CSC 225

Algorithms and Data Structures I

Rich Little
rlittle@uvic.ca
ECS 516

The Graph ADT

- A graph is a positional container of elements that are stored at the graph's vertices and/or edges
  - (for now just at the vertices)
- The *positions* in the graph are its vertices (and edges if weighted)
- We can store elements in a graph at either its edges or its vertices or both

General Graph Methods

- **numVertices():** Return the number of vertices in *G* = n
- **numEdges():** Return the number of edges in *G* = m
- **vertices():** Return an iterator of the vertices of *G*
- **edges():** Return an iterator of the edges of *G*
- **aVertex():** Return an arbitrary vertex of *G*

Graph Methods with Vertex and Edge Positions as Arguments

- **degree(*v*)*:*** Return the degree of *v*
- **adjacentVertices(*v*):** Return an iterator of the vertices adjacent to *v*
- **incidentEdges(*v*):** Return an iterator of the edges incident upon *v*
- **endVertices(*e*):** Return an array of size 2 storing the end vertices of *e*
- **opposite(*v,e*)*:*** Return the endpoint of edge *e* distinct from *v*
- **areAdjacent (*v,w*):** Return whether vertices *v* and *w* are adjacent

Graph Methods for Directed Edges

- **directedEdges():** Return an iterator of all directed edges
- **undirectedEdges():** Return an iterator of all undirected edges
- **destination(*e*):** Return the destination of the directed edge *e*
- **origin(*e*):** Return the origin of the directed edge *e*
- **isDirected(*e*):** Return true if and only if the edge *e* is directed

Graph Methods for Directed Edges

- **inDegree(*v*):** Return the in-degree of *v*
- **outDegree(*v*):** Return the out-degree of *v*
- **inIncidentEdges(*v*):** Return an iterator of all the incoming edges to *v*
- **outIncidentEdges(*v*):** Return an iterator of all the outgoing edges from *v*
- **inAdjacentVertices(*v*):** Return an iterator of all the vertices adjacent to *v* along incoming edges to *v*
- **outAdjacentVertices(*v*):** Return an iterator of all the vertices adjacent to *v* along outgoing edges from *v*

Graph Methods for Updating

- **insertEdge(*v*,*w*,*o*):** Insert and return an undirected edge between vertices *v* and *w,* storing the object *o* at this position.
- **insertDirectedEdge(*v*,*w*,*o*):** Insert and return a directed edge fro vertex *v* to vertex *w,* storing the object *o* at this position.
- **insertVertex(*o*):** Insert and return a new (isolated) vertex storing the object *o* at this position.

Graph Methods for Updating

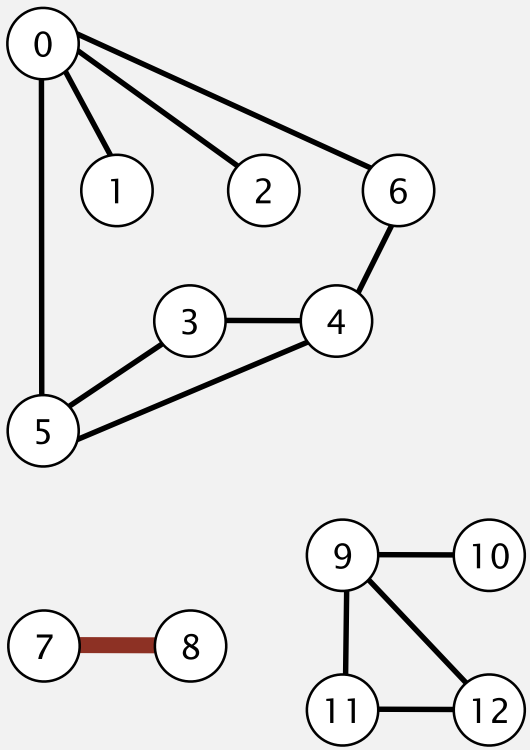
- **removeVertex(*v*):** Remove vertex *v* and all its incident edges
- **removeEdge(*e*):** Remove edge *e*
- **makeUndirected(*e*):** Make edge *e* undirected
- **reverseDirection(*e*):** Reverse direction of directed edge *e*
- **setDirectionFrom(*e*,*v*):** Make edge *e* directed away from vertex *v*
- **setDirectionTo(*e*,*v*):** Make edge *e* directed into vertex *v*

Graph Representations

- Node centric
  - Edge-list structure with vertex and edge objects
  - Adjacency list
  - Labeled adjacency list
  - Adjacency matrix (i.e., 0 or 1 entries)
  - Labeled adjacency matrix (i.e., edge label entries)

Graph representation: set of edges

Maintain a list of the edges (linked list or array).



