Tutorial 10 — Shortest Paths

CS2040S Semester 1 2023/24

By Wu Biao, adapted from previous semesters

Shortest Path

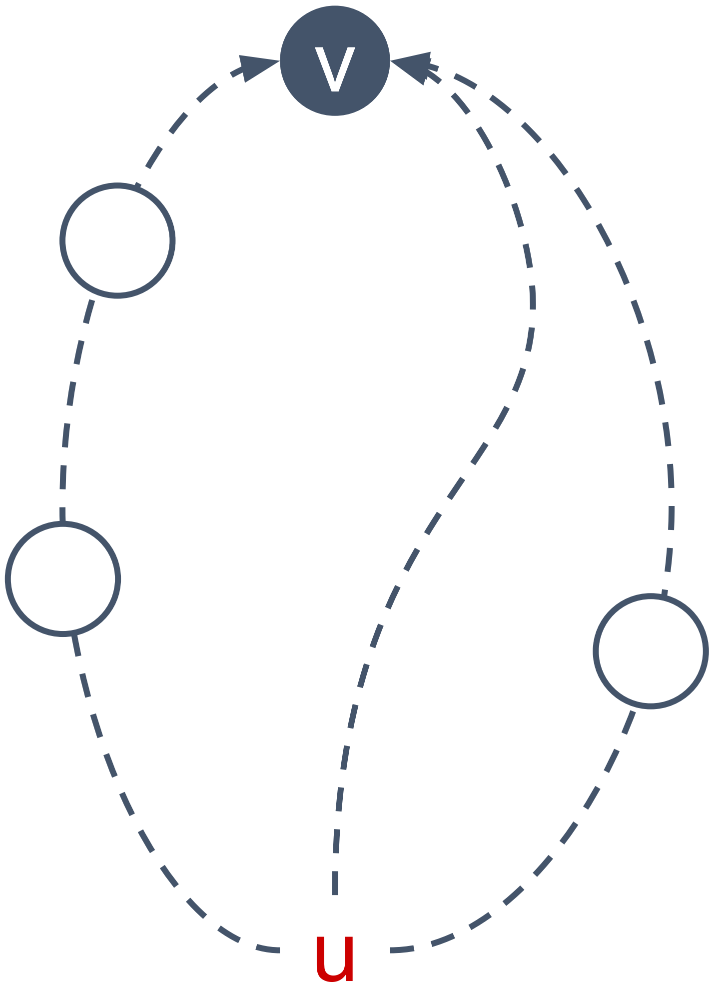
SSSP

Shortest Path Problems

Given a graph with weighted/unweighted edges,

what is $\delta(u, v)$, the **shortest path** from a vertex u to another vertex v?

Shortest means the path with lowest "length".



Shortest Path Problems

The definition of “length” varies from problem to problem:

- [Most common] Sum of all edge weights in the path.
- Number of edges in the path. Applies for unweighted graph or graph with all edge weight being equal.
- Sum of all *vertex weights* in the path.
- *Product* of all edge weights in the path
  - Only applicable if all weights are positive.

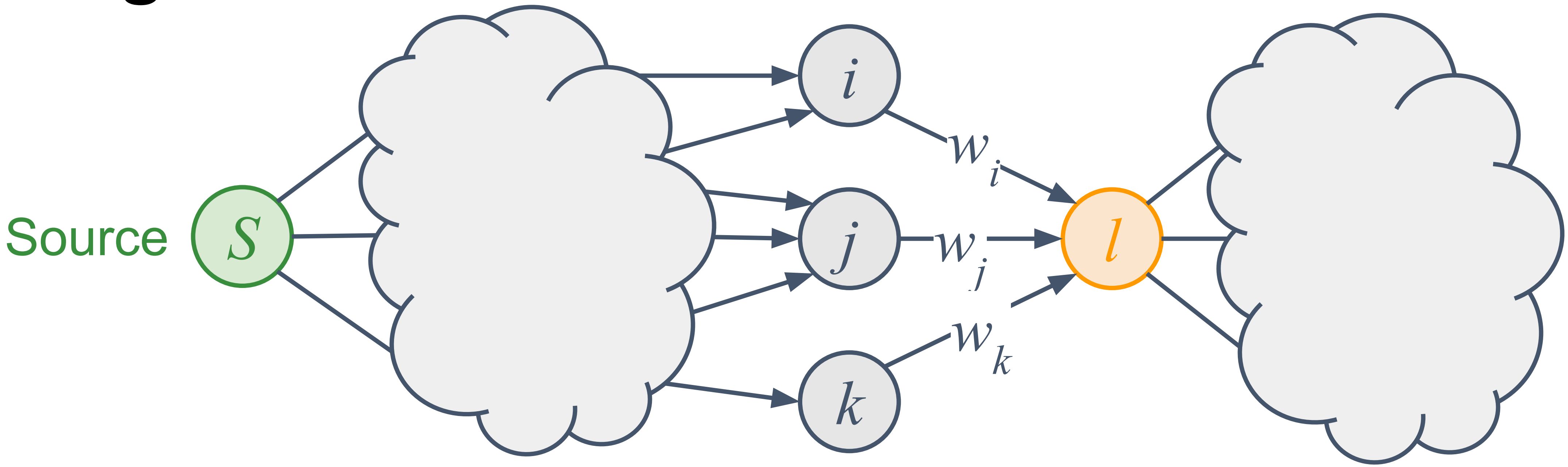
SSSP Algorithms

SSSP Algorithms

Algorithm	Time complexity
*BFS	$O(V+E)$
Bellman Ford	$O(VE)$
Dijkstra	$O((V+E)\log V)$
Modified Dijkstra	$\approx O((V+E)\log V)$

* BFS can be used to solve SSSP, ONLY if all edges are of same weight. In this case BFS is much faster than other standard weighted SSSP algorithms

SSSP Algorithms — An observation



We shall denote the shortest distance from source S to any vertex u to be $\delta[u]$.

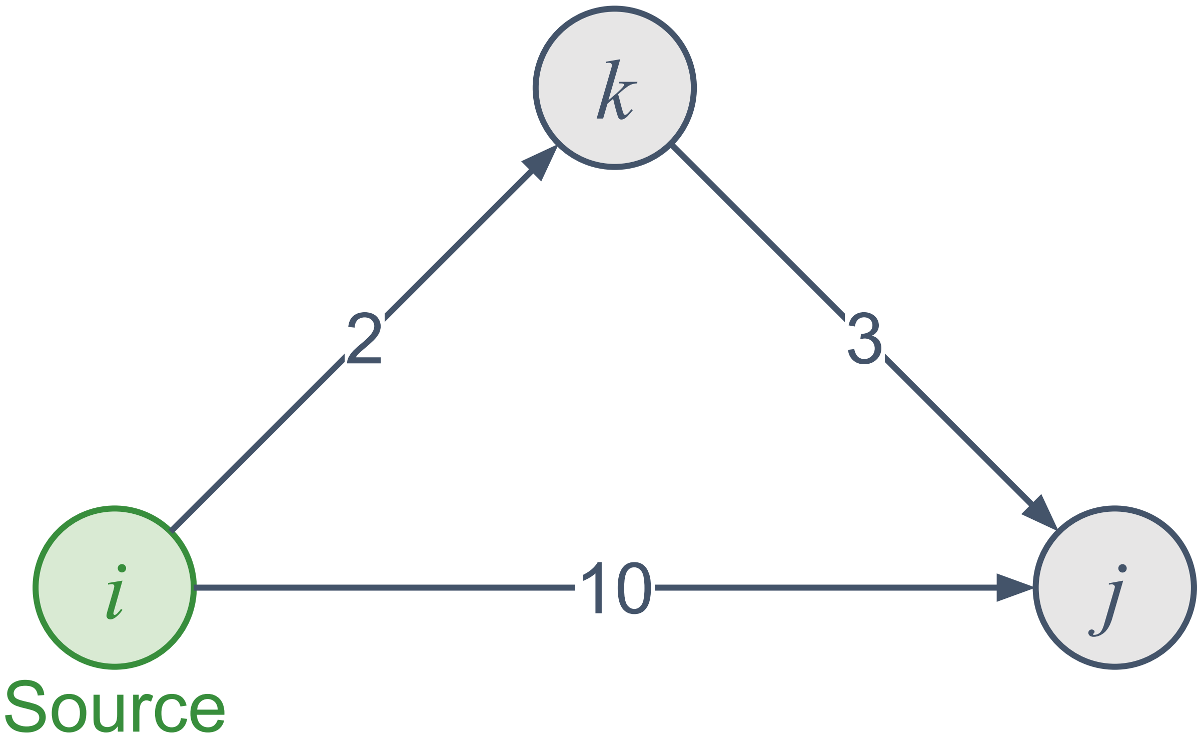
Given the graph, realize that $\delta[l] = \text{Minimum} \begin{pmatrix} \delta[i] + w_i \\ \delta[j] + w_j \\ \delta[k] + w_k \end{pmatrix}$

SSSP Algorithms — Main idea

- Maintain a tentative “shortest distance” table D.
- Incrementally work towards an optimal solution by “Relaxing” vertices.
- At the end of the process, D converges to optimal solution and reflects the actual shortest path to every vertex from the given source vertex. i.e. $\mathrm{D} = \delta$.
- Different algorithms essentially carry out the relaxations in different order!

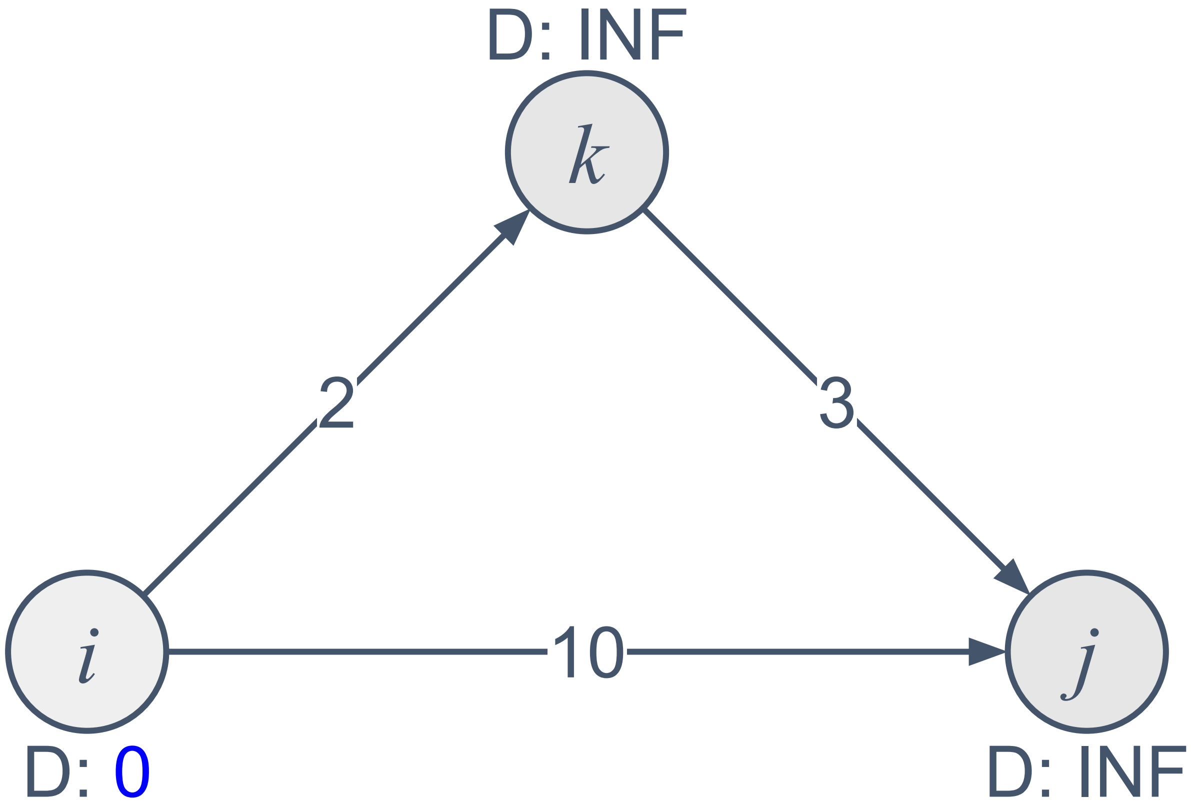
What is Relax?

Let's illustrate by running SSSP on this graph with the source being vertex i.



What is Relax?

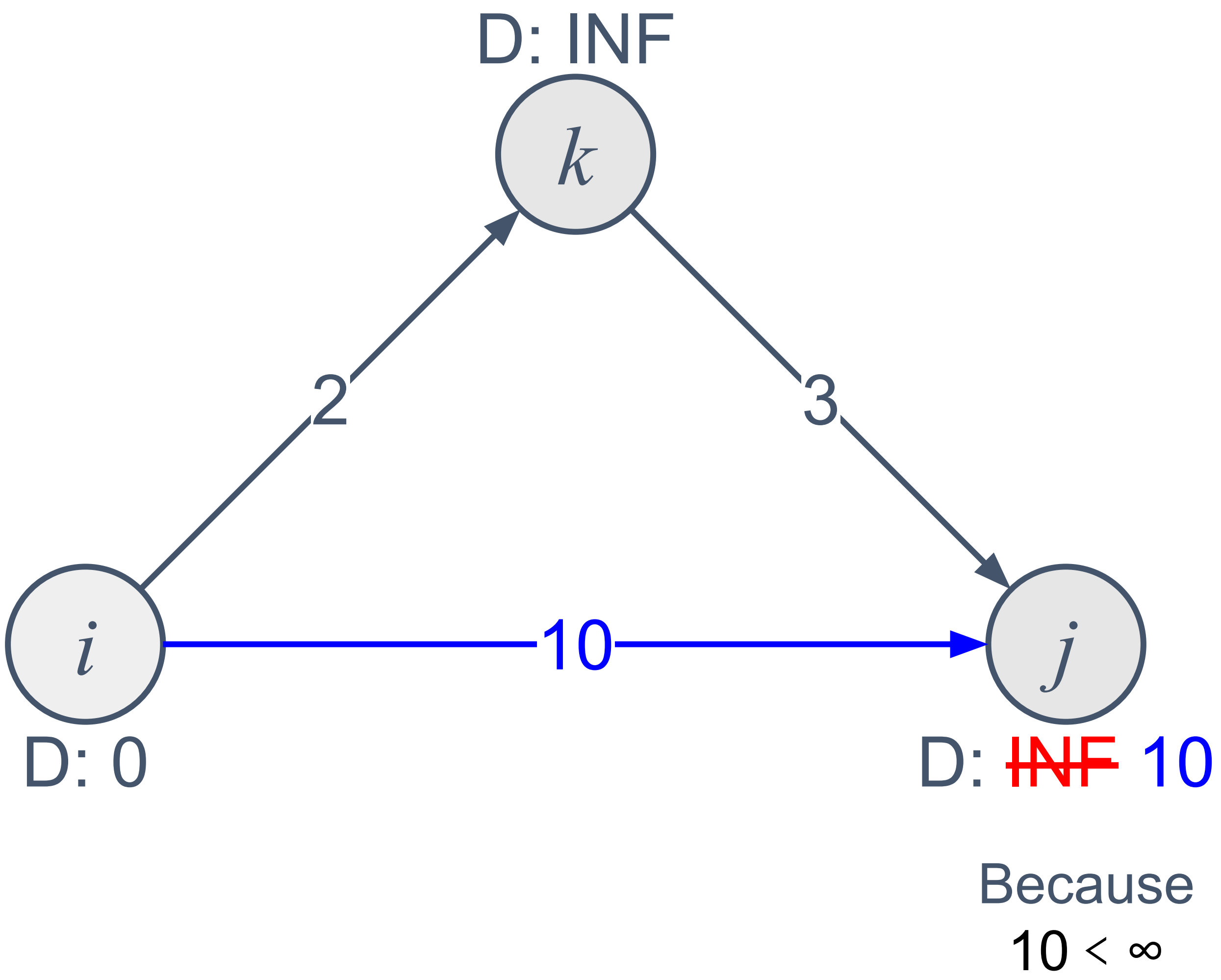
We initialize all distances to be infinity, with the exception of source vertex, which trivially has a distance of 0.



What is Relax?

We found a path from i to j with total weight 10 (i→j), which is shorter than INF!

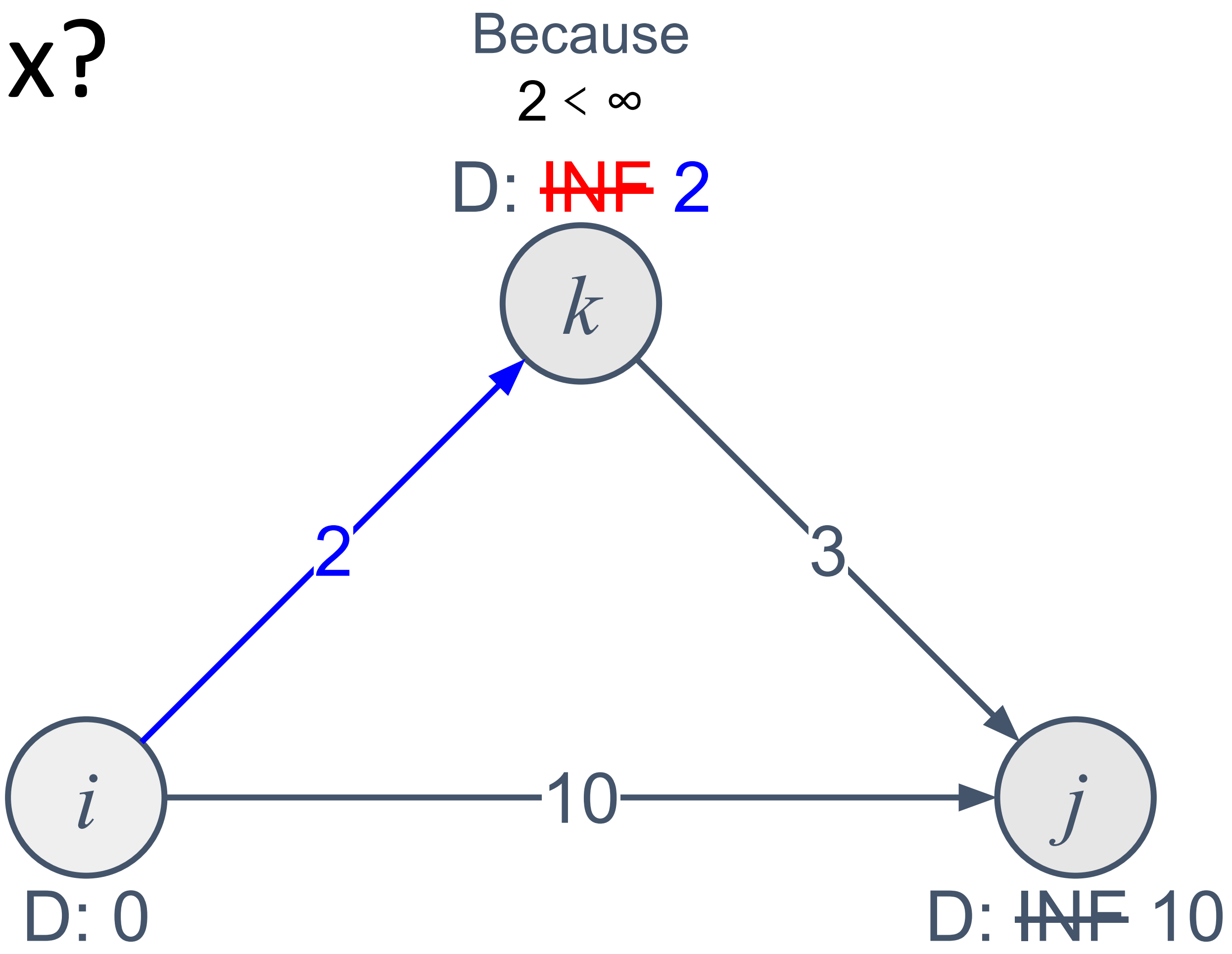
So we relax i→j.



What is Relax?

We found a path from i to k with total weight 2 ($i \rightarrow k$), which is shorter than INF!

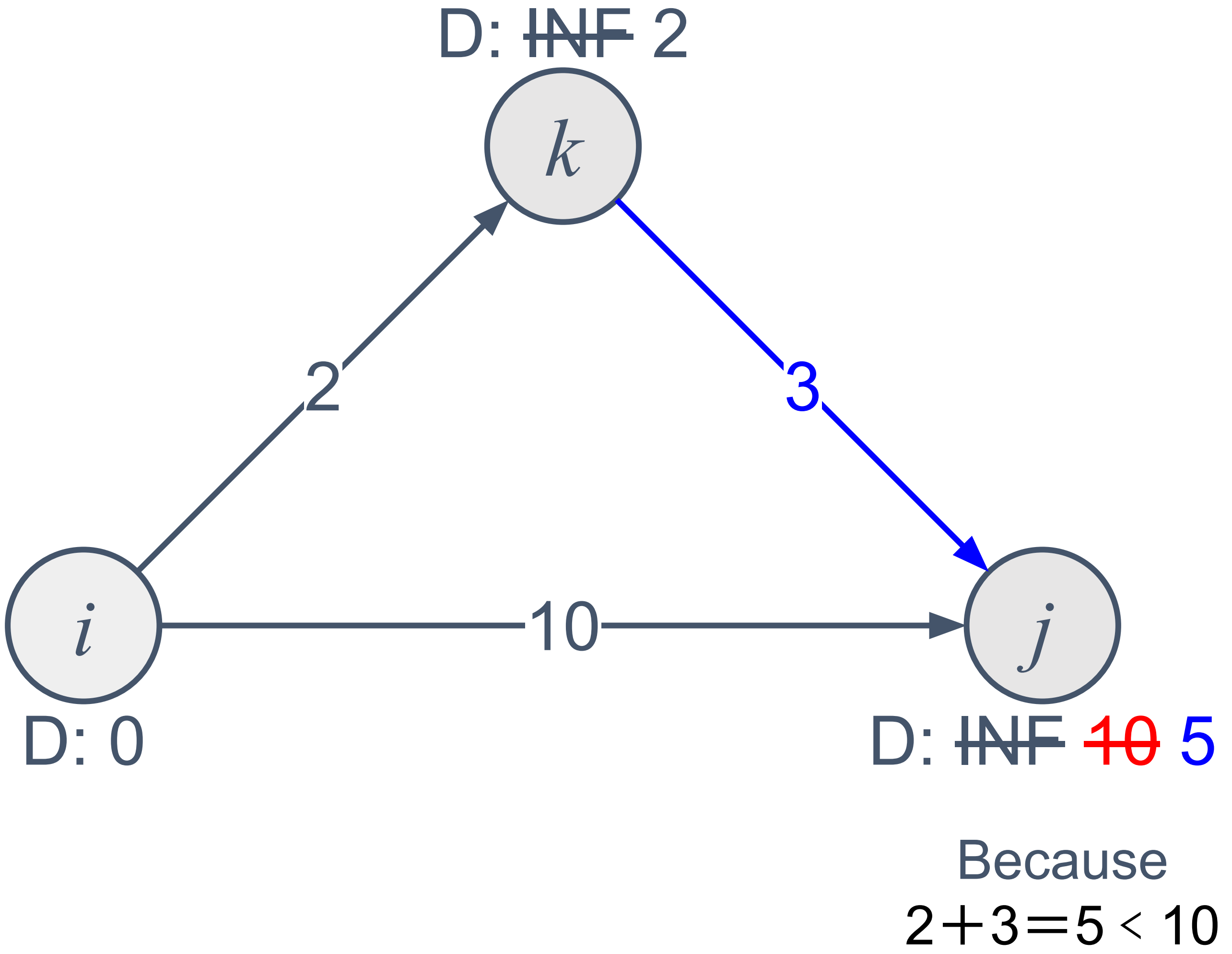
So we relax $i \rightarrow k$.



