CS 2351 Data Structures

Graphs (I)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

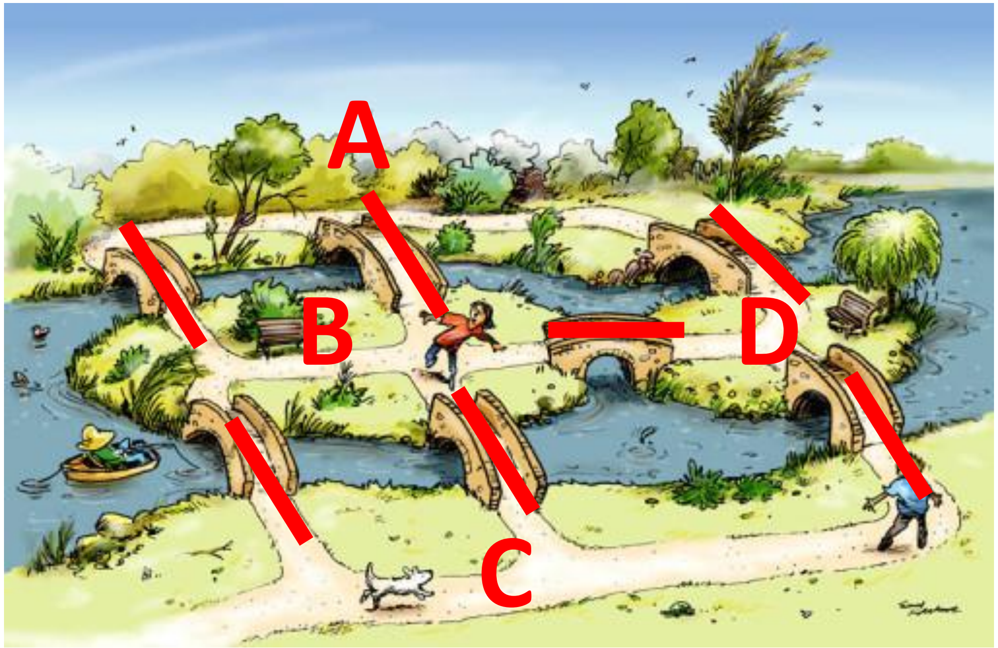
Outline

- Introduction to graphs (Sec. 6.1)
  - Definitions, terminologies
  - Representations

- Elementary graph operations (Sec. 6.2)
  - Depth first search, breadth first search, connected components, spanning trees

Konigsberg Bridge Problem (1736 AD)

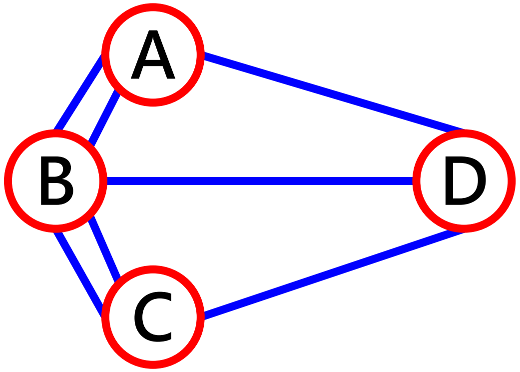
- Given 4 lands with 7 bridges
- Problem: Starting at one land, is it possible to walk across all the bridges exactly once and returning to the starting land?



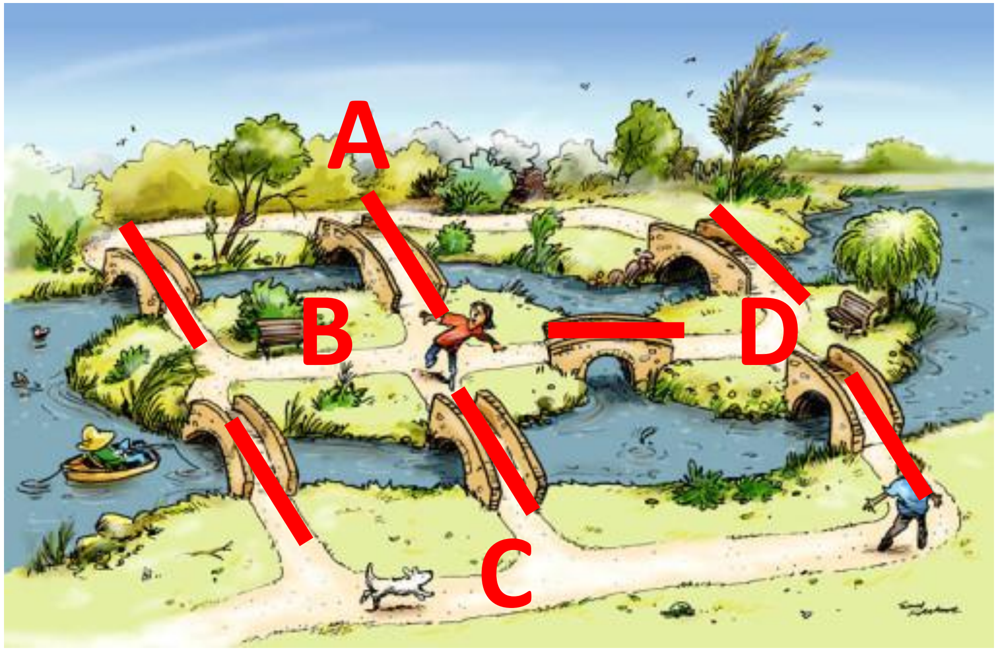
http://simonkneebone.com/2011/11/29/konigsberg-bridge-puzzle/

Konigsberg Bridge Problem

- Leonhard Euler formulated the problem as a graph

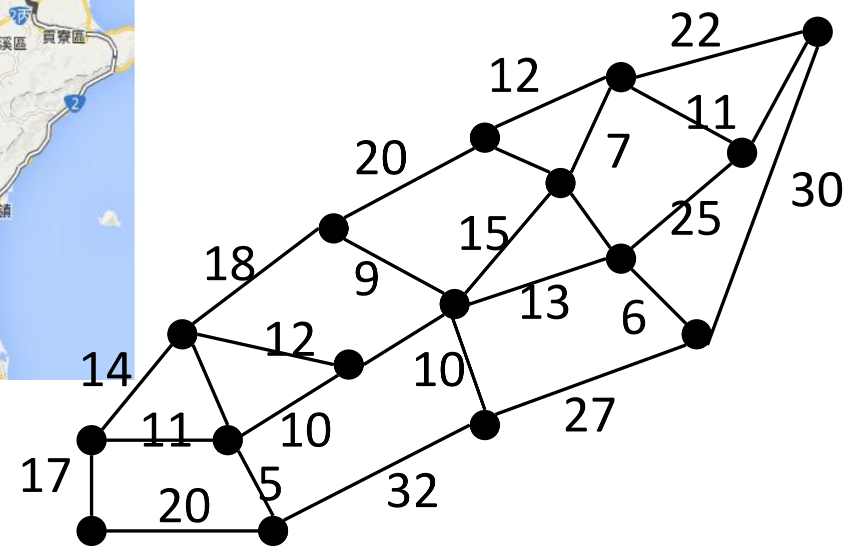


- Proved that the answer to the problem is possible iff **the degree of each vertex is even**



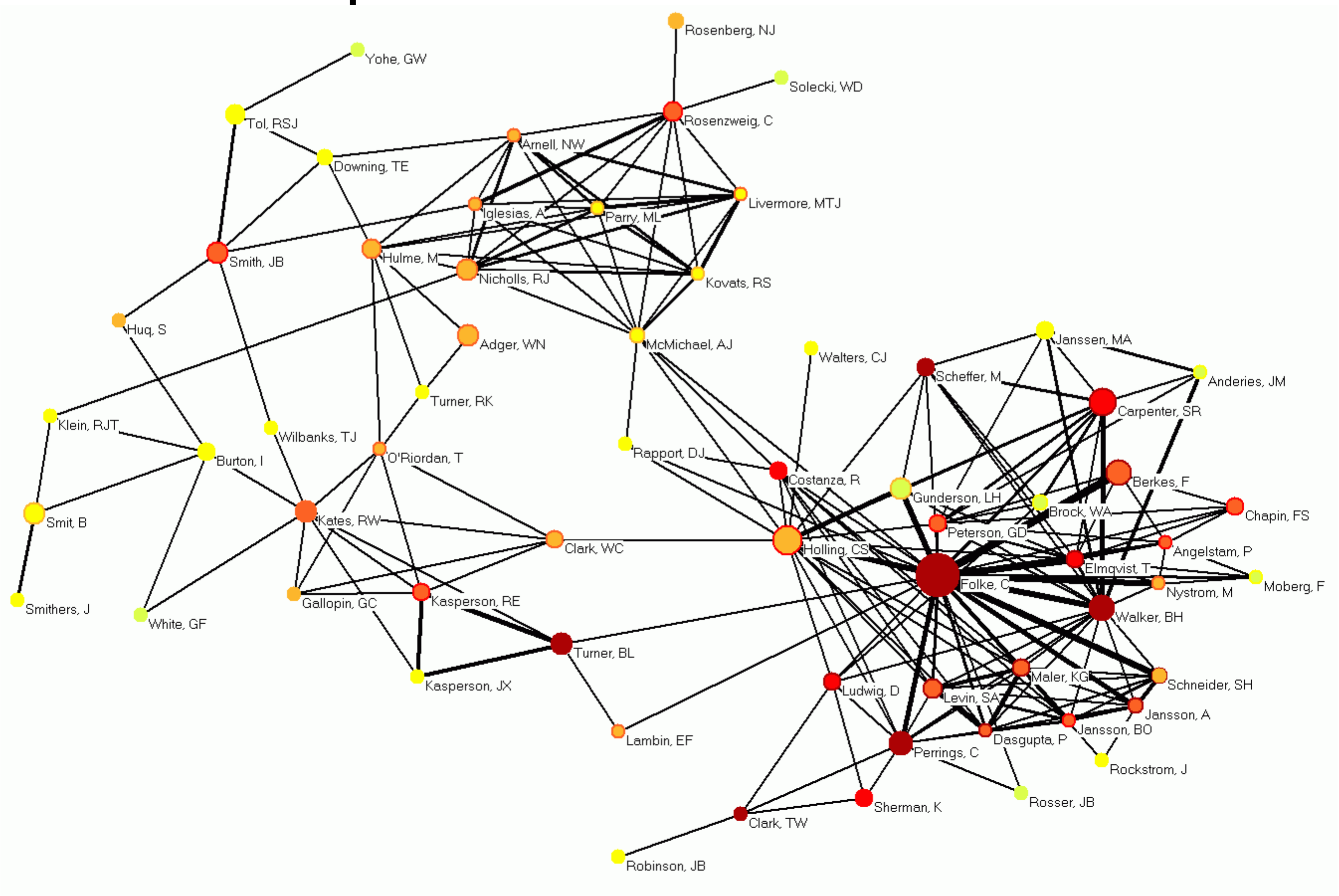
Many Applications of Graphs

- Find the shortest path from Taipei to Hsinchu



Many Applications of Graphs

- Co-authorship

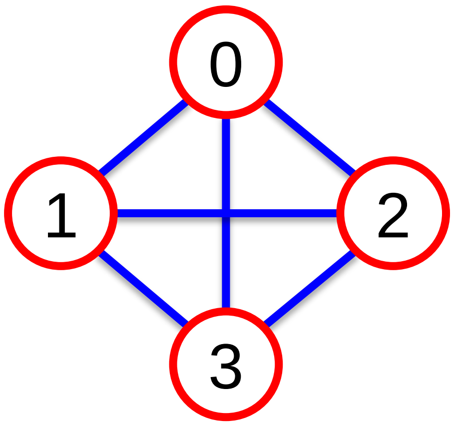


http://www.public.asu.edu/~majansse/pubs/SupplementIHDP.htm

Graph Definition

- A graph, ***G = (V, E)***, consists of two sets, ***V*** and ***E***
  - ***V*** : a set of **vertices**
  - ***E*** : a set of *pairs of vertices* called ***edges***
- **Undirected graph** (simply **graph**)
  - (u,v) and (v,u) represent the same edge
- **Directed graph** (**digraph**)
  - <u,v> ≠ <v,u>
  - <u,v>: u is tail and v is head of edge

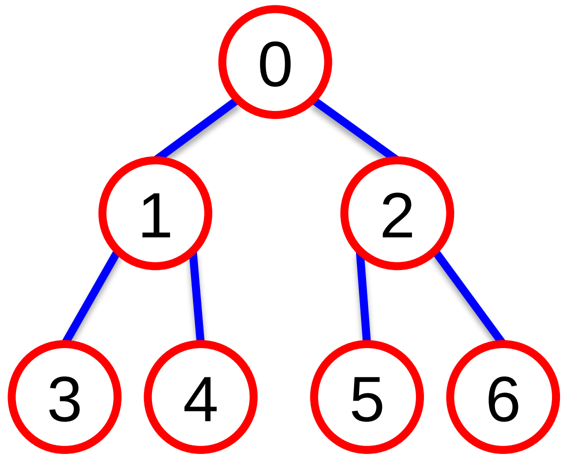
Examples



Undirected Graph

V(G)={0, 1, 2, 3}

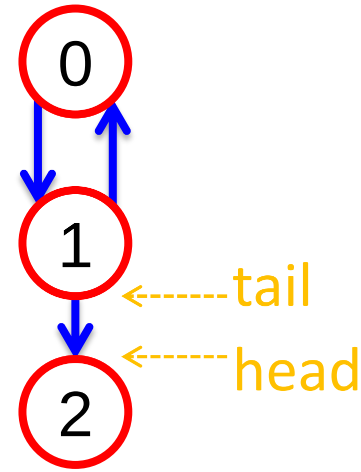
E(G)={(0,1), (0,2), (0,3),
(1,2), (1,3), (2,3)}



Undirected Graph

V(G)={0, 1, 2, 3, 4, 5, 6}

E(G)={(0,1), (0,2), (1,3),
(1,4), (2,5), (2,6)}



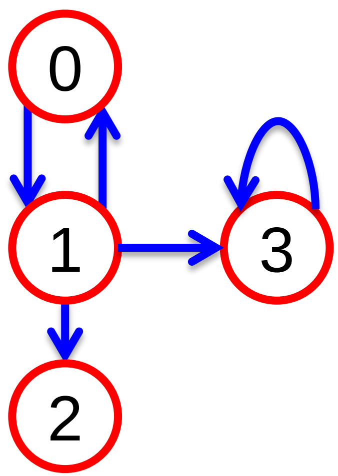
Directed Graph

V(G)={0, 1, 2}

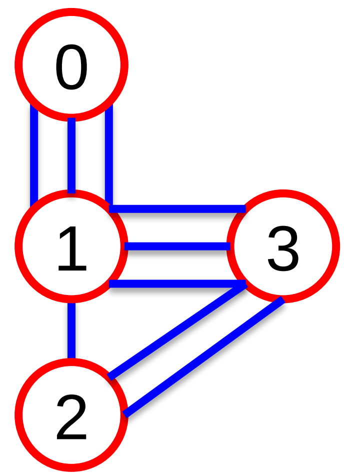
E(G)={<0,1>, <1,0>, <1,2>)}

Restrictions

- *Self edges* and *self loops* are not permitted!
  - Edges of the form (v, v) and <v, v> are not legal
- A graph should not have multiple occurrences of the same edge (otherwise, it is called a *multigraph*)



Graph with self edge



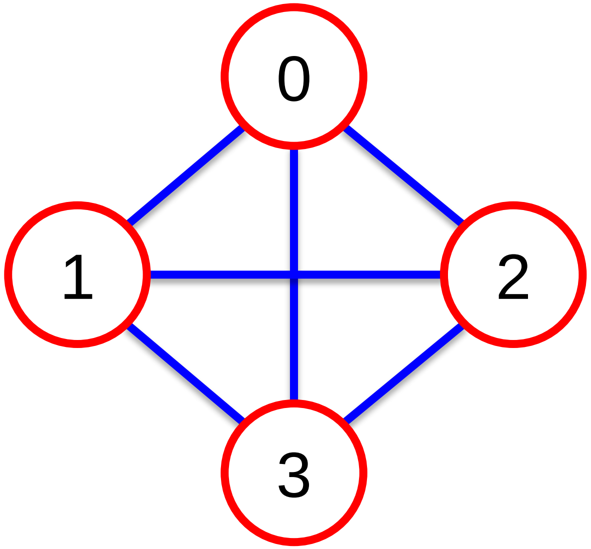
Multigraph

Terminology

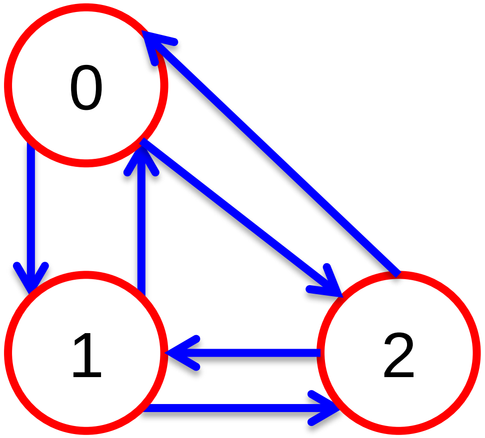
- For a graph with n vertices, the max # of edges of:
  - Undirected graph is *n(n-1)/2*
  - Directed graph is *n(n-1)*
- Vertices u and v are *adjacent* if (u,v) $\in$ E and edge (u,v) is *incident* on vertices u and v
- For a direct edge <u,v>, u is *adjacent to* v and v is *adjacent from* u, and edge <u,v> is *incident* to vertices u and v

Terminology

- Complete undirected graph
  - Graph with n vertices has exactly n(n-1)/2 edges

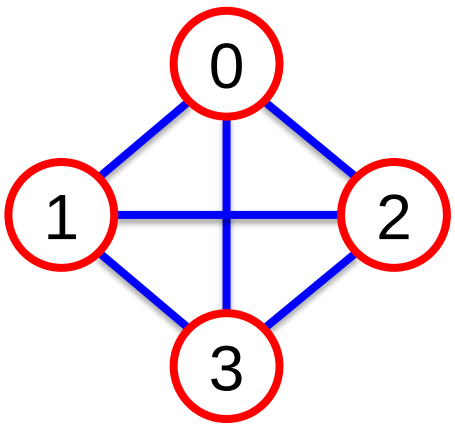


- Complete directed graph
  - Graph with n vertices has exactly n(n-1) edges

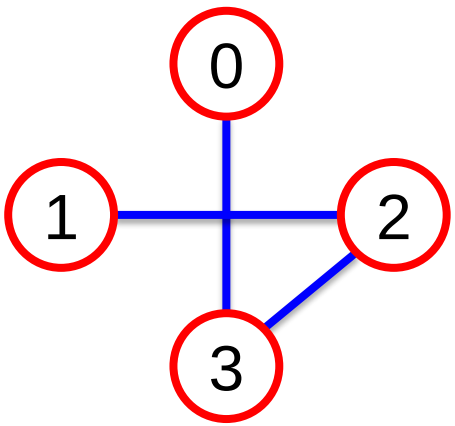


Terminology

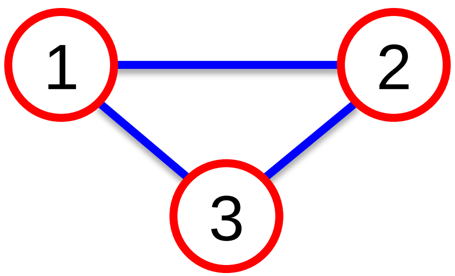
- **Subgraph**:
  - G' is a subgraph of G if V(G') $\subseteq$ V(G) and E(G') $\subseteq$ E(G)



Graph



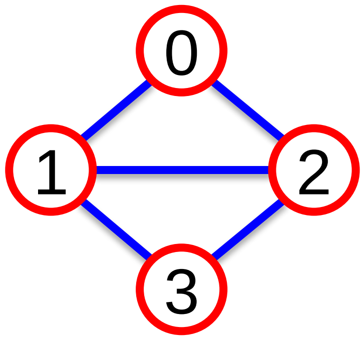
Subgraph



Subgraph

Terminology

- **Path**:
  - A path from **u** to **v** represents a sequence of vertices **u, i_1, i_2 , …, i_k, v,** such that (**u, i_1**), (**i_1 ,i_2**), …, (**i_k ,v**) are edges in the graph
- **Simple path:**
  - A simple path is a path in which all vertices except possibly the first and the last are distinct



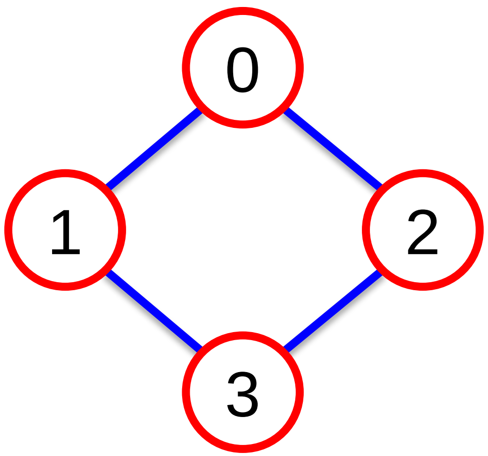
Sequence	Path?	Simple path?
0,1,3,2	Yes	Yes
0,2,0,1	Yes	No
0,3,2,1	No	No

Terminology

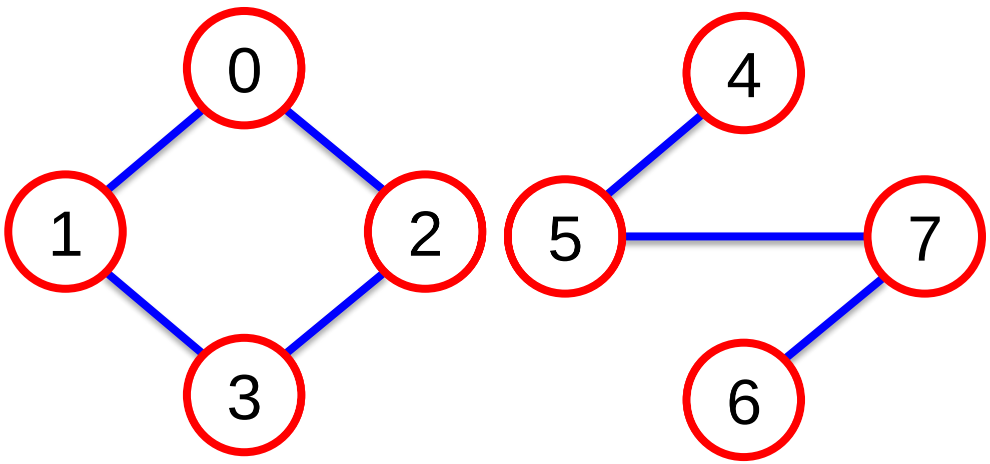
- **Cycle**:
  - A cycle is a simple path in which the first and the last vertices are the same
- Notes: if the graph is a directed graph, we usually add the prefix “directed” to the above terms:
  - Directed path
  - Directed simple path
  - Directed cycle

Terminology

- *Undirected graph* G is said to be **connected** iff for **every pair of distinct vertices u and v**, there is a **path from u to v** in G



Connected graph

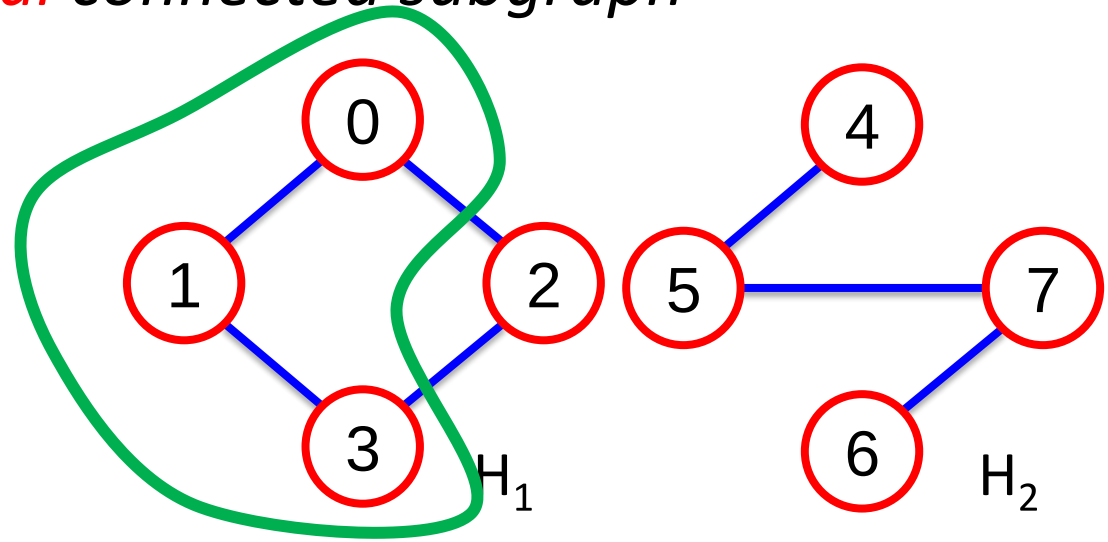


Not a connected graph

Terminology

- A **connected component**, H, of an undirected graph is a *maximal connected subgraph*

A connected subgraph, but not a maximal connected subgraph

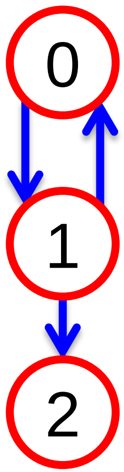


Graph with two connected components

- Tree:
  - A connected acyclic graph
  - n vertex connected graph with n-1 edges

Terminology

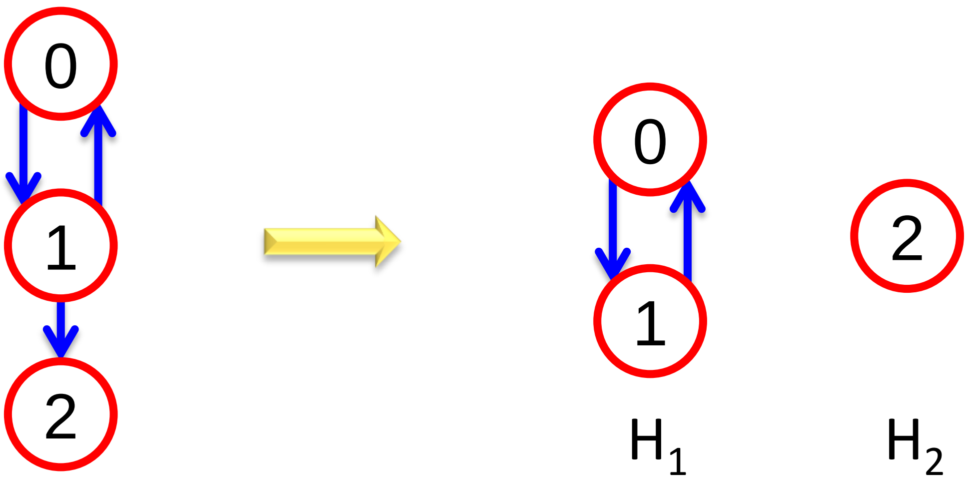
- *Directed graph* G is said to be **strongly connected** iff for **every pair of distinct vertices u and v**, there is a **directed path from u to v** and **also from v to u** in G.



Not a strongly connected digraph!
There is no directed path from 2 to 0

Terminology

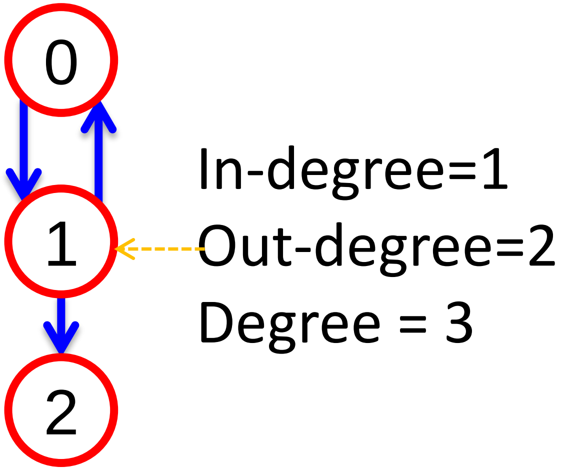
- A **strongly connected component** is a maximal subgraph that is strongly connected



Two strongly connected components

Terminology

- **Degree** of a vertex v:
  - The # of edges incident to v
- In a directed graph:
  - **In-degree** of v: # of edges for which v is the head
  - **Out-degree** of v: # of edges for which v is the tail
  - **Degree of v = in-degree + out-degree**



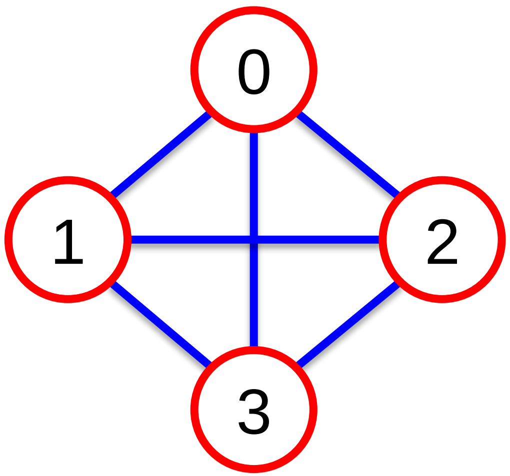
Outline

- Introduction to graphs (Sec. 6.1)
  - Definitions, terminologies
  - Representations

- Elementary graph operations (Sec. 6.2)
  - Depth first search, breadth first search, connected components, spanning trees

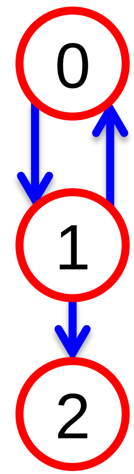
Adjacency Matrix

- A two dimensional array with the property that a[i][j] = 1 iff the edge (i,j) or <i,j> is in E(G)



$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Symmetric

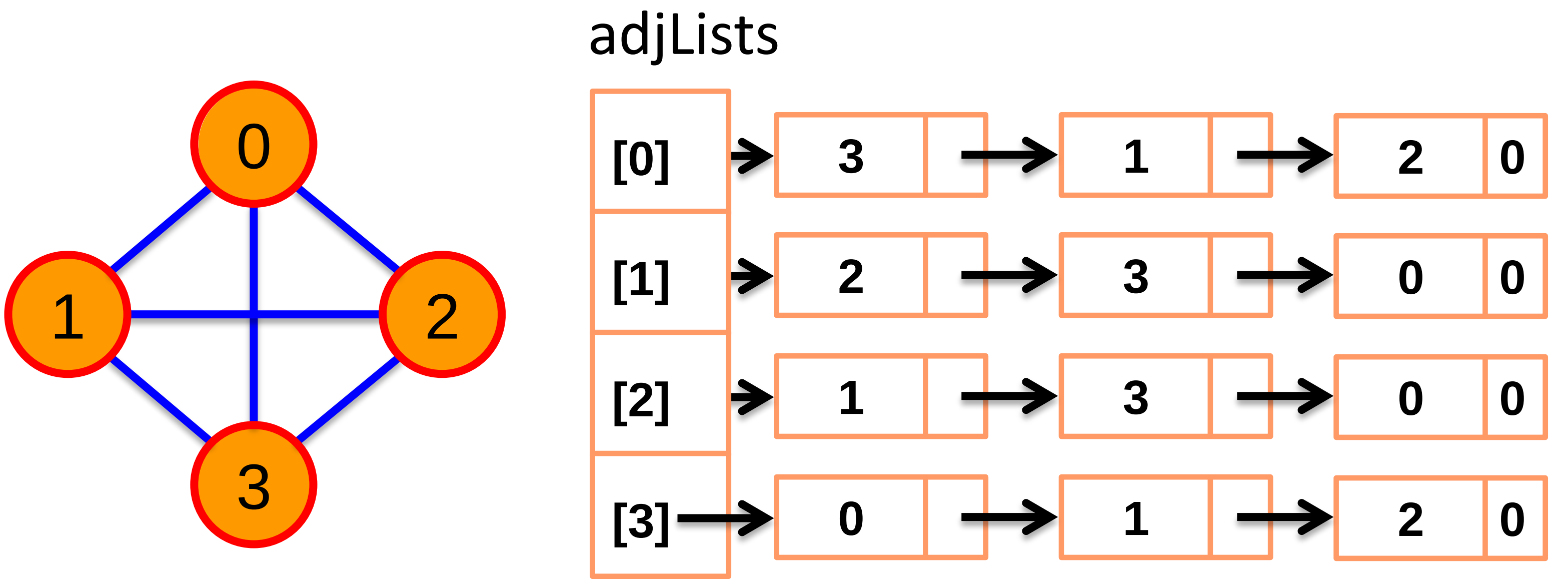


$$\begin{array}{c|c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

  - Row sum = degree or out-degree; column sum = in-degree
- Waste of memory & time, esp. when graph is sparse
  - Storage complexity = $O(n^2)$

Adjacency List

- Undirected graph: use a **chain** to represent each vertex and its adjacent vertices

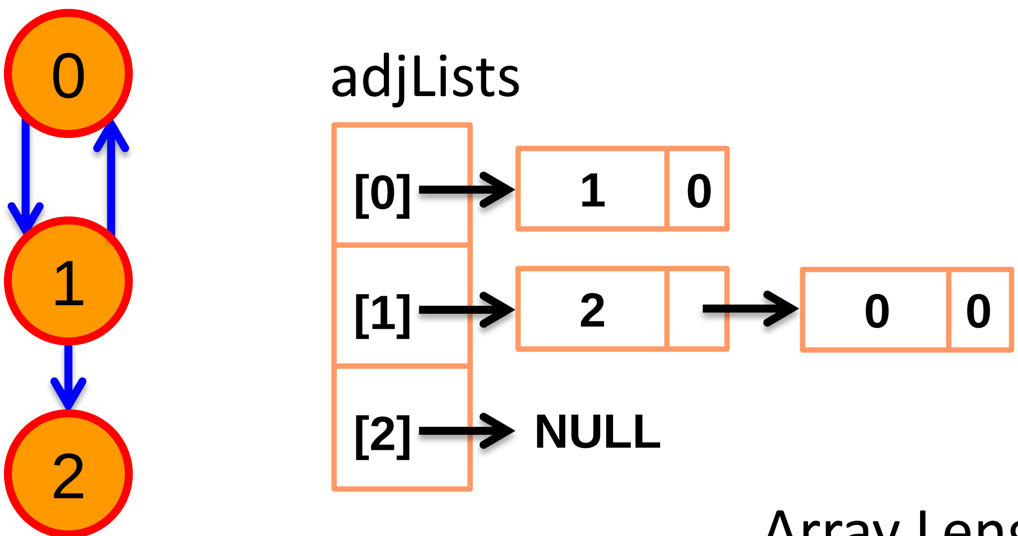


Array Length = n

# of chain nodes = 2e

Adjacency List

- Digraph: use a **chain** to represent each vertex and its adjacent to-vertices
  - Length of list = Out-degree of v

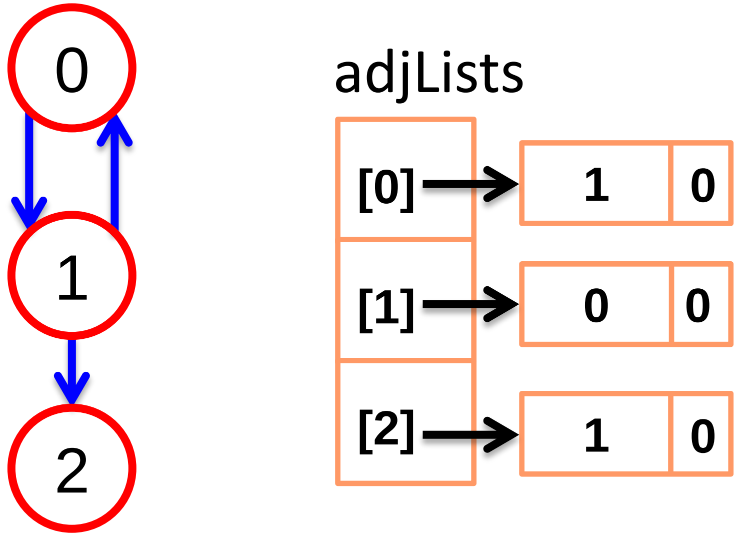


Array Length = n

# of chain nodes = e

Inverse Adjacency Lists

- Digraph: use a **chain** to represent each vertex and its adjacent from-vertices
  - Length of list = In-degree of v



Weighted Edges

- Edges of a graph sometimes have weights associated with them, e.g.,
  - Distance from one vertex to another
  - Cost of going from one vertex to an adjacent vertex
- We use additional field in each vertex to store the weight
- A graph with weighted edges is called a **network**

ADT of Graph

```
class Graph {
public:
  virtual ~Graph() {}
  bool IsEmpty() const{return n == 0};
  int NumberOfVertices() const{return n};
  int NumberOfEdges() const{return e};
  virtual int Degree(int u) const = 0;
  virtual bool ExistsEdge(int u, int v) const = 0;
  virtual void InsertVertex(int v) = 0;
  virtual void InsertEdge(int u, int v) = 0;
  virtual void DeleteVertex(int v) = 0;
  virtual void DeleteEdge(int u, int v) = 0;
protected:
  int n;     // number of vertices
  int e;     // number of edges
};
```

Implementation Notes

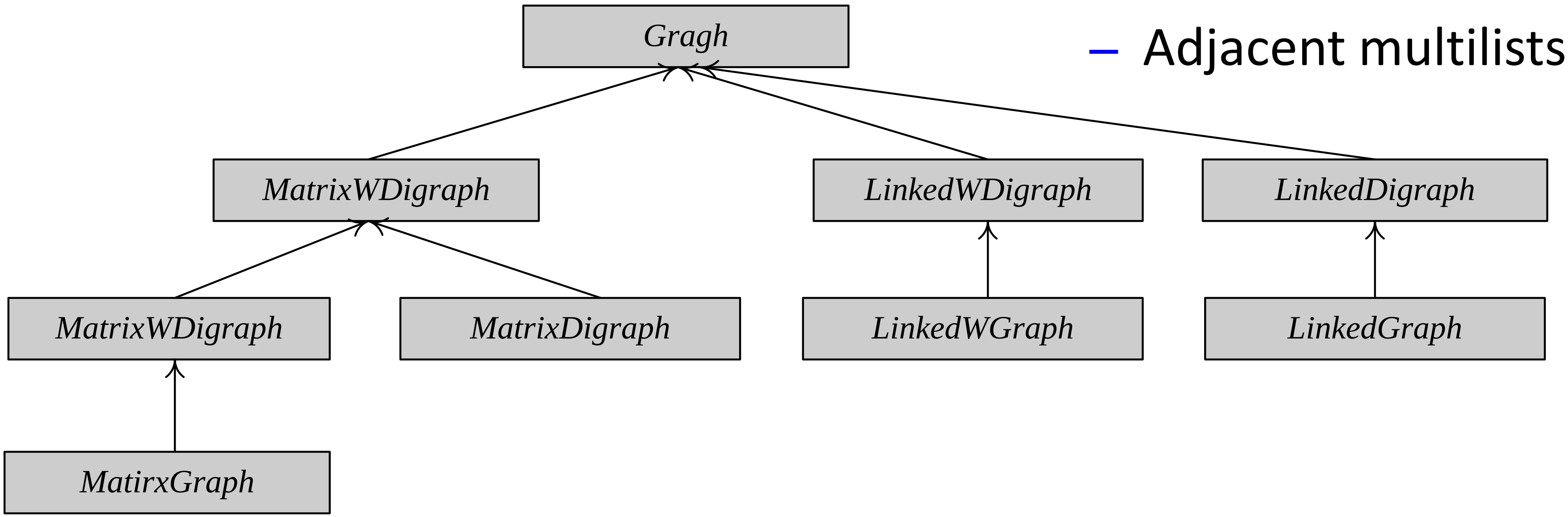
To accommodate various types of graphs, we make the following assumptions:

- Data type of edge weight is **double** (or you could use template to abstract it)
- Operations which are **independent** of specific graph representation are implemented inside graph class
- There is an **iterator** to visit adjacent vertices

Various graph classes can then inherit from the abstract class of Graph

A Possible Derivation Hierarchy

- 2 types:
  - Directed
  - Undirected
- 2 edge types
  - Weighted
  - Unweighted
- 4 representations:
  - Adjacency matrix
  - Adjacency list
  - Sequential list
  - Adjacent multilists



Note: The hierarchy shows only 2 representations

Example: LinkedGraph

```
void Graph::foo(void){
  // use iterator to visit adjacent vertices of v
  for (each vertex w adjacent to v)…
}
```

```
class LinkedGraph : public Graph {
public:
  // constructor
  LinkedGraph(const int vertices=0):n(vertices),e(0){
     adjLists = new Chain<int>[n];   };
  void foo(void) { ... }; // specialized foo()
  // more customized operations…
private:
  Chain<int> *adjLists; // adjacency lists
};
```

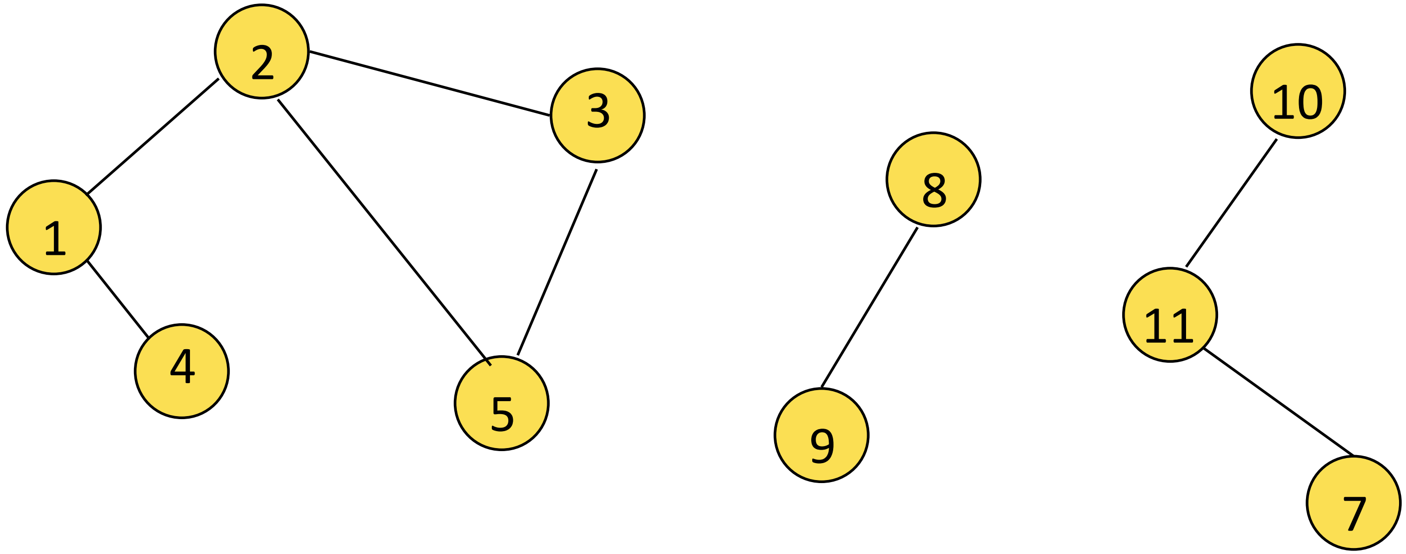
Outline

- Introduction to graphs (Sec. 6.1)
  - Definitions, terminologies
  - Representations

- Elementary graph operations (Sec. 6.2)
  - Depth first search, breadth first search, connected components, spanning trees

Graph Search Operation

- A vertex u is *reachable* from vertex v iff there is a path from v to u
- A graph search operation starts at a given vertex v and visits/marks every vertex reachable from v



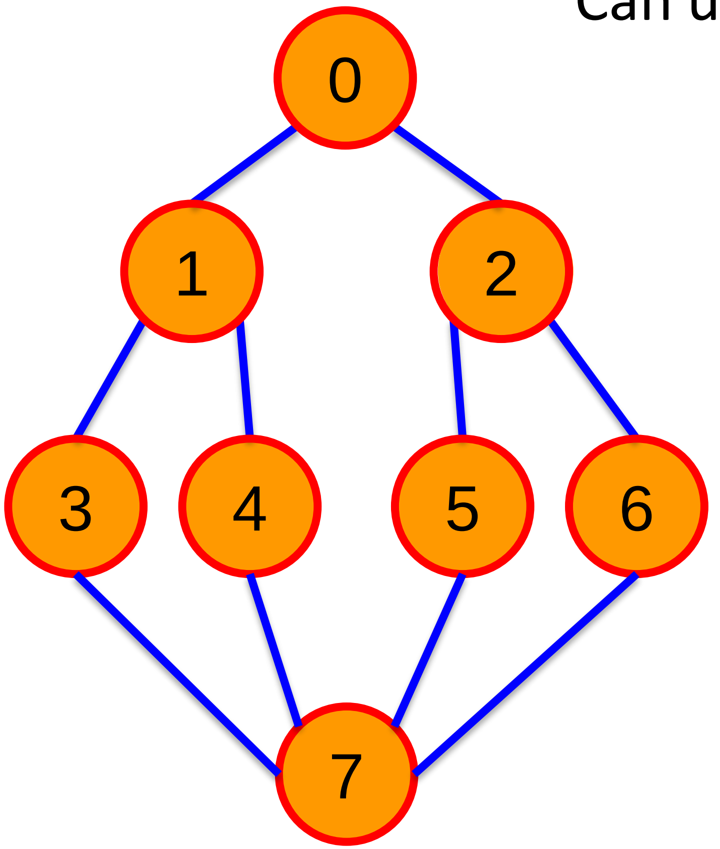
Graph Search Operation

- Many graph problems can be solved using a search operation
  - Path from one vertex to another
  - Is the graph connected?
  - Find a spanning tree
  - ...
- Commonly used search methods:
  - Depth-first search
  - Breadth-first search

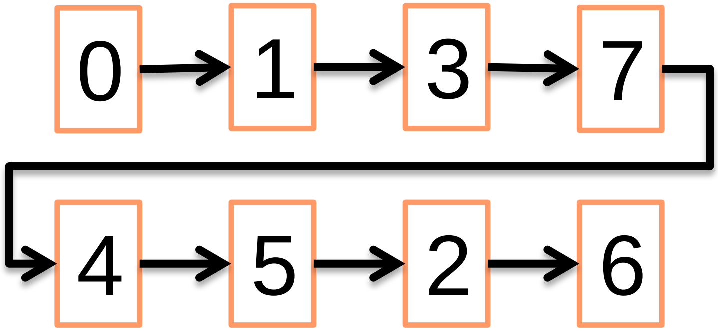
Depth-First Search (DFS)

- Starting from a vertex v, visit all vertices in G that are reachable from v, i.e., all vertices connected to v
  - Visit the vertex v → DFS(v)
  - For each vertex w adjacent to v, if w is not visited yet, then visit w → DFS(w)
  - If a vertex u is reached such that all its adjacent vertices have been visited, we go back to the last visited vertex
- The search terminates when no unvisited vertex can be reached from any of the visited vertices
  - All vertices reachable from the start vertex (including the start vertex) are visited

Depth-First Search (DFS)



Can use a stack



Note that there may have more than one order, depending on graph representation

What if G is a tree?

Recursive DFS Using Adjacency Lists

```
void Graph::DFS() { // public driver
   visited = new bool[n]; // data member of Graph
   fill(visited, visited + n, false);
   DFS(0); // start search at vertex 0
   delete [] visited;
}
void Graph::DFS(const int v){ // private worker
   // visit all previously unvisited vertices
   // that are reachable from v
   visited[v]=true;
   for(each vertex w adjacent to v)
     if(!visited[w]) DFS(w);
}
```

Non-Recursive DFS

```
void Graph::DFS(int v){
   visited = new bool[n]; // data member of Graph
   fill(visited, visited + n, false);
   Stack<int> s;
   s.Push(v);
   while(!s.IsEmpty()){
      u = s.Pop();
      if(!visited[u]){
         visited[u]=true;
         for(each vertex w adjacent to u)
           if(!visited[w]) s.Push(w);
      }
   }
}
```

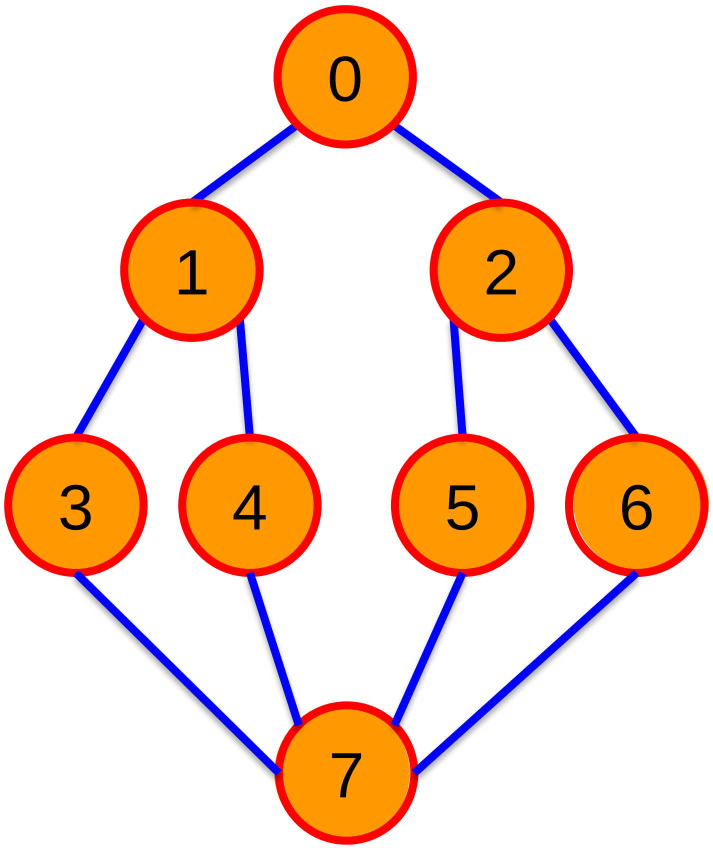
DFS Complexity

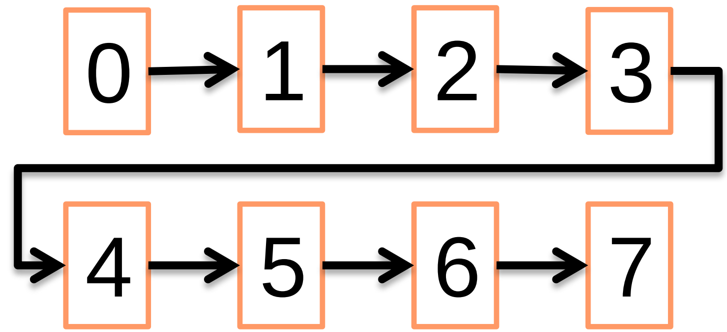
- Adjacency matrix
  - Time to determine all adjacent vertices to v: O(n)
  - At most n vertices are visited: $O(n \times n) = O(n^2)$
- Adjacency list
  - There are n+2e chain nodes
  - Each node in the adjacency list is examined at most once
    → time complexity = O(e)

Breadth-First Search (BFS)

- Starting from a vertex v
  - Visit the vertex v
  - Visit all unvisited vertices adjacent to v
  - Unvisited vertices adjacent to these newly visited vertices are then visited and so on
- Can use a queue to track the vertices

Breadth-First Search (BFS)





Note that there may be more than one order depending on the graph representation

What if G is a tree?

Non-Recursive BFS

```
void Graph::BFS(int v){
   visited = new bool[n]; // data member of Graph
   fill(visited, visited+n, false);
   Queue<int> q;   q.Push(v);
   visited[v]=true;
   while(!q.IsEmpty()){
      v = q.Front();    q.Pop();
      for(each vertex w adjacent to v){
         if(!visited[w]){
            q.Push(w);   visited[w]=true; }
      }
   }
   delete [] visited;
}
```

Time complexity is the same as DFS

Finding a Path from Vertex v to Vertex u

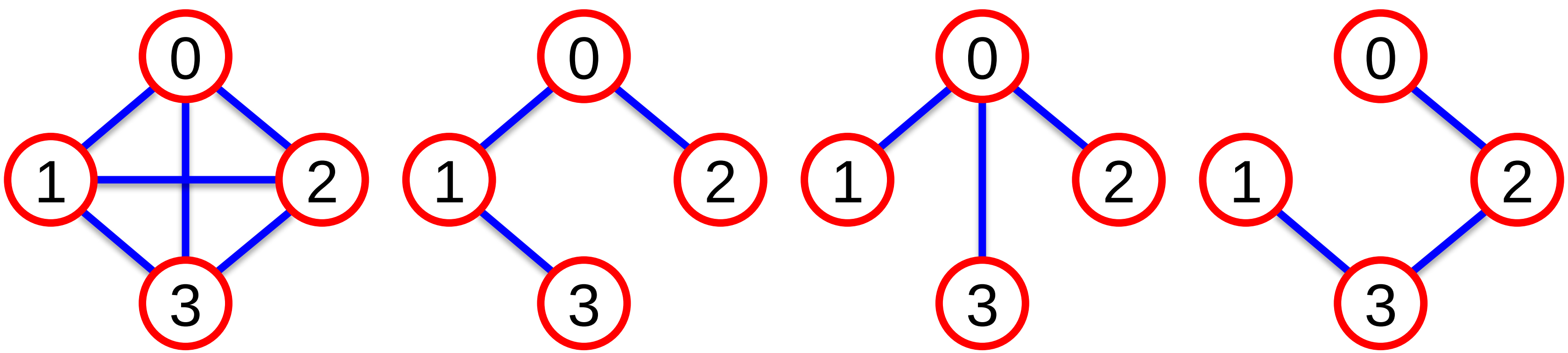
- Start a depth-first search at vertex v
- Terminate when vertex u is visited or when DFS ends (whichever occurs first)
- Time complexity:
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

Connected Components

- How to determine whether a graph is connected or not?
  - Call DFS or BFS once and check if there is any unvisited vertices; if Yes, then the graph is not connected
- How to identify connected components
  - Make a repeated calls to DFS or BFS
  - Each call will output a connected component
  - Start next call at an unvisited vertex

Spanning Trees

- Definition: any tree consisting solely of edges in E(G) and including all vertices of V(G)
- # of tree edges is **n-1**
- Add a non-tree edge will create a **cycle**

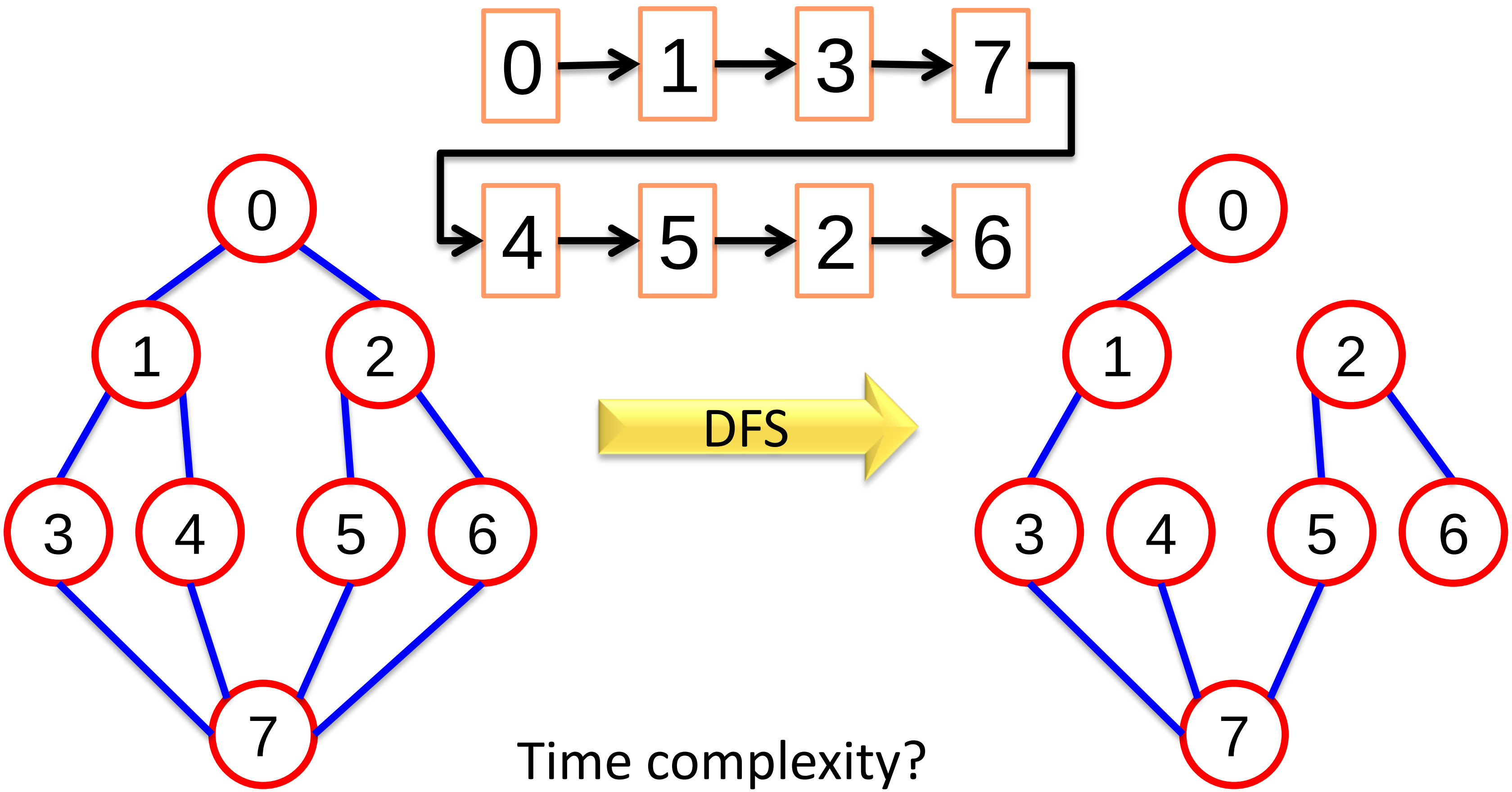


Complete graph

Possible spanning trees

DFS Spanning Tree

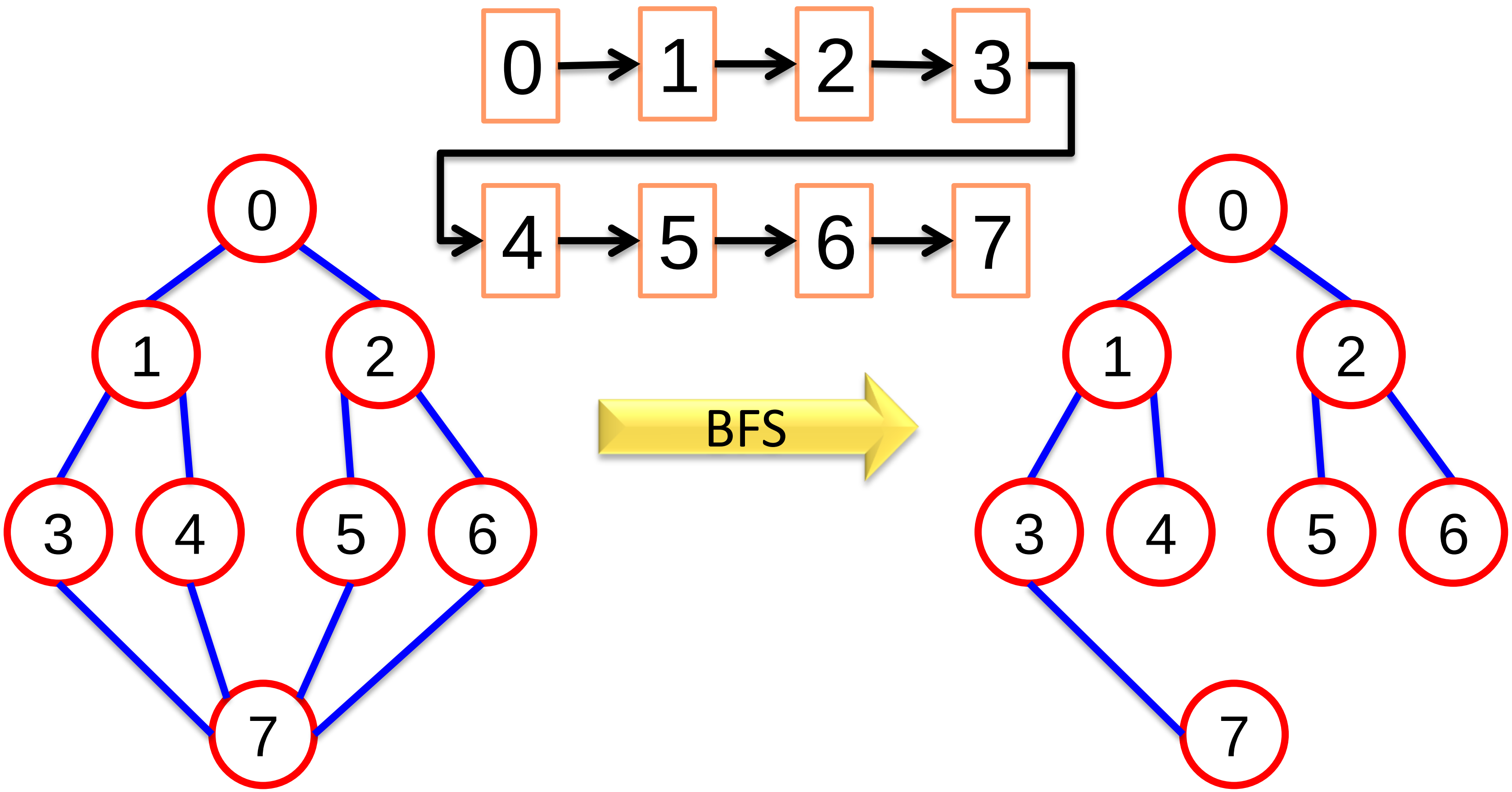
- Tree edges are those edges met during DFS traversal



Time complexity?

BFS Spanning Tree

- Tree edges are those edges met during BFS traversal



Summary

- Graphs are very important data structures
  - Terminologies and representations
- There are many operations associated with graphs
  - Depth first search, breadth first search, finding connected components, finding spanning trees
- Self-study topics
  - Graph representations: sequential lists, adjacency multilists
  - Graph operations: biconnected components