0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Q. How long to iterate over vertices adjacent to v ?

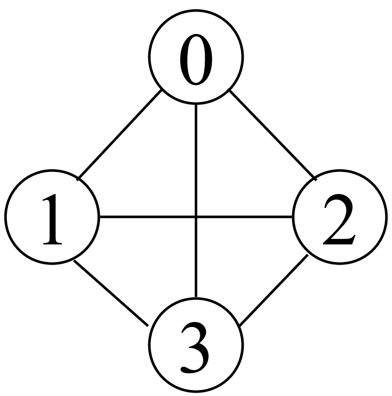
Edge List Structure

- vertex and edge containers are typically lists, vectors or dictionaries
  - most commonly a list
- main feature: direct access from edges to vertices they are incident upon
- simple algorithms for edge based methods (endVertices, origin, destination, etc.)
- Problem: vertex based methods are time dependent on number of edges
  - Iterators for incident edges or adjacent vertices of a vertex run in $O(m)$ time
  - Also, areAdjacent(u,v) and removeVertex(v) take $O(m)$ time

Adjacency-Matrix Structure

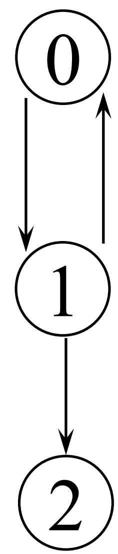
- Vertices are numbered $0,1,..,n-1$
- (i,j) denotes the edge between vertex with number i and vertex with number j
- The Graph is represented by an $(n \times n)$-array A such that $A[i,j]$ stores a 1 if (i,j) exists, and 0 otherwise
- If (i,j) is undirected, store reference in $A[i,j]$ and $A[j,i]$
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Adjacency Matrices



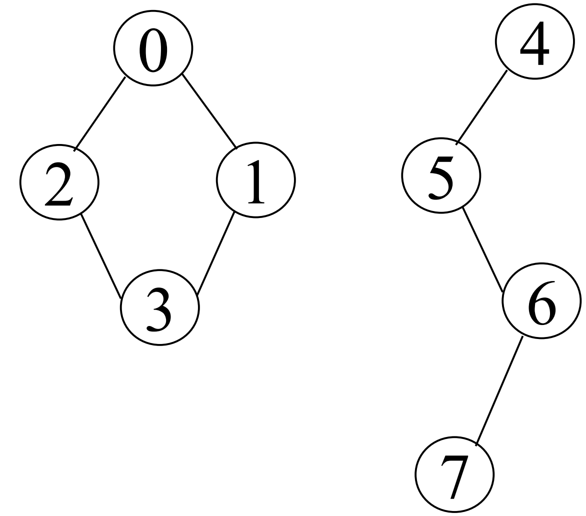
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G1



$$\begin{array}{cc} & \text{to} \\ \text{from} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

G2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G3

Useful Computations on Adjacency Matrices

- The outdegree of vertex *k* is the sum of the adjacency matrix elements in row *k*
- The indegree of vertex *k* is the sum of the adjacency matrix elements in column *k*
- The degree of a vertex *k* is the sum of the adjacency matrix elements in row or column *k*

