CodeIT

Lesson 3 Objectives:

To gain an understanding of:

1. What are **undirected / directed graphs** and how we may represent and implement them

2. **Depth-first / breadth-first** traversal of graphs

3. The **topological sort algorithm** in an acyclic directed graph

Graphs

Graphs

Graphs are a useful data structure to represent network like relationships

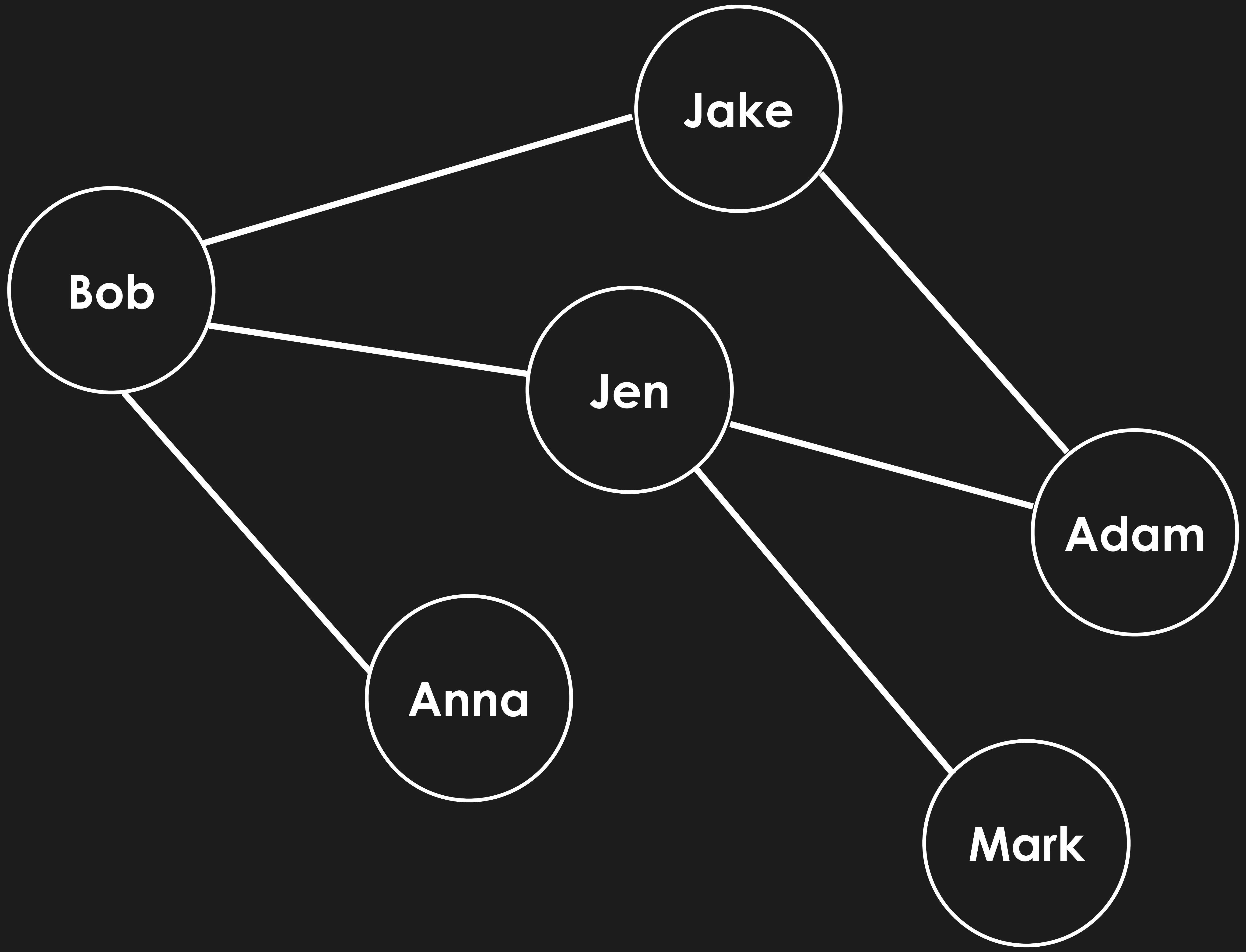
Graphs

Graphs are a useful data structure to represent network like relationships

Graphs are made up of:

- **vertices** (nodes)
- **edges** (connections)

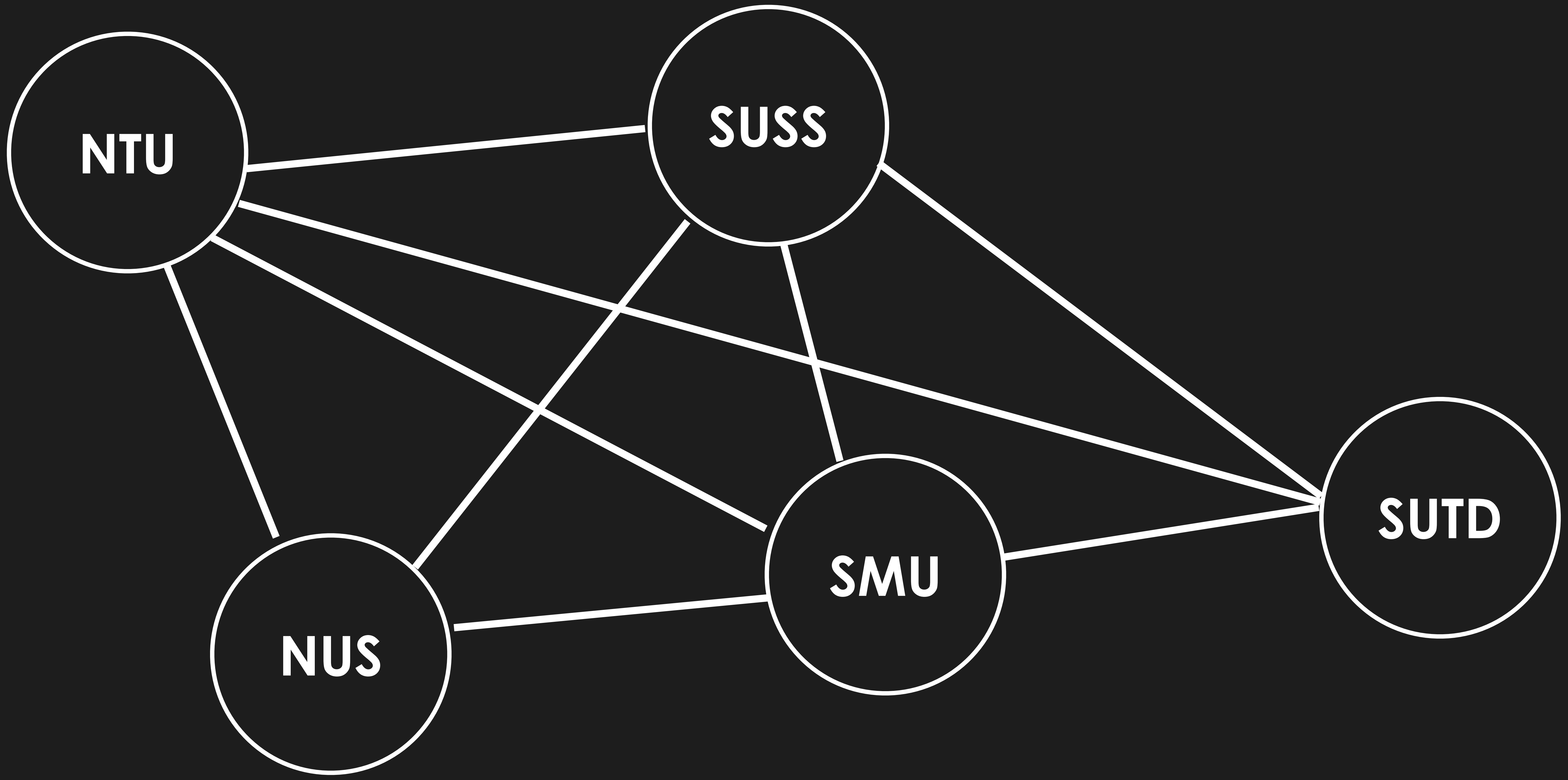
Graph Example: Social Network



vertices: Users (People)
edges: Friendships

CodeIT

Graph Example: Roads



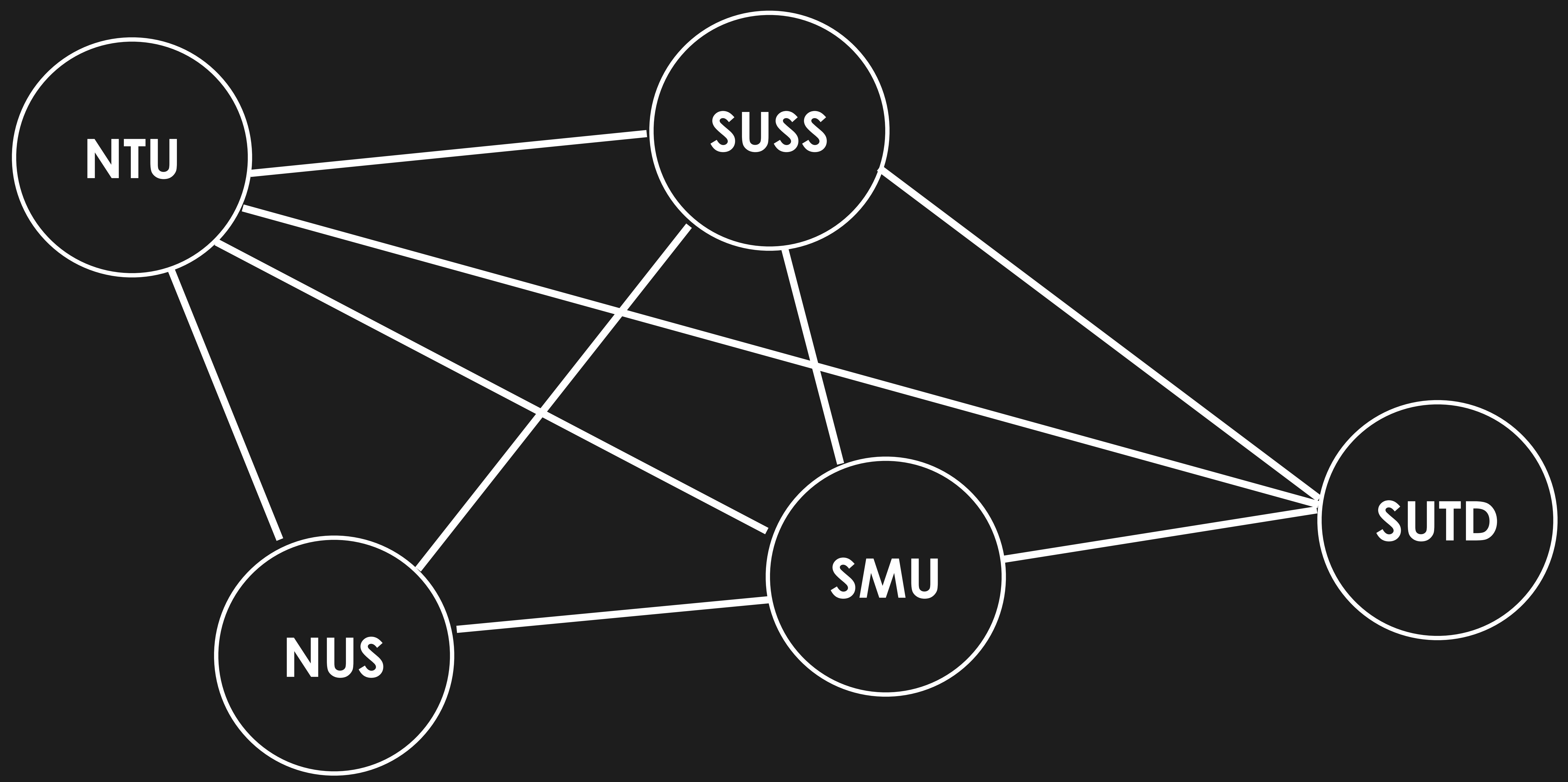
vertices: Addresses (Schools)
edges: Roads

Why represent these as graphs?

Why represent these as graphs?

By using graphs to represent complex networks, we have a toolbox of algorithms to help answer important questions

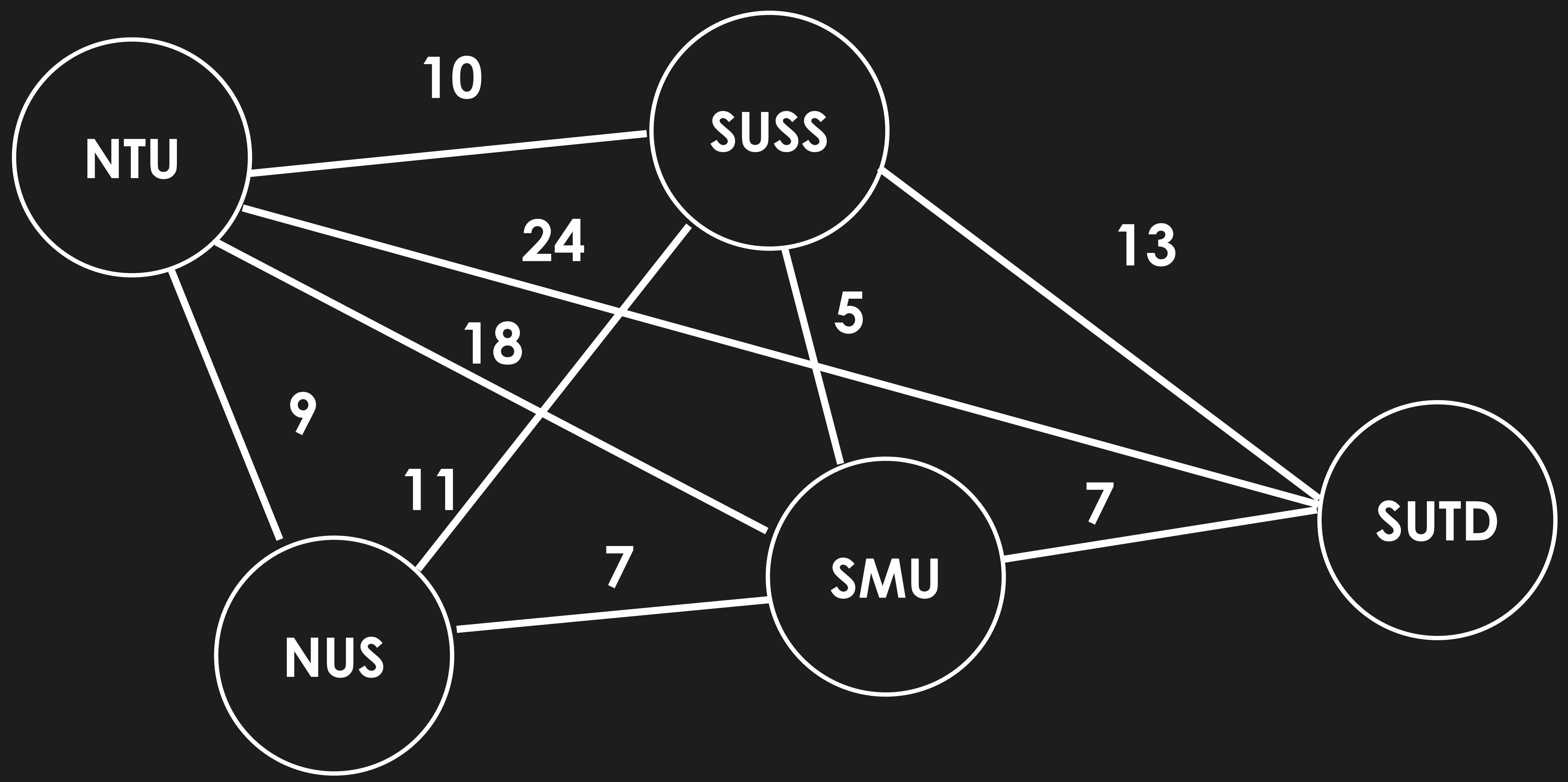
Graph Example: Roads



nodes: Addresses (Schools)
edges: Roads

What is the shortest path from NTU to SUTD?

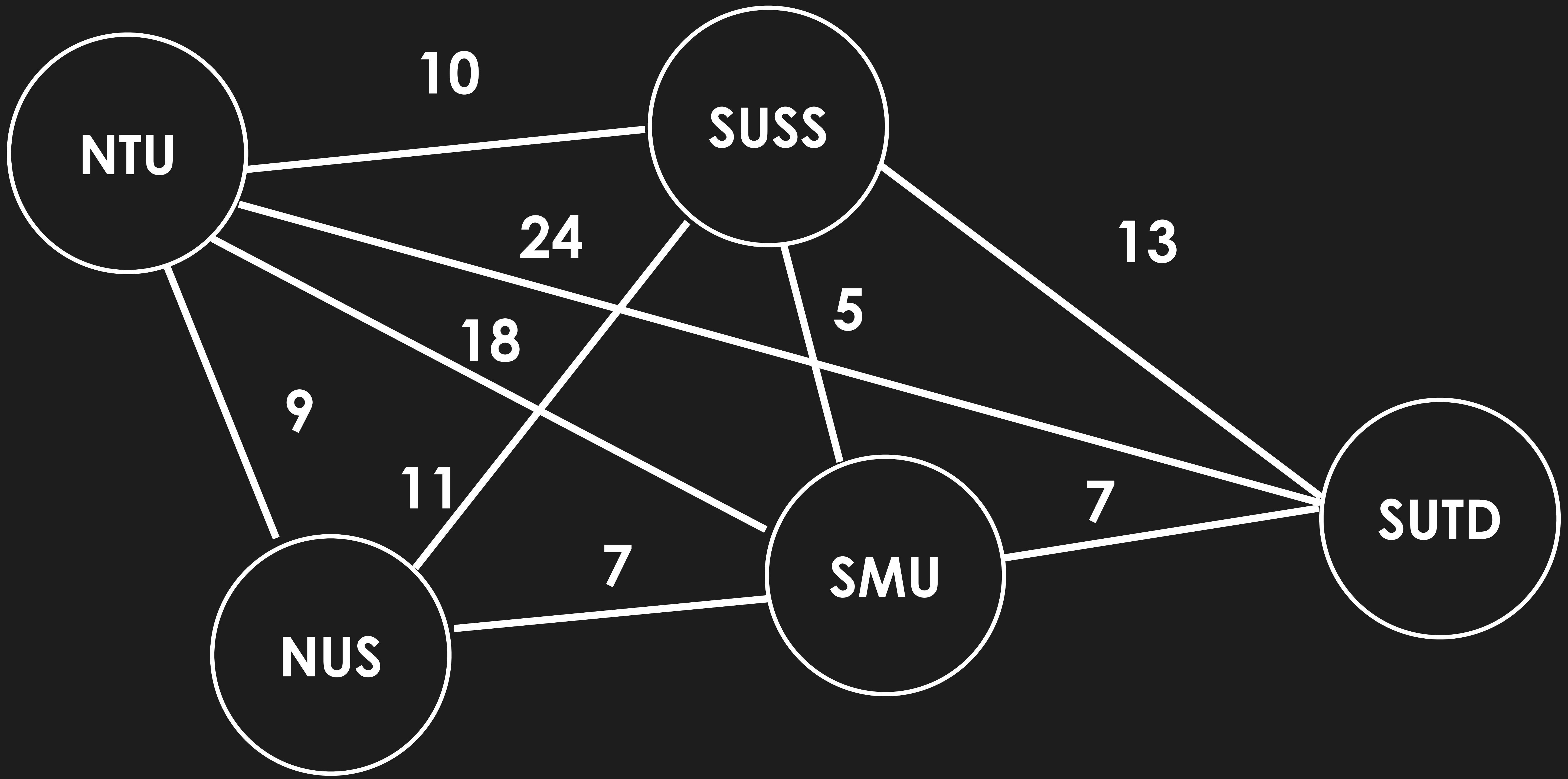
Graph Example: Roads



vertices: Addresses (Schools)
edges: Roads

Let's give each edge a weight representing the distance of the road

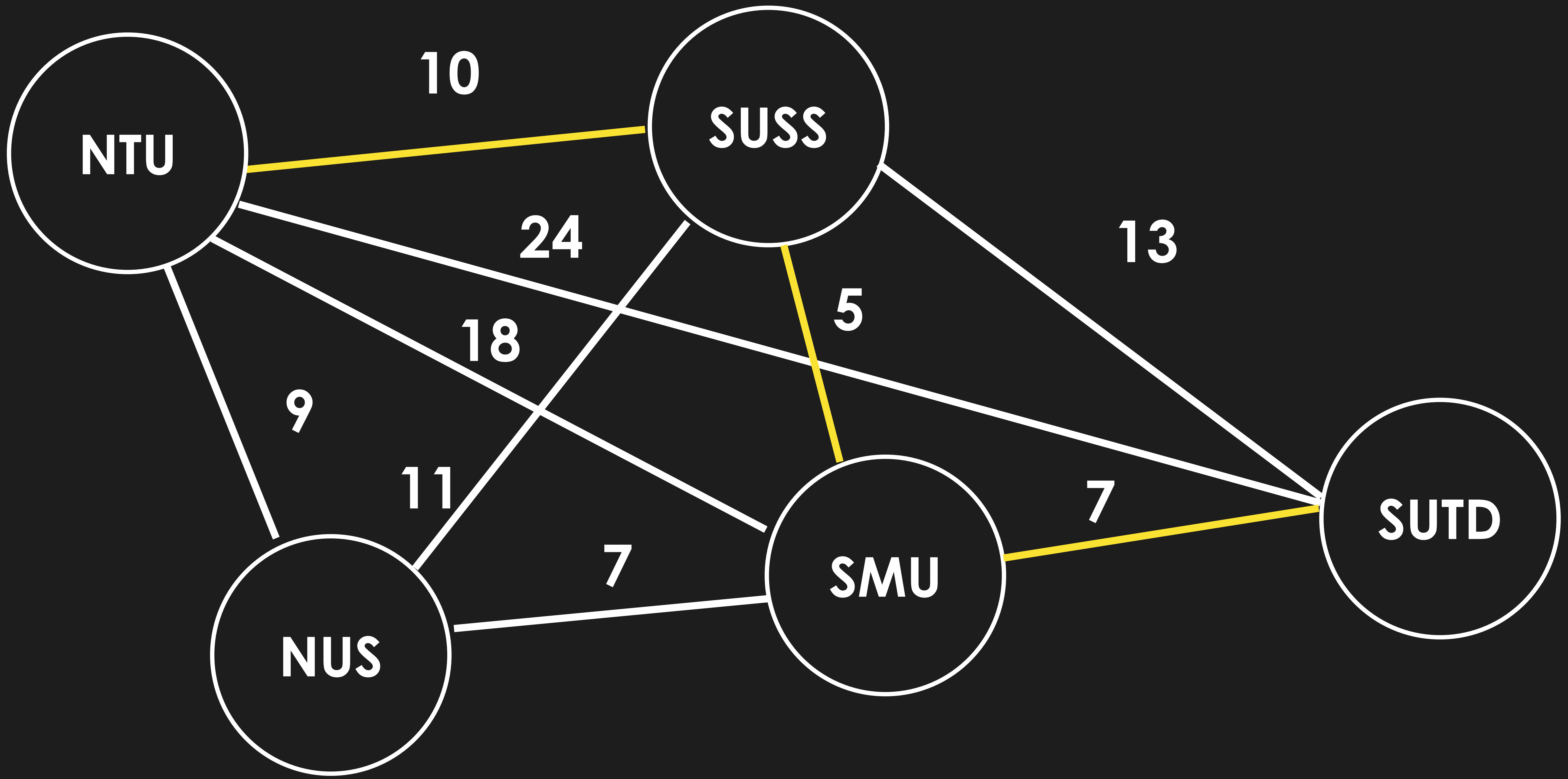
Graph Example: Roads



vertices: Addresses (Schools)
edges: Roads

What is the shortest path from NTU to SUTD?

Graph Example: Roads

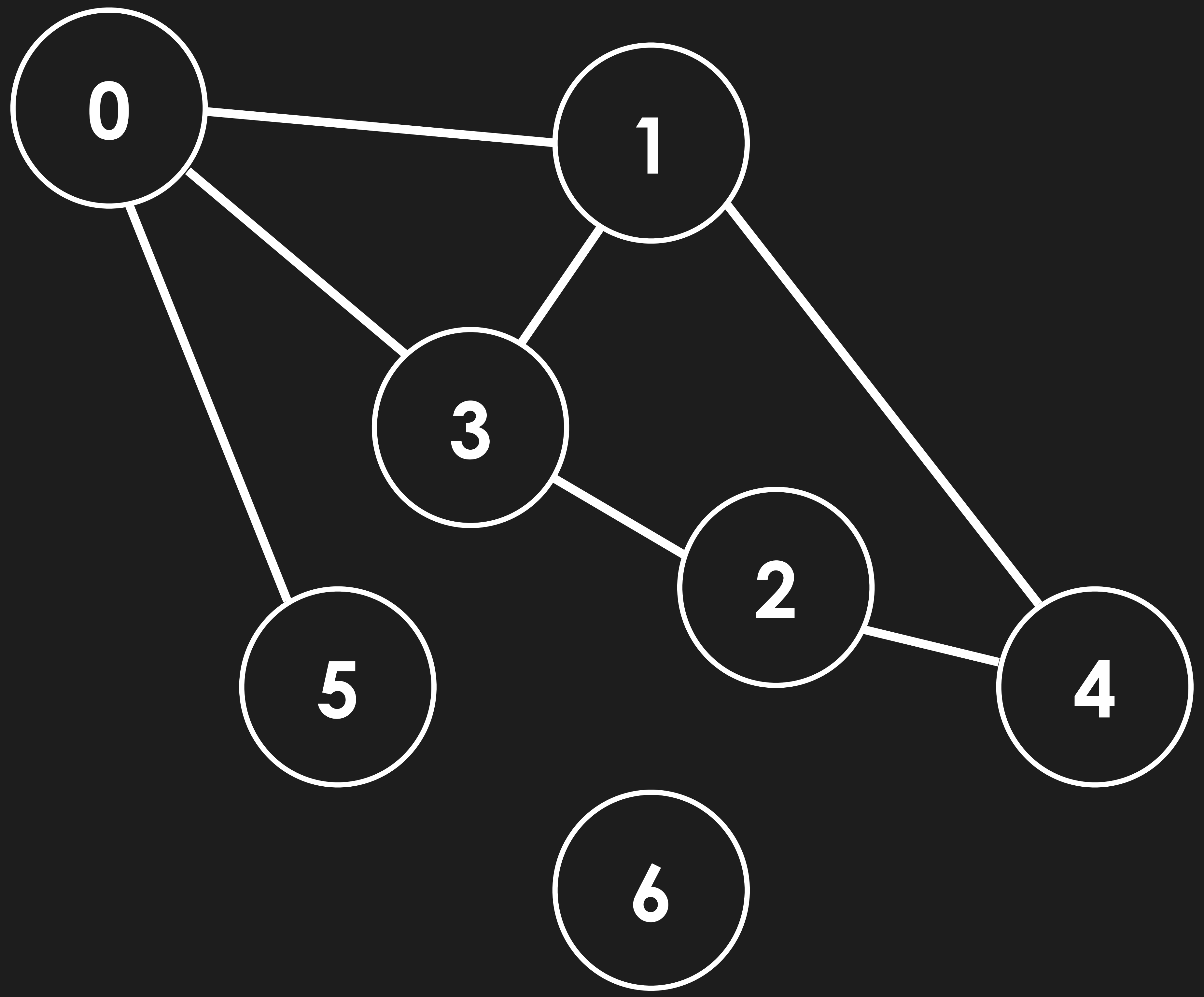


vertices: Addresses (Schools)
edges: Roads

What is the shortest path from NTU to SUTD?

Representation of Graphs

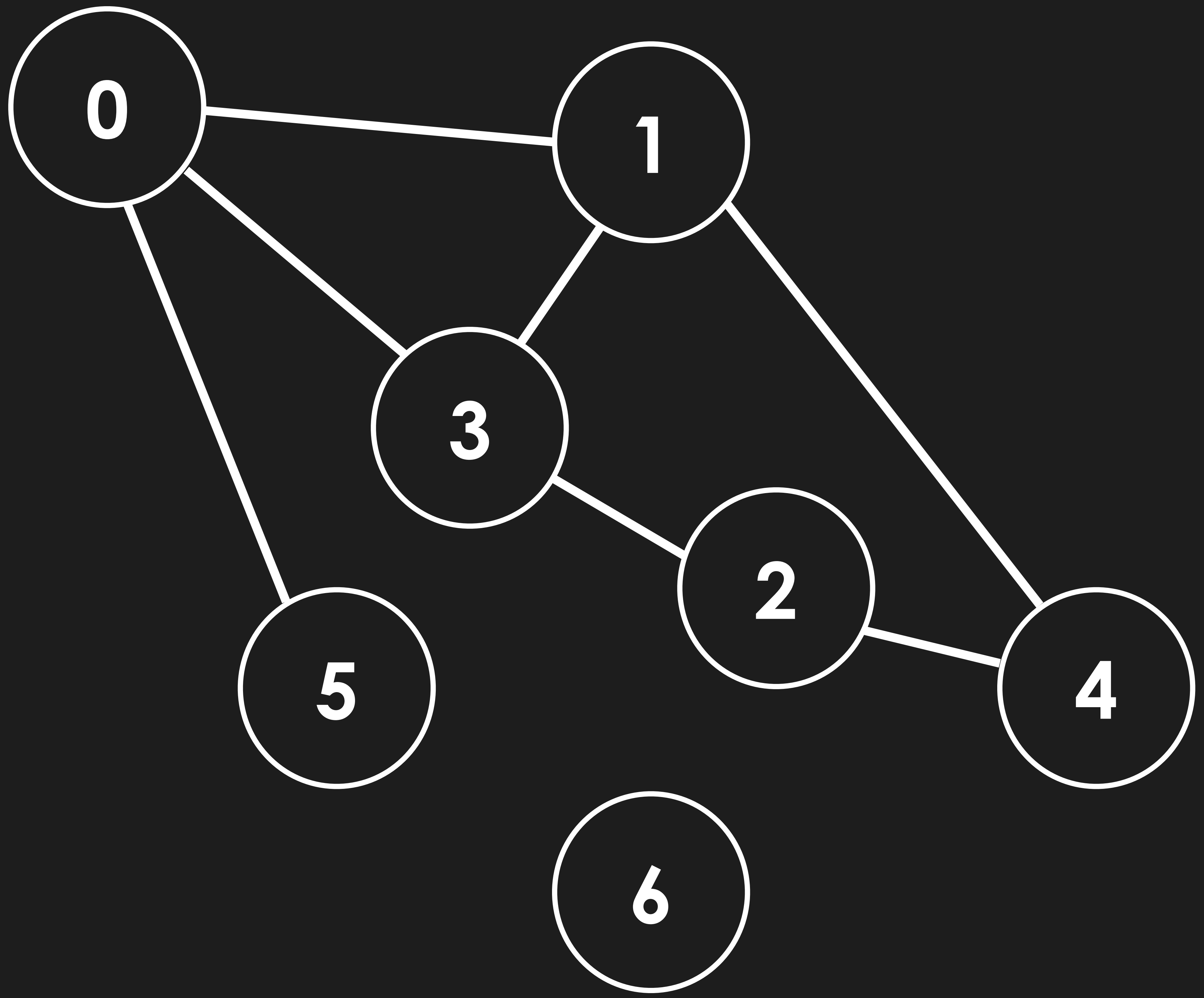
Let's assume we have the following graph:



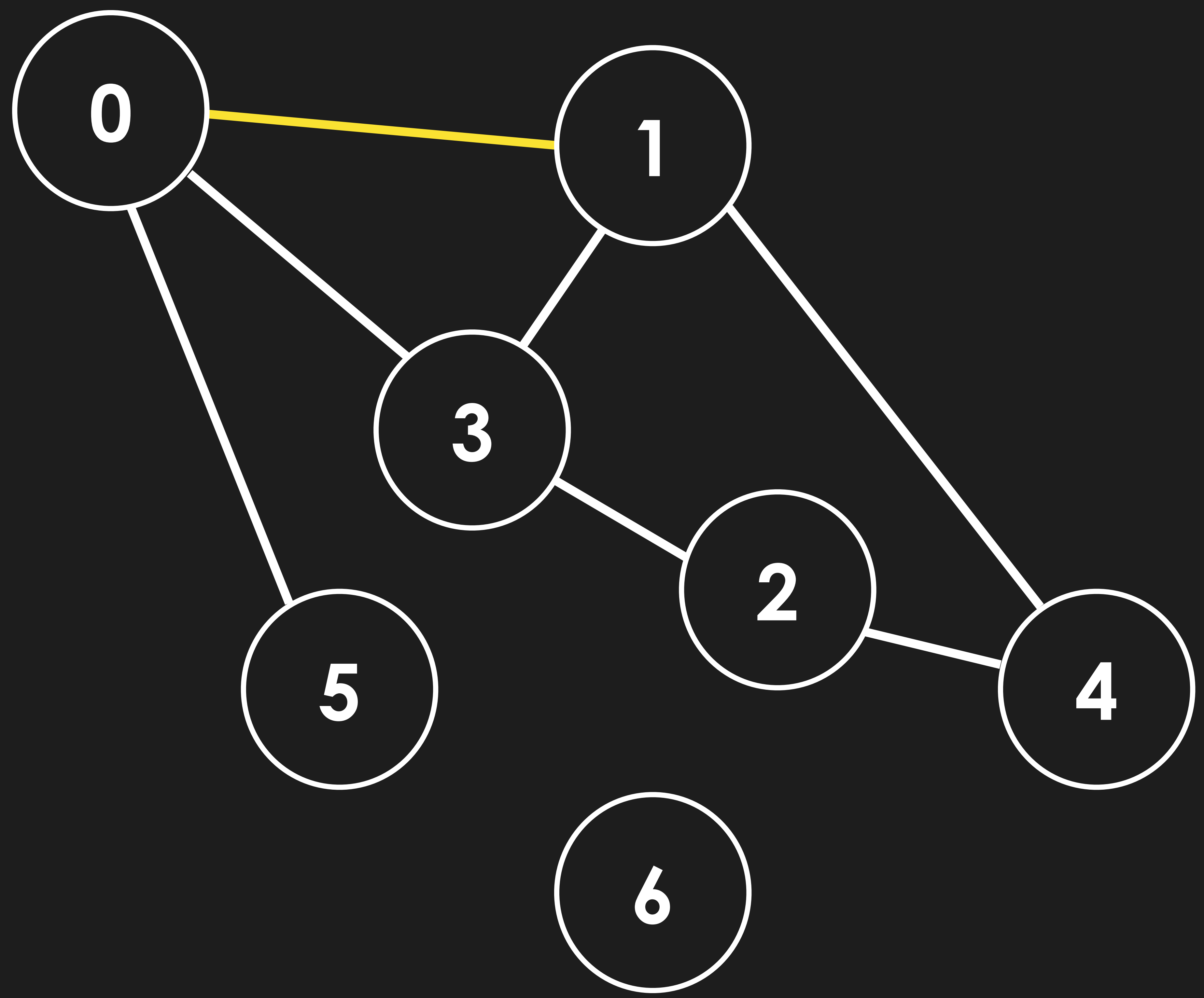
How might we represent this?

Adjacency Matrix

Adjacency Matrix

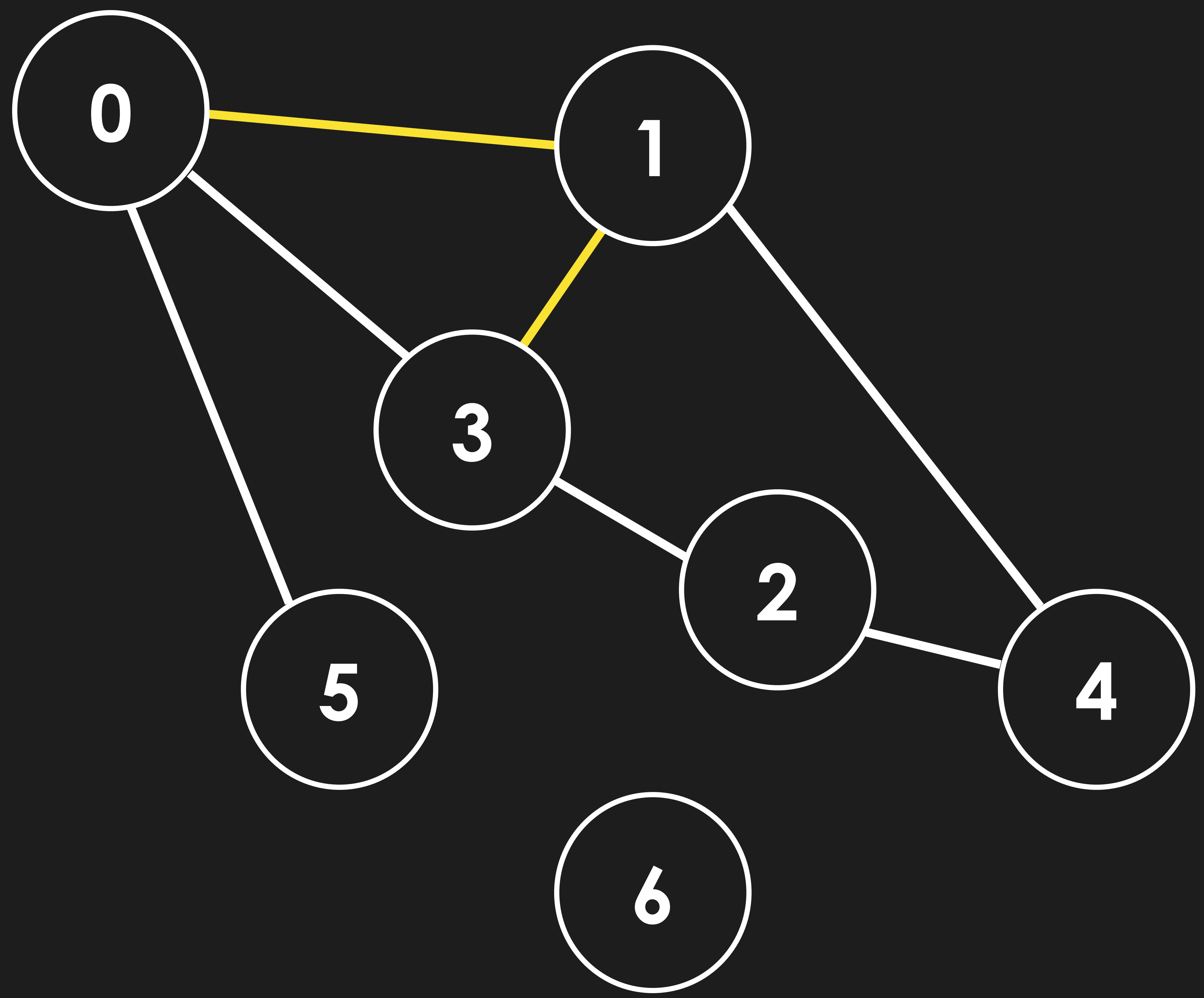


	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0



Adjacency Matrix

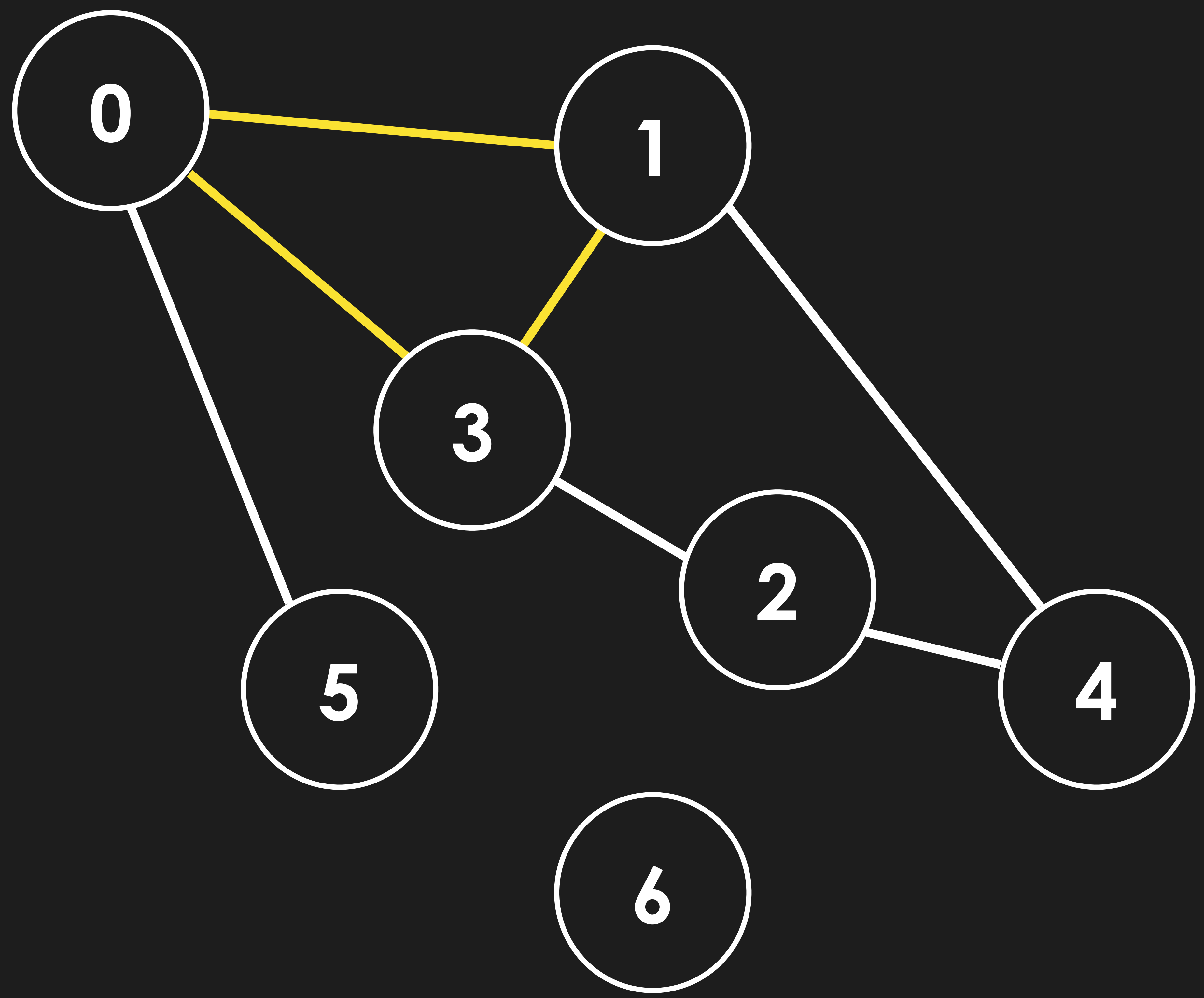
	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0



Adjacency Matrix

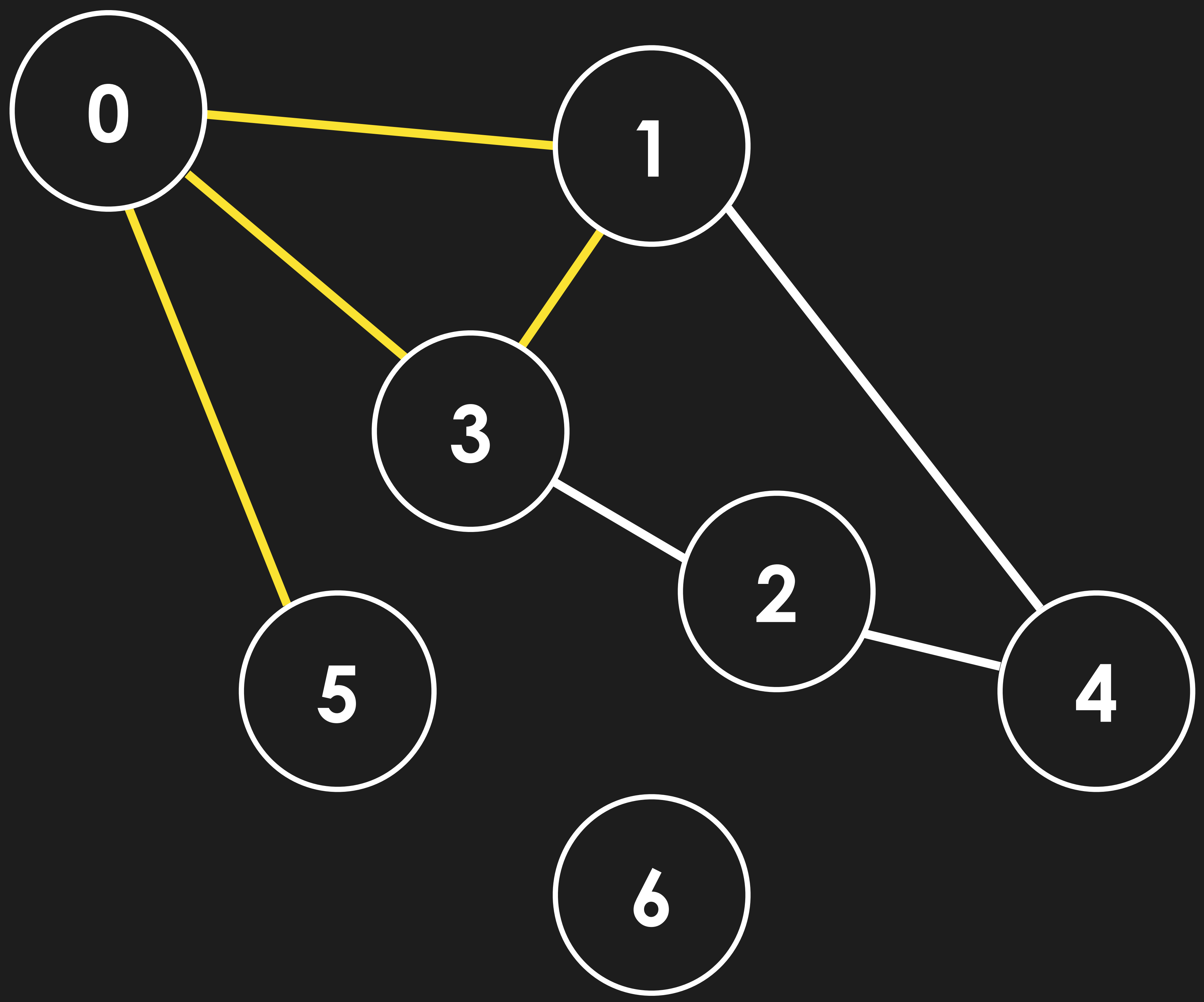
	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0
2	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

