CodeIT

Lesson 4 Objectives:

To gain an understanding of:

1. **Weighted** Undirected / Directed Graphs

2. What the **Minimum Spanning Tree & Shortest Path Problems** are and the intuition behind solving them

3. How we might use greedy algorithms like **Prim's MST & Dijkstra's SP** to solve these problems

Weighted Directed / Undirected Graphs

Weighted Directed / Undirected Graphs

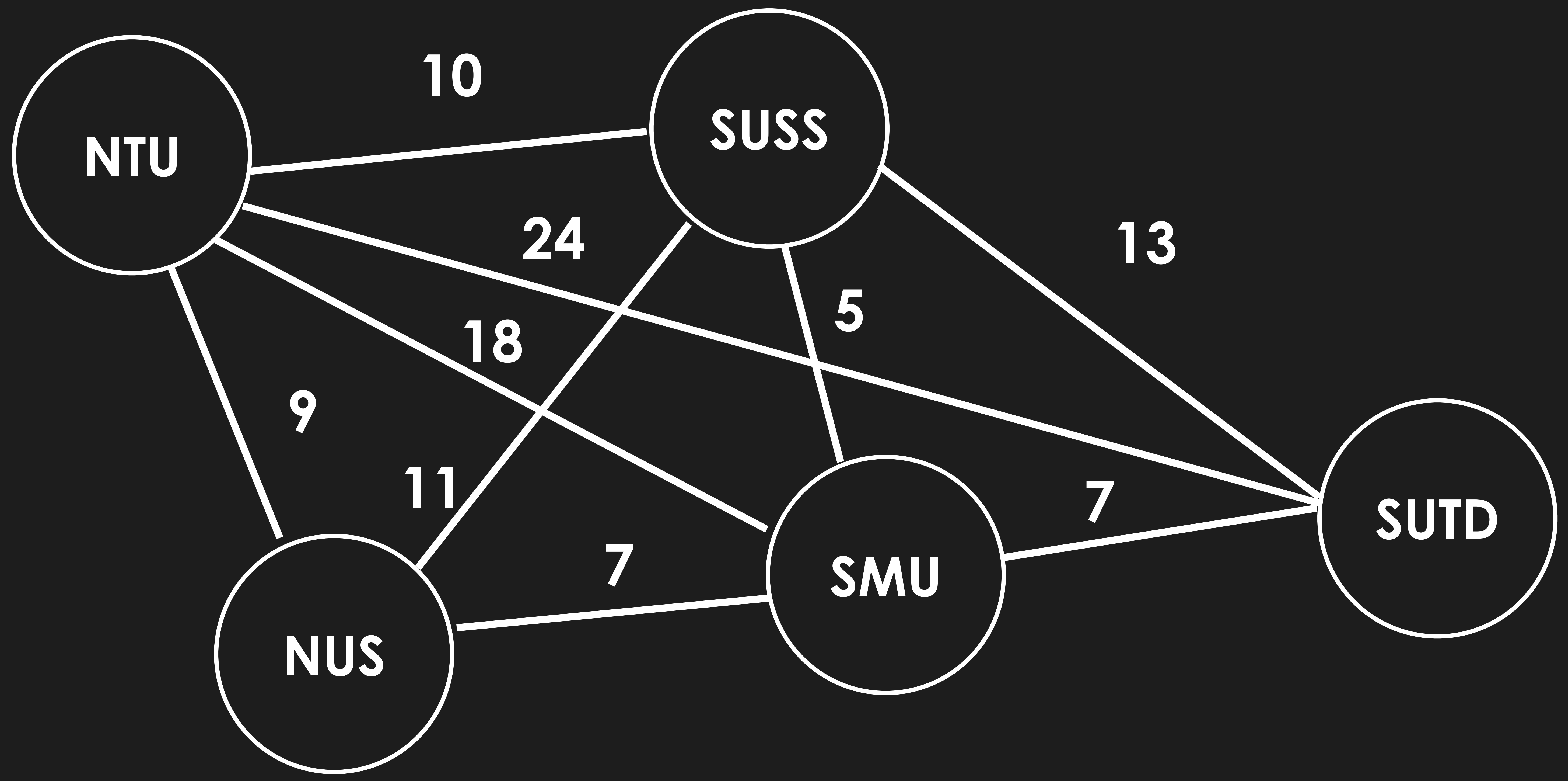
Weighted graphs are ones where edges are assigned a **weight**

Weighted Directed / Undirected Graphs

Weighted graphs are ones where edges are assigned a **weight**

Weights are useful to represent particular traffic situations in networks, for instance, **traffic flow on roads, edge capacities, etc.**

Weighted Graph Example: Roads



vertices: Addresses (Schools)

edges: Roads

weights: Distance of each road

CodeIT

Representing weighted undirected graphs

WeightedEdge

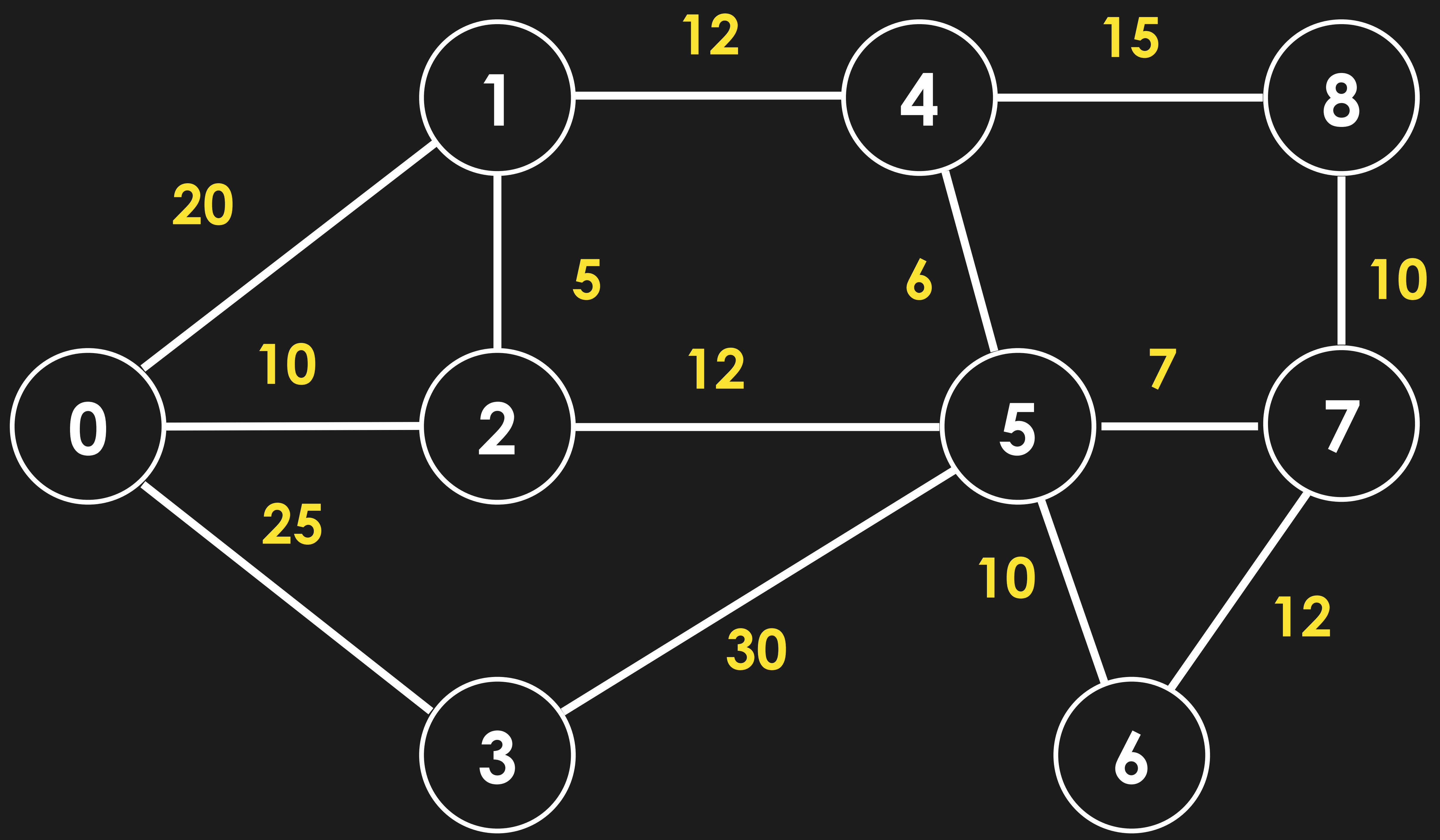
```
class WeightedEdge:
    def __init__(self, src, dest, weight) -> None:
        self.src = src
        self.dest = dest
        self.weight = weight
```

We now have an additional parameter for weight in our edge class

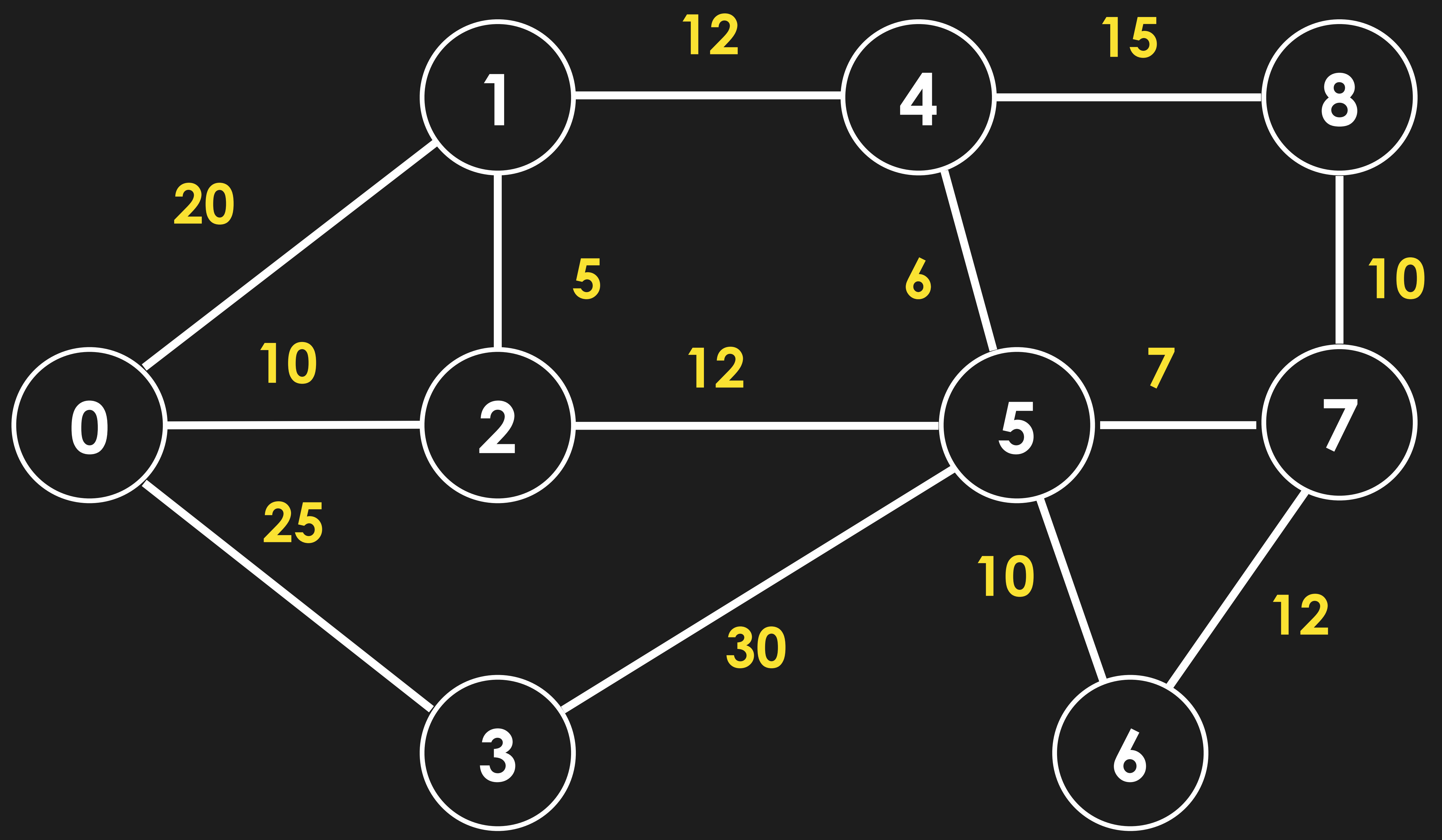
WeightedDigraph

```
class WeightedGraph:
    def __init__(self, V):
        self.adjList = [[] for i in range(V)]

    def addEdge(self, src, dest, weight):
        newEdge1 = WeightedEdge(src, dest, weight)
        self.adjList[src].append(newEdge1)

        newEdge2 = WeightedEdge(dest, src, weight)
        self.adjList[dest].append(newEdge2)
```

Let's try to represent the following graph!



Let's try to represent the following graph!



0 → (0, 1, 20) → (0, 2, 10) → (0, 3, 25)

1 → (1, 0, 20) → (1, 2, 5) → (1, 4, 6)

2 → (2, 0, 10) → (2, 1, 5) → (2, 5, 12)

3 → (3, 0, 25) → (3, 5, 30)

4 → (4, 1, 12) → (4, 5, 6) → (4, 8, 15)

5 → (5, 2, 12) → (5, 4, 6) → (5, 7, 7) ...

6 → (6, 5, 10) → (6, 7, 12)

7 → (7, 8, 10) → (7, 5, 7) → (7, 6, 12)

8 → (8, 7, 10) → (8, 4, 15)

CodeIT

Weighted Graph Problems

Weighted Graph Algorithms

Weighted graph algorithms give us the opportunity to understand and analyse networks meaningfully

Weighted Graph Algorithms

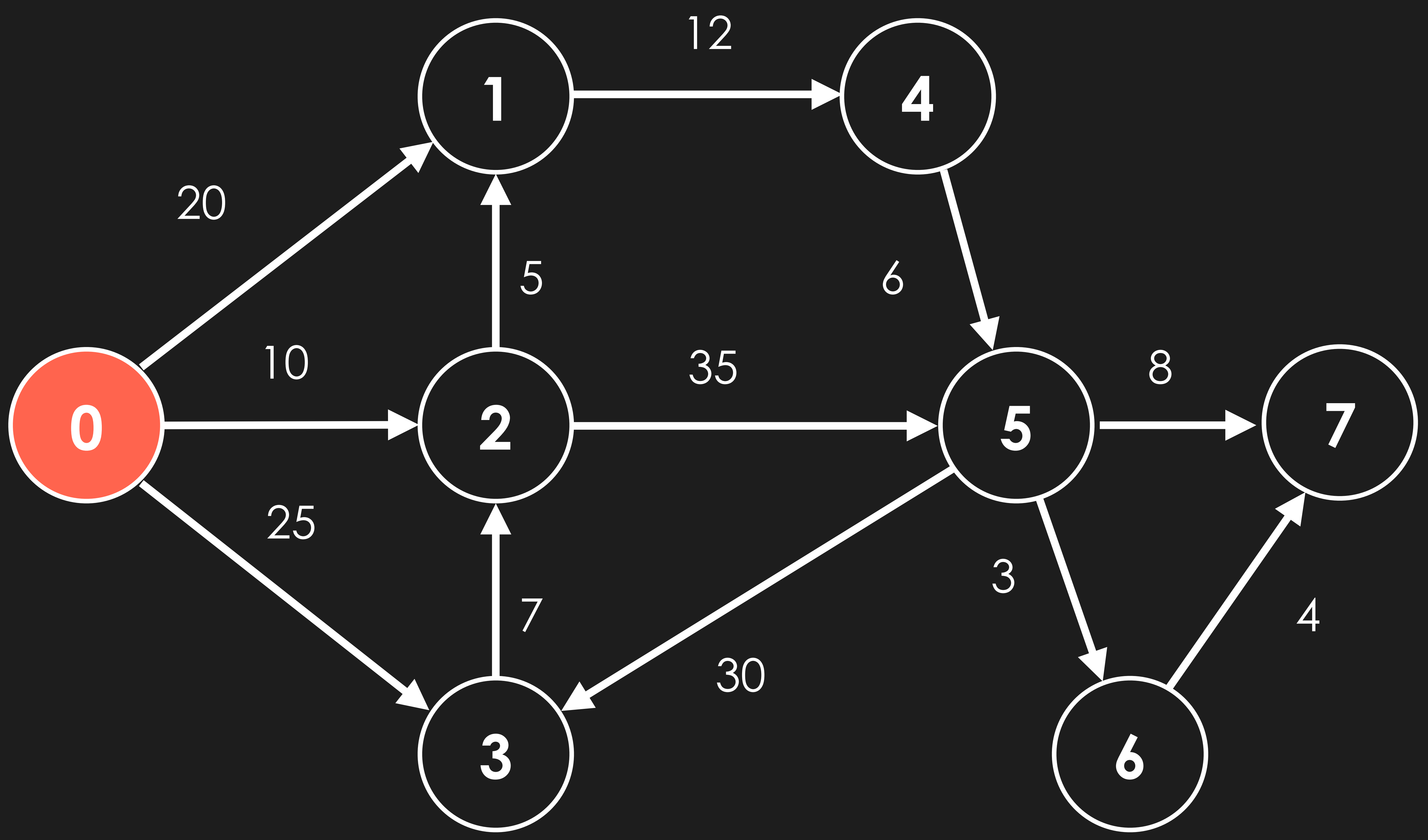
Weighted graph algorithms give us the opportunity to understand and analyse networks meaningfully

We will learn about 2 famous **graph** problems: **minimum spanning trees** and **shortest paths** in a weighted graph, and view how we can use greedy strategies to solve these problems efficiently

Shortest Path Problem in Weighted Graphs

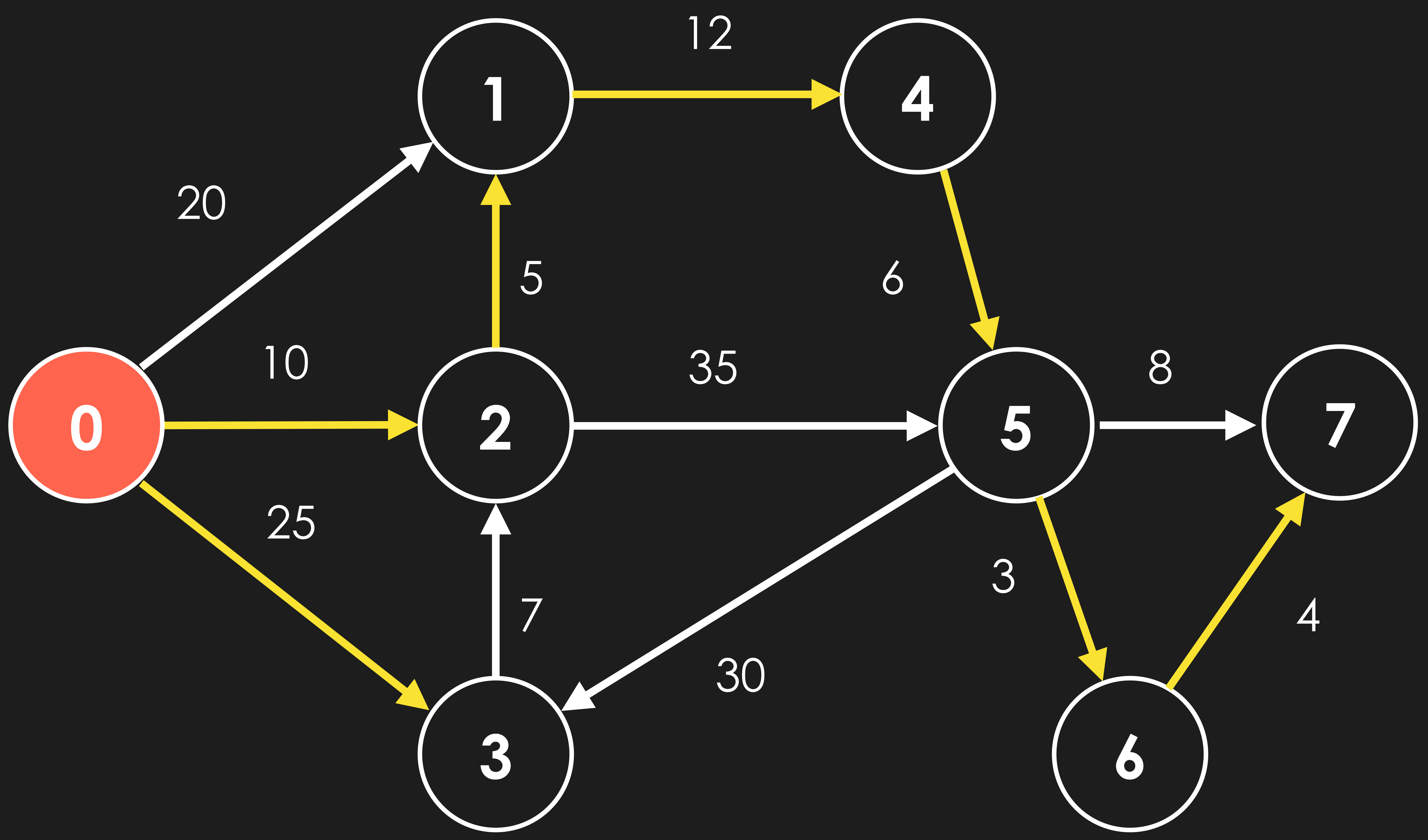
Shortest Path Problem in Weighted Graphs

In a weighted undirected / directed graph, determine the shortest path from a source vertex to every other vertex (aka **Shortest Path Tree**)



Shortest Path Problem in Weighted Graphs

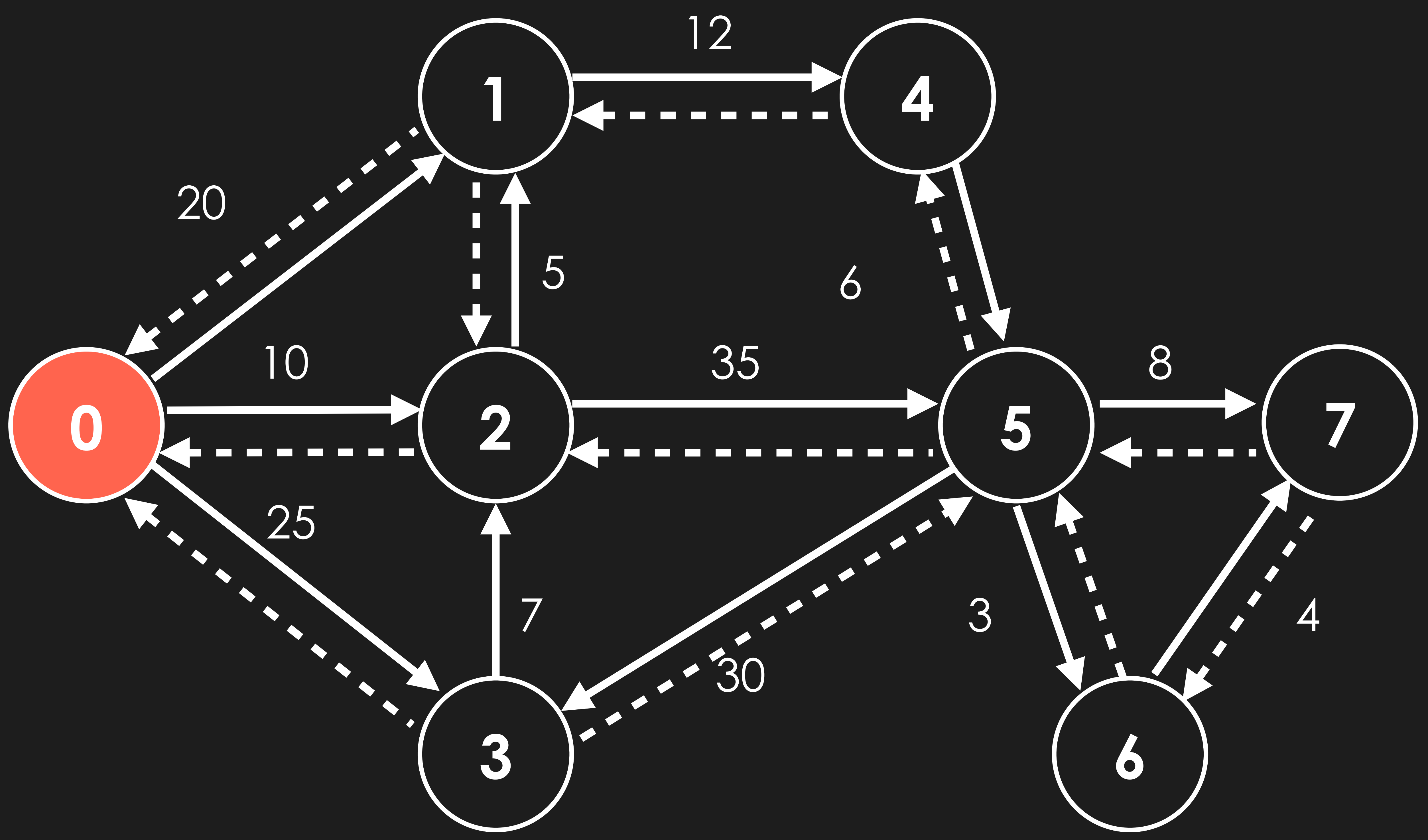
In a weighted undirected / directed graph, determine the shortest path from a source vertex to every other vertex (aka **Shortest Path Tree**)



This is a shortest path tree!

Shortest Path Problem in Weighted Graphs

In a weighted undirected / directed graph, determine the shortest path from a source vertex to every other vertex (aka **Shortest Path Tree**)



Weighted Undirected Graphs have shortest paths too, just that each edge goes both ways!

Quiz: Where might shortest path algorithms in Weighted Graphs be useful?

Quiz: Where might shortest path algorithms in Weighted Graphs be useful?

Google Maps!

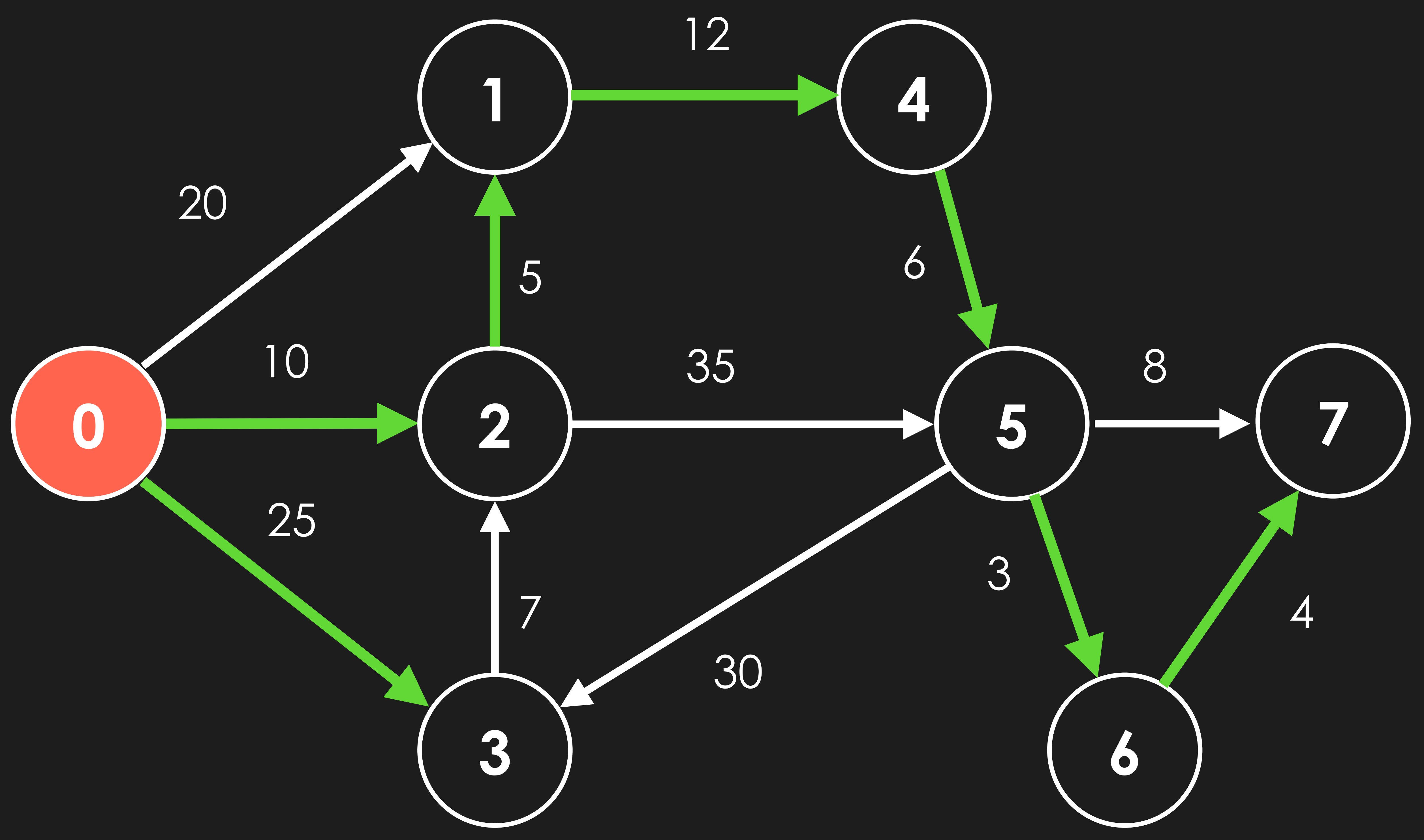
CodeIT

General Intuition of SPP

We can represent our shortest path tree in terms of two arrays:

1. **distTo**: The min distance from source to that vertex
2. **edgeTo**: The last edge in the shortest path to that vertex

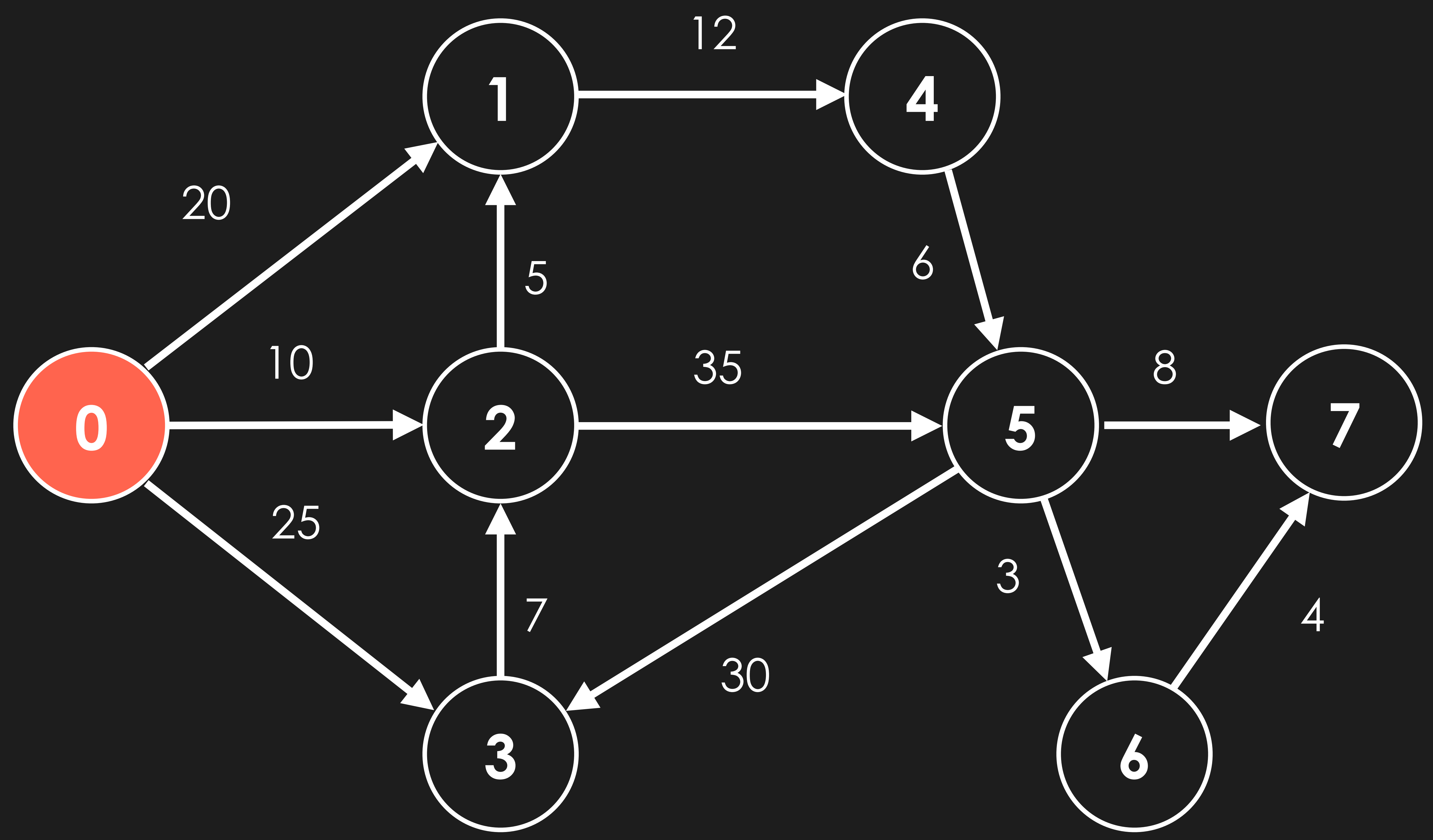
We can then construct the shortest path from src to node easily by backtracking



	distTo		edgeTo
0	0	**0**	-1
1	15	**1**	2 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	27	**4**	1 - 4
5	33	**5**	4 - 5
6	36	**6**	5 - 6
7	40	**7**	5 - 7

Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

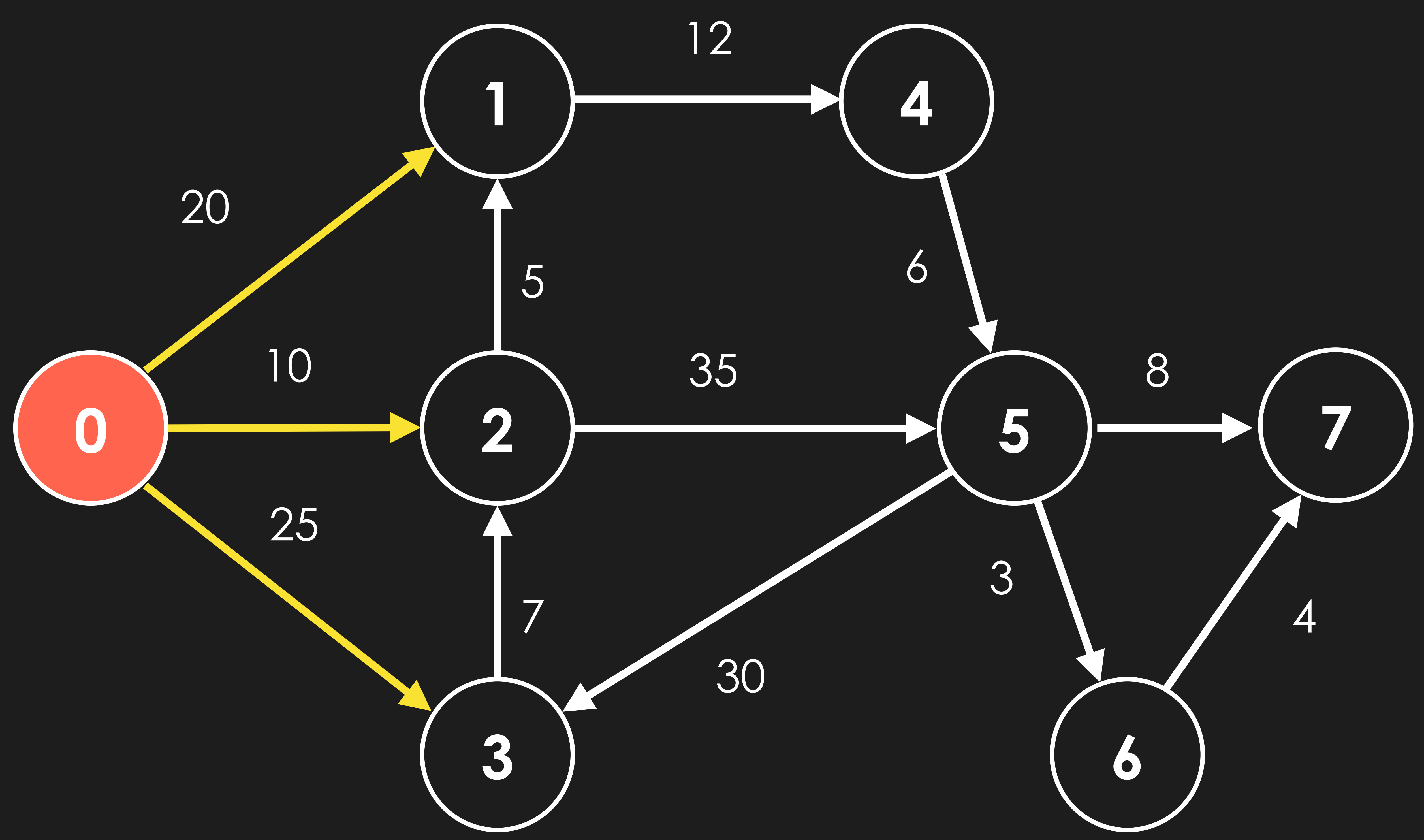


	distTo		edgeTo
0	0	0	-1
1	INF	1	-1
2	INF	2	-1
3	INF	3	-1
4	INF	4	-1
5	INF	5	-1
6	INF	6	-1
7	INF	7	-1

Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

Assume that we have explored the following nodes:

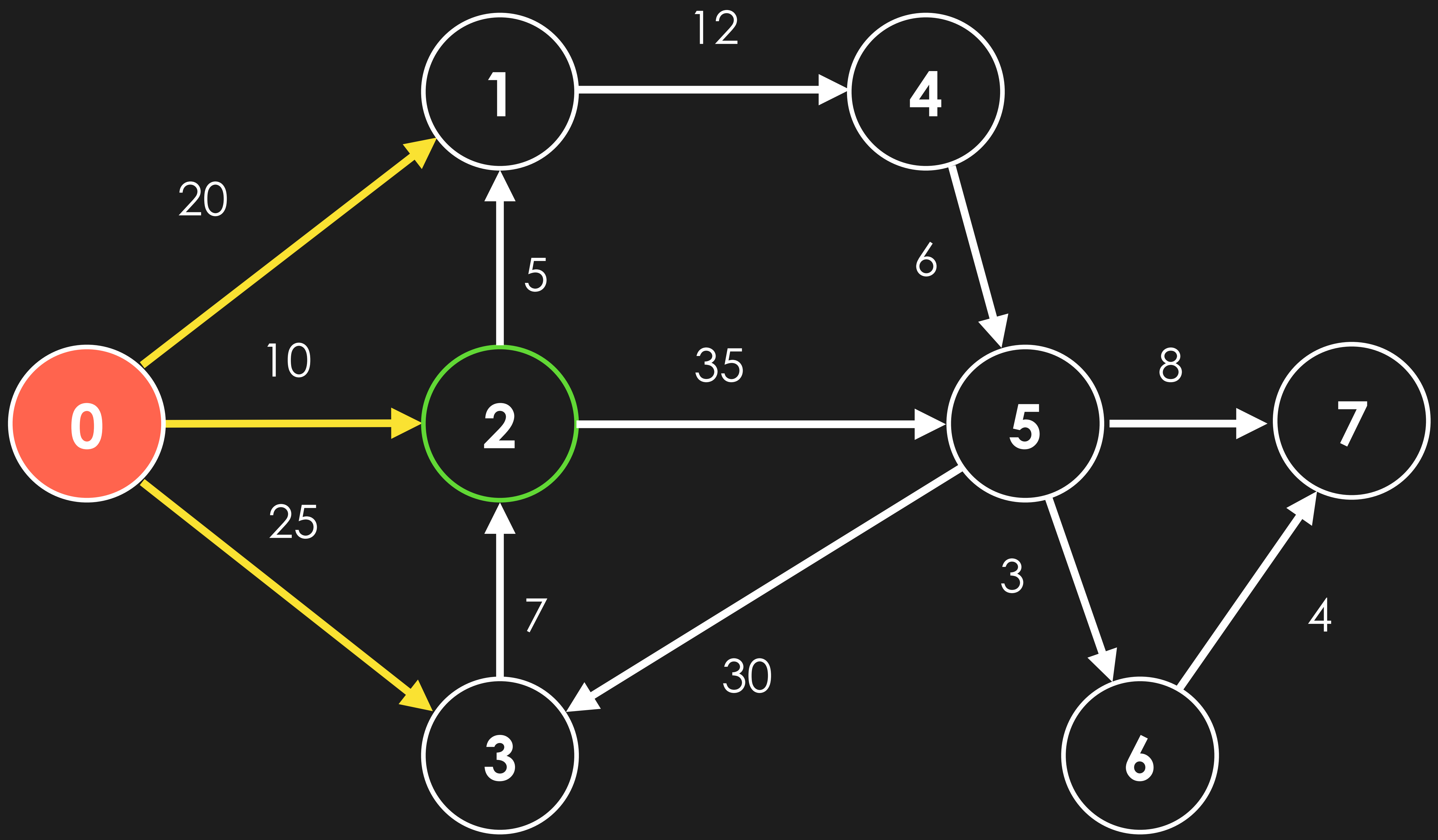


	distTo
0	0
1	20
2	10
3	25
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

CodeIT

Let's say we are now traversing adjacent edges of vertex 2



Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

distTo

0	0
1	20
2	10
3	25
4	INF
5	INF
6	INF
7	INF

edgeTo

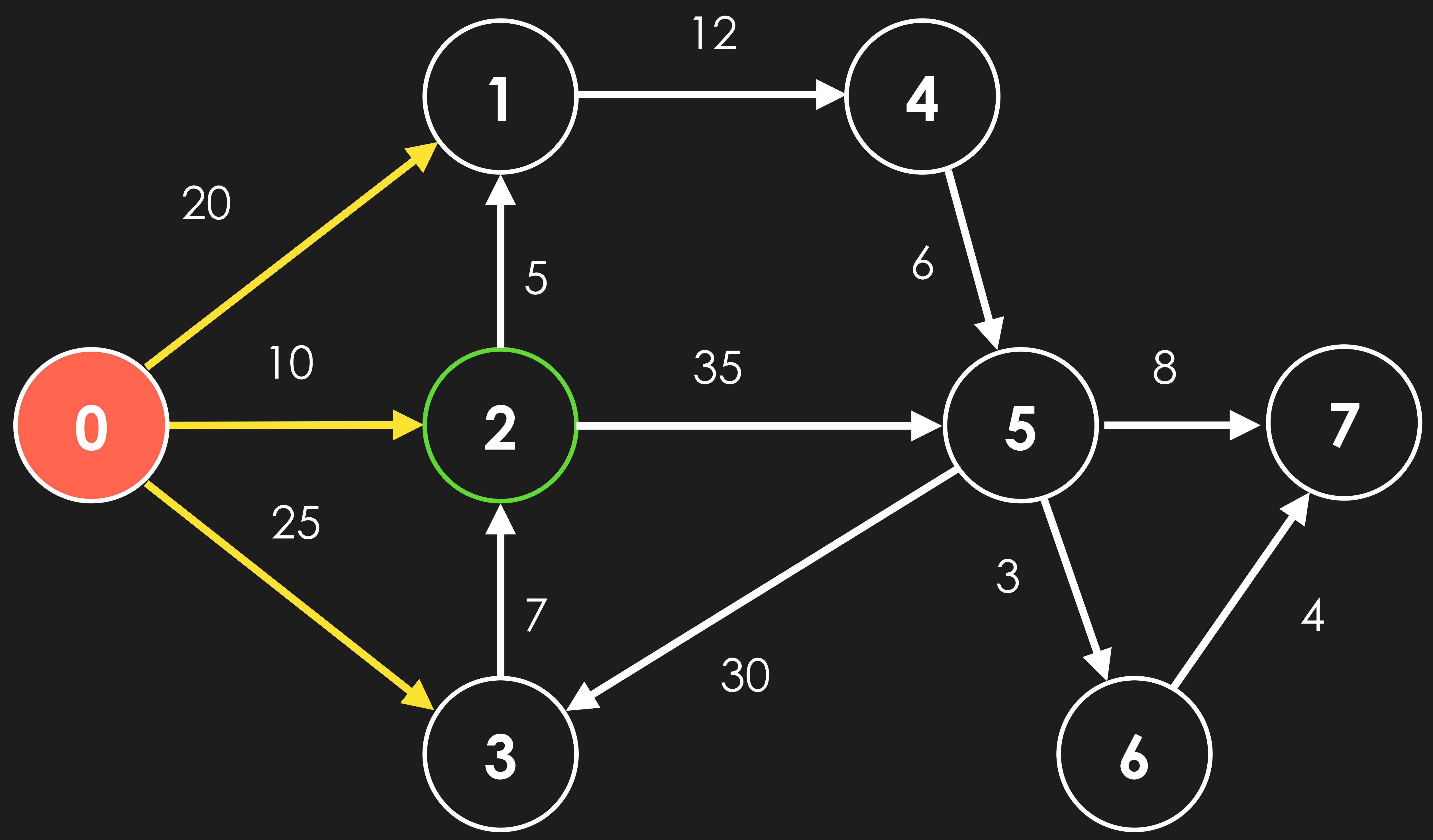
0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

For edge v - w:

- To relax an edge, we check if **distTo[v] + weight < distTo[w]**
- If yes, we then set **distTo[w] = distTo[v] + weight** AND **edgeTo[w] = v**



	distTo
0	0
1	20
2	10
3	25
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

CodeIT

For edge v - w:

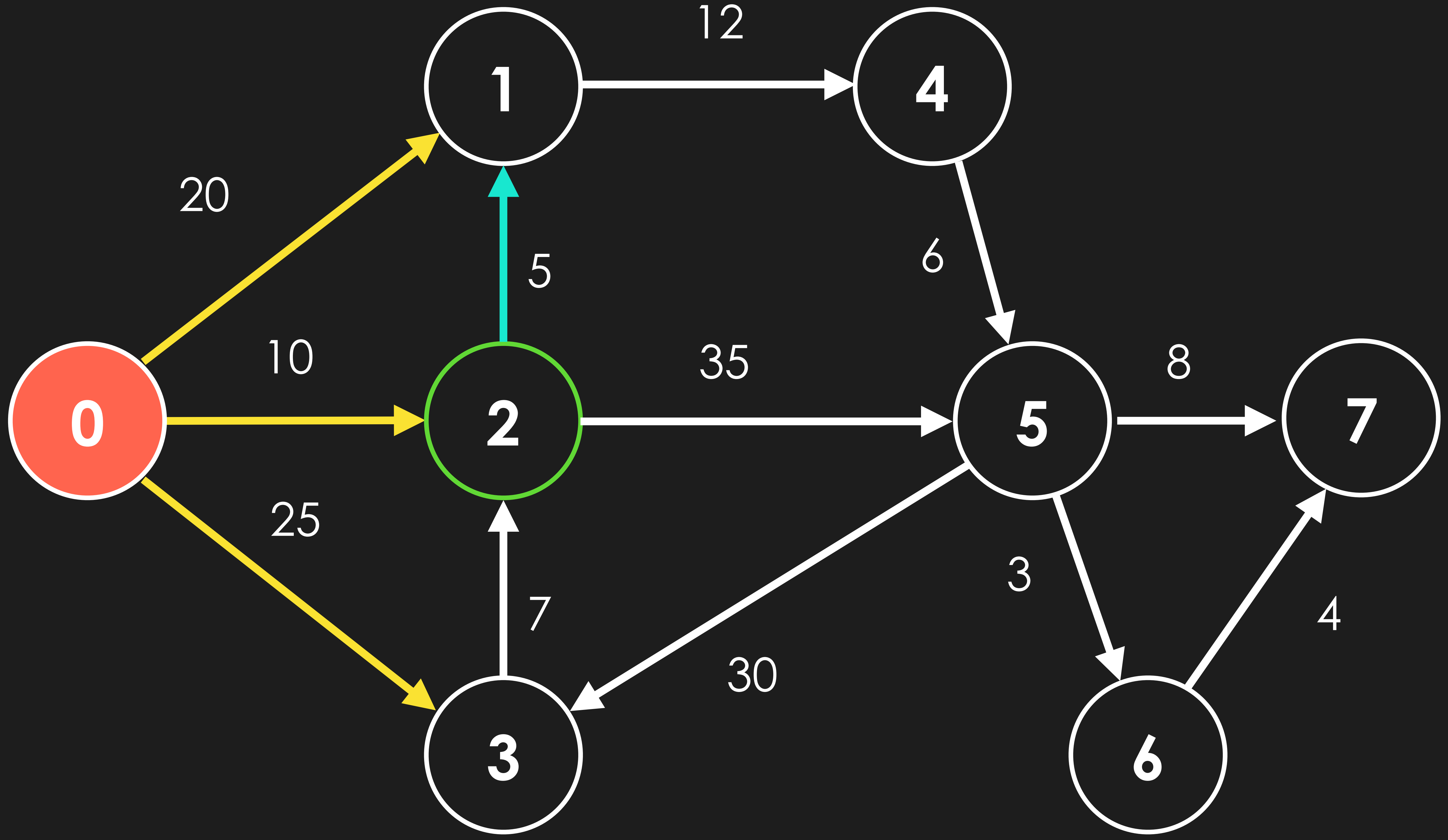
- To relax an edge, we check if **distTo[v] + weight < distTo[w]**
- If yes, we then set **distTo[w] = distTo[v] + weight** AND **edgeTo[w] = v**

Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

Checking 2 - 1:

distTo[2] + weight = 15, distTo[1] = 20 | **15 < 20**



	distTo		edgeTo
0	0	**0**	-1
1	20	**1**	0 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	INF	**4**	-1
5	INF	**5**	-1
6	INF	**6**	-1
7	INF	**7**	-1

Relaxing an edge

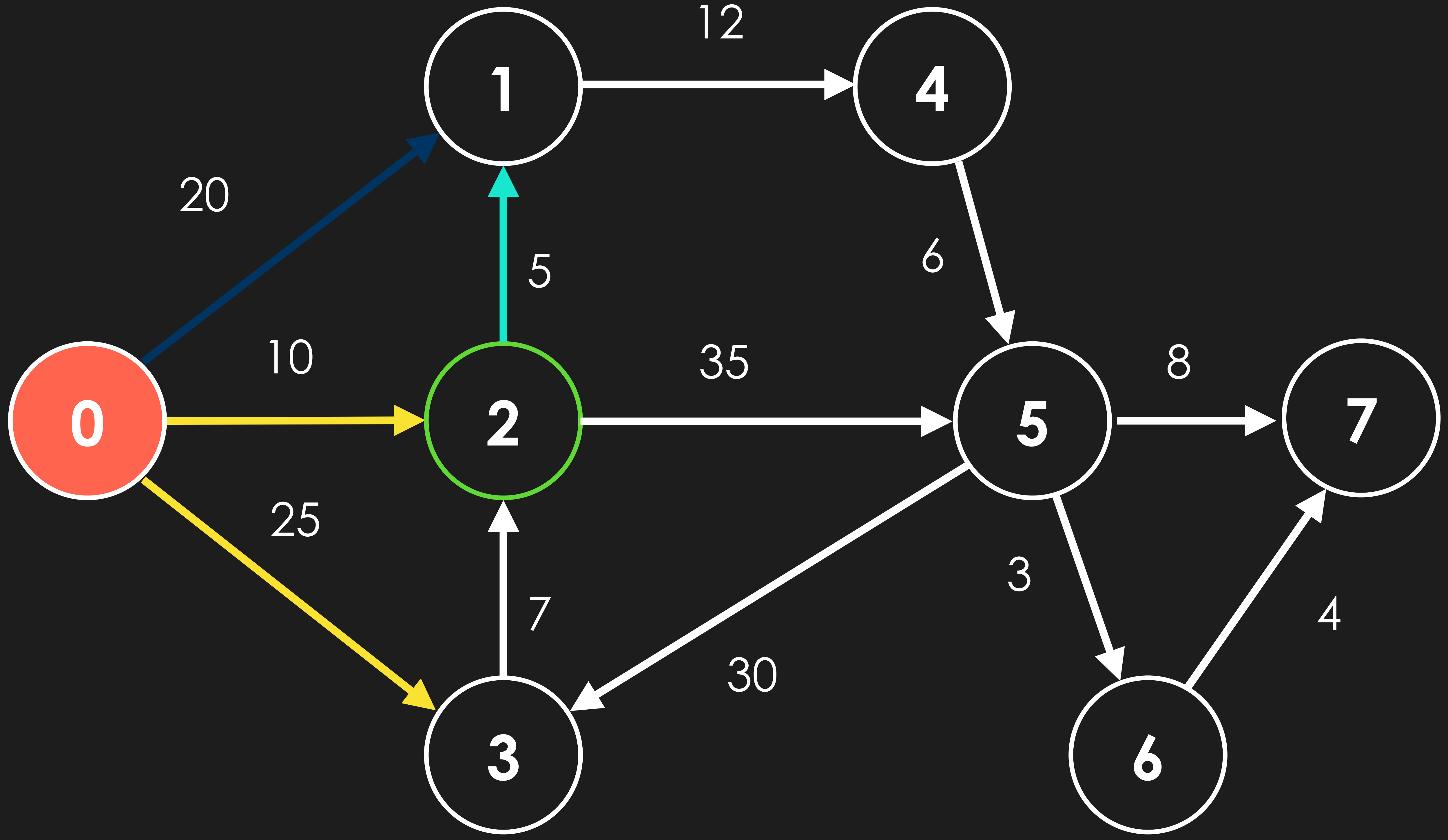
Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

For edge v - w:

- To relax an edge, we check if **distTo[v] + weight < distTo[w]**
- If yes, we then set **distTo[w] = distTo[v] + weight** AND **edgeTo[w] = v**

Checking 2 - 1:

distTo[2] + weight = 15, distTo[1] = 20 | **15 < 20**



	distTo	edgeTo
0	0	-1
1	**15**	**2 - 1**
2	10	0 - 2
3	25	0 - 3
4	INF	-1
5	INF	-1
6	INF	-1
7	INF	-1

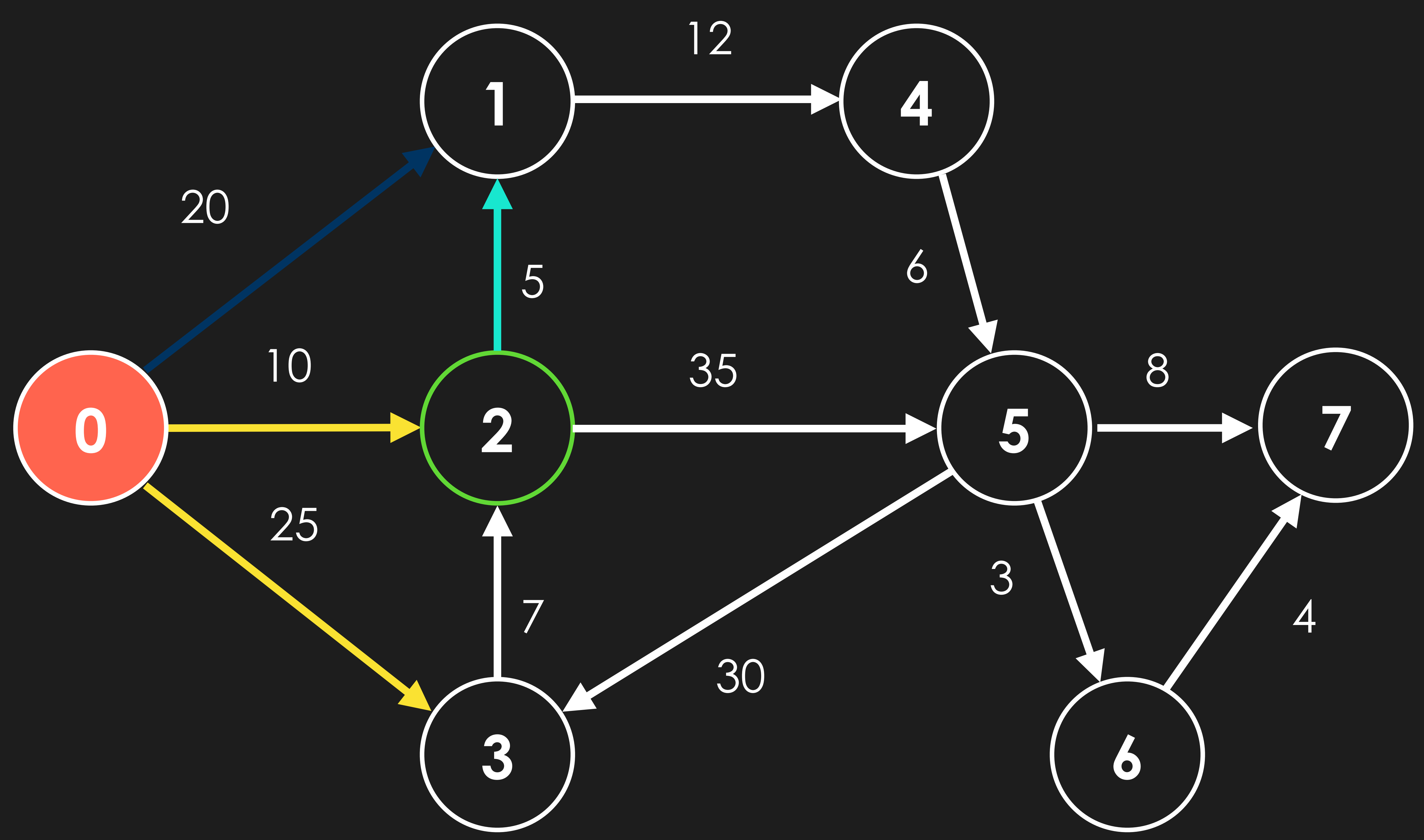
For edge v - w:

- To relax an edge, we check if **distTo[v] + weight < distTo[w]**
- If yes, we then set **distTo[w] = distTo[v] + weight** AND **edgeTo[w] = v**

Relaxing an edge

Relaxing an edge means to try to get a lower cost of reaching a vertex through another path

Edge 2 - 1 has been relaxed!



	distTo
0	0
1	**15**
2	10
3	25
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	**2 - 1**
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

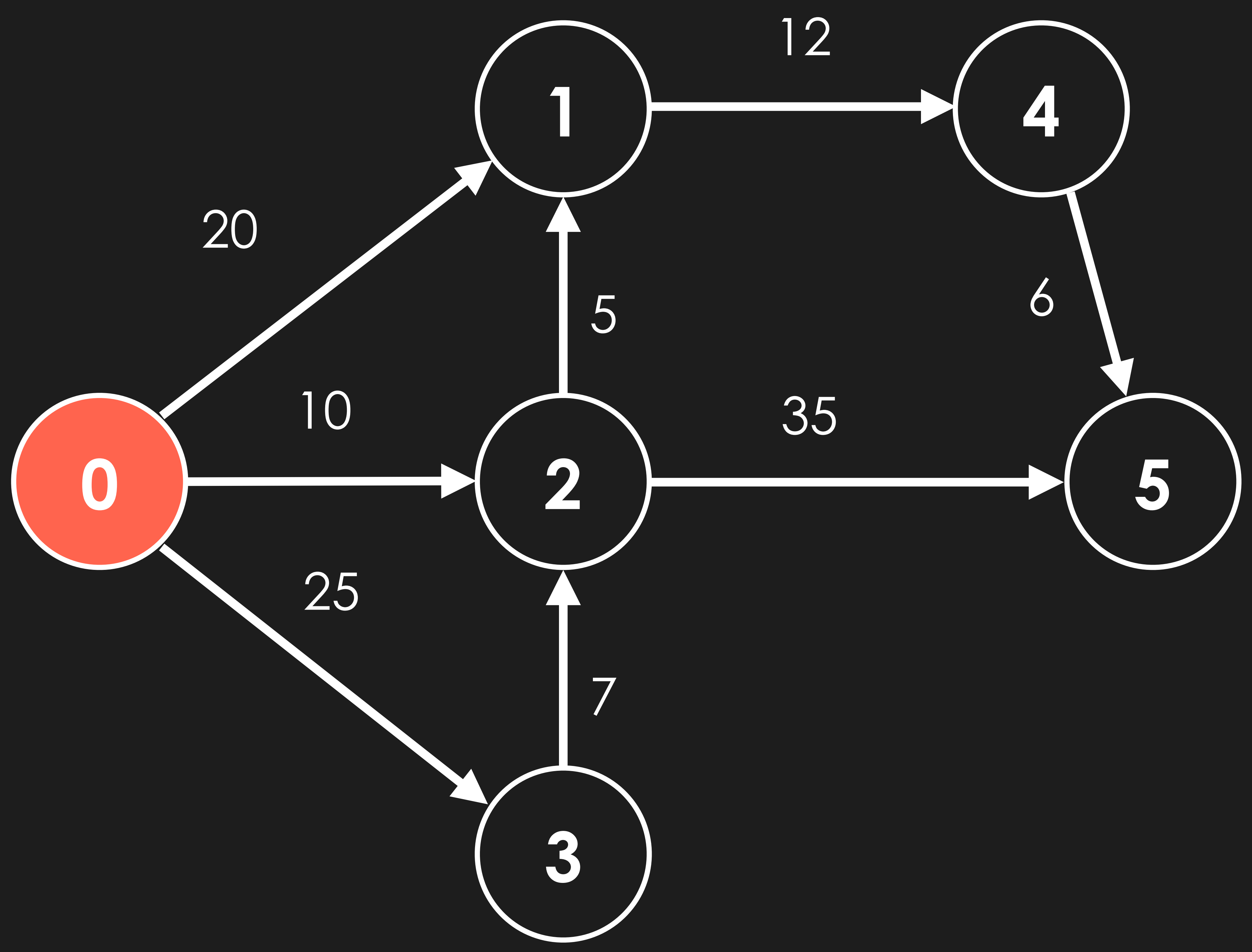
Topological Sort

Topological Sort

If the graph is **acyclic**:

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex, from left to right in the sorted order, **relax all adjacent edges**

