Data Structure and Algorithm Analysis

Chapter 11: Graph

Slides by: Yuhao Yi, and Tristan Wenzheng Xu

College of Computer Science

Sichuan University

Contents

Study Four common problems in graphs

1. Graphs have wide, wide applications

- Modeling relationships (families, organizations)
  - e.g., Model friendships in social networks
- Modeling connectivity in computer networks
- Representing maps
  - E.g., google map
- Finding paths from start to goal
- …
- Binary trees, B trees, B+ trees are special graphs

2. Notations in Graphs

- Unweighted graph vs. weighted graph
- Undirected graph vs. directed graph
- Degrees
- Path and cycle
- Path length
- Connectivity
- Connected components
- Acyclic directed graph

Graph properties

The importance of vertices in a graph

Relationship between vertices in a graph

Definition of an unweighted graph

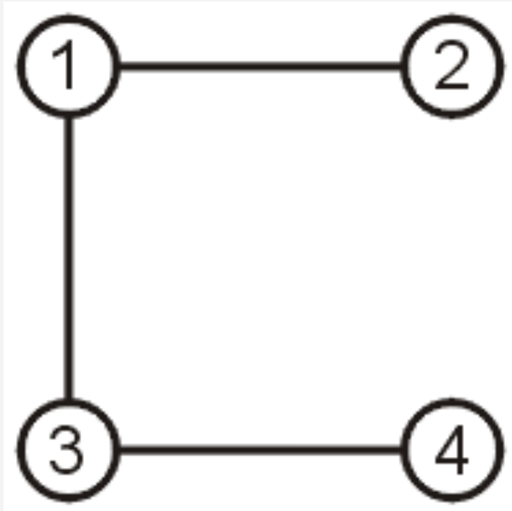
- A graph **G = (V, E)** consists of a set **V** of vertices, and a set of edges **E**, such that each edge in **E** is a connection between a pair of vertices in **V**
  - n=|V|, m=|E|
- Example: given the vertices

$$V = \{v_1, v_2, v_3, v_4\}$$

and the edges

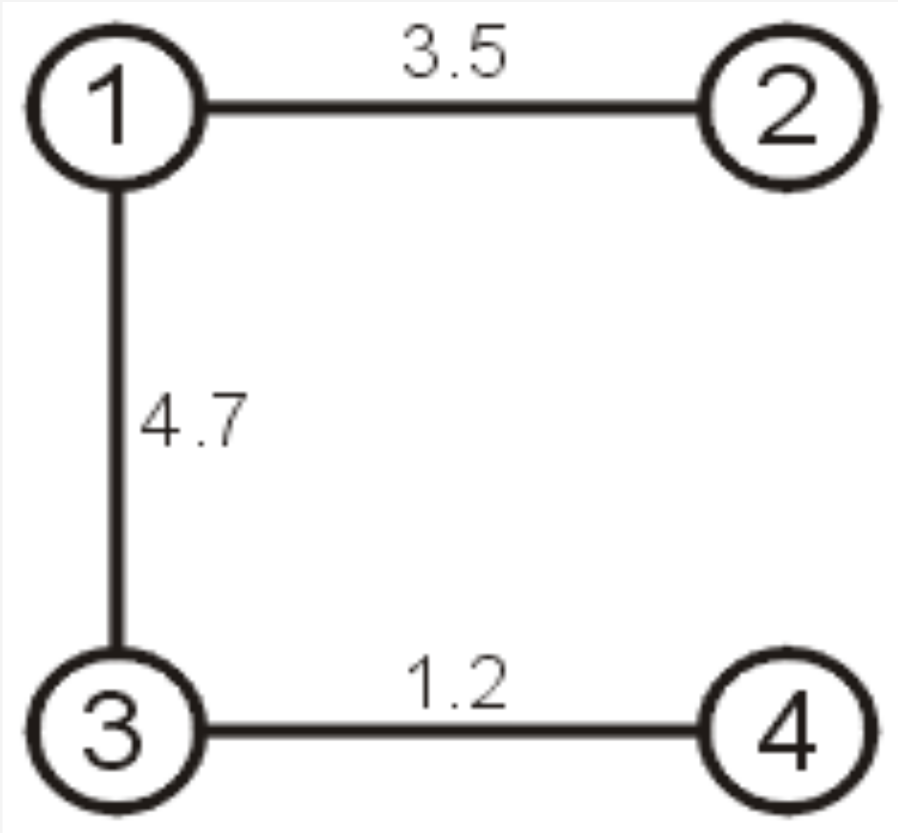
$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}\}$$

the graph has three edges connecting four vertices



Weighted Graphs

- Each edge may be associated with a weight
- This could represent distance, time, energy consumption, cost, etc

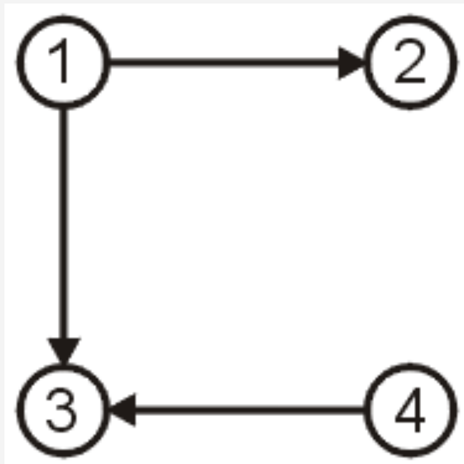


Directed Graphs

- Each edge in a graph may be associated with a direction
- An edge from v_i to v_j *does not imply* an edge from v_j to v_i
- All edges are ordered pairs (v_i, v_j) where this denotes a connection from v_i to v_j
- Such a graph is termed a *directed graph*
- For example,

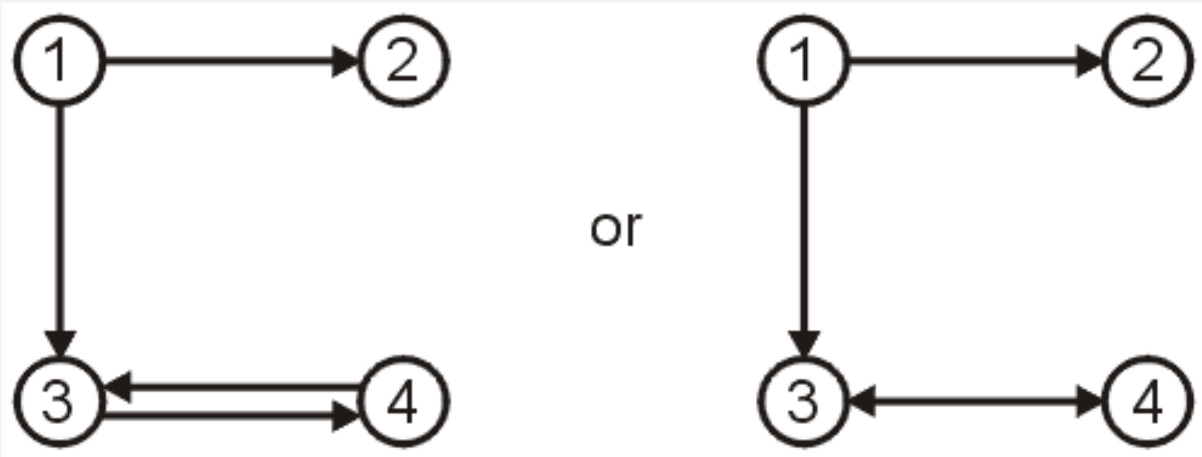
  $V = \{1, 2, 3, 4\}$

  $E = \{(1, 2), (1, 3), (4, 3)\}$



Directed Graphs

- If there is an edge from v_i *to* v_j and an edge from v_j *to* v_i , *plotted as*

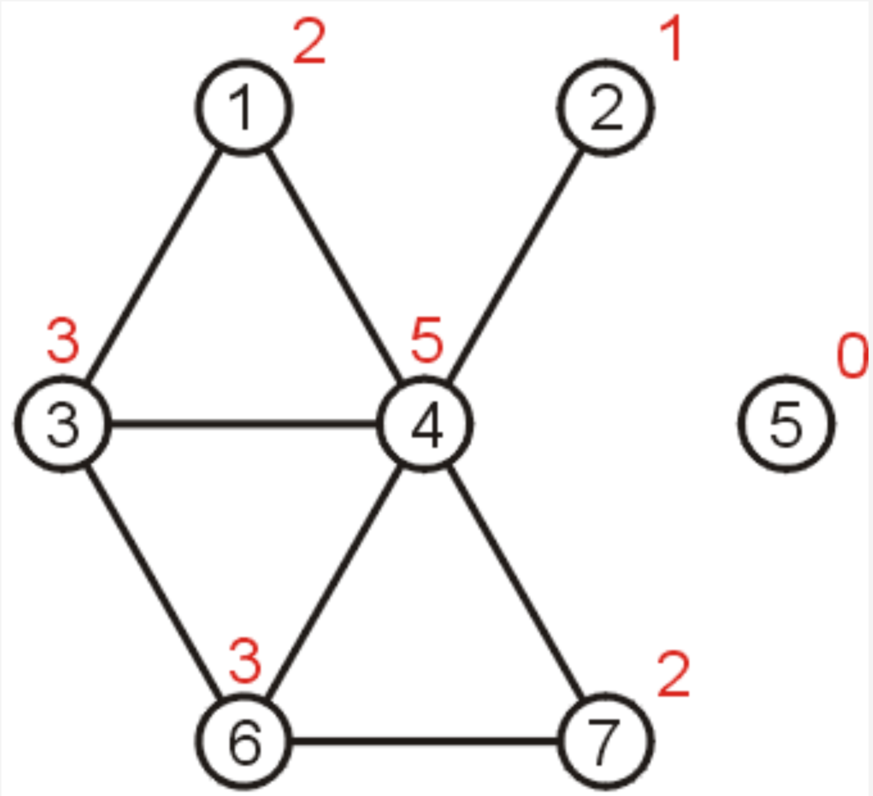


Directed Graphs vs. undirected graphs

- Graphs without directions are termed *undirected graphs*
- An undirected graph can be considered as a directed graph with each edge on both directions

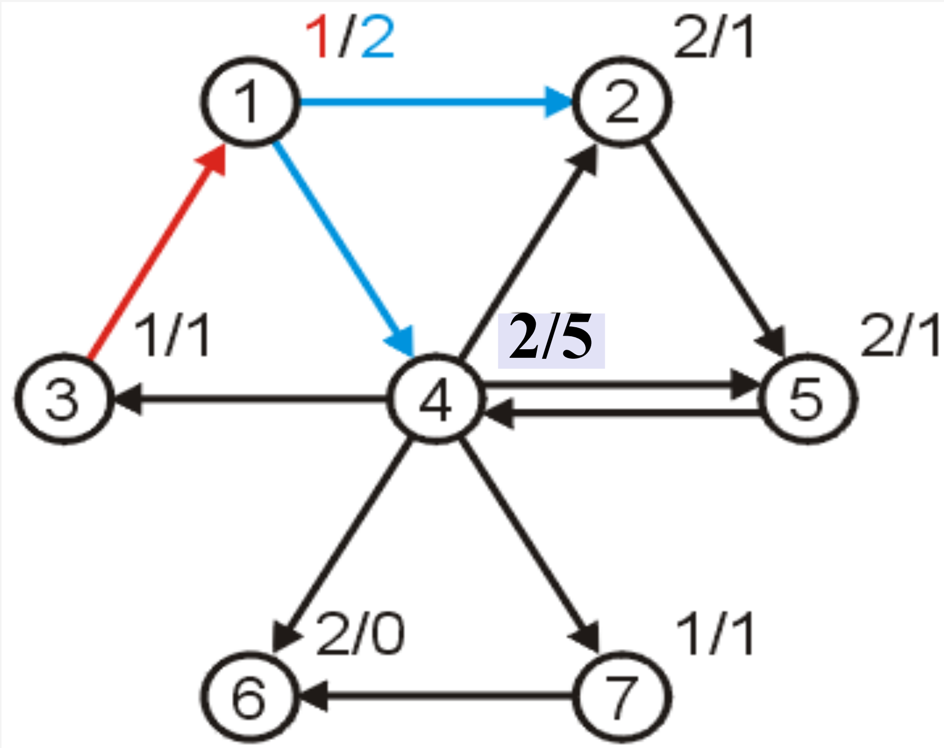
Degrees in an undirected graph

- We usually care how many neighbors of each vertex,
  - Especially the vertices with many neighbors
- The degree of a vertex is the number of neighbors



In and Out Degrees in a directed graph

- The in (incoming) degree of a vertex is the number of its incoming neighbors
- The out (out-going) degree of a vertex is the number of its out-going neighbors
- in/out



Paths

- A path *from* v_0 to v_k is an ordered sequence of vertices

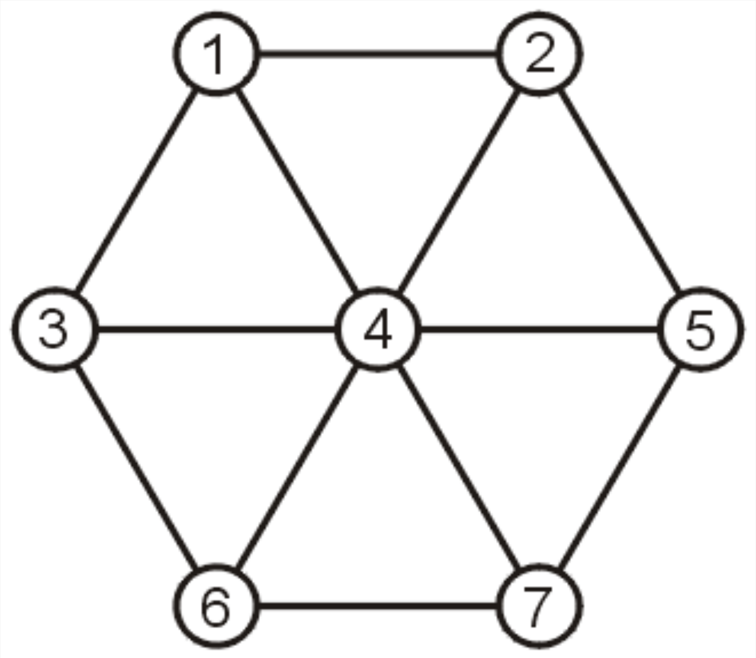
  $(v_0, v_1, v_2, \ldots, v_k)$

  where $\{v_{i-1}, v_i\}$ is an edge for $\mathrm{i} = 1, \ldots, k$
- Examples of paths from 1 to 5:

  (1, 2, 5)

