Graph Problems

Traversals and Shortest Paths

Lucas Wood

Outline

- Graph Traversal
  - DFS
  - BFS
  - Applications of the two

Outline

- Minimum Spanning trees
    - Motivation
    - Kruskal's algorithm
    - Prim's algorithm

Outline

- Shortest Path algorithms
    - Dijkstra’s
    - Bellman Ford
    - Floyd-Warshall
    - Shortest path variants
    - Problem

Depth First Search

- Traverse a graph by depth
  - Start at a some source node
  - When you get to a branching point choose an unvisited neighbour and go to it
  - When you reach a dead end return to the previous node(s) until you find a new unvisited node to go to

Depth First Search

- $O(V + E)$ with adjacency list
- $O(V^2)$ with adjacency matrix
- Simple to code
- Limited application

Breadth First Search

- Traverse a graph by breadth
  - Place starting node into a queue
  - For every node, add its neighbours to the queue
  - Repeat for every node in the Graph

Breadth First Search

- O(V+E) complexity
- $O(V^2)$ with adjacency matrix
- Simple to code
- Can solve SSSP on an unweighted graph

BFS and DFS Applications

Find Connected Components

- On an undirected graph simply perform a DFS or BFS on a single node
- This tells you which nodes are connected to the starting node

Flood Fill

- When you have a graph (usually a grid) of different components: want to count number of each component
- Iterate through the graph until you find component you wish to count
- Use a modified DFS to “colour in” connected components
- Continue until you’ve gone through the entire graph

Topological Sort (DAG)

- Topological Sort: a linear ordering of a graph's vertices such that for every edge *uv* from vertex *u* to vertex *v, u* comes before *v*

- *Not* a unique solution

- Real world problem: course requirements

Topological Sort (DAG)

- Uses a modified DFS
- Simply push the vertex being searched back onto the list of explored vertices after visiting all the subtrees below it

Even More

- Check if a graph is bipartite
- Edge property check
- Find articulation points and bridges
- Find *Strongly* connected components in a directed graph
- Bidirectional Search

Minimum Spanning Trees

MST - Motivation

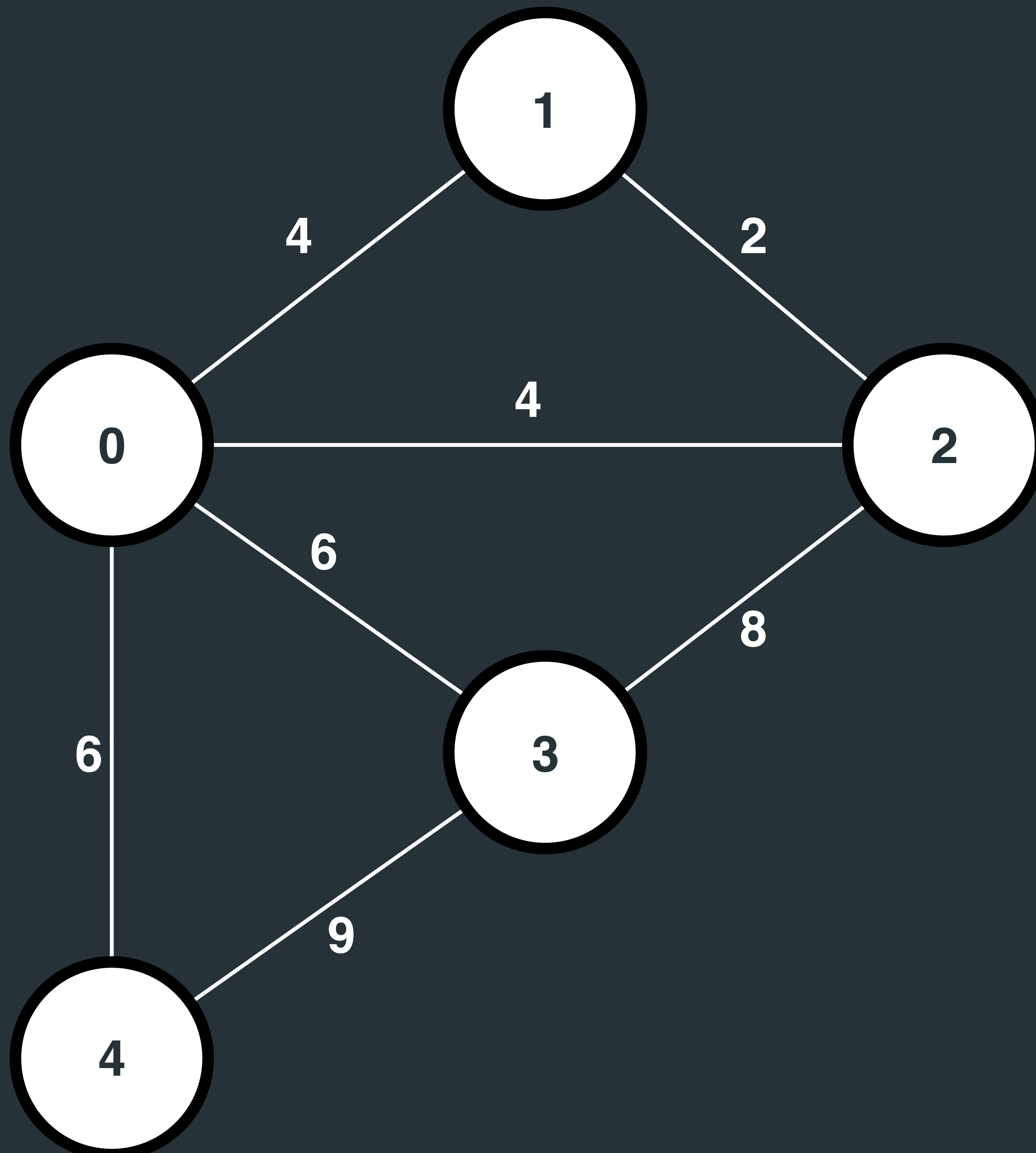
- Given a connected, undirected, weighted graph G, select a subset of edges such that the graph G is still connected and the total weight of the selected edges is minimal.