What is Relax?

We found a path from i to j with total weight 5 (i⟶k⟶j), which is shorter than 10!

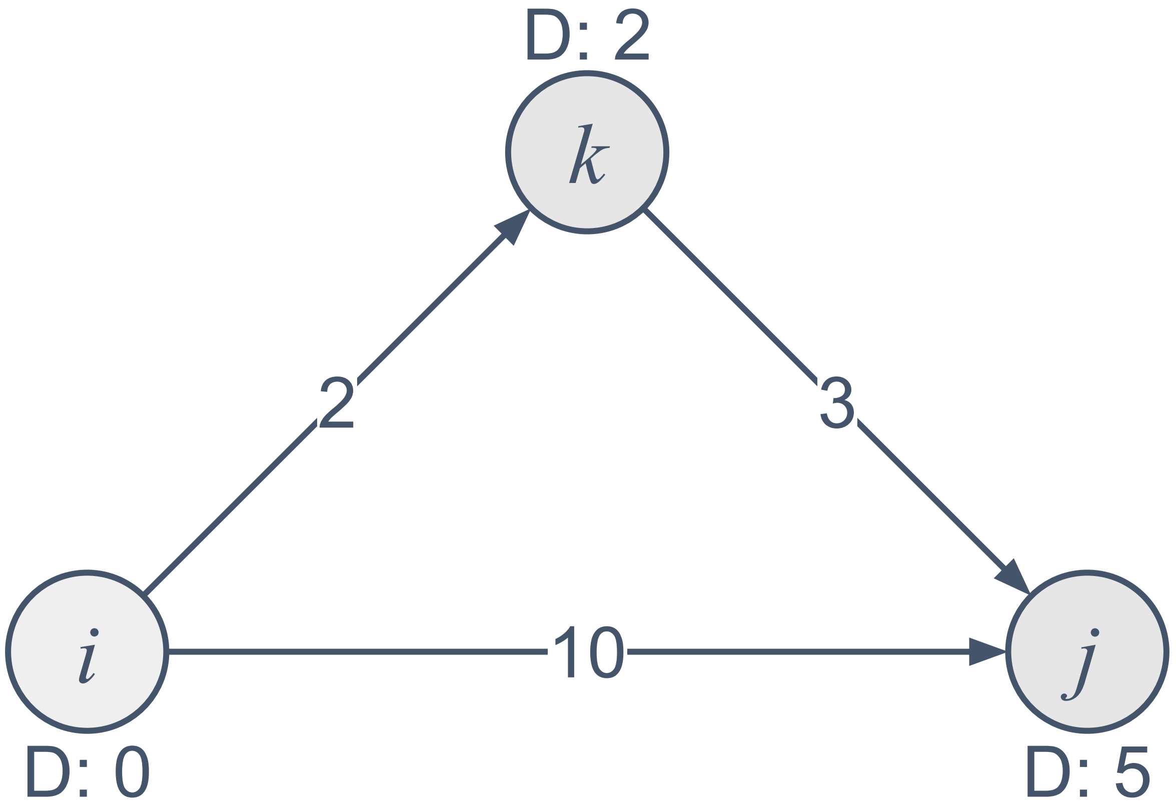
So we relax k⟶j.



What is Relax?

We are now done and D captures all the shortest path lengths from i!

Essentially we traversed each edge and updated the tentative 'shortest distance' every time.



What is Relax?

So formally, for an edge u→v with weight w, when we say “relax **edge** u→v”, we mean the following:

- If D[u]＋w < D[v]: D[v] = D[u]＋w.
- Shortest path to v is now shortest path to u followed by u→v

We sometimes also refer to this as “relaxing **vertex** v with edge u→v”.

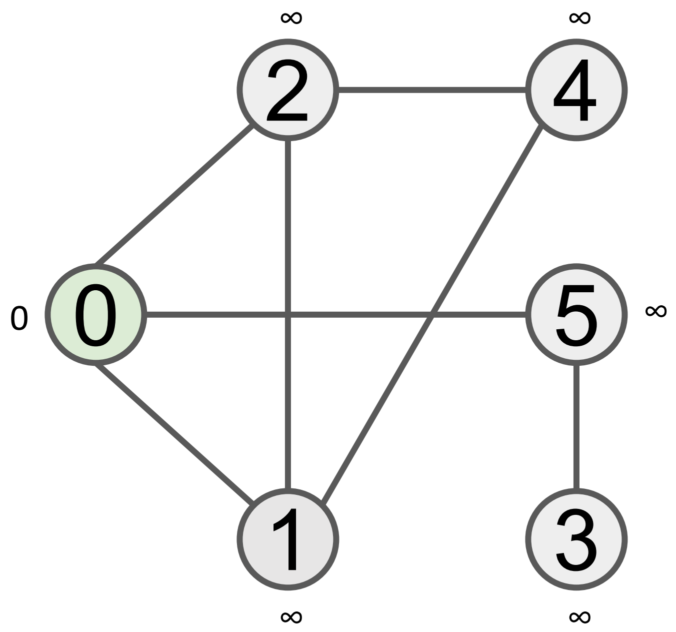
Note relaxing has direction, so for undirected graph you can relax at both direction.

BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: [0]

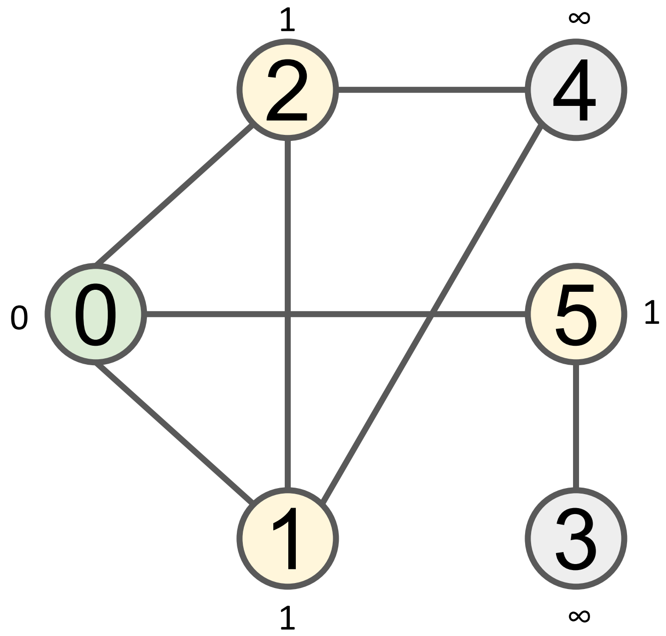


BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: [1, 2, 5]



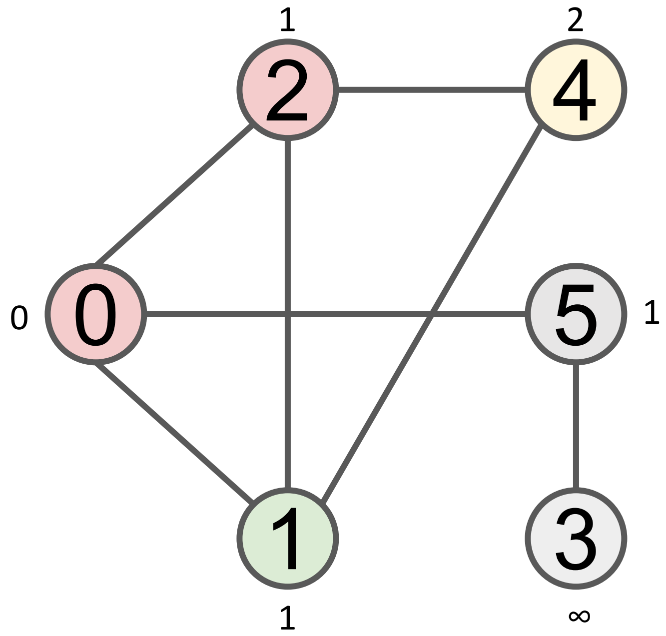
Relax edges (0, 1), (0, 2), (0, 5)

BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: [2, 5, 4]



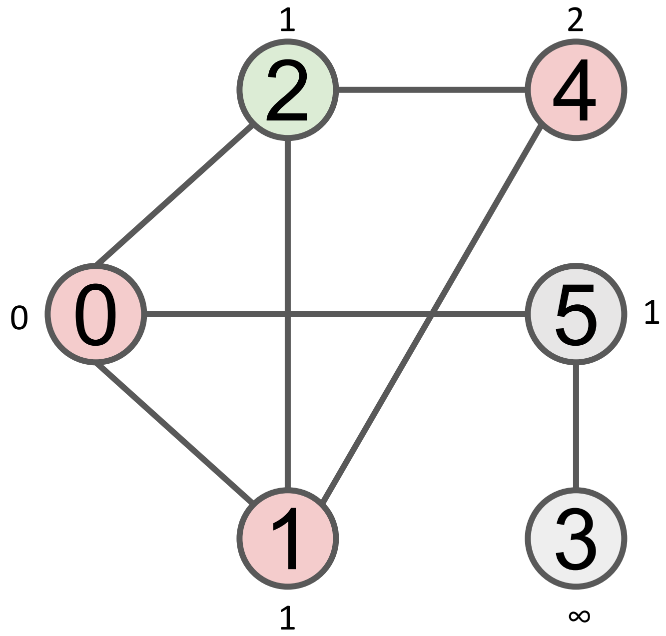
Cannot relax (1, 0) and (1, 2).
Relax edge (1, 4).

BFS - Algorithm

Queue: [5, 4]

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```



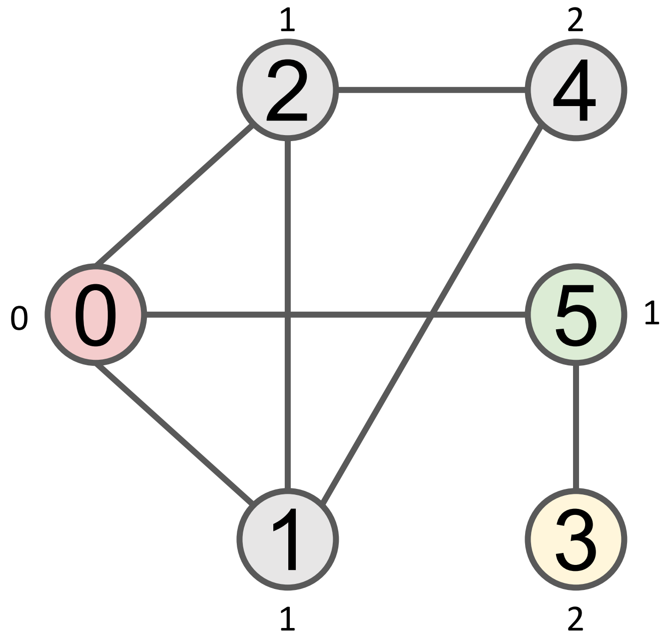
Cannot relax (2, 0)
or (2, 1), or (2, 4).

BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: [4, 3]



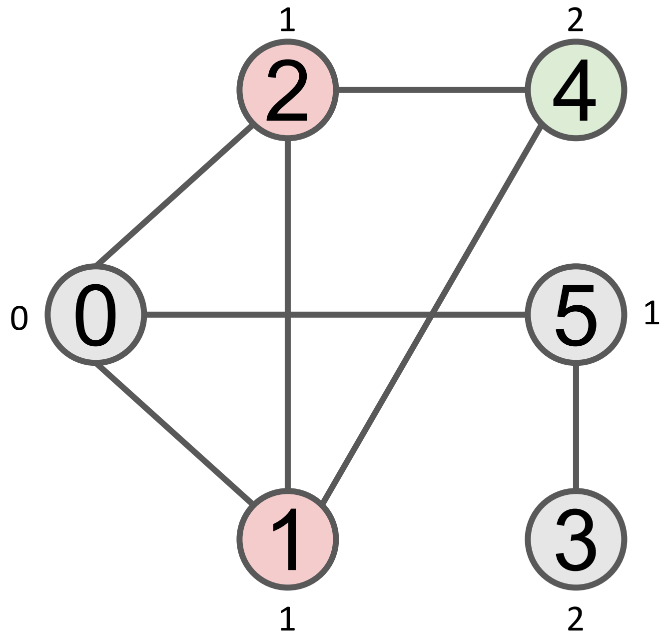
Cannot relax (5, 0).
Relax edge (5, 3).

BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: [3]



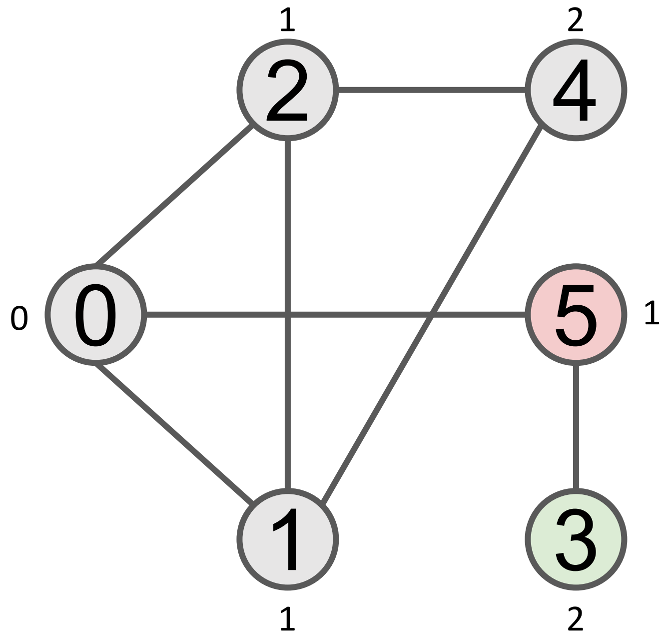
Cannot relax (4, 1), (4, 1).

BFS - Algorithm

Relax the edges level by level.

```
Queue<Integer> q = new ArrayDeque<>();
level[src] = 0;
q.push(src);
while (!q.empty()) {
    int u = q.poll();
    for (int v : adj[u]) {
        if (level[v] > level[u] + 1) {
            level[v] = level[u] + 1;
            q.push(v);
        }
    }
}
```

Queue: []



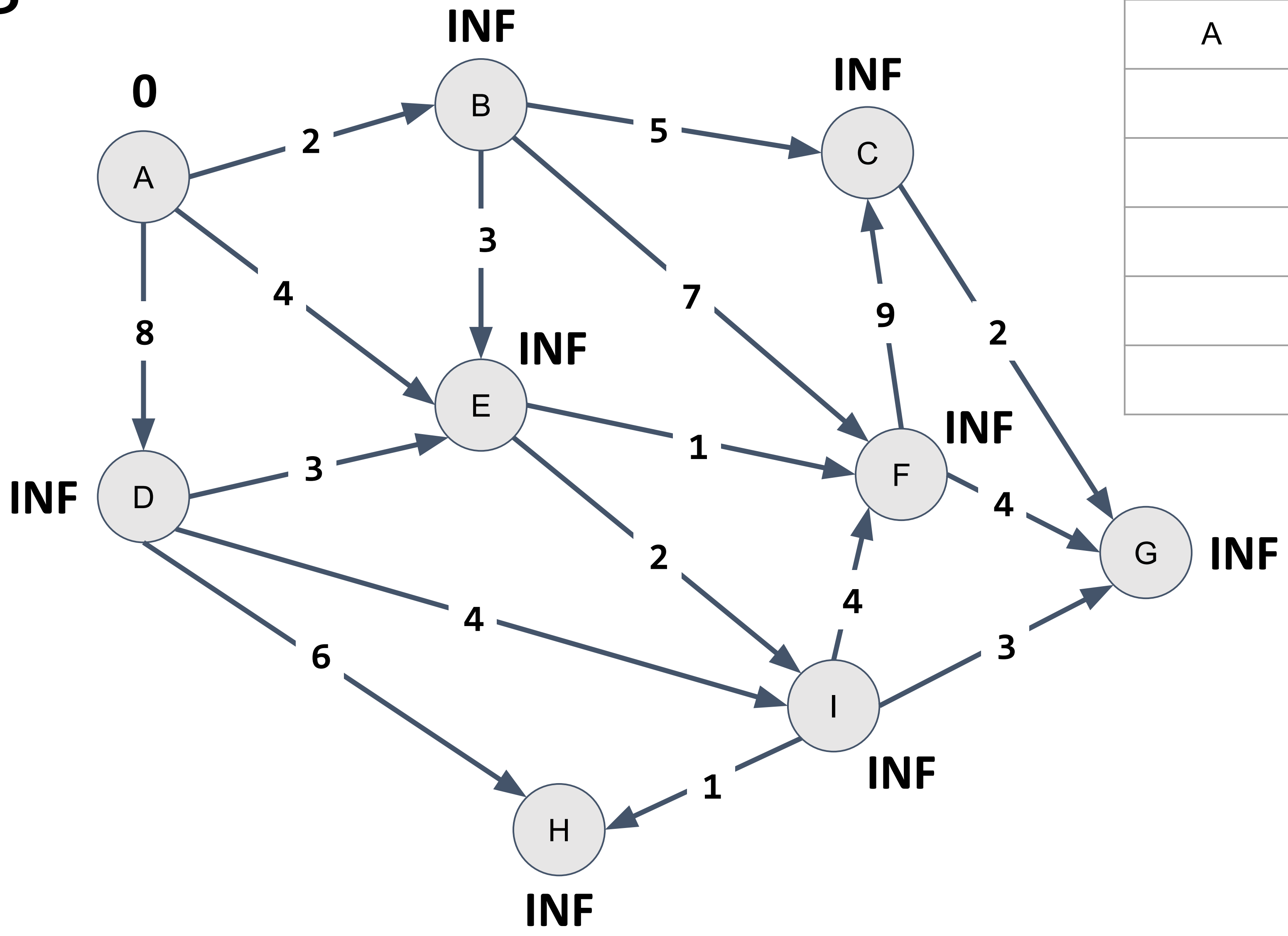
Cannot relax (5, 3).
Queue empty; done.

Dijkstra — Algorithm

1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0.
2. Mark all vertices to as "unresolved".
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far:
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph.

