UMBC

AN HONORS UNIVERSITY IN MARYLAND

IS 709/809: Computational Methods in IS Research

Graph Algorithms: Introduction

Nirmalya Roy

Department of Information Systems

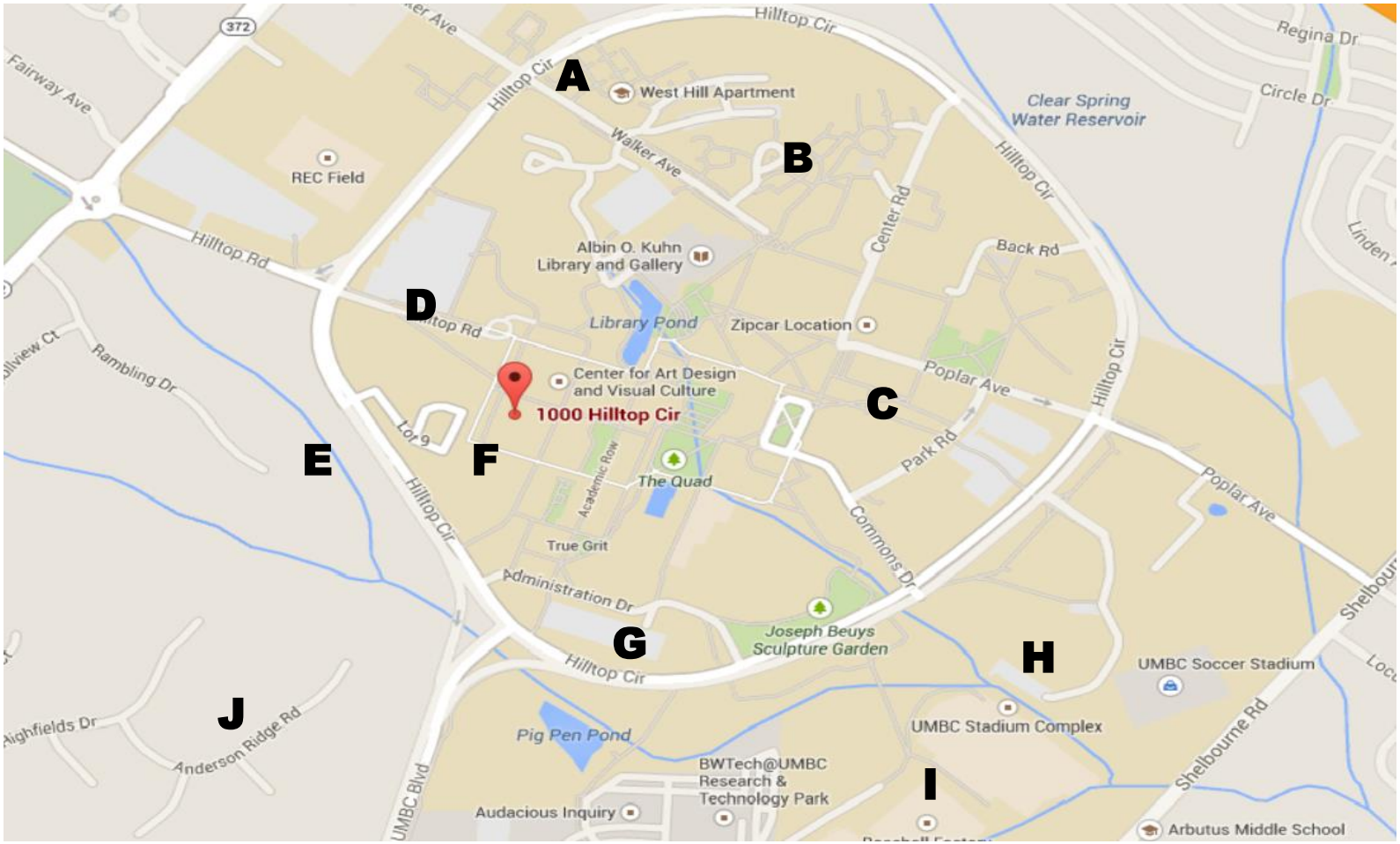
University of Maryland Baltimore County

Motivation

- Several real-life problems can be converted to problems on graphs
- Graphs are one of the pervasive data structures used in computer science
- Graphs are a useful tool for modeling real-world problems
- Graphs allow us to abstract details and focus on the problem
- We can represent our domain as graphs and apply graph algorithms to solve our problem