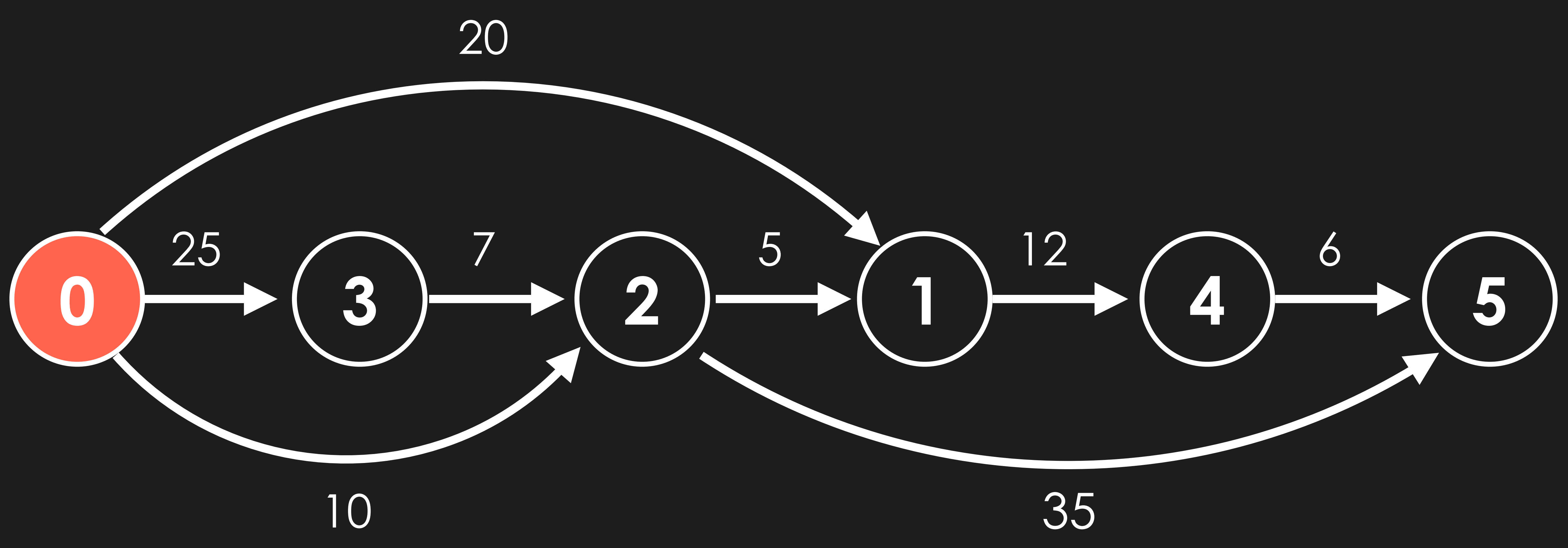
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo
0	INF
1	INF
2	INF
3	INF
4	INF
5	INF

	edgeTo
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



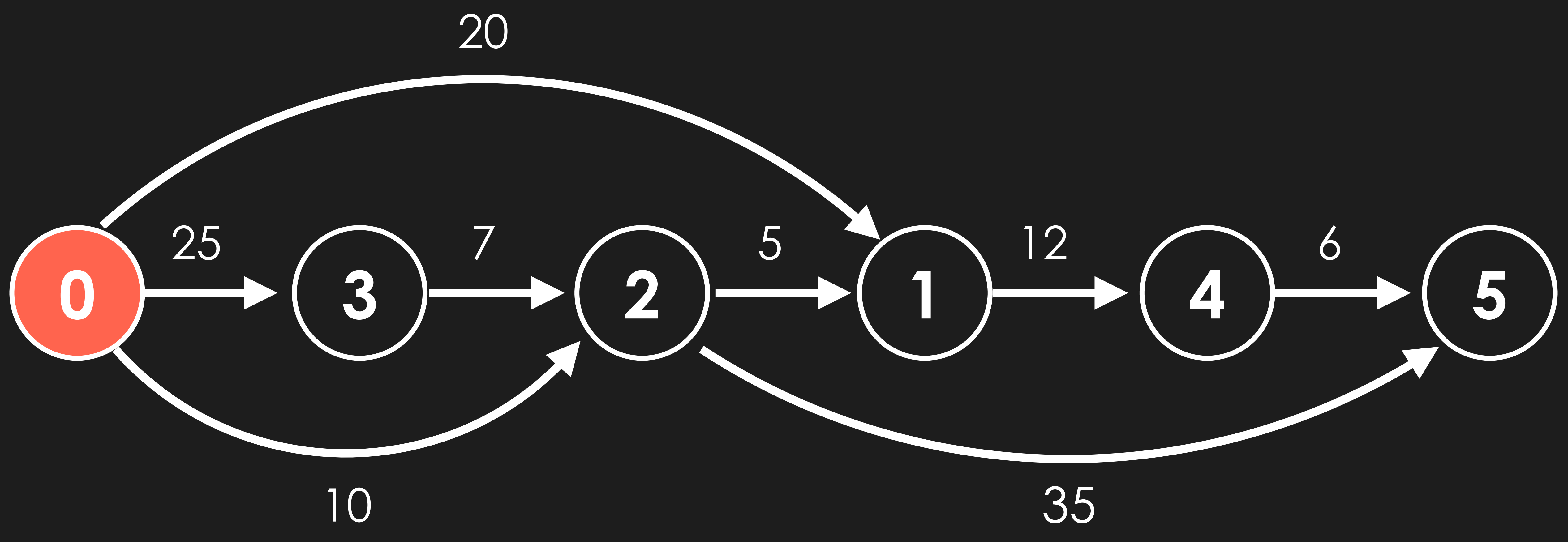
distTo

0	INF
1	INF
2	INF
3	INF
4	INF
5	INF

edgeTo

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**

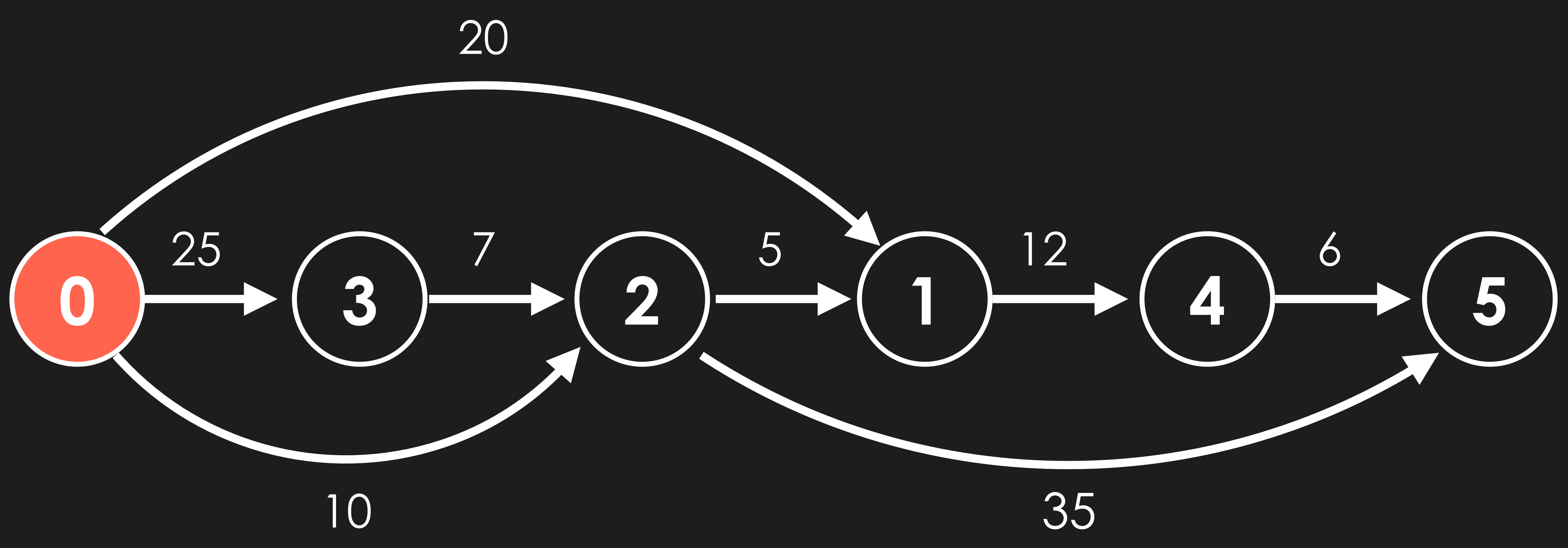


	distTo
0	0
1	INF
2	INF
3	INF
4	INF
5	INF

	edgeTo
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1

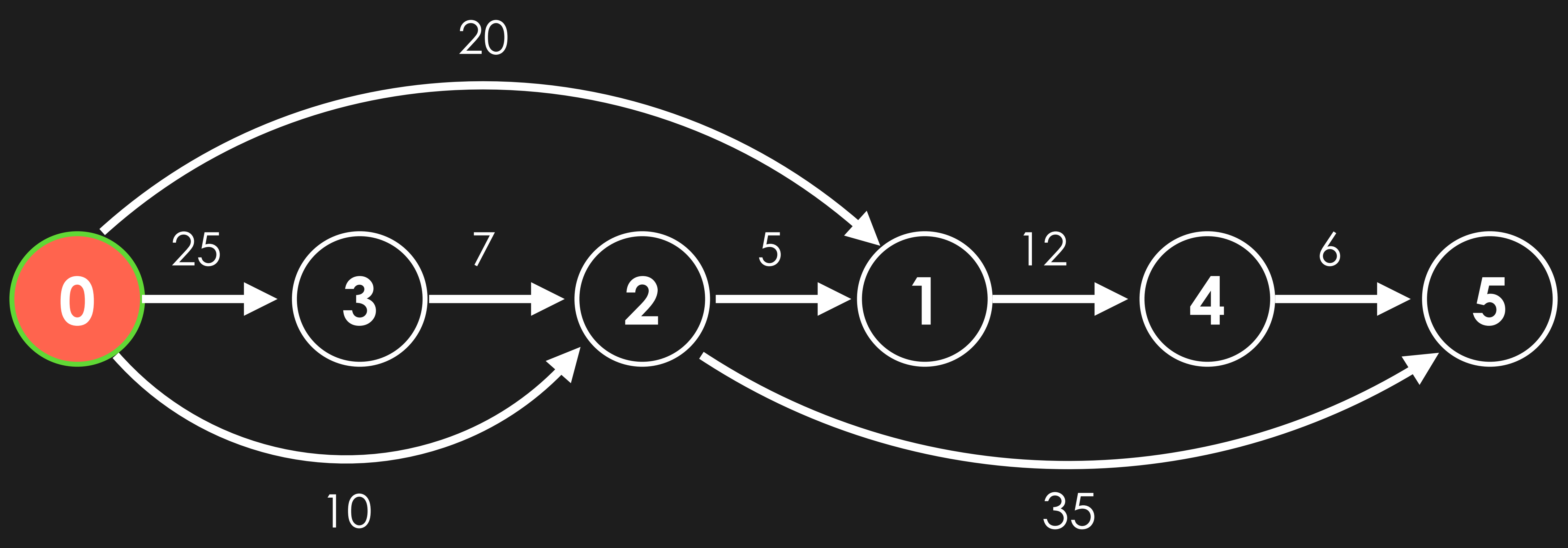
CodeIT

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	**0**	-1
1	INF	**1**	-1
2	INF	**2**	-1
3	INF	**3**	-1
4	INF	**4**	-1
5	INF	**5**	-1

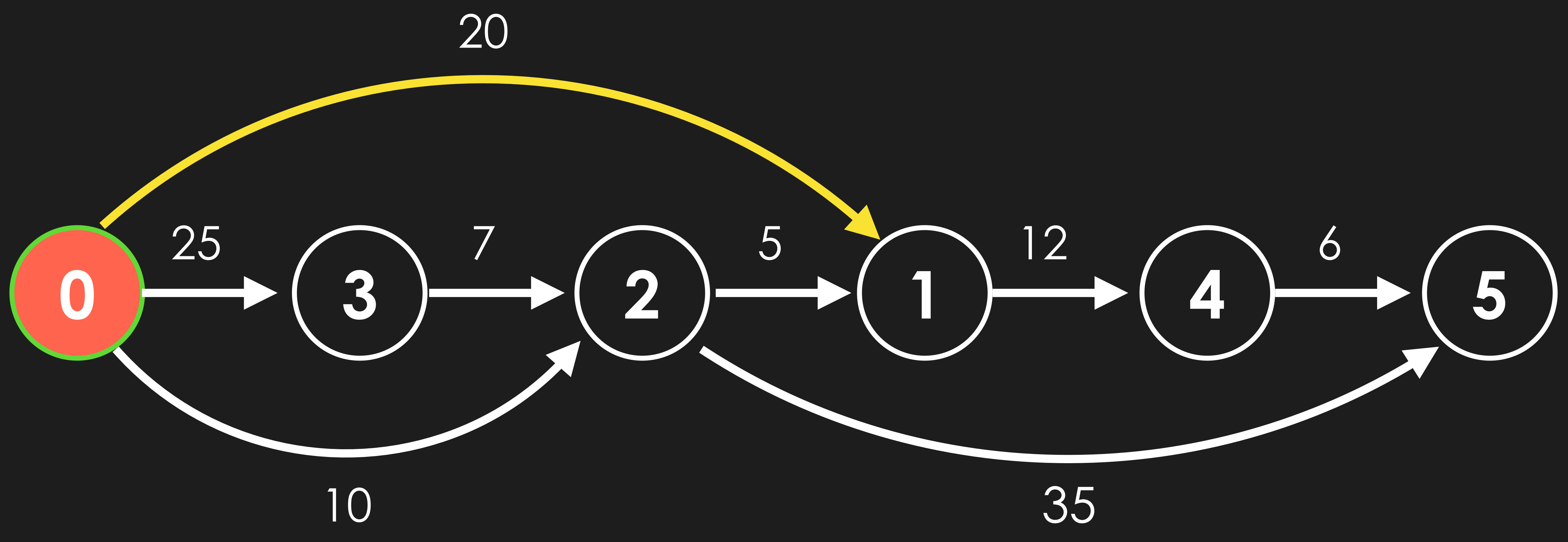
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo
0	0
1	INF
2	INF
3	INF
4	INF
5	INF

	edgeTo
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**

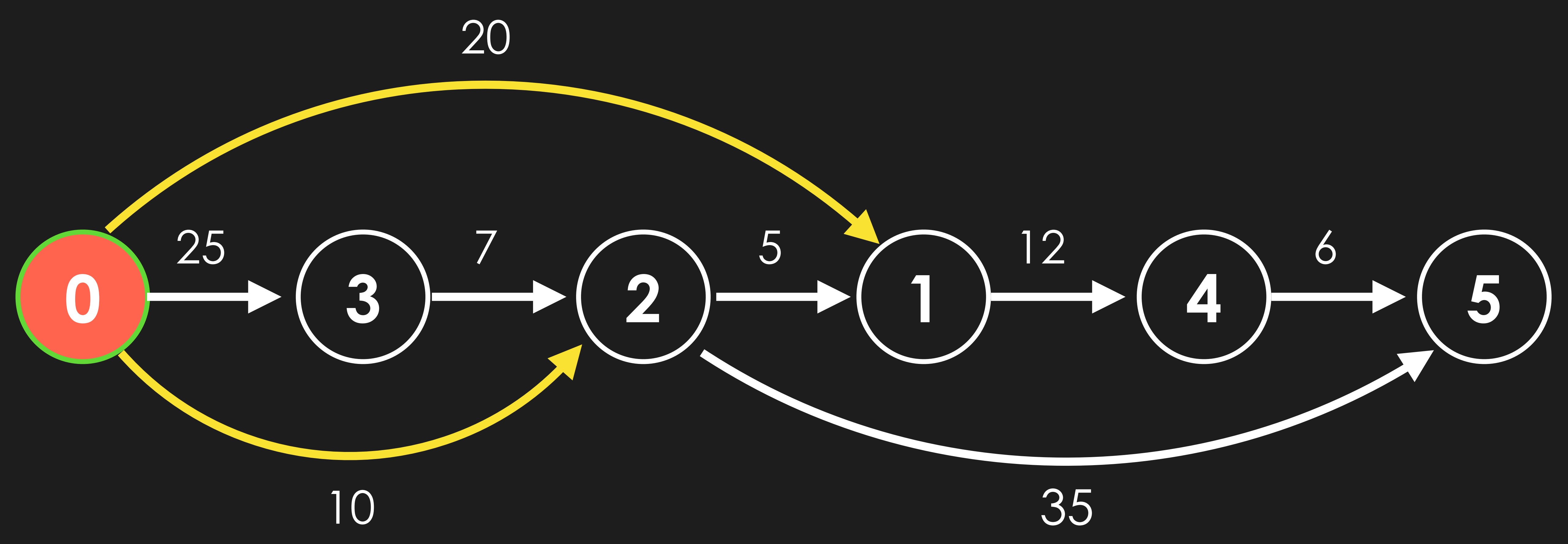


	distTo
0	0
1	20
2	INF
3	INF
4	INF
5	INF

	edgeTo
0	-1
1	0 - 1
2	-1
3	-1
4	-1
5	-1

CodeIT

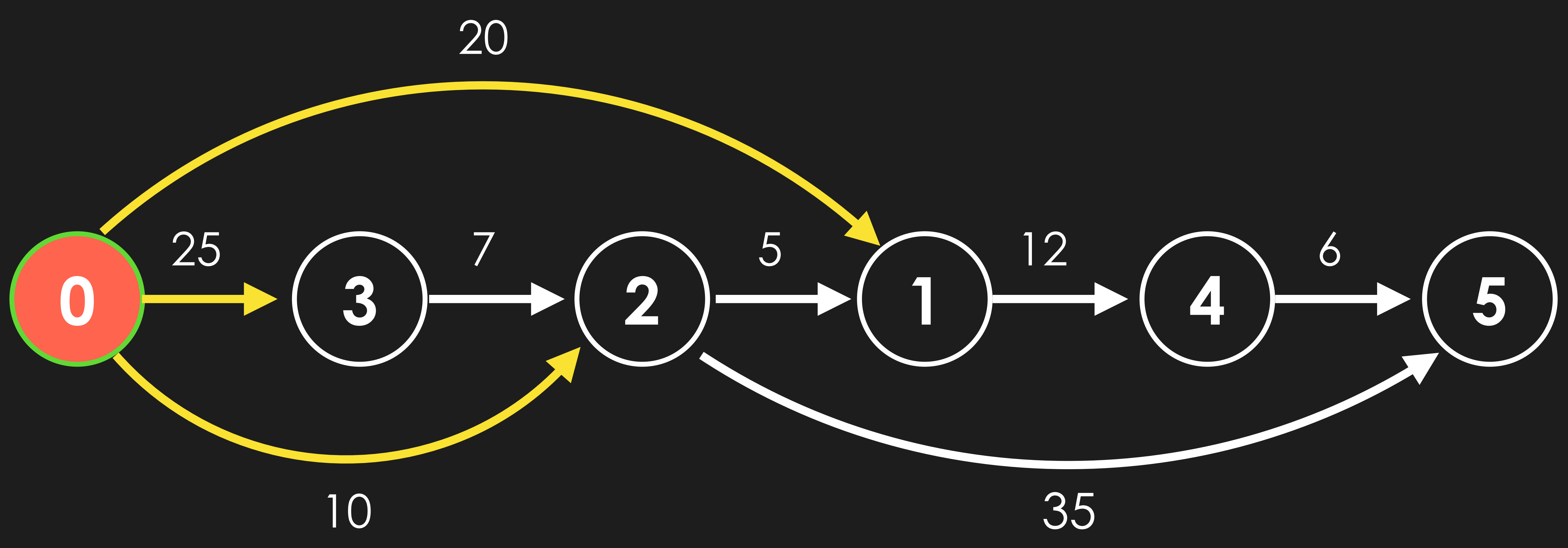
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	0	-1
1	20	1	0 - 1
2	10	2	0 - 2
3	INF	3	-1
4	INF	4	-1
5	INF	5	-1

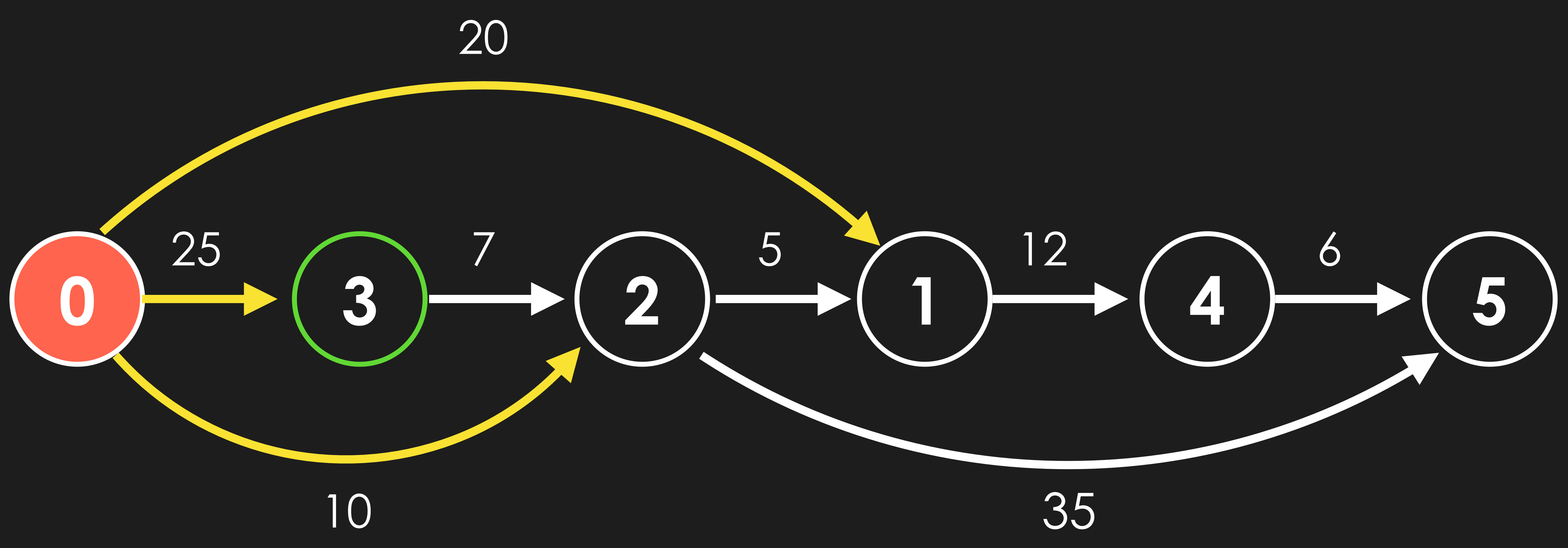
CodeIT

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	0	-1
1	20	1	0 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	INF	4	-1
5	INF	5	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



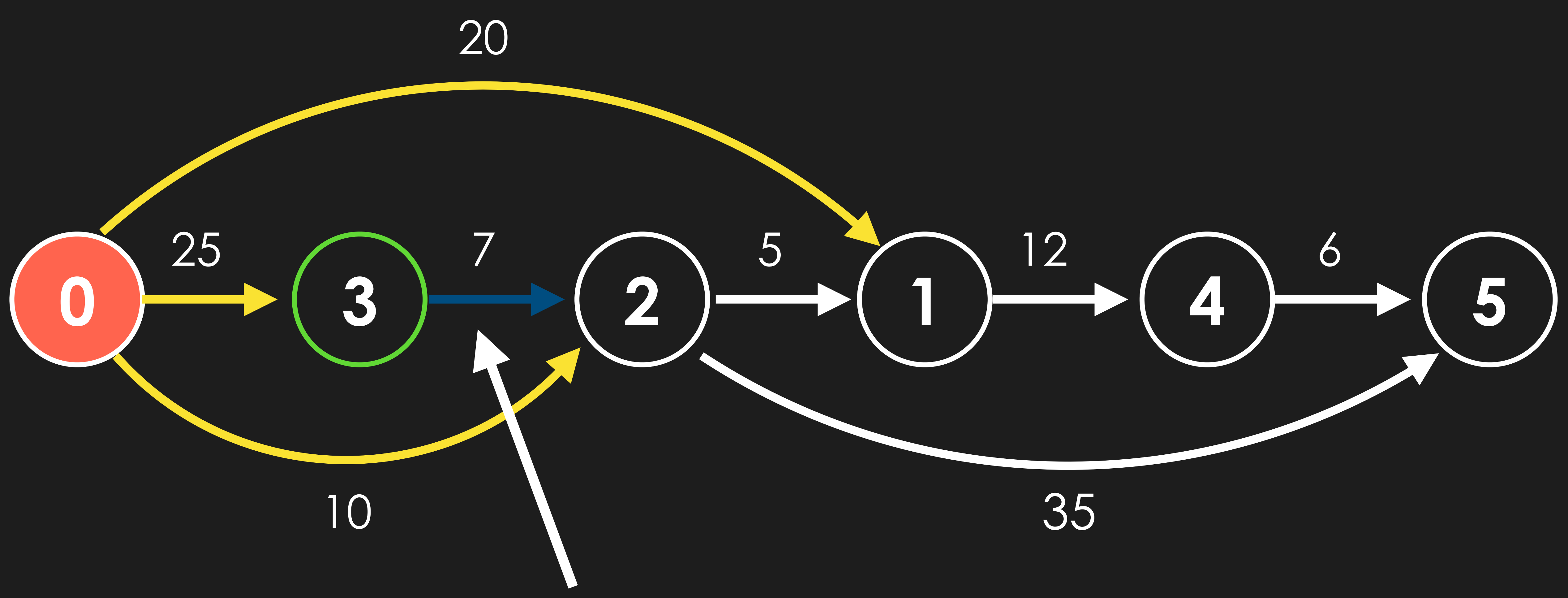
distTo

0	0
1	20
2	10
3	25
4	INF
5	INF

edgeTo

0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



distTo[3] + 7 = **25 + 7 >** distTo[2] = **10**

distTo

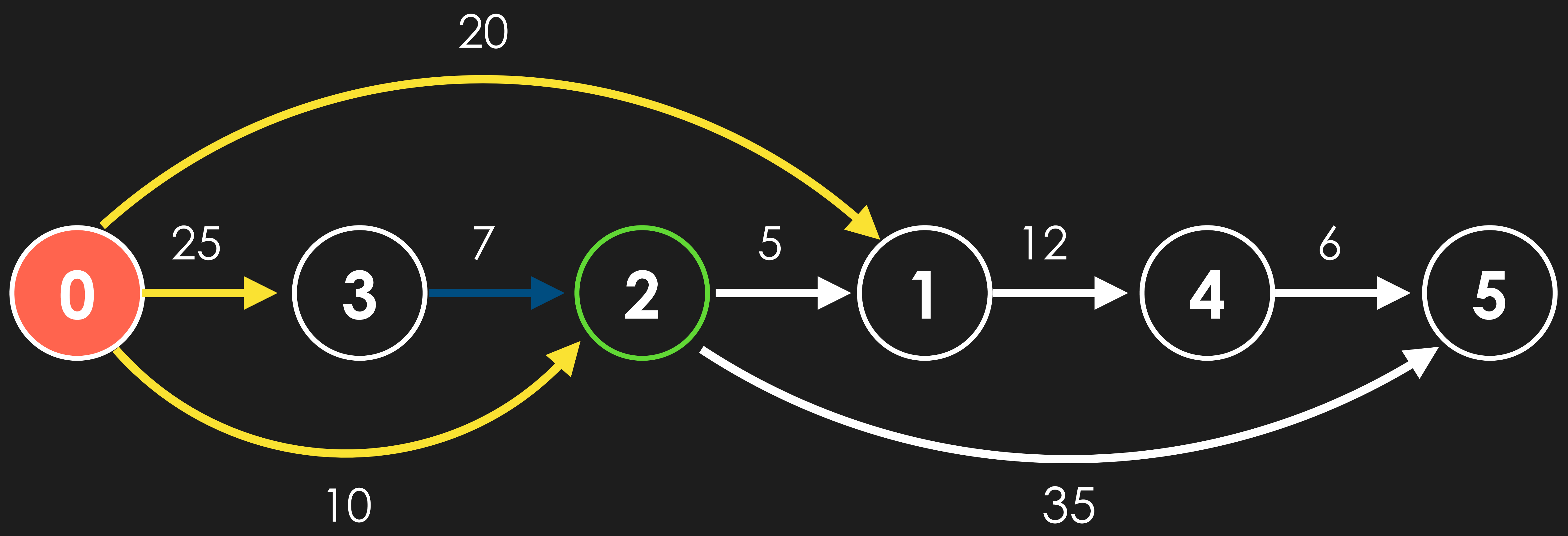
0	0
1	20
2	10
3	25
4	INF
5	INF

edgeTo

0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1

CodeIT

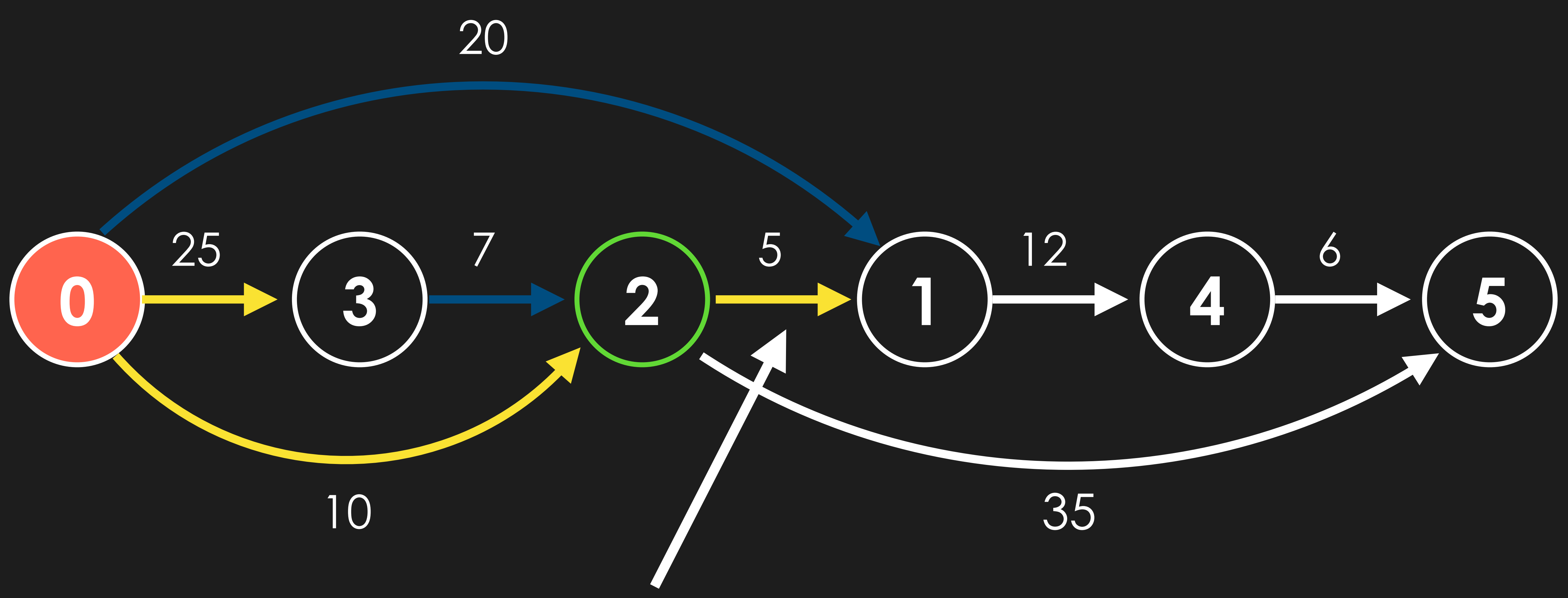
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo
0	0
1	20
2	10
3	25
4	INF
5	INF

	edgeTo
0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1

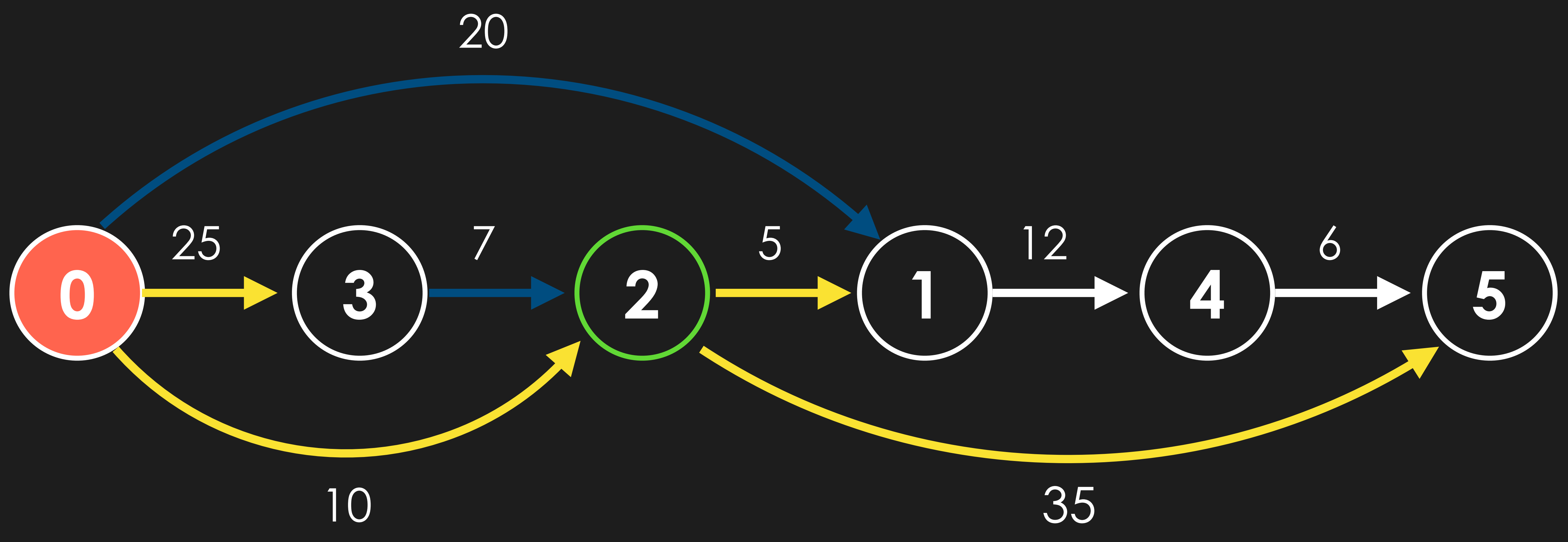
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



relaxed since **distTo[2] + weight** (10 + 5) **< distTo[1]** (20)

	distTo		edgeTo
0	0	**0**	-1
1	**15**	**1**	**2 - 1**
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	INF	**4**	-1
5	INF	**5**	-1

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



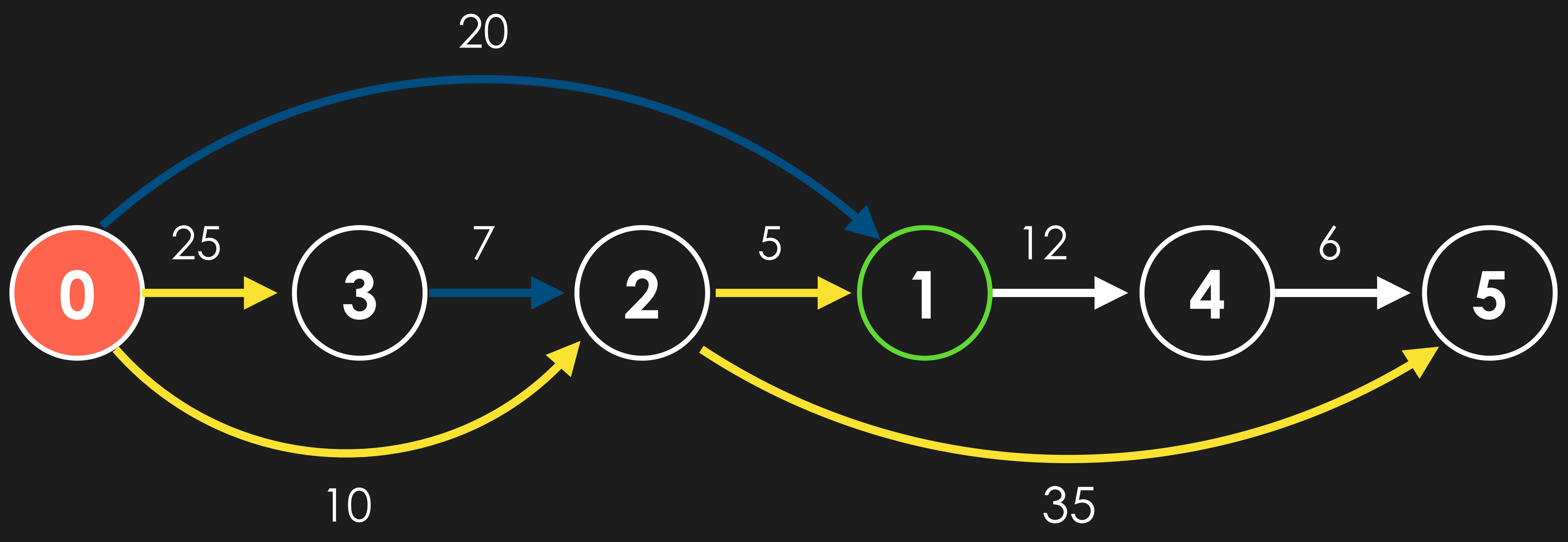
distTo

0	0
1	**15**
2	10
3	25
4	INF
5	**45**

edgeTo

0	-1
1	**2 - 1**
2	0 - 2
3	0 - 3
4	-1
5	**2 - 5**

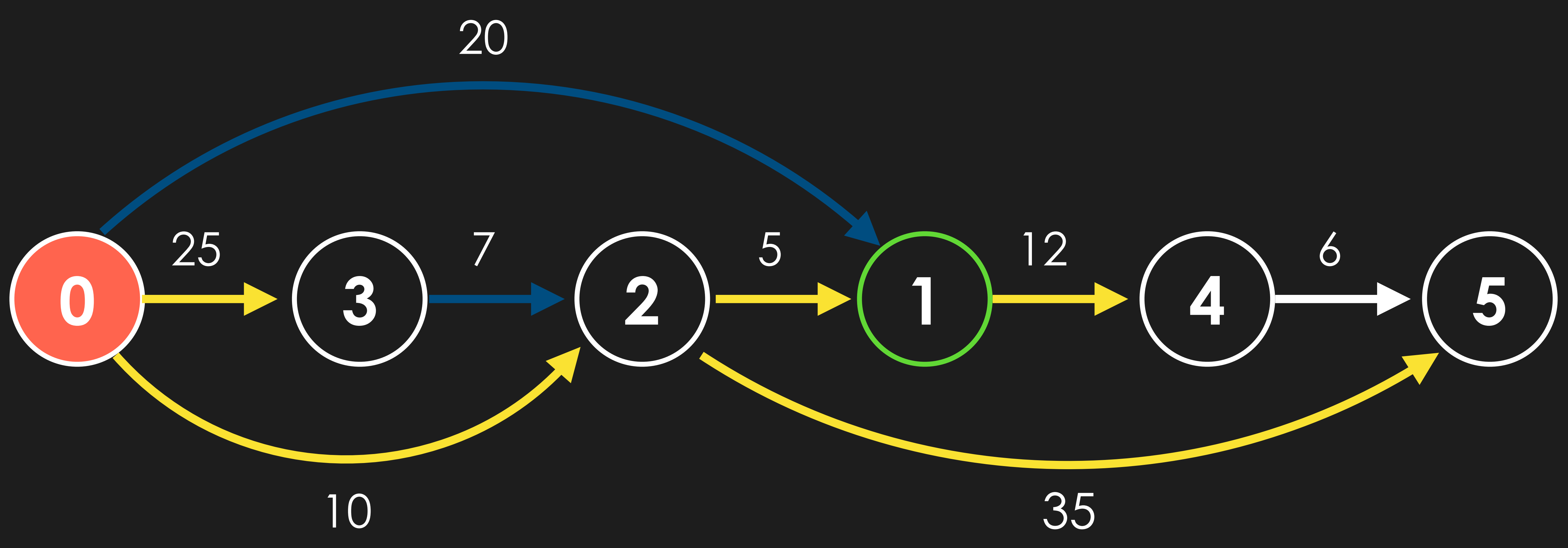
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	**0**	-1
1	15	**1**	2 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	INF	**4**	-1
5	45	**5**	2 - 5

CodeIT

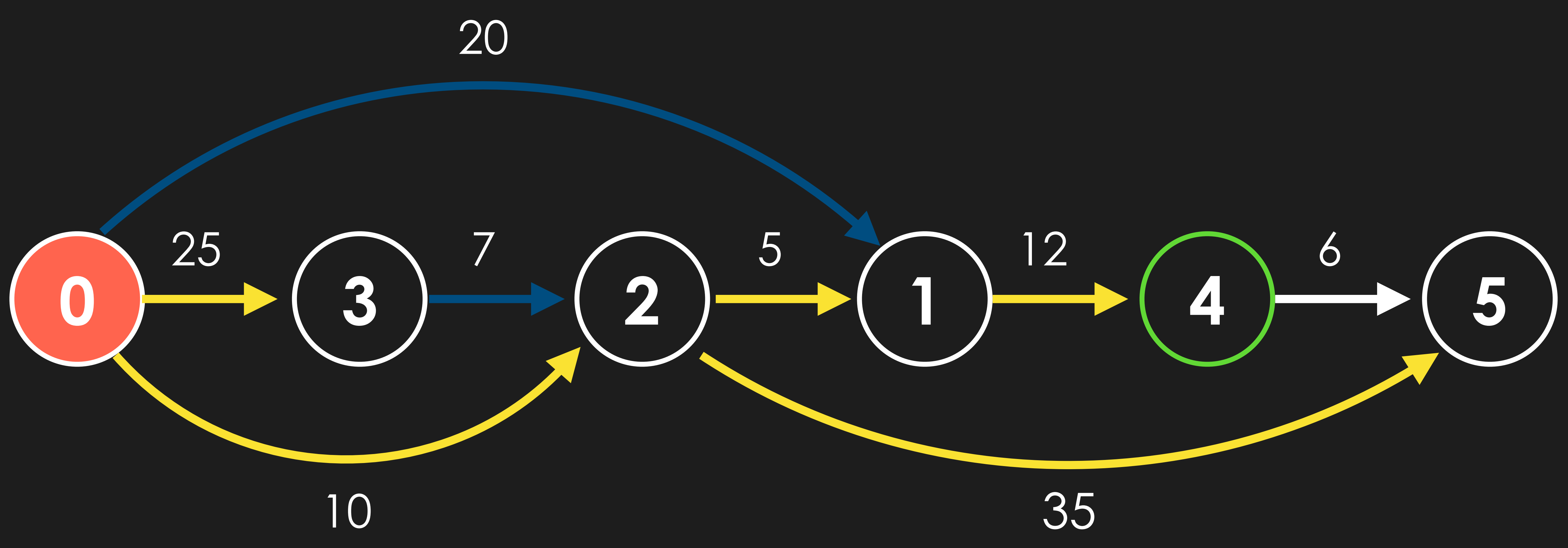
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	**0**	-1
1	15	**1**	2 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	**27**	**4**	**1 - 4**
5	45	**5**	2 - 5

CodeIT

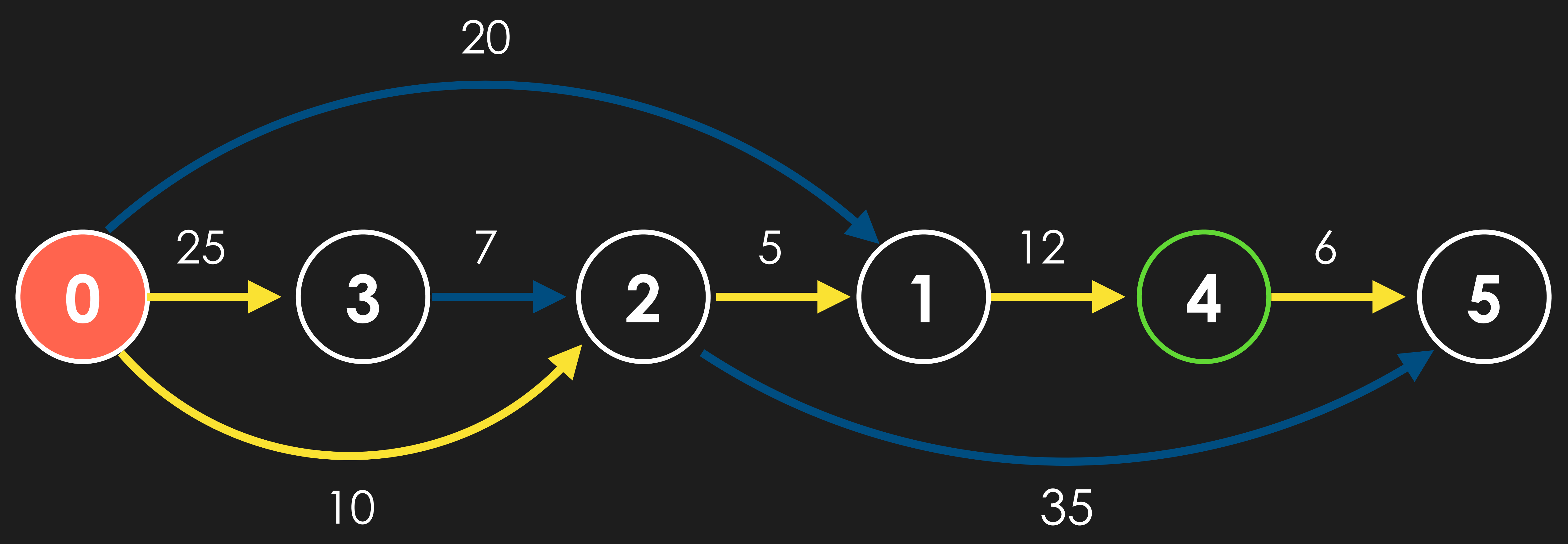
1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo
0	0
1	15
2	10
3	25
4	27
5	45

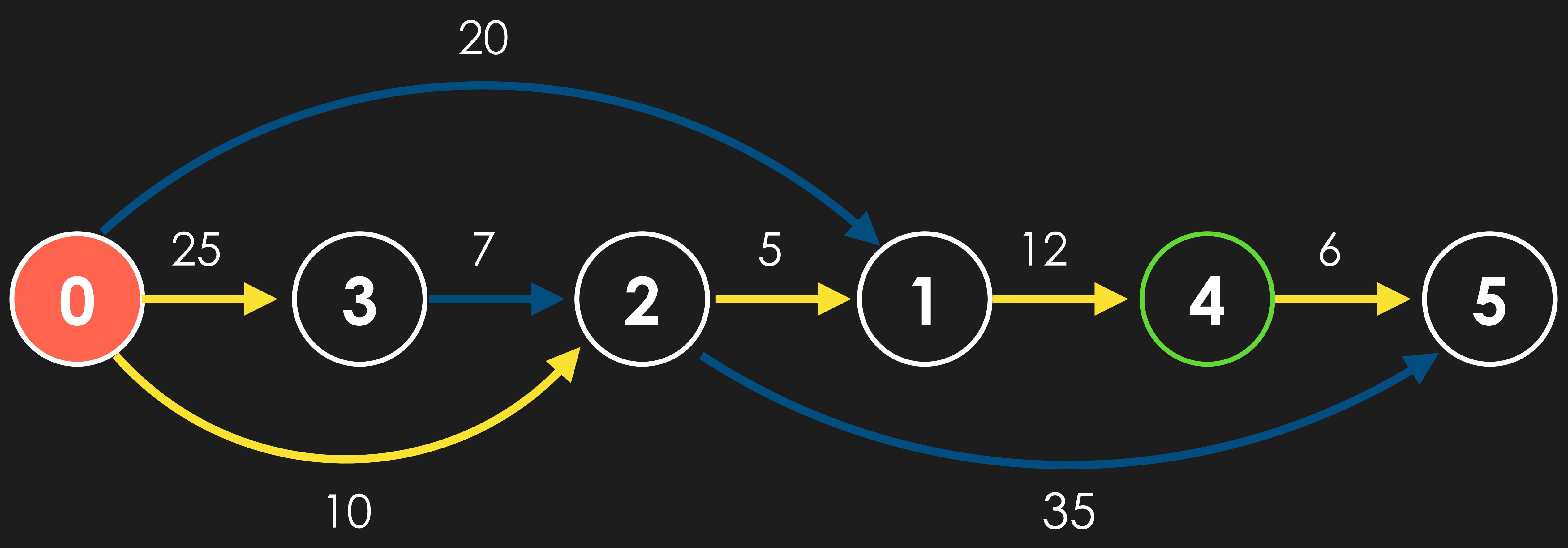
	edgeTo
0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	**0**	-1
1	15	**1**	2 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	27	**4**	1 - 4
5	**33**	**5**	**4 - 5**

1. Perform **top sort** on the graph
2. Set distTo source vertex to be 0
3. For each vertex from left to right in the sorted order, **relax all adjacent edges**



	distTo		edgeTo
0	0	0	-1
1	15	1	2 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	27	4	1 - 4
5	**33**	5	**4 - 5**

What if the graph contains cycles?

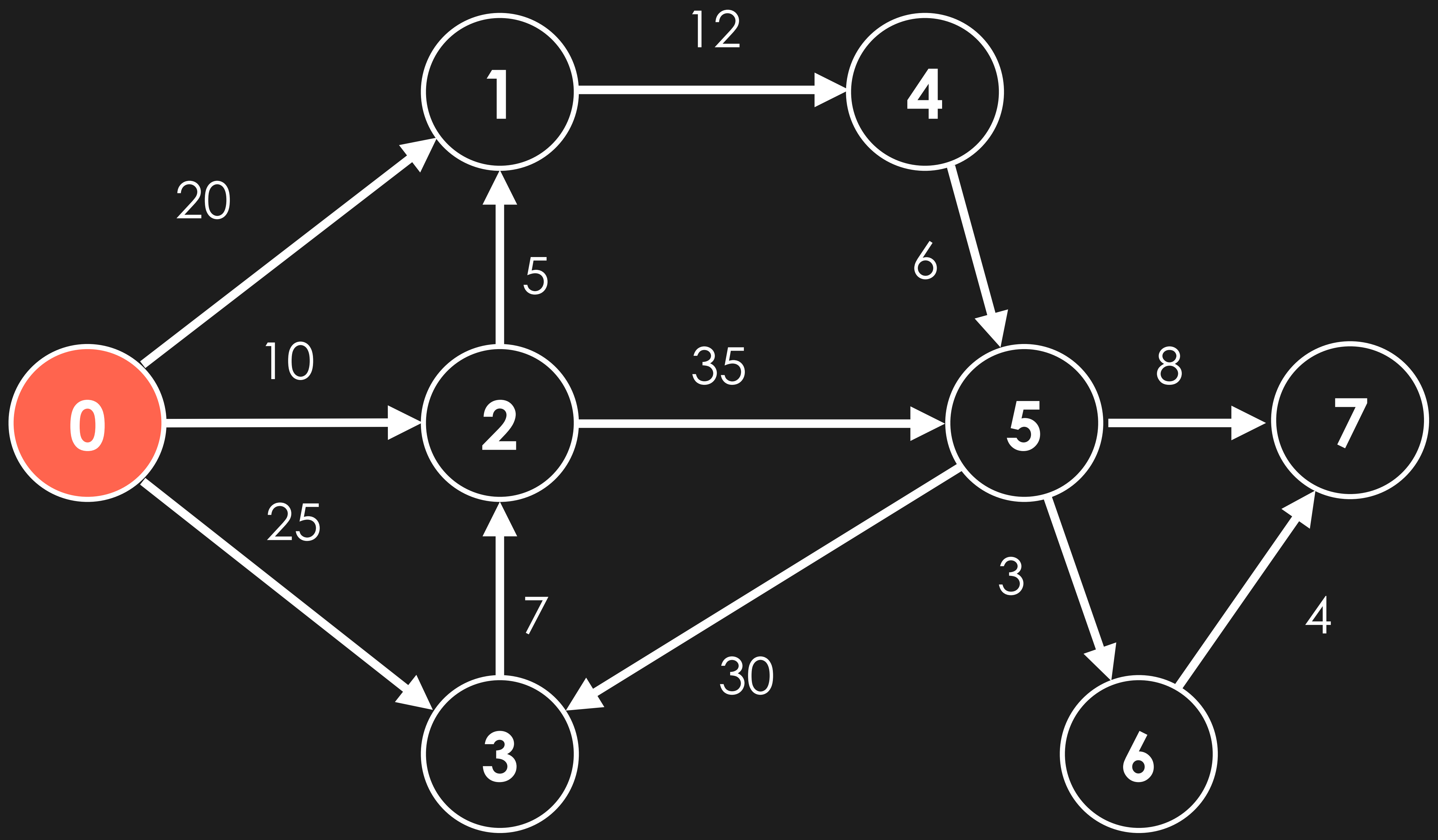
Dijkstra's Algorithm

CodeIT

Dijkstra's Algorithm

1. Initialise all vertices in **distTo** to **positive infinity** (or None)
2. Initialise all vertices in **edgeTo** to -1
3. Set **distTo** **source vertex to 0** and insert into **pq** where **key is vertex and value is distTo[vertex]**
4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**

1. Initialise all vertices in **distTo** to infinity (or None)
2. Initialise all vertices in **edgeTo** to -1
3. Set **distTo for source vertex to 0** and insert into **pq** where key is vertex and value is distTo[vertex]
4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



pq

vertex	distTo

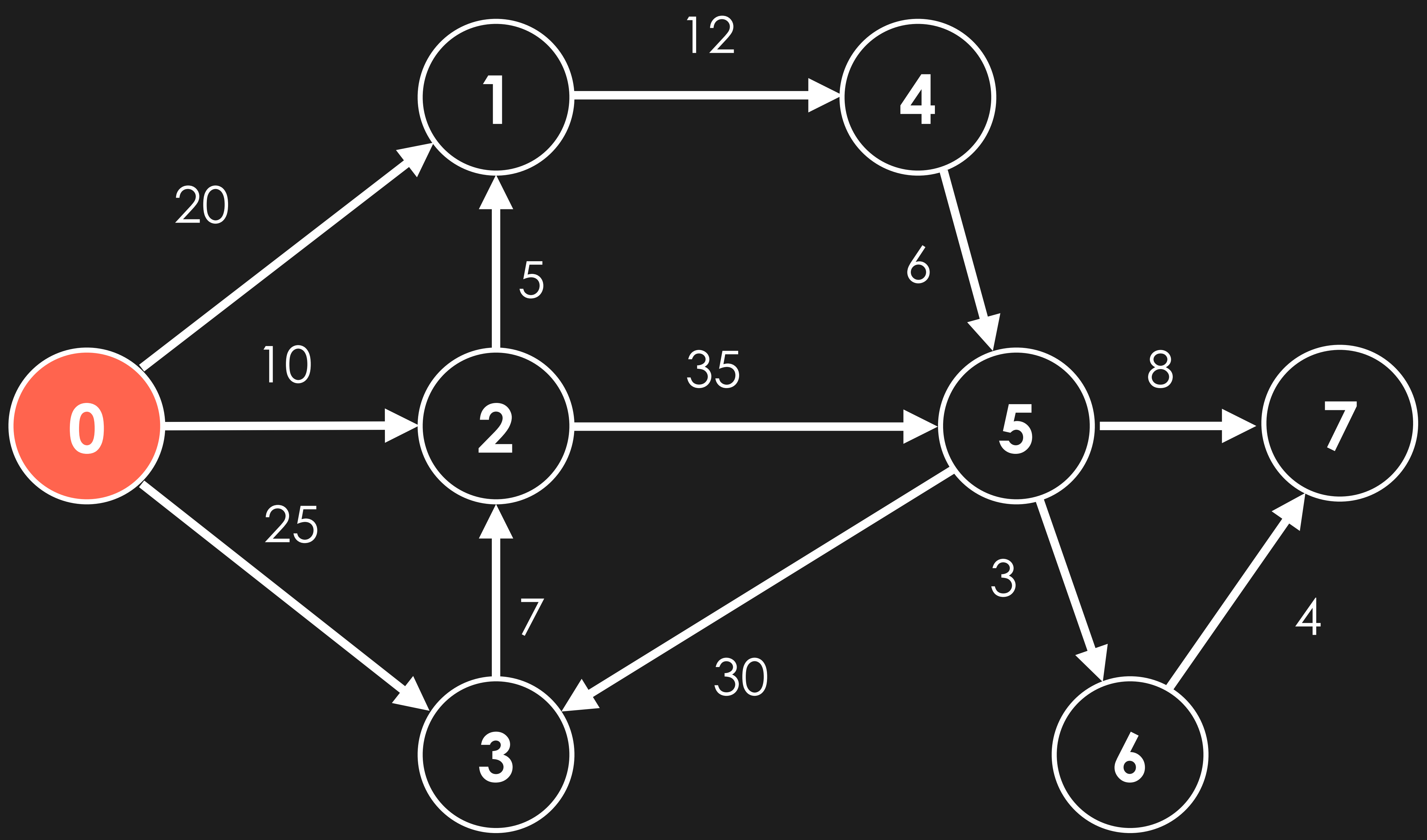
distTo

0	INF
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

1. Initialise all vertices in **distTo** to infinity (or None)
2. Initialise all vertices in **edgeTo** to -1
3. Set **distTo for source vertex to 0** and insert into **pq** where key is vertex and value is distTo[vertex]
4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



pq

vertex	distTo
0	0

distTo

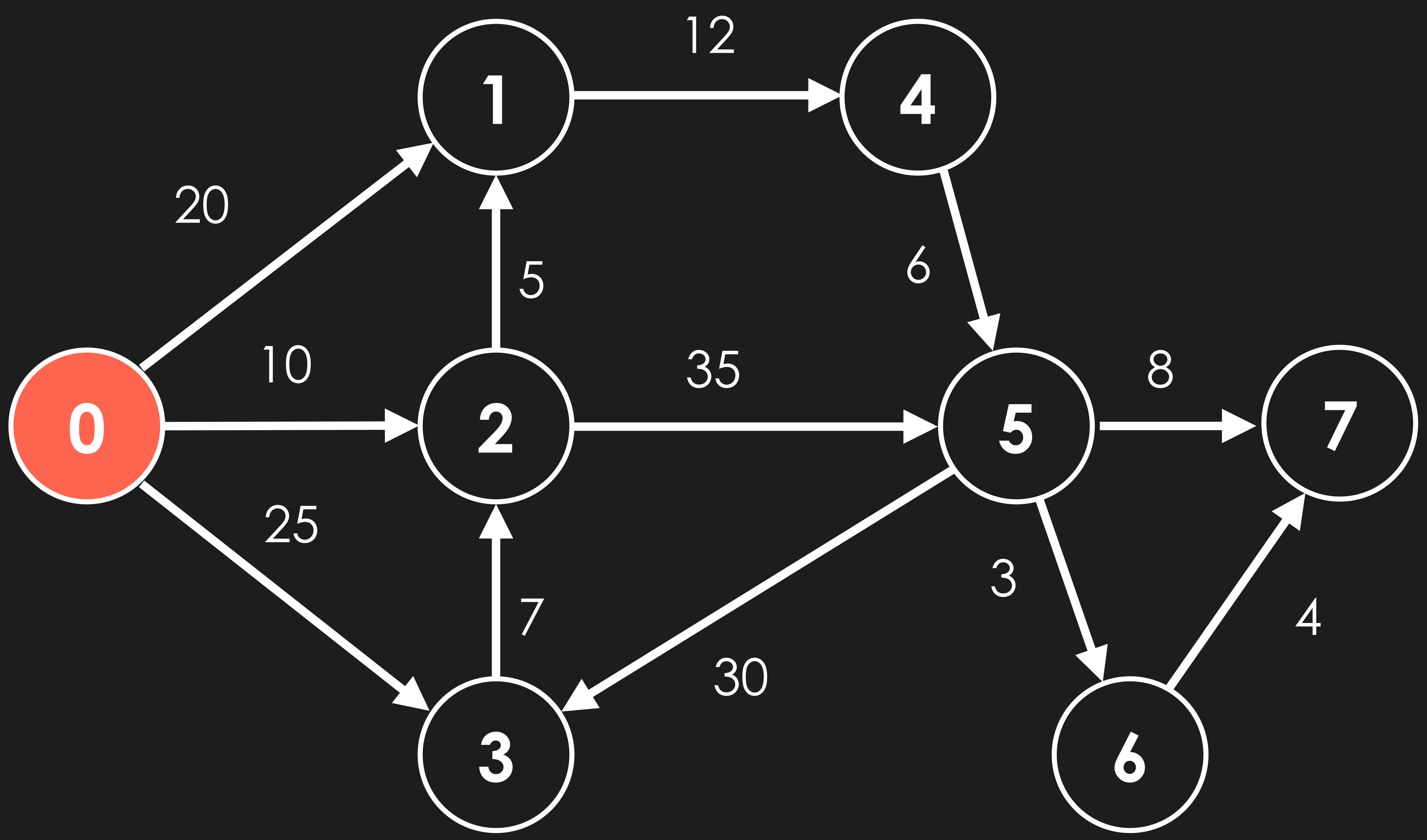
0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



pq

vertex	distTo
0	0

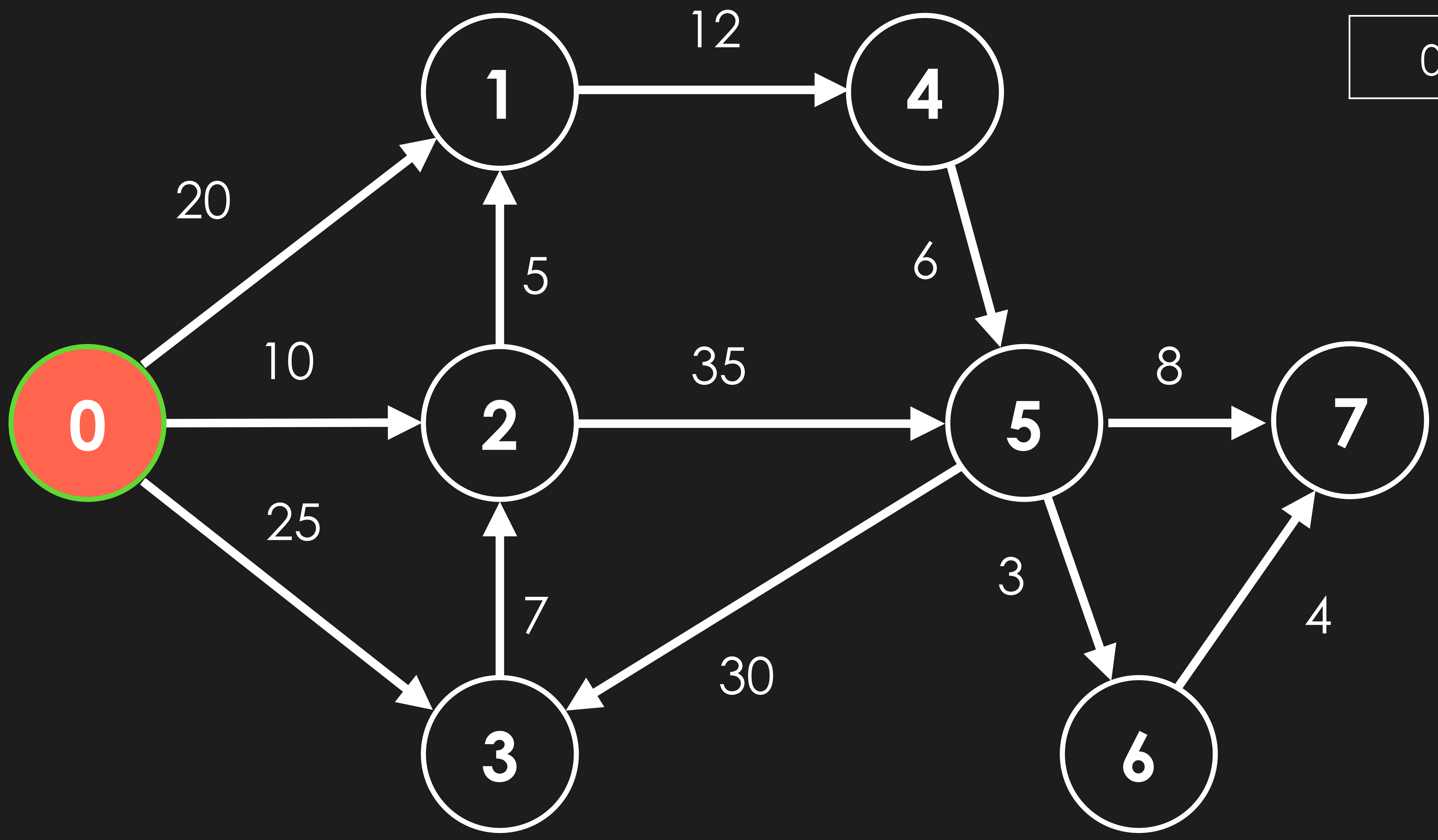
distTo

0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



0	0

pq

vertex	distTo

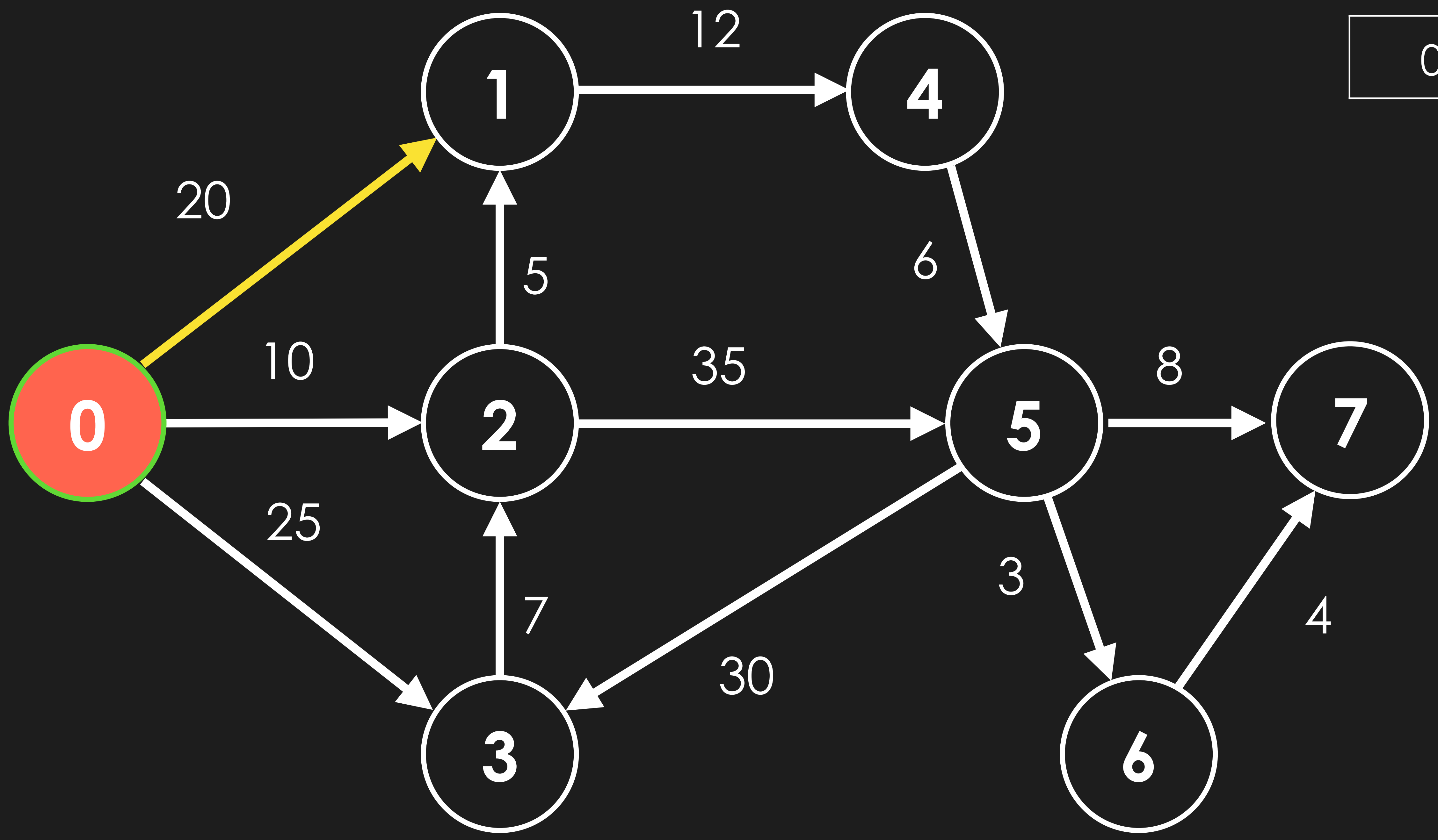
distTo

0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



0	0

pq

vertex	distTo
1	20

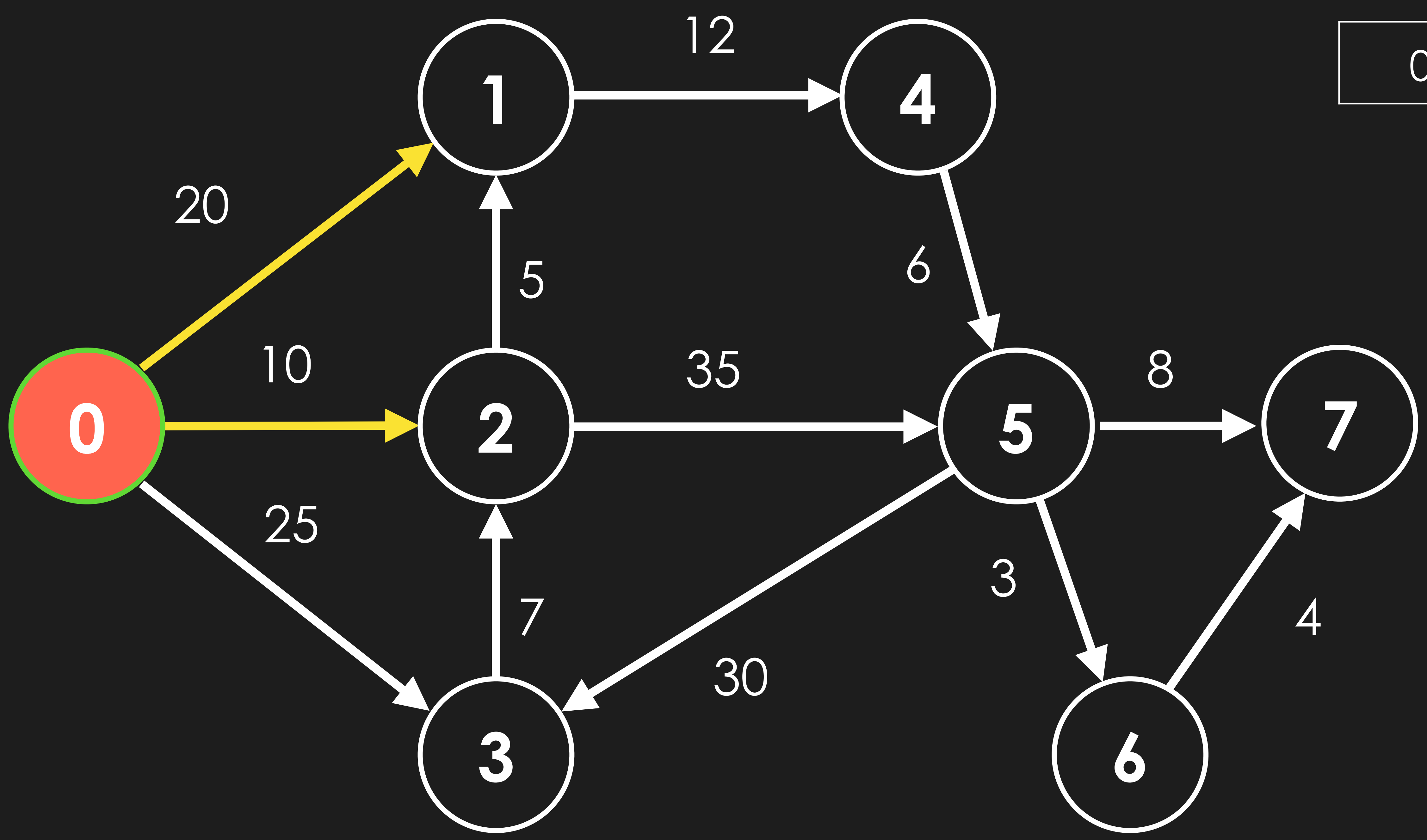
distTo

0	0
1	20
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	0 - 1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