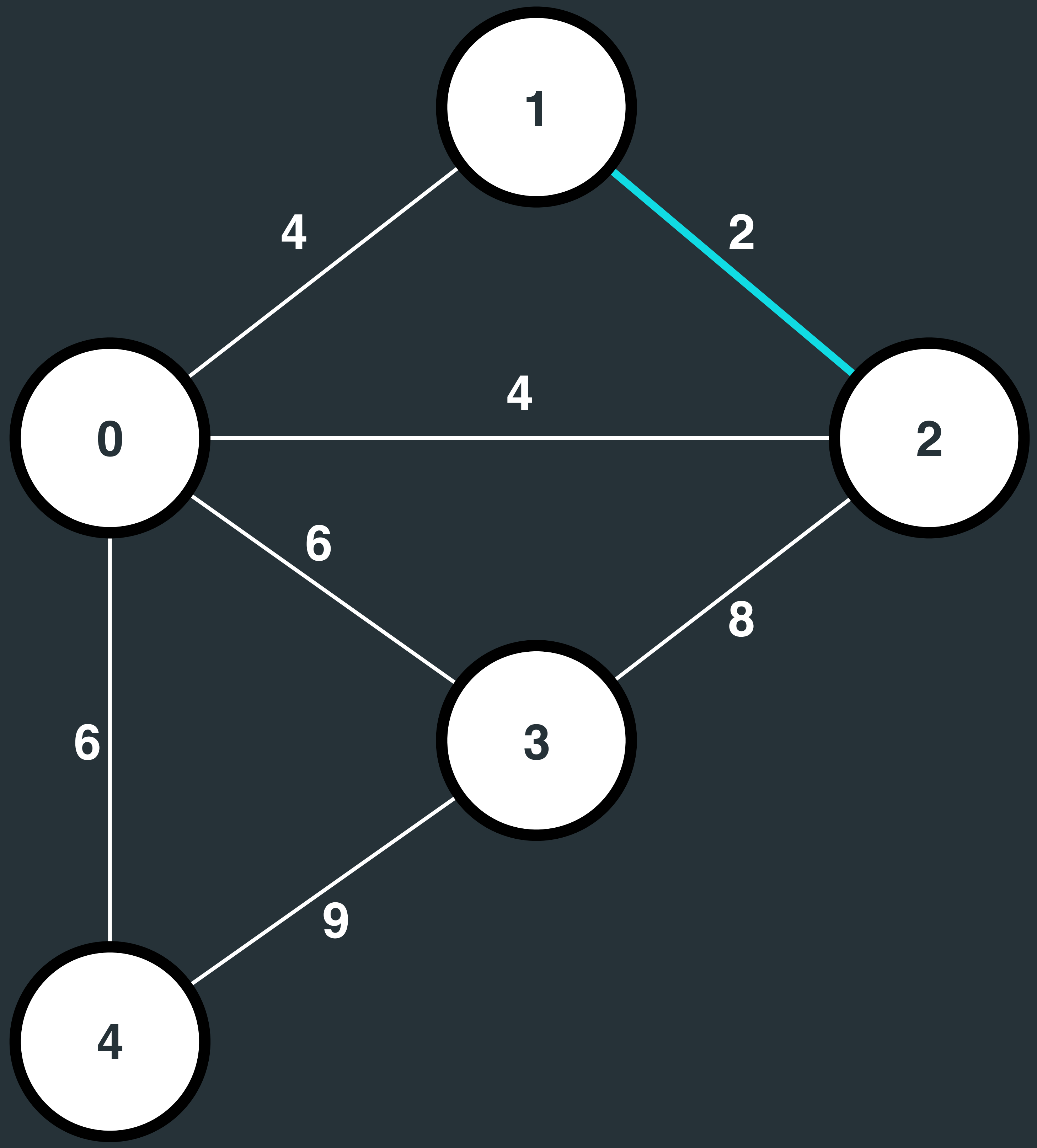
MST - Kruskal’s

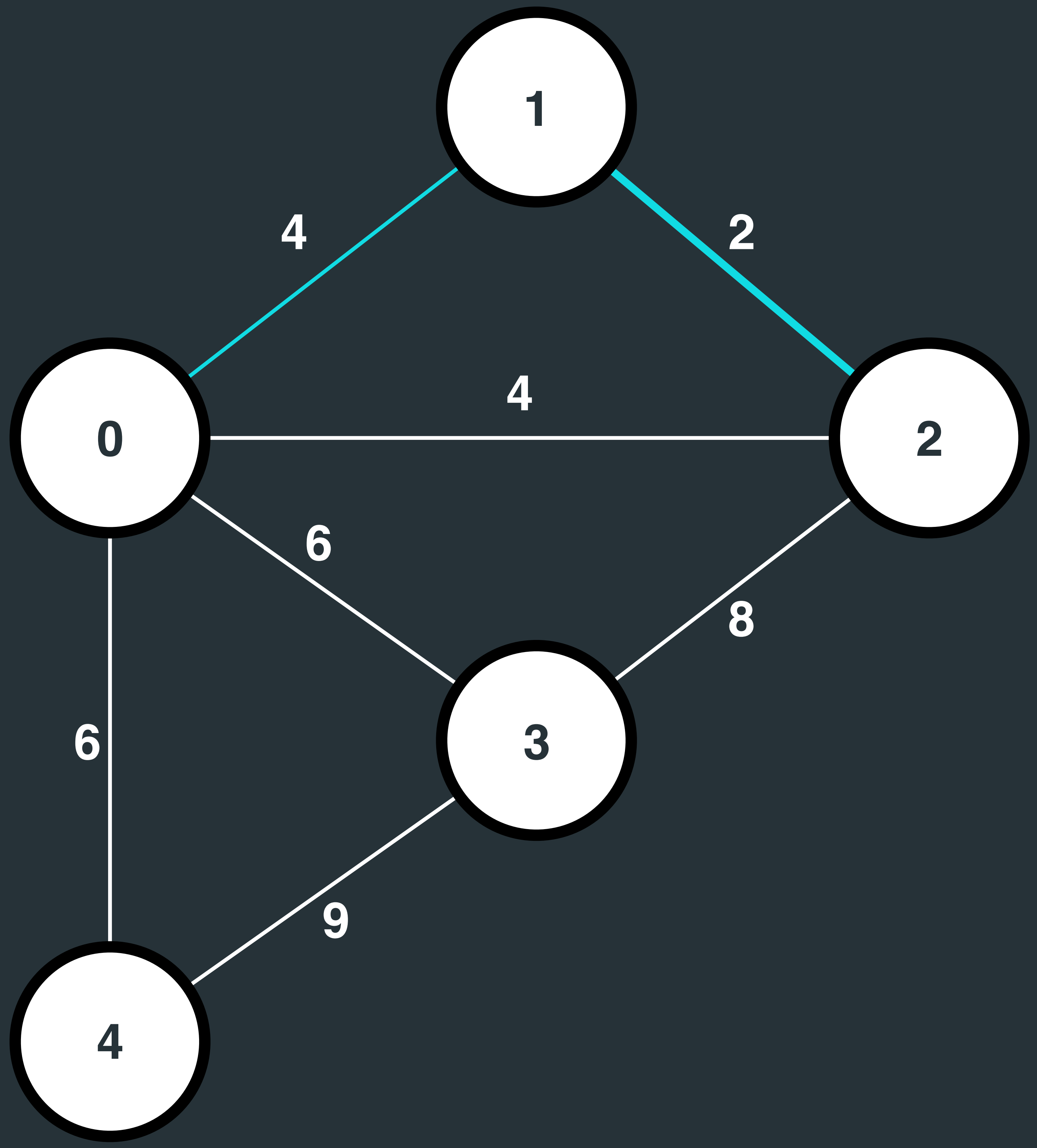
- Uses Union-Find to find a minimum spanning tree
- O(E logV) complexity

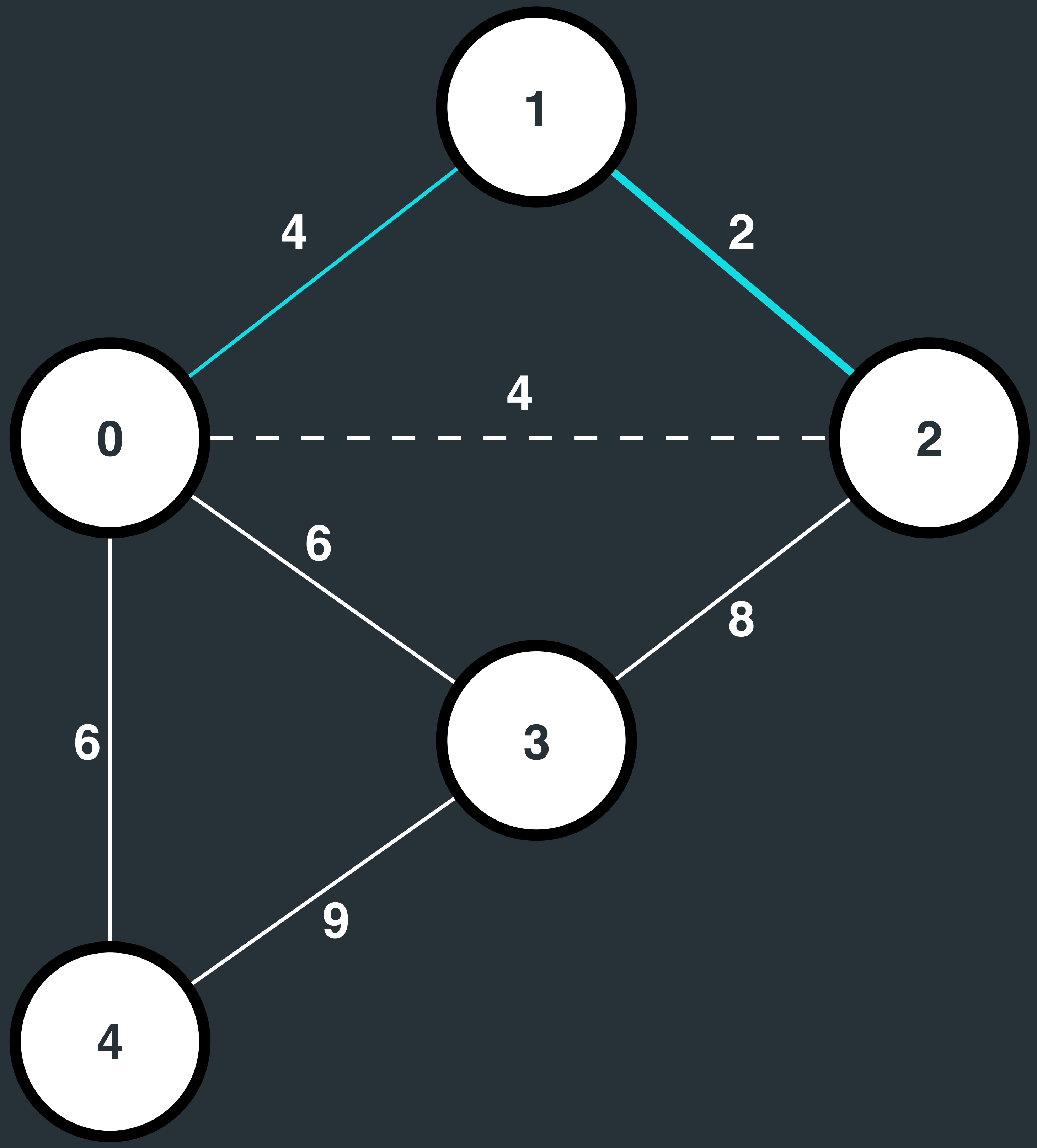
MST - Kruskal’s

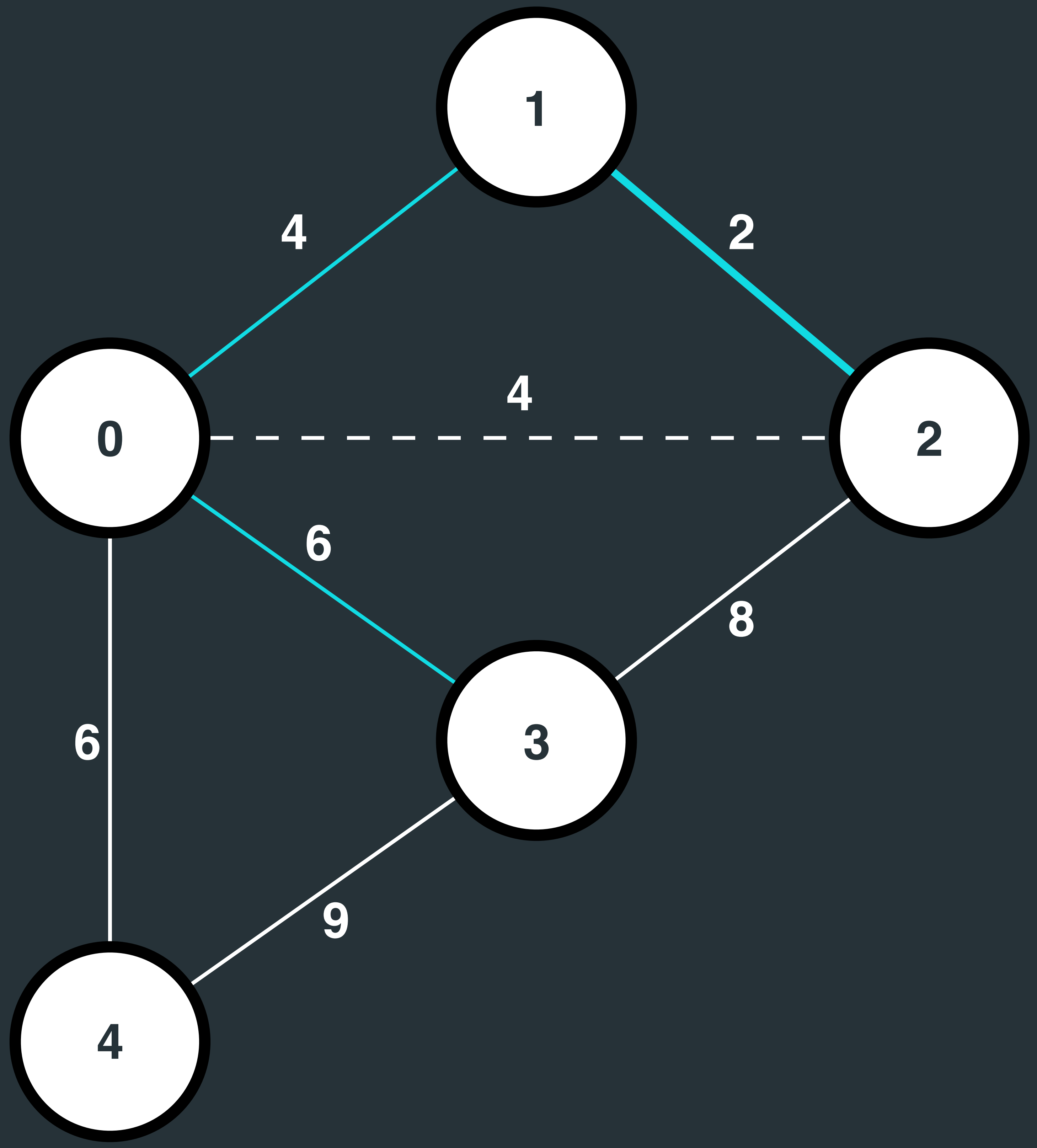
- Sort edges by ascending weight
- Greedily adds each edge into the MST - as long as the addition does not have a cycle
- This check is done with a lightweight Union-Find


