# CHAPTER 6
GRAPHS

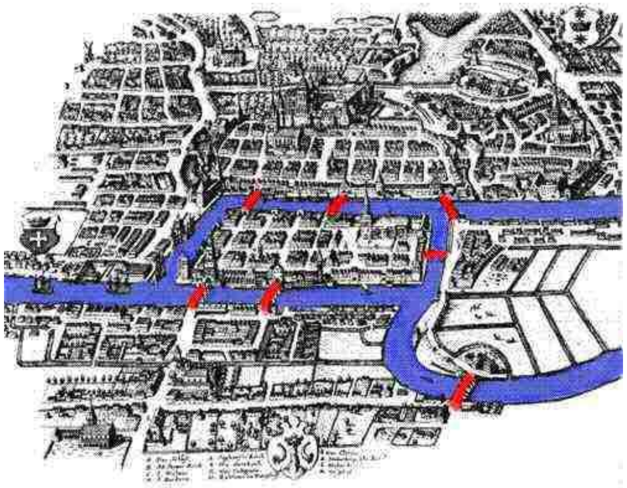
Iris Hui-Ru Jiang Fall 2008

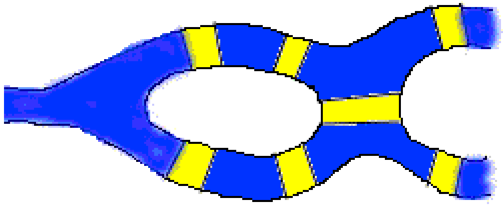
# Graphs

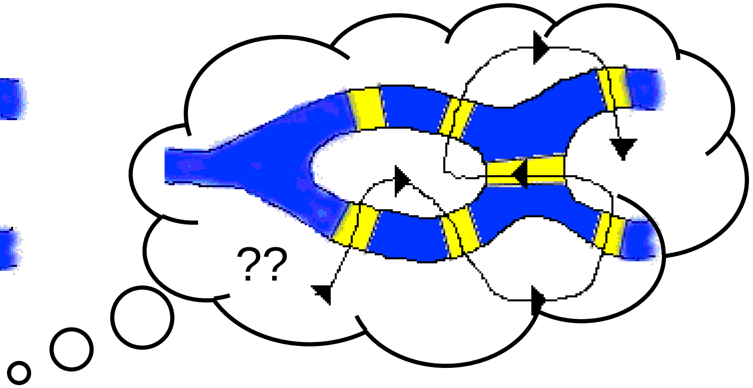
- **Contents**
  - Graphs
  - Graph traversals
  - Connected and biconnected components
  - Minimum-cost spanning trees
  - Shortest paths and transitive closure
  - Topological sort
  - Optional: bipartite graphs
- **Reading**
  - Chapter 6

# The Königsberg Bridge Problem (1/2)

- **Is it possible to walk across all the bridges exactly once and return to the starting land area?**
  - All who tried ended up in failure, including Euler, but he proved
    pronounced "oiler”
  - Is cited as the first paper in both topology and graph theory
    - L. Euler, Solutio problematis ad geometriam situs pertinentis, *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, Vol. 8, pp. 128—140, 1736 (published 1741).

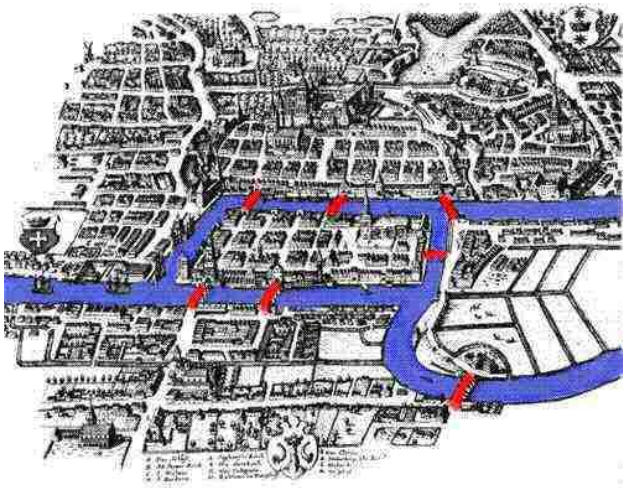


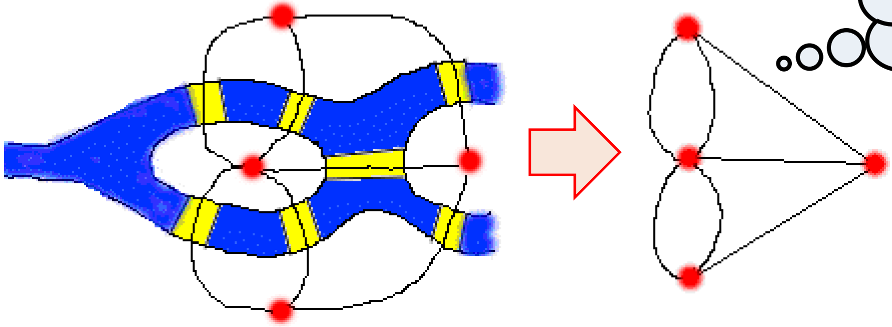




# The Königsberg Bridge Problem (2/2)

- **Euler proved by abstraction!**
  - Eliminate all features except land areas and bridges
  - Replace each land area with a dot (vertex/node), and each bridge with a line (edge/link) ⇒ Obtain a graph
  - Problem: Can you draw this picture without retracing any line and without picking your pencil up off the paper?
  - Observation: Anyone has to enter a land area via one bridge and leave by another ⇒ Each land area needs an even # of bridges





No *Eulerian* walk

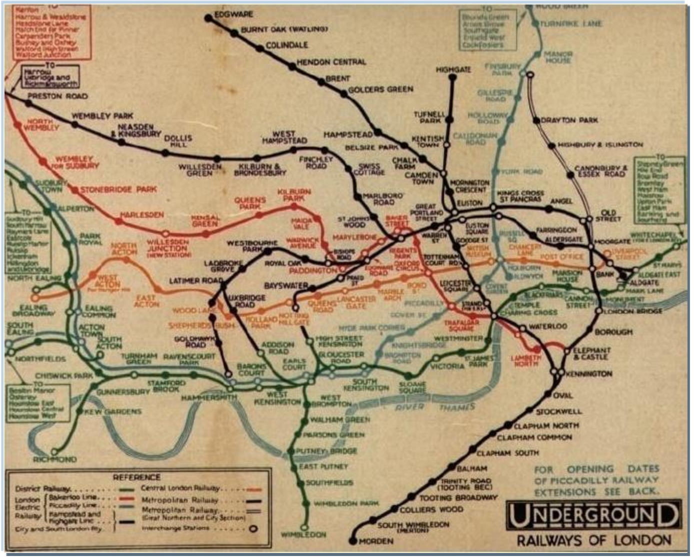
# Abstraction

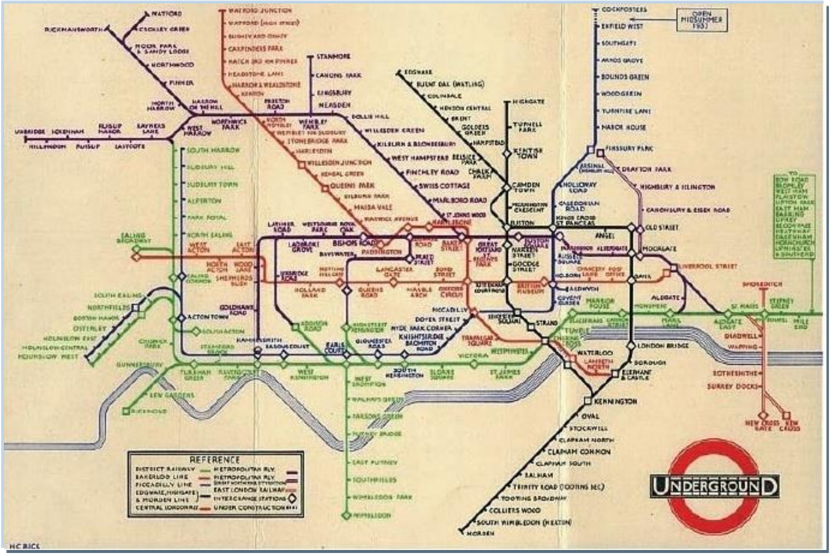
- **All about abstraction!**
  - Vertices: objects
  - Edges: relationship!
  - Examples:

See through the essence

H MATISSE 52

H. Matisse, "Naked Blue IV," 1952; paper cutouts.


UNDERGROUND
RAILWAYS OF LONDON
REFERENCE
FOR OPENING DATES OF PICCADILLY RAILWAY EXTENSIONS SEE BACK.
RIVER THAMES



The London Underground Map (a) the 1928 map (b) the 1933 map by H. Beck.

