Middlebury

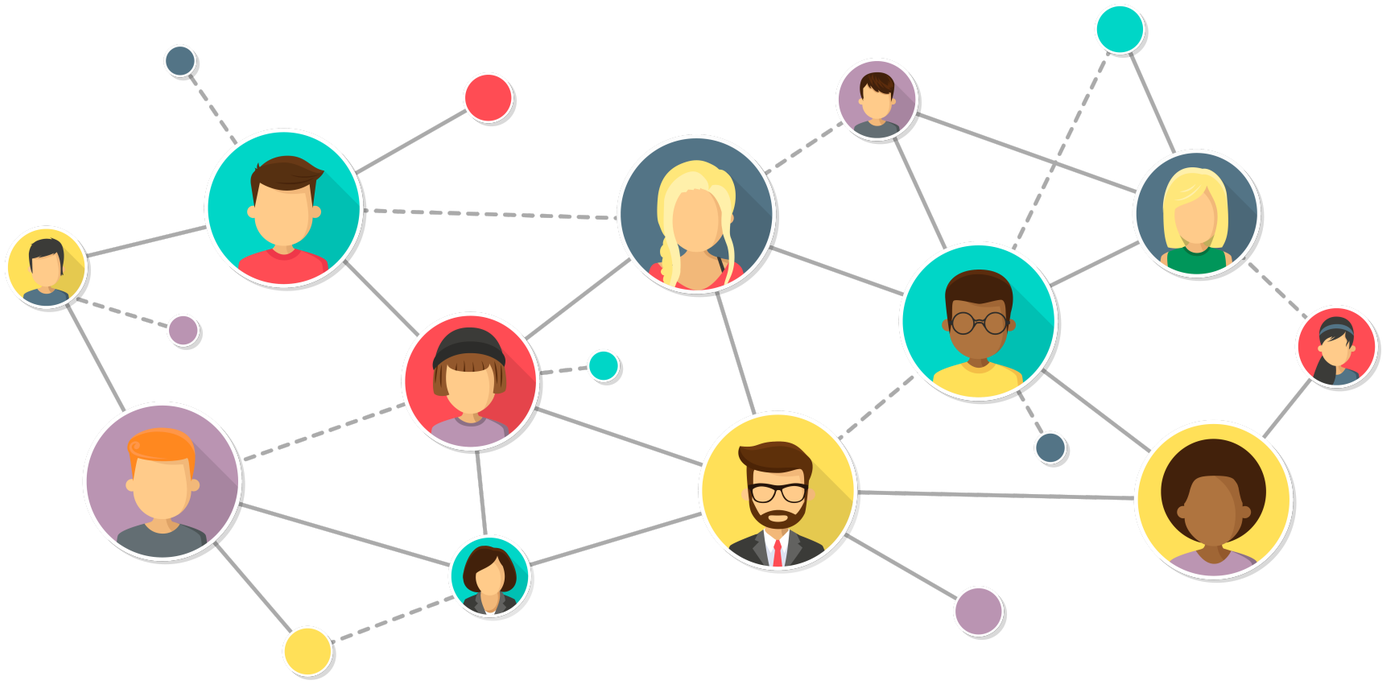
CSCI 201: Data Structures

Fall 2024

Lecture 11T: Graphs

Goals for today:

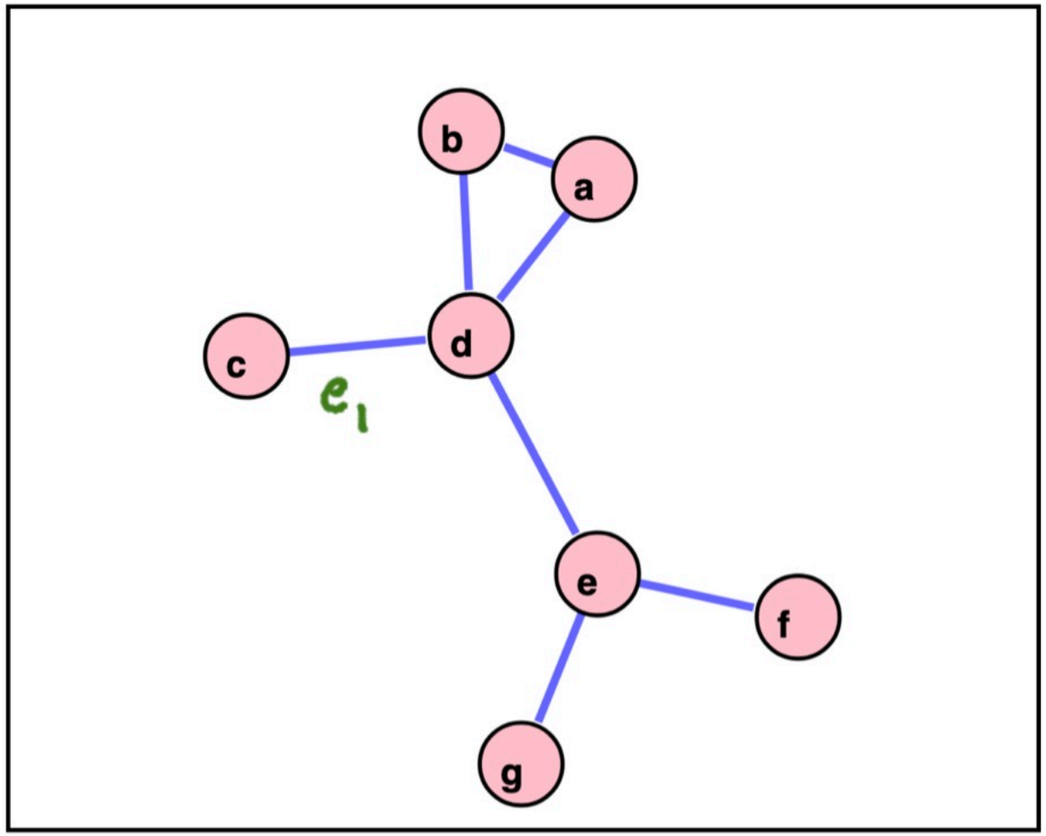
- What is a **graph**? What are **edges**? What are **vertices**/**nodes**? What is a **path**? What is a **directed edge**? What is a **cycle**? What do **weights** represent? What is the **degree** of a node?
- Different types of graphs: **directed**, **complete**, **connected**.
- Represent a graph using an **adjacency matrix**.
- Represent a graph using **adjacency lists**.
- Represent graphs in `Java`.



Terminology: a graph consists of *two sets*: nodes (a.k.a. vertices) and edges.

'c' and 'd' are adjacent

e_1 is incident to nodes 'c' and 'd'



degree('d') = 4

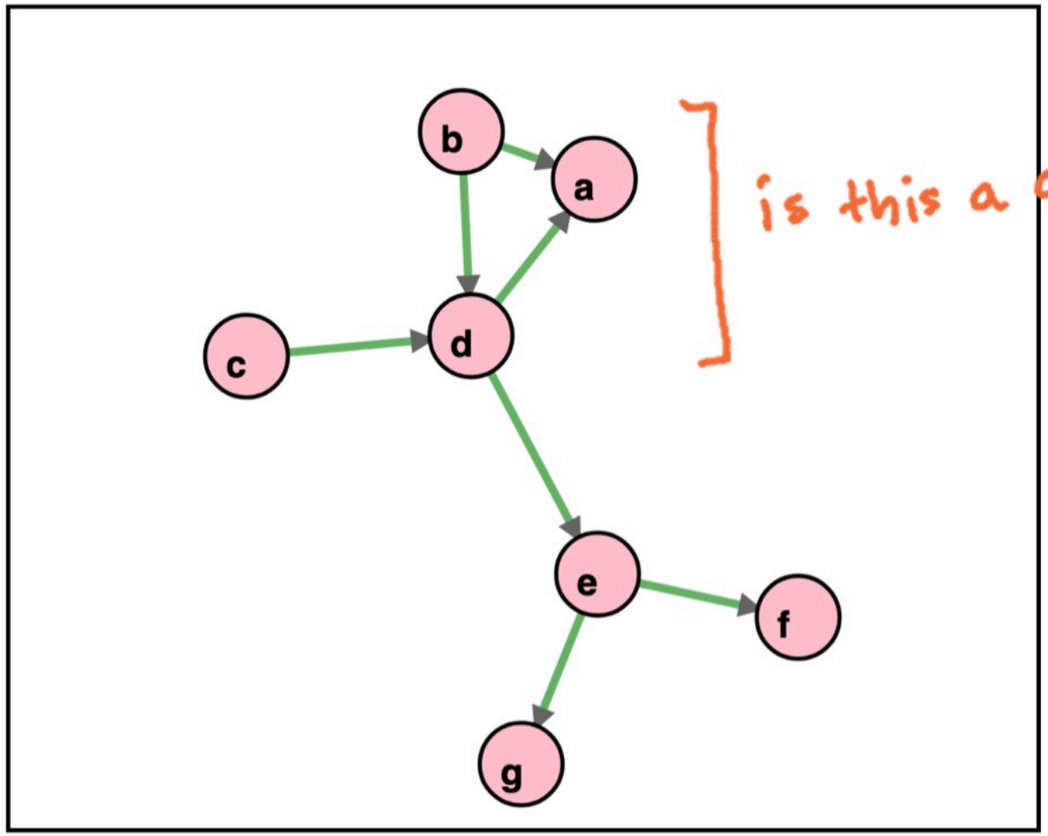
degree('f') = 1

b-d-e-f is a path

- Two vertices are **adjacent** if there is an edge between them. The edge is **incident** to both vertices.
- The **degree** of a node is the number of edges incident to it.
- A **path** is a sequence of nodes $p_1,\ p_2, \ldots, p_k$ where there is an edge $(p_i,\ p_{i+1})$ in the set of edges. No edge is repeated. A *simple path* has no repeated vertices.
- A **cycle** is a path where the first and last node are the same.

b-d-a-b is a cycle

Edges can also have a direction.



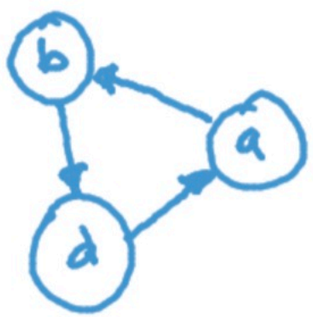
is this a cycle? no

b → a is an edge

but

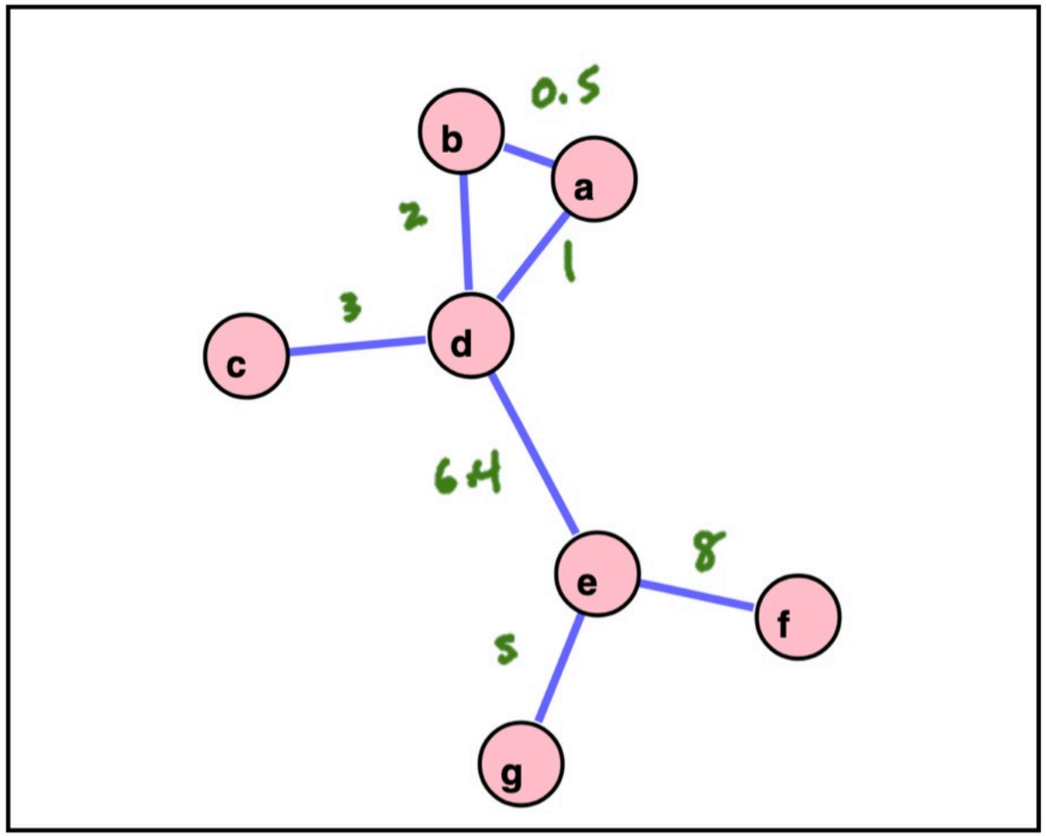
a → b is not an edge

Is there a cycle in this graph?



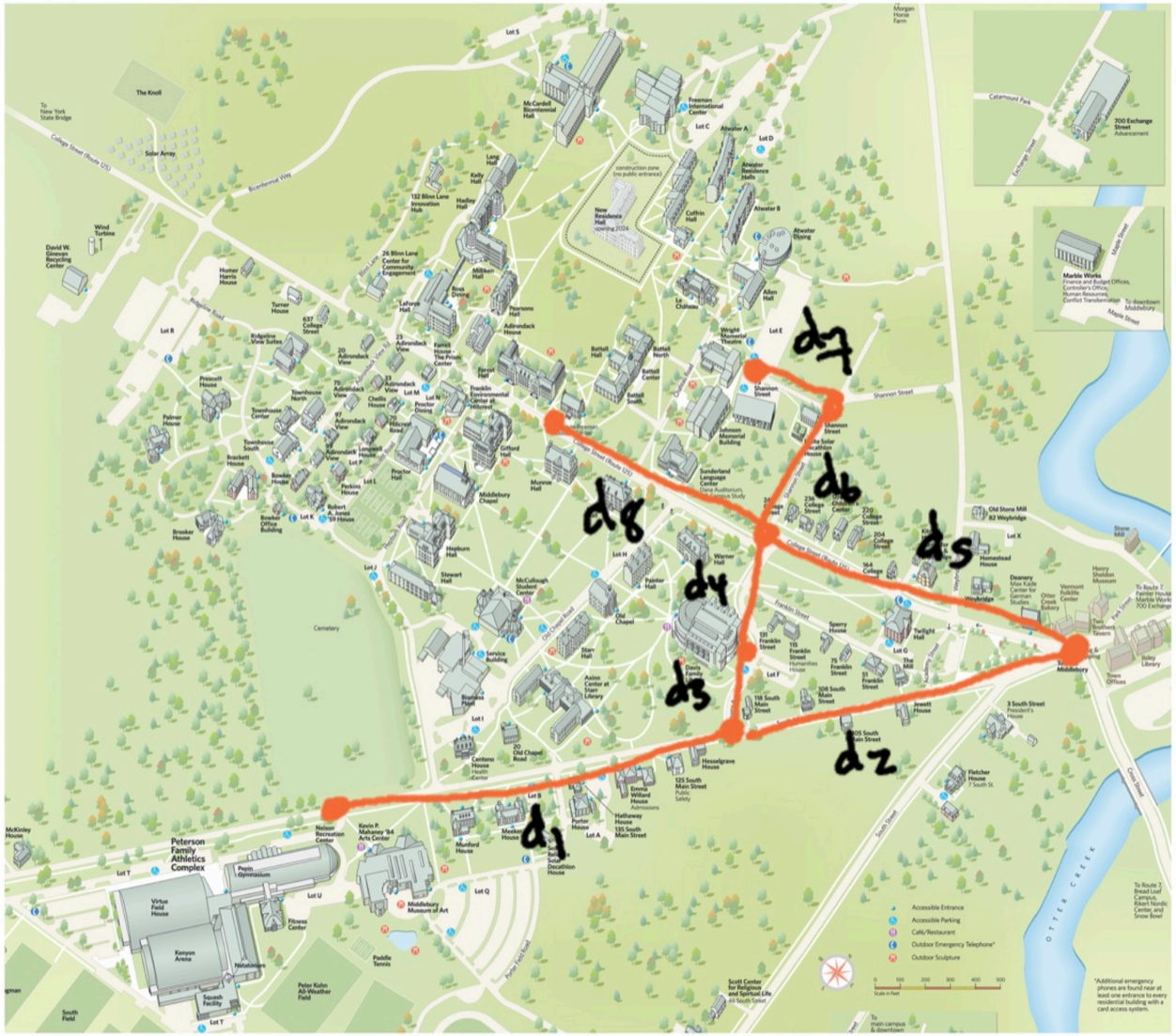
← this would have a cycle.

Edges can also have weights.



What can weights be used for?

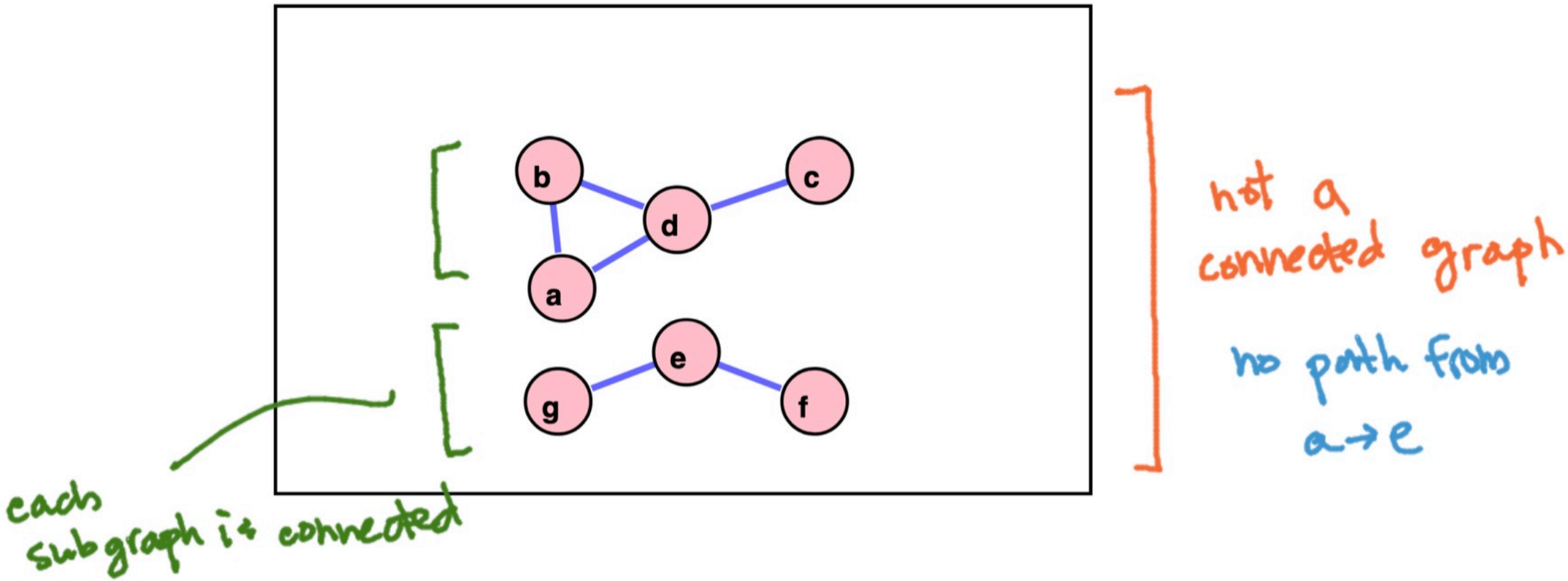
We might want to calculate the *shortest path* between two nodes.



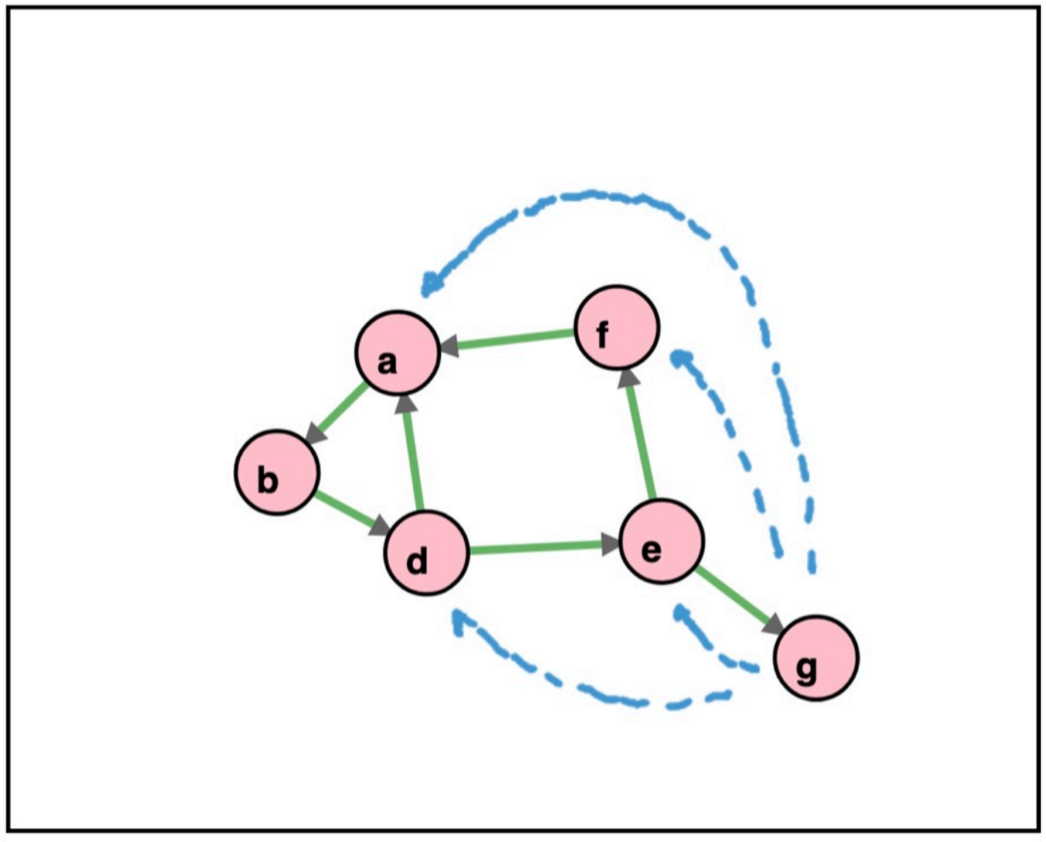
treat weights as the distance between nodes. (d_i's)

We could also treat weights as driving time (depends on application)

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



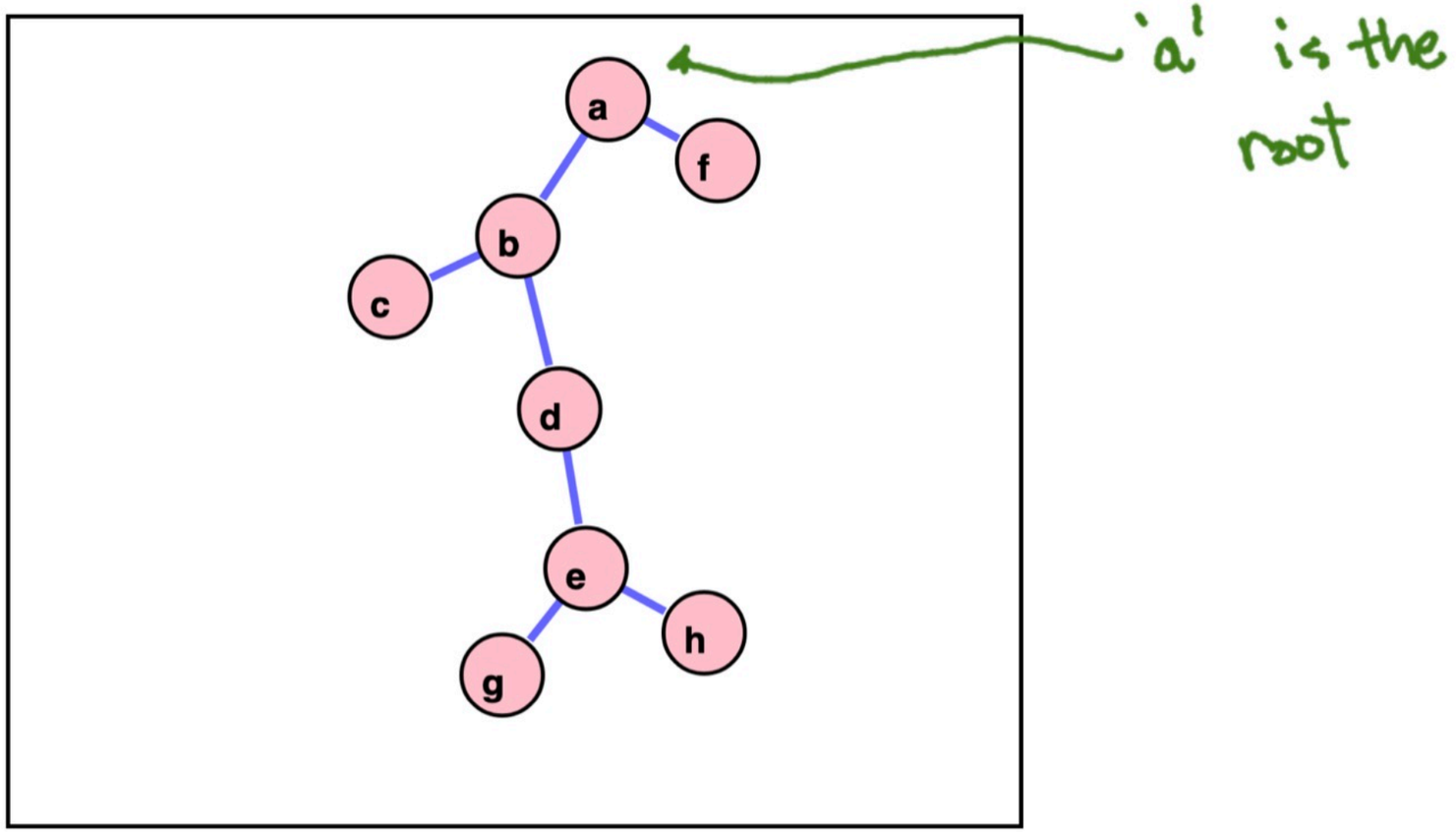
Strongly connected (for directed graphs): every pair of nodes is reachable by a path.



Which edge can we add to make this graph strongly connected?

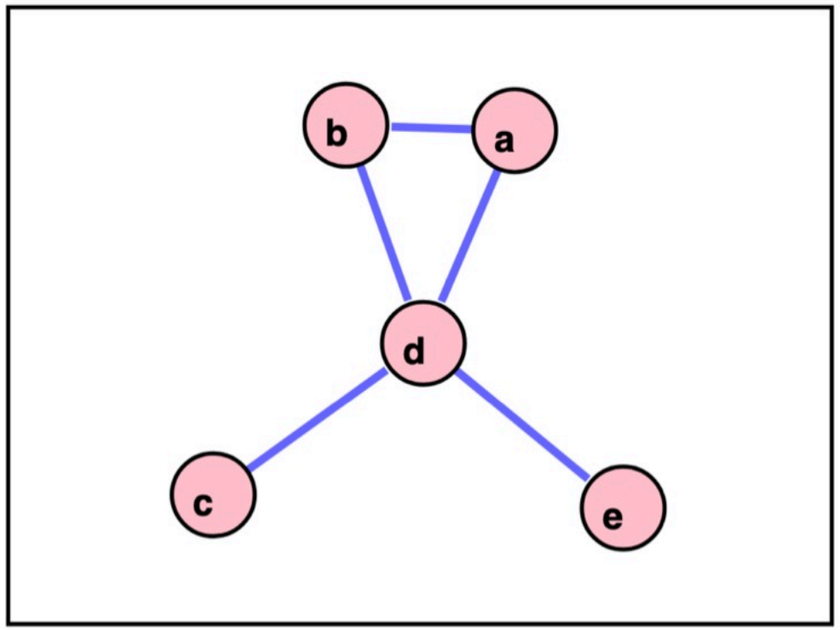
We have seen graphs before! A tree is a special type of graph.

A tree is a **connected**, **undirected** graph without any cycles.



Trees can be rooted or free.

Representing graphs: adjacency matrix and adjacency lists.



2d array

↑

adjacency matrix:

each entry is { 1 if edge u to v
0 otherwise.

	a	b	c	d	e
a	0	1	0	1	0
b	1	0	0	1	0
c	0	0	0	1	0
d	1	1	1	0	1
e	0	0	0	1	0

↑

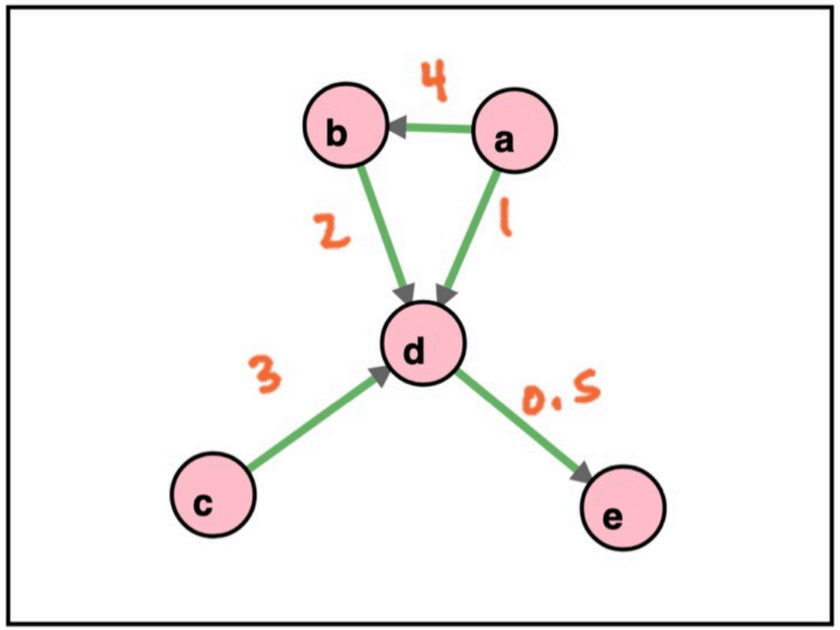
#entries = #nodes x #nodes

wasting memory storing 0's

adjacency lists

node	list
a	[b, d]
b	[a, d]
c	[d]
d	[a, b, c, e]
e	[d]

Representing directed graphs (also with edge weights).



adjacency matrix:

	a	b	c	d	e
a	0	4	0	1	0
b	0	0	0	2	0
c	0	0	0	3	0
d	0	0	0	0	0.5
e	0	0	0	0	0

adjacency lists

node	list of adj. (node, weight)
a	[(b,4), (d,1)]
b	[(d,2)]
c	[(d,3)]
d	[(e,0.5)]
e	[]

Implementing this in `Java` using adjacency lists (or adjacency sets).

```java
public class Graph<Node> {

  // adjacency lists: node -> set of adjacent nodes
  private HashMap<Node, HashSet<Node>> adj;

  public Graph() {
    adj = new HashMap<>();
  }

  public void addEdge(Node a, Node b) {
    if (!adj.containsKey(a)) {
      adj.put(a, new HashSet<>());
    }
    if (!adj.containsKey(b)) {
      adj.put(b, new HashSet<>());
    }
    adj.get(a).add(b);
    adj.get(b).add(a);
  }

  public boolean hasEdge(Node a, Node b) {
    // or adj.get(b).contains(a) since undirected
    return adj.get(a).contains(b);
  }

  Set<Node> getNodes() {
    return adj.keySet();
  }
}
```

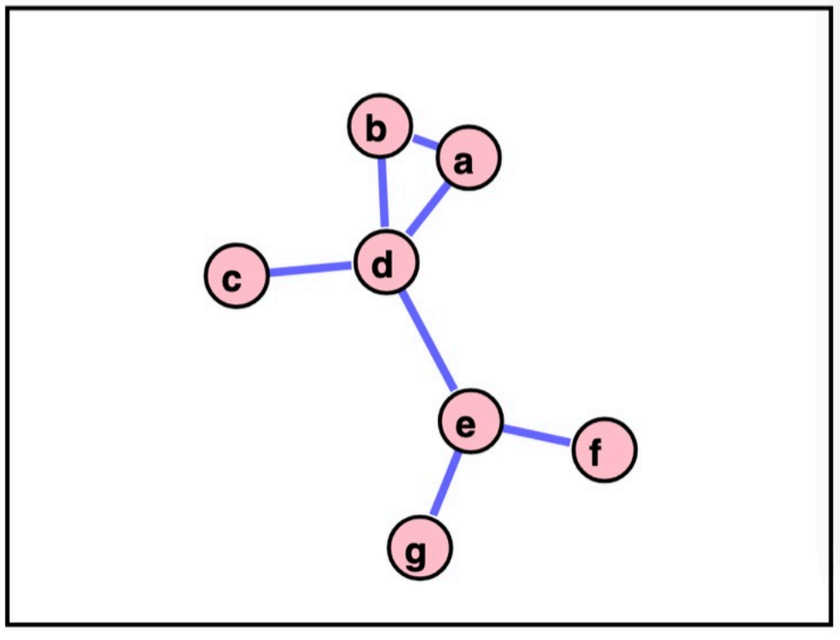
```java
public static void main(String[] args) {
  Graph<Character> graph = new Graph<>();

  graph.addEdge('a', 'b');
  graph.addEdge('a', 'd');
  graph.addEdge('b', 'd');
  graph.addEdge('c', 'd');
  graph.addEdge('d', 'e');
  graph.addEdge('e', 'f');
  graph.addEdge('e', 'g');

  System.out.println(graph.hasEdge('b', 'a'));

  System.out.println(graph.getNodes());
}
```

Brainstorm: how can we design a `getEdges()` method?

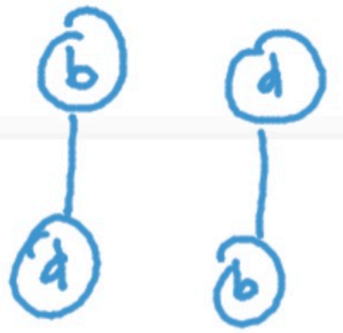
Retrieving the set of edges.

Maybe use a helper Edge class?

```java
public class Graph<Node> {

  private HashMap<Node, HashSet<Node>> adj;

  HashSet<Edge> getEdges() {
    HashSet<Edge> edges = new HashSet<>();
    for (Node u : adj.keySet()) {
      HashSet<Node> list = adj.get(u);
      for (Node v : list) {
        edges.add(new Edge(u, v));
      }
    }
    return edges;
  }
}
```




```java
class Edge {
  public Node u;
  public Node v;

  public Edge(Node u, Node v) {
    this.u = u;
    this.v = v;
  }
}
```



should be the same

int hashCode() and boolean equals methods

```java
class Edge {
  public Node u;
  public Node v;

  public Edge(Node u, Node v) {
    this.u = u;
    this.v = v;
  }

  @Override
  public int hashCode() {
    return 31 * Math.min(u.hashCode(), v.hashCode())
             + Math.max(u.hashCode(), v.hashCode());
  }

  @Override
  @SuppressWarnings("unchecked")
  public boolean equals(Object otherObj) {
    Edge otherEdge = (Edge) otherObj;
    return (u.equals(otherEdge.u) && v.equals(otherEdge.v)) ||
           (u.equals(otherEdge.v) && v.equals(otherEdge.u));
  }

  public String toString() {
    return "(" + u.toString() + ") -- (" + v.toString() + ")";
  }
}
```

Additional notes:

- Next class: graph searching algorithms.
- Complete Exit Ticket 11T by end of today.
- Rest of class (if time): work on Homework 9.
- **Homework 9** currently due on Friday 11/22: implement a hash table with linear probing to handle collisions. Extend? Please complete the poll on our Ed discussion board.