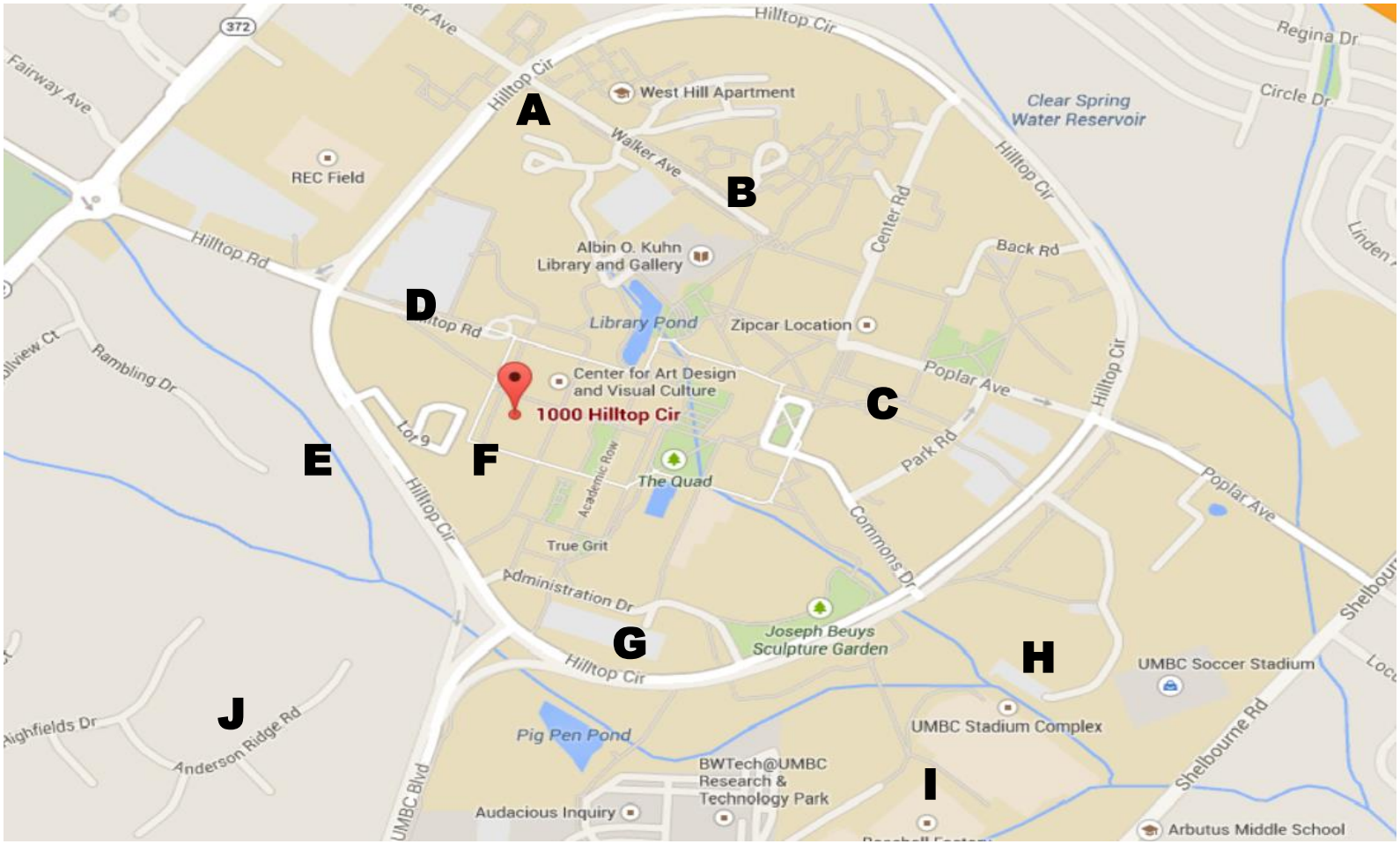
Examples

- Given a map of UMBC and the surrounding area, how to get from one place to another?



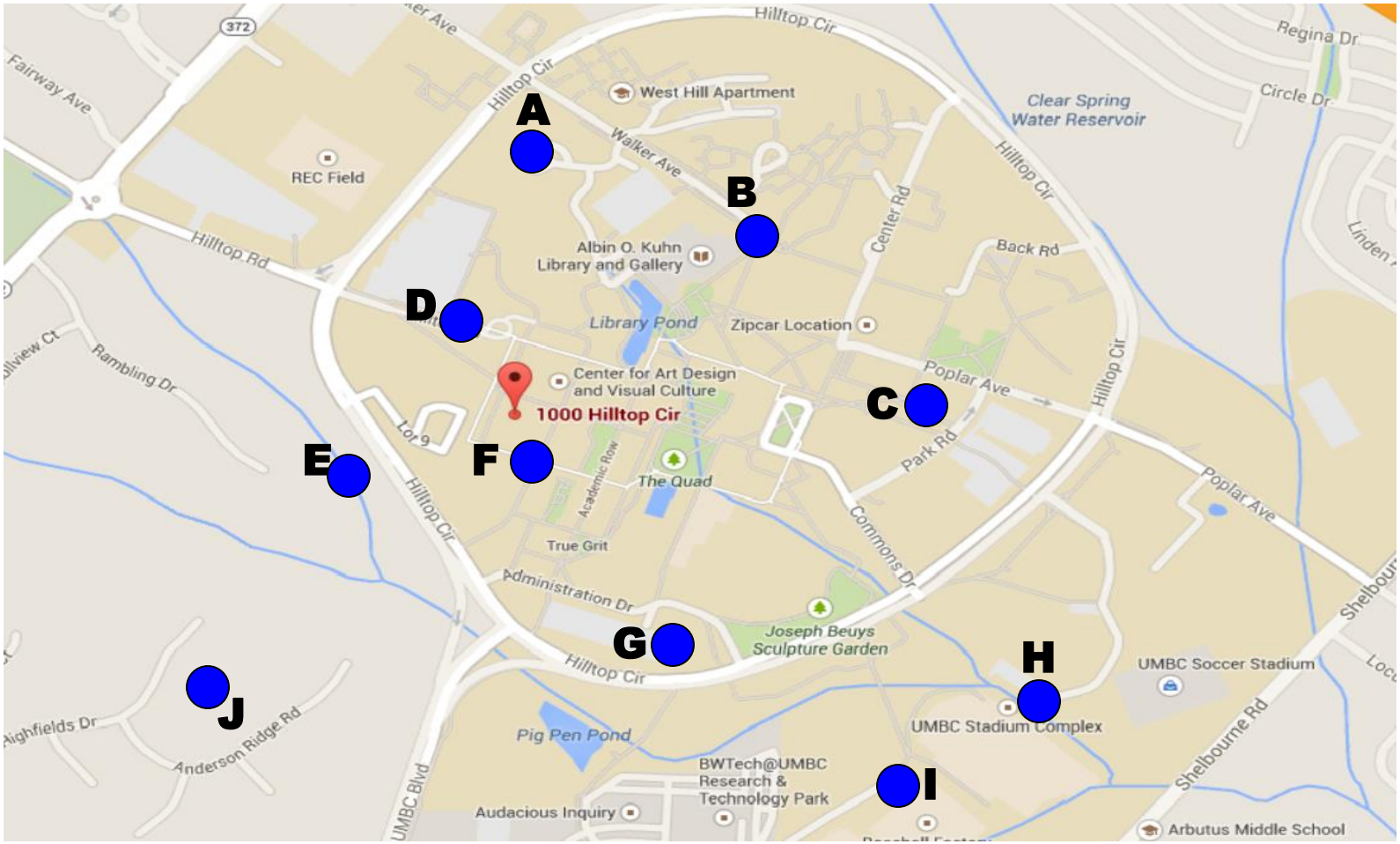
Examples (cont'd)

- What data structure to use to represent the problem?
- How do you even think about the problem?



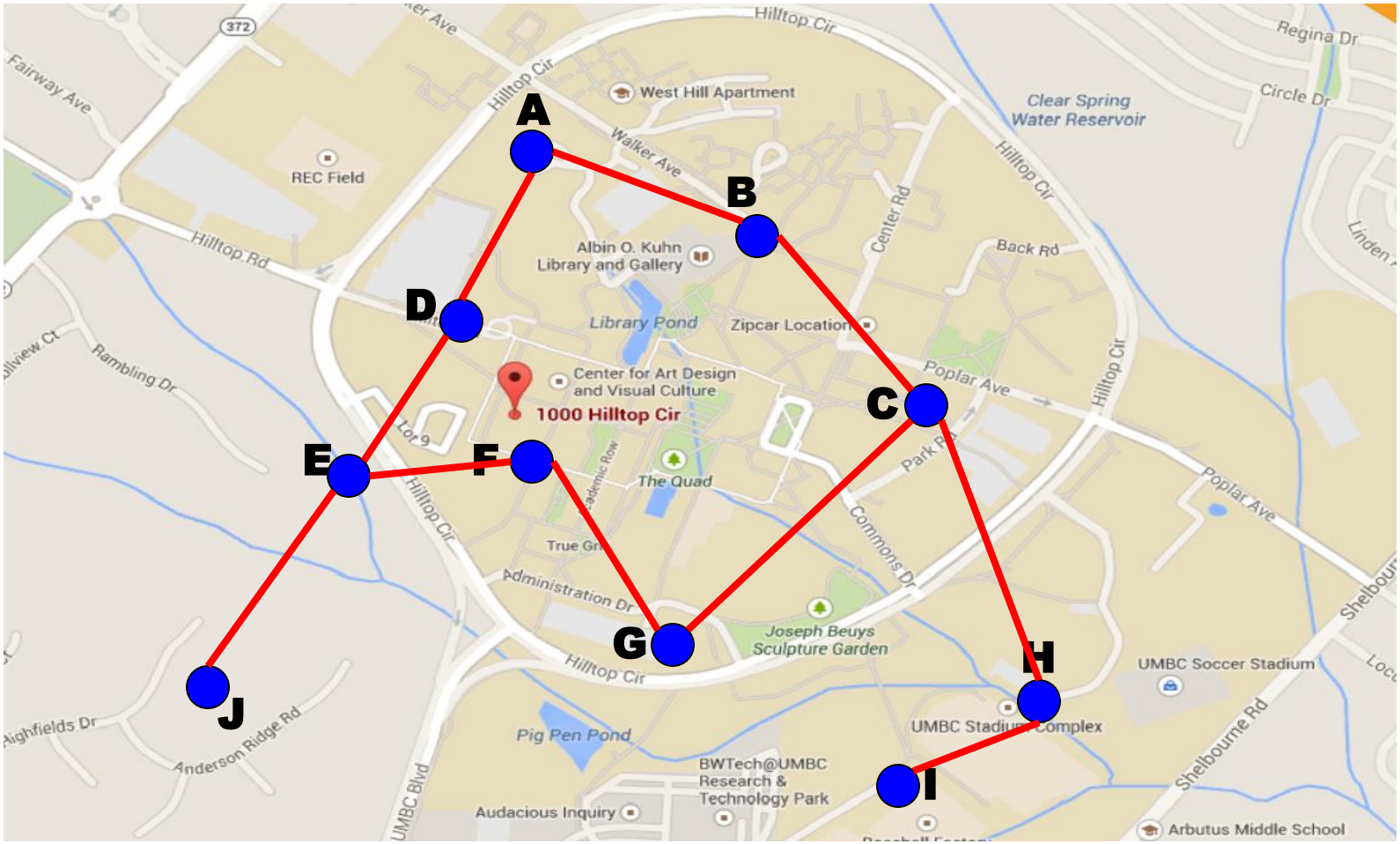
Examples (cont'd)

