

# Graph

---

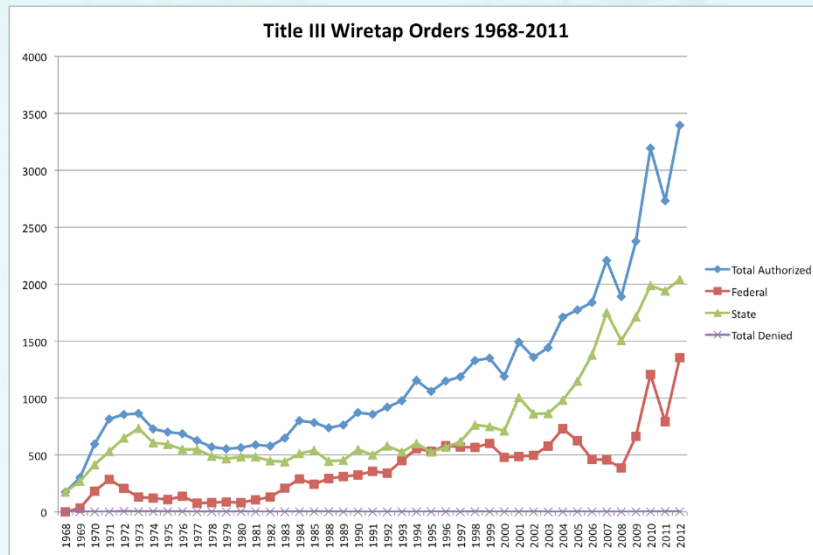
- **Introduction**
- **Graph API**
- Elementary Graph Operations
  - DFS: Depth first search
  - BFS: Breadth first search
  - CC: Connected components

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4<sup>th</sup> edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

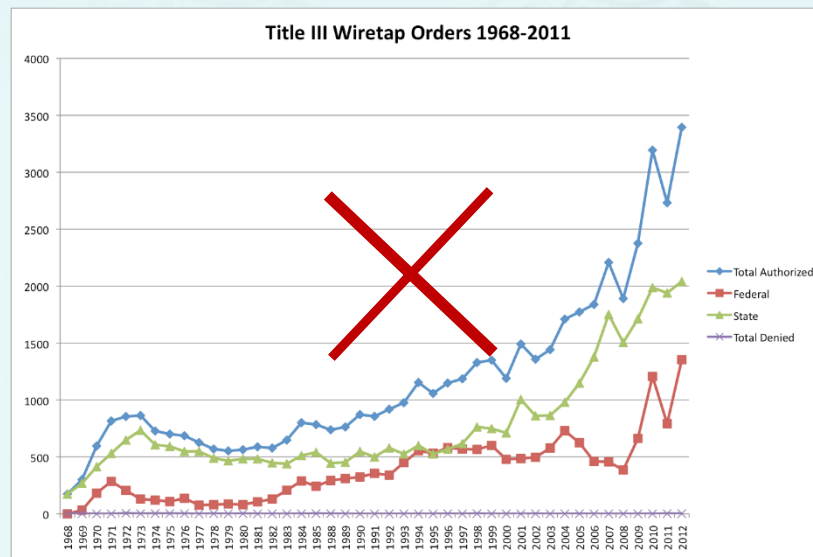
# Undirected graphs

Graph: Set of vertices connected pairwise by edges.



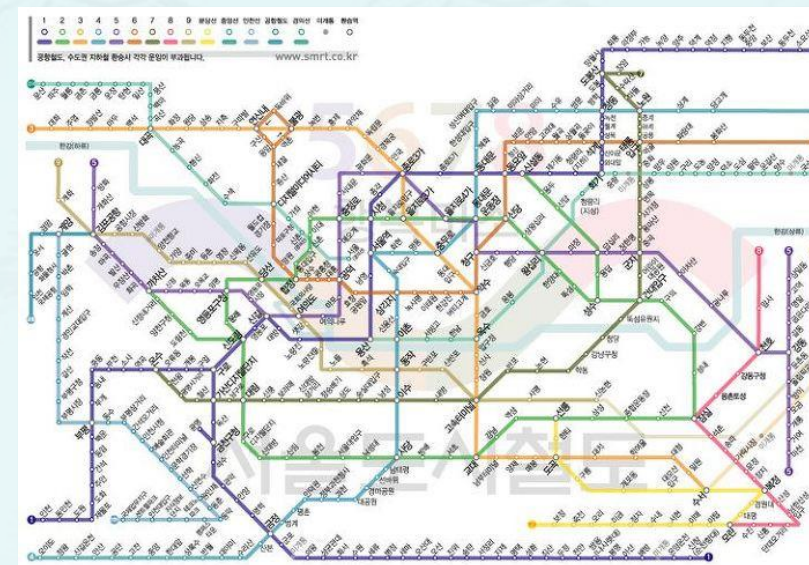
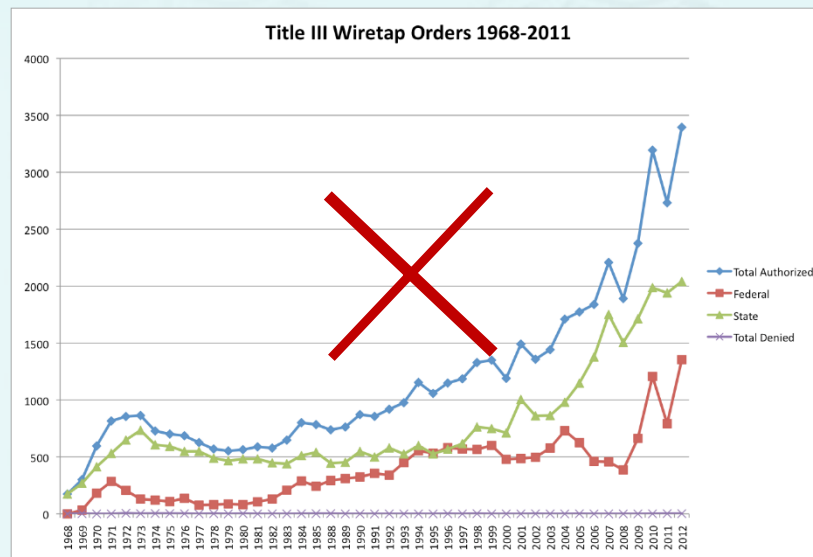
# Undirected graphs

Graph: Set of vertices connected pairwise by edges.



# Undirected graphs

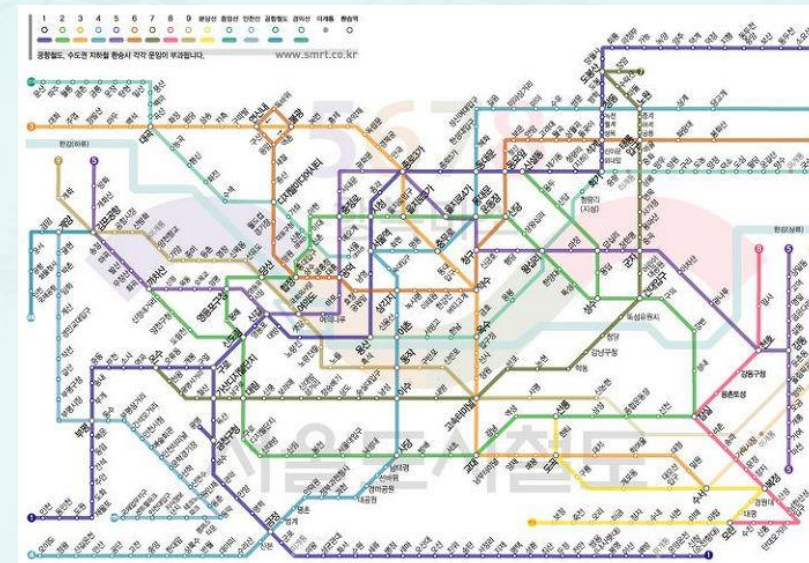
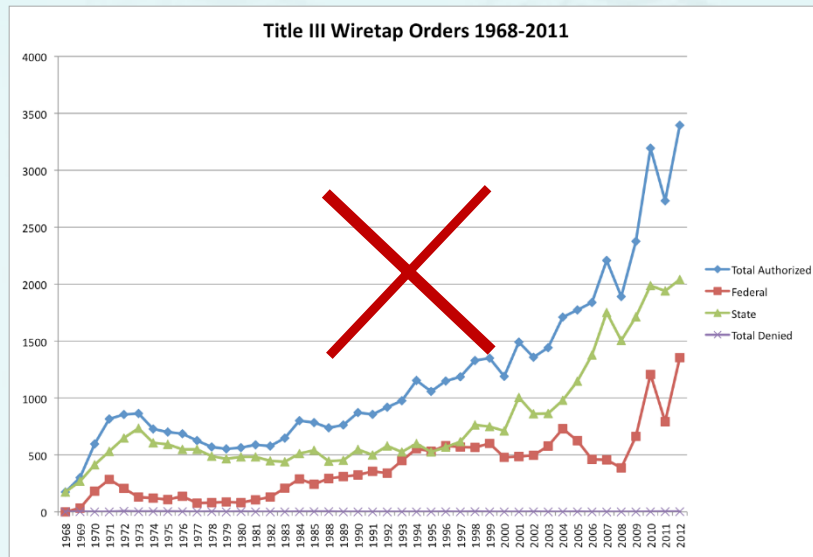
Graph: Set of vertices connected pairwise by edges.



# Undirected graphs

Graph: Set of vertices connected pairwise by edges.

- *Why study graph algorithms?*

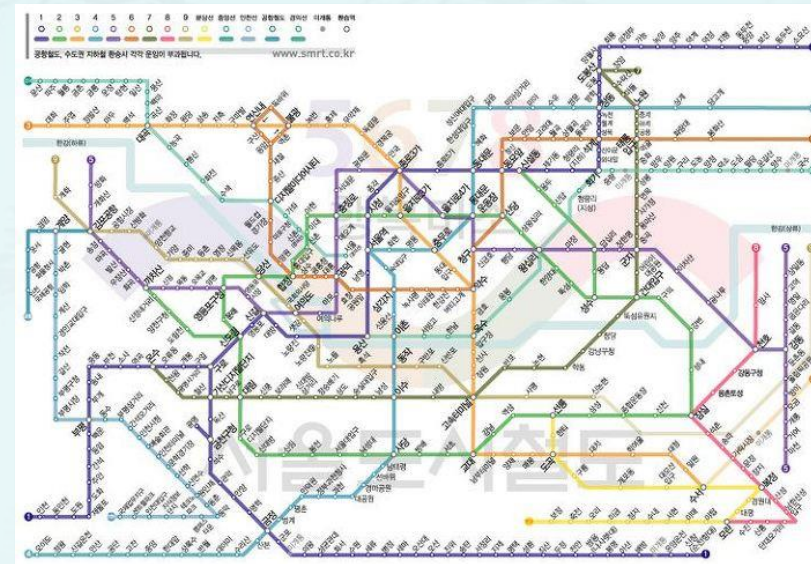
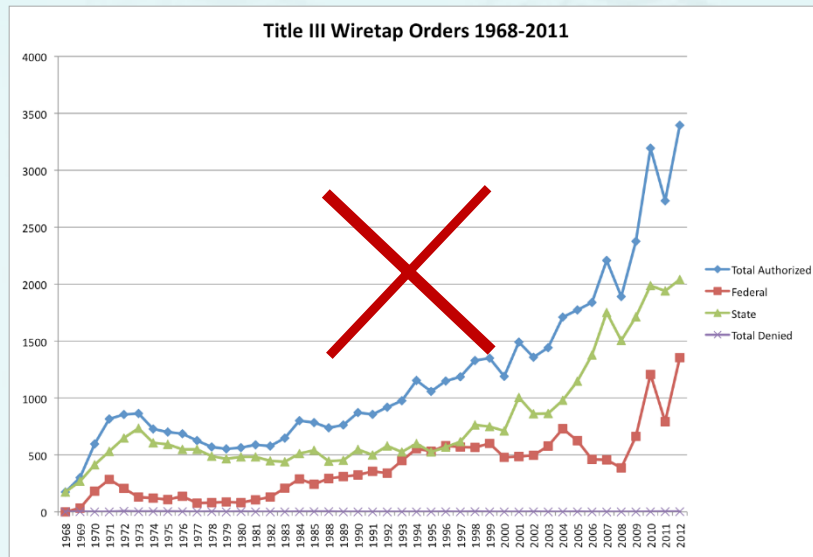




# Undirected graphs

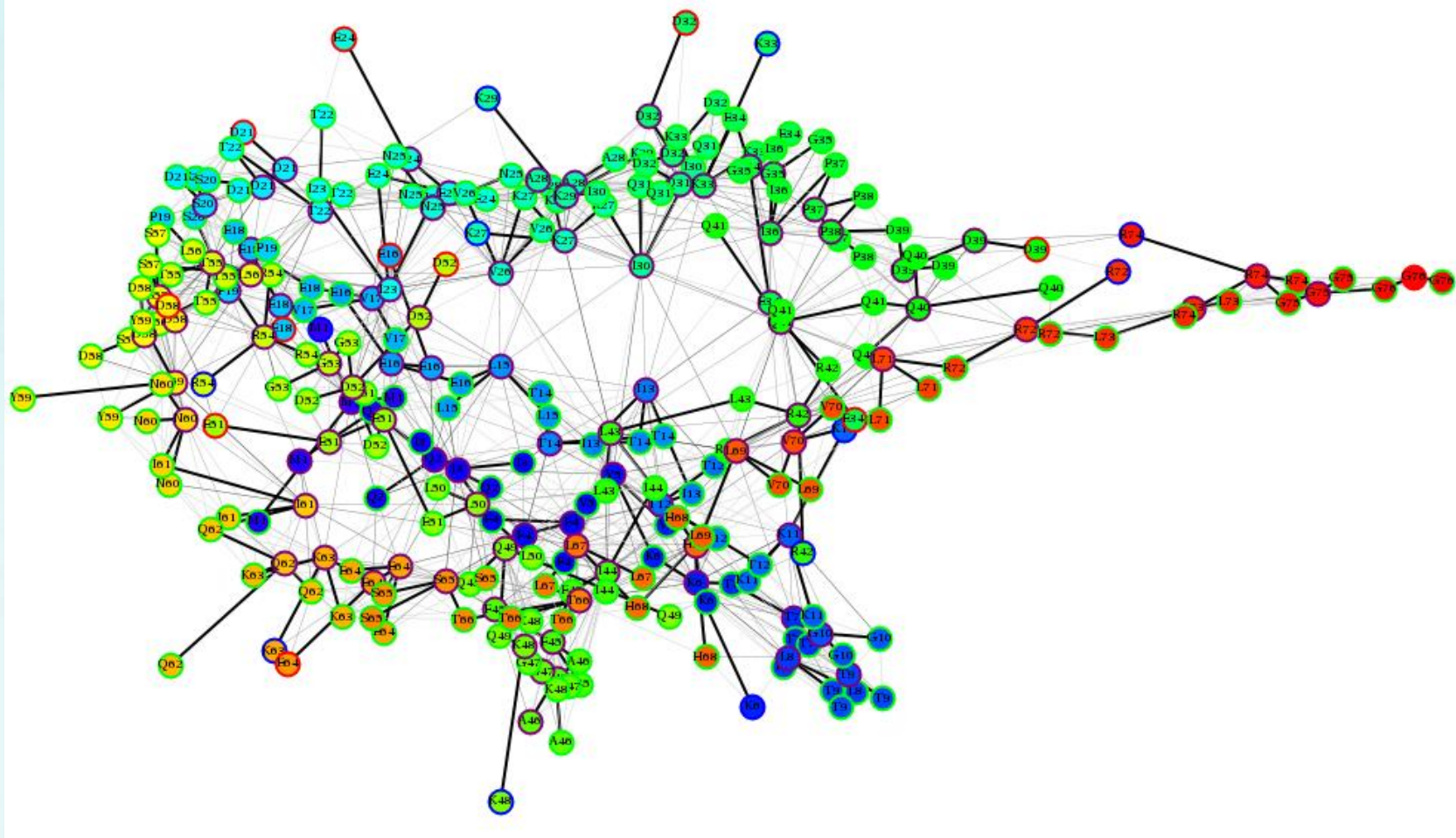
**Graph:** Set of vertices connected pairwise by edges.

- *Why study graph algorithms?*
  - Thousands of practical applications.
  - Hundreds of graph algorithms known.
  - Interesting and broadly useful abstraction.
  - Challenging branch of computer science and discrete math.



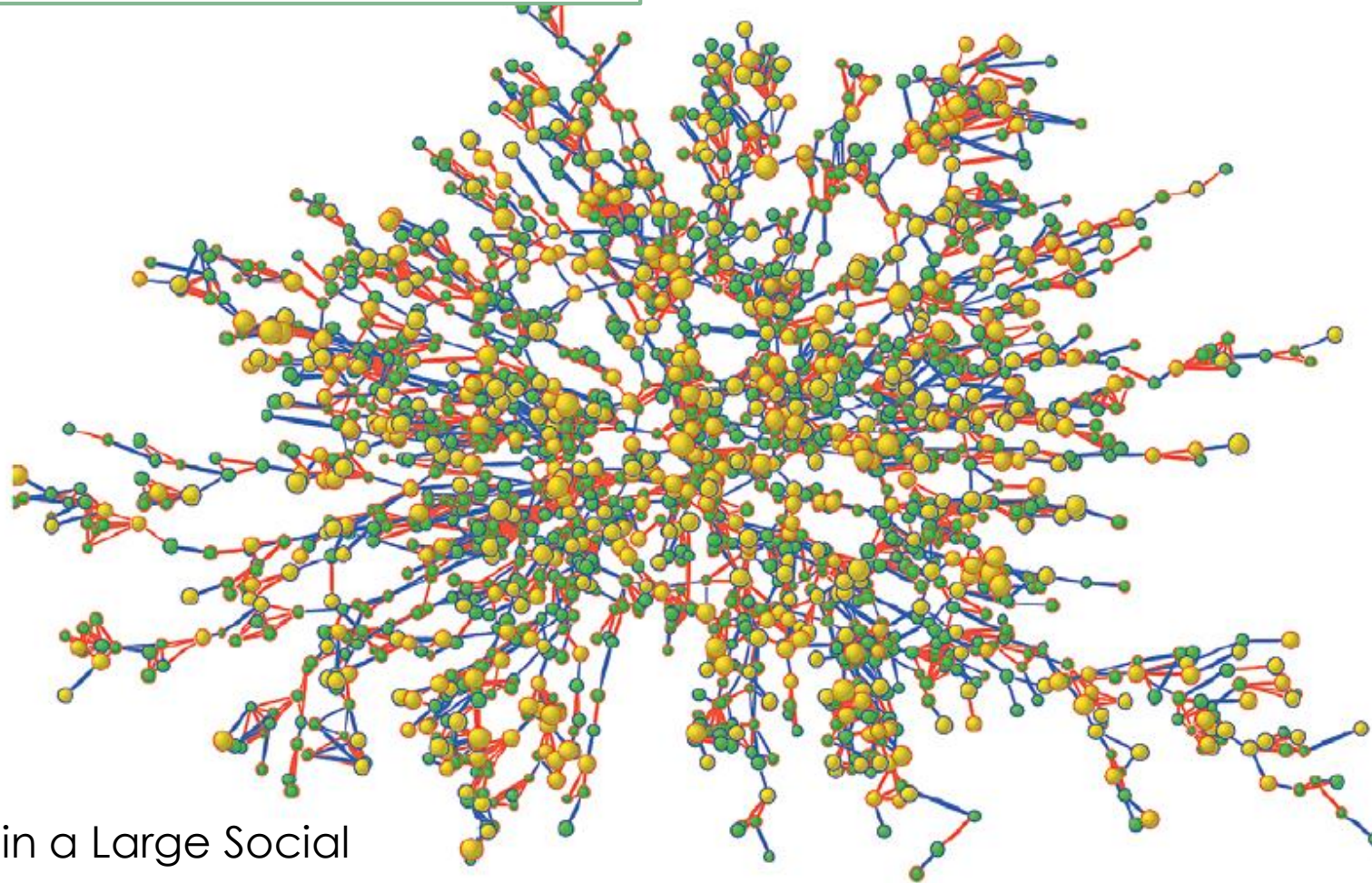
# Undirected graphs

## Chemical Environments: Protein Graphs



Reference: **Benson NC**, Daggett V (2012) A comparison of methods for the analysis of molecular dynamics simulations. *J. Phys. Chem. B* **116**(29): 8722-31.





## The Spread of Obesity in a Large Social Network over 32 Years

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

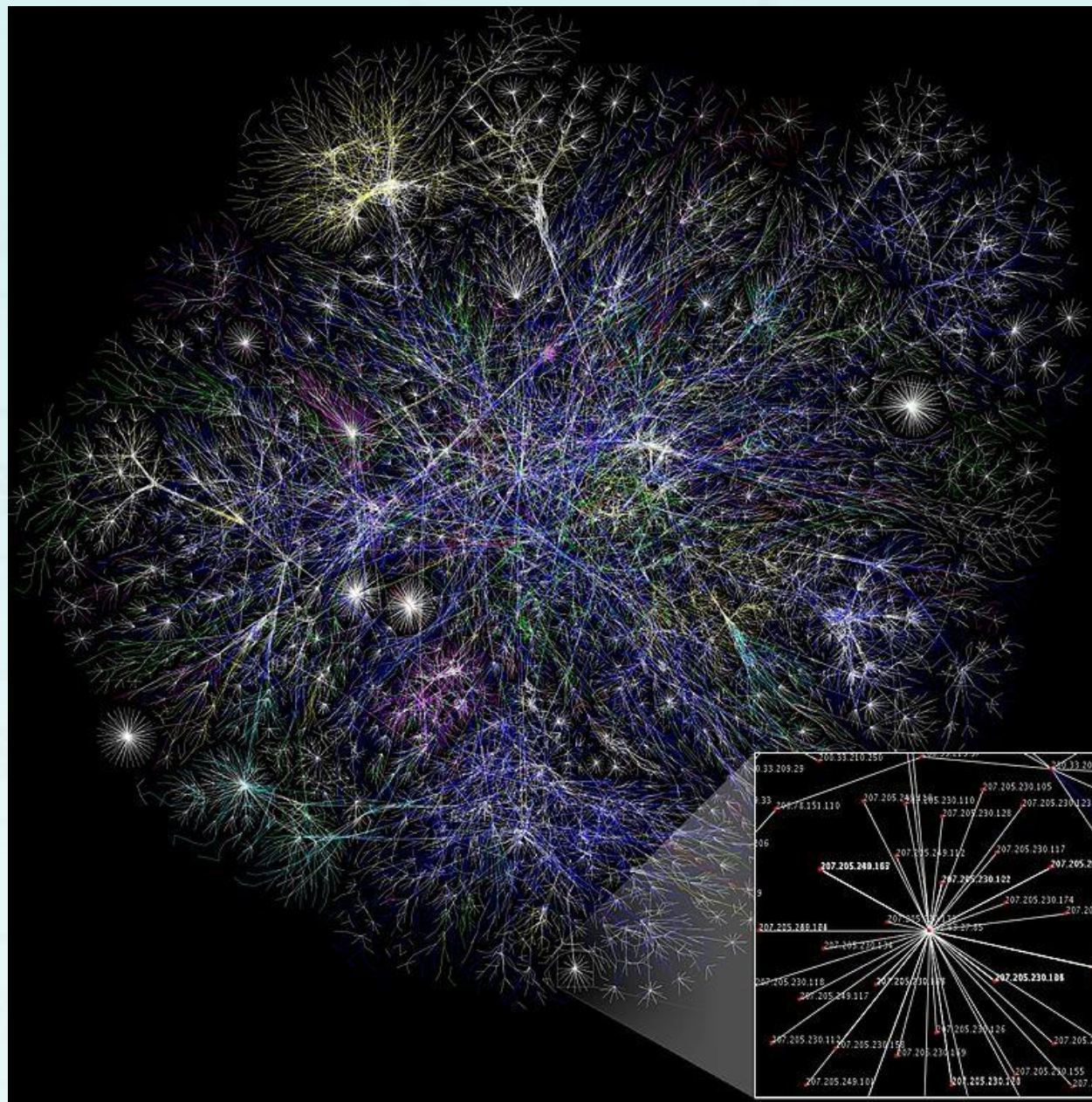
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index,  $\geq 30$ ) 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

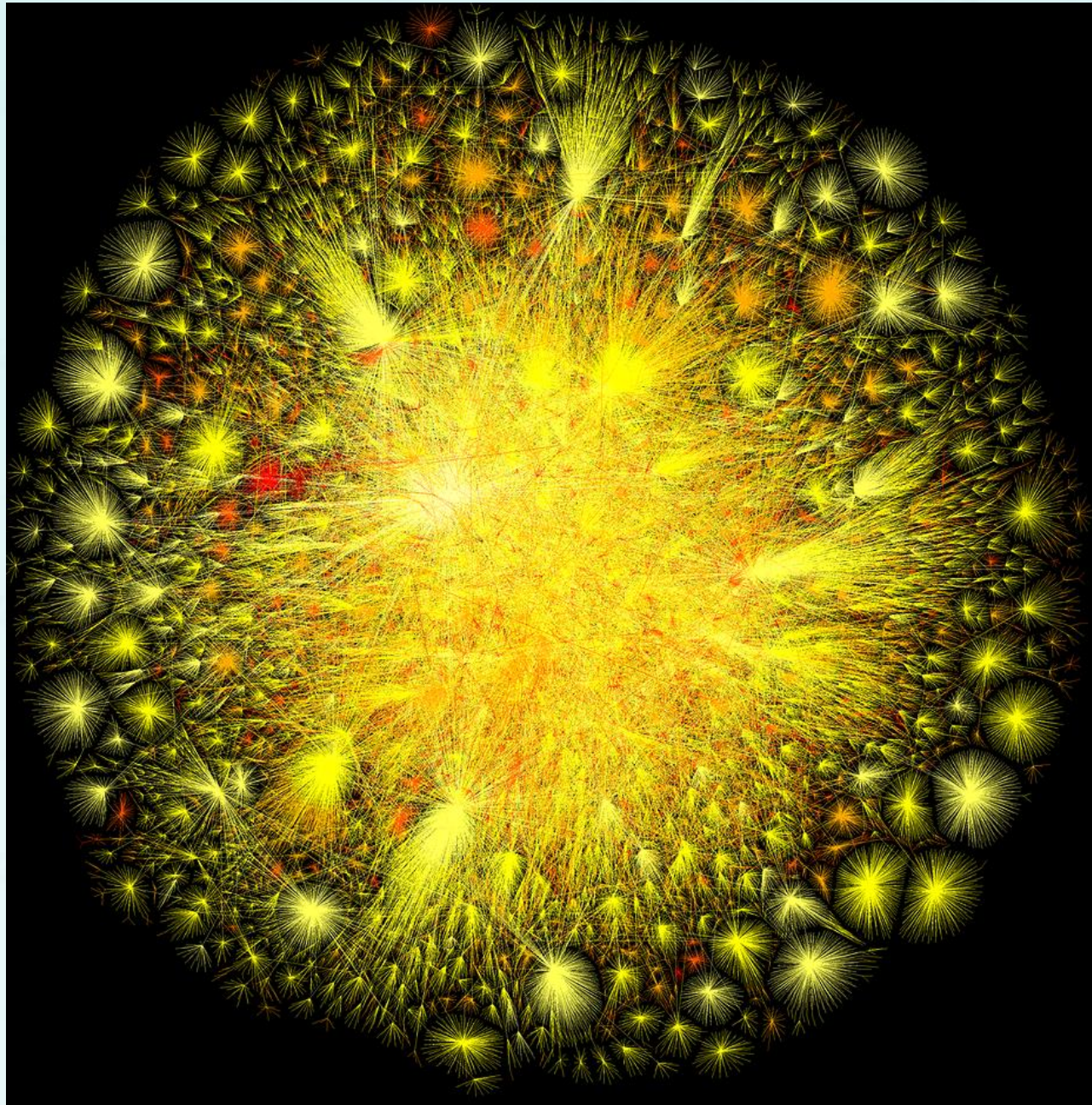
<http://www.nejm.org/doi/full/10.1056/NEJMsa066082>

<http://www.youtube.com/watch?v=pJfq-o5nZQ4>

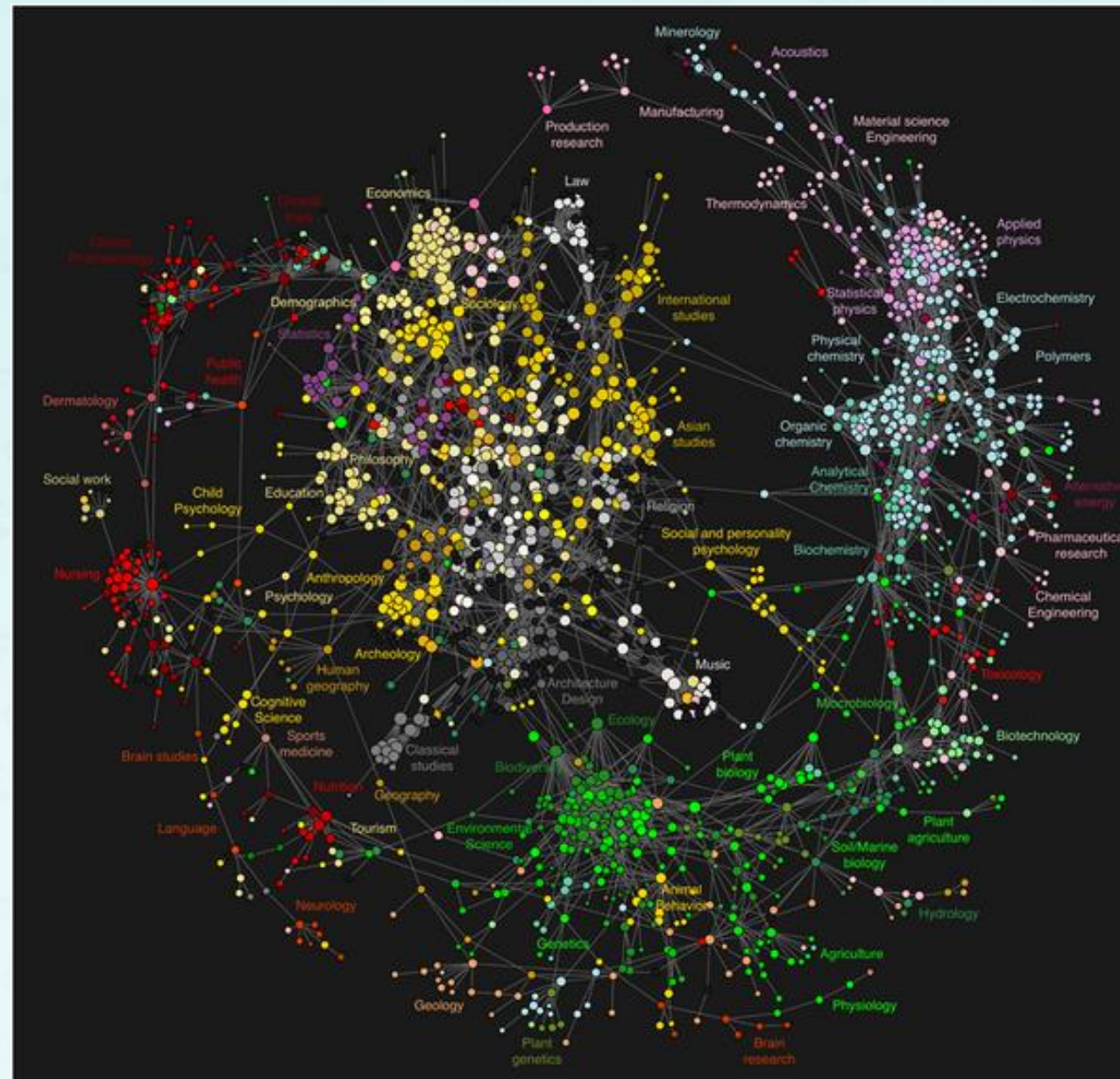


**the Opte Project:** Visualization of the various routes through a portion of the Internet









Clickstream Data Yields High-Resolution Maps of Science.

<http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803>





"Visualizing Friendships" by Paul Butler – an intern at Facebook

## Graph Applications

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond



## Graph Terminology

---

**Path:** Sequence of vertices connected by edges.

**Cycle:** Path whose first and last vertices are the same.

## Graph Terminology

---

**Path:** Sequence of vertices connected by edges.

**Cycle:** Path whose first and last vertices are the same.

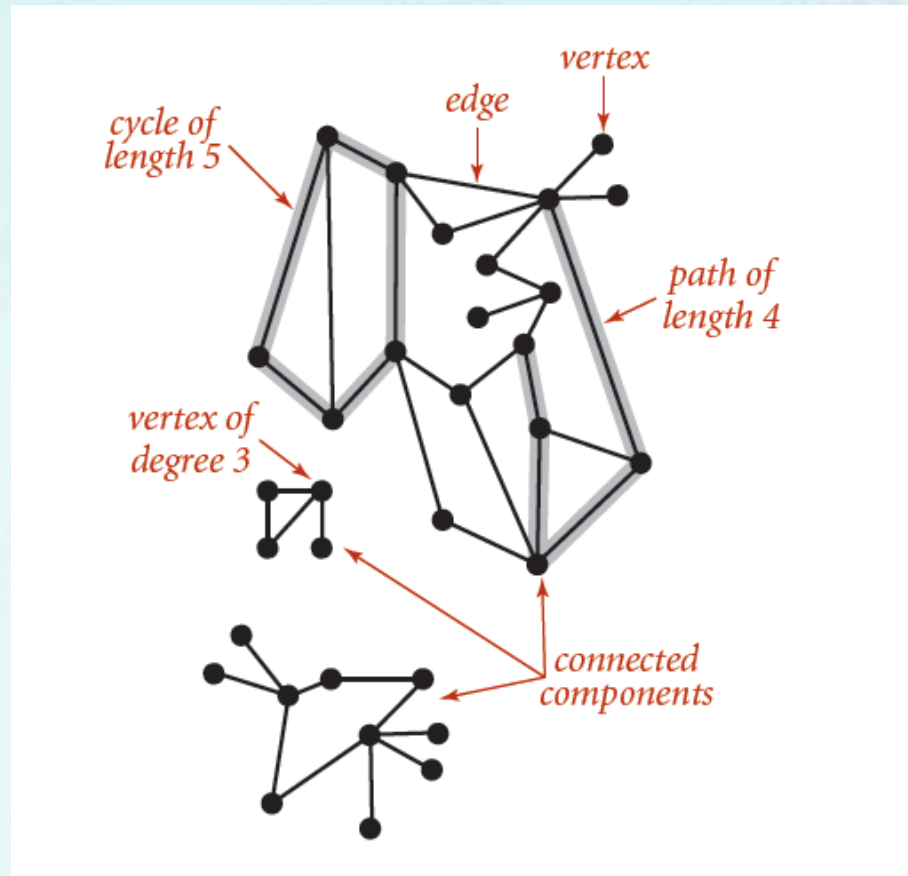
Two vertices are **connected** if there is a path between them.

# Graph Terminology

**Path:** Sequence of vertices connected by edges.

**Cycle:** Path whose first and last vertices are the same.

Two vertices are **connected** if there is a path between them.





## Some graph-processing problems

---

**Path**

Is there a path between  $s$  and  $t$  ?

**Shortest Path**

What is the shortest path between  $s$  and  $t$  ?

## Some graph-processing problems

---

<b>Path</b>	Is there a path between $s$ and $t$ ?
<b>Shortest Path</b>	What is the shortest path between $s$ and $t$ ?
<b>Cycle</b>	Is there a cycle in the graph?
<b>Euler tour</b>	Is there a cycle that uses each <b>edge</b> exactly once?
<b>Hamilton tour</b>	Is there a cycle that uses each <b>vertex</b> exactly once.

## Some graph-processing problems

---

<b>Path</b>	Is there a path between $s$ and $t$ ?
<b>Shortest Path</b>	What is the shortest path between $s$ and $t$ ?
<b>Cycle</b>	Is there a cycle in the graph?
<b>Euler tour</b>	Is there a cycle that uses each <b>edge</b> exactly once?
<b>Hamilton tour</b>	Is there a cycle that uses each <b>vertex</b> exactly once.
<b>Connectivity</b>	Is there a way to connect all of the vertices?
<b>MST</b>	What is the best way to connect all of the vertices?
<b>BiConnectivity</b>	Is there a vertex whose removal disconnects the graph?



## Some graph-processing problems

---

<b>Path</b>	Is there a path between $s$ and $t$ ?
<b>Shortest Path</b>	What is the shortest path between $s$ and $t$ ?
<b>Cycle</b>	Is there a cycle in the graph?
<b>Euler tour</b>	Is there a cycle that uses each <b>edge</b> exactly once?
<b>Hamilton tour</b>	Is there a cycle that uses each <b>vertex</b> exactly once.
<b>Connectivity</b>	Is there a way to connect all of the vertices?
<b>MST</b>	What is the best way to connect all of the vertices?
<b>BiConnectivity</b>	Is there a vertex whose removal disconnects the graph?
<b>Planarity</b>	Can you draw the graph in the plane with no crossing edges
<b>Graph isomorphism</b>	Do two adjacency lists represent the same graph?

## Some graph-processing problems

---

<b>Path</b>	Is there a path between $s$ and $t$ ?
<b>Shortest Path</b>	What is the shortest path between $s$ and $t$ ?
<b>Cycle</b>	Is there a cycle in the graph?
<b>Euler tour</b>	Is there a cycle that uses each <b>edge</b> exactly once?
<b>Hamilton tour</b>	Is there a cycle that uses each <b>vertex</b> exactly once.
<b>Connectivity</b>	Is there a way to connect all of the vertices?
<b>MST</b>	What is the best way to connect all of the vertices?
<b>BiConnectivity</b>	Is there a vertex whose removal disconnects the graph?
<b>Planarity</b>	Can you draw the graph in the plane with no crossing edges
<b>Graph isomorphism</b>	Do two adjacency lists represent the same graph?
<b>Challenge</b>	Which of these problems are easy? difficult? intractable?

# Graph

---

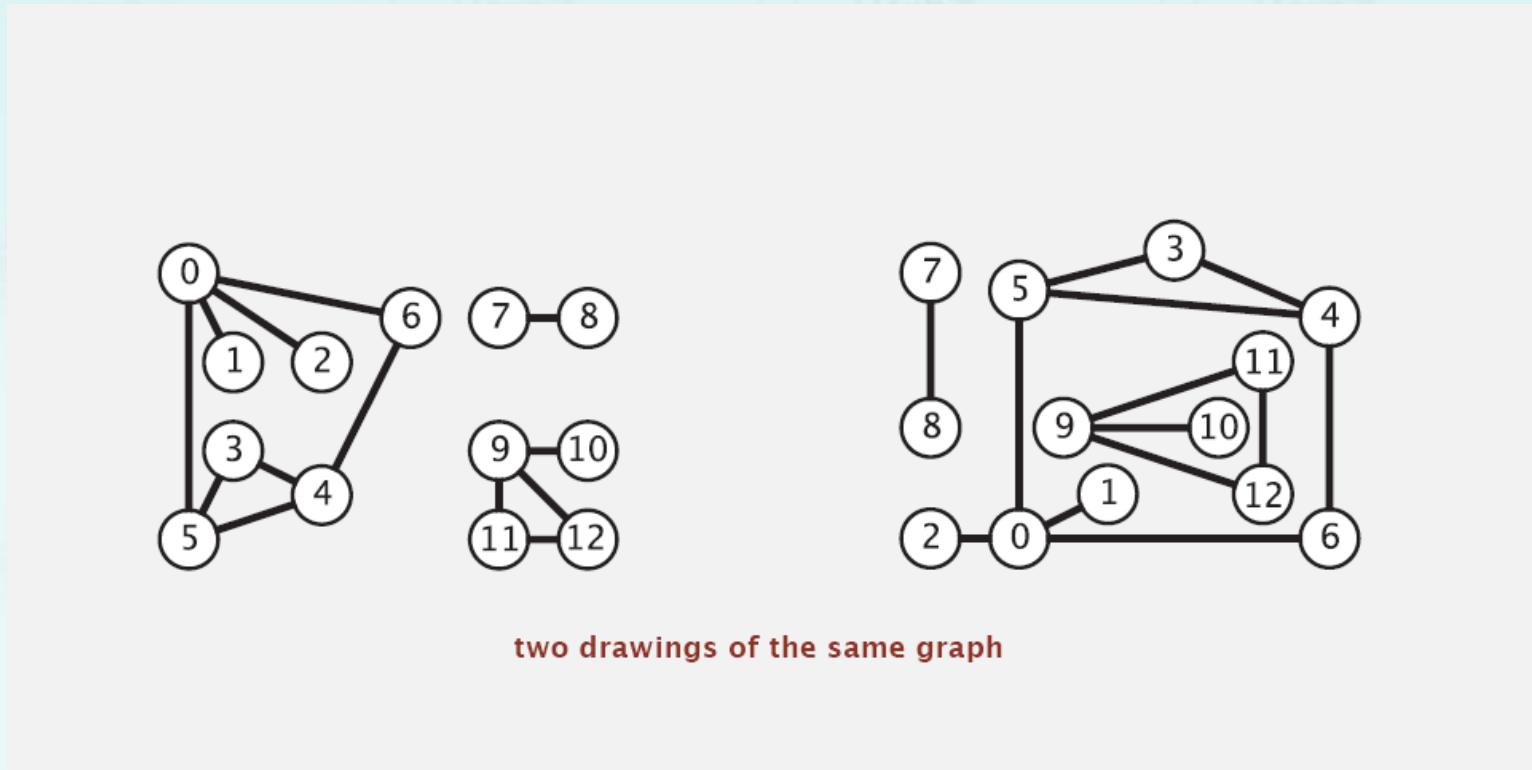
- Introduction
- **Graph API**
- Elementary Graph Operations
  - DFS: Depth first search
  - BFS: Breadth first search
  - CC: Connected components

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4<sup>th</sup> edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

# Graph Representation

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

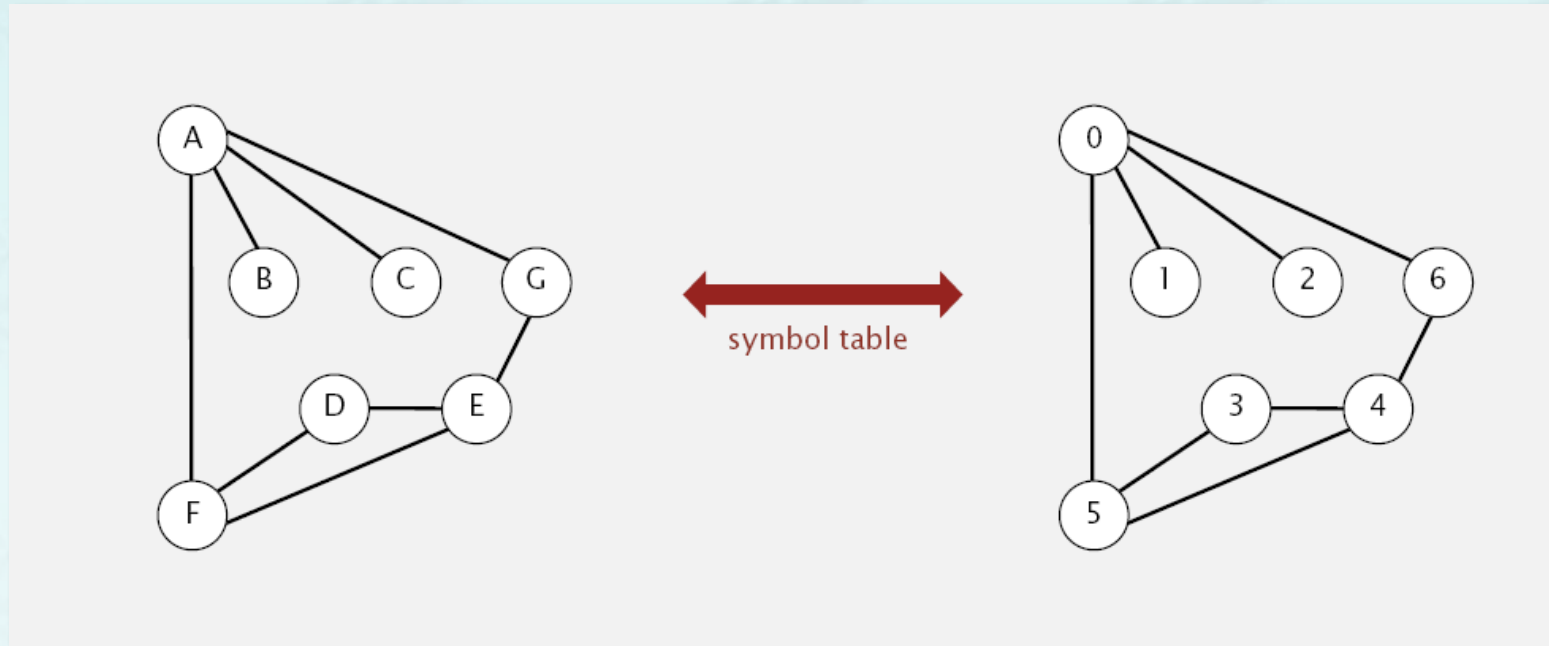




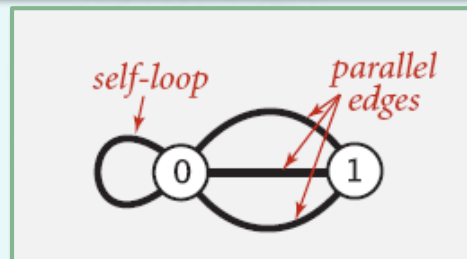
# Graph Representation

## Vertex representation.

- We use integers between **0** and  **$V - 1$** .
- Applications: convert between names and integers with symbol table.



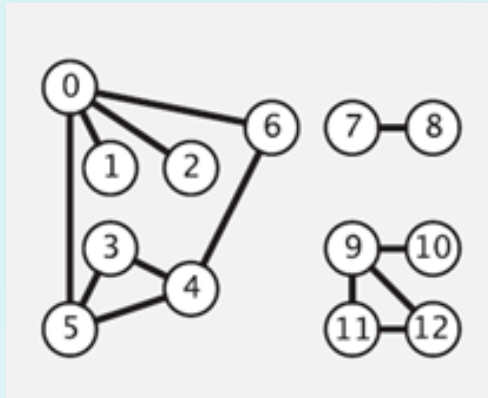
## Anomalies.



## Graph ADT in Java

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

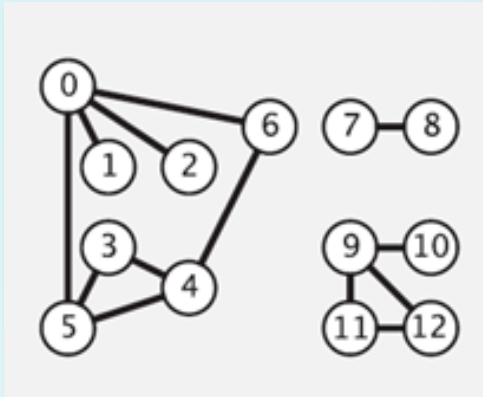
# Graph Input Format



⇒ graph3.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



# Graph Input Format



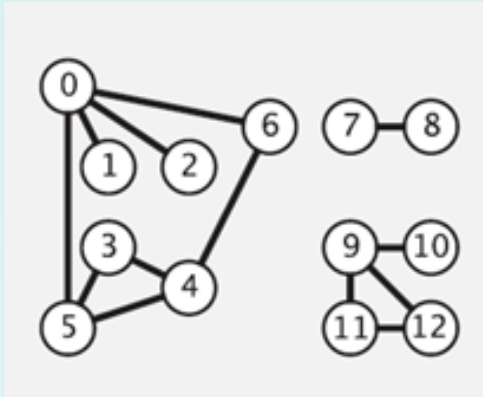
⇒ graph3.txt V  
13  
13 E  
0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3

graph.cpp

```
Graph g = graph_by_file(argv[1]);           read graph from input stream

for (int v = 0; v < V(g); ++v) {           print out edge list by vertices
    cout << "V[" << v << "]: ";
    for (gnode w = g->adj[v].next; w; w = w->next) {
        cout << w->item << " ";
        (w->next == nullptr) ? (cout << endl) : (cout << "-> ");
    }
}
```

# Graph Input Format



graph3.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

graph.cpp

```
Graph g = graph_by_file(argv[1]);
```

```
for (int v = 0; v < V(g); ++v) {
    cout << "V[" << v << "]: ";
    for (gnode w = g->adj[v].next; w; w = w->next) {
        cout << w->item << " ";
        (w->next == nullptr) ? (cout << endl) : (cout << "-> ");
    }
}
```

```
C:\GitHub\nowicx\Debug\graph.exe

Adjacency-list:
V[0]: 6 -> 2 -> 1 -> 5
V[1]: 0
V[2]: 0
V[3]: 5 -> 4
V[4]: 5 -> 6 -> 3
V[5]: 3 -> 4 -> 0
V[6]: 0 -> 4
V[7]: 8
V[8]: 7
V[9]: 11 -> 10 -> 12
V[10]: 9
V[11]: 9 -> 12
V[12]: 11 -> 9

[0]-----[6]
  \-----[2]
   \-----[1]
    \-----[3]
     \-----[4]
      \-----[5]

[7]-----[8]
[9]-----[10]
  \-----[11]
   \-----[12]

vertex[0..12] = 0 1 2 3 4 5 6 7 8 9 10 11 12
color[0..12] = 0 0 0 0 0 0 0 0 0 0 0 0 0
```

## Graph Coding

### Compute the **degree** of V

```
int degree(graph g, int v) {  
    if (!validVertex(g, v)) return -1;  
    int deg = 0;  
    for (gnode w = g->adj[v].next; w; w = w->next, deg++);  
    return deg;  
}
```

### Compute maximum degree

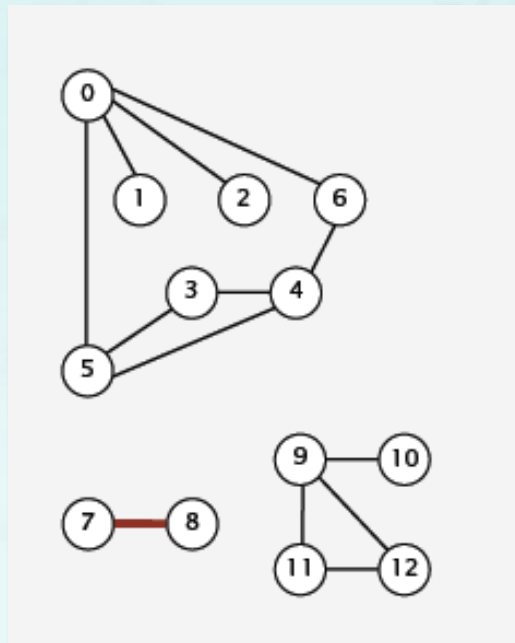
```
int degree(graph g) {  
    int max = 0;  
    for (int v = 0; v < V(g); ++v) {  
        int deg = degree(g, v);  
        if (deg > max) max = deg;  
    }  
    return max;  
}
```

### Compute average degree

```
double degree_average(graph g) {  
    int return 2.0 * E(g) / V(g);  
}
```



## Graph Coding – edge list

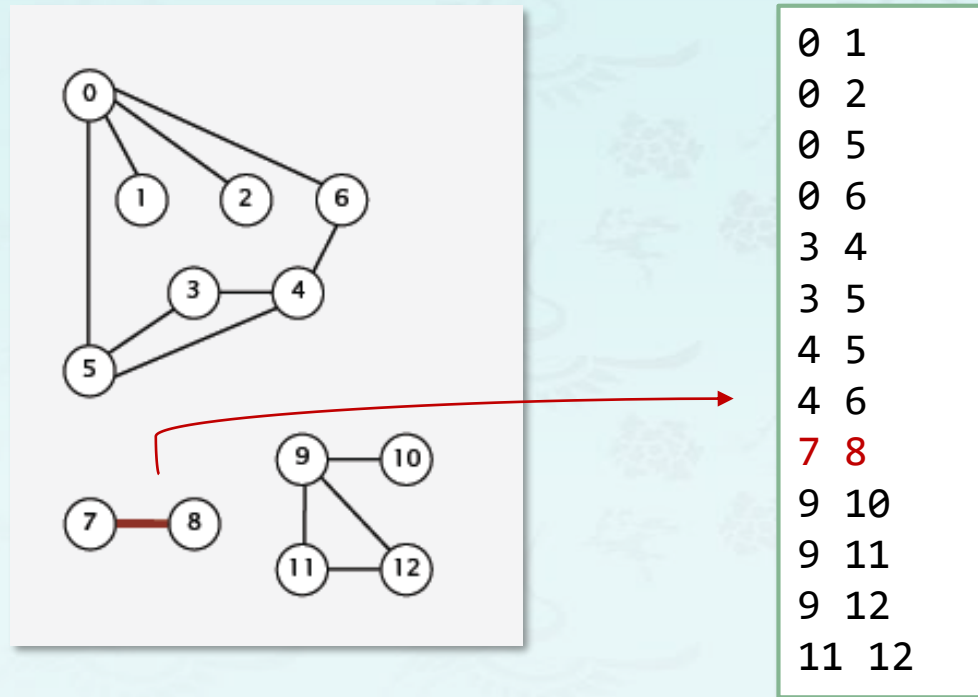


graph3.txt V  
13  
13 ← E  
0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3

1. Edge list
2. Adjacency matrix
3. Adjacency list

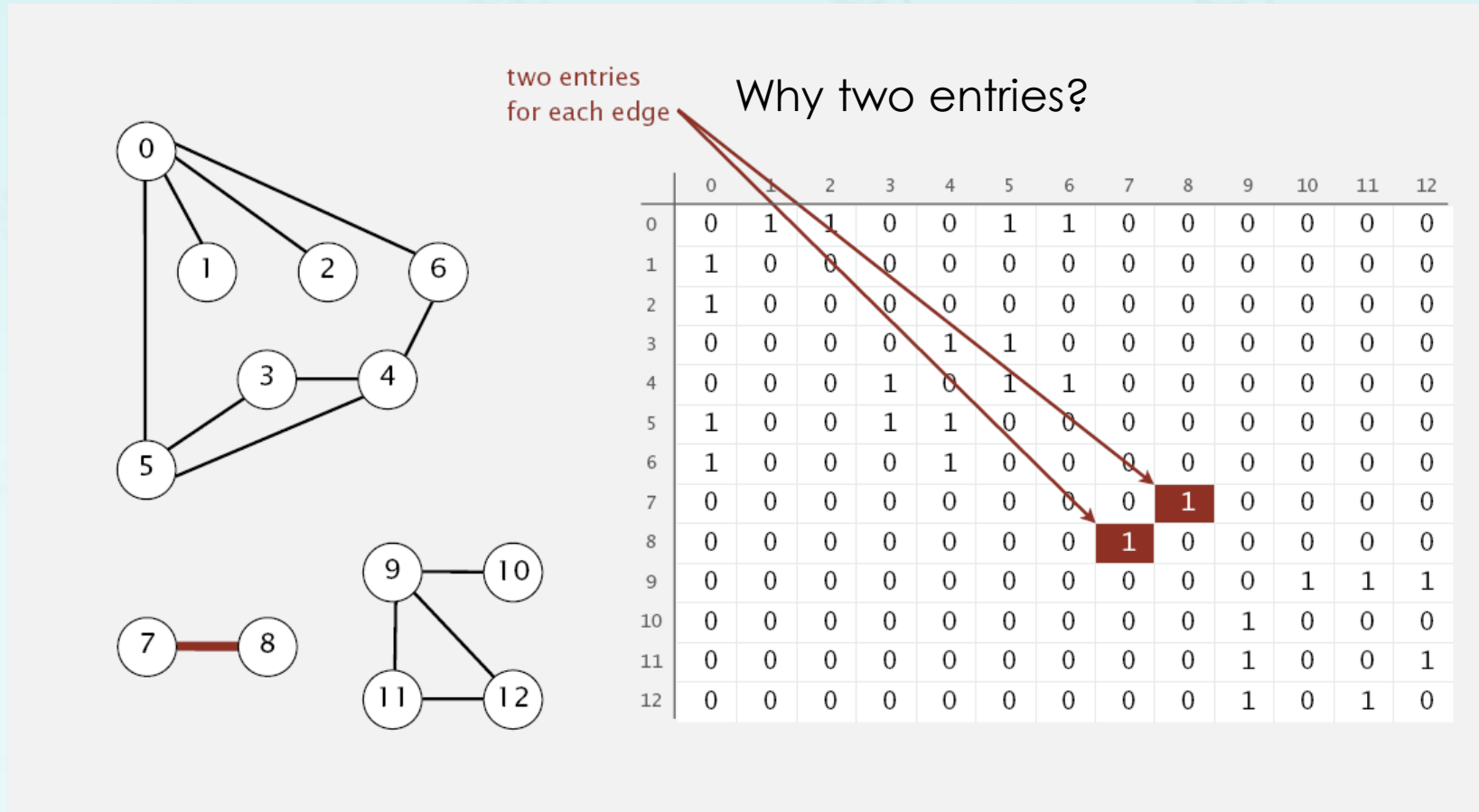
# Graph Coding – edge list

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



## Graph Coding – Adjacency-matrix 인접행렬

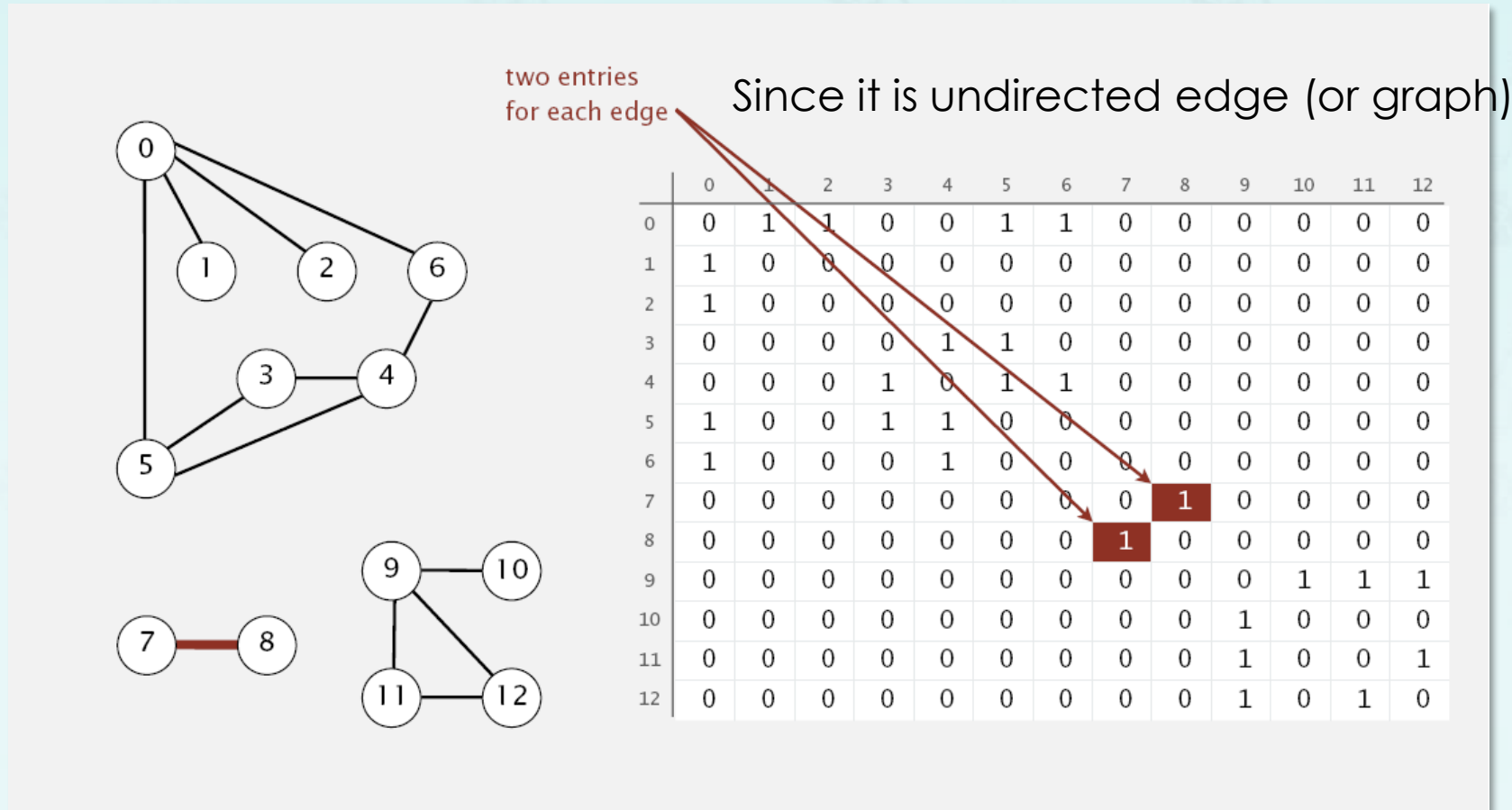
2. Maintain a two-dimensional V-by-V Boolean array;  
for each edge v-w in graph:  $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$ .





## Graph Coding – Adjacency-matrix 인접행렬

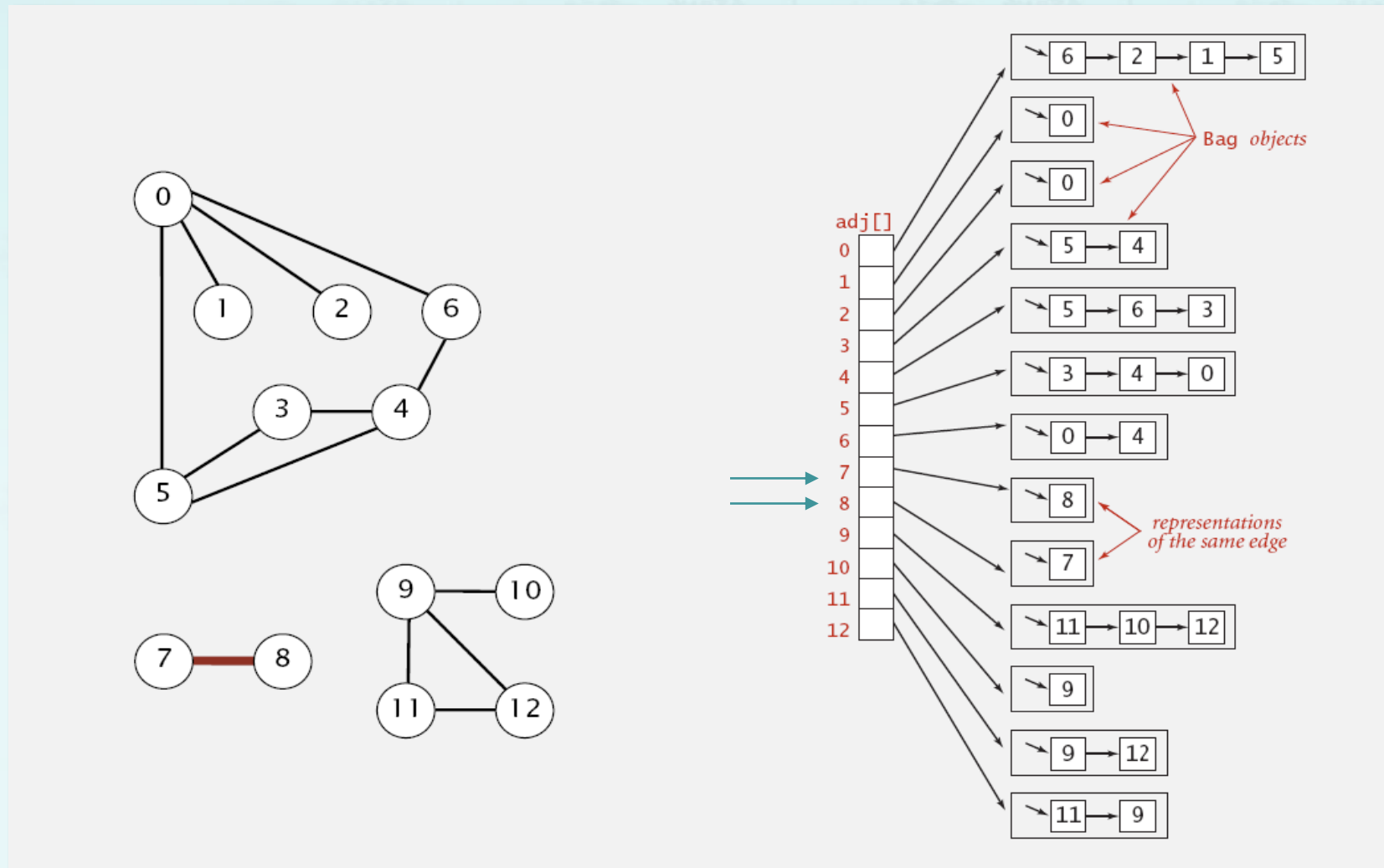
2. Maintain a two-dimensional V-by-V Boolean array;  
for each edge v-w in graph:  $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$ .



## Graph Coding – Adjacency list 인접리스트

3. Maintain vertex-index array of lists.

use Bag in Java.  
use a linked list in C.




## Graph Coding – graph.h


```
// a structure to represent an adjacency list node
struct Gnode {
    int    item;
    Gnode* next;
    Gnode (int i = 0, Gnode *p = nullptr) {
        item = i;  next = p;
    }
    ~Gnode() {}
};

using gnode = Gnode *;
```

adjacency list nodes (using a linked list)



## Graph Coding – graph.cpp

```
struct Graph {  
    int V;           // number of vertices in the graph  
    int E;           // number of edges in the graph  
    gnode adj;       // an array of adjacency lists (or gnode pointers)  
  
    Graph(int v = 0) { // constructs a graph with v vertices  
        V = v;  
        E = 0;  
  
        // initialize each adjacency list as an empty list;  
        for (int i = 0; i < V; i++) {  
             ← set each adj list nullptr  
            ← unused; but may store the size of degree.  
        }  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```



## Graph Coding – graph.cpp

```
struct Graph {  
    int V;           // number of vertices in the graph  
    int E;           // number of edges in the graph  
    gnode adj;       // an array of adjacency lists (or gnode pointers)  
  
    Graph(int v = 0) { // constructs a graph with v vertices  
        V = v;  
        E = 0;  
  
        // initialize each adjacency list as an empty list;  
        for (int i = 0; i < V; i++) {  
            g->adj[i].next = nullptr; ← set each adj list nullptr  
            g->adj[i].item = i;      unused; but may store the size of degree.  
        }  
    }  
    ~Graph() {}  
};  
using graph = Graph *;
```

// add an edge to an **undirected** graph

```
void addEdgeFromTo(graph g, int v, int w) {  
    // add an edge from v to w.  
    // A new vertex is added to the adjacency list of v.  
    // The vertex is added at the beginning
```

```
    gnode node = new Gnode(w);  
    g->adj[v].next = node;  
    g->E++;  
}
```

← instantiate a node w insert it  
at **the front of** adjacency list[v]

// add an edge to an undirected graph

```
void addEdge(graph g, int v, int w) {  
    addEdgeFromTo(g, v, w); // add an edge from v to w.  
    addEdgeFromTo(g, w, v); // if graph is undirected, add both  
}
```

← add an edge for undirected graph


## Graph Coding – Adjacency-matrix 인접행렬

---

**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

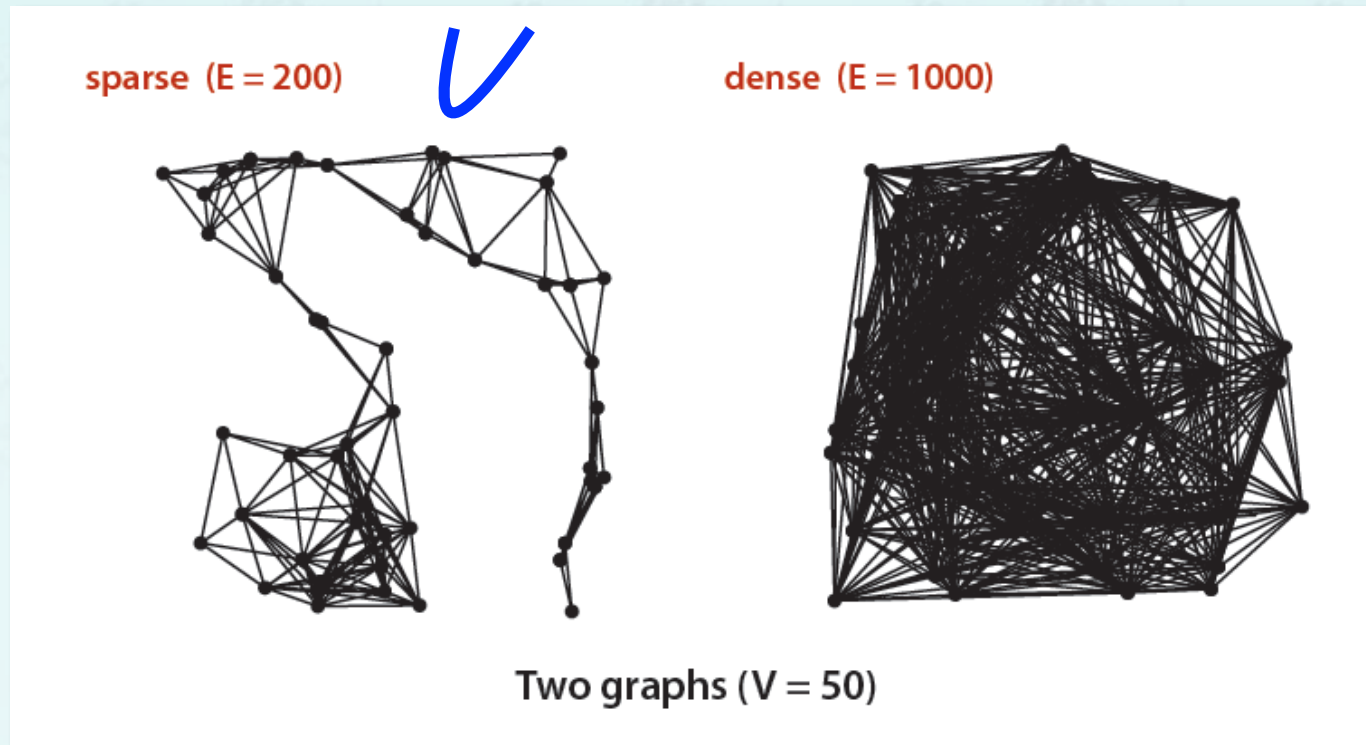


## Graph Coding – Adjacency-matrix 인접행렬

**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





## Graph Coding – Adjacency-matrix 인접행렬

**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	<div></div>	
adjacency lists	$E + V$	1		

## Graph Coding – Adjacency-matrix 인접행렬

**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	$V$
<u>adjacency lists</u>	$E + V$	1	$degree(v)$	$degree(v)$

# Graph

---

- **Introduction**
- **Graph API**
- Elementary Graph Operations
  - DFS: Depth first search
  - BFS: Breadth first search
  - CC: Connected components

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4<sup>th</sup> edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet