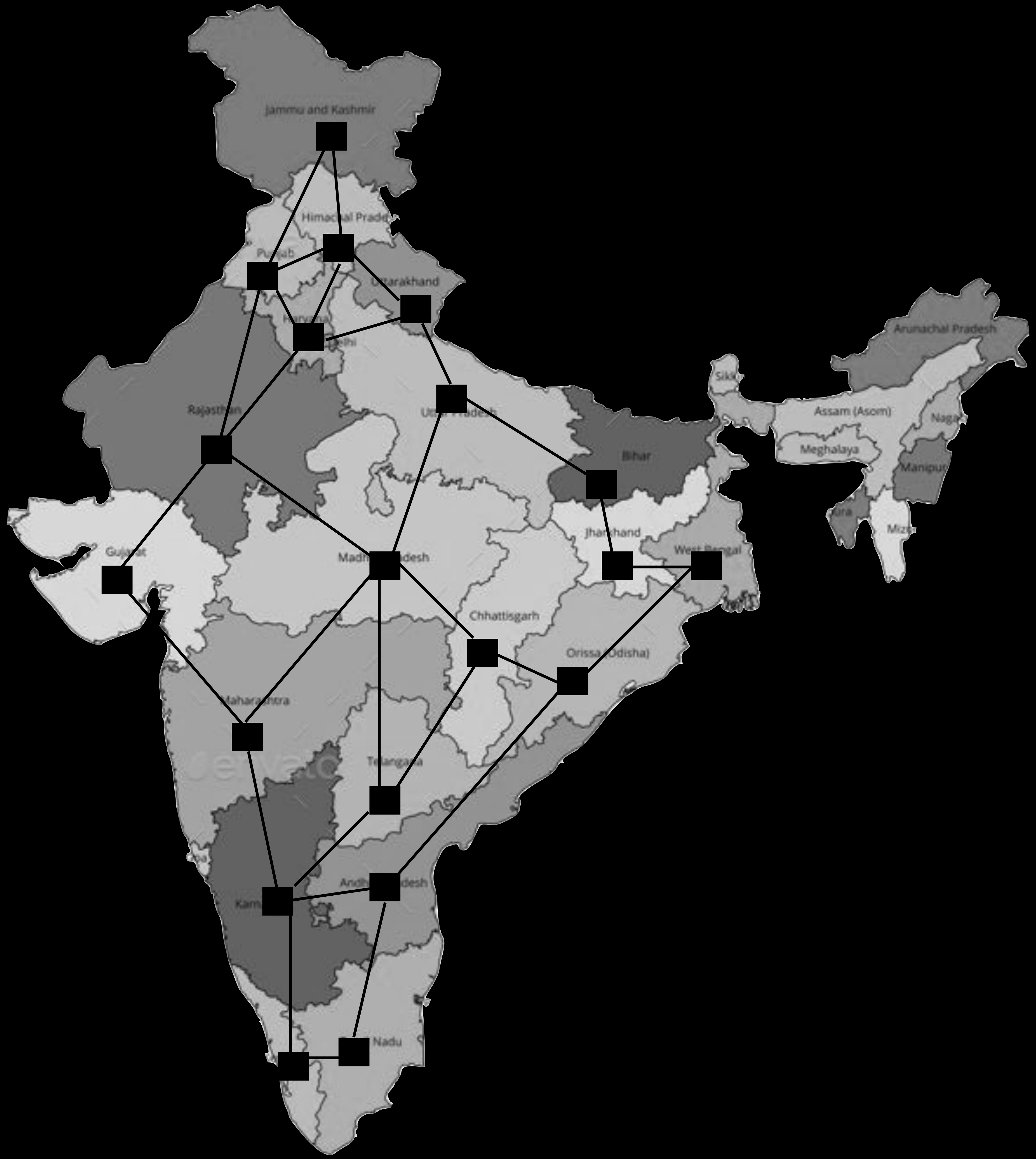
ADVANCED PROGRAMMING

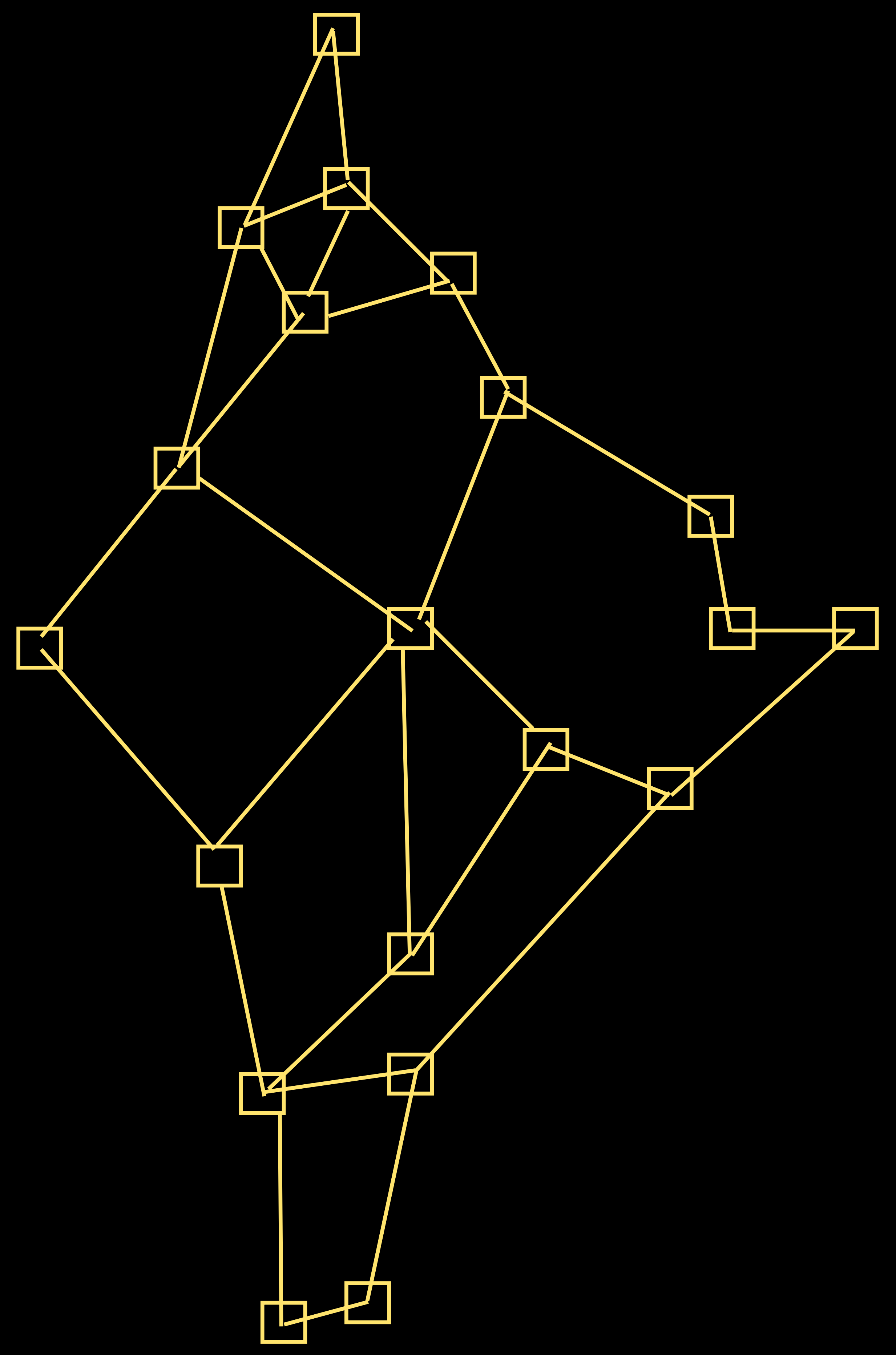
Graphs

Ramaseshan Ramachandran

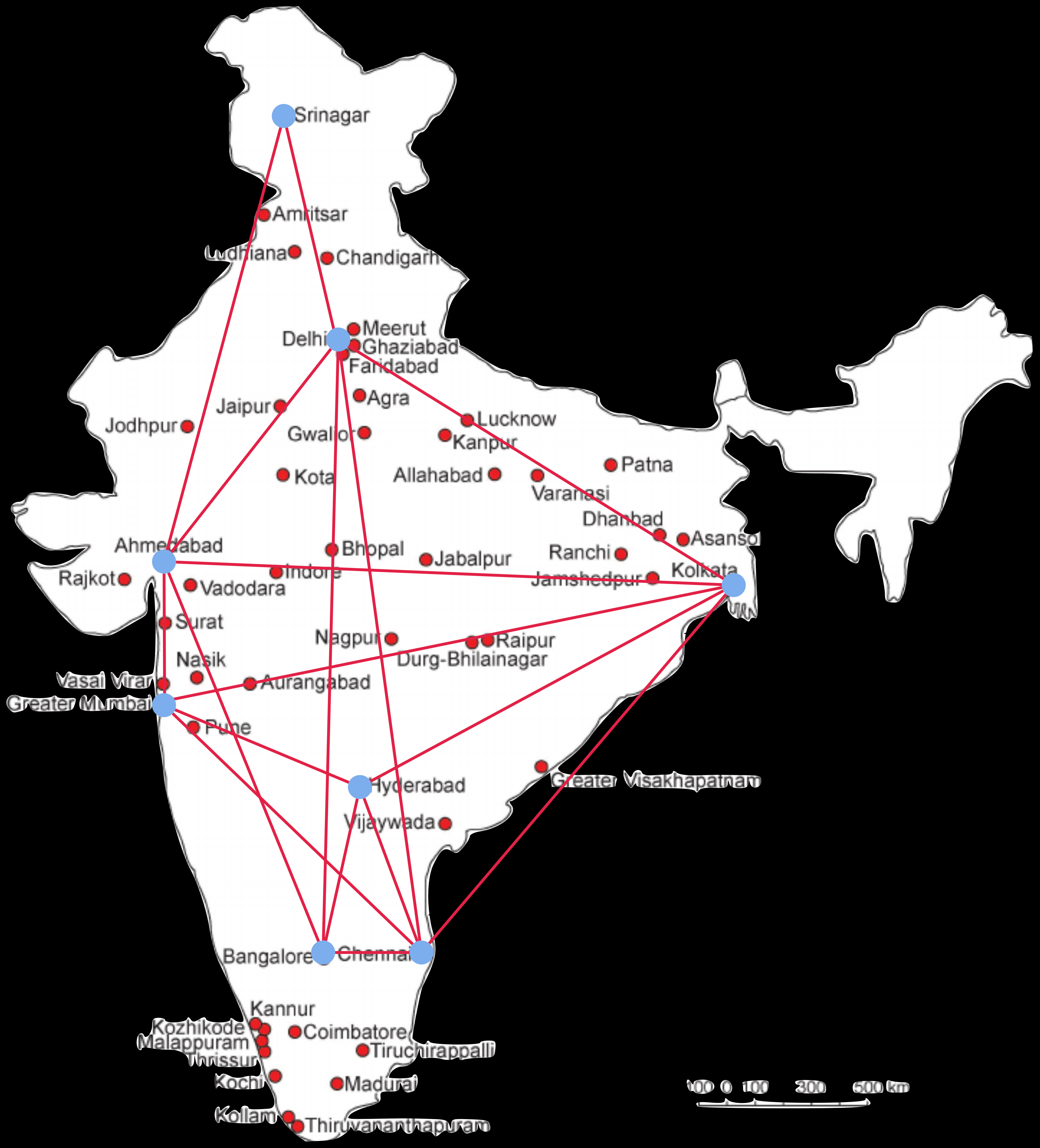
COLOURING A MAP



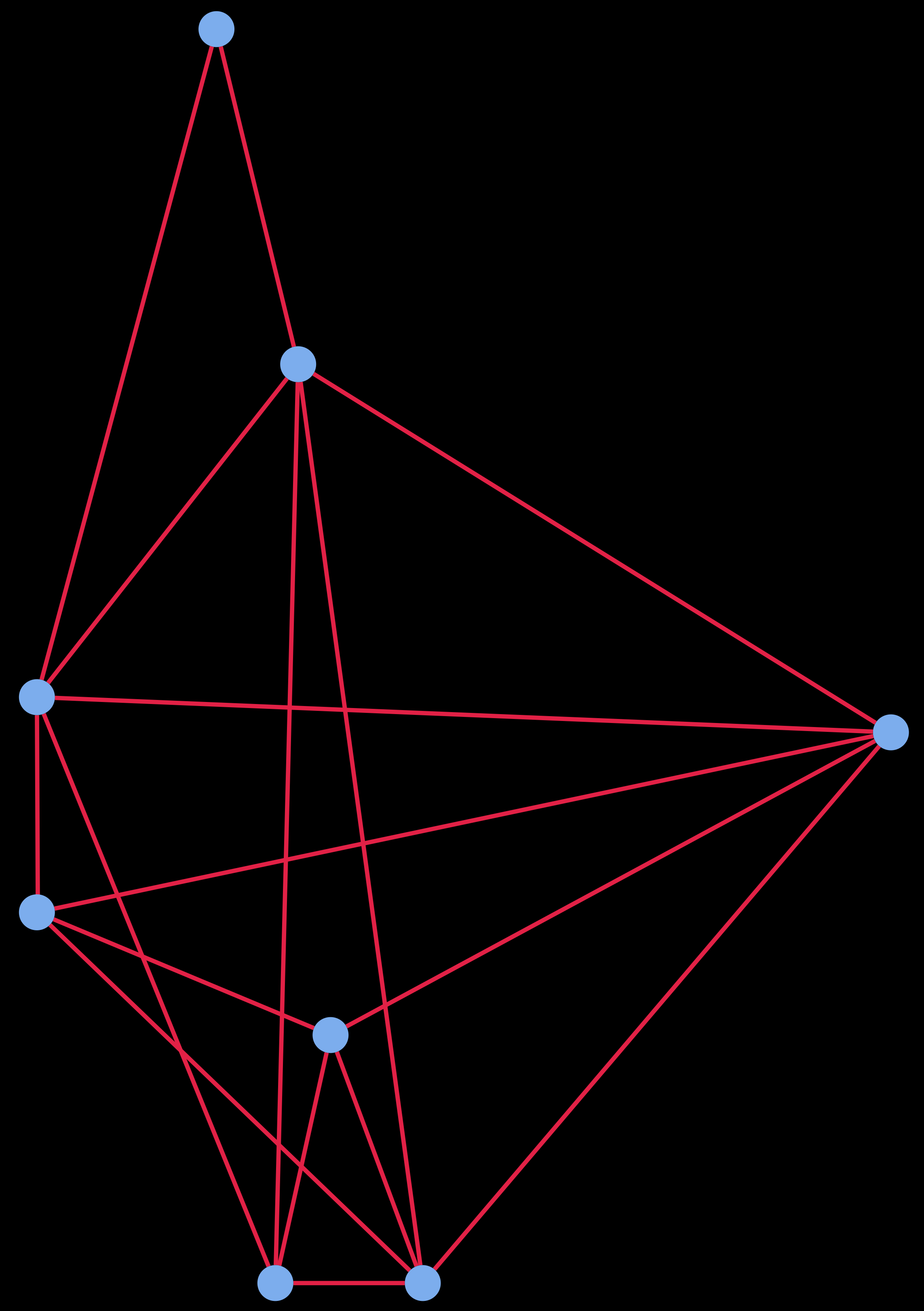
COLOURING A MAP



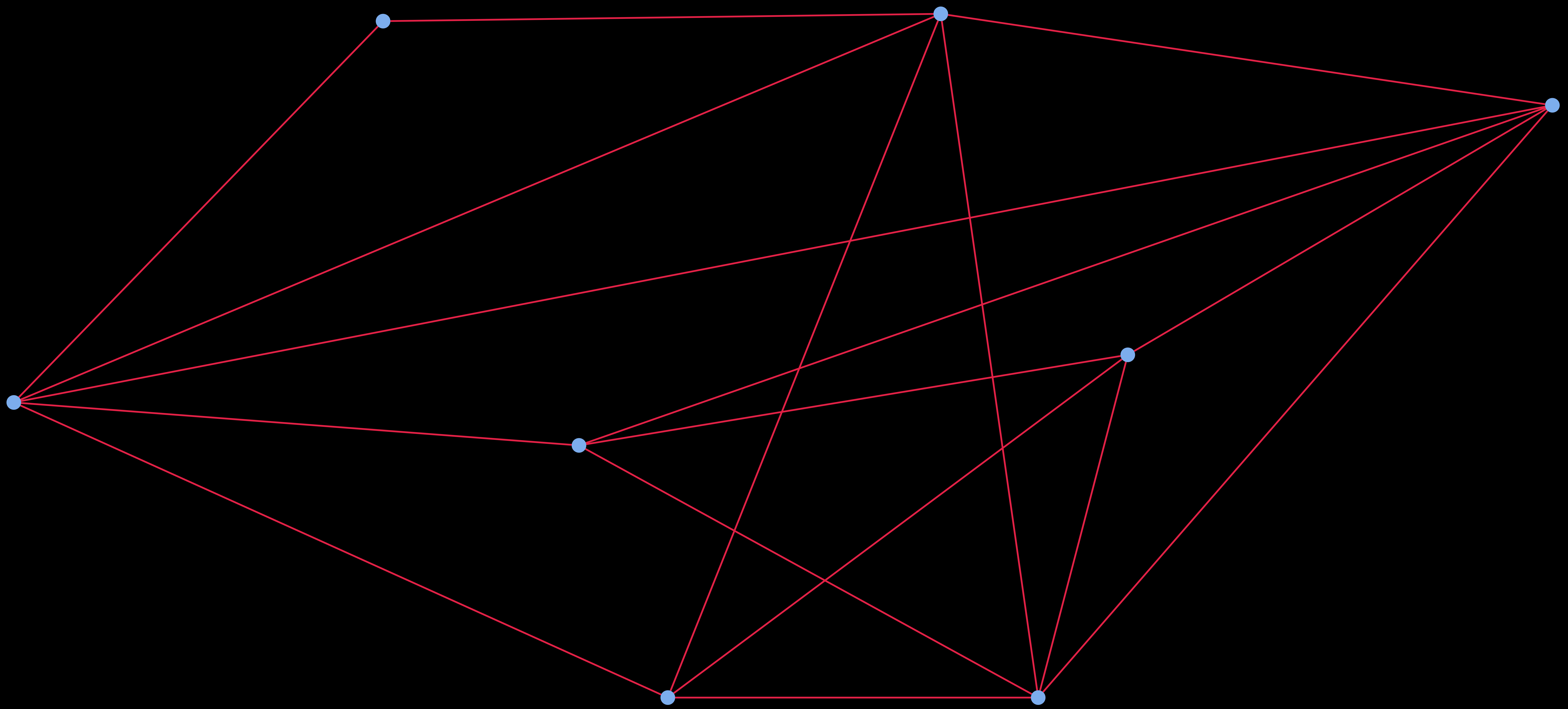
FLIGHT CONNECTIONS



CITIES AS A GRAPH

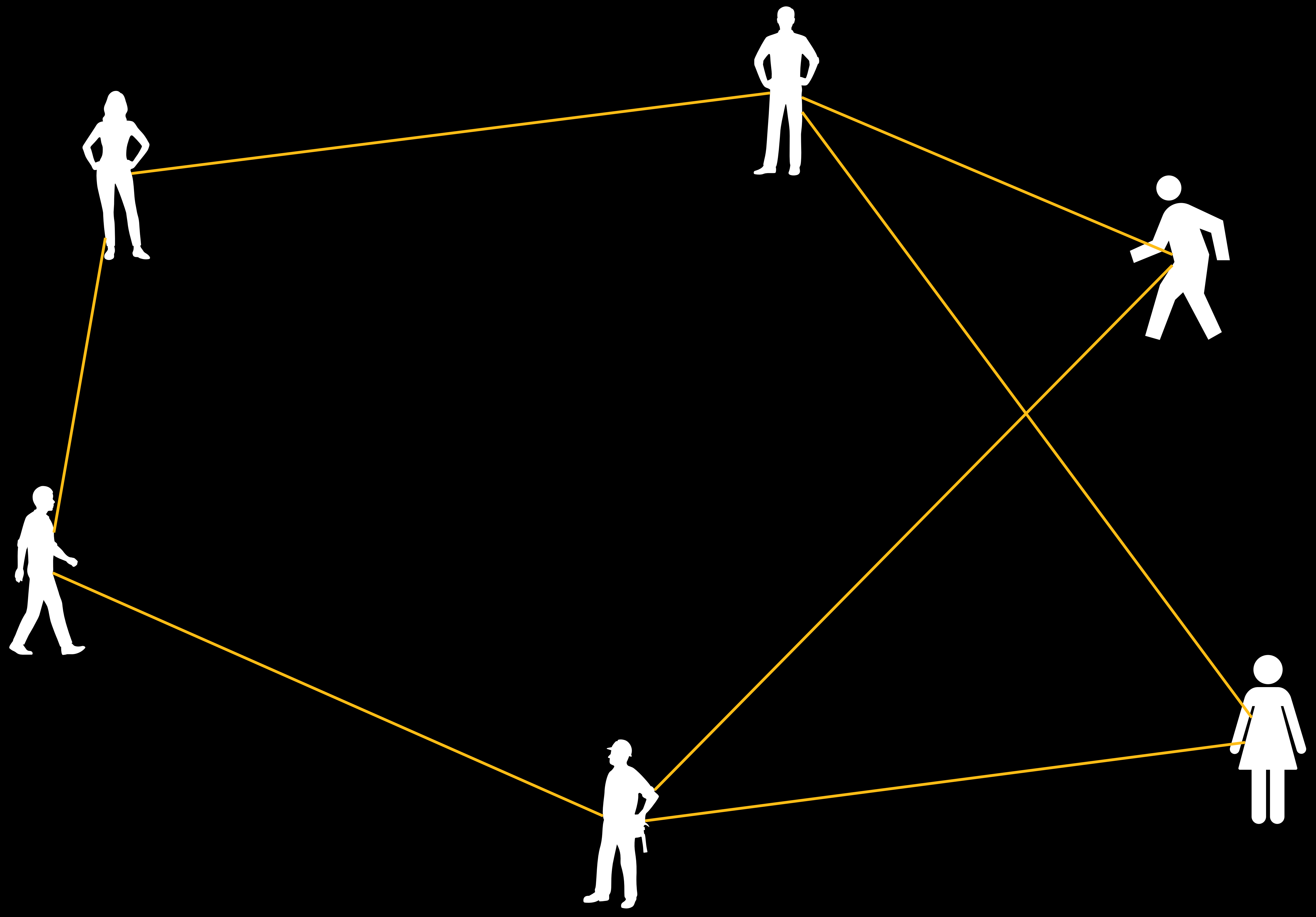


CITIES AS A GRAPH



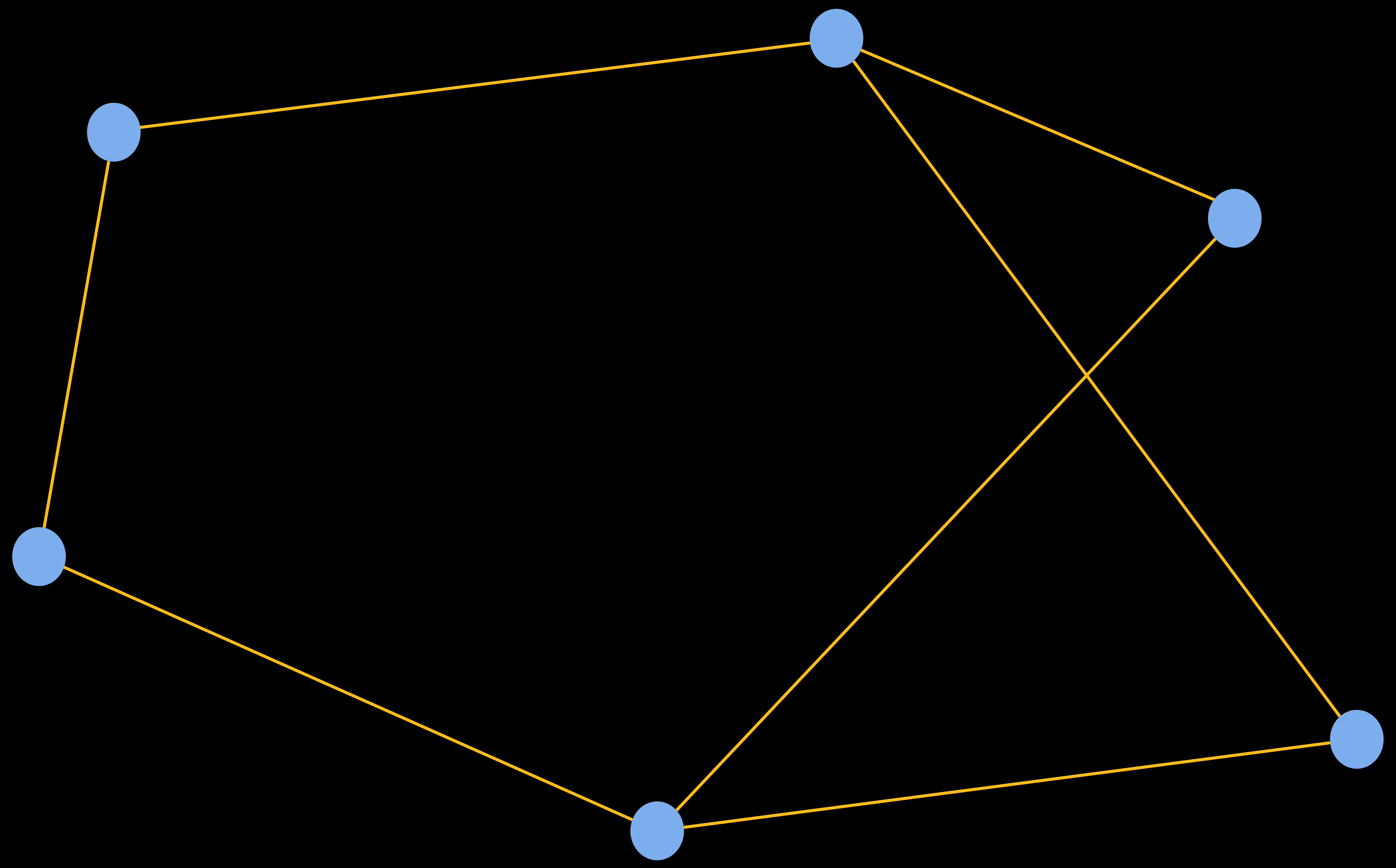
FRIENDSHIP

A relation is defined as a <u>subset</u> of the Cartesian product of two sets



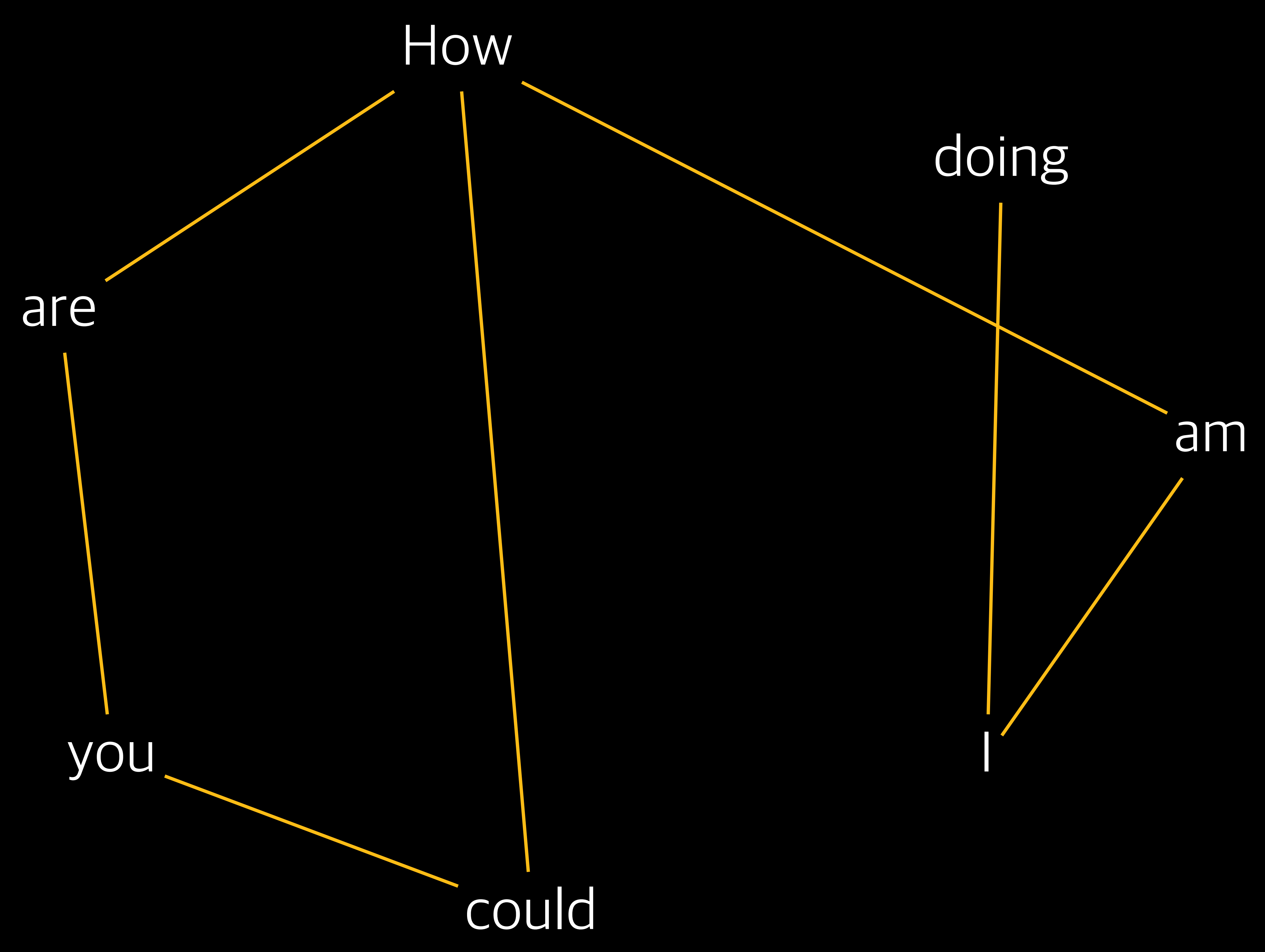
RELATIONSHIP AS A GRAPH

A relation is defined as a subset of the Cartesian product of two sets



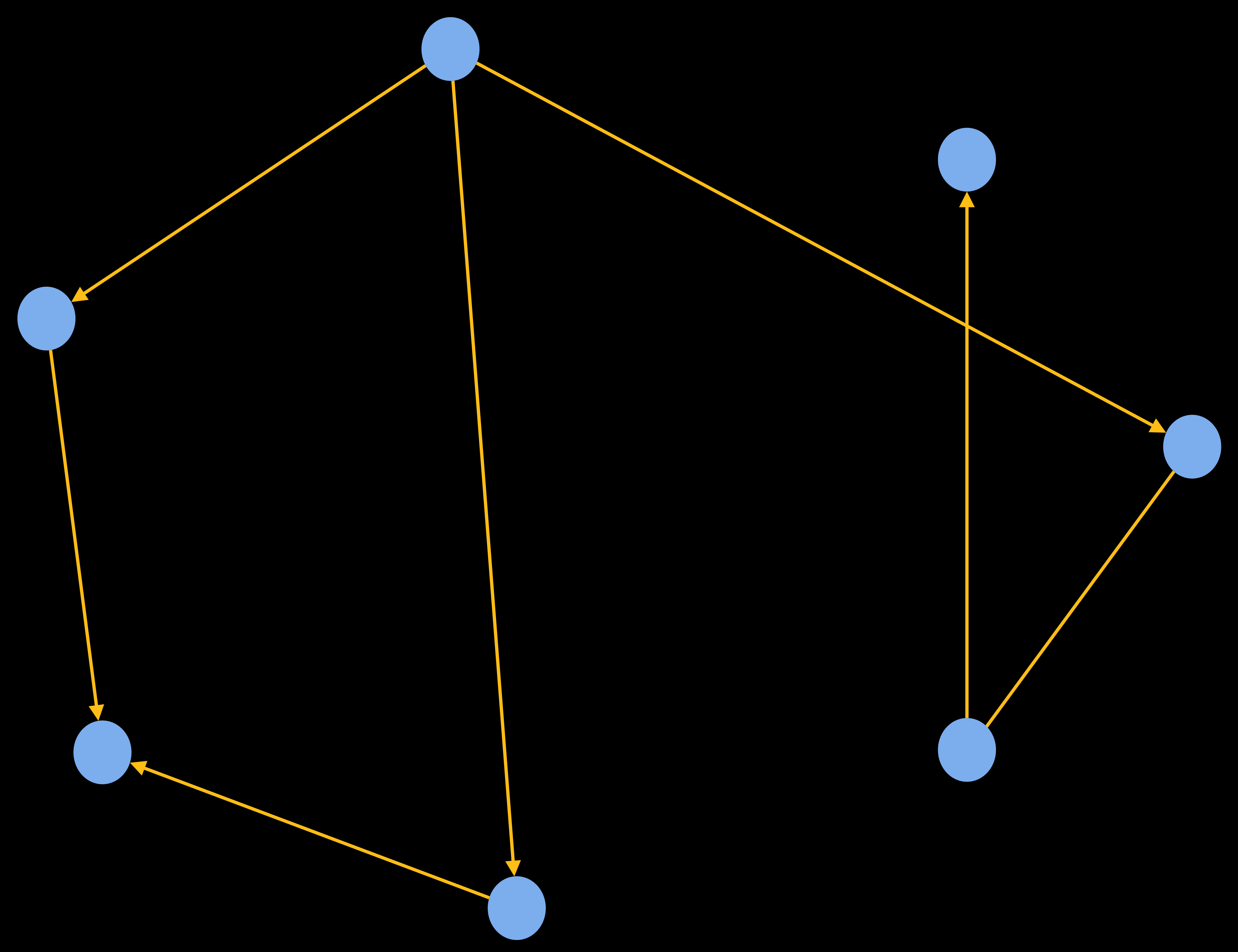
SENTENCES AS A GRAPH

How are you? How could you? How am I doing?



