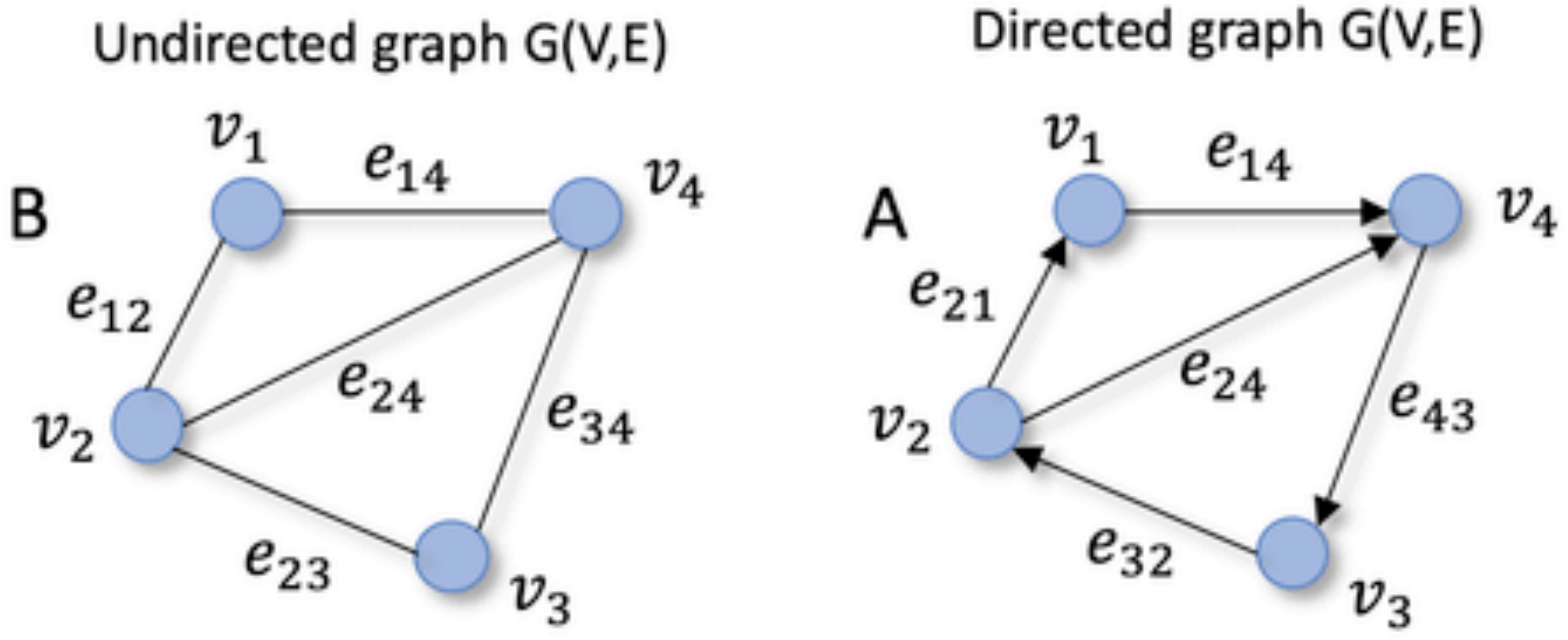
# CS62 Class 22: Graphs (intro, BFS/DFS)

Graphs




F

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | 1 | 0 | 1 |
| $v_2$ | 1 | 0 | 1 | 1 |
| $v_3$ | 0 | 1 | 0 | 1 |
| $v_4$ | 1 | 1 | 1 | 0 |

E

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | 0 | 0 | 1 |
| $v_2$ | 1 | 0 | 0 | 1 |
| $v_3$ | 0 | 1 | 0 | 0 |
| $v_4$ | 0 | 0 | 1 | 0 |

adjacency matrix

# Agenda

- Undirected graphs
  - Depth-first search
  - Breadth-first search
- Directed graphs
  - Depth-first search
  - Breadth-first search

# Why study graphs?

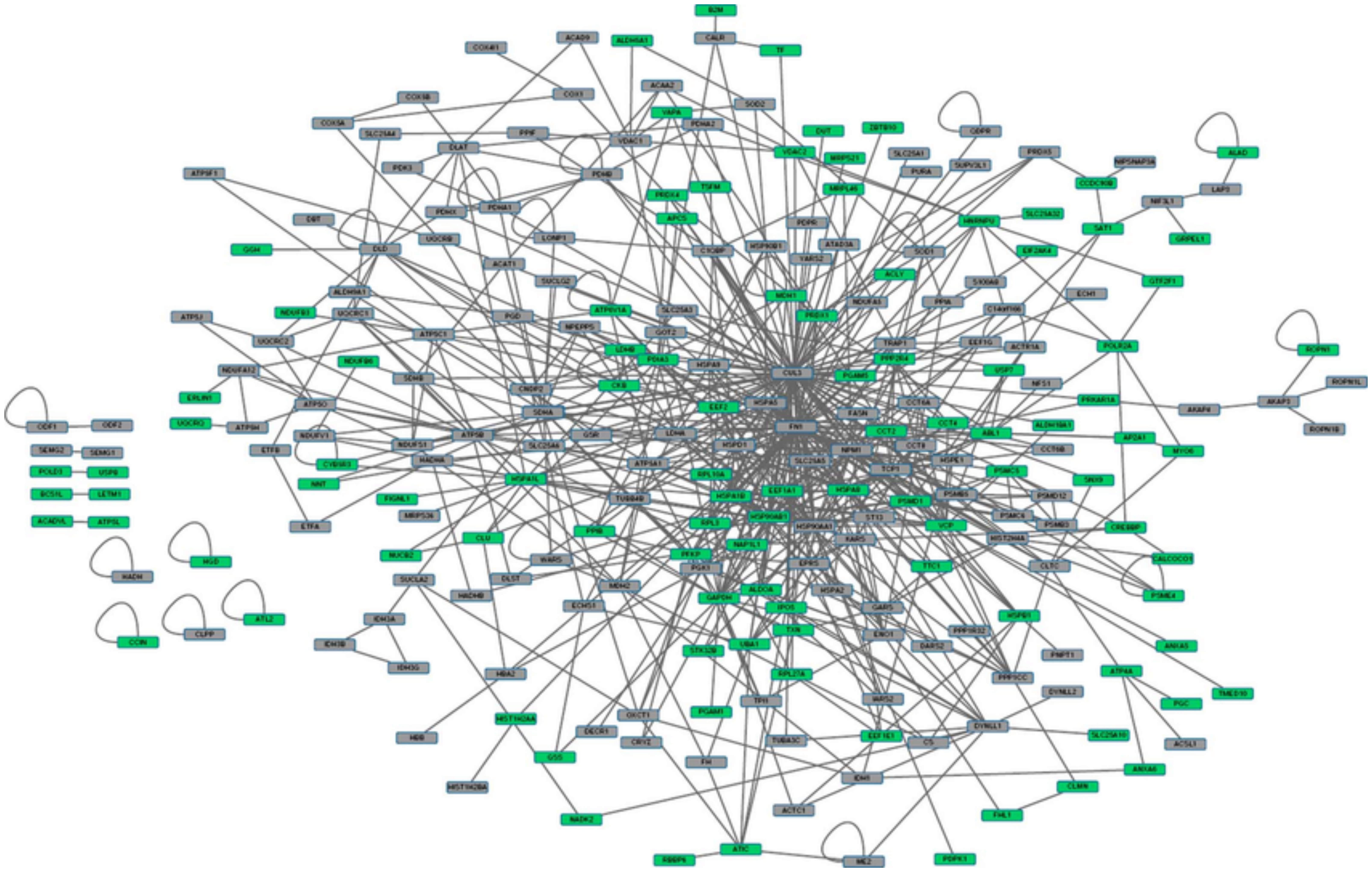
- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of theoretical computer science.

# Undirected graphs

# Undirected Graphs

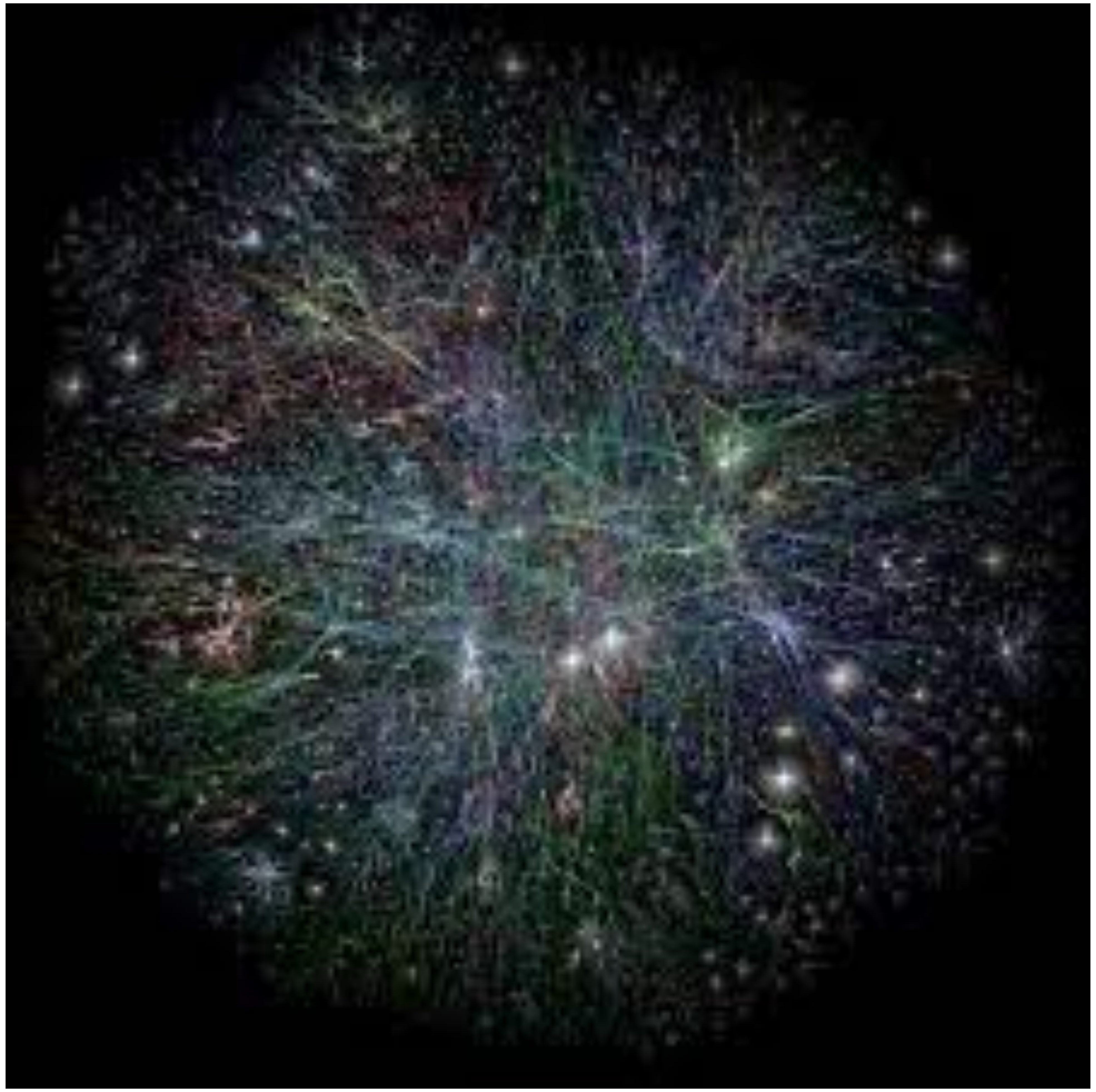
- Graph: A set of *vertices* connected pairwise by *edges*.
- Undirected graph: The edges do not point in a specific direction

# Protein-protein interaction graph



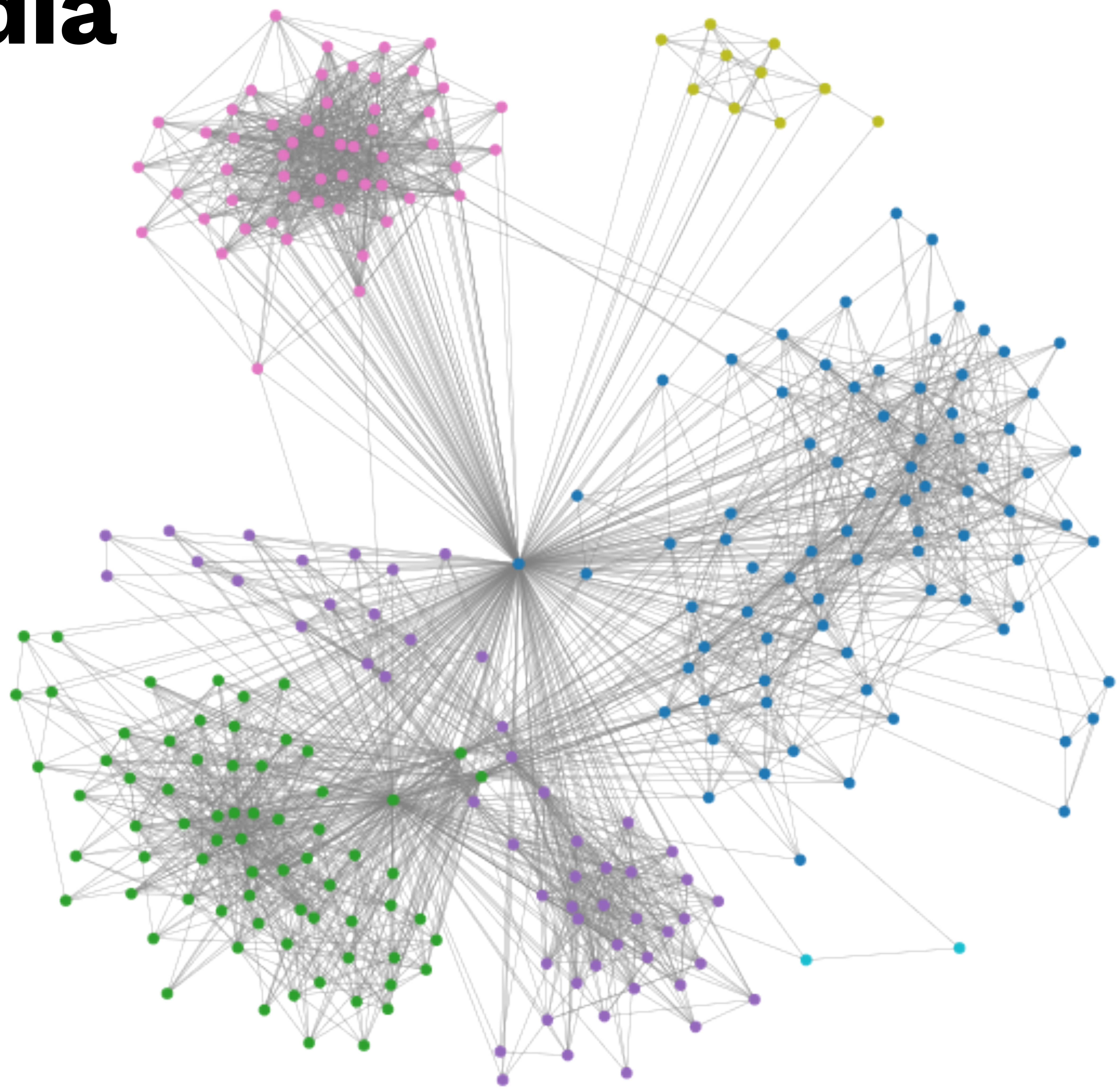
https://www.researchgate.net/figure/Network-graph-of-the-protein-protein-interactions-Green-color-represents-proteins_fig4_272297002

# The Internet



https://www.opte.org/the-internet

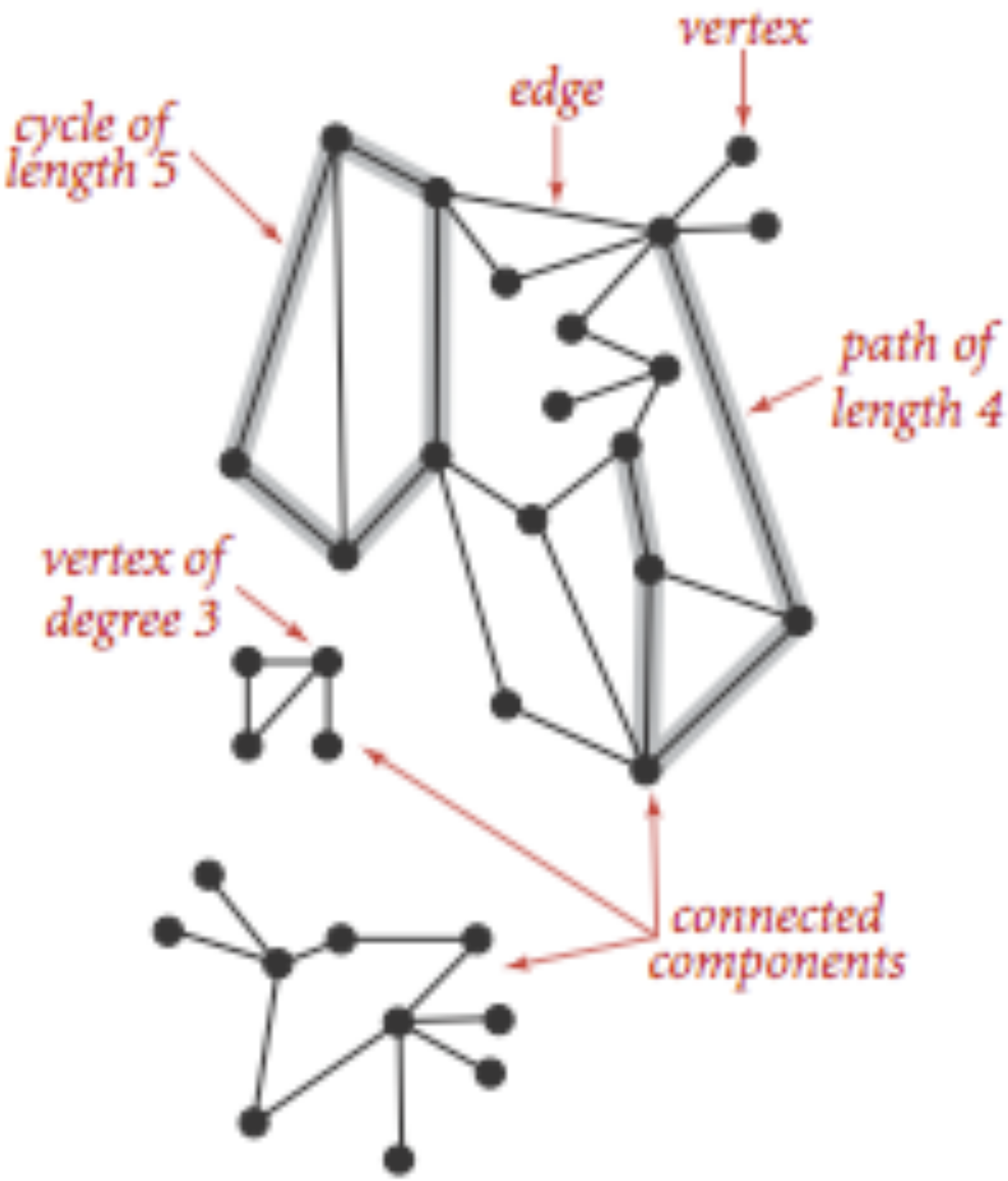
# Social media



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

# Graph terminology

- Path: Sequence of vertices connected by edges
- Cycle: Path whose first and last vertices are the same
- Two vertices are connected if there is a path between them




Anatomy of a graph

# Examples of graph-processing problems

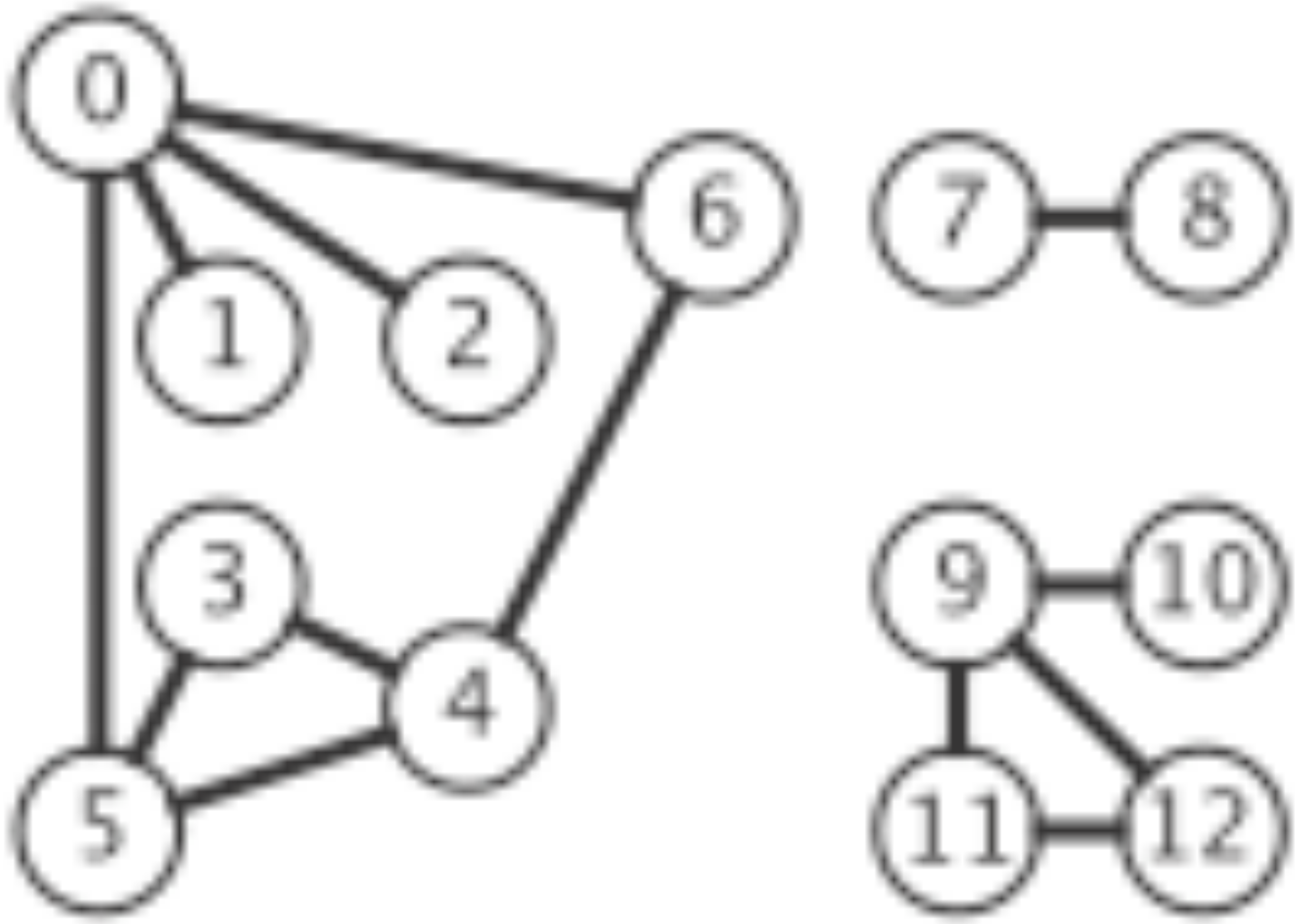
- Is there a path between vertex s and t?
  - What is the shortest path between s and t?
- Is there a cycle in the graph?
  - Euler Tour: Is there a cycle that uses each edge exactly once?
  - Hamilton Tour: Is there a cycle that uses each vertex exactly once?
- Is there a way to connect all vertices?
  - What is the shortest way to connect all vertices?
- Is there a vertex whose removal disconnects the graph?

# Graph representation

- Vertex representation: integers between 0 and V-1 (but can be generalized to any type, e.g., custom Nodes).

0 5 means there's an edge between vertices 0 and 5

```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



# Basic Graph API

`public class Graph`

- `Graph(int V)`: create an empty graph with V vertices.
- `void addEdge(int v, int w)`: add an edge v-w.
- `Iterable<Integer> adj(int v)`: return vertices adjacent to v.
- `int V()`: number of vertices.
- `int E()`: number of edges.

# Example of how to use the Graph API to process the graph

*The degree of a vertex v is the number of vertices connected to v (i.e., the number of edges.)*

```
public static int degree(Graph g, int v){
    int count = 0;
    for(int w : g.adj(v))
        count++;
    return count;
}
```

# Graph density

|V| = number of vertices

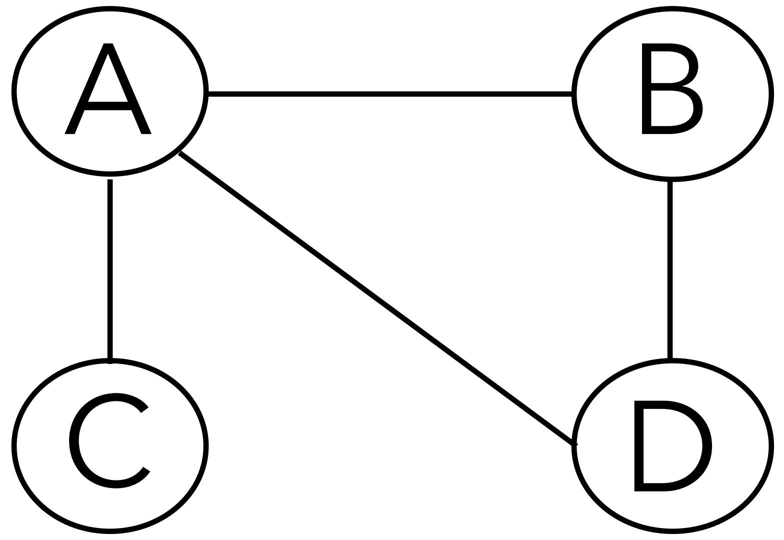
- In a simple graph (no parallel edges or loops), if $|V| = n$, then:
  - minimum number of edges is 0 and
  - maximum number of edges is $n(n-1)/2$. O(n^2) - all vertices are connected to each other
- Dense graph -> edges closer to maximum.
- Sparse graph -> edges closer to minimum.

# Graph representation: adjacency matrix

- Maintain a $|V|$-by-$|V|$ boolean array; for each edge `v-w`:
  - `adj[v][w] = adj[w][v] = true;`
- Good for dense graphs (edges close to $|V|^2$).
- Constant time for lookup of an edge.
- Constant time for adding an edge.
- $|V|$ time for iterating over vertices adjacent to $v$.
- Symmetric, therefore wastes space in undirected graphs ($|V|^2$).
- Not widely used in practice.

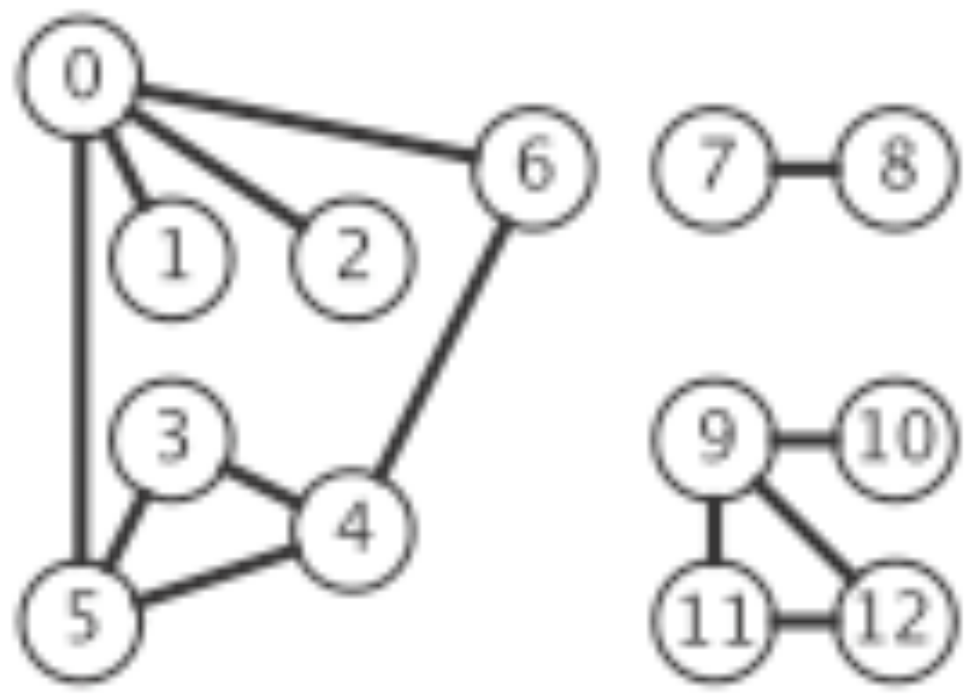
*For undirected graphs, adjacency matrices are always symmetric along the diagonal*

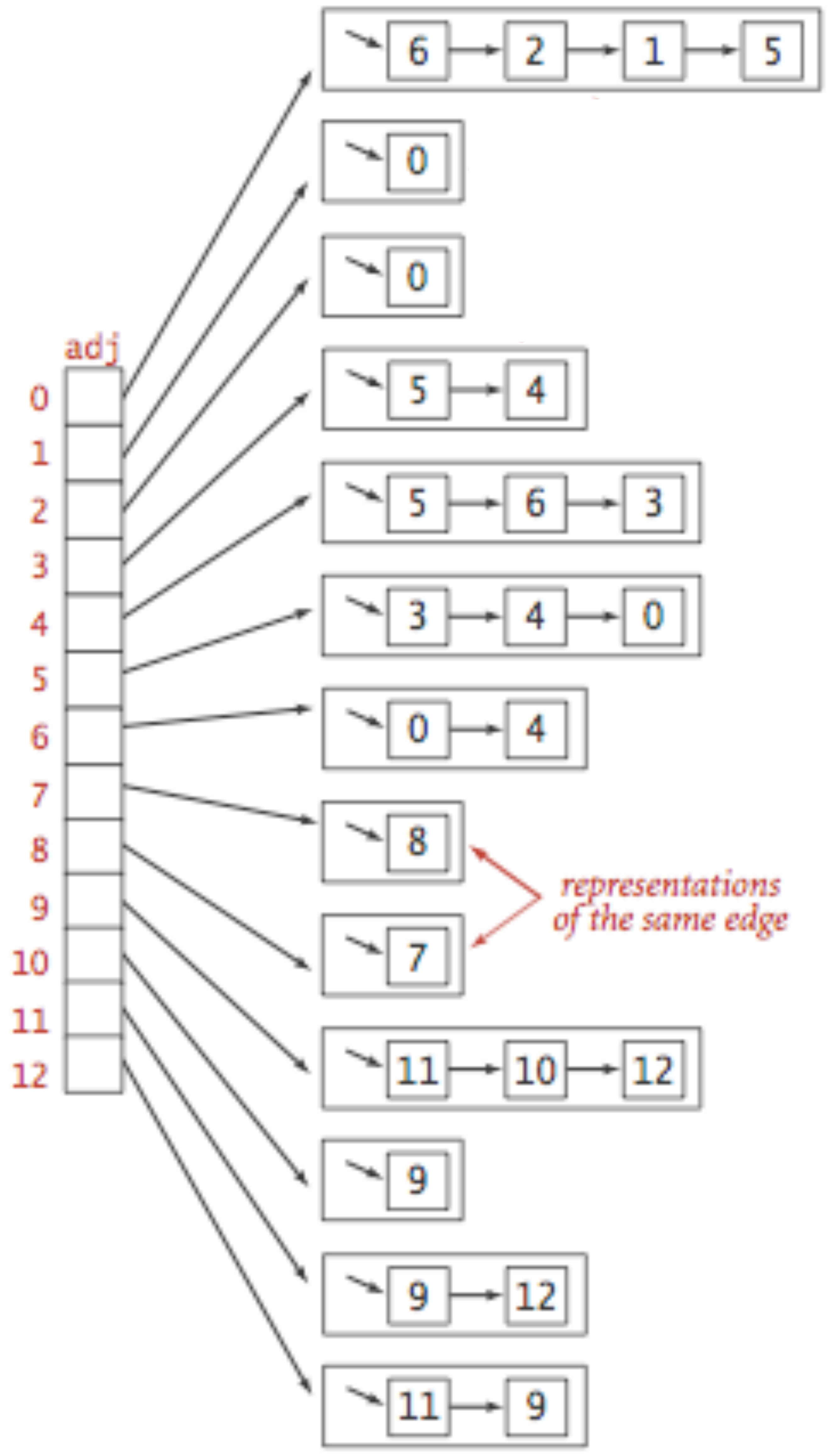
| | A | B | C | D |
|---|---|---|---|---|
| **A** | 0 | 1 | 1 | 1 |
| **B** | 1 | 0 | 0 | 1 |
| **C** | 1 | 0 | 0 | 0 |
| **D** | 1 | 1 | 0 | 0 |



# Graph representation: adjacency list

- Maintain vertex-indexed array of lists. The list stores vertices adjacent to $v$.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Space efficient ($|E| + |V|$).
- Constant time for adding an edge.
- Lookup of an edge or iterating over vertices adjacent to $v$ is $degree(v)$.



# Adjacency-list graph representation in Java

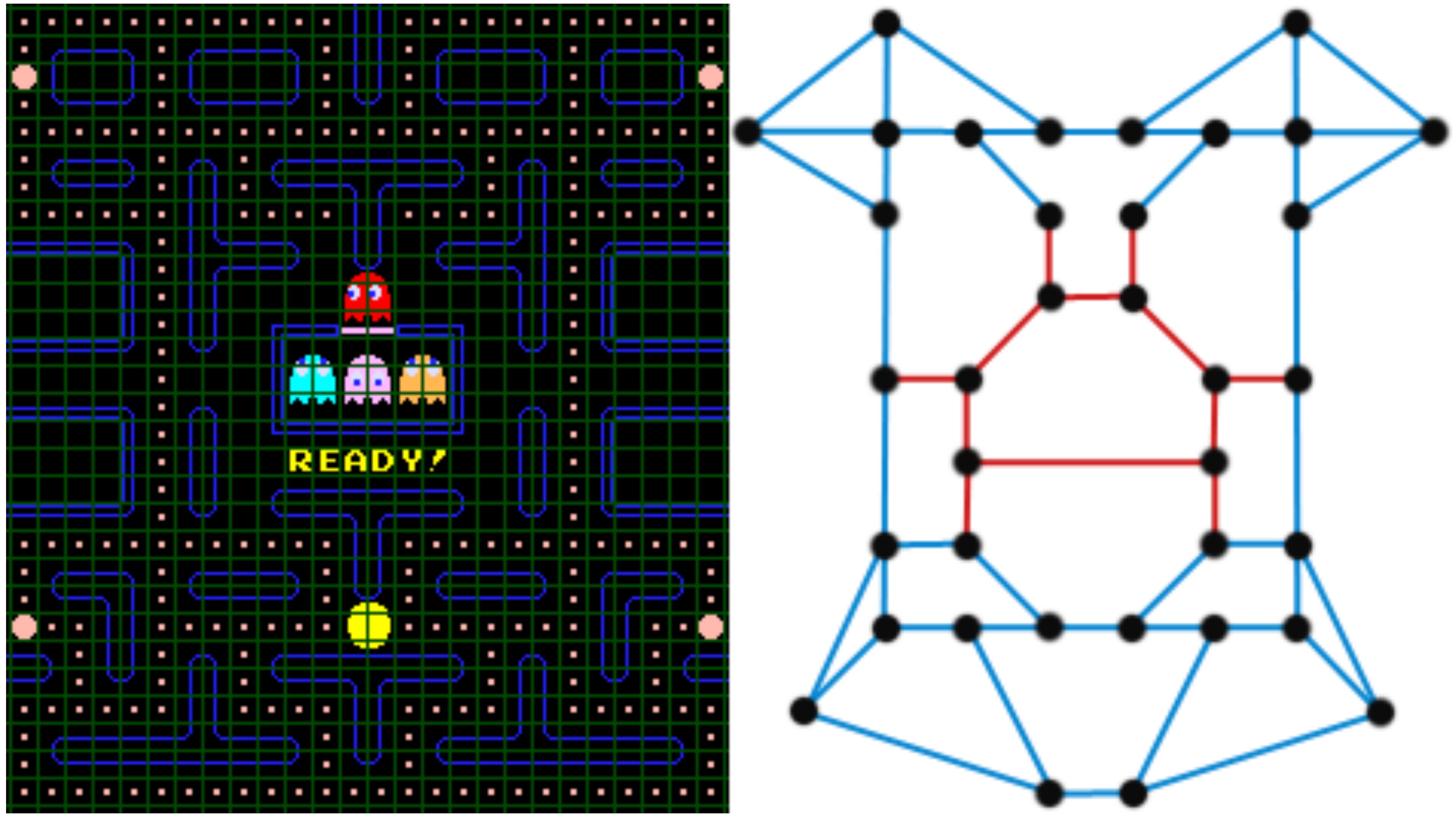
```java
public class Graph {
    private final int V;  // number of vertices
    private int E;        // number of edges
    private final List<Integer>[] adj;  // adjacency lists

    //init empty graph with V vertices and 0 edges
    @SuppressWarnings("unchecked")
    public Graph(int V) {
        this.V = V;
        this.E = 0;
        adj = (List<Integer>[]) new List[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new ArrayList<>();
        }
    }
```

```java
    //adds undirected edge v-w to graph. parallel edges and self-loops allowed
    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
        adj[w].add(v);
    }
    //returns vertices adjacent to vertex v
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
```

# Depth-first search

# Mazes as graphs

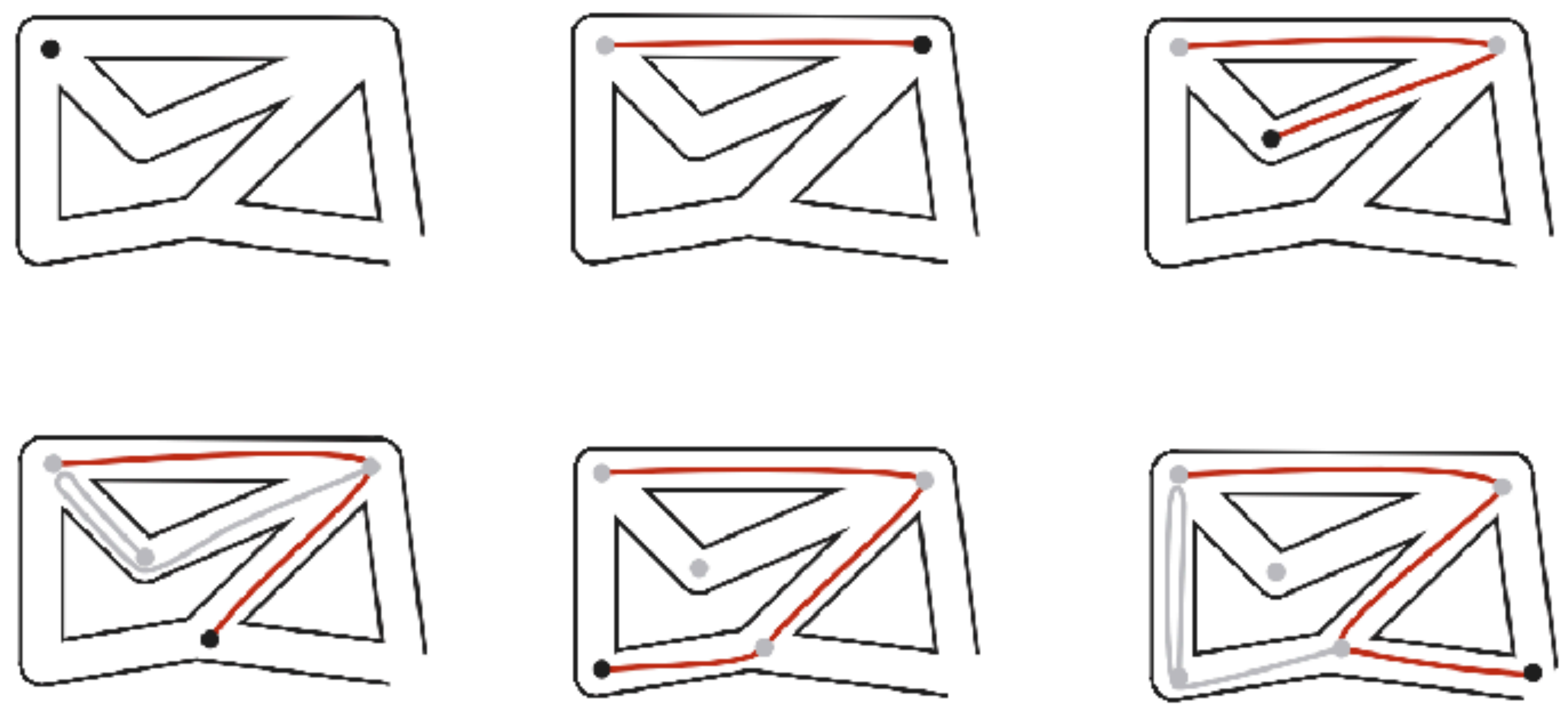
- Vertex = intersection; edge = passage

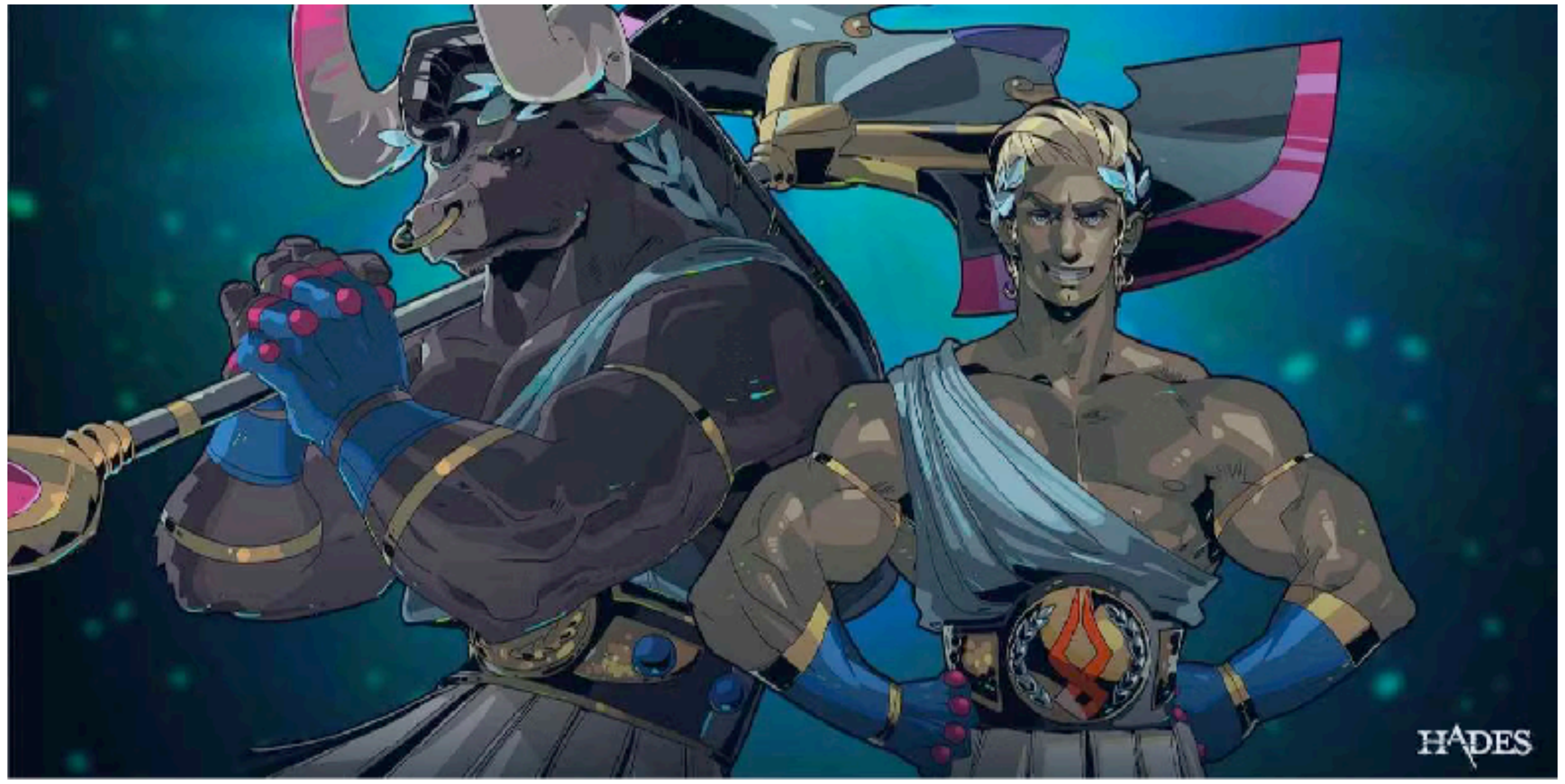



http://oatzy.blogspot.com/2011/09/playing-with-pac-man.html

# How to survive a maze: a lesson from a Greek myth

- Theseus escaped from the labyrinth after killing the Minotaur with the following strategy instructed by Ariadne:
  - Unroll a ball of string behind you.
  - Mark each newly discovered intersection and passage.
  - Retrace steps when no unmarked options.
- Also known as the Trémaux algorithm.





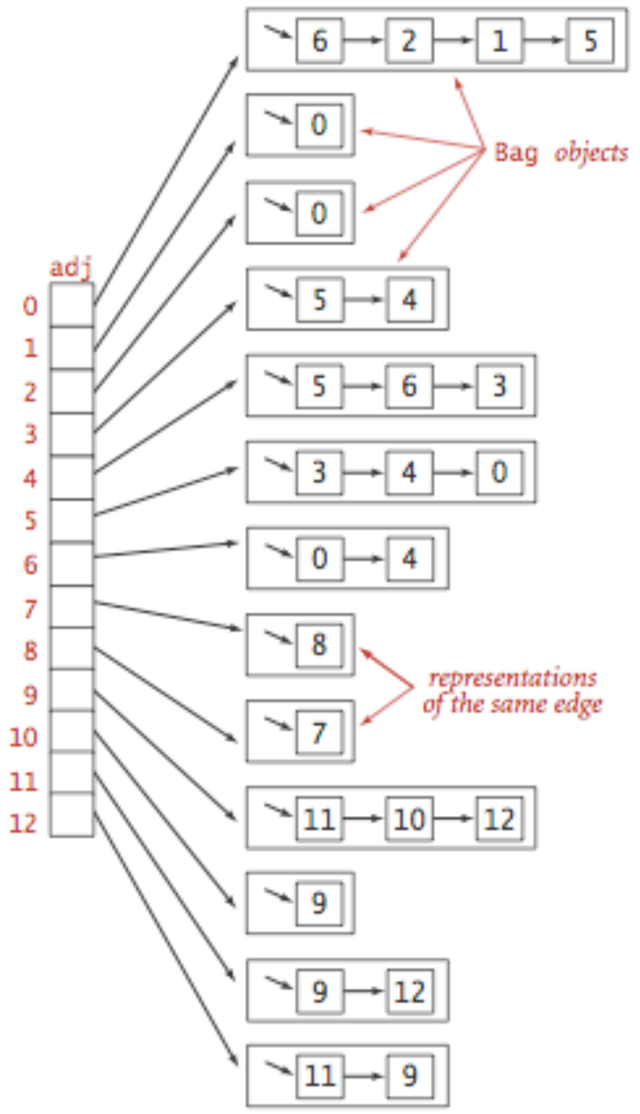
# Depth-first search

- Basic idea: Go deep in a graph until you can't anymore, visiting all vertices. Then retrace your steps.
- Goal: Systematically traverse a graph.
- DFS (to visit a vertex v)
  - Mark vertex v.
  - Recursively visit all unmarked vertices w adjacent to v.

- Typical applications:
  - Find all vertices connected to a given vertex.
  - Find a path between two vertices.


http://algs4.cs.princeton.edu

# 4.1 Depth-First Search Demo




Order visited: 0, 6, 4, 5, 3, 2, 1

# Depth-first search

- Goal: Find all vertices connected to s (and a corresponding path).
- Idea: Mimic maze exploration.
- Algorithm:
  - Use recursion (ball of string).
  - Mark each visited vertex (and keep track of edge taken to visit it).
  - Return (retrace steps) when no unvisited options.

- When started at vertex *s*, DFS marks all vertices connected to s (and no other).

# Implementation of depth-first search in Java

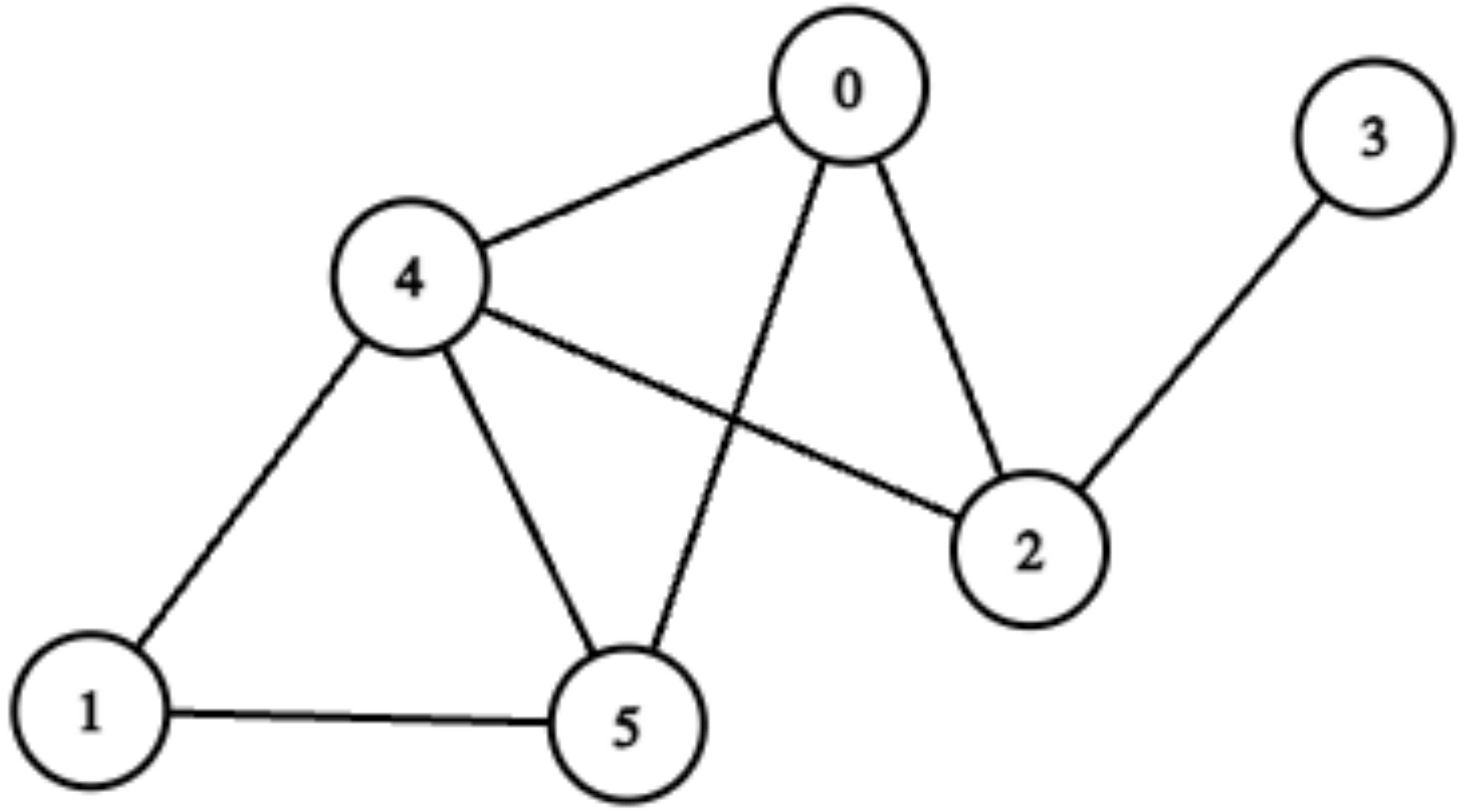
```
public void dfs(int s) {
    boolean[] marked = new boolean[V]; //marked[v] - is there an s-v path?
    int[] edgeTo = new int[V]; //edgeTo[v] = previous vertex on path from s to v
    int[] distTo = new int[V]; //distTo[v] - distance from s to v

    for (int i = 0; i < V; i++) {
        distTo[i] = -1;  // initialize distances to -1
    }

    marked[s] = true;
    distTo[s] = 0;
    dfsHelper(s, marked, edgeTo, distTo);
```

```
private void dfsHelper(int v, boolean[] marked, int[] edgeTo, int[] distTo) {
    for (int w : adj[v]) {
        if (!marked[w]) {
            marked[w] = true;
            edgeTo[w] = v;
            distTo[w] = distTo[v] + 1;
            dfsHelper(w, marked, edgeTo, distTo);
        }
    }
}
```

for each adjacent vertex, mark it and call DFS on it

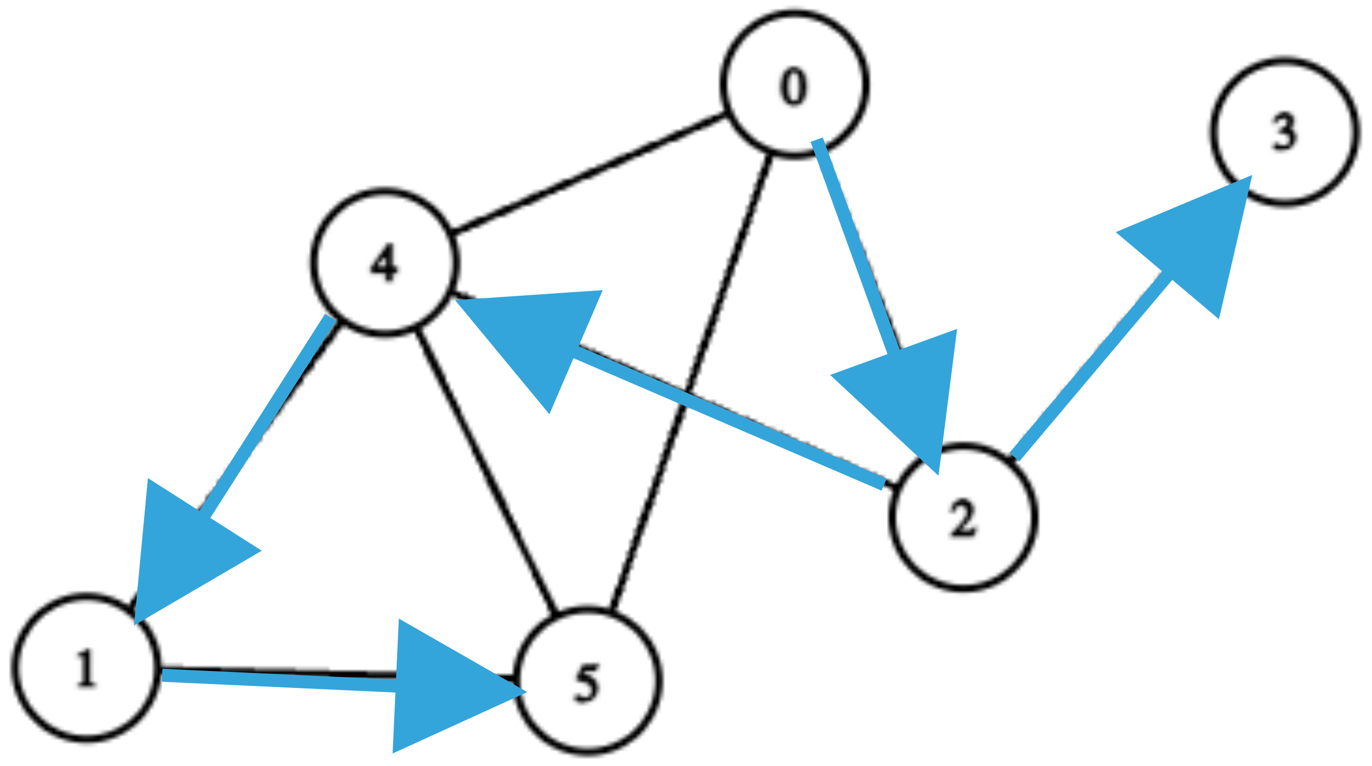
# ***Worksheet time!***

- Run DFS on the following graph starting at vertex 0 and return the **vertices in the order of being marked.** Assume that the adjacent vertices are returned in increasing numerical order.



# Worksheet answers

- Vertices marked as visited: 0, 2, 3, 4, 1, 5




| V | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 4 |
| 2 | T | 0 |
| 3 | T | 2 |
| 4 | T | 2 |
| 5 | T | 1 |

# Depth-first search analysis

- DFS marks all vertices connected to `s` in time proportional to $|V| + |E|$ in the worst case.
  - Initializing arrays `marked` and `edgeTo` takes time proportional to $|V|$.
  - Each adjacency-list entry is examined exactly once and there are $2|E|$ such entries (two for each edge in an undirected graph).
- Once we run DFS, we can check if vertex `v` is connected to `s` in constant time (look into the marked array). We can also find the `v`-`s` path (if it exists) in time proportional to its length.

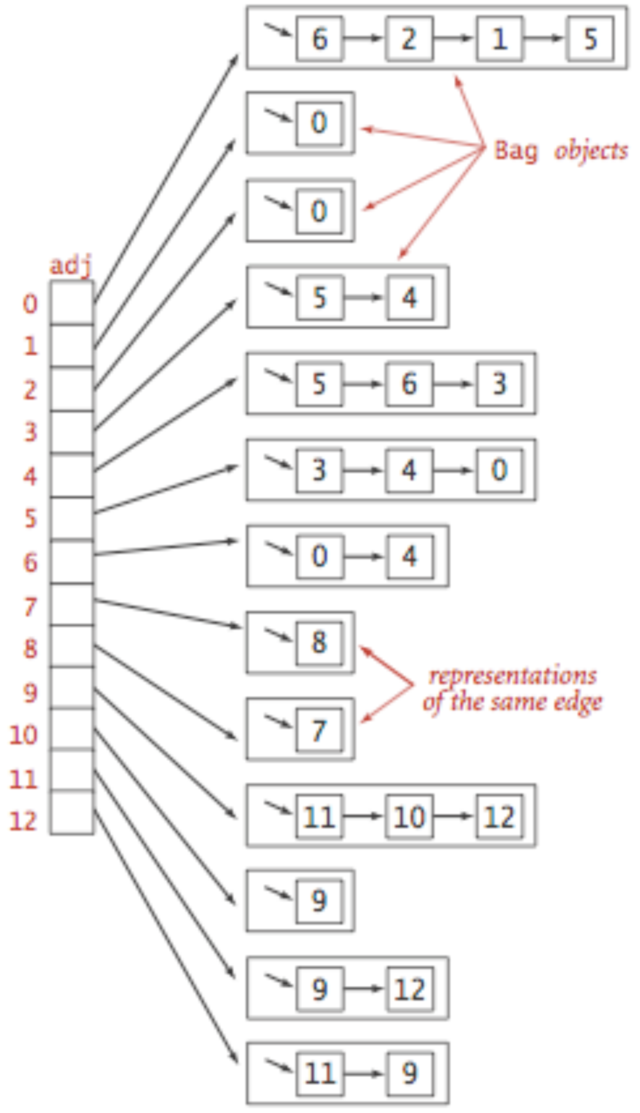
# Breadth-first search

# Breadth-first search

- Basic idea: BFS traverses vertices in order of distance from s. (All of s's adjacent vertices get seen first, then the ones 2 away, then the ones 3 away…)
- BFS (from source vertex s)
  - Put s  on a queue and mark it as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex v.
    - Enqueue each of v's unmarked neighbors and mark them.


http://algs4.cs.princeton.edu

# 4.1 Breadth-First Search Demo




Order visited: 0, 2, 1, 5, 3, 4

# Breadth-first search in Java

```java
public void bfs(int s) {
    boolean[] marked = new boolean[V];
    int[] edgeTo = new int[V];
    int[] distTo = new int[V];

    Queue<Integer> queue = new LinkedList<>();
    marked[s] = true;
    distTo[s] = 0;
    queue.add(s);

    while (!queue.isEmpty()) {
        int v = queue.remove();
        for (int w : adj[v]) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
                queue.add(w);
            }
        }
    }
}
```

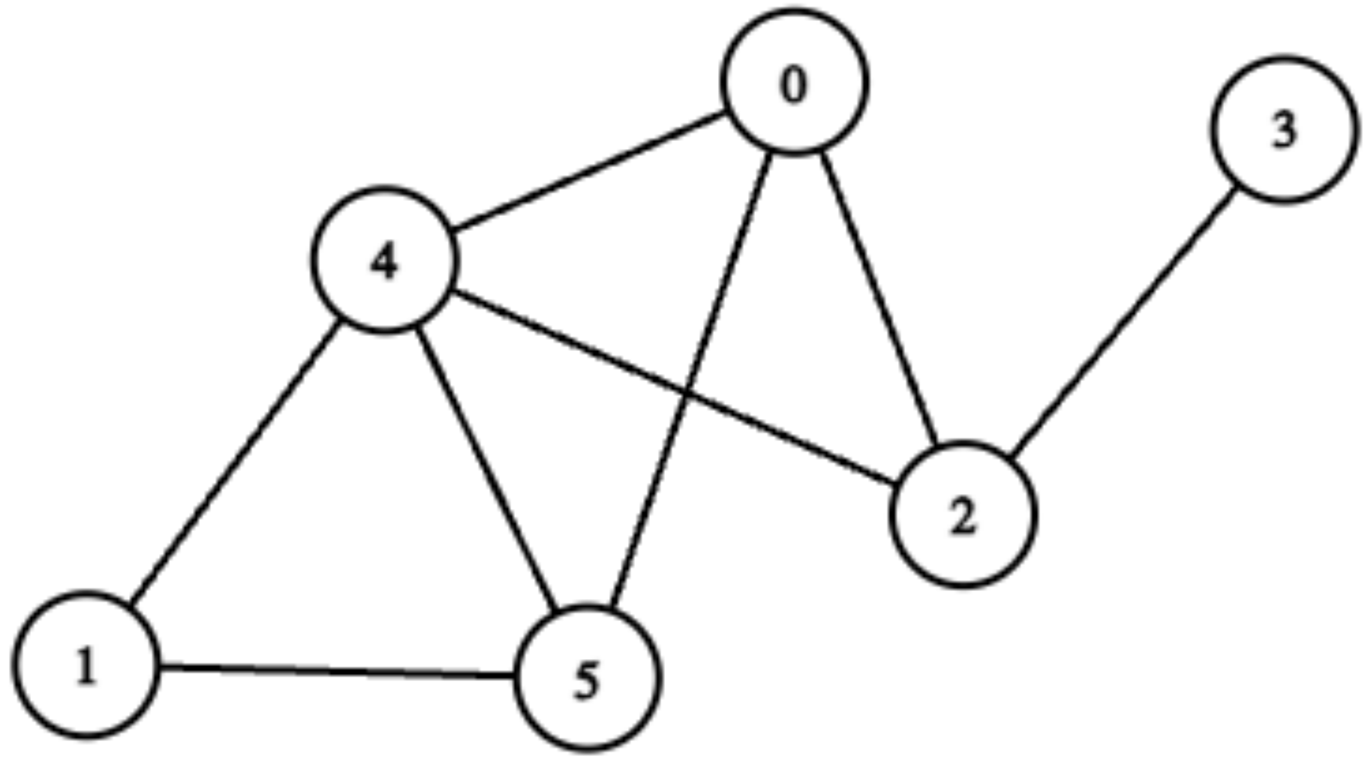
enqueue s

dequeue v

enqueue adjacent vertices, w

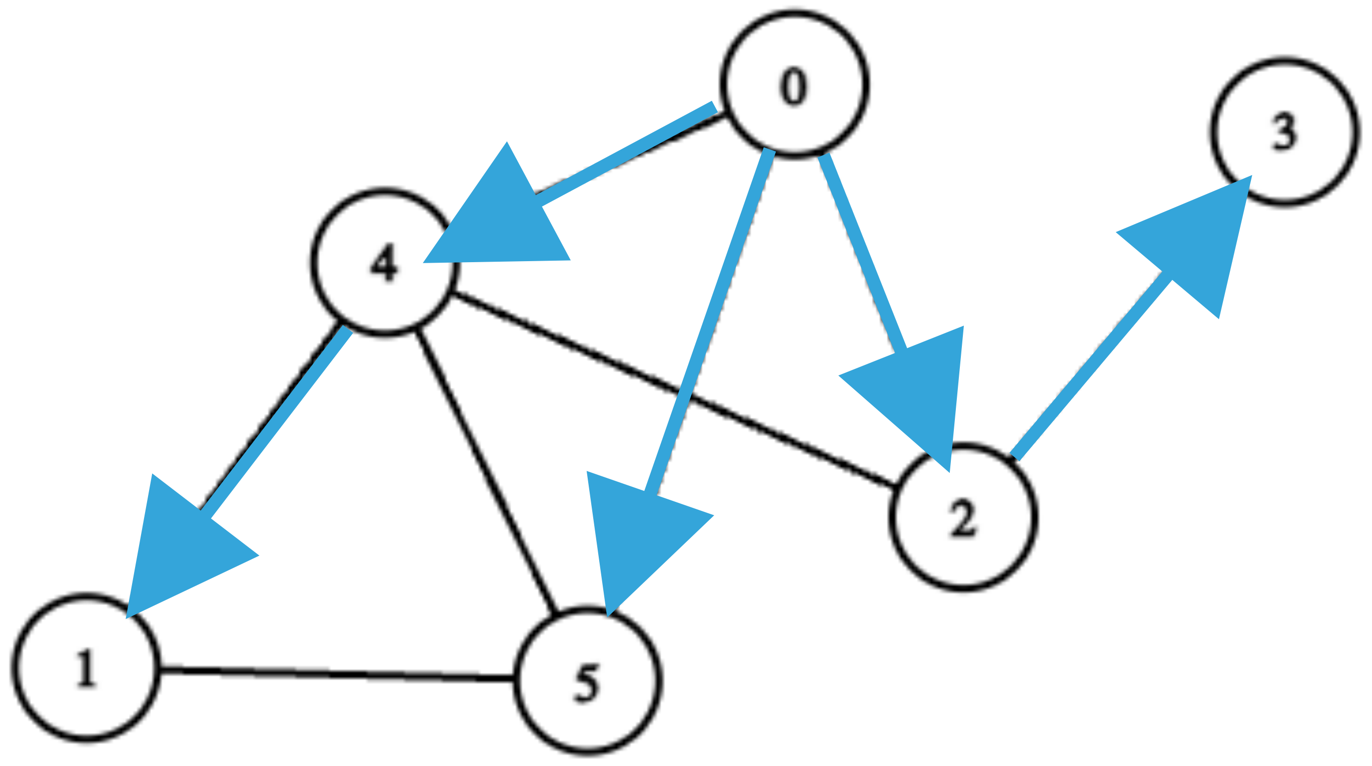
# *Worksheet time!*

- Run the BFS on the following graph starting at vertex 0 and return the vertices in the order of being marked. Assume the adjacent vertices are returned in increasing numerical order.



# Worksheet answers

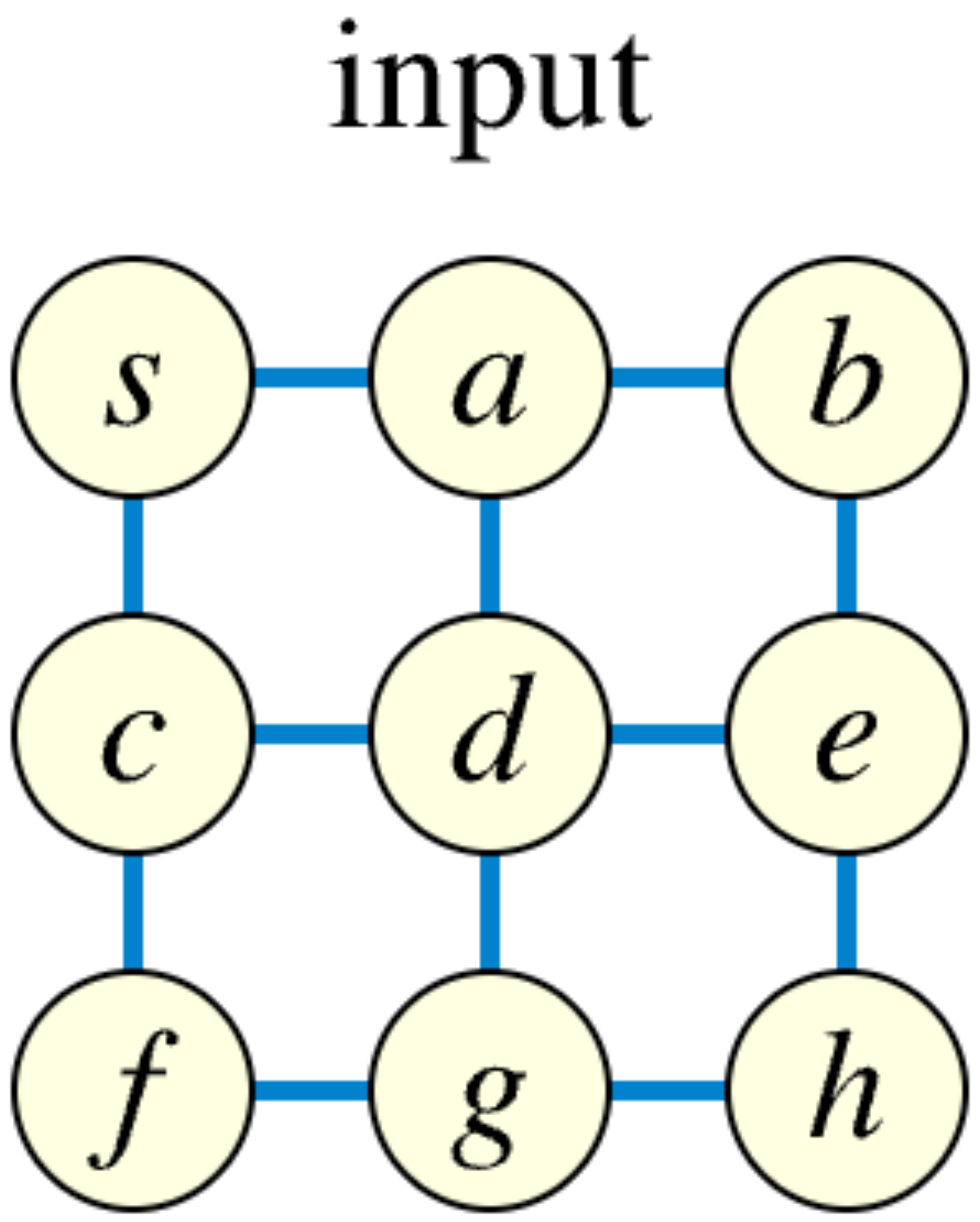
Vertices marked as visited: 0, 2, 4, 5, 3, 1




| V | marked | edgeTo | distTo |
|---|---|---|---|
| 0 | T | - | 0 |
| 1 | T | 4 | 2 |
| 2 | T | 0 | 1 |
| 3 | T | 2 | 2 |
| 4 | T | 0 | 1 |
| 5 | T | 0 | 1 |

# *Worksheet time!*

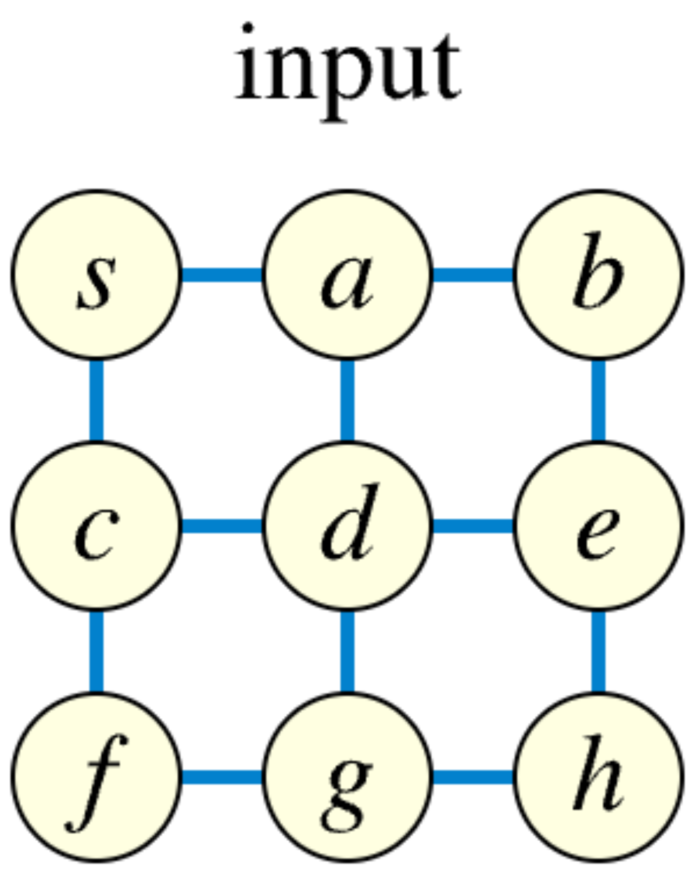
- Run DFS and BFS on the following graph starting at vertex s. Assume the adjacent vertices are returned in lexicographic (i.e., alphabetical) order.

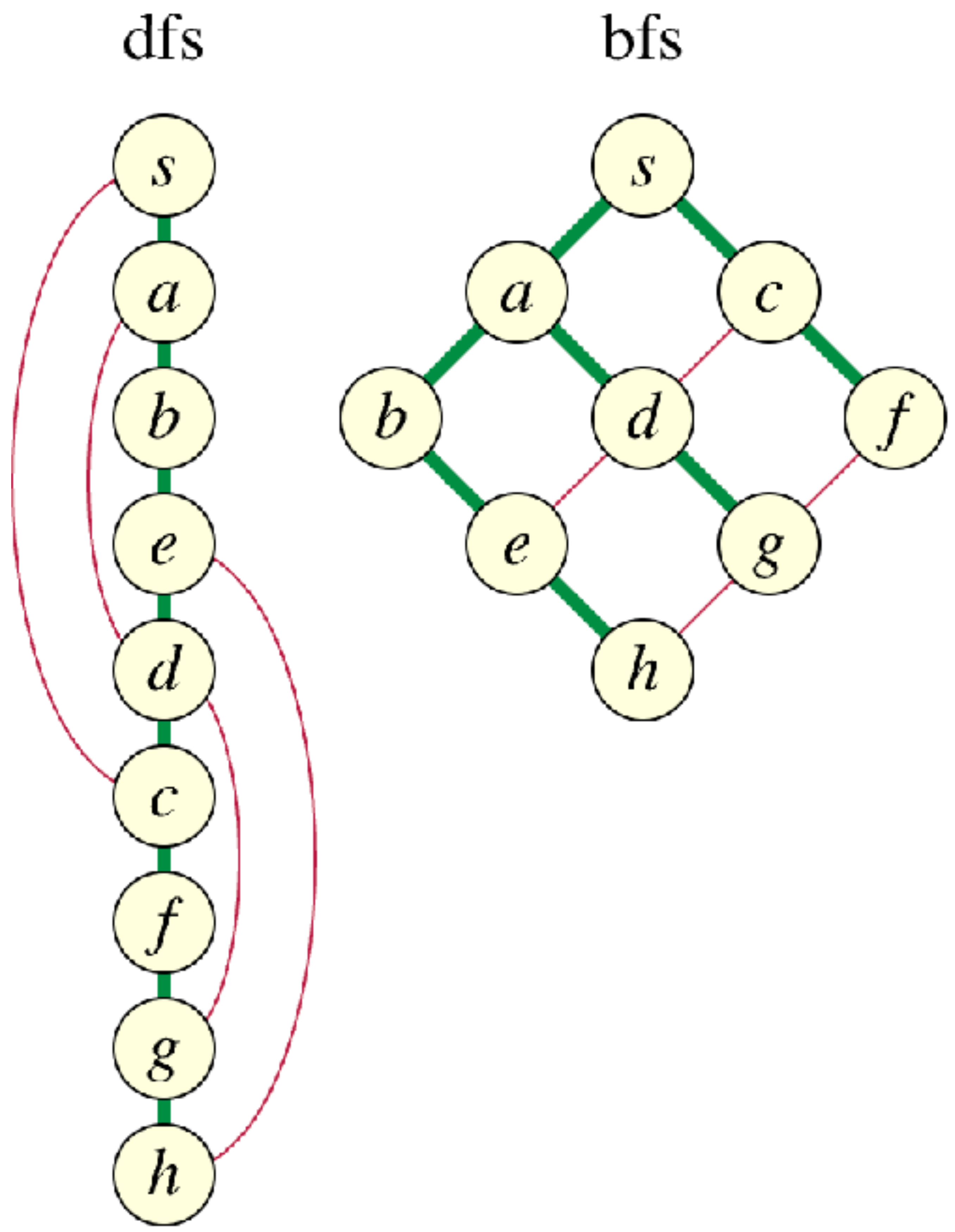



https://11011110.github.io/blog/2013/12/17/stack-based-graph-traversal.html

# Worksheet answer

- Run DFS and BFS on the following graph starting at vertex s. Assume that the adj method returns back the adjacent vertices in lexicographic order.
- DFS: s->a->b->e->d->c->f->g->h
- BFS: s->a->c->b->d->f->e->g->h



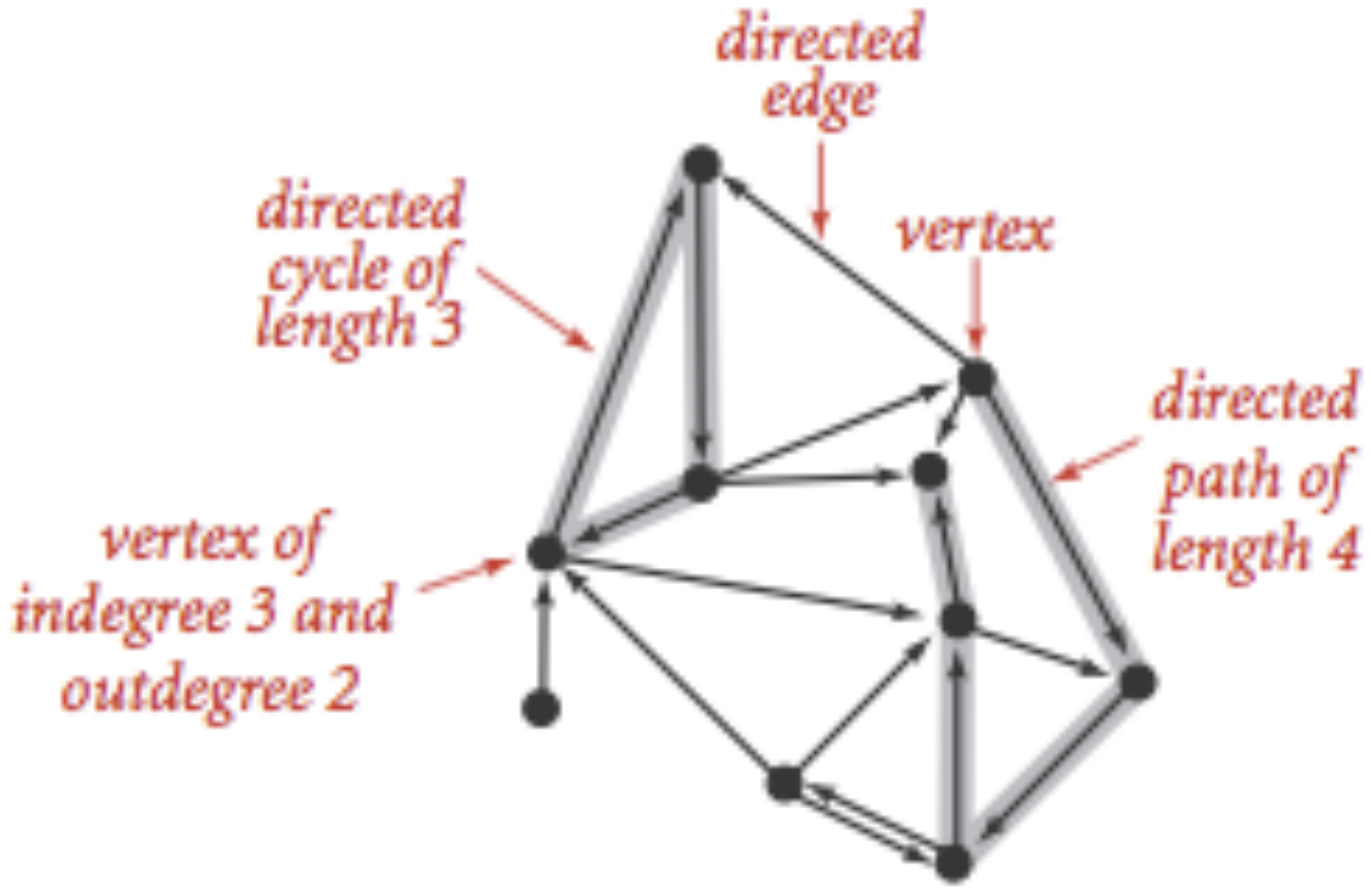
# Summary

- DFS: Uses recursion.
- BFS: Put unvisited vertices on a queue.
- Shortest path problem: Find path from `s` to `t` that uses the fewest number of edges.
  - E.g., calculate the fewest numbers of hops in a communication network.
  - E.g., calculate the Kevin Bacon number or Erdös number.
- **BFS computes shortest paths** from `s` to all vertices in a graph in time proportional to $|E| + |V|$
  - The queue always consists of zero or more vertices of distance `k` from `s`, followed by zero or more vertices of `k+1`.
  - DFS, on the other hand, will find *a path*, but it's not guaranteed to be the shortest one.

# Directed graphs

# Directed Graph Terminology

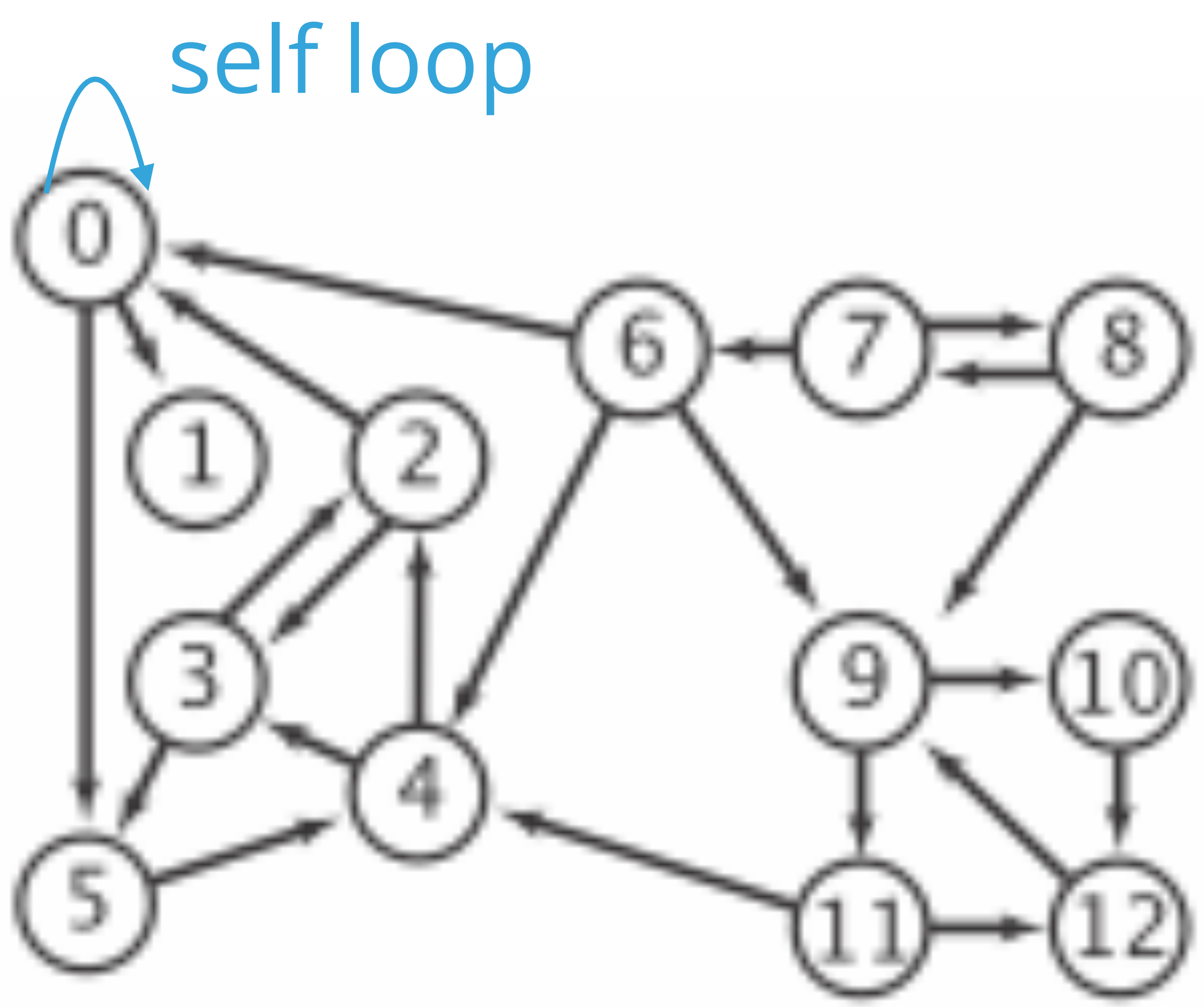
- Directed Graph (digraph) : a set of vertices V connected pairwise by a set of directed edges E.
- Directed path: a sequence of vertices in which there is a directed edge pointing from each vertex in the sequence to its successor in the sequence, with no repeated edges. (Basically just a path in the graph.)
  - A simple directed path is a directed path with no repeated vertices.
- Directed cycle: Directed path with at least one edge whose first and last vertices are the same.
  - A simple directed cycle is a directed cycle with no repeated vertices (other than the first and last).
- The length of a cycle or a path is its number of edges.




Anatomy of a digraph

# Directed Graph Terminology

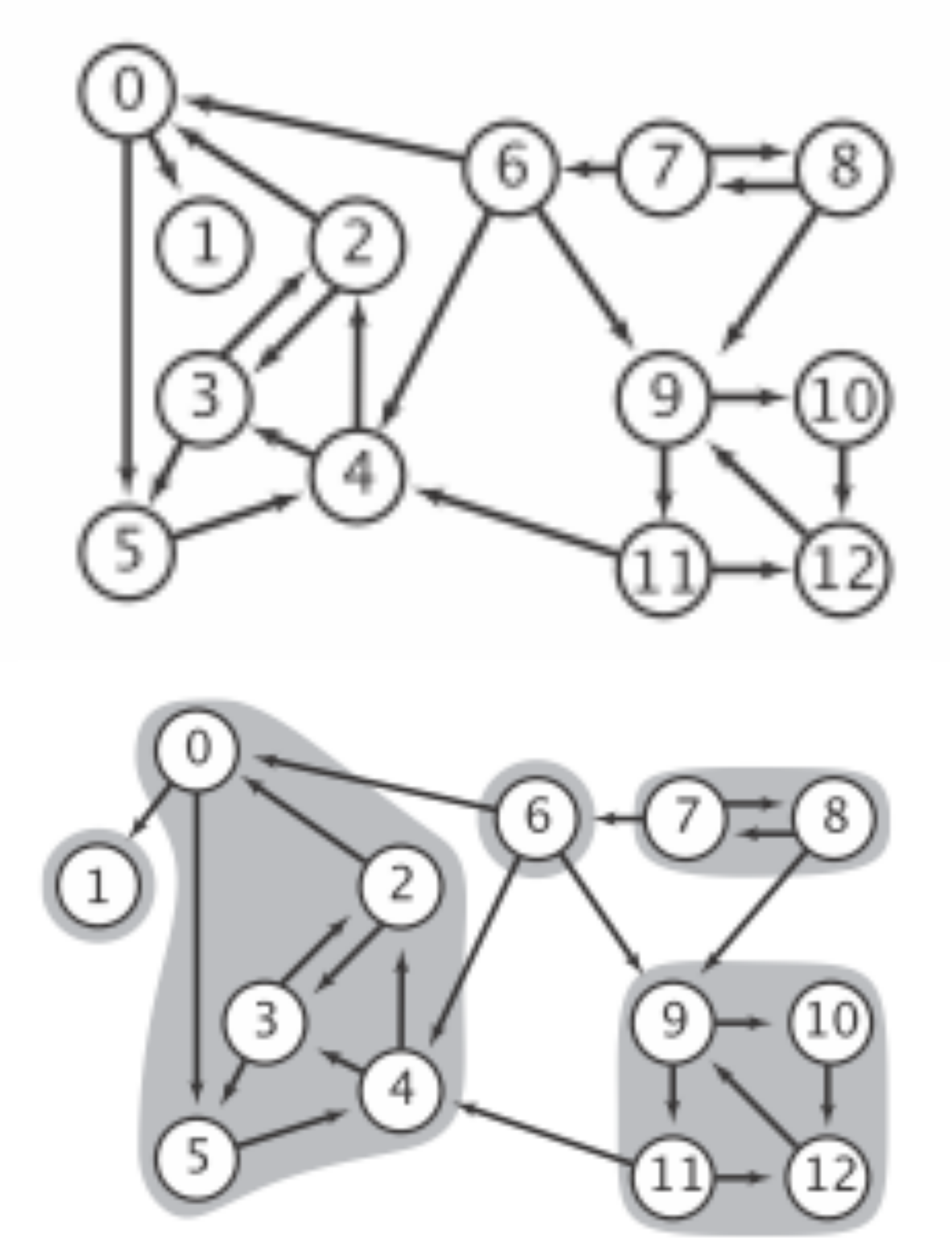
- Self-loop: an edge that connects a vertex to itself.
- Two edges are parallel if they connect the same pair of vertices.
- The outdegree of a vertex is the number of edges pointing from it.
- The indegree of a vertex is the number of edges pointing to it.
- A vertex w is reachable from a vertex v if there is a directed path from v to w.
- Two vertices v and w are strongly connected if they are mutually reachable.



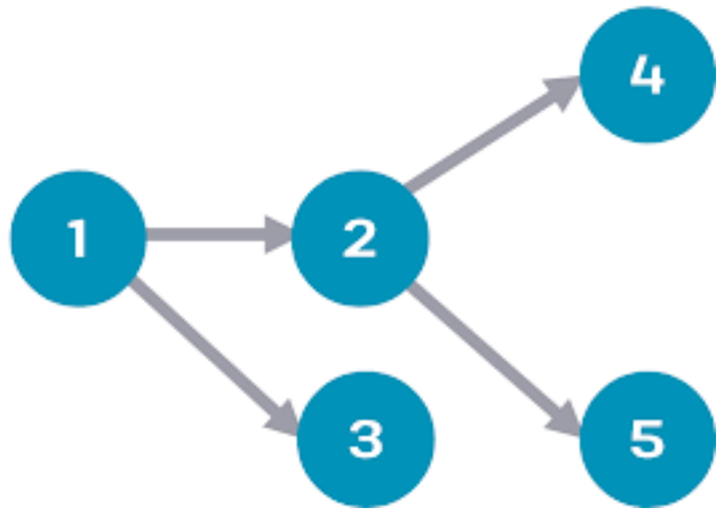

```
V =
{0,1,2,3,4,5,6,7,8,9,10,11,12
}

E = {{0,0}, {0,1}, {0,5},
{2,0}, {2,3},{3,2},{3,5},
{4,2},{4,3},{5,4},{6,0},
{6,4},{6,9},{7,6}{7,8},{8,7},
{8,9},{9,10},{9,11},{10,12},
{11,4},{11,12},{12,9}}.
```

# Directed Graph Terminology

- A digraph is strongly connected if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
- A directed acyclic graph (DAG) is a digraph with no directed cycles.



A digraph and its strong components



# Digraph Applications

| Digraph | Vertex | Edge |
|---|---|---|
| Web | Web page | Link |
| Cell phone | Person | Placed call |
| Financial | Bank | Transaction |
| Transportation | Intersection | One-way street |
| Game | Board | Legal move |
| Citation | Article | Citation |
| Infectious Diseases | Person | Infection |
| Food web | Species | Predator-prey relationship |

# Popular digraph problems

| Problem | Description |
| --- | --- |
| s->t path | Is there a path from s to t? |
| Shortest s->t path | What is the shortest path from s to t? |
| Directed cycle | Is there a directed cycle in the digraph? |
| Topological sort | Can vertices be sorted so all edges point from earlier to later vertices? |
| Strong connectivity | Is there a directed path between every pair of vertices? |

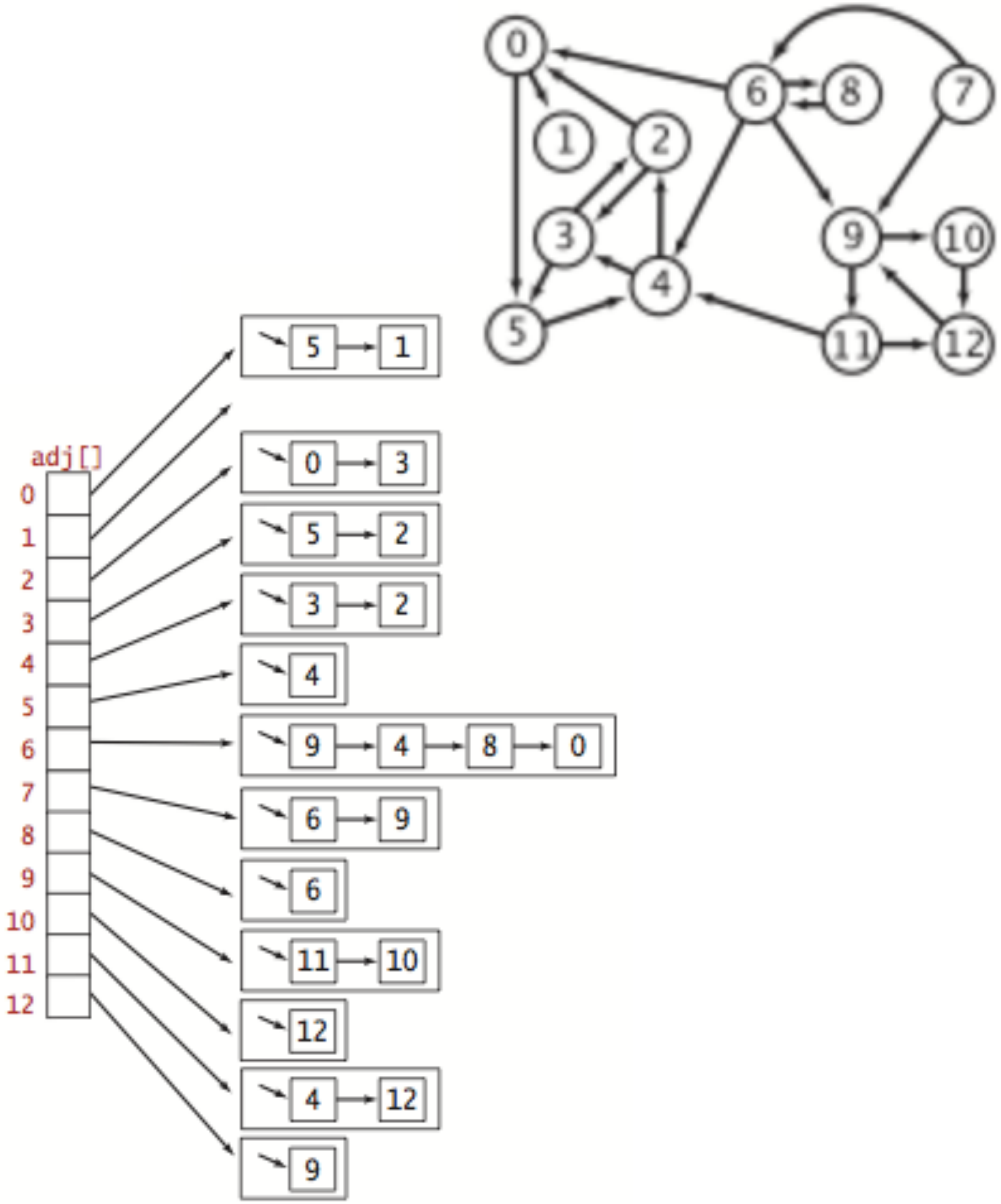
# Basic Graph API

`public class Digraph`

- `Digraph(int V)`: create an empty digraph with V vertices.
- `void addEdge(int v, int w)`: add an edge v->w.
- `Iterable<Integer> adj(int v)`: return vertices adjacent from v.
- `int V()`: number of vertices.
- `int E()`: number of edges.
- `Digraph reverse()`: reverse edges of digraph.

# Digraph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent from $v$.
- Space efficient ($|E| + |V|$).
- Constant time for adding a directed edge.
- **New difference:** Lookup of a directed edge or iterating over vertices adjacent from $v$ is $outdegree(v)$.



# Adjacency-list digraph representation in Java

```java
public class DirectedGraph {
    private final int V;
    private int E;
    private final List<Integer>[] adj;

    @SuppressWarnings("unchecked")
    public DirectedGraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (List<Integer>[]) new List[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new ArrayList<>();
        }
    }

    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w); // Directed edge from v to w
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

```

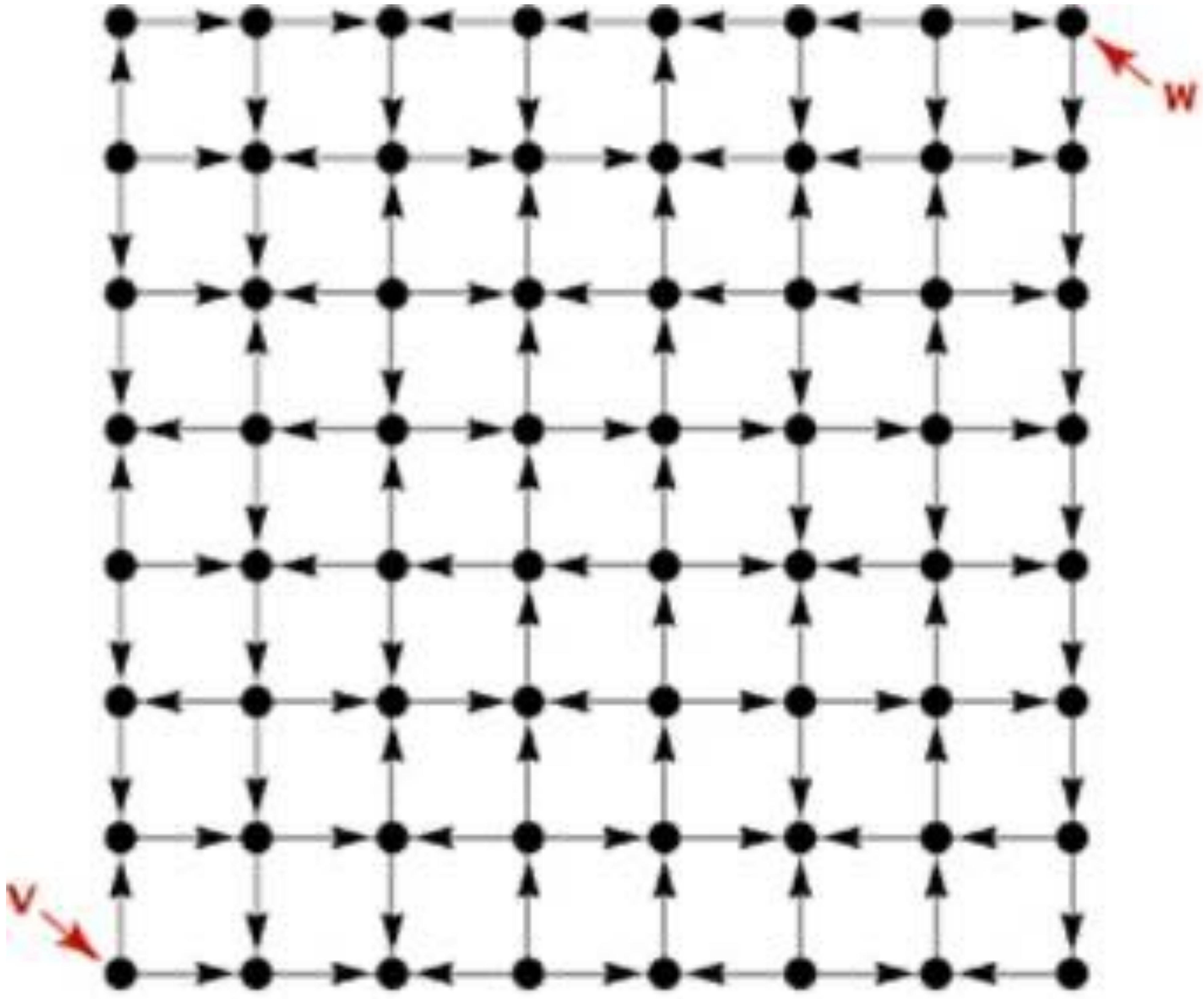
Very similar to undirected graph implementation, main change is adding directed edges (1 edge, not 2)

```java
//adds undirected edge v-w to graph. parallel edges and self-loops allowed
public void addEdge(int v, int w) {
    E++;
    adj[v].add(w);
    adj[w].add(v);
}
```

# DFS in Directed graphs

# Reachability

- Find all vertices reachable from s along a directed path.



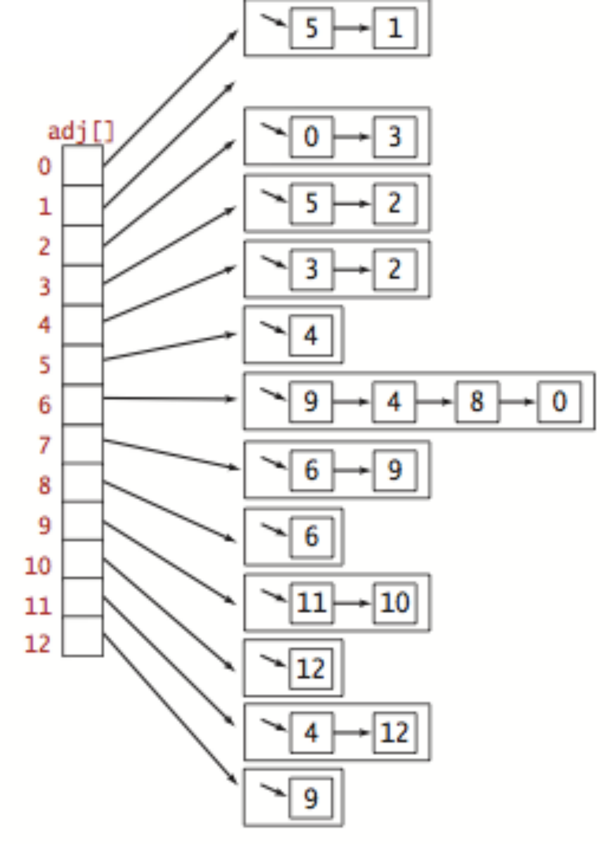

Is w reachable from v in this digraph?

https://apprize.info/science/algorithms_2/2.html

# Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
  - Maximum number of edges in a simple digraph is $n(n-1)$.
- DFS (to visit a vertex v)
  - Mark vertex v.
  - Recursively visit all unmarked vertices w adjacent from v.
- Typical applications:
  - Find a directed path from source vertex s to a given target vertex v.
  - Topological sort (sort so dependencies are ordered, e.g. for fulfilling course pre-reqs).
  - Directed cycle detection.

Algorithms    Robert Sedgewick | Kevin Wayne



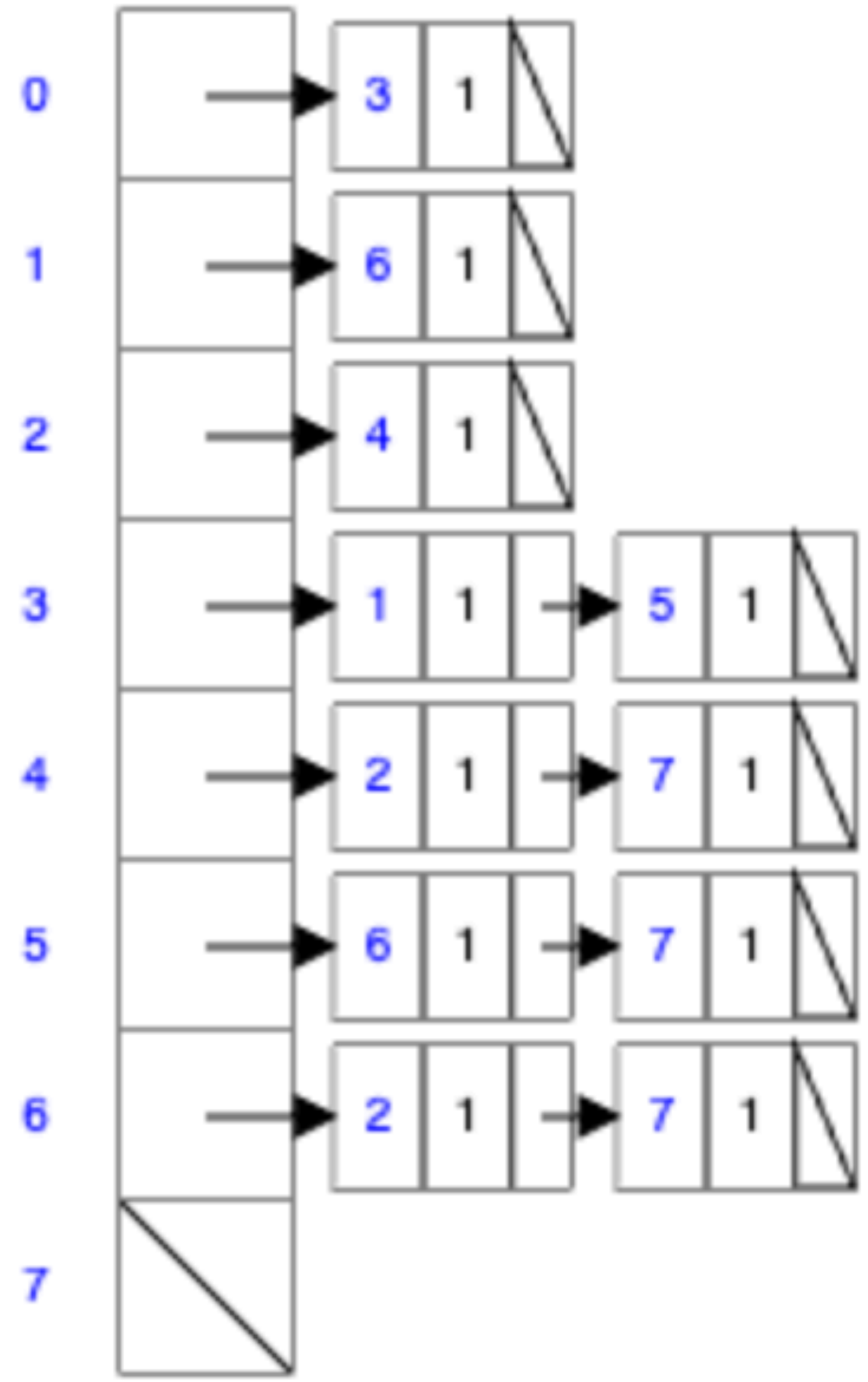


http://algs4.cs.princeton.edu

# 4.2 Directed DFS Demo

# Worksheet time!

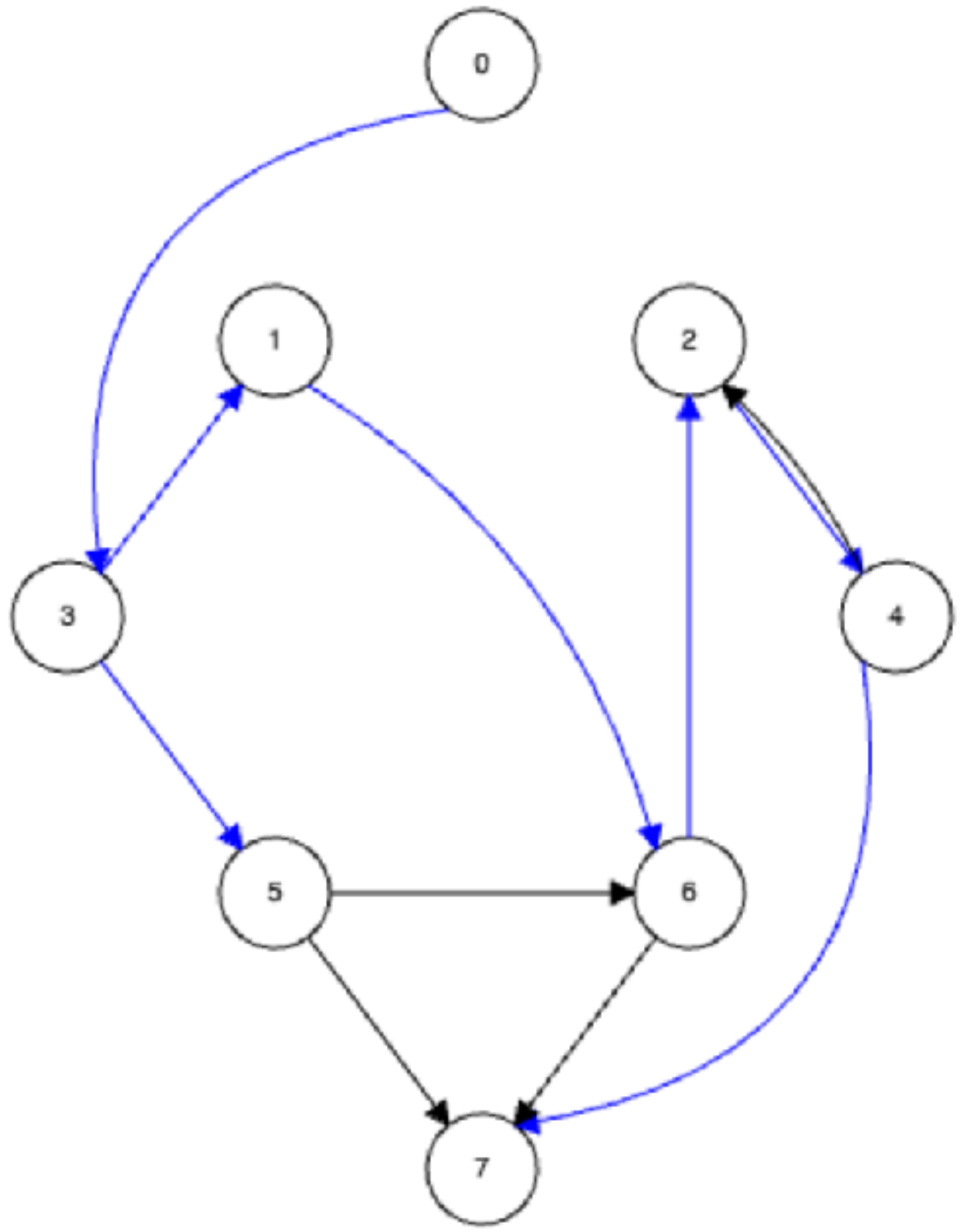
- Given the following adjacency list, visualize the resulting digraph and run DFS on it starting at vertex 0. In what order did you visit the vertices?



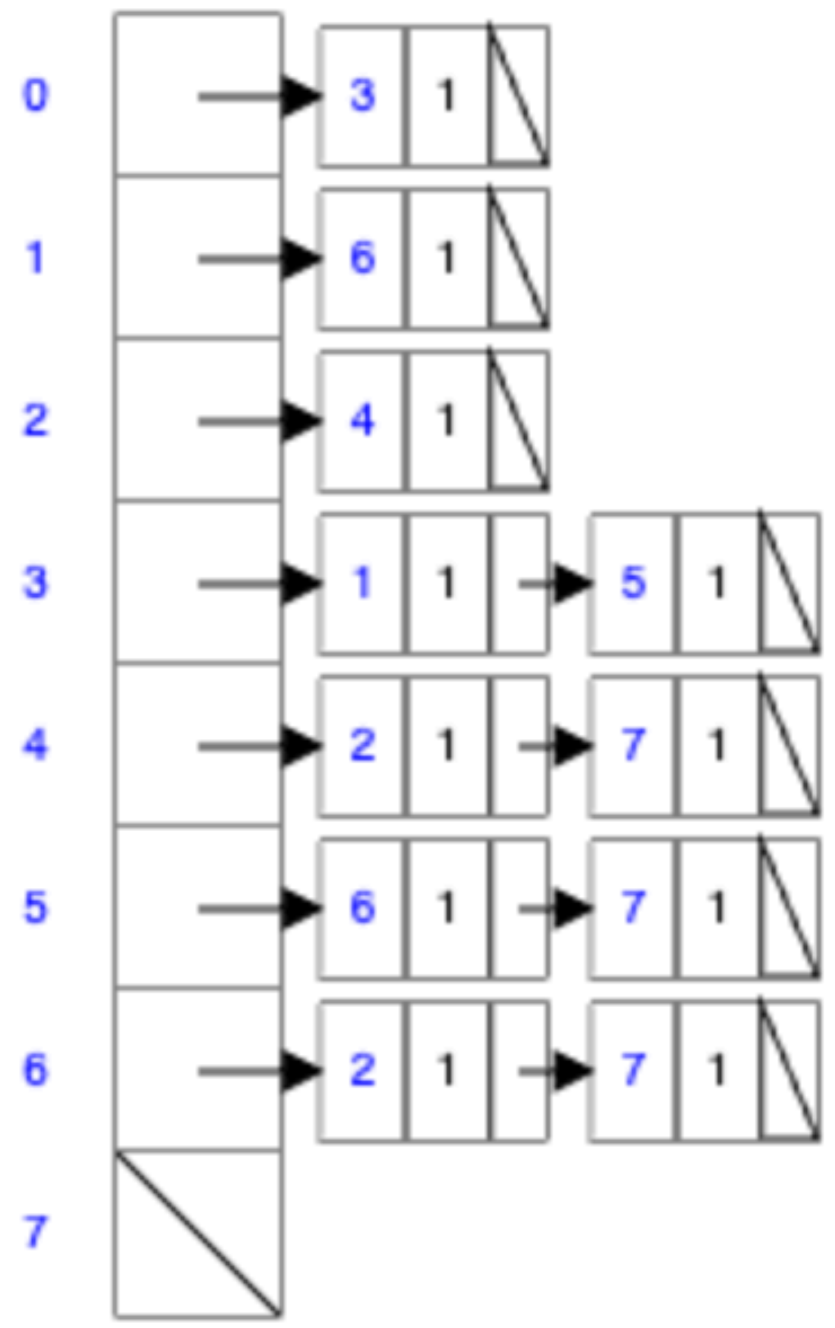

Note: Ignore the “1” value

# Worksheet answer

- Given the following adjacency list, visualize the resulting digraph and run DFS on it starting at vertex 0.




Order: 0, 3, 1, 6, 2, 4, 7, 5




| V | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 3 |
| 2 | T | 6 |
| 3 | T | 0 |
| 4 | T | 2 |
| 5 | T | 3 |
| 6 | T | 1 |
| 7 | T | 4 |

# Depth-first search analysis

- DFS marks all vertices reachable from `s` in time proportional to $|V| + |E|$ in the worst case.
  - Initializing arrays `marked` takes time proportional to $|V|$.
  - Each adjacency-list entry is examined exactly once and there are $E$ such edges (different than undirected graphs, which have 2|E| edges).
- Once we run DFS, we can check if vertex `v` is reachable from `s` in constant time (look into the marked array). We can also find the `s->v` path (if it exists) in time proportional to its length.

# BFS in Directed graphs

# Breadth-first search

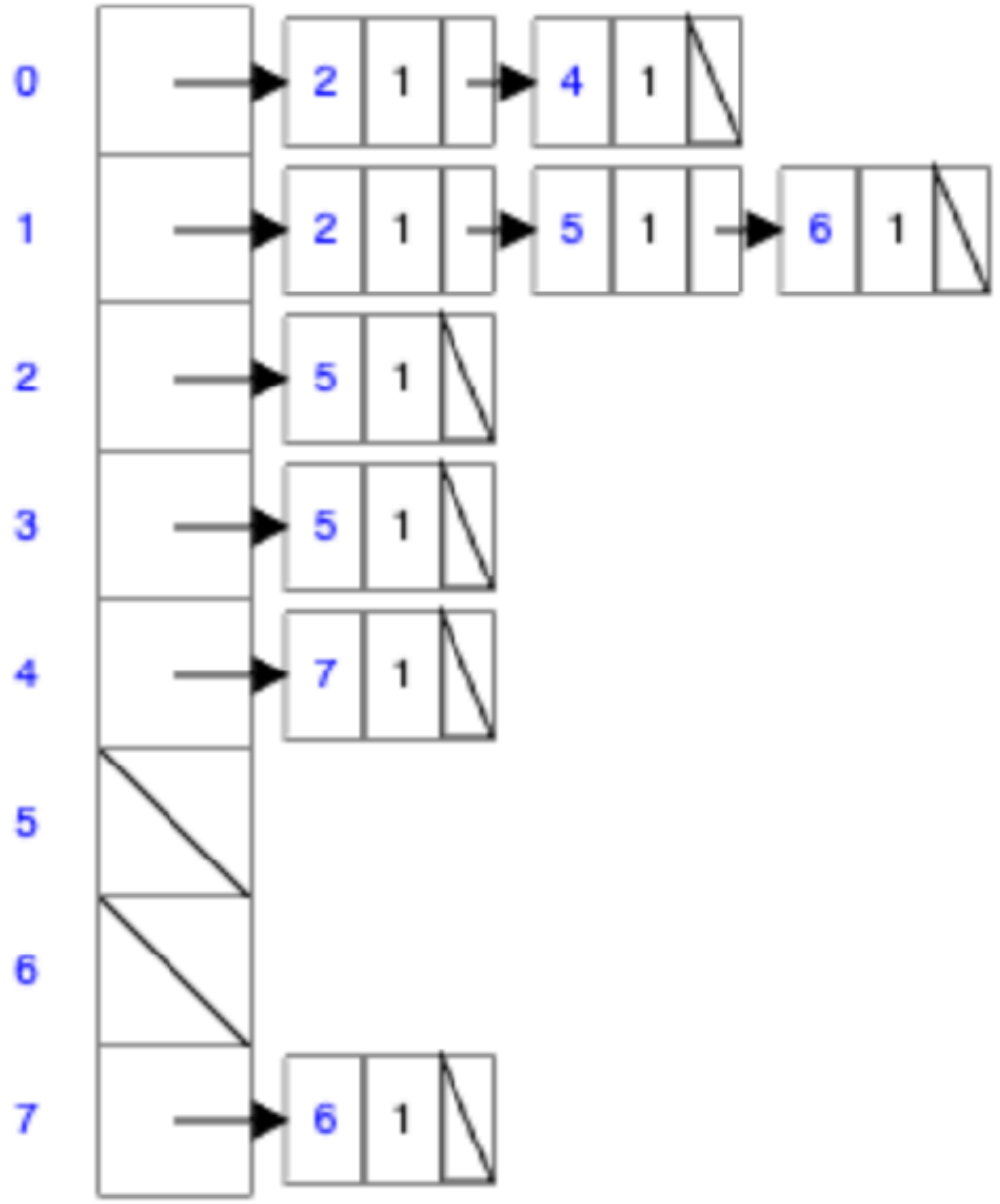
- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
- BFS (from source vertex s)
  - Put s on queue and mark s as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex v.
    - Enqueue all unmarked vertices adjacent from v, and mark them.
- Typical applications:
  - Find the shortest (in terms of number of edges) directed path between two vertices in time proportional to $|E| + |V|$.


http://algs4.cs.princeton.edu

# 4.2 DIRECTED BFS DEMO

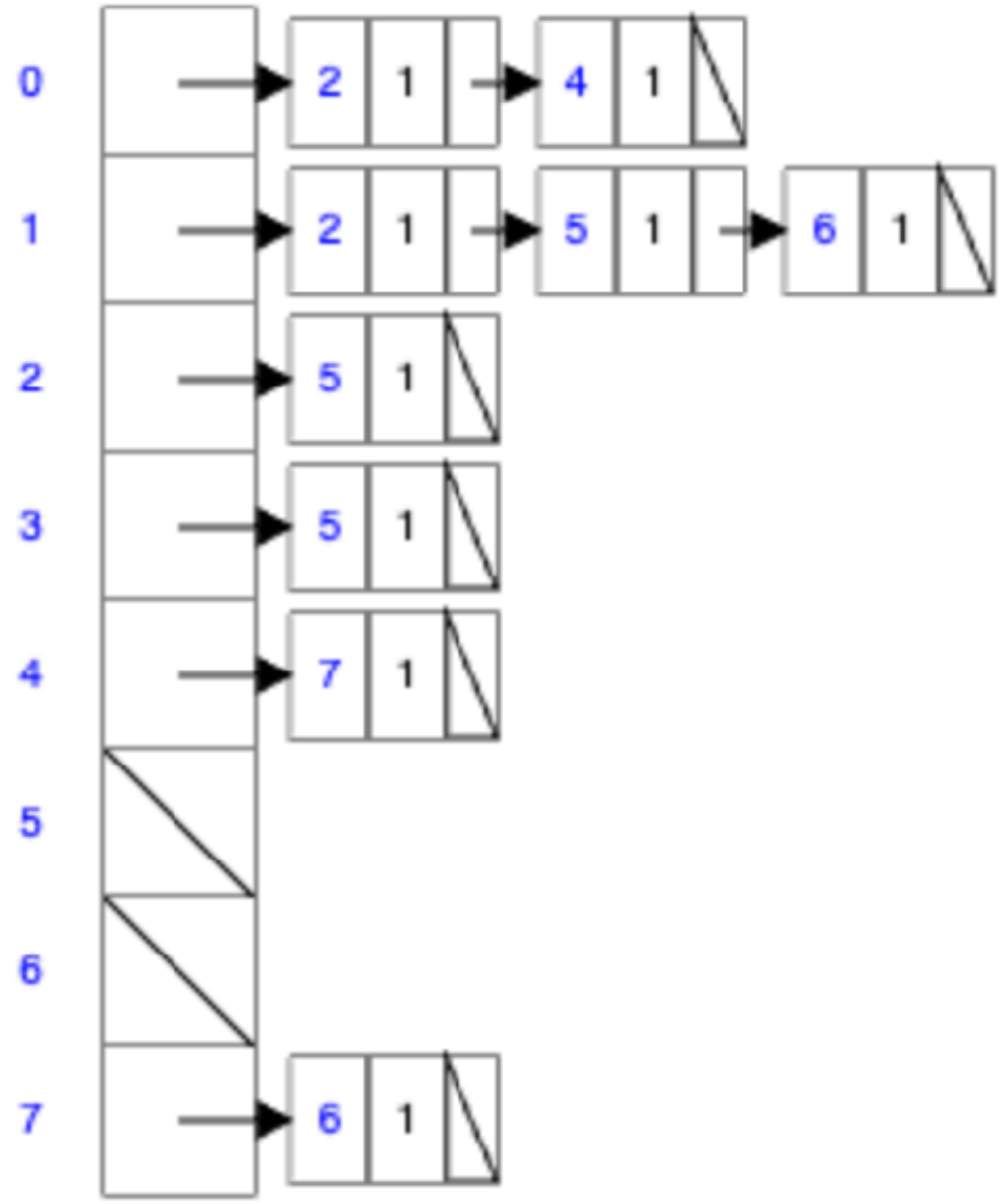
# *Worksheet time!*

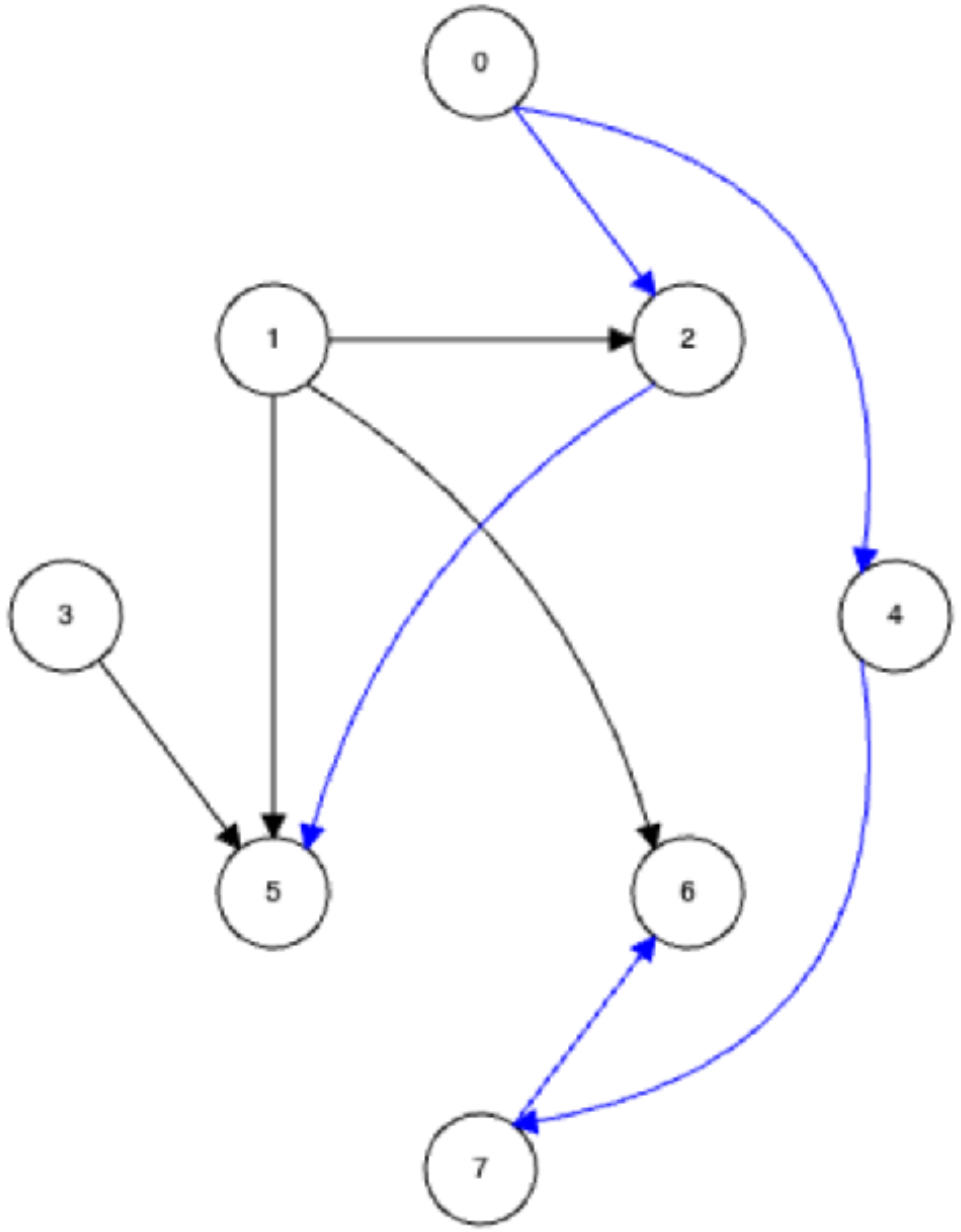
- Given the following adjacency list, visualize the resulting digraph and run BFS on it starting at vertex 0. In what order did you visit the vertices?



# Worksheet answer

- Given the following adjacency list, visualize the resulting digraph and run BFS on it starting at vertex 0. In what order did you visit the vertices?
- 0, 2, 4, 5, 7, 6




| V | marked | edgeTo | distTo |
|---|---|---|---|
| 0 | T | - | 0 |
| 1 | F | | |
| 2 | T | 0 | 1 |
| 3 | F | | |
| 4 | T | 0 | 1 |
| 5 | T | 2 | 2 |
| 6 | T | 7 | 3 |
| 7 | T | 4 | 2 |

# Summary

- Single-source reachability in a digraph: DFS/BFS.
- Shortest path in a digraph: BFS.

# Algorithm: Is a digraph strongly connected?

- A strongly connected digraph is a directed graph in which it is possible to reach any vertex starting from any other vertex by traversing edges.
- Pick a random starting vertex s.
- Run DFS/BFS starting at s.
  - If have not reached all vertices, return false.
- Reverse edges.
- Run DFS/BFS again on reversed graph.
  - If have not reached all vertices, return false.
  - Else return true.

# Lecture 22 wrap-up

- Exit ticket: https://forms.gle/JYzmdDxt58XcrBJ28
- HW9: Transplant Manager due next Tues 11:59pm
- Final project (groups of 2-3 or 3-4…haven't decided yet) released in lab next week

# Resources

- Recommended Textbook: Chapter 4.1 (Pages 522-556), Chapter 4.2 (Pages 566-594)
- Website: https://algs4.cs.princeton.edu/41graph/, https://algs4.cs.princeton.edu/42digraph/
- Visualization: https://visualgo.net/en/dfsbfs
- Practice problems (3!) behind this slide

# Problem 1

- What is the maximum number of edges in an undirected graph with V vertices and no parallel edges?
- What is the minimum number of edges in an undirected graph with V vertices, none of which are isolated (have degree 0)?
- What is the maximum number of edges in a digraph with V vertices and no parallel edges?
- What is the minimum number of edges in a digraph with V vertices, none of which are isolated?

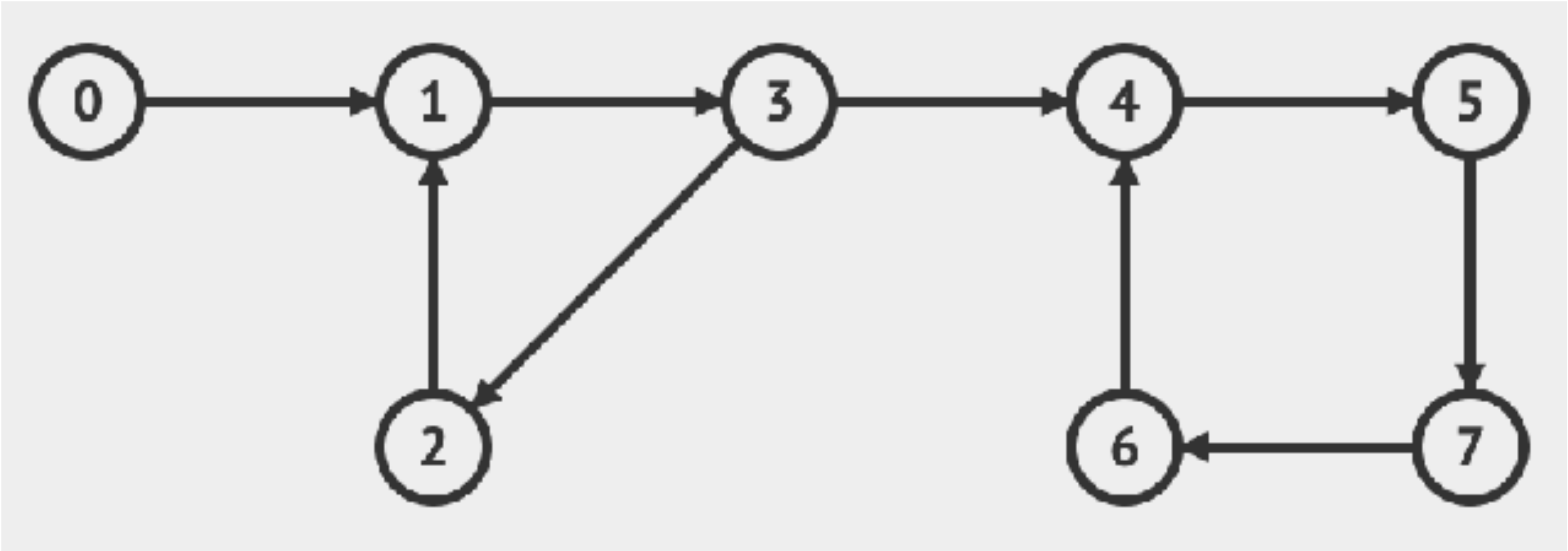
# Problem 2

- Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:
- 8-4
- 2-3
- 1-11
- 0-6
- 3-6
- 10-3
- 7-11
- 7-8
- …

- 11-8
- 2-0
- 6-2
- 5-2
- 5-10
- 5-0
- 8-1
- 4-1

# Problem 3

- Run DFS and BFS on the following digraph starting at vertex 0.



# Answer 1

- What is the maximum number of edges in an undirected graph with V vertices and no parallel edges?
  - $n(n-1)/2$, where $n = |V|$.
- What is the minimum number of edges in an undirected graph with V vertices, none of which are isolated (have degree 0)?
  - $n-1$.
- What is the maximum number of edges in a digraph with V vertices and no parallel edges?
  - $n(n-1)$, where $n = |V|$.
- What is the minimum number of edges in a digraph with V vertices, none of which are isolated?
  - $n-1$.

# Answer 2

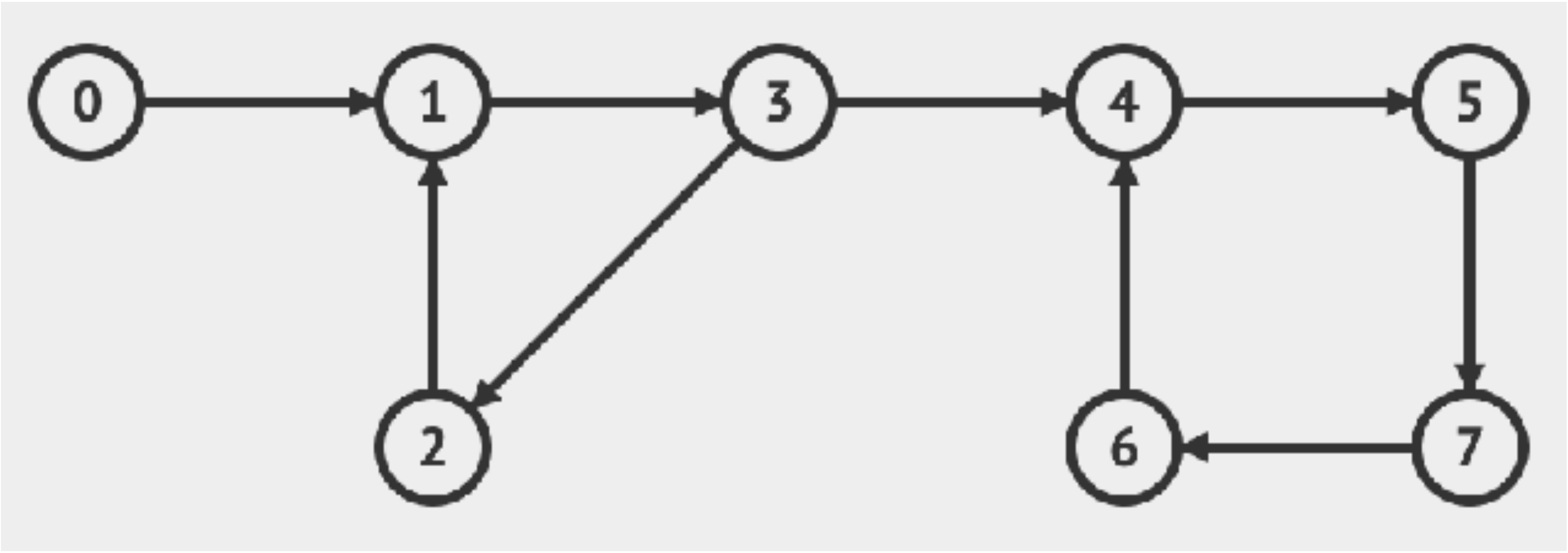
- Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:
- 8-4
- 2-3
- 1-11
- 0-6
- 3-6
- 10-3
- 7-11
- 7-8
- …

- 11-8
- 2-0
- 6-2
- 5-2
- 5-10
- 5-0
- 8-1
- 4-1

- 0 -> 5 -> 2 -> 6
- 1 -> 4 -> 8 -> 11
- 2 -> 5 -> 6 -> 0 -> 3
- 3 -> 10 -> 6 -> 2
- 4 -> 1 -> 8
- 5 -> 0 -> 10 -> 2
- 6 -> 2 -> 3 -> 0
- 7 -> 8 -> 11
- 8 -> 1 -> 11 -> 7 -> 4
- 9 ->
- 10 -> 5 -> 3
- 11 -> 8 -> 7 -> 1

# Answer 3

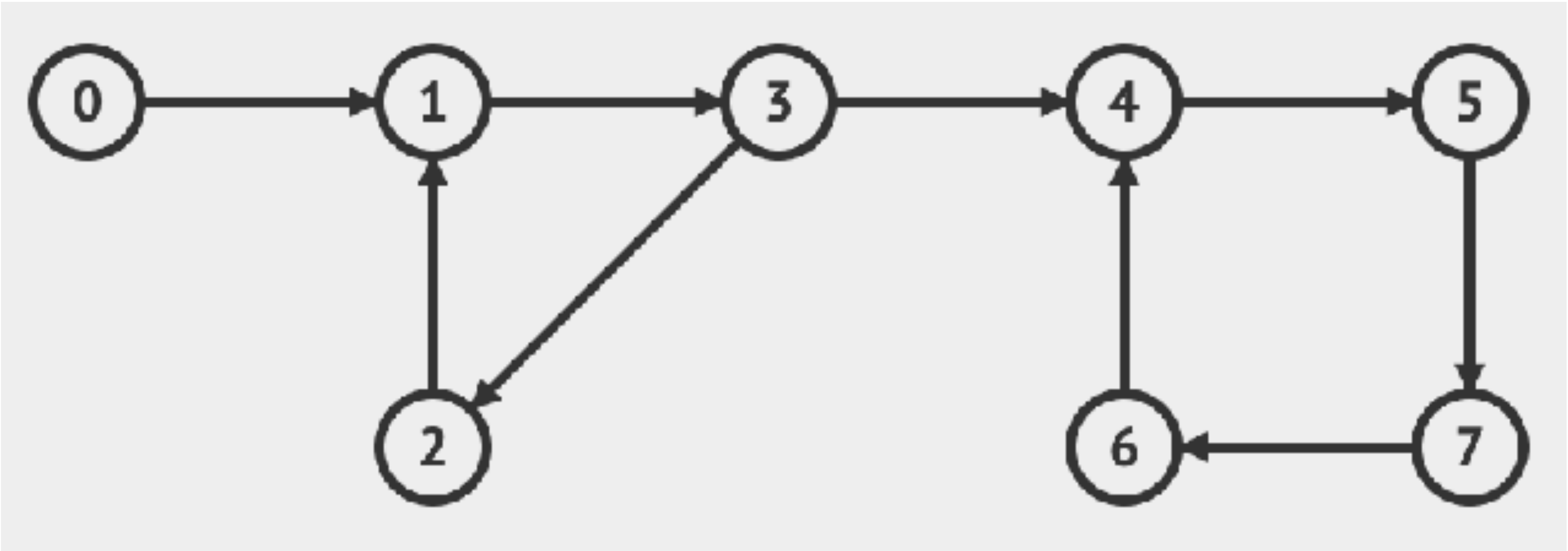
- DFS - Order of visit: 0, 1, 3, 2, 4, 5, 7, 6




| V | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 3 |
| 3 | T | 1 |
| 4 | T | 3 |
| 5 | T | 4 |
| 6 | T | 7 |
| 7 | T | 5 |

# Answer 3

- BFS - Order of visit: 0, 1, 3, 2 4, 5, 7, 6




| V | marked | edgeTo | distTo |
|---|---|---|---|
| 0 | T | - | 0 |
| 1 | T | 1 | 1 |
| 2 | T | 3 | 2 |
| 3 | T | 1 | 2 |
| 4 | T | 3 | 3 |
| 5 | T | 4 | 4 |
| 6 | T | 7 | 6 |
| 7 | T | 5 | 5 |