# CSC 226

Algorithms and Data Structures: II

Graphs

Tianming Wei

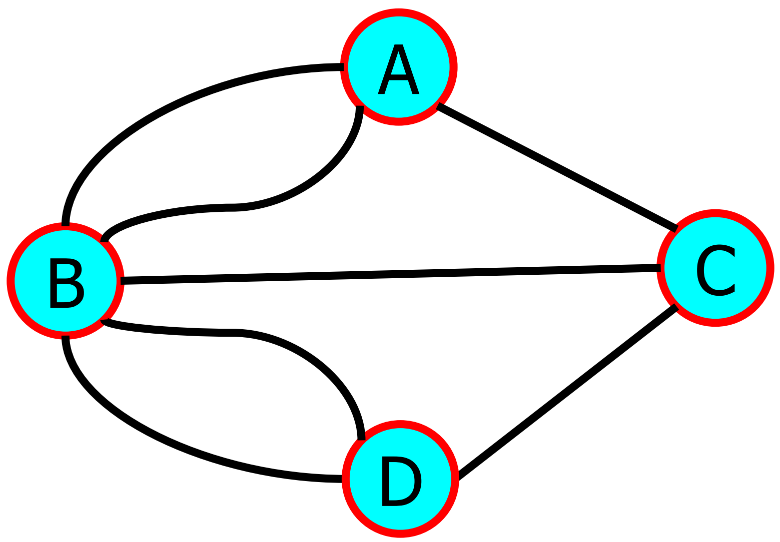
twei@uvic.ca

ECS 466

# Abstract Meaning of the Term Graph

- A ***graph*** $G = (V, E)$ is a set $V$ of ***vertices*** (*nodes*) and a collection $E$ of pairs from $V$, called ***edges*** (*arcs*).

- **Graph Example:**



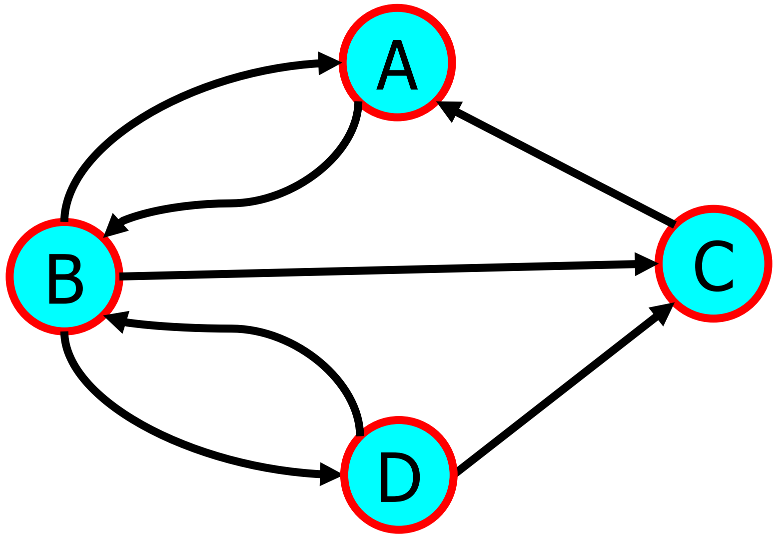

$$V = \{A, B, C, D\}$$

$$E = \left\{ \begin{matrix} \{A, B\}, \{A, B\}, \{A, C\}, \\ \{B, C\}, \{B, D\}, \{B, D\}, \{C, D\} \end{matrix} \right\}$$

# Abstract Meaning of the Term Graph

- A ***digraph*** $G = (V, E)$ is a set $V$ of ***vertices*** (*nodes*) and a collection $E$ of ordered pairs from $V$, called ***edges*** (*arcs*).

- **Digraph Example:**

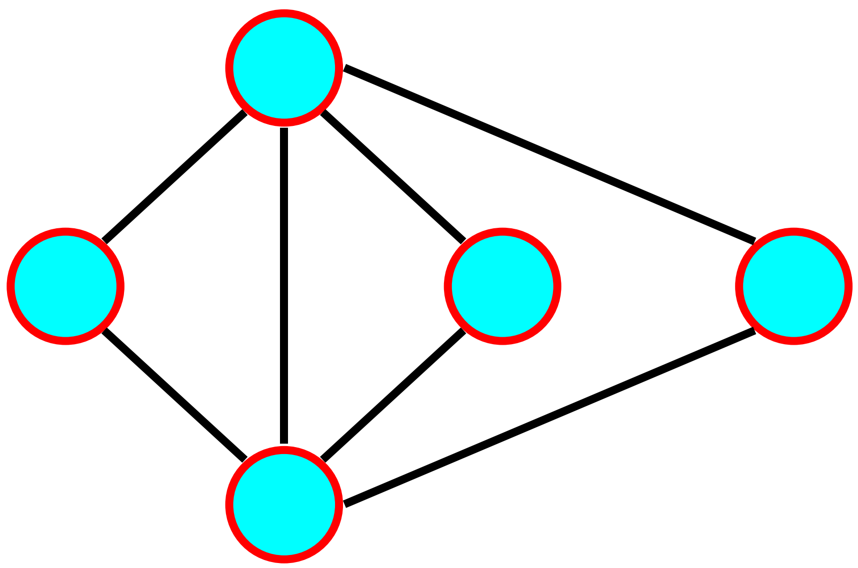


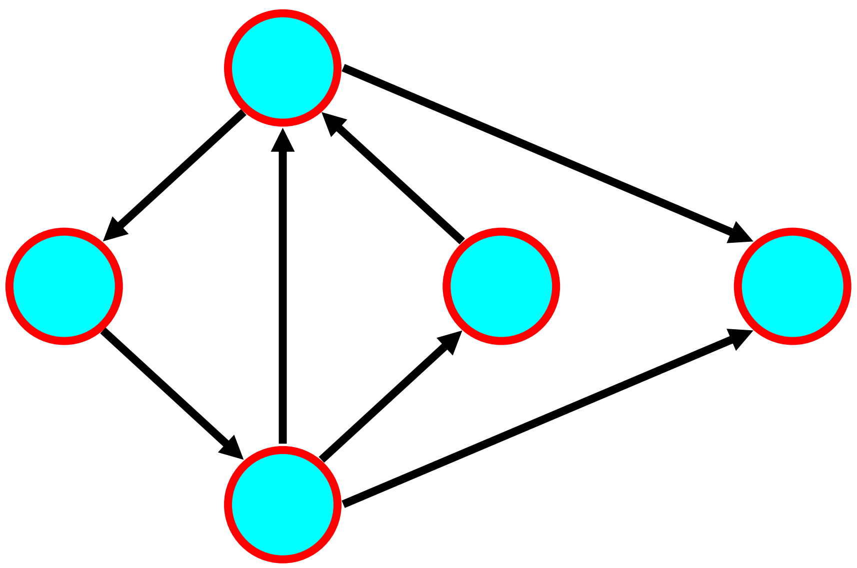

$$V = \{A, B, C, D\}$$

$$E = \left\{ \begin{matrix} (A, B), (B, A), (B, D), \\ (D, B), (B, C), (D, C), (C, A) \end{matrix} \right\}$$

# Graph Terminology

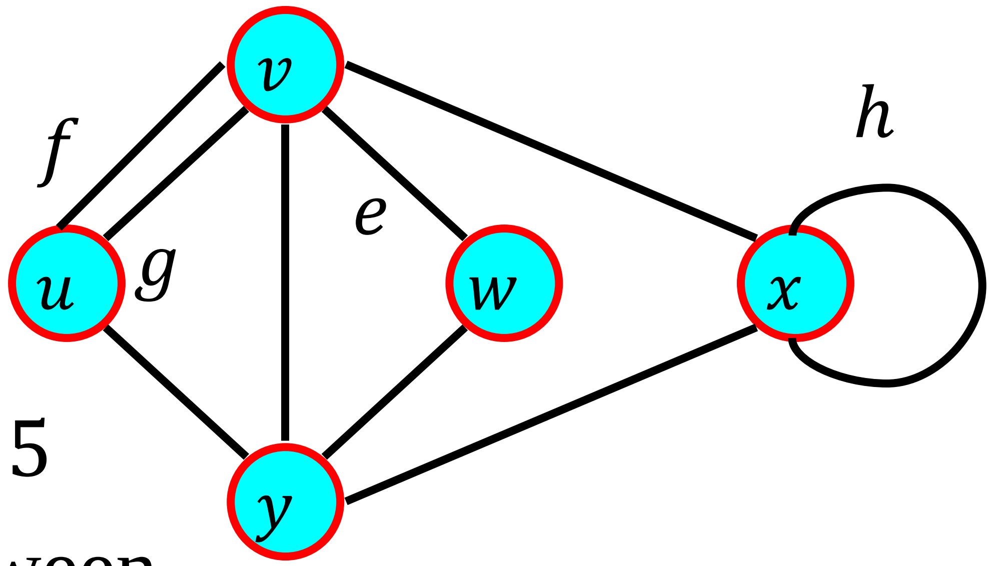
- Much of the terminology for graphs is applicable to undirected graphs and directed graphs