1
4
2
0
4
2
6
8
3
6
9
4

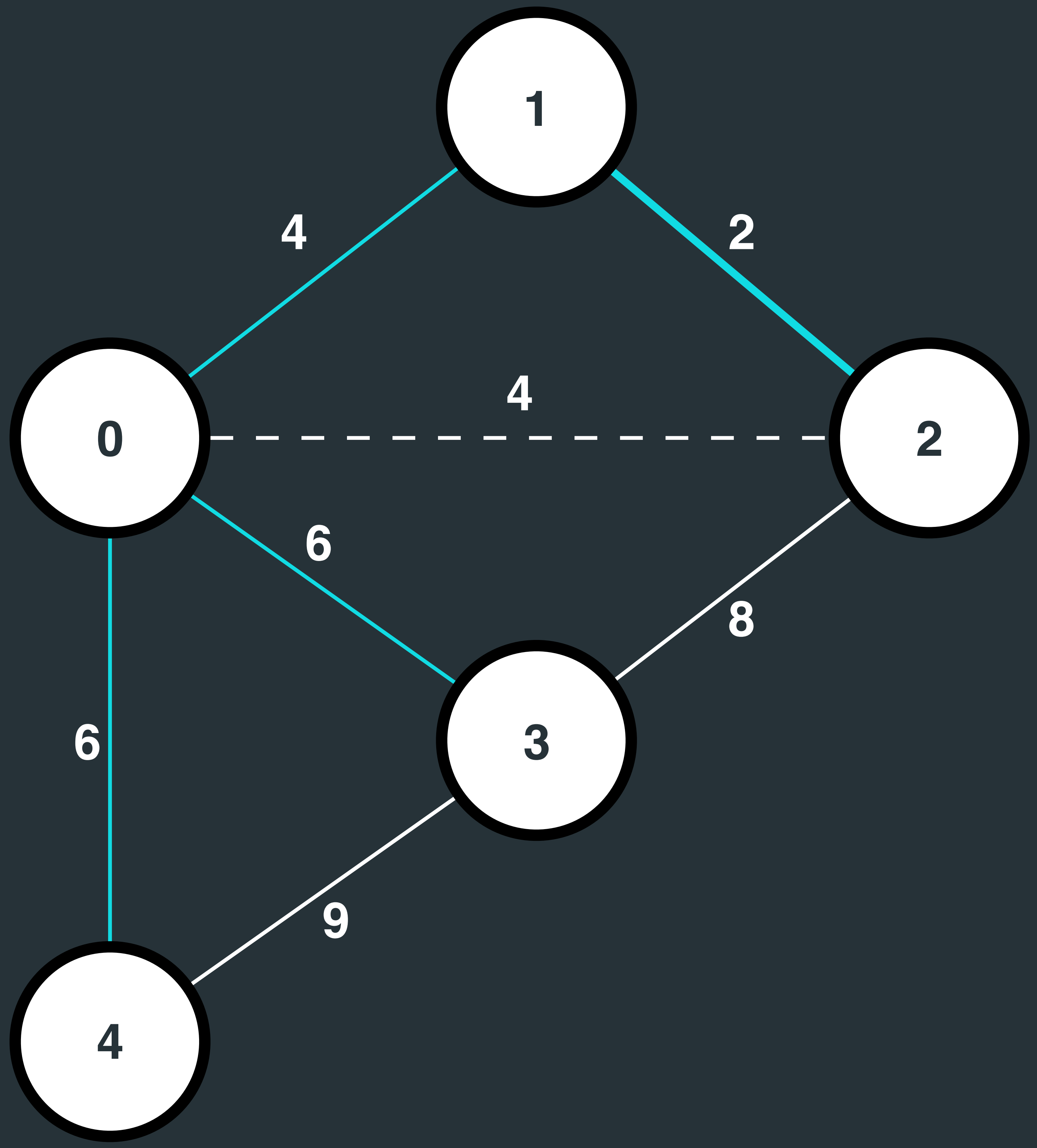

1
4
2
0
4
2
6
8
6
3
9
4

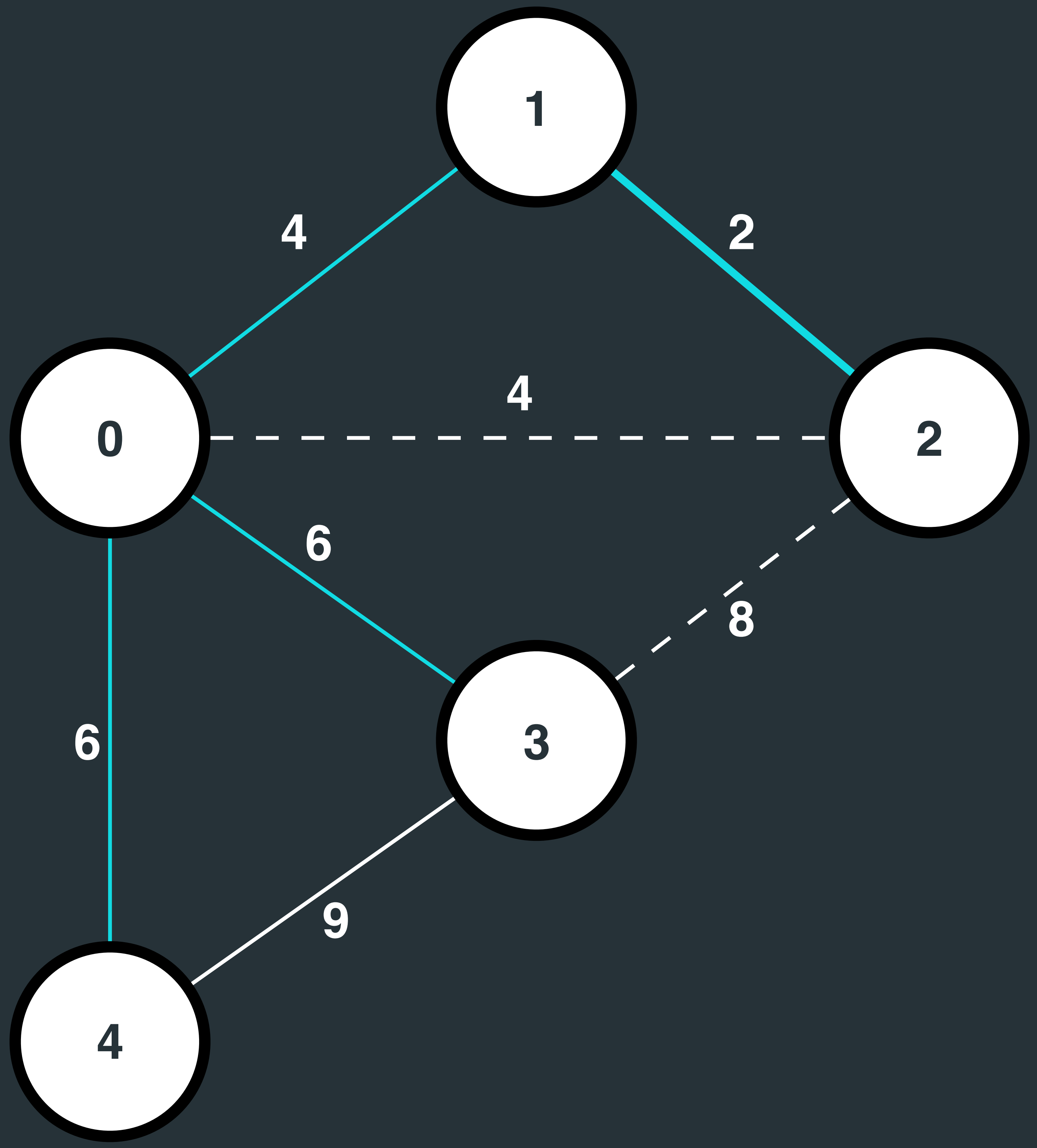

1
2
0
3
4
4
2
4
6
8
6
9

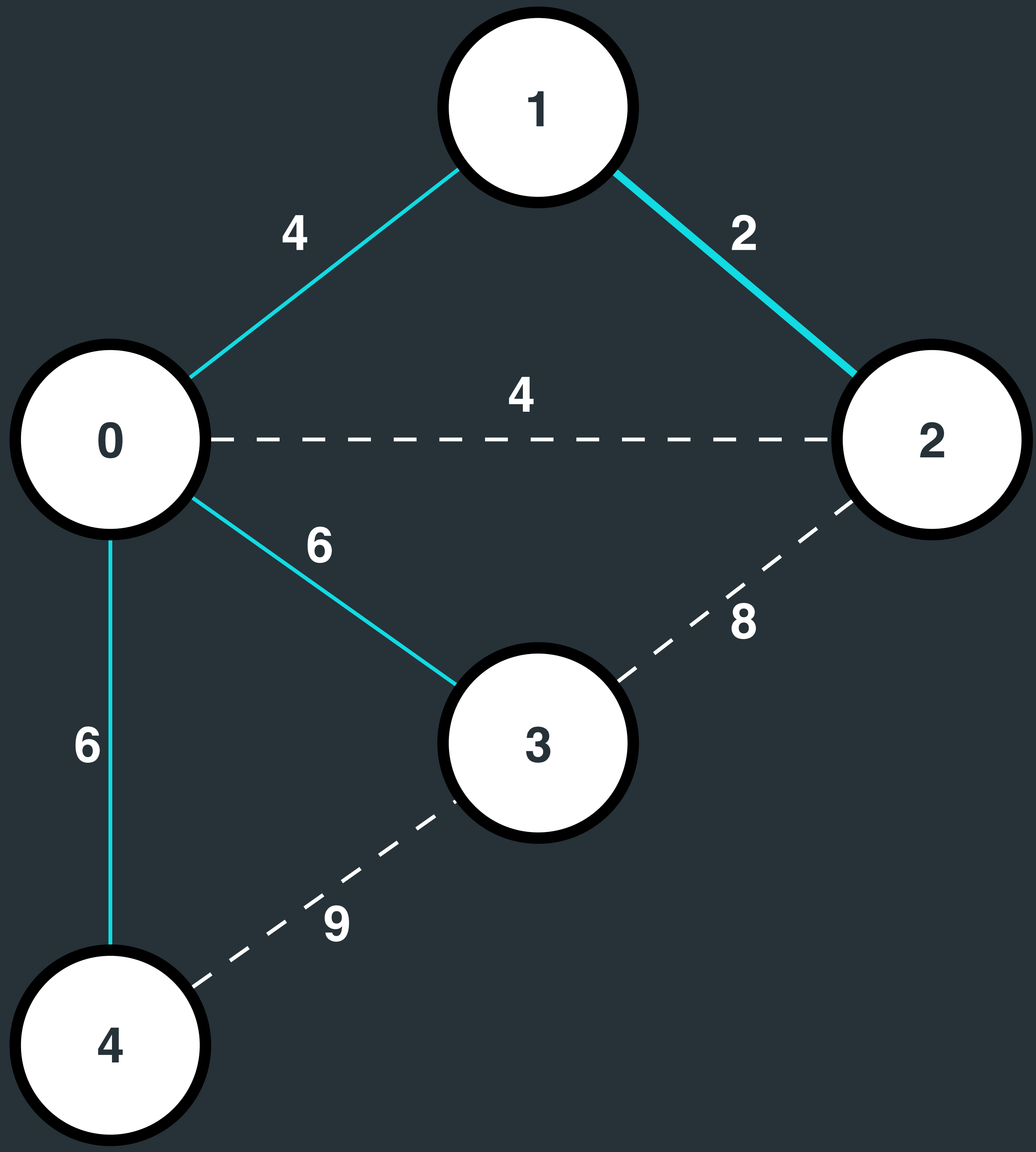

1
4
2
0
4
2
6
8
6
3
9
4



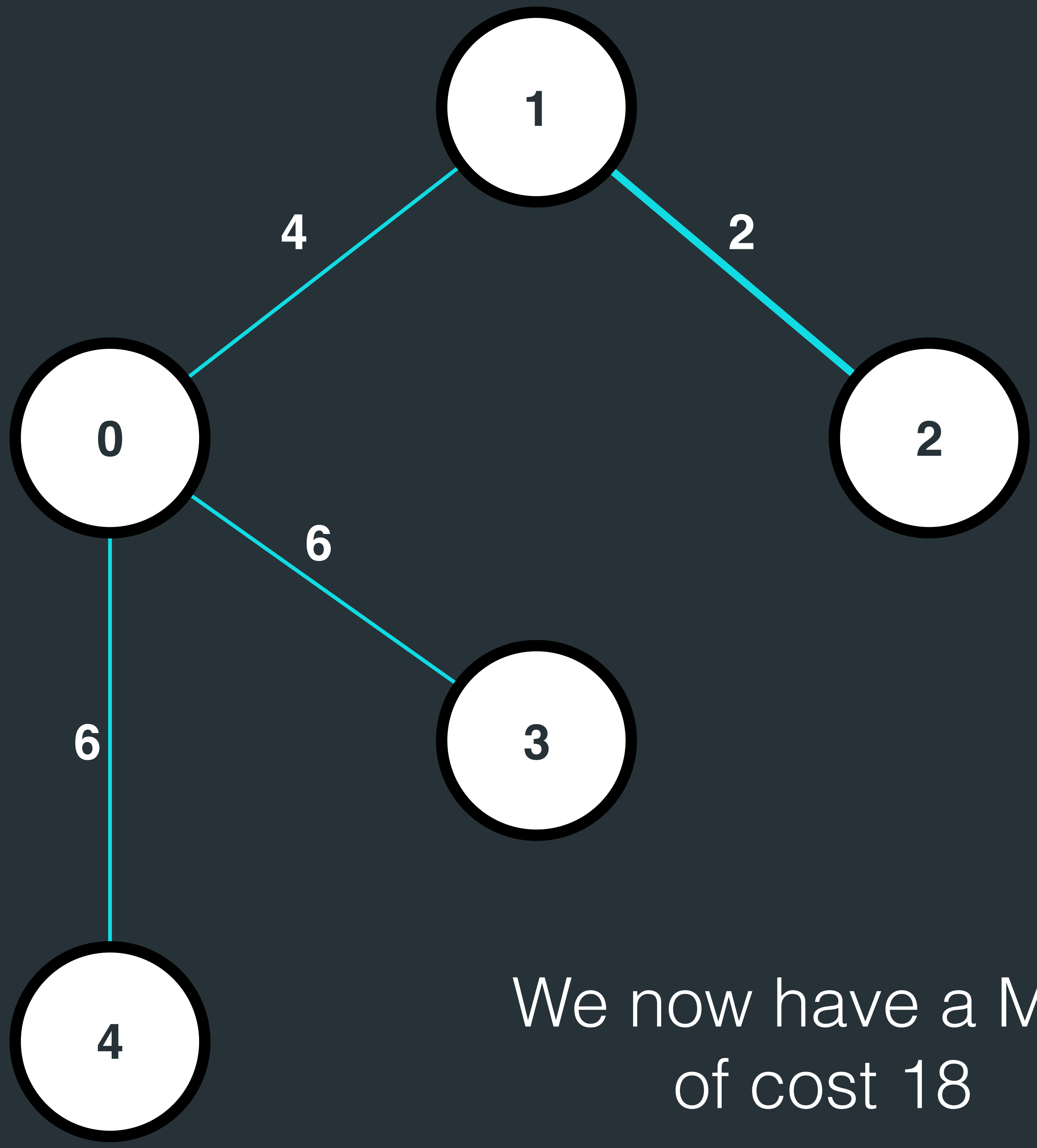
1
4
2
0
4
2
6
8
6
3
9
4


1
4
2
0
4
2
6
8
6
3
9
4


1
4
2
0
4
2
6
8
3
6
9
4



1
4
2
0
4
2
6
8
6
3
9
4



We now have a MST
of cost 18

MST - Prim’s

- Uses a priority queue to pick which unvisited edges to the tree
- O(V logE)

MST - Prim’s

- Select a single vertex in the graph chosen arbitrarily and add it to your tree.
- Pick the edge connected to the vertex not yet in the tree with the lowest weight and add it to your tree
- Repeat until all vertices are in the tree

MST Variants