SENTENCES

How are you? How could you? How am I doing?



GRAPHS

Most impressive and immersive representation of combinatorics is Graph

A graph can have cycles, and multiple paths between two nodes.

Non-linear data structure

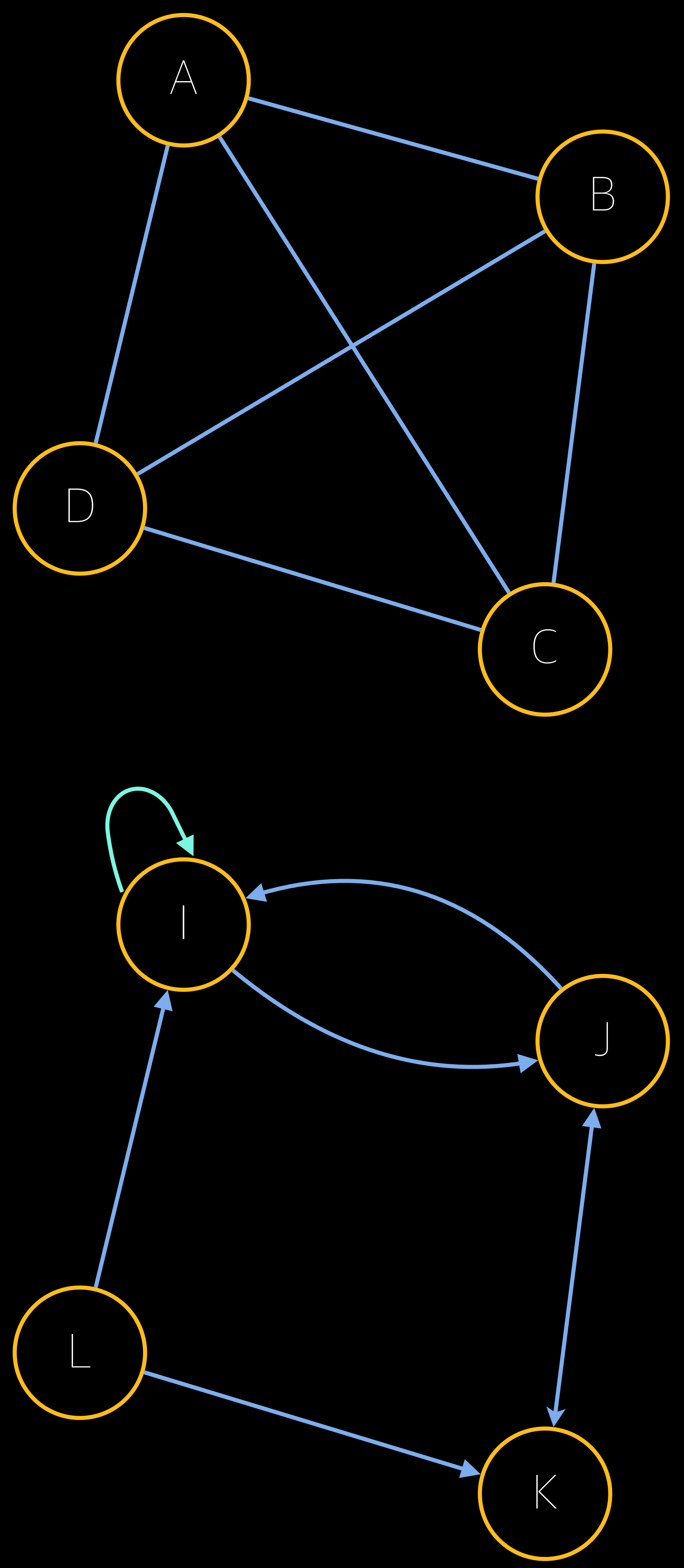
MODELLING A PROBLEM

- Representing the information of the problem
- How to use the information?
- How do I manipulate the information?
- We need some abstract notation and structures to model the problem
- Identify key elements and relationships
- What data structure?
- Any mathematical representation?

Modelling a problem is the process of creating a simplified representation of a real-world situation. This representation, called a model, helps us understand, analyze, and potentially solve the problem more effectively

FORMAL DEFINITION

- A graph is a pair of vertices (or nodes) connected by edges
- Represented as $G = (V, E)$
- $V = \{v_1, v_2, \cdots, v_n\}$ or $v \in V$
- and $E = \{e_1, e_2, \cdots, e_m\}$ or $e \in E$
- $E \subseteq \{(u, v) \mid u, v \in V\}$
- n = $|V|$ - number of vertices
- m = $|E|$ - number of edges



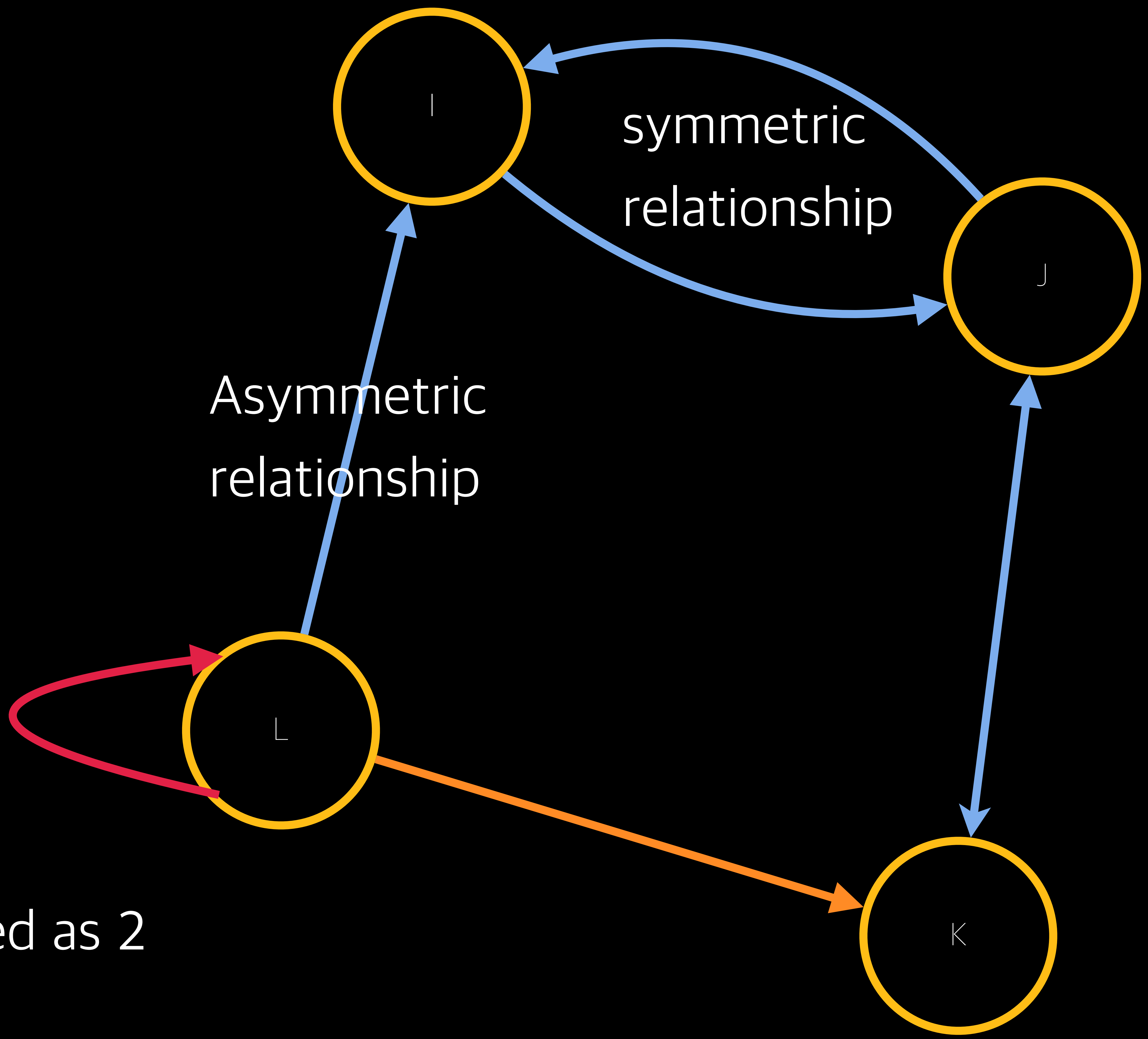
DIRECTED GRAPH(DIGRAPH)

A directed graph of digraph is a pair $G = (V, E)$ where

- ✦ V is a set of vertices and
- ✦ E is a set of directed edges $E \subseteq V \times V$ and $0 \leq E \leq V(V-1)$
- ✦ Edges are ordered pairs of vertices
    - If vertices u and v are connected, then there is a directed edge from u to v

A digraph can have self loops (u, u) and it is counted as 2

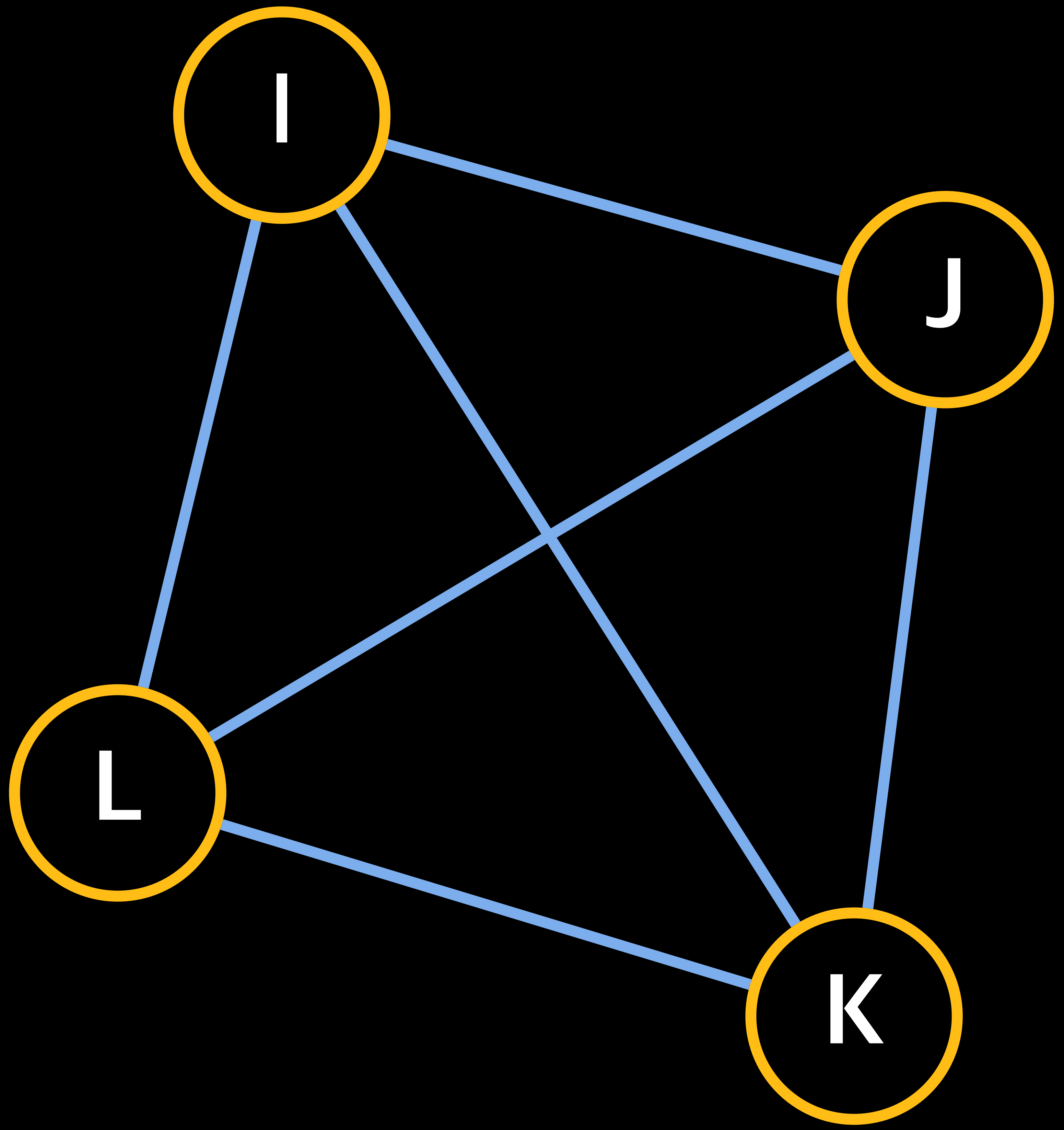
It can have at the most n^2 edges



UNDIRECTED GRAPH

A undirected graph is a pair $G = (V, E)$where

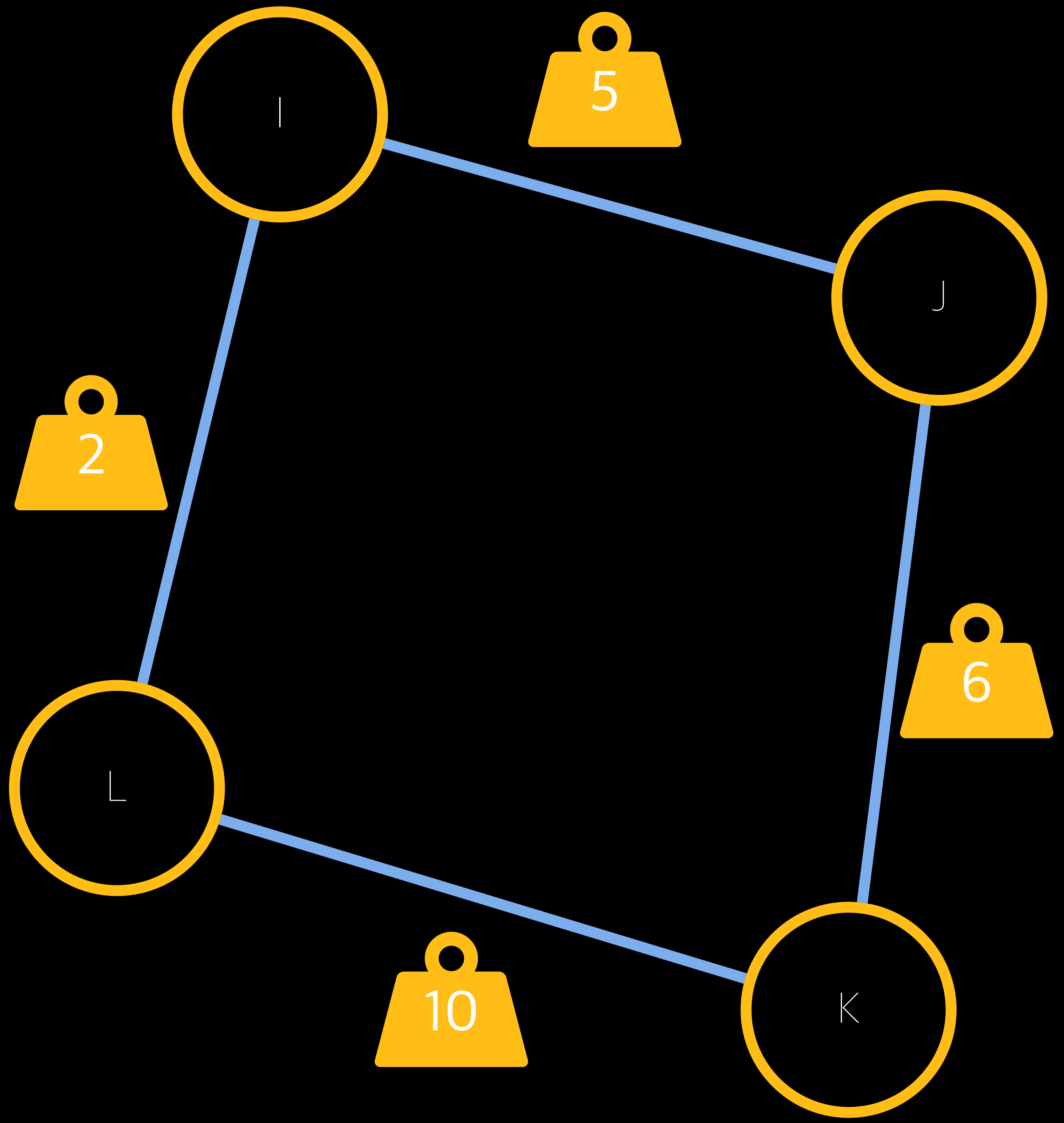
- V is a set of vertices and
- E is a set of edges $E \subseteq V \times V$ and $0 \leq E \leq \binom{V}{2}$
- Undirected graph can have at most $\dfrac{n(n-1)}{2}$ edges
- A undirected graph cannot have self loops (u, u).



WEIGHTED GRAPH

A weighted graph is a pair $G = (V, E)$where

- V is a set of vertices and
- E is a set of edges $E \subseteq V \times V$
- Every $e \in E$ will be associated with a number



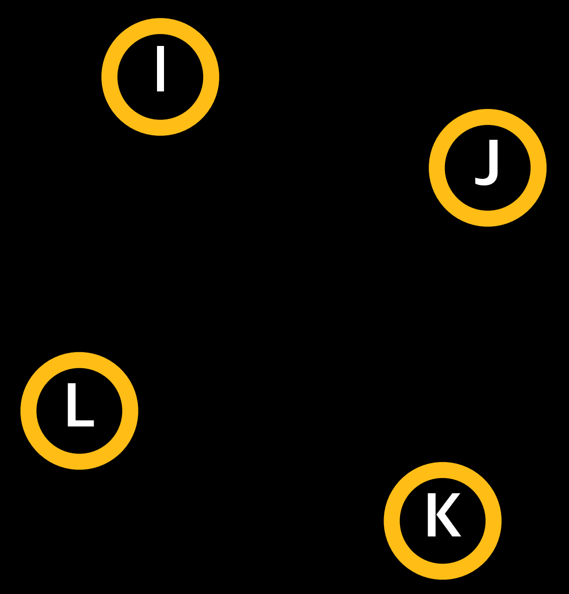
TERMINOLOGY

- A vertex u is a neighbour of (adjacent to) a vertex v in a graph $G = (V, E)$, if there is an edge $\{u, v\} \in E$. Here u is a neighbour of v and vice versa. It is written as $u \sim v$
- In directed graphs, for any directed edge $u \rightarrow v$, we call u a **predecessor** of v, and we call v a **successor** of u.
- The **neighbourhood** of a vertex $v \in V$ is its set of all neighbours of v:
  $v_{nh} = \{u \mid \{u, v\} \in E\}$
- The **degree** of a vertex v is the size of the neighbourhood $|\ v_{nh}\ |$
- Edge is **incident** on a vertex if the vertex is one of its endpoints. A vertex is incident on an edge if it is one of the endpoints of the edge
- A directed graph is strongly connected if all vertices are reachable from all other vertices
- In an undirected graph, edge (u, v)always implies(v, u)

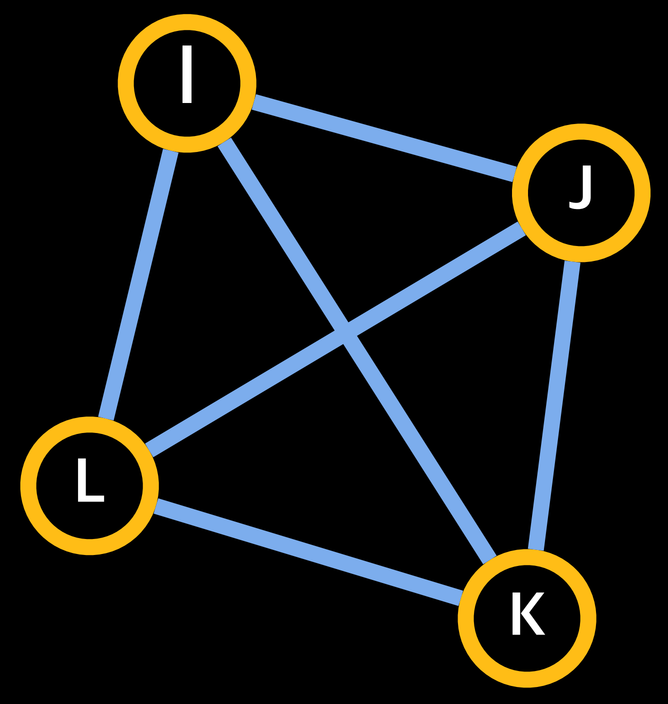
TERMINOLOGY

- A **walk** is a sequence of vertices $v_1, v_2, \cdots, v_k$ such that $\forall i \in 1,2,\cdots,k-1, (v_i \sim v_{i+1})$
- A **path** is a walk where $v_i \neq v_j, \forall i \neq j$ - A path is a walk that visits each vertex at most once
- A **closed walk** is a walk where $v_1 = v_k$.
- A **cycle** is a **closed path**, i.e. a path combined with the edge (v_k, v_1). An undirected graph is connected if all vertices are reachable from all other vertices
- A vertex v is **reachable** from a vertex u, if there is a path starting at v and ending at u
- A graph **acyclic** if no subgraph is a **cycle**
- An undirected graph with no **cycles** is a **forest** and if it is connected it is called **a tree**
- A **tree** is a connected acyclic graph

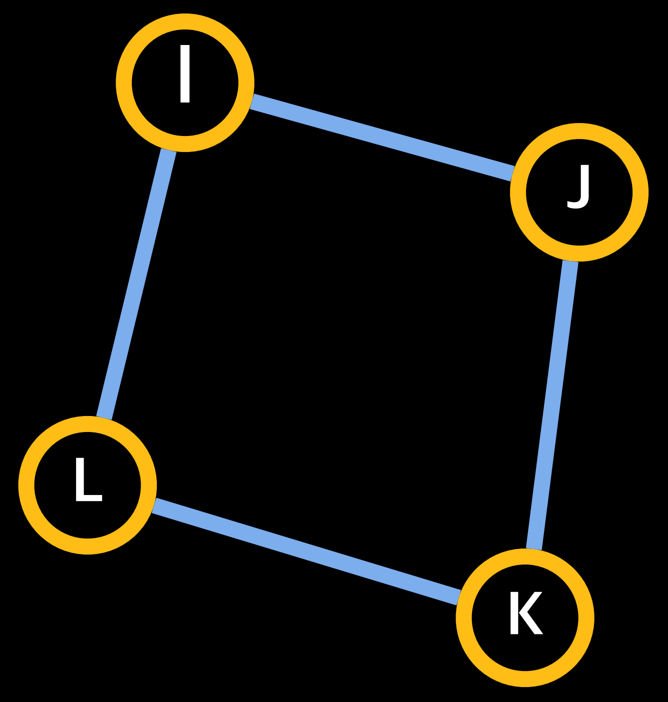
EXAMPLES



Null Graph



Complete Graph

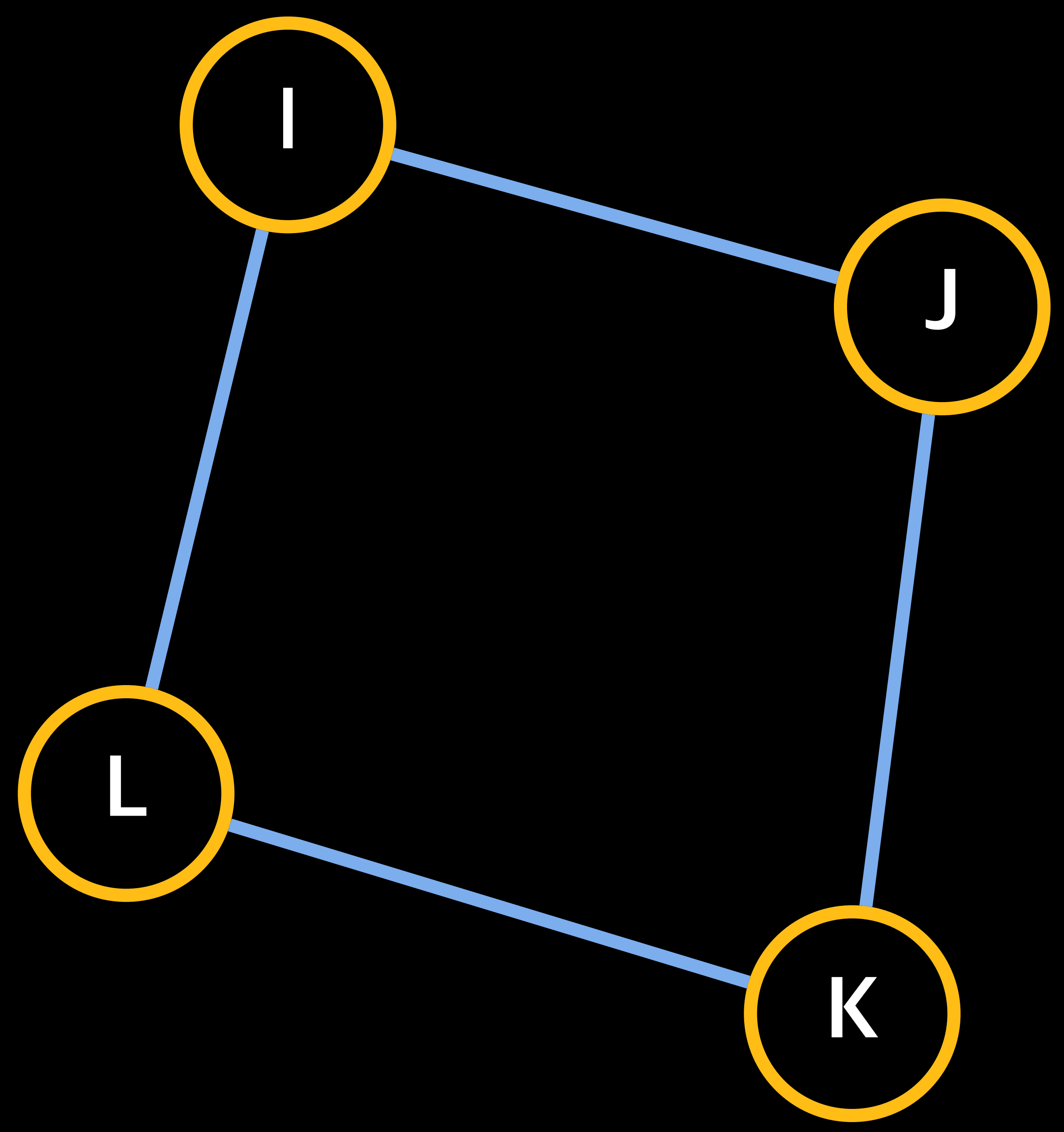


Cycle Graph

DATA STRUCTURE OF A GRAPH

- A diagrammatic representation using points joined by and lines
- How do we store it in the computer?
- Can we list vertices adjacent to each vertex of the graph as a dictionary/hash map?

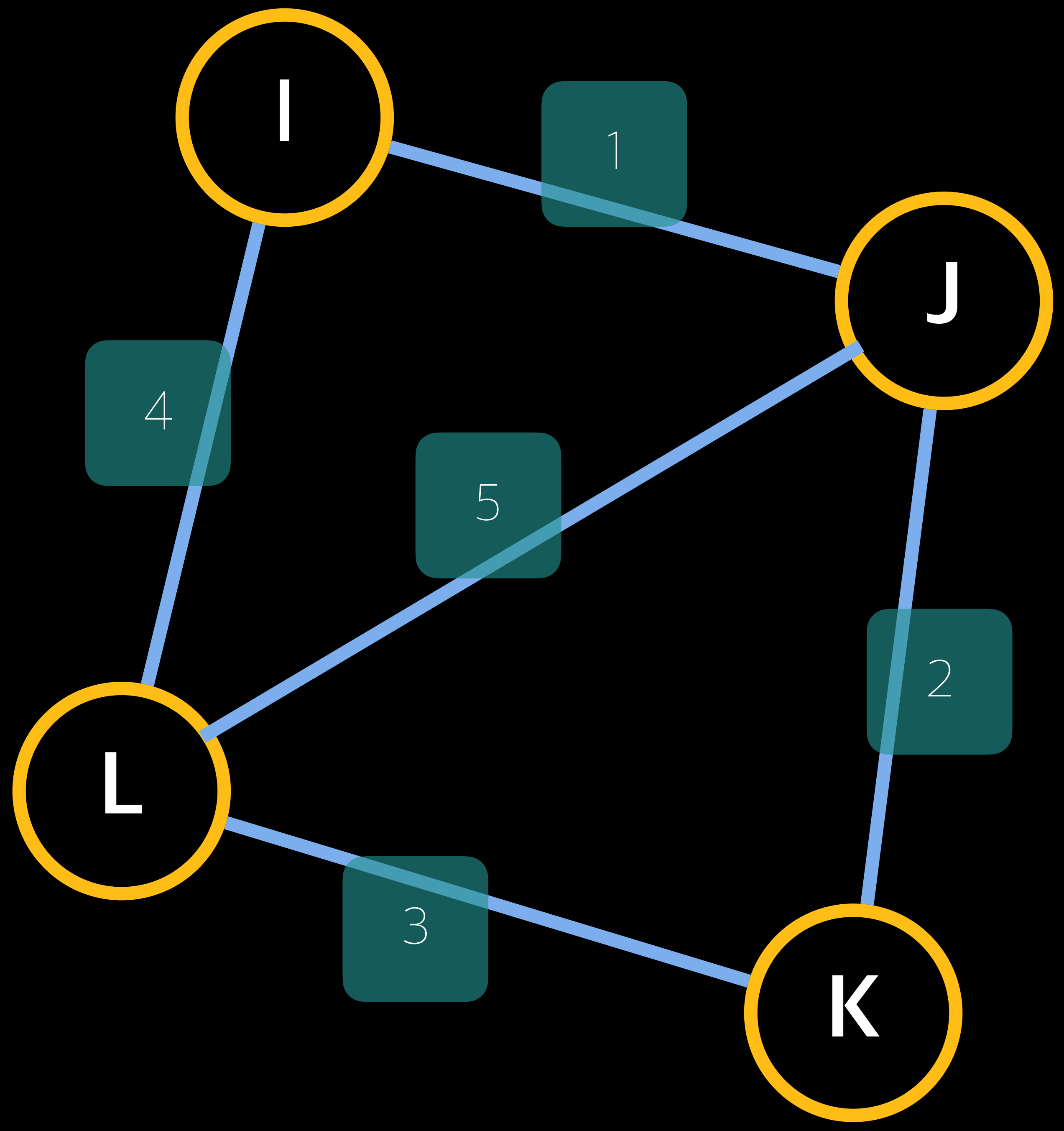
```
G = {
        "I":["J", "L"],
        "J":["I", "K"],
        "K":["J","L"],
        “L":["I","K"]
    }
```



- Any other representation?