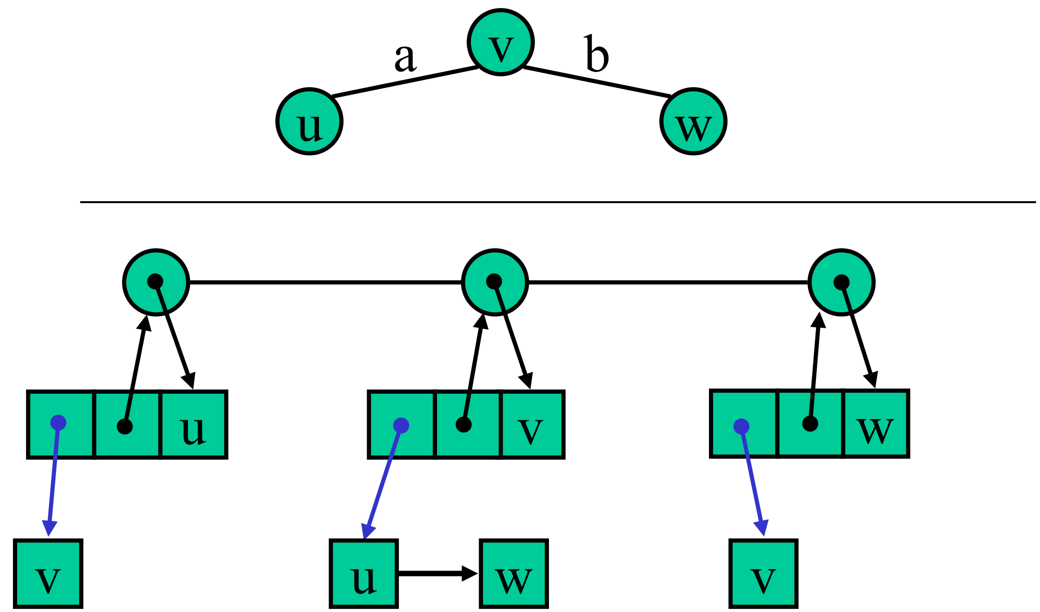
Adjacency Matrix Structure

- Method areAdjacent(u,v) now runs in $O(1)$ time
- Space is now $O(n^2)$
- Also, does slow down other methods
  - incidentEdges(v) and adjacentVertices(v) now require examining an entire column of the matrix, thus $O(n)$ time

    vs. adj. list
- Generally, stick to the adjacency list structure
  - Unless a large number of edges

Adjacency List Structure

- In its simplest form an adjacency list only needs to connect each vertex to its adjacent vertices.
- For example, if its not a weighted graph we do not need explicit structures for the edges.

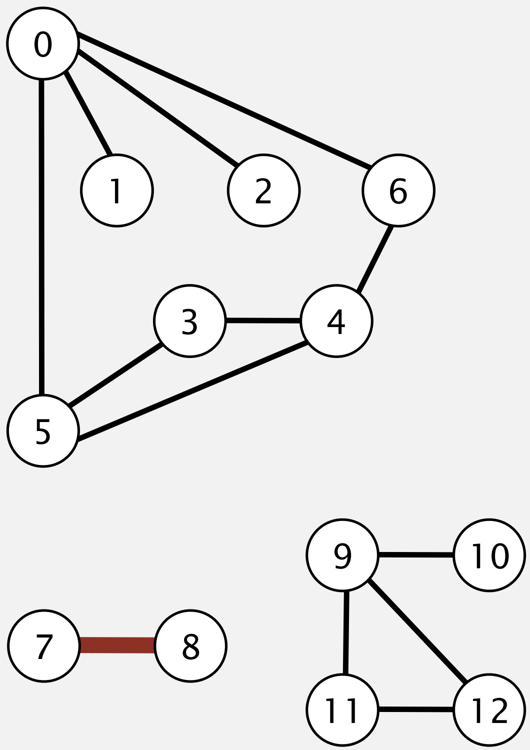


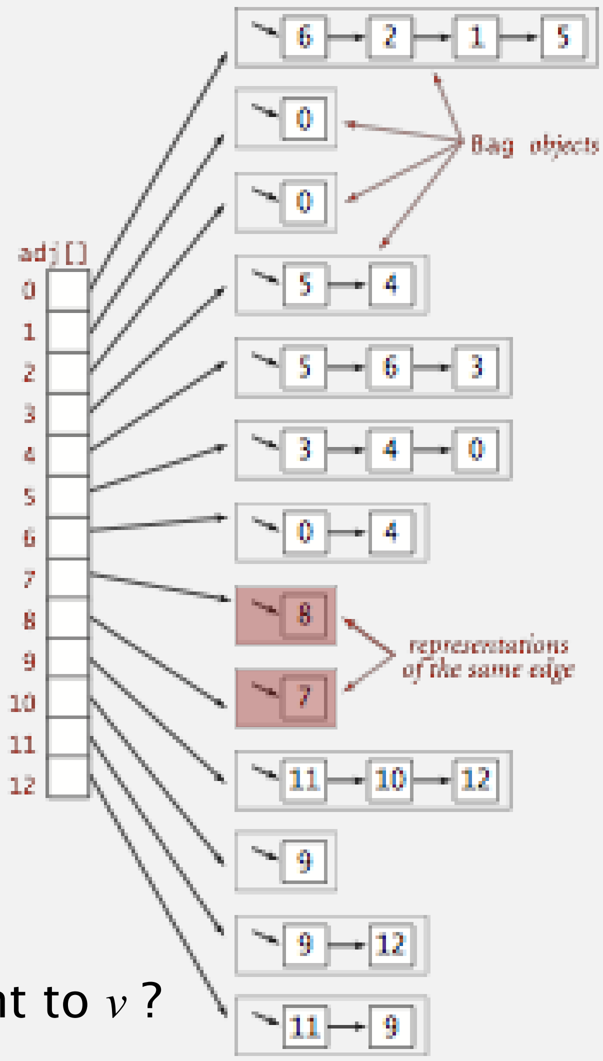
Adjacency List Structure

- Incidence container traditionally realized by a list
- If a digraph, the incidence container is partitioned into in edges and out edges
- Provides access from both vertices to edges and edges to vertices
- Speeds up a number of methods
  - Iterators of incident edges or adjacent vertices for a vertex now run in $O(\deg(v))$ time $\in O(n)$
  - areAdjacent(u,v) runs in $O(\min\{\deg(u),\deg(v)\})$
  - removeVertex(v) is also deg(v) (calls incidentEdges(v))

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.





Q. How long to iterate over vertices adjacent to v ?

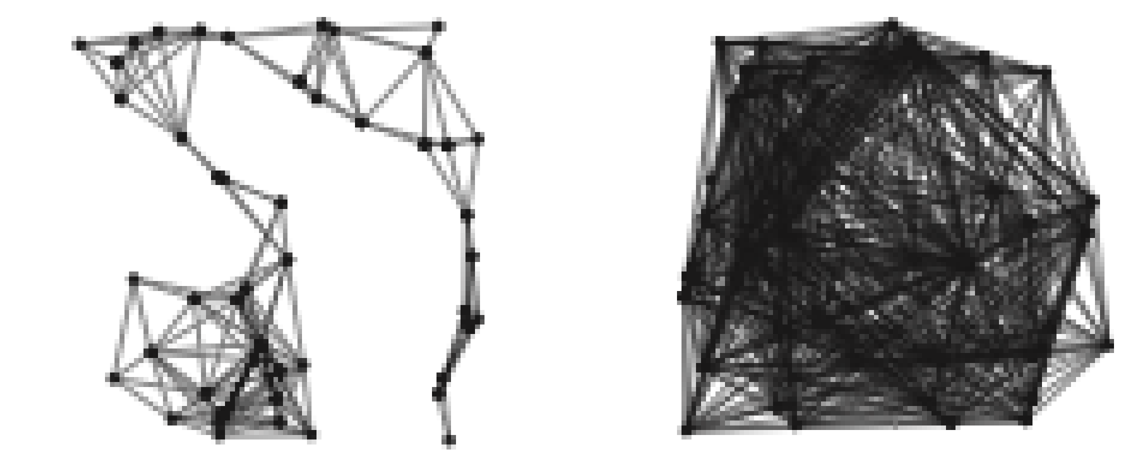
Asymptotic Performance

- n vertices, m edges - no parallel edges - no self-loops - Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse.

huge number of vertices,
small average vertex degree



Two graphs (V = 50)