Data Structures for implementing Graph ADT

Lecture 26
by Marina Barsky

Abstract data Type: *Graph*

Specification

Graph is an Abstract Data Type which models relationships between entities.

The entities are modeled as vertices, and the connections as edges.

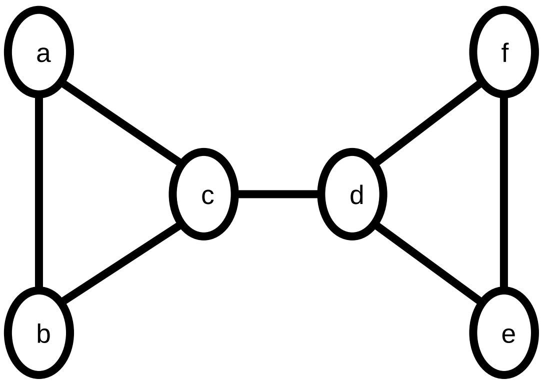
Abstract data Type: *Graph*

Supported operations

- **Vertices()** – returns the ***set*** of all vertices
- **Edges()** – returns all the edges (not necessarily a set)
- **AddEdge (v_1, v_2, [*cost*])** – adds a new edge between v_1 and v_2 , optionally with *cost*
- **AddVertex(*v*)** – adds a new vertex
- **RemoveEdge(*e*)** – removes edge e
- **RemoveVertex(*v*)** – removes vertex v with all its incident edges
- **AreAdjacent(v_1, v_2)** – returns *True* if vertices $v_1$and v_2 are adjacent
- **GetIncidentEdges(*v*)** – returns all the incident edges of vertex *v*
- **GetNeighbors(*v*)** – returns all adjacent vertices of *v*

Representing Graph as Edge Set (Edge List)

The most straightforward mathematical way of storing graphs is to create a set of all graph vertices, and a list 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)}

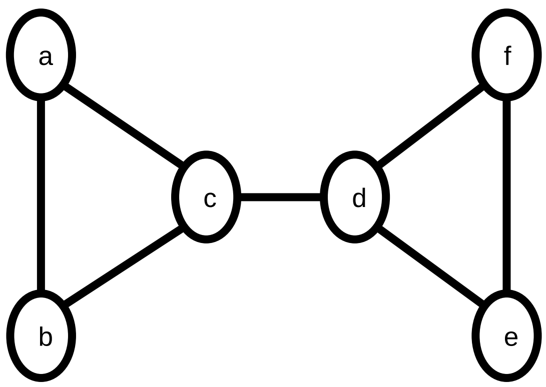
- Edge lists are simple, but if we want to find whether the graph contains a particular edge, we have to search through the entire edge list.
- If the edges appear in the edge list in no particular order, that's a linear search through *m* edges.

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

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 (with more than one edge between the same vertices) use adjacency *counts* instead of 0/1 in the adjacency matrix.
- Non-simple graphs repeat vertices or use edge counts 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

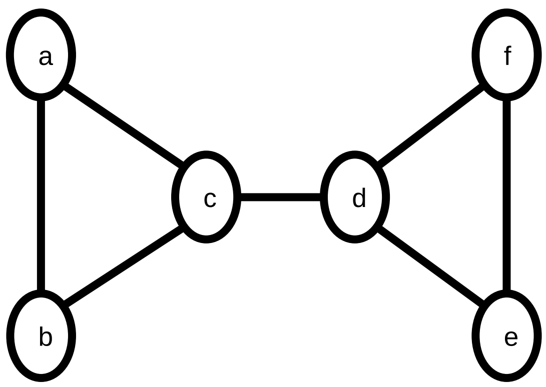
Adjacency List

	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

Adjacency Lists vs Adjacency Matrices: space

- For a sparse graph: where $m = O(n)$ – use ***adjacency lists*** (linear vs. quadratic storage)
- For a dense graph: where $m = O(n^2)$ – use ***adjacency matrices*** (save on links)



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

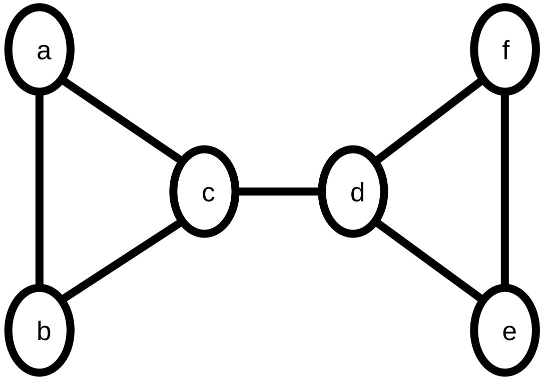
Adjacency Matrix

Efficiency of operations

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

Operation	**Winner**
areAdjacent(x,y)	
degree(v)	
addEdge ($e_{x,y}$)	
removeEdge ($e_{x,y}$)	

n = |V|, m = |E|



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

Which data structure is most efficient for the following 3 operations?

	Operation
1	areAdjacent(x,y)
2	degree(x)
3	Add/remove Edge ($e_{x,y}$)

A. (1) matrix (2) matrix (3) matrix
B. (1) matrix (2) list (3) list
C. (1) list (2) list (3) list
D. (1) matrix (2) list (3) matrix
E. None of the above

Efficiency of operations

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

Operation	**Winner**
areAdjacent(x,y)	**Adj. matrix** O(1) vs. O(degree(x))
degree(x)	**Adj. list** O(degree(x)) vs. O(n)
addEdge ($e_{x,y}$)	**Adj. matrix** O(1) vs. O(degree(x))
removeEdge ($e_{x,y}$)	**Adj. matrix** O(1) vs. O(degree(x))

$n = |V|$, $m = |E|$

Most graph implementations use **adjacency lists** because most graphs are large and sparse → quadratic storage space is infeasible