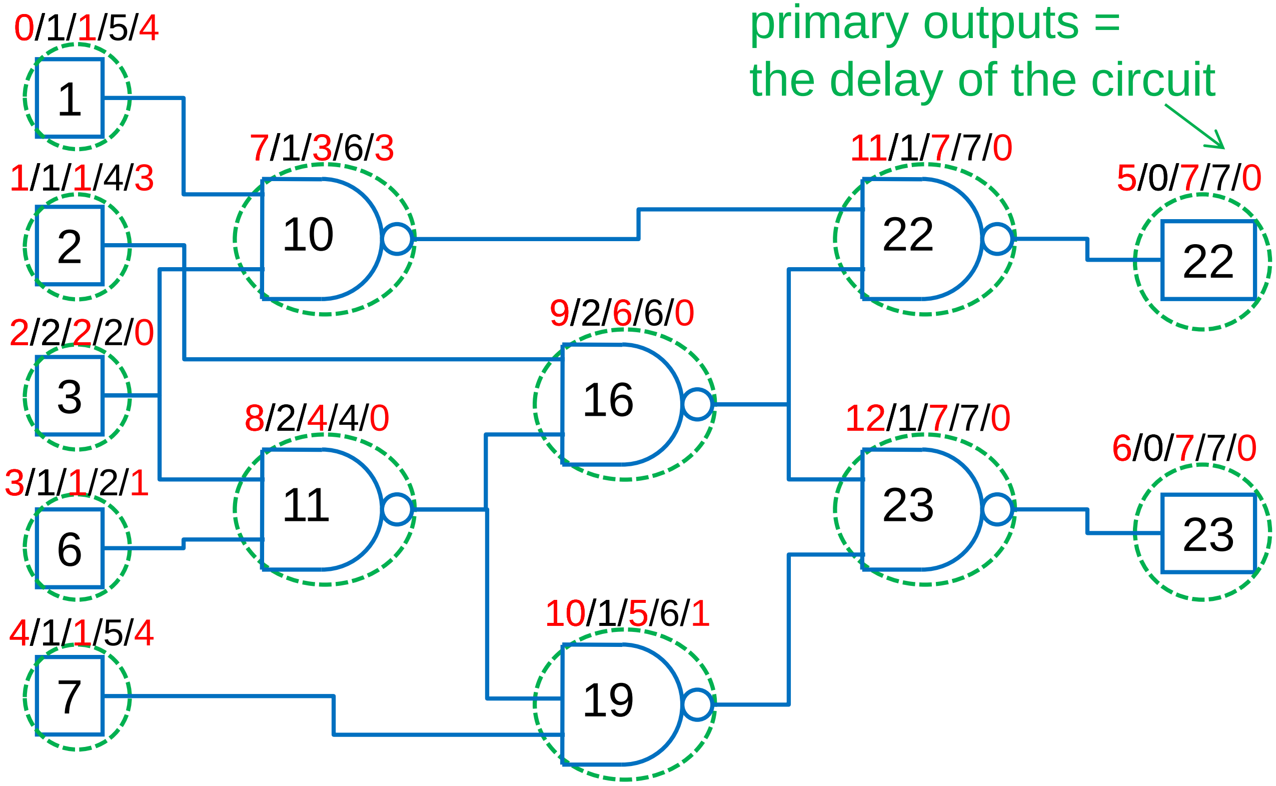
# Example: Static Timing Analysis on C17

- **Vertex: gate/input/output: Assume delay = # of fanout gates**
- **Edge: wire**

| | |
|---|---|
| 0 | INPUT(1) |
| 1 | INPUT(2) |
| 2 | INPUT(3) |
| 3 | INPUT(6) |
| 4 | INPUT(7) |
| 5 | OUTPUT(22) |
| 6 | OUTPUT(23) |
| 7 | 10 = NAND(1, 3) |
| 8 | 11 = NAND(3, 6) |
| 9 | 16 = NAND(2, 11) |
| 10 | 19 = NAND(11, 7) |
| 11 | 22 = NAND(10, 16) |
| 12 | 23 = NAND(16, 19) |

VertexID

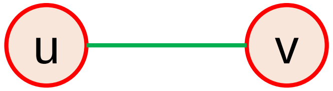
The required time at all primary outputs = the delay of the circuit



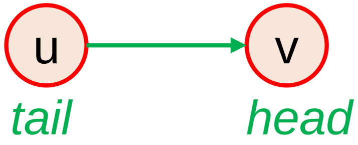
VertexID/delay/arrival/required/slack

# Graphs

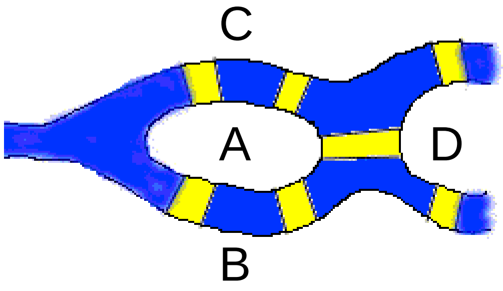
- **Definition: A graph *G* = (*V*, *E*) consists two sets *V* and *E***
  - *V*(*G*): a finite nonempty set of vertices
  - *E*(*G*): a set of edges (pairs of vertices)
- **In an undirected graph:**
  - Edges are undirected, i.e., (*u*, *v*) == (*v*, *u*)

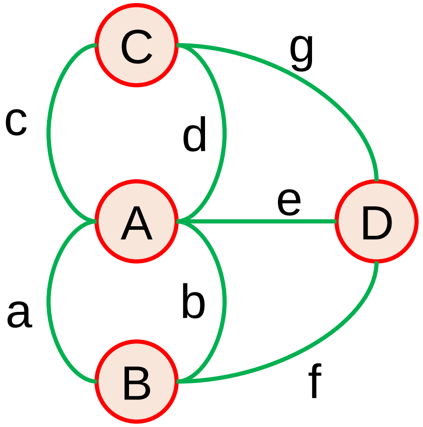


- **In a directed graph:**
  - Edges are directed, i.e., <*u*, *v*> != <*v*, *u*>



- **Example:** the Königsberg bridge problem

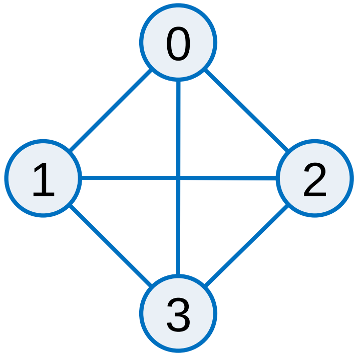




*V*(*G*)={*A*, *B*, *C*, *D*}
*E*(*G*)={*a*, *b*, *c*, *d*, *e*, *f*, *g*}
*a*: (*A*, *B*); *b*: (*B*, *A*);
*c*: (*A*, *C*); *d*: (*C*, *A*);
*e*: (*A*, *D*); *f*: (*B*, *D*); *g*: (*C*, *D*)

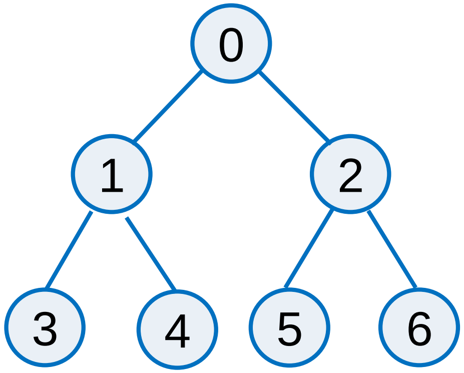
# Sample Graphs

**$G_1$: undirected**



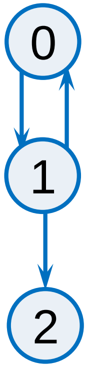
$V(G_1)$={0, 1, 2, 3}
$E(G_1)$={(0,1), (0, 2), (0,3), (1,2), (1,3), (2,3)}

**$G_2$: undirected**



$V(G_2)$={0, 1, 2, 3, 4, 5, 6}
$E(G_2)$={(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}

