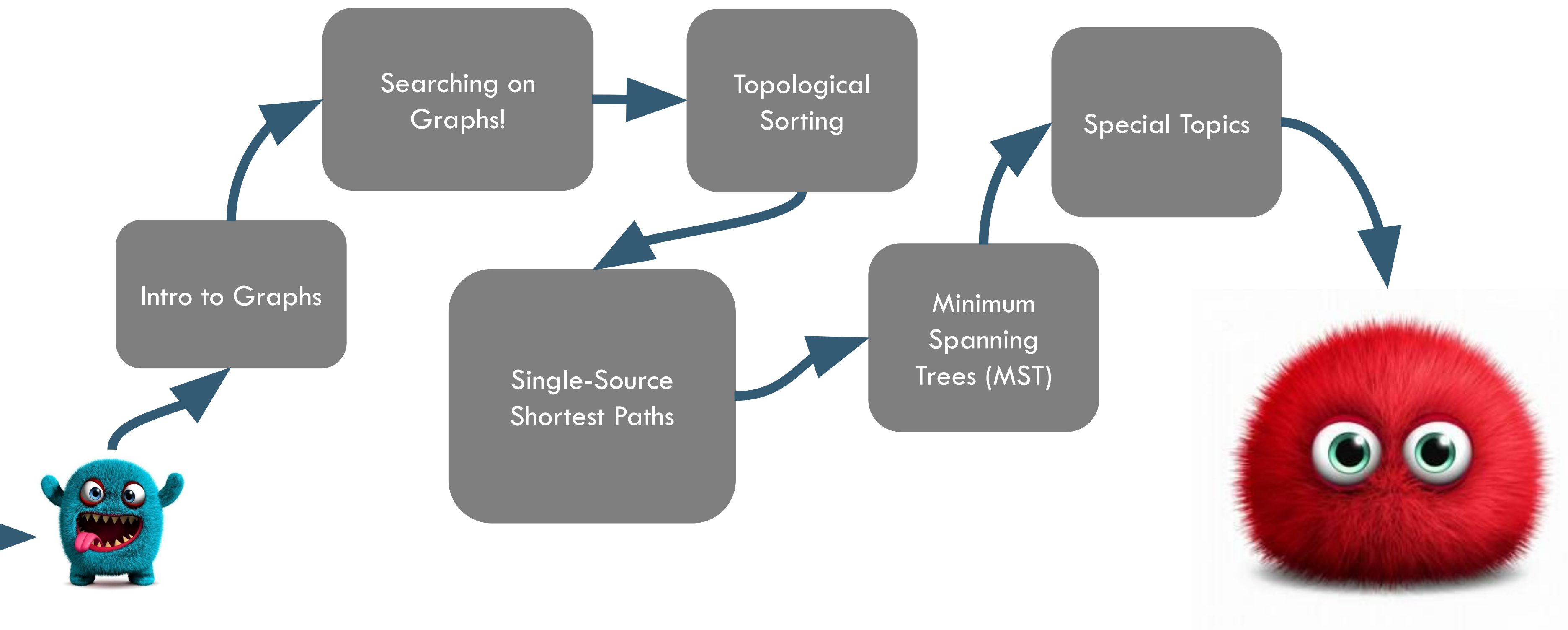
# CS2040S
# Data Structures and Algorithms

Graphs!

# COURSE STRUCTURE




# POST-MIDTERM: MODELLING AND SOLVING PROBLEMS

# Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

Shortest Pathfinding for Unweighted Graphs

# Roadmap

## Next: Searching Graphs

- Searching graphs
- More Shortest path problems
- Bellman-Ford Algorithm
- Dijkstra's Algorithm

# Roadmap

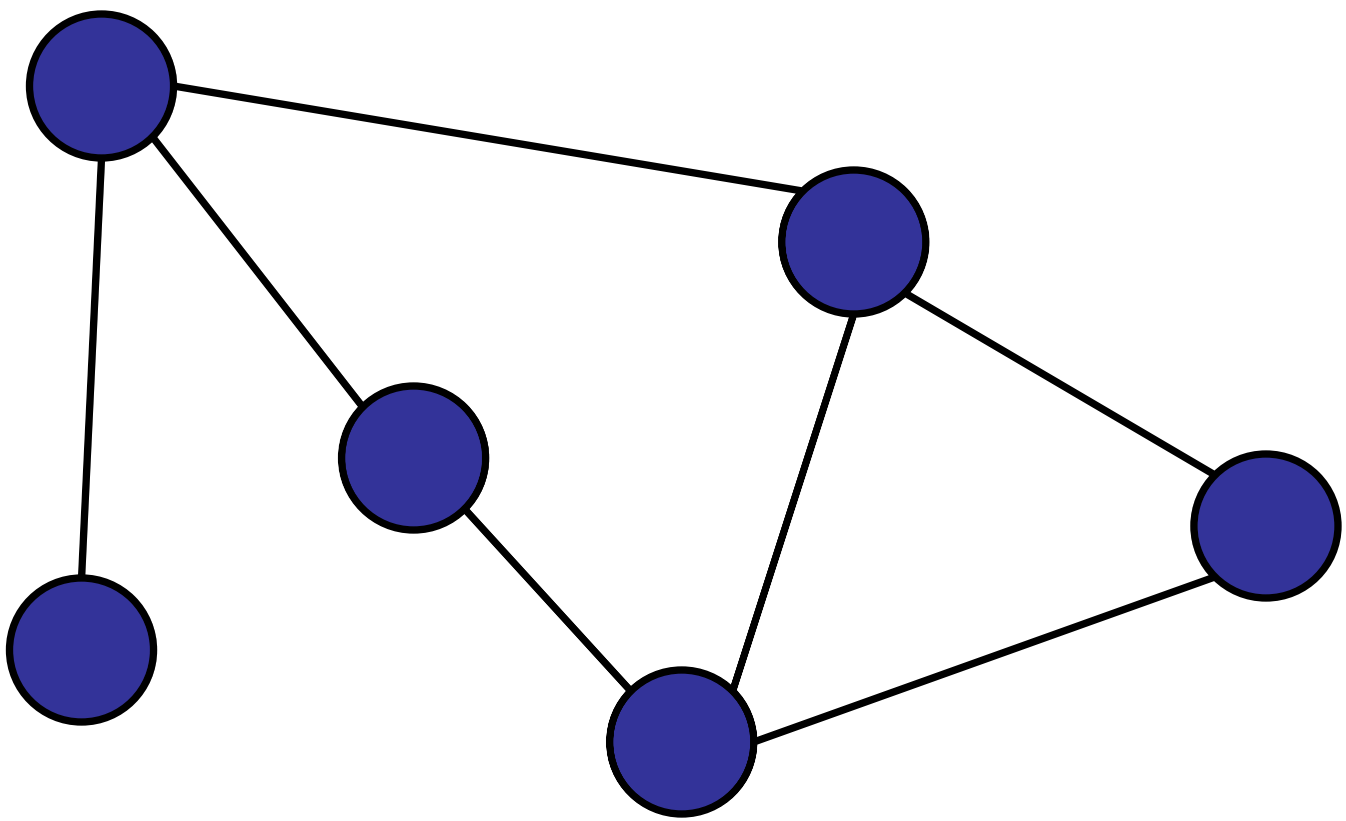
Next next:

- Connected component problem
  - Union-Find data structure
- The Minimum Spanning Tree Problem
  - Kruskal’s Algorithm
  - Prim’s Algorithm

# What is a graph?

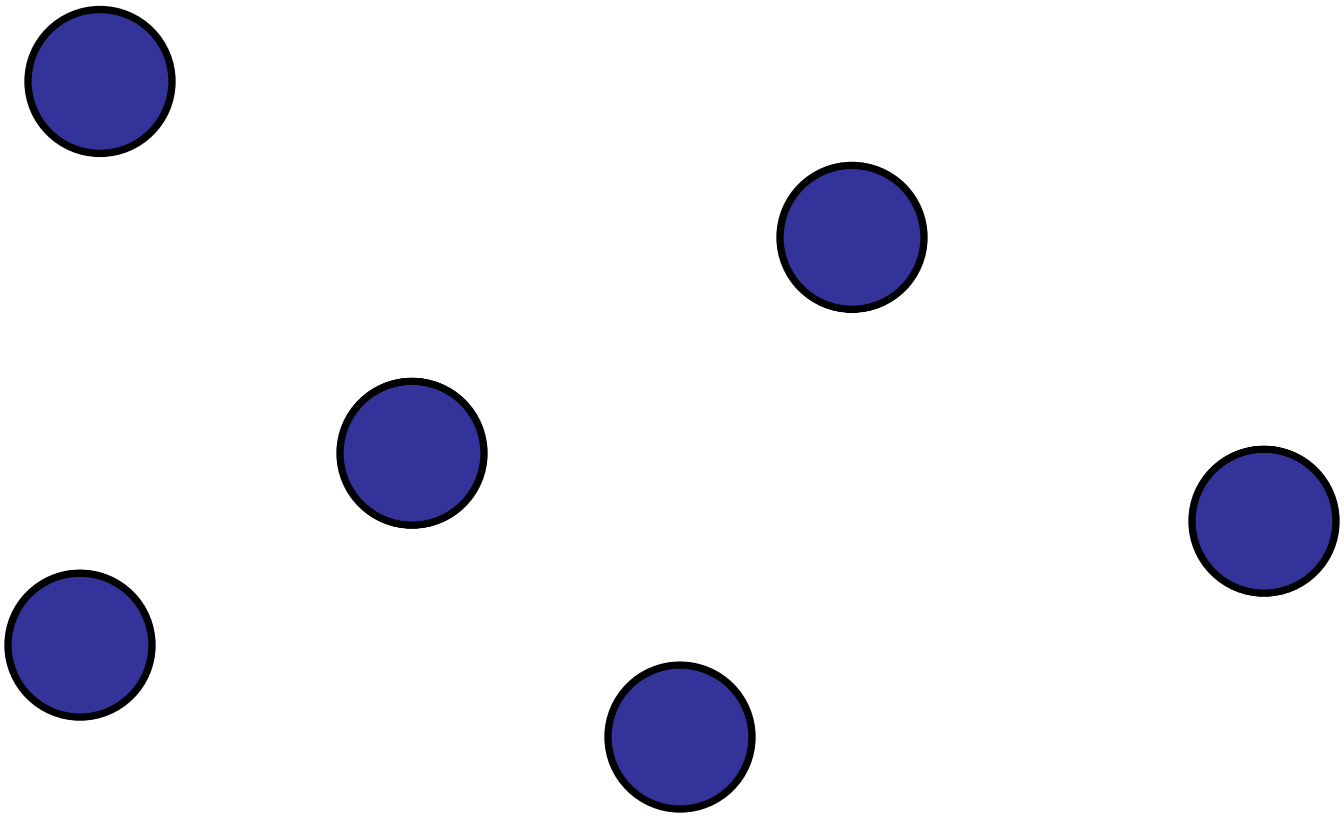
Is it a graph?

1. Yes
2. No.



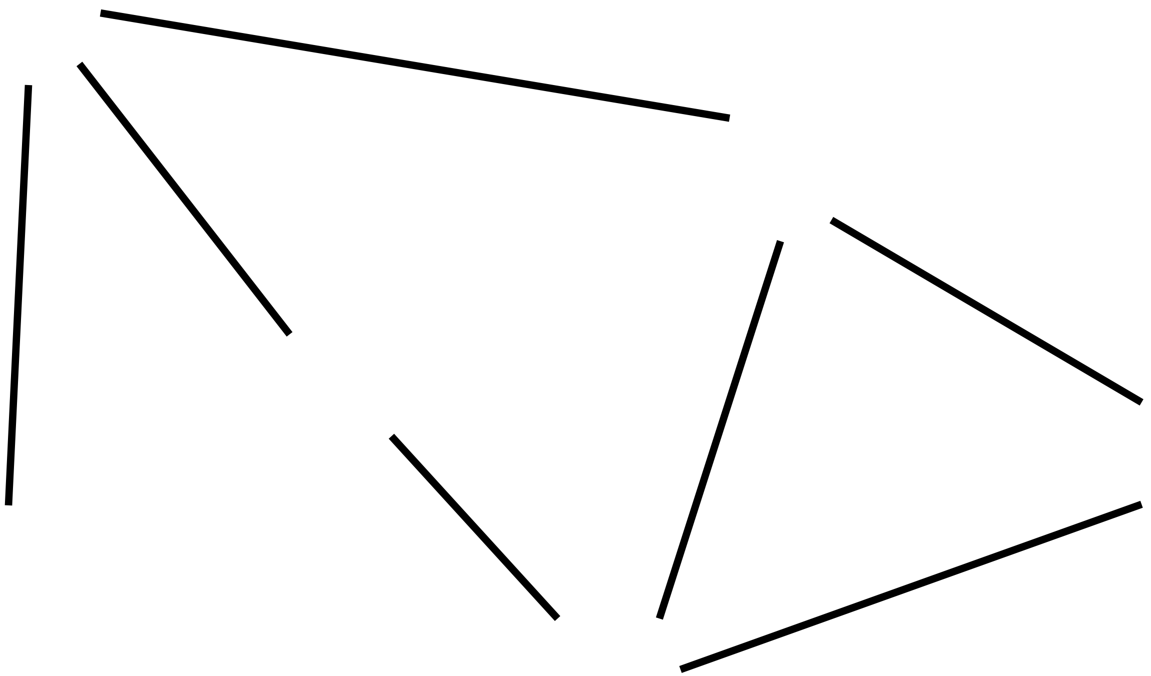
Is it a graph?

1. Yes
2. No.



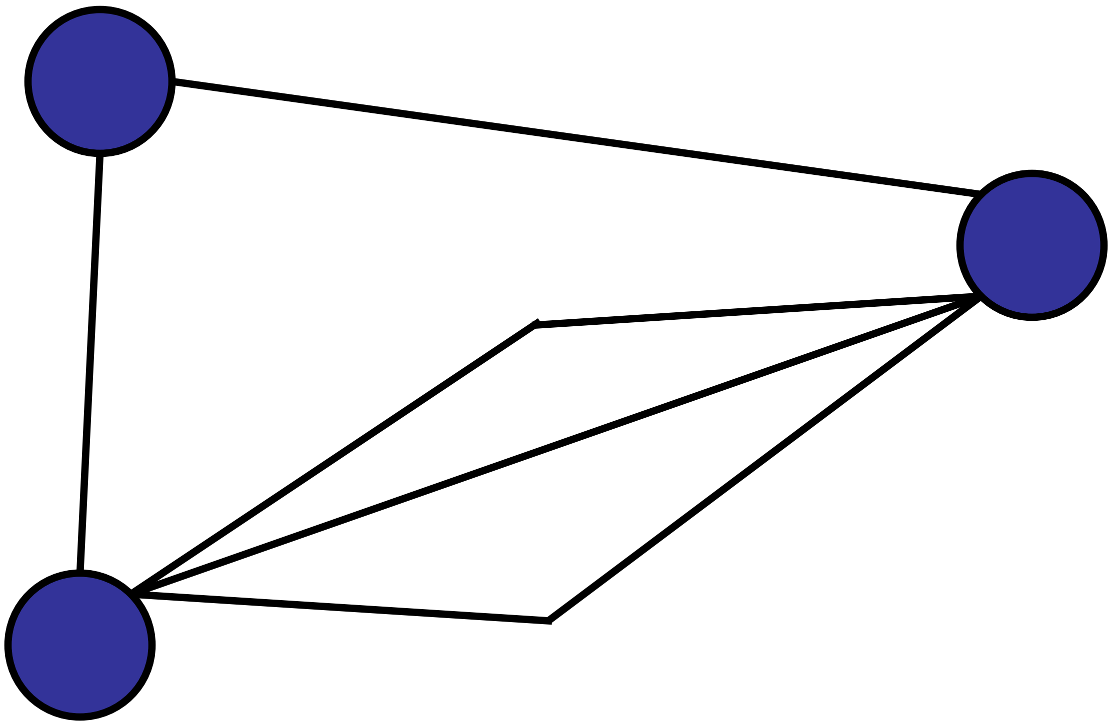
# Is it a graph?

1. Yes
2. No.



# Is it a graph?

1. Yes
2. No.



# Is it a graph?

1. Yes
2. No.

# What is a graph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.

- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.

# What is a graph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.

- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.

No two edges share the same two endpoints

No self loops

# What is a graph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.

No two edges share the same two endpoints

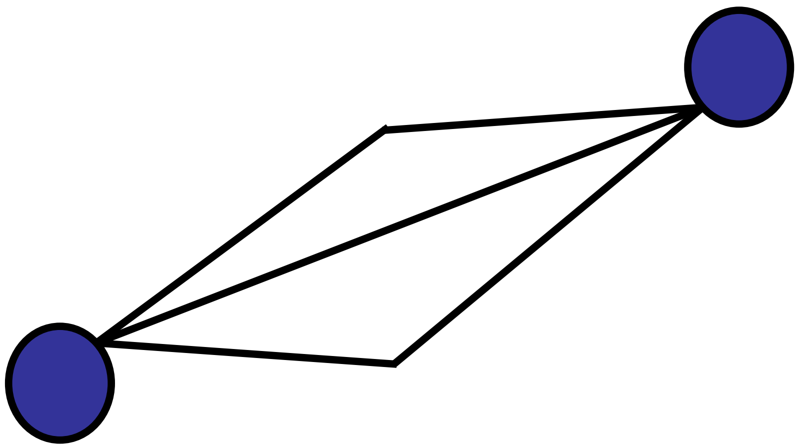
No self loops

I.e. Simple

# What is a multigraph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.



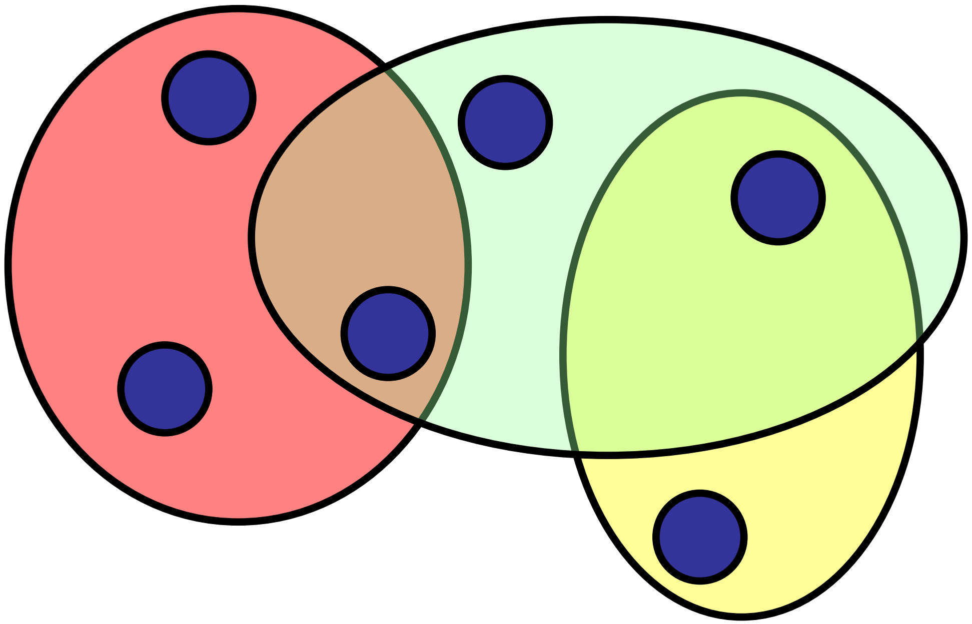
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Two nodes may be connected by more than one edge.

(Rare in CS2040S.)

# What is a hypergraph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.



- Edges (or arcs)
  - Each edge connects >= 2 nodes in the graph
  - Each edge is unique.

(Not common in CS2040S)

# What is a graph?

Graph $G = \langle V, E \rangle$

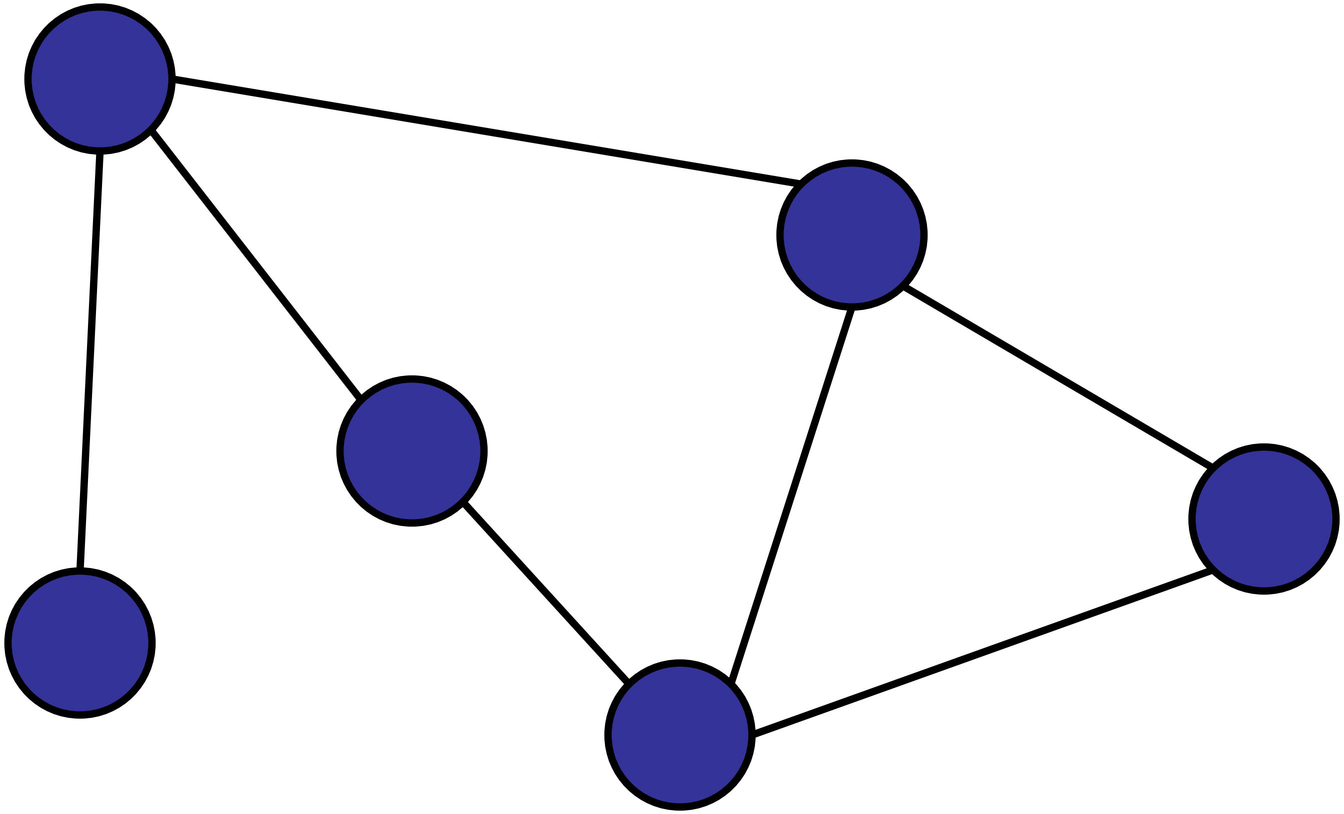
V is a set of nodes

- At least one: $|V| > 0$.

E is a set of edges:

- $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
- $e = (v,w)$
- For all $e_1, e_2 \in E : e_1 \neq e_2$

# Graph Terminology

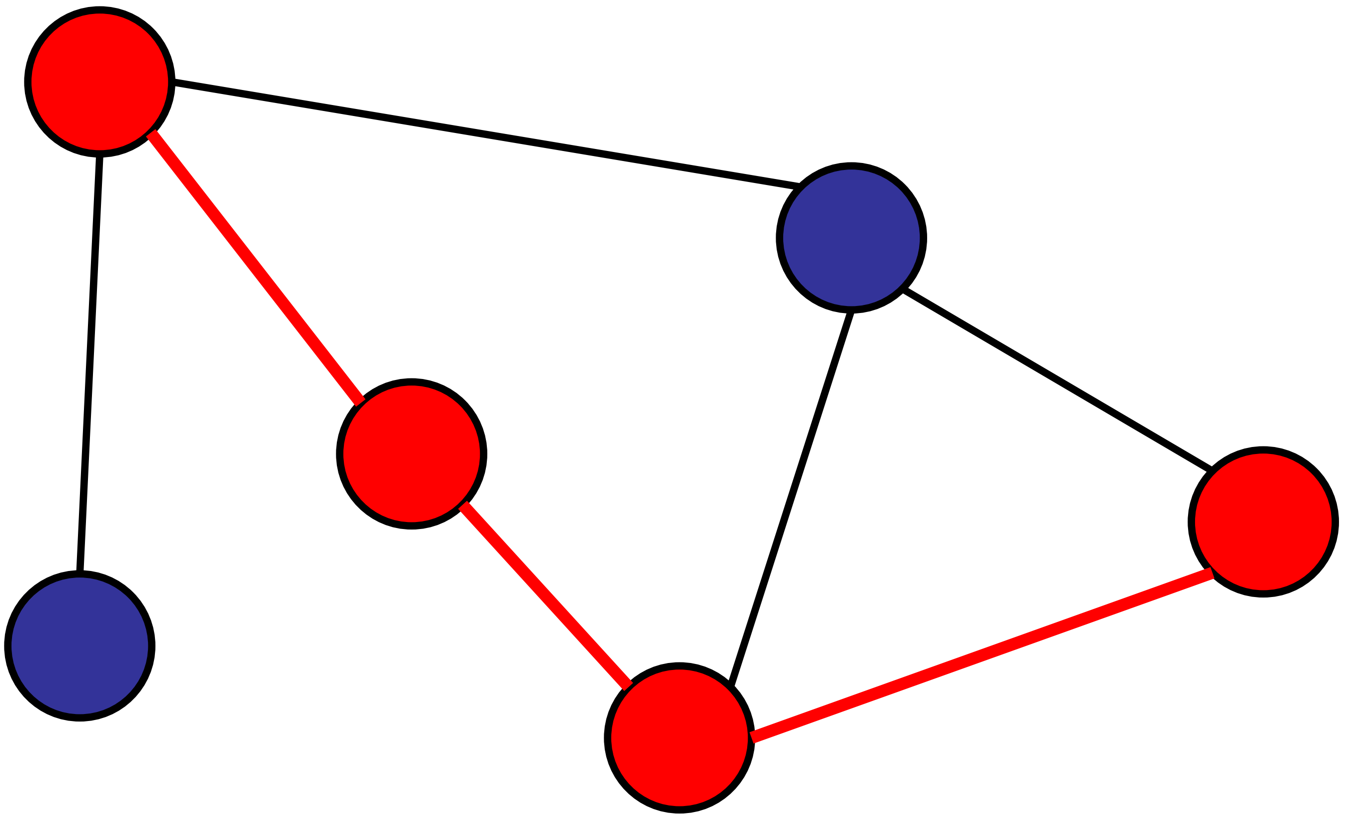


# Graph Terminology

**(Simple) Path**:

Set of edges connecting two nodes.

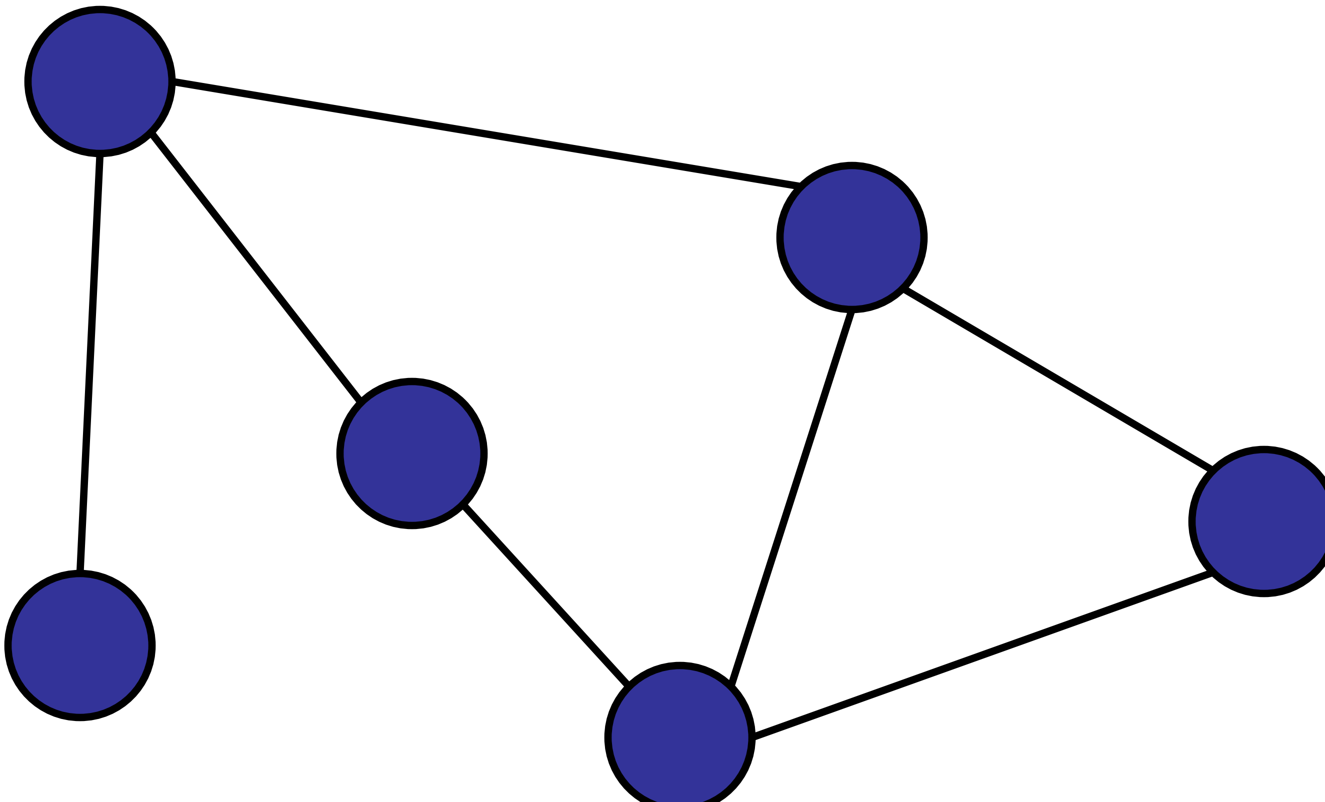
Path intersects each node at most once.



# Graph Terminology

**Connected**:

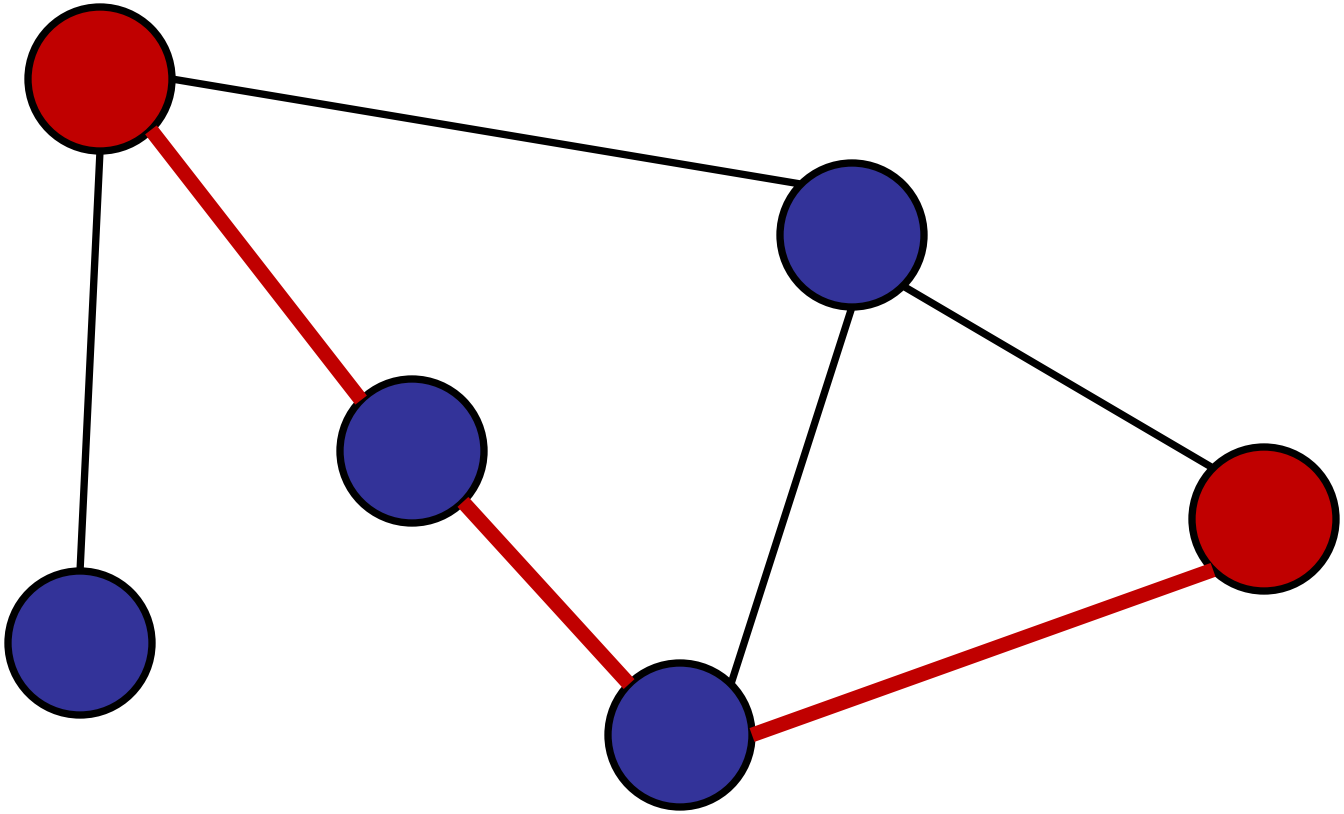
Every pair of nodes is connected by a path.



# Graph Terminology

**Connected**:

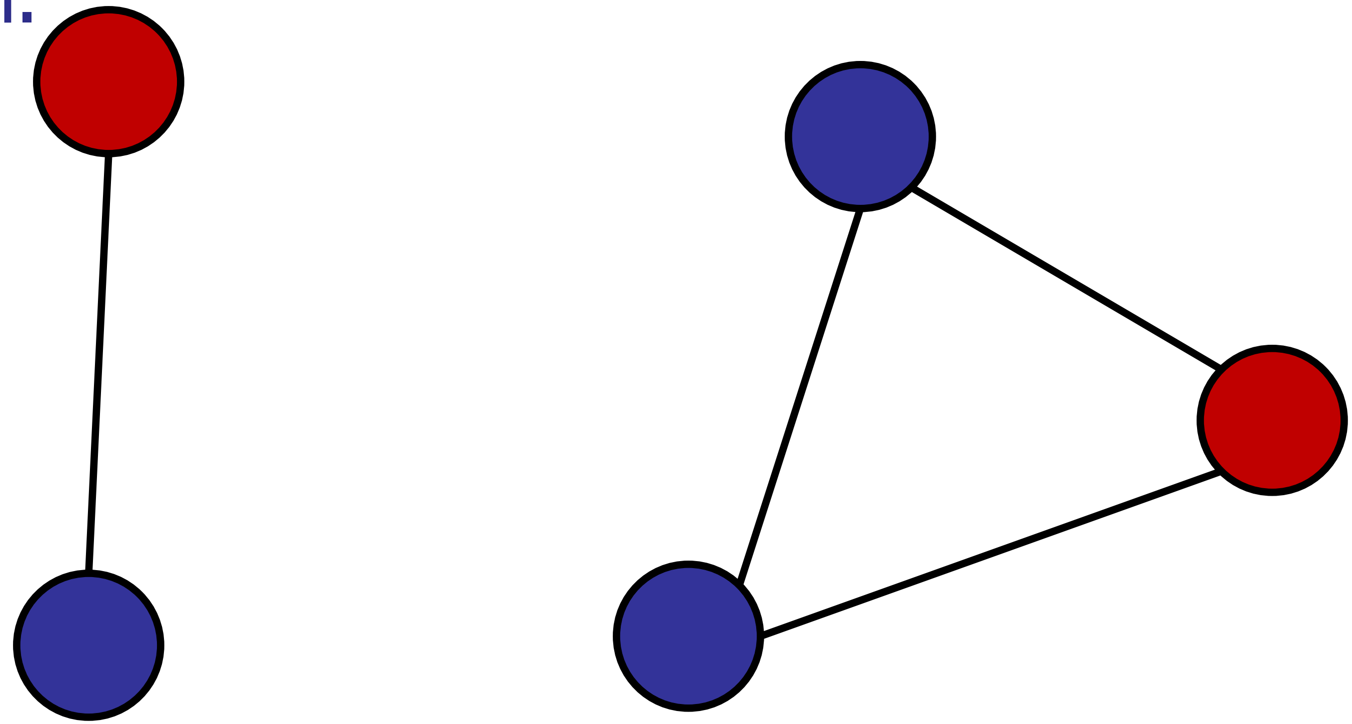
A graph is **connected** if every pair of nodes is connected by a path.



# Graph Terminology

**Disconnected**:

– Exists some pair of nodes that is not connected by a path.

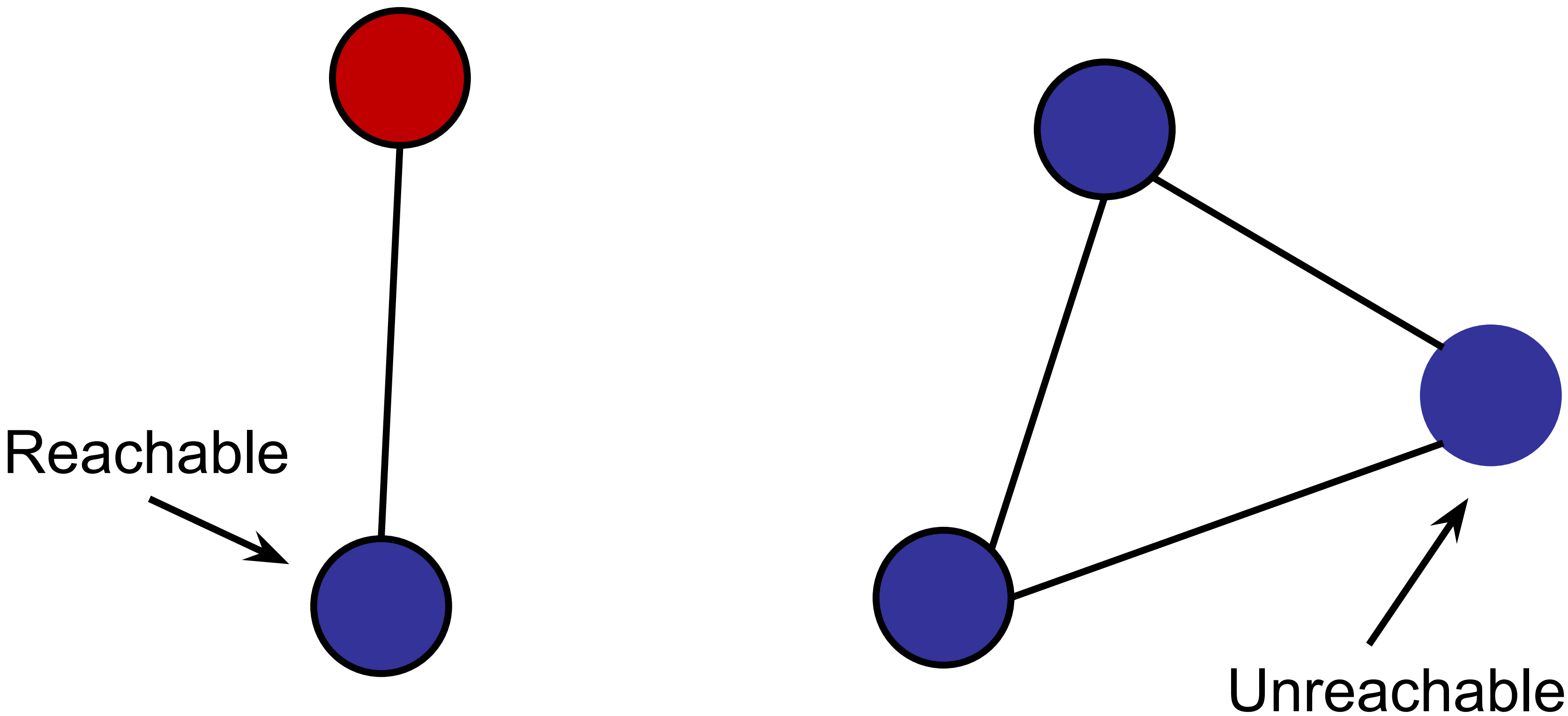


– Two **connected components**.

# Graph Terminology

**Disconnected**:

– Some pair of nodes is not connected by a path.

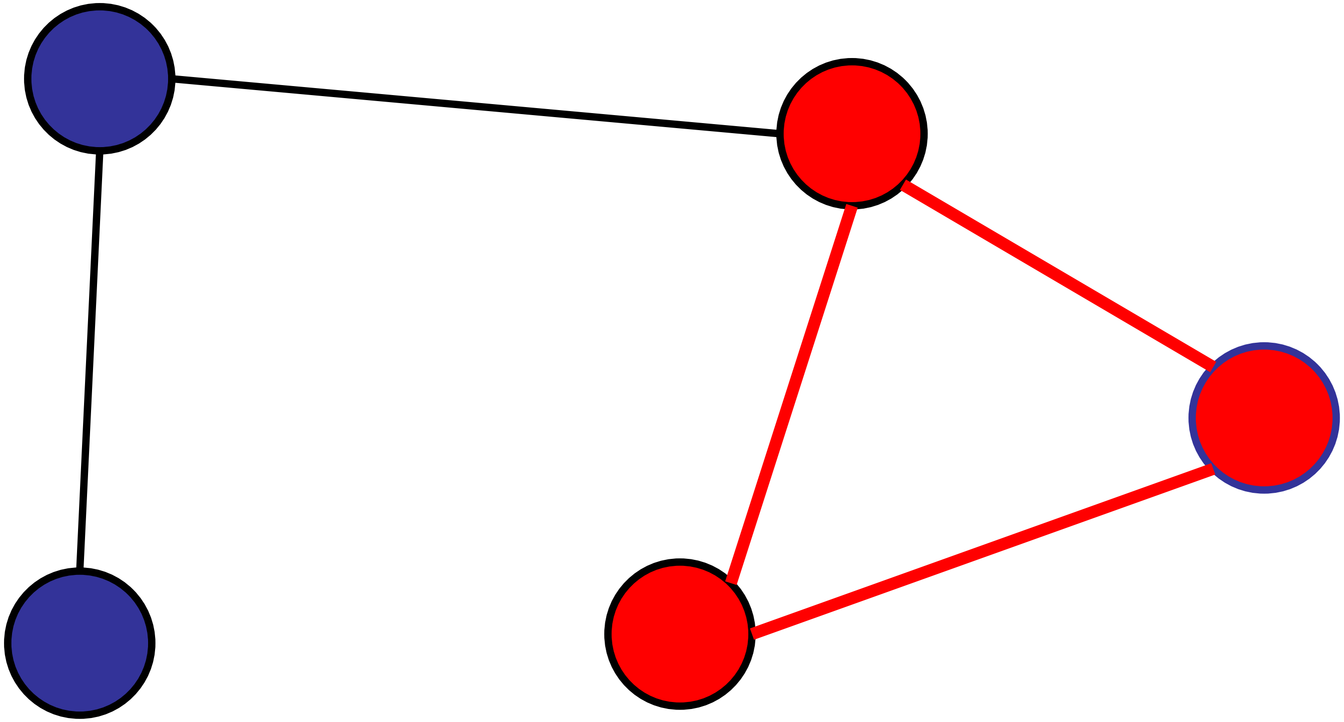



– Two **connected components**.

# Graph Terminology

**Cycle**:

"Path" where first and last node are the same.

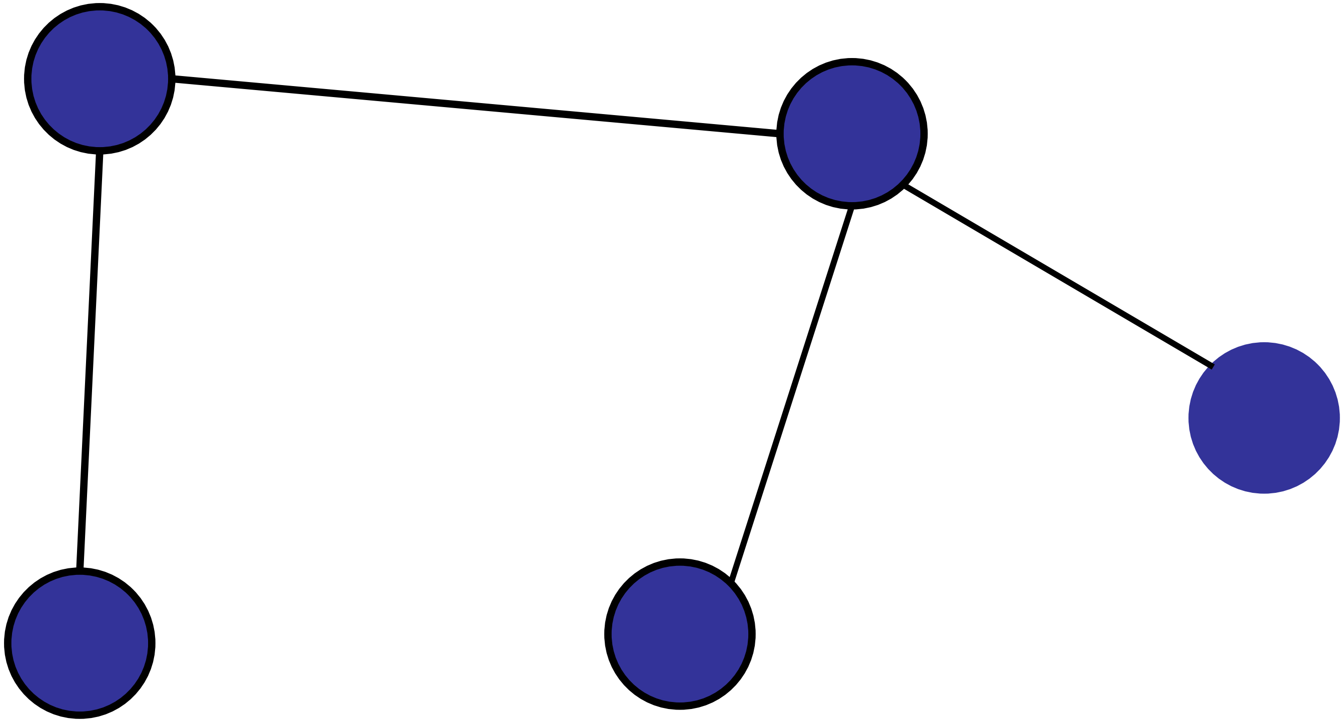


(**Not actually a path, since one node appears twice.)

# Graph Terminology

**(Unrooted) Tree:**

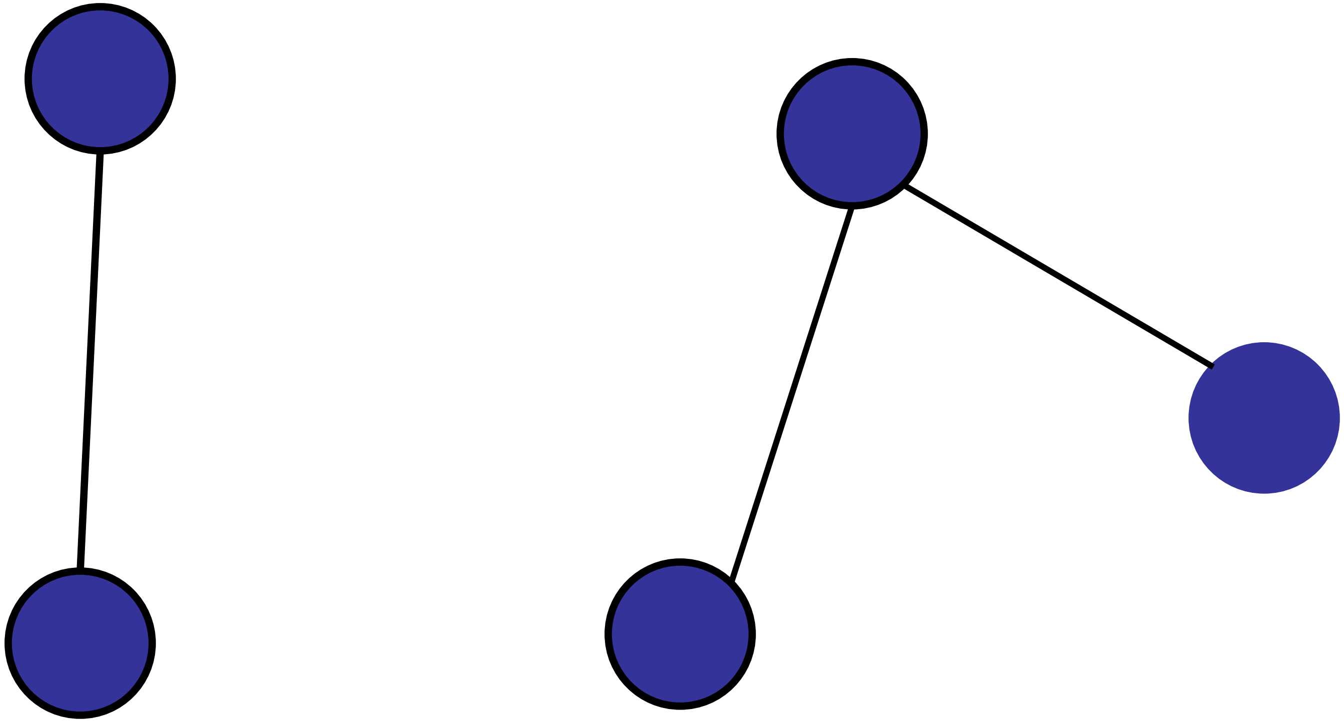
Connected graph with no cycles.



# Graph Terminology

**Forest:**

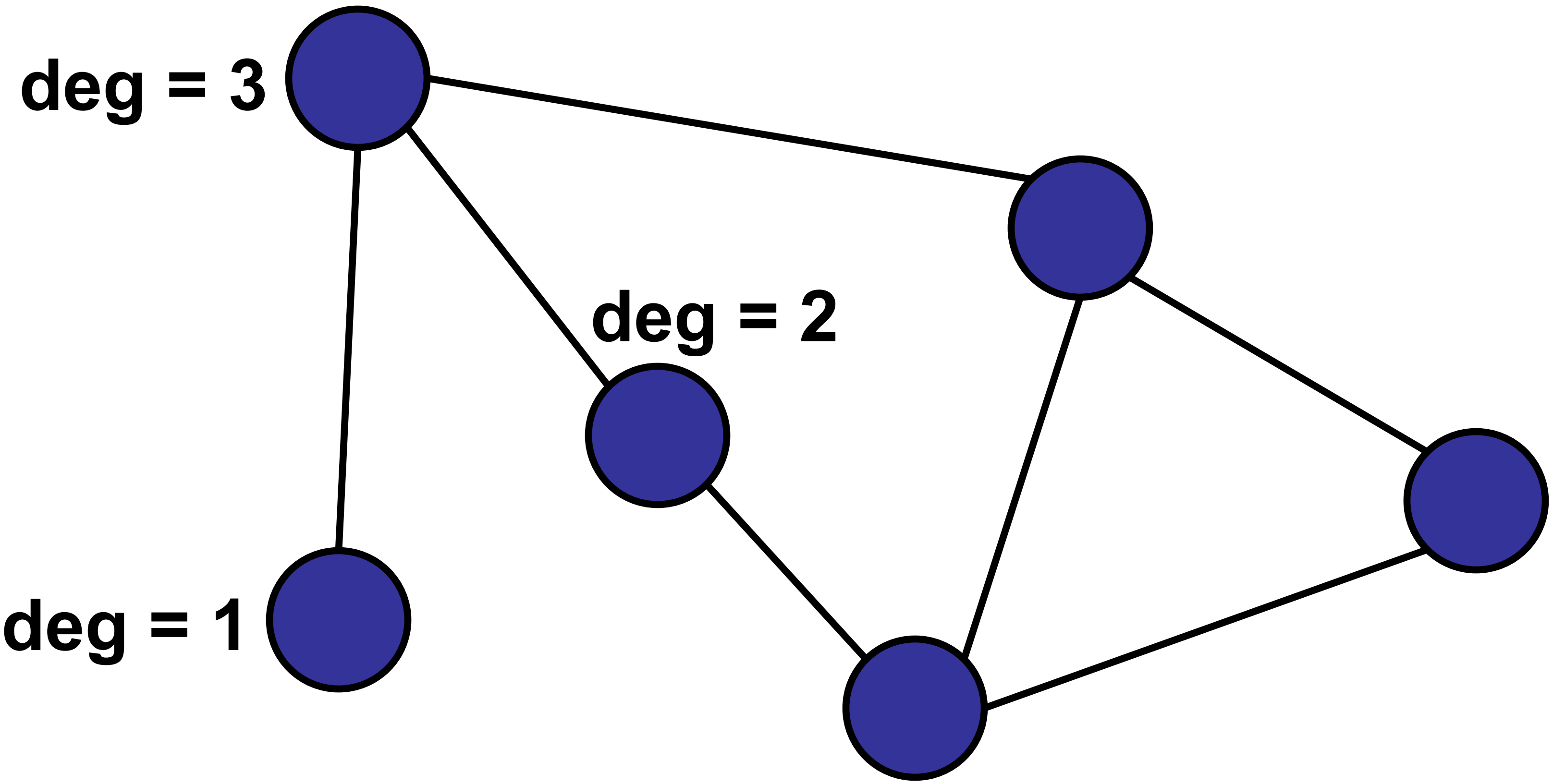
Graph with no cycles.



# Graph Terminology

**Degree of a node:**

- Number of **adjacent** edges.

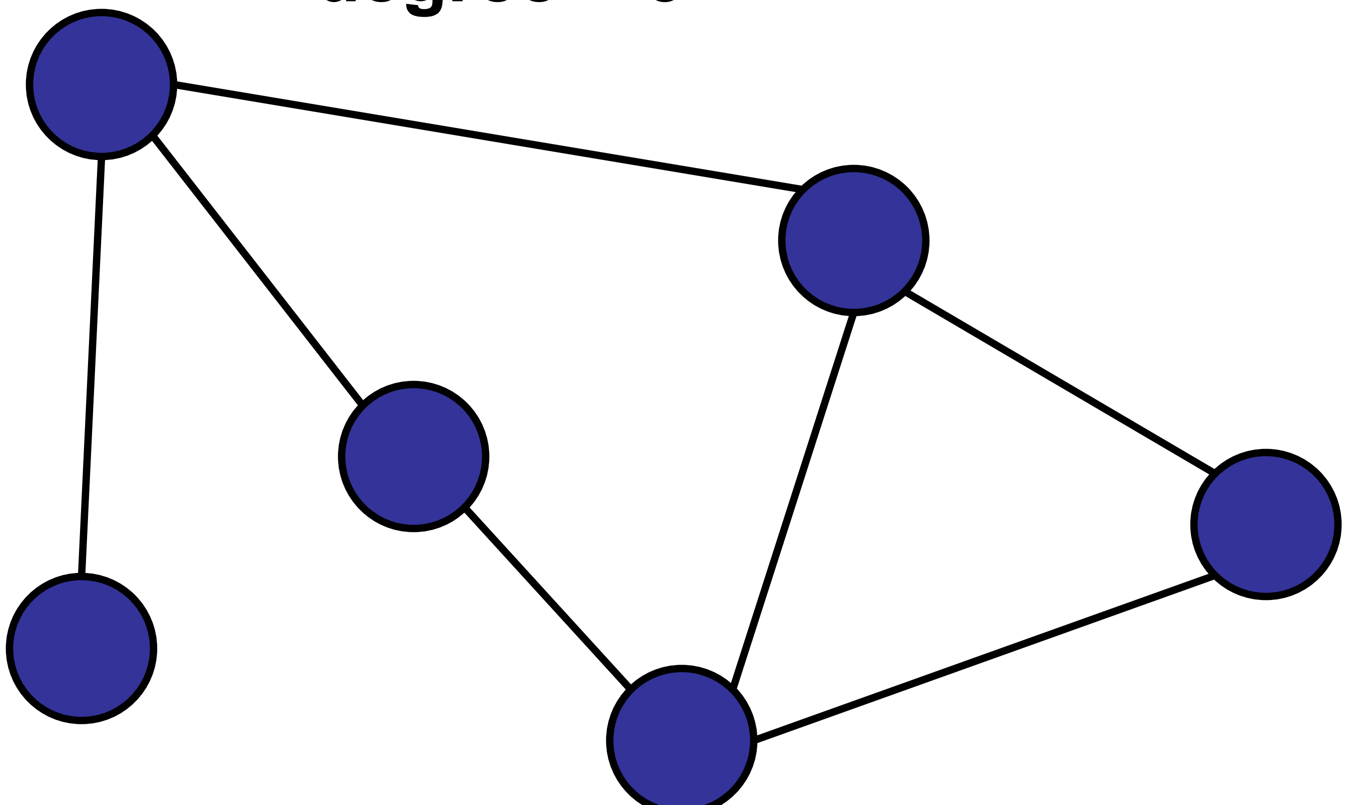


# Graph Terminology

**Degree of a graph**:

– Maximum number of **adjacent** edges.

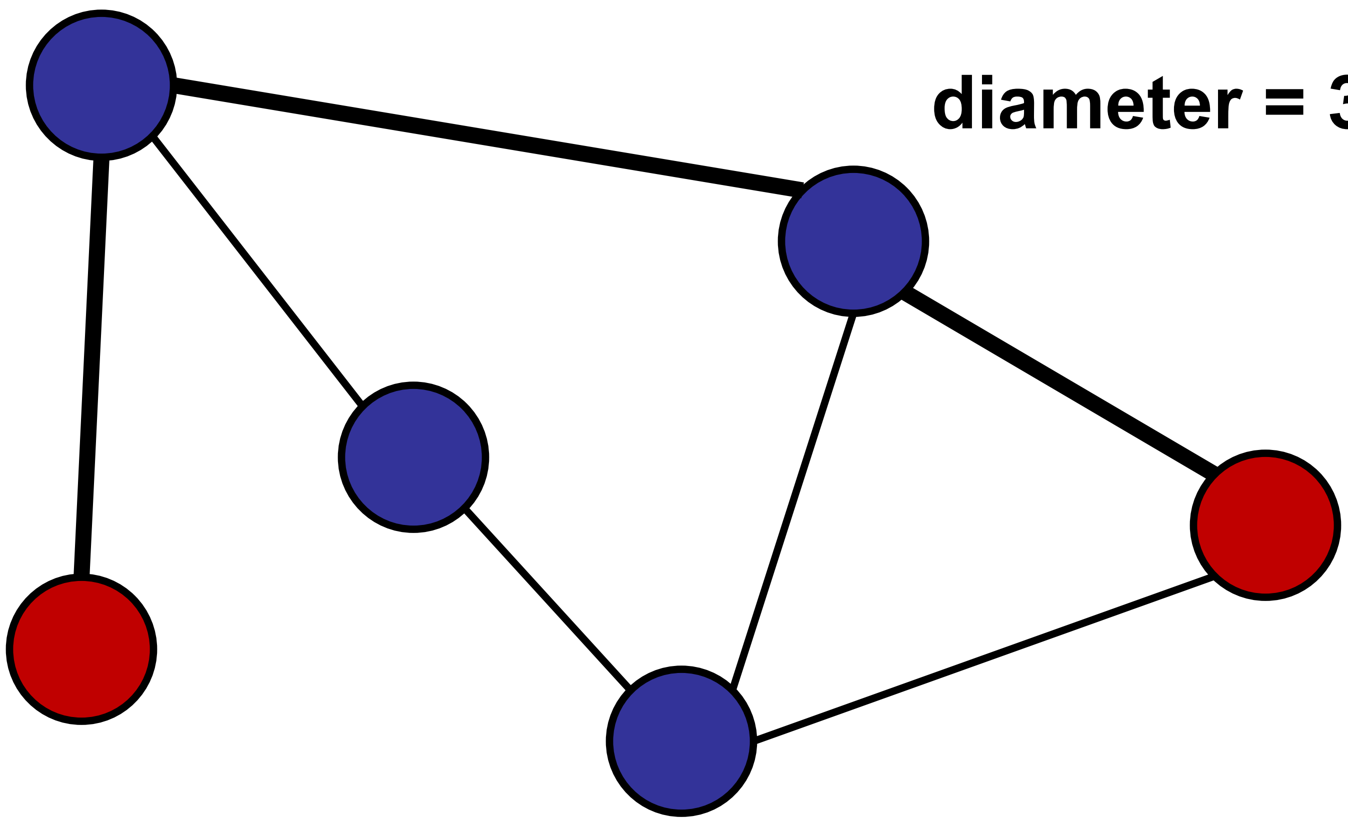
**degree = 3**



# Graph Terminology

**Diameter**:

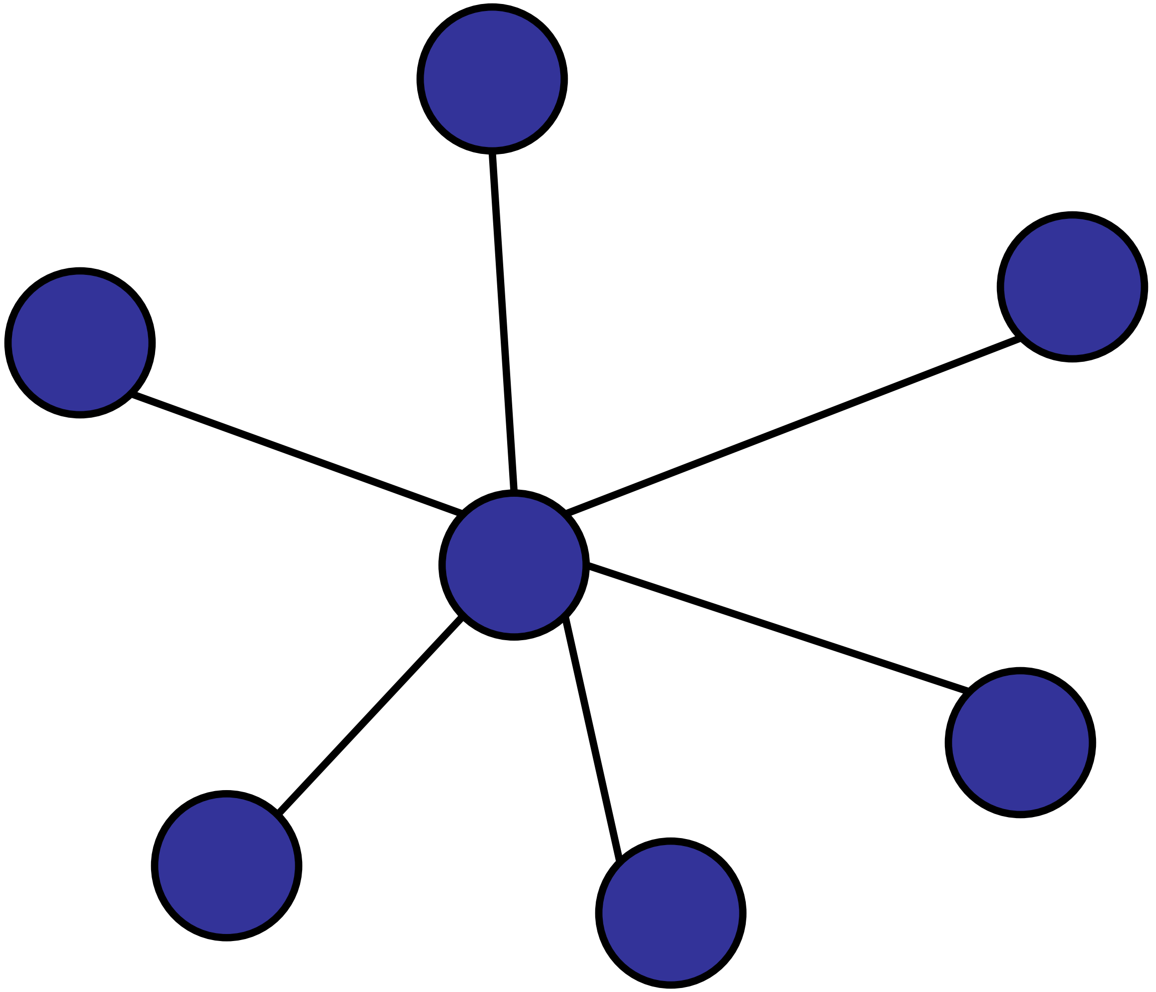
– Maximum distance between two nodes, following the shortest path.



# Special Graphs

# Special Graphs

## Star



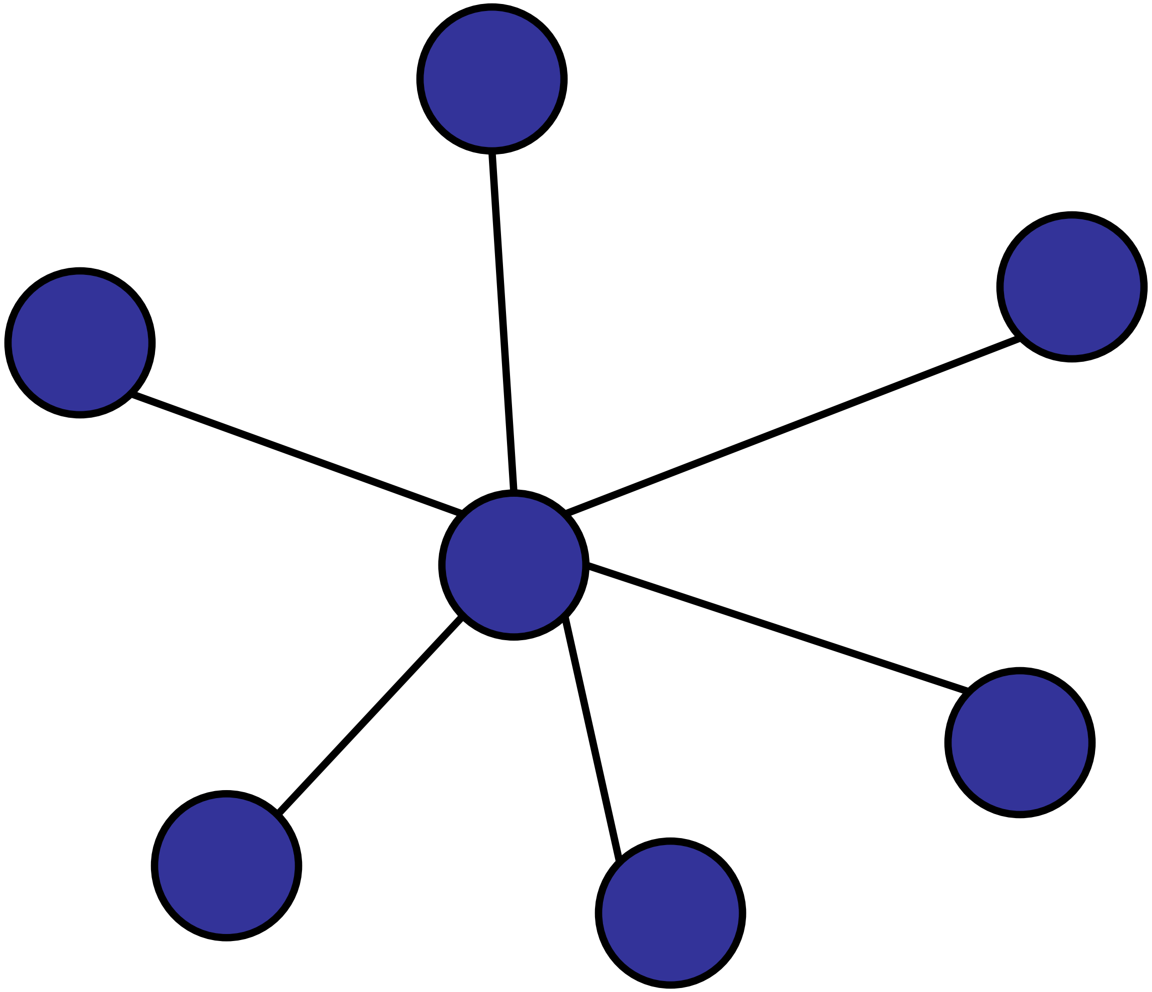
One central node, all edges connect center to edges.

Degree of n-node star is:

1. 1
2. 2
3. n/2
4. n-2
5. n-1
6. n

# Special Graphs

## Star



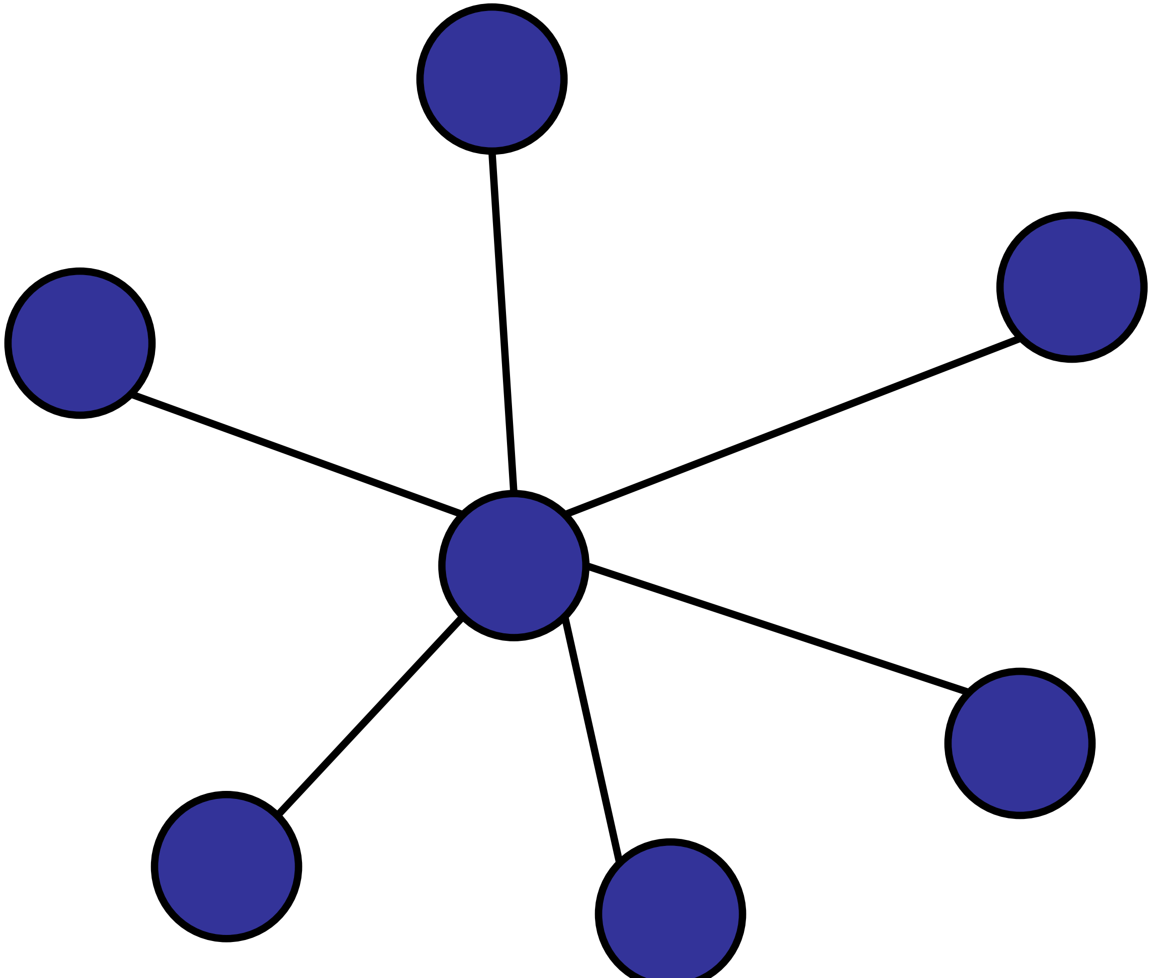
One central node, all edges connect center to edges.

Diameter of n-node star:

1. 1
2. 2
3. n/2
4. n-2
5. n-1
6. n

# Special Graphs

## Star

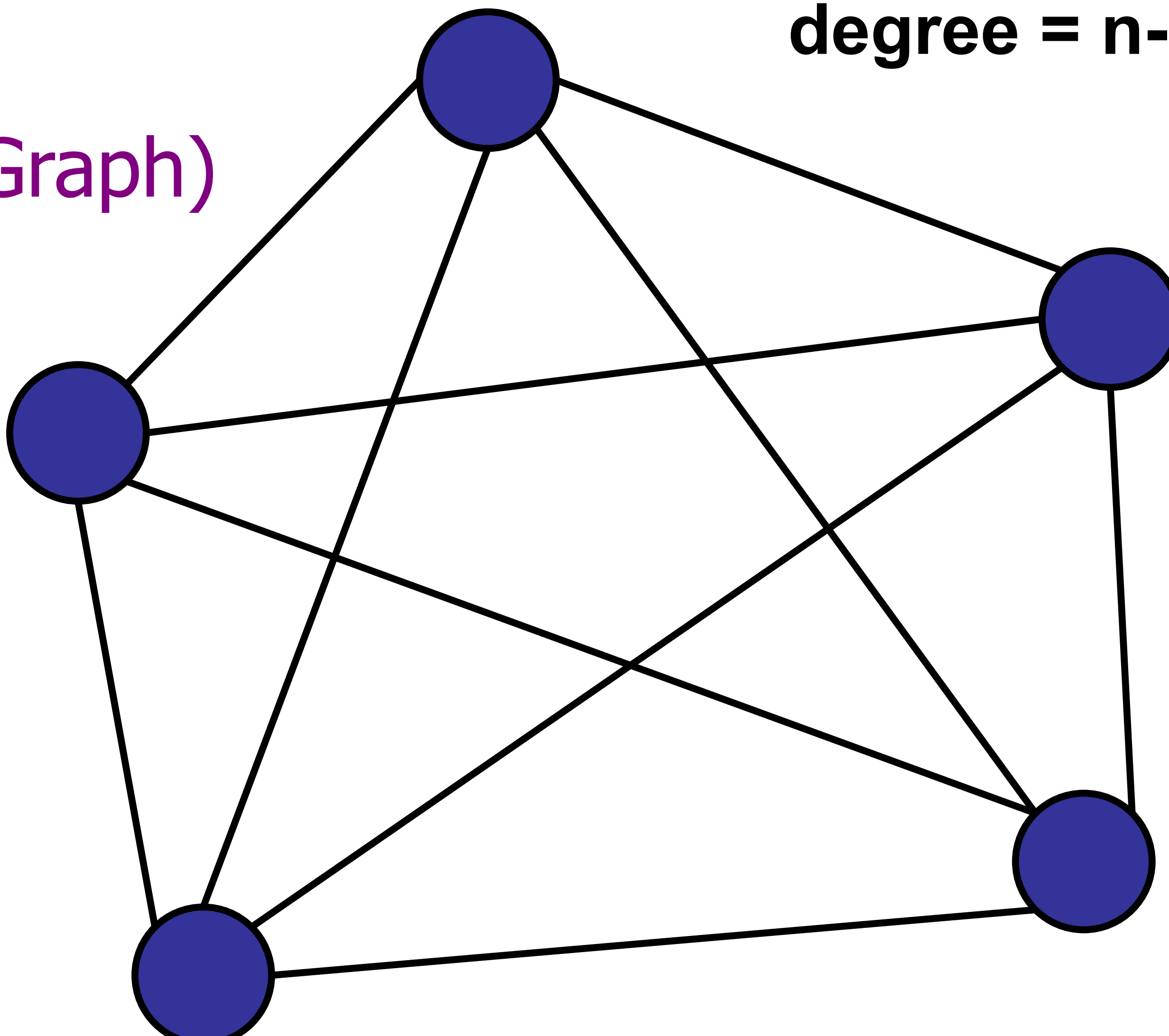


One central node, all edges connect center to edges.

# Special Graphs

Clique
(Complete Graph)

**diameter = 1**
**degree = n-1**



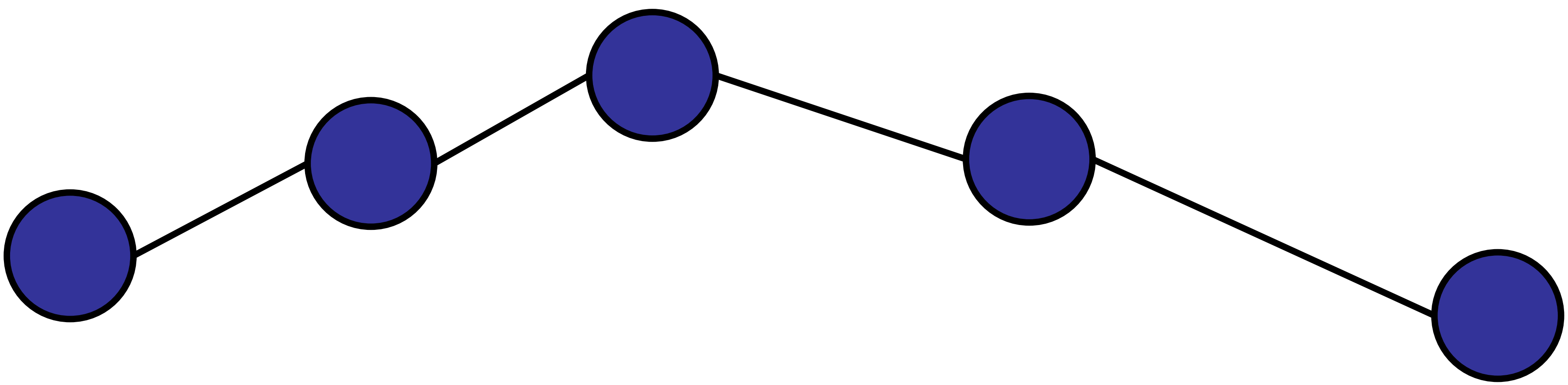
All pairs connected by edges.

# Special Graphs

## Line (or path)

**diameter = n-1**

**degree = 2**

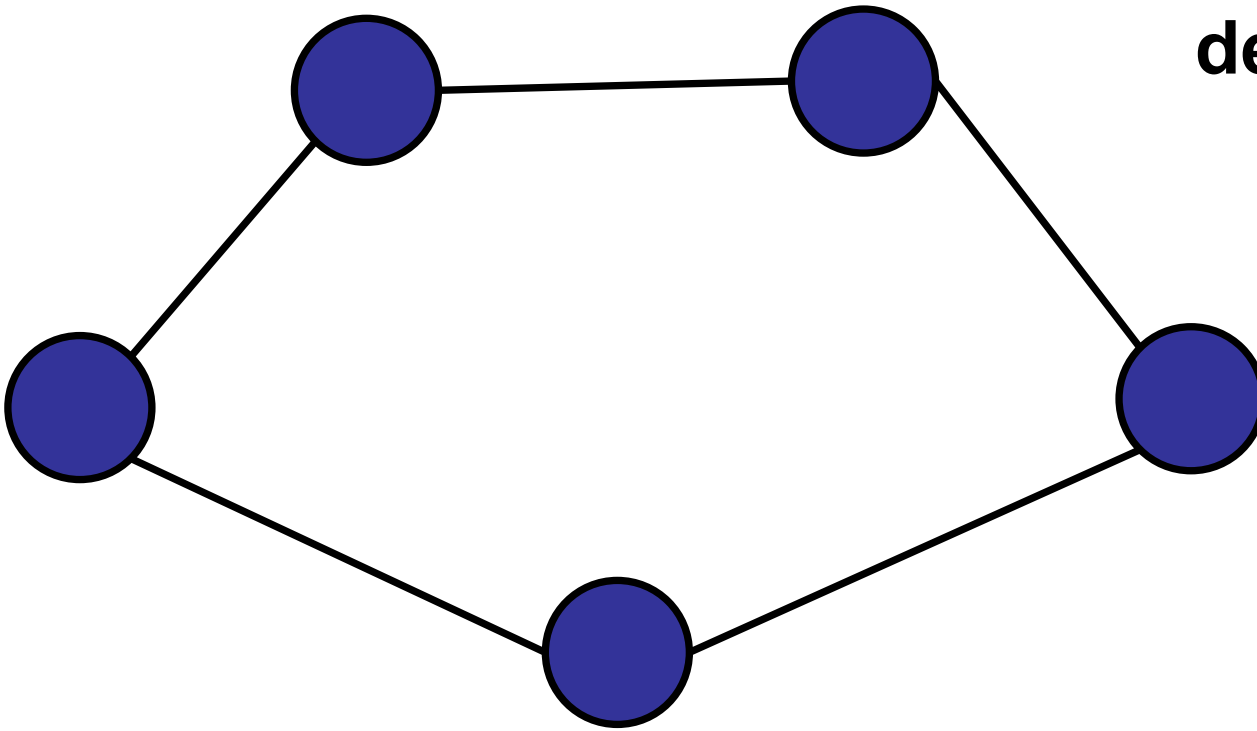


# Special Graphs

## Cycle

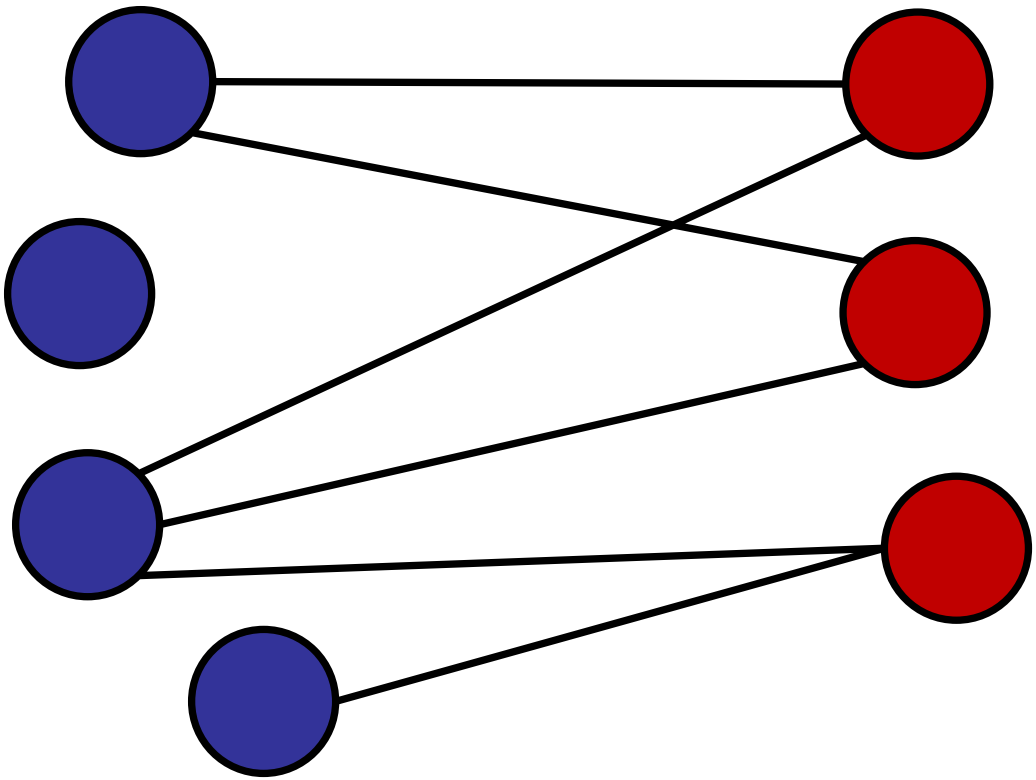
**diameter = n/2**
**or**
**diameter = n/2-1**

**degree = 2**



# Special Graphs

## Bipartite Graph



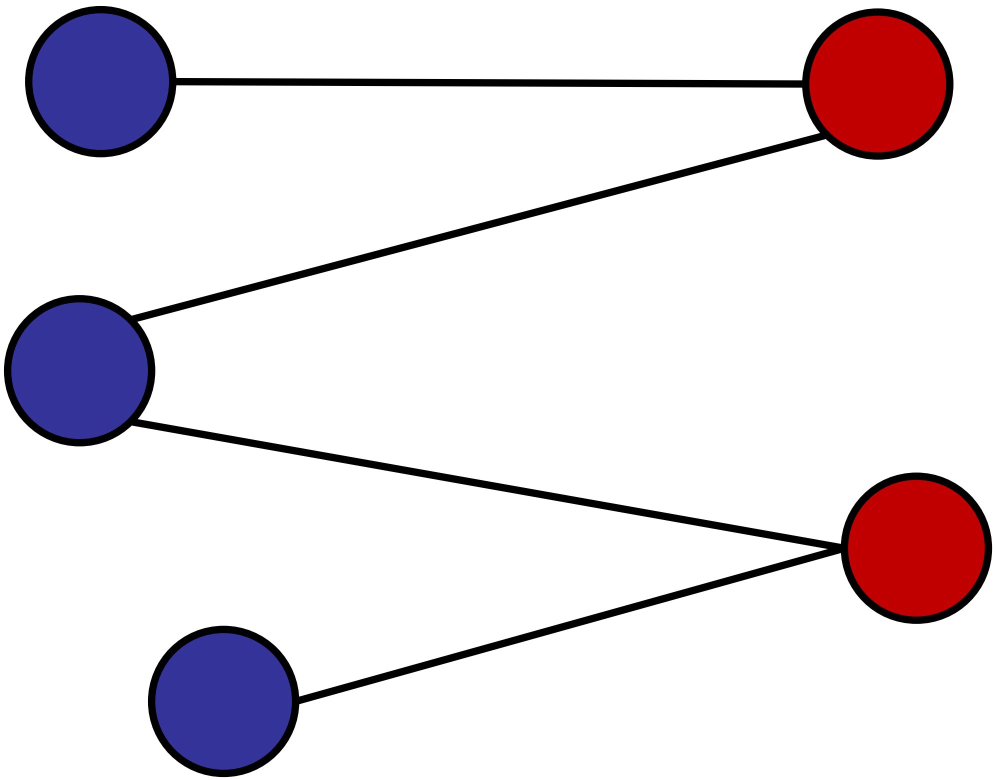
Nodes divided into two sets with no edges between nodes in the same set.

# Max. diameter of n-node bipartite graph is:

1. 1
2. 2
3. n/2-1
4. n/2
5. n-1
6. n

# Special Graphs

## Bipartite Graph



Nodes divided into two sets with no edges between nodes in the same set.

# Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

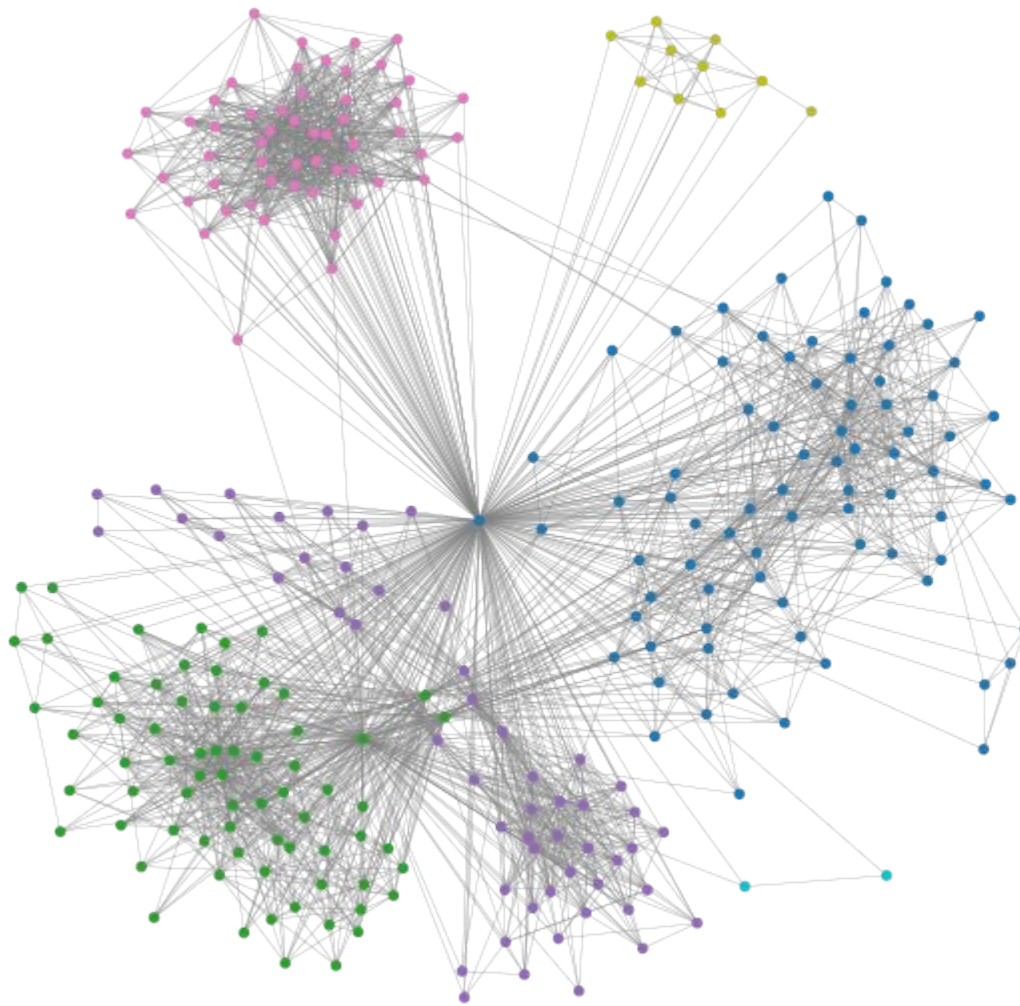
# Where do we find graphs?

(How to model real problems as a graph!)

# Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

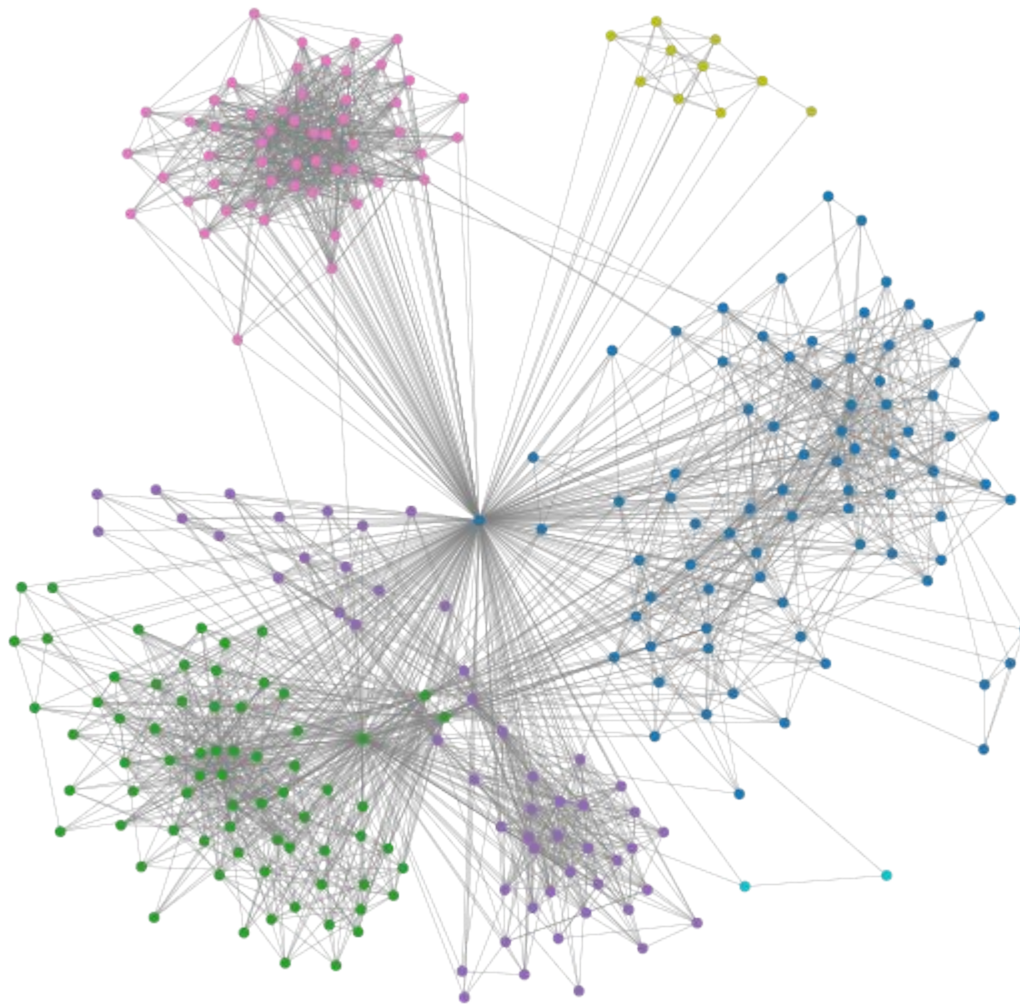
# Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

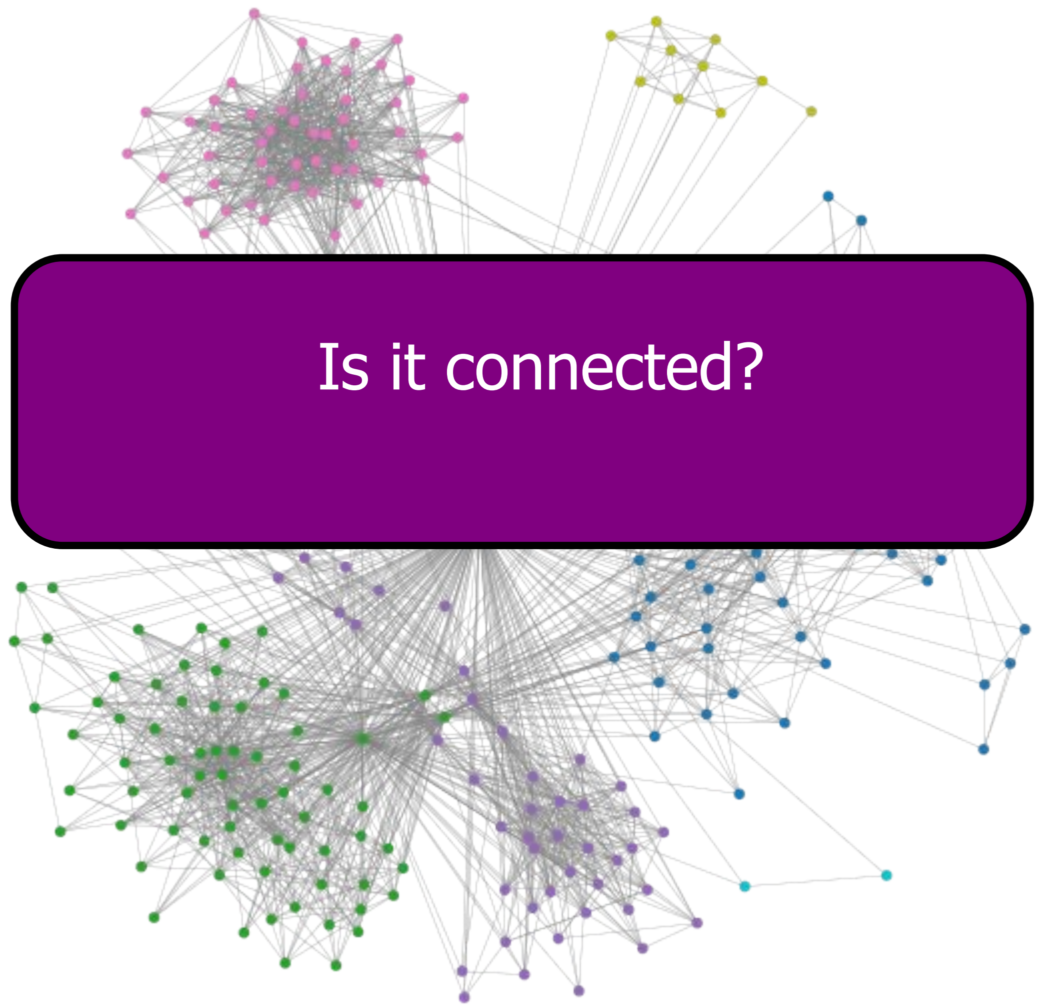
# Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?




Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

# Where do we find graphs?

## Six degrees of separation

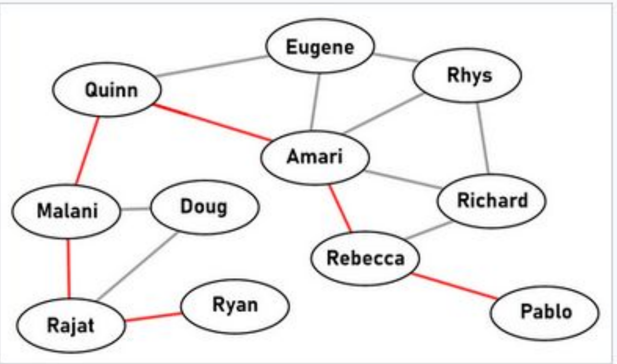
文A 26 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

*For other uses, see Six degrees (disambiguation).*
*Not to be confused with Six degrees of freedom.*

**Six degrees of separation** is the idea that all people are six or fewer social connections away from each other. As a result, a chain of "friend of a friend" statements can be made to connect any two people in a maximum of six steps. It is also known as the **six handshakes rule**.[1] Mathematically it means that a person shaking hands with 30 people, and then those 30 shaking hands with 30 other people, would after repeating this 6 times allow every person in a population as large as the United States to have shaken hands (7 times for the whole world).




A map of several branches and degrees of a small social group: Ryan is six degrees of separation from Pablo

The concept was originally set out in a 1929 short story by Frigyes Karinthy, in which a group of people play a game of trying to connect any person in the world to themselves by a chain of five others. It was popularized in John Guare's 1990 play *Six Degrees of Separation*.

# Where do we find graphs?

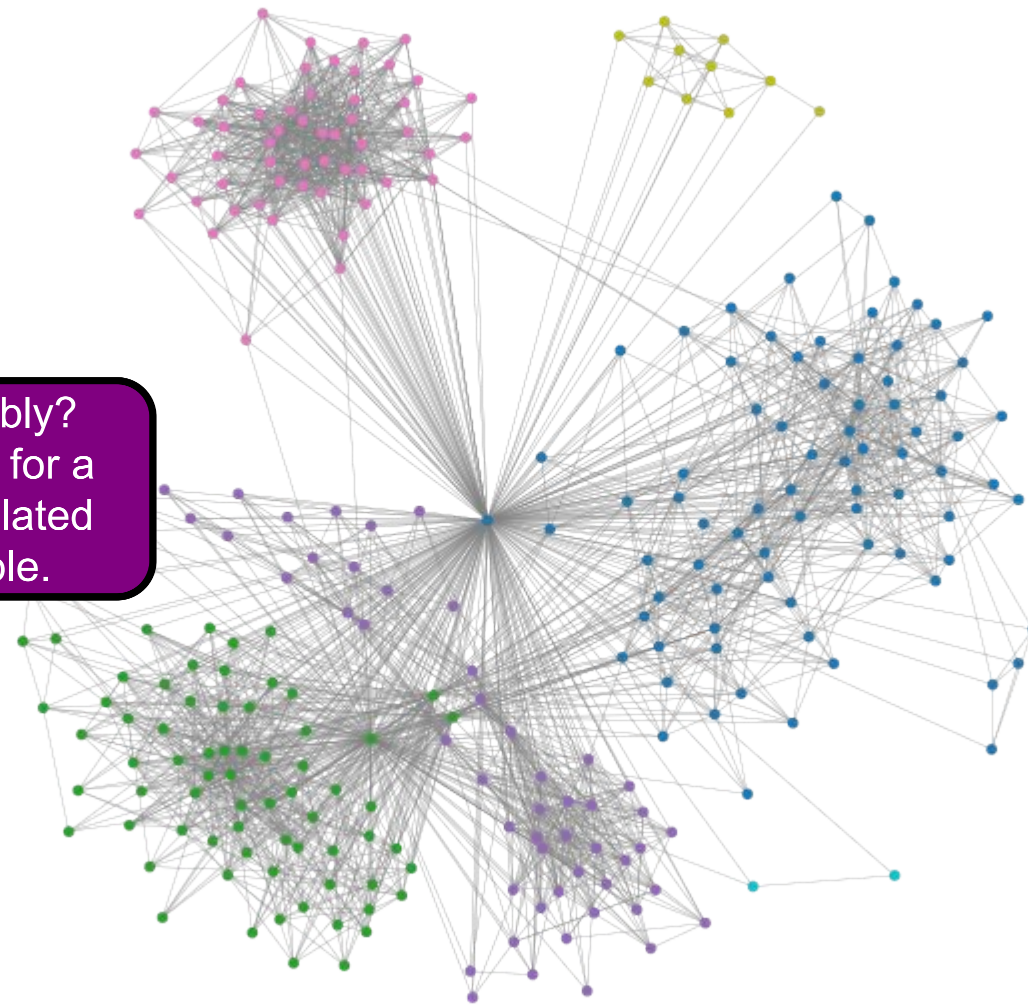
Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?

Probably? Except for a few isolated people.

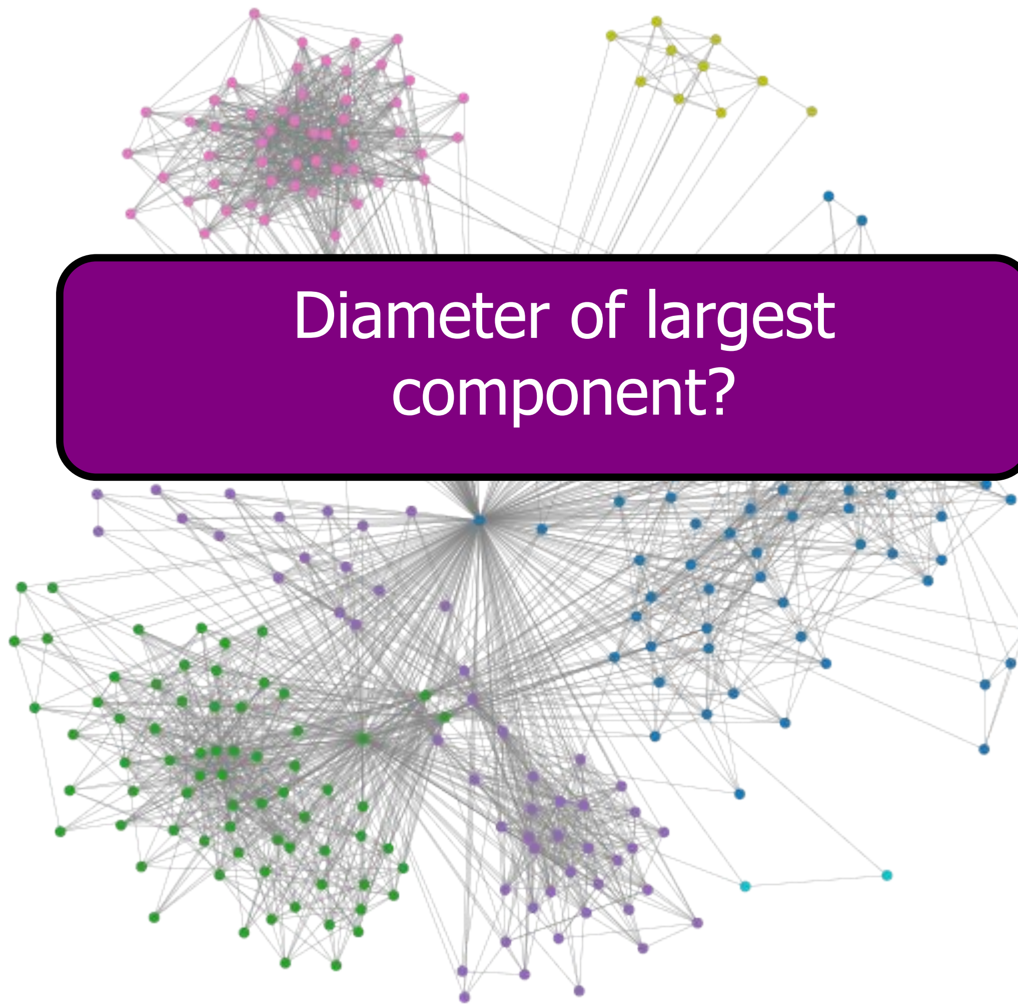


Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

# Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship




Questions:

- Connected?
- Diameter?
- Degree?

Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

# Where do we find graphs?

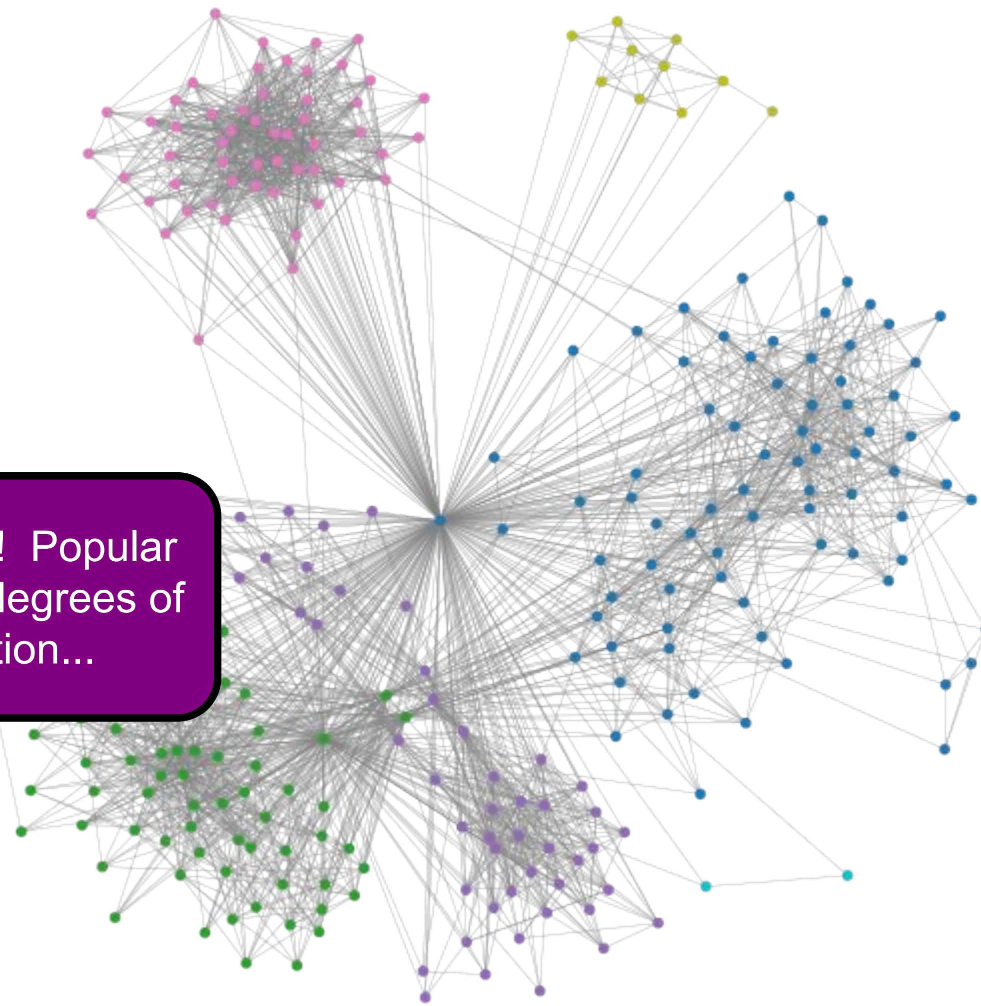
Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?

Fairly small! Popular saying: six degrees of separation...



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

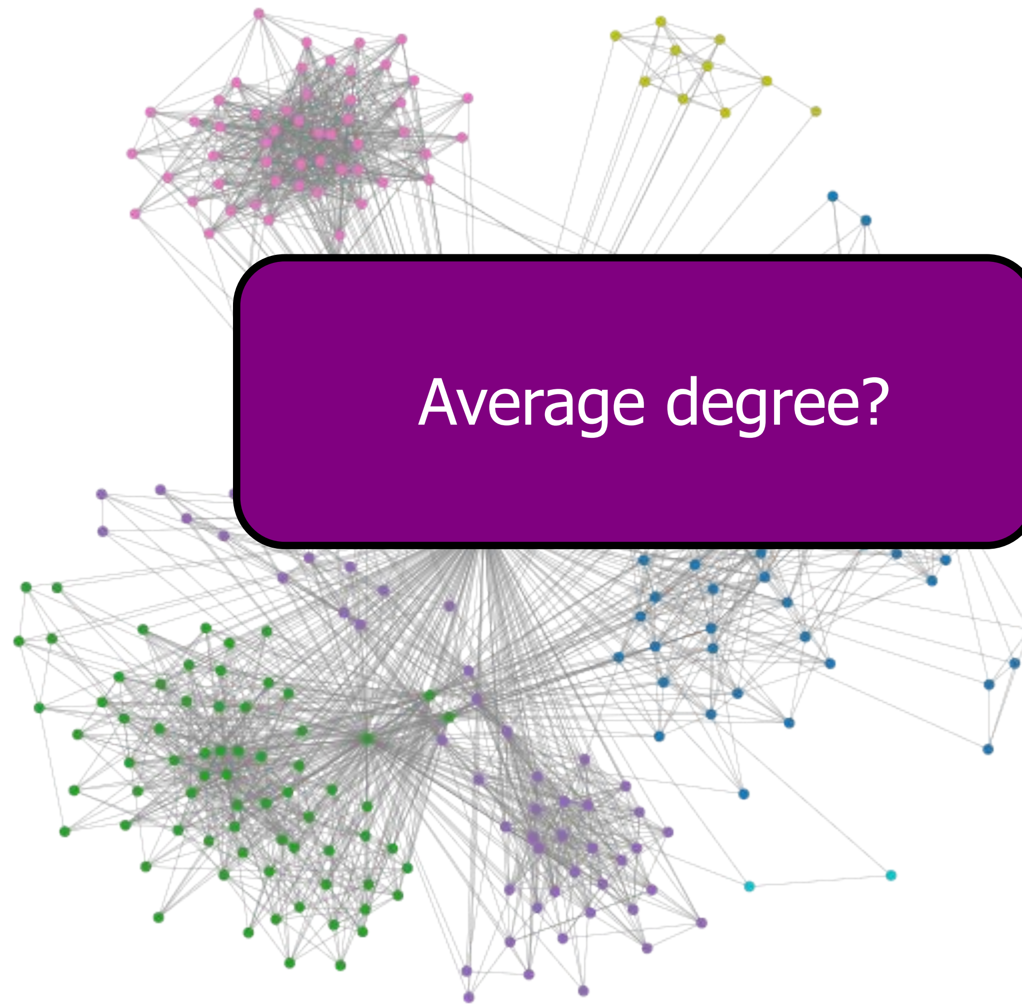
# Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?



Average degree?

Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

# Where do we find graphs?

## Social network:

- Nodes are people
- Edge = friendship

## Questions:

- Connected
- Diameter?
- Degree?

What does big mean? Probably very small compared to number of nodes. (Graph is sparse.)

Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

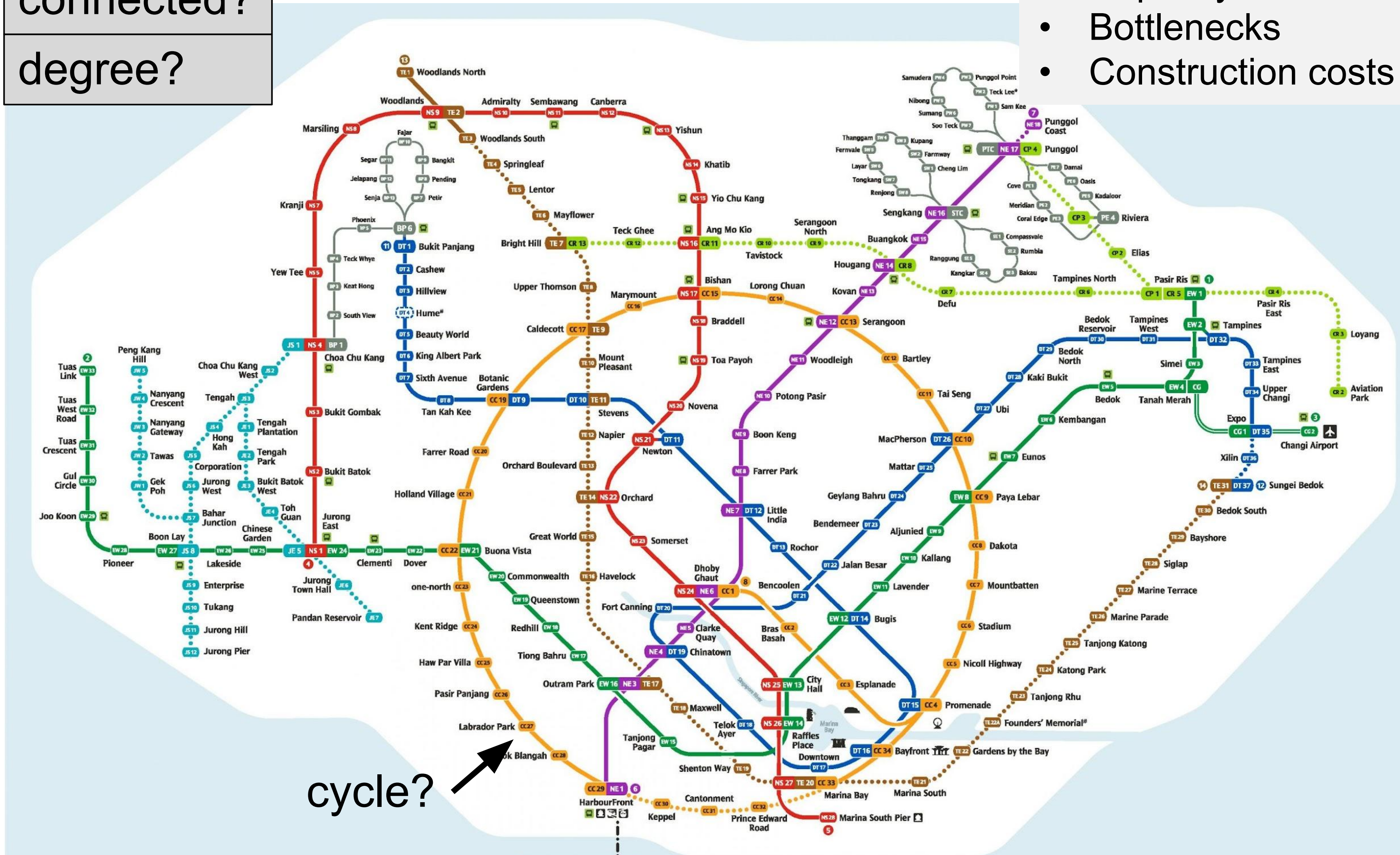
# Transportation Network

connected?

degree?

Questions:

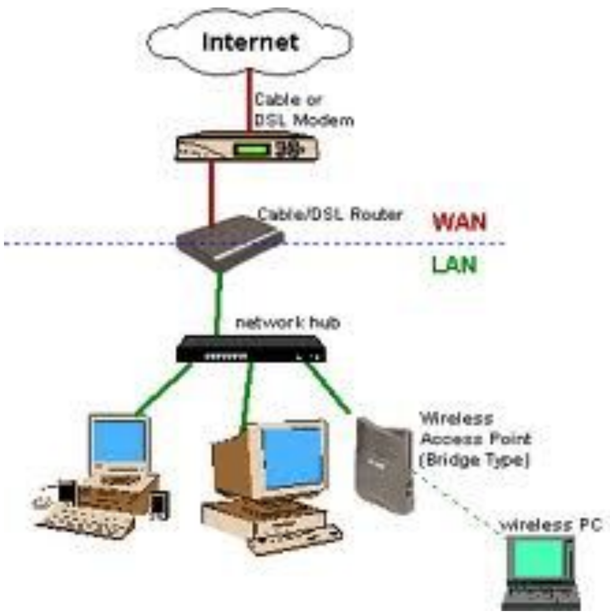
- Shortest path
- Capacity
- Bottlenecks
- Construction costs

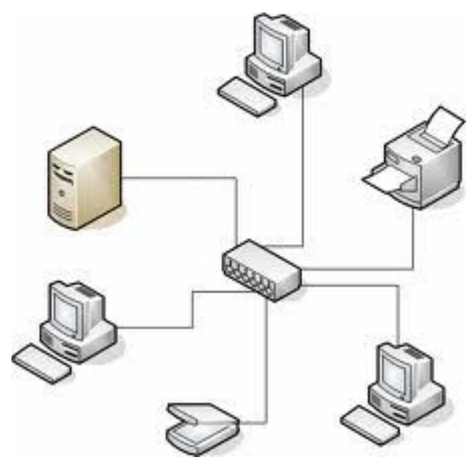


# Internet / Computer Networks

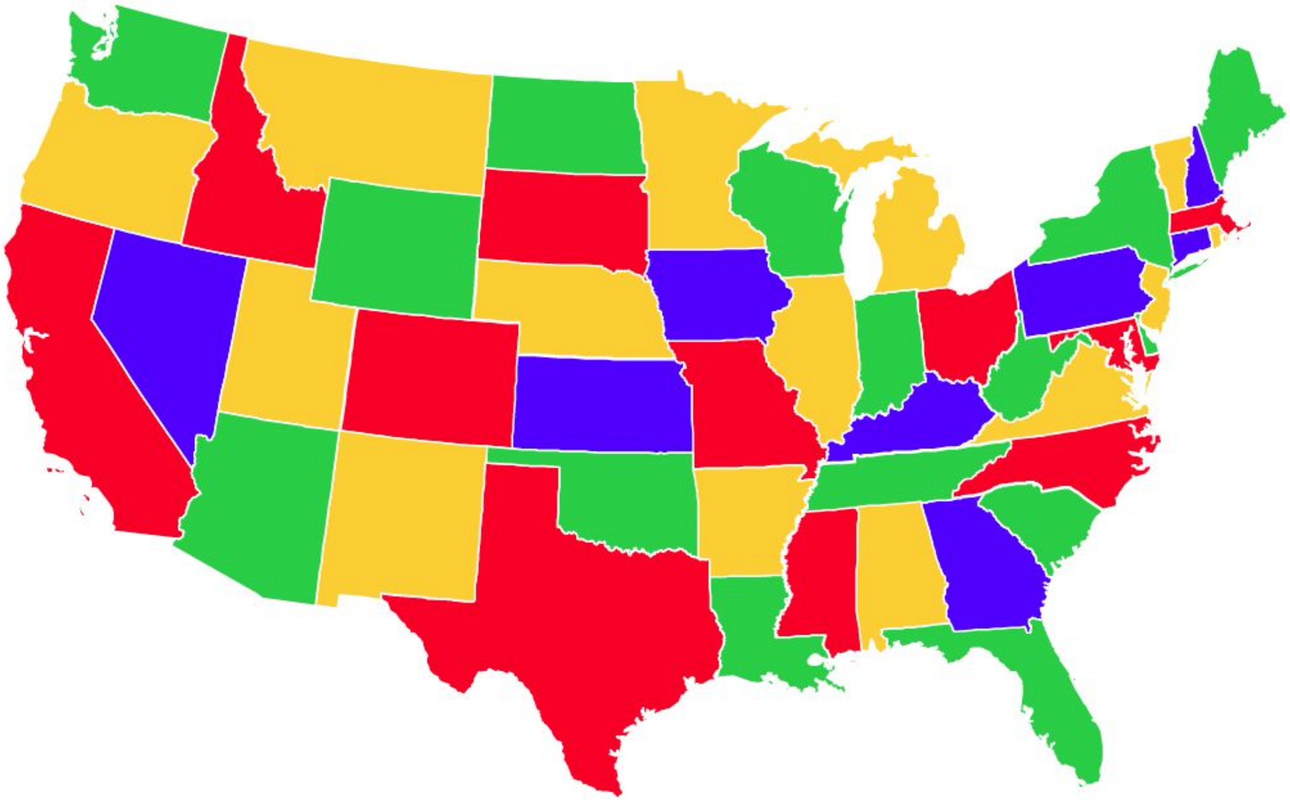
Questions:

- How to find routes for packets?
- How to maximise data flow rate from multiple servers to multiple clients?



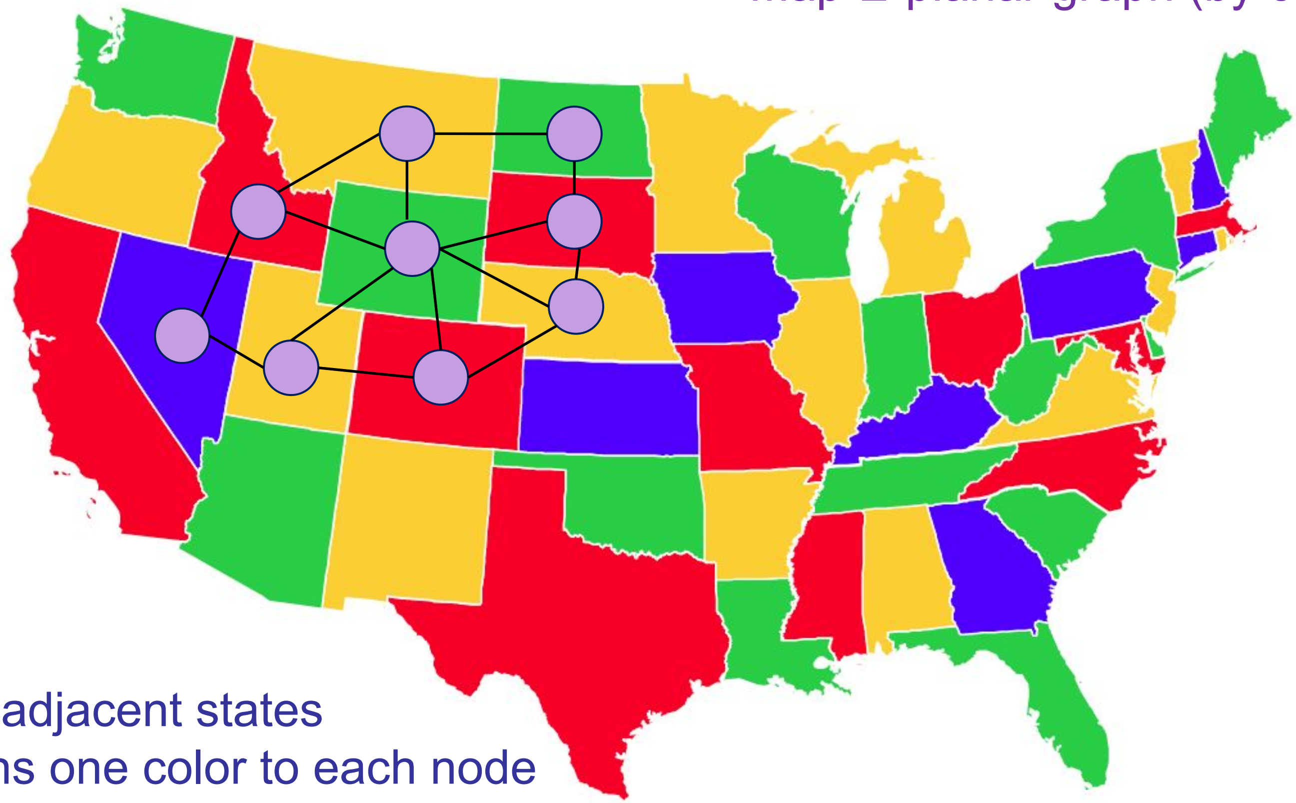
# Optimization: 4-Coloring



Can you color a map using only 4-colors so that no two adjacent countries/states have the same color?

# 4-Coloring Theorem:

Map □ planar graph (by construction)

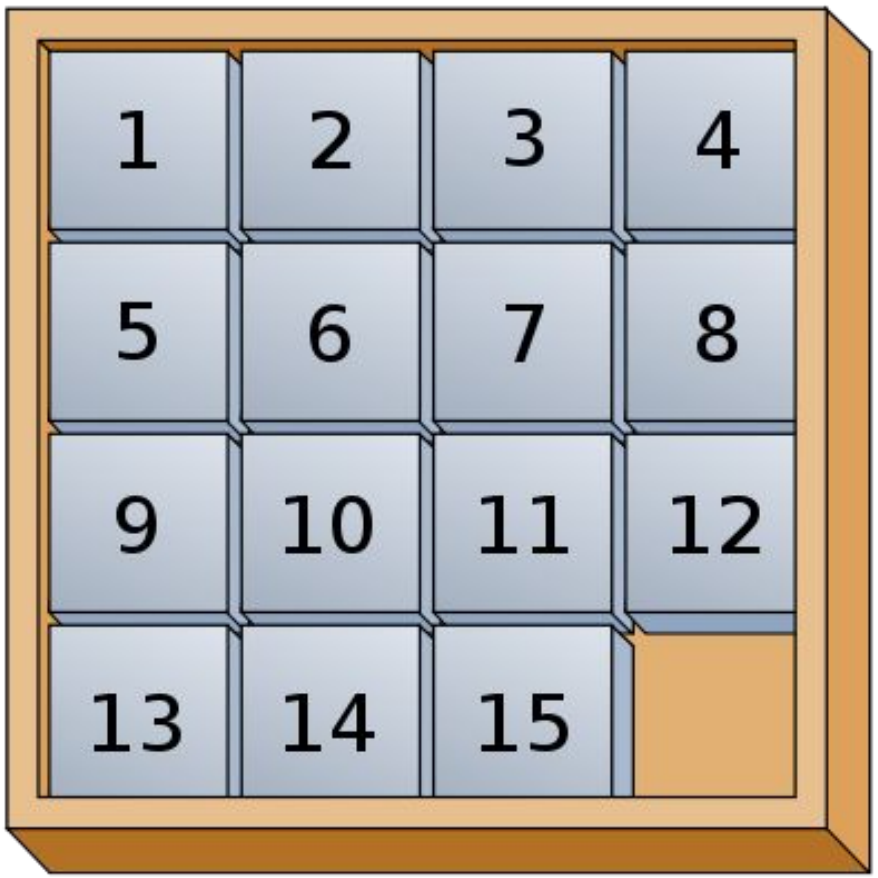


Nodes: states
Edges: pairs of adjacent states
Coloring: assigns one color to each node

For any planar graph, you can color it using only 4-colors so that no two adjacent countries/states have the same color?

Can be drawn on a 2d-plane with no crossing edges.

# Sliding Puzzle



# Sliding Puzzle

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

# Sliding Puzzle

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

# Sliding Puzzle

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

# Sliding Puzzle

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

# Sliding Puzzle

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

# Sliding Puzzle

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

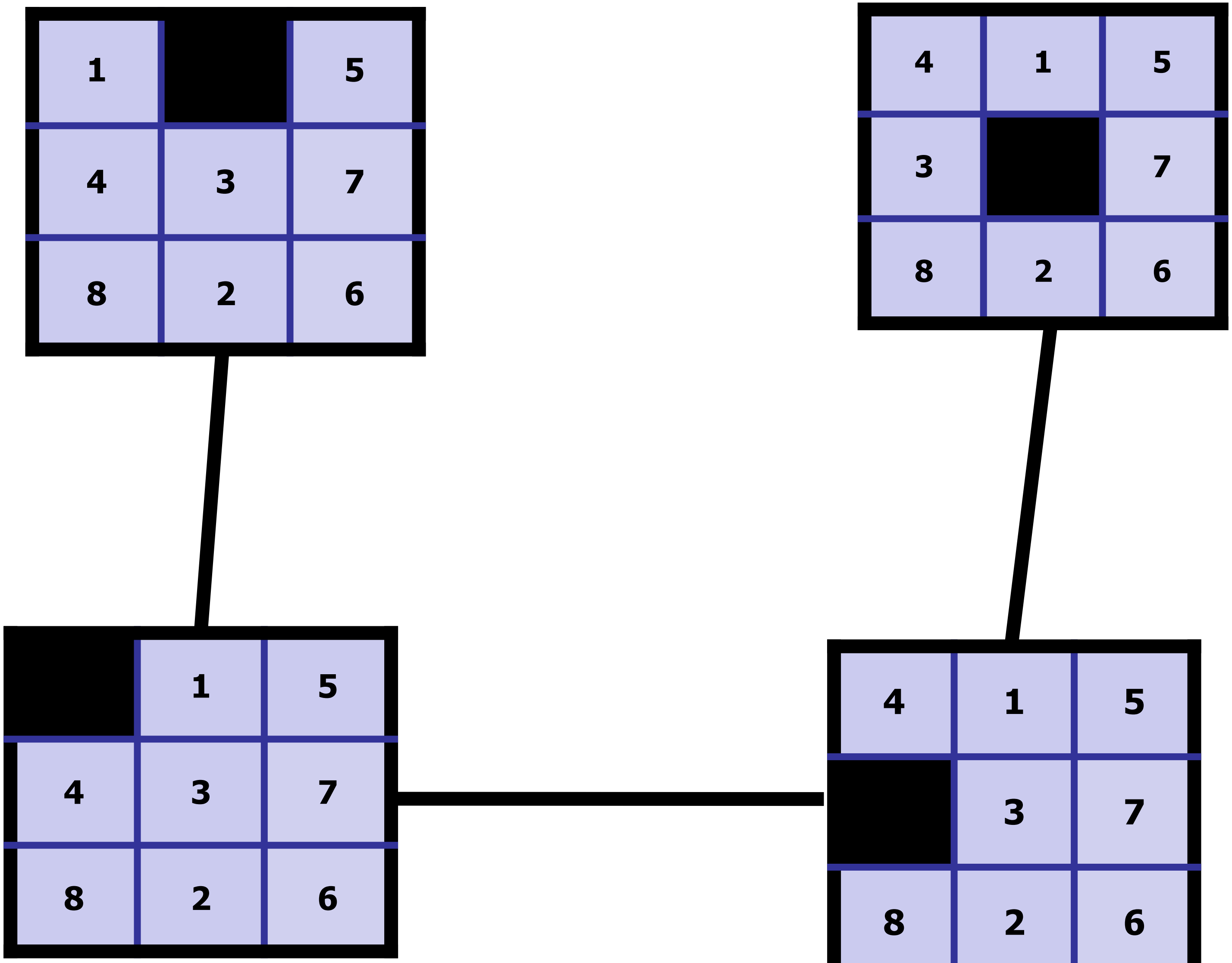
# Sliding Puzzle

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

# Sliding Puzzle

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

# Sliding Puzzle is a Graph



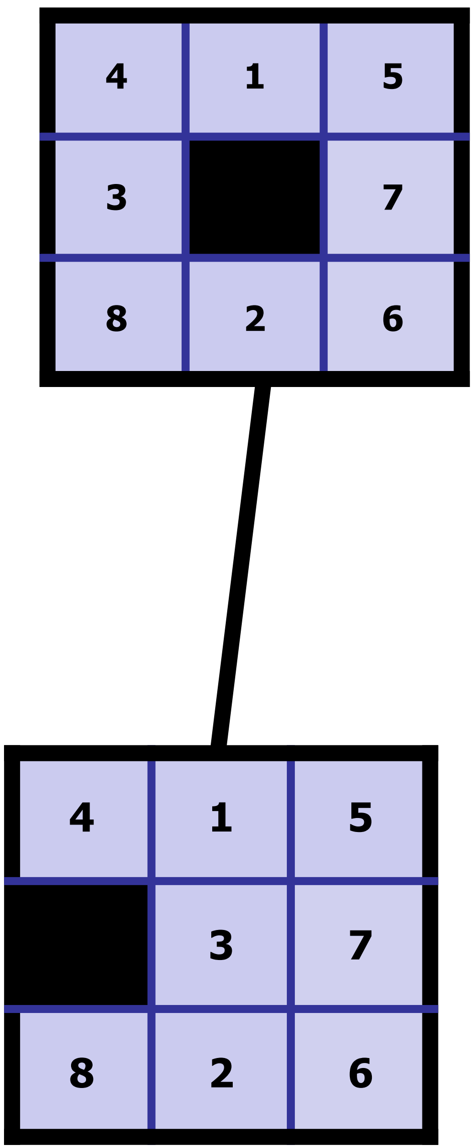
# Sliding Puzzle

Nodes:

- State of the puzzle
- Permutation of nine tiles

Edges:

- Two states are edges if they differ by only one move.



What is the maximum degree of the Sliding Puzzle graph?

1. 1
2. 2
3. 3
4. ✓ 4
5. n/2
6. n
7. n!

Can either slide from above/below/left/right

# Sliding Puzzle

Nodes:

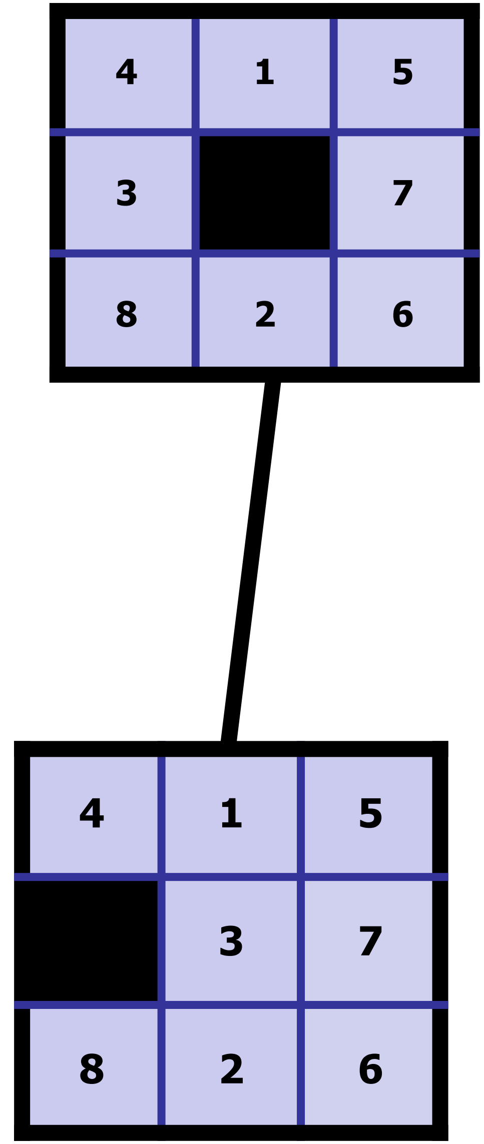
- State of the puzzle
- Permutation of nine tiles + 1 blank tile

Edges:

- Two states are edges if they differ by only one move.

Nodes = 9! = 362880

Edges < 4*9! < 1451520



# Sliding Puzzle

Number of moves to solve the puzzle?

Initial, scrambled state:

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

Final, unscrambled state:

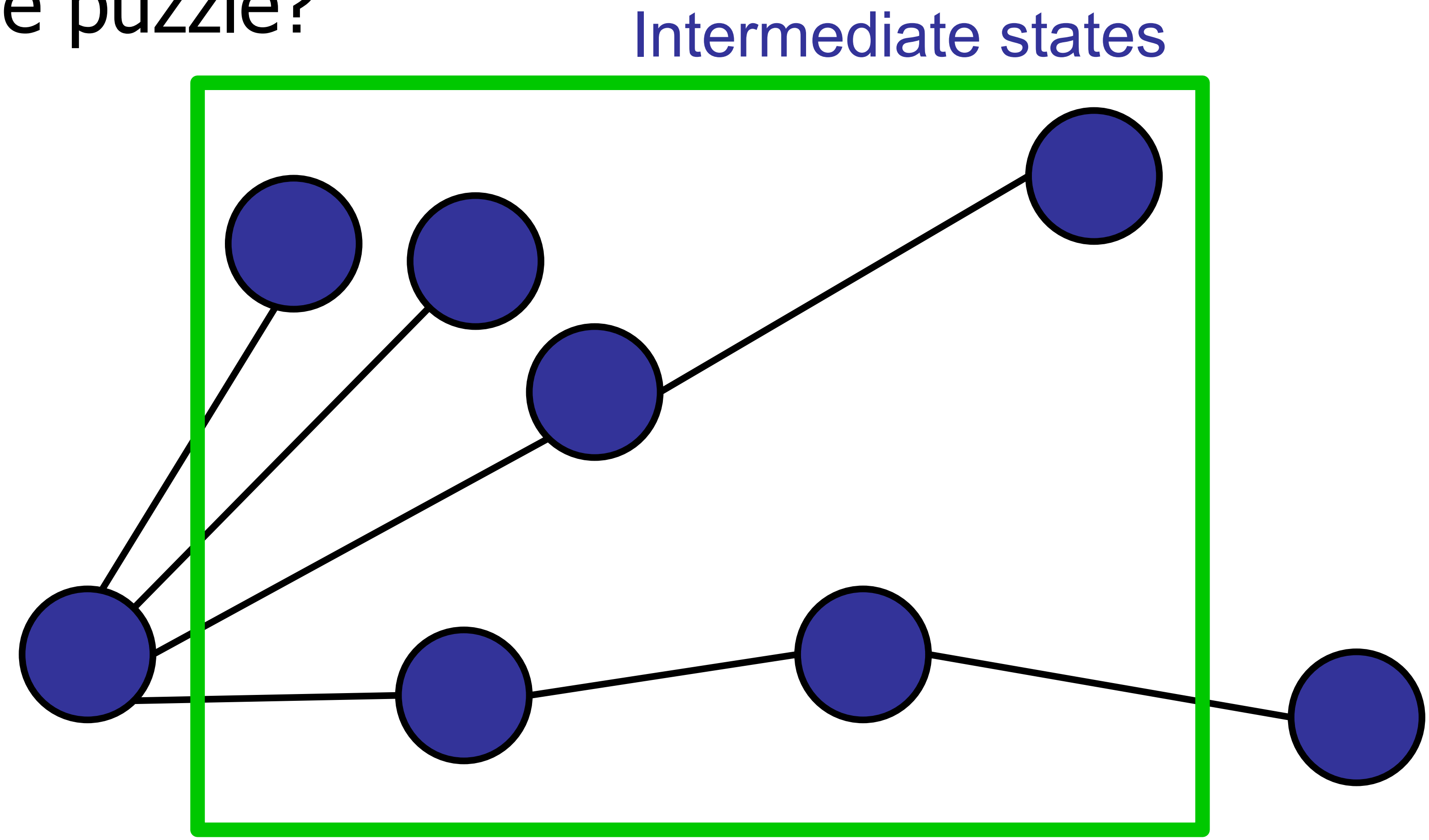
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

# Sliding Puzzle

Number of moves to solve the puzzle?

# Sliding Puzzle

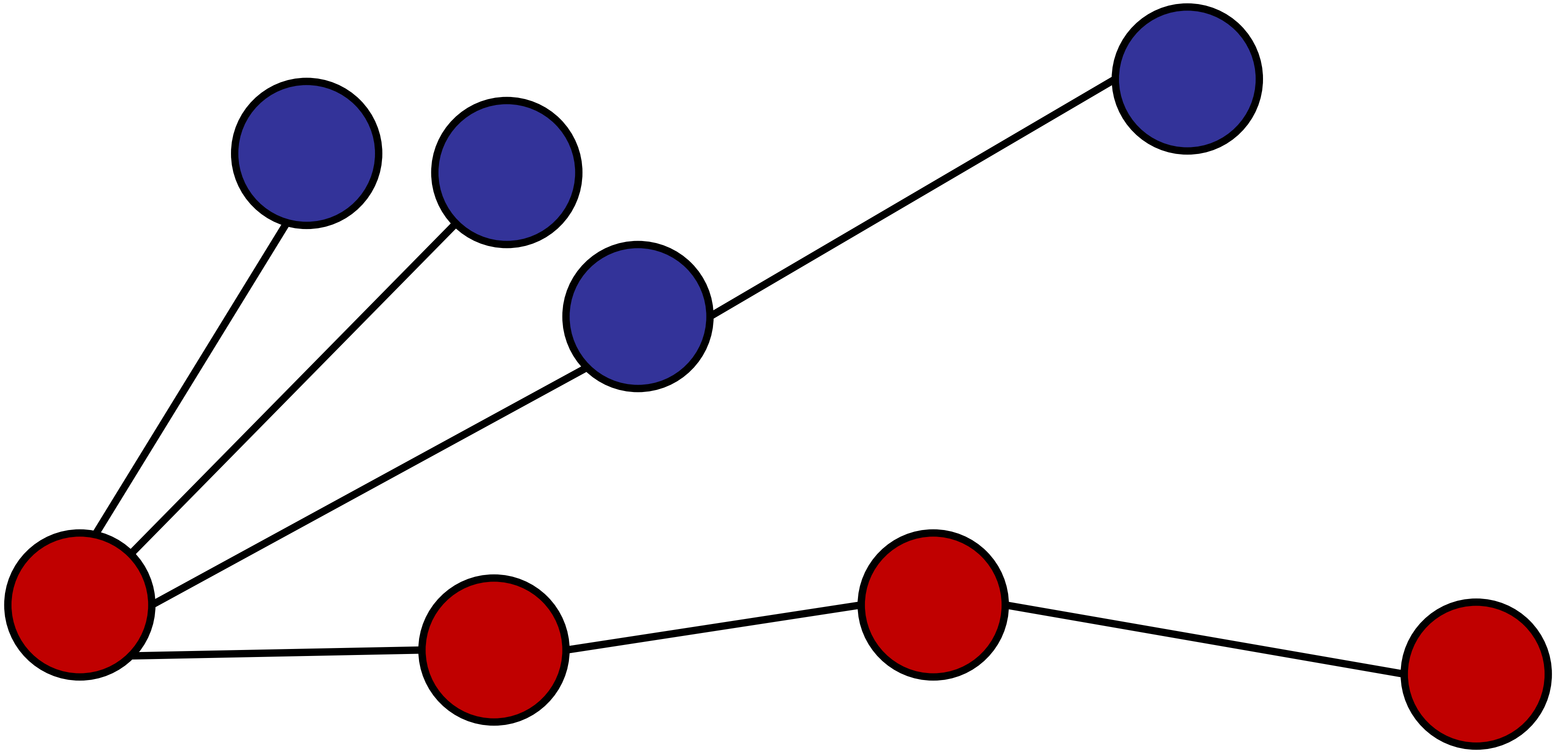
Number of moves to solve the puzzle?



# Sliding Puzzle

Number of moves to solve the puzzle?

= Shortest path from starting state to final state



# Sliding Puzzle

Maximum number of moves needed to solve puzzle from **any** possible input?

Initial, scrambled state:

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

Final, unscrambled state:

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

# Sliding Puzzle

Maximum number of moves needed to solve puzzle from **any** possible input?

≤ diameter of graph

Initial, scrambled state:

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

Final, unscrambled state:

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

# 2 x 2 x 2 Rubik's Cube

# 2 x 2 x 2 Rubik's Cube

Record solve time: 0.69 seconds

# 2 x 2 x 2 Rubik's Cube

Configuration Graph

- Vertex for each possible state
- Edge for each basic move
  - 90 degree turn
  - 180 degree turn

Puzzle: given initial state, find a path to the solved state.

# 2 x 2 x 2 Rubik's Cube

How many vertices?

$$8! \cdot 3^8 = 264{,}539{,}520$$

\# cubelets

Each cubelet is
in one of 8 positions.

Each of the 8 cubelets
can be in one of three
orientations

# 2 x 2 x 2 Rubik's Cube

How many vertices?

$7! \cdot 3^7 =$

11,022,480

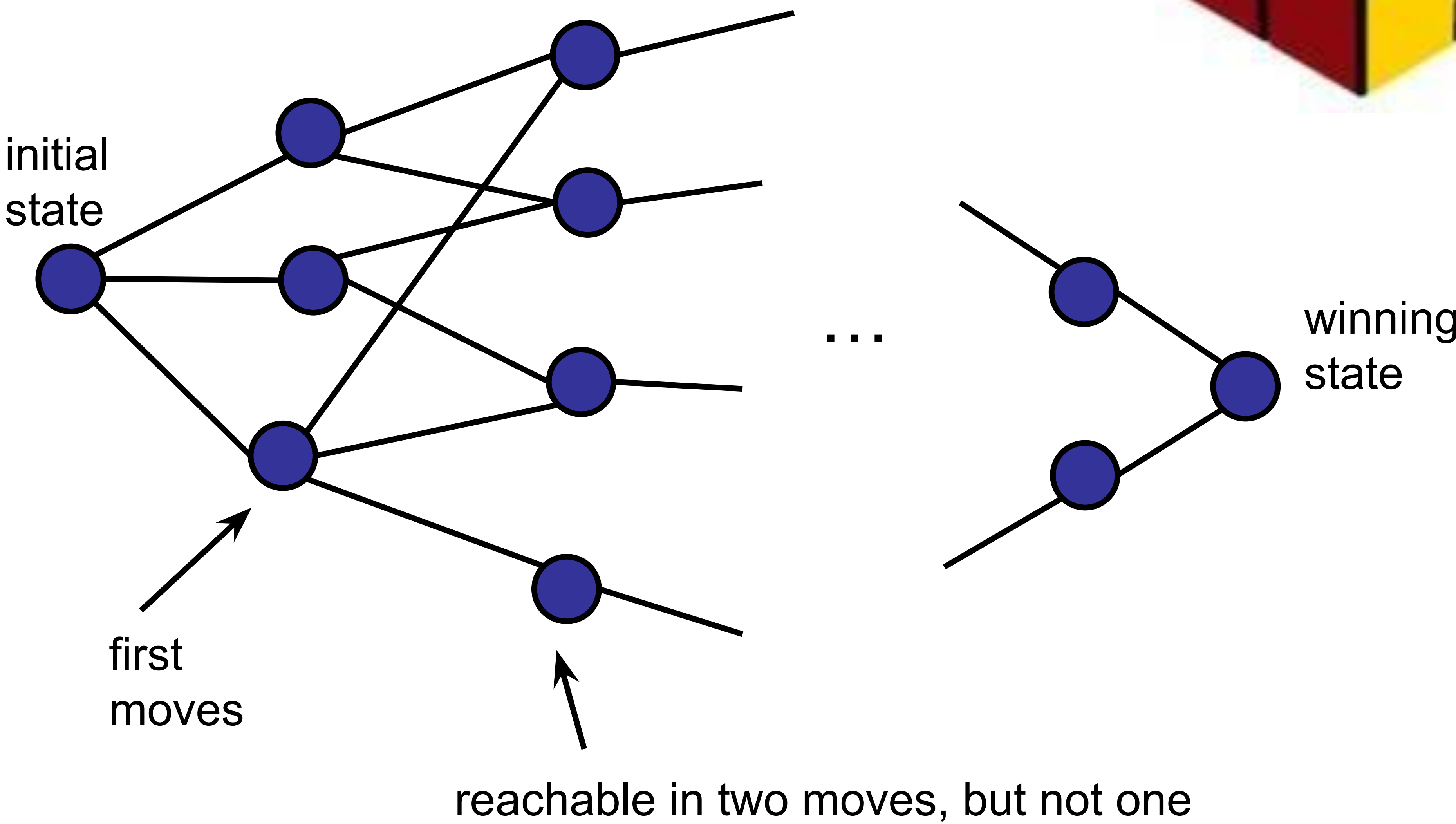
Symmetry:

Fix one cubelet.
As reference

Each of the 8 cubelets can be in one of three orientations

# 2 x 2 x 2 Rubik's Cube

Geography of Rubik's configurations:



# Reachable configurations

| Distance | 90 deg. turns | 90/180 deg. turns |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 6 | 9 |
| 2 | 27 | 54 |
| 3 | 120 | 321 |
| 4 | 534 | 1,847 |
| 5 | 2,256 | 9,992 |
| 6 | 8,969 | 50,136 |
| 7 | 33,058 | 227,536 |
| 8 | 114,149 | 870,072 |
| 9 | 360,508 | 1,887,748 |
| 0 | 930,588 | 623,800 |
| 11 | 1,350,852 | 2,644 |
| 12 | 782,536 | |
| 13 | 90,280 | |
| 14 | 276 | |

diameter

# Reachable configurations

| Distance | 90 deg. turns | 90/120 deg. turns |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 6 | 9 |
| 2 | 27 | 54 |
| 9 | 360,508 | 1,887,748 |
| 0 | 930,588 | 623,800 |
| 11 | 1,350,852 | 2,644 |
| 12 | 782,536 | |
| 13 | 90,280 | |
| 14 | 276 | |

diameter

Challenge:
How do you generate this table?

# 3 x 3 x 3 Rubik's Cube

<u>Configuration Graph</u>

- 43 quintillion vertices (approximately)
- Diameter: 20
  - 1995: require at least 20 moves.
  - 2008: 20 moves is enough from every position.
  - Using Google server farm.
  - 35 CPU-years of computation.
  - 20 seconds / set of 19.5 billion positions.
  - Lots of mathematical and programming tricks.

# 3 x 3 x 3 Rubik's Cube

What is the diameter of an (n x n x n) cube?

$\Theta(n^2 / \log n)$

# Representing a Graph

Question: How should we represent a graph?

- What data structures should we create?
- Which operations on the graph should we support?

# Representing a Graph

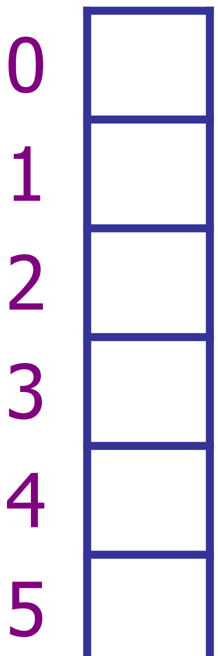
Graph consists of:

- Nodes
- Edges

# Representing a Graph

Graph consists of:

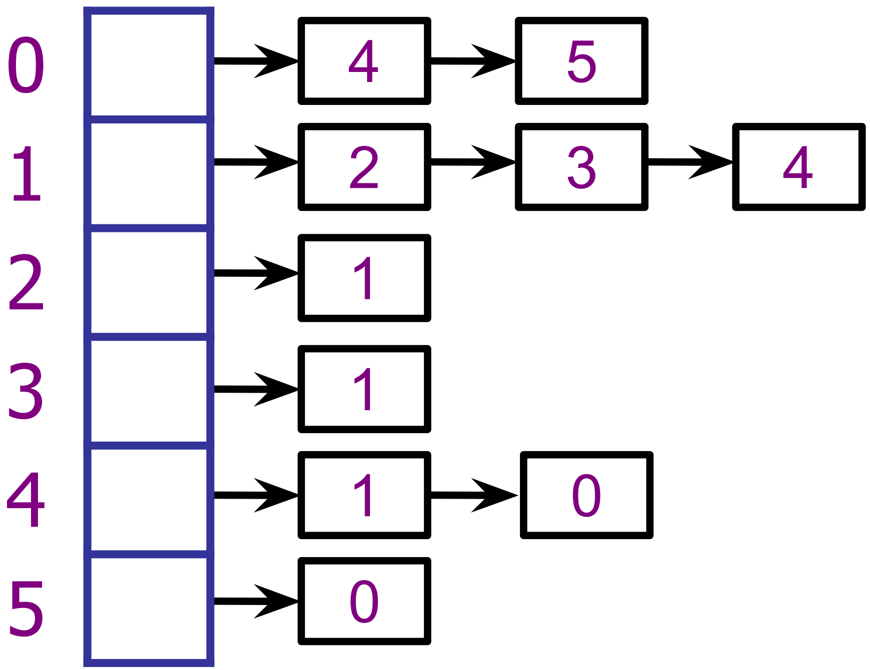
- Nodes: typically ID ranges from [0, n-1]
- Edges

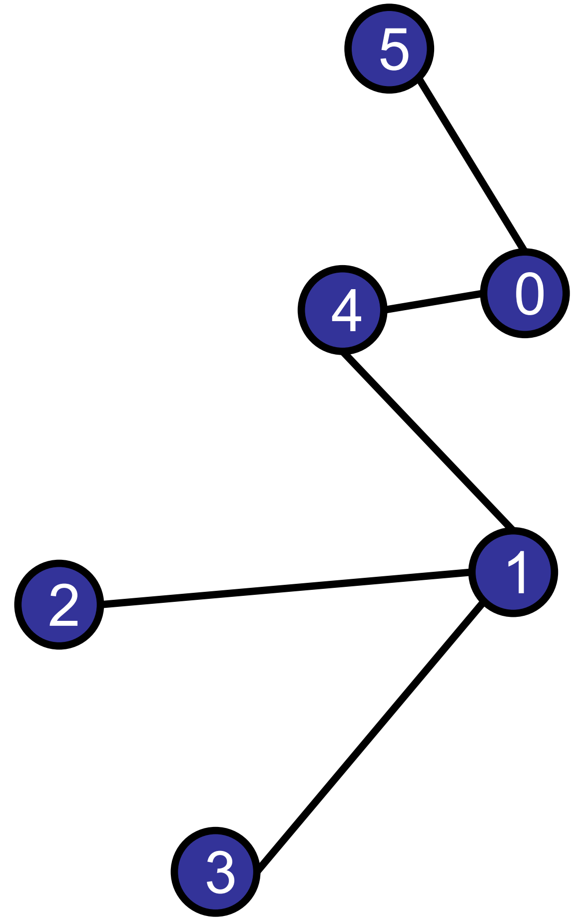


# Adjacency List

Graph consists of:

- Nodes: stored in an array
- Edges: linked list per node

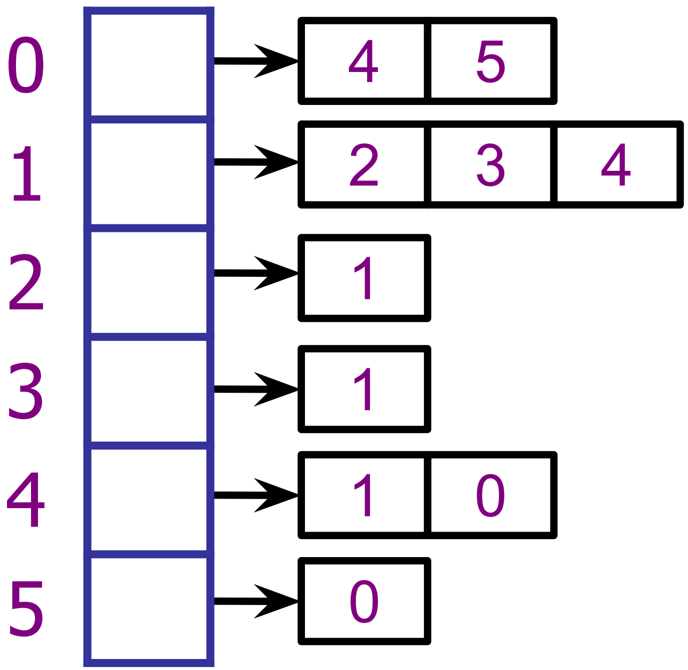


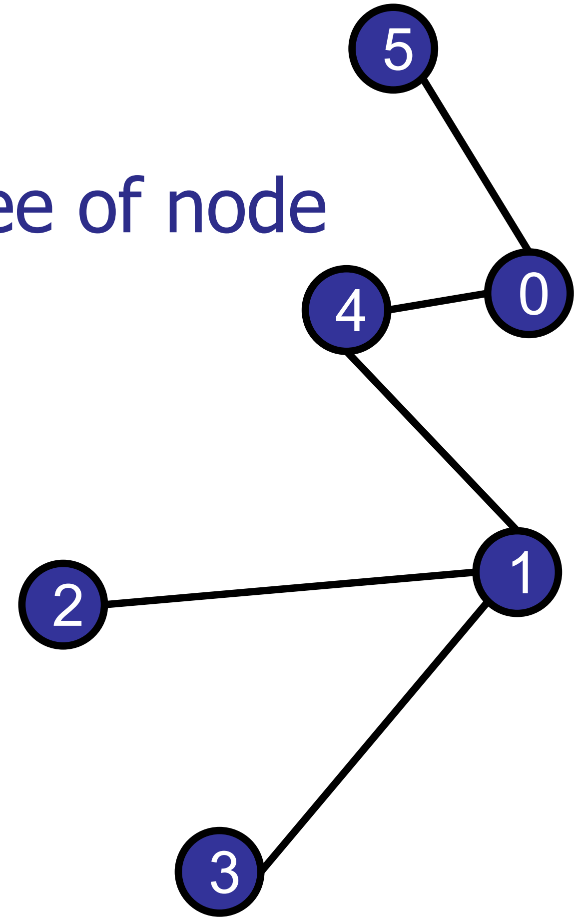



# Adjacency List

Graph consists of:

- Nodes: stored in an array
- Edges: array as large as degree of node

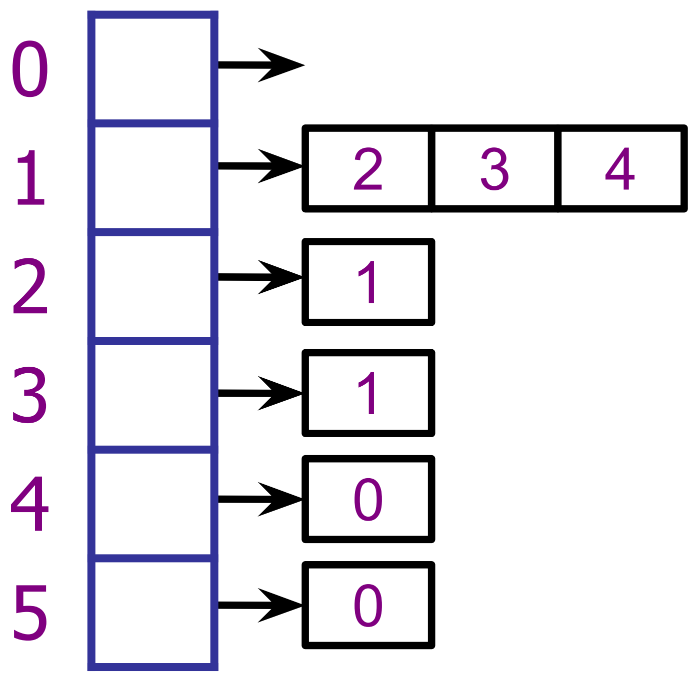


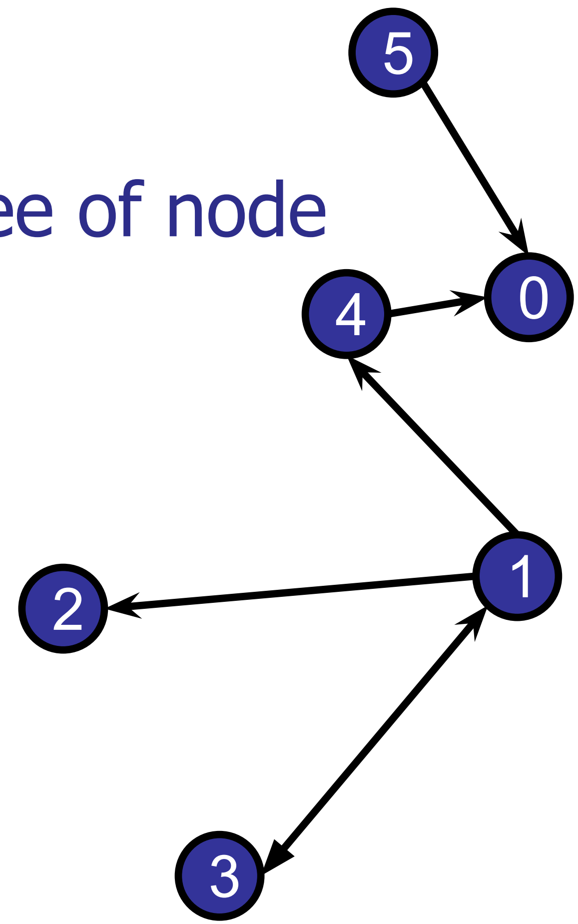



# Adjacency List

Graph consists of:

- Nodes: stored in an array
- Edges: array as large as degree of node



# Adjacency List in Java

```
class NeighborList extends LinkedList<Integer> {

}

class Node {

  NeighborList nbrs;
}

class Graph {
    Node[] nodeList;
}

```

# Adjacency List in Java

```
class NeighborList extends ArrayList<Integer> {
}

class Node {

  NeighborList nbrs;
}

class Graph {
    Node[] nodeList;
}

```

# Adjacency List in Java

```
class Graph {
    ArrayList<ArrayList<Integer>> adjacency_list;
}
```

Given some node ID, how long will it take to print out all the neighbouring IDs?

1. O(1)
2. O(deg(v))
3. O(V)

V = number of vertices/nodes

deg(v) = degree of input vertex v

Given two nodes x, y. Determining whether x and y are neighbours takes:

1. O(1)
2. O(min(deg(x), deg(y))
3. O(V)

V = number of vertices/nodes

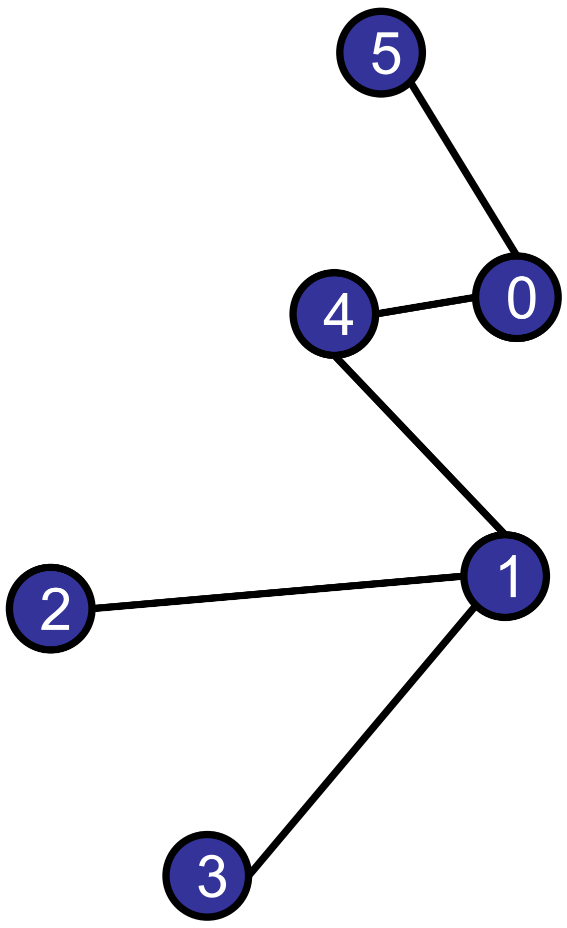
deg(v) = degree of input vertex v

# Adjacency Matrix

Graph consists of:

- Nodes
- Edges = pairs of nodes

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



# Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

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

# Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

- Undirected graphs:
  - A is symmetric!

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

# Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

Neat property:

- $A^2$ = length 2 paths

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

# Adjacency Matrix

To find out if c and d are 2-hop neighbors:

- Let $B = A^2$
- B[c][d] = (A[c][0]*A[0][d] + A[c][1]*A[1][d] + … + A[c][n - 1] * A[n-1][d])

- B[c, d] >= 1 iff A[c, x] == A[x, d] for some x.

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

# Adjacency Matrix

Graph represented as:

$A[v][w] >= 1$ iff $(v,w) \in E$

Neat properties:

- $A^2$ = length 2 paths
- $A^4$ = length 4 paths

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

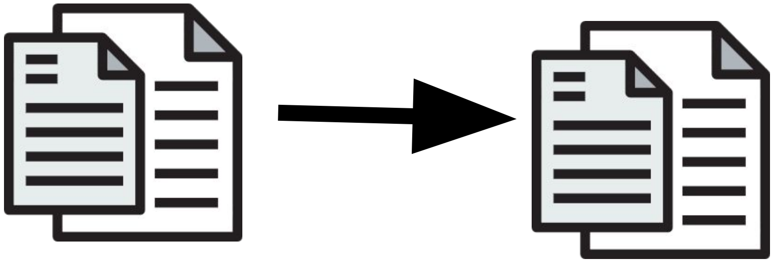
Neat way to figure out connectivity…
Neat way to figure out diameter…
Not always the most efficient…
Parallelizes well….

# Adjacency Matrix

Old Google Pagerank Idea:

Webpages are nodes.

If a webpage a links to webpage b



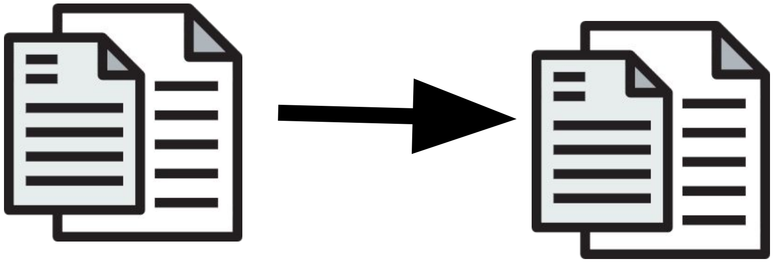
# Adjacency Matrix

Old Google Pagerank Idea:

Webpages are nodes.

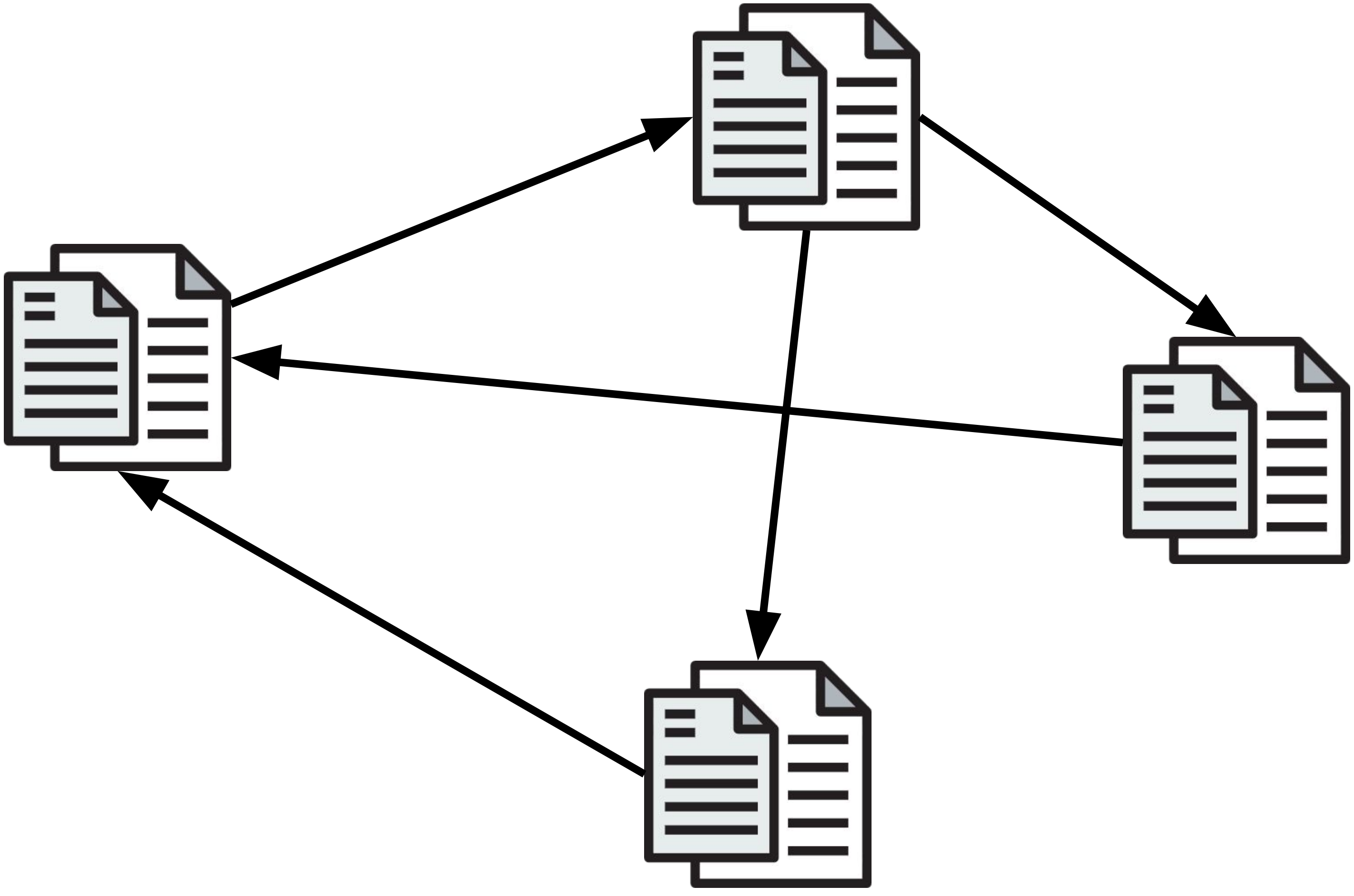
If a webpage a links to webpage b

add edge from a to b



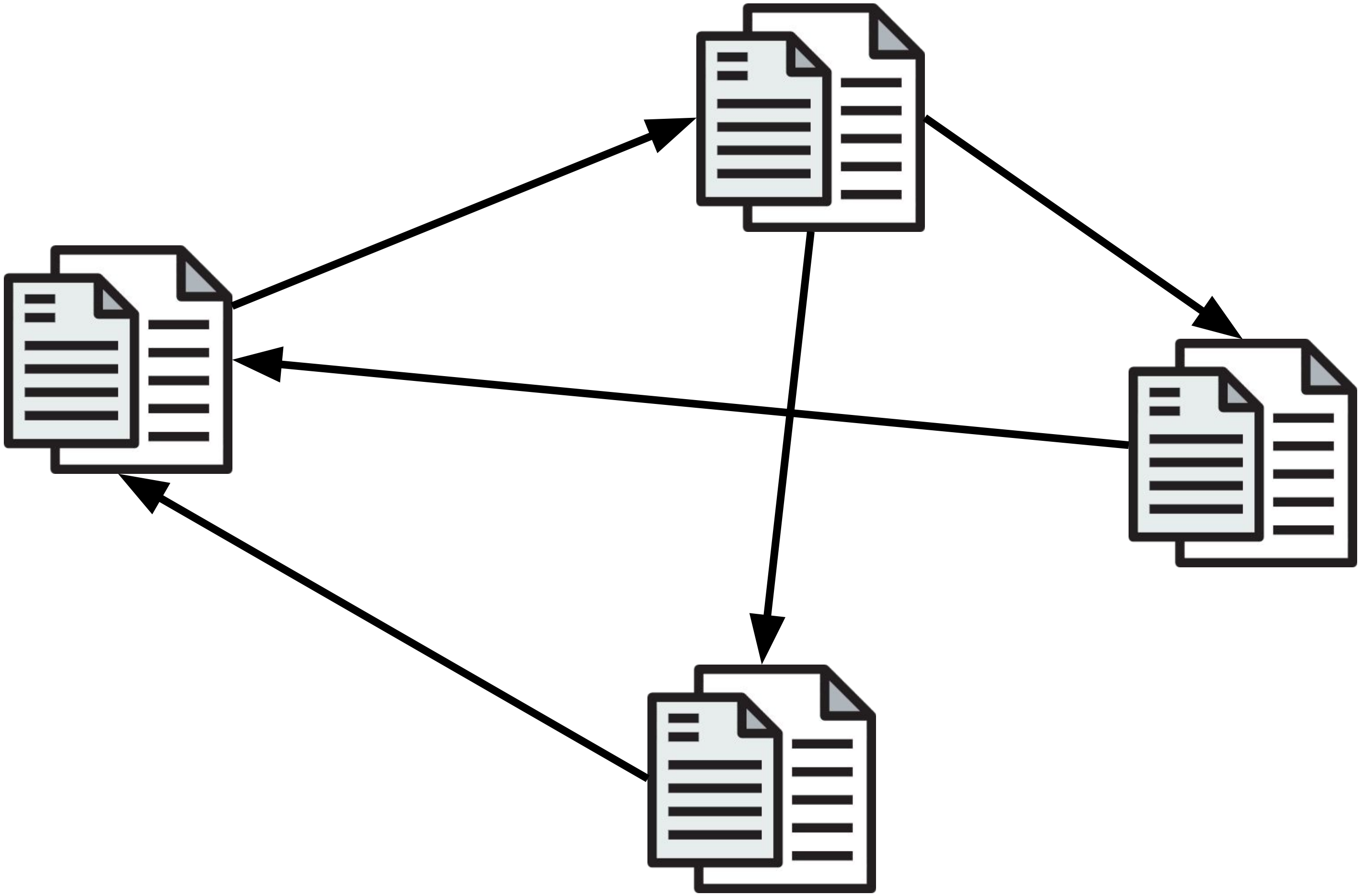
# Adjacency Matrix

So now we have a directed graph:



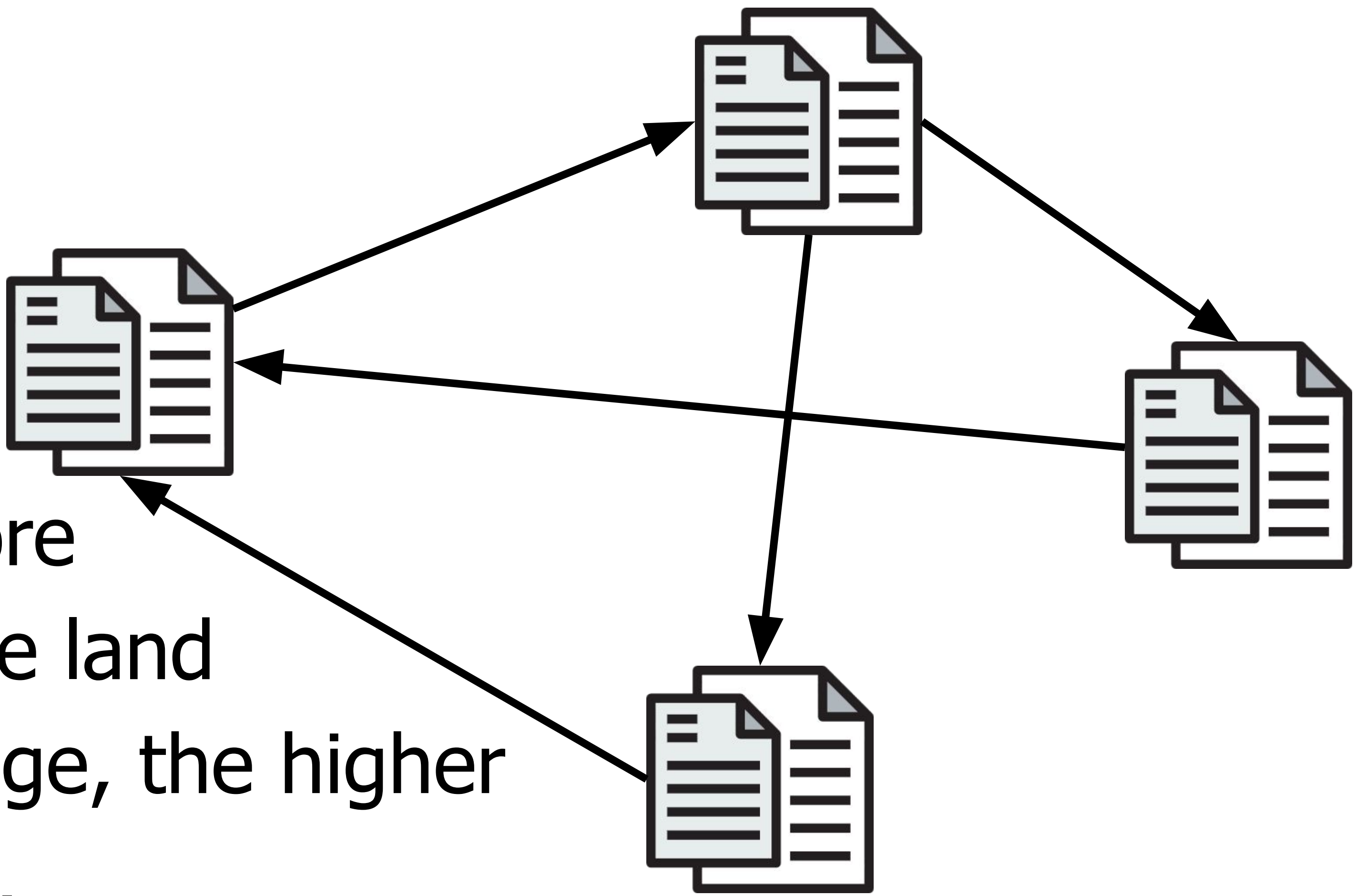
# Adjacency Matrix

Intuition: If we randomly picked a page, and did a random walk on it



# Adjacency Matrix

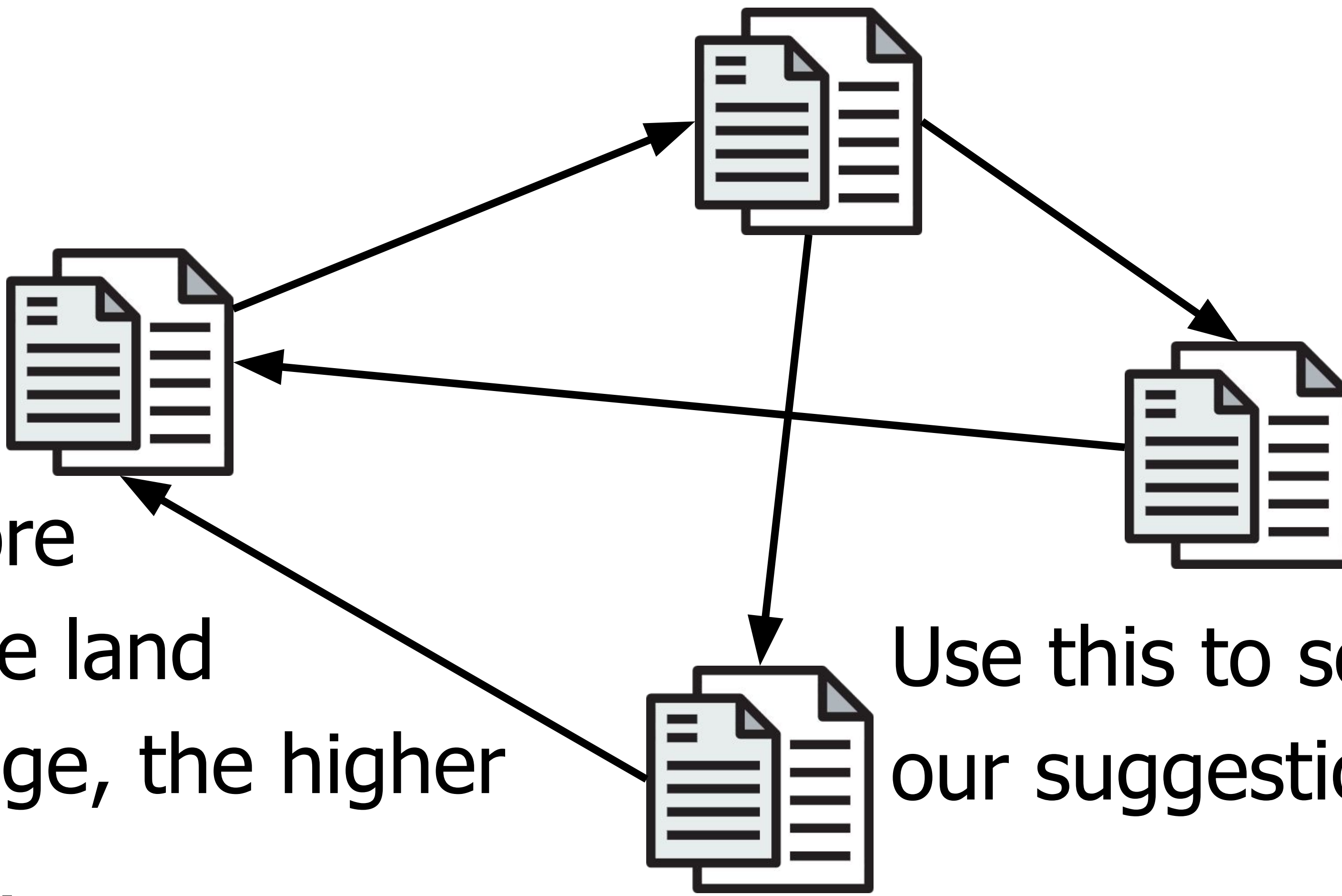
Intuition: If we randomly picked a page, and did a random walk on it



The more likely we land on a page, the higher its rank.

# Adjacency Matrix

Intuition: If we randomly picked a page, and did a random walk on it

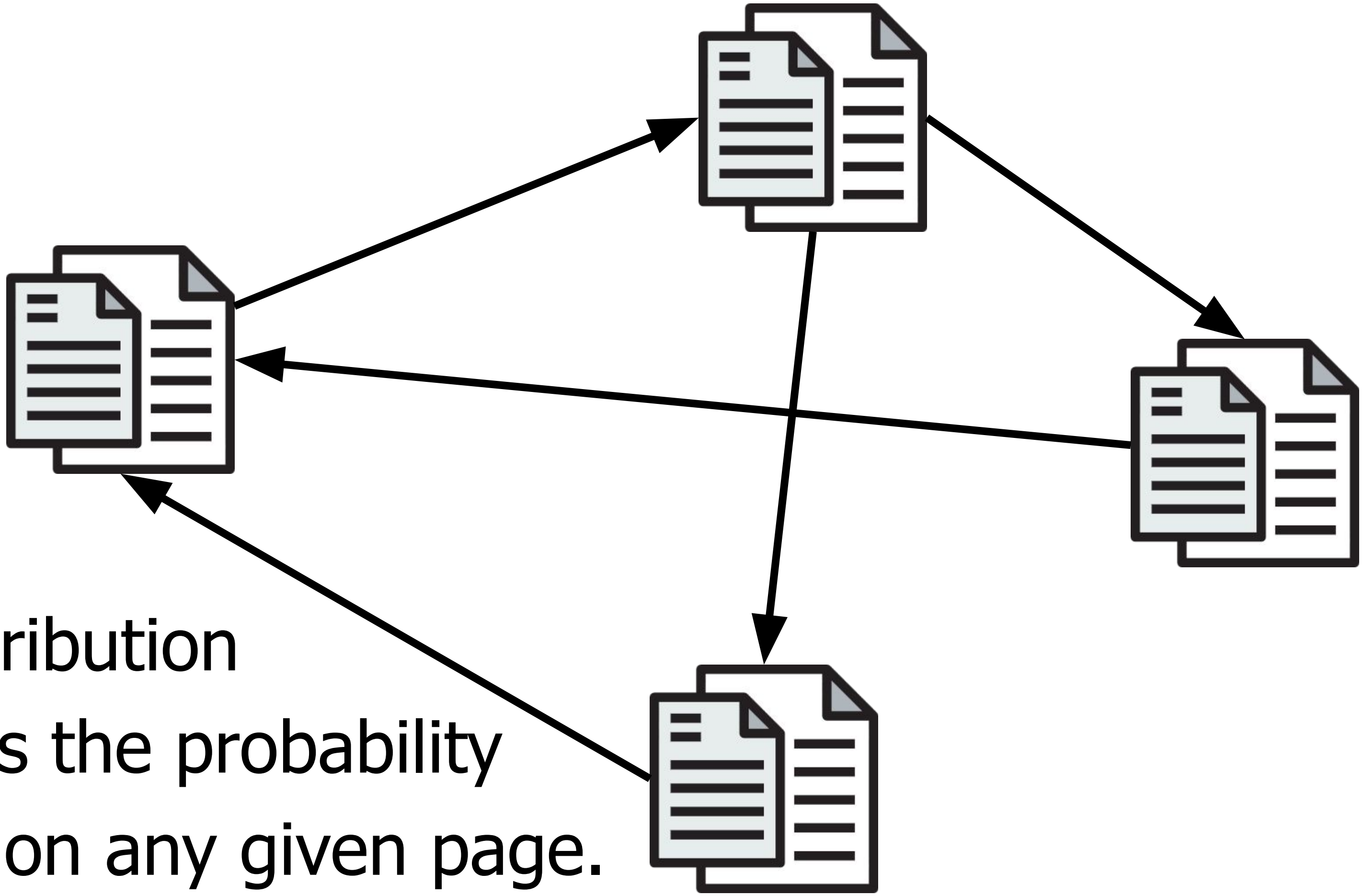


The more likely we land on a page, the higher its rank.

Use this to sort our suggestions!

# Adjacency Matrix

As the length t of our random walk approaches infinity, find the **stationary distribution** of the walk.

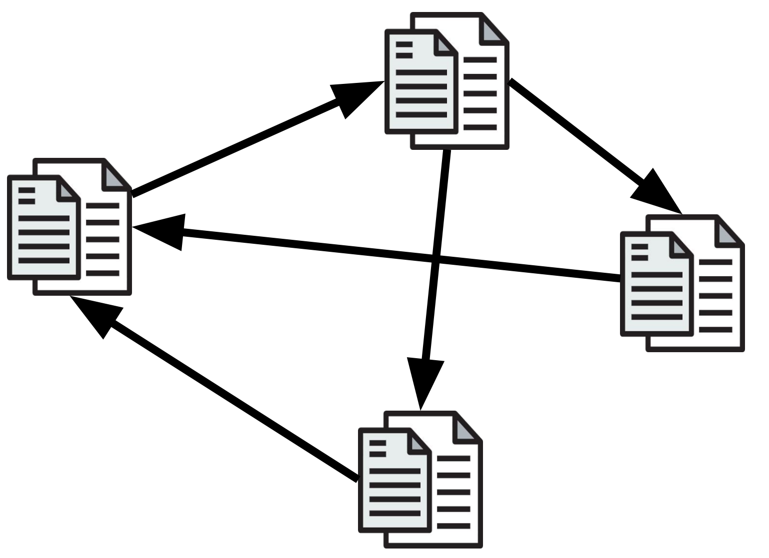


This distribution describes the probability we land on any given page.

# Adjacency Matrix

In linear algebra terms: The pagerank vector, is the vector that describes the distribution.

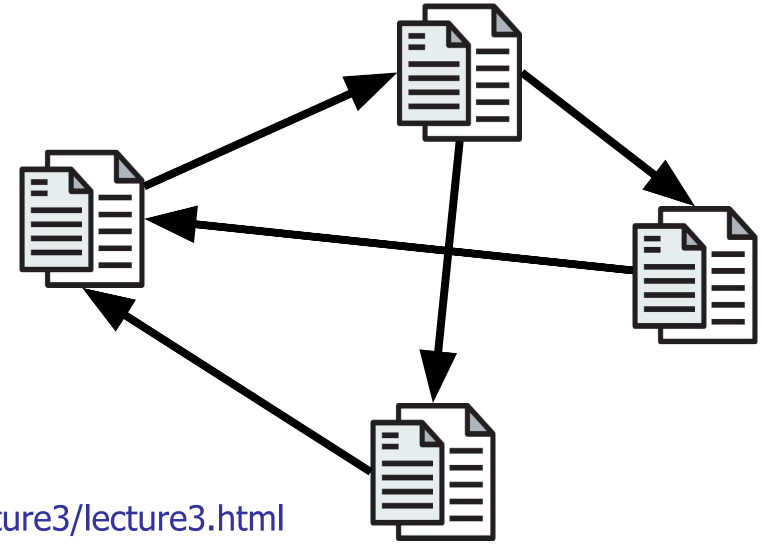
This is the eigenvector of the matrix, with eigenvalue 1



# Adjacency Matrix

In linear algebra terms: The pagerank vector, is the vector that describes the distribution.

This is the eigenvector of the matrix, with eigenvalue 1



Read: https://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html

# Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

Neat properties:

- $A^2$ = length 2 paths
- $A^4$ = length 4 paths
- $A^\infty \approx$ Google pagerank?

(Simulate random walk by replacing '1' with probability.)

$$
\begin{array}{c|cccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 \\
4 & 1 & 1 & 0 & 0 & 0 & 0 \\
5 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

# Adjacency Matrix in Java

```
class Graph {
    ArrayList<ArrayList<Integer>> adjacency_matrix;
}
```

# Trade-offs

Adjacency Matrix vs. Array?

Given some node ID, how long will it take to print out all the neighbouring IDs?

1. $O(1)$
2. $O(deg(v))$
3. $O(V)$

V = number of vertices/nodes

deg(v) = degree of input vertex v

Given two nodes x, y. Determining whether x and y are neighbours takes:

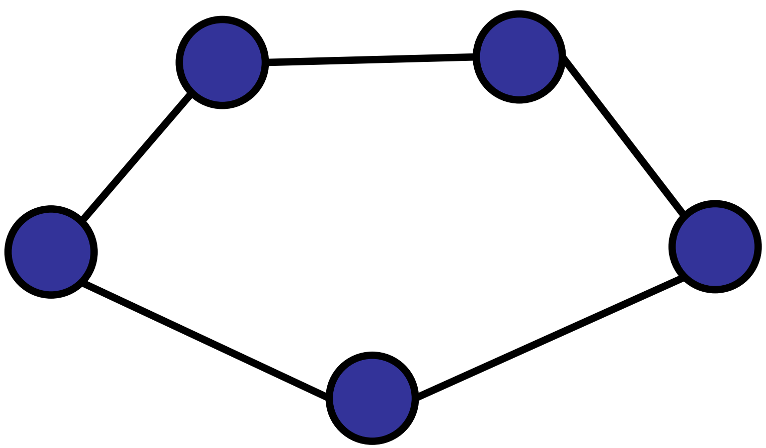
1. O(1)
2. O(min(deg(x), deg(y))
3. O(V)

V = number of vertices/nodes

deg(v) = degree of input vertex v

For a cycle, which representation uses less space?

1. Adjacency list
2. Adjacency matrix
3. Equivalent

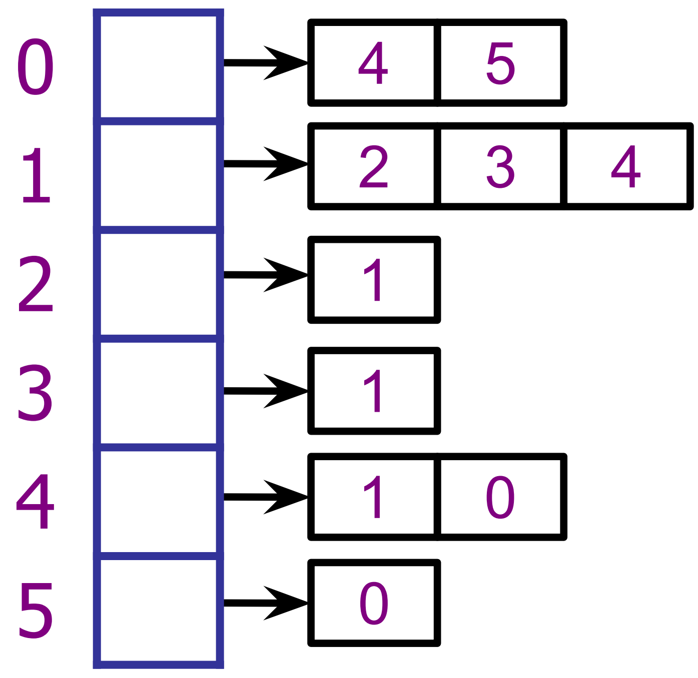


# Adjacency List

Memory usage for graph $G = (V, E)$:

- array of size $|\mathbf{V}|$
- array lists of size $\mathbf{deg(v)}$

Total: $O(|V| + |E|)$



# Adjacency Matrix

Memory usage for graph $G = (V, E)$:

- matrix of size $|\mathbf{V}|*|\mathbf{V}|$

Total: $O(|V|^2)$

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

For a complete graph, which representation uses asymptotically less space?

1. Adjacency matrix
2. Adjacency list
3. Equivalent

# Adjacency List vs. Matrix

Memory usage for graph $G = (V, E)$:

- array of size $|\mathbf{V}|$
- array lists of size $\mathbf{deg(v)}$

Total: $O(|V| + |E|)$

Since for a complete graph: $|E| = \Theta(|V|^2)$

Total: $O(|V|^2)$

# Adjacency List vs. Matrix

If space usage is at a premium and |E| is much smaller than $|V|^2$, consider using an adjacency list.

# Adjacency List vs. Matrix

If space usage is at a premium and $|E|$ is much smaller than $|V|^2$, consider using an adjacency list.

If we need to compute something like PageRank, adjacency matrix is the way to go.

# Adjacency List vs. Matrix

If space usage is at a premium and $|E|$ is much smaller than $|V|^2$, consider using an adjacency list.

If we need to compute something like PageRank, adjacency matrix is the way to go.

Other choices will depend on what graph algorithm we use.

# Trade-offs

Adjacency Matrix:

- Fast query: are v and w neighbors?
- Slow query: find me any neighbor of v.
- Slow query: enumerate all neighbors.

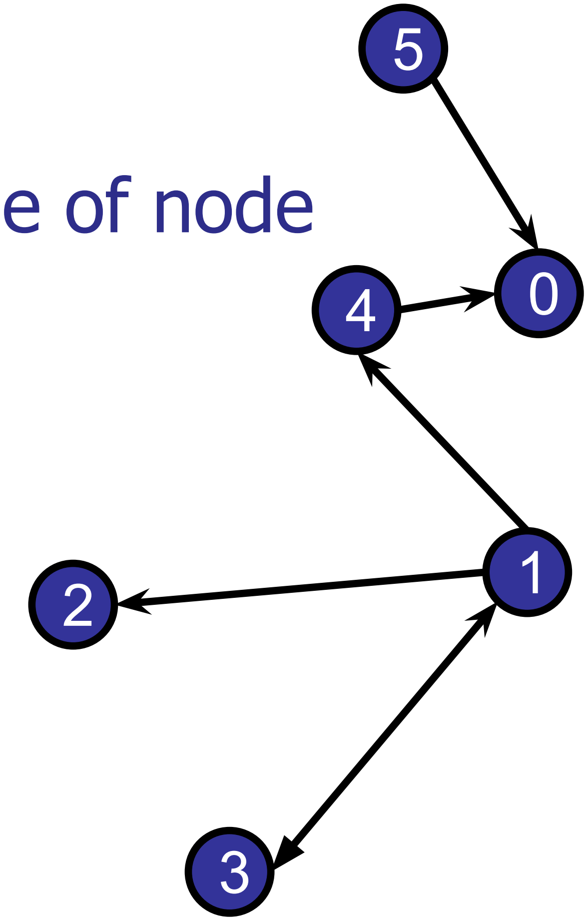
Adjacency List:

- Fast query: find me any neighbor.
- Fast query: enumerate all neighbors.
- Slower query: are v and w neighbors?

# Edge List

Graph consists of:

- Nodes: stored in an array
- Edges: array as long as degree of node




| | | | | | |
|---|---|---|---|---|---|
| (5, 0) | (4, 0) | (1, 4) | (1, 2) | (1, 3) | (3, 1) |

Given some node ID, how long will it take to print out all the neighbouring IDs?

1. O(1)
2. O(deg(v))
3. O(V)
4. O(E)

V = number of vertices/nodes

E = number of edges

deg(v) = degree of input vertex v

Given two nodes x, y. Determining whether x and y are neighbours takes:

1. O(1)
2. O(min(deg(x), deg(y))
3. O(V)
4. O(E)

V = number of vertices/nodes

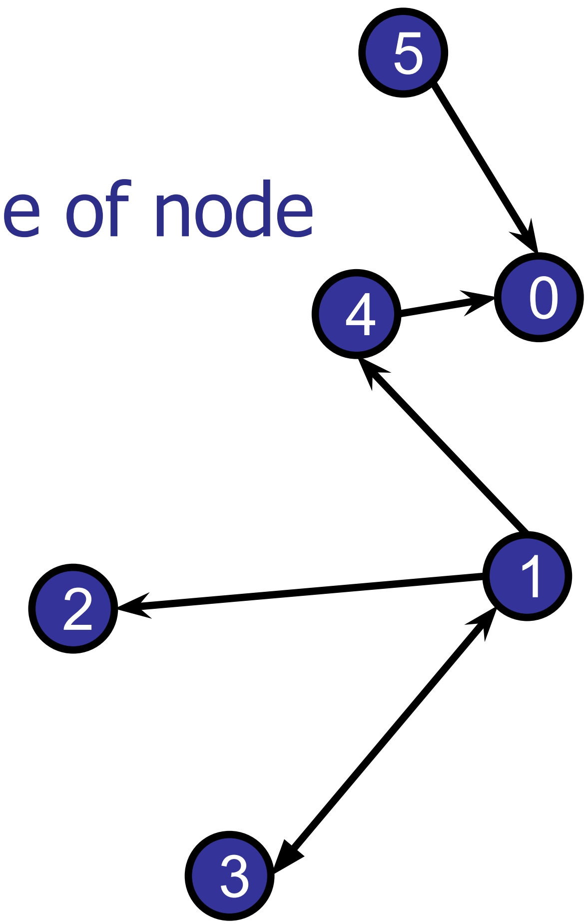
E = number of edges

deg(v) = degree of input vertex v

# Edge List

Graph consists of:

- Nodes: stored in an array
- Edges: array as long as degree of node




Space usage:

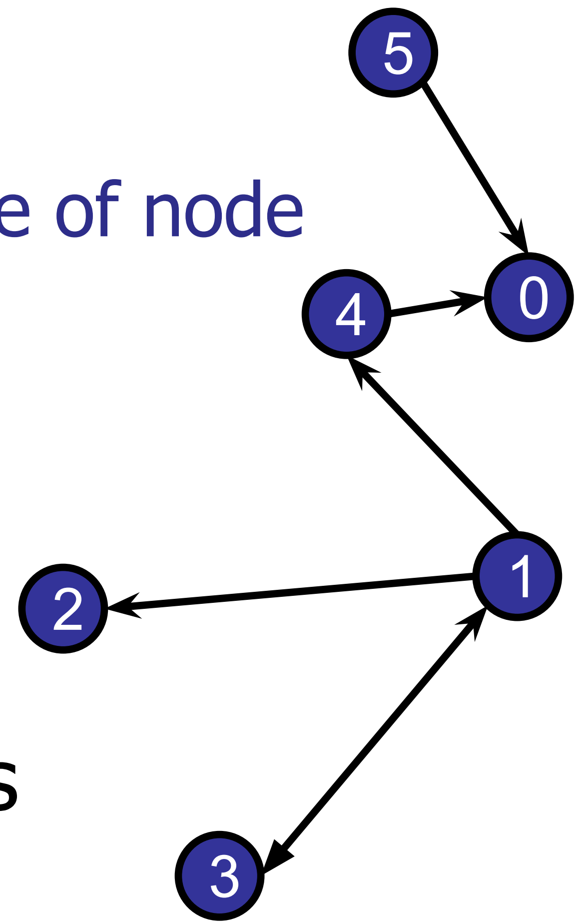
O(|E|)

| | | | | | |
|---|---|---|---|---|---|
| (5, 0) | (4, 0) | (1, 4) | (1, 2) | (1, 3) | (3, 1) |

# Edge List

Graph consists of:

- Nodes: stored in an array
- Edges: array as long as degree of node




Space usage:

O(|E|)

Will later see an algorithm where this is best representation.

| | | | | | |
|---|---|---|---|---|---|
| (5, 0) | (4, 0) | (1, 4) | (1, 2) | (1, 3) | (3, 1) |

# Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

# BFS & DFS

Simple, but very, very important:

Recent* conversation with CS2109S instructor (Introduction to AI and Machine Learning):

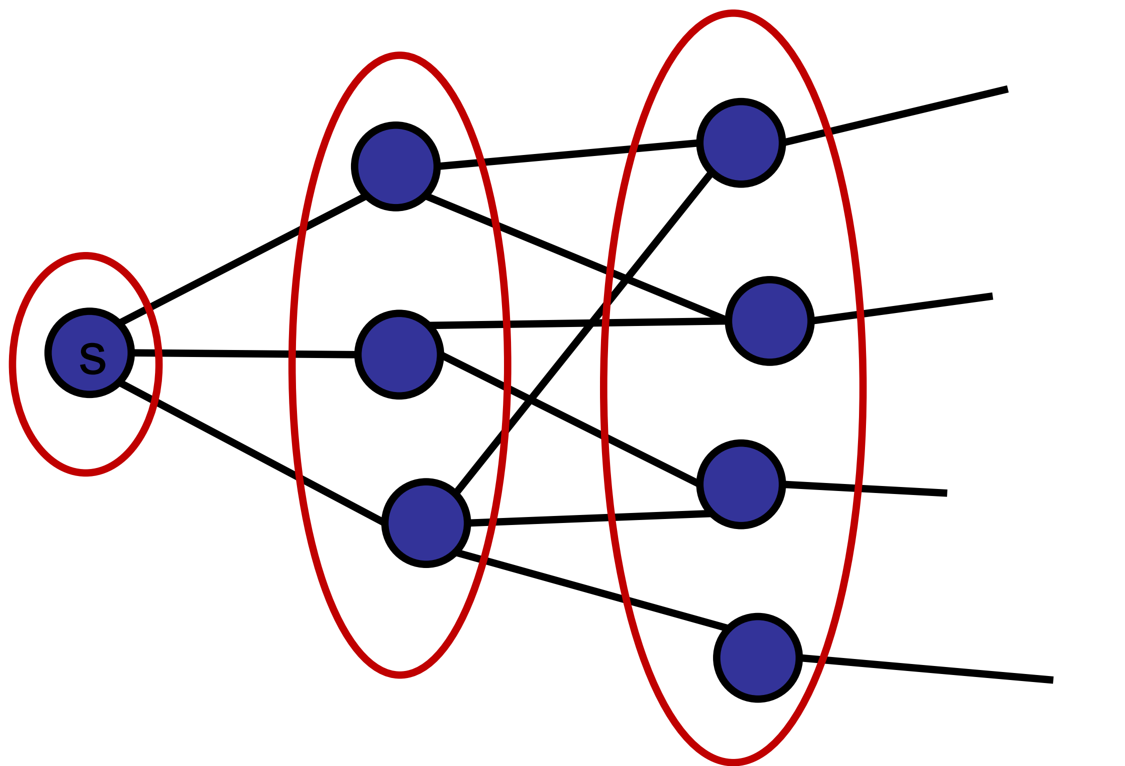
"It is really, really important that every student who take CS2109S can easily code up a simple BFS or DFS. Critical foundation before we can do anything more interesting."

*At the time these slides were made

# Searching a graph

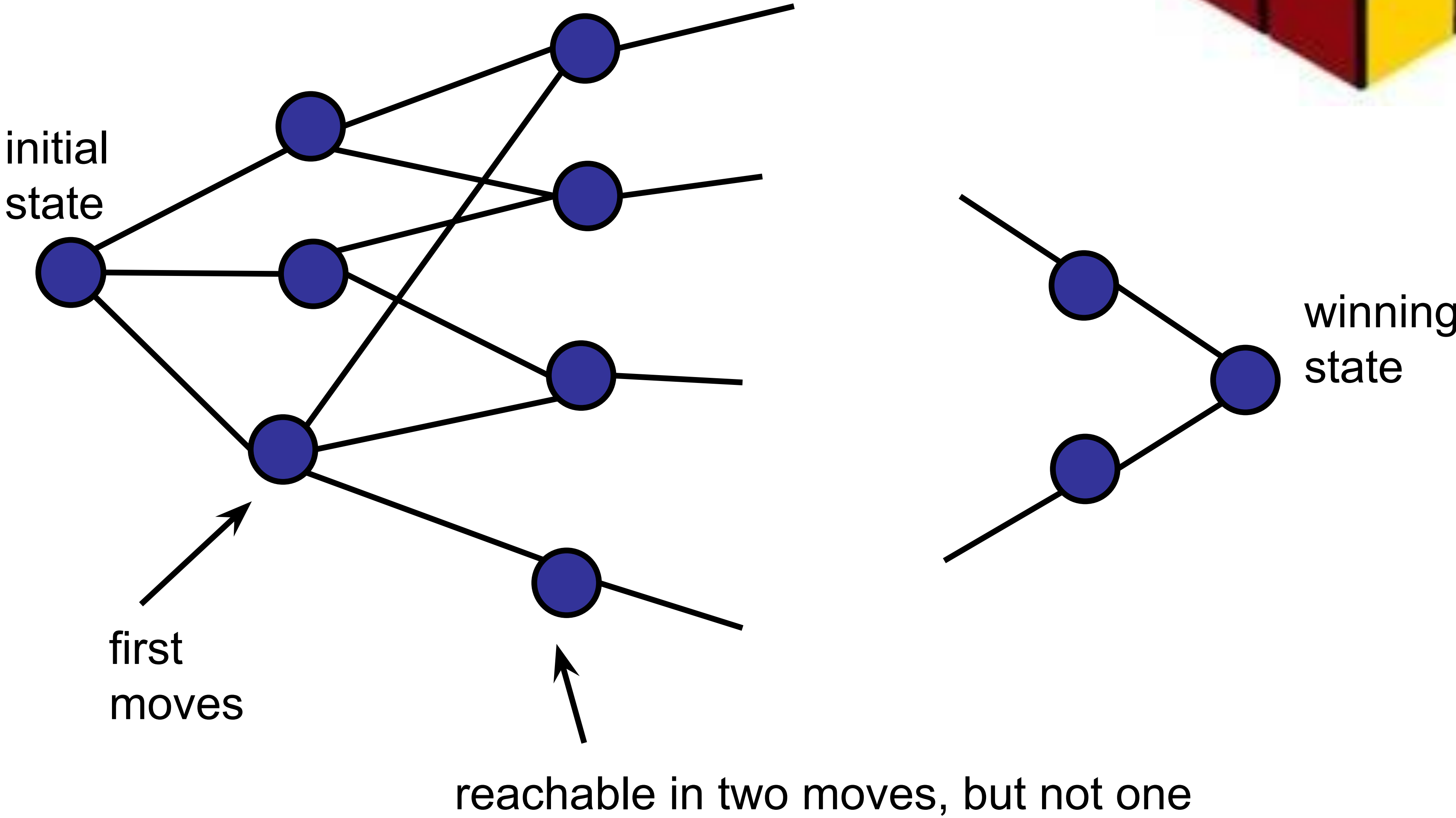
Breadth-First Search:

Explore level by level



# 2 x 2 x 2 Rubik's Cube

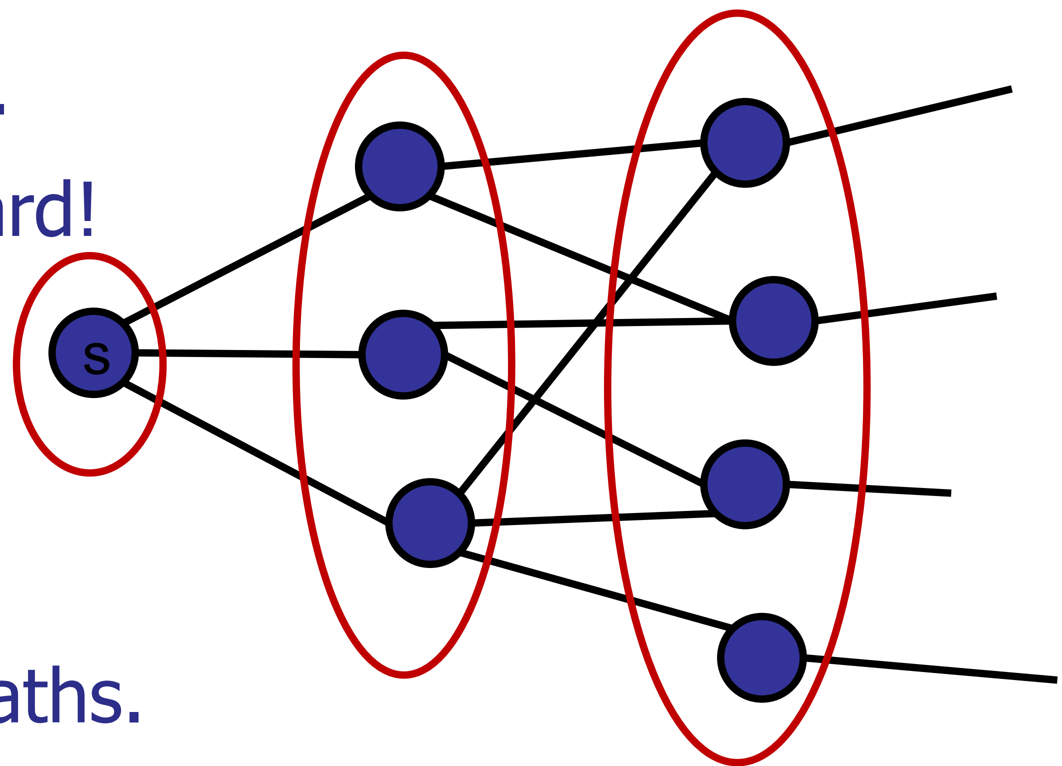
Geography of Rubik's configurations:



# Searching a graph

Breadth-First Search:

- Explore level by level
- Frontier: current level
- Initially: {s}
- Advance frontier.
- Don't go backward!

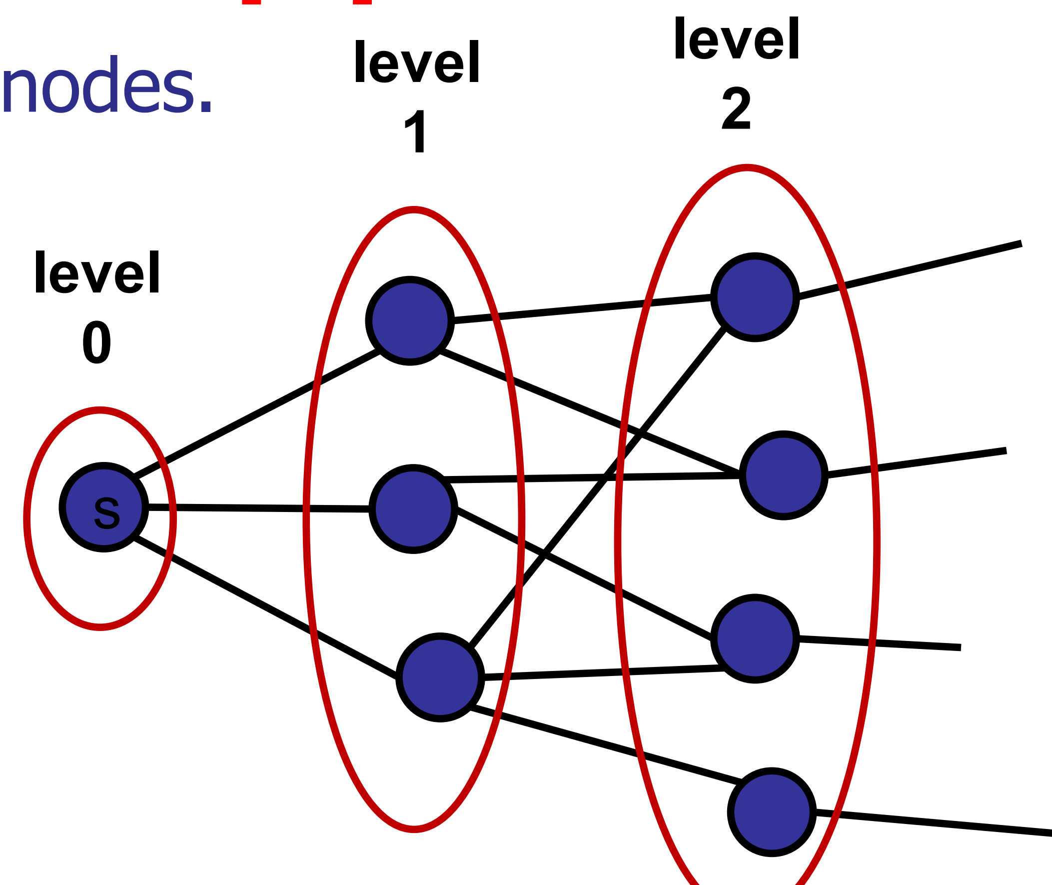



- Finds shortest paths.

# Searching a graph

Breadth-First Search:

- Build levels.
- Calculate level[i] from level[i-1]
- Skip already visited nodes.



# Searching a graph

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
   a. Take next node out of queue.
   b. <u>Go through all neighbours of node</u>
   c. If they have already been visited, skip.
   d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

# Searching a graph

First in first out

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
   a. Take next node out of queue.
   b. <u>Go through all neighbours of node</u>
   c. If they have already been visited, skip.
   d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

# Searching a graph

Operation on our graph data structure

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
   a. Take next node out of queue.
   b. Go through all neighbours of node
   c. If they have already been visited, skip.
   d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

## Which data structure should we use to go through all neighbours?

1. Adjacency list
2. Adjacency matrix
3. Edge list
4. Node list

# Searching a graph

Checking whether a node has been visited.

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
   a. Take next node out of queue.
   b. <u>Go through all neighbours of node</u>
   c. If they have already been visited, skip.
   d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

Which data structure should we use to check for being visited?

1. ArrayList
2. Hashtable
3. Balanced BST
4. Priority Queue
5. Stack
6. Queue

# Searching a graph

Recall:

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

# Searching a graph

Recall:

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | true |

# Searching a graph

Recall:

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | true |

node 0 has been visited

# Searching a graph

Recall:

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | true |

node 1 has been visited

# Searching a graph

Recall:

**Assume**:

For **n** nodes, the node IDs are in the range [0, n-1]

| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | true |

node 2 has been visited

# Searching a graph

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
   a. Take next node out of queue.
   b. Go through all neighbours of node
   c. If they have already been visited, skip.
   d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

# Searching a graph

```
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}

```

initial state

# Searching a graph

```
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}
```

next node to consider

# Searching a graph

```
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}
```

go through all neighbours

# Searching a graph

```
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}
```

skip if visited

# Searching a graph

```java
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}
```

otherwise mark as visited, enqueue neighbour node

# Searching a graph

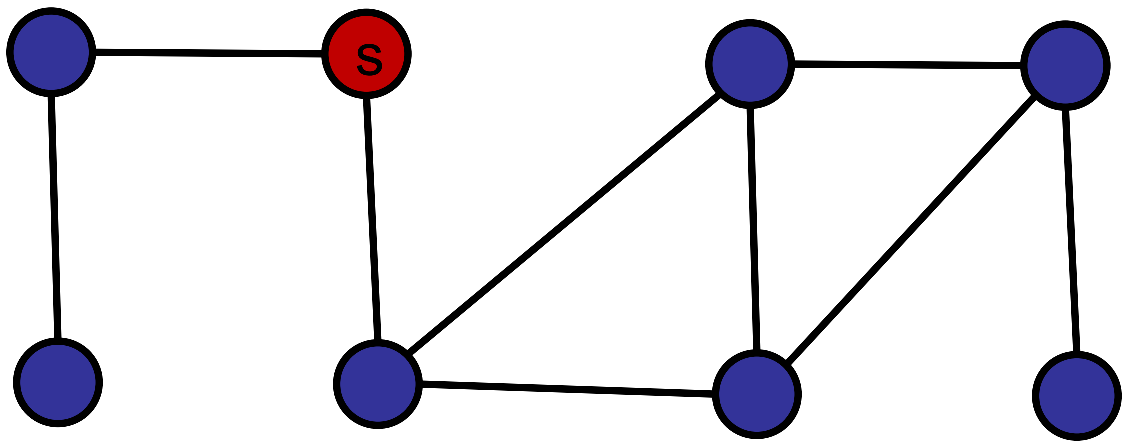
```java
void bfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Queue<Integer> queue = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  queue.add(src);
  visited[src] = true;
  while(!queue.isEmpty()){
    int current_node = queue.poll();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.offer(neighbour_node);
    }
  }
  return visited[dst];
}
```

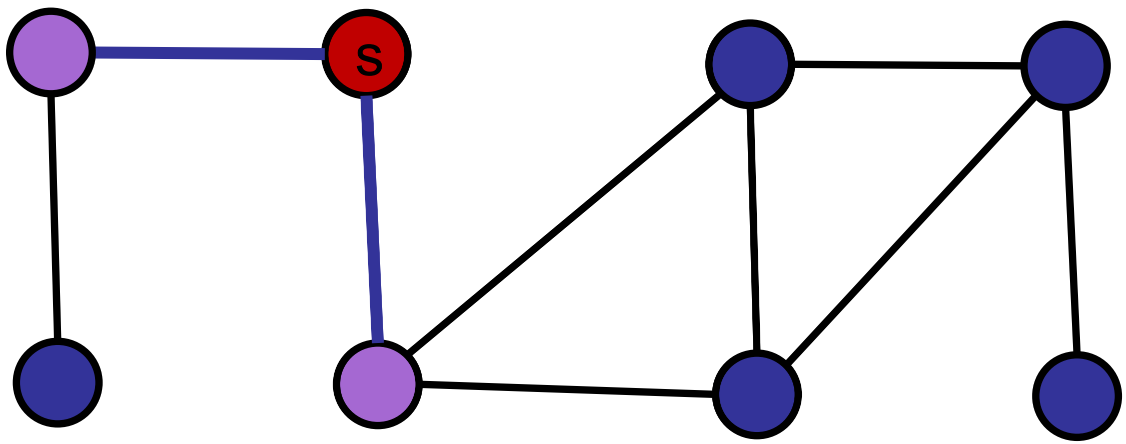
check whether dst was visited

# Breadth-First Search Example




Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# Breadth-First Search Example



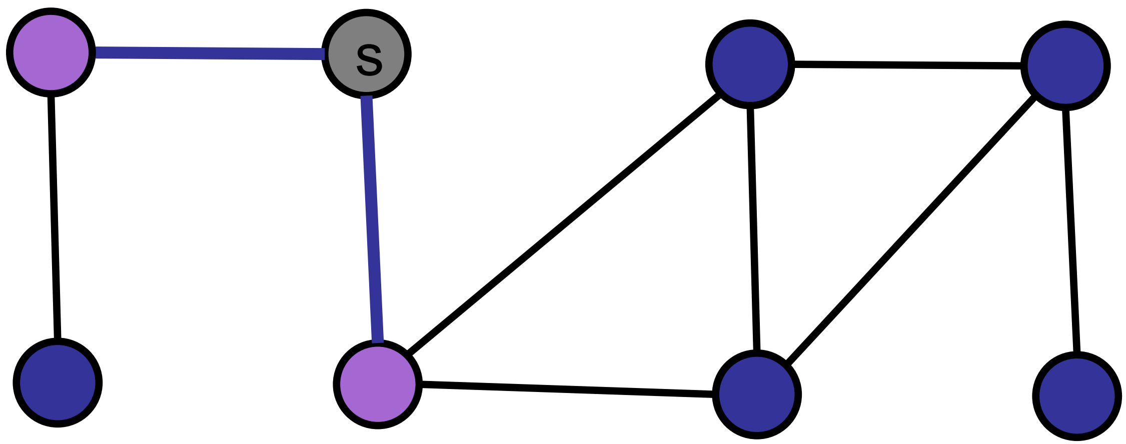

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



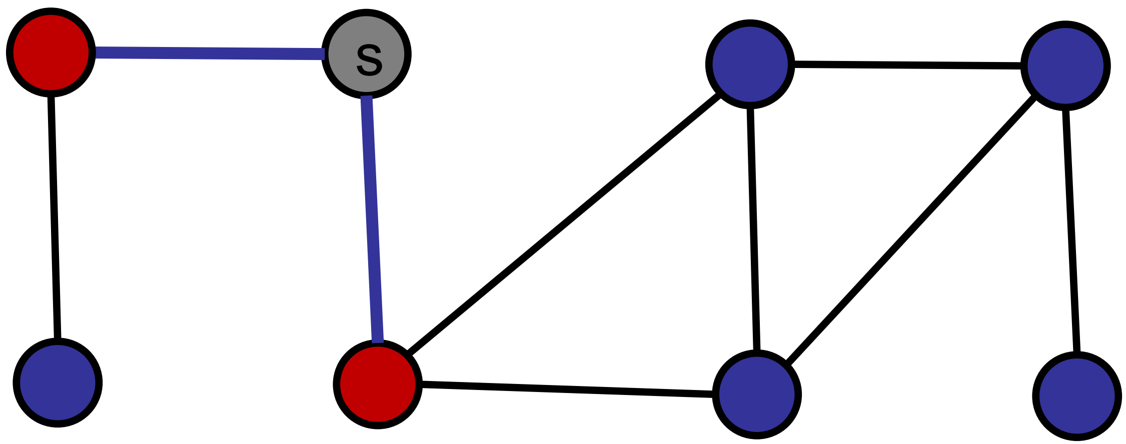

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



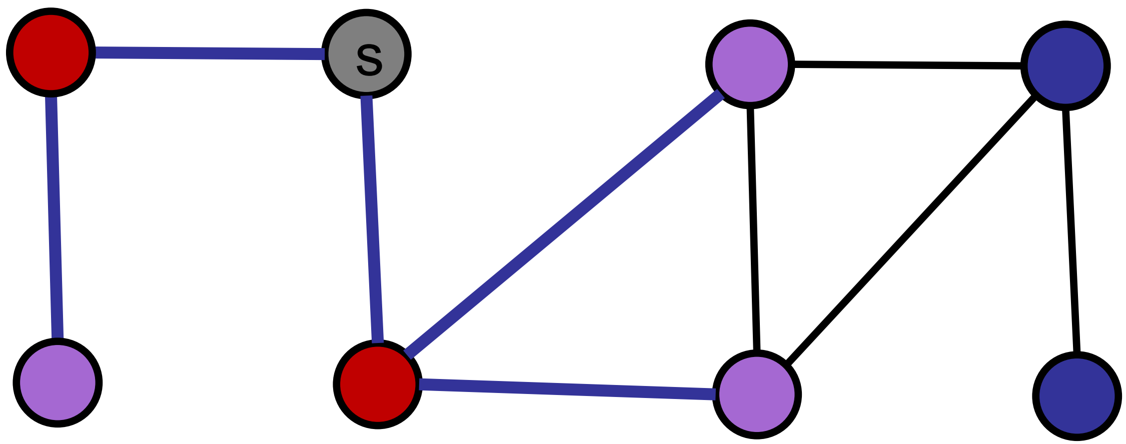

Red = active frontier

Purple = next

Gray = visited

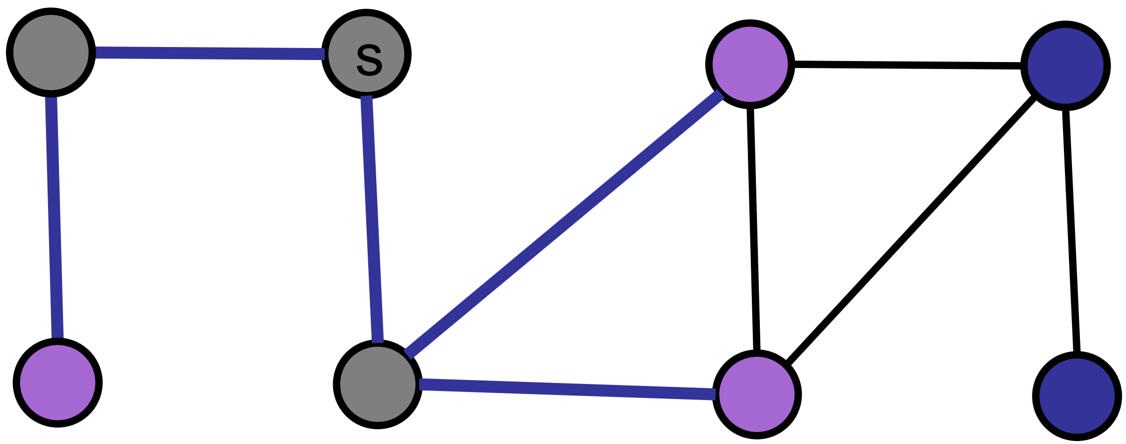
Blue = unvisited

# Breadth-First Search Example




Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# Breadth-First Search Example



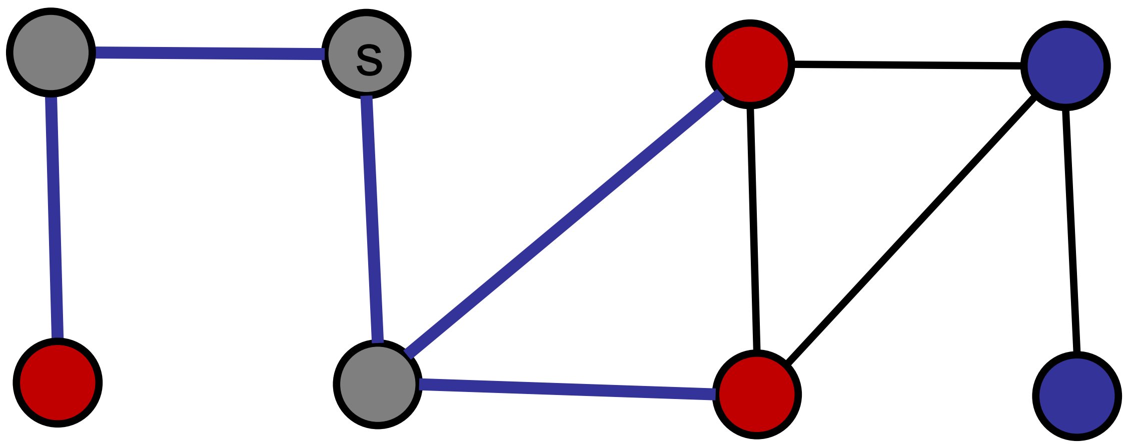

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



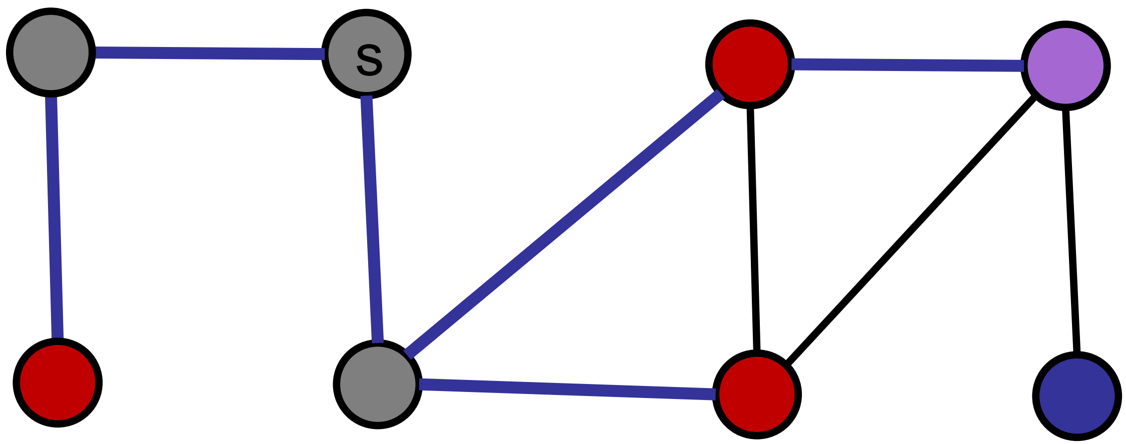

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



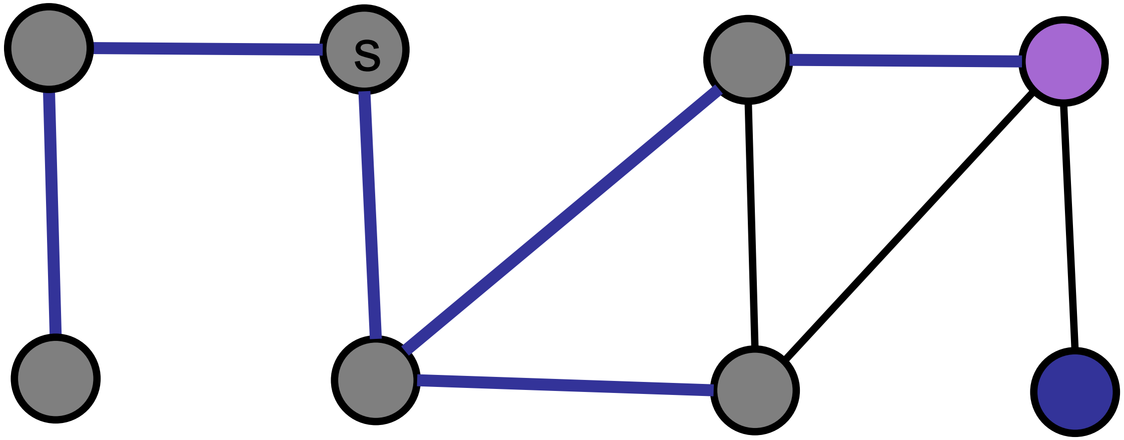

Red = active frontier

Purple = next

Gray = visited

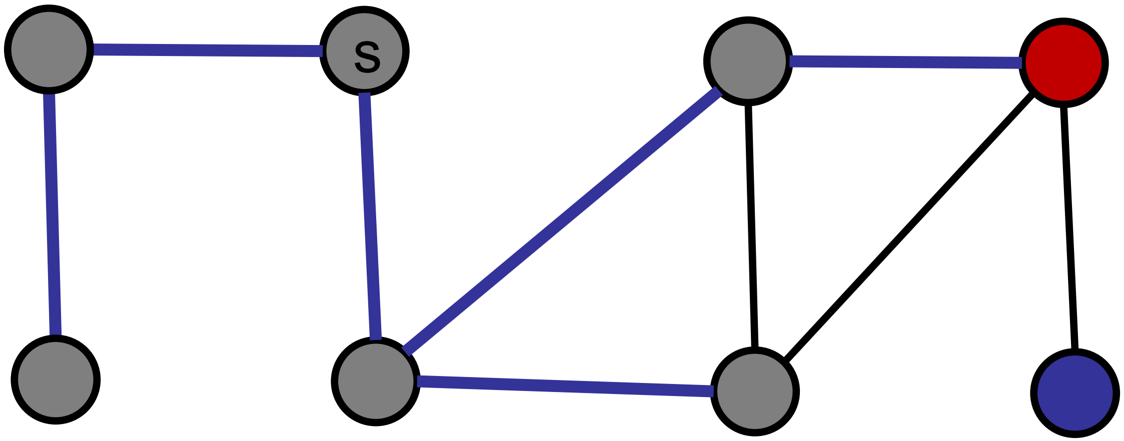
Blue = unvisited

# Breadth-First Search Example



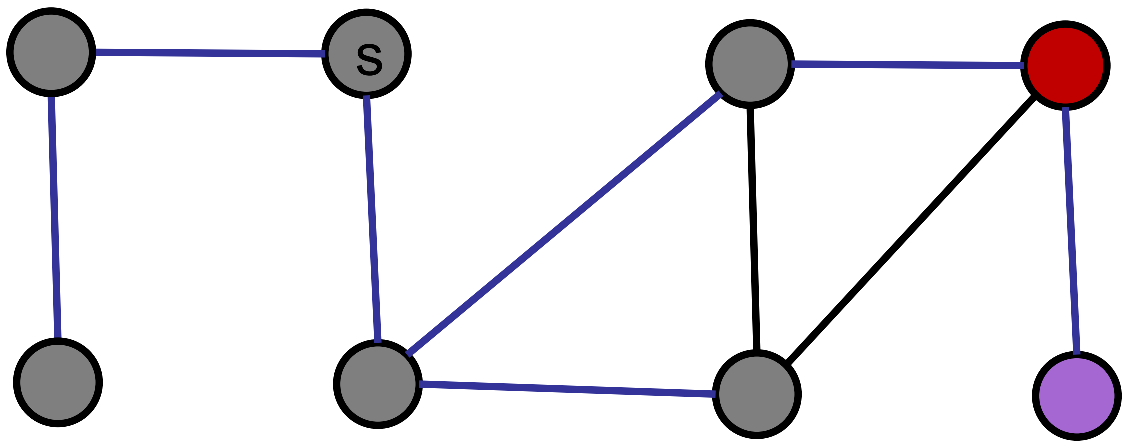

Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# Breadth-First Search Example




Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# Breadth-First Search Example



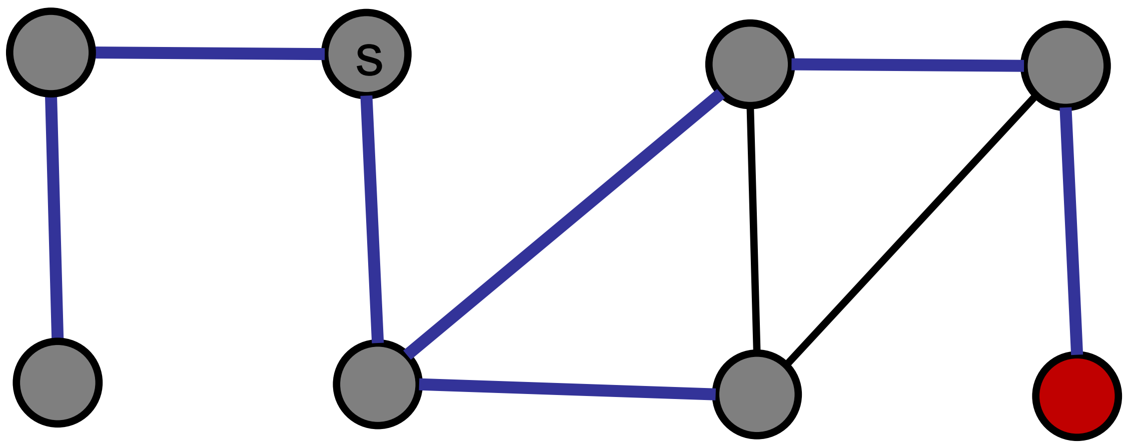

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



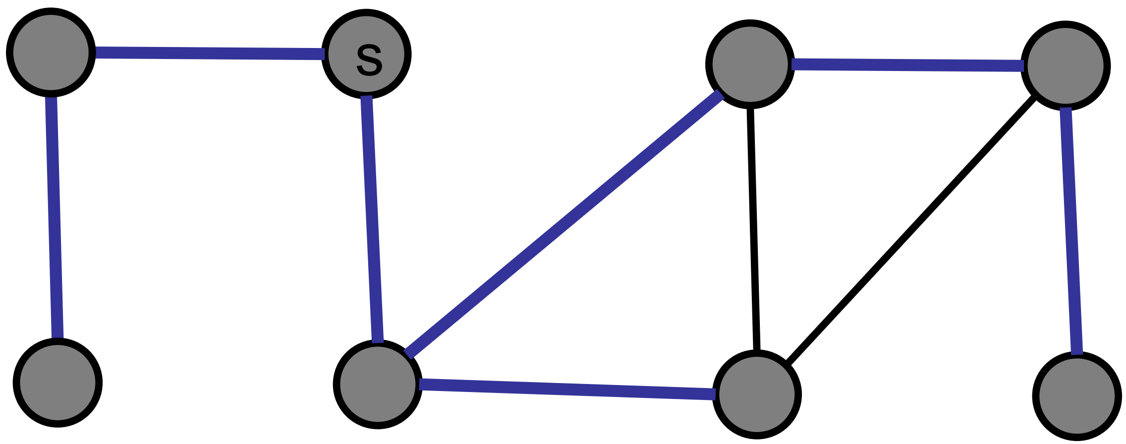

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example



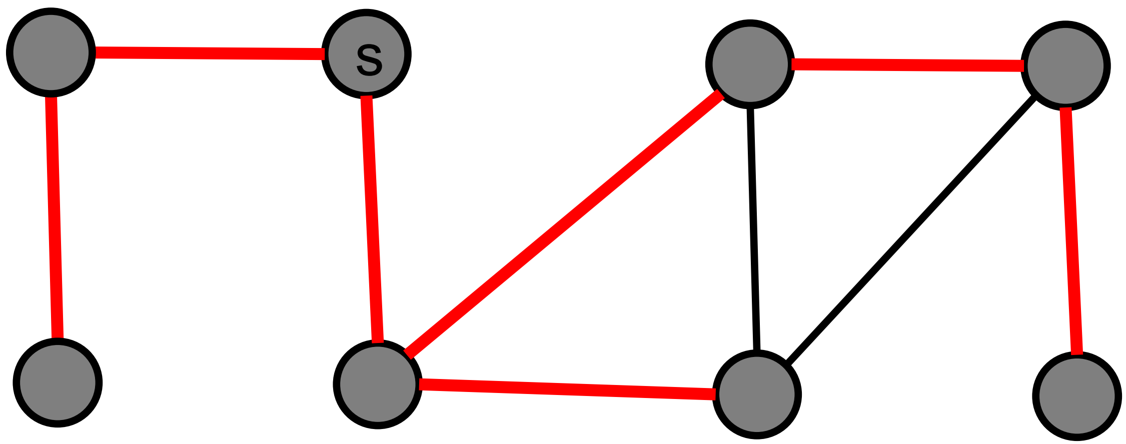

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Breadth-First Search Example




Red = active frontier

Purple = next

Gray = visited

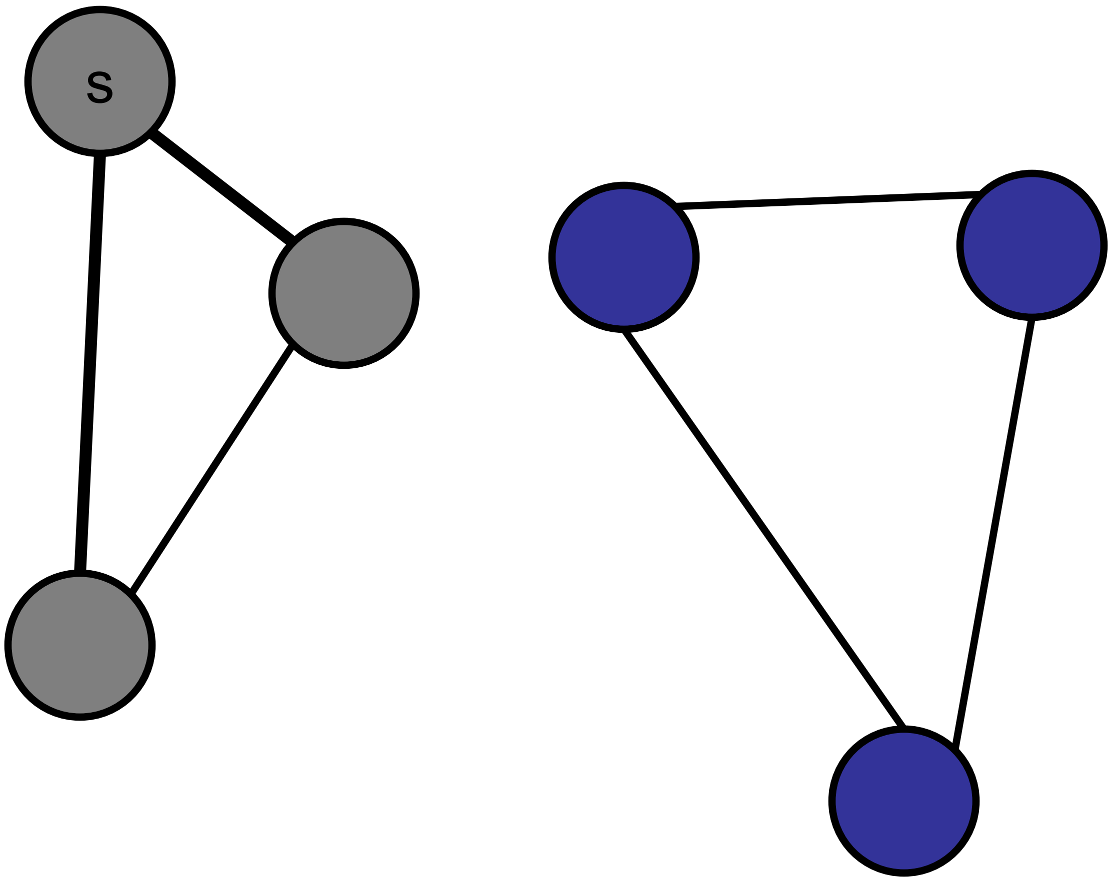
Blue = unvisited

# When does BFS fail to visit every node?

1. In a clique.
2. In a cycle.
3. In a graph with two components.
4. In a sparse graph.
5. In a dense graph.
6. Never.

# BFS on Disconnected Graph

Example:



The running time of BFS (using adjacency list) is:

1. O(V)
2. O(E)
3. O(V+E)
4. O(VE)
5. $(V^2)$
6. I have no idea.

# Searching a graph

Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
    a. Take next node out of queue.
    b. <u>Go through all neighbours of node</u>
    c. If they have already been visited, skip.
    d. Otherwise, mark them as visited, enqueue them as well.

When queue is empty, check whether destination node was marked as visited.

# Breadth-First Search

Analysis:

- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most twice.

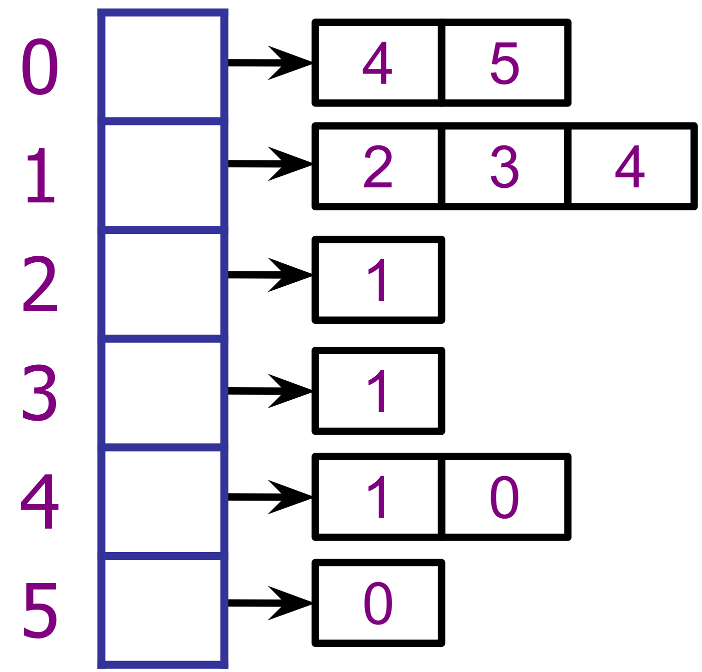
# Breadth-First Search

Analysis:

- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most twice.

Easy way to see this:

When a node is first visited,
we iterate through its neighbour list.



# Breadth-First Search

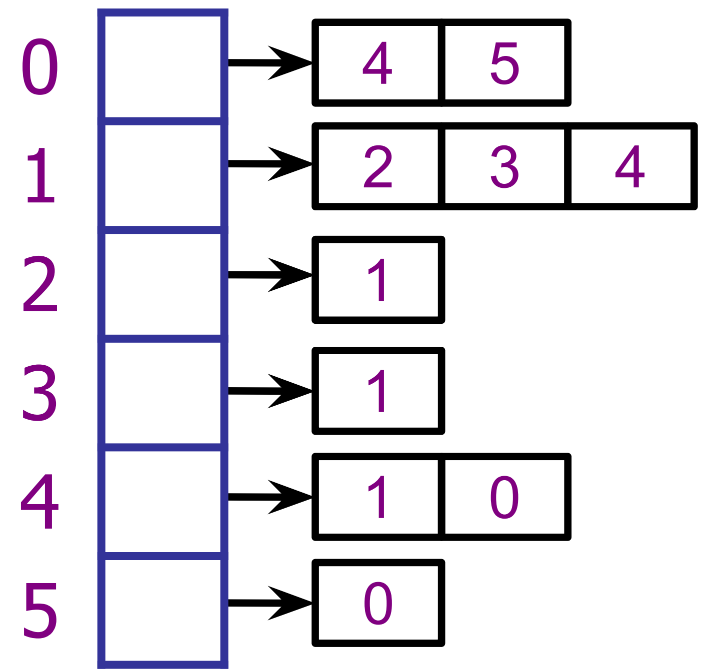
Analysis:

- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most twice.

Easy way to see this:

When a node is first visited,
we iterate through its neighbour list.

cost: O(deg(v))



# Breadth-First Search

Analysis:

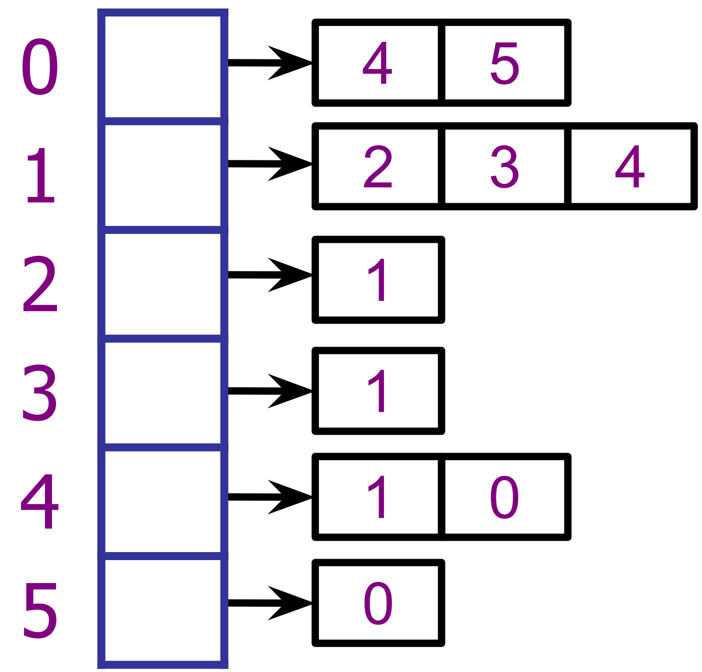
- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most twice.

Easy way to see this:

When a node is first visited,
we iterate through its neighbour list.

cost: O(deg(v))

We only do this once.



# Breadth-First Search

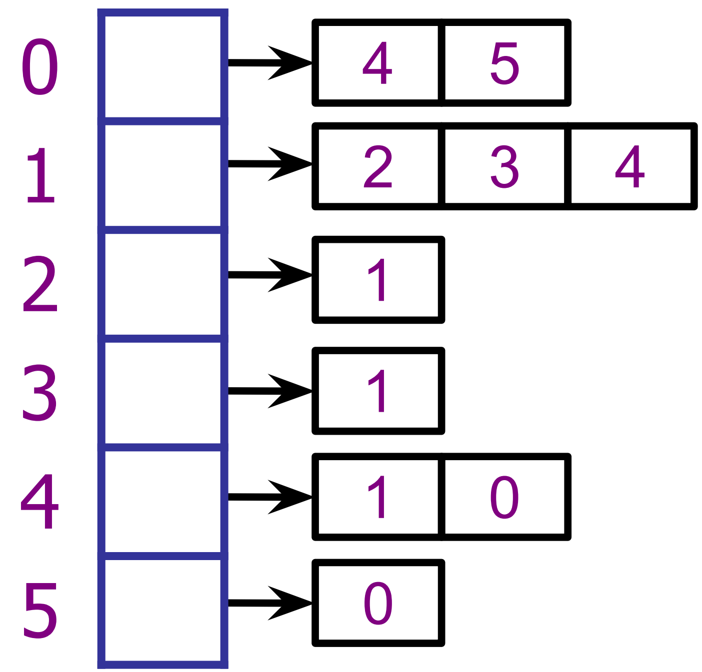
correction from lecture:
every edge is visited at
most once

## Analysis:

- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most once.

Total cost:

deg(0) + deg(1) + … + deg(n - 1)



The sum of all node degrees in any graph is:

1. O(V)
2. O(E)
3. O(VE)
4. $(V^2)$
5. I have no idea.

Handshake lemma

# Breadth-First Search

addition from lecture:

Analysis:

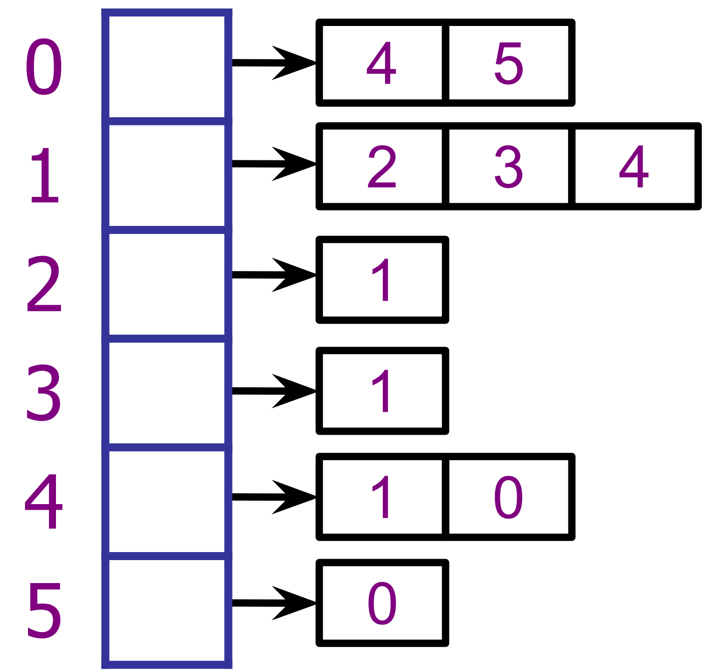
- Claim: Every node is enqueued and dequeued once.
- Claim: Every edge is visited at most once.

Total cost:

deg(0) + deg(1) + … + deg(n - 1)

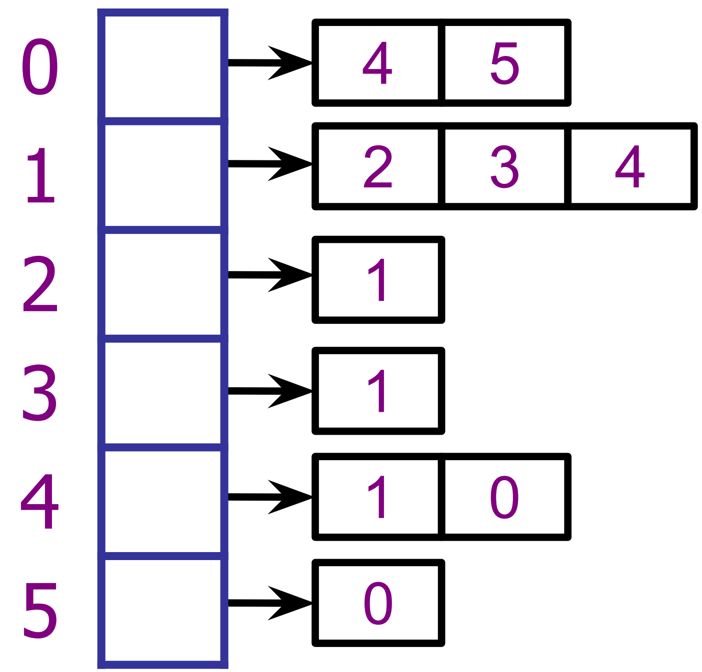
Based on claim:

Cost is O(V + E)



# Breadth-First Search

What if we wanted to know what is the shortest path from any node to node 0?



# Searching a graph

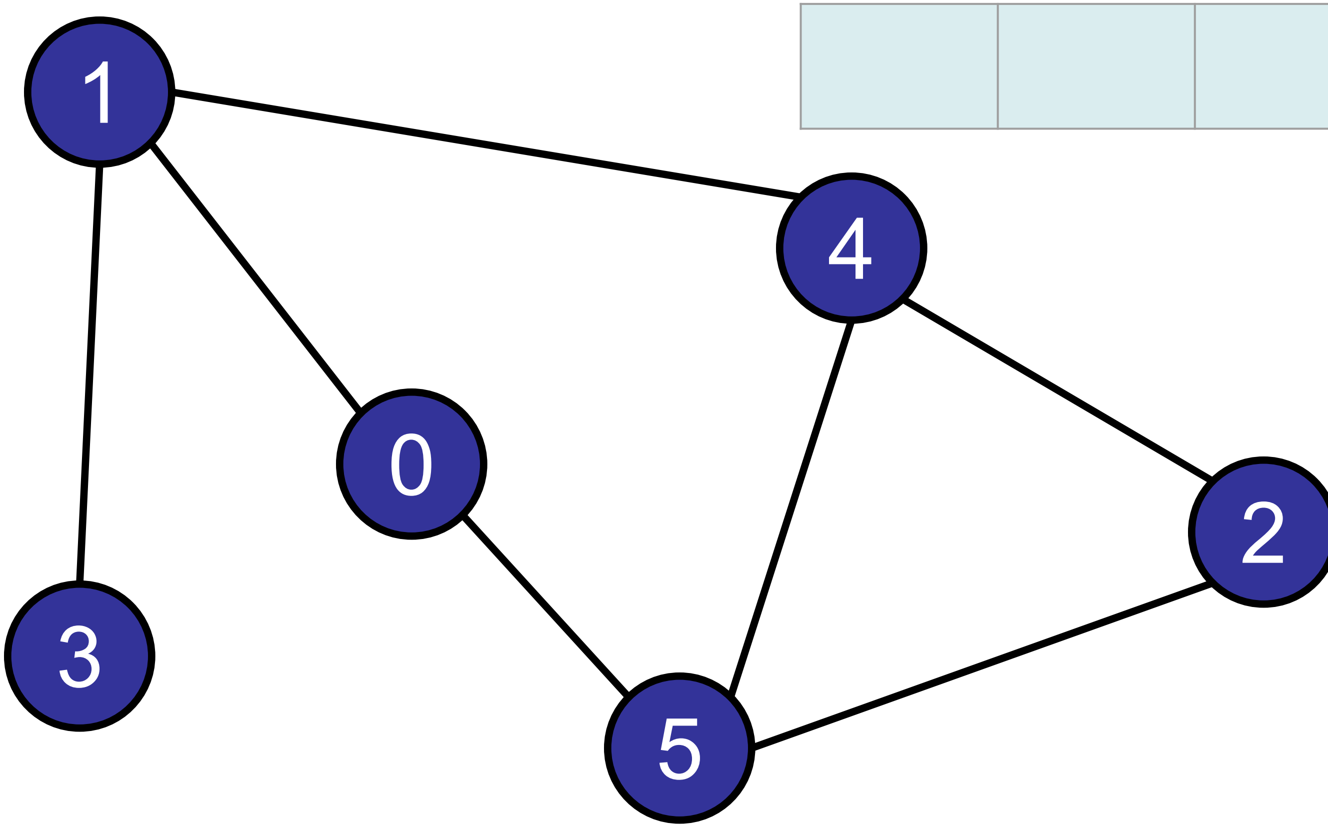
Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
    a. Take next node out of queue.
    b. <u>Go through all neighbours of node</u>
    c. If they have already been visited, skip.
    d. Otherwise, mark them as visited, enqueue them as well.
    e. (New step) Set neighbour's parent to be node

# Breadth-First Search

queue:

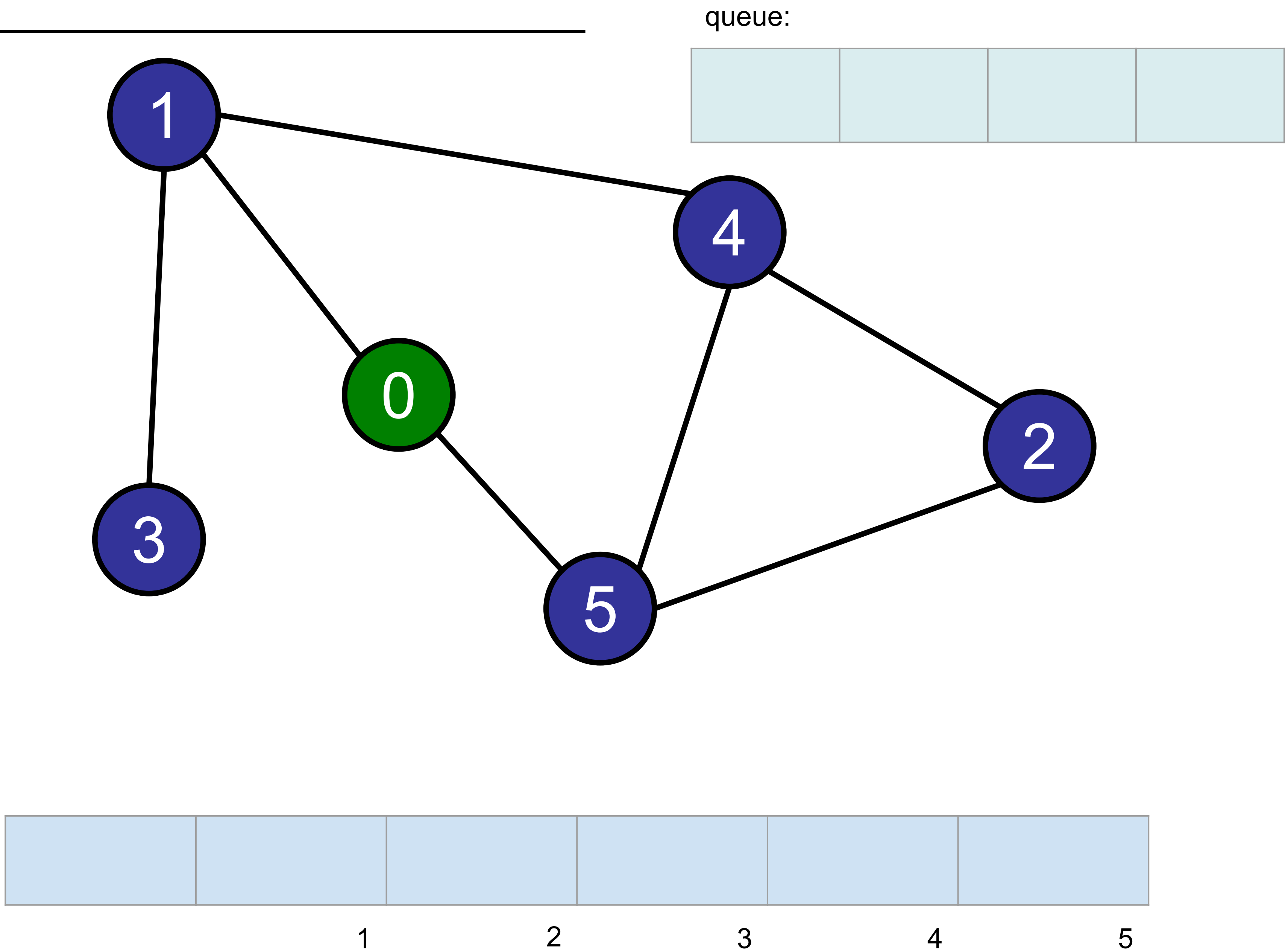
| | | | |
|---|---|---|---|
| | | | 0 |




| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

1 2 3 4 5

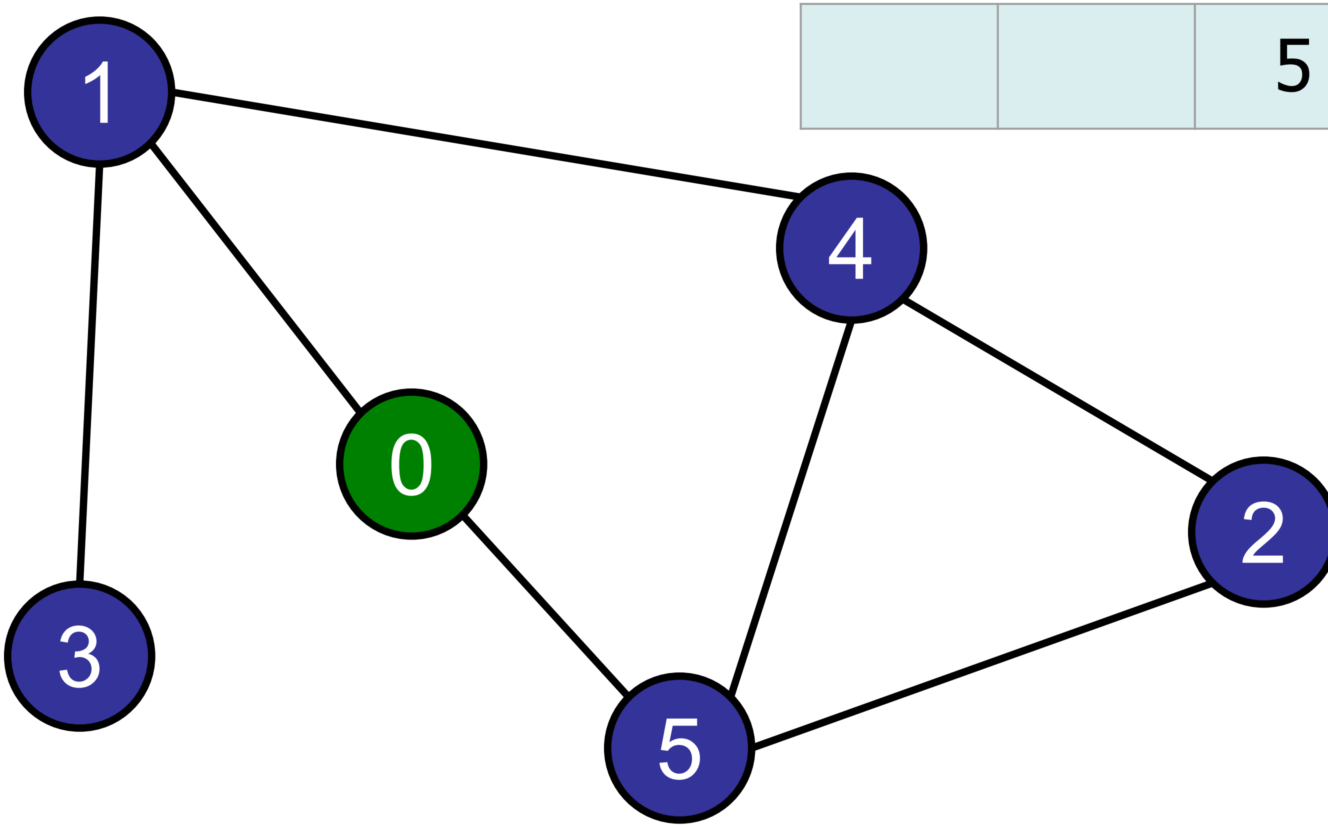
# Breadth-First Search



# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | | 5 | 1 |



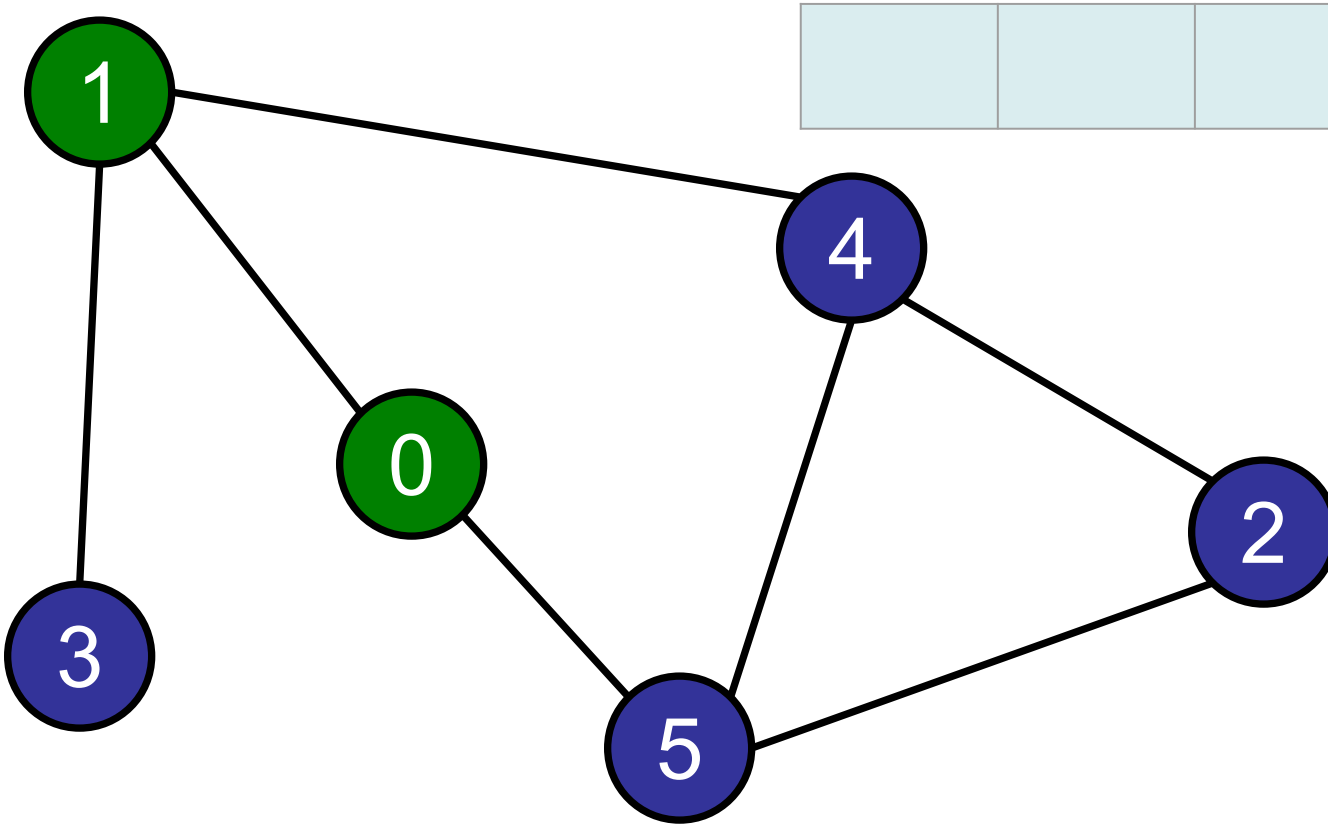

| | | | | | |
|---|---|---|---|---|---|
| | 0 | | | | 0 |

1 2 3 4 5

# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | | | 5 |

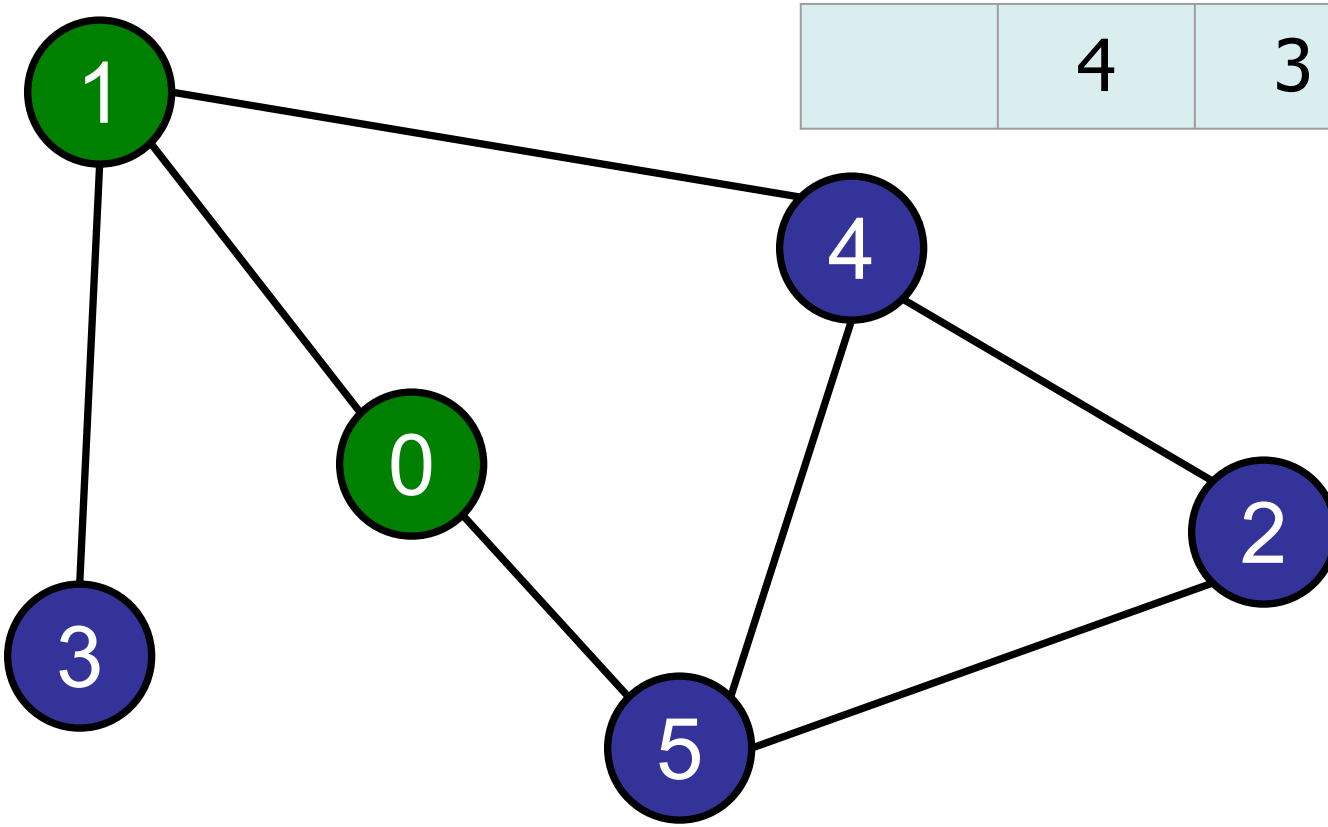



| | | | | | |
|---|---|---|---|---|---|
| | 0 | | | | 0 |
| | 1 | 2 | 3 | 4 | 5 |

# Breadth-First Search

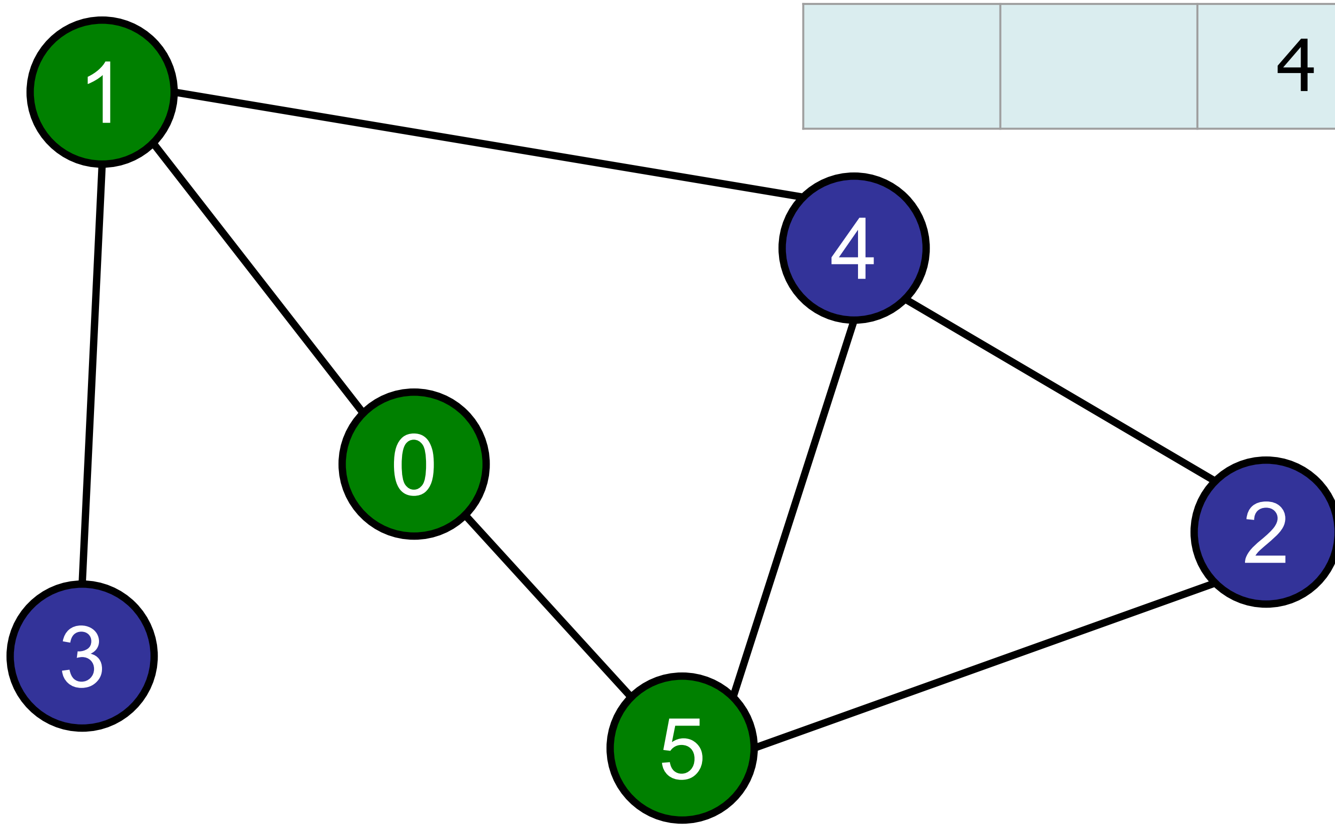
queue:

| | | | |
|---|---|---|---|
| | 4 | 3 | 5 |




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

# Breadth-First Search




queue:

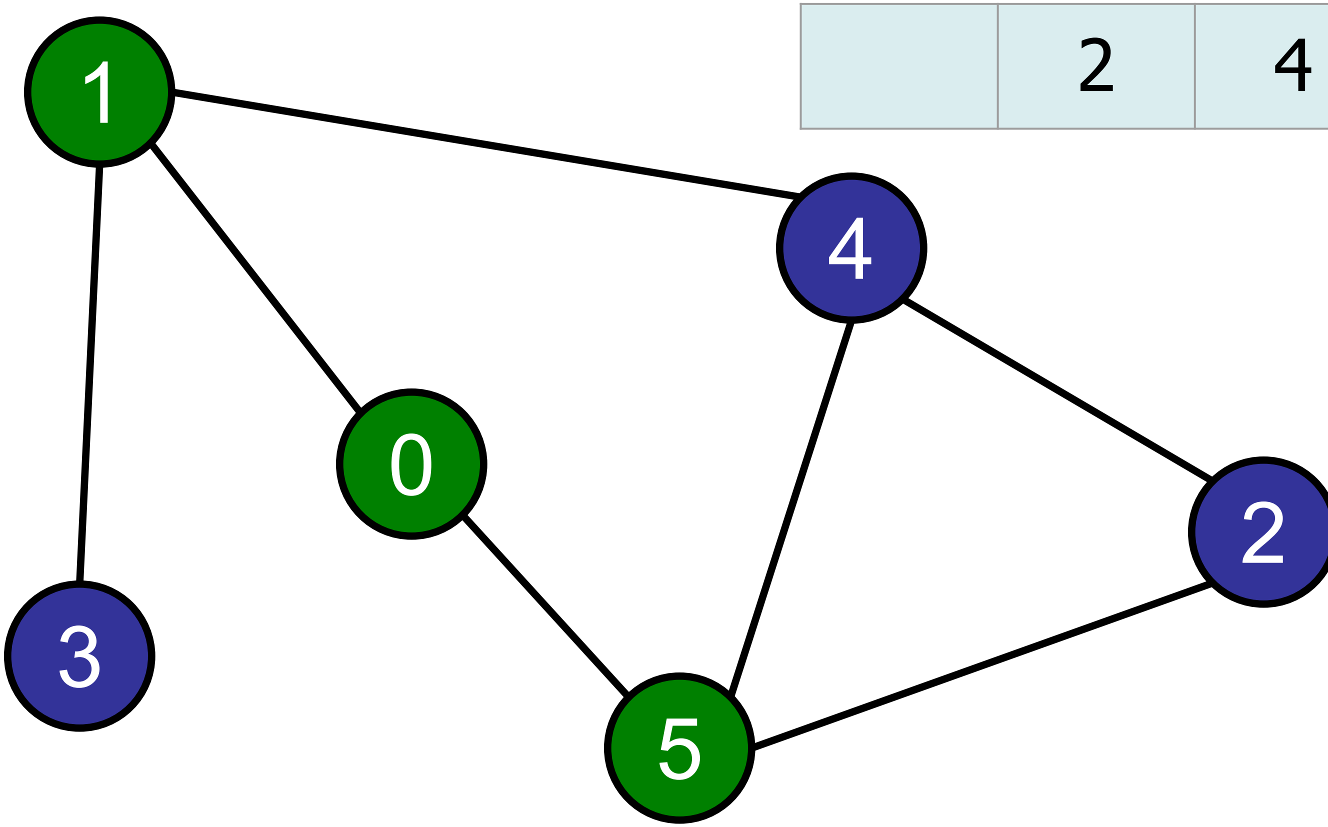
| | | | |
|---|---|---|---|
| | | 4 | 3 |

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

# Breadth-First Search

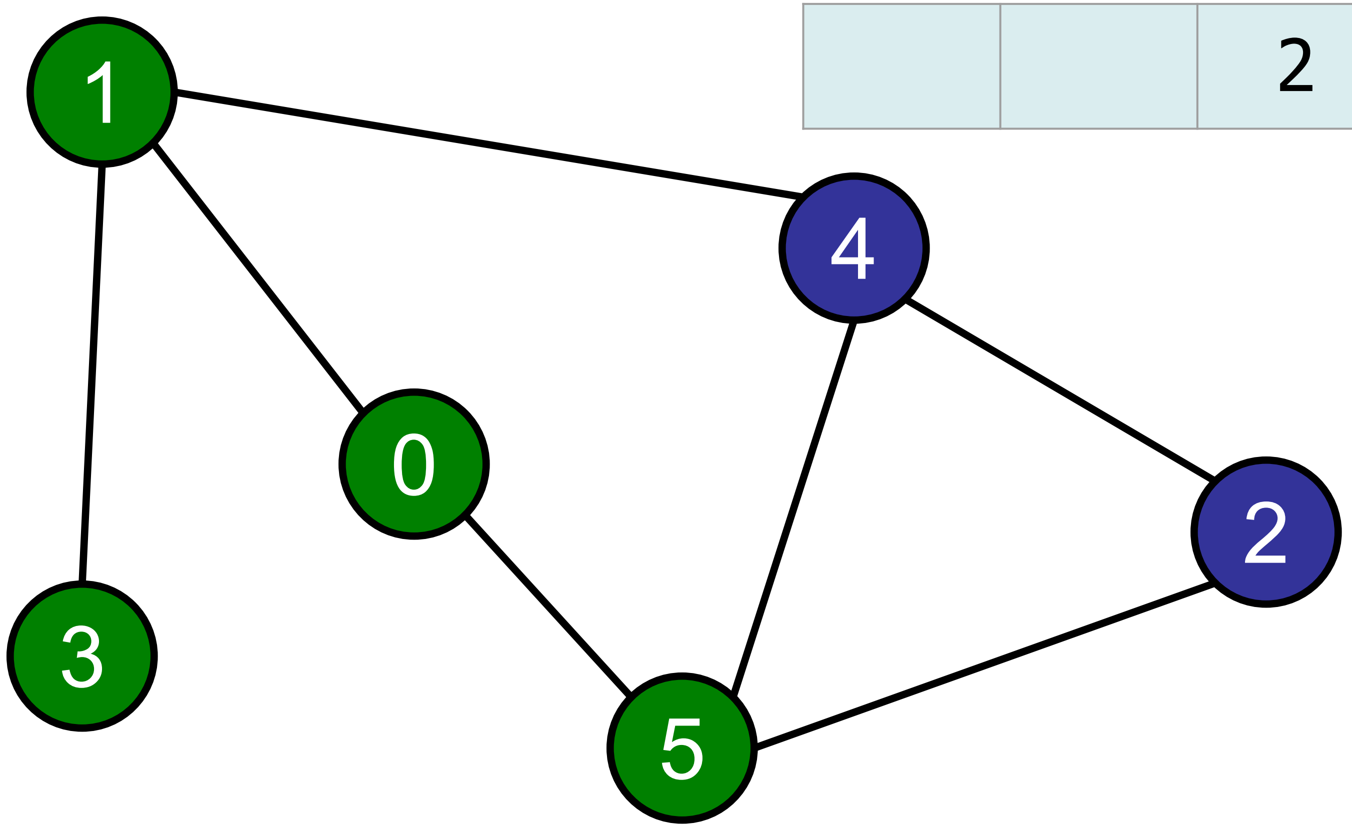
queue:

| | | | |
|---|---|---|---|
| | 2 | 4 | 3 |




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

# Breadth-First Search




queue:

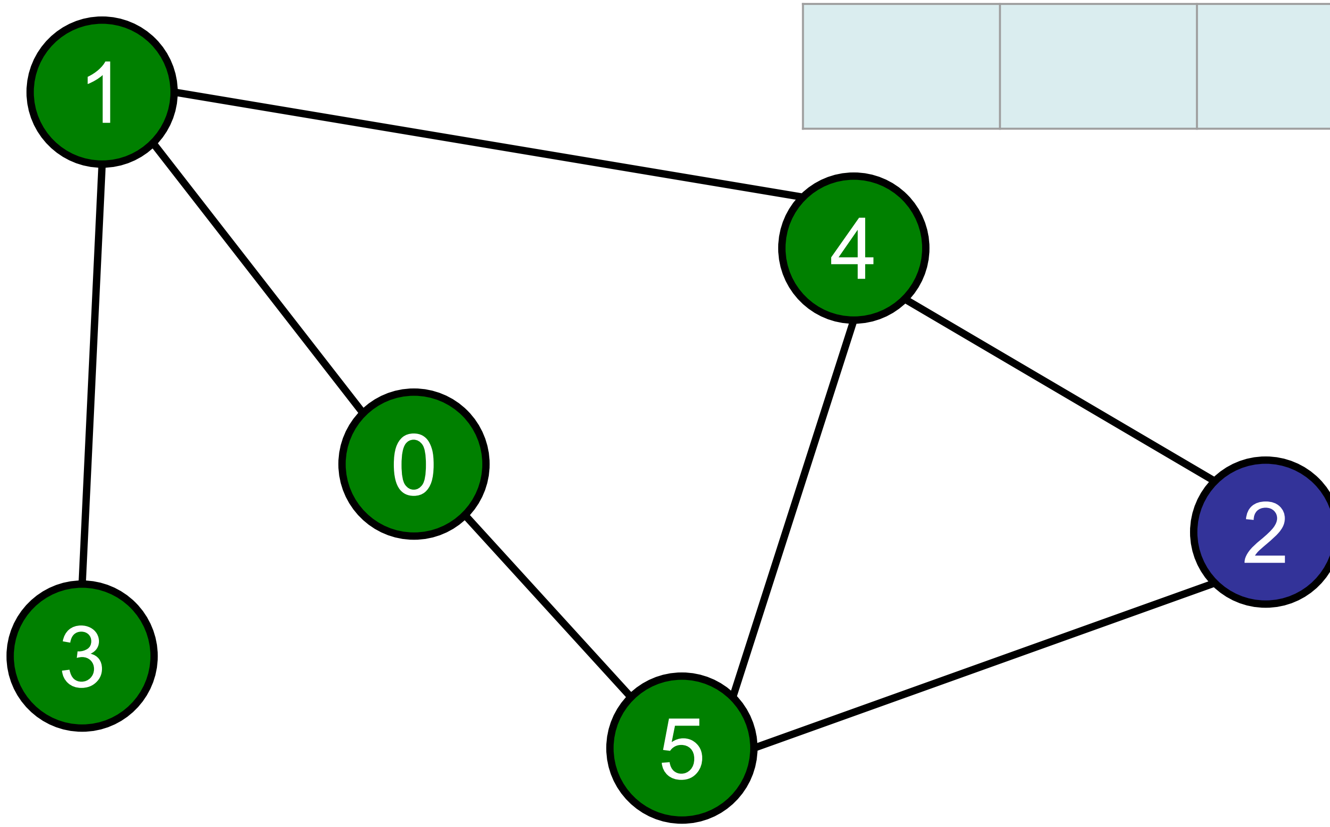
| | | | |
|---|---|---|---|
| | | 2 | 4 |

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

# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | | | 2 |

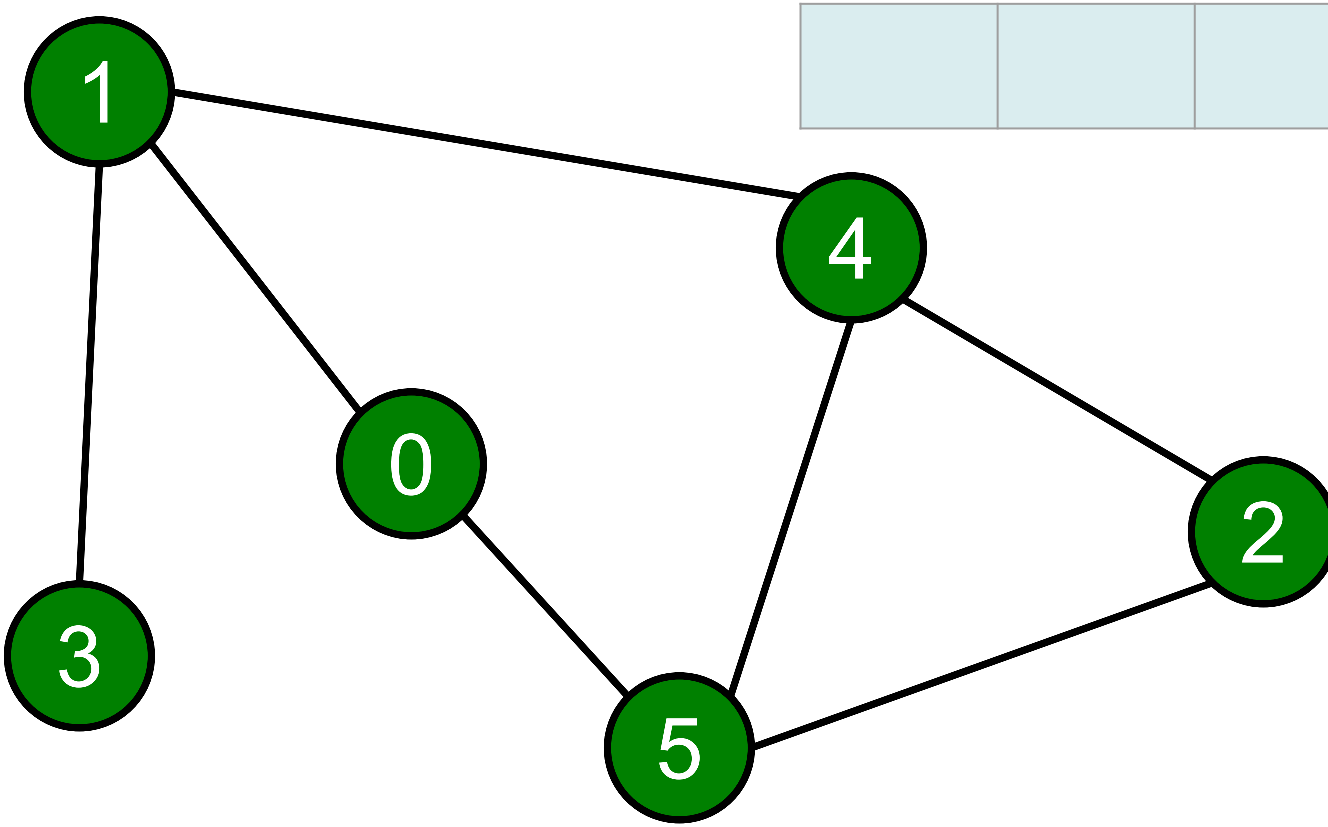



| | | | | | |
|---|---|---|---|---|---|
| | 0 | 5 | 1 | 1 | 0 |
| | 1 | 2 | 3 | 4 | 5 |

# Breadth-First Search

queue:

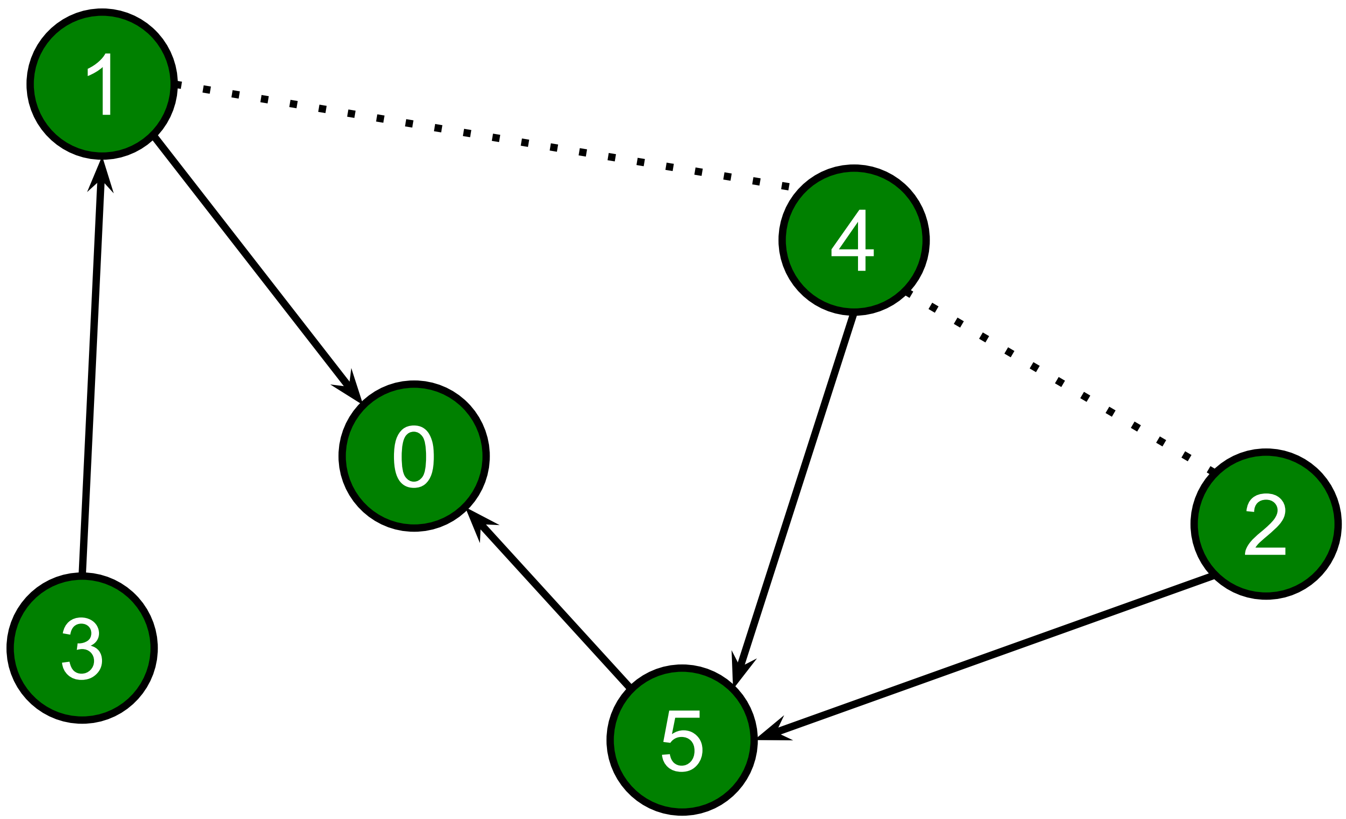
| | | | |
|---|---|---|---|
| | | | |




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

# Breadth-First Search

addition from lecture:
this is the shortest path graph



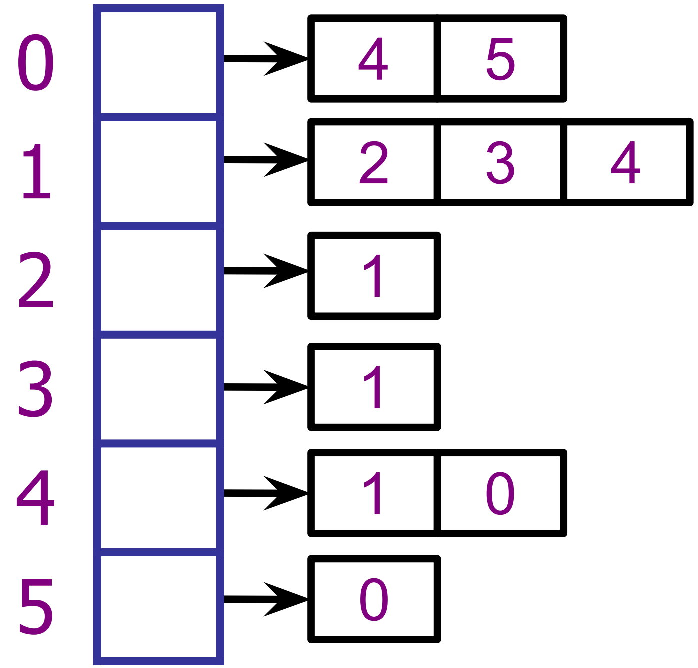

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 5 | 1 | 1 | 0 |
| | 1 | 2 | 3 | 4 | 5 |

## Which is true? (More than one may apply.)

1. Shortest path graph is a cycle.
2. Shortest path graph is a tree.
3. Shortest path graph has low-degree.
4. Shortest path graph has low diameter.
5. None of the above.

# Breadth-First Search

What if we wanted to know what is the **length of the** shortest path to any node from node 0?



# Searching a graph

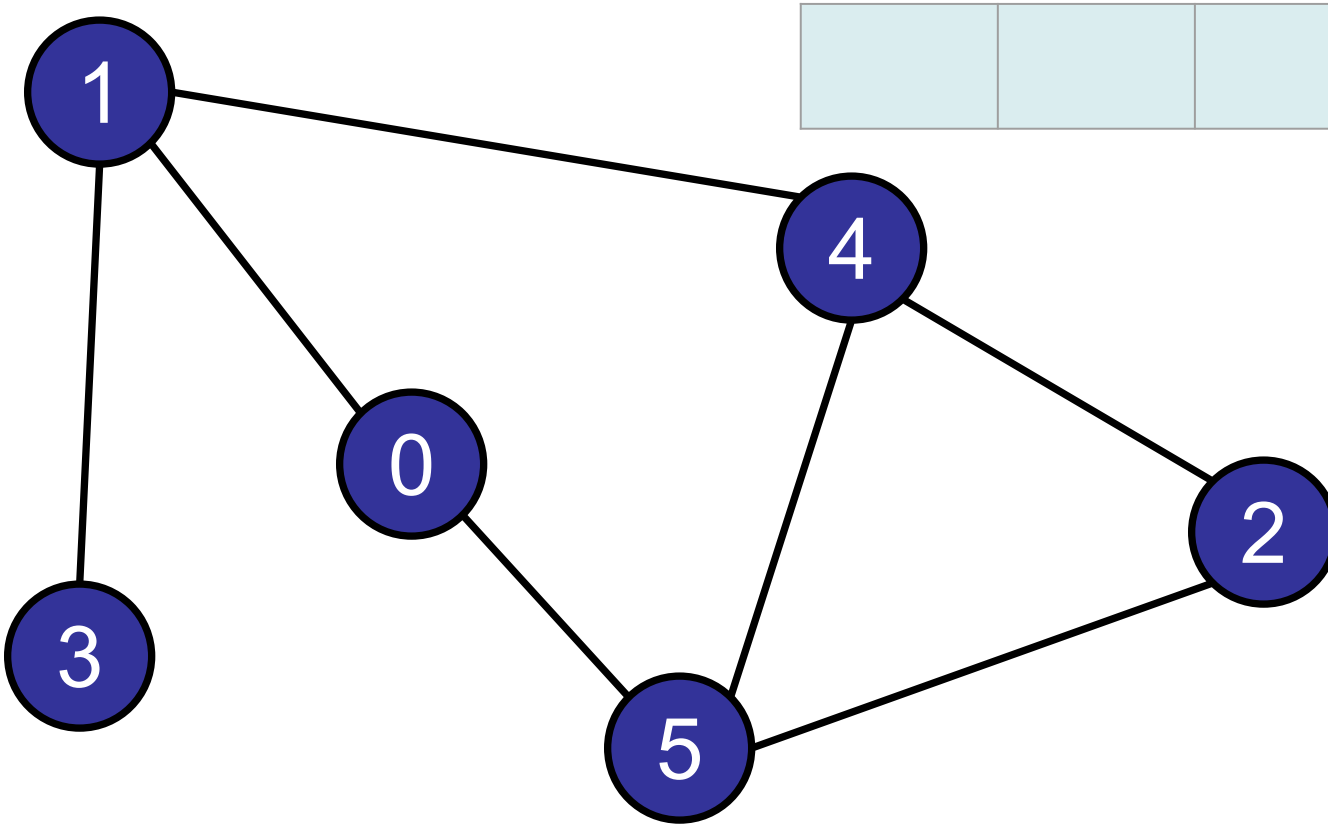
Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
    a. Take next node out of queue.
    b. Go through all neighbours of node
    c. If they have already been visited, skip.
    d. Otherwise, mark them as visited, enqueue them as well.
    e. (New step) Set neighbour's parent to be node
    f. (New step) Set neighbour's distance to be node's distance + 1

# Breadth-First Search

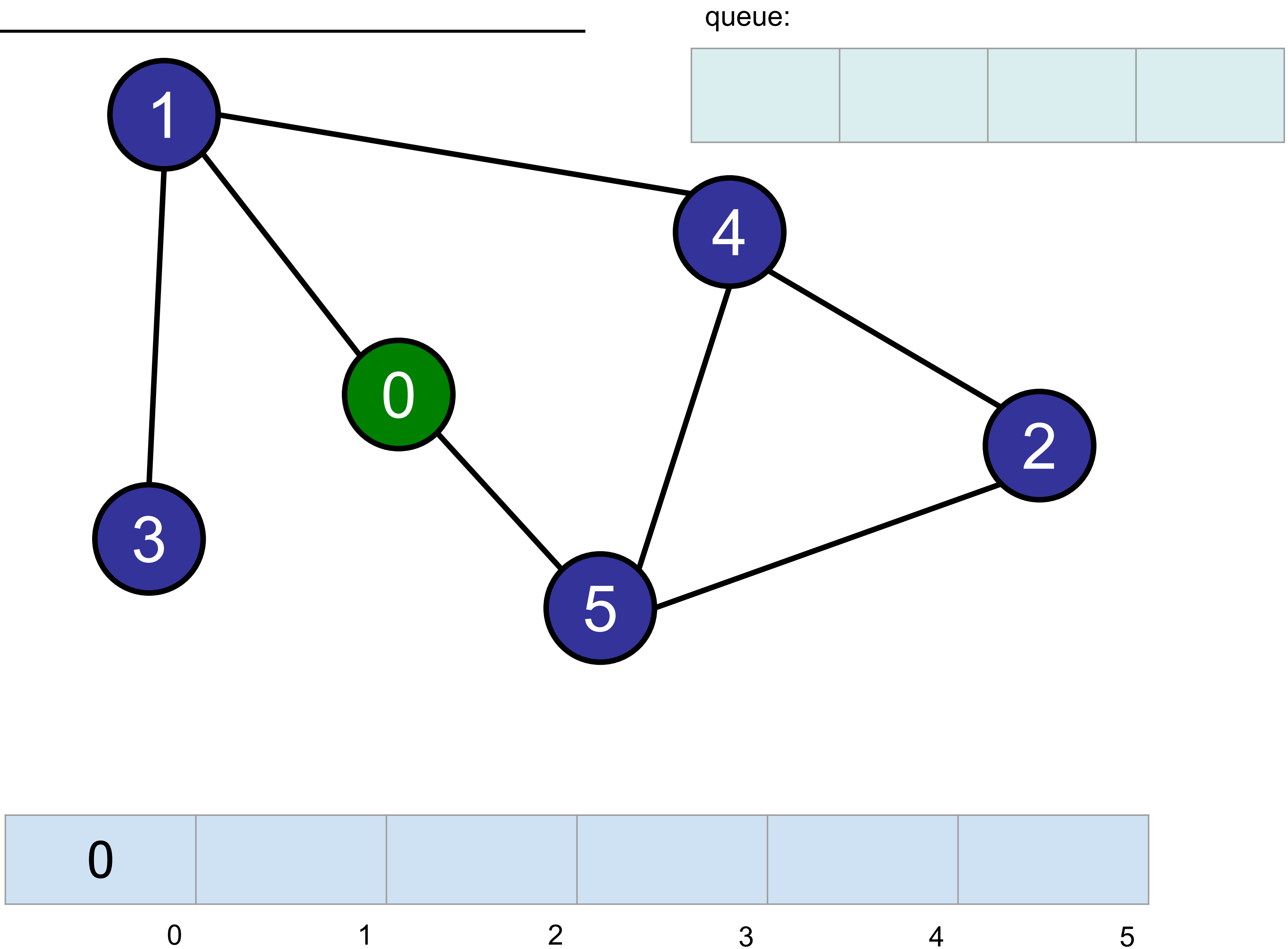
queue:

| | | | |
|---|---|---|---|
| | | | 0 |




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

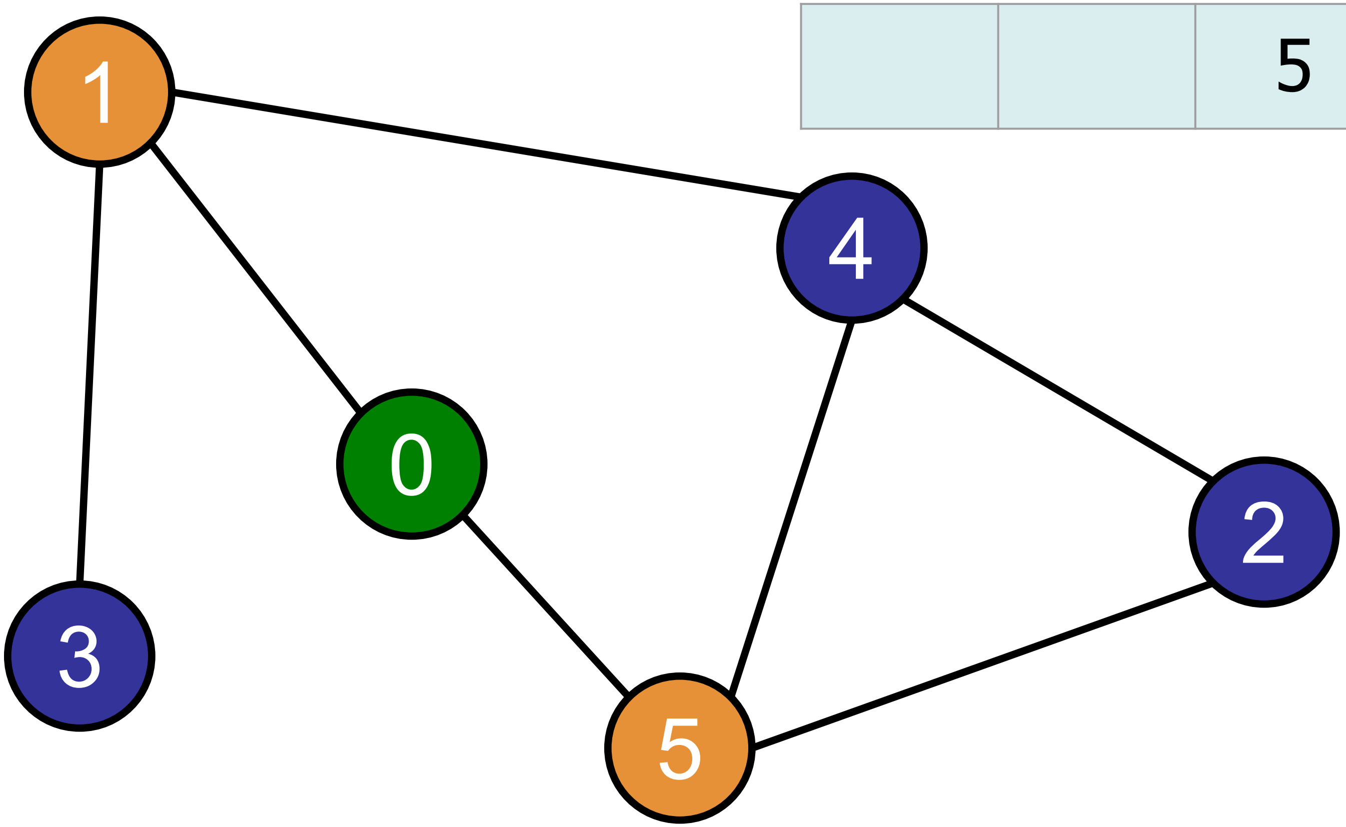
# Breadth-First Search



# Breadth-First Search

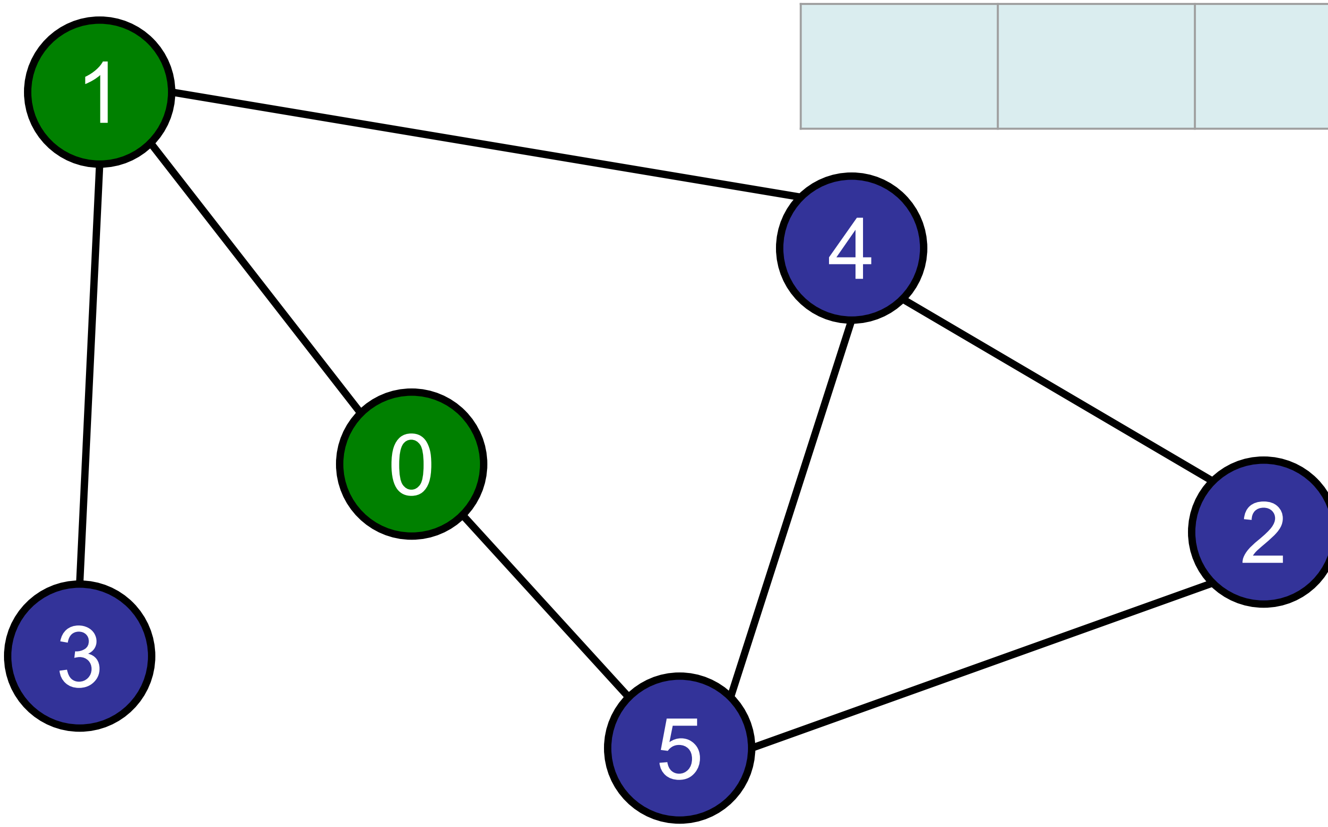
queue:

| | | | |
|---|---|---|---|
| | | 5 | 1 |




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

# Breadth-First Search




queue:

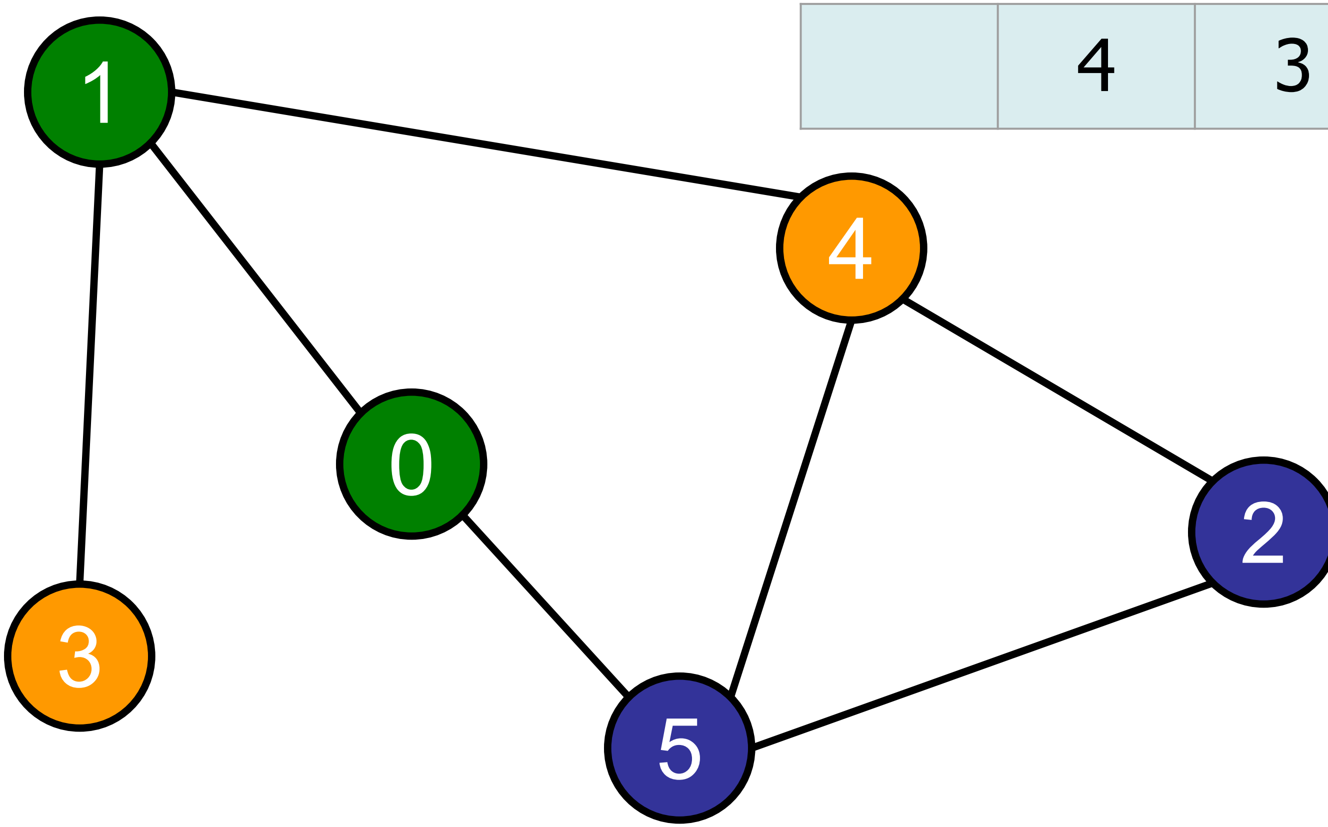
| | | | |
|---|---|---|---|
| | | | 5 |

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

# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | 4 | 3 | 5 |

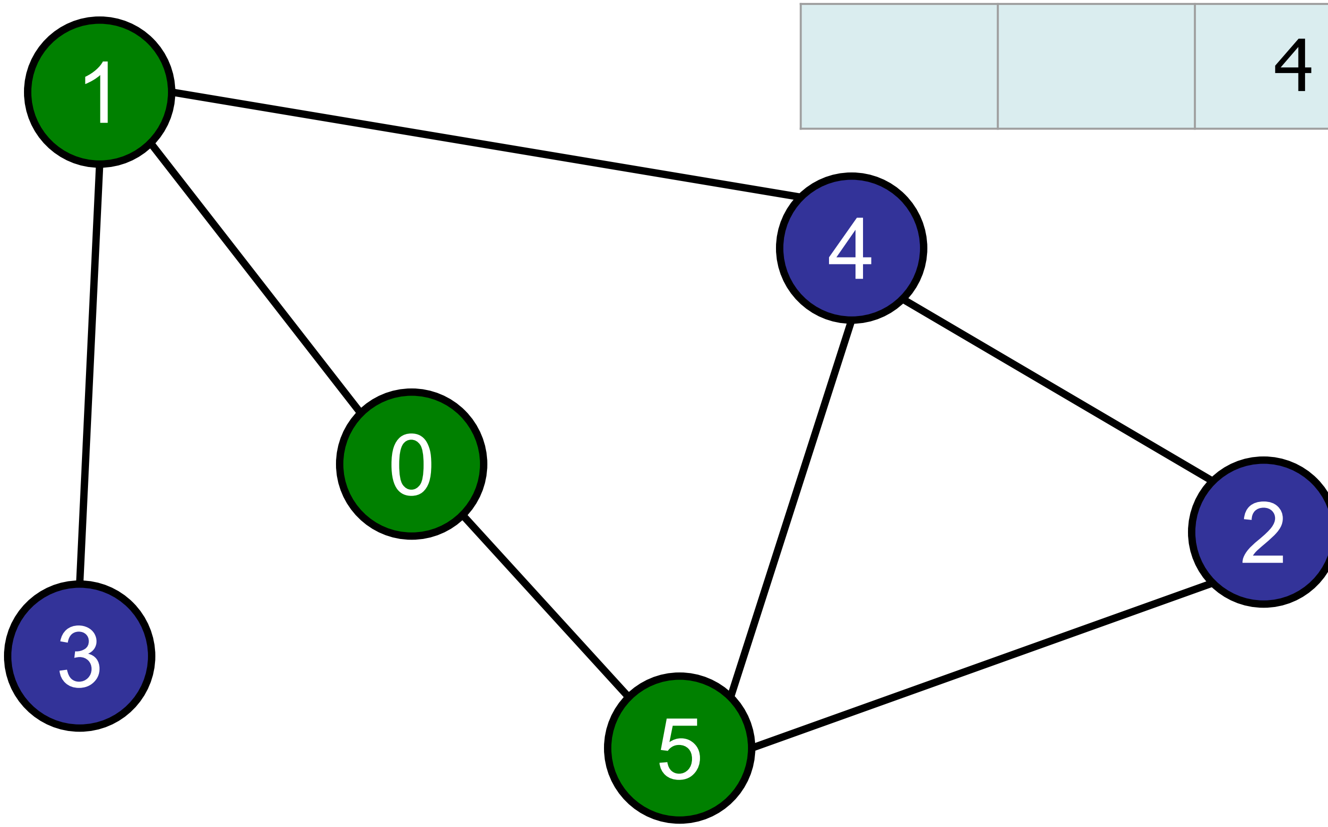



| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | 1 | | 2 | 2 | 1 |

# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | | 4 | 3 |

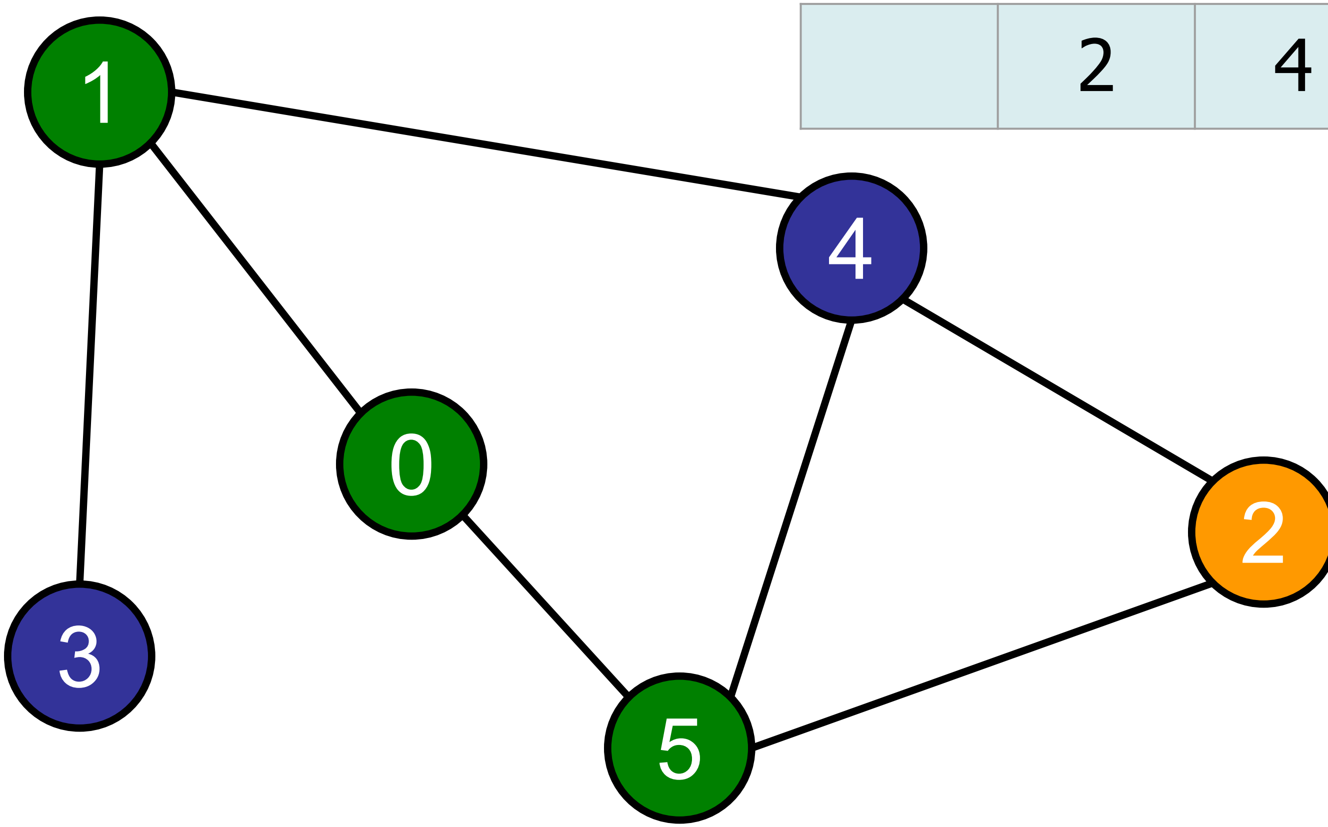



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | | 2 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Breadth-First Search

queue:

| | | | |
|---|---|---|---|
| | 2 | 4 | 3 |

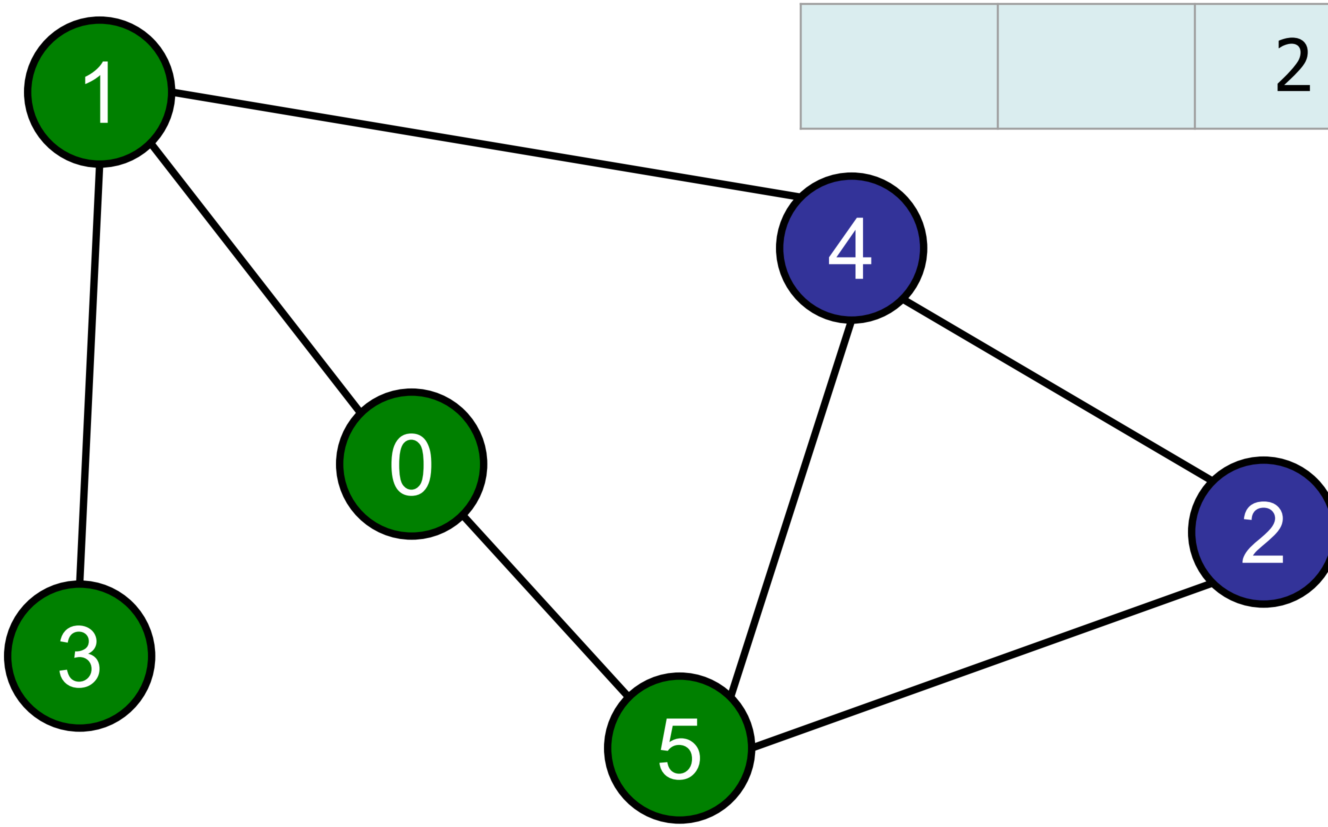



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 2 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Breadth-First Search

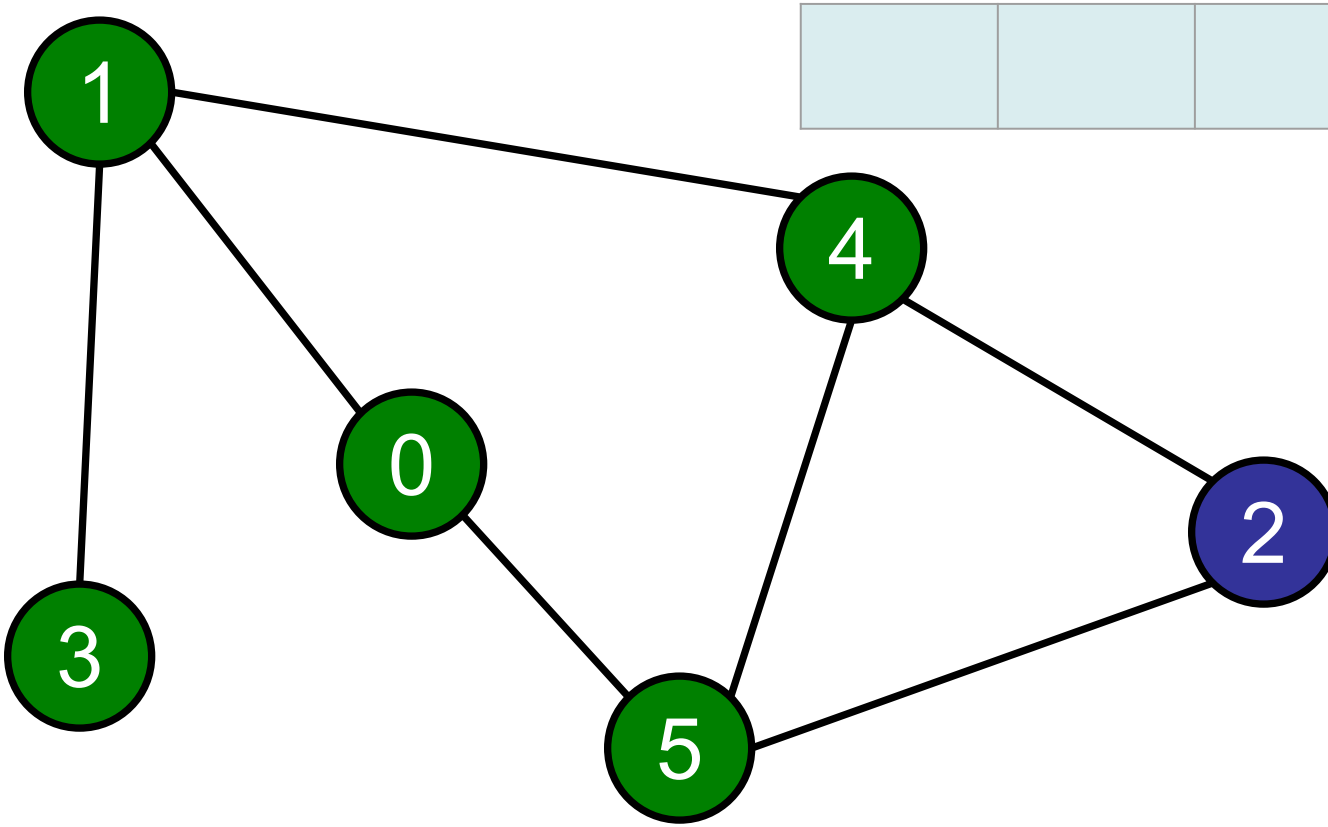
queue:

| | | | |
|---|---|---|---|
| | | 2 | 4 |




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

# Breadth-First Search




queue:

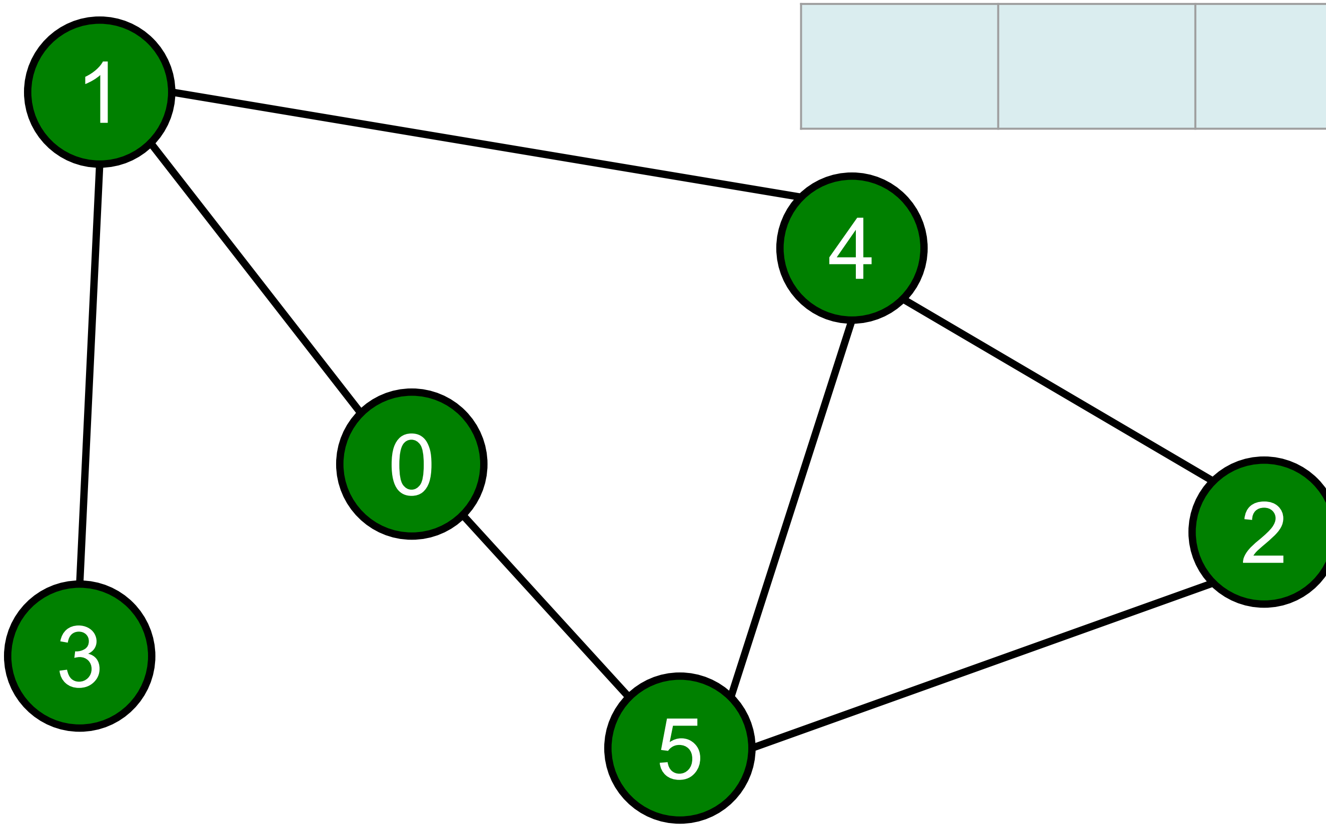
| | | | |
|---|---|---|---|
| | | | 2 |

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

# Breadth-First Search

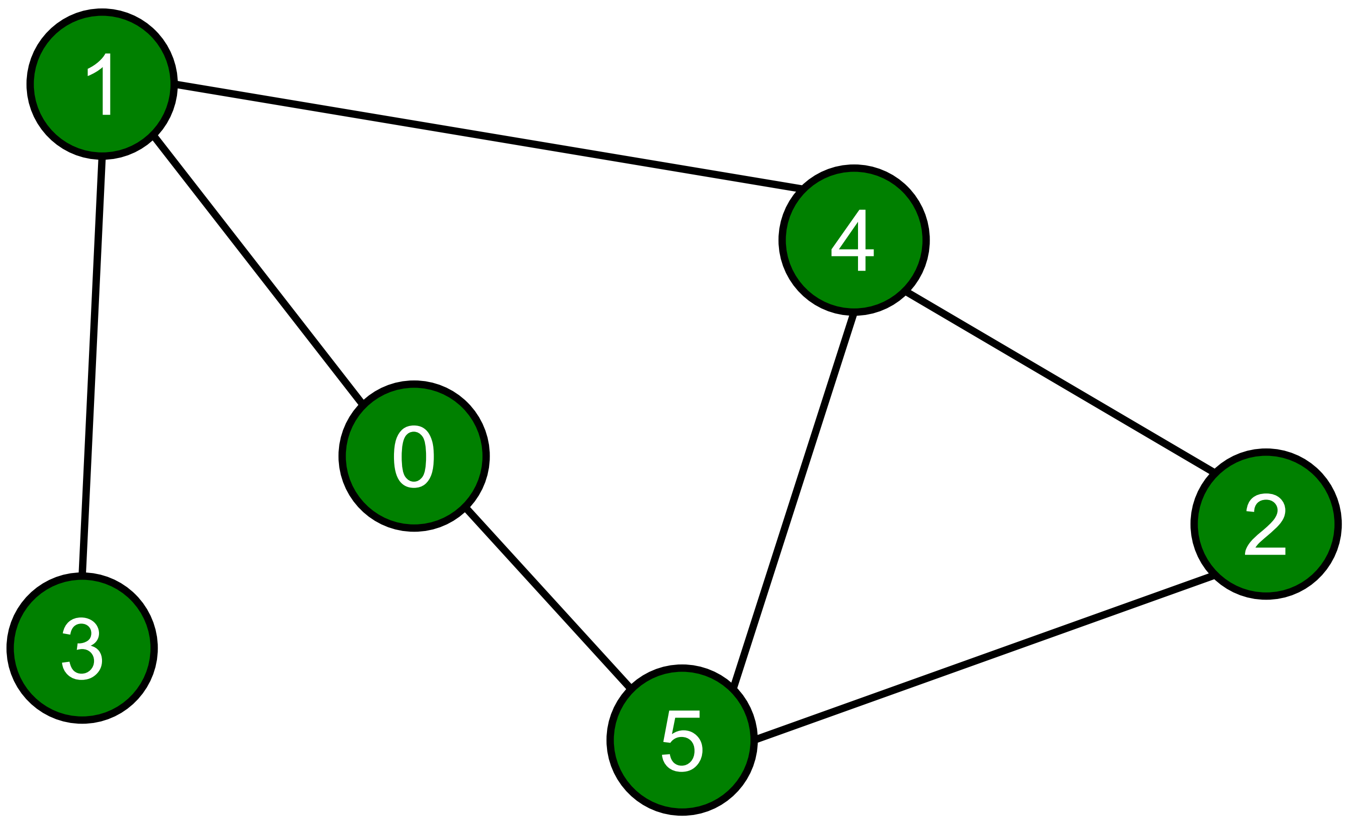
queue:

| | | | |
|---|---|---|---|
| | | | |




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

# Breadth-First Search




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

# Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

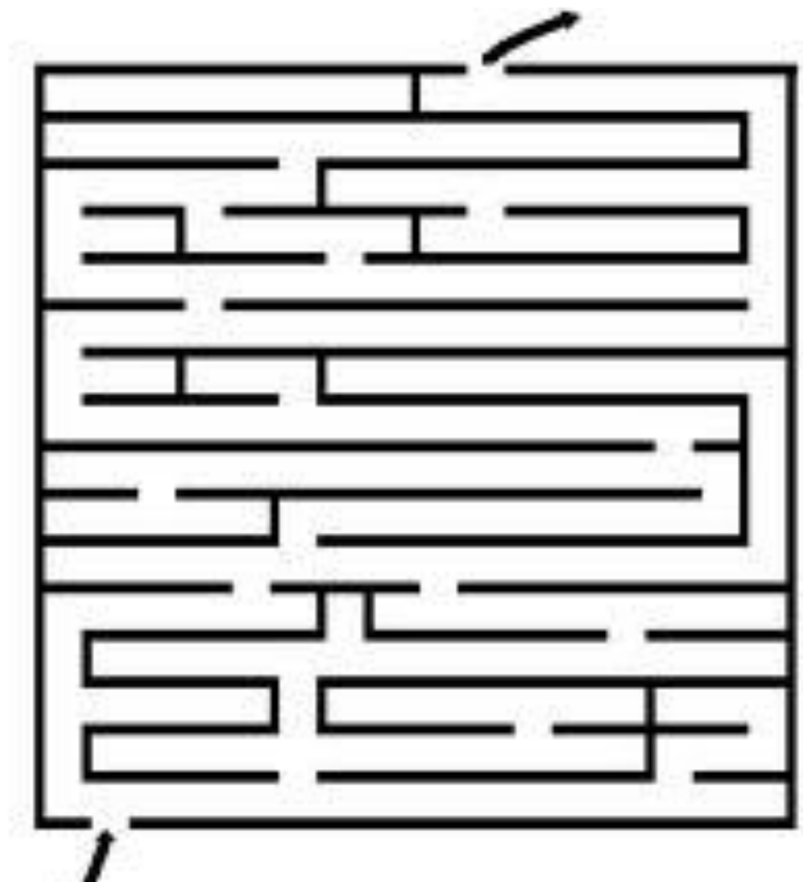
Graph representation:

- Adjacency list

# Depth-First Search

Exploring a maze:

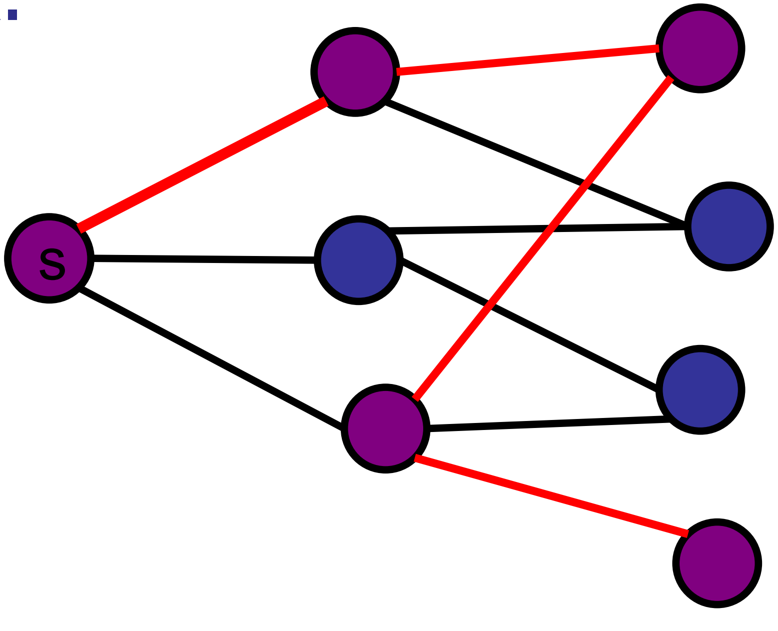
- Follow path until stuck.
- Backtrack along breadcrumbs until reach unexplored neighbor.
- Recursively explore.



# Searching a graph

Depth-First Search:

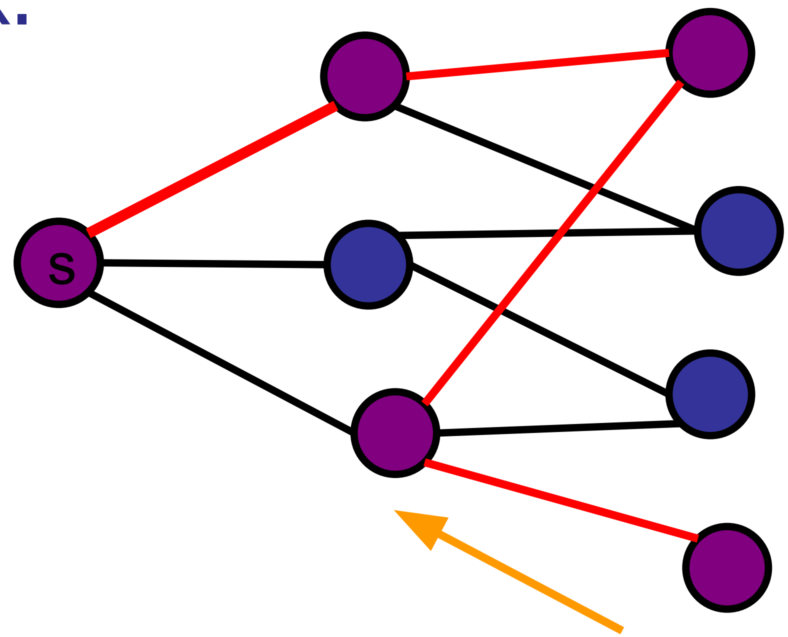
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



# Searching a graph

Depth-First Search:

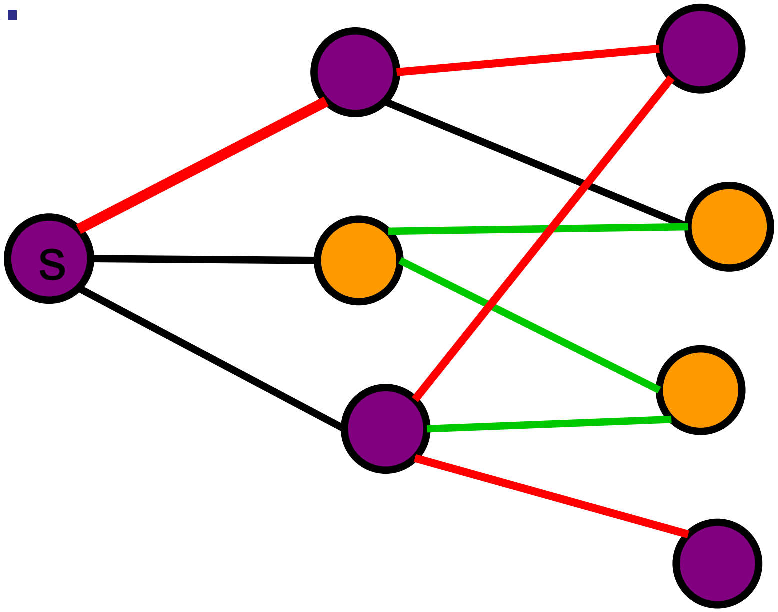
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



# Searching a graph

Depth-First Search:

- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don't repeat a vertex.



# Depth-First Search

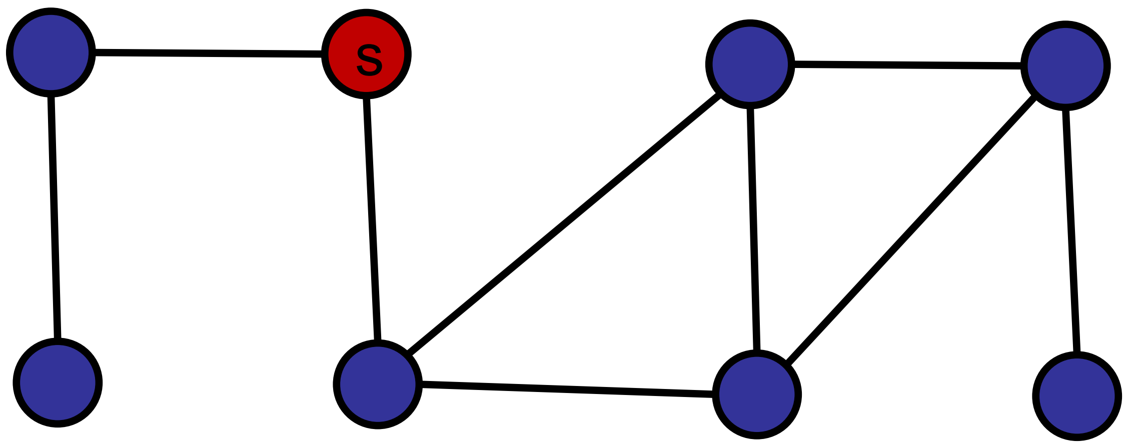
```
void dfs(ArrayList<ArrayList<Integer>> adjacency_list, int src, int dst){
  int num_nodes = adjacency_list.size();
  Stack<Integer> stack = new LinkedList<>();

  // values defaulted to false
  boolean[] visited = new boolean[num_nodes];

  stack.push(src);
  visited[src] = true;
  while(!stack.isEmpty()){
    int current_node = stack.pop();
    for(int neighbour_node : adjacency_list.get(current_node)){
      if(visited[neighbour_node]){
        continue;
      }
      visited[neighbour_node] = true;
      queue.push(neighbour_node);
    }
  }
  return visited[dst];
}
```

Just change it to a stack

# Depth-First Search Example



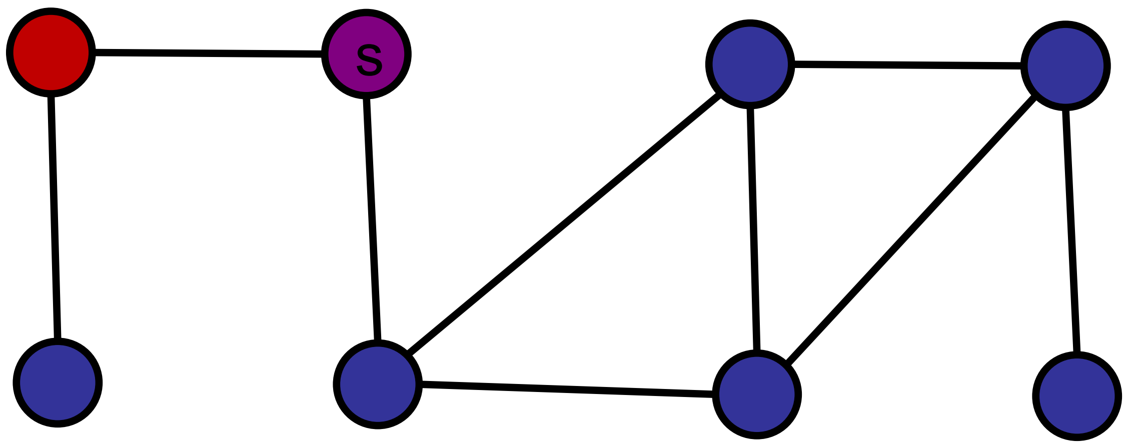

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



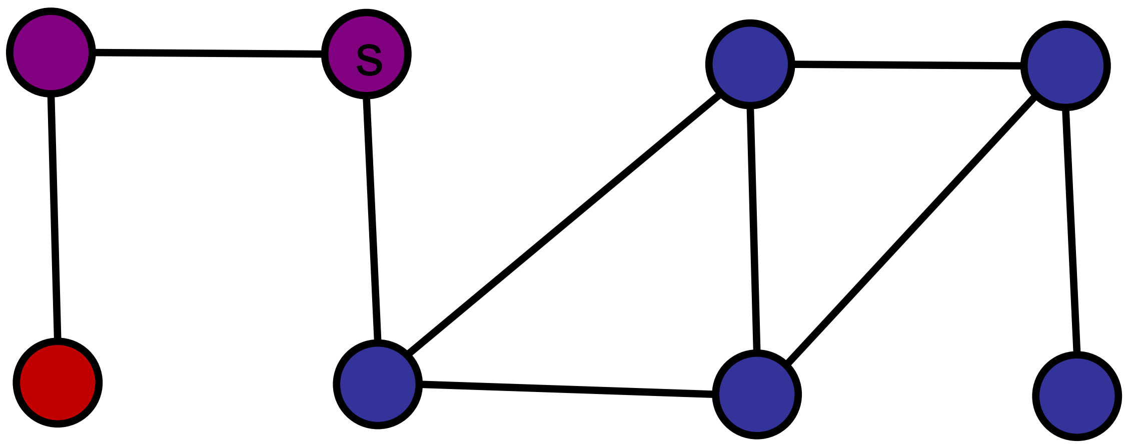

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



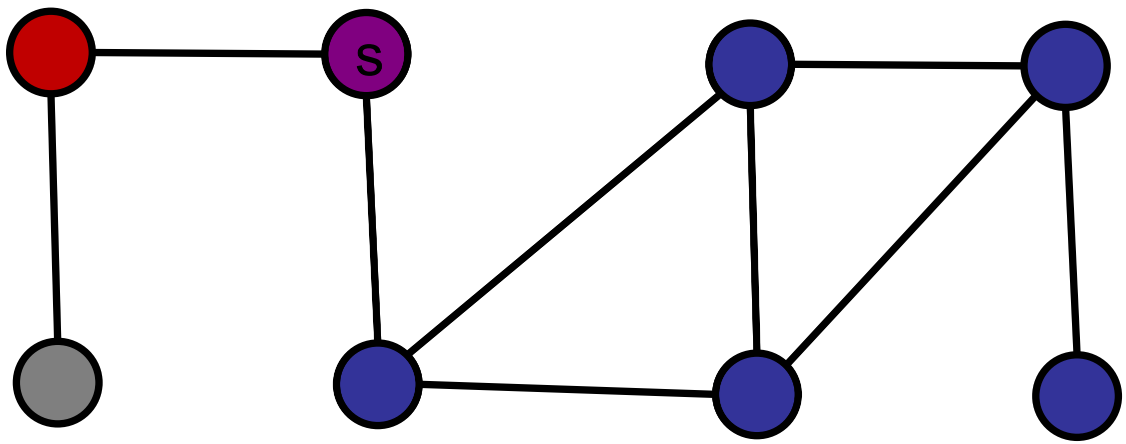

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



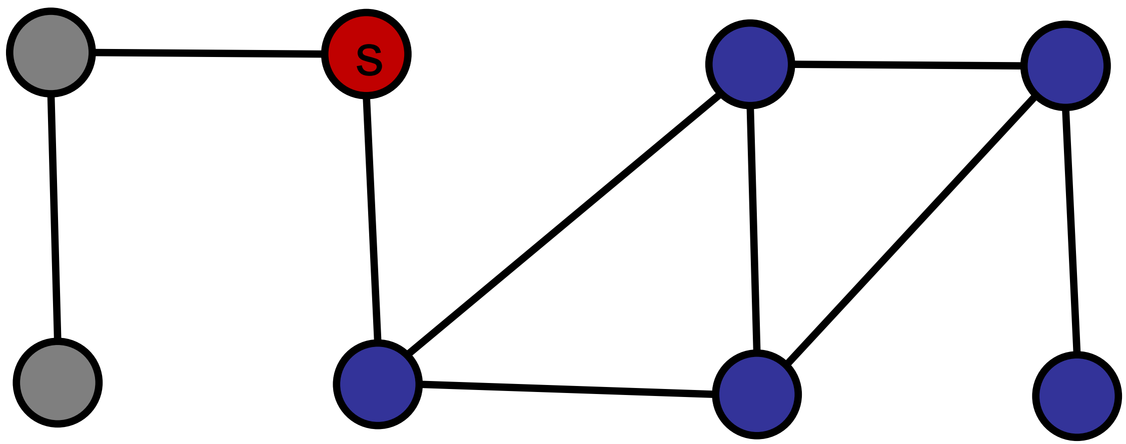

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



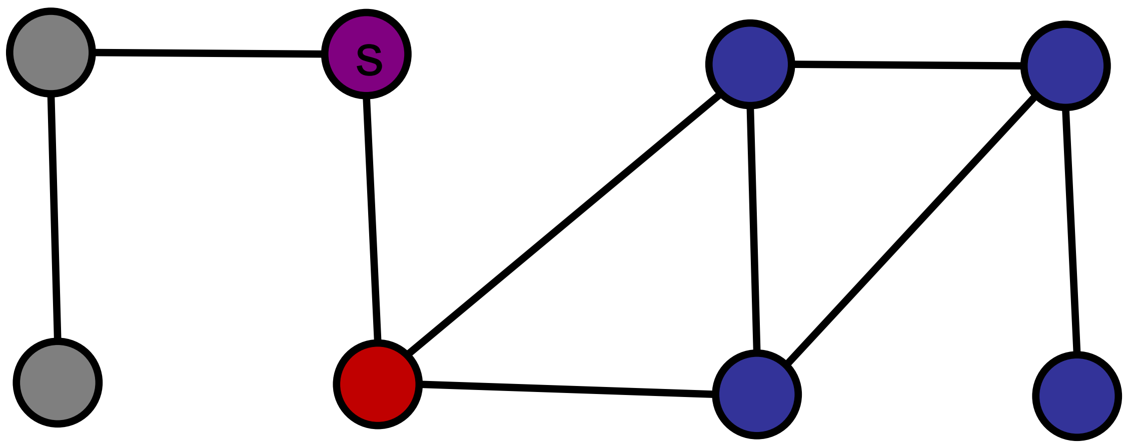

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



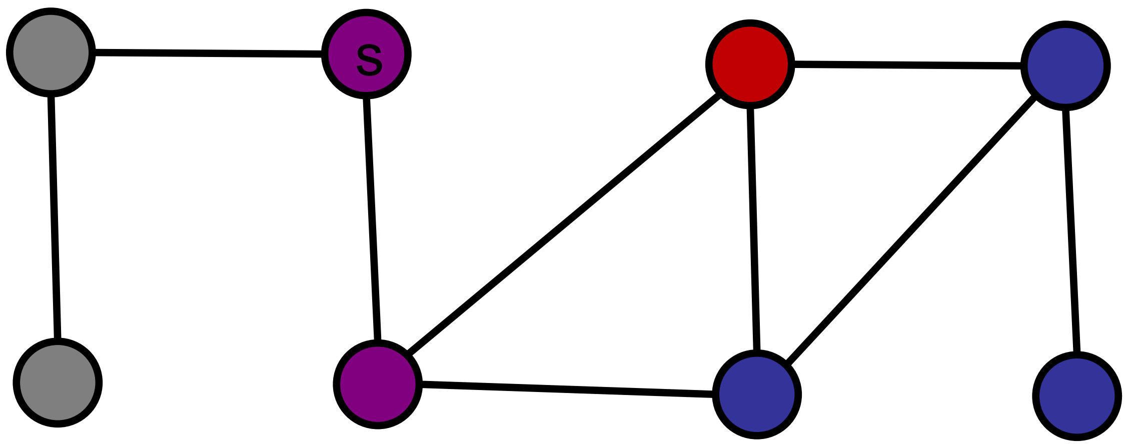

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



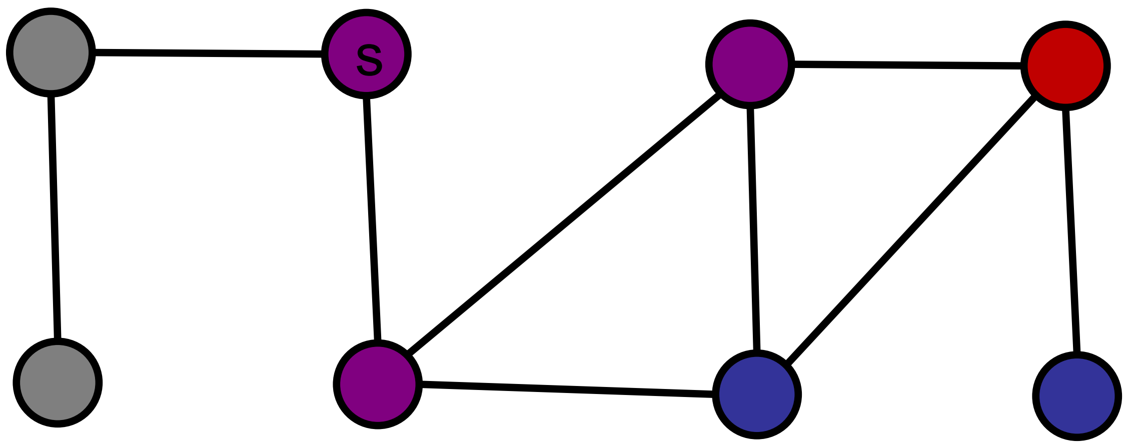

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



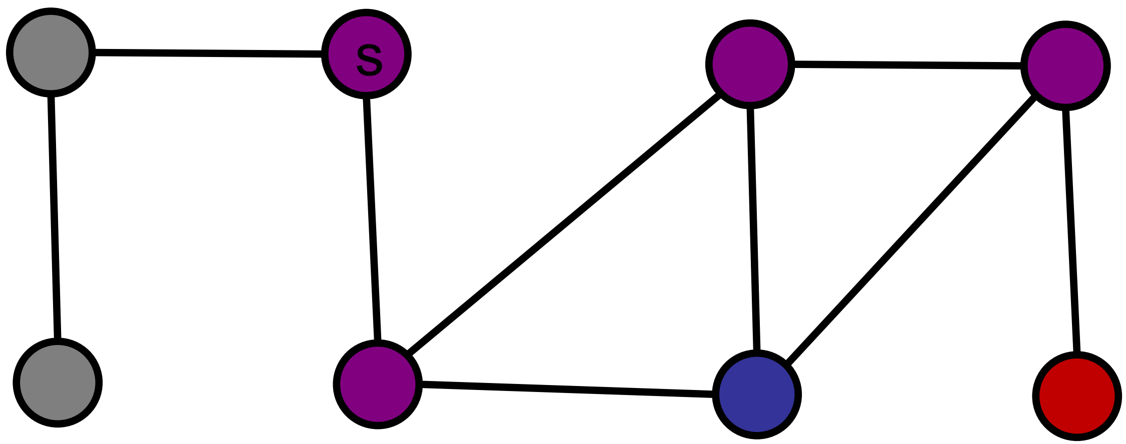

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



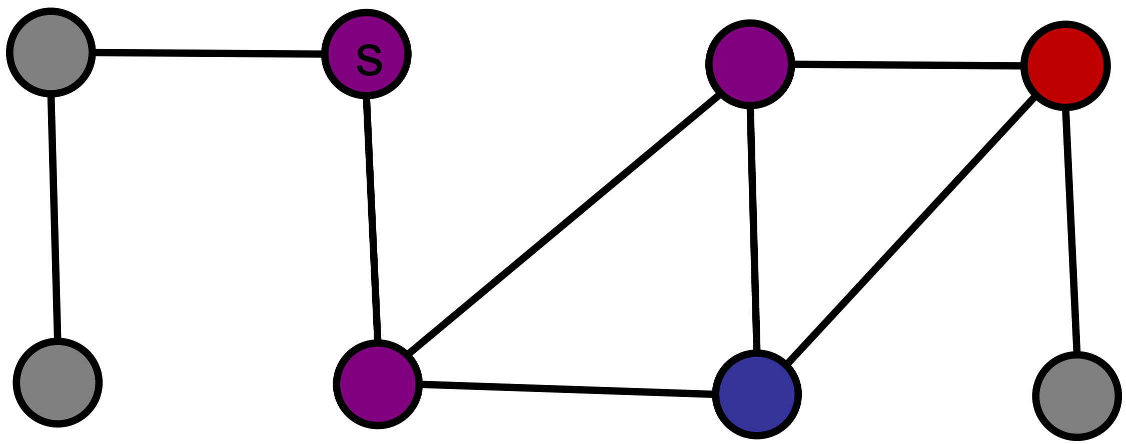

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



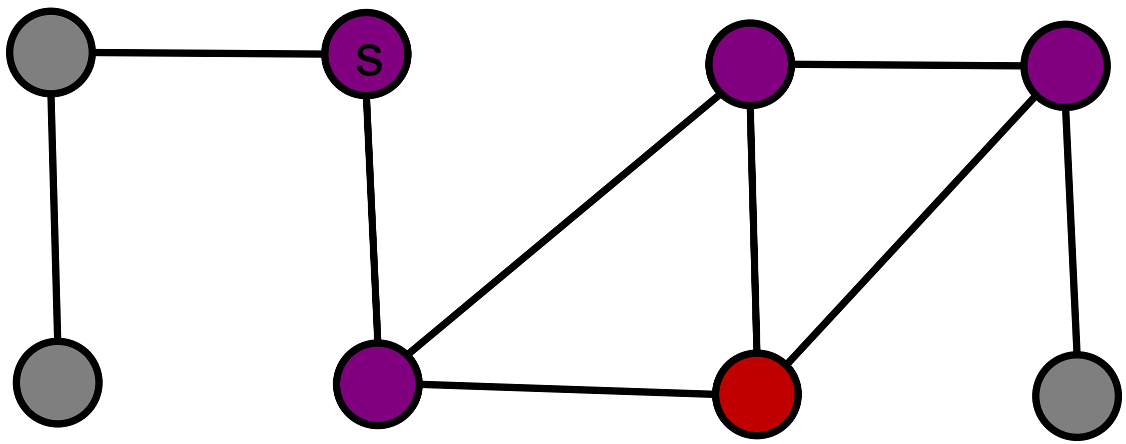

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



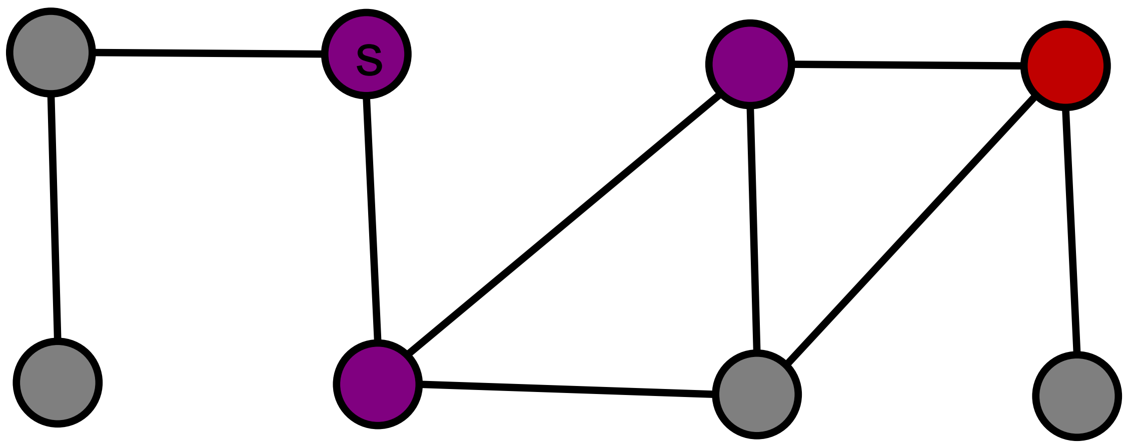

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



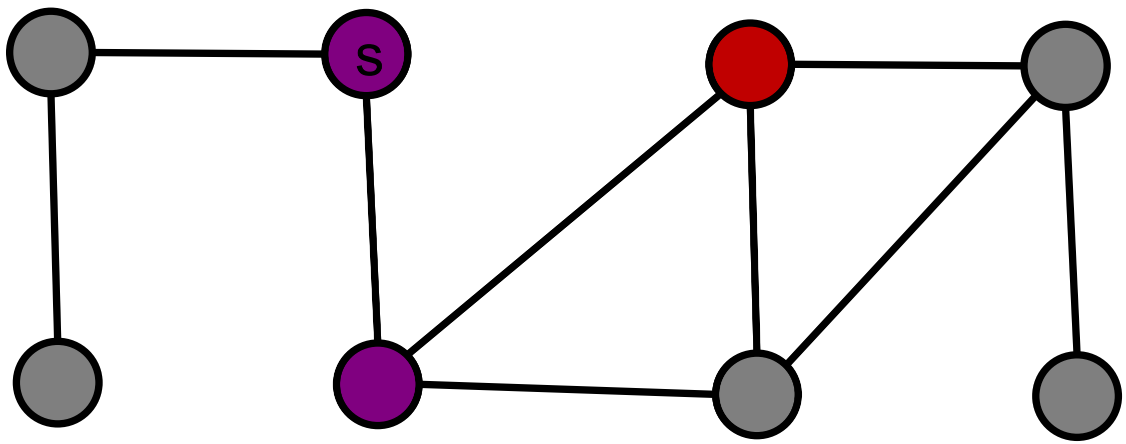

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



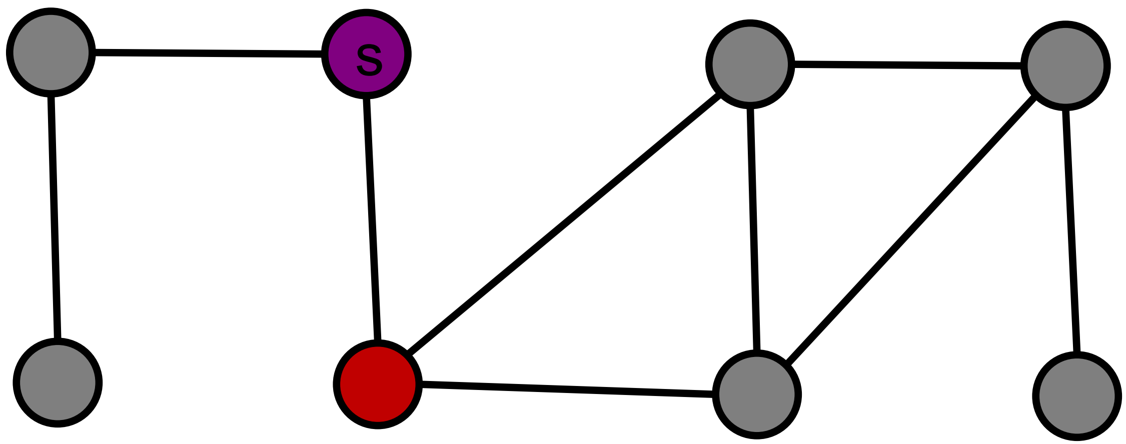

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



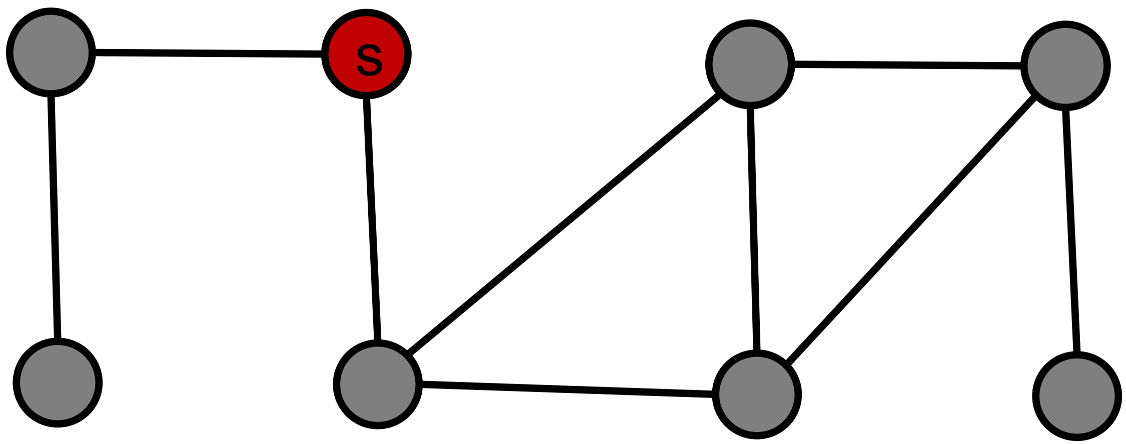

Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

# Depth-First Search Example



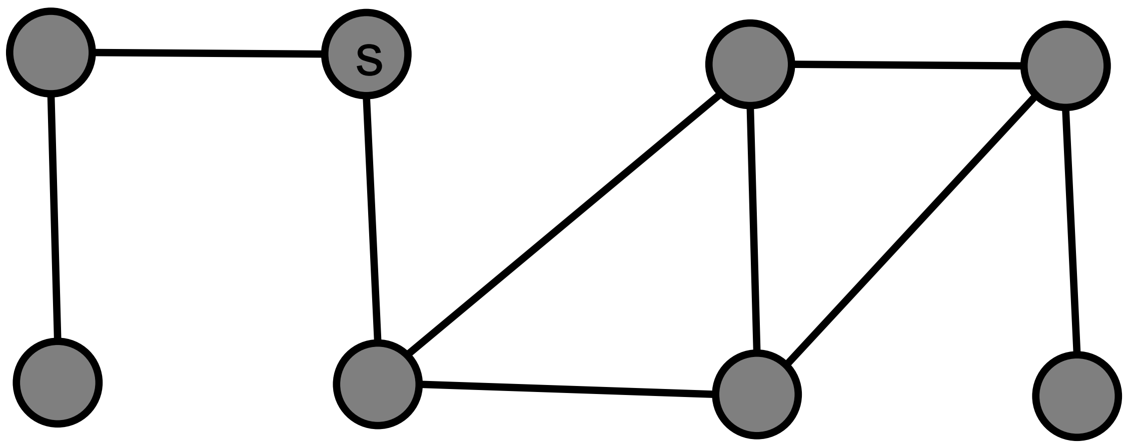

Red = active frontier

Purple = next

Gray = visited

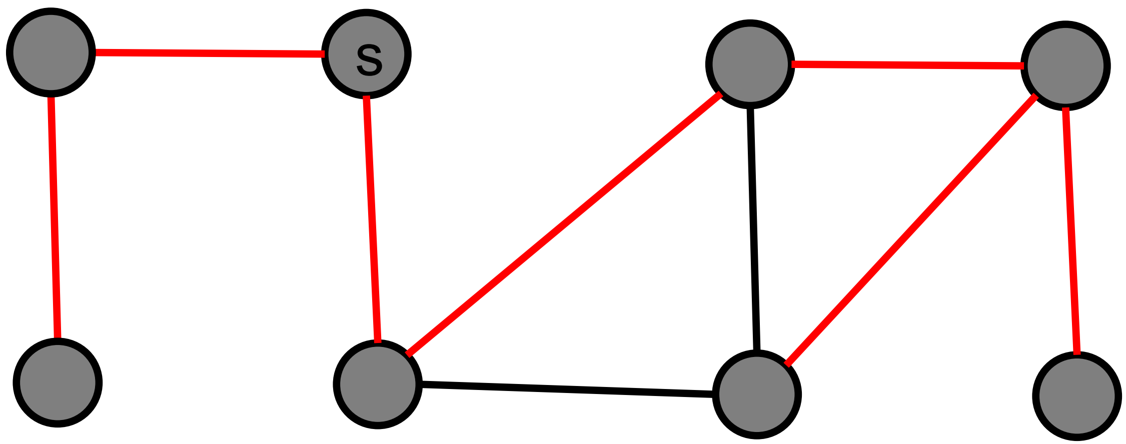
Blue = unvisited

# Depth-First Search Example



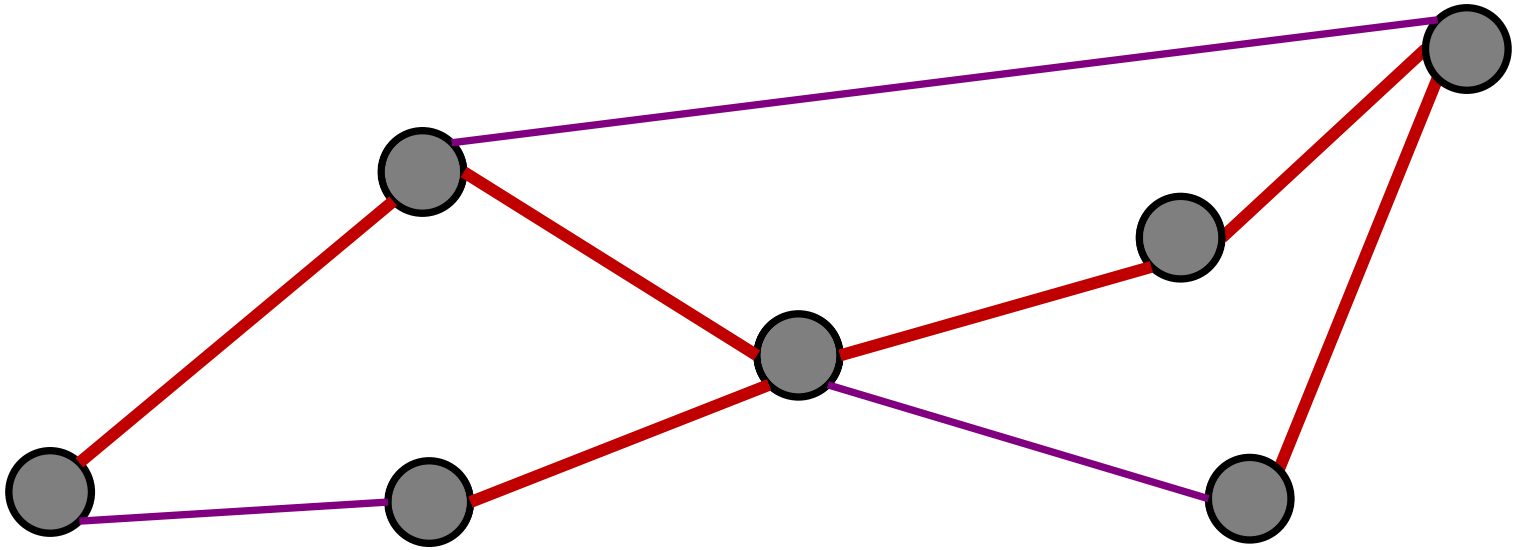

Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# Depth-First Search Example




Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

# DFS parent edges

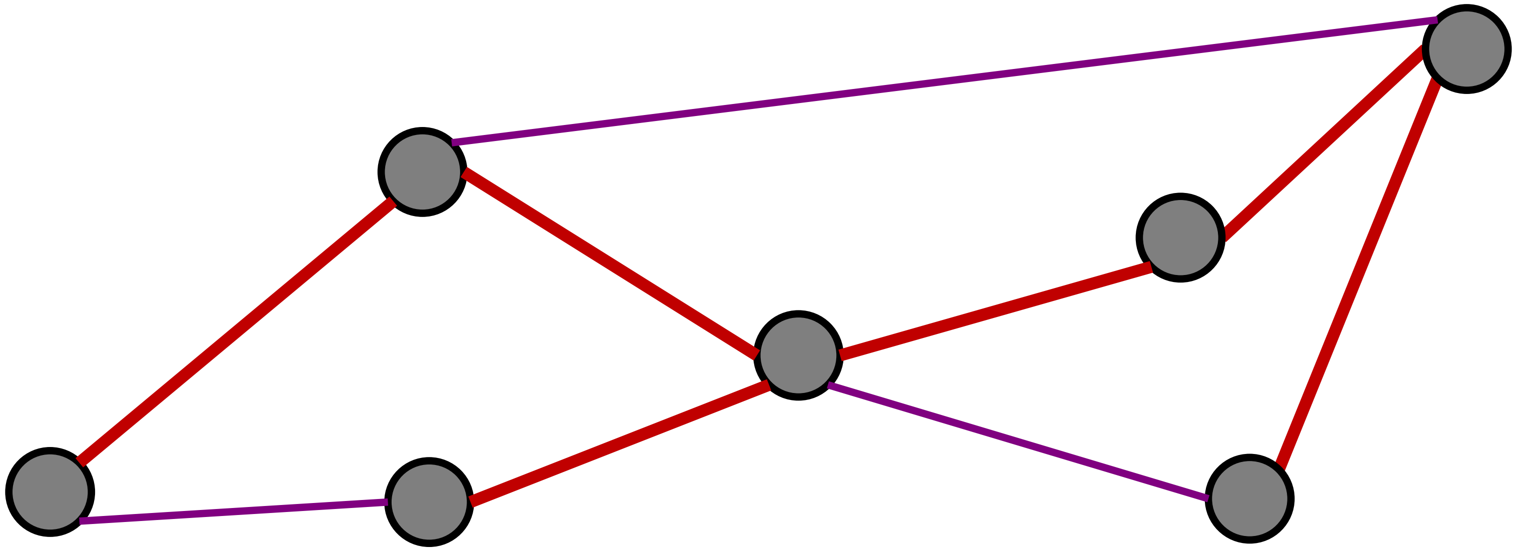


Red = Parent Edges
Purple = Non-parent edges

## Which is true? (More than one may apply.)

1. DFS parent graph is a cycle.
2. DFS parent graph is a tree.
3. DFS parent graph has low-degree.
4. DFS parent graph has low diameter.
5. None of the above.

# DFS parent edges = tree



Red = Parent Edges
Purple = Non-parent edges

**Note: not shortest paths!**

The running time of DFS (using an adjacency list) is:

1. O(V)
2. O(E)
3. O(V+E)
4. O(VE)
5. O($V^2$)
6. I have no idea.

# Depth-First Search

Analysis:

DFS visits each node at most once.

O(V)

DFS enumerates each neighbour at most once.

O(E)

If the graph is stored as an adjacency matrix, what is the running time of DFS?

1. $O(V)$
2. $O(E)$
3. $(V+E)$
4. $O(VE)$
5. $O(V^2)$
6. $O(E^2)$

# Graph Search

BFS and DFS are the similar algorithms:

- BFS: use a queue
  - Every time you visit a node, add all unvisited neighbors to the queue.
- DFS: use a stack
  - Every time you visit a node, add all unvisited neighbors to the stack.

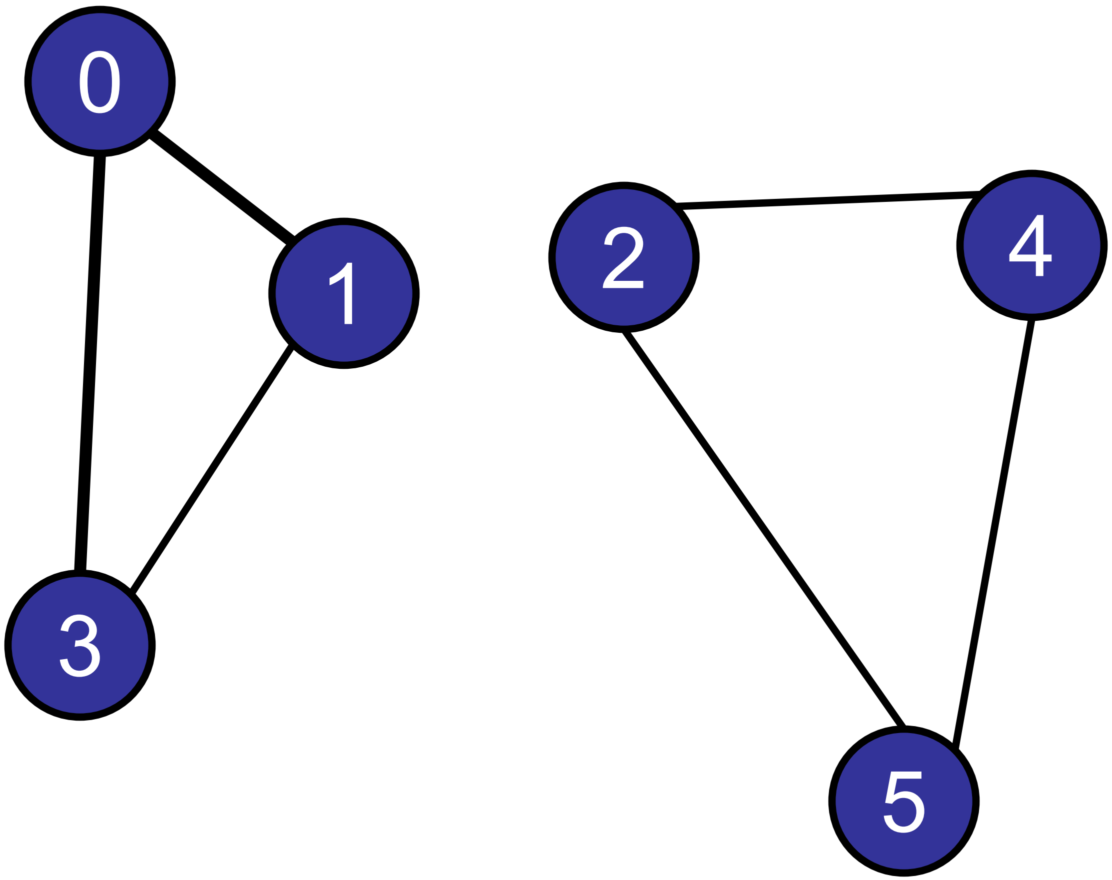
# Handling Disconnected Graphs

How do we visit every node?

How do we count connected components?

# Handling Disconnected Graphs

Example:

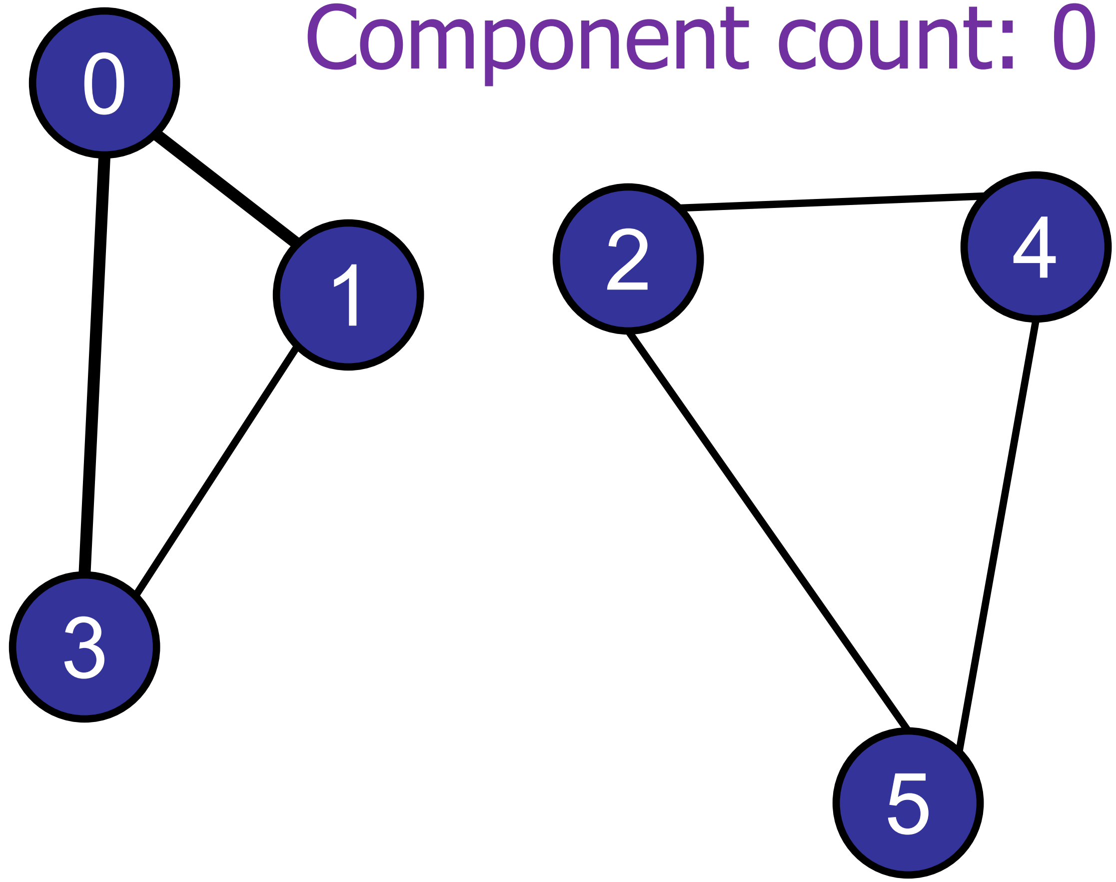


# Handling Disconnected Graphs

Idea:
Try BFS from
node 0 to n - 1

Component count: 0

Trying
node 0




| | | | | | |
|---|---|---|---|---|---|
| false | false | false | false | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

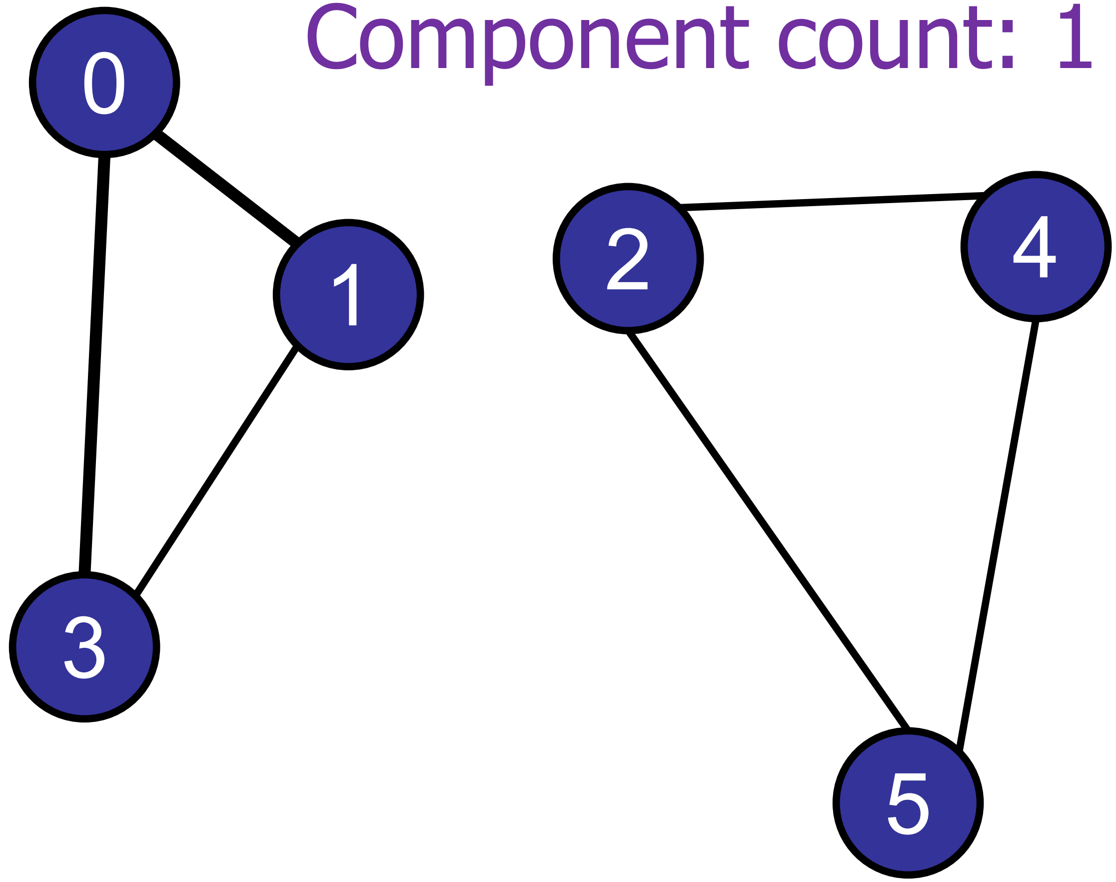
# Handling Disconnected Graphs

Component count: 1

Idea:
Try BFS from
node 0 to n - 1

Trying
node 0

new
component!

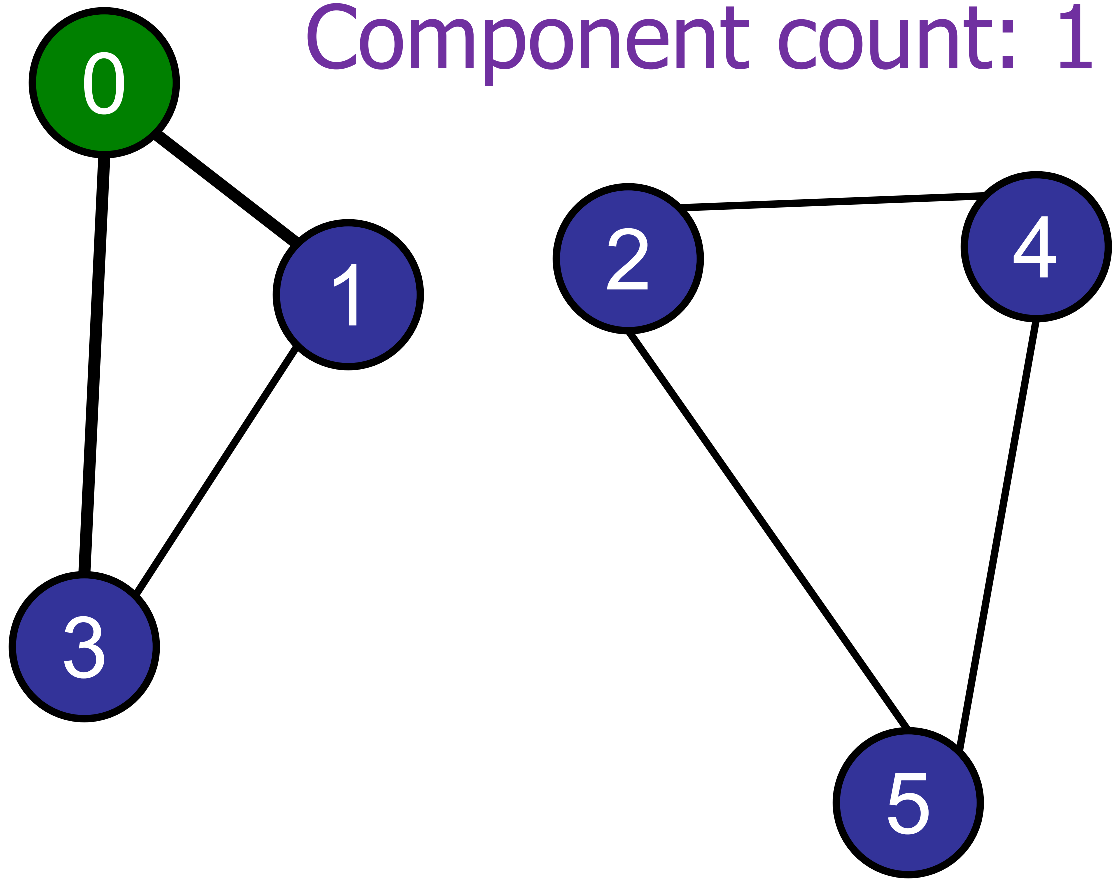



| | | | | | |
|---|---|---|---|---|---|
| false | false | false | false | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Component count: 1

Idea:
Try BFS from
node 0 to n - 1



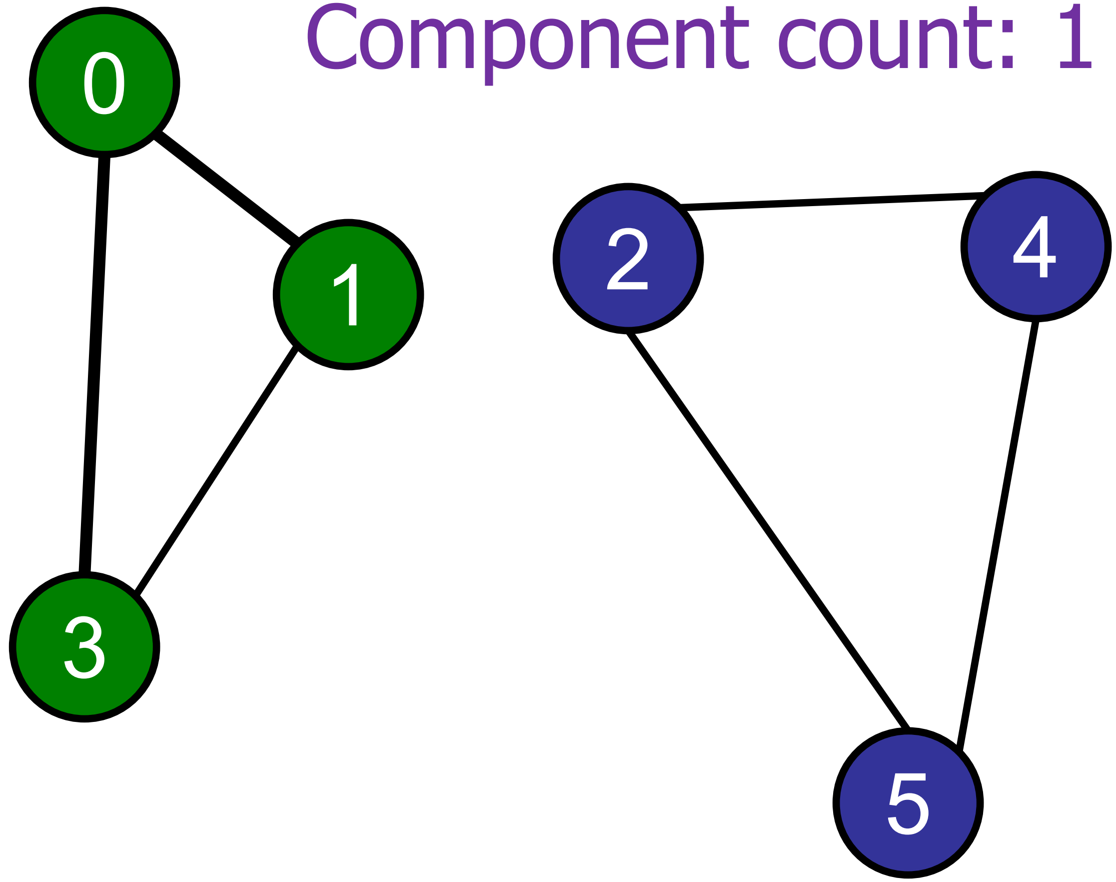

Trying
node 0

| | | | | | |
|---|---|---|---|---|---|
| true | false | false | false | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Component count: 1

Idea:
Try BFS from
  node 0 to n - 1




Trying
node 0

| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | false |

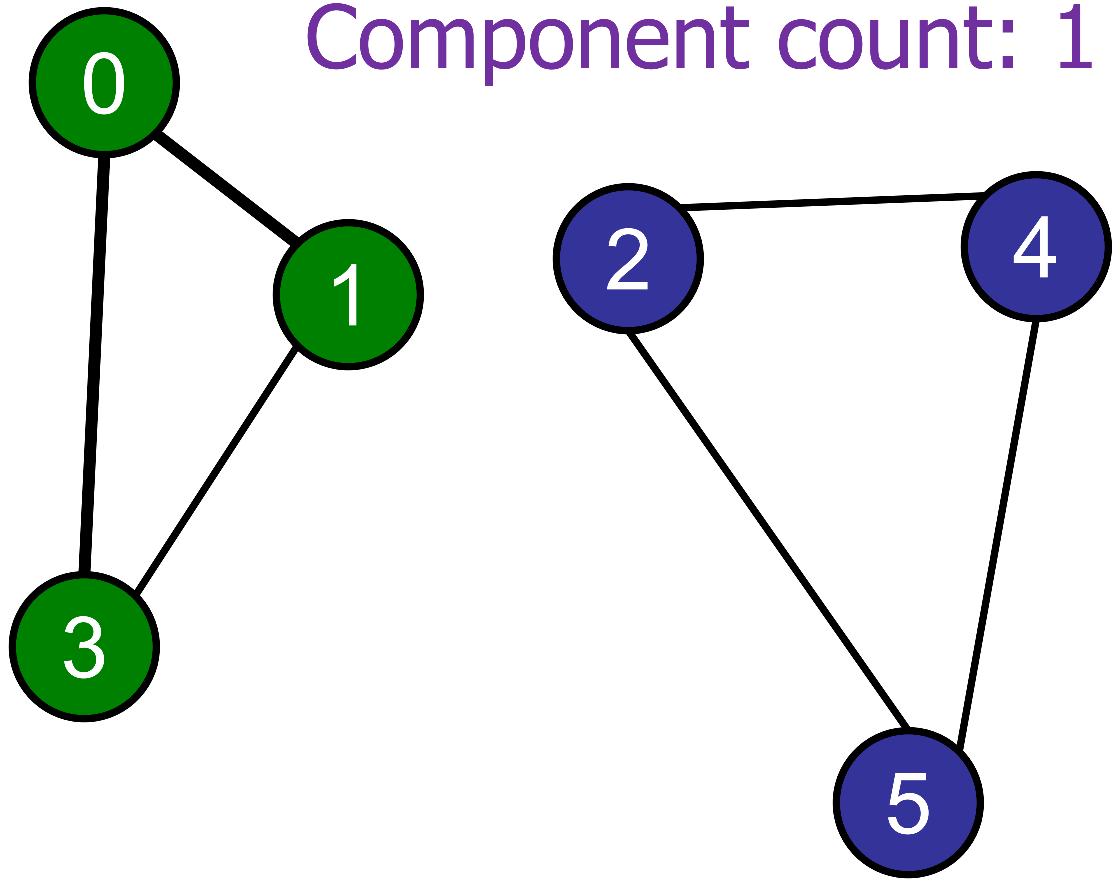
0 1 2 3 4 5

# Handling Disconnected Graphs

Idea:
Try BFS from
  node 0 to n - 1

BFS done!

Component count: 1

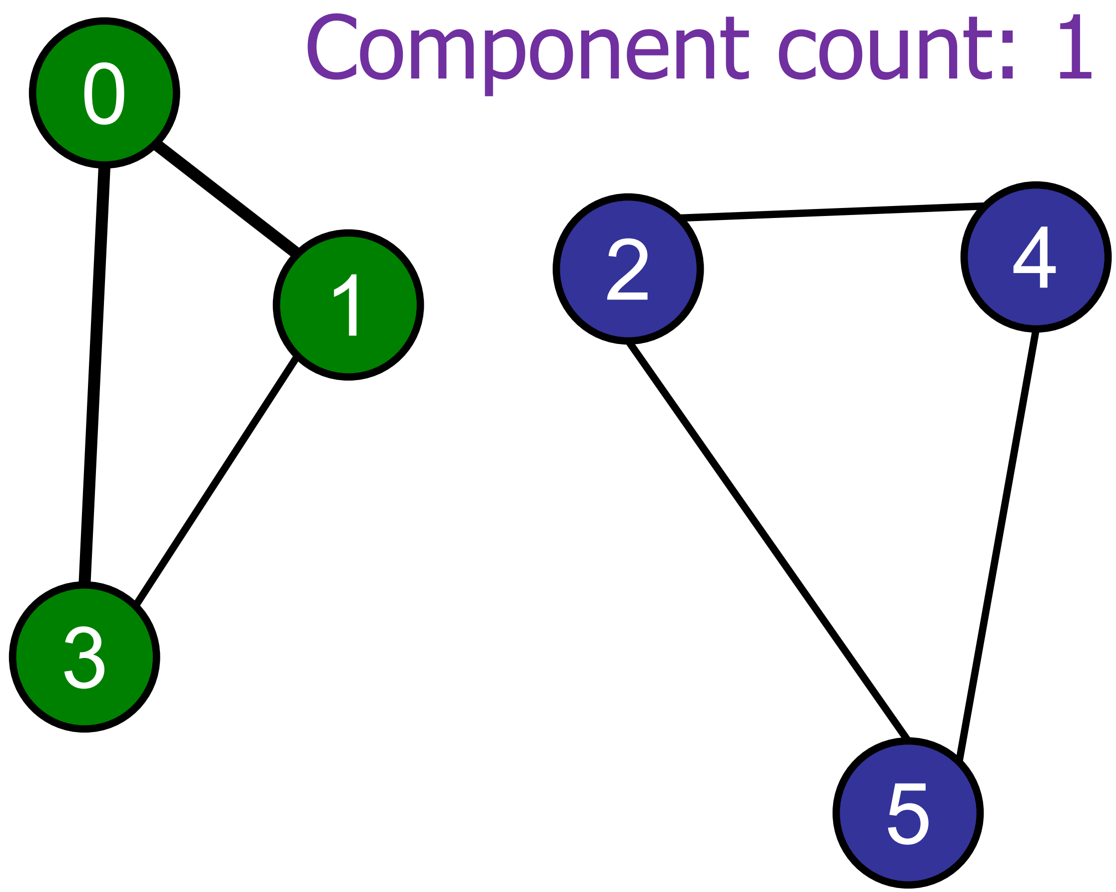



| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from node 0 to n - 1

Trying node 1

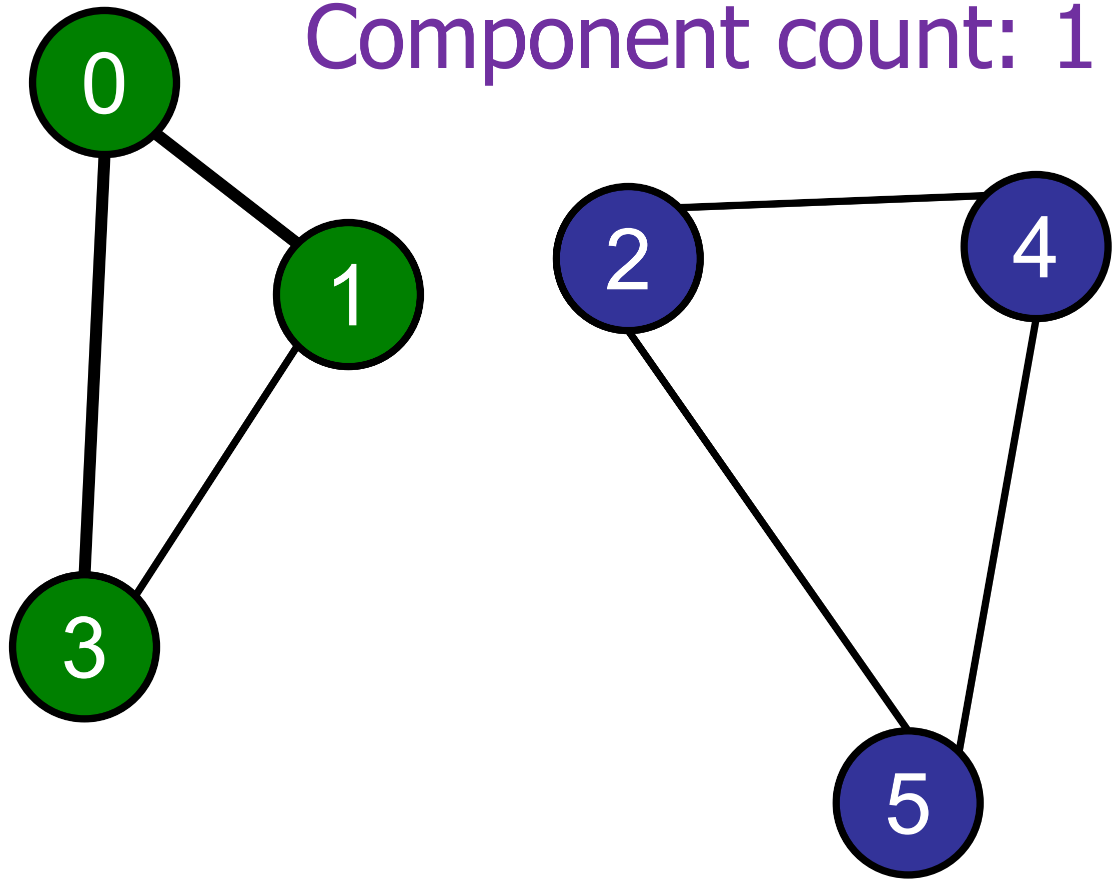



| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from
node 0 to n - 1

Component count: 1




Trying
node 1

Already visited!

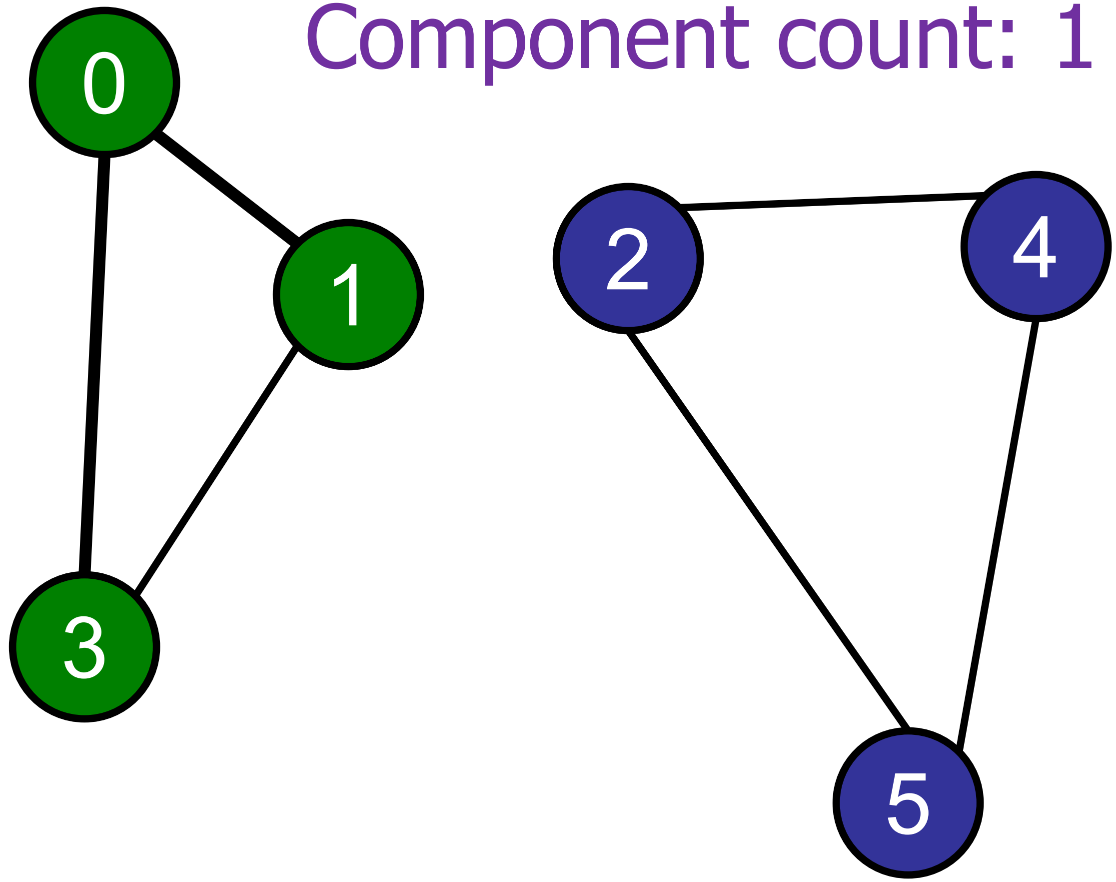
| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from
node 0 to n - 1

Trying
node 2

Component count: 1

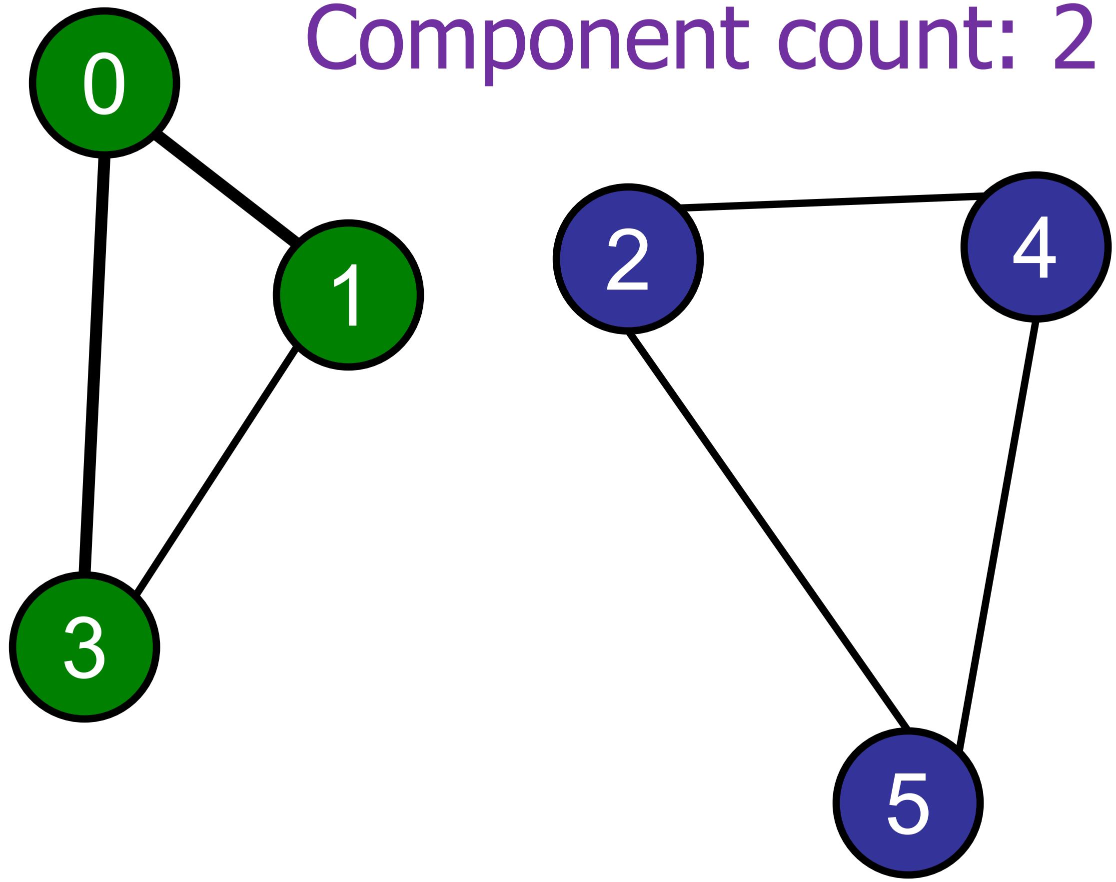



| true | true | false | true | false | false |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from node 0 to n - 1

Component count: 2




Trying node 2

new component!

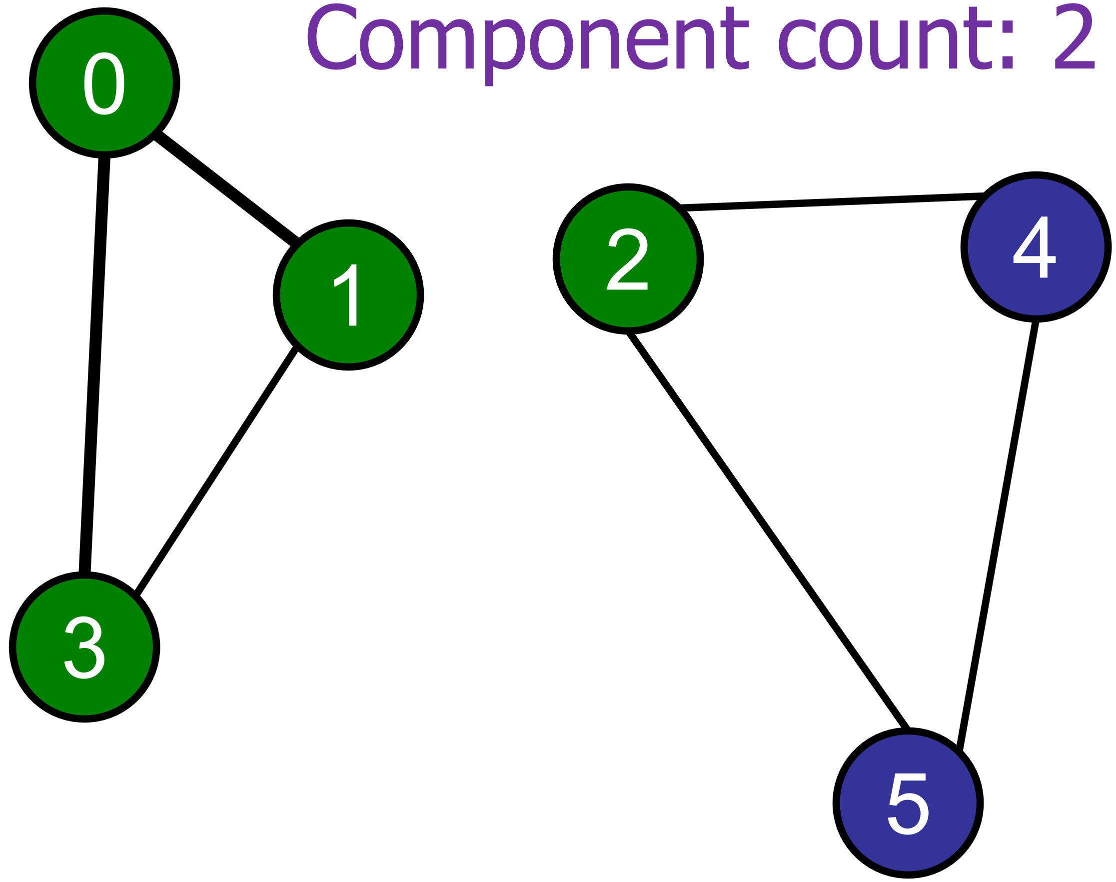
| | | | | | |
|---|---|---|---|---|---|
| true | true | false | true | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Component count: 2

Idea:
Try BFS from
node 0 to n - 1

Trying
node 2

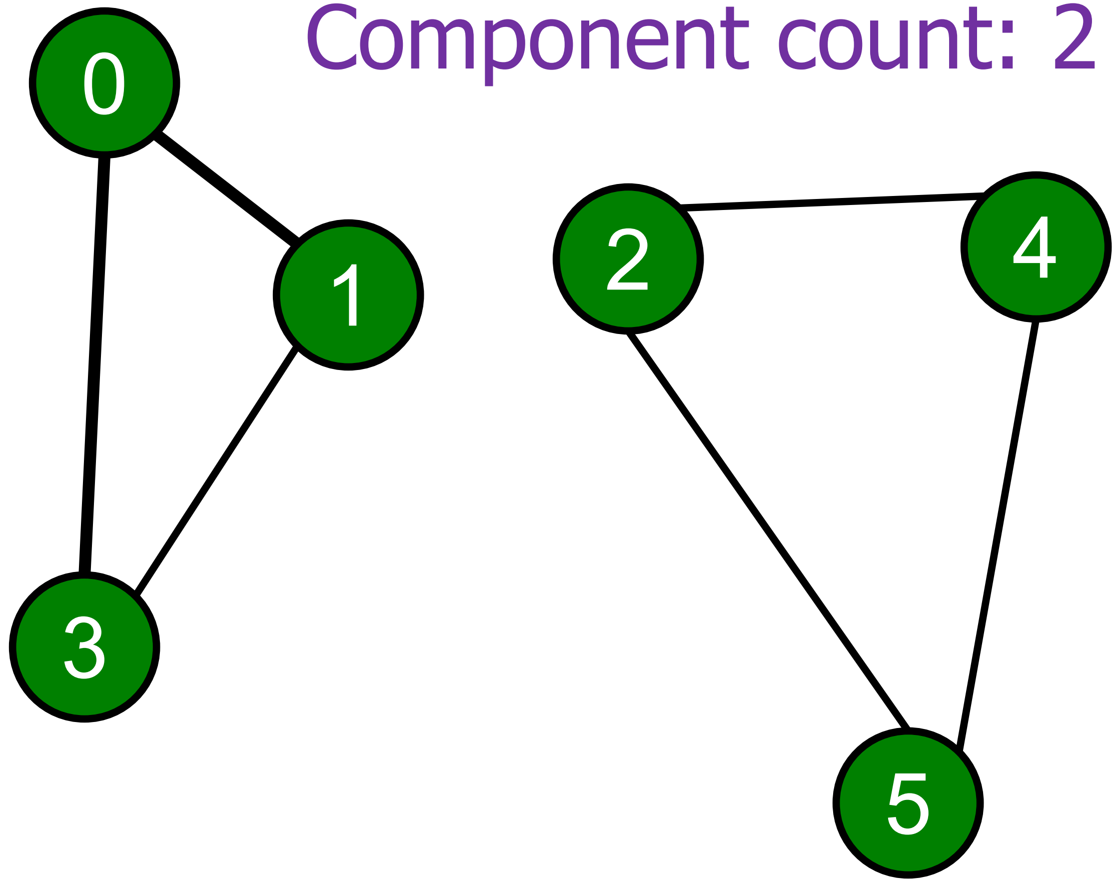



| | | | | | |
|---|---|---|---|---|---|
| true | true | true | true | false | false |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from node 0 to n - 1

Component count: 2



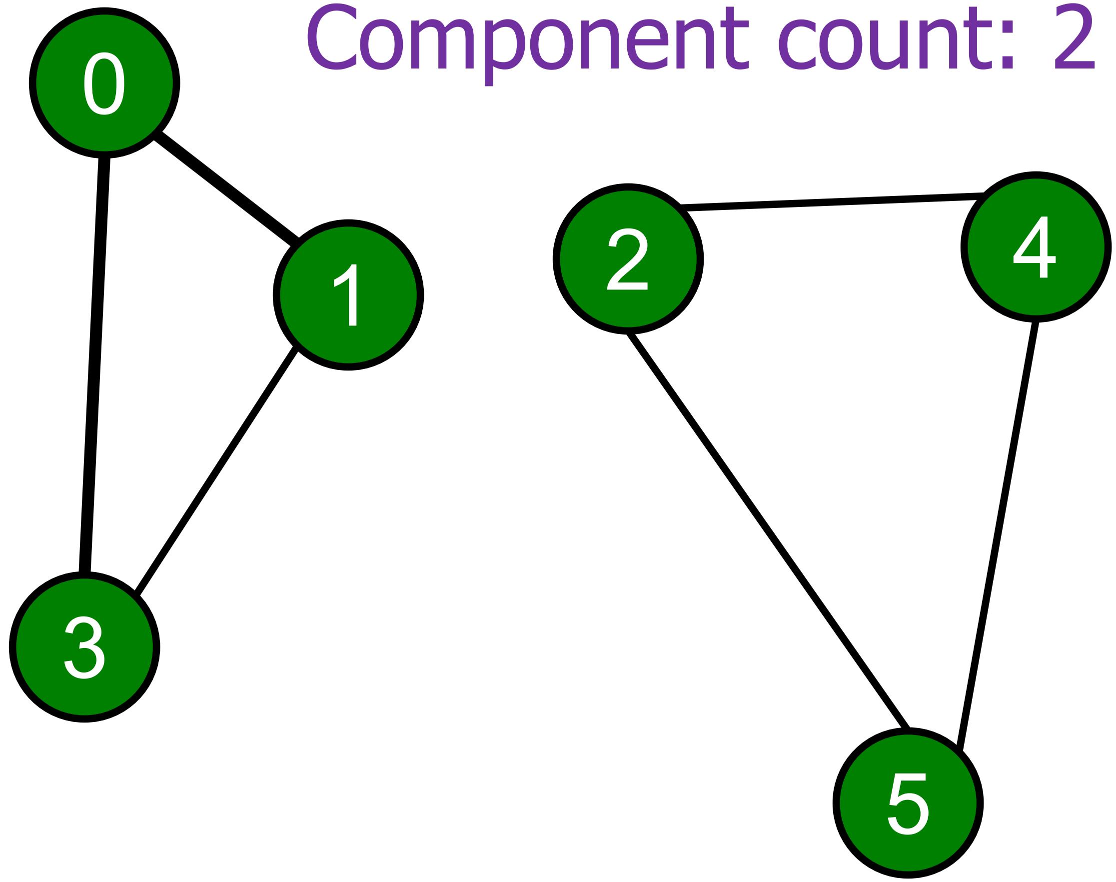

Trying node 2

| | | | | | |
|---|---|---|---|---|---|
| true | true | true | true | true | true |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from
node 0 to n - 1

Component count: 2




Trying
node 3

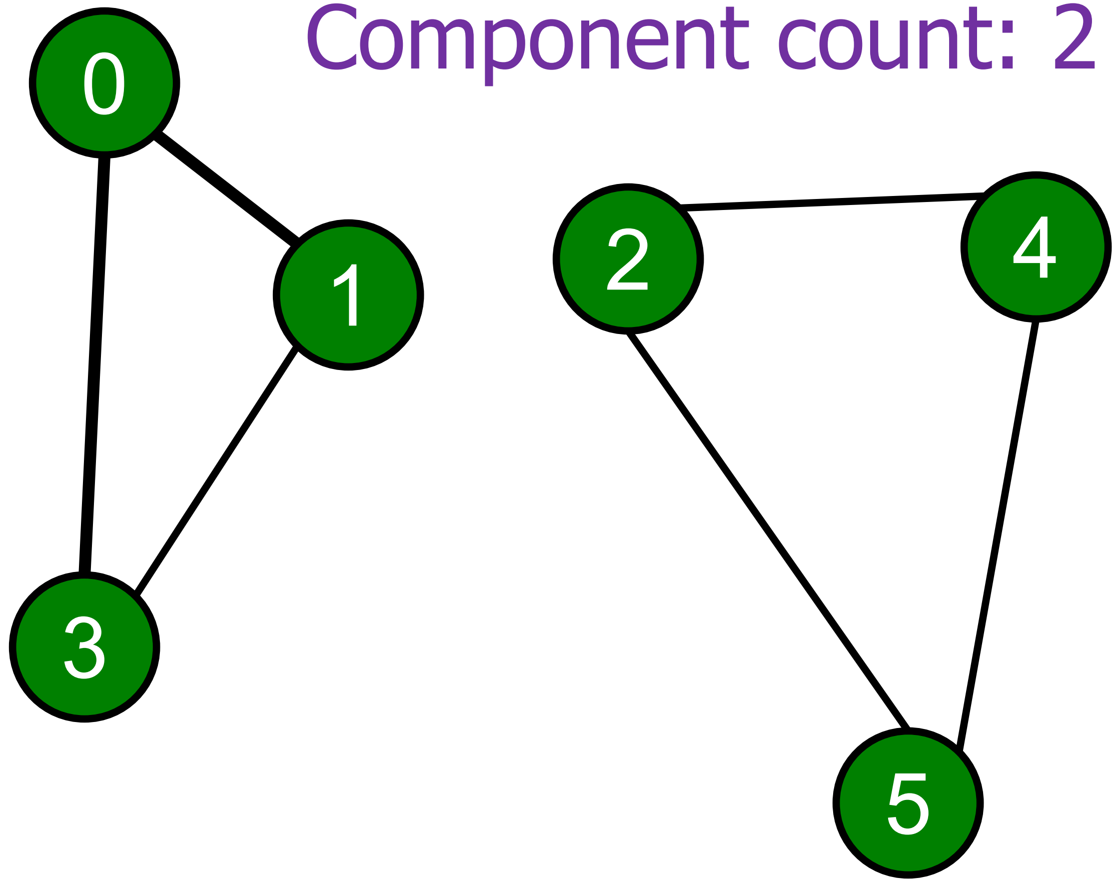
Already visited!

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| true | true | true | true | true | true |

# Handling Disconnected Graphs

Idea:
Try BFS from
 node 0 to n - 1

Component count: 2




Trying
node 4

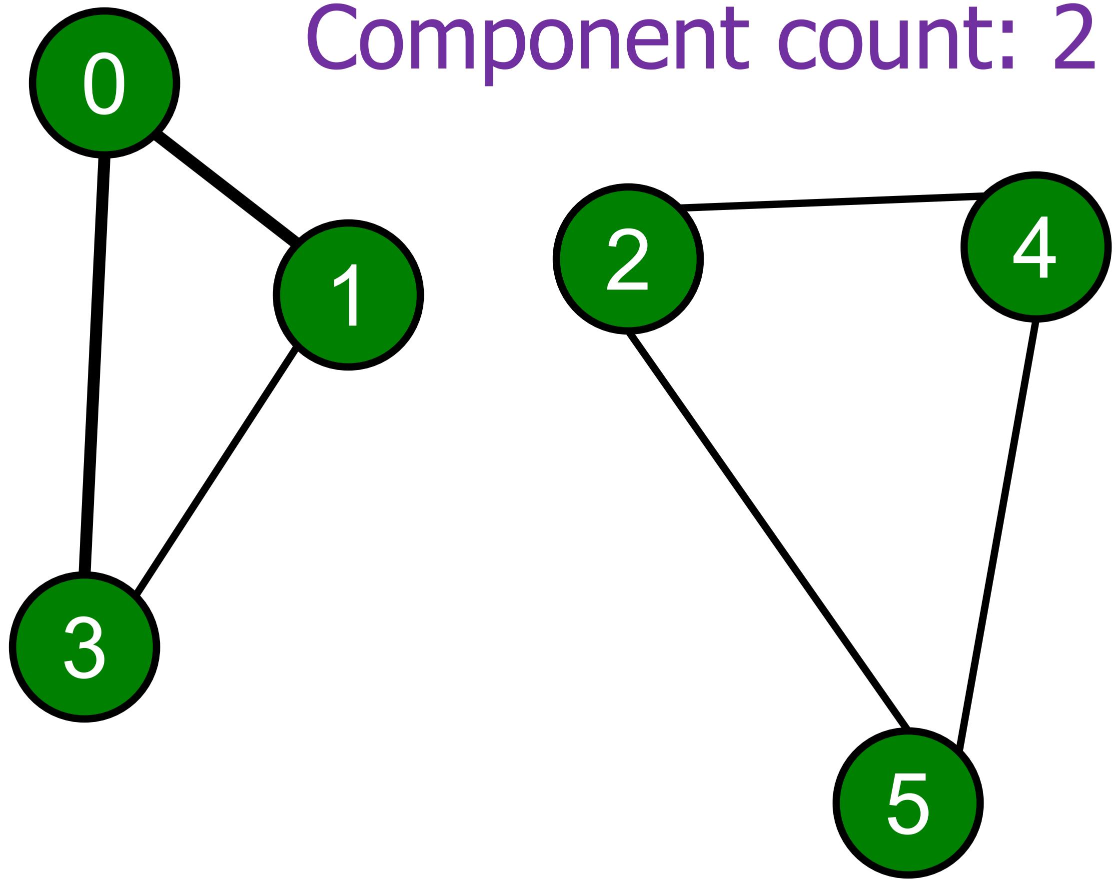
Already visited!

| | | | | | |
|---|---|---|---|---|---|
| true | true | true | true | true | true |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Handling Disconnected Graphs

Idea:
Try BFS from
node 0 to n - 1

Component count: 2




Trying
node 5

Already visited!

| | | | | | |
|---|---|---|---|---|---|
| true | true | true | true | true | true |
| 0 | 1 | 2 | 3 | 4 | 5 |

# Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

Shortest Pathfinding for Unweighted Graphs