INCIDENCE MATRIX

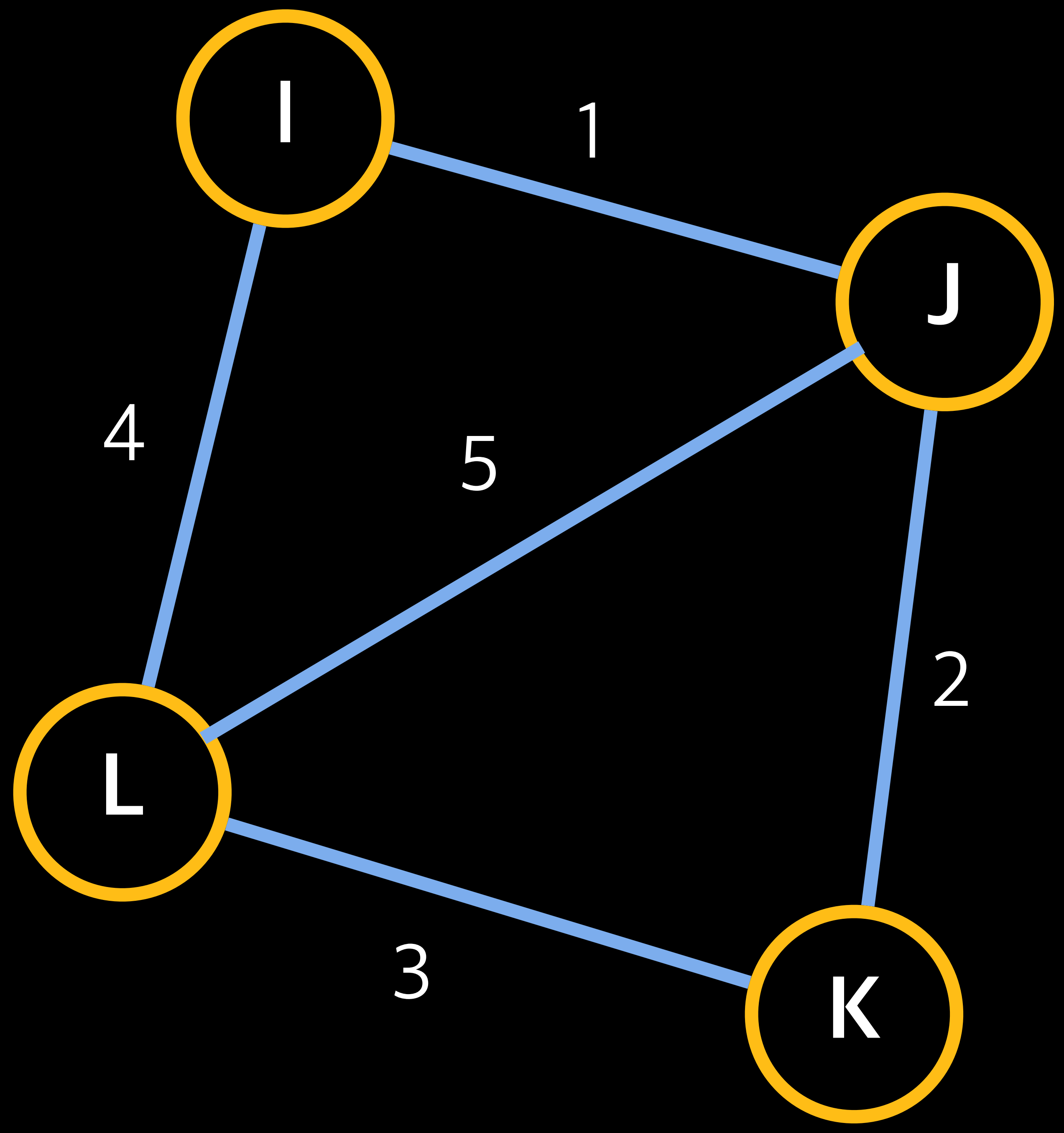
- Incidence Matrix M is a $n \times m$
- $m_{ij} = \begin{cases} 1, & \text{if the vertex is connected by edge} \\ 0, & \text{otherwise} \end{cases}$

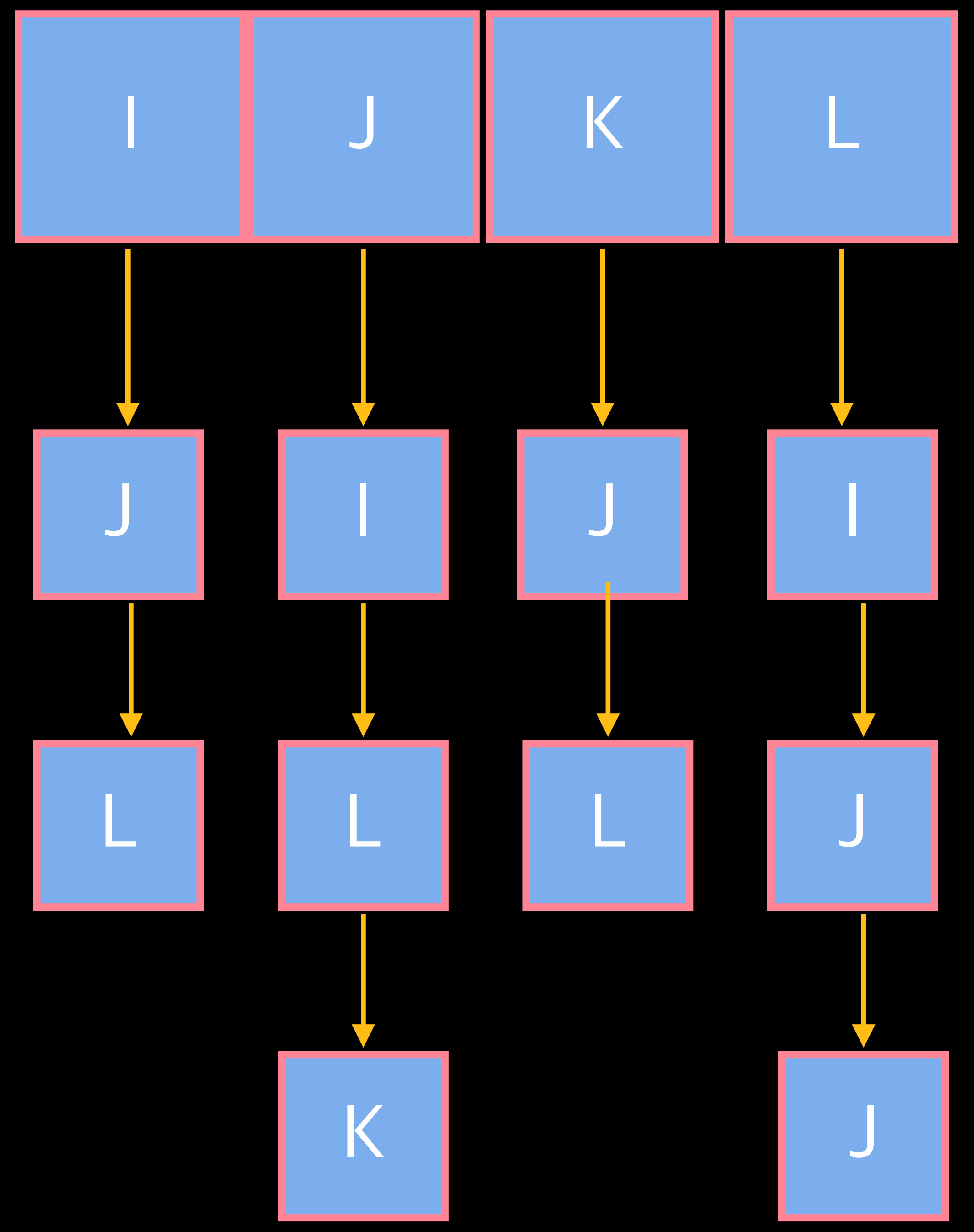


$$\mathrm{M} = \begin{array}{c} \\ I \\ J \\ K \\ L \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

ADJACENCY LIST

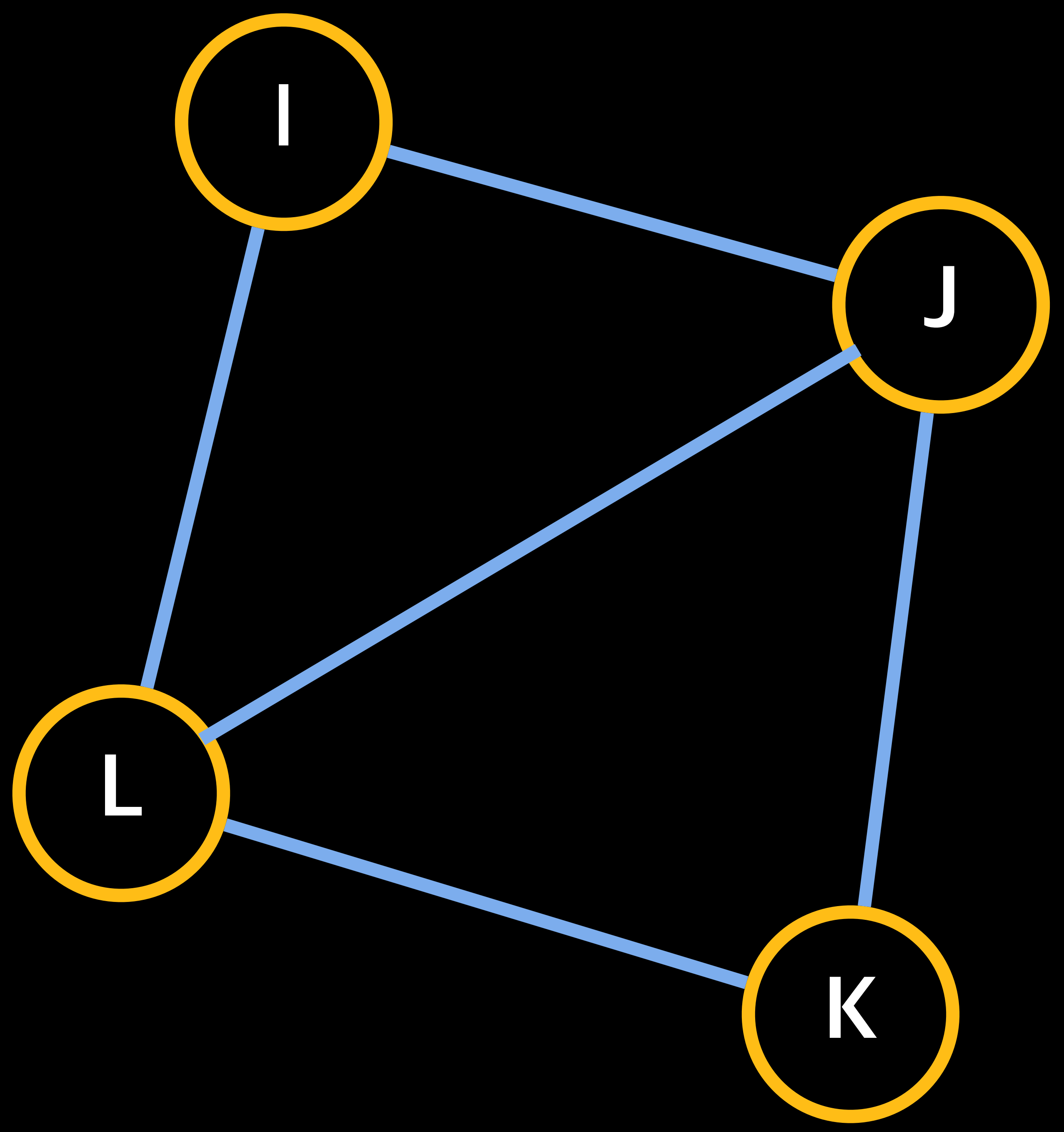
Represents pairwise connections between vertices





ADJACENCY MATRIX

- Represents pairwise connections between vertices
- Square Matrix, A, whose ij^{th} entry is the number of edges joining vertex i and vertex j



$$\mathbf{A} = \begin{array}{c} \\ I \\ J \\ K \\ L \end{array} \begin{array}{c} \begin{array}{cccc} I & J & K & L \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

PROPERTY OF THE ADJ MATRIX

For all undirected graphs, the adjacency matrix is always symmetric $a_{ij} = a_{ji} \forall i, j$ and $a_{ij} = 0$ when $i = j$

For digraphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero.

Time Complexity of finding connectivity of any two vertices is $O(1)$

Time Complexity of listing all the neighbours is $O(n)$

Space complexity is $O(n^2)$

If the application required only to find the maximum number of edges in a graph, what data structure would you choose?

SUMMARY OF COMPLEXITY

Adjacency Matrix	Faster to add or remove modes Faster to check if a node exists Better for dense graphs - $\lvert E \rvert \approxeq O(\lvert V \rvert^2)$ Storage $= = O(\lvert V \rvert^2)$
Edge List	Faster if the graph is sparse Better for sparse graphs - $\lvert E \rvert \approxeq O(\lvert V \rvert)$ Storage $= = O(\lvert V \rvert + \lvert E \rvert)$

	Standard adjacency list (linked lists)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)$

Times for basic operations on standard graph data structures

OPERATIONS ON A GRAPH

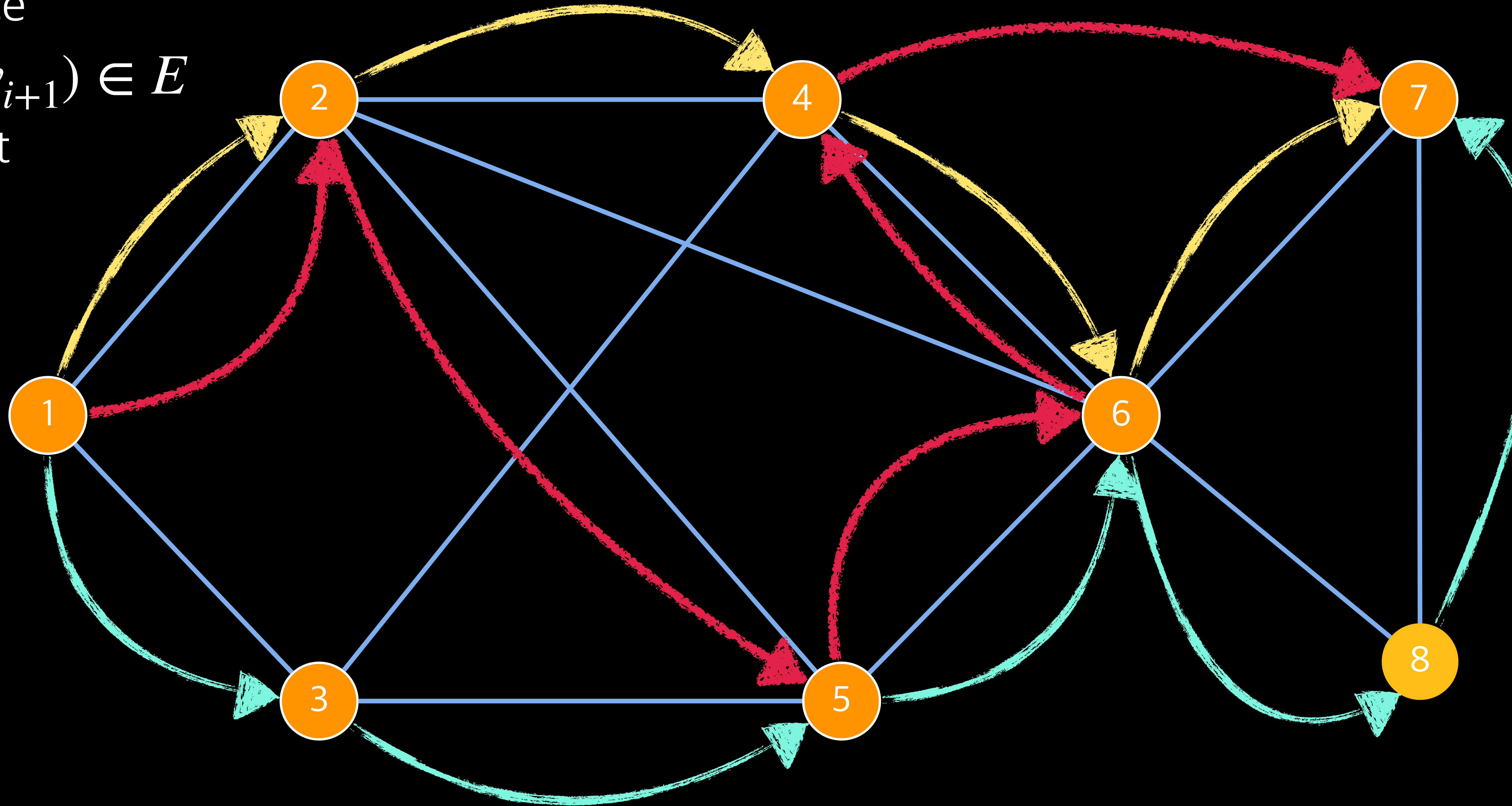
Basic operations

- Search for a node
- Insert a node
- Delete a node
- Degree of a node
- Number of vertices
- Number of edges
- Most connected node

FINDING A PATH

Find a sequence of vertices $v_0, v_1, \cdots, v_n$ such that

- v_0 is source
- Each $(v_i, v_{i+1}) \in E$
- v_k is target



FINDING A PATH

- In general, start at the start node and follow the edges to find its neighbours
- Mark vertices that have been visited
- Keep track of vertices whose neighbours have already been explored
- Avoid going round indefinitely in circles
- Two approaches – Breadth first and Depth First – known as BFA and DFS
- BFS – Start at the first node, visit all the direct neighbours recursively
- DFS – Visit all the way down in one potential path, then backtrack to visit all potential paths

BREADTH-FIRST SEARCH

- Start from i, visit a neighbour j
  Suspend the exploration of i and explore j instead
- Continue till a vertex with no unexplored neighbours is reached
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a stack
- Last in, first out: most recently suspended is checked first

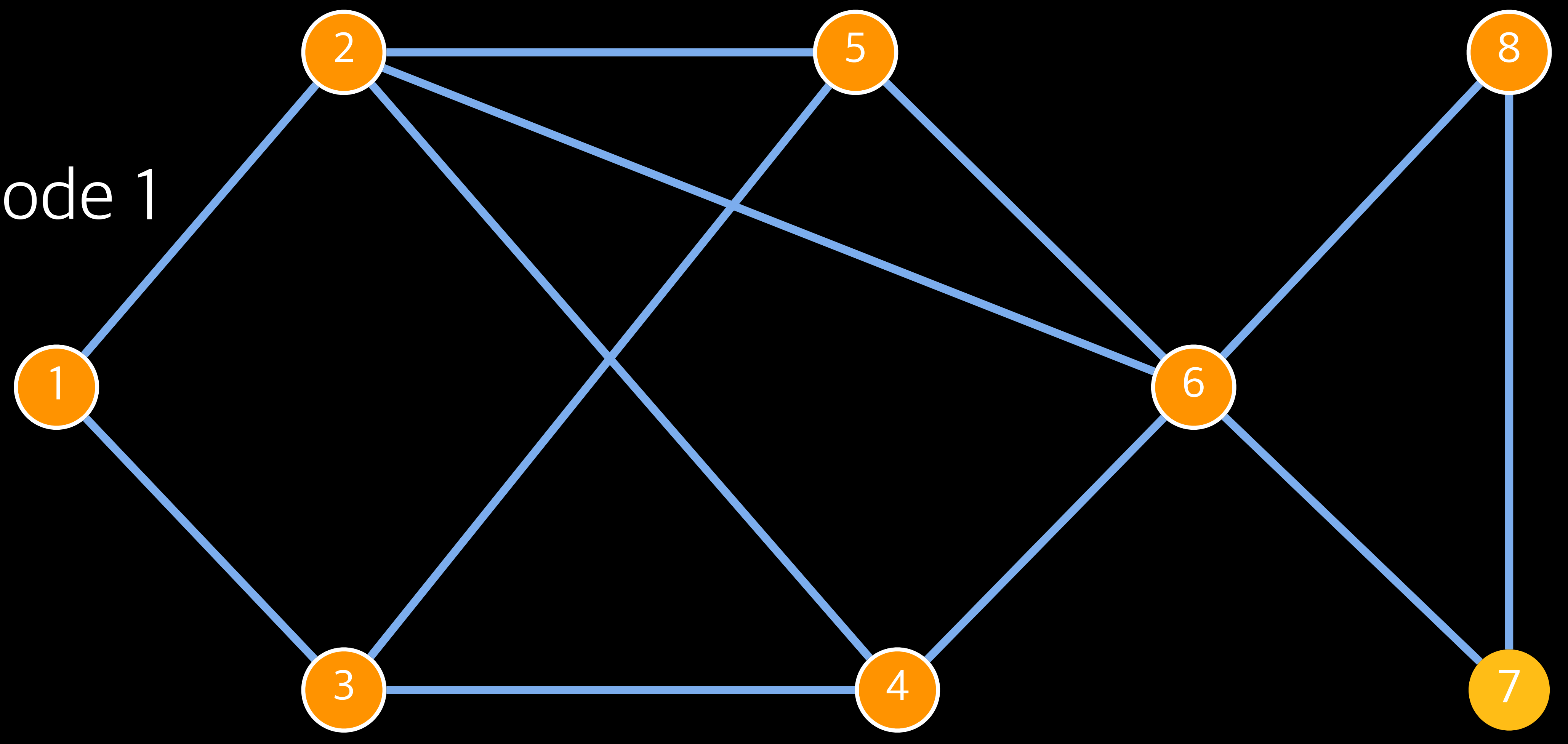
BREADTH-FIRST SEARCH

- Explore the graph level by level
- Visit all the neighbourhood that are one-step away or nearest neighbour
- Then second level two steps

  …
- Memorise all the visited vertices
- Keep track of vertices visited, but whose neighbours are unexplored/investigared

BREADTH-FIRST TRAVERSAL

Starting at node 1




```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```

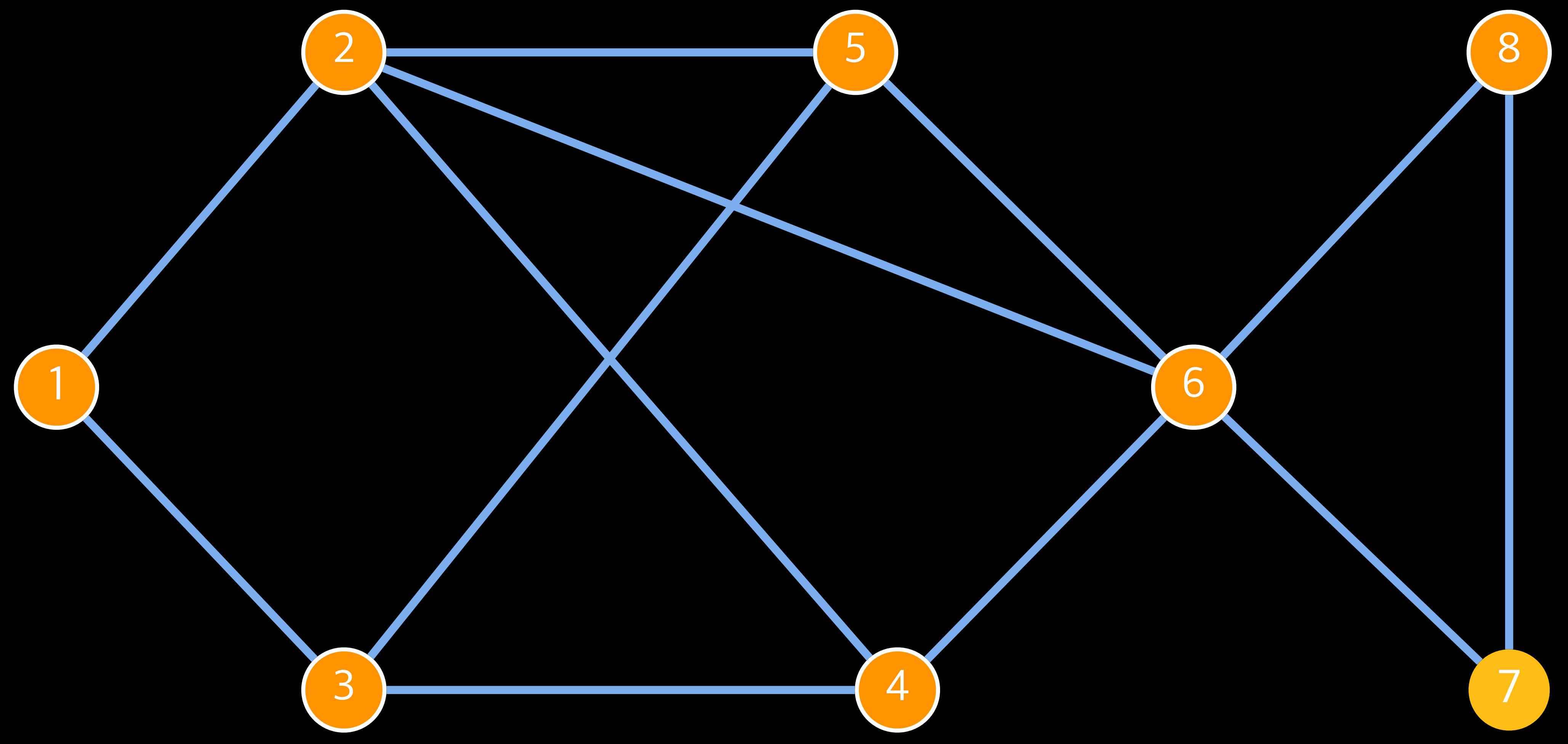
Visited Nodes

1	2	3	4	5	6	7	8

Queue

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

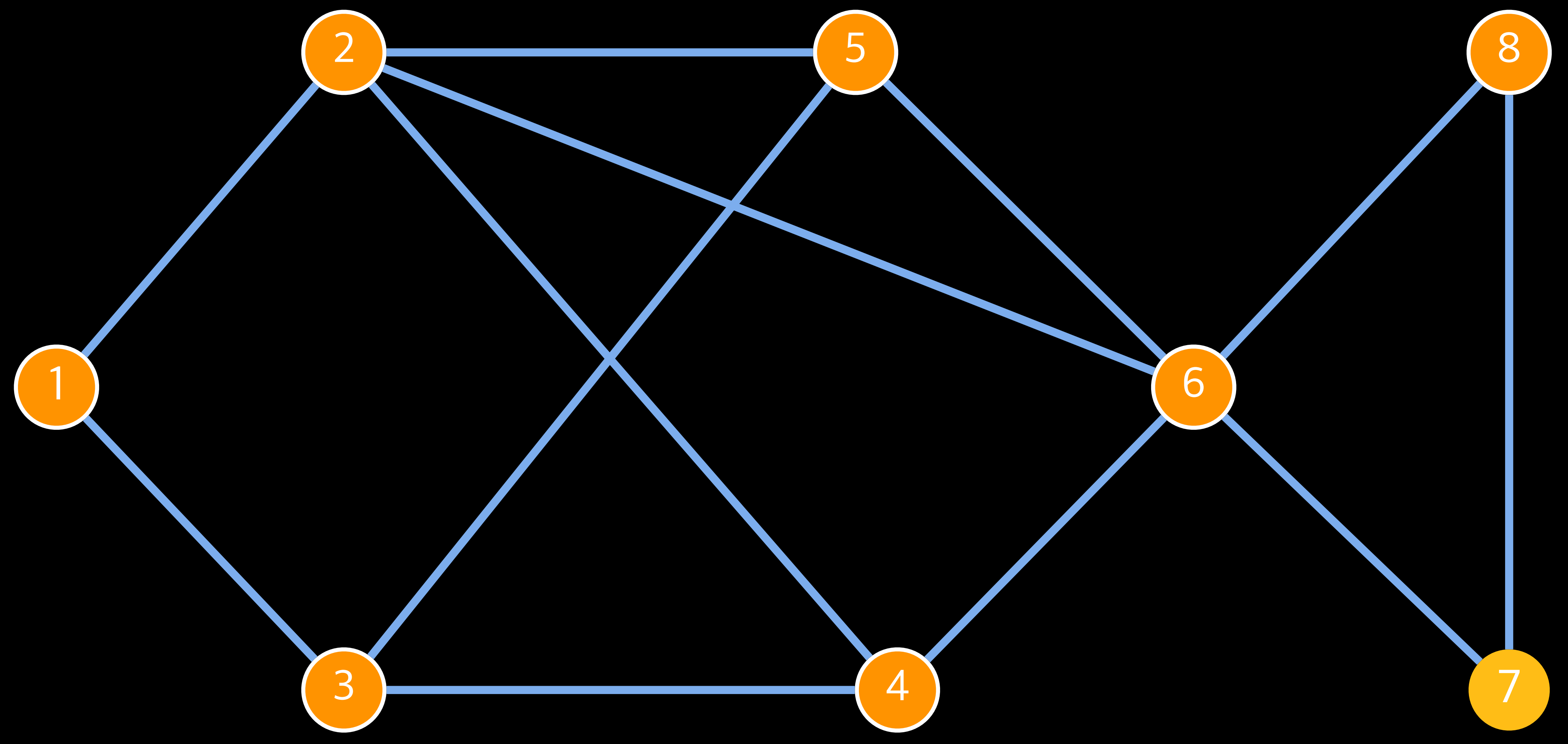
1	2	3	4	5	6	7	8
1							

Queue

1							

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



Visited Nodes

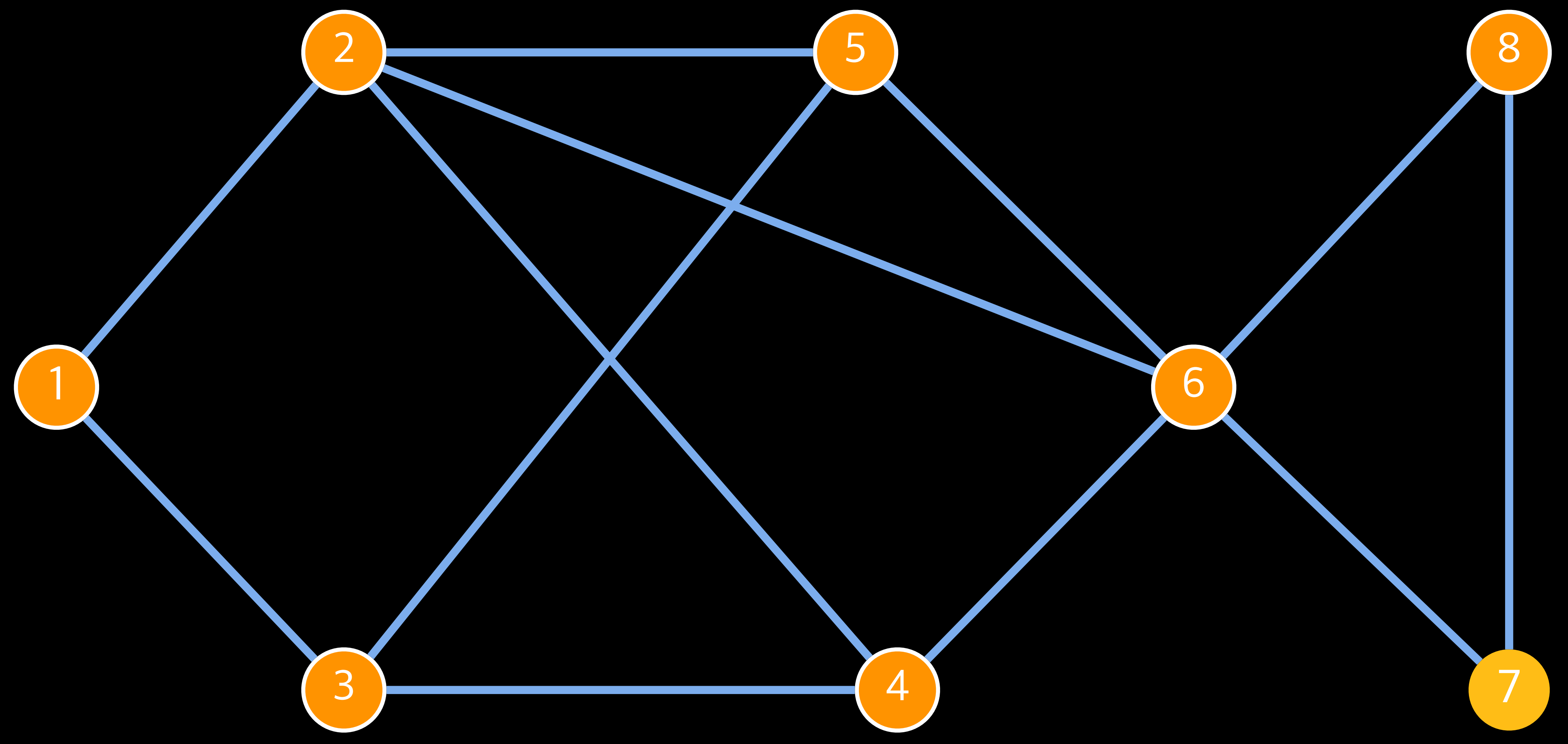
1	2	3	4	5	6	7	8
1	2						

Queue

2							

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

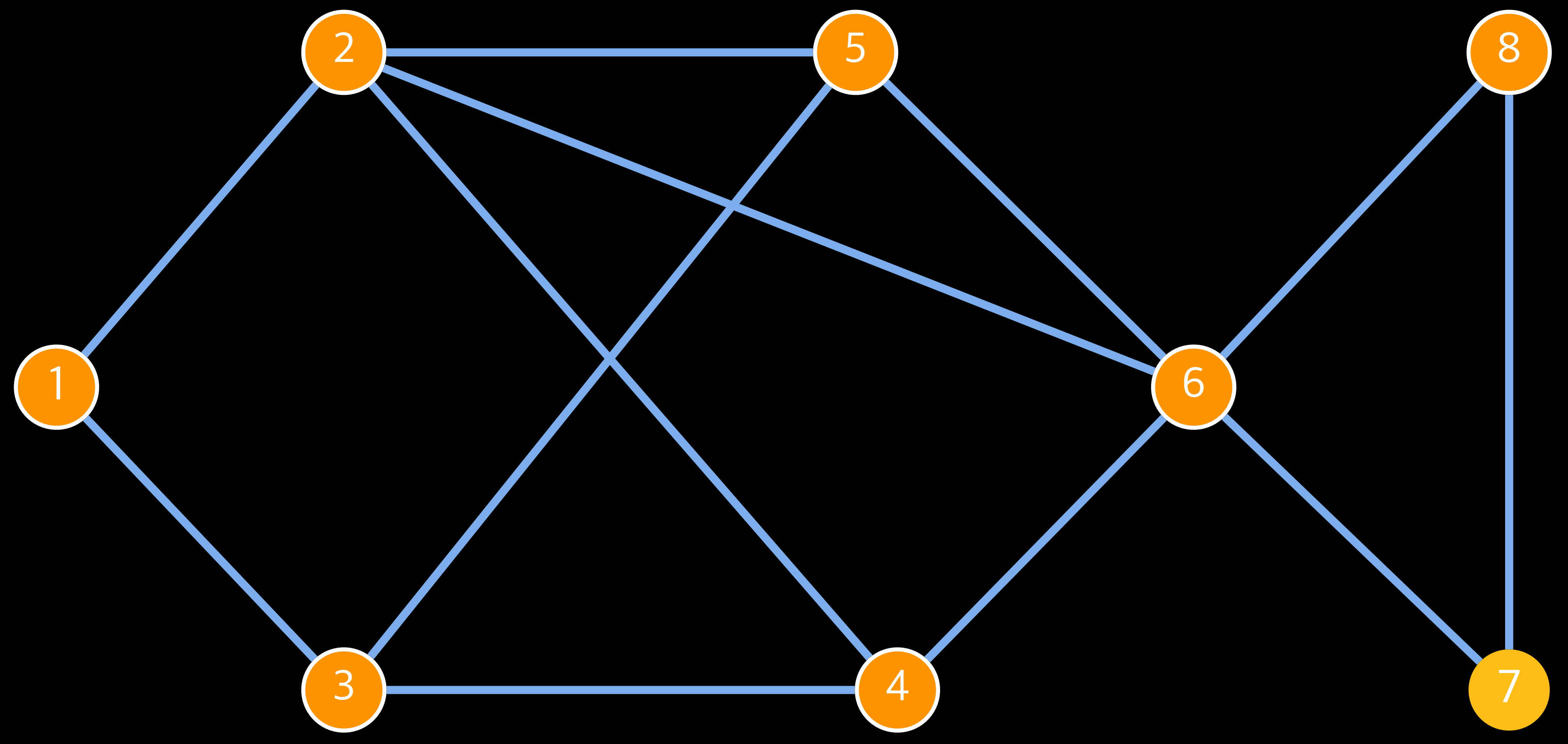
1	2	3	4	5	6	7	8
1	2	3					

Queue

2	3						

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Queue

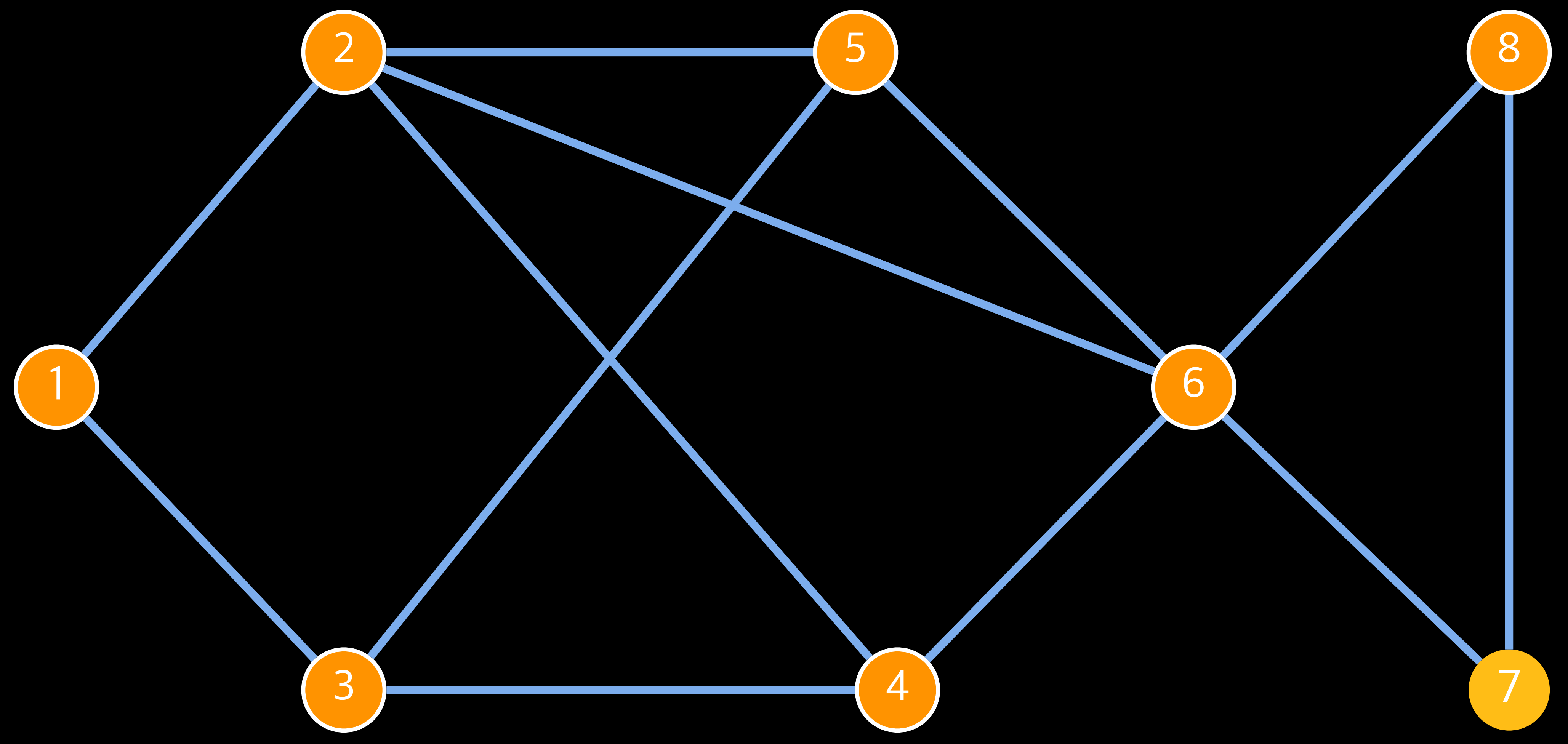
3	4						

Visited Nodes

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

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



Visited Nodes

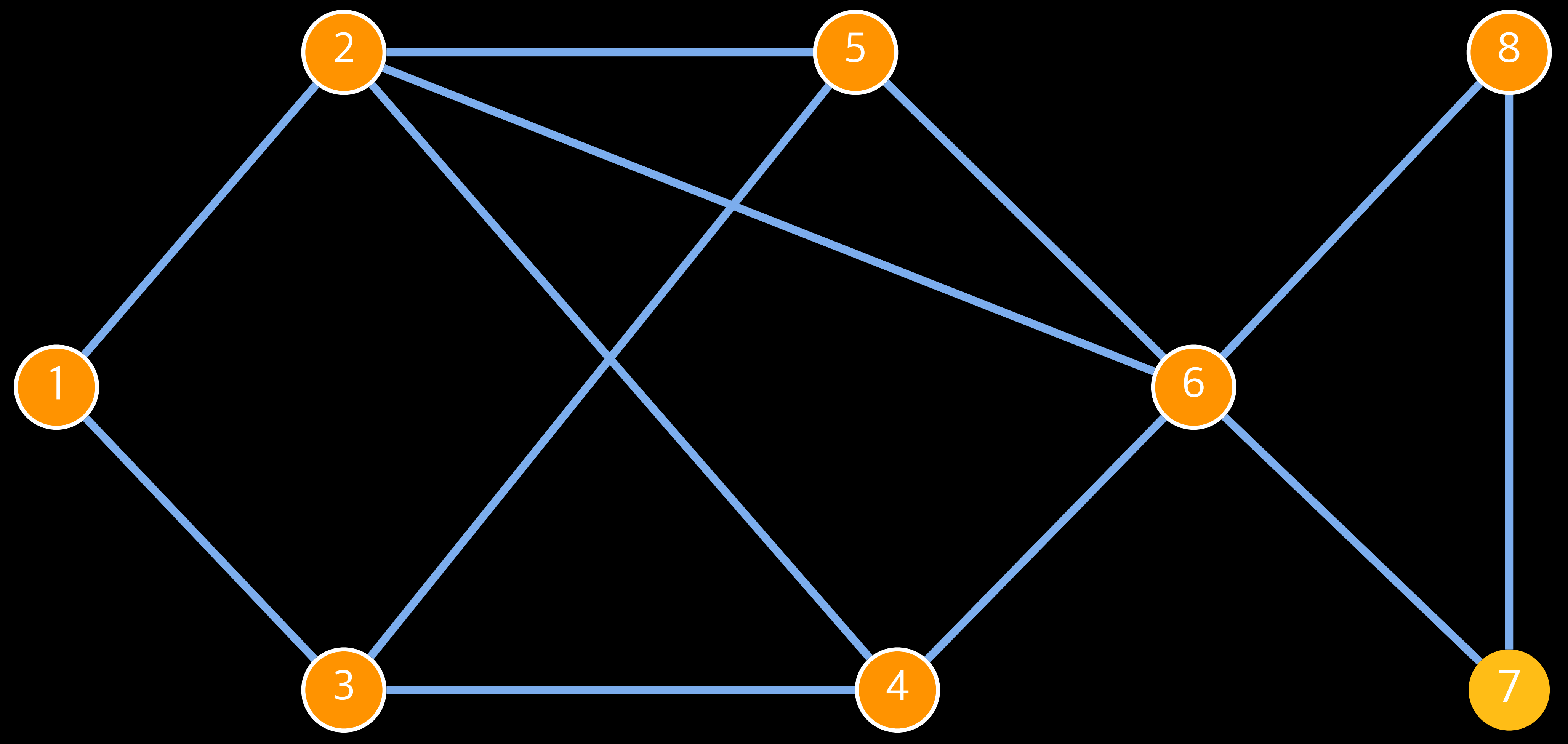
1	2	3	4	5	6	7	8
1	1	1	1	1			

Queue

3	4	5					

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

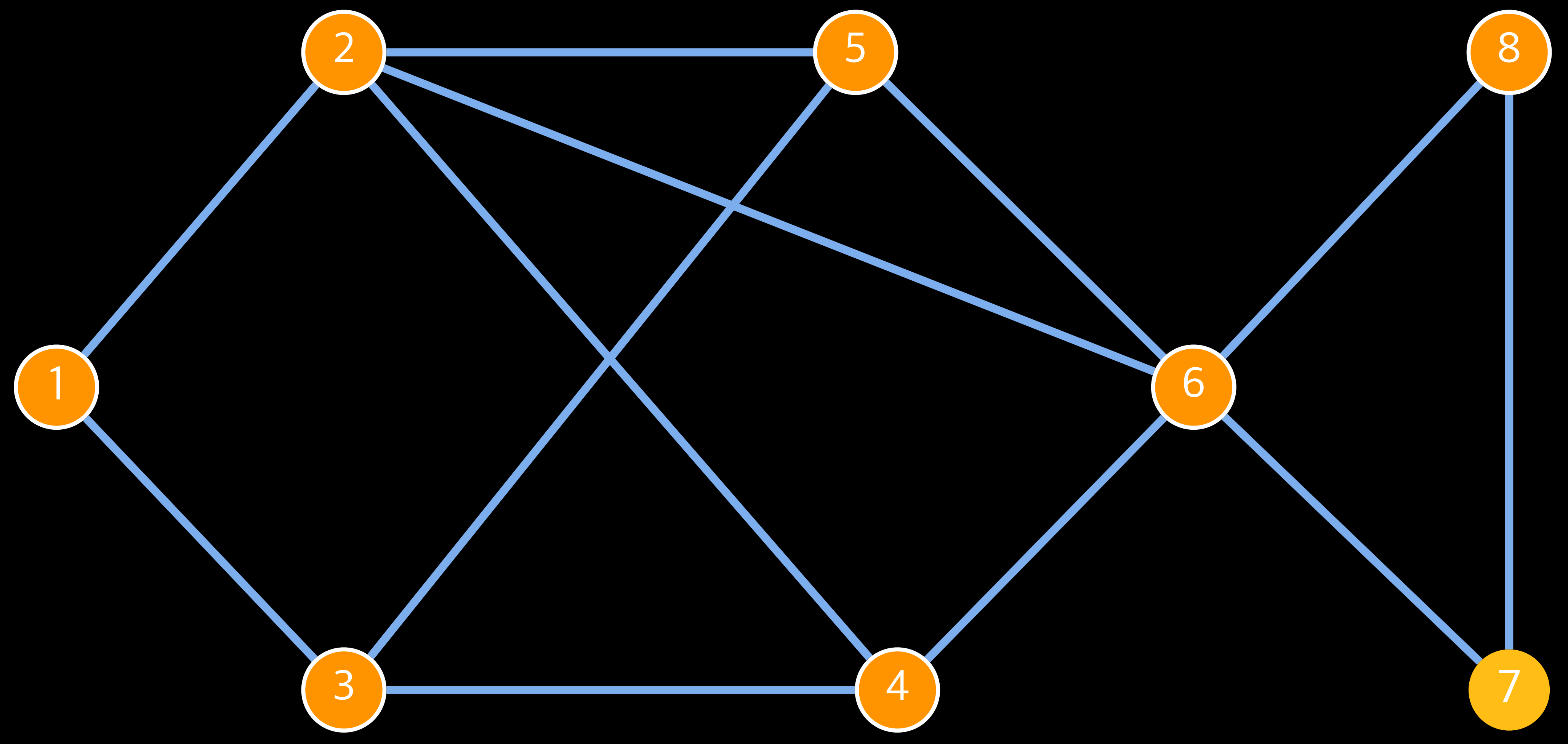
1	2	3	4	5	6	7	8
1	1	1	1	1	1		

Queue

3	4	5	6				

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

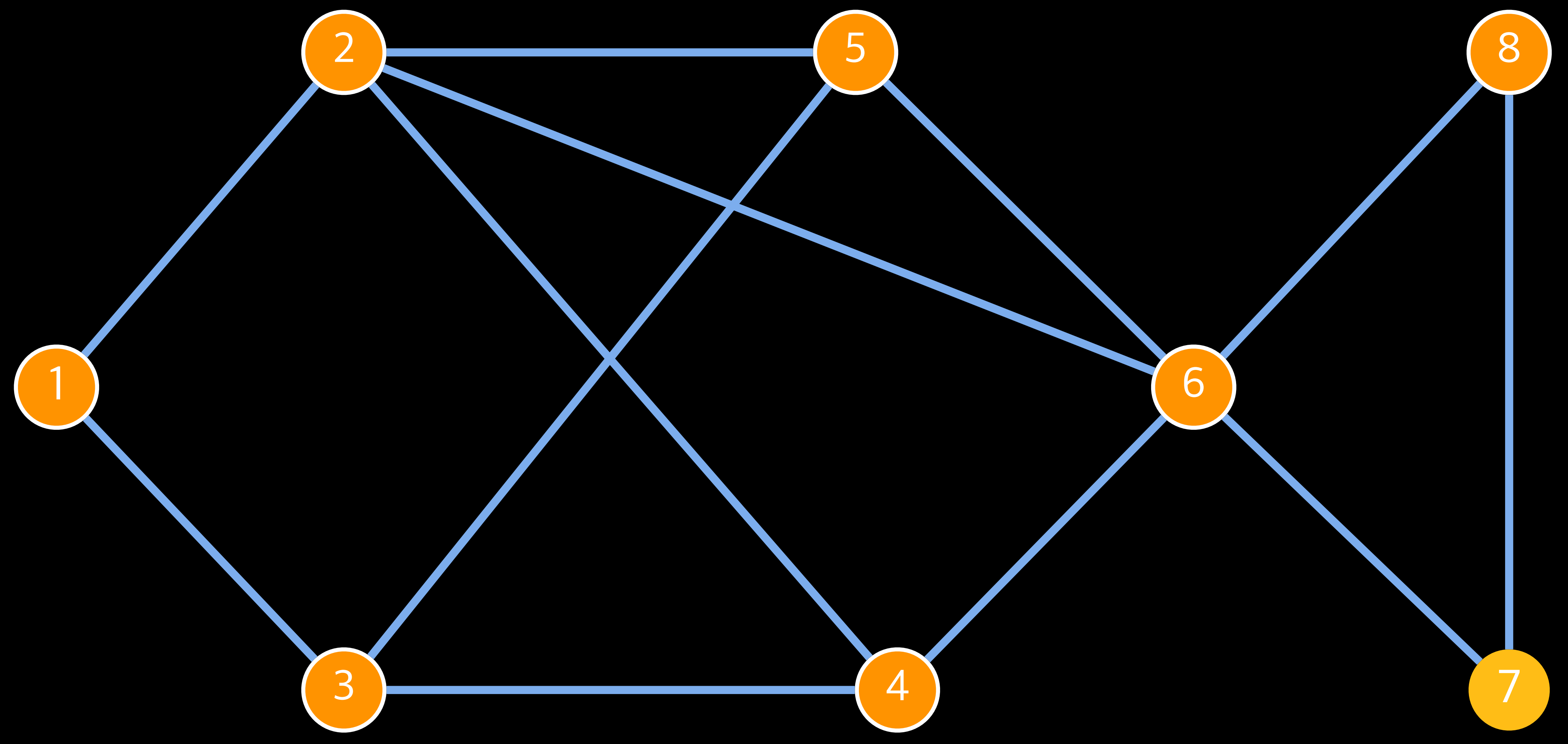
1	2	3	4	5	6	7	8
1	1	1	1	1	1		

Queue

4	5	6					

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Queue

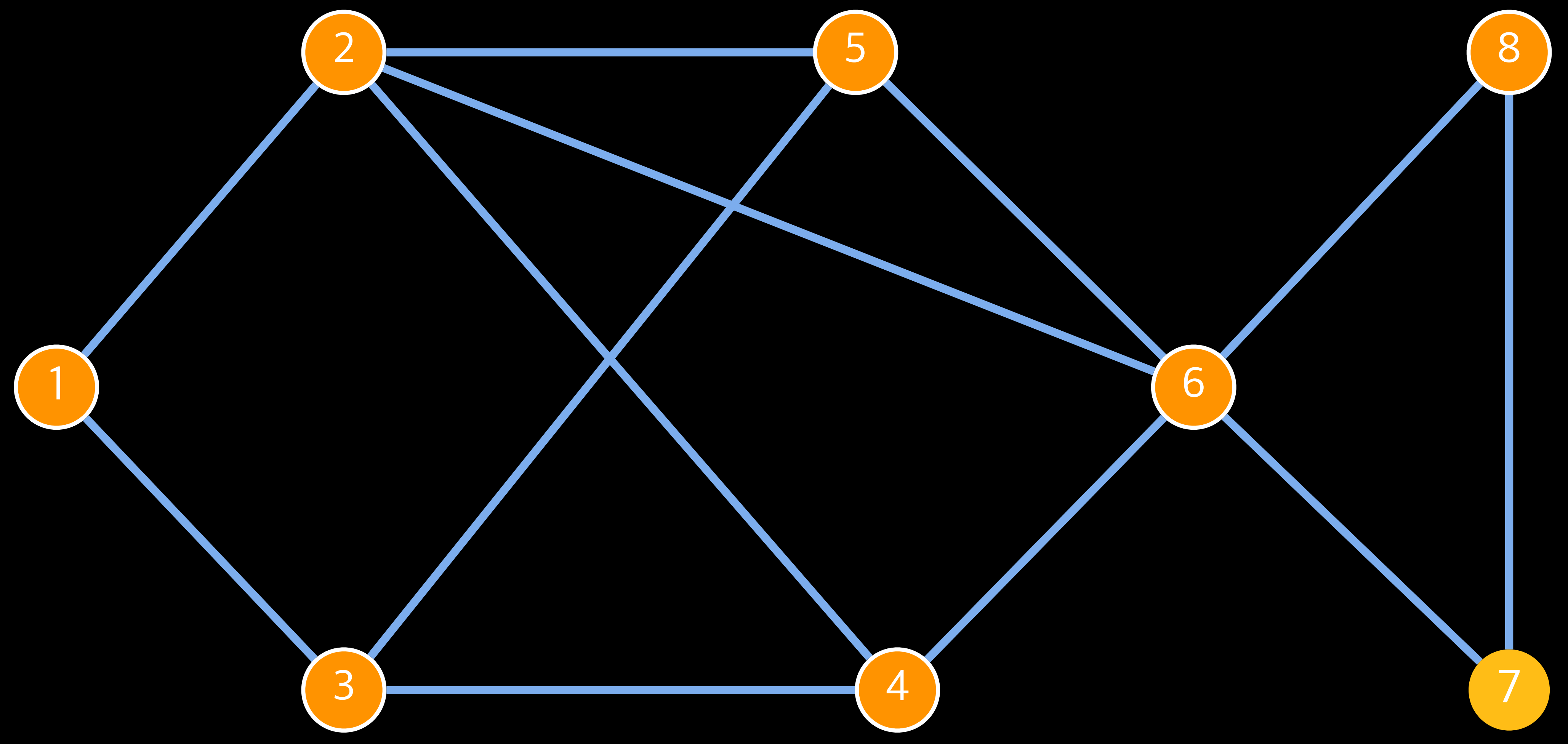
5	6						

Visited Nodes

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

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

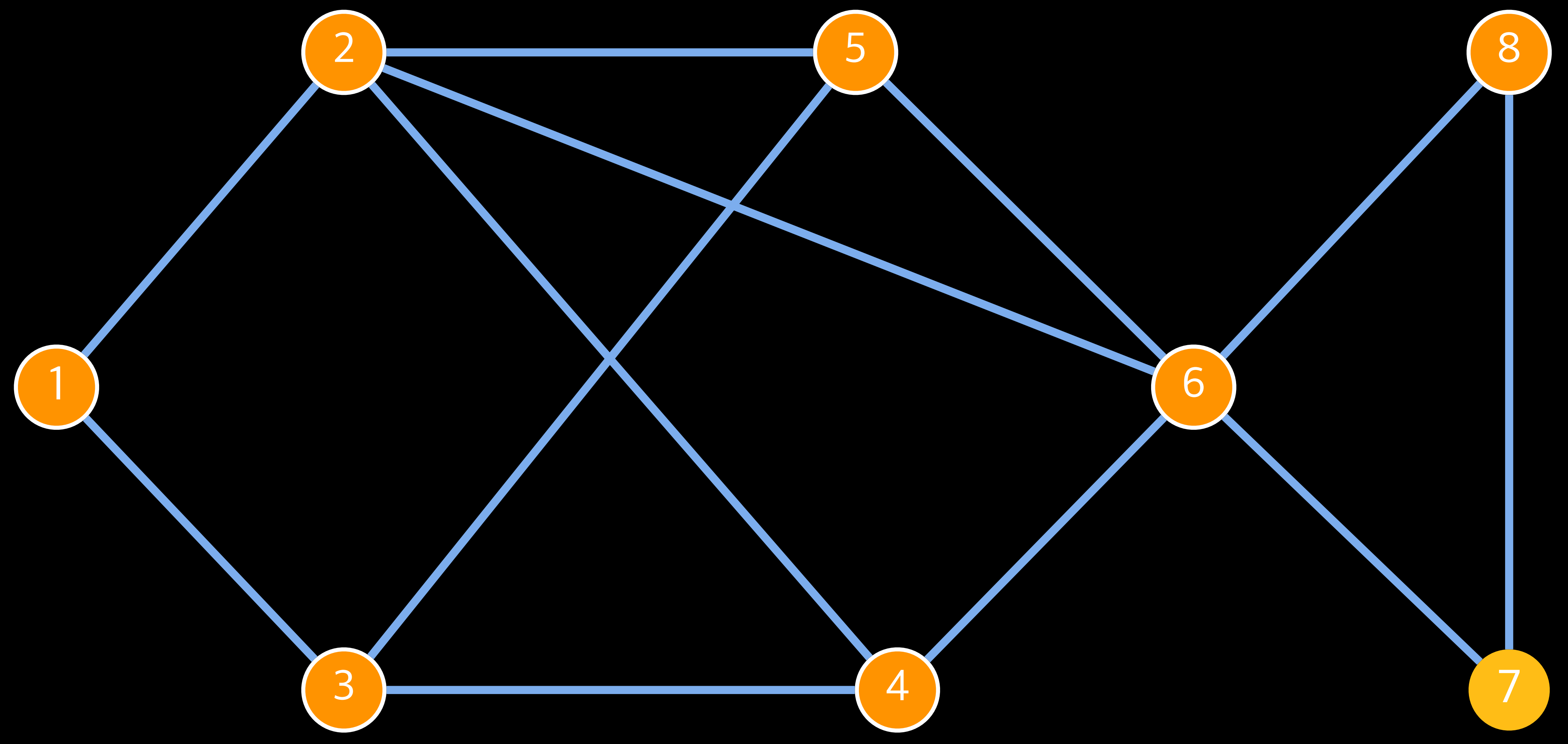
1	2	3	4	5	6	7	8
1	1	1	1	1	1		

Queue

6							

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Queue

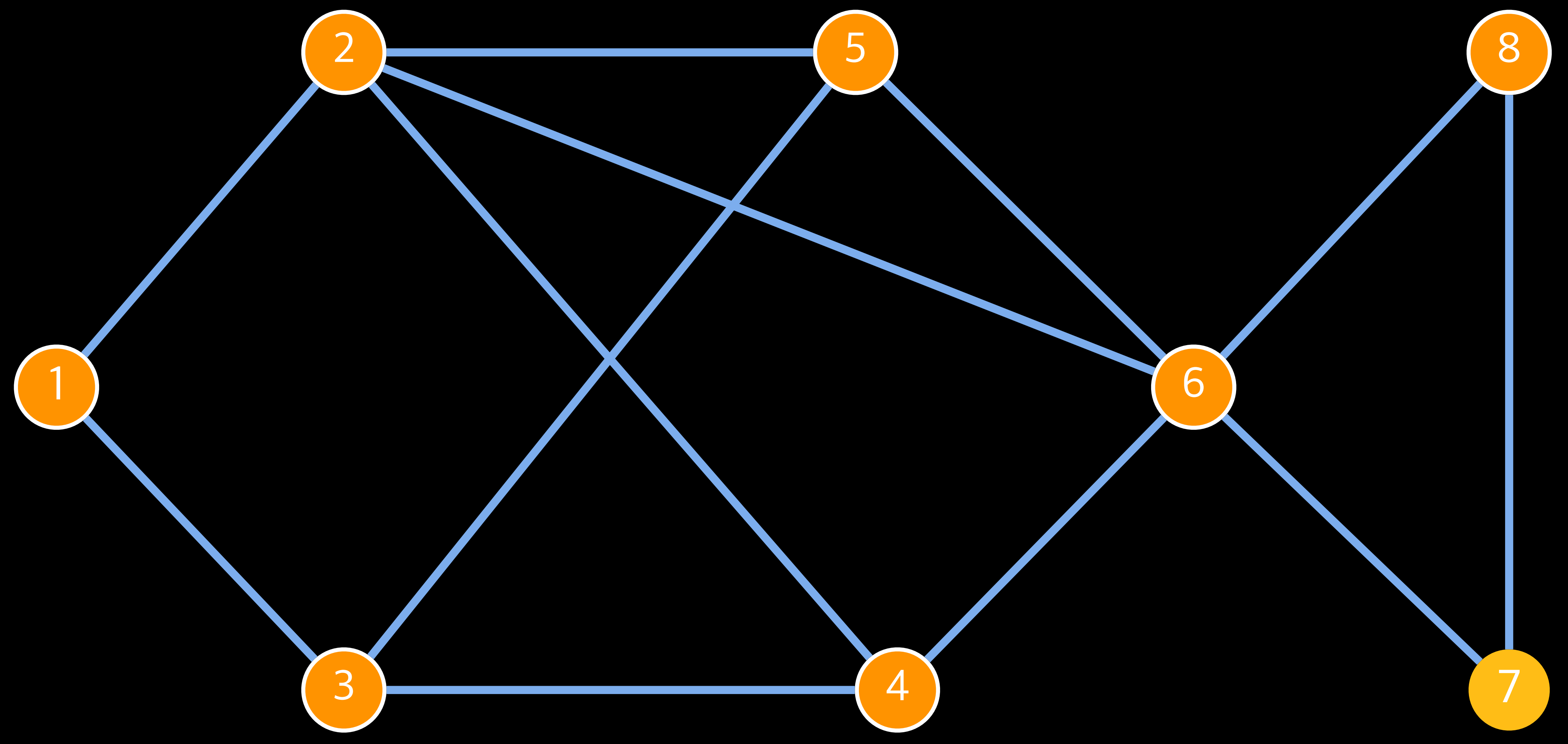
7							

Visited Nodes

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

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Queue

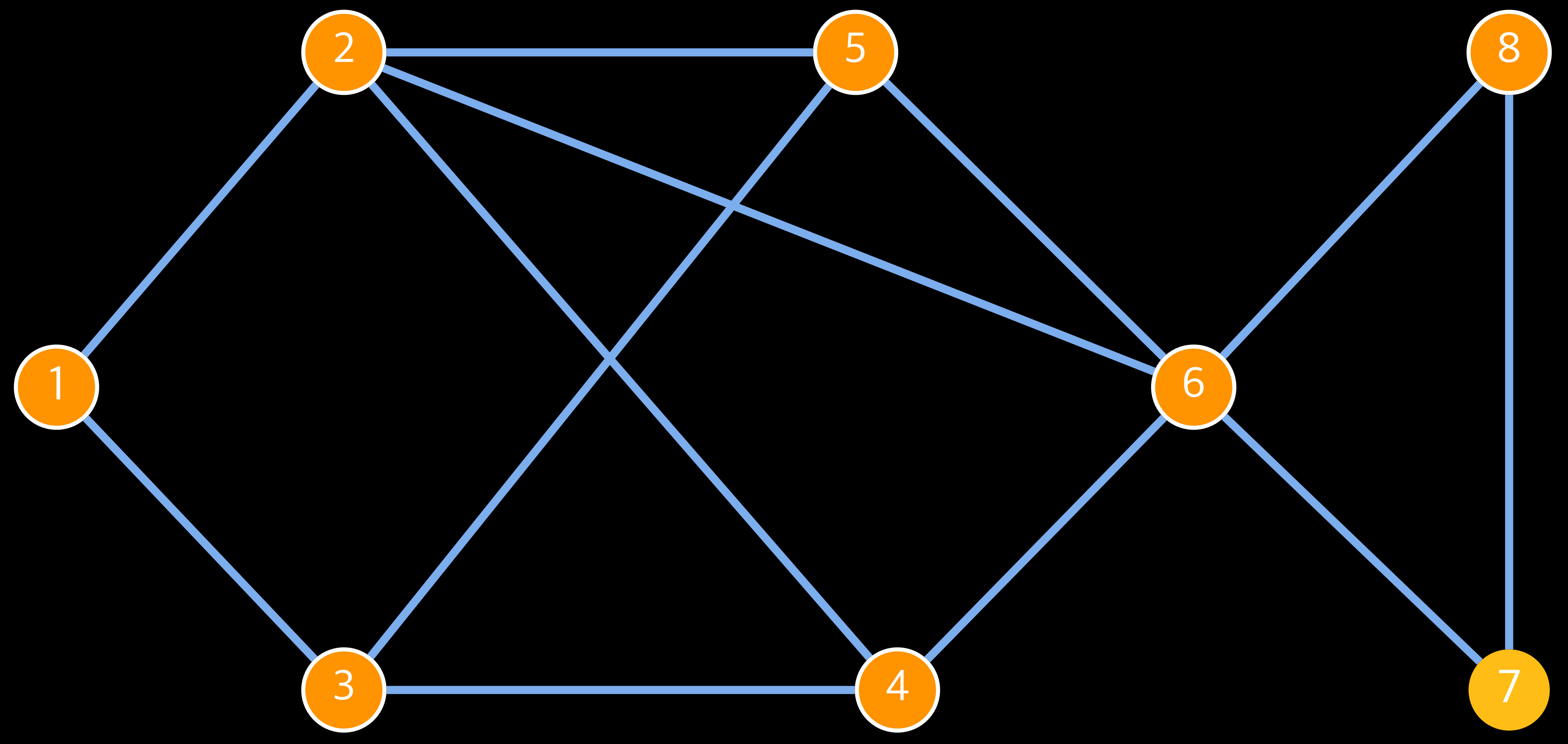
7	8						

Visited Nodes

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

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



Visited Nodes

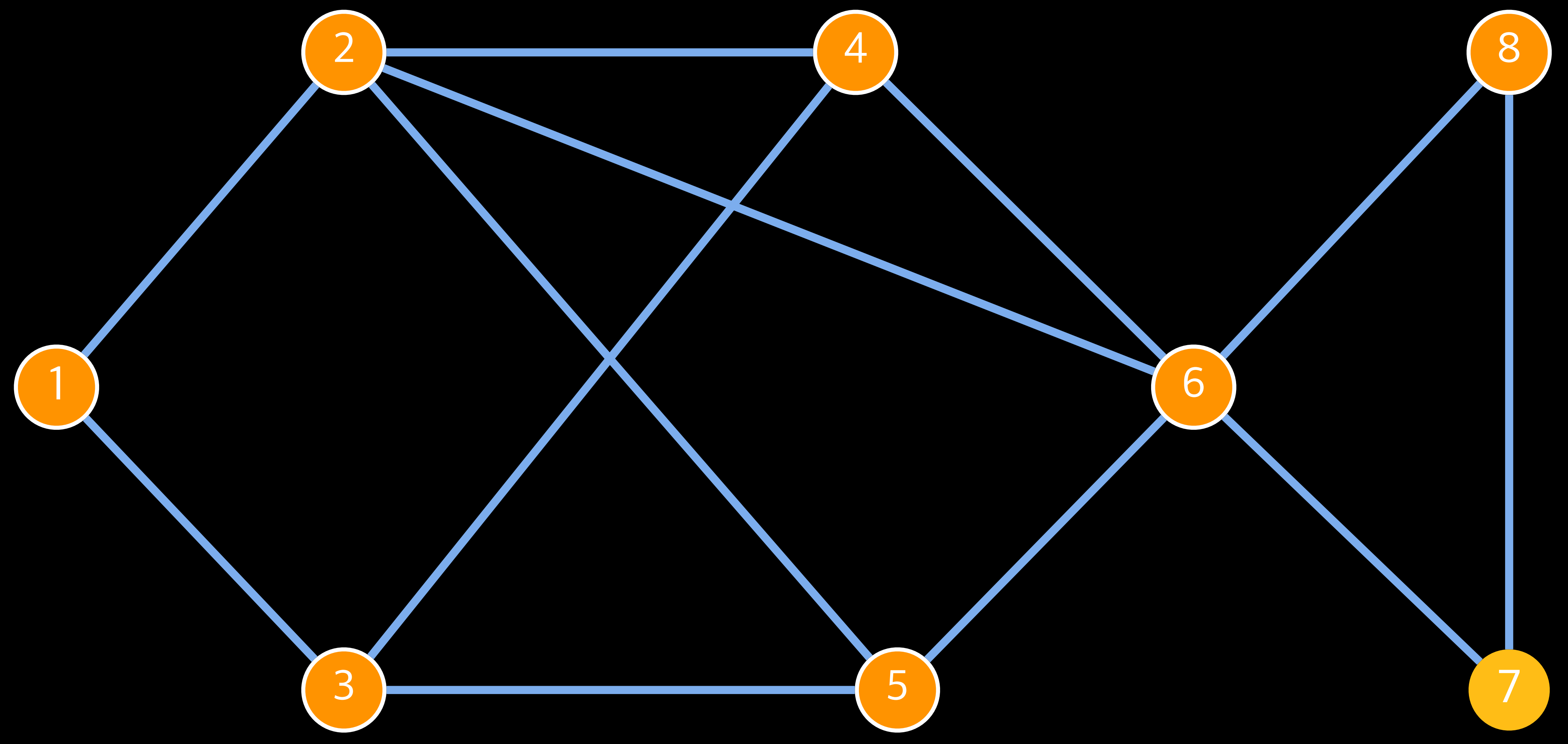
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

Queue

8							

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



Visited Nodes

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

Queue

DFS ALGORITHM

$\mathit{function}\ breadth\_first\_search(start\_node)$

$visited \leftarrow$ a set to store references to all visited nodes

$queue \leftarrow$ a queue to store references to unexplored nodes

$queue\,.\,enqueue(start\_node)$

$visited\,.\,append(start\_node)$

$\mathit{While}\ queue \neq \phi$

$current\_node \leftarrow queue\,.\,dequeue()$

$process(current\_node)$

$\mathit{For}\ neighbour \in current\_node\,.\,neighbours$

$\mathit{If}\ neighbor \notin\ visited$

$queue\,.\,enqueue(neighbour)$

$visited\,.\,append(neighbour)$

COMPLEXITY OF BFS

Runtime Analysis

- Let $|V|$ and $|E|$ be the number od vertices and edges, respectively
- Every $v \in V$ is enqueued and processed exactly once, resulting in $O(|V|)$time
- Every edge is checked exactly once in the for loop resulting in $O(|E|)$ time

The Complexity is $O(|V| + |E|)$

COMPLEXITY OF BFS

Runtime Analysis

- Each vertex enters Q exactly once
- If the graph is connected, while loop to check Q iterated n times
- For each j extracted from Q, need to examine all neighbours of j
- Rows are scanned n times in the adjacency matrix

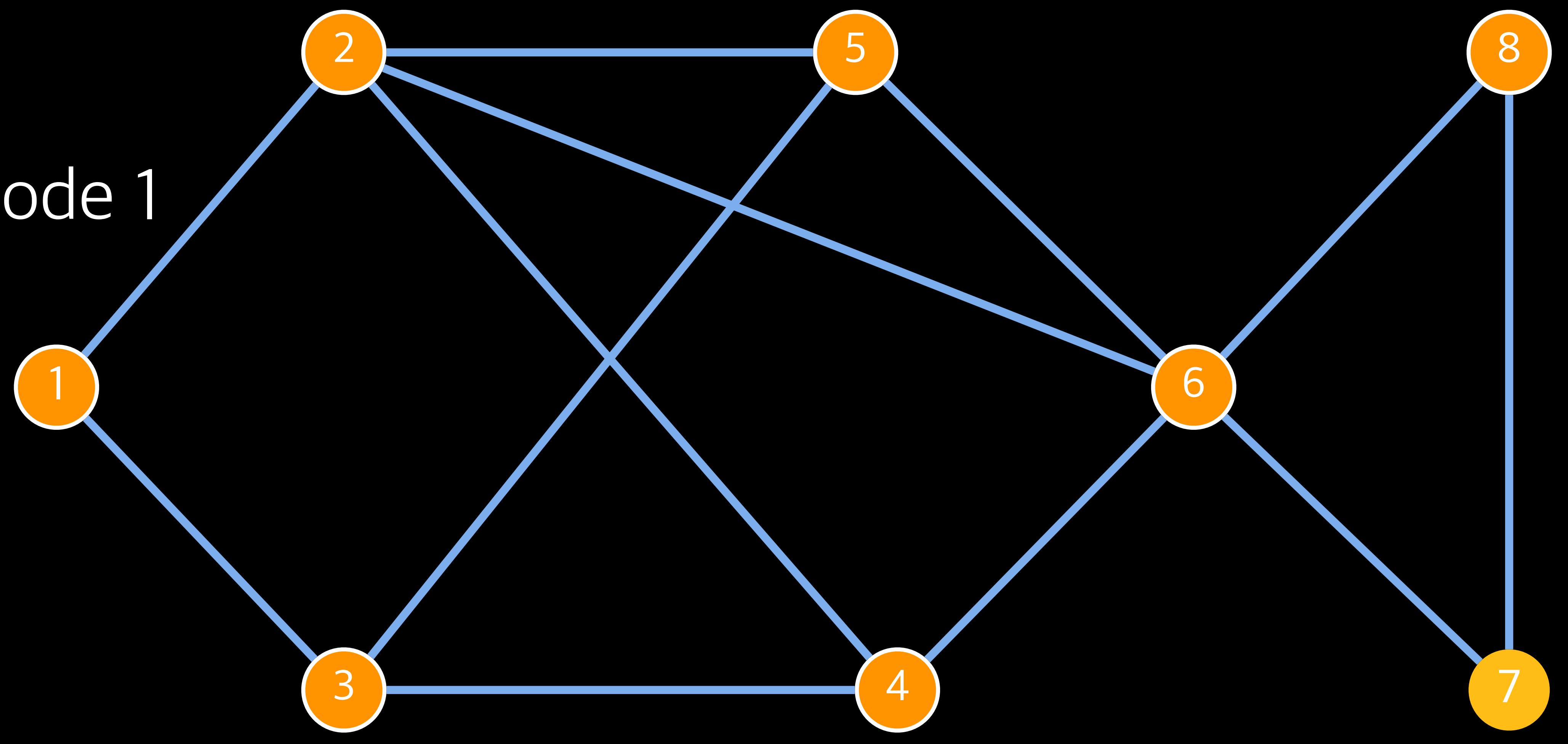
The complexity is $O(n^2)$

BFS - FIND THE PATH

Can we identify the path traversed from the DFS Algorithm present in slide #

BREADTH-FIRST TRAVERSAL

Starting at node 1



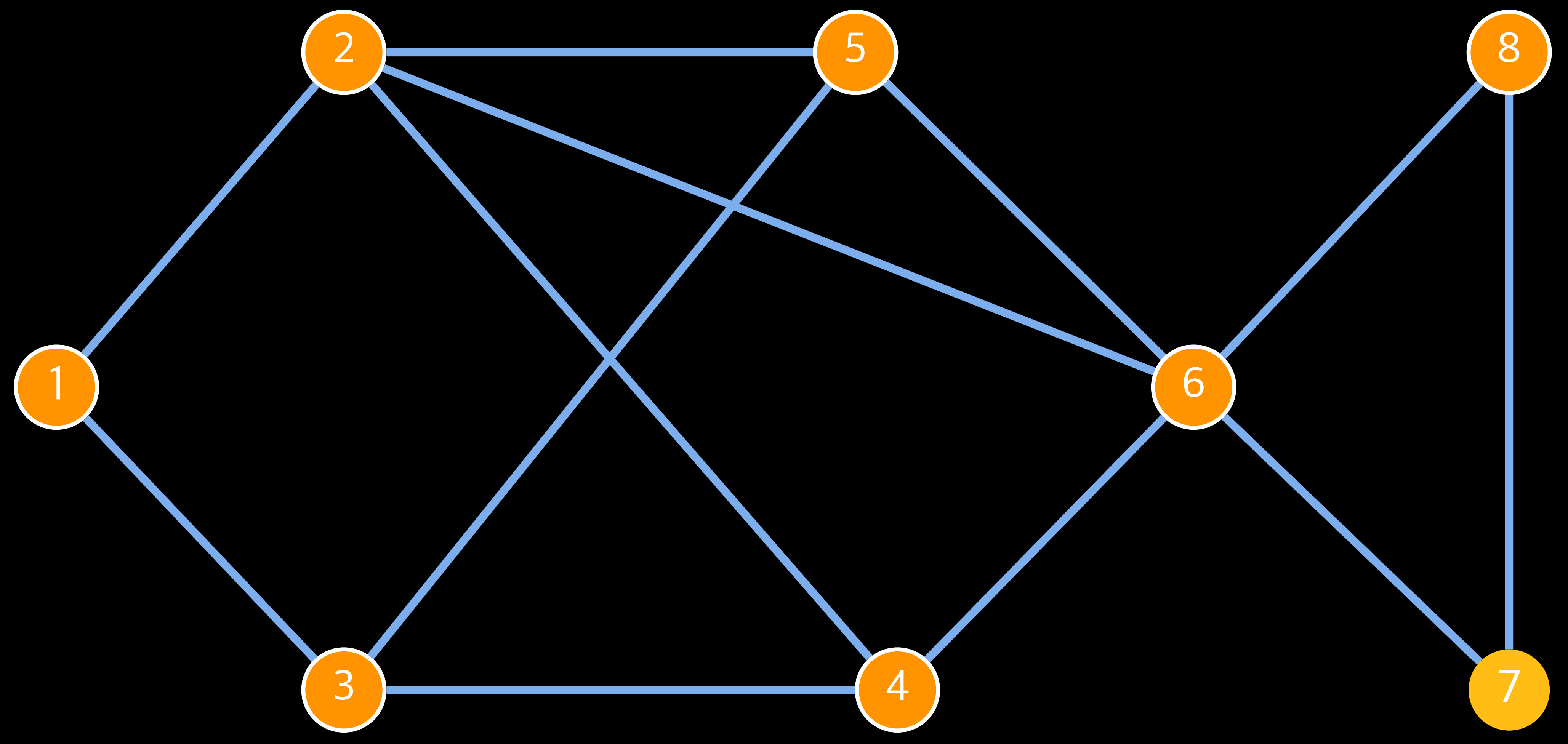

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```

Queue

	1	2	3	4	5	6	7	8
L								
P								

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



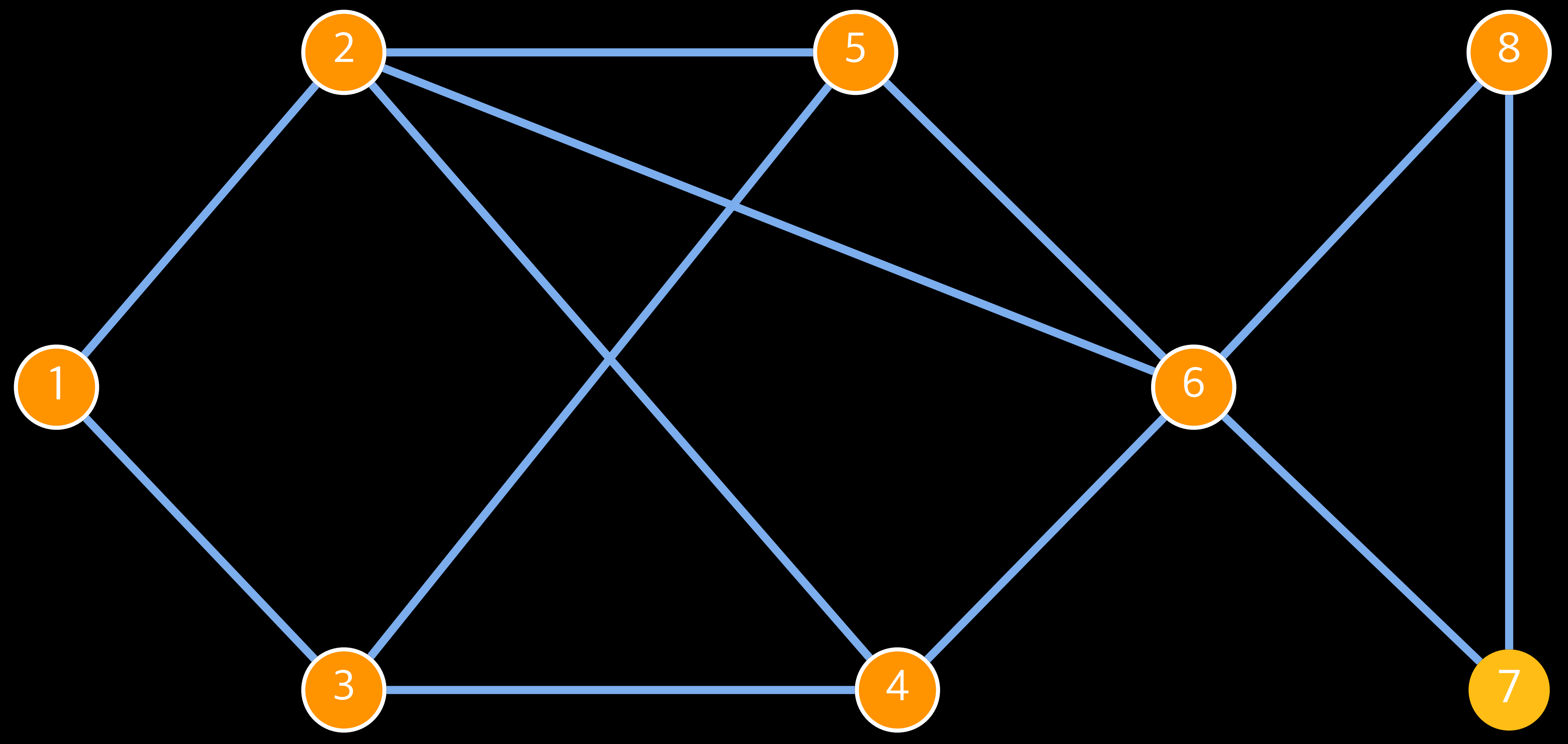
Queue

1							

	1	2	3	4	5	6	7	8
L	0							
P	-							

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



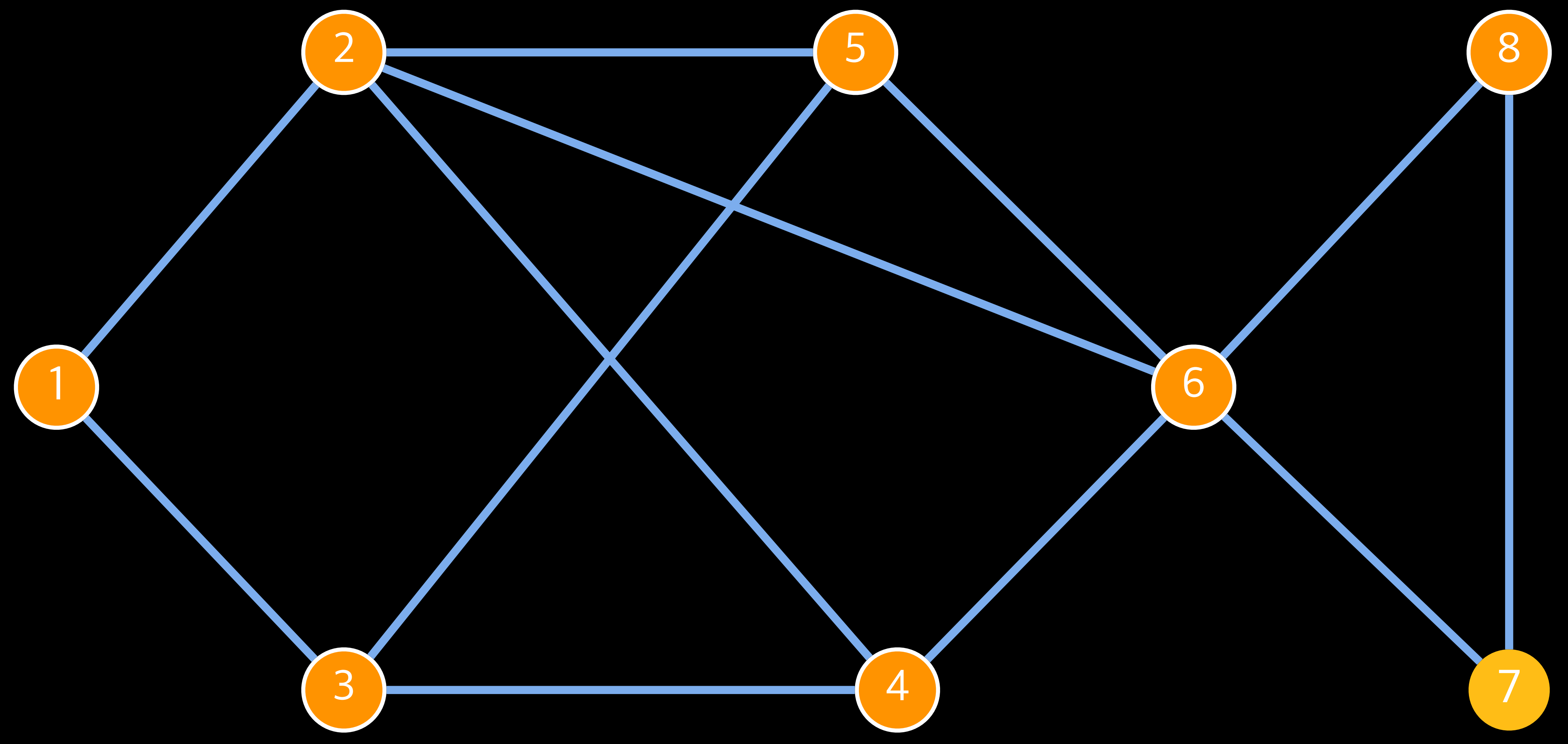
Queue

2							

	1	2	3	4	5	6	7	8
L	0	1						
P	-	1						

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



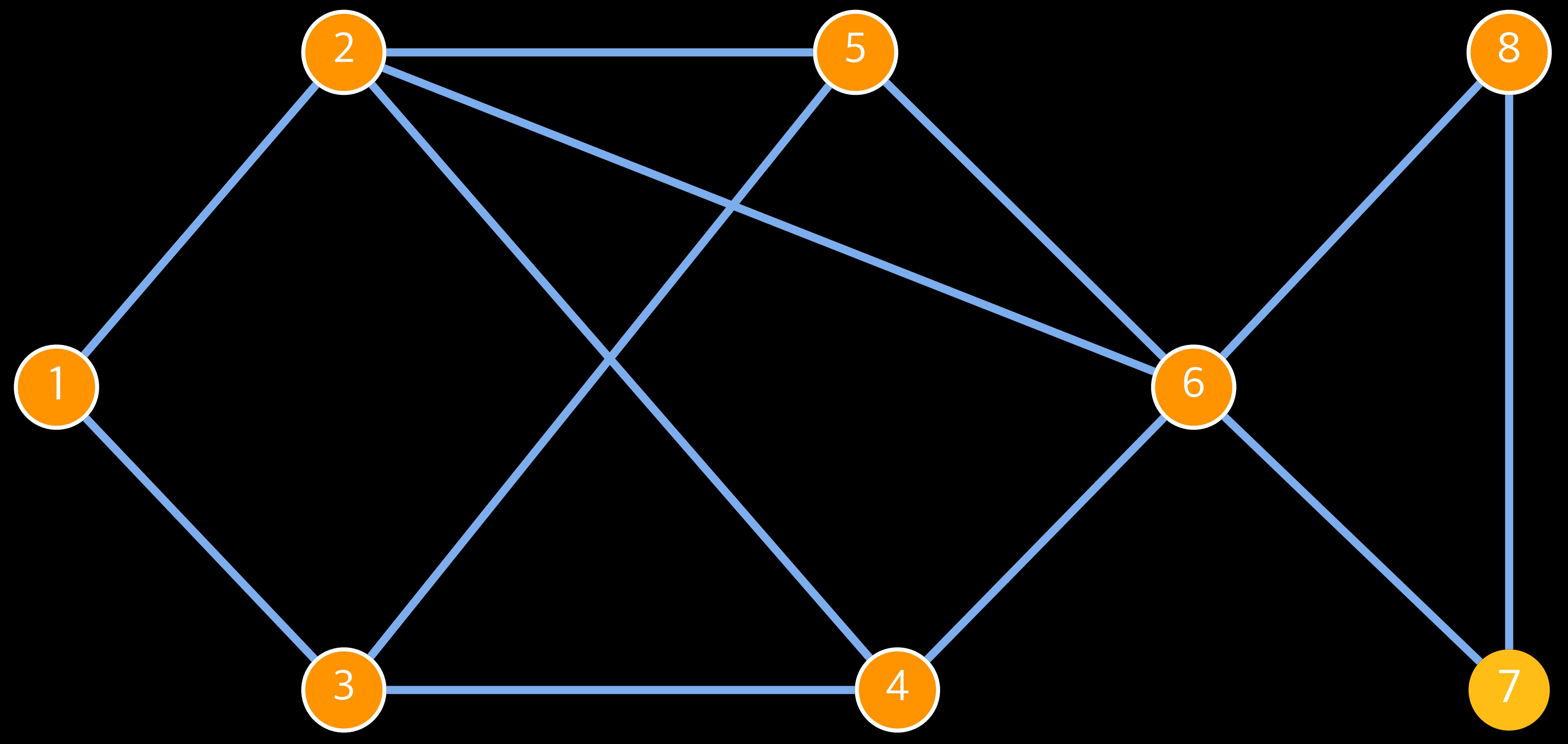
Queue

2	3						

	1	2	3	4	5	6	7	8
L	0	1	1					
P	-	1	1					

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



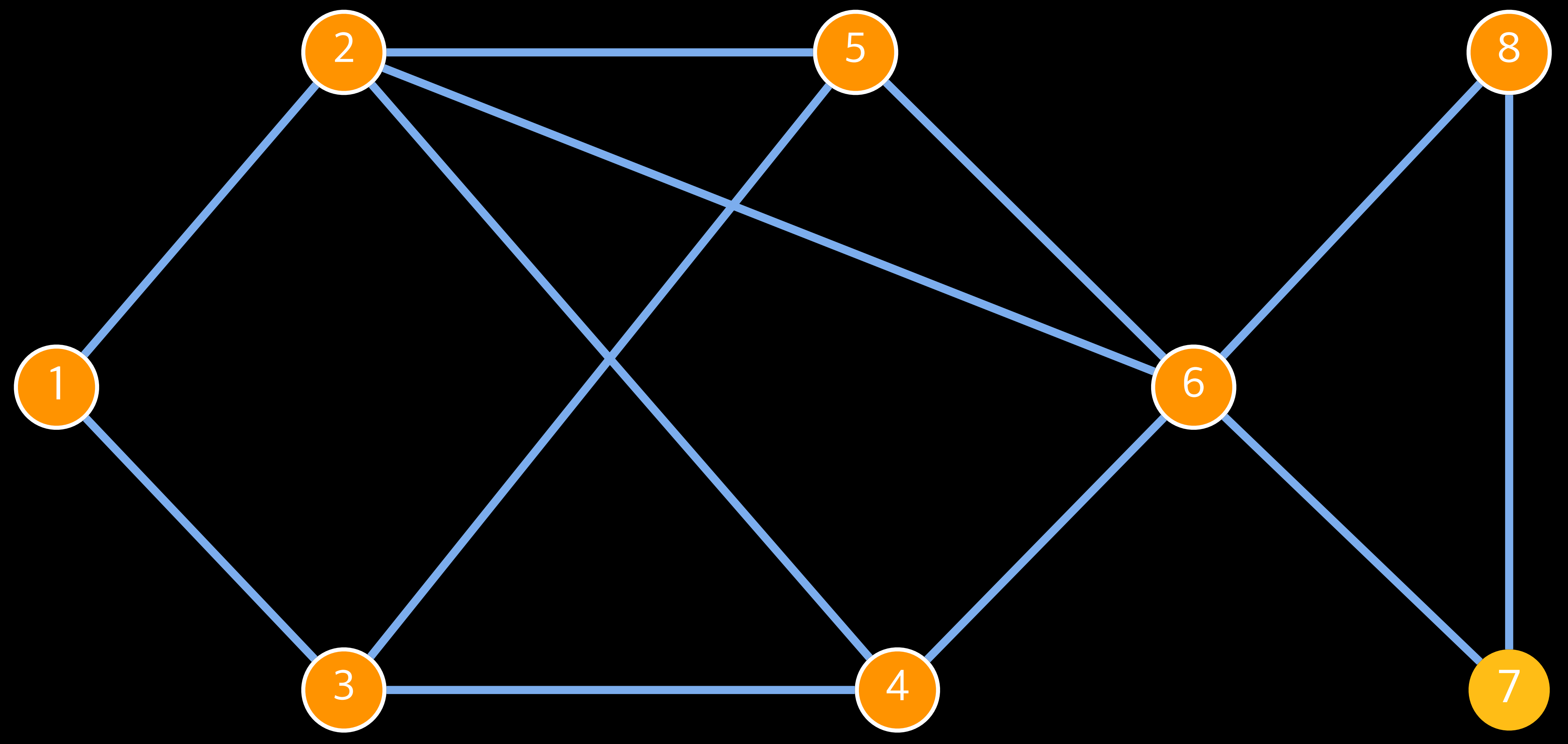
Queue

3	4						

	1	2	3	4	5	6	7	8
L	0	1	1	2				
P	-	1	1	2				

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



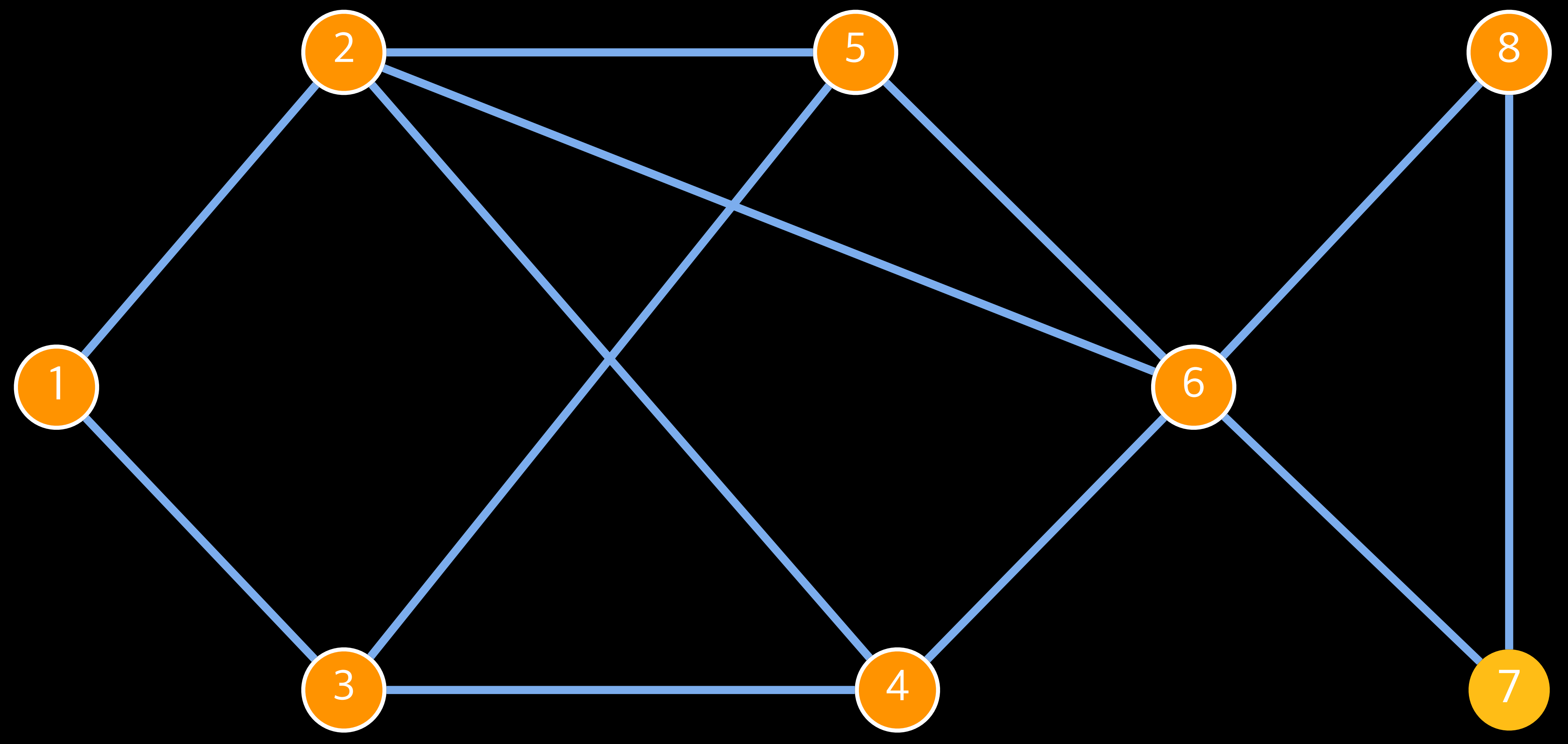
Queue

3	4	5					

	1	2	3	4	5	6	7	8
L	0	1	1	2	2			
P	-	1	1	2	2			

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



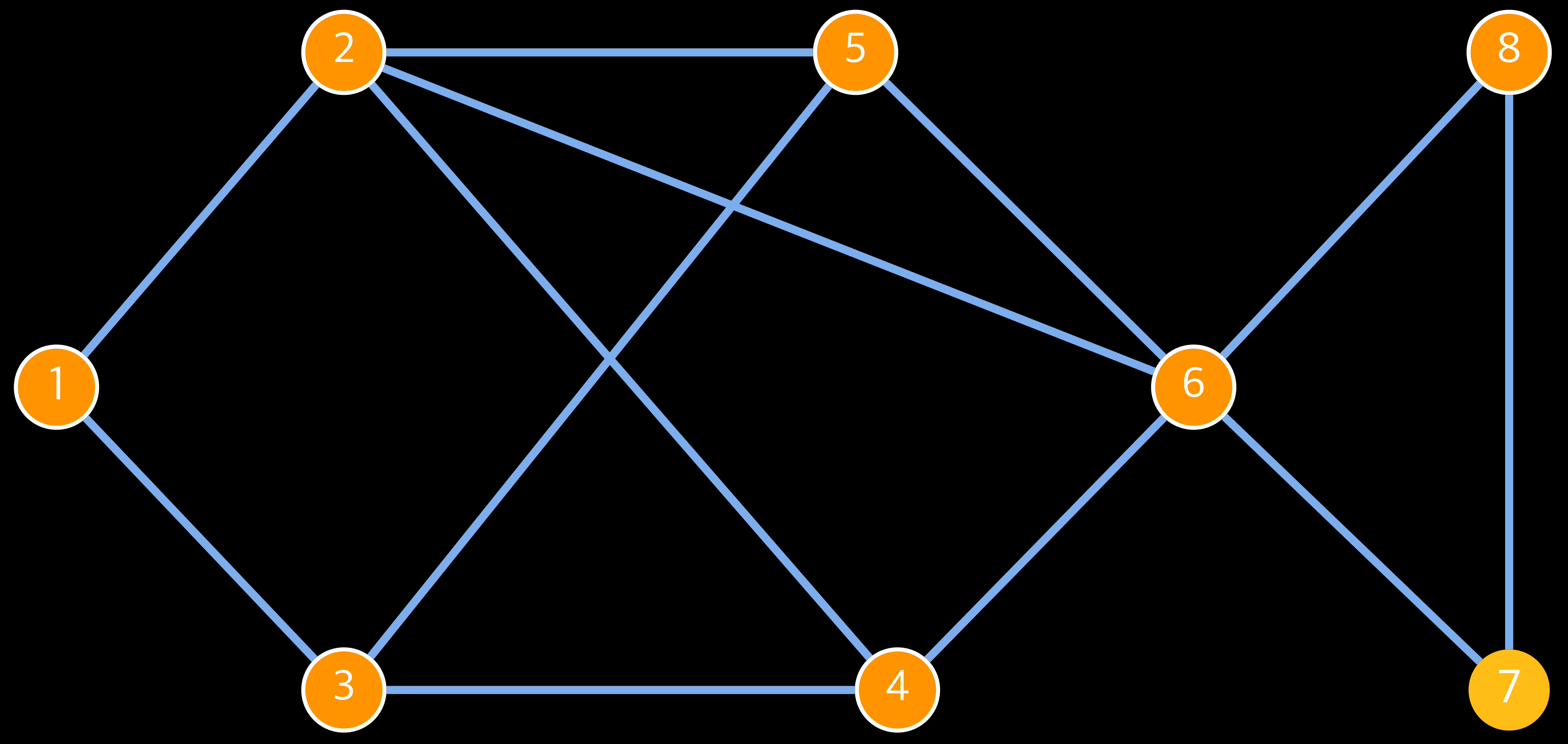
Queue

3	4	5	6				

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2		
P	-	1	1	2	2	2		

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



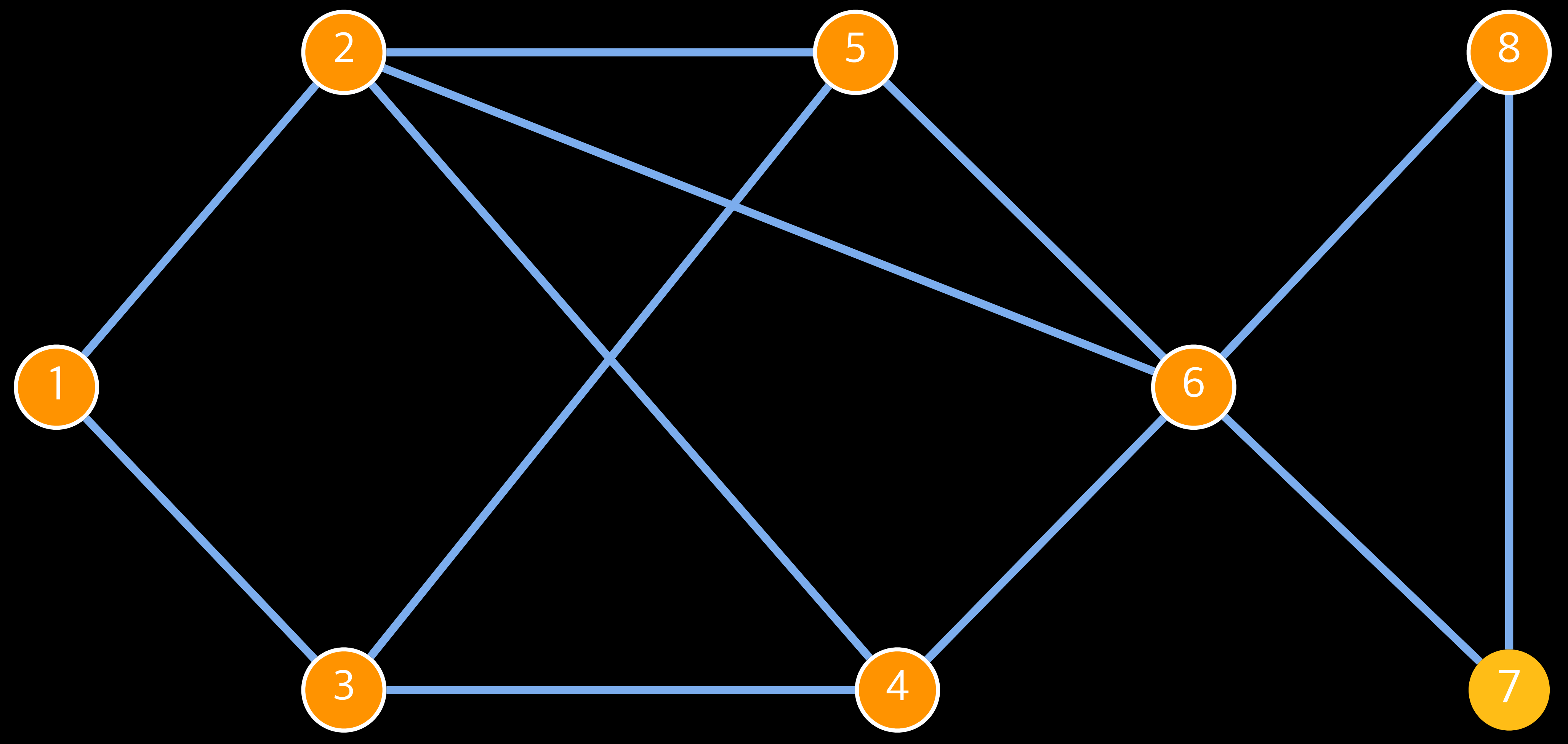
Queue

4	5	6					

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2		
P	-	1	1	2	2	2		

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



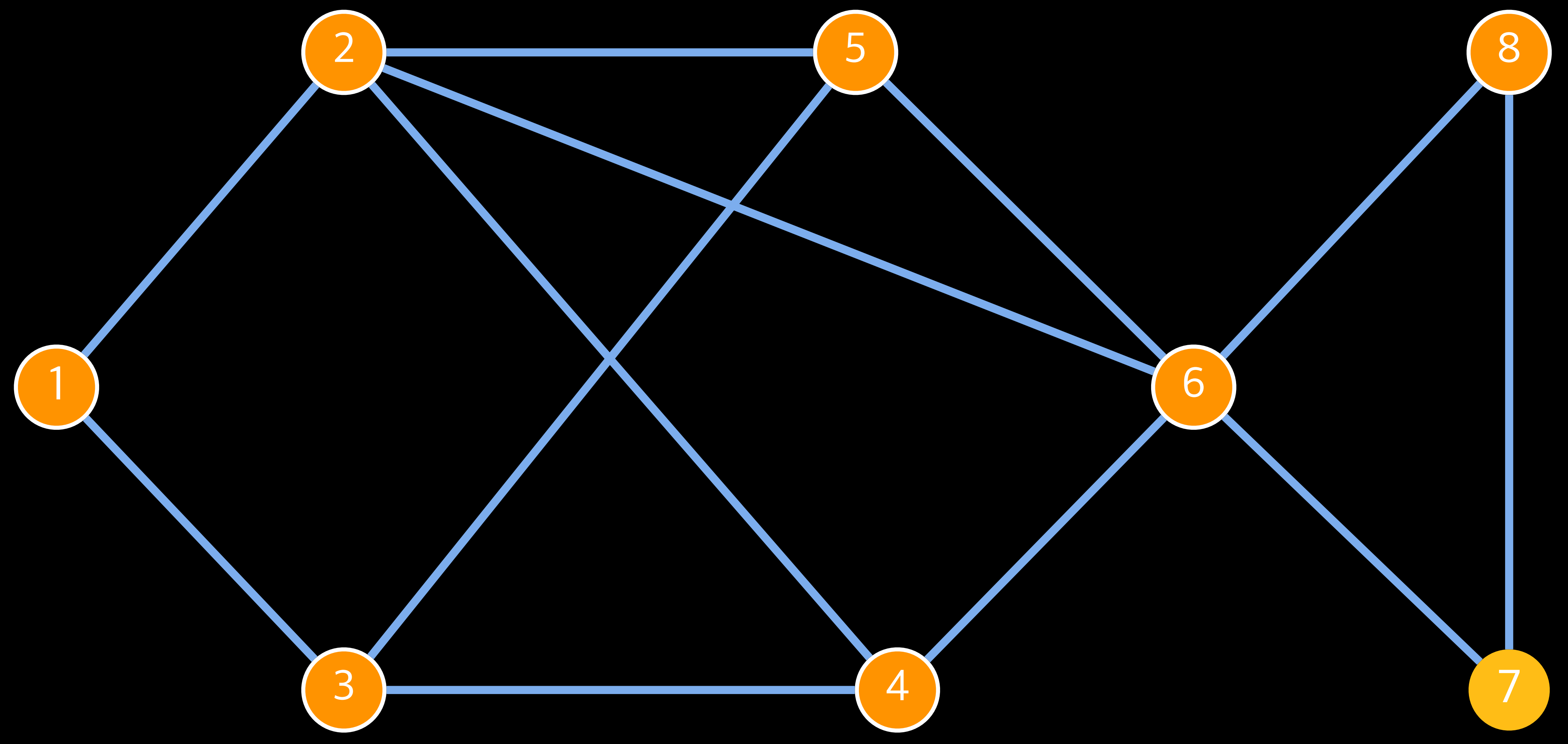
Queue

5	6						

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2		
P	-	1	1	2	2	2		

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



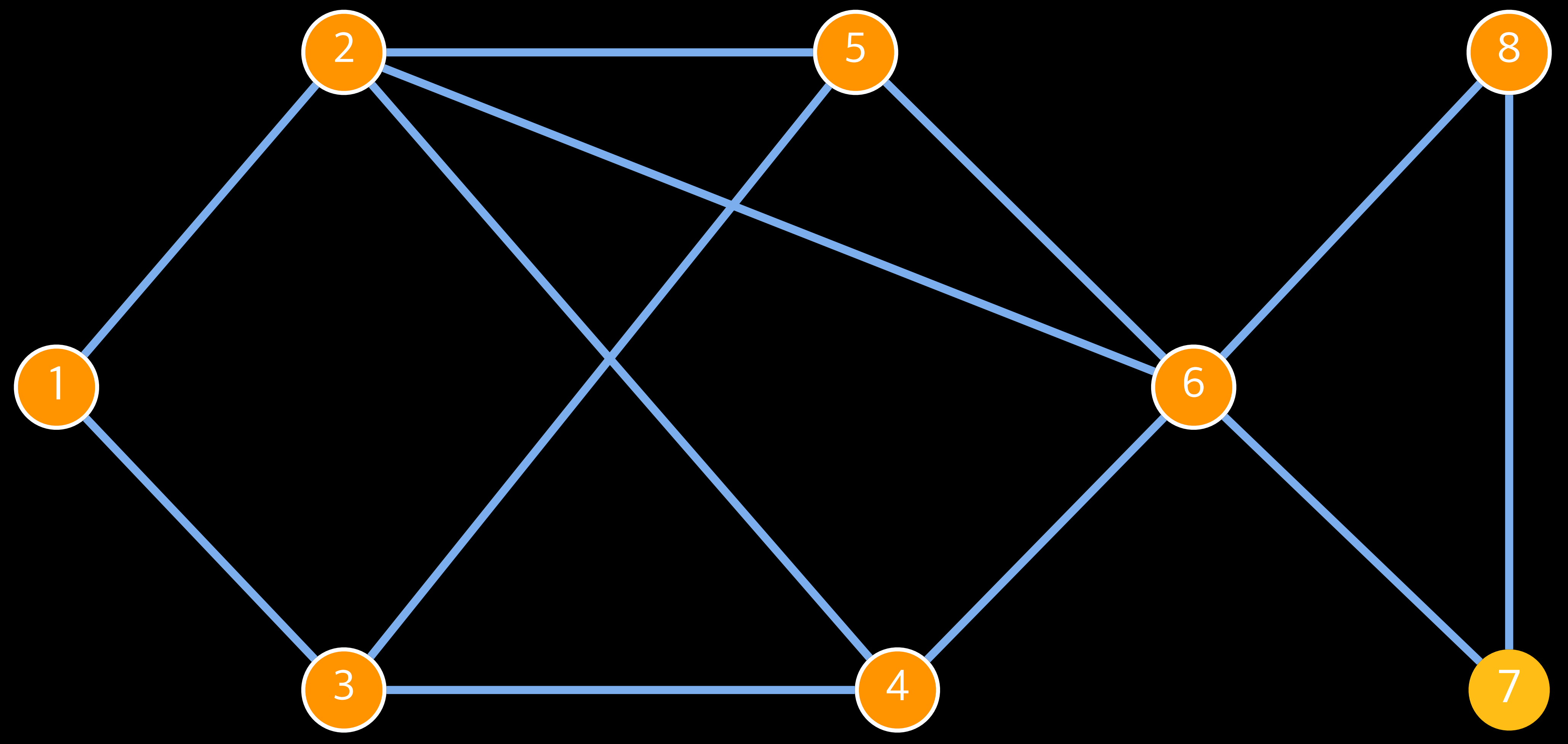
Queue

6							

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2		
P	-	1	1	2	2	2		

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```



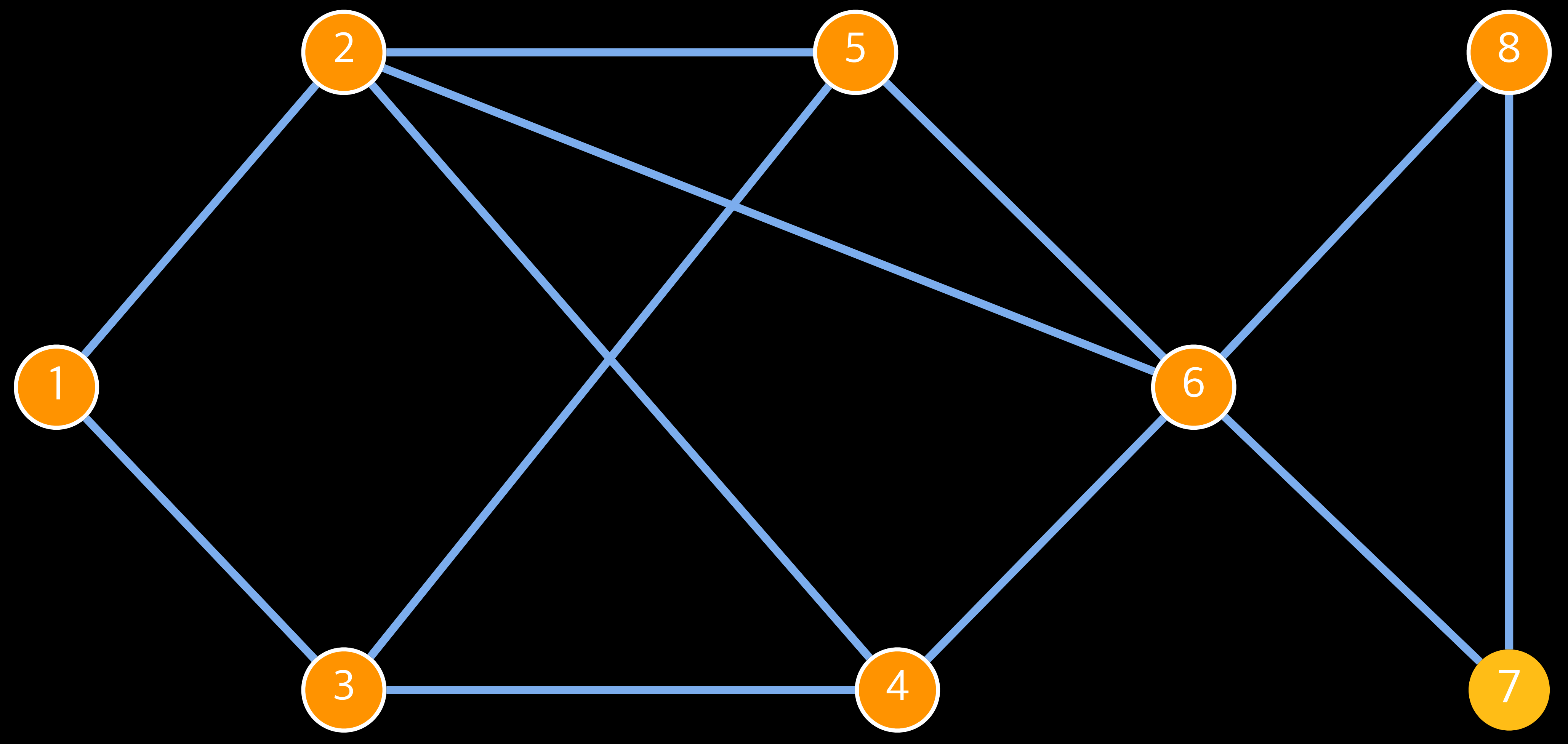
Queue

7							

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2	3	
P	-	1	1	2	2	2	6	

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```



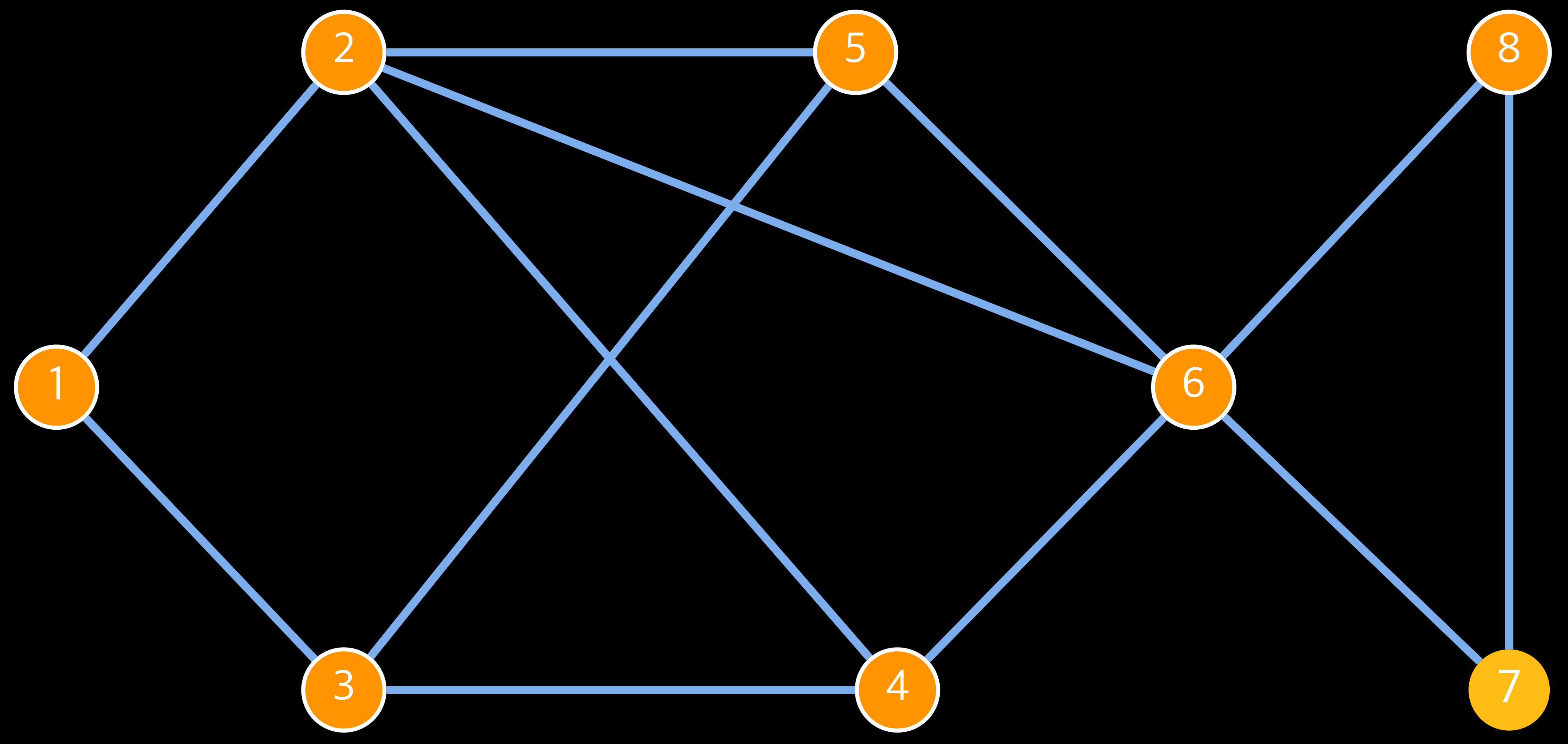
Queue

7	8						

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2	3	3
P	-	1	1	2	2	2	6	6

BREADTH-FIRST TRAVERSAL

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```

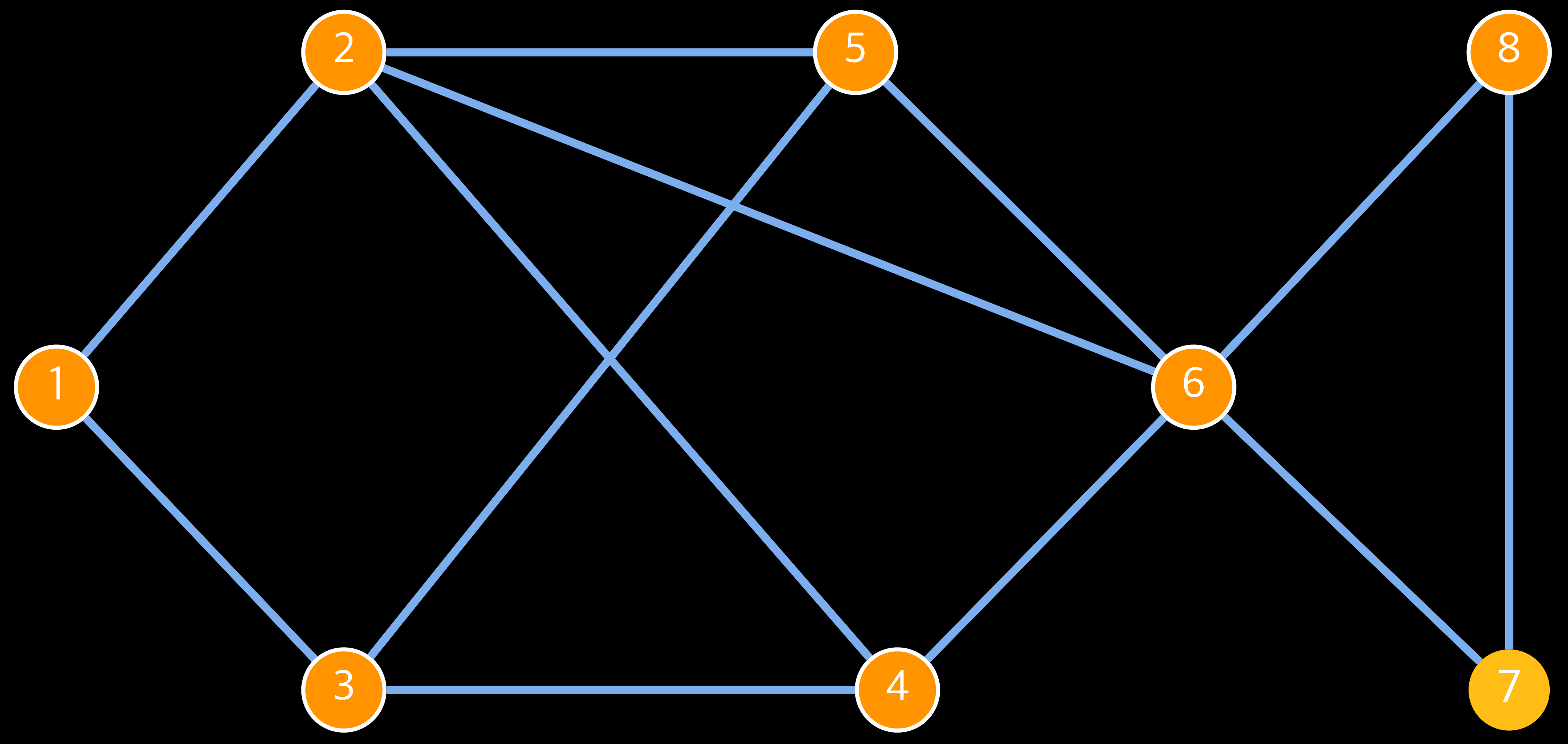


Queue

8							

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2	3	3
P	-	1	1	2	2	2	6	6

BREADTH-FIRST TRAVERSAL




```
for each edge (i,j)
   if visited[j] == 0
      visited[j] = 1
      append j to queue
```

Queue

	1	2	3	4	5	6	7	8
L	0	1	1	2	2	2	3	3
P	-	1	1	2	2	2	6	3

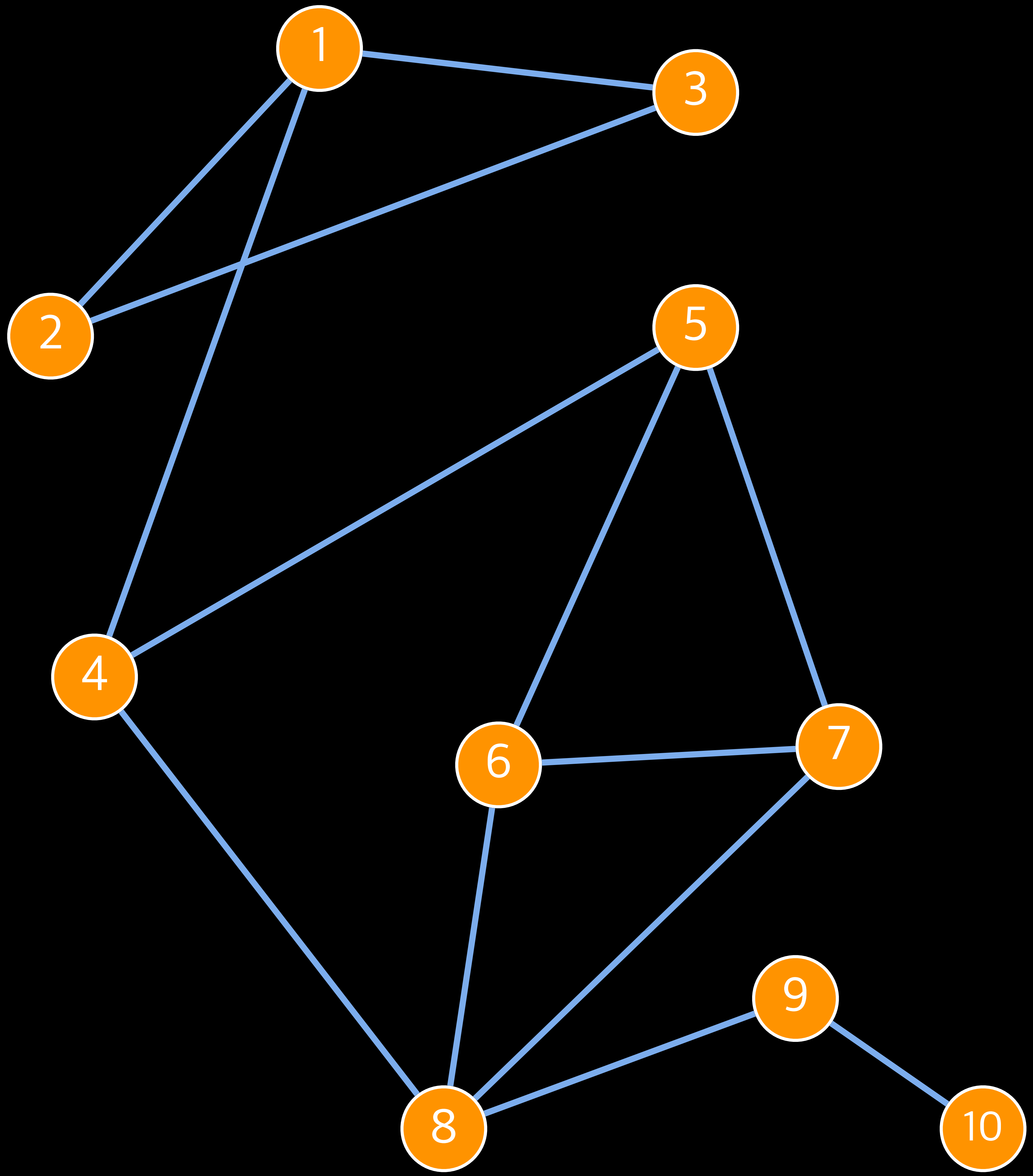
SHORTEST PATH

- BFS with $level[]$ gives the shortest path to each node in terms of number of edges
- The modified BFS computes shortest paths, iff $e_{jw} = 1, \forall e_{jw} \in E$
- If the edges are weighted money, time, distance and $\neq 1$, then the $level[]$ may not give the optimal shortest path in terms of the # of edges
  - OR, the path walked may not be the shortest in spite of small number of edges

DEPTH-FIRST SEARCH

- Start from i, visit a neighbour j
- Suspend the exploration of i and explore j instead
- Continue till there is no more unexplored neighbours for j
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a stack
  - LIFO: most recently suspended is checked first

DEPTH-FIRST SEARCH



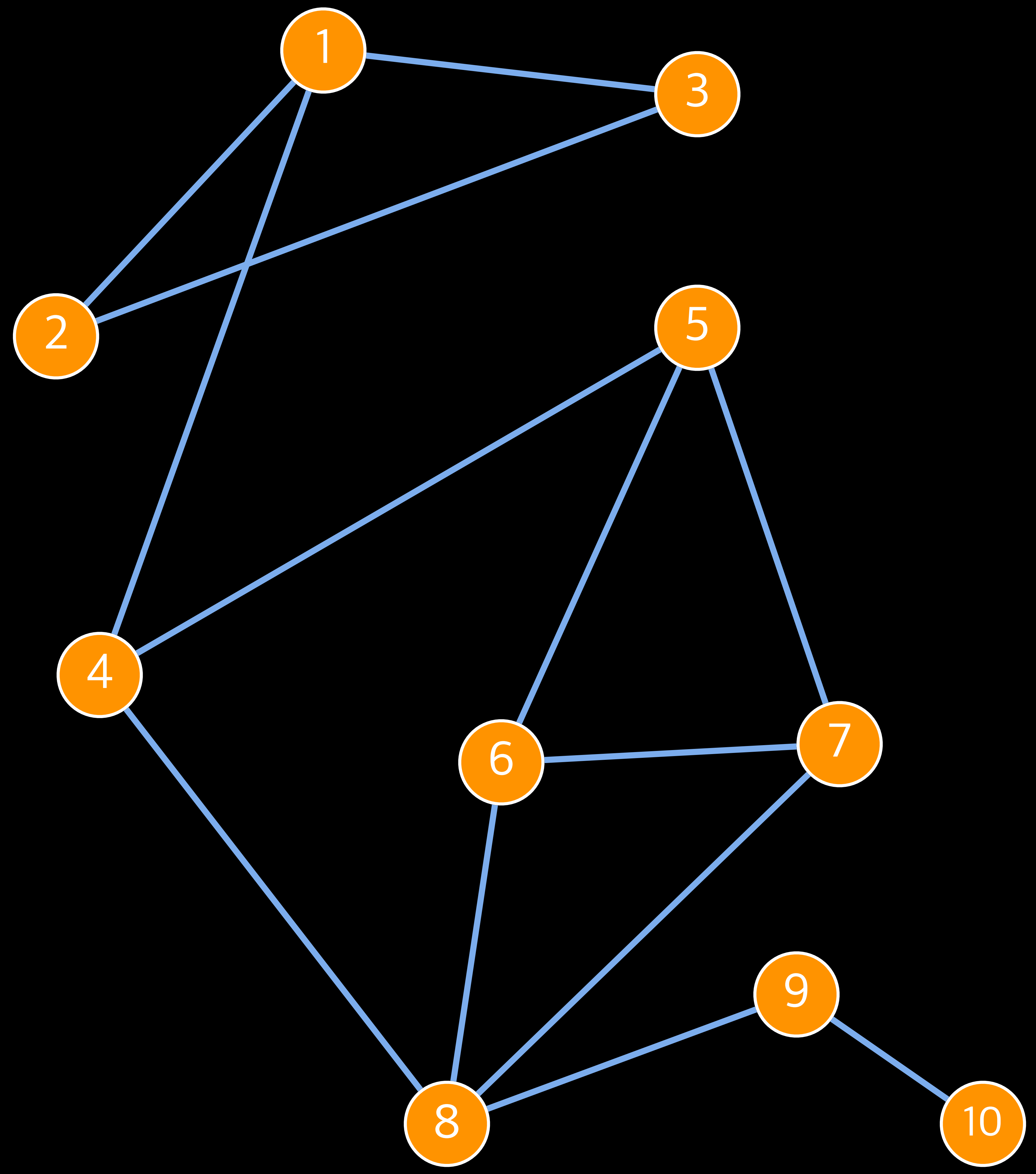
Start at the node 4

Stack of suspended vertices

Visited

1	2	3	4	5	6	7	8
			1				

DEPTH-FIRST SEARCH



Start at the node 4

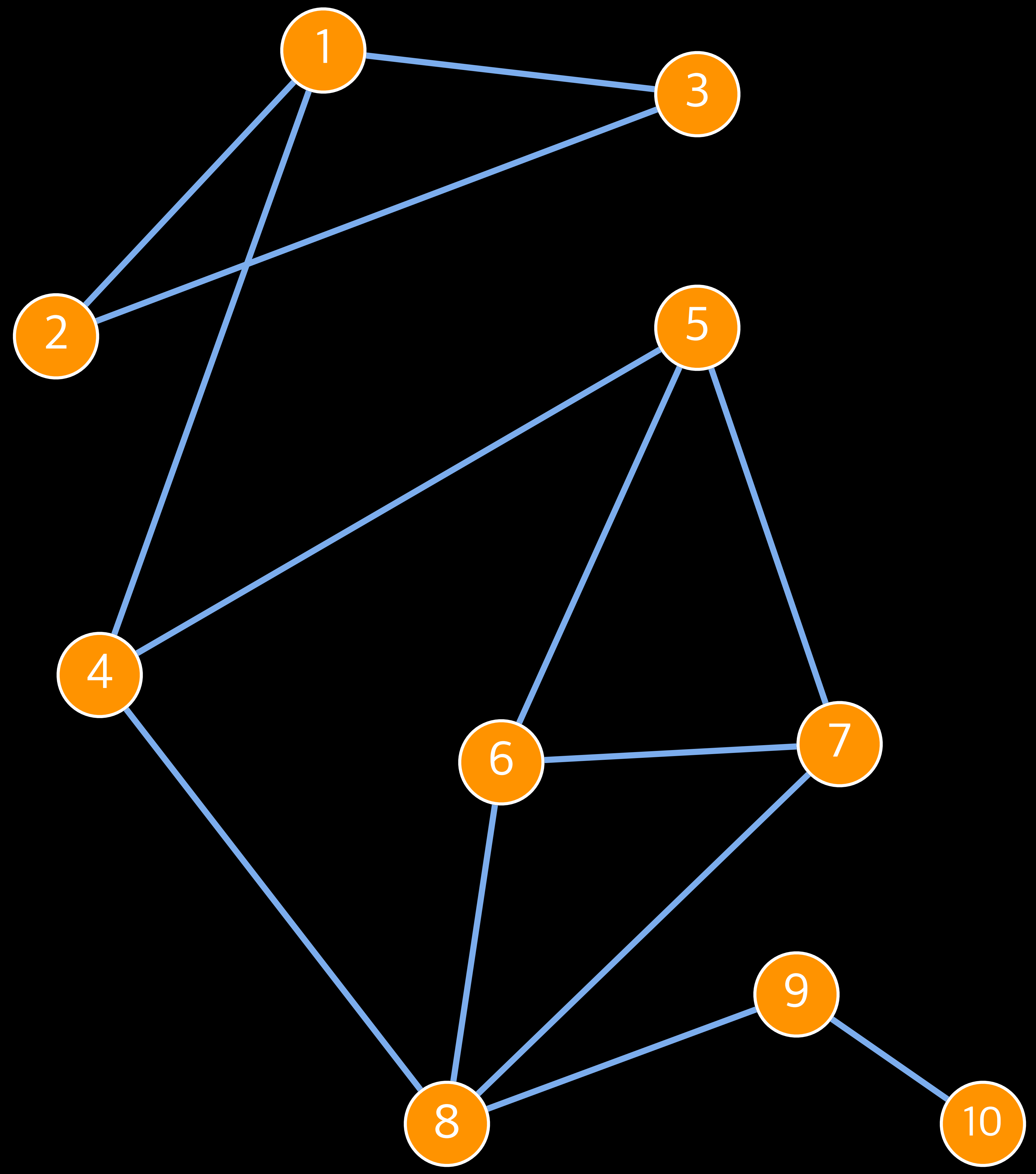
Stack of suspended vertices

4							

Visited

1	2	3	4	5	6	7	8
1			1				

DEPTH-FIRST SEARCH



Start at the node 4

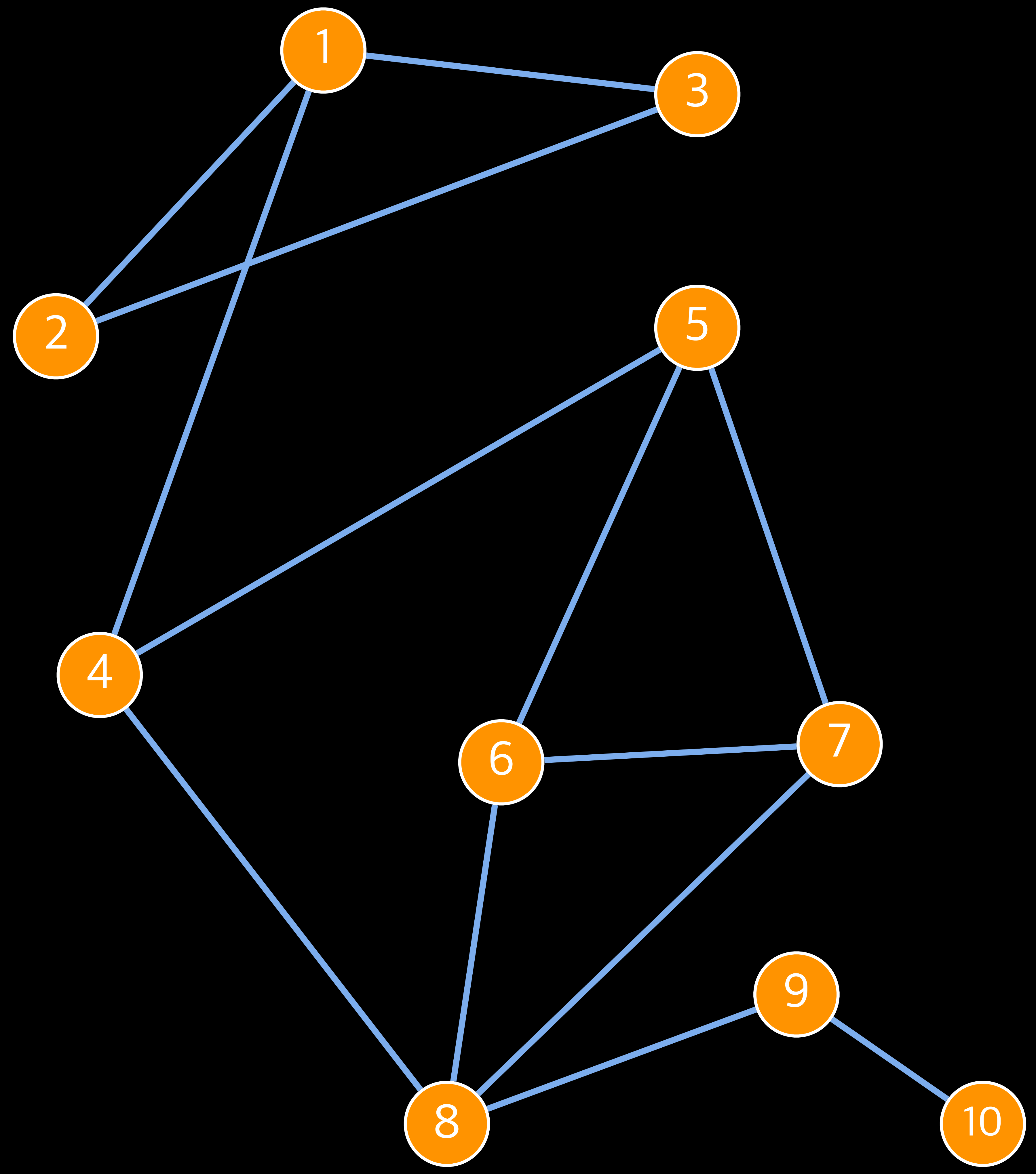
Stack of suspended vertices

4	1						

Visited

1	2	3	4	5	6	7	8
1	1		1				

DEPTH-FIRST SEARCH



Start at the node 4

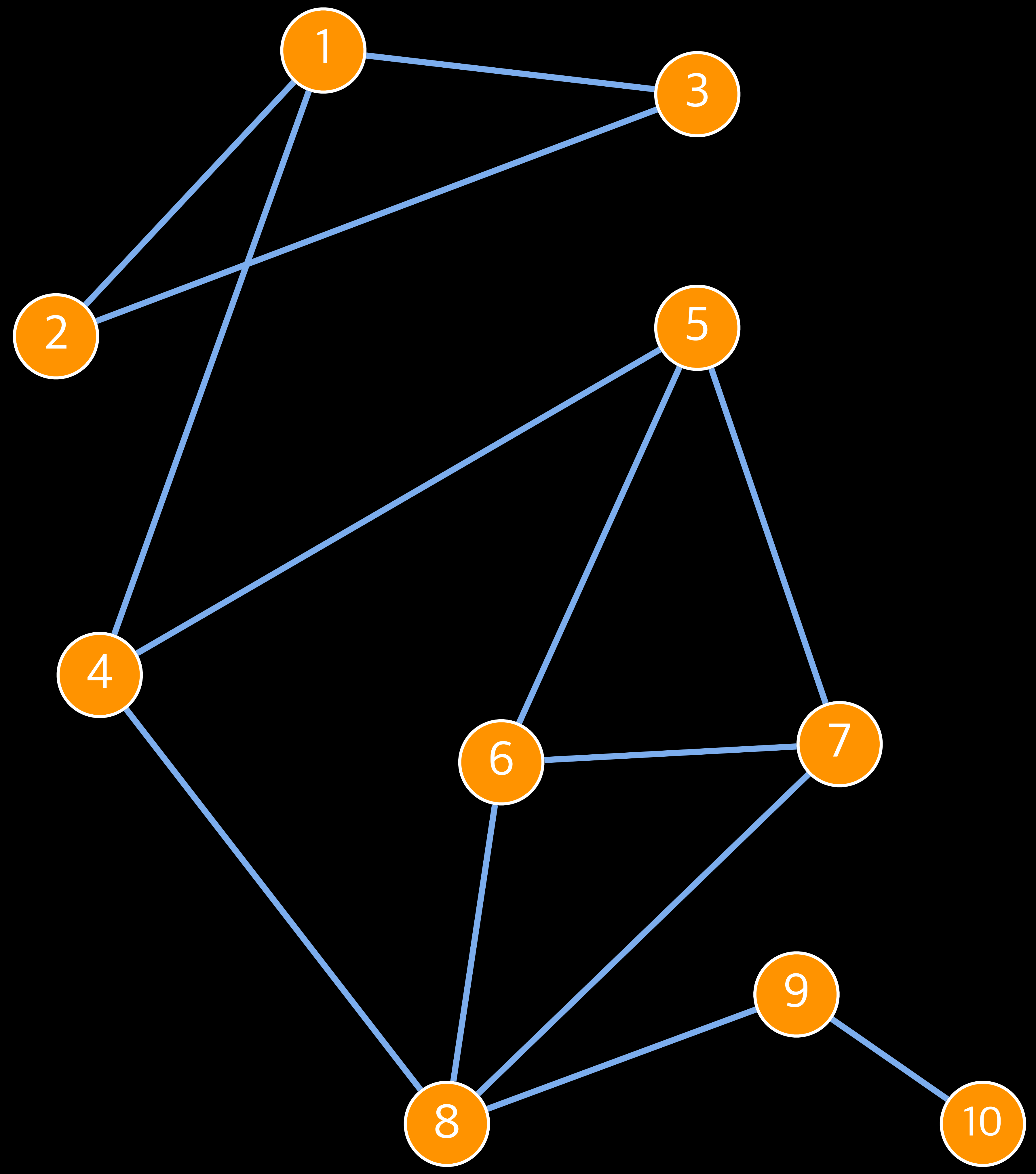
Stack of suspended vertices

4	1	2					

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

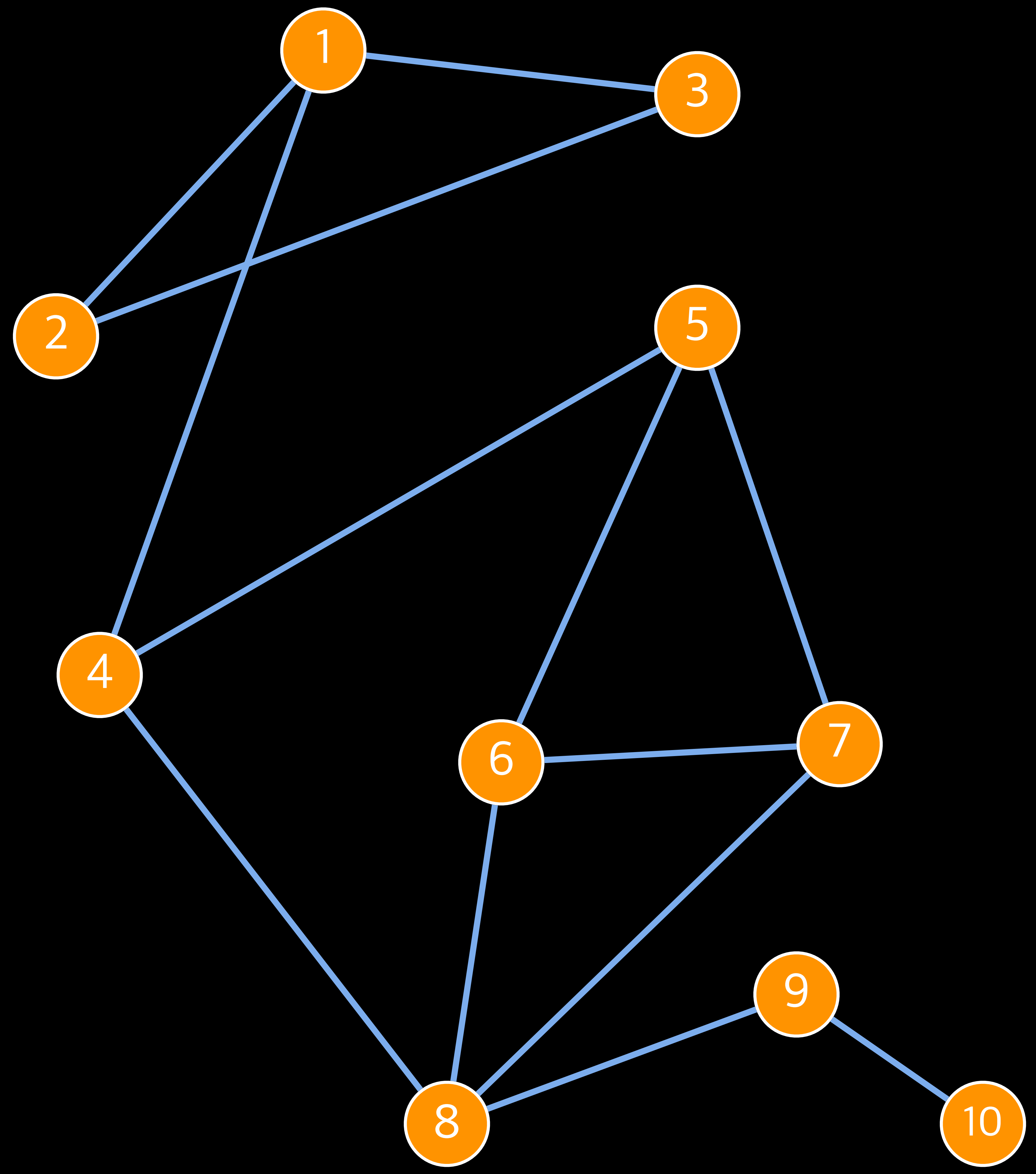
Stack of suspended vertices

4	1						

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

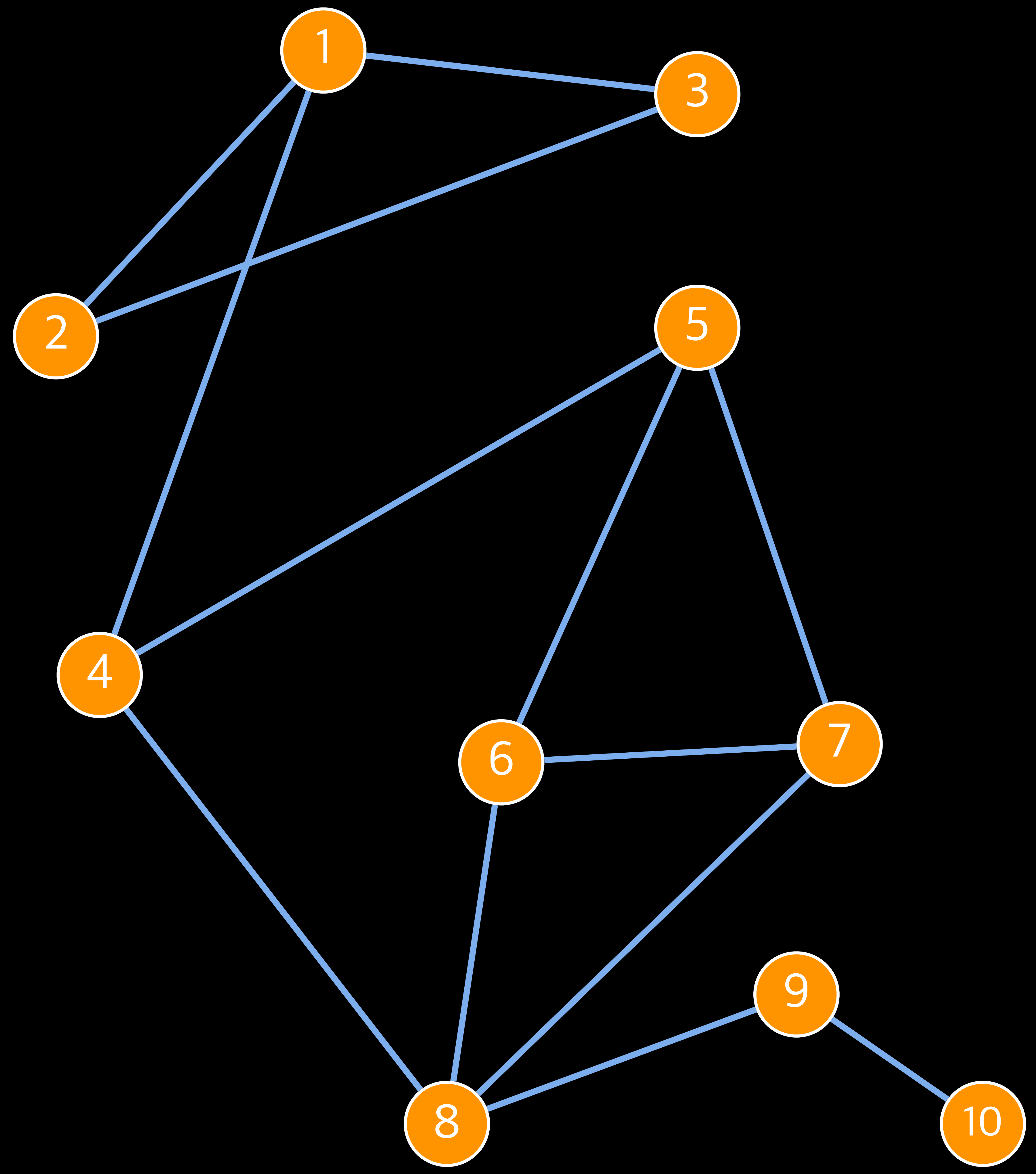
Stack of suspended vertices

4							

Visited

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

DEPTH-FIRST SEARCH



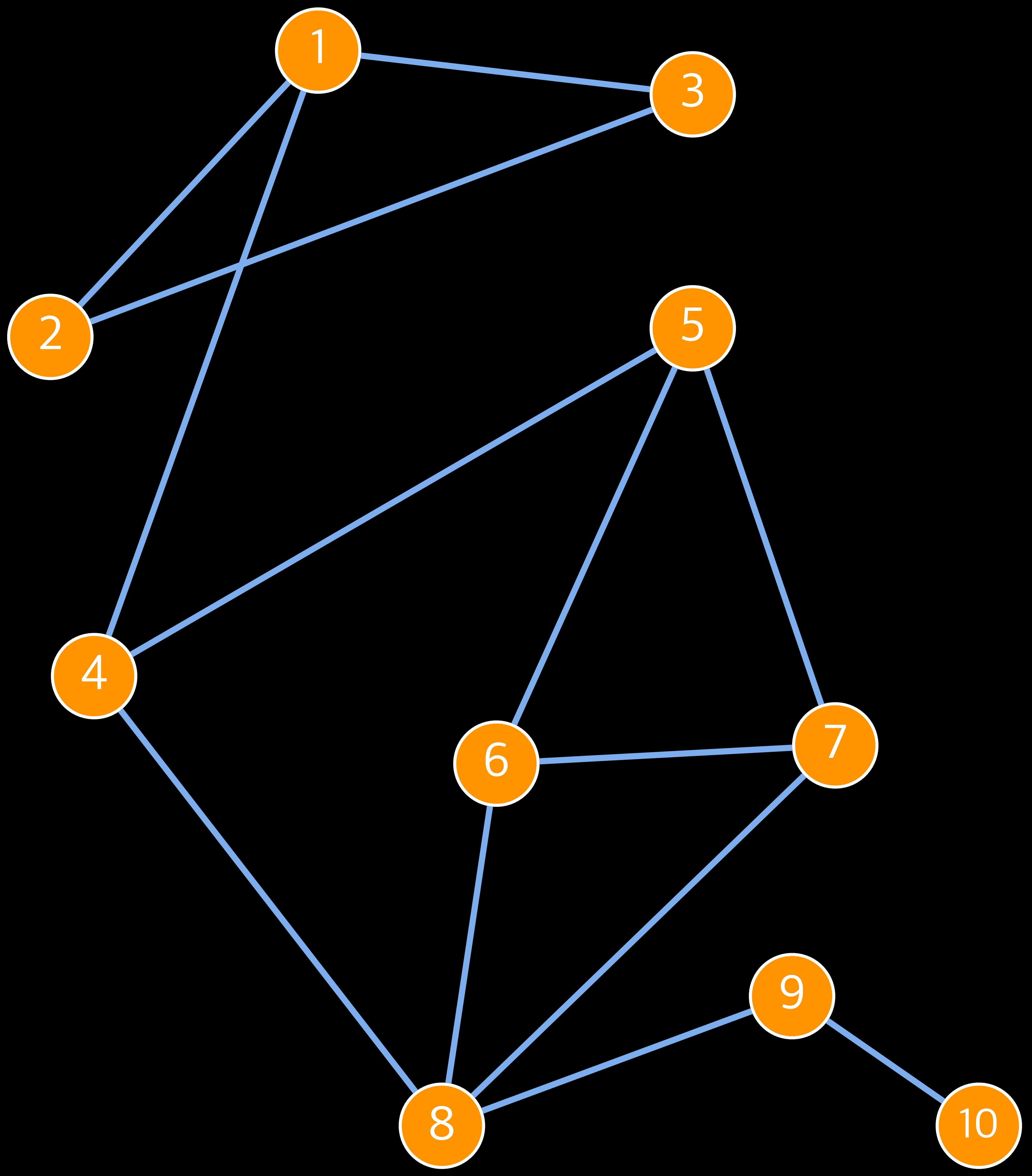
Start at the node 4

Stack of suspended vertices

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

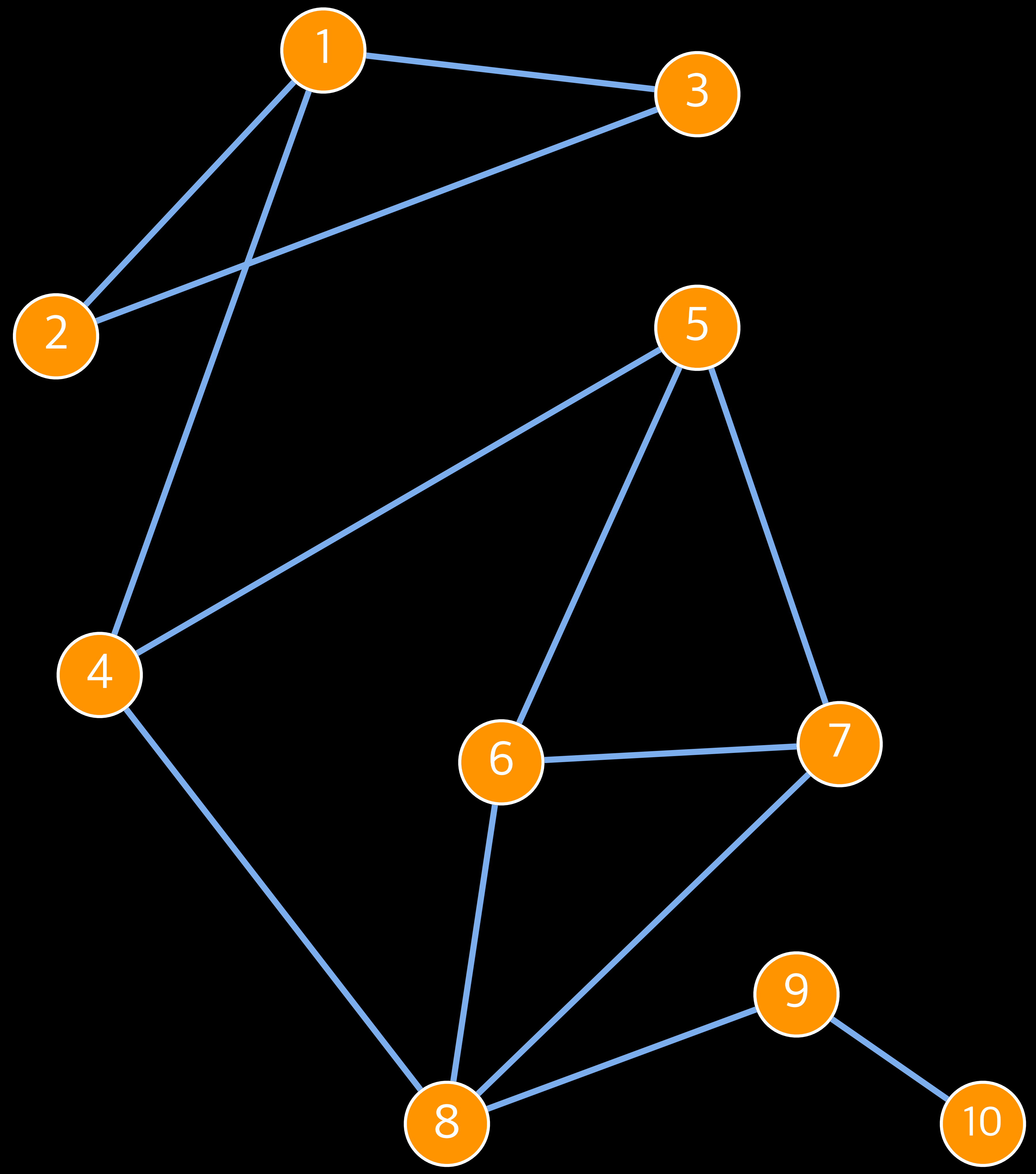
Stack of suspended vertices

4							

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

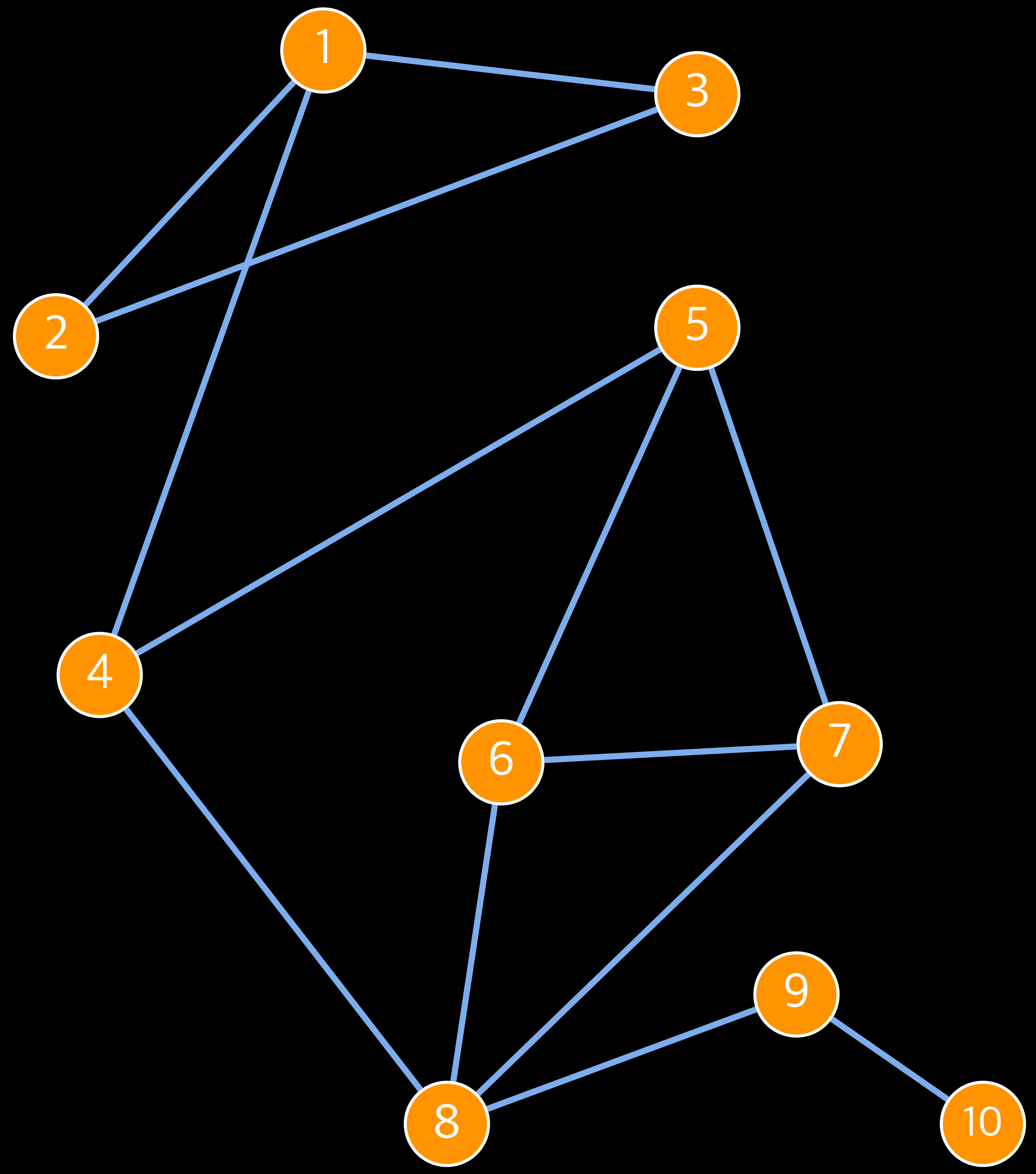
Stack of suspended vertices

4	5						

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

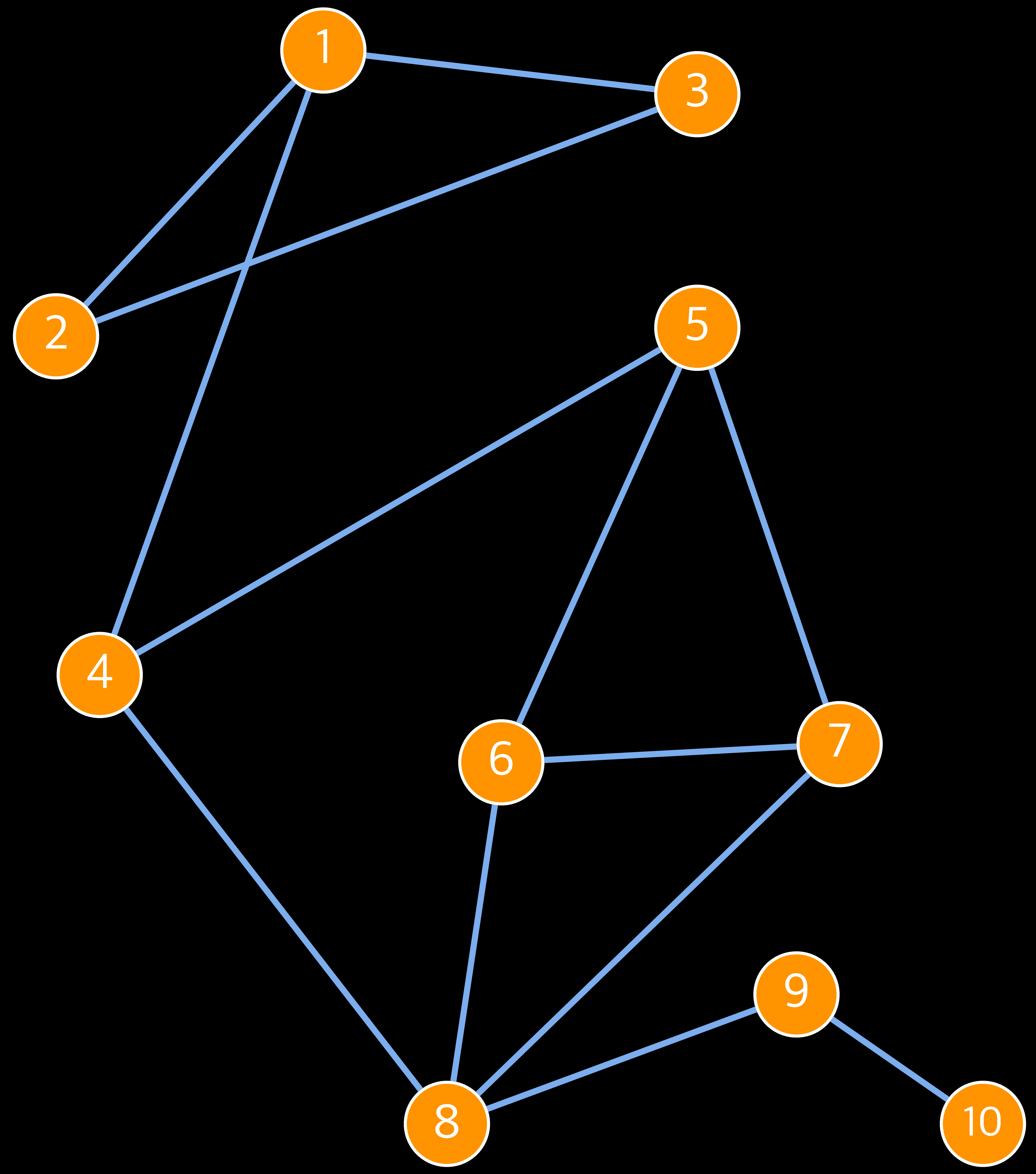
Stack of suspended vertices

4	5	6					

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

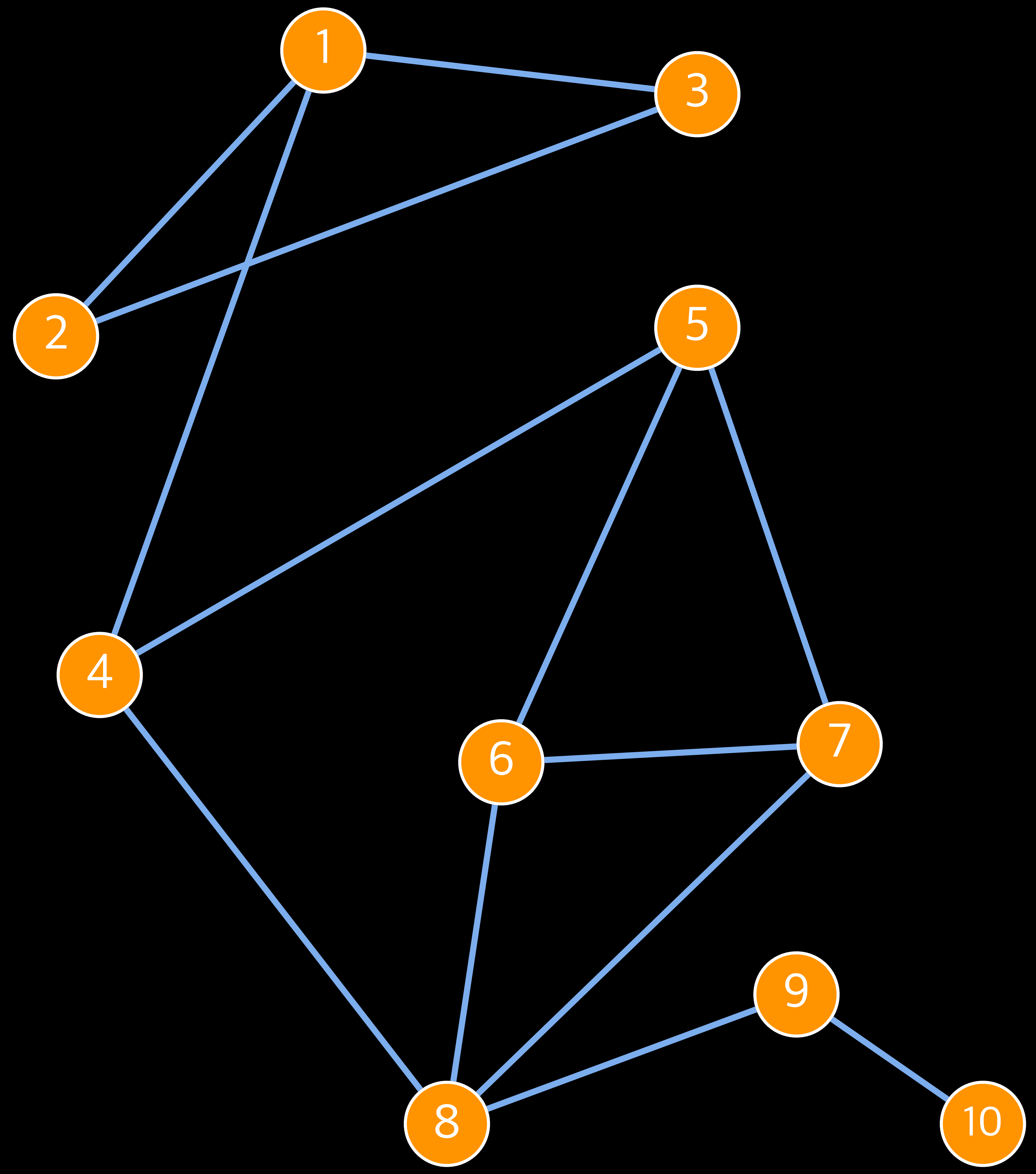
Stack of suspended vertices

4	5	6					

Visited

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

DEPTH-FIRST SEARCH



Start at the node 4

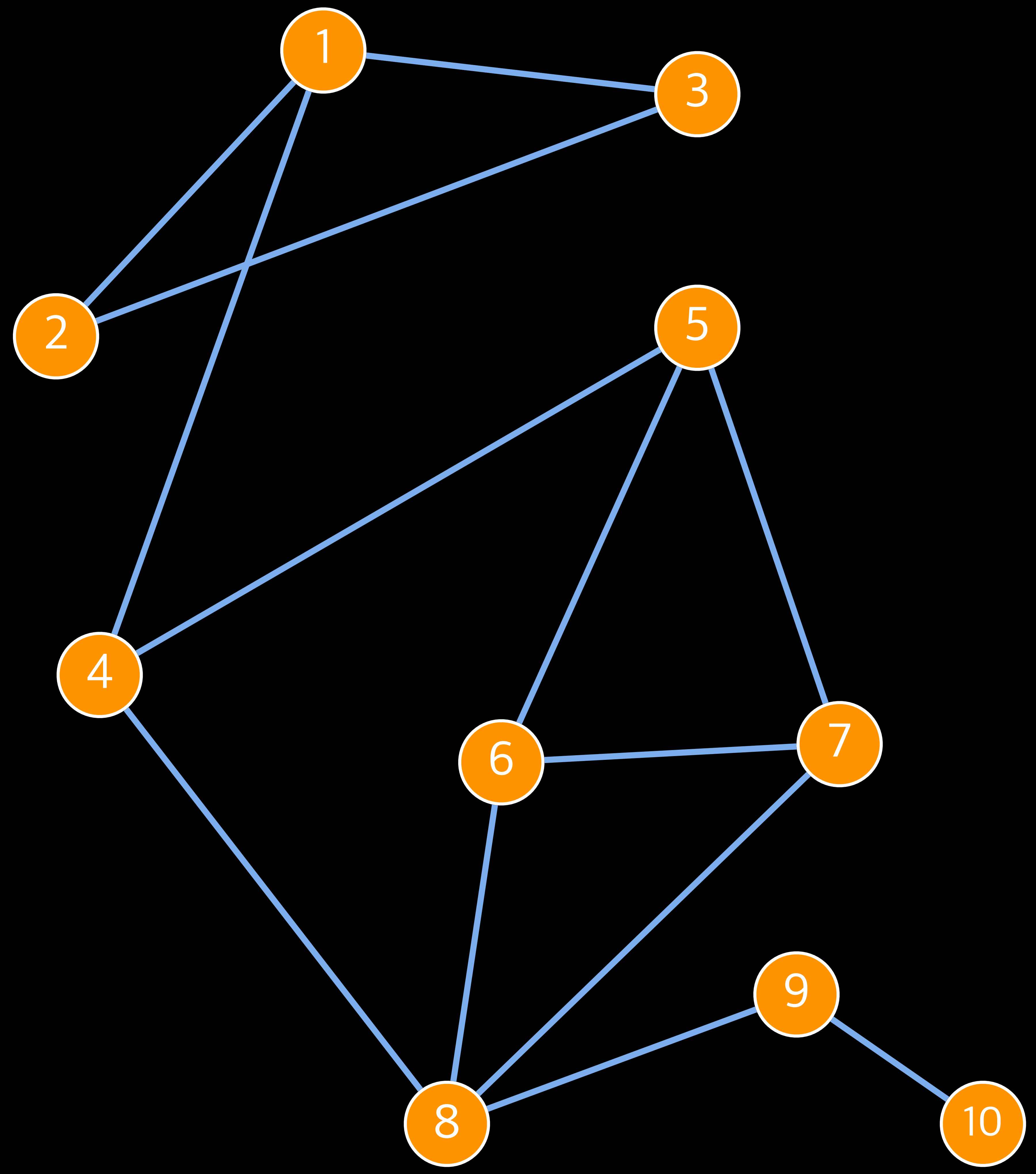
Stack of suspended vertices

4	5	6	8				

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	

DEPTH-FIRST SEARCH



Start at the node 4

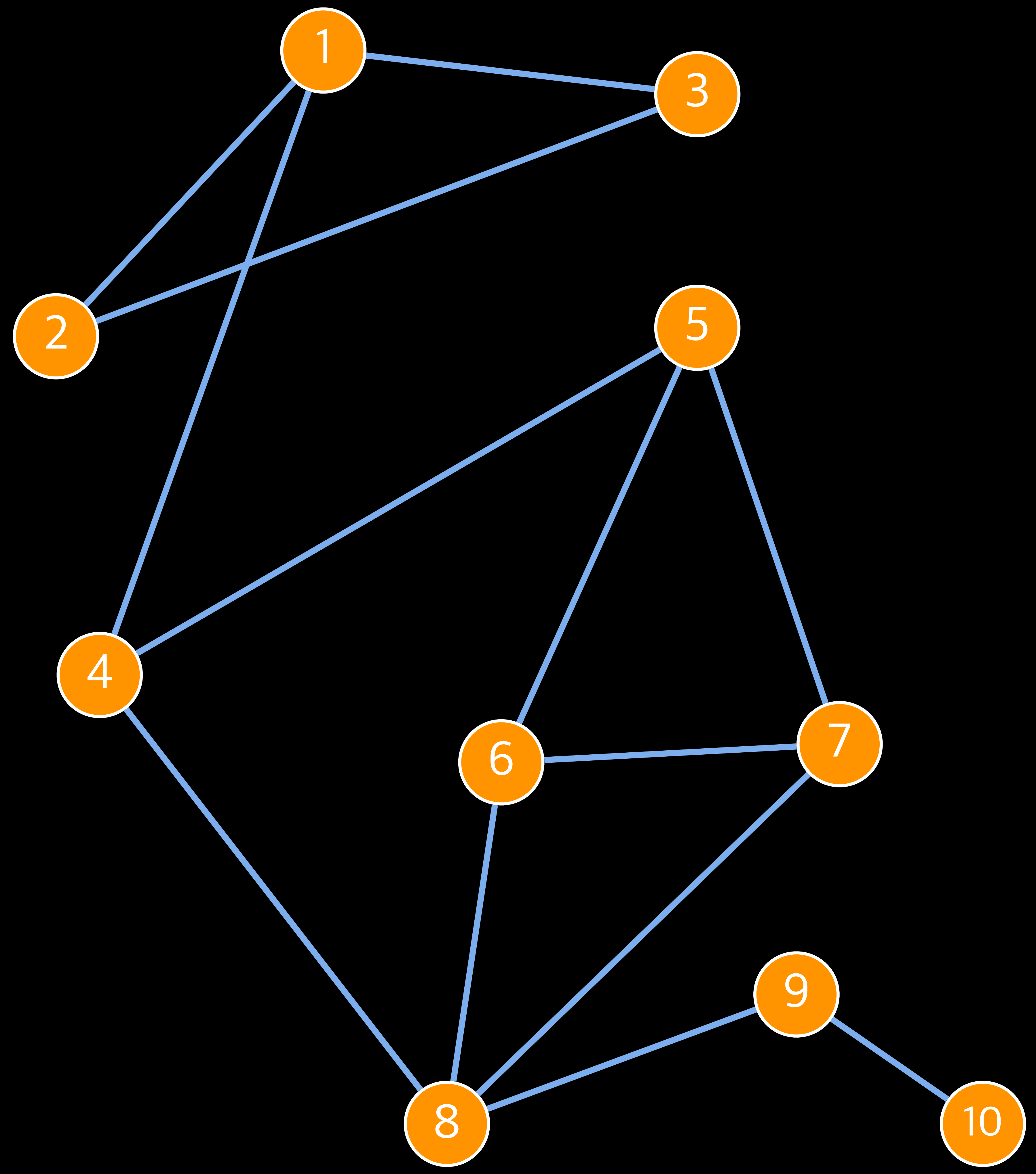
Stack of suspended vertices

4	5	6	8	9			

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DEPTH-FIRST SEARCH



Start at the node 4

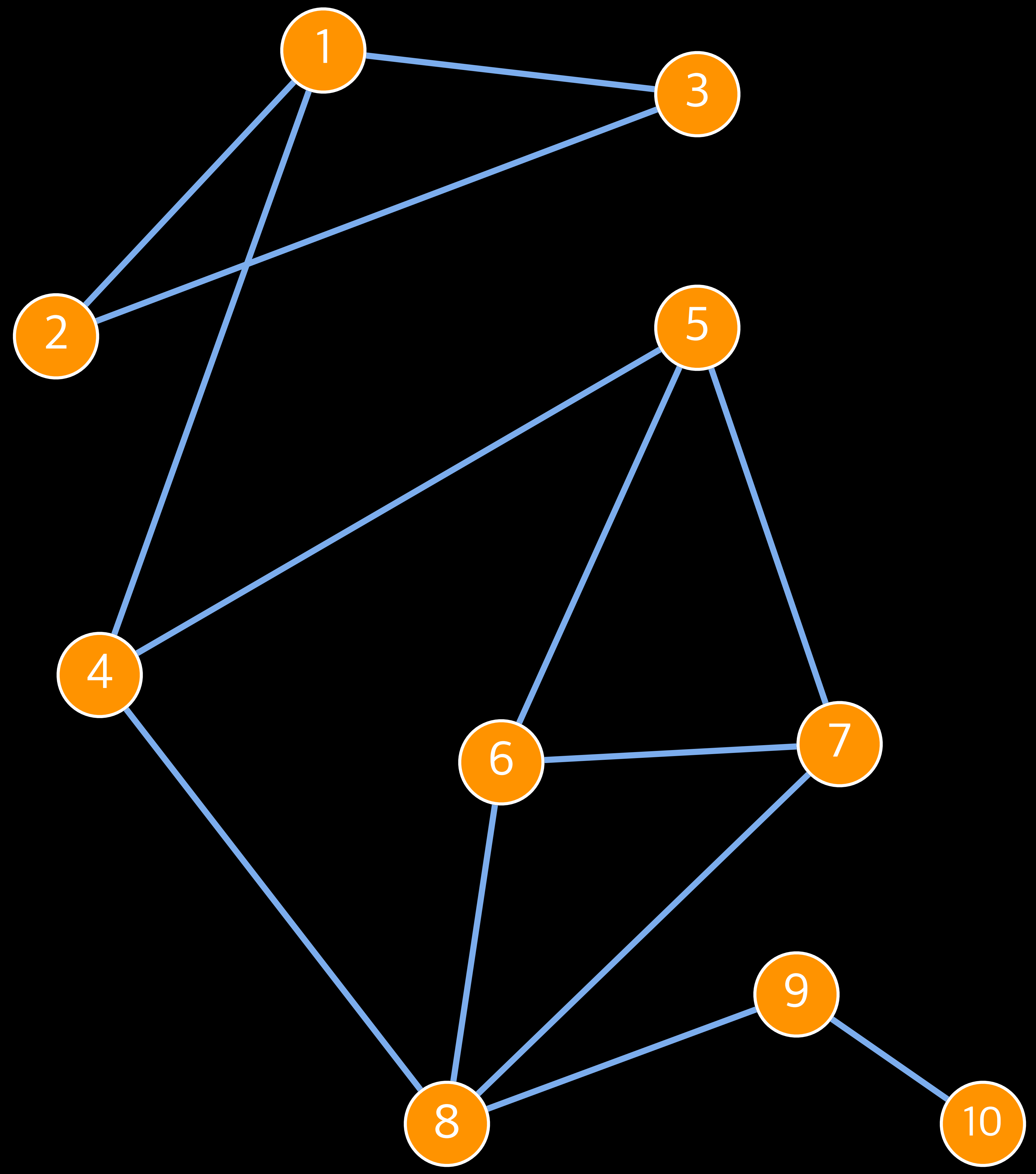
Stack of suspended vertices

4	5	6	8				

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DEPTH-FIRST SEARCH



Start at the node 4

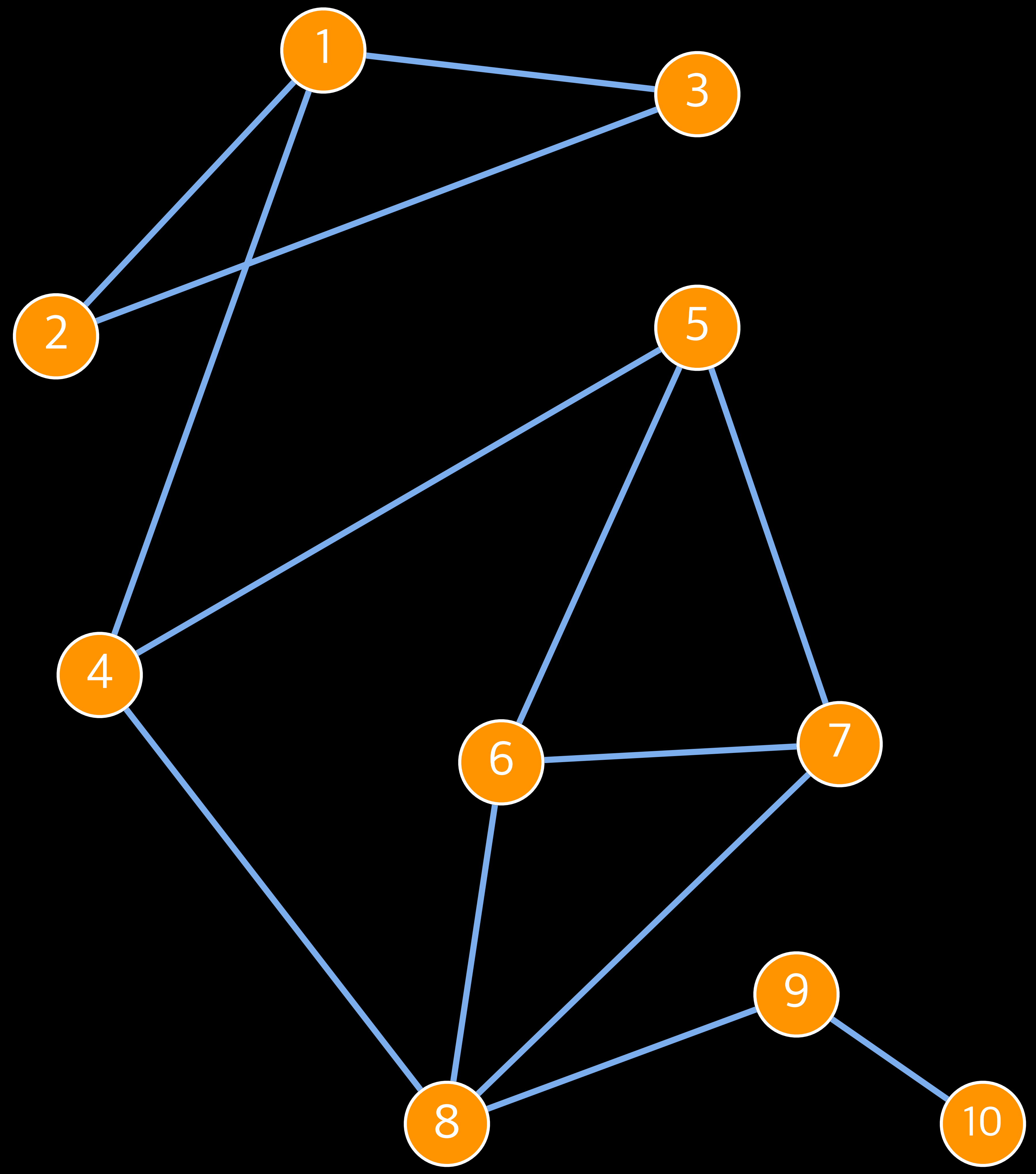
Stack of suspended vertices

4	5	6					

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DEPTH-FIRST SEARCH



Start at the node 4

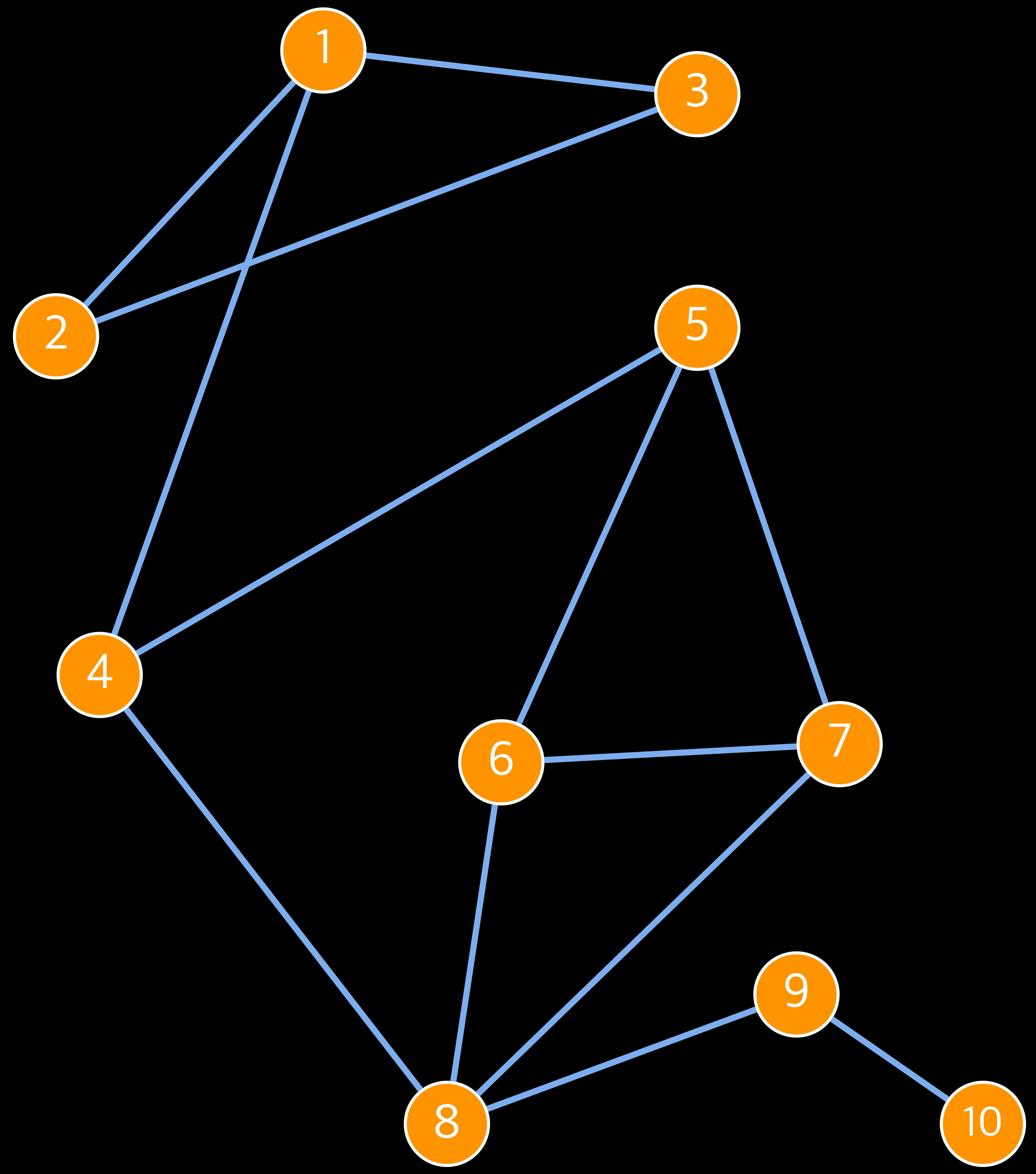
Stack of suspended vertices

4	5						

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DEPTH-FIRST SEARCH



Start at the node 4

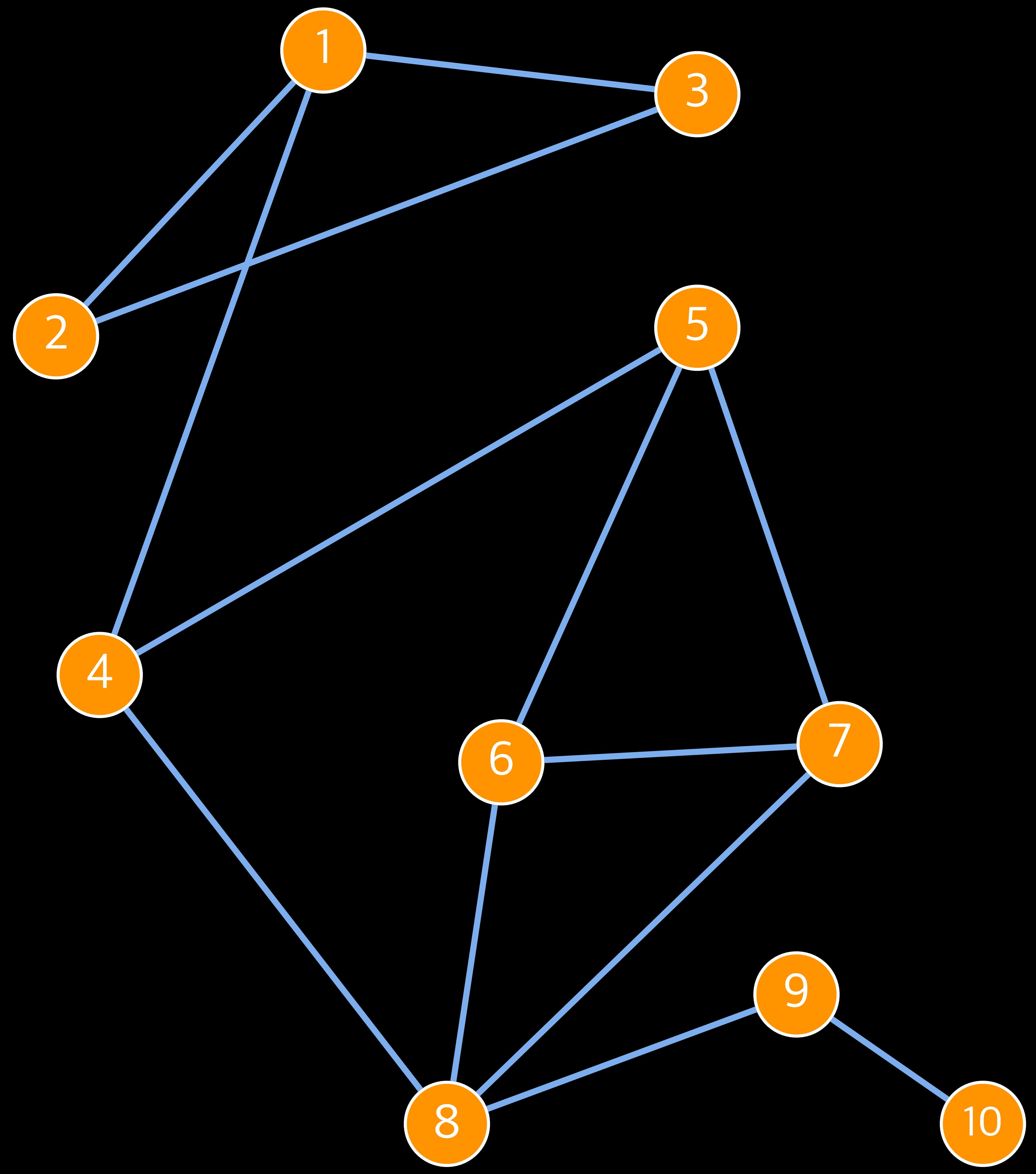
Stack of suspended vertices

4							

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DEPTH-FIRST SEARCH



Start at the node 4

Stack of suspended vertices

Visited

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1

DFS ALGORITHM

$$
\begin{aligned}
&\mathit{For}\ j = 1 \cdots n \\
&\quad \mathit{visited}[j] \leftarrow 0 \\
&\quad \mathit{parent}[j] \leftarrow -1 \\
&\mathit{function}\ \mathit{depth\_first\_search}(i) \\
&\quad \mathit{visited}[j] \leftarrow 1 \\
&\quad \mathit{Foreach}\ (i, j) \in E \\
&\quad\quad \mathit{If}\ \mathit{visited}[j] \leftarrow 1 \\
&\quad\quad\quad \mathit{parent}[j] \leftarrow i \\
&\quad\quad\quad \mathit{depth\_first\_search}(j)
\end{aligned}
$$

APPLICATION IN THE INFORMATION RETRIEVAL

To build a term-Document binary incidence matrix, let us consider Shakespeare's plays as our corpus. The terms are the vertices and the names of the plays are considered as edges. This incidence matrix does not represent any information related word order or its frequency

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
antony	1	1	0	0	0
brutus	1	1	0	1	0
caesar	1	1	0	1	1
calpurnia	0	1	0	0	0
cleopatra	1	0	0	0	0
...	...	...	...	...	...
...	...	...	...	...	...

$$x_{td} = \begin{cases} 1, & \text{if the word } t \in d \\ 0, & \text{if } \$t \notin d \end{cases}$$

USE OF GRAPHS IN INFORMATION RETRIEVAL

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
antony	1	1	0	0	0
brutus	1	1	0	1	0
caesar	1	1	0	1	1
calpurnia	0	1	0	0	0
cleopatra	1	0	0	0	0
...	...	...	...	...	...
...	...	...	...	...	...

To answer the query `Brutus AND Caesar AND NOT Calpurnia`, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:
11010 AND 11011 AND 10111 $=$ 10010.
The answer to this query is found in the following plays:

`Antony and Cleopatra and Hamlet|`

SHORTEST PATHS IN WEIGHTED GRAPHS