**$G_3$: directed**

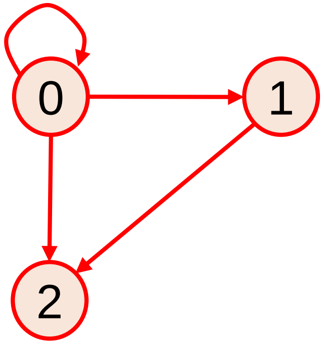
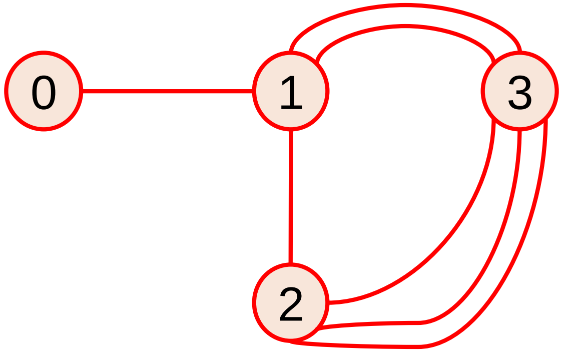


$V(G_3)$={0, 1, 2}
$E(G_3)$={<0,1>, <1,0>, <1,2>}

Note: $G_2$ is also a tree; tree is a special case of graph

# Beyond Simple Graphs (1/2)

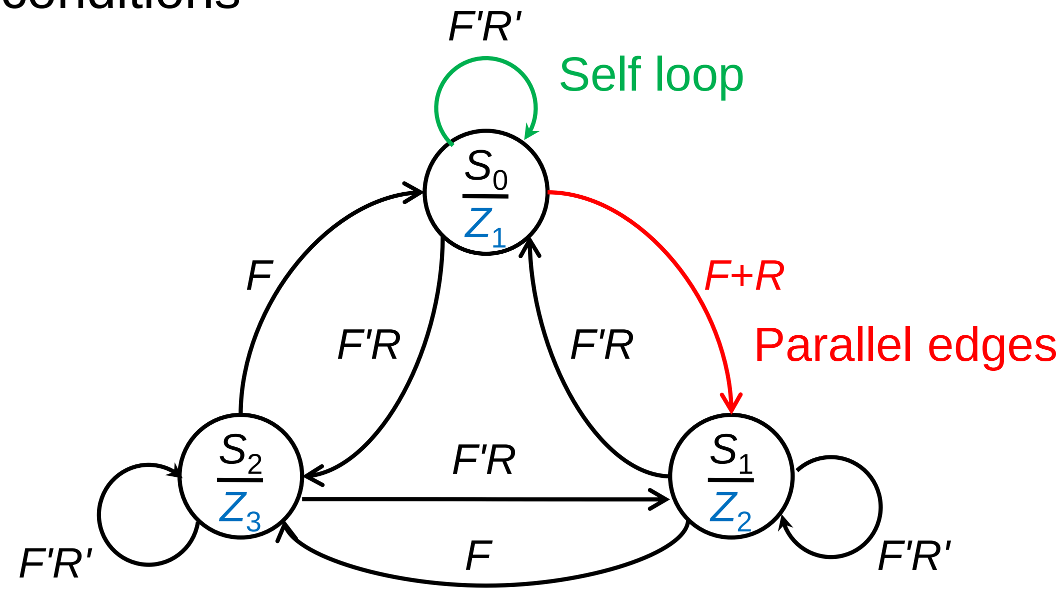
- **Do not consider the following two cases in this book**

| Self edges | Parallel edges |
|---|---|
|  |  |
| Self edge or self loop: (*v*, *v*) or <*v*, *v*> | A multigraph has multiple occurrences of the same edge |

- **You met them before…**
  - State transition graphs in logic design

# Beyond Simple Graphs (2/2)

- **Example: state transition graph**
  - Vertex (object): state
  - Edge (relationship): state transition
  - May have self loops and/or parallel edges
    - Self loop: next state = current state
    - Parallel edges: the same current and next states for several conditions



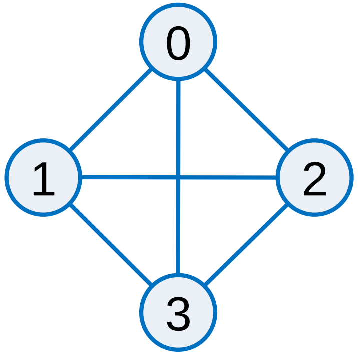
Self loop

Parallel edges

# Complete Graphs

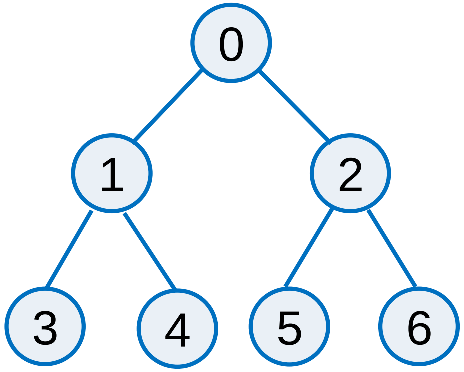
- **A graph is complete if it has the max # of edges**
  - i.e., any pair of vertices have an edge
  - A.k.a. clique
  - For an *n*-vertex graph:
    - Undirected: $n(n-1)/2$ edges
    - Directed: $n(n-1)$ edges

**$G_1$: undirected**



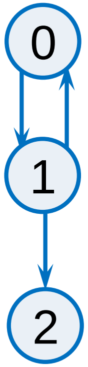
complete
$n(n-1)/2=6$

**$G_2$: undirected**



incomplete
only $n-1=6$ in a tree

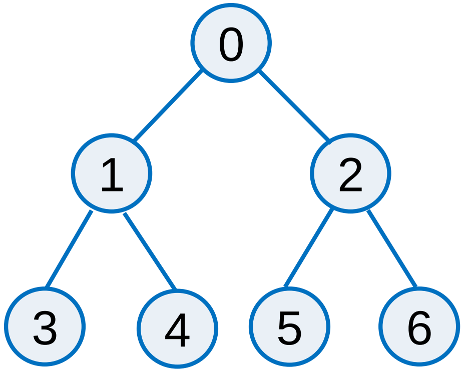
**$G_3$: directed**



incomplete

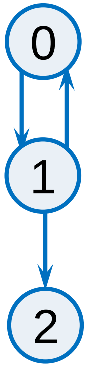
# Adjacency and Incidence

## Undirected



- **If (*u*, *v*) ∈ *E*(*G*)**
  - *u* and *v* are adjacent
  - (*u*, *v*) is incident on *u* and *v*
  - 0, 3, 4 are adjacent to 1
  - (0,2), (2,5), (2,6) are incident on 2

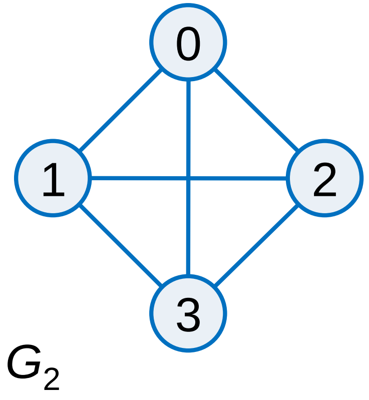
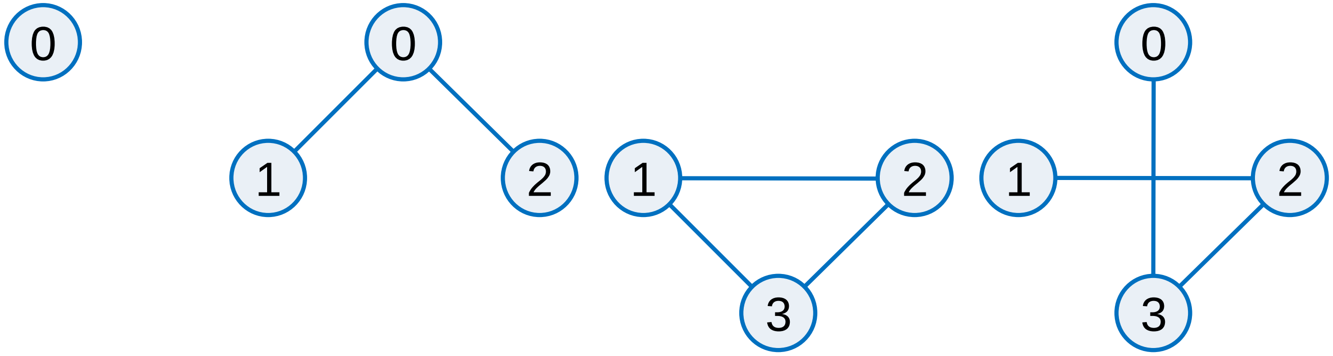
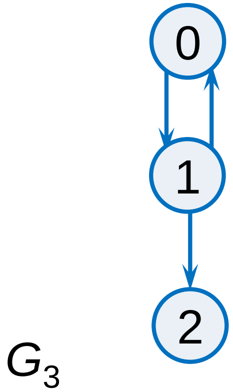
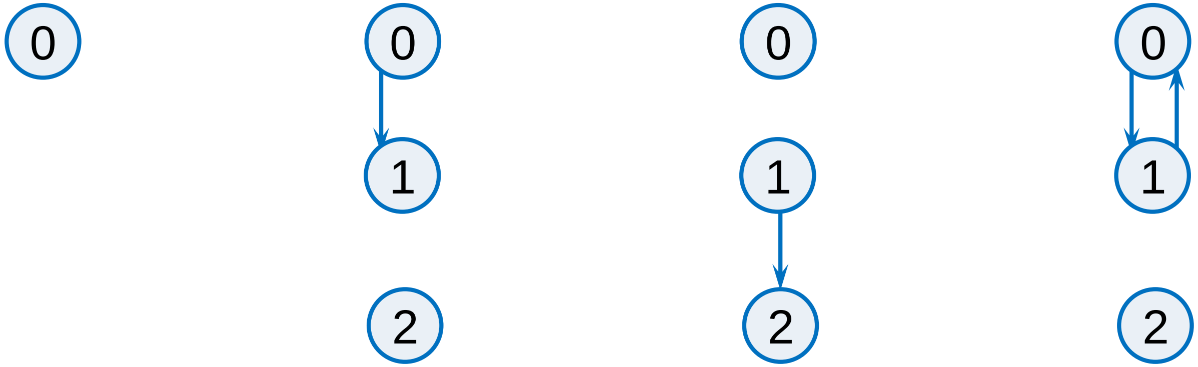
## Directed



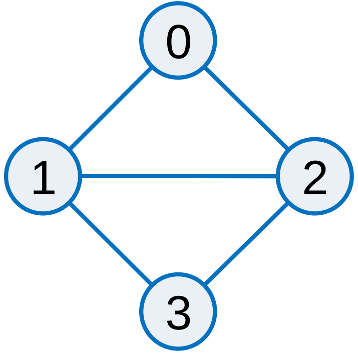
- **If <*u*, *v*> ∈ *E*(*G*)**
  - *u* is adjacent to *v*
  - *v* is adjacent from *u*
  - <*u*, *v*> is incident to *u* and *v*
  - <0,1>, <1,0>, <1,2> are incident to 1

# Subgraphs

- A **subgraph** of $G$ is a graph $G'$ s.t. $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

| Graph | Subgraphs |
|---|---|
|  |  |
|  |  |

# Paths and Cycles



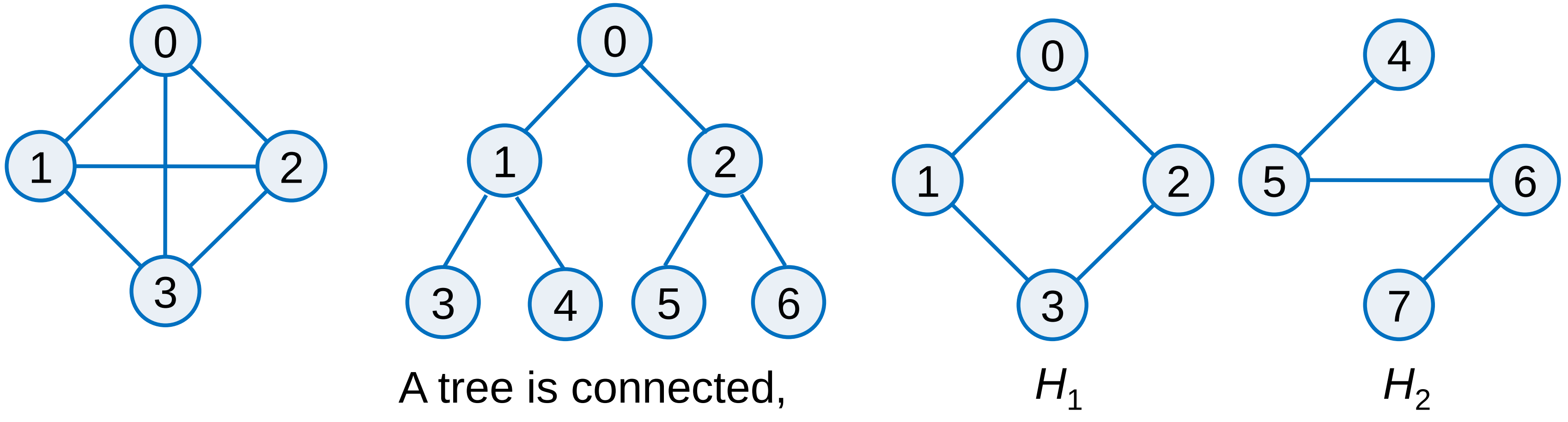
- **A path from *u* to *v* in *G* is**
  - a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ s.t.
    - Undirected: $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ in $E(G)$
    - Directed: $<u, i_1>, <i_1, i_2>, \ldots, <i_k, v>$ in $E(G)$
- **The length of a path is the # of edges on it**
- **A simple path is a path where all vertices except possibly the first and last are distinct**
- **A cycle is a simple path where the first and last vertices are the same**
  - An acyclic graph has no cycles, e.g., a tree
- **Example:**
  - 0, 1, 3, 2 and 0, 1, 2, 3 are paths of length 3
  - 0, 3, 2, 1 is not a path ((0, 3) is not in $E(G)$)
  - 0, 1, 3, 1 is not simple
  - 0, 1, 2, 0 is a cycle

# Connected Graphs

How many edges is required to form a connected graph?

- **In an undirected graph *G*, vertices *u* and *v* are connected iff**
  - There is a path from *u* to *v* (or a path from *v* to *u*)
- **An undirected graph *G* is connected iff**
  - $\forall\ u, v \in V(G), u \neq v$, there exists a path from *u* to *v*
- **A connected component *H* of an undirected graph *G* is a maximal connected subgraphs**
  - Maximal: *No* other connected subgraph in *G* properly contains *H*

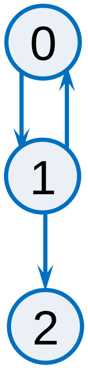
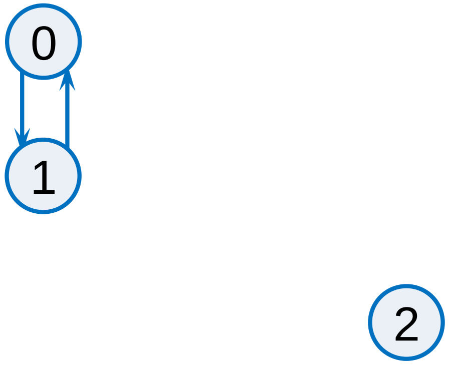
**$G_1$: connected** | **$G_2$: connected** | **$G_4$: two connected components**



A tree is connected, while a forest is not.

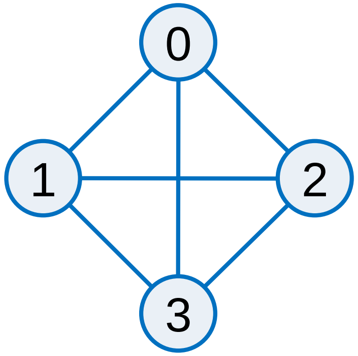
# Strongly Connected Components

- **A directed graph *G* is strongly connected iff**
  - ∀ $u, v \in V(G)$, $u \neq v$, there exists a path from $u$ to $v$ and a path from $v$ to $u$ (bi-directional)
- **A strongly connected component of a directed graph is a maximal strongly connected subgraphs**

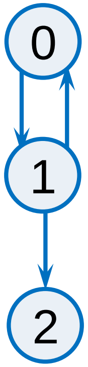
| **$G_3$ is not strongly connected** | **two strongly connected components** |
|---|---|
|  |  |

# Degree

- **The degree of a vertex is the # of edges incident to it**
  - For a directed graph (digraph):
    - in-degree: # of in-coming edges
    - out-degree: # of out-going edges
    - degree = in-degree + out-degree
- **If *G* has *n* vertices and *e* edges,** $e = \frac{1}{2}\sum_{i=1}^{n} d_i$ **(*$d_i$* = degree of vertex *i*)**
- **Example:**



*degree*(0) = 3



*degree*(1) = 3
*in-degree*(1) = 1
*out-degree*(1) = 2

# Graph Representations

**Adjacency matrix**

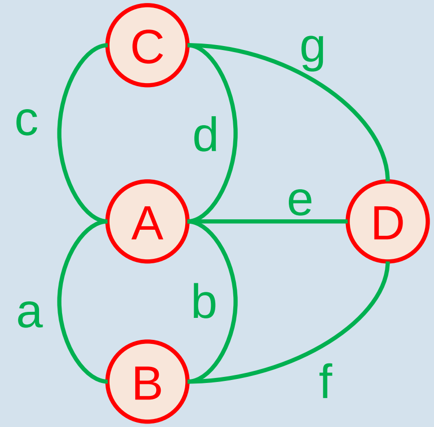
**Adjacency list**

**Adjacency multilist**

# ADT *Graph*

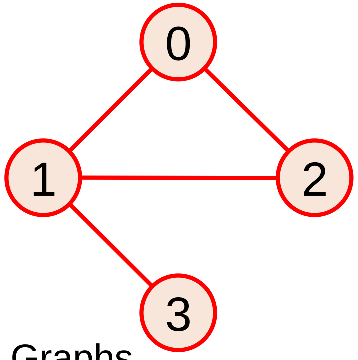
- **Definition: A graph *G* = (*V*, *E*) consists two sets *V* and *E***
  - *V*(*G*): a finite nonempty set of vertices
  - *E*(*G*): a set of edges (pairs of vertices, undirected/directed)

```
class Graph {
// objects: a nonempty set of vertices and
// a set of undirected/directed edges where each edge is a pair of vertices
public:
    virtual ~Graph(); // virtual dtor
    bool IsEmpty() const; {return n == 0};
    int NumberOfVertices() const {return n};
    int NumberOfEdges() const {return e};
    virtual int Degree(int u) const = 0;
    virtual bool ExistsEdge(int u, int v) const = 0;
    virtual void InsertVertex(int v) = 0;
    virtual void InsertEdge(int u, int v) = 0;
    virtual void DeleteVertex(int v) = 0;
    virtual void DeleteEdge(int u, int v) = 0;
private:
    int n;    // # of vertices
    int e;    // # of edges
};
```



# Adjacency Matrix

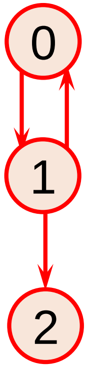
- **The adjacency matrix *a* of *n*-vertex *G* is an *nxn* martix where**
  - $a[i][j] = 1$ iff $(i, j) \in E(G)$ (or $<i, j> \in E(G)$)
  - $a[i][j] = 0$, otherwise
- **Degree?**
  - Undirected: row sum or column sum
  - Directed: in-degree = column sum; out-degree = row sum
- **Space: O($n^2$)**
  - Suitable for dense graph
  - How to save space if undirected?
  - What if sparse graph?



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

symmetric

no self loops



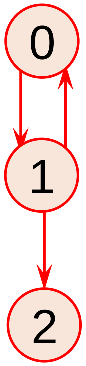
$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

non-symmetric

no self loops

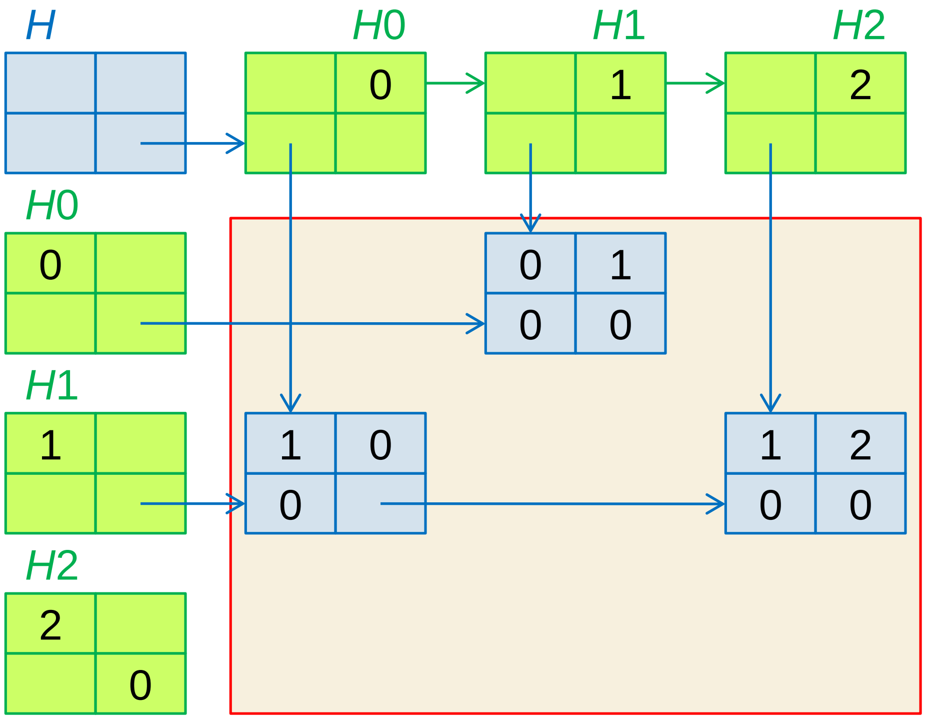
# Orthogonal List Representation

- □ **Use simplified sparse matrix representation (ref. Chap 4)**



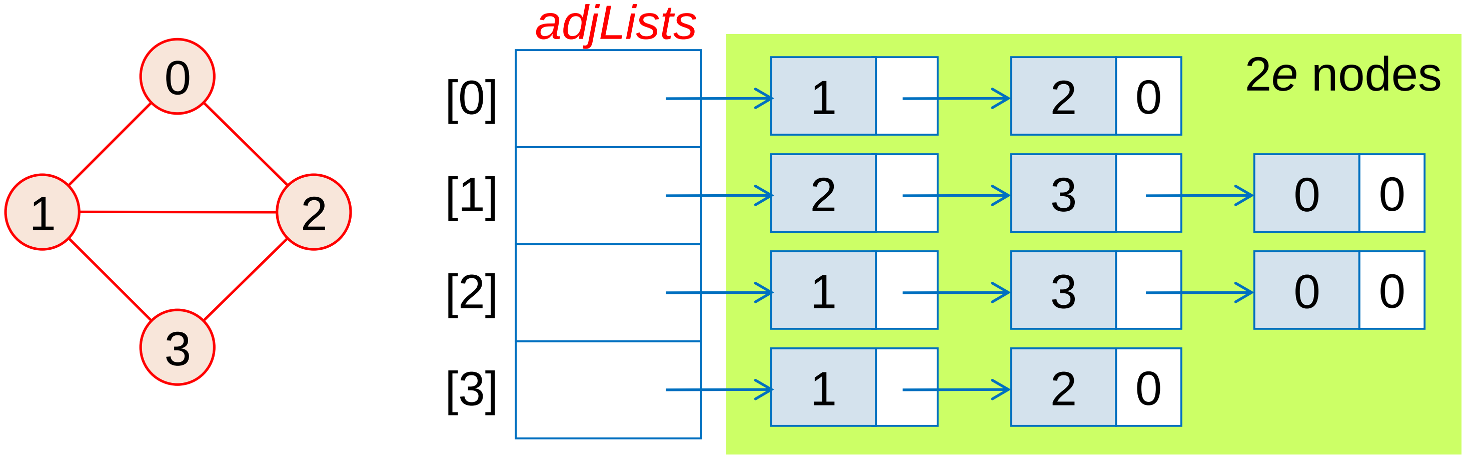
$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

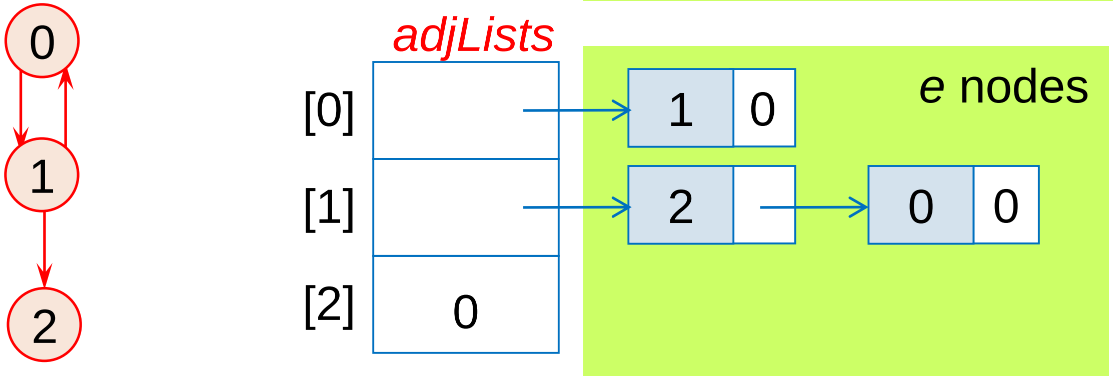
| | |
|---|---|
| *tail* | *head* |
| *link-in* | *link-out* |



# Adjacency List

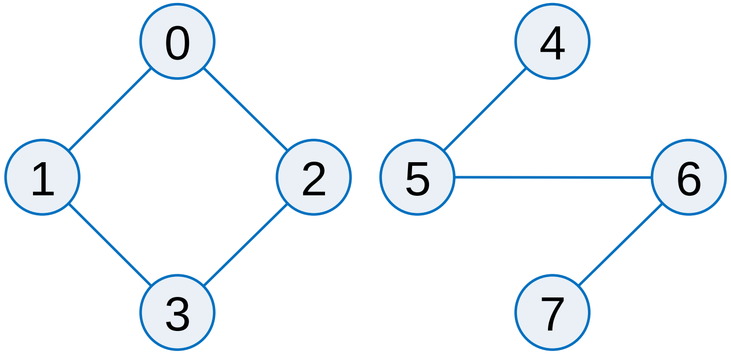
- **The adjacency list is an array *adjLists* of *n* chains, one for each vertex represents vertices adjacent from it**
- **Space: O(*n*+*e*)**
  - Good for sparse graph





# Sequential Representation of Adjacency List

- **Pack the adjacency lists into an array *node*[*n*+2*e*+1]**
- **Small storage but slow insertion/deletion**



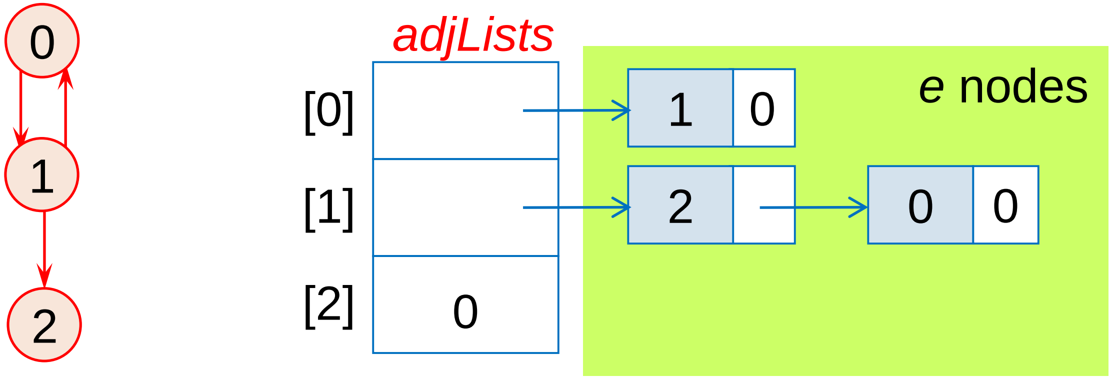
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] | [20] | [21] | [22] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 11 | 13 | 15 | 17 | 18 | 20 | 22 | 23 | 2 | 1 | 3 | 0 | 0 | 3 | 1 | 2 | 5 | 6 | 4 | 5 | 7 | 6 |

*n*+1 entries

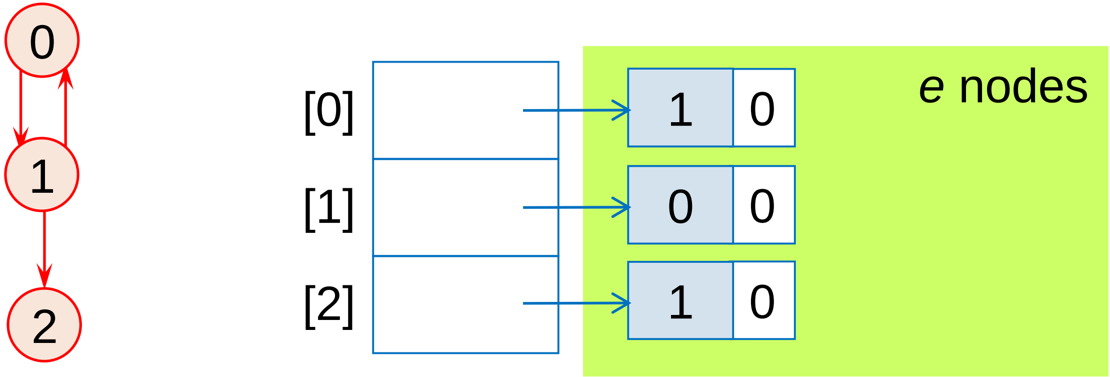
2*e* entries

# Inverse Adjacency List

- **How to determine in-degree in an adjacency list?**



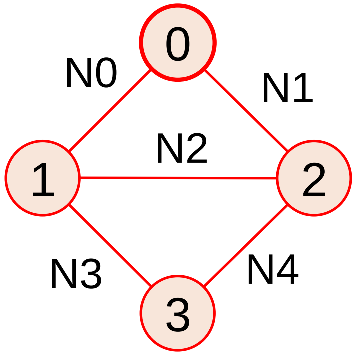
- **Inverse adjacency list records vertices adjacent to it**

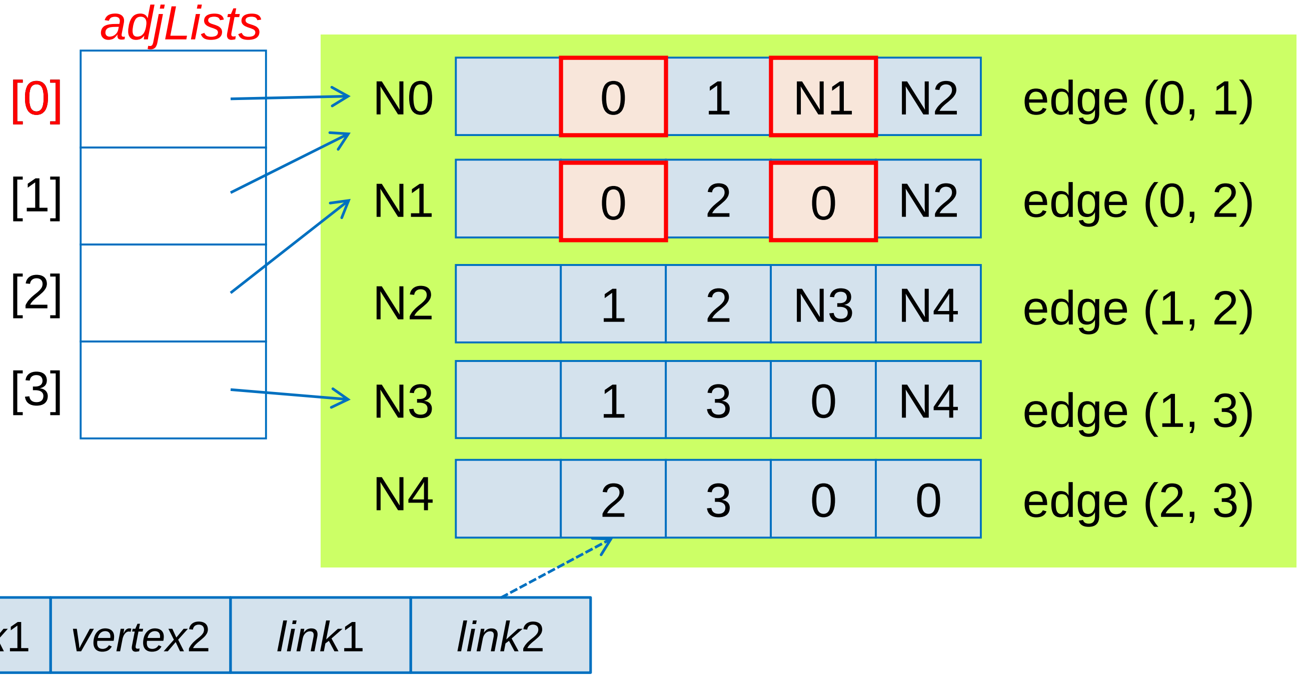


- **Keep both if repeatedly accessing all vertices**

# Adjacency Multilists

- **Represent each edge by one node**
  - ⇒ Each edge is in two lists
  - Vertex 0: N0 -> N1
  - Vertex 1: N0 -> N2 -> N3
  - Vertex 2: N1 -> N2 -> N4
  - Vertex 3: N3 -> N4



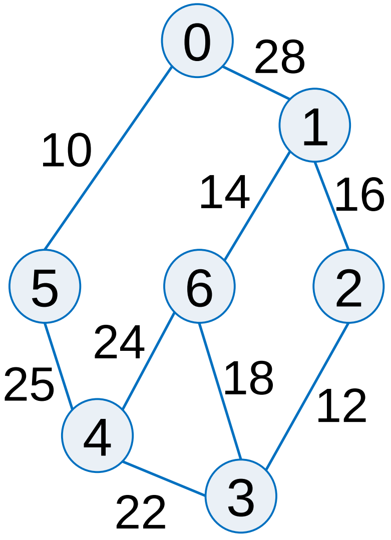


| | | | | | | |
|---|---|---|---|---|---|---|
| N0 | | 0 | 1 | N1 | N2 | edge (0, 1) |
| N1 | | 0 | 2 | 0 | N2 | edge (0, 2) |
| N2 | | 1 | 2 | N3 | N4 | edge (1, 2) |
| N3 | | 1 | 3 | 0 | N4 | edge (1, 3) |
| N4 | | 2 | 3 | 0 | 0 | edge (2, 3) |

| | | | | |
|---|---|---|---|---|
| *m* | *vertex*1 | *vertex*2 | *link*1 | *link*2 |

# Weighted Edges

- **A graph with weighted edges is called a network**
  - Weights can be distance, cost, or other quantities…
  - Adjacency matrix: *a*[*i*][*j*] keeps non-zero weights instead of 1
  - Adjacency list: require an additional field



# Adjacency Matrix vs. Adjacency List

| Comparison | Winner |
|---|---|
| Faster to find an edge? | Matrix |
| Faster to find degree? | List |
| Faster to traverse the graph? | List |
| Storage for sparse graph? | List |
| Storage for dense graph? | Matrix |
| Edge insertion or deletion? | Matrix |
| Weighted-graph implementation? | ? |
| Better for most applications? | List |

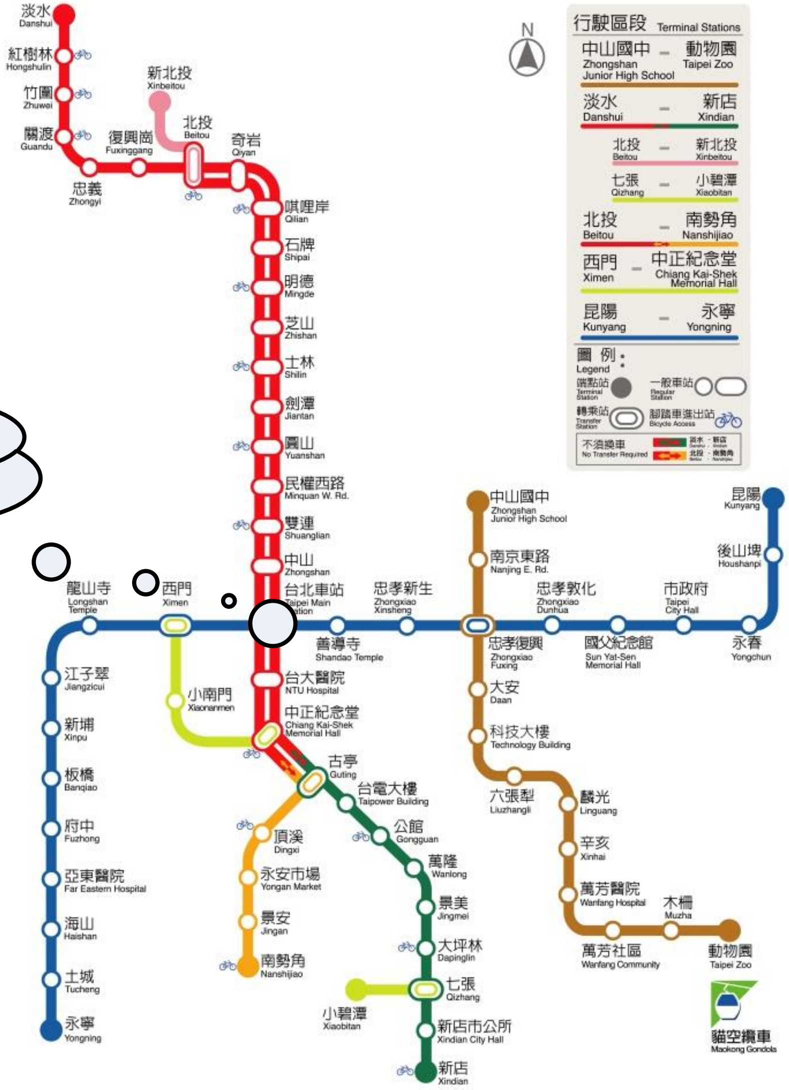
# Elementary Graph Operations

**Traversals and applications**

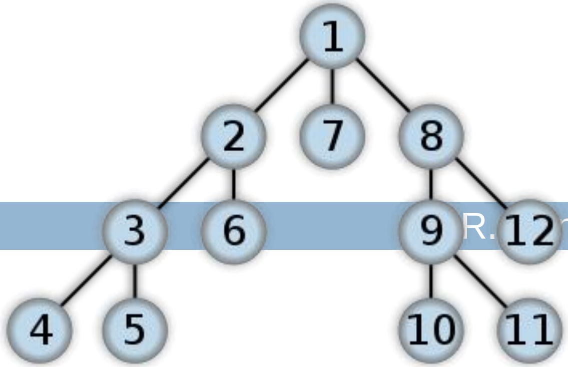
# Graph Traversals

- □ **Given *G* = (*V*, *E*) and vertex *v*, visit all vertices connected to *v***
  - Find vertices reachable from *v*
- □ **Depth First Search (DFS)**
  - Cf. Preorder tree traversal
- □ **Breadth First Search (BFS)**
  - Cf. Level-order tree traversal

From Taipei main station,
where can you go by MRT?



# Depth-First-Search (DFS)

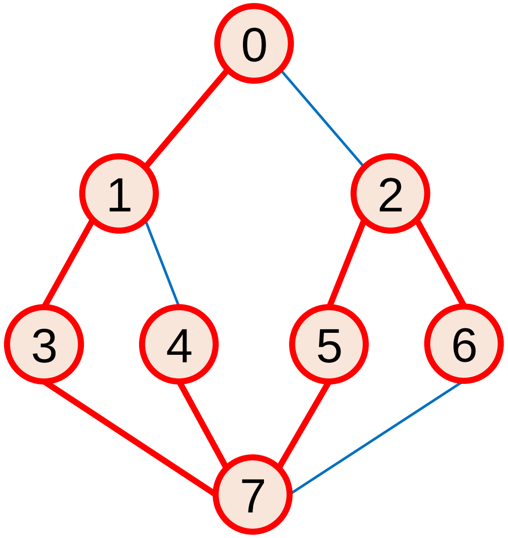


Order in which the nodes are expanded

- **Idea: reach out by path finding**
- **Time complexity:**
  - Adjacency list: O($n$+$e$)
  - Adjacency matrix: O($n^2$)

```
void Graph::DFS() { // Driver
    visited = new bool[n]; // n vertices: 0..n-1
    for (int i = 0; i < n; i++)
        visited[i] = false; // initially, all vertices are unvisited
    DFS(0); // begin with vertex 0
    delete [] visited;
}

void Graph::DFS(int v) { // Workhorse
// Visit all unvisited vertices reachable from vertex v
    visited[v] = true;
    for (each vertex w adjacent to v)
        if (!visited[w]) DFS(w); // recursive call
}
```

Stack-based

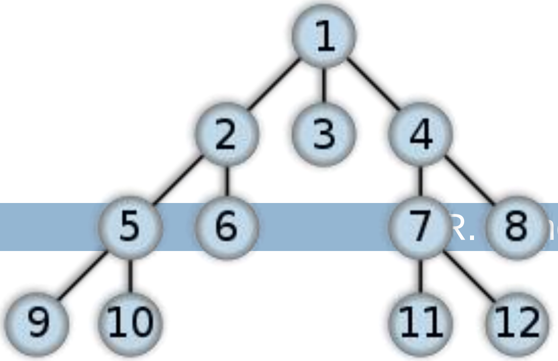


DFS traversal order:
0, 1, 3, 7, 4, 5, 2, 6

Cf. Preorder

# Breadth-First-Search (BFS)

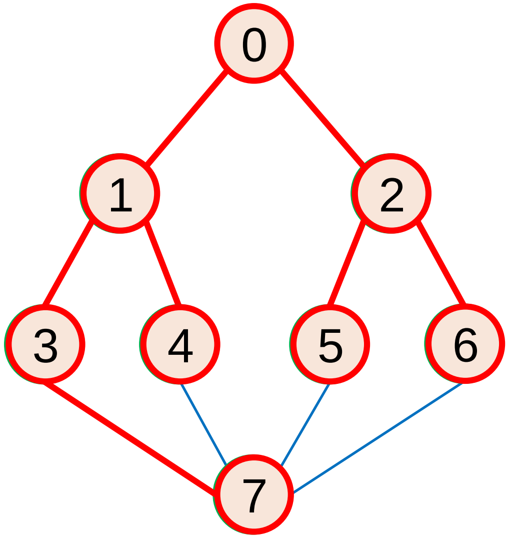
- Idea: propagate the waves
- Time complexity:
  - Push each vertex into queue once
  - Adjacency list: O($n$+$e$)
  - Adjacency matrix: O($n^2$)



Order in which the nodes are expanded

```
void Graph::BFS(int v) {
    visited = new bool[n];
    for (int i = 0; i < n; i++) visited[i] = false;
    Queue<int> q;
    visited[v] = true; q.Push(v); // begin with vertex v
    while (!q.IsEmpty()) {
        v = q.Front(); q.Pop();
        for (each vertex w adjacent to v)
            if (!visited[w]) {
                visited[w] = true; q.Push(w);
      } // end of while
  }   delete [] visited;
}
```

Queue-based

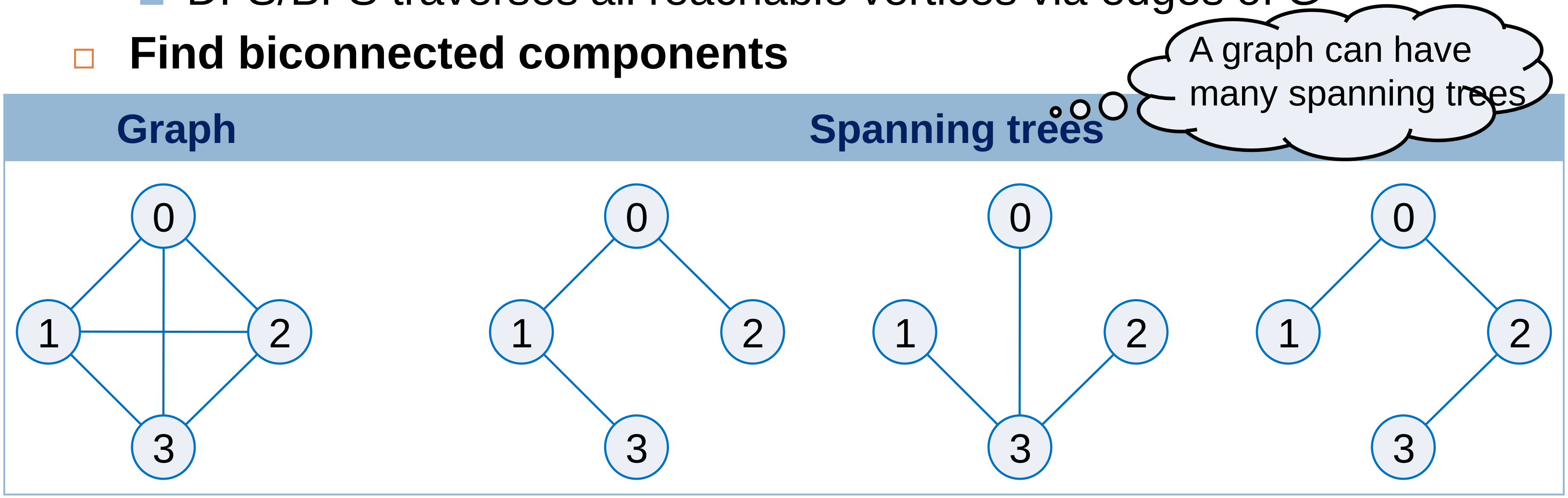


BFS traversal order:
0, 1, 2, 3, 4, 5, 6, 7

Cf. Level-order

# Applications of Graph Traversals

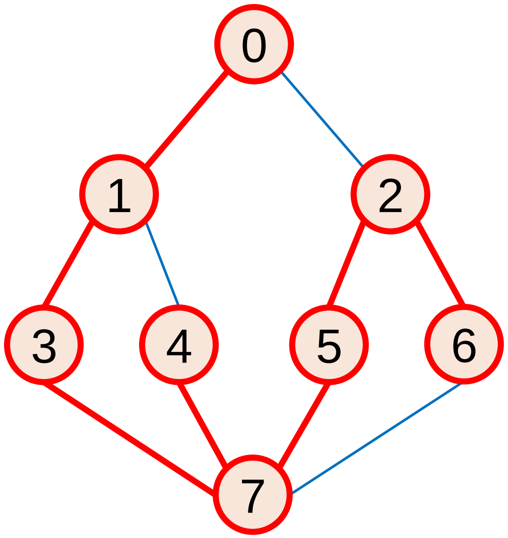
- **Find connected components**
  - Since DFS/BFS visits all vertices connected to some vertex
- **Find a spanning tree of a connected graph**
  - A spanning tree of *G* is a tree that
    - Includes all vertices in *G* (i.e., spans over vertices)
    - Uses partial/all edges in *G* (i.e., *n*-1 edges)
  - DFS/BFS traverses all reachable vertices via edges of *G*
- **Find biconnected components**



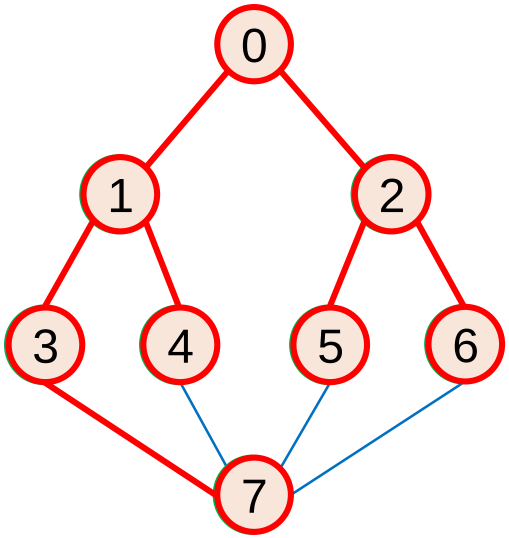
A graph can have many spanning trees

# Depth-First/Breadth-First Spanning Trees

- DFS/BFS visits each reachable vertex via some edge of *G*
  - The reachable vertices and these edges form a spanning tree



*DFS*(0)

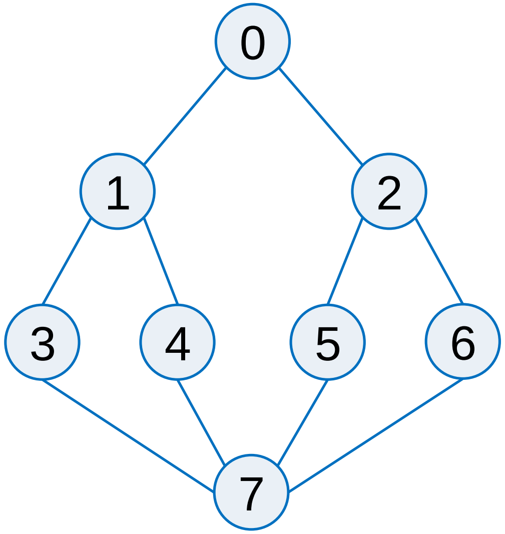


*BFS*(0)

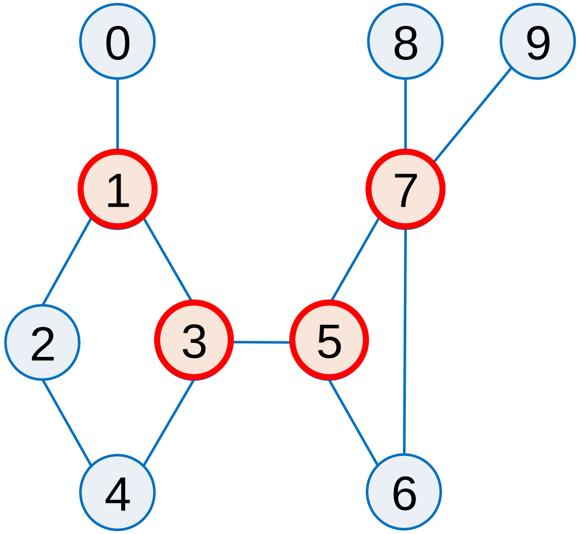
- Q: What will you get if the graph is not fully connected?

# Biconnected Graphs

- **Definition: An articulation point *v* of a connected graph *G*:**
  - Deleting *v* and all edges incident to *v* results in *G* disconnected
- **A biconnected graph is a connected graph without articulation points**
  - e.g., a communication network is desirable to be biconnected



