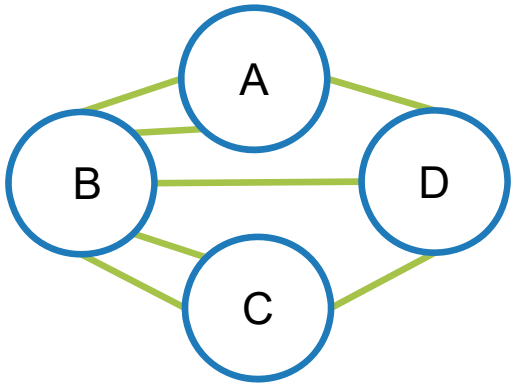
# Graphs

Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

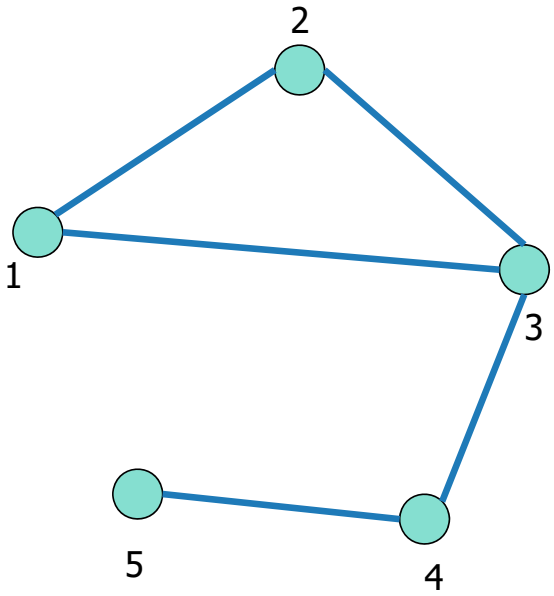
# Konigsberg Bridge Problem

- The first record (1736)
  - Solved by Euler
- Problem: Walk across all the bridges exactly once
- Formulate as a graph
- Prove: possible
  - Iff the degree of each vertex is even



# Undirected Graph

- Graph G=(V,E)
  - V = set of vertices
  - E = set of edges
- Undirected graph
  - E={(1,2),(1,3),(2,3),(3,4),(4,5)}
- Directed graph
  - <u,v> ≠ <v,u>
  - <u,v> → u is tail and v is head of edge
  - <5,4>




# Restrictions

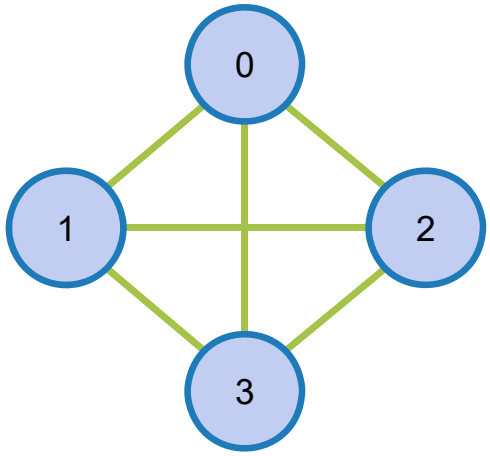
- ***Self edges*** and ***self loops*** are not permitted
  - Edges of the form (v, v) and <v, v> are not legal
- A graph may not have multiple occurrences of the same edge (***multigraph***)

# Terminology

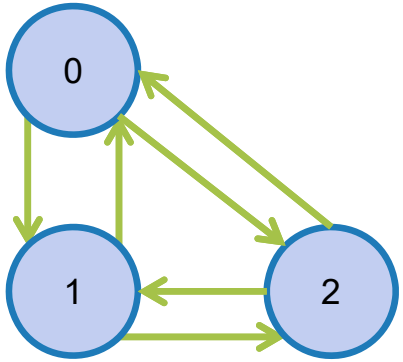
- For a graph with n vertices, the maximum # of edges:
  - $n(n-1)/2$ for undirected graph
  - $n(n-1)$ for directed graph
- Vertices u and v are **adjacent** if (u,v) ∈ E
  - Edge (u,v) is **incident** on vertices u and v
- <u,v>, u is **adjacent to** v and v is **adjacent from** u
  - Edge <u,v> is **incident** on vertices u and v

# Complete Graph

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

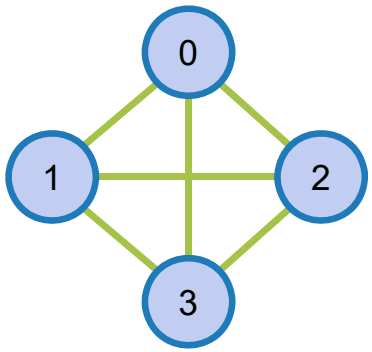



- Complete directed graph
  - Graph with n vertices has exactly $n(n-1)$ edges

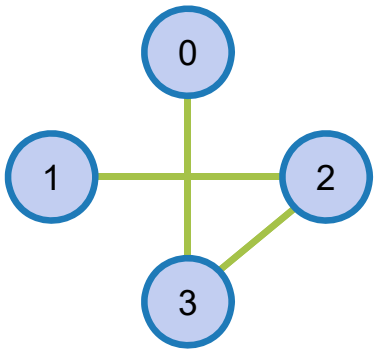


# Subgraph

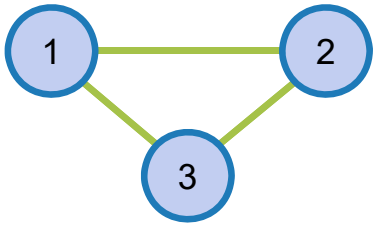
- G' is a subgraph of G
  - V(G') $\subseteq$ V(G) and E(G') $\subseteq$ E(G).




Graph



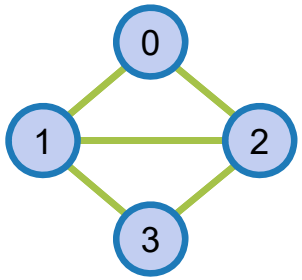

Subgraph




Subgraph

# Path

- A path from **u** to **v**
  - A sequence of vertices **u,$i_1$ ,$i_2$ ,…,$i_k$ , v**
  - (**u, $i_1$**), (**$i_1$ ,$i_2$**),…, (**$i_k$ ,v**) are edges in graph
- Simple path:
  - A path in which all vertices are distinct
    - Except possibly the first and the last




| Sequence | Path? | Simple path? |
|---|---|---|
| 0,1,3,2 | | |
| 0,2,0,1 | | |
| 0,3,2,1 | | |

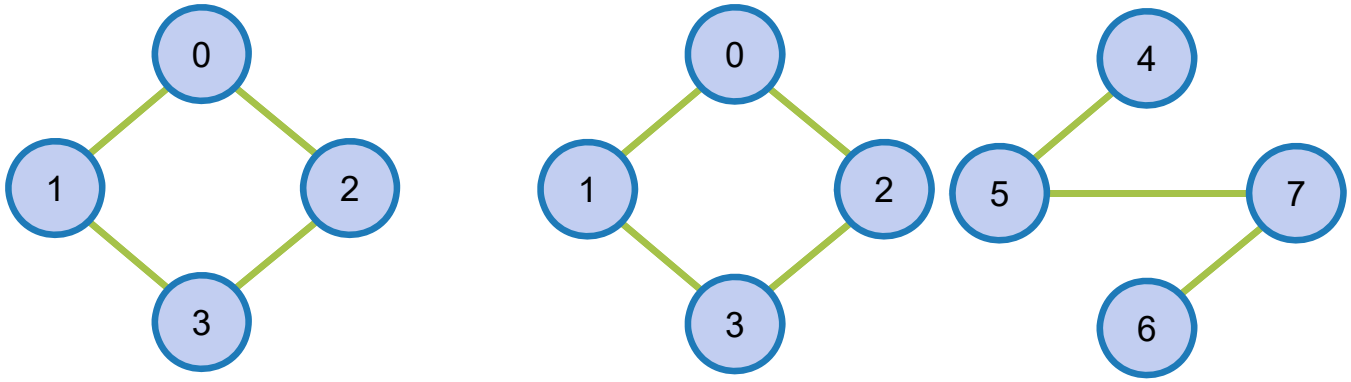
# Cycle

- A cycle is a simple path
  - The first and the last vertices are the same

- Notes: if the graph is a directed graph:
  - Directed path
  - Directed simple path
  - Directed cycle

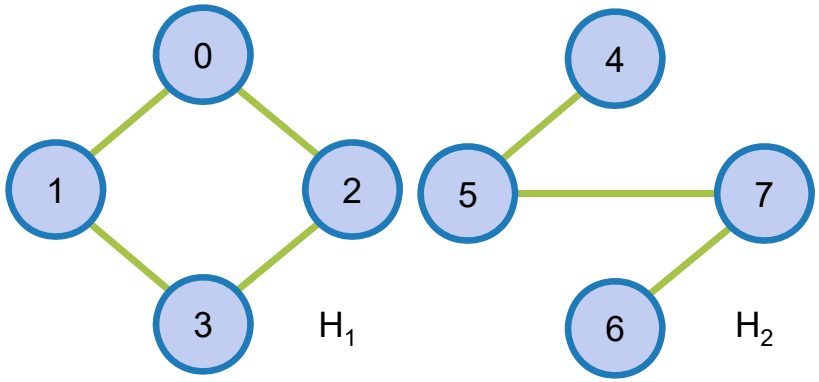
# Connected

- Undirected graph G is said to be connected
  - iff there is a path for every pair of distinct vertices



# Connected Component

- A maximal connected subgraph



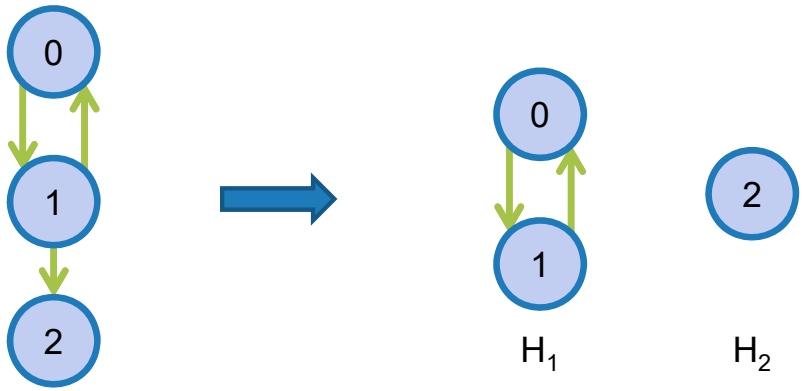

- Tree:
  - A connected acyclic graph

# Strongly Connected

- Directed graph G is strongly connected
  - iff there is a directed path for every pair of distinct vertices

# Strongly Connected Component

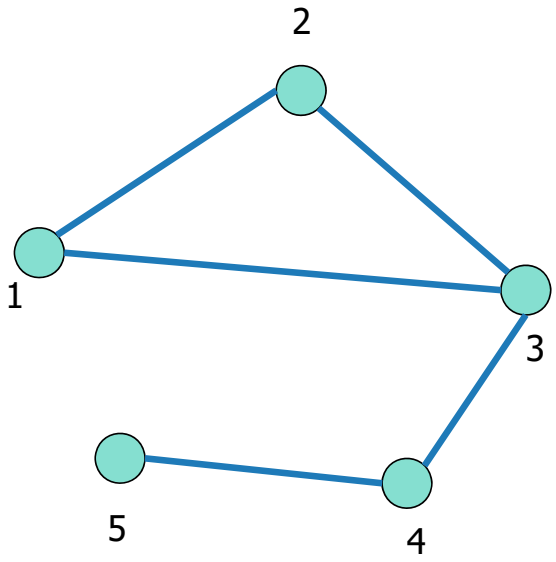
- A maximal subgraph that is strongly connected




Two strongly connected components

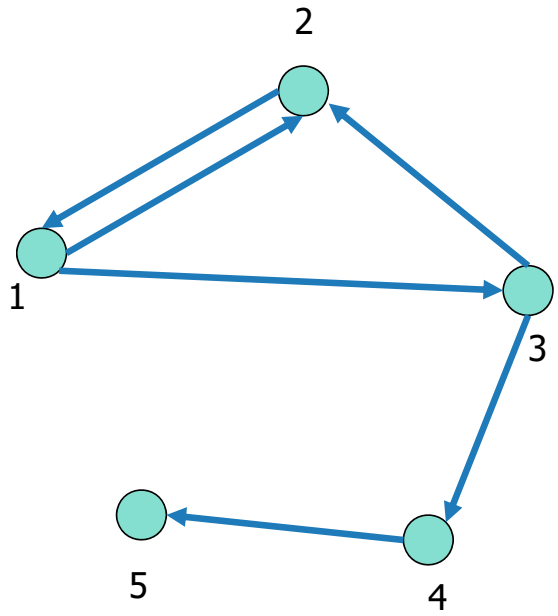
# Degree of Undirected Graph

- degree d(i) of node i
  - number of edges a node i involved
- degree sequence
  - [d(1),d(2),d(3),d(4),d(5)]
  - [2,2,3,2,1]



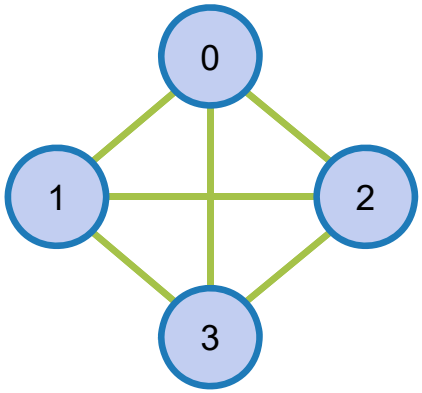
# Degree of Directed Graph

- in-degree $d_{in}(i)$ of node i
  - number of edges pointing
- out-degree $d_{out}(i)$ of node i
  - number of edges leaving node i
- Degree of v = in-degree + out-degree
- in-degree sequence
  - [1,2,1,1,1]
- out-degree sequence
  - [2,1,2,1,0]



