CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 24. Maximum Flow

Overview

- In the data structures course (CSE3080), we learned the graph as an advanced data structure, and looked at some graph problems such as finding minimum spanning trees and calculating shortest paths.

- In this chapter, we model a "flow network" as a directed graph and solve some related problems.

- Flow networks: A system of nodes where a material is produced at the source, flows through the network, and consumed at the sink.
  - The source produces the material at some steady rate, and the sink consumes the material at the same rate.
- Example of flow networks
  - Liquids flowing through pipes
  - parts flowing through assembly lines
  - current flowing through electrical networks
  - information flowing through communication networks

Overview

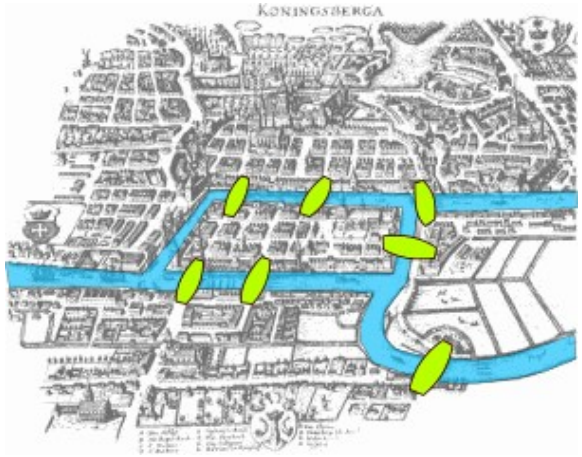
- Elements of a flow network
  - Each directed edge in a flow network is a conduit (pipe) for the material
  - Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit.
    - e.g.) 200 gallons of liquid per hour through a pipe
    - e.g.) 20 amperes of electrical current through a wire
  - Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them.
    - The rate at which material enters a vertex must equal the rate at which it leaves.
    - This property is called "flow conservation".
    - e.g.) When the material is electrical current, flow conservation is equivalent to the "Kirchhoff's current law".

- The maximum-flow problem
  - The goal of the maximum-flow problem is to compute the greatest rate for shipping material from the source to the sink without violating any capacity constraints.

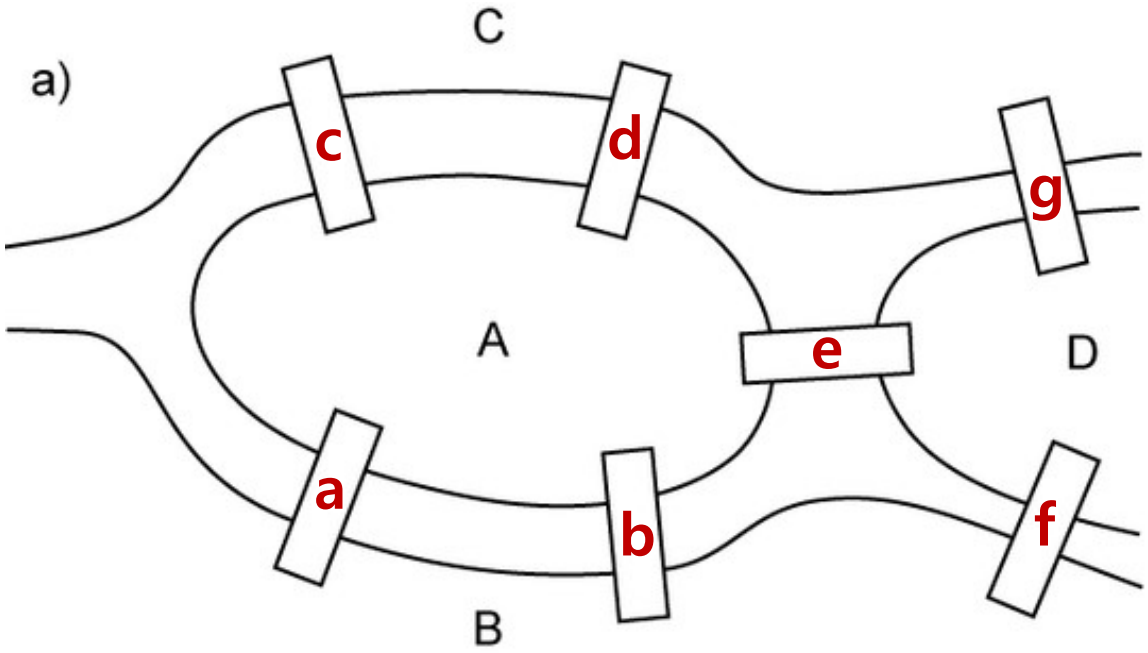
24.0 Graphs: Review

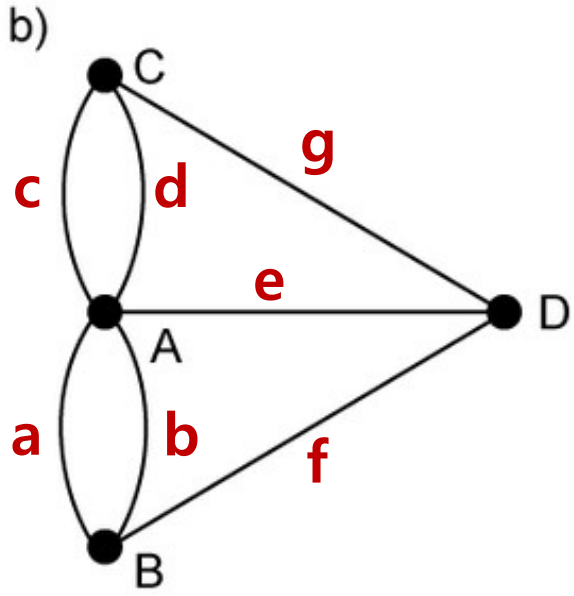
This review is from the lecture sides of CSE3080 Data Structures.

Introduction



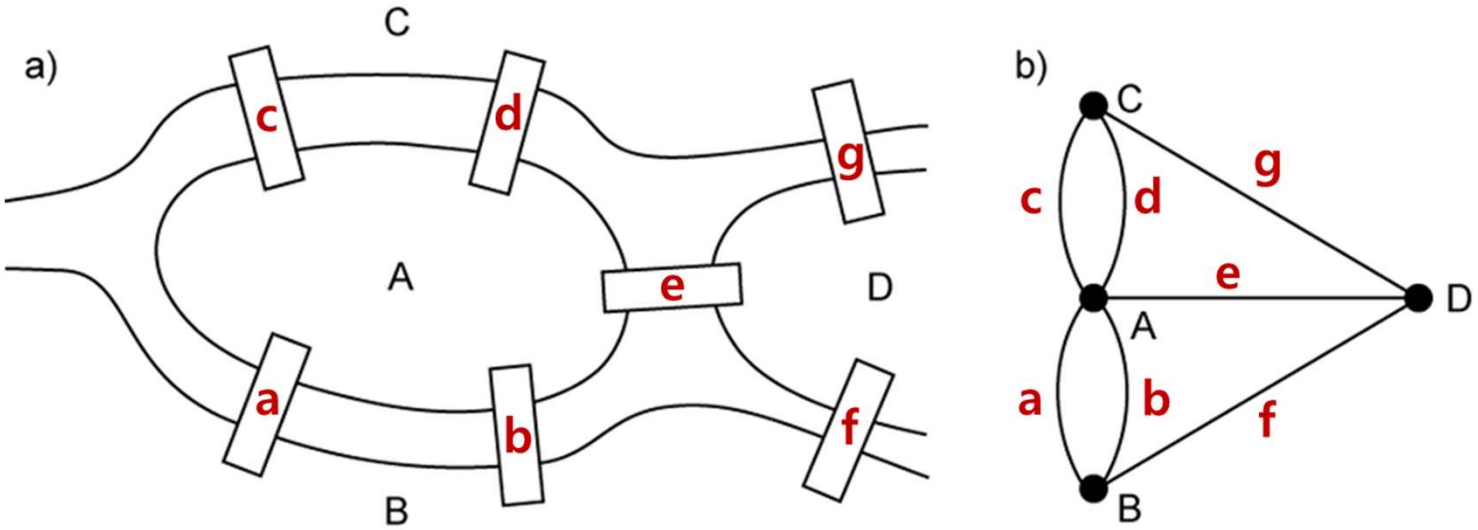
- The Königsberg bridge problem
  - In the town of Königsberg, four land areas divided by the river Pregel are interconnected by seven bridges.
  - The problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area.
  - In 1736, Leonhard Euler used the concept of a graph to address this problem.





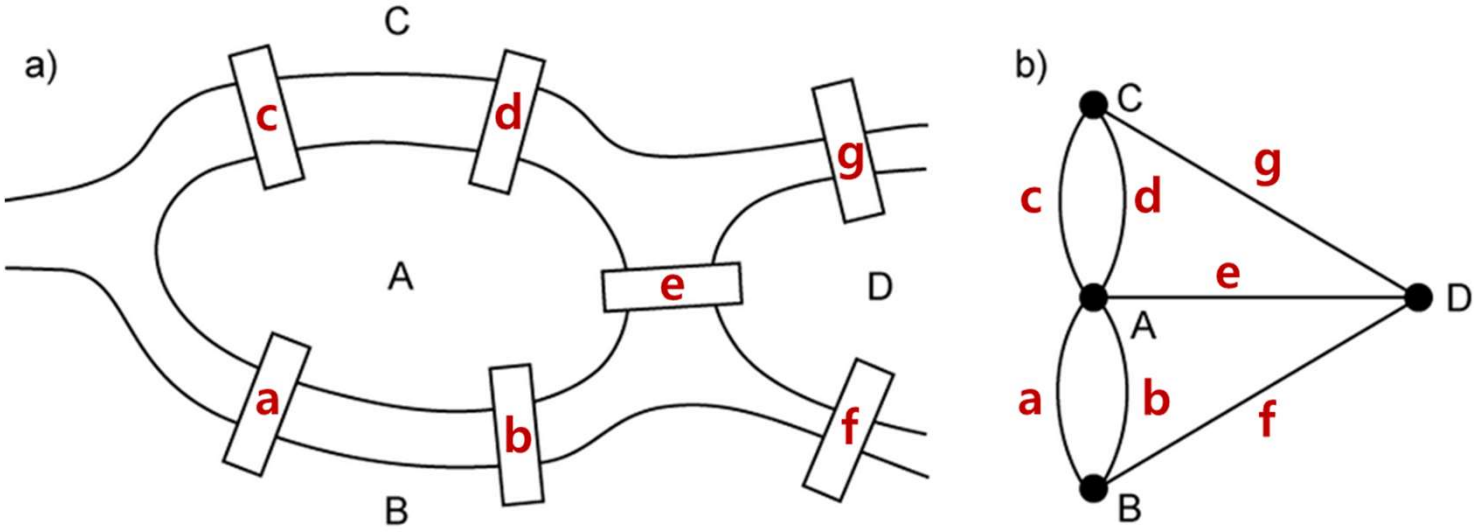
Introduction

- An example walk in the town of Königsberg
  - start from area B
  - walk across bridge a to area A
  - take bridge e to area D
  - take bridge g to C
  - take bridge d to A
  - take bridge b to B
  - take bridge f to D
- This walk does not go across all bridges exactly once, nor does it return to the starting point.



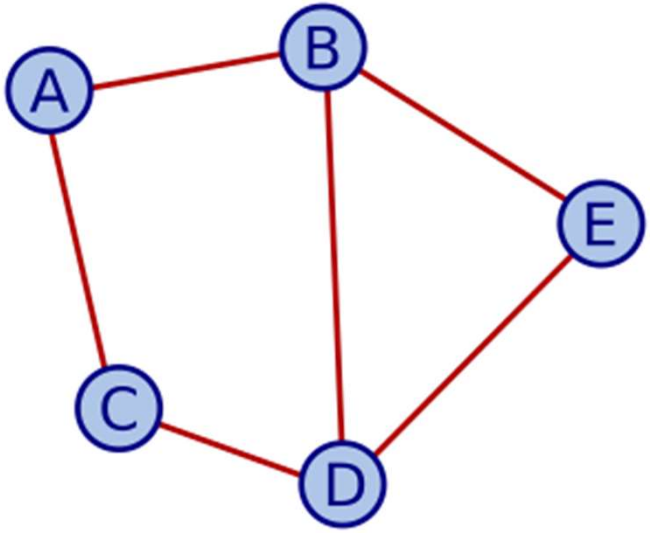
Introduction

- Euler's solution
  - We can draw a graph which consists of vertices and edges.
  - Each land area will become a vertex in the graph.
  - Each bridge will become an edge that connects vertices corresponding to the land areas it interconnects.
  - We define the degree of a vertex to be the number of edges incident to it.
  - Then, there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even.
  - Since the degree of each vertex is not even in the Könisberg example, we cannot find the walk in this problem.



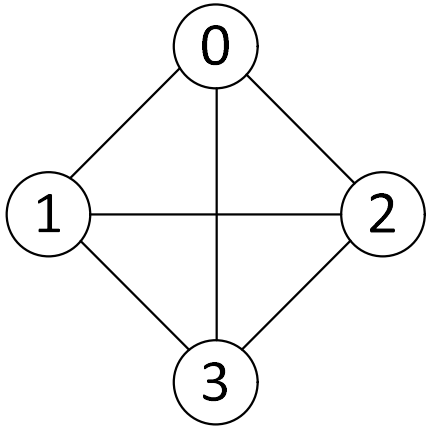
Introduction

- Applications of graphs
  - analysis of electrical circuits
  - finding shortest routes
  - project planning
  - identification of chemical compounds
  - statistical mechanics
  - genetics
  - cybernetics
  - linguistics
  - social sciences
  - etc.
  - A graph is one of the most widely used data structures

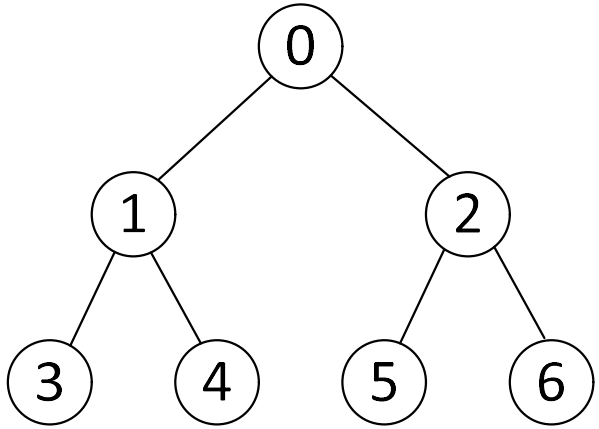


Definitions

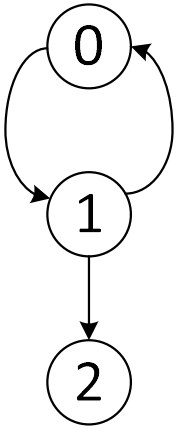
- A graph G consists of two sets, V and E.
- V is a finite, nonempty set of vertices.
- E is a set of pairs of vertices; these pairs are called edges.
- V(G) and E(G) represent the sets of vertices and edges of G.
- We often use G=(V, E) to represent a graph.



G_1



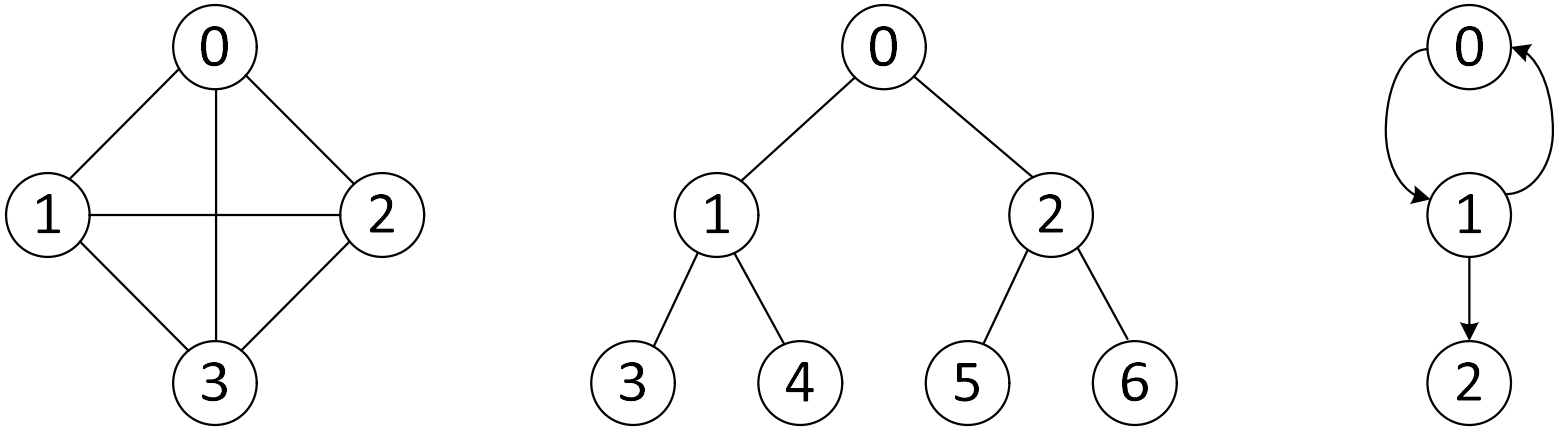
G_2



G_3

Definitions

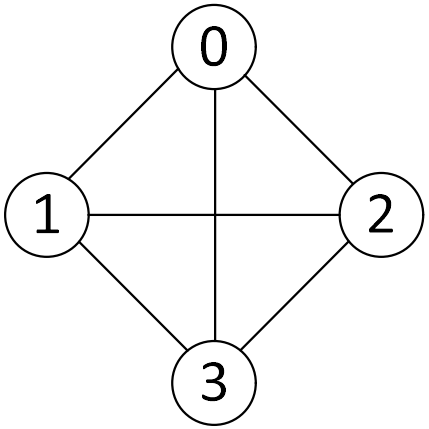
- In an undirected graph the pair of vertices representing any edge is unordered.
  - The pairs (u, v) and (v, u) represent the same edge.
- In a directed graph, each edge is represented by a directed pair <u, v>; u is the tail and v the head of the edge. A directed graph is also called a digraph.
  - The pairs <u, v> and <v, u> represent two different edges.
- G_1 and G_2 are undirected graphs; G_3 is a directed graph.



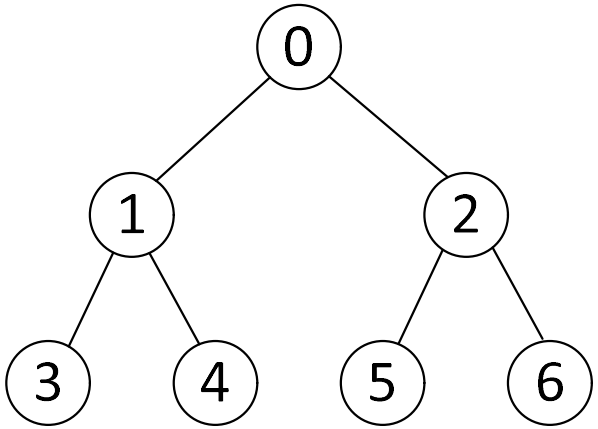
G_1 G_2 G_3

Definitions

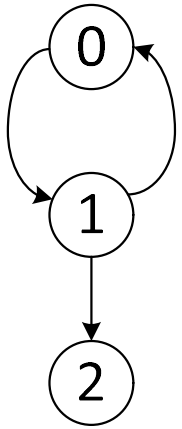
- Set representation of the sets
  - $V(G_1) = \{0,1,2,3\}$; $E(G_1) = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
  - $V(G_2) = \{0,1,2,3,4,5,6\}$; $E(G_2) = \{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
  - $V(G_3) = \{0,1,2\}$; $E(G_3) = \{<0,1>,<1,0>,<1,2>\}$



G_1



G_2

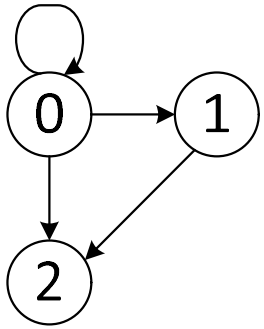


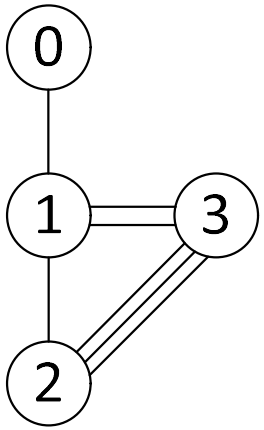
G_3

Definitions

- Restrictions on graphs
  - A graph may not have an edge from a vertex, v, back to itself.
    - (v, v) and <v, v> are not legal.
    - Such edges are called self edges or self loops.
    - Graphs permitting self loops are called a *graph with self loops*.
  - A graph may not have multiple occurrences of the same edge.
    - A graph allowing multiple edges between a pair of nodes is called a multigraph.
    - A graph that does not allow multiple edges between a pair of nodes is called a simple graph to distinguish it from a multigraph.

