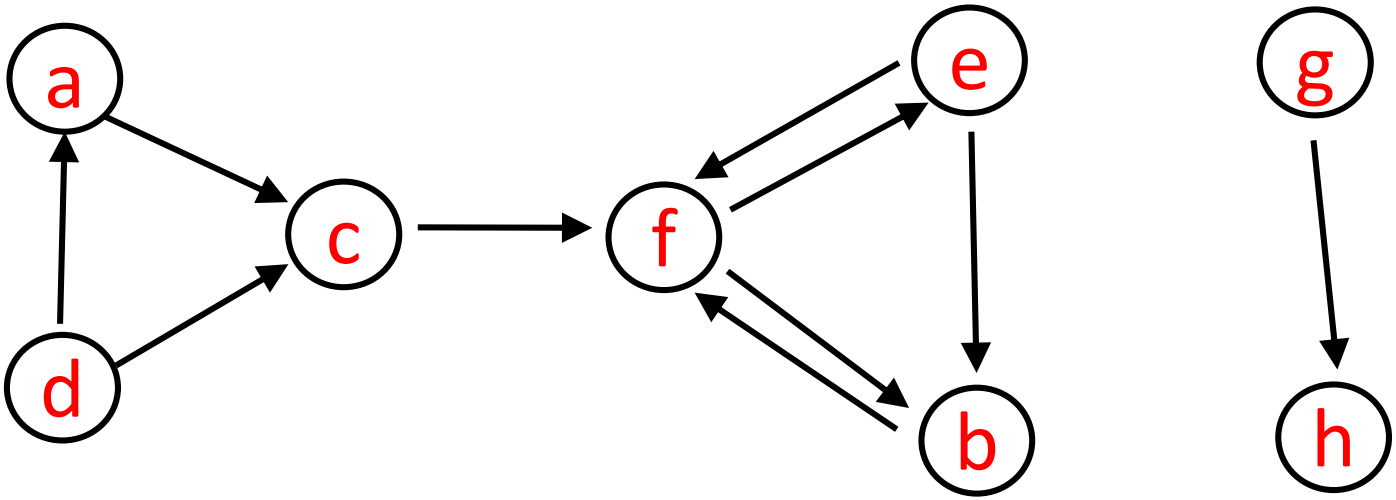
COMP 250

Lecture 30

graphs

Nov. 19, 2018

Example



Definition

A *directed graph* is a set of *vertices*

$$V = \{v_i \ : \ \ i \in 1, \ldots, n \}$$

and set of ordered pairs of these vertices called *edges*.

$$E = \{ \left(v_i \ , v_j\right) \ : \ \ i, j \in 1, \ldots, n \}$$

Examples (Directed)

Vertices	Edges
airports	
web pages	
Java objects	
methods in program (compile time)	

Examples (Directed)

Vertices	Edges
airports	flights
web pages	links (URLs)
Java objects	references
methods in program (compile time)	method A calls B (compile time)

Definition

A *undirected graph* is a set of *vertices*

$$V = \{v_i \ : \ \ i \in \ 1, \ldots, n \ \}$$

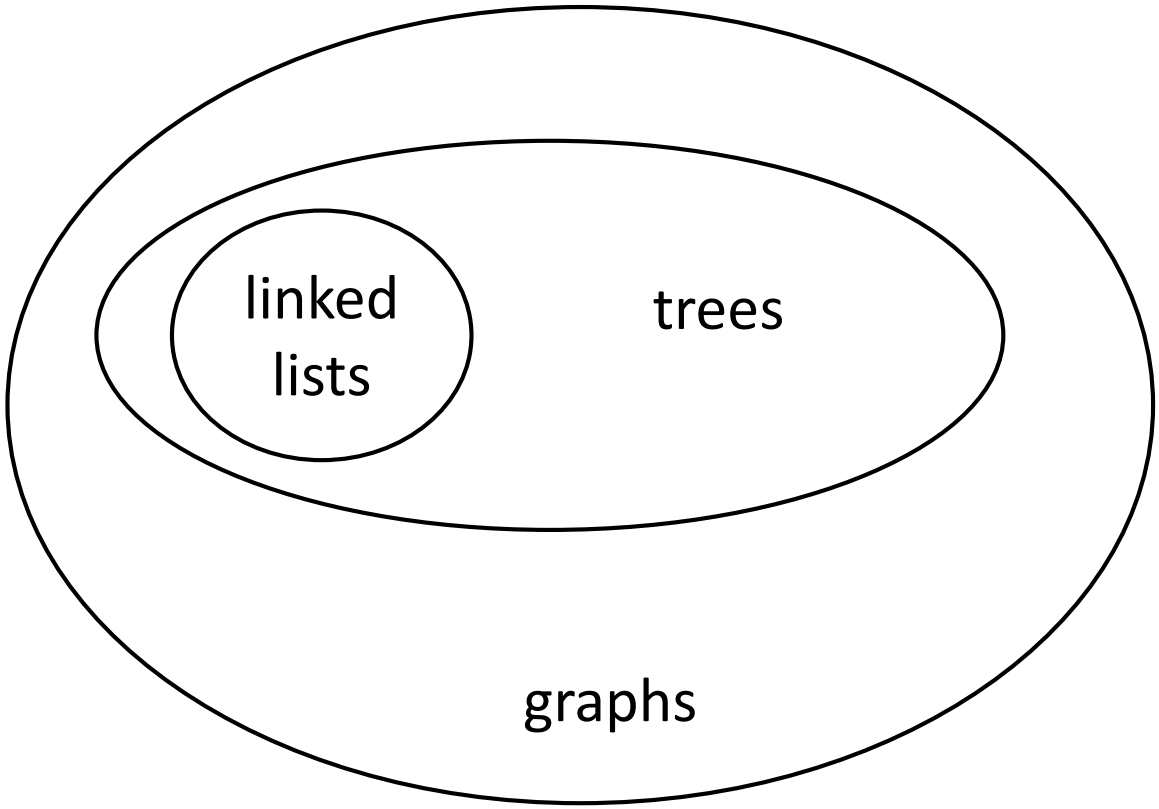
and set of unordered pairs, again called *edges*.

$$E = \{ \ \{v_i, v_j\} : \ \ i, j \in \ 1, \ldots, n \ \}$$

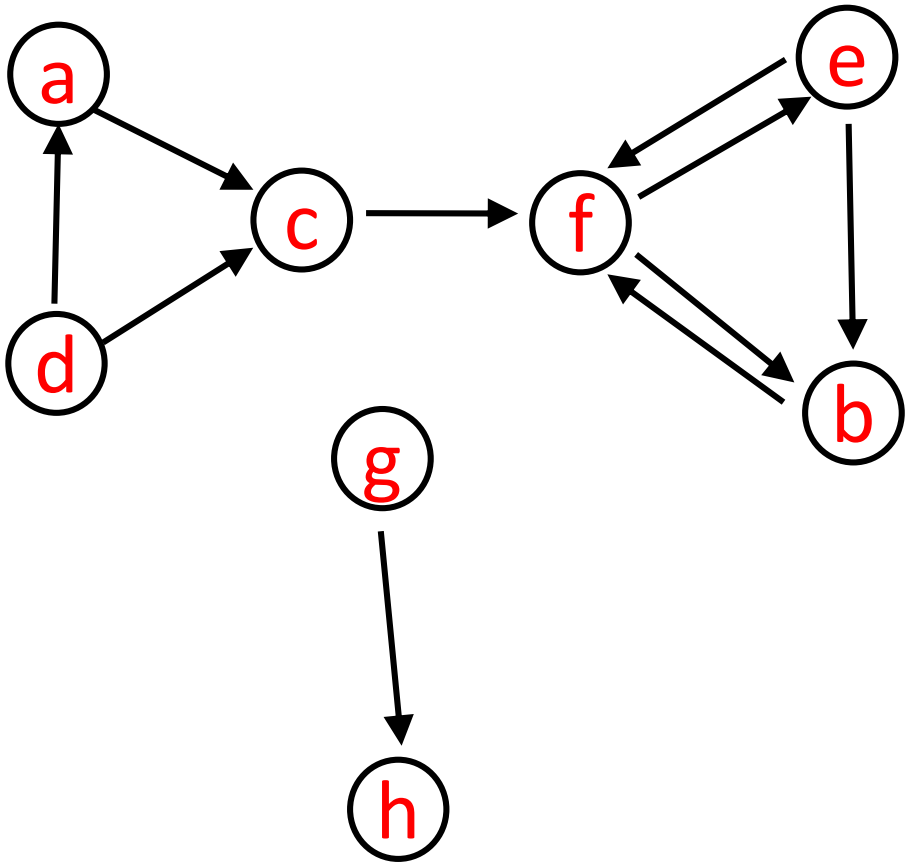
Examples (Undirected)

Vertices	Edges
Facebook users	friends
events(to be scheduled)	conflicts (someone needs to attend both)
road intersections	roads (two way)

We will deal with directed graphs only.

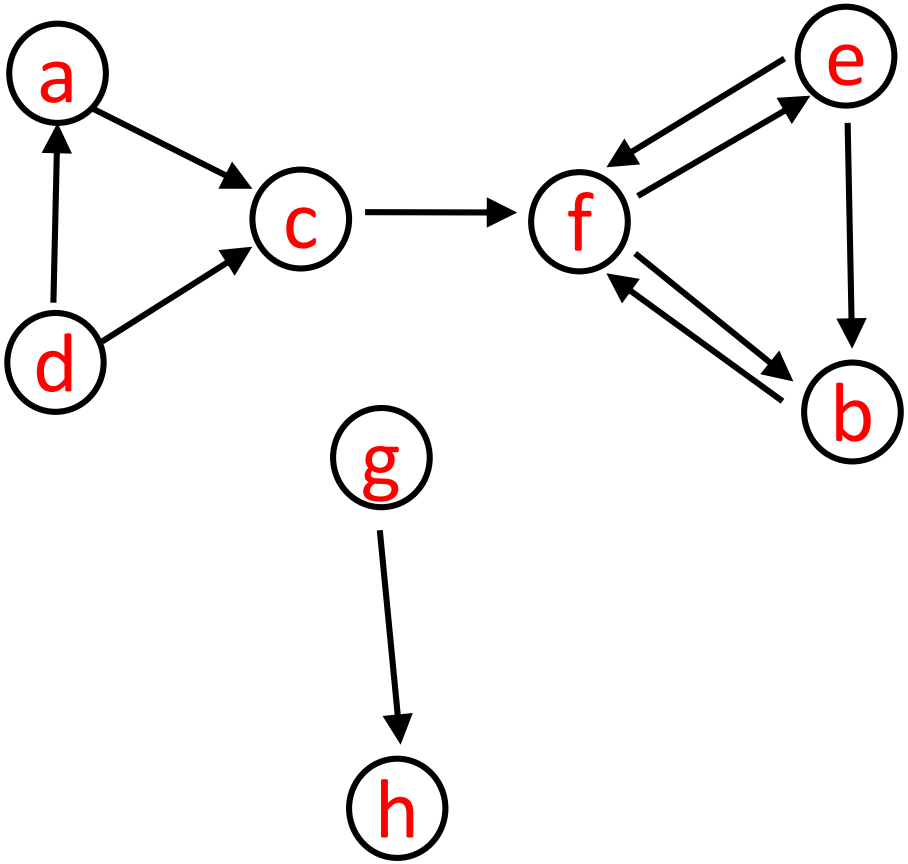

linked
lists
trees
graphs

Terminology: "in degree"



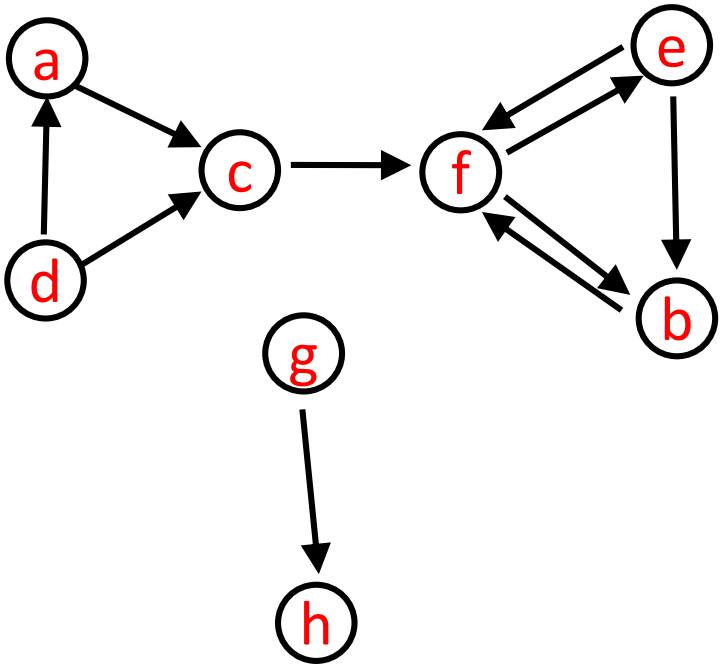
v	in degree
a	1
b	2
c	2
d	0
e	1
f	3
g	0
h	1

Terminology: "out degree"



v	out degree
a	1
b	1
c	1
d	2
e	2
f	2
g	1
h	0

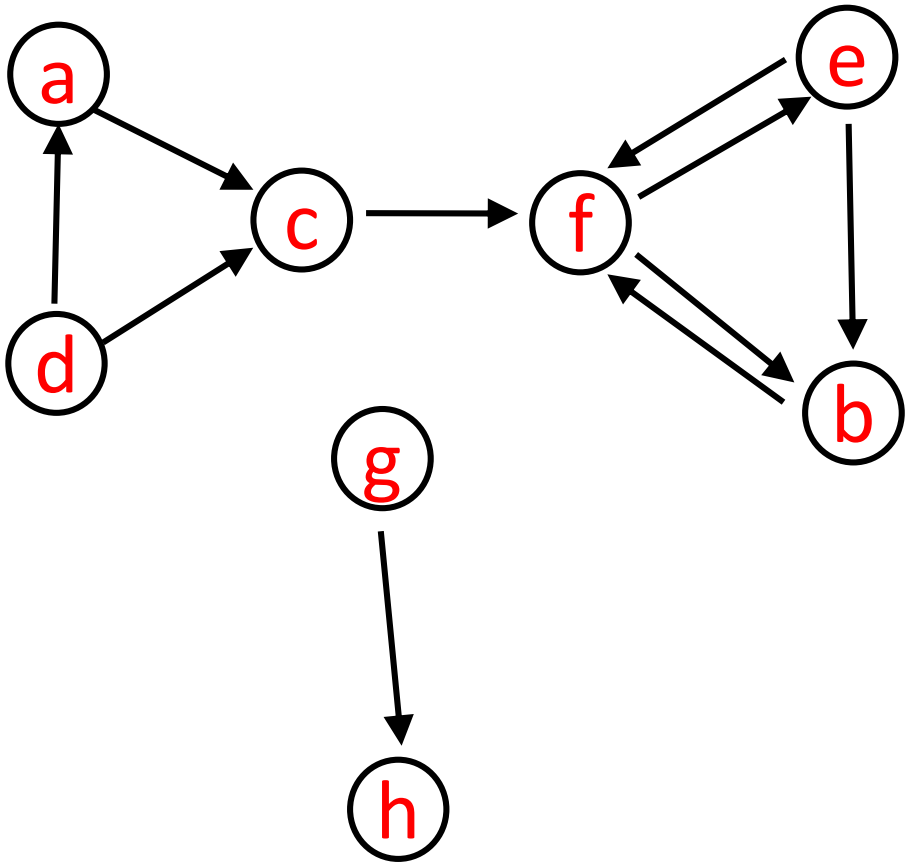
Example: web pages



In degree: How many web pages link to some web page (e.g. to f) ?

Out degree: How many web pages does some web page link to (e.g. from f) ?

