# Lecture 21

**GRAPHS**

# Outline?

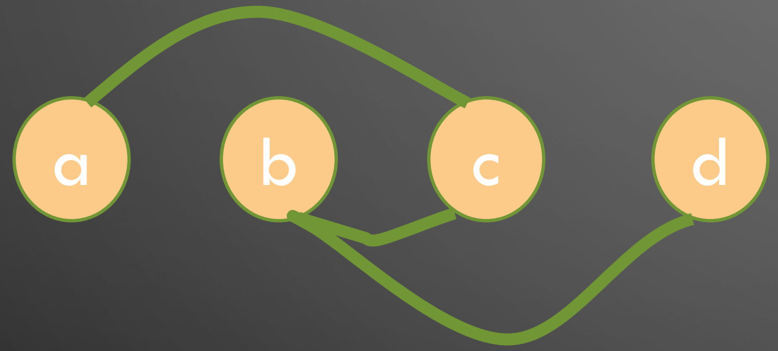
- Define a Graph
- Graph terminology
- Example uses of graphs
- Graph representations

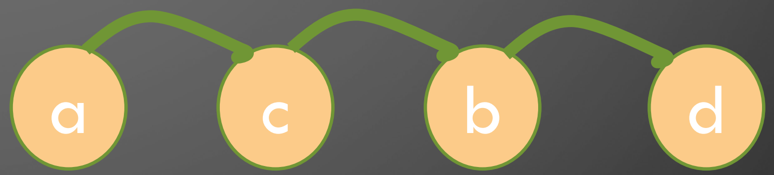
# Definition

- A graph is a collection of vertices (nodes), V, and a set of pairs of vertices, E.

  G = {V, E}

***<u>Example:</u>*** V = {a, b, c, d}, E = {(a, c) (c, b) (b, d)}



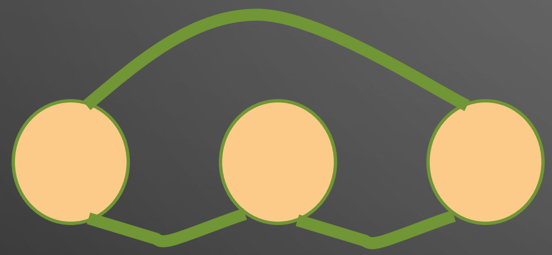

a and c are

***adjacent***

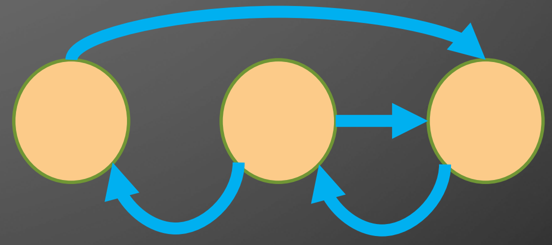
# Edges

- The edge can be directed or undirected
- An undirected edge has its order of vertex pairs neglected.
- A directed edge has its order of vertex pairs to determine direction

***<u>Example:</u>***



**Undirected**

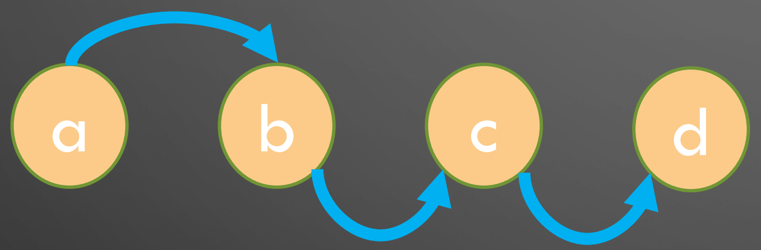


**Directed**

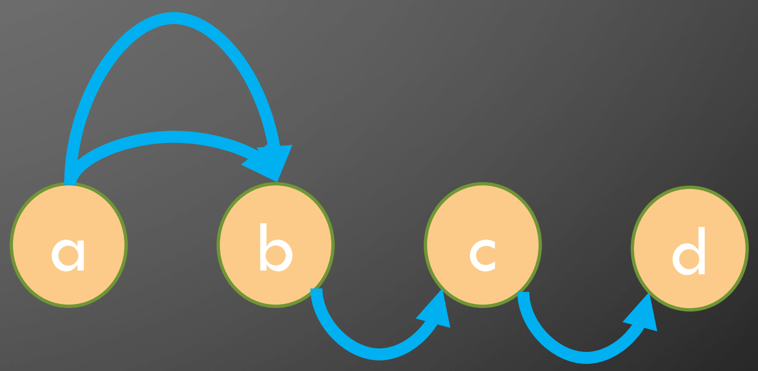
# Graphs versus multi-graphs

- Graphs have at most one edge between two vertices
- Multi-graphs allow duplicate edges

***Example:***



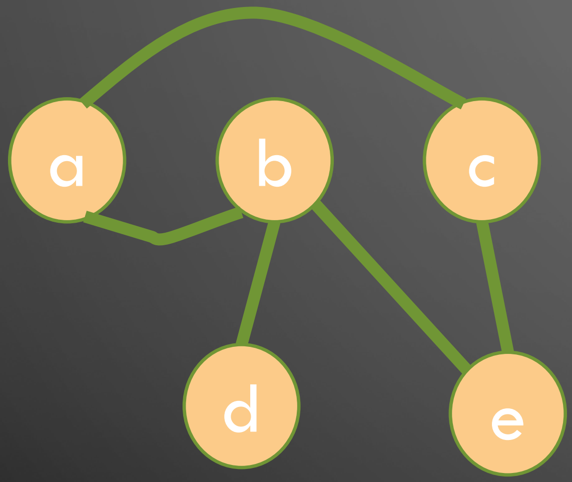

V = {a, b, c, d}, E = {(a, c) (c, b) (b, d)}



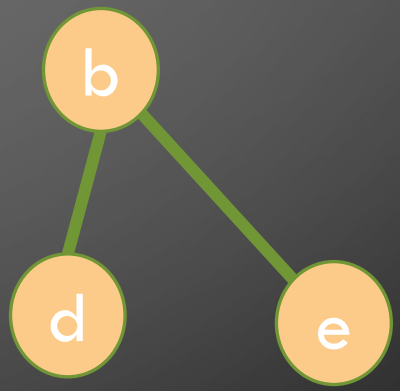
# Subgraph

- A subgraph is a graph formed from a subset of the vertices of the graph

***<u>Example:</u>***




**Graph G**

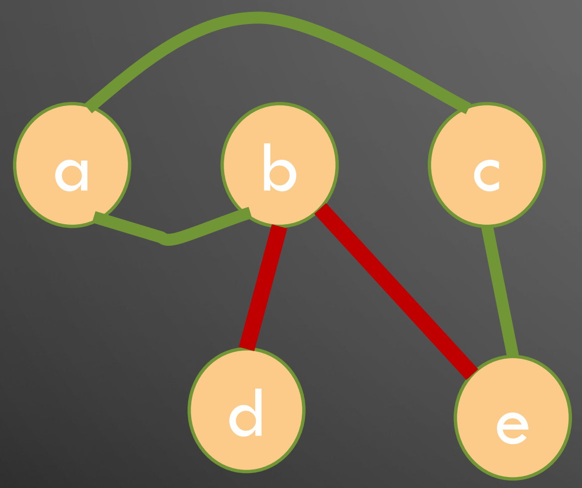



**Subgraph of G**

# Paths

- A path is a sequence of vertices connected by edges

***__Example:__***

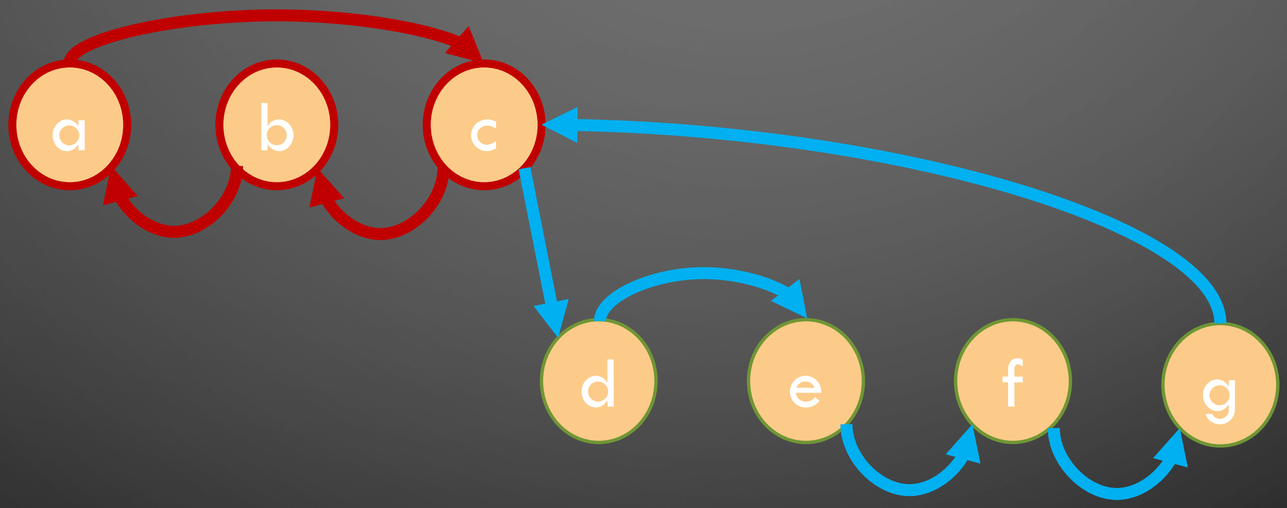



- A path can be directed or undirected

# Cycle

- A cycle is a path that starts and stops at the same vertex

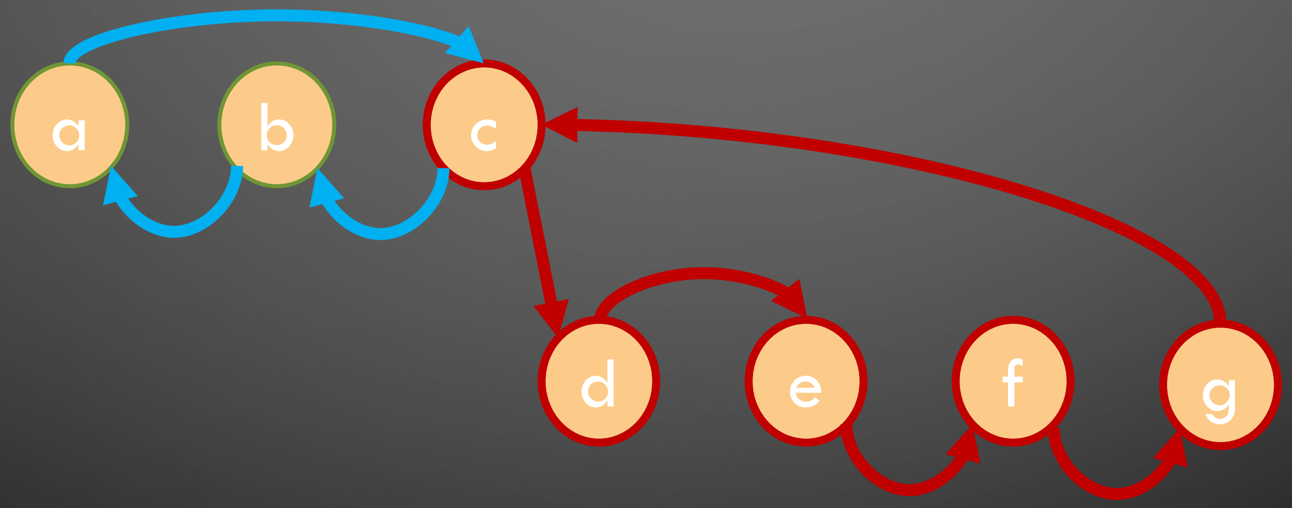
***<u>Example:</u>***



# Cycle

- A cycle is a path that starts and stops at the same vertex

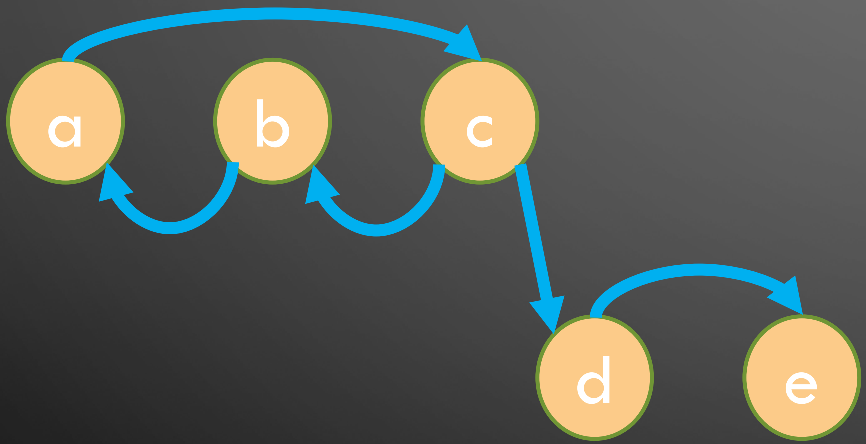
***<u>Example:</u>***



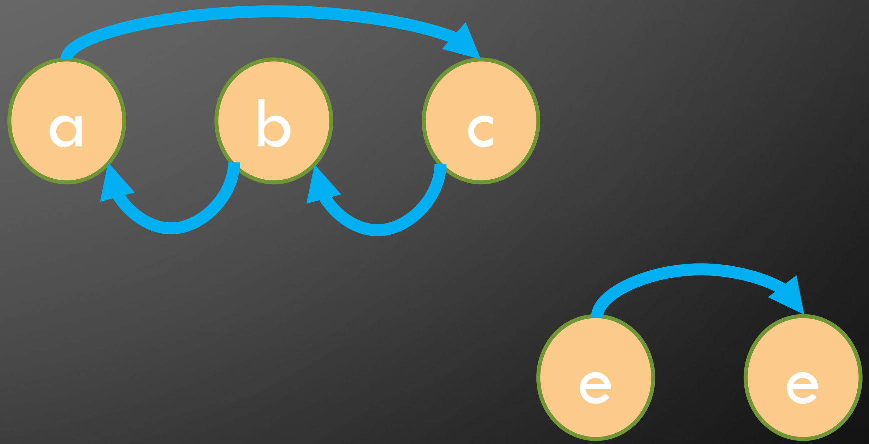
# Connected/Disconnected

- A graph is connected if every pair of vertices are connected by at least one path
- Otherwise it is disconnected.

***Example:***




**Connected**

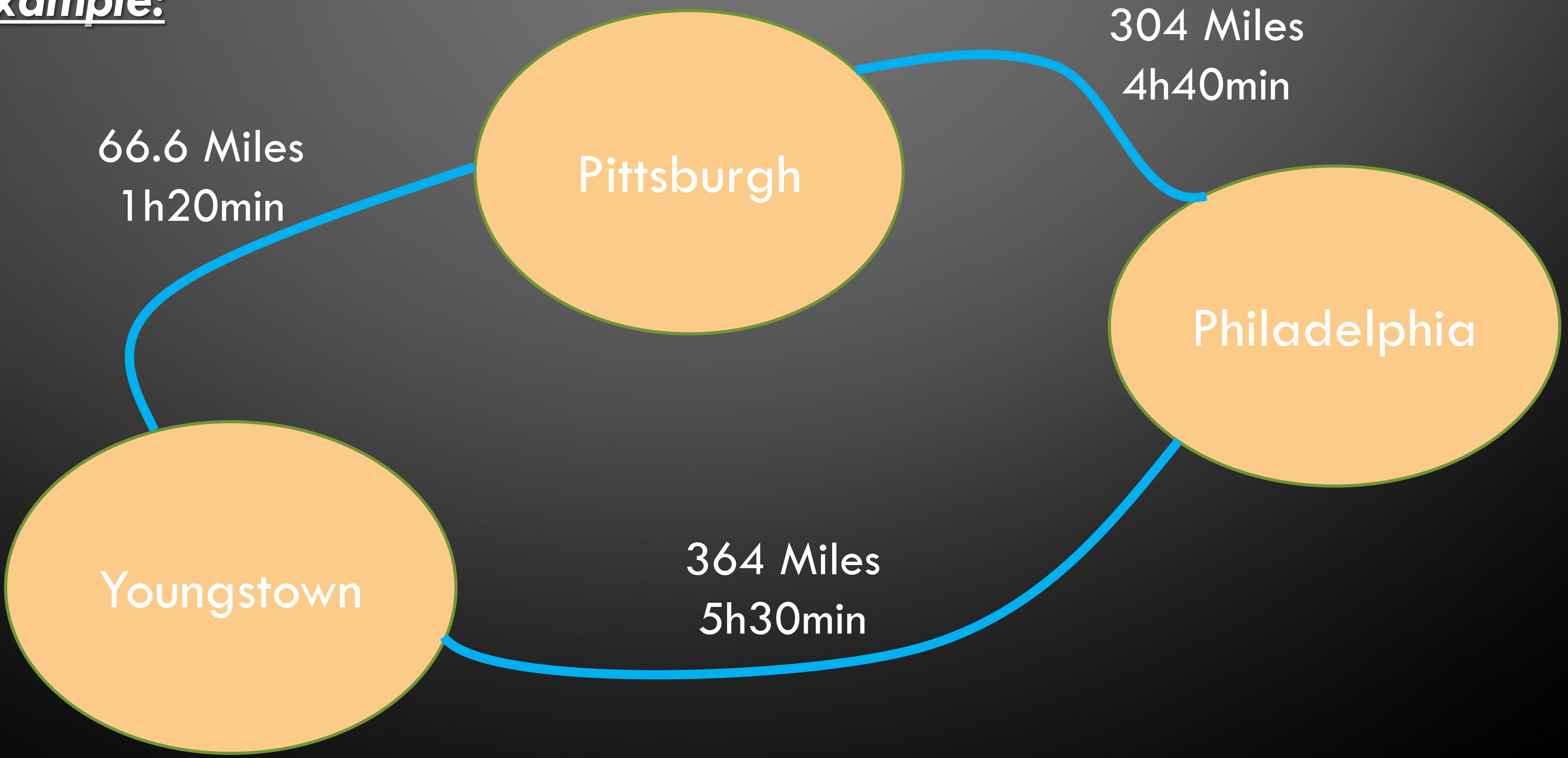



**Disconnected**

# Additional Information

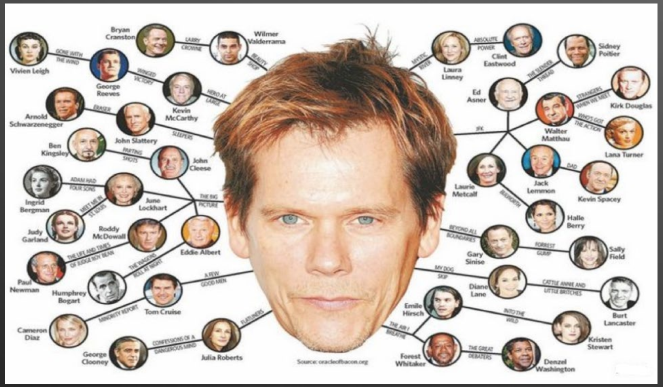
- Vertices and Edges can have properties or attributes attached to them.
- **Weighted graphs** have edges whose property is a cost

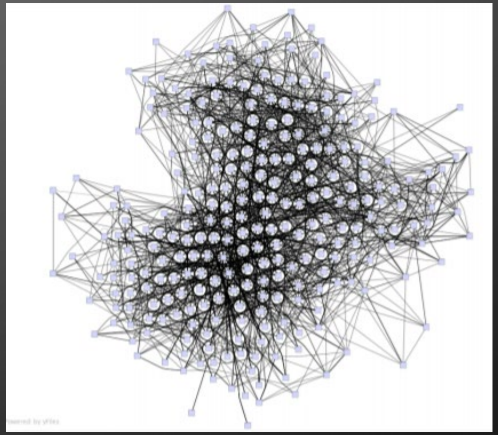
***<u>Example:</u>***



# Small-World Graphs

- Small world graphs often appear in the real world and have very interesting properties.
    - Six degrees of Kevin Bacon
    - Large Scale Computer Networks
    - Brains



Brains

# Example uses of graphs

- Path planning
- Layout routing
- Games and puzzles
- Many kinds of circuits
- Networked systems
- Optimization
- Constraint Satisfaction
- Logical Inference
- Probabilistic Inference
- ..... on and on .....

# Implementing Graphs

- There is no graph data structure in the current standard C++ library.
- It is easy to roll your own using existing standard library containers
- There is also the boost graph library (www.boost.org)

# Graphs Representation

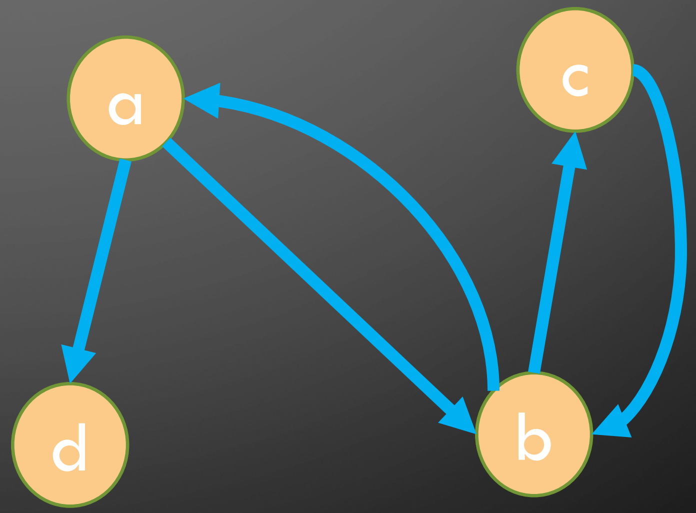
- Adjacency matrix
- Adjacency List
- Pointer based

# Adjacency Matrix

- Given N vertices, the edges are indicated by an NxN matrix
- Undirected graphs have a symmetric matrix
- Weighted graphs have integer or real entries

***Example:***

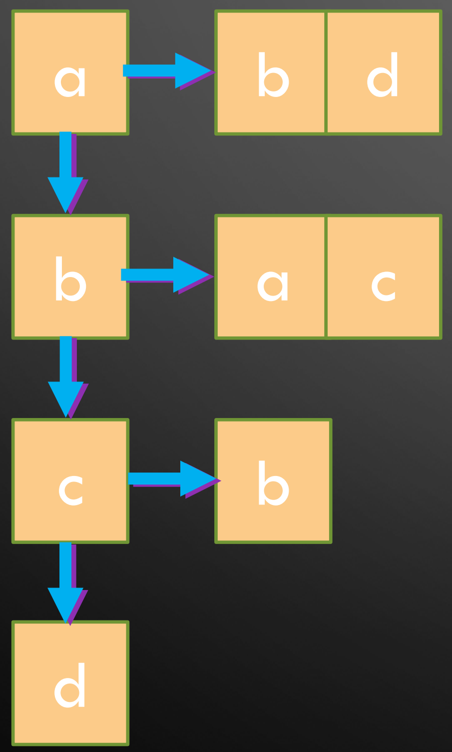
| | a | b | c | d |
|---|---|---|---|---|
| **a** | 0 | 1 | 0 | 1 |
| **b** | 1 | 0 | 1 | 0 |
| **c** | 0 | 1 | 0 | 0 |
| **d** | 0 | 0 | 0 | 0 |

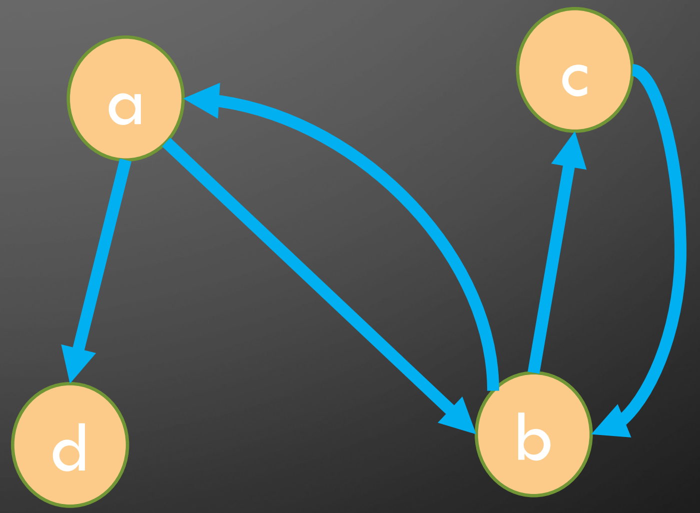


# Adjacency List

- Given N vertices, the edges are indicated by a list of connected vertices for each vertex.
- The lists could be vectors, linked, or trees

***Example:***

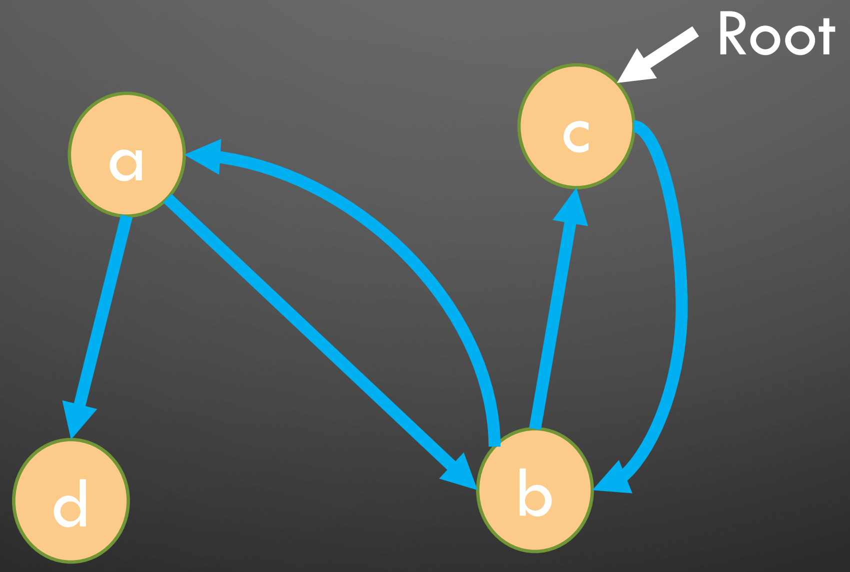


# Pointer based

- Given a pointer to a vertex, which contains pointers to its adjacent vertices.
- Graph must have a root and be connected.

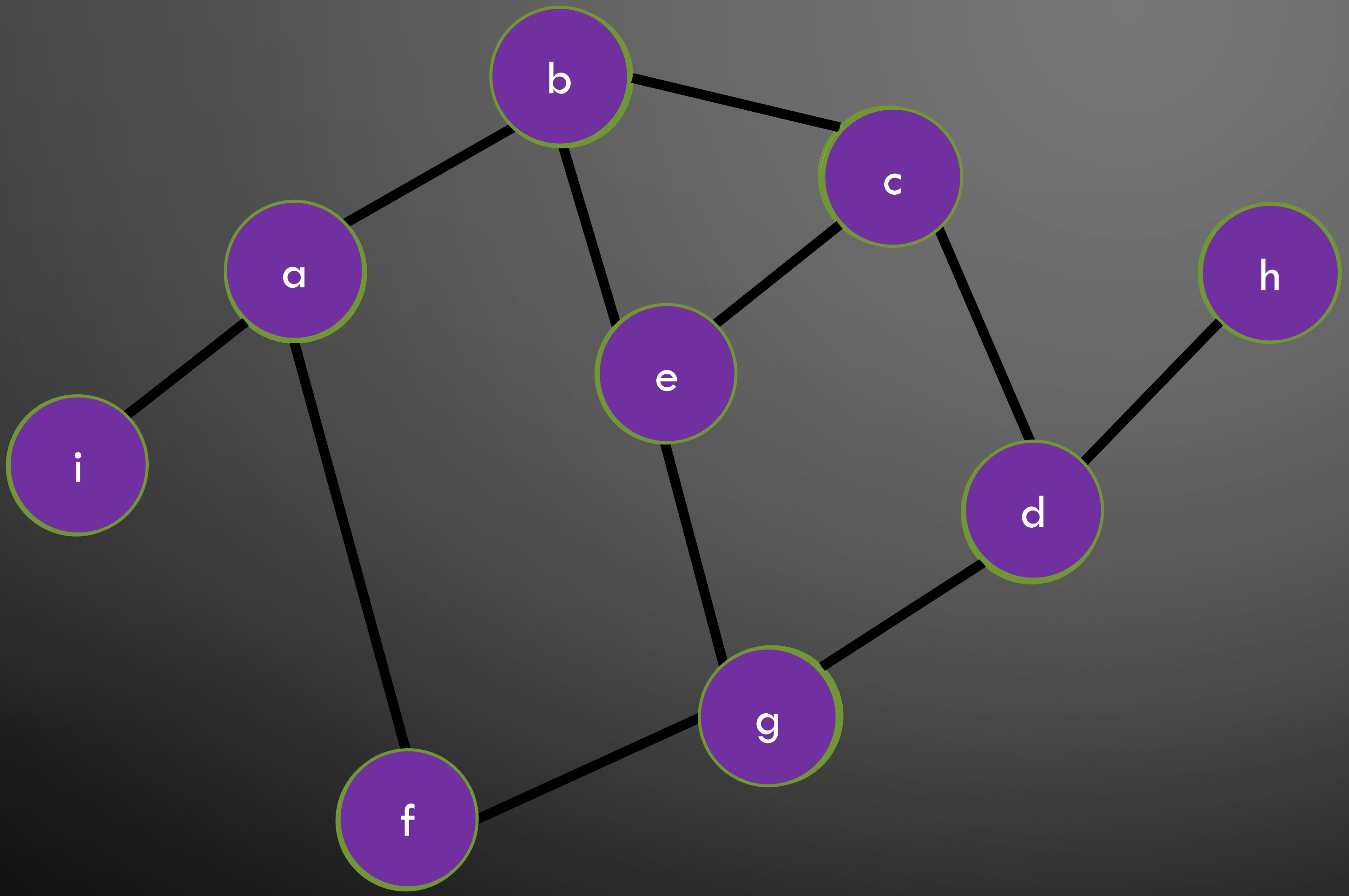
***Example:***



# Advantages/Disadvantages

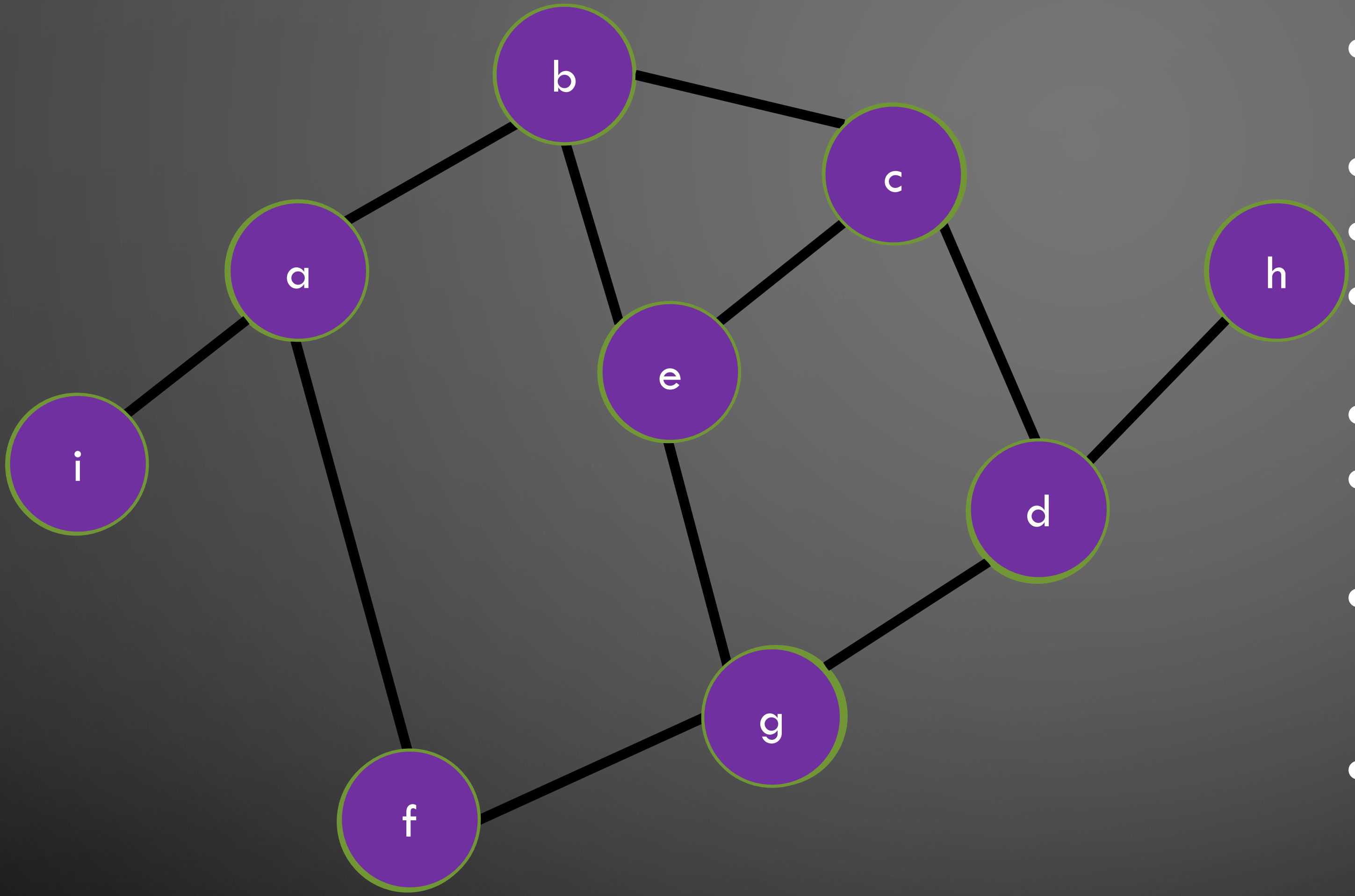
| | **Advantages** | **Disadvantage** |
|---|---|---|
| Adjacency Matrix | -Simple<br>-Fast access to all edges<br>-space efficient for dense graphs | -Space inefficient for sparse graphs |
| Adjacency List | -Space efficient for sparse graphs | -Space inefficient for dense graphs<br>-Access to arbitrary edges slower |
| Pointer based | -Space efficient for sparse graphs | -Space inefficient for dense graphs<br>-Access to arbitrary edges slower<br>-Cannot represent disconnected graphs (easily) |

# Depth First Search




| Visited | Stack |
|---|---|
| • a | a |
| • b | a b |
| • c | a b c |
| • d | a b c d |
| • g | a b c d g |
| • e | a b c d g e |
| • backtrack | a b c d g |
| • f | a b c d g f |
| • backtrack | a b c d |
| • h | a b c d h |
| • backtrack | a |
| • i | a i |
| • backtrack | (empty) |

# Breadth First Search



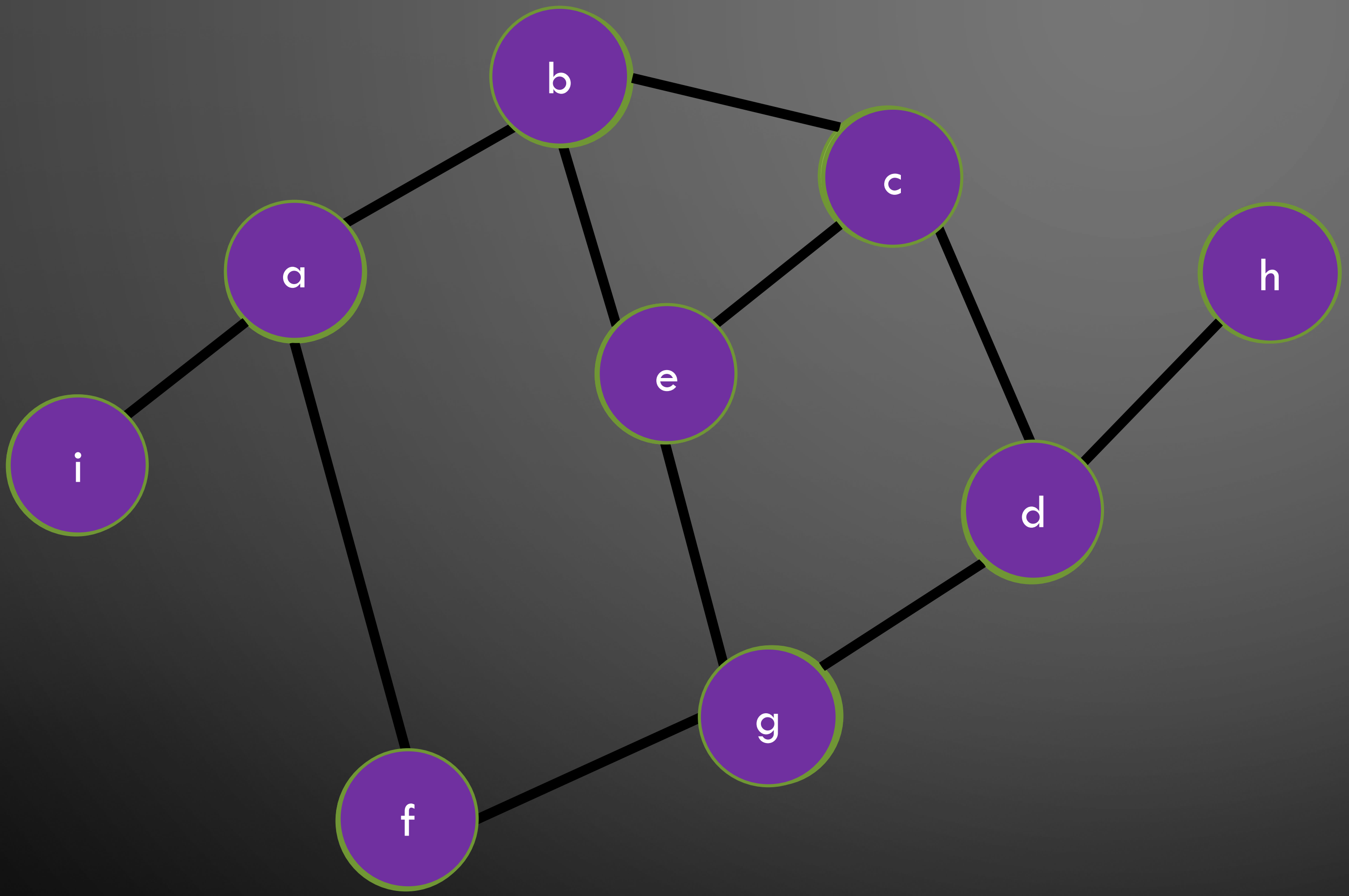

| Visited | Queue |
|---|---|
| • a | a |
|   • deque | (empty) |
| • b | b |
| • f | b f |
| • i | b f i |
|   • deque | f i |
| • c | f i c |
| • e | f i c e |
|   • deque | i c e |
| • g | i c e g |
|   • deque | c e g |
|   • deque | e g |
| • d | e g d |
|   • deque | g d |
|   • deque | d |
|   • deque | (empty) |
| • h | h |
|   • deque | (empty) |

- BFS traversal
  - Visits all vertices adjacent to a vertex before going forward
- BFS is a first visited, first explored strategy
  - Contrast DFS as last visited, first explored

# Spanning Trees

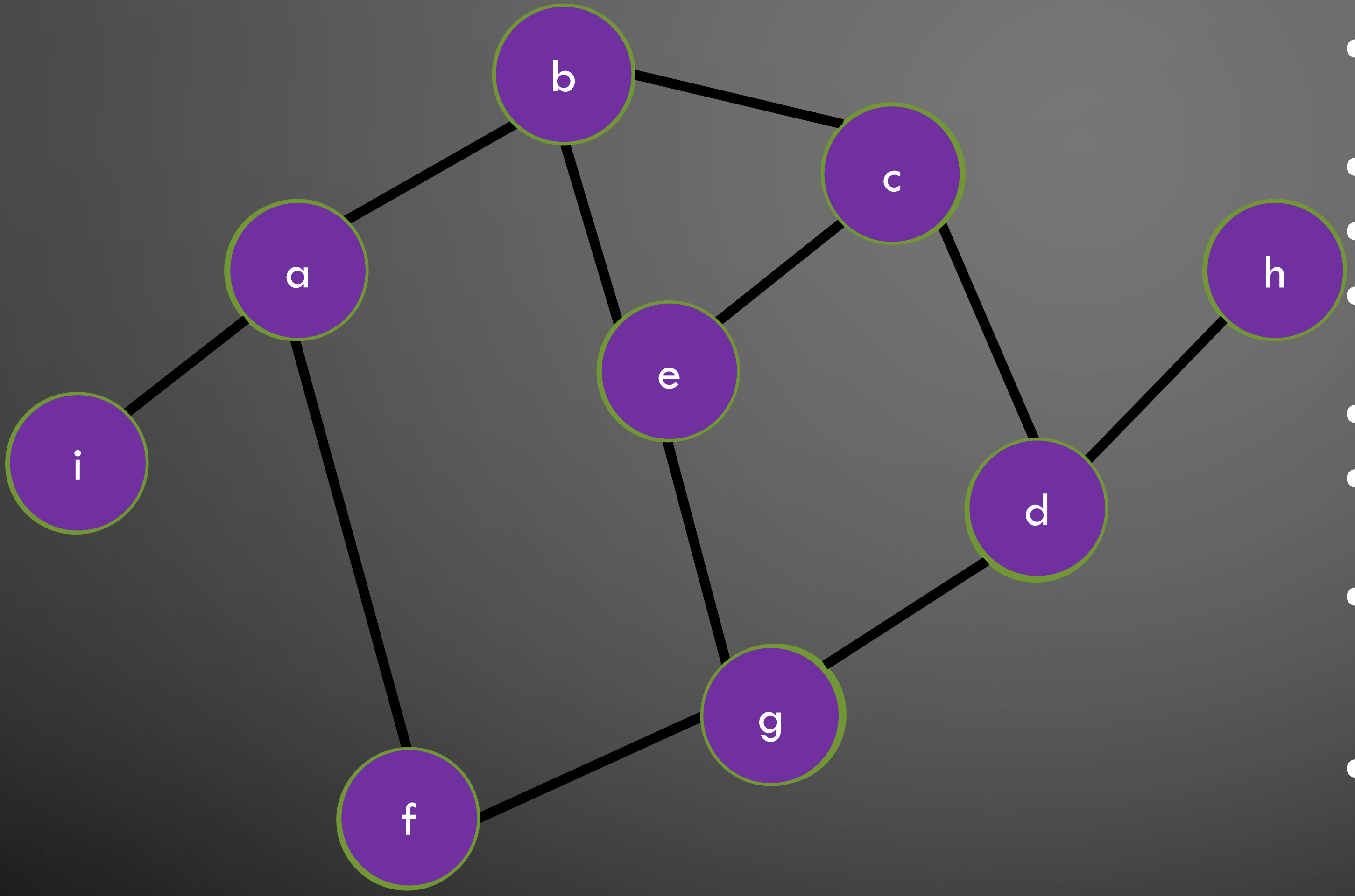
- A tree is an undirected connected graph without cycles
- How to detect a cycle in an undirected graph
  - Connected undirected graph with $n$ vertices must have at least $n - 1$ edges
  - If it has exactly $n - 1$ edges, it cannot contain a cycle
  - With more than $n - 1$ edges, must contain at least one cycle

# Spanning Tree (DFS)



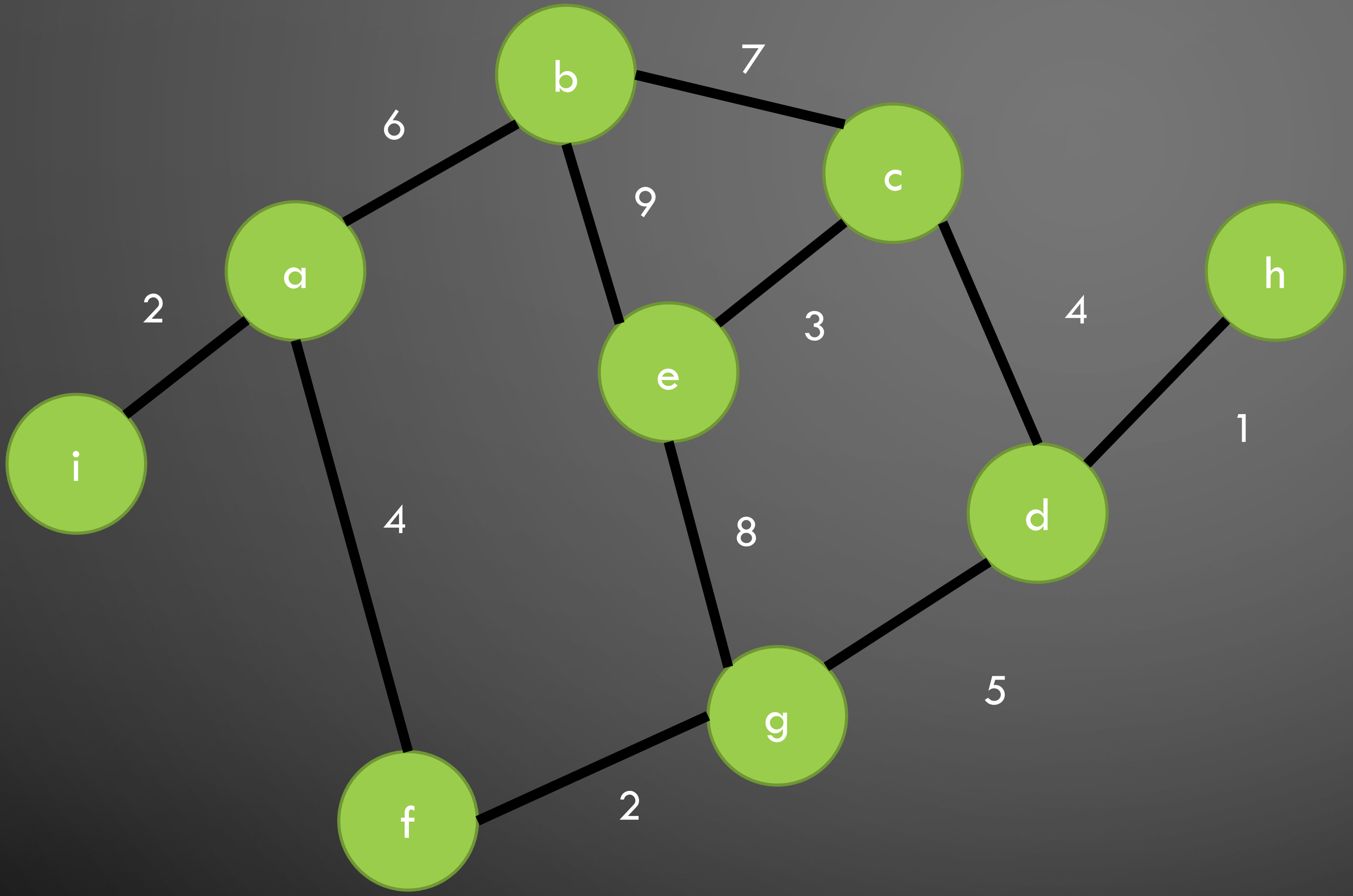

| Visited | Stack |
|---|---|
| • a | a |
| • b | a b |
| • c | a b c |
| • d | a b c d |
| • g | a b c d g |
| • e | a b c d g e |
| • backtrack | a b c d g |
| • f | a b c d g f |
| • backtrack | a b c d |
| • h | a b c d h |
| • backtrack | a |
| • i | a i |
| • backtrack | (empty) |

# Spanning Tree (BFS)

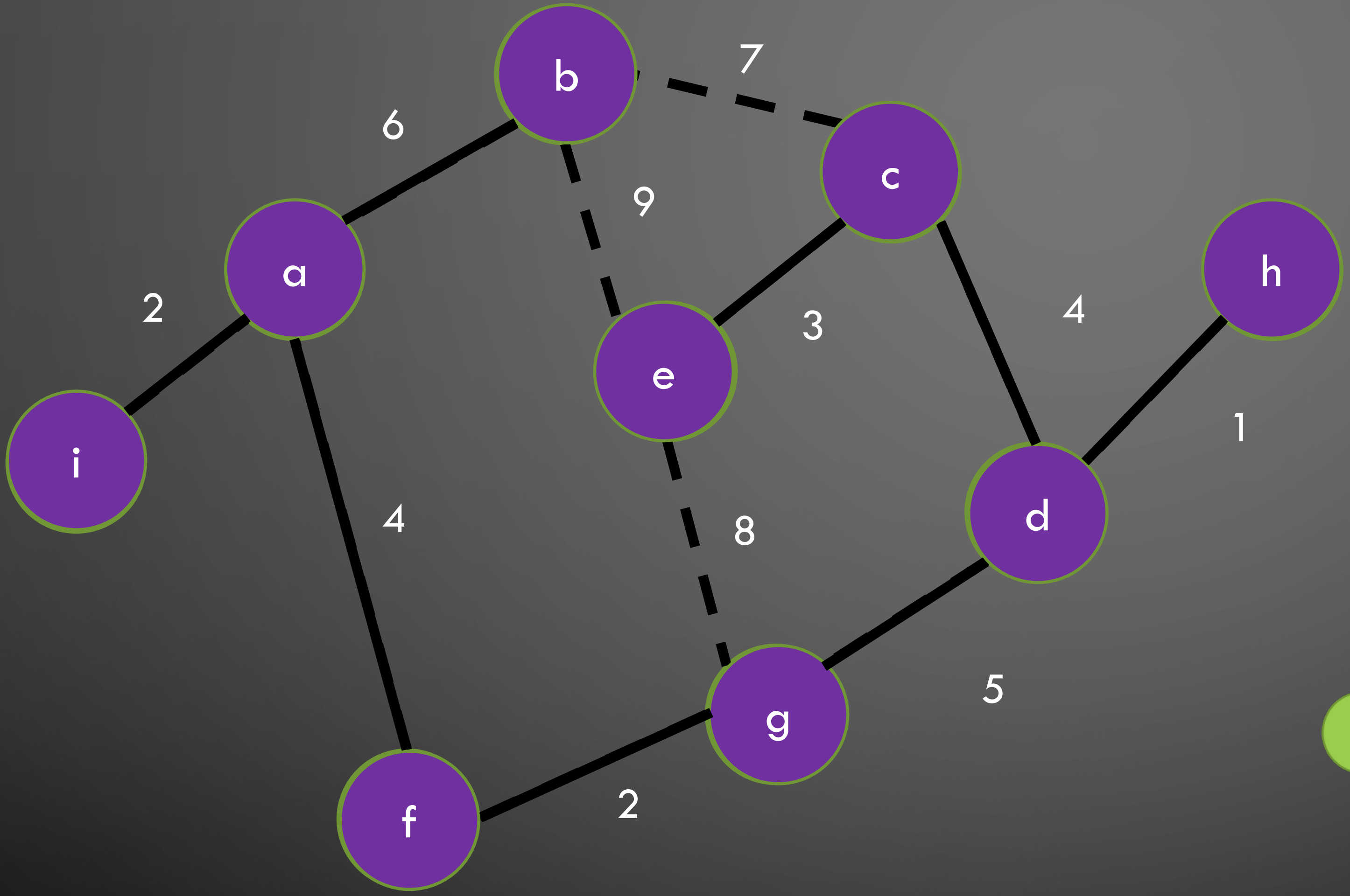


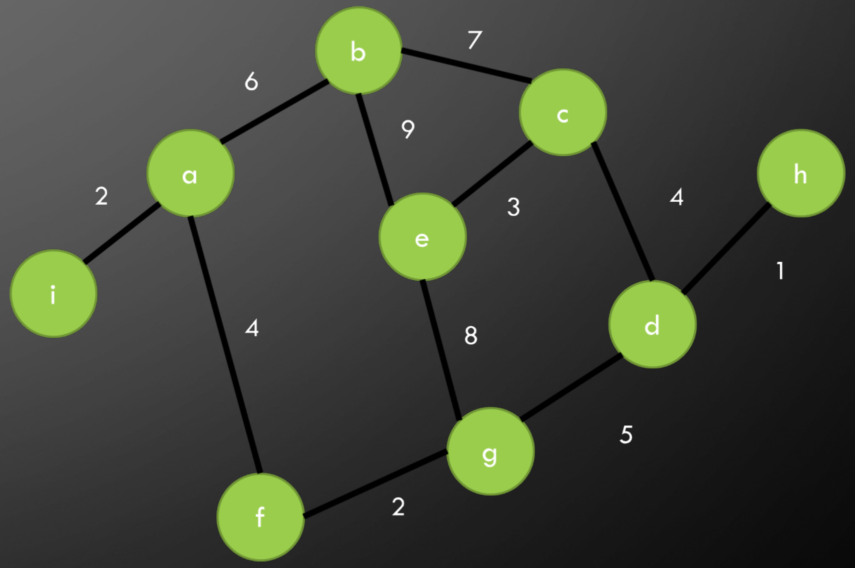

| Visited | Queue |
|---|---|
| • a | a |
|   • deque | (empty) |
| • b | b |
| • f | b f |
| • i | b f i |
|   • deque | f i |
| • c | f i c |
| • e | f i c e |
|   • deque | i c e |
| • g | i c e g |
|   • deque | c e g |
|   • deque | e g |
| • d | e g d |
|   • deque | g d |
|   • deque | d |
|   • deque | (empty) |
| • h | h |
|   • deque | (empty) |

# Minimum Spanning Tree



# Minimum Spanning Tree

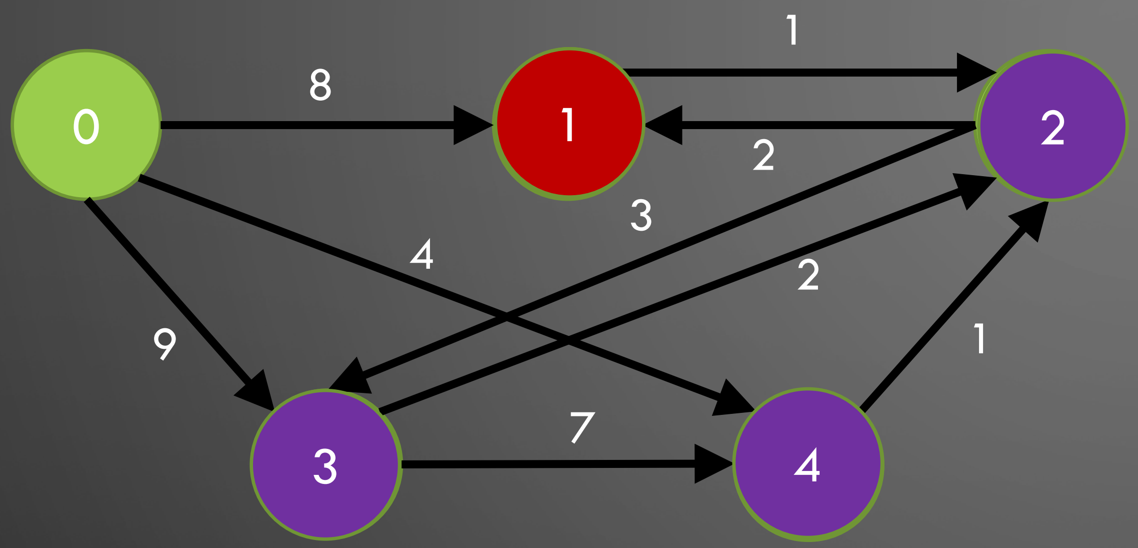


# Shortest path

Cost = 4+1+3

Cost = 4+1+2




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | | 8 | | 9 | 4 |
| **1** | | | 1 | | |
| **2** | | 2 | | 3 | |
| **3** | | | 2 | | 7 |
| **4** | | | 1 | | |

Travel 0→1

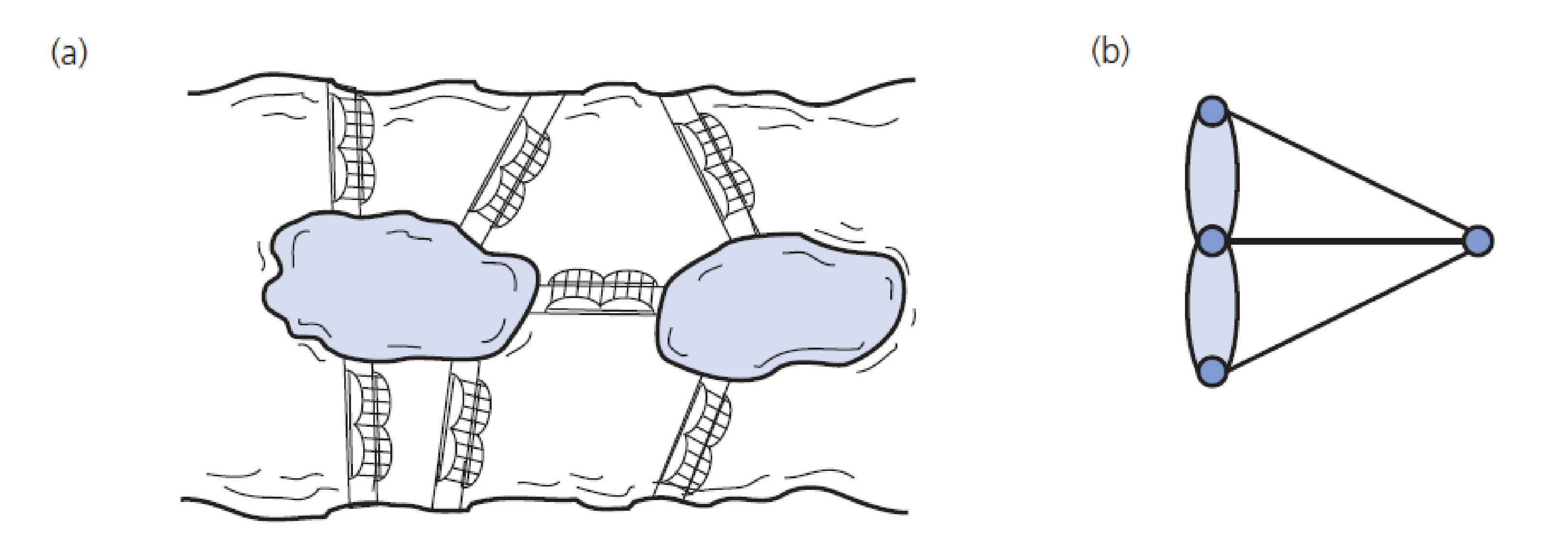
- Cheaper 0→4 than directly 0→1
- 4→2 is my only option
- 2→1 is the cheapest option
  - And where I happen to want to go
  - Cost is 7 < 8, make a note of it?

Travel from 0→3

- Cheaper 0→4 than directly 0→3
- 4→2
- 2→1
  - No path to 3
- 2→3

# What's next?

- Graphs are the natural starting point for a class on algorithms….
  - Euler's bridge problem (aka Euler circuit)
  - Can you be a vertex v, pass through every edge exactly once and end at v?

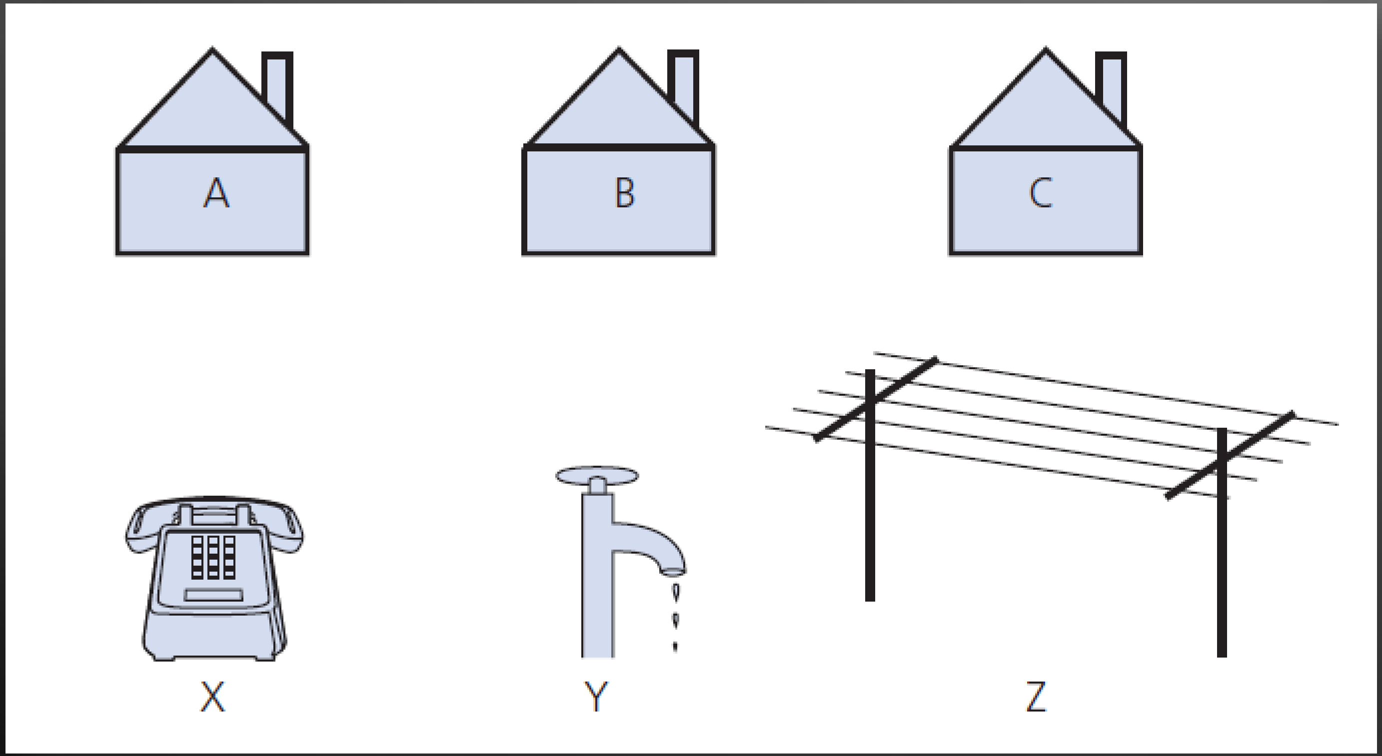


# What's Next?

- Hamilton circuit
  - Begins at vertex *v*
  - Passes through every vertex exactly once
  - Terminates at *v*
- Variation is "traveling salesperson problem"
  - Visit every city on his route exactly once
  - Edge (road) has associated cost (mileage)
  - Goal is determine least expensive cycle
- This is NP Complete

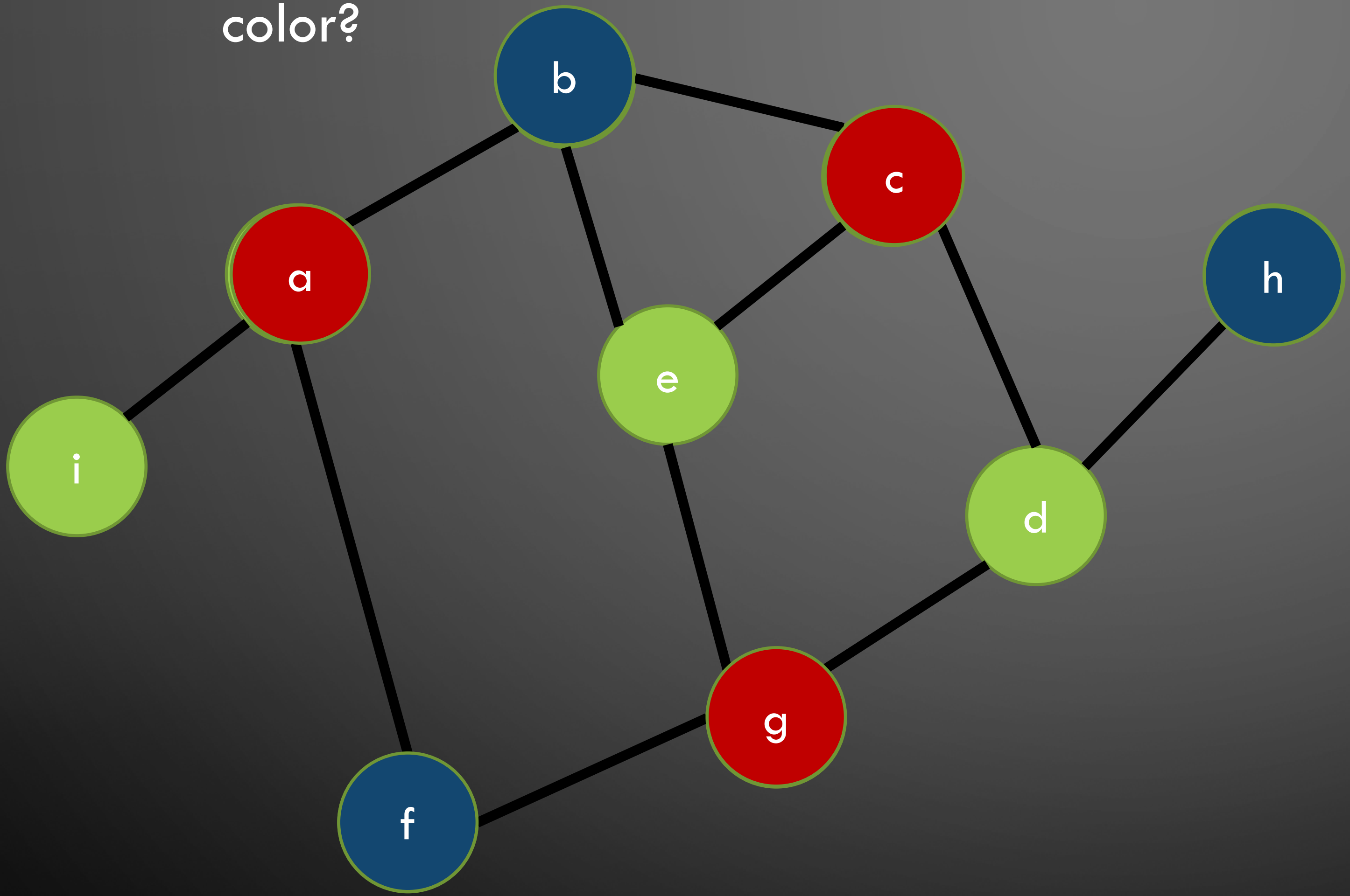
# What’s next?

- Can I draw a graph that connects every utility to every house without crossing?
  - Try it!  I bet you can’t!
- A graph that does not cross is a *planar graph*



# What's Next? Graph Coloring

- Can I color a planar graph such that every adjacency doesn't have the same color?




- Let's say I want to partition my graph for parallel processing, now I can!
- Is this a good partitioning? Minimum number of colors? Best balance?

# Exercise (no submission required)

- Carrano Chapter 20: Exs. 1, 4, 11, 14
- HW 8 due today
- HW 9, ICE 10, and P5 due on Tuesday
- Final Exam on Tuesday