- Maximum spanning tree: simply modify Kruskal's to sort based on descending weight
- Subgraph: Rather than starting on a clean graph, you must fix a certain number of edges. Simply take into account the fixed edges before running the algorithm
- Spanning Forest: Limit the number of connected components by terminating early

MST Remarks

- Most (probably all) MST problems can be solved by Kruskal’s — relies on union find
- The trick is to identify the problem as a MST problem

Shortest Paths

Single Source Shortest Path

- Motivation: Given a weighted graph and a starting vertex, what are the shortest paths from the starting vertex to every other vertex in the graph

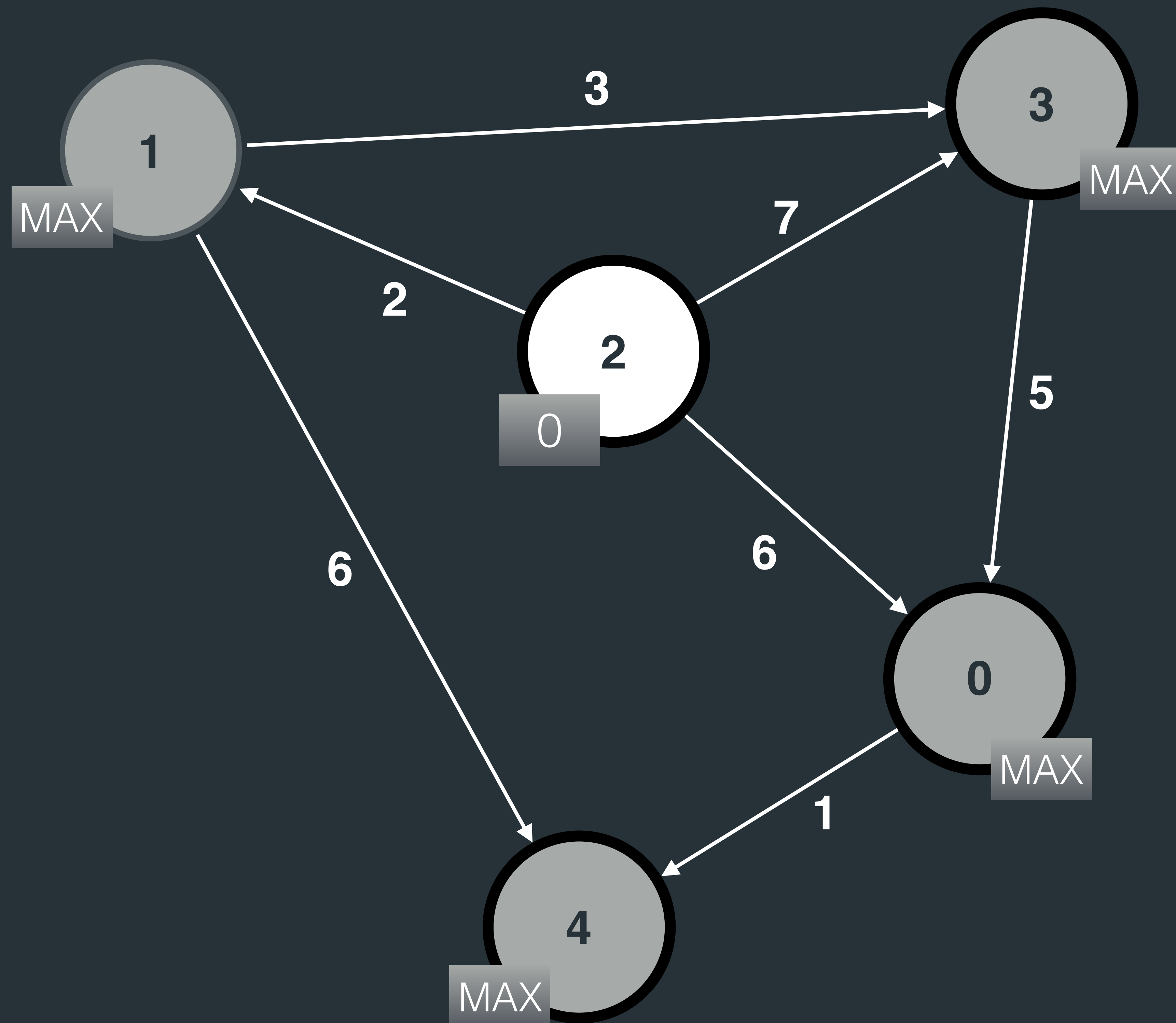
- If you have an unweighted graph, just use BFS