Graph with a self edge



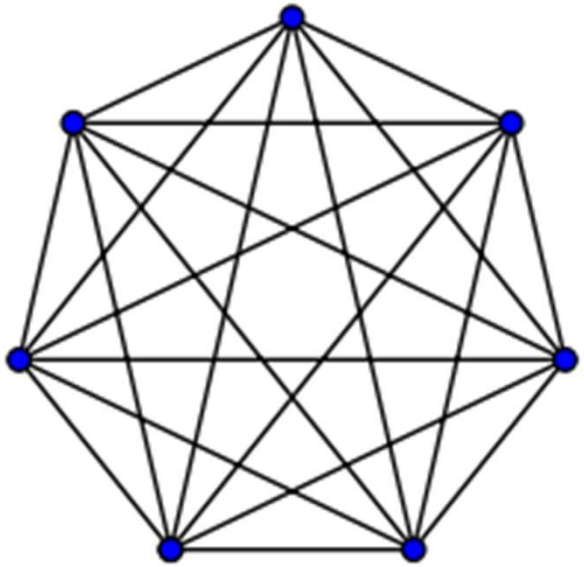


Multigraph

Definitions

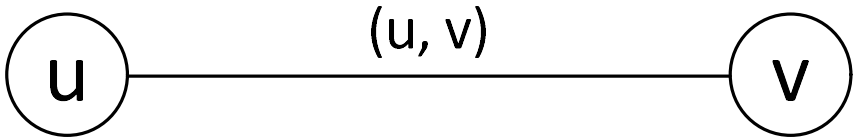
- The maximum number of edges in any n-vertex undirected graph: n(n-1)/2.
  - Number of distinct unordered pairs (u, v).

- If a graph with n vertices has exactly n(n-1)/2 edges, the graph is called a complete graph.

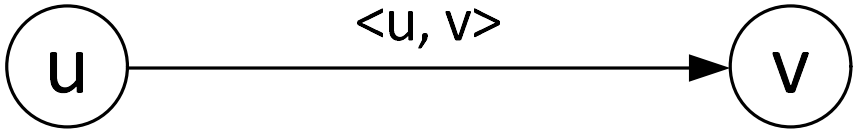


Definitions

- If (u, v) is an edge in E(G), we say the vertices u and v are adjacent, and the edge (u, v) is incident on vertices u and v.



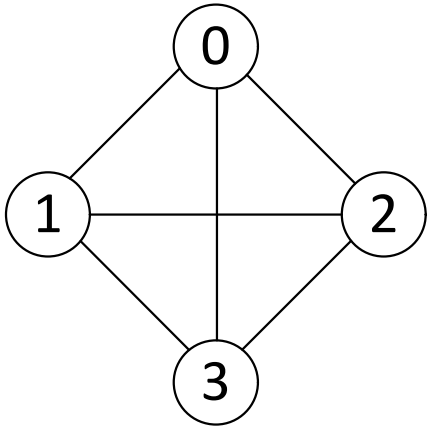
- For a directed edge <u, v>, we say u is adjacent to v, and v is adjacent from u.
  - <u, v> is incident to u and v.



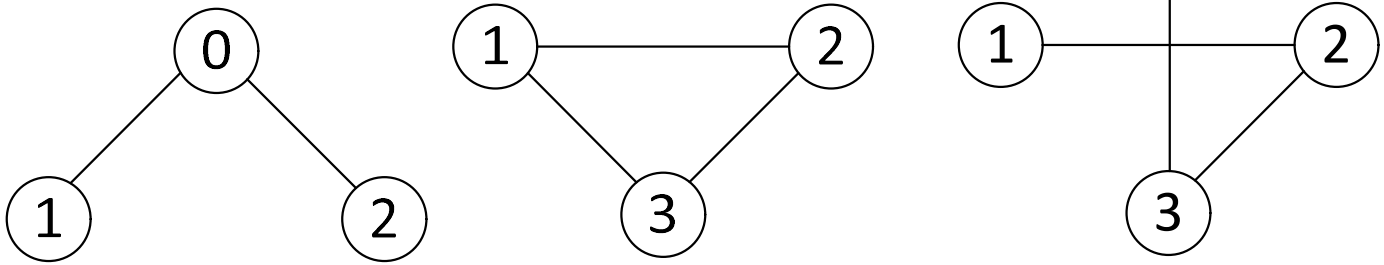
Definitions

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

Graph G



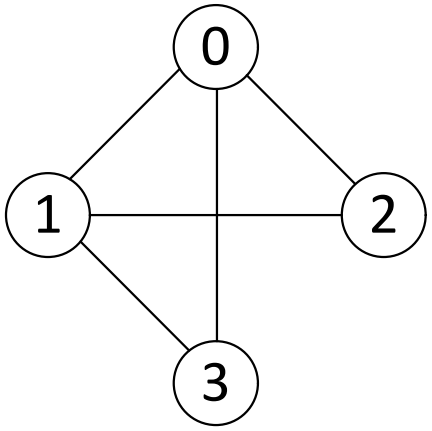
Subgraphs of G



Definitions

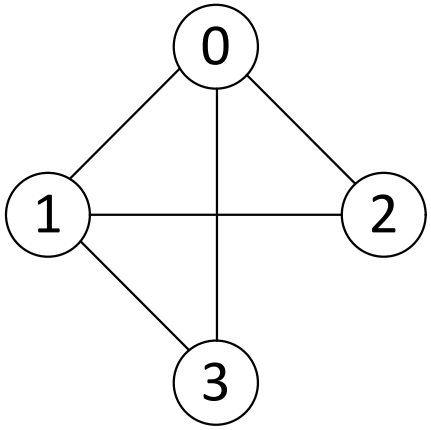
- A path from vertex u to vertex v in graph G is a sequence of vertices u, i_1, i_2, ..., i_k, v such that (u, i_1), (i_1, i_2), ..., (i_k, v) are edges in E(G).
- If G' is directed, then the path consists of <u, i_1>, <i_1, i_2>, ..., <i_k, v> in E(G').
- The length of a path is the number of edges on it.

- An example path from 0 to 3: (0, 1), (1, 2), (2, 0), (0, 3).
  - The length of the path is 4.
- (0, 1), (1, 2), (2, 3) is not a path from 0 to 3 because (2, 3) is not in E(G).



Definitions

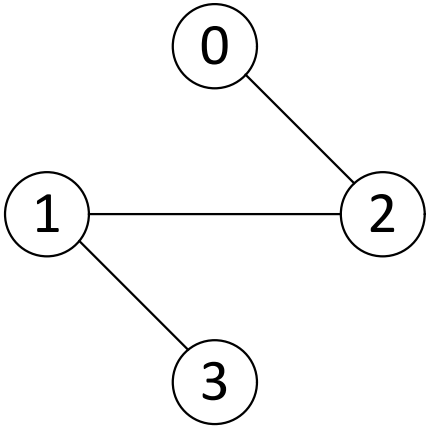
- A simple path is a path in which all vertices except possibly the first and the last are distinct.
  - (0, 2), (2, 1), (1, 3) is a simple path from vertex 0 to vertex 3.
  - (0, 2), (2, 1), (1, 3), (3, 0) is a simple path from vertex 0 to vertex 0.
  - (0, 1), (1, 2), (2, 0), (0, 3) is not a simple path because vertex 0 is visited multiple times.
- A cycle is a simple path in which the first and last vertices are the same.
  - (0, 2), (2, 1), (1, 3), (3, 0) is a cycle.



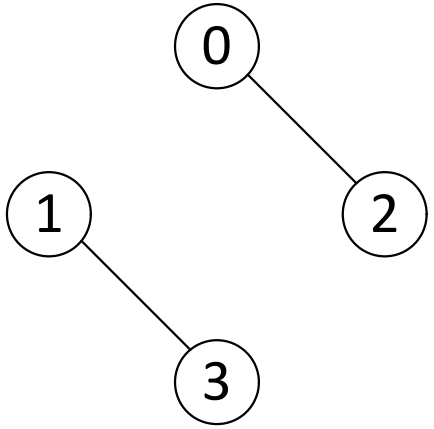
Definitions

- In an undirected graph G, two vertices u and v in V(G) are said to be connected iff there is a path in G from u to v.

- An undirected graph is said to be connected iff for every pair of distinct vertices u and v in V(G) is connected.



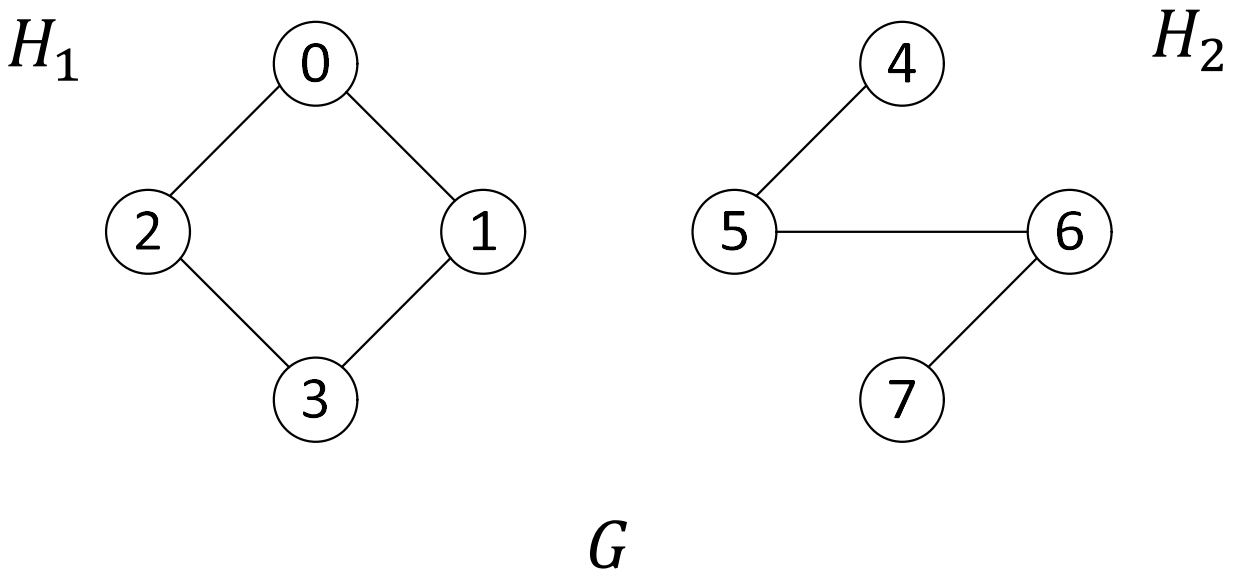
a connected graph



not connected

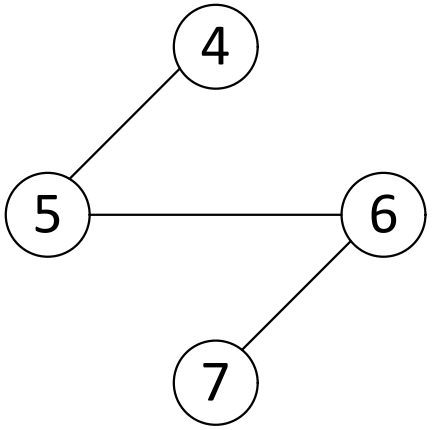
Definitions

- A connected component H of an undirected graph G is a maximal connected subgraph.
  - By maximal, we mean that G contains no other subgraph that is both connected and properly contains H.

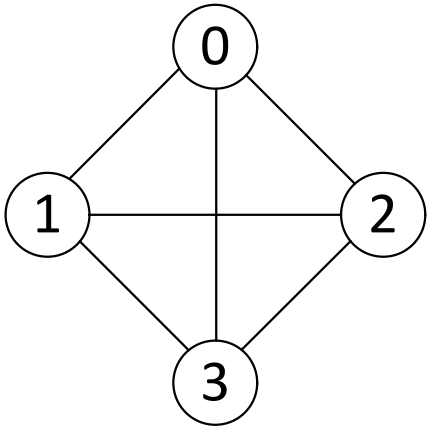


Definitions

- A tree is a connected acyclic graph.
  - Acyclic means the graph has no cycles.



tree

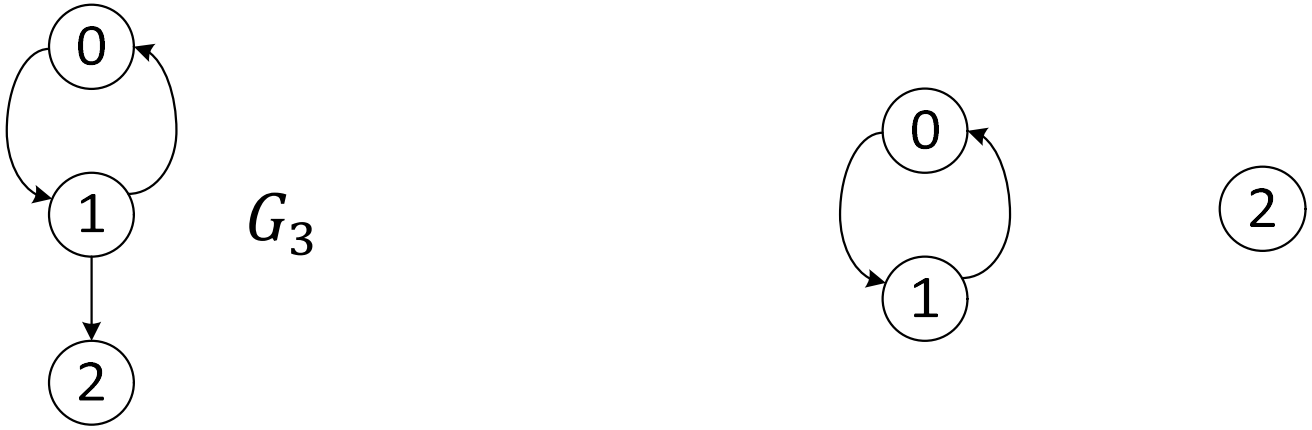


not a tree

Definitions

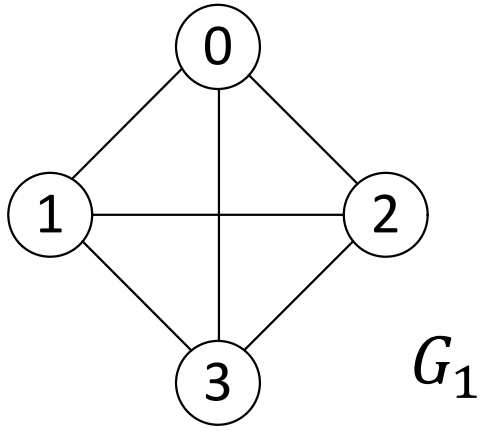
- A directed graph G is said to be strongly connected iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u.
  - G_3 is not strongly connected, as there is no path from vertex 2 to 1.

- A strongly connected component is a maximal subgraph that is strongly connected. G_3 has two strongly connected components.

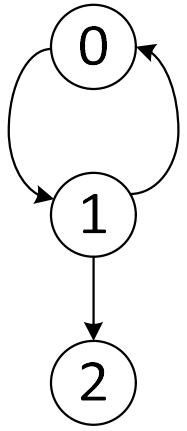


Definitions

- The degree of a vertex is the number of edges incident to that vertex.
  - The degree of vertex 0 in G_1 is 3.
- If G is a directed graph, we define the in-degree of a vertex v to be the number of edges for which v is the head. The out-degree is defined to be the number of edges for which v is the tail.
  - Vertex 1 of G_3 has in-degree 1, out-degree 2, and degree 3.
- If d_i is the degree of vertex i in a graph G with n vertices and e edges, then $e = (\sum_{i=0}^{n-1} d_i)/2$



G_1



G_3

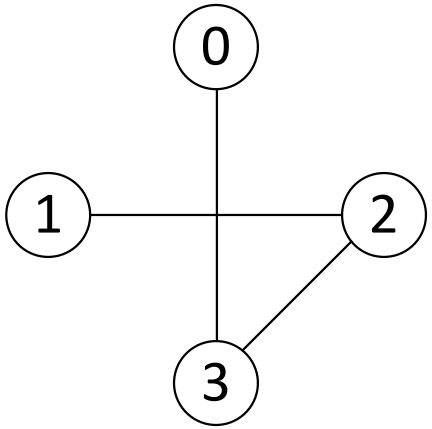
Graph Representations

- The two most commly used representations
  - adjacency matrix
  - adjacency list

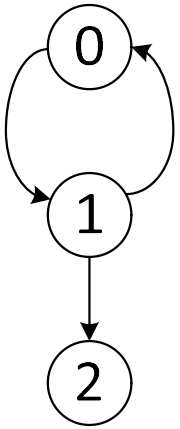
- Choice of a particular representation will depend upon the application

Adjacency Matrix

- The adjacency matrix of G=(V,E) is a two dimensional nxn array, a, with the property that a[i][j] = 1 iff the edge (i, j) is in E(G).
  - a[i][j] = 0 if (i, j) is not in E(G).
  - For a digraph, a[i][j] = 1 if <i, j> is in E(G).
- For a undirected graph, the adjacency matrix is symmetric.
- For a digraph, the adjacency matrix may not be symmetric.



$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

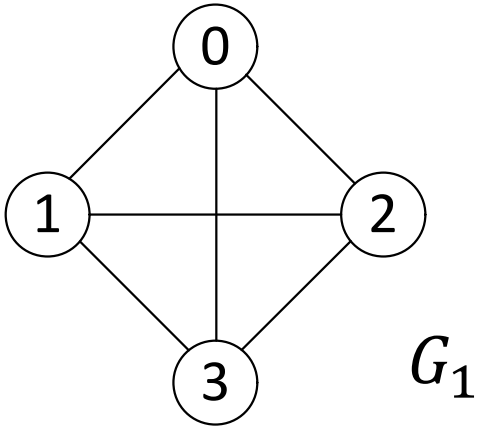
Adjacency Matrix

- Space needed to represent a graph: n^2 bits.
- For an undirected graph, the degree of a vertex i is equal to its row sum: $\sum_{j=0}^{n-1} a[i][j]$
- For a directed graph, the row sum is the out-degree, and the column sum is the in-degree.

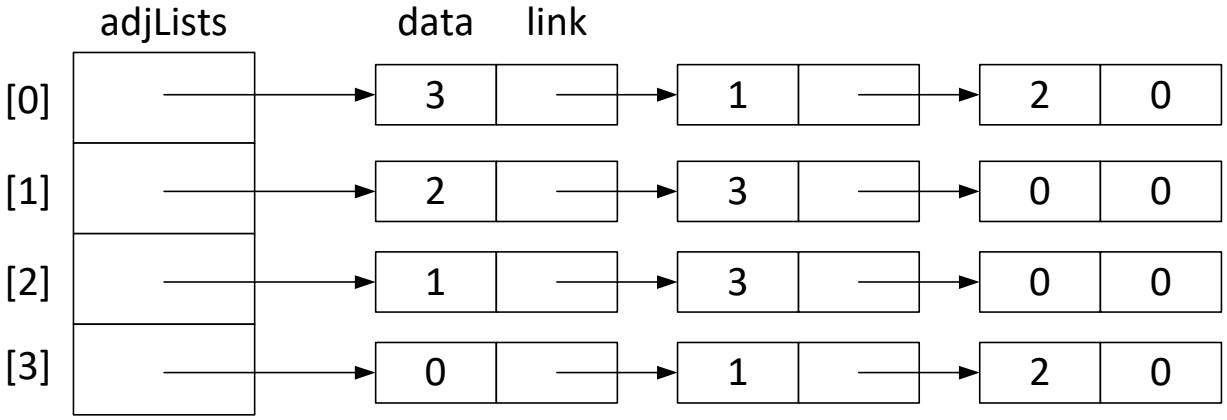
- If we want to answer questions about a graph such as:
  - How many edges are there in G?
  - Is G connected?
- An adjacency matrix requires $O(n^2)$ time.
  - This may be inefficient when number of edges is small. (The graph is sparse.)