# Adjacency Matrix

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

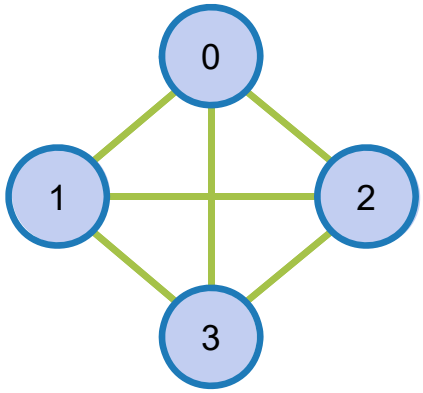


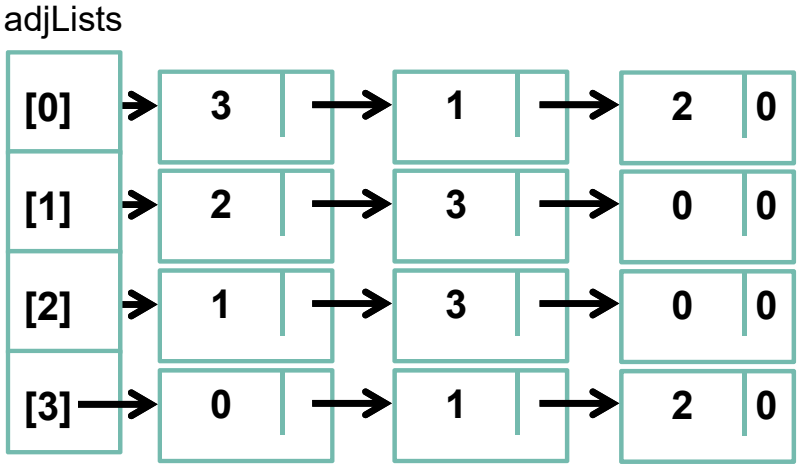


- Waste of memory when a graph is sparse
  - Storage

# Adjacency Lists in Undirected Graph

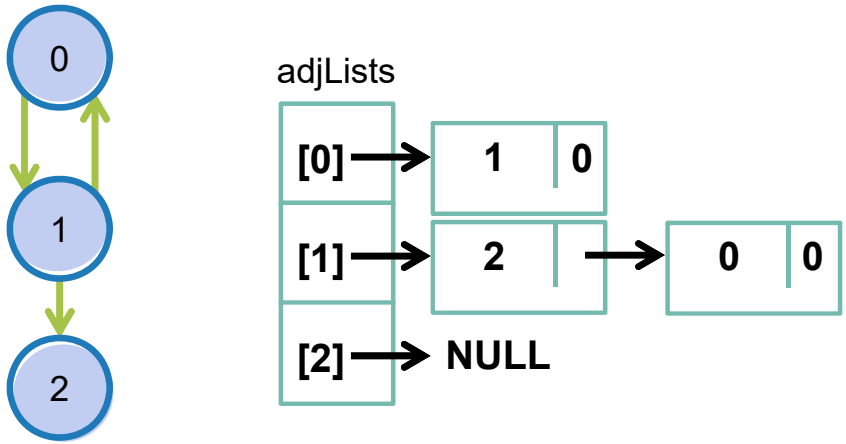
- Use a chain to represent each vertex and its **adjacent** vertices



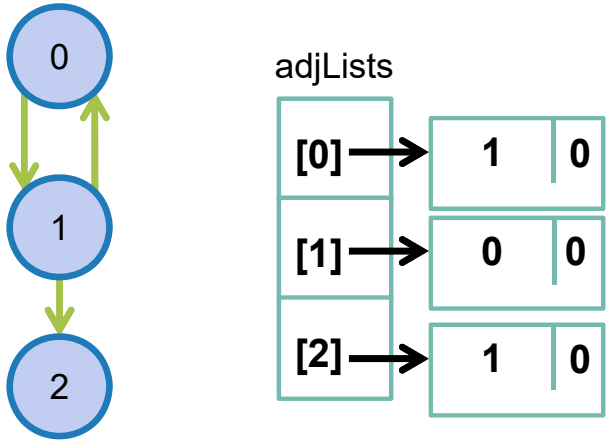
# Adjacency Lists in Directed Graph

- Use a chain to represent each vertex and its adjacent to vertices
  - Length of list = Out-degree of v



# Inverse Adjacency Lists in Directed Graph

- 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
  - Distance from one vertex to another.
  - Cost of going from one vertex to an adjacent vertex.
- Use additional field in each edge to store the weight
- A graph with weighted edges is called a network

# How Many Kinds of Graphs ?

- 2 types:
  - Directed
  - Undirected
- 2 edge types
  - Weighted
  - Unweighted
- 4 Representations:
  - Adjacent matrix
  - Adjacent lists
  - Sequential lists
  - Adjacent multilists

# ADT: Graph

```
class Graph
{// object: A nonempty set of vertices and a set of undirected edges.
public:
  virtual ~Graph() {}                          // virtual destructor
  bool IsEmpty() const{return n == 0};         // return true iff graph has no vertices
  int NumberOfVertices() const{return n}; // return the # of vertices
  int NumberOfEdges() const{return e};         // return the # of edges
  virtual int Degree(int u) const = 0;         // return the degree of a vertex
  virtual bool ExistsEdge(int u, int v) const = 0; // check the existence of edge
  virtual void InsertVertex(int v) = 0;              // insert a vertex v
  virtual void InsertEdge(int u, int v) = 0;         // insert an edge (u, v)
  virtual void DeleteVertex(int v) = 0;              // delete a vertex v
  virtual void DeleteEdge(int u, int v) = 0;         // delete an edge (u, v)
  // More graph operations…
protected:
  int n; // number of vertices
  int e; // number of edges
};
```

# Implementation Notes

- Several assumptions:
  - Data type of edge weight is **double**
    - Or represented as a template parameter
- Operations are **independent** of specific graph representation
- The **iterator** is used to visit adjacent vertices

# 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];
  }
  // more customized operations…
private:
  Chain<int> *adjLists          // adjacency lists
};
```

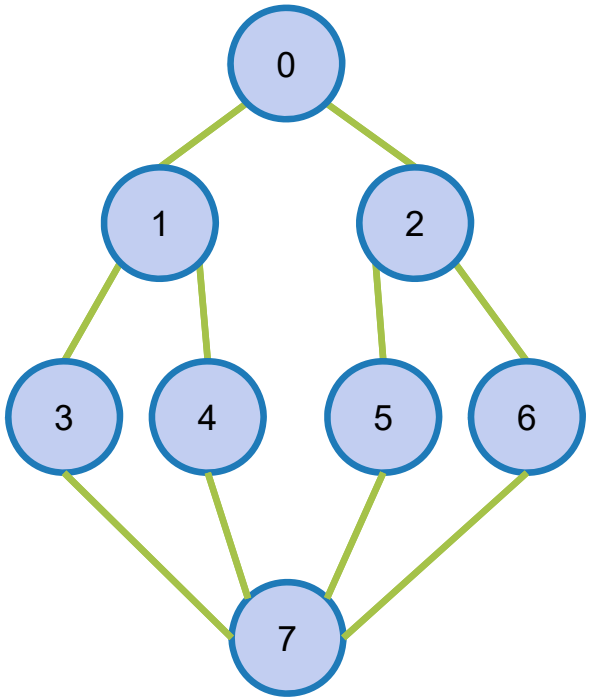
# Graph Operations

- Graph traversal
  - Depth-first search
  - Breadth-first search
- Connected components
- Spanning trees

# Depth-First Search (DFS)

- Starting from a vertex 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.

# Example of DFS



# Recursive DFS

```
void Graph::DFS(void){
   visited = new bool[n]; // this is a data member of Graph
   fill(visited, visited+n, false);
   DFS(0); // start search at vertex 0
   delete [] visited;
}