Dijkstra's

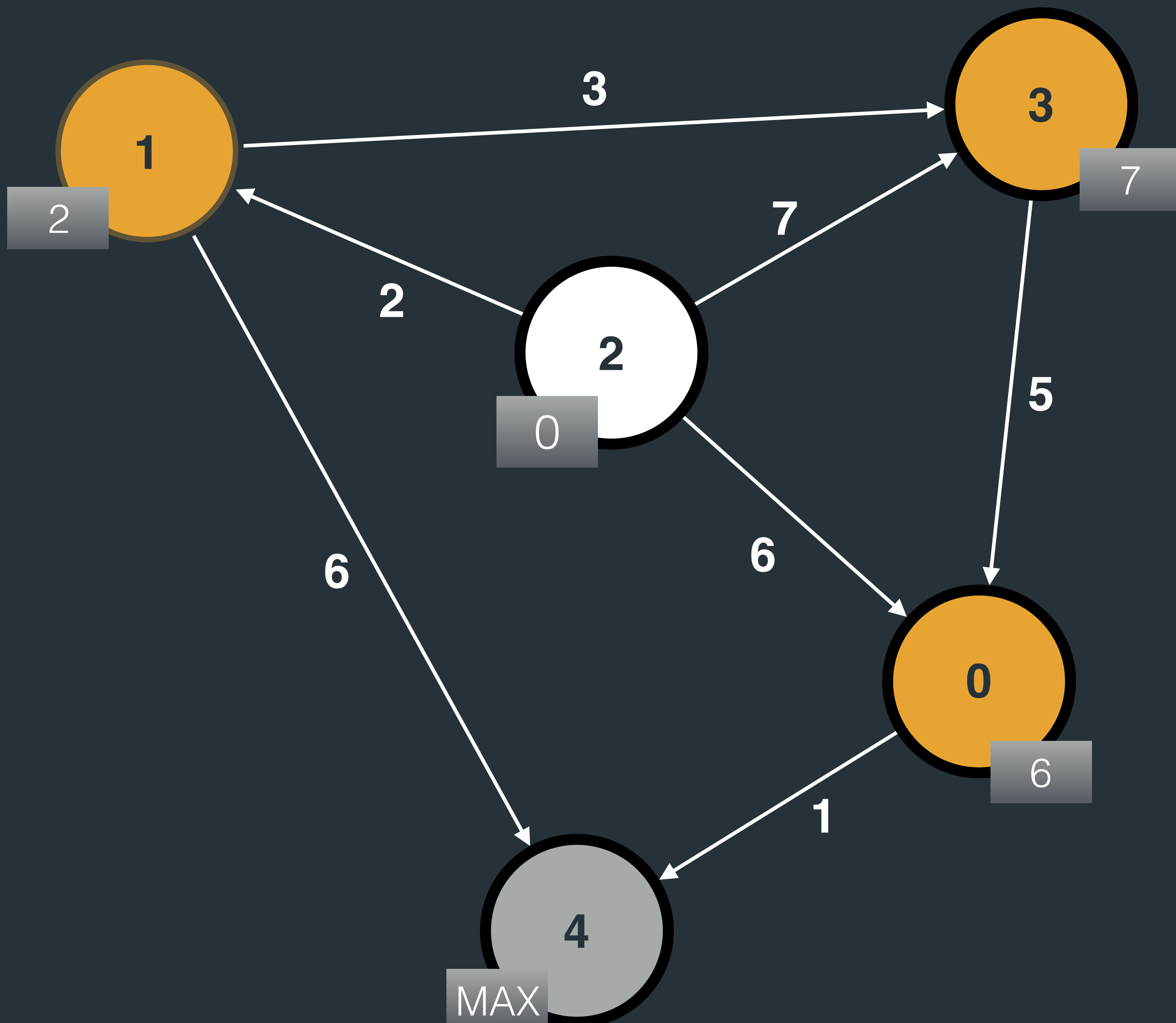
- For weighted graphs
- Uses a priority queue
- $O(E + V \log V)$ time complexity

Dijkstra’s

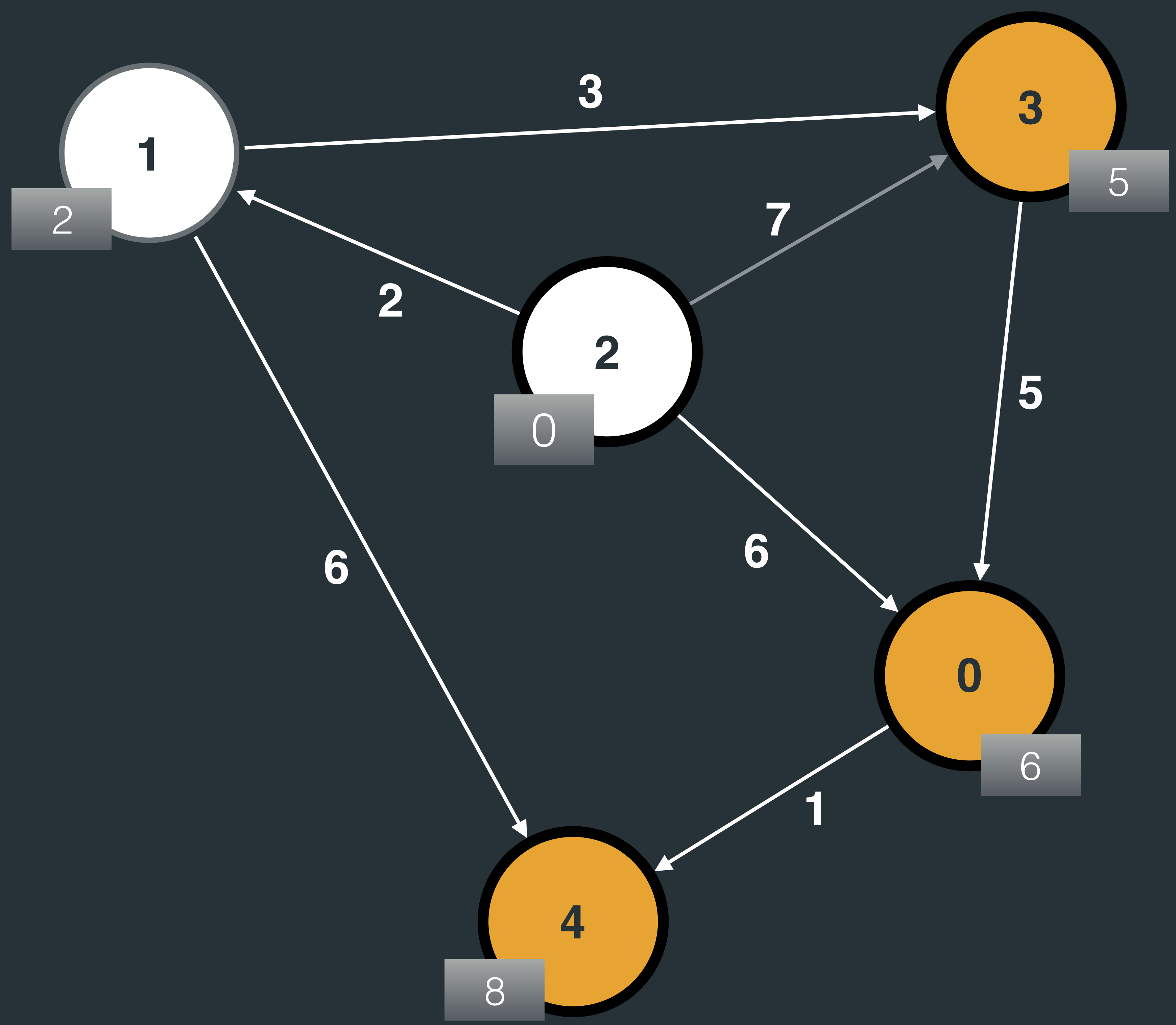
- Assign the distance of all nodes to the initial node to infinity
- Add the initial node to the PQ
- Compare the top node in the PQ to all nodes connected. Add to PQ if the distance is less
- Repeat until PQ is empty

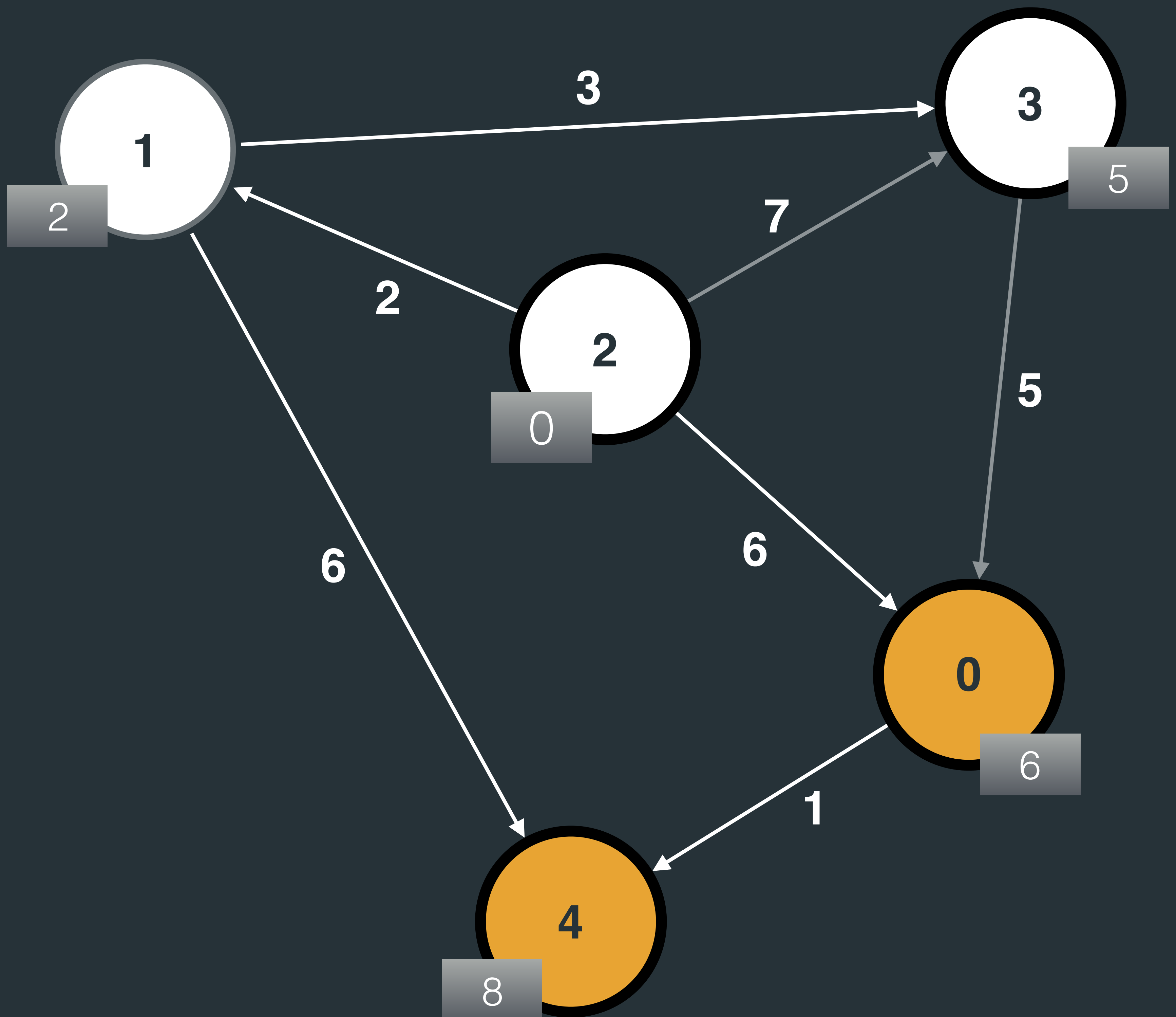


1
MAX
3
3
MAX
7
2
2
0
5
6
6
0
MAX
1
4
MAX

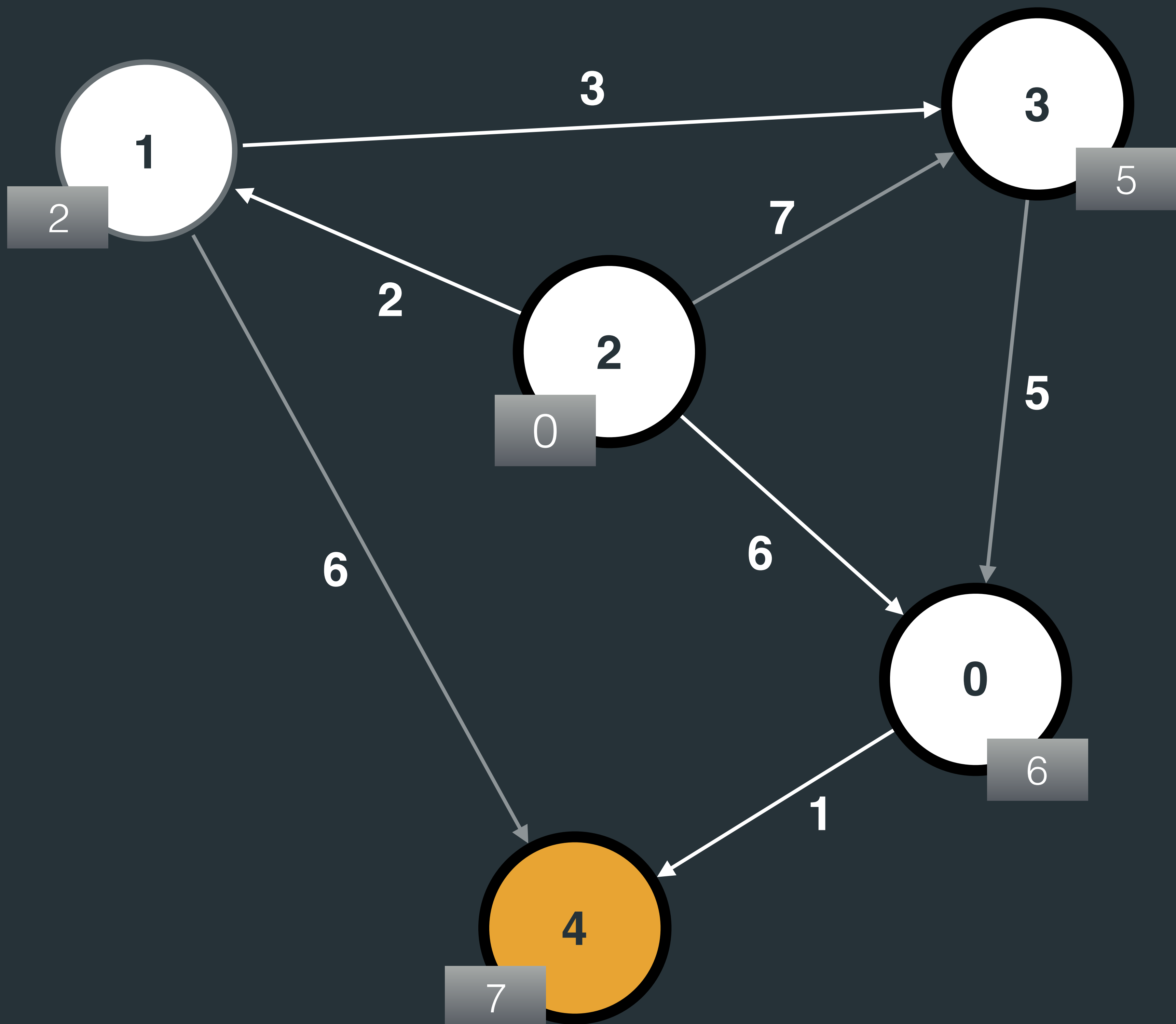


1
2
3
3
7
7
2
2
0
5
6
6
0
6
1
4
MAX

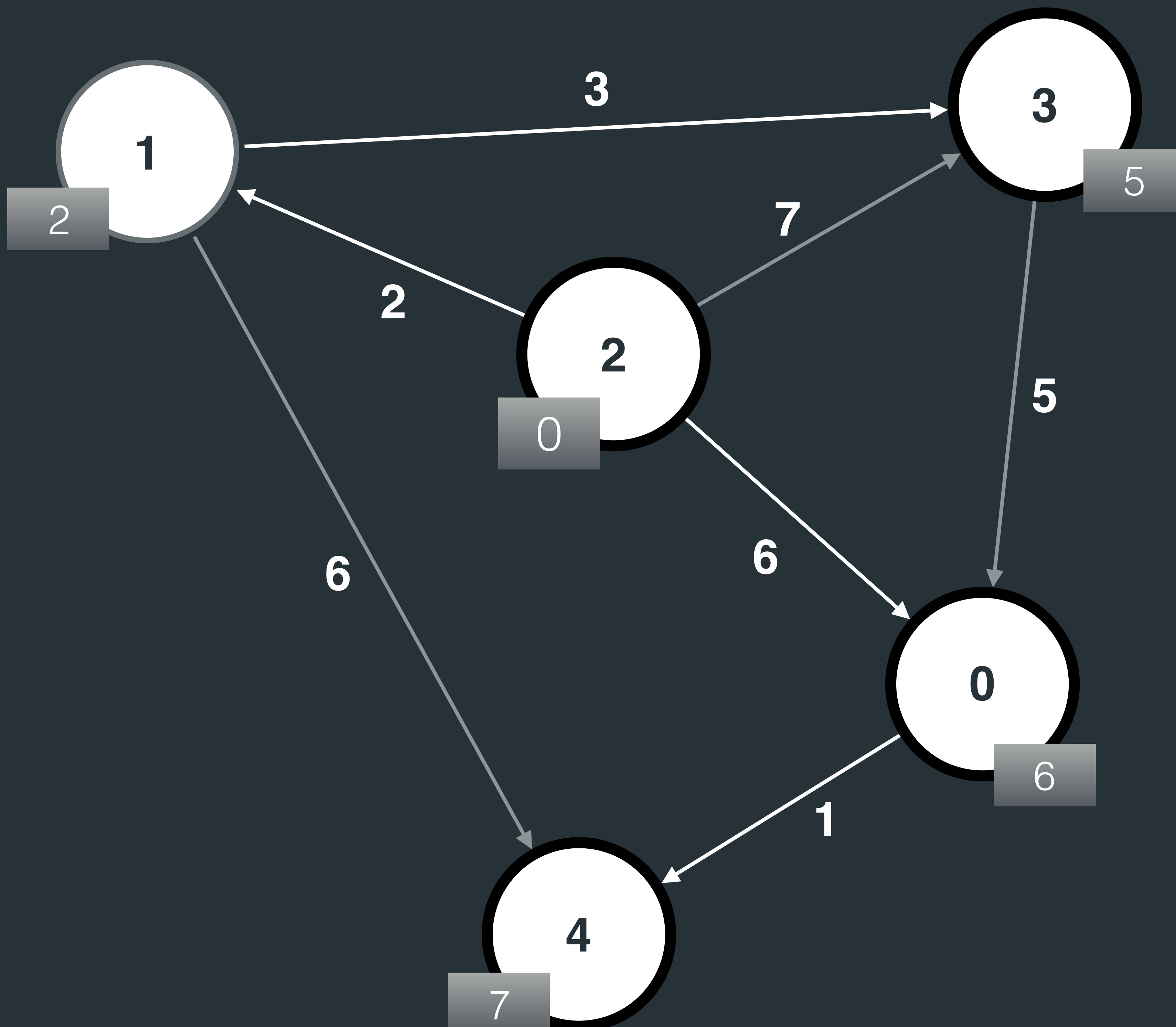

3
1
2
7
2
3
5
2
0
5
6
6
0
6
1
4
8



1
2
3
5
2
0
0
6
4
8
3
7
2
6
6
5
1



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


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

Negative weights

- If there are no negative weight *cycles* Dijkstra's will work

- SSSP with negative cycles requires Bellman Ford algorithm

Bellman-Ford

- Simple concept : relax all edges V-1 times in arbitrary order
- Simple to code : three for loops and two statements
- Complexity: O(VE) complexity (stored as an Adjacency List)
- If you have a small graph even without negative cycles much faster to code

All-Pairs Shortest Path

- Motivation: Given a connected, weighted graph want to find *all* possible shortest paths between all vertices in a graph
- Use Floyd-Warshall

Floyd-Warshall

- Uses an Adjacency Matrix
- Can only solve graphs with V <= 400
- O(V^3) — very bad

Floyd-Warshall

```
for (int k = 0; k < V; k++)
    for(int i = 0; i < V; i++)
        for(int j = 0; j < V; j++)
            adjMat[i][j] = min(adjMat[i][j], adjMat[i][k] + adjMat[k][j]);
```

- Might want to replace min() with a check

Shortest Path Remarks

- These algorithms are used for general case graphs
- Problems usually require some form of modification
- Graph problems may be a sub-problem to a much more complex problem

Shortest Path Remarks

Graph Critera	BFS	Dijkstra’s	Bellman Ford	Floyd Warshall
Max size	V, E <= 10M	V, E <= 300k	VE < 10M	V <= 400
Unweighted	Best	Okay	Bad	Bad
Weighted	no	Best	Okay	Bad
Negative weight	no	Okay	Okay	Bad
Negative cycle	no	no	Can detect	Can detect
Small graph	only if unweighted	Overkill	Overkill	Best

Problem - Get Shorty

- Mikael is trapped in a dungeon
- The dungeon is a set of corridors and intersection. Each corridor joins two intersections.
- Each corridor has a “factor weapon” which reduces the size of its target to a factor of *f* of it’s original size
- Goal: Make it through the dungeon while losing as little size as possible

Problem - Get Shorty

- at most 20 test cases
- n = number of intersections, m= number of corridors
- 2 <= n <= 10 000, 1 <= m <= 15 000
- Time limit = 3 seconds

Problem - Get Shorty

- each line has x, y, f indicating that corridor x,y has a weapon factor of f
- Intersections numbered n to n-1, goal is located at intersection n-1
- Output: a single line with four decimals indicating how big of a fraction Mikael will be left when he reaches the exit in the best possible path

Problem - Get Shorty

- Solution: Modified Dijkstra's — reverse order, shrinking is multiplicative
- Time complexity O((30 000) + (10 000log10 000) = ~ 10^5 times 20 test cases = ~ 10^6

```java
//Start Dijkstra's
double[] dist = new double[n];
Arrays.fill(dist, 0.0);
pq.add(new Pair(0 , 1.0));
dist[0] = 1.0;
//PQ is sorted in the opposite order
while(!pq.isEmpty()){
    Pair top = pq.poll();

    if(Math.abs(top.y - dist[top.x]) > 0.0)
        continue;

    for(Pair p : adjList.get(top.x)) {
        if(top.y * p.y > dist[p.x]){ //multiplicative, >
            dist[p.x] = top.y * p.y;
            pq.add(new Pair(p.x, dist[p.x]));
        }
    }
}
System.out.printf("%.4f%n", dist[n-1]);
```

- adjList = ArrayList<ArrayList<Pair>
- Pair = custom int/double class
- PQ : has a custom Pair comparator

References

- open.kattis.com/problems/getshorty

- Competitive Programming 3 - Steven Halim & Felix Halim