Adjacency Lists

- In an adjacency list, the n rows of the adjacency matrix are represented as n chains (or sequential lists).
  - One chain for each vertex in G
  - Nodes in chain i represent the vertices that are adjacent from vertex i.
    - The vertices in each chain are not required to be ordered.
  - An array adjLists is used so that we can access the adjacency list for any vertex in O(1) time.
    - adjList[i] is the pointer to the first node in the adjacency list for vertex i.



G_1



Adjacency Lists

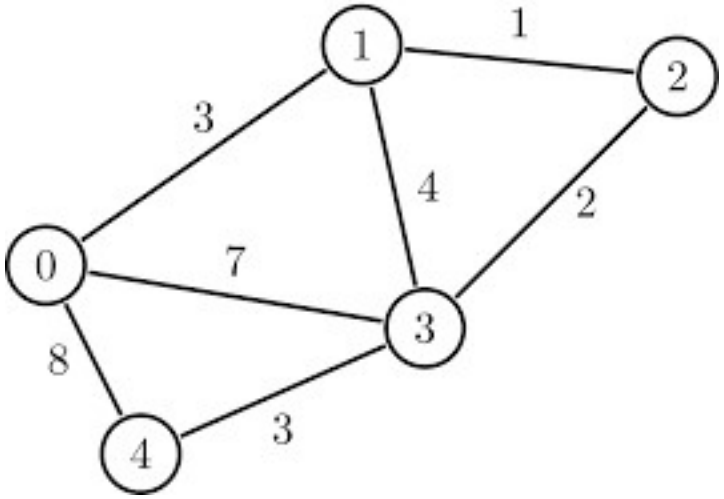
- For a graph with n vertices and e edges, the linked adjacency list representation requires an array of size n and 2e chain nodes.

- In order to answer questions such as
  - How many edges are there in G?
  - Is G connected?
- An adjacency list requires O(n+e) time.

- If e is much smaller than n^2, then using adjacency list can be more efficient than using an adjacency matrix.

Weighted Edges

- In many applications, the edges of a graph have weights assigned to them.
- These weights may represent the distance from one vertex to another or the cost of going from one vertex to another.
- In an adjacency matrix, a[i][j] will now be the edge weight.
- In an adjacency list, the list nodes should have an additional field, weight.
- A graph with weighted edges is called a network.



Elementary Graph Operations

- Here, we look at how to visit vertices that are reachable from a starting vertex.
  - In other words, starting from a vertex v, we visit all vertices connected to v.

- Two ways of visiting connected vertices
  - depth first search
  - breadth first search

Depth First Search (DFS)

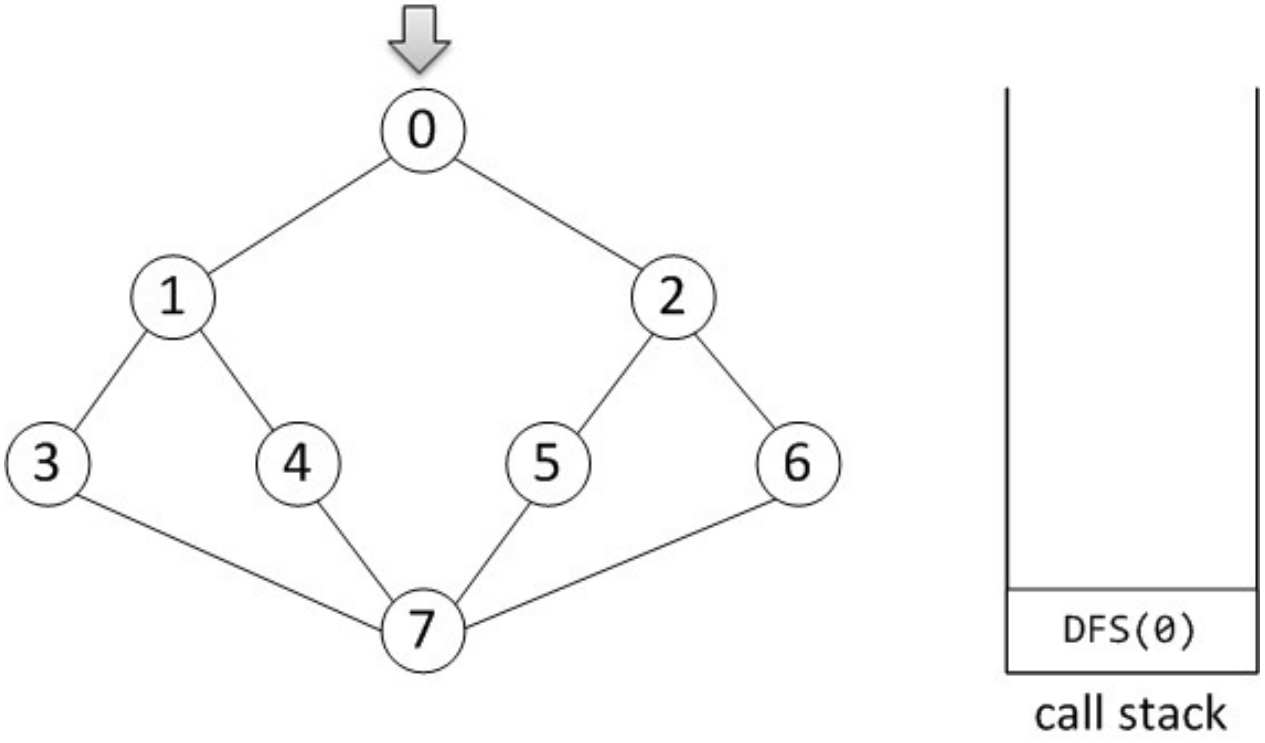
- The procedure of function DFS
  - Initially, we are given a starting vertex v, where we begin our search.
  - First, we "visit" the starting vertex v.
    - Here, we assume "visiting a vertex" means printing out its name.
  - Then, we select an unvisited vertex from v's adjacent vertices.
  - We carry out a DFS on that vertex.
    - Call function DFS recursively.

- In order to implement this procedure, we first need to select a representation between an adjacency matrix and an adjacency list.

- Also, we need a separate data structure for checking whether a vertex is visited or not.
  - We could use an array of size n, the number of vertices.

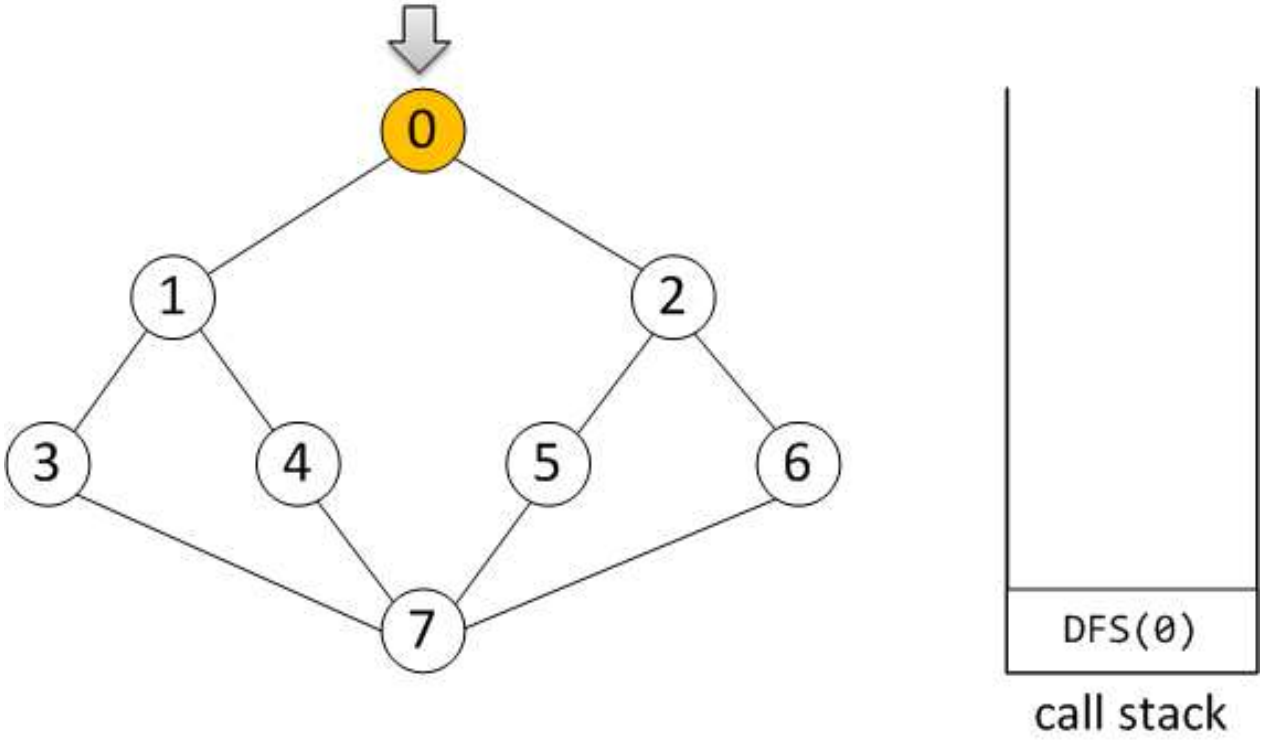
Depth First Search (DFS)

- Suppose we would like to perform DFS on graph G, starting from vertex 0.
- So we make vertex 0 the "current" vertex.
  - Call DFS on vertex 0.



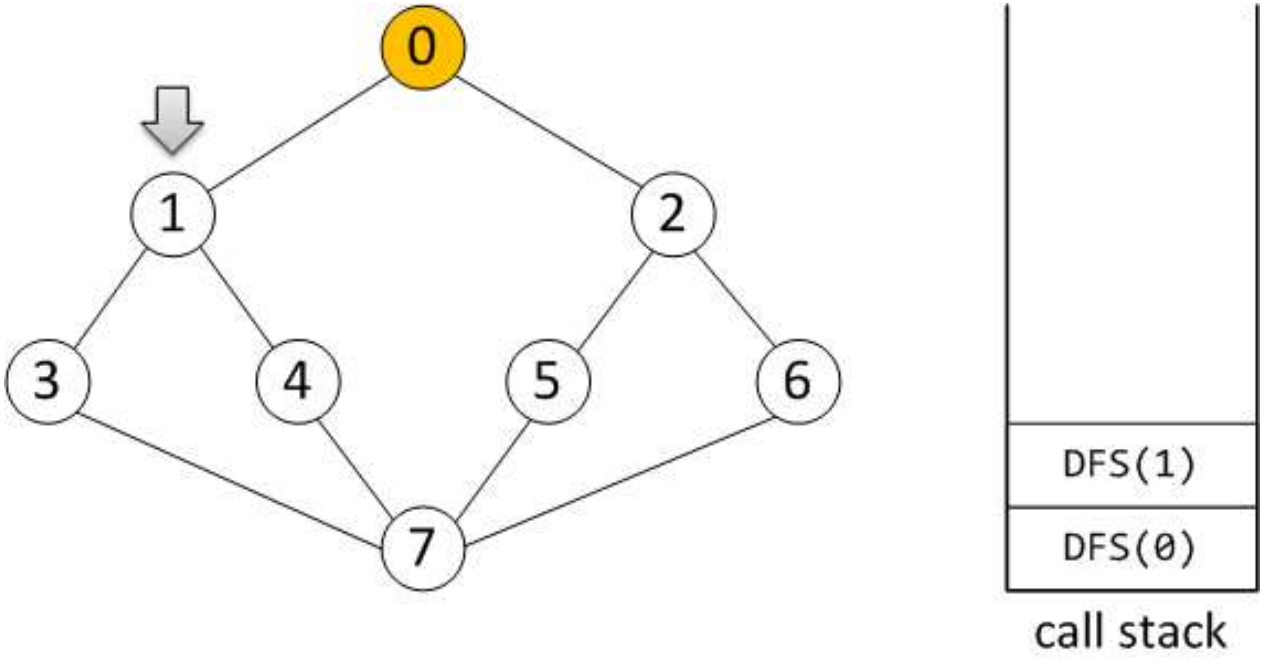
Depth First Search (DFS)

- First, we visit the current vertex.
- After visiting a vertex, we mark the vertex so we know it is already visited.



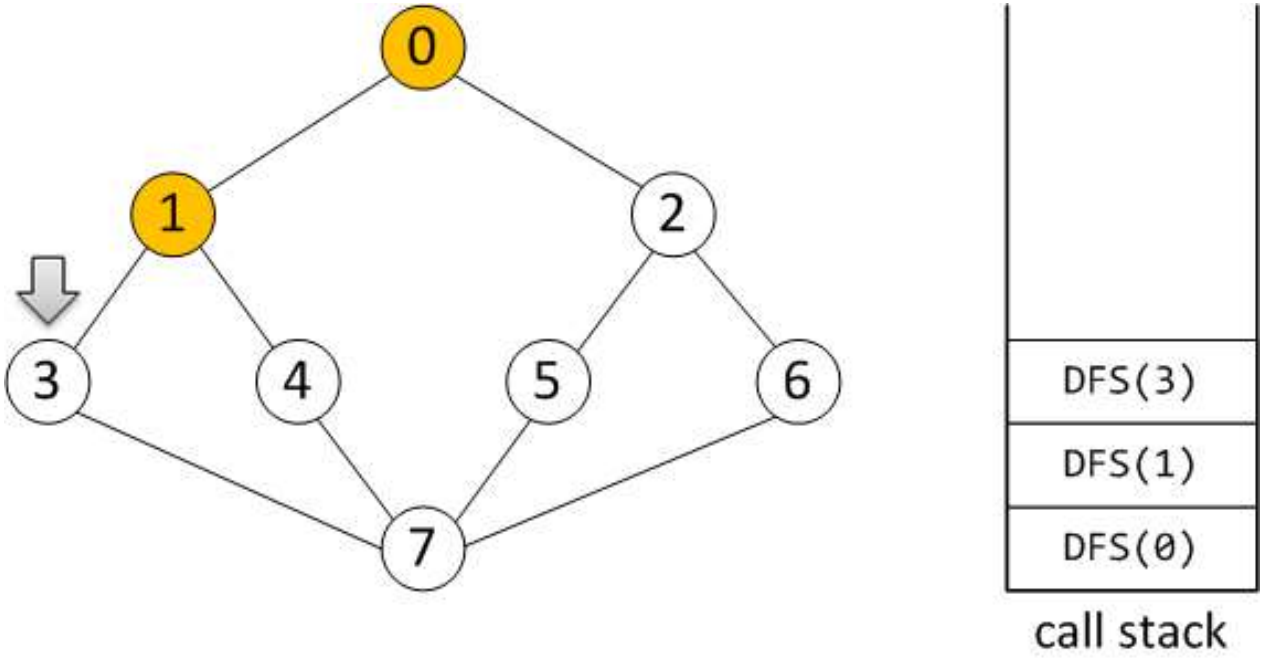
Depth First Search (DFS)

- Then, we pick one of the vertices that is
  - adjacent to the current vertex, and
  - is unvisited yet.
  - candidates are vertex 1 and vertex 2.
- We pick vertex 1, and so we call DFS on that vertex.



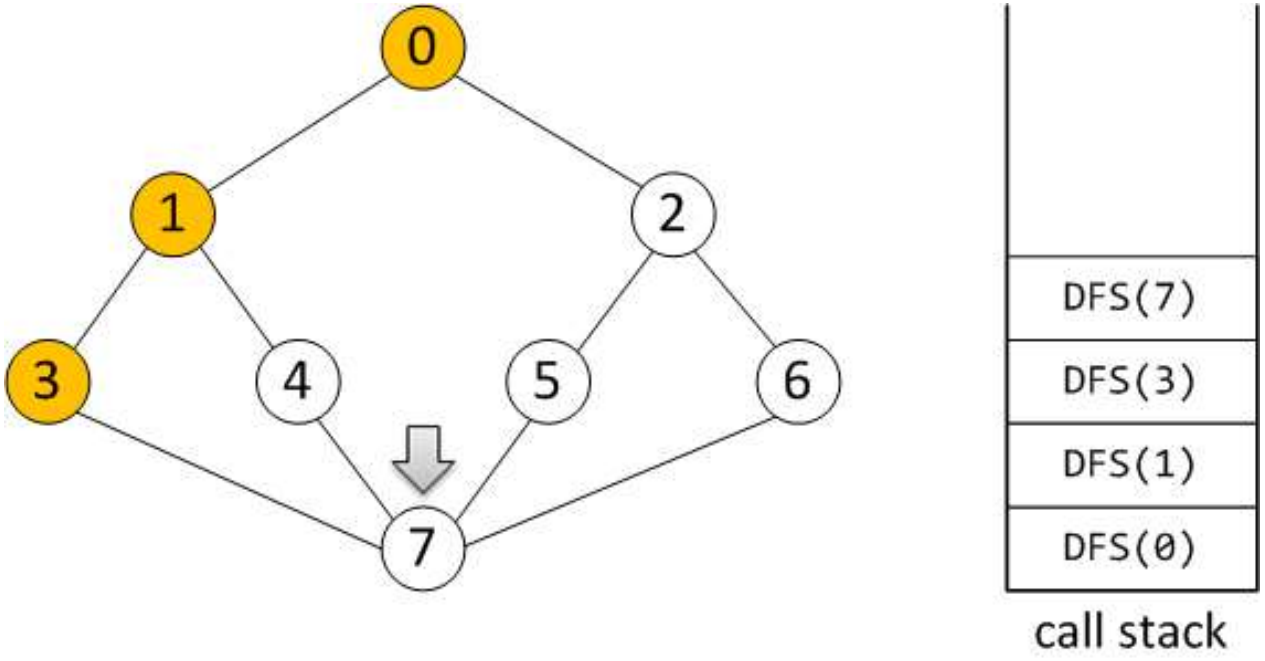
Depth First Search (DFS)

- Here, we go through the same procedure.
- We visit the current vertex, which is vertex 1.
- Then, we pick an adjacent vertex of 1 which is unvisited.
  - candidates are vertex 3 and 4.
- We pick vertex 3, and so we call DFS on vertex 3.



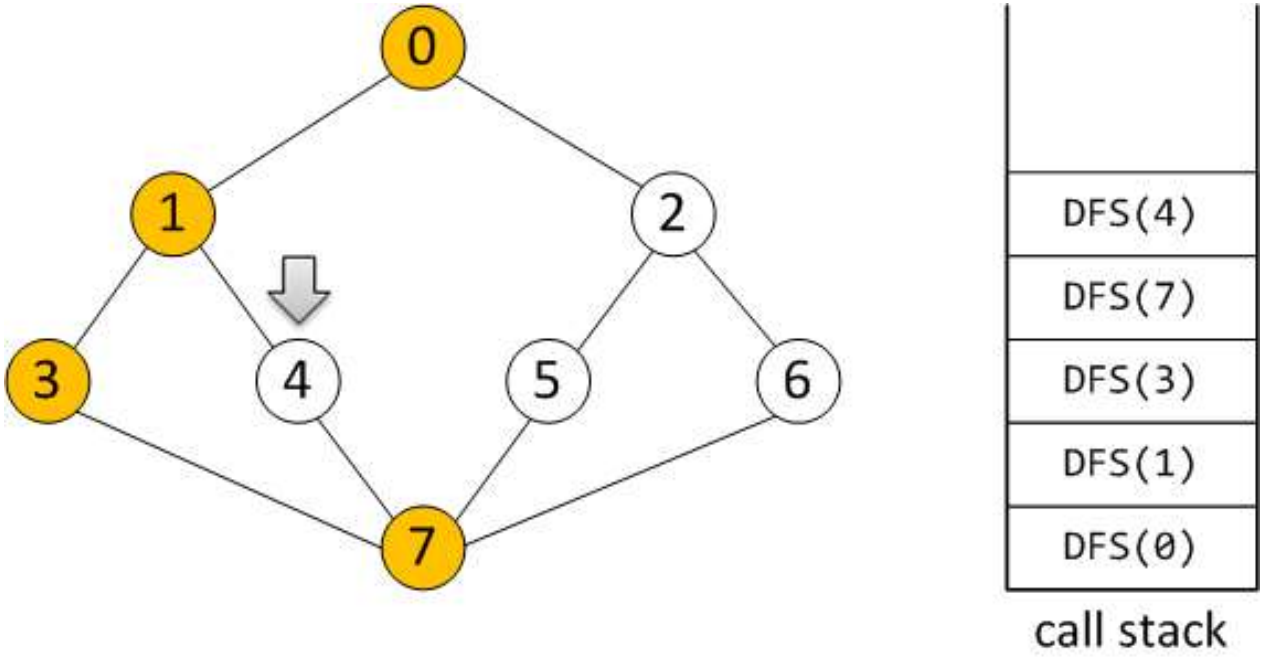
Depth First Search (DFS)

- We visit vertex 3, and pick an adjacent vertex that is unvisited.
  - The only candidate is vertex 7.
- We call DFS on vertex 7.



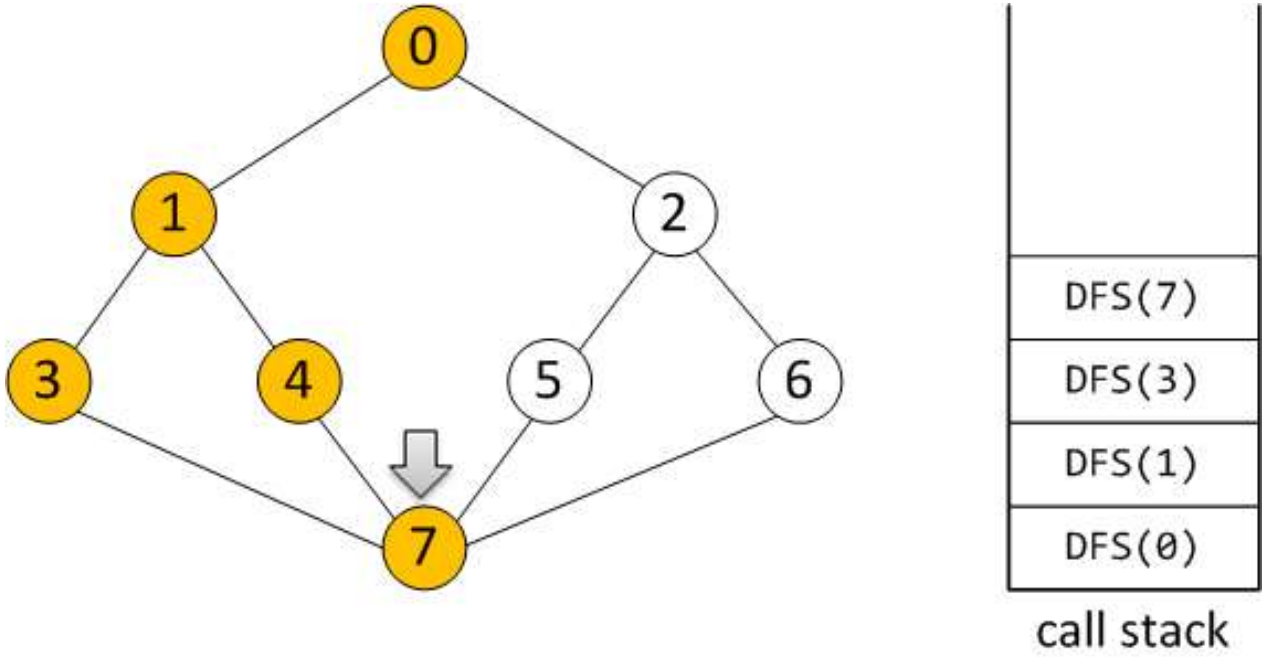
Depth First Search (DFS)

- We visit vertex 7, and pick an adjacent vertex that is unvisited.
  - candidates are vertex 4, 5, and 6.
- We pick vertex 4, and call DFS on that vertex.



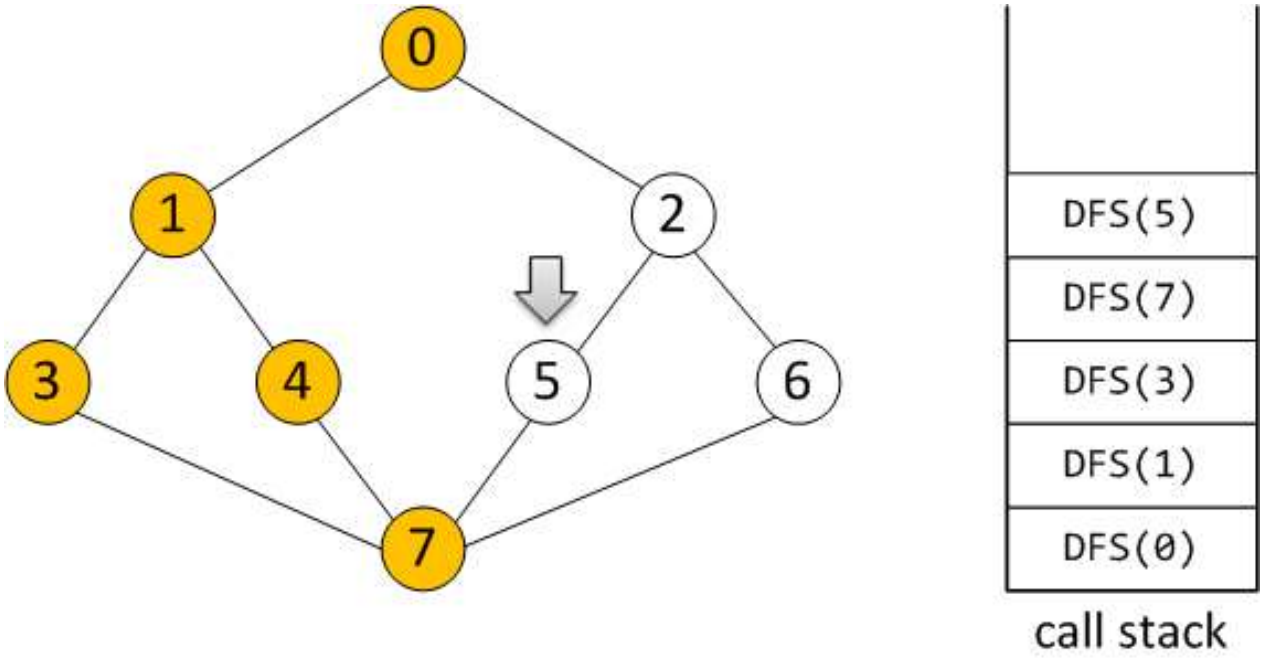
Depth First Search (DFS)

- We visit vertex 4, and pick an adjacent vertex that is unvisited.
- Since there are no vertex that meets the condition, we return to the caller.
  - The caller was DFS(7).



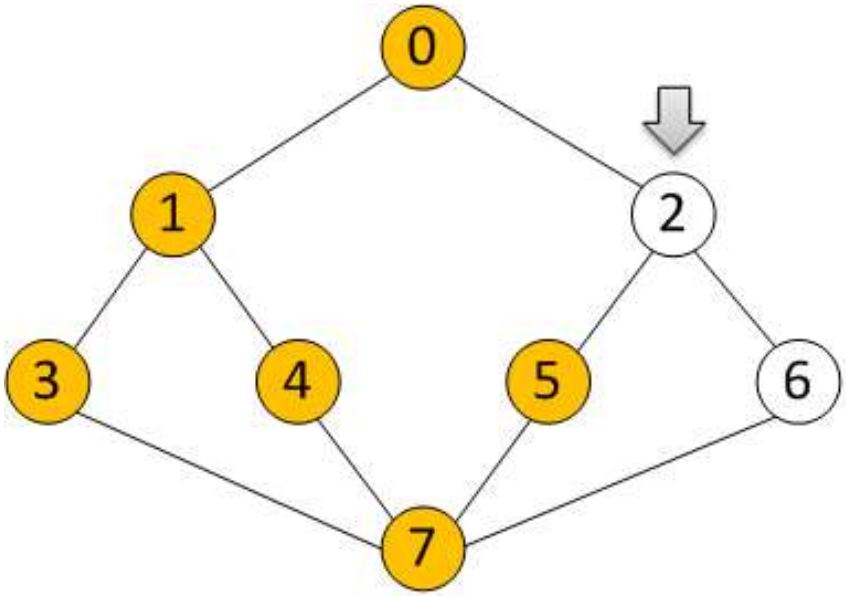
Depth First Search (DFS)

- Since vertex 7 has adjacent vertices that are unvisited, the function continues.
  - We pick vertex 5, and call DFS on that vertex.



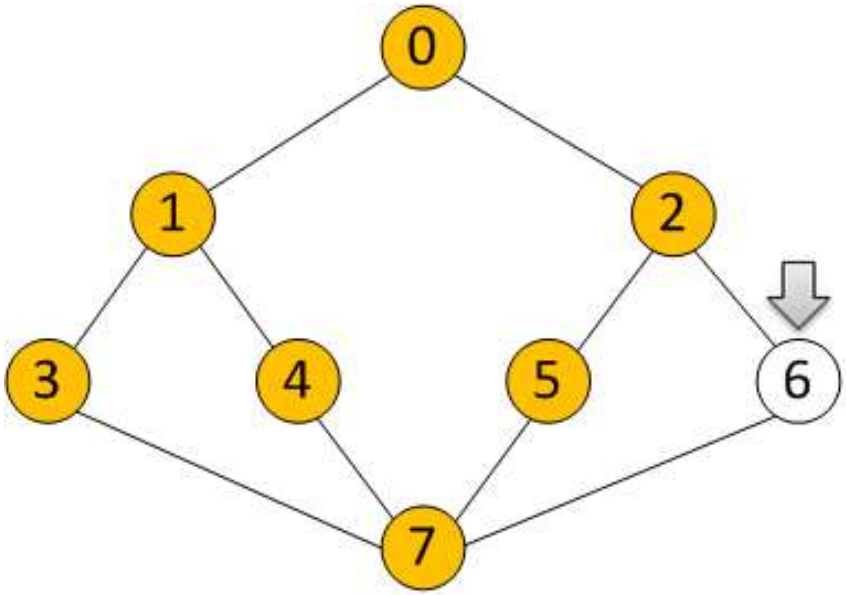
Depth First Search (DFS)

- We visit vertex 5, and then call DFS on vertex 2.



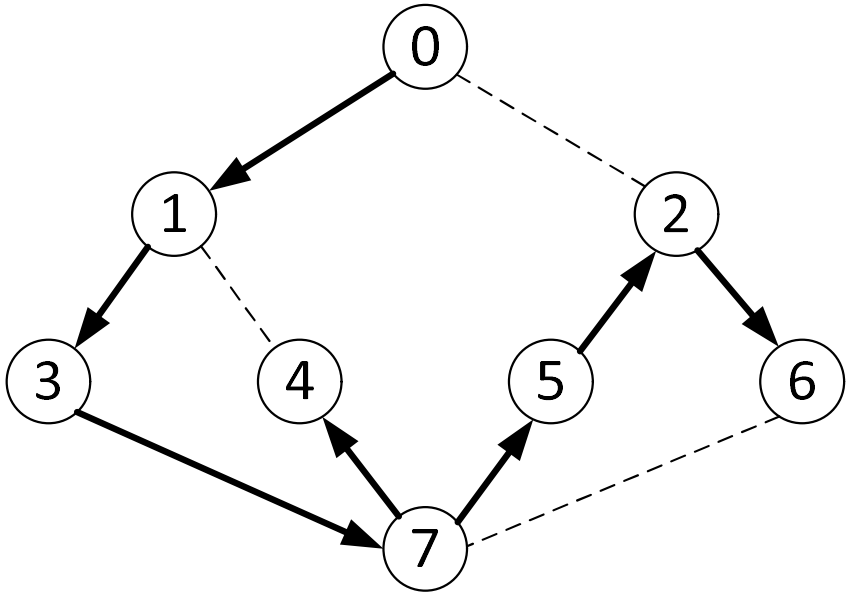
Depth First Search (DFS)

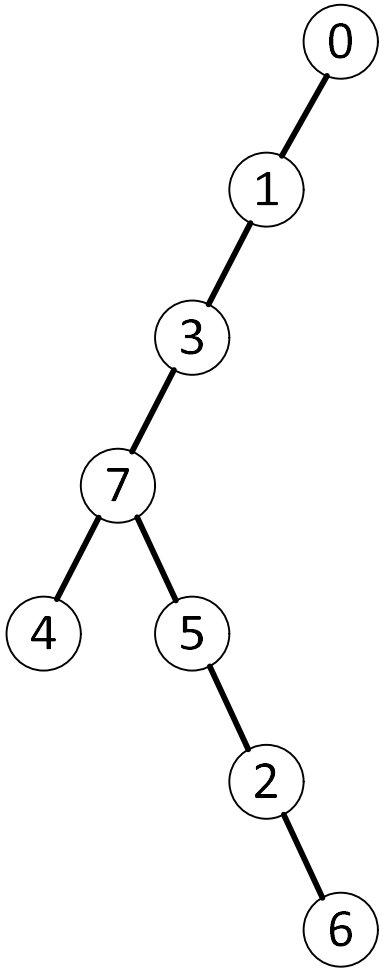
- We visit vertex 2, and we call DFS on vertex 6.
  - There is no unvisited vertex now, so the functions will return.
- All vertices connected to vertex 0 have been searched.



Depth First Search (DFS)

- If we select the edges we passed through and remove all other edges, it becomes a tree.
- In the depth first search, we move far away from the starting vertex until we cannot go further. The resulting tree is "deep", and that is why this scheme is called depth first search.





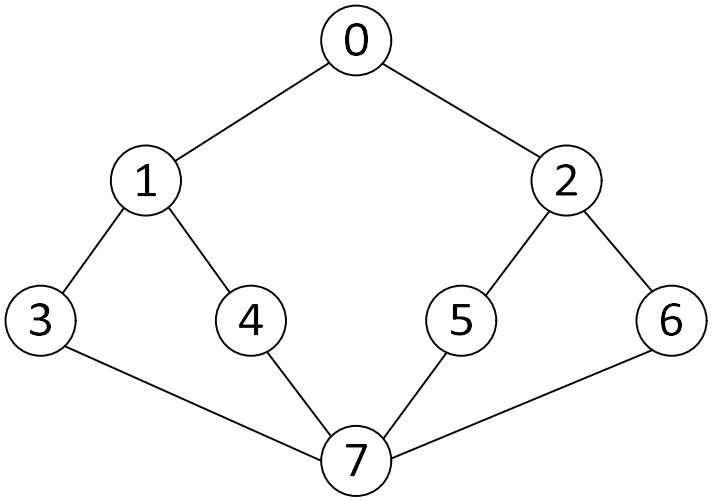
Depth First Search (DFS)

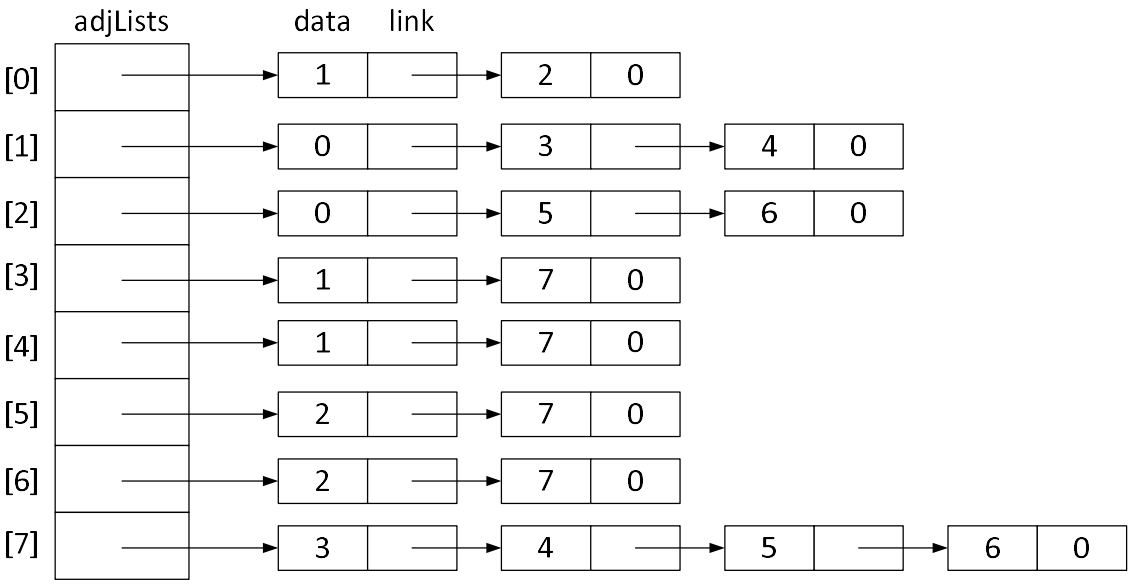
- Recursive implementation of DFS
  - Which representation of graph is function dfs using?
    - adjacency matrix or adjacency list?
- Try completing the rest of the program
  - Let graph[] be the example graph shown in the previous slides.
  - The program should print the vertex ID in the visiting order.

```
void dfs(int v) {
  /* depth first search of a graph beginning at v */
  nodePointer w;
  visited[v] = TRUE;  // visited[] is a global variable
  printf("%5d", v);
  for(w = graph[v]; w; w = w->link)
    if(!visited[w->vertex])
      dfs(w->vertex);
}
```

Depth First Search (DFS)

- Representation of the example graph





adjacency list

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

adjacency matrix

Depth First Search (DFS)

- Analysis of function DFS
  - Let us assume we have a connected graph with n vertices.
  - In DFS, for each vertex, we need to check and process each of its adjacent vertices.
  - If we use the adjacency list, we need to traverse the linked list of adjacent vertices to check whether the vertex is visited or not.
    - The total number of chain nodes in an adjacency list is 2e.
    - We need to visit all n vertices.
    - Time complexity: $O(n+e)$
  - If we use the adjacency matrix, we need to traverse the row in order to check for adjacent vertices.
    - We need to visit all n vertices, and for each vertex we need to traverse all rows in the adjacency matrix.
    - Time complexity: $O(n^2)$

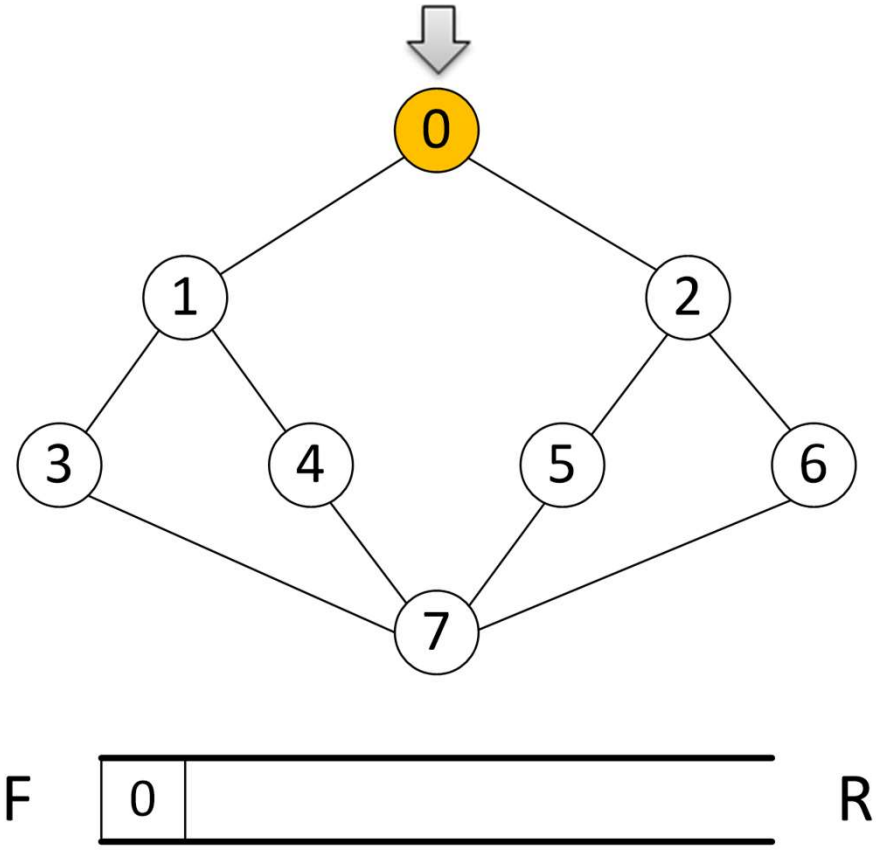
Breadth First Search (BFS)

- The procedure of function BFS
  - Initially, we are given a starting vertex v, where we begin our search.
  - First, we visit the starting vertex v.
  - Then, we visit each of the adjacent vertices of v.
  - Then, we visit all the unvisited vertices that are adjacent to the first adjacent vertex of v.
  - Then, we visit all the unvisited vertices that are adjacent to the second adjacent vertex of v.
  - Continue in this manner until we have visited all vertices.

