Graph Terminology

Review 03.02

By Marina Barsky

[What is a graph?]

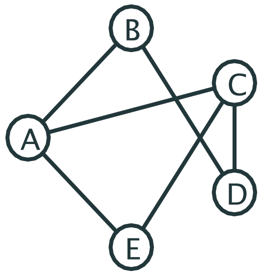
A graph *G* = (*V, E*) is an Abstract Data Type that consists of 2 sets:

- Set of objects (*vertices*, nodes)

  V = {A, B, C, D, E}

- Relation on set of objects (*edges*)

  E = {(A,B), (A,C), (A,E), (B.D), (C,D), (C,E)}



Running time of Graph algorithms uses **two** input sizes:

- n = |V|
- m = |E|

[Vertices and edges]

- Edge *e* connects vertices *u* and *v*
- Vertices *u* and *v* are end points of edge *e*
- Vertex *u* and edge *e* are incident
- Two edges are also called incident, if they are incident to the same vertex
- *Vertices u* and *v* are adjacent
- *Vertices u* and *v* are neighbors

- This is a dictionary of undirected graph

[The degree of a vertex]

- The degree of a vertex is the number of its incident edges.
  I.e., the degree of a vertex is the number of its neighbors

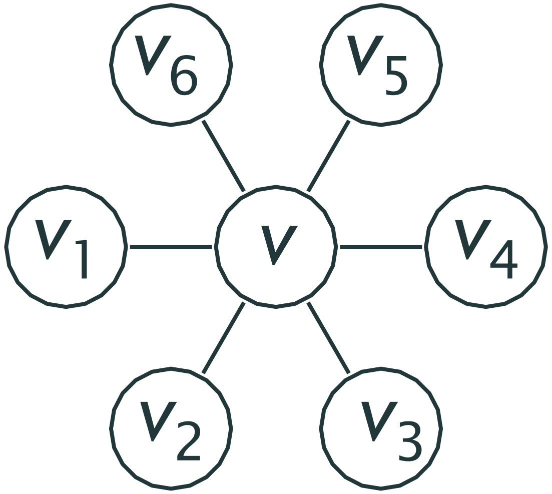
- The degree of a vertex v is denoted by deg(v)

- The degree of a graph is sum of degree of its vertices.
  The degree of undirected graph with m edges is $2m$

Example

The degree of v is 6: deg(v) = 6

The degree of v_6 is 1: deg(v_6) = 1



The degree of *this graph*: deg(G) = 2m = 12

[Directed graphs]

Nodes: {A,B,C,D}

Edges (ordered pairs):
{(A,C),(D,A),(B,D),(C,B)}

Edges (ordered pairs):
{(C,A),(D,A),(B,D),(C,B)}

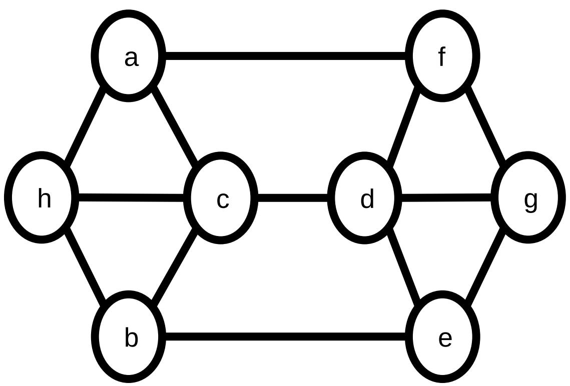
These two graphs are different

[Subgraphs]

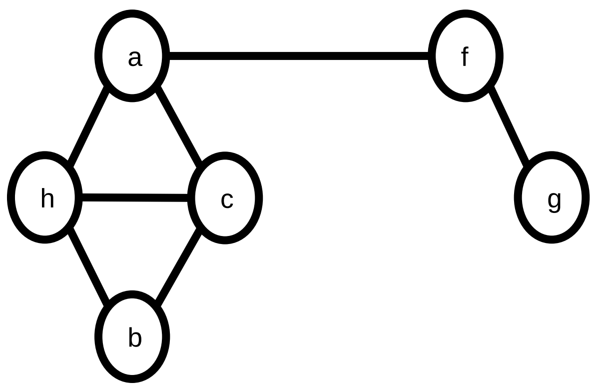
A ***subgraph*** of a graph is obtained by deleting any subset of vertices and edges.

- If a vertex is deleted, then all of its incident edges are also deleted.

A subgraph is *spanning* if it includes all of the vertices (only some edges are deleted).

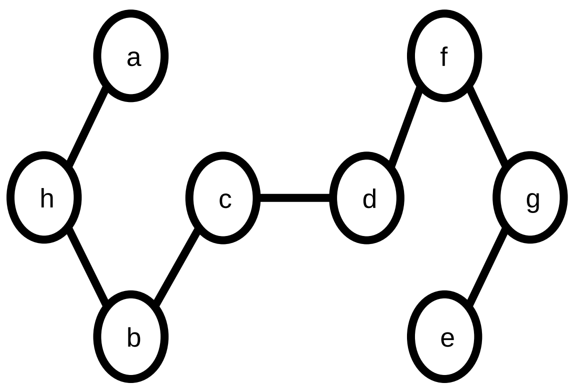


A graph.



An induced subgraph
G[{a, b, c, h, f, g}].

A non-spanning subgraph.



A spanning subgraph.

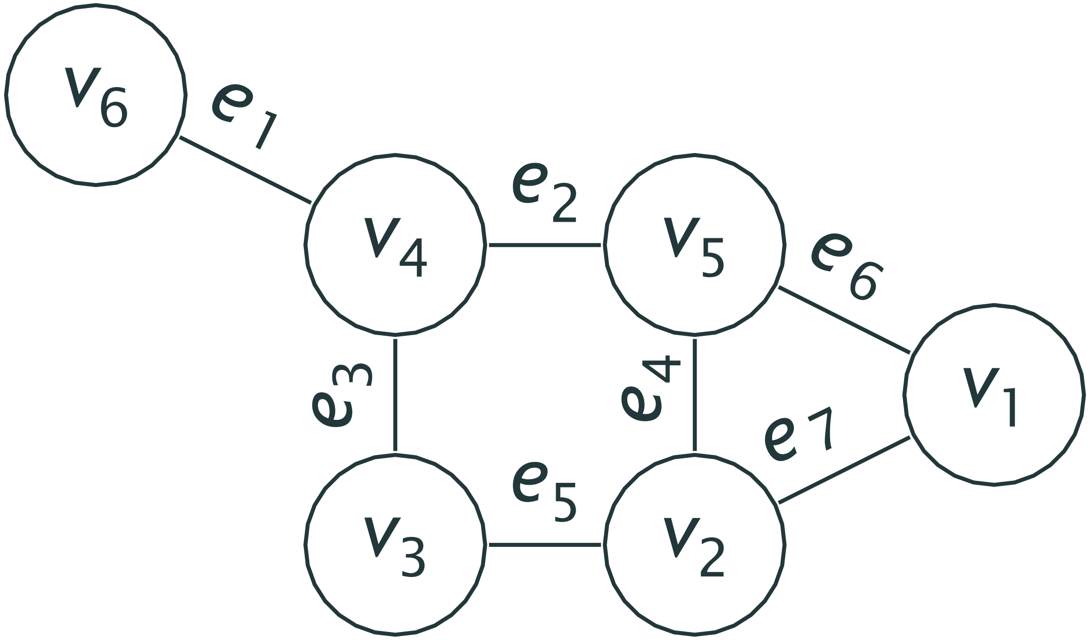
An ***induced subgraph*** is obtained by deleting any subset of vertices.
It is denoted by G[U] where U is the set of vertices that are not deleted.

[Walks and Paths]

- A walk in a graph is a sequence of incident edges
- The length of a walk is the number of edges in it
- A path is a walk where all edges are distinct
- A simple path is a walk where all vertices are distinct

Example 1

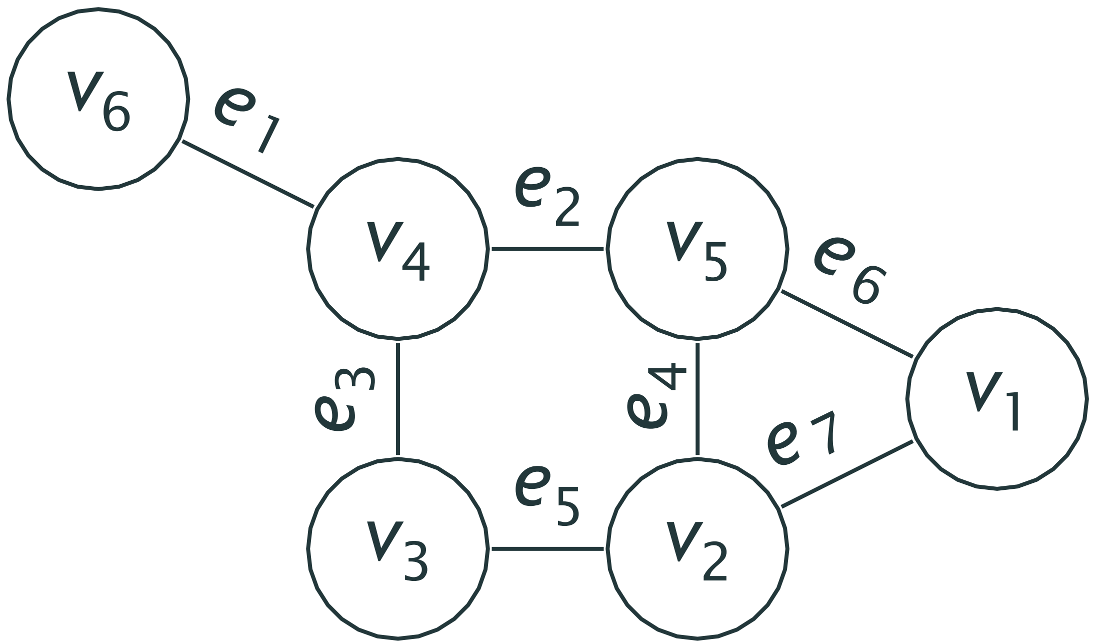
A walk of length 6: (e_1, e_2, e_4, e_5, e_3, e_1)



Example 1

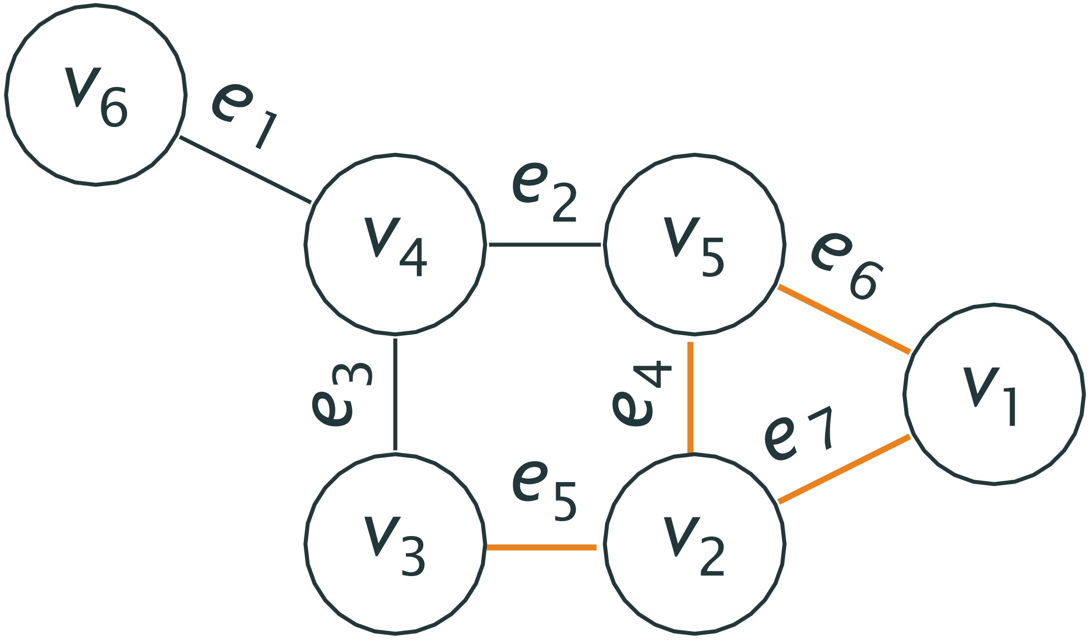
A walk of length 6: (e_1, e_2, e_4, e_5, e_3, e_1)

Not a path: uses e_1 twice



Example 2

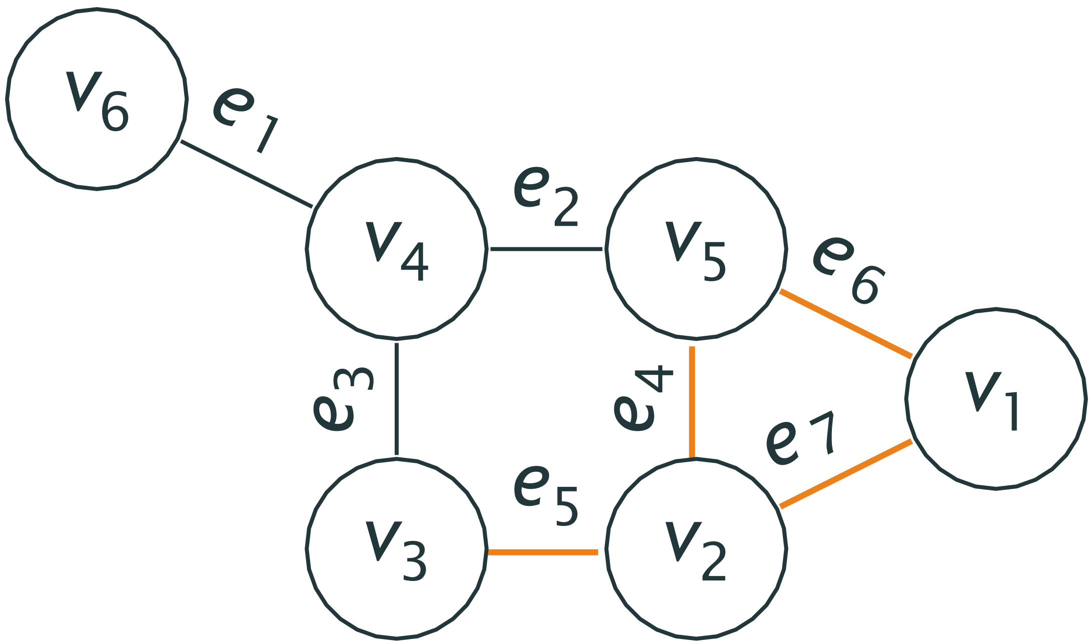
A path of length 4: (e_7, e_6, e_4, e_5)



Example 2

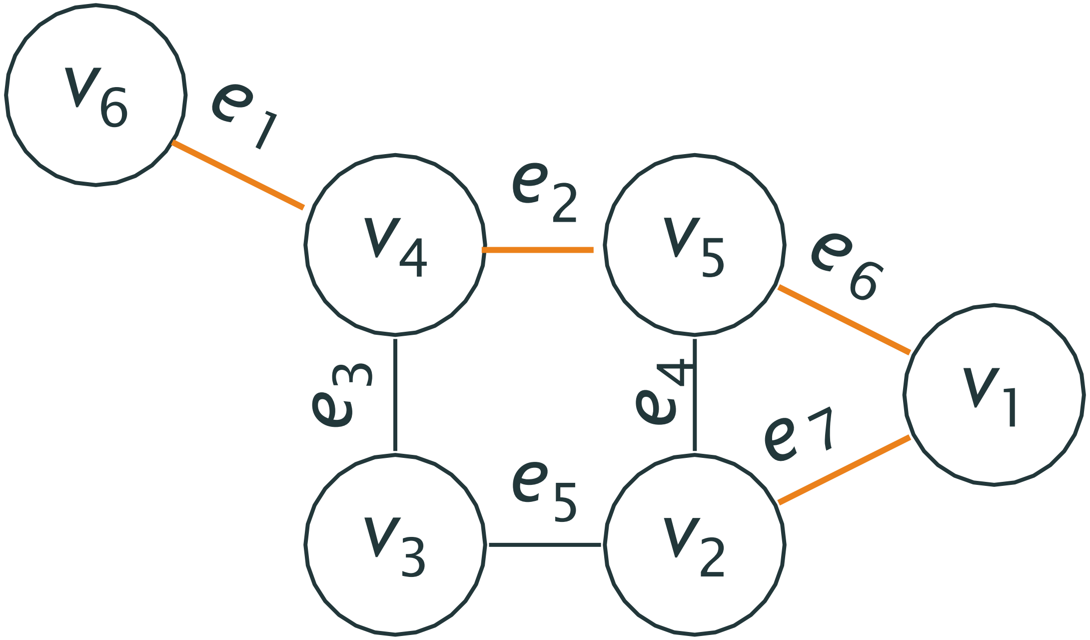
A path of length 4: (e_7, e_6, e_4, e_5)

Not a simple path: visits v_2 twice



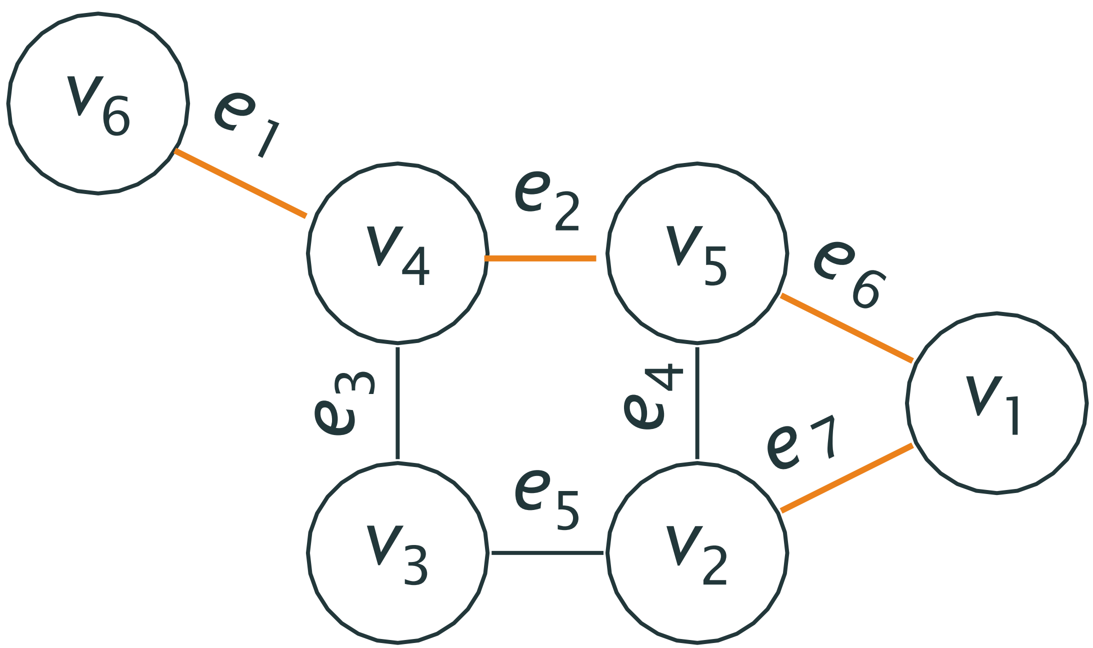
Example 3

A simple path of length 4: (e_7, e_6, e_2, e_1)



It is sometimes more convenient to specify a path (walk) by a list of vertices rather than edges

A path of length 4: $(v_2, v_1, v_5, v_4, v_6)$

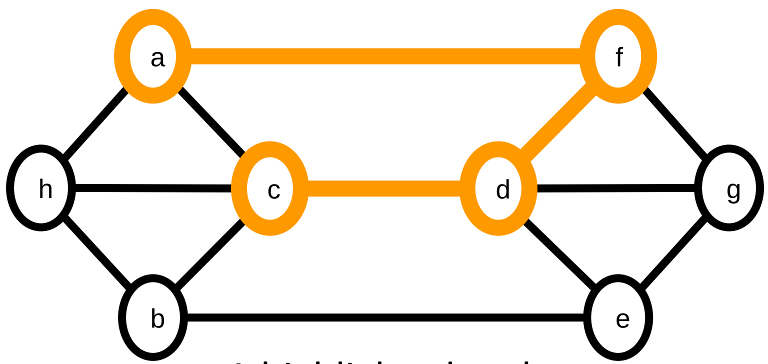


The length of the path

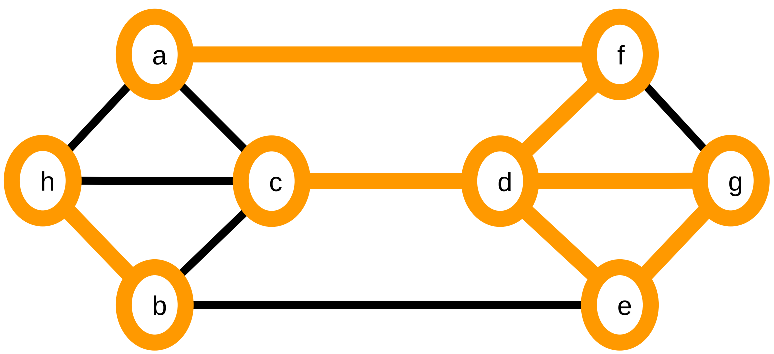
In general, a *path* of length k is a sequence of k incident edges (and k+1 vertices):

$$v_1,\ (v_1, v_2),\ v_2,\ (v_2, v_3),\ v_3,\ \ldots,\ v_k,\ (v_k, v_{k+1}),\ v_{k+1} \text{ where } v_i \neq v_j \text{ if } i \neq j.$$

In other words, there are k+1 vertices and k edges, and each edge connects adjacent vertices on the path.



A highlighted path
a, (a,f), f, (f,d), d, (d,c), c



This is not a path since it is disconnected and also d appears multiple times.

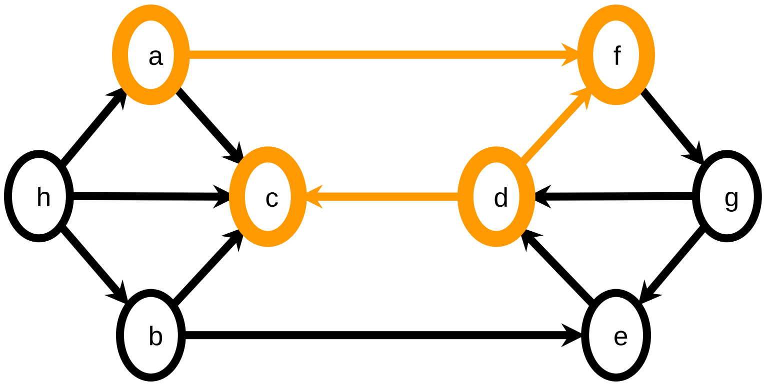
The ***length of a path*** is the number of traversed edges.

A path from u to v is a ***shortest path*** if there is no shorter path from u to v. For example, there are two shortest paths from f to e above.

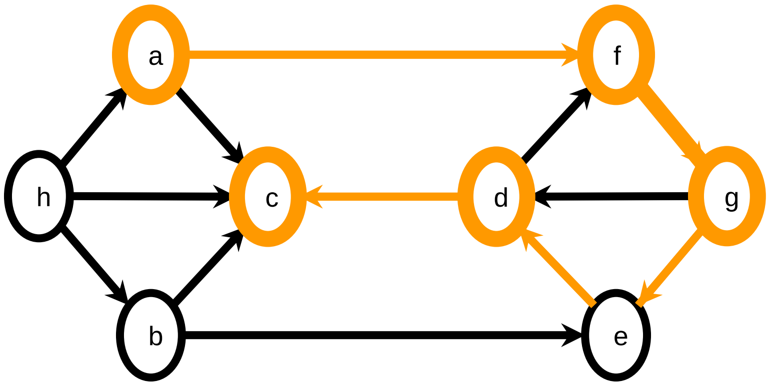
Directed Paths

In a directed graph each edge is oriented in one of two ways with respect to a path:

- The edge is *forward* if it has the form v_i, (v_i, v_{i+1}), v_{i+1}.
- The edge is *backward* if it has the form v_i, (v_{i+1}, v_i), v_{i+1}.



A highlighted path
a, (a,f), f, (f,d), d, (d,c), c
where (f,d) is the only backwards edge.



A directed path from a to c.

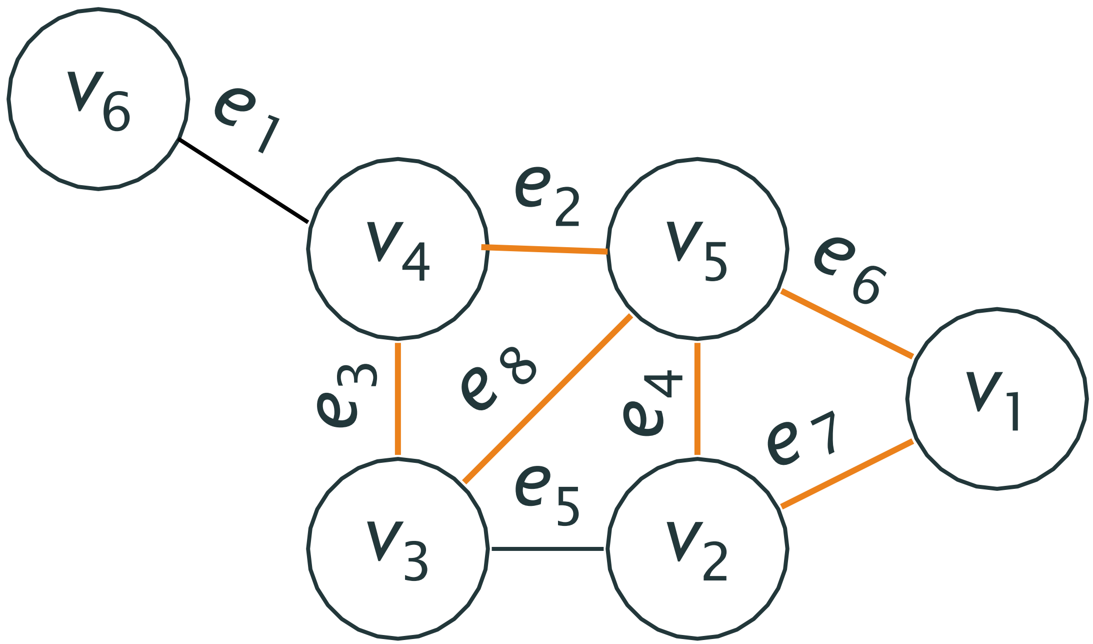
A path is a *directed path* if every edge is a forward edge.

Cycles

- A cycle (sometimes called a circuit) in a graph is a path where the first vertex is the same as the last one
- All the edges in a cycle are distinct
- A simple cycle is a cycle where all vertices except for the first=last are distinct

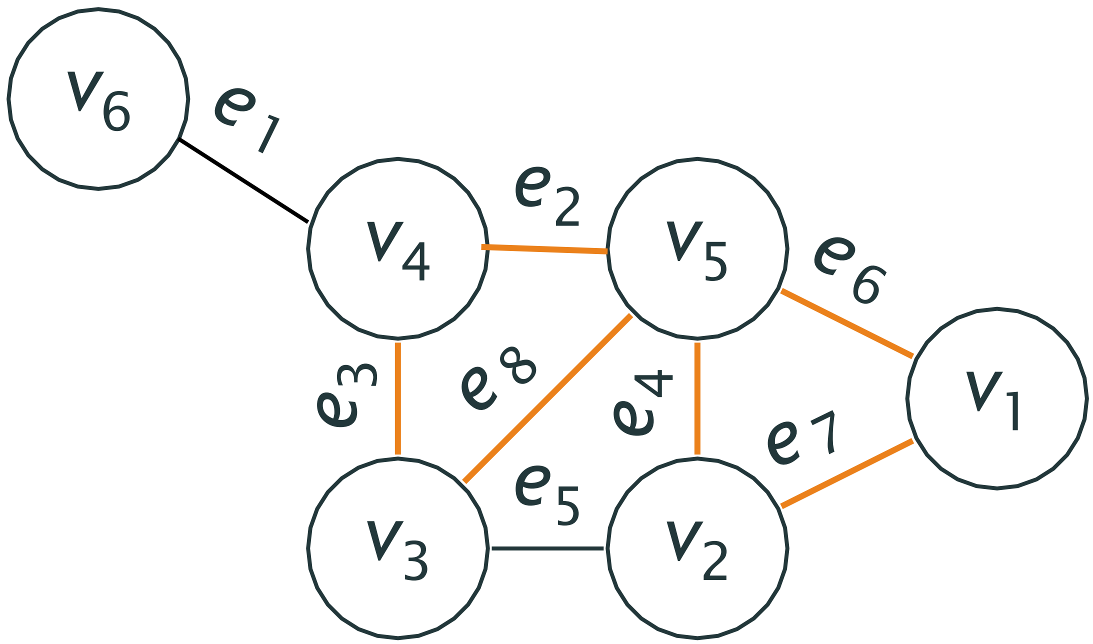
Example 1

A cycle of length 6: $(e_2, e_3, e_8, e_4, e_7, e_6)$



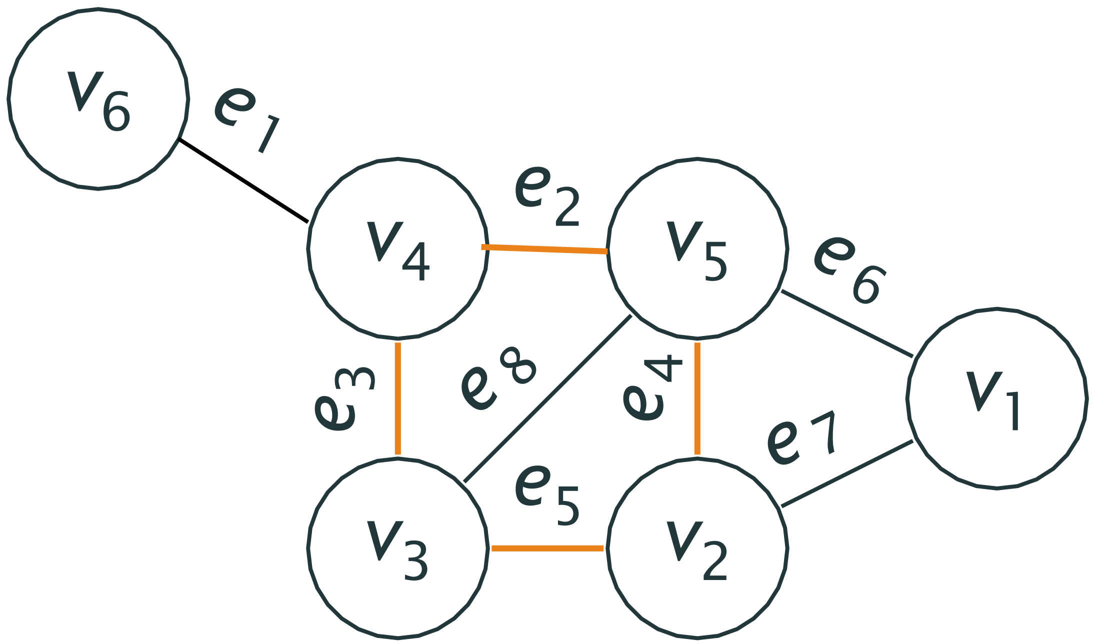
Example 1

A cycle of length 6: (e_2, e_3, e_8, e_4, e_7, e_6)
Not a simple cycle: visits v_5 three times



Example 2

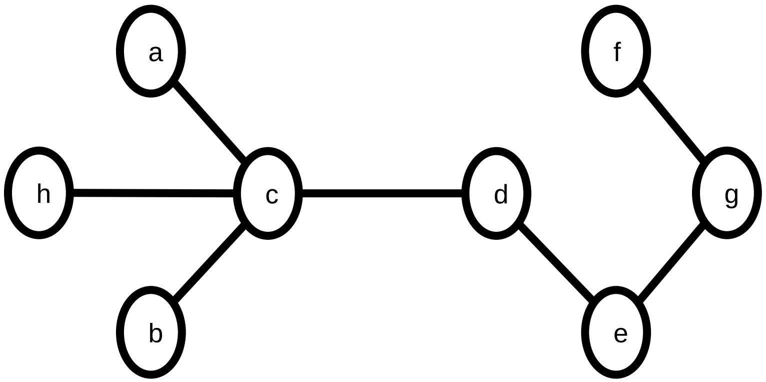
A simple cycle of length 4: (e_5, e_4, e_2, e_3)



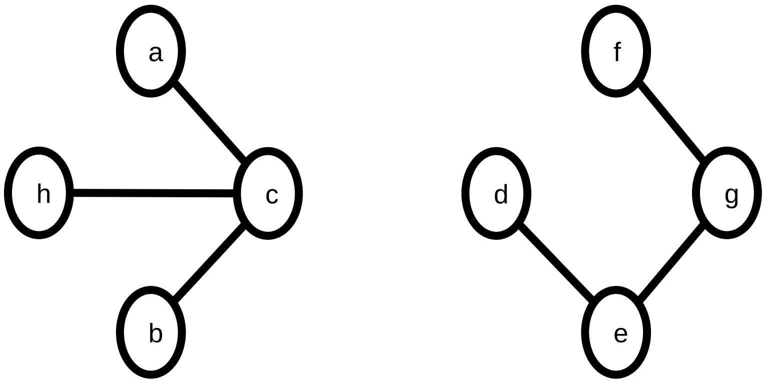
Trees and Forests

A ***tree*** is a connected **acyclic** graph. That is, each node is connected to some other node, and there are no cycles.
A *forest* is an acyclic graph (i.e. its connected components are trees.)

A *leaf* is a vertex of degree one, and the other vertices are *internal nodes*.



A tree with four leaves and four internal vertices.



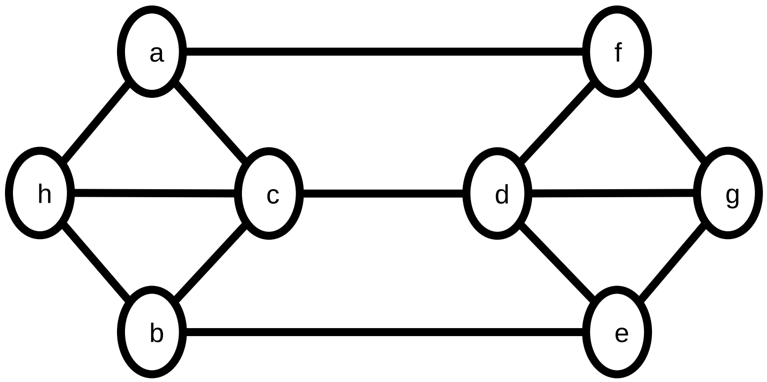
A forest with two component trees.

Lemmas:

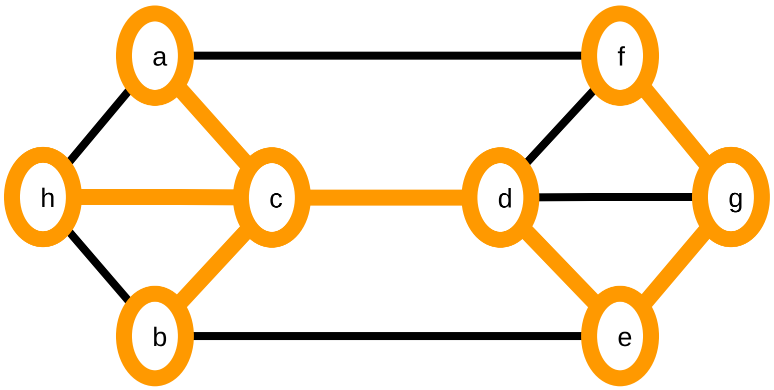
- A tree on n vertices has n-1 edges.
- A forest with n vertices and c components has n-c edges.
- There is a unique path between any two vertices within a tree.

Spanning Trees

A ***spanning tree*** is a subgraph that is spanning and is a tree.



A connected graph.

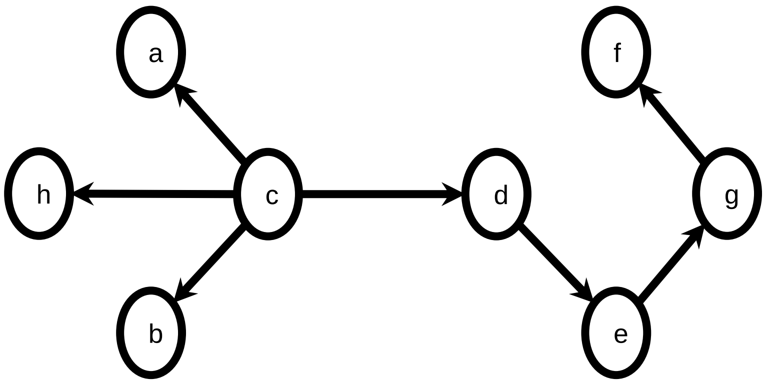


A spanning tree of the graph.

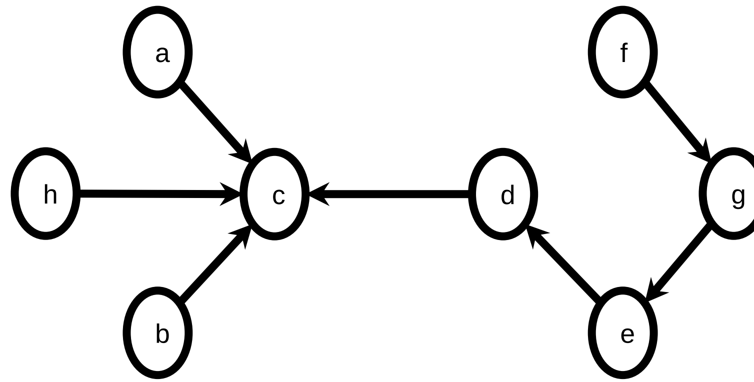
Lemma:
A graph is connected if and only if it has a spanning tree.

Rooted Trees

A ***rooted tree*** has a specified *root vertex*.
Every edge joins a *parent* and a *child* vertex, where the parent is closer to the root.



A rooted tree from vertex c.
Edges are directed outward from the root (i.e. parent to child).



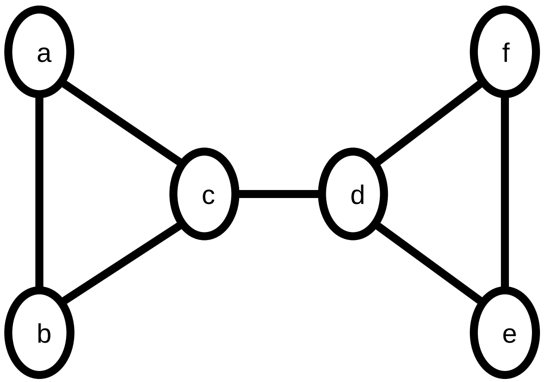
A rooted tree from vertex c.
Edges are directed inward to the root (i.e. child to parent).

Sometimes we direct edges *outward* from the root or *inward* to the root.
When rooted trees are drawn the root is typically placed at the top and every parent is placed above its children.

Data Structures

Representing Graph as Edge Set (**Edge List**)

The most straightforward way of storing graphs is to create a set of all graph vertices, and a set of all edges in form of tuples:



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

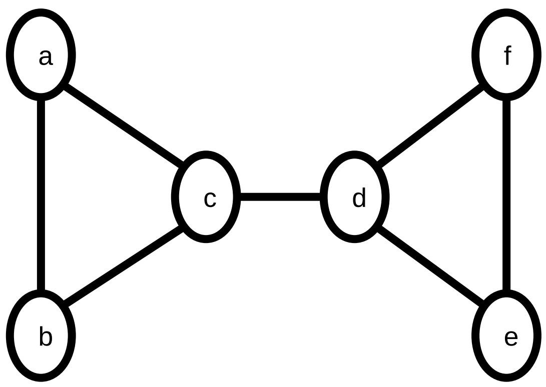
- Edge lists are simple, but if we want to find whether the graph contains a particular edge, we have to search through the edge list.
- If the edges appear in the edge list in no particular order, that's a linear search through *m* edges.

Question: How would you organize an edge list to make searching for a particular edge take O(log m) time?

[Adjacency Lists and Adjacency Matrices]

Graphs are commonly stored as ***adjacency lists*** or ***adjacency matrices*****.**

- In undirected graphs each edge is stored twice.
- Non-simple graphs use adjacency counts instead of 0/1 in the adjacency matrix.
- Non-simple graphs repeat vertices or use edge numbers in the adjacency list.



Graph

a	b, c
b	a, c
c	a, b, d
d	c, e, f
e	d, f
f	d, e

Adjacency List

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

Adjacency Matrix

Efficient Representation

The data structure used to store a graph affects the efficiency of algorithms running on it.

Task	**Winner**
To test if (x,y) is in graph?	
Find a degree of a vertex	
Store a sparse graph: m = O(n)	
Store a dense graph: m = O(n^2)	
Insert/delete an edge	
Traverse the graph	
Most problems	

n = \|V\|, m = \|E\|

Efficient Representation

The data structure used to store a graph affects the efficiency of algorithms running on it.

Task	**Winner**
To test if (x,y) is in graph?	Adj. matrix O(1)
Find a degree of a vertex	Adj. list O(d) vs. O(n)
Store a sparse graph: m = O(n)	Adj. list (n + m) vs. n^2
Store a dense graph: m = O(n^2)	Adj. matrix (save on links)
Insert/delete an edge	Adj. matrix O(1) vs. O(d)
Traverse the graph	Adj. list (n + m) vs. n^2
Most problems	Adj. list