CS171 Introduction to Computer Science II

Graphs

Graphs

- Examples
- Definitions
- Implementation/Representation of graphs

Graphs

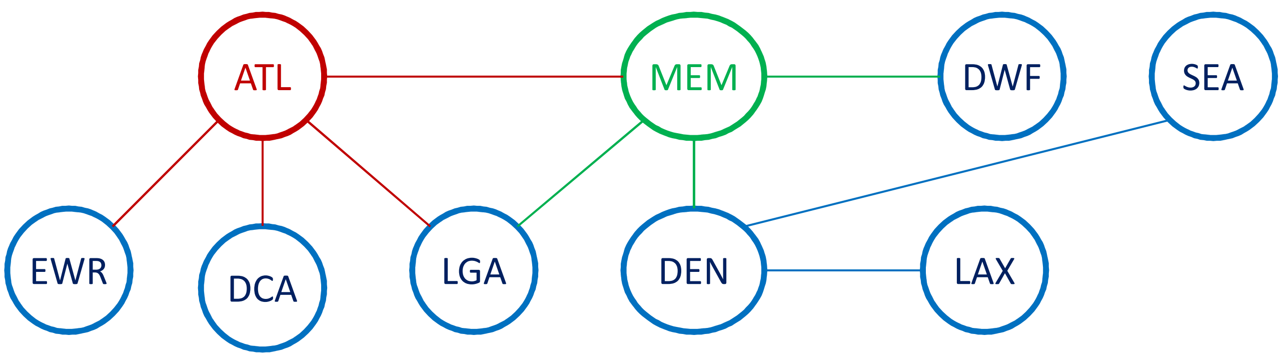
- Graphs: set of vertices connected pairwise by edges
- Interesting and useful structure
- Many practical applications
  - Maps
  - Web content
  - Schedules
  - Social networks
  - …