- Let us strip away irrelevant details
- We have a set of vertices {A, B, C, D, E, F, G, H, I, J}



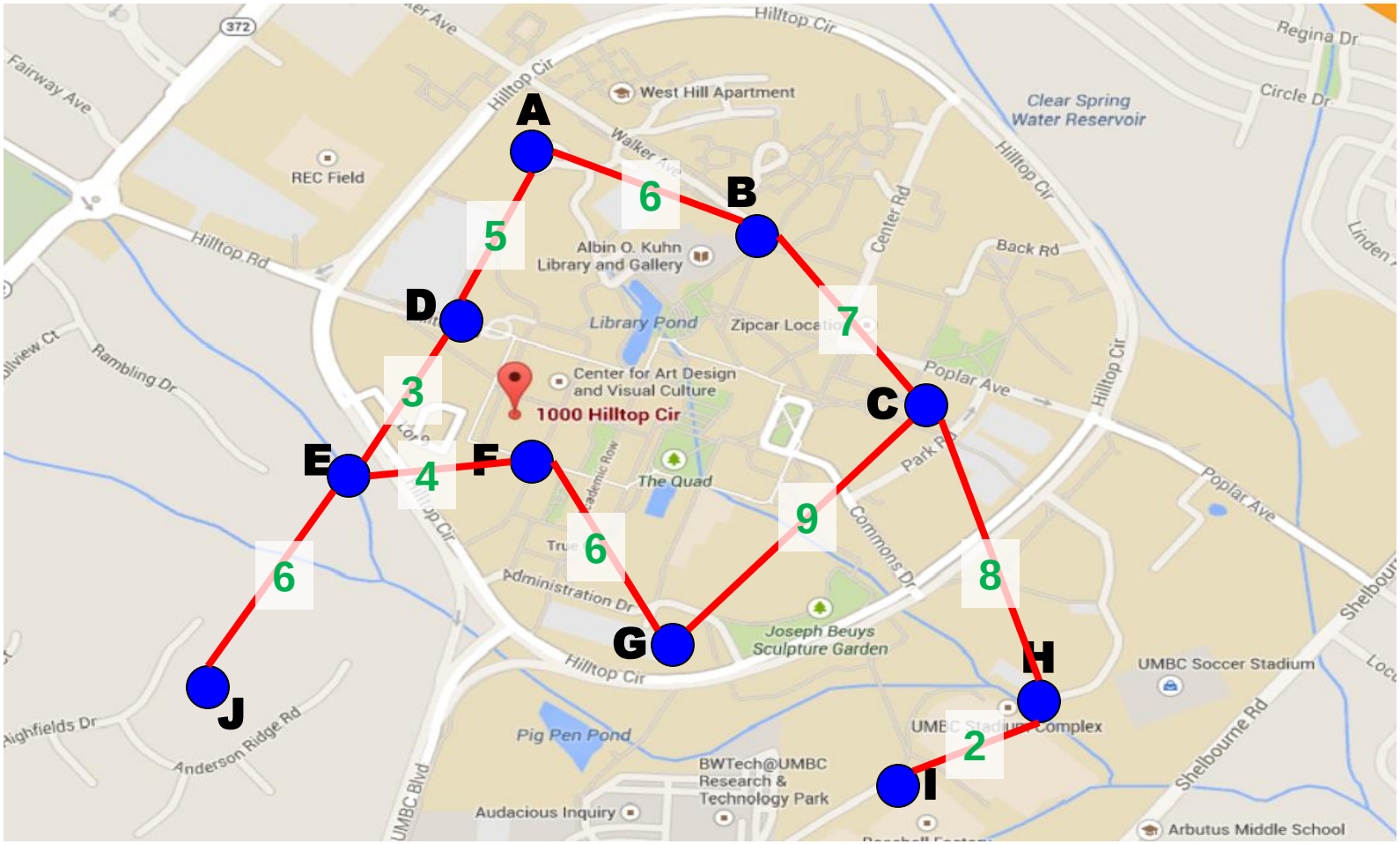
Examples (cont'd)

- Let us strip away irrelevant details
- We have a set of edges connecting the vertices



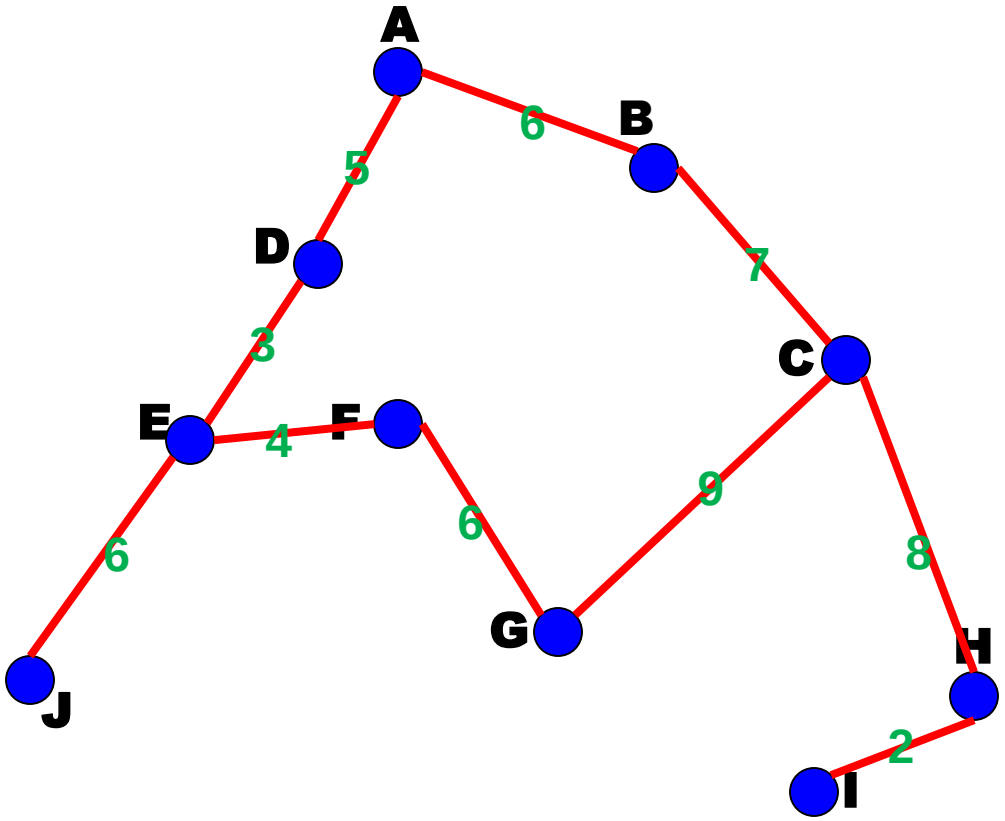
Examples (cont'd)

- Let us strip away irrelevant details
- Edges can be assigned weights

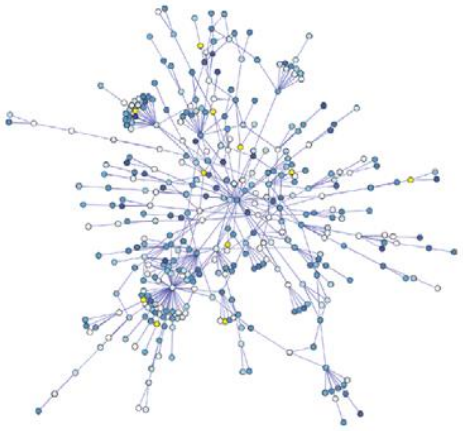


Examples (cont'd)

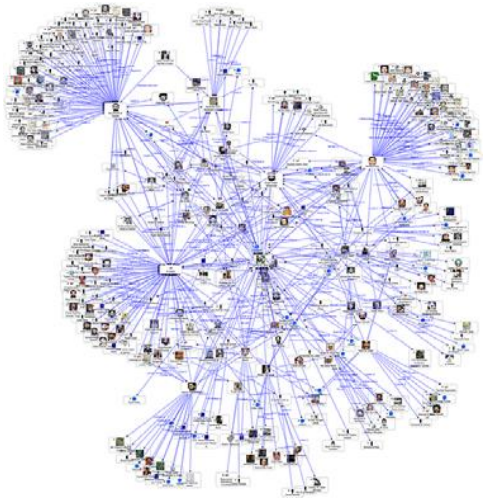
- Let us strip away irrelevant details



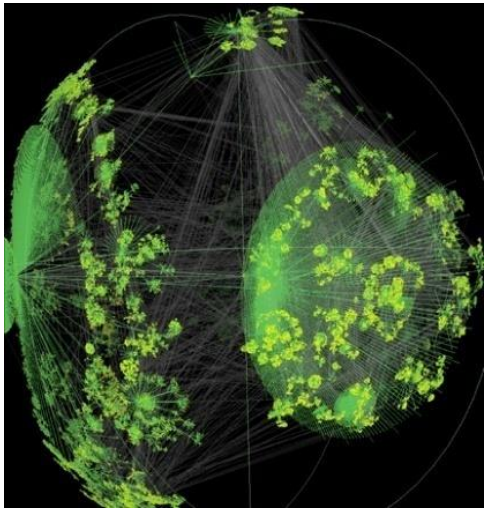
Other Examples



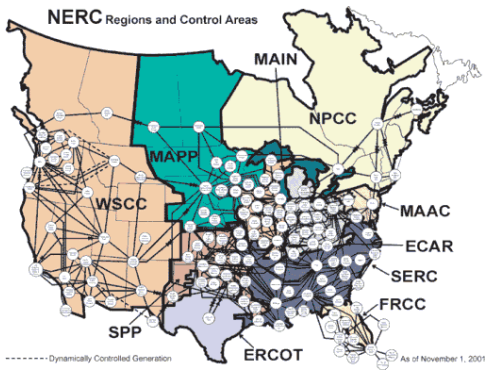
Protein-protein interaction



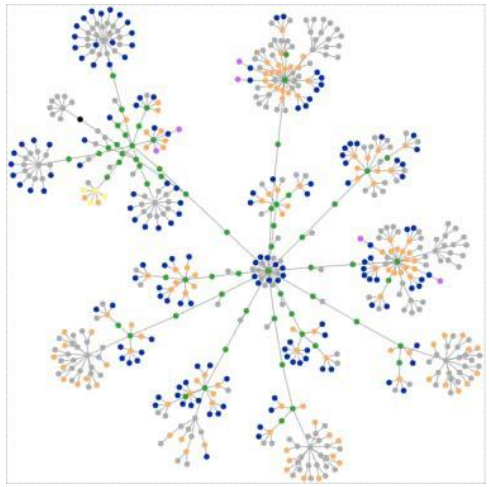
Social network



Internet

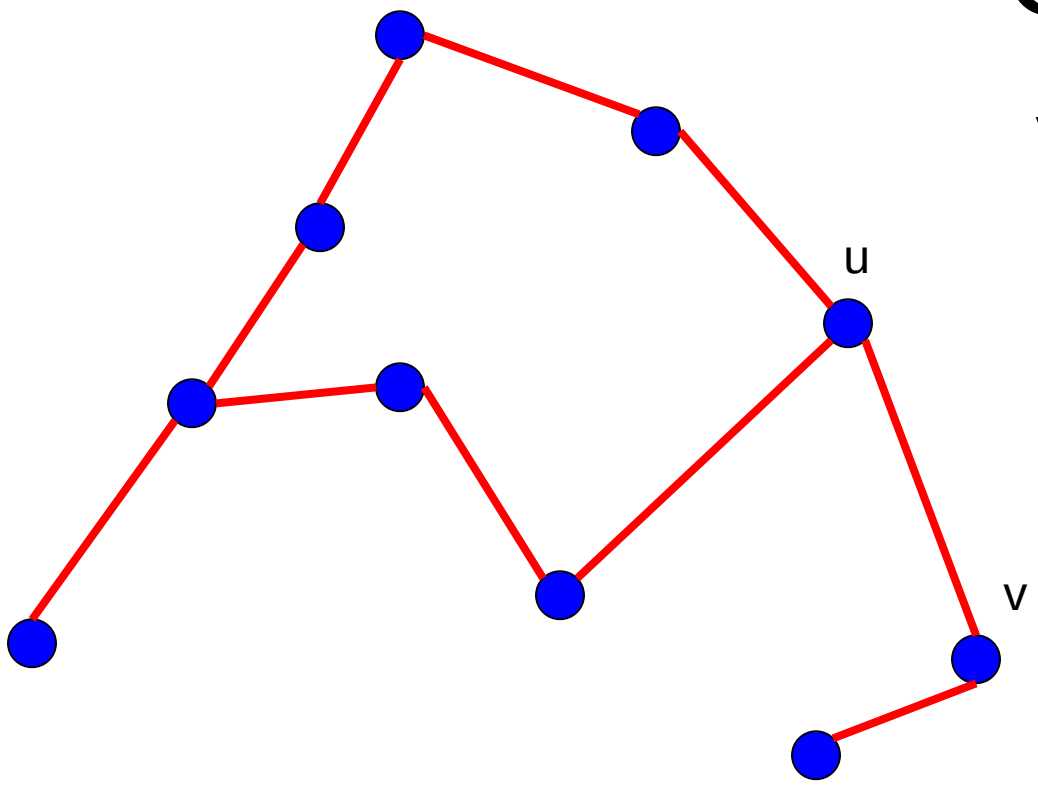


Power grid



WWW

Simple Graphs

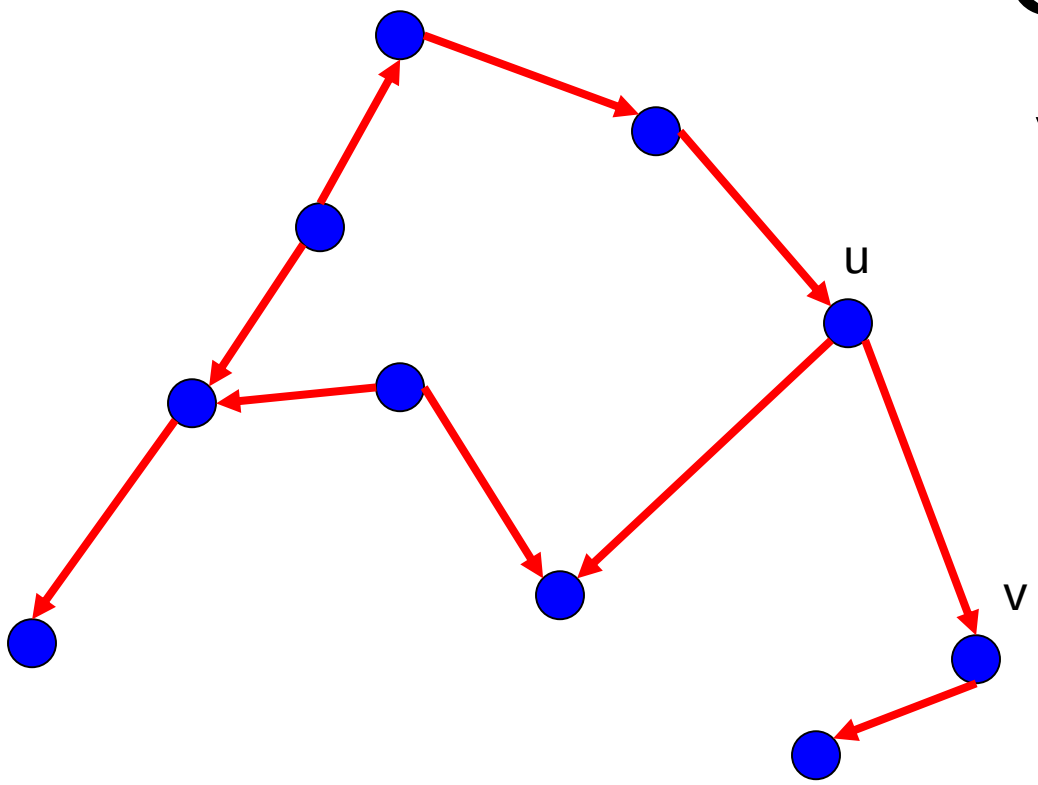


$G = (V, E)$

V is a set of vertices.
E is a set of edges.

$u \in V, v \in V$
$(u, v) \in E$

Directed Graphs

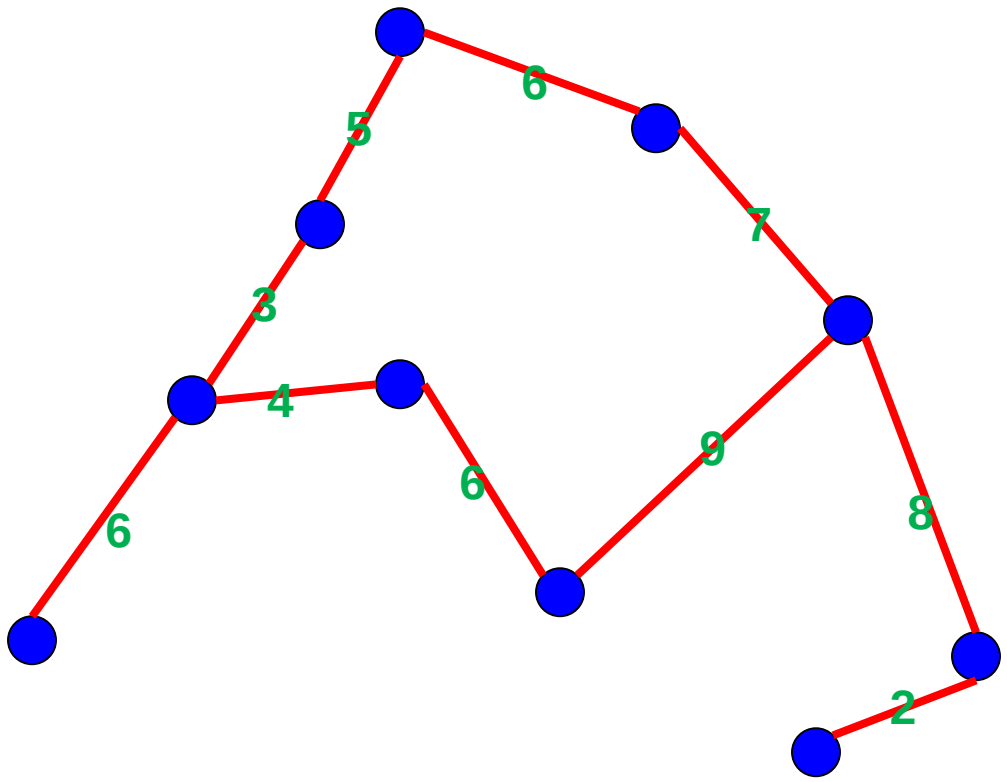


$G = (V, E)$

V is a set of vertices.
E is a set of edges.

$u \in V, v \in V$
$(u, v) \in E$
$(v, u) \notin E$

Weighted Graphs



$$G = (V, E)$$

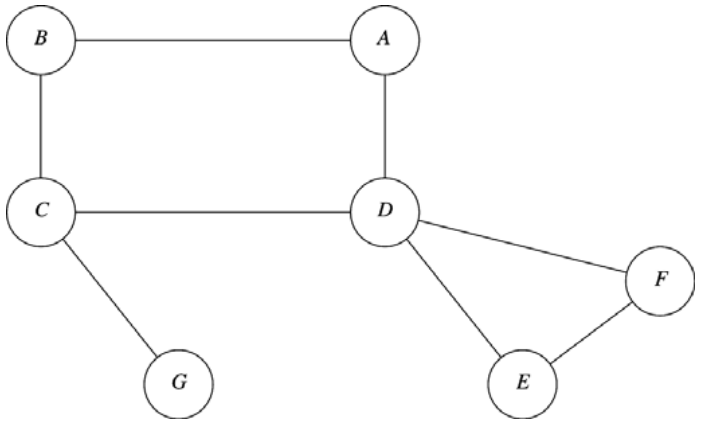
V is a set of vertices
E is a set of edges

Cardinality of a Set

- "The number of elements in a set"
- Let A be a finite set
  - If A = $\varnothing$ (the empty set), then the cardinality of A is 0
  - If A has exactly n elements, n a natural number, then the cardinality of A is n
- The cardinality of a set A is denoted by |A|

Definition of a Graph

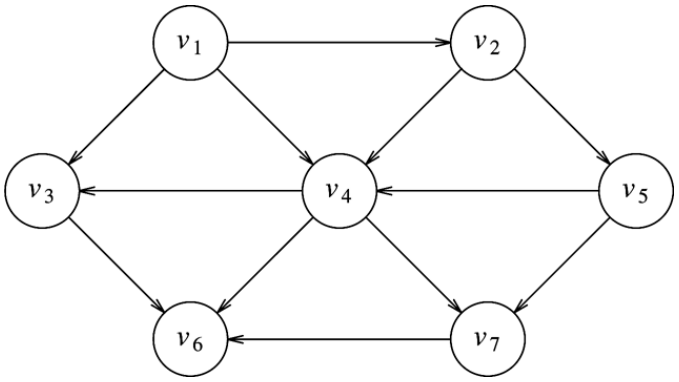
- A graph G = (V, E) consists of a set of vertices V and a set of edges E
- E = {(u, v} | u, v ∈ V}
  - Vertex v is adjacent to vertex u
  - Edges are sometimes called arcs

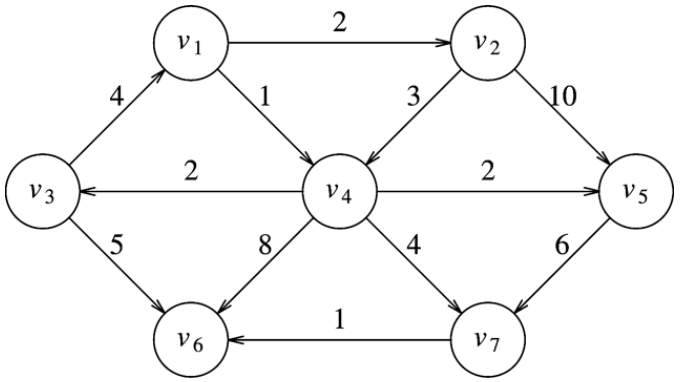


- Example
  - V = {A, B, C, D, E, F, G}
  - E = {(A, B), (A, D), (B, C), (C, D), (C, G), (D, E), (D, F), (E, F)}
- Cardinality of Vertex Set denoted by |V|
  - |V| = 7 = number of vertices in set V
- Cardinality of Edge Set denoted by |E|
  - |E| = 8 = number of edges in set E

Definitions

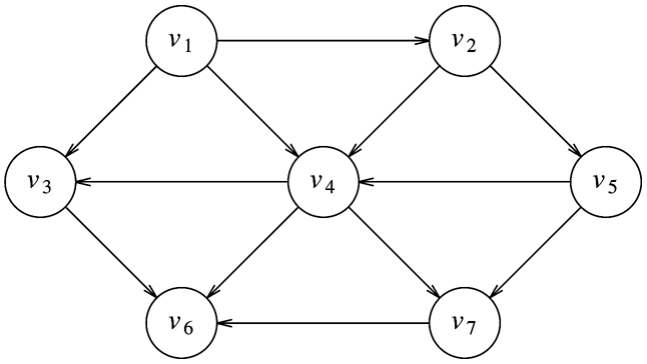
- Undirected graphs
  - Edges are unordered (i.e. (u, v) is the same as (v, u))
- Directed graphs (digraphs)
  - Edges are ordered (i.e. <u, v> ≠ <v, u>)
- Weighted graphs
  - Edges have a weight w(u, v) or cost (u, v)





Definitions (cont'd)

- Degree of a vertex
  - Number of edges incident on a vertex
- Indegree
  - Number of directed edges to vertex
- Outdegree
  - Number of directed edges from vertex



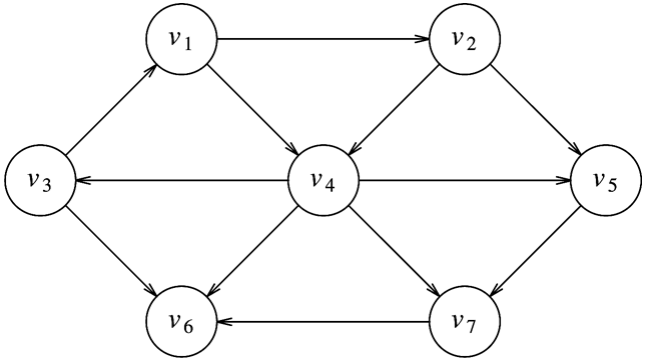
degree(v4) = 6
indegree(v4) = 3
outdegree(v4) = 3

indegree(v1) = 0
outdegree(v1) = 3

indegree(v6) = 3
outdegree(v6) = 0

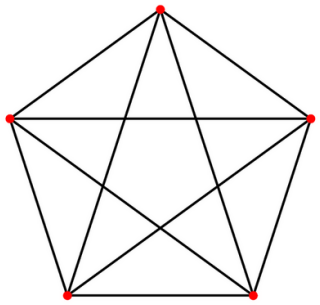
Definitions (cont'd)

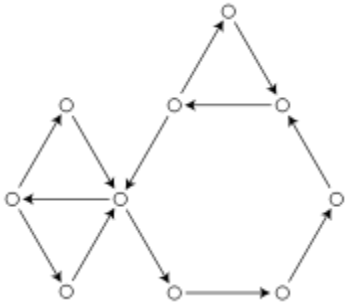
- Path
  - Sequence of vertices $v_1, v_2, \ldots, v_N$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq N$
  - Path length is the number of edges on the path (i.e., N – 1)
  - Simple path has unique intermediate vertices
- Cycle
  - Path where $v_1 = v_N$
  - Usually simple and directed
  - Acyclic graphs have no cycles

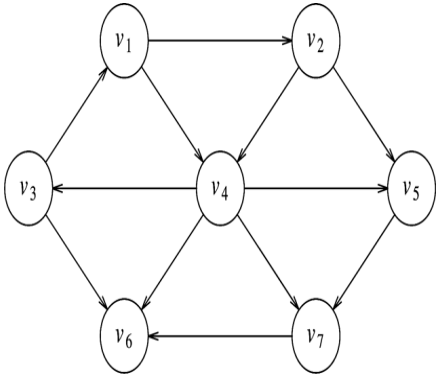


Definitions (cont'd)

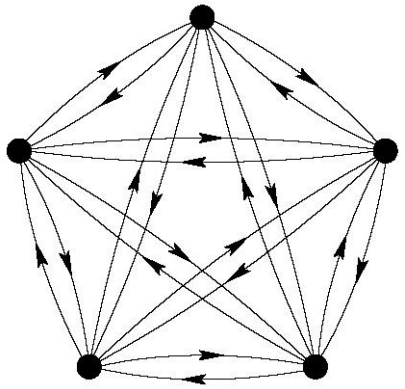
- Undirected graph is connected if there is a path between every pair of vertices
- Connected, directed graph is called strongly connected
- Complete graph has an edge between every two vertices





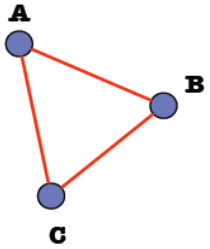


v_6 to v_1, v_7 to v_1



Representations

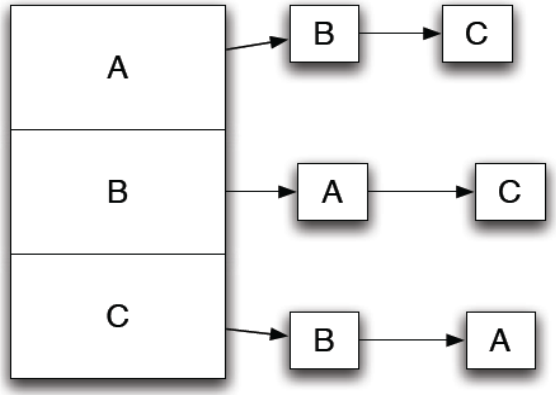
- Adjacency matrix is a two dimensional array
  - For each edge (u,v), A[u][v] is `true` otherwise it is `false`



	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Adjacency Matrix

$|V^2|$

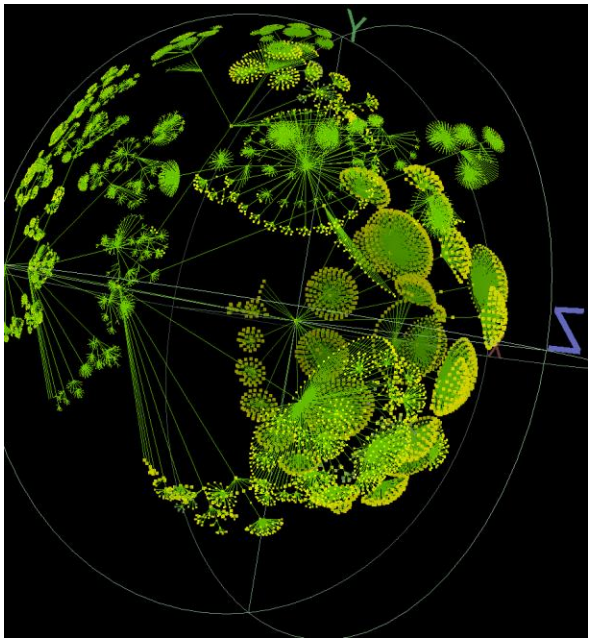


Adjacency List

$|V| + |E|$

Representations (cont'd)

- It is good to use adjacency lists for sparse graphs
- Sparse means not dense
- Graph is dense means $|E| = \Theta(|V|^2)$



Practical Problem Representations

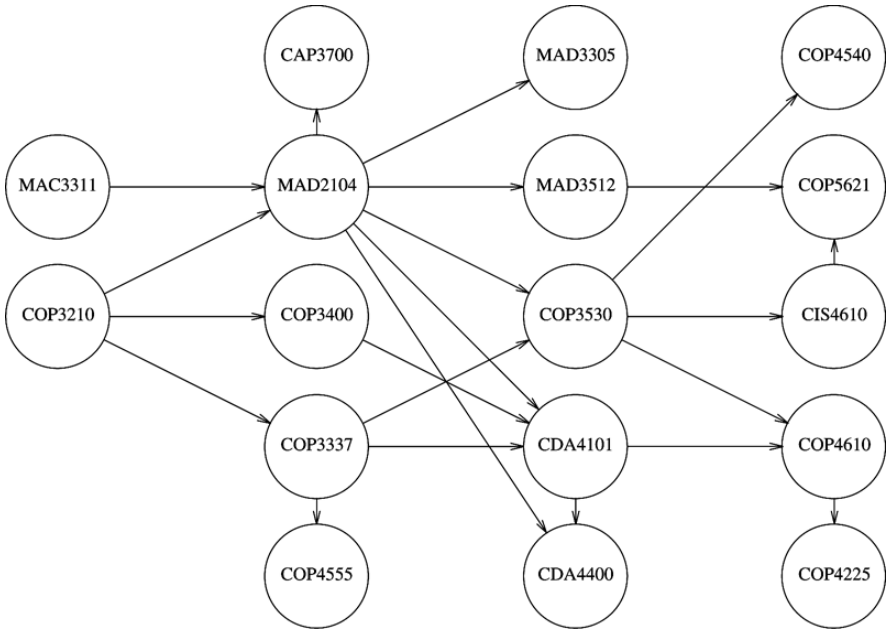
- Graph represents a street map with Manhattan-like orientation
  - Streets run mainly on north-south or east-west
  - Any intersection is attached with four streets
  - Graph is directed and all streets are two way, then $|E| = 4|V|$
  - Example: Assume 3000 intersections = 3000-vertex graph and 12000 edges
    - Array size for **Adjacency Matrix** = 9,000,000
    - Most of these entries would contain 0
- If the graph is **sparse** a better solution is **adjacency list**
  - For each vertex we keep a list of all adjacent vertices
  - Space requirement is $O(|E|+|V|)$, linear in the size of the graph

Graph Algorithms:

Topological Sort

Topological Sort

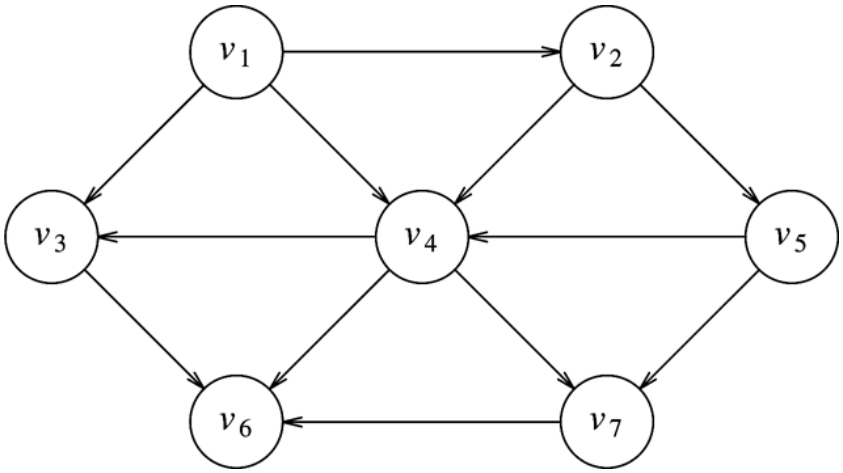
- Order the vertices in a directed acyclic graph (DAG), such that if (u, v) $\in$ E, then u appears before v in the ordering
- Example