- Implementation
  - As we visit each unvisited vertex we place the vertex in a queue.
  - Once we are finished with the adjacent vertices, we remove a vertex from the queue and visit its adjacent vertices.

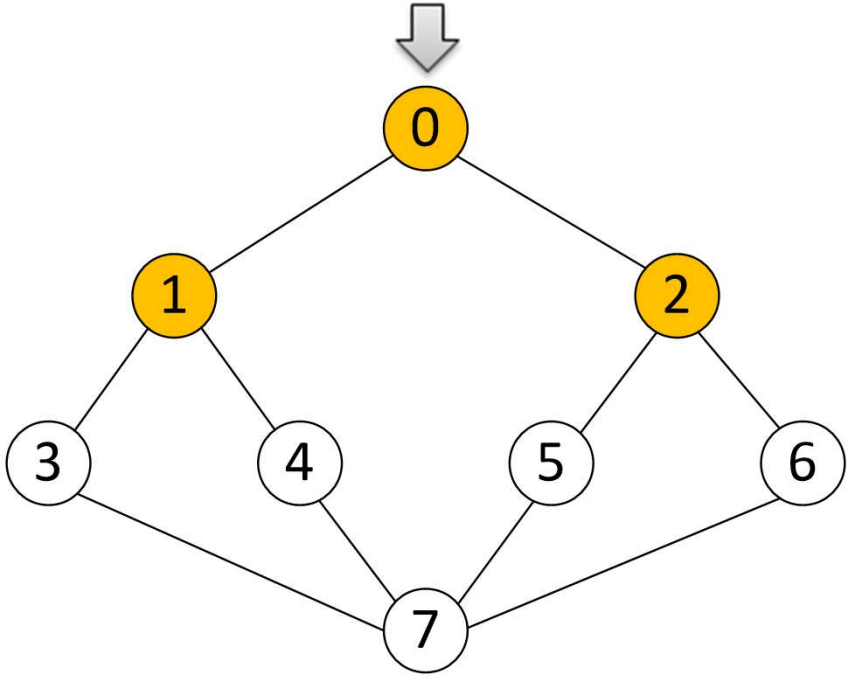
Breadth First Search (BFS)

- Suppose we would like to perform DFS on graph G, starting from vertex 0.
- We prepare a queue.
- We first visit vertex 0, and add vertex 0 to the queue.



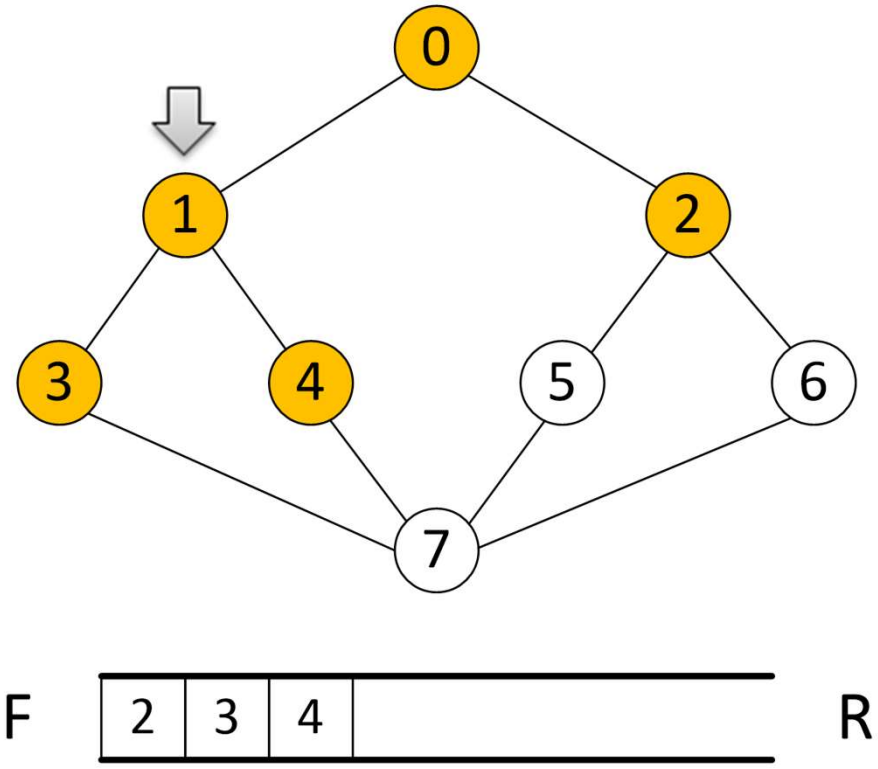
Breadth First Search (BFS)

- Then we go into a loop.
  - We remove a vertex (we will call this "v") from the queue.
  - For all unvisited adjacent vertices of v, we visit them and also add them to the queue.
  - Here, we visit vertices 1 and 2, and add them to the queue.



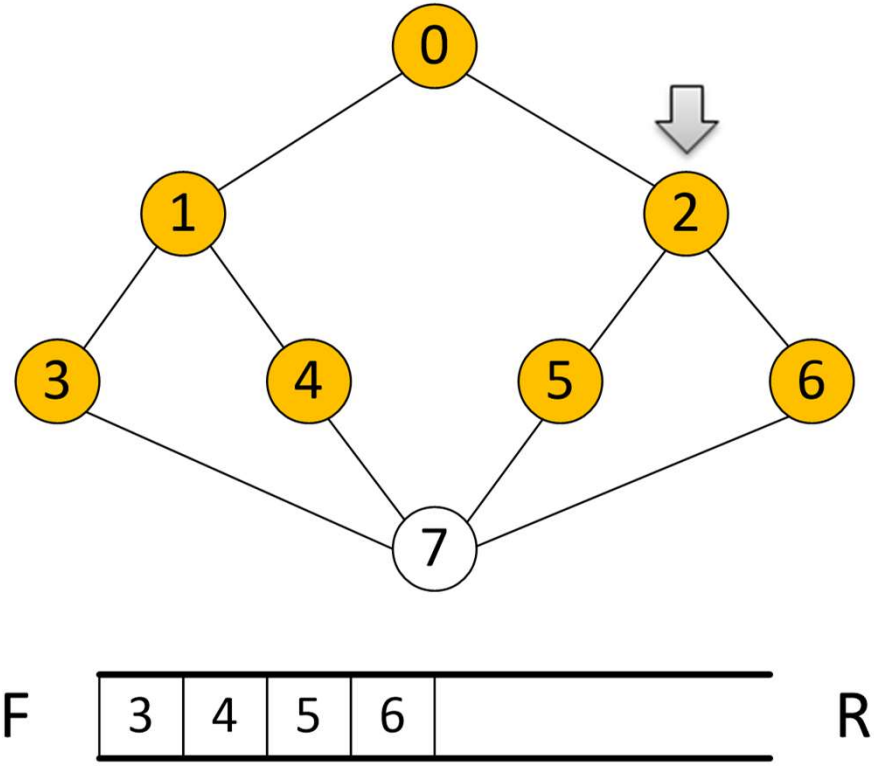
Breadth First Search (BFS)

- We repeat the loop.
  - Remove a vertex from the queue → vertex 1.
  - Visit all unvisited adjacent vertices of vertex 1, and add them to the queue.



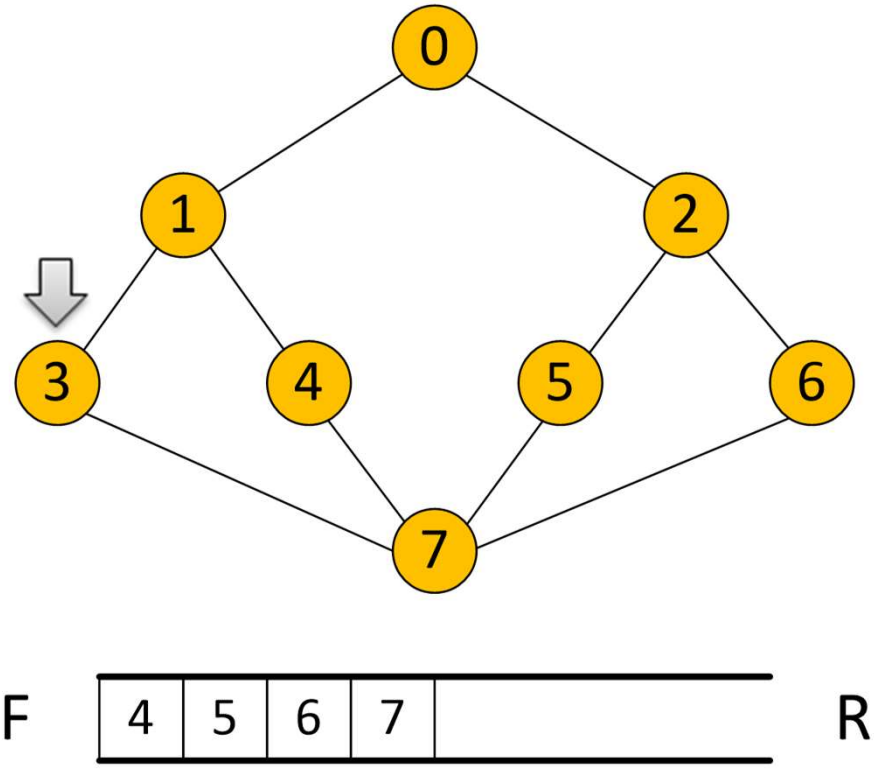
Breadth First Search (BFS)

- We repeat the loop.
  - Remove a vertex from the queue → vertex 2.
  - Visit all unvisited adjacent vertices of vertex 2, and add them to the queue.



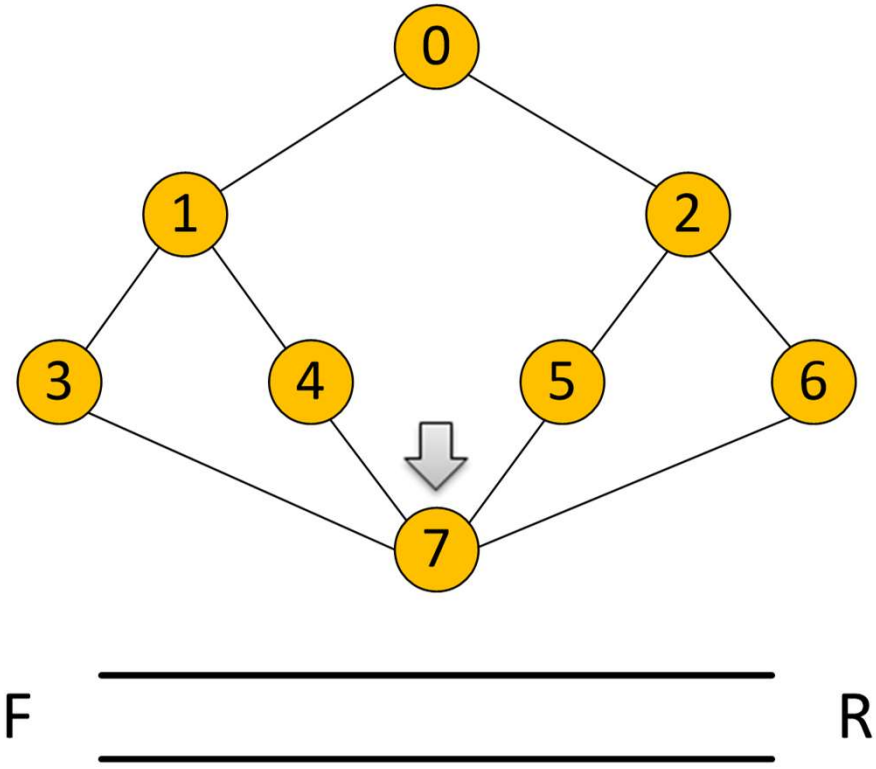
Breadth First Search (BFS)

- We repeat the loop.
  - Remove a vertex from the queue → vertex 3.
  - Visit all unvisited adjacent vertices of vertex 3, and add them to the queue.



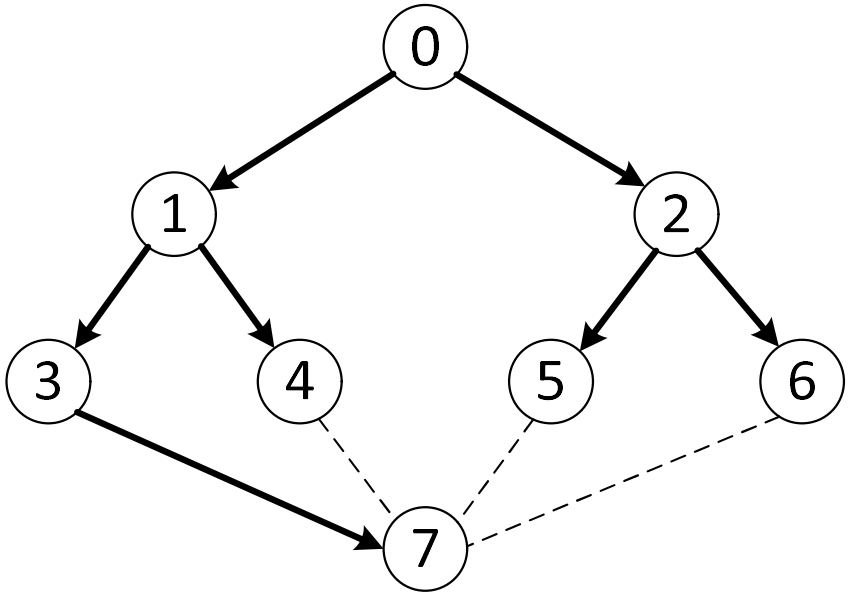
Breadth First Search (BFS)

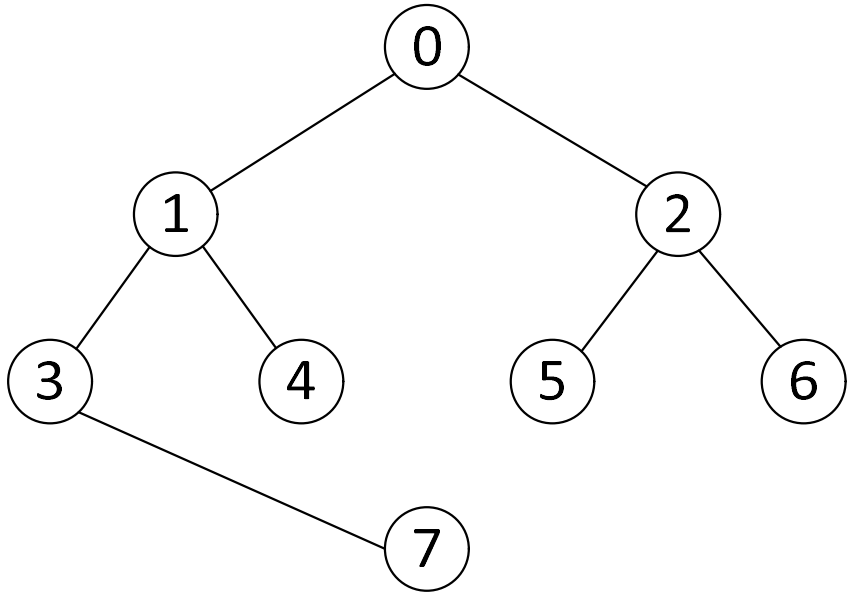
- We repeat the loop until the queue is empty.
- We are done with BFS.



Breadth First Search (BFS)

- In a breadth first search, we stay as close to the starting vertex while searching for vertices. The resulting tree is "broad". That is why it is called breadth first search.





Breadth First Search (BFS)