CodeIT



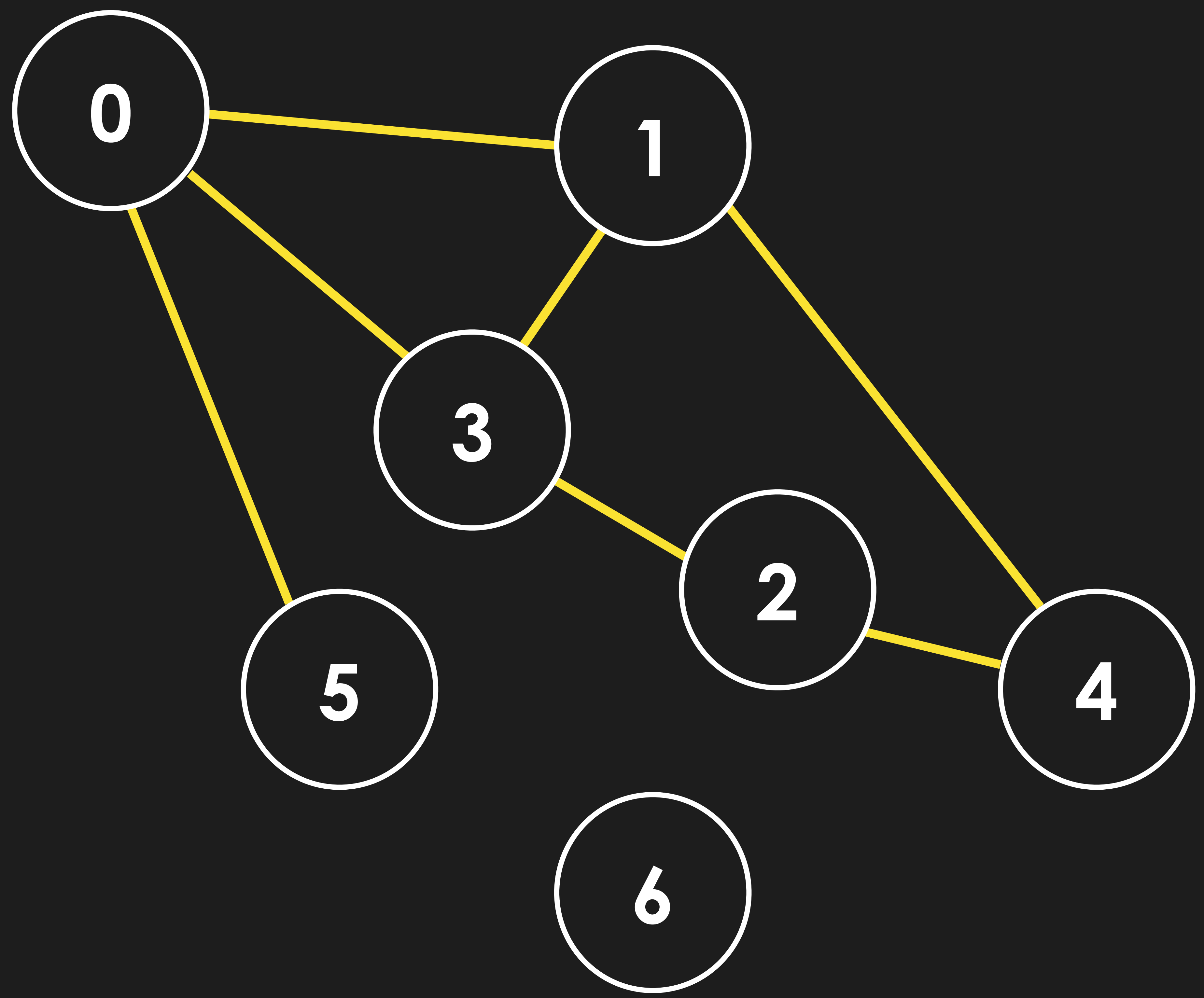
Adjacency Matrix

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



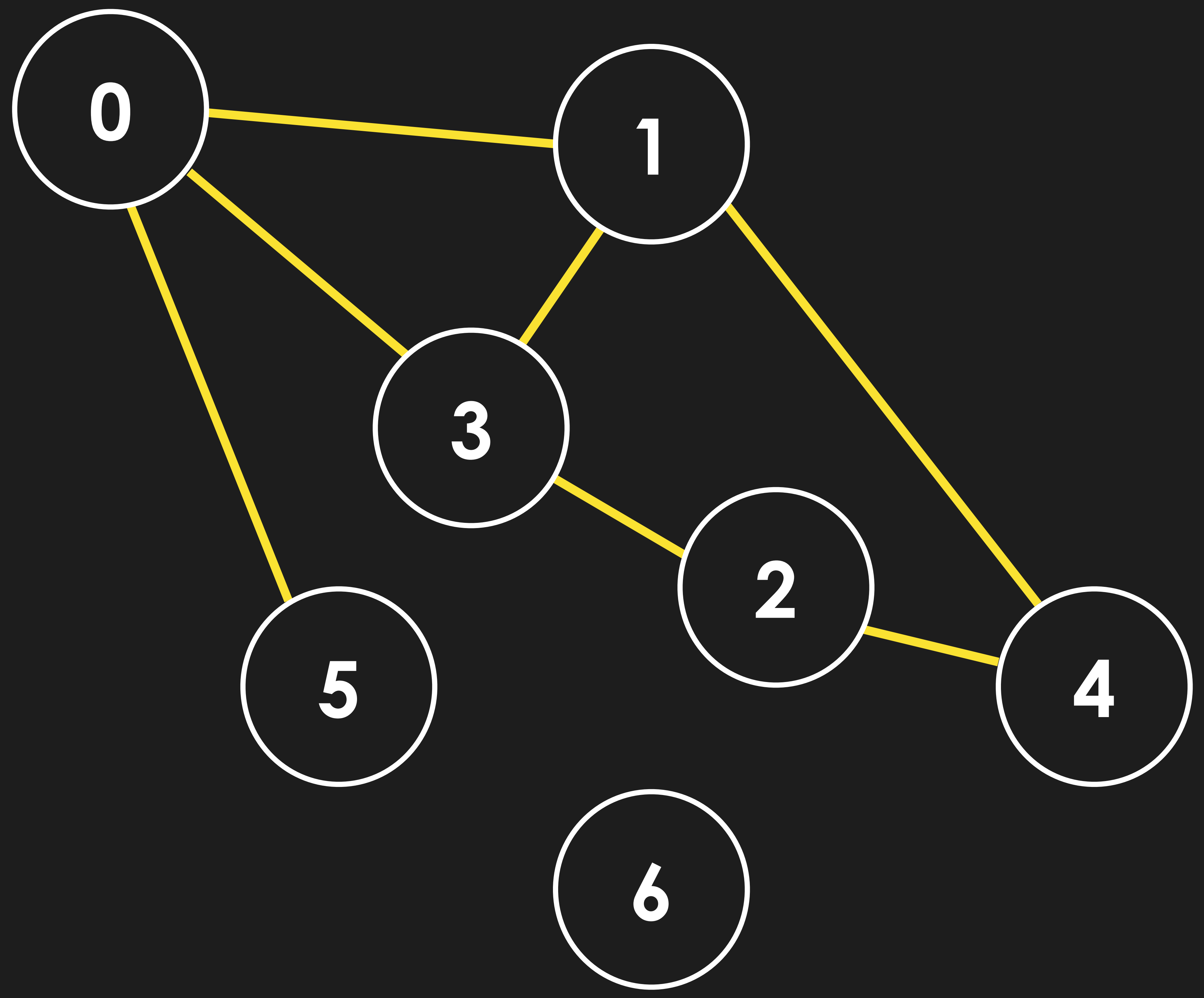
Adjacency Matrix

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



Adjacency Matrix

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



Adjacency Matrix

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

We can store the graph in a 2D list, where **adjMatrix[i][j]** is **True** if there is an edge between node i and j and **False** if not

Let **V** be the number of vertices and **E** be the number of edges

Let **V** be the number of vertices and **E** be the number of edges

An adjacency matrix, whilst efficient for checking if edges exist (constant time), takes up $\mathbf{V^2}$ space. This may be problematic for graphs with a huge number of nodes

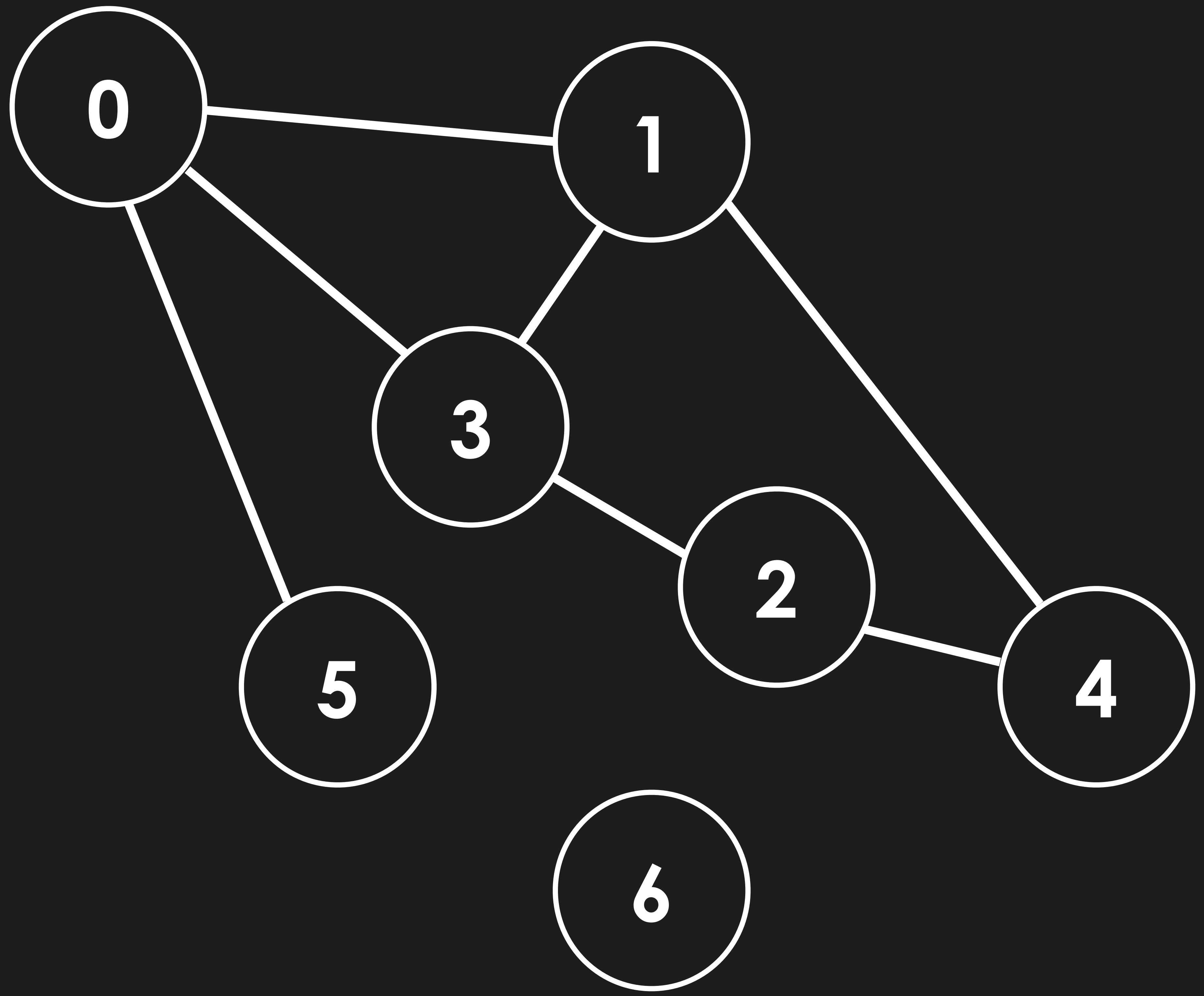
0	1	0	1	0	1	0
1	0	0	1	0	0	0
0	0	0	1	1	0	0
1	1	1	0	0	0	0
0	0	1	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	0

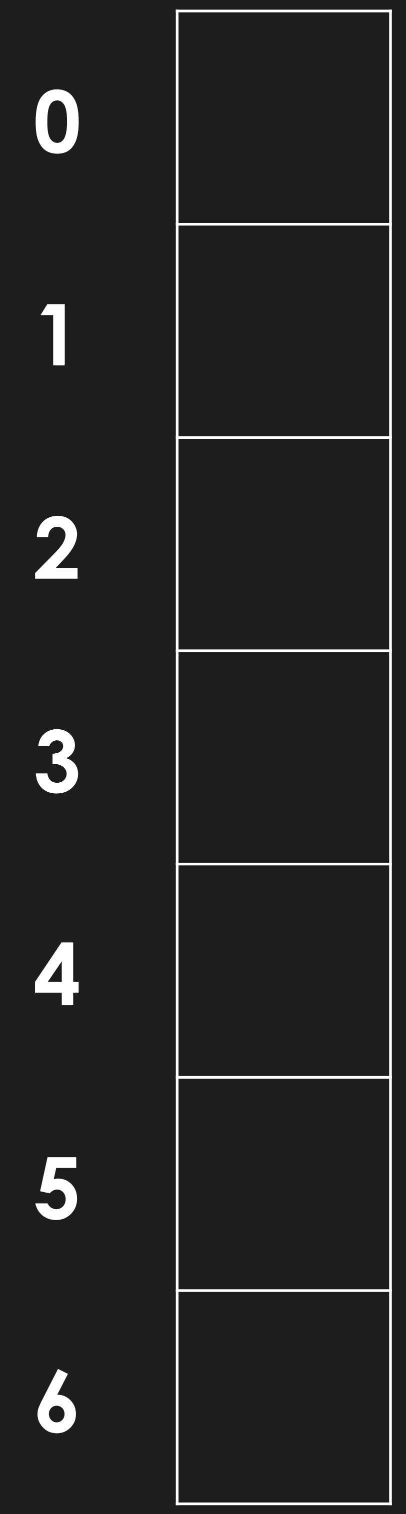
Additionally, note how many cells are empty. This is the problem with adjacency matrices. They are space inefficient

Adjacency matrices are useful when the number of edges are large

Adjacency List

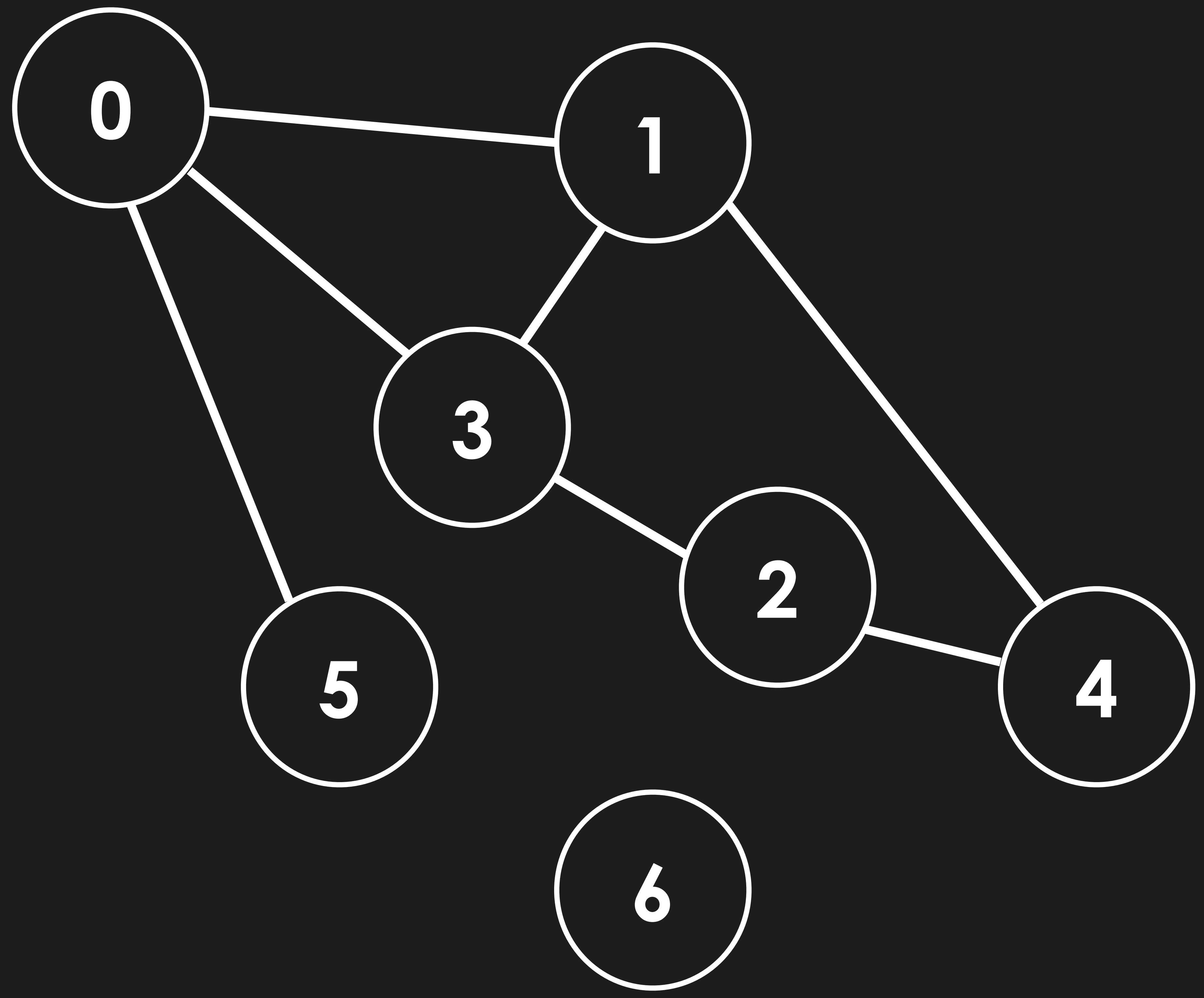
Adjacency List



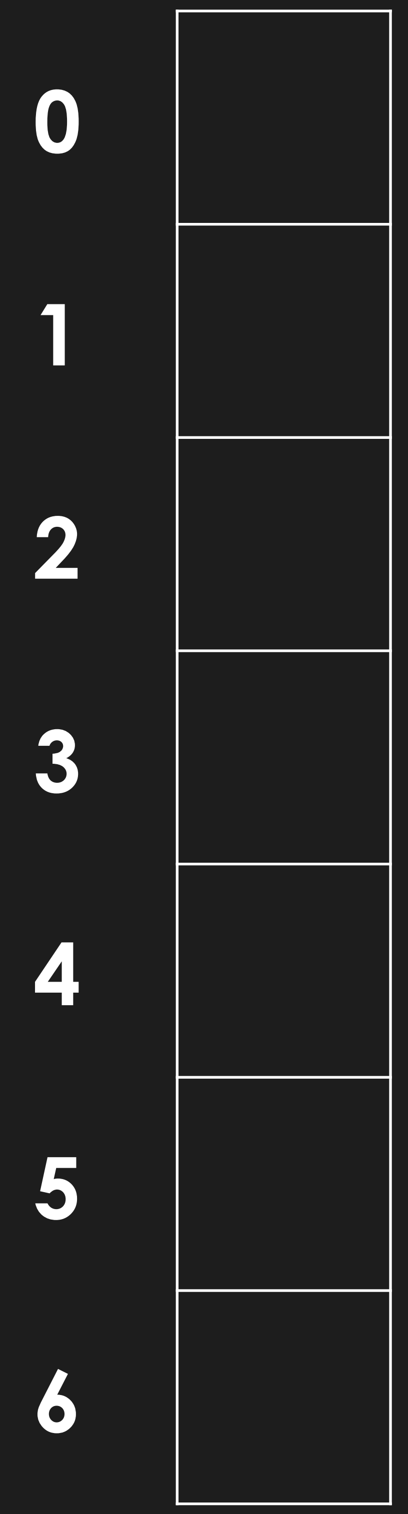


First, we initialise an empty linked list for each vertex

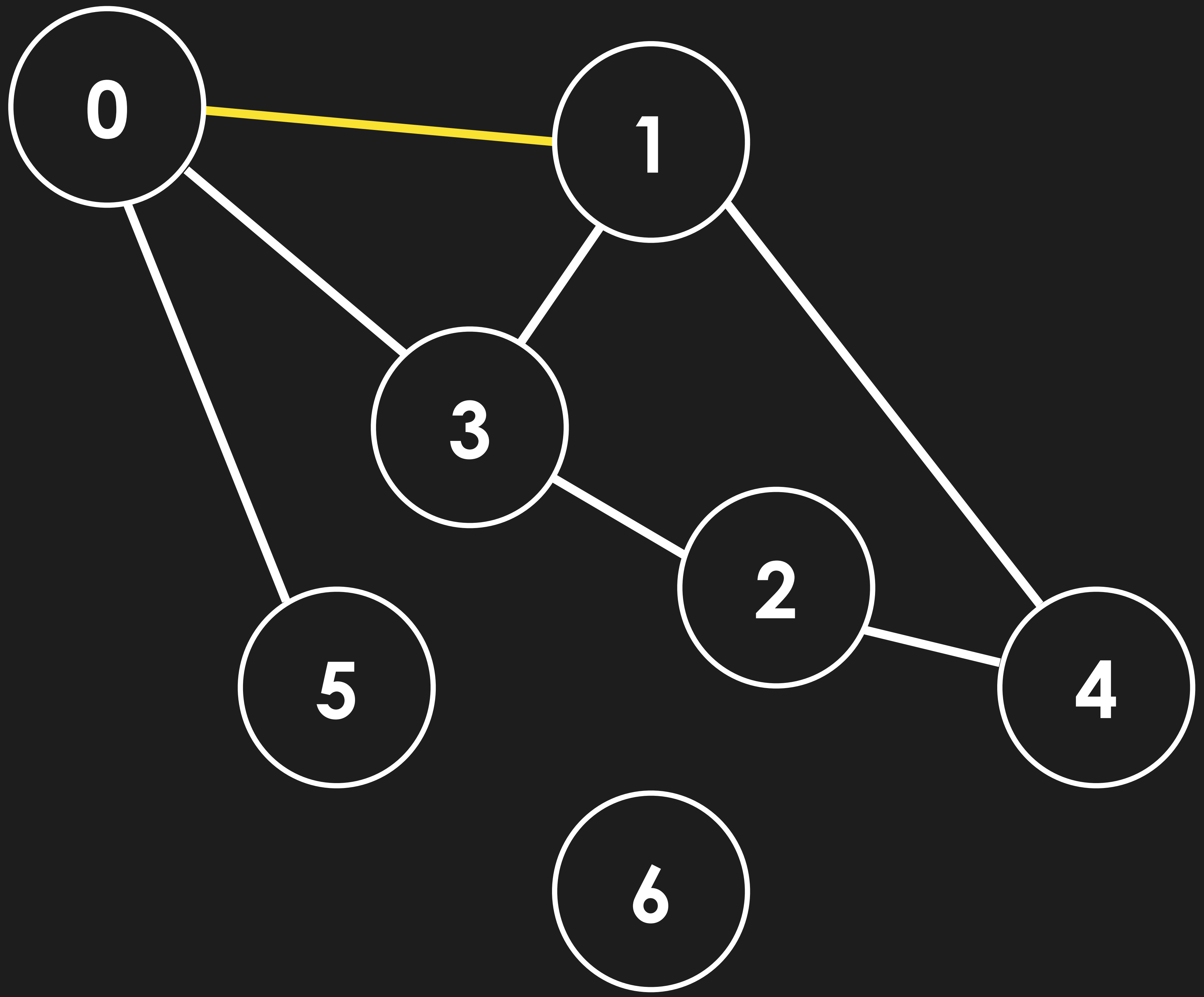
Adjacency List

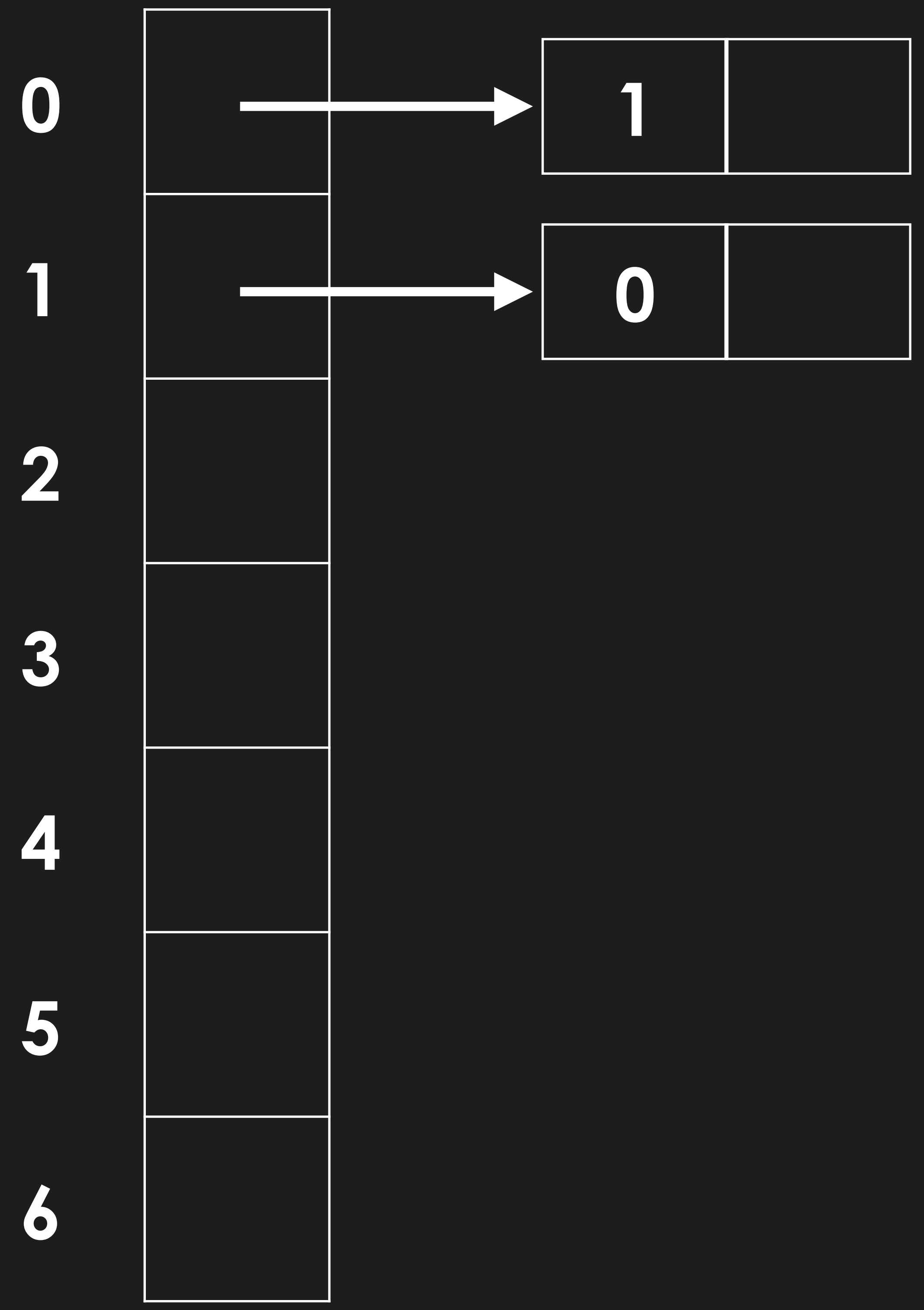


For each **edge**, we add the destination to the source vertex's linked list



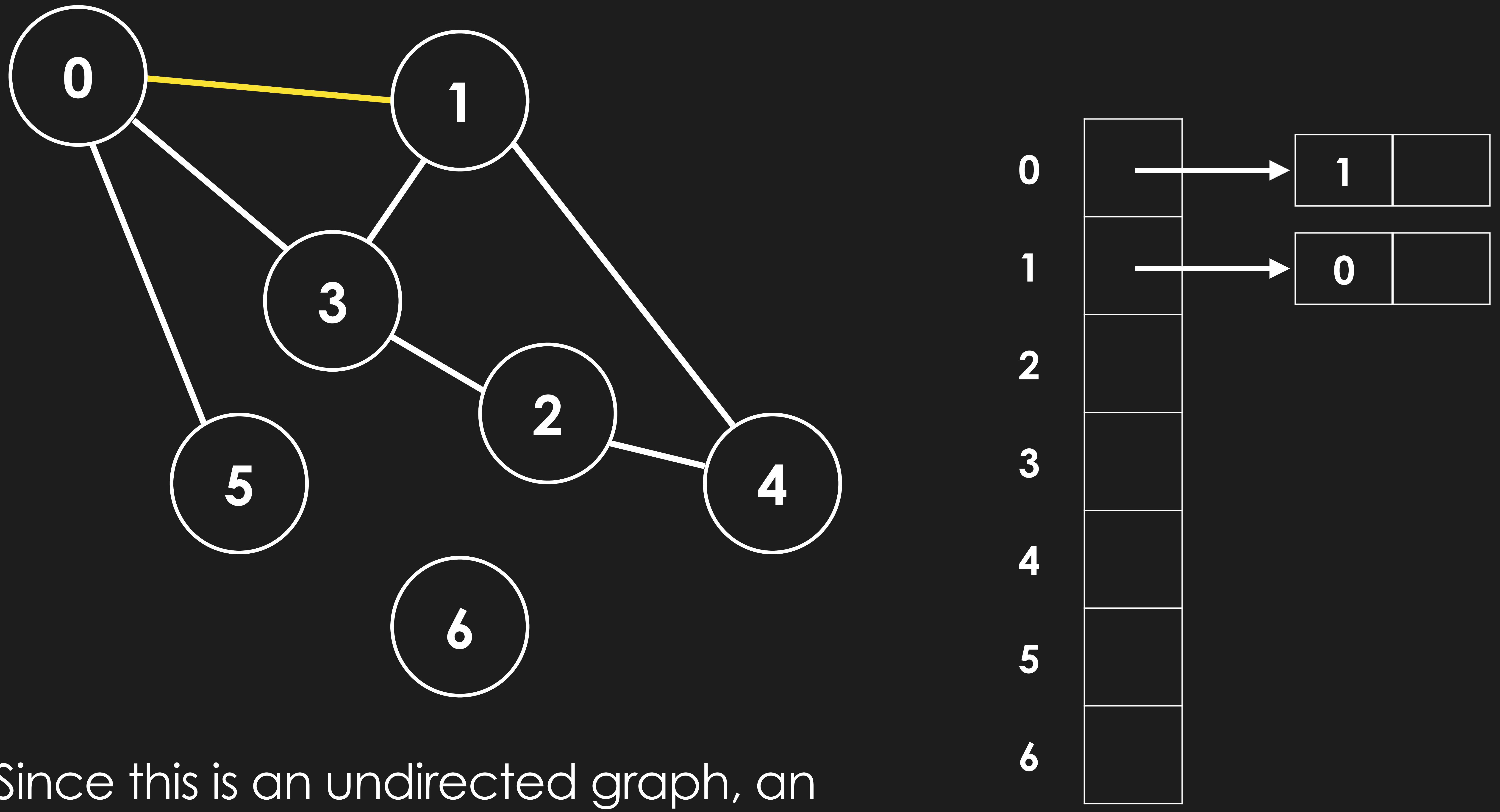
Adjacency List





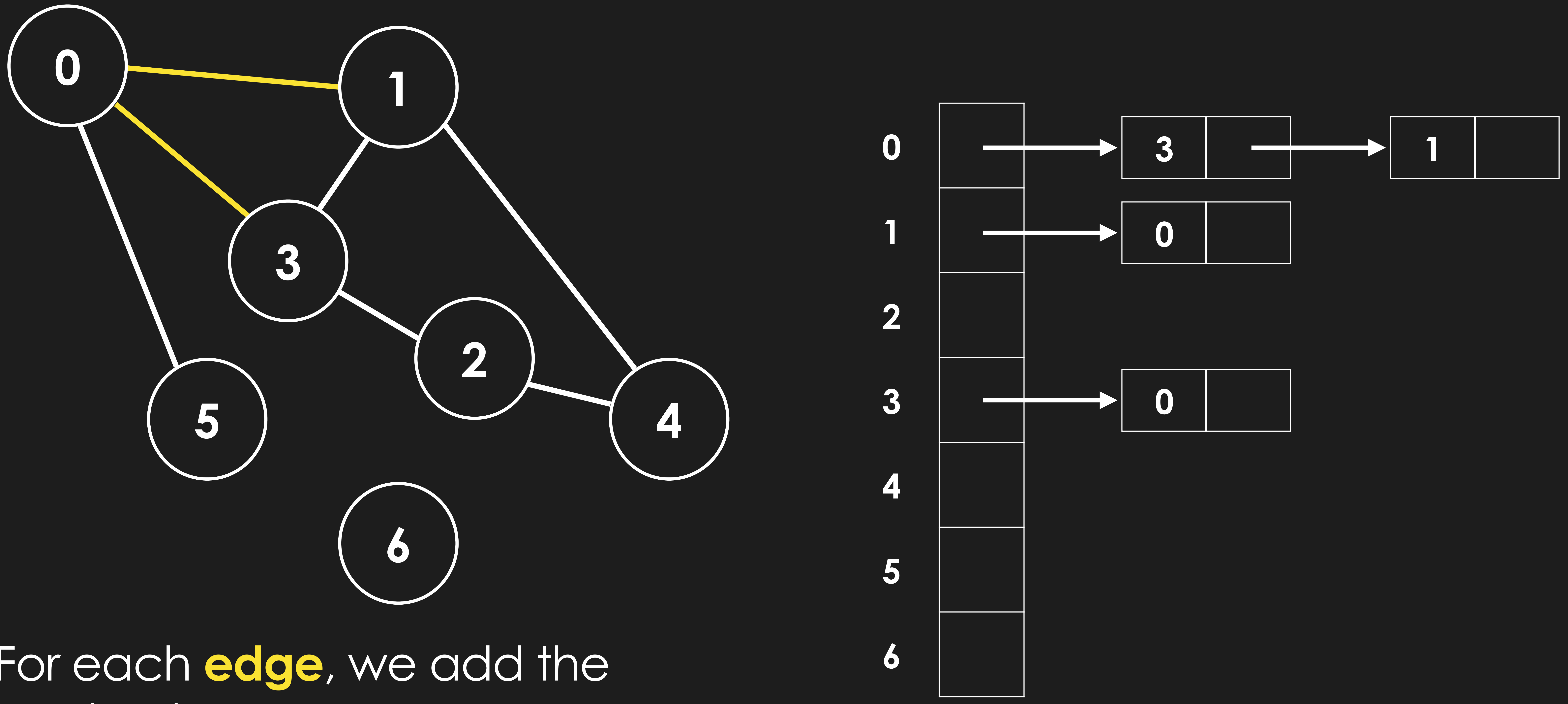
For each **edge**, we add the destination to the source vertex's linked list

Adjacency List



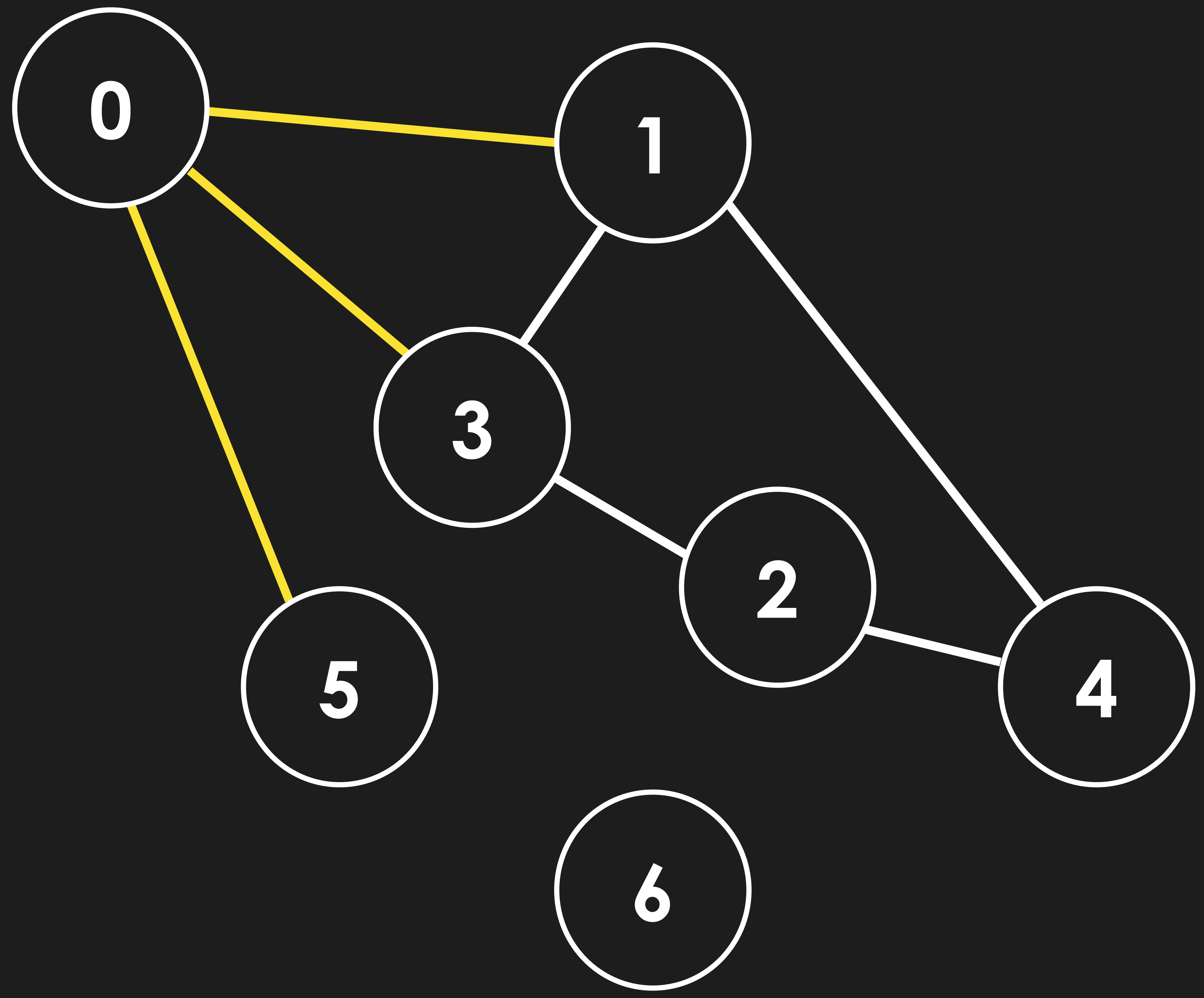
Since this is an undirected graph, an edge goes both ways, and thus we must add a node to both ends of the vertex linked list

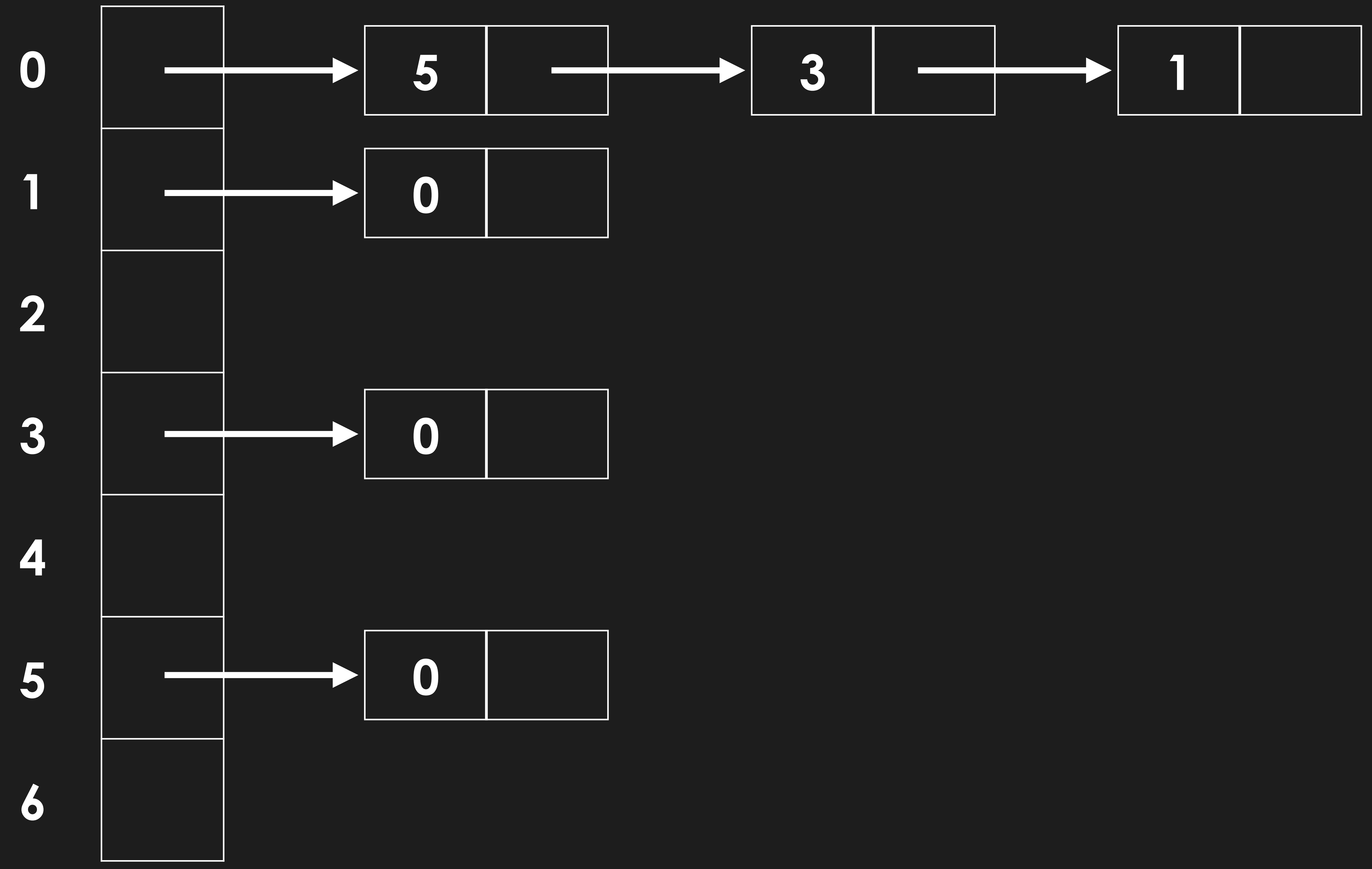
Adjacency List



For each **edge**, we add the destination to the source vertex's linked list

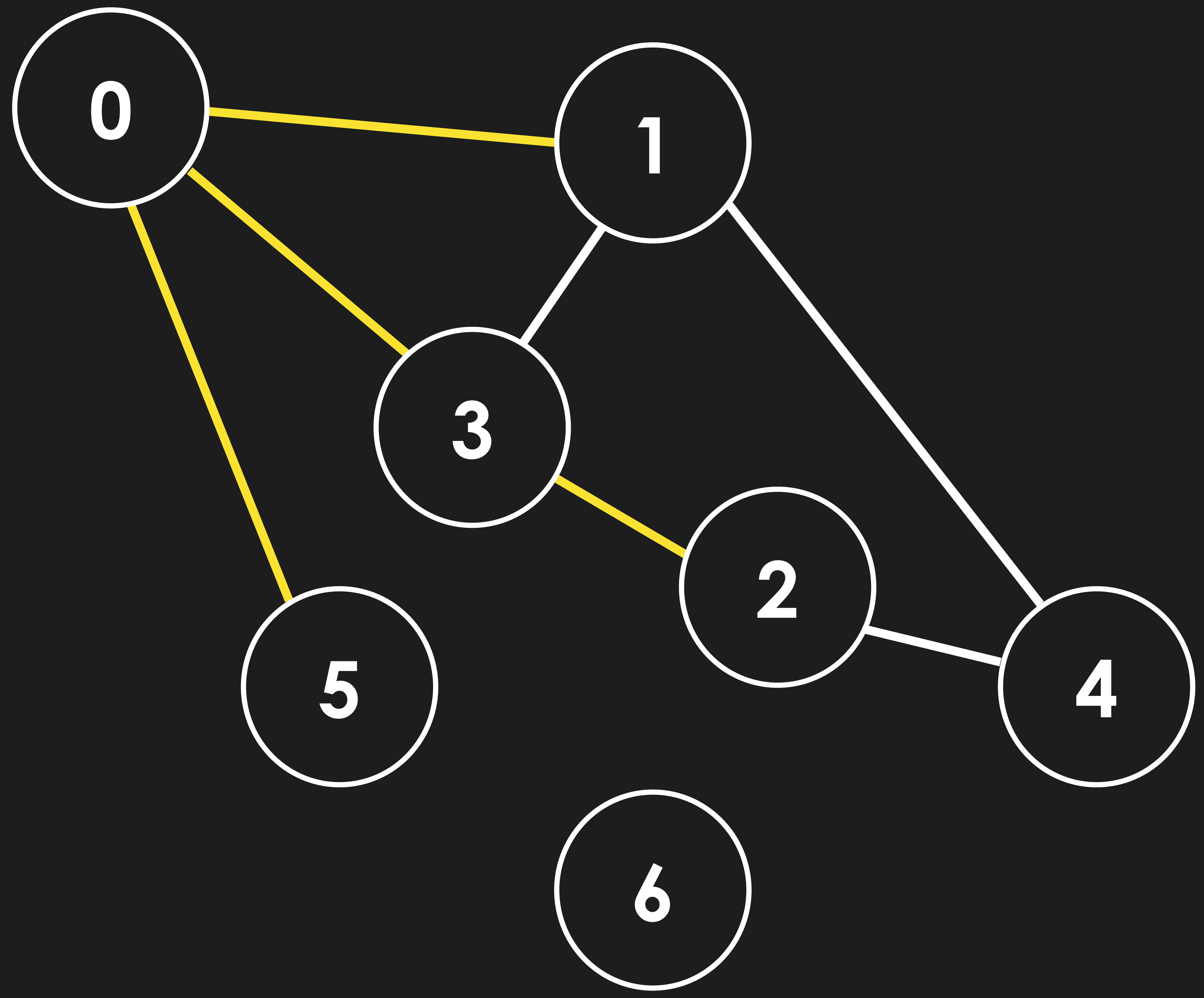
Adjacency List





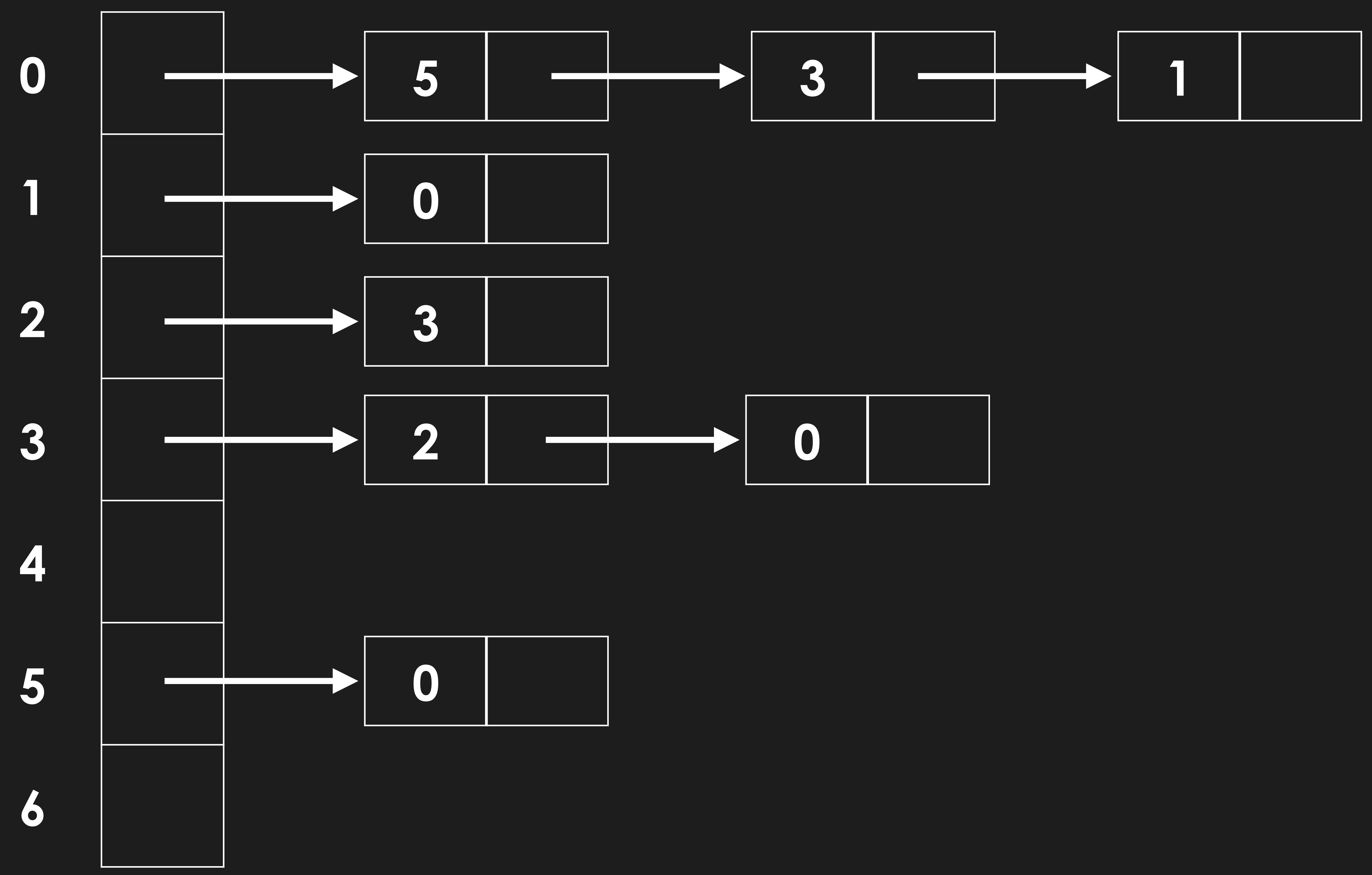
For each **edge**, we add the destination to the source vertex's linked list

CodeIT

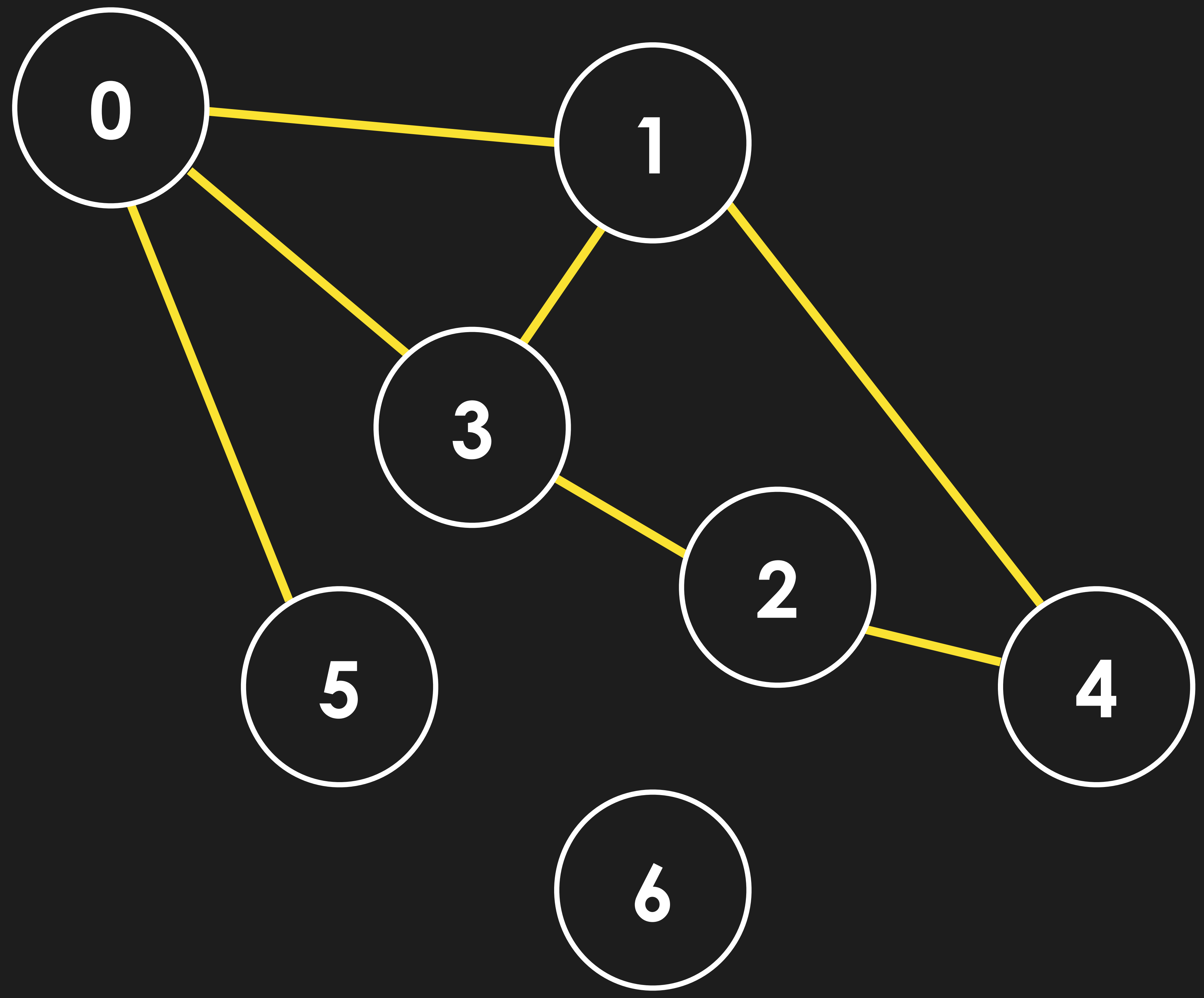


For each **edge**, we add the destination to the source vertex's linked list

Adjacency List

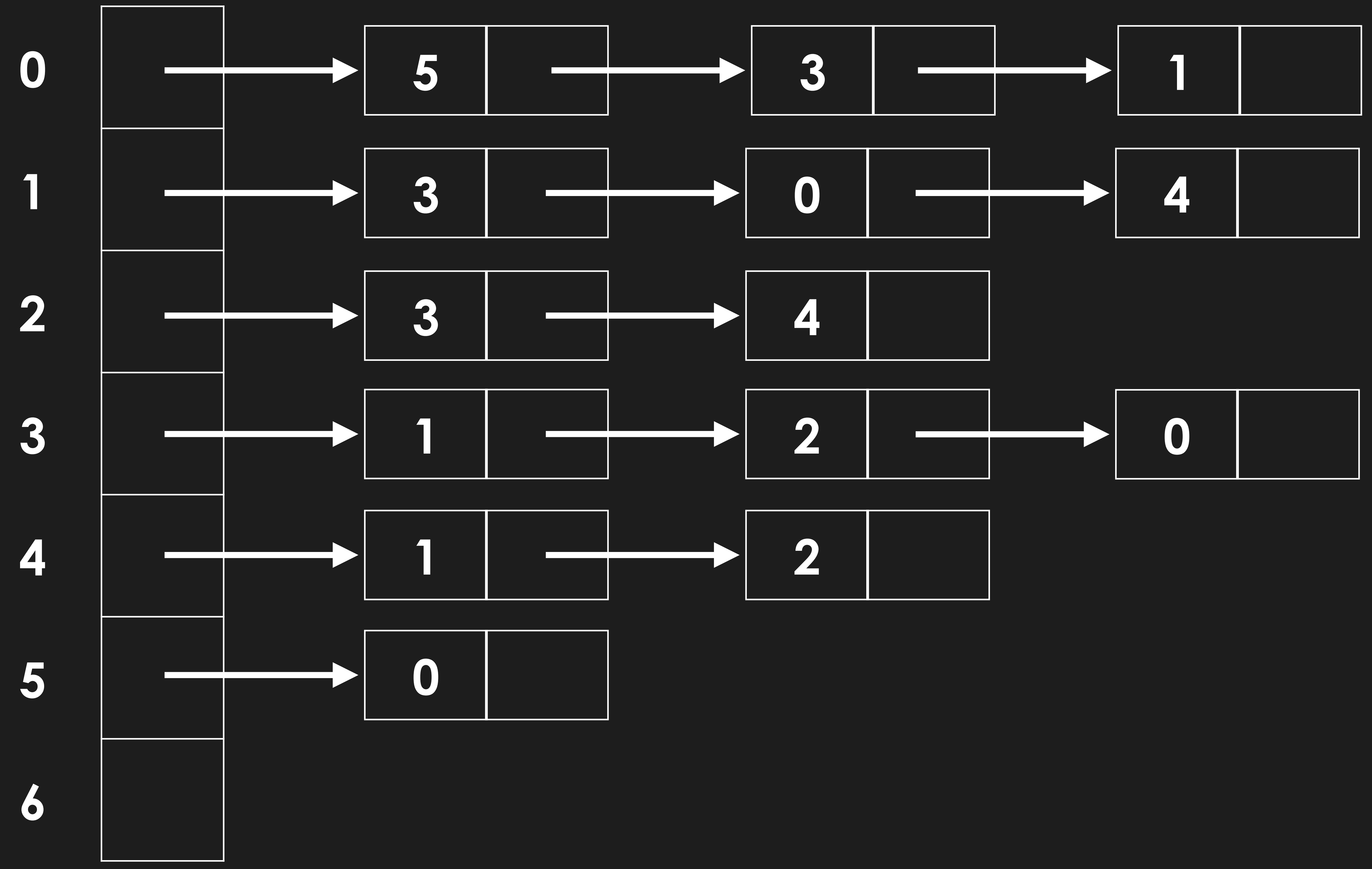


CodeIT



For each **edge**, we add the destination to the source vertex's linked list

Adjacency List



Adjacency List: Advantage

Adjacency List representation of graphs are useful when the number of edges are small. This is because the space taken is **V + E** (number of vertices + number of edges).

However, in the worst case scenario, if each vertex is connected to every other vertex, then space taken will be $\mathbf{V^2}$, the same as an adjacency matrix

Adjacency List: Disadvantage

Adjacency Lists have the added disadvantage of edge access. To check for an edge between two vertices, one has to traverse that vertex's linked list, which can in worst case has **V** complexity

Throughout the lessons on graphs, we will usually stick to adjacency list representation, but do remember the difference!

Implementation of Graphs

Graphs

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

    def addEdge(self, edge):
        src, dest = edge

        self.adjList[src].append(dest)
        self.adjList[dest].append(src)
```

Graphs

```
def printGraph(self):
    for i in range(len(self.adjList)):
        print("vertex {}".format(i), end='')
        for dest in self.adjList[i]:
            print(" -> {}".format(dest), end="")

        print()
```

Graphs

```python
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 3), (1, 4), (3, 2), (2, 4)]
for edge in edges:
    graph.addEdge(edge)

graph.printGraph()
```

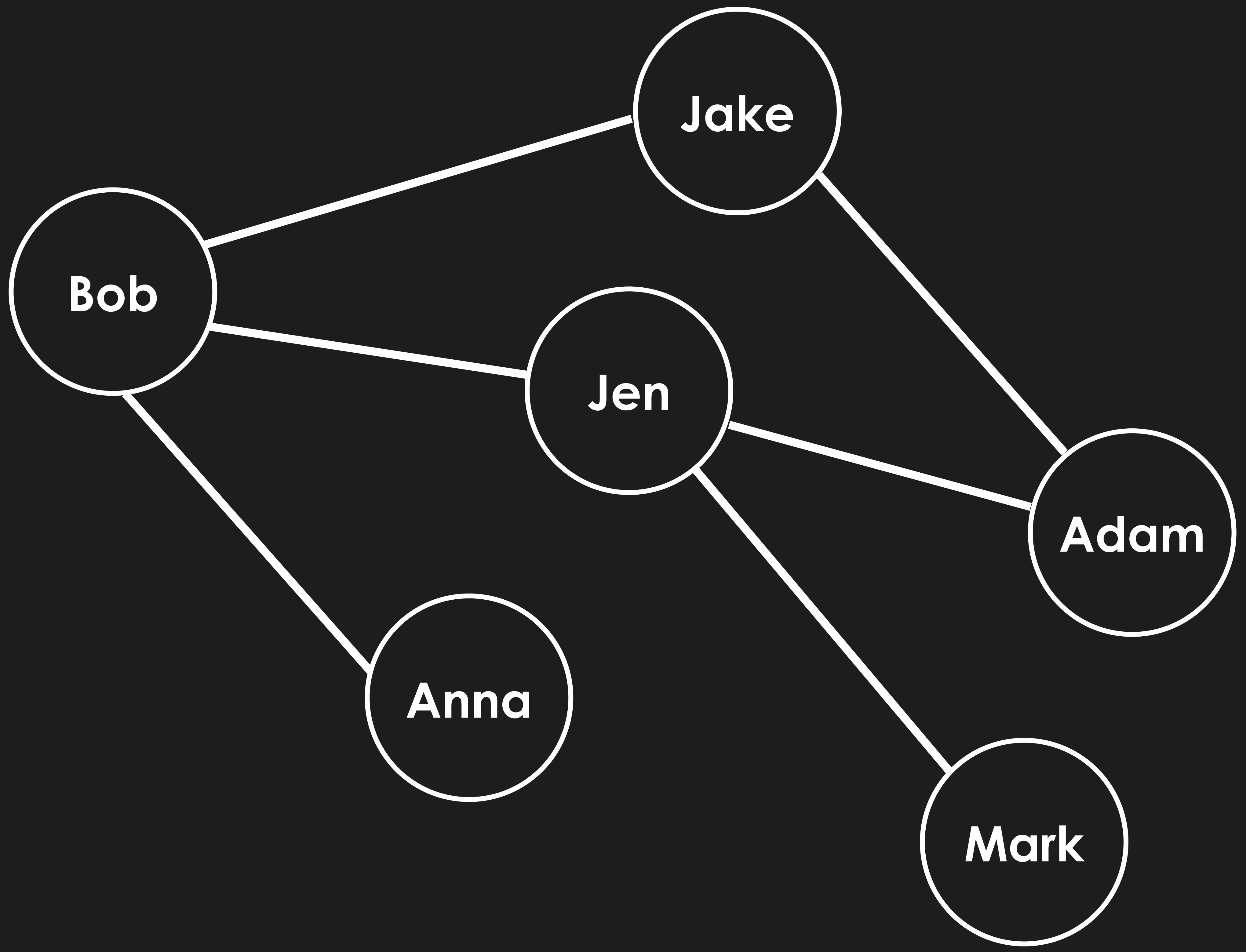
Graphs

```
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 3), (1, 4), (3, 2), (2, 4)]
for edge in edges:
    graph.addEdge(edge)

graph.printGraph()
```

```
vertex 0 -> 1 -> 3 -> 5
vertex 1 -> 0 -> 3 -> 4
vertex 2 -> 3 -> 4
vertex 3 -> 0 -> 1 -> 2
vertex 4 -> 1 -> 2
vertex 5 -> 0
vertex 6
```

CodeIT

SIDEBAR: What if our graph wasn't made up of integer vertices:



How do we represent this?

Graph Traversals

Graph Traversals

To solve graph problems (e.g. shortest path from node A to B), we need **traversal methods**

Traversal is the act of **visiting edges and vertices** in a graph

Most **graph problems** rely on traversals

Depth First Search

Depth First Search

At a high level, you **follow some random path** until you reach a block

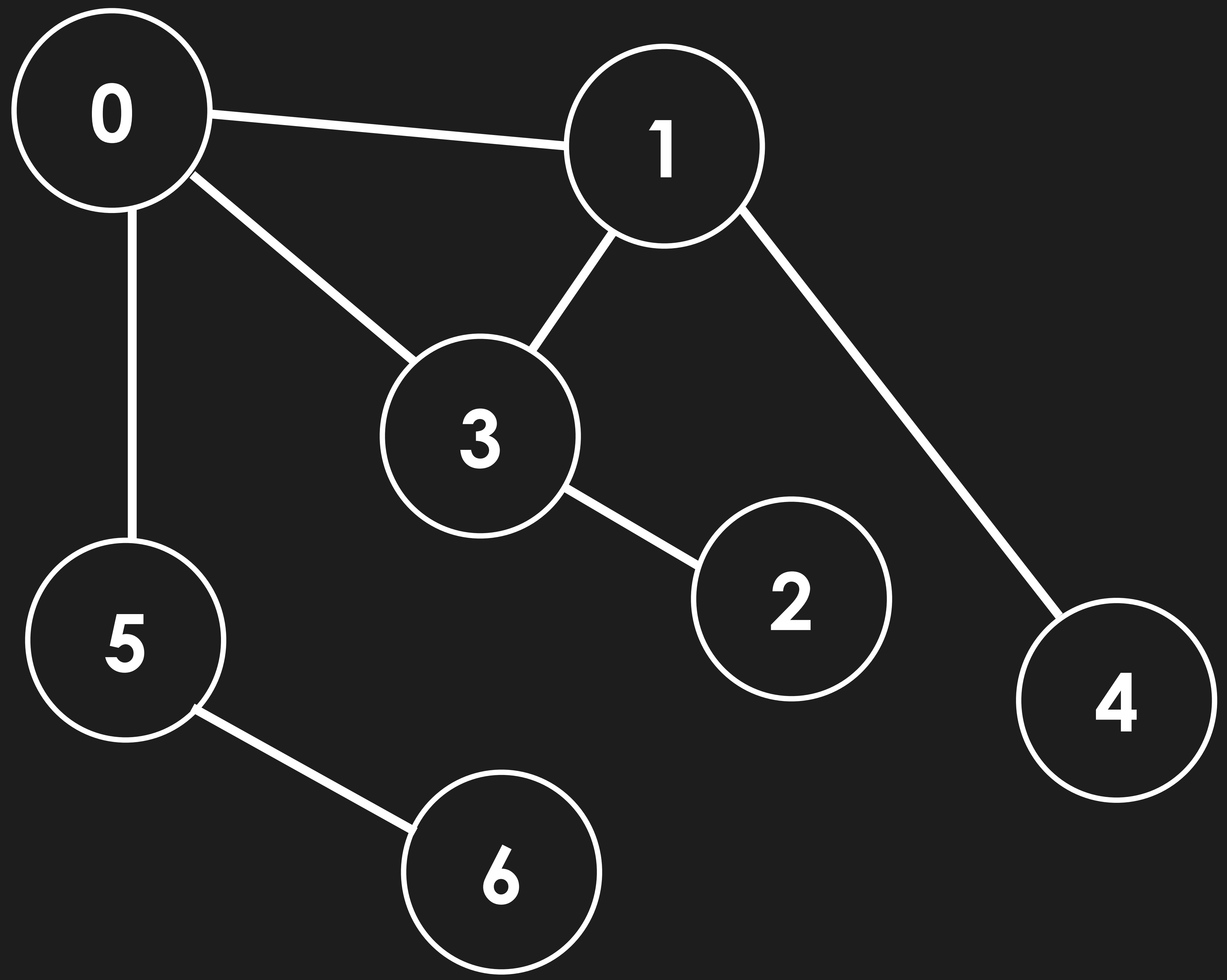
After that, **retrace your steps** to the earliest vertex with a next path

Repeat on that path

dfs

```
def dfs(graph, start)
```

dfs(graph, 0)

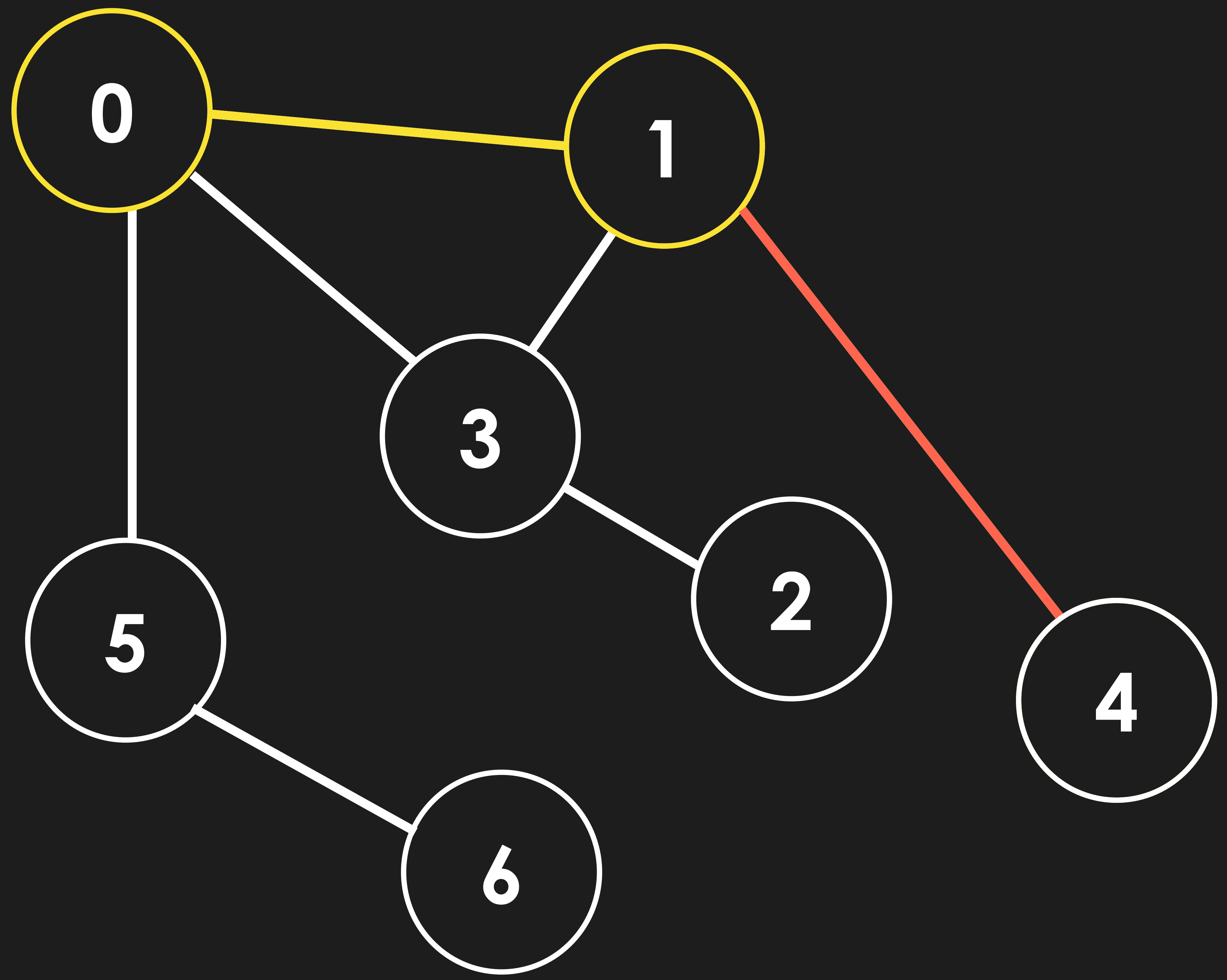


visited

0	
1	
2	
3	
4	
5	
6	

dfs(graph, 0)

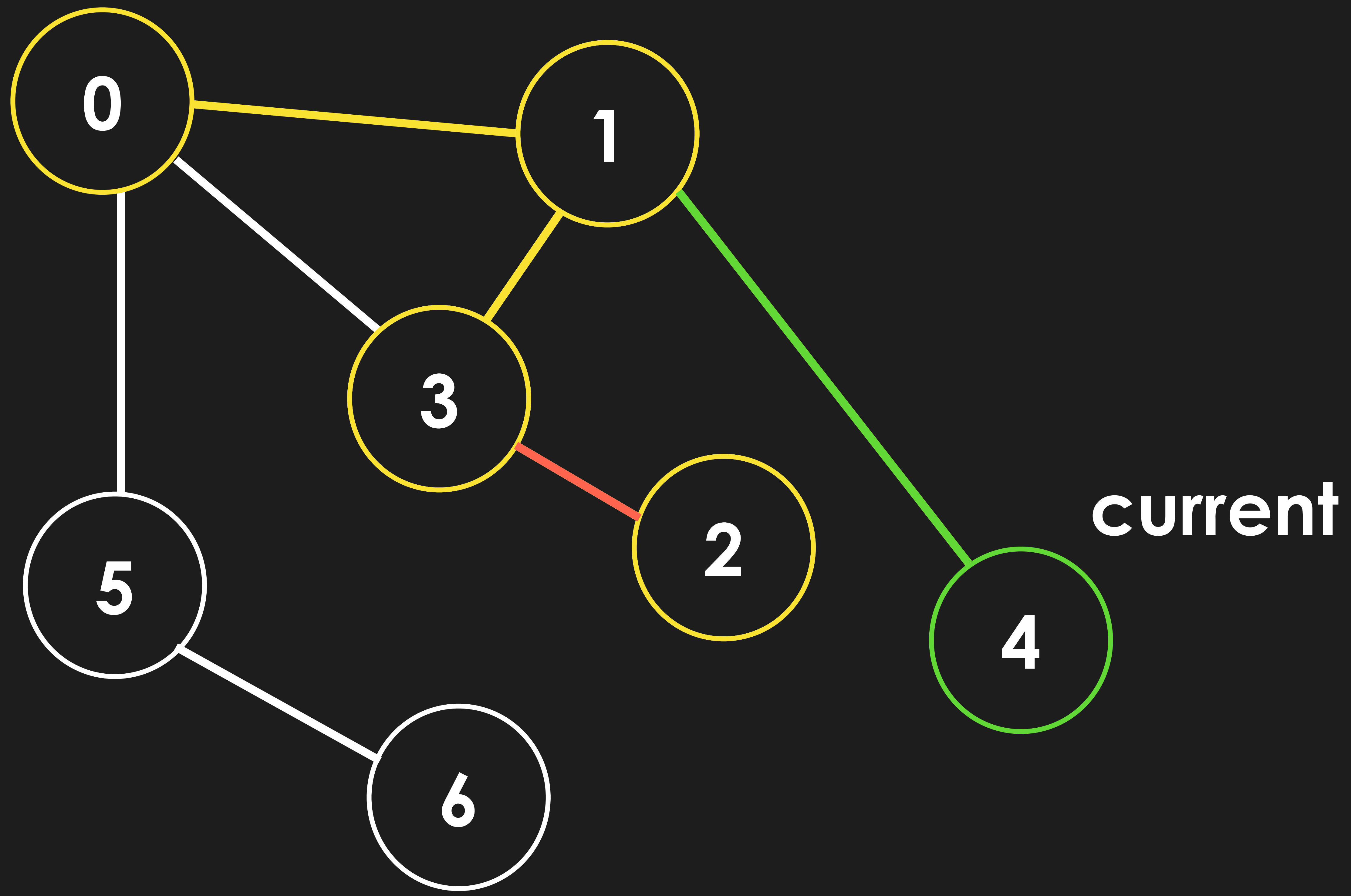
current



visited

0	True
1	True
2	
3	
4	True
5	
6	

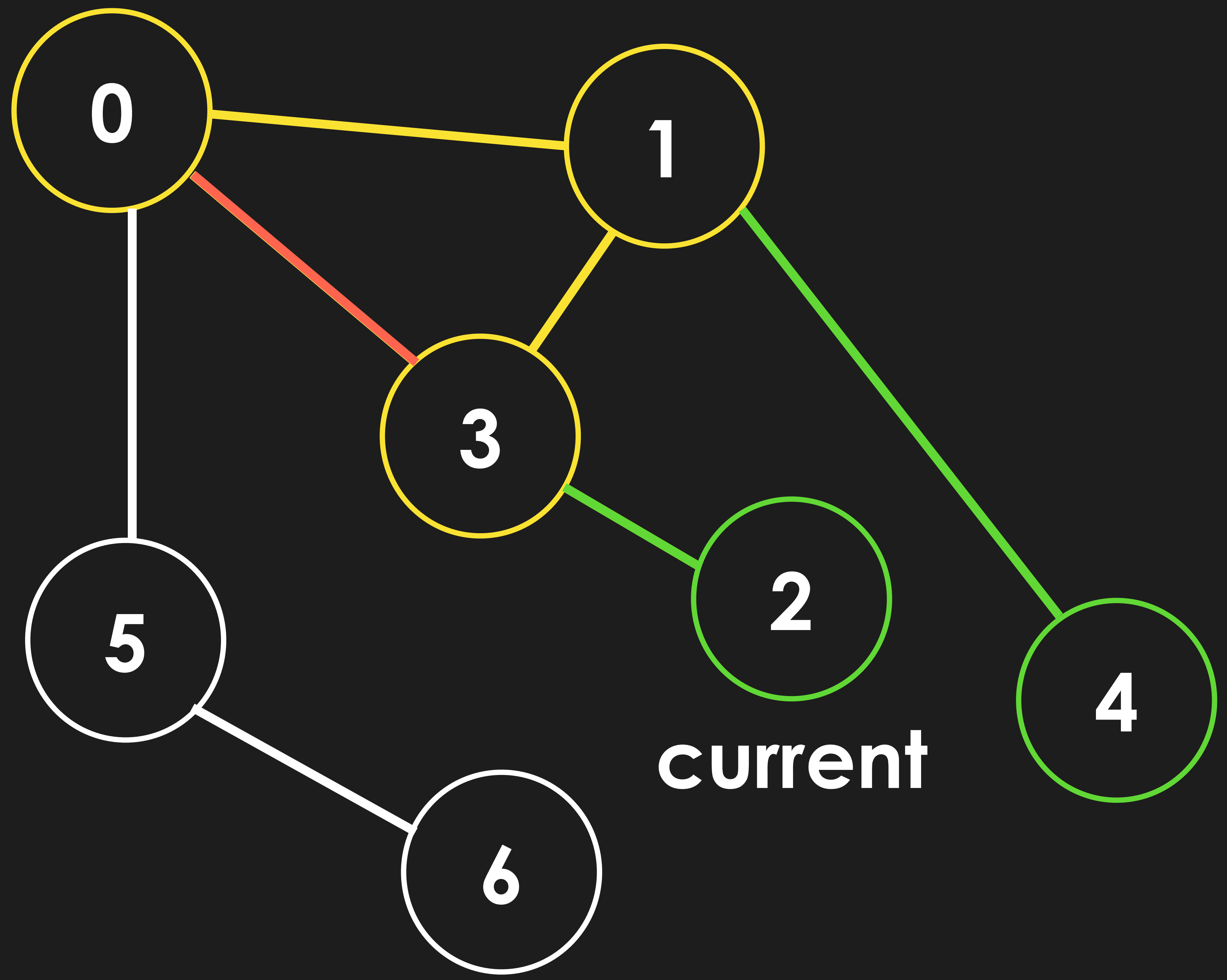
dfs(graph, 0)



visited

0	True
1	True
2	True
3	True
4	True
5	
6	

dfs(graph, 0)

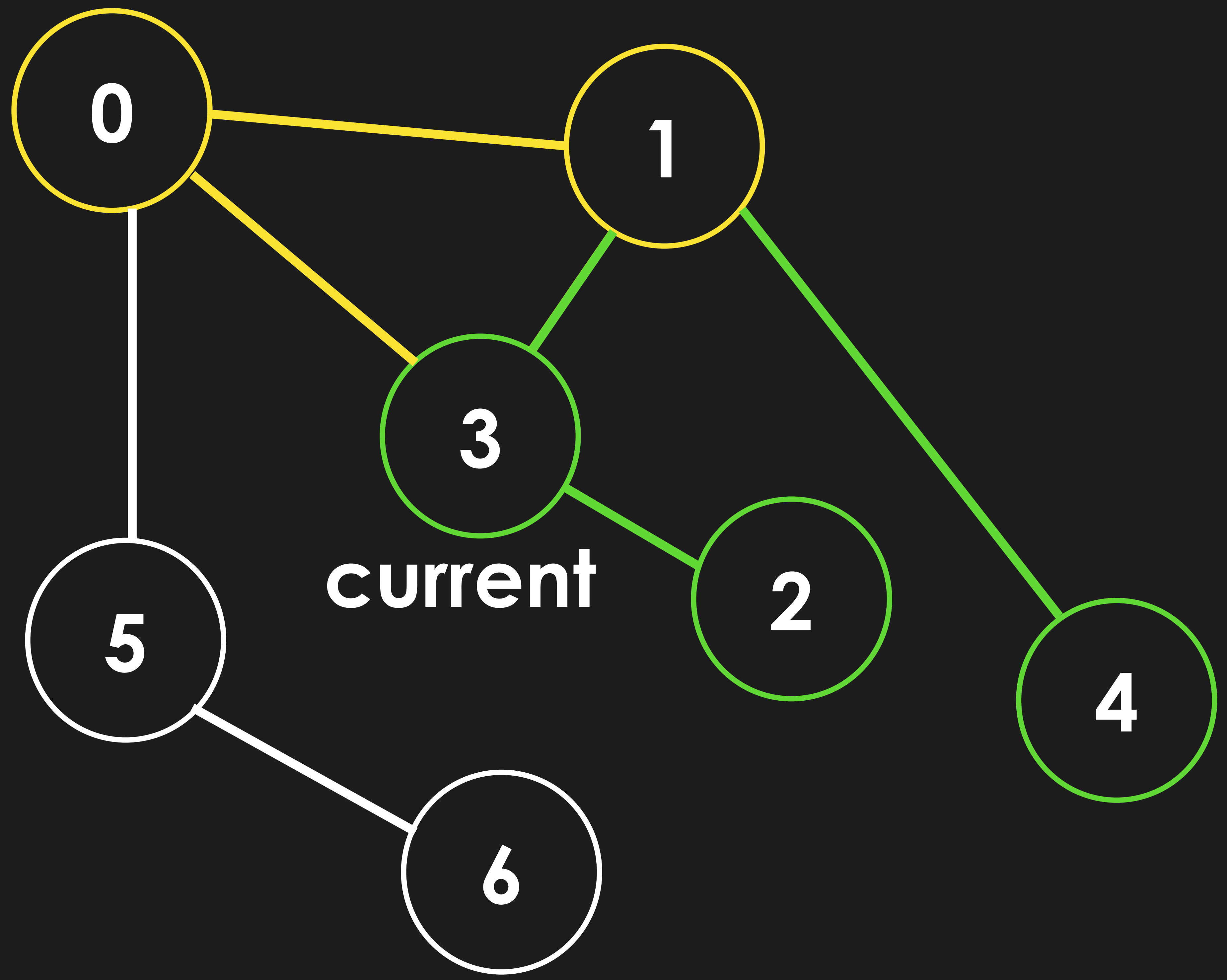


visited

0	True
1	True
2	True
3	True
4	True
5	
6	

CodeIT

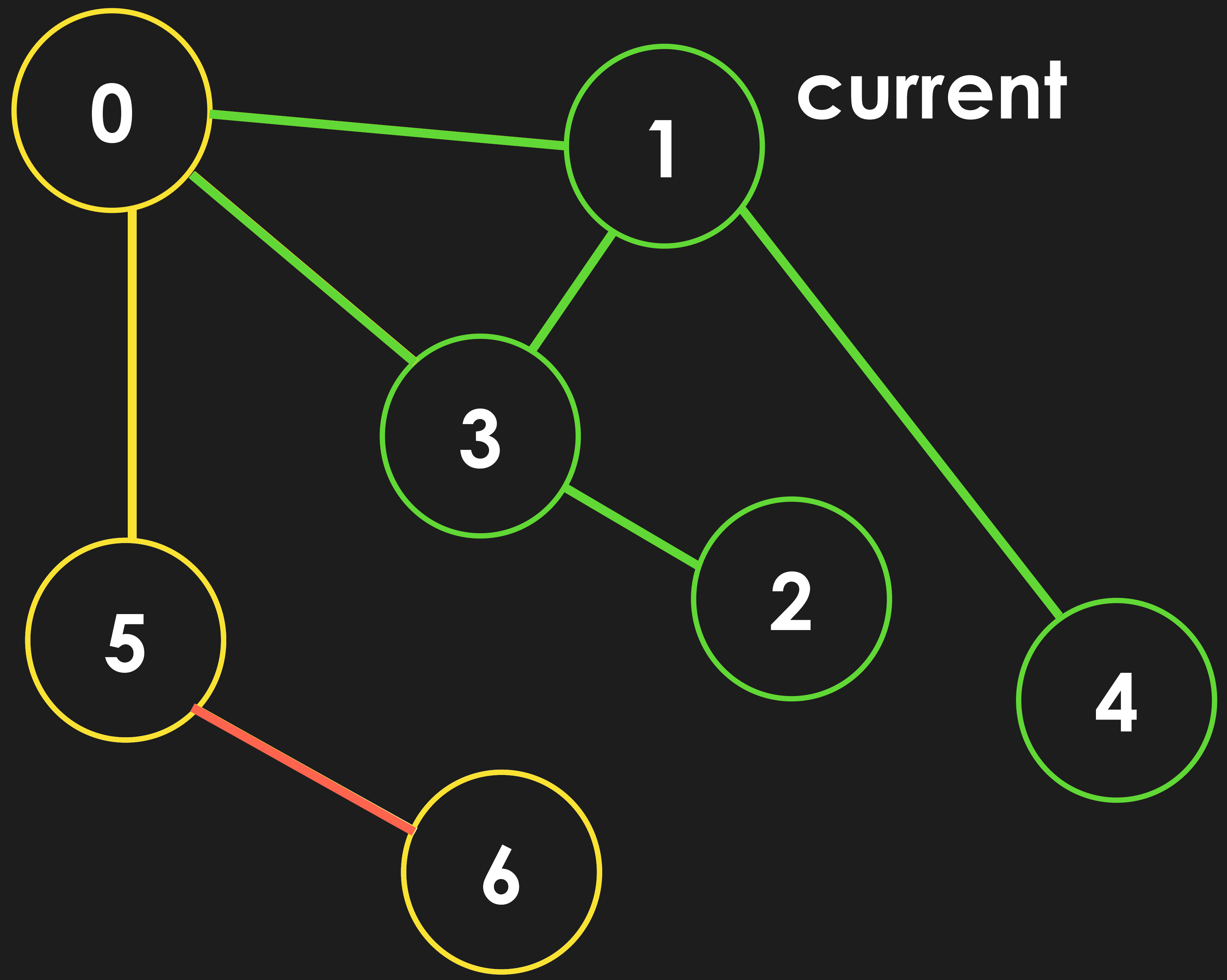
dfs(graph, 0)



visited

0	True
1	True
2	True
3	True
4	True
5	
6	

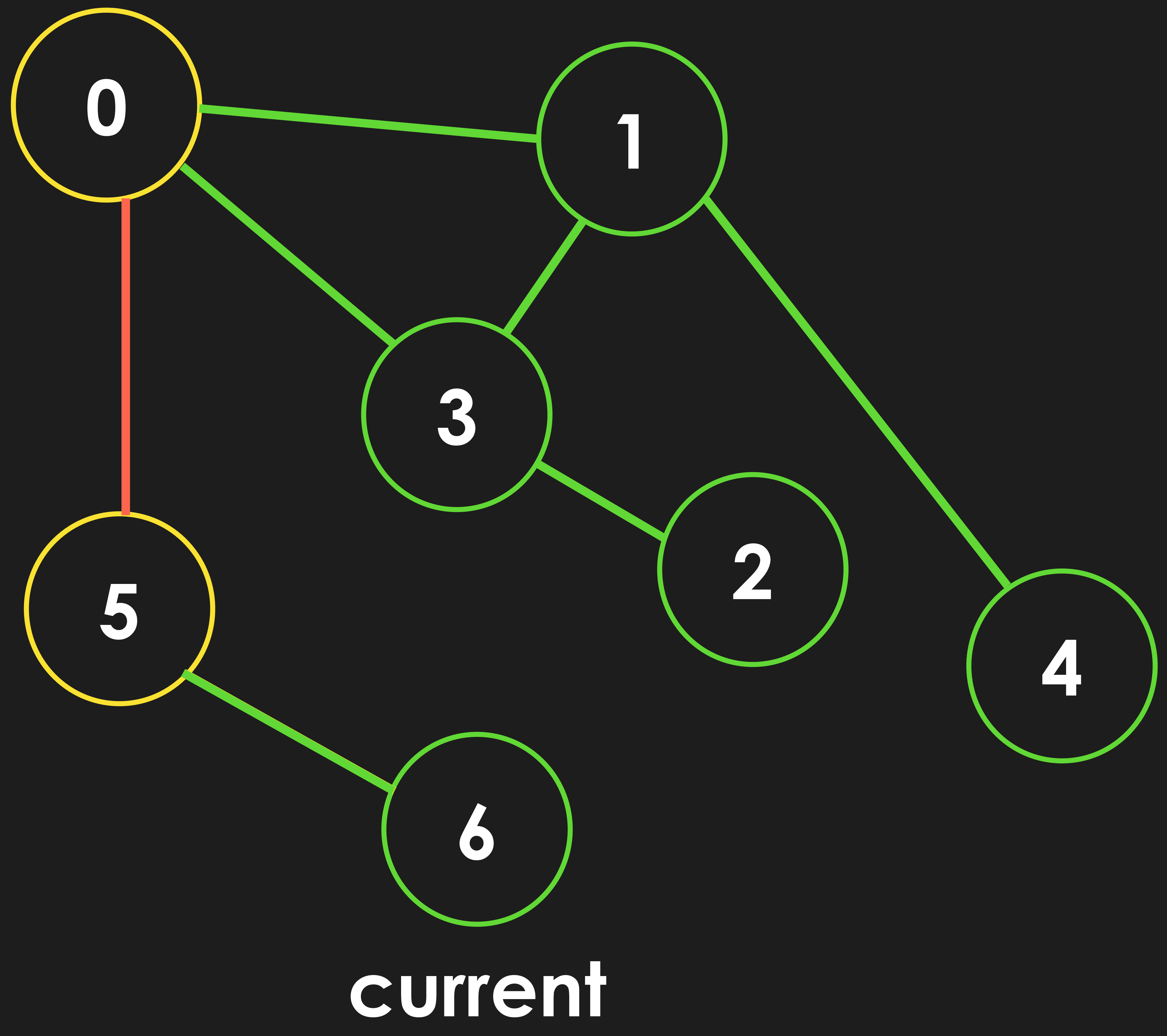
dfs(graph, 0)



visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

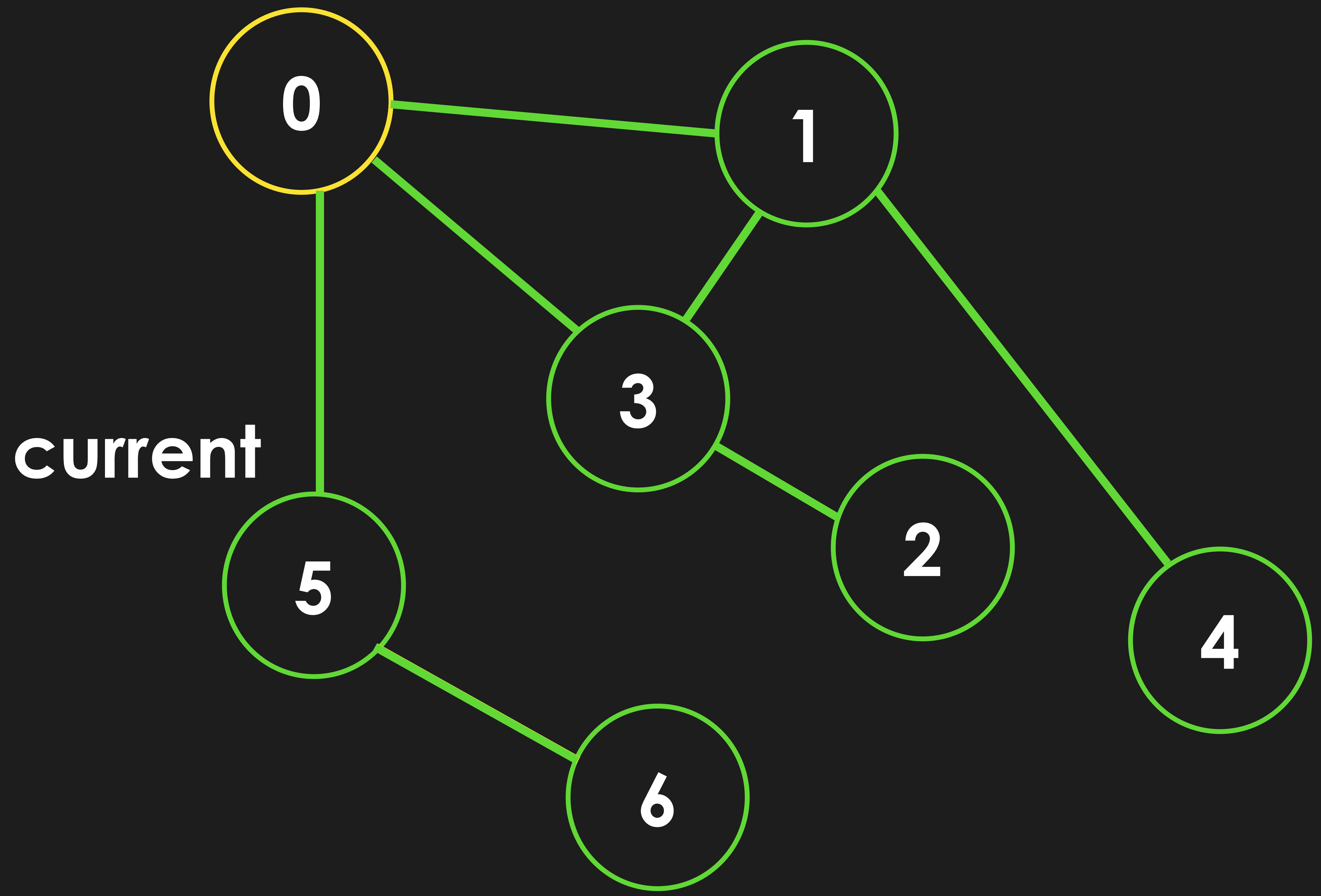
dfs(graph, 0)



visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

dfs(graph, 0)

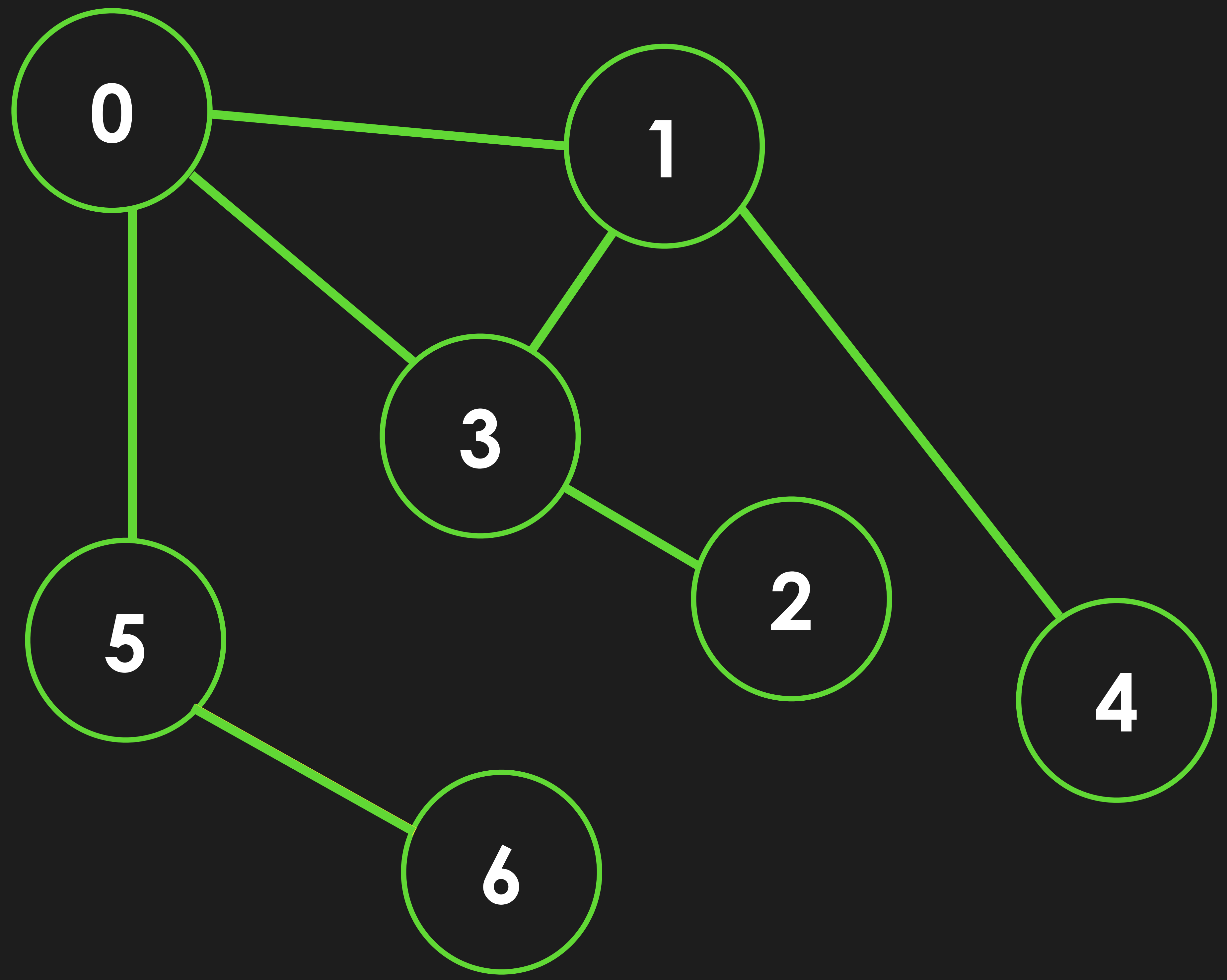


visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

dfs(graph, 0)

current



visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

Implementation of DFS

dfs

```python
def dfsRecurse(graph, v, visited):
    visited[v] = True
    print("visiting vertex {}".format(v))

    for dest in graph.adjList[v]:
        if not visited[dest]:
            dfsRecurse(graph, dest, visited)
    return


def dfs(graph, start):
    visited = [False] * len(graph.adjList)
    dfsRecurse(graph, start, visited)
```

dfs demo

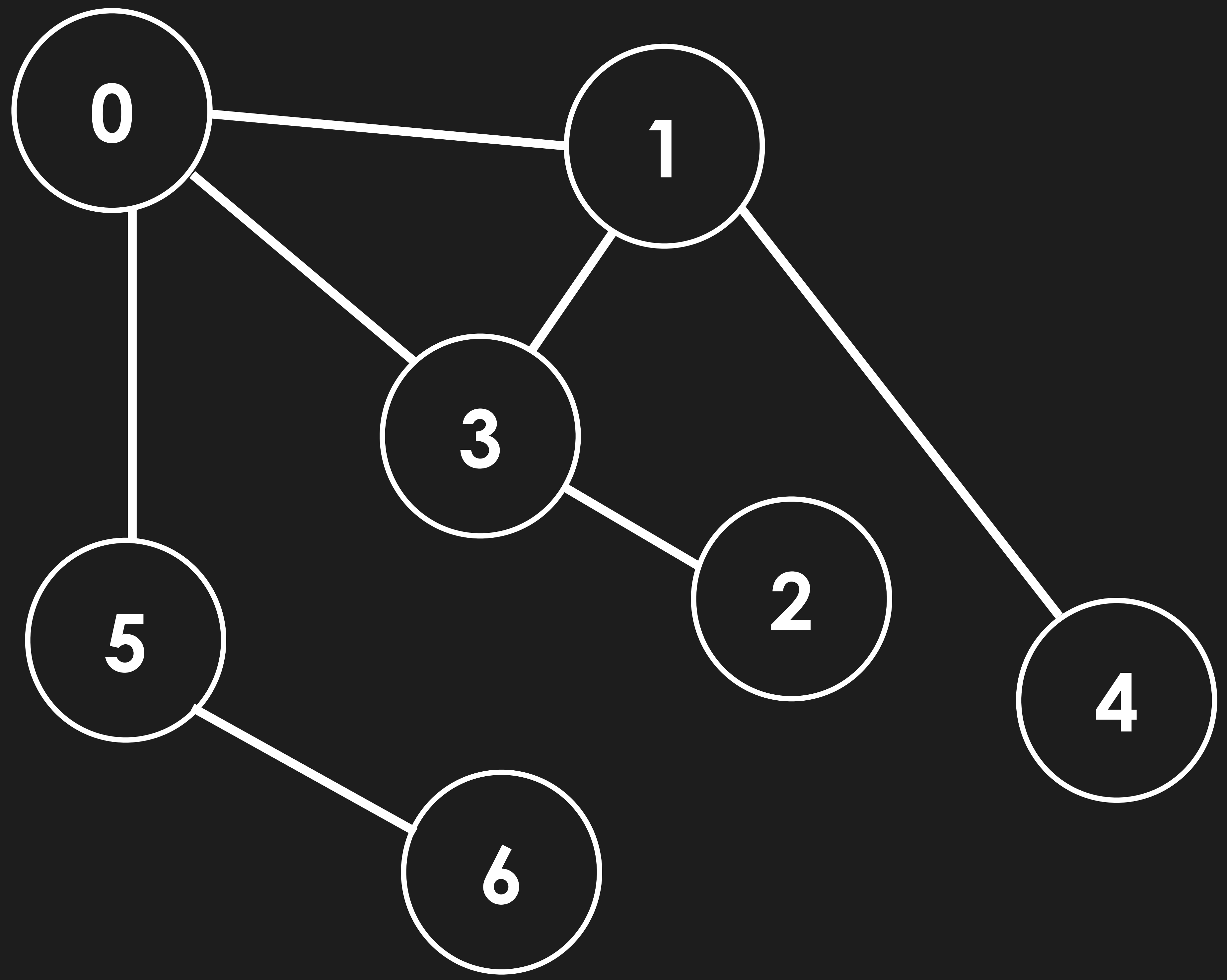
```python
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 4), (1, 3), (3, 2), (5, 6)]
for edge in edges:
    graph.addEdge(edge)

dfs(graph, 0)
```

dfs demo

```
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 4), (1, 3), (3, 2), (5, 6)]
for edge in edges:
    graph.addEdge(edge)

dfs(graph, 0)
```

```
visiting vertex 0
visiting vertex 1
visiting vertex 4
visiting vertex 3
visiting vertex 2
visiting vertex 5
visiting vertex 6
```

dfs demo



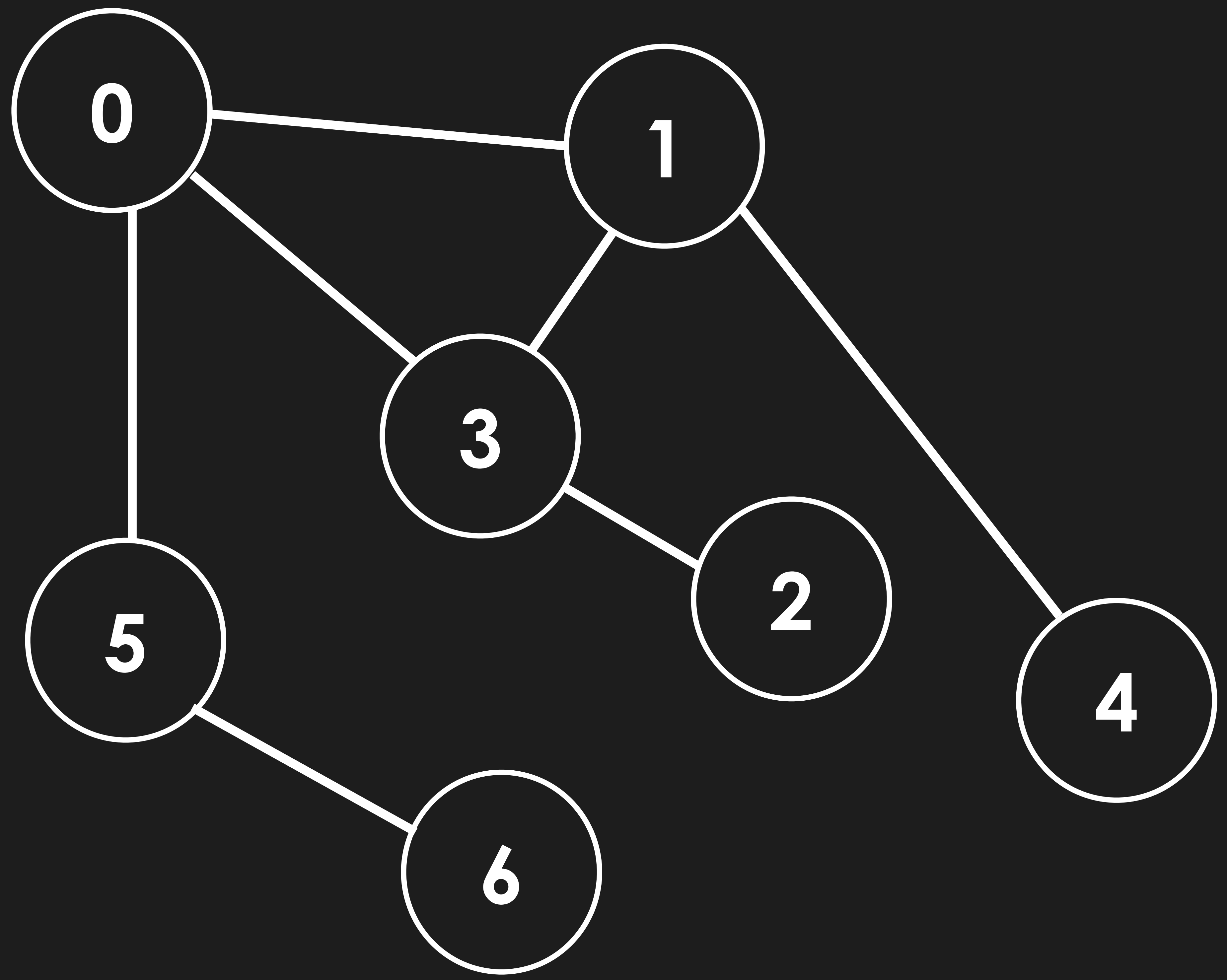

```
visiting vertex 0
visiting vertex 1
visiting vertex 4
visiting vertex 3
visiting vertex 2
visiting vertex 5
visiting vertex 6
```

Application of DFS

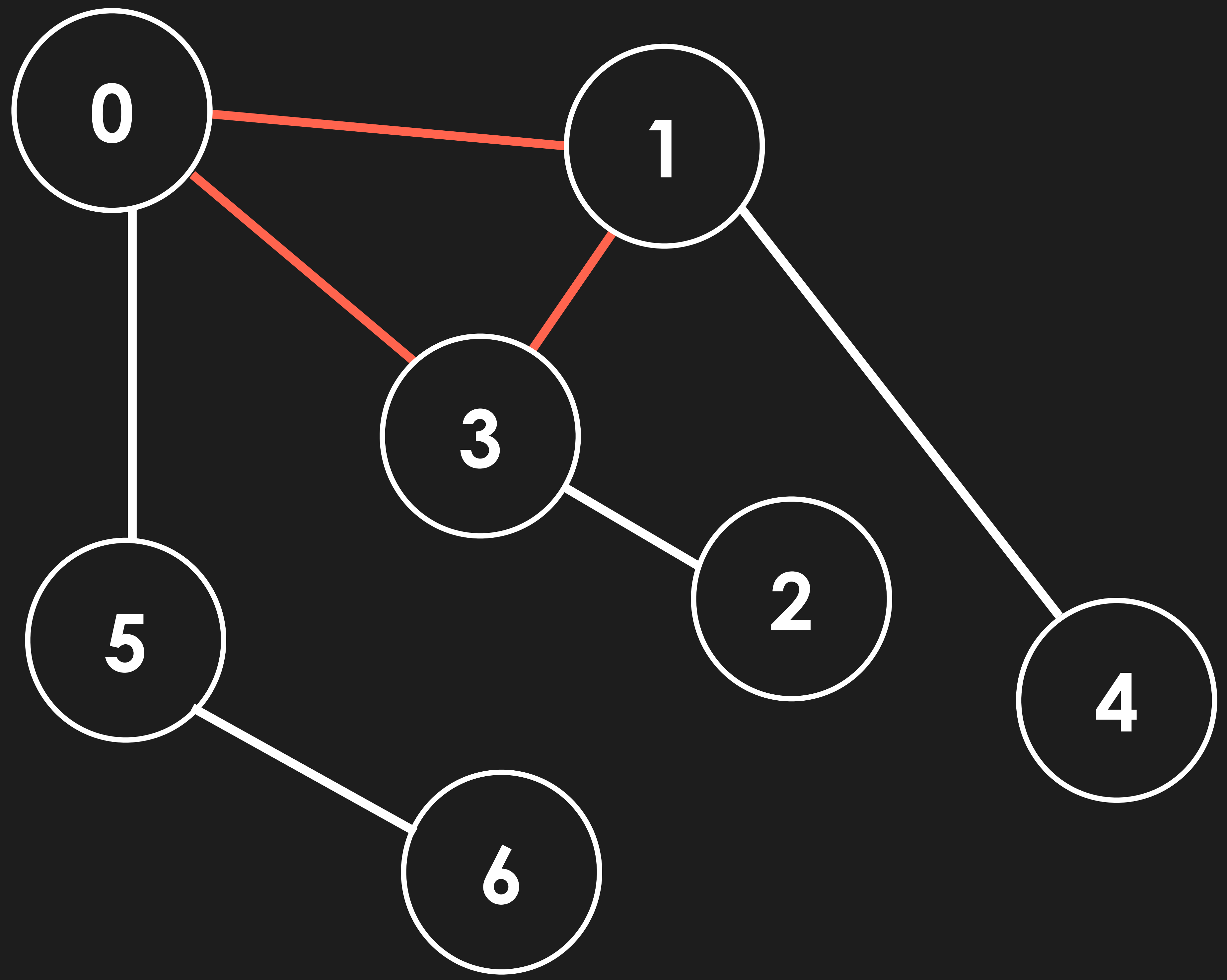
Application of DFS:

1. **BST**: Inorder, Postorder, Preorer Traversal
2. **Graphs**: Cyclic Detection
3. **Acyclic Digraphs** (Topological Sort) [You will learn this later on]

Cyclic Detection



Cyclic Detection

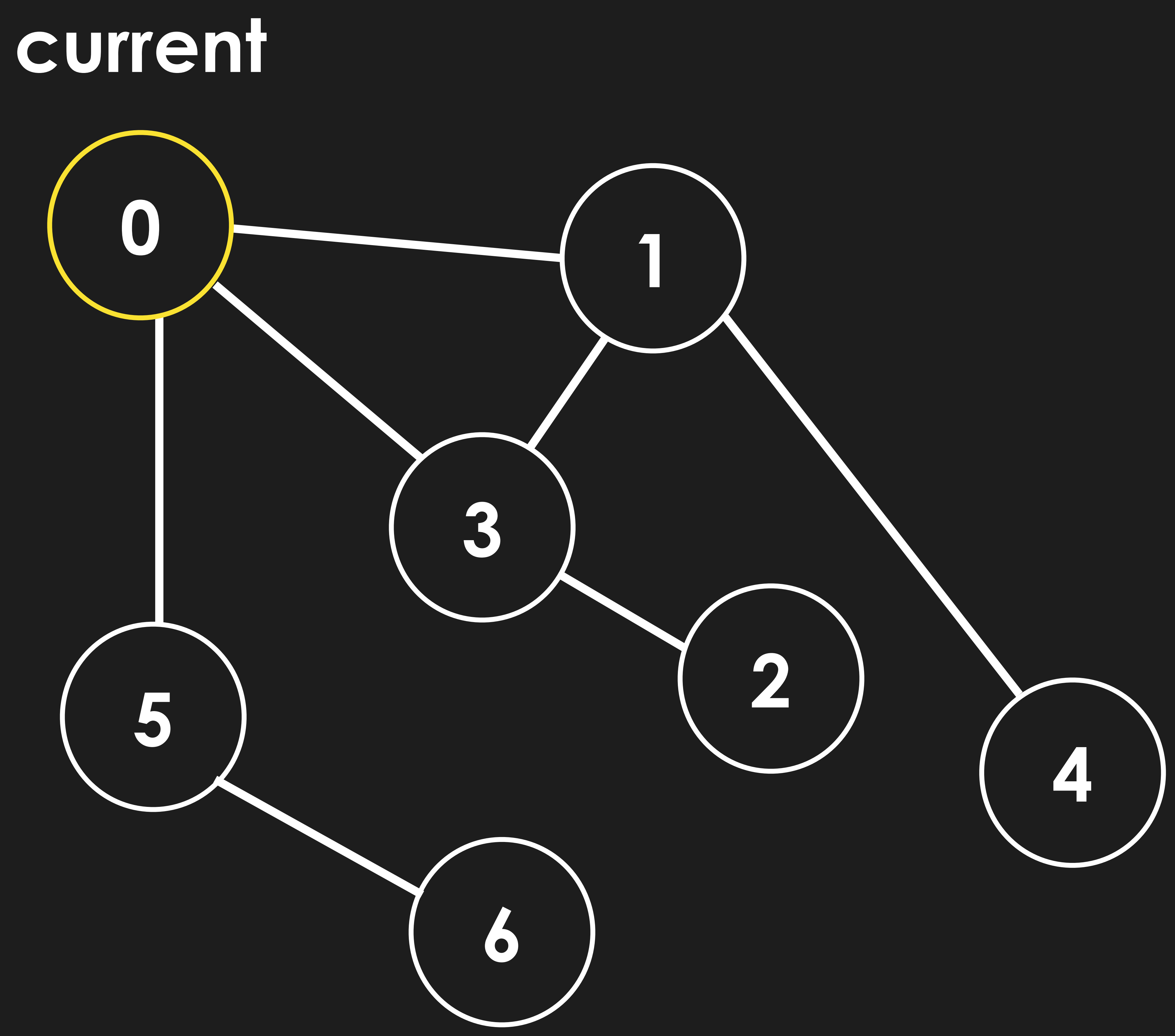


This is a cycle!

Pseudo code for cyclic detection:

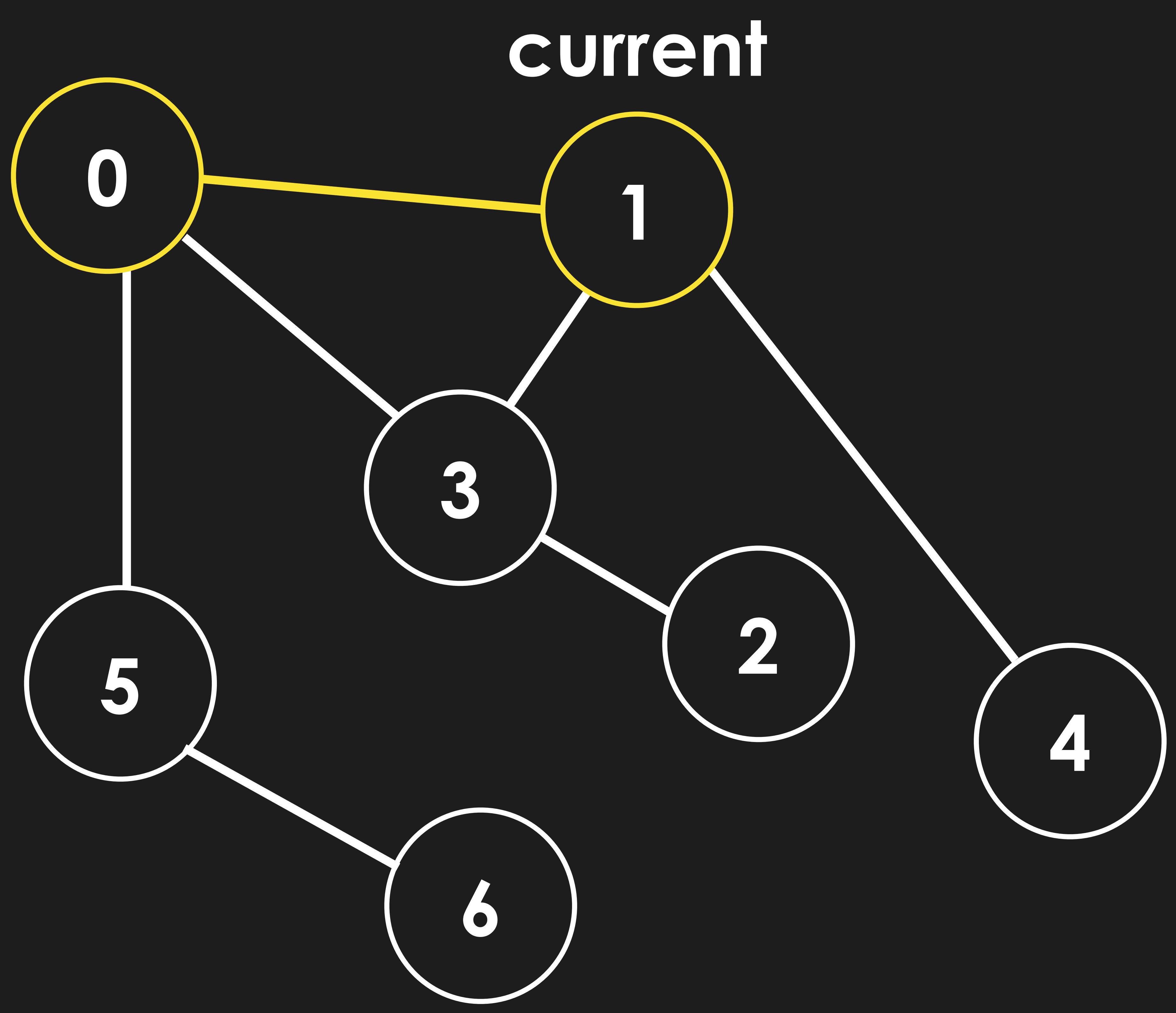
1. **dfs** graph
2. keep track of nodes in **current path**
3. if **visiting node in current path**, cycle is detected

Cycle Check Visualisation (In Undirected Graphs)

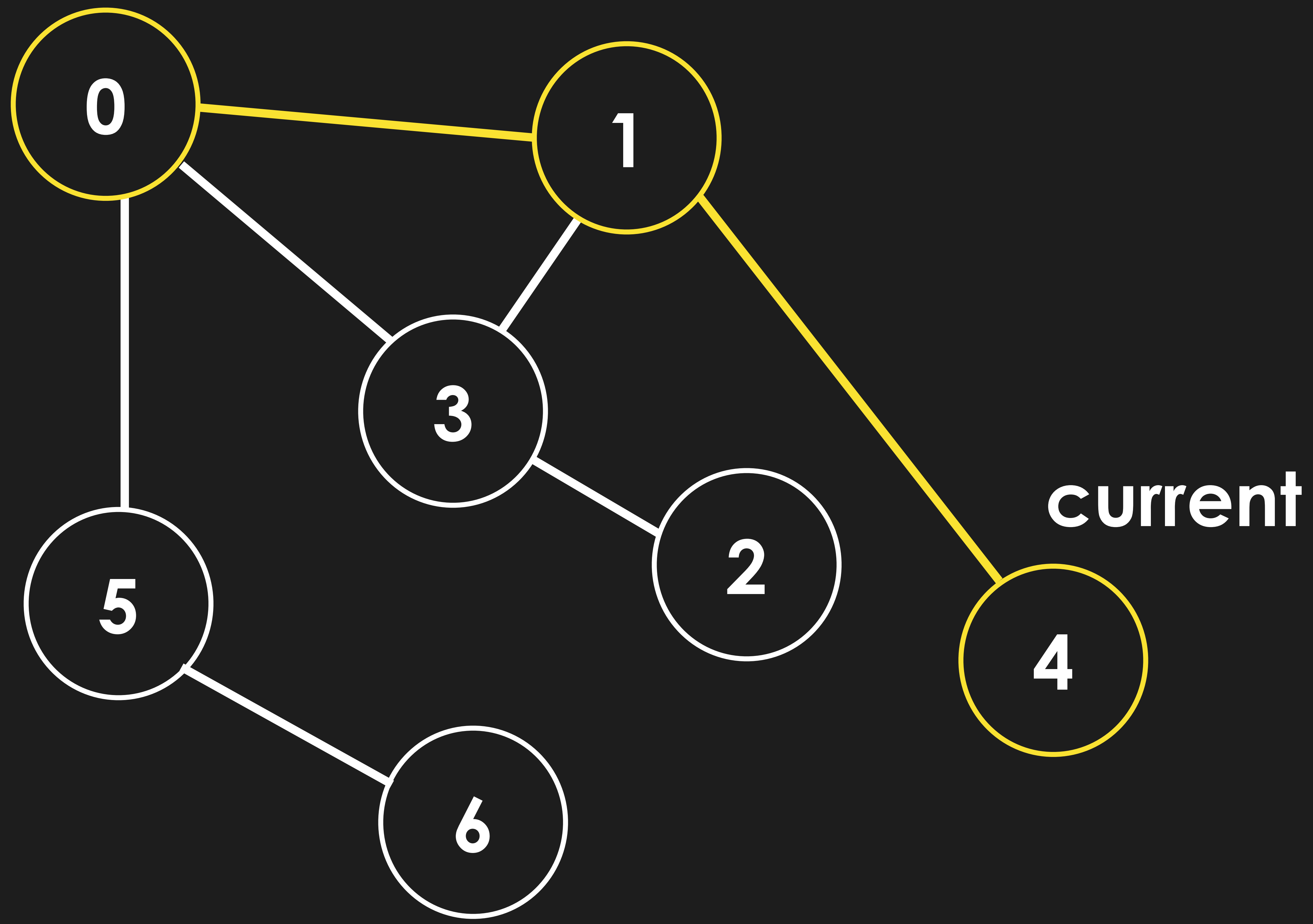


	visited
0	True
1	
2	
3	
4	
5	
6	

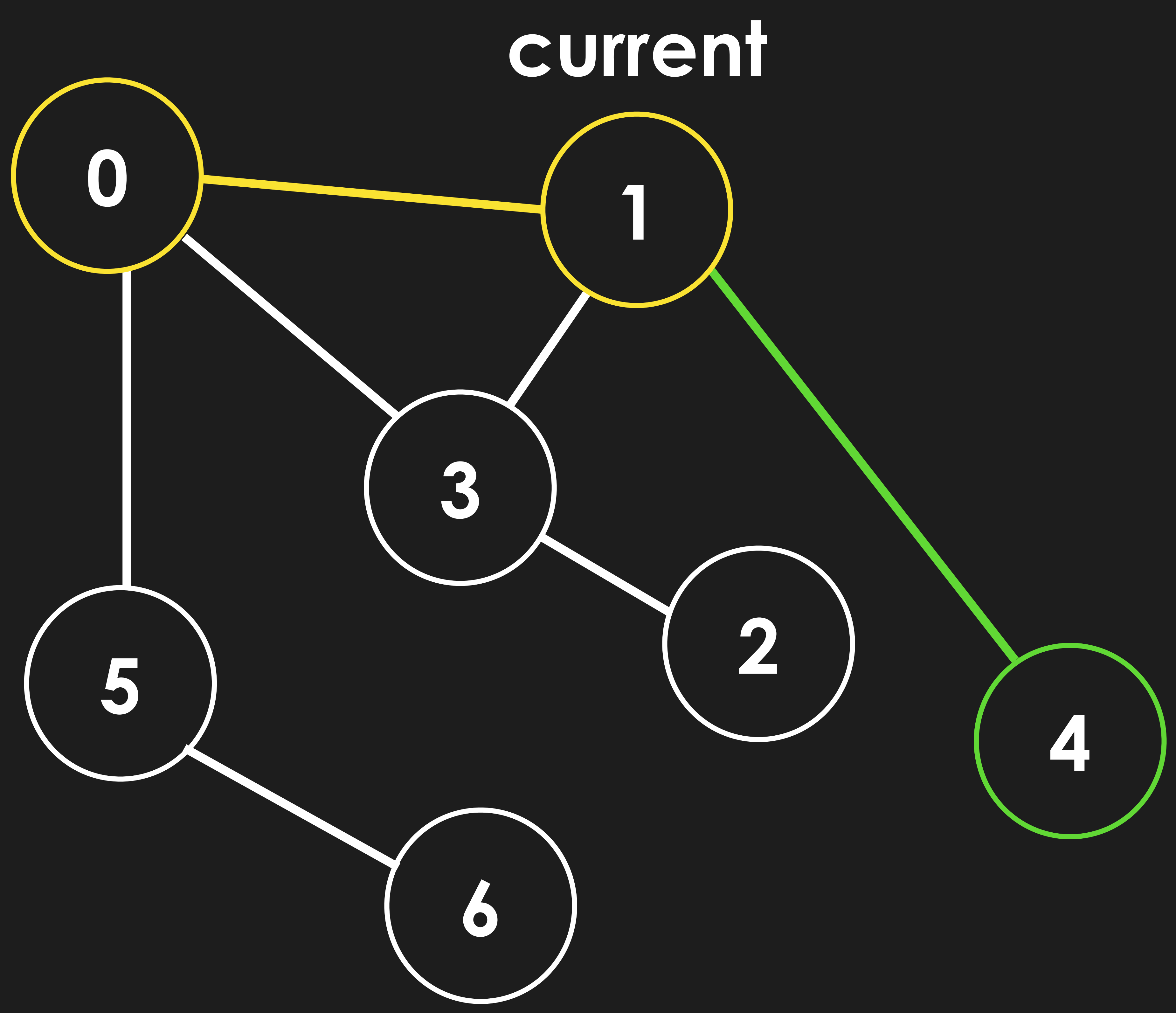
	currentPath
0	True
1	
2	
3	
4	
5	
6	



	visited	currentPath
0	True	True
1	True	True
2		
3		
4		
5		
6		



	visited	currentPath
0	True	True
1	True	True
2		
3		
4	True	True
5		
6		



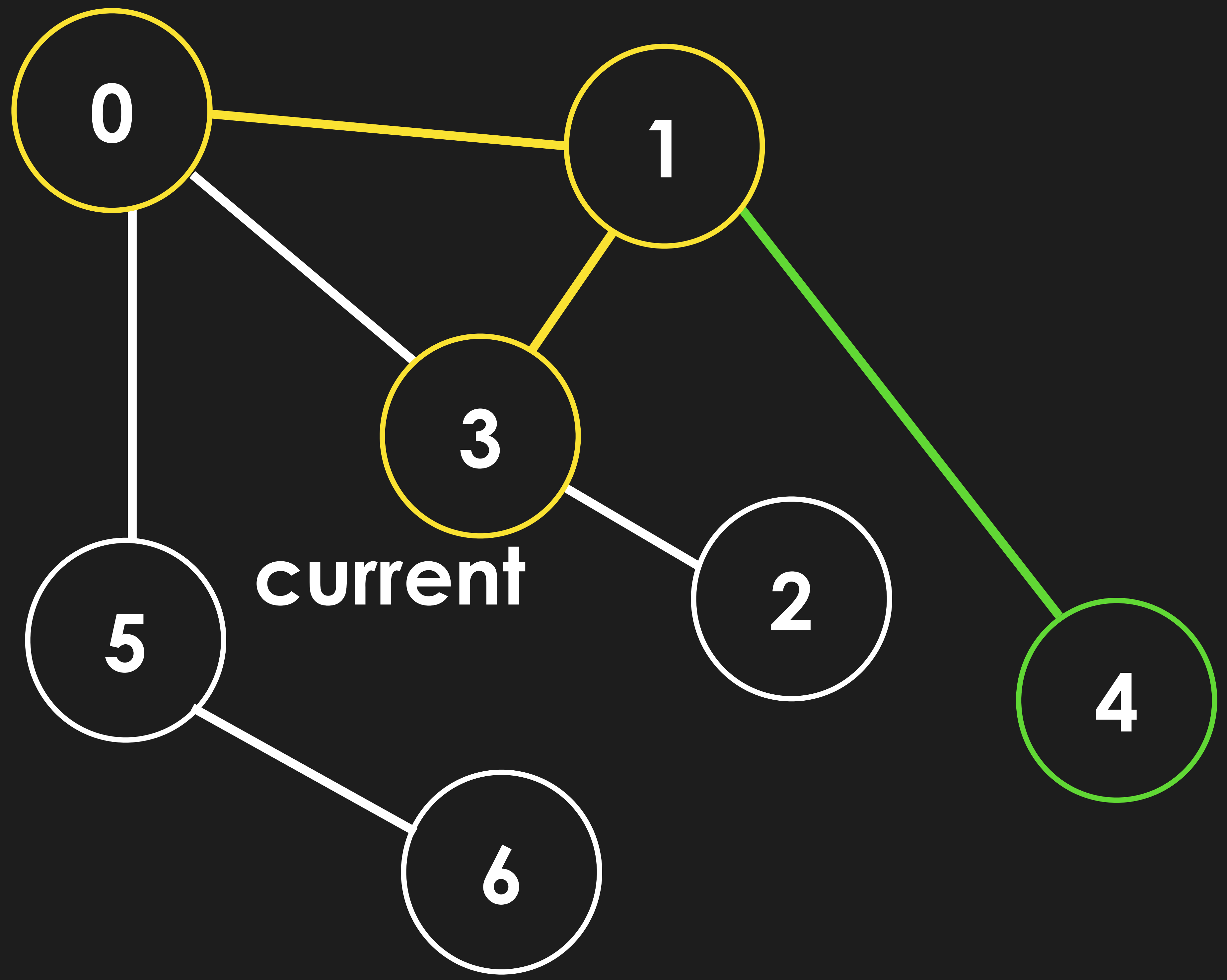
visited

0	True
1	True
2	
3	
4	True
5	
6	

currentPath

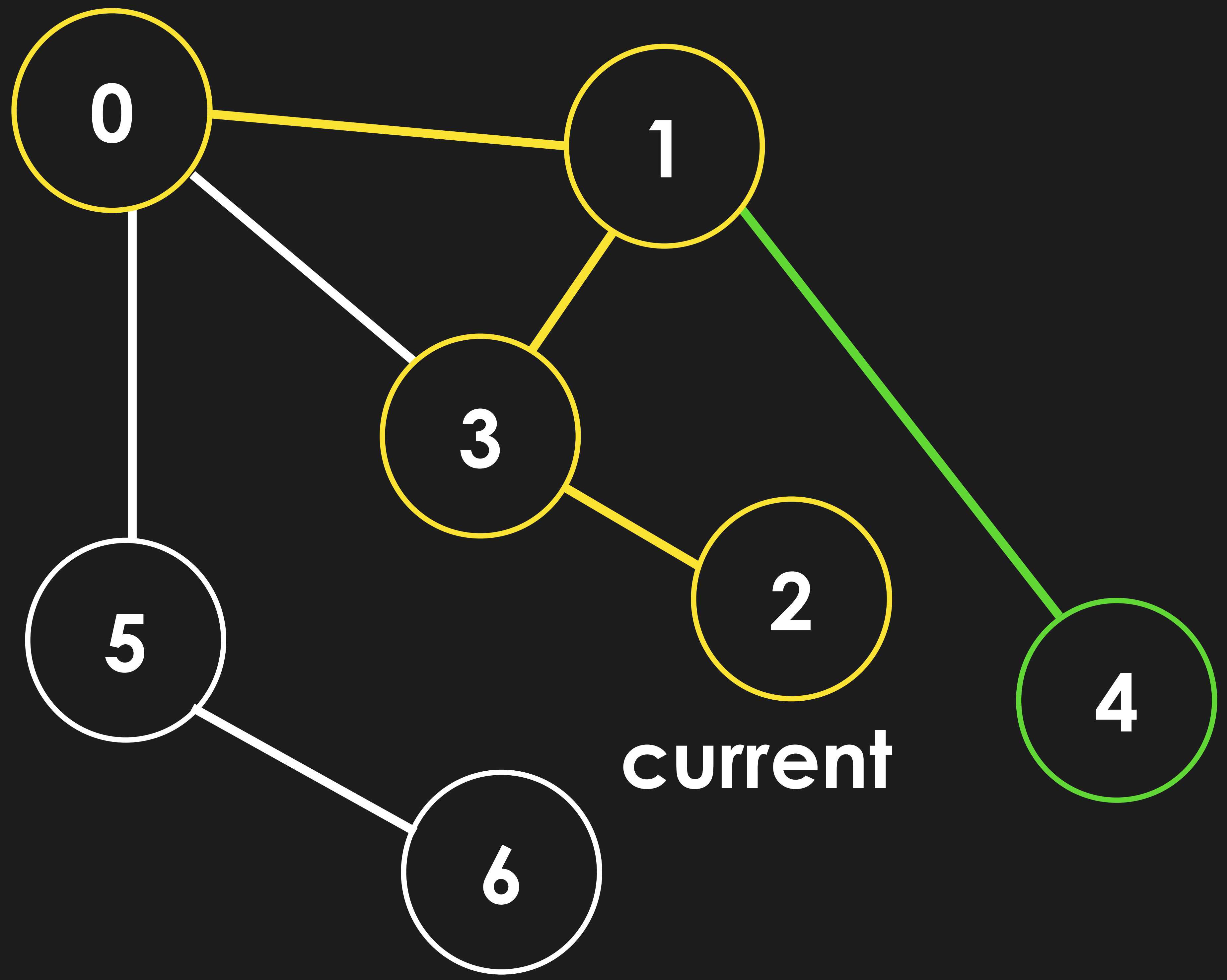
0	True
1	True
2	
3	
4	
5	
6	

CodeIT

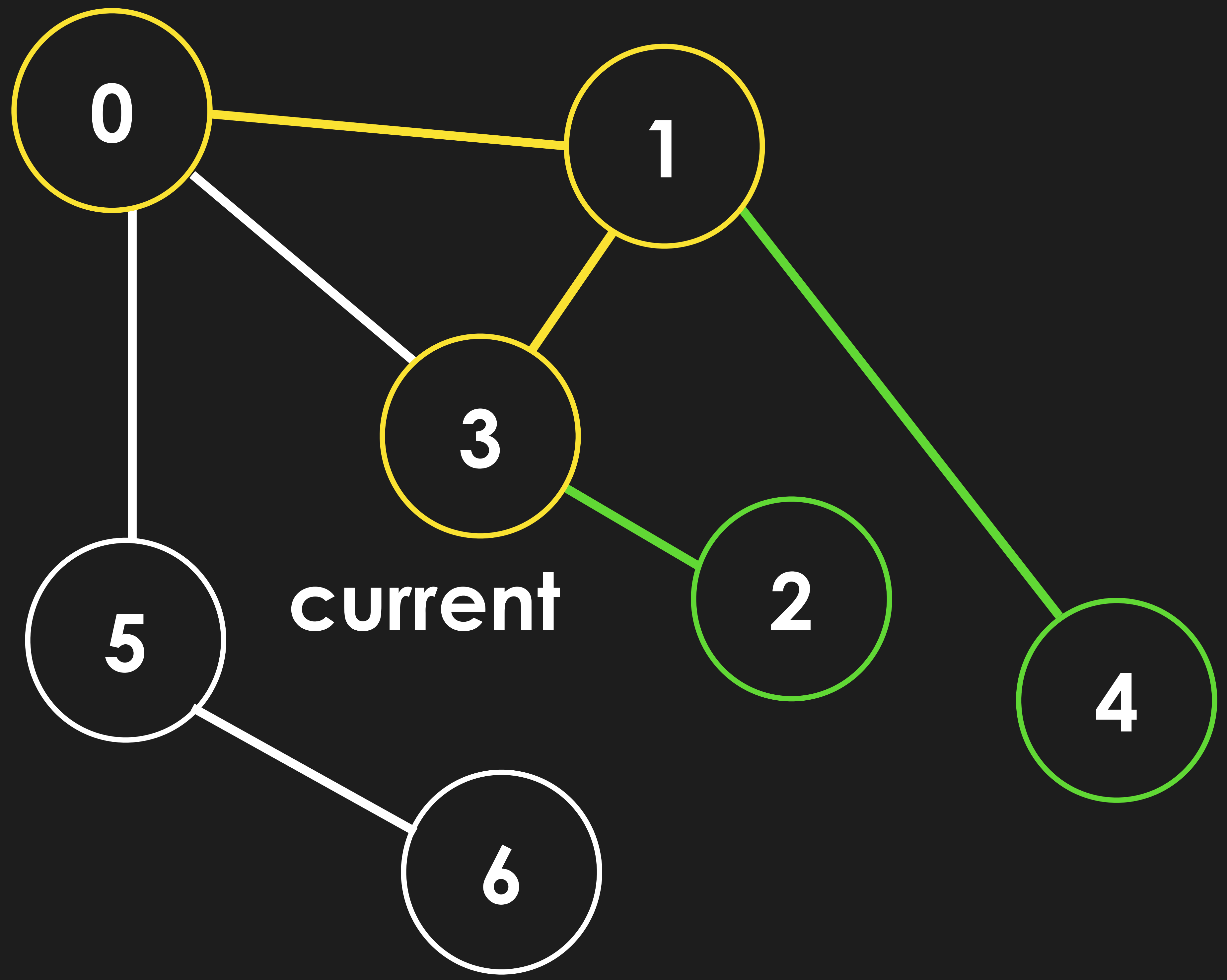


	visited
0	True
1	True
2	
3	True
4	True
5	
6	

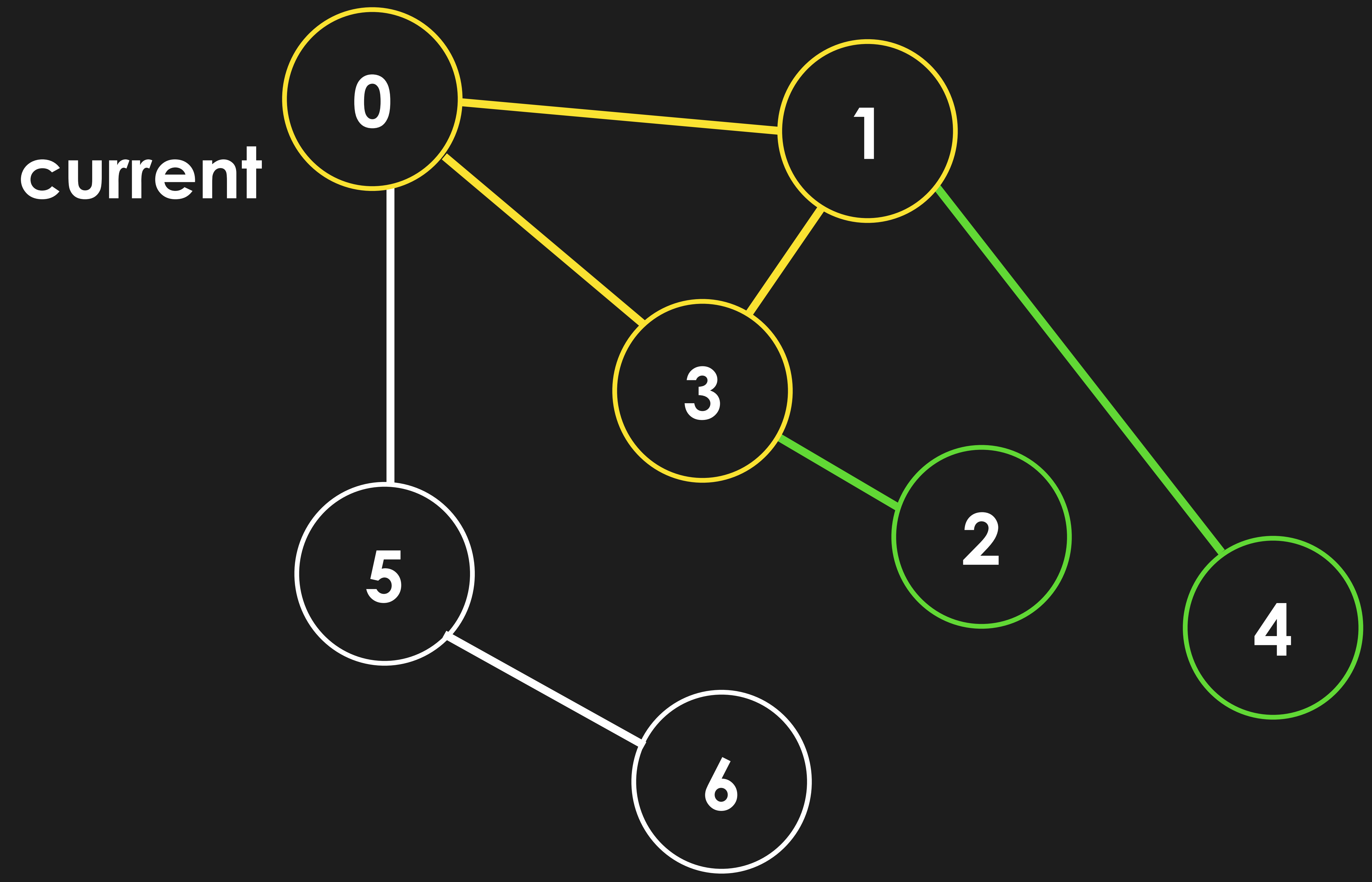
	currentPath
0	True
1	True
2	
3	True
4	
5	
6	



	visited	currentPath
0	True	True
1	True	True
2	True	True
3	True	True
4	True	
5		
6		

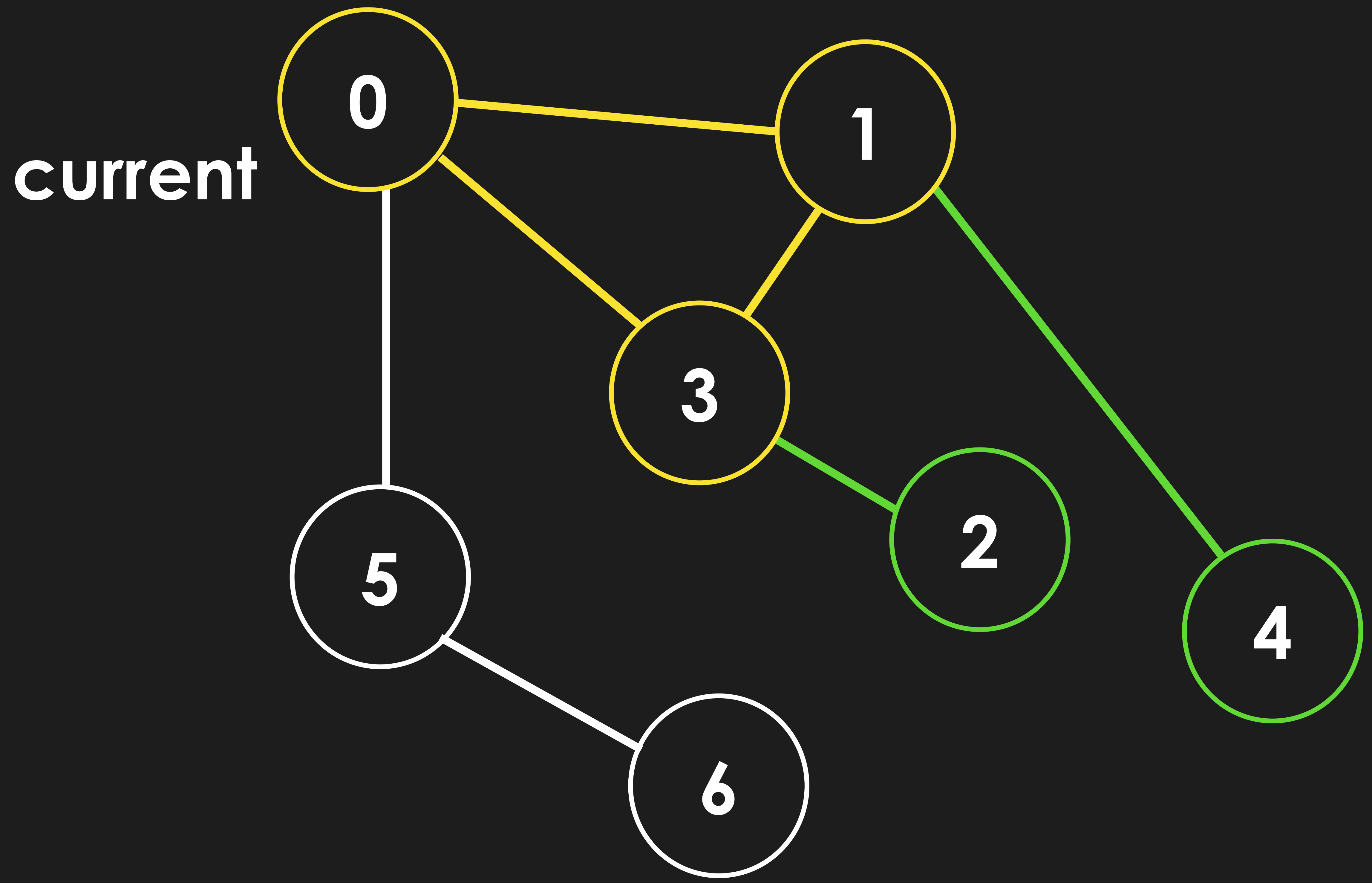


	visited	currentPath
0	True	True
1	True	True
2	True	
3	True	True
4	True	
5		
6		



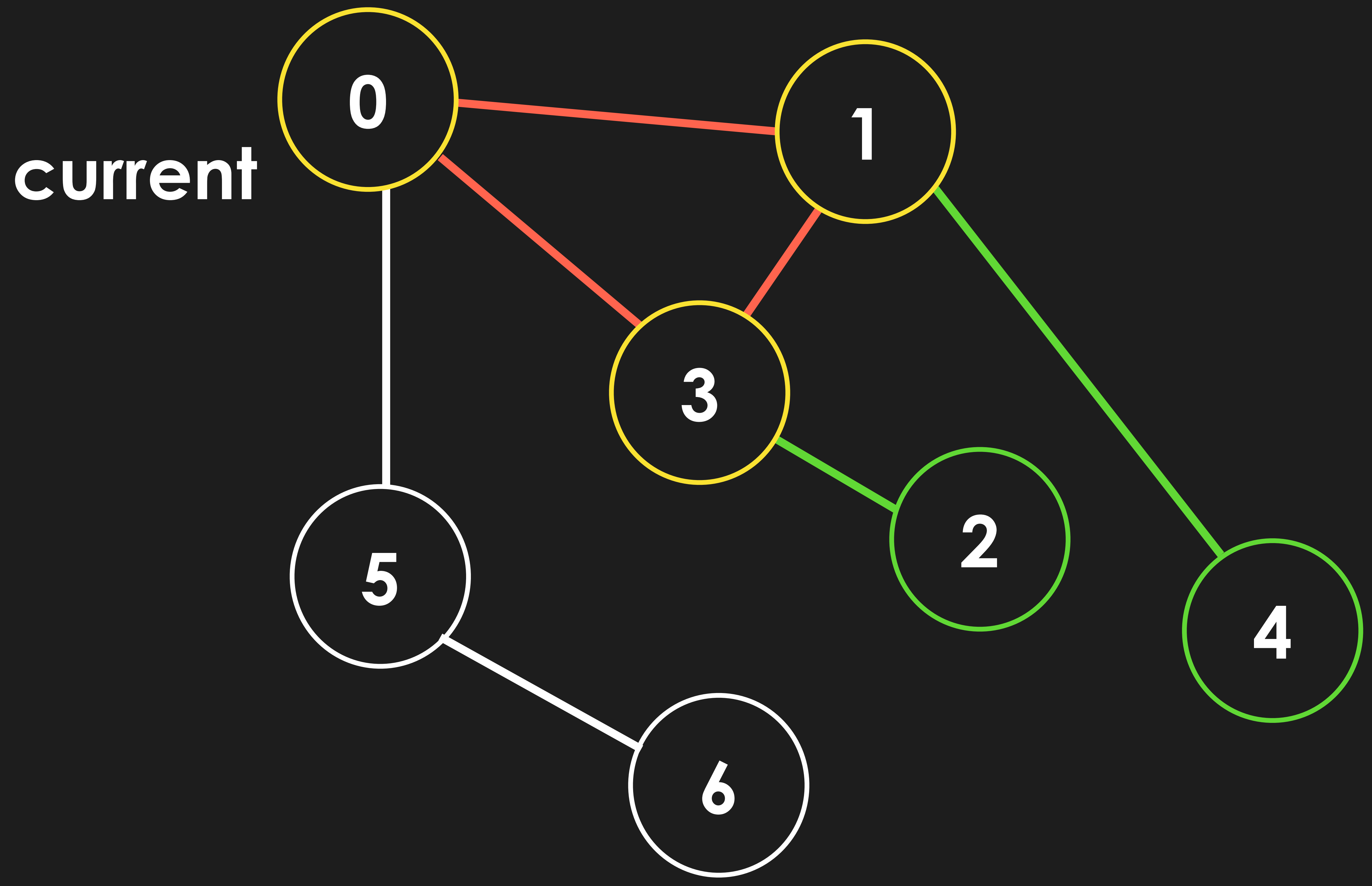
	visited
0	True
1	True
2	True
3	True
4	True
5	
6	

	currentPath
0	True
1	True
2	
3	True
4	
5	
6	



	visited	currentPath
0	True	True
1	True	True
2	True	
3	True	True
4	True	
5		
6		

0 is in current path!



	visited	currentPath
0	True	True
1	True	True
2	True	
3	True	True
4	True	
5		
6		

0 is in current path!

Breadth First Search

Breadth First Search

At each node, **visit all it's edges**

Repeat for each edge visited

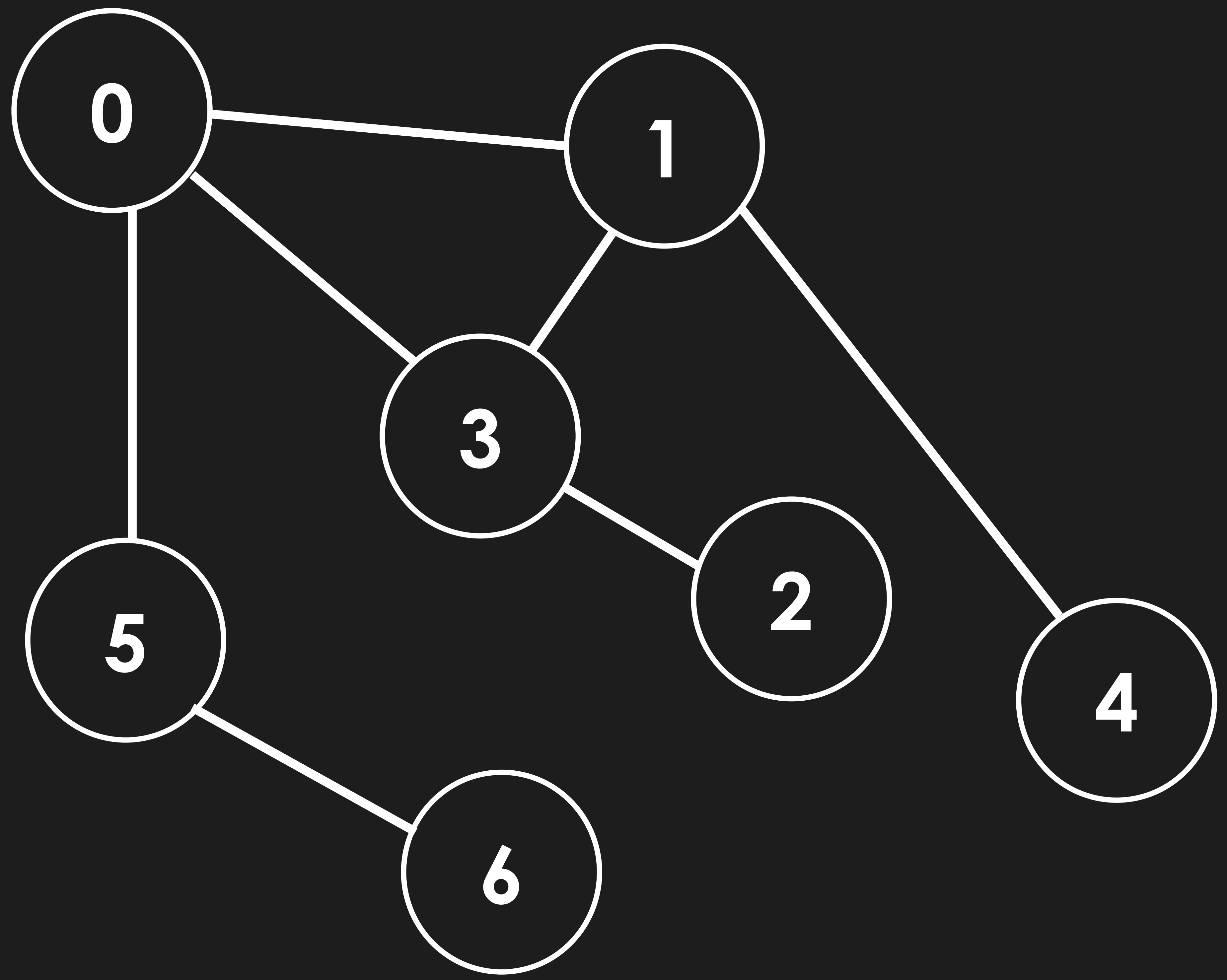
We achieve this by using a **queue ADT**

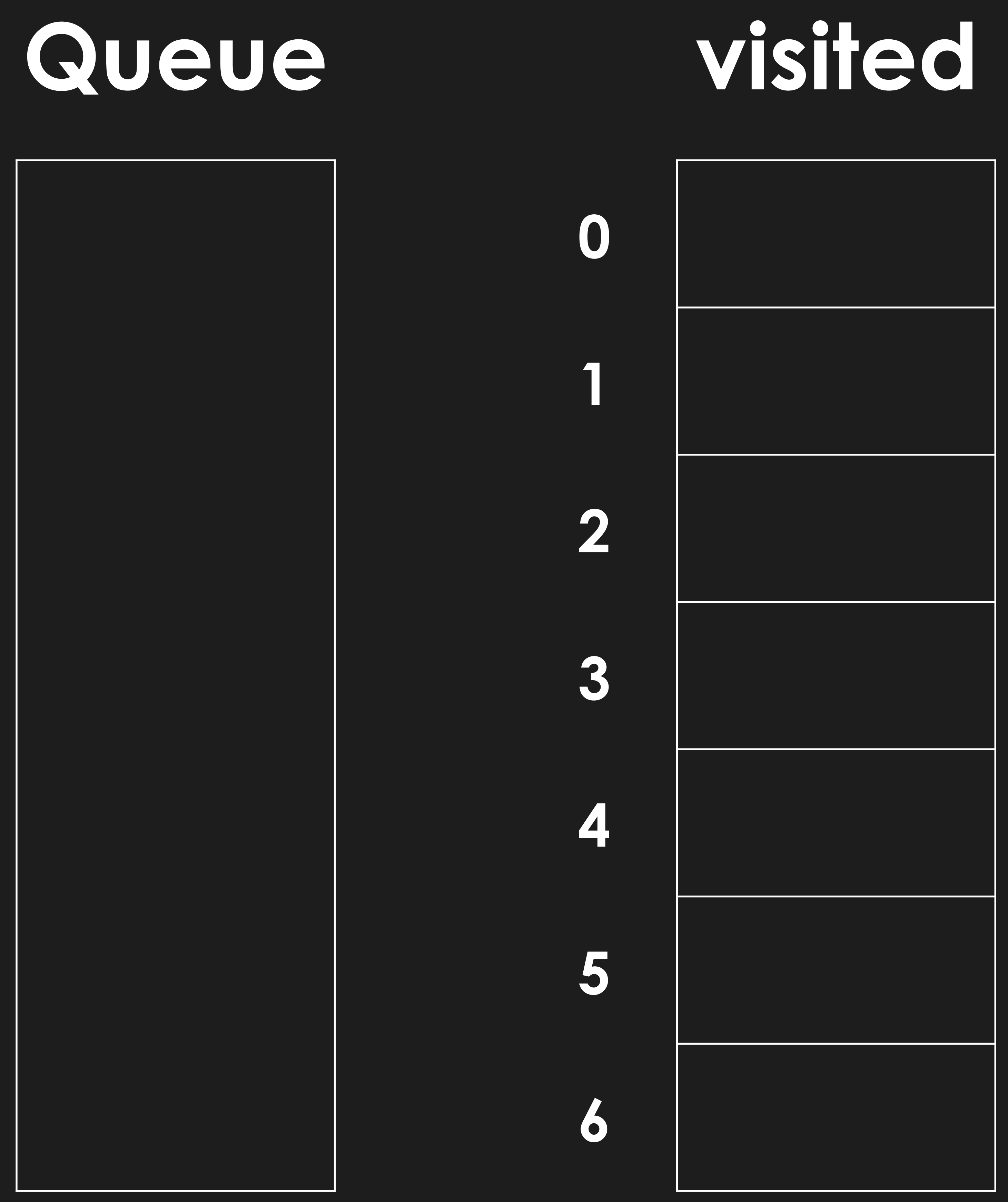
bfs

```python
def bfs(graph, start)
```

CodeIT

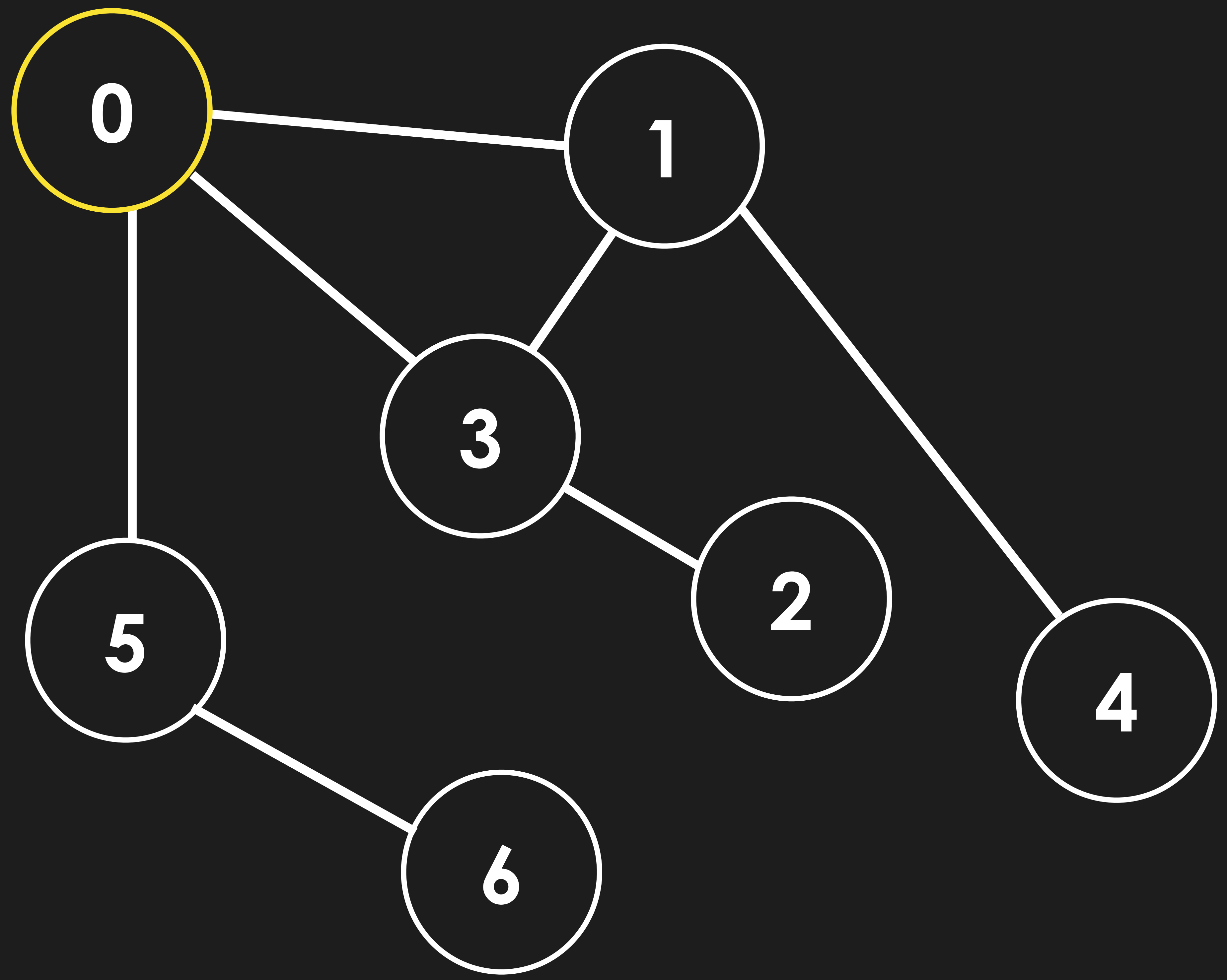
bfs(graph, 0)





bfs(graph, 0)

Queue start node



Queue

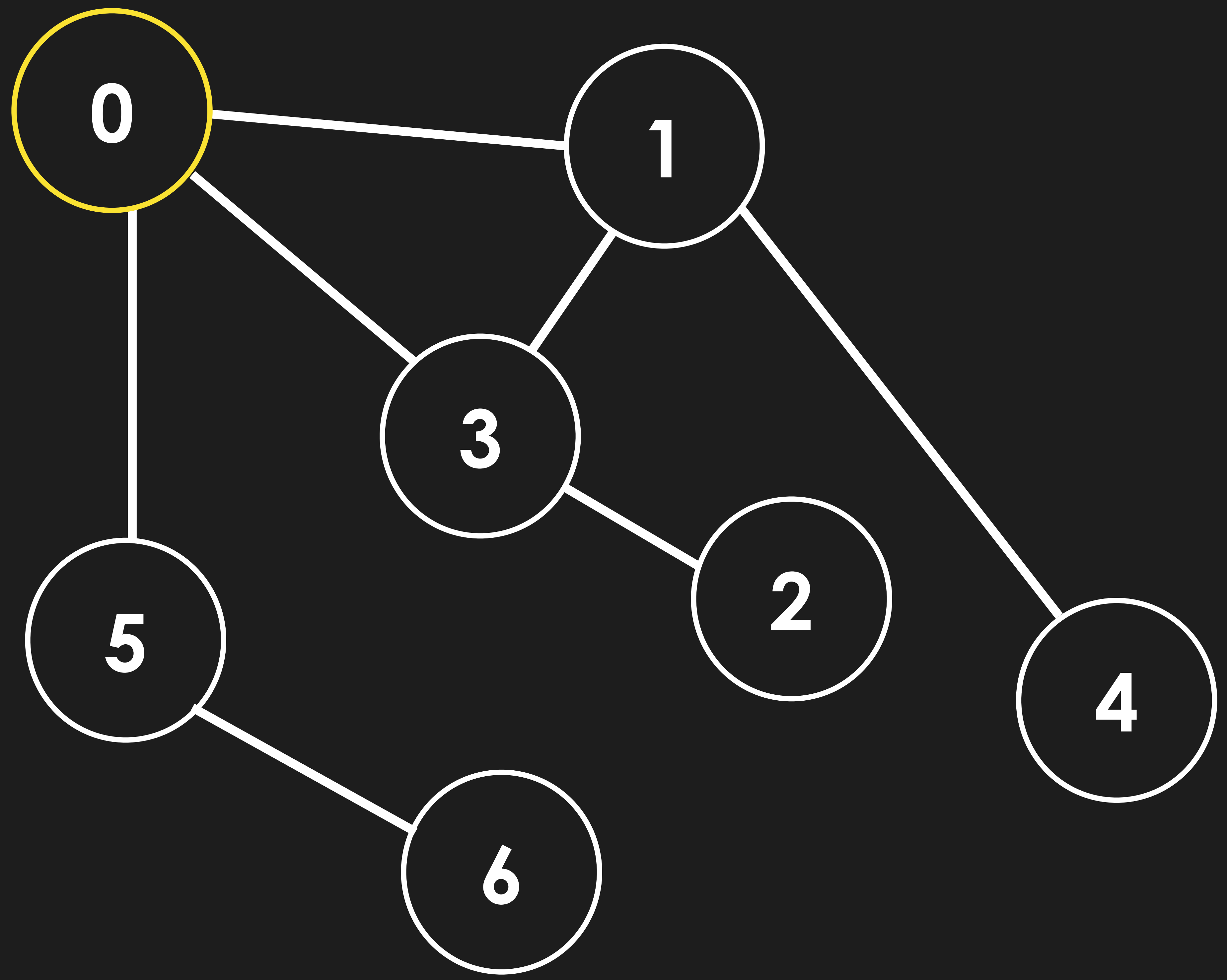
0

visited

0	True
1	
2	
3	
4	
5	
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



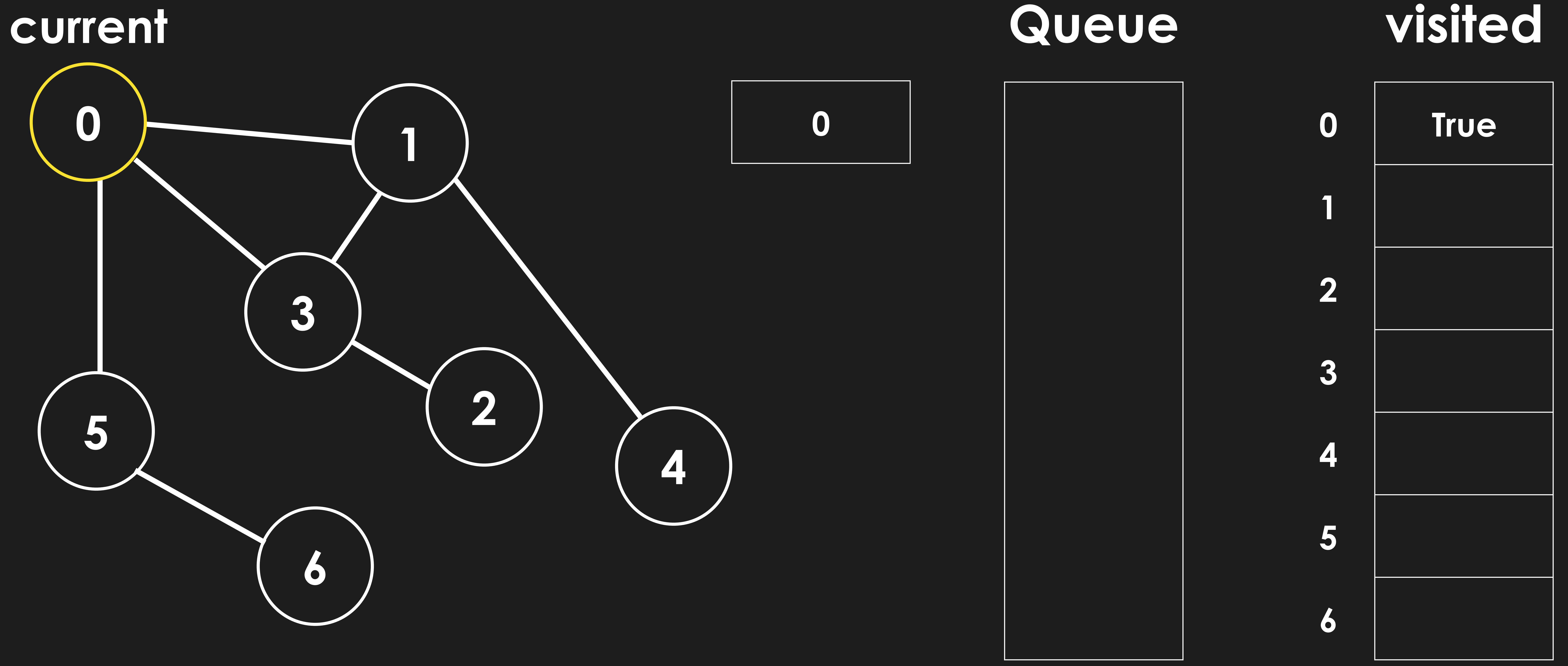
Queue
0

	visited
0	True
1	
2	
3	
4	
5	
6	

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges

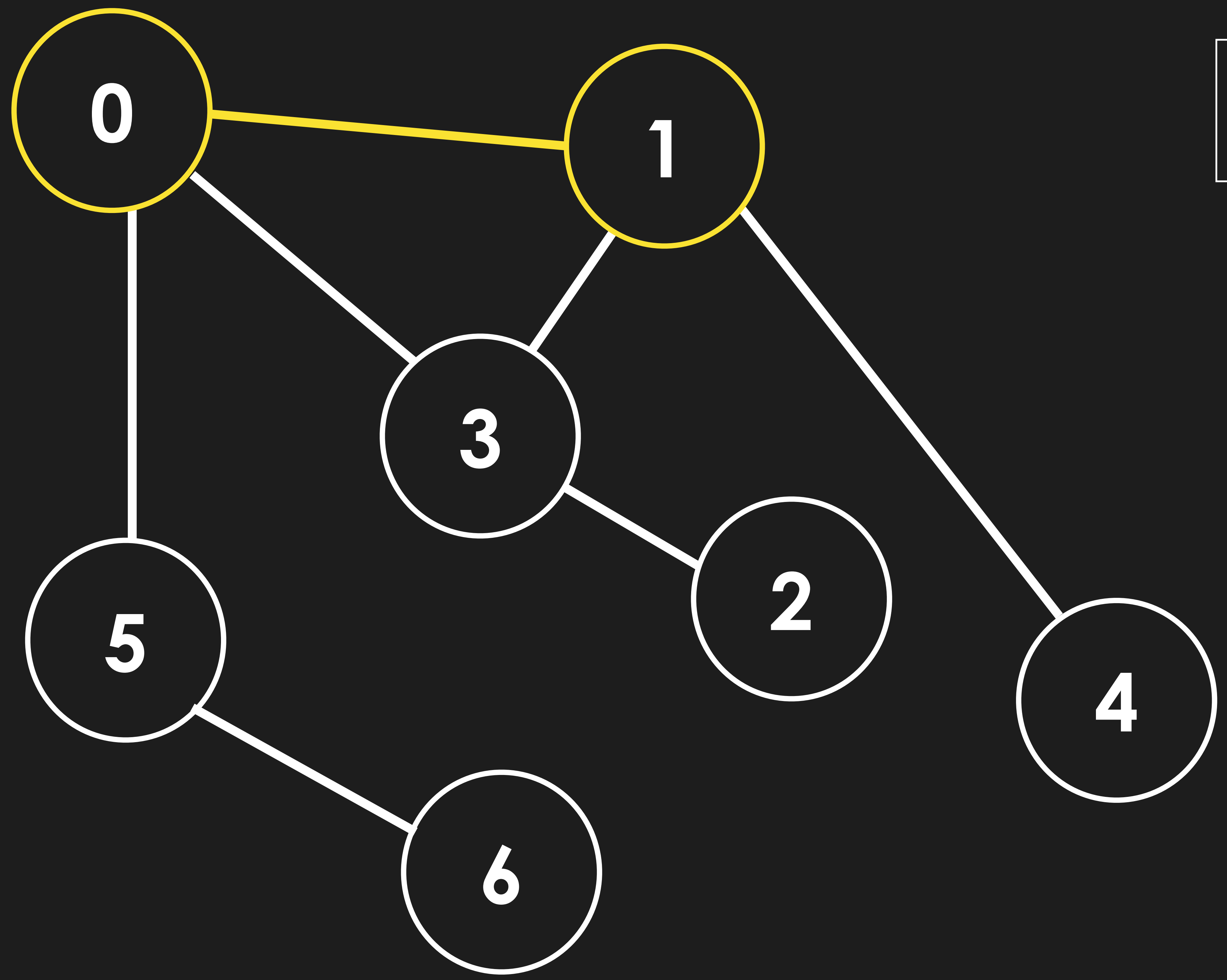


CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges

current



0

Queue

1

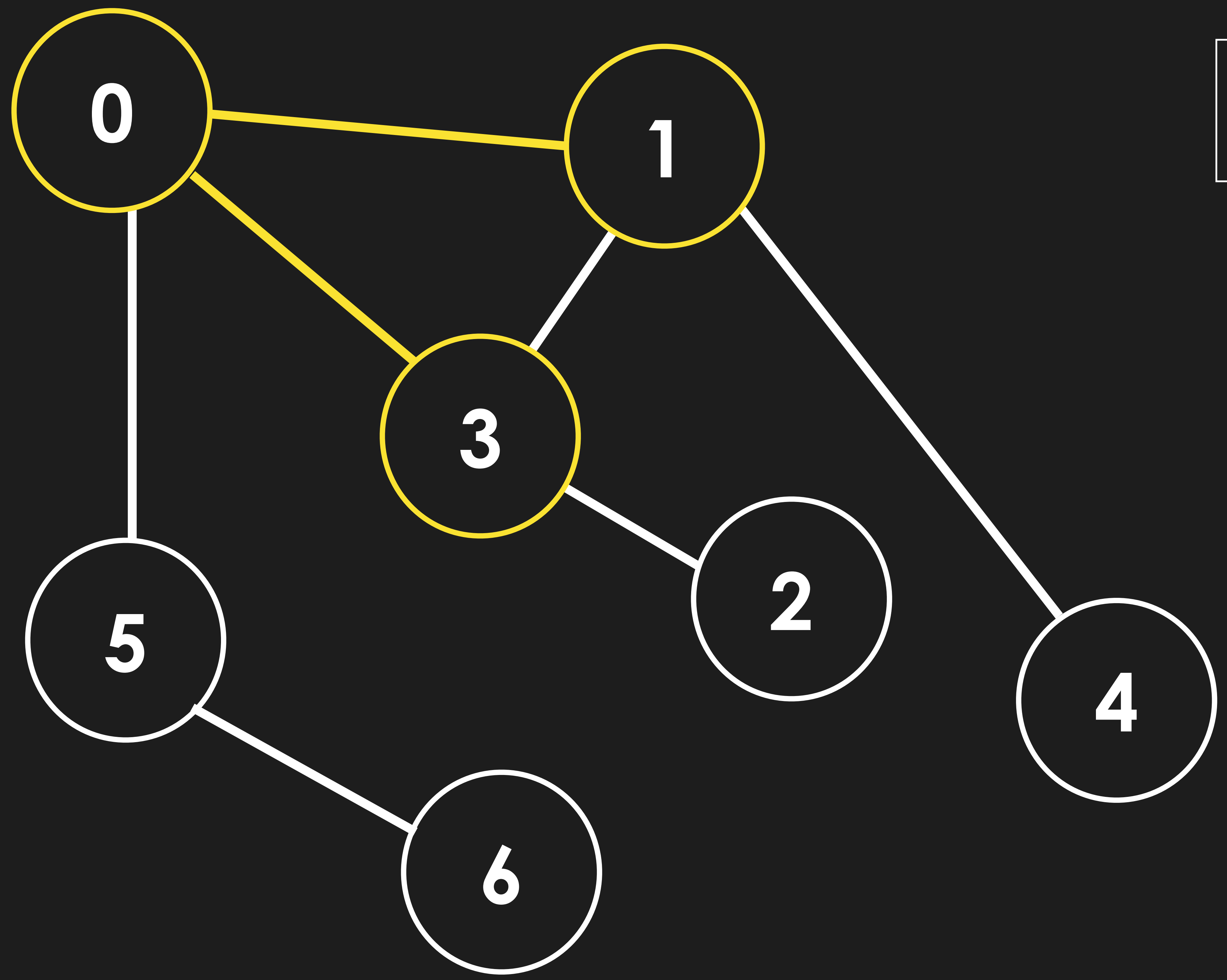
visited

0	True
1	True
2	
3	
4	
5	
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges

current



current
0

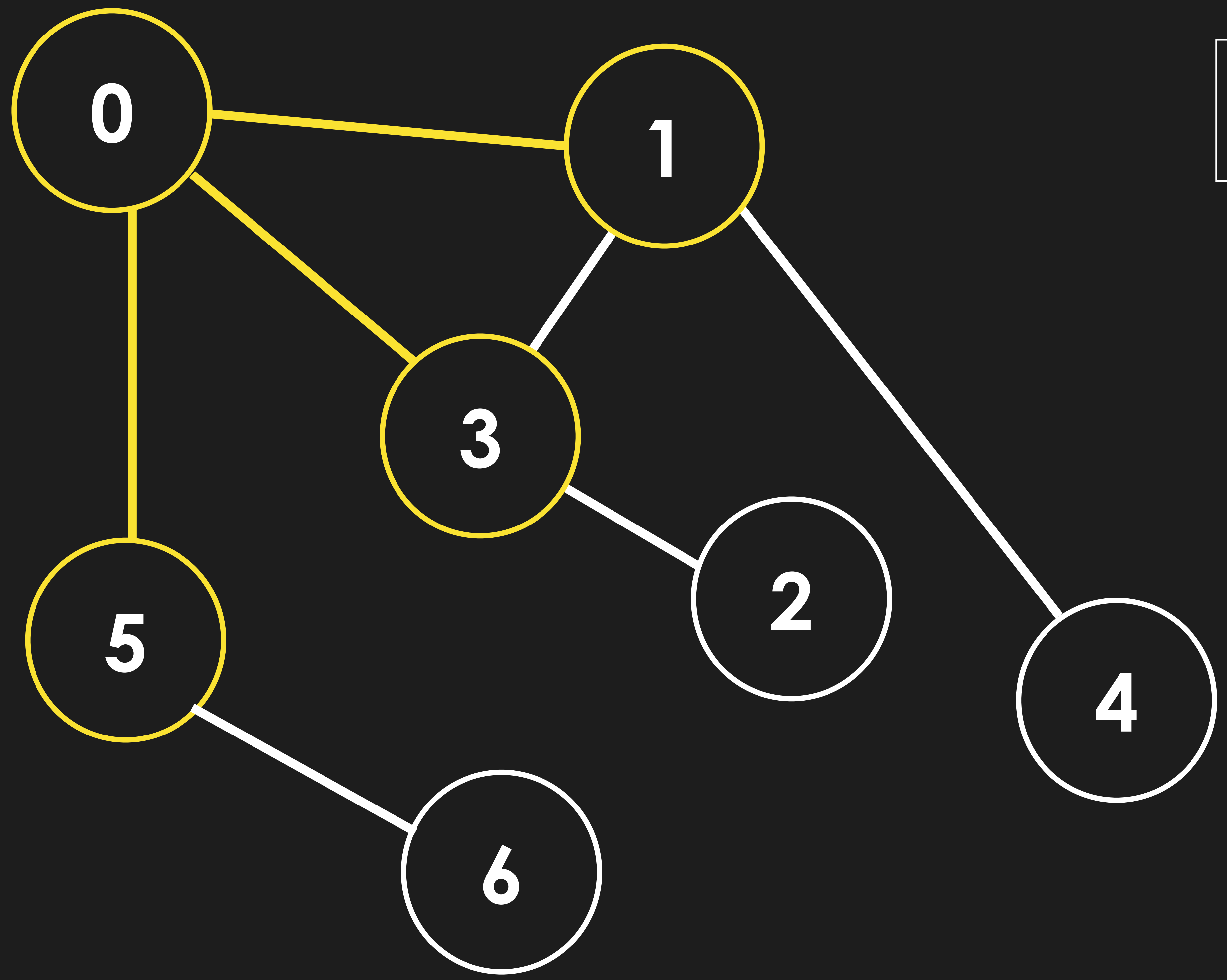
Queue
1
3

	visited
0	True
1	True
2	
3	True
4	
5	
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges

current



current
0

Queue
1
3
5

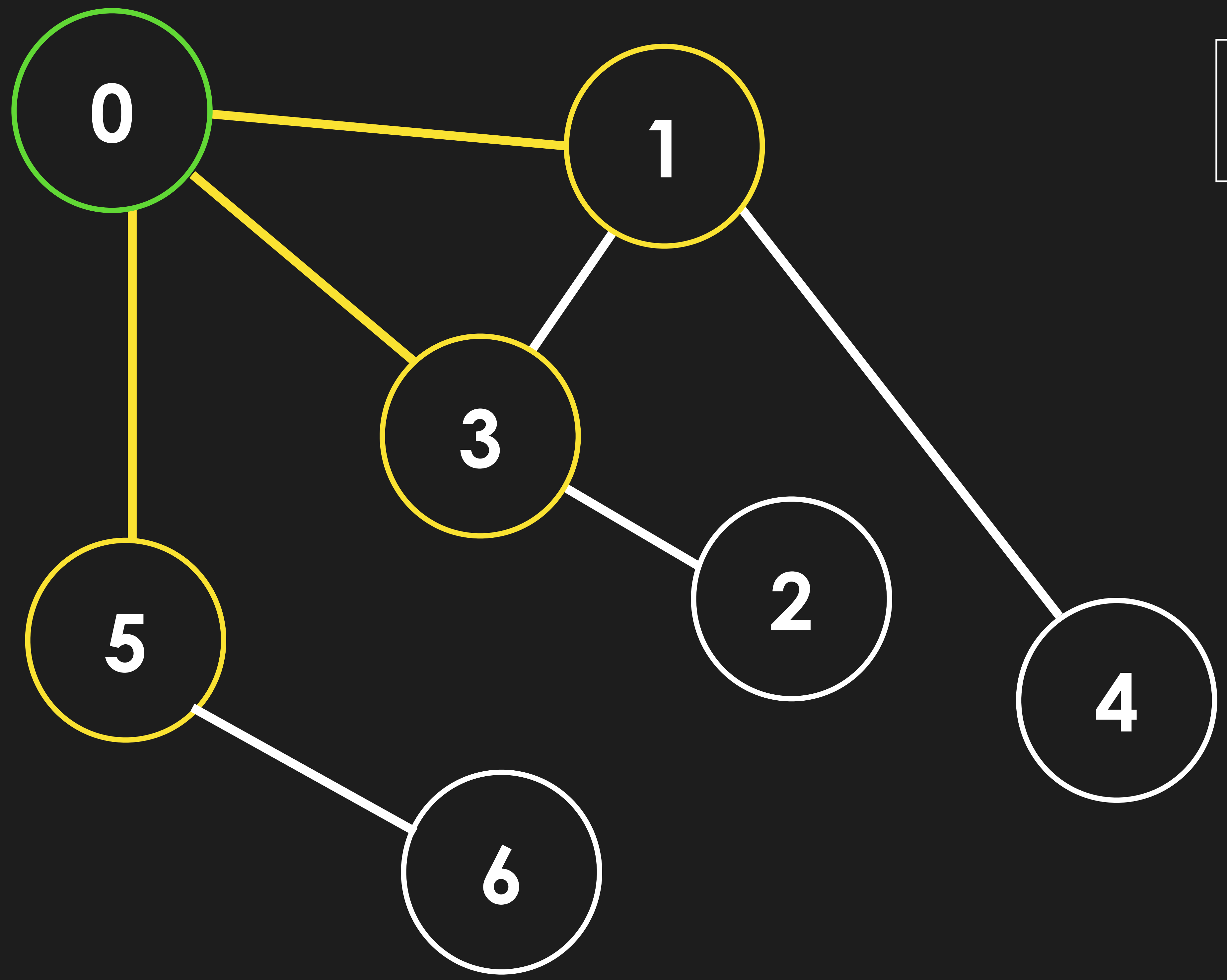
	visited
0	True
1	True
2	
3	True
4	
5	True
6	

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges

current



0

Queue

1
3
5

visited

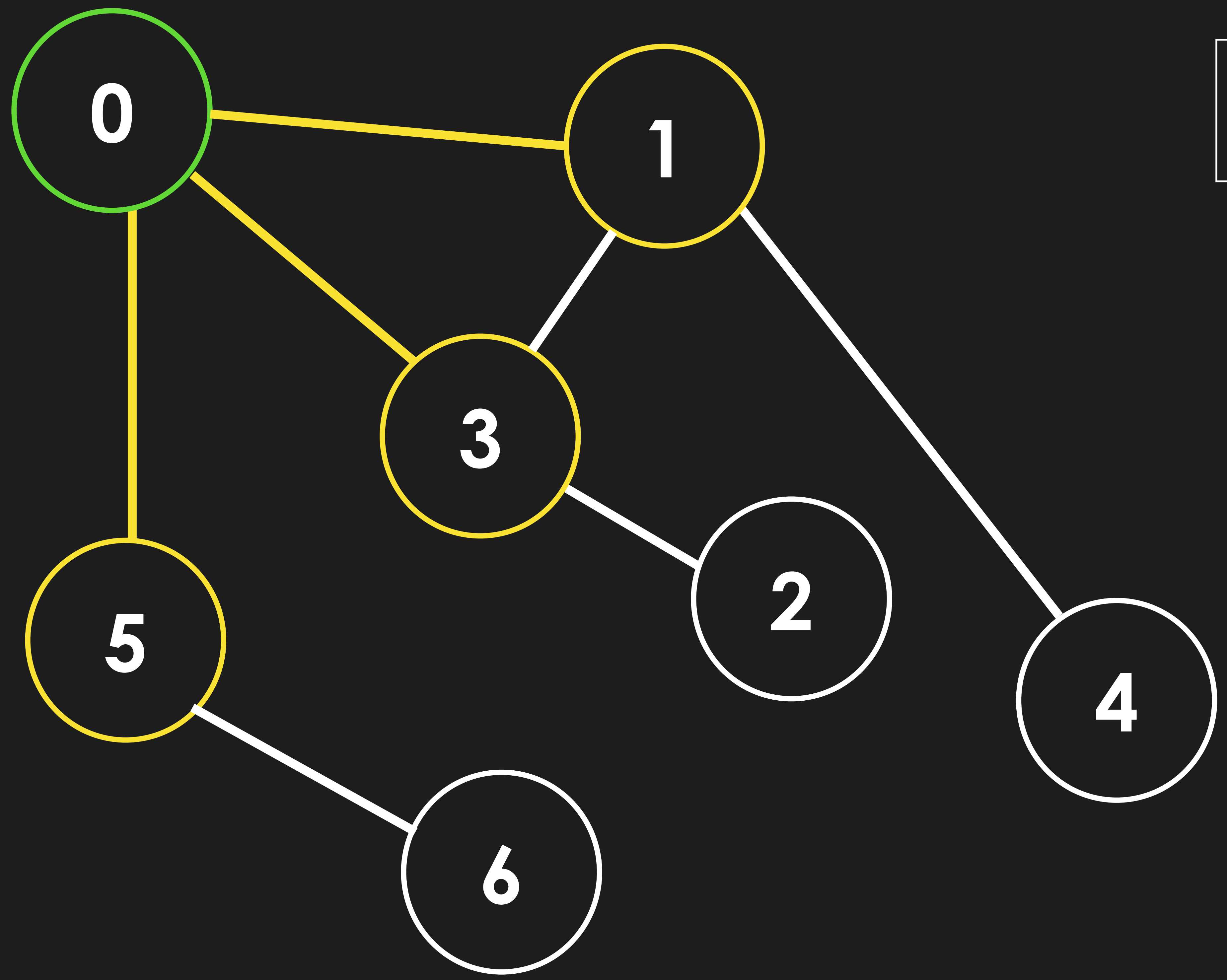
0	True
1	True
2	
3	True
4	
5	True
6	

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges

current



current
1

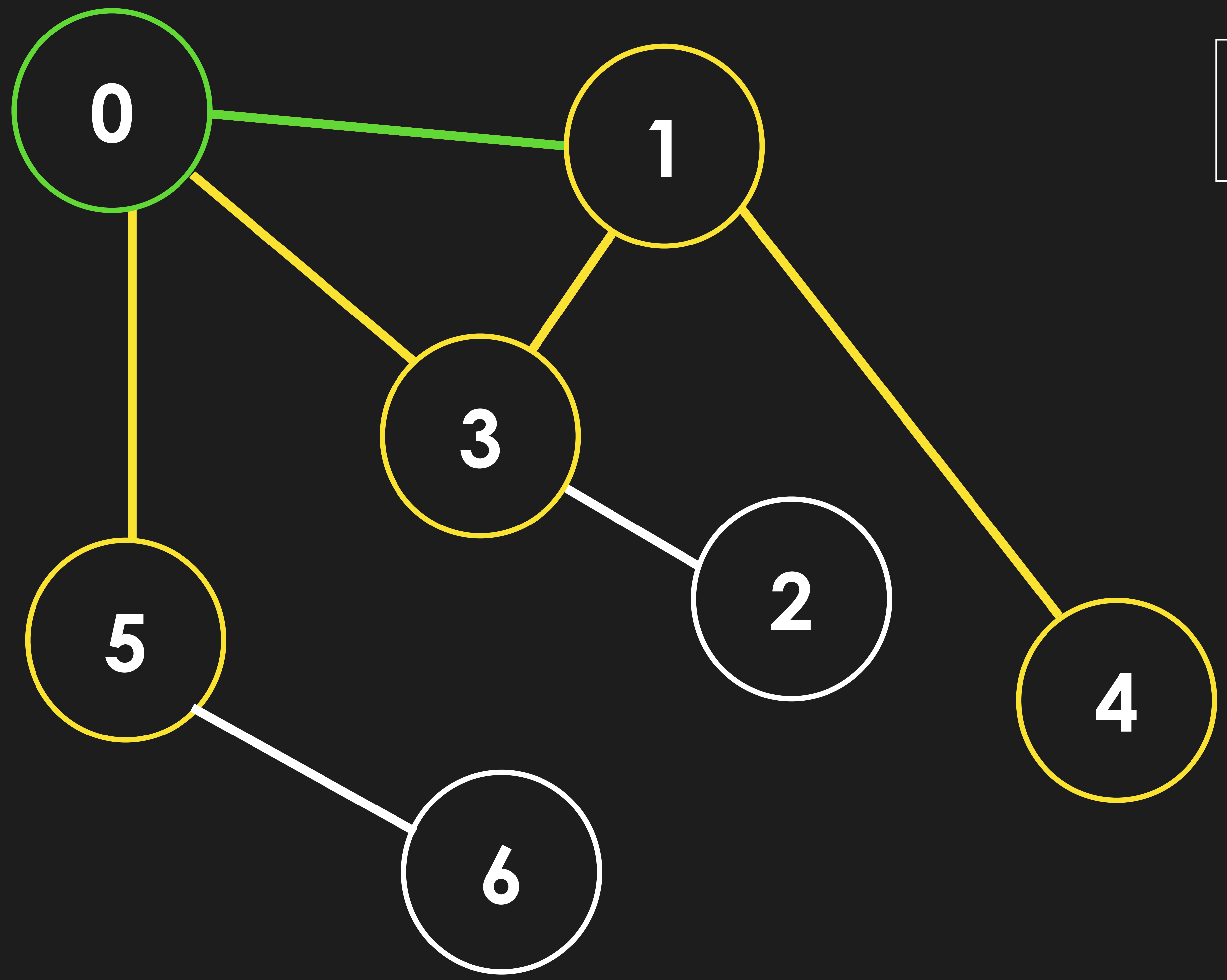
Queue
3
5

	visited
0	True
1	True
2	
3	True
4	
5	True
6	

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges



current

1

Queue

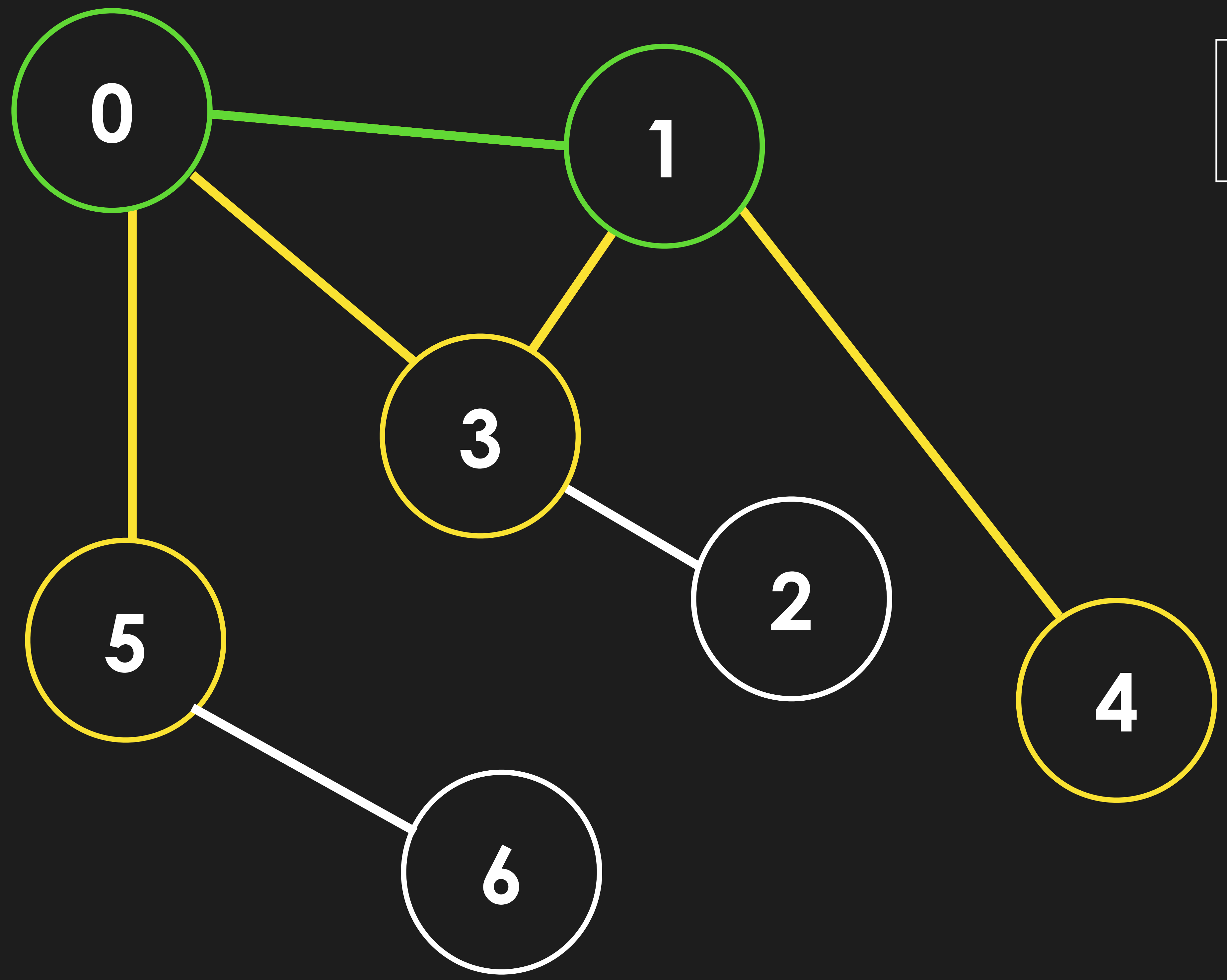
3
5
4

visited

0	True
1	True
2	
3	True
4	True
5	True
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



current

1

Queue

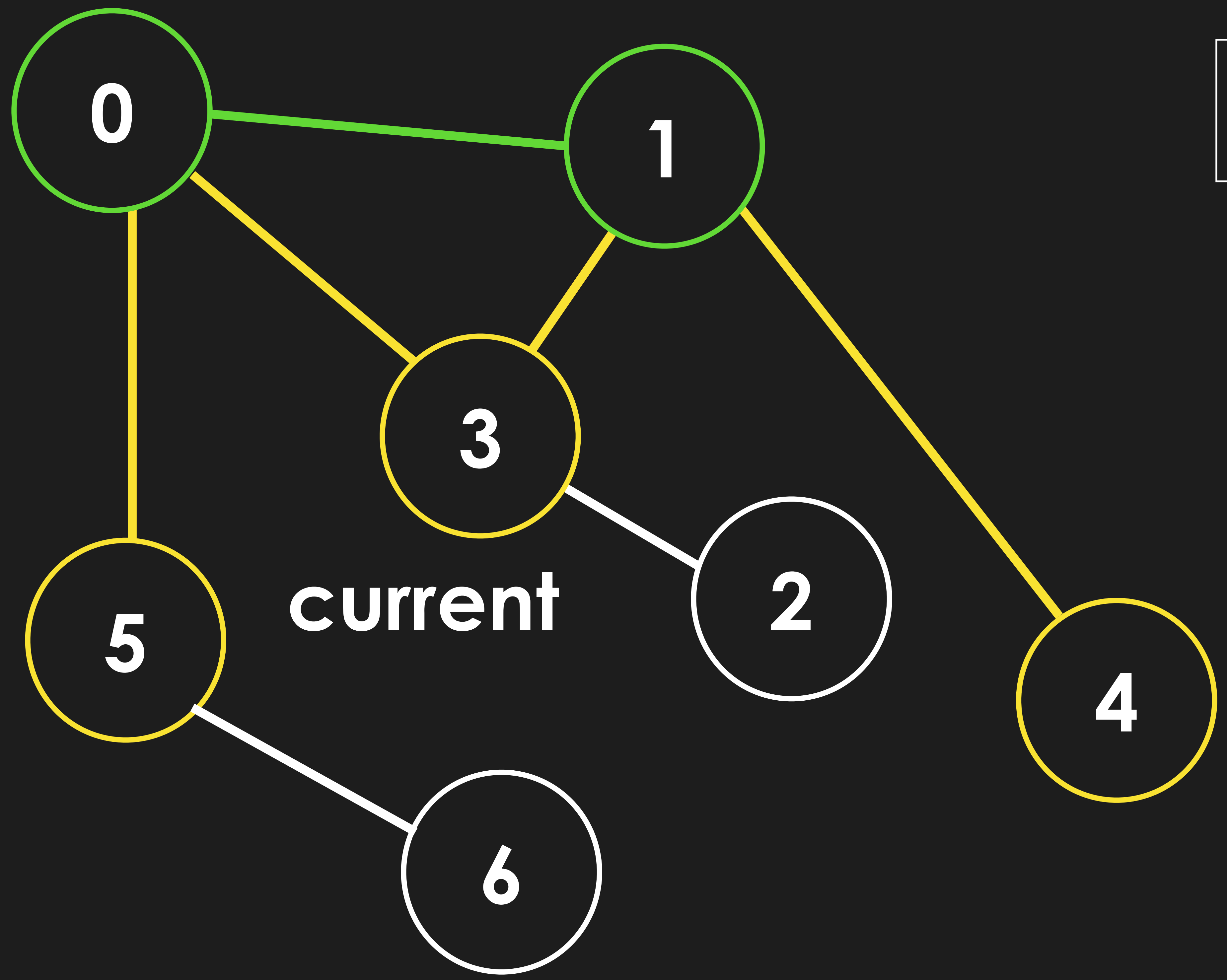
3
5
4

visited

0	True
1	True
2	
3	True
4	True
5	True
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



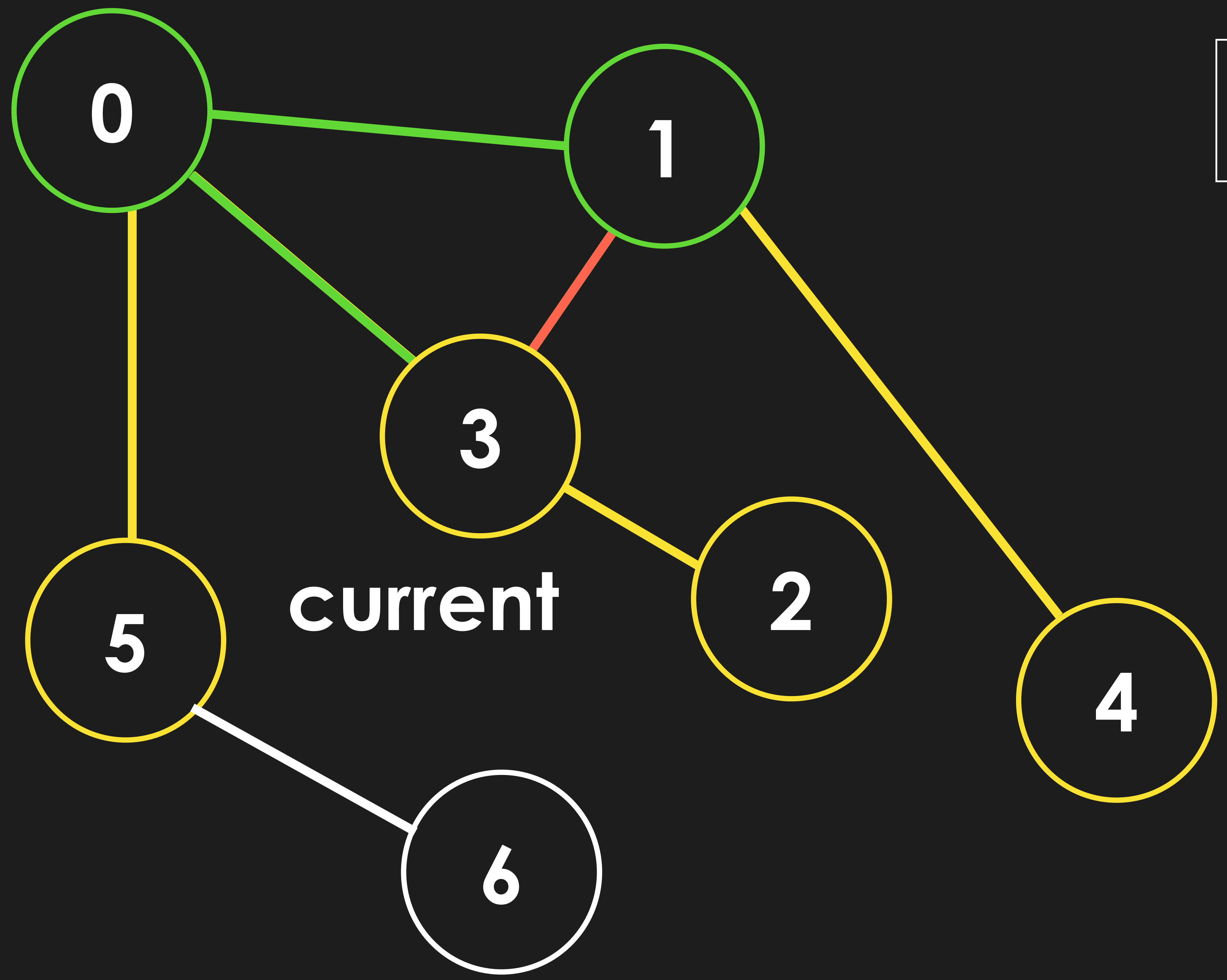
3

Queue
5
4

	visited
0	True
1	True
2	
3	True
4	True
5	True
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



3

Queue

5
4
2

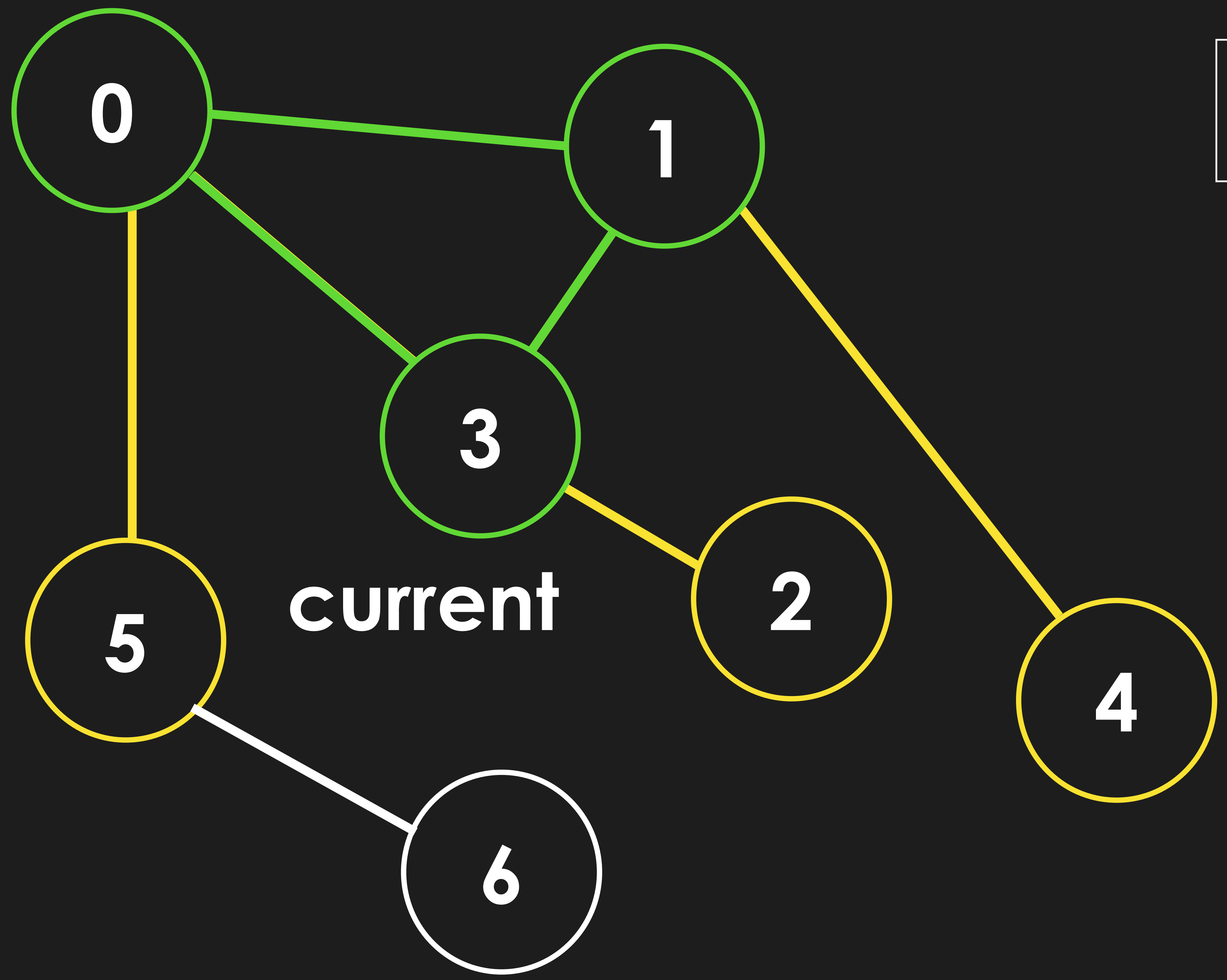
visited

0	True
1	True
2	True
3	True
4	True
5	True
6	

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges



3

Queue

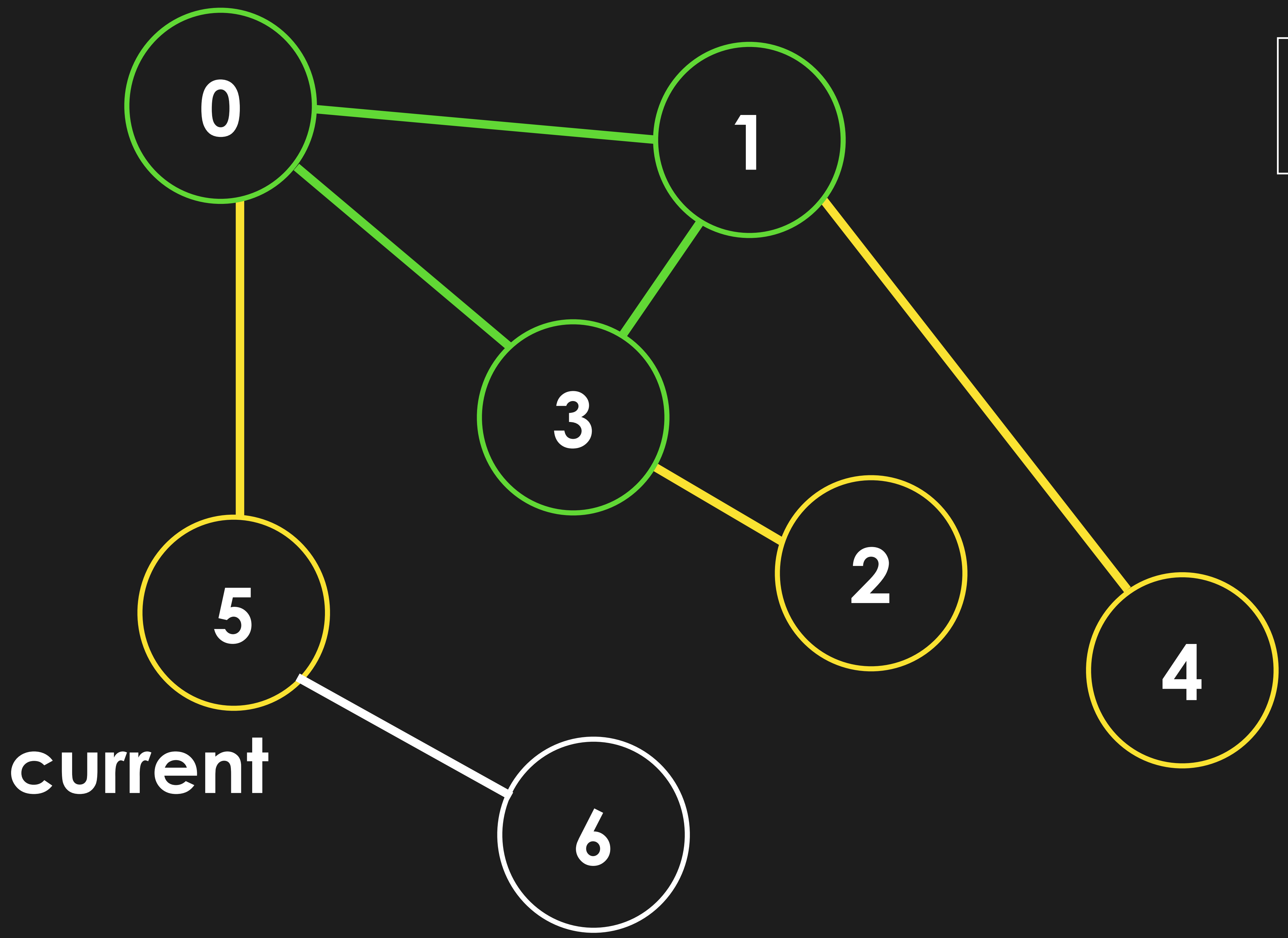
5
4
2

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



5

Queue

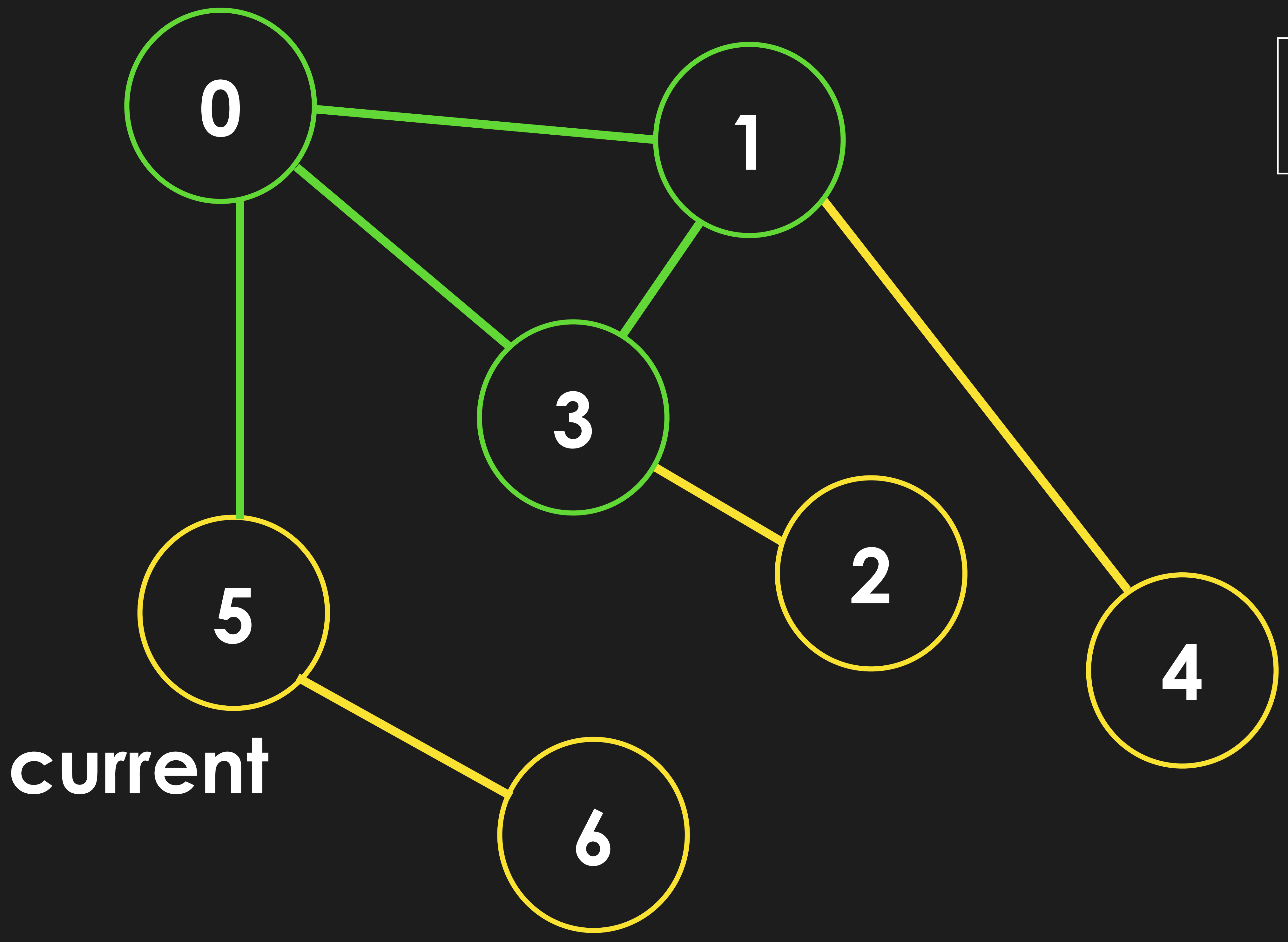
4
2

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	

bfs(graph, 0)

Dequeue first item, and visit it's edges



5

Queue

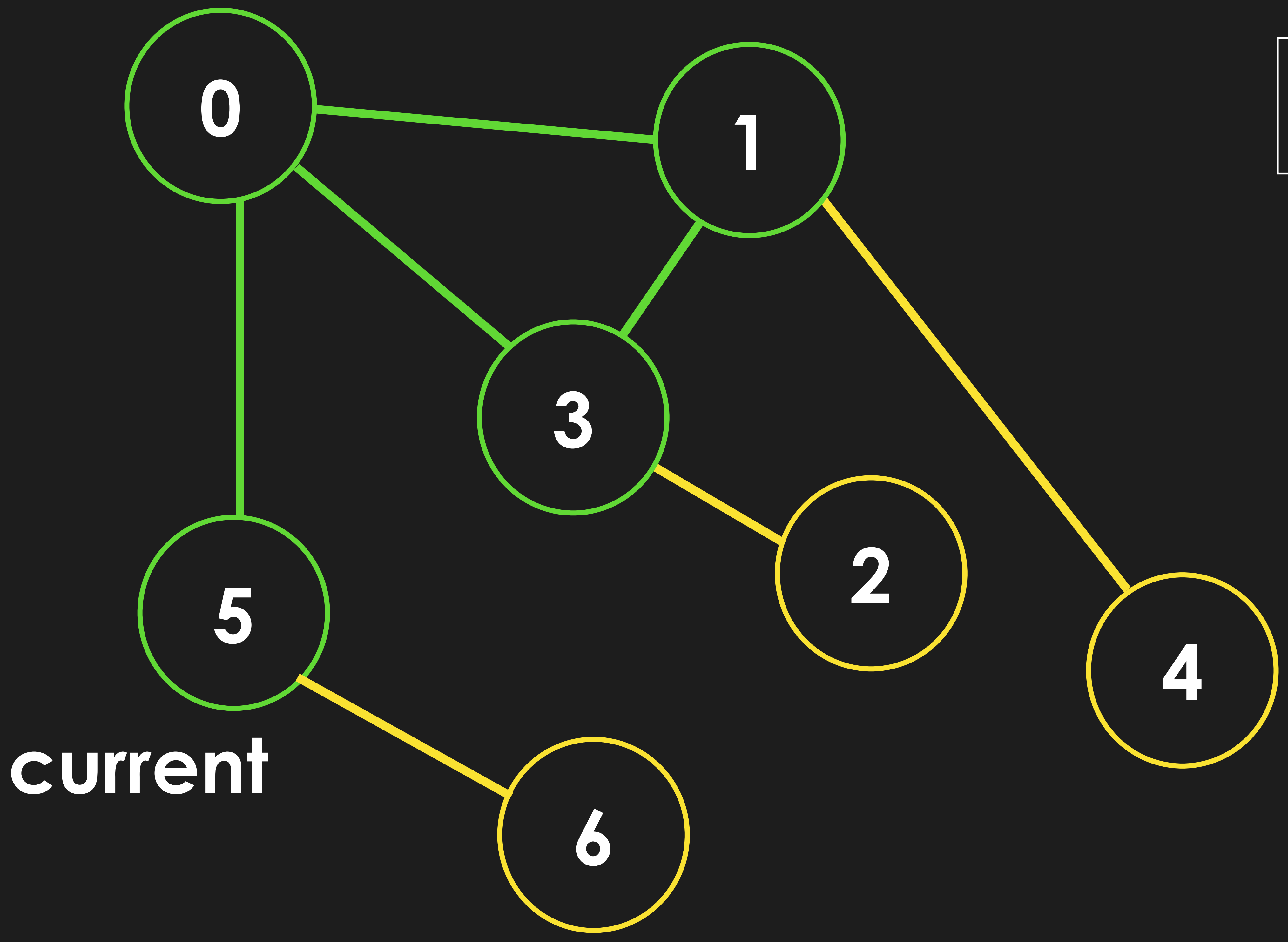
4
2
6

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



5

Queue

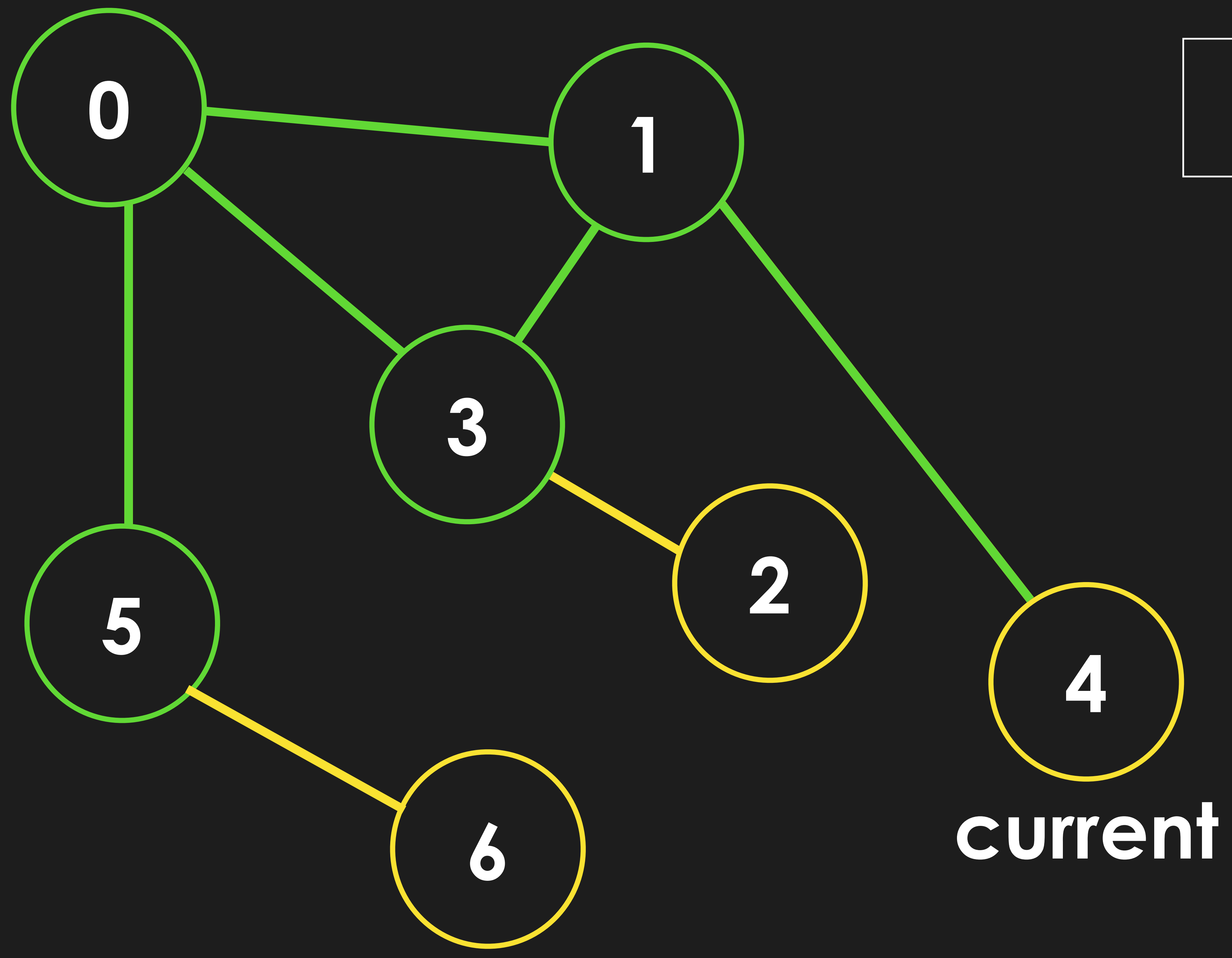
4
2
6

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



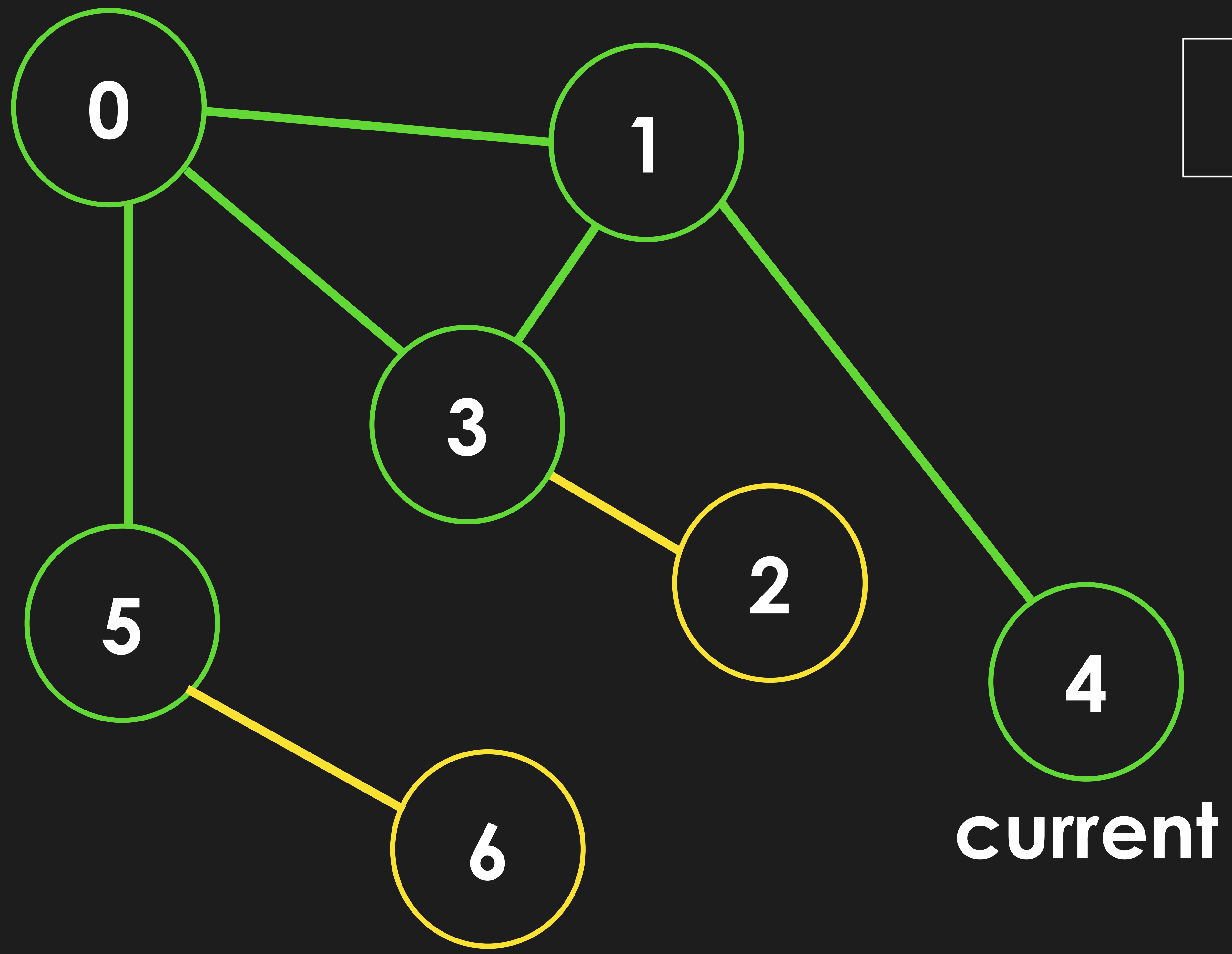
4

Queue
2
6

	visited
0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



4

Queue

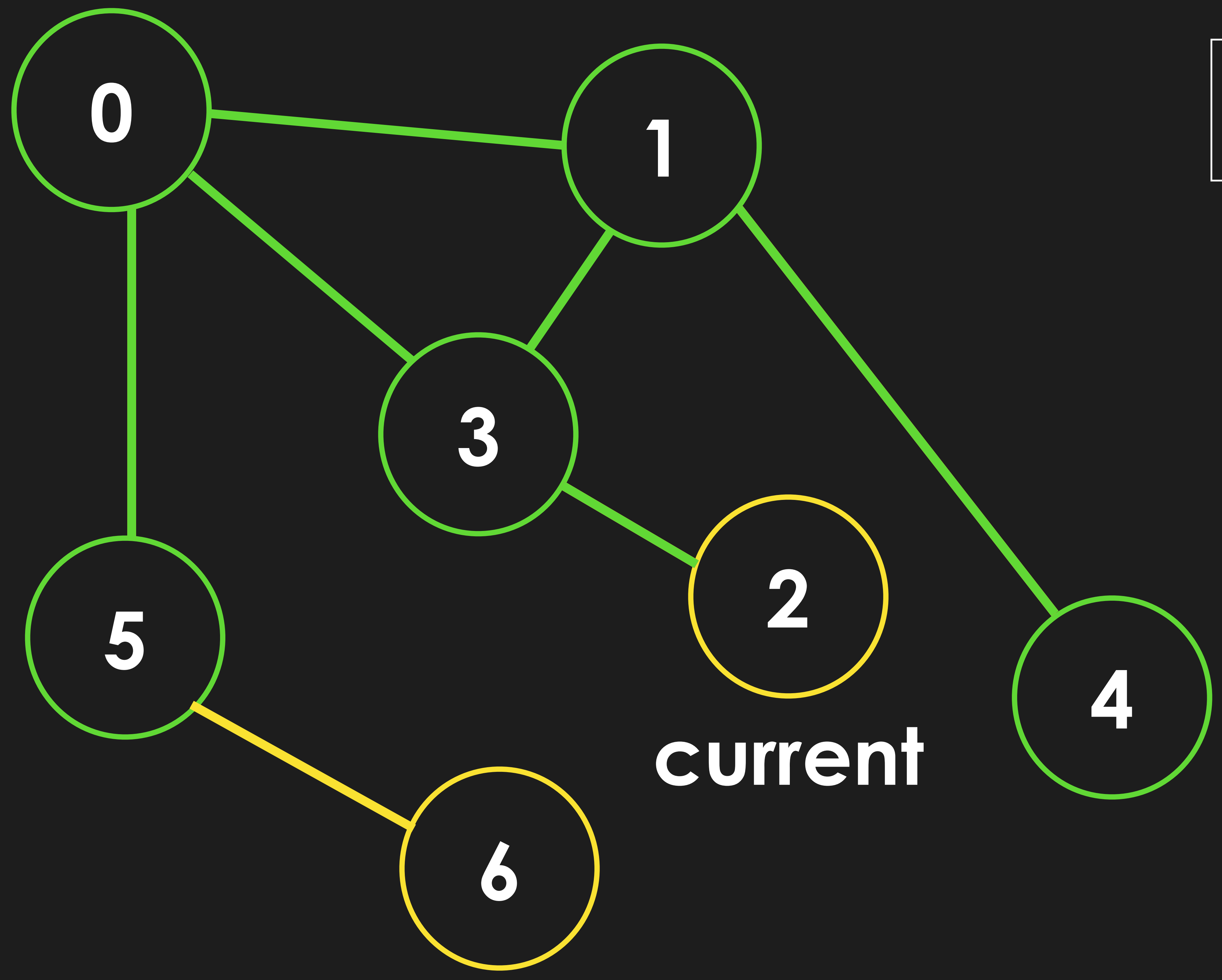
2
6

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



2

Queue

6

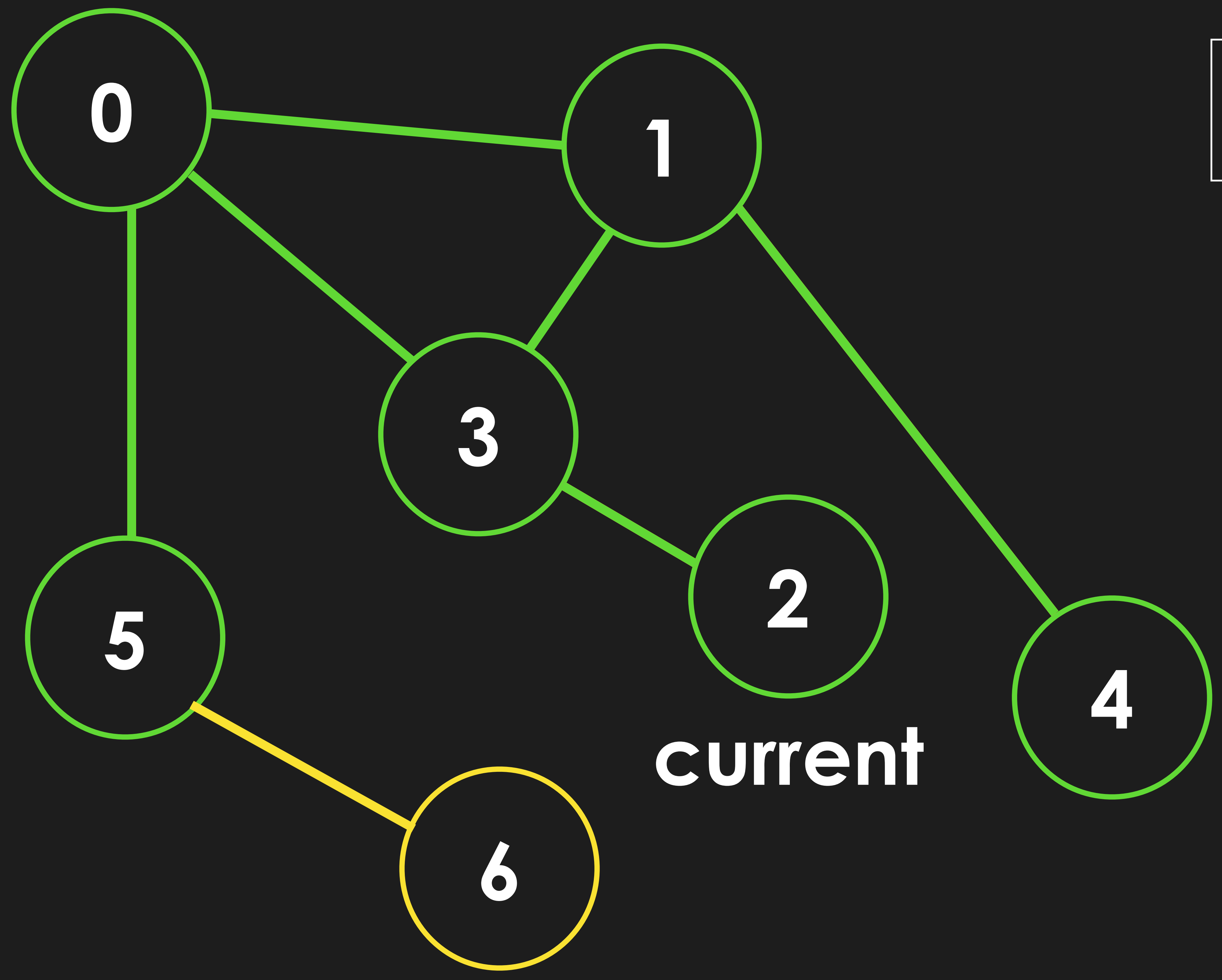
visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

CodeIT

bfs(graph, 0)

Dequeue first item, and visit it's edges



2

Queue

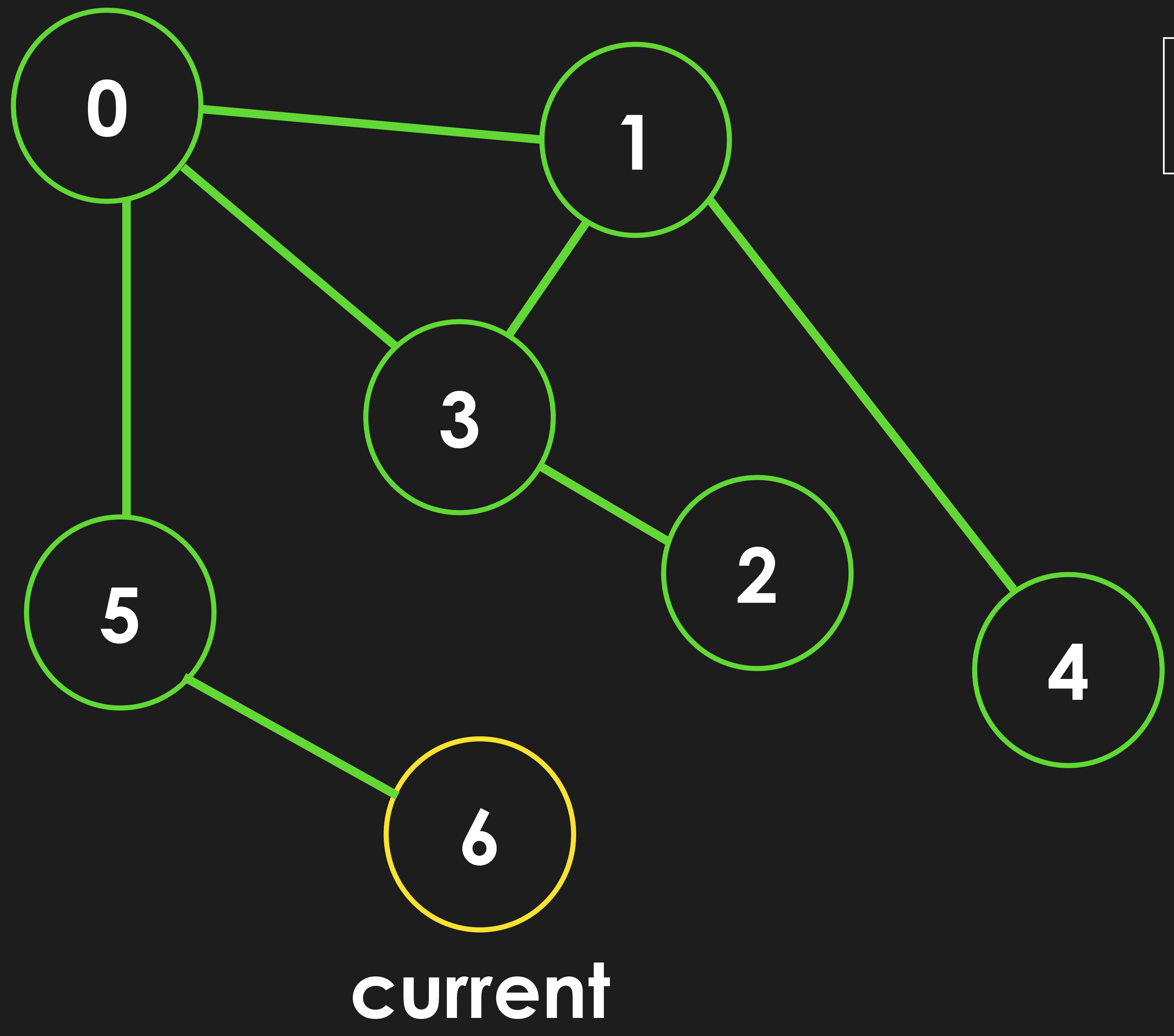
6

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



6

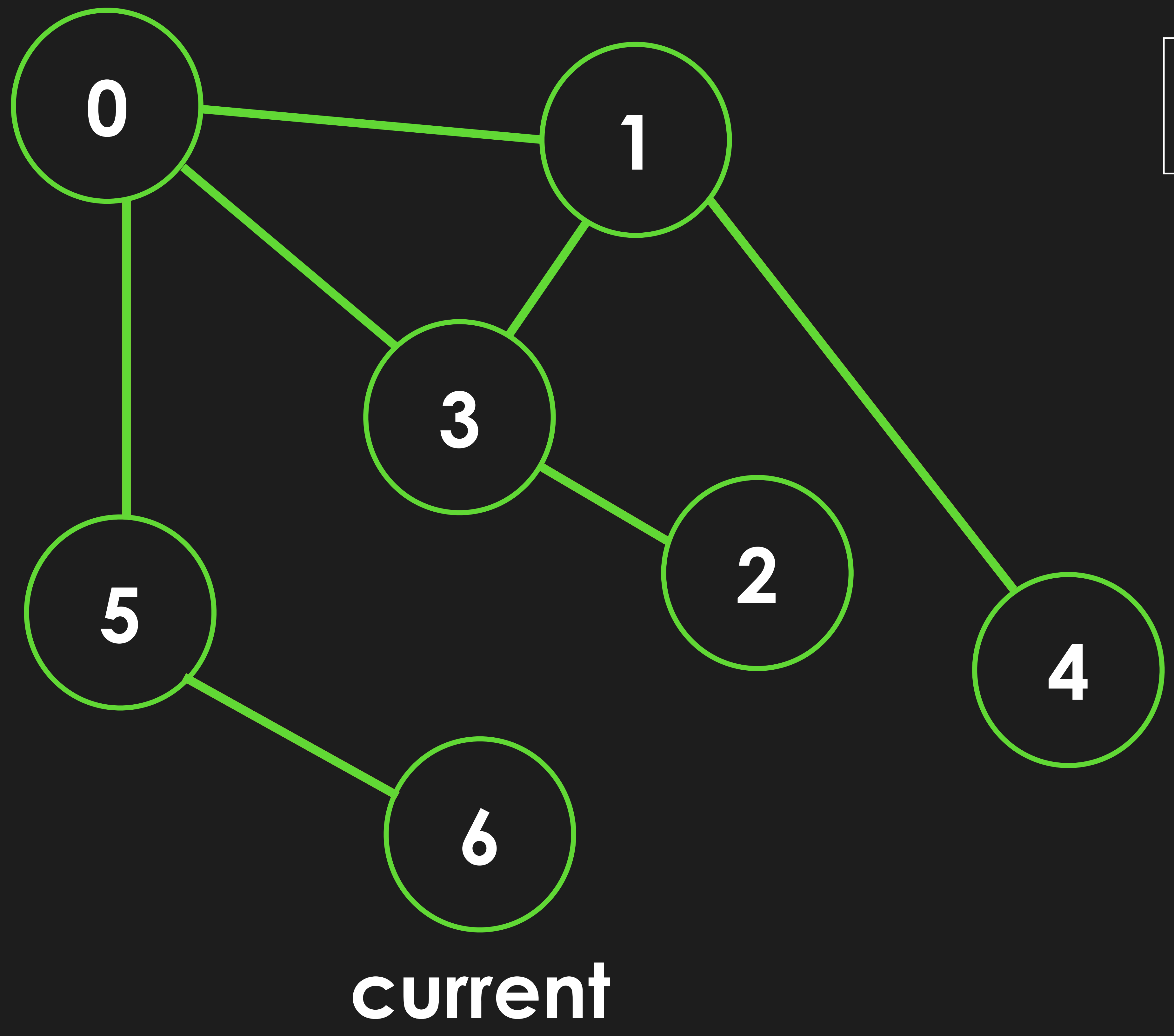
Queue

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

bfs(graph, 0)

Dequeue first item, and visit it's edges



6

Queue

visited

0	True
1	True
2	True
3	True
4	True
5	True
6	True

Implementation of BFS

bfs

```python
def bfs(graph, start):
    '''
        enqueue: queue.append()
        dequeue: queue.pop(0)
    '''

    visited = [False] * len(graph.adjList)
    queue = []
    queue.append(start)
    visited[start] = True

    while len(queue) != 0:
        v = queue.pop(0)
        print("visiting vertex {}".format(v))

        for dest in graph.adjList[v]:
            if not visited[dest]:
                visited[dest] = True
                queue.append(dest)
```

bfs demo

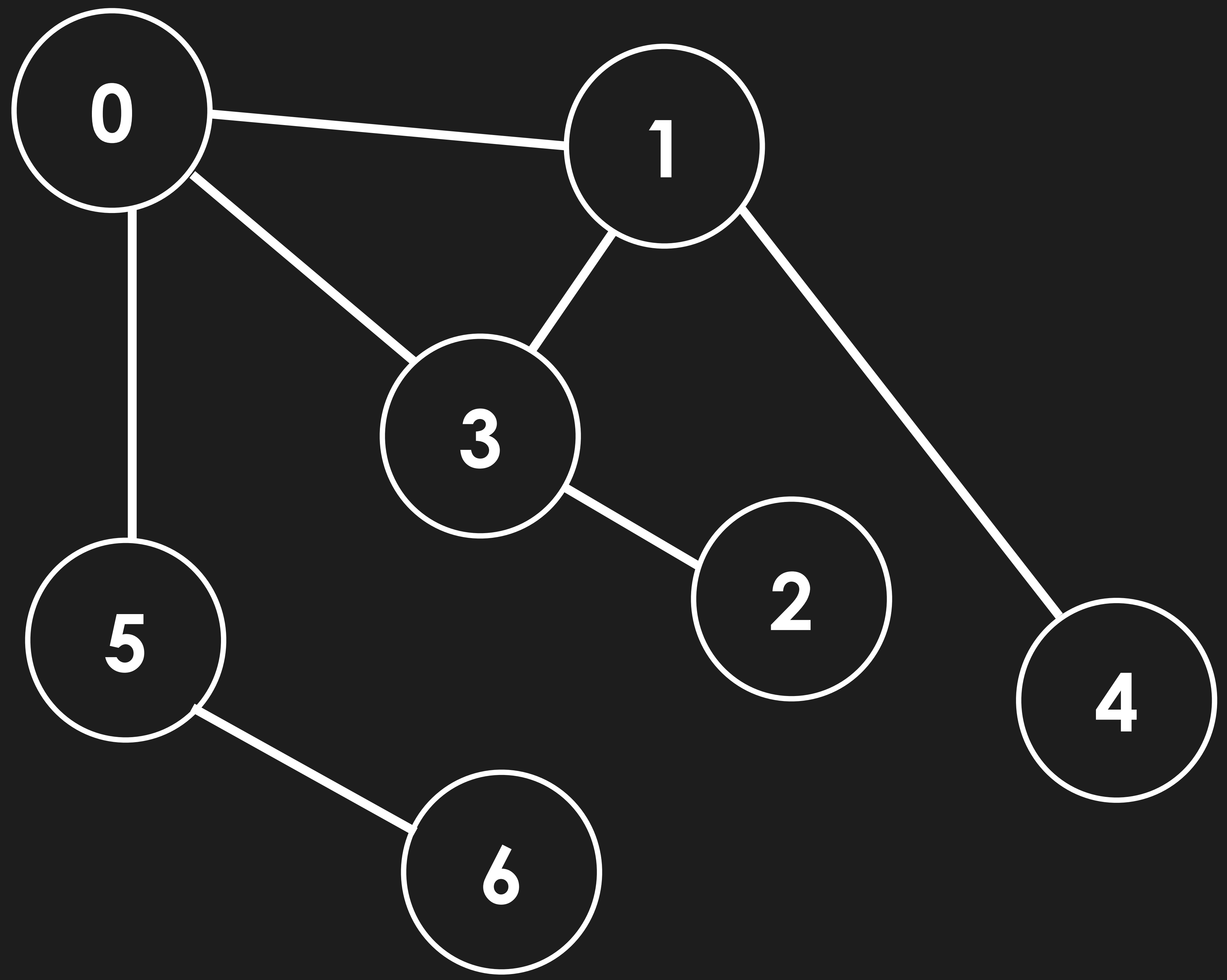
```
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 4), (1, 3), (3, 2), (5, 6)]
for edge in edges:
    graph.addEdge(edge)

bfs(graph, 0)
```

bfs demo

```
V = 7
graph = Graph(V)

edges = [(0, 1), (0, 3), (0, 5), (1, 4), (1, 3), (3, 2), (5, 6)]
for edge in edges:
    graph.addEdge(edge)

bfs(graph, 0)
```

```
visiting vertex 0
visiting vertex 1
visiting vertex 3
visiting vertex 5
visiting vertex 4
visiting vertex 2
visiting vertex 6
```

bfs demo




```
visiting vertex 0
visiting vertex 1
visiting vertex 3
visiting vertex 5
visiting vertex 4
visiting vertex 2
visiting vertex 6
```

Application of BFS

Application of BFS:

1. **Hierachal Tree**: Visit each level
2. **Unweighted Graphs**: Find shortest path from start node to every other node

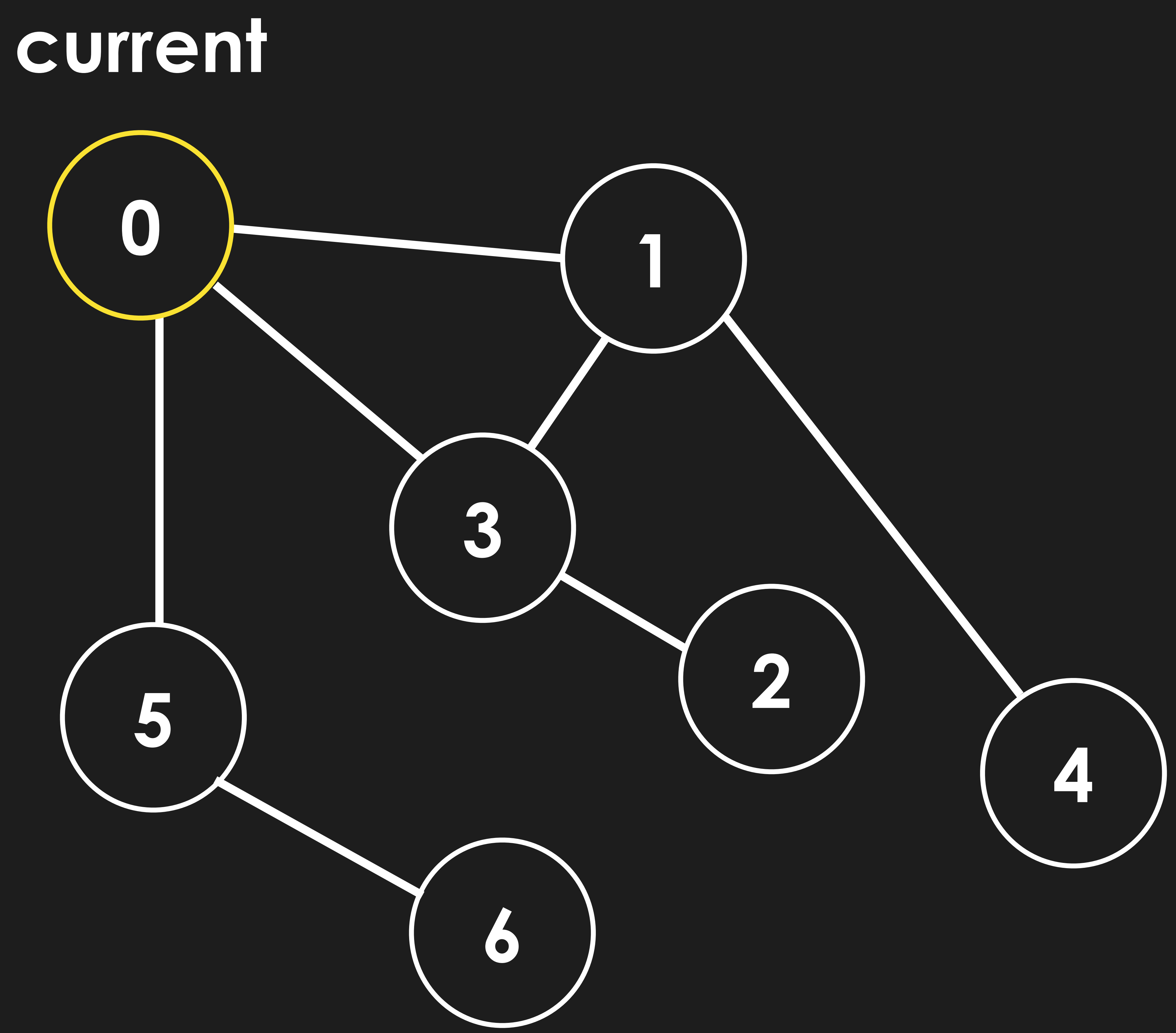
Pseudo code for unweighted graph shortest path:

1. Keep an array of **shortest distance to** each node and array of **edge to** each node (where the node was visited from)

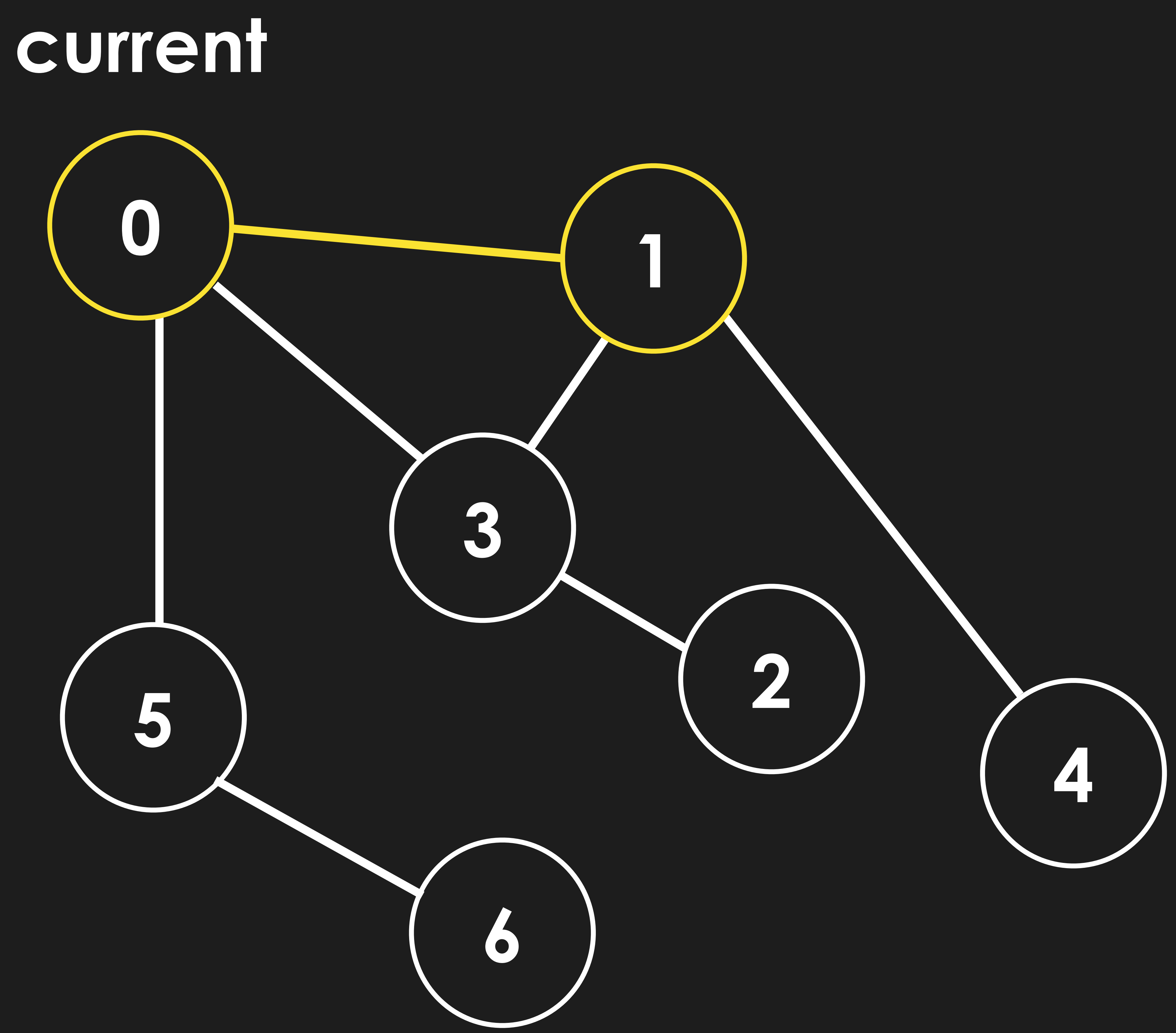
2. Every time you visit a node, shortest distance to that node is the **shortest distance to prev node + 1**

3. To construct shortest path, backtrack using **edgeTo array**

Shortest Path

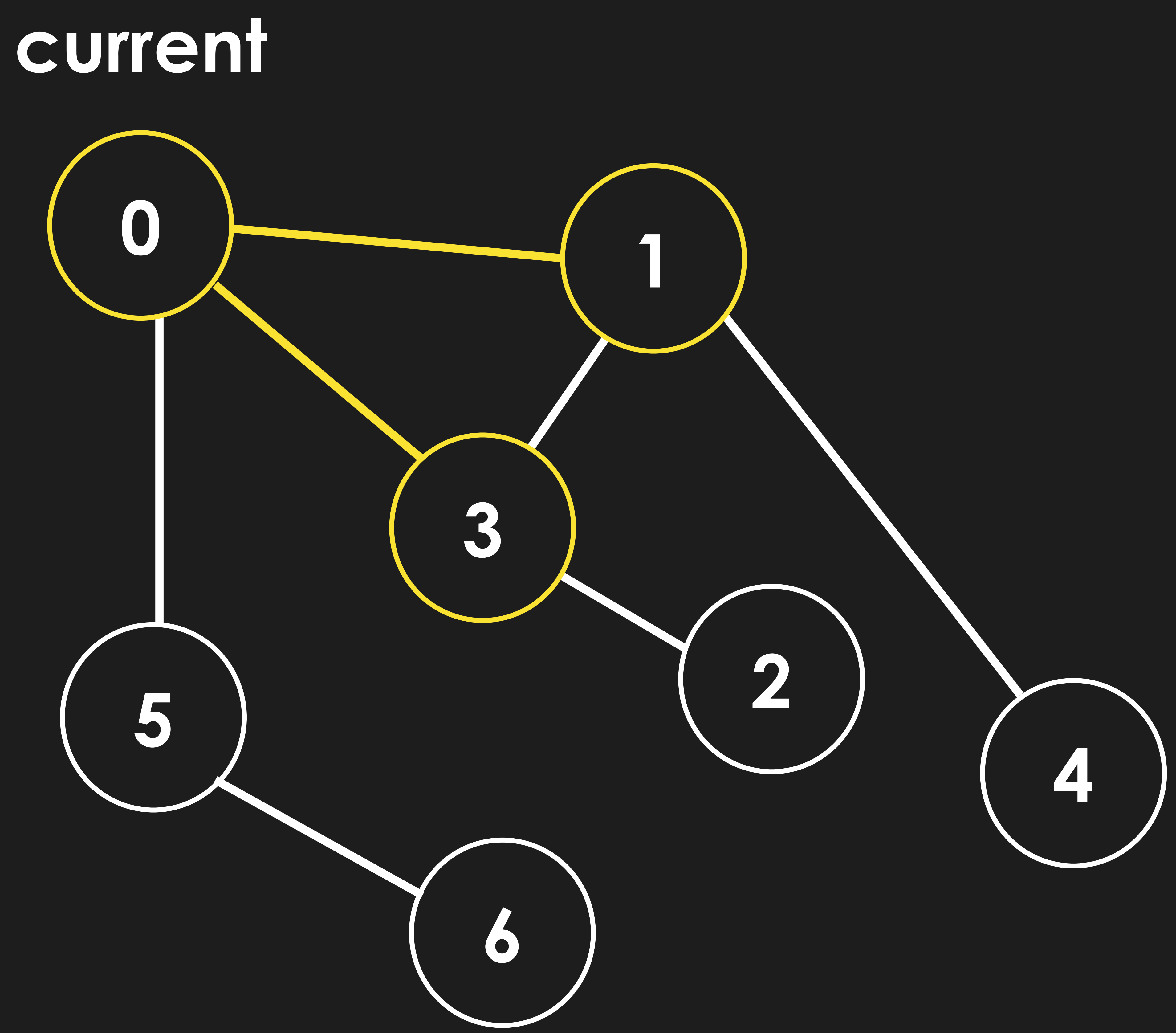


	visited	shortest dist	edgeTo
0	TRUE	0	-1
1			
2			
3			
4			
5			
6			



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3			
4			
5			
6			

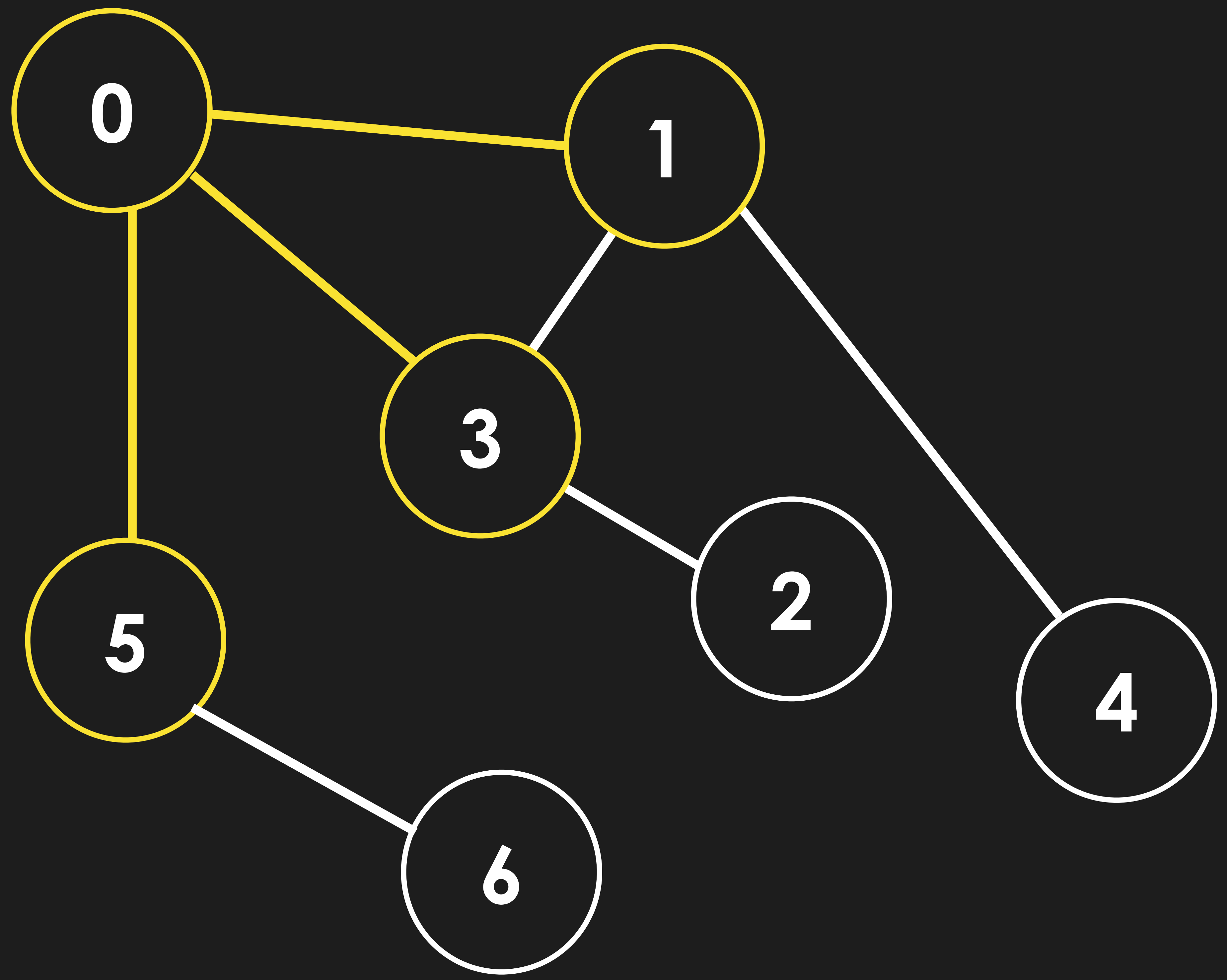
CodeIT



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4			
5			
6			

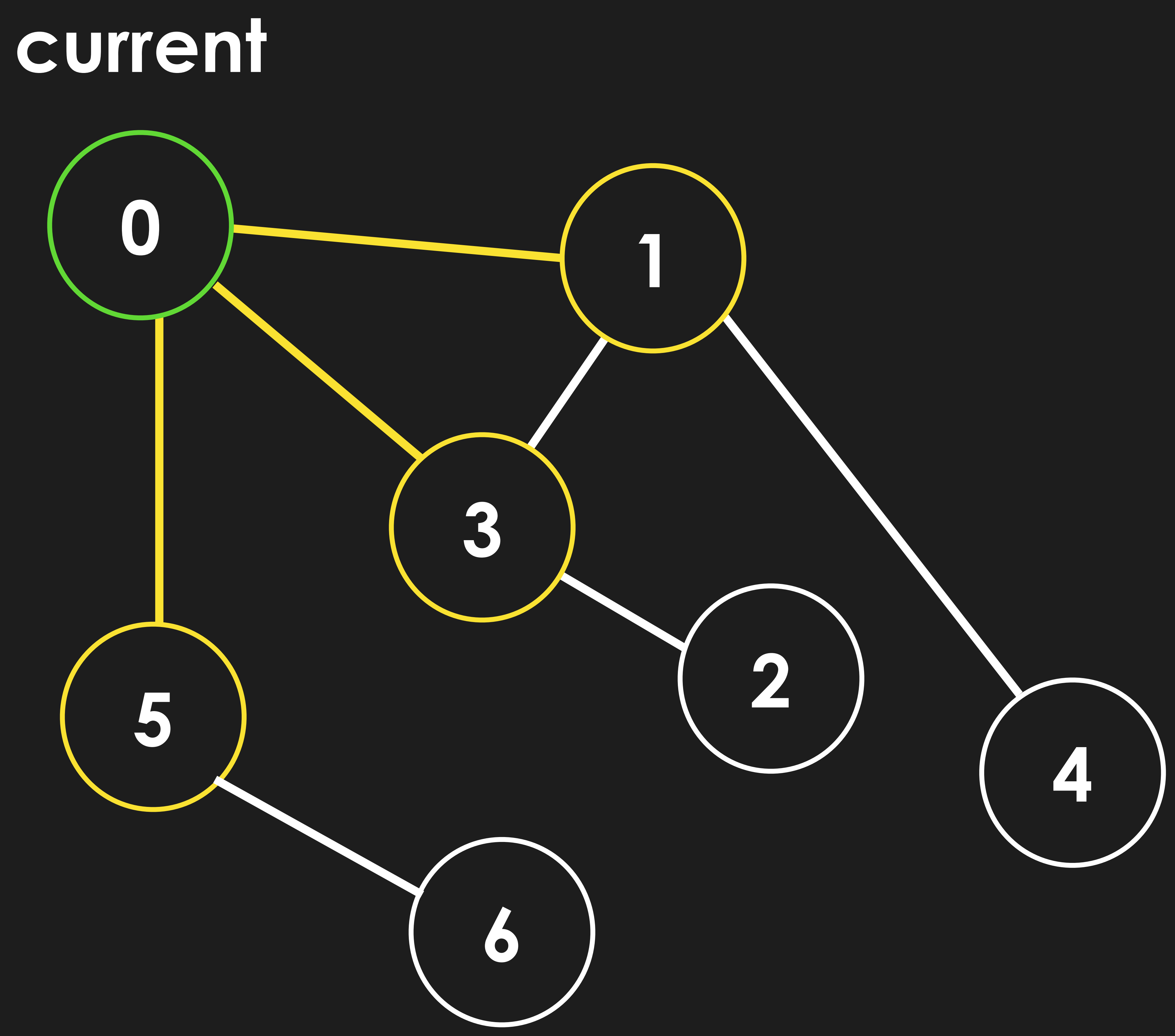
CodeIT

current

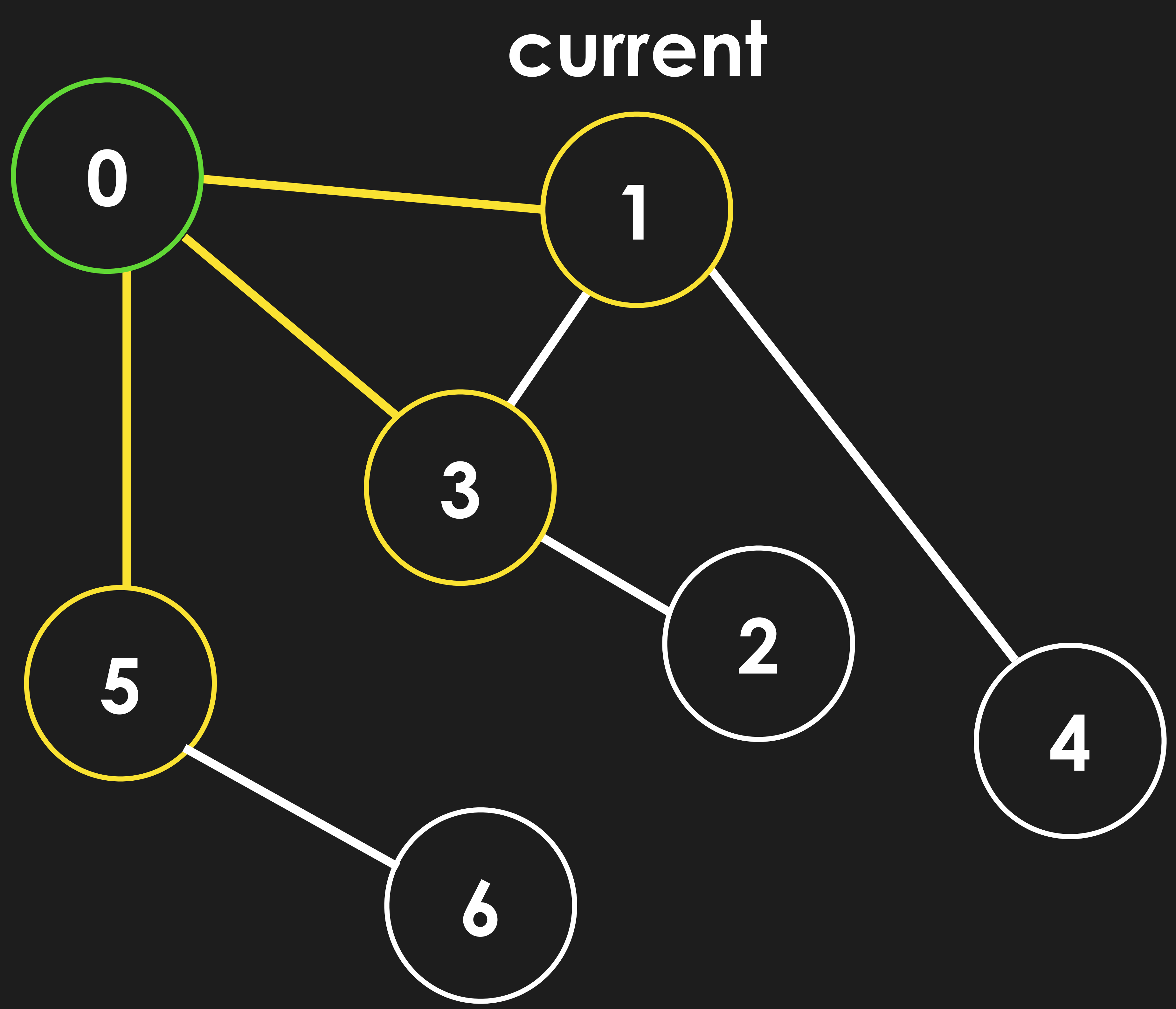


	visited		shortest dist		edgeTo
0	TRUE	0	0	0	-1
1	TRUE	1	1	1	0
2		2		2	
3	TRUE	3	1	3	0
4		4		4	
5	TRUE	5	1	5	0
6		6		6	

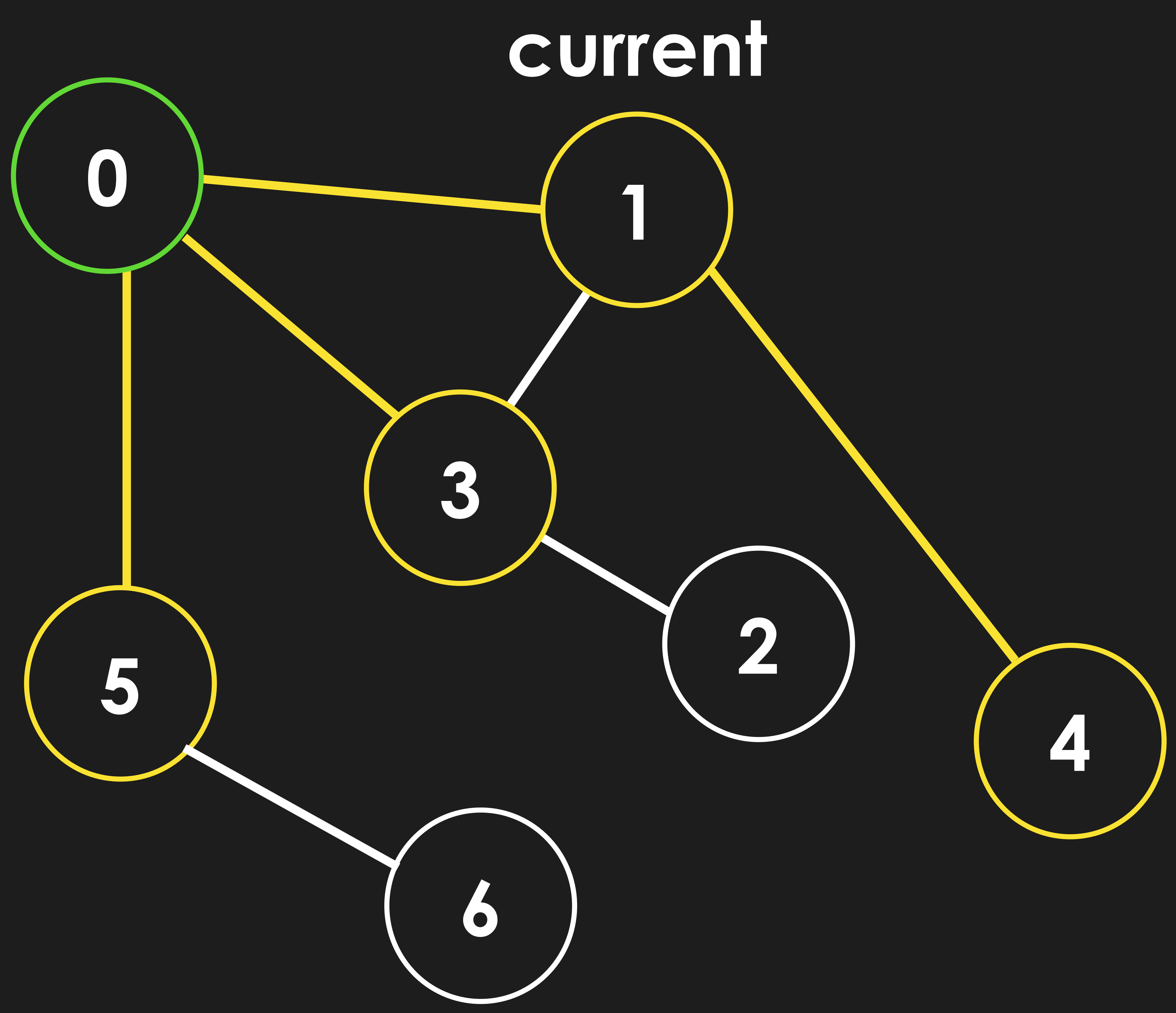
CodeIT



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4			
5	TRUE	1	0
6			



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4			
5	TRUE	1	0
6			



visited

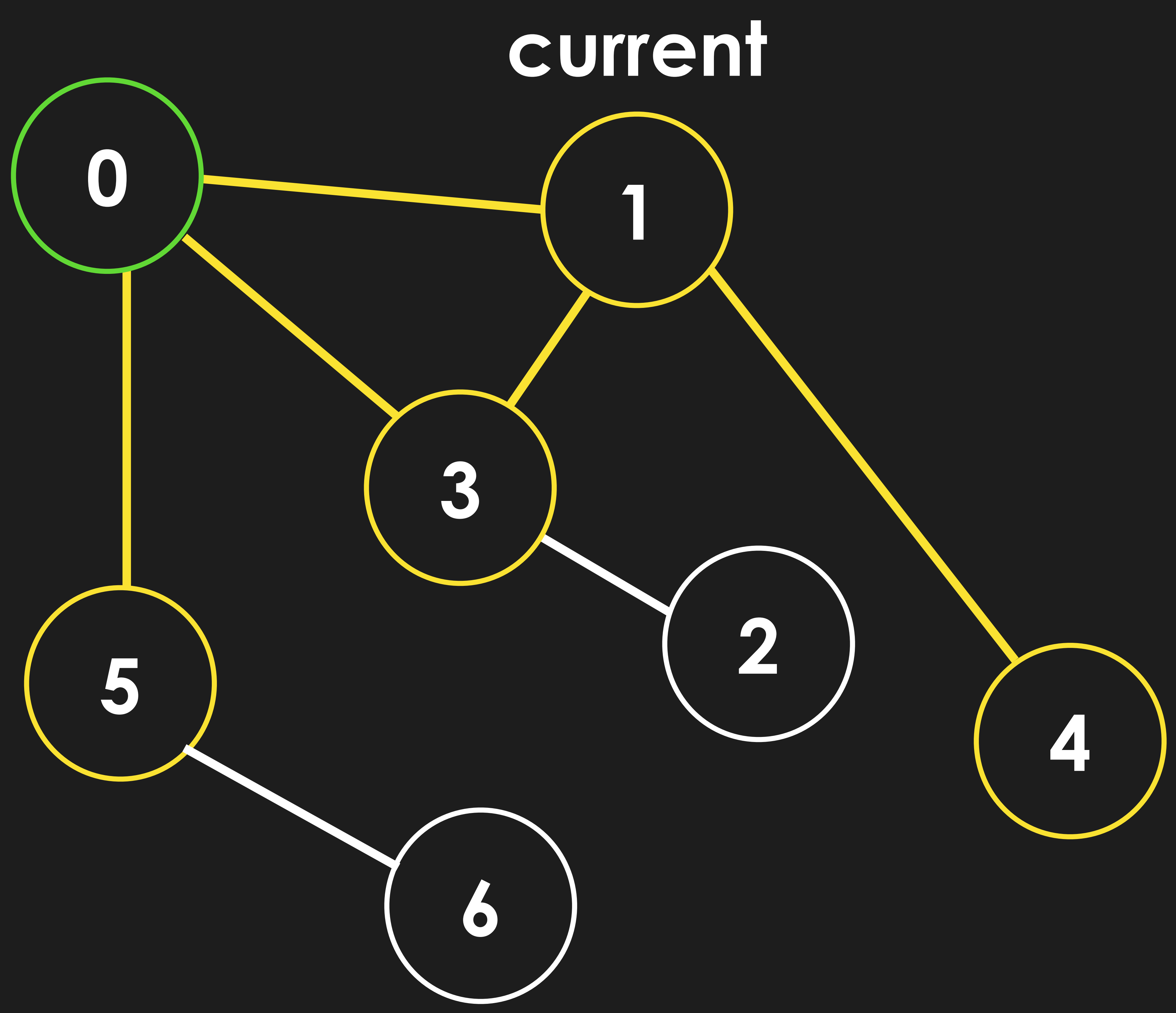
0	TRUE
1	TRUE
2	
3	TRUE
4	TRUE
5	TRUE
6	

shortest dist

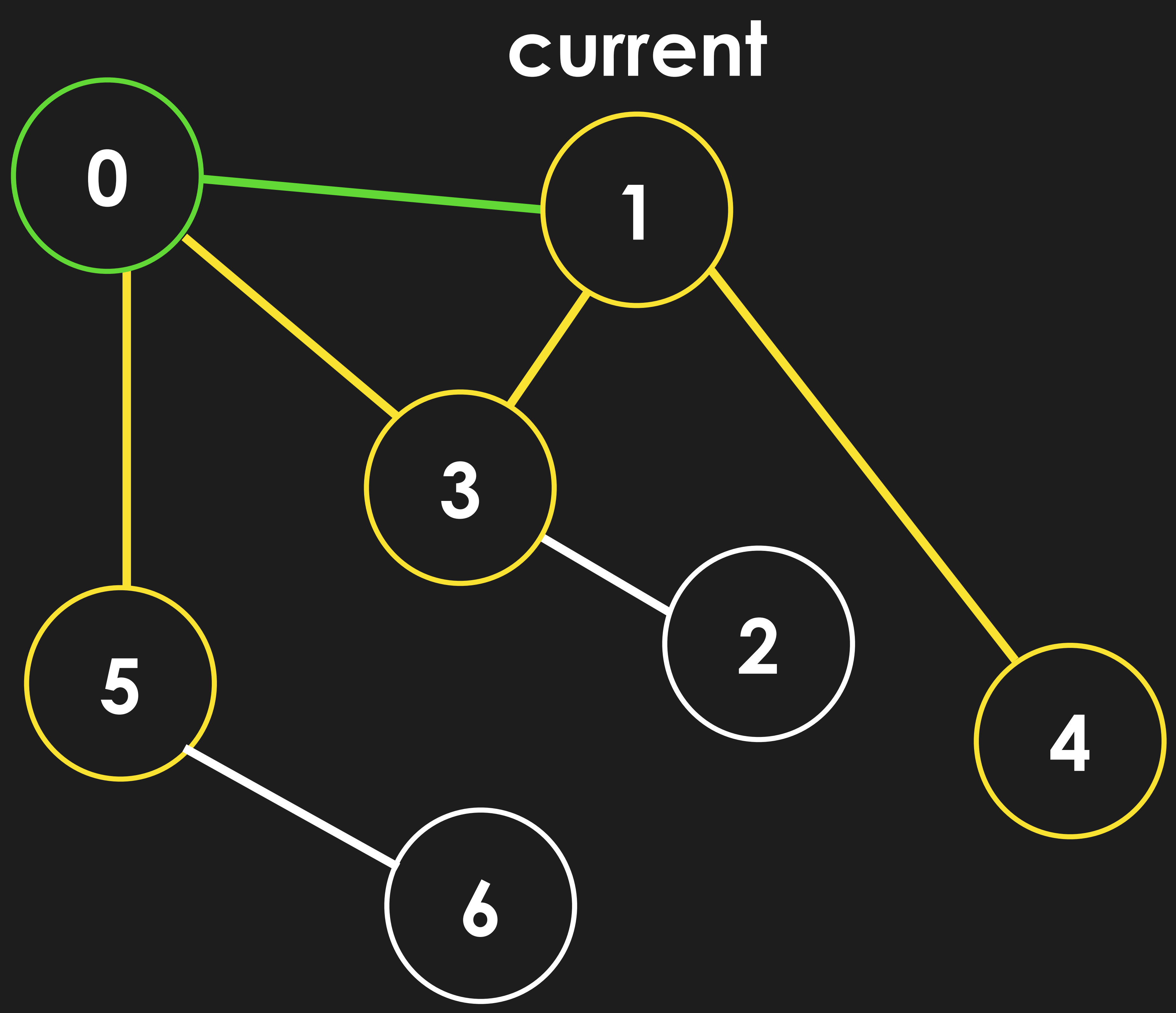
0	0
1	1
2	
3	1
4	2
5	1
6	

edgeTo

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



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



visited

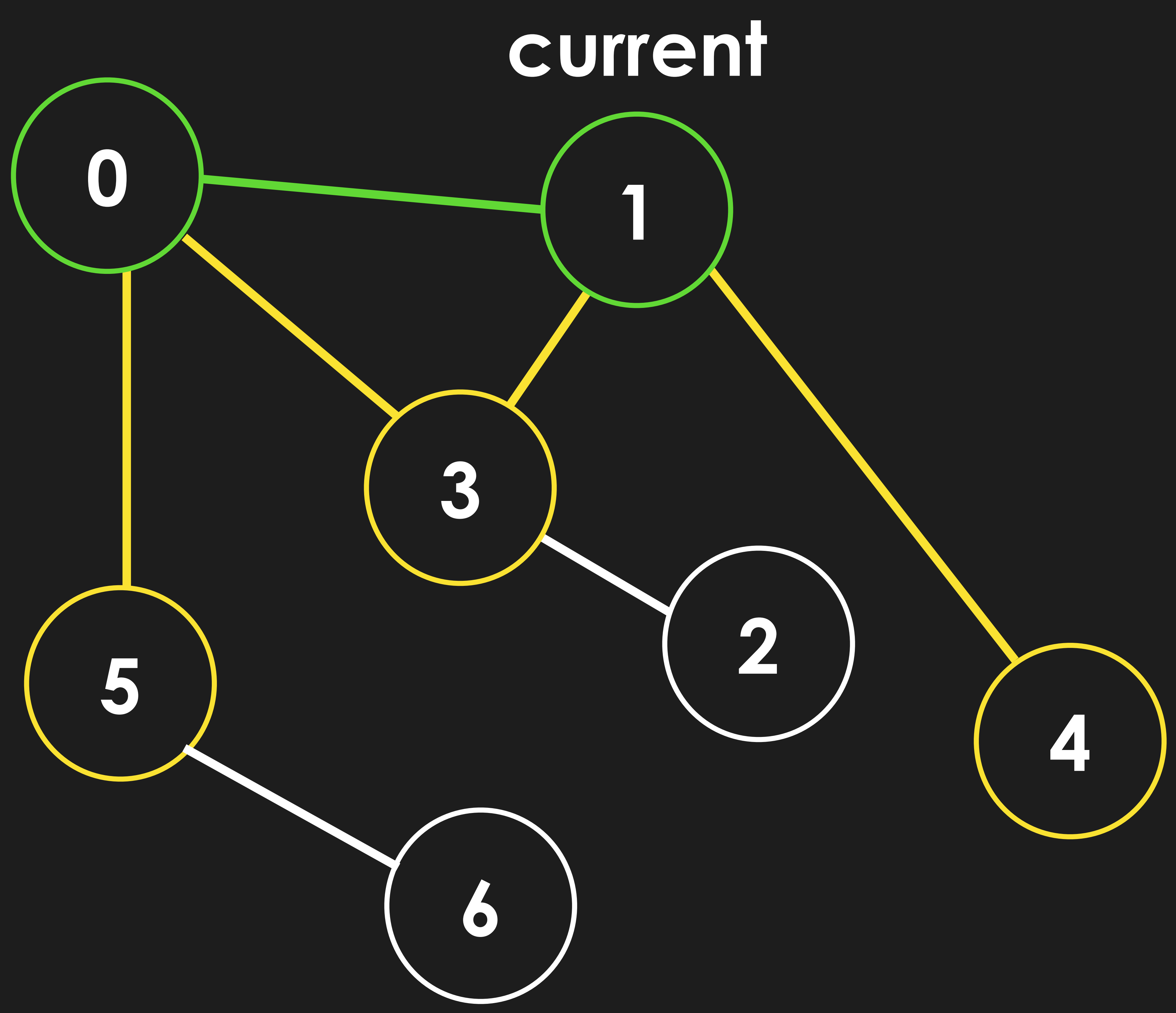
0	TRUE
1	TRUE
2	
3	TRUE
4	TRUE
5	TRUE
6	

shortest dist

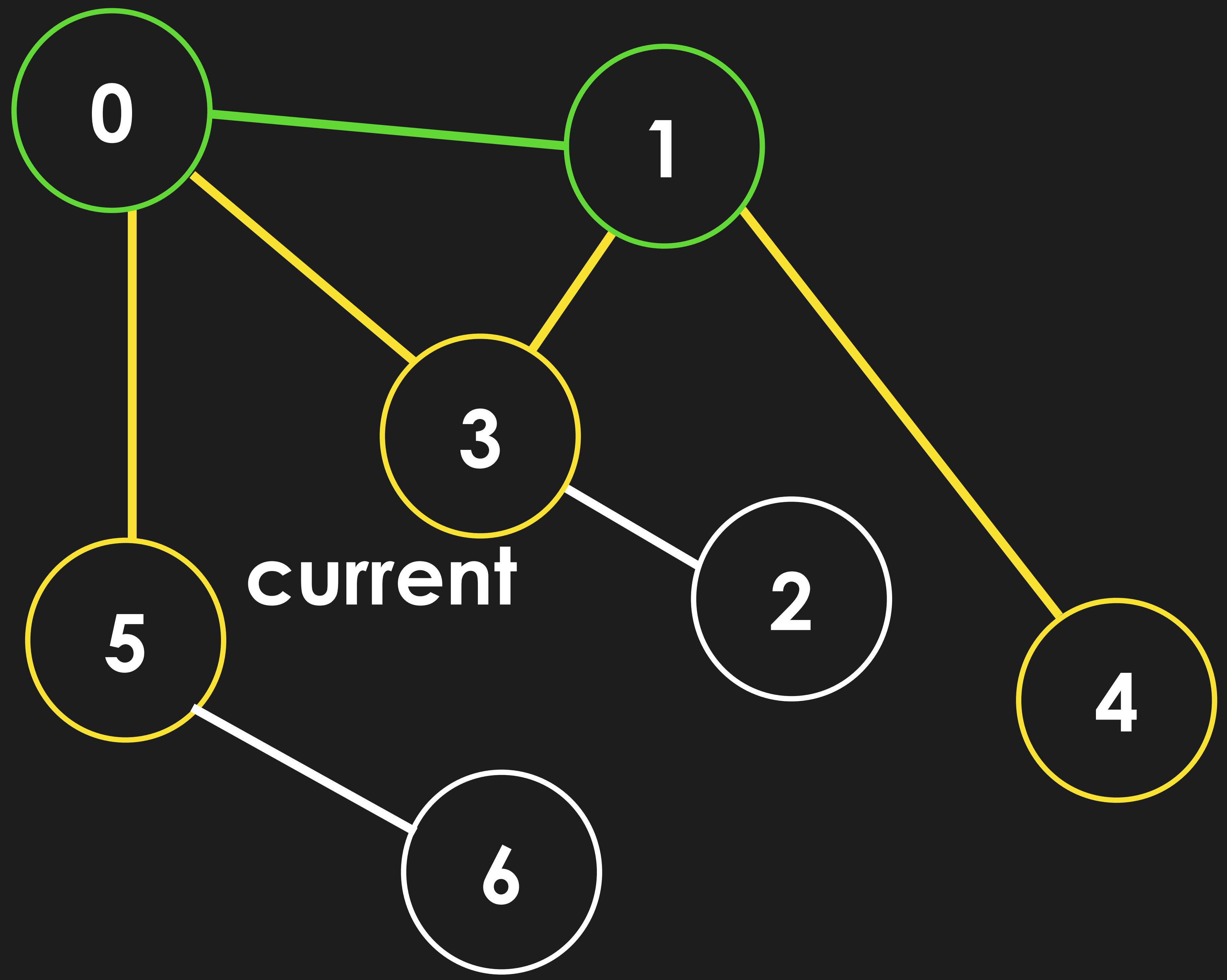
0	0
1	1
2	
3	1
4	2
5	1
6	

edgeTo

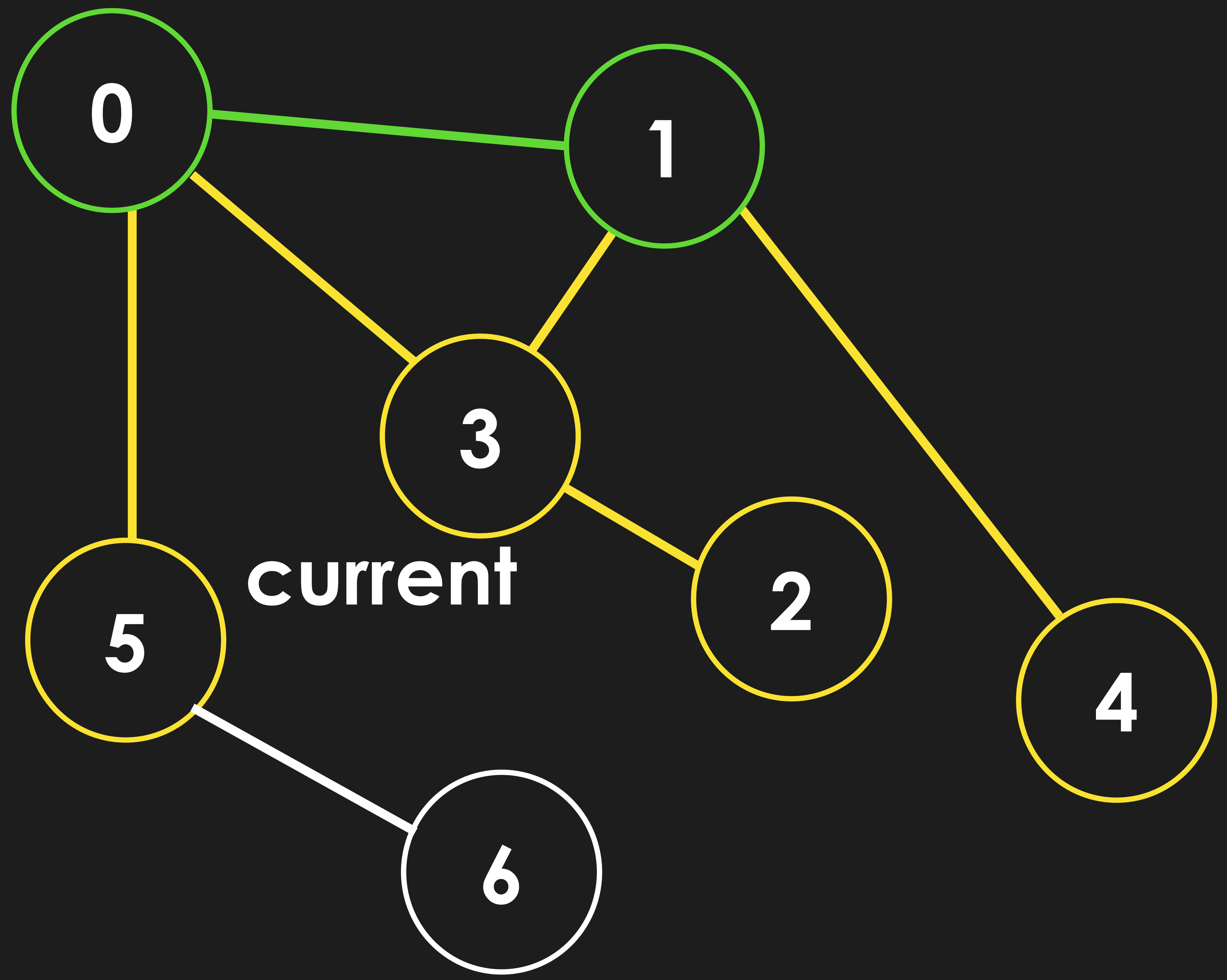
0	-1
1	0
2	
3	0
4	3
5	0
6	



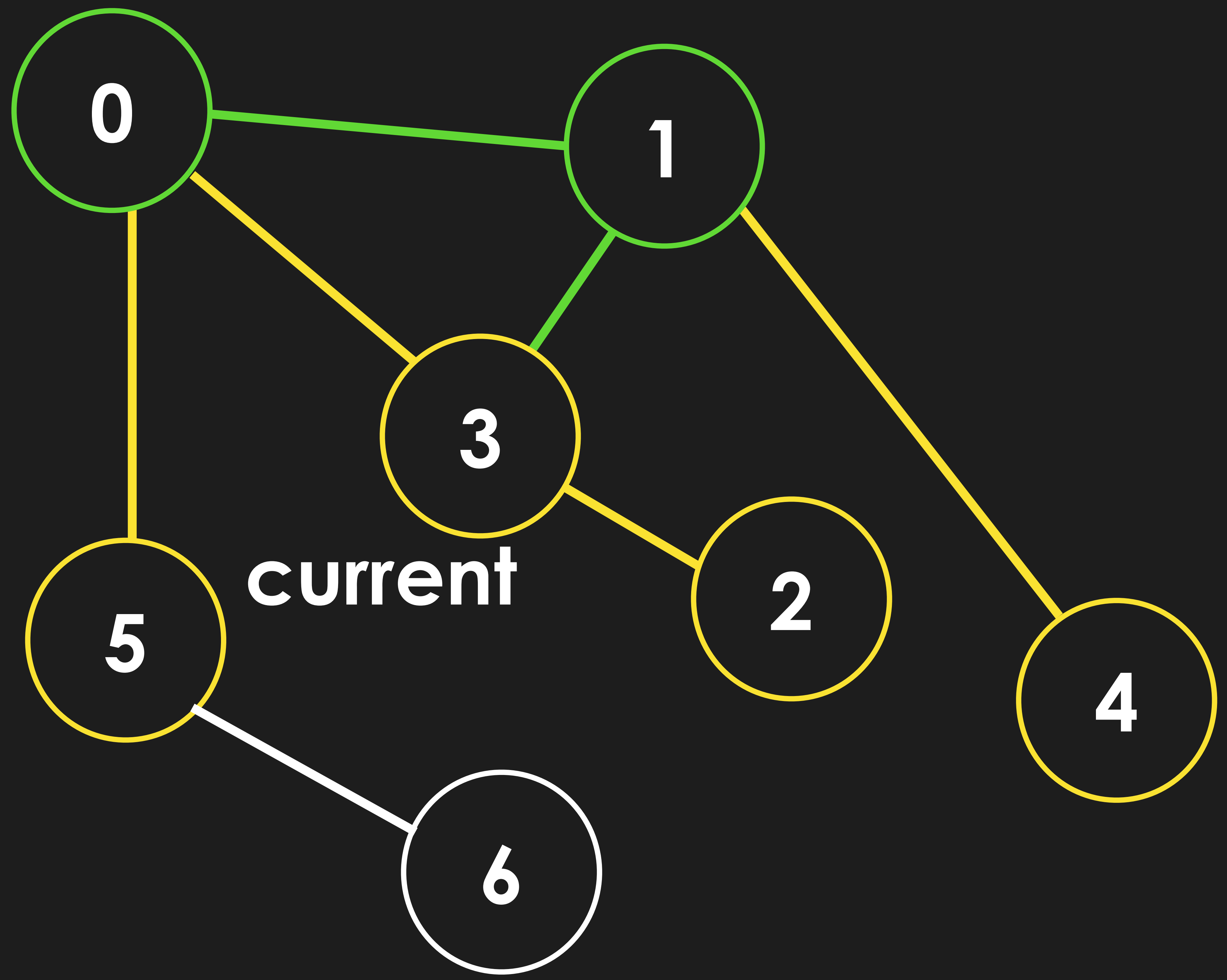
	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



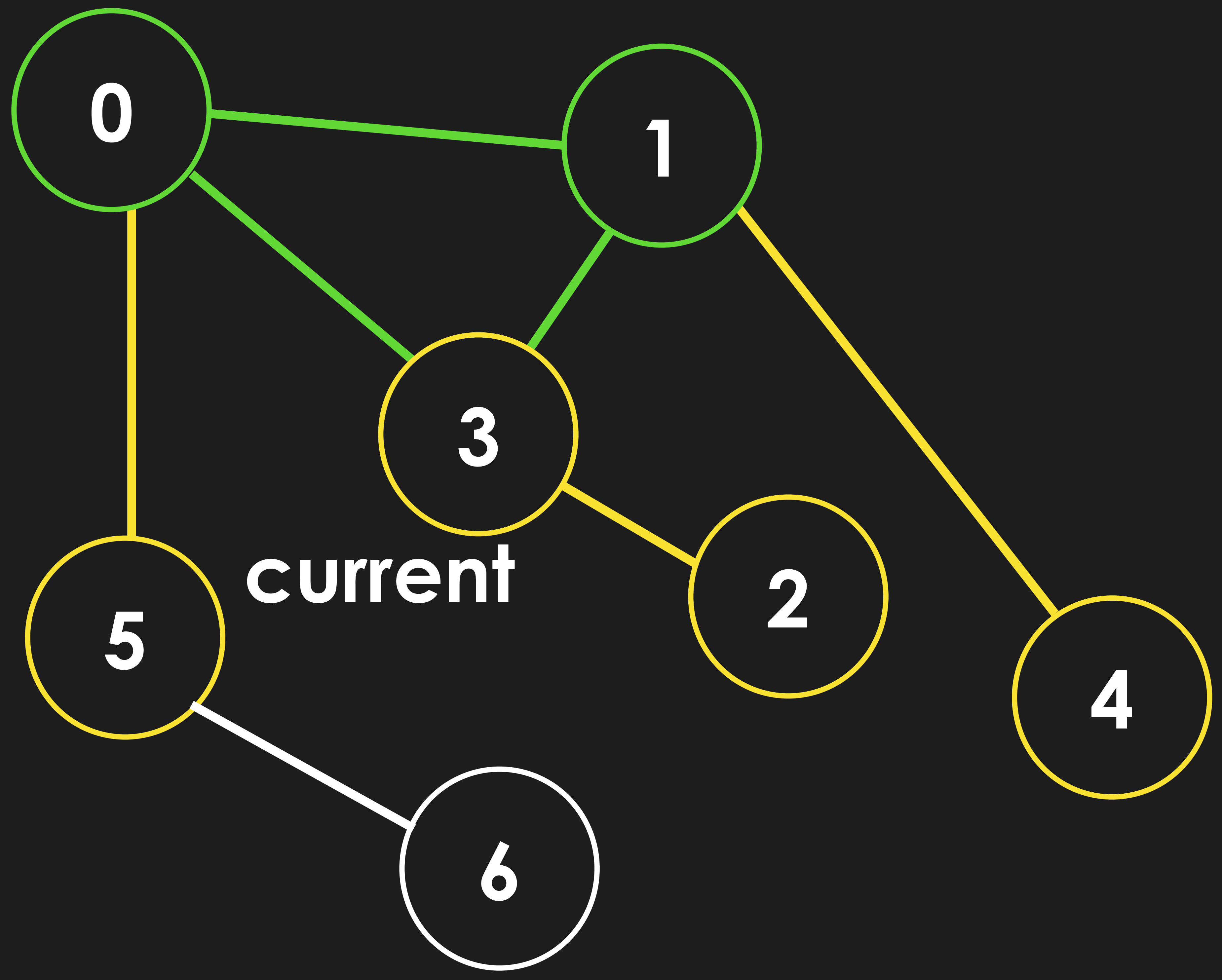
	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2			
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



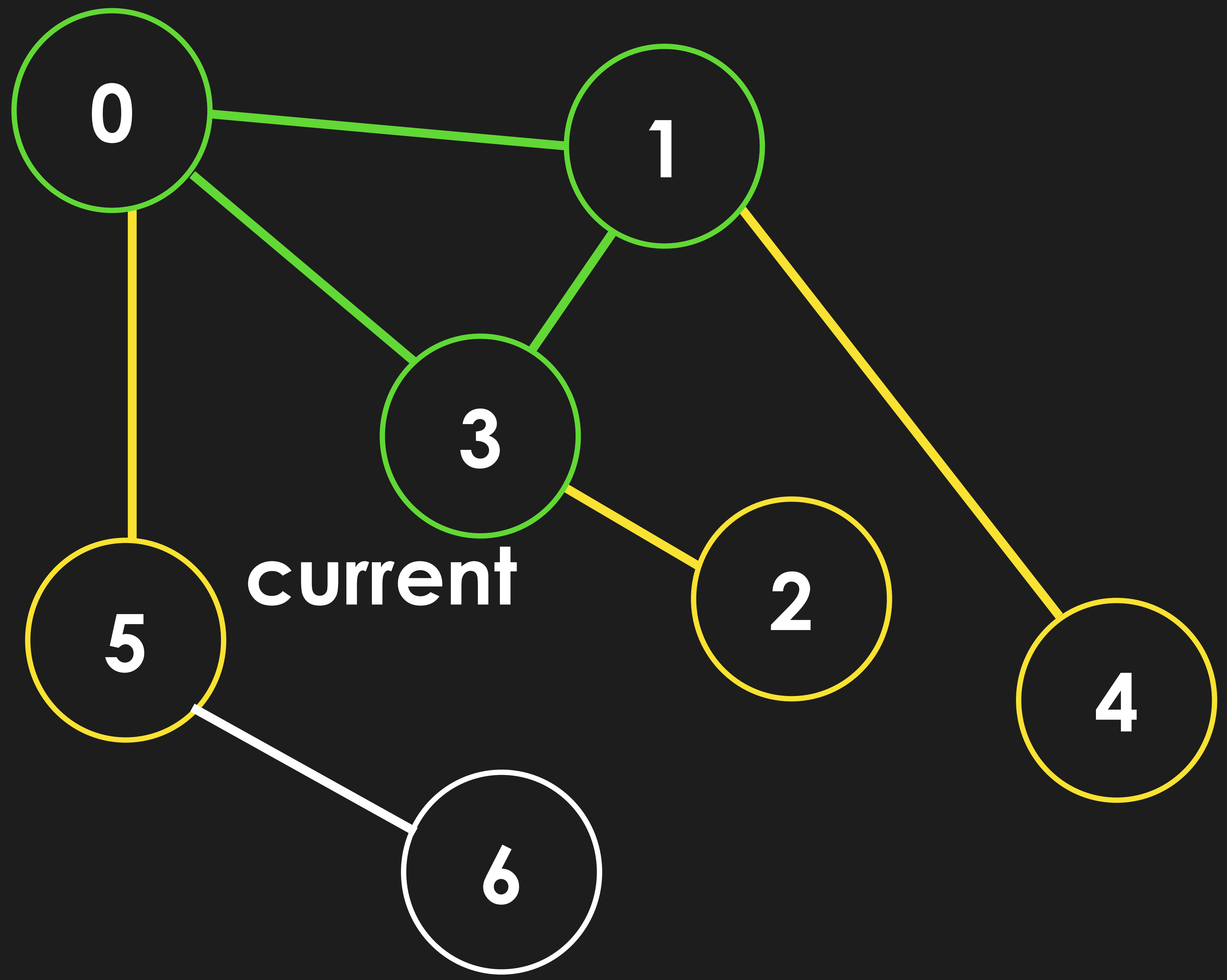
	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



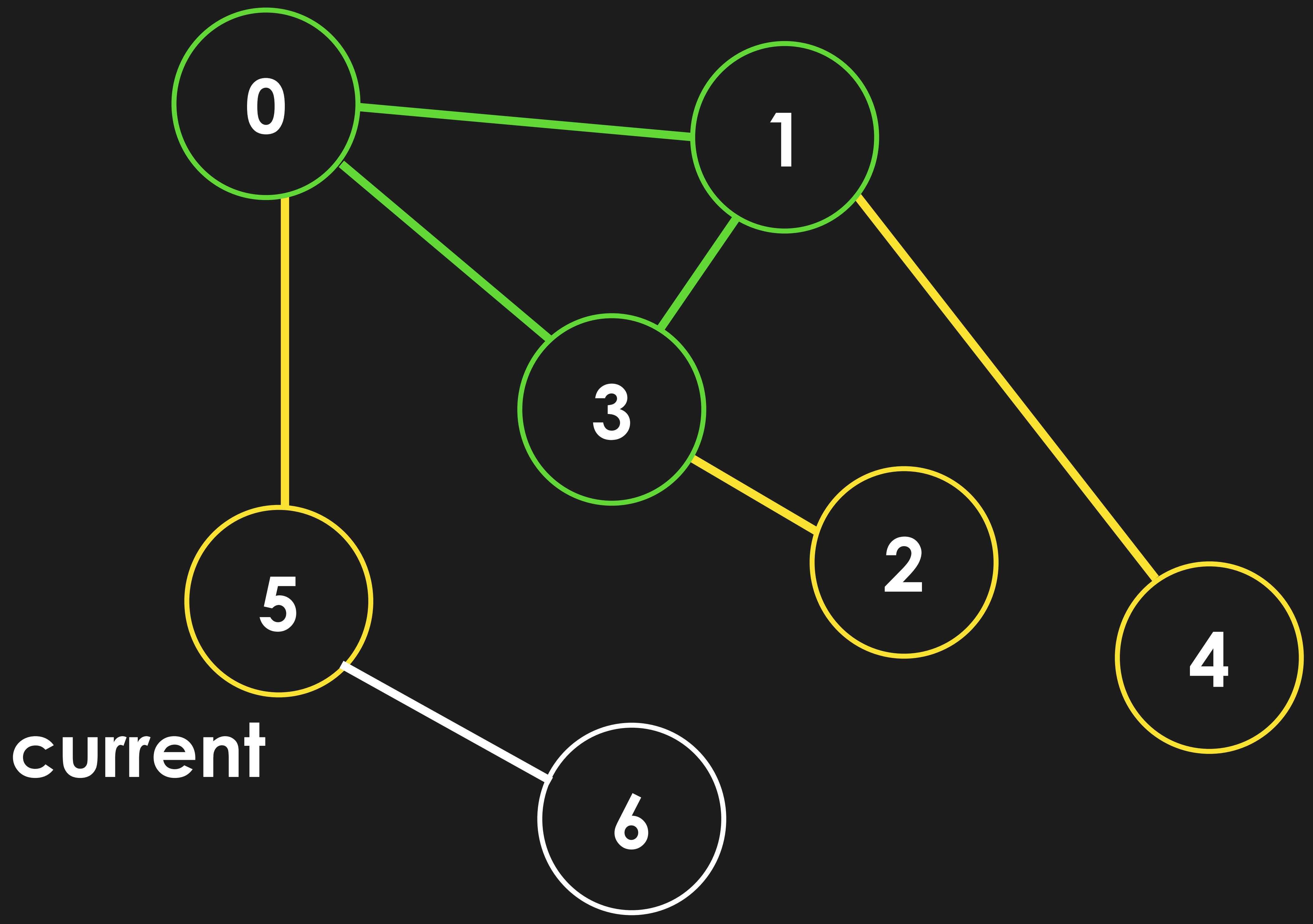
	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



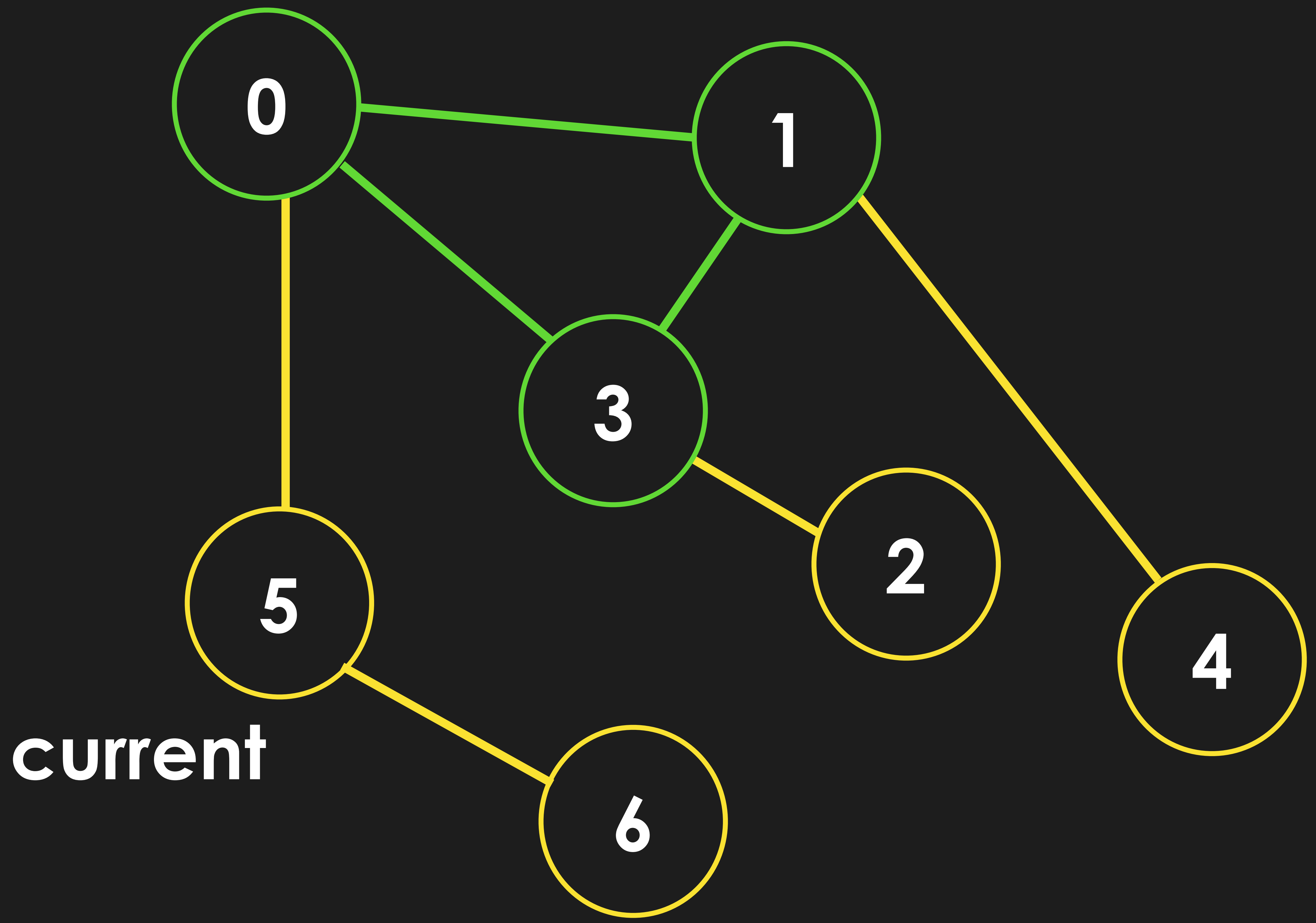
	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6			



	visited	shortest dist	edgeTo
0	TRUE	0	-1
1	TRUE	1	0
2	TRUE	2	3
3	TRUE	1	0
4	TRUE	2	3
5	TRUE	1	0
6	TRUE	2	5

Time complexity of DFS & BFS

Time complexity of DFS & BFS

DFS and BFS both have a time complexity of **O (E + V)**

In both DFS and BFS, we **visit every node once and every edge twice**, hence the time complexity!

CodeIT

Lab Session 1

- In this lab session, you will be implementing **traversal.py**
- Your task is to implement the DFS & BFS algorithms starting from a source
- Performing traversals starting from a source means that only those vertices with a path from the source will be visited
- To test, run `` `python utils/traversal_test.py` ``

Graph

- This class represents an undirected graph and has been implemented for you.
- This graph contains an attribute `adjList` where `adjList[v]` contains all adjacent vertices to vertex v. For instance, if `adjList[0] = [1, 2, 3]`, then the edges `(0, 1)`, `(0, 2) and (0, 3)` exist.

dfs solution

```python
def dfsRecurse(v, graph, visited, result):
    visited[v] = True
    result.append(v)

    for dest in graph.adjList[v]:
        if not visited[dest]:
            dfsRecurse(dest, graph, visited, result)

def dfs(graph, start):

    visited = [False] * len(graph.adjList)
    result = []
    dfsRecurse(start, graph, visited, result)

    return result
```

bfs solution

```python
def bfs(graph, start):

    visited = [False] * len(graph.adjList)
    pq = []
    pq.append(start)
    visited[start] = True
    result = [start]

    while (len(pq) != 0):
        v = pq.pop(0)
        for dest in graph.adjList[v]:
            if not visited[dest]:
                result.append(dest)
                visited[dest] = True
                pq.append(dest)

    return result
```