Course Prerequisite Structure at a University

Topological Sort (cont'd)

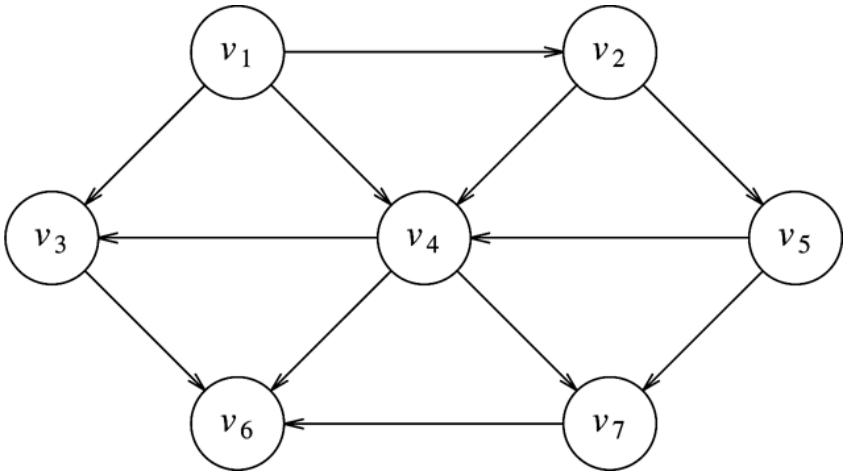
- Topological ordering is not possible if the graph has a cycle, since for two vertices u and v on the cycle, u precedes v and v precedes u
- The ordering is not necessarily unique; any legal ordering will do



Possible topological orderings: $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$.

Topological Sort (cont'd)

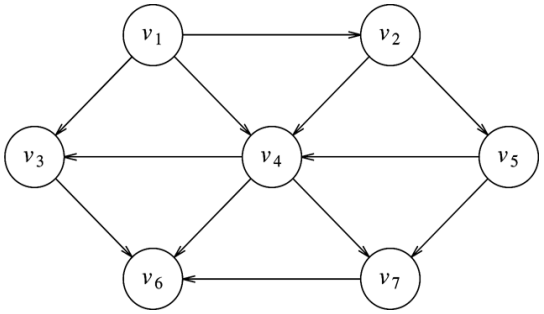
- Solution #1
  - While there are vertices left in the graph
    - Find vertex v with indegree equals to 0
    - Output v
    - Remove v from the graph together with all edges to and from v
- Running of Solution #1 is $O(|V|^2)$



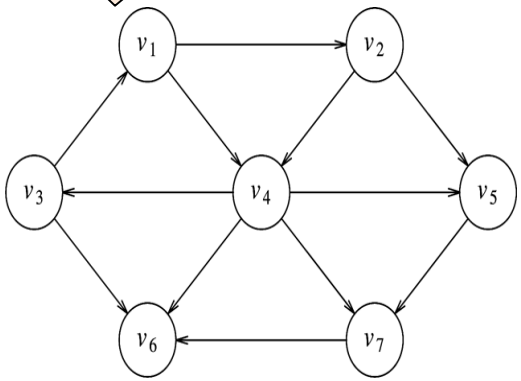
Possible topological ordering:
$v_1, v_2, v_5, v_4, v_3, v_7, v_6$.

Topological Sort Pseudocode

```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );//Not been assigned a topological number
        if( v == NOT_A_VERTEX )                   Takes O(|V|) times, |V| such calls
            throw CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```



CycleFoundException()

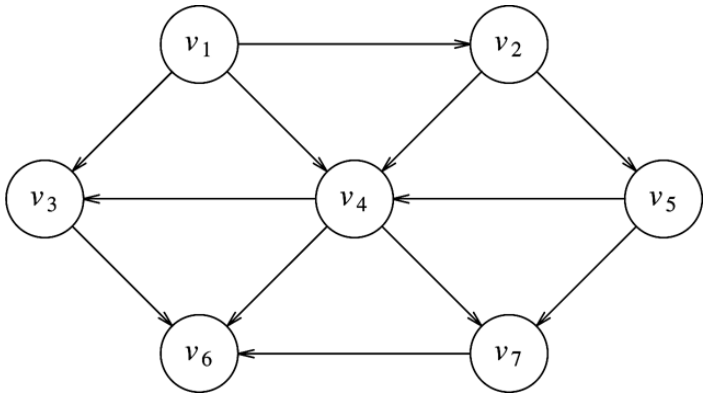


Topological Sort (cont'd)

- Solution #2
  - Don't need to search over all vertices for indegree = 0
  - Only vertices that lost an edge from the previous vertex's removal need to be searched
  - Algorithm
    - Compute the indegree for every vertex
    - Place all vertices of indegree 0 to an initially empty queue (note: we can also use a stack)
    - While the queue is not empty
      - Remove a vertex v from the queue
      - Output v
      - Decrement indegrees of all vertices adjacent to v
      - Put a vertex on the queue as soon as its indegree falls to 0

Topological Sort (cont'd)

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

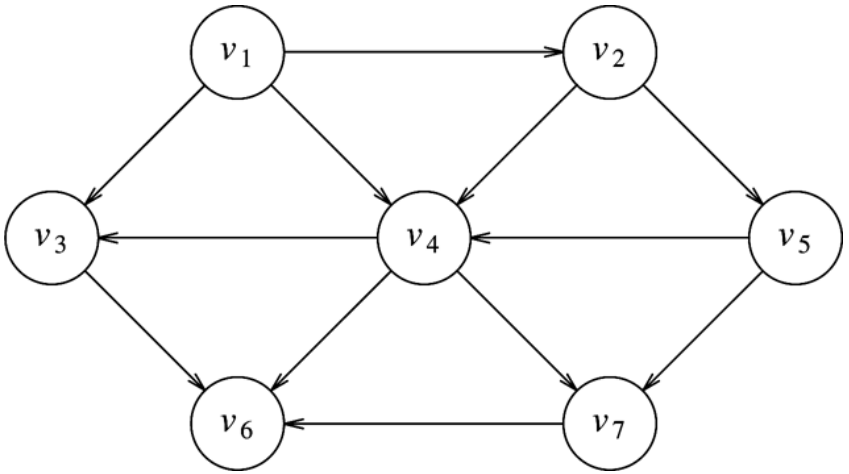


Topological Sort Pseudocode

```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException( );
}
```

Topological Sort (cont'd)

- Solution #2
  - Assume that the graph is already read into an adjacency list
  - Assume the indegrees are computed and stored with the vertices
- Running time of Solution #2 is O(|V| + |E|)



Possible topological ordering:
v_1, v_2, v_5, v_4, v_3, v_7, v_6.

Graph Algorithms

- Topological sort
- Shortest paths
- Network flow
- Minimum spanning tree
- Applications