0	0

pq

vertex	distTo
2	**10**
1	**20**

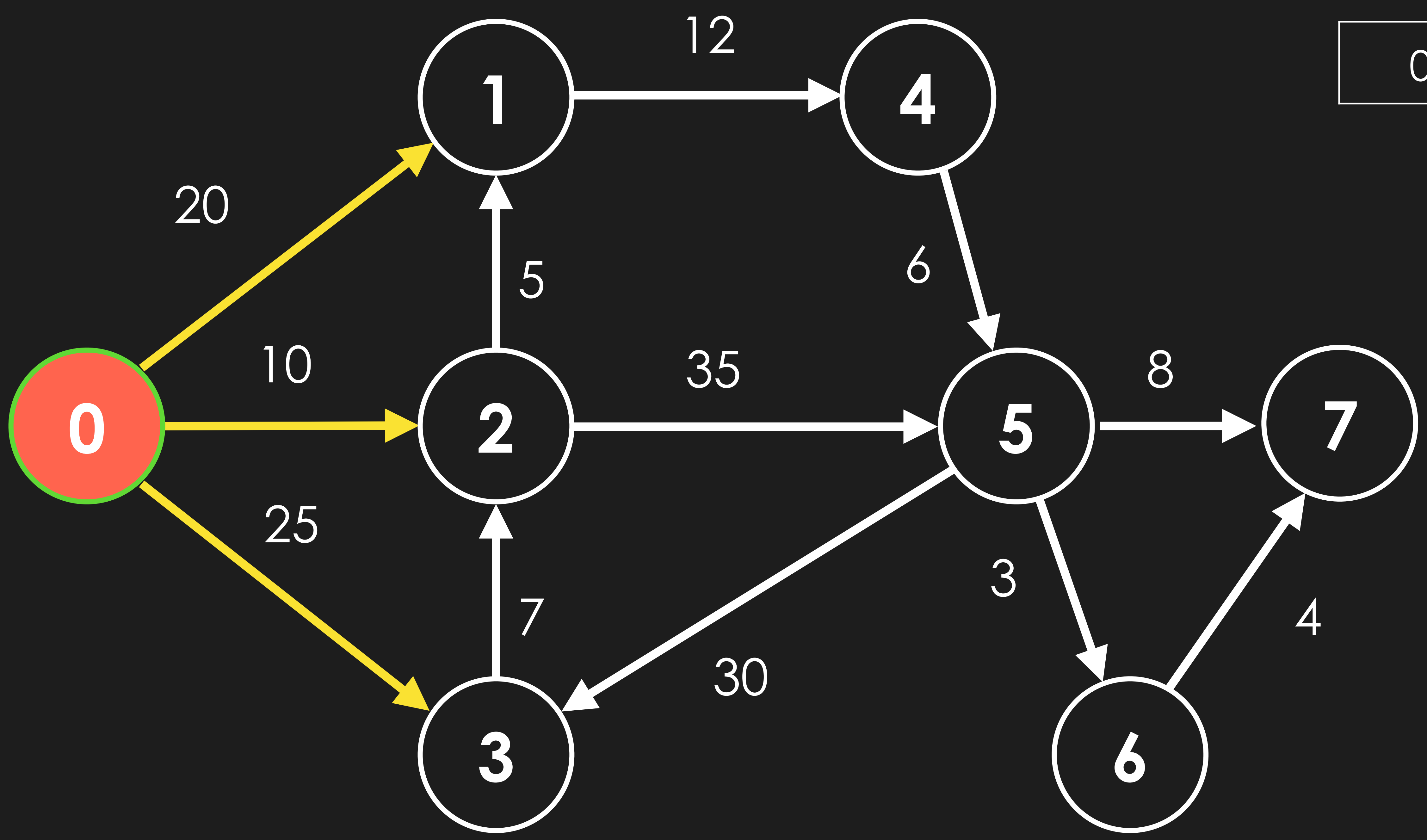
distTo

0	0
1	**20**
2	**10**
3	INF
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	**0 - 1**
2	**0 - 2**
3	-1
4	-1
5	-1
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



0	0

pq

vertex	distTo
2	10
1	20
3	25

distTo

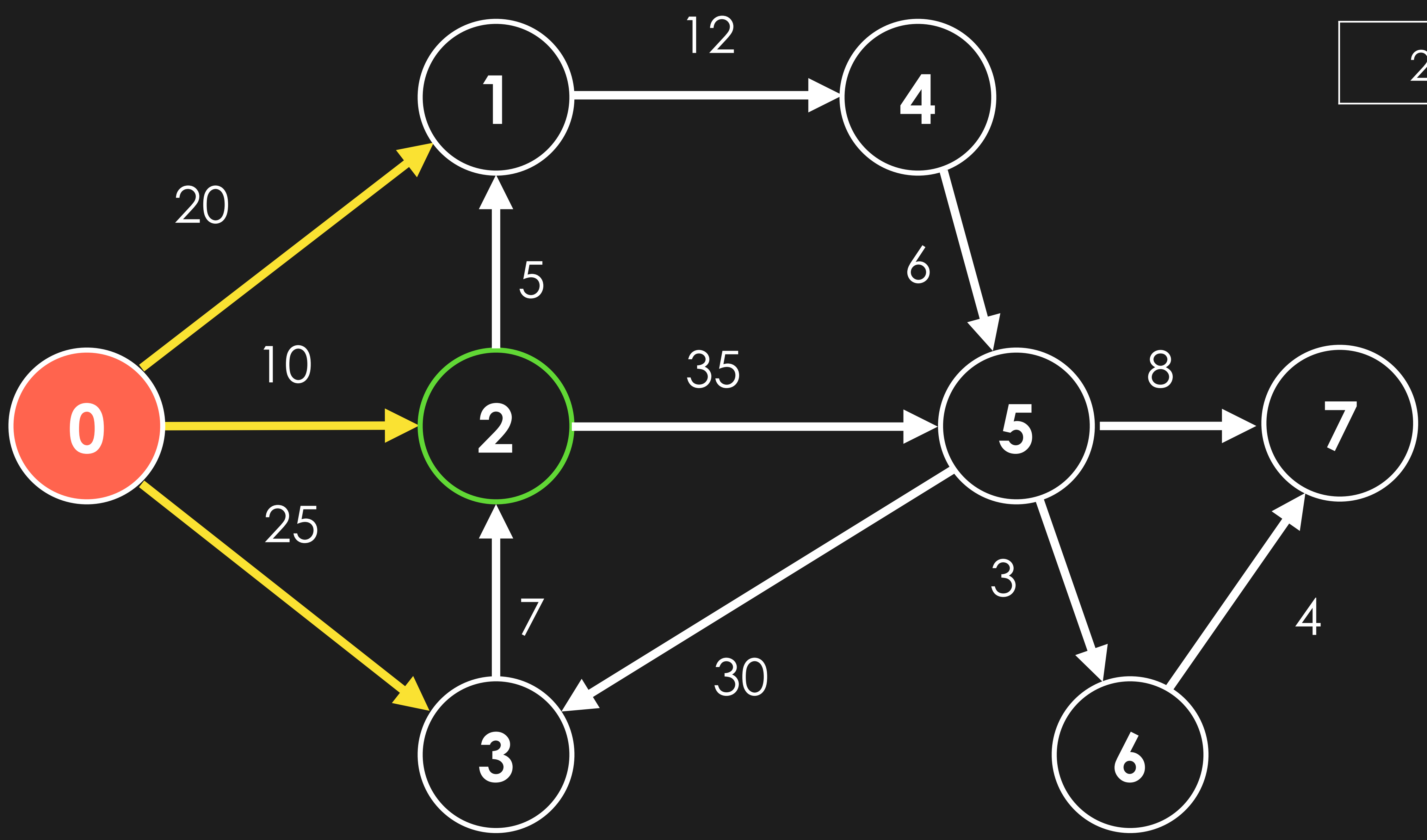
0	0
1	20
2	10
3	25
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



2	10

pq

vertex	distTo
1	20
3	25

distTo

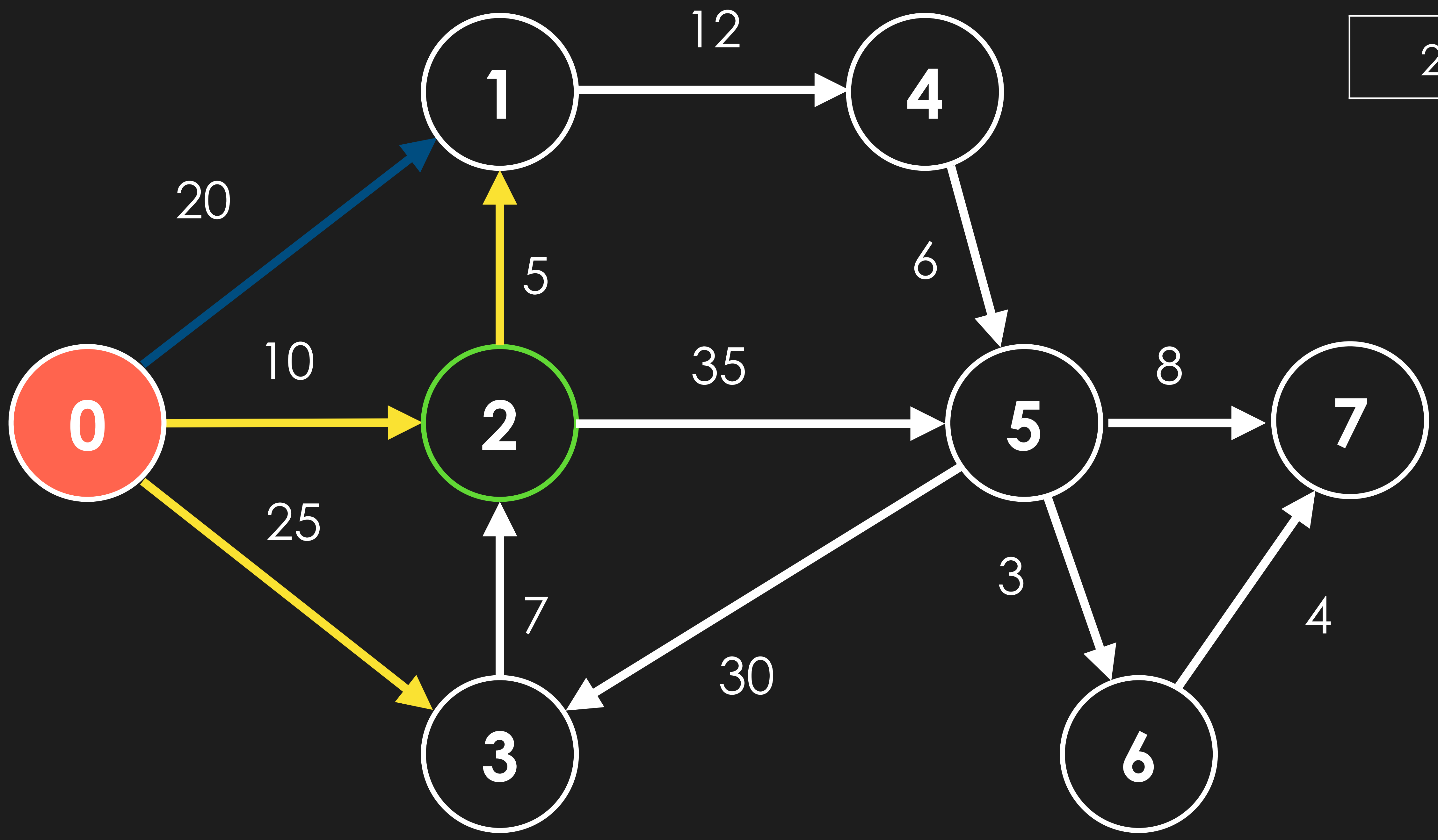
0	0
1	20
2	10
3	25
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



2	10

pq

vertex	distTo
1	**15**
3	25

distTo

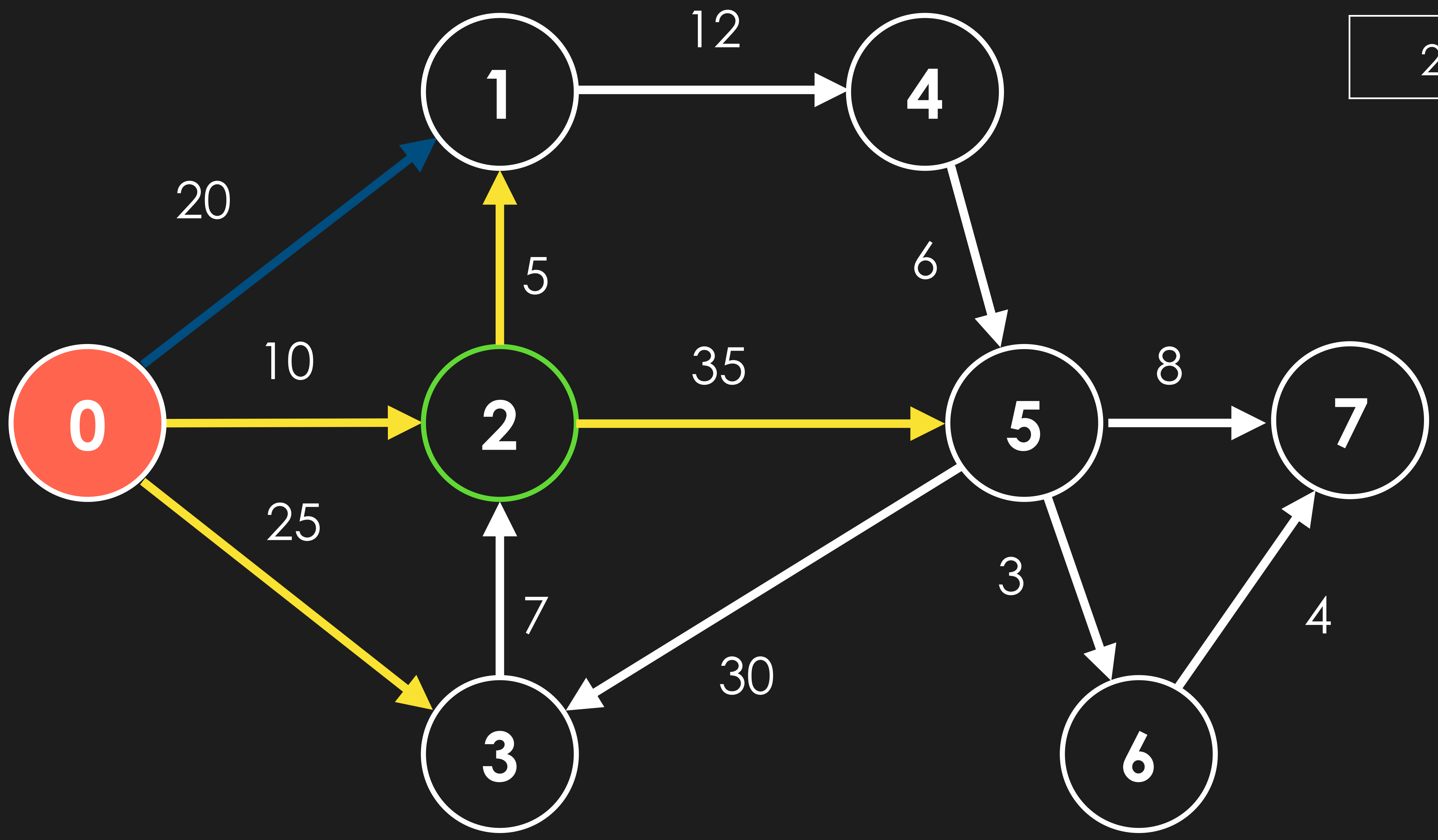
0	0
1	**15**
2	10
3	25
4	INF
5	INF
6	INF
7	INF

edgeTo

0	-1
1	**2 - 1**
2	0 - 2
3	0 - 3
4	-1
5	-1
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



2	10

pq

vertex	distTo
1	15
3	25
5	45

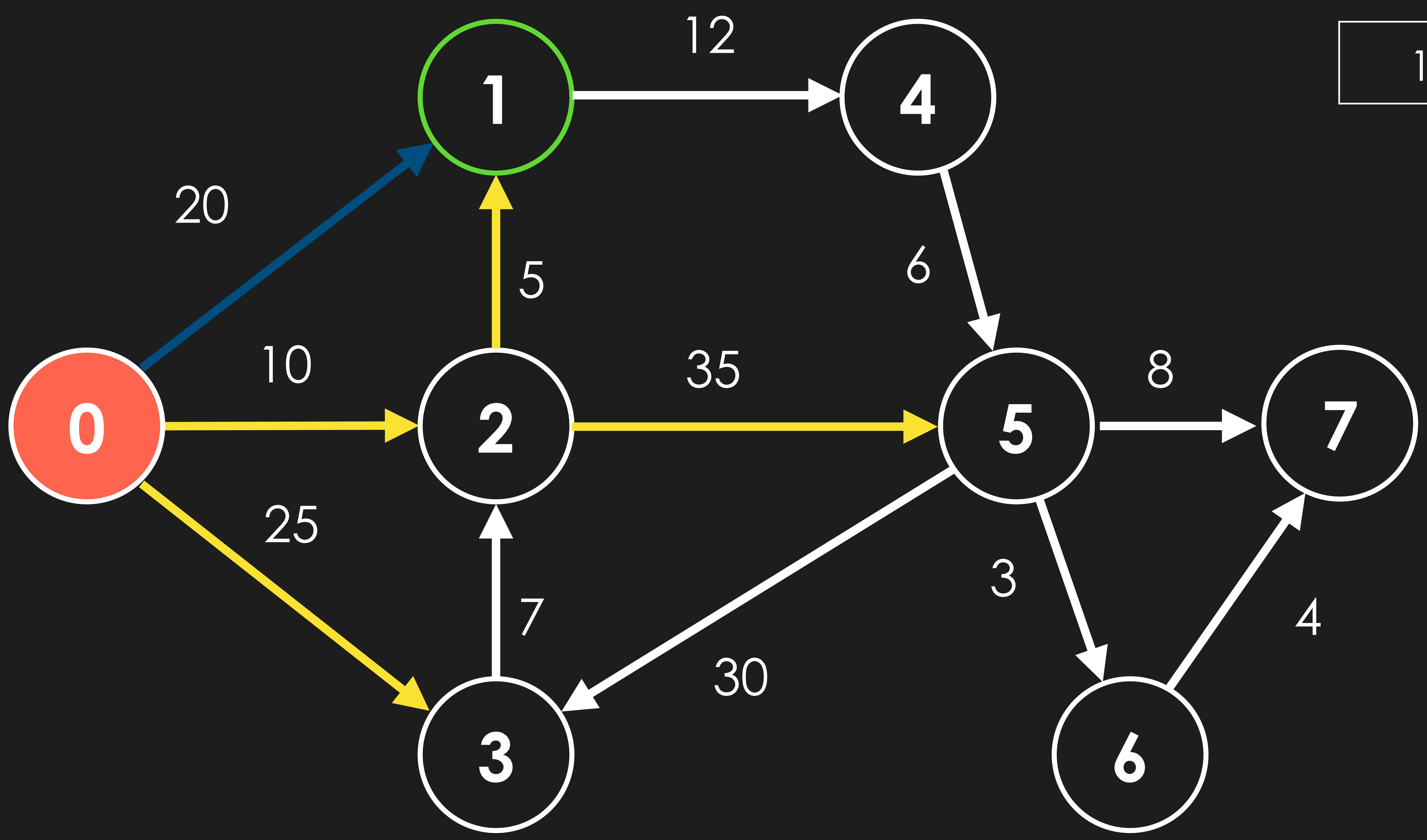
distTo

0	0
1	15
2	10
3	25
4	INF
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	-1
5	2 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



1	15

pq

vertex	distTo
3	25
5	45

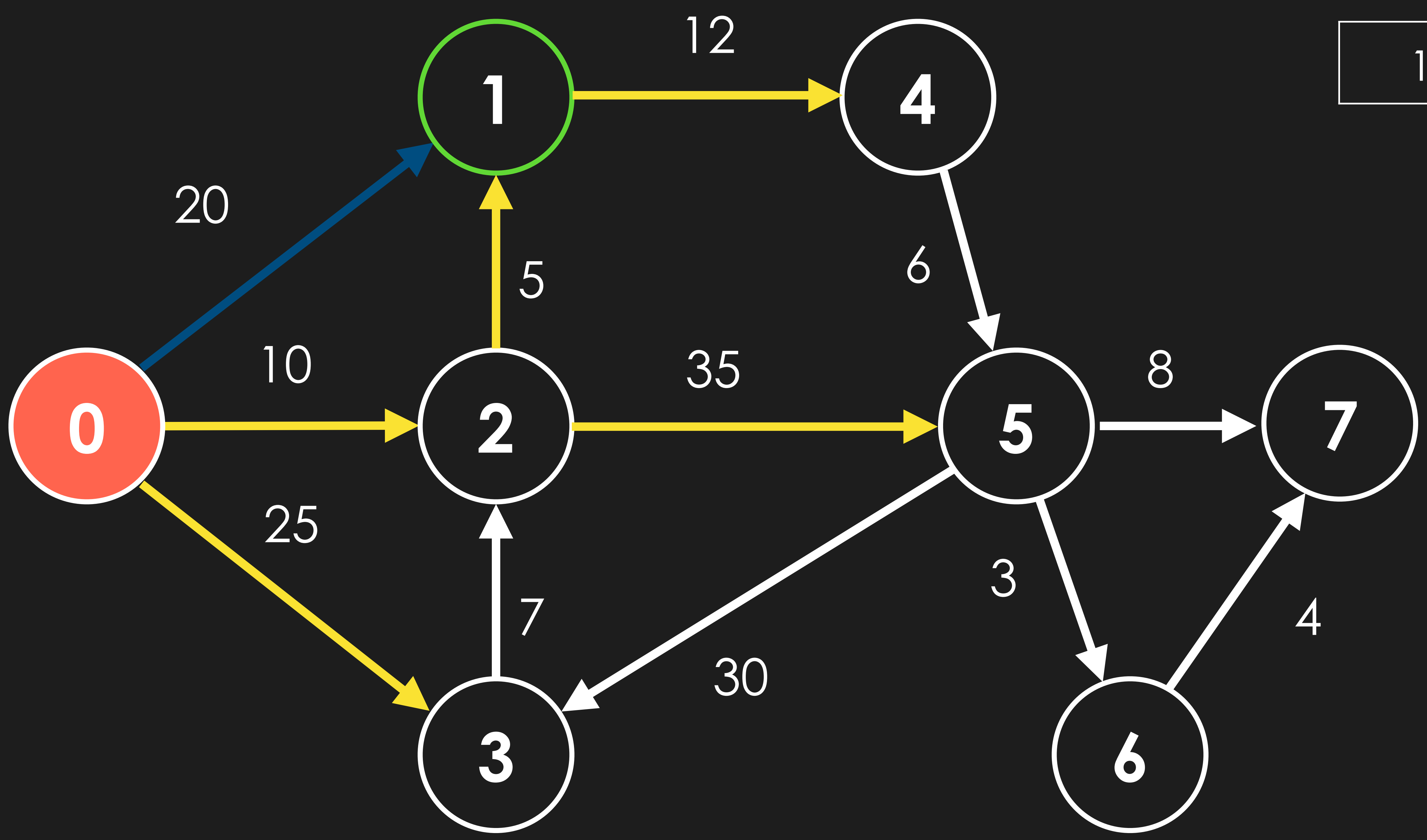
distTo

0	0
1	15
2	10
3	25
4	INF
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	-1
5	2 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



1	15

pq

vertex	distTo
3	25
4	**27**
5	45

distTo

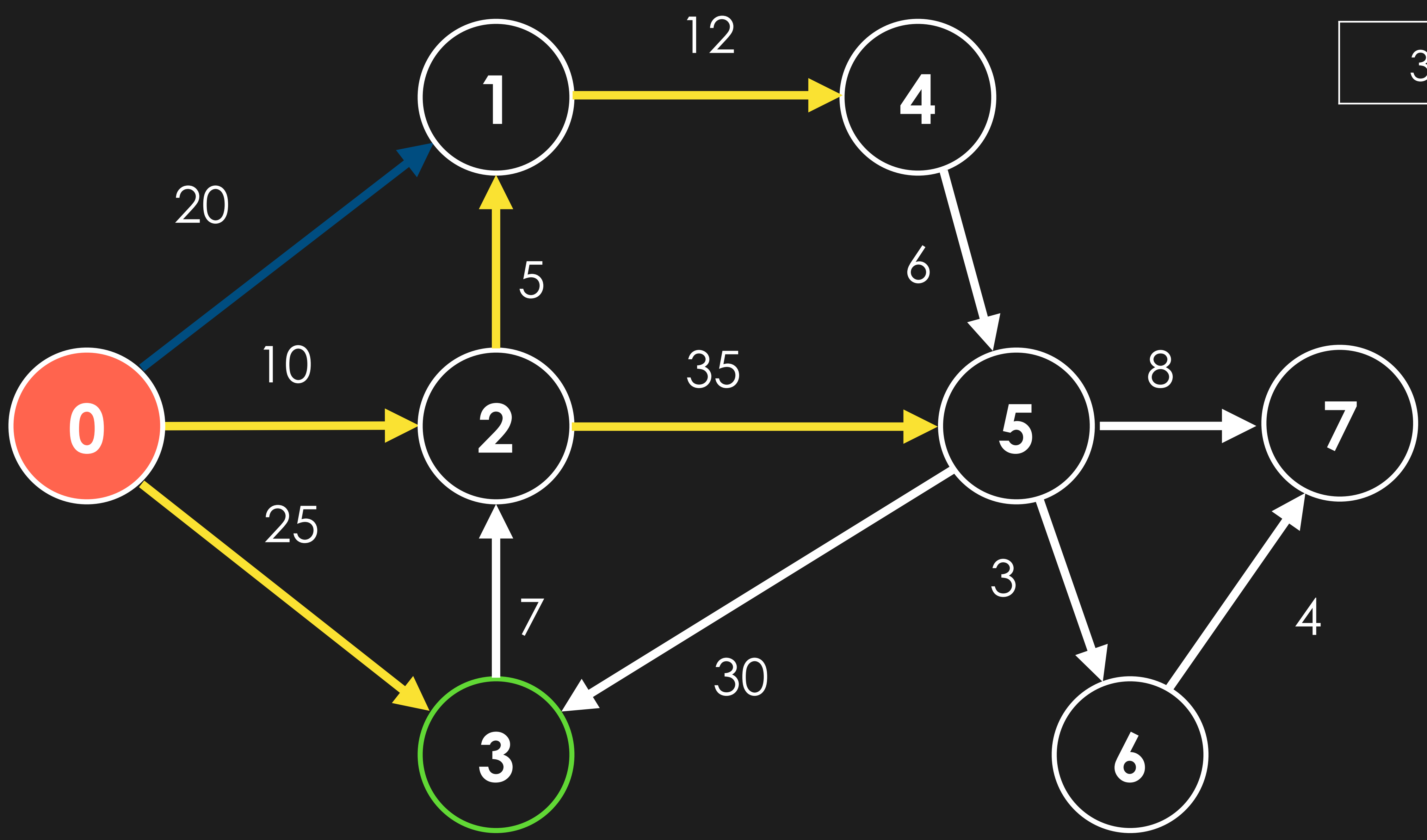
0	0
1	15
2	10
3	25
4	**27**
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	**1 - 4**
5	2 - 5
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



3	25

pq

vertex	distTo
4	27
5	45

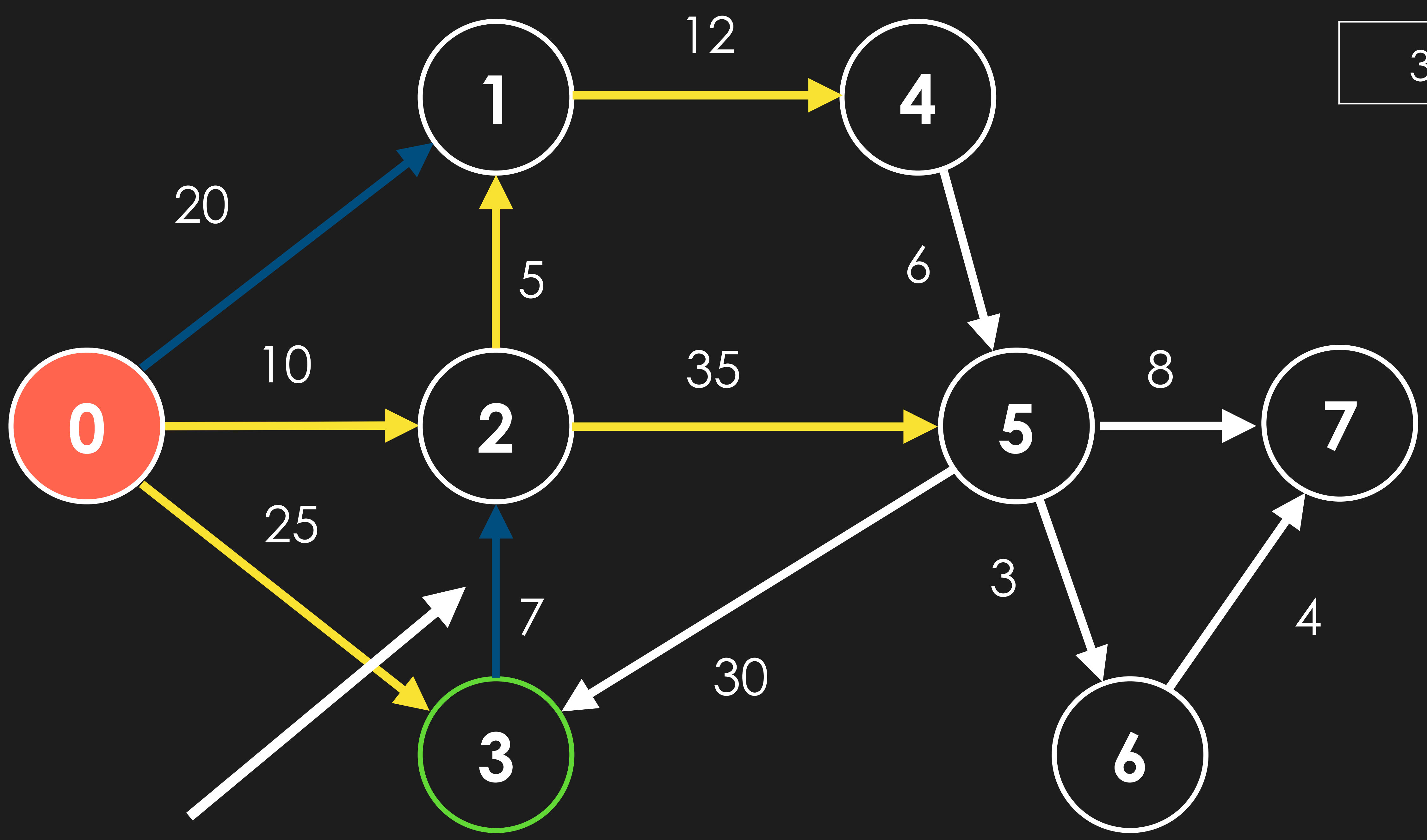
distTo

0	0
1	15
2	10
3	25
4	27
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



distTo[3] + weight = **25 + 7 >** distTo[2] = **10**

3	25

pq

vertex	distTo
4	27
5	45

distTo

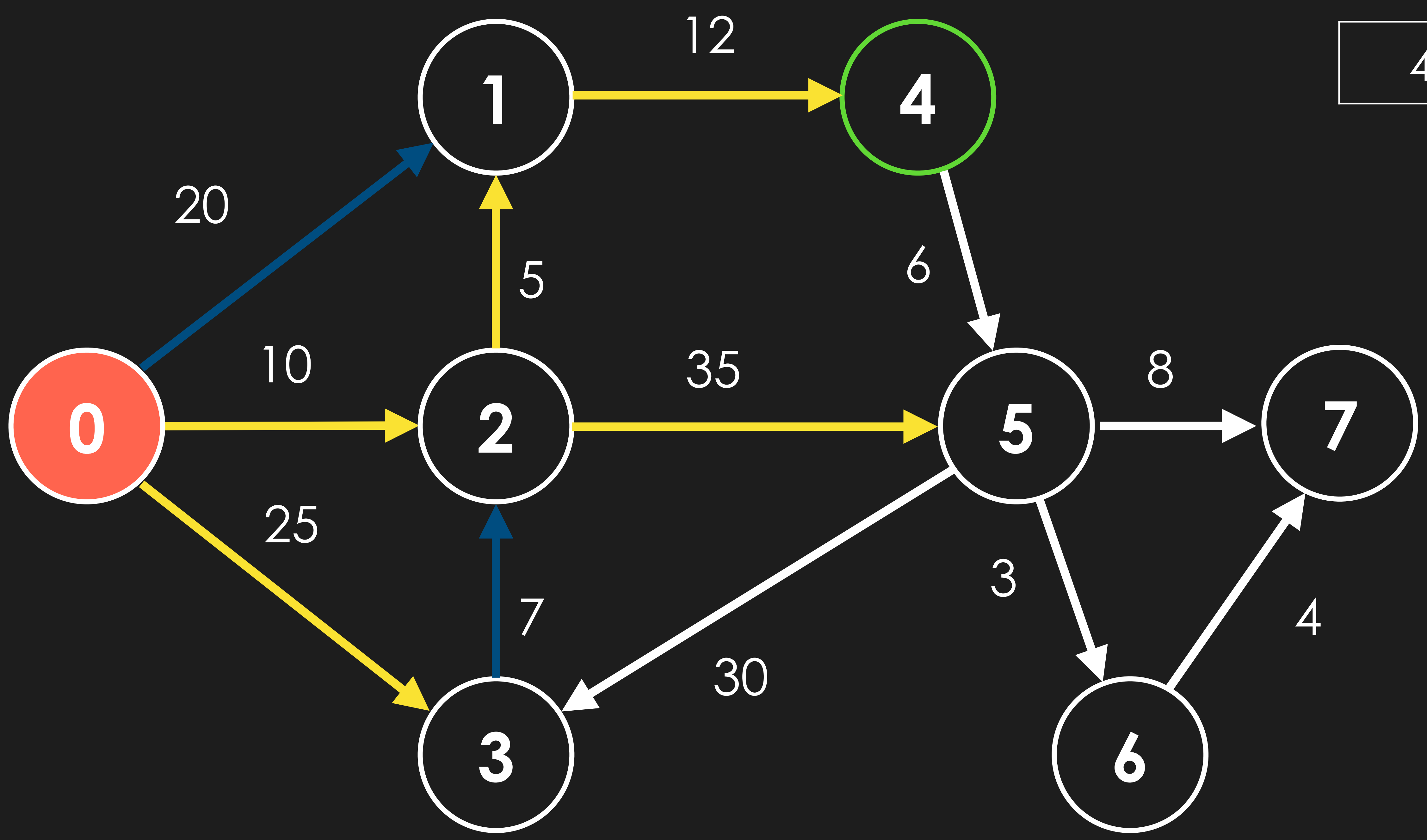
0	0
1	15
2	10
3	25
4	27
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



4	27

pq

vertex	distTo
5	45

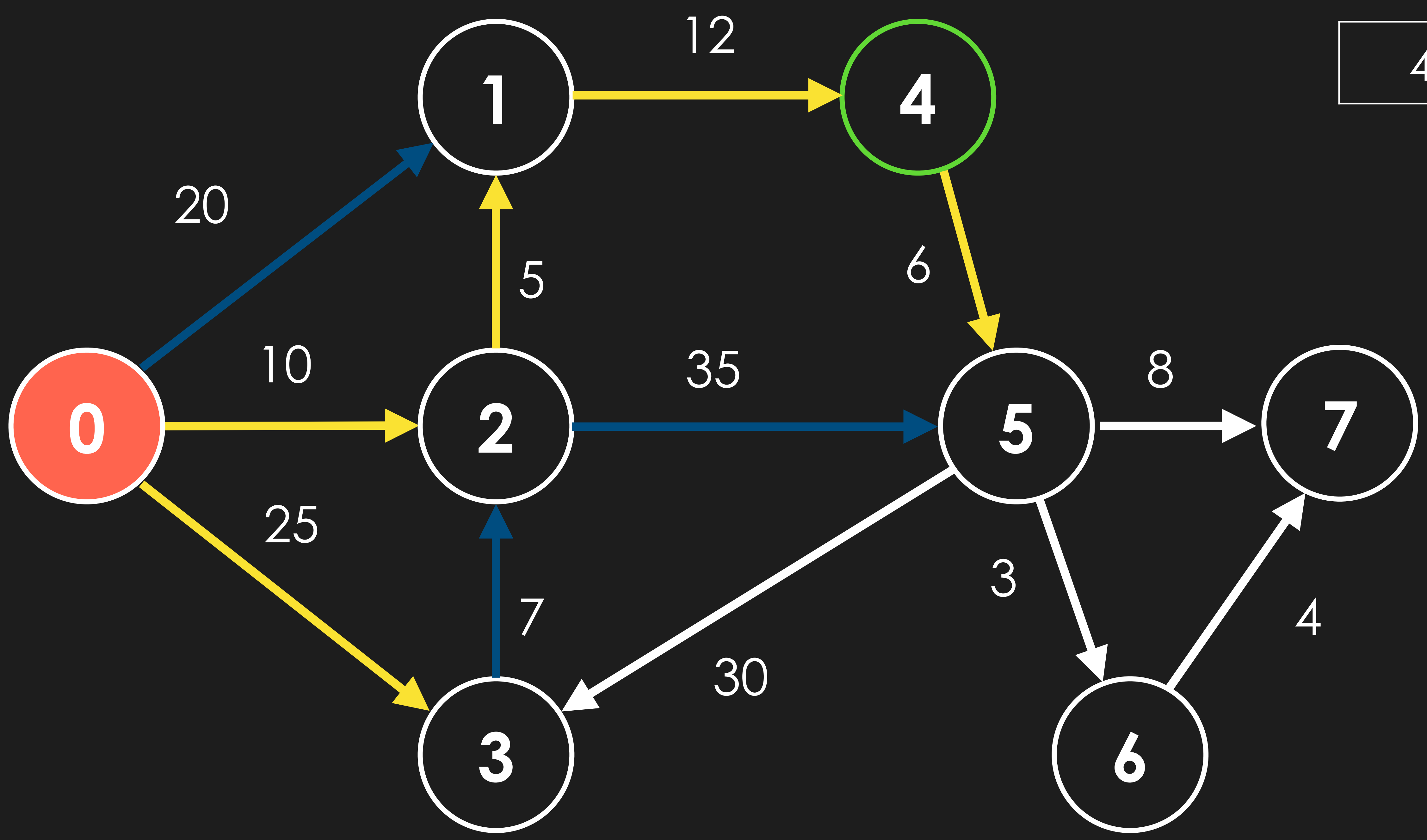
distTo

0	0
1	15
2	10
3	25
4	27
5	45
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



4	27

pq

vertex	distTo
5	33

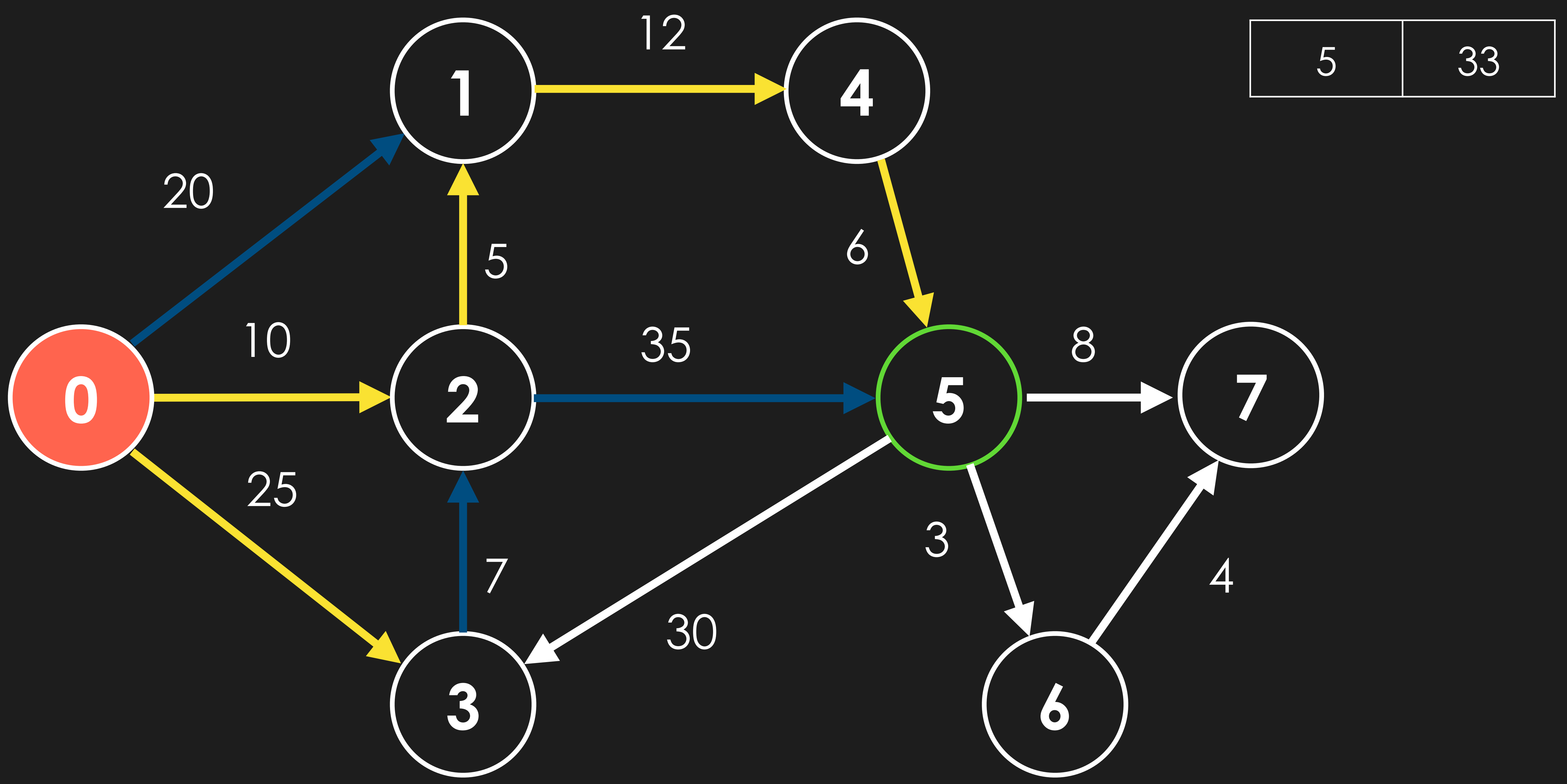
distTo

0	0
1	15
2	10
3	25
4	27
5	33
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



pq

vertex	distTo

distTo

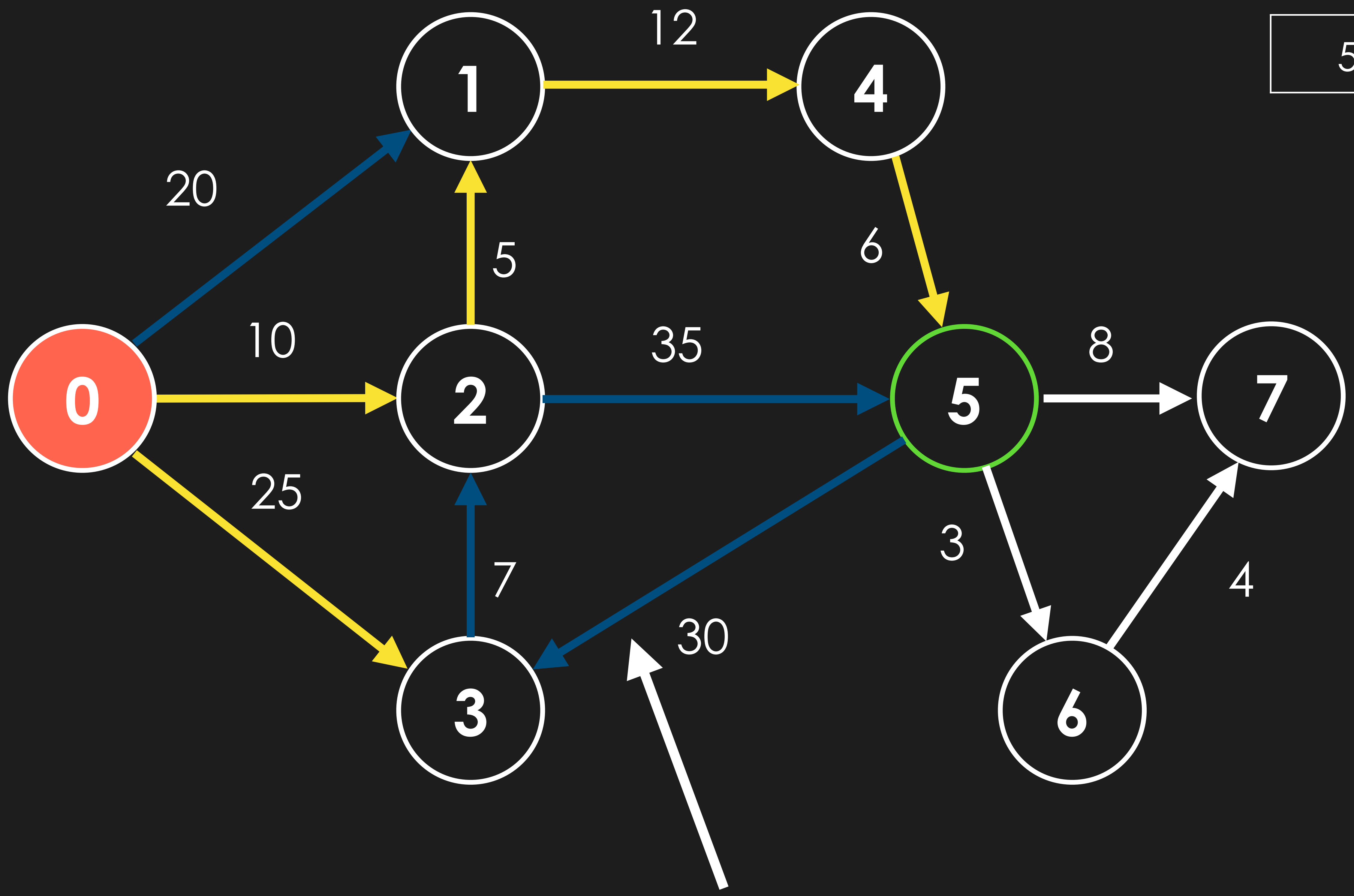
0	0
1	15
2	10
3	25
4	27
5	33
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	-1
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



5	33

distTo[5] + weight = **33 + 30 >** distTo[3] = **25**

pq

vertex	distTo

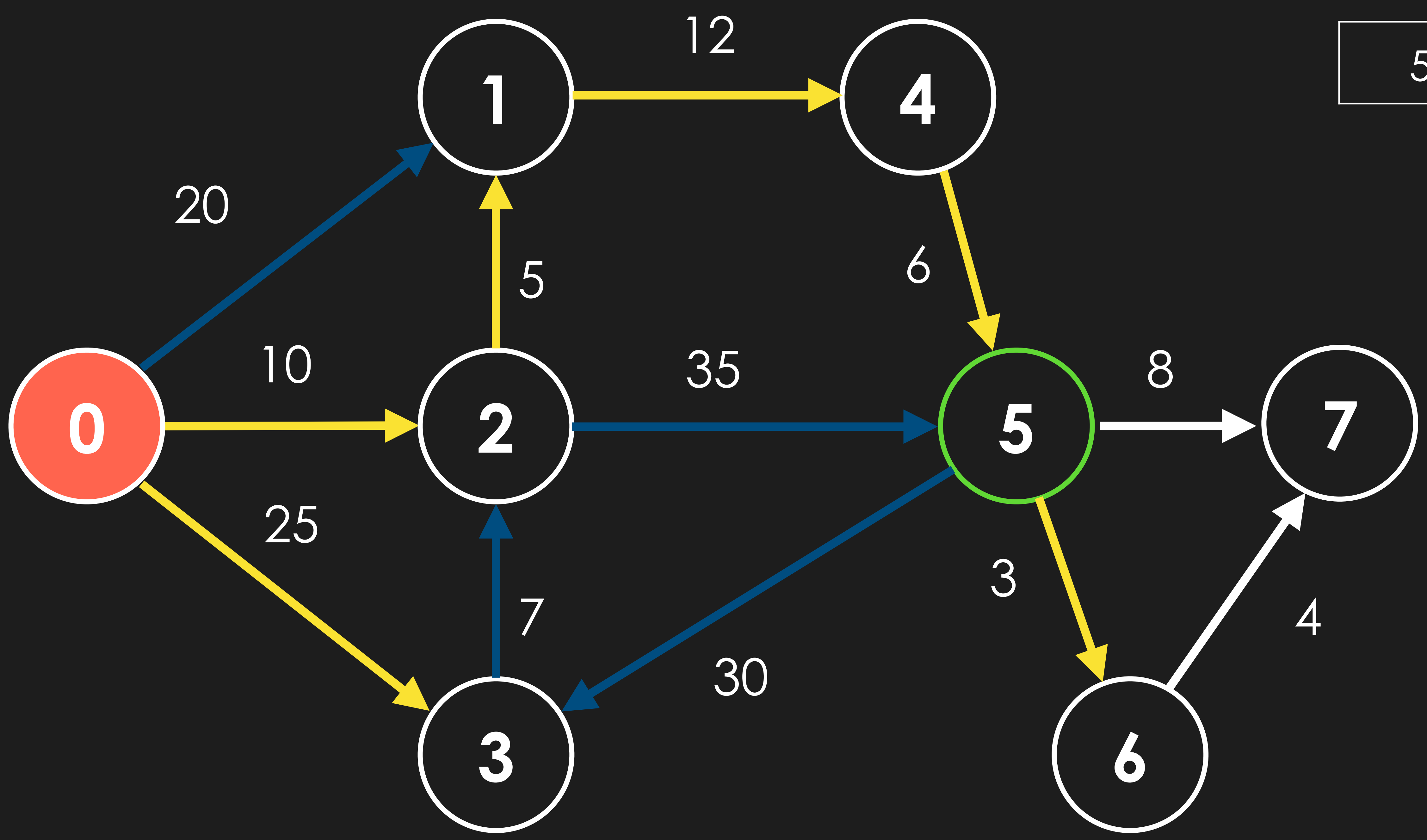
distTo

0	0
1	15
2	10
3	25
4	27
5	33
6	INF
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	-1
7	-1

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



5	33

pq

vertex	distTo
6	36

distTo

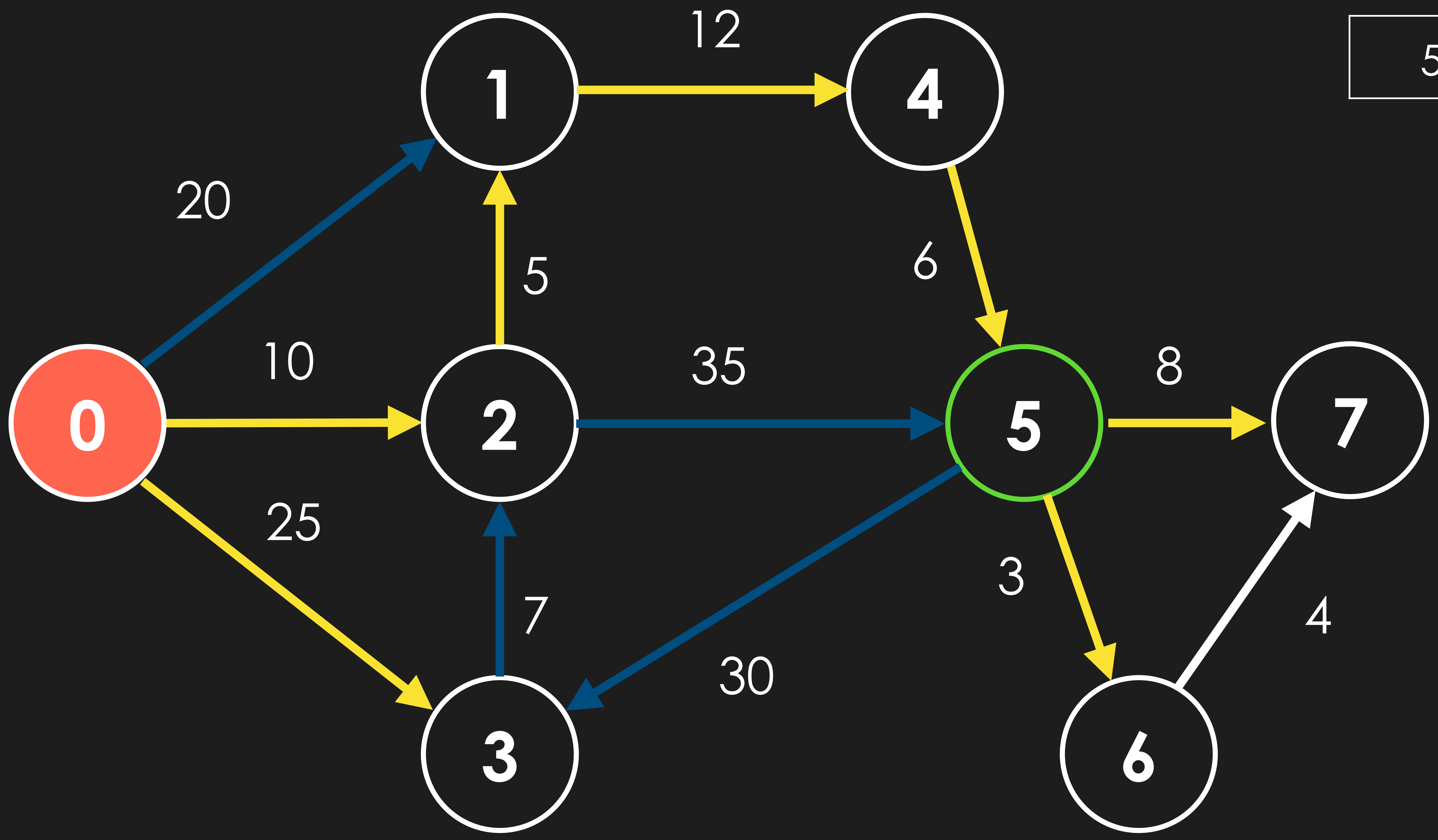
0	0
1	15
2	10
3	25
4	27
5	33
6	**36**
7	INF

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	**5 - 6**
7	-1

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



5	33

pq

vertex	distTo
6	**36**
7	**41**

distTo

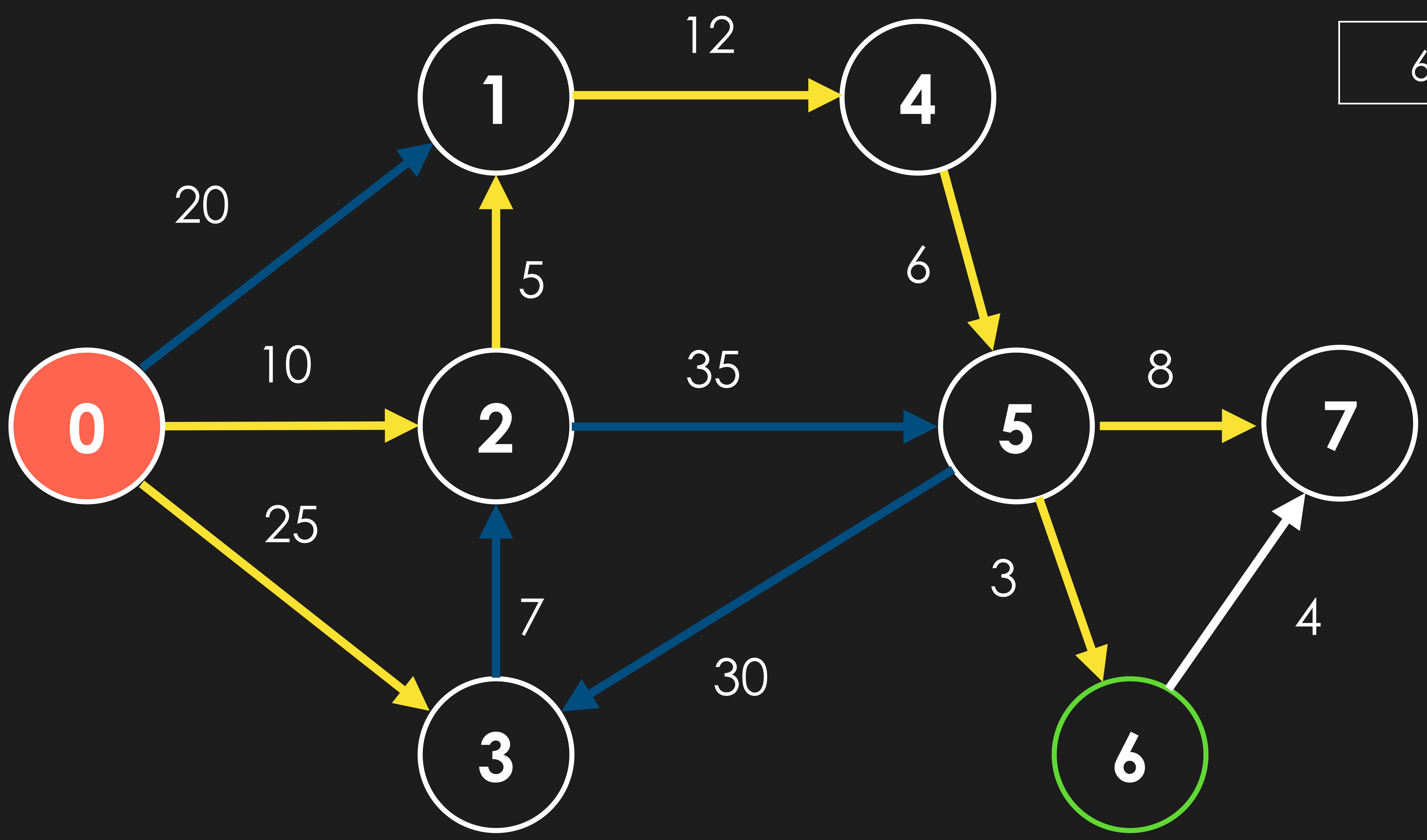
0	0
1	15
2	10
3	25
4	27
5	33
6	**36**
7	**41**

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	**5 - 6**
7	**5 - 7**

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



6	36

pq

vertex	distTo
7	41

distTo

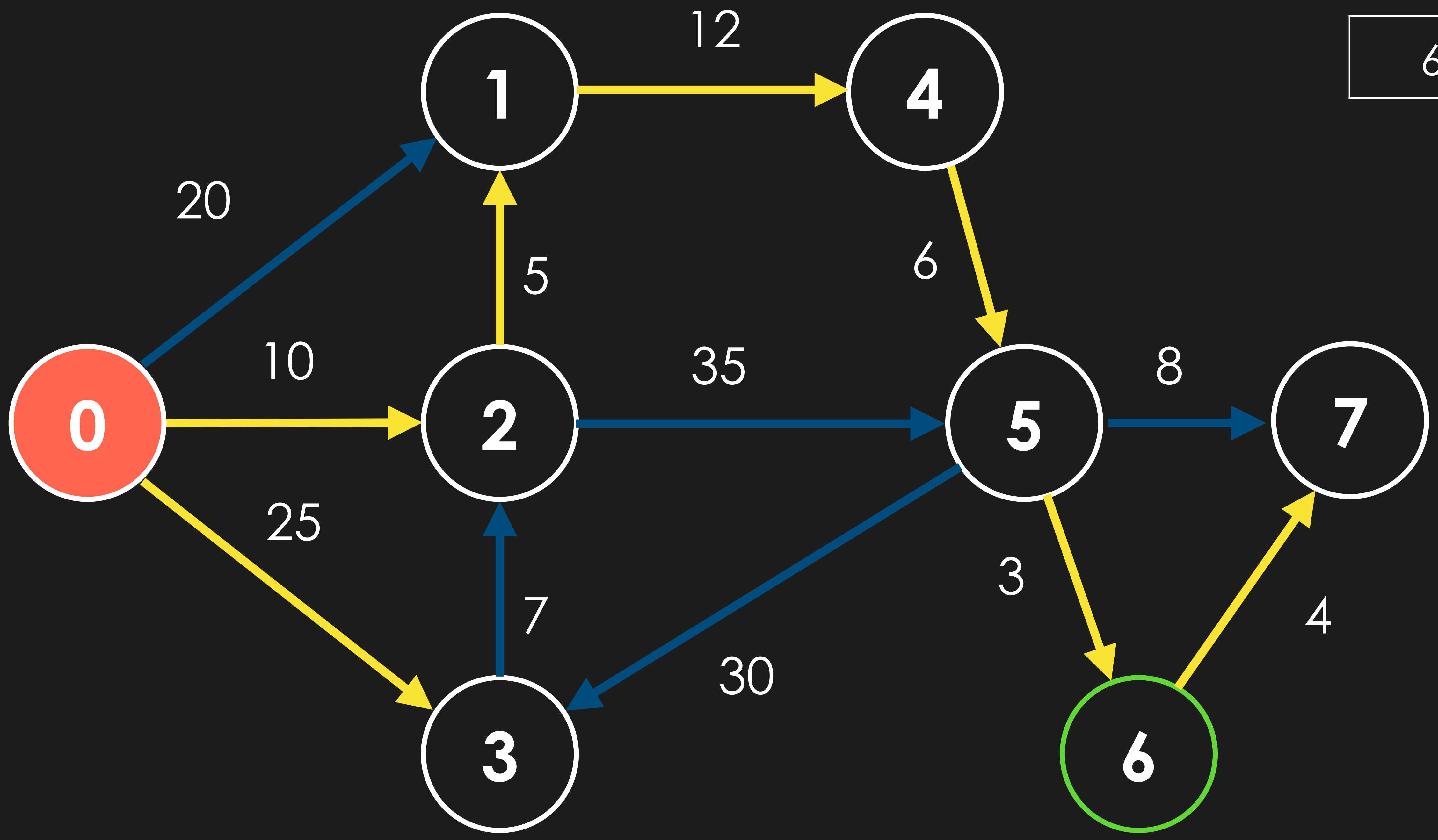
0	0
1	15
2	10
3	25
4	27
5	33
6	36
7	41

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

CodeIT

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



6	36

pq

vertex	distTo
7	40

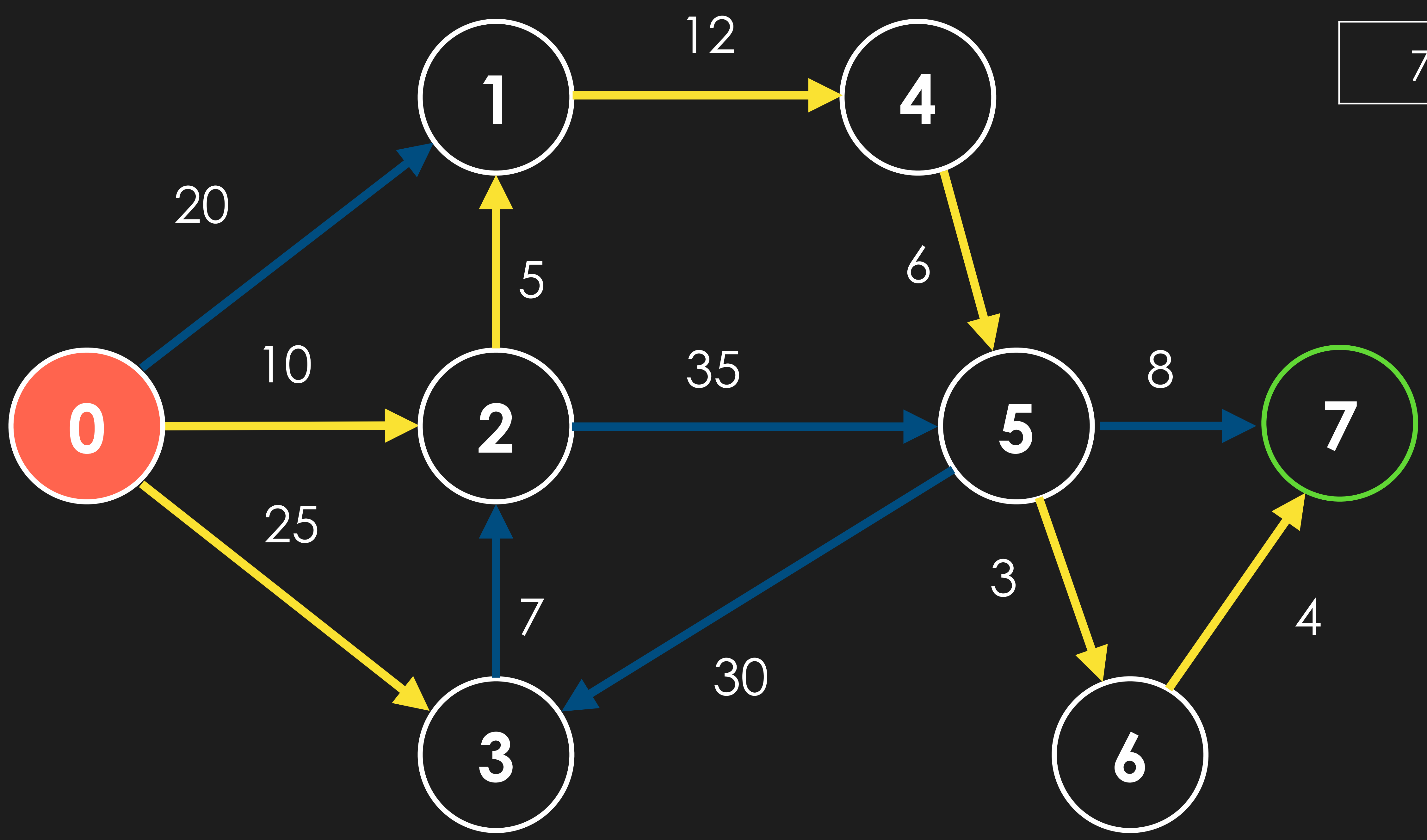
distTo

0	0
1	15
2	10
3	25
4	27
5	33
6	36
7	40

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	6 - 7

4. While **pq is not empty:**
   - extract min vertex from pq
   - **relax adjacent edges**



7	40

pq

vertex	distTo

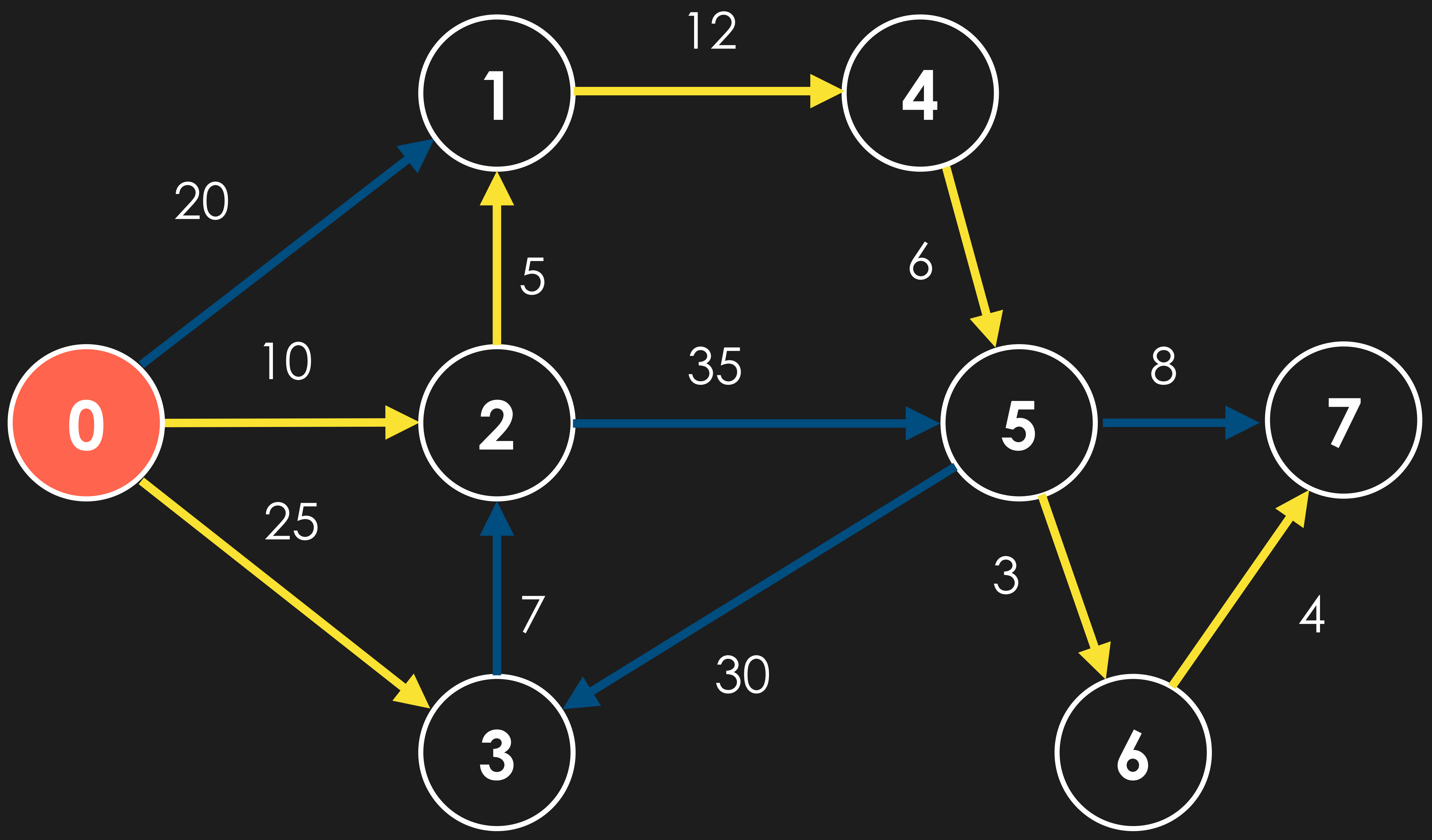
distTo

0	0
1	15
2	10
3	25
4	27
5	33
6	36
7	40

edgeTo

0	-1
1	2 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	6 - 7

Now we have our SPT!



	distTo		edgeTo
0	0	0	-1
1	15	1	2 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	27	4	1 - 4
5	33	5	4 - 5
6	36	6	5 - 6
7	40	7	6 - 7

Implementation of Dijkstra's Algorithm

relaxEdge

```python
def relaxEdge(edge, pq, distTo, edgeTo):
    v = edge.src
    w = edge.dest

    if distTo[v] + edge.weight < distTo[w]:
        distTo[w] = distTo[v] + edge.weight
        edgeTo[w] = edge
        if w in pq.positions:
            pq.decreaseKey(w, distTo[w])
        else:
            pq.insert(w, distTo[w])
```

Dijkstra

```
def Dijkstra(graph, s):
    INF = 99999

    V = len(graph.adjList)
    edgeTo = [None] * V
    distTo = [INF] * V
    pq = MinHeap(V)

    distTo[s] = 0
    pq.insert(s, distTo[s])

    while (pq.size != 0):
        v = pq.getMin().key

        for edge in graph.adjList[v]:
            relaxEdge(edge, pq, distTo, edgeTo)

    return edgeTo, distTo
```

constructShortestPath

```python
def constructShortestPath(edgeTo, v):

    cur = edgeTo[v]
    path = [cur.dest]
    while (cur != -1):
        path.append(cur.src)
        cur = edgeTo[cur.src]

    path.reverse()
    return path
```

Analysis of Dijkstra's Algorithm

In Dijkstra's algorithm, we perform the **extract operation V times** and the **insert operation E times**

Assuming we use an pq with **decreaseKey** operation, then our pq will have at most **V items**, if not it will have **E items.** Thus, our **insert / extract operations** will run in **logV** time

Overall, our time complexity is **O(ElogV)**

Applications of Shortest Path Problem

GPS Systems

Network Betweenness Centrality

Seam Carving (Image Manipulation)

Limitations of Dijkstra's Algorithm

Limitations of Dijkstra's Algorithm

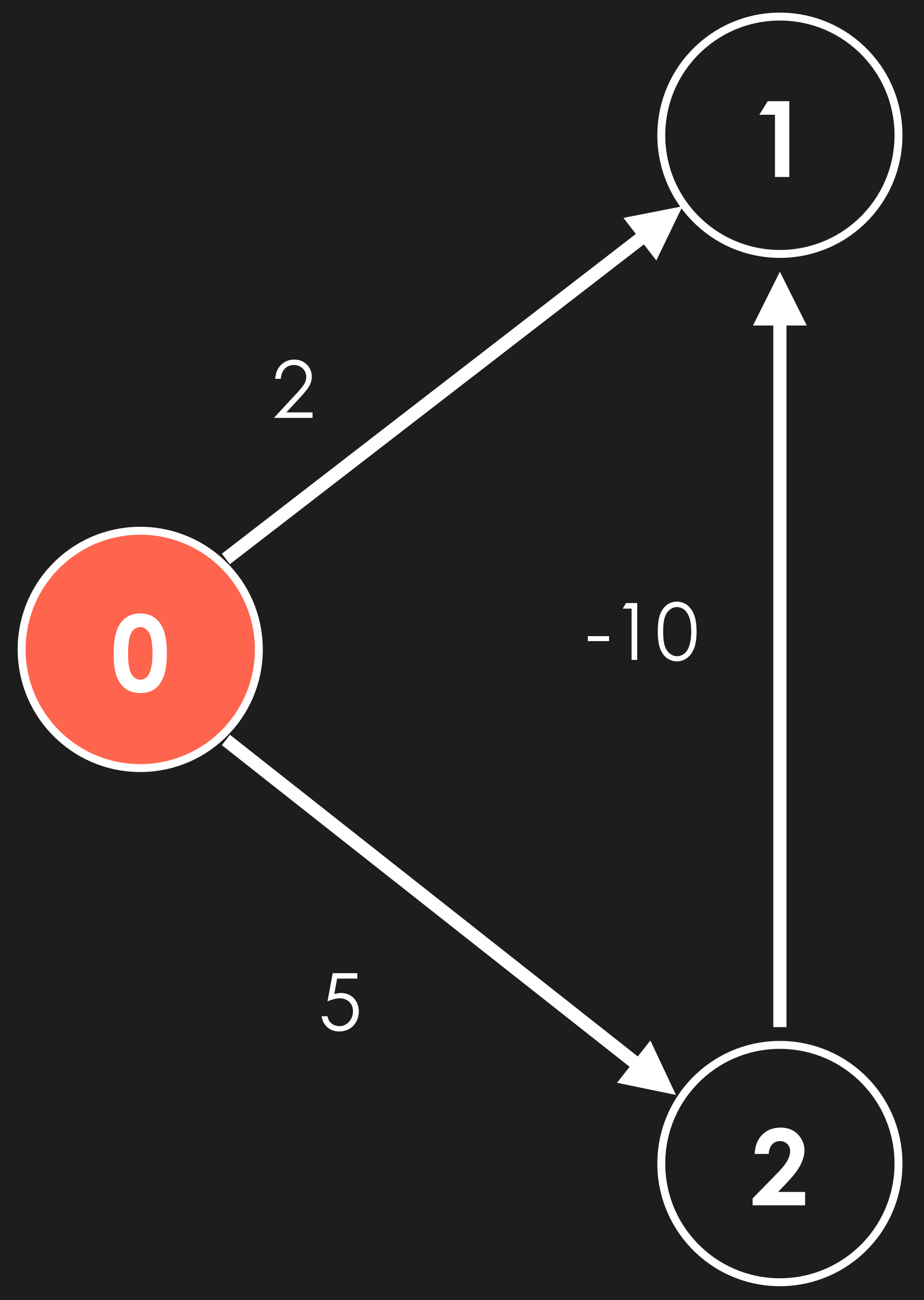
1. Graph with negative weight edges
2. Graph with negative cycles

Graphs with negative weight edges

- Dijkstra's algorithm is a greedy algorithm. The assumption is that **upon relaxation of each edge**, the shortest distance to the dest vertex has been found.

- Graphs with negative weight edges will interfere with this. In some cases, Dijkstra's algorithm will **terminate with the wrong answer**, while in others, the time complexity is no longer guaranteed to be ElogV. Our implementation is affected by the latter.

Graphs with negative weight edges



pq

vertex	distTo
0	0

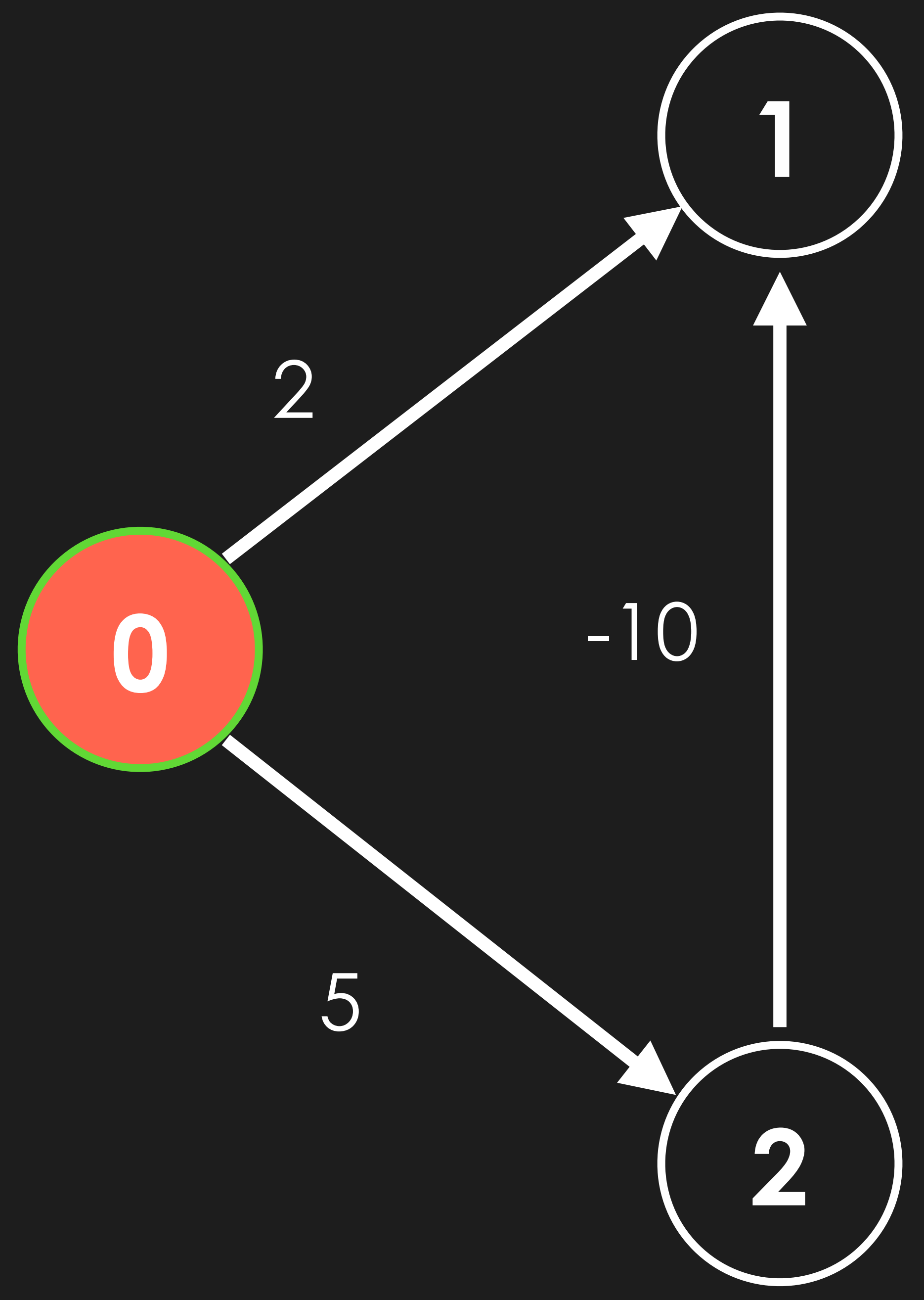
distTo

0	0
1	INF
2	INF

edgeTo

0	-1
1	-1
2	-1

Graphs with negative weight edges



0	0

pq

vertex	distTo

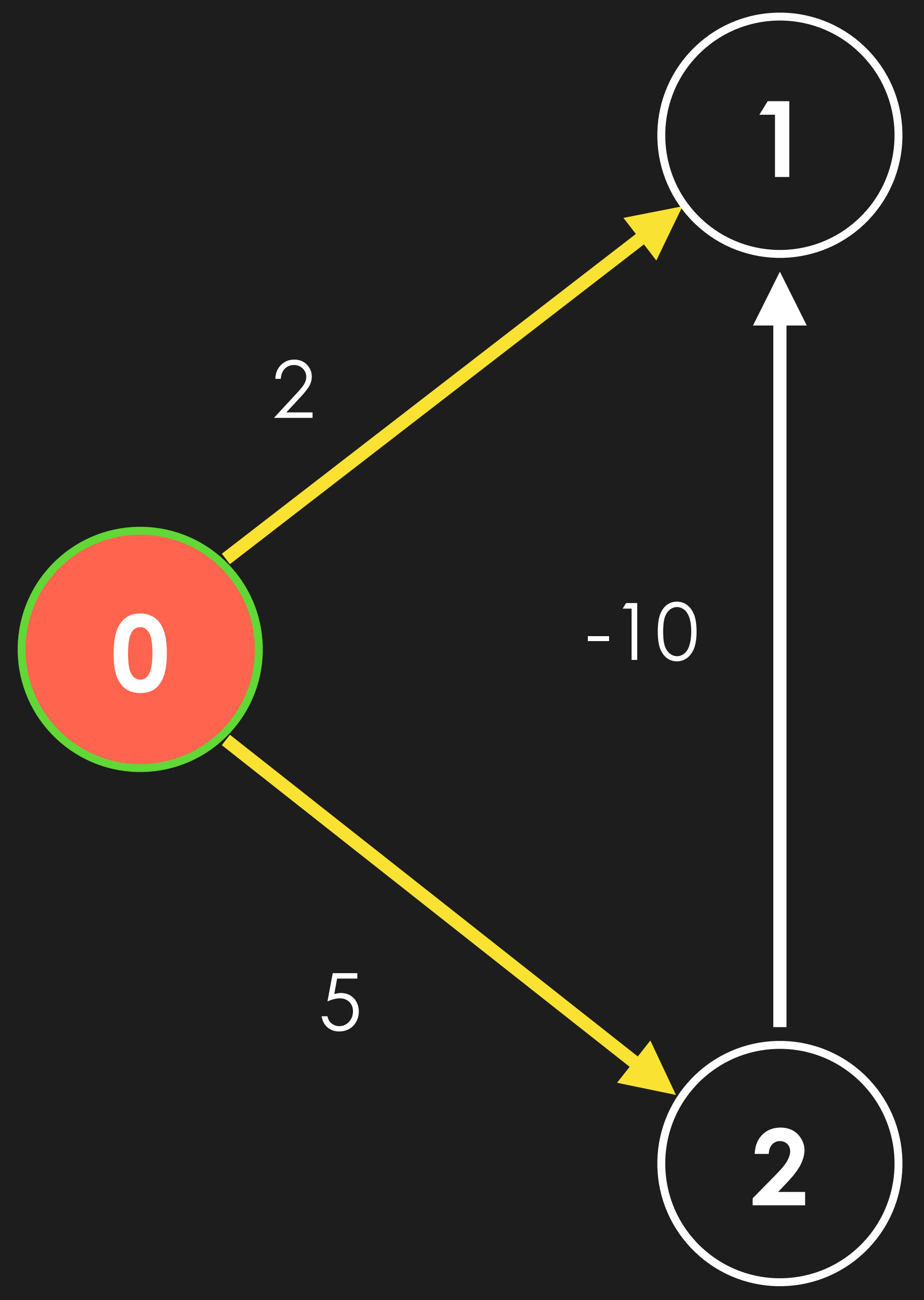
distTo

0	0
1	INF
2	INF

edgeTo

0	-1
1	-1
2	-1

Graphs with negative weight edges



0	0

pq

vertex	distTo
1	2
2	5

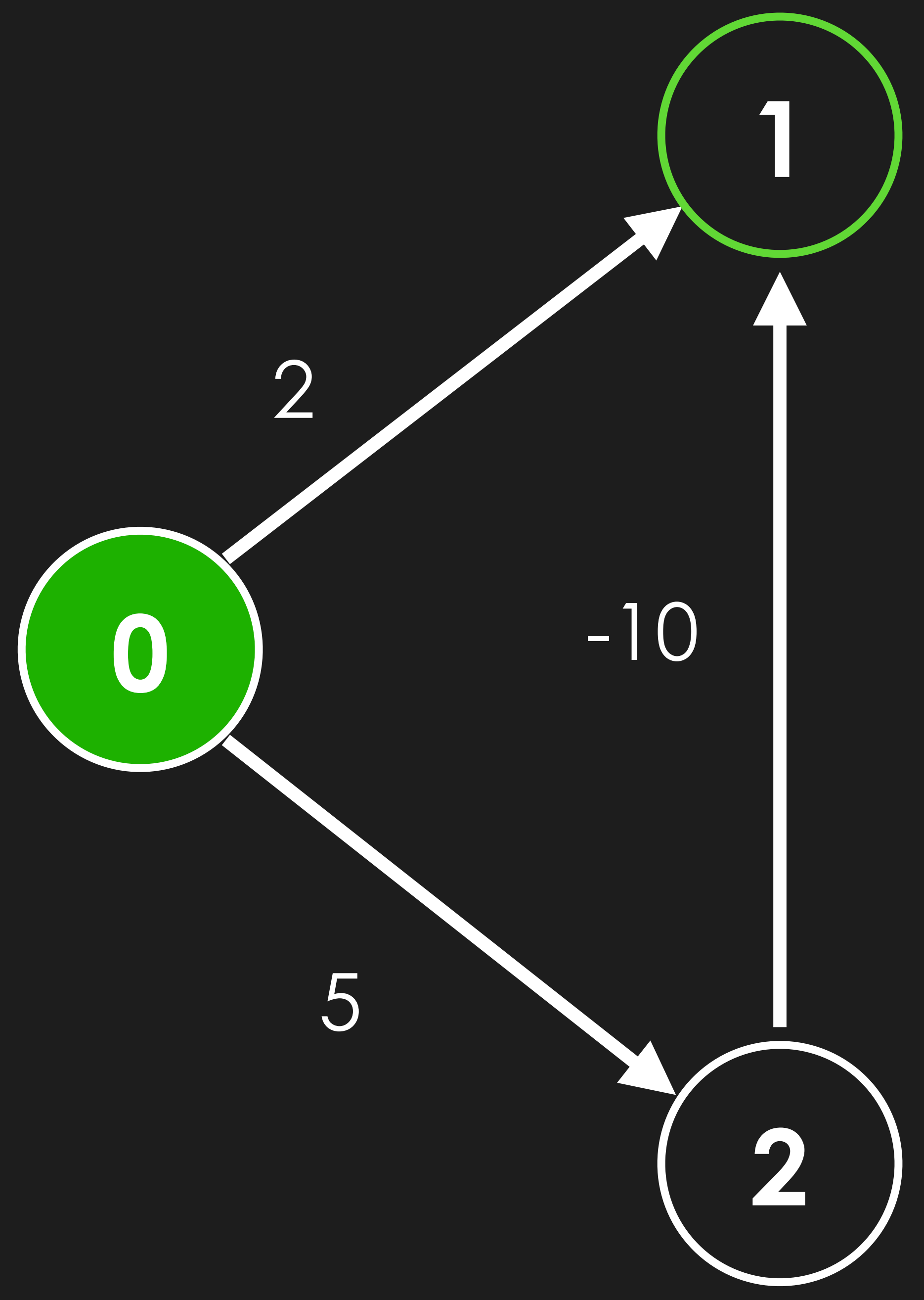
distTo

0	0
1	2
2	5

edgeTo

0	-1
1	0
2	0

Graphs with negative weight edges



1	2

pq

vertex	distTo
2	5

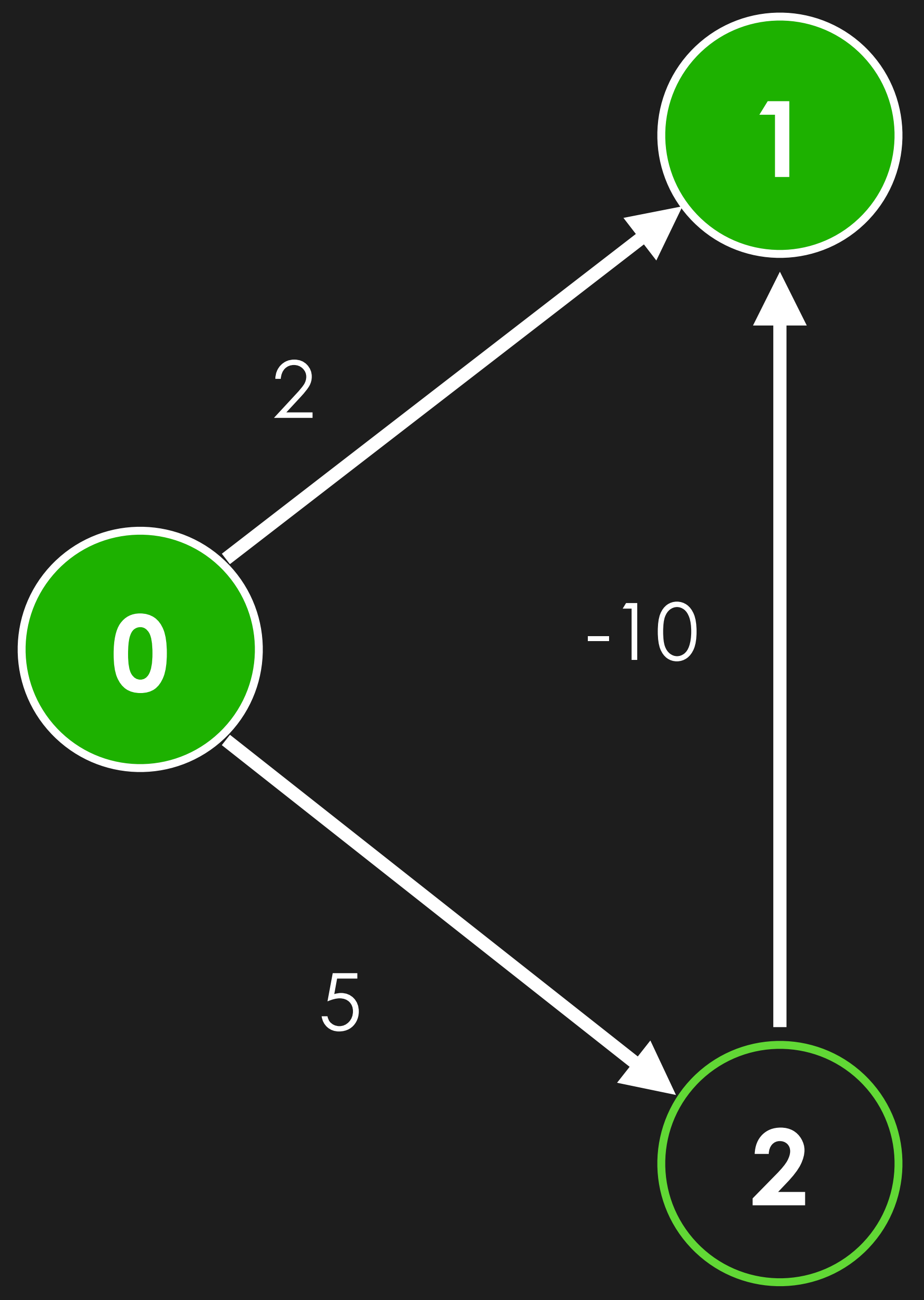
distTo

0	0
1	2
2	5

edgeTo

0	-1
1	0
2	0

Graphs with negative weight edges



2	5

pq

vertex	distTo

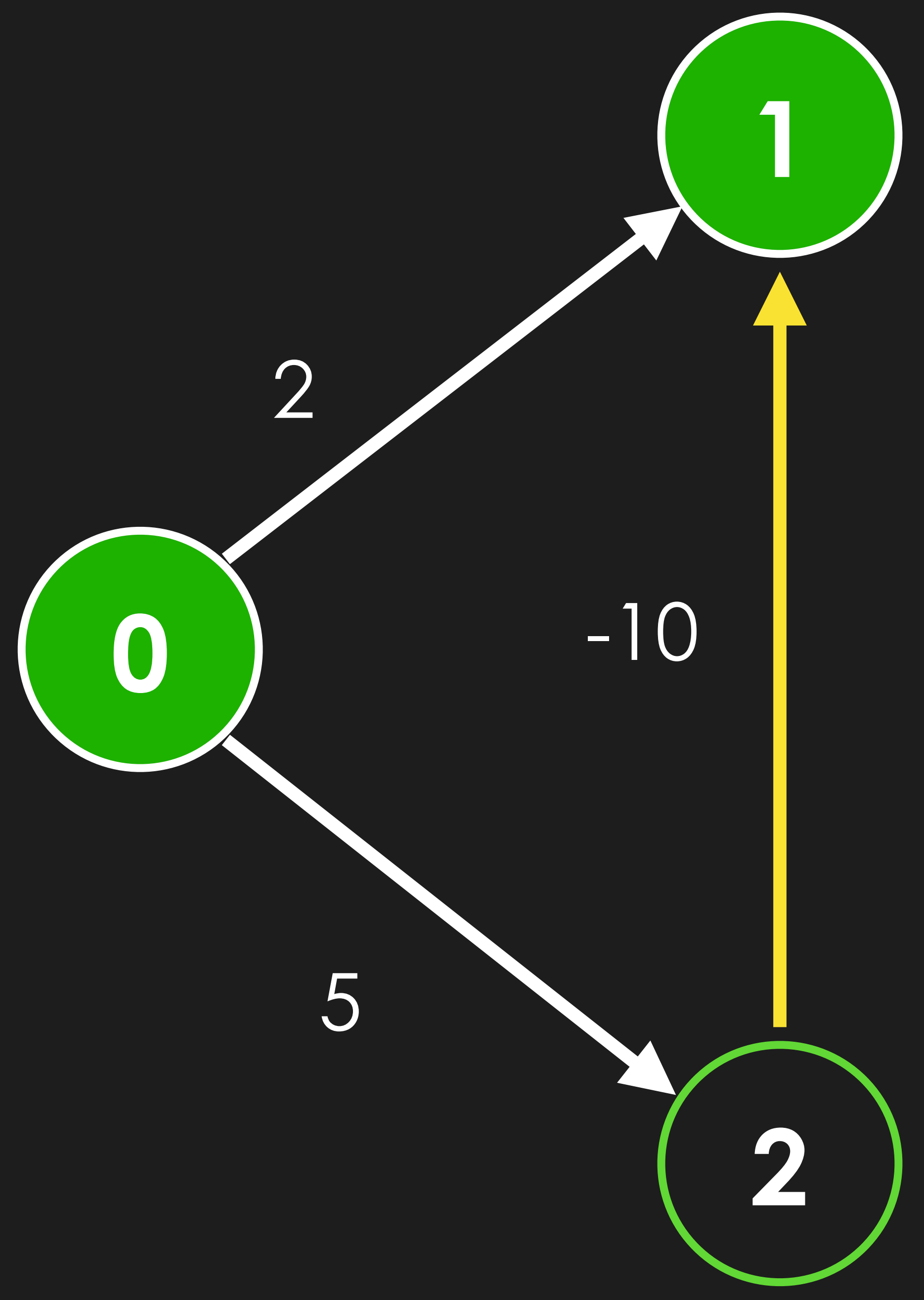
distTo

0	0
1	2
2	5

edgeTo

0	-1
1	0
2	0

Graphs with negative weight edges



2	5

pq

vertex	distTo
1	-5

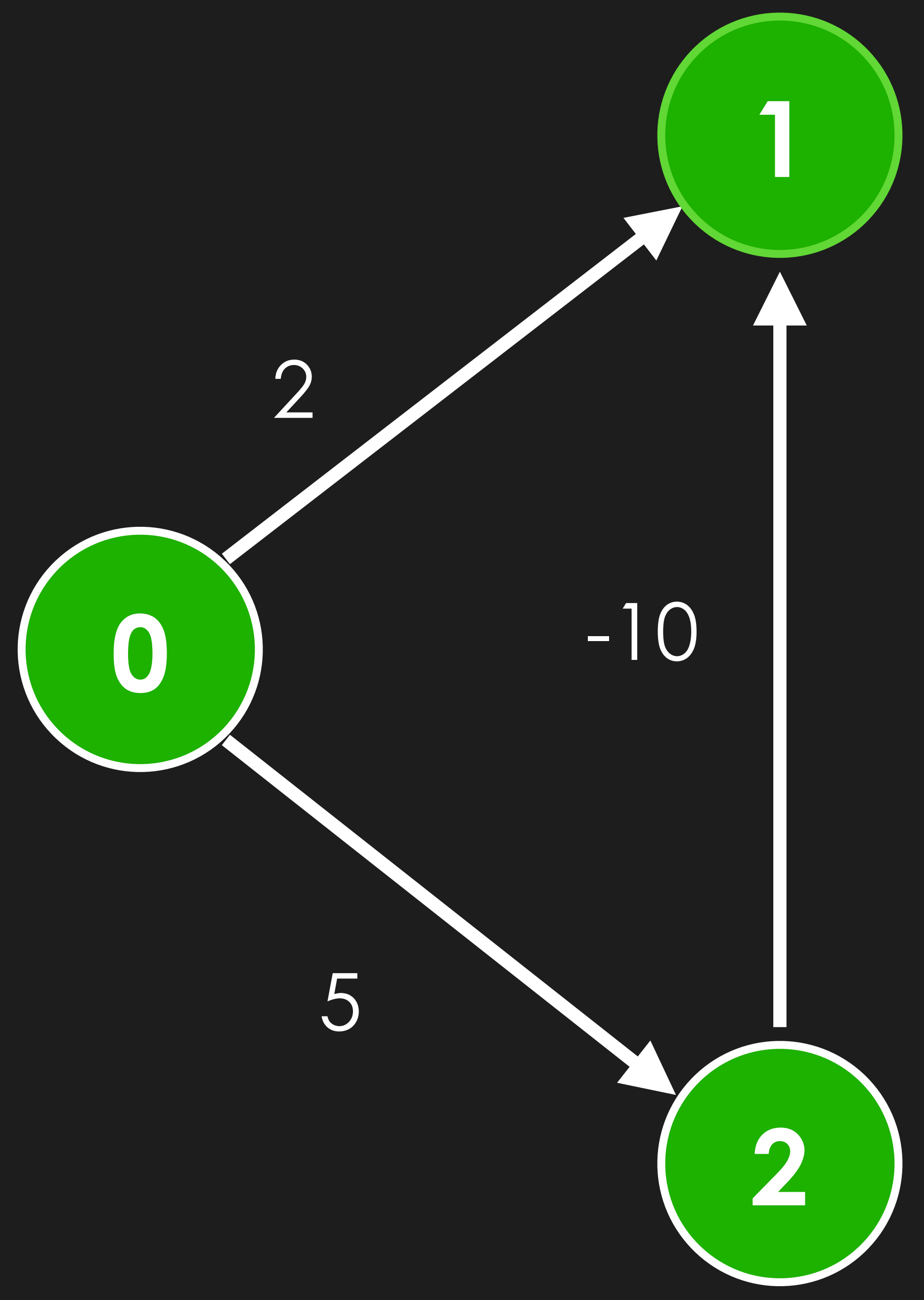
distTo

0	0
1	**-5**
2	5

edgeTo

0	-1
1	**2**
2	0

Graphs with negative weight edges



1	-5

pq

vertex	distTo

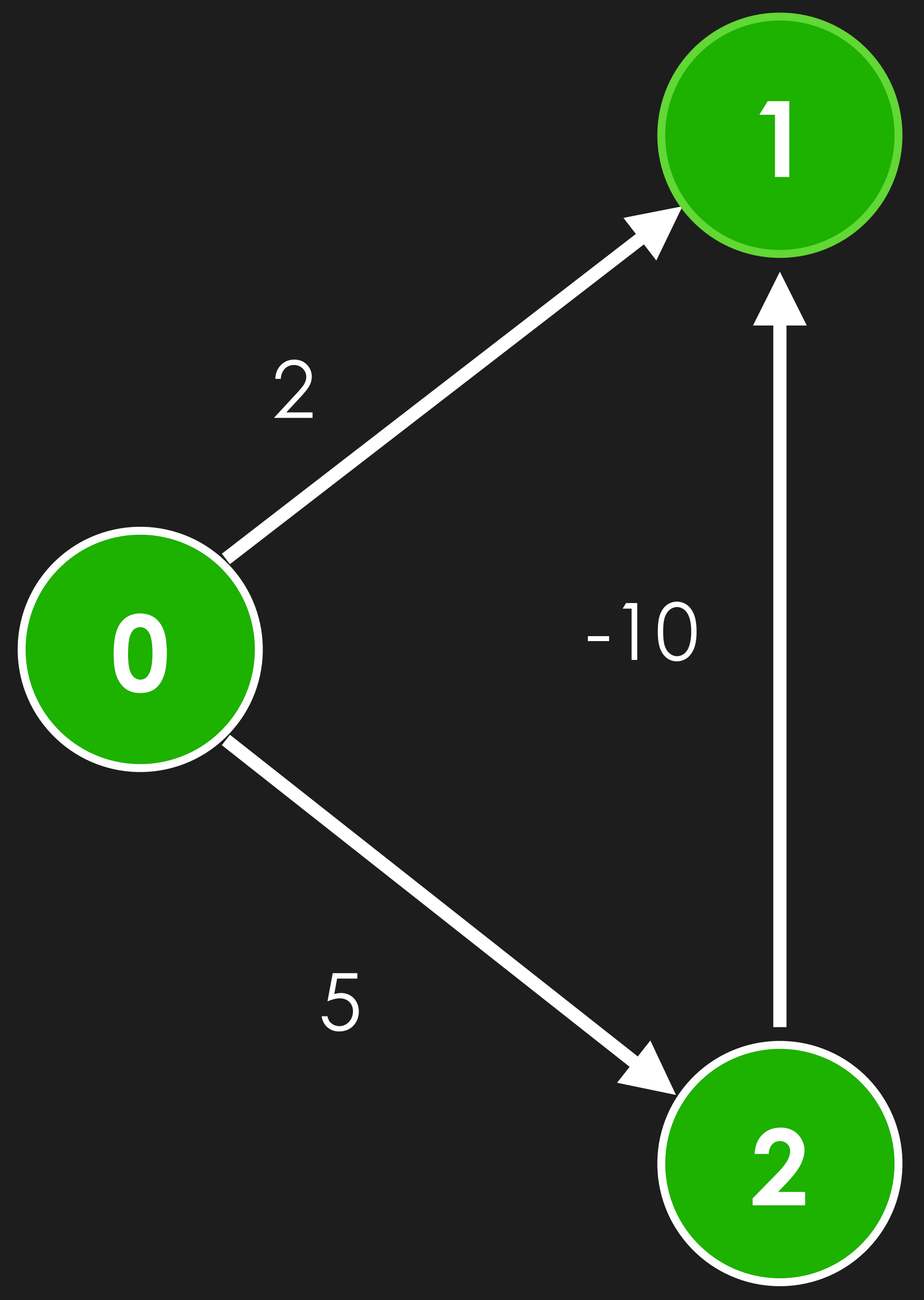
distTo

0	0
1	-5
2	5

edgeTo

0	-1
1	2
2	0

Graphs with negative weight edges



1	-5

pq

vertex	distTo

distTo

0	0
1	-5
2	5

edgeTo

0	-1
1	2
2	0

This is the second time we are relaxing vertex 1's edges. As such, our time complexity is no longer **ElogV**!

Solution: Check for negative edges before performing Dijkstra

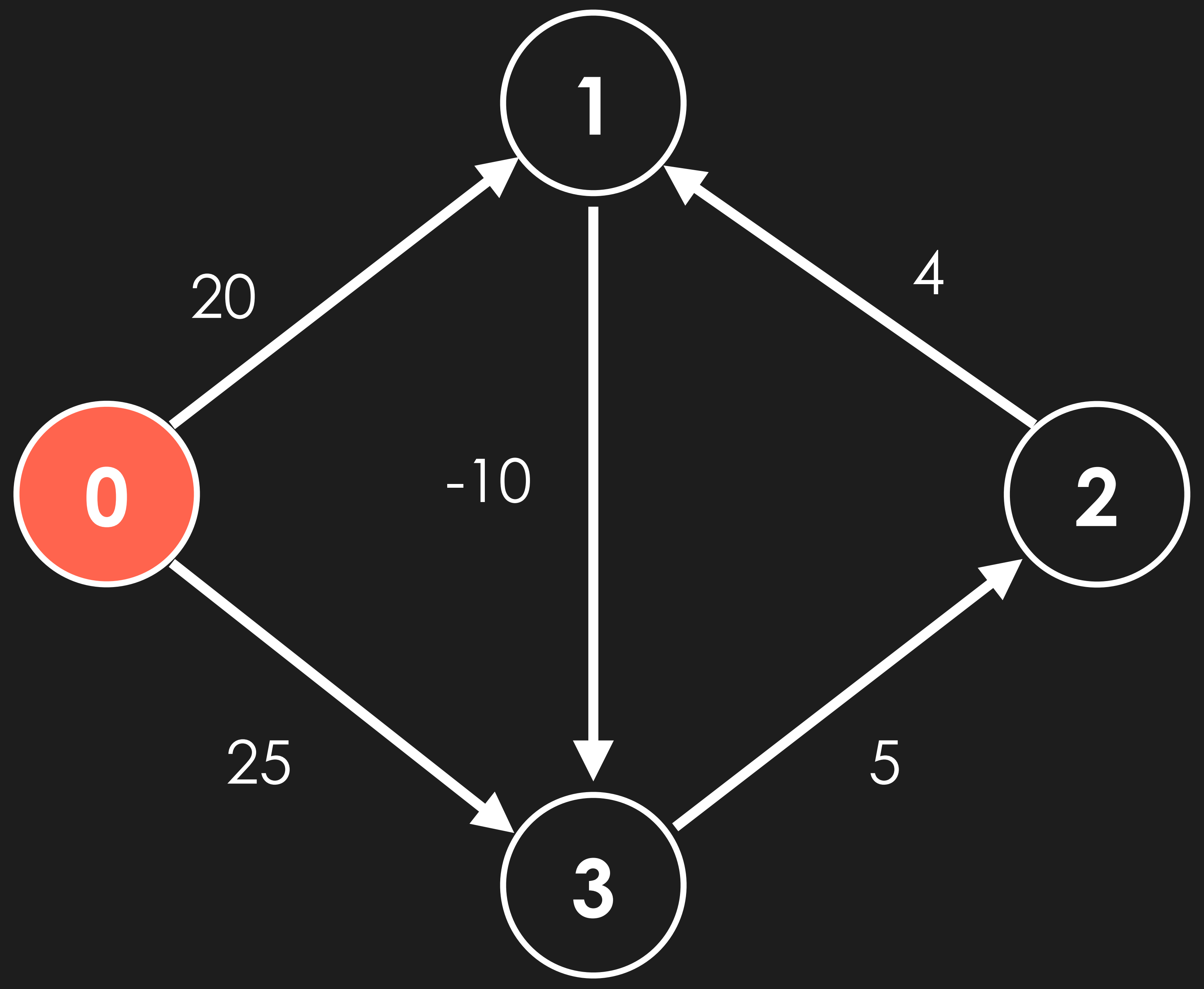
Solution: Check for negative edges before performing Dijkstra

```
def Dijkstra(graph, s):
    V = len(graph.adjList)

    for v in range(len(graph.adjList)):
        for edge in graph.adjList[v]:
            if edge.weight < 0:
                print("Negative weight edge detected")
                return

    edgeTo = [-1] * V
    distTo = [None] * V
    pq = MinHeap(V)

    distTo[s] = 0
    pq.insert(s, distTo[s])

    while (pq.size != 0):
        v = pq.getMin().key

        for edge in graph.adjList[v]:
            relax(edge, pq, distTo, edgeTo)

    return edgeTo, distTo
```

CodeIT

Graphs with negative cycles

Consider the following graph



pq

vertex	distTo
0	0

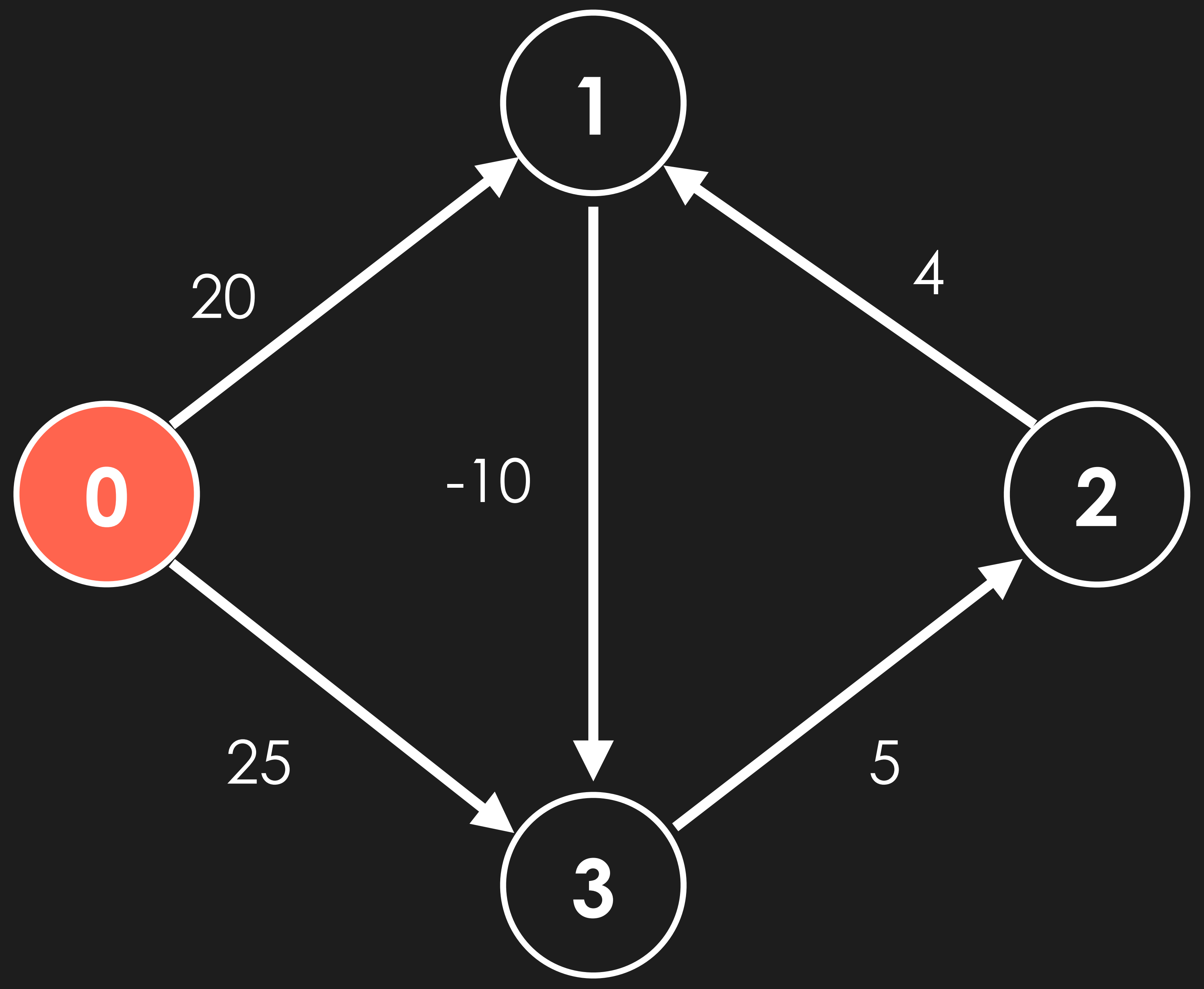
distTo

0	0
1	INF
2	INF
3	INF

edgeTo

0	-1
1	-1
2	-1
3	-1

Consider the following graph



pq

vertex	distTo
0	0

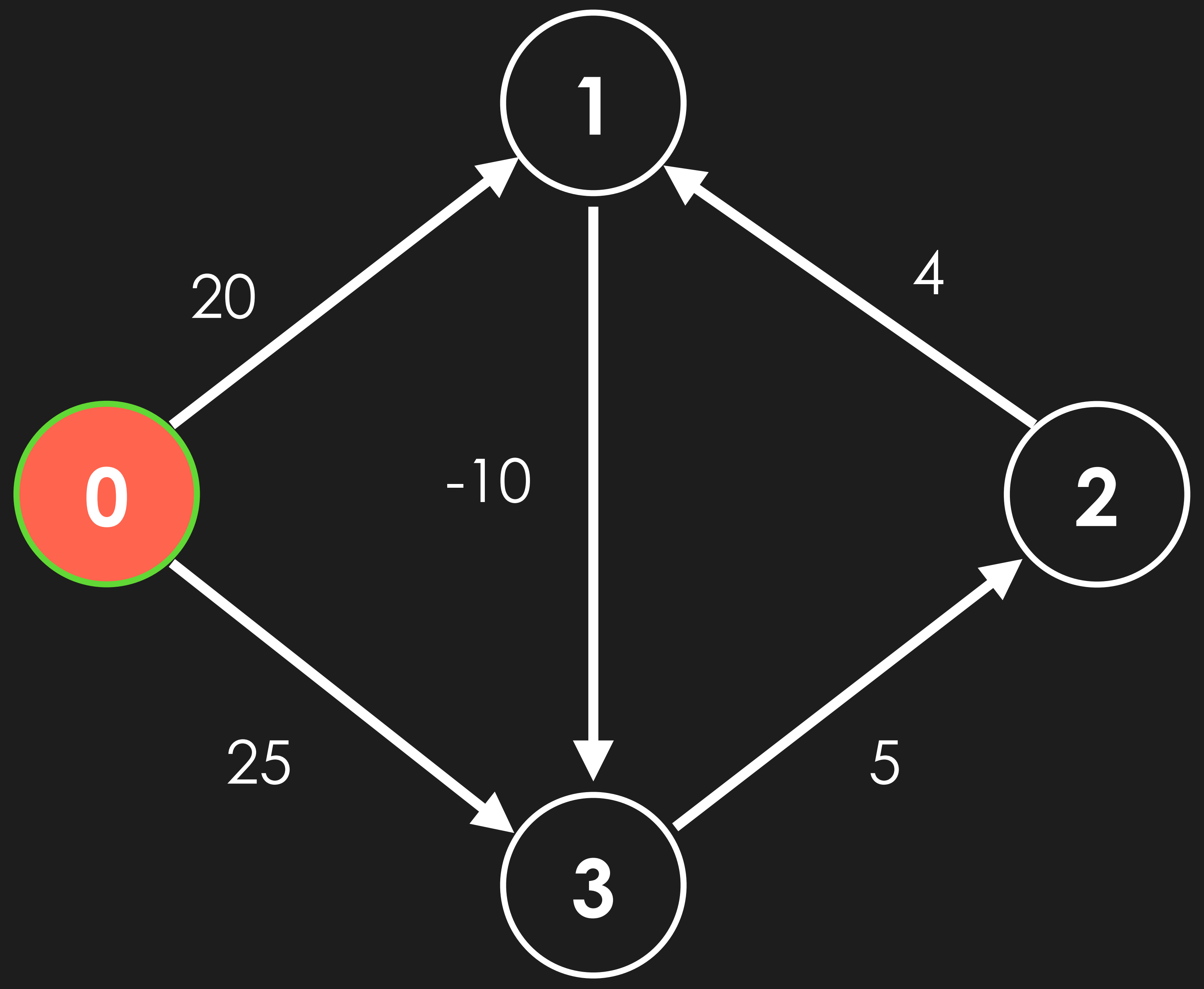
distTo

0	0
1	INF
2	INF
3	INF

edgeTo

0	-1
1	-1
2	-1
3	-1

Consider the following graph



0	0

pq

vertex	distTo

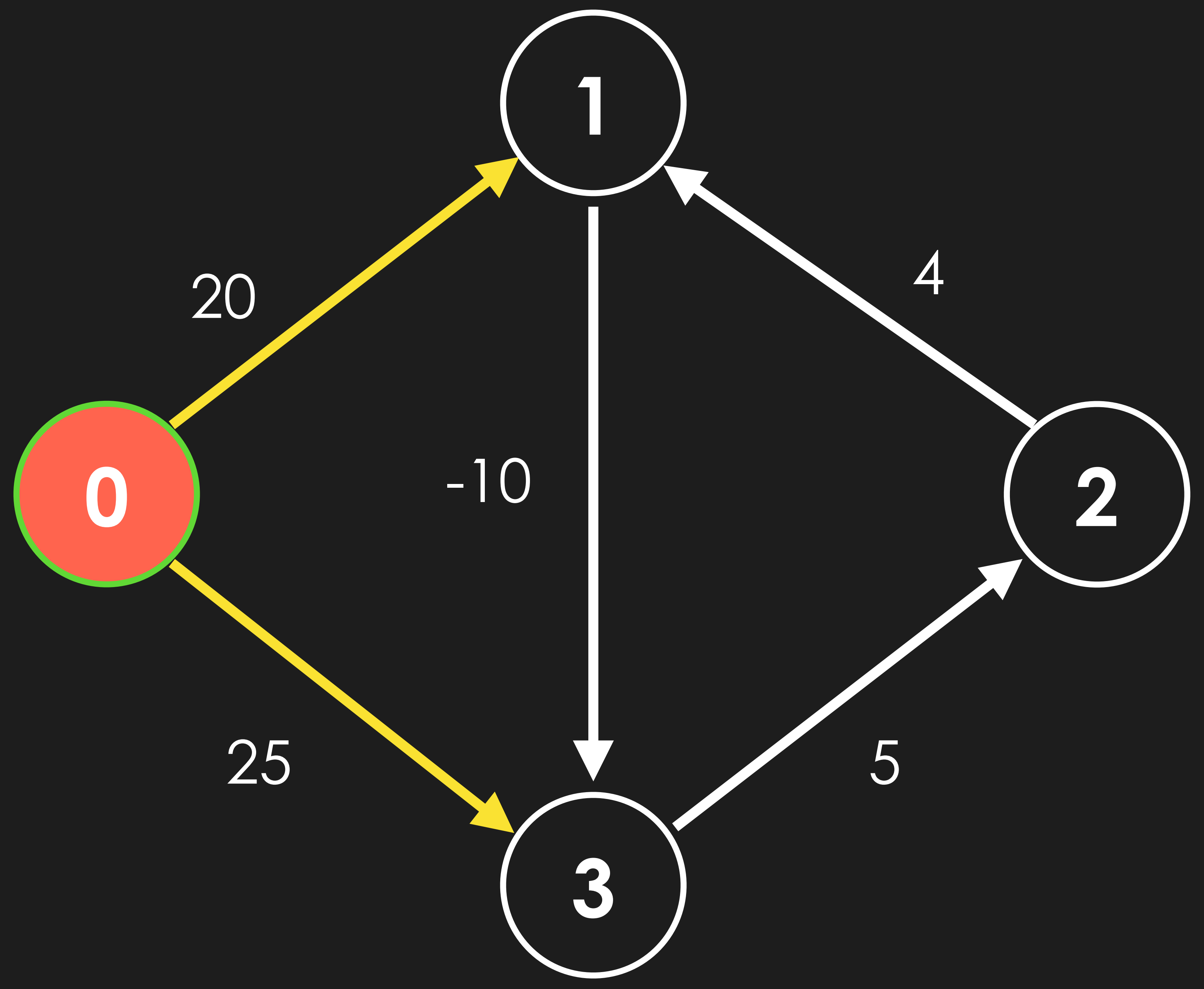
distTo

0	0
1	INF
2	INF
3	INF

edgeTo

0	-1
1	-1
2	-1
3	-1

Consider the following graph



0	0

pq

vertex	distTo
1	20
3	25

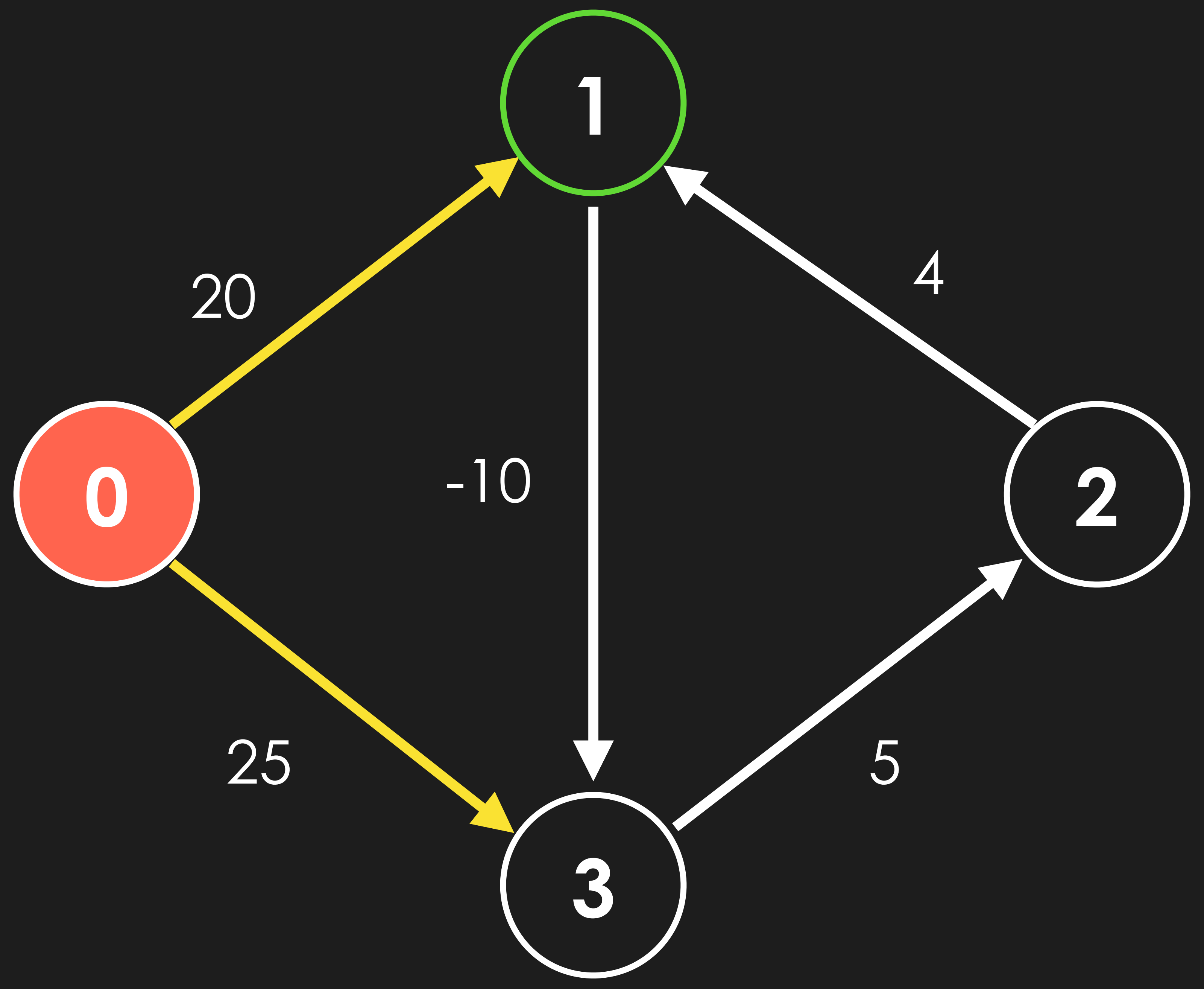
distTo

0	0
1	20
2	INF
3	25

edgeTo

0	-1
1	0 - 1
2	-1
3	0 - 3

Consider the following graph



1	20

pq

vertex	distTo
3	25

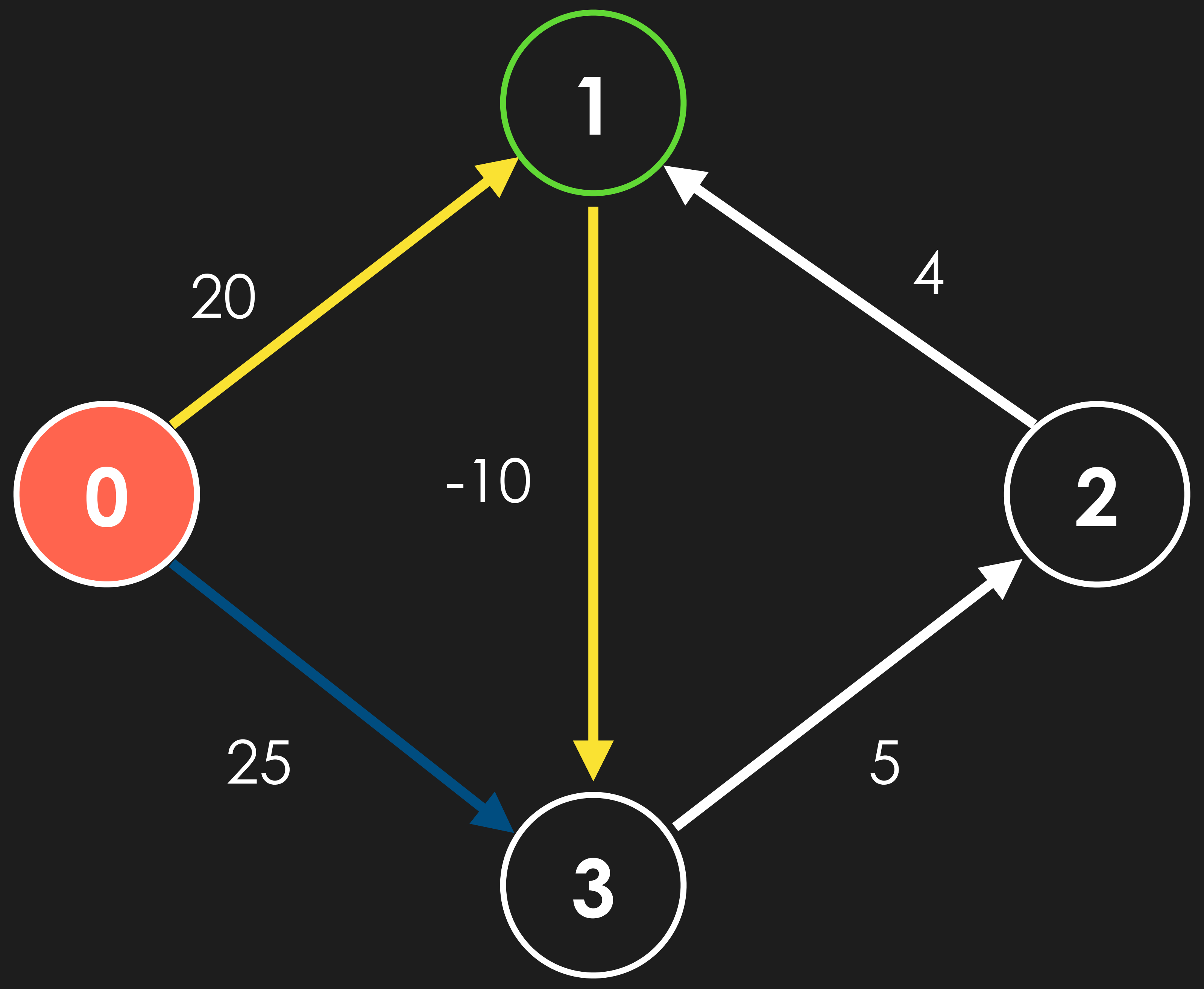
distTo

0	0
1	20
2	INF
3	25

edgeTo

0	-1
1	0 - 1
2	-1
3	0 - 3

Consider the following graph



1	20

pq

vertex	distTo
3	**10**

distTo

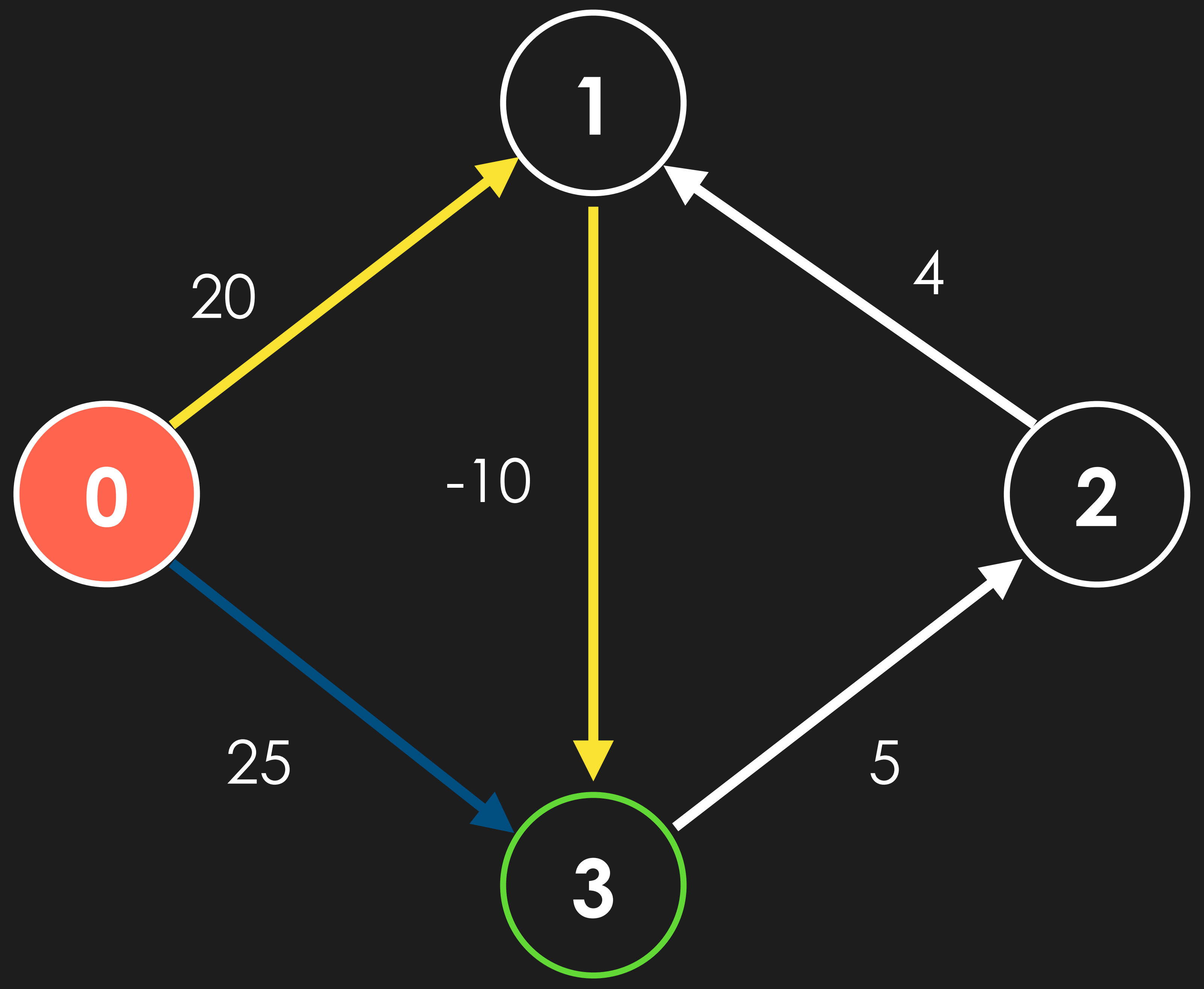
0	0
1	20
2	INF
3	**10**

edgeTo

0	-1
1	0 - 1
2	-1
3	**1 - 3**

CodeIT

Consider the following graph



3	10

pq

vertex	distTo

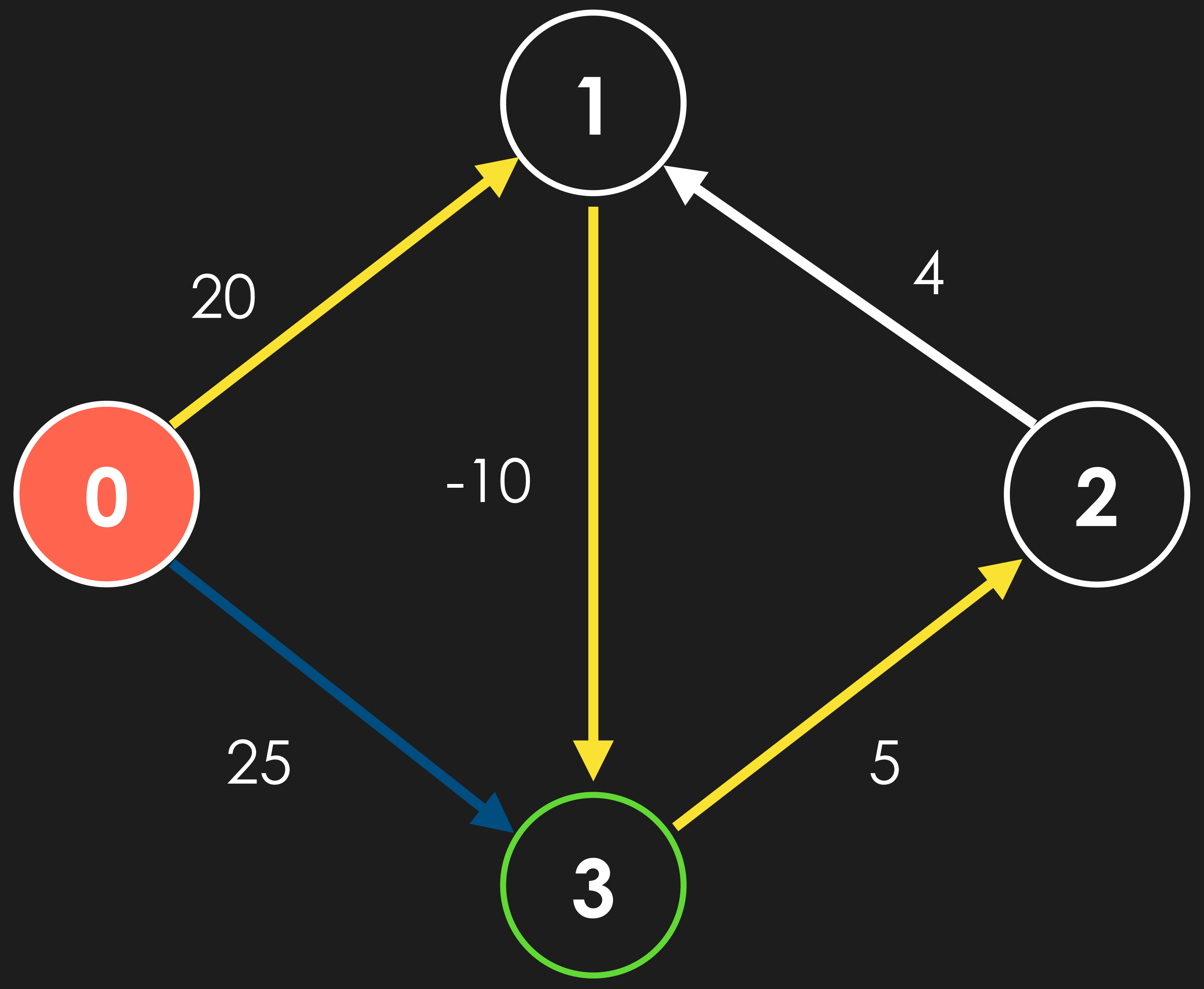
distTo

0	0
1	20
2	INF
3	10

edgeTo

0	-1
1	0 - 1
2	-1
3	1 - 3

Consider the following graph



3	10

pq

vertex	distTo
2	15

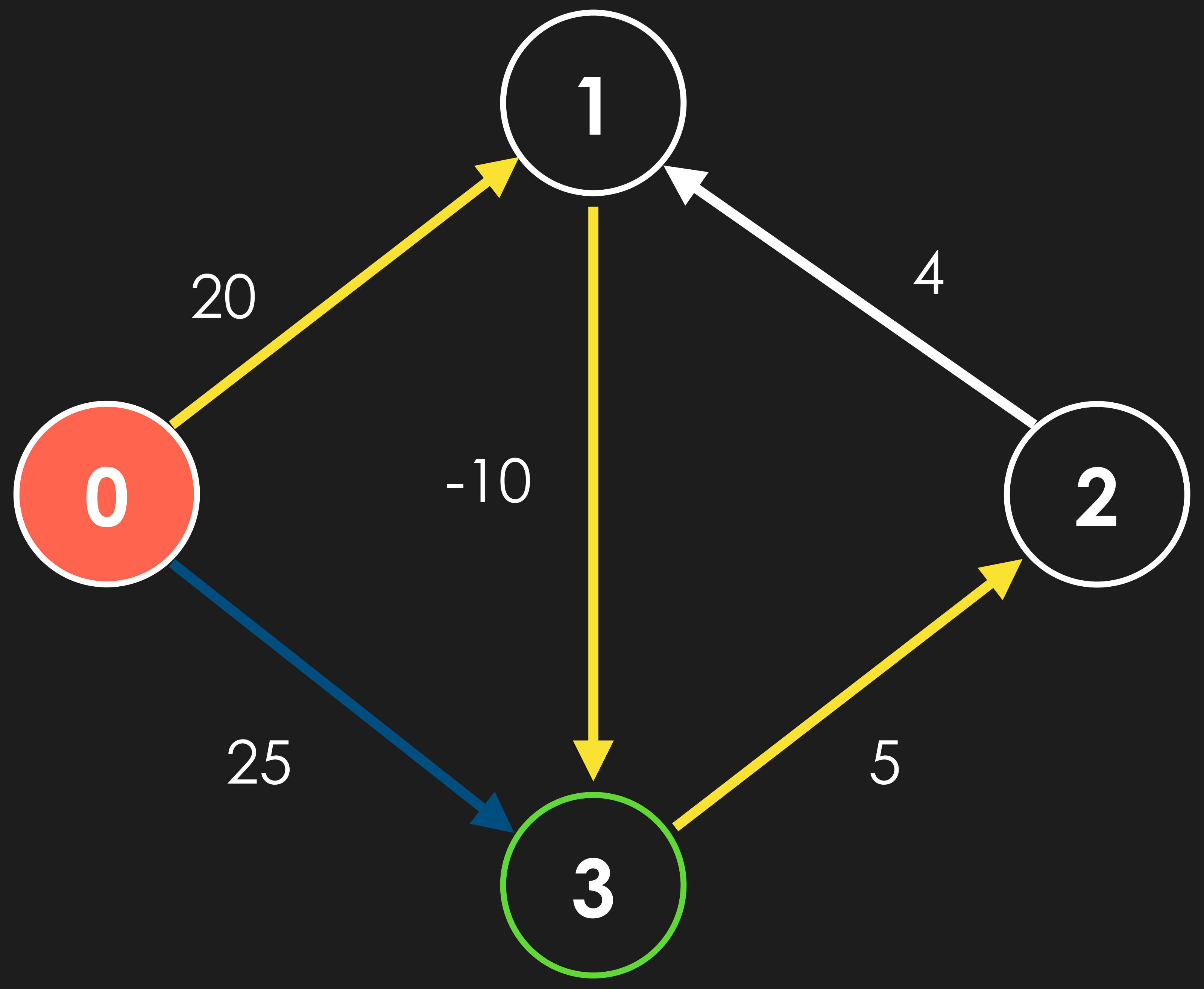
distTo

0	0
1	20
2	15
3	10

edgeTo

0	-1
1	0 - 1
2	3 - 2
3	1 - 3

Consider the following graph



3	10

pq

vertex	distTo
2	**15**

distTo

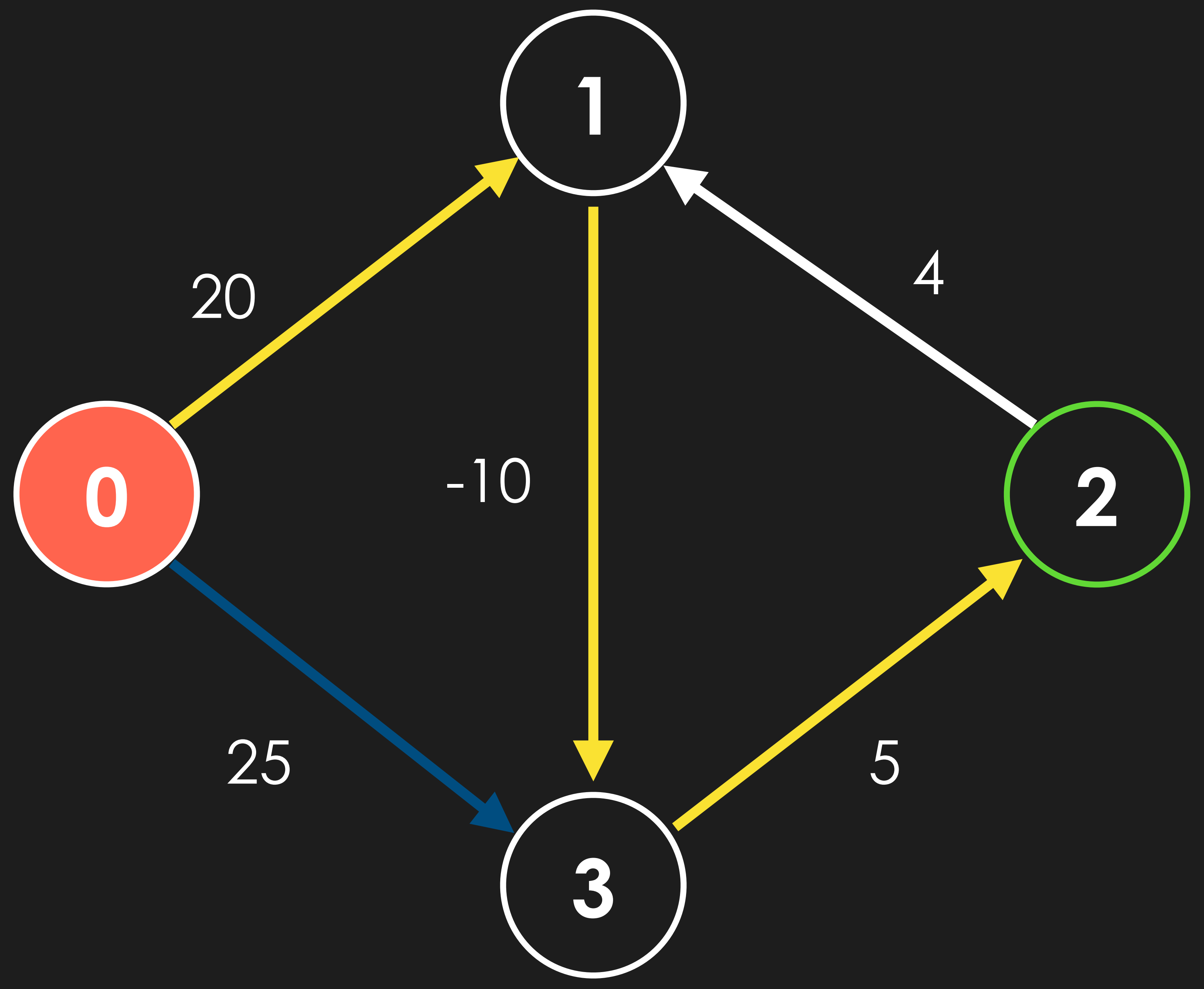
0	0
1	20
2	**15**
3	10

edgeTo

0	-1
1	0 - 1
2	**3 - 2**
3	1 - 3

We have reached the end state of the **problem!**

Consider the following graph



2	15

pq

vertex	distTo

distTo

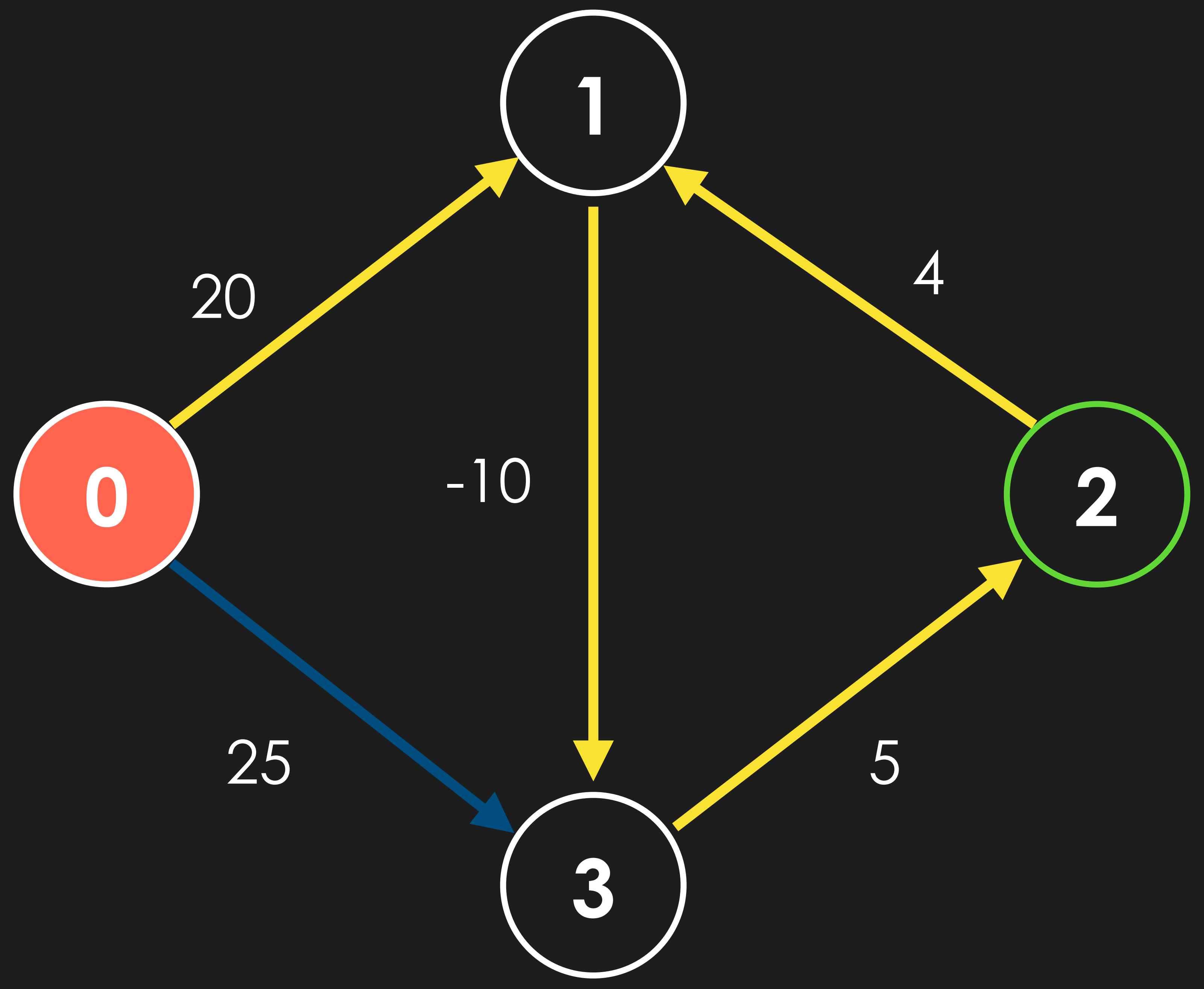
0	0
1	20
2	15
3	10

edgeTo

0	-1
1	0 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



2	15

pq

vertex	distTo
1	**19**

distTo

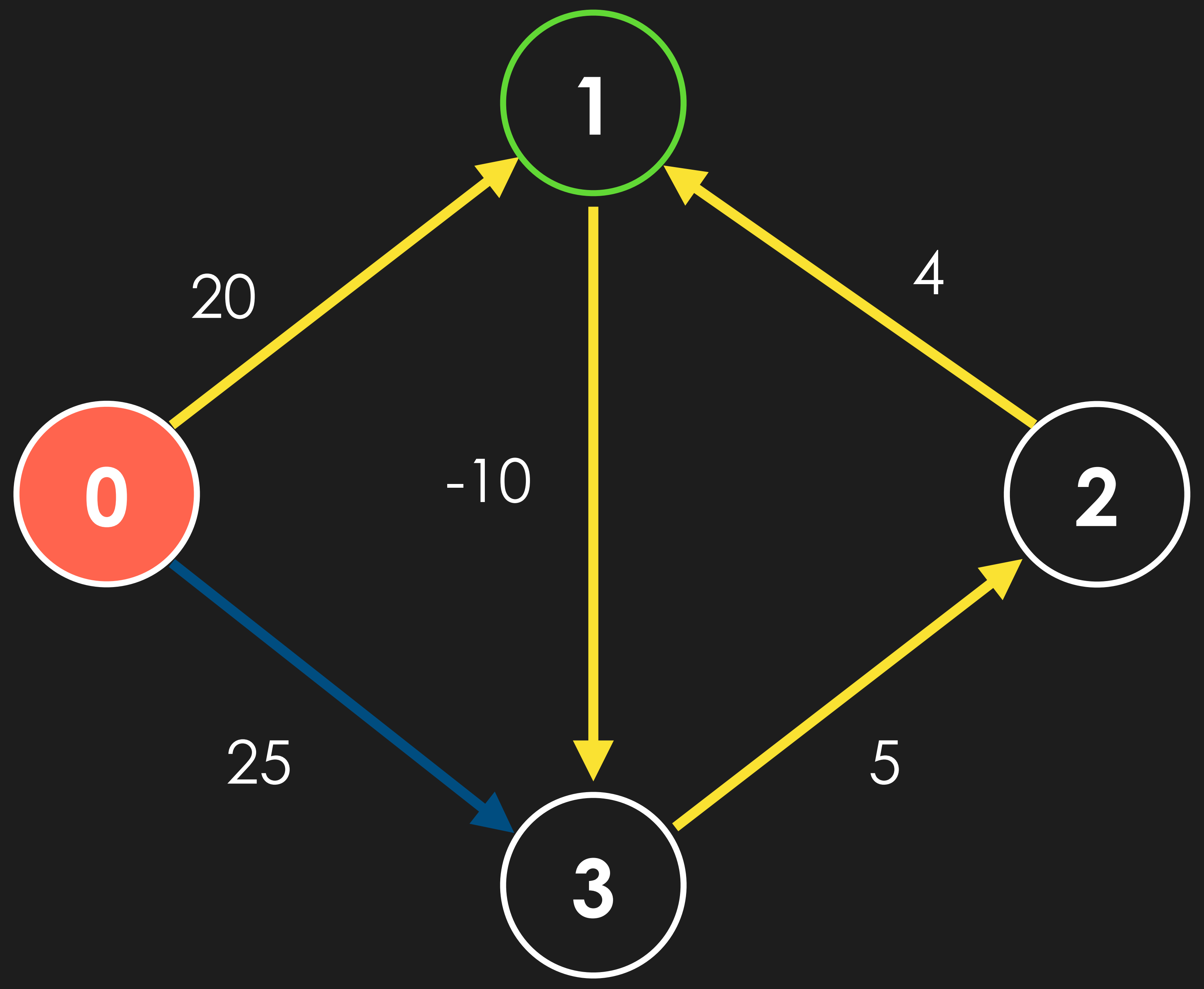
0	0
1	**19**
2	15
3	10

edgeTo

0	-1
1	**2 - 1**
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



1	19

pq

vertex	distTo

distTo

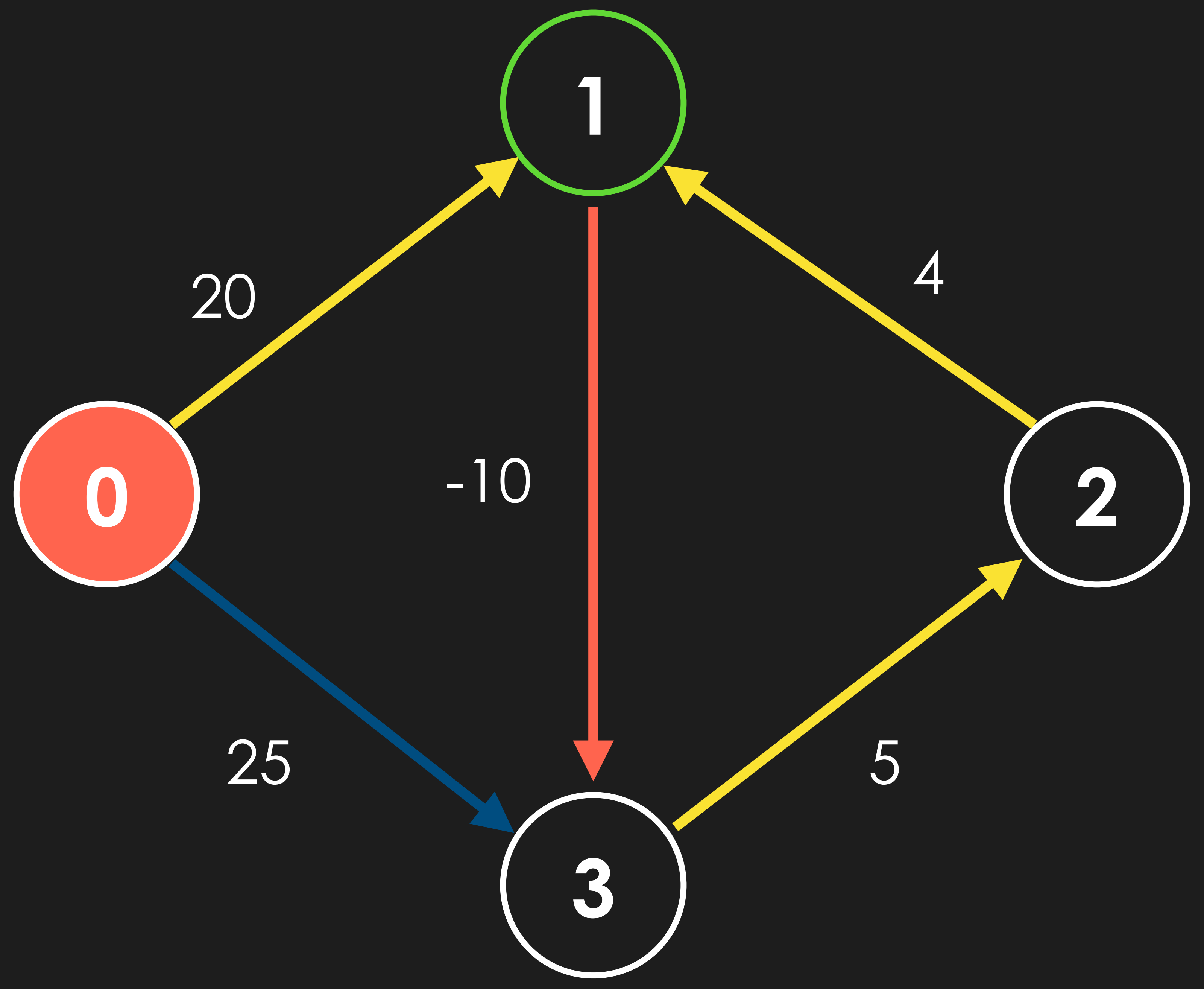
0	0
1	19
2	15
3	10

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



1	19

pq

vertex	distTo
3	9

distTo

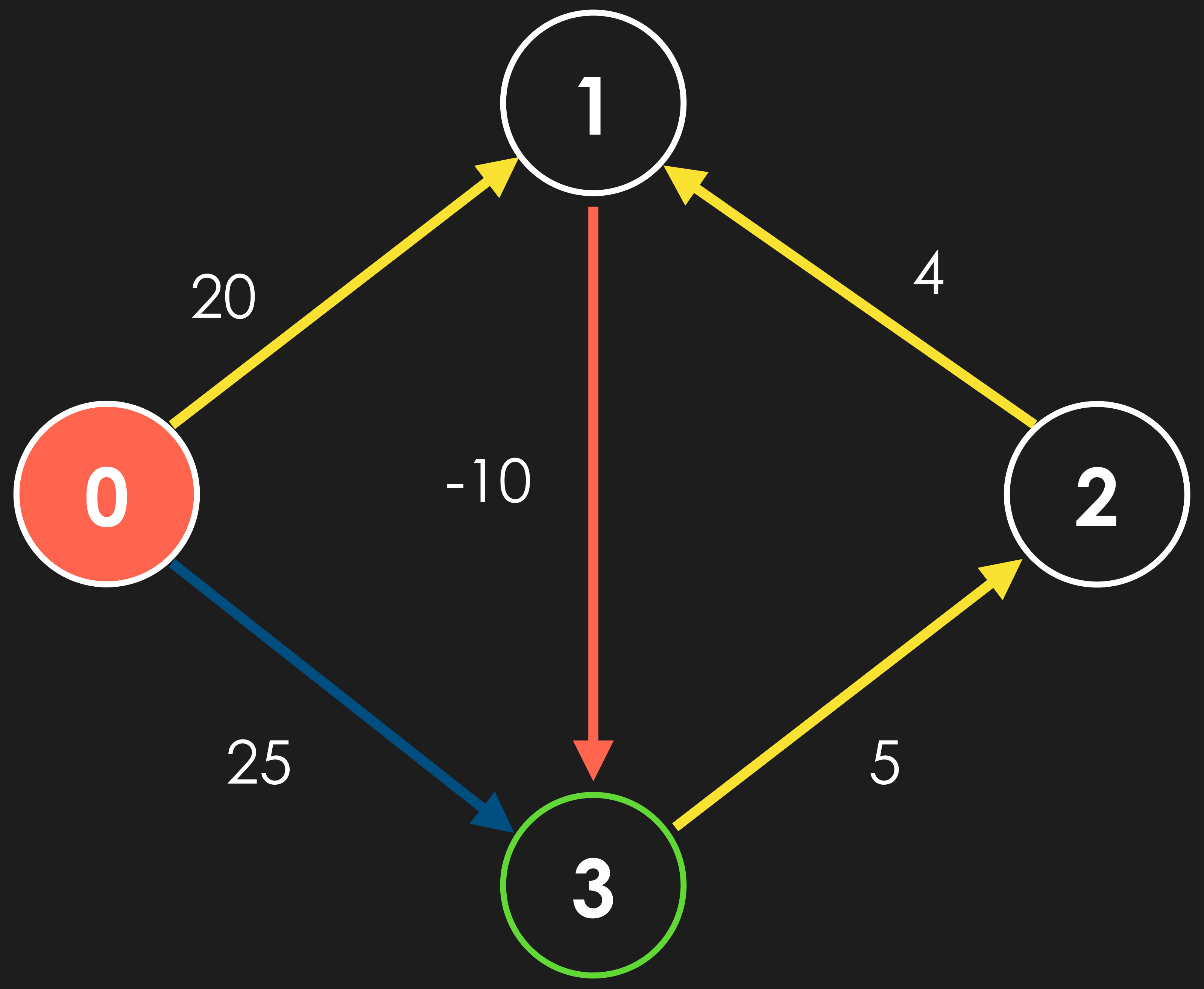
0	0
1	19
2	15
3	9

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



3	9

pq

vertex	distTo

distTo

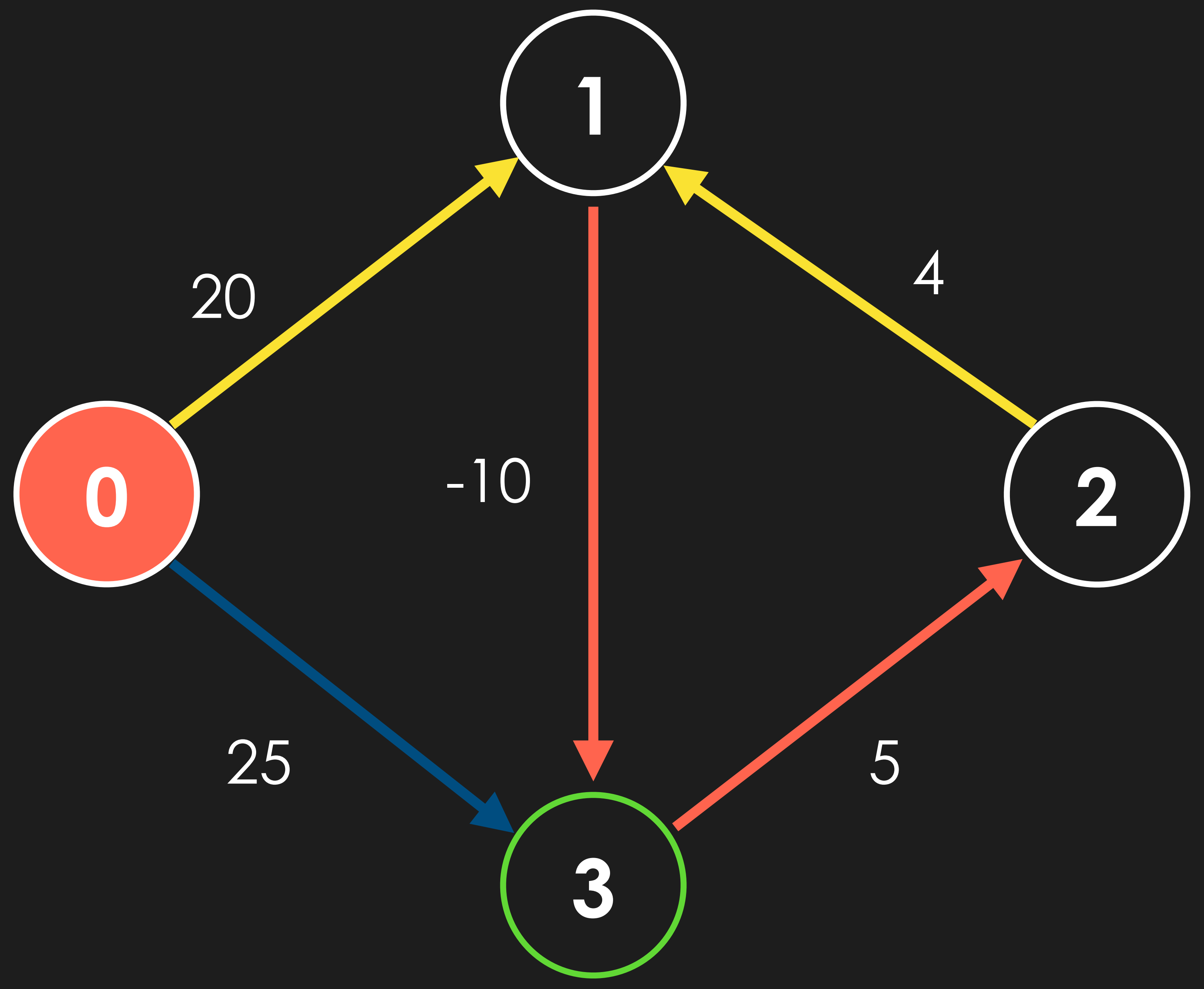
0	0
1	19
2	15
3	9

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



3	9

pq

vertex	distTo
2	**14**

distTo

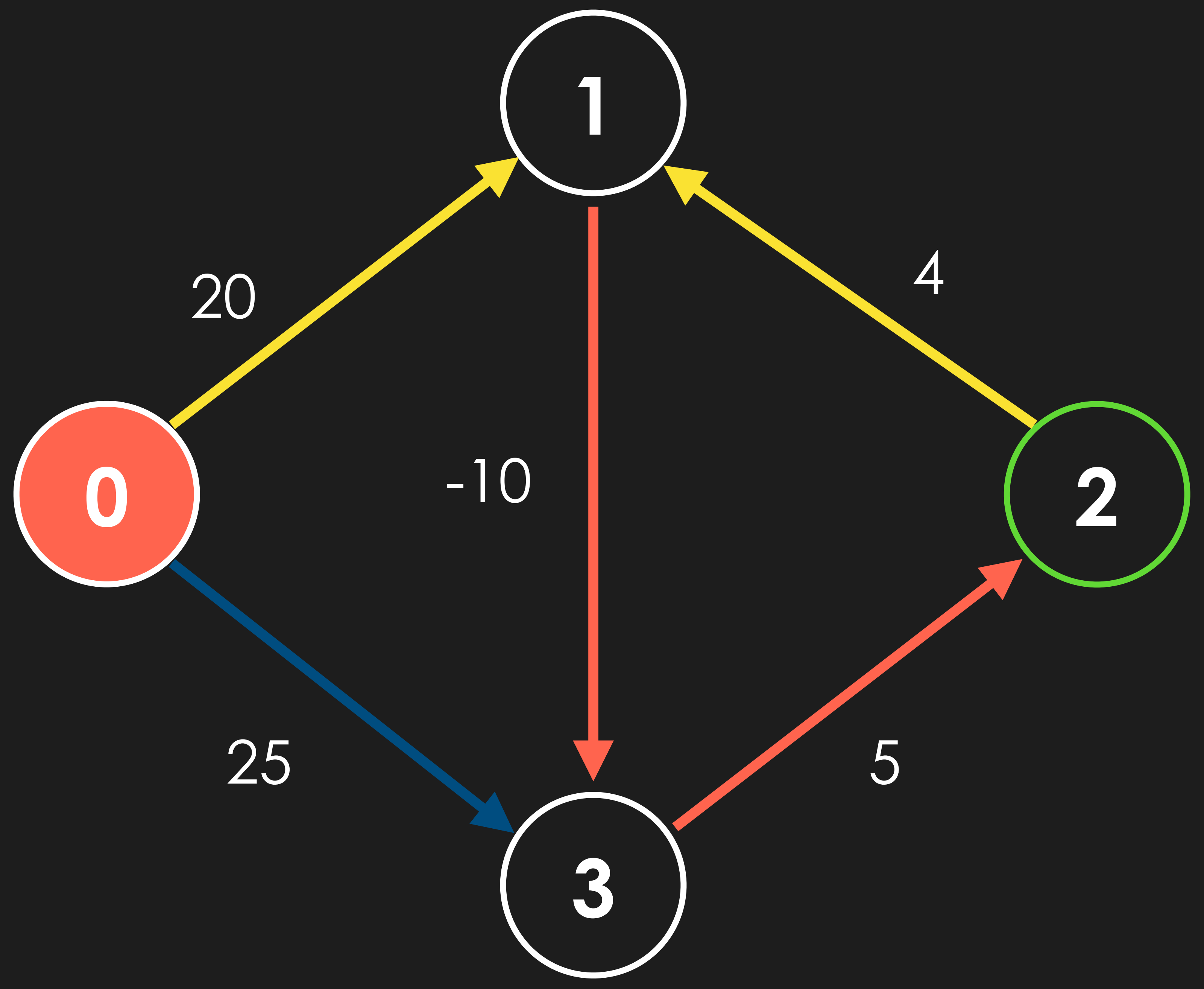
0	0
1	19
2	**14**
3	9

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



2	14

pq

vertex	distTo

distTo

0	0
1	19
2	14
3	9

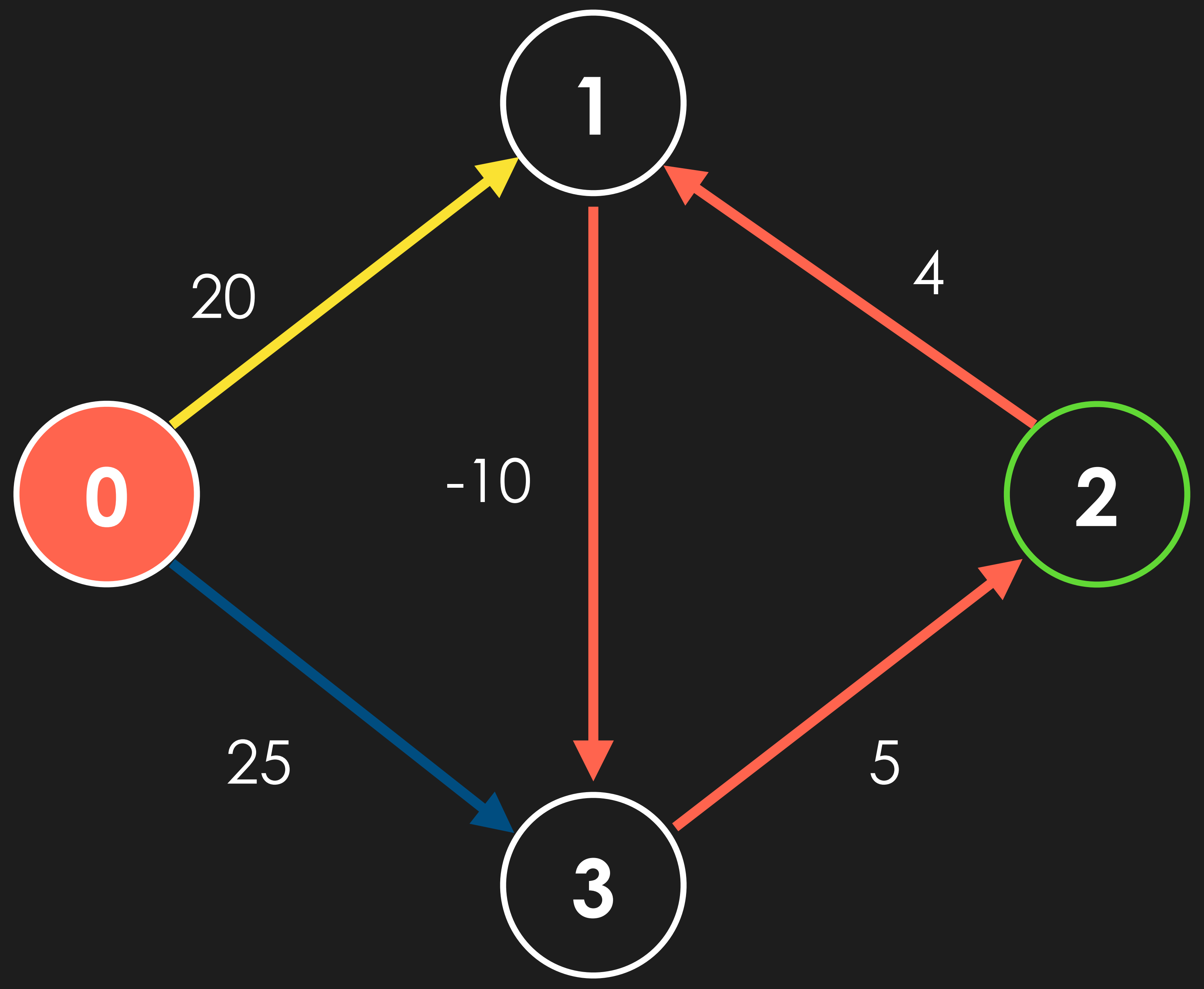
edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

CodeIT

Consider the following graph



2	14

pq

vertex	distTo
1	**18**

distTo

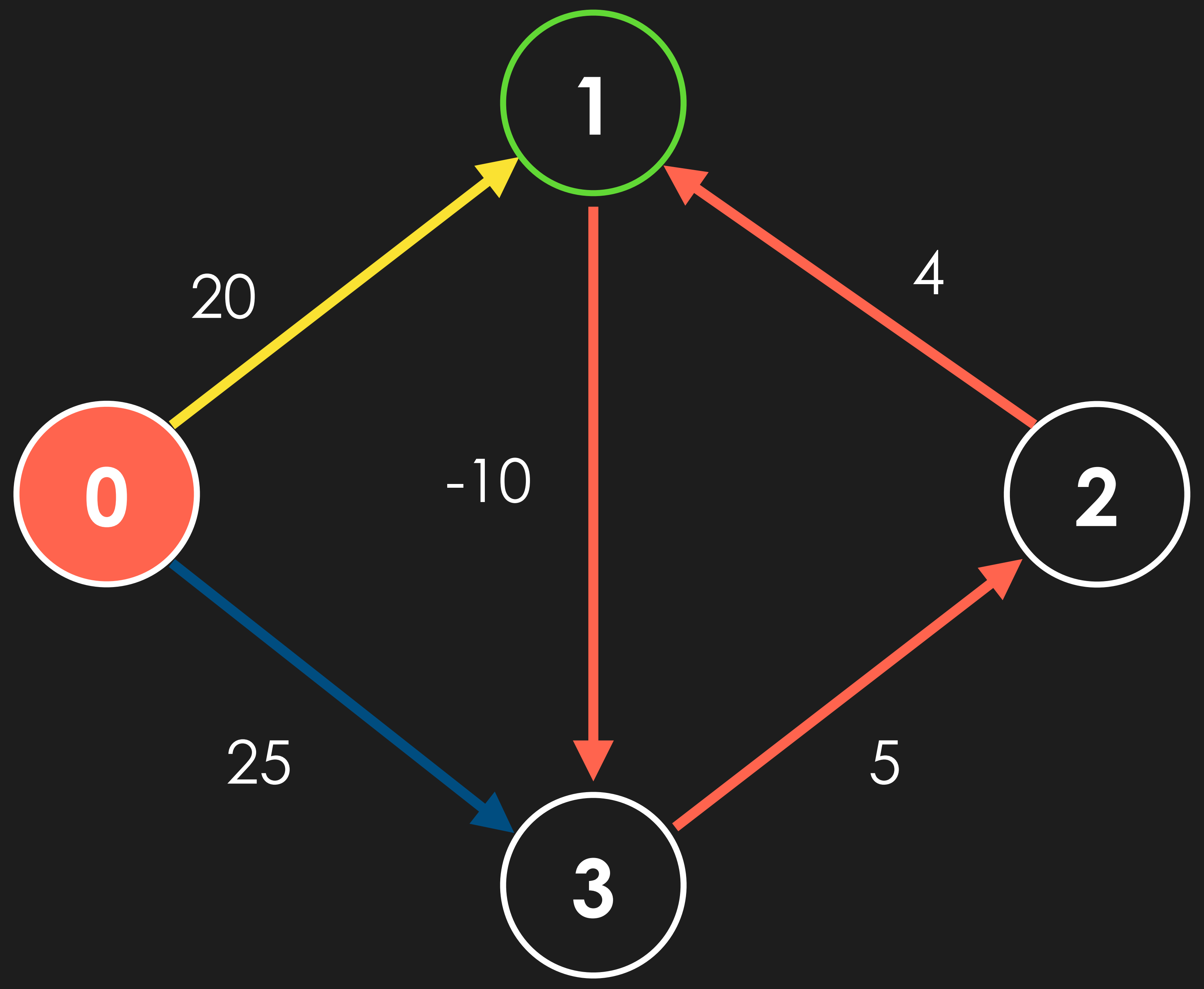
0	0
1	**18**
2	14
3	9

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



1	18

pq

vertex	distTo

distTo

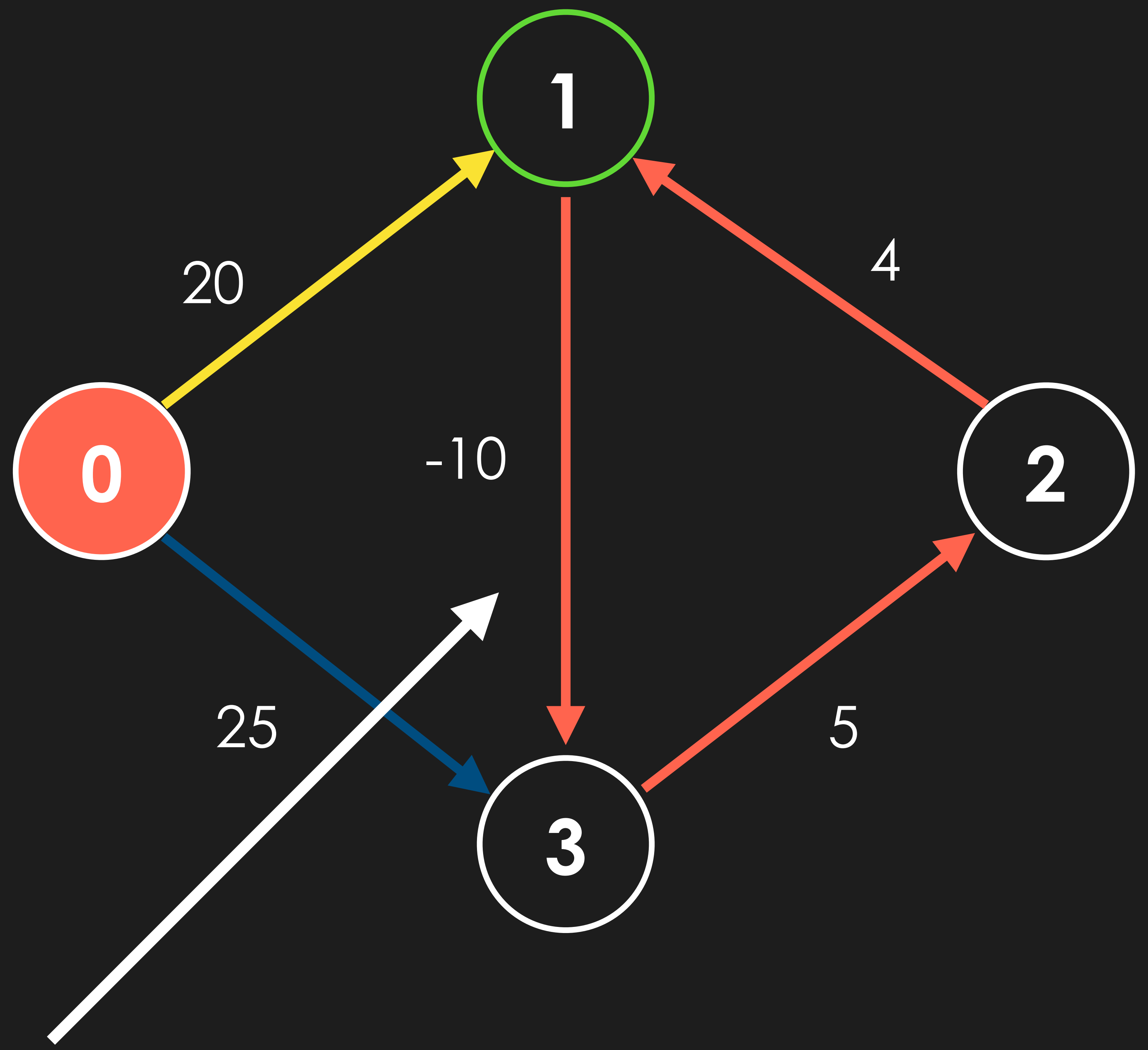
0	0
1	14
2	14
3	9

edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

However, the algorithm doesn't **terminate**!

Consider the following graph



1	18

pq

vertex	distTo

distTo

0	0
1	14
2	14
3	9

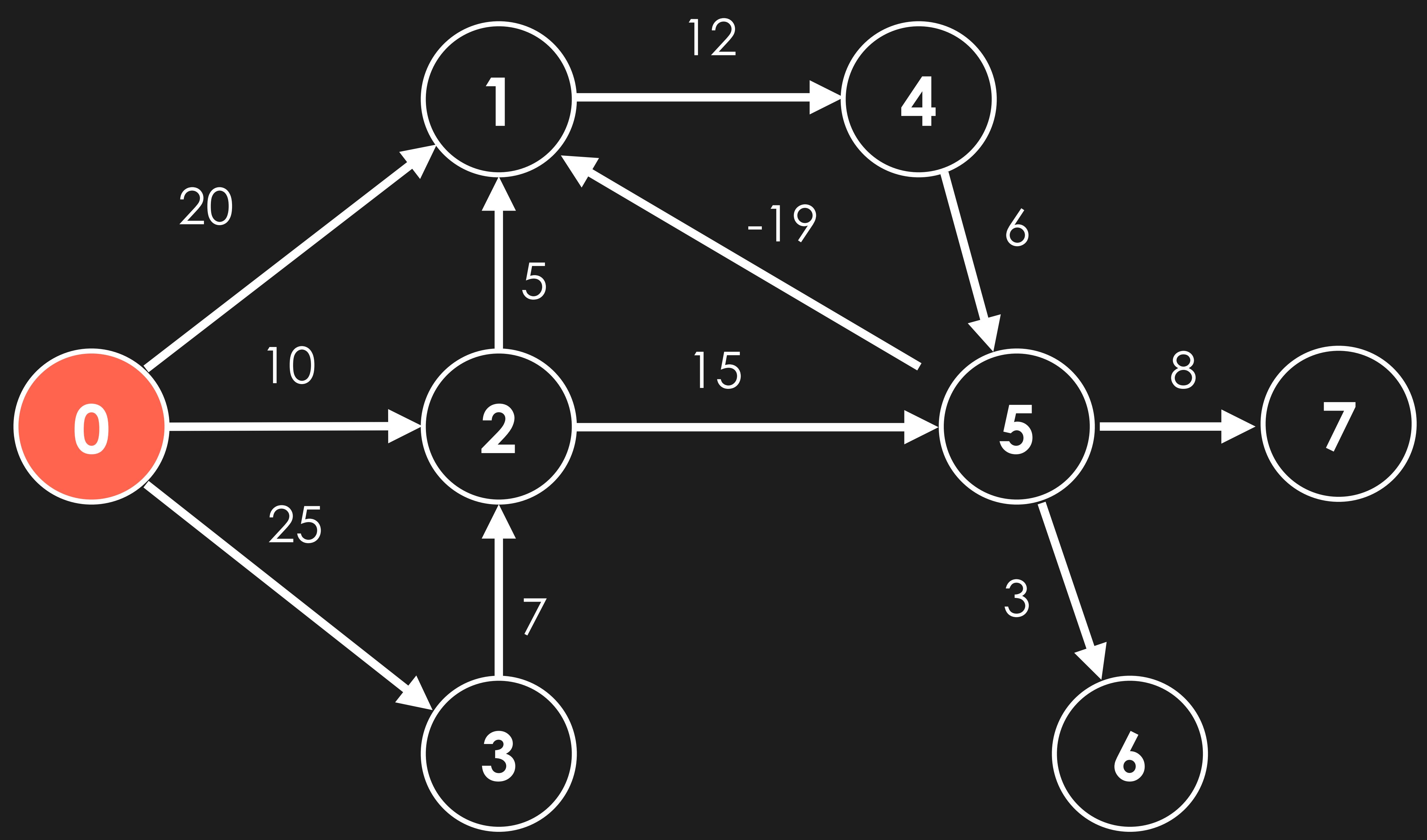
edgeTo

0	-1
1	2 - 1
2	3 - 2
3	1 - 3

This is because a **negative cycle** occurs (-10 + 4 + 5 = -1)

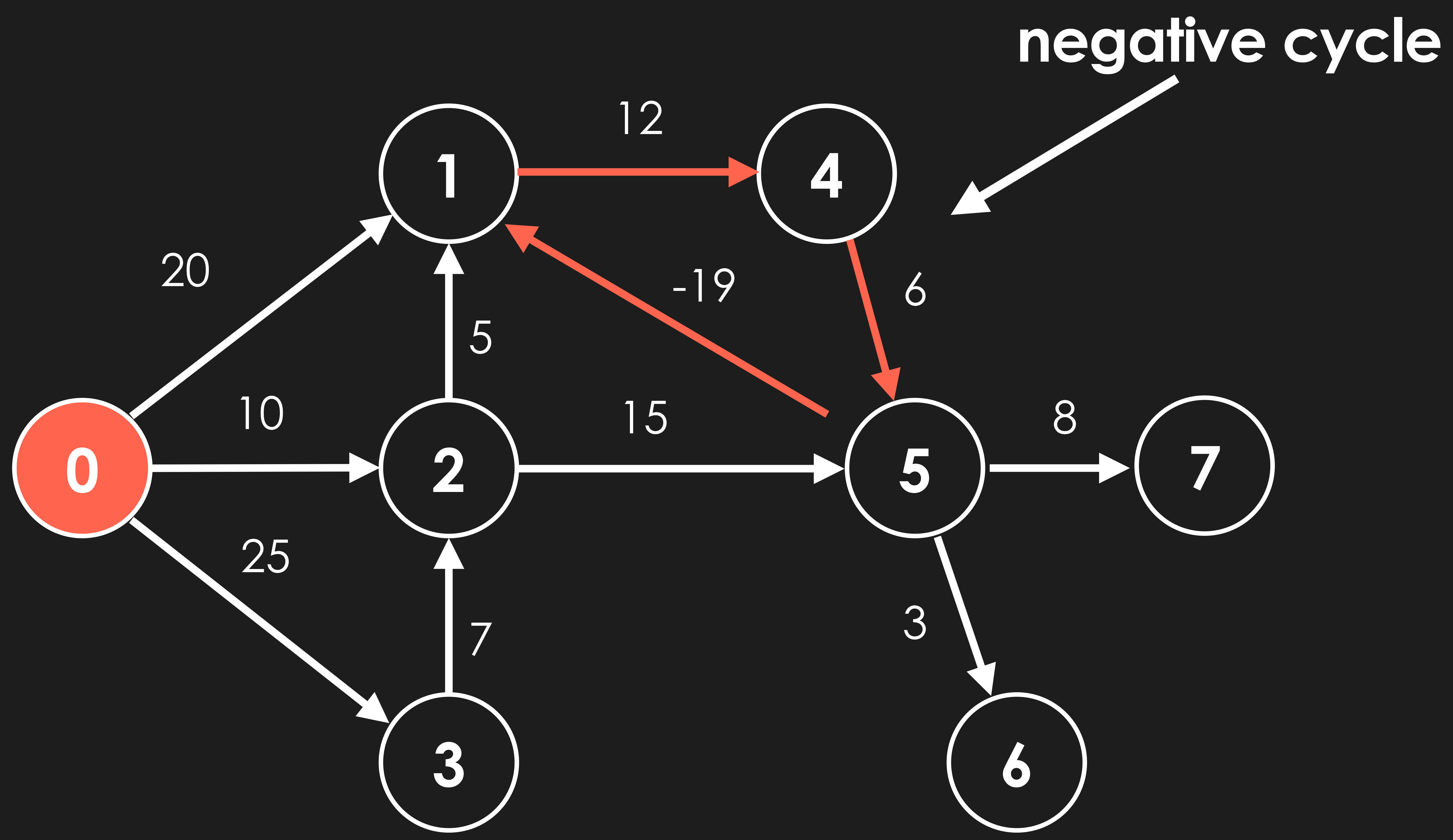
How then, can we solve the shortest path problem in a weighted digraph with negative cycles?

Consider the following graph:



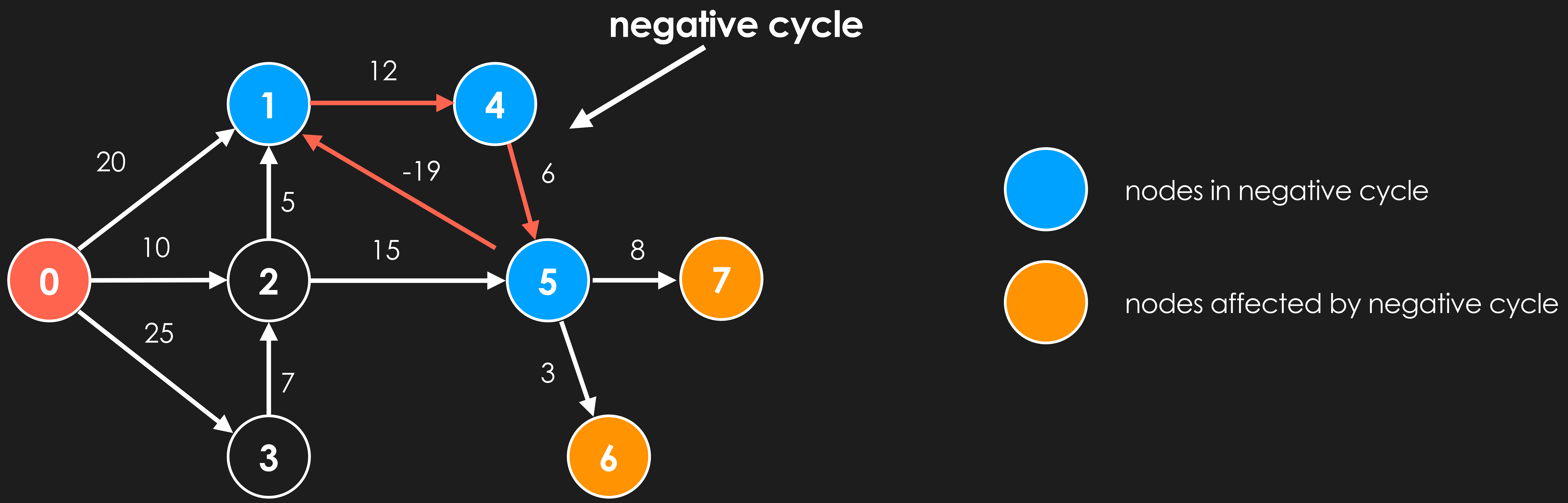
CodeIT

Consider the following graph:



CodeIT

Consider the following graph:



Vertices involved / affected by a negative cycle have no shortest path because the shortest path to these vertices are negative infinity

CodeIT

Bellman-Ford Algorithm

Bellman-Ford Algorithm

1. For each vertex, relax every edge to reach the **shortest path tree**

2. Relax every edge once, and if a distTo[edge.dest] ever decreases, a negative cycle is detected

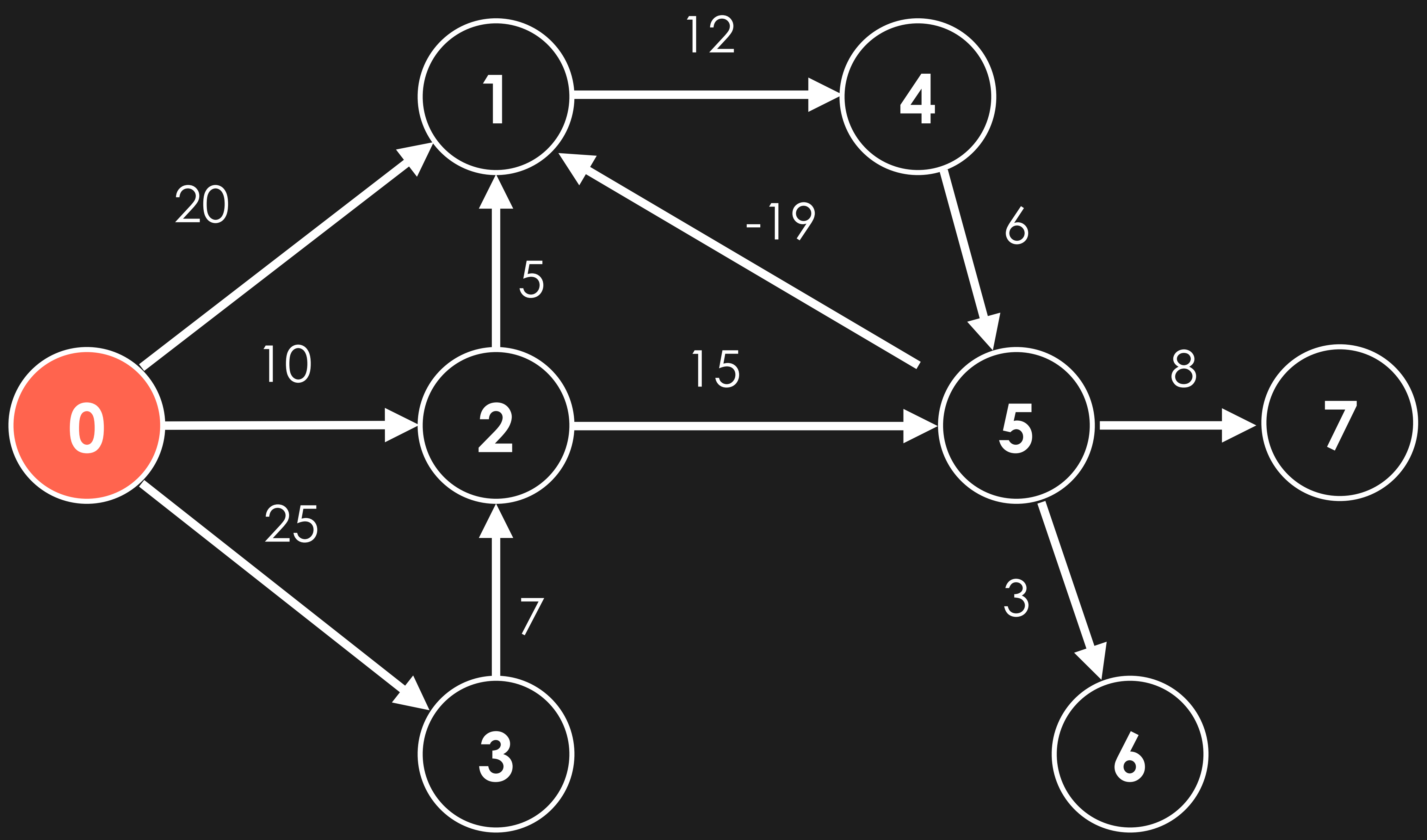
Bellman-Ford Algorithm

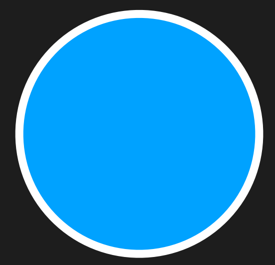
1. For each vertex, relax every edge to reach the **shortest path tree**

2. Relax every edge once, and if a distTo[edge.dest] ever decreases, a negative cycle is detected

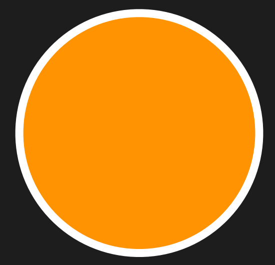
Note: The Bellman-Ford algorithm allows you to **detect** a negative cycle, but not every negative cycle in a graph, especially if they are intersecting

1. For each vertex, relax every edge to reach the **shortest path tree**





nodes in negative cycle

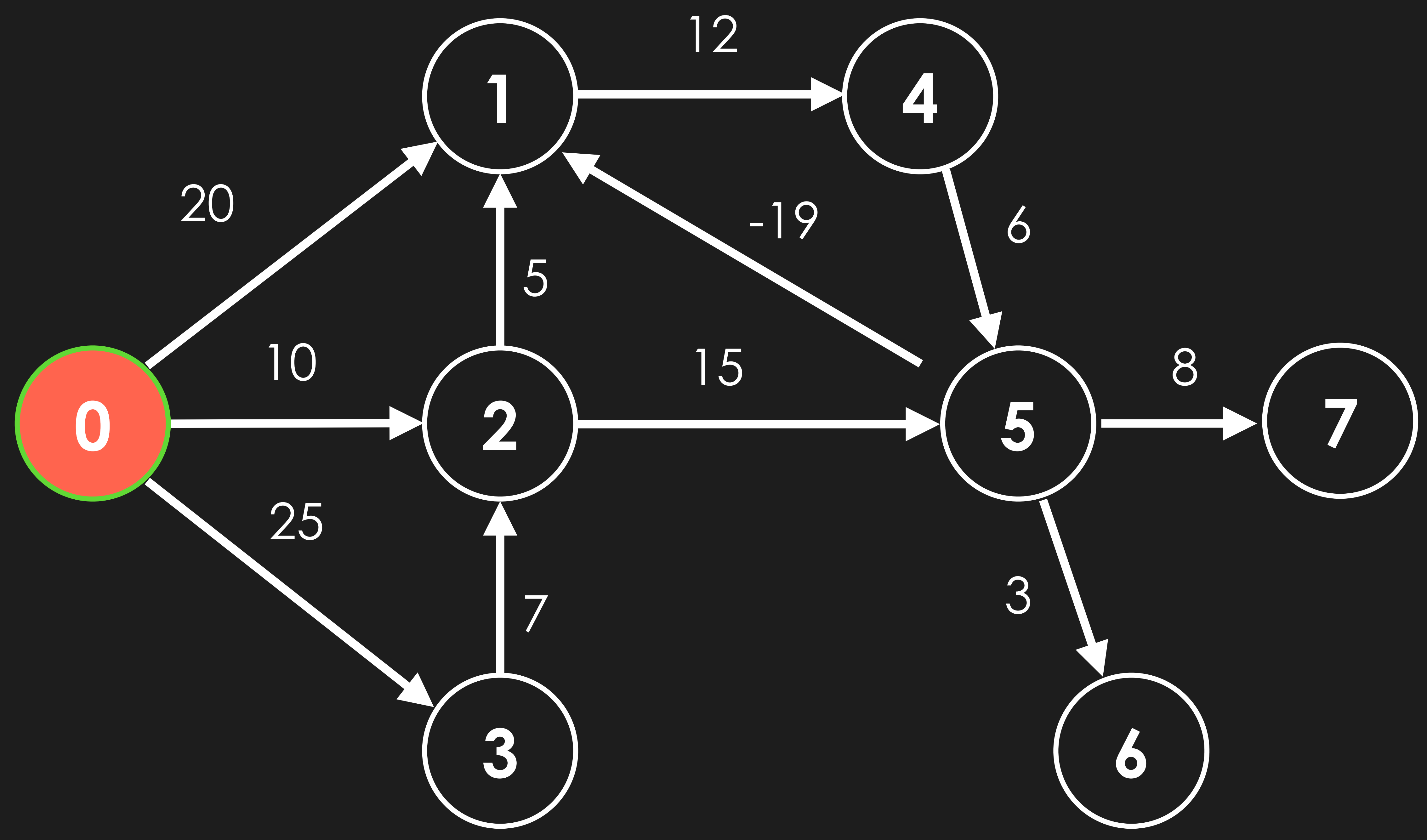


nodes affected by negative cycle

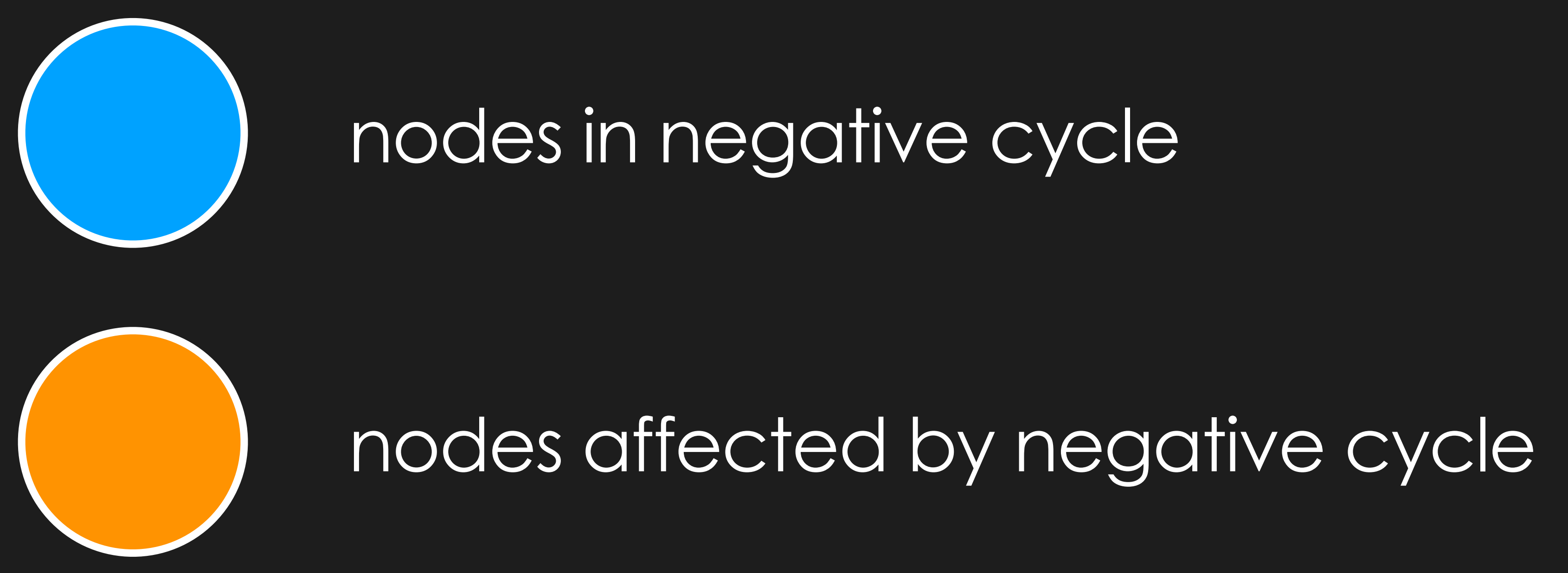
	distTo
0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

1. For each vertex, relax every edge to reach the **shortest path tree**



vertex 0

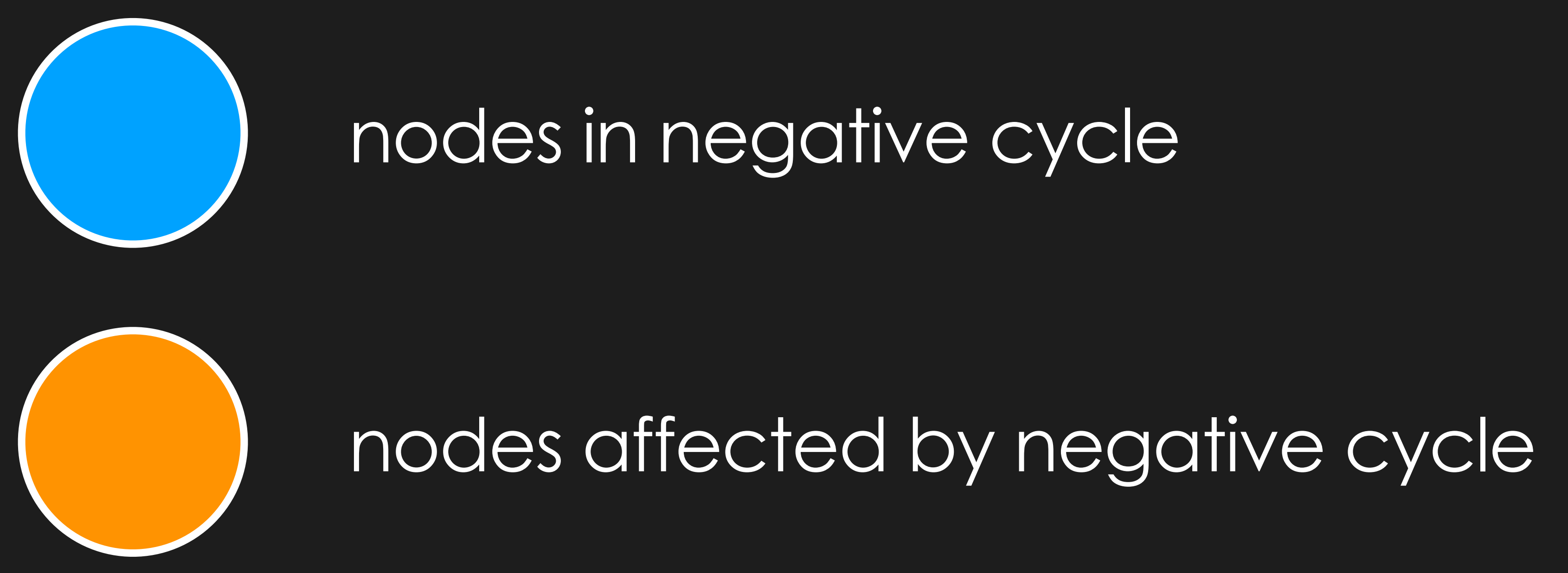


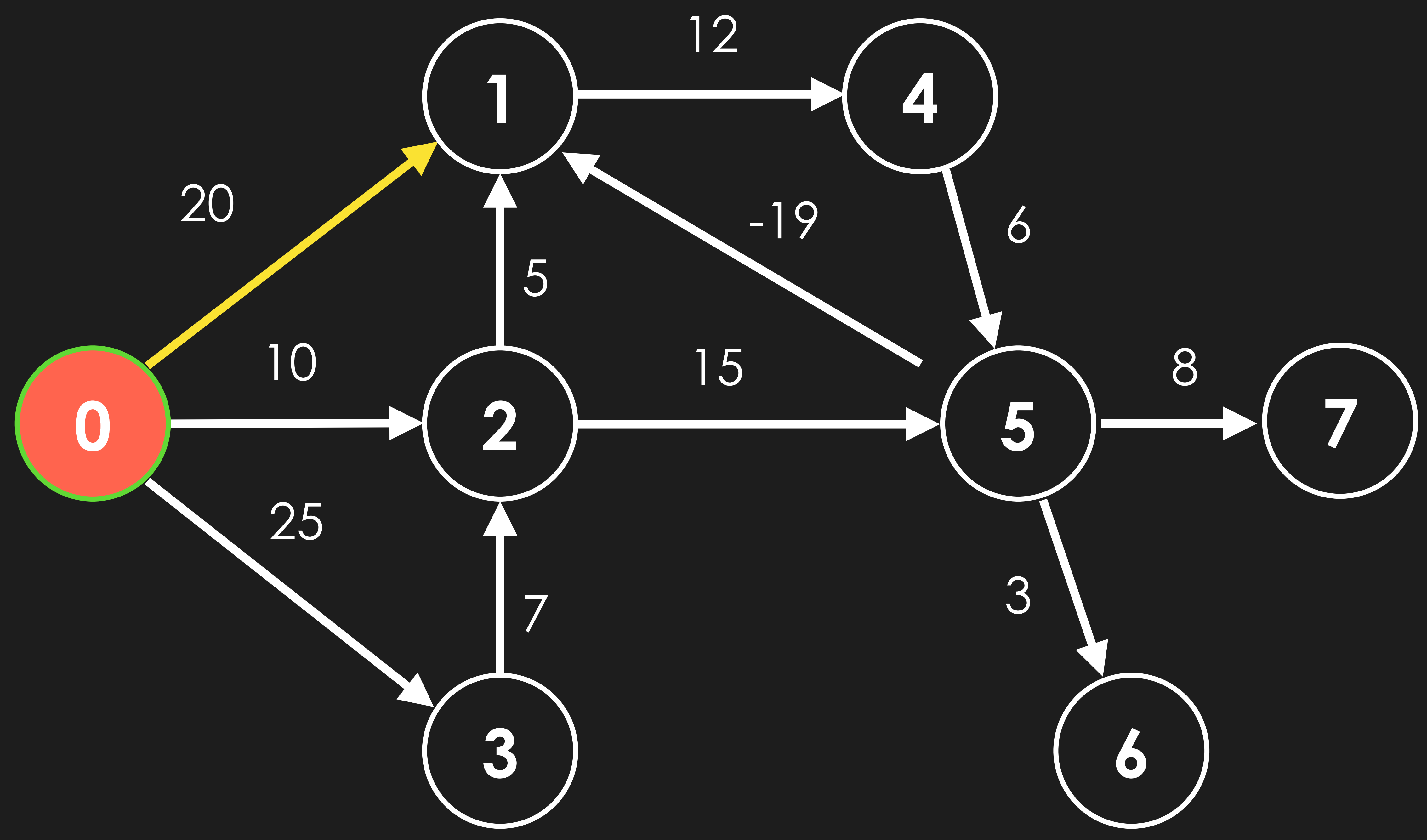
	distTo
0	0
1	INF
2	INF
3	INF
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**



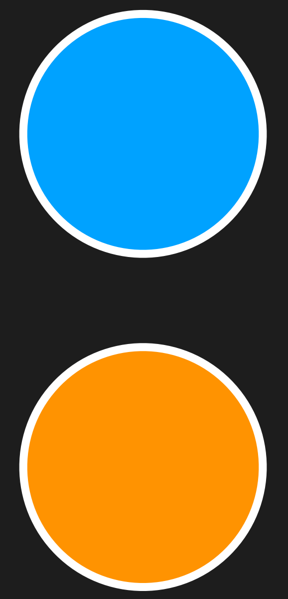


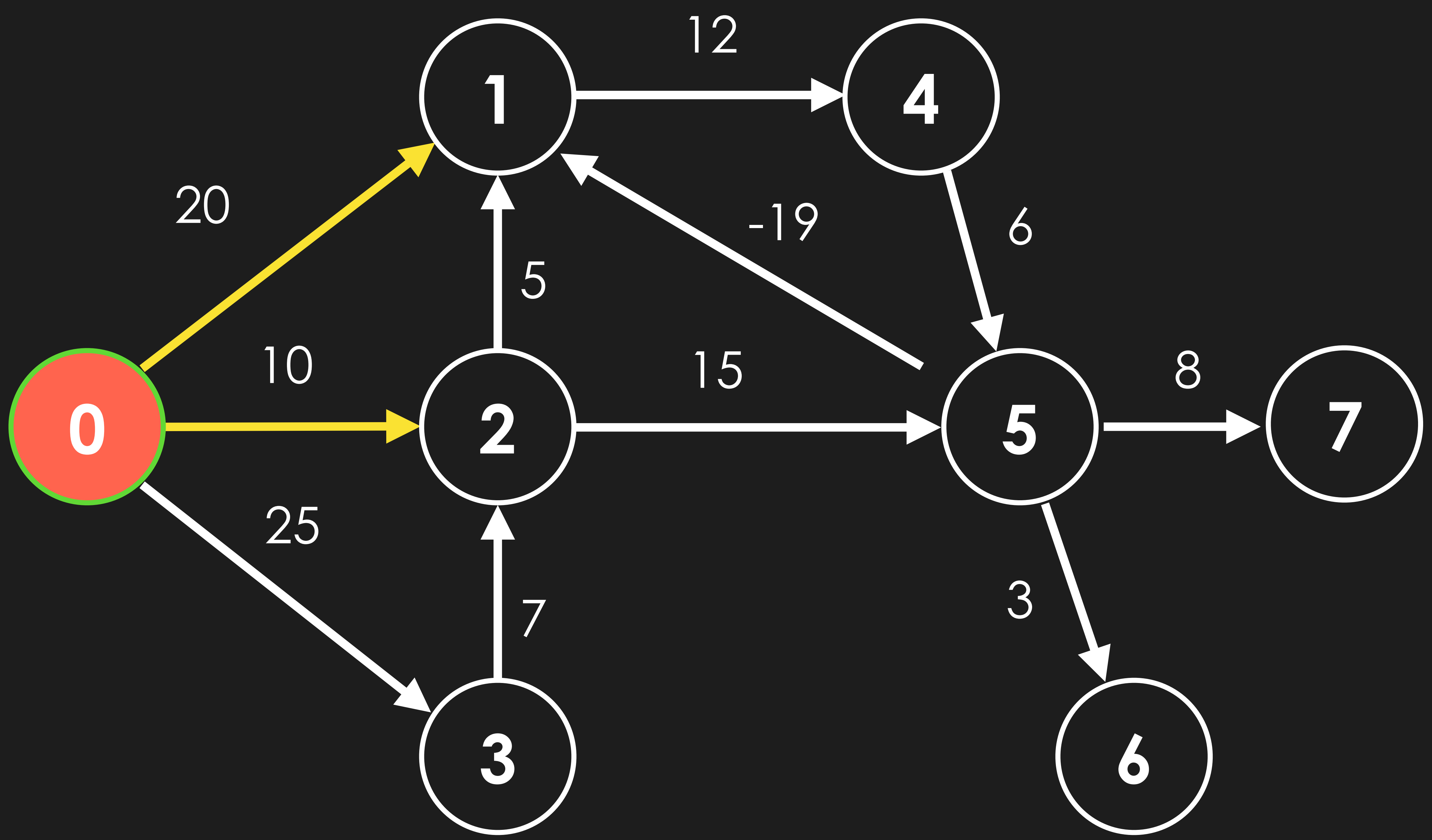
	distTo		edgeTo
0	0	0	-1
1	20	1	0 - 1
2	INF	2	-1
3	INF	3	-1
4	INF	4	-1
5	INF	5	-1
6	INF	6	-1
7	INF	7	-1

vertex 0

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**





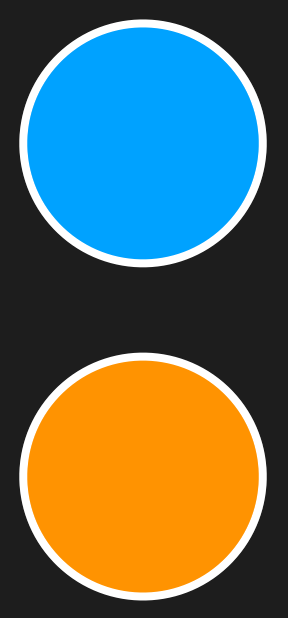
vertex 0

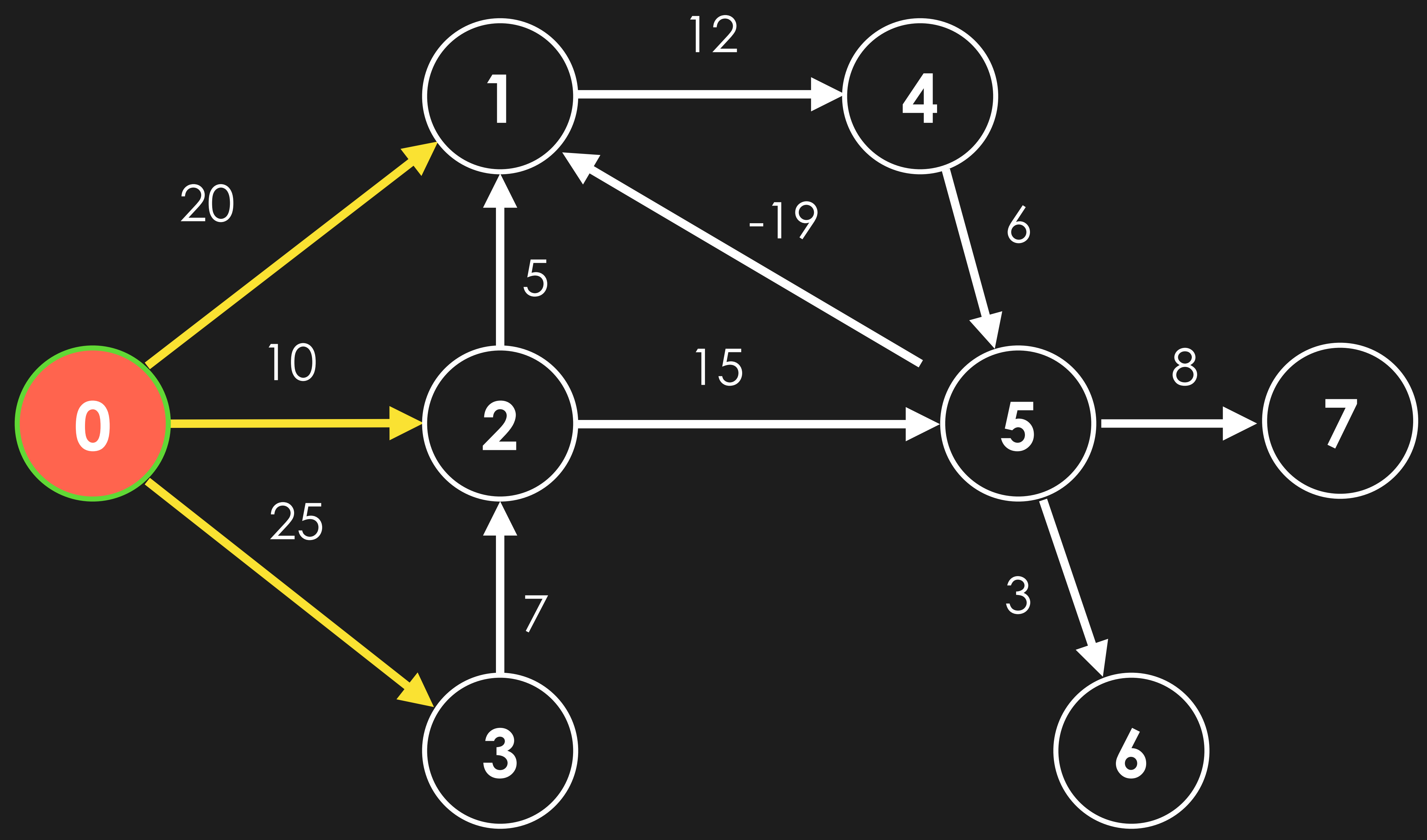
	distTo
0	0
1	20
2	10
3	INF
4	INF
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	0 - 1
2	0 - 2
3	-1
4	-1
5	-1
6	-1
7	-1

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**

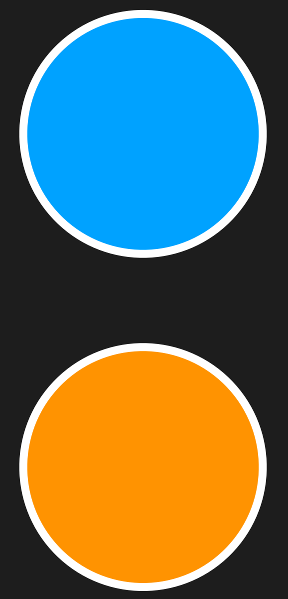


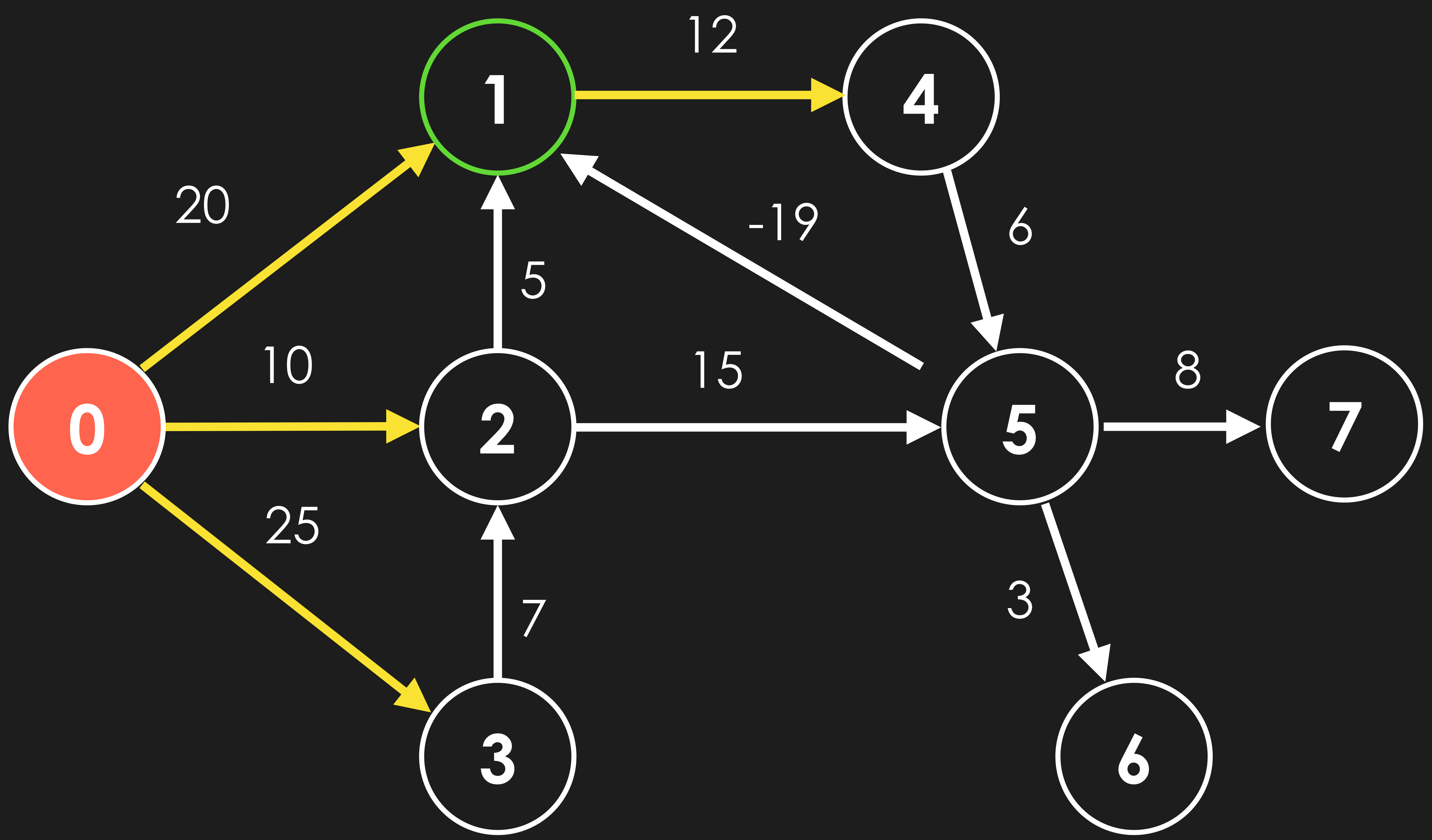


	distTo		edgeTo
0	0	0	-1
1	20	1	0 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	INF	4	-1
5	INF	5	-1
6	INF	6	-1
7	INF	7	-1

vertex 0

1. For each vertex, relax every edge to reach the **shortest path tree**





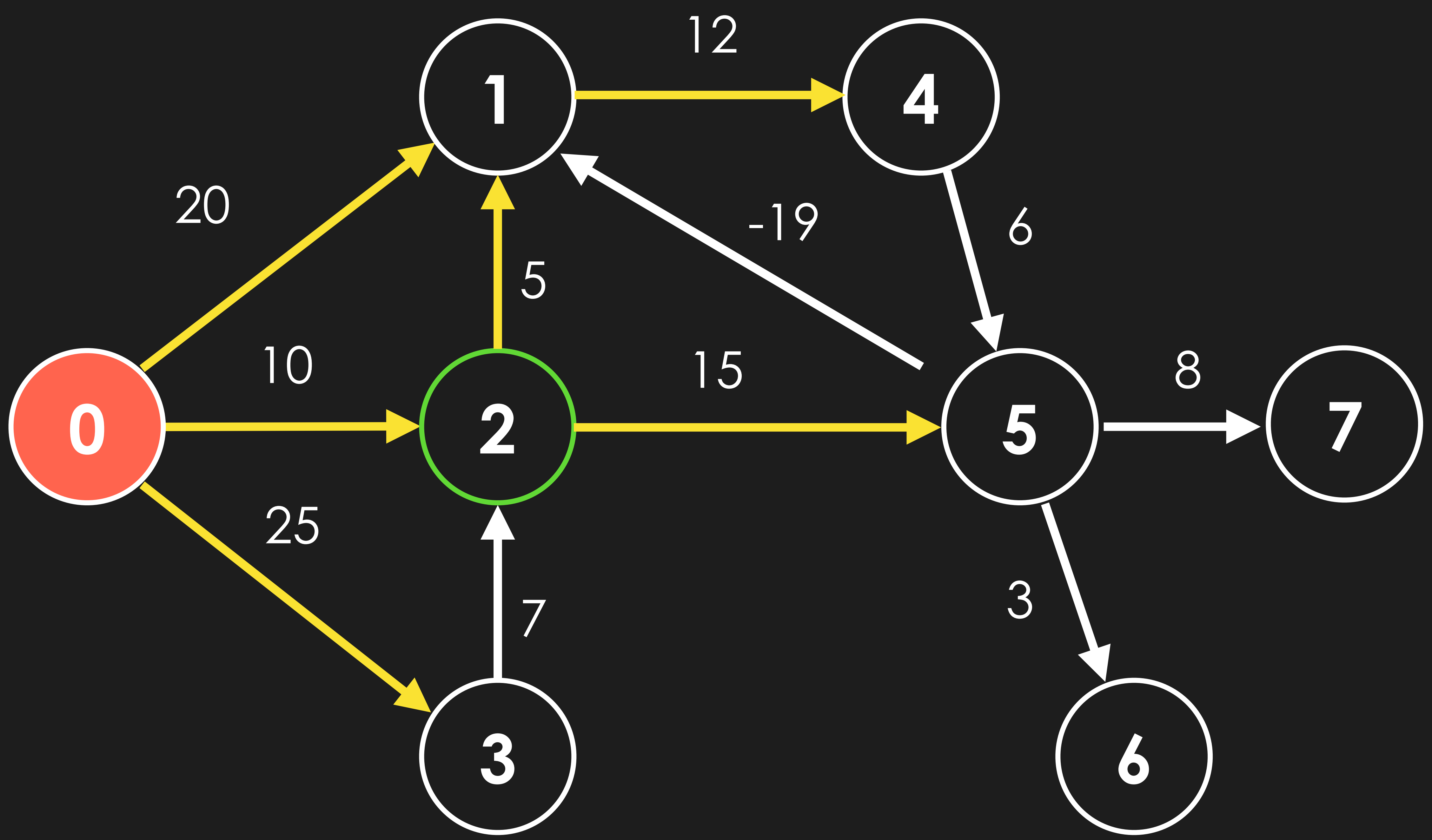
	distTo
0	0
1	20
2	10
3	25
4	**32**
5	INF
6	INF
7	INF

	edgeTo
0	-1
1	0 - 1
2	0 - 2
3	0 - 3
4	**1 - 4**
5	-1
6	-1
7	-1

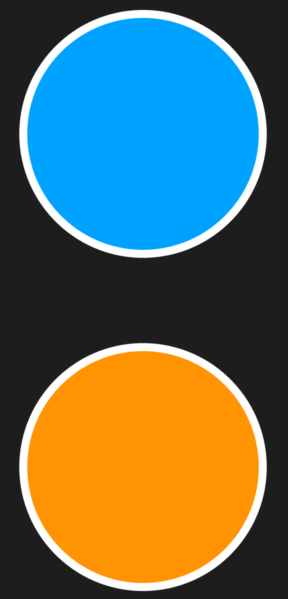
vertex 0

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**



vertex 0



distTo

0	0
1	**15**
2	10
3	25
4	32
5	**25**
6	INF
7	INF

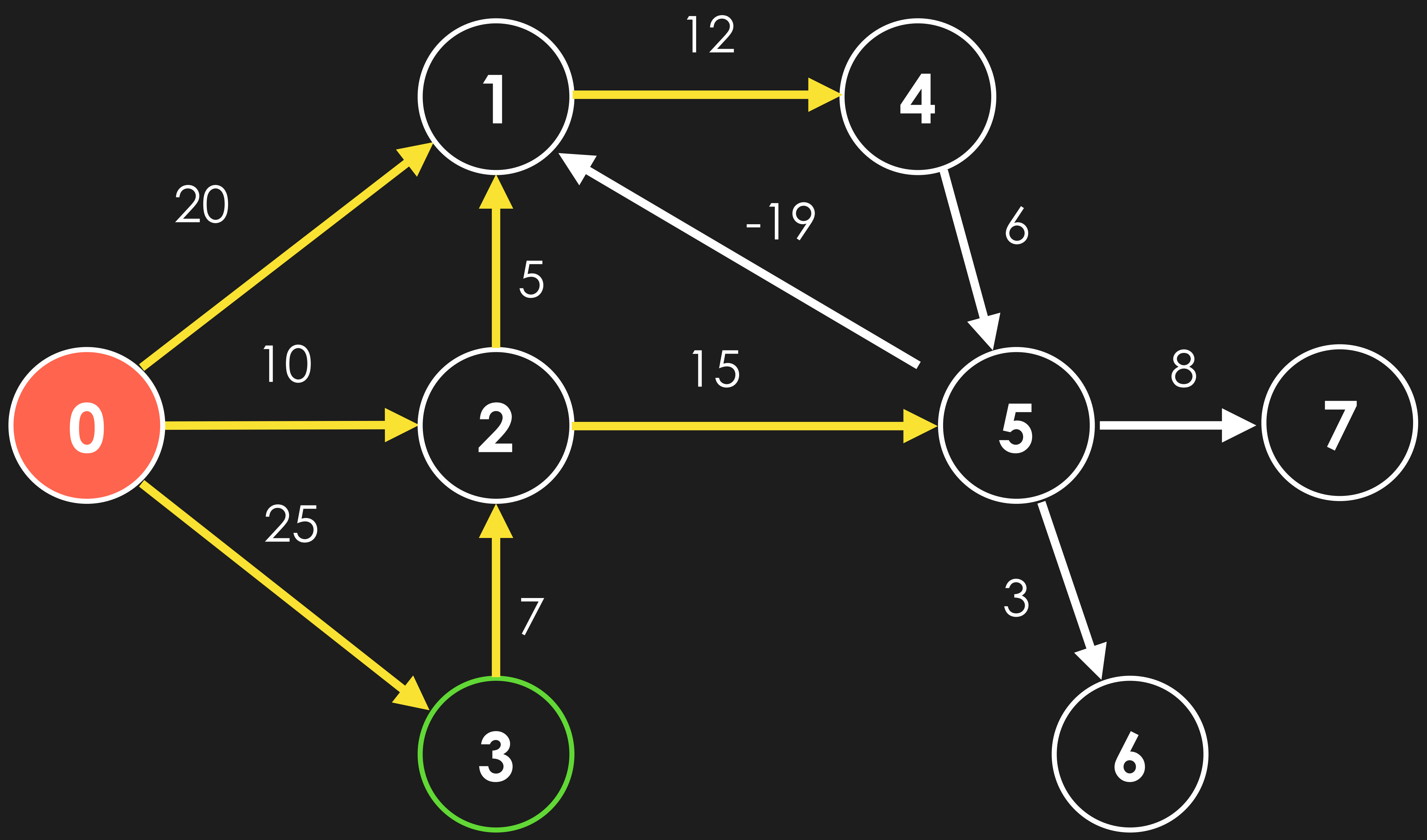
edgeTo

0	-1
1	**2 - 1**
2	0 - 2
3	0 - 3
4	1 - 4
5	**2 - 5**
6	-1
7	-1

1. For each vertex, relax every edge to reach the **shortest path tree**

nodes in negative cycle

nodes affected by negative cycle



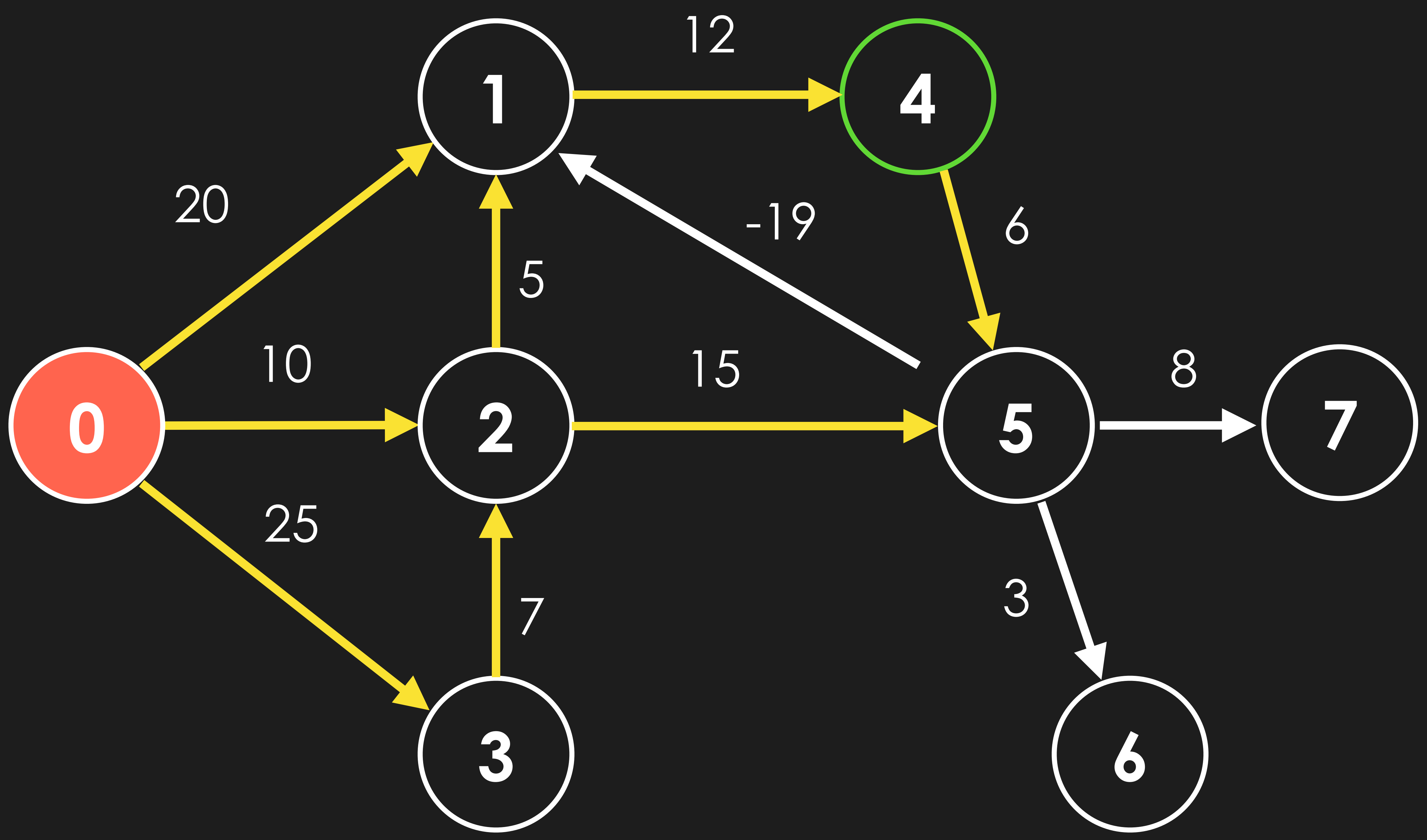
	distTo	edgeTo
0	0	-1
1	15	2 - 1
2	10	0 - 2
3	25	0 - 3
4	32	1 - 4
5	25	2 - 5
6	INF	-1
7	INF	-1

vertex 0

1. For each vertex, relax every edge to reach the **shortest path tree**

nodes in negative cycle

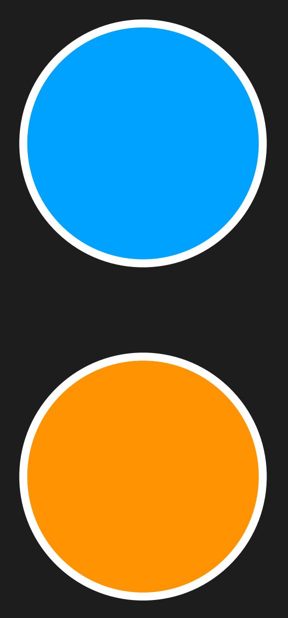
nodes affected by negative cycle

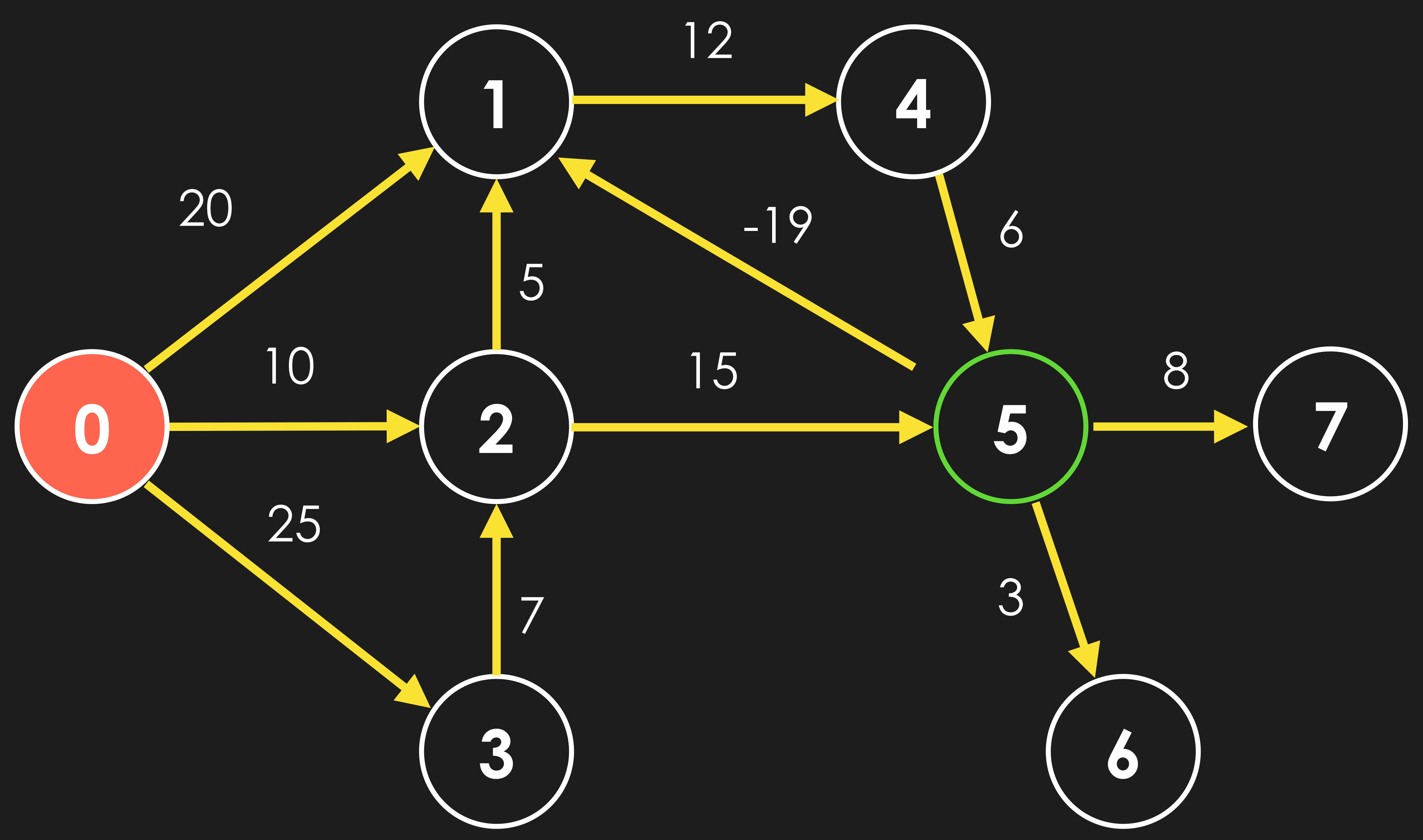


vertex 0

	distTo	edgeTo
0	0	-1
1	15	2 - 1
2	10	0 - 2
3	25	0 - 3
4	32	1 - 4
5	25	2 - 5
6	INF	-1
7	INF	-1

1. For each vertex, relax every edge to reach the **shortest path tree**



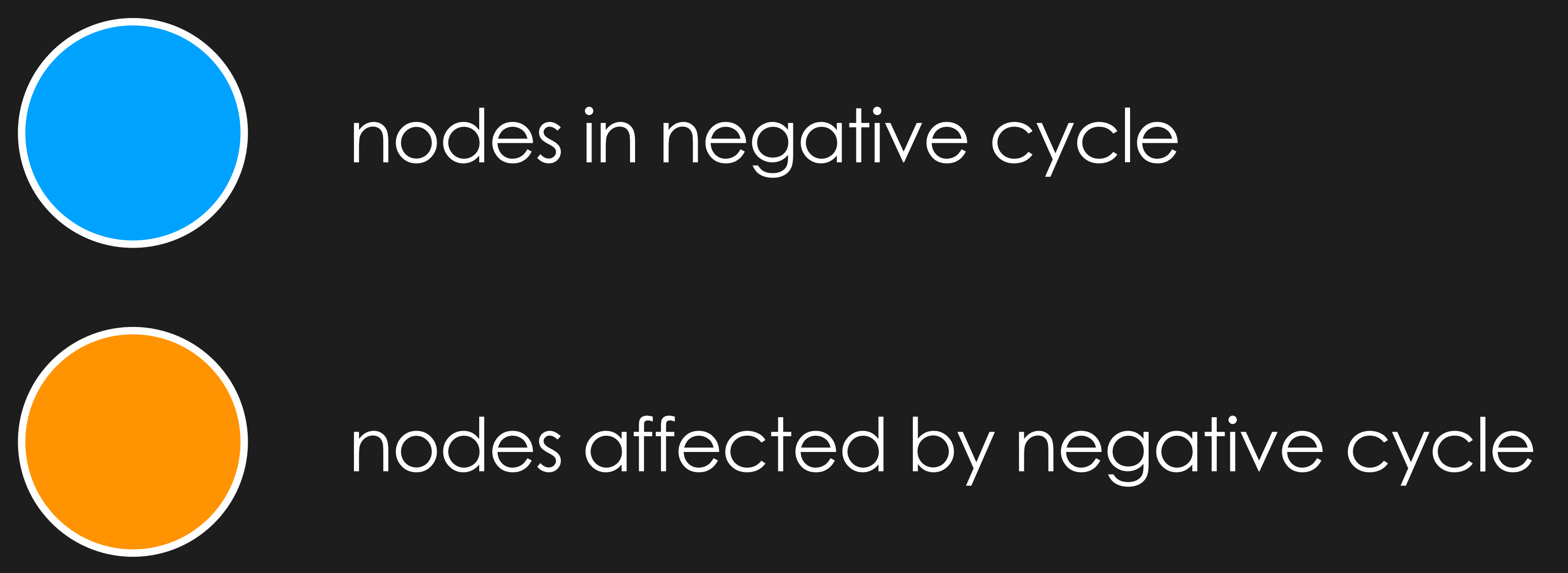


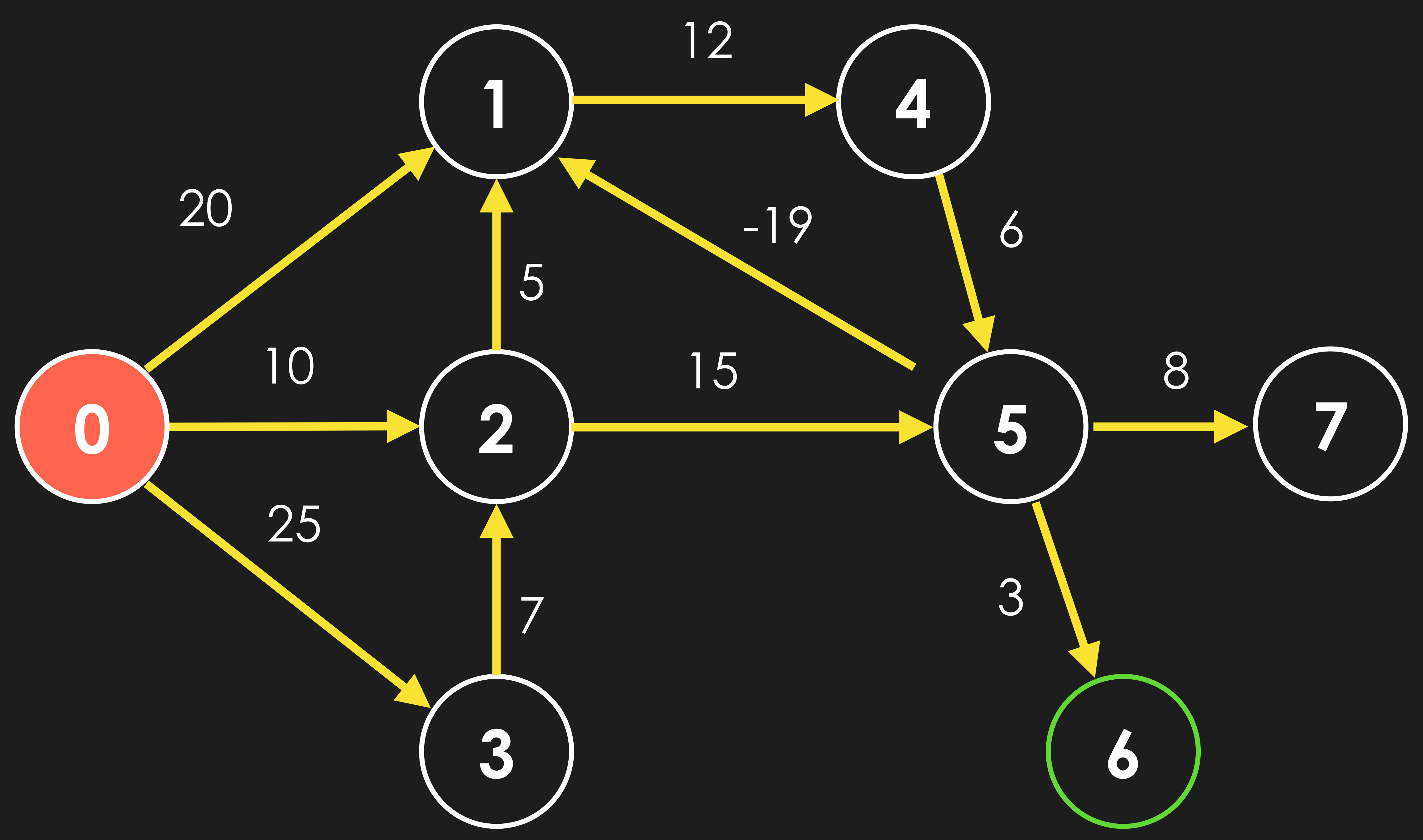
	distTo		edgeTo
0	0	0	-1
1	**11**	1	**5 - 1**
2	10	2	0 - 2
3	25	3	0 - 3
4	32	4	1 - 4
5	25	5	2 - 5
6	**28**	6	**5 - 6**
7	**33**	7	**5 - 7**

vertex 0

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**



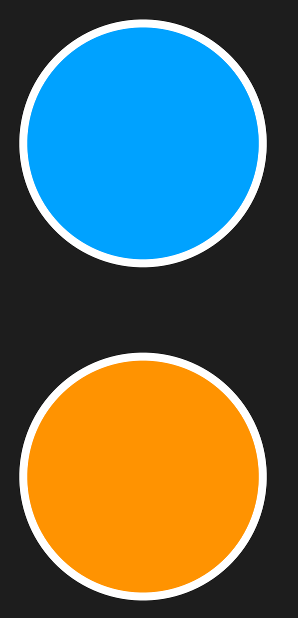


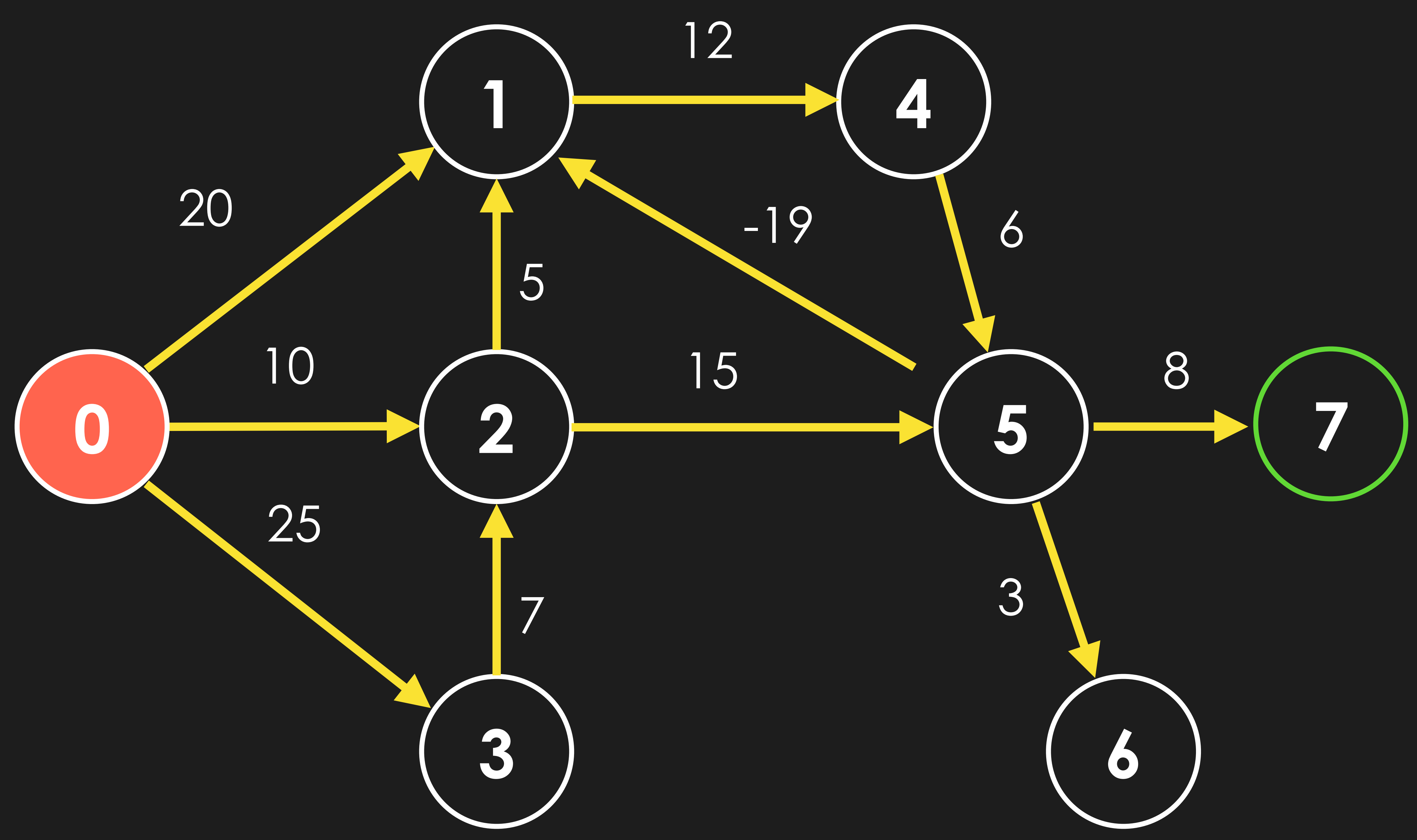
	distTo	edgeTo
0	0	-1
1	11	5 - 1
2	10	0 - 2
3	25	0 - 3
4	32	1 - 4
5	25	2 - 5
6	28	5 - 6
7	33	5 - 7

vertex 0

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**



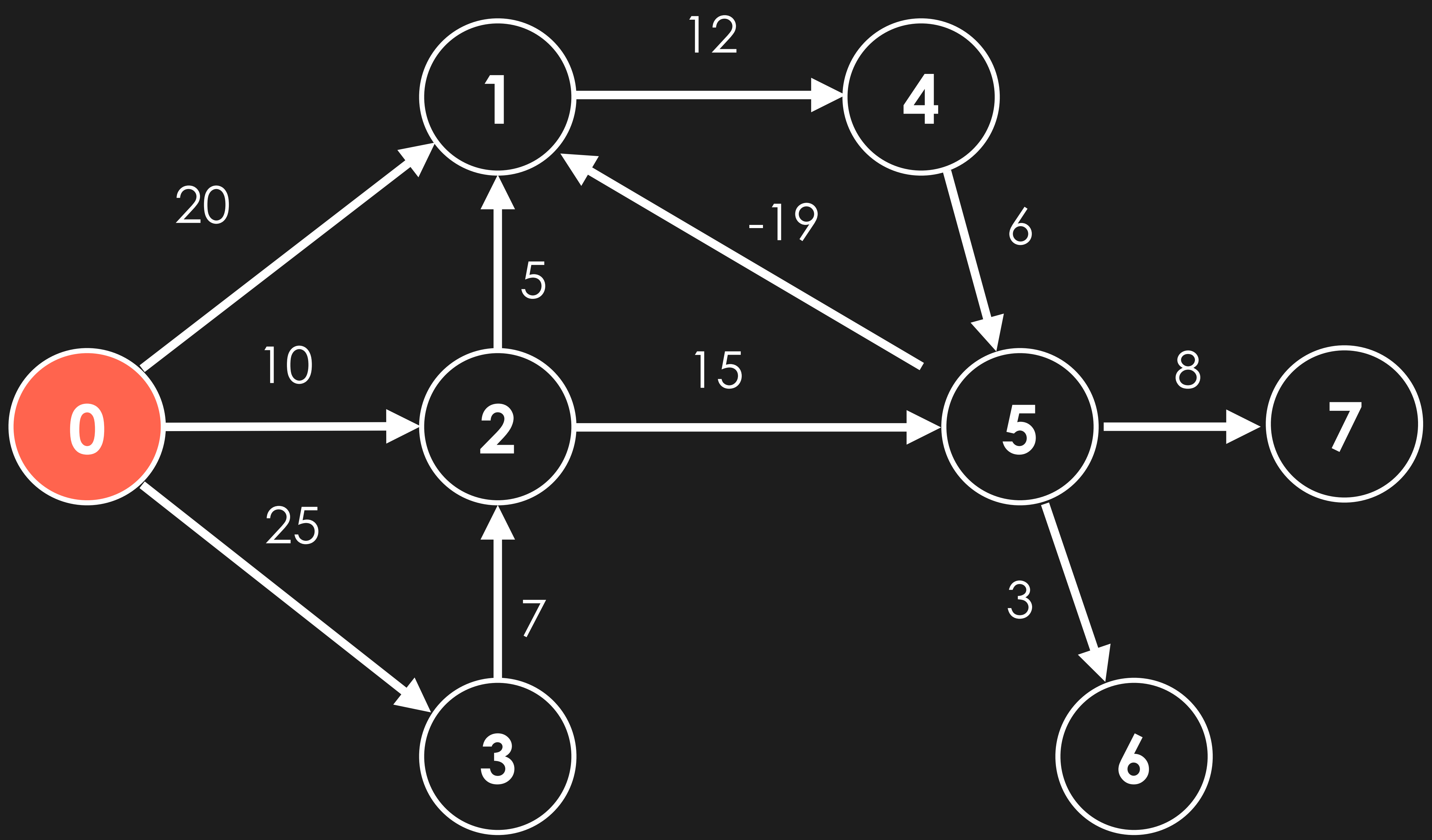


vertex 0

	distTo
0	0
1	11
2	10
3	25
4	32
5	25
6	28
7	33

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	5 - 6
7	5 - 7

1. For each vertex, relax every edge to reach the **shortest path tree**



vertex 1

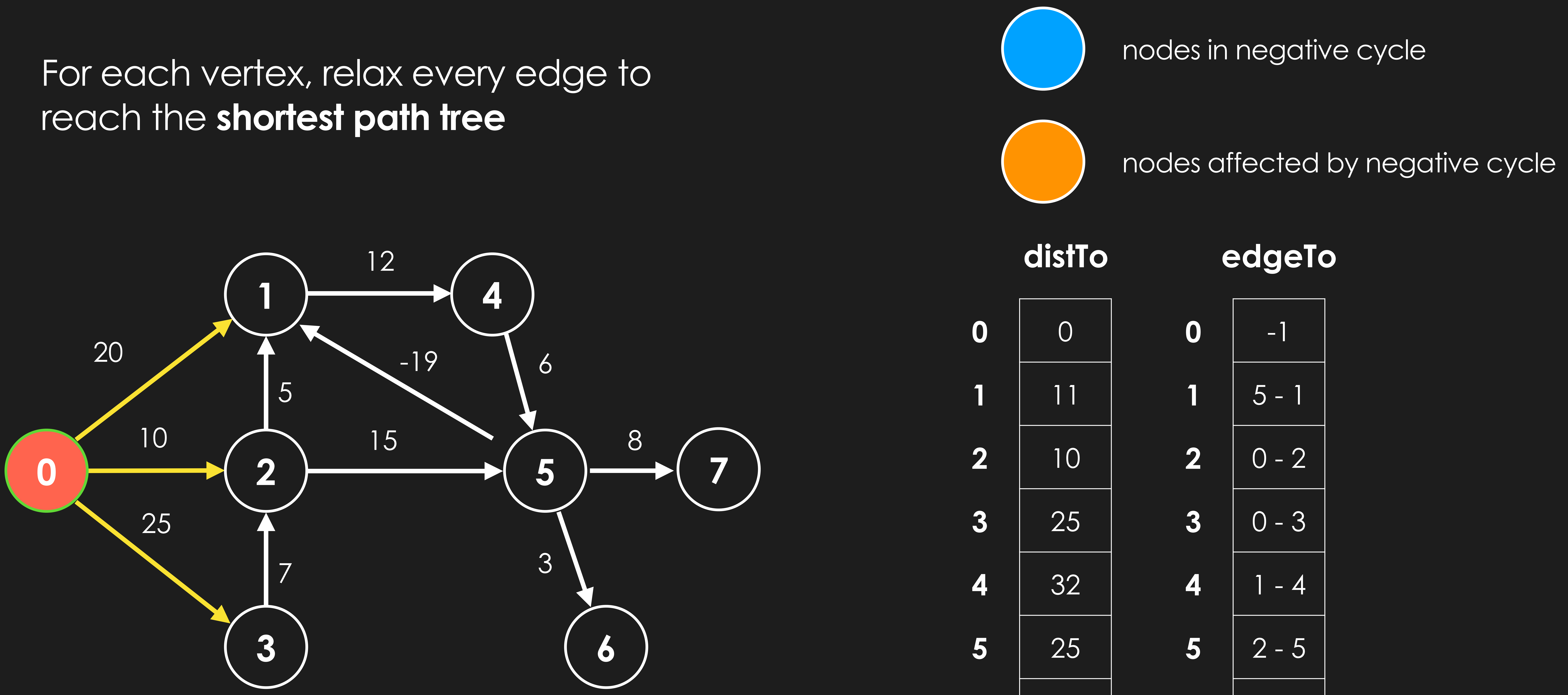
nodes in negative cycle

nodes affected by negative cycle

	distTo
0	0
1	11
2	10
3	25
4	32
5	25
6	28
7	33

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	5 - 6
7	5 - 7

1. For each vertex, relax every edge to reach the **shortest path tree**



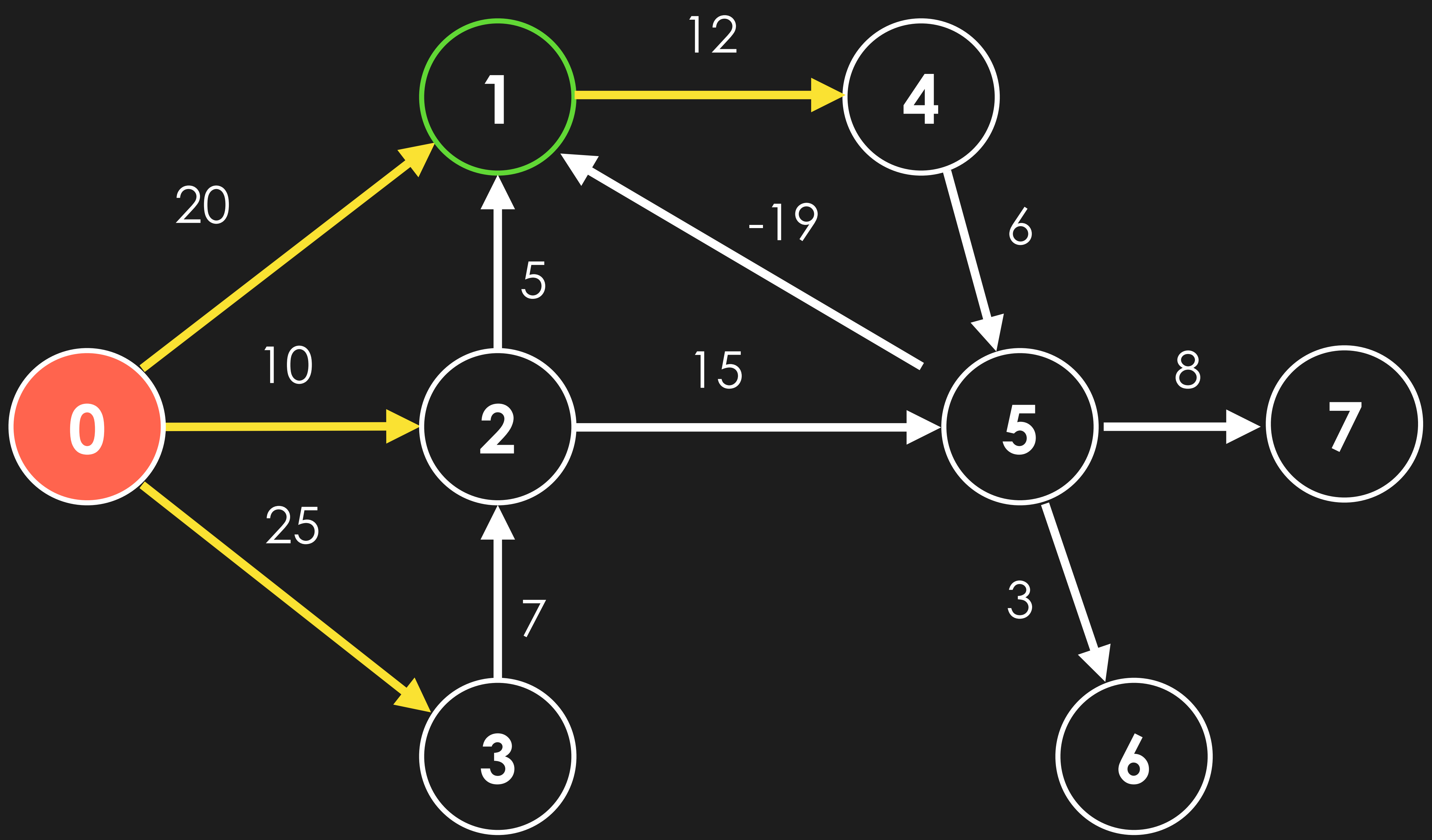
	distTo		edgeTo
0	0	0	-1
1	11	1	5 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	32	4	1 - 4
5	25	5	2 - 5
6	28	6	5 - 6
7	33	7	5 - 7

vertex 1

1. For each vertex, relax every edge to reach the **shortest path tree**

nodes in negative cycle

nodes affected by negative cycle

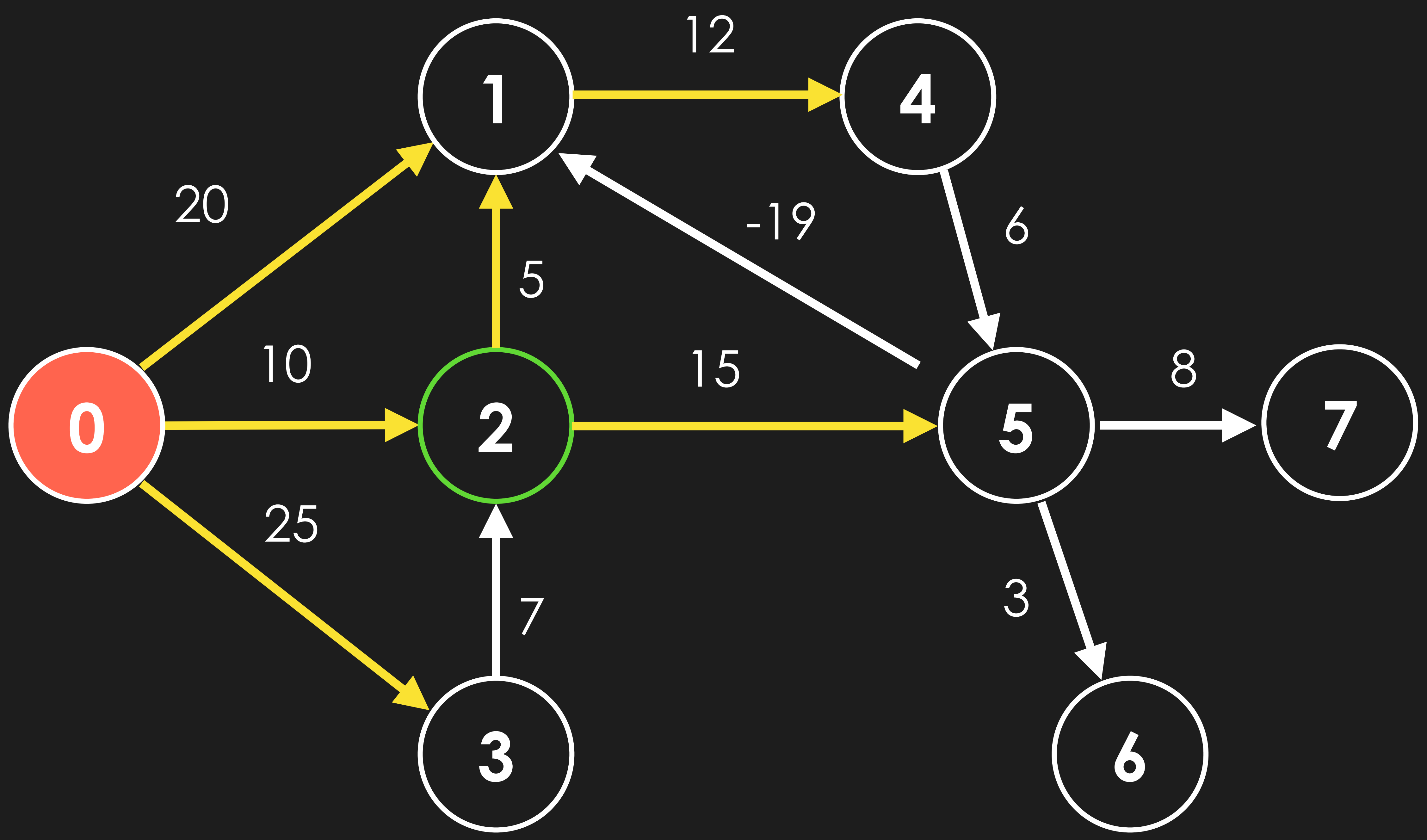


	distTo		edgeTo
0	0	**0**	-1
1	11	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	23	**4**	1 - 4
5	25	**5**	2 - 5
6	28	**6**	5 - 6
7	33	**7**	5 - 7

vertex 1

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**



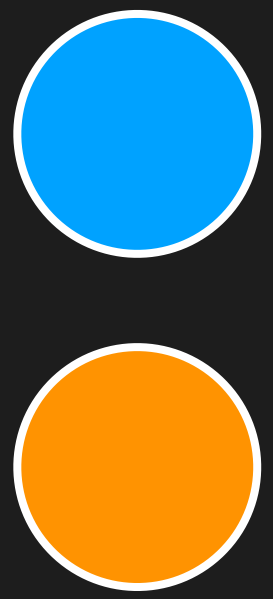
vertex 1

nodes in negative cycle

nodes affected by negative cycle

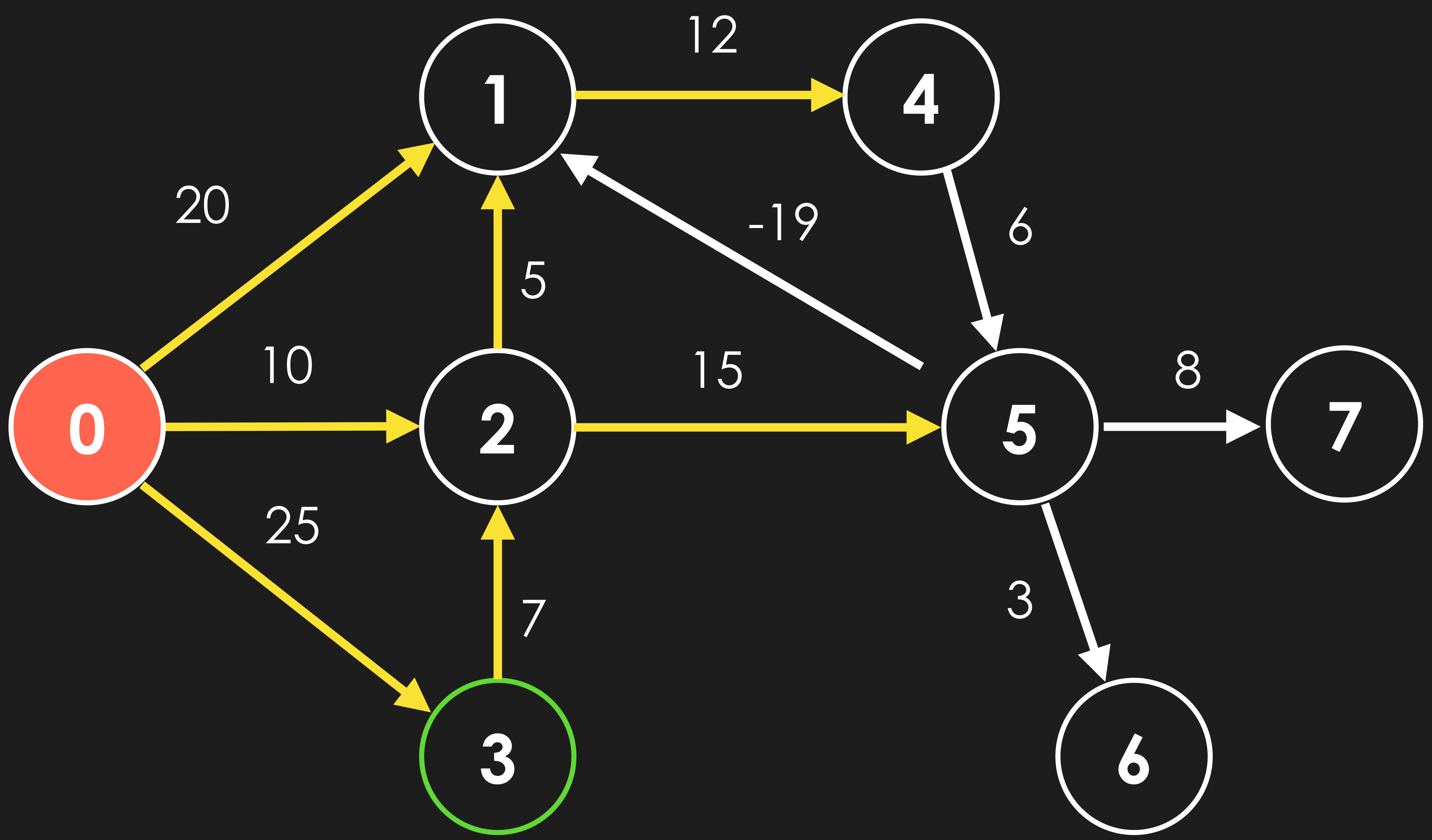
	distTo	edgeTo
0	0	-1
1	11	5 - 1
2	10	0 - 2
3	25	0 - 3
4	23	1 - 4
5	25	2 - 5
6	28	5 - 6
7	33	5 - 7

1. For each vertex, relax every edge to reach the **shortest path tree**



nodes in negative cycle

nodes affected by negative cycle

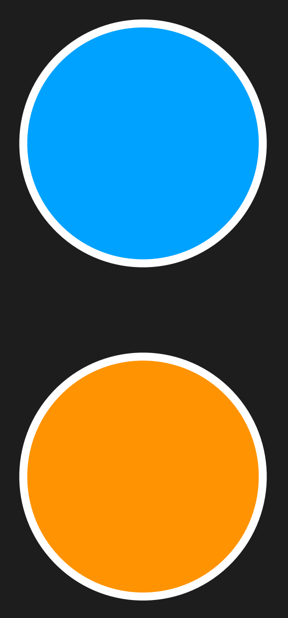


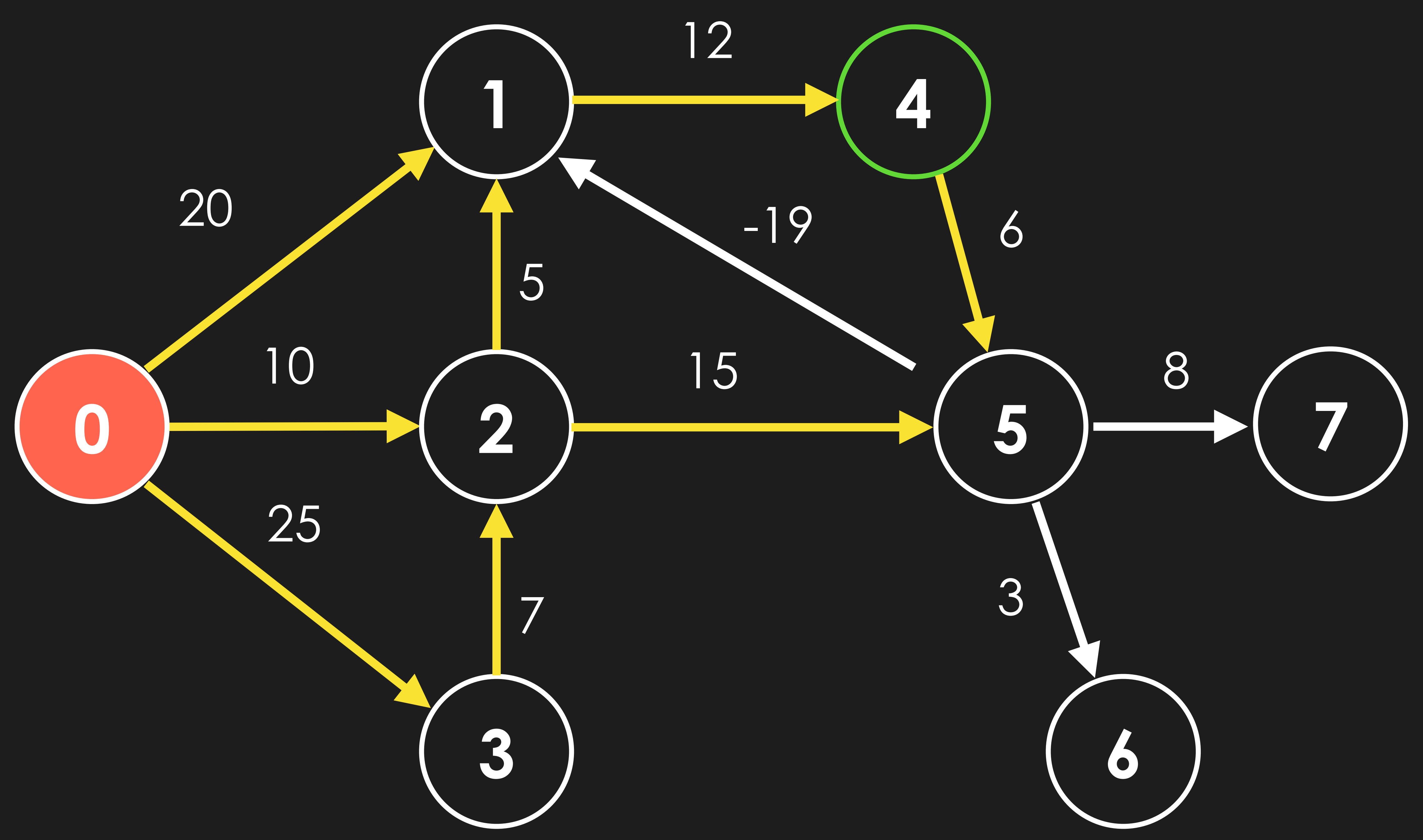
	distTo		edgeTo
0	0	**0**	-1
1	11	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	23	**4**	1 - 4
5	25	**5**	2 - 5
6	28	**6**	5 - 6
7	33	**7**	5 - 7

vertex 1

CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**

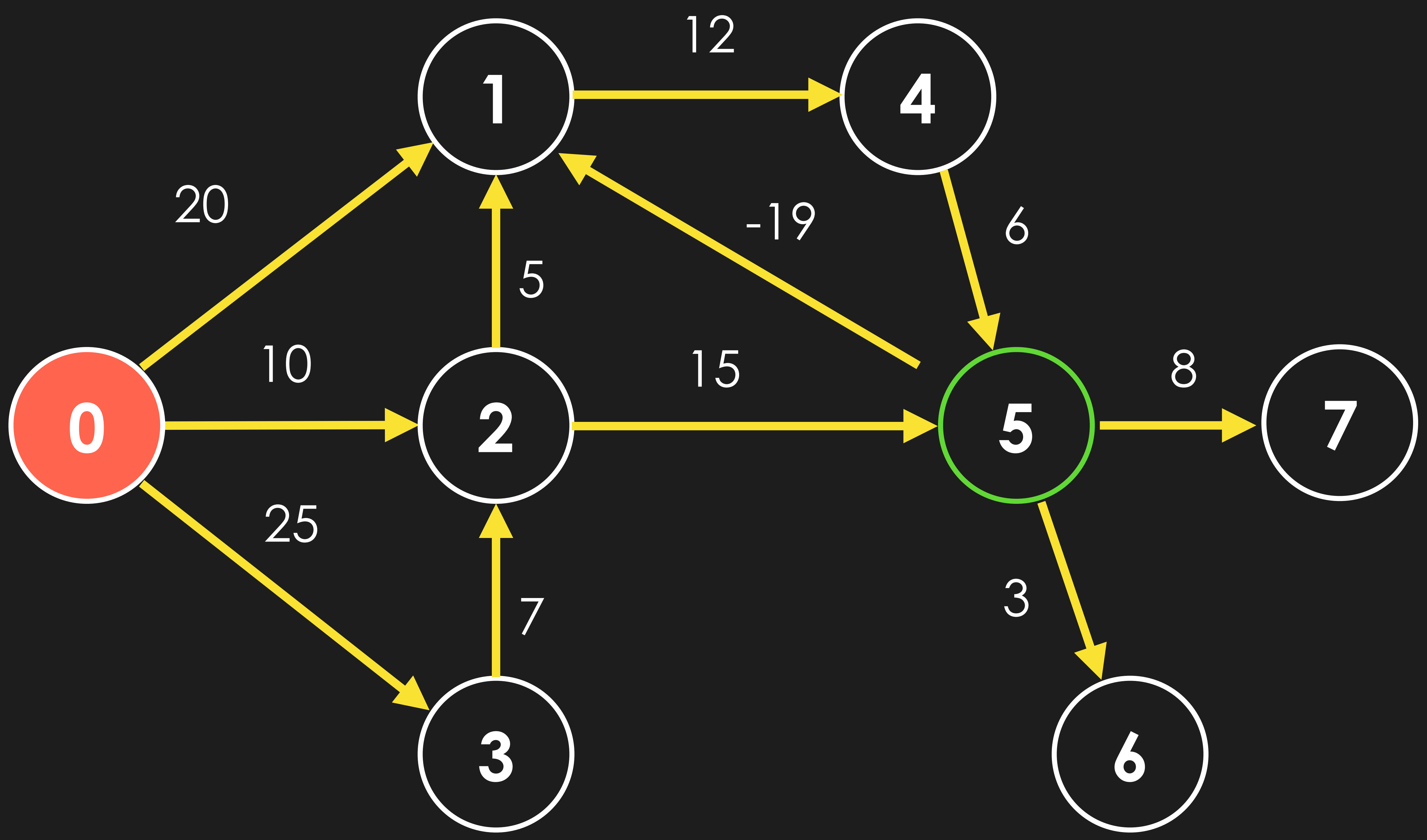




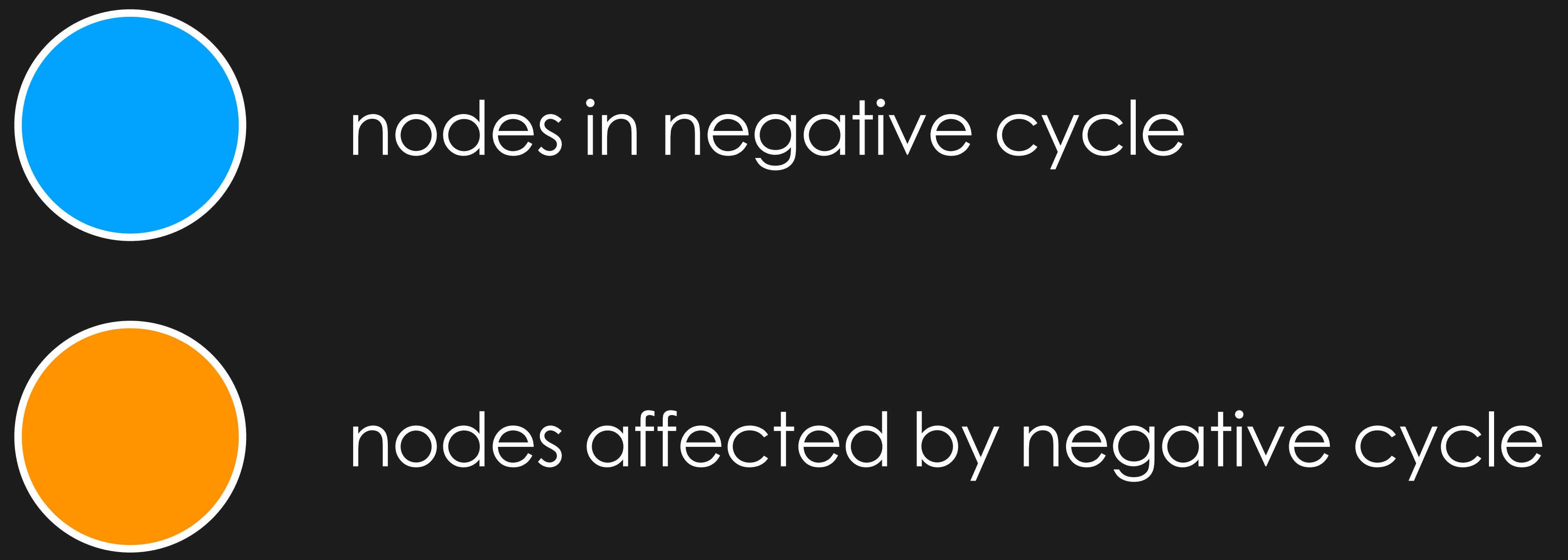
	distTo		edgeTo
0	0	**0**	-1
1	11	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	23	**4**	1 - 4
5	25	**5**	2 - 5
6	28	**6**	5 - 6
7	33	**7**	5 - 7

vertex 1

1. For each vertex, relax every edge to reach the **shortest path tree**

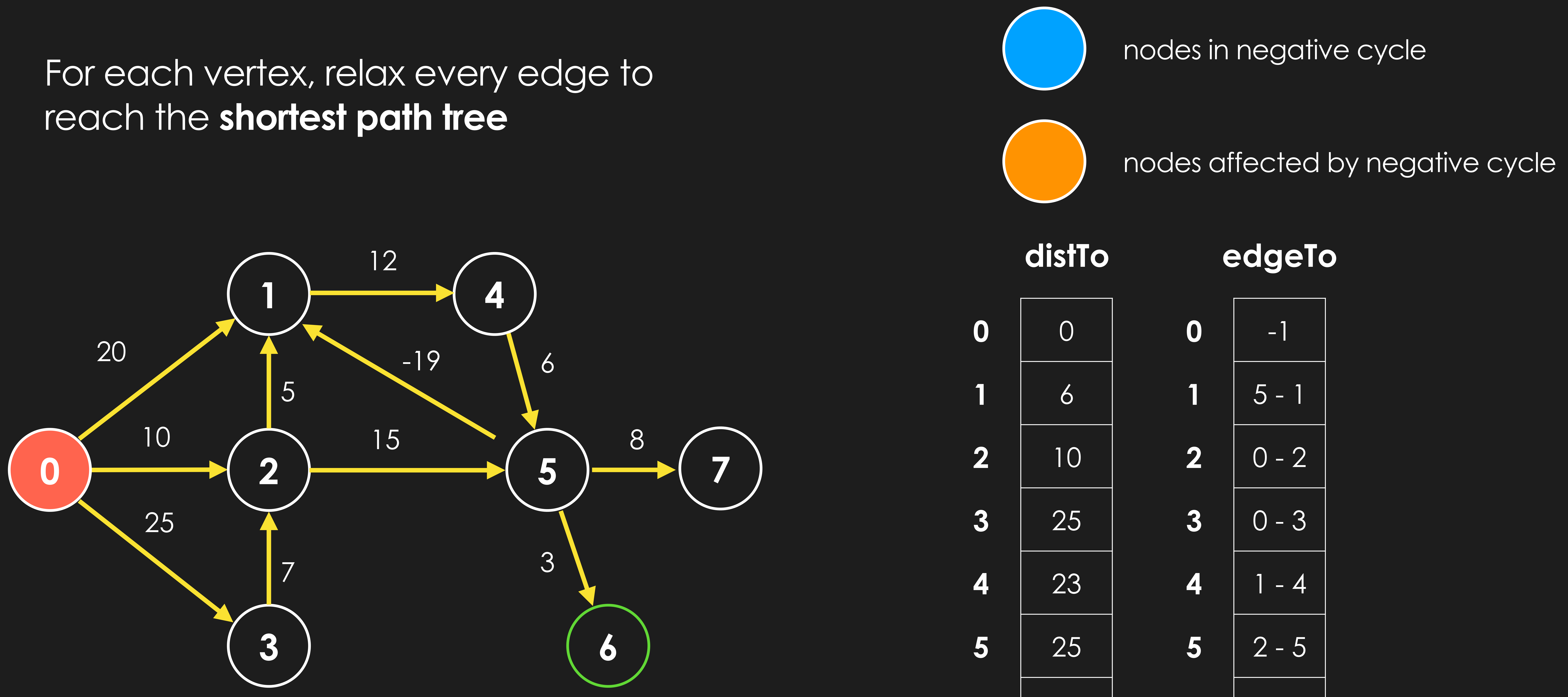


vertex 1



	distTo		edgeTo
0	0	**0**	-1
1	6	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	23	**4**	1 - 4
5	25	**5**	2 - 5
6	28	**6**	5 - 6
7	33	**7**	5 - 7

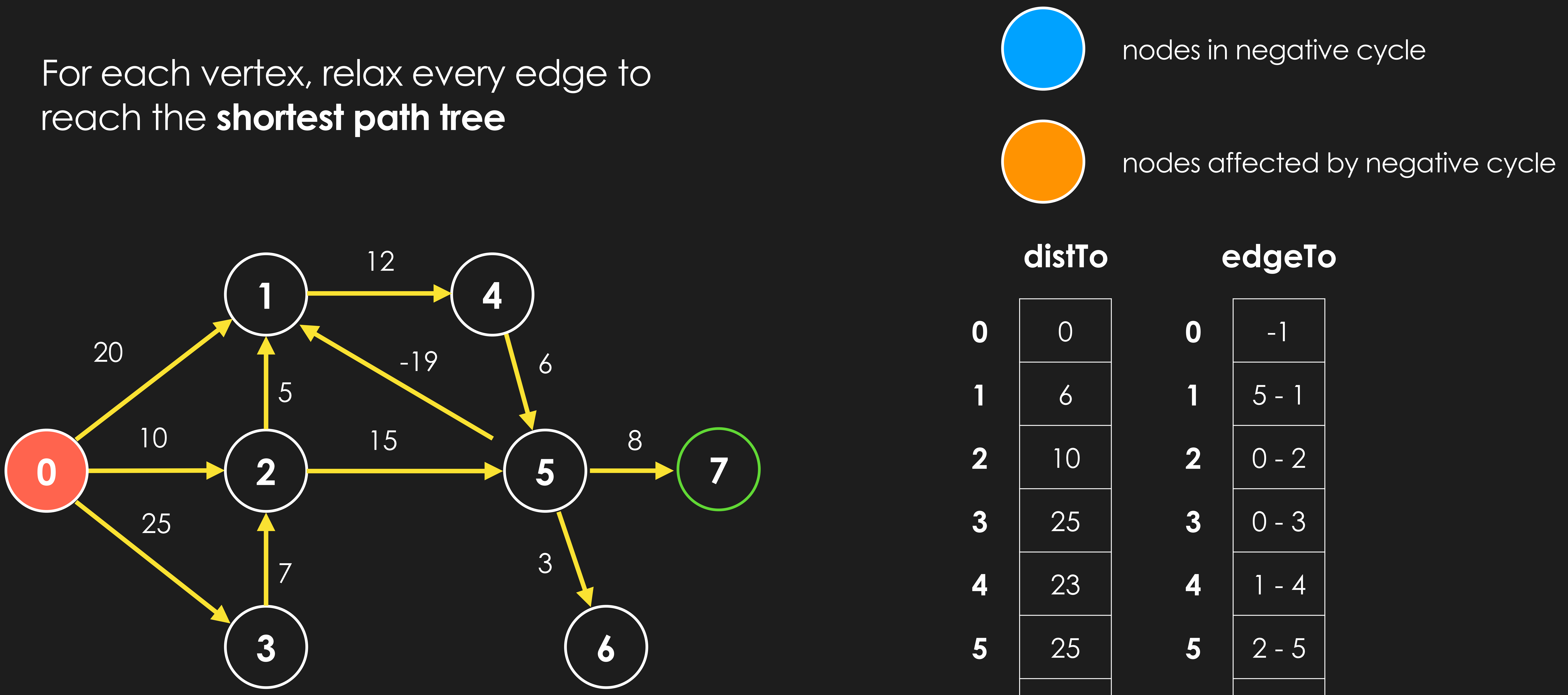
1. For each vertex, relax every edge to reach the **shortest path tree**



	distTo		edgeTo
0	0	**0**	-1
1	6	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	23	**4**	1 - 4
5	25	**5**	2 - 5
6	28	**6**	5 - 6
7	33	**7**	5 - 7

vertex 1

1. For each vertex, relax every edge to reach the **shortest path tree**

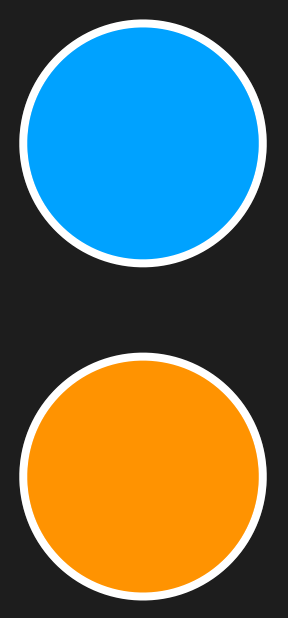


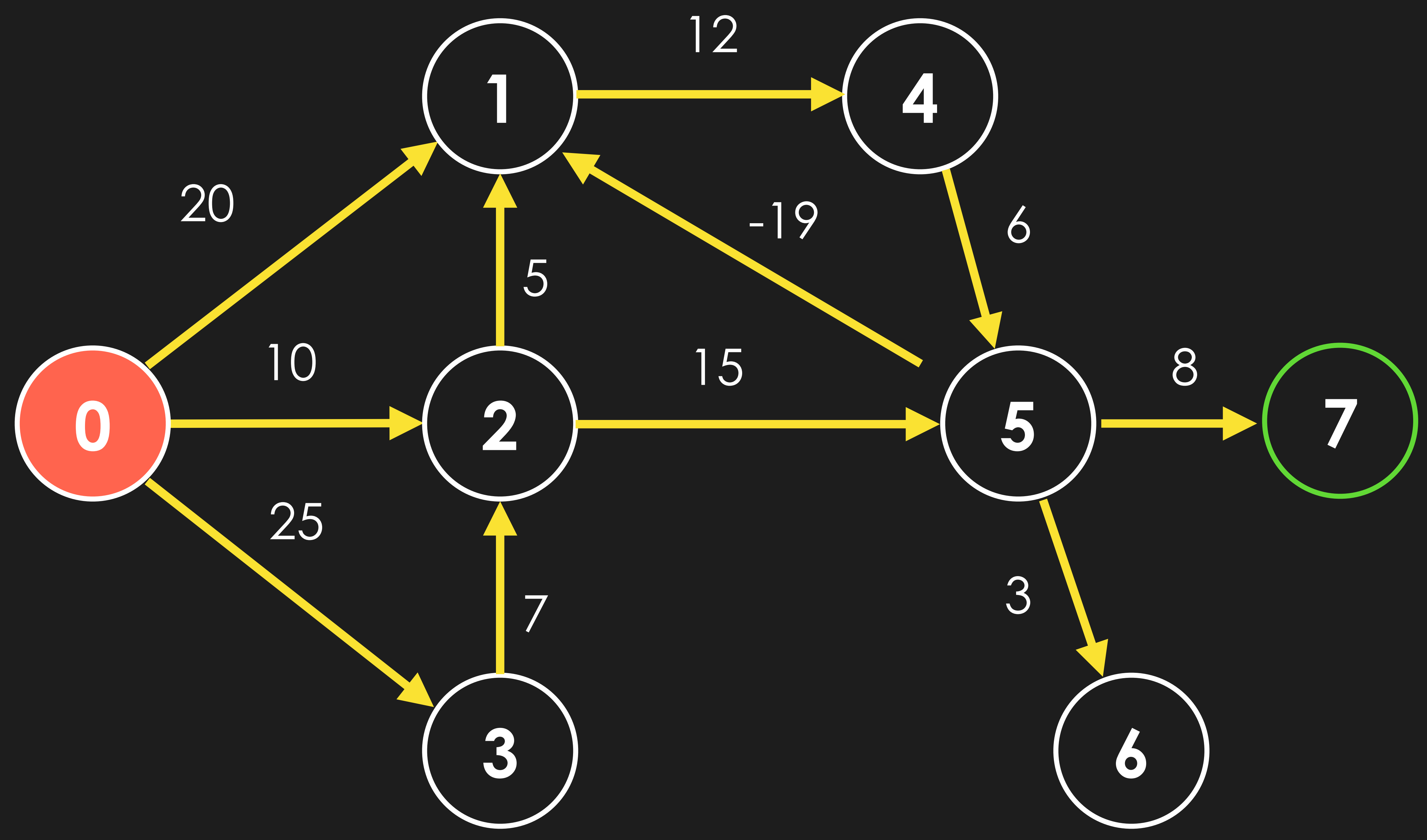
	distTo
0	0
1	6
2	10
3	25
4	23
5	25
6	28
7	33

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	2 - 5
6	5 - 6
7	5 - 7

vertex 1

1. For each vertex, relax every edge to reach the **shortest path tree**

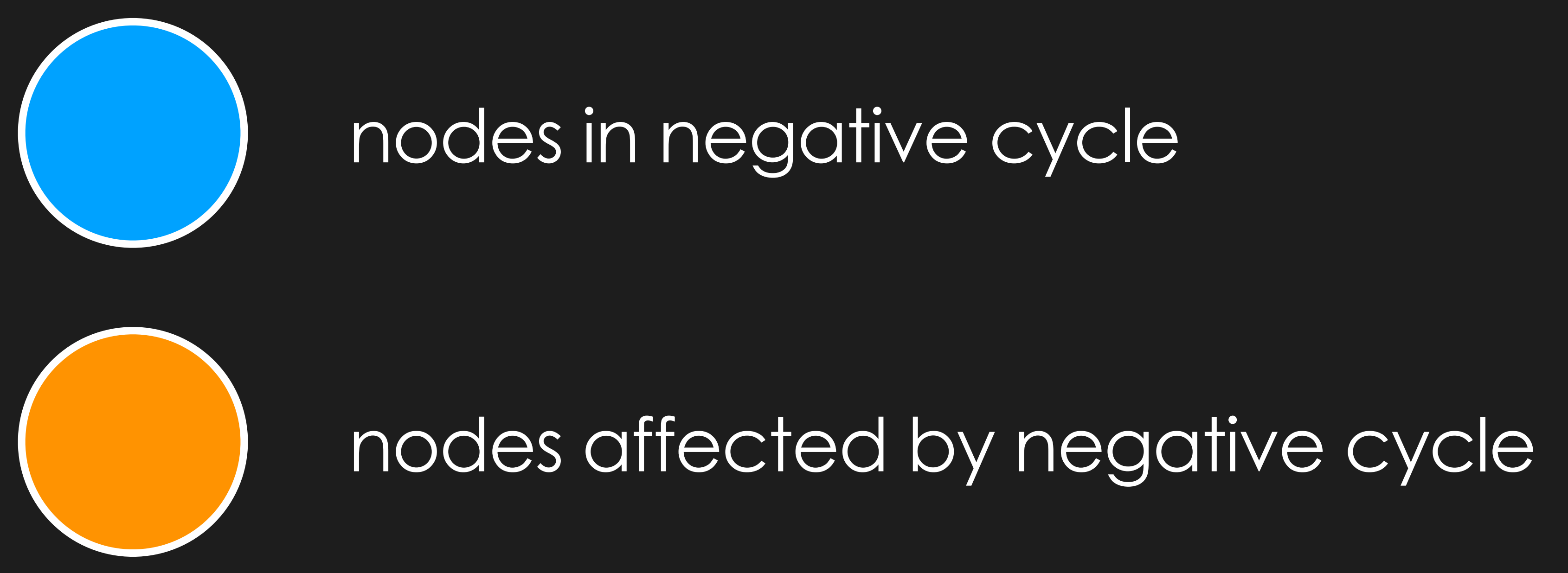


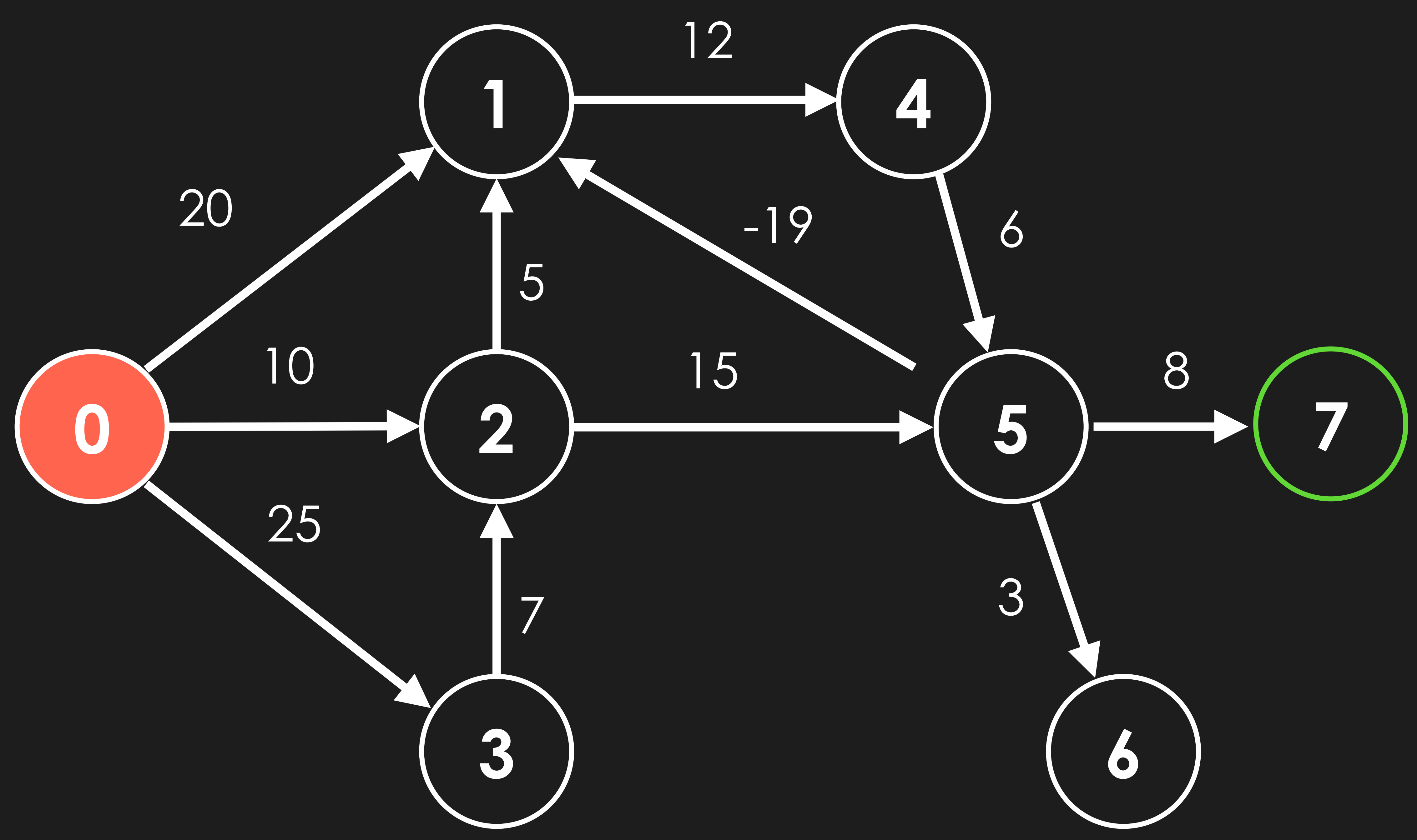


	distTo	edgeTo
0	0	-1
1	**-1**	5 - 1
2	10	0 - 2
3	25	0 - 3
4	**12**	1 - 4
5	**18**	**4 - 5**
6	**21**	5 - 6
7	**26**	5 - 7

Fast Forward: End of Vertex 7

1. For each vertex, relax every edge to reach the **shortest path tree**





	distTo		edgeTo
0	0	0	-1
1	-1	1	5 - 1
2	10	2	0 - 2
3	25	3	0 - 3
4	12	4	1 - 4
5	18	5	4 - 5
6	21	6	5 - 6
7	26	7	5 - 7

Fast Forward: End of Vertex 7

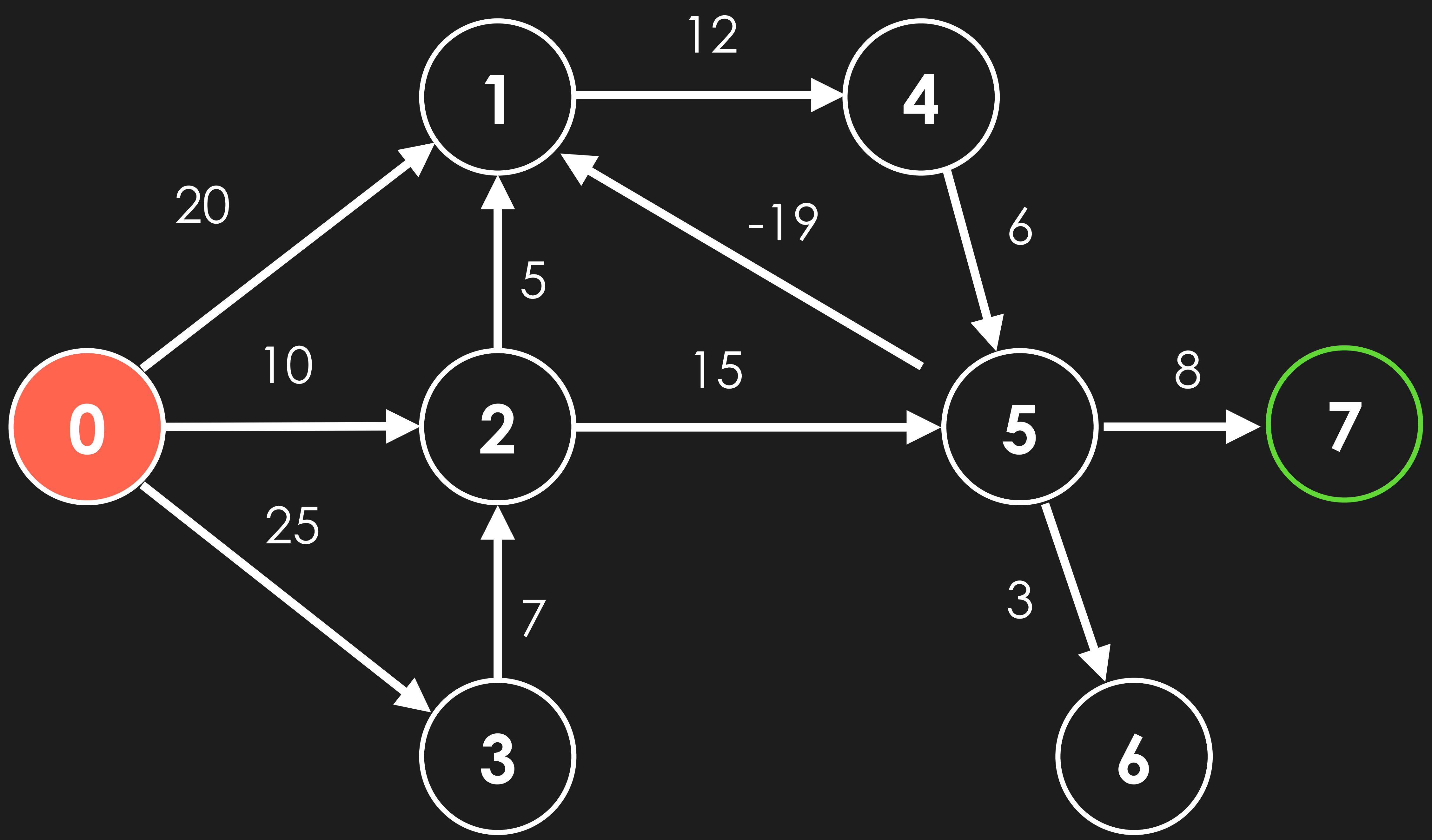
CodeIT

1. For each vertex, relax every edge to reach the **shortest path tree**

At this point, we can export **edgeTo** to get the **shortest path tree**

nodes in negative cycle

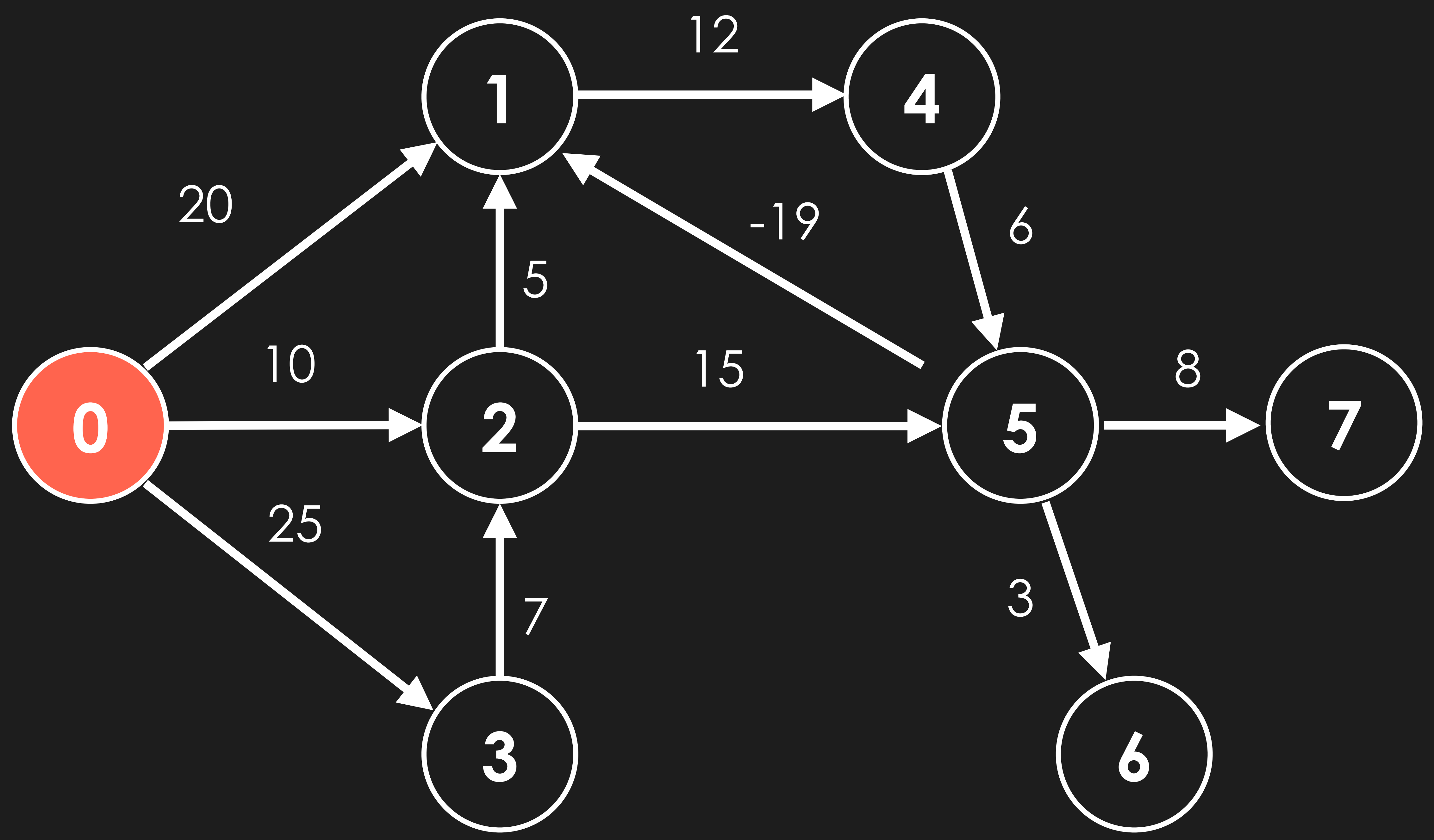
nodes affected by negative cycle



	distTo	edgeTo
0	0	-1
1	**-1**	5 - 1
2	10	0 - 2
3	25	0 - 3
4	**12**	1 - 4
5	**18**	4 - 5
6	**21**	5 - 6
7	**26**	5 - 7

Fast Forward: End of Vertex 7

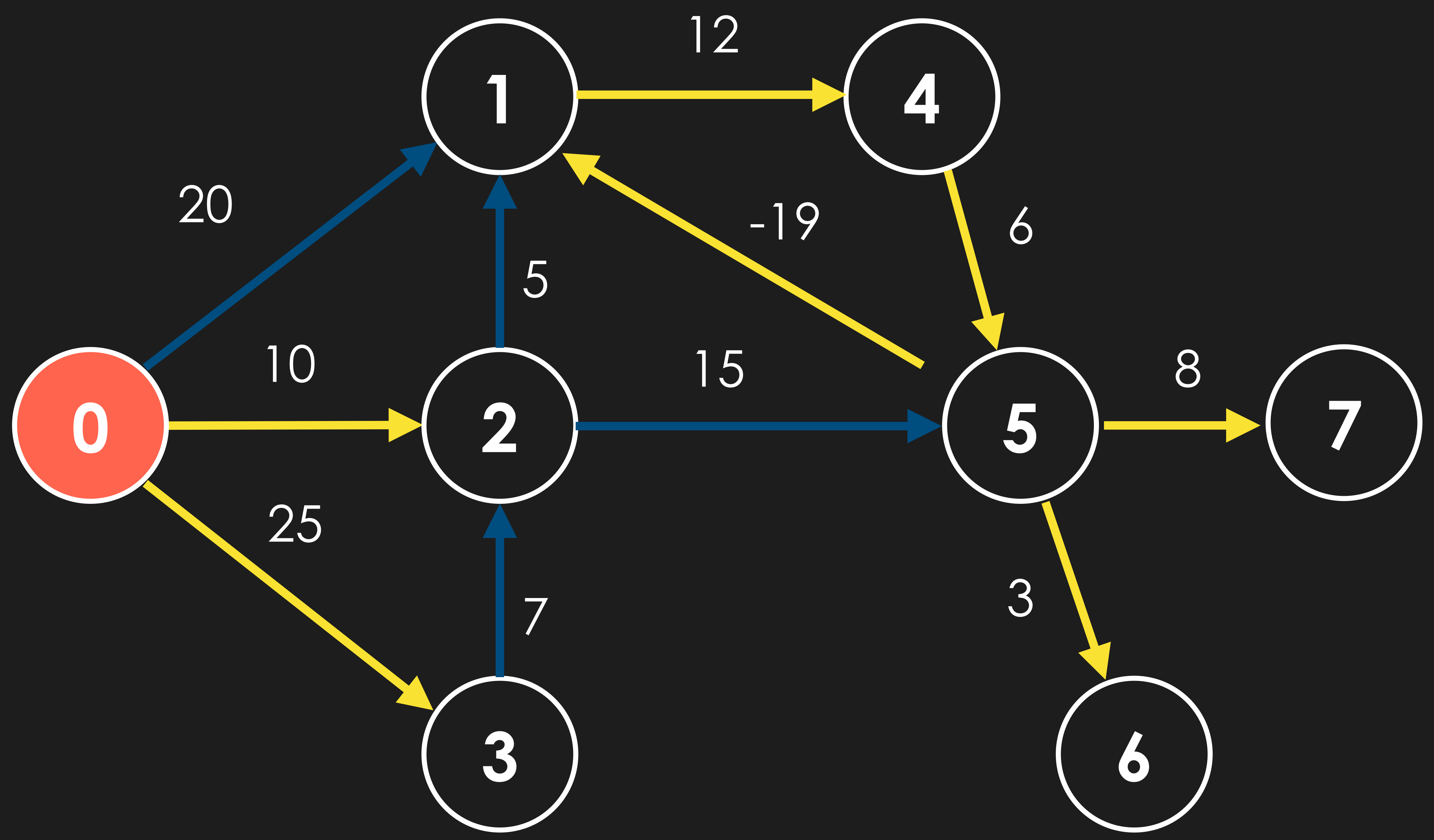
At this point, we can export **edgeTo** to get the **shortest path tree**



edgeTo

0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

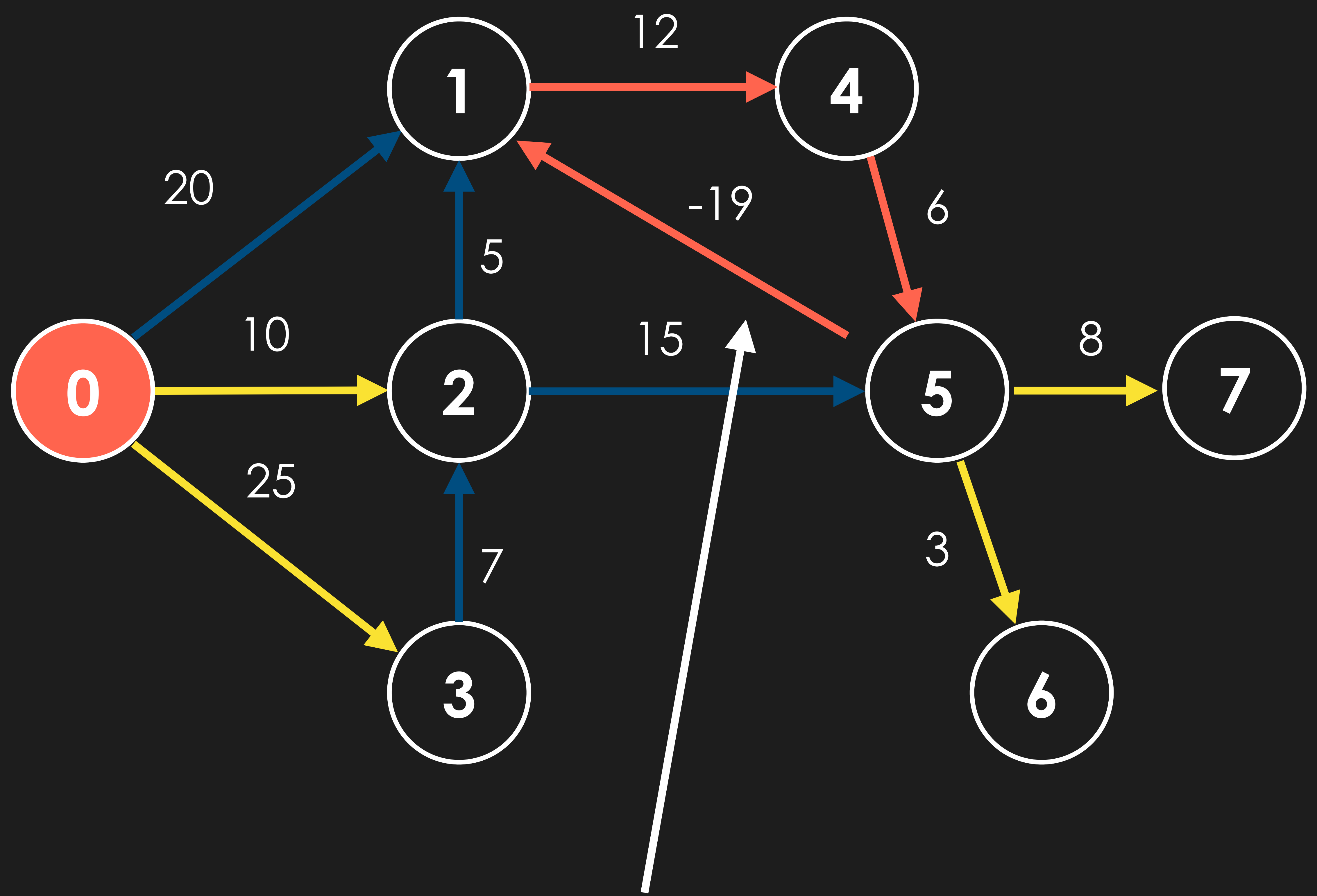
At this point, we can export **edgeTo** to get the **shortest path tree**



edgeTo

0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

At this point, we can export **edgeTo** to get the **shortest path tree**



edgeTo

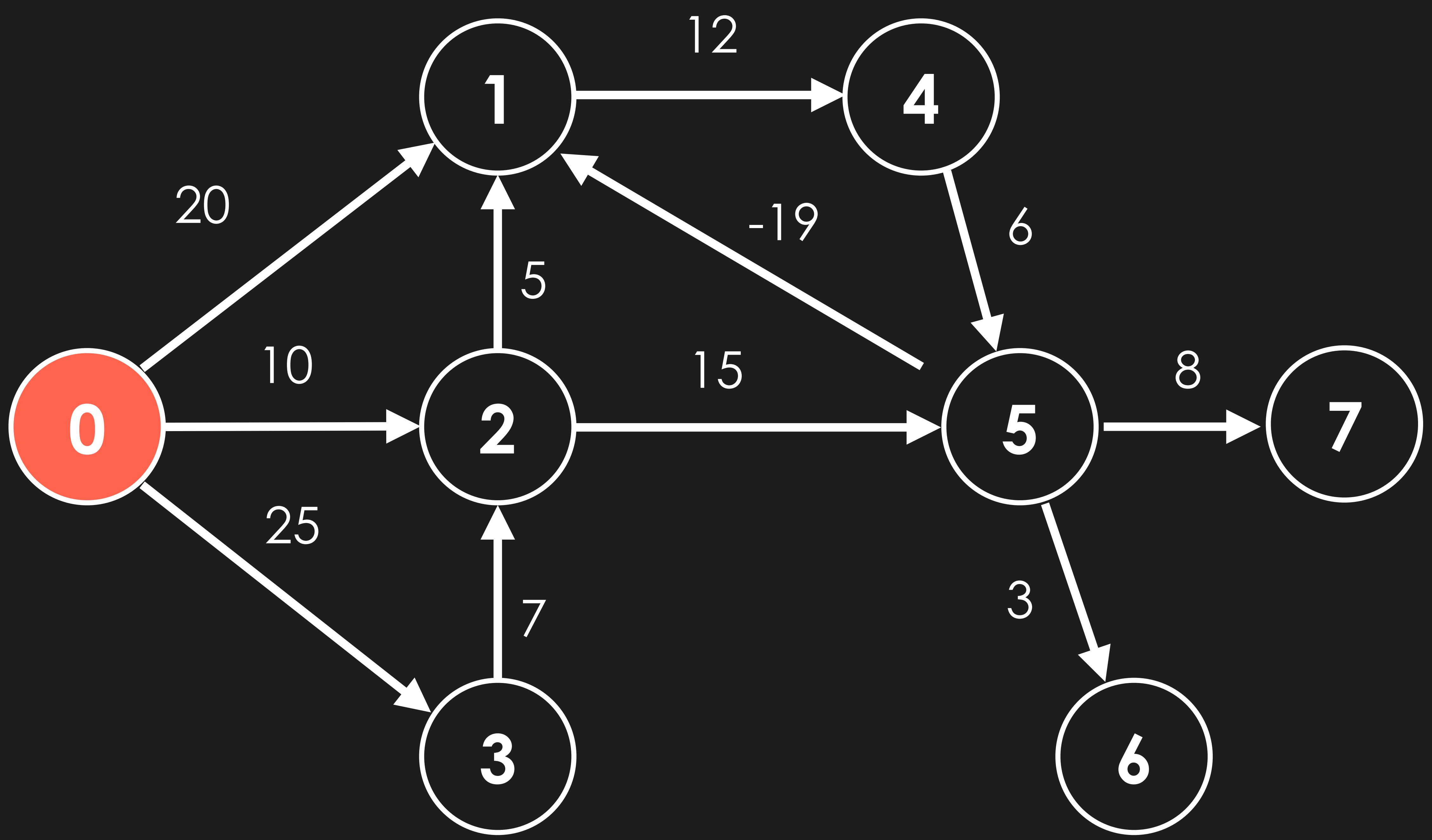
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

Note: If a negative cycle exists, it will be part of the **SPT**

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

nodes in negative cycle

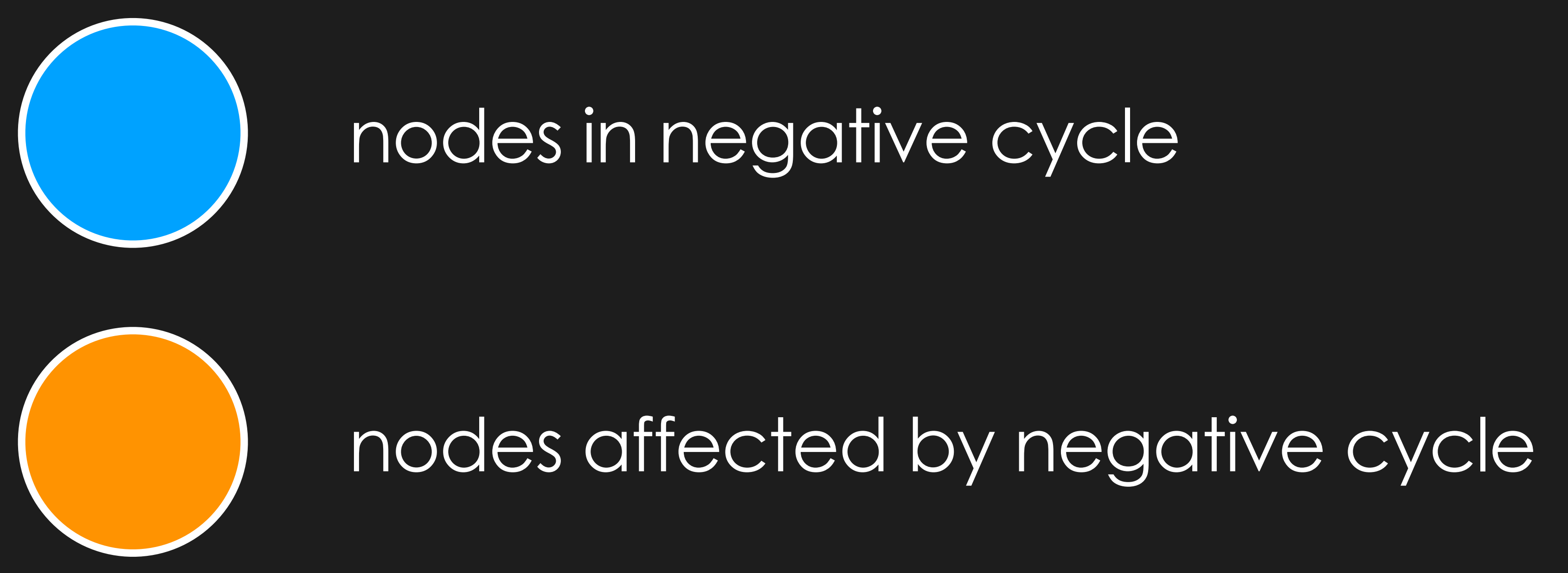
nodes affected by negative cycle

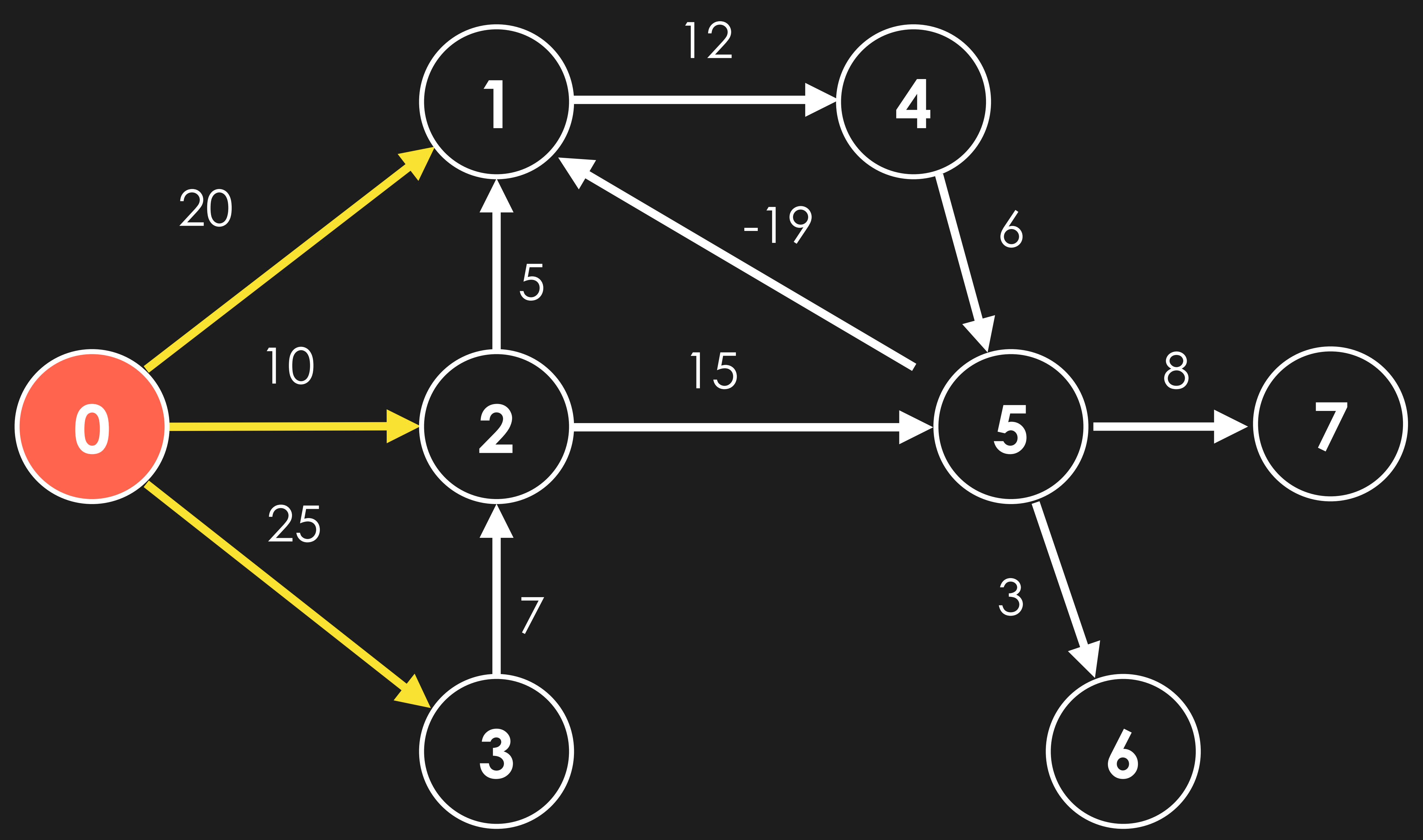


	distTo	edgeTo
0	0	-1
1	-1	5 - 1
2	10	0 - 2
3	25	0 - 3
4	12	1 - 4
5	18	4 - 5
6	21	5 - 6
7	26	5 - 7

CodeIT

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**





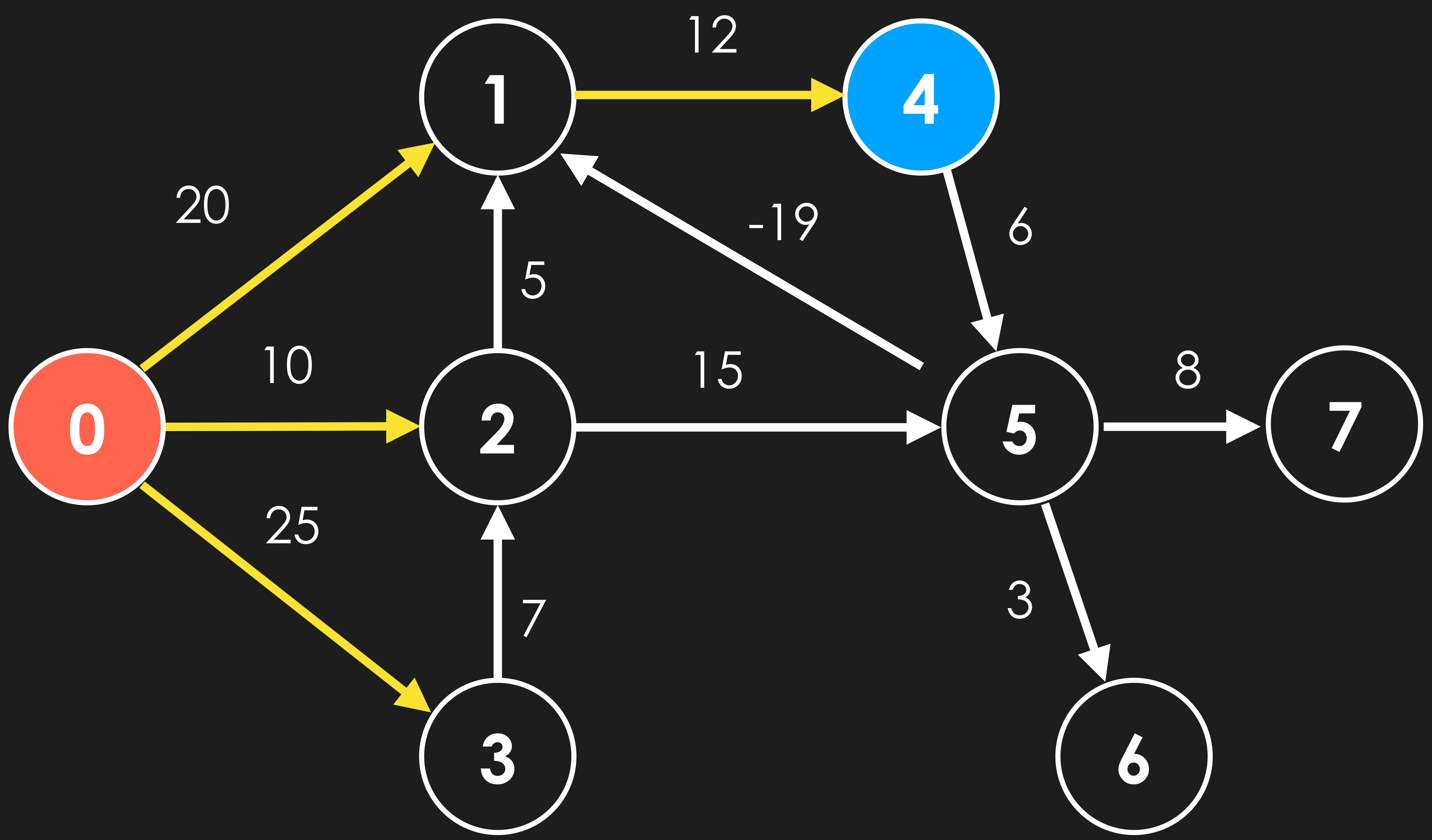
	distTo
0	0
1	-1
2	10
3	25
4	12
5	18
6	21
7	26

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

nodes in negative cycle

nodes affected by negative cycle



	distTo
0	0
1	-1
2	10
3	25
4	11
5	18
6	21
7	26

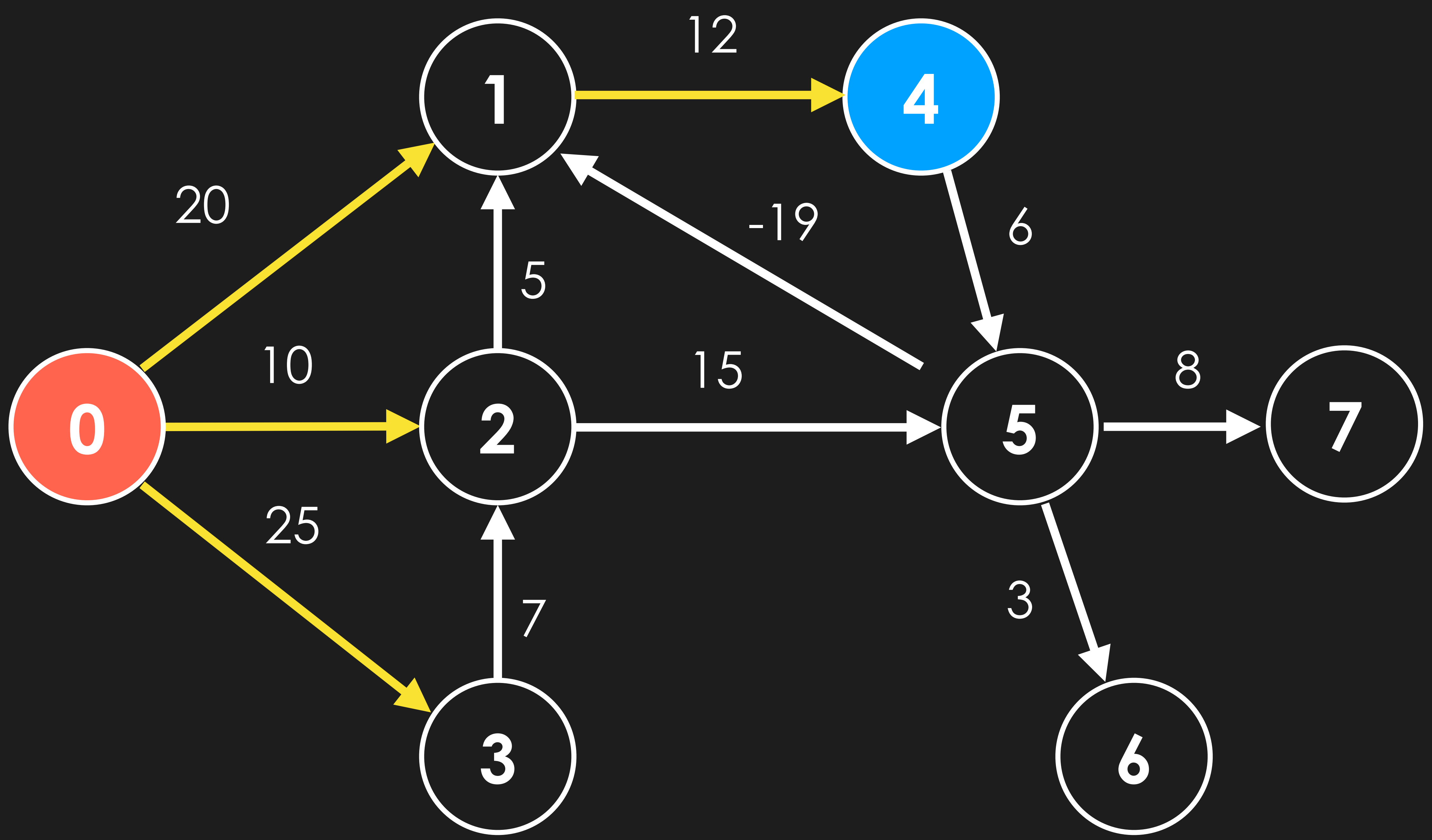
	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

CodeIT

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

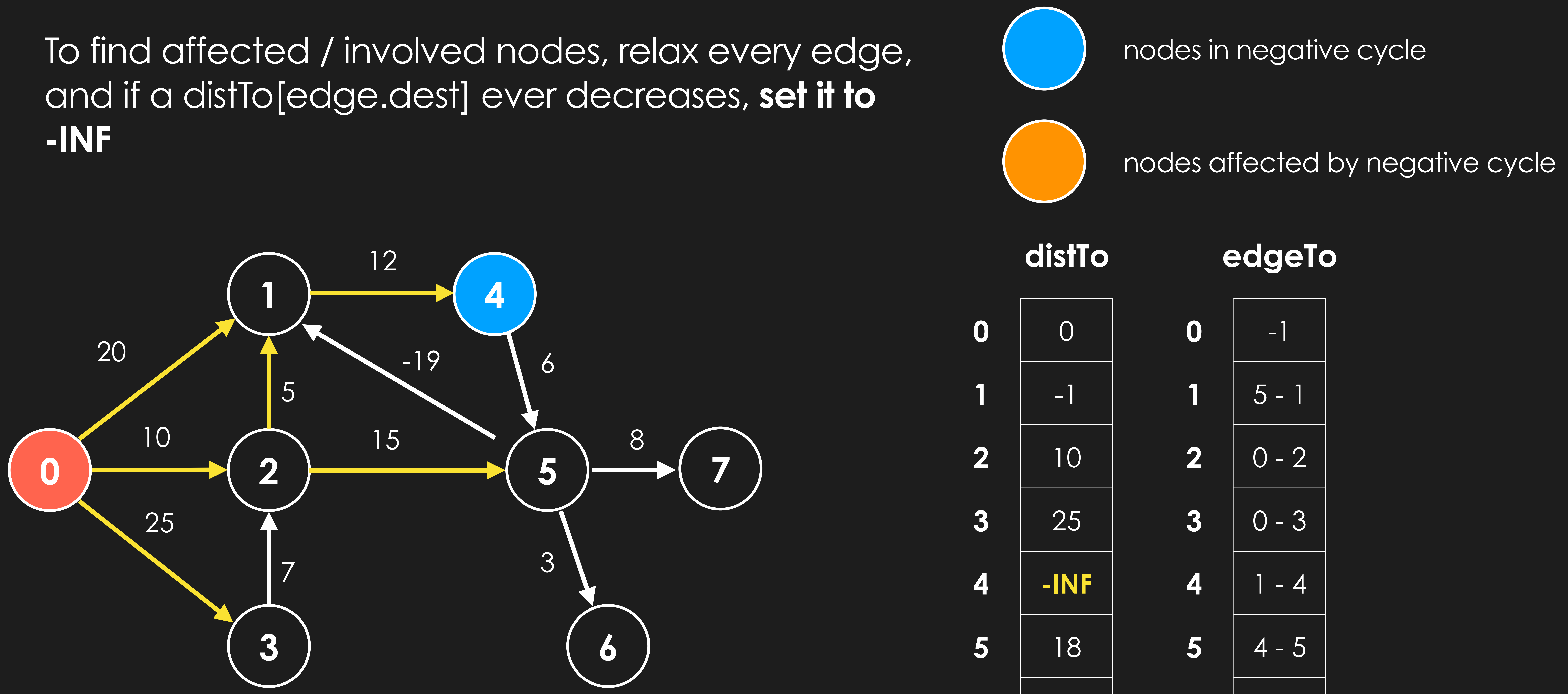
nodes in negative cycle

nodes affected by negative cycle



	distTo	edgeTo
0	0	-1
1	-1	5 - 1
2	10	0 - 2
3	25	0 - 3
4	-INF	1 - 4
5	18	4 - 5
6	21	5 - 6
7	26	5 - 7

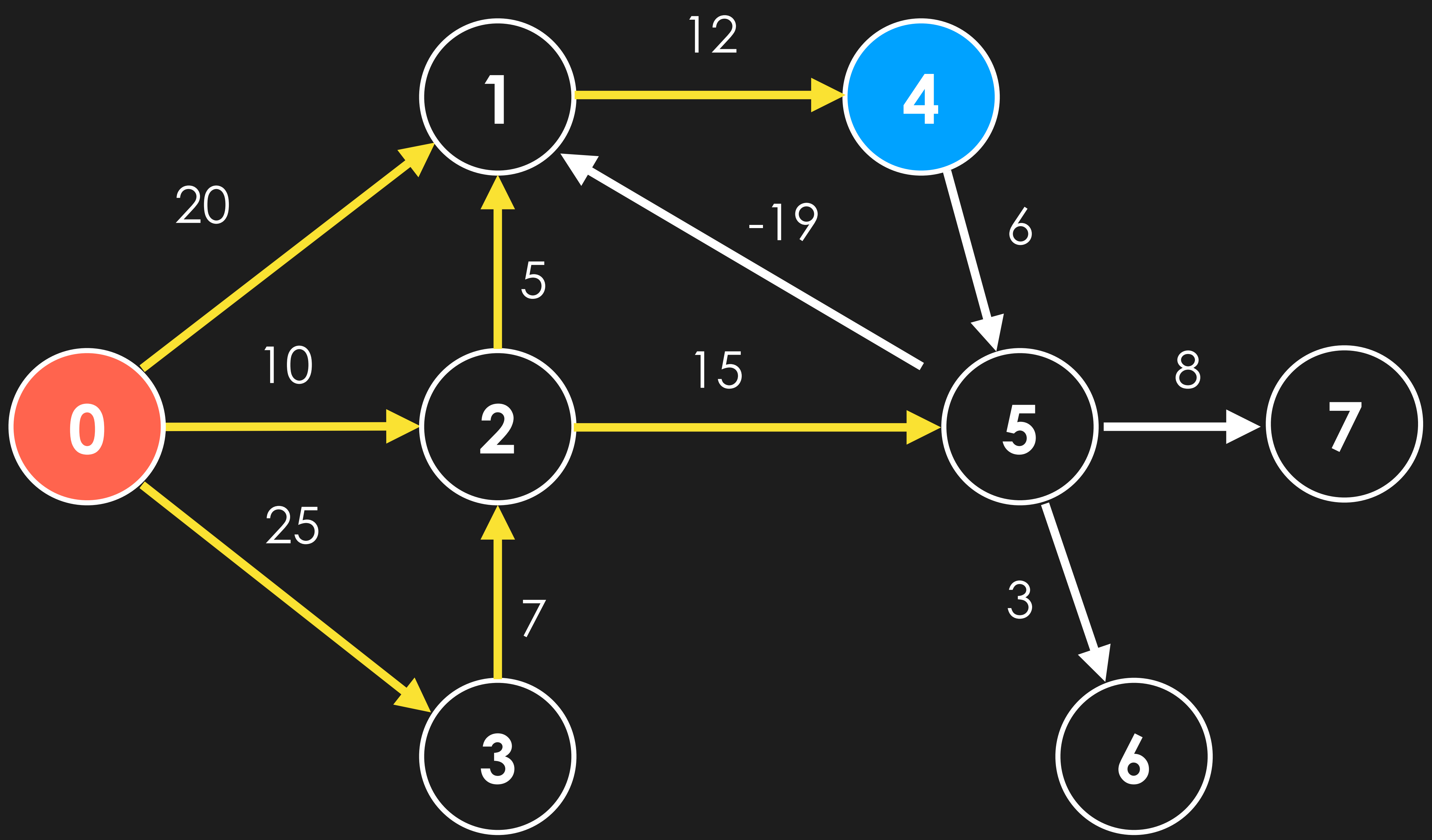
2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

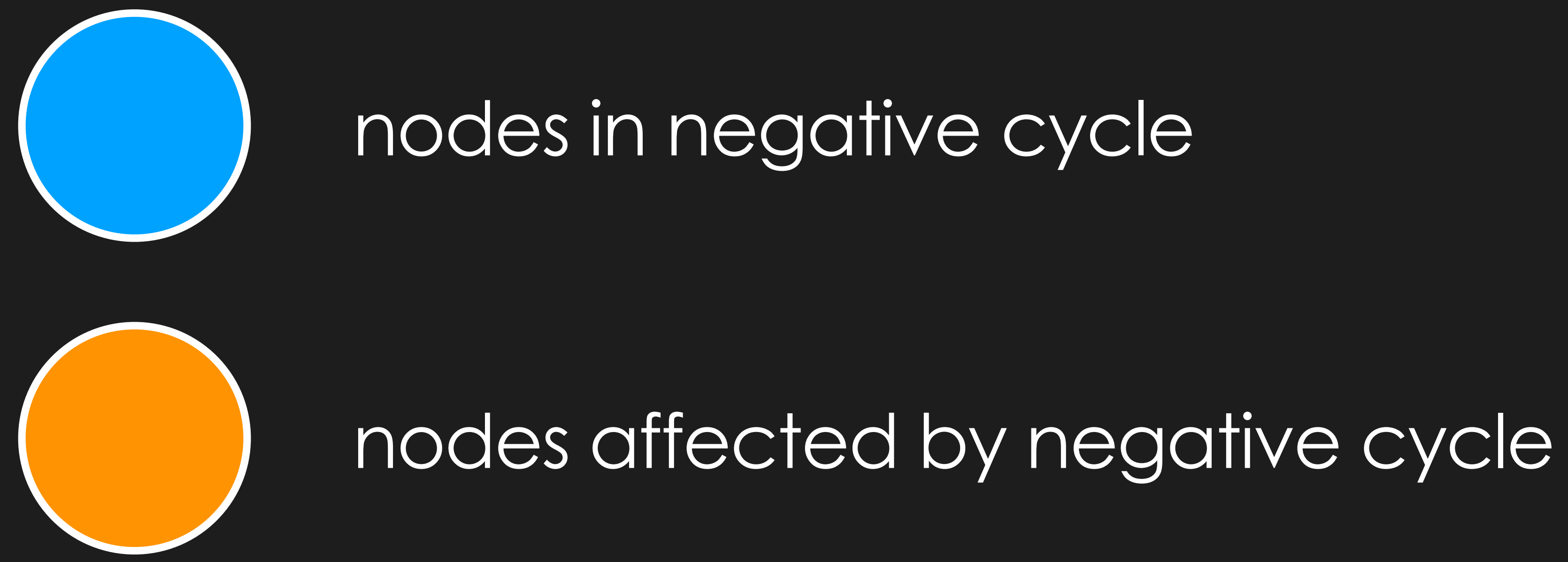


	distTo
0	0
1	-1
2	10
3	25
4	-INF
5	18
6	21
7	26

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**



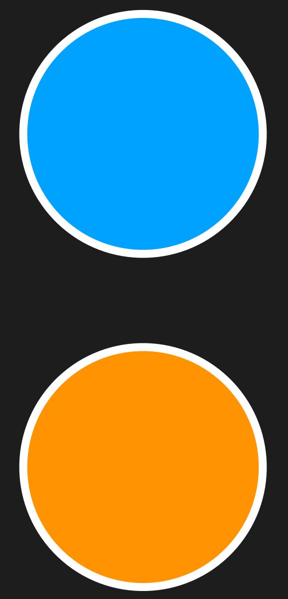


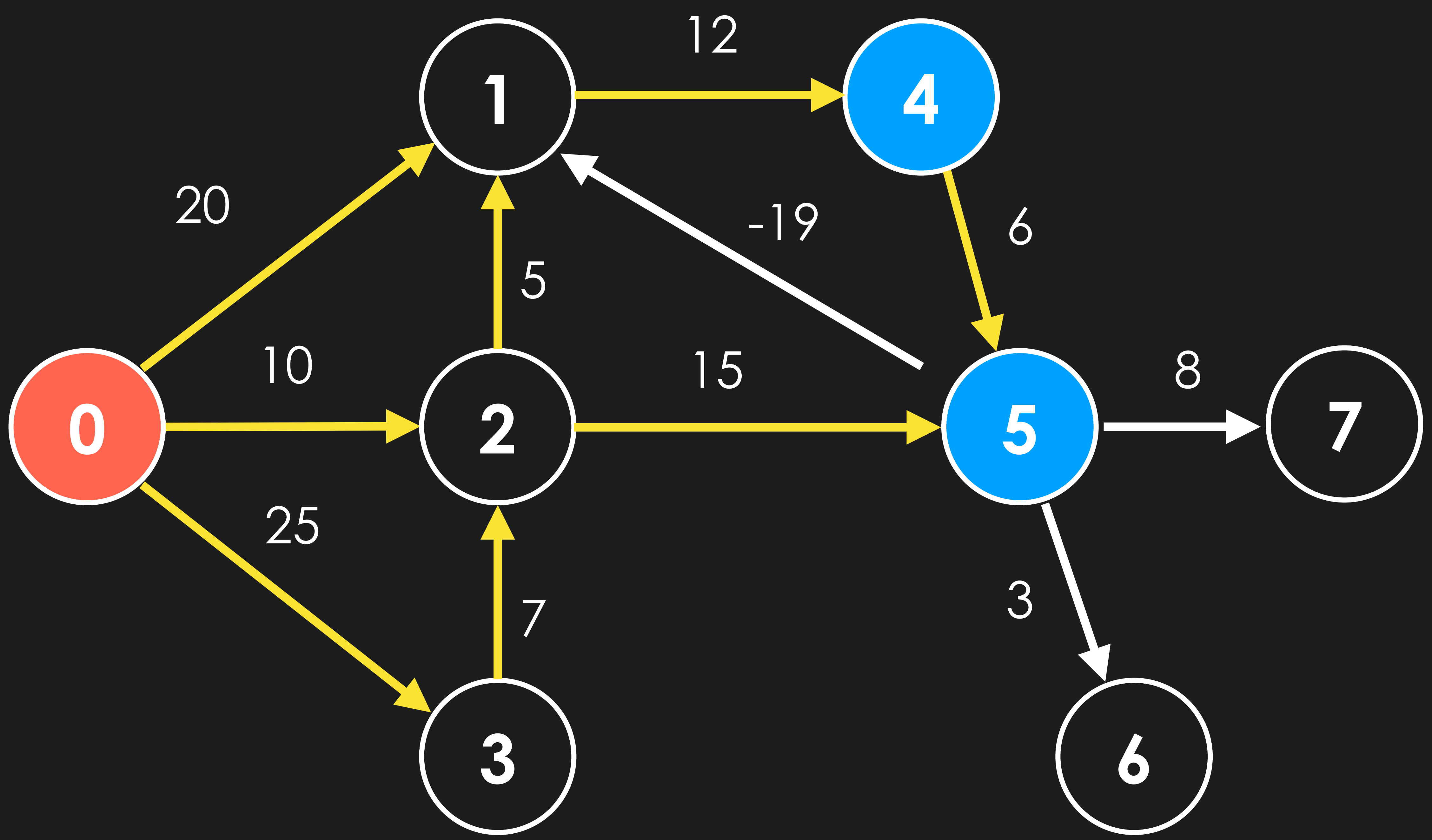
	distTo
0	0
1	-1
2	10
3	25
4	-INF
5	18
6	21
7	26

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

CodeIT

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**



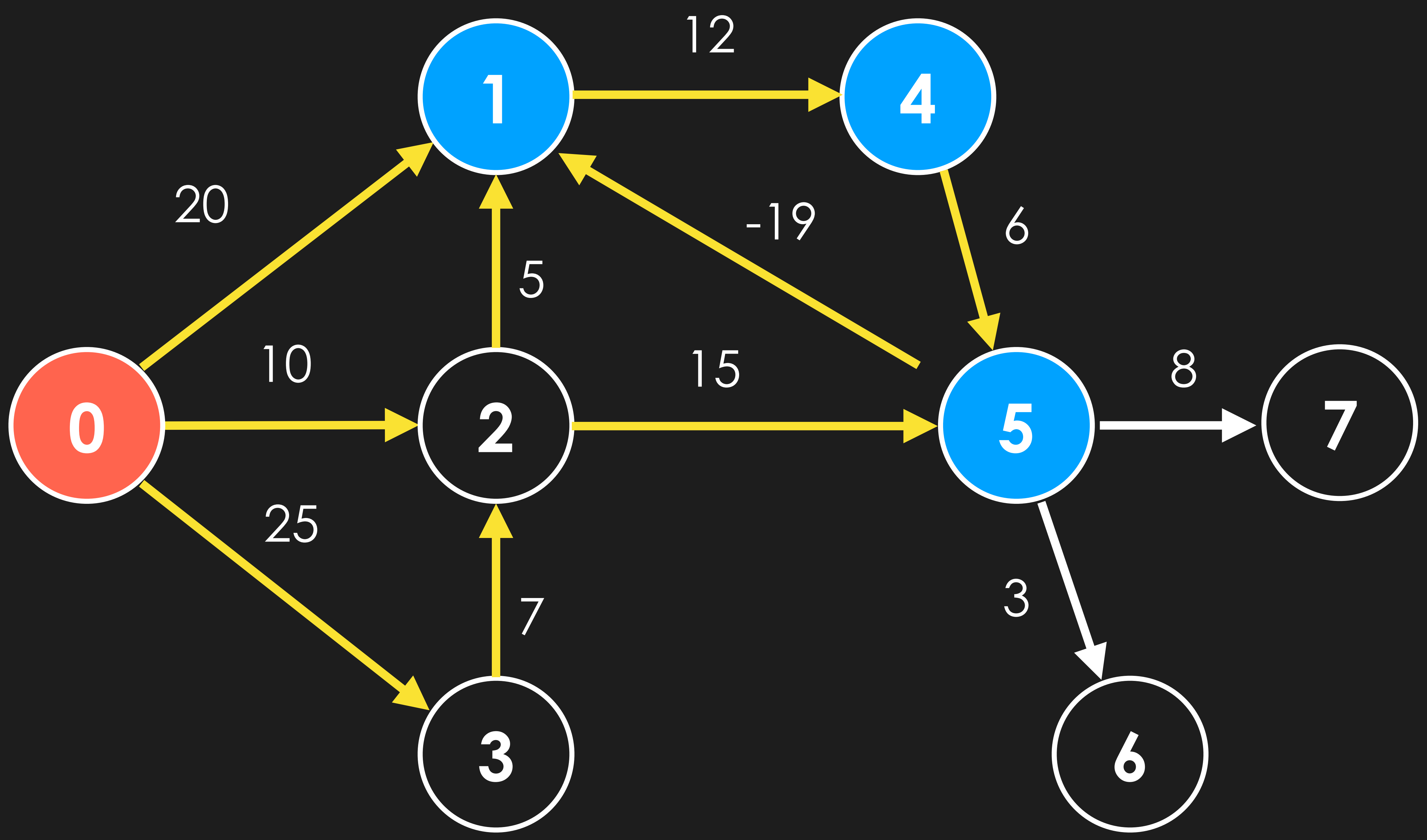


	distTo	**edgeTo**
0	0	-1
1	-1	5 - 1
2	10	0 - 2
3	25	0 - 3
4	-INF	1 - 4
5	-INF	4 - 5
6	21	5 - 6
7	26	5 - 7

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

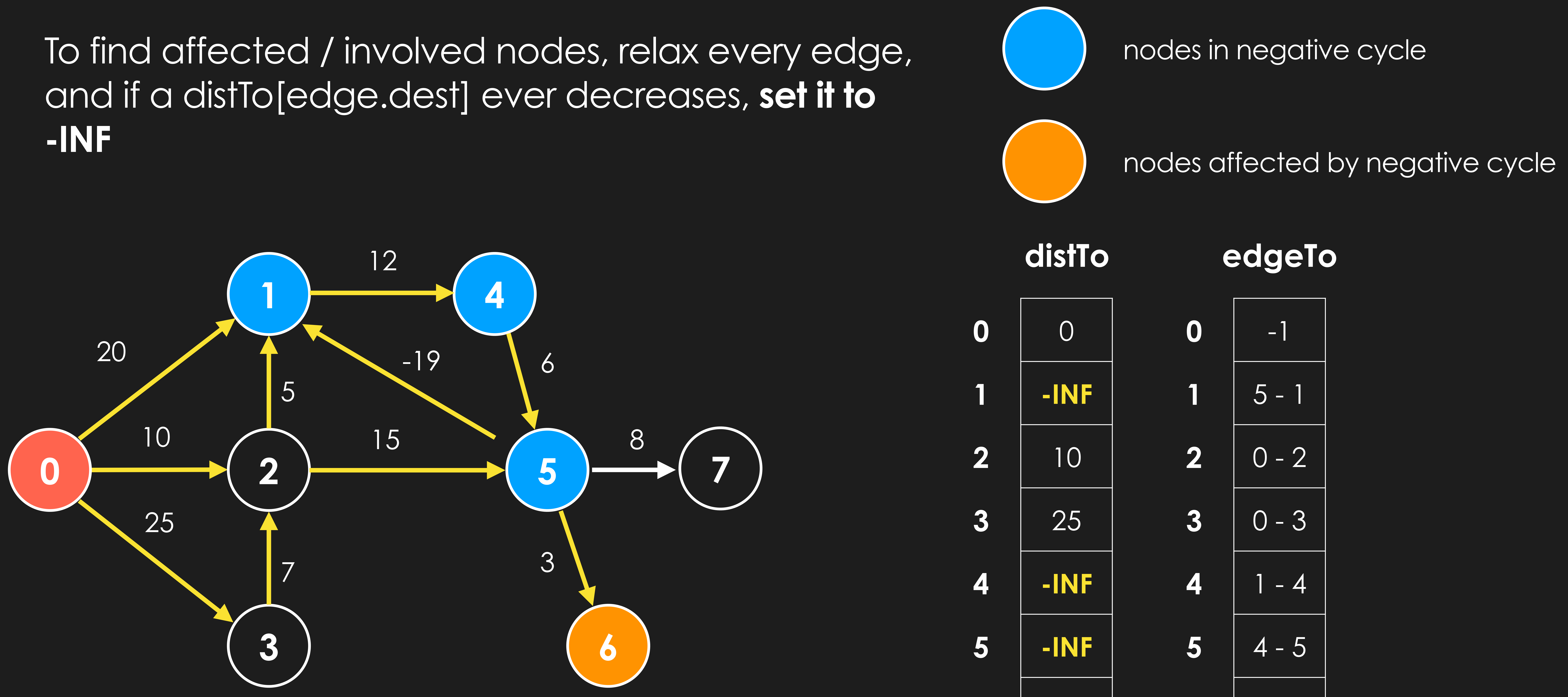
nodes in negative cycle

nodes affected by negative cycle



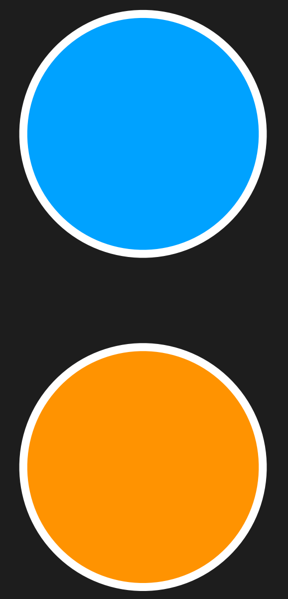
	distTo	edgeTo
0	0	-1
1	**-INF**	5 - 1
2	10	0 - 2
3	25	0 - 3
4	**-INF**	1 - 4
5	**-INF**	4 - 5
6	21	5 - 6
7	26	5 - 7

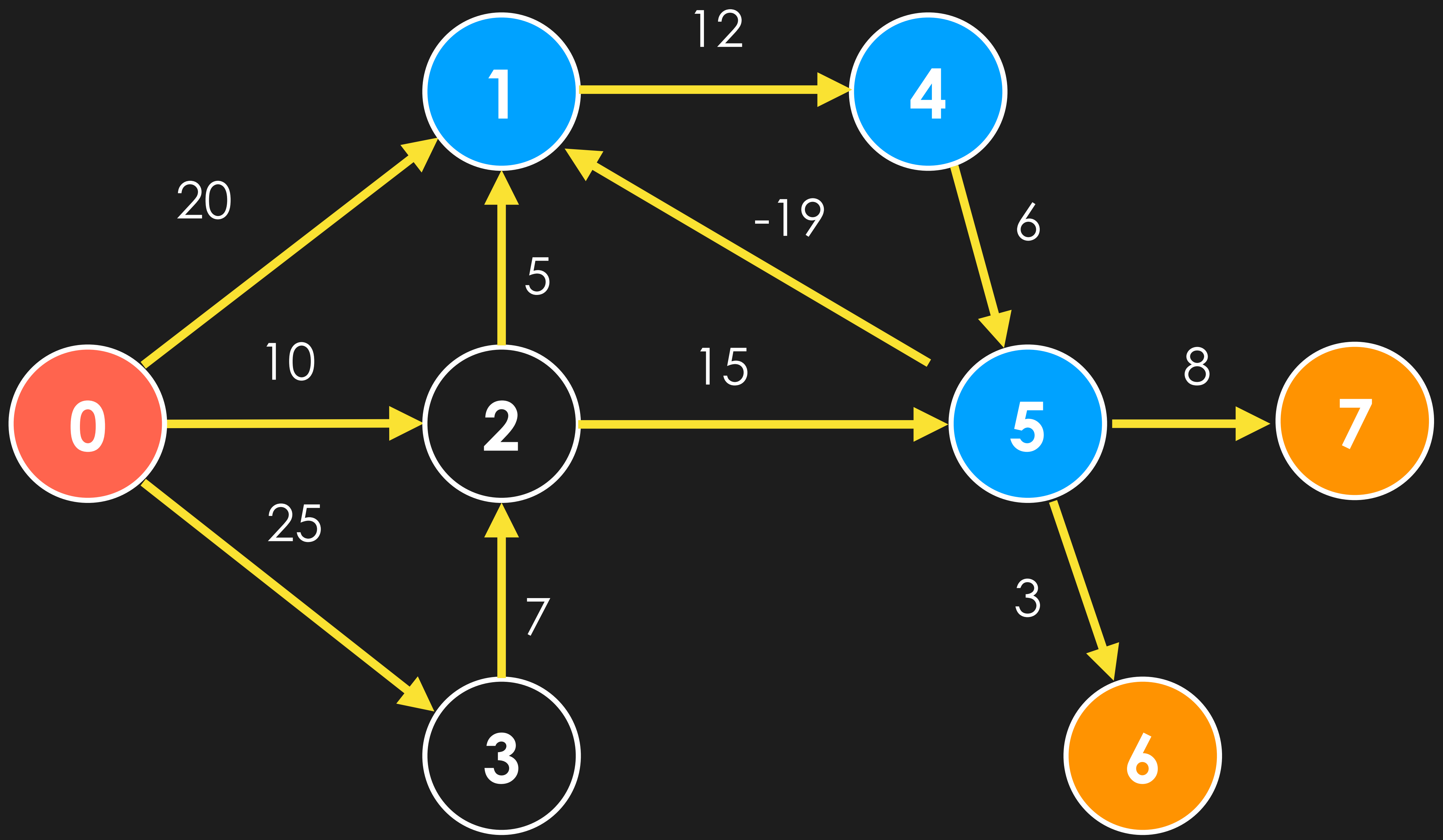
2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**



	distTo		edgeTo
0	0	**0**	-1
1	-INF	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	-INF	**4**	1 - 4
5	-INF	**5**	4 - 5
6	-INF	**6**	5 - 6
7	26	**7**	5 - 7

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**





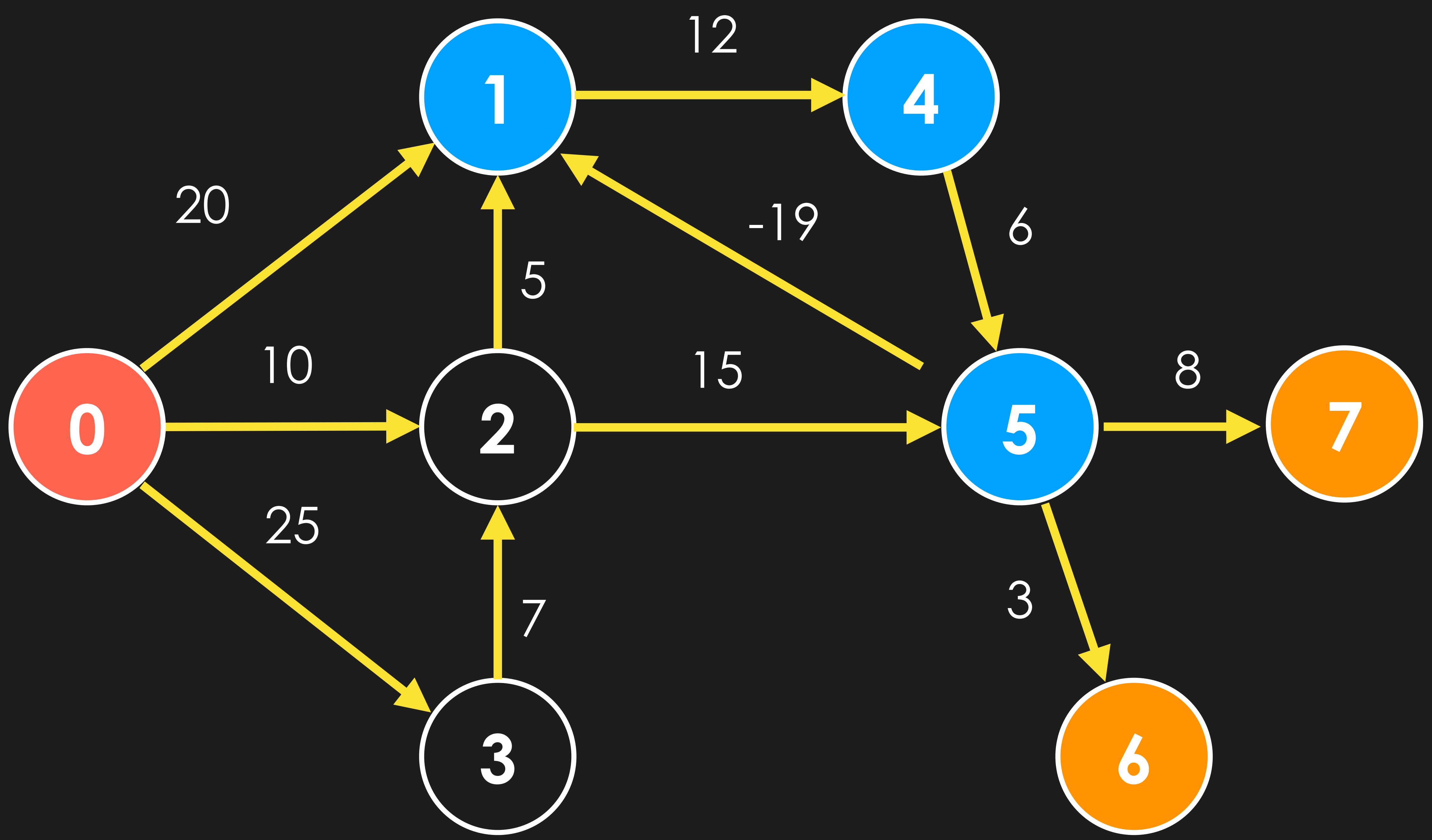
	distTo
0	0
1	-INF
2	10
3	25
4	-INF
5	-INF
6	-INF
7	-INF

	edgeTo
0	-1
1	5 - 1
2	0 - 2
3	0 - 3
4	1 - 4
5	4 - 5
6	5 - 6
7	5 - 7

2. To find affected / involved nodes, relax every edge, and if a distTo[edge.dest] ever decreases, **set it to -INF**

nodes in negative cycle

nodes affected by negative cycle



	distTo		**edgeTo**
0	0	**0**	-1
1	**-INF**	**1**	5 - 1
2	10	**2**	0 - 2
3	25	**3**	0 - 3
4	**-INF**	**4**	1 - 4
5	**-INF**	**5**	4 - 5
6	**-INF**	**6**	5 - 6
7	**-INF**	**7**	5 - 7

Only **vertex 2 and 3** are not affected / in a negative cycle!

Implementation of Bellman-Ford Algorithm

```python
def BellmanFord(graph, s):
    INF = 99999
    V = len(graph.adjList)
    distTo = [INF] * V
    edgeTo = [None] * V
    distTo[s] = 0

    for v in range(V):
        for v in range(V):
            for edge in graph.adjList[v]:
                w = edge.dest
                if distTo[w] > distTo[v] + edge.weight:
                    distTo[w] = distTo[v] + edge.weight
                    edgeTo[w] = edge

    for v in range(V):
        for edge in graph.adjList[v]:
            w = edge.dest
            if distTo[w] > distTo[v] + edge.weight:
                distTo[w] = -INF
```

CodeIT

```python
def BellmanFord(graph, s):
    INF = 99999
    V = len(graph.adjList)
    distTo = [INF] * V
    edgeTo = [None] * V
    distTo[s] = 0

    for v in range(V):
        for v in range(V):
            for edge in graph.adjList[v]:
                w = edge.dest
                if distTo[w] > distTo[v] + edge.weight:
                    distTo[w] = distTo[v] + edge.weight
                    edgeTo[w] = edge

    for v in range(V):
        for edge in graph.adjList[v]:
            w = edge.dest
            if distTo[w] > distTo[v] + edge.weight:
                distTo[w] = -INF
```

Check each edge one more time to catch vertices affected in negative cycle

CodeIT

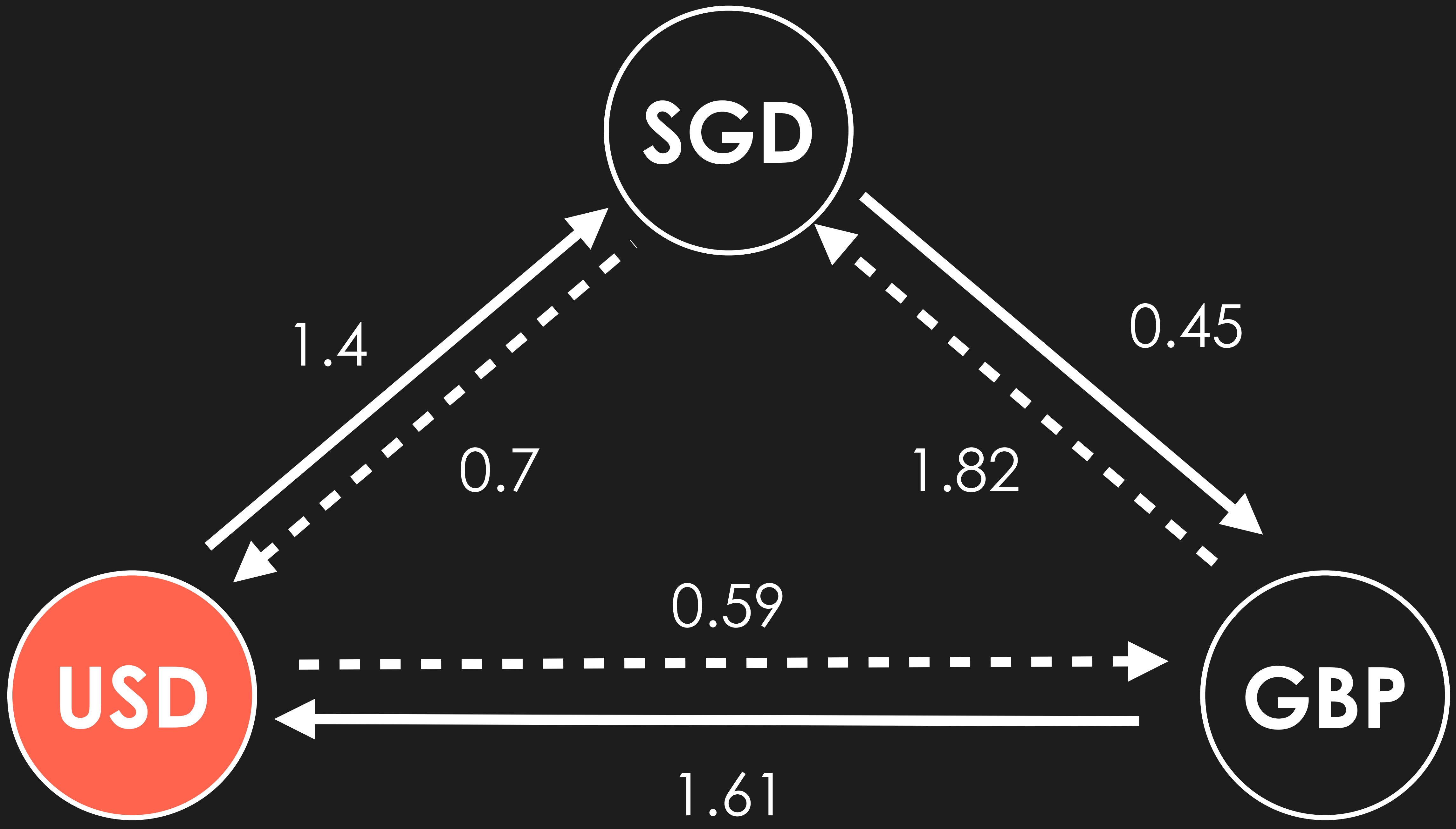
Analysis of Bellman-Ford

For each vertex, we iterate over every edge, thus the time complexity of Bellman-Ford is **E * V**

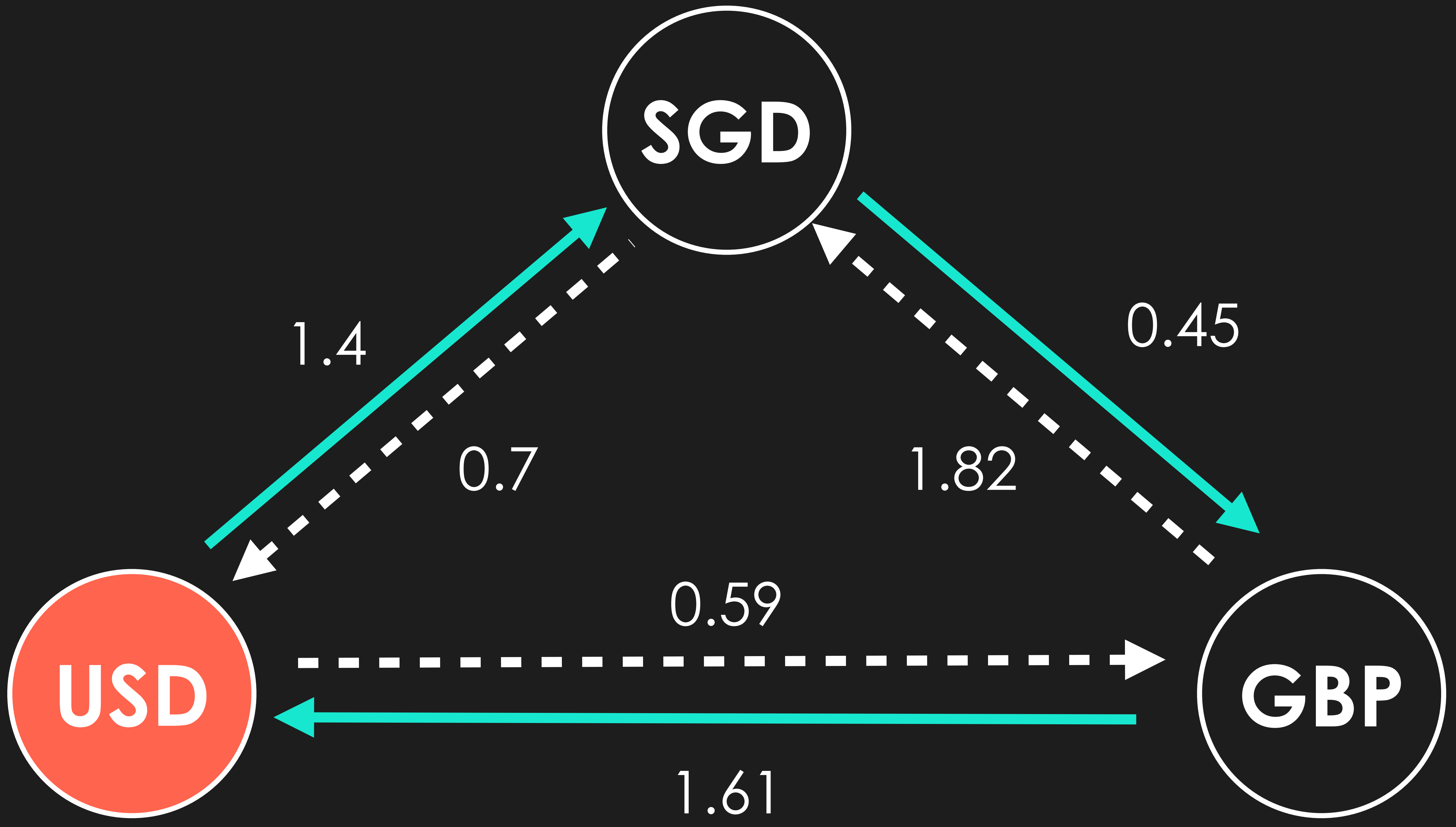
Applications of negative cycle detection

Arbitrage

Imagine if we represented currency exchanges as graphs:

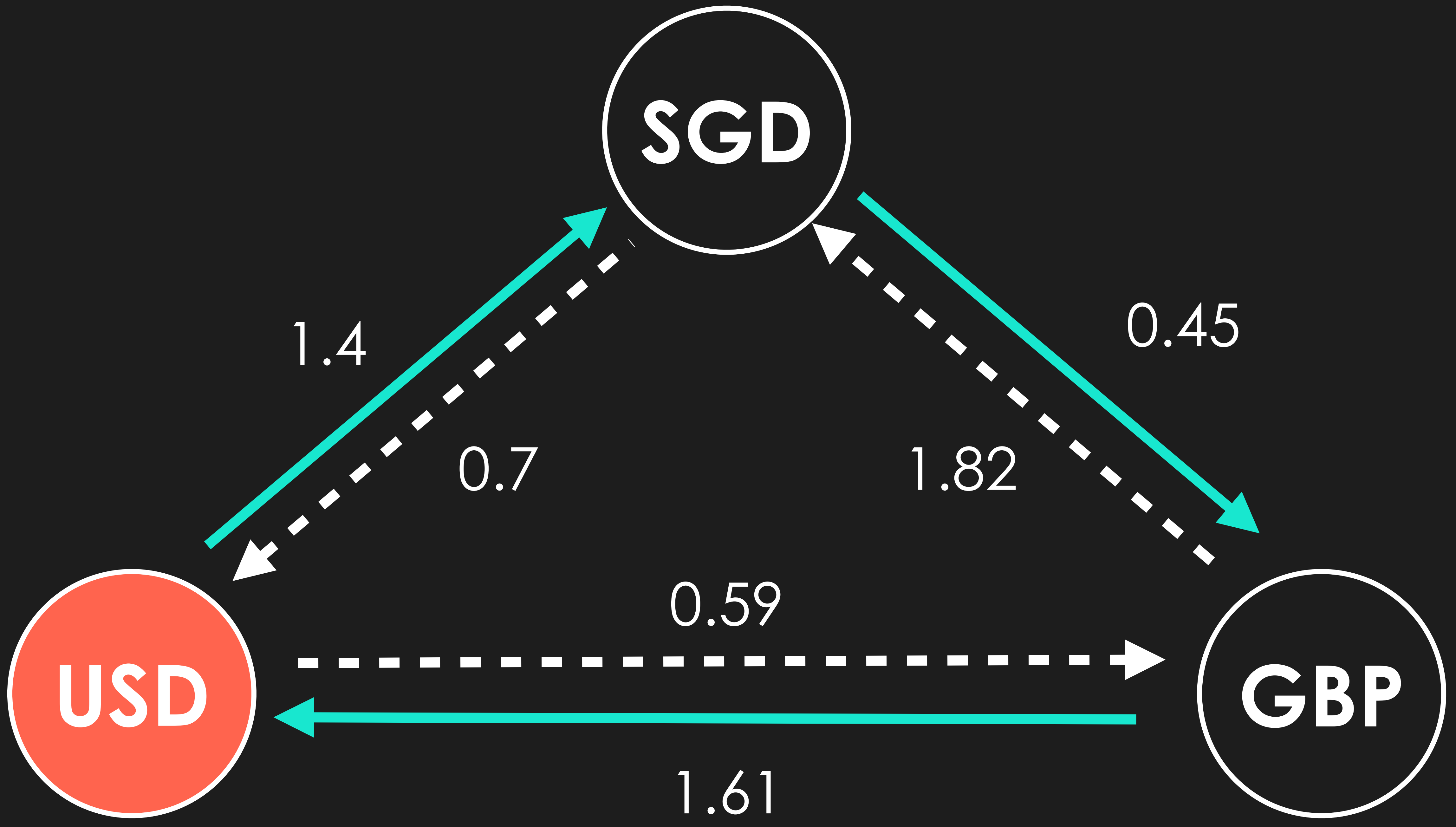


Imagine if we represented currency exchanges as graphs:



`USD->SGD->GBP->USD = 1 * 1.4 * 0.45 * 1.61 = 1.0143 (Profit!)`

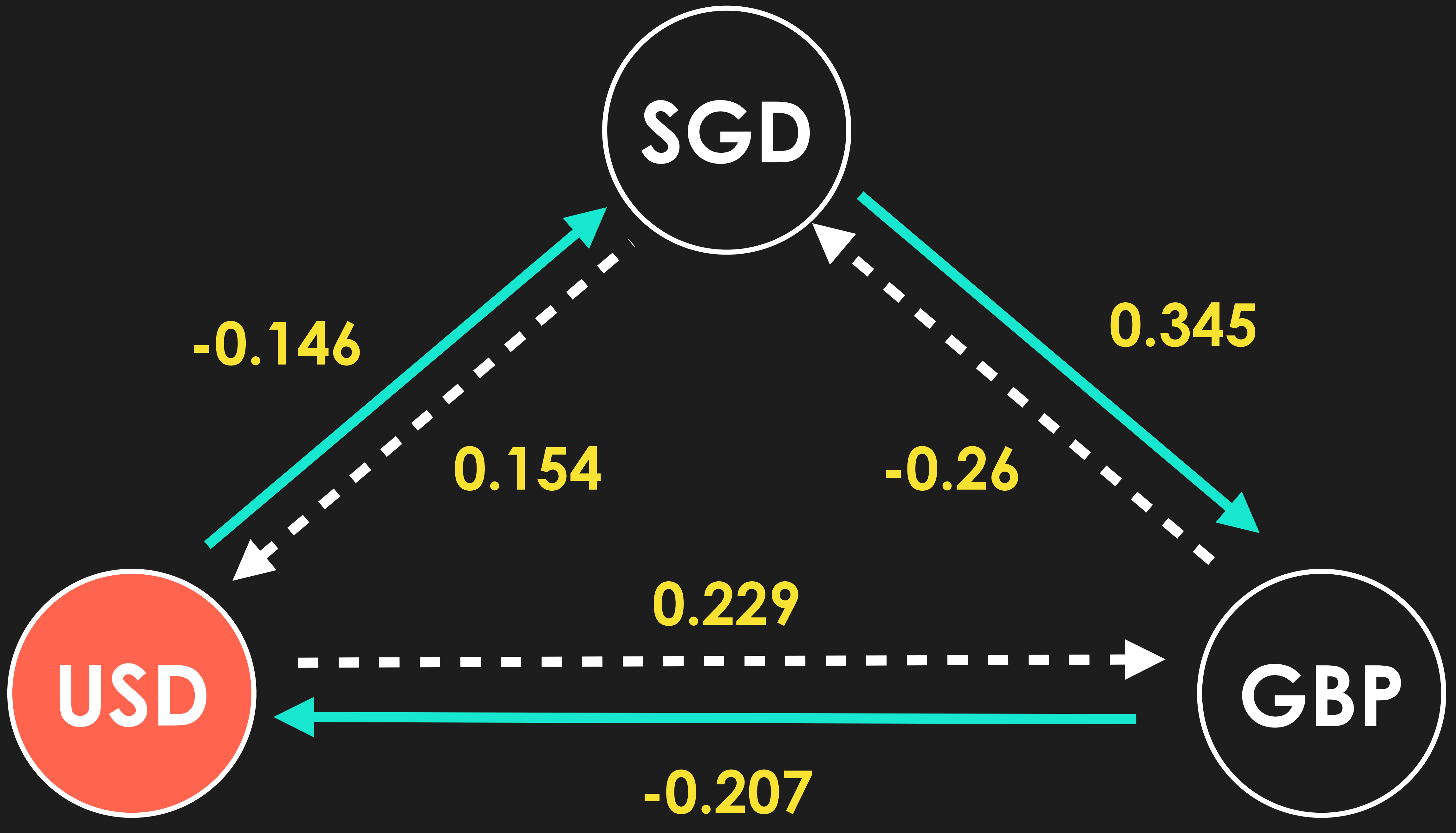
Imagine if we represented currency exchanges as graphs:



`USD->SGD->GBP->USD = 1 * 1.4 * 0.45 * 1.61 = 1.0143 (Profit!)`

How can we identify such cycles in exchange rates?

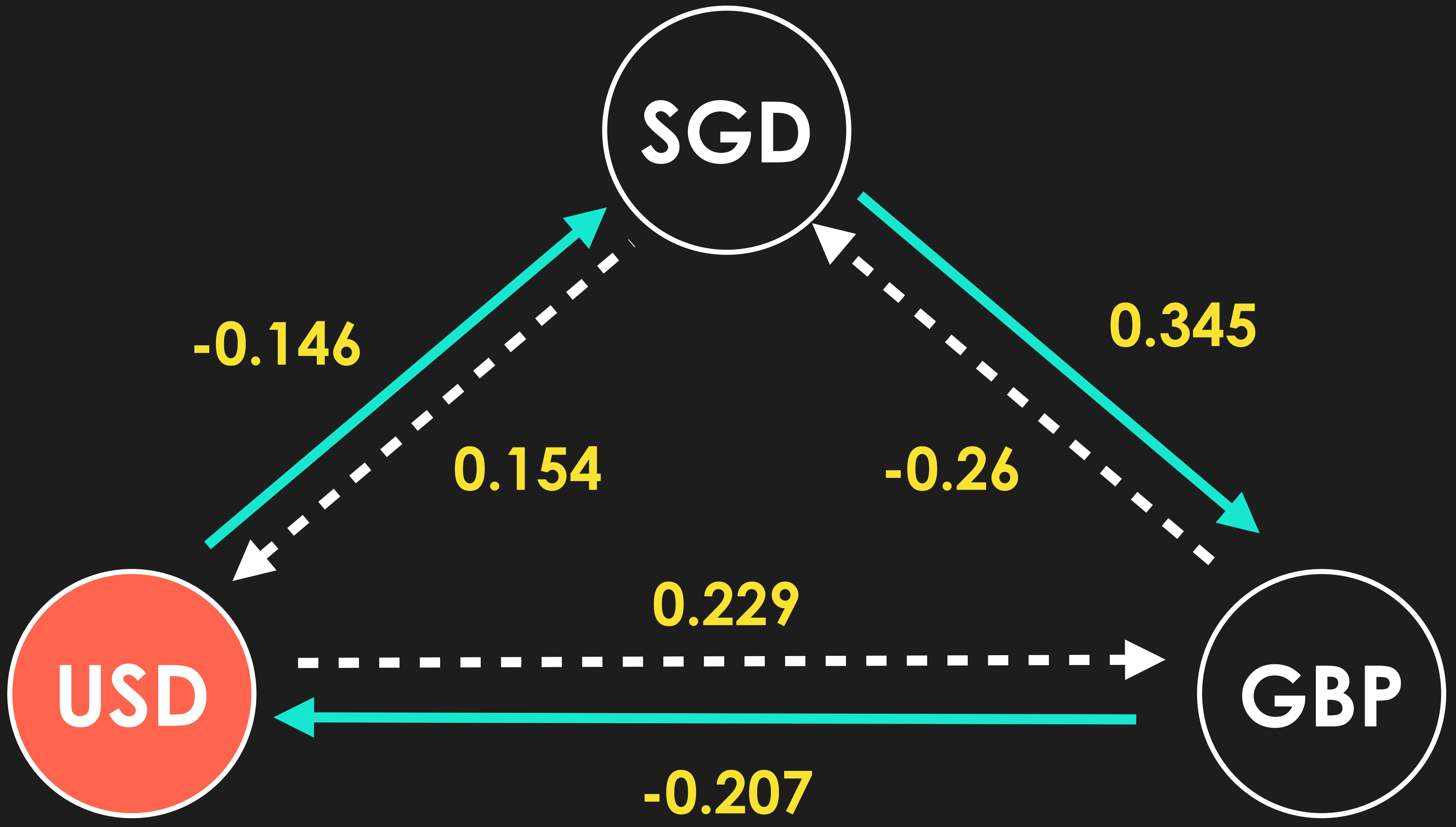
Imagine if we represented currency exchanges as graphs:



`USD->SGD->GBP->USD = - 0.146 + 0.345 - 0.207`

Convert each weight to negative log!

Imagine if we represented currency exchanges as graphs:



USD->SGD->GBP->USD = − 0.146 + 0.345 − 0.207 = −0.008

Negative Cycle indicates Profit!

Arbitrage

Arbitrage is the **strategy of detecting imbalances in markets**, and leveraging that imbalance to make profits, as shown in the diagram previously

Arbitrage

Arbitrage is the **strategy of detecting imbalances in markets**, and leveraging that imbalance to make profits, as shown in the diagram previously

Since market imbalances only **last for such a short time**, we need **efficient algorithms** to detect these imbalances **quickly**

Other Graph Algorithms / Problems

Max Flow Ford Fulkerson Algorithm

Bipartite Matching

Travelling Salesman Problem

Lab Session 1

- In this lab session, you will be implementing **dijkstra.py**
- Your task is to implement **Dijkstra's Shortest Path** algorithm for a Weighted Digraph with only non-negative weights
- Your algorithm should be of **ElogV time complexity**
- The DijkstraSP function takes in two arguments: graph & start, where start is the starting vertex for the shortest paths
- The WeightedEdge & WeightedDigraph class has been implemented for you, and functions as that shown in the lesson.
- You should handle changing values of keys in your minPQ using the decreaseKey() method given to you.
- To check if a key already exists inside the MinPQ, you may check the positions dict attribute `` `if key in MinHeap.positions` ``
- To test, run `` `**python utils/dijkstra_test.py**` ``

WeightedEdge & WeightedDigraph

```python
class WeightedEdge:
    def __init__(self, src, dest, weight):
        self.src = src
        self.dest = dest
        self.weight = weight

    def __str__(self):


class WeightedDiGraph:
    def __init__(self, V):
        self.adjList = [[] for i in range(V)]

    def addEdge(self, src, dest, weight):

    def printGraph(self):
```

MinHeap

```python
class HeapItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value


class MinHeap:
    def __init__(self, maxsize):
        self.maxsize = maxsize
        self.size = 0
        self.heap = [None] * (maxsize + 1)
        self.positions = {}

    def insert(self, newKey, newValue) -> None:

    def decreaseKey(self, key, newValue) -> None:

    def getMin(self) -> HeapItem:
```

```python
def DijkstraSP(graph: WeightedDigraph, start: int):

    V = len(graph.adjList)
    pq = MinHeap(V)

    for v in range(V):
        for edge in graph.adjList[v]:
            if edge.weight < 0:
               return False

    edgeTo = [None] * V
    distTo = [None] * V

    distTo[start] = 0
    pq.insert(start, distTo[start])

    while (pq.size != 0):
        v = pq.getMin().key

        for edge in graph.adjList[v]:
            relax(edge, pq, distTo, edgeTo)

    return edgeTo, distTo
```

```python
def relax(edge, pq, distTo, edgeTo):
    v = edge.src
    w = edge.dest

    if distTo[w] == None or distTo[v] + edge.weight < distTo[w]:
        distTo[w] = distTo[v] + edge.weight
        edgeTo[w] = edge
        if w in pq.positions:
            pq.decreaseKey(w, distTo[w])
        else:
            pq.insert(w, distTo[w])
```