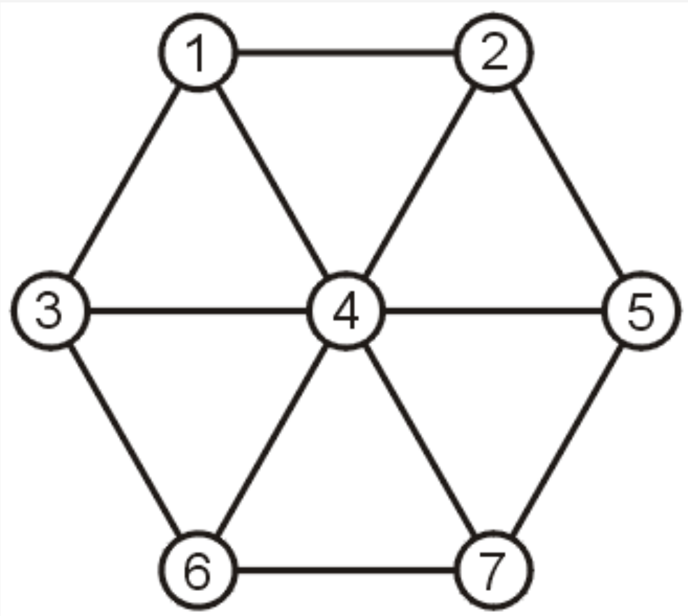
  (1, 4, 7, 5)

  (1, 2, 4, 1, 2, 5)



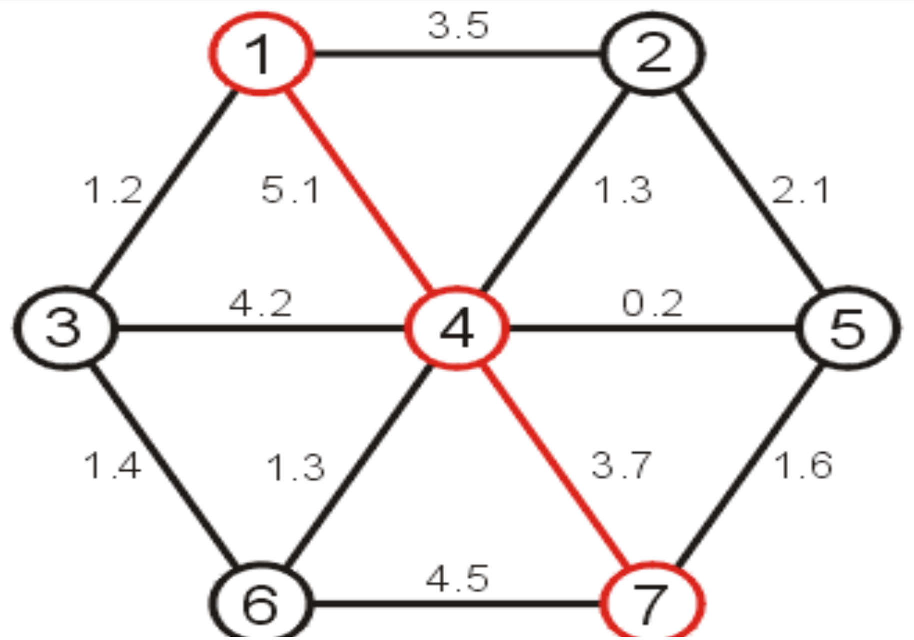
Simple Paths

- A *simple path* has no repetitions other than perhaps the first and last vertices
  - (1, 2, 5) simple path
  - (1, 2, 4, 1, 2, 5) not simple path
- A *simple path* where the first and last vertices are equal is said to be a *cycle*
  - e.g., (1, 2, 4, 1)



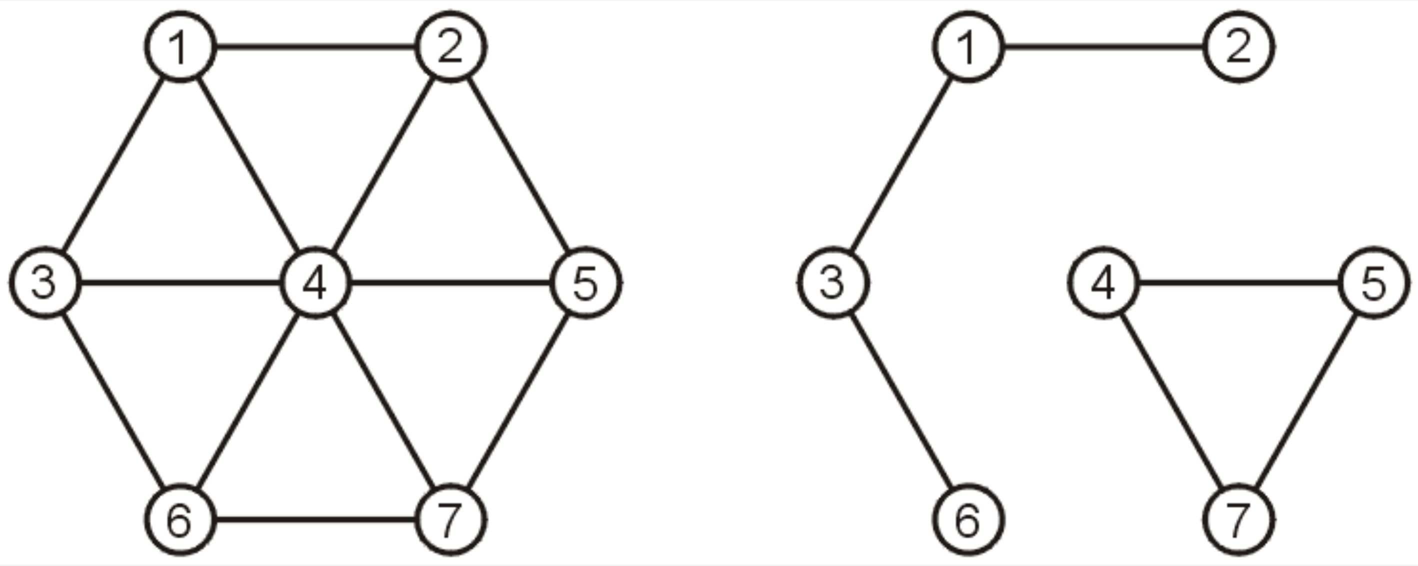
Path length

- The length of an unweighted path is the number of edges in the path
- The *length* of a weighted path is the weighted sum of the edges in the path
  - The length of the path 1→4 →7 in the following graph is 5.1 + 3.7 = 8.8



Connectivity

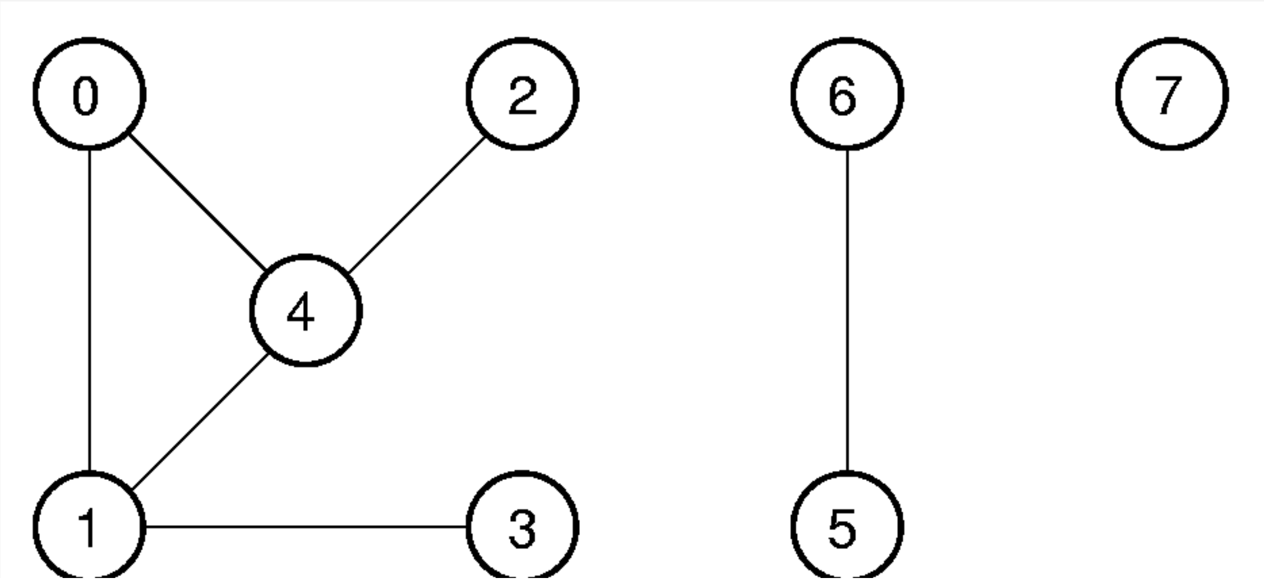
- Two vertices v_i, v_j are said to be *connected* if there is a path between v_i to v_j
- A graph is connected if there is a path between any two vertices



Connected graph **Disconnected graph**

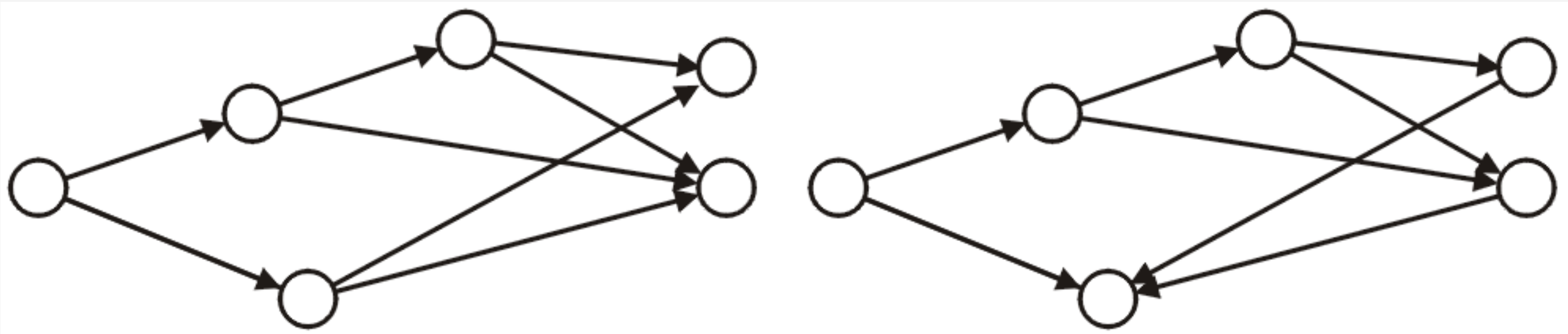
Connected Components

- A graph may be disconnected
- But a subgraph may be connected
- A maximum connected subgraph of a graph is called a connected component (CC), e.g.,
  - CC1 with vertices 0, 1, 2, 3, 4
  - CC2 with vertices 5, and 6
  - CC3 with only vertex 7

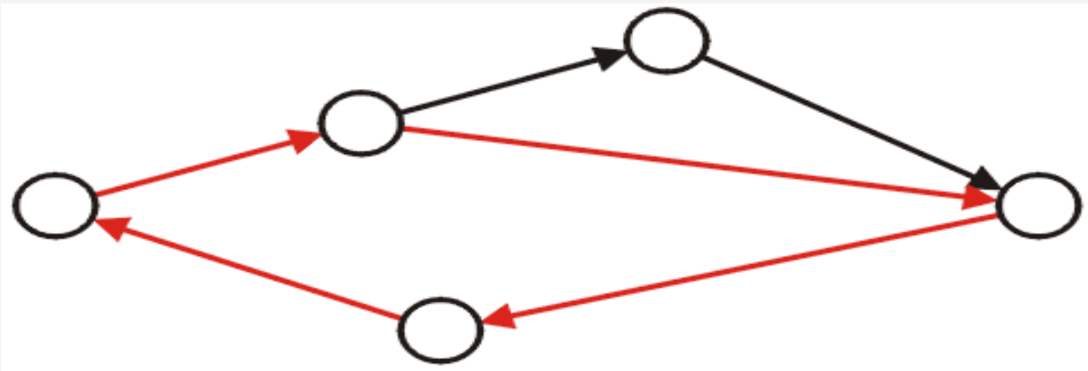


Directed Acyclic Graphs

- A *directed acyclic graph (DAG)* is a directed graph which has no cycles
- Two example DAGs



- Not a DAG



Applications of Directed Acyclic Graphs

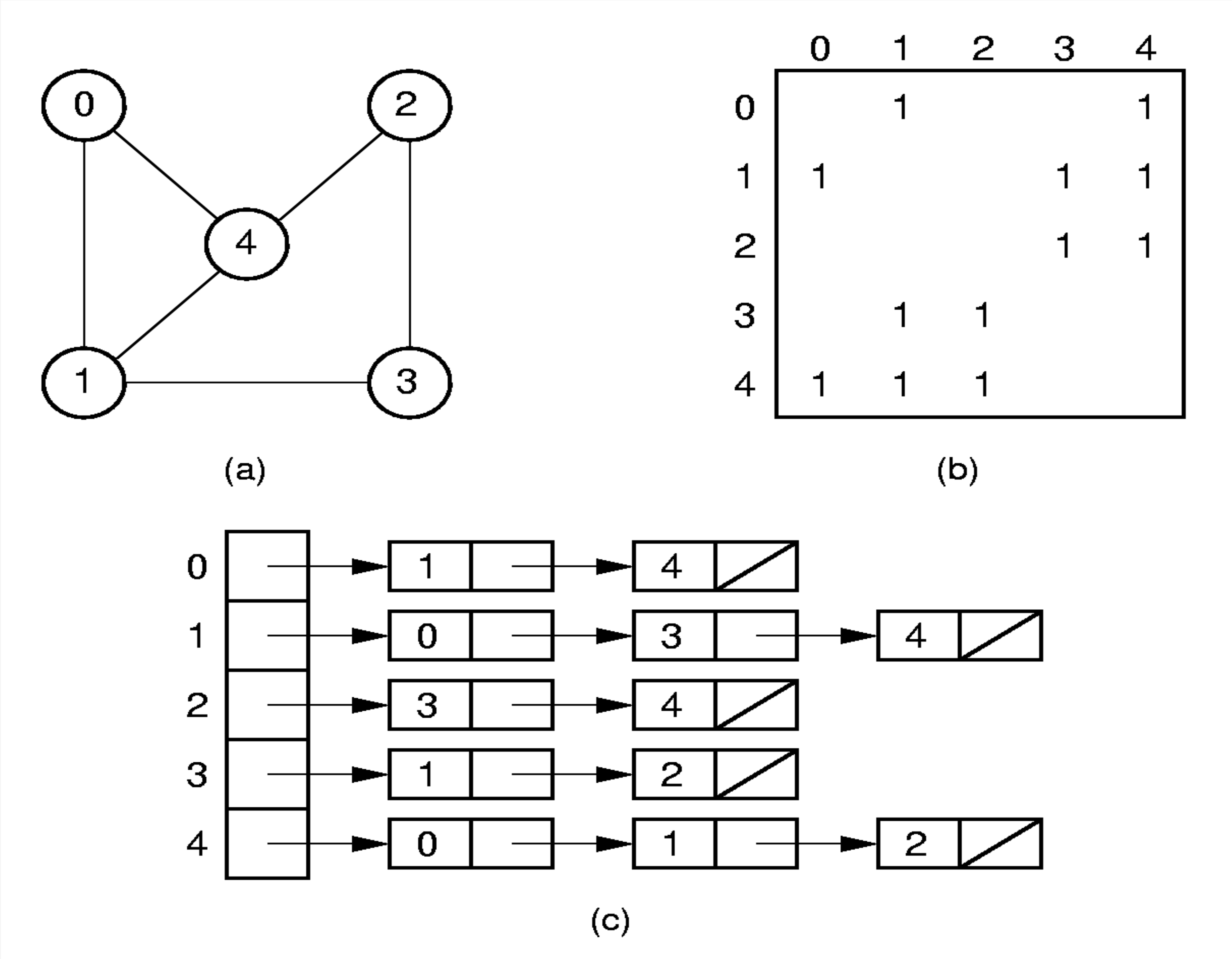
- Applications of DAGs include:
  - Family trees
  - Course pre-requisites
  - Folders and sub folders in an Operation system
  - …

3. Representations of a graph in computers

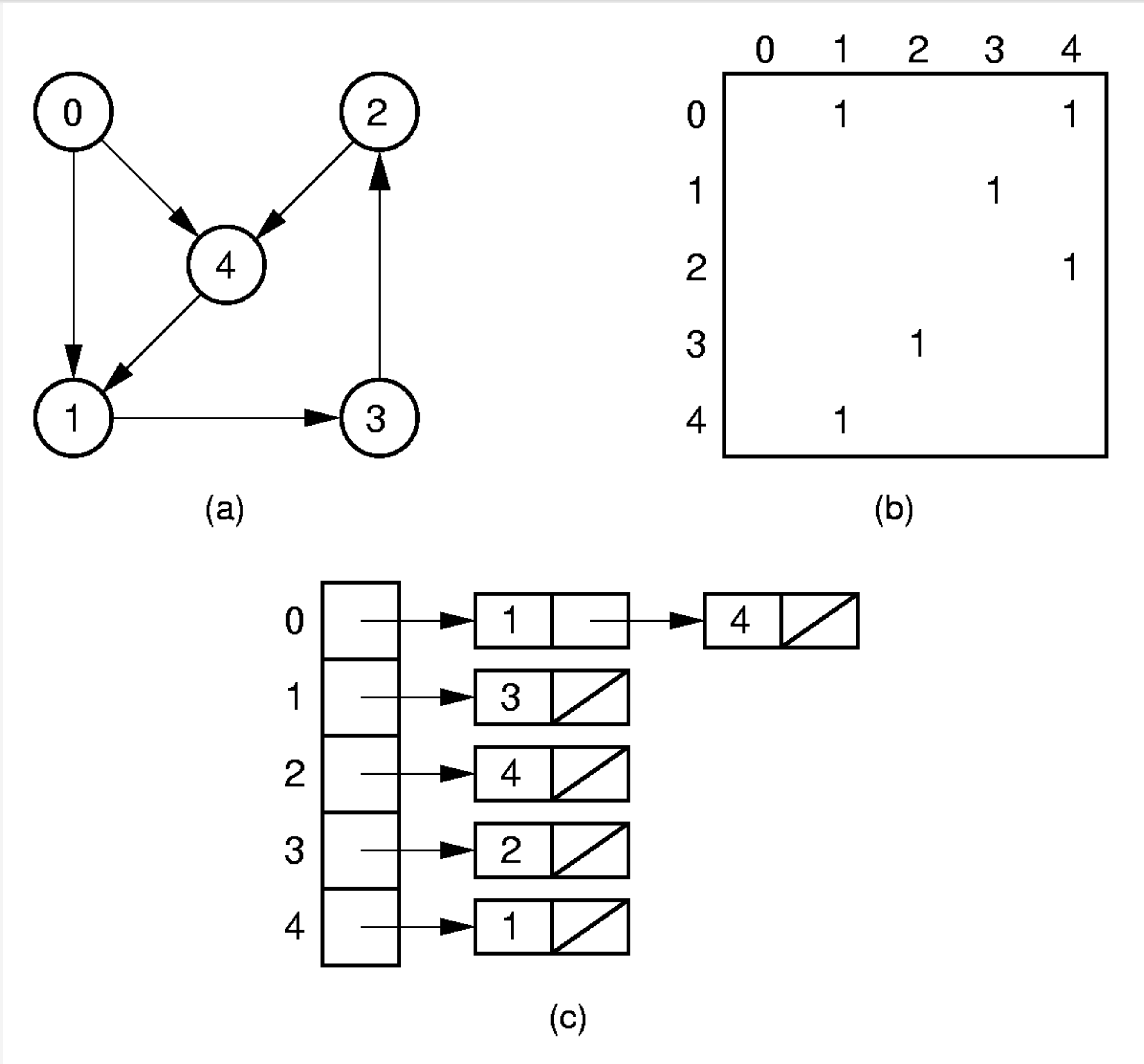
- Adjacency Matrix
- Adjacency List

Representations for an Undirected graph

a) Graph structure b) Adjacency matrix for the graph c) Adjacency list for the graph



Representations for a directed graph



Representation Space costs

- Adjacency Matrix:
  - $\Theta(n^2)$
  - $n=|V|$ and $m=|E|$
  - Suitable for dense graphs
- Adjacency List
  - $\Theta\ (n+m)$
  - $m \leq n(n-1)$
  - Suitable for sparse graphs
  - Most real graphs are sparse

4. Graph Traversals

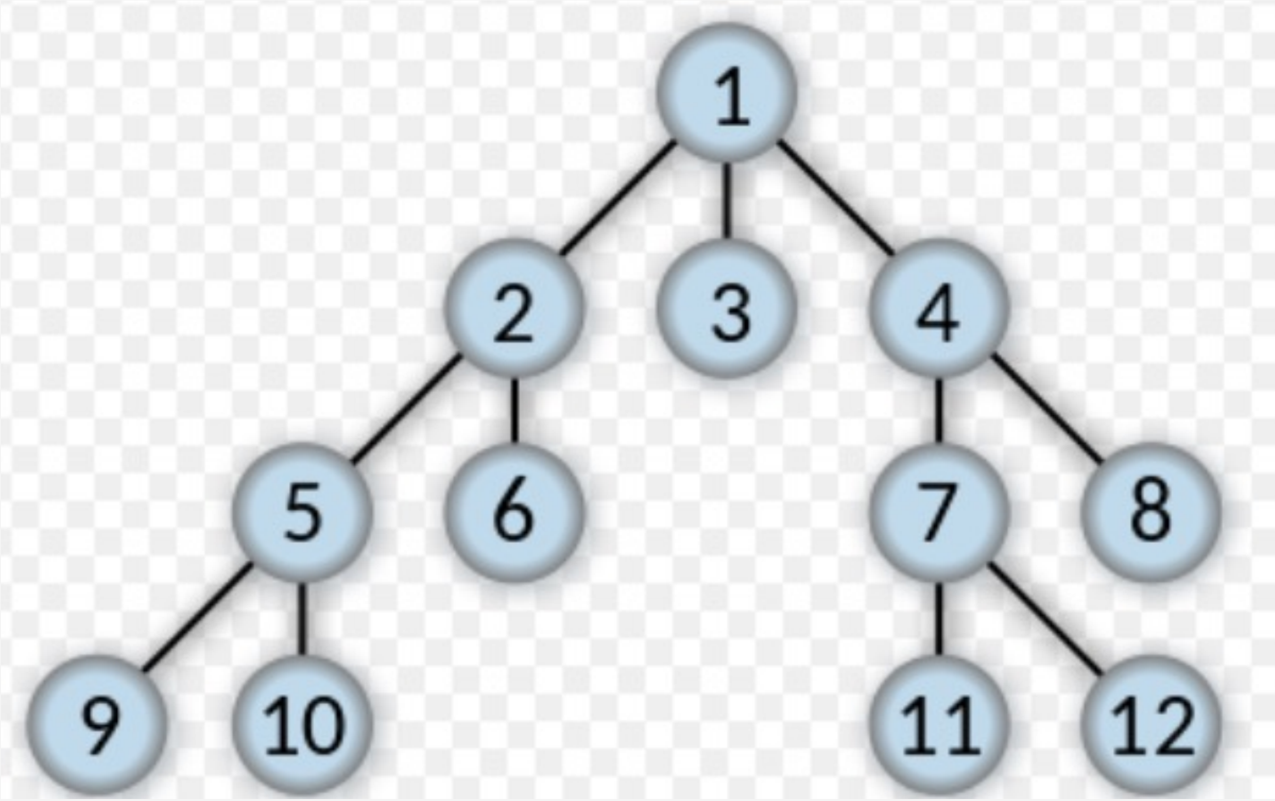
- Some applications require visiting every vertex in the graph exactly once, in some special order based on graph topology
- Two orders of graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)

Breadth-first search (BFS)

- It starts at a root vertex *s*, the root at level 0
- Visit first the root vertex in level 0, then vertices in level 1, vertices in level 2,…
- Level means the shortest distance to the root
- Need an auxiliary queue in the search

BFS example in a tree

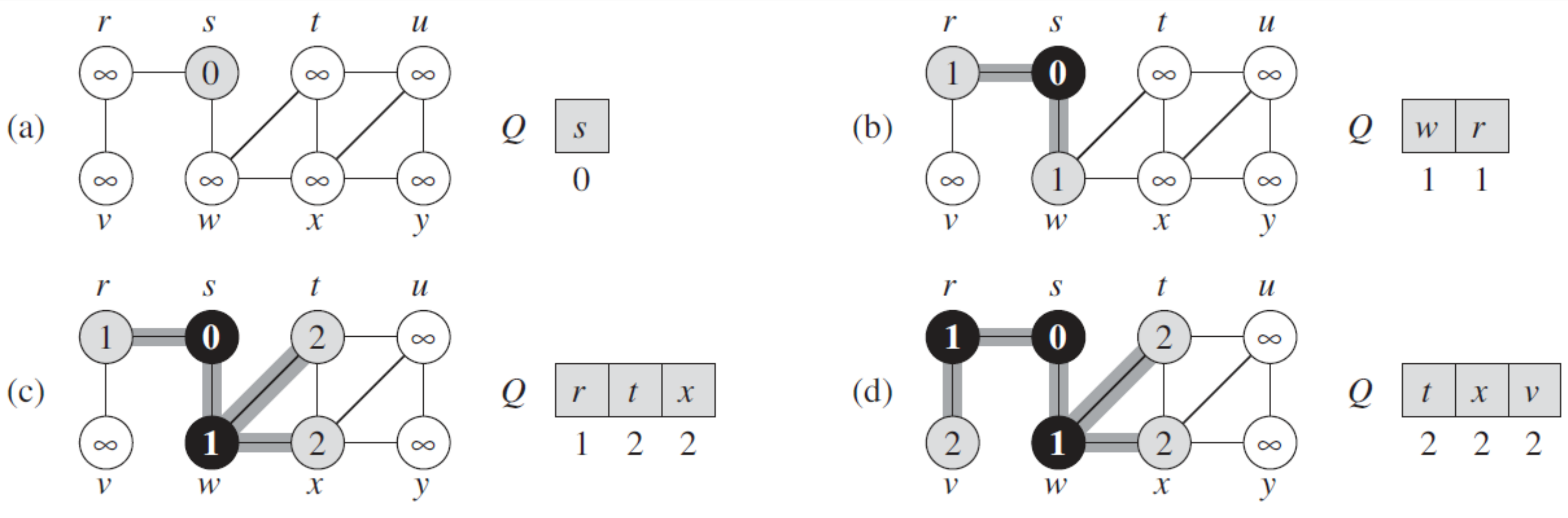
- A tree is a special graph
- BFS starts from vertex 1



Order in which the nodes are visited

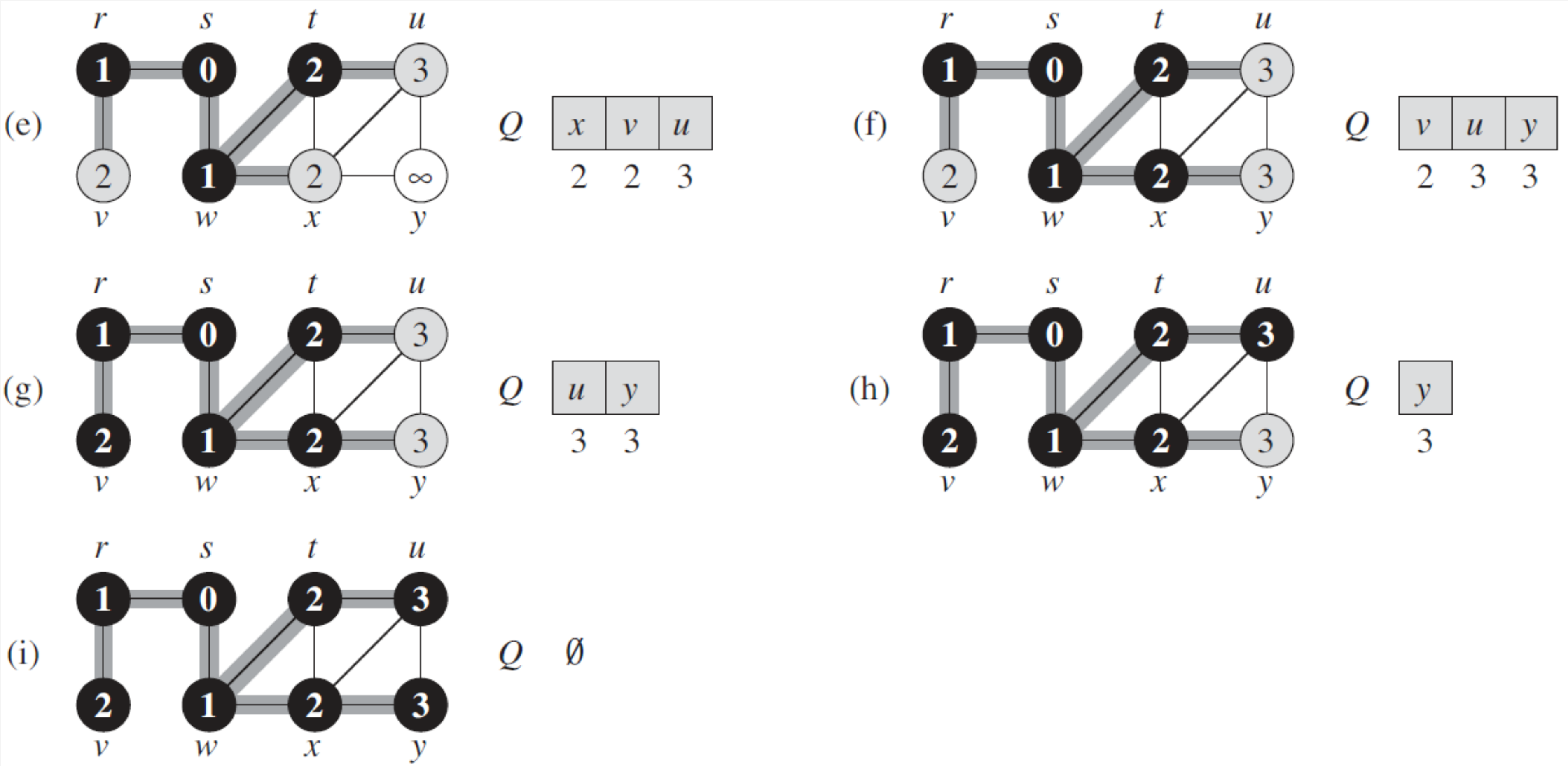
BFS example in a graph, starts from vertex *s*

- Queue *Q* stores the vertices visited, but has not explored their neighbors
- Once the neighbors of a vertex is explored, it is removed out from queue *Q*



BFS example-cont.

- BSF calculates the *shortest distance* of each vertex to root *s*, assume each edge weight is 1
- Time complexity of BFS: Θ (n+m)



BFS algorithm

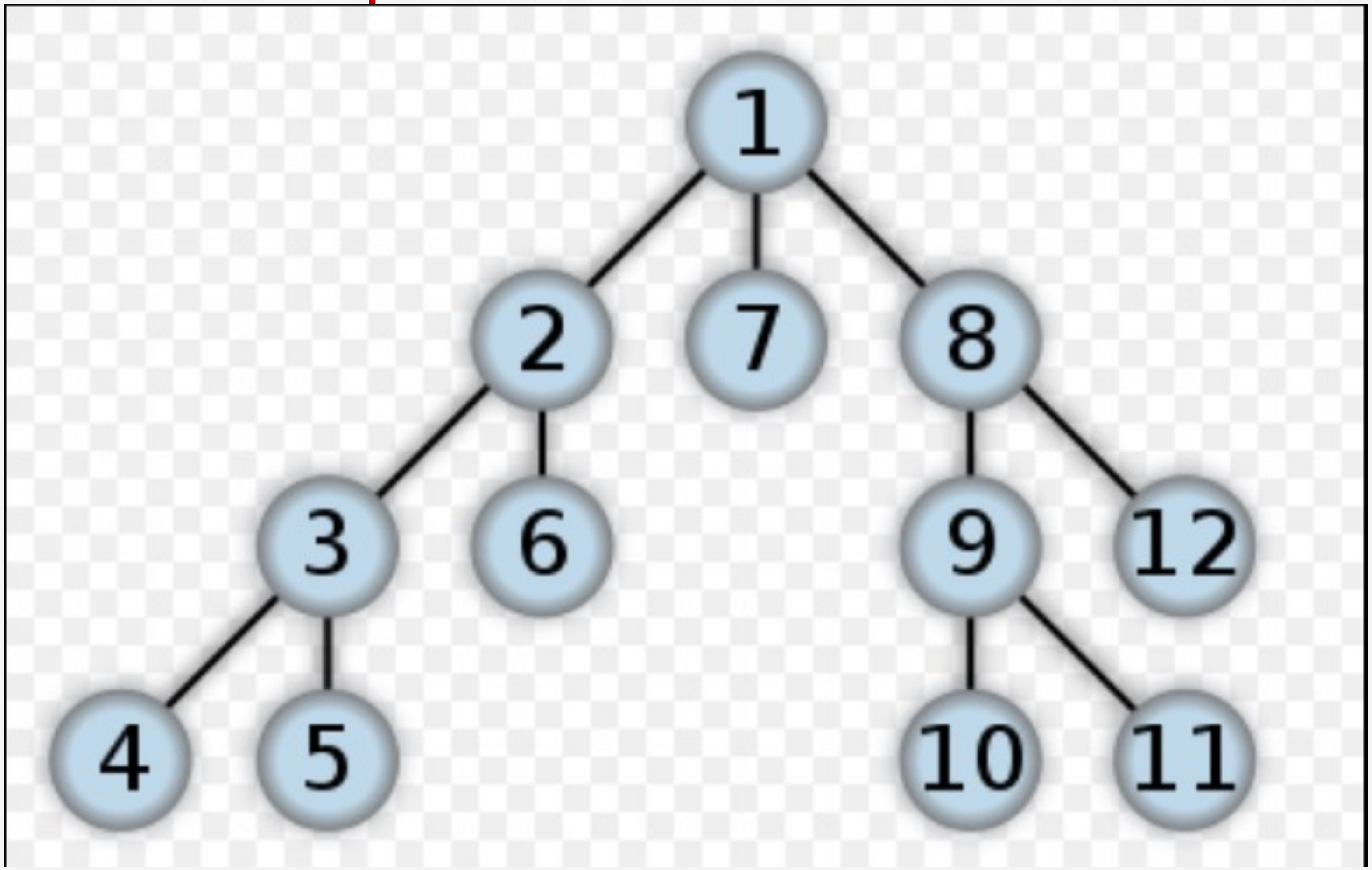
```
void BFS(Graph* G, int s) {
  Queue<int> Q;
  bool *visited = new bool[G->n()];
  for(int i=0; i<G->n(); ++i) visited[i] = false;
  Q->enqueue(s);            // Initialize Q
  visited[s]= true;
  int v, w;
  Node *cur;
  while (Q->length() > 0) { // Process Q
    Q->dequeue(v);
    PreVisit(G, v);    // Take action
    for(cur = G->adjList[v]; cur != NULL; cur=cur-
  >next ){
      w = cur->nodeID;
      if( false == visited[w] ){
        visited[w] = true;
        Q->enqueue(w);
      }
    }
  }
  delete []visited;
}
```

Depth-first search (DFS)

- It starts at a root vertex
- Explore one branch of a vertex as far as possible, before exploring another branch of the vertex
- If no branches can be explored, backtrack

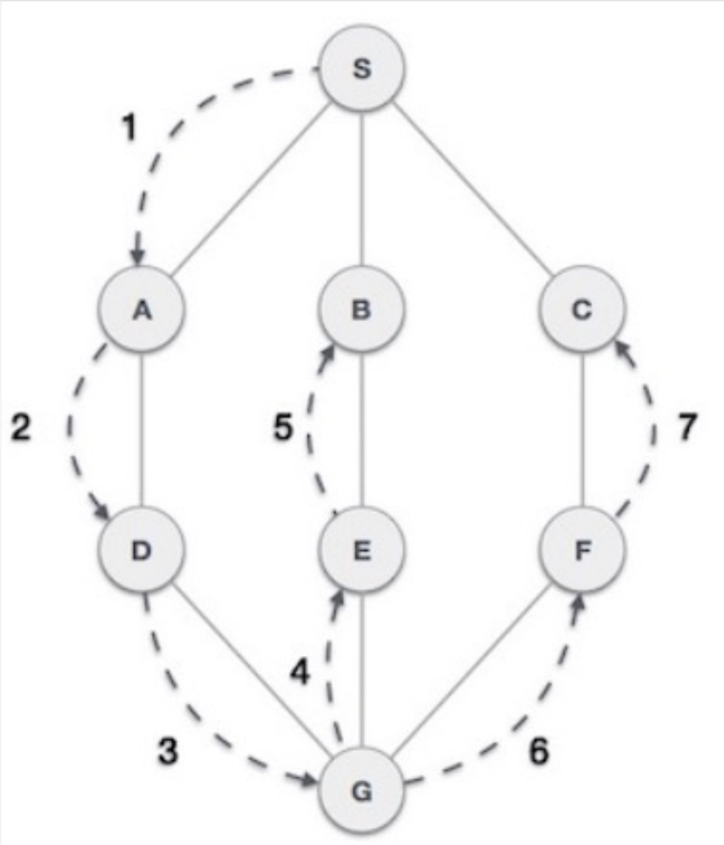
DFS example in a tree

- DFS starts from vertex 1
- Similar to a pre-order traversal in a tree



Order in which the nodes are visited

DFS example in a graph, start from vertex *s*



- Vertices are visited in order: s->A->D->G->E->B->F->C
- There may be multiple orders
- Another order is: s->B->E->G->F->C->D->A

DFS Algorithm

```
void DFS(Graph* G, int v) {
   PreVisit(G, v);   // Take action
   visited[ v ] = true;
   Node *cur;
   for ( cur=G->adjList[v]; cur !=NULL;
  cur=cur->next){
       w = cur->nodeID;
       if( false == visited[ w ] )
           DFS(G, w );
   }
}
```

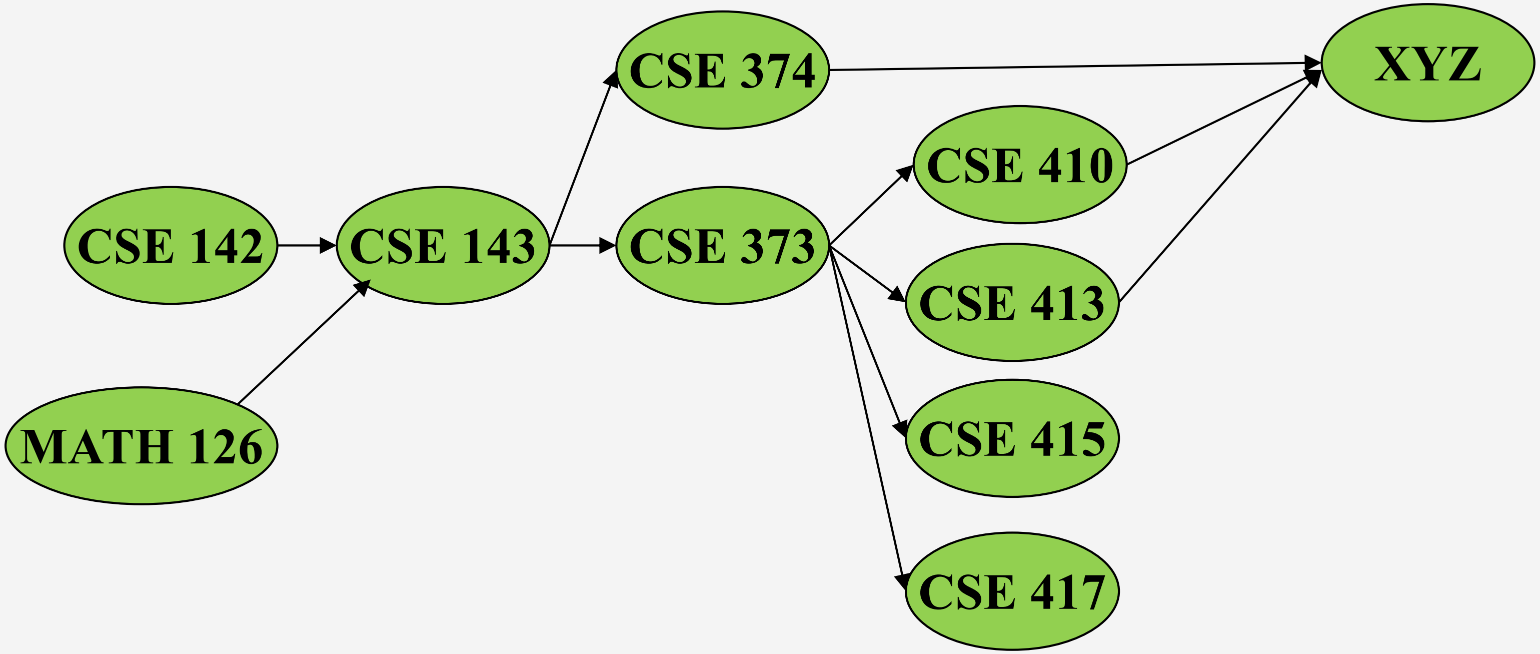
Time complexity: Θ (n+m)

5. Topological Sort, applications:

1. Consider all courses you will learn, some course must be learned before another
   - e.g., You must learn C before this course
   - List all courses in order, such that no prerequisite courses is after each course in the order
   - E.g., you cannot learn this course before C
2. Given a set of jobs to be done by a computer, and some jobs must be finished before other jobs
   - List all jobs in order, such that no prerequisite jobs is after each job in the order

Topological Sort

- Problem: Given a DAG `G=(V,E)`, output all vertices in an order such that no vertex v_j appears before another vertex v_i if there is an edge from v_i to v_j in G

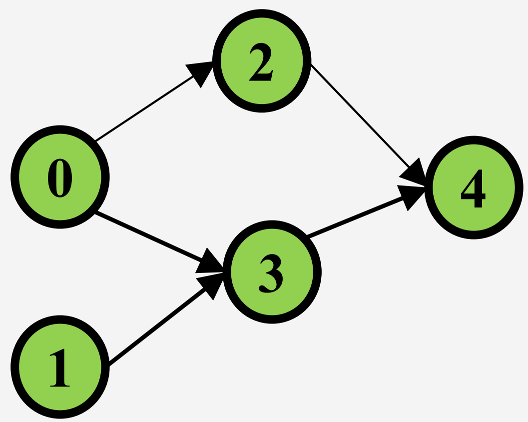


One example output:

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415

Questions and comments

- Why do we perform topological sorts only on DAGs?
  - Because a cycle means there is no correct answer
- Is there always a unique order?
  - No, there can be multiple orders; depends on the graph

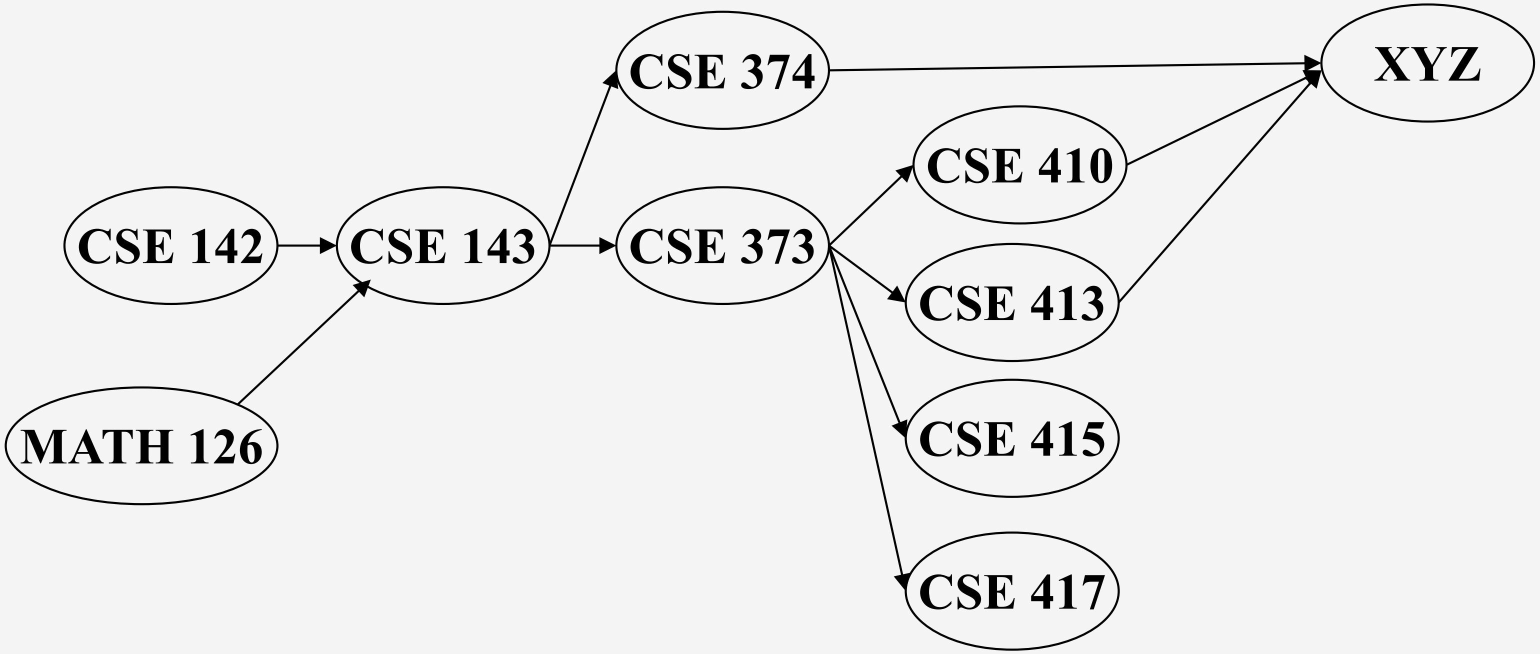


- Do some DAGs have exactly 1 order?
  - Yes, e.g., the DAG is a linked list

Algorithm for Topological Sort

- While there are vertices not yet output:
  - Choose a vertex **v** with in-degree of 0, i.e., no dependency
  - Output **v** and remove it from the graph
  - For each out-going neighbor **u** of **v**, decrease the in-degree of **u** **by 1**

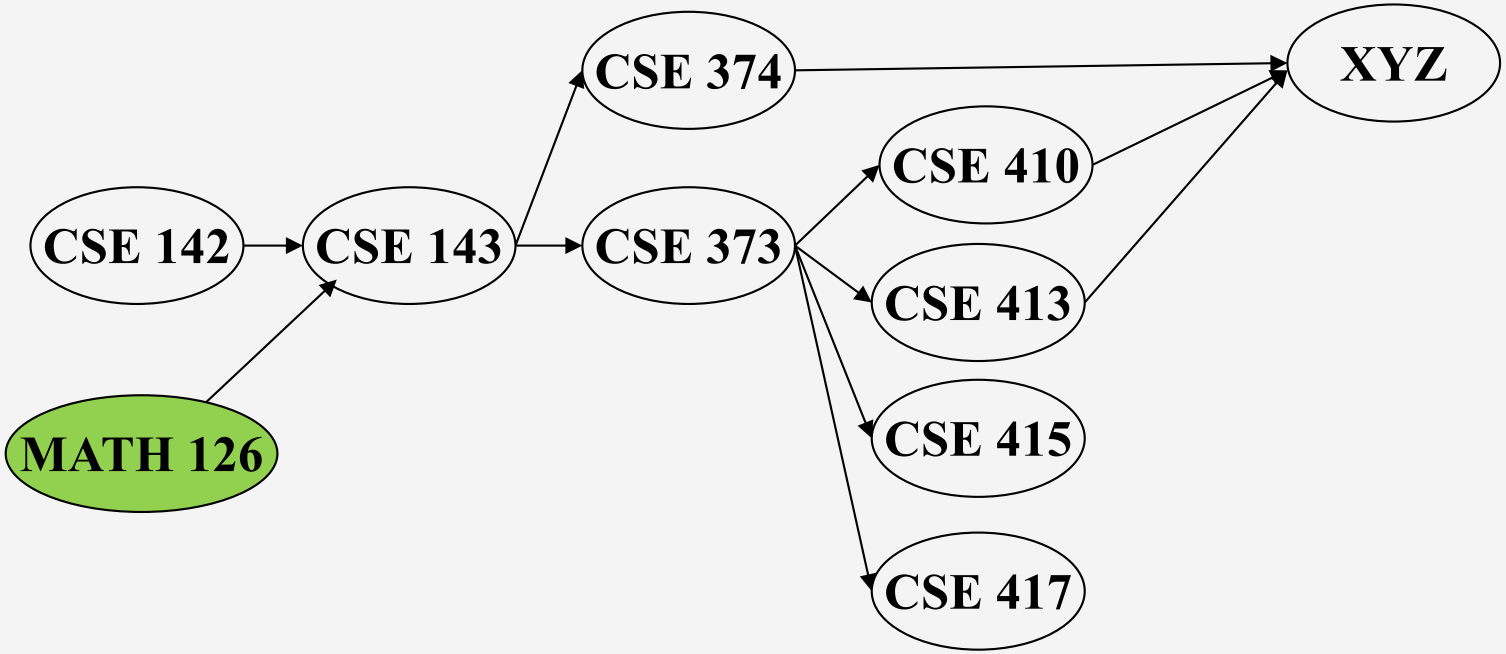
Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?										
In-degree:	0	0	2	1	1	1	1	1	1	3

Output:

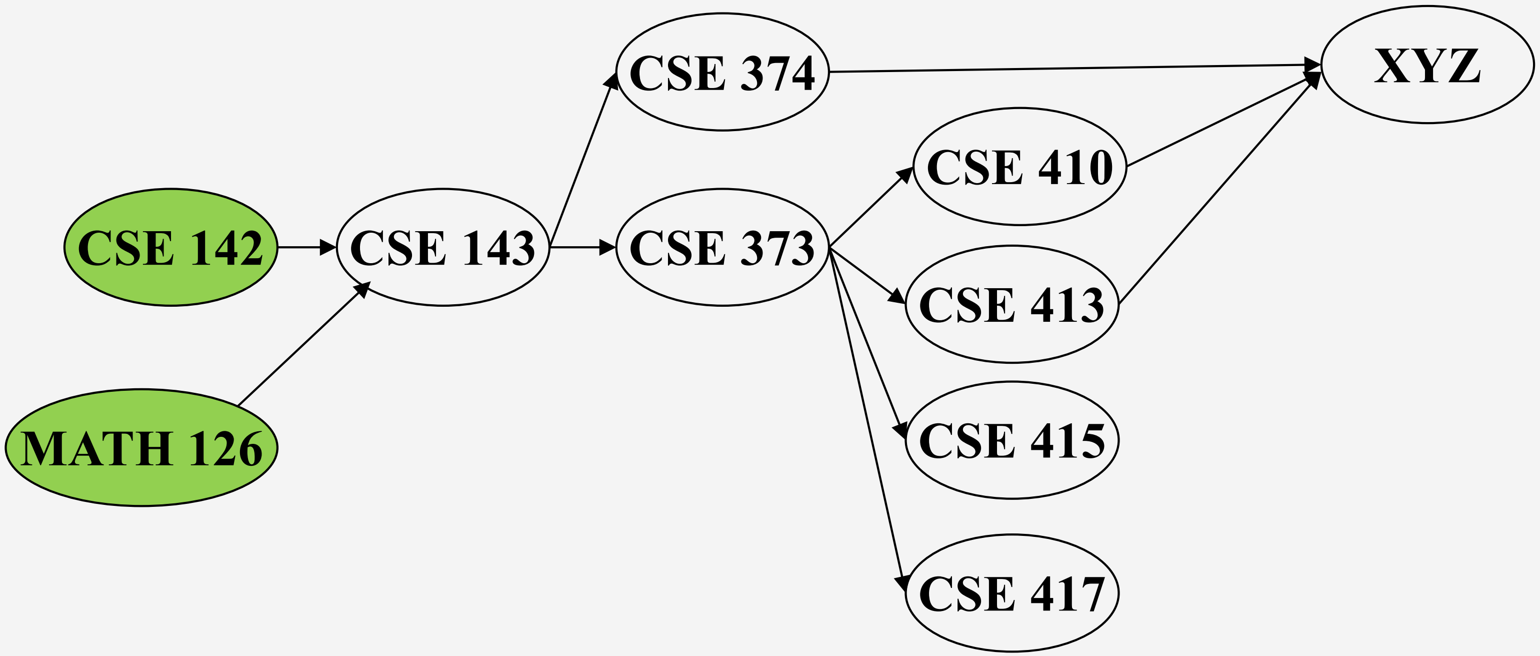
Example



Output:
126

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x									
In-degree:	0	0	2	1	1	1	1	1	1	3
			1							

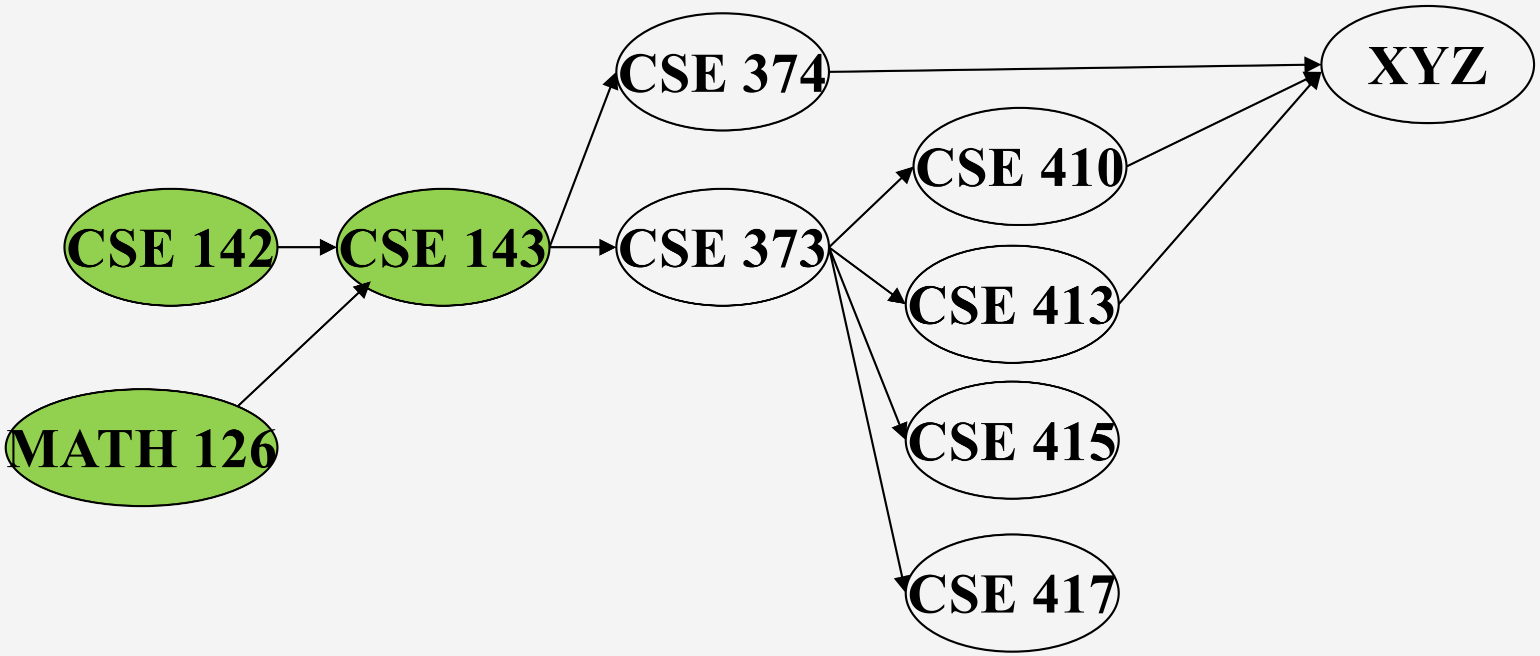
Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x								
In-degree:	0	0	2	1	1	1	1	1	1	3
			1							
			0							

Output:
126
142

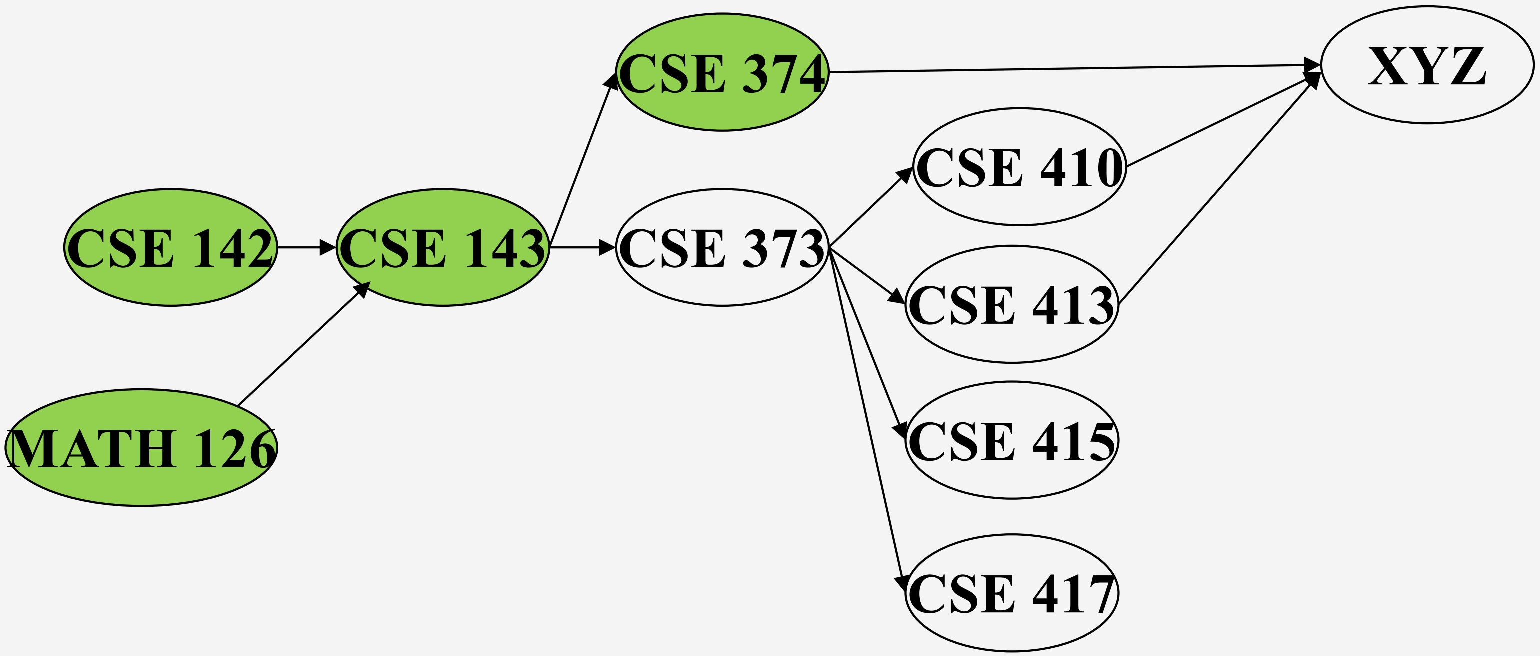
Example



Output:
126
142
143

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x							
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0					
			0							

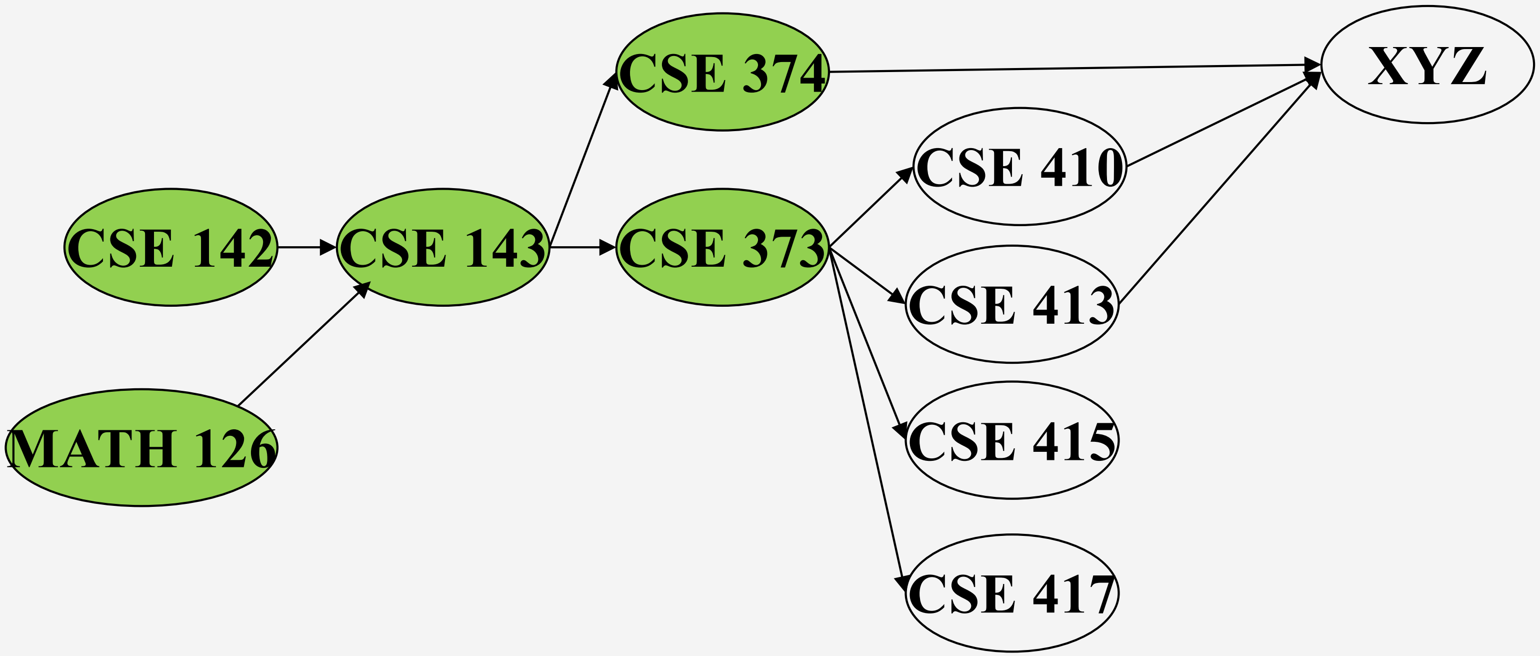
Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x						
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0					2
			0							

Output:
126
142
143
374

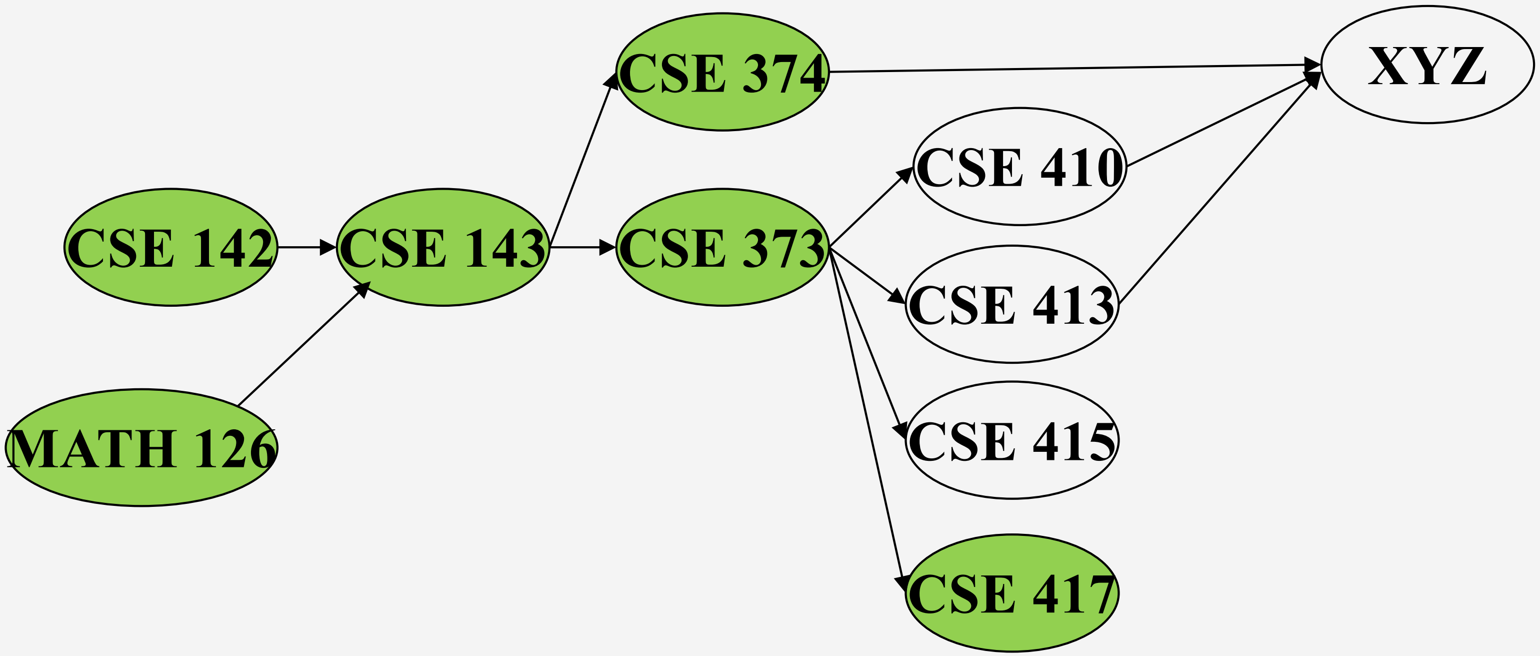
Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x					
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							

Output:
126
142
143
374
373

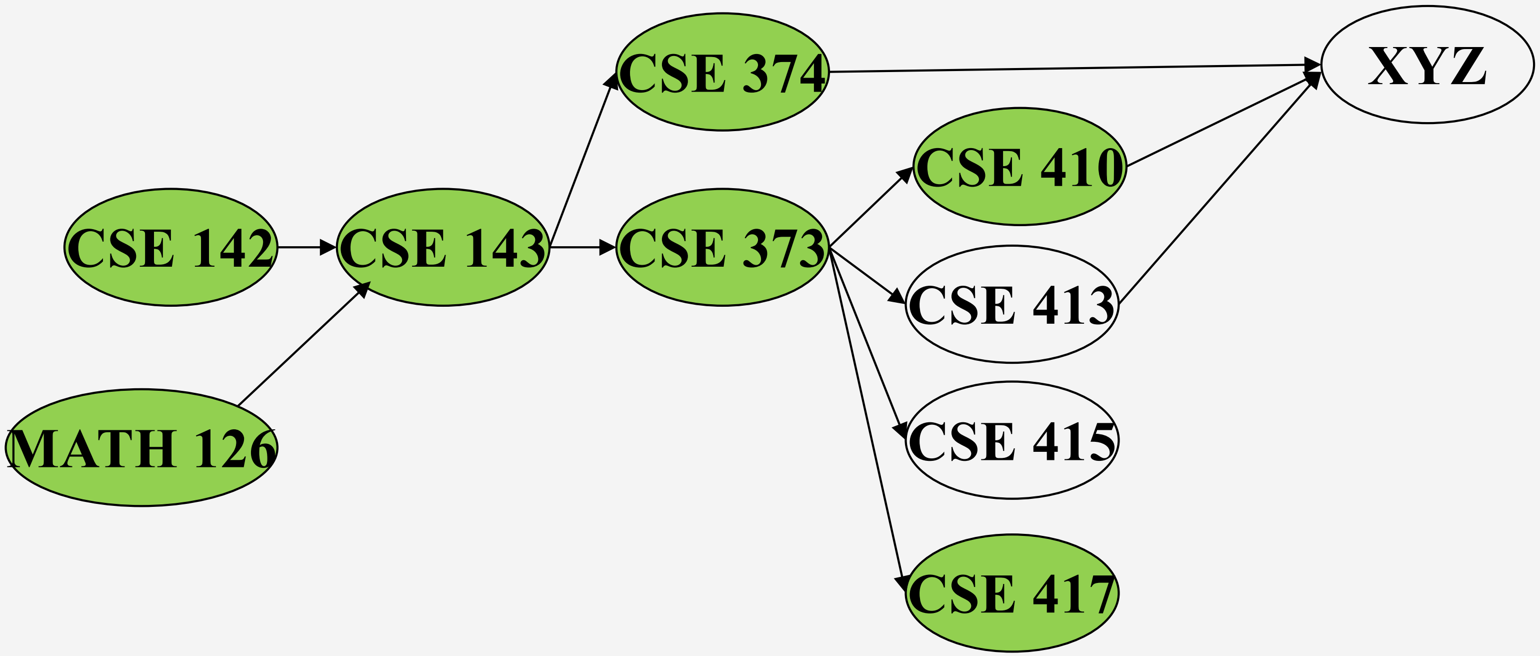
Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x				x	
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							

Output:
126
142
143
374
373
417

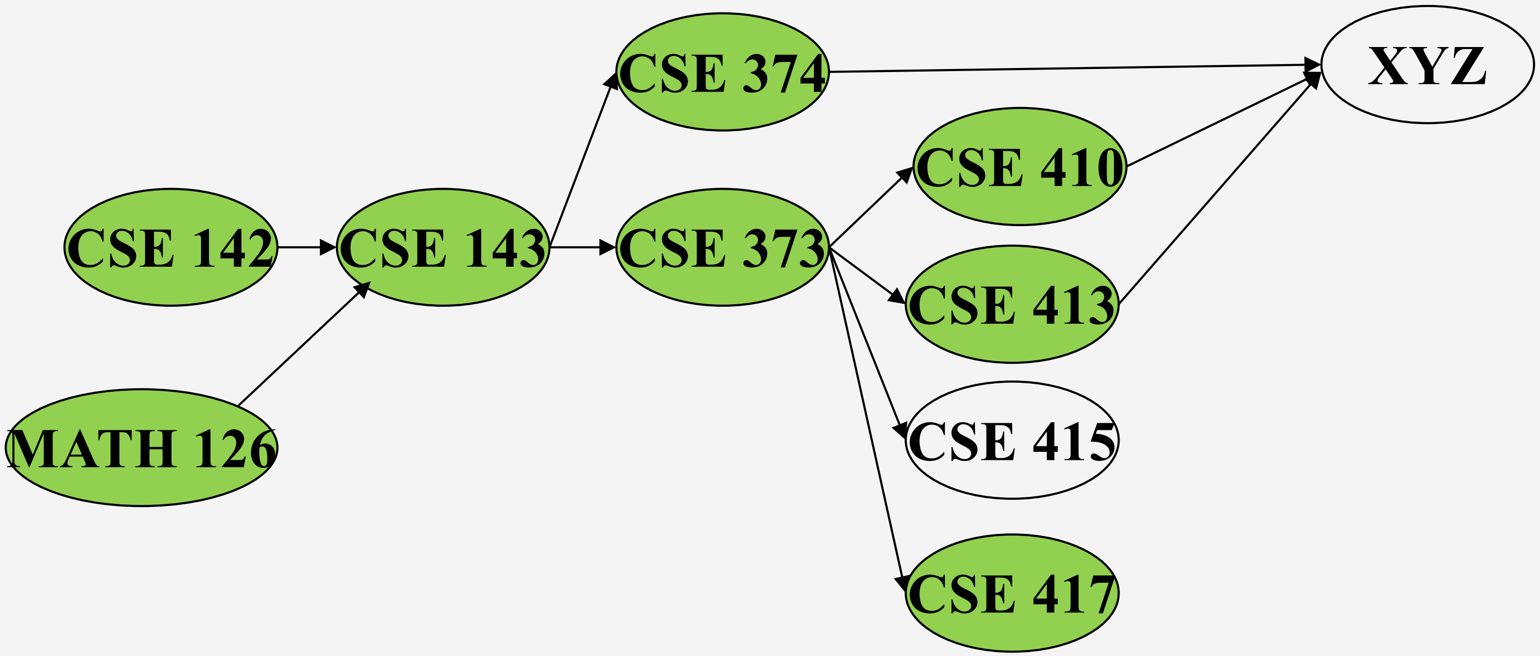
Example



Output:
126
142
143
374
373
417
410

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x			x	
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							1

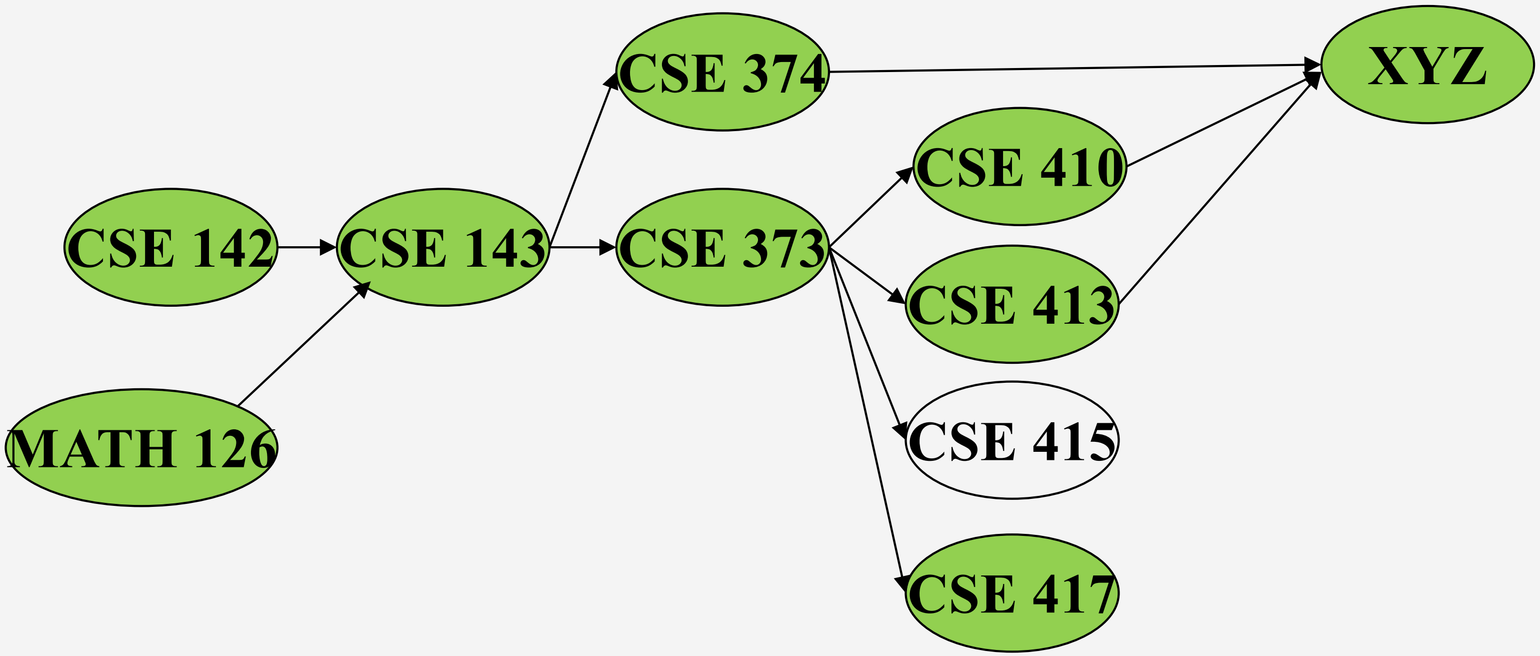
Example



Output:
126
142
143
374
373
417
410
413

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							1
										0

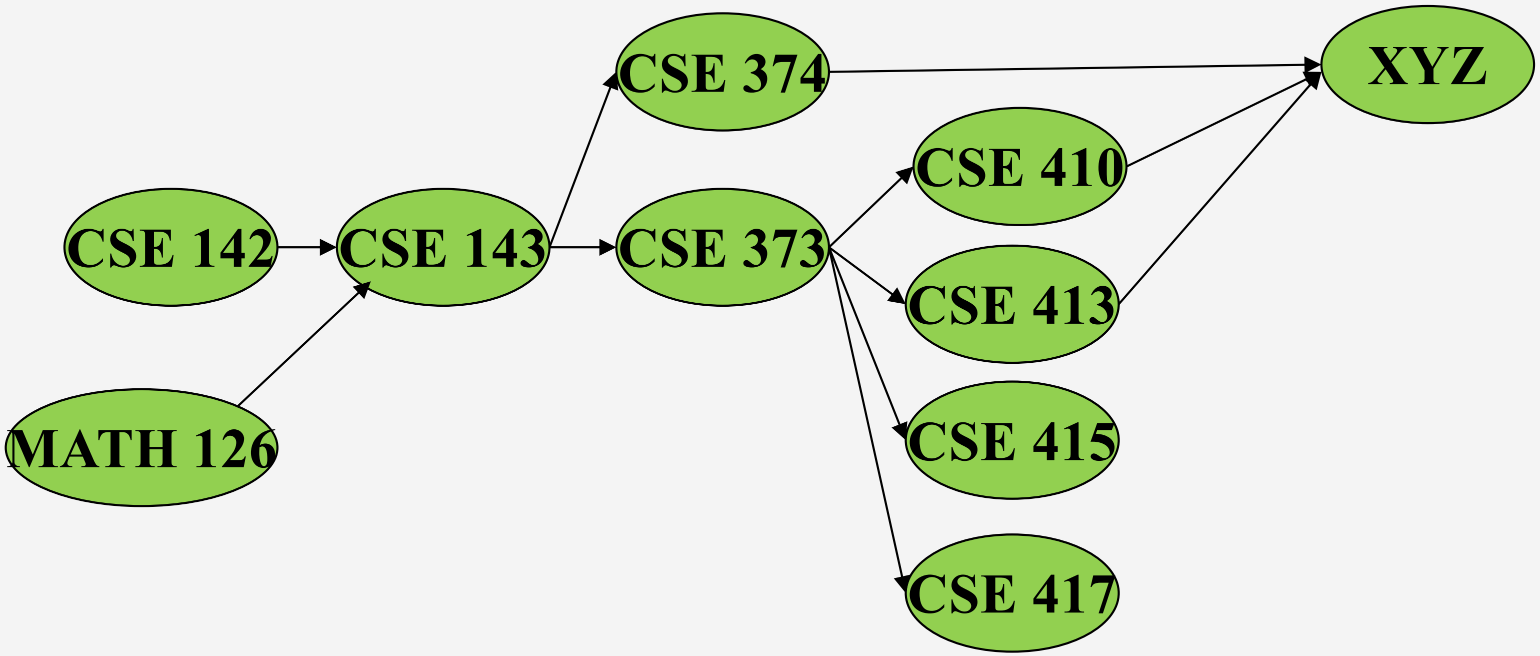
Example



Output:
126
142
143
374
373
417
410
413
XYZ

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x		x	x
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							1
										0

Example



Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x	x	x	x
In-degree:	0	0	2	1	1	1	1	1	1	3
			1	0	0	0	0	0	0	2
			0							1
										0

Output:
126
142
143
374
373
417
410
413
XYZ
415

Notice

- Need a vertex with in-degree 0 to start
  - We can do this because a DAG has no cycles
- Ties among multiple vertices with in-degrees of 0 can be broken arbitrarily
- There are multiple answers to a topological sort

queue based Topological Sort

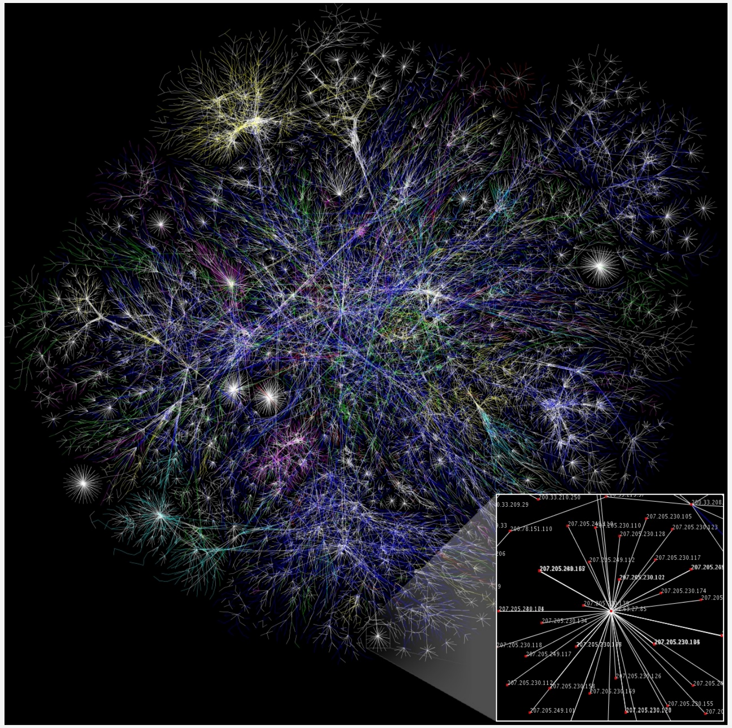
```
void topSort(Graph* G) {
  Queue<int> Q;
  int inDegrees[G->n()];
  int v, w;
  Node *cur;
  for (v=0; v<G->n(); v++) inDegrees[v] = 0;
  for (v=0; v<G->n(); v++) // Process edges
    for (cur=G->adjList[v]; cur!=NULL;
  cur=cur->next )// out-neighbors of vertex v
       inDegrees[cur->nodeID]++;
  for (v=0; v<G->n(); v++) // Initialize Q
    if (inDegrees[v] == 0)// No in-neighbors
      Q->enqueue(v);
  while (Q->length() > 0) {
    Q->dequeue(v);
    printout(v);     // PreVisit for V
    for (cur=G->adjList[v]; cur!=NULL;
  cur=cur->next ) {
      w = cur->nodeID;
      inDegrees[w]--;     // One less in-neigb.
      if (inDegrees[w] == 0) // Now free
        Q->enqueue(w);
    }}}
```

Running time

- Initializing queue Q, array inDegrees takes Θ (n+m) (assuming adjacency list)
- Notice that each vertex enqueues only once, and explore its out-going neighbors when it dequeues from queue Q
  - Takes time Θ (n+m)
- Total time: Θ (n+m)

6. Applications of shortest paths

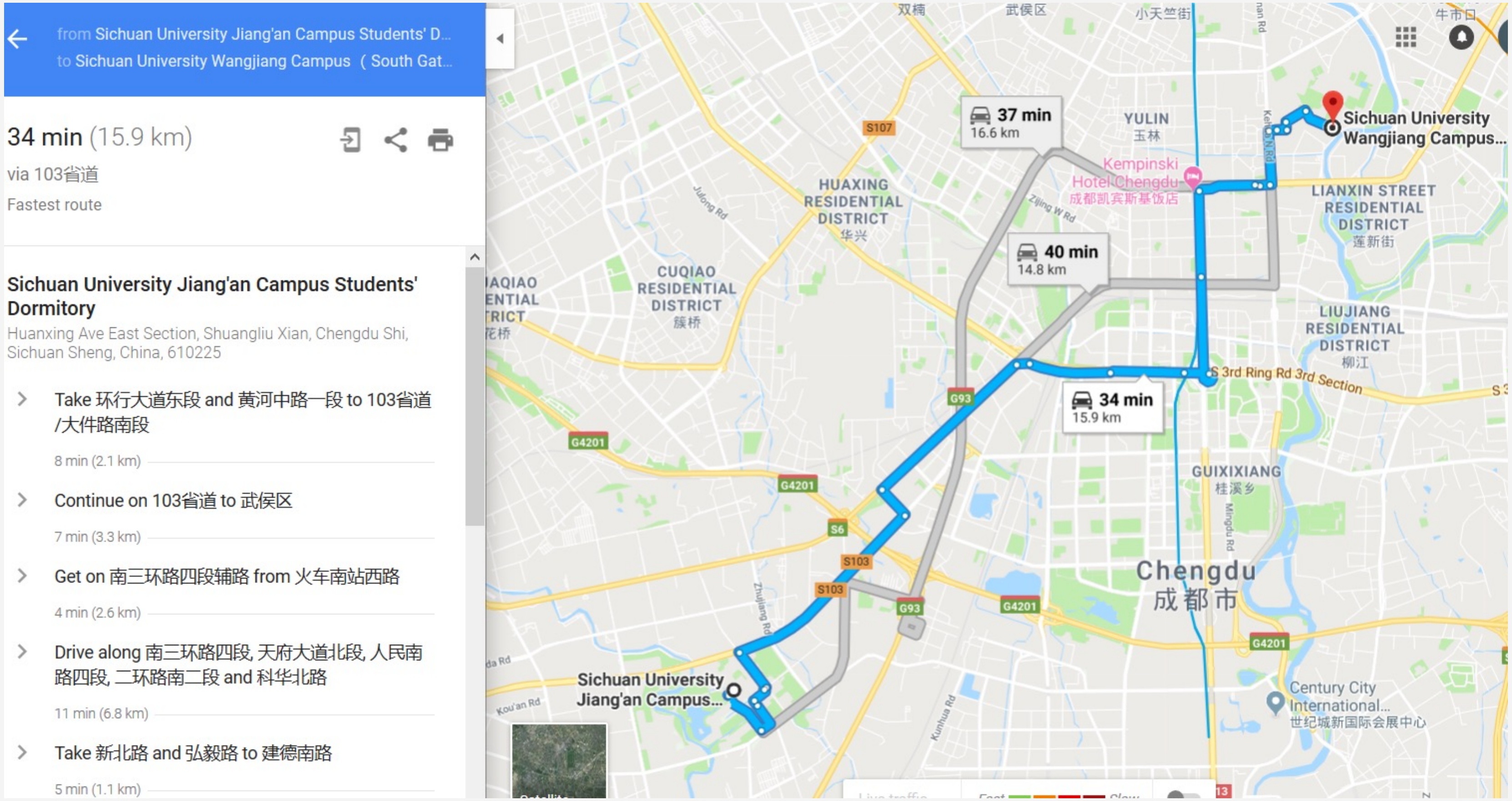
- The Internet is a collection of interconnected computer networks
- Information is passed from a source host, through routers, to its destination server
- e.g. a portion of Internet



- How to send the information along some routers with shortest delay?

Application – google map navigation

- The driving path from Jiang'an campus to Wangjiang campus

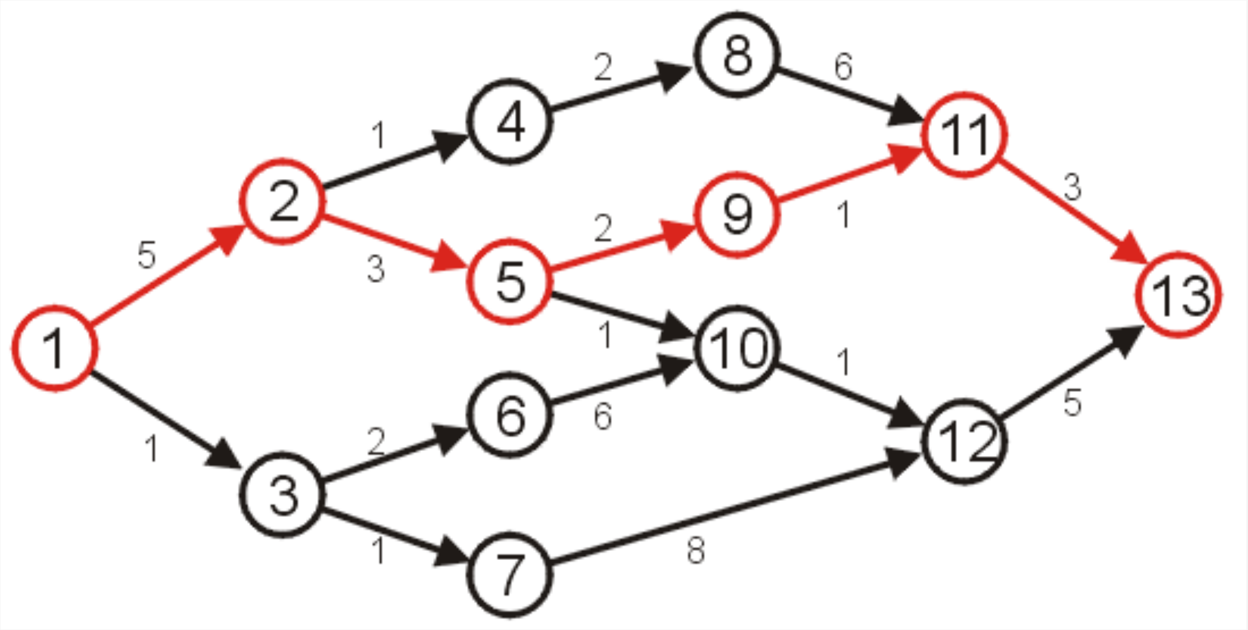


6. Shortest Paths Problems

- Problem 1: Given a weighted graph, one common problem is to find the shortest path from a source vertex ***s*** to a destination vertex ***t***
- Problem 2: find shortest paths from a source vertex ***s*** to all other vertices
- The problem 1 is not easier than problem 2

Shortest Path

- Find the shortest path from vertex 1 to vertex 13
- Path 1→2 → 5 → 9 → 11 → 13 is shortest, with distance 14
- Other paths are longer, e.g,
  - path 1→2 → 4 → 8 → 11 → 13, distance is 17

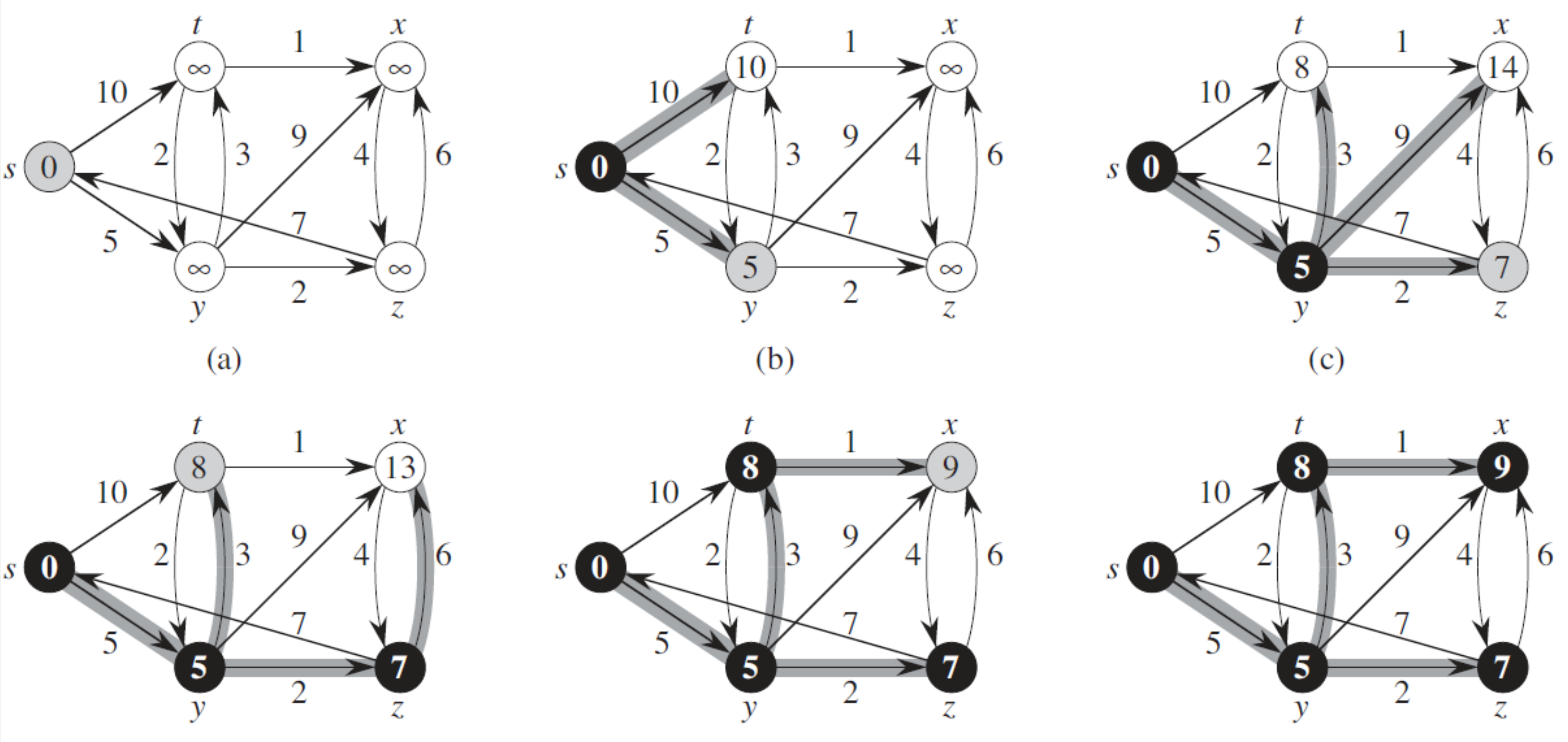


Basic idea of Dijkstra’s algorithm

- Find shortest paths from a *source* vertex ***s*** to other vertices
- It first estimates the shortest distance to each vertex
- Assume that we have found the shortest paths from s to a set ***S*** of vertices
- It repeatedly selects the vertex *u* in ***V\S*** with the minimum shortest-path **estimate**, adds u to S
- After the adding of u, update the shortest distance estimates of vertices still in *V\S*

Example of Dijkstra's algorithm

- The value on each vertex is the shortest distance estimate or shortest distance from s to the vertex



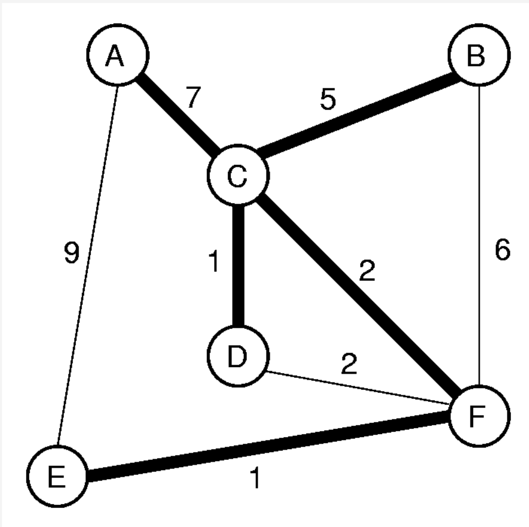
(a) (b) (c) (d) (e) (f)

All-Pairs Shortest Paths

- Calculate the shortest paths for all pairs of vertices
- Run Dijkstra’s algorithm *n* times, each time starting from each vertex

7. Minimum Spanning Tree (MST)

- Given an undirected, connected graph G=(V, E), and an edge weight function: w: E->*R*,
- the minimum spaning tree is a spanning tree T=(V, E') of G such that the weighted sum of edges in T is minimized
  - A spanning tree T=(V, E') of G is a subgraph of G so that the subgraph contains no cycles and spans vertices in V

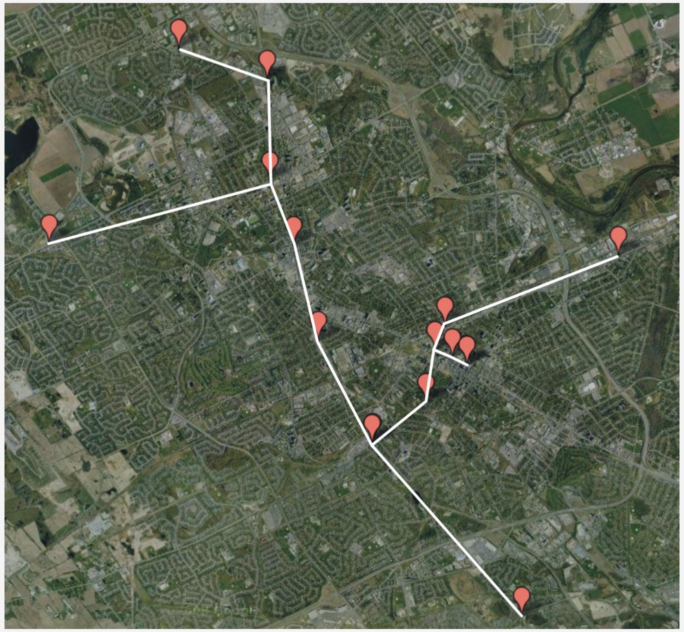


Applications of MST

- Direct applications in
  - Computer networks
  - telecommunication network
  - transportation networks
  - water supply networks
  - electrical grids
- Invoked as a subroutine for other problems
  - Approximating the travelling salesman problem
  - Steiner tree problem

An application example of MST in telecommunication networks

- A telecommunication company wants to lay cables to a new neighbourhood and must bury cables along roads. G=(V, E), w: E->R
  - Each vertex is V represents a building
  - Each edge (u, v) in E represents the road connects buildings u and v
  - w(u,v): the cost of burying cables to connect buildings u and v
- How to lay cables to connect the buildings so that the total cost is minimized?

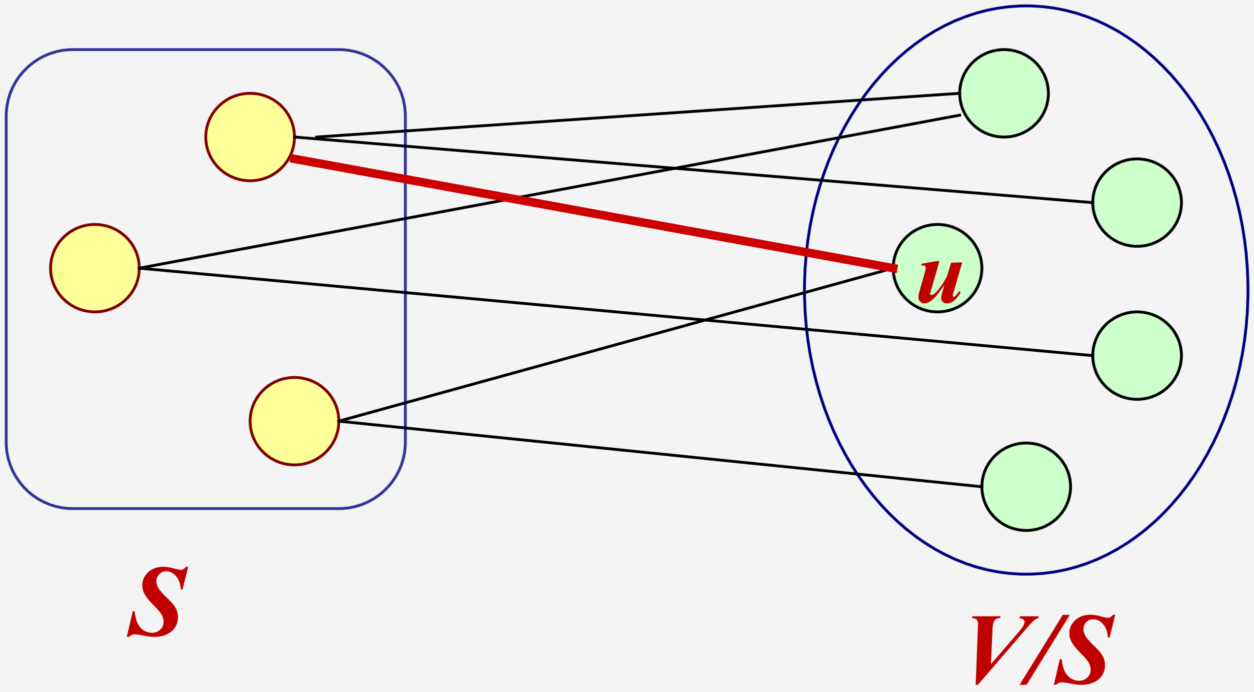


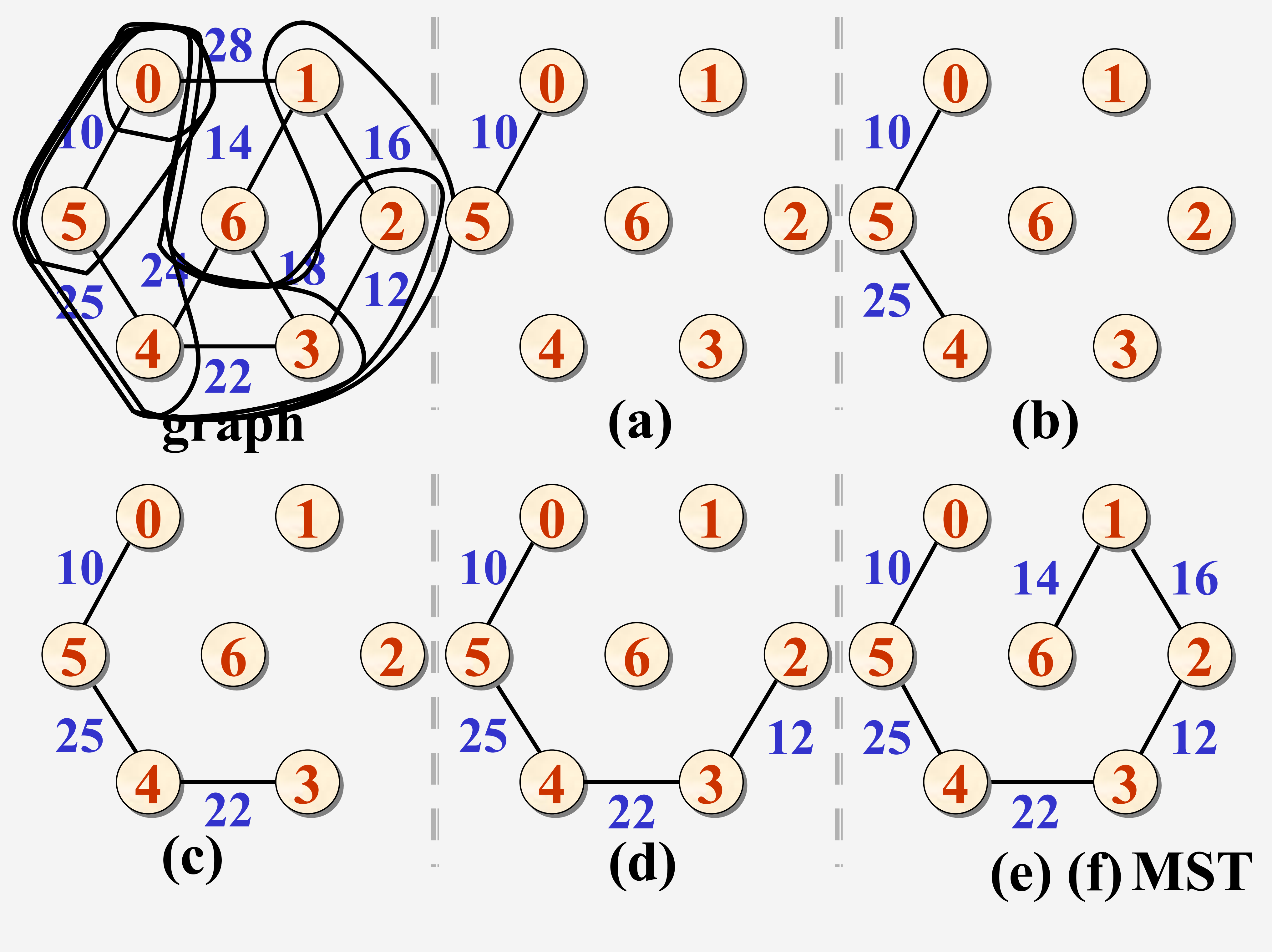
Two optimal algorithms to the MST problem

- Kruskal's algorithm
  - $\Theta(n+m*\log n)$
  - $m = |E|$, $n = |V|$
- Prim's algorithm
  - $\Theta(m+ n*\log n)$
- Both construct the MST in a greedy way
- Introduce the Prim's algorithm as follows, as it is usually faster than Kruskal's algorithm

Basic idea of Prim's Algorithm

- The MST *T* grows from a single vertex
- Assume that *T* has already spanned some vertices in set S, iteratively extend *T* by removing the nearest vertex *u* in set *V*\S to *S*.
- After (n-1) times of growing, *T* spans all nodes in V




graph
(a)
(b)
(c)
(d)
(e) (f) MST

Conclusions

1. Applications of graphs
2. Notations in graphs
3. Graph representations in computers
4. Graph traversals
5. Topological sort
6. Shortest Path
7. Minimum Spanning Tree

Study Four common problems in graphs

Homework 4

- See course webpage
- Deadline: midnight before next lecture
- Submit to: cs_scu@foxmail.com
- File name format:
  - CS311_Hw4_yourID_yourLastName.doc (or .pdf)