Dijkstra — Algorithm

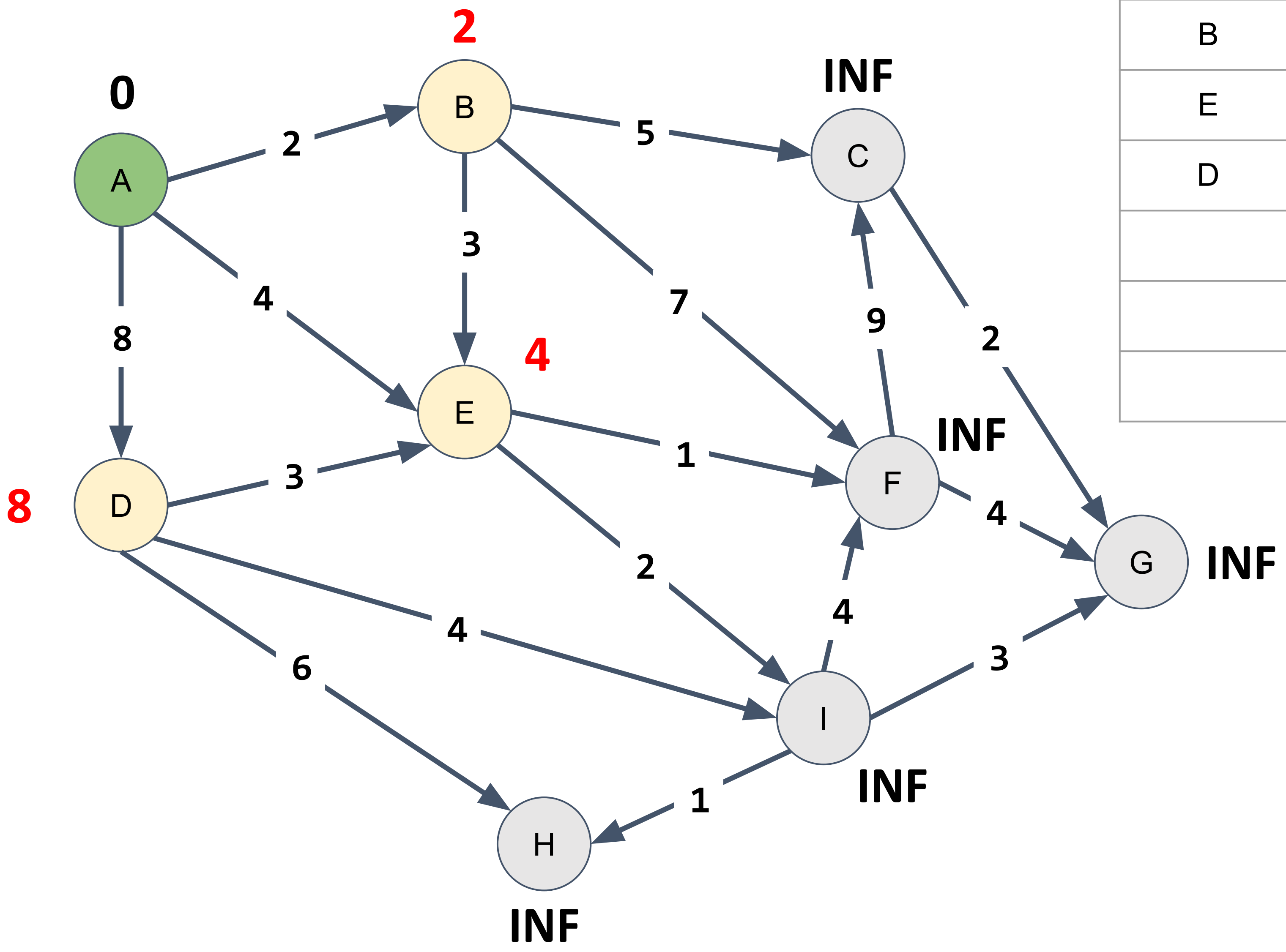
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
    a. Mark it as **resolved**
    b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
A	0

Dijkstra — Algorithm

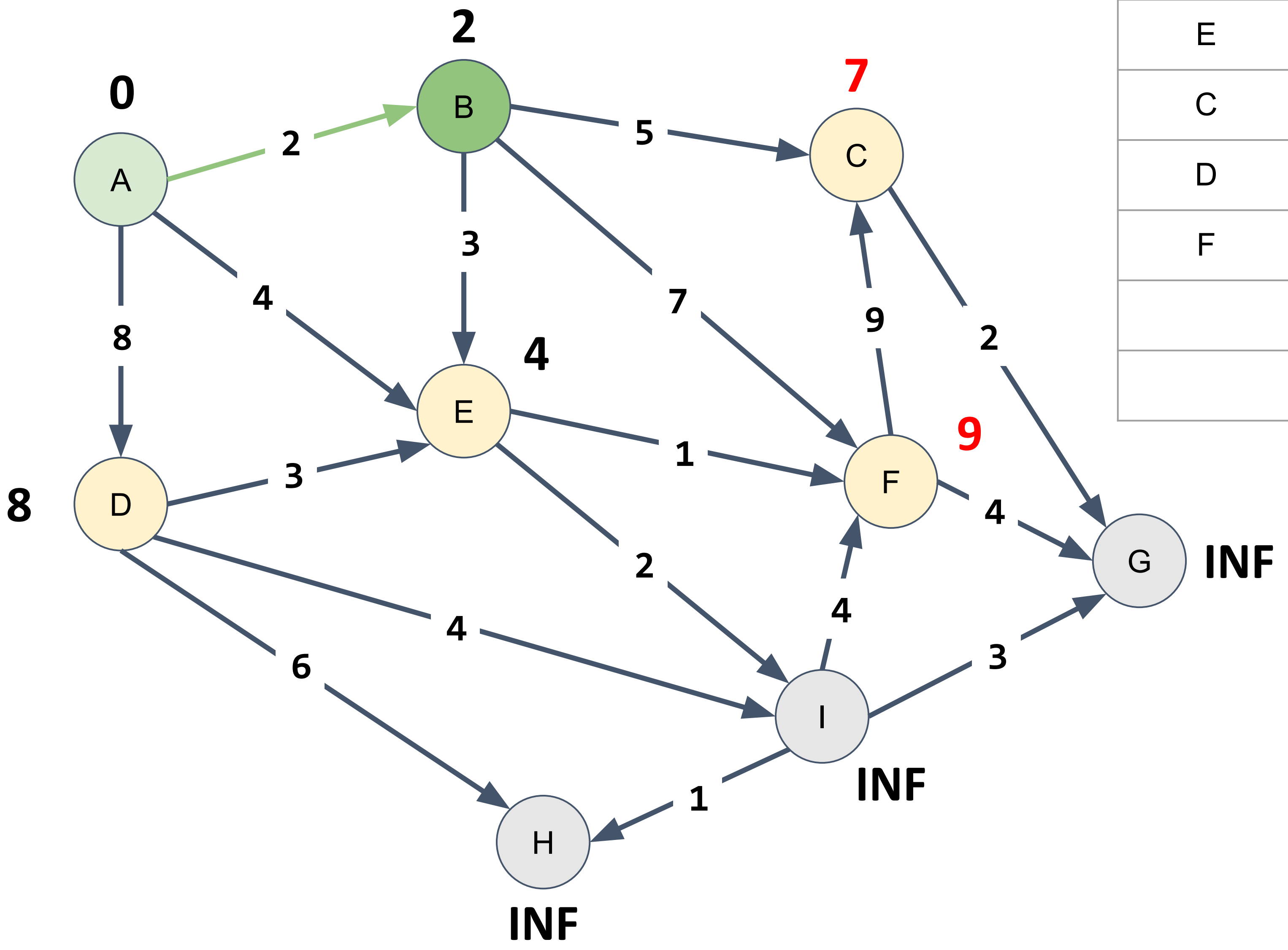
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
B	2
E	4
D	8

Dijkstra — Algorithm

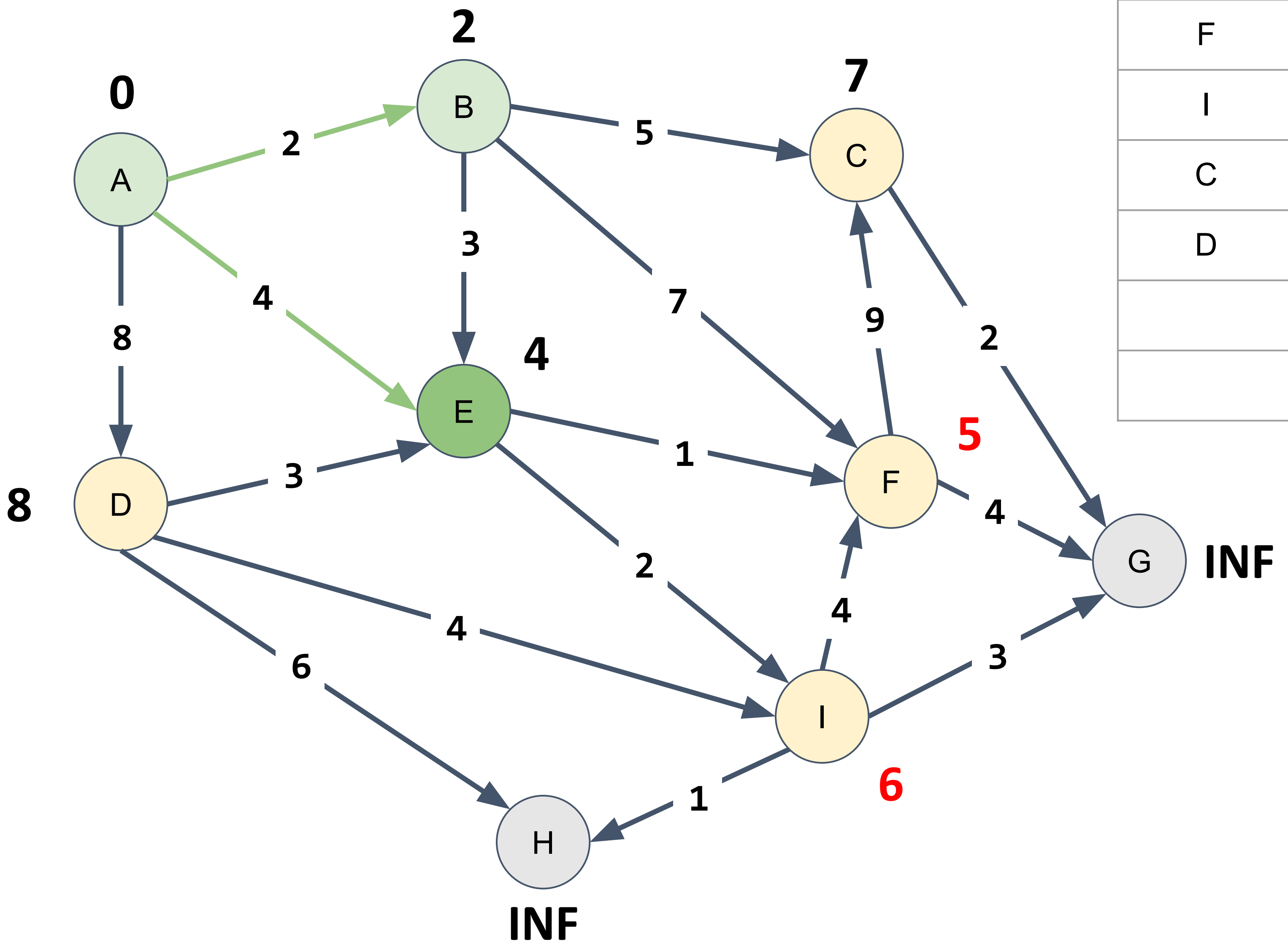
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
E	4
C	7
D	8
F	9

Dijkstra — Algorithm

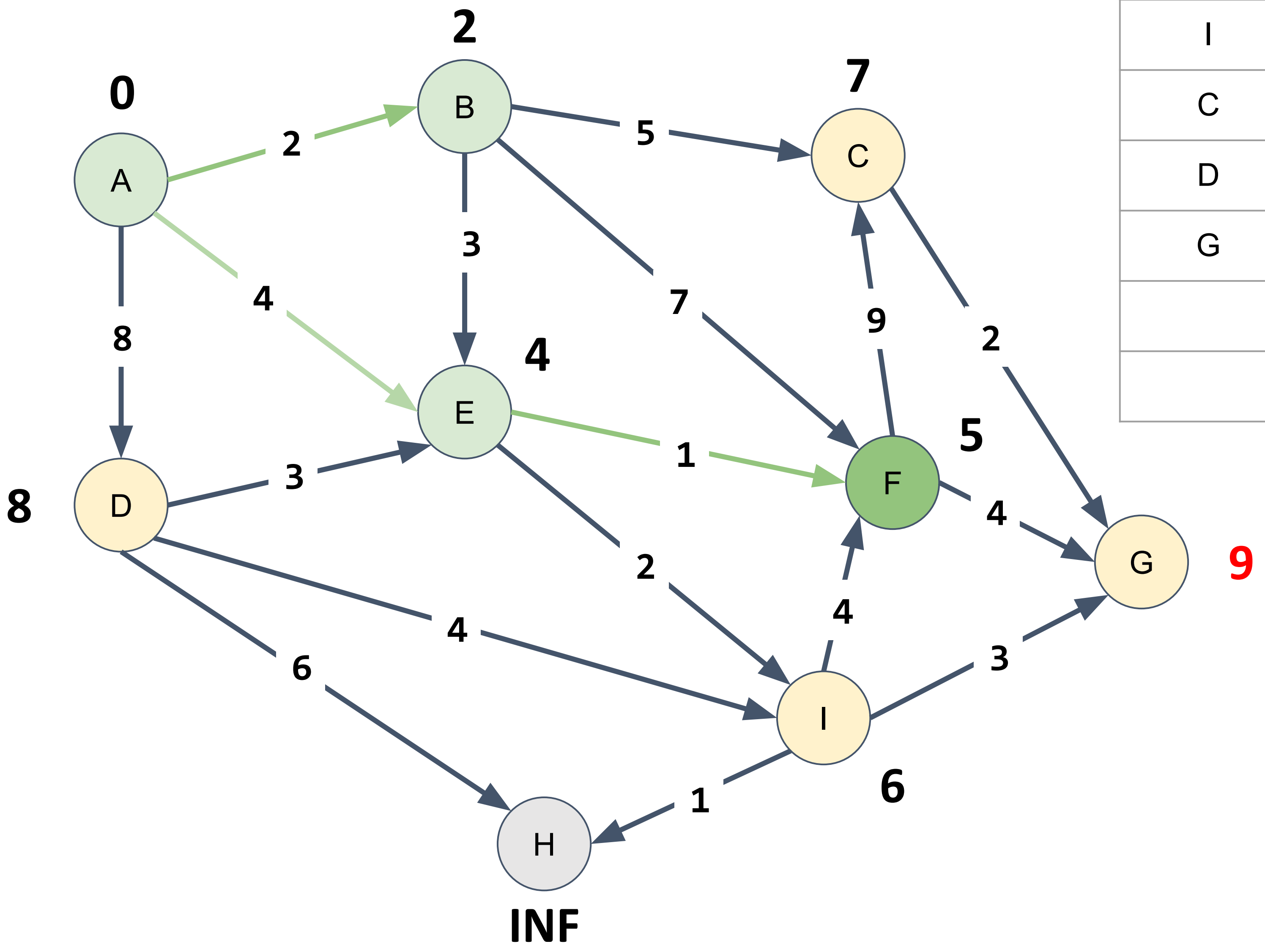
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
    a. Mark it as **resolved**
    b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
F	5
I	6
C	7
D	8

Dijkstra — Algorithm

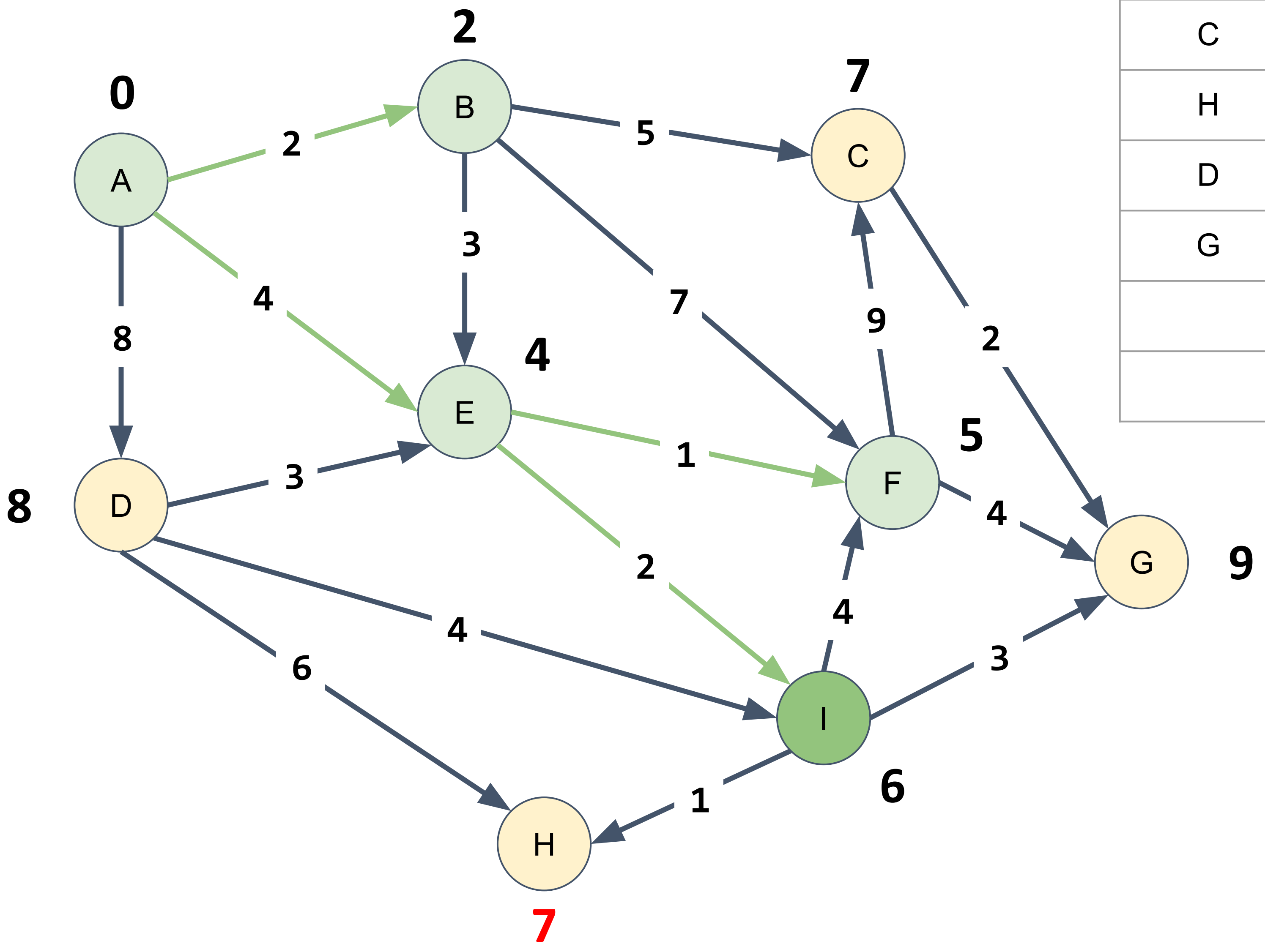
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
    a. Mark it as **resolved**
    b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
I	6
C	7
D	8
G	9

Dijkstra — Algorithm

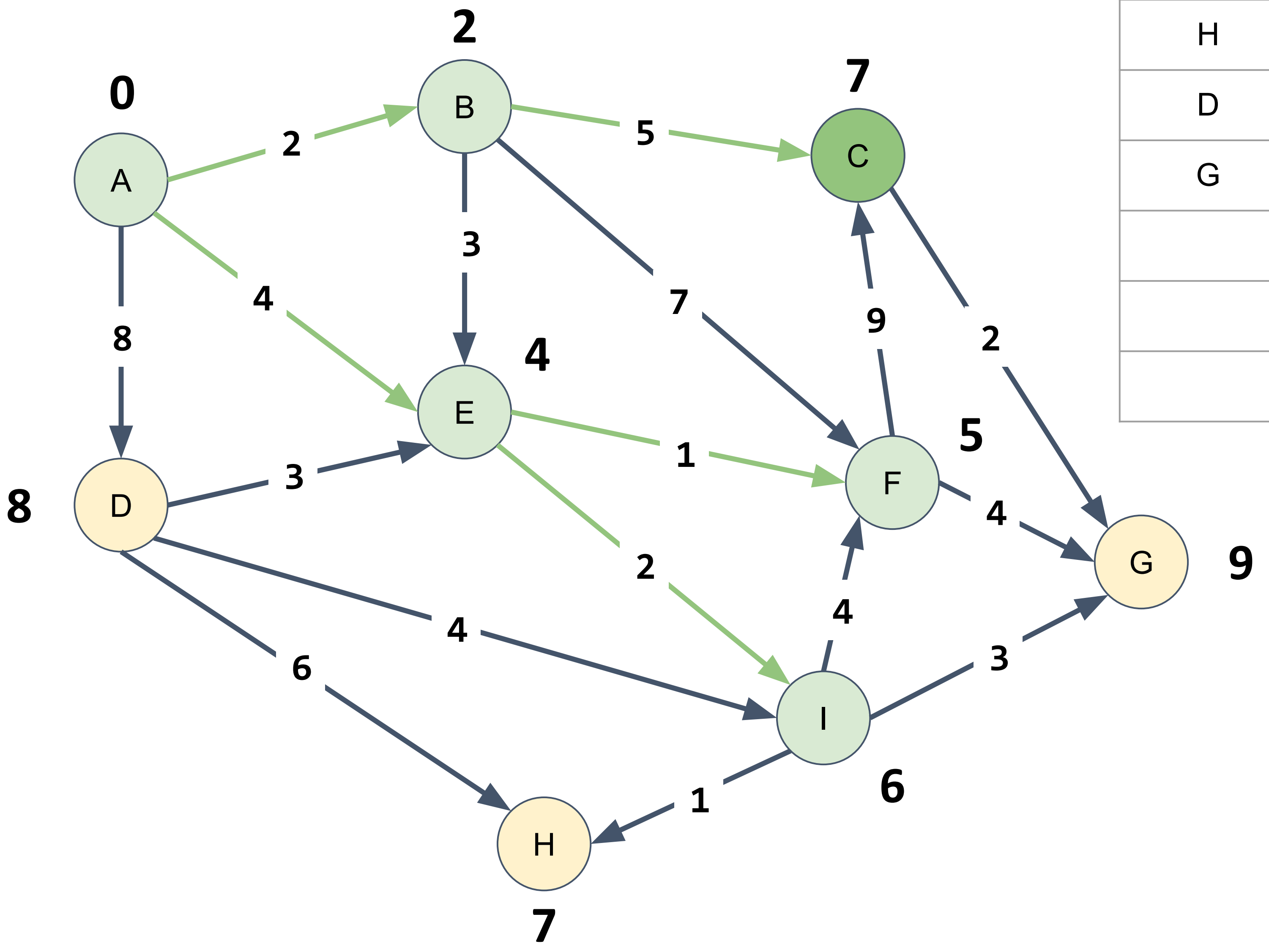
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
    a. Mark it as **resolved**
    b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
C	7
H	7
D	8
G	9