Terminology: path

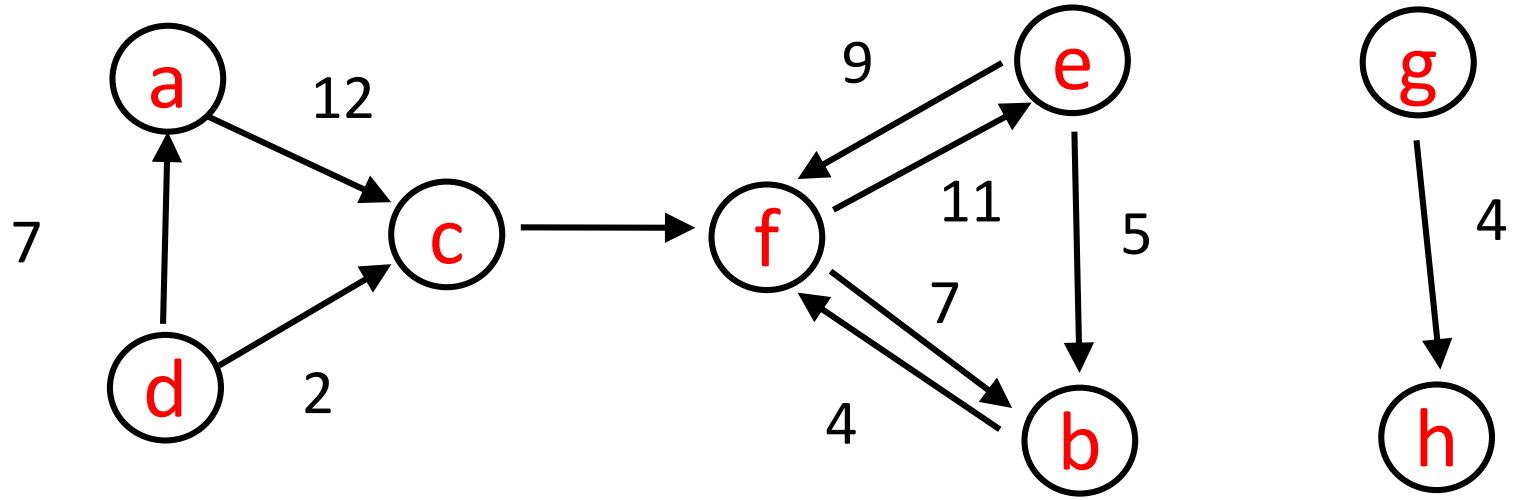


A *path* is a sequence of edges such that end vertex of one edge is the start vertex of the next edge. No vertex may be repeated except first and last.

Examples

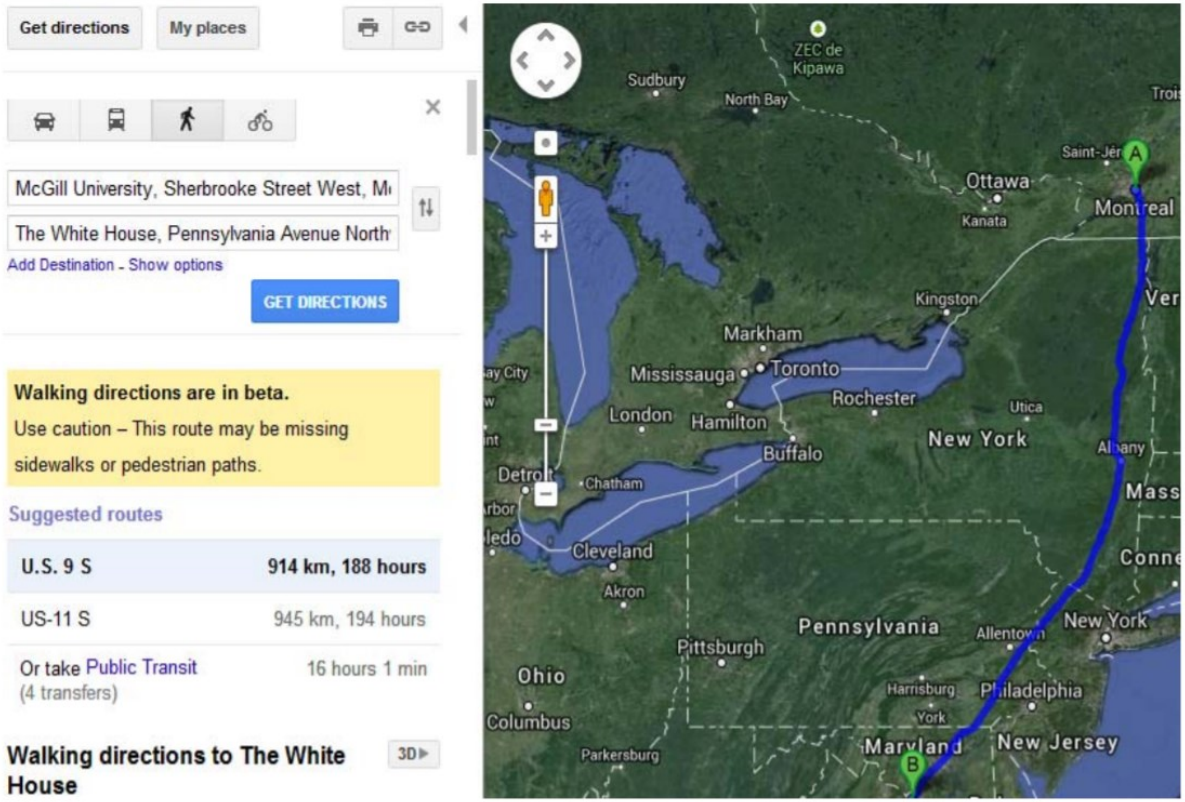
- acfeb
- dac
- febf
- …..