void Graph::DFS(const int v){
   // visit all previously unvisited vertices that are adjacent to v
   output(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]; // this is a data member of Graph
   fill(visited, visited+n, false);
   Stack<int> s;          // declare and init a stack
   s.Push(v);
   while(!s.IsEmpty()){
      v = s.Top(); s.Pop();
      if(!visited[v]){
         output(v);
         visited[v]=true;
         for(each vertex w adjacent to v)
           if(!visited[w]) s.Push(w);
      }
   }
}
```

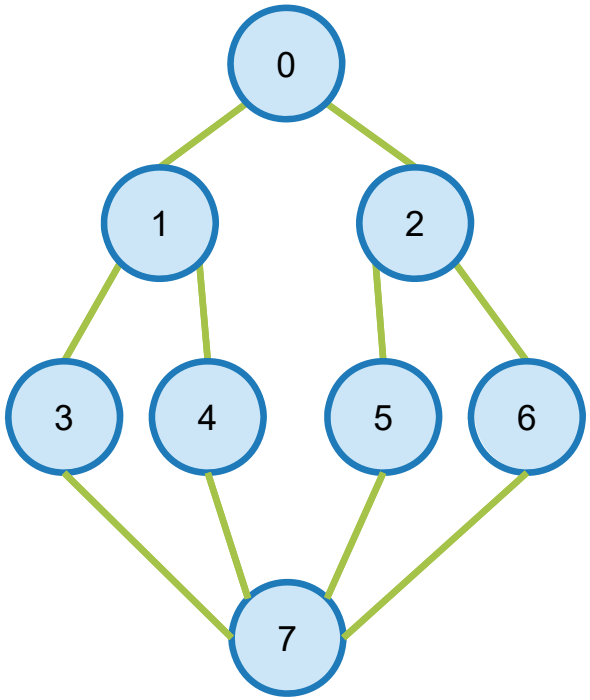
# DFS Complexity

- Adjacency matrix
  - Time to determine all adjacent vertices: O(n)
  - At most n vertices are visited: $O(n \cdot n) = O(n^2)$
- Adjacency lists
  - There are n+2e chain nodes
  - Each node in the adjacency lists 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

# Example of BFS



# BFS: Implementation

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

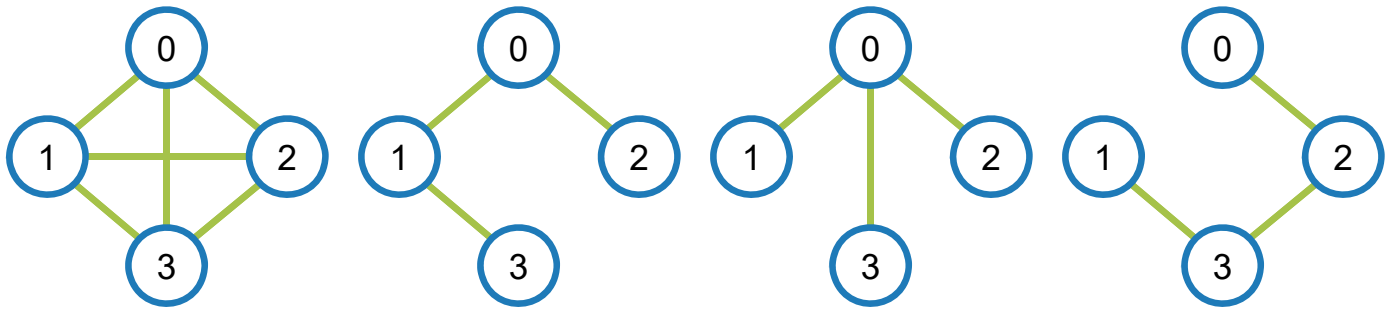
Time complexity is the same as DFS

# Connected Components

- Determine whether a graph is connected
  - Call DFS or BFS once
  - Check if there is any unvisited vertices
  - Yes, then the graph is not connected.
- How to identify connected components
  - Call DFS or BFS repeatedly
  - Each call will output a connected component
  - Start next call at an unvisited vertex

# Spanning Trees

- Any tree consists of solely of edges in E(G) and including all vertices of V(G)
  - Number of tree edges is n-1.
  - Add a non-tree edge will create a cycle

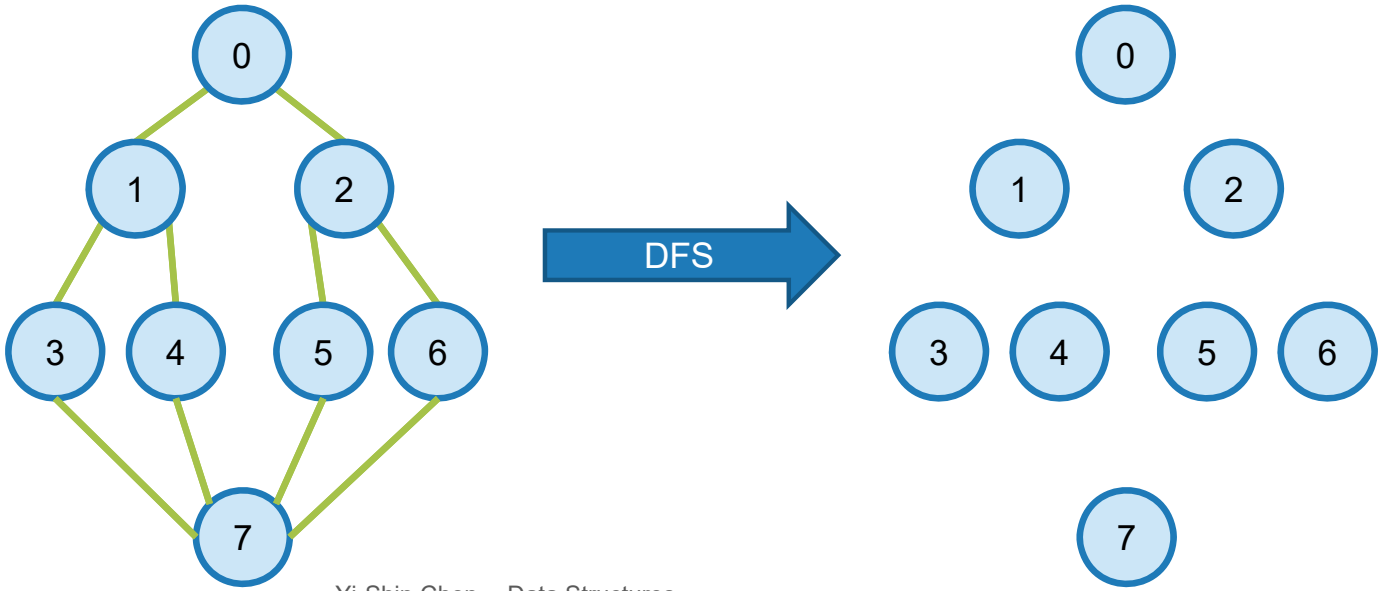



Complete graph

Possible spanning trees

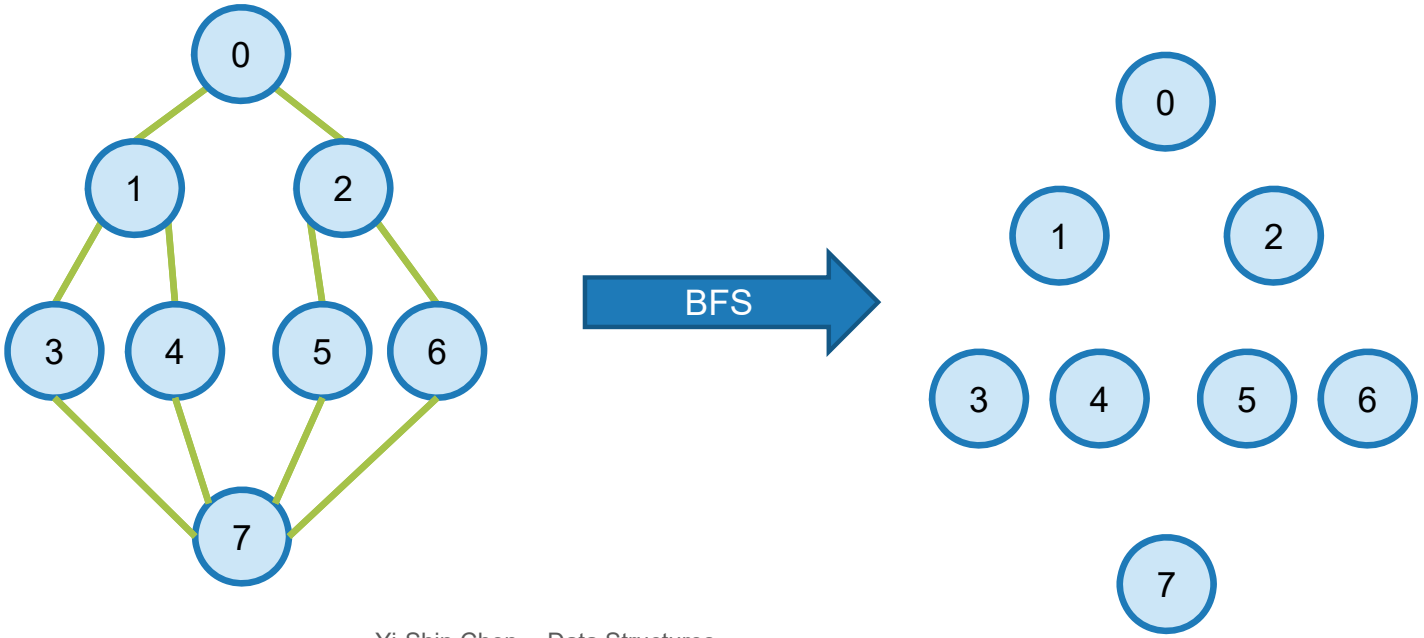
# DFS Spanning Tree

- Tree edges are those edges met during the traversal



# BFS Spanning Tree

- Tree edges are those edges met during the traversal

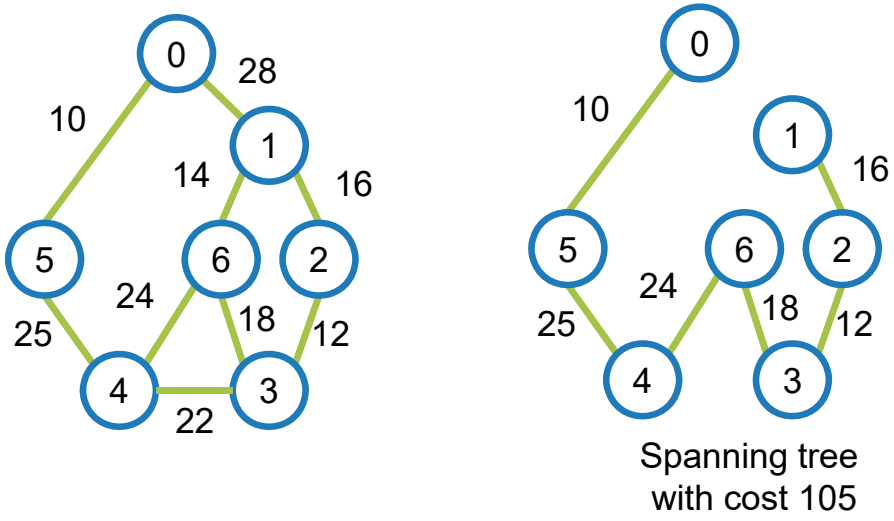


# Self-Study Topics

- Graph representations
  - Sequential lists
  - Adjacency multilists
- Graph operation
  - Biconnected components

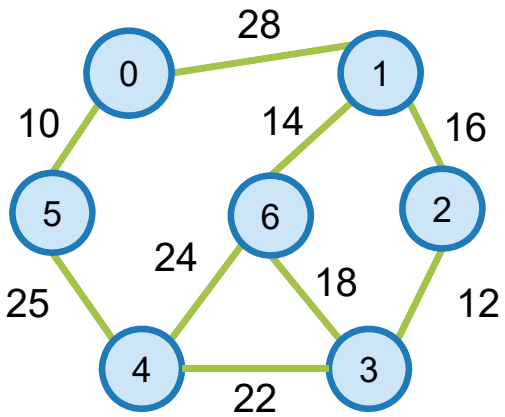
# Minimum-Cost Spanning Trees

- For a weighted undirected graph
  - Find a spanning tree with least cost of the sum of the edge weights
- Three greedy algorithms:
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's Algorithm

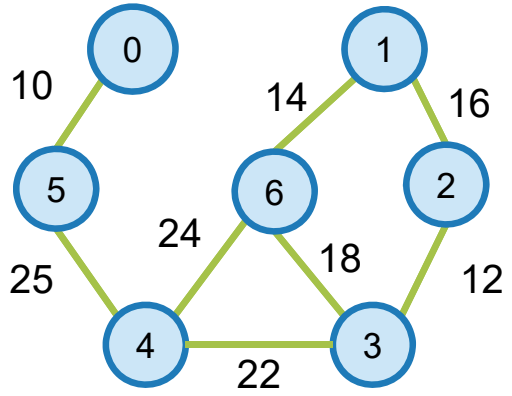


# Kruskal's Algorithm

- Idea: Add edges with minimum cost one at a time
  - Step 1: Find an edge with minimum cost
  - Step 2: If it creates a cycle, discard the edge
  - Step 3: Repeat step 1 and 2 until we find n-1 edges




Connected graph



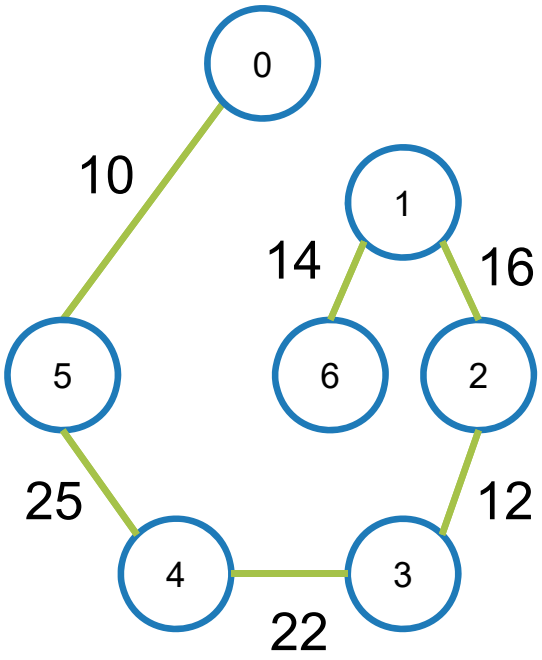

Spanning tree with cost 99

# Kruskal's Algorithm

```
Kruskal's algorithm
1.  T = ϕ
2.  While((T contains less than n-1 edges)&&(E is not empty)){
3.    choose an edge (v,w) from E of lowest cost;
4.    delete (v,w) from E
5.    if((v,w) does not create a cycle) add (v,w) to T;
6.    else discard (v,w)
7.  }
8.  If(T contains less than n-1 edges)
9.    cout << "there is no spanning tree!" <<endl;
```

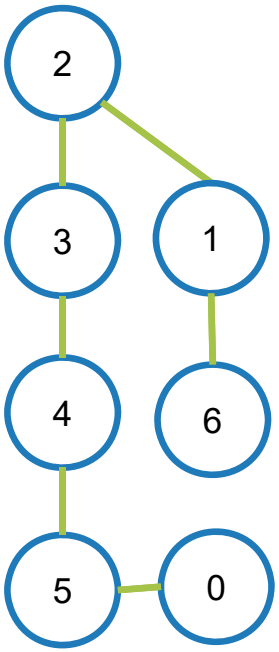
- Step 3 & 4: use **min heap** to store edge cost.
- Step 5: use **set representation** to group all vertices in the same connected component into a set.
  - For an edge (v,w) to be added, if vertices are in the same set, discard the edge, else merge two sets

# Running Example




Spanning tree with cost 99

Disjoint set

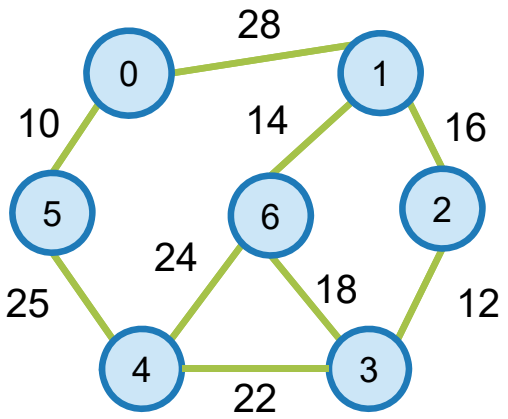


# Time Complexity

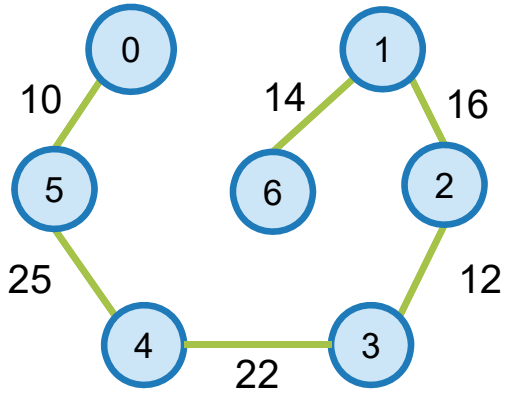
- Min heap:
  - Step 3&4 : O(log e)
- Set:
  - Step 5: O(a(e))
- At most execute e-1 rounds:
  - (e-1)·(log e + a(e)) = O(e log e)

# Prim's Algorithm

- Idea: Add edges with minimum edge weight to tree
  - The set of selected edges form a tree
  - Step 1: Start with a tree T contains a single arbitrary vertex
  - Step 2: Add a least cost edge (u,v) to T, $T \cup (u, v)$ is still a tree
  - Step 3: Repeat step 2 until T contains n-1 edges




Connected graph



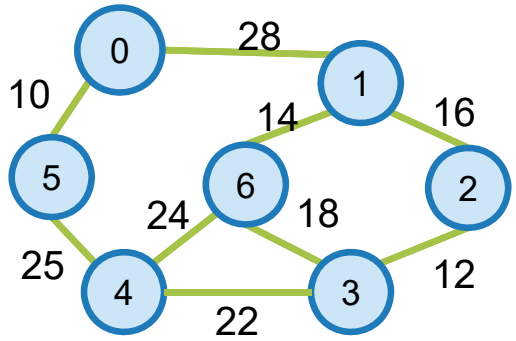

Spanning tree with cost 99

# Prim's Algorithm

```
Prim's algorithm
1. V(T) = {0} // start with vertex 0
2. for(T=ϕ ; T contains less than n-1 edges; add (u,v) to T){
3.   Let (u,v) be a least cost edge such that u⊆V(T) and v⊈V(T);
4.   if(there is no such edge) break;
5.   add v to V(T);
6. }
7. If(T contains fewer than n-1 edges)
8.    cout << "there is no spanning tree!" <<endl;
```

- Step 3: use a **near-to-tree** data structure
  - Create an array to record the nearest distance of vertices to T
  - Only vertices not in V(T) and adjacent to T are recorded

# Running Example



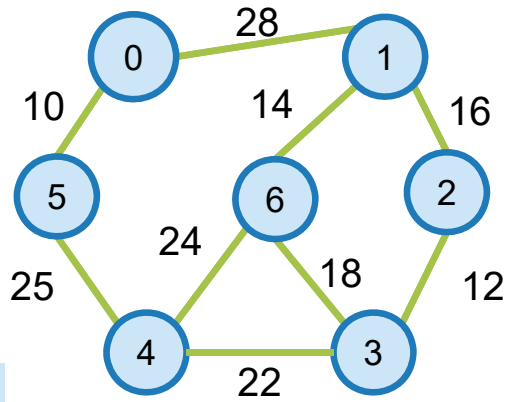

| near-to-tree | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| V(T)={0} | * | 28 | ∞ | ∞ | ∞ | 10 | ∞ |
| V(T)={0,5} | | | | | | | |
| V(T)={0,5,4} | | | | | | | |
| V(T)={0,5,4,3} | | | | | | | |
| V(T)={0,5,4,3,2} | | | | | | | |
| V(T)={0,5,4,3,2,1} | | | | | | | |
| V(T)={0,5,4,3,2,1,6} | | | | | | | |

# Time Complexity

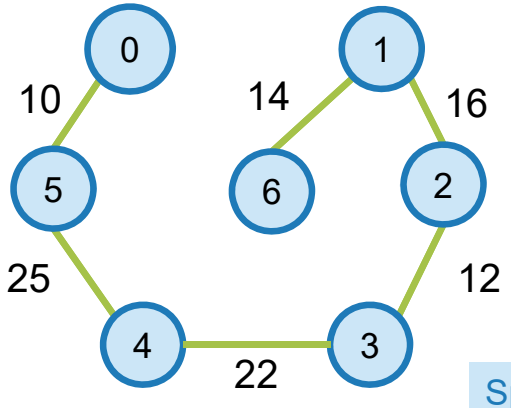
- Near-to-tree
  - Step 3 : O(n)
- At most execute n rounds: $O(n^2)$

# Sollin's (Borůvka's) Algorithm

- Idea: Select several edges at each stage
  - Step 1: Start with a forest that has n spanning trees
  - Step 2: Select one minimum cost edge for each tree
  - Step 3: Delete multiple copies of selected edges and if two edges with the same cost connecting two trees, keep only one of them
  - Step 4: Repeat until we obtain only one tree.



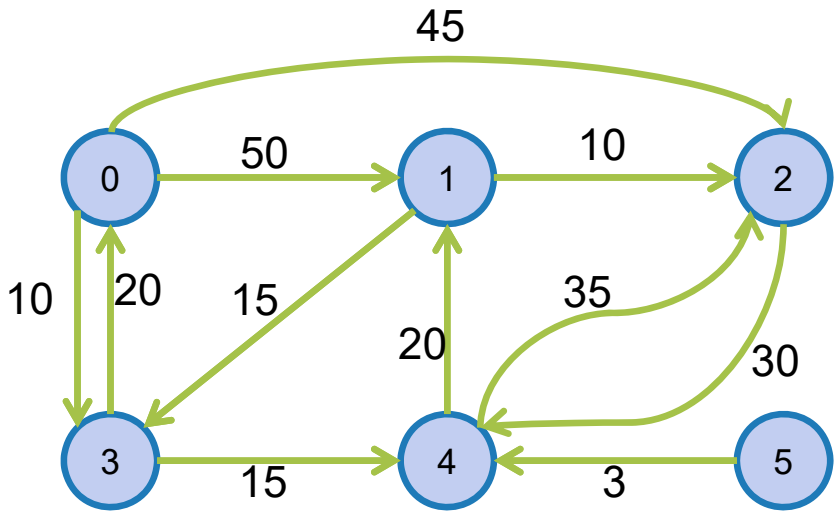

Connected graph




Spanning tree with cost 99

# Single Source Shortest Paths

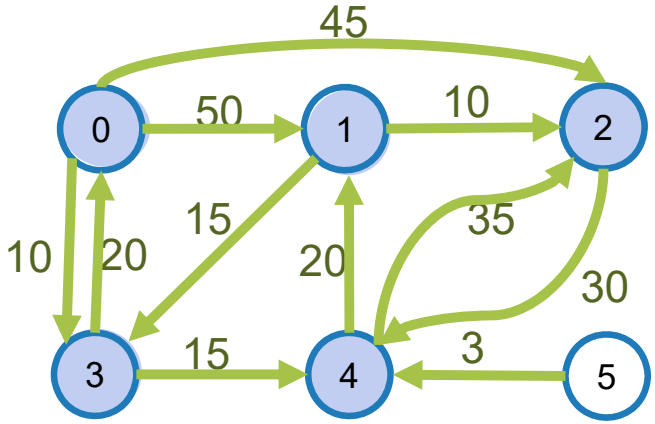
- Single source/all destinations problem
  - Given a digraph with nonnegative edge costs and a source vertex *v*, compute a shortest path from *v* to each of the remaining vertices



# Dijkstra's Algorithm

- Use a set S:
  - Store the vertices whose shortest path have been found
- An array *dist*:
  - Store the shortest distances from source v to all visited vertices
  - When a new vertex *w* is visited, update *dist*
    - $dis[w] = min(dist[u] + length(\langle u, w \rangle), dist[w])$
    - u is the previously visited vertex adjacent to w

# Example for Dijkstra's Algorithm



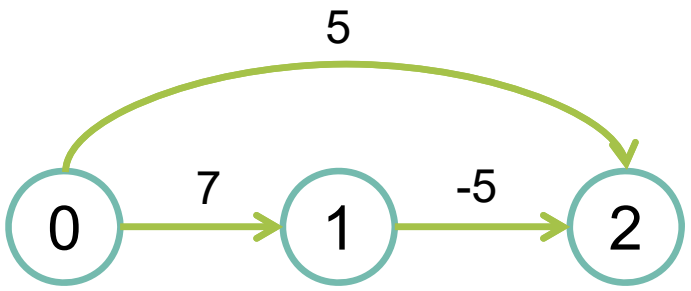

| Visited | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| {0} | | | | | | |
| {0, 3} | | | | | | |
| {0, 3, 4} | | | | | | |
| {0, 3, 4, 1} | | | | | | |
| {0, 3, 4, 1, 2} | | | | | | |

# Dijkstra's Algorithm

```
1.  void MatrixWDigraph::ShortestPath(const int n, const int v)
2.  { // dist[j], 0 ≤ j < n, stores the shortest path from v to j
3.    // length[i][j] stores length of edge <i, j>
4.    for(int i=0; i<n; i++){ s[i]=false; dist[i]=length[v][i];}
5.    s[v] = true;
6.    dist[v] = 0;
7.    // find n - 1 paths starting from v
8.    for(int i=0; i<n-1 ;i++){
9.      // Choose a vertex u, such that dist[u]
        // is minimum and s[u] = false
10.     int u = Choose(n);
11.     s[u] = true;
12.     for(int w=0; w<n; w++)
13.     if(!s[w] && dist[u] + length[u][w] < dist[w])
14.        dist[w] = dist[u] + length[u][w];
15.    } // end of for (i = 0; ...)
16. }
```

# Digraph with Negative Costs

- This algorithm can apply to digraph with negative cost edges
  - Restriction: The digraph MUST NOT contain cycles of negative length




Digraph with a negative cost edge

# All-Pairs Shortest Paths

- Apply single source shortest path to each of n vertices
- Floyd-Warshall's algorithm
  - Dynamic programming approach

# Dynamic Programming

- Divide-and-conquer approach
- Usually applied to optimization problems
- Improve the inefficient problems
  - The same recursive call is called repeatedly

# Dynamic Programming

- Divide-and-conquer approach
- Usually applied to optimization problems
- Improve the inefficient problems
  - The same recursive call is called repeatedly
- Used when sub-problems share sub-sub-problems
- “Programming” refers to a tabular method
  - Remember the solutions
- Compute solution in a bottom-up fashion

# All-Pairs Shortest Paths

- Apply single source shortest path to each of n vertices
- Floyd-Warshall's algorithm
  - Dynamic programming approach
  - $A^{-1}[i][j]$: the length[i][j]
  - $A^{k}[i][j]$: the length of the shortest path from i to j going through no intermediate vertex of index greater than k
  - $A^{k}[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$
  - $A^{n-1}[i][j]$: the length of the shortest i-to-j path in G

# Intuition of Floyd-Warshall's Algorithm

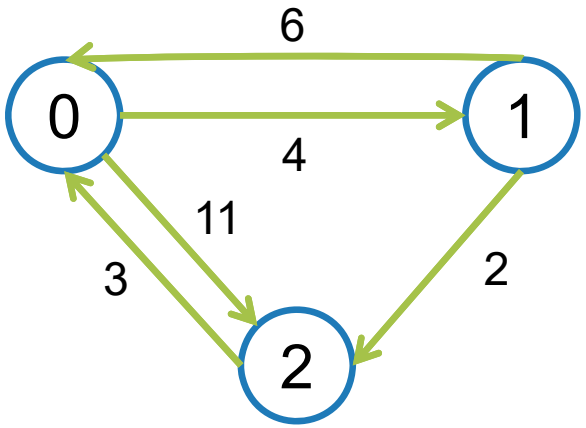
- There are only two possible paths for $A^k[i][j]$
  - The path dose not pass the $k^{th}$ vertex
  - The path dose pass the $k^{th}$ vertex

- $A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$

# Floyd-Warshall's Algorithm

```
1. void MatrixWDigraph::AllLengths(const int n)
2. {// length[n][n] stores edge length between
    // adjacent vertices
3.  // a[i][j] stores the shortest path from i to j
4.  for (int i = 0; i<n; i++)
5.   for (int j = 0; j<n; j++)
6.       a[i][j]= length[i][j];
7.
8.  // path with top vertex index k
9.  for (int k= 0; k<n; k++)
10.  // all other possible vertices
11.  for (int i= 0; i<n; i++)
12.   for (int j= 0; j<n; j++)
13.    if((a[i][k]+a[k][j])<a[i][j])
14.      a[i][j] = a[i][k] + a[k][j];
15. }
```

# Example of Floyd-Warshall's Algorithm




| $A^2$ | 0 | 1 | 2 |
|---|---|---|---|
| **0** | 0 | 4 | 6 |
| **1** | 5 | 0 | 2 |
| **2** | 3 | 7 | 0 |

# Floyd-Warshall's Algorithm