- Implementation of BFS using a queue.
- Try completing the rest of the program.
  - You will need to implement the queue and its operations.
  - The program should print the vertex ID in the visiting order.

```
void bfs(int v) {
  nodePointer w;
  front = rear = NULL;  /* initialize queue */
  printf("%5d", v);
  visited[v] = TRUE;
  addq(v);
  while(front) {
    v = deleteq();
    for(w = graph[v]; w; w = w->link)
      if(!visited[w->vertex]) {
        printf("%5d", w->vertex);
        addq(w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

Breadth First Search (BFS)

- Analysis of function BFS
  - Each vertex is placed on the queue exactly once, so the while loop is iterated at most n times.
  - For adjacency list representation, the loop has a total cost of $d_0 + ... + d_{n-1} = O(e)$, where $d_i = \text{degree}(v_i)$. Thus, the time complexity is $O(n+e)$.
  - For adjacency matrix representation, the while loop takes O(n) time for each vertex visited. Thus, the time complexity is $O(n^2)$.

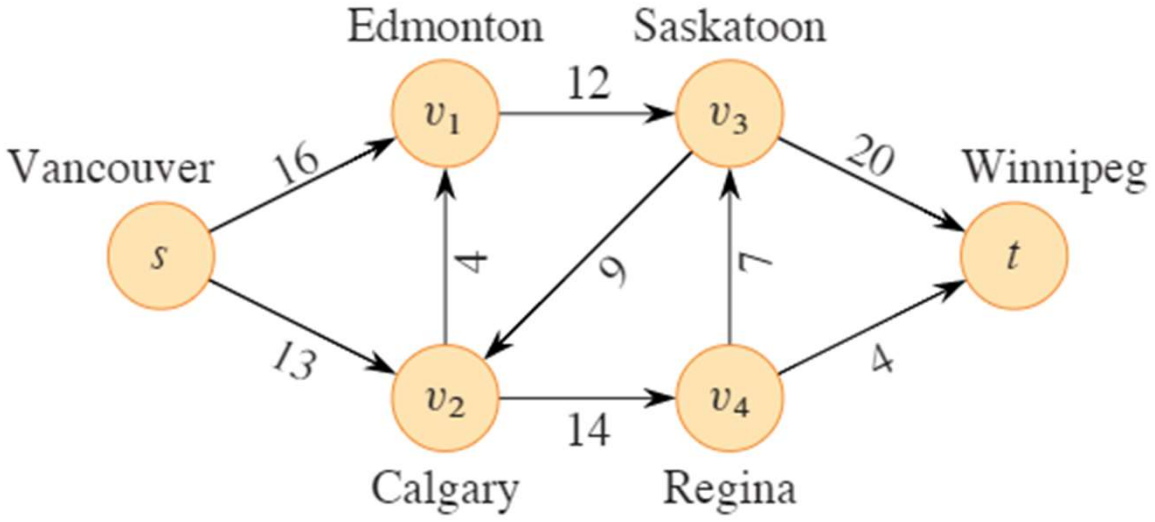
24.1 Flow Networks

Flow Networks and Flows

- A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$.
- We further require that if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction.
- If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$.
- We disallow self-loops.
- A flow network contains two distinguished vertices: a source s and a sink t.
- We assume that each vertex lies on some path from the source to the sink.
  - For each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$.
- Because each vertex other than s has at least one entering edge, we have $|E| \geq |V| - 1$.

Flow Network: Example

- A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem.
- The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t.
- The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity.



Flow

- Let $G = (V, E)$ be a flow network with a capacity function c.
- Let s be the source of the network, and let t be the sink.
- A flow in G is a real-valued function $f: V \times V \to \mathbb{R}$ that specifies the following two properties:
- Capacity constraint
  - For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.
  - The flow from one vertex to another must be nonnegative and must not exceed the given capacity.
- Flow conservation
  - For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

  - The total flow into a vertex other than the source or sink must equal the total flow out of that vertex ("flow in equals flow out")

Flow

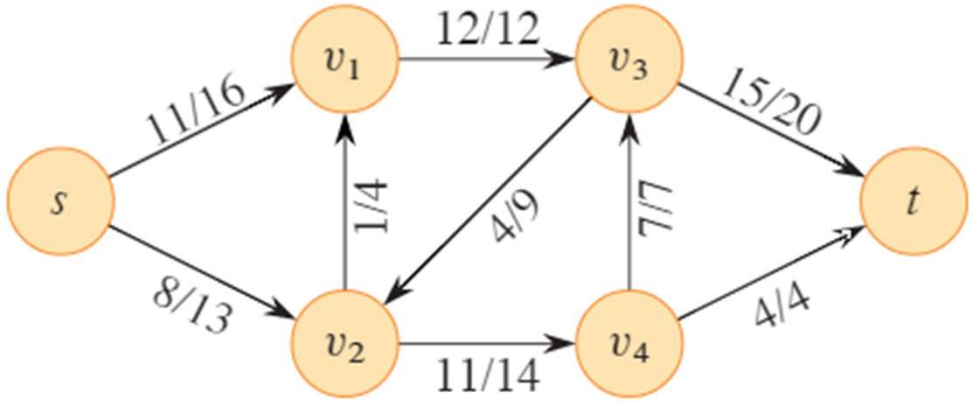
- When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.
- We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v.
- The value $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

- that is, the total flow out of the source minus the flow into the source.
- Typically, a flow network does not have any edges into the source, and thus $\sum_{v \in V} f(v, s)$ is 0.
  - For "residual networks", the flow into the source can be positive.
- In the maximum-flow problem, the input is a flow network G with source s and sink t, and the goal is to find a flow of maximum value.

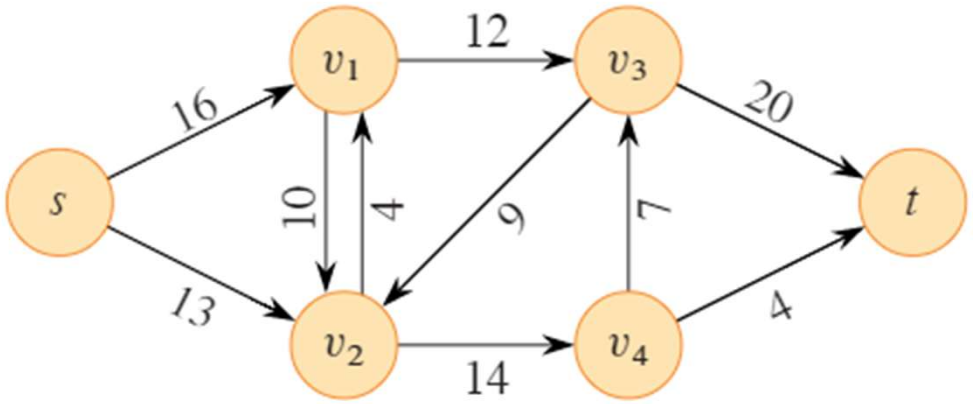
Flow: Example

- For the flow network G modeling the Lucky Puck company, the figure at the bottom shows an example flow f in G with value $|f| = 19$.
- Each edge is labeled by $f(u, v)/c(u, v)$.
  - The slash notation does not indicate division.



Modeling Problems with Antiparallel Edges

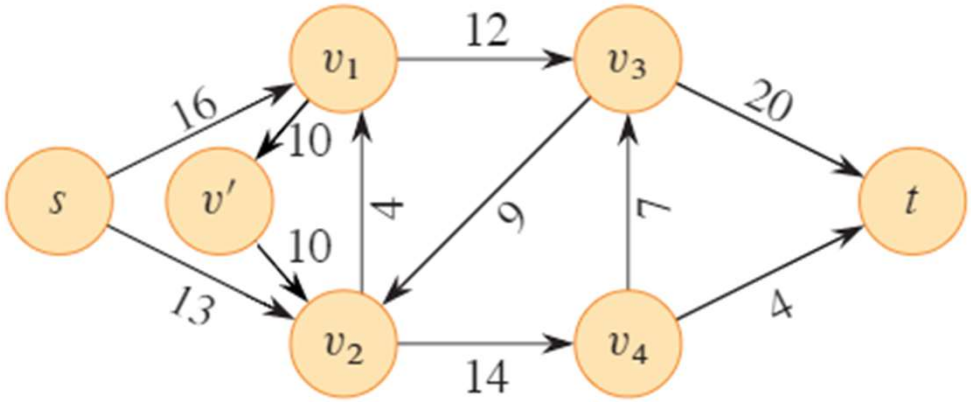
- Suppose the Lucky Puck company newly leases spaces for 10 crates in trucks going from Edmonton to Calgary. Then, the flow network will become:



- This network violates the condition: if $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$.
  - We call the two edges antiparallel.

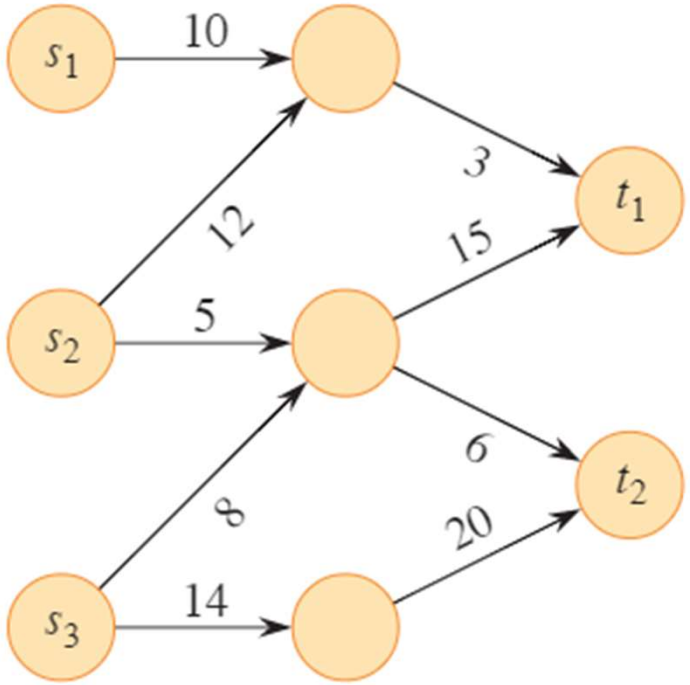
Modeling Problems with Antiparallel Edges

- In this case, we transform the network into an equivalent one containing no antiparallel edges.
- To do that, we introduce a new vertex between v_1 and v_2.



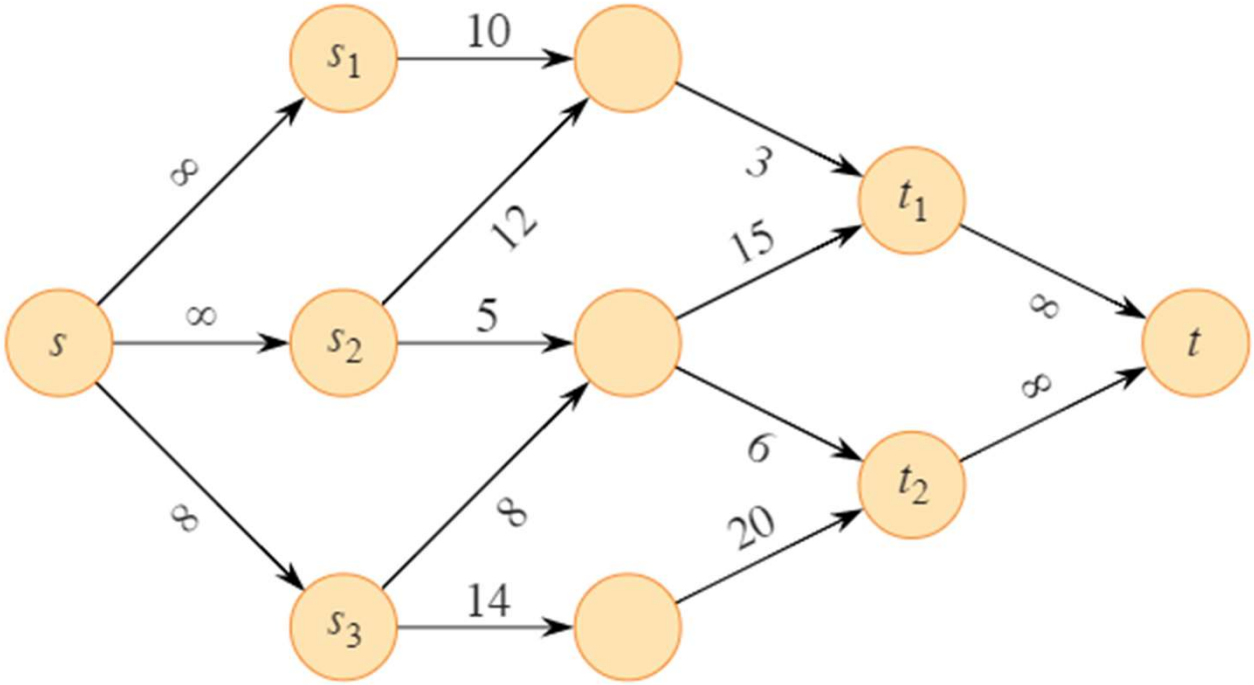
Networks with Multiple Sources and Sinks

- A maximum-flow problem may have several sources and sinks.
- For example, the Lucky Puck company might have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$.



Networks with Multiple Sources and Sinks

- We can convert this network into an ordinary flow network by adding a supersource and a supersink.



24.2 The Ford-Fulkerson Method

Ford-Fulkerson Method

- A method for solving the maximum-flow problem

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

- The method iteratively increases the value of the flow.
- It starts with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0.
- Each iteration increases the flow value in G by finding an "augmenting path" in an associated "residual network".
- The method repeatedly augments the flow until the residual network has no augmenting paths.

Residual Networks

- Intuitively, given a flow network G and a flow f, the residual network G_f consists of edges whose capacities represent how the flow can change on edges of G.

- An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge.

- If that value is positive, that edge goes into G_f with a "residual capacity" of $c_f(u, v) = c(u, v) - f(u, v)$.

- The only edges of G that belong to G_f are those that can admit more flow.
  - Edges (u, v) whose flow equals their capacity have $c_f(u, v) = 0$, and they do not belong to G_f.

Residual Networks: Reverse Edges

- The residual network G_f can also contain edges that are not in G.

- As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge in order to increase the flow elsewhere.

- In order to represent a possible decrease in the positive flow $f(u, v)$ on an edge in G, the residual network G_f contains an edge (v, u) with residual capacity $c_f(u, v) = f(u, v)$ – that is, an edge that can admit flow in the opposite direction to (u, v), at most canceling out the flow on (u, v).

- These reverse edges allow an algorithm to send back flow it has already sent along the edge. Sending flow back along an edge is equivalent to decreasing the flow on the edge.

Residual Network: Formal Definition

- For a flow network $G = (V, E)$ with source s, sink t, and a flow f, consider a pair of vertices $u, v \in V$.
- We define the residual capacity $c_f(u, v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

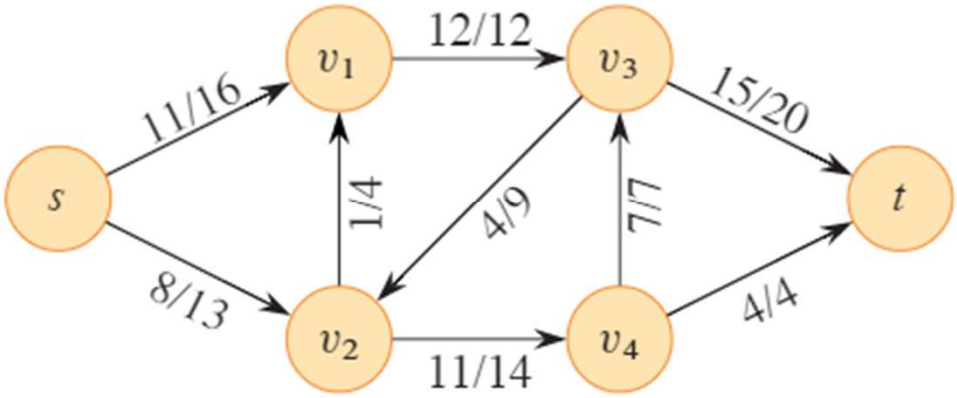
- Since $(u, v) \in E$ implies $(v, u) \notin E$, exactly one case in the equation applies to each ordered pair of vertices.

- Example
  - Suppose $c(u, v) = 16$ and $f(u, v) = 11$
  - $c_f(u, v) = 5$
  - $c_f(v, u) = 11$

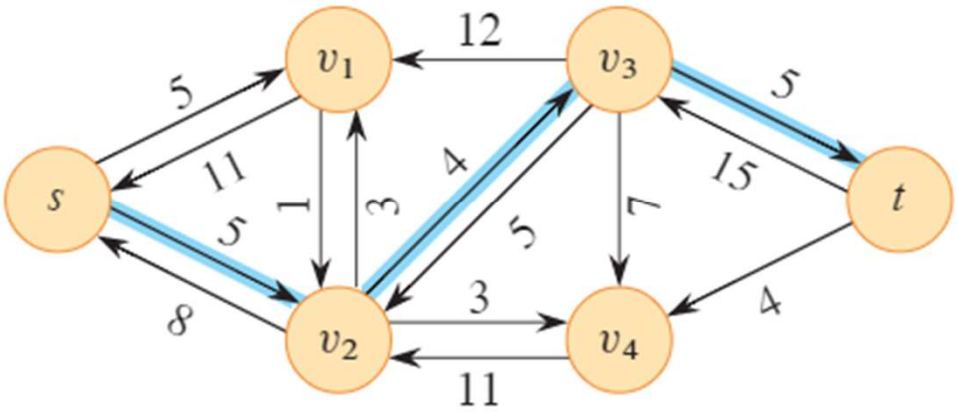
Residual Network: Formal Definition

- Given a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is $G_f = (V, E_f)$, where
- $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

- Each edge of the residual network, or residual edge, can admit a flow that is greater than 0.
- The edges in E_f are either edges in E or their reversals, thus $|E_f| \leq 2|E|$.



flow network G and flow f



residual network G_f

Residual Networks: Properties

- The residual network G_f is similar to a flow network, but it is not, because it could contain antiparallel edges.

- Other than this difference, a residual network has the same properties as a flow network.

- We can define a flow in the residual network as one that satisfies the definition of a flow but with respect to capacities c_f in the residual network G_f.

- A flow in a residual network provides a roadmap for adding a flow in the original flow network.

Augmentation and Cancellation

- If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the augmentation of flow f by f', to be a function from $V \times V$ to $\mathbb{R}$, defined by:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

  - Equation (24.4)

- The flow on (u,v) increases by $f'(u,v)$, but decreases by $f'(v,u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network.

- Pushing flow on the reverse edge in the residual network is also known as cancellation.

Lemma 24.1: Augmenting flow yields greater flow value

- Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with $|f \uparrow f'| = |f| + |f'|$.

- Proof
  - We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex $V - \{s, t\}$.
  - If $(u, v) \in E$, $c_f(v, u) = f(u, v)$.
  - Because f' is a flow in G_f, we have $f'(v, u) \leq c_f(v, u)$.
  - Thus, $f'(v, u) \leq f(u, v)$. Therefore,

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (24.4))} \\
&\geq f(u, v) + f'(u, v) - f(u, v) && \text{(because } f'(v, u) \leq f(u, v)\text{)} \\
&= f'(u, v) \\
&\geq 0 .
\end{aligned}$$

Lemma 24.1: Augmenting flow yields greater flow value

- In addition,

$$
\begin{aligned}
&(f \uparrow f')(u,v) \\
&\qquad = f(u,v) + f'(u,v) - f'(v,u) && \text{(by equation (24.4))} \\
&\qquad \le f(u,v) + f'(u,v) && \text{(because flows are nonnegative)} \\
&\qquad \le f(u,v) + c_f(u,v) && \text{(capacity constraint)} \\
&\qquad = f(u,v) + c(u,v) - f(u,v) && \text{(definition of } c_f\text{)} \\
&\qquad = c(u,v) \ .
\end{aligned}
$$

Lemma 24.1: Augmenting flow yields greater flow value

- To show that flow conservation holds and that $|f \uparrow f'| = |f| + |f'|$, we first prove the claim that for all $u \in V$, we have

$$\sum_{v \in V} (f \uparrow f')(u, v) - \sum_{v \in V} (f \uparrow f')(v, u)$$
$$= \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u)$$

  - Equation (24.5)

- Because we disallow antiparallel edges in G, for each vertex u, there can be an edge (u, v) or (v, u) in G, but not both.

- For a fixed vertex u, define $V_l(u) = \{v : (u, v) \in E\}$ to be the set of vertices with edges in G leaving u, and define $V_e(u) = \{v : (v, u) \in E\}$ to be the set of vertices with edges in G entering u.

Lemma 24.1: Augmenting flow yields greater flow value

- By the definition of flow augmentation, only vertices v in $V_l(u)$ can have positive $(f \uparrow f')(u,v)$, and only vertices v in $V_e(u)$ can have positive $(f \uparrow f')(v,u)$. We use this fact and modify the equation.

$$
\begin{aligned}
&\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f \uparrow f')(u,v) - \sum_{v \in V_e(u)} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f(u,v) + f'(u,v) - f'(v,u)) - \sum_{v \in V_e(u)} (f(v,u) + f'(v,u) - f'(u,v)) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) + \sum_{v \in V_l(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) \\
&\qquad\qquad - \sum_{v \in V_e(u)} f(v,u) - \sum_{v \in V_e(u)} f'(v,u) + \sum_{v \in V_e(u)} f'(u,v) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) \\
&\qquad\qquad + \sum_{v \in V_l(u)} f'(u,v) + \sum_{v \in V_e(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) - \sum_{v \in V_e(u)} f'(v,u) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) + \sum_{v \in V_l(u) \cup V_e(u)} f'(u,v) - \sum_{v \in V_l(u) \cup V_e(u)} f'(v,u) \,. \qquad (24.6)
\end{aligned}
$$

Lemma 24.1: Augmenting flow yields greater flow value

- From Equation (24.6), all four summations can extend to sum over V, since each additional term has value 0.

- Taking all four summations over V proves the claim in Equation (24.5).

Lemma 24.1: Augmenting flow yields greater flow value

- Now we prove flow conservation for $f \uparrow f'$ and that $|f \uparrow f'| = |f| + |f'|$.

- To prove $|f \uparrow f'| = |f| + |f'|$, let $u = s$ in equation (24.5). Then, we have

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s, v) - \sum_{v \in V} (f \uparrow f')(v, s) \\ &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \\ &= |f| + |f'| \ . \end{aligned}$$

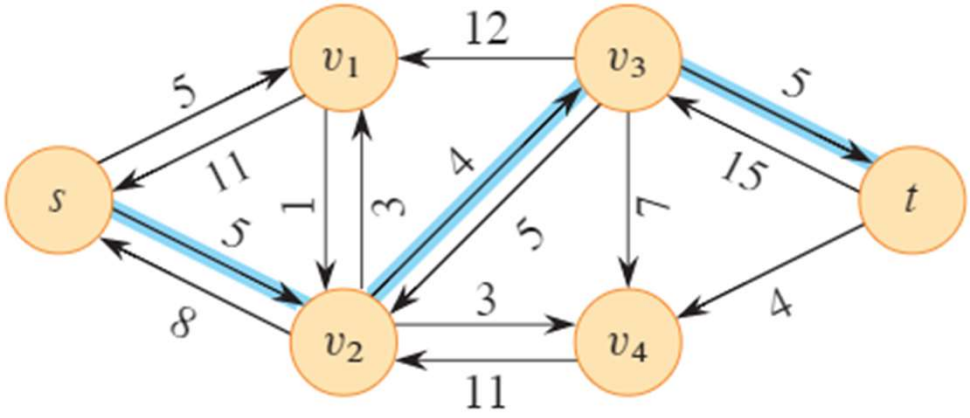
- For flow conservation, for any vertex u that is neither s not t, flow conservation holds for f and f', which means that the right-hand side of equation (24.5) is 0. Therefore,

$$\sum_{v \in V} (f \uparrow f')(u, v) = \sum_{v \in V} (f \uparrow f')(v, u)$$

Augmenting Paths

- Given a flow network $G = (V, E)$ and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f.

- By the definition of the residual network, the flow on an edge (u, v) of an augmenting path may increase up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) belongs to the original flow network G.

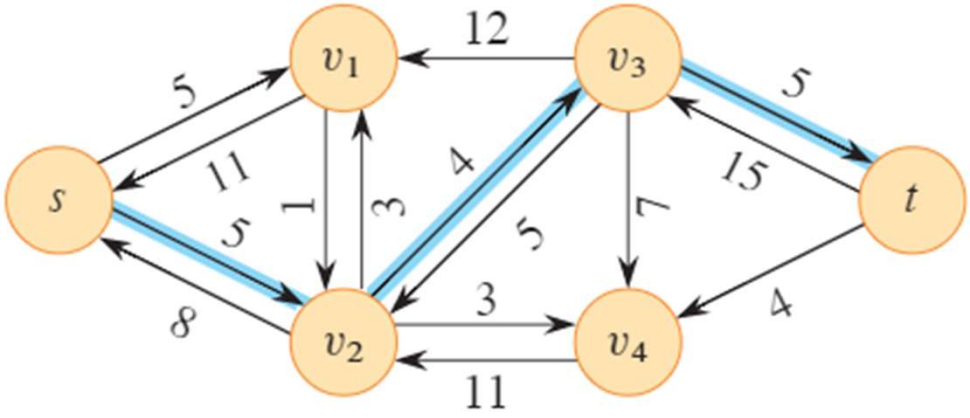


an augmenting path (blue) on the residual network G_f

Augmenting Paths

- The flow through each edge of this path can increase up to 4 units without violating a capacity constraint.
  - The smallest residual capacity on this path is $c_f(v_2, v_3) = 4$.

- We call this maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by
  - $c_f(p) = \min\{c_f(u, v): (u, v) \text{ is in } p\}$



an augmenting path (blue) on the residual network G_f

Lemma 24.2

- Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

  - Equation (24.7)

- Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Proof of Lemma 24.2

- We must show that f_p obeys the capacity constraint and flow conservation.
- Capacity constraint
  - $f_p(u, v) = 0$ if edge (u, v) is not in the augmenting path p.
  - If (u, v) is in path p, then $f_p(u, v) = c_f(p) \leq c_f(u, v)$.
- Flow conservation
  - If vertex $u \in V - \{s, t\}$ is not on path p, then for all $v \in V$, we have $f_p(u, v) = f_p(v, u) = 0$.
  - If u is on path p, let x and y be u's predecessor and successor, respectively, in p, so that $f_p(x, u) = f_p(u, y) = c_f(p)$. Then, we have

$$\begin{aligned}
&\sum_{v \in V} f_p(v, u) - \sum_{v \in V} f_p(u, v) \\
&\quad = \left(\sum_{v \in V - \{x\}} f_p(v, u) + f_p(x, u) \right) - \left(\sum_{v \in V - \{y\}} f_p(u, v) + f_p(u, y) \right) \\
&\quad = (0 + f_p(x, u)) - (0 + f_p(u, y)) \\
&\quad = c_f(p) - c_f(p) \\
&\quad = 0 ,
\end{aligned}$$

  - Thus, f_p obeys flow conservation.

Corollary 24.3

- Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f.
- Let f_p be defined as in equation (24.7), and suppose that f is augmented by f_p.
- Then, the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.
- Proof: immediate from lemma 24.1 and 24.2.

Flow after Augmenting

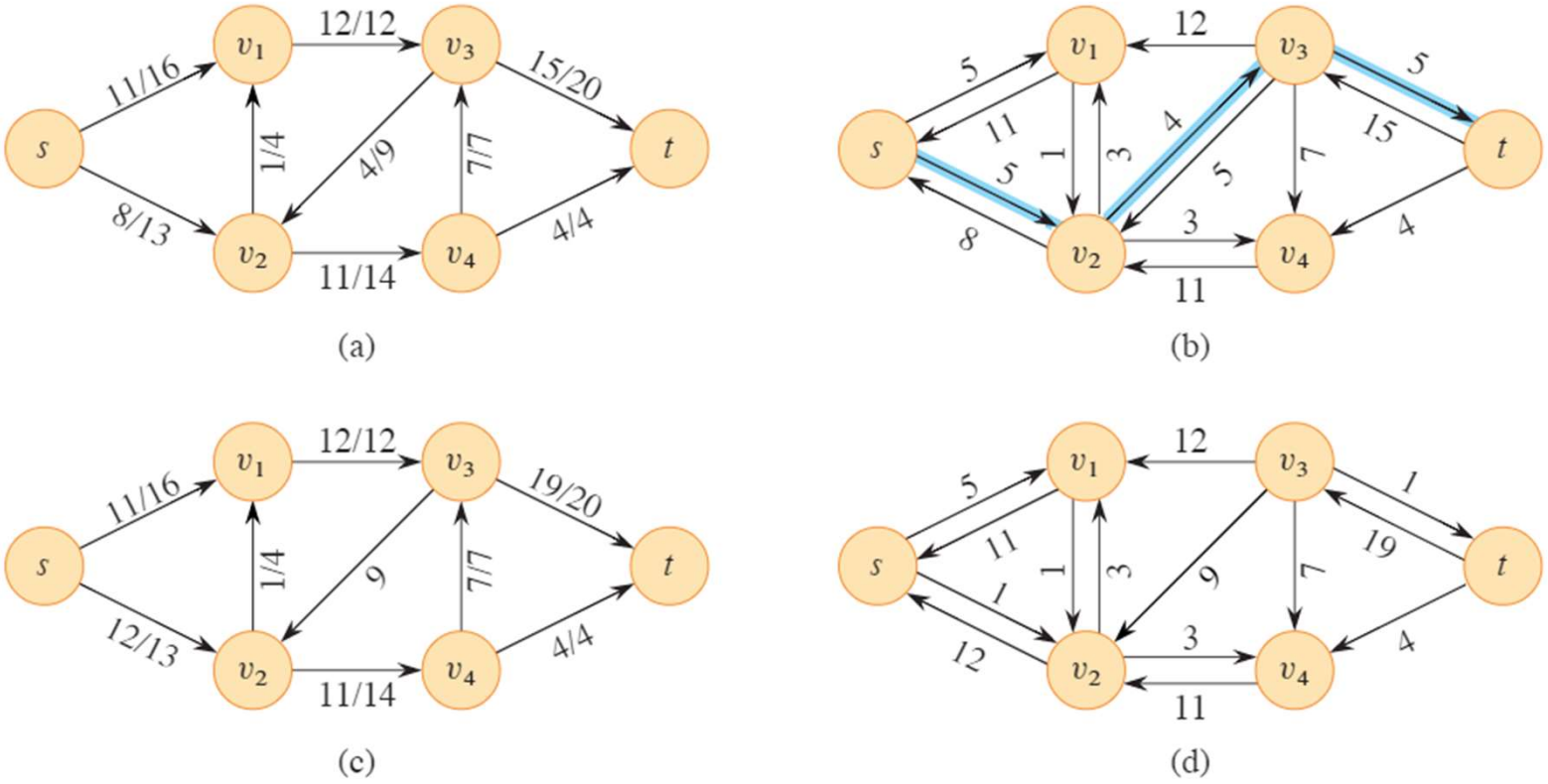


Figure 24.4 (**a**) The flow network G and flow f of Figure 24.1(b). (**b**) The residual network G_f with augmenting path p, having residual capacity $c_f(p) = c_f(v_2, v_3) = 4$, in blue. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. (**c**) The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. (**d**) The residual network induced by the flow in (c).

Cuts of Flow Networks

- The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow.

- How do we know that when the algorithm terminates, it has actually found a maximum flow?

- The max-flow min-cut theorem tells us that a flow is maximum if and only if its residual network contains no augmenting path.

Cut

- A cut (S, T) of a flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

- If f is a flow, then the net flow $f(S, T)$ across the cut (S, T) is defined as:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) .$$

- The capacity of the cut (S, T) is:

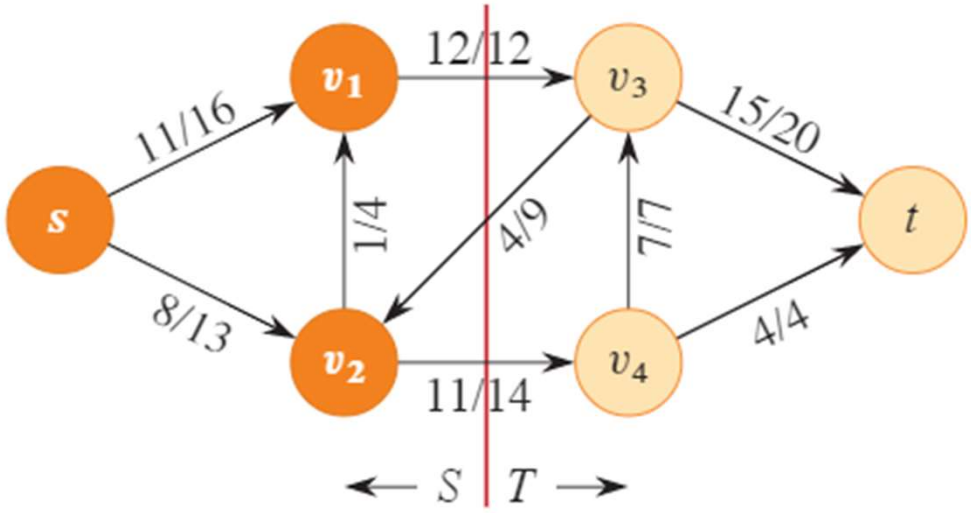
$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

- A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Cut: Example

- The net flow across this cut is $f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) = 12 + 11 - 4 = 19$.

- The capacity of this cut is $c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26$.



Lemma 24.4: Net flow across any cut is the same

- Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then, the net flow across (S, T) is $f(S, T) = |f|$.

- Proof:
- For any vertex $u \in V - \{s, t\}$, rewrite the flow conservation condition as

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(u, v) = 0$$

  - Equation (24.10)
- Taking the definition of $|f|$ and adding the left-hand side of equation (24.10), which equals 0, summed over all vertices in $S - \{s\}$, gives

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) .$$

Lemma 24.4: Net flow across any cut is the same

- Expanding the right-hand summation and regrouping terms yields:

$$
\begin{aligned}
|f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S-\{s\}} \sum_{v \in V} f(u,v) - \sum_{u \in S-\{s\}} \sum_{v \in V} f(v,u) \\
&= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S-\{s\}} f(u,v) \right) - \sum_{v \in V} \left(f(v,s) + \sum_{u \in S-\{s\}} f(v,u) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u) \,.
\end{aligned}
$$

Lemma 24.4: Net flow across any cut is the same

- Because $V = S \cup T$ and $S \cap T = \emptyset$, splitting each summation over V into summations over S and T gives

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&\quad + \left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right) .
\end{aligned}$$

- The two summations within the parentheses are actually the same, since for all vertices $x, y \in S$, the term $f(x,y)$ appears once in each summation. Thus,

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= f(S,T) .
\end{aligned}$$

Corollary 24.5

- The value of any flow f in a flow network G is bounded from above by the capacity of any cut in G.
- Proof: Let (S, T) be any cut of G and let f be any flow. By Lemma 24.4 and the capacity constraint,

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &\le \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \,. \end{aligned}$$

- The value of a maximum flow in a network is bounded from above by the capacity of a minimum cut of the network.

Theorem 24.6 (Max-flow Min-cut Theorem)

- If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network $\mathrm{G_f}$ contains no augmenting paths.
3. $|\mathrm{f}| = \mathrm{c}(\mathrm{S}, \mathrm{T})$ for some cut (S, T) of G.

- The value of a maximum flow is equal to the capacity of a minimum cut.

- Proof (1)→(2)
- Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by corollary 24.3, the flow found by augmenting f by f_p is a flow in G with value strictly greater than f, contradicting the assumption that f is a maximum flow.

Theorem 24.6 (Max-flow Min-cut Theorem)

- Proof (2)→(3)
- Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t.
- Define $S = (v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$ and $T = V - S$.
- The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f.
- Now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S.
- If neither (u, v) nor (v, u) belongs to E, then $f(u, v) = f(v, u) = 0$. Thus,

$$\begin{aligned} f(S,T) &= \sum_{u \in S}\sum_{v \in T} f(u,v) - \sum_{v \in T}\sum_{u \in S} f(v,u) \\ &= \sum_{u \in S}\sum_{v \in T} c(u,v) - \sum_{v \in T}\sum_{u \in S} 0 \\ &= c(S,T)\,. \end{aligned}$$

- By Lemma 24.4, $|f| = f(S, T) = c(S, T)$.

Theorem 24.6 (Max-flow Min-cut Theorem)

- Proof (3)→(1)
- By corollary 24.5, $|f| \leq c(S,T)$ for all cuts (S,T).
- The condition $|f| = c(S,T)$ thus implies that f is a maximum flow.

The basic Ford-Fulkerson algorithm

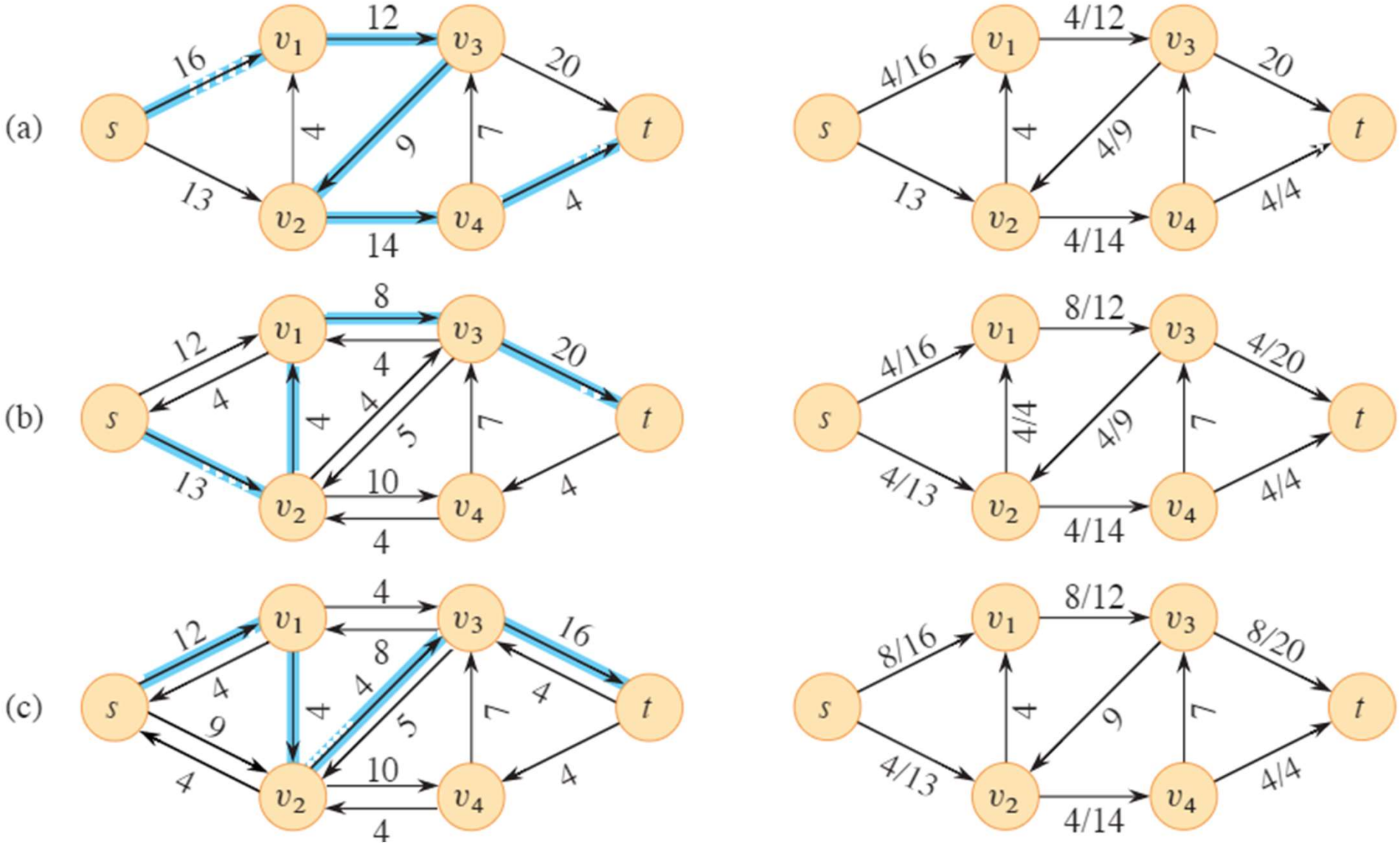
- Each iteration of the Ford-Fulkerson method finds some augmenting path p and uses p to modify the flow f.
- Replacing f by $f \uparrow f_p$ produces a new flow whose value is $|f| + |f_p|$.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ G.E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
9  return f
```

