SCC.121: Fundamentals of Computer Science Sorting, Trees and Graphs

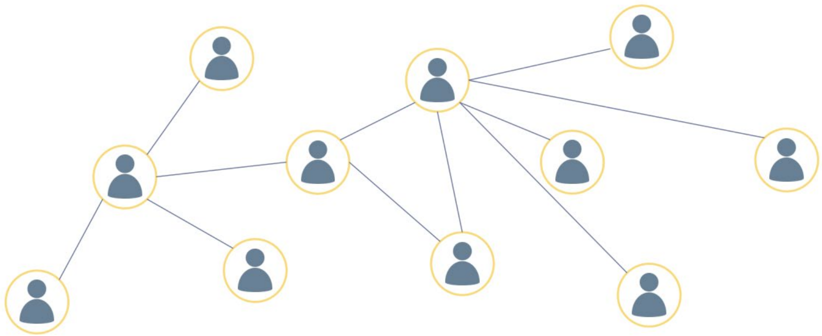
Graphs

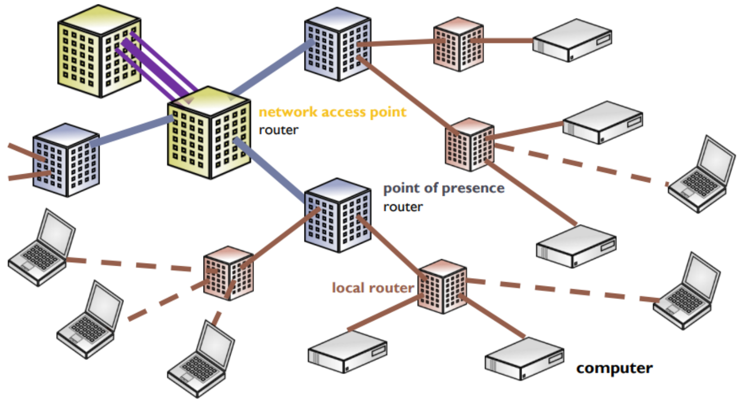
Today’s Lecture

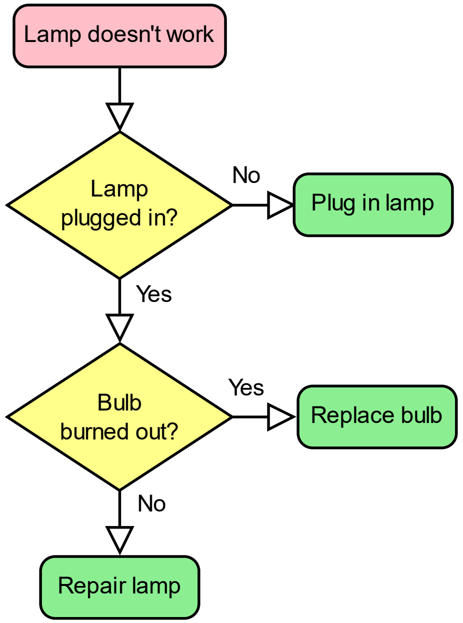
Aim:

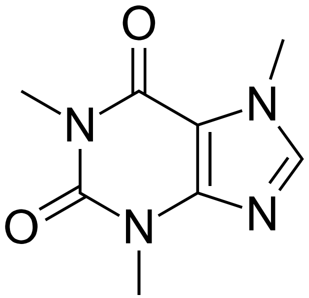
- Introduce **graph** (and ADT),
- Describe graph properties,
- Implement graph ADT,
- DFS and BFS traversals on graphs.

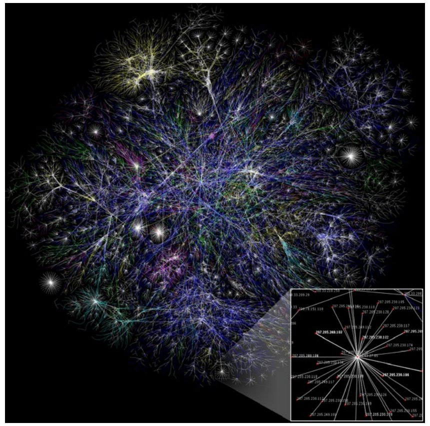
Graphs in the Real World

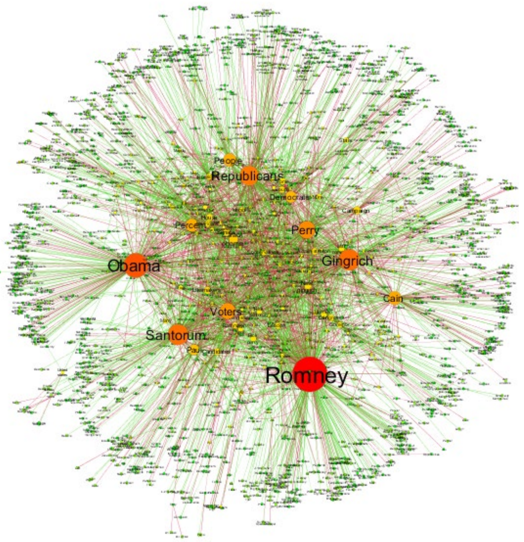






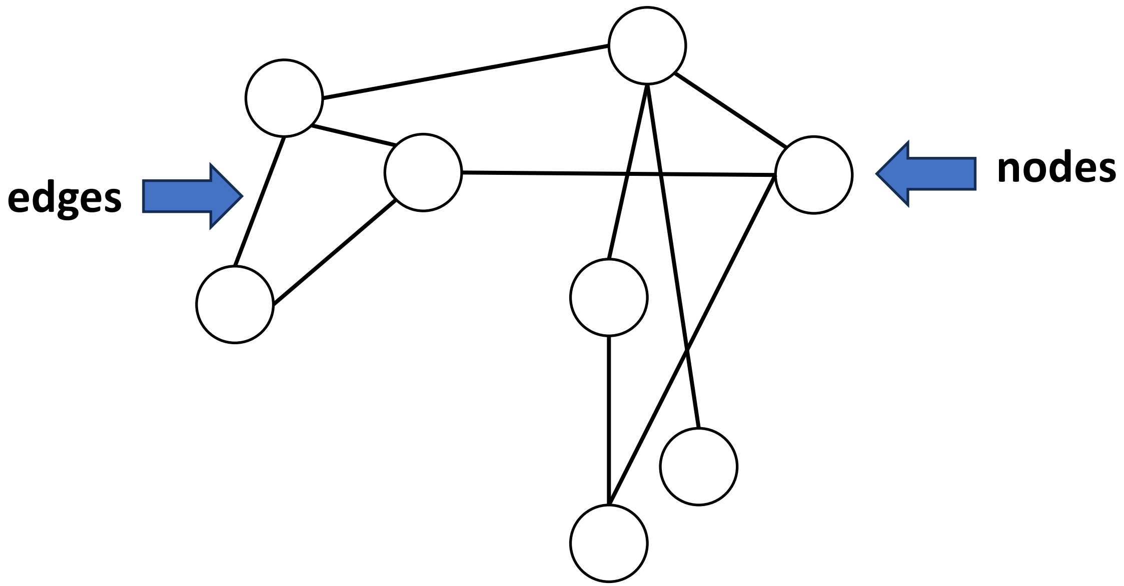






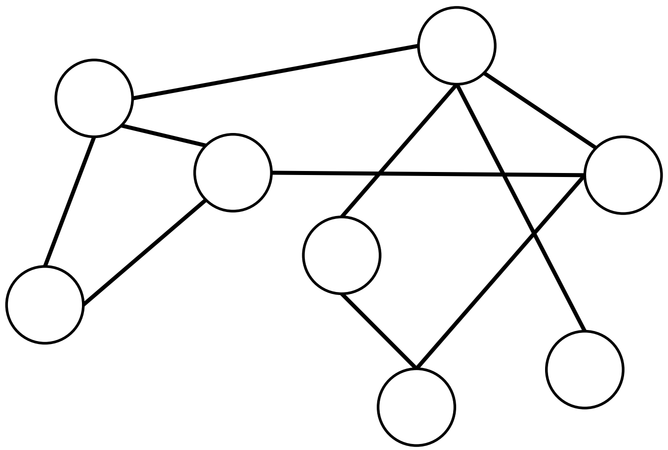
Graphs: Basic Definitions

- **Graph** = Set of **nodes (or vertices)** and **edges (or arcs or links)**

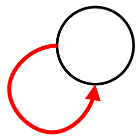


Graphs: Basic Definitions

- **Simple graph**: no self-loop



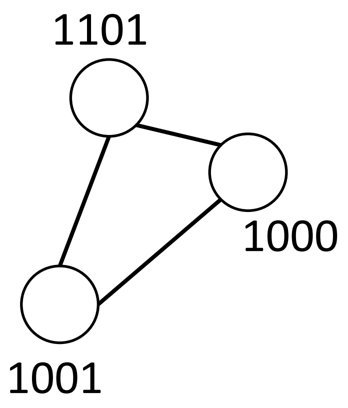
- If a graph has self-loops, then some of the graph analysis calculations change.

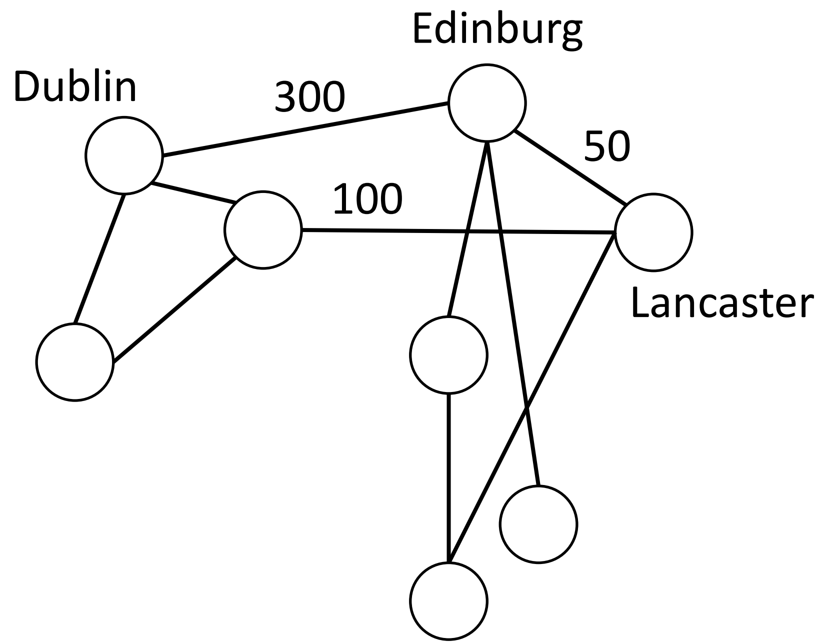


- We will assume simple graphs.

Graphs: Basic Definitions

- Both nodes and edges can have values associated to them (also called **labels**).
- Labels can be integers, string, etc…

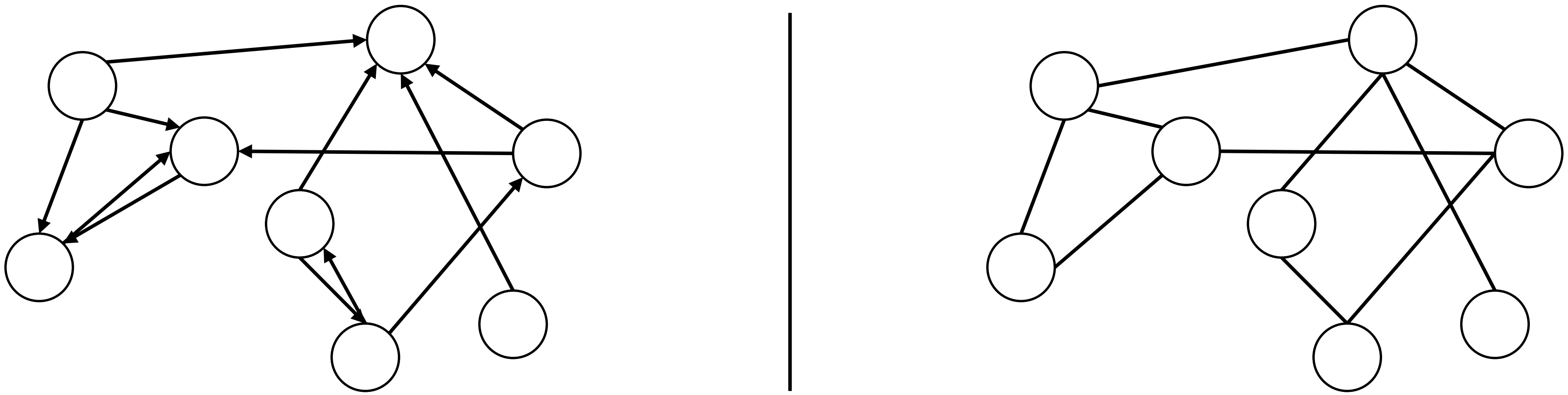




- **Natural setting**:
  - Integers for edges (called weights),
  - Names for nodes (called identifiers).

Graphs: Basic Definitions

- Graph can be **directed** or **undirected**

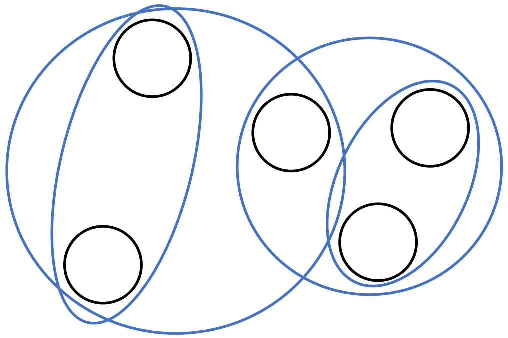


- Undirected edges can be traversed in both directions,
- Whereas directed edges can only be traversed in the indicated direction.

Why care about graphs?