```
1. void MatrixWDigraph::AllLengths(const int n)
2. {// length[n][n] stores edge length between
    // adjacent vertices
3.  // a[i][j] stores the shortest path from i to j
4.  for (int i = 0; i<n; i++)
5.   for (int j = 0; j<n; j++)
6.       a[i][j]= length[i][j];
7.
8.  // path with top vertex index k
9.  for (int k= 0; k<n; k++)
10.  // all other possible vertices
11.  for (int i= 0; i<n; i++)
12.   for (int j= 0; j<n; j++)
13.    if((a[i][k]+a[k][j])<a[i][j])
14.      a[i][j] = a[i][k] + a[k][j];
15. }
```

# Transitive Closure

- Determine if there is a path from $i$ to $j$ in a graph with unweighted edges
  - Only positive path lengths → transitive closure
  - Only nonnegative path lengths → reflexive transitive closure
- The transitive closure matrix $A^+$:
  - $A^+[i][j] = 1$ if there is a path of length > 0 from i to j in the graph; otherwise, $A^+[i][j] = 0$
- The reflexive transitive closure matrix $A^*$:
  - $A^*[i][j] = 1$ if there is a path of length >= 0 from i to j in the graph; otherwise, $A^*[i][j] = 0$
- Use Floyd-Warshall's algorithm!
  - $A^k[i][j] = A^{k-1}[i][j]$ || ( $A^{k-1}[i][k]$ && $A^{k-1}[k][j]$ );

# Transitive Closure Example


| $A^+$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 |

Transitive closure matrix

| $A^*$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 1 |

Reflexive transitive closure matrix

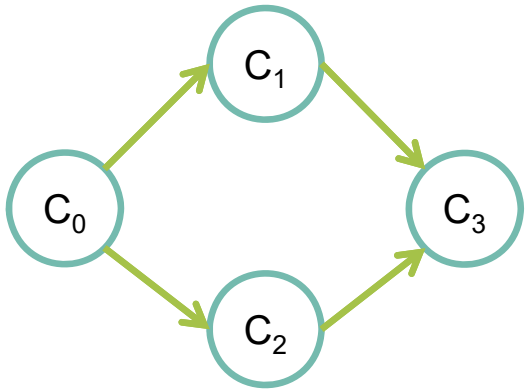
# Activities

- Projects can be subdivided into several subprojects
  - Each subproject is called activity
  - The completion of activities $\rightarrow$ the completion of the project

- Activity-on-Vertex (AOV) Networks
  - A digraph G with the vertices represent tasks or activities and the edges represent precedence relations between tasks
  - Predecessor: Vertex $i$ is a predecessor of vertex $j$, iff there is a directed path from vertex $i$ to vertex $j$

# AOV: Topological order

- A **linear ordering** of the vertices of a graph:
  - For any two vertices $i$ and $j$, if $i$ is a predecessor of $j$ in the network, then $i$ precedes $j$ in the linear ordering

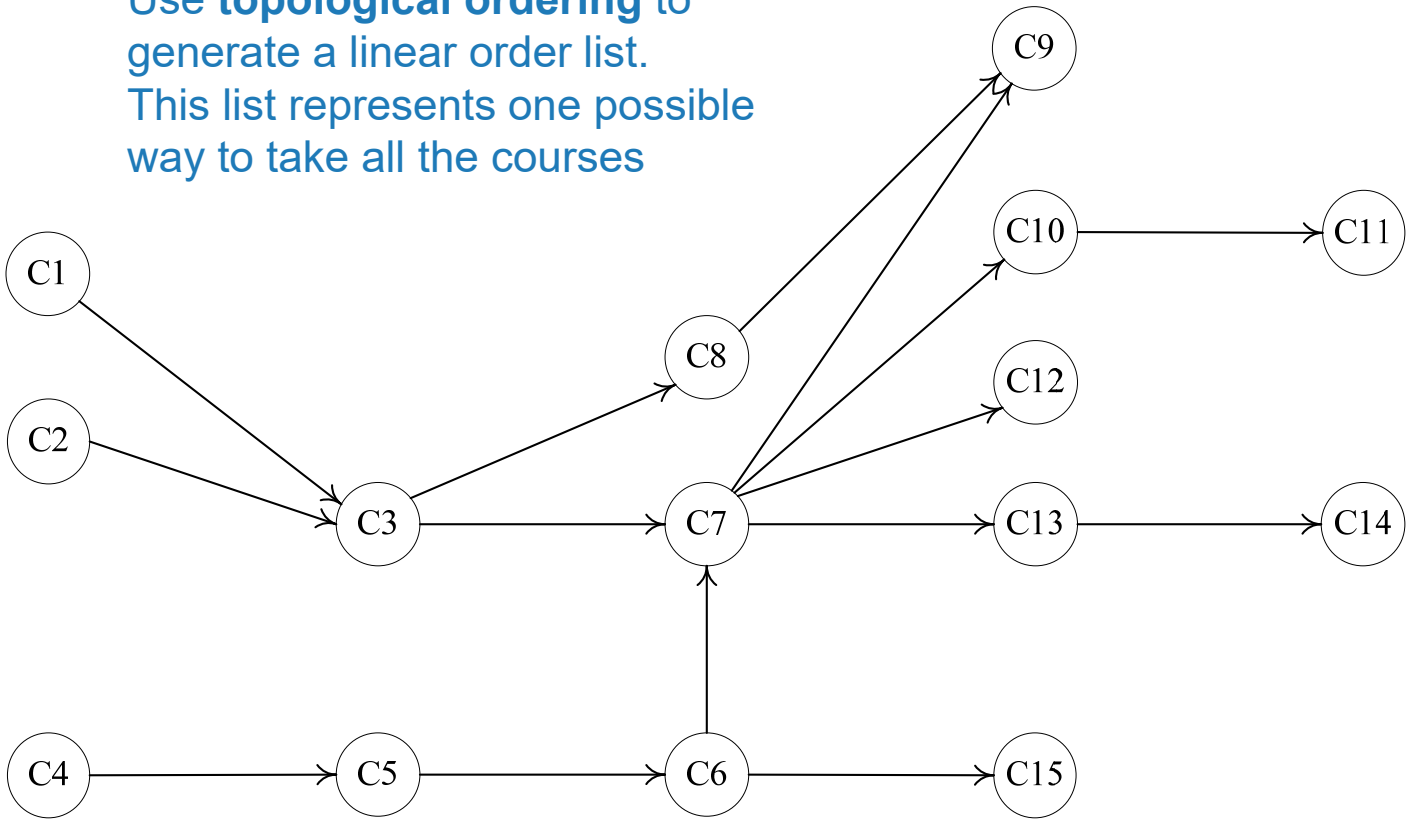


# Courses Need for CS Degree

| Course No. | Course | Prerequisites |
|---|---|---|
| C1 | Programming I | None |
| C2 | Discrete Mathematics | None |
| C3 | Data Structures | C1, C2 |
| C4 | Calculus I | None |
| C5 | Calculus II | C4 |
| C6 | Linear Algebra | C5 |
| C7 | Analysis of Algorithms | C3, C6 |
| C8 | Assembly Language | C3 |
| C9 | Operating Systems | C7, C8 |
| C10 | Programming Languages | C7 |
| C11 | Compiler Design | C10 |
| C12 | Artificial Intelligence | C7 |
| C13 | Computational Theory | C7 |
| C14 | Parallel Algorithms | C13 |
| C15 | Numerical Analysis | C5 |

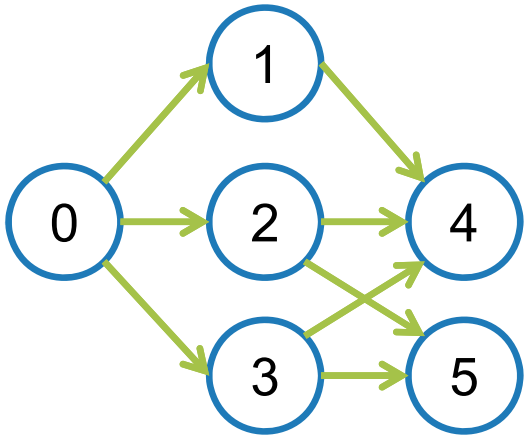
# AOV Network of Courses

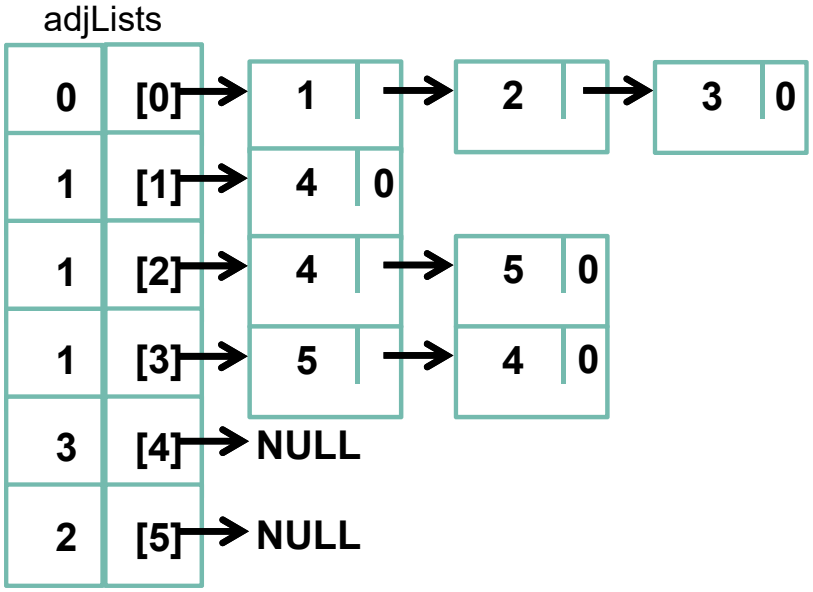
Use **topological ordering** to generate a linear order list.
This list represents one possible way to take all the courses



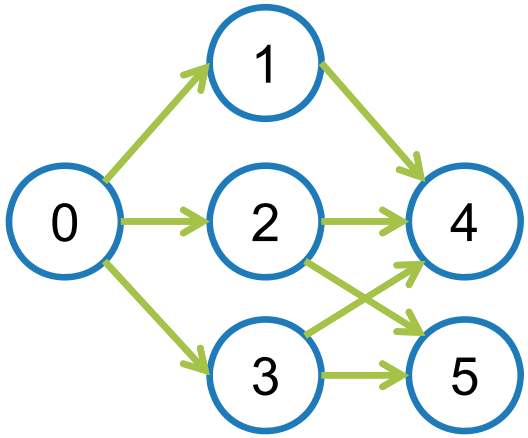
# Topological Ordering

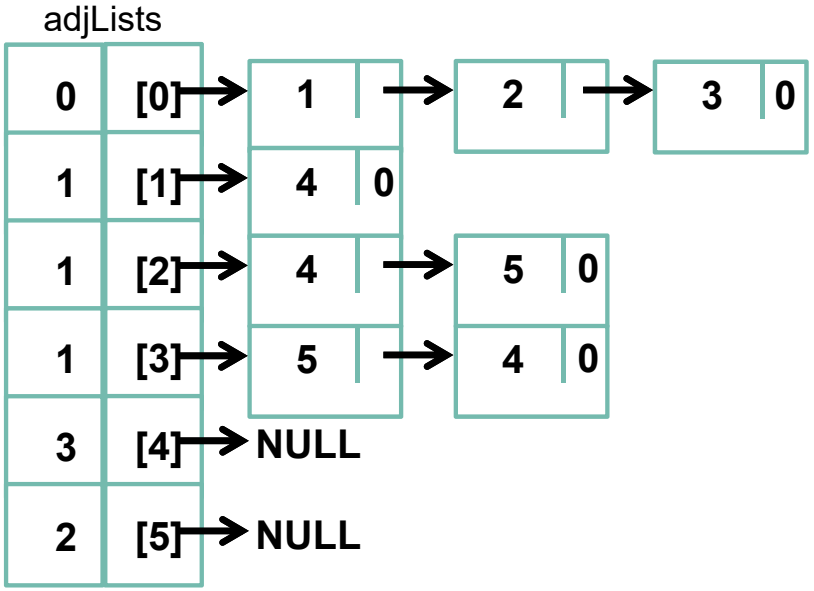
- Iteratively pick a vertex $v$ that has no predecessors
  - Use a field "count" to record the "in-degree" value of each vertex



# Topological Sorting




Ordered list:

# Self-Study Topics

- Single source shortest path
  - Bellman-Ford's algorithm (Digraph with negative edge costs)
- Activity-on-Edge (AOE) Networks
  - Critical path analysis