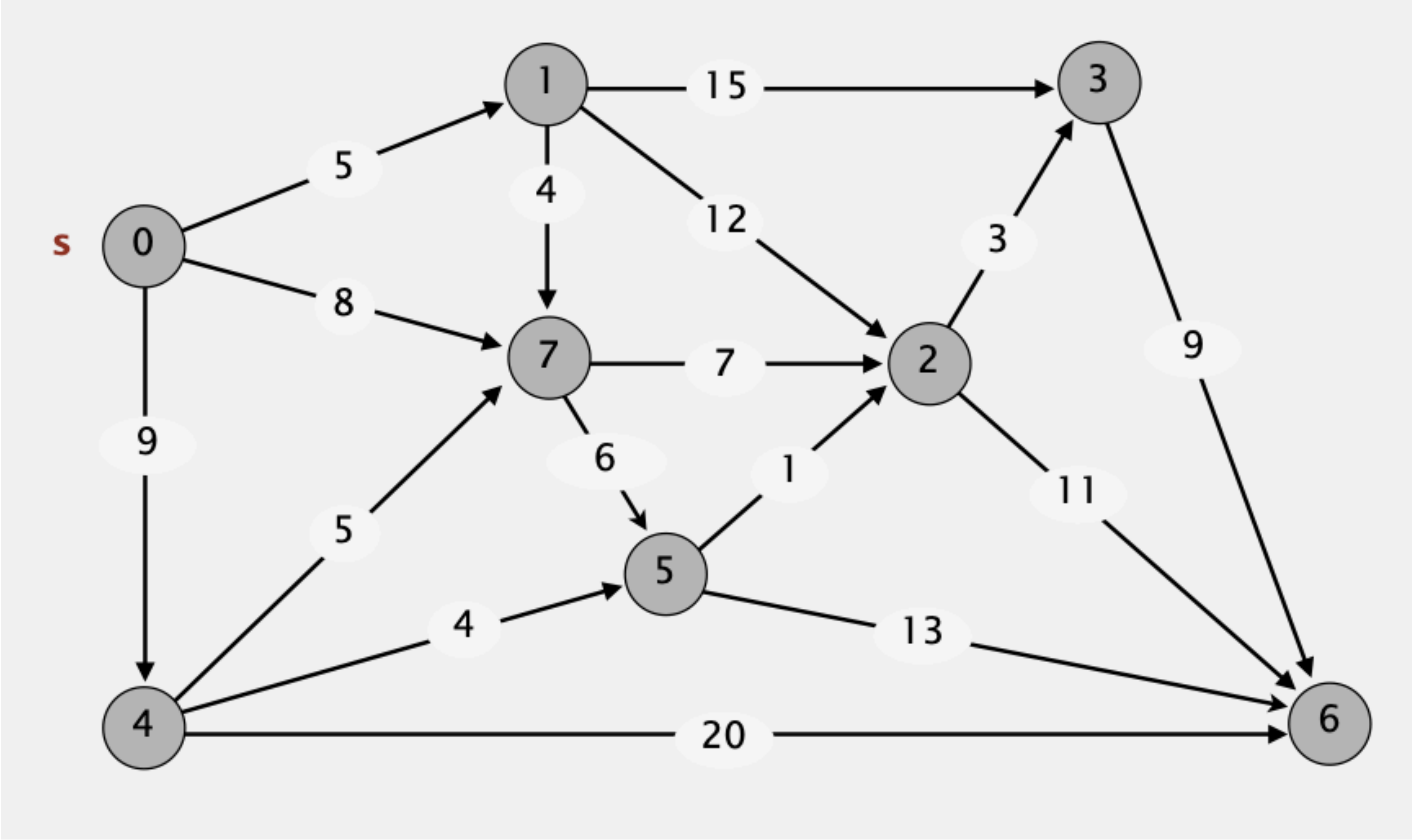
Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



Fringe: []

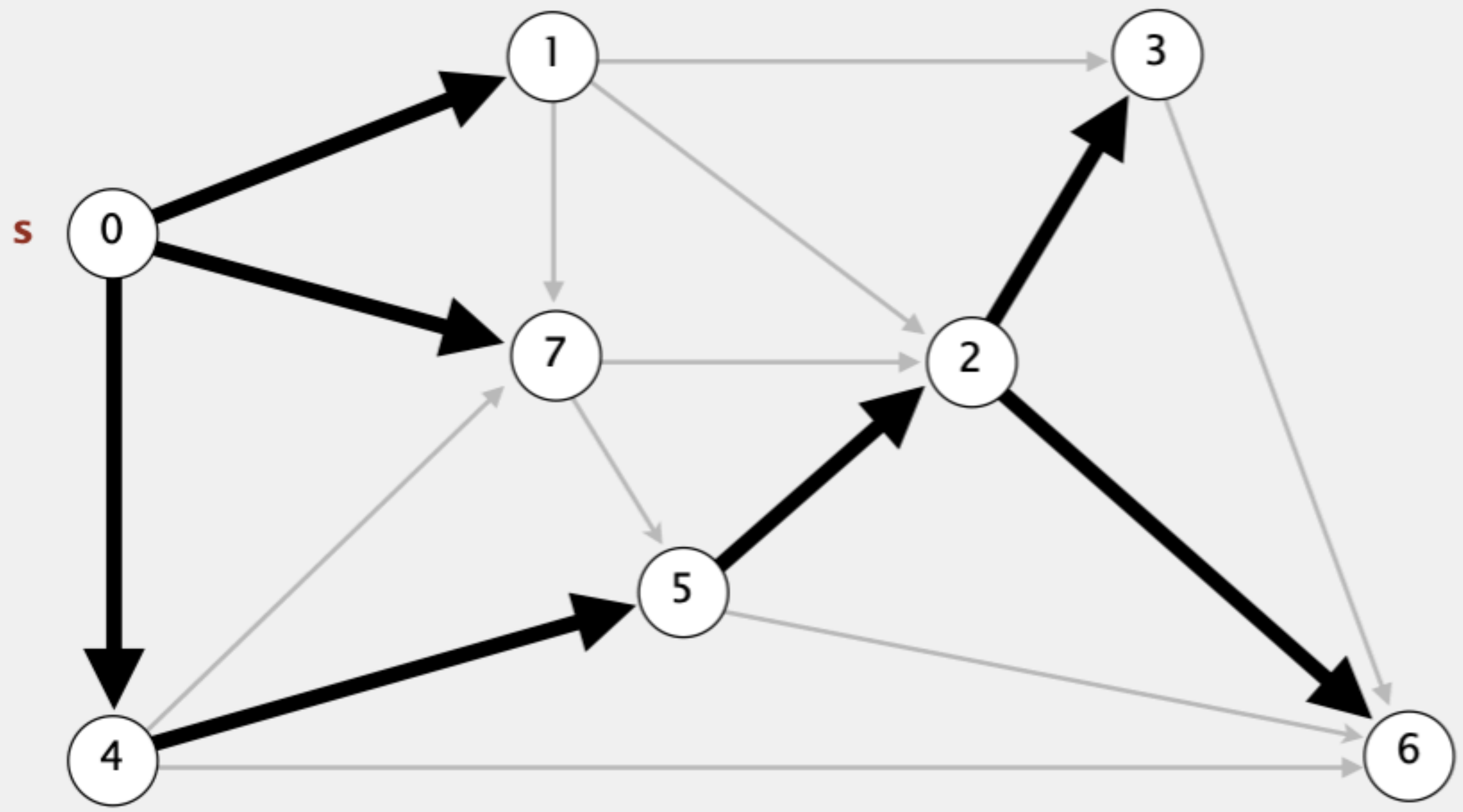
Worksheet time!

- Run Dijkstra's algorithm to generate the shortest path tree from s below.



Worksheet answers!

- Run Dijkstra’s algorithm to generate the shortest path tree from s below.
- For a full walkthrough, see the slides in the appendix



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Dijkstra’s Implementation

Dijkstra's Algorithm Pseudocode

Dijkstra's:

- PQ.add(source, 0)
- For other vertices v, PQ.add(v, infinity)
- While PQ is not empty:
  - p = PQ.removeSmallest()
  - Relax all edges from p

Relaxing an edge p → q with weight w:

- If distTo[p] + w < distTo[q]:
  - distTo[q] = distTo[p] + w
  - edgeTo[q] = p
  - PQ.changePriority(q, distTo[q])

Key invariants:

- edgeTo[v] is the best known predecessor of v.
- distTo[v] is the best known total distance from source to v.
- PQ contains all unvisited vertices in order of distTo.

Important properties:

- Always visits vertices in order of total distance from source.
- Relaxation always fails on edges to already visited vertices.

Framework for shortest-paths algorithm

```
public class DijkstraSP {
    private double[] distTo;          // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;    // edgeTo[v] = last edge on shortest s->v path
    private IndexMinPQ<Double> pq;    // priority queue of vertices


    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        // relax vertices in order of distance from s
        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    // relax edge e and update pq if changed
    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
```

Dijkstra's Algorithm Runtime

Priority Queue operation count, assuming min-binary heap based PQ:

- add: max V times, each costing O(log V) time.
- removeSmallest: max V times, each costing O(log V) time.
- changePriority: max E times, each costing O(log V) time.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just **O(E log V)** for a connected graph.

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ removeSmallest	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

Lecture 23 wrap-up

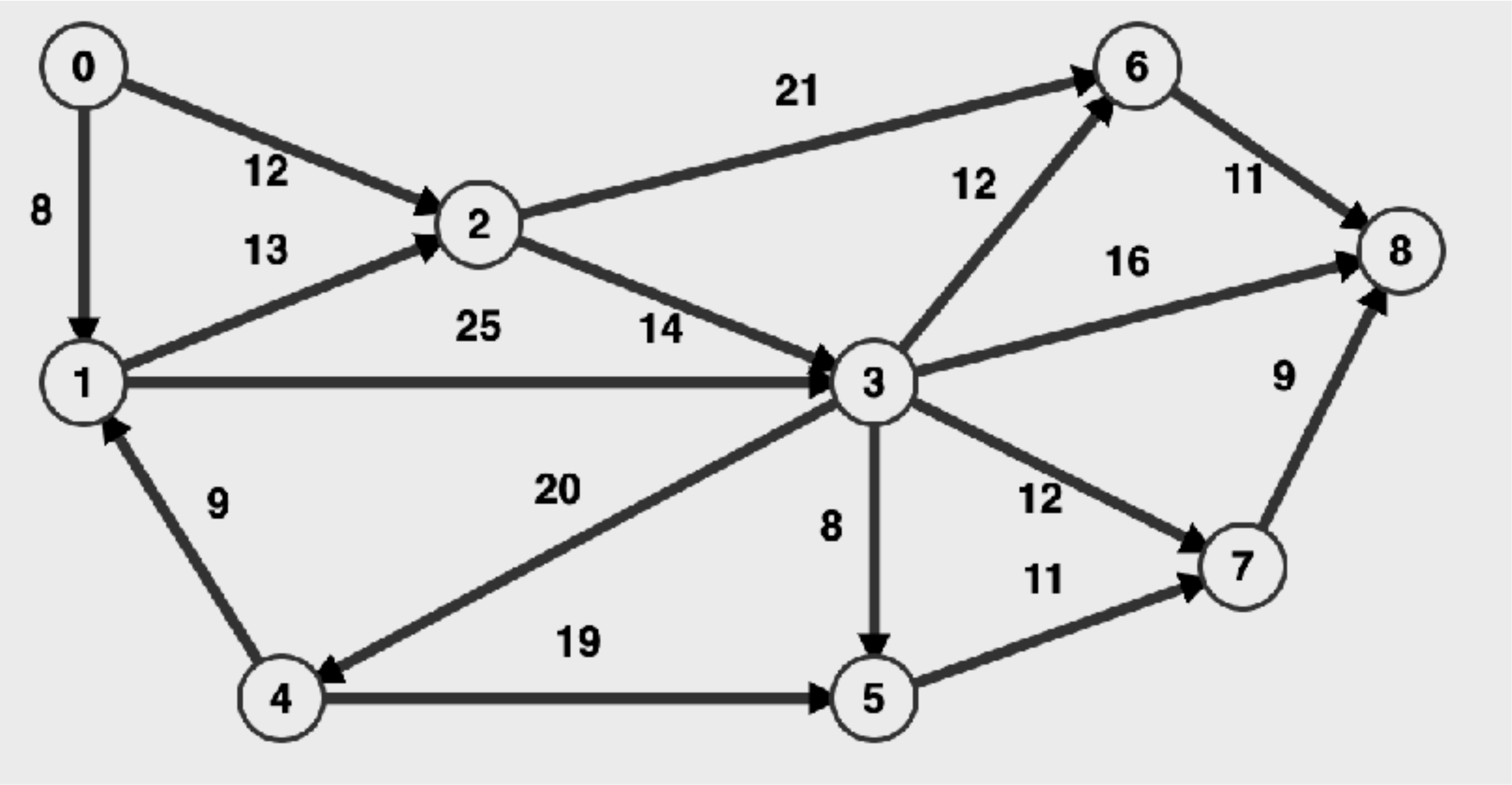
- HW9: Transplant Manager due 11:59pm
- Last(!) HW (10, text generator), quiz, & graded lab this week
- Next week: Zoom class
  - Tues: Course evals 2:45-3pm, 3-4pm Careers panel (Google, CS PhD, video game company)
  - Thurs: Final project pt 1 check-ins (more in lab tomorrow!), sign up for 10 min slot

Resources

- Recommended Textbook: Chapter 4.4 (Pages 638-676)
- Website: https://algs4.cs.princeton.edu/44sp/
- Visualization: https://visualgo.net/en/sssp
- Practice problems behind this slide

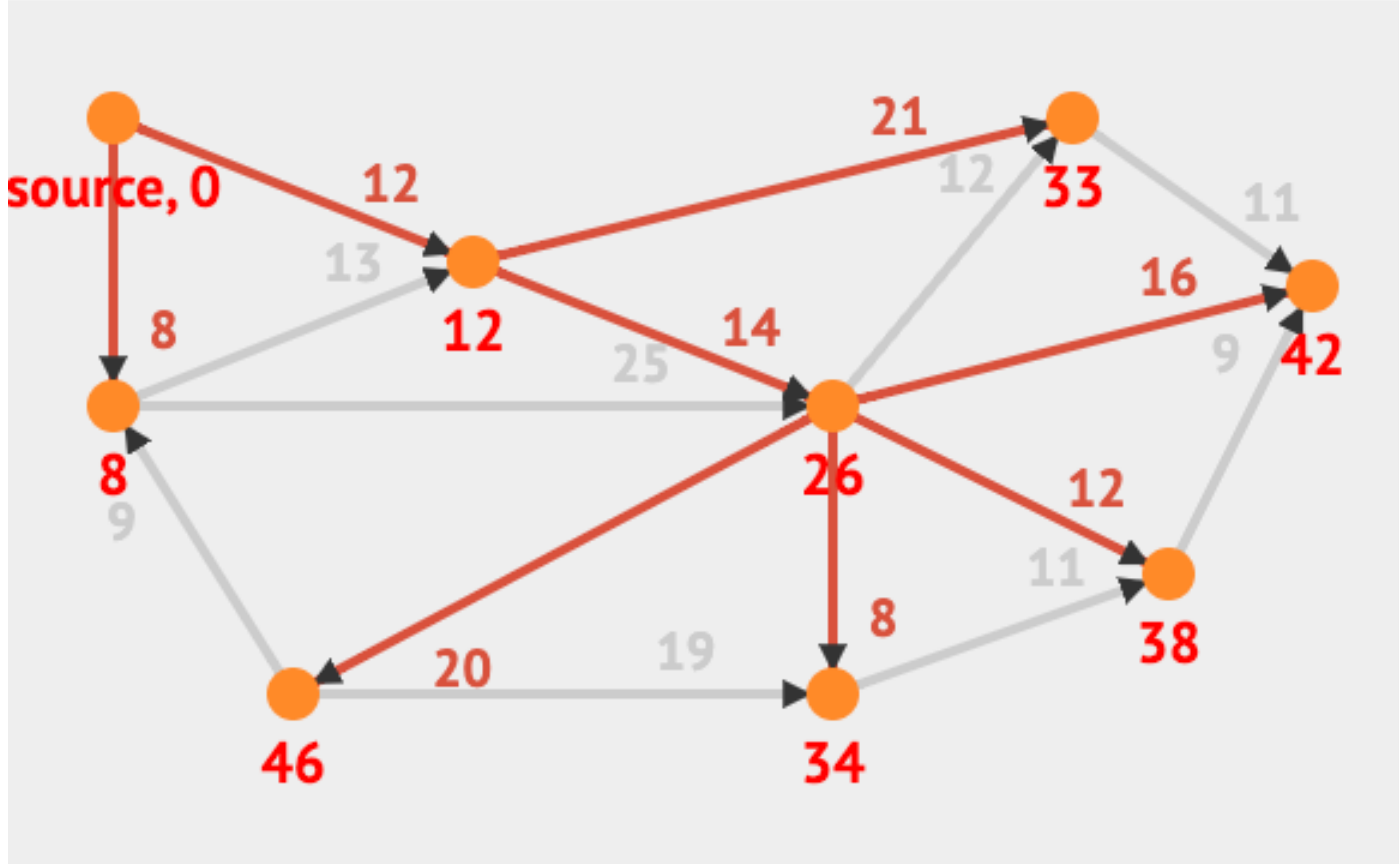
Problem 1

- Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.



Answer 1

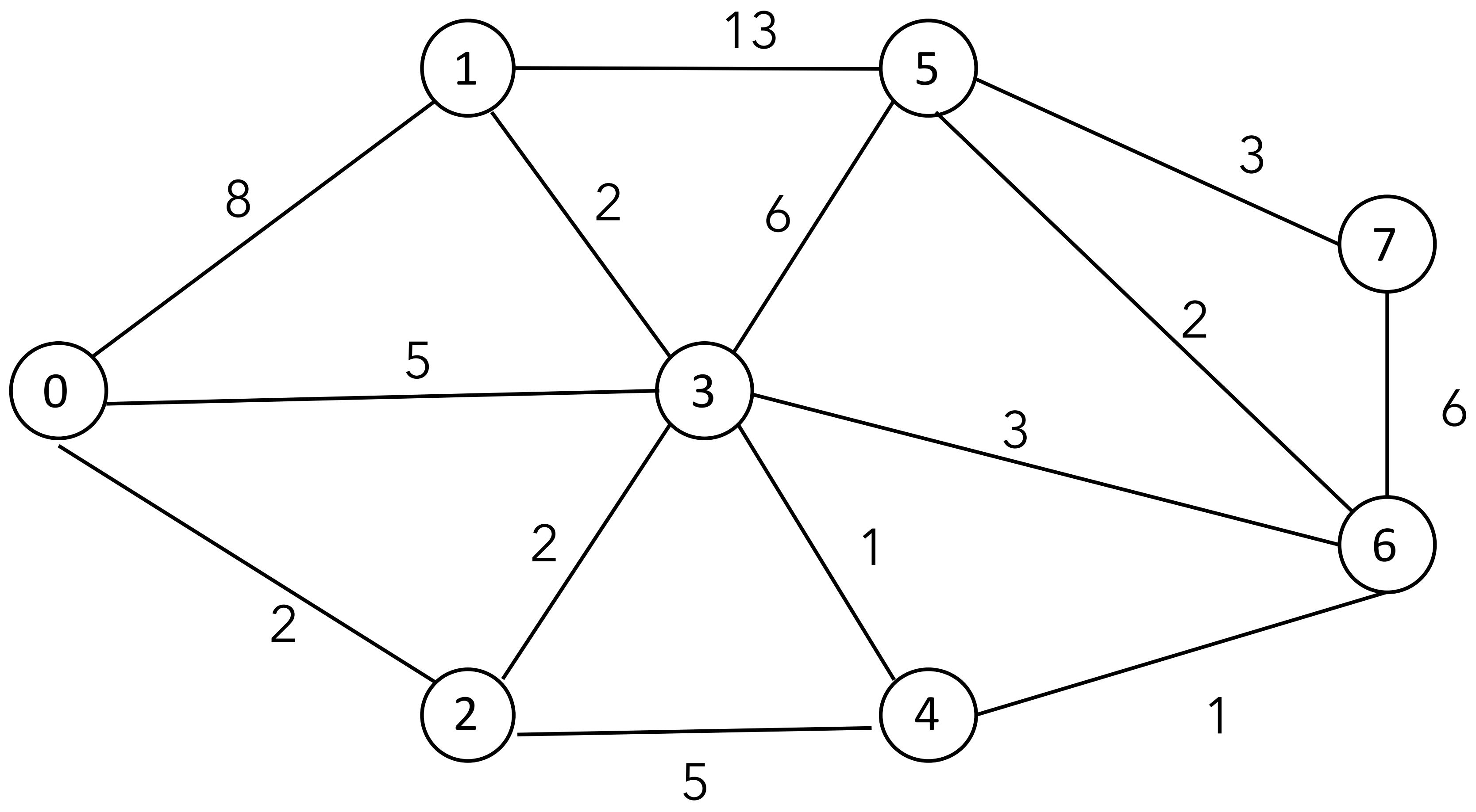
- Run Dijkstra’s algorithm on the following graph with 0 being the starting vertex.



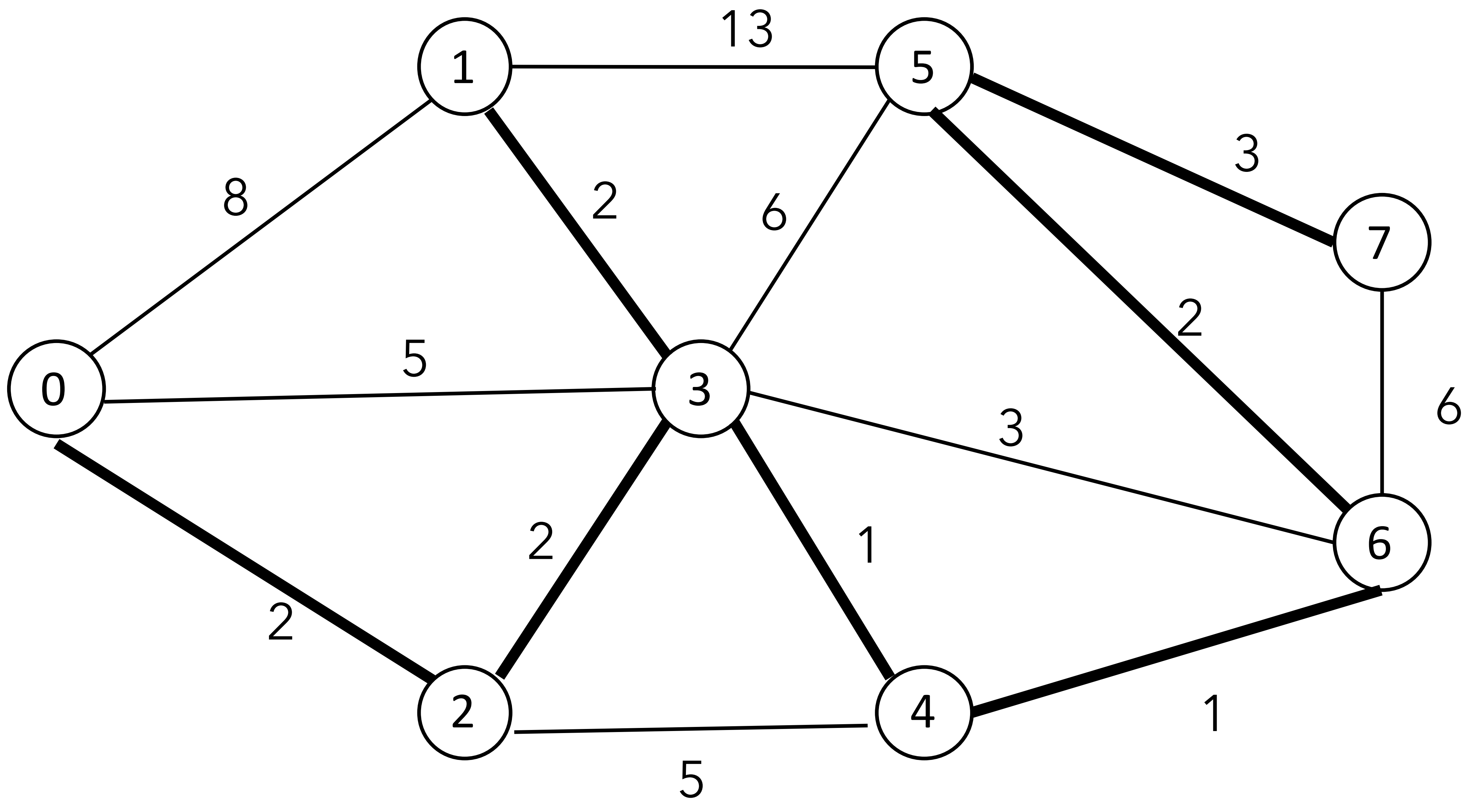
v	distTo[]	edgeTo[]
0	0	-
1	8	0->1
2	12	0->2
3	26	2->3
4	46	3->4
5	34	3->5
6	33	3->6
7	38	3->7
8	42	3->8

Problem 2

Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.



Answer 2



v	distTo[]	edgeTo[]
0	**0**	**-**
1	6	3->1
2	2	0->2
3	4	2->3
4	5	3->4
5	8	6->5
6	6	4->6
7	11	5->7

Problem 3

Dijkstra’s algorithm is guaranteed to be optimal so long as there are no negative edges. Sketch a proof by induction proving why.

- Hint: The proof relies on the property that relaxation always fails on edges to visited (white) vertices.

Answer 3

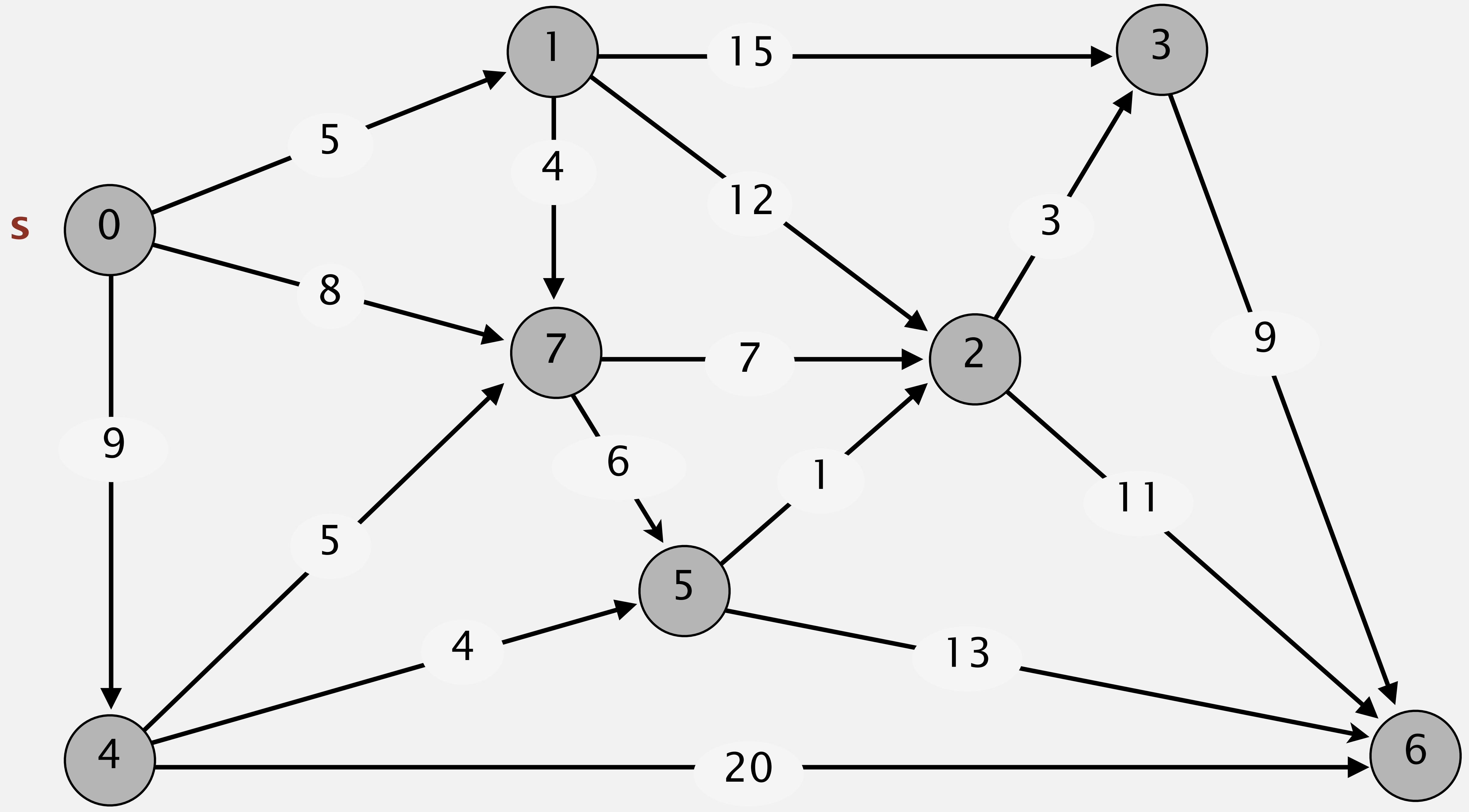
Proof sketch: Assume all edges have non-negative weights.

- At start, distTo[source] = 0, which is optimal.
- After relaxing all edges from source, let vertex v1 be the vertex with minimum weight, i.e. that is closest to the source. Claim: distTo[v1] is optimal, and thus future relaxations will fail. Why?
  - distTo[p] ≥ distTo[v1] for all p, therefore
  - distTo[p] + w ≥ distTo[v1]
- Can use induction to prove that this holds for all vertices after dequeuing.

Worksheet #3 full walkthrough

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



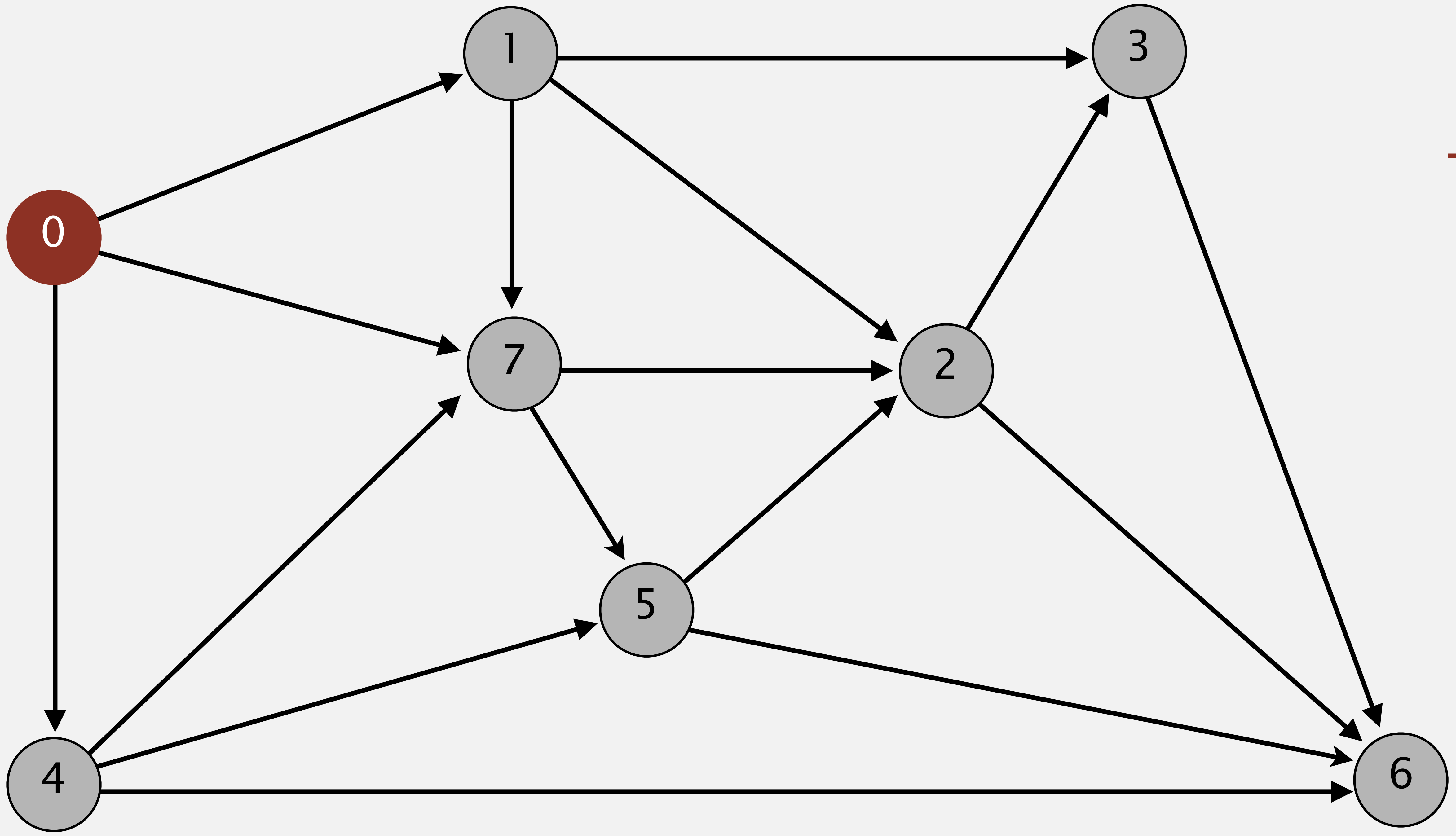
an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



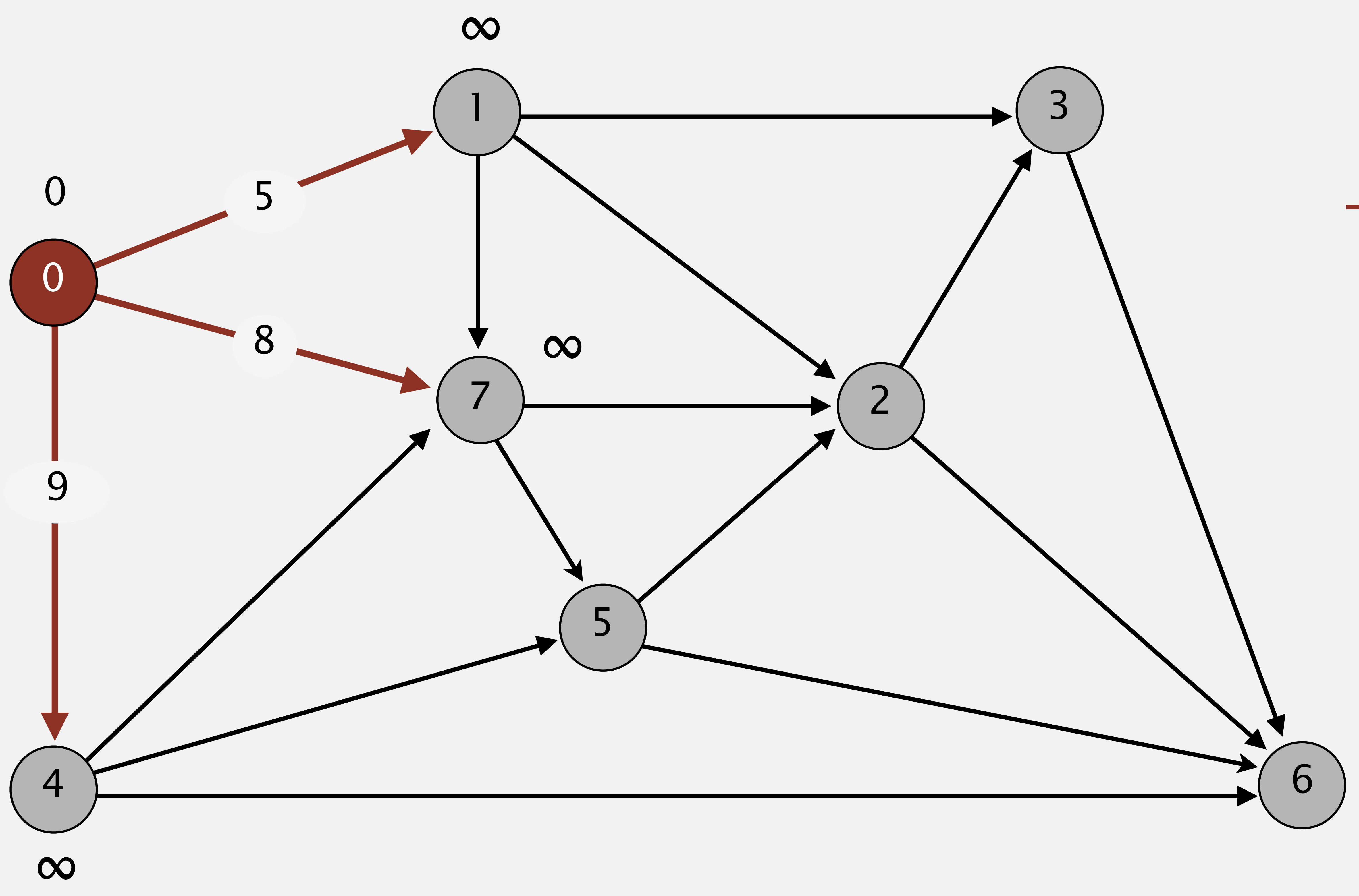
v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

choose source vertex 0

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



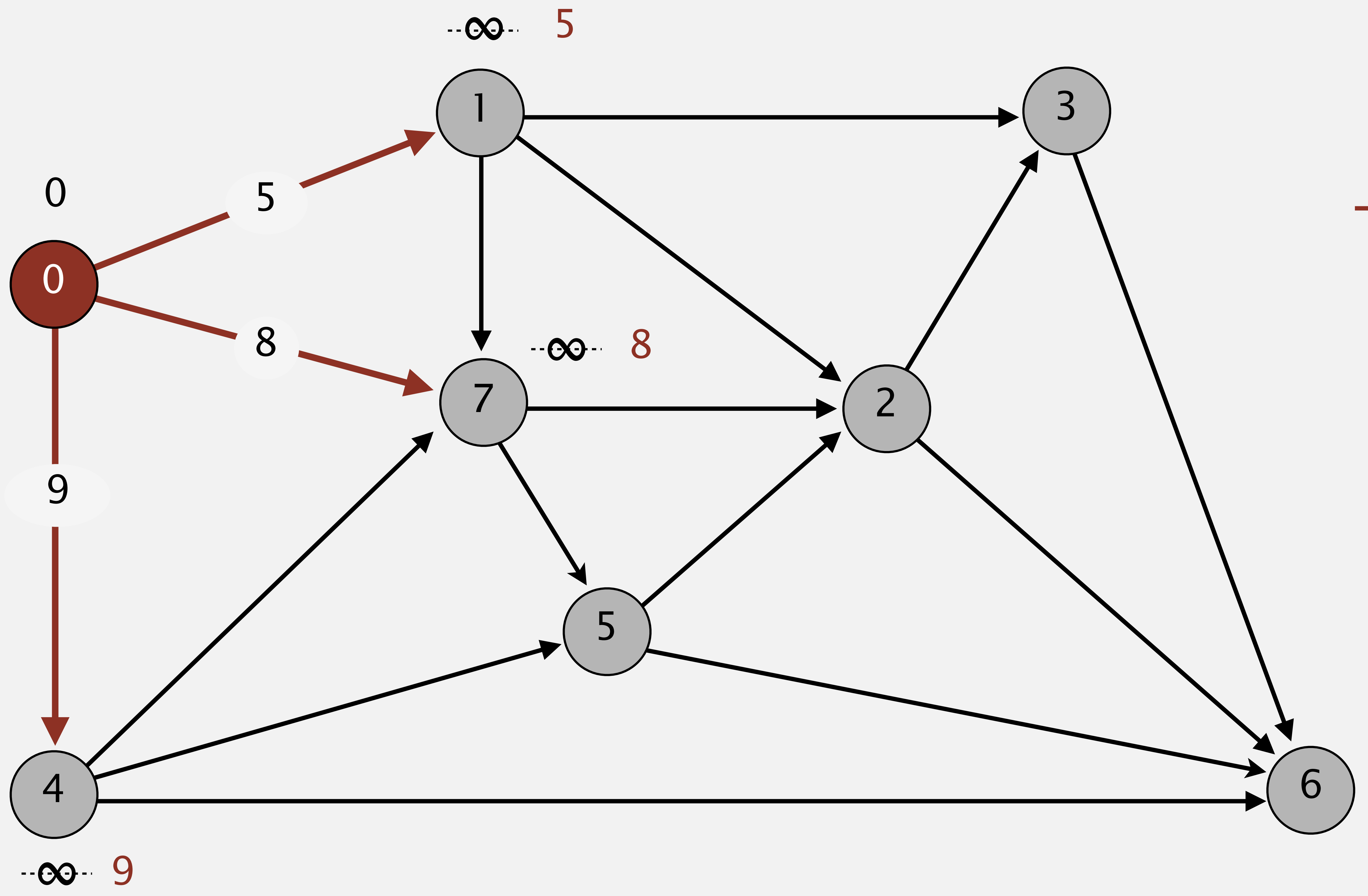
v	distTo[]	edgeTo[]
→ 0	0.0	-
1		
2		
3		
4		
5		
6		
7		

relax all edges adjacent from 0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



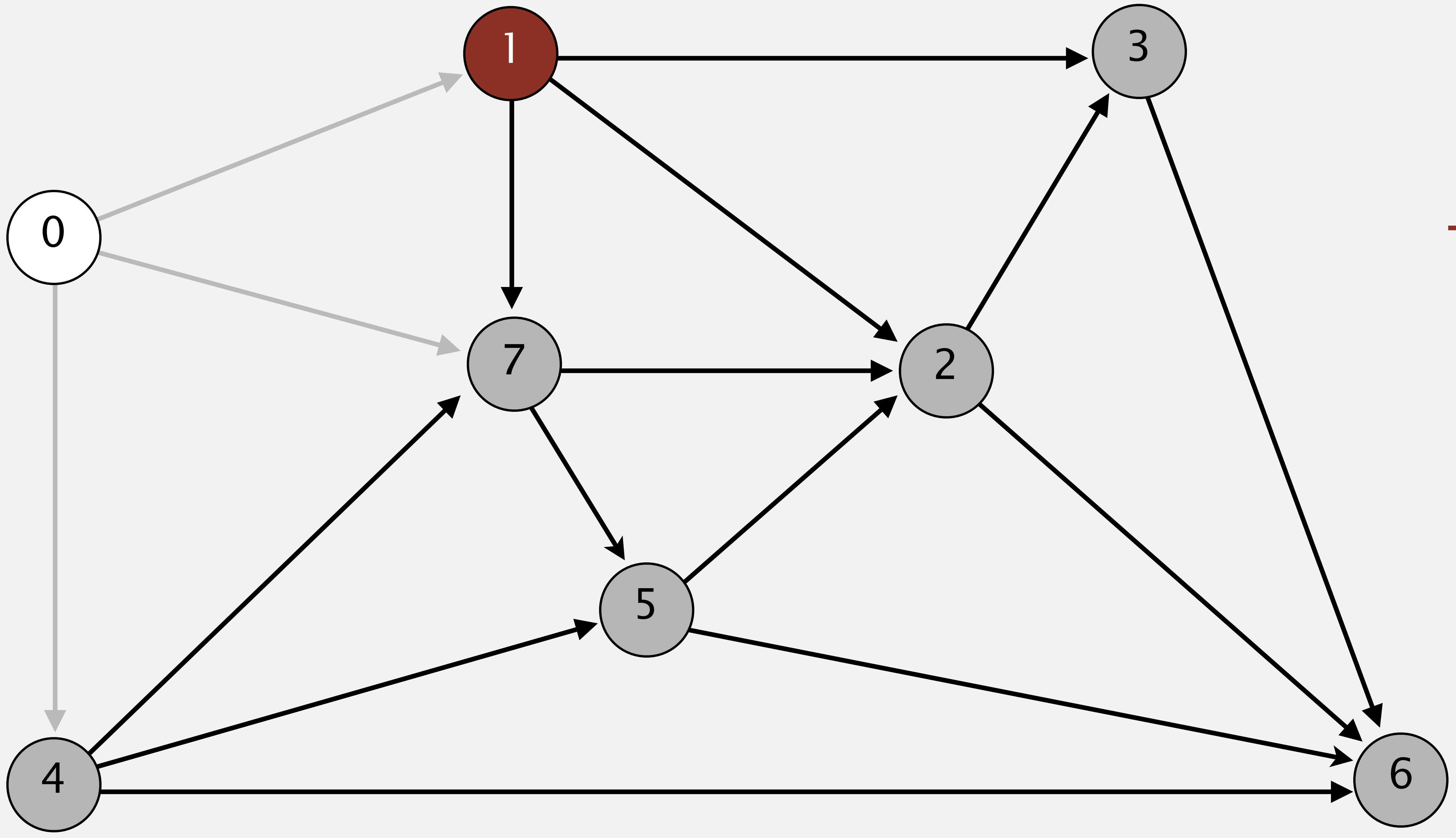
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

relax all edges adjacent from 0

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



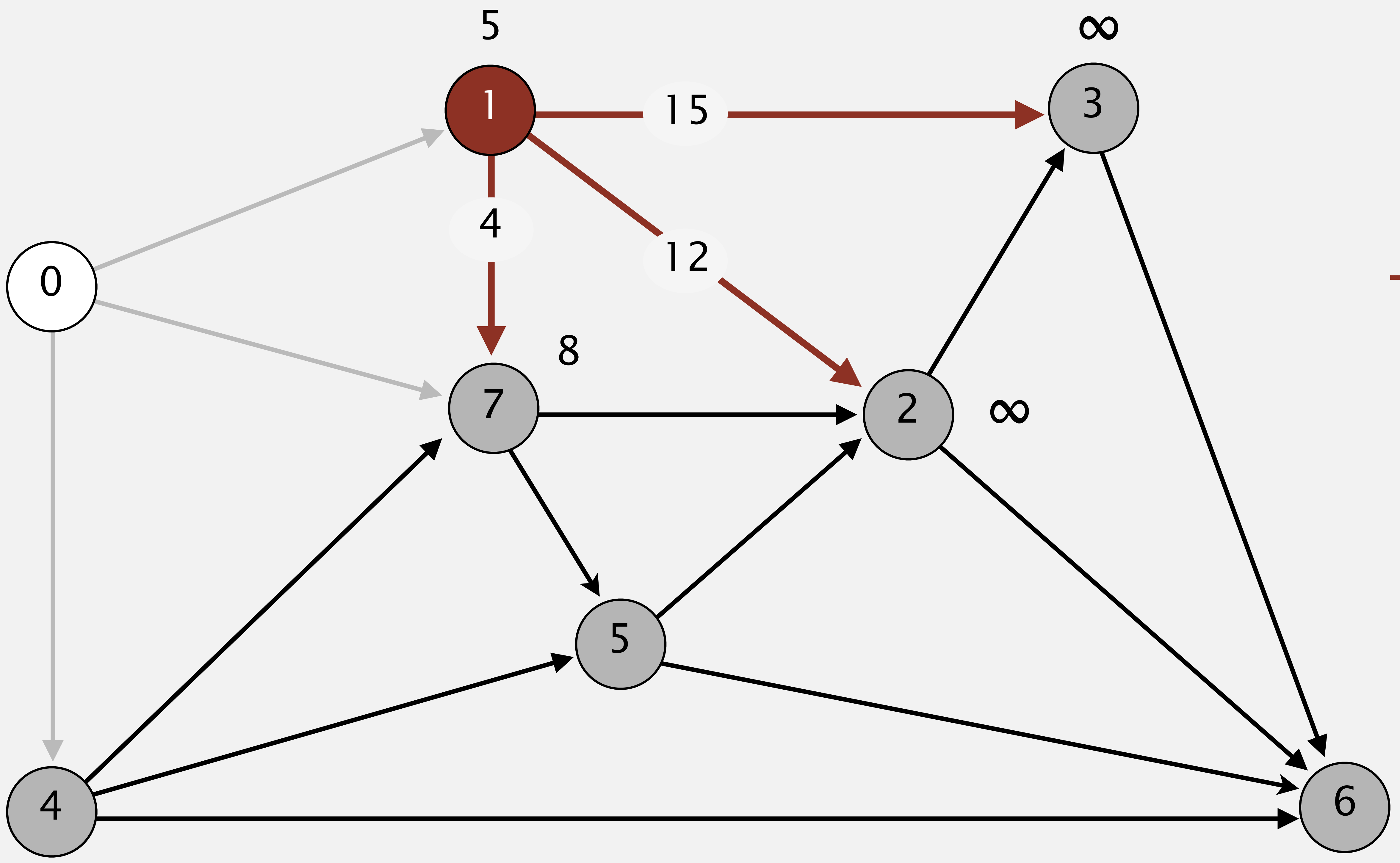
	v	distTo[]	edgeTo[]
	0	0.0	-
→	1	5.0	0→1
	2		
	3		
	4	9.0	0→4
	5		
	6		
	7	8.0	0→7

choose vertex 1

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



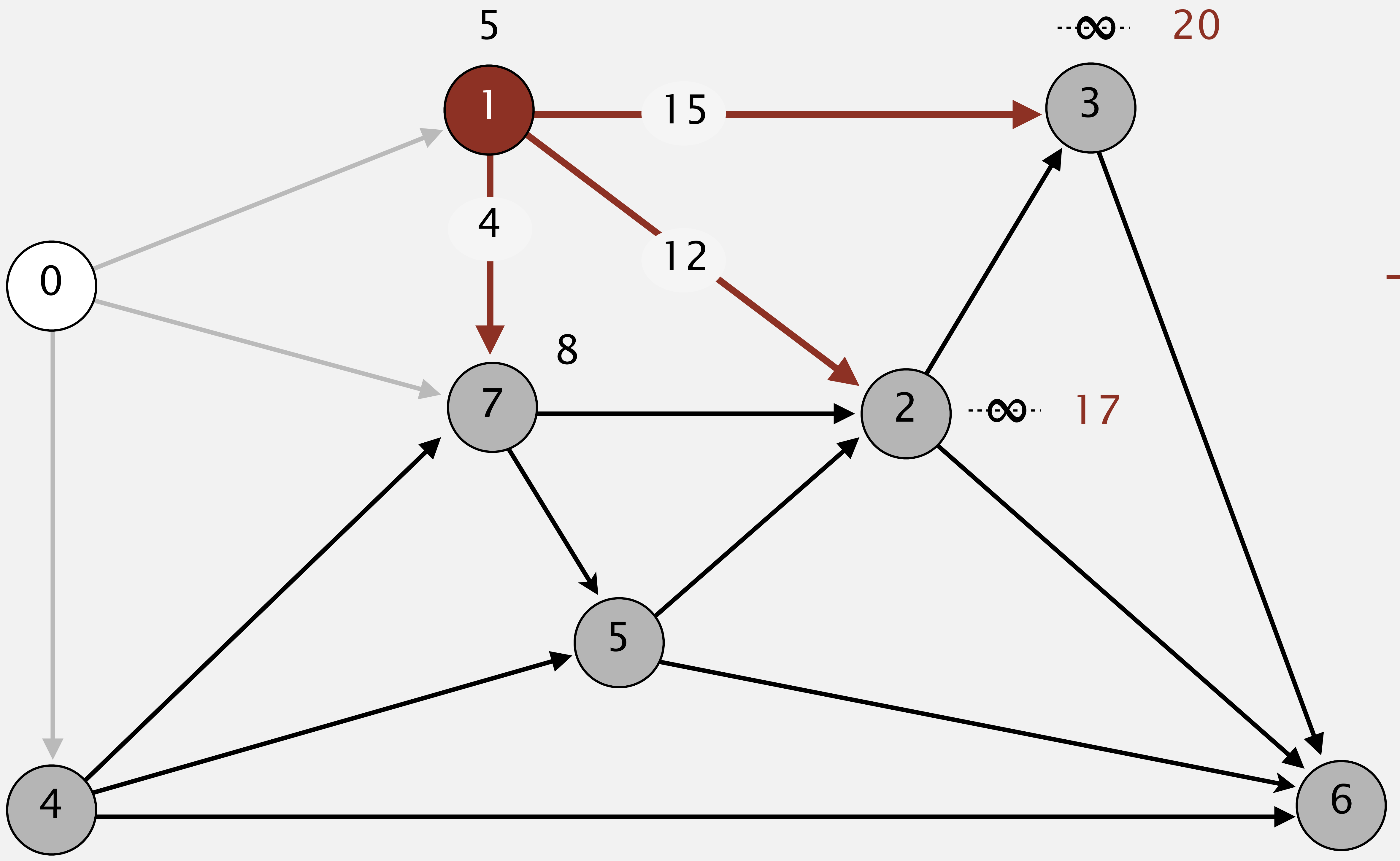
v	distTo[]	edgeTo[]
0	0.0	-
→ 1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

relax all edges adjacent from 1

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



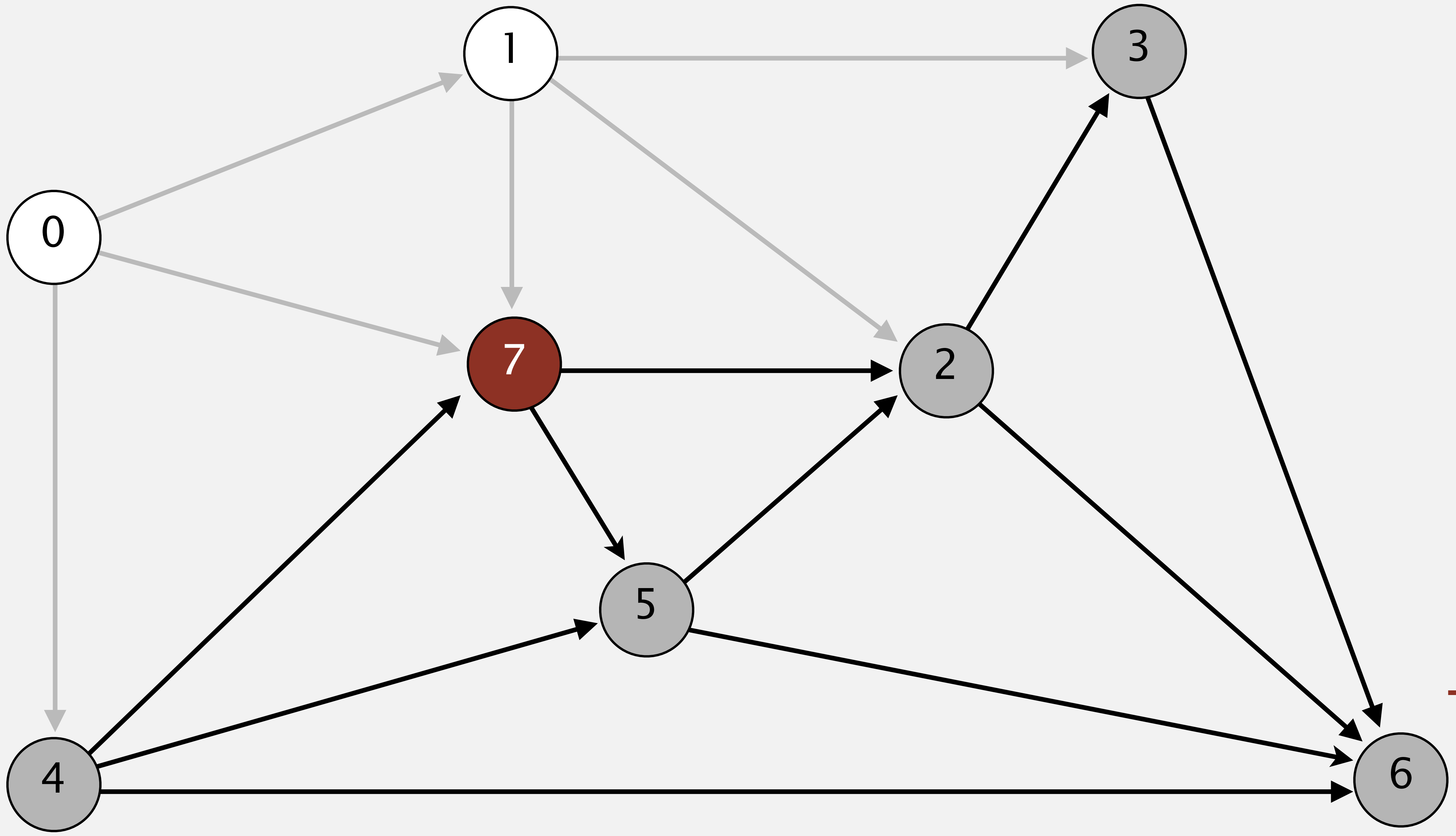
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0 ✔	0→7

relax all edges adjacent from 1

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



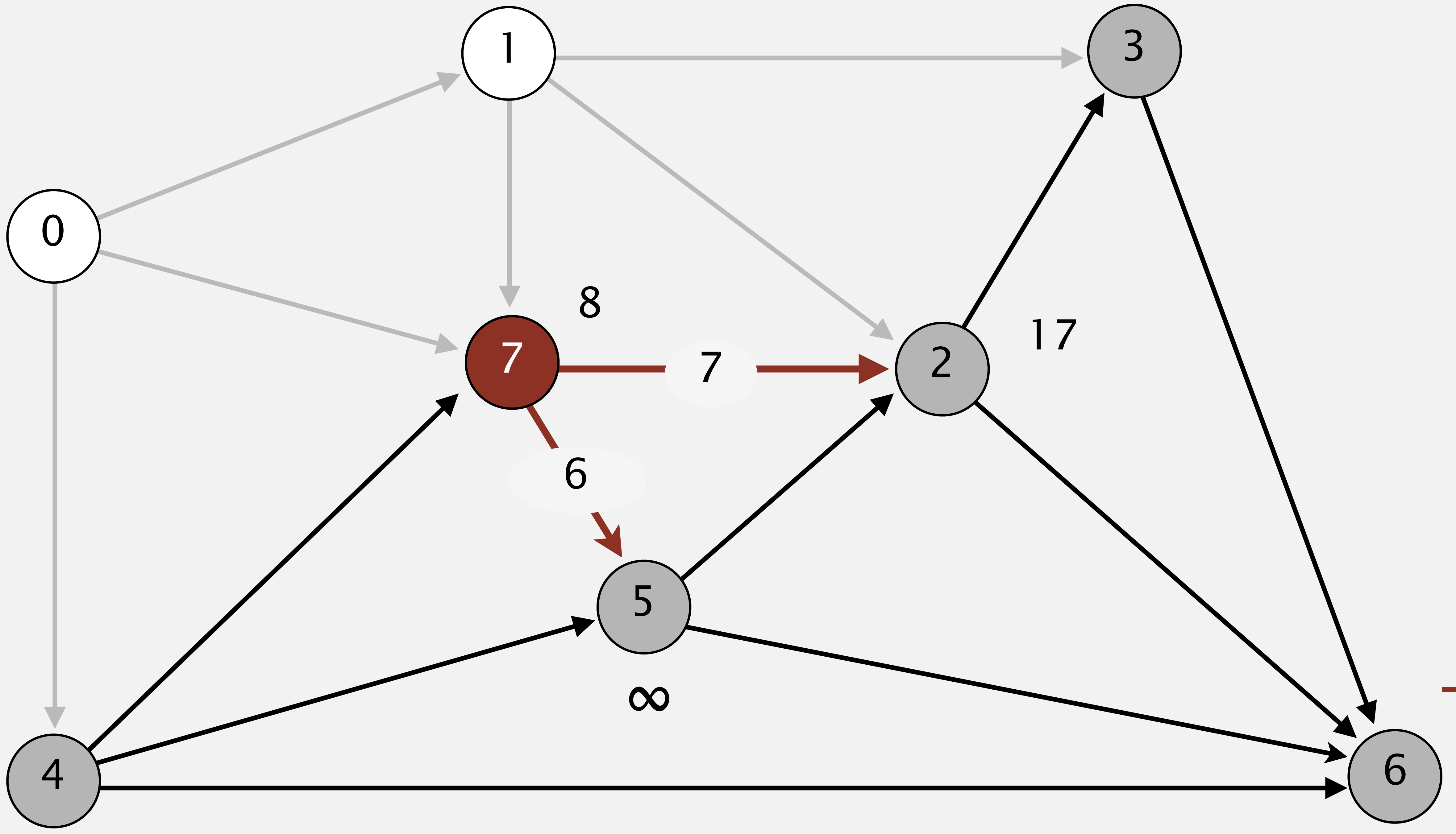
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
→ 7	8.0	0→7

choose vertex 7

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



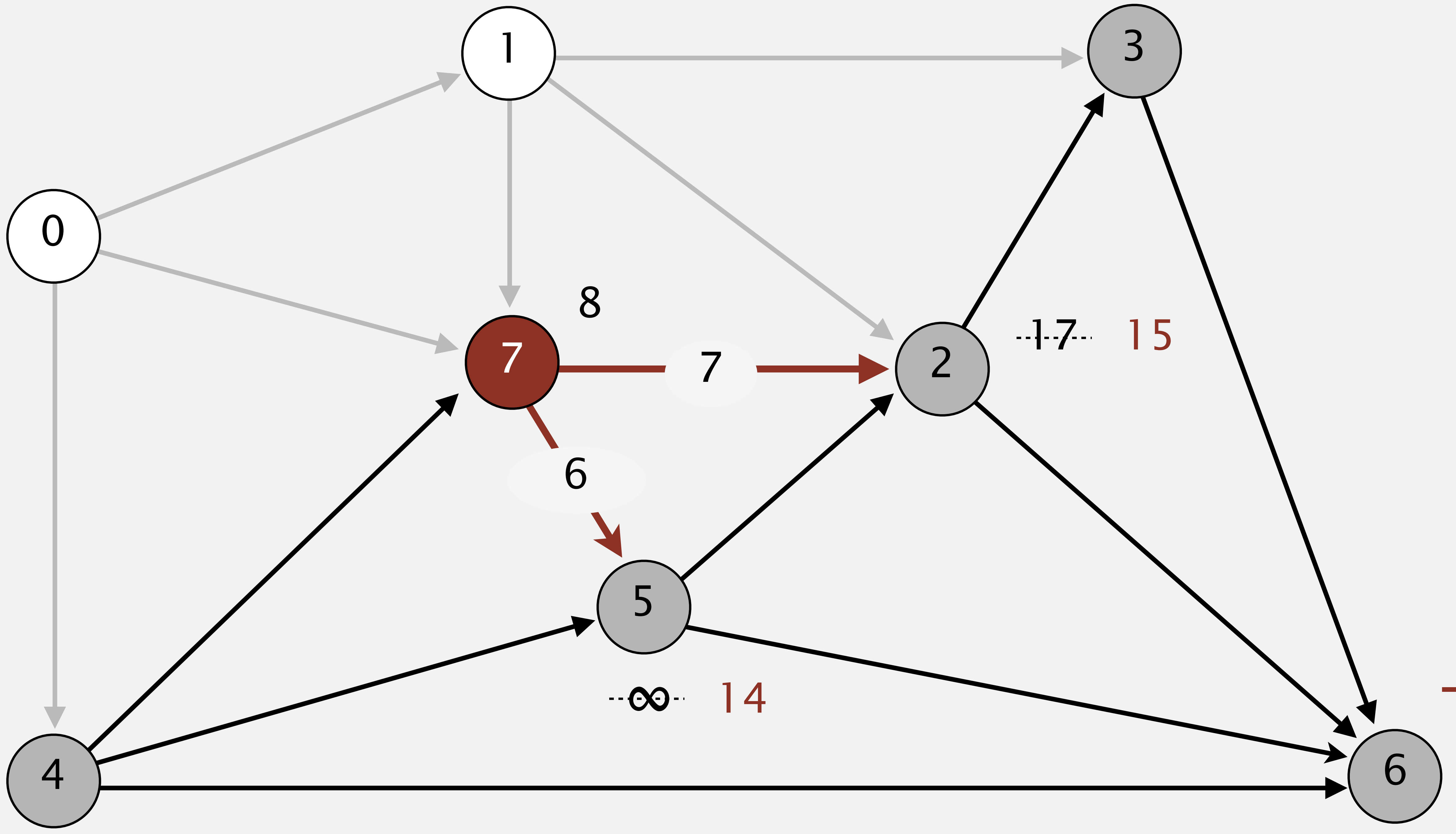
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
→ 7	8.0	0→7

relax all edges adjacent from 7

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



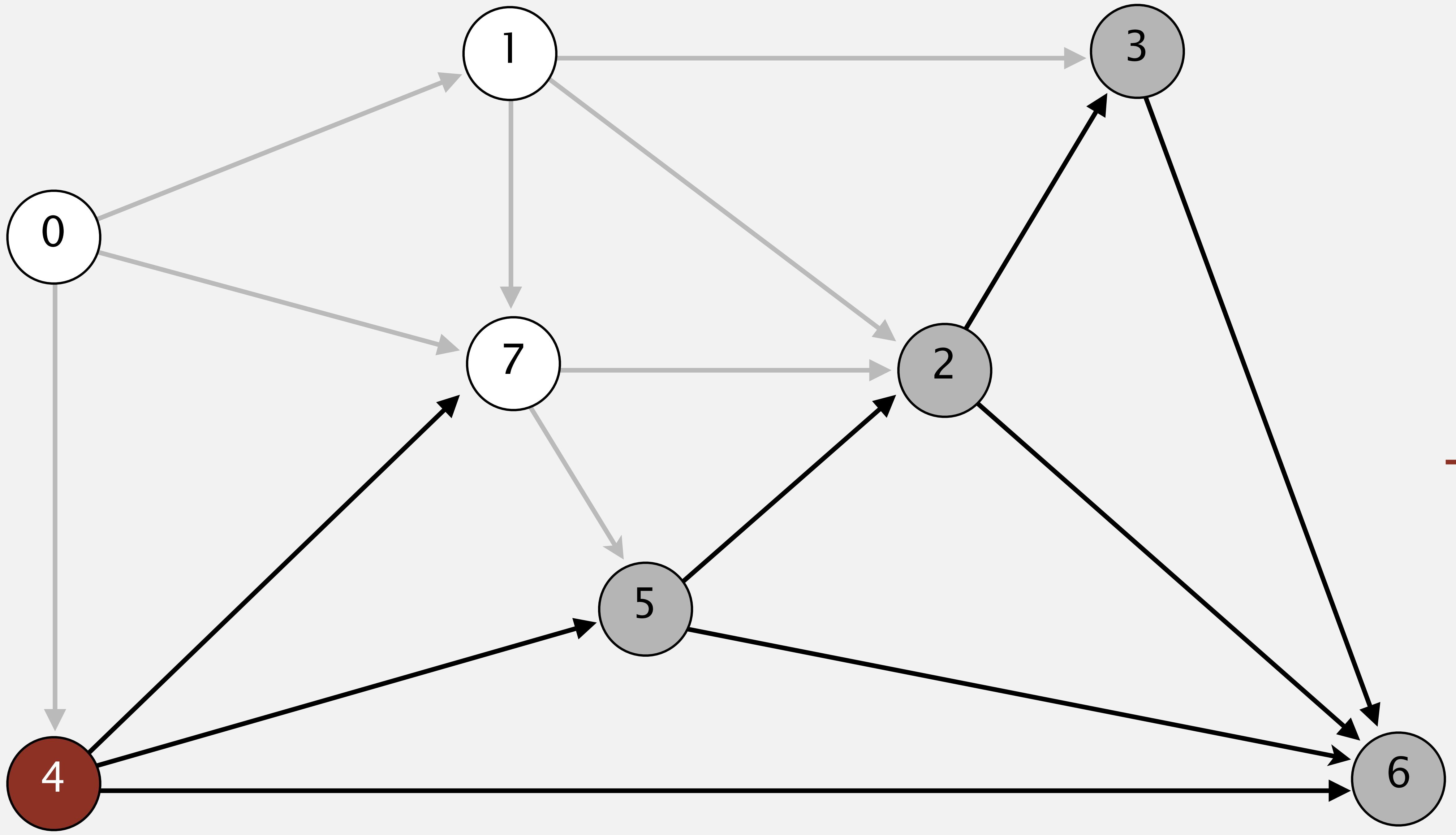
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

relax all edges adjacent from 7

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



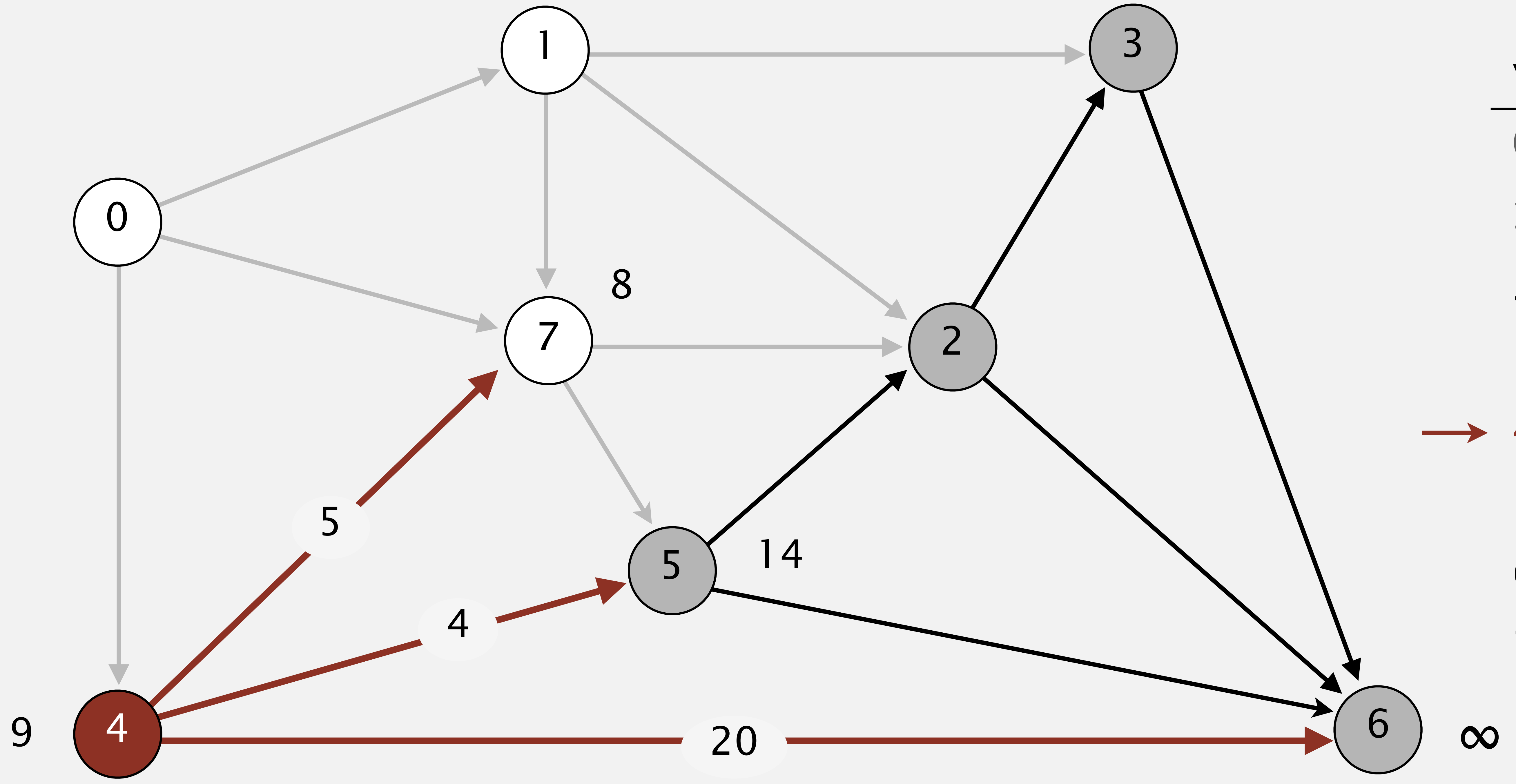
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
→ 4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

select vertex 4

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



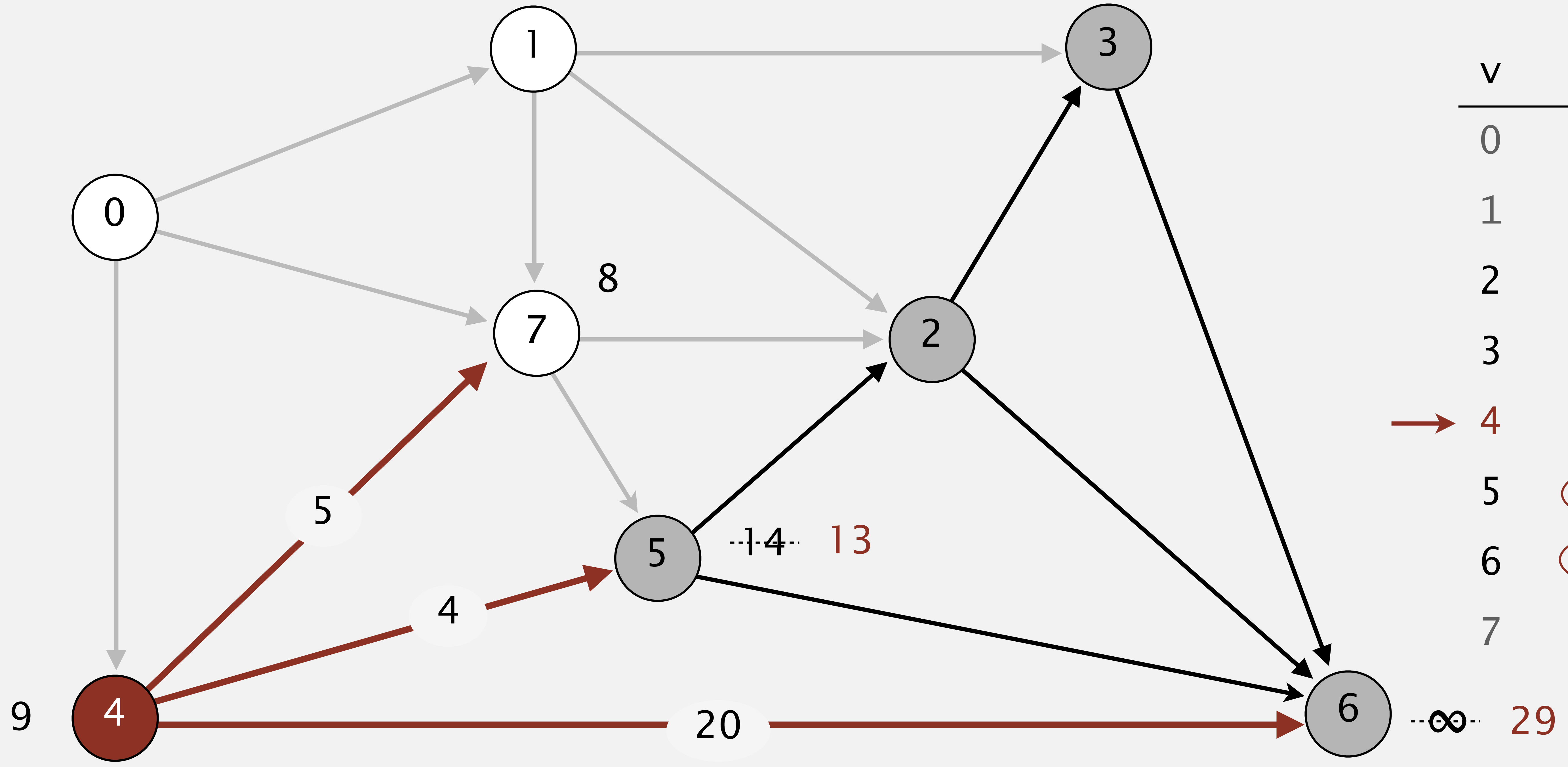
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
→ 4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

relax all edges adjacent from 4

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



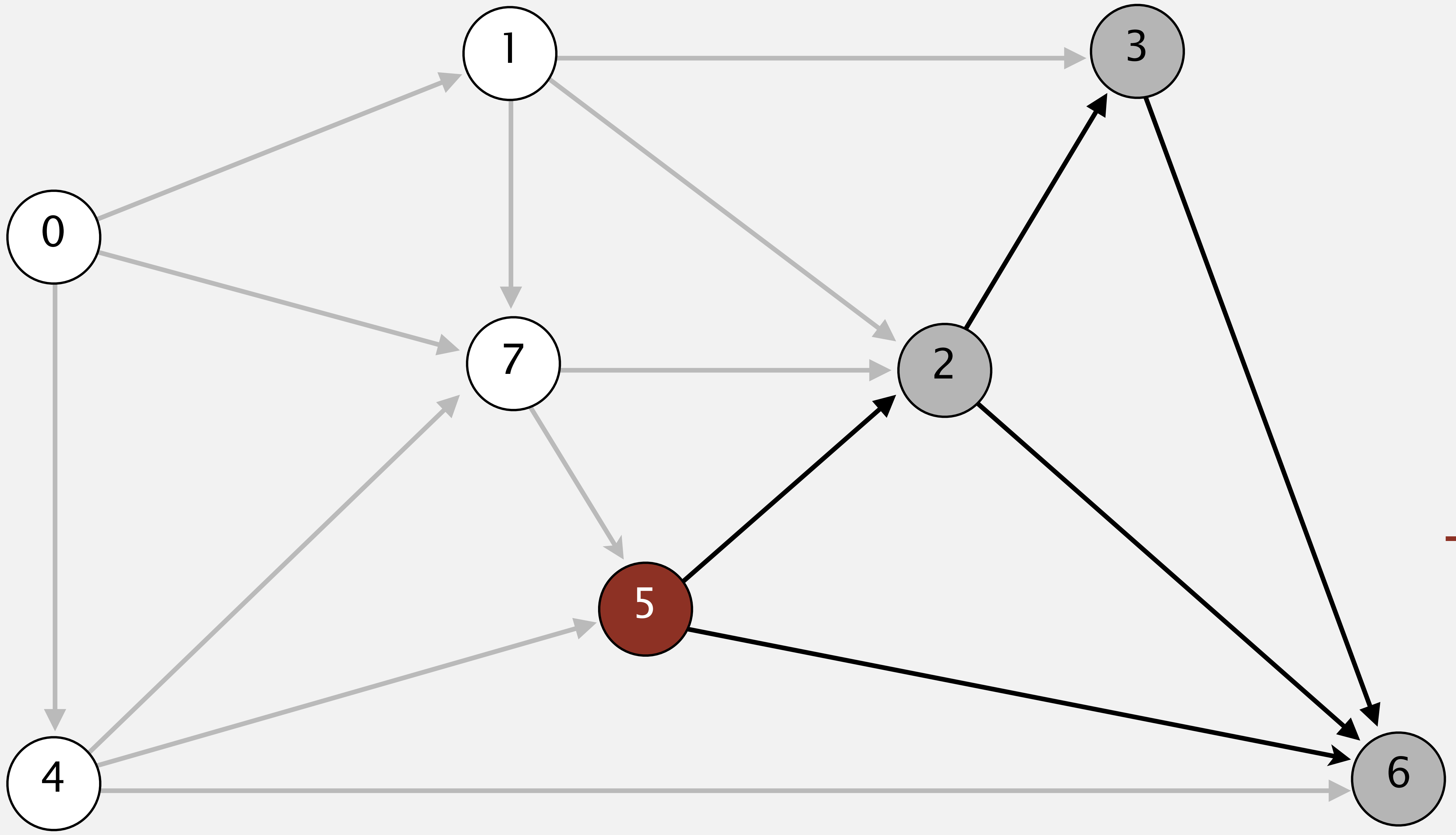
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
→ 4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0 ✔	0→7

relax all edges adjacent from 4

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



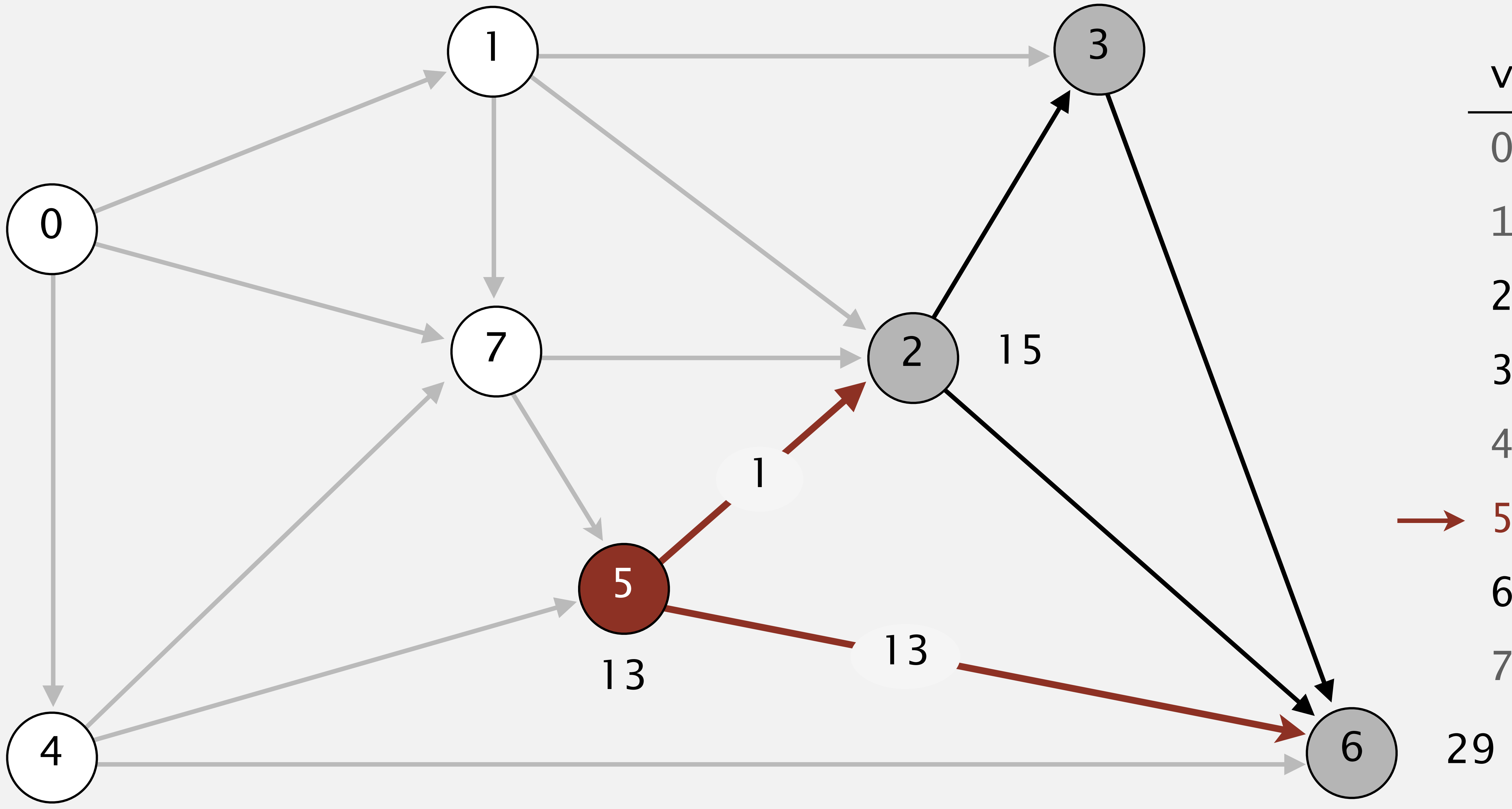
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
→ 5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

select vertex 5

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



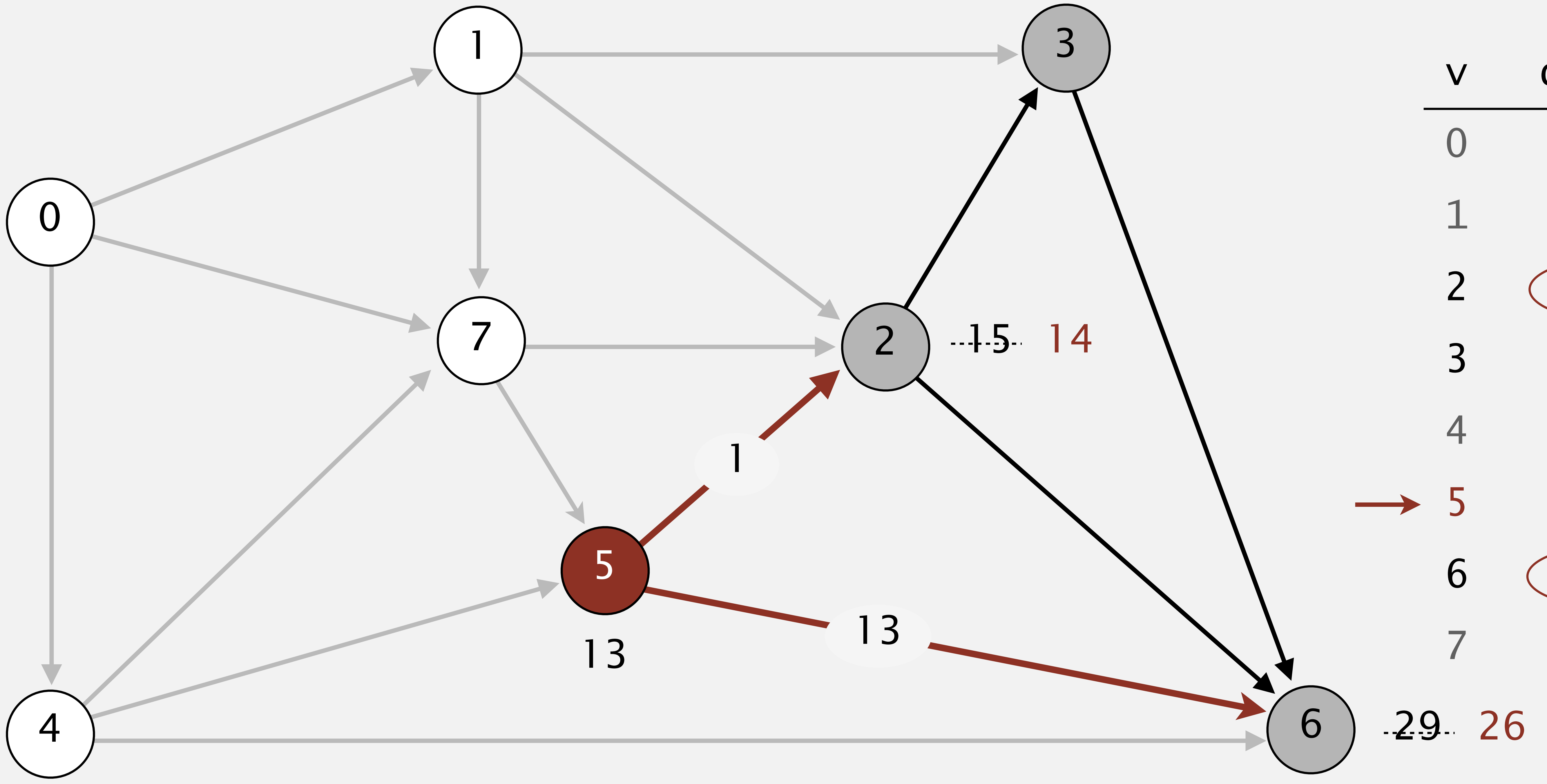
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
→ 5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

relax all edges adjacent from 5

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



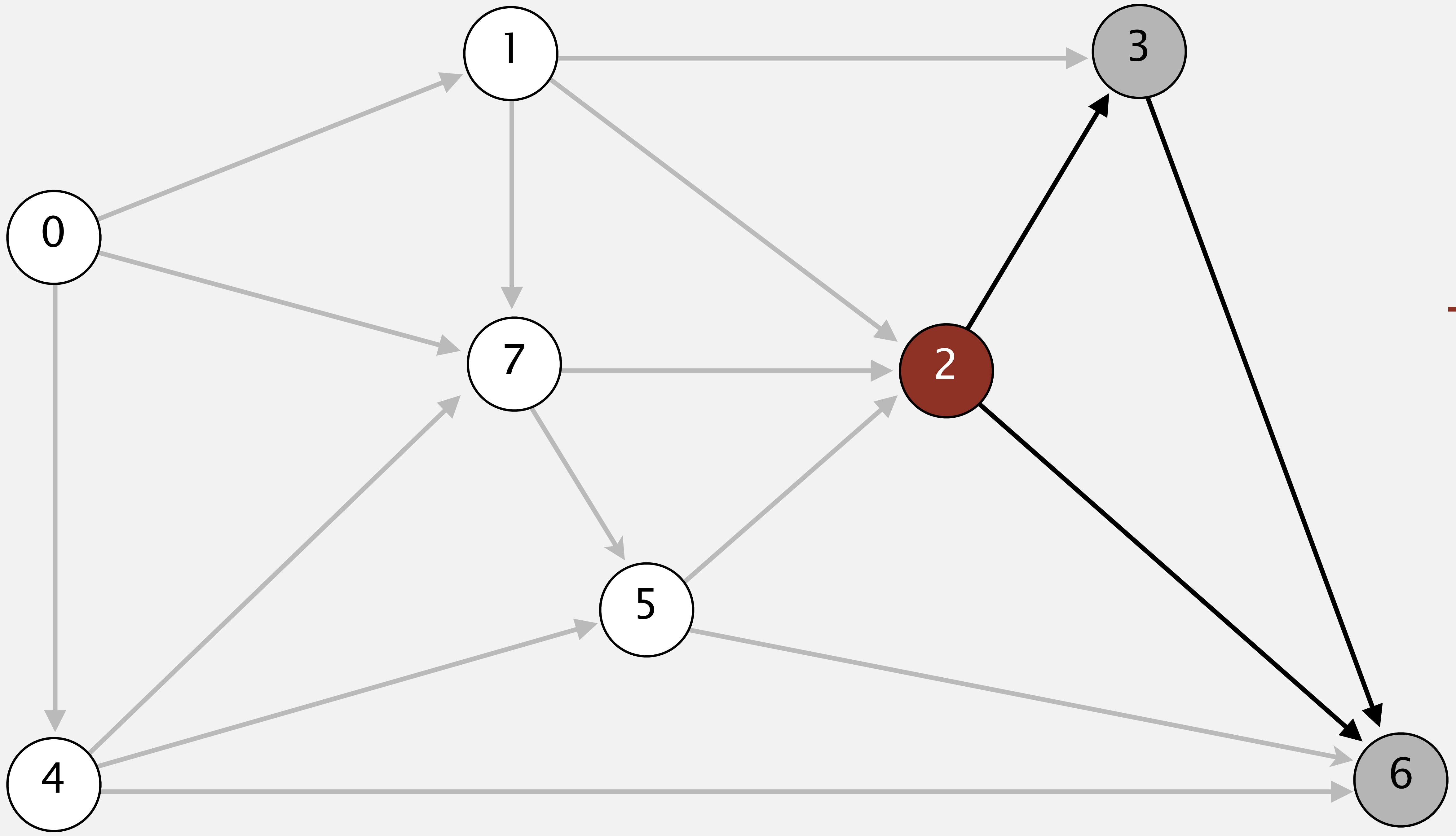
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
→ 5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

relax all edges adjacent from 5

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



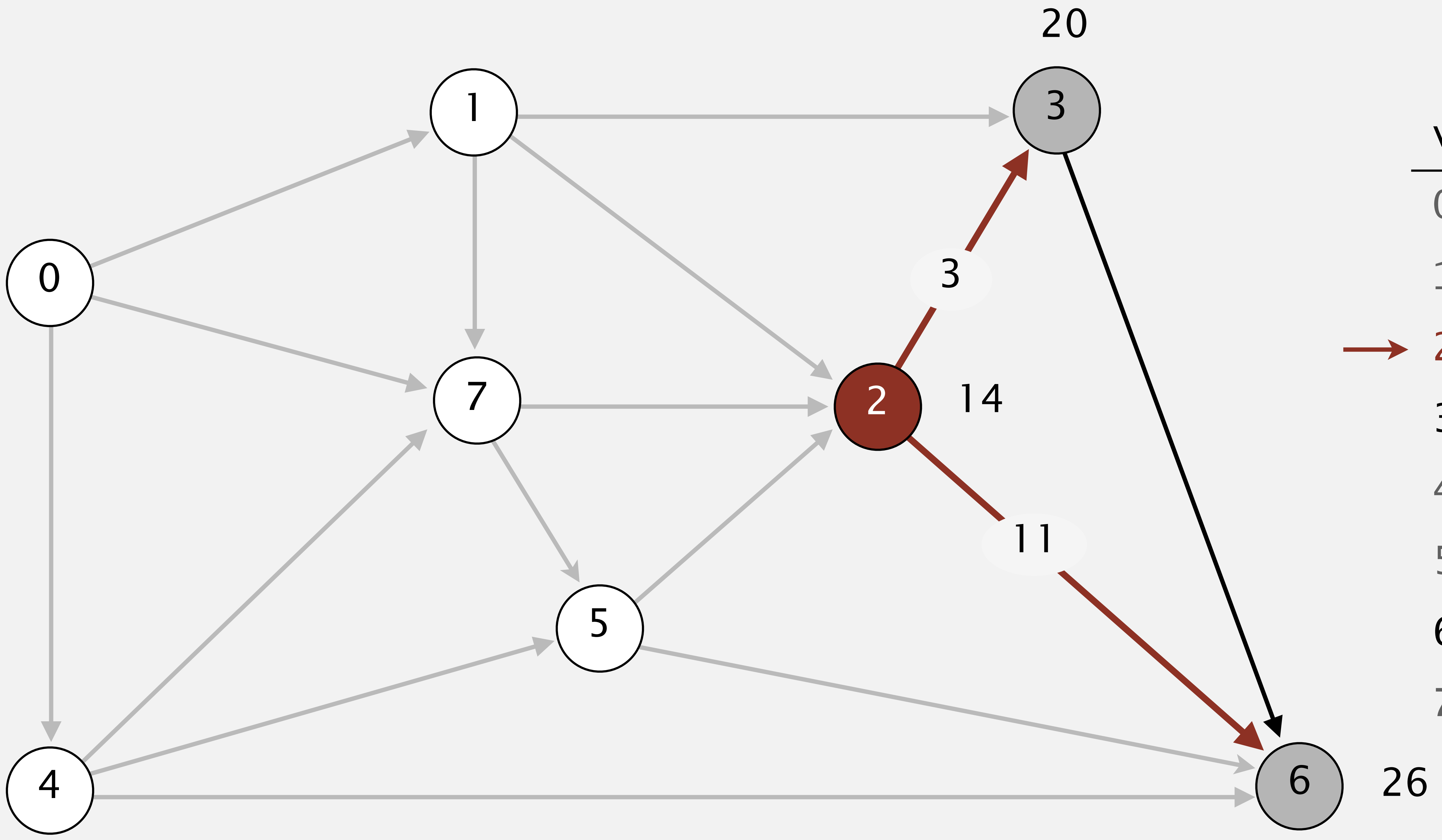
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
→ 2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

select vertex 2

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



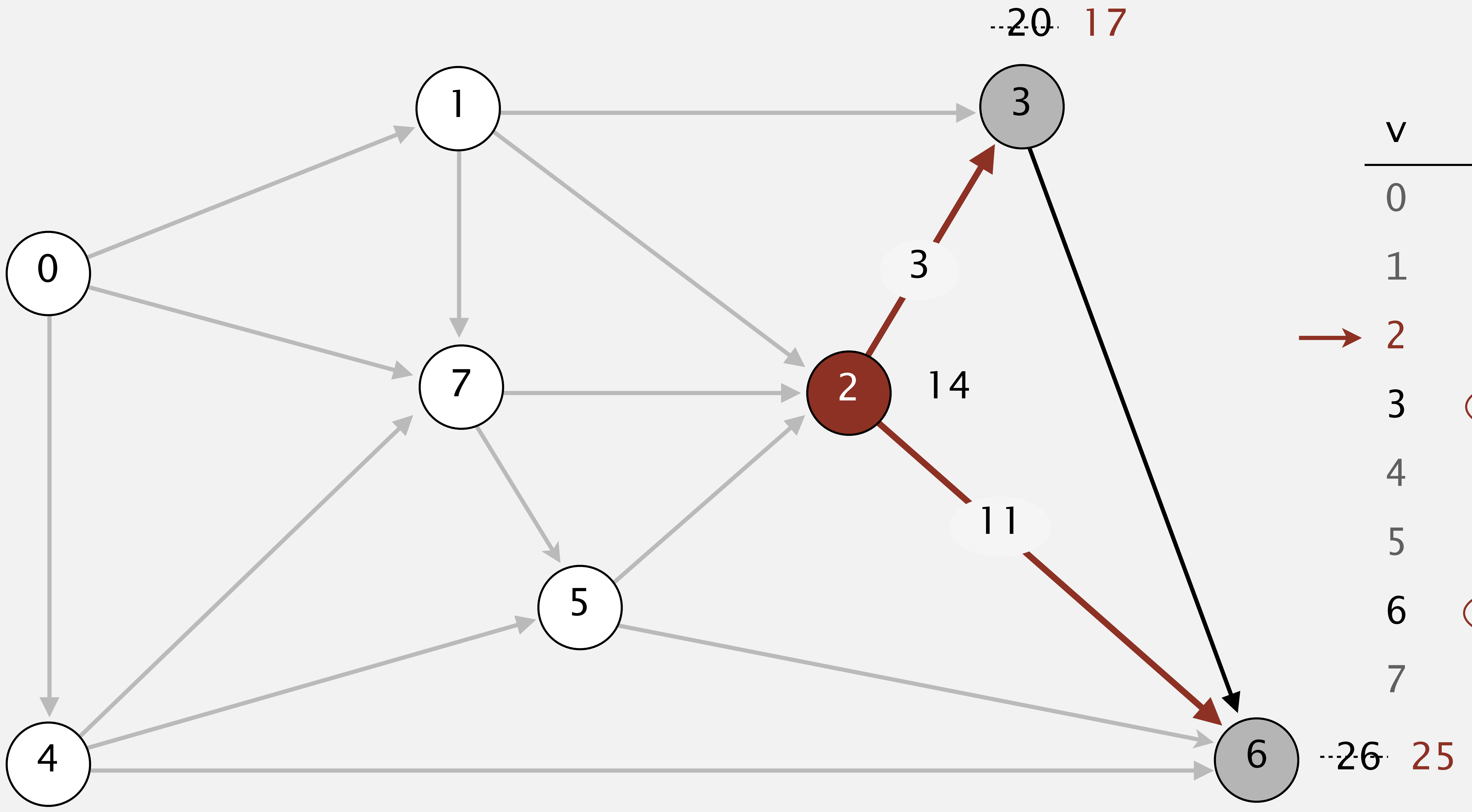
	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
→	2	14.0	5→2
	3	20.0	1→3
	4	9.0	0→4
	5	13.0	4→5
	6	26.0	5→6
	7	8.0	0→7

relax all edges adjacent from 2

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



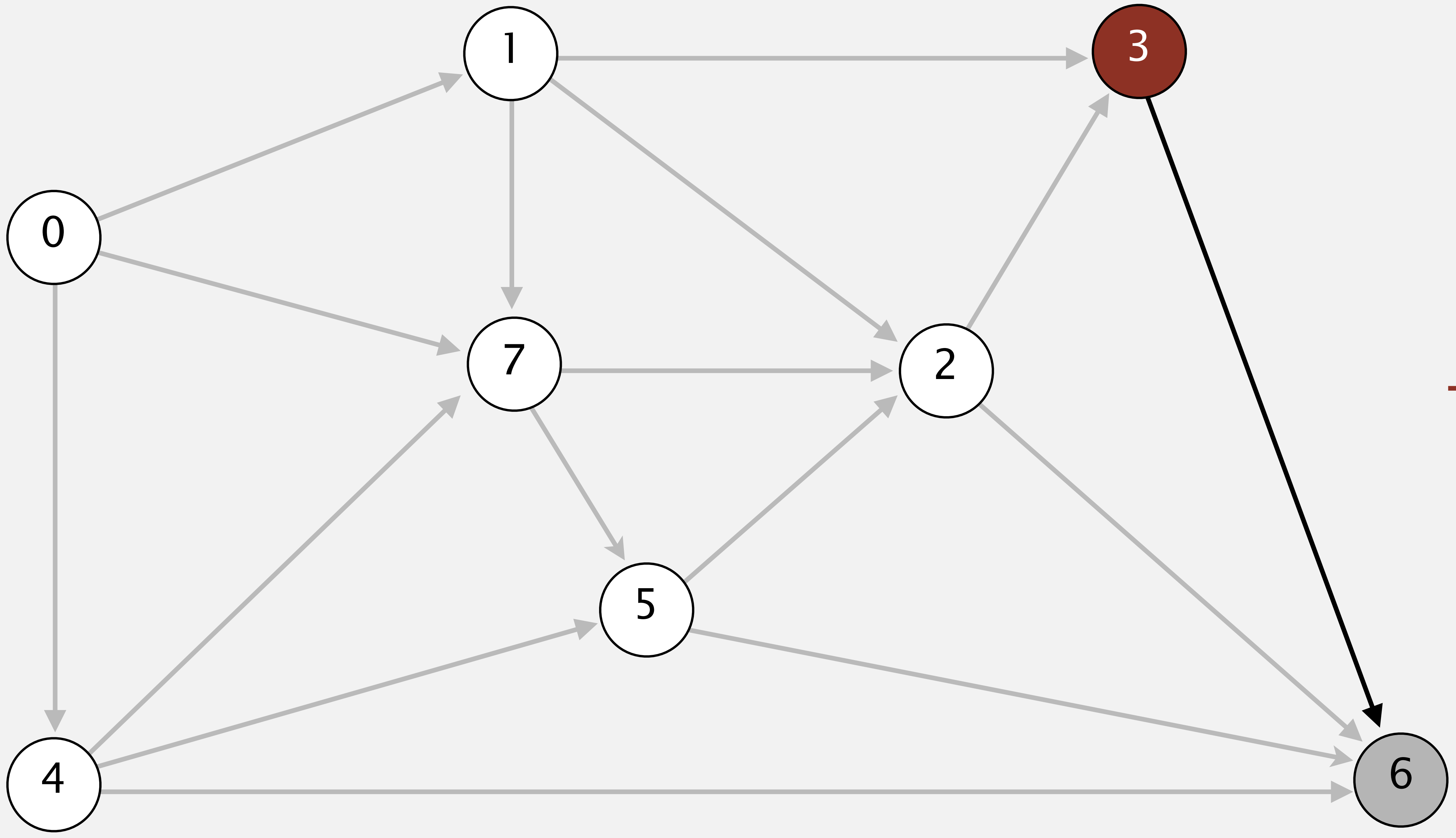
	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
→	2	14.0	5→2
	3	17.0	2→3
	4	9.0	0→4
	5	13.0	4→5
	6	25.0	2→6
	7	8.0	0→7

relax all edges adjacent from 2

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



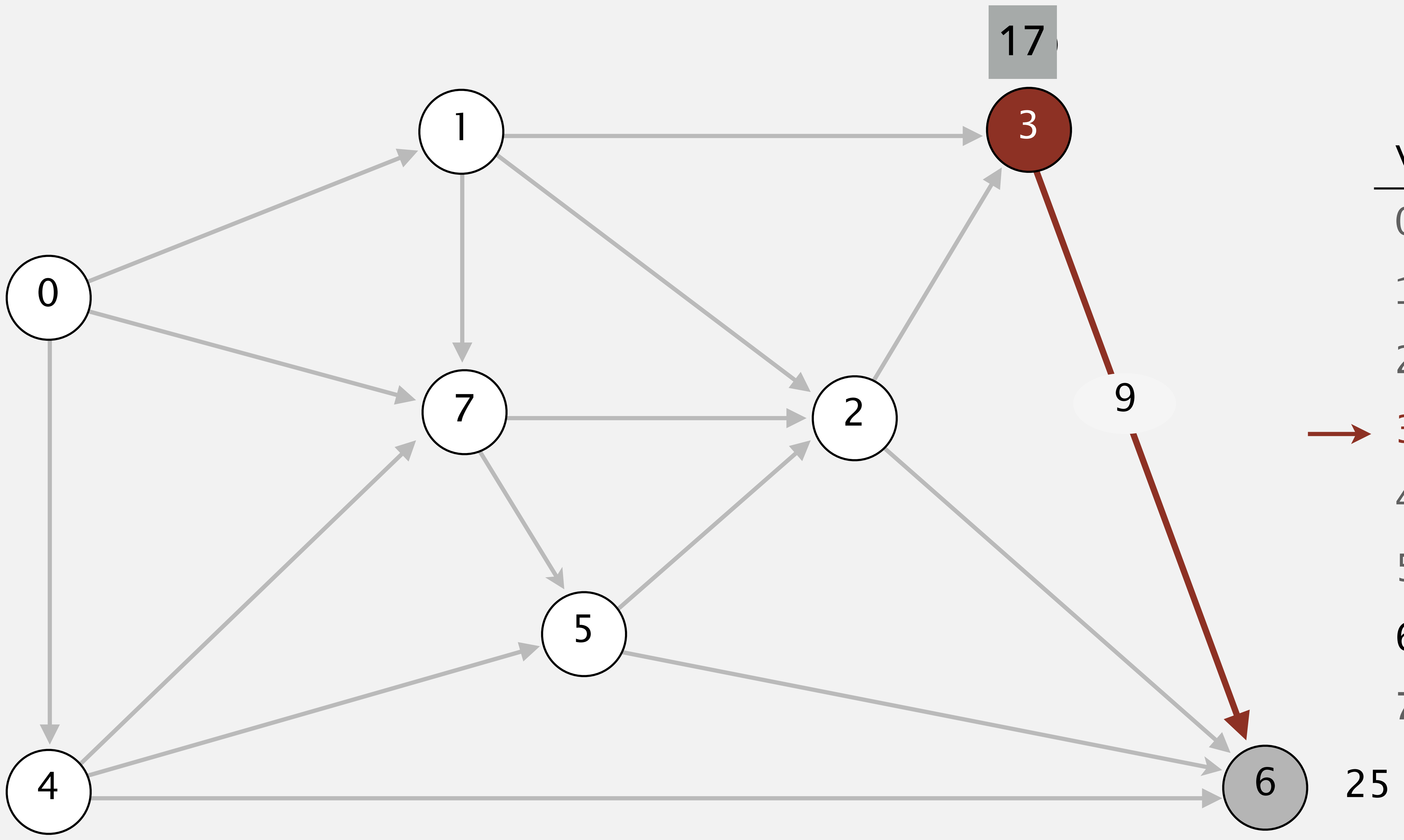
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
→ 3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

select vertex 3

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



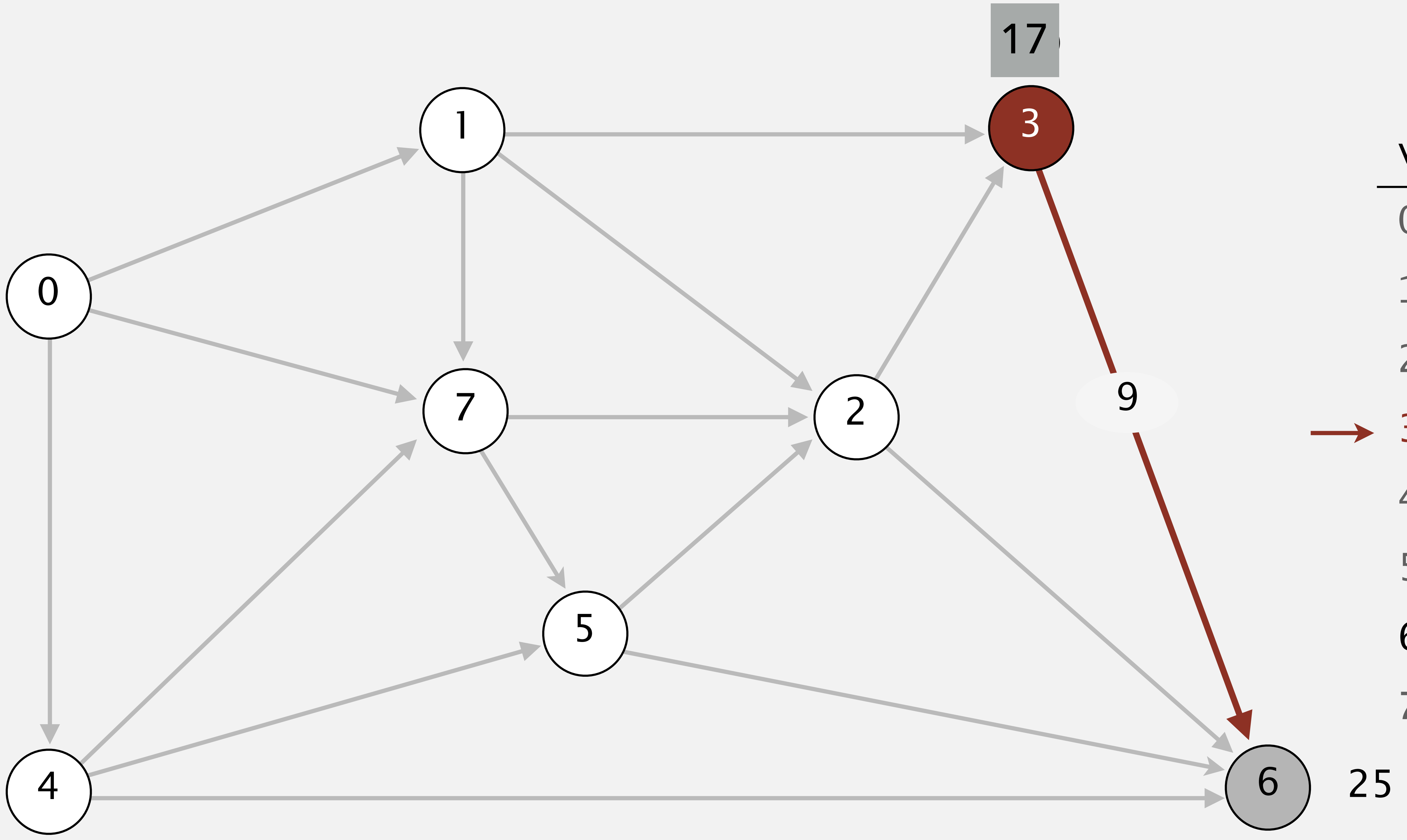
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
→ 3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

relax all edges adjacent from 3

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



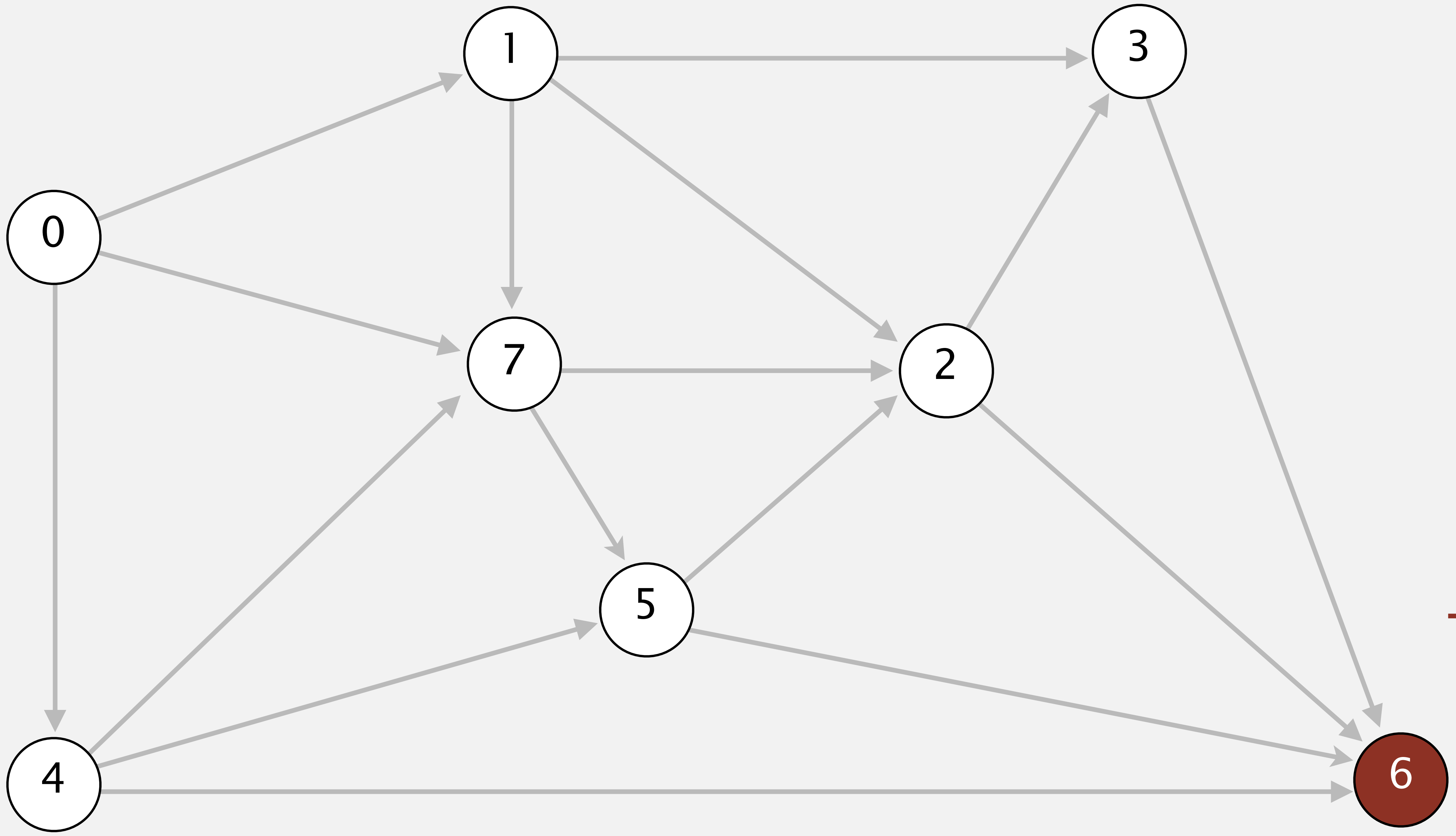
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
→ 3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0 ✔	2→6
7	8.0	0→7

relax all edges adjacent from 3

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



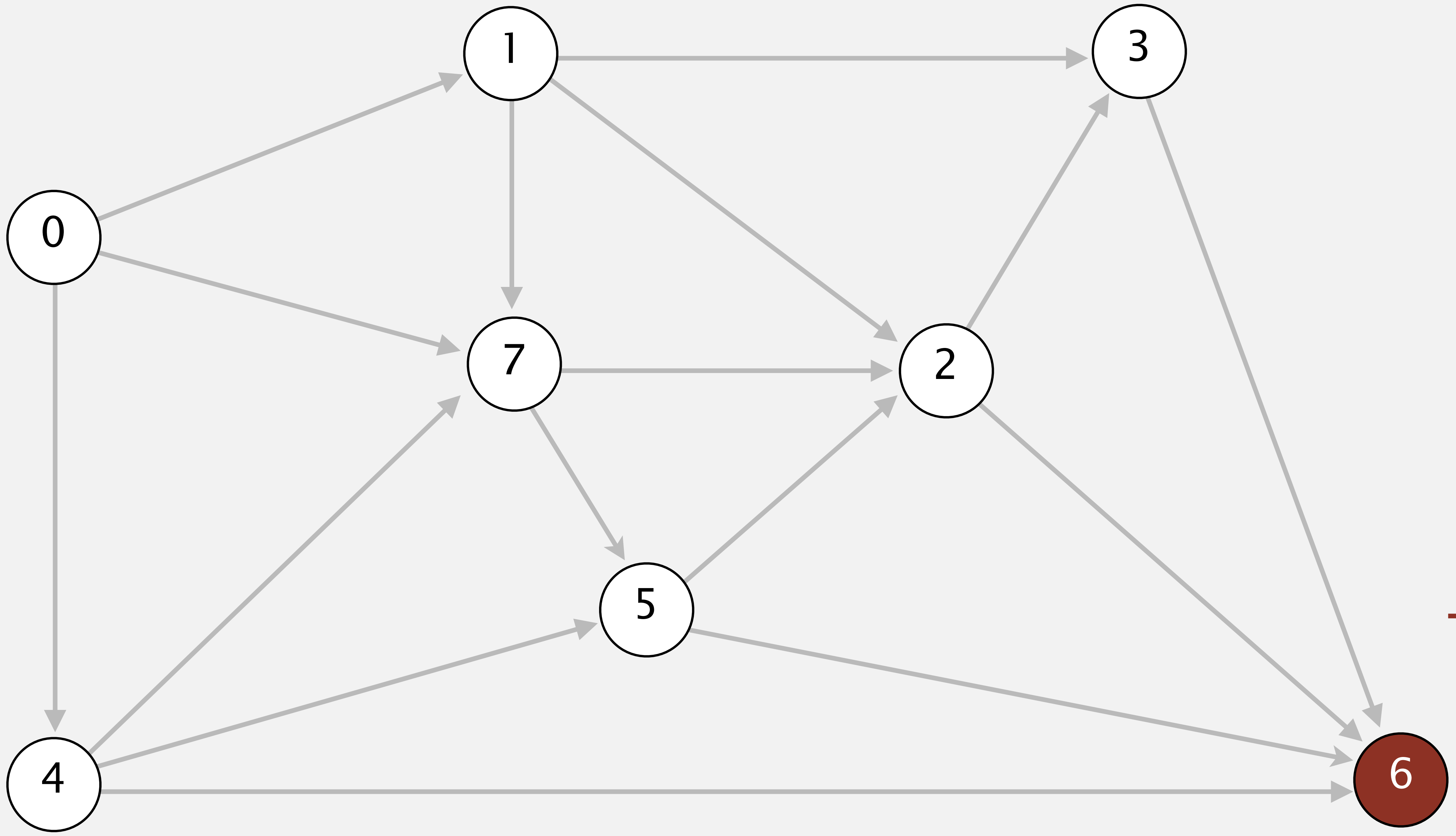
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
→ 6	25.0	2→6
7	8.0	0→7

select vertex 6

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



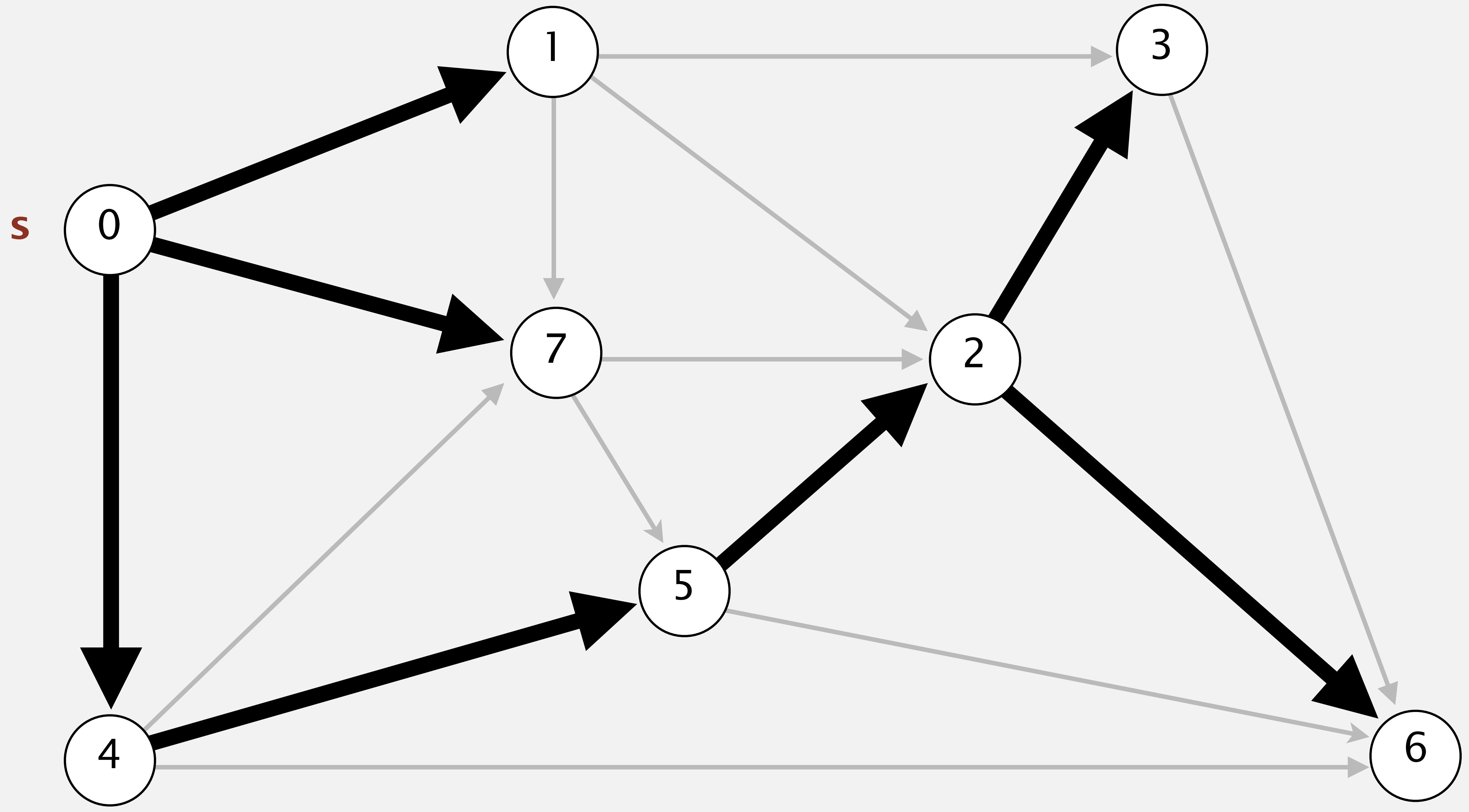
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
→ 6	25.0	2→6
7	8.0	0→7

relax all edges adjacent from 6

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s