Dijkstra — Algorithm

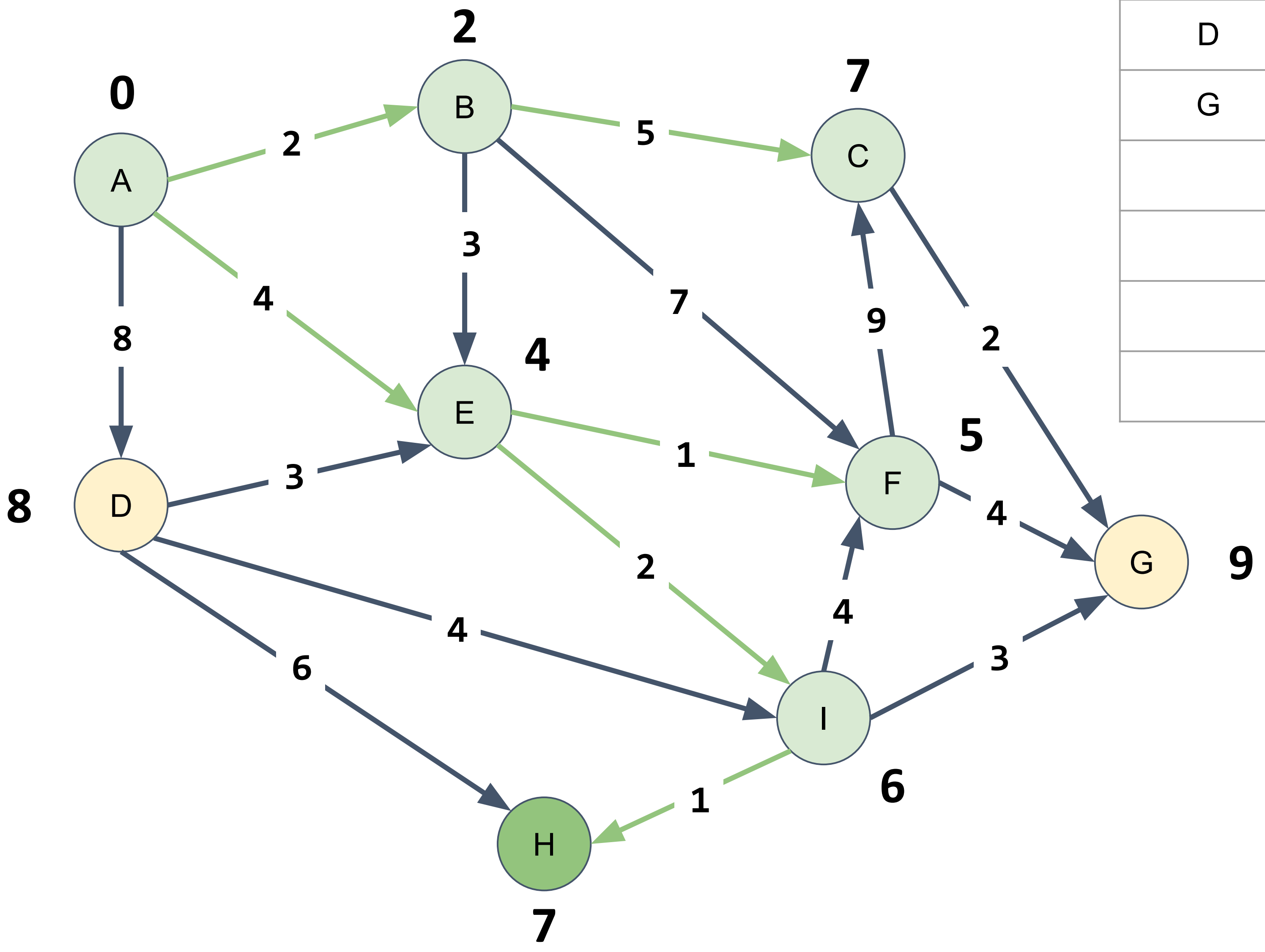
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
H	7
D	8
G	9

Dijkstra — Algorithm

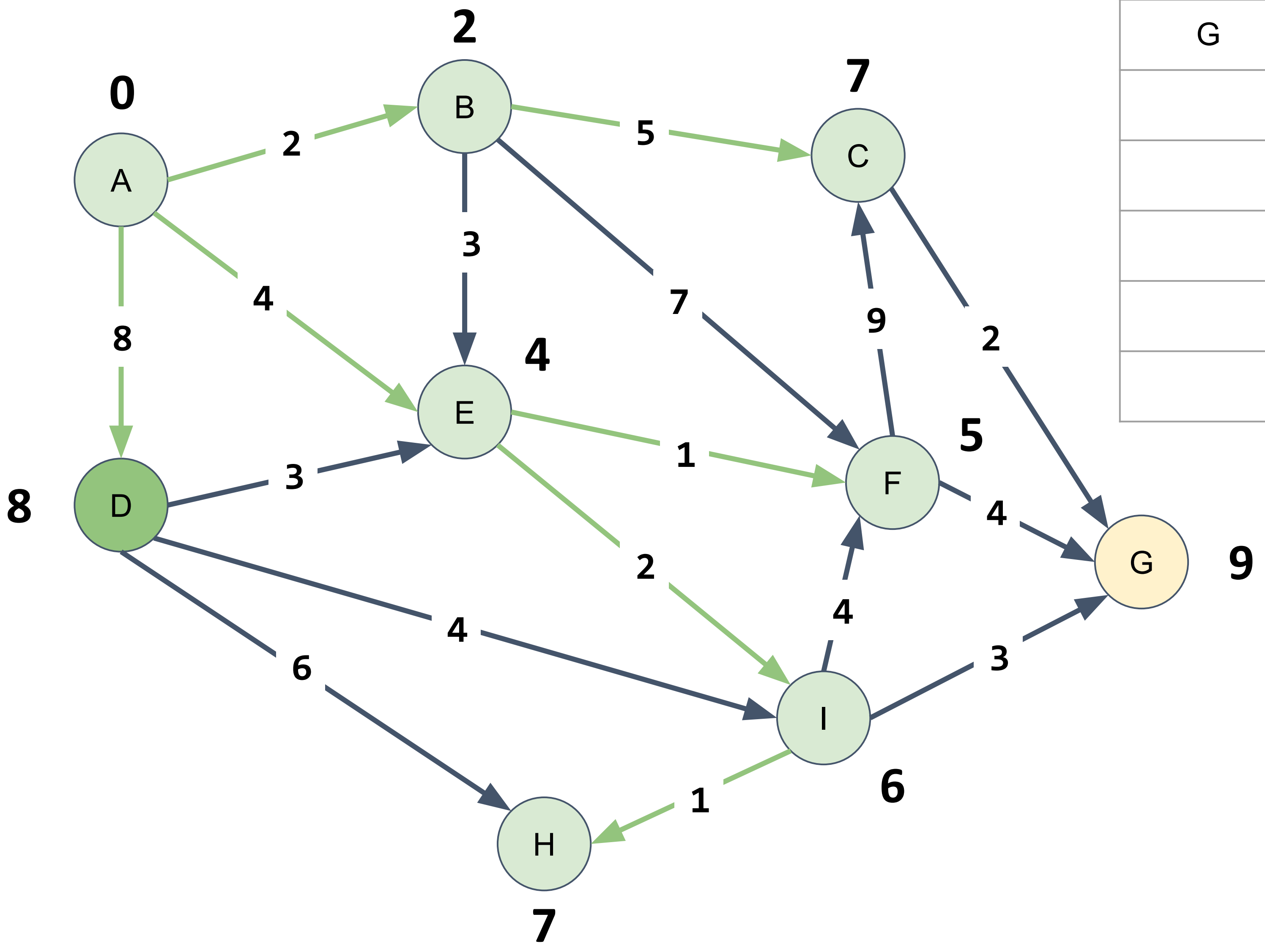
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \to v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
D	8
G	9

Dijkstra — Algorithm

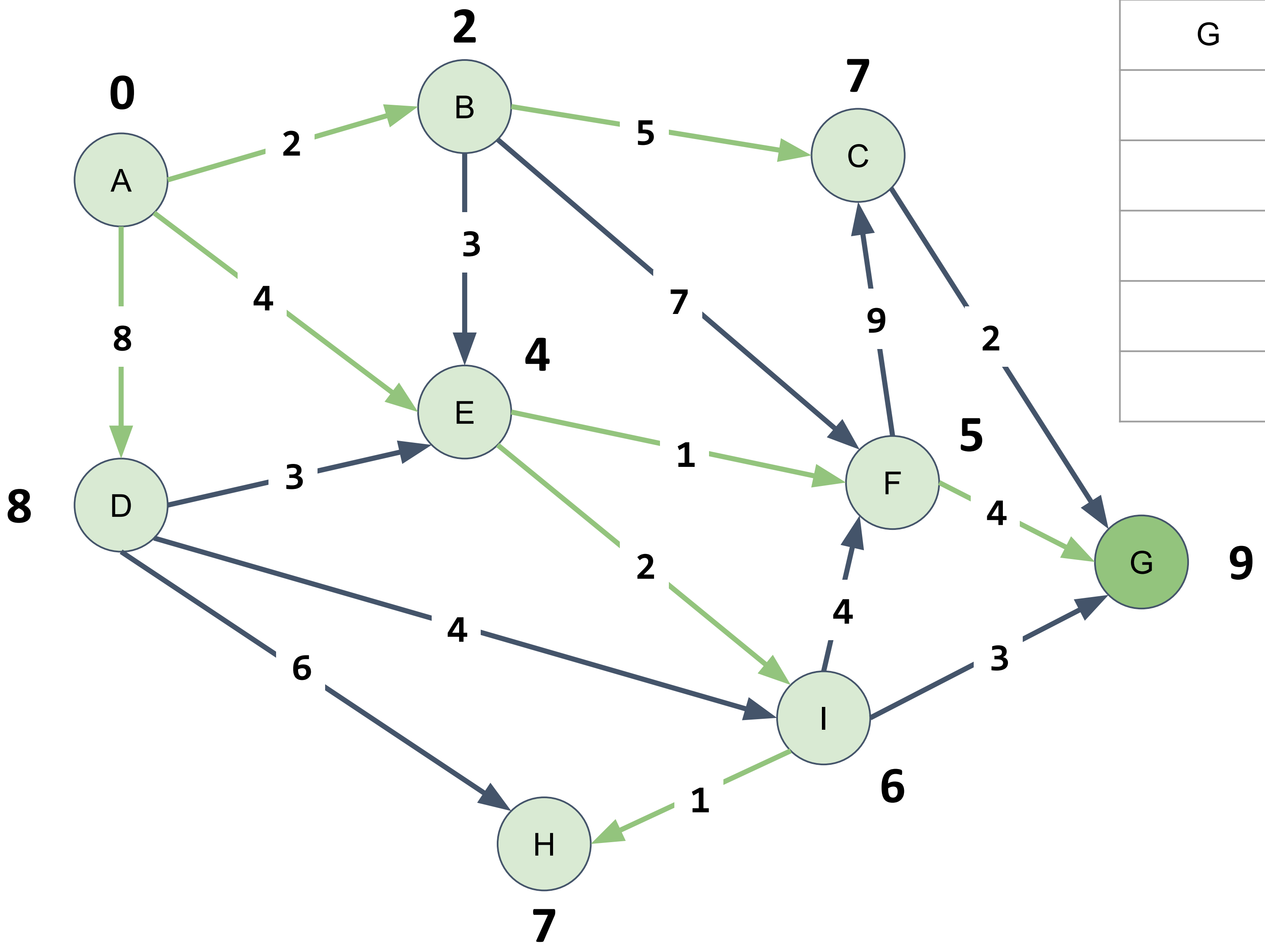
1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
    a. Mark it as **resolved**
    b. For all its neighbours v, relax v with $u \to v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
G	9

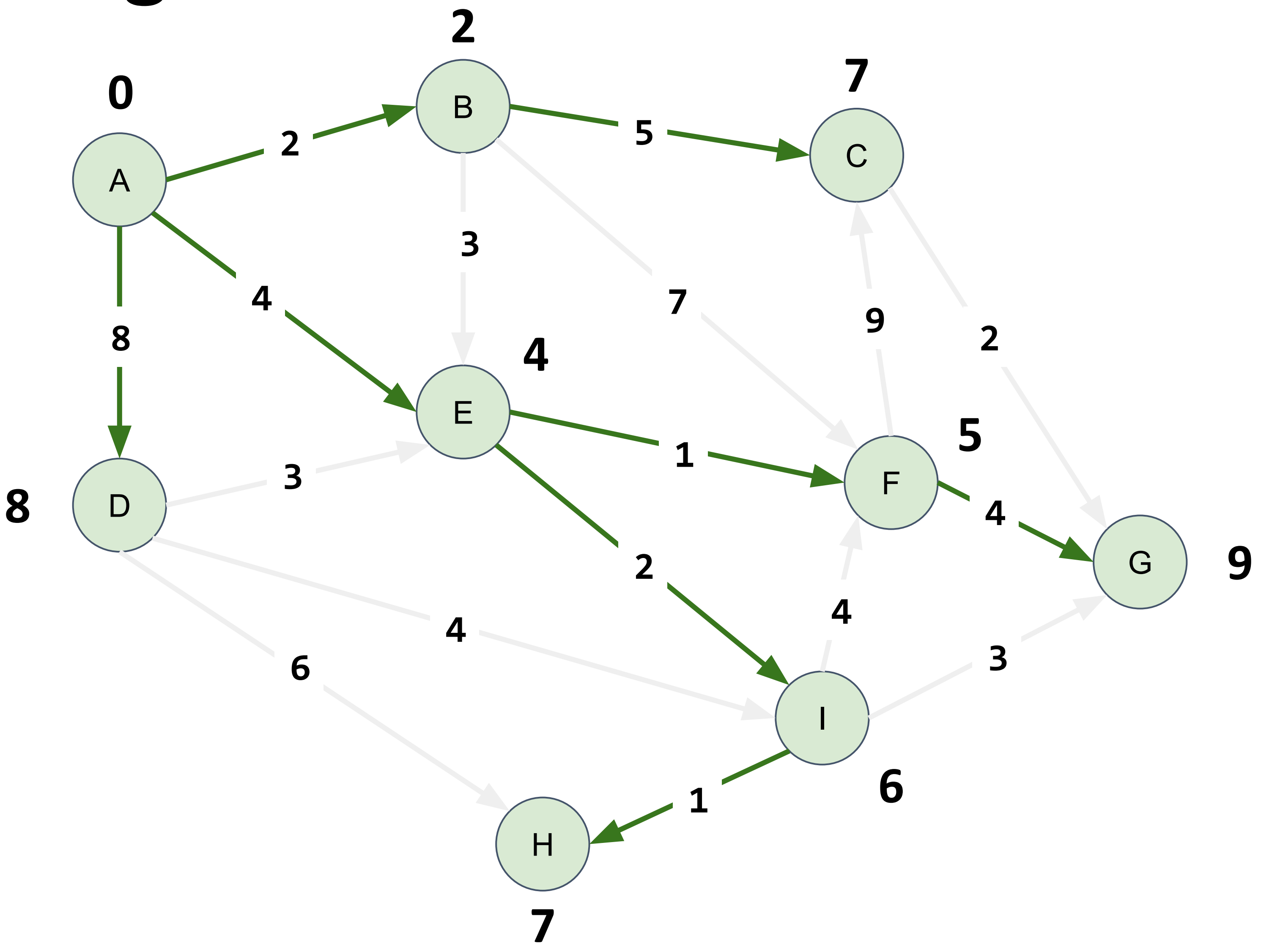
Dijkstra — Algorithm

1. Initialize distances of all vertices to be infinity, except the source vertex's which is trivially 0
2. Mark all vertices to as "unresolved"
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \rightarrow v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph



Vertex	Distance
G	9

Dijkstra — Algorithm



Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection?

$O(N \log N)$ sort every time? That would take

$$O(V \log V) + O((V-1) \log (V-1)) + \ldots + O(2 \log 2) + O(1 \log 1)$$

$$= O(V^2 \log V)$$

Which is pretty bad! :O

Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection **in an efficient manner**?

Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection **in an efficient manner**?

Answer: We could use a data structure!

Dijkstra—How to be greedy?

Which ADT do we have that maintains ordered data and what data-structure(s) can be used to implement it?

Dijkstra — How to be greedy?

Which ADT do we have that maintains ordered data and what data-structure(s) can be used to implement it?

Answer: Priority Queue ADT!

Can be implemented via Heap or BBST.

Original Dijkstra — Implementation

```
ArrayList<ArrayList<Edge>> AL; // Adjacency list of (vertex, weight) pairs
int V, source;

/* Declarations and initializations */
ArrayList<Boolean> resolved;                    // initialize to false
ArrayList<Boolean> dist;                        // initialize to INF
dist[source] = 0;                               // Assign source to distance 0
MinPriorityQueue q = MinPriorityQueue();        // Instantiate custom Priority Queue
for (int u = 0; u < V; u++)                     // Initialize q with (distance,vertex)
    q.enqueue({D[u], u});
```

Continued on next slide

Original Dijkstra — Implementation

```
While (!q.empty()){
    Edge e = q.poll();
    int D_u = e.distance, u = e.vertex;
    resolved[u] = true;                     // Mark current as resolved
    /* Iterate each edge of u */
    for (Edge n : AL[u]) {                // v: neighbour, w: weight
        int v = e.v, D_v = D_u + e.w;     // New distance to check
        if (!resolved[v] && D_v < D[v]) { // If can relax
            q.update_key({D[v], v}, {D_v, v}); // Update key
            D[v] = D_v;                        // Relax u→v
        }
    }
}
```

Dijkstra — Implementation woes

Realize we would need to call `update_key(...)` whenever a vertex is relaxed.

If we don't have a custom implemented Min Priority Queue supporting that operation, then we're in a bit of trouble! This is because neither C++ nor Java Heap/BBST supports `update_key()`! So in practice, if using:

- Library BBST: Find vertex data with lowest distance, remove invalid vertex data and insert new vertex data with updated distance
- Library Heap: Lazy removal! :O

Original Dijkstra — "Lazy removal"

```
ArrayList<ArrayList<Edge>> AL; // Adjacency list of (vertex, weight) pairs
int V, source;

/* Declarations and initializations */
ArrayList<Boolean> resolved;                // initialize to false
ArrayList<Boolean> dist;                    // initialize to INF
dist[source] = 0;                           // Assign source to distance 0
PriorityQueue<Pair> q = new PriorityQueue<>(); // We will remove lazily
for (int u = 0; u < V; u++)                 // Initialize q with (distance,vertex)
    q.enqueue({D[u], u});
```

Continued on next slide

Self-read

Original Dijkstra — "Lazy removal"

```
while (!q.empty()){
    Edge e = q.poll();
    int D_u = e.distance, u = e.vertex;
    if (resolved[u]) continue;                // Lazy removal
    resolved[u] = true;                       // Mark current as resolved
    for (Edge n : AL[u]) {                    // v: neighbour, w: weight
        int v = e.v, D_v = D_u + e.w;         // New distance to check
        if (!resolved[v] && D_v < D[v]) {     // If can relax
            q.push({D_v, v}); // Update key
            D[v] = D_v;                            // Relax u→v
        }
    }
}
```

Adapted from materials for NOI 2015 Dec Training Team

Original Dijkstra—Looks familiar?

Realize that Dijkstra’s algorithm is indeed just a few simple modifications of BFS:

1. Swapping out the Queue in BFS with a Min-Priority Queue
2. Resolved table is just visitation table in BFS
3. Update a neighbour’s distance if it can be relaxed via the edge

Original Dijkstra — Time complexity

Each time we dequeue a vertex from the min-heap, we incur $O(\log V)$ time. Since the algorithm terminates when the min-heap is empty, the total time complexity due to dequeuing every vertex from the heap is $O(V \log V)$

Each time we dequeue a vertex, we will potentially relax all its edges. When we relax an edge, we have to call `update_key` in the heap which incurs $O(\log V)$ time. Since we will potentially relax all E edges in the graph when the algorithm terminates, the total time complexity due to all relaxations is $O(E \log V)$.

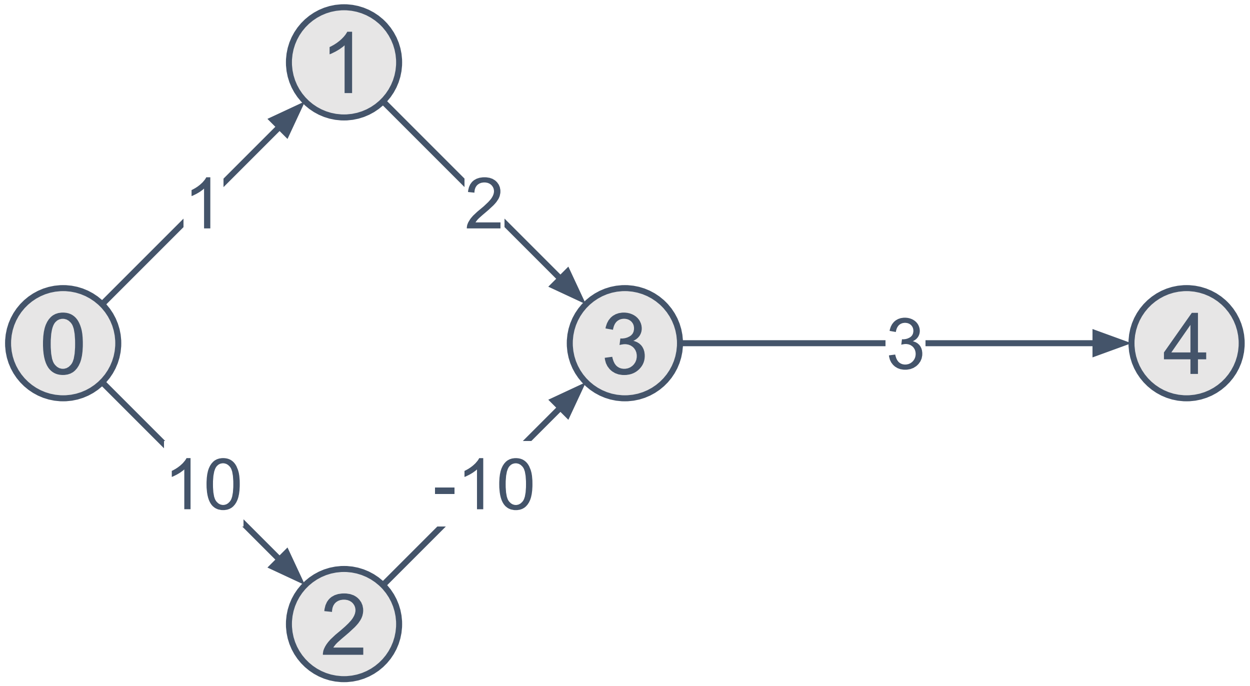
Hence total time complexity is $O(V \log V) + O(E \log V) = O((V + E) \log V)$

Original Dijkstra — A problem

What happens when we run original Dijkstra on the following graph that has a negative weighted edge?

It's a good exercise to work it out by hand!

Verify on VisuAlgo using Example Graph "CP3 4.18 -ve weight"



Modified Dijkstra

We observe that each dequeued vertex must be allowed to be relaxed again in the future, so we need to make the following changes:

- Do away with resolved/visited table. i.e. until the algorithm has terminated, we cannot determine if a vertex’s distance is finalized
- Enqueue a neighbour v whenever it can be relaxed with the current edge, regardless if v has been visited before
- No longer need to enqueue all distances at the beginning since we have a new propagation criteria
- Lazy removal of invalid vertex distances in queue. Invalid when distance table has a lower value

Modified Dijkstra — Implementation

```
ArrayList<ArrayList<Edge>> AL; // Adjacency list of (vertex, weight) pairs
int V, source;

/* Declarations and initializations */
ArrayList<Boolean> resolved;                 // initialize to false
ArrayList<Boolean> dist;                     // initialize to INF
dist[source] = 0;                            // Assign source to distance 0
PriorityQueue<Pair> q = new PriorityQueue<>(); // We will remove lazily
q.push({0, source});                         // Enqueue source only
```

Continued on next slide

Modified Dijkstra—Implementation

```
while (!q.empty()){
    Edge e = q.poll();
    int D_u = e.distance, u = e.vertex;
    if (D_u > D[u]) continue;                    // relaxed with smaller distance already
    for (Edge n : AL[u]) {                       // v: neighbour, w: weight
        int v = e.v, D_v = D_u + e.w;            // New distance to check
        if (!resolved[v] && D_v < D[v]) {        // If can relax
            q.push({D_v, v});                    // Update key
            D[v] = D_v;                          // Relax u→v
        }
    }
}
```

Test yourself!

What happens if we **removed** this line in lazy implementation of (un)modified Dijkstra:

```
if (D_u > D[u]) continue;
```

Test yourself!

What happens if we **removed** this line in lazy implementation of (un)modified Dijkstra:

```
if (D_u > D[u]) continue;
```

Answer: It degenerates into Bellman-Ford by making many redundant checks! If a vertex `u` has an invalid (i.e. outdated) distance `D_u` in PQ, we can skip relaxation checks for all its outgoing edges since all of its neighbours must now have distances shorter than `D_u + w`.

Test yourself!

What if we also want the actual shortest **path taken**, from source to every vertex in the graph? How might we modify the algorithms?

Test yourself!

What if we also want the actual shortest **path taken**, from source to every vertex in the graph? How might we modify the algorithms?

Answer: Maintain a parent/predecessor/previous table P that also gets updated during every vertex's relax operation.

See Kattis problem Flowery Trails.

Facebook Privacy Setting

CS2010 Finals AY2013/2014 Sem 1

$O(V+E)$ solution

$O(k)$ solution

Problem statement

- You have a friendship graph of V vertices, E edges
- You are given a pair of vertices i and j.
  - Compute whether vertex i and j are at most 2 edges apart. i.e. degree of separation is less than 2
- Given: The friend list of profile i is stored in `adjList[i]` and sorted ascending
- $O(V+E)$ for 7 marks
- $O(k)$ for 19 marks
  - k is sum of number of adjacent vertices of i and j

How to get $O(V+E)$ solution?

Bellman Ford?

- $O(VE)$

Dijkstra?

- $O((V+E) \log V)$

$O(V + E)$ solution

BFS is enough since edges are unweighted!

- Start from vertex i, compute shortest path from i to every other vertex.
- Check if distance(i, j) ≤ 2.
- Can be further improved by only expand till distance≤ 2, then check if j is reached. This is still worst case O(V+E).

Can we do better?

Observation

We only need $\text{distance}(i, j) \leq 2$

3 cases:

1. $\text{distance}(i, j) = 0$ $i = j$
2. $\text{distance}(i, j) = 1$ j is a neighbour of i
3. $\text{distance}(i, j) = 2$ j is a neighbour of a neighbour of i

Case 1: $\mathrm{distance}(i, j) = 0$

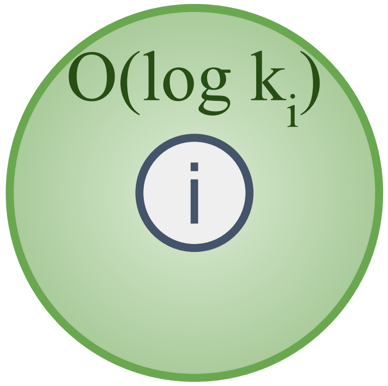
This is trivial, just $O(1)$ to check if $i = j$

Case 2: $\mathrm{distance}(i, j) = 1$

To check if j is a neighbour of i, we search for j in each of i's k_i neighbours.

Since given that friends list is sorted, binary search will take $O(\log k_i)$

Is j within green?



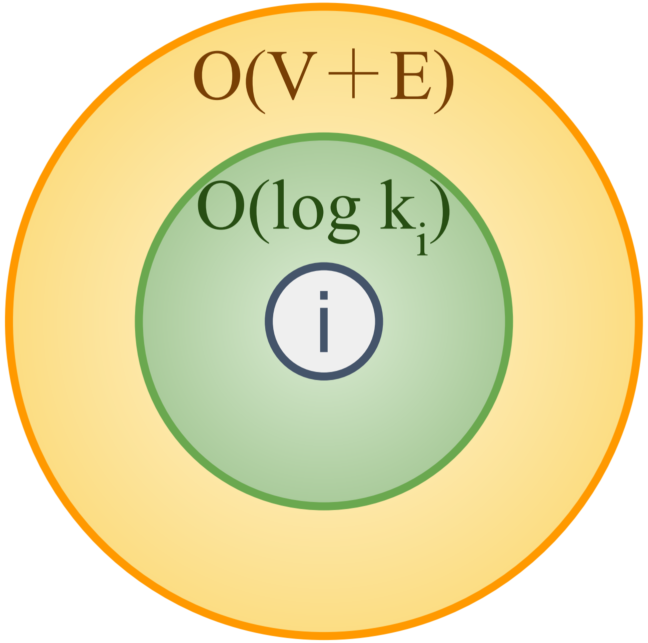
Green: $\mathrm{distance} = 1$

Case 3: $\mathrm{distance}(i, j) = 2$

To check if j is a neighbour of a neighbour of i, we potentially traverse the entire graph and so this will take $\mathrm{O}(\mathrm{V}+\mathrm{E})$ in worst case.

Can we do better than this?

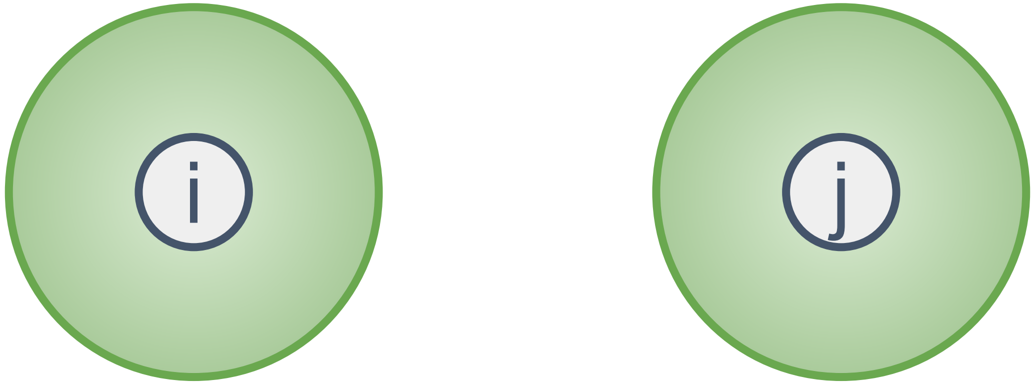
Is j within orange?



Green: $\mathrm{distance} = 1$
Orange: $\mathrm{distance} = 2$

Case 3: $\text{distance}(i, j) = 2$

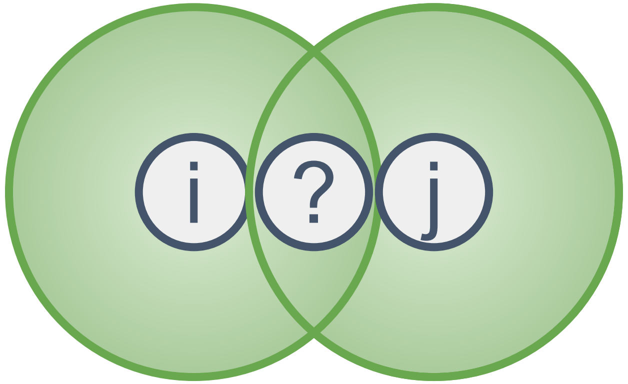
If vertex i and j are distance 2 away from each other, what do you realize?



Case 3: $\text{distance}(i, j) = 2$

If vertex i and j are distance 2 away from each other, what do you realize?

Answer: There must be at least one common friend!



Transformed Problem

Given 2 integer arrays with sizes up to N each, find whether there exist an integer that is in both arrays.

Approaches:

1. For every number in one array, search for a correspondence in the other array $O(N^2)$
2. Sort both arrays $O(N \log N)$, then iterate through both arrays with 2 pointers to look for a corresponding pair of numbers $O(N)$

Facebook Privacy Setting

Realize that the transformed problem is essentially what we want to solve in the original problem!

Solution 2 (previous slide) would just take $O(k)$ in the original problem since we are already given that the friends list is sorted in the adjacency list! We just need to iterate down the friends list of i and j once!

* Hashtable can do the trick too since insertion and check are all O(1), but linear scan using 2 pointers just like merge step in merge sort is better since AL is sorted.

SSSP on “SLL”

CS2040 AY1718 Sem 4

C.1

You are given a Singly Linked List of **N** nodes labelled from 0 to **N-1**. They are linked with edges of **weight 1**.
You are also given an integer **S**.

For every **i, j > 0** that **i % S == 0** and **j % S == 0**, there is an *extra* edge with **weight 2**.

C.1

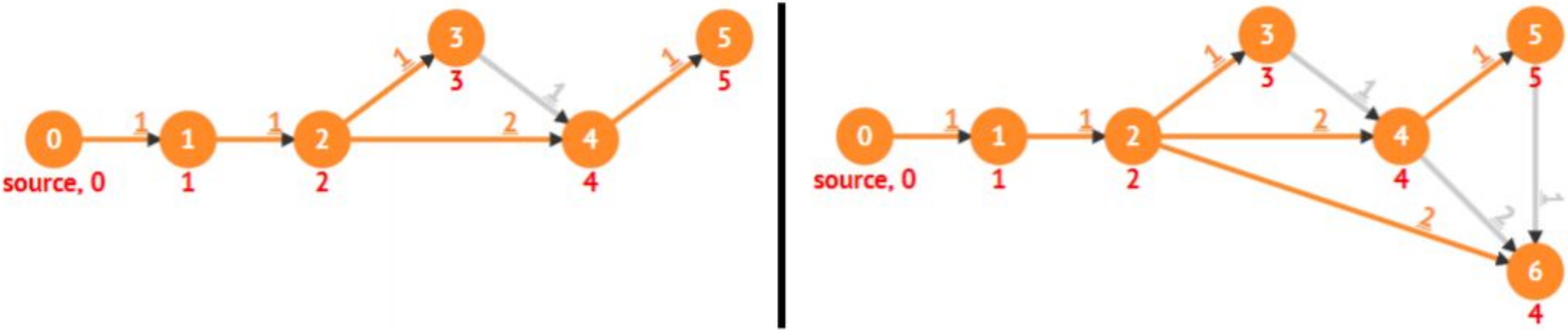


Figure 1: Left picture: $n = 6$, $S = 2$, shortest path value $= 5$ (via $0 \to 1 \to 2 \to 4 \to 5$ or $0 \to 1 \to 3 \to 4 \to 5$); Right picture: $n = 7$, $S = 2$, shortest path value $= 4$ (only via $0 \to 1 \to 2 \to 6$).

C.1

Given **N** and **S**, one can construct the ‘graph’.

Task: Find shortest path from vertex 0 to vertex **N-1**.

C.1

Naive Solution

Construct the graph and run Dijkstra’s algorithm/Bellman Ford.

Can have up to $\mathbf{O(N^2)}$ edges if **S = 2**.

Time complexity: slower than $\mathbf{O(E)}$ = $\mathbf{O(N^2)}$

C.1

Observation

If there are no extra edges, the answer is obvious: **N-1**.

How would the extra edges affect our answer?

C.1

Observation

If there are no extra edges, the answer is obvious: **N-1**.

How would the extra edges affect our answer?

We will never use two extra edges.

Proof by contradiction: If we use a -> b and c -> d extra edges then a, b, c, d all are divisible by S. Therefore, we could have used a -> d instead, which is more optimal.

C.1

Where is the earliest possible "extra edge"?

Lowest vertex number **i > 0** such that **i % S == 0**.

→ Vertex **S**

Where can it lead to?

Highest vertex number **0 < j < N** such that **j % S == 0**.

→ Vertex **N - 1 - (N-1 % S)**

It takes cost 2 to get from vertex **S to (N-1-(N-1%S))**.

C.1

Cost to get from **0** to **S**: **S-1**

Cost to get from **S** to **[(N-1) - (N-1)%S]**: **2**

Cost to get from **[(N-1) - (N-1)%S]** to **N-1**: **(N-1)%S**

To get from 0 to N-1 using ‘extra edges’:

= (S-1) + 2 + (N-1)%S

= **S + 1 + (N-1)%S**

C.1

Solution

Without using special edges: **N-1**

Using special edges: **S + 1 + (N-1)%S**

Choose the *minimum* of the two options!

Time complexity: O(1)

Online Quiz

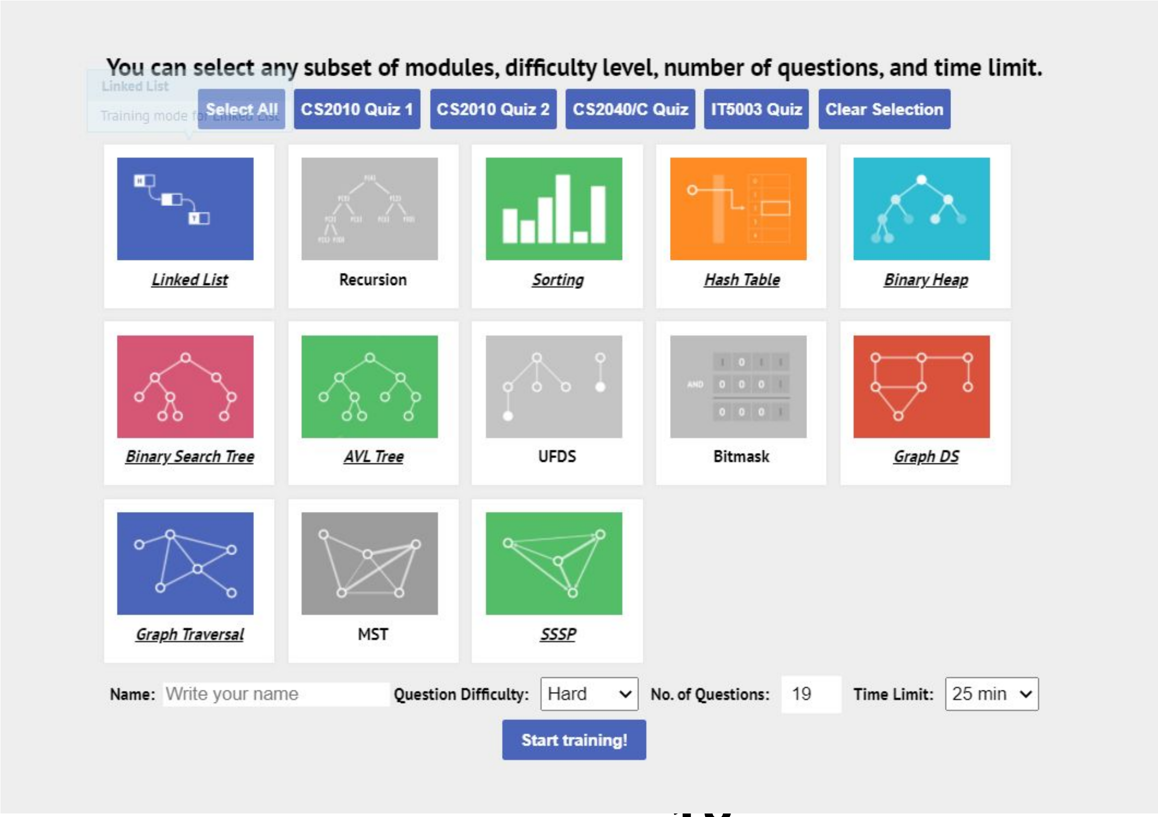
General Instruction

Hard Problems

Quiz Configuration

Material: all CS2040S topics excluding MST.

15 (hard) questions from [VisuAlgo](https://visualgo.net)



18

Some sample quiz questions

DAG questions

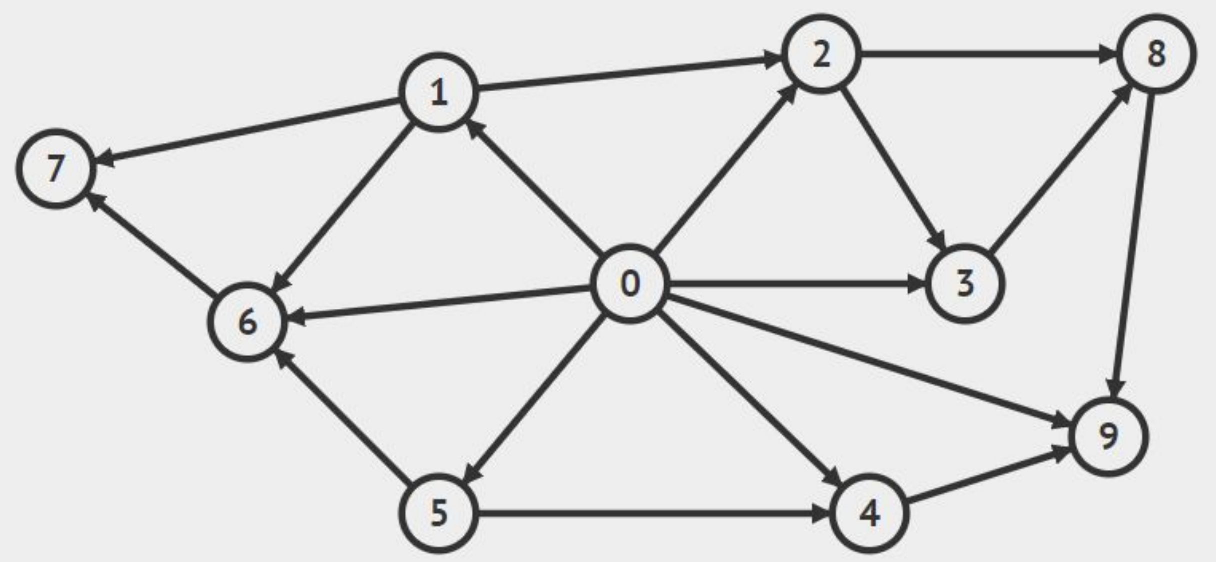
Select the topo sort

"Manual topological sort" / Kahn's Algorithm

1. In any order, identify vertices with no *incoming edges*
2. Add them to toposort
3. Remove them and all the edges they have
4. Repeat steps 1-3 until no more vertices left in graph

DAG questions

3. Click the sequence of vertices such that when all the outgoing edges of these vertices are relaxed in this order using One-Pass Bellman-Ford's algorithm, the SSSP problem can be solved in O(V+E) time.

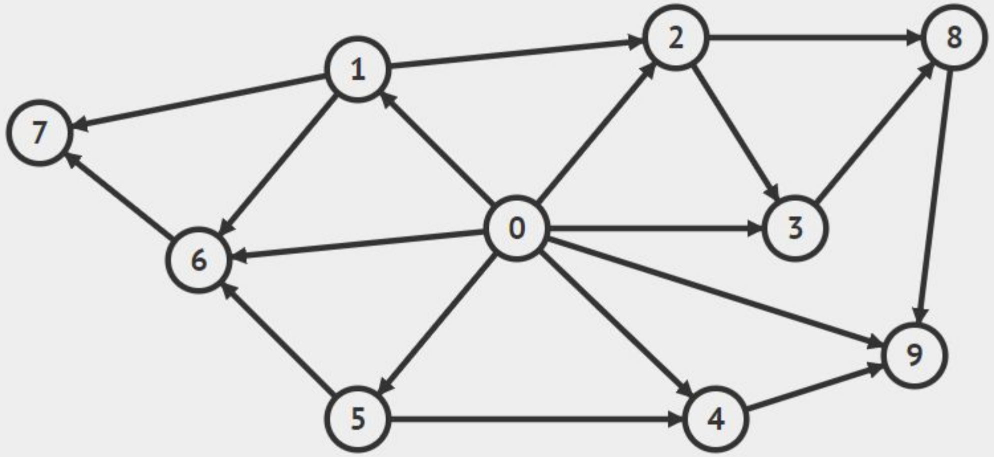


This question is indirectly asking you for the topological order:

[0, 1, 5, 6, 7, 2, 3, 4, 8, 9]

DAG questions

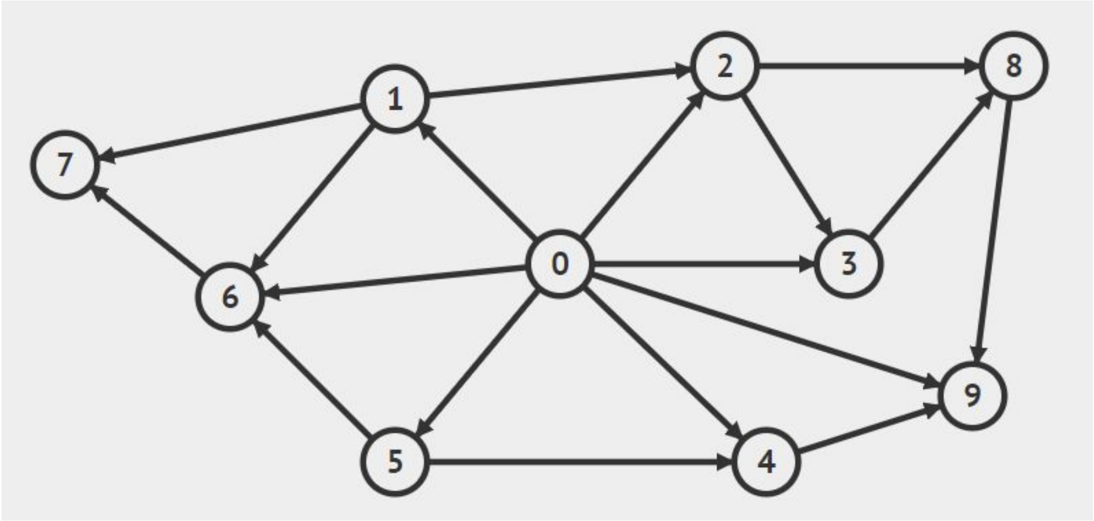
4. In the following Directed Acyclic Graph, how many simple paths are there from vertex **0** to **9**?



DAG questions

- Do in topological or reverse topological order
- Source/Destination Vertex

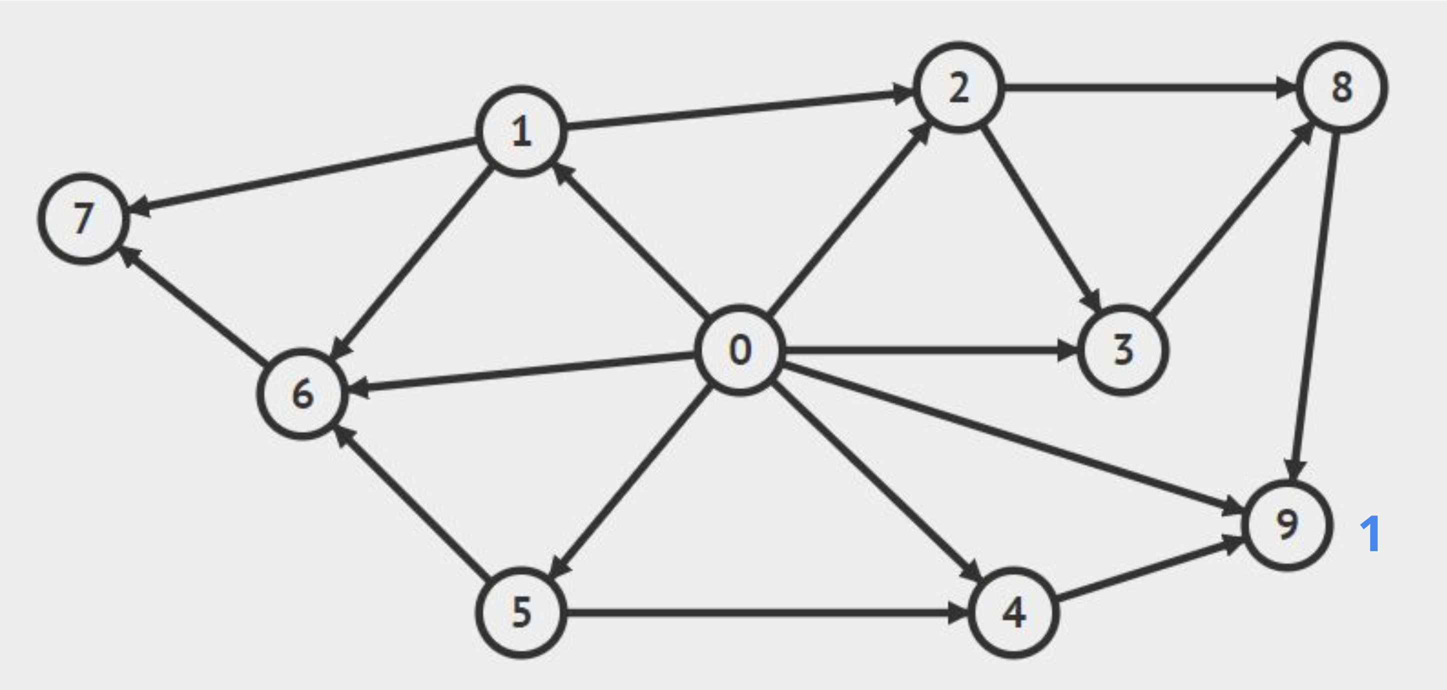
[0, 1, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

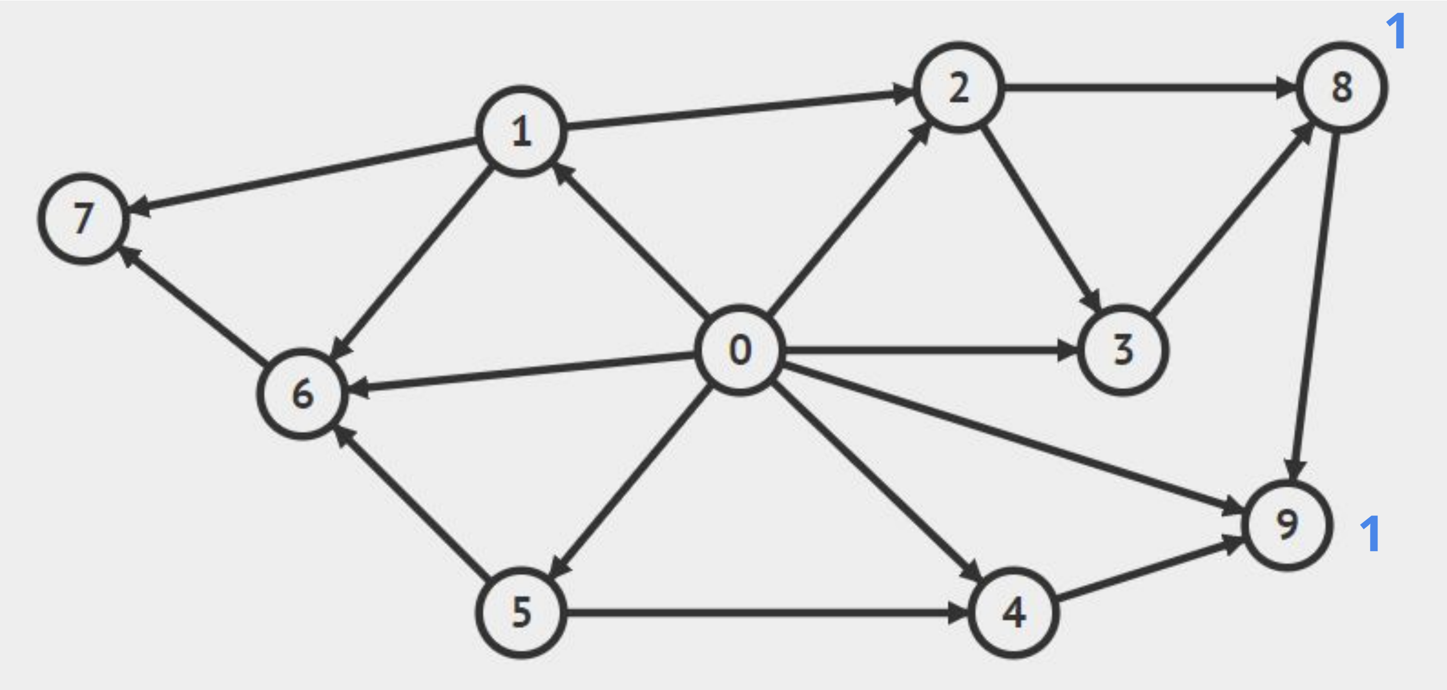
[0, 1, 5, 6, 7, 2, 3, 4, 8, **9**]



DAG questions

To count number of paths, we start at dest vertex

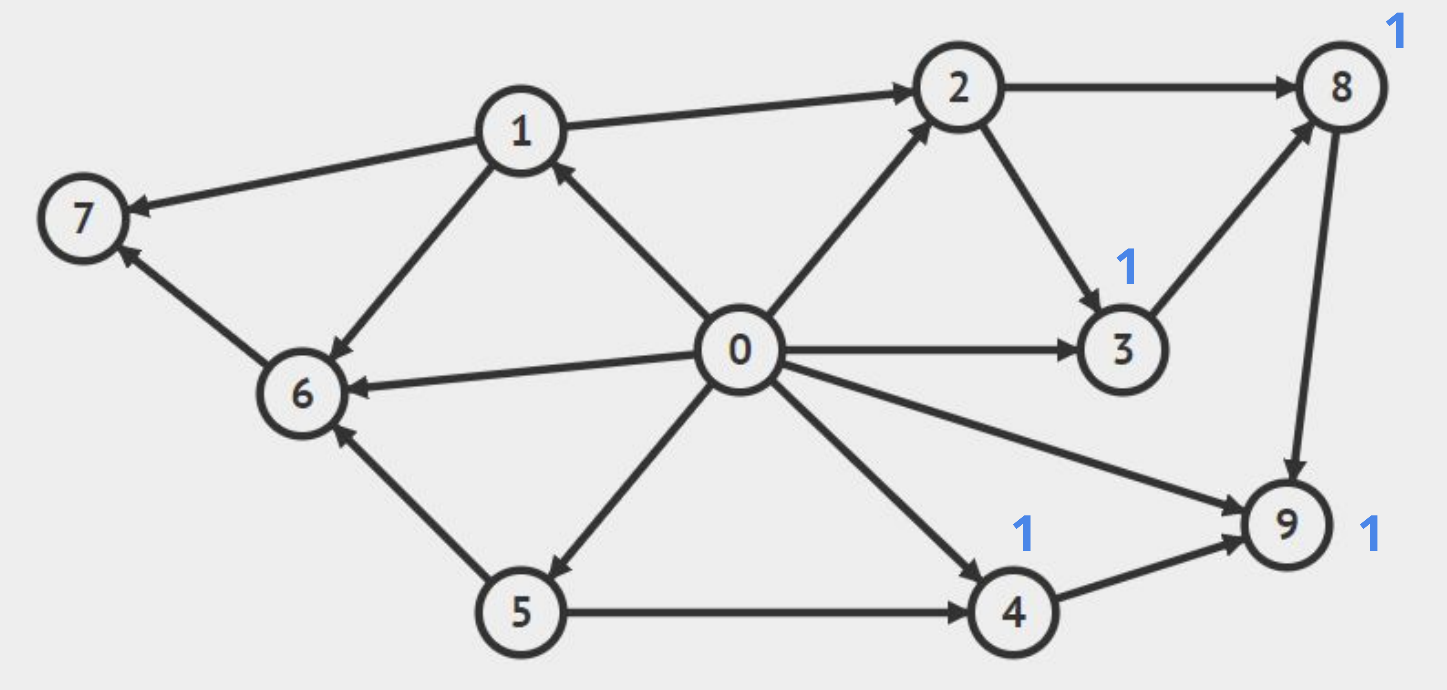
[0, 1, 5, 6, 7, 2, 3, 4, **8**, 9]



DAG questions

To count number of paths, we start at dest vertex

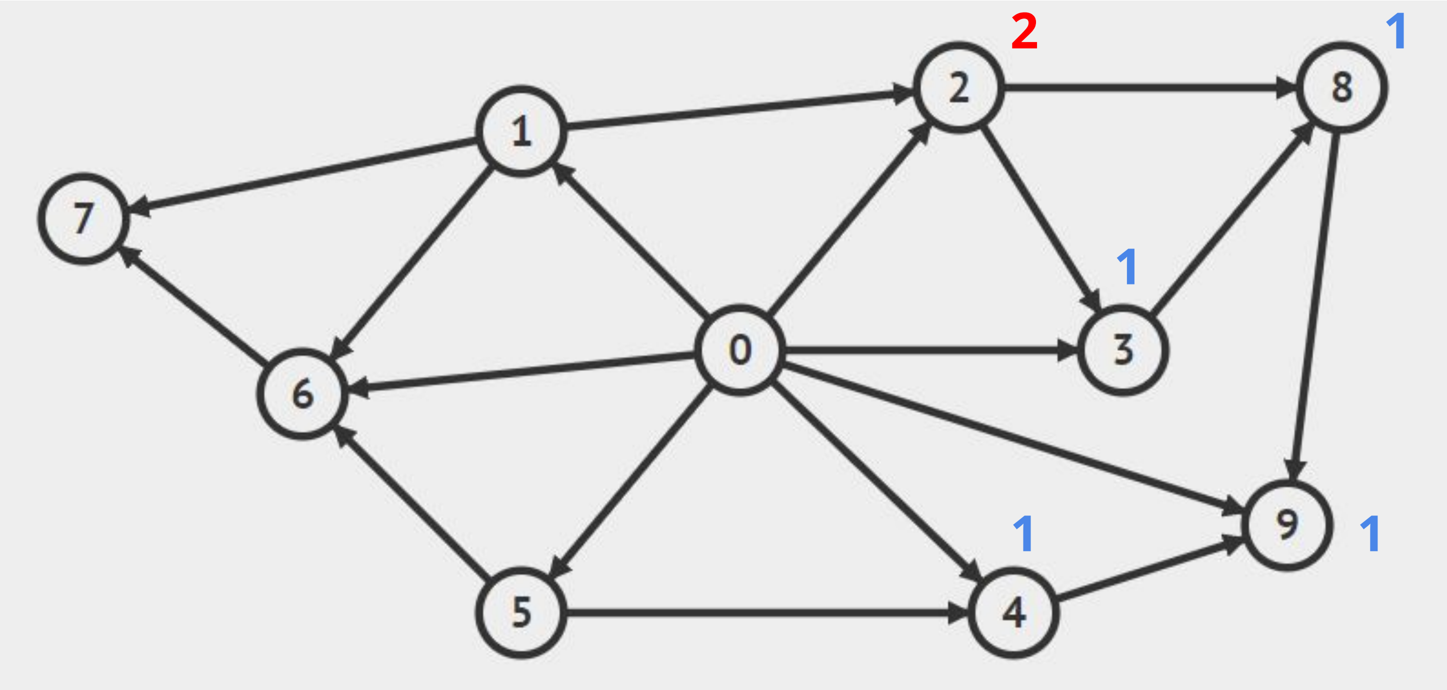
[0, 1, 5, 6, 7, 2, **3, 4**, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

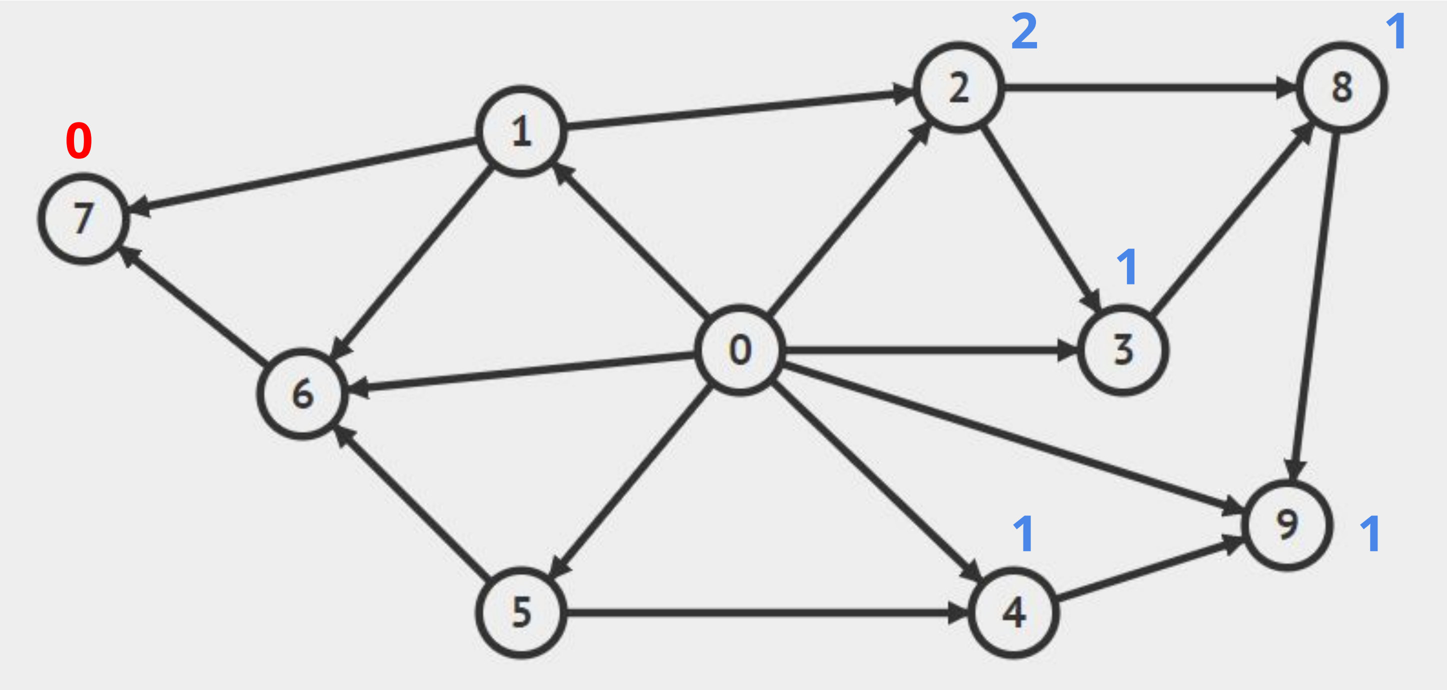
[0, 1, 5, 6, 7, **2**, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

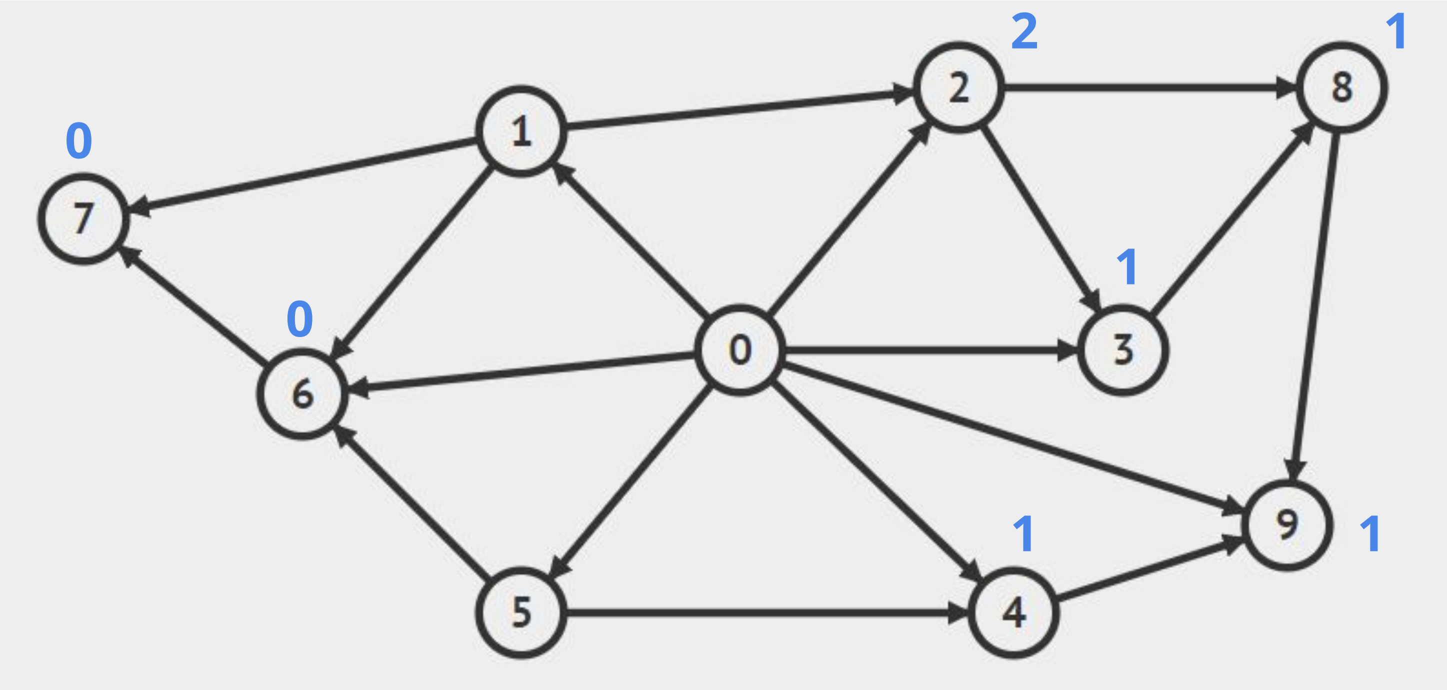
[0, 1, 5, 6, **7**, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

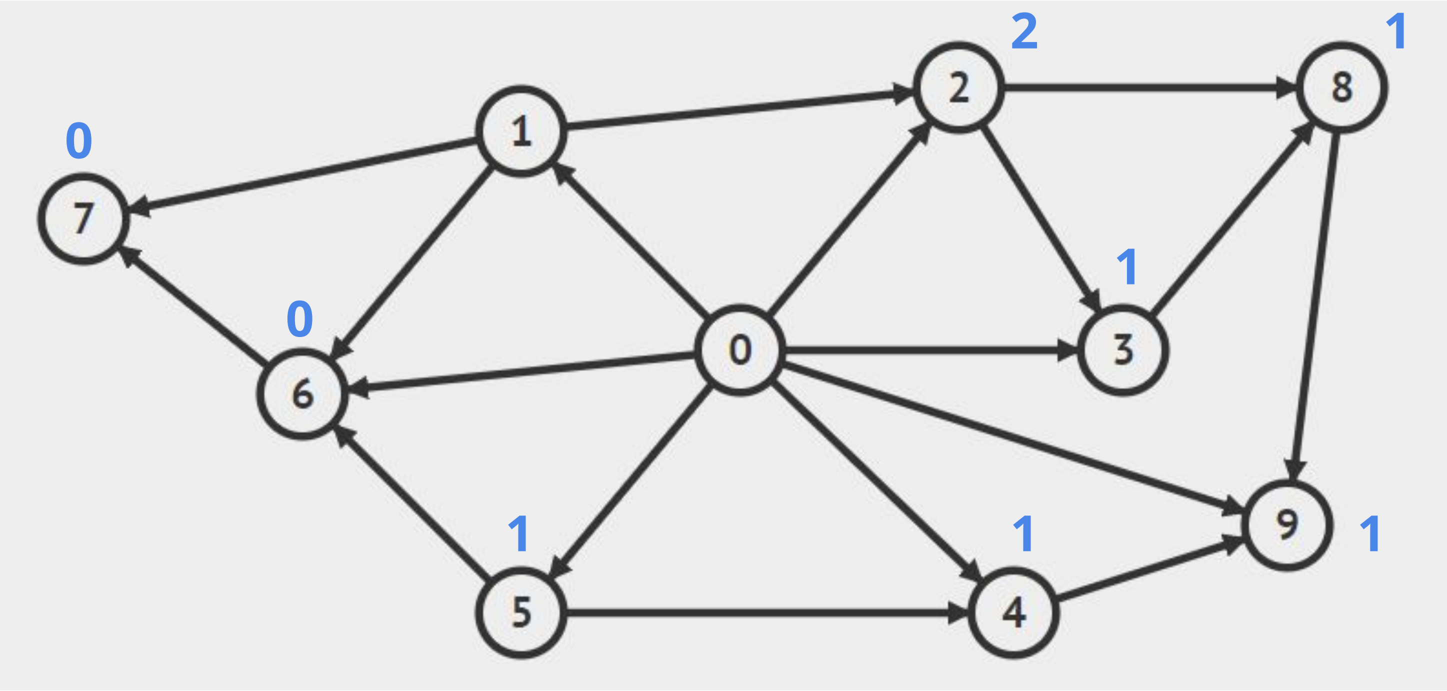
[0, 1, 5, **6**, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

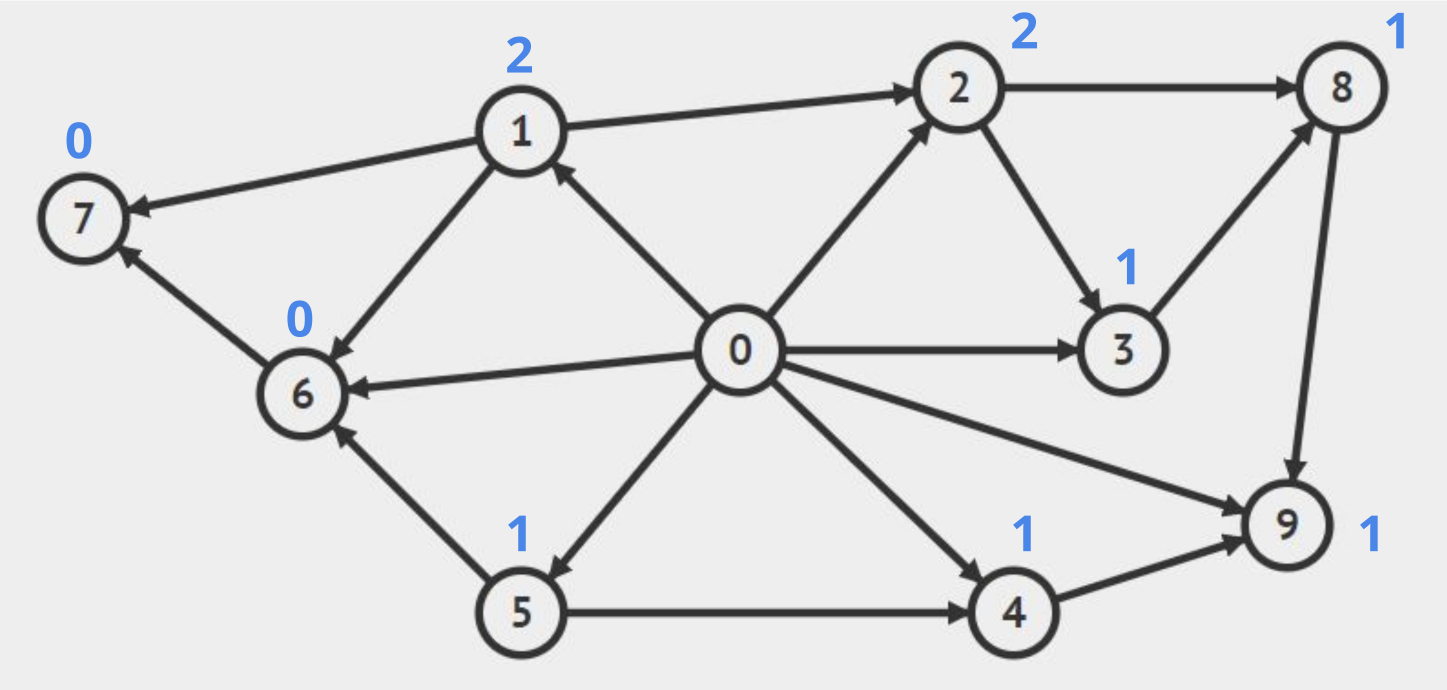
[0, 1, **5**, 6, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

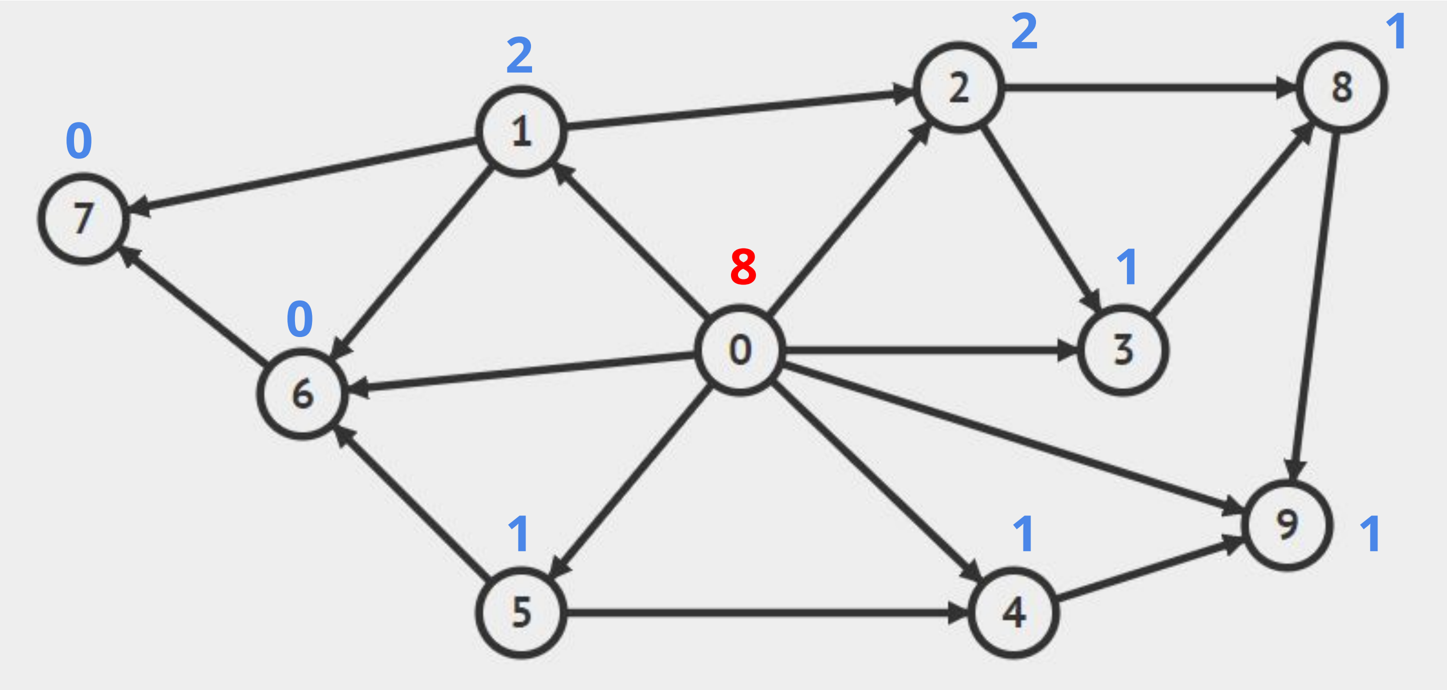
[0, **1**, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

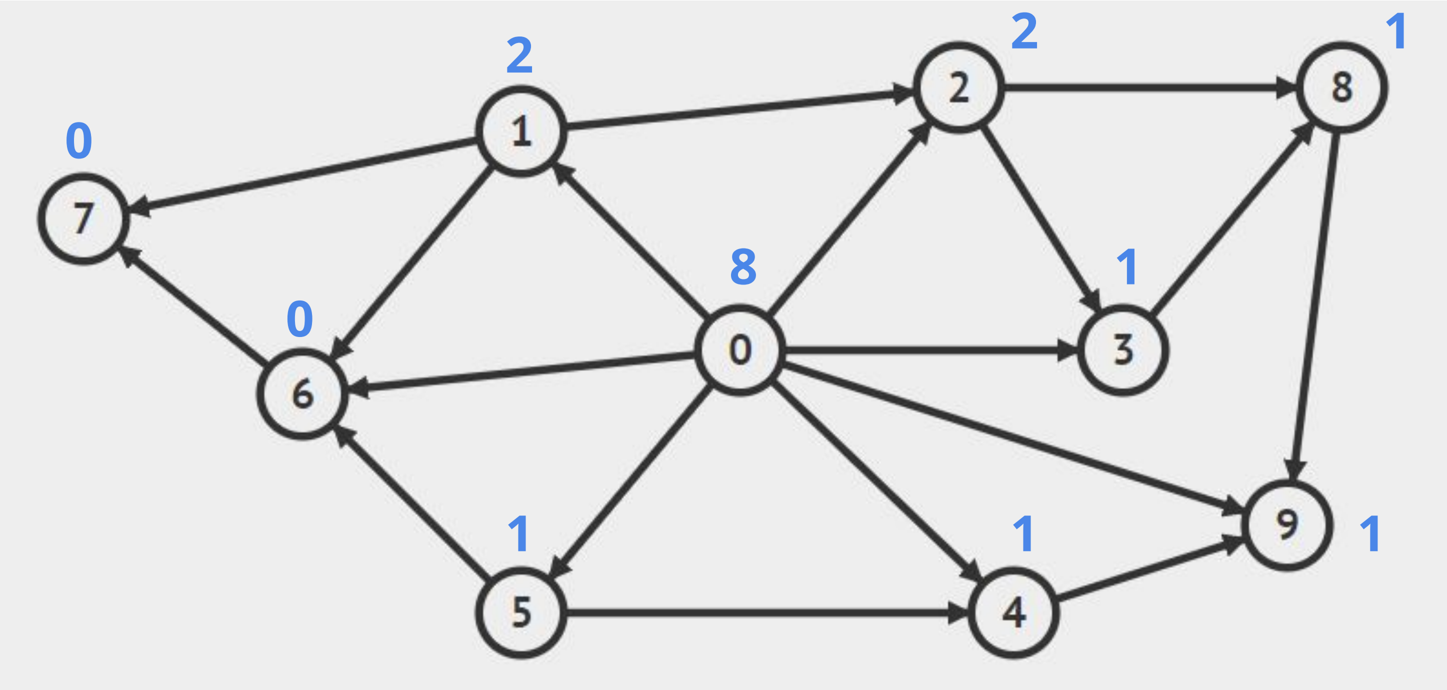
To count number of paths, we start at dest vertex

[**0**, 1, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

1. Process in reverse topo order
2. Each vertex = sum of vertices of outgoing edges

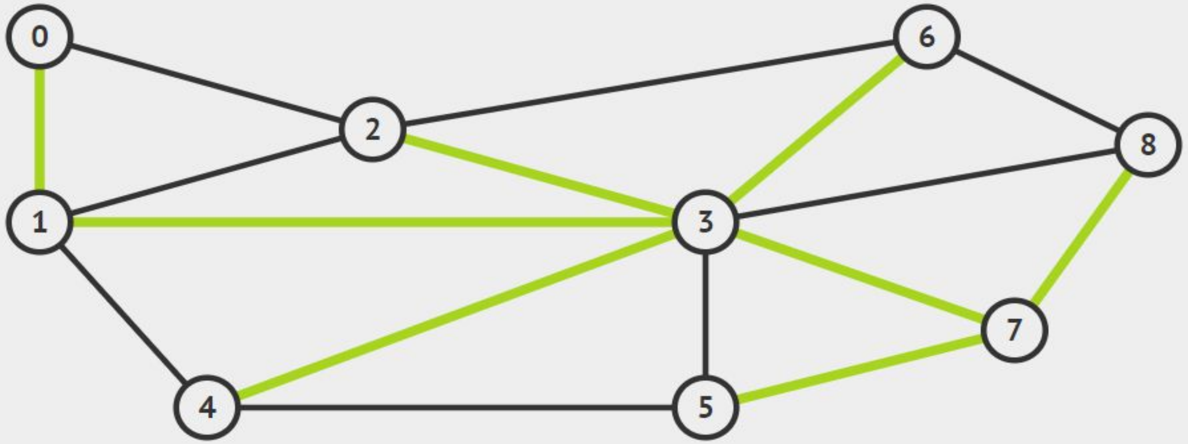


Spanning Tree

9. Click all the edges that make up the spanning tree induced by **BFS** starting from source vertex **7** for this graph.
The neighbours of a vertex are listed in ascending vertex number.
Please select the edges in the order that they are processed.

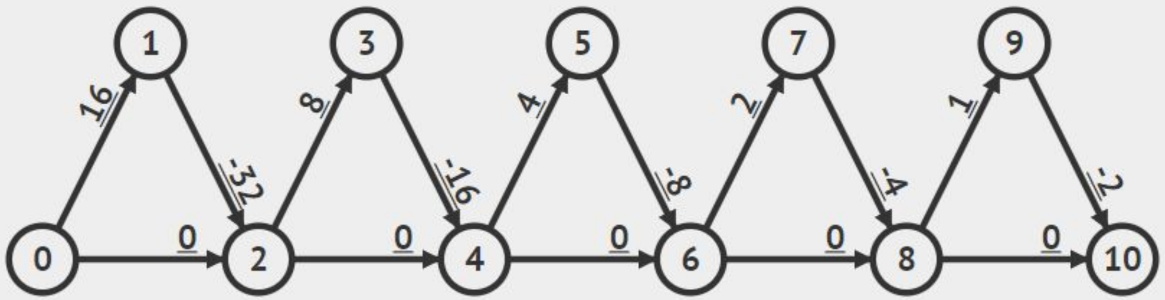
No answer

Your answer is: **(3,7),(5,7),(7,8),(1,3),(2,3),(3,4),(3,6),(0,1)**



Drawing Graph Question

2. Draw a simple connected weighted directed graph with **9** vertices and at most **10** edges such that running **Modified Dijkstra's algorithm** from source vertex 0 successfuly relaxes $\geq$**16** edges to get the correct shortest paths.
We count 1 successful relaxation if relax(u, v, w_u_v) decreases D[v].
Your graph cannot contain a negative weight cycle. **As many successful relaxes as possible.**

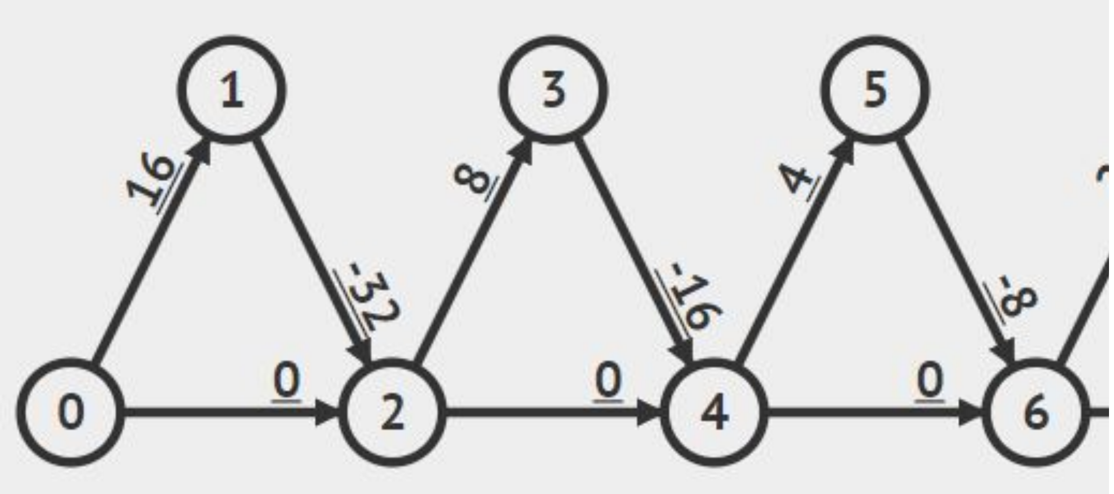


You only have 10 edges, but you need 16 ‘relaxes’. How?

What graph can make Modified Dijkstra relax the same edge multiple times?

Answer: **Dijkstra Killer!**

Drawing Graph Question

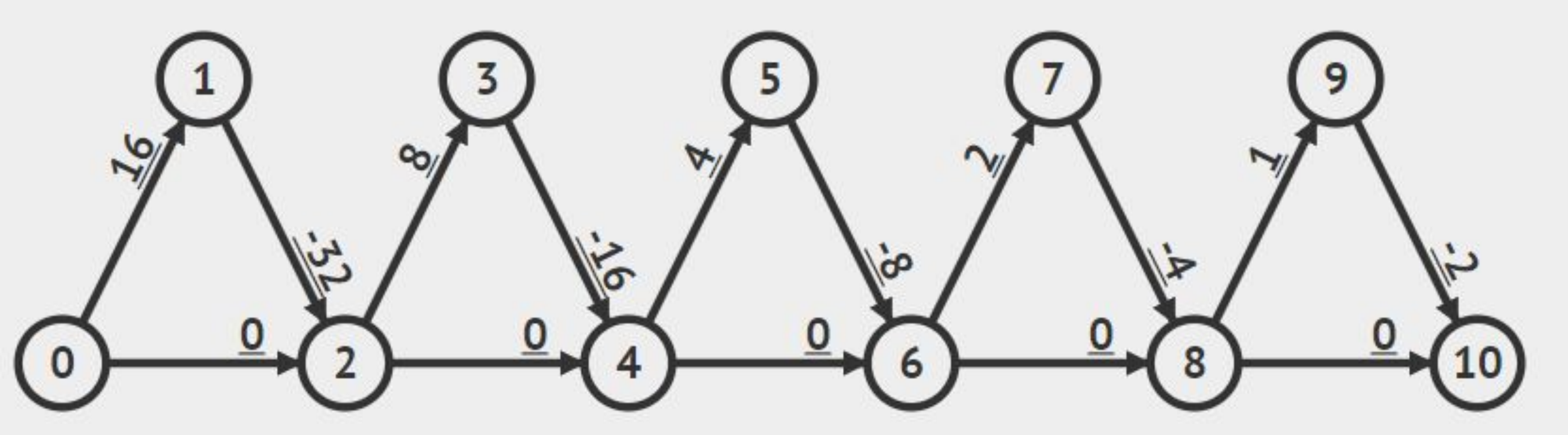


Okay.. now I have 7 vertices and 9 edges.

Can I join 2 vertices to the graph with 1 edge?

No.

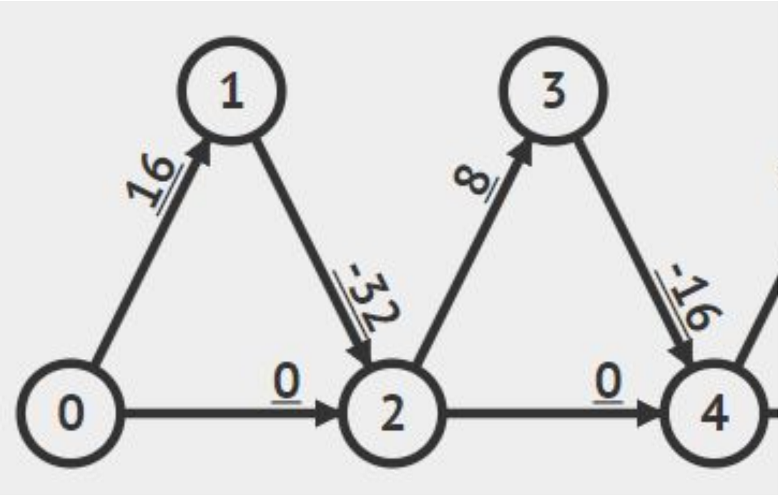
Drawing Graph Question



Oh no, I have 11 vertices and 15 edges.

That’s 2 vertex and 5 edges too many!

Drawing Graph Question



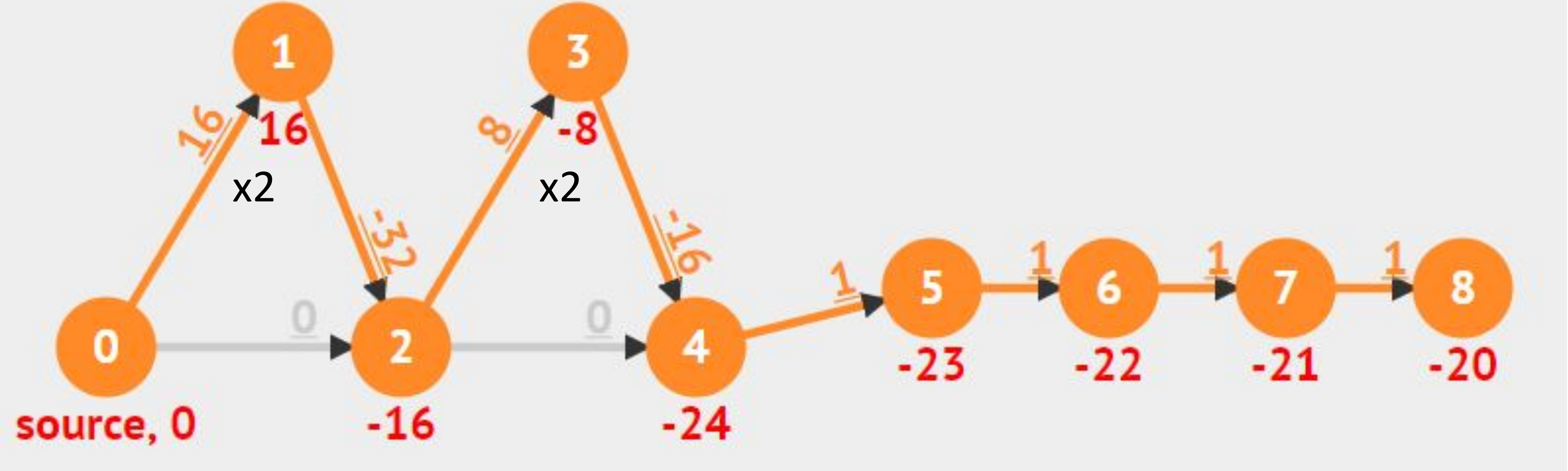
Okay.. now I have 5 vertices and 6 edges.

Can I join 4 vertices to the graph with 4 edge?

Yes!

Drawing Graph Question

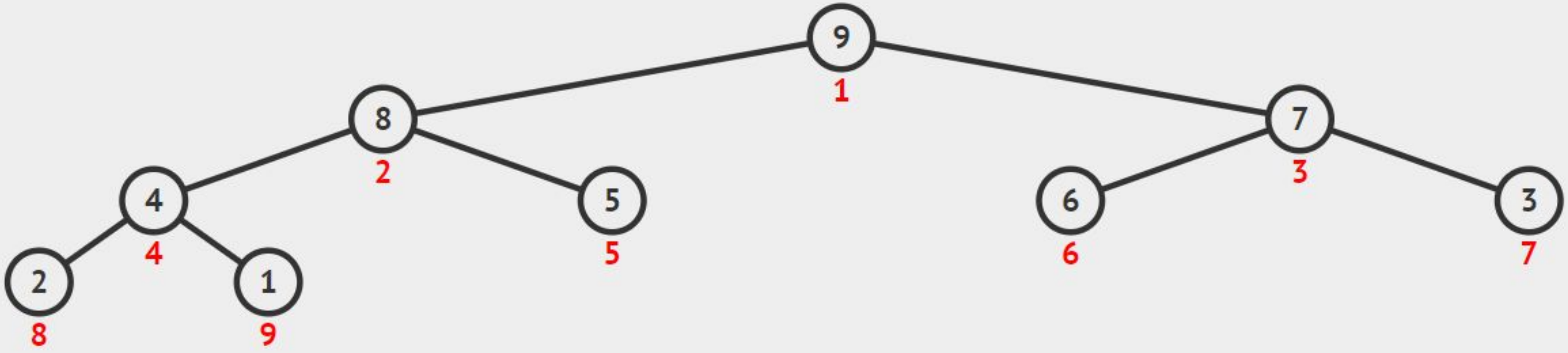
Can relax $2 \times 2 = 4$ times



$3 + 2 \times (3 + 2 \times 4) = 25 > 16$

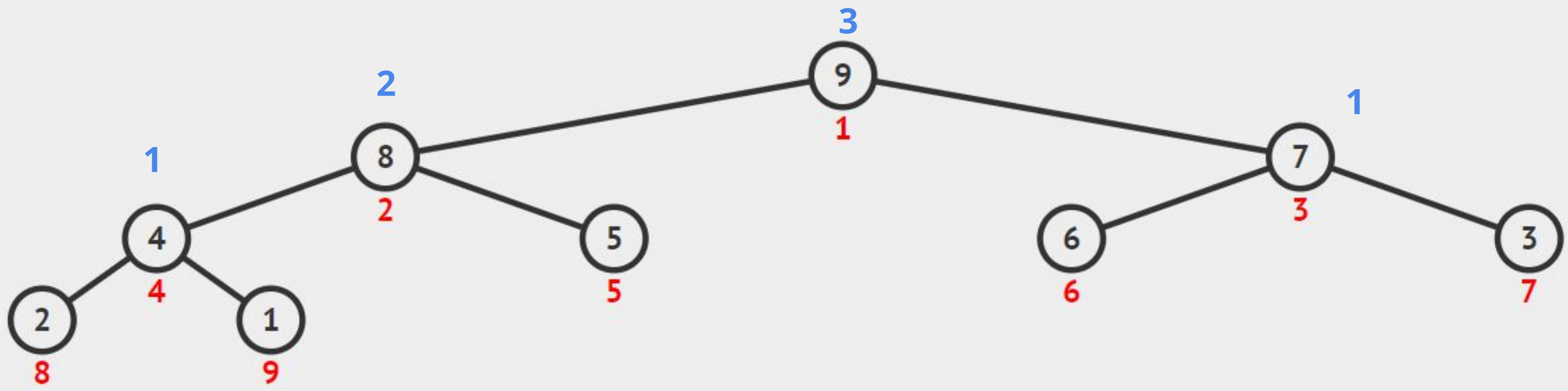
Binary Heap

12. What is the **MAXimum** number of **swaps** between heap elements required to construct a max heap of 9 elements using the **O(n)** BuildHeap(array)?



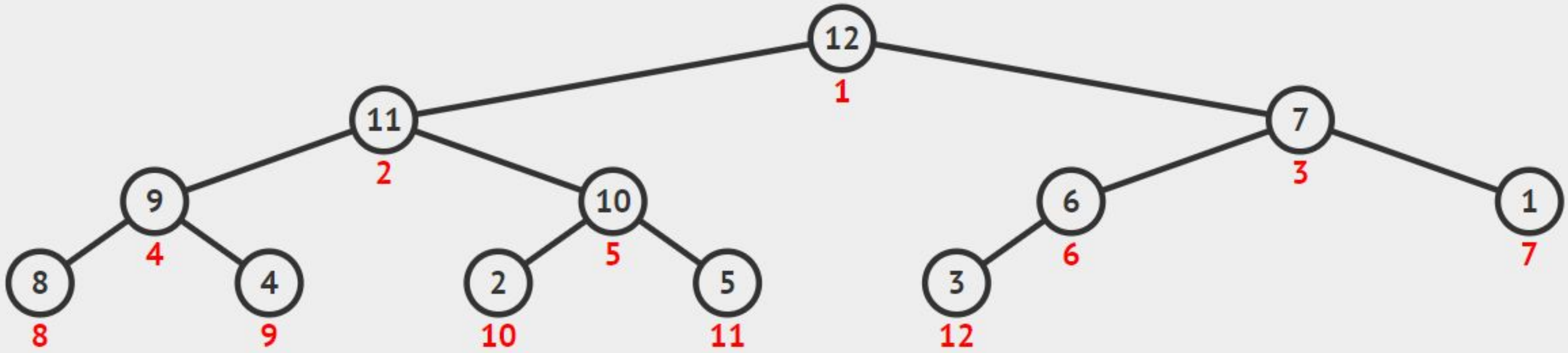
Binary Heap

12. What is the **MAXimum** number of **swaps** between heap elements required to construct a max heap of 9 elements using the **O(n)** BuildHeap(array)?



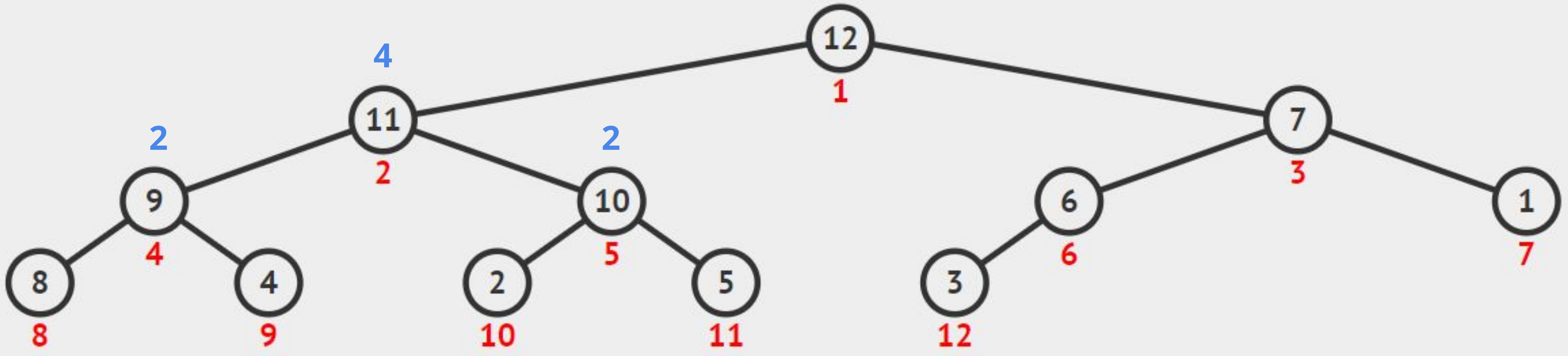
Binary Heap

6. What is the **MAXimum** number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



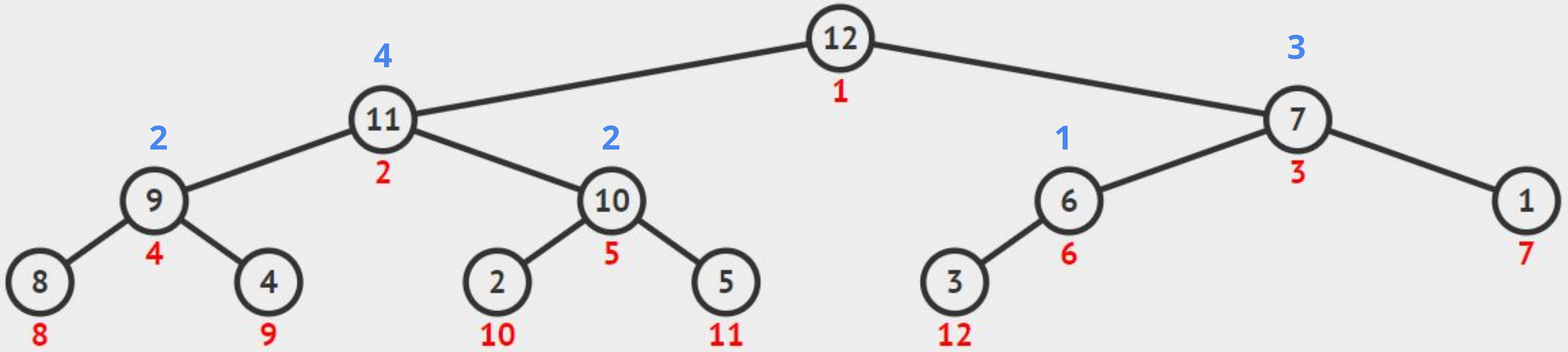
Binary Heap

6. What is the **MAXimum** number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



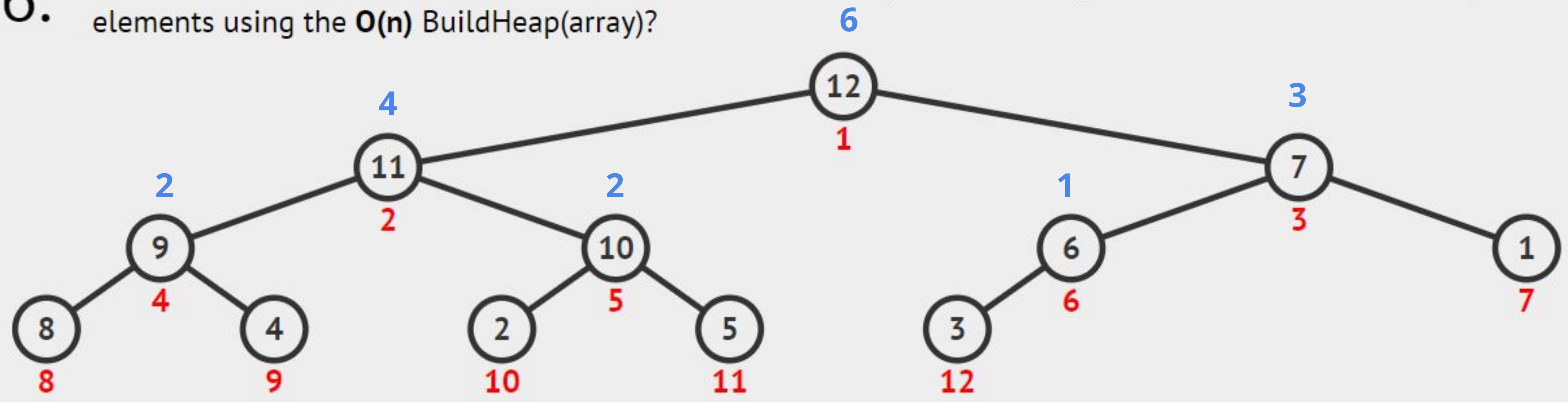
Binary Heap

6. What is the MAXimum number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



Binary Heap

6. What is the MAXimum number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



Binary Search Tree

8. What is the **minimum** number of vertices in an AVL tree of <u>**height**</u> **6**?

Recall <u>height as the maximum number of edges from root to leaf.</u>

Let $S(n)$ be min number of vertices of a height n AVL tree.

$S(0) = 1,\ S(1) = 2,\ S(2) = 4$

Binary Search Tree

8. What is the **minimum** number of vertices in an AVL tree of **height** 6?

Recall height as the maximum number of edges from root to leaf.

Let $S(n)$ be min number of vertices of a height n AVL tree.

$S(0)=1,\ S(1)=2,\ S(2)=4$

$S(n)=S(n-1)+S(n-2)+1$

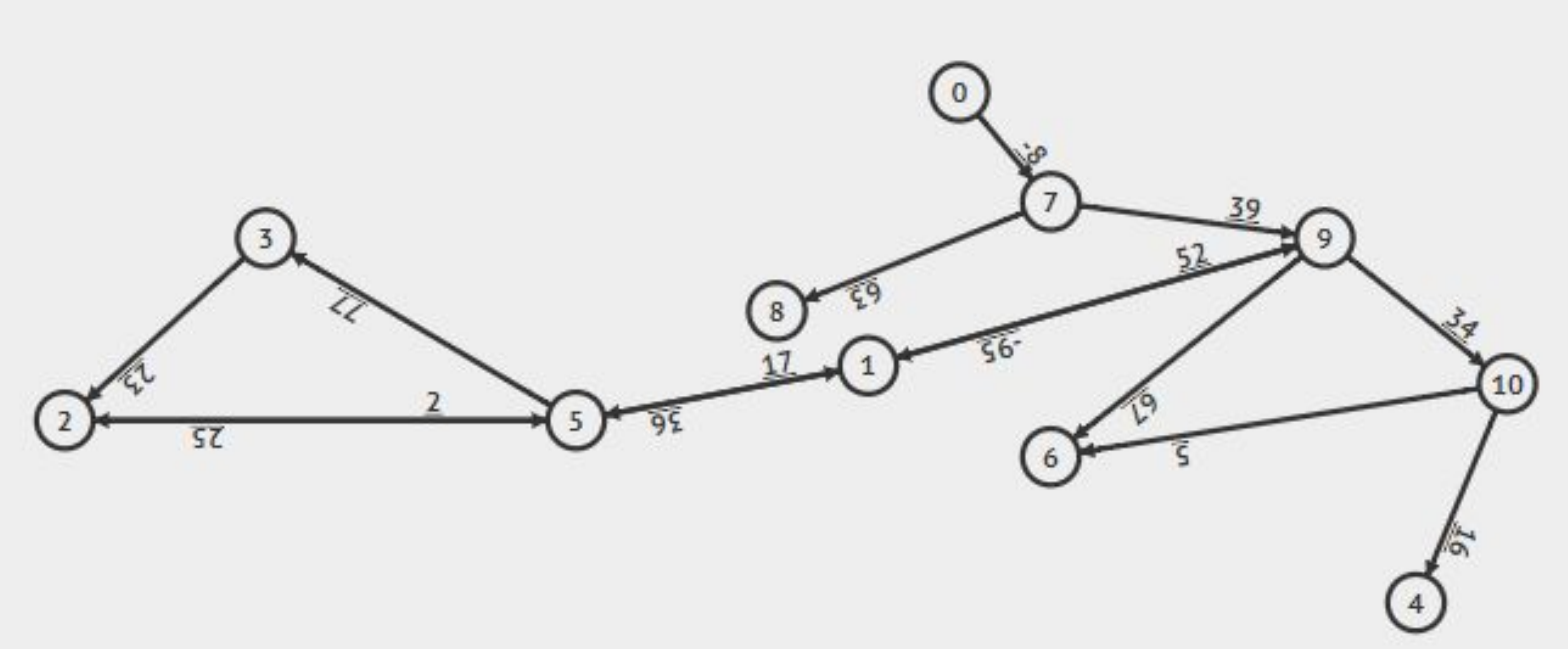
$S(3) = 7$

$S(4) = 12$

$S(5) = 20$

$S(6) = 33$

Termination



Termination

Graph property	Optimised Bellman-Ford		Original Dijkstra		Modified Dijkstra	
	Terminate	Result	Terminate	Result	Terminate	Result
Negative **cycle**	YES	WA	YES	WA	NO	NA
Negative **edge**	YES	AC	YES	WA/AC*	YES	AC†
Dijkstra Killer	YES	AC	YES	WA	YES	AC‡
BF Killer	YES	AC	YES	AC	YES	AC

Special graphs:

- Dijkstra Killer: No negative **cycle**
- BF Killer: No negative **edge**

Footnotes:

*: Depends on the graph, could be either!

†: Might take more than $O((V+E)\ log\ V)$

‡: Takes exponential time (very long)!

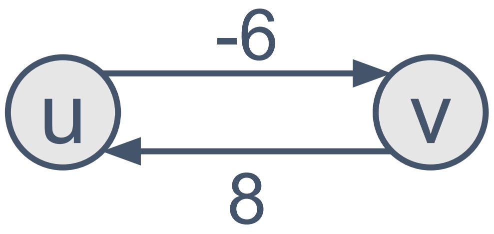
SSSP Strategies

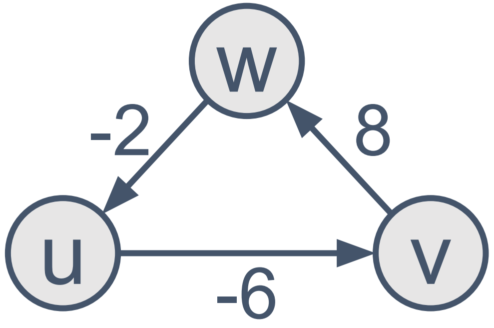
Graph property	Best strategy	Time complexity
Tree	BFS/DFS	$O(V+E)$
DAG	Relax vertices in topologically sorted order	$O(V+E)$
Unweighted	BFS	$O(V+E)$
No negative weighted edge	Original Dijkstra	$O((V+E) \log V)$
No negative weighted cycle	Modified Dijkstra	$\approx O((V+E) \log V)$
Negative weighted cycle	None! Distance is ill defined. Can be detected using Bellman-Ford	N.A

Some caveats

Be careful!

- Some edges are **bidirectional** and so if they have negative weight then they entail a negative cycle! i.e. $u \to v \to u \to \ldots$ will get more and more negative.
- Bidirectional edges can have different weights for different directions.
- Cycles of edge weight sum zero is **NOT** a negative weight cycle!





Terminologies

Bipartite Graph

A graph which vertices can be partitioned into 2 disjoint sets such that there are no edges between vertices in the same set. Such graph must have no odd length cycle

For example:

- Male and Female vertices
- No edges between male-male, female-female.

Terminologies

Spanning tree

"Click all the edges that must belong to every spanning tree of the graph shown below."

Select all the **edges** that will disconnect the graph when removed.

Final Advice (tips)?

- Some questions are *tedious*, cannot be memorized.
- For those with *small number of input cases*, it might be wise to prepare answers beforehand for example:
  - Max number of swaps for heap with n vertices
  - Preprocess for $5 < n \le 13$.

Final Advice (tips)?

Give each question a chance

- Every question has equal weightage, easy or hard.
- Skip first when you encounter hard(er) questions.

Attendance
Questions?
Break

https://visualgo.net/training?diff=Medium&n=4&tl=4&module=ufds,graphds,dfsbfs,sssp

PS5 Discussion

PS5: /fendofftitan

There is a graph, but edges may have shamans or titans. You want to find a path with minimum number of shamans, if tie then minimum number of titans, if tie than minimum distance.

Subtask 1: It’s a line graph! You have no choice but to face everything
Subtask 2: It’s a tree! There is only one path from start to finish.
Subtask 3: With no shamans or titans, this is a standard Dijkstra problem.

Other hints will be given closer to deadline in Discord – Prof.

PS5: /treehouses

This is an easy Minimum Spanning Tree problem.

You have not learnt this algorithm yet.

Live problem solving:
/onaveragetheyrepurple

On Average They're Purple

Alice and Bob are playing a game on a simple connected graph with N nodes and M edges.

Alice colors each edge in the graph red or blue.

A path is a sequence of edges where each pair of consecutive edges have a node in common. If the first edge in the pair is of a different color than the second edge, then that is a "color change."

After Alice colors the graph, Bob chooses a path that begins at node 1 and ends at node N. He can choose any path on the graph, but he wants to minimize the number of color changes in the path. Alice wants to choose an edge coloring to maximize the number of color changes Bob must make. What is the maximum number of color changes she can force Bob to make, regardless of which path he chooses?

Input

The first line contains two integer values N and M with $2 \leq N \leq 100\,000$ and $1 \leq M \leq 100\,000$. The next M lines contain two integers a_i and b_i indicating an undirected edge between nodes a_i and b_i ($1 \leq a_i, b_i \leq N$, $a_i \neq b_i$).

All edges in the graph are unique.

Output

Output the maximum number of color changes Alice can force Bob to make on his route from node 1 to node N.

Sample Input 1

```
3 3
1 3
1 2
2 3
```

Sample Output 1

```
0
```

Sample Input 2

```
7 8
1 2
1 3
2 4
3 4
4 5
4 6
5 7
6 7
```

Sample Output 2

```
3
```

Hands-On: /onaveragetheyrepurple

Hints will be provided at 5 min intervals

5 min:

Hands-On: /onaveragetheyrepurple

Hints will be provided at 5 min intervals

5 min: Try to draw a graph out and attempt to colour it

10 min:

Hands-On: /onaveragetheyrepurple

Hints will be provided at 5 min intervals

5 min: Try to draw a graph out and attempt to colour it

10 min: Alice would want to alternate the colours as much as possible…

15 min:

Hands-On: /onaveragetheyrepurple

Hints will be provided at 5 min intervals

5 min: Try to draw a graph out and attempt to colour it

10 min: Alice would want to alternate the colours as much as possible…

15 min: Bob wants to take the shortest possible path to prevent Alice from spamming reds and blues.

Thank You!

Official Feedback: https://blue.nus.edu.sg