CSCE 3110
Data Structures & Algorithms

- Graphs
- Reading: Chap. 9, Weiss

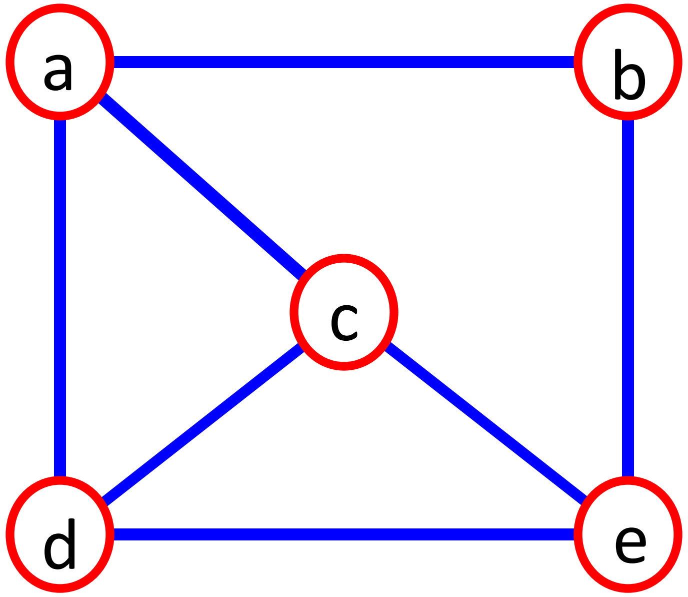
Contents

Graph Definition

- A graph $G = (V, E)$ is composed of:

  V: set of vertices (nodes)

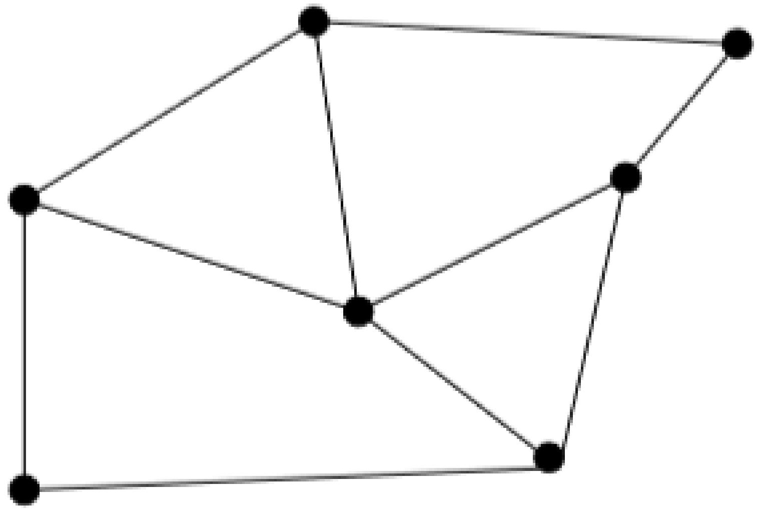
  E: set of edges (arcs) connecting the vertices in V
- An edge $e = (u, v)$ is a pair of vertices
- Example:

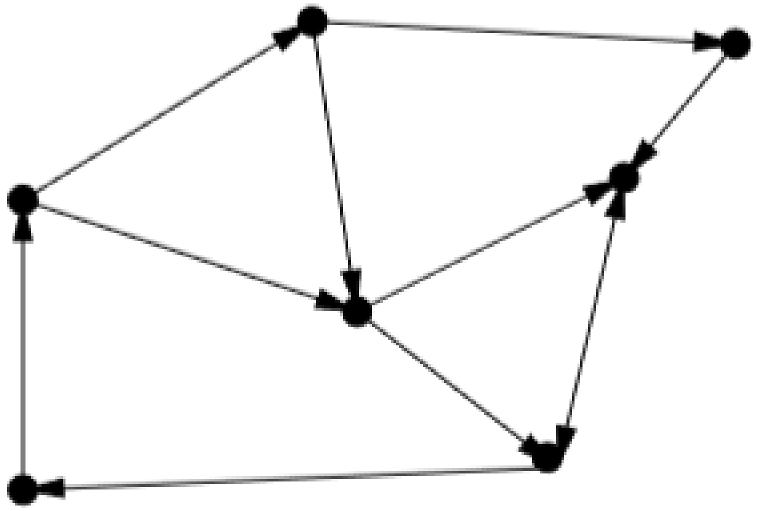


$$V = \{a, b, c, d, e\}$$

$$E = \begin{Bmatrix} (a,b),(a,c),(a,d), \\ (b,e),(c,d),(c,d),(d,e) \end{Bmatrix}$$

Undirected vs. Directed Graph (digraph)





Undirected Graph
– edge is bidirectional

Directed Graph
– edge has oriented vertex

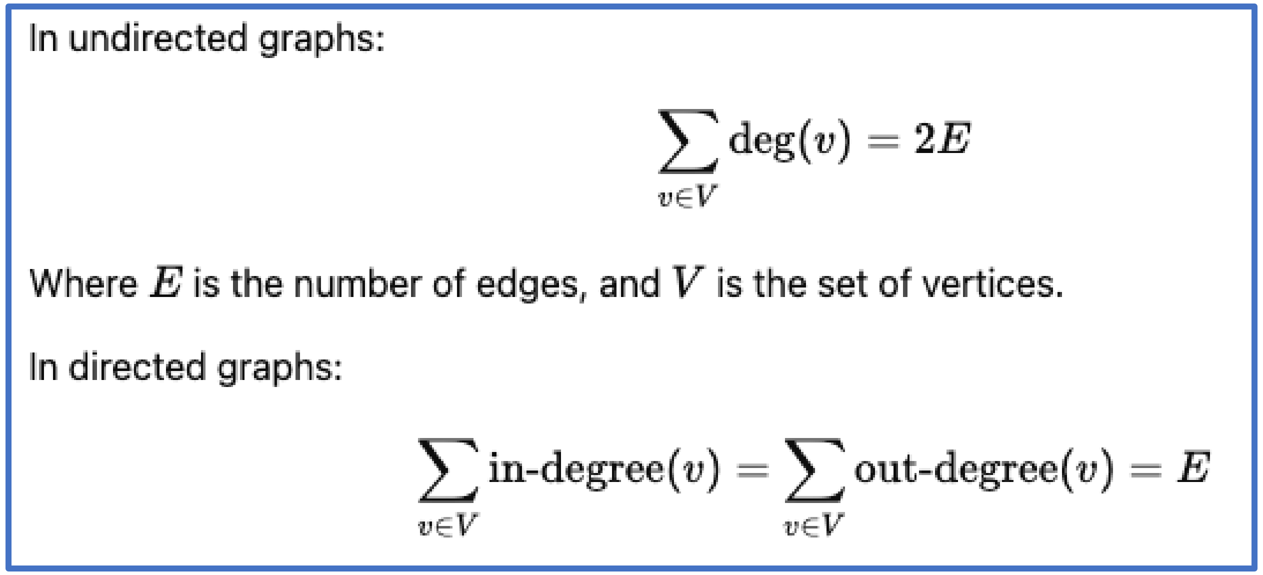
Road networks between cities are typically undirected.
Street networks within cities are almost always directed because of one-way streets.
Most graphs of graph-theoretic interest are undirected.

Degree of a vertex

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
- If d_i is the degree of a vertex i in a graph G with n vertices and m edges, the number of edges is

$$m = \frac{\sum_0^{n-1} d_i}{2}$$

Hint: Adjacent vertices are counted twice.


In undirected graphs:

$$\sum_{v \in V} \deg(v) = 2E$$

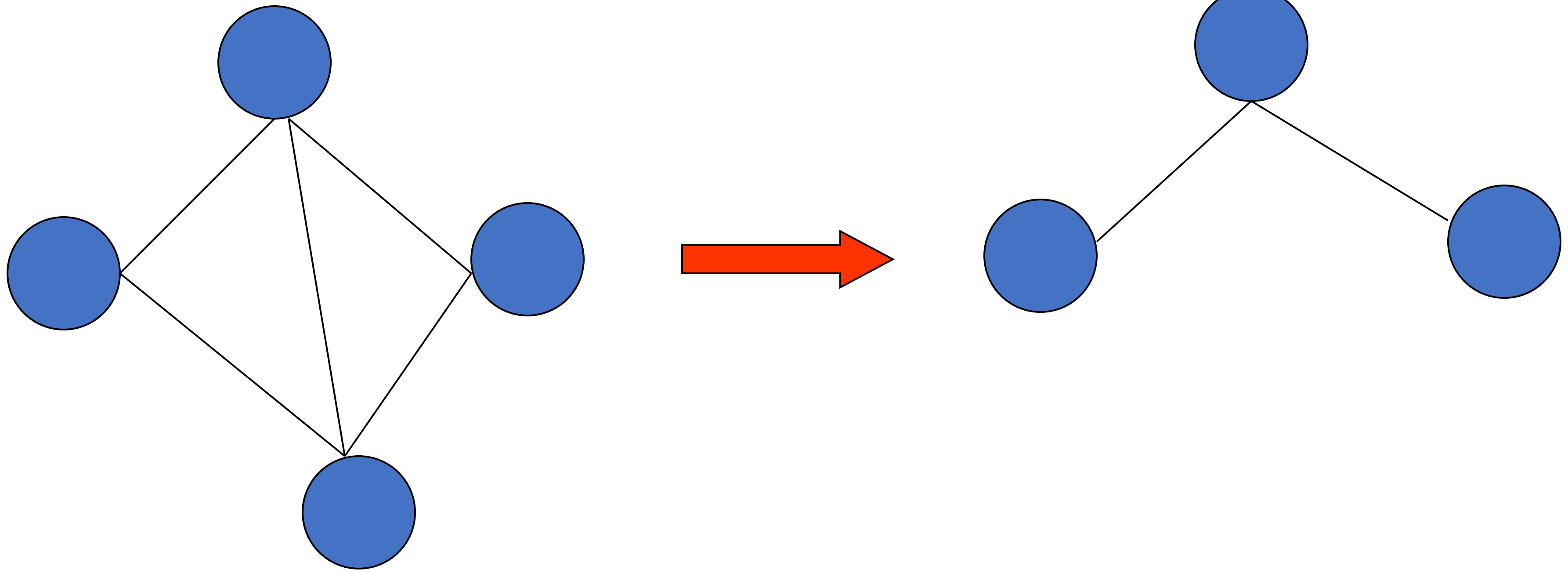
Where E is the number of edges, and V is the set of vertices.

In directed graphs:

$$\sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v) = E$$

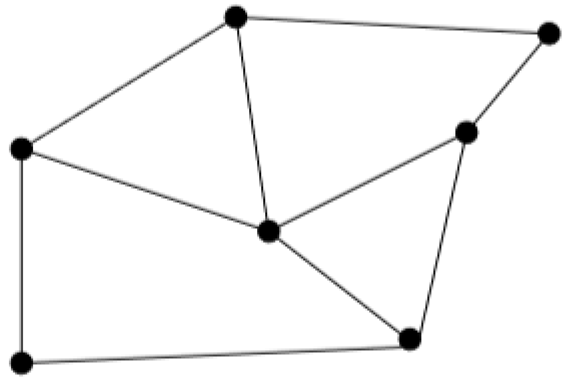
Subgraph

- Subgraph:
  - subset of vertices and edges

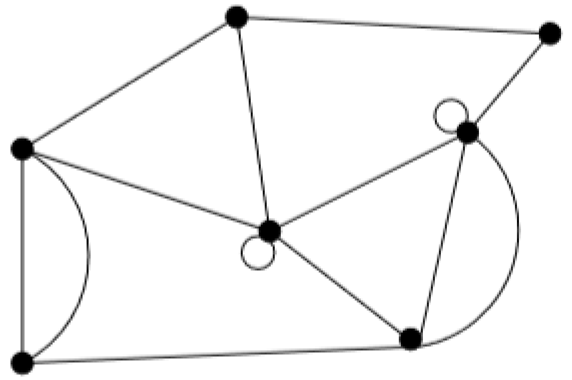


Simple vs. Non-simple Graphs

- Certain types of edges complicate working with graphs.
- A self-loop is an edge (u, u) involving only one vertex.
- An edge (u, v) is a multi-edge if it occurs more than once in the graph.



simple

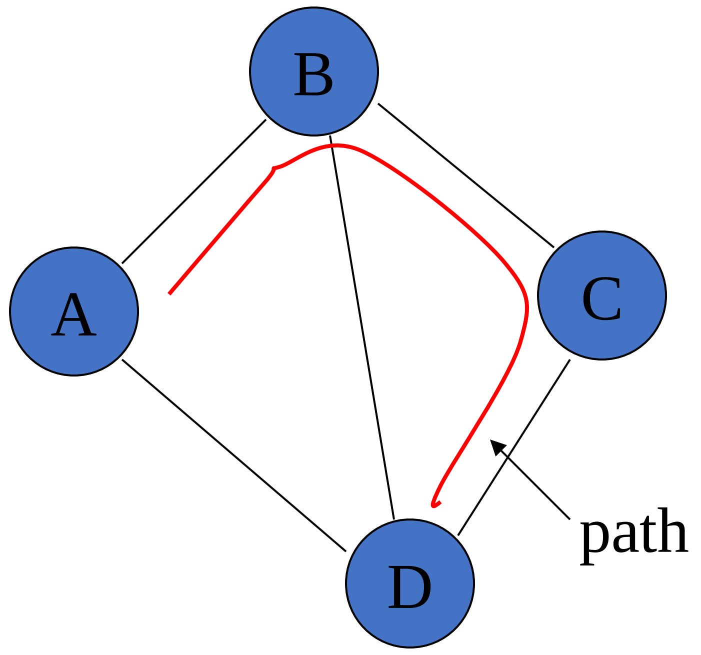


non–simple

- Any graph which avoids these structures is called simple.

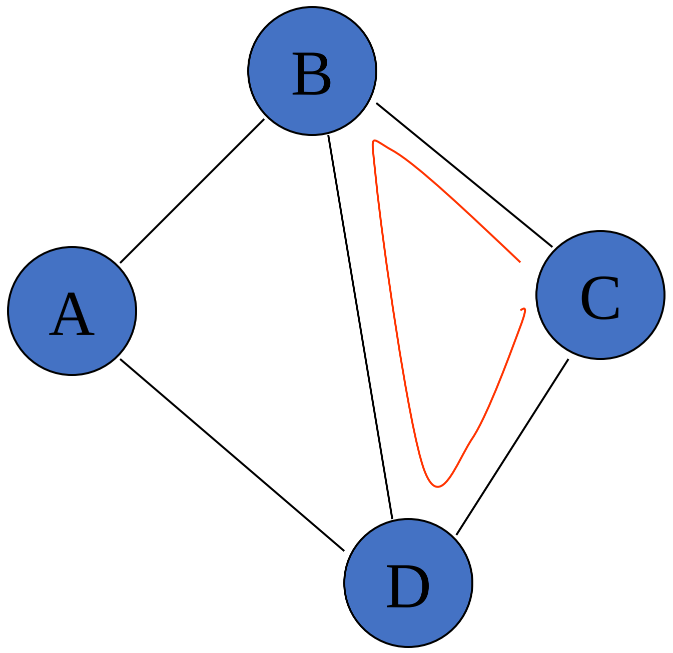
Simple Path

- A simple path is a path such that all vertices are distinct, except that the first and the last could be the same.
    - *ABCD* is a simple path

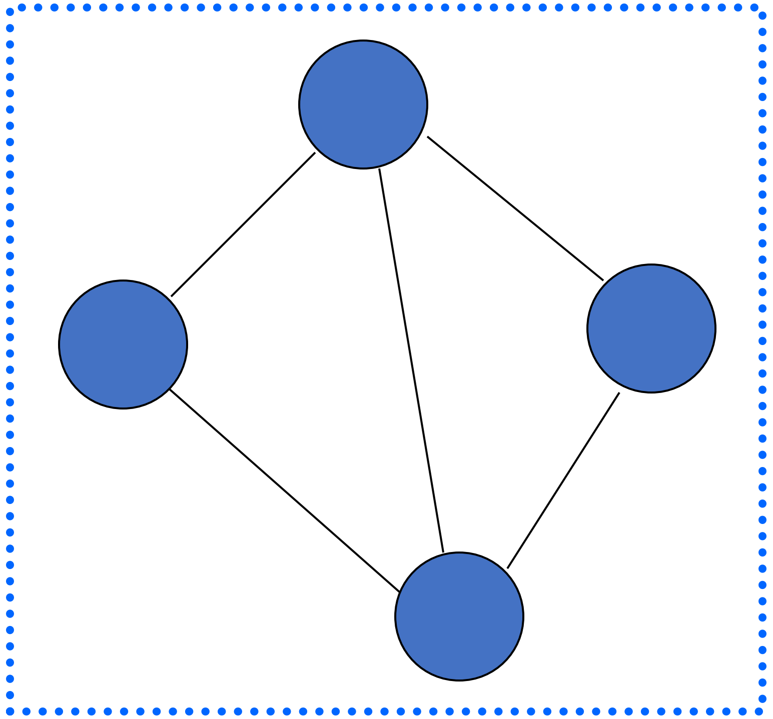


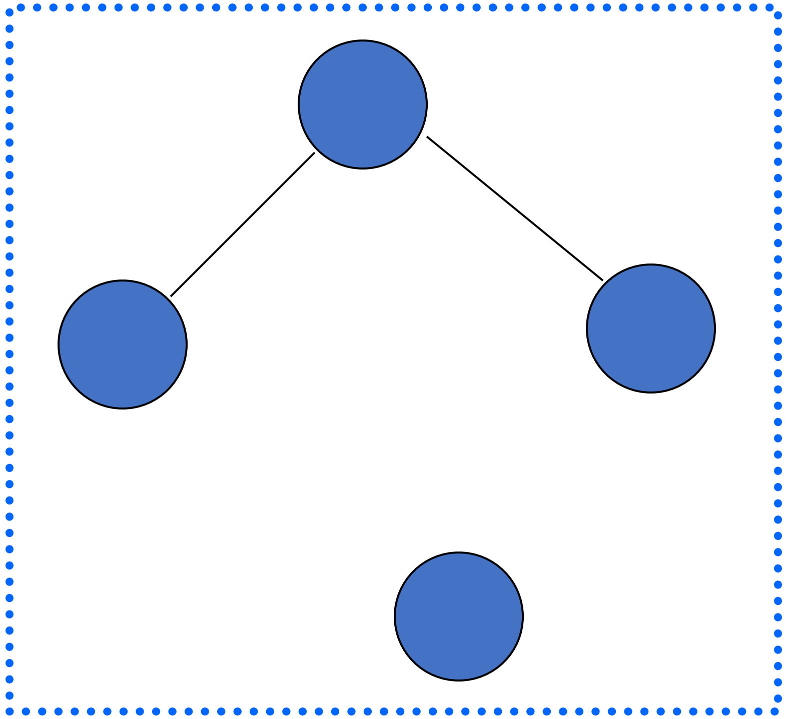
Cycle

- A cycle is a path that starts and ends at the same point. For undirected graph, the edges are distinct.
    - *CBDC* is a cycle



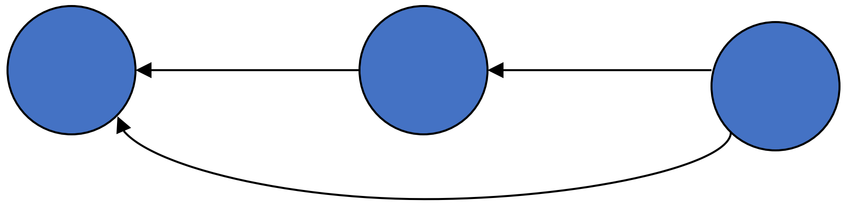
Connected vs. Disconnected Graphs


Connected Graph


Unconnected Graph

Directed Acyclic Graph

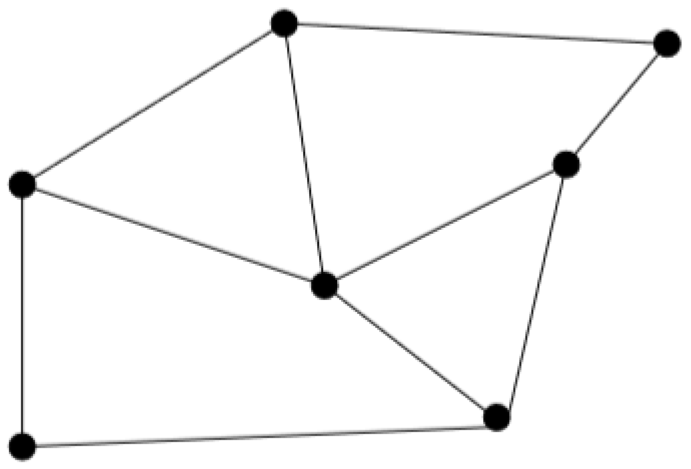
- Directed Acyclic Graph (DAG) : directed graph without cycle

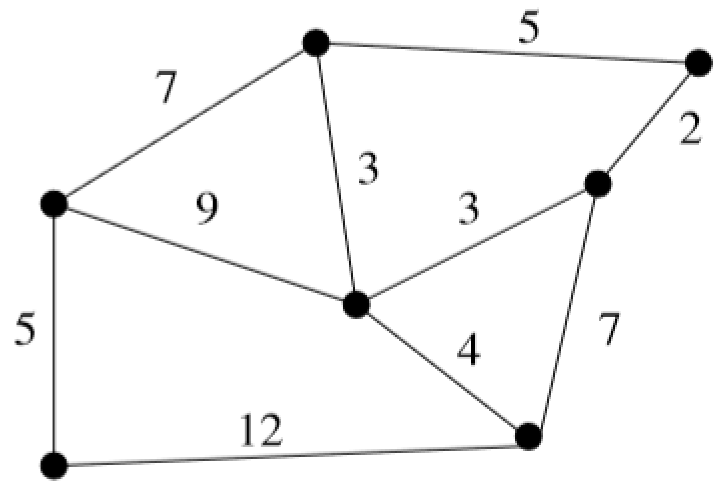


- what type of relationship could be represented here?

Weighted Graph

- Weighted graph: a graph with numbers assigned to its edges
- Weight: cost, distance, travel time, hop, etc.

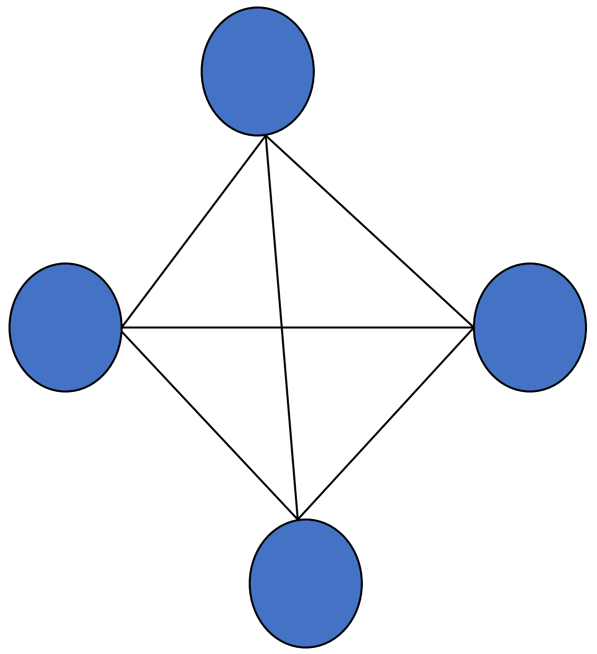

unweighted



weighted

Complete Graph

- There is an edge between any two vertices (undirected graph, K_n)

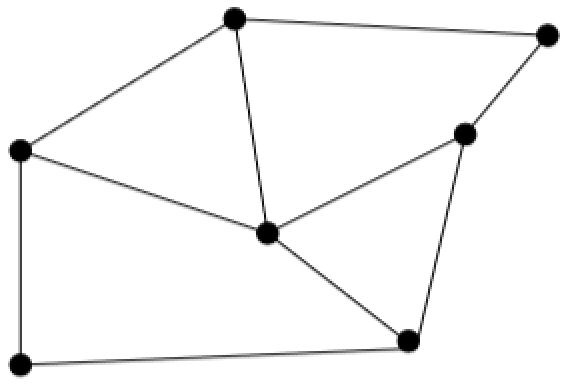


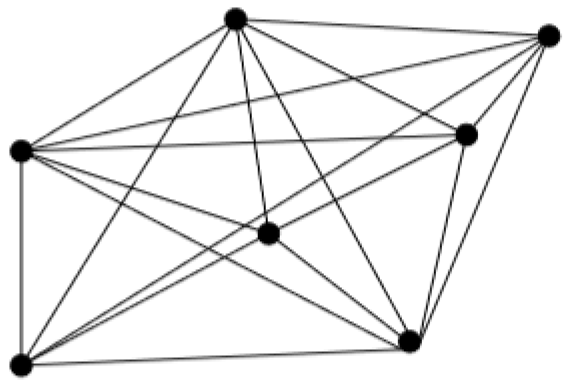
Total number of edges in graph:

$$m = \frac{n(n-1)}{2} = O(n^2)$$

Sparse vs. Dense Graphs

- There is a very small number of edges in the graph.
- Graphs are usually sparse due to application-specific constraints.
- Road networks must be sparse because of road junctions.

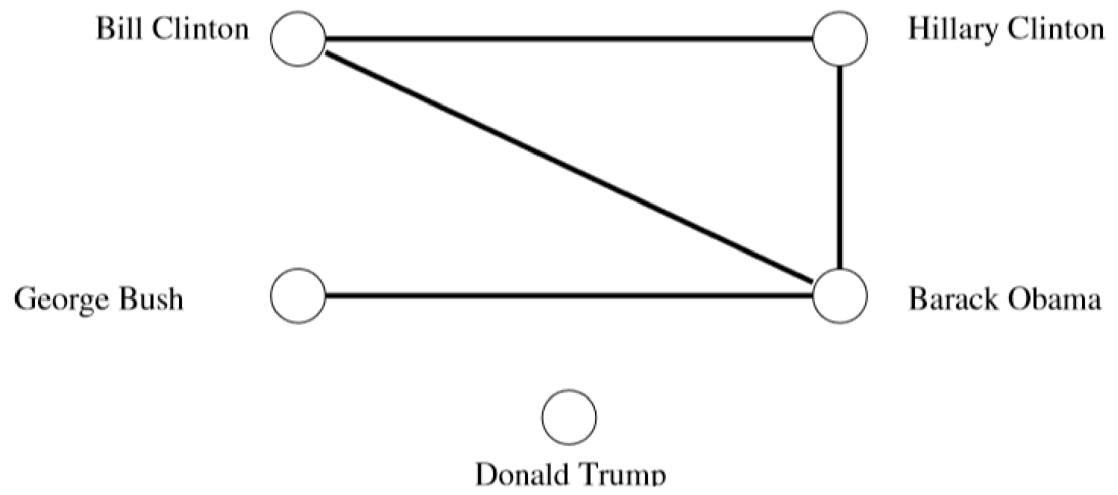

sparse


dense

- Typically, dense graphs have a quadratic number of edges while sparse graphs are linear in size.

The Friendship Graph

- Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



- This graph is well-defined on any set of people: UNT, Dallas, or the world.
- What questions might we ask about the friendship graph?

Is there a path of friends between any two people?

- A graph is connected if there is a path between any two vertices.
- The notation of six degrees of separation presumes the world’s social network is a connected graph.
  - Six degrees of separation is the idea that all people are six, or fewer, social connections away from each other.

Who has the most friends?

- The degree of a vertex is the number of edges adjacent to it.

What is the largest clique?

- A social clique is a group of mutual friends who all hang around together.
- A graph theoretic clique is a complete subgraph, where each vertex pair has an edge between them. Cliques are the densest possible subgraphs.
- Within the friendship graph, we would expect that large cliques correspond to workplaces, neighborhoods, religious organizations, schools, and the like.

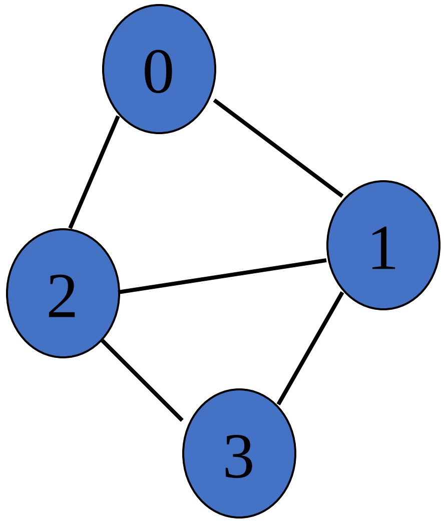
Representation Of Graph

- Two representations:
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix

- Assume n nodes in graph
- Use Matrix $A[0 \ldots n-1][0 \ldots n-1]$
  - if vertex i and vertex j are adjacent in graph, $A[i][j] = 1$, otherwise $A[i][j] = 0$
  - if vertex i has a loop, $A[i][i] = 1$
  - if vertex i has no loop, $A[i][i] = 0$

Example of Adjacency Matrix



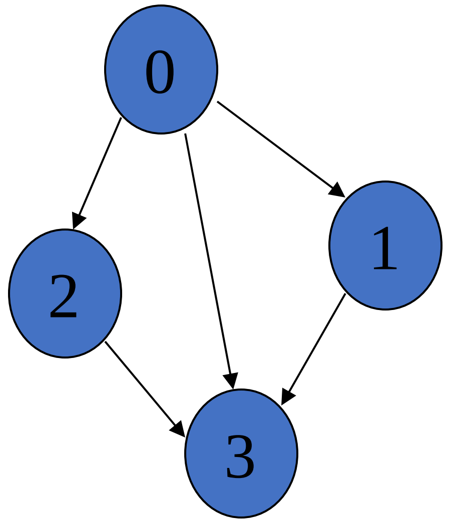
A[i][j]	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

So, Matrix A = $$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Undirected vs. Directed

- Undirected graph
  - adjacency matrix is symmetric
  - A[i][j]=A[j][i]
- Directed graph
  - adjacency matrix may not be symmetric
  - A[i][j] $\neq$ A[j][i]

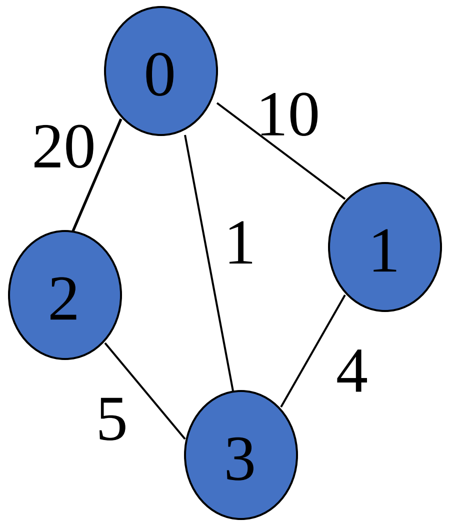
Example of Directed Graph



A[i][j]	0	1	2	3
0	0	1	1	1
1	0	0	0	1
2	0	0	0	1
3	0	0	0	0

So, Matrix A = $\begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$

Example of Weighted Graph

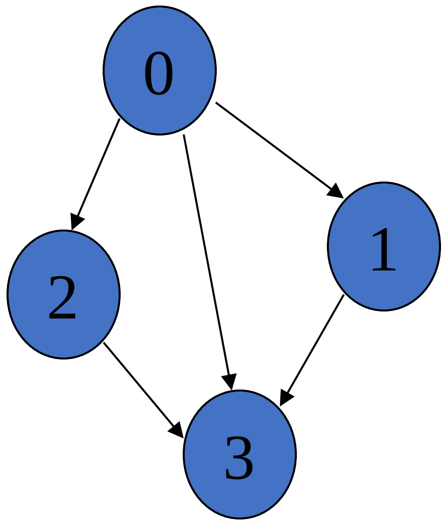


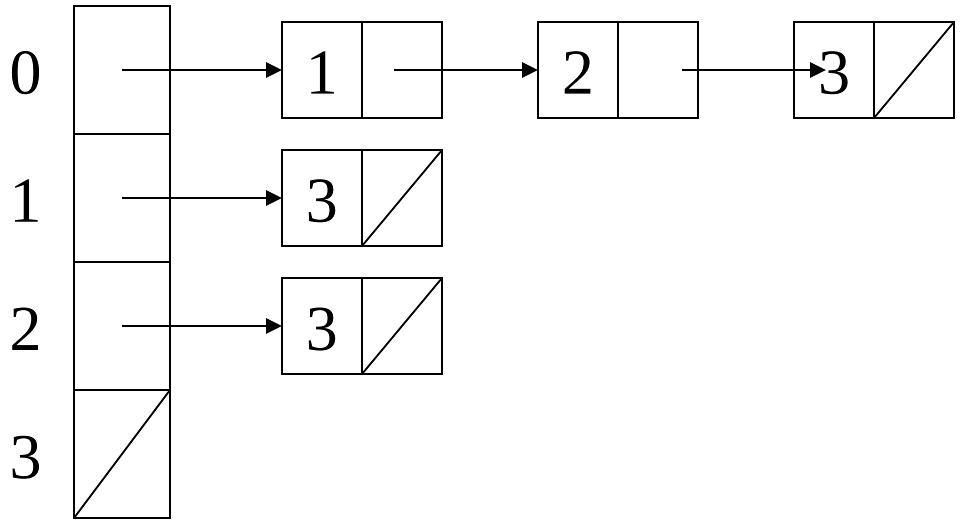
A[i][j]	0	1	2	3
0	0	10	20	1
1	10	0	0	5
2	20	0	0	4
3	1	5	4	0

$$\text{So, Matrix } A = \begin{pmatrix} \mathbf{0} & \mathbf{20} & \mathbf{10} & \mathbf{1} \\ \mathbf{20} & \mathbf{0} & \mathbf{0} & \mathbf{5} \\ \mathbf{10} & \mathbf{0} & \mathbf{0} & \mathbf{4} \\ \mathbf{1} & \mathbf{5} & \mathbf{4} & \mathbf{0} \end{pmatrix}$$

Adjacency List

- An array of lists
- The ith element of the array is a list of vertices that connect to vertex i

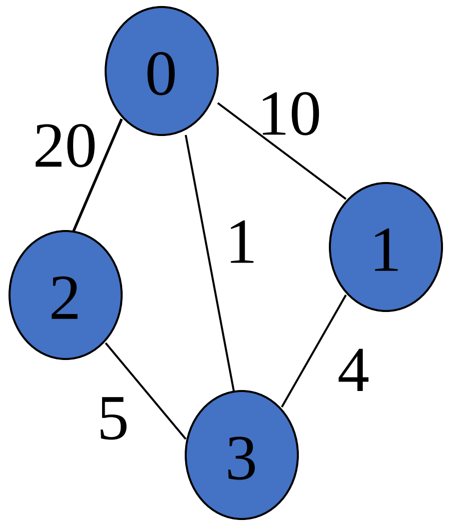


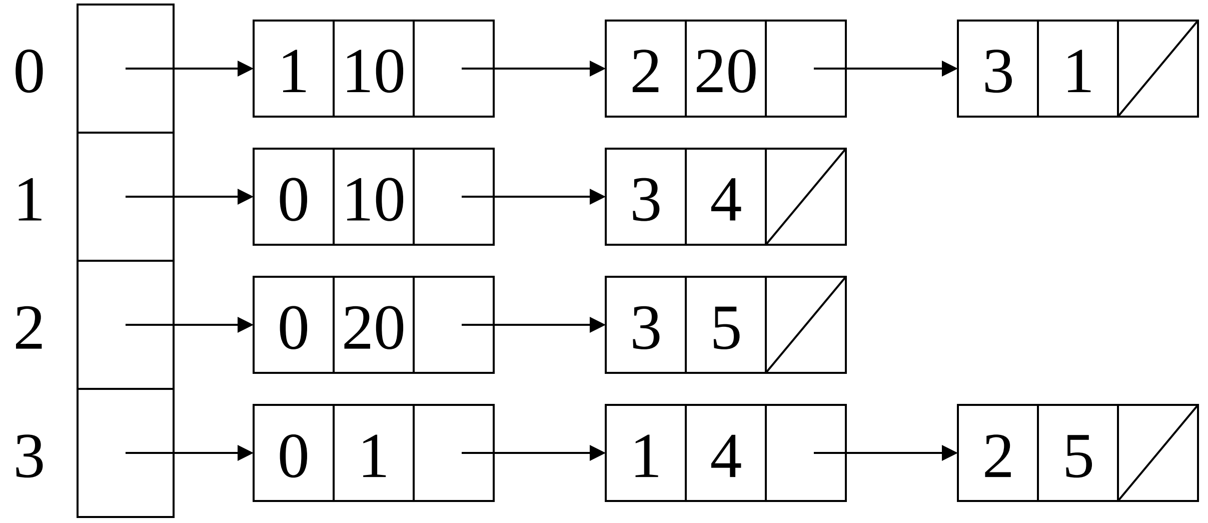


vertex 0 connects to vertex 1, 2 and 3
vertex 1 connects to 3
vertex 2 connects to 3

Example of Weighted Graph

- Weighted graph: extend each data point with an addition field: weight





Edges in Adjacency Lists

```
typedef struct{
     int y;
     int weight;
     struct edge *next;
}edge;
```

Graph in Adjacency Lists

```
typedef struct{
     edge *edges[MAXV+1];
     int degree[MAXV+1];
     int nvertices;
     int nedges;
     bool directed;
}graph;
```

Comparison Of Representations

Time Cost	Adjacency Matrix	Adjacency List
Given two vertices *u* and *v*: find out whether *u* and *v* are adjacent		
Given a vertex *u*: enumerate all neighbors of *u*		
For all vertices: enumerate all neighbors of each vertex		

Comparison Of Representations

Time Cost	Adjacency Matrix	Adjacency List
Given two vertices *u* and *v*: find out whether *u* and *v* are adjacent	$O(1)$	degree of node $O(n)$
Given a vertex *u*: enumerate all neighbors of *u*	$O(n)$	degree of node $O(n)$
For all vertices: enumerate all neighbors of each vertex	$O(n^2)$	Summations of all node degree $O(m)$

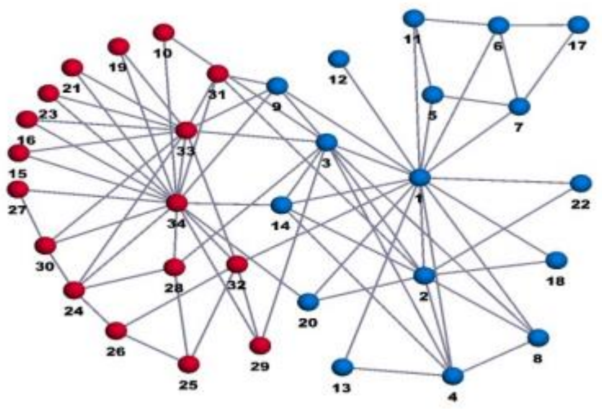
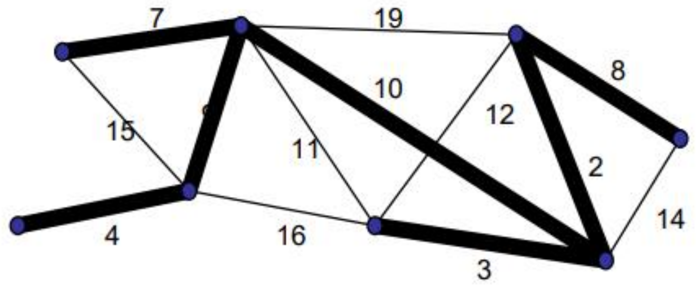
Space Requirements

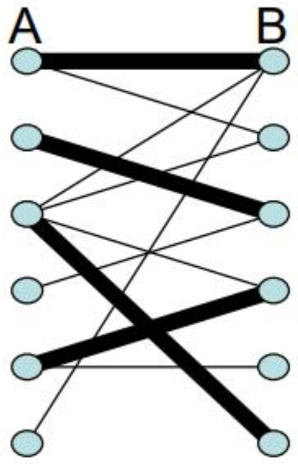
- Memory space:
  - adjacency matrix $O(n^2)$
  - adjacency list $O(m)$

- Sparse graph
  - adjacency list is better
- Dense graph
  - same running time

Some Classical Graph Problems

- **[Shortest Paths]:** Find the shortest path between two nodes, or from one node to all other nodes.
- **[Minimum Spanning Trees]:** Find a min-cost subset of edges that connects all nodes.
- **[Matchings]:** Pair up as many nodes as possible or pair up all nodes at minimum total cost.
- **[Flow / Routing]:** Route a maximum amount of some commodity through a capacitated network, possibly at minimum total cost.



Graph Traversal

“Explore a graph”, e.g.:

- Specify an order to search through the nodes of a graph.

Graph Traversal Algorithms (for unweighted/weighted graph):

  - Depth First Traversal
  - Breadth First Traversal

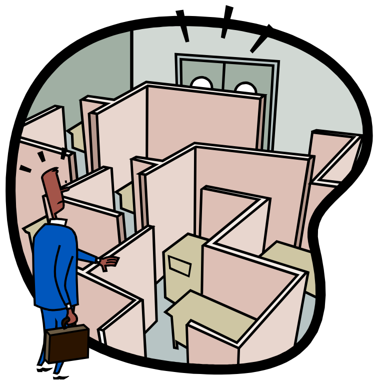
Graph Search:

- Start at the source node and keep searching until find the target node

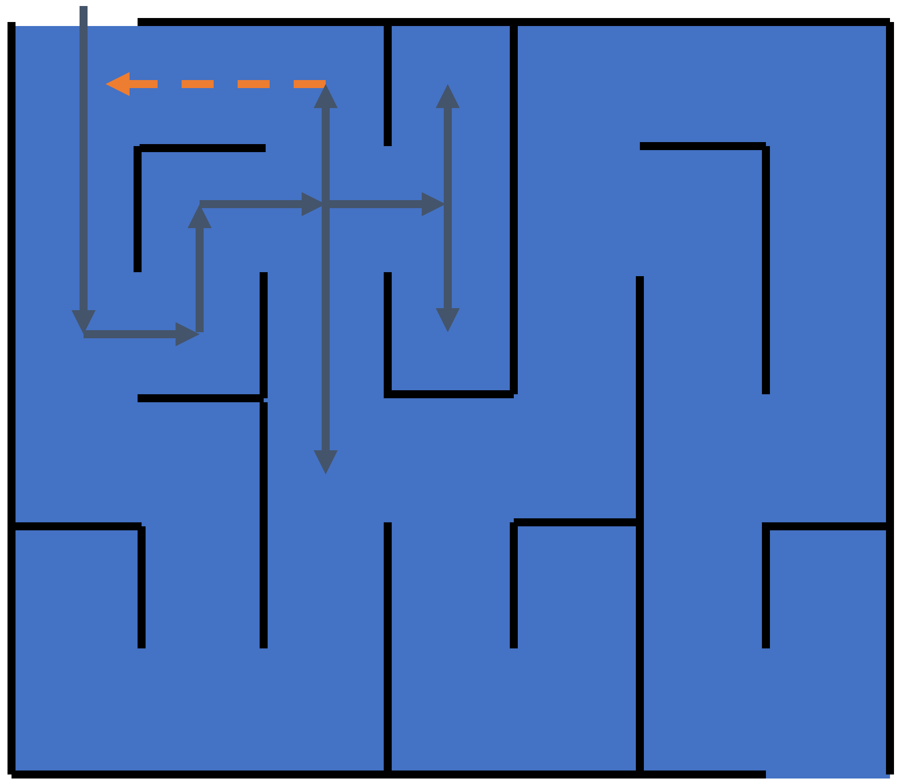
Depth-first Search

- Starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible.
- Simple Connectivity Questions:
    - Are nodes i and j connected by some path?
    - If so, determine such a path.
    - In a digraph, is there a directed path from i to j?
    - Does a (directed) graph have a (directed) cycle?

DFS and Maze Traversal



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Depth-First Traversal

```
DFS(i):
    Visited[i] = True
    For all j such that (i,j) is an edge:
        If Visited[j] = False:
            Pred[j] = i
            DFS(j)
DFT:
    For all i: Pred[i] = null, Visited[i] = False
    For all i: If Visited[i] = False: DFS(i)
```

Depth-First Traversal

- Works in undirected and directed graphs.

- The pred[i] pointers define what is called a depth-first search tree/forest.
- To find a path from i to j (if it exists):
  - Initialize pred and status values for all nodes.
  - Call DFS(i).
  - Then follow pred pointers backward from j to i.

DFS Time Analysis

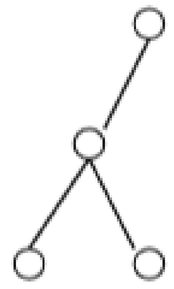
```
DFS(i):
    Visited[i] = True
    For all j such that (i,j) is an edge:
        If Visited[j] = False:
            Pred[j] = i
            DFS(j)
DFT:
    For all i: Pred[i] = null, Visited[i] = False
    For all i: If Visited[i] = False: DFS(i)
```

- DFS gets called with a vertex only once → time in DFS = $\sum_{s \in V} |Adj[s]| = O(m)$
- DFS outer loop adds just $O(n)$ → total time $O(n + m)$, linear time

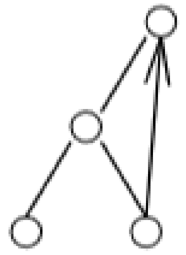
Edge Classification for DFS

Every edge is either:

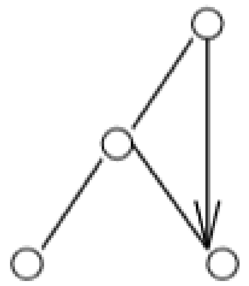
1. A Tree Edge



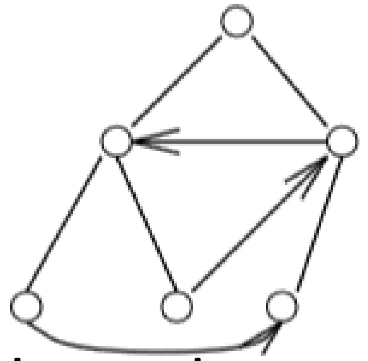
2. A Back Edge
   to an ancestor



3. A Forward Edge
   to a decendant



4. A Cross Edge
   to a different node



On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above

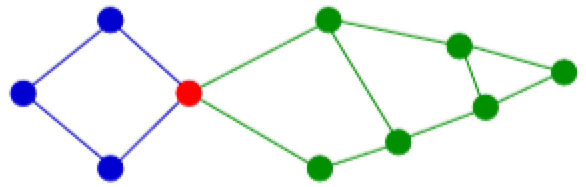
DFS Application: Finding Cycles

- Back edges are the key to finding a cycle in an undirected graph.
- Any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x.

- In a DFS of an undirected graph, every edge is either a tree edge or a back edge.
- Pred[x] != y // found back edge (y, x)

DFS Application: Articulation Vertices

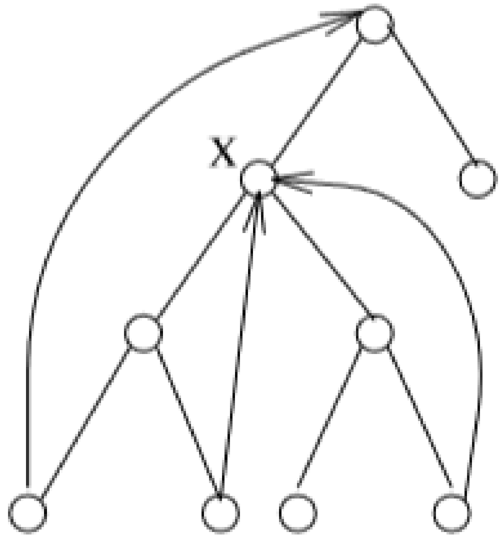
- Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



- An articulation vertex is a vertex of a connected graph whose deletion disconnects the graph. Clearly connectivity is an important concern in the design of any network.
- Articulation vertices can be found in $O\big(n(m+n)\big)$–just delete each vertex to do a DFS on the remaining graph to see if it is connected.

DFS Application: Articulation Vertices (Cont.)

In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident to a proper ancestor of v.



Leaves cannot be articulation vertices

The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

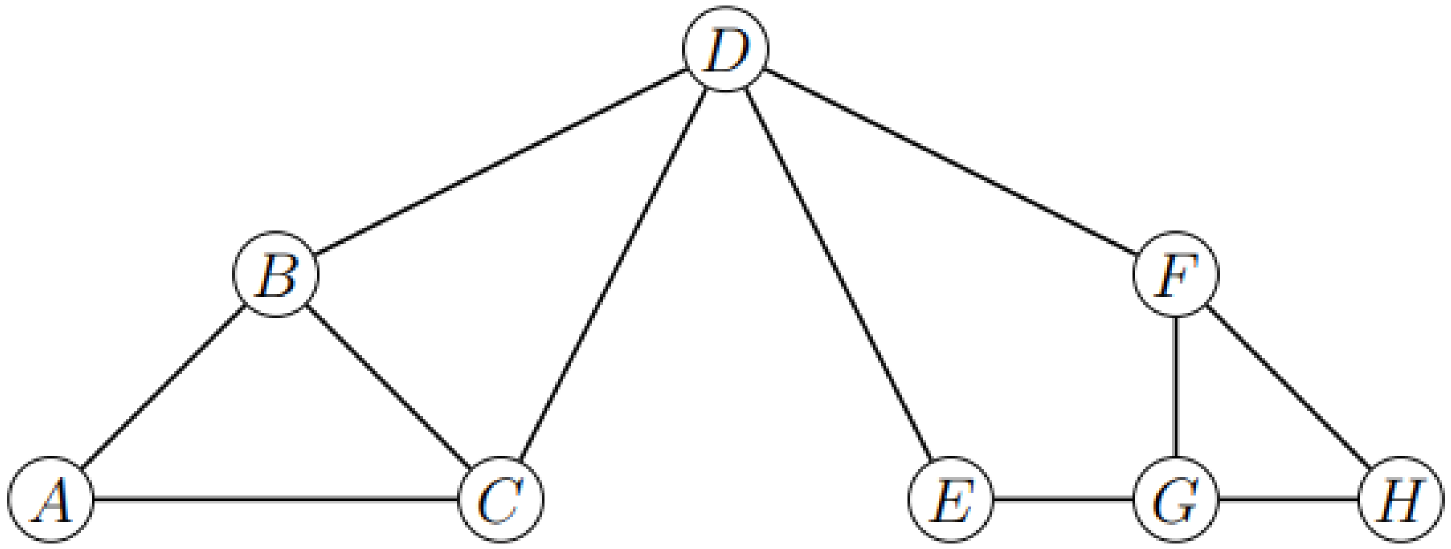
Edge Classification

```
For all i: Processed[i] = False, Discovery[i] = 0
Time = 1
DFS(i):
     Visited[i] = True
     Discovery[i] = Time++
     For all j such that (i,j) is an edge:
          If Visited[j] = False:
               Pred[j] = i
               DFS(j)
          Edge_classification(i,j)
     Processed[i] = True
```

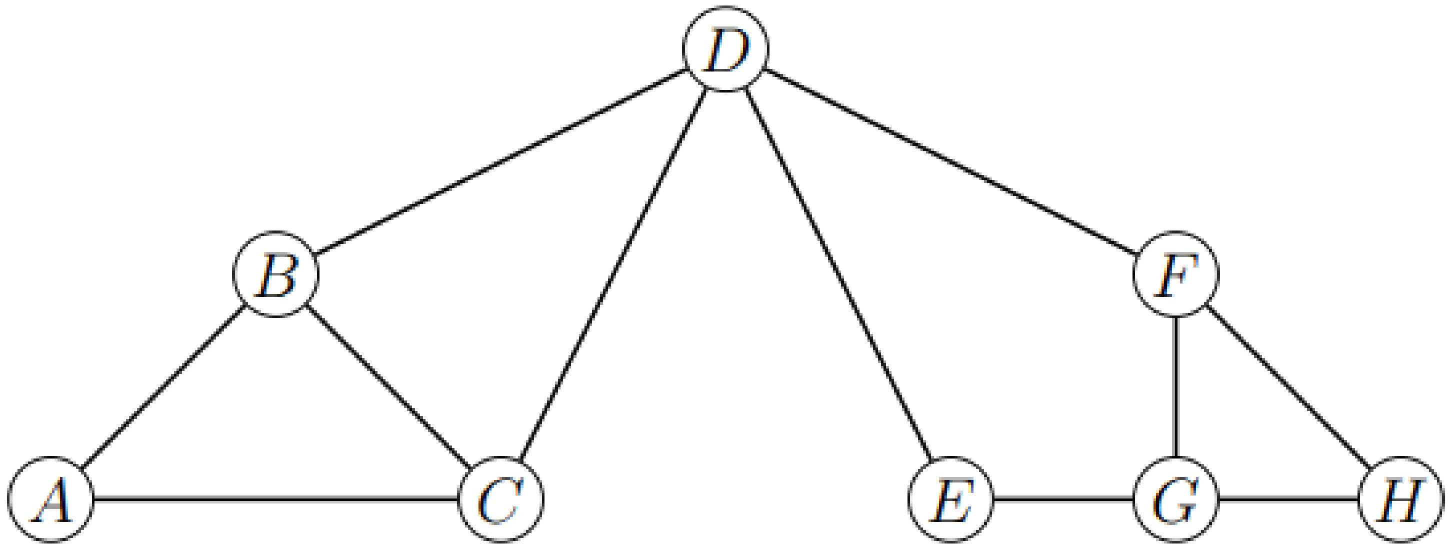
Edge Classification (Cont.)

```
Edge_classification(i,j):

       if Pred[j] = i : return ‘Tree Edge’

       if Visited[j] && !Processed[j]: return ‘Back Edge’

       if Processed[j] && Discovery[i] < Discovery[j]: return ‘Forward Edge’

       if Processed[j] && Discovery[i] > Discovery[j]: return ‘Cross Edge’
```

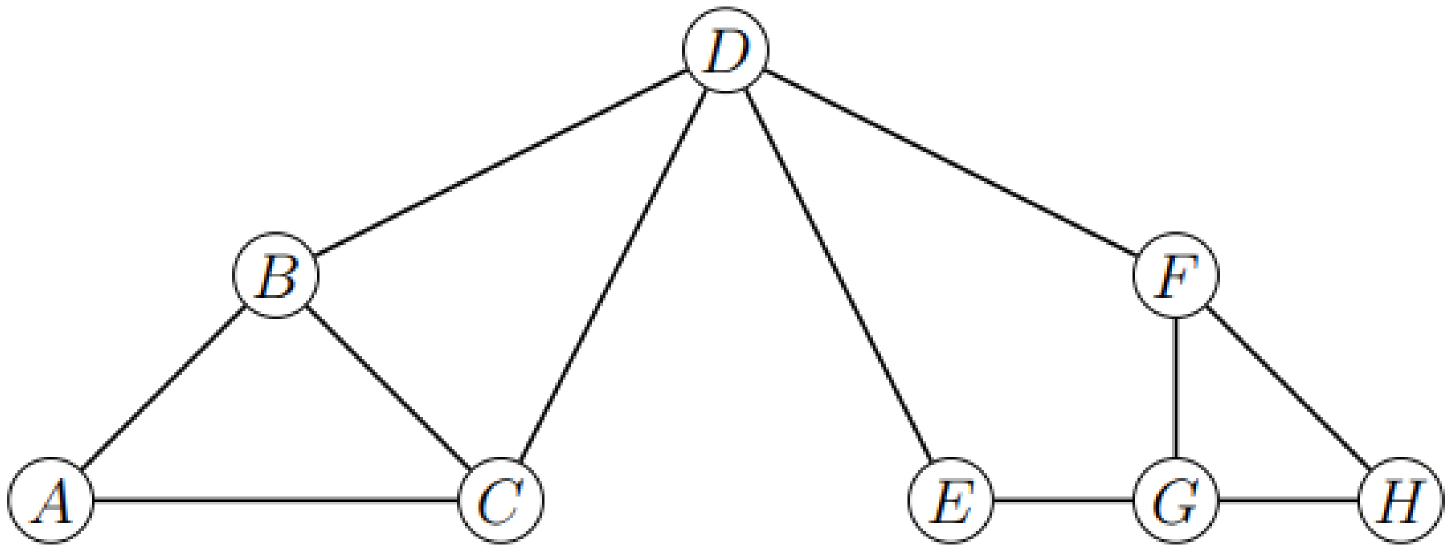
- For the following questions, assume that the graph is represented using adjacency lists, and that **all adjacency lists are sorted**, i.e., the vertices in an adjacency list are always sorted alphabetically.



- If you use depth-first search to find a path from A to H, write down the resulting path as a sequence of vertices.

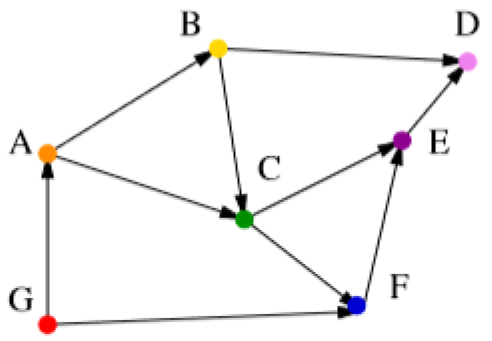


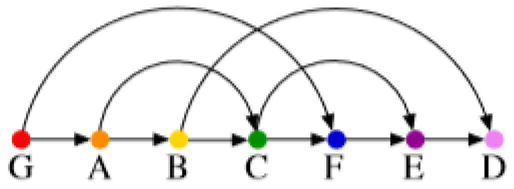
- To determine whether the graph has cycles, you decide to use the edge classification of depth-first search. Run depth-first search, starting from vertex A, and label every edge with T if it's a tree edge, B if it's a back edge, F if it's a forward edge, and C if it's a cross edge.



Topological Sorting

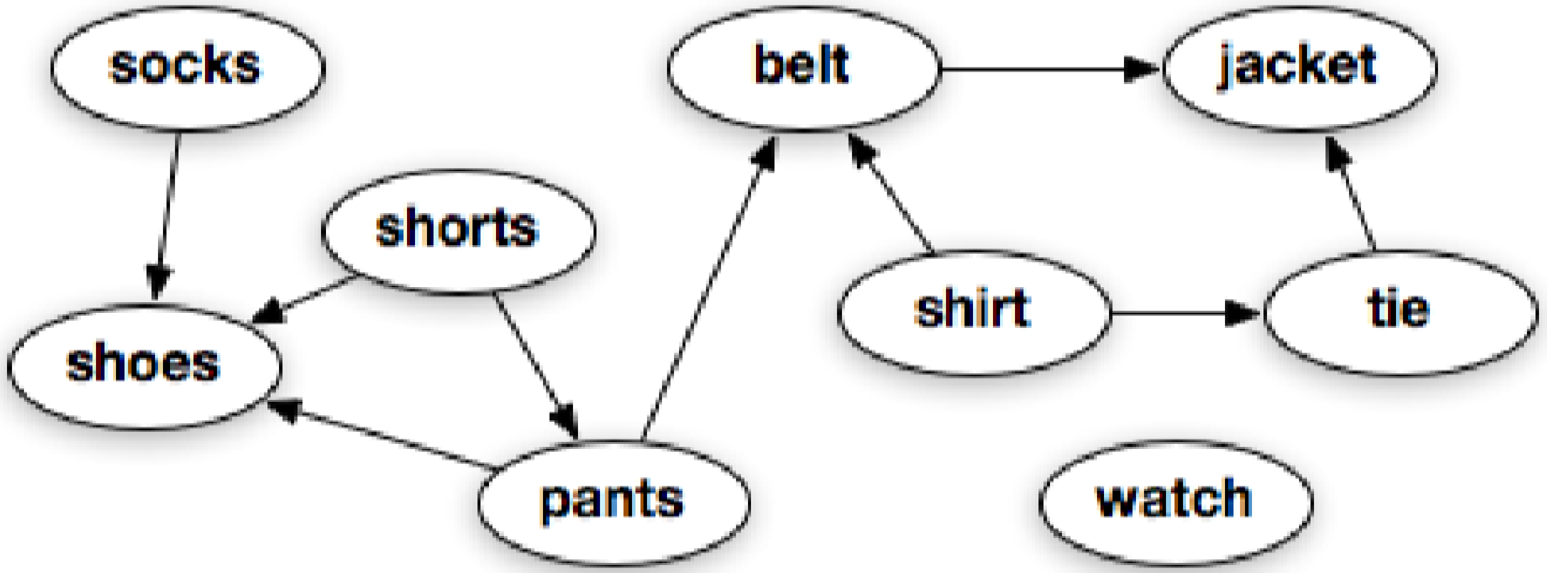
- A directed, acyclic graph has no directed cycles.
- A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.
- DAGs (and only DAGs) has at least one topological sort (here G,A,B,C,F,E,D).





Applications of Topological Sorting

- Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given precedence constraints.
- Example: Dressing priority schedule



Topological Sorting

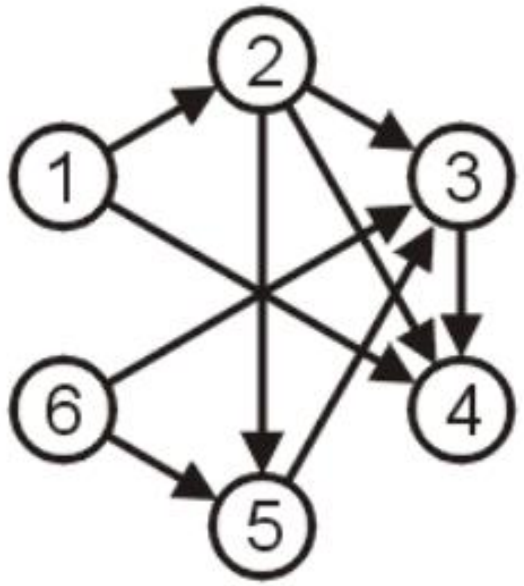
There are several ways to topologically sort in $O(n + m)$ time; for example:

- Find a node with no incoming edges, add it next to the ordering, remove it from our graph, repeat.
  - If we ever find that every node has an incoming edge, then our graph must contain a cycle (so this gives an alternate way to do cycle detection).

- Depth-first search gives us another very simple topological sorting algorithm…

Topological Sort: Example

- Let us define the table of in-degrees



Vertex	In-degree
1	0
2	1
3	3
4	3
5	2
6	0

- Queue: 1, 6, 2, 5, 3, 4

Topological Sorting via DFS

```
For all i: Discovery[i] = 0, Finish[i] = 0
Time = 1
DFS(i):
        Visited[i] = True
        Discovery[i] = Time++
        For all j such that (i,j) is an edge:
                If Visited[j] = False:
                        Pred[j] = i
                        DFS(j)
        Finish[i] = Time++
```

- To topologically sort a DAG, just perform a Full- DFS and then output nodes in reverse order of finish times.
    - We can output each node when it is “finished”, or we can sort the nodes by their finish times (in linear time with counting sort) as a post-processing step.
    - When a node finishes insert it to the top of a linked list.

Topological Sorting

- **Claim**: In a DAG with an edge/path from node i to node j, we will have Finish(i) > Finish(j).
- Proof: Consider two cases:
- (a) DFS i first.
- (b) DFS j first.
- In both cases, we have Finish(i) > Finish(j) as long as our graph contains no directed cycles (which it doesn’t, since it’s a DAG!)

Breadth-First Traversal

- Breadth-first search (BFS) starts by visiting an arbitrary vertex, then visits all vertices whose distance from the starting vertex is one, then all vertices whose distance from the starting vertex is two, and so on.
- If your graph is an undirected tree, BFS performs a level-order tree traversal.
- Breadth-first search is appropriate if we are interested in the shortest paths on unweighted graphs.

Breadth-First Search

```
BFS(s):
        For all nodes i:
                pred[i] = null
                dist[i] = Infinity
        Q = {s}
        dist[s] = 0
        While Q is nonempty:
                i = next node in Q
                For all nodes j such that (i,j) is an edge:
                        If dist[j] = Infinity,
                                pred[j] = i
                                dist[j] = dist[i] + 1
                                Append j to the end of Q
```

- `Q`: a FIFO queue, how about a stack?

Breadth-First Search Time Analysis

- Adjacent list of u looped through only once, time = $\sum_{u \in V} |Adj[u]| = O(m)$

- $O(n+m)$ ("LINEAR TIME") to also list vertices unreachable from u (those still not assigned level)

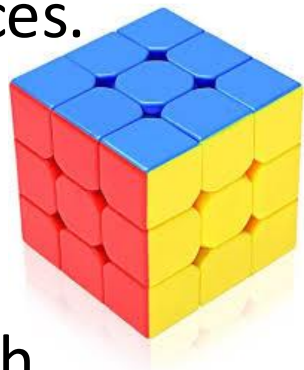
BFS: Path Finding

- We can reconstruct this path by following the chain of ancestors from x to the root. Note that we have to work backward. We cannot find the path from the root to x, since that does not follow the direction of the parent pointers. Instead, we must find the path from x to the root.

```
Find_path(s, x, pred):
    if s = x: print(s)
    else:
        Find_path(s, pred[x], pred)
        print(x)
```

BFS: Connected Components

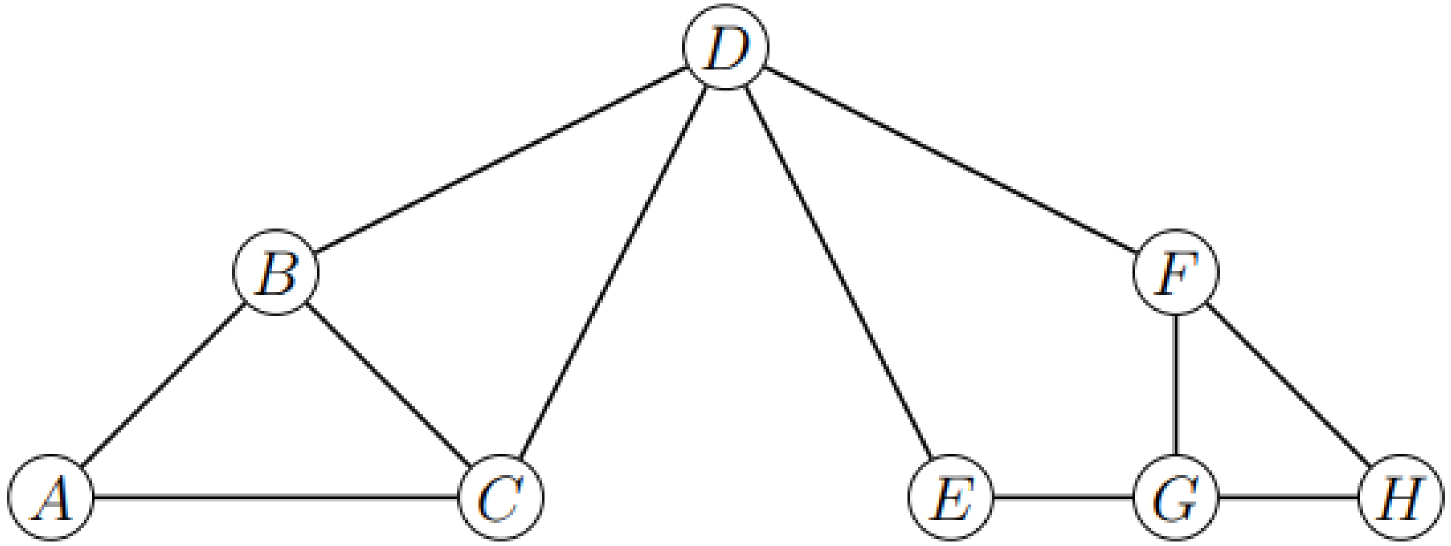
- The connected components of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces. Many seemingly complicated problems reduce to finding or counting connected components.
- For example, testing whether a puzzle such as Rubik's cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.
- Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found.



BFS: Connected Components

```
Connected_components:
    c = 0
    For all i: Pred[i] = null, Visited[i] = False
    For all i:
        If Visited[i] = False:
            c = c + 1
            print(“Component: ”, c)
            BFS(i)
```

- For the following questions, assume that the graph is represented using adjacency lists, and that **all adjacency lists are sorted**, i.e., the vertices in an adjacency list are always sorted alphabetically.



- Suppose that you want to find a path from A to H. If you use breadth-first search, write down the resulting path as a sequence of vertices.