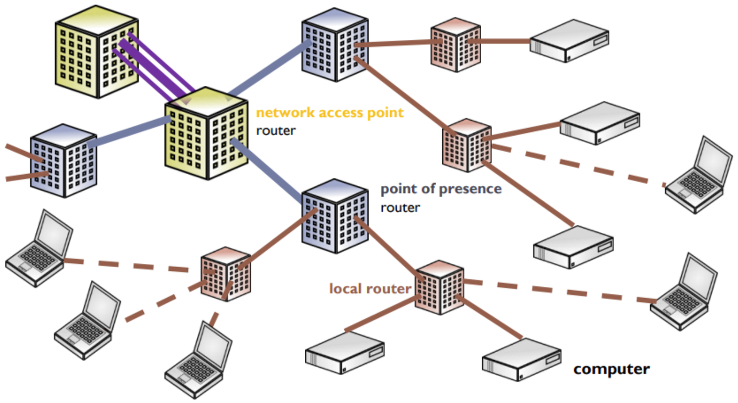
- **Graph structures are ubiquitous** in computer systems and programming
  - Because they embody the **concepts** of "**entities**" and "**relationships**"
  - Where relationships = **pairwise relationships**

- Hypergraphs (nodes & hyperedges) capture ***k*-wise relationships** for $k > 2$



Graphs in Networking

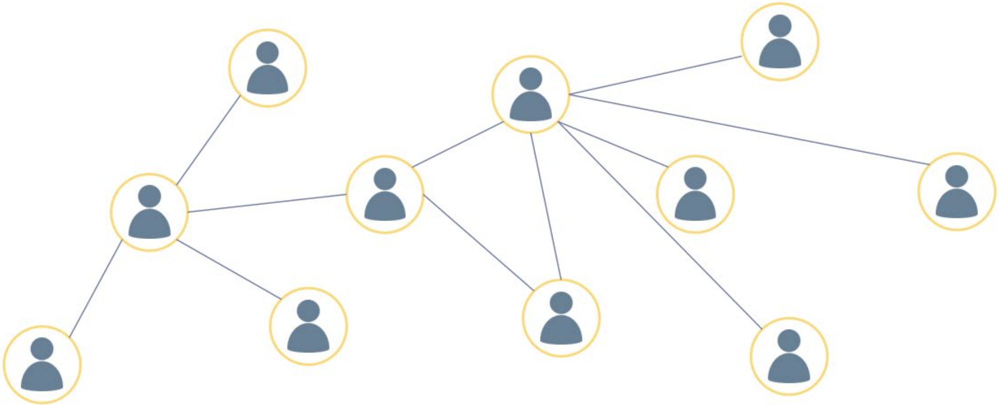
- The **internet** is organized as a graph, with a **core** of very high performance routers but increasingly less costly routers towards the "periphery"



- **Efficiently routing packets** around the Internet is done through a "distributed algorithm"

Graphs in Social Media

- **Social networks** like Facebook and Twitter maintain graphs of users and their connections (“friends”/”follow”)



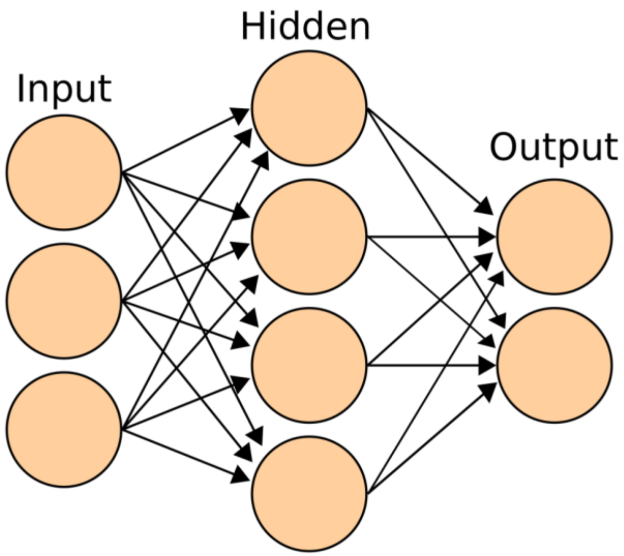
- **Graph analysis:**
  - How to visualize these social networks?
  - Do they have strong **cores** (strong communities)?
  - How far apart are any two nodes (e.g., **6 degrees of separation**)?

Graphs in Public Transport

- **Transit systems** such as air travel routes or city metros are often represented as a graph, with many advantages:
    1. Clear **visualization**
    2. Can compute **minimum time (or congestion) itineraries**
    3. Can compute **whether the network is “efficient”** (Steiner trees)

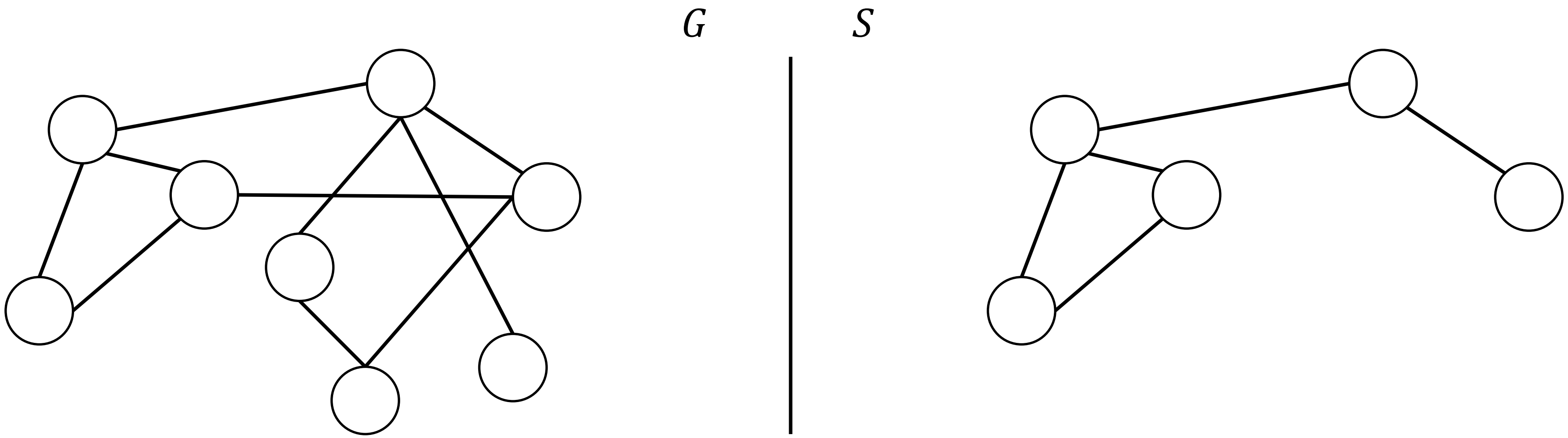
Examples of Graph Usages

- **Artificial Neural Networks** are directed graphs with "**layers**" of nodes
  - Weighted edges,
  - Edges and weights decide how input transforms into output.
  - If output is "wrong", update weights.



Graphs: Subgraphs

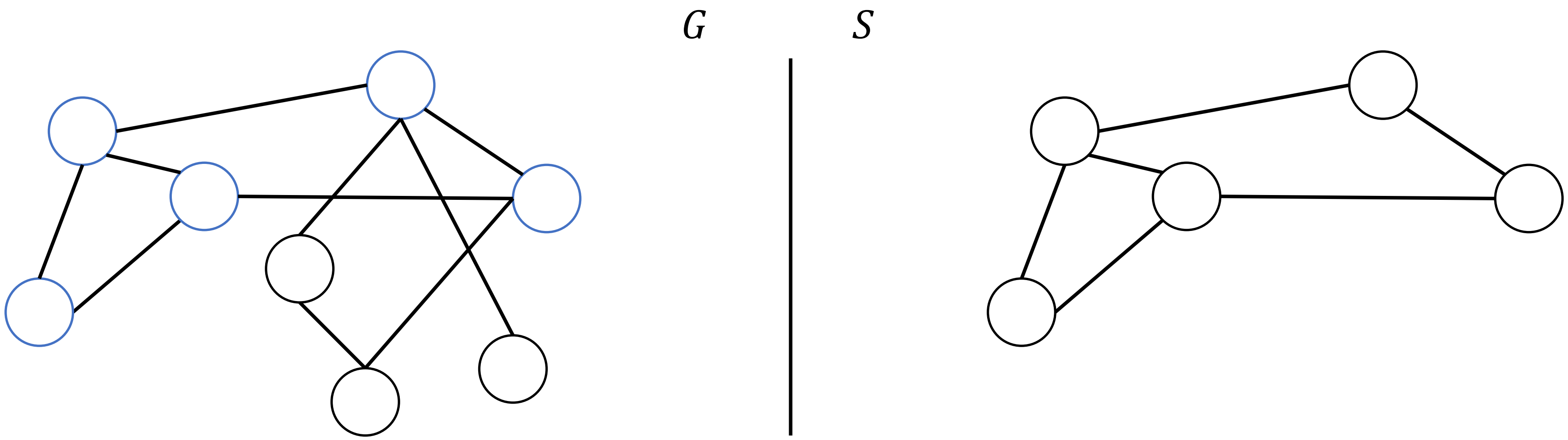
- The **subgraph** S of some graph G is a subset of the nodes and edges of G



- For both undirected and directed graph.

Graphs: Induced Subgraphs

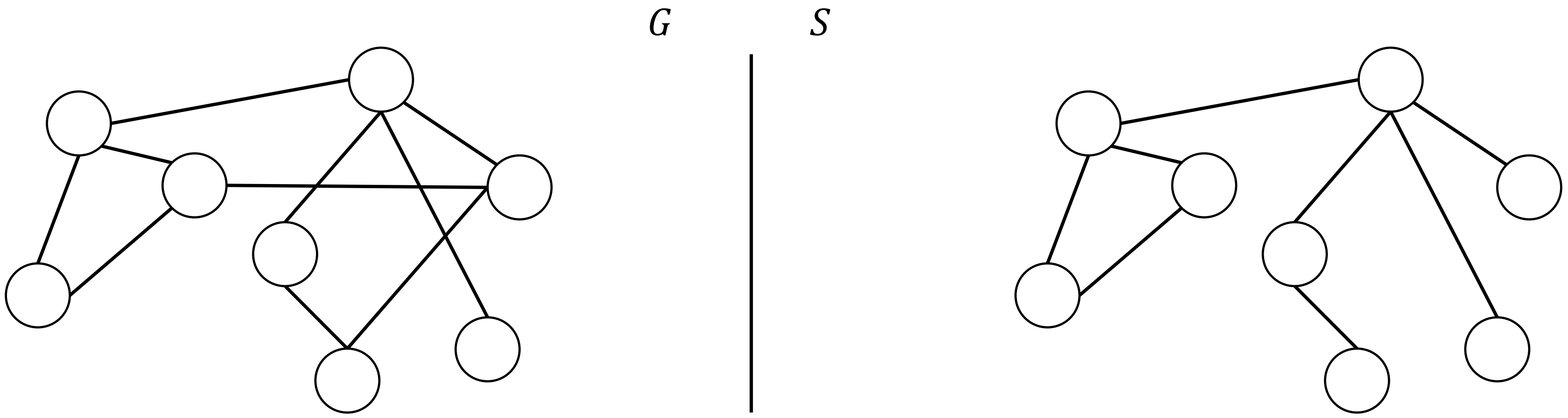
- S is the **subgraph induced** by subset $U \subseteq V$ if its nodes are U and its edges are all the edges between nodes in U within G



- For both undirected and directed graph

Graphs: Spanning Subgraphs and Trees

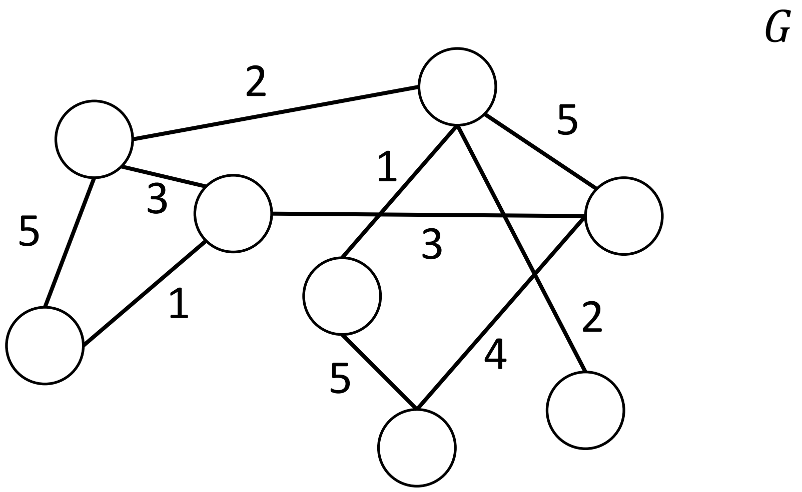
- S is a **spanning subgraph** of G if it is formed from **all nodes of G** and some of the edges of G.

