CSCI 136: Data Structures and Advanced Programming

Lecture 30

Graphs, part 2

Instructor: Kelly Shaw

Williams

Topics

Reachability

Connectedness

Graph operations

Graph representations

Your to-dos

1. Read **before Wed**: *Bailey*, Ch. 13.1.
2. Lab 10 (partner lab), **due Tuesday 11/29 by 10pm**.
3. Last quiz this Fri/Sat.

Announcements

Sean Barker '09, Bowdoin College

Friday, Dec 2 @ 2:35pm
Computer Science Colloquium – Wege TCL 123
Smart Meters for Smart Cities: Data Analytics in Energy-Aware Buildings

The proliferation of smart energy meters has resulted in many opportunities for next-generation buildings. Energy-aware "smart buildings" may optimize their energy consumption and provide convenience and economic benefits through analysis of their meter data. However, storing and analyzing this data presents computational challenges, especially when conducted at scale. In this talk, I discuss our work on several problems in this space, focusing particularly on efficient compression of smart meter data and the disaggregation of building-wide consumption into individual device consumption. Our work in these areas aims to support the development of sustainable, energy-efficient smart cities and smart grids.

Graphs

Useful theorems

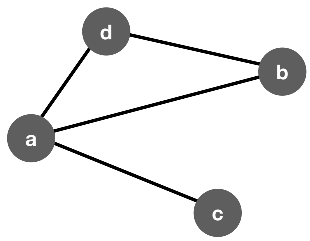
(about undirected graphs)

- If there is a **walk** from **u** to **v**, then there is a **walk** from **v** to **u**.
- If there is a **walk** from **u** to **v**, then there is a **path** from **u** to **v** (and from **v** to **u**).
- If there is a **path** from **u** to **v**, then there is a **simple path** from **u** to **v** (and **v** to **u**).
- Every **circuit** through **v** contains a **cycle** through **v**.
- Not every **closed walk** through **v** contains a **cycle** through **v**.

Degree

The **degree** of a vertex **v** is the **number of edges** incident to **v**.

Denoted: `deg(v)`

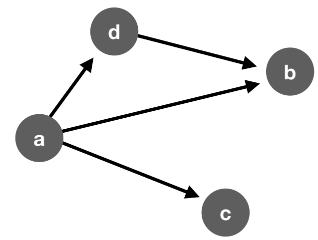


What is the degree of **c**?   of **a**?

Degree on Digraphs

The **in-degree** of a vertex **v** is the **number of incoming edges** incident to **v**.

Denoted: `in-deg(v)`

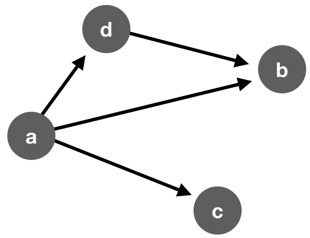


What is the in-degree of **c**?   of **a**?

Degree on Digraphs

The **out-degree** of a vertex **v** is the **number of outgoing edges** incident to **v**.

Denoted: `out-deg(v)`



What is the out-degree of **c**? of **a**?

Degree theorem

For any graph **G = (V, E)**

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

where **|E|** is the number of edges in **G**.

Proof: by induction on **|E|**.

Hint: How does **removing an edge** change the equation?

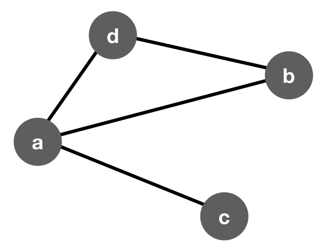
Reachability and Connectedness

“Siri, can I drive from Boston to Hong Kong?”

“Siri, which places can I drive to?”

Reachability

A vertex **v** in **G** is **reachable** from vertex **u** in **G** if there is a **path** from **u** to **v**.

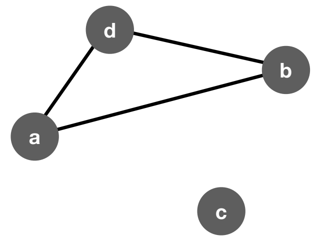


For an **undirected** graph **G**, **v** is **reachable** from vertex **u** iff **u** is **reachable** from vertex **v**.

Is **c** **reachable** from **d**? Yes.

Connectedness

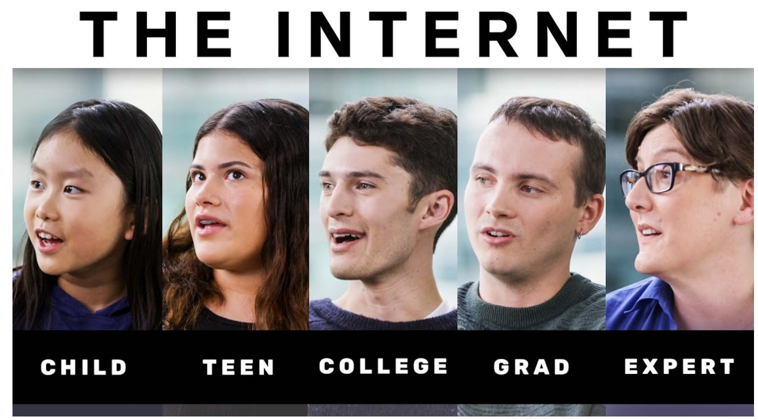
An undirected graph **G** is **connected** if for every pair of vertices **u, v** in **G**, **v** is **reachable** from **u**.



The set of all **vertices reachable from v**, along with all **edges** of **G** connecting any two of them, is called the **connected component of v**.

(note that the connected component is itself a graph)

Bonus video to watch on your own

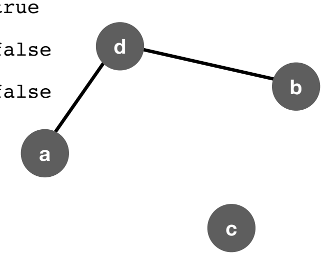


youtu.be/0EqKnvzo3no

Graph ADT operations

Fundamental graph ADT operations

```
adjacent(a, d) = true
adjacent(a, b) = false
adjacent(a, c) = false
```



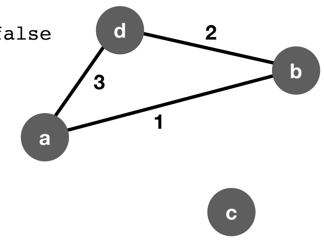

```
bool adjacent(Vertex u, Vextex v):
```

Given vertices **u** and **v**, are they **adjacent**?

(i.e., share an edge?)

Fundamental graph ADT operations

```
incident(a, 1) = true
incident(a, 2) = false
```



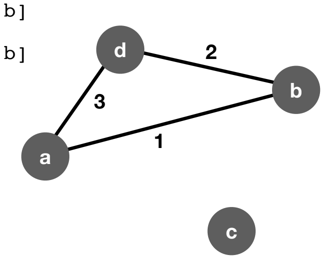

```
bool incident(Vertex v, Edge e):
```

Given vertex **v** and edge **e**, are they **incident**?

(i.e., is v an endpoint of edge e?)

Fundamental graph ADT operations

```
vertices(1) = [a, b]
vertices(2) = [d, b]
```

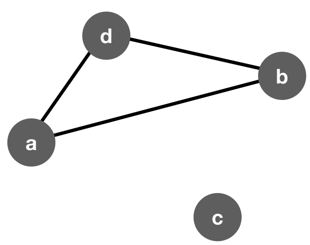



```
Vertex[] vertices(Edge e):
```

Given edge **e**, what are its **end points**?

Fundamental graph ADT operations

```
degree(a) = 2
degree(c) = 0
```

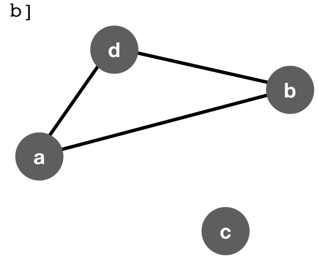



```
int degree(Vertex v):
```

Given vertex **v** how many vertices are **adjacent**?

Fundamental graph ADT operations

```
neighbors(a) = [d, b]
neighbors(c) = []
```




```
Vertex[] neighbors(Vertex v):
```

Given vertex **v** what other vertices are **adjacent**?

Reachability?

How might we implement the following method

```
bool reachable(Vertex u, Vextex v)
```

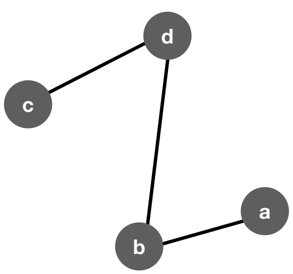
using the fundamental operations just described?

```
bool adjacent(Vertex u, Vextex v)

bool incident(Vertex v, Edge e)

Vertex[] vertices(Edge e)

int degree(Vertex v)

Vertex[] neighbors(Vertex v)
```

Graph data structures

Adjacency list

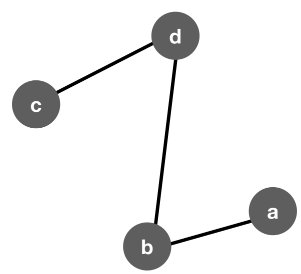
An **adjacency list** is a data structure for representing a finite graph. It consists of a **list of unordered lists**.




```
[[c,d],[d,b],[a,b]]
```

Object-oriented adjacency list

There are many variants on adjacency lists. The most common is the **object-oriented adjacency list** that stores **a list of adjacent vertices** in each vertex object.

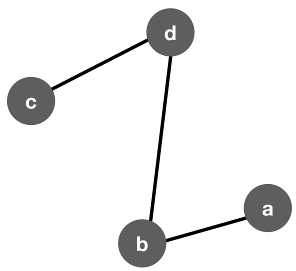


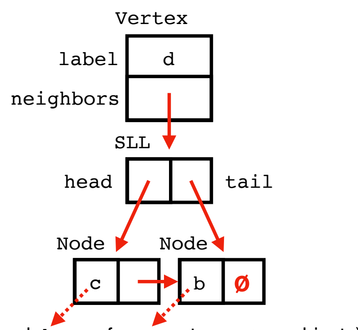

```
a: [b]
b: [a,d]
c: [d]
d: [b,c]
```

Adjacency list

Object-oriented adjacency list:

```
public class Vertex<T> {
    T label;
    List<Vertex<T>> neighbors = new SinglyLinkedList<>();
    …
}
```

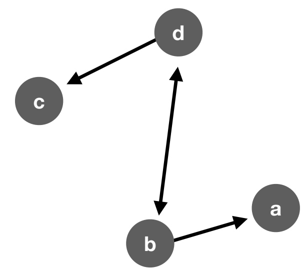




(strictly speaking, `c` and `d` are references to `Vertex` objects)

Adjacency list

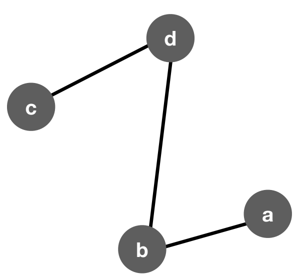
This latter version is **especially thrifty** for **directed graphs**.




```
a: []
b: [a,d]
c: []
d: [b,c]
```

Adjacency matrix

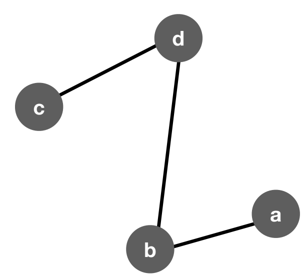
An **adjacency matrix** is a data structure for representing a finite graph. It consists of a **square matrix** (usually implemented as an array of arrays). In the simplest case, the **elements** of the matrix indicate **whether an edge is present**. Elements on the diagonal are **defined as zero**.



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

Adjacency matrix

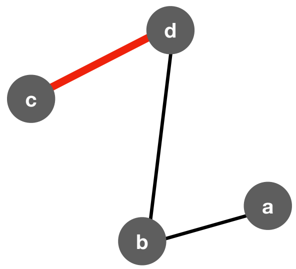
In an **undirected graph**, the adjacency matrix is **symmetric**.



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

Adjacency matrix

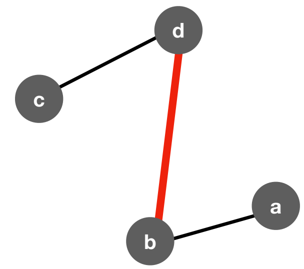
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	1
c	0	0	0	**1**
d	0	1	**1**	0

Adjacency matrix

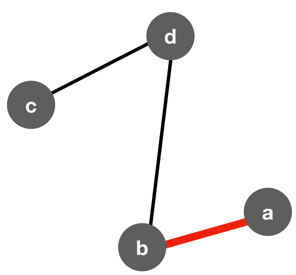
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	1	0	0
b	1	0	0	**1**
c	0	0	0	1
d	0	**1**	1	0

Adjacency matrix

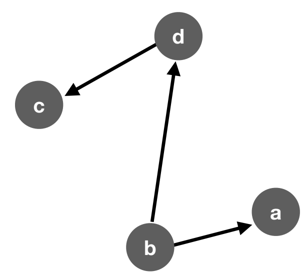
In an **undirected graph**, the adjacency matrix is **symmetric**.



	a	b	c	d
a	0	**1**	0	0
b	**1**	0	0	1
c	0	0	0	1
d	0	1	1	0

Adjacency matrix

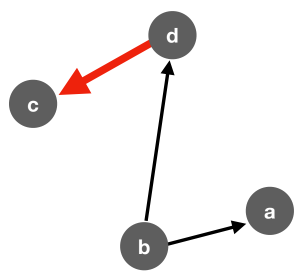
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form, with **from** being on the **vertical** axis.



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

Adjacency matrix

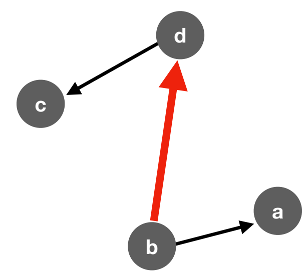
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form, with **from** being on the **vertical** axis.



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

Adjacency matrix

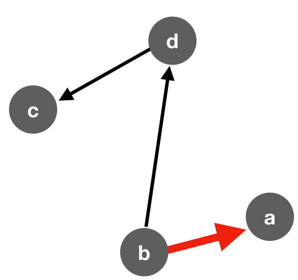
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form, with **from** being on the **vertical** axis.



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

Adjacency matrix

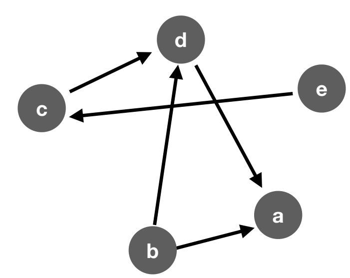
In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, **from→to**, is conventionally encoded in **row-major** form, with **from** being on the **vertical** axis.



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

Activity

Write down both **adjacency matrix** and **adjacency list** representations for this graph.



Which one do you think is better for this graph?

Recap & Next Class

Today:

Graph operations

Graph representations

Next class:

Connectedness algorithms