The basic Ford-Fulkerson algorithm

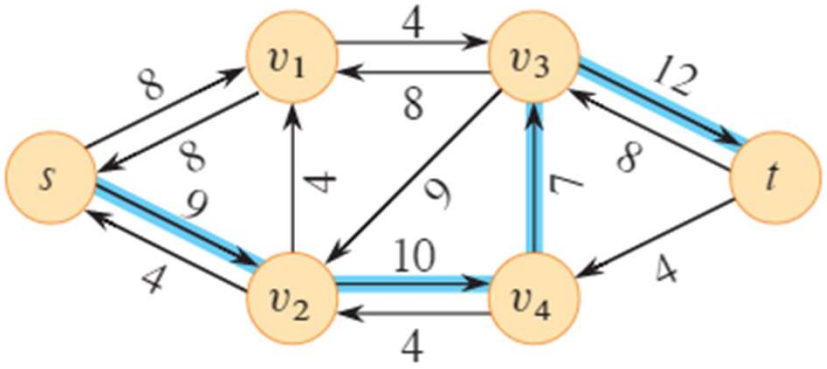
- Line 1-2 initialize the flow f to 0.
- The while loop of lines 3-8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$.
- Each residual edge in path p is either an edge in the original network or the reversal of an edge in the original network.
- Lines 6-8 update the flow in each case appropriately, adding flow when the residual edge is an original edge and subtracting it otherwise.
- When no augmenting paths exist, the flow f is a maximum flow.