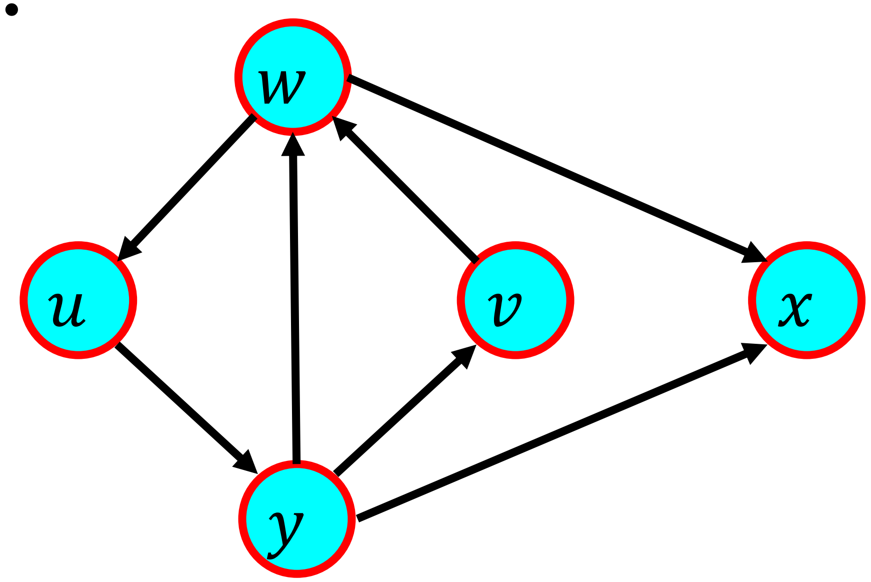
# Undirected Edges

- An *undirected edge e* represents a *symmetric* relation between two vertices *v* and *w* represented by the vertices.
  - We usually write $e = \{v, w\}$, where $\{v, w\}$ is an unordered pair.
  - *v*, *w* are the *endpoints* of the edge
  - *v* is *adjacent* to *w*
  - *e* is *incident* upon *v* and *w*
  - The *degree* of a vertex is the number of incident edges, eg. $\deg(v) = 5$
  - *parallel* edges – more than one edge between a pair of vertices, eg. $f$ and $g$
  - *self-loop* – edge that connects a vertex to itself, eg. *h*
  - Typically, the number of vertices is denoted by *n* and the number of edges by *m*.



# Directed Edges or Arcs

- A *directed edge* (*or arc*) *e* represents an *asymmetric* relation between two vertices *v* and *w*.
  *e* = (*v*, w) denotes an ordered pair.
  - *v*, *w* are the endpoints of the edge
  - *v* is *adjacent* to *w*
  - *e* is *incident* upon *v* and *w*
  - The arc goes from the *source* vertex *v* to the *destination* vertex *w*
- The *indegree* of a vertex is the number of incoming arcs
- The *outdegree* of a vertex is the number of outgoing arcs



# Walks

- A ***walk*** in a graph is a sequence of vertices $v_1, v_2, \dots, v_n$ such that there exist edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, …, $\{v_{n-1}, v_n\}$
- If $v_1 = v_n$ it's ***closed***, otherwise it's ***open***.
- The ***length*** of a walk is the number of edges.
- If no edge is repeated it's a ***trail***. If closed a ***circuit***.
- If no vertex is repeated it's a ***path***. If closed a ***cycle***.

# Graphs

- A graph is ***connected*** if every pair of vertices is connected by a path.
- A ***simple graph*** is a graph with no self-loops and no parallel or multi-edges
- A ***complete graph*** is a simple graph where an edge connects every pair of vertices

# Connected Digraphs

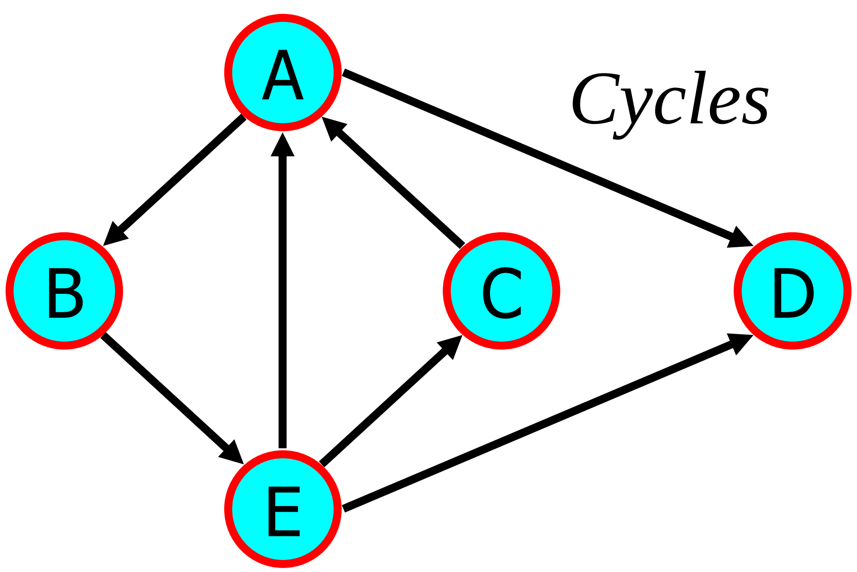
- Given vertices $u$ and $v$ of a digraph $G$, we say $v$ is ***reachable*** from $u$ if $G$ has a directed path from $u$ to $v$.

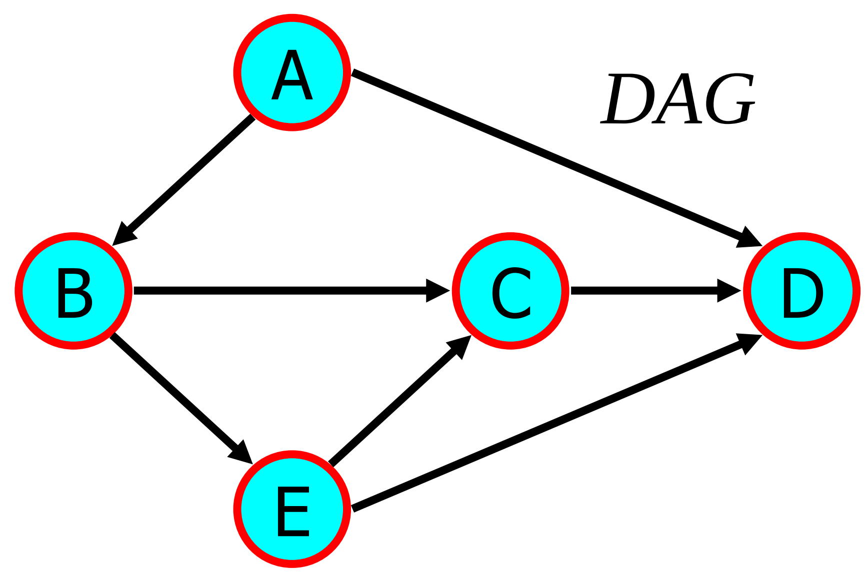
- A digraph $G$ is ***connected*** if every pair of vertices is connected by an undirected path.

- A digraph $G$ is ***strongly connected*** if for every pair of vertices $u$ and $v$ of $G$, $u$ is reachable from $v$ and $v$ is reachable from $u$.

# Directed Acyclic Graphs (DAGs)

- A ***directed acyclic graph*** (DAG) is a directed graph with no cycles.



# Subgraphs

- A ***subgraph*** of $G = (V, E)$ *is a graph* $G' = (V', E')$ where
  - $V'$ is a subset of $V$
  - $E'$ consists of edges $\{v, w\}$ in $E$ such that both $v$ and $w$ are in $V'$
- A ***spanning subgraph*** of $G$ contains all the vertices of $G$

# Theorem

- **Theorem:** If $G = (V, E)$ is an undirected graph, then

$$\sum_{v \in V} \deg(v) = 2|E|.$$

- *Proof:*
  - Every edge contributes 2 to the total degree.
- **Corollary:** For any undirected graph, the number of vertices of odd degree must be even.

# Euler Circuits

- Let $G = (V, E)$ be an undirected graph with no isolated vertices. Then $G$ is said to have an ***Euler circuit*** if there is a circuit in $G$ that traverses every edge exactly once.
  - If there is a trail from vertex $a$ to $b$ which traverses every edge exactly once, it is an ***Euler trail***.

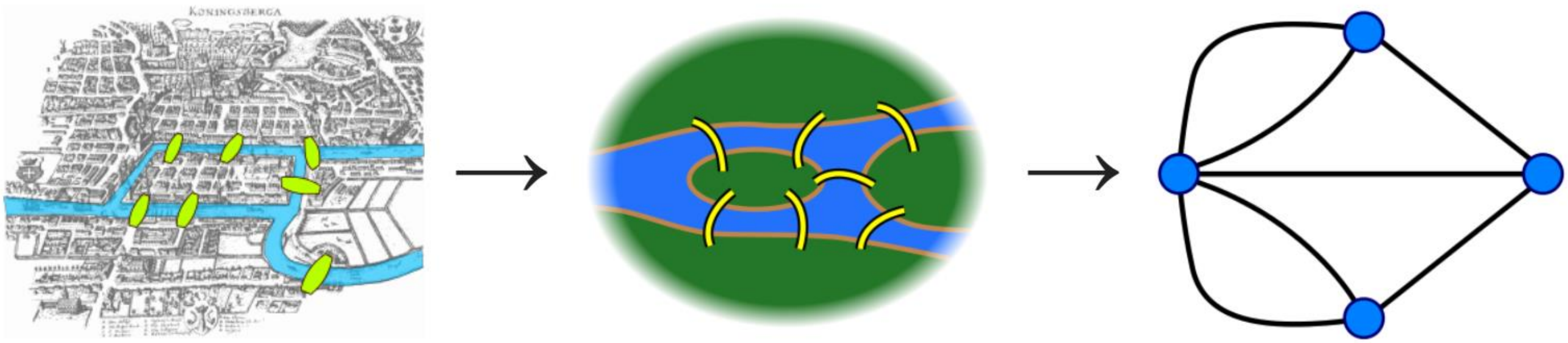
# Theorem

- **Theorem:** Let $G = (V, E)$ be an undirected graph with no isolated vertices. Then, $G$ has an Euler circuit if and only if $G$ is connected and every vertex has an even degree.
  - This is the 7 bridges of Konigsberg.

- ***Corollary:*** There exists an Euler trail in $G$ if and only if $G$ is connected and has exactly two vertices of odd degree.

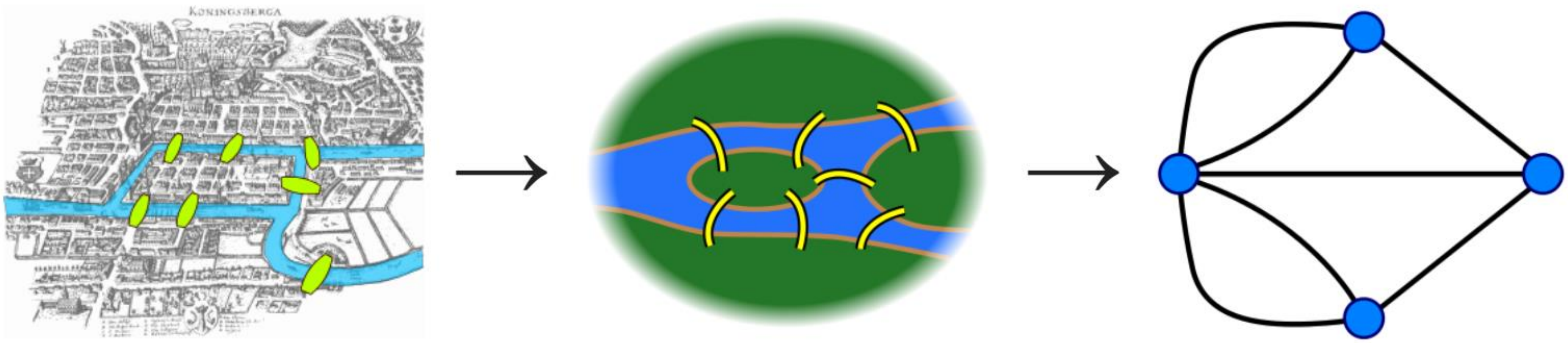
# 7 bridges of Konigsberg

- Is it possible to devise a walk through the city that would cross each of those bridges once and only once?



# 7 bridges of Konigsberg

- Is it possible to devise a walk through the city that would visit each of towns once and only once?
  - **Hamiltonian cycle**



# Depth-first search

Goal. Systematically traverse a graph.

Idea. Mimic maze exploration. ← function-call stack acts as ball of string

**DFS (to visit a vertex v)**

**Mark v as visited.**

**Recursively visit all unmarked vertices w adjacent to v.**

Typical applications.

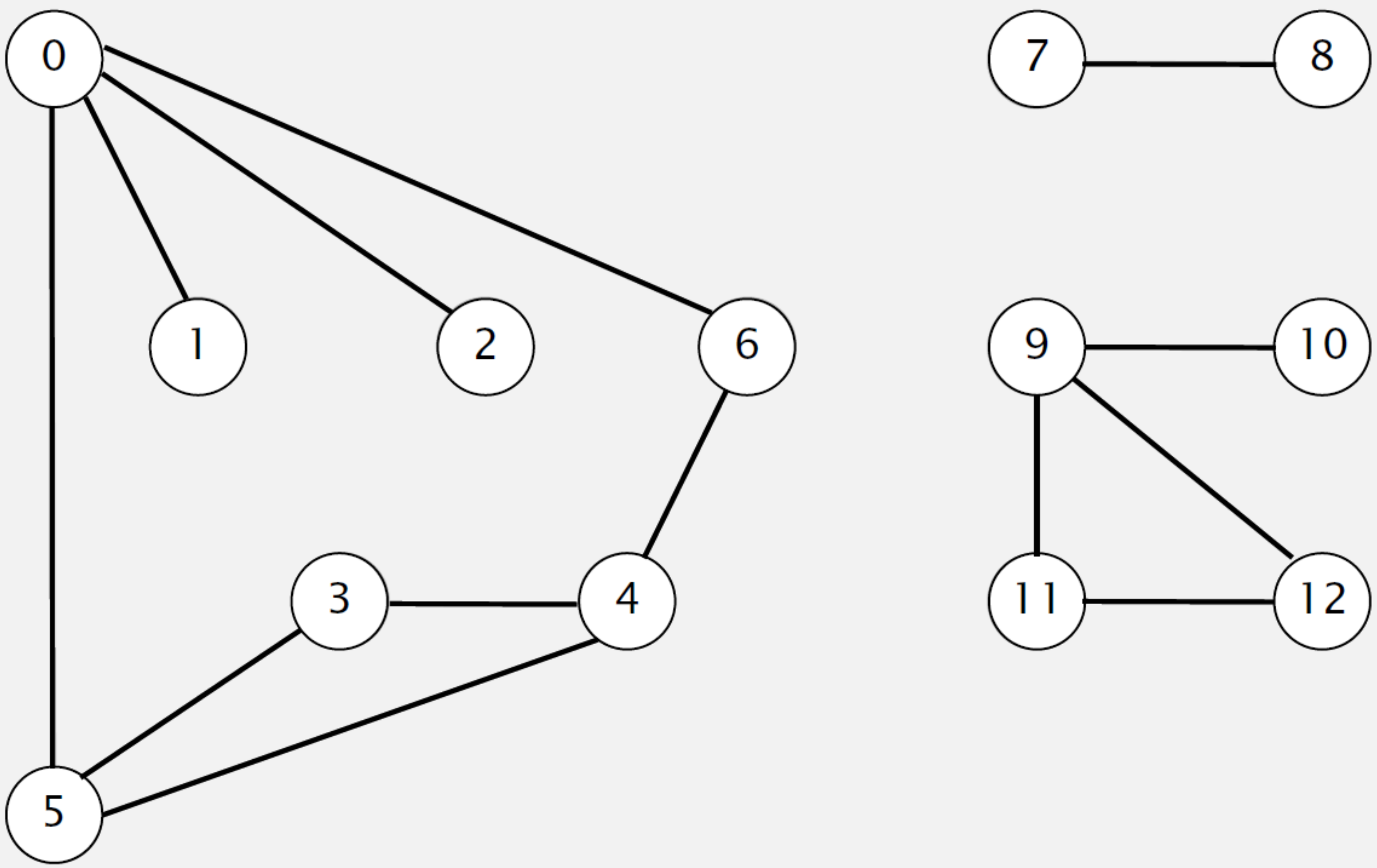
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

# Depth-first search demo

To visit a vertex $v$ :

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.



graph G

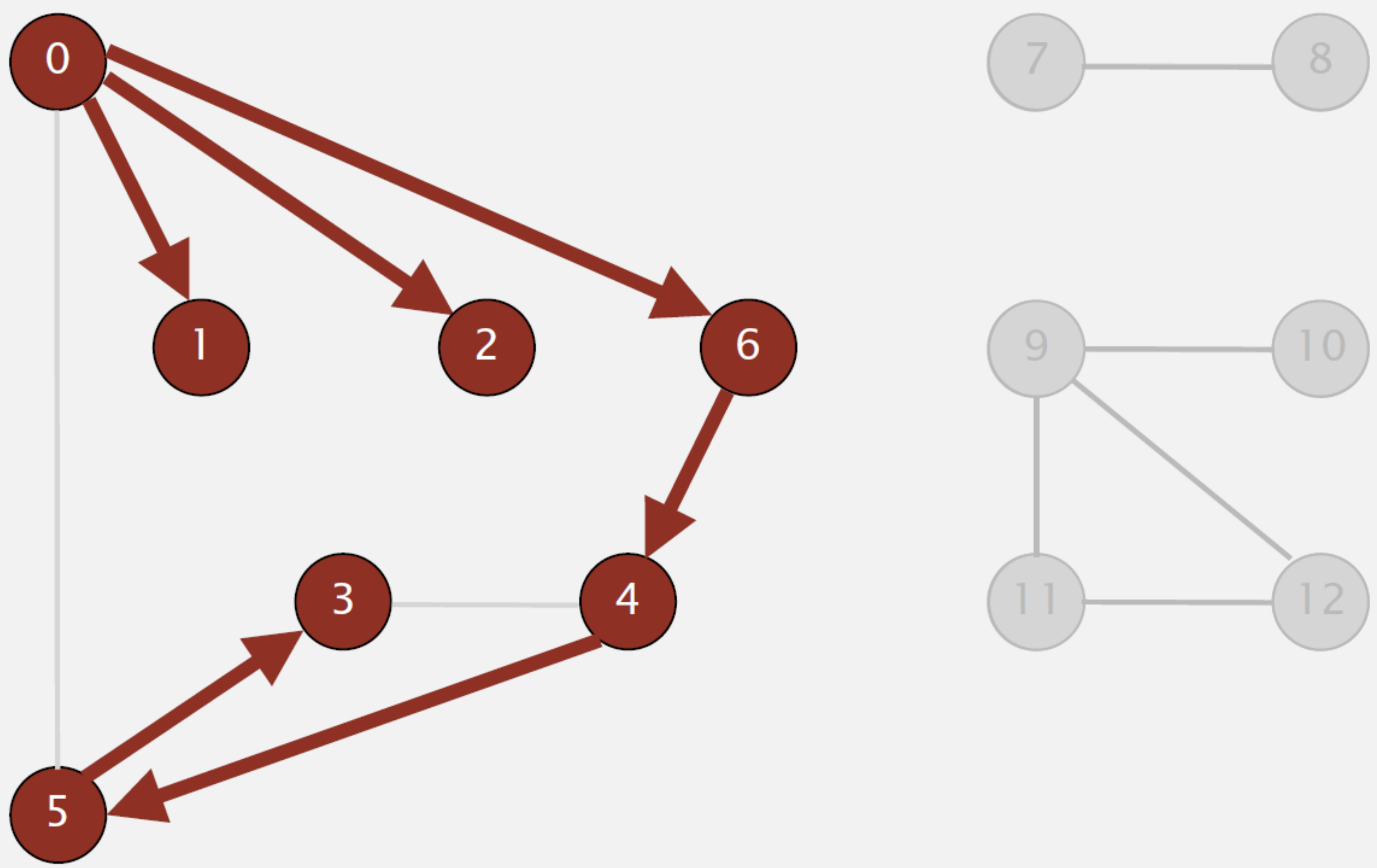
**tinyG.txt**

$V$ → 13
13 ← $E$
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

# Depth-first search demo

To visit a vertex $v$ :

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.




| v | marked[] | edgeTo[] |
|---|---|---|
| 0 | T | – |
| 1 | T | 0 |
| 2 | T | 0 |
| 3 | T | 5 |
| 4 | T | 6 |
| 5 | T | 4 |
| 6 | T | 0 |
| 7 | F | – |
| 8 | F | – |
| 9 | F | – |
| 10 | F | – |
| 11 | F | – |
| 12 | F | – |

vertices reachable from 0

# Breadth-first search

Repeat until queue is empty:

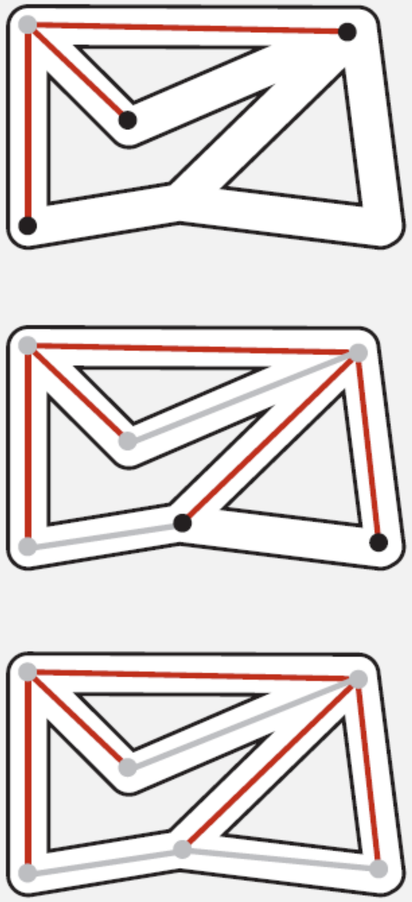
- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

**BFS (from source vertex s)**

**Put s onto a FIFO queue, and mark s as visited.**

**Repeat until the queue is empty:**

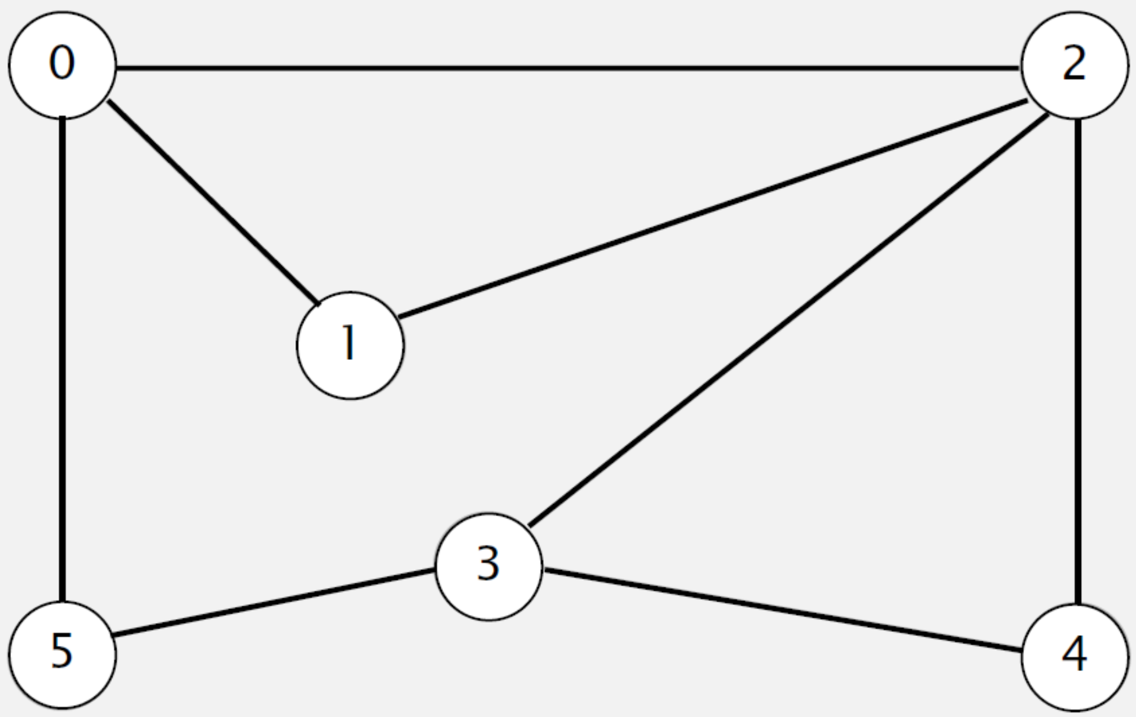
- **remove the least recently added vertex v**
- **add each of v's unvisited neighbors to the queue, and mark them as visited.**



# Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.




graph G

**tinyCG.txt**

$V$ → 6

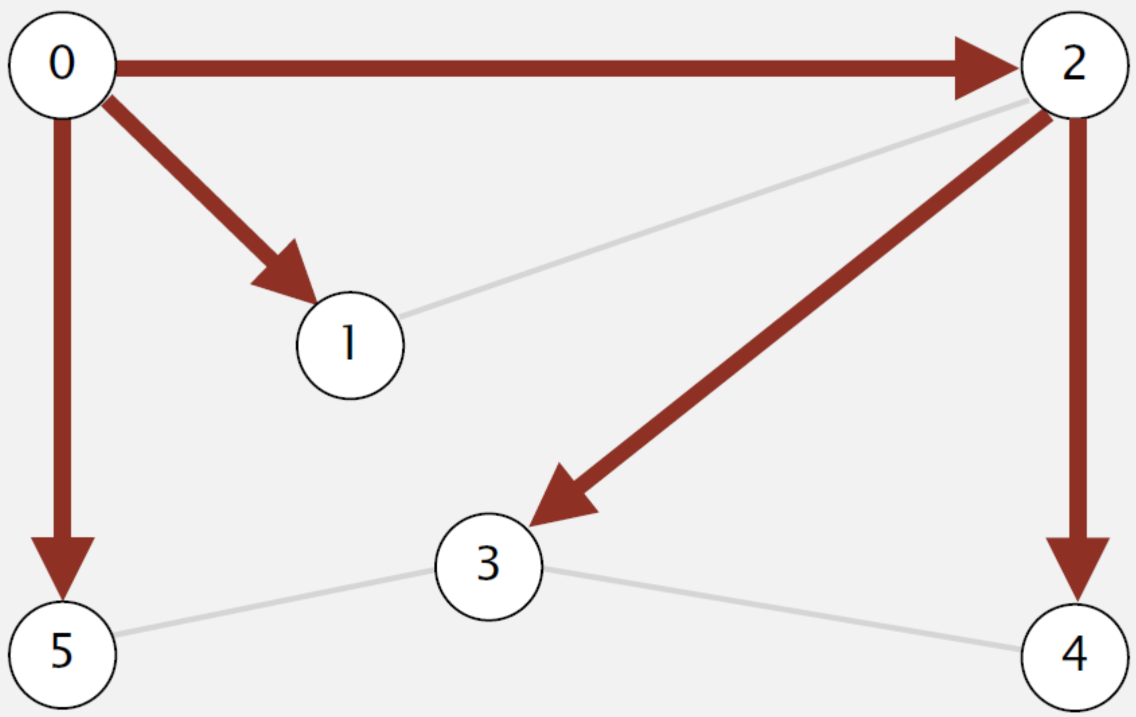
8 ← $E$

```
6
8
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2
```

# Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.




| v | edgeTo[] | distTo[] |
|---|---|---|
| 0 | – | 0 |
| 1 | 0 | 1 |
| 2 | 0 | 1 |
| 3 | 2 | 2 |
| 4 | 2 | 2 |
| 5 | 0 | 1 |

**done**

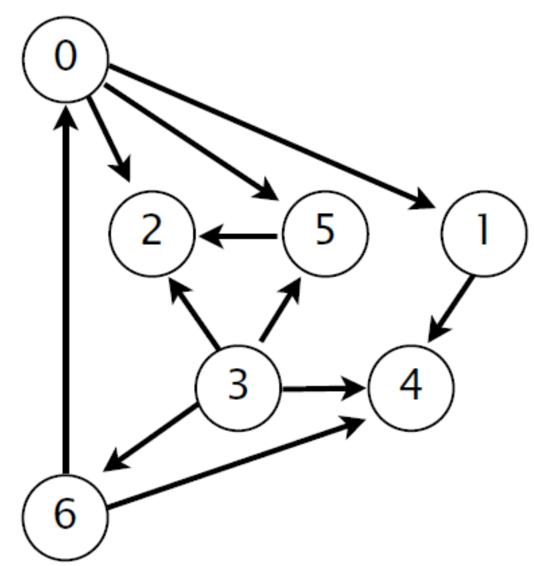
# Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

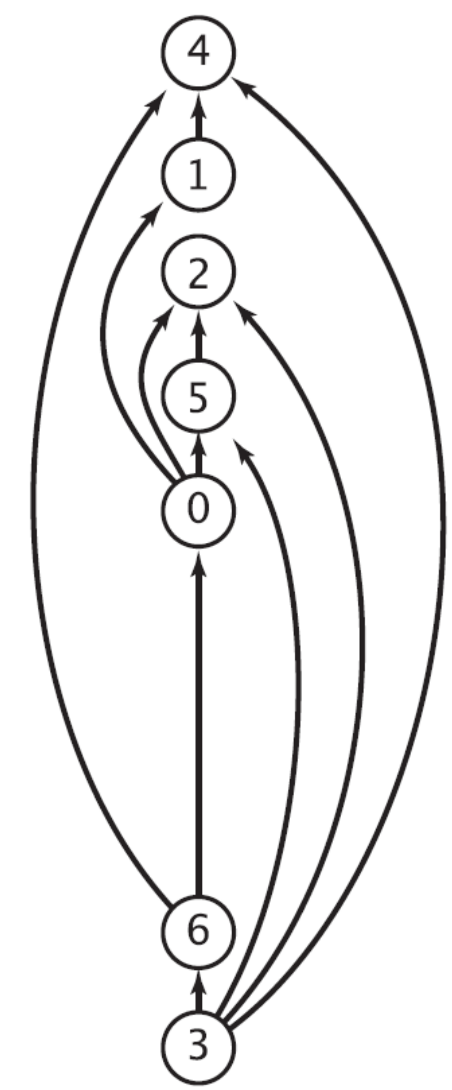
Digraph model. vertex = task; edge = precedence constraint.

0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Advanced Programming

**tasks**




**precedence constraint graph**




**feasible schedule**

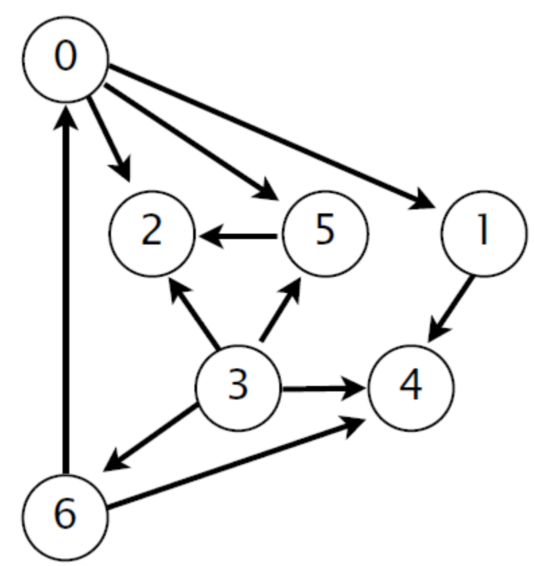
# Topological sort

DAG. Directed acyclic graph.

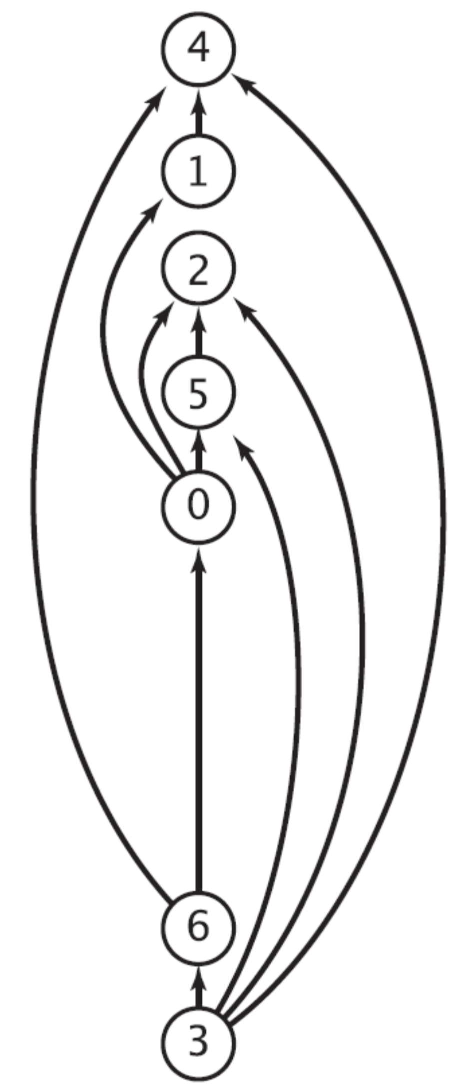
Topological sort. Redraw DAG so all edges point upwards.

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

**directed edges**




**DAG**

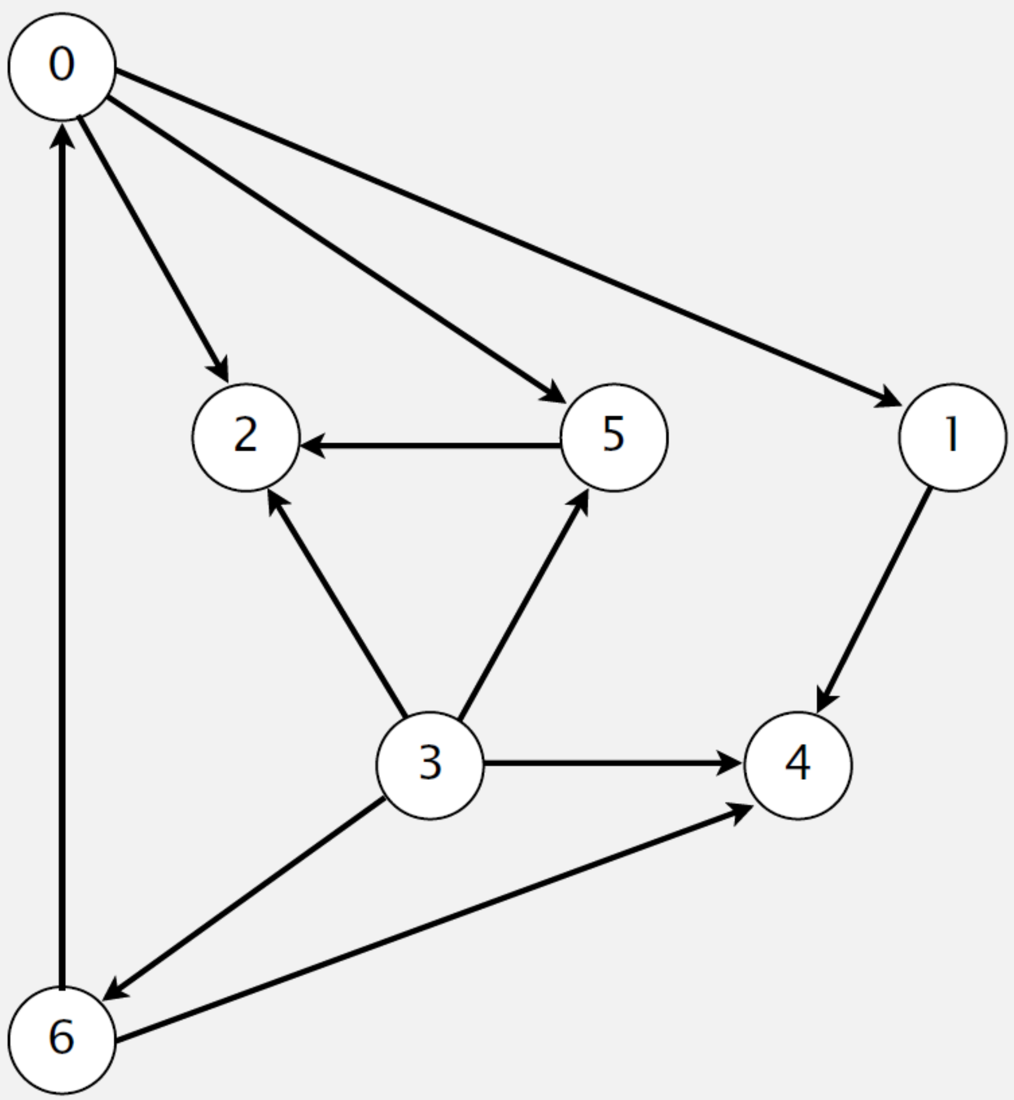



**topological order**

Solution. DFS. What else?

# Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.




**postorder**

4 1 2 5 0 6 3

**topological order**

3 6 0 5 2 1 4

**done**

# Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
   ...
}
```

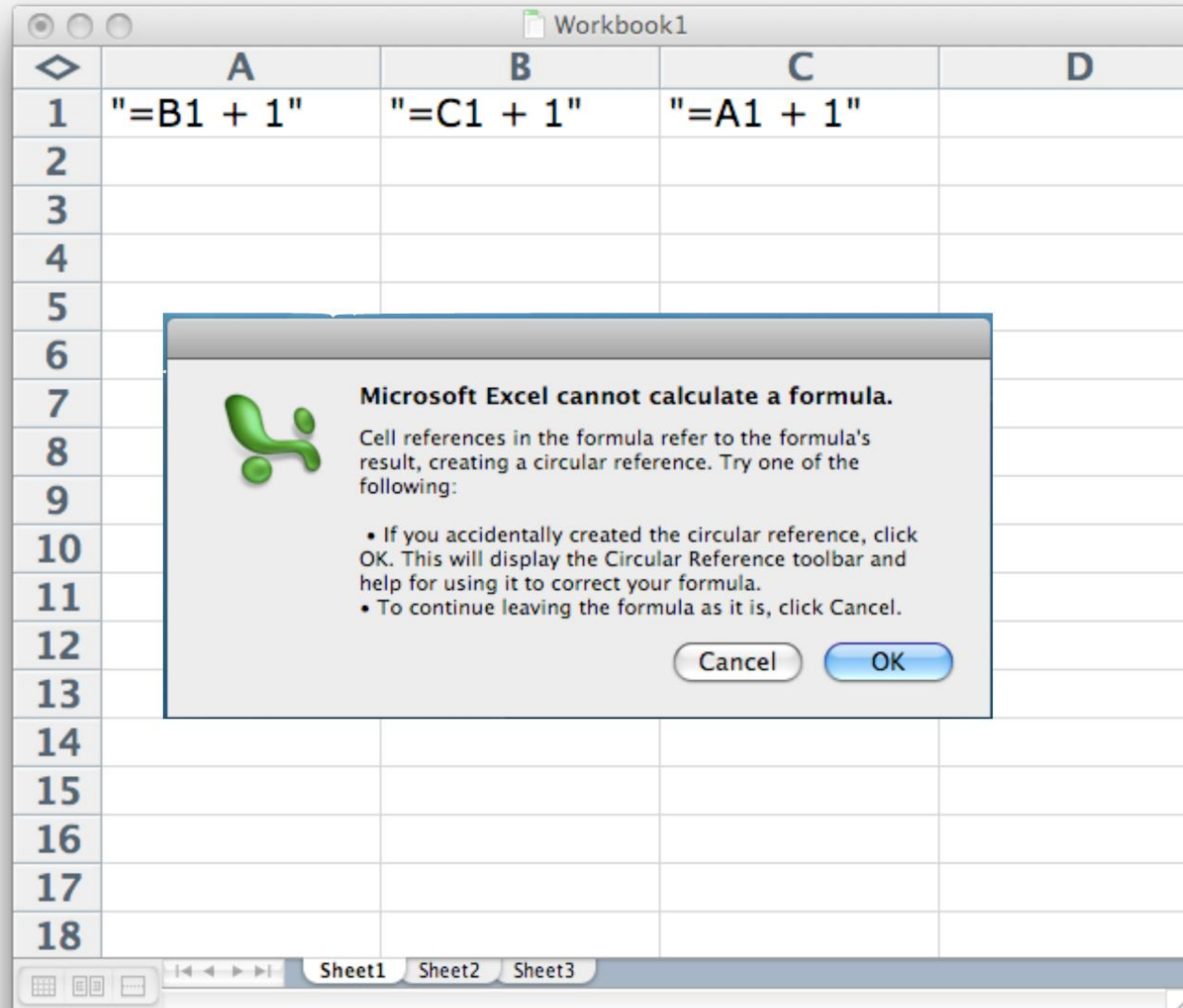
```
public class B extends C
{
   ...
}
```

```
public class C extends A
{
   ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
       ^
1 error
```

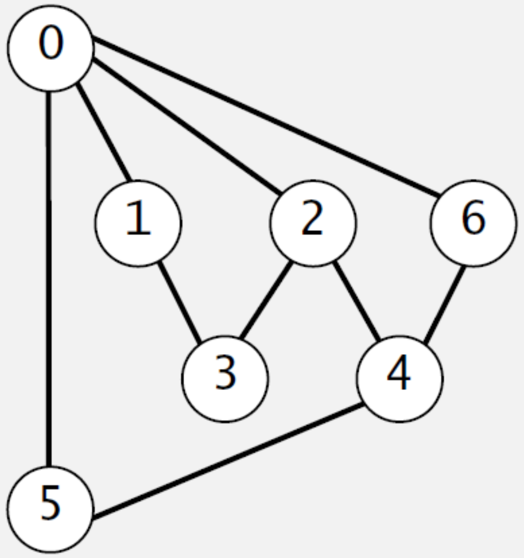
# Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)



## Graph-processing challenge

Problem. Find a cycle.




0–1
0–2
0–5
0–6
1–3
2–3
2–4
4–5
4–6

0–5–4–6–0

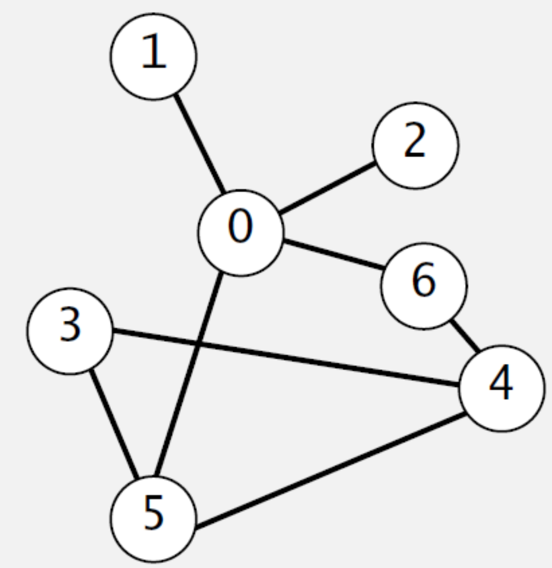
How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution
(see textbook)

# Graph-processing challenge

Problem. Lay out a graph in the plane without crossing edges?

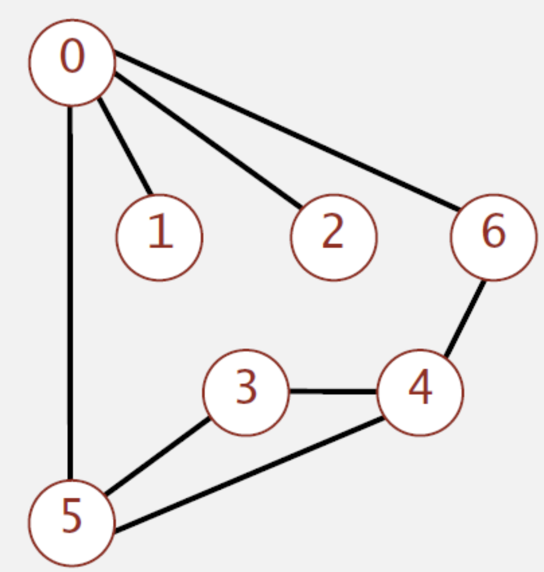



0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

How difficult?

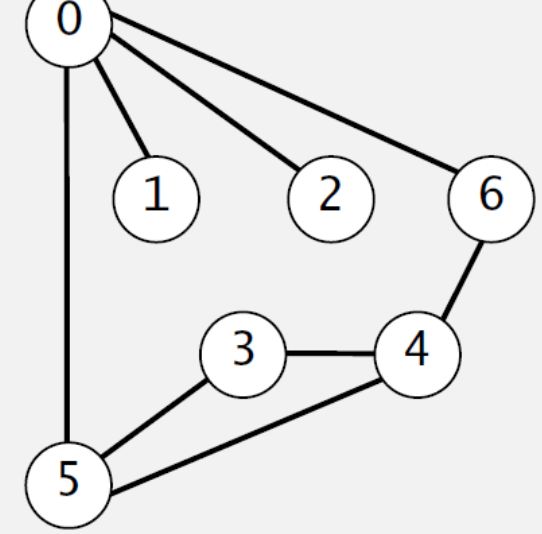
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- ✓ Hire an expert.
- Intractable.
- No one knows.
- Impossible.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for most practitioners)



# Graph-processing challenge

Problem. Are two graphs identical except for vertex names?

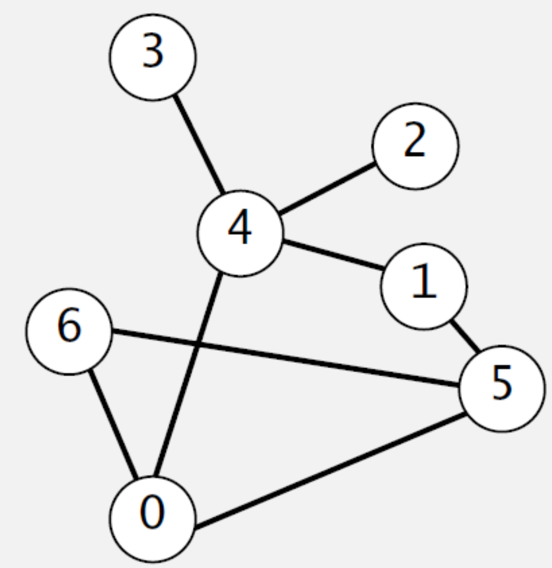



0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- ✓ No one knows.
- Impossible.

graph isomorphism is longstanding open problem




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

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

# Some graph-processing problems

| problem | description |
|---|---|
| **s-t path** | *Is there a path between s and t ?* |
| **shortest s-t path** | *What is the shortest path between s and t ?* |
| **cycle** | *Is there a cycle in the graph ?* |
| **Euler circuit** | *Is there a cycle that uses each edge exactly once ?* |
| **Hamilton cycle** | *Is there a cycle that uses each vertex exactly once ?* |
| **connectivity** | *Is there a way to connect all of the vertices ?* |
| **biconnectivity** | *Is there a vertex whose removal disconnects the graph ?* |
| **planarity** | *Can the graph be drawn in the plane with no crossing edges ?* |
| **graph isomorphism** | *Do two adjacency lists represent the same graph ?* |

Challenge. Which graph problems are easy? difficult? intractable?

# Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

| problem | BFS | DFS | time |
|---|---|---|---|
| **path between s and t** | ✔ | ✔ | $E + V$ |
| **shortest path between s and t** | ✔ | | $E + V$ |
| **connected components** | ✔ | ✔ | $E + V$ |
| **biconnected components** | | ✔ | $E + V$ |
| **cycle** | ✔ | ✔ | $E + V$ |
| **Euler circuit** | | ✔ | $E + V$ |
| **Hamilton cycle** | | | $2^{1.657\,V}$ |
| **bipartiteness** | ✔ | ✔ | $E + V$ |
| **planarity** | | ✔ | $E + V$ |
| **graph isomorphism** | | | $2^{c\sqrt{V \log V}}$ |

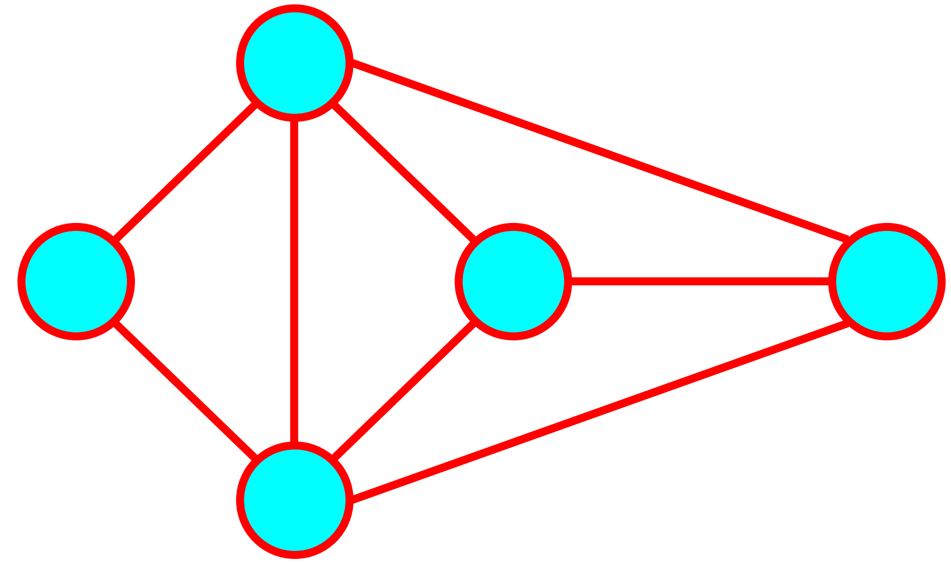
# Trees and Forests

- A ***(free) tree*** is an undirected graph *T* such that
  - *T* is connected
  - *T* has no cycles

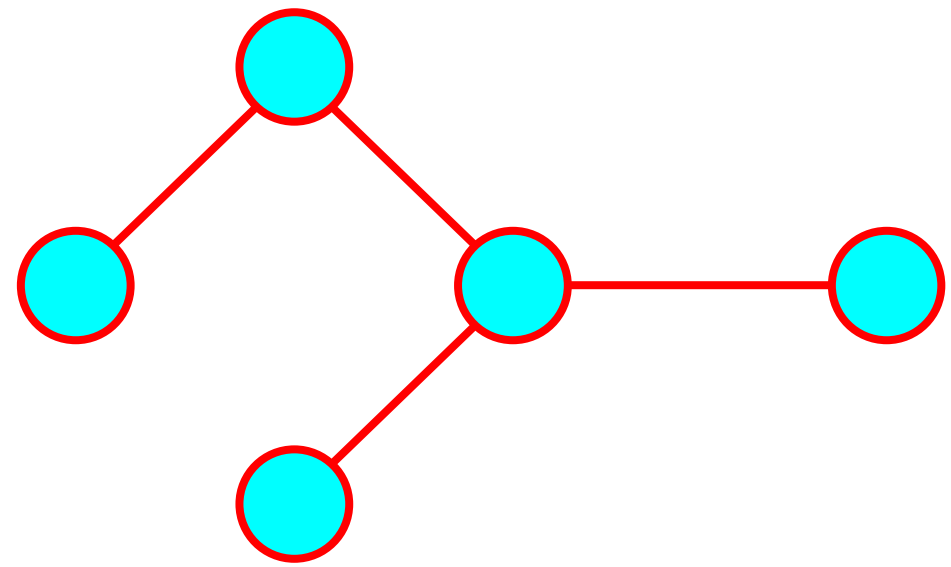
  This definition of tree is different from the one of a rooted tree
- A ***forest*** is an undirected graph without cycles
  - The connected components of a forest are trees

# Spanning Trees and Forests

- A ***spanning tree*** of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A ***spanning forest*** of a graph is a spanning subgraph that is a forest



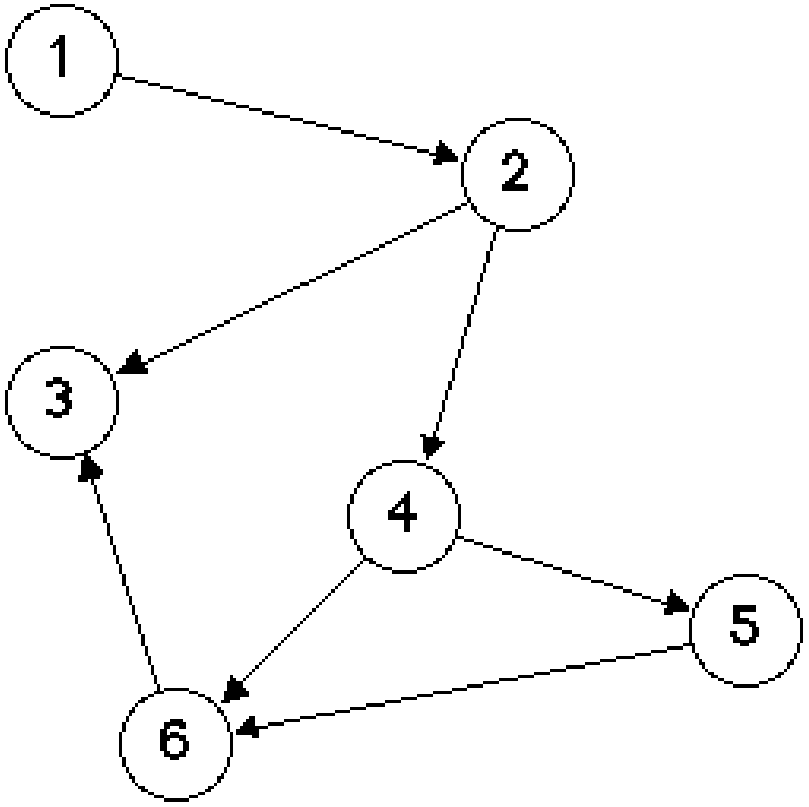
Graph

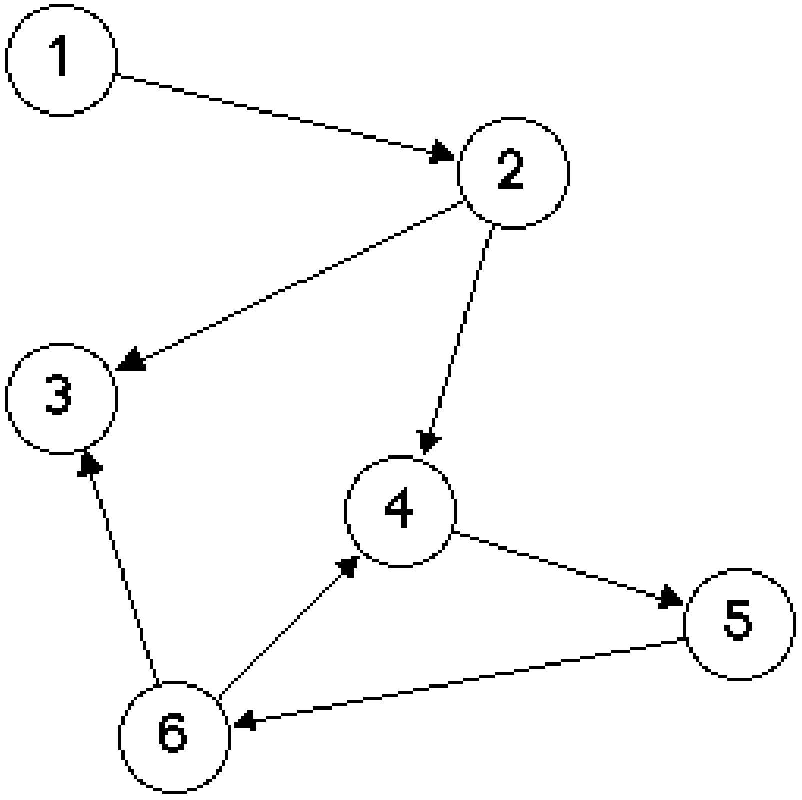


Spanning tree

# Some Practice Questions

- **Which one is a DAG?**

- A. 


B. 

# Some Practice Questions

- **Hamiltonian path in DAGs.**
  - Given a DAG, design a linear-time algorithm to determine whether there is a directed path that visits each vertex exactly once.

# Some Practice Questions

- **Hamiltonian path in DAGs.**
  - ➢ Given a DAG, design a linear-time algorithm to determine whether there is a directed path that visits each vertex exactly once.
  - ➢ *Solution*: Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order.

# Some Practice Questions

- **How to find a “Directed Eulerian Circuit”.**
  - A directed Eulerian circuit is a directed circuit that contains each edge exactly once.
  - *Hint*: Prove that a digraph G has a directed Eulerian circuit if and only if vertex in G has its indegree equal to its outdegree and all vertices with nonzero degree belong to the same strong component.