Delta Airlines Domestic Routes

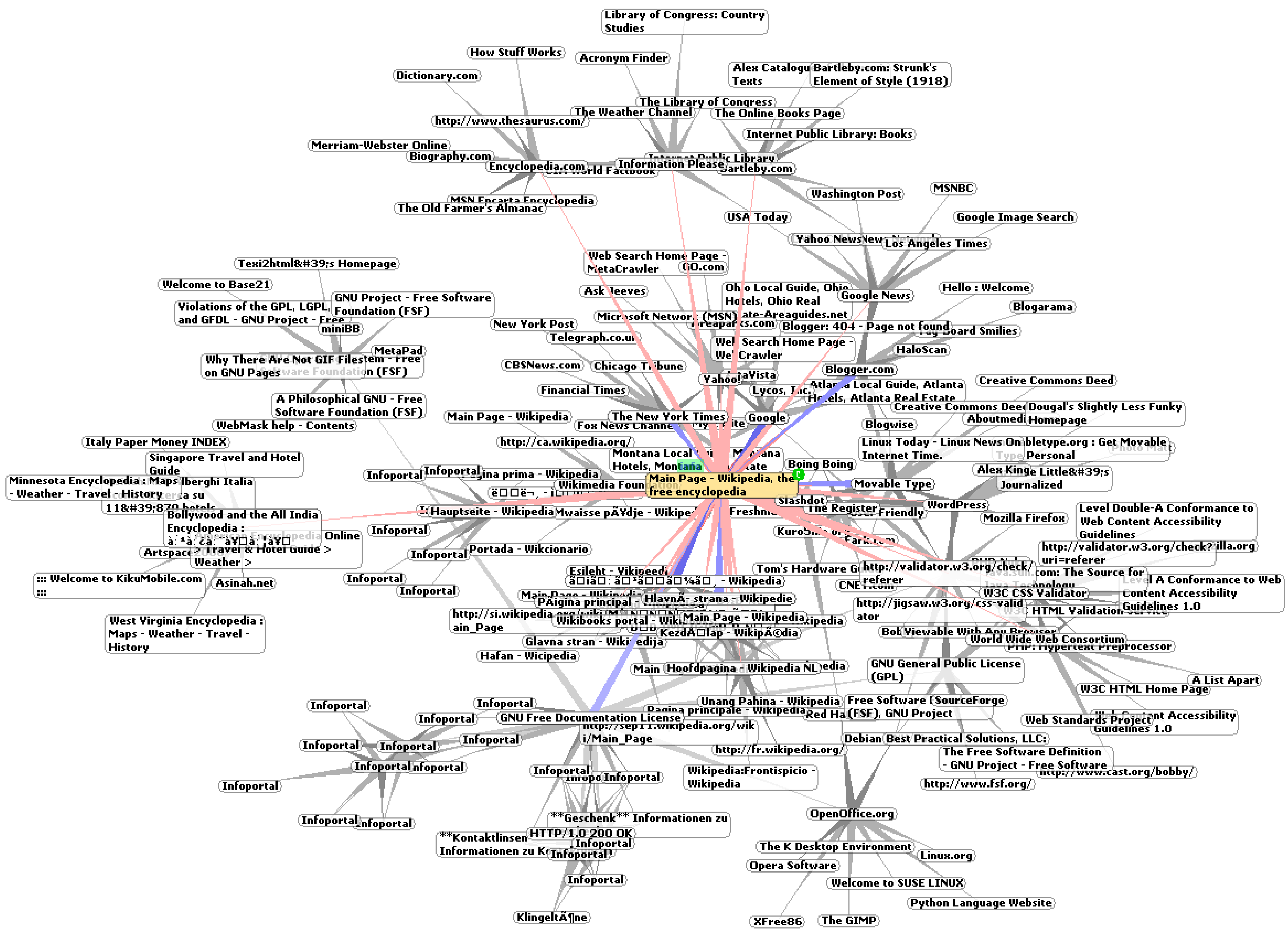
Delta Airlines domestic routes

From Atlanta

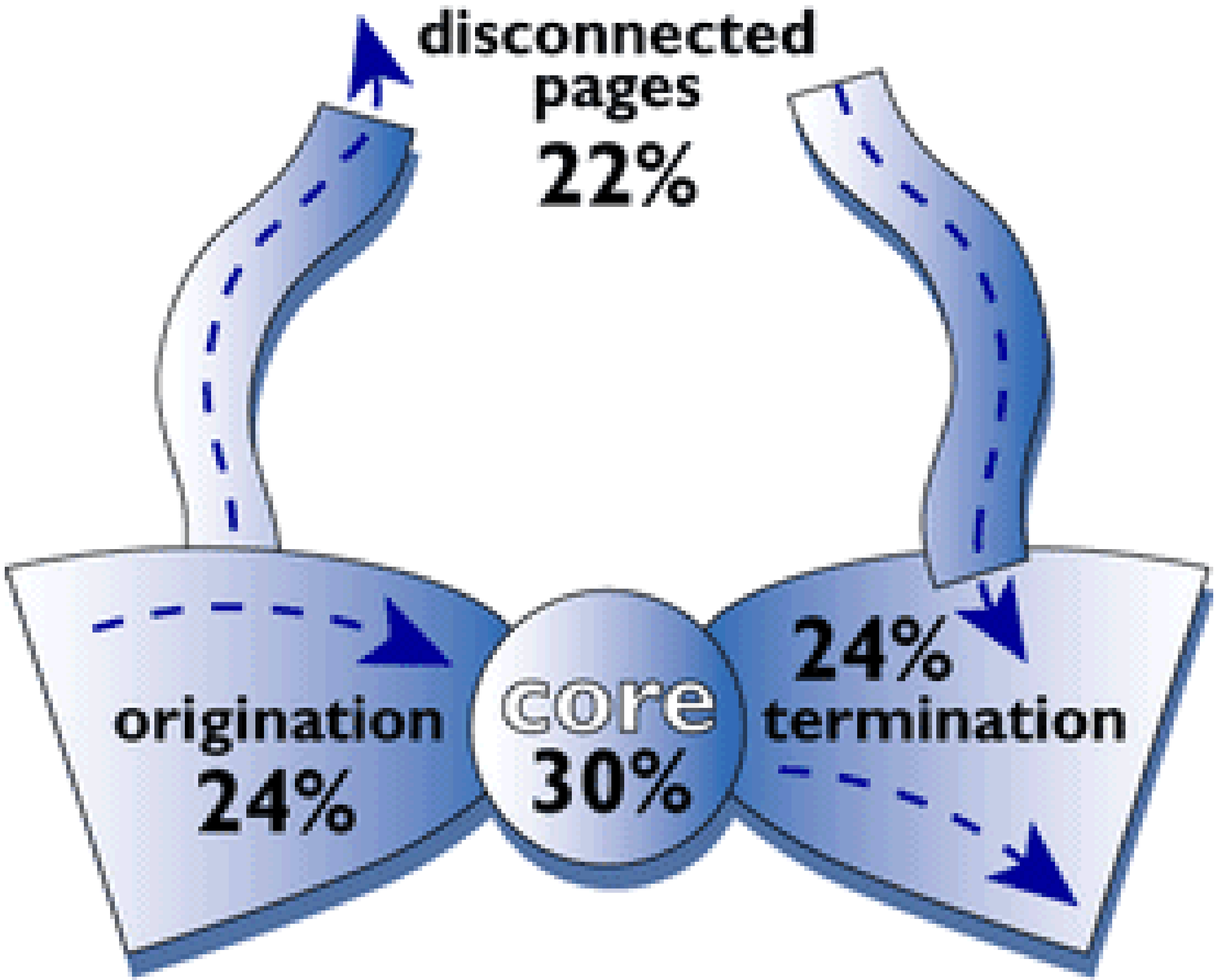
From Memphis



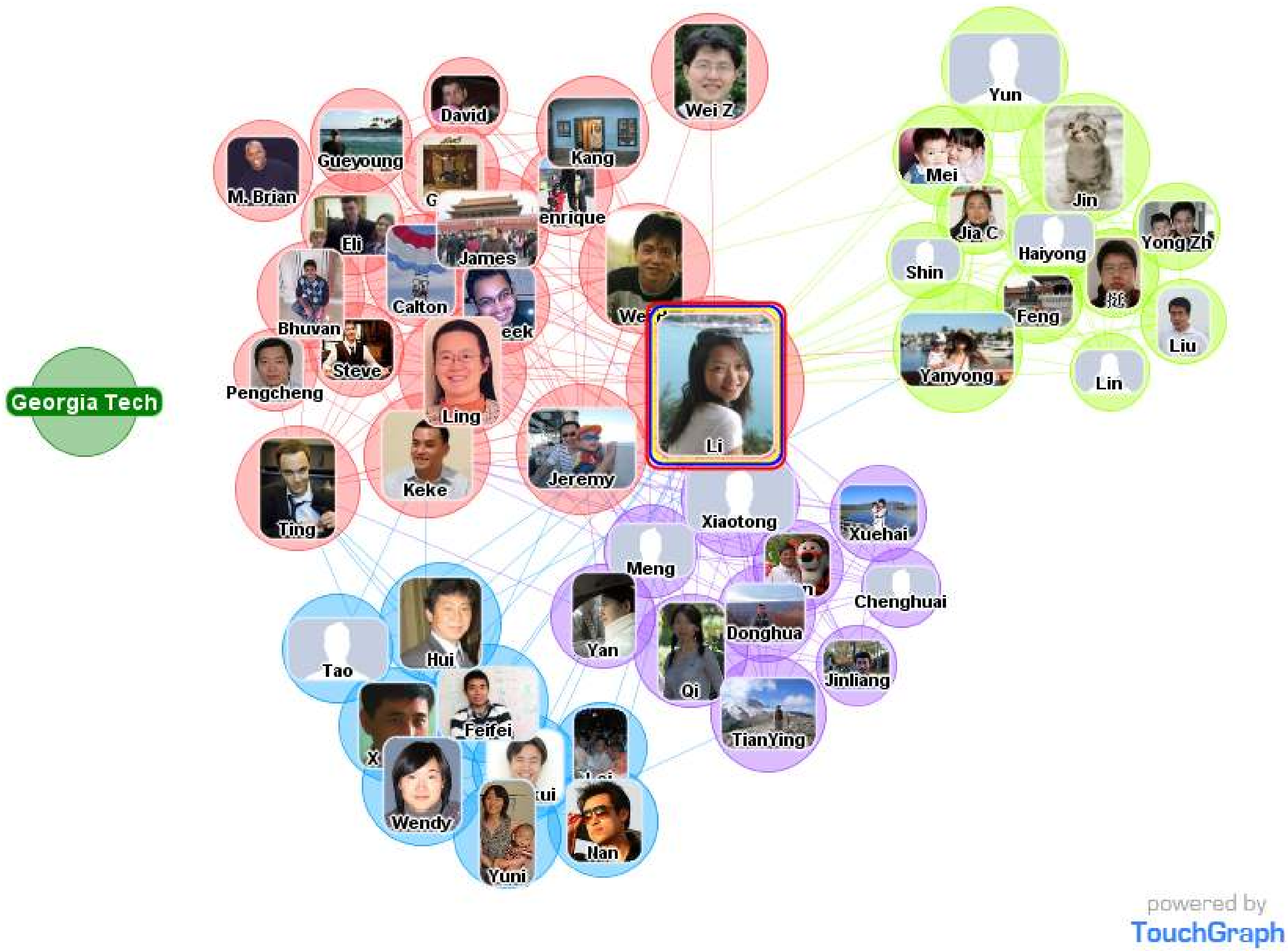
WWW



Bow Tie Theory



Facebook Friend Graph



10 million Facebook friends

"Visualizing Friendships" by Paul Butler

Obesity study in social networks

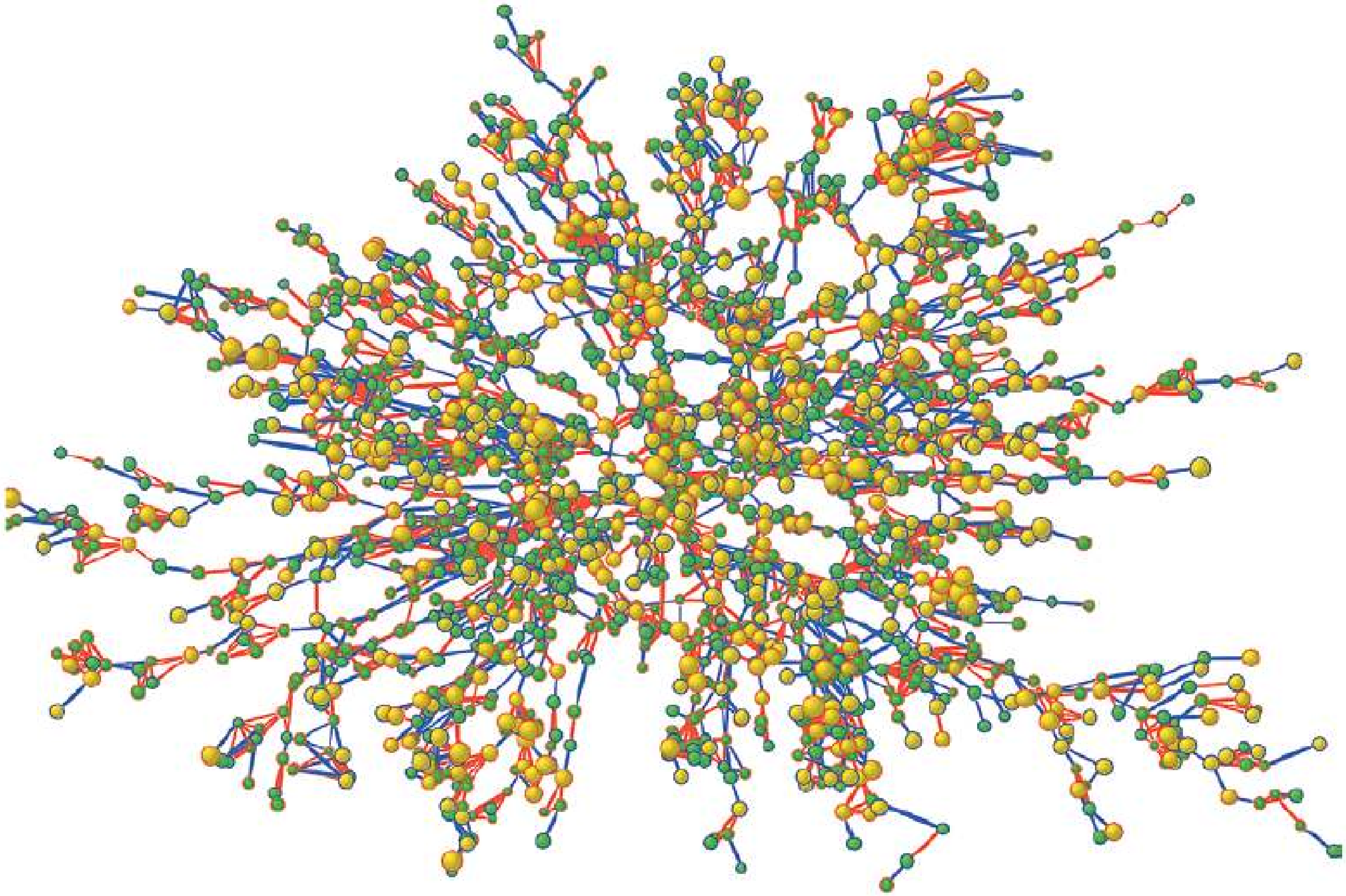
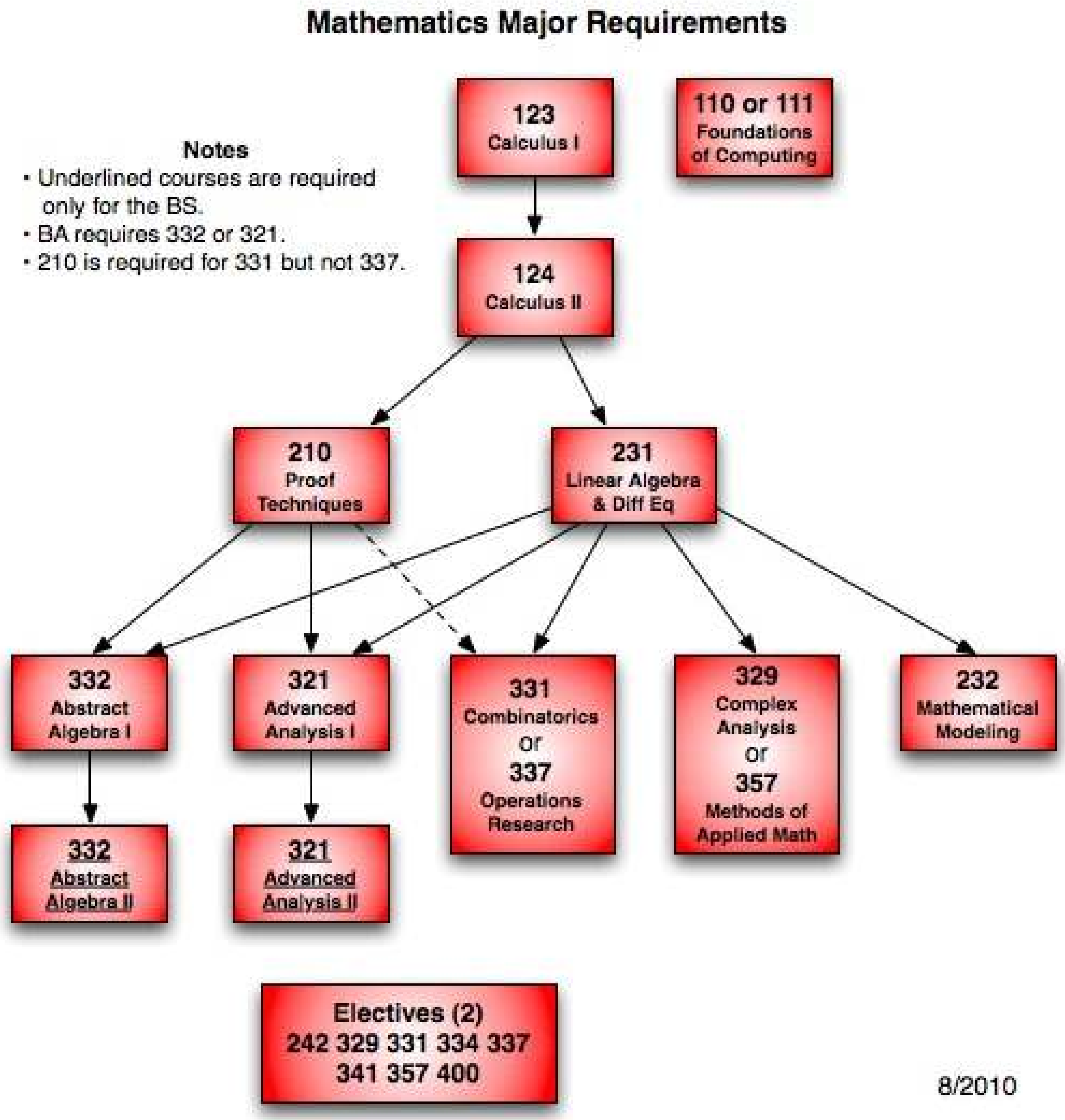


Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.

Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

"The Spread of Obesity in a Large Social Network over 32 Years" by Christakis and Fowler in New England Journal of Medicine, 2007

Course Prerequisite Graph



Graphs

- Undirected graphs
  - simple connections
- Digraphs
  - each connection has a direction
- Edge-weighted graphs
  - each connection has an associated weight
- Edge-weighted digraphs
  - each connection has both a direction and a weight

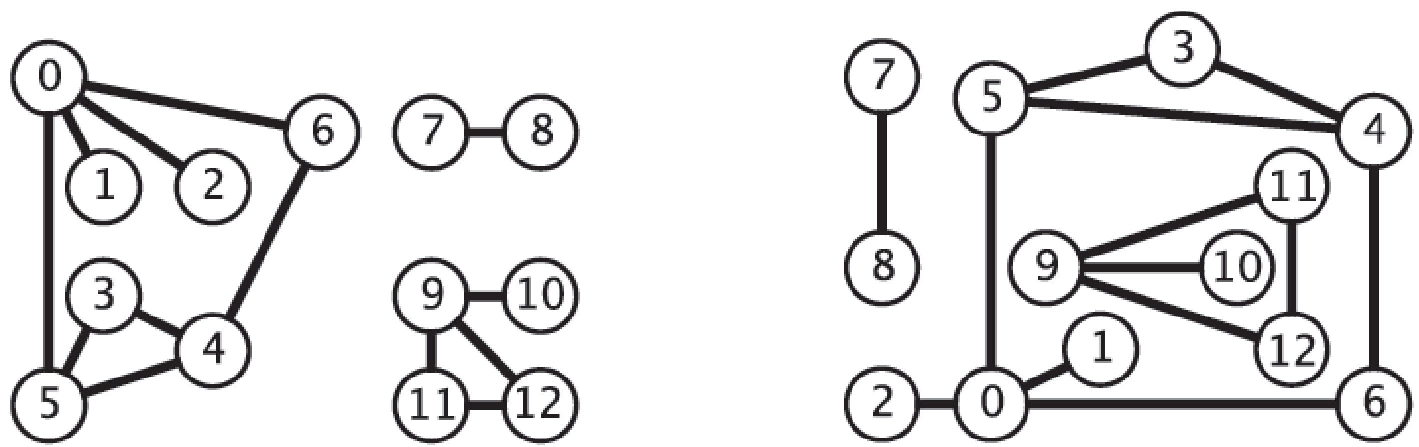
Undirected Graphs

- A graph is a set of vertices and a collection of edges that each connect a pair of vertices

Graph representation

Graph drawing. Provides intuition about the structure of the graph.

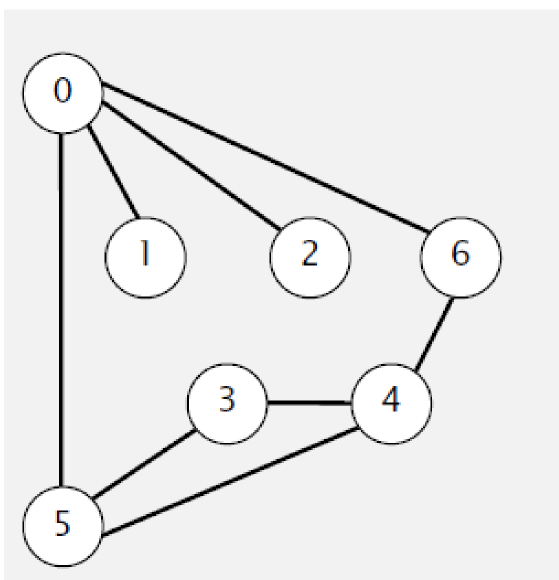
Caveat. Intuition can be misleading.

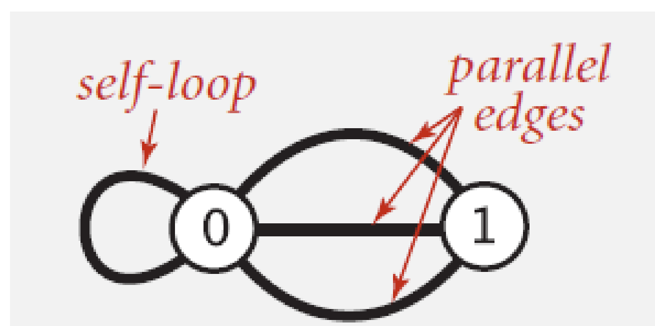


two drawings of the same graph

Glossary

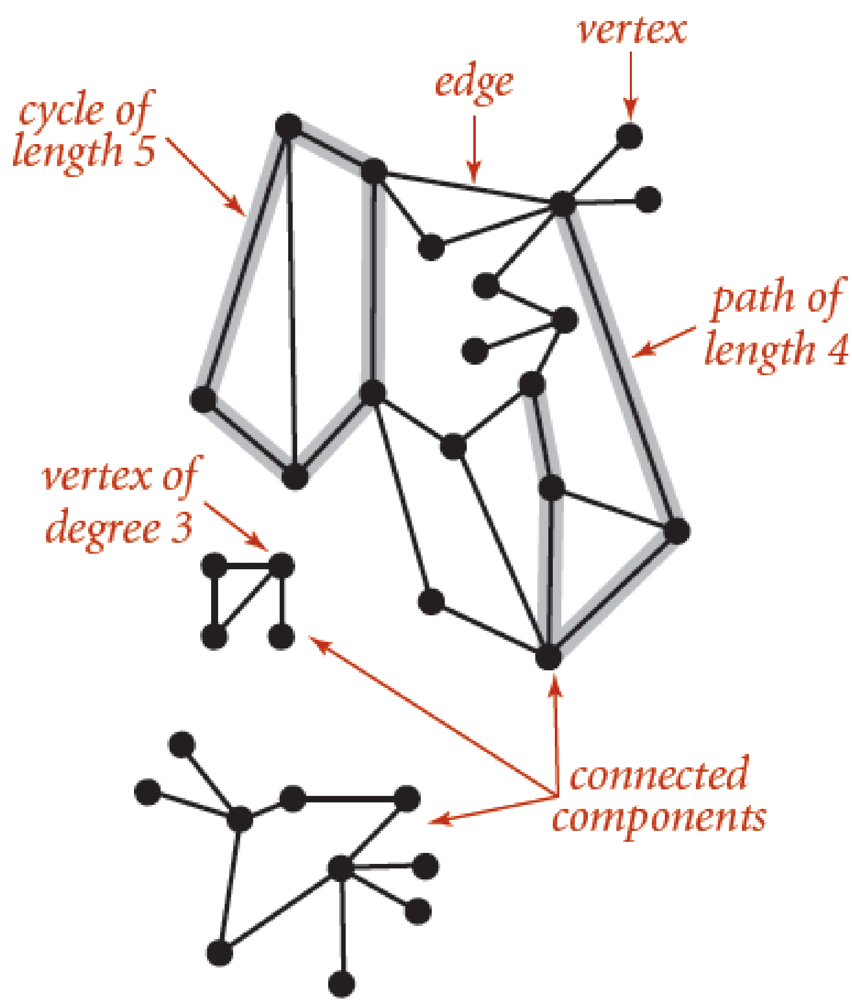
- When there is an edge connecting two vertices, the vertices are **adjacent to** one another and the edge is **incident to** both vertices
- A **self-loop** is an edge that connects a vertex to itself
- Two edges that connect the same pair of vertices are **parallel**
- The **degree** of a vertex is the number of edges incident to the vertex, with loops counted twice
- A **subgraph** is a subset of a graph's edges (and associated vertices) that constitutes a graph





Glossary

- A **path** in a graph is a sequence of vertices connected by edges
  - A **simple path** is one with no repeated vertices
  - A **cycle** is a path with at least one edge whose first and last vertices are the same
  - A **simple cycle** is a cycle with no repeated edges or vertices (except the first and last vertices)
  - The **length** of a path is its number of edges
- One vertex is **connected to** another if there exists a path that contains both of them
- A graph is **connected** if there is a path from every vertex to every other vertex in the graph
  - A graph that is **not connected** consists of a set of **connected components**
- An **acyclic** graph is a graph with no cycles.


vertex
edge
cycle of
length 5
path of
length 4
vertex of
degree 3
connected
components

Graphs

- Examples
- Definitions
- Implementation/Representation of graphs

Graph API

	public class Graph	
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*
String	toString()	*string representation*

```
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
```

read graph from input stream

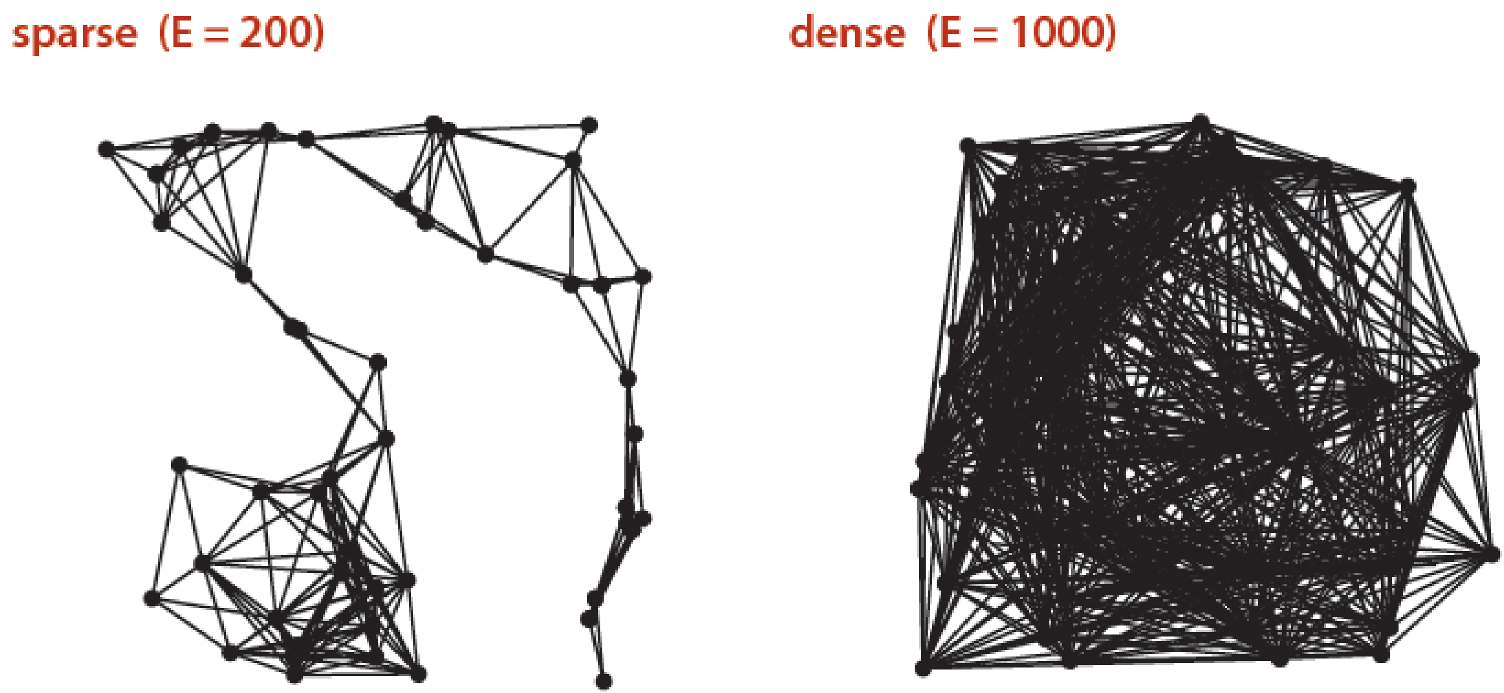
print out each edge (twice)

How to represent/implement a graph?

- Space-efficient
  - Accommodate types of graphs that likely to encounter
- Time-efficient
  - Add an edge
  - If there is edge between v and w
  - Iterate over vertices adjacent to v
  - …

Real-world graphs

- Real-world graphs tend to be "sparse"
  - Huge number of vertices, small average vertex



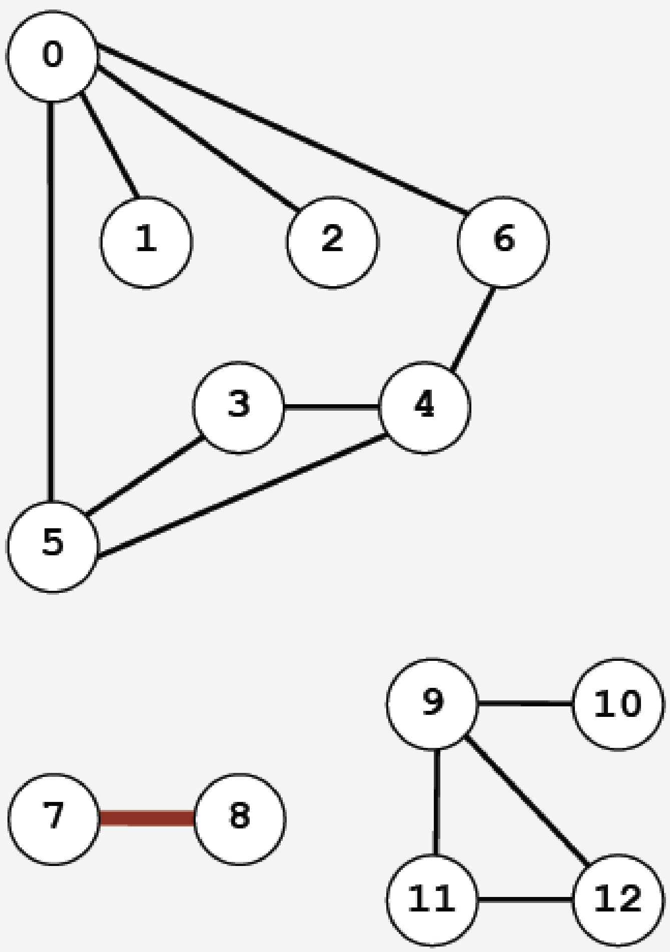
Two graphs (V = 50)

Representation Options

- Edge list
- Adjacency matrix
- Adjacency lists

Set-of-edges graph representation

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

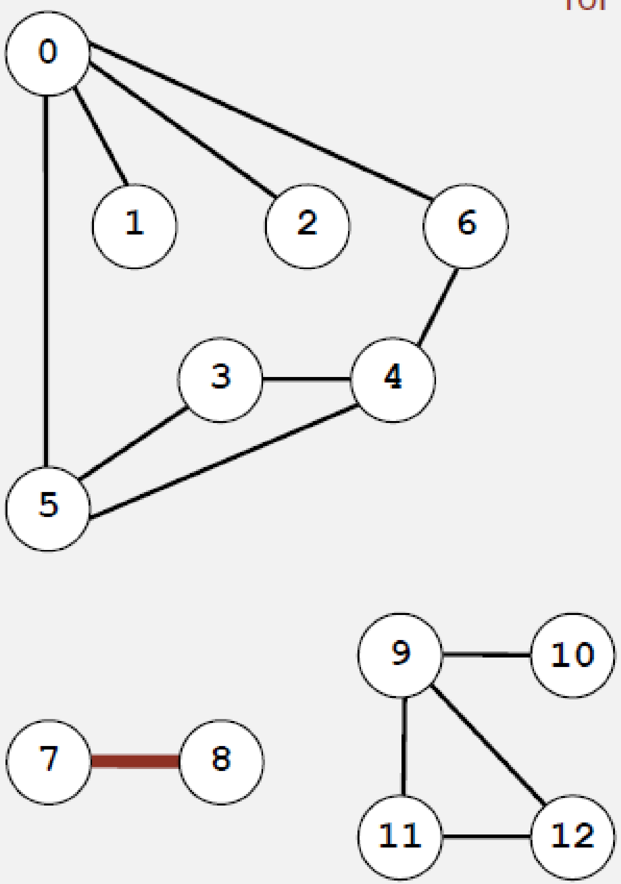


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v–w in graph: `adj[v][w] = adj[w][v] = true.`

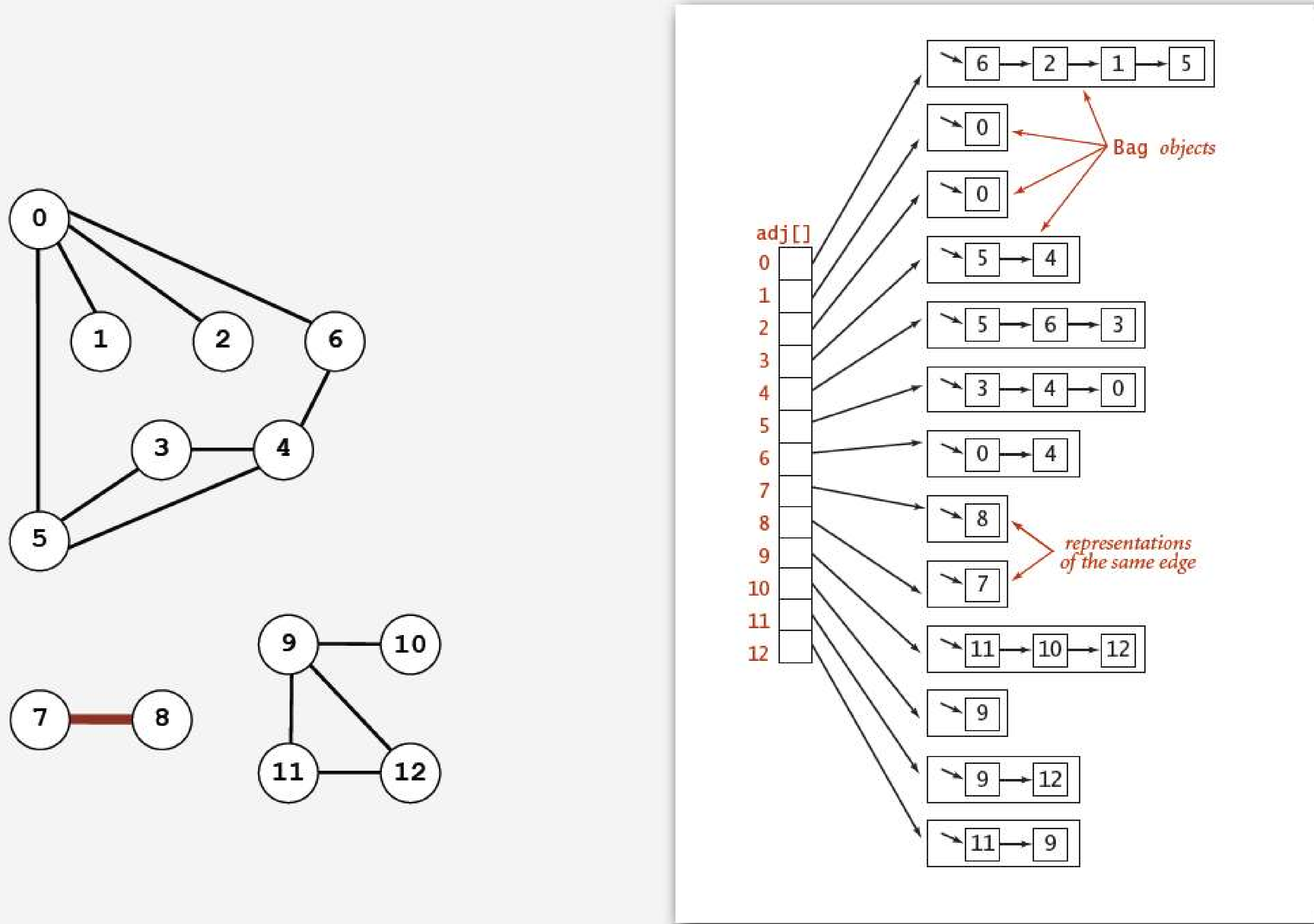


two entries
for each edge

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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



Graph representations

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

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

* disallows parallel edges

Adjacency-list graph representation: Java implementation

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists
(using **`Bag`** data type)

create empty graph
with **`V`** vertices

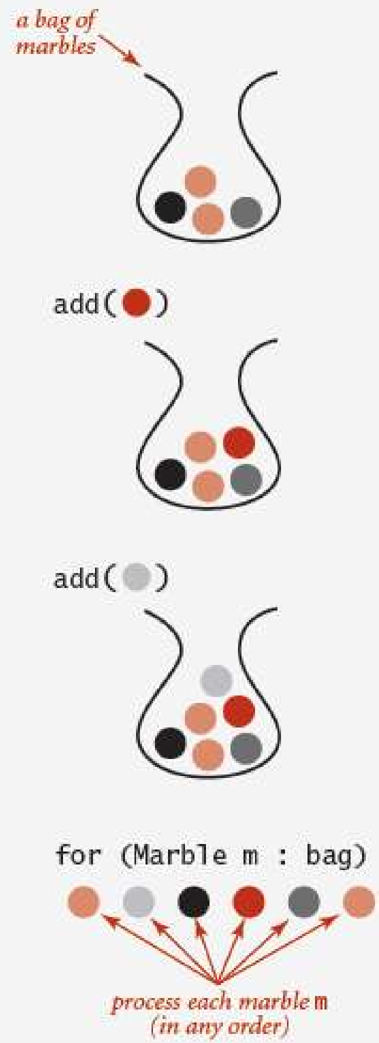
add edge **`v-w`**
(parallel edges allowed)

iterator for vertices adjacent to **`v`**

Bag API

Main application. Adding items to a collection and iterating (when order doesn't matter).

public class Bag<Item> implements Iterable<Item>		
	`Bag()`	*create an empty bag*
`void`	`add(Item x)`	*insert a new item onto bag*
`int`	`size()`	*number of items in bag*
`Iterable<Item>`	`iterator()`	*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).

Full implementation

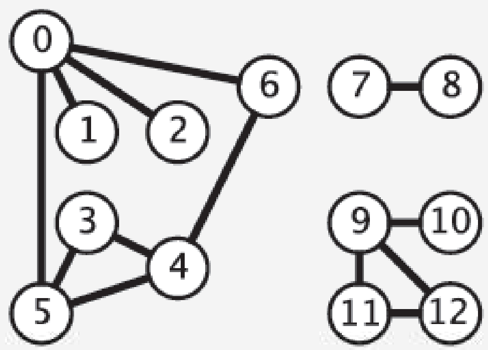
- http://algs4.cs.princeton.edu/41undirected/Graph.java.html

Graph API: sample client

Graph input format.

tinyG.txt

```
V → 13
    13 ← E
    0 5
    4 3
    0 1
    9 12
    6 4
    5 4
    0 2
    11 12
    9 10
    0 6
    7 8
    9 11
    5 3
```




```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
...
12-11
12-9
```

```
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
```

read graph from input stream

print out each edge (twice)

Typical graph-processing code

compute the degree of v	<pre>public static int degree(Graph G, int v)
	{
	   int degree = 0;
	   for (int w : G.adj(v)) degree++;
	   return degree;
	}</pre>

compute maximum degree	<pre>public static int maxDegree(Graph G)
	{
	   int max = 0;
	   for (int v = 0; v < G.V(); v++)
	      if (degree(G, v) > max)
	         max = degree(G, v);
	   return max;
	}</pre>

compute average degree	<pre>public static int avgDegree(Graph G)
	{
	   return 2 * G.E() / G.V();
	}</pre>

count self-loops	<pre>public static int numberOfSelfLoops(Graph G)
	{
	   int count = 0;
	   for (int v = 0; v < G.V(); v++)
	      for (int w : G.adj(v))
	         if (v == w) count++;
	   return count/2;
	}</pre>