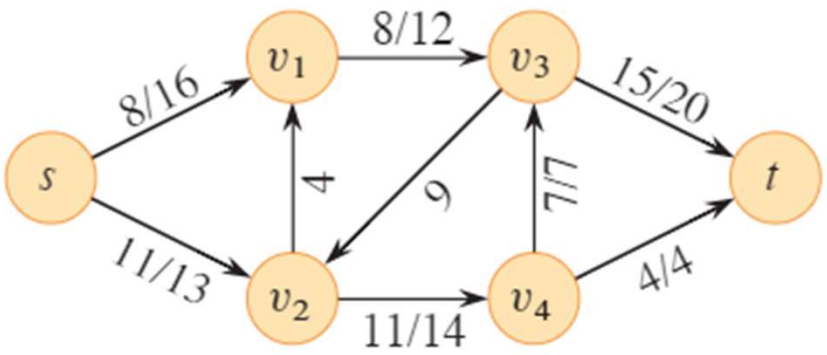
The basic Ford-Fulkerson algorithm: Example



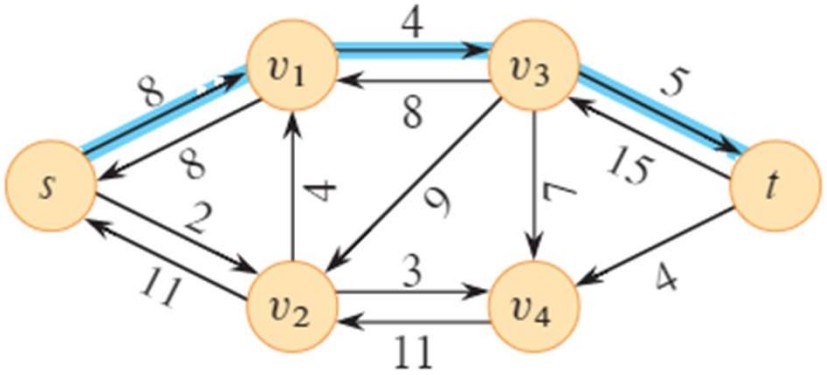
The basic Ford-Fulkerson algorithm: Example

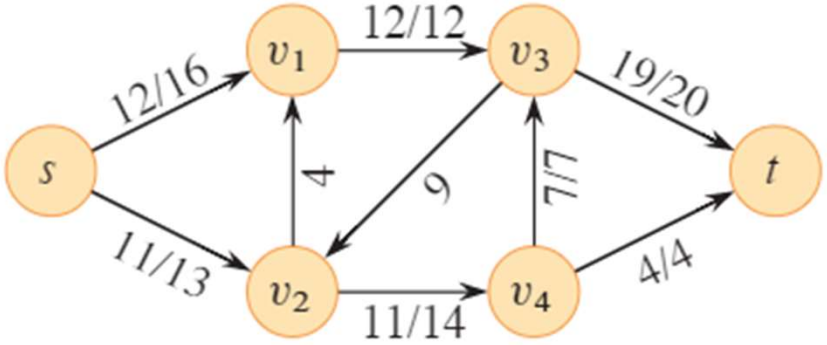
(d)



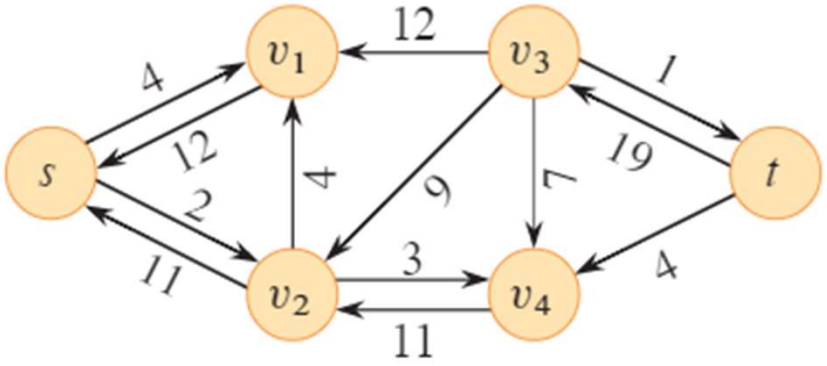


(e)





(f)



Analysis of Ford-Fulkerson Method

- The running time of FORD-FULKERSON depends on the augmenting path p and how it is found in line 3.

- First, we can find a path from s to t using graph search algorithms such as DFS (Depth-First Search) and BFS (Breadth-First Search) which run in $O(E)$ time.
  - E is the number of edges in the graph.
  - Actually, the running time of DFS and BFS is $O(V + E')$, when using adjacent list to represent a graph. V is the number of vertices in the flow network, and E' is the number of edges in the residual network.
  - As mentioned earlier, a flow network has the property $|E| \geq |V| - 1$.
  - Also, $|E| \leq |E'| \leq 2|E|$.
  - Thus, $O(V + E') = O(E)$.

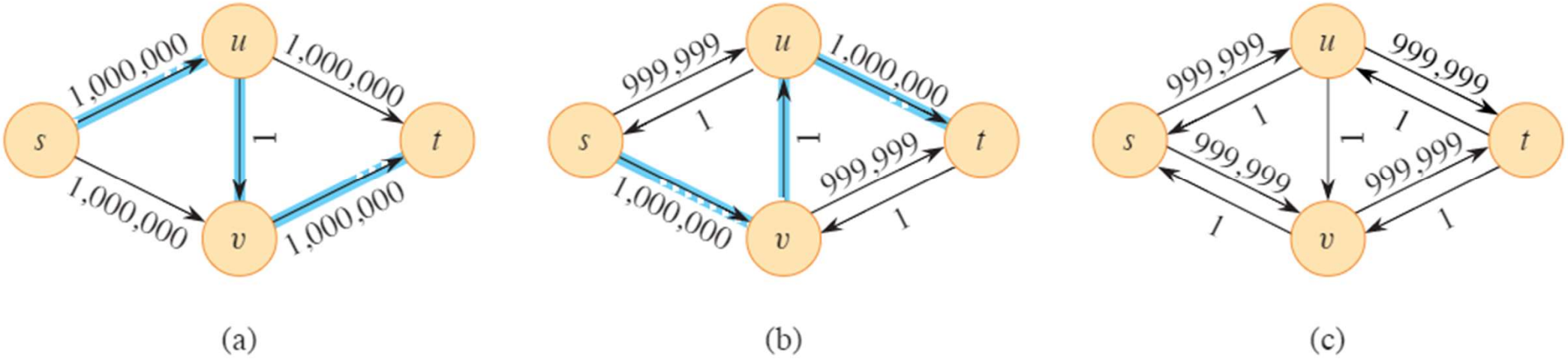
Analysis of Ford-Fulkerson Method

- Suppose the capacity of edges are integers. Then, the lines 3-8 of FORD-FULKERSON can run at most $|f^*|$ times, where $|f^*|$ is the maximum flow in the network.
  - The flow value increases by at least 1 unit in each iteration.
  - If the capacities are rational numbers, then they can be multiplied by a proper scaling factor to become integers.

- Thus, the total running time of FORD-FULKERSON is $O(E|f^*|)$.

An example where $|f^*|$ is large

- A maximum flow in this network has value 2,000,000
  - 1,000,000 units flow through $s \to u \to t$, and 1,000,000 units through $s \to v \to t$.
  - If the first augmenting path found by FORD-FULKERSON is $s \to u \to v \to t$, the augmented flow has value 1.
  - If the second iteration finds the augmenting path $s \to v \to u \to t$, then the flow will have value 2.
  - If the augmenting paths are found in this pattern, we need a total of 2,000,000 augmentation.



The Edmonds-Karp algorithm

- In the previous example, the algorithm does not choose an augmenting path with the fewest edges.
- By using breadth-first search (BFS) to find an augmenting path in the residual network, the algorithm runs in polynomial time, independent of the maximum flow value.
- We call this implementation of Ford-Fulkerson method the Edmonds-Karp algorithm.
- We can prove that the Edmonds-Karp algorithm runs in $O(VE^2)$ time.
- Let us denote $\delta_f(u, v)$ the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 24.7

- If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

- We will suppose that a flow augmentation occurs that causes the shortest-path distance from s to some vertex $v \in V - \{s, t\}$ to decrease and then derive a contradiction.
- Let f be the flow just before an augmentation that decreases some shortest-path distance, and let f' be the flow just afterward.
- Let v be a vertex with the minimum $\delta_{f'}(s, v) < \delta_f(s, v)$.
- Let $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.
- Because of how we chose v, we know that the distance of vertex u from the source s did not decrease, that is, $\delta_{f'}(s, u) \geq \delta_f(s, u)$.

Lemma 24.7

- We claim that $(u, v) \notin E_f$. If we have $(u, v) \in E_f$, then we also have
- $\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v)$ which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.
  - $\delta_f(s, v) \leq \delta_f(s, u) + 1$, because $(u, v) \in E_f$

- So, now we have $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$.
- This means that the augmentation must have increased the flow from v to u, so that edge (v, u) was in the augmenting path.
- The augmenting path was a shortest path from s to t in G_f, and since any subpath of a shortest path is itself a shortest path, this augmenting path includes a shortest path from s to u in G_f that has (v, u) as its last edge.
- Therefore, $\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$, contradicting our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.
- We conclude that our assumption that such a vertex v exists is incorrect.

Theorem 24.8

- If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

- We say that an edge (u, v) in a residual network G_f is critical on an augmenting path p if the residual capacity of p is the residual capacity of (u, v), that is, if $c_f(p) = c_f(u, v)$.
- After flow is augmented along an augmenting path, any critical edge on the path disappears from the residual network.
- Moreover, at least one edge on any augmenting path must be critical.

Theorem 24.8

- We'll show that each of the $|E|$ edges can become critical at most $|V|/2$ times.

- Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have $\delta_f(s, v) = \delta_f(s, u) + 1$.
- Once the flow is augmented, the edge (u, v) disappears from the residual network.
- It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs if (v, u) appears on an augmenting path.
- If f' is the flow in G when this event occurs, then $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$.
- Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$ by Lemma 24.7, we have
- $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$.

Theorem 24.8

- Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2.
- The distance of u from the source is initially at least 0.
- Because edge (u, v) is on an augmenting path, and augmenting path ends at t, we know that u cannot be t, so that in any residual network that has a path from s to u, the shortest such path has at most $|V| - 2$ edges.
- Thus, after the first time (u, v) becomes critical, it can become critical at most $\frac{|V|-2}{2} = \frac{|V|}{2} - 1$ times more, for a total of $\frac{|V|}{2}$ times.
- Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$.

Running Time of Edmonds-Karp Algorithm

- Because each iteration of FORD-FULKERSON takes $O(E)$ time when it uses breadth-first search to find the augmenting path, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$.

24.3 Maximum Bipartite Matching

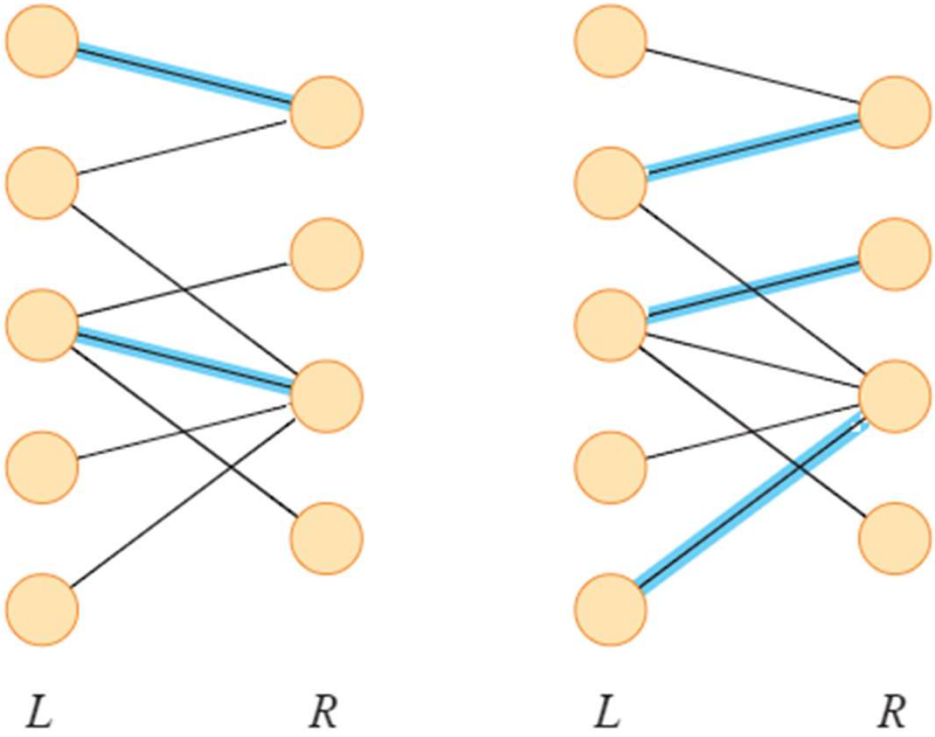
Maximum Bipartite Matching

- Given an undirected graph $G = (V, E)$, a matching is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.
- We say that a vertex $v \in V$ is matched by the matching M if some edge in M is incident on v, and otherwise, v is unmatched.
- A maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$.

- We consider finding maximum matching in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R.
- We further assume that every vertex in V has at least one incident edge.

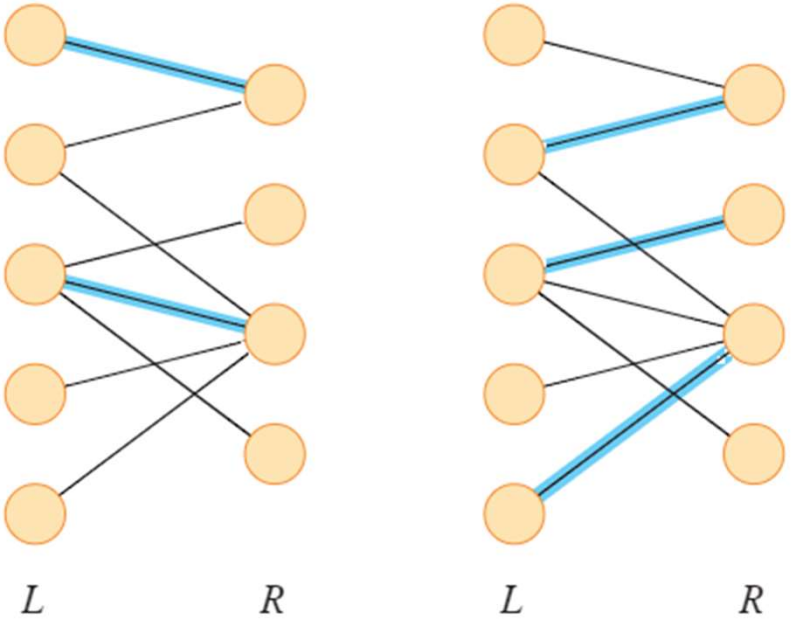
Maximum Bipartite Matching

- Example matchings in a bipartite graph



Maximum Bipartite Matching

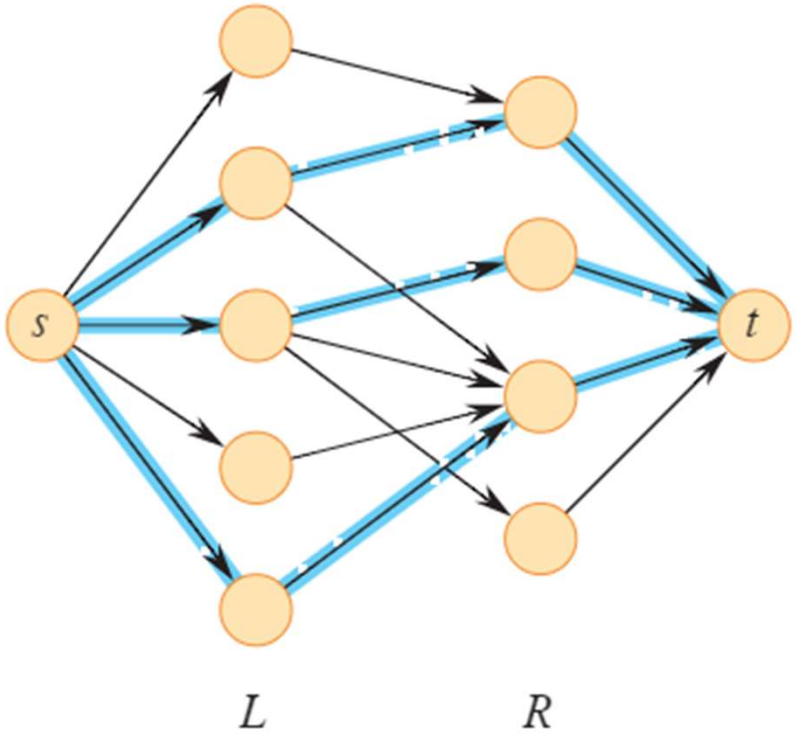
- Example application
  - Matching a set L of machines with a set R of tasks to be performed simultaneously.
  - An edge (u, v) in E signifies that a particular machine $u \in L$ is capable of performing a particular task $v \in R$.
  - A maximum matching provides work for as many machines as possible.



Finding a maximum bipartite matching

- We can reduce the problem of maximum bipartite matching to a maximum flow problem.

- The trick is to construct a flow network in which flows correspond to matchings.



Constructing Flow Network

- We define the corresponding flow network $G' = (V', E')$ for the bipartite graph G as follows.

- Let the source s and sink t be new vertices not in V, and let $V' = V \cup \{s, t\}$.
- If the vertex partition of G is $V = L \cup R$, the directed edges of G' are the edges of E, directed from L to R, along with $|V|$ new directed edges:

$$\begin{aligned} E' = & \{(s,u) : u \in L\} \\ & \cup \{(u,v) : u \in L, v \in R, \text{ and } (u,v) \in E\} \\ & \cup \{(v,t) : v \in R\} \ . \end{aligned}$$

- To complete the construction, assign unit capacity to each edge in E'.
- Since each vertex in V has at least one incident edge, $|E| \geq |V|/2$.
- Thus, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, and so $|E'| = \Theta(E)$.

Matching is a Flow

- Lemma 24.9 shows that a matching in G corresponds directly to a flow in G's corresponding flow network G'.

- We say that a flow f on a flow network $G = (V, E)$ is integer-valued if $f(u, v)$ is an integer for all $(u, v) \in V \times V$.

Lemma 24.9

- Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$ consisting of edges $(u, v) \in E$ such that $f(u, v) > 0$.

- Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.
- Intuitively, each edge $(u, v) \in M$ corresponds to 1 unit of flow in G' that traverses the path $s \to u \to v \to t$.
- Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t.
- The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$, and thus, by Lemma 24.4, the value of the flow is $|f| = |M|$.

Lemma 24.9

- To prove the converse, let f be an integer-valued flow in G' and, as in the statement of the lemma, let

$$M = \{(u, v) : u \in L, v \in R, \text{and } f(u, v) > 0\}.$$

- Each vertex $u \in L$ has only one entering edge, namely (s, u), and its capacity is 1.
- Thus, each $u \in L$ has at most 1 unit of flow entering it, and if 1 unit of flow does enter, by flow conservation, 1 unit of flow must leave.
- Furthermore, since the flow f is integer-valued, for each $u \in L$, the 1 unit of flow enters u if and only if there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow.
- A symmetric argument applies to each $v \in R$.
- The set M is therefore a matching.

Lemma 24.9

- To see that $|M| = |f|$, observe that of the edges $(u, v) \in E'$ such that $u \in L$ and $v \in R$,

$$f(u,v) = \begin{cases} 1 & \text{if } (u,v) \in M\ , \\ 0 & \text{if } (u,v) \notin M\ . \end{cases}$$

- Consequently, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$.
- Lemma 24.4 gives that $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$.

- Based on Lemma 24.9, we can conclude that a maximum matching in a bipartite graph G corresponds to a maximum flow in its corresponding flow network G'.
- Therefore, running a maximum-flow algorithm of G' provides a maximum matching in G.

Integrality

- The only problem left is that the maximum flow algorithm might return a flow in G' for which some $f(u, v)$ is not an integer, even though the flow value $|f|$ must be an integer.

- However, we do not have the problem when we use the Ford-Fulkerson method because we can prove the following theorem.

- Theorem 24.10 (Integrality Theorem)
  - If the capacity function c takes on only integer values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

Summary

- To find a maximum matching in a bipartite undirected graph G, create a flow network G', run the Ford-Fulkerson method on G', and convert the integer-valued maximum flow found into a maximum matching for G.

- Since any matching in a bipartite graph has cardinality at most $\min\{|L|, |R|\} = O(V)$, the value of the maximum flow in G' is $O(V)$.

- Therefore, finding a maximum matching in a bipartite graph takes $O(VE') = O(VE)$ time, since $|E'| = \Theta(E)$.

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr