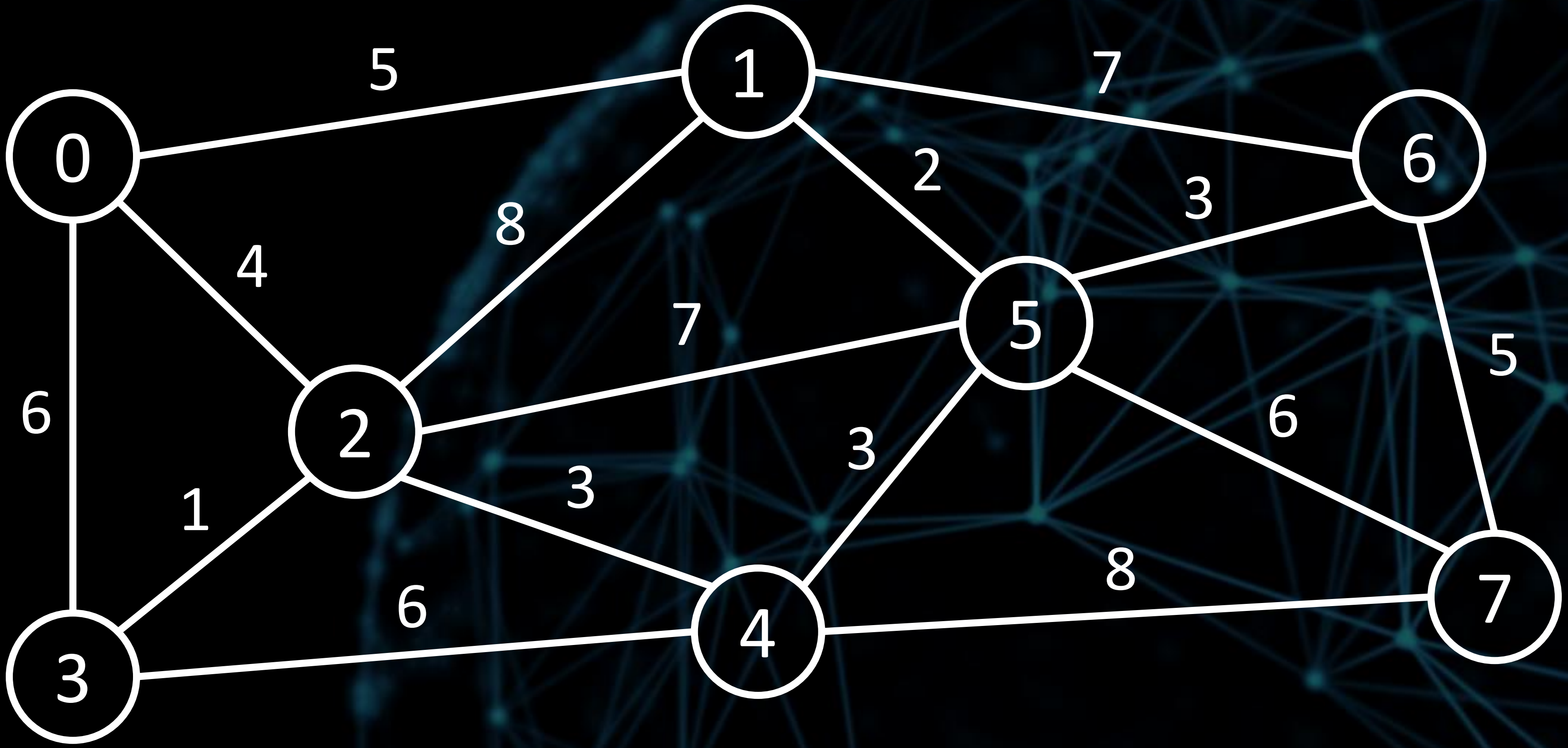
Dijkstra’s Algorithm

dist contains the **shortest known distance** from the starting vertex to each vertex. The distance from the starting vertex to itself is (unsurprisingly) 0.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

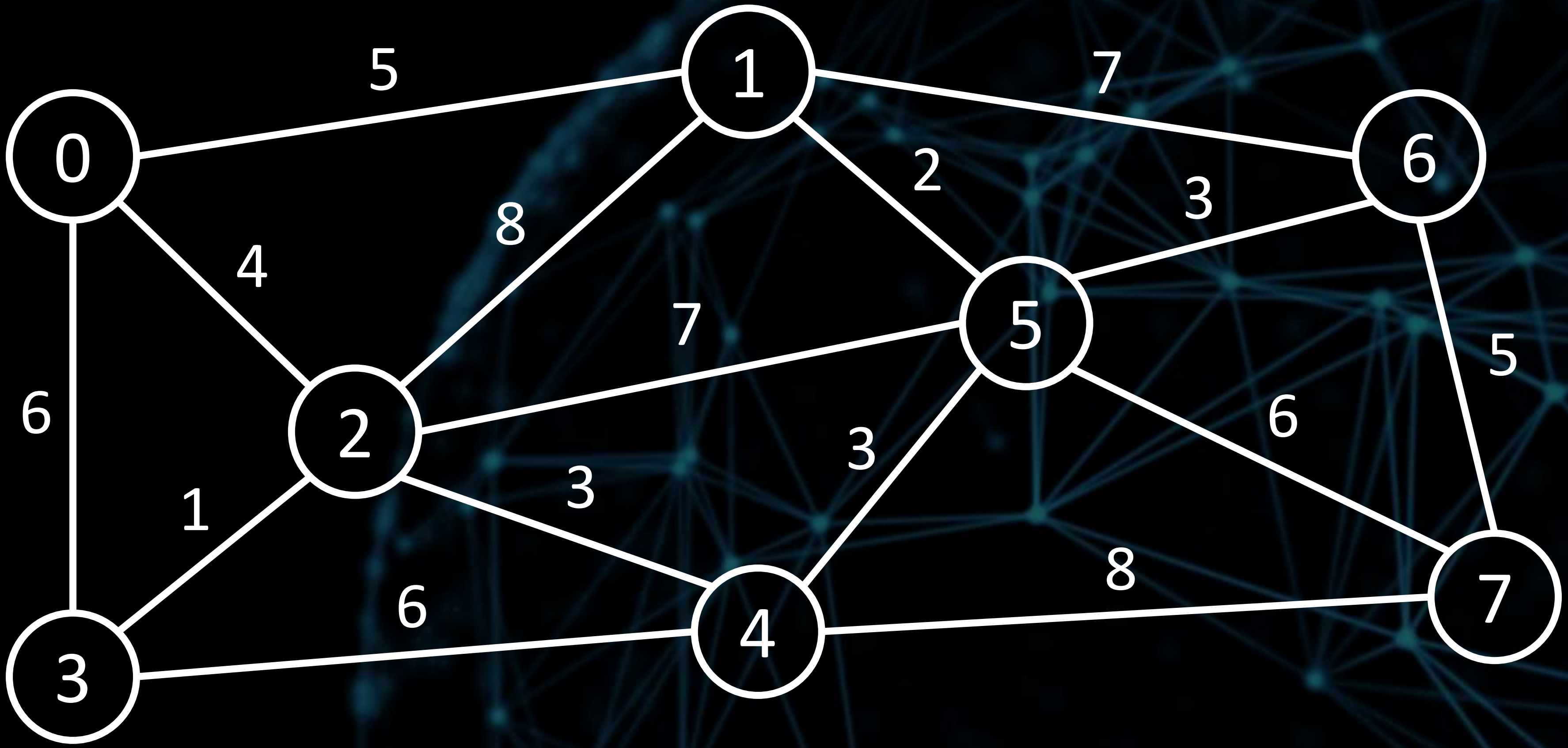
vSet = {0, 1, 2, 3, 4, 5, 6, 7}



pred contains the predecessor of each vertex on the shortest path from the starting vertex (0) to that vertex.

	0	1	2	3	4	5	6	7
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

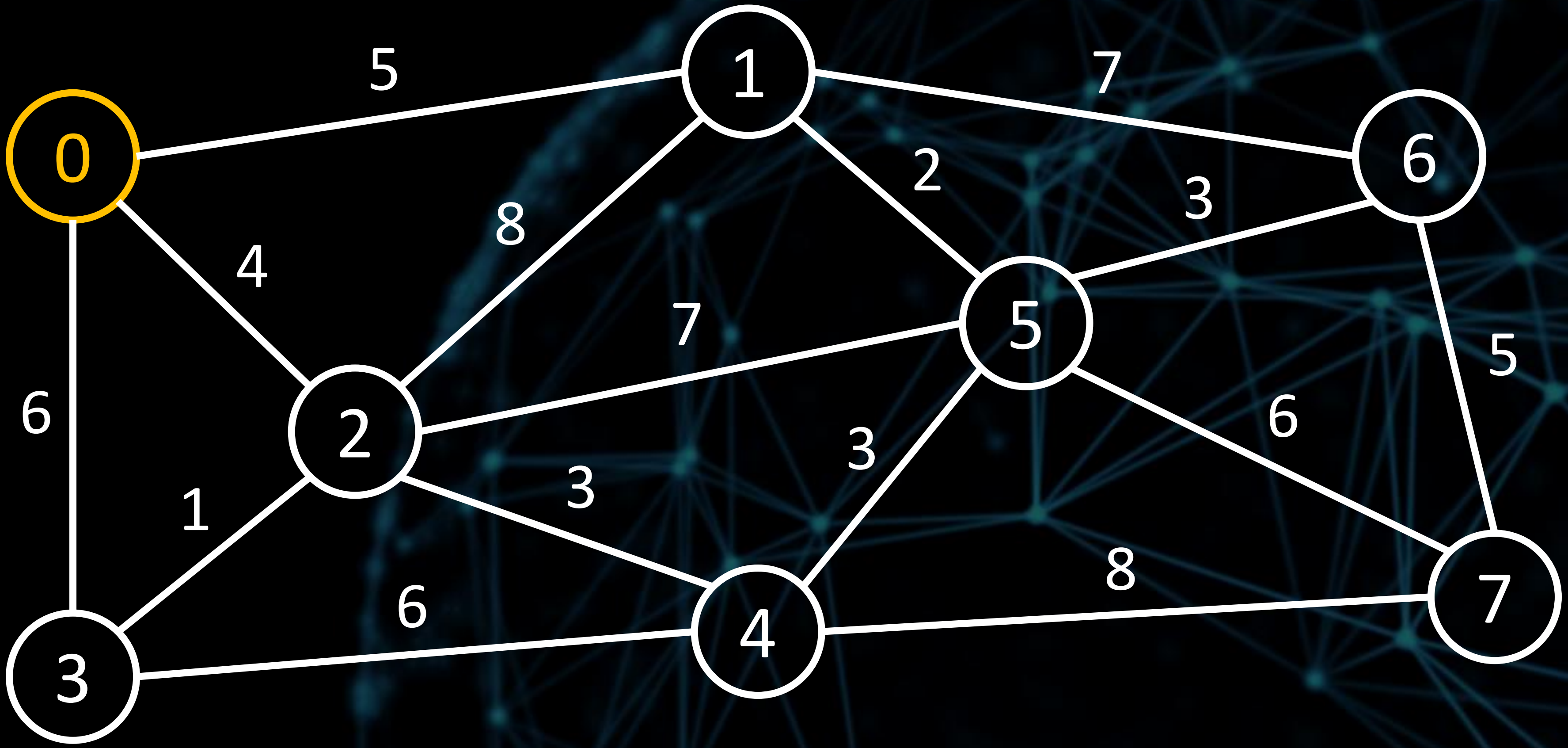
vSet = {0, 1, 2, 3, 4, 5, 6, 7}



Step 1:
Choose the vertex from vSet that is closest to the starting vertex, and remove it from vSet.
That vertex is 0.

	0	1	2	3	4	5	6	7
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

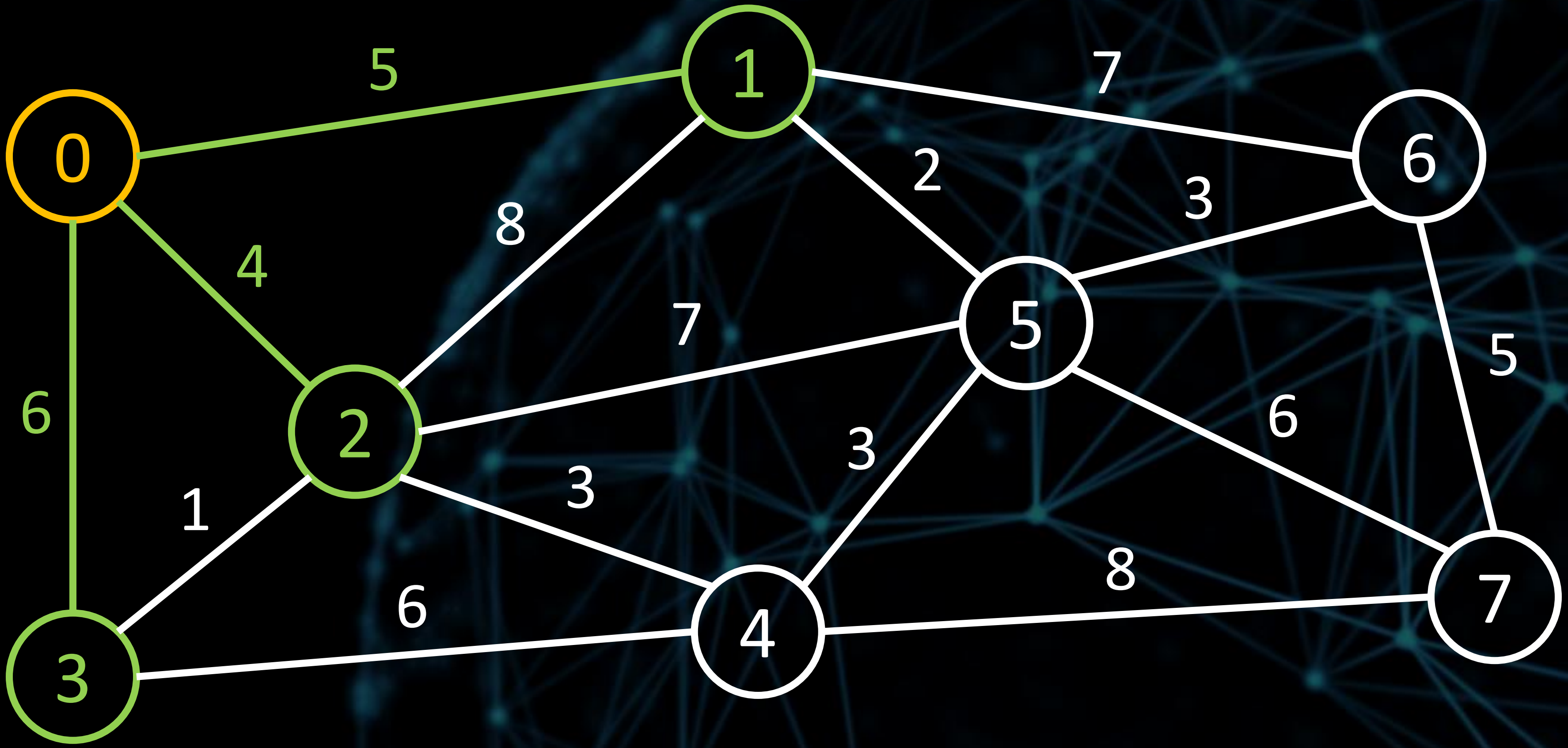
vSet = {0, 1, 2, 3, 4, 5, 6, 7}



Step 2:
For each neighbour of vertex 0, we check if there is a shorter path to the neighbour **via 0**.

	0	1	2	3	4	5	6	7
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

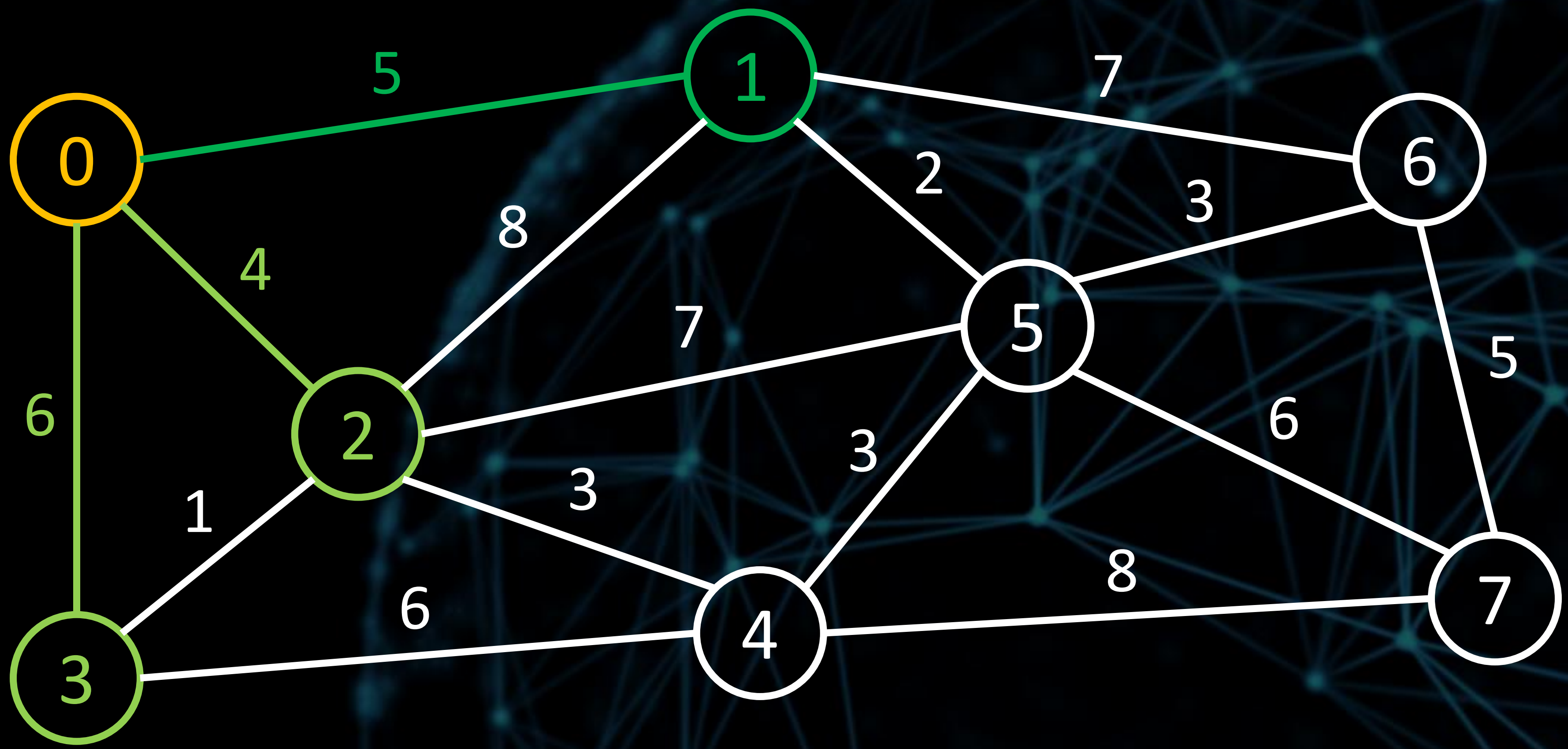
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
Let's start with the neighbour 1.

	0	1	2	3	4	5	6	7
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

vSet = {1, 2, 3, 4, 5, 6, 7}



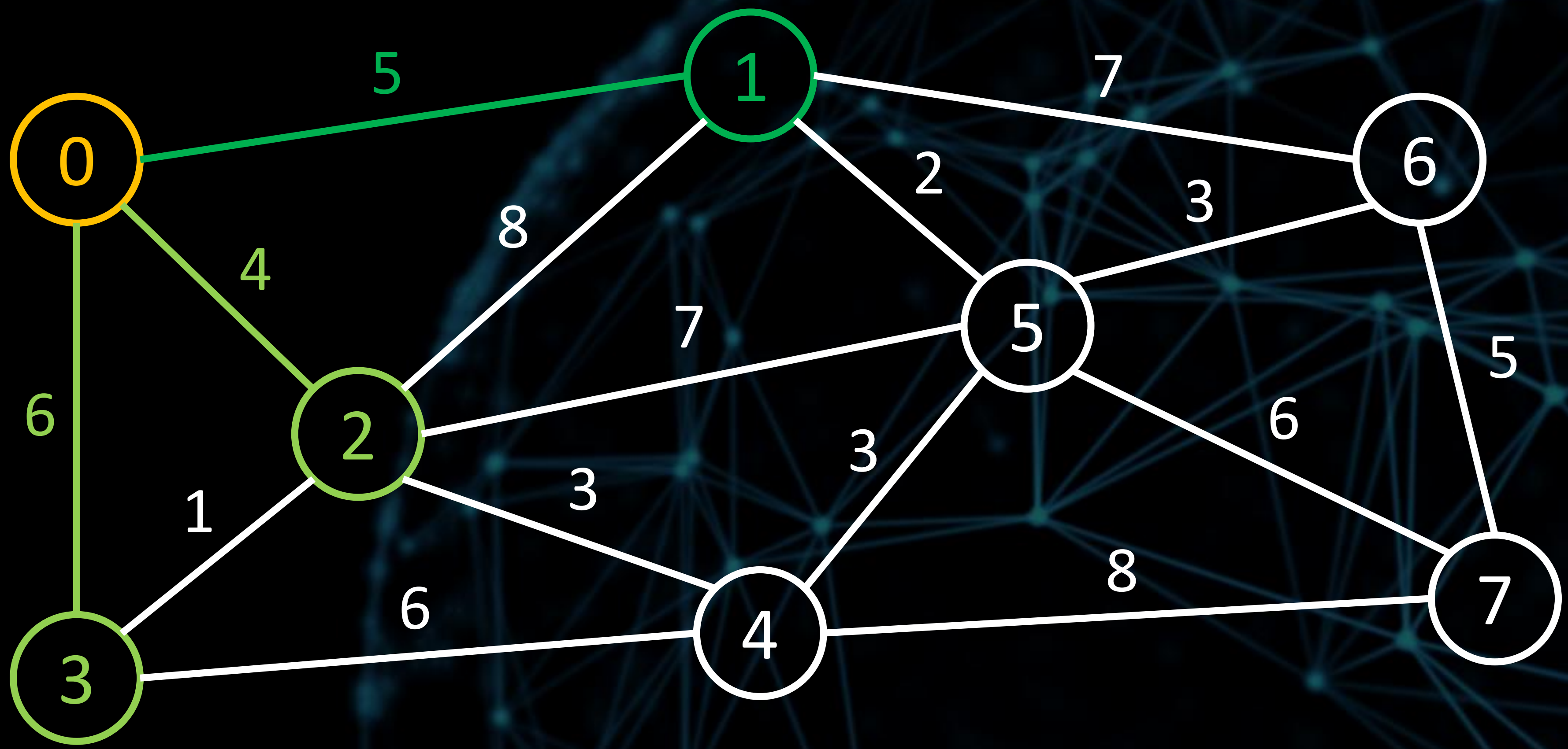
Step 2:

The distance from the starting vertex to 1 **via 0** is the sum of the shortest known distance from the starting vertex to 0 and the weight of the edge from 0 to 1.

0 + 5 = 5

	0	1	2	3	4	5	6	7
dist	0	∞	∞	∞	∞	∞	∞	∞
pred	-1	-1	-1	-1	-1	-1	-1	-1

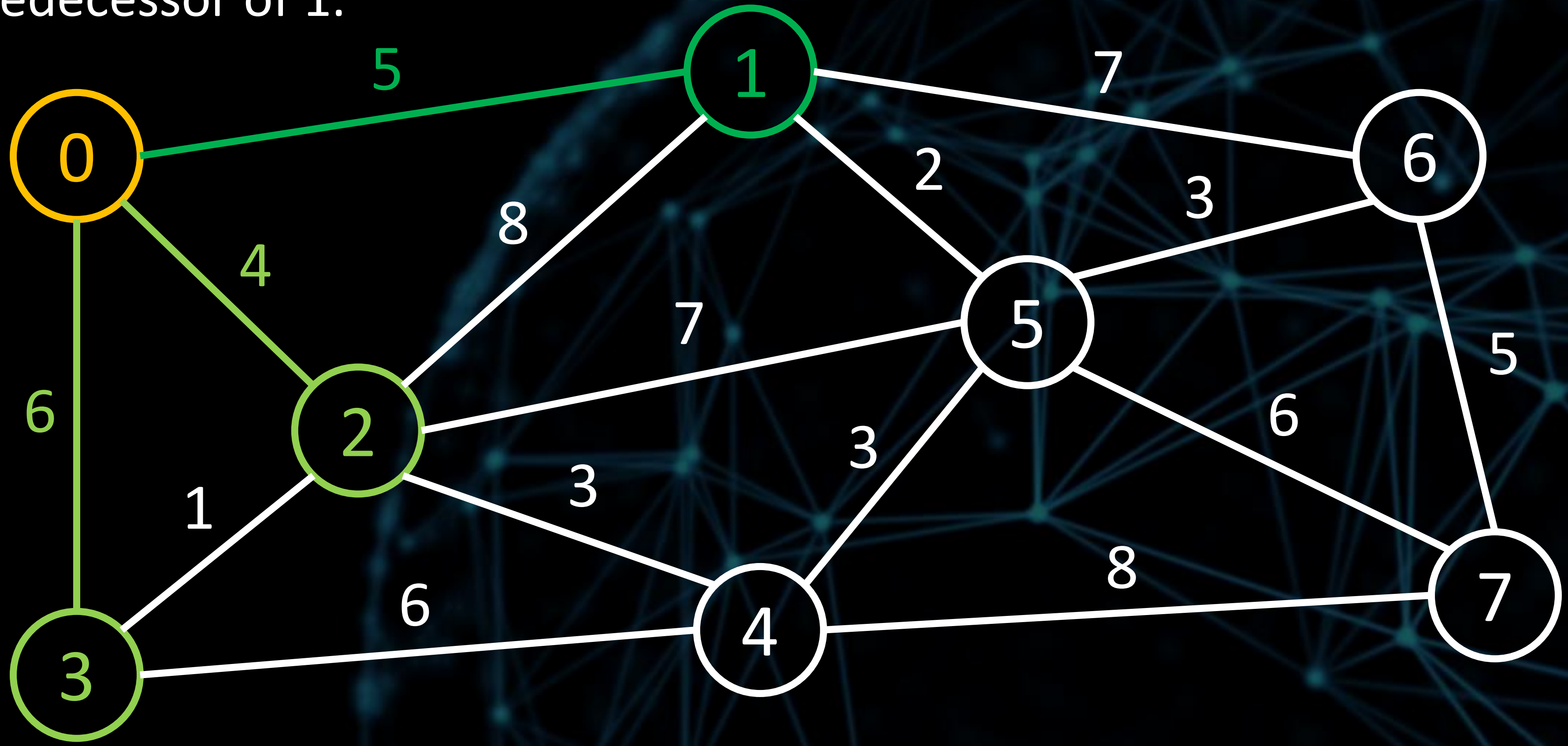
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
If this distance is smaller than the **currently known shortest distance** to 1, we update the distance array. $5 < \infty$, so we update the distance to 1 in the distance array. We also store the predecessor of 1.

	0	1	2	3	4	5	6	7
dist	0	5	∞	∞	∞	∞	∞	∞
pred	-1	0	-1	-1	-1	-1	-1	-1

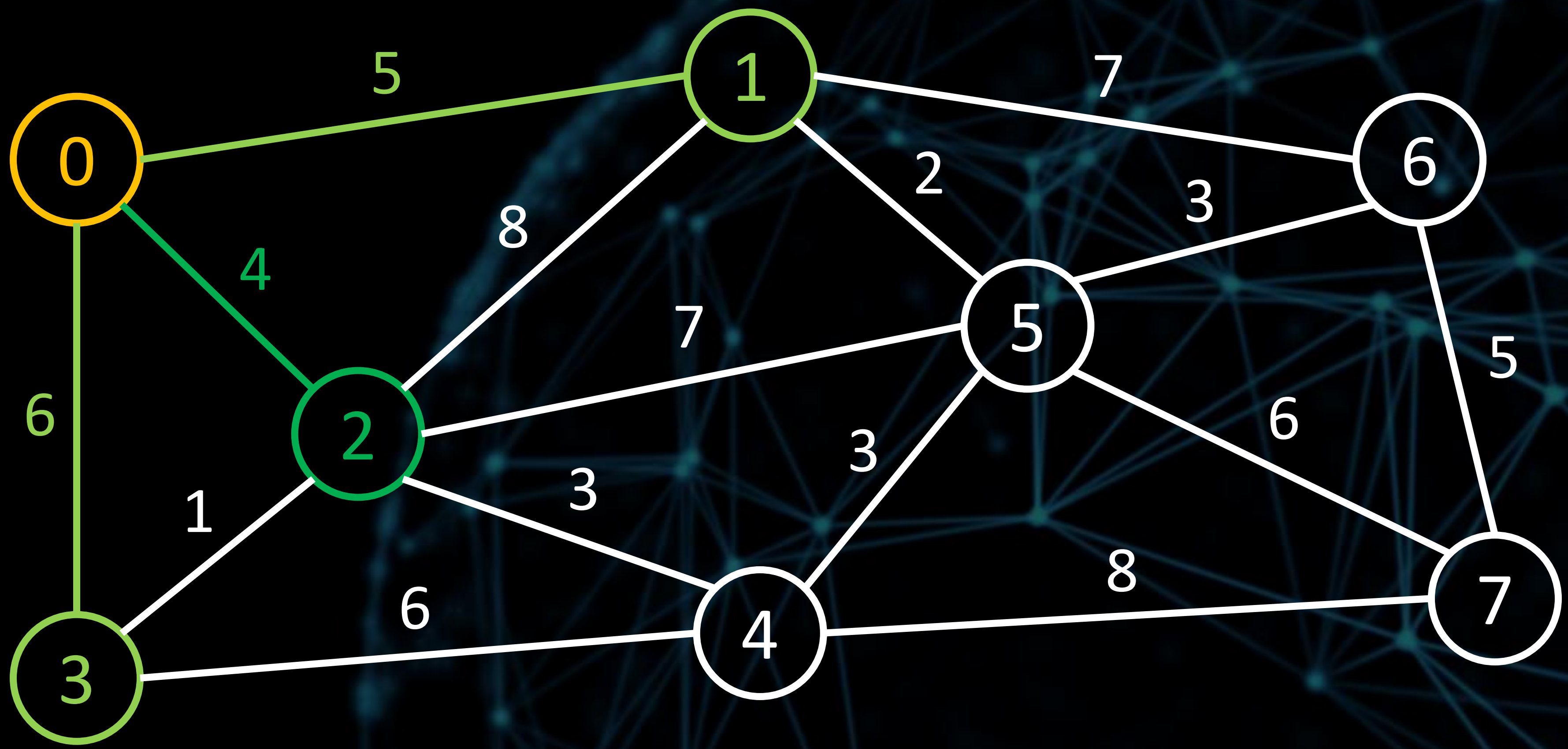
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
Now let's look at the neighbour 2.

	0	1	2	3	4	5	6	7
dist	0	5	∞	∞	∞	∞	∞	∞
pred	-1	0	-1	-1	-1	-1	-1	-1

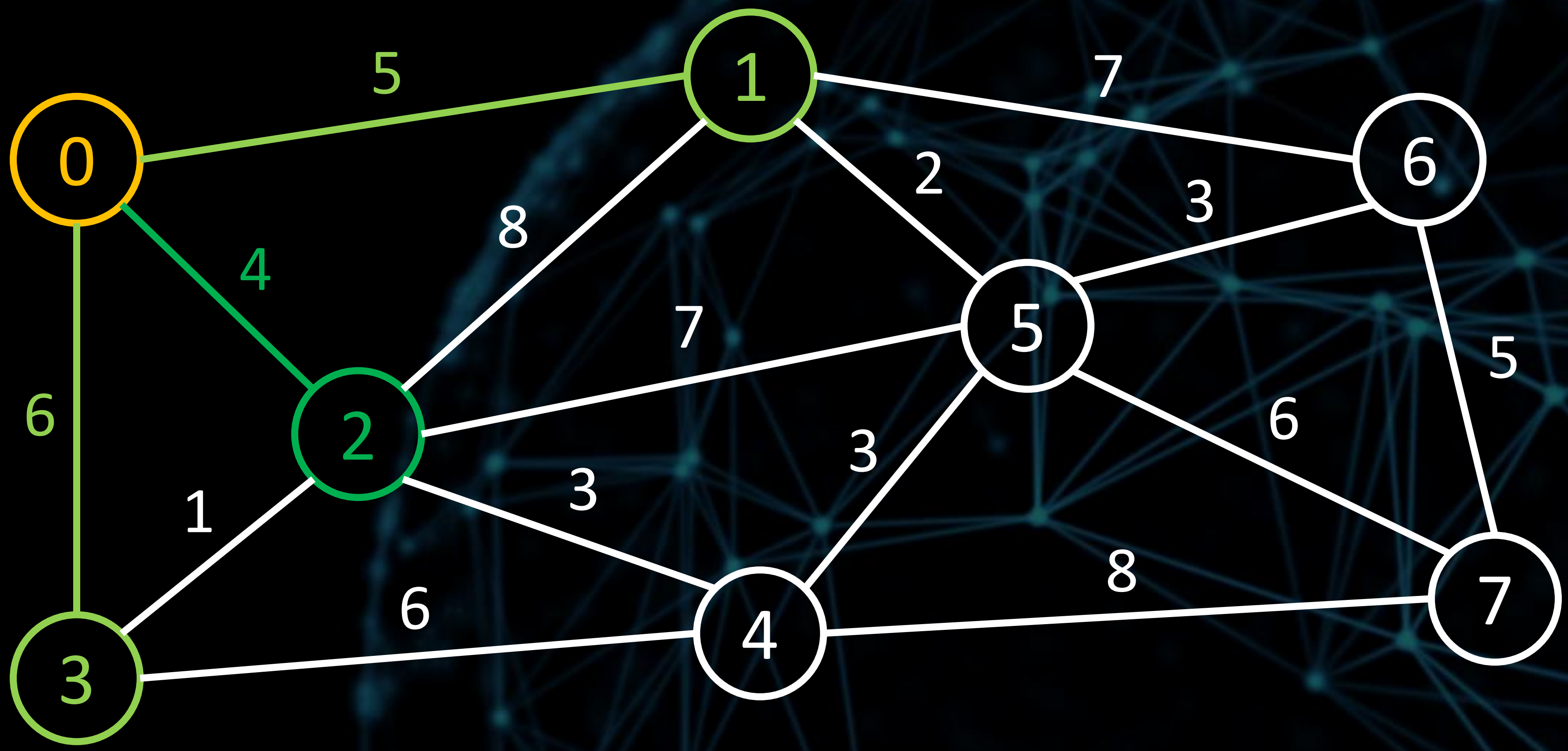
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
The distance from the starting vertex to 2 **via 0** is the sum of the shortest known distance from the starting vertex to 0 and the weight of the edge from 0 to 2.
0 + 4 = 4

	0	1	2	3	4	5	6	7
dist	0	5	∞	∞	∞	∞	∞	∞
pred	-1	0	-1	-1	-1	-1	-1	-1

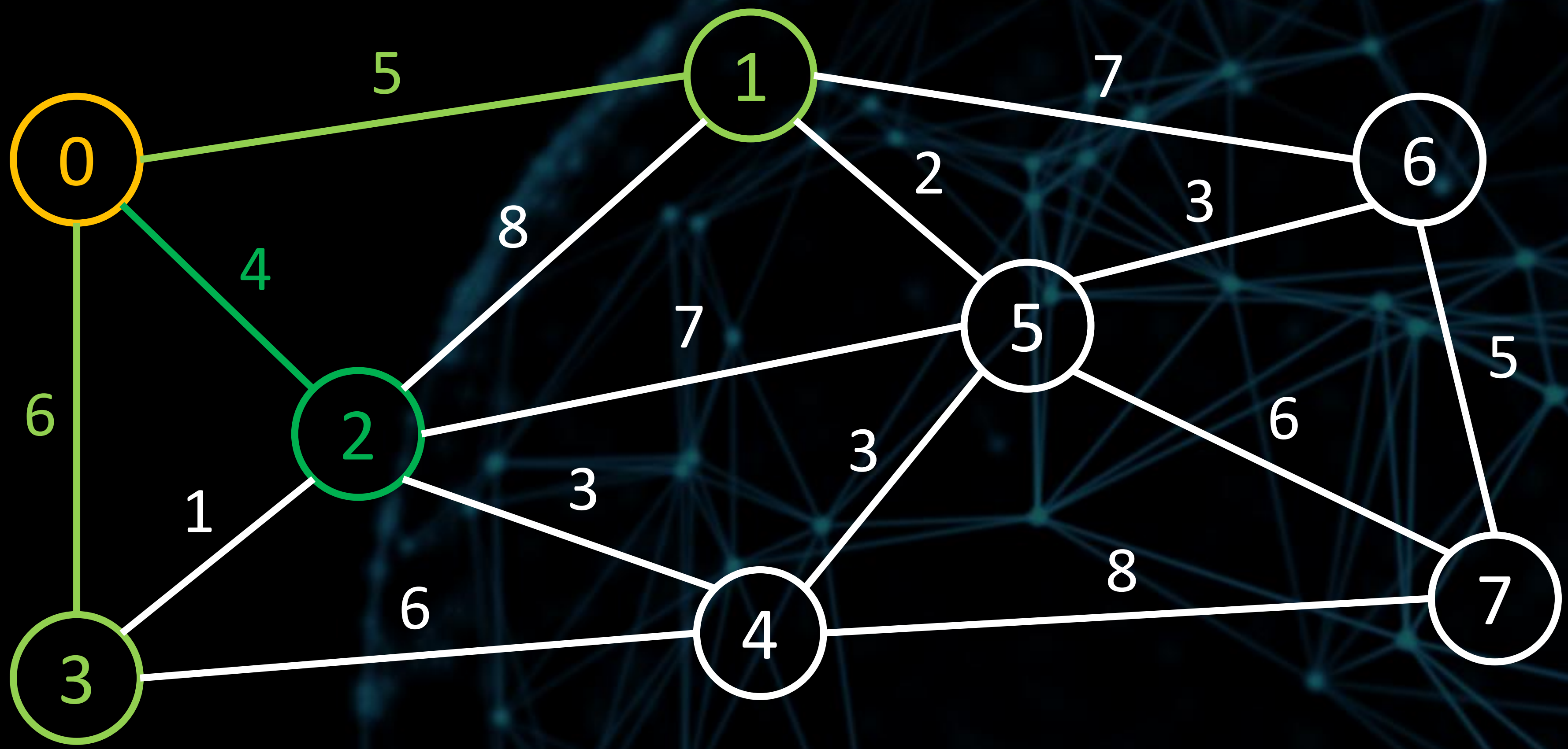
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
This distance (4) is smaller than the **currently known shortest distance** to 2 (∞), so we update the distance and predecessor arrays.

	0	1	2	3	4	5	6	7
dist	0	5	4	∞	∞	∞	∞	∞
pred	-1	0	0	-1	-1	-1	-1	-1

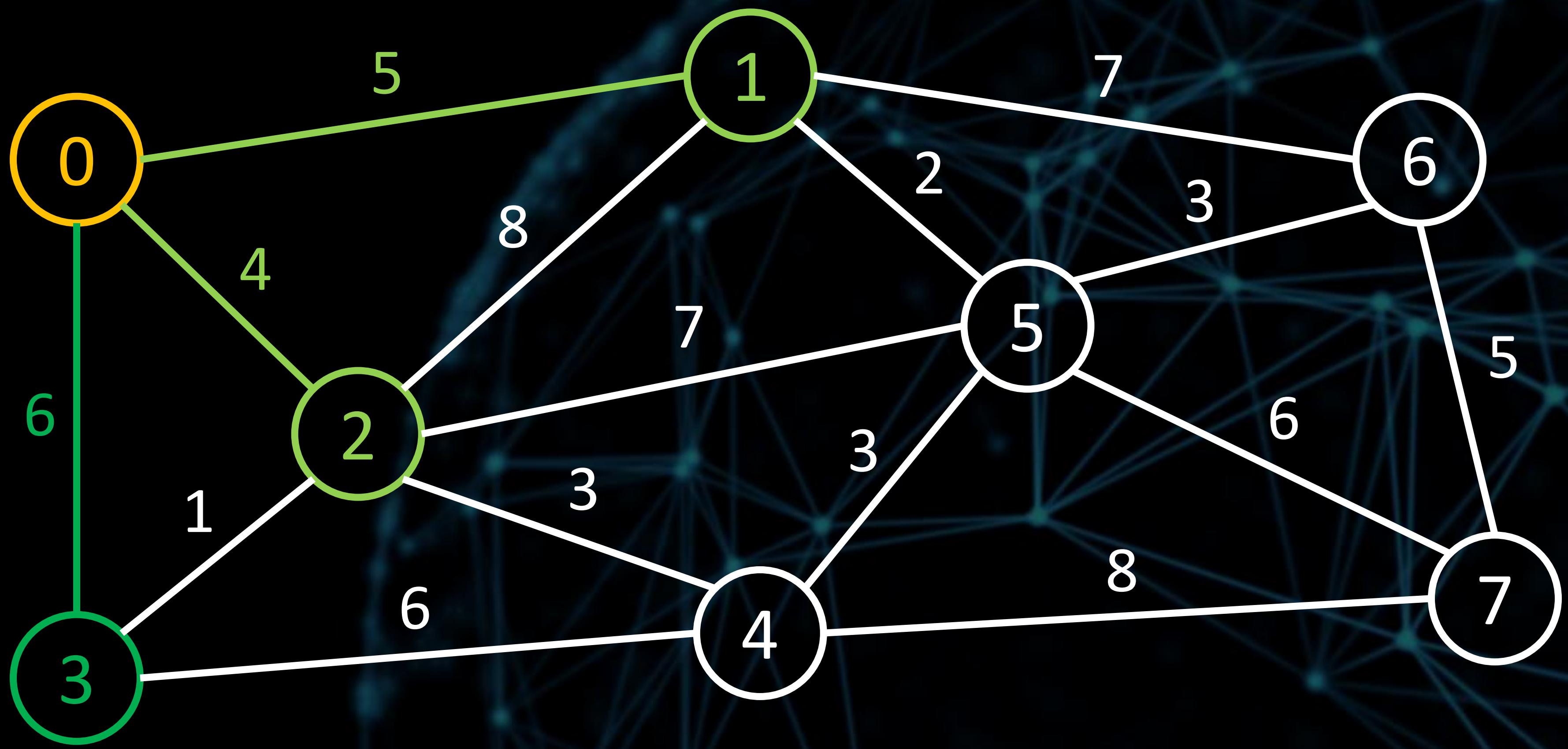
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
Now let’s look at the neighbour 3.

	0	1	2	3	4	5	6	7
dist	0	5	4	∞	∞	∞	∞	∞
pred	-1	0	0	-1	-1	-1	-1	-1

vSet = {1, 2, 3, 4, 5, 6, 7}



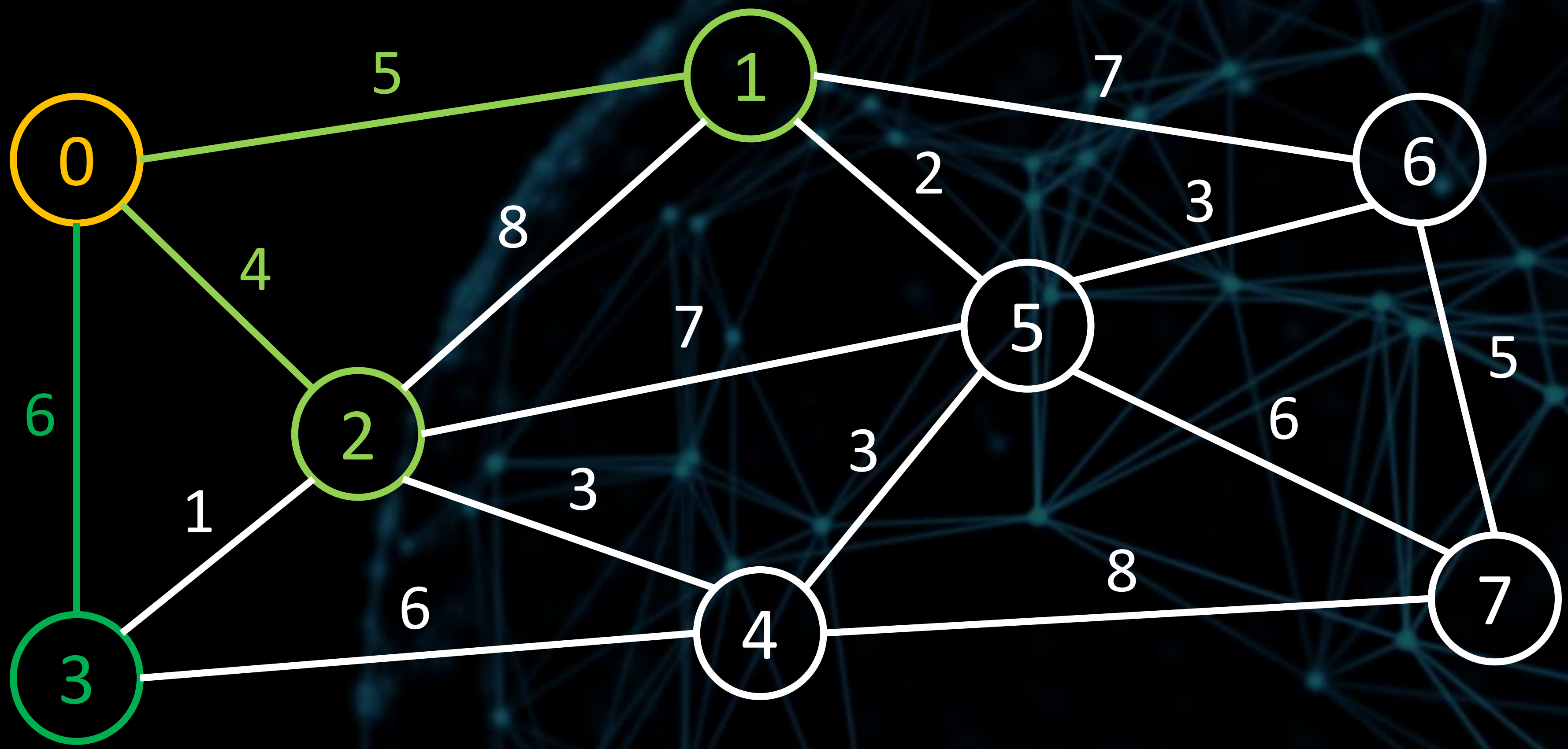
Step 2:

The distance from the starting vertex to 3 **via 0** is the sum of the shortest known distance from the starting vertex to 0 and the weight of the edge from 0 to 3.

0 + 6 = 6

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	∞	∞	∞	∞	∞
pred	-1	0	0	-1	-1	-1	-1	-1

vSet = {1, 2, 3, 4, 5, 6, 7}

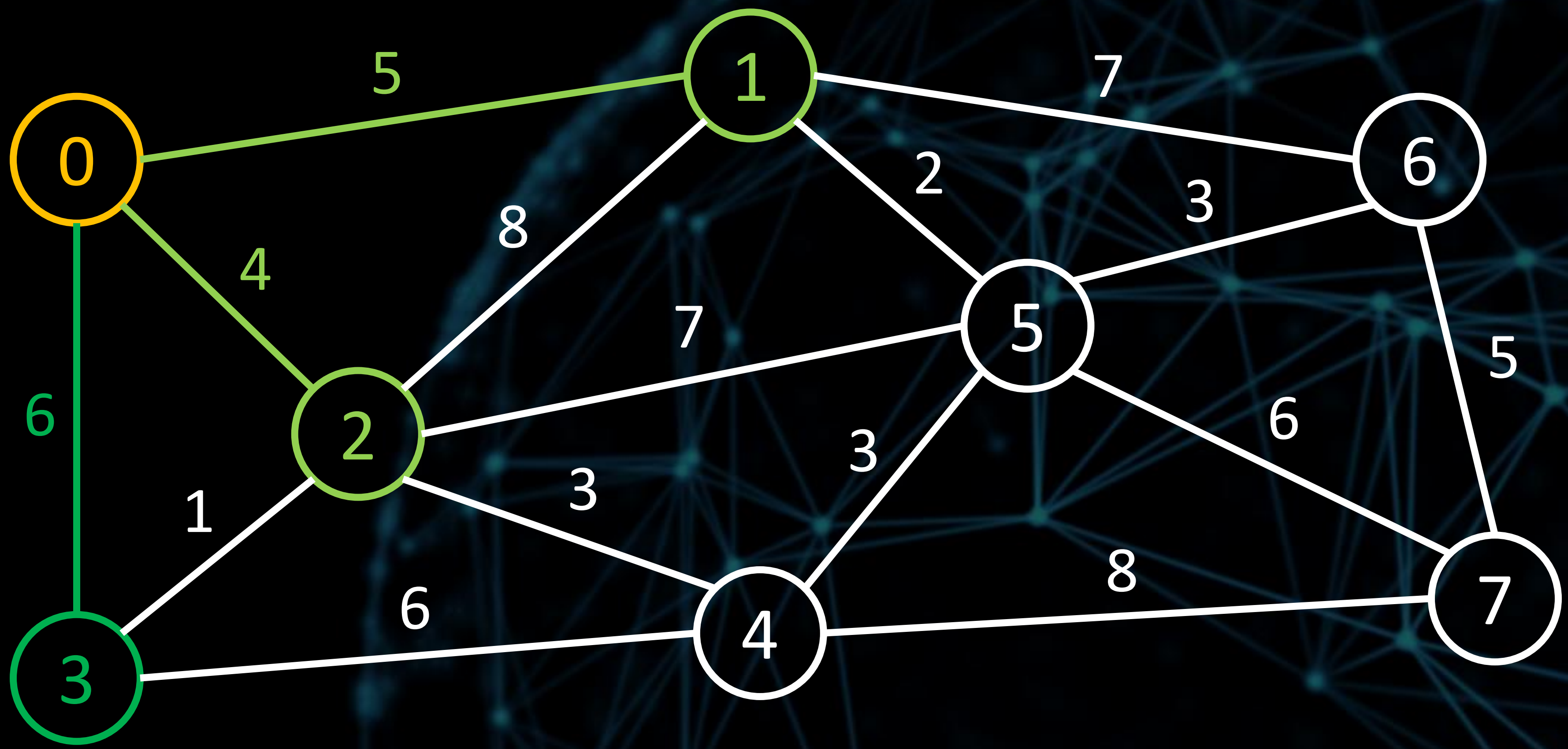


Step 2:

This distance (6) is smaller than the **currently known shortest distance** to 3 (∞), so we update the distance and predecessor arrays.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

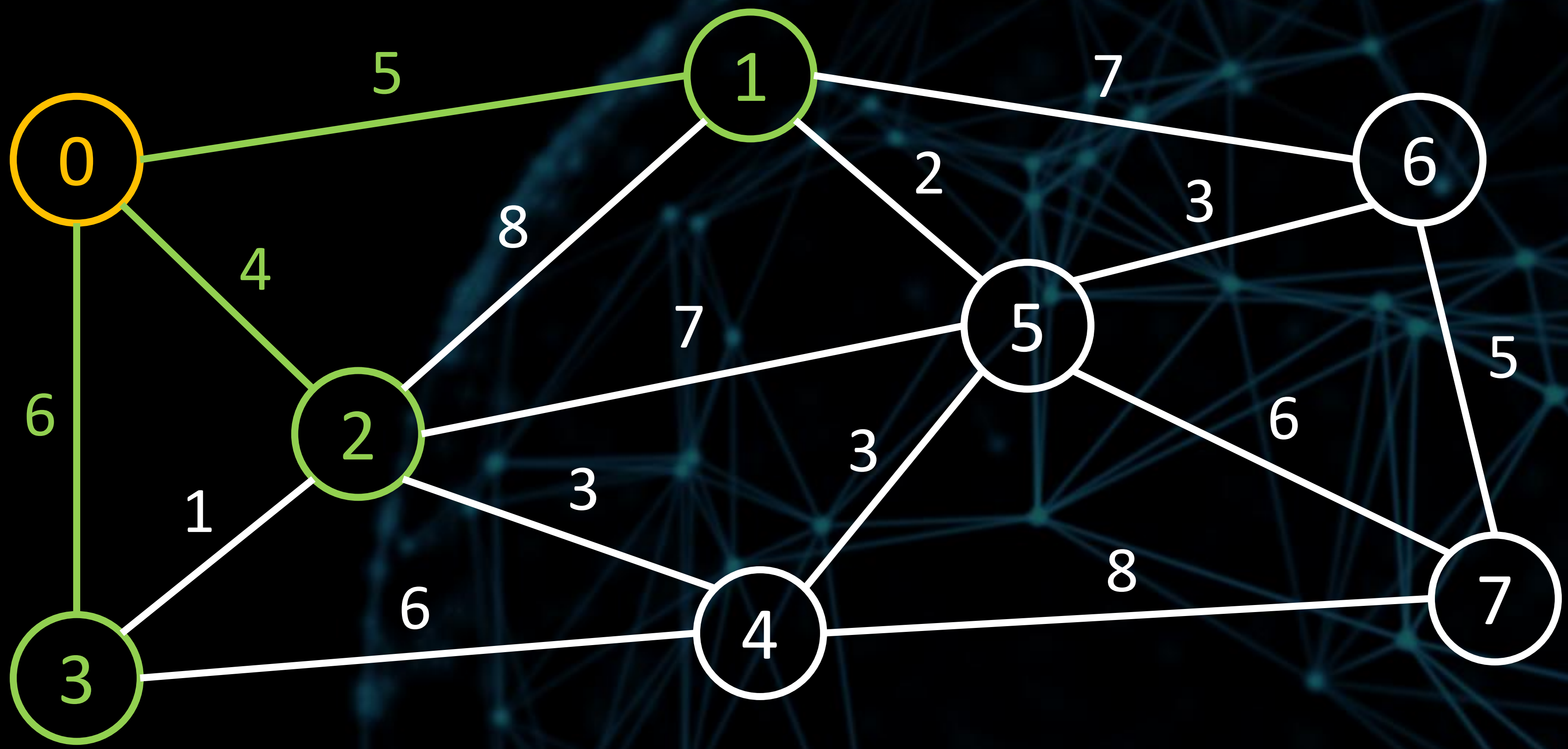
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
We've checked all the neighbours of 0, so we've completed one iteration of the algorithm.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

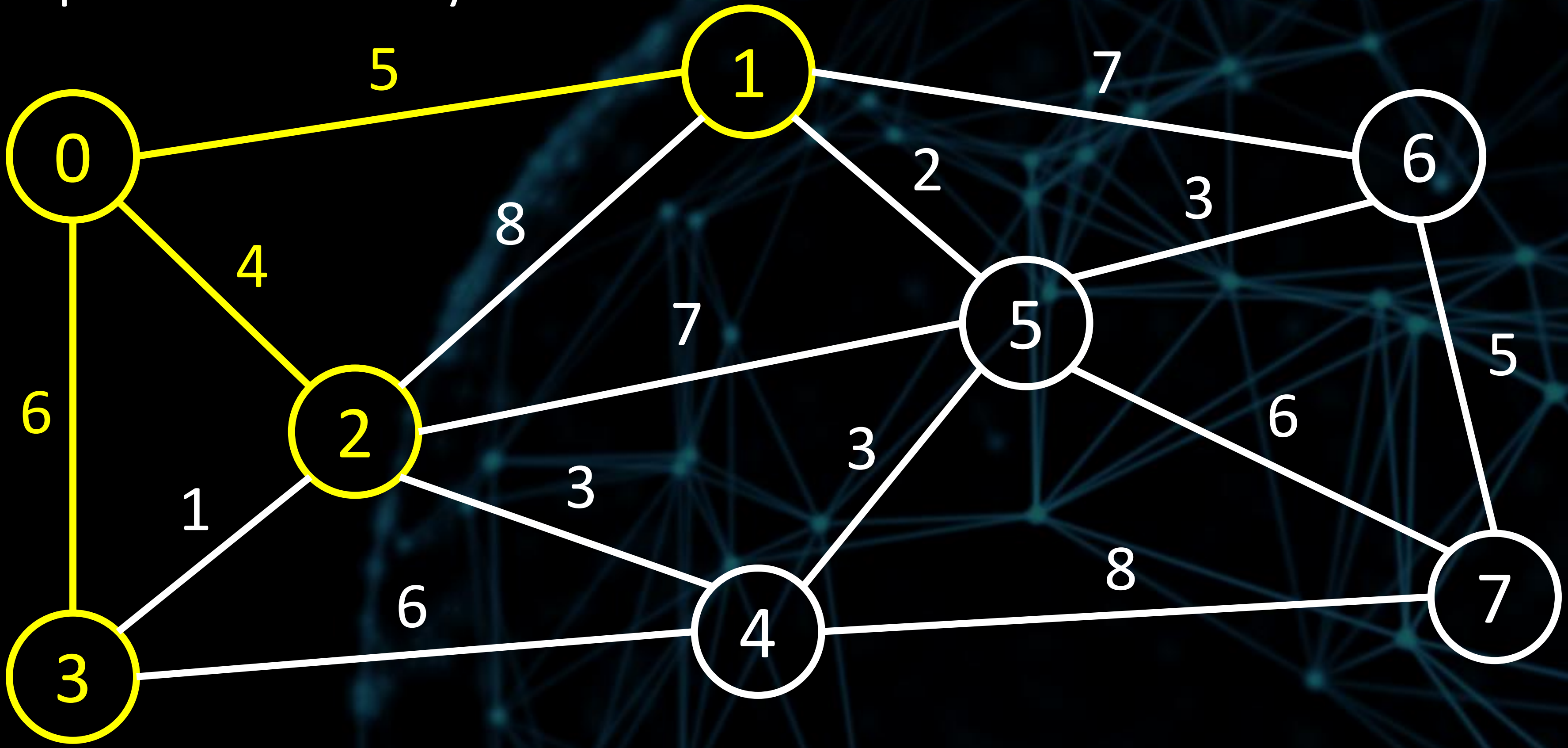
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

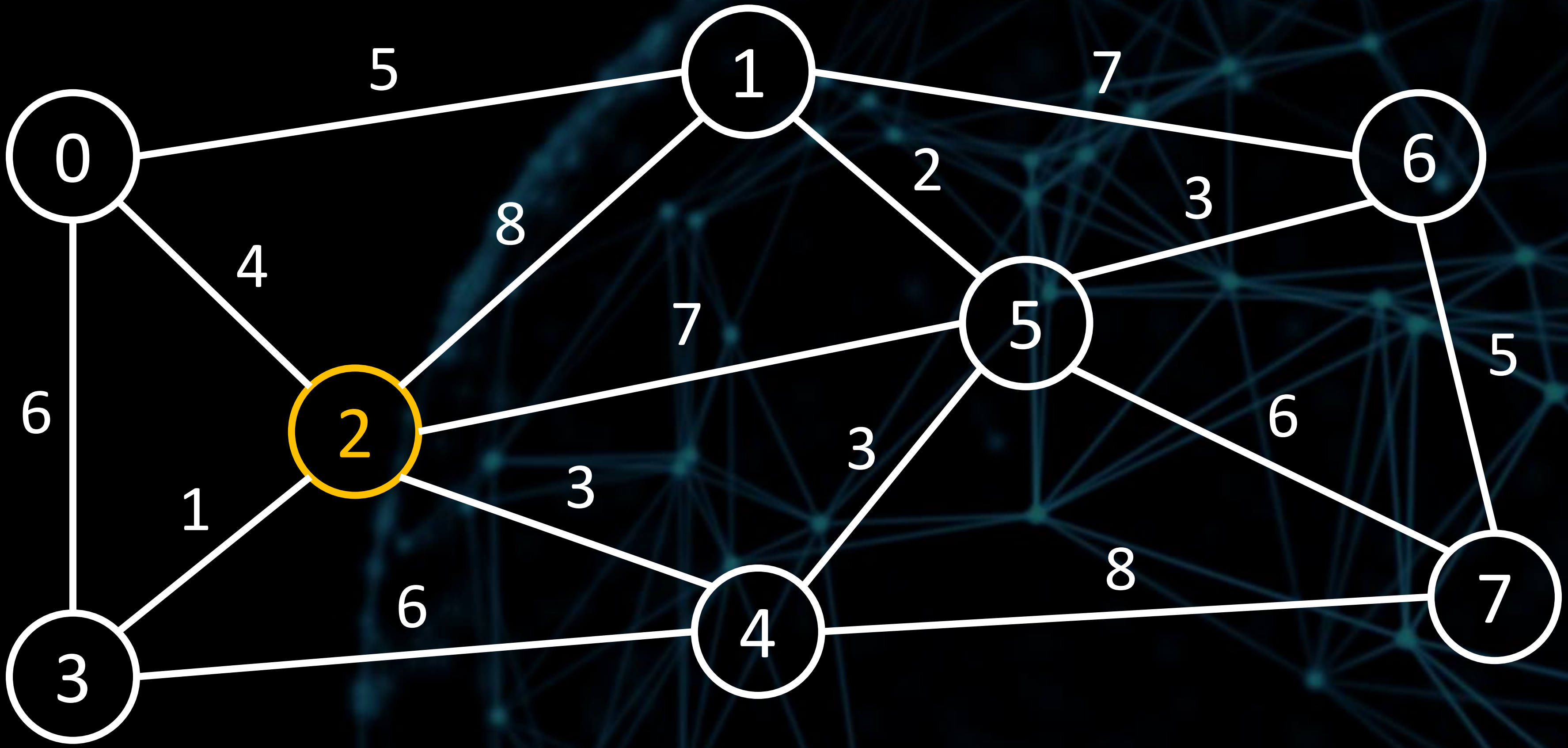
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 1:
Choose the vertex from vSet that is closest to the starting vertex, and remove it from vSet.
That vertex is 2.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

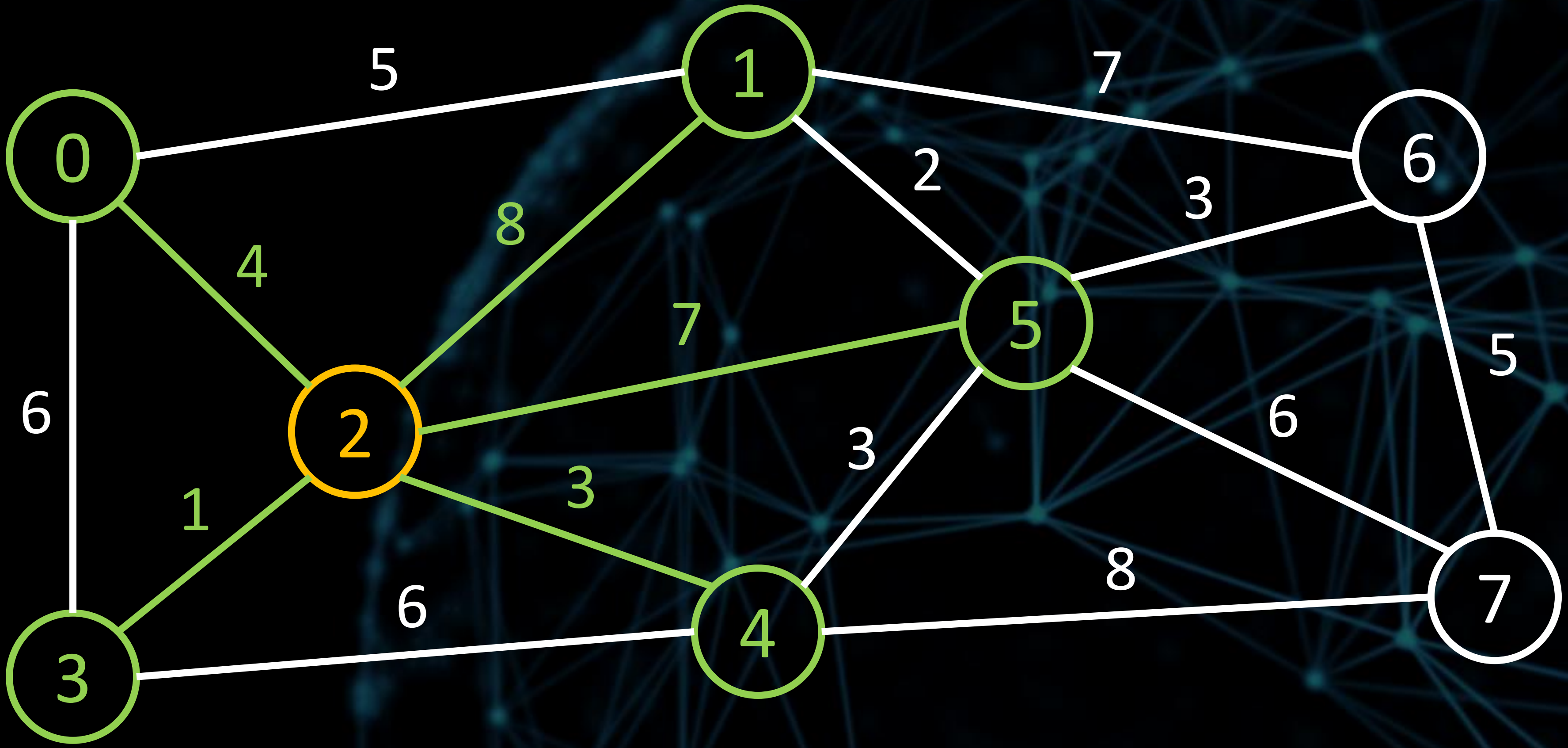
vSet = {1, 2, 3, 4, 5, 6, 7}



Step 2:
For each neighbour of vertex 2, we check if there is a shorter path to the neighbour **via 2**.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

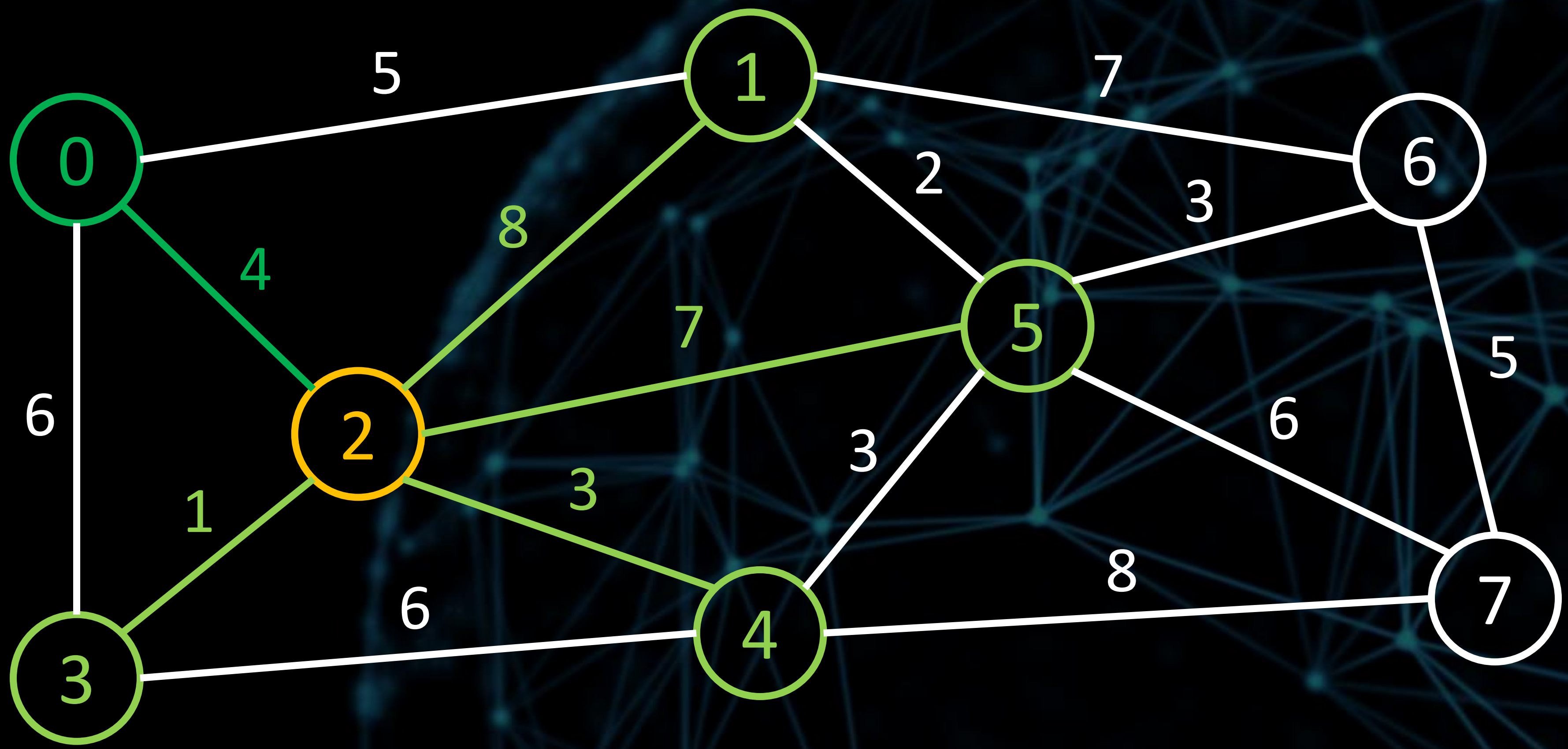
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
Let's start with the neighbour 0.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

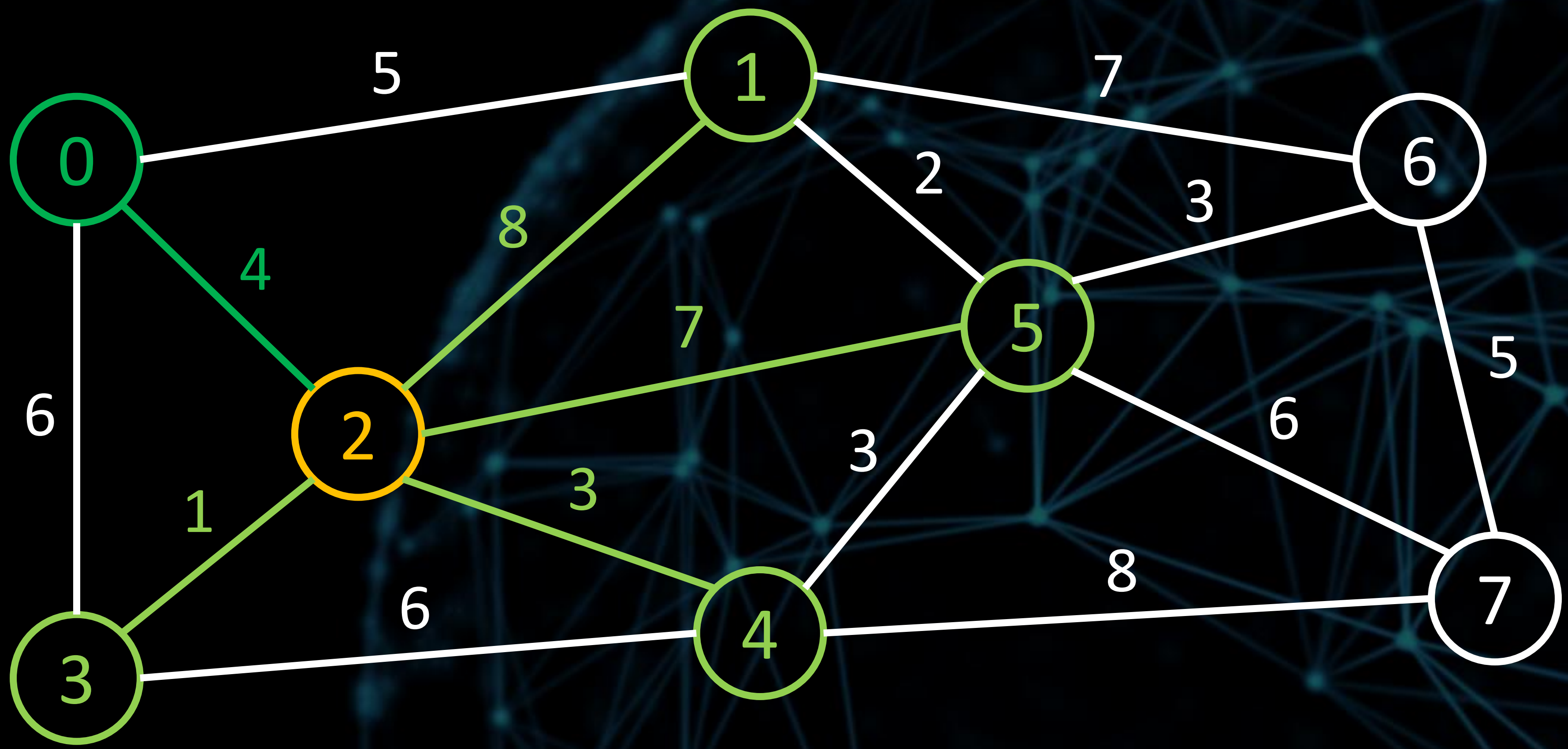
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
The distance from the starting vertex to 0 **via 2** is the sum of the shortest known distance from the starting vertex to 2 and the weight of the edge from 2 to 0.
4 + 4 = 8

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

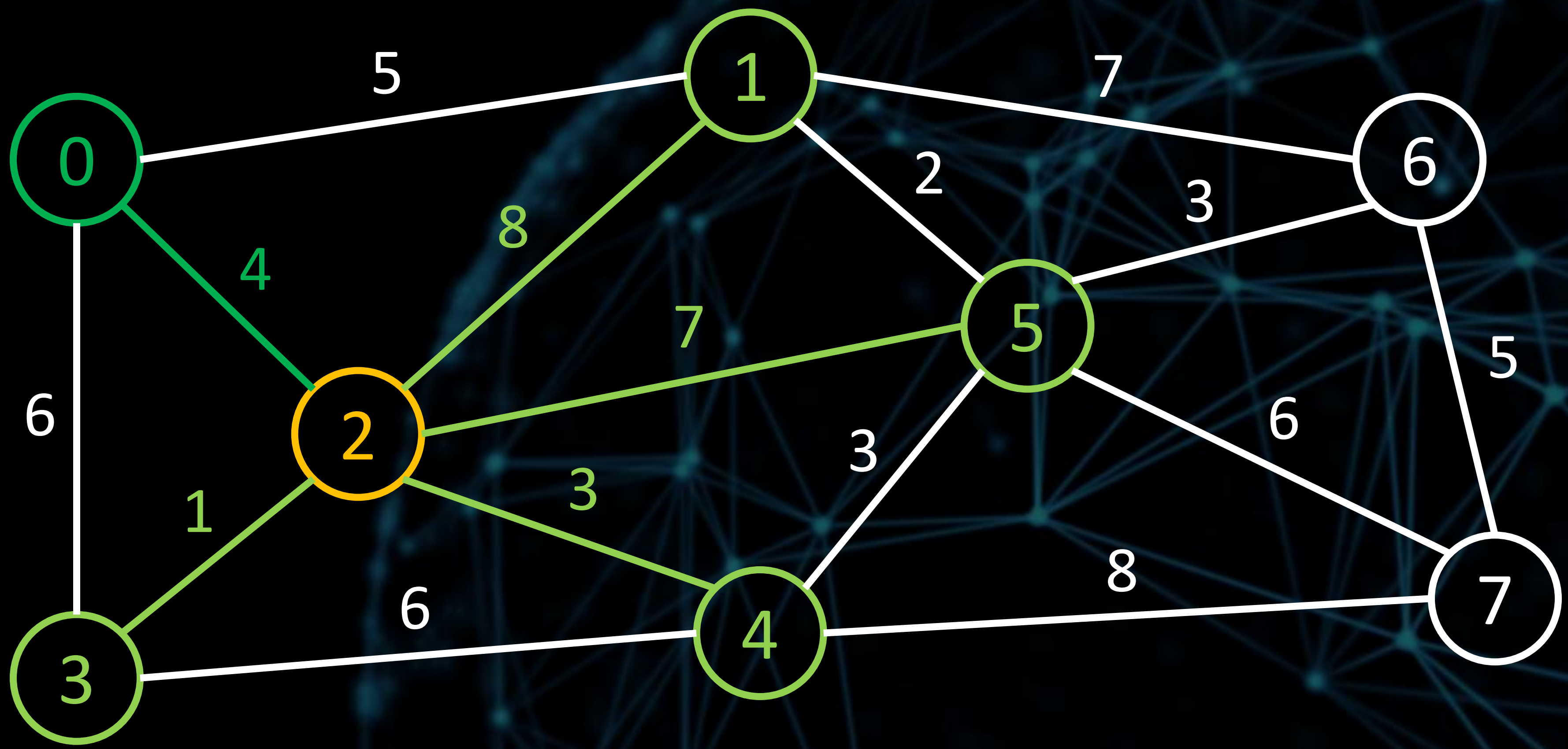
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
This distance (8) is NOT smaller than the **currently known shortest distance** to 0 (0), so we don't make any updates.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

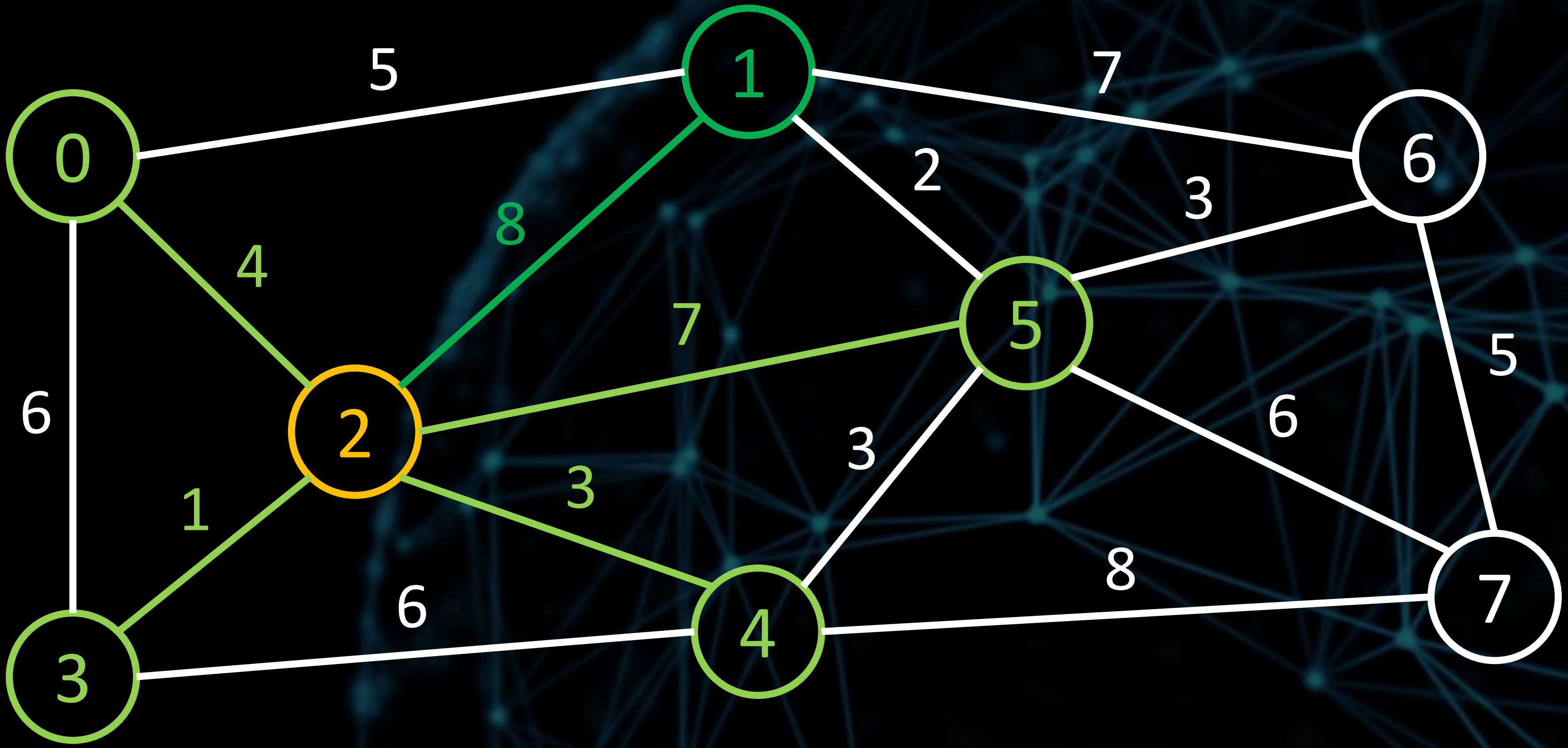
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
Now let's look at the neighbour 1.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

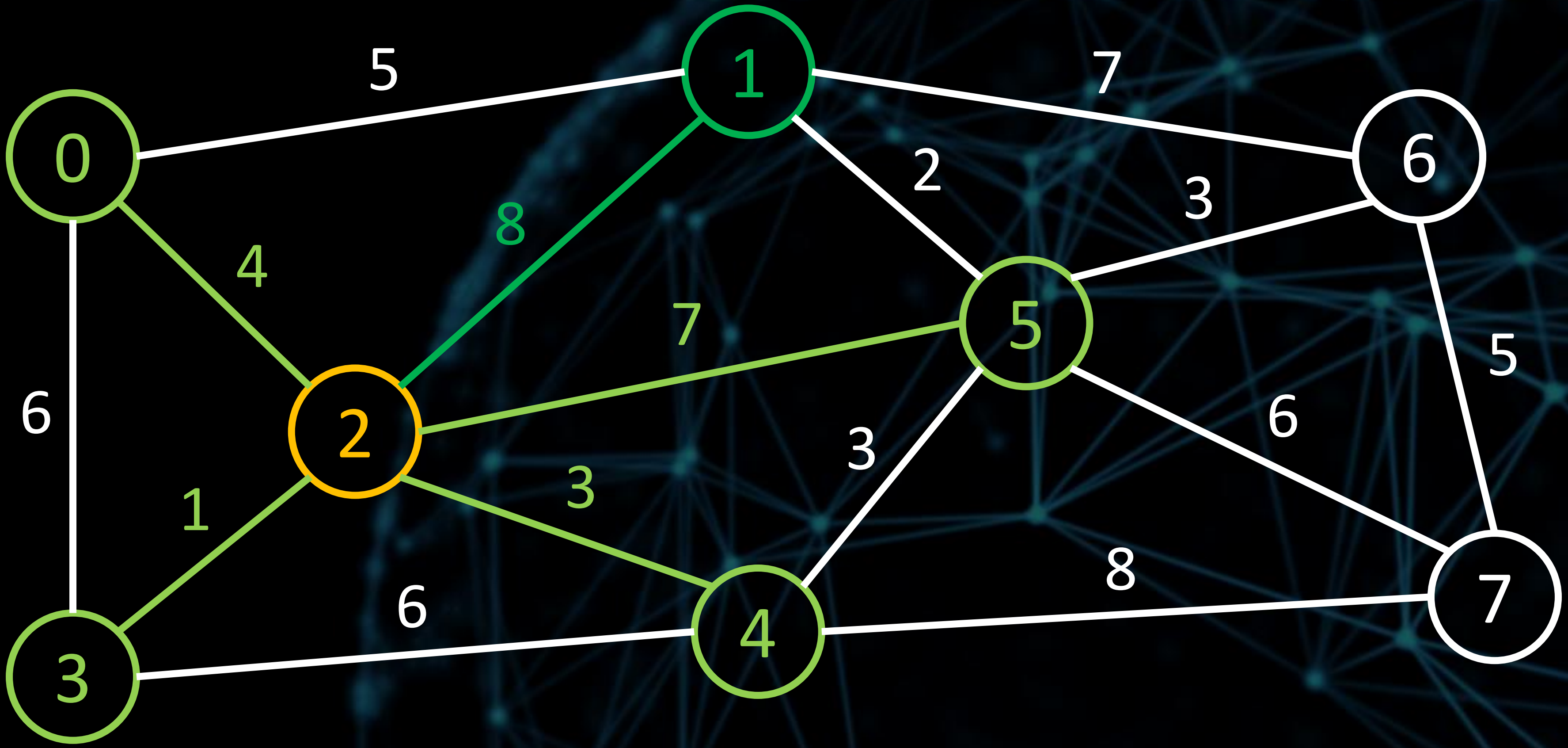
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
The distance from the starting vertex to 1 **via 2** is the sum of the shortest known distance from the starting vertex to 2 and the weight of the edge from 2 to 1.
4 + 8 = 12

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

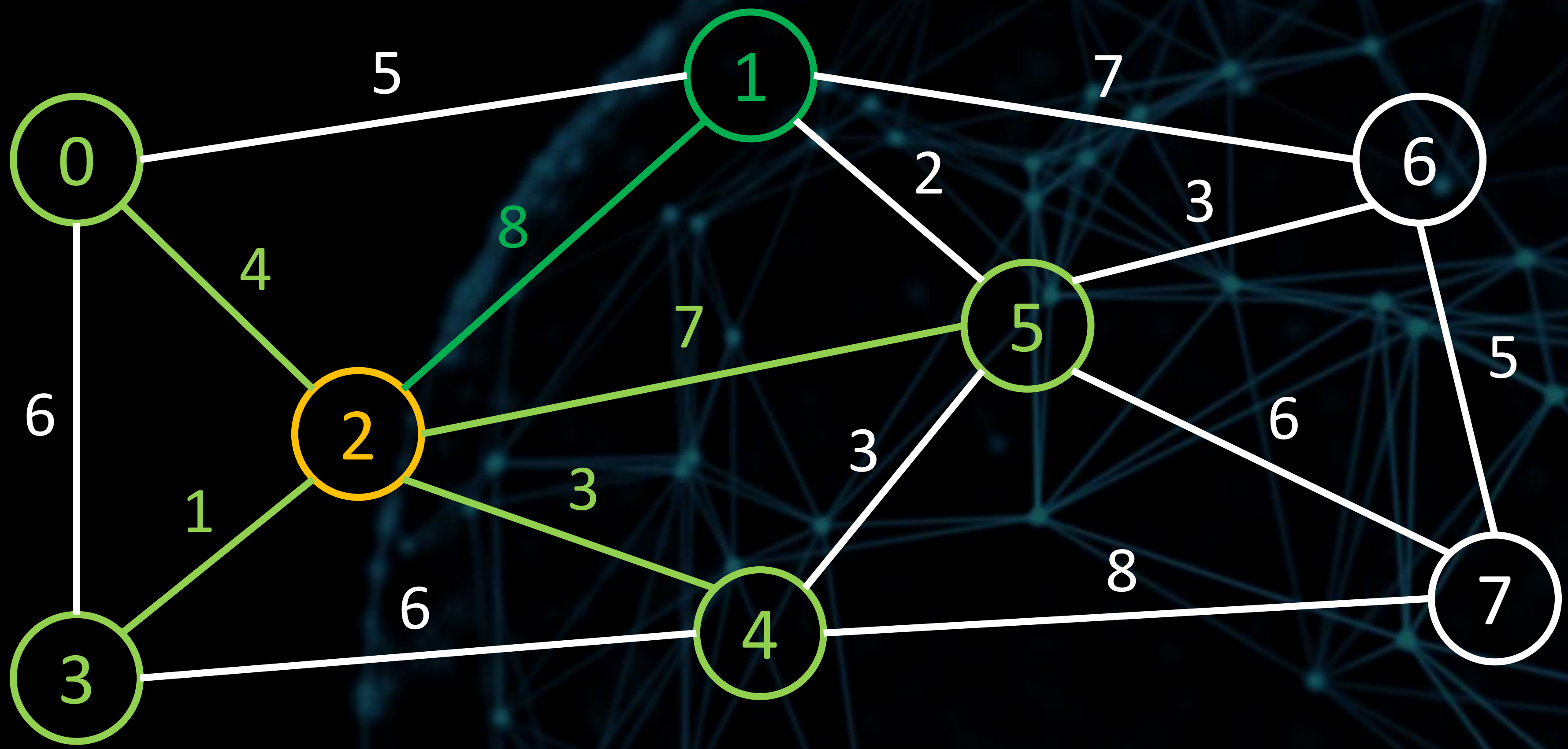
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
This distance (12) is NOT smaller than the **currently known shortest distance** to 1 (5), so we don't make any updates.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

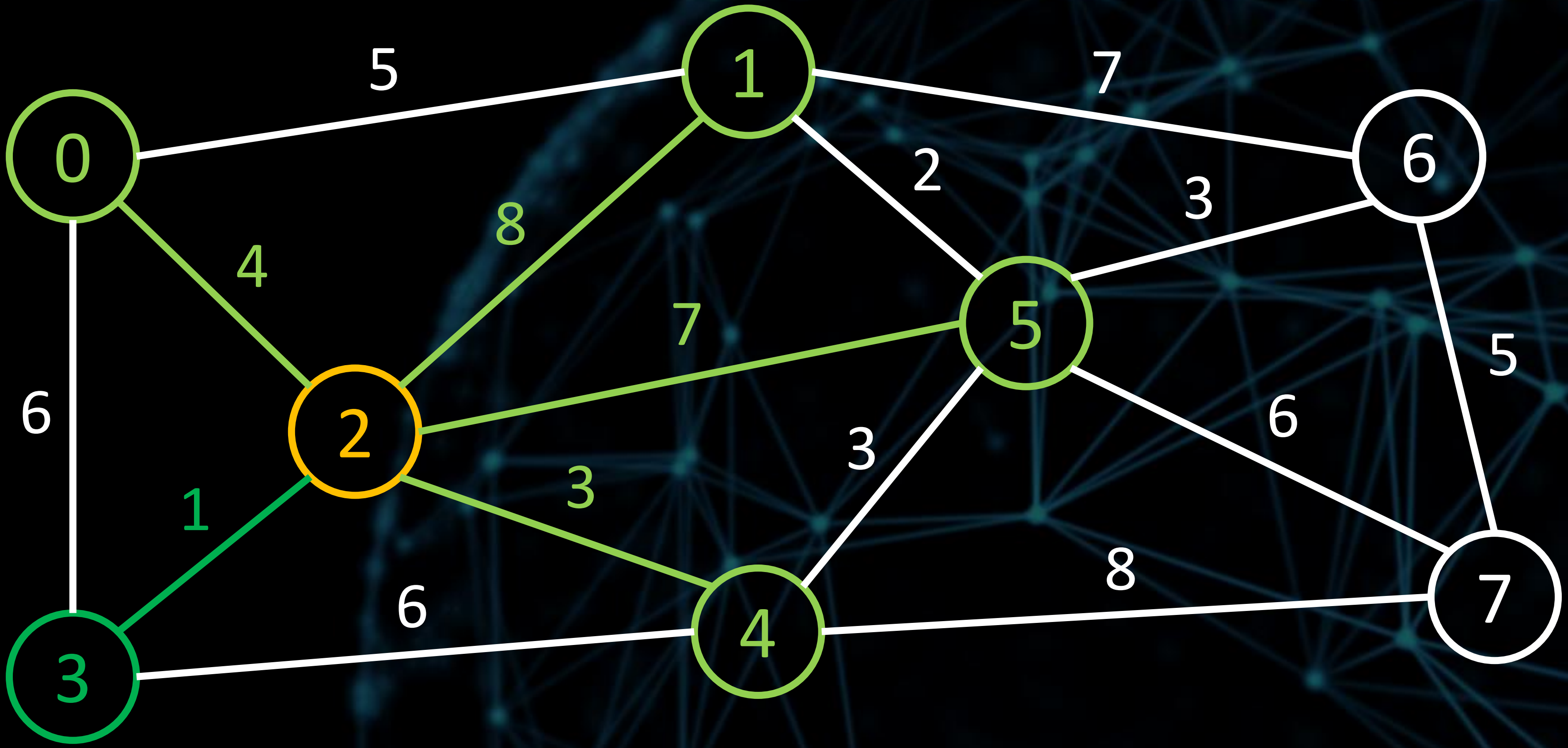
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
Now let's look at the neighbour 3.

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

vSet = {1, 3, 4, 5, 6, 7}

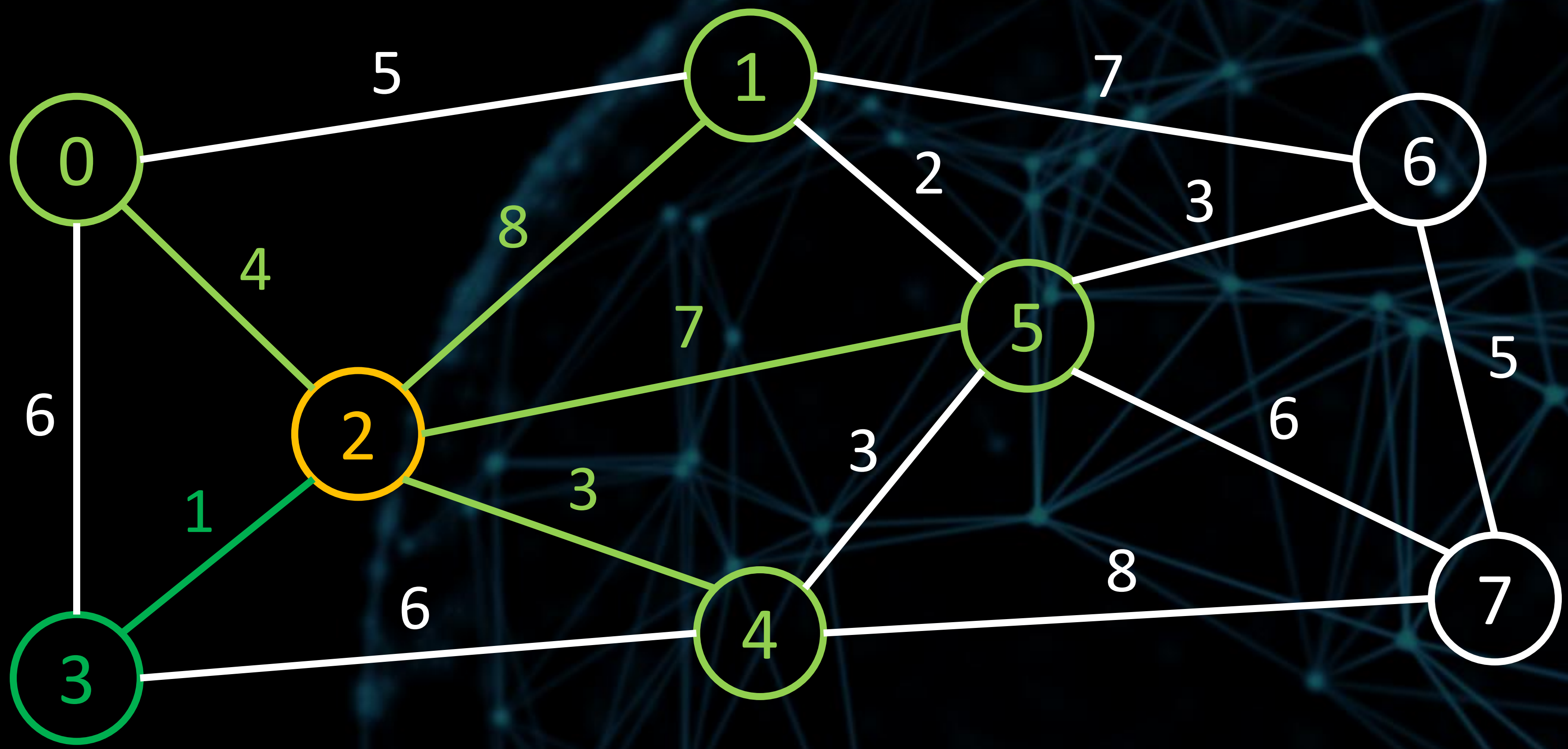


Step 2:
The distance from the starting vertex to 3 **via 2** is the sum of the shortest known distance from the starting vertex to 2 and the weight of the edge from 2 to 3.

4 + 1 = 5

	0	1	2	3	4	5	6	7
dist	0	5	4	6	∞	∞	∞	∞
pred	-1	0	0	0	-1	-1	-1	-1

vSet = {1, 3, 4, 5, 6, 7}



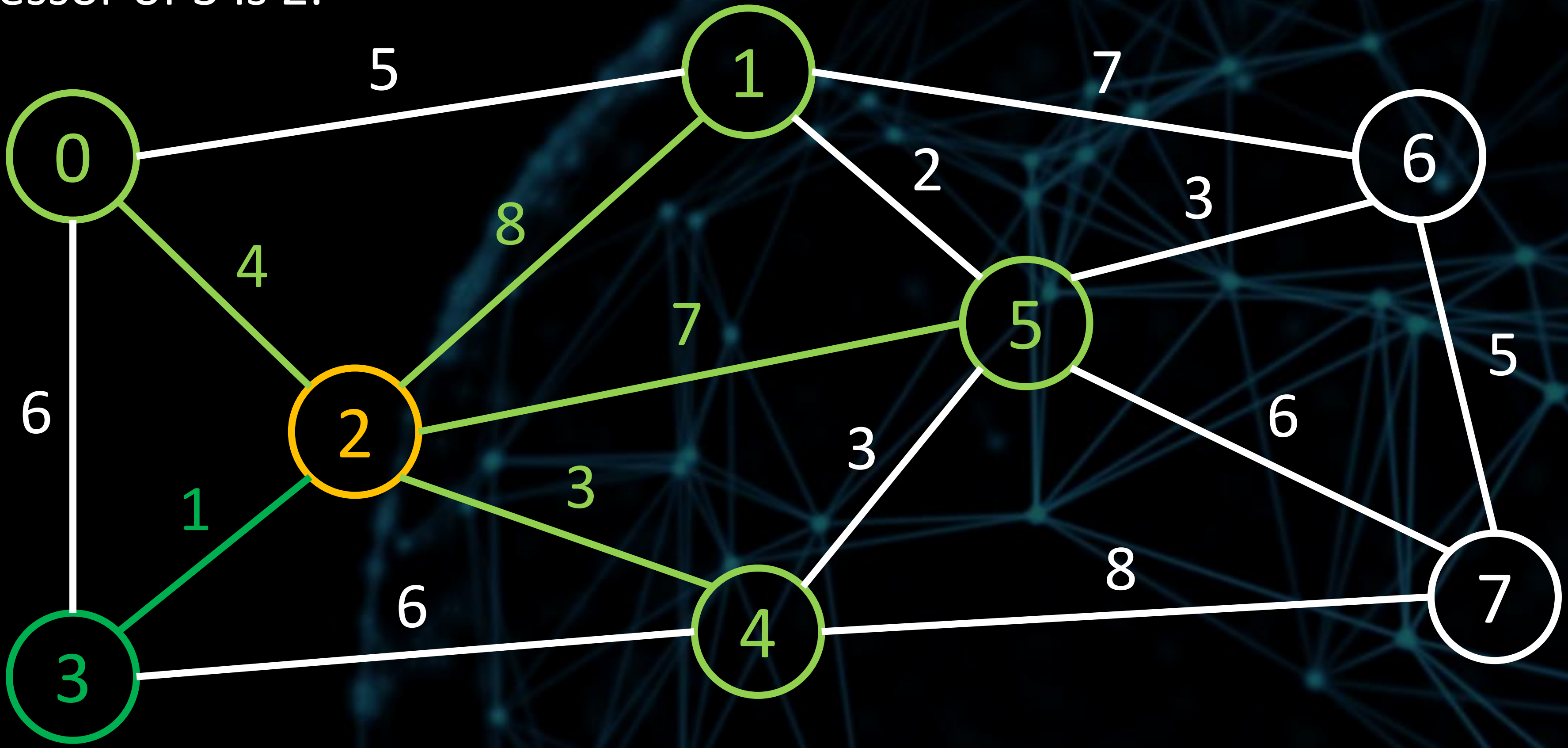
Step 2:

This distance (5) is smaller than the **currently known shortest distance** to 3 (6), so we update the dist and predecessor arrays.

The new distance of 3 is 5 and the new predecessor of 3 is 2.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	∞	∞	∞	∞
pred	-1	0	0	2	-1	-1	-1	-1

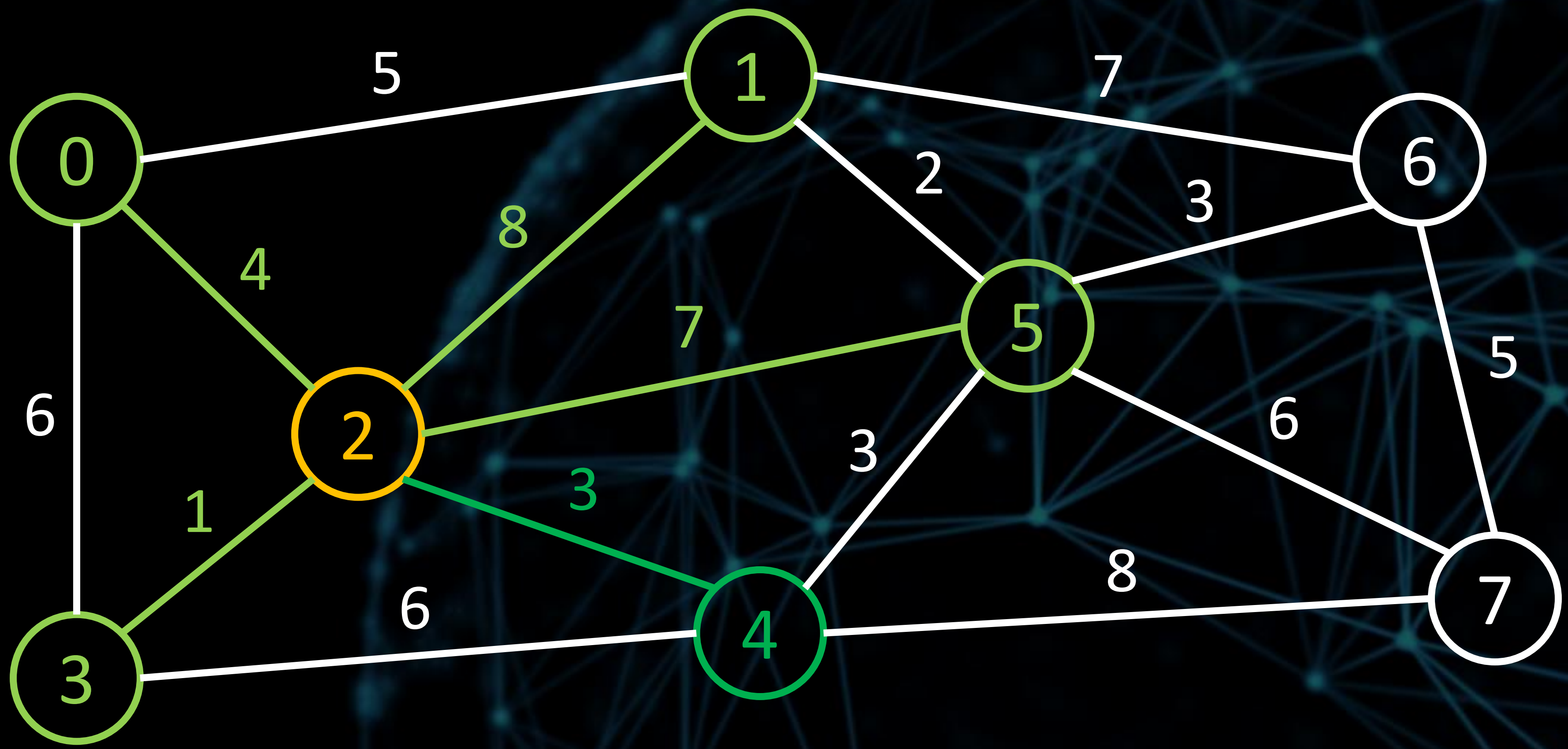
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
Now let's look at the neighbour 4.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	∞	∞	∞	∞
pred	-1	0	0	2	-1	-1	-1	-1

vSet = {1, 3, 4, 5, 6, 7}



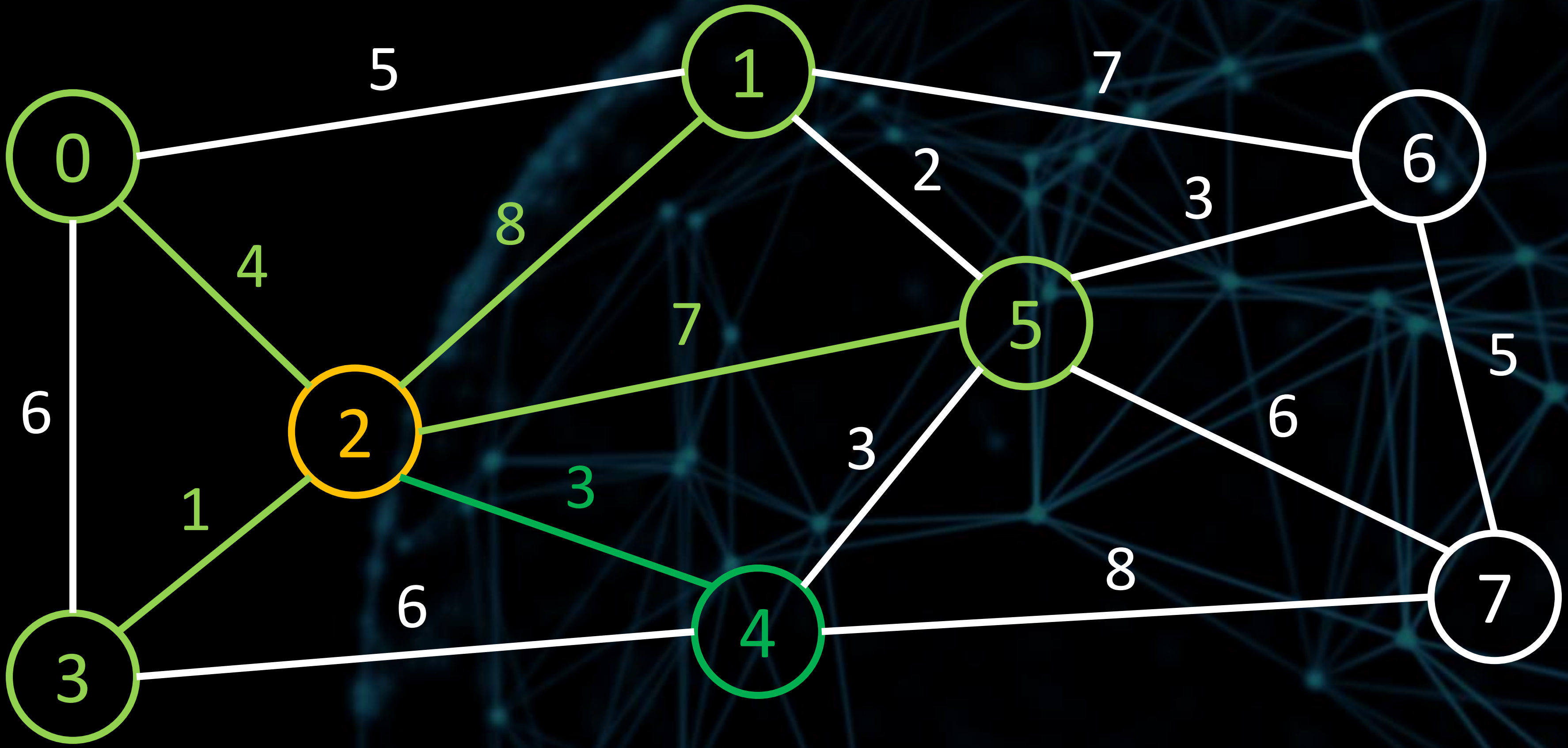
Step 2:

The distance from the starting vertex to 4 **via 2** is the sum of the shortest known distance from the starting vertex to 2 and the weight of the edge from 2 to 4.

4 + 3 = 7

	0	1	2	3	4	5	6	7
dist	0	5	4	5	∞	∞	∞	∞
pred	-1	0	0	2	-1	-1	-1	-1

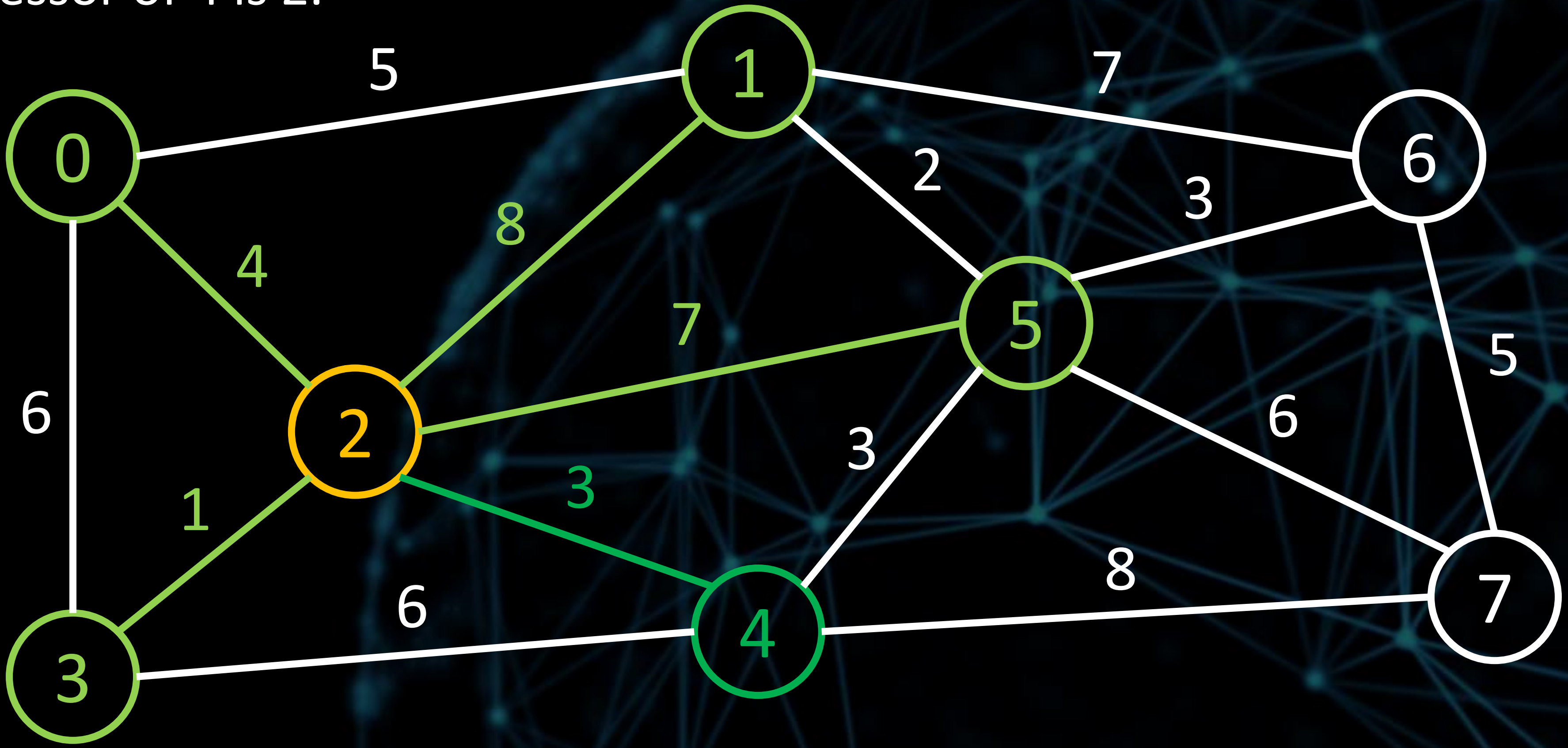
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
This distance (7) is smaller than the **currently known shortest distance** to 4 (∞), so we update the dist and predecessor arrays.
The new distance of 4 is 7 and the new predecessor of 4 is 2.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	∞	∞	∞
pred	-1	0	0	2	2	-1	-1	-1

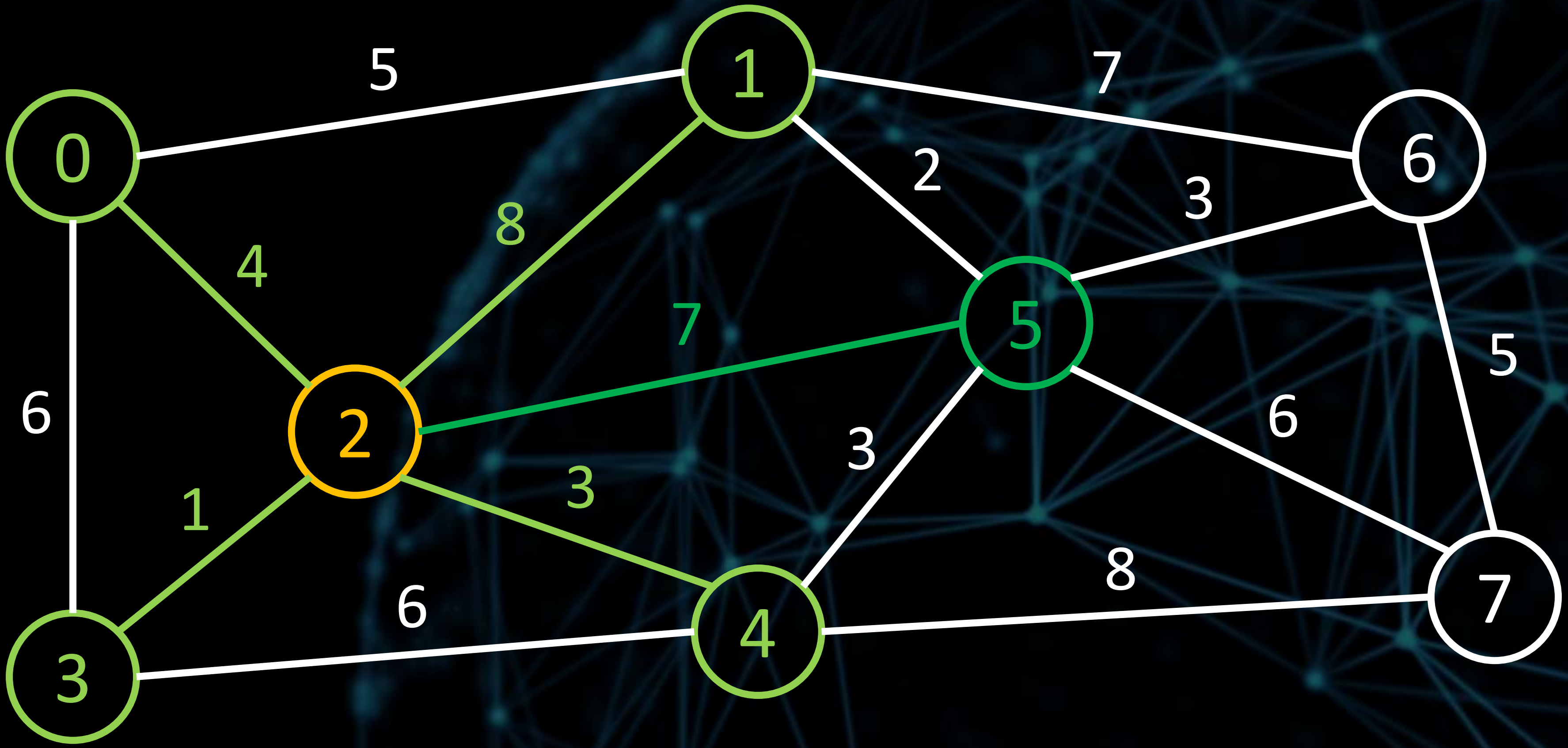
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
Now let's look at the neighbour 5.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	∞	∞	∞
pred	-1	0	0	2	2	-1	-1	-1

vSet = {1, 3, 4, 5, 6, 7}



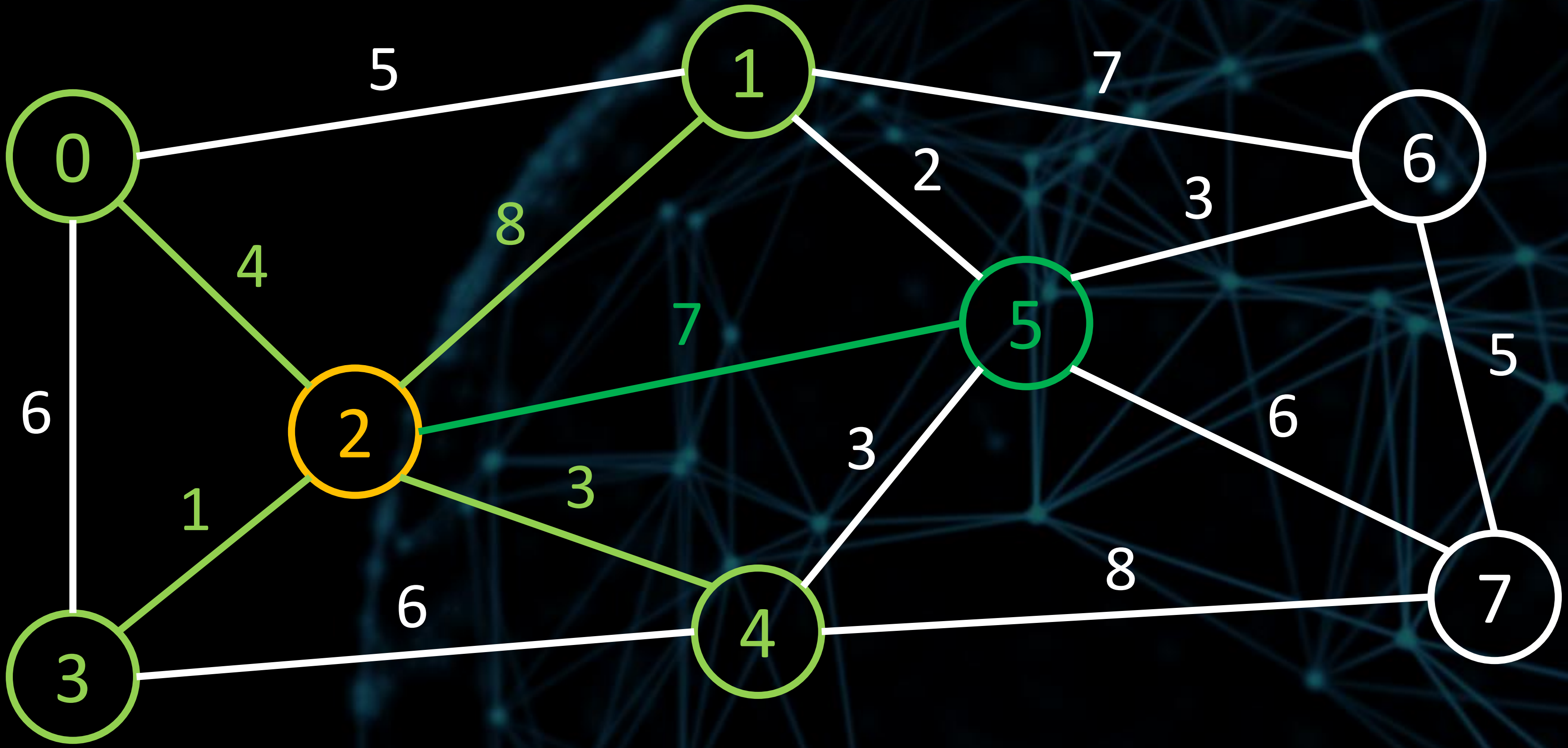
Step 2:

The distance from the starting vertex to 5 **via 2** is the sum of the shortest known distance from the starting vertex to 2 and the weight of the edge from 2 to 5.

4 + 7 = 11

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	5	7	∞	∞	∞
pred	-1	0	0	2	2	-1	-1	-1

vSet = {1, 3, 4, 5, 6, 7}



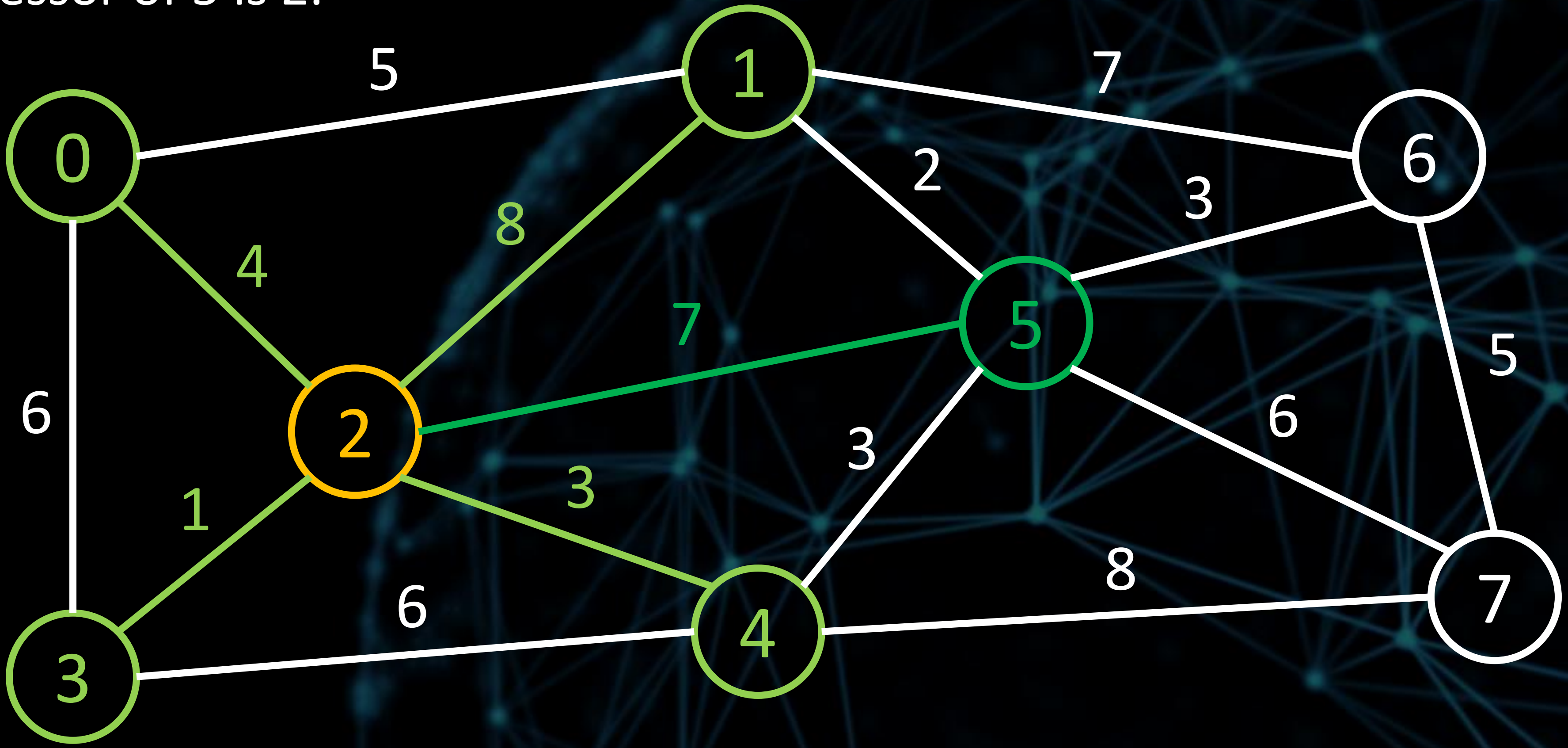
Step 2:

This distance (11) is smaller than the **currently known shortest distance** to 5 (∞), so we update the dist and predecessor arrays.

The new distance of 5 is 11 and the new predecessor of 5 is 2.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

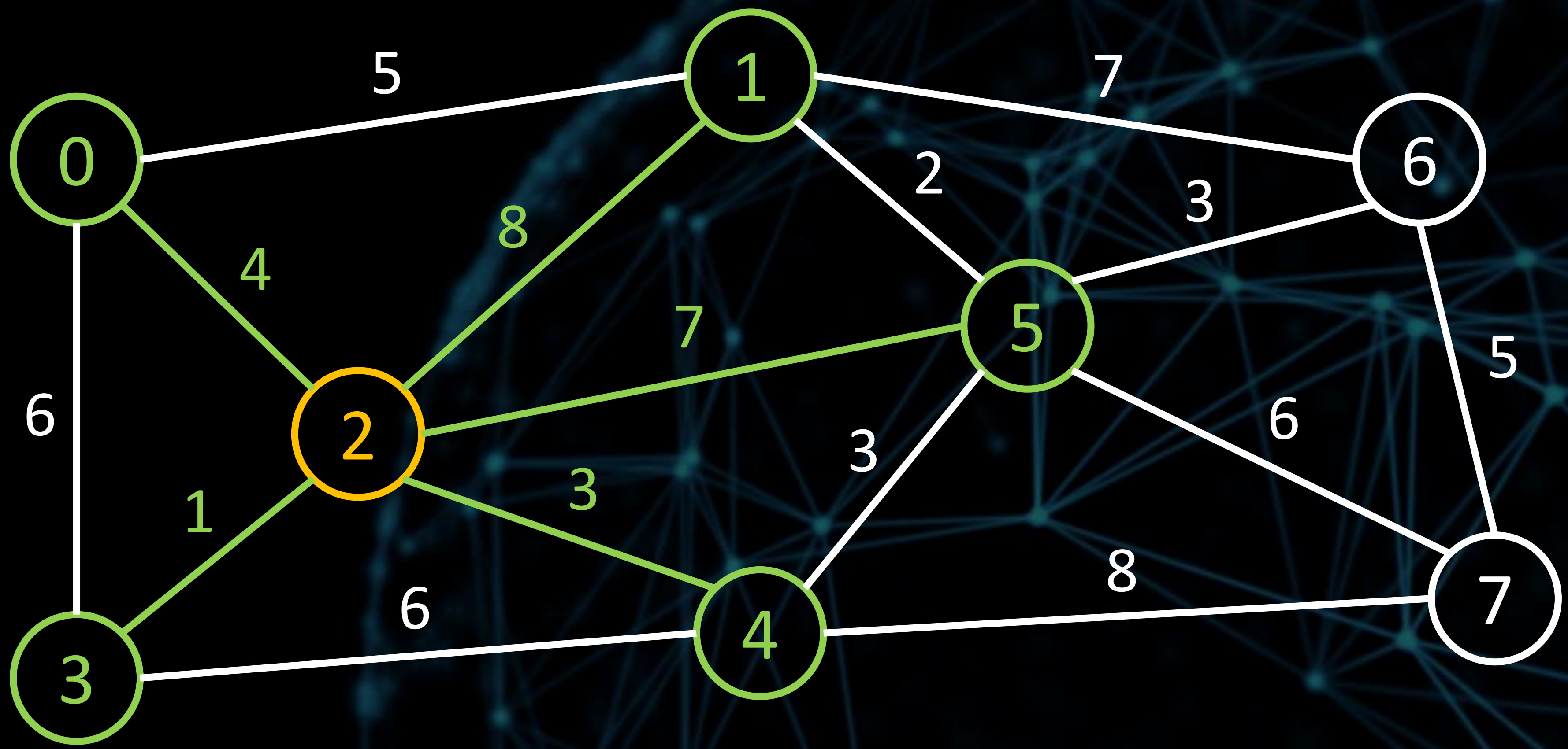
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
We've checked all the neighbours of 2, so we've completed another iteration of the algorithm.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

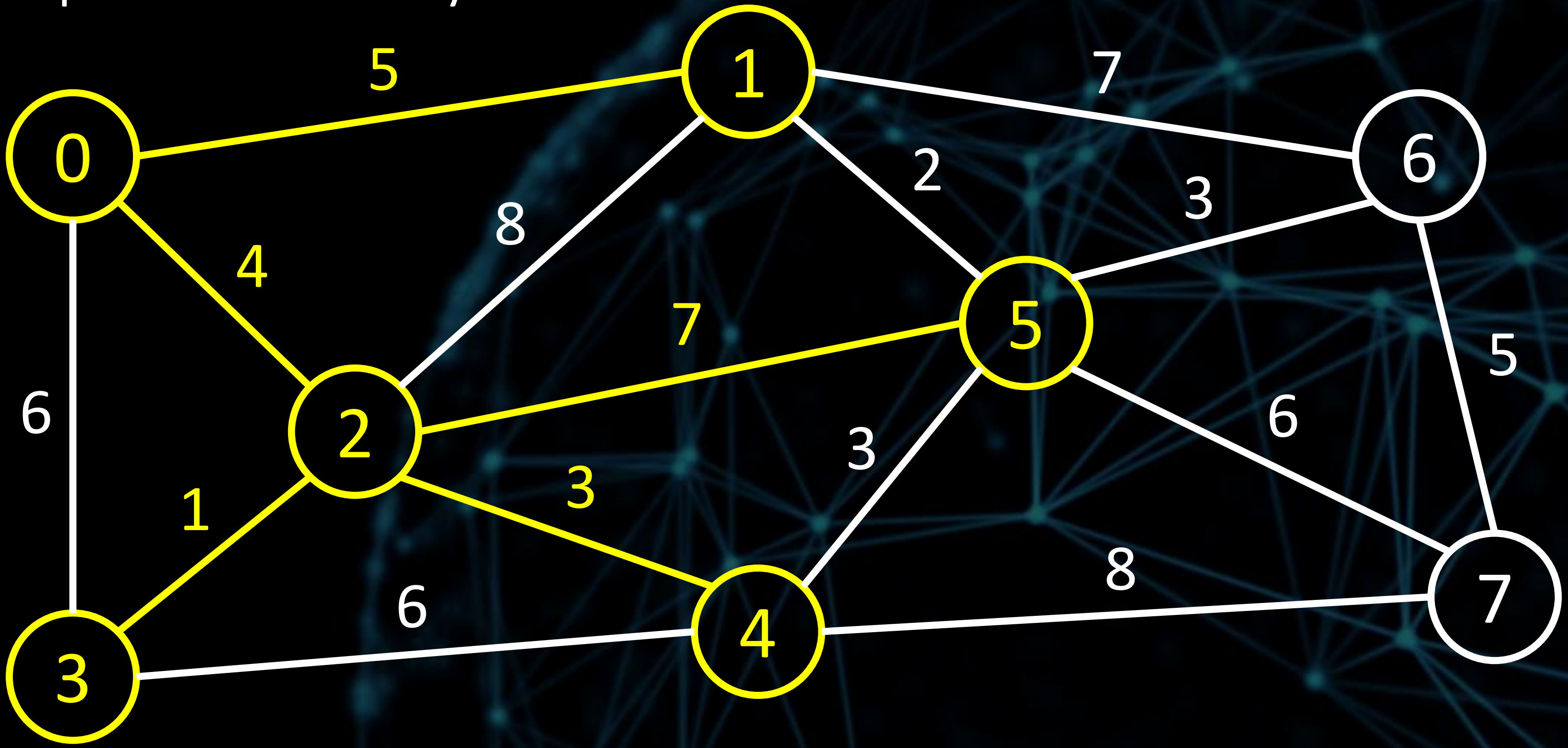
vSet = {1, 3, 4, 5, 6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

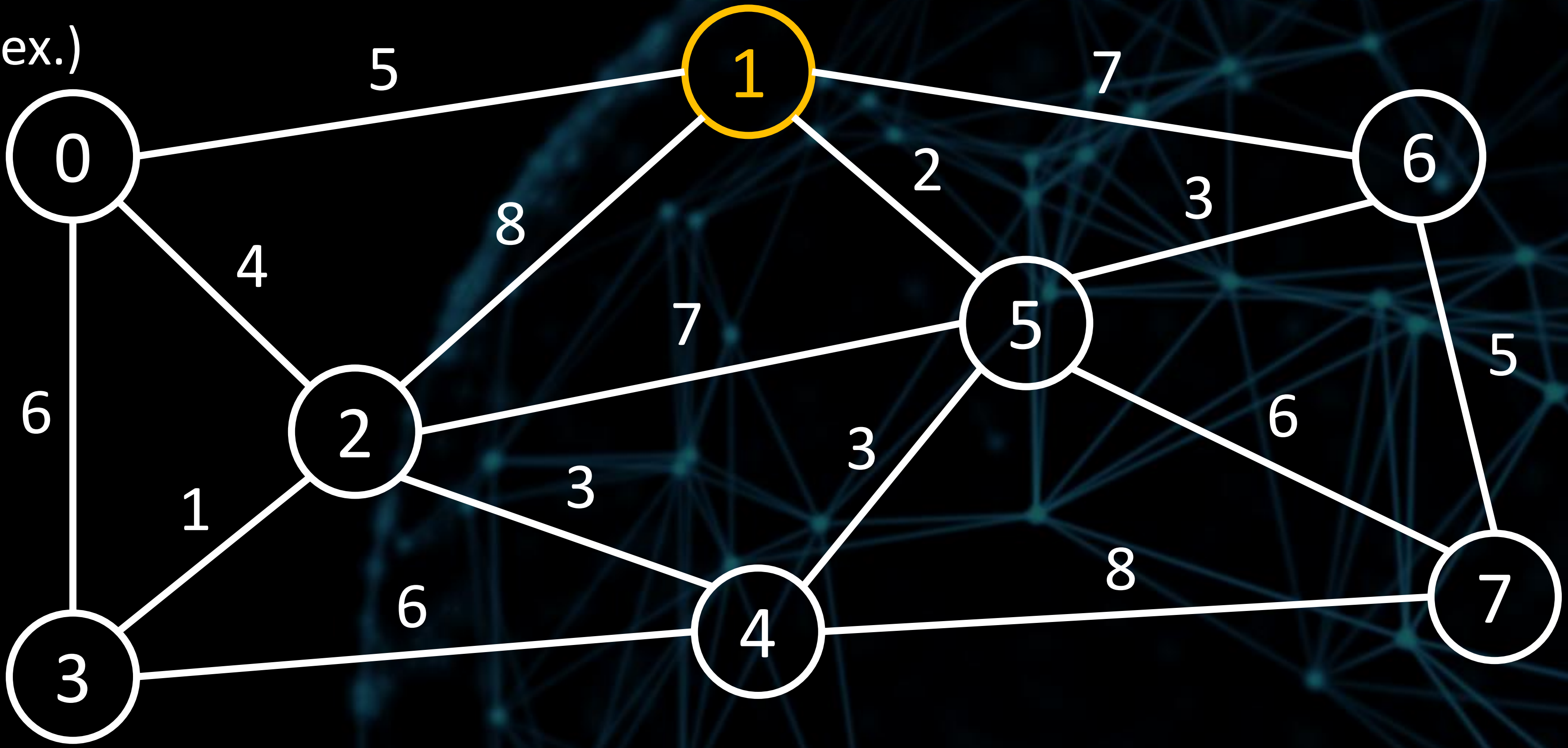
vSet = {1, 3, 4, 5, 6, 7}



Step 1:
Choose the vertex from vSet that is closest to the starting vertex, and remove it from vSet.
That vertex is 1. (We could also choose 3 but we choose the smaller vertex.)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

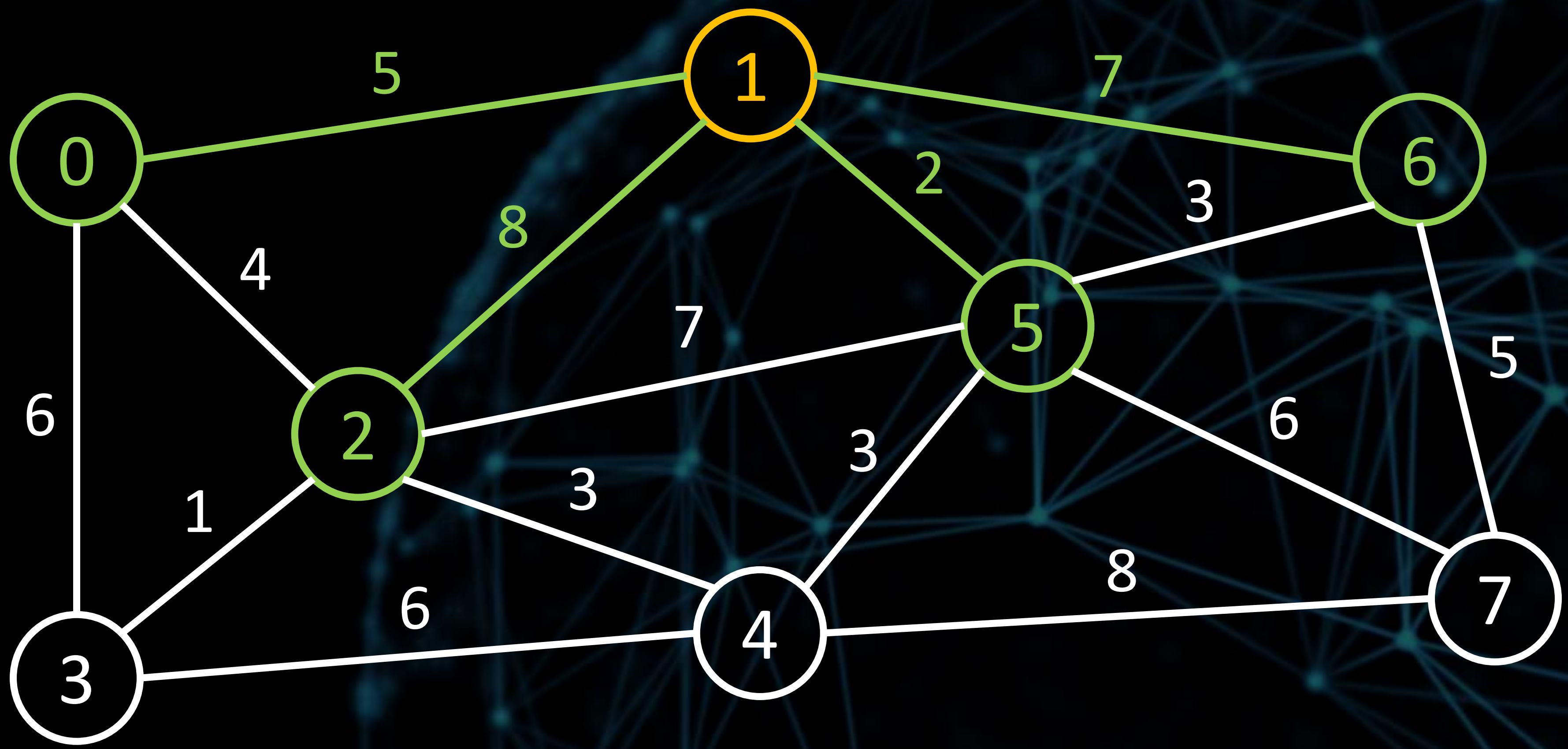
vSet = {1, 3, 4, 5, 6, 7}



Step 2:
For each neighbour of vertex 1, we check if there is a shorter path to the neighbour **via 1**.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

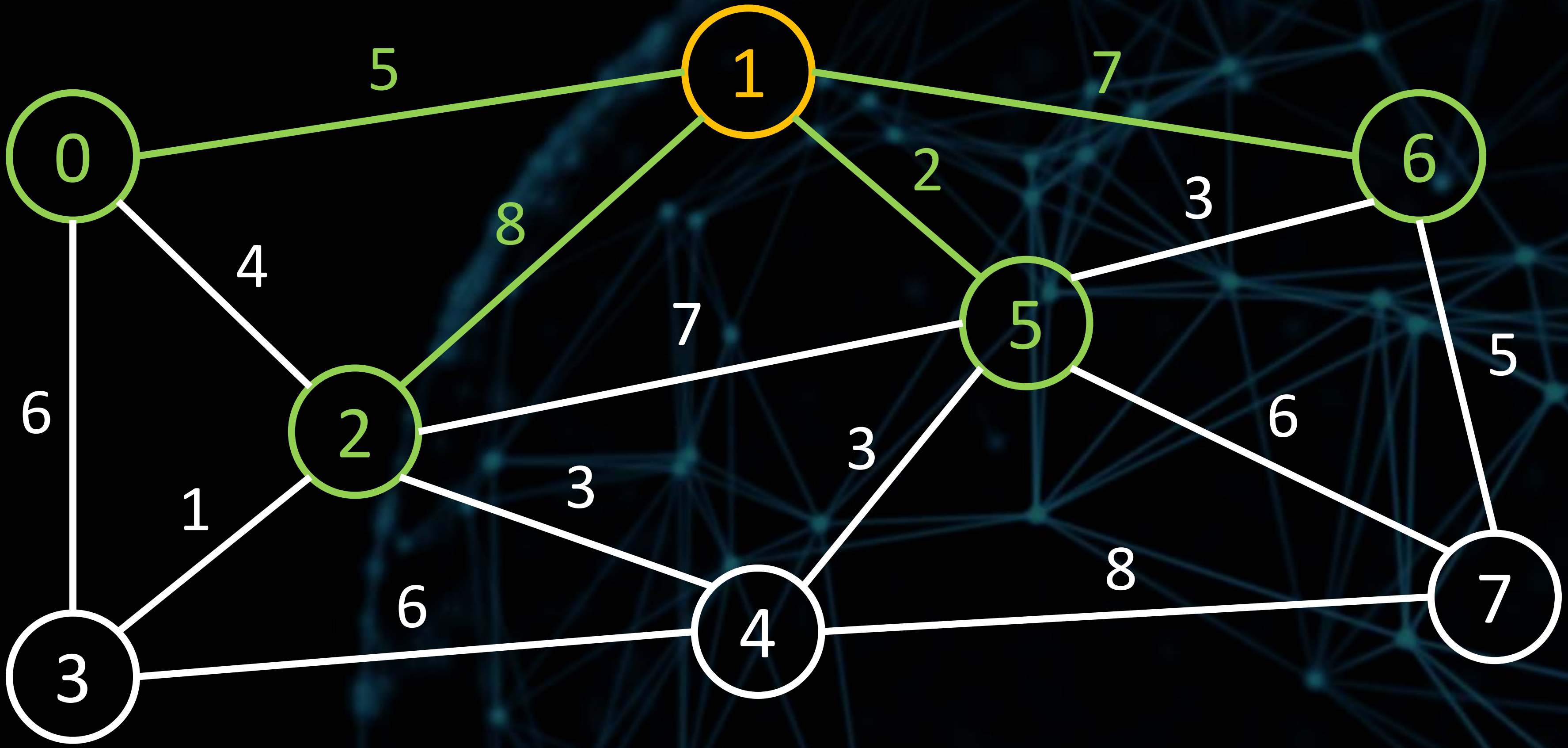
vSet = {3, 4, 5, 6, 7}



Step 2:
It's your turn! For each neighbour of vertex 1, make any updates to the dist and pred arrays as necessary.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	11	∞	∞
pred	-1	0	0	2	2	2	-1	-1

vSet = {3, 4, 5, 6, 7}

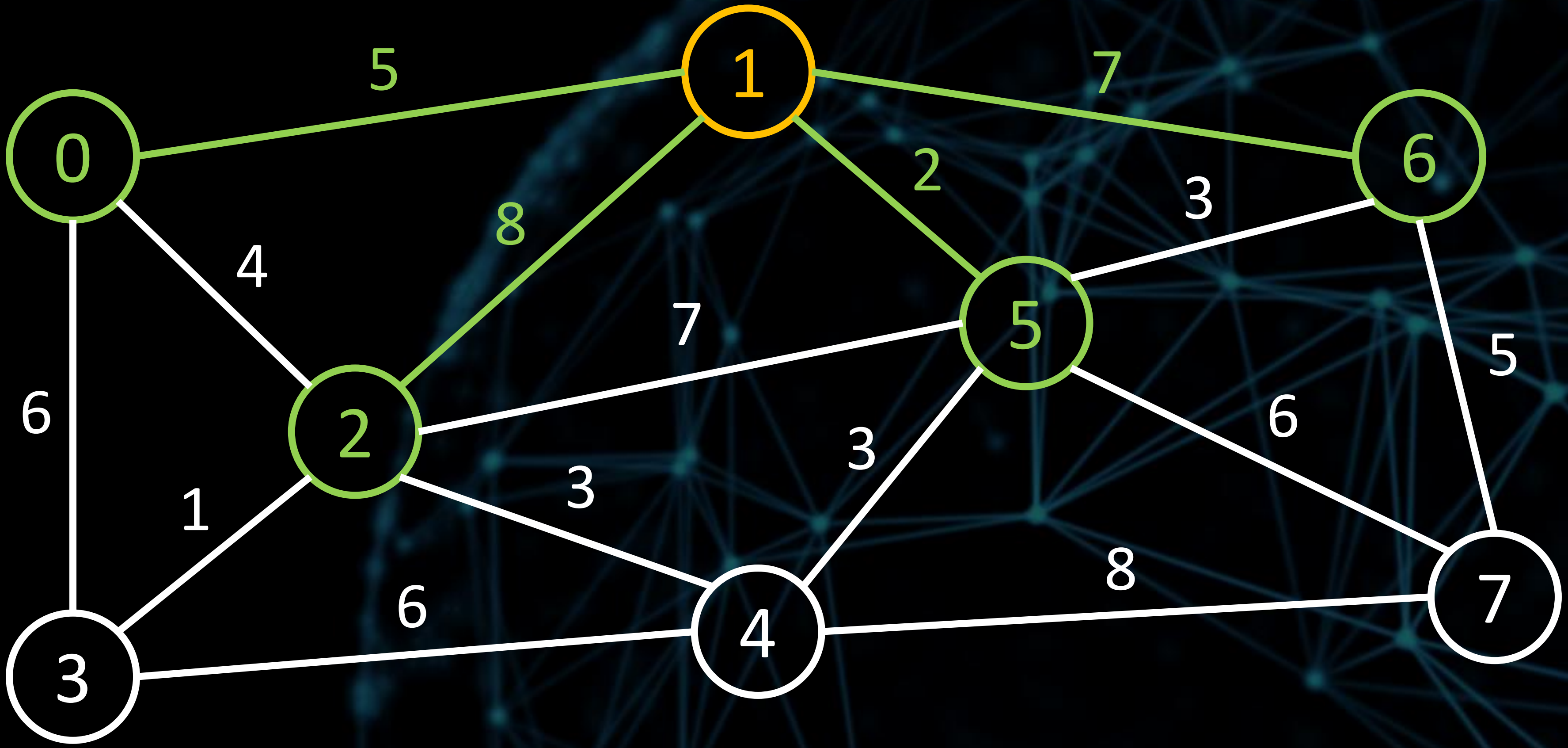


Check your answer on the next slide…

Step 2:
After making all necessary updates to the dist and pred arrays

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

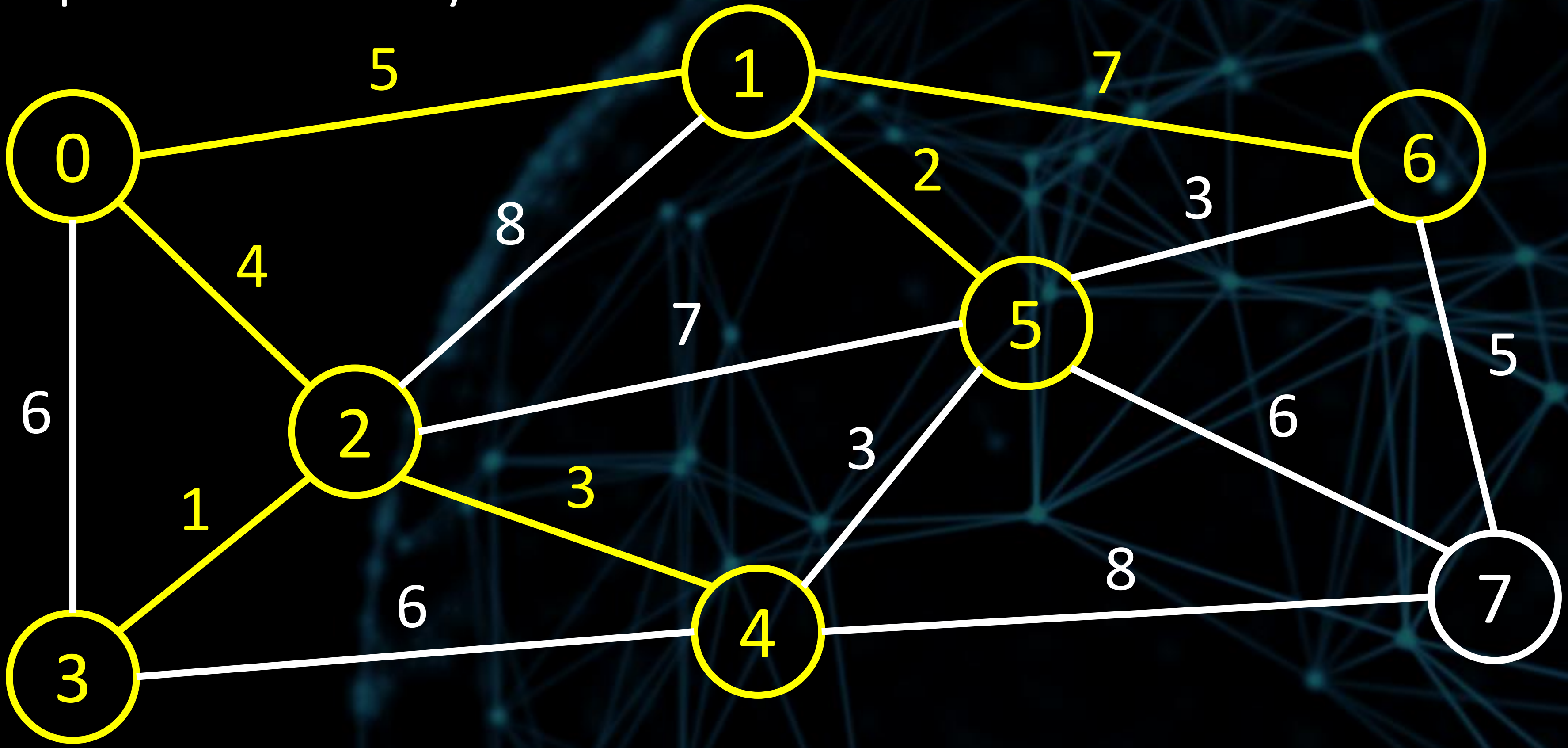
vSet = {3, 4, 5, 6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

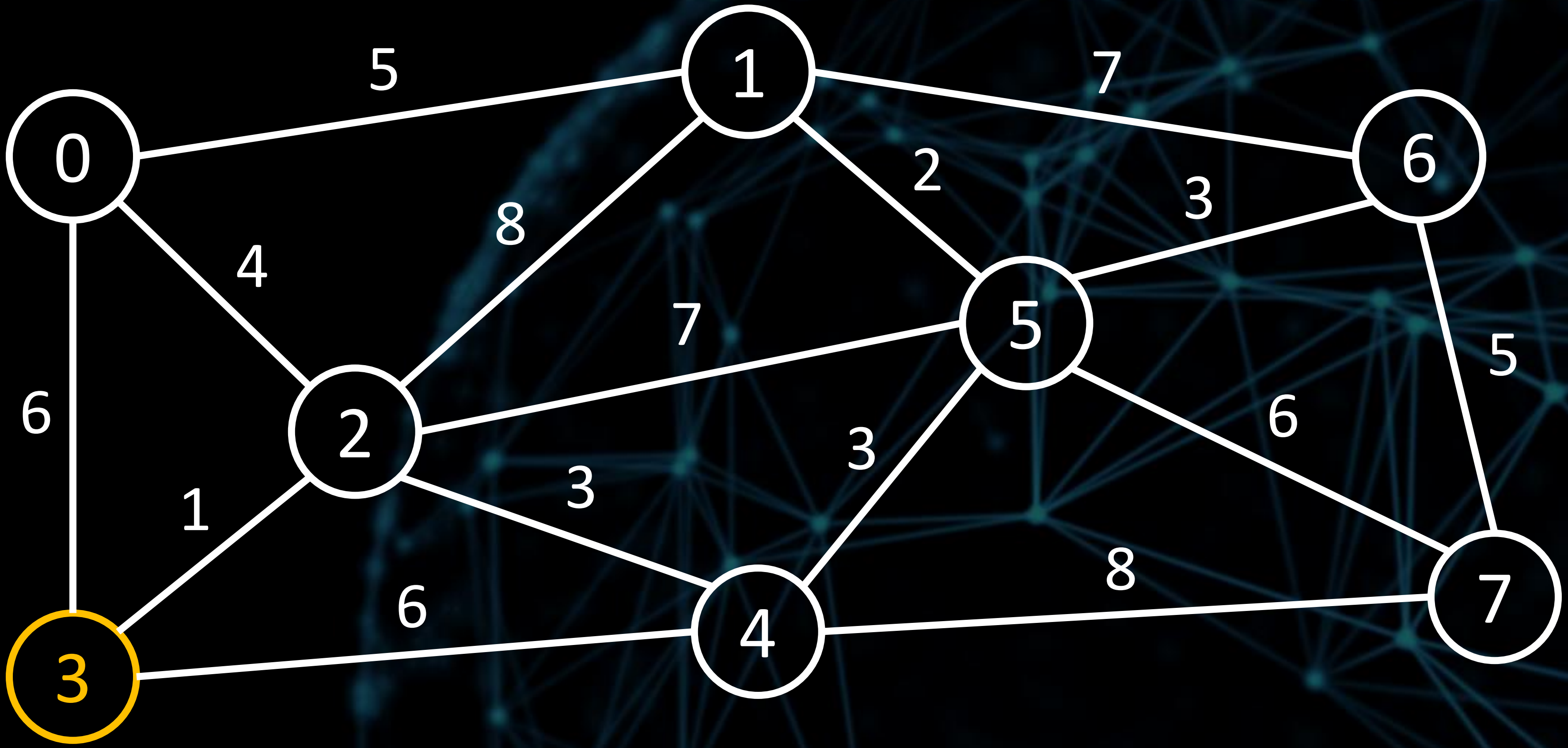
vSet = {3, 4, 5, 6, 7}



Step 1:
Choose the vertex from vSet that is closest to the starting vertex, and remove it from vSet.
That vertex is 3.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

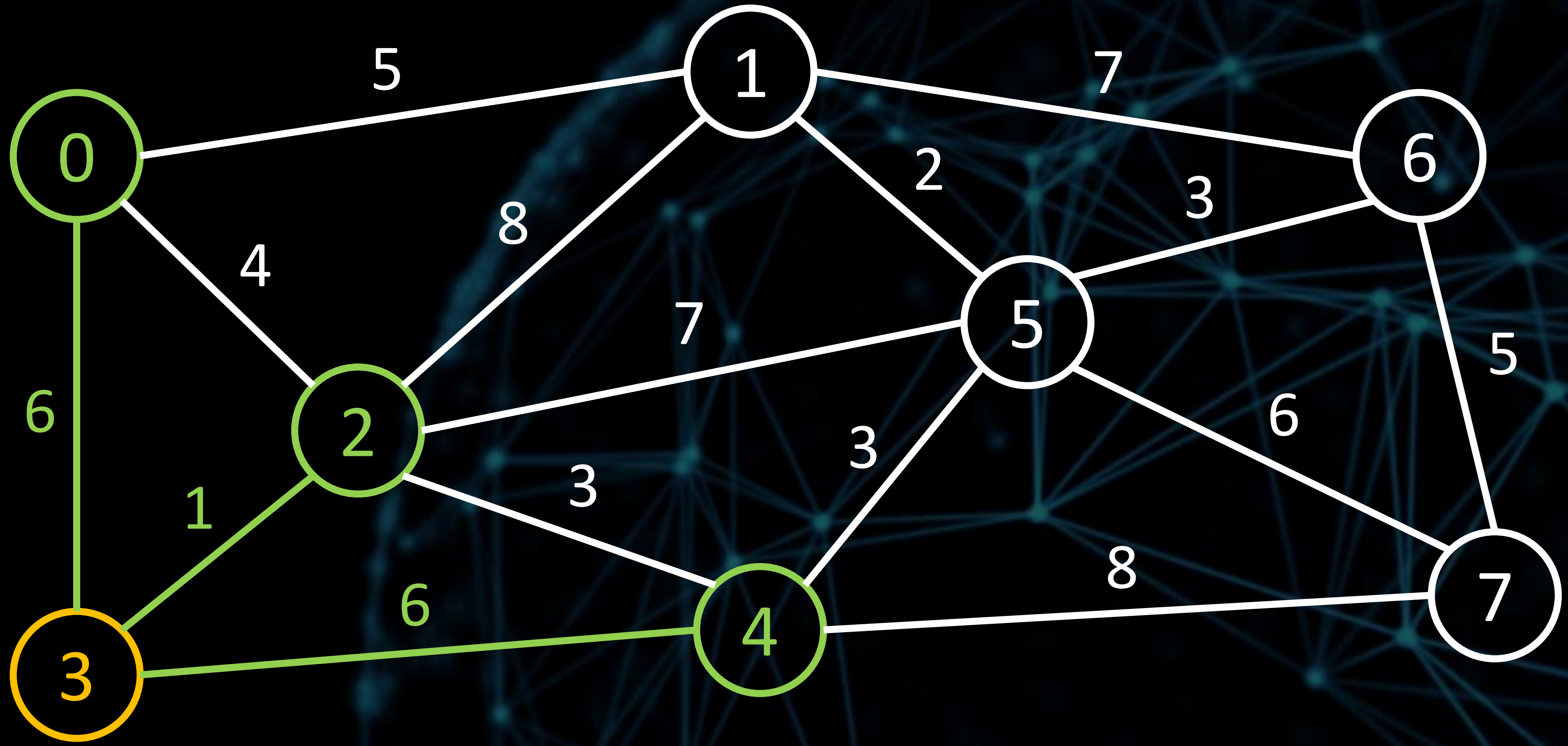
vSet = {3, 4, 5, 6, 7}



Step 2:
For each neighbour of vertex 3, we check if there is a shorter path to the neighbour **via 3**.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

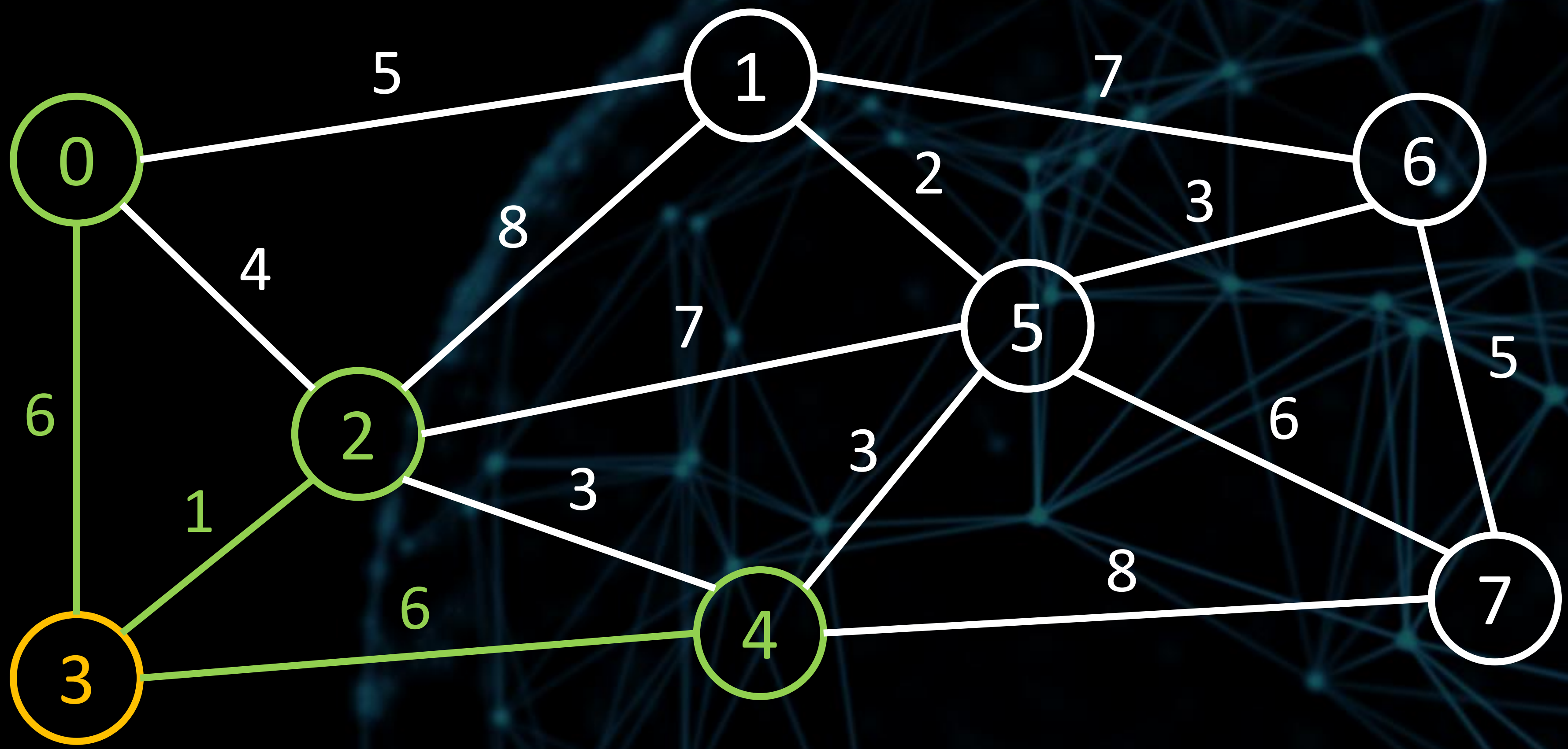
vSet = {4, 5, 6, 7}



Step 2:
It's your turn! For each neighbour of vertex 3, make any updates to the dist and pred arrays as necessary.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

vSet = {4, 5, 6, 7}

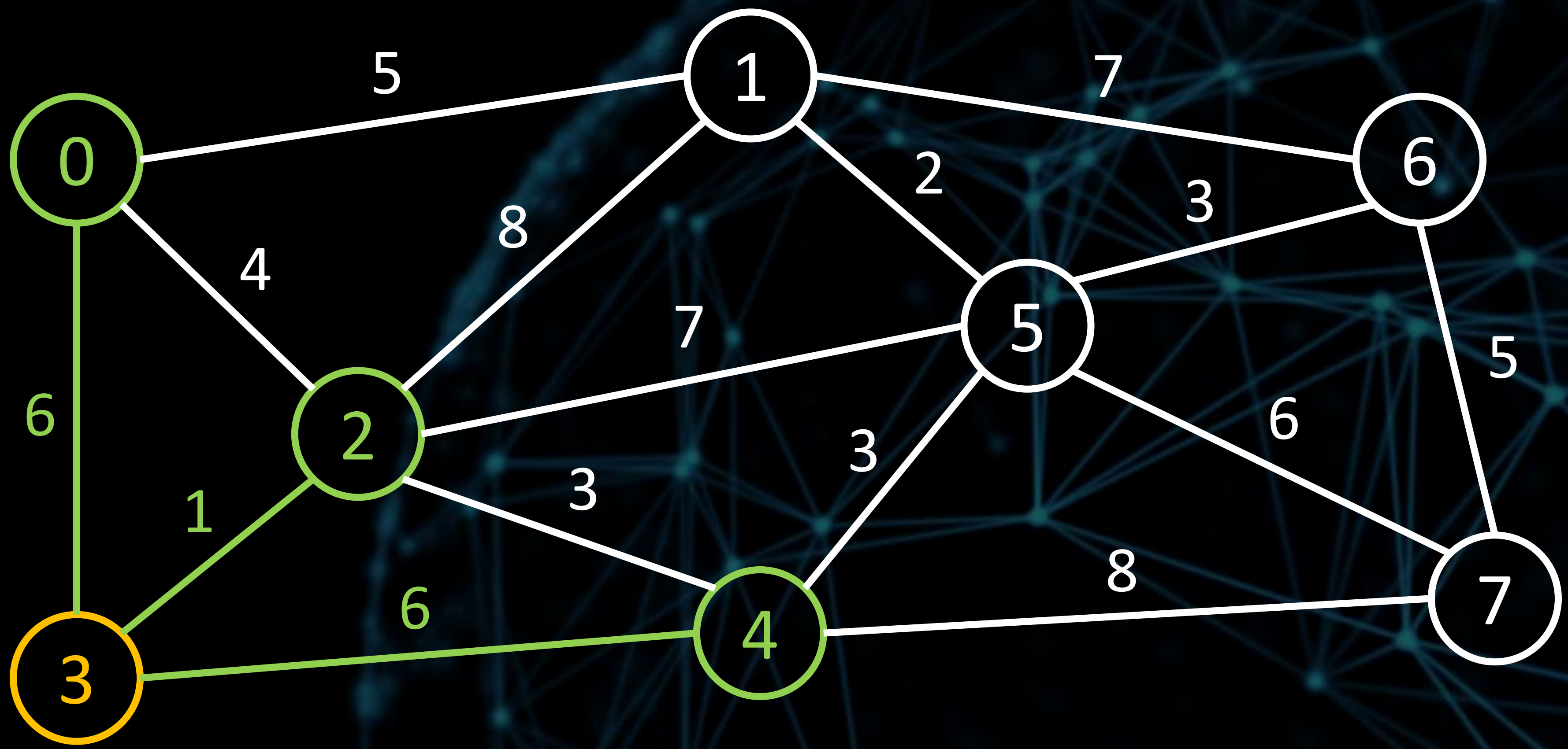


Check your answer on the next slide…

Step 2:
After making all necessary updates to the dist and pred arrays (NO updates were made!)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

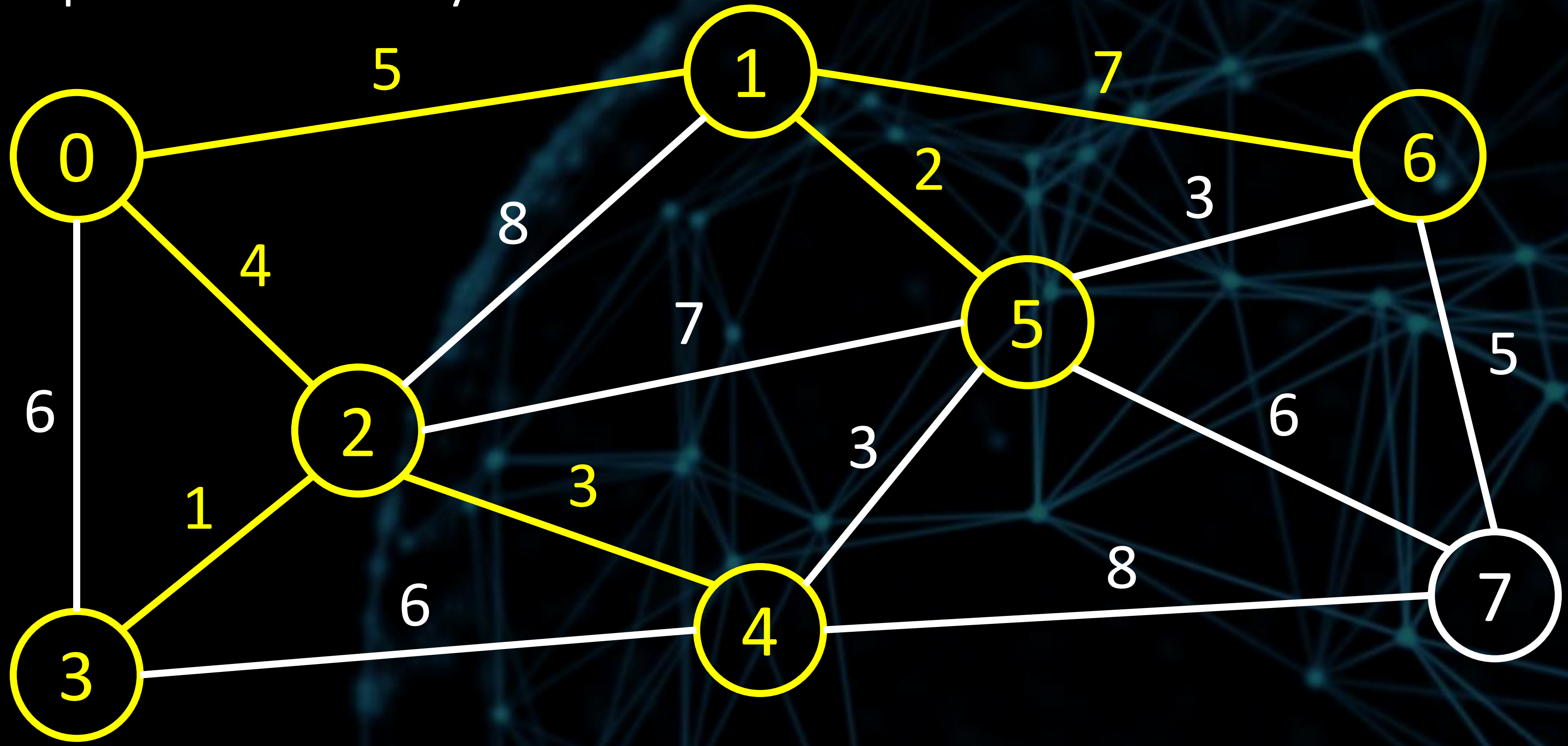
vSet = {4, 5, 6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

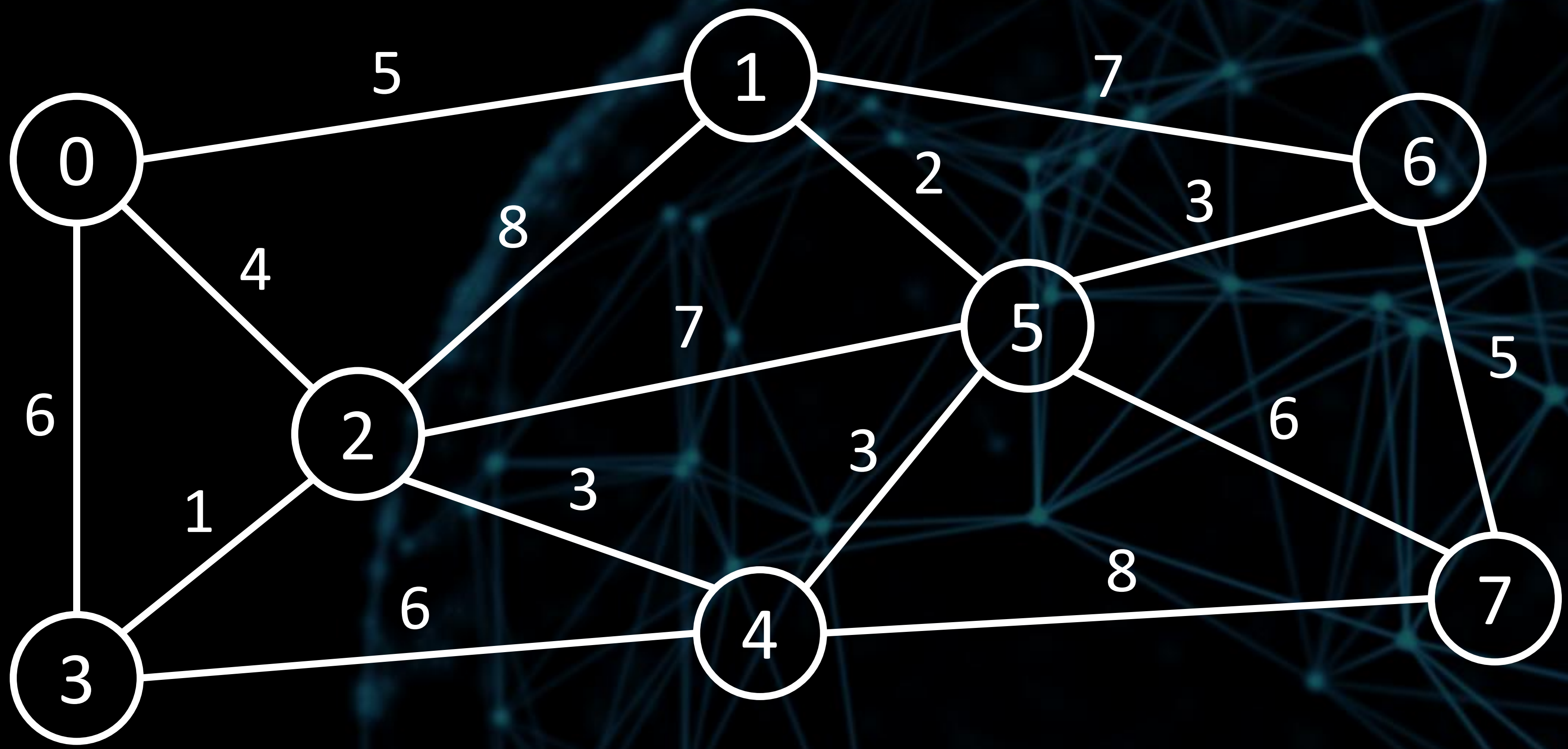
vSet = {4, 5, 6, 7}



It’s your turn to perform one more iteration of the algorithm.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	5	7	7	12	∞
pred	-1	0	0	2	2	1	1	-1

vSet = {4, 5, 6, 7}

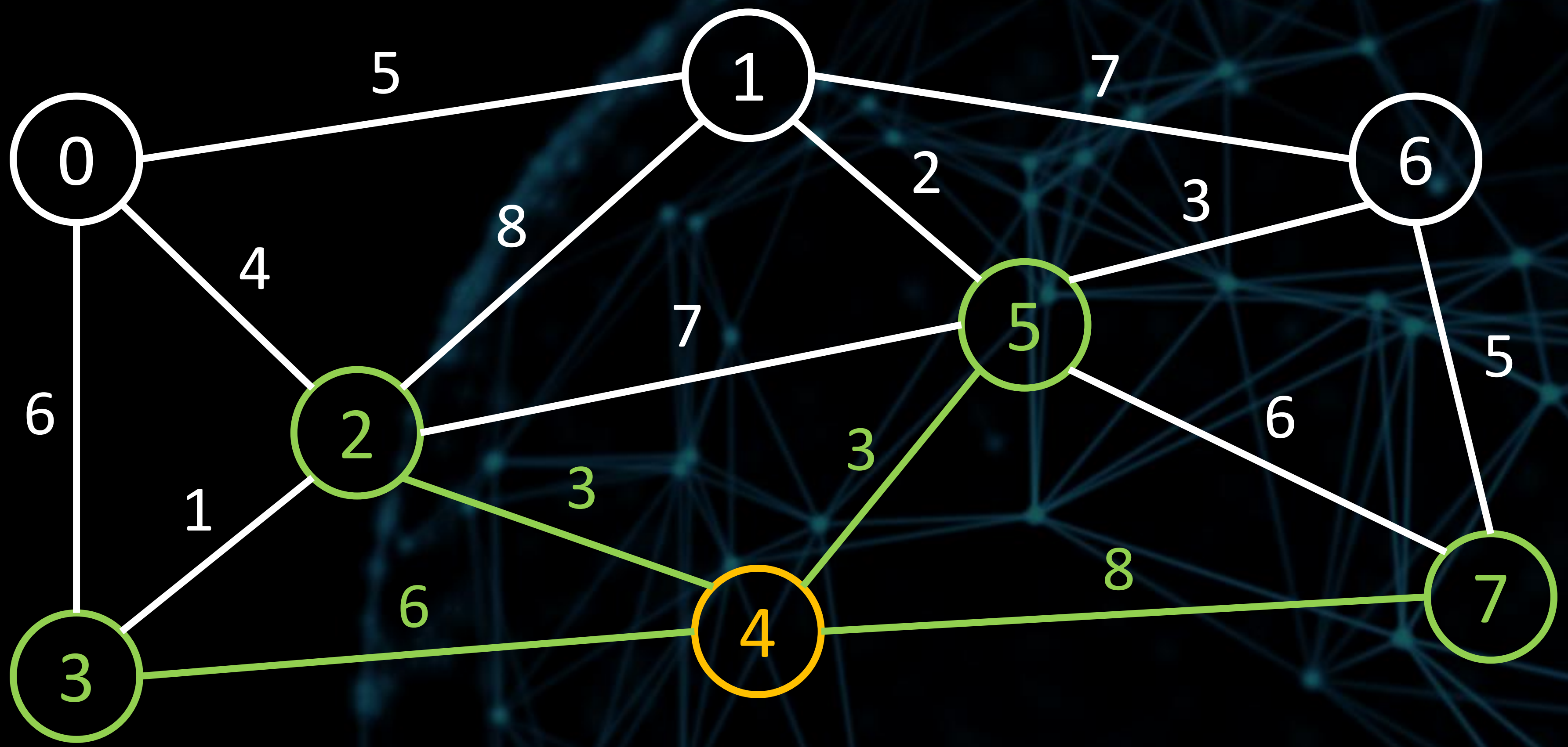


Check your answer on the next slide…

Step 1: We chose vertex 4.
Step 2: We made all necessary updates to the dist and pred arrays (only one update was made)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	15
pred	-1	0	0	2	2	1	1	4

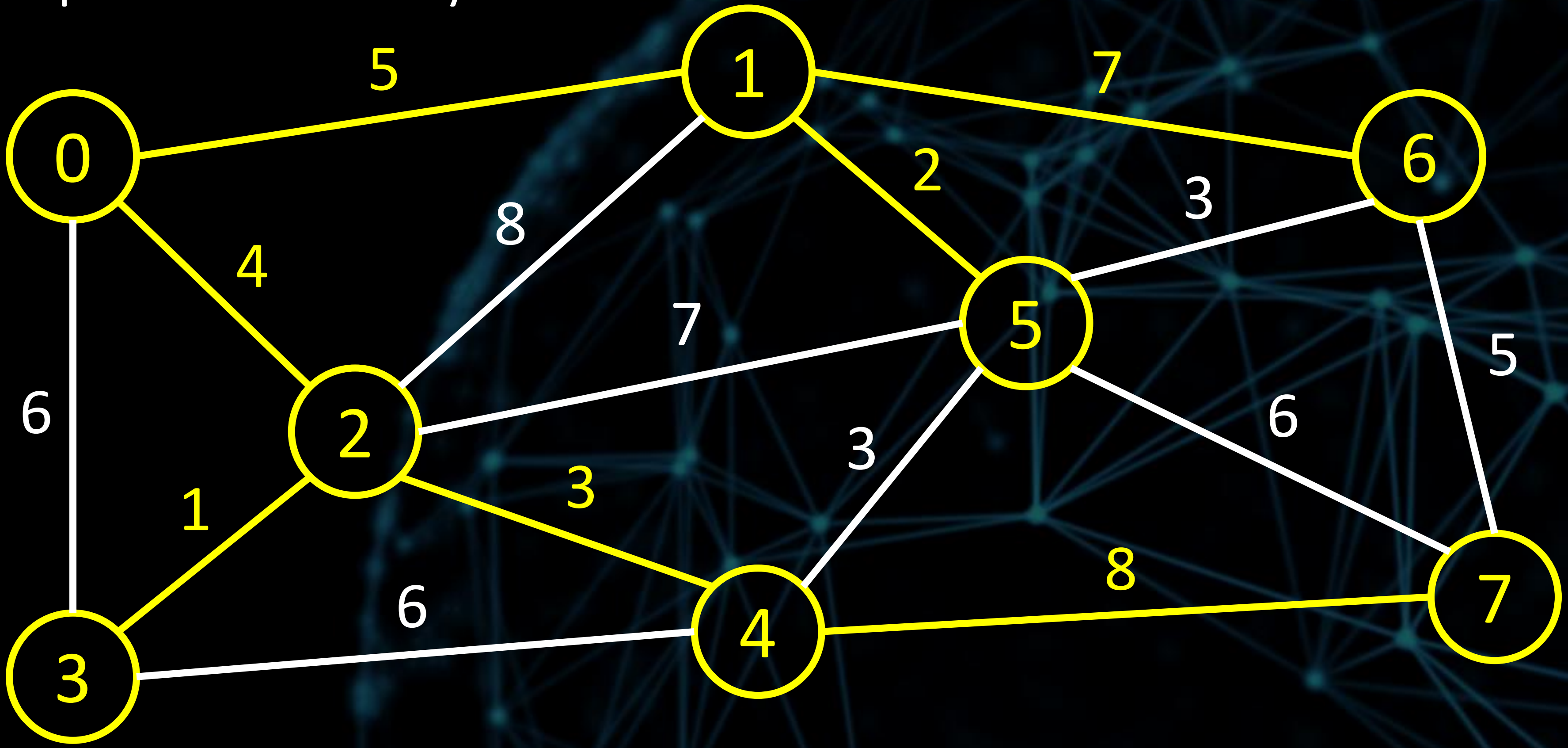
vSet = {5, 6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	15
pred	-1	0	0	2	2	1	1	4

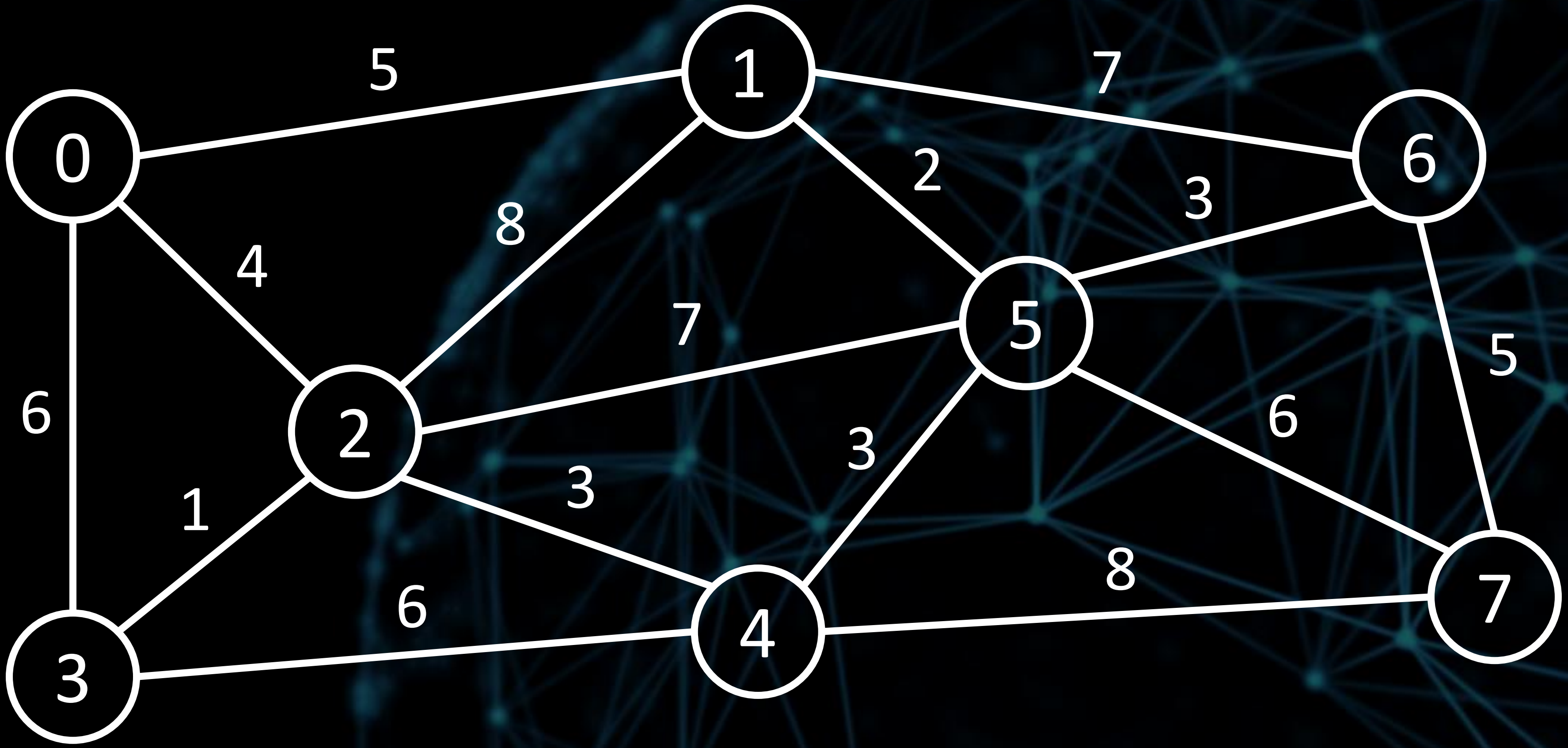
vSet = {5, 6, 7}



It’s your turn to perform one more iteration of the algorithm.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	12	15
pred	-1	0	0	2	2	1	1	4

vSet = {5, 6, 7}

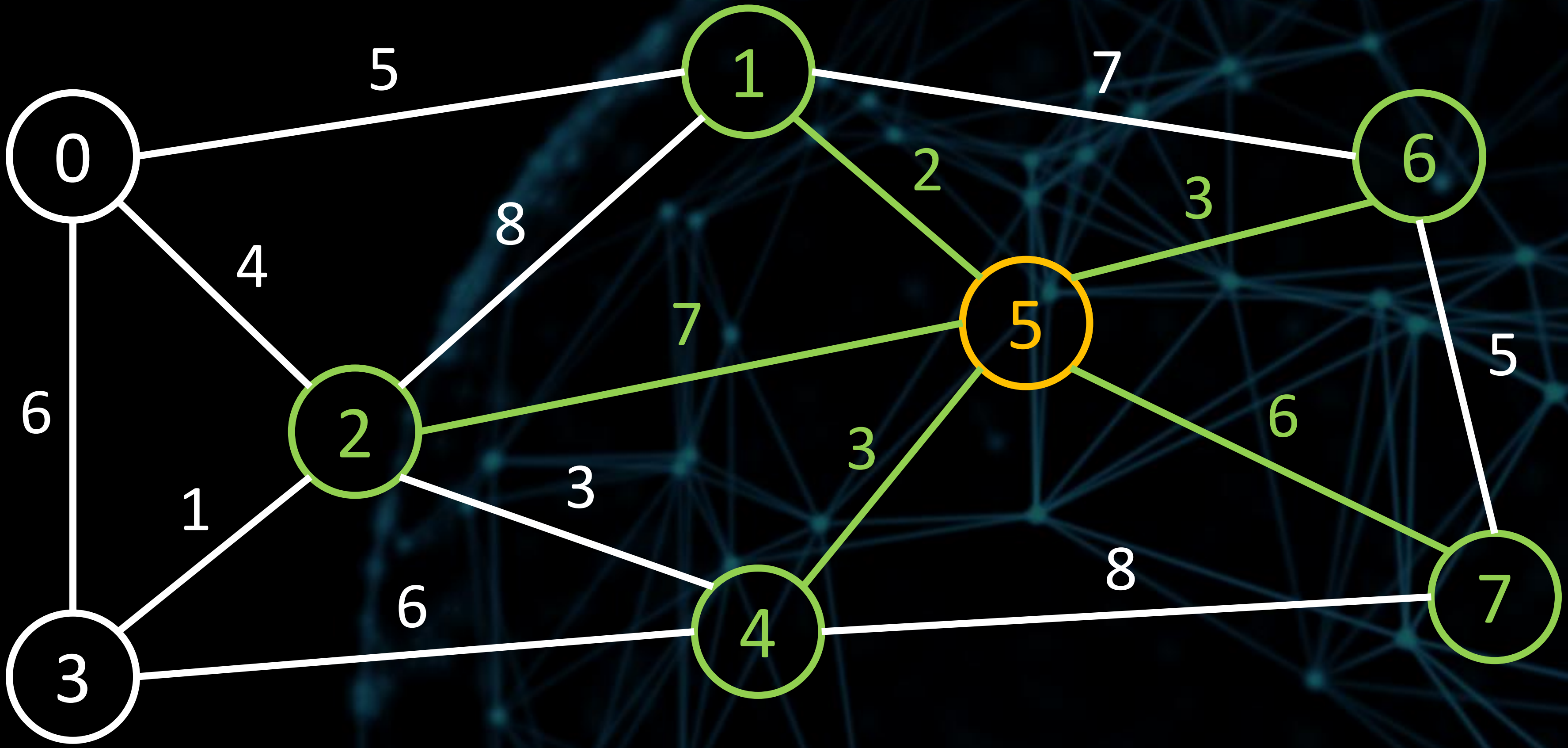


Check your answer on the next slide…

Step 1: We chose vertex 5.
Step 2: We made all necessary updates to the dist and pred array (two updates were made)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

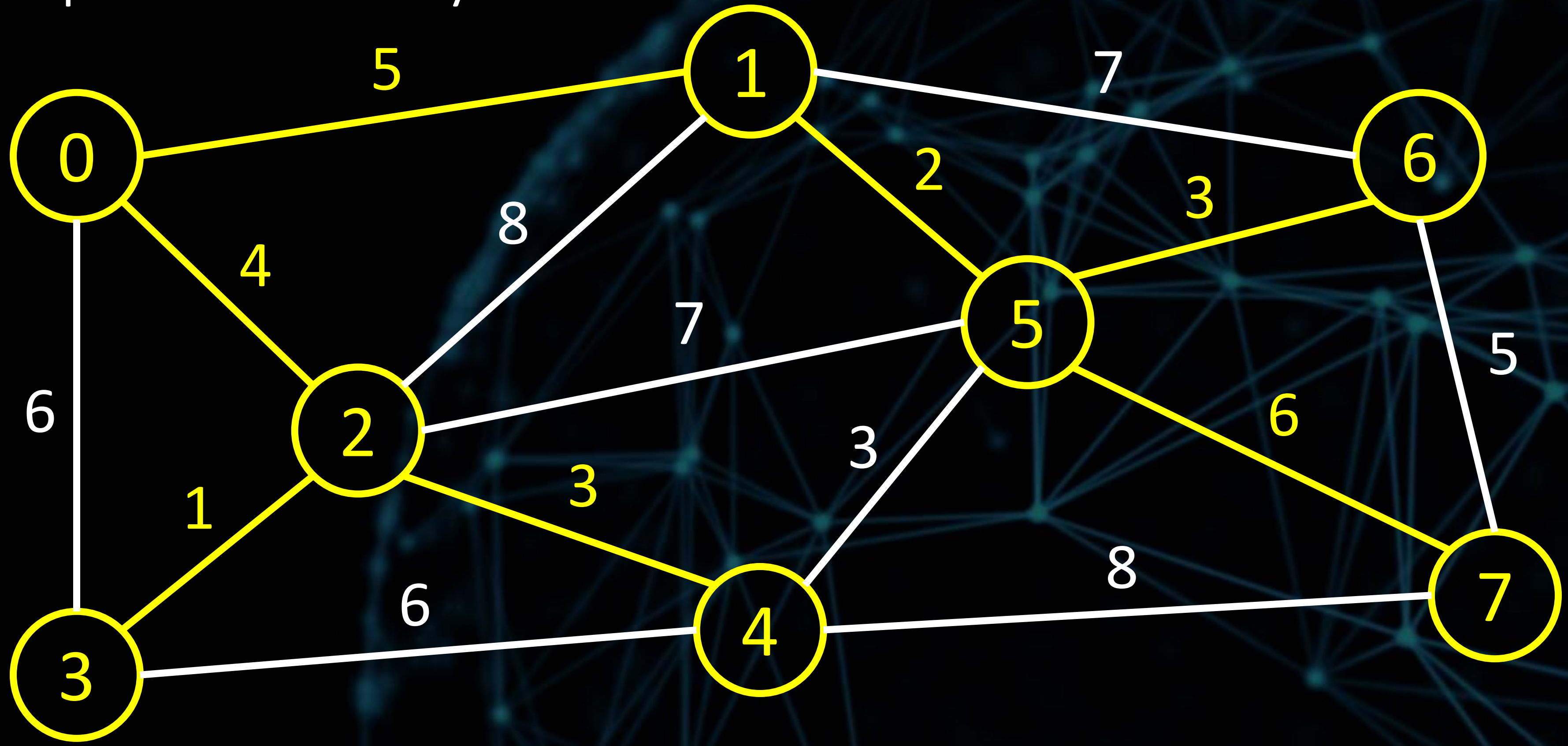
vSet = {6, 7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array.

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

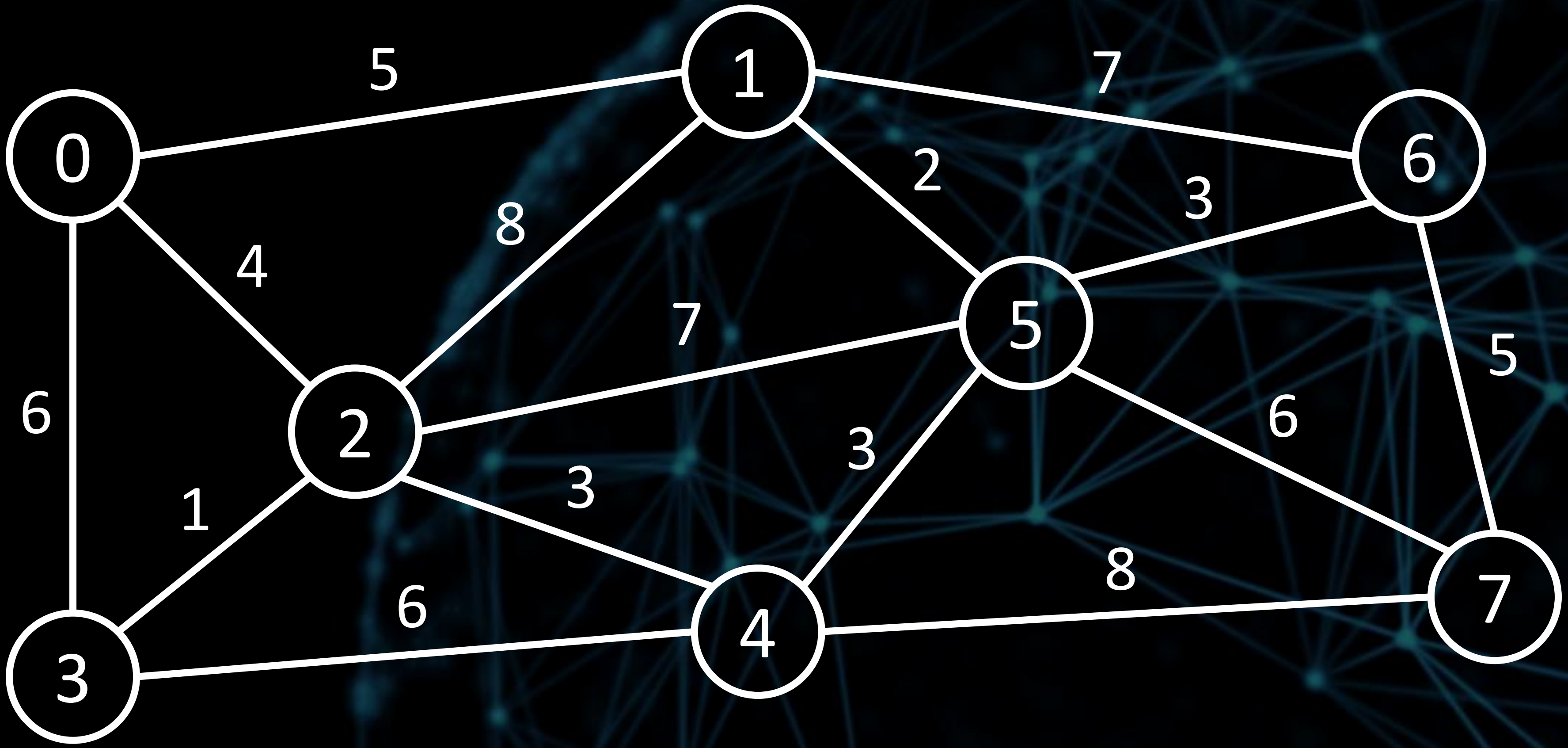
vSet = {6, 7}



It’s your turn to perform one more iteration of the algorithm.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

vSet = {6, 7}

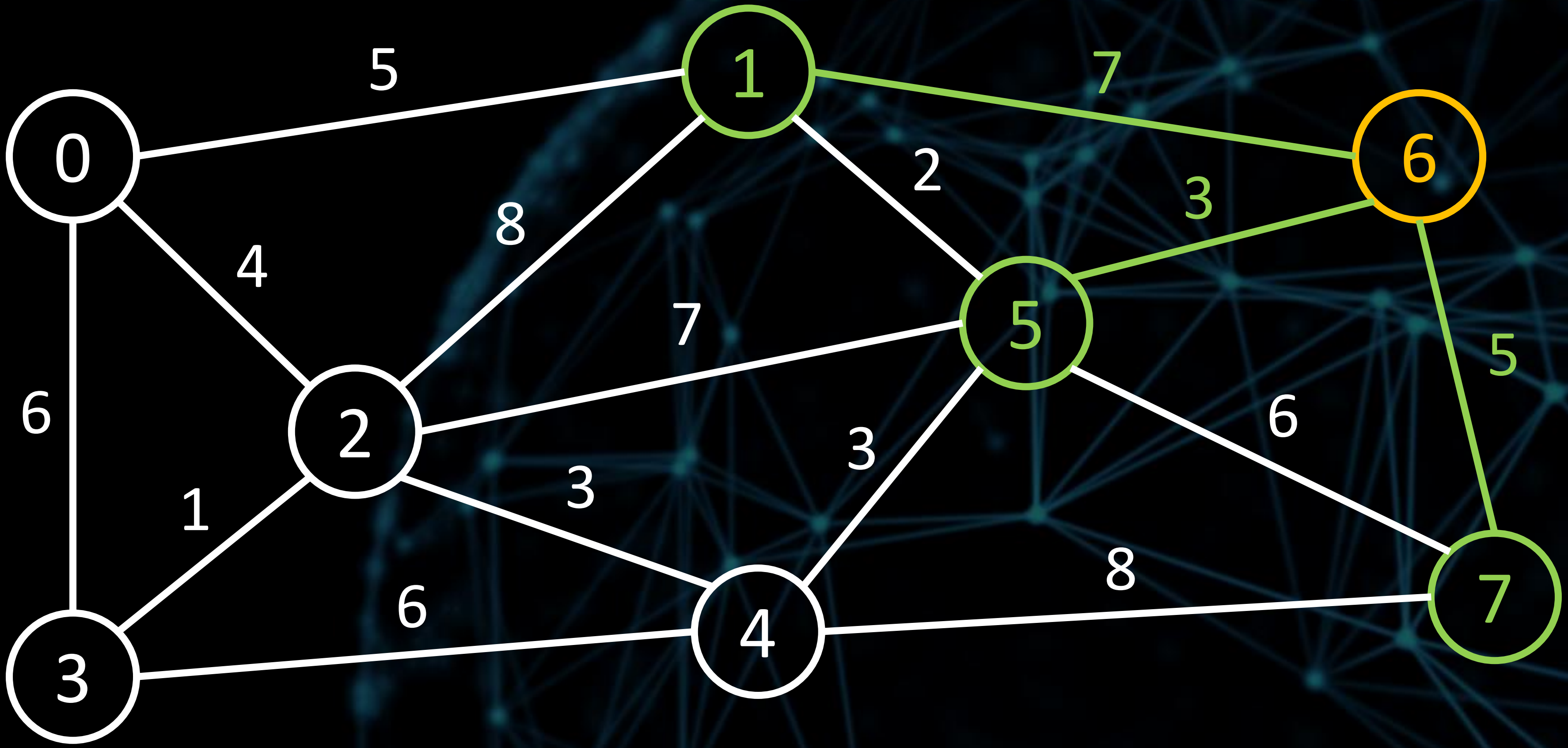


Check your answer on the next slide…

Step 1: We chose vertex 6.
Step 2: We made all necessary updates to the dist and pred array (NO updates were made!)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

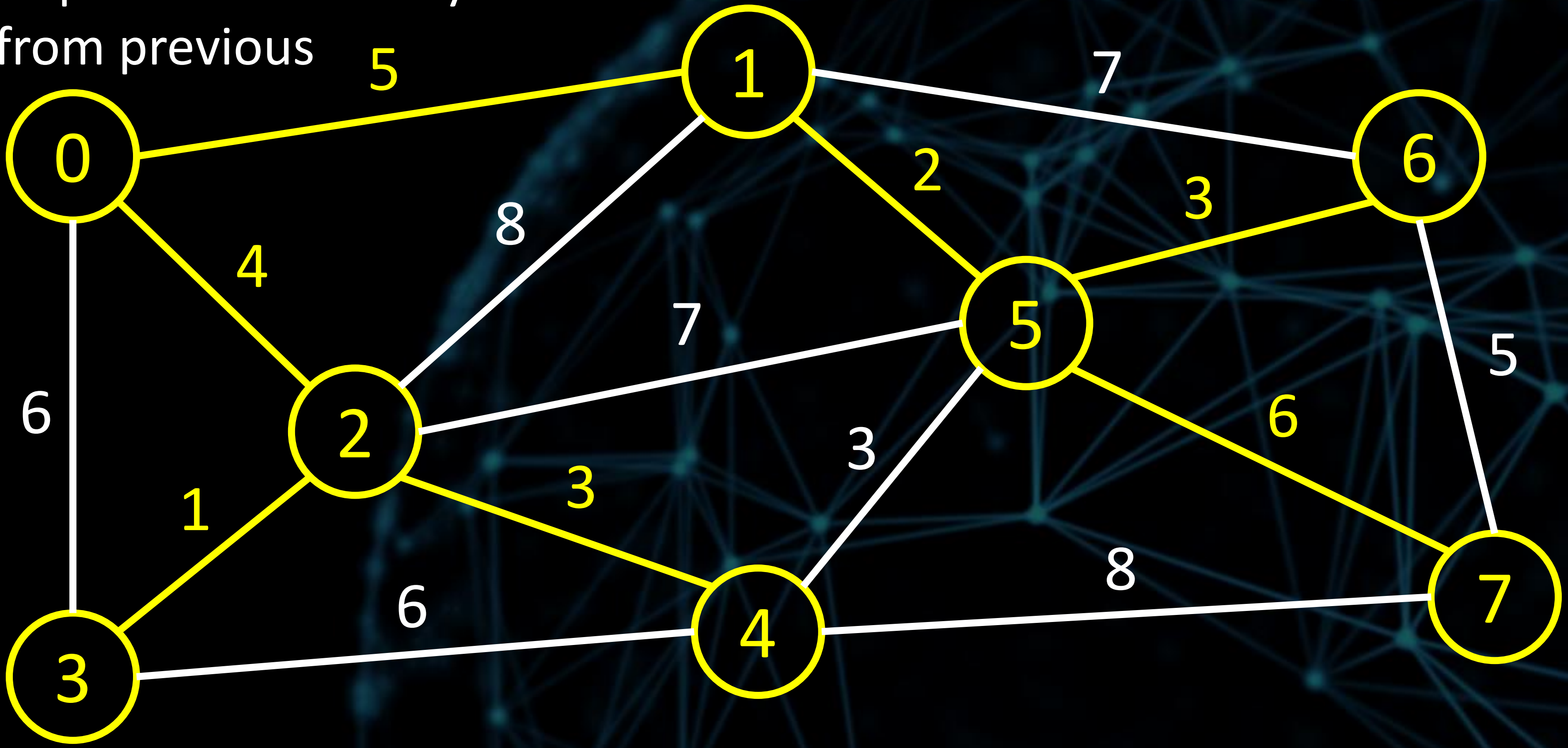
vSet = {7}



Step 3:
Repeat steps 1 and 2 until vSet is empty.
yellow shows the shortest paths from 0 to all other vertices so far, built from the predecessor array (no change from previous iteration)

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

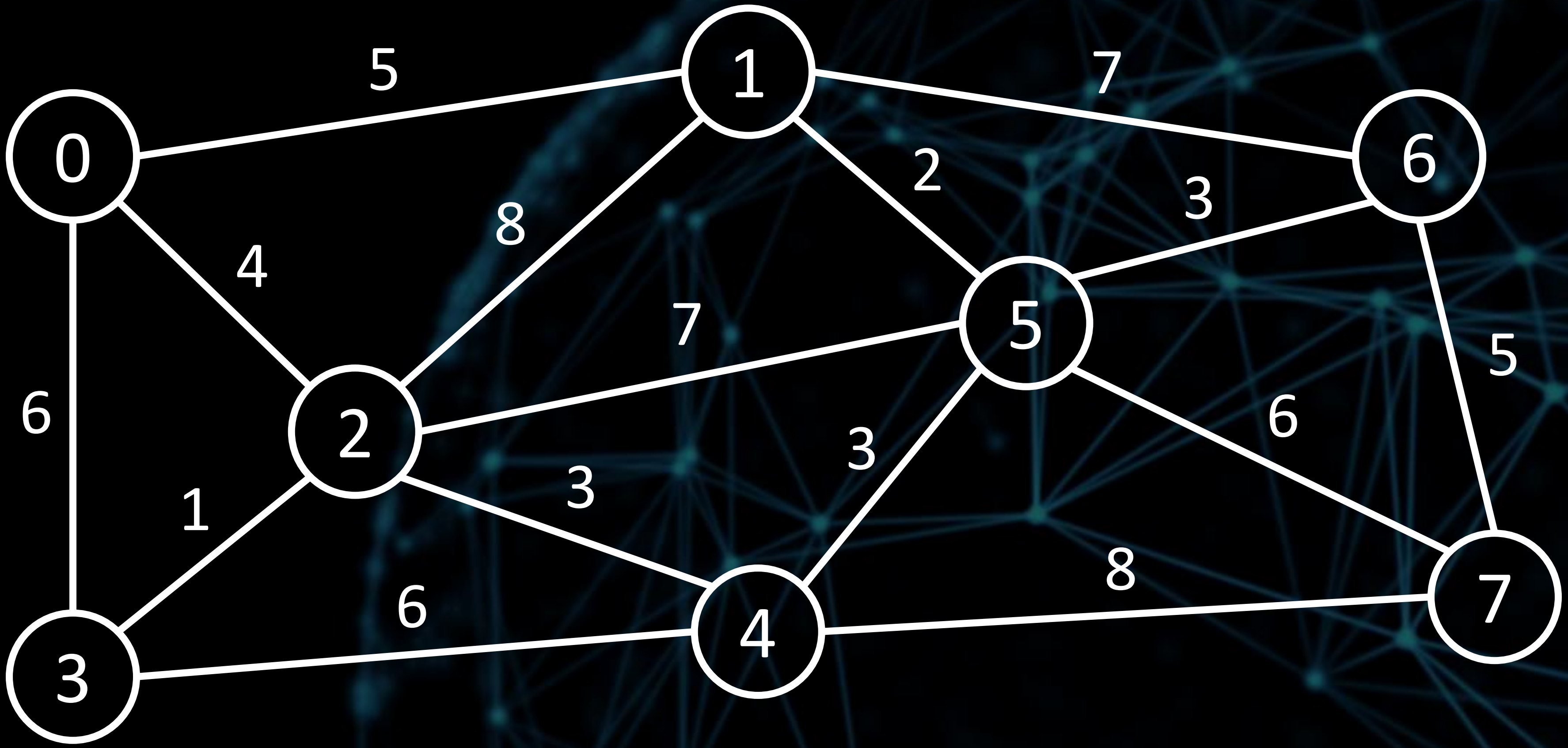
vSet = {7}



Only one more iteration left!
What do you think the result will be?

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

vSet = {7}

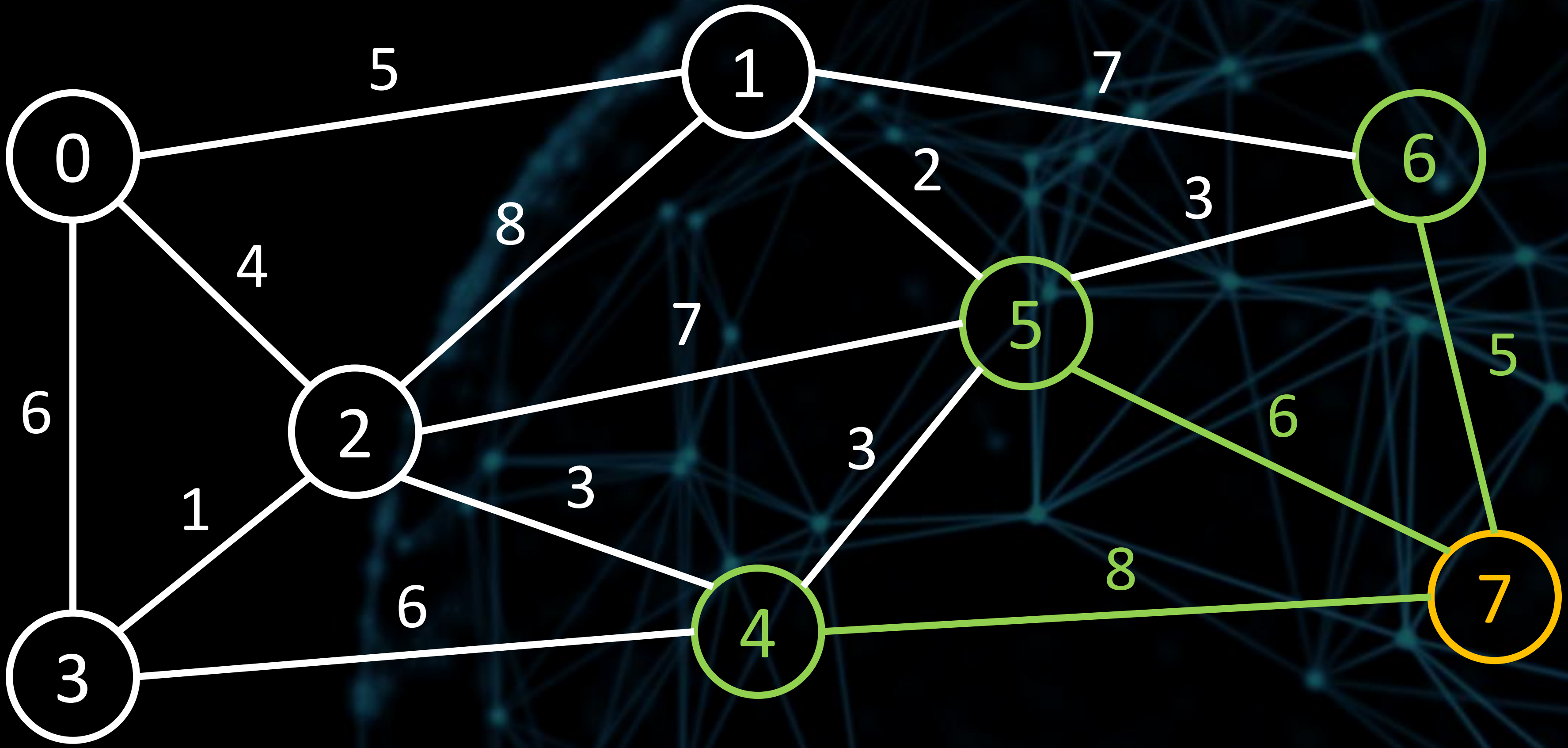


Check your answer on the next slide…

Step 1: We had no choice but to choose 7.
Step 2: No changes were made to the dist and pred arrays (not surprising).

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

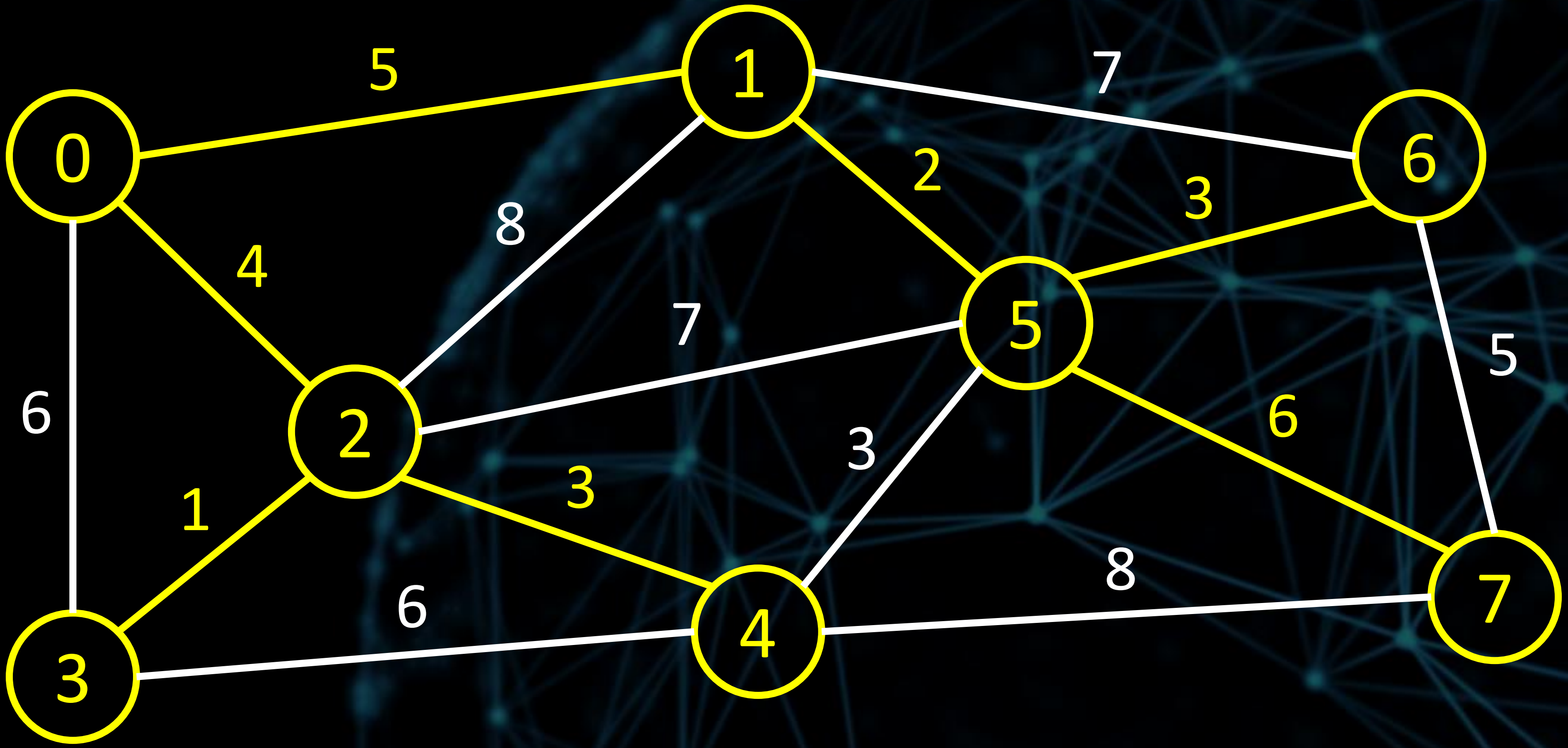
vSet = {}



Step 3:
Here are the final dist and pred arrays.

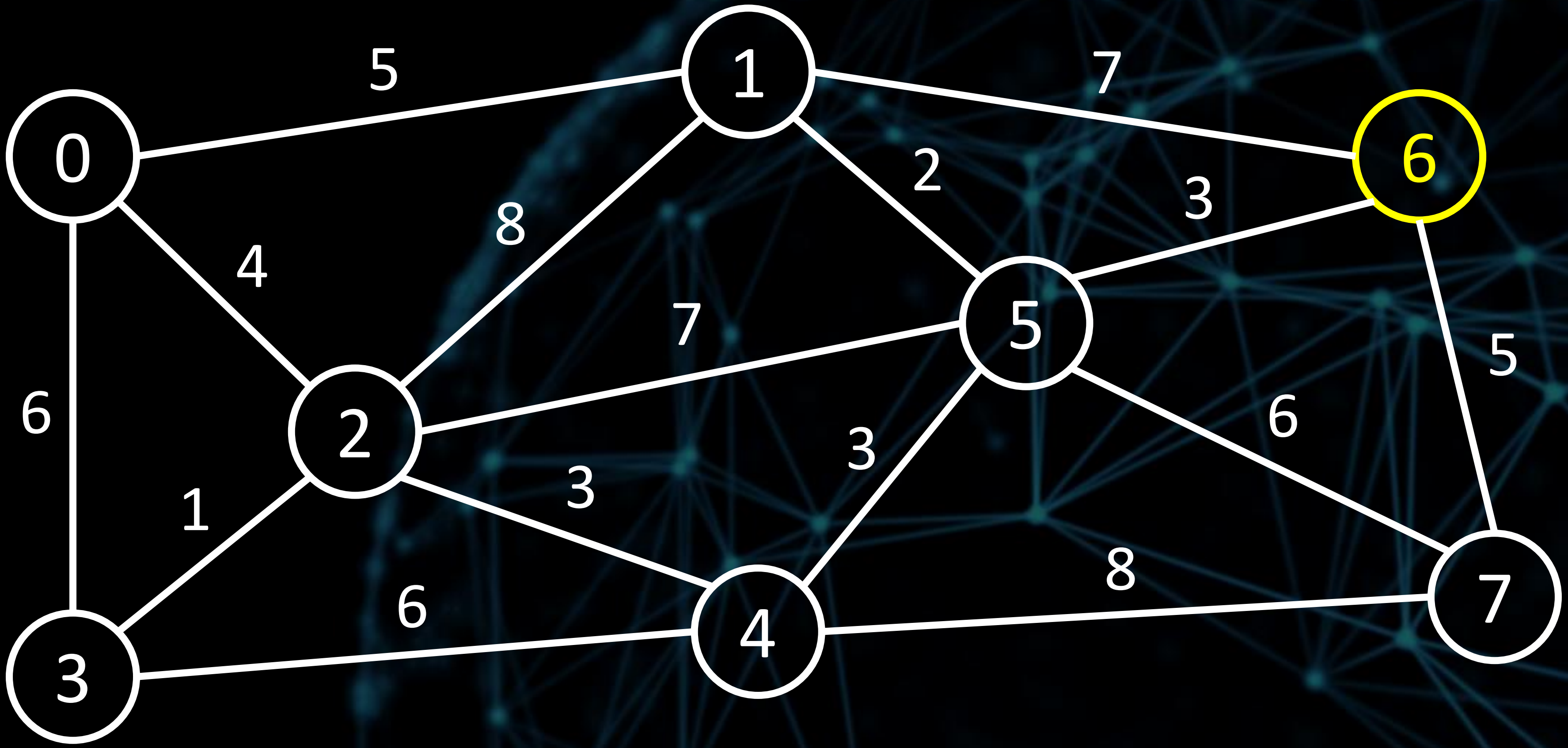
	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5

vSet = {}



Now suppose we wanted to find the shortest weighted path from vertex 0 to vertex 6. How do we do this, given the dist and pred arrays?

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5



Check your answer on the next slide…

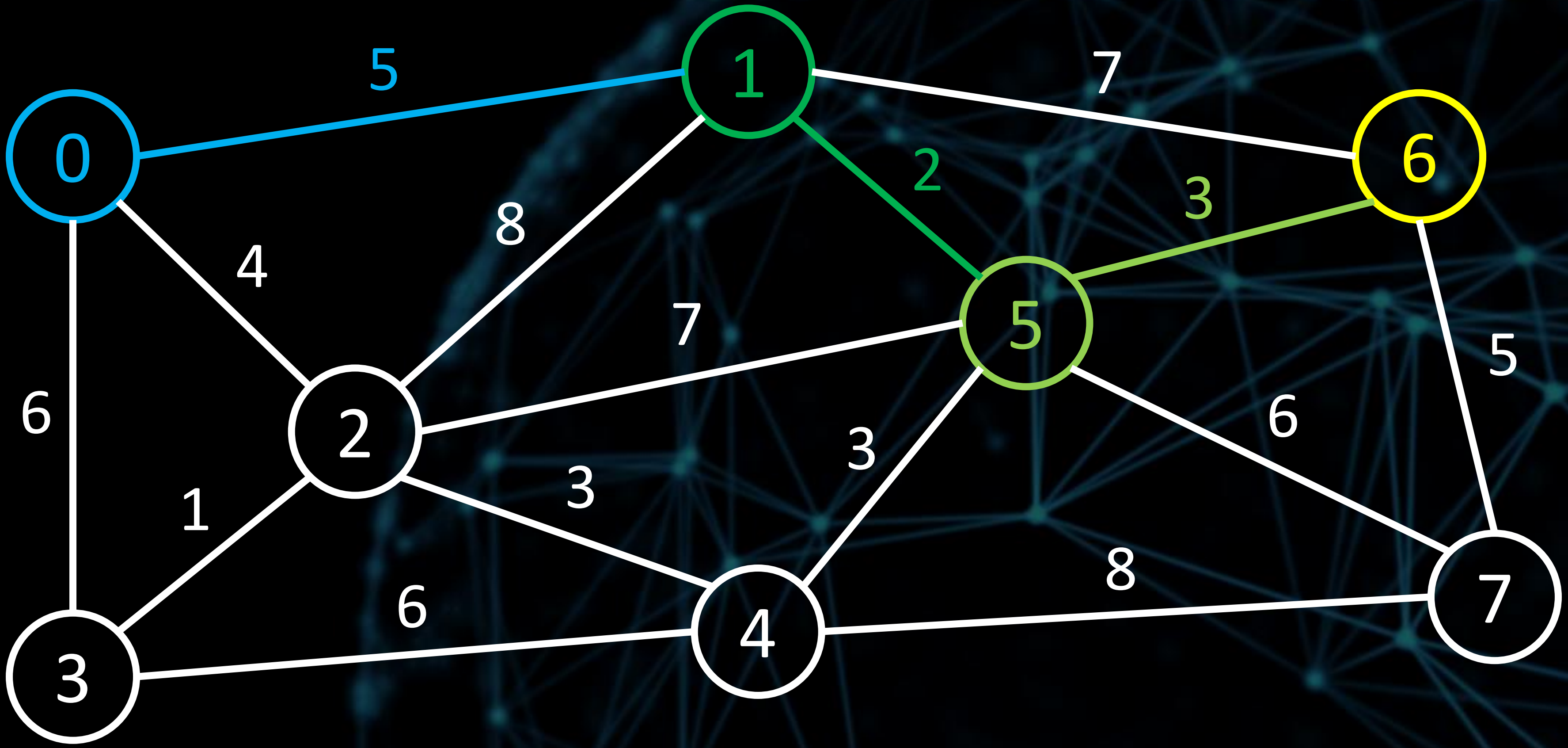
We follow the predecessors from vertex 6 until we reach vertex 0.

6 → 5 → 1 → 0

This gives us the path in reverse, so we just have to reverse it…

0 → 1 → 5 → 6

	0	1	2	3	4	5	6	7
dist	0	5	4	5	7	7	10	13
pred	-1	0	0	2	2	1	5	5



Time complexity of Dijkstra’s algorithm

There are two main contributors to the time complexity of Dijkstra’s algorithm:

1. Determining which vertex to remove from vSet
2. Exploring the neighbours of each vertex and updating the dist and pred arrays

First, let’s suppose we don’t use a priority queue.

1. Determining which vertex to remove from vSet

In the first iteration, we need to loop through all V elements of vSet to find which one we should remove.

vSet = {0, 1, 2, 3, 4, 5, 6, 7}

In the second iteration, we need to loop through $V - 1$ elements of vSet to find which one we should remove.

vSet = {1, 2, 3, 4, 5, 6, 7}

And so on…

1. Determining which vertex to remove from vSet

So the cost is

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

which is equal to

$$V\,(V + 1)\,/\,2$$

which is

$$O(V^2)$$

2. Exploring the neighbours of each vertex and updating the dist and pred arrays

Exploring the neighbours of each vertex is the same as exploring the edges from each vertex. It is known from graph theory that the sum of the degrees of all vertices in a graph is

$$2E$$

The updates to the dist and pred arrays can be done in constant time, since they just involve indexing the arrays. So the cost is

$$O(2E) = O(E)$$

So, if we don’t use a priority queue, the time complexity is $O(E + V^2) = O(V^2)$

Now suppose that we used a priority queue.

If we use a priority queue, then as well as keeping a dist array, we also store the distances to each vertex in the priority queue.

At the beginning of each iteration, when we are deciding which vertex to explore, we remove the vertex with the highest priority (i.e., smallest distance) from the PQ.

Also, whenever we update the distance of a vertex, we also update the vertex in the PQ.

So the priority queue acts as our vSet.

1. Determining which vertex to remove from vSet

For an efficient priority queue implementation, the cost of removing of the highest priority element is O(log N), where N is the number of items in the priority queue.

The queue starts with V elements, and each iteration, we remove one vertex, so the overall cost is

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

which is

$$O(V \log V)$$

(click here for details)

2. Exploring the neighbours of each vertex and updating the dist and pred arrays

Exploring the neighbours of each vertex is the same as exploring the edges from each vertex. It is known from graph theory that the sum of the degrees of all vertices in a graph is

$$2E$$

2. Exploring the neighbours of each vertex and updating the dist and pred arrays

Updating the dist and pred arrays is O(1), and updating an item in an efficient priority queue is also O(1).

So the cost is $O(2E) = O(E)$.

So, if we use a priority queue, the time complexity is $O(E + V \log V)$

EXTRAS

Why is ($\log V + \log(V - 1) + \log(V - 2) + \ldots + \log(1)$) equivalent to $O(V \log V)$?

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

$$= \log(V \times (V - 1) \times (V - 2) \times \ldots \times 1)$$

$$= \log(V!)$$

$$\approx V \log V - V + O(\log V)$$ (by Stirling's approximation)

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