Weighted Graph

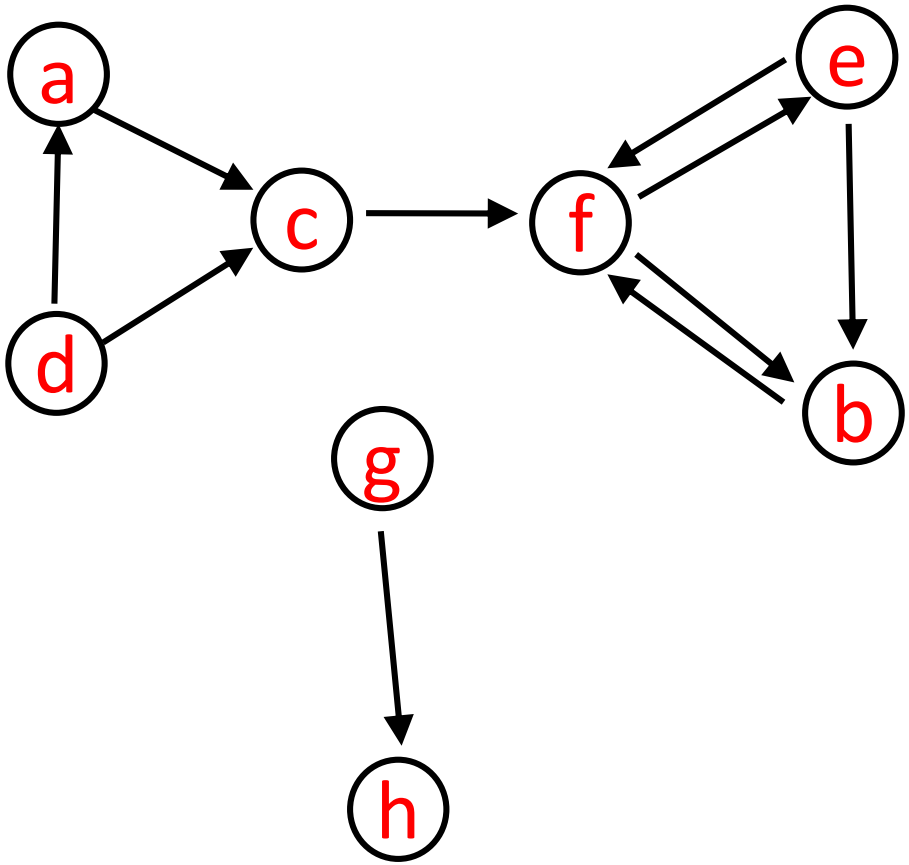


Graph algorithms in COMP 251

Given a graph, what is the shortest (weighted) path between two vertices?



Terminology: cycle

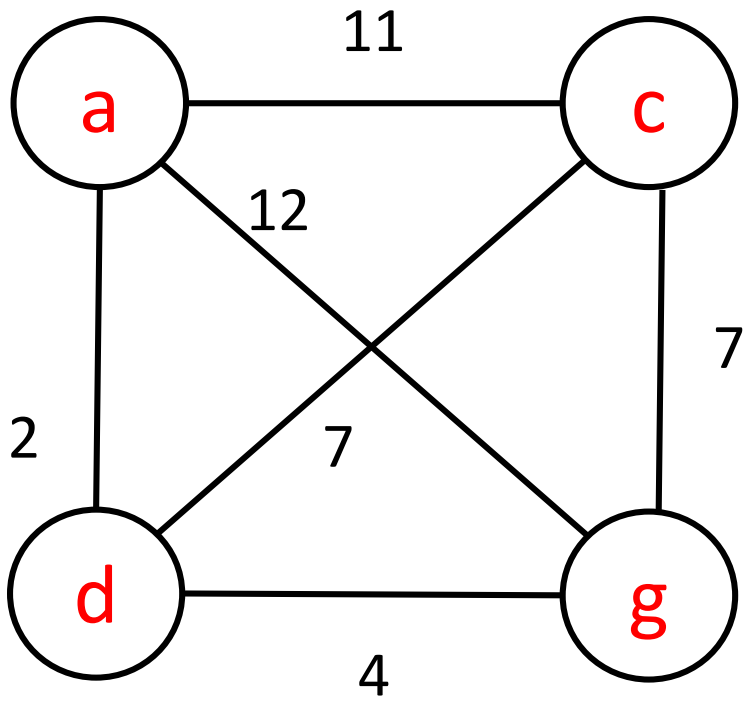


A *cycle* is a path such that the last vertex is the same as the first vertex.

Examples

- febf
- efe
- fbf
- …

“Travelling Salesman” COMP 360
(Hamiltonian circuit)

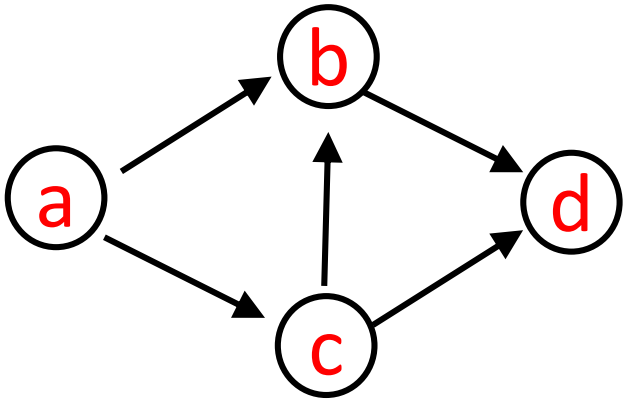


Find the shortest cycle *that visits all vertices once.* (except first & last)

This is an example of a hard problem (called NP complete).

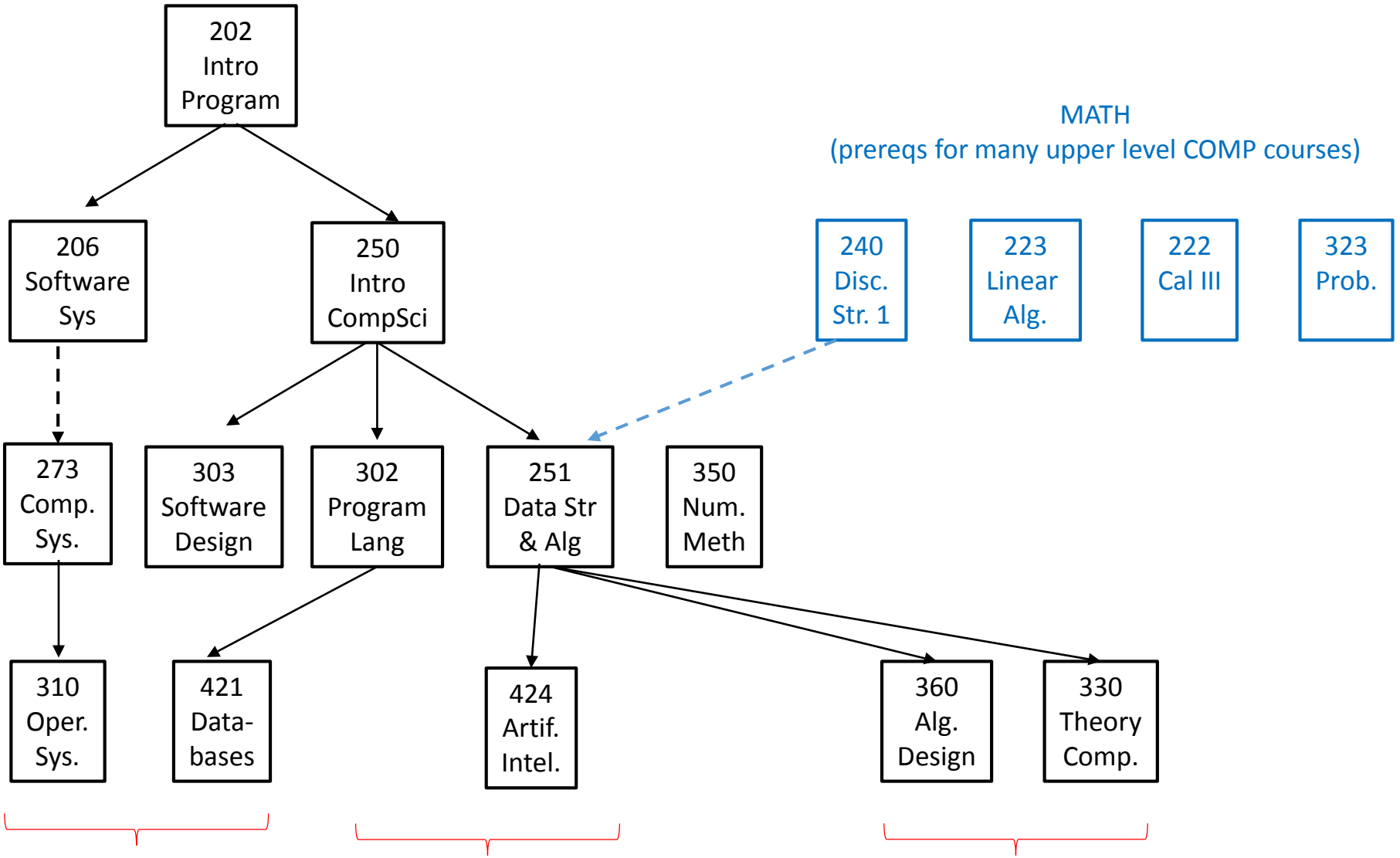
Directed *Acyclic* Graph

no cycles



There are three paths from a to d.

Used to capture dependencies.
e.g. a implies b, or a must happen before b can happen, etc.


202
Intro
Program
206
Software
Sys
250
Intro
CompSci
MATH
(prereqs for many upper level COMP courses)
240
Disc.
Str. 1
223
Linear
Alg.
222
Cal III
323
Prob.
273
Comp.
Sys.
303
Software
Design
302
Program
Lang
251
Data Str
& Alg
350
Num.
Meth
310
Oper.
Sys.
421
Data-
bases
424
Artif.
Intel.
360
Alg.
Design
330
Theory
Comp.
SYSTEMS
(compilers, networks,
distributed sys,
concurrency, web,..)
APPLICATIONS
(graphics, vision,
bioinf, games,
machine learning..)
THEORY
(crypto, optimization, game theory,
logic, correctness, computability..)

Graph ADT

- addVertex(), addEdge()
- removeVertex(), removeEdge()
- getVertex(), getEdge()

- containsVertex(), containsEdge()
- numVertices(), numEdges()
- …

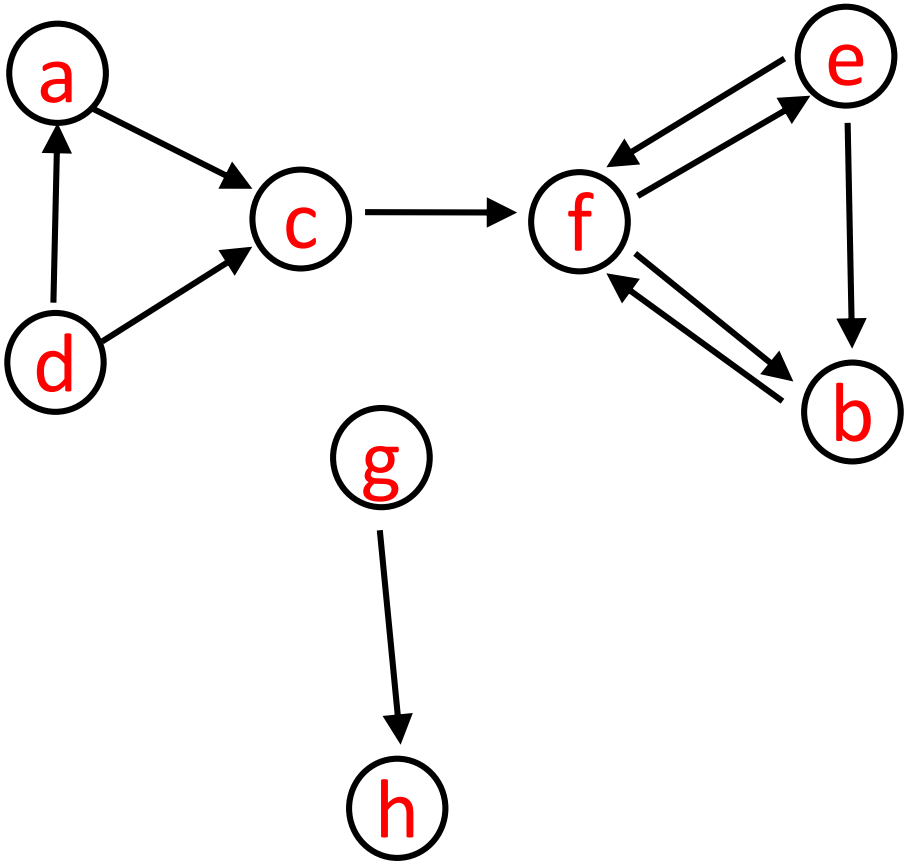
How to implement a Graph class?

- Graphs are a generalization of trees, but a graph does not have a root vertex.

- Outgoing edges from a vertex in a graph are like children of a vertex in a tree. Incoming edges are like parent(s).

1. Adjacency List (for edges)

(generalization of children for graphs)



v	**v.adjList**
a	c
b	f
c	f
d	a, c
e	b, f
f	b, e
g	h
h	

Here each adjacency list is sorted, but that is not always possible (or necessary).

How to implement a Graph class in Java?

```
class Graph<T>  {
     :
    class Vertex<T>                    //   Could have called it GNode
     {
        ArrayList<Vertex>      adjList;          //  end vertex of edge
        T                              element;
     }
    :
}
```

How to implement a Graph class in Java?

```
class Graph<T> {                    //  this would be a weighted graph

    class Vertex<T>  {
        ArrayList<Edge>     adjList;    //    end vertex of an edge  (start is 'this' vertex)
        T                   element;
        boolean             visited;
    }

    class Edge {
      Vertex            endVertex;
      double            weight;
                :
    }
}
```

How to access vertices?

We can have a string name (key) for each vertex.

e.g. YUL for Trudeau airport, LAX for Los Angeles, …

HashMap< String, Vertex<T> > vertexMap;

We could also just have a list of vertices.

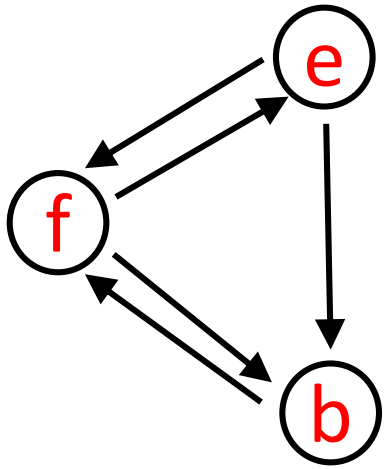
How to access vertices?

We can have a string name (key) for each vertex.

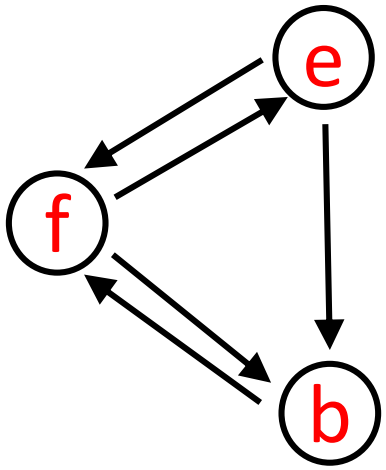
e.g. YUL for Trudeau airport, LAX for Los Angeles, …

```
class  Graph<T> {
        HashMap< String, Vertex<T> >    vertexMap;
           :
       class Vertex<T> { …}
       class  Edge<T>  { …}
}
```

How to implement a Graph class in Java?



How many objects?



Graph

HashMap

Vertex

Vertex

Vertex

ArrayList

ArrayList

ArrayList

Edge

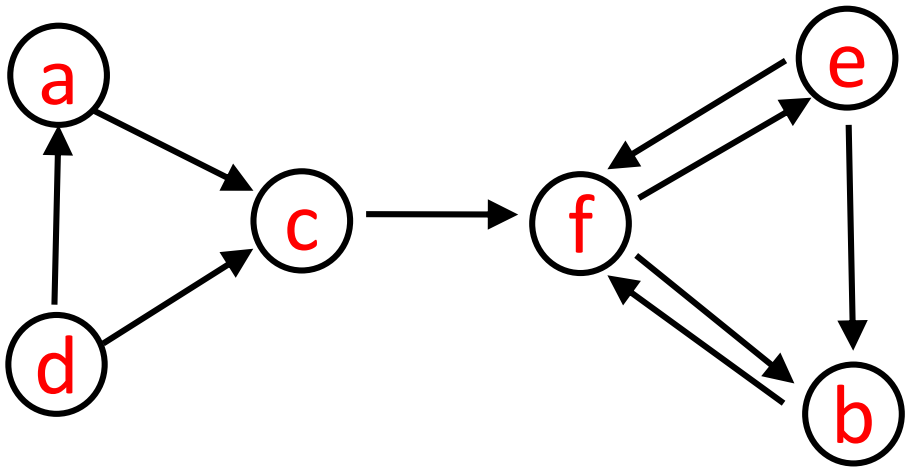
Edge

Edge

Edge

Edge

2. Adjacency Matrix

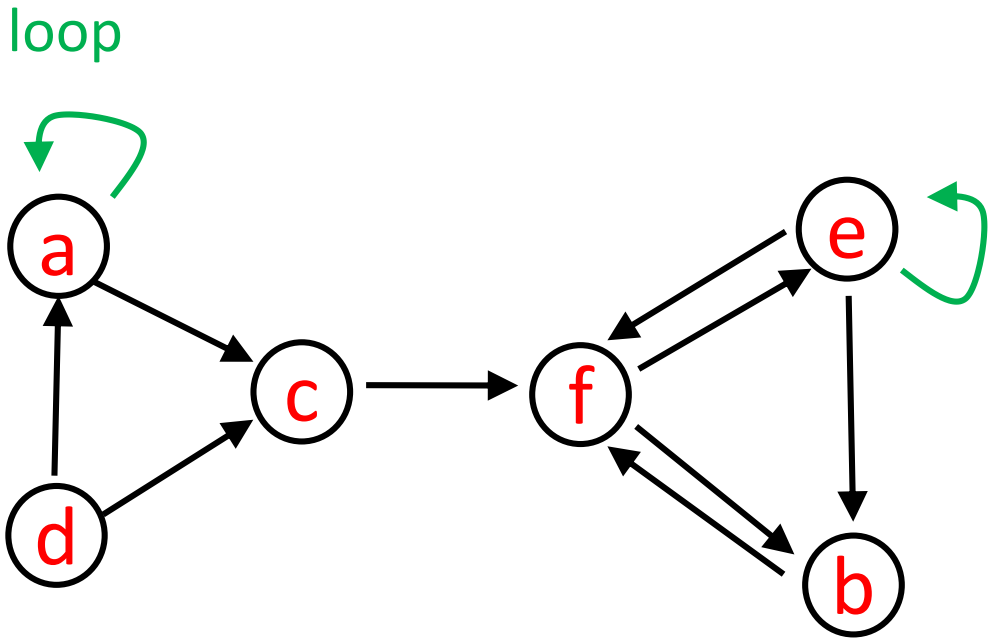


	a	b	c	d	e	f
a	0	0	1	0	0	0
b	0	0	0	0	0	1
c	0	0	0	0	0	1
d	1	0	1	0	0	0
e	0	1	0	0	0	1
f	0	1	0	0	1	0

Assume we have a list of vertex names i.e. a unique mapping from vertex names to 0, 1, …. , n-1 (not a hashmap).

```
boolean   adjMatrix[ 6 ][ 6 ]
```

2. Adjacency Matrix



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

boolean adjMatrix[6][6]

Suppose a graph has n vertices.

We say:

- the graph is *dense* if number of edges is close to n^2.
- the graph is *sparse* if number of edges is close to n.

(These are not formal definitions.)

Exercise

Adjacency list versus an *adjacency matrix* ?
When would you use one versus the other?

v	**v.adjList**
a	c
b	f
c	f
d	a, c
e	b, f
f	b, e
g	h
h	

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

Exercise

Would you use an *adjacency list* or *adjacency matrix* for each of the following?

- The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.

Exercise

Would you use an *adjacency list* or *adjacency matrix* for each of the following?

- The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.
- The graph is dense e.g. 10,000 vertices and 20,000,000 edges, and we want to use as little space as possible.

Exercise

Would you use an *adjacency list* or *adjacency matrix* for each of the following?

- The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.
- The graph is dense e.g. 10,000 vertices and 20,000,000 edges, and we want to use as little space as possible. .
- Answer the query areAdjacent() as quickly as possible, no matter how much space you use.

Exercise

Would you use an *adjacency list* or *adjacency matrix* for each of the following?

- The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.
- The graph is dense e.g. 10,000 vertices and 20,000,000 edges, and we want to use as little space as possible. .
- Answer the query areAdjacent() as quickly as possible, no matter how much space you use.
- Perform operation insertVertex(v).

Exercise

Would you use an *adjacency list* or *adjacency matrix* for each of the following?

- The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.
- The graph is dense e.g. 10,000 vertices and 20,000,000 edges, and we want to use as little space as possible. .
- Answer the query areAdjacent() as quickly as possible, no matter how much space you use.
- Perform operation insertVertex(v).
- Perform operation removeVertex(v).

Next lecture

- Recursive graph traversal
  - depth first
- Non-recursive graph traversal
  - depth first
  - breadth first