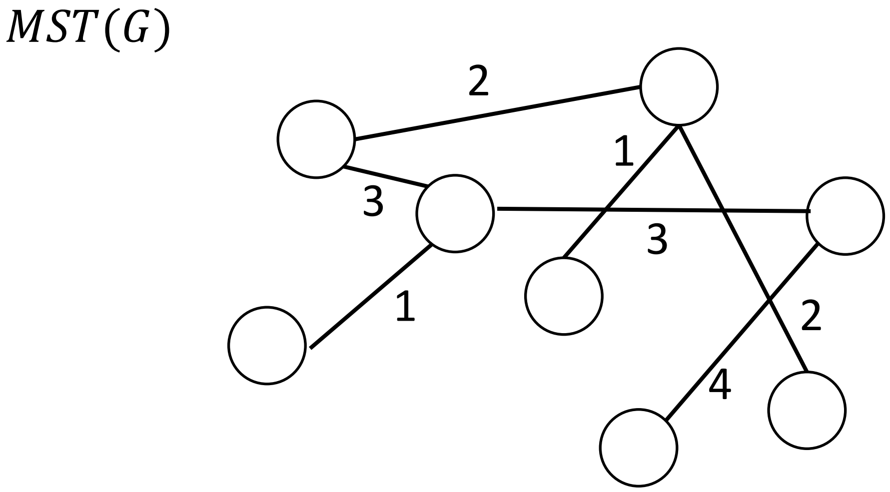


- **Spanning tree** = spanning subgraph & tree

Graphs: Minimum Spanning Tree

- **Minimum spanning tree** = Spanning tree with **minimum total sum of edge weights**

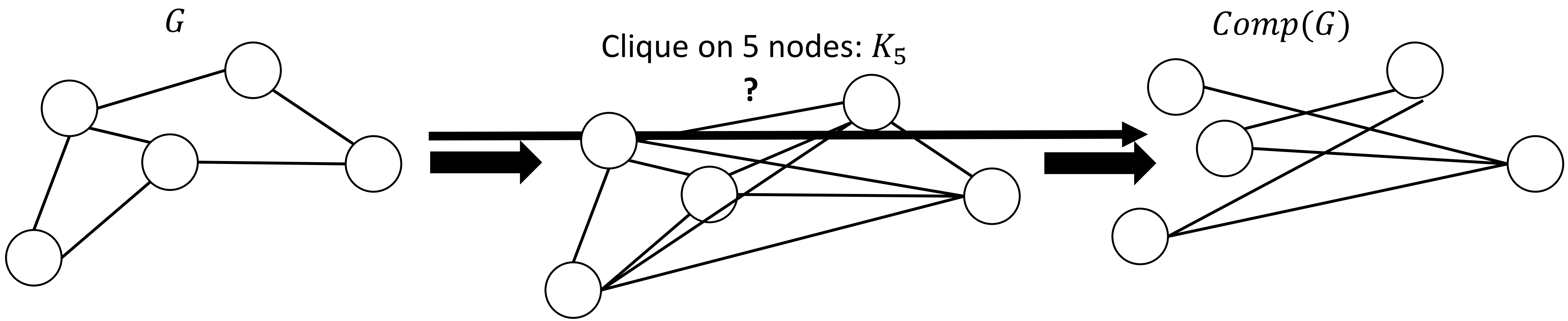




$Weight\_MST(G)$ = 16

Graphs: Complementary Graph

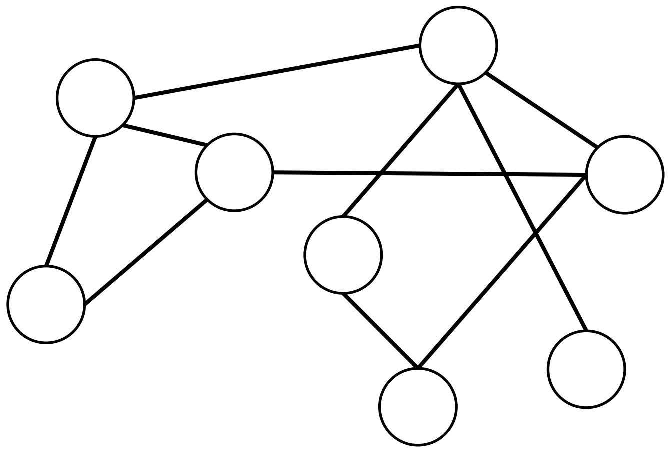
- **Complementary graph** = Same set of nodes and contains all edges that are not in G



- Create a clique (fully-connected graph) and remove the edges that are in G

Graphs: Density

- **Density** of a graph is a measure between 0 and 1 that indicates how heavily connected its nodes are
  - $\frac{2|E|}{n\,(n-1)}$ for undirected,
  - $\frac{|E|}{n\,(n-1)}$ for directed

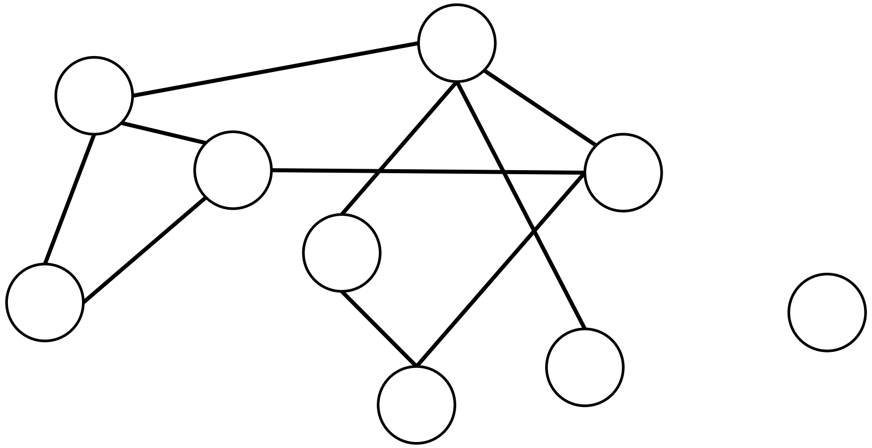


- In example, density is $\frac{20}{56} \approx 0.36$

- In practice, **more complex density measures =** variations of local density

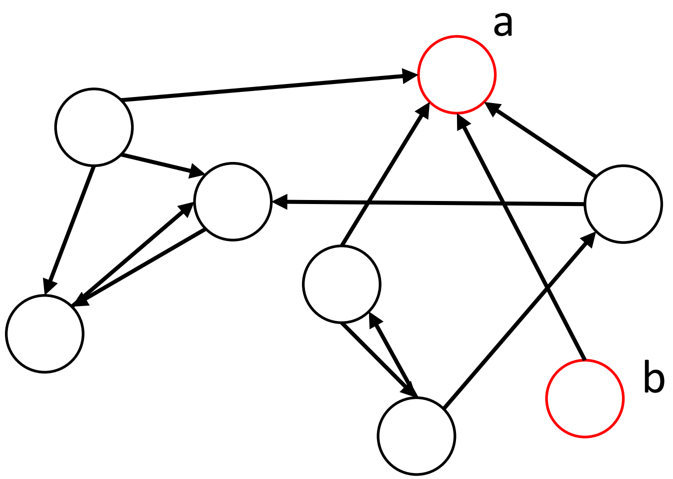
Graphs: Connectivity in Undirected Graphs

- An undirected graph is **connected** if there is a path between any two vertices

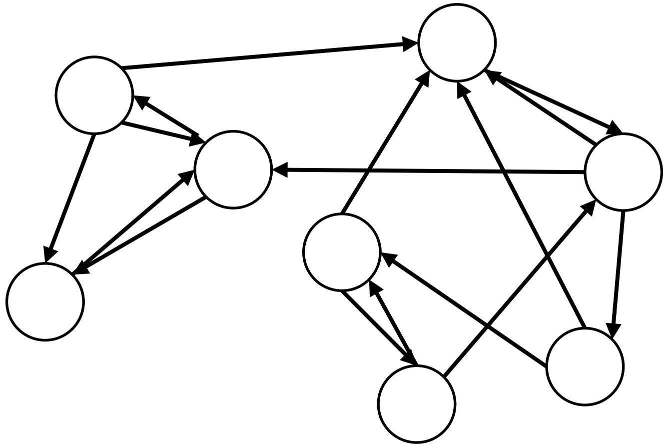


Graphs: Connectivity in Directed Graphs

- Directed graph is **strongly connected** if we can reach any node from any other



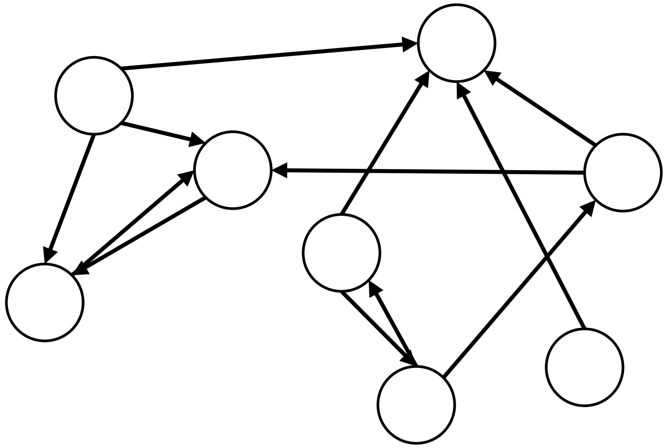
Not strongly connected



strongly connected

Graphs: Connectivity in Directed Graphs

- Directed graph is **weakly connected** if its undirected version is connected.



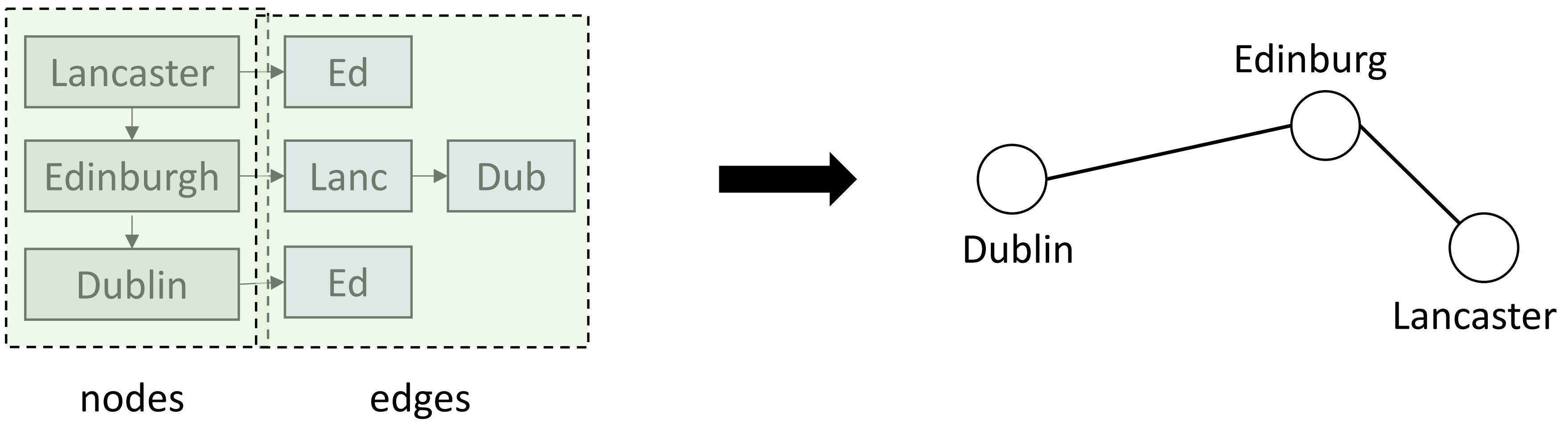
Graph Abstract Data Type (ADT)

- **Common set of operations** for graphs,
- But applicable to **many different settings** (networking, social media, public transport, artificial intelligence,…)

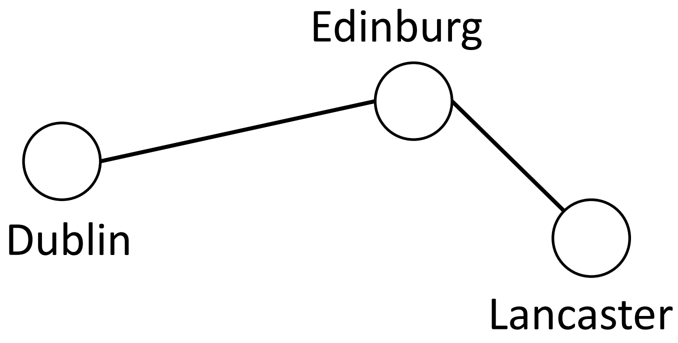
- **Basic operations** = node and edge additions/removals, and adjacency queries:
  - void **addNode**(Node n)
  - void **removeNode**(Node n)
  - void **addEdge**(Node n, Node m)
  - void **removeEdge**(Node n, Node m)
  - boolean **adjacent**(Node n, Node m)
  - Node[] **getNeighbours**(Node n)

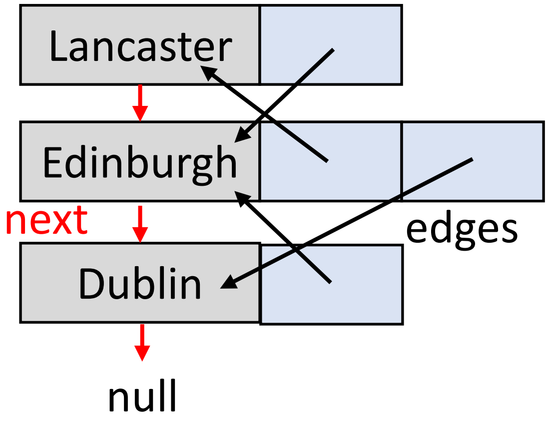
Graph Encoding via lists

- More precisely, the graph is a **list of lists**



List-based Implementation

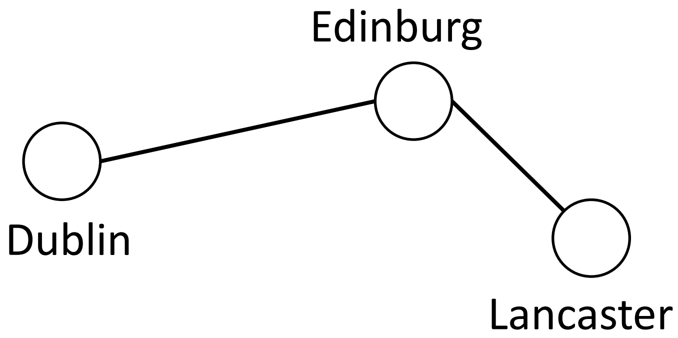


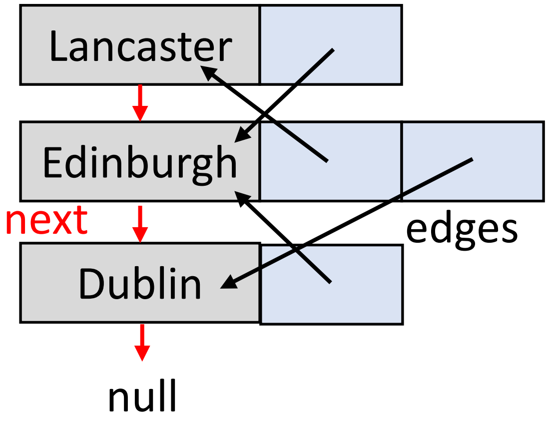



```
public class GraphList<Label>{
    GraphNode headNode;

    public class GraphNode<Label>{
        Label id;
        LinkedHashSet<GraphNode> edges;
        GraphNode next;

        public GraphNode(Label label){
            this.id = label;
            this.edges = new LinkedHashSet<GraphNode>();
        }
    }

    public GraphList(){
    }
…
```

List-based: Adding and finding nodes

Lancaster University

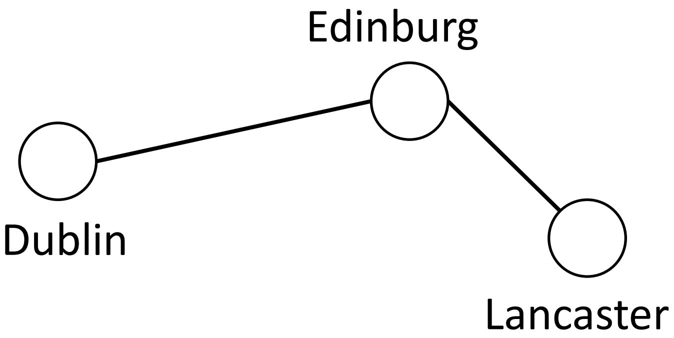


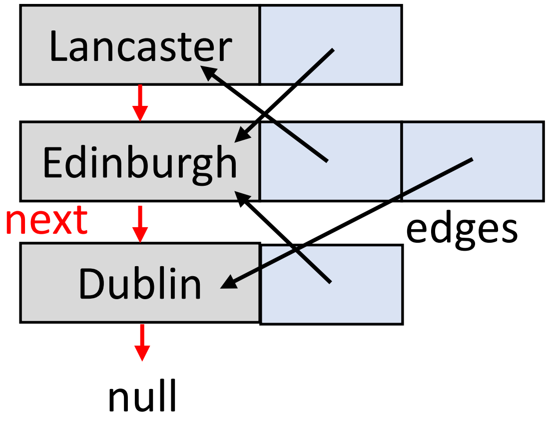



```java
public void addNode(Label label){
    GraphNode node = new GraphNode(label);
    node.next = headNode;
    headNode = node;
}
```

```java
private GraphNode findNode(Label label){
    GraphNode node = headNode;
    while(node != null){
        if (node.id == label) return node;
        node = node.next;
    }
    return null;
}
```

List-based: Adding and removing edges

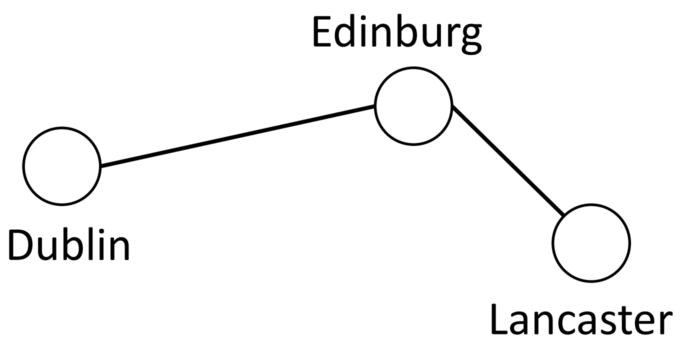


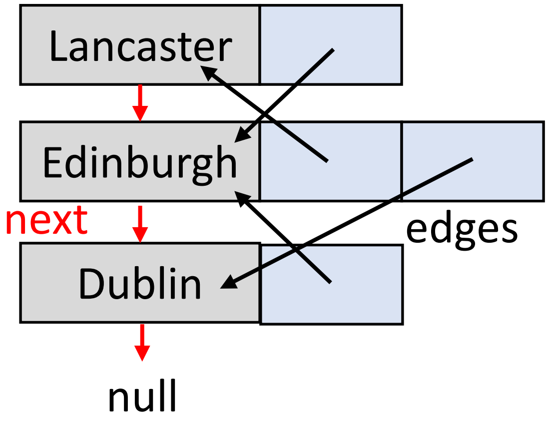



```java
public void addEdge(Label l1, Label l2){
    GraphNode node1 = findNode(l1);
    GraphNode node2 = findNode(l2);
    if(node1 != null && node2 != null){
        node1.edges.add(node2);
        node2.edges.add(node1);
    }
}
```

```java
public void removeEdge(Label l1, Label l2){
    GraphNode node1 = findNode(l1);
    GraphNode node2 = findNode(l2);
    if(node1 != null && node2 != null){
        node1.edges.remove(node2);
        node2.edges.remove(node1);
    }
}
```

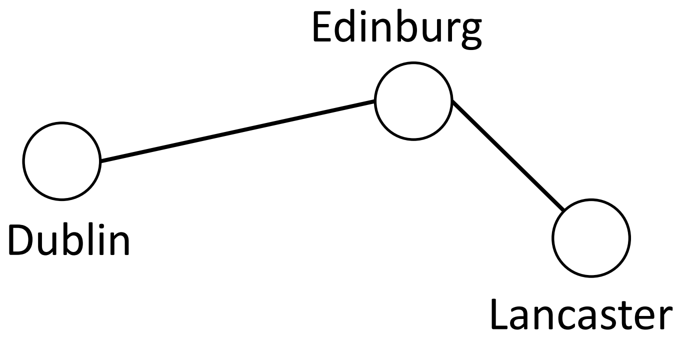
List-based: Removing a node

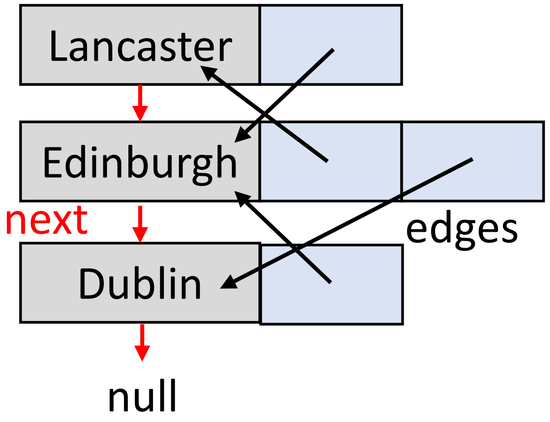





```
public void removeNode(Label label){
    GraphNode<Label> node = headNode;
    GraphNode<Label> prevNode = null;
    while(node != null){
        if (node.id == label) {
            // Remove all edges
            for(GraphNode<Label> neighbor: node.edges){
                removeEdge(node.id, neighbor.id);
            }
            // Now remove node from the "list"
            if (prevNode != null) prevNode.next = node.next;
            else headNode = node.next;
        }
        prevNode = node;
        node = node.next;
    }
}
```

List-based: Neighborhood and adjacency

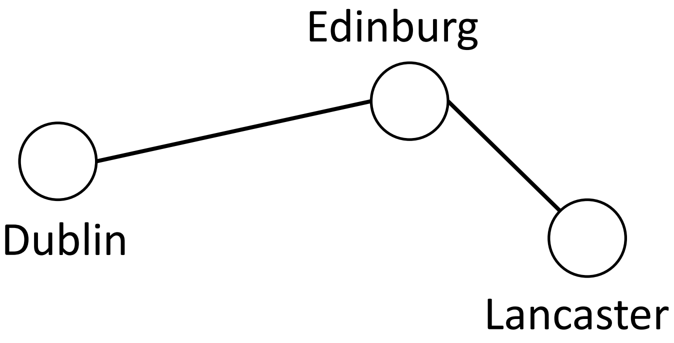


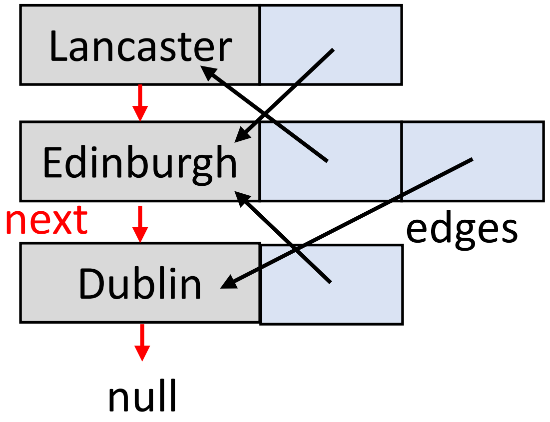



```java
public GraphNode[] getNeighbors(Label label){
    GraphNode<Label> node = findNode(label);
    GraphNode[] neighbors = node.edges.toArray(new
                                GraphNode[0]);
    return neighbors;
}
```

```java
public boolean adjacent(Label label1, Label label2){
    GraphNode node1 = findNode(label1);
    GraphNode node2 = findNode(label2);
    return node1.edges.contains(node2);
}
```

List-based: Displaying the graph






```java
public void print(){
    GraphNode<Label> node = headNode;
    while(node != null){
        System.out.print("[ " + node.id + " : ");
        for(GraphNode<Label> neighbor: node.edges){
            System.out.print(neighbor.id + " ");
        }
        System.out.println("]");
        node = node.next;
    }
}
```

```
[ Dublin : Edinburgh ]
[ Edinburgh : Lancaster Dublin ]
[ Lancaster : Edinburgh ]
```

Graph ADT: Memory and time complexity

1. **Memory** $= O(N+E)$ for graphs with N nodes and E edges (memory-efficient)

2. **Worst-case time complexity**:
   - Adding node is $O(1)$,
   - Removing and finding node is $O(N)$,
   - Adding and removing an edge is $O(N)$,
   - Adjacency check is $O(N)$.

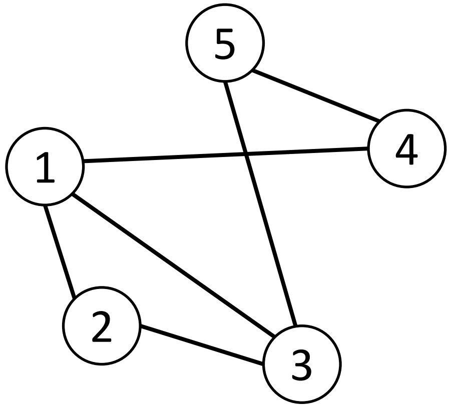
Graph Encoding via matrices

- Use a two-dimensional array called **adjacency matrix**

Columns

Rows

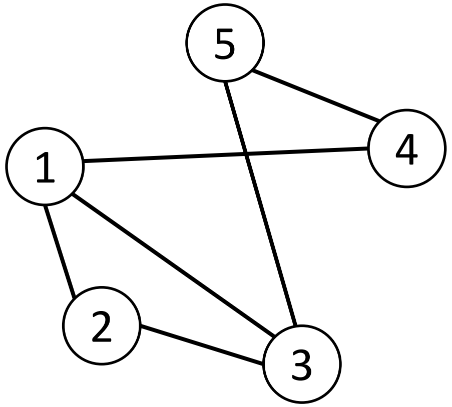
	(1)	(2)	(3)	(4)	(5)
(1)	0	1	1	1	0
(2)	1	0	1	0	0
(3)	1	1	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0



Graph Encoding via matrices

Columns

		(1)	(2)	(3)	(4)	(5)
	(1)	0	1	1	1	0
	(2)	1	0	1	0	0
Rows	(3)	1	1	0	0	1
	(4)	1	0	0	0	1
	(5)	0	0	1	1	0

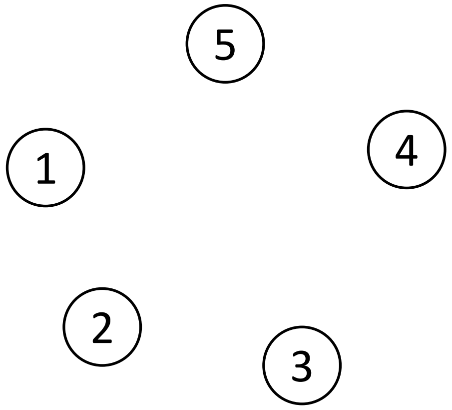


- For row i and column j (with $i \neq j$),
  $A[i][j] = 1$ if and only if there is an edge between nodes i and j

- Adjacency matrix is **symmetric** for undirected graphs: $A[i][j] = A[j][i]$

Adjacency Matrices

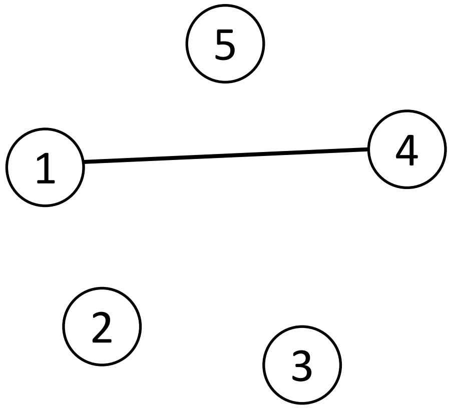
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	0	0	0	0
(2)	0	0	0	0	0
(3)	0	0	0	0	0
(4)	0	0	0	0	0
(5)	0	0	0	0	0

Adjacency Matrices

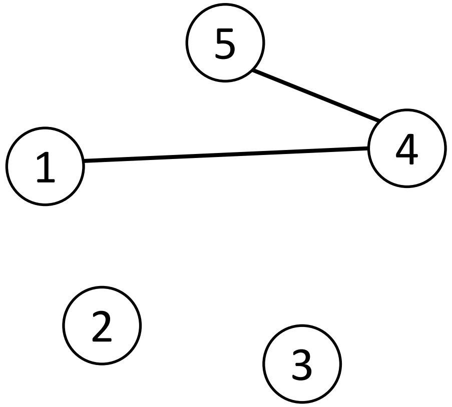
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	0	0	1	0
(2)	0	0	0	0	0
(3)	0	0	0	0	0
(4)	1	0	0	0	0
(5)	0	0	0	0	0

Adjacency Matrices

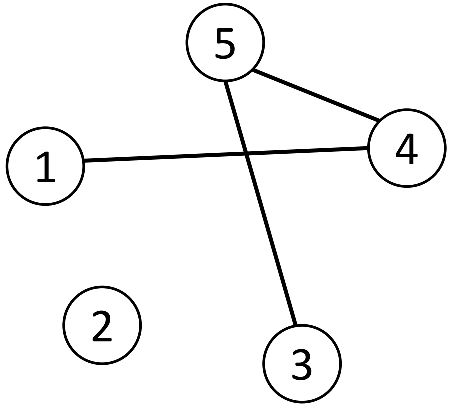
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	0	0	1	0
(2)	0	0	0	0	0
(3)	0	0	0	0	0
(4)	1	0	0	0	1
(5)	0	0	0	1	0

Adjacency Matrices

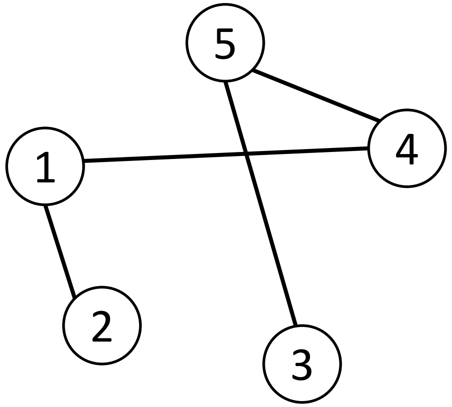
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	0	0	1	0
(2)	0	0	0	0	0
(3)	0	0	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0

Adjacency Matrices

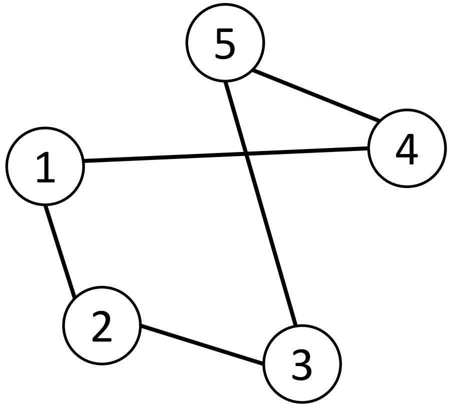
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	1	0	1	0
(2)	1	0	0	0	0
(3)	0	0	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0

Adjacency Matrices

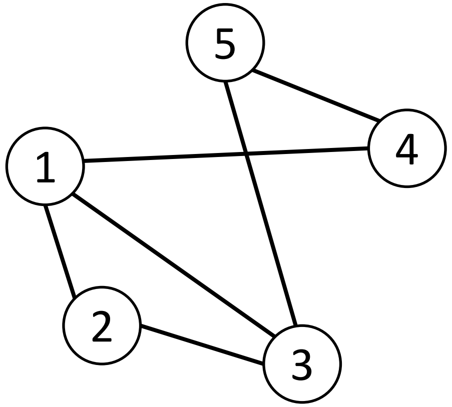
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	1	0	1	0
(2)	1	0	1	0	0
(3)	0	1	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0

Adjacency Matrices

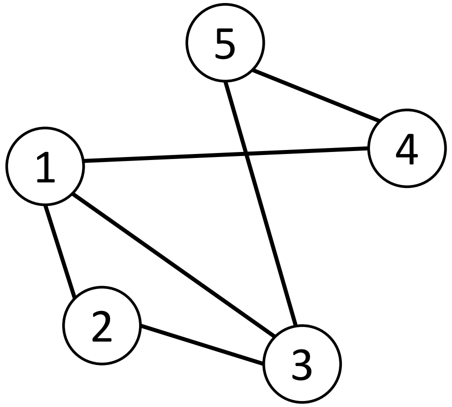
- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	1	1	1	0
(2)	1	0	1	0	0
(3)	1	1	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0

Adjacency Matrices

- For an undirected graph, **two cells** are always modified when we **add or remove an edge**



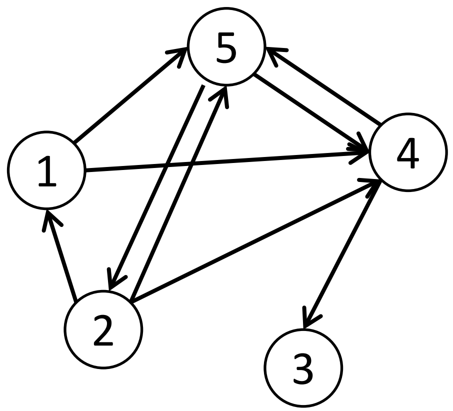
	(1)	(2)	(3)	(4)	(5)
(1)	0	1	1	1	0
(2)	1	0	1	0	0
(3)	1	1	0	0	1
(4)	1	0	0	0	1
(5)	0	0	1	1	0

The sum value of a row tells us how many edges go out of this node

The sum value of a column tells us how many edges go in this node

Adjacency Matrices

- For directed graph, **single cell** modified when we **add or remove a directed edge**



	(1)	(2)	(3)	(4)	(5)
(1)	0	0	0	1	1
(2)	1	0	0	1	1
(3)	0	0	0	0	0
(4)	0	0	1	0	1
(5)	0	1	0	1	0

- Read the 1s in row i as: “node i has an edge to column X”

Matrix-based: Adding a node

```
public class GraphMatrix{

    int graph[][];
    int size;

    public GraphMatrix(){ this.size = 0;}

    public void addNode(){
        size++;
        int[][] newGraph = new int[size][size];
        for(int i = 0; i < size-1;i++){
            for(int j = 0; j < size-1;j++){
                newGraph[i][j] = graph[i][j];
            }
        }
        this.graph = newGraph;
    }
…
```

Matrix-based: Removing a node

```
public void removeNode(int k){
    if (k <= 0 || k > size)
        throw new IllegalArgumentException("Bad index.");
    size--;
    int[][] newGraph = new int[size][size];
    for(int i = 0; i < size;i++){
        for(int j = 0; j < size;j++){
            int ip = (i >= k-1) ? 1 : 0;
            int jp = (j >= k-1) ? 1 : 0;
            newGraph[i][j] = graph[i+ip][j+jp];
        }
    }
    this.graph = newGraph;
}
```

Matrix-based: Adding and removing edges

```
public void addEdge(int i, int j){
    if (i <= 0 || i > size || j <= 0 || j > size)
        throw new IllegalArgumentException("Bad indices.");
    graph[i-1][j-1] = 1;
    graph[j-1][i-1] = 1;
}
```

```
public void removeEdge(int i, int j){
    if (i <= 0 || i > size || j <= 0 || j > size)
        throw new IllegalArgumentException("Bad indices.");
    graph[i-1][j-1] = 0;
    graph[j-1][i-1] = 0;
}
```

Matrix-based: Neighbors and adjacency

```
int[] getNeighbors(int i){
    if (i <= 0 || i > size)
        throw new IllegalArgumentException("Bad indices.");
    return graph[i-1];
}
```

```
public boolean adjacent(int i, int j){
    if (i <= 0 || i > size || j <= 0 || j > size)
        throw new IllegalArgumentException("Bad indices.");
    return (graph[i-1][j-1] == 1);
}
```

Graph ADT: Memory and time complexity

1. **Memory** = $O(N^2)$ for graphs with N nodes and E edges (memory-inefficient)

2. **Worst-case time complexity**:
   - Adding node is $O(N^2)$,
   - Removing node is $O(N^2)$,
   - Adding and removing an edge is $O(1)$,
   - Adjacency check is $O(1)$

Graph Traversals

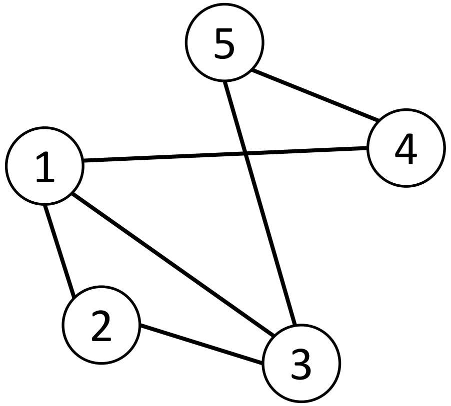
- Say you want to discover whether from a specific (source) node s:
  - Another specific node v is **reachable** from s, and what is the **shortest path** from s to v ,
  - Or which nodes are reachable from s within some k hops.

- How do we solve? We **traverse the graph using:**
  1. Depth-First Search
  2. Breadth-First Search
  3. Etc..

- We went over these traversals previously, but only for trees. Now we need to **deal with cycles and loops**.

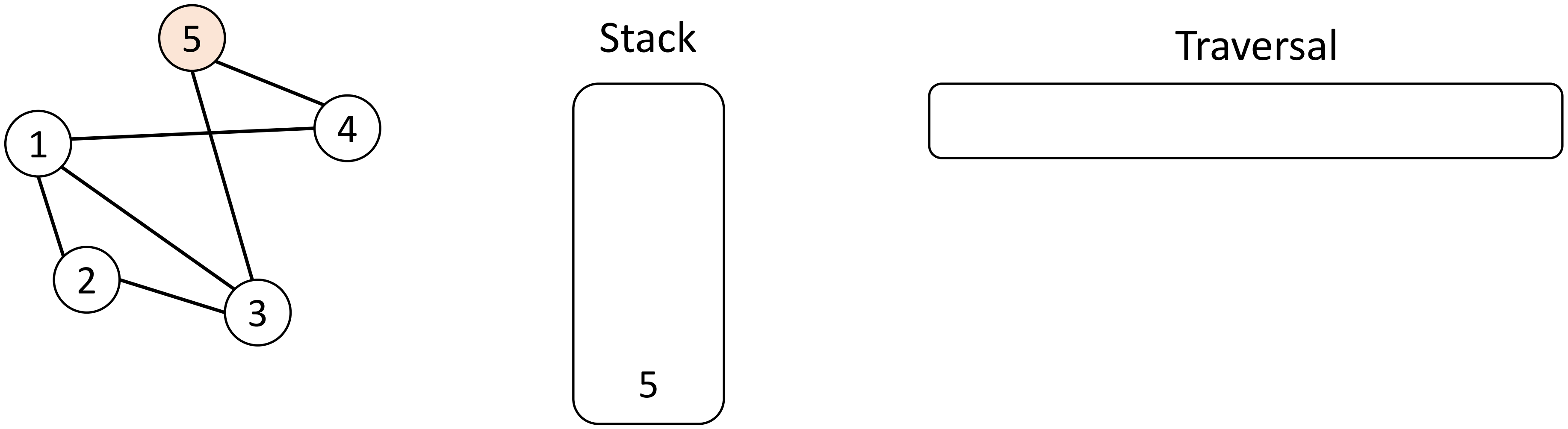
Depth-first search (DFS) traversal

1. **Start at the source, and visit a neighbor (and mark it as visited)**,
2. From that neighbor, visit one of its neighbors.
3. Repeat **until you hit an already visited node**.
4. At which point, you **backtrack (i.e., go back)**, and visit another neighbor instead.
5. Until you go back to the source.



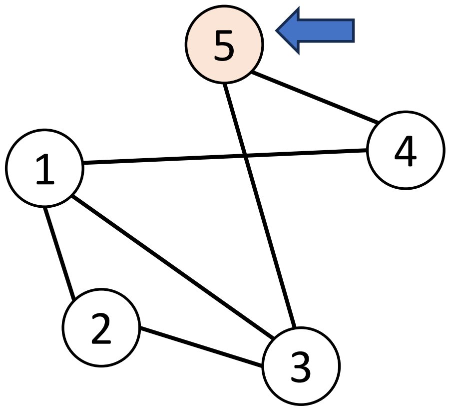
Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



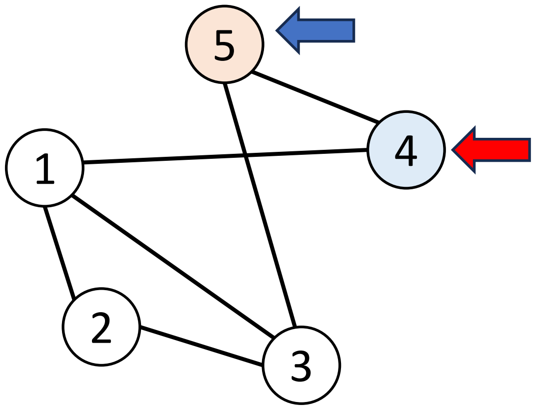
Stack

Traversal

5

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

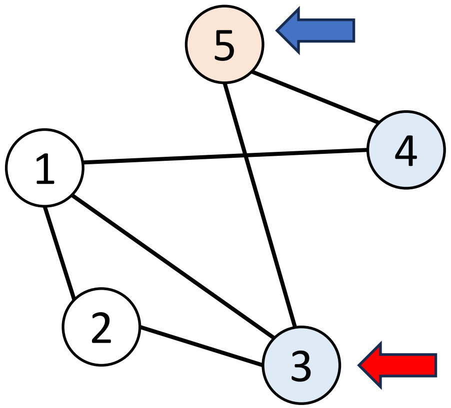
4

Traversal

5

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

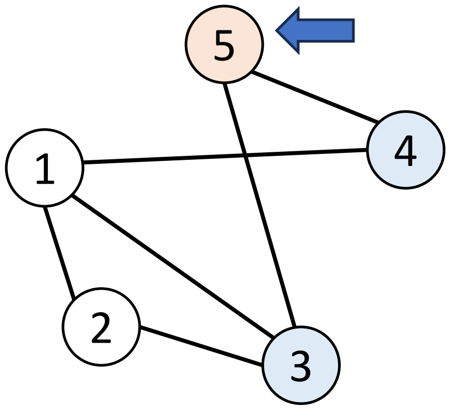
3
4

Traversal

5

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

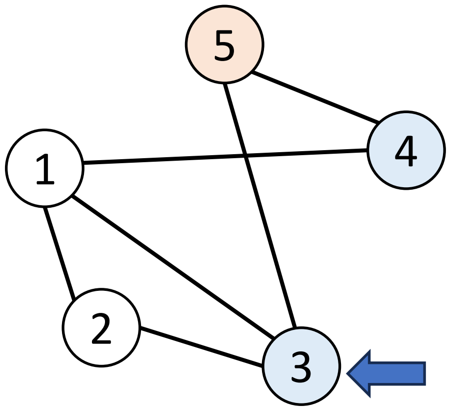
3
4

Traversal

5

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

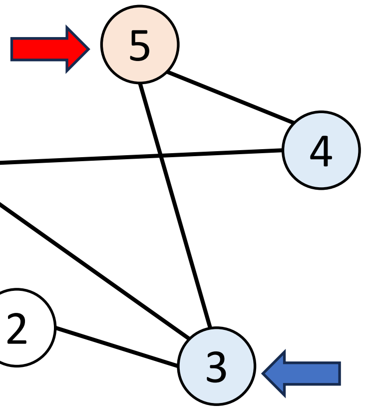
4

Traversal

5, 3

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

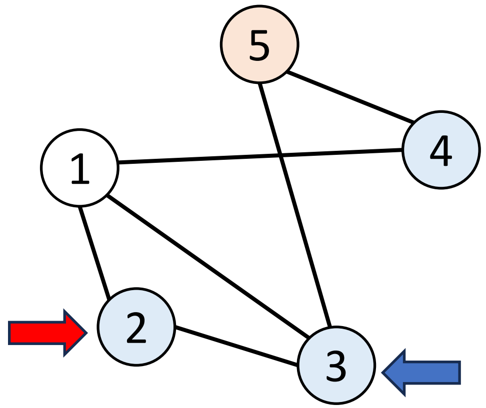
4

Traversal

5, 3

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

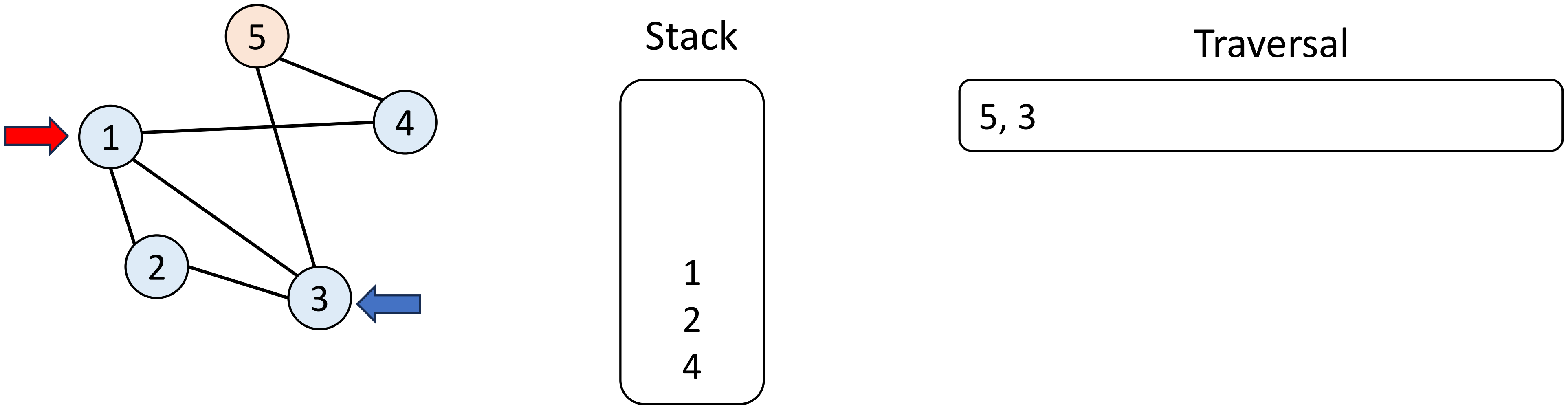
2
4

Traversal

5, 3

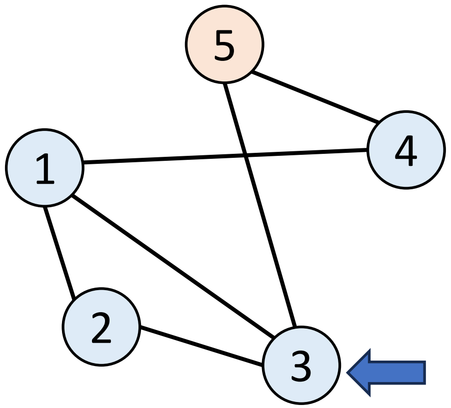
Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

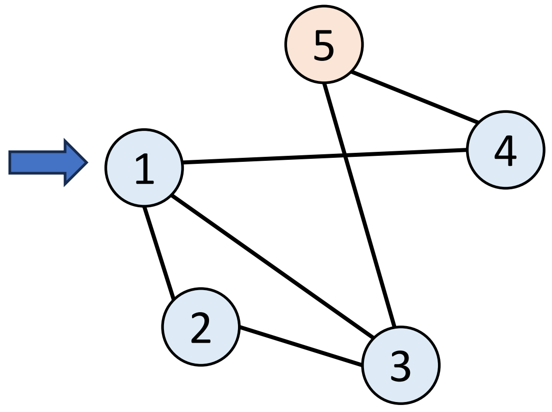
1
2
4

Traversal

5, 3

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

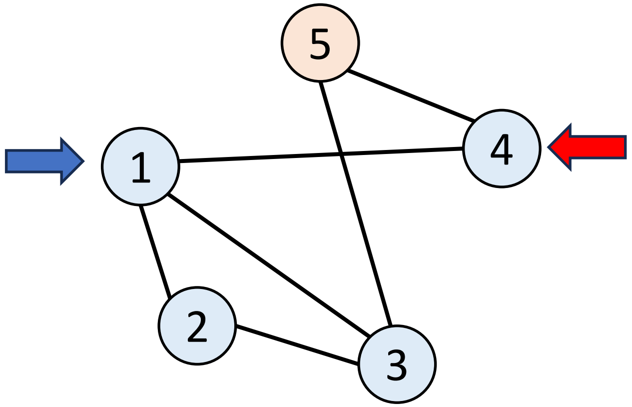
2
4

Traversal

5, 3, 1

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

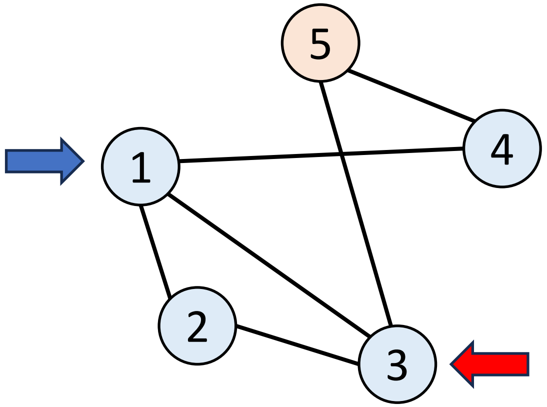
2
4

Traversal

5, 3, 1

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

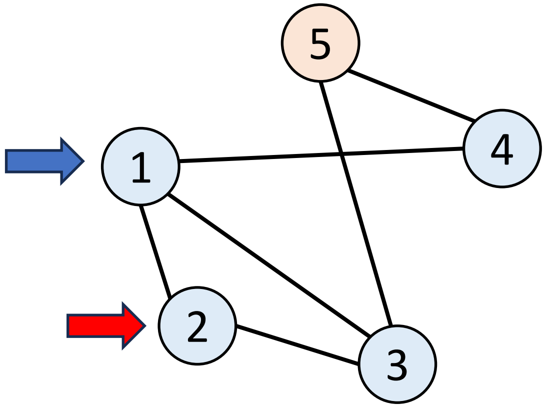
2
4

Traversal

5, 3, 1

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

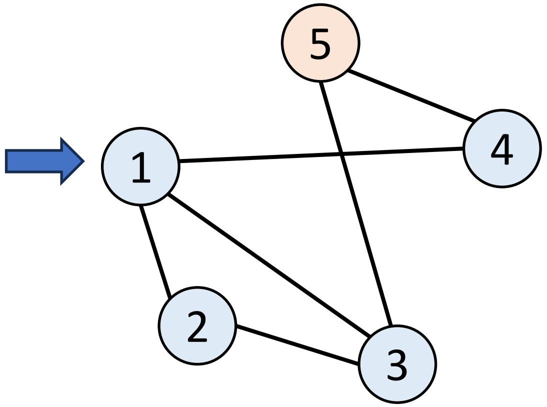
2
4

Traversal

5, 3, 1

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

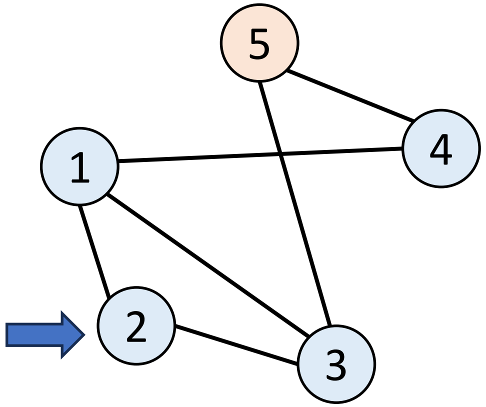
2
4

Traversal

5, 3, 1

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

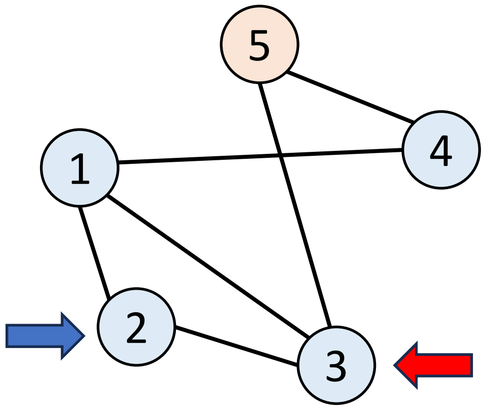
4

Traversal

5, 3, 1, 2

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

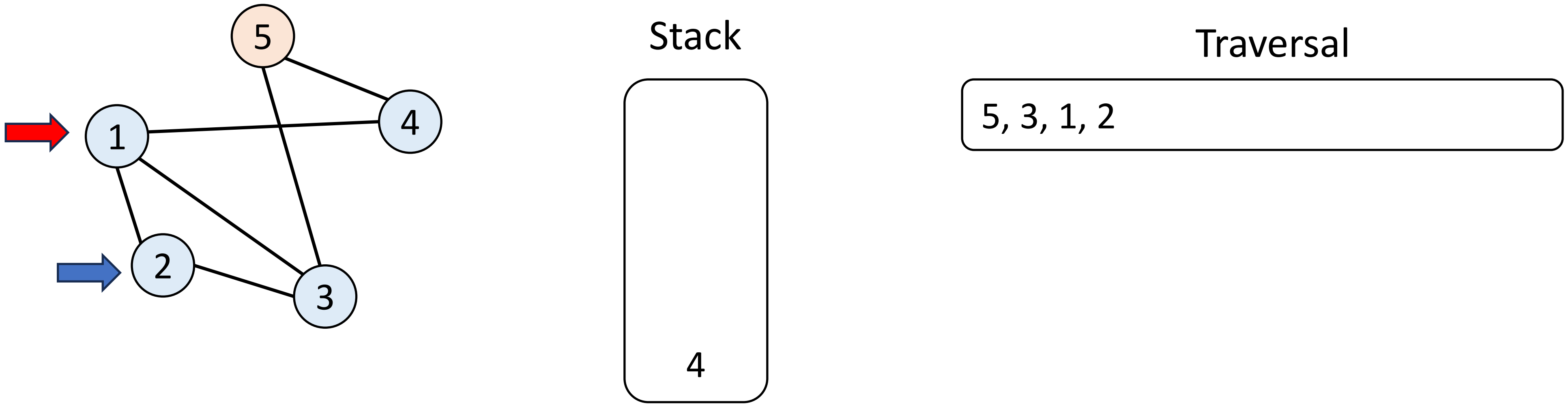
4

Traversal

5, 3, 1, 2

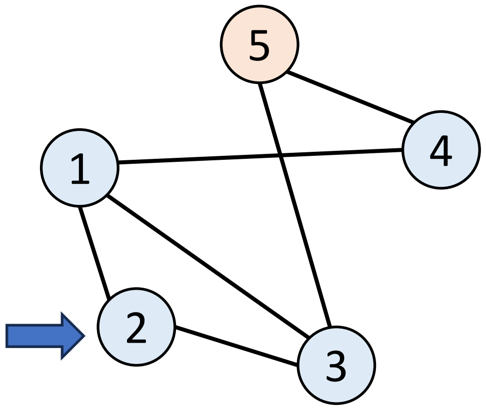
Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack

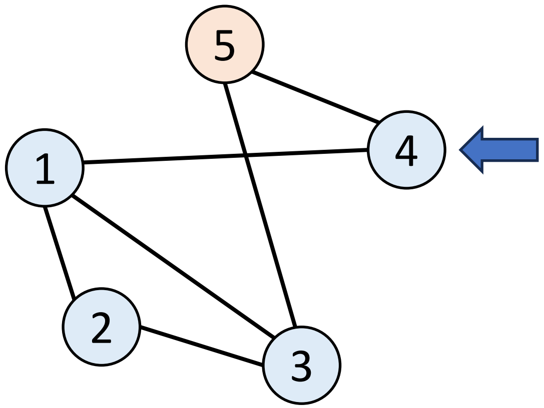
4

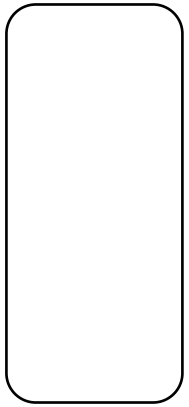
Traversal

5, 3, 1, 2

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



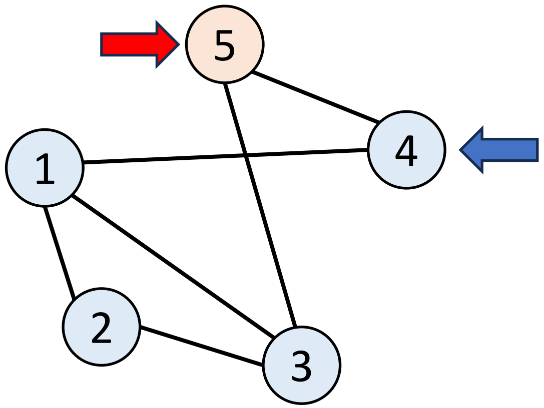


Traversal

5, 3, 1, 2, 4

Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



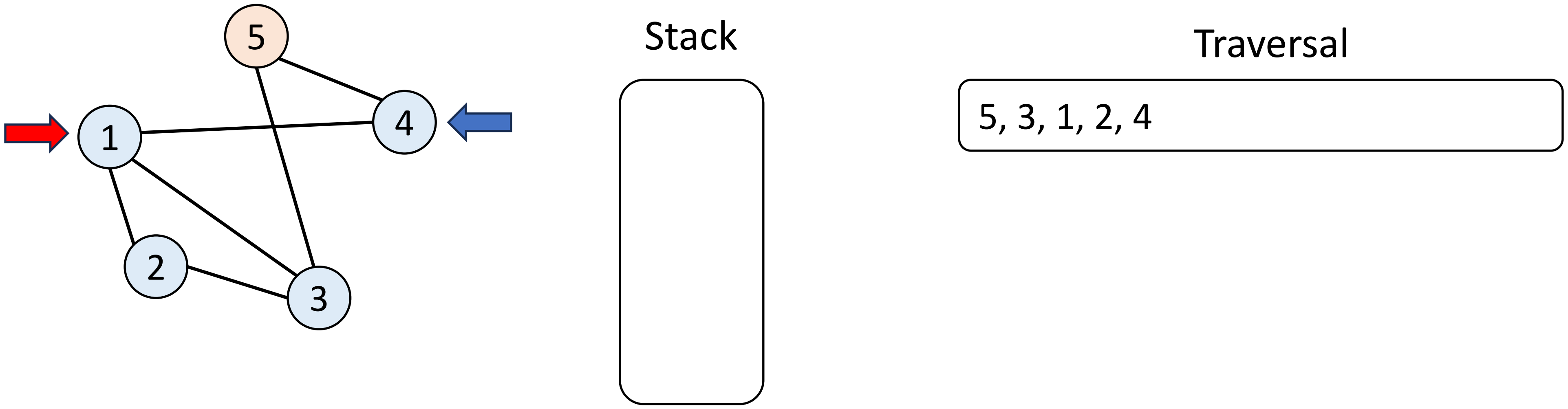
Stack

Traversal

5, 3, 1, 2, 4

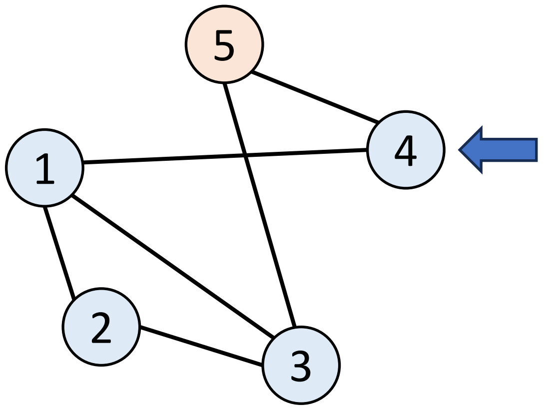
Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.

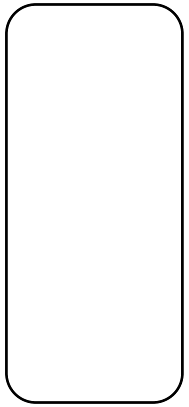


Depth-first search (DFS) traversal

- Start at the source, visit a neighbor, then one of its neighbors and so on until you hit an already visited node. At which point, you backtrack.



Stack



Traversal

5, 3, 1, 2, 4

Graph ADT: DFS Code (Iterative, Preorder)

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class GraphSearch{
    boolean[] visitedNodes;
    GraphMatrix graph;
    int graphSize;

    public GraphSearch(GraphMatrix g){
        this.graph = g;
        this.graphSize = g.graph.length;
        this.visitedNodes = new boolean[graphSize];
    }

    public LinkedList<Integer> DFS_traverse(int source){
    …
    }
}
```

Graph ADT: DFS Code (Iterative, Preorder)

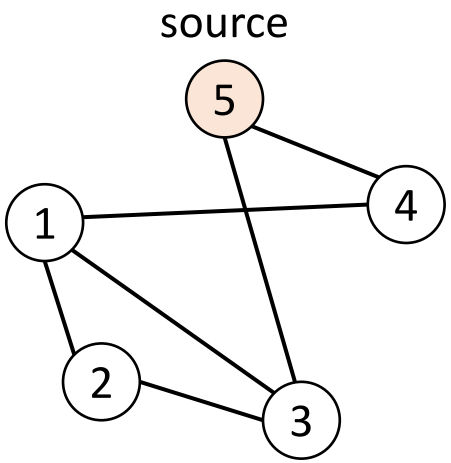
```
public LinkedList<Integer> DFS_traverse(int source){
    Stack<Integer> stack = new Stack<Integer>();
    LinkedList<Integer> traversal = new LinkedList<Integer>();
    stack.push(source);
    visitedNodes[source-1] = true;
    while(stack.size() > 0){
        int currentVisitedNode = stack.pop();
        traversal.add(currentVisitedNode);
        for (int i = graphSize-1; i >= 0; i--) {
            if (graph.graph[currentVisitedNode-1][i] == 1
                    && ! visitedNodes[i]){
                visitedNodes[i] = true;
                stack.push(i+1);
            }
        }
    }
    return traversal;
}
```

Graph ADT: DFS Code (Recursive version)

```
public LinkedList<Integer> DFS_traverse_recursive(int source){
    LinkedList<Integer> traversal = new LinkedList<Integer>();
    visitedNodes[source-1] = true;
    return DFS_traverse_recursive_aux(source, traversal);
}
private LinkedList<Integer> DFS_traverse_recursive_aux(int currentNode, LinkedList traversal){
    traversal.add(currentNode);
    for (int i = 0; i < graphSize; i++) {
        if (graph.graph[currentNode-1][i] == 1
                && ! visitedNodes[i]){
            visitedNodes[i] = true;
            DFS_traverse_recursive_aux(i+1, traversal);
        }
    }
    return traversal;
}
```

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

5

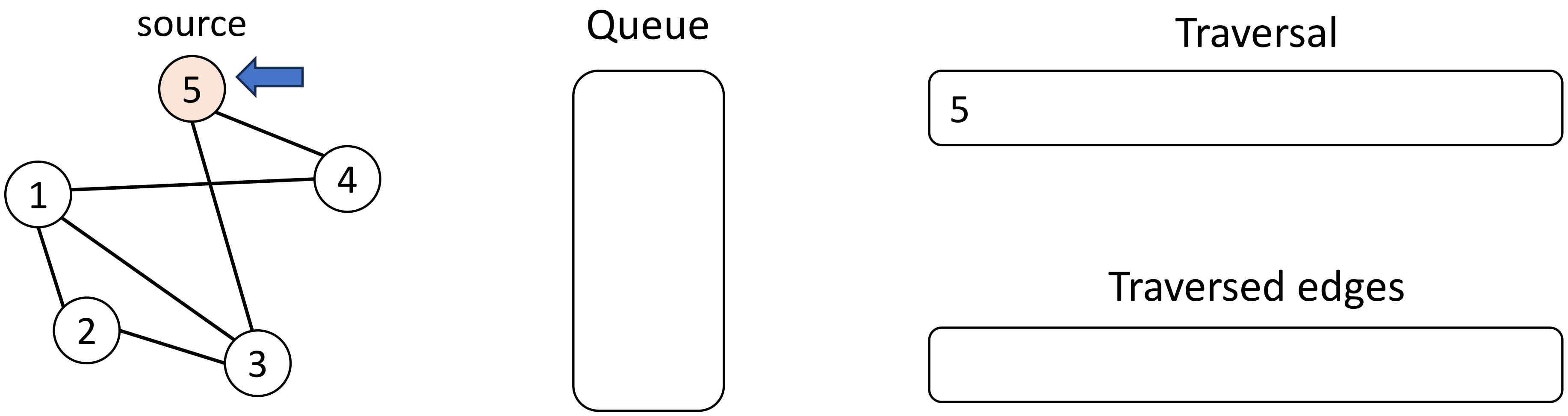
Traversal

5

Traversed edges

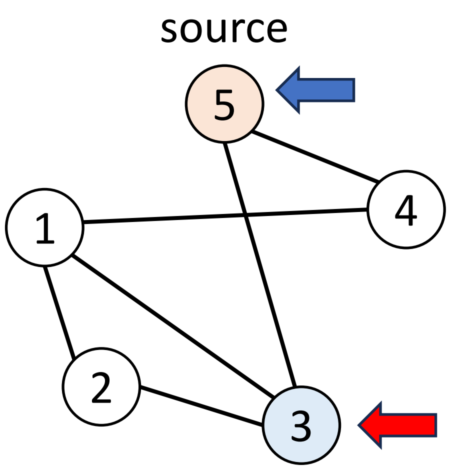
Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

3

Traversal

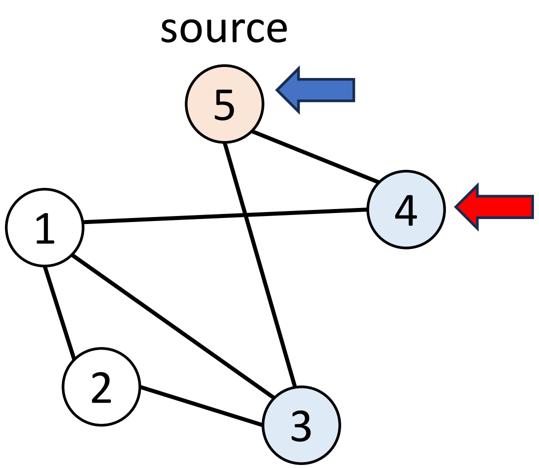
5, 3

Traversed edges

(5,3)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

4
3

Traversal

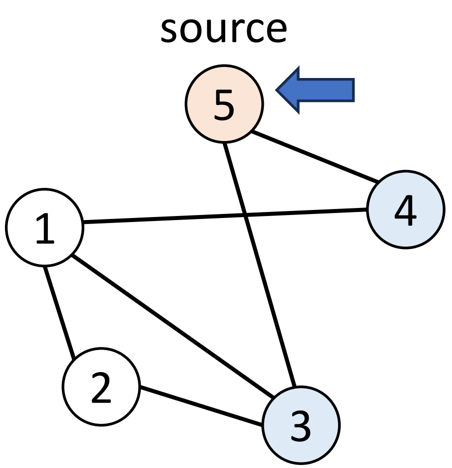
5, 3, 4

Traversed edges

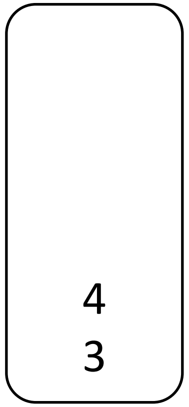
(5,3), (5,4)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue



Traversal

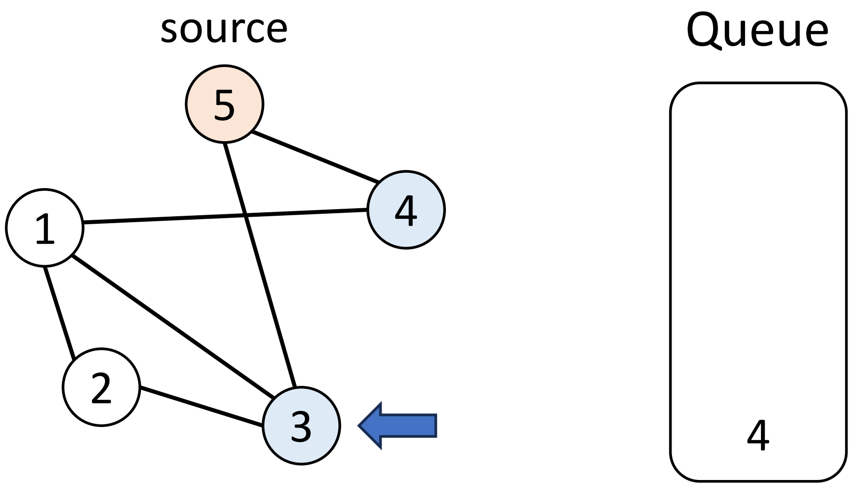
5, 3, 4

Traversed edges

(5,3), (5,4)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Traversal

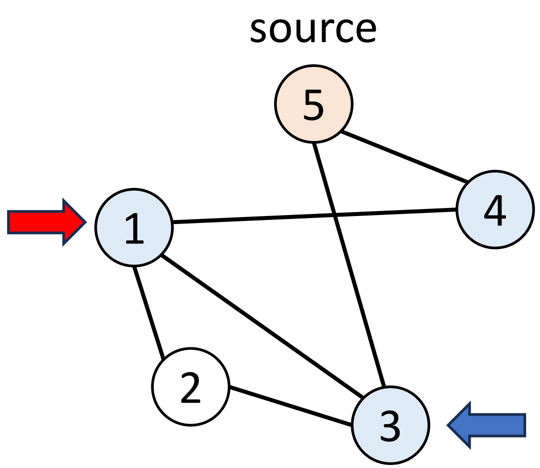
5, 3, 4

Traversed edges

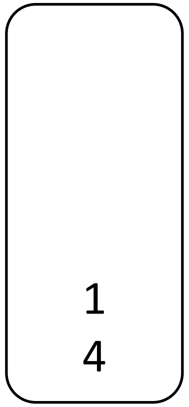
(5,3), (5,4)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue



Traversal

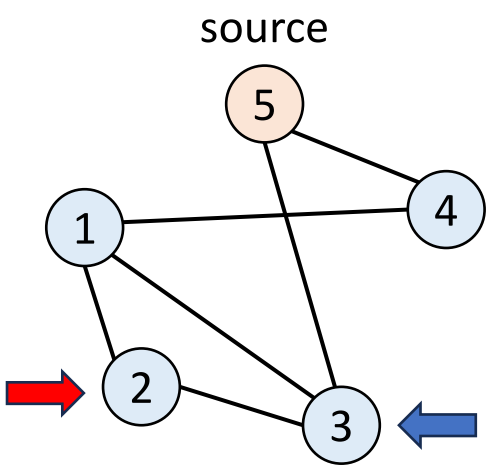
5, 3, 4, 1

Traversed edges

(5,3), (5,4), (3,1)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1
4

Traversal

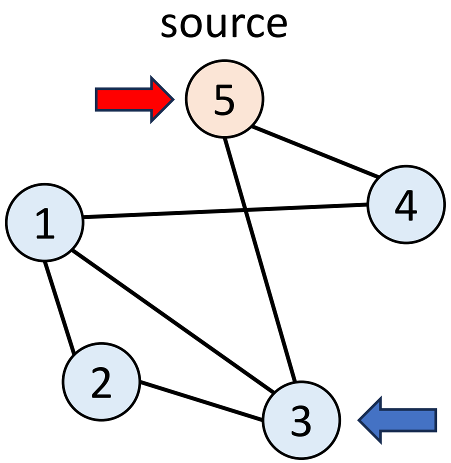
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1
4

Traversal

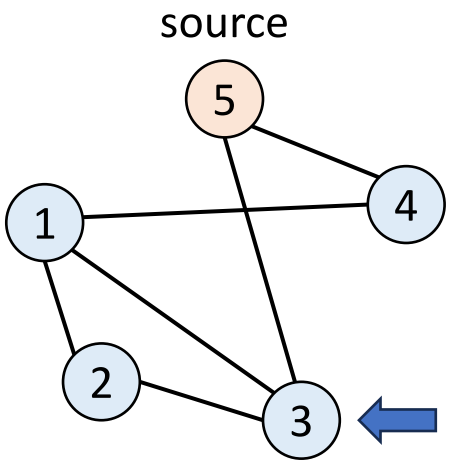
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1
4

Traversal

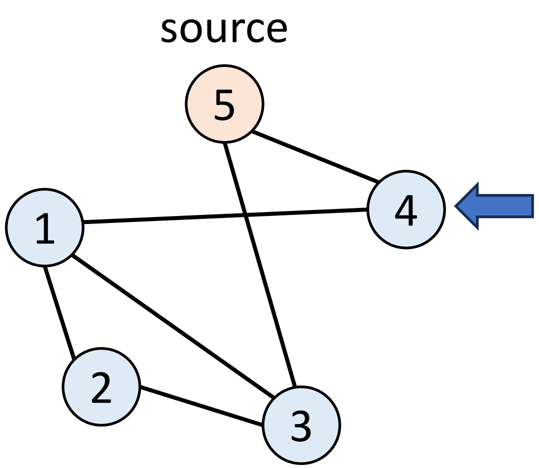
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1

Traversal

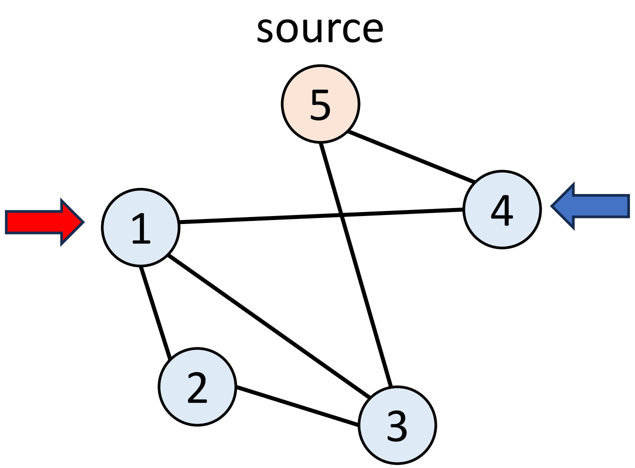
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1

Traversal

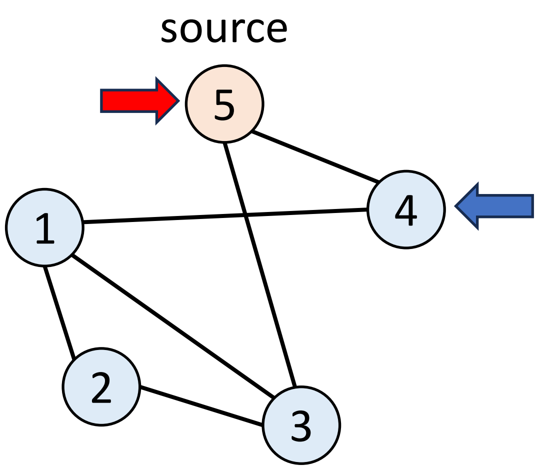
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1

Traversal

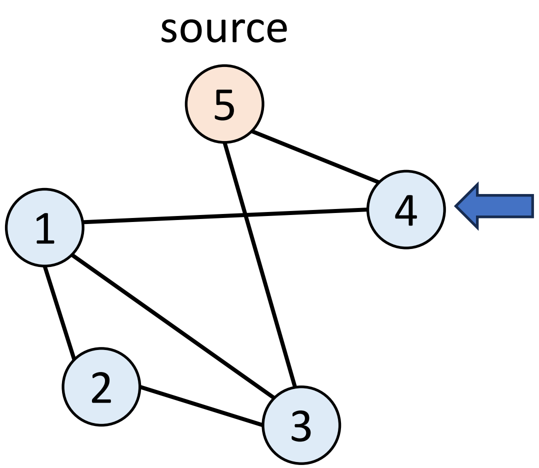
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2
1

Traversal

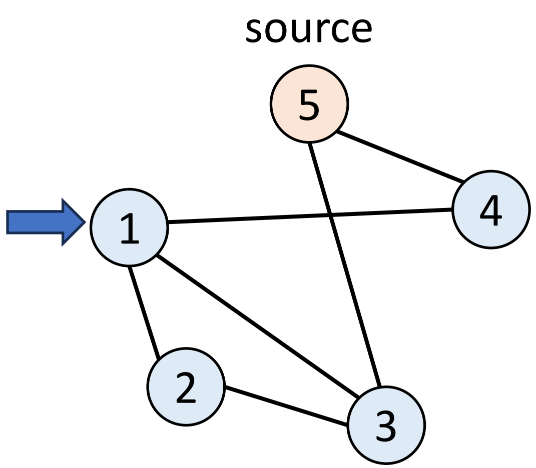
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2

Traversal

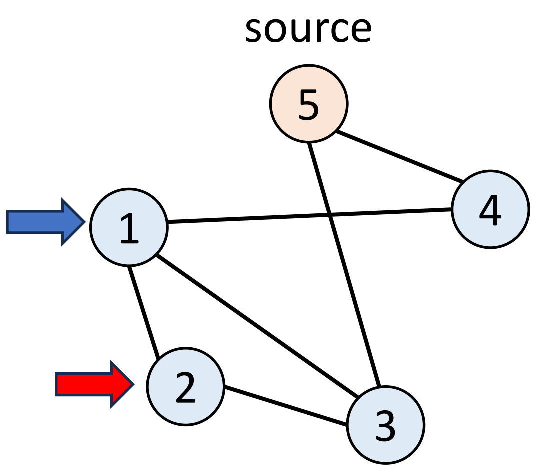
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2

Traversal

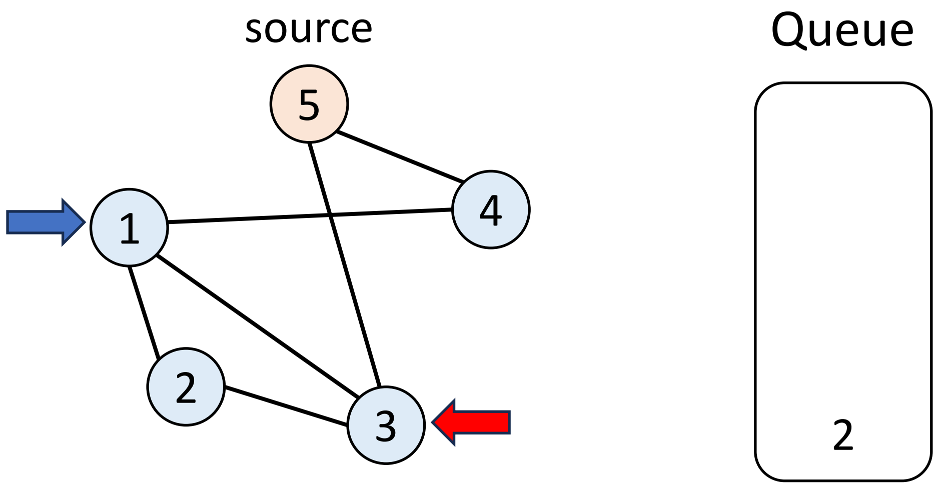
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Traversal

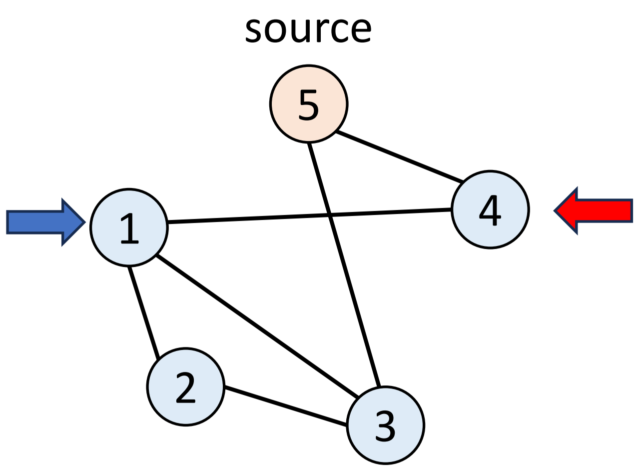
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2

Traversal

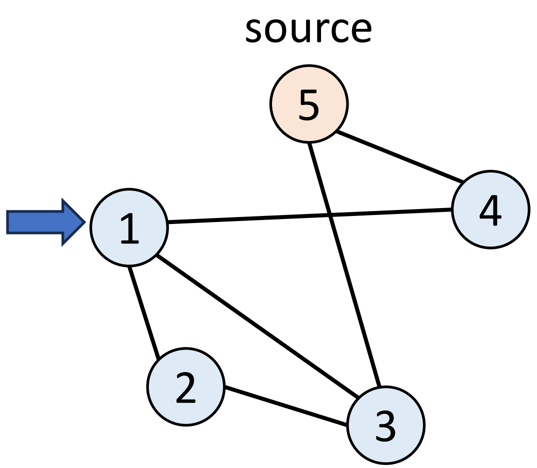
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

2

Traversal

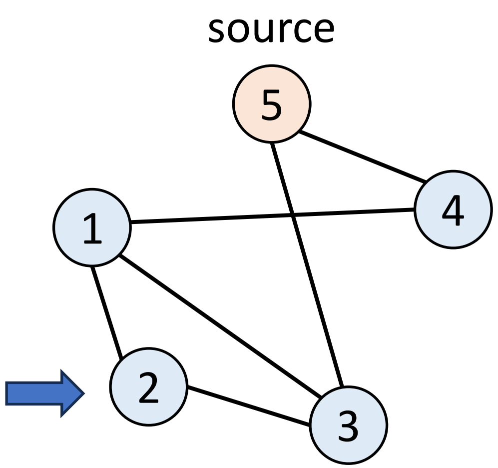
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

Traversal

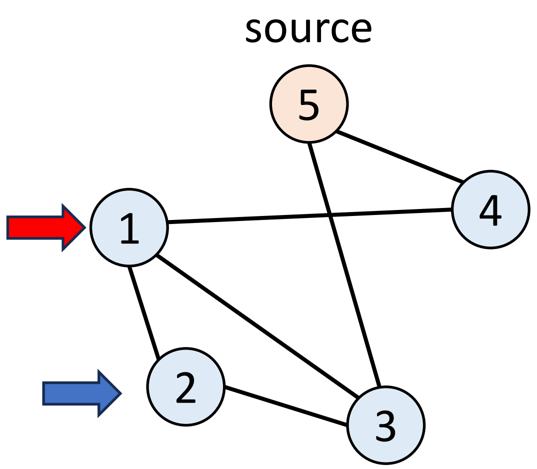
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

Traversal

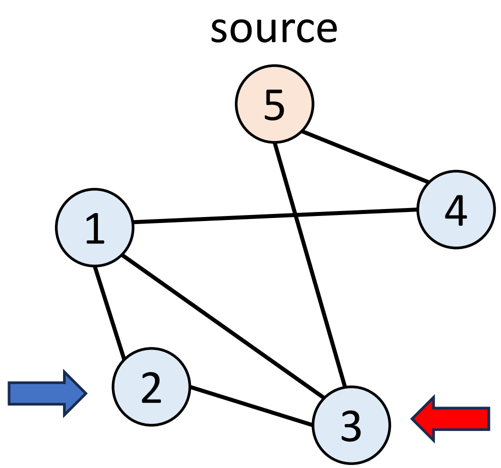
5, 3, 4, 1, 2

Traversed edges

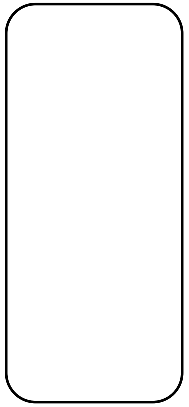
(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue



Traversal

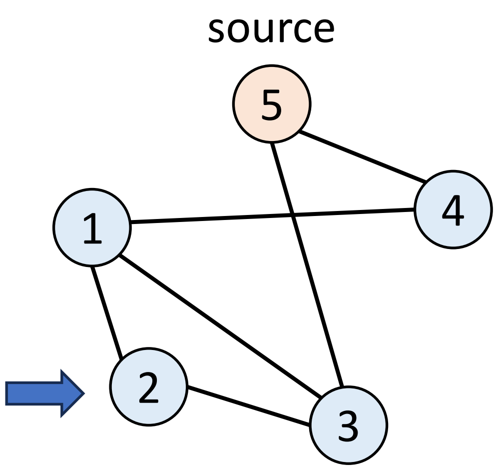
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

Traversal

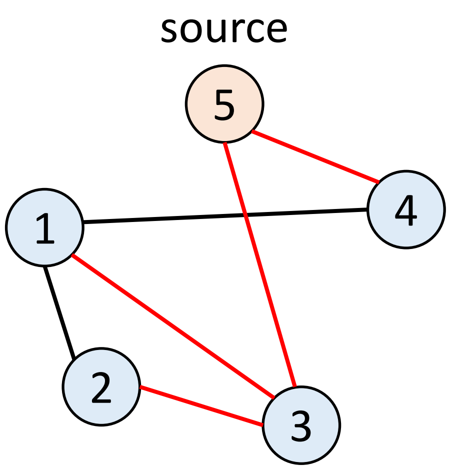
5, 3, 4, 1, 2

Traversed edges

(5,3), (5,4), (3,1), (3,2)

Breadth-first search (BFS) traversal

- Start at the source, and visit all nodes at distance 1, then all nodes at distance 2, …



Queue

Traversal

5, 3, 4, 1, 2 (end)

Traversed edges

(5,3), (5,4), (3,1), (3,2)

- When done, the traversed edges form a **BFS tree**
- **The BFS tree gives the shortest paths from *s* (if no edge weights).**

Graph ADT: BFS Code

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class GraphSearch{
    boolean[] visitedNodes;
    GraphMatrix graph;
    int graphSize;

    public GraphSearch(GraphMatrix g){
        this.graph = g;
        this.graphSize = g.graph.length;
        this.visitedNodes = new boolean[graphSize];
    }

    public LinkedList<Integer> BFS_traverse(int source){
    …
    }
}
```

Graph ADT: BFS Code

```
public LinkedList<Integer> BFS_traverse(int source){
    Queue<Integer> queue = new LinkedList<Integer>();
    LinkedList<Integer> traversal = new LinkedList<Integer>();
    queue.add(source);
    traversal.add(source);
    visitedNodes[source-1] = true;
    while(queue.size() > 0){
        int currentVisitedNode = queue.remove();
        for (int i = 0; i < graphSize; i++) {
            if (graph.graph[currentVisitedNode-1][i] == 1
                    && ! visitedNodes[i]){
                traversal.add(i+1);
                visitedNodes[i] = true;
                queue.add(i+1);
            }
        }
    }
    return traversal;
}
```

Summary

Today's lecture:

- Introduced to graphs (including ADT),
- Graphs apply to wide range of situations!
  - As long as we have entities (data) and relationships between these entities,
  - We can apply many graph analysis techniques to the data to understand it better.
- DFS and BFS graph traversals.

- **Next Lecture:** Computing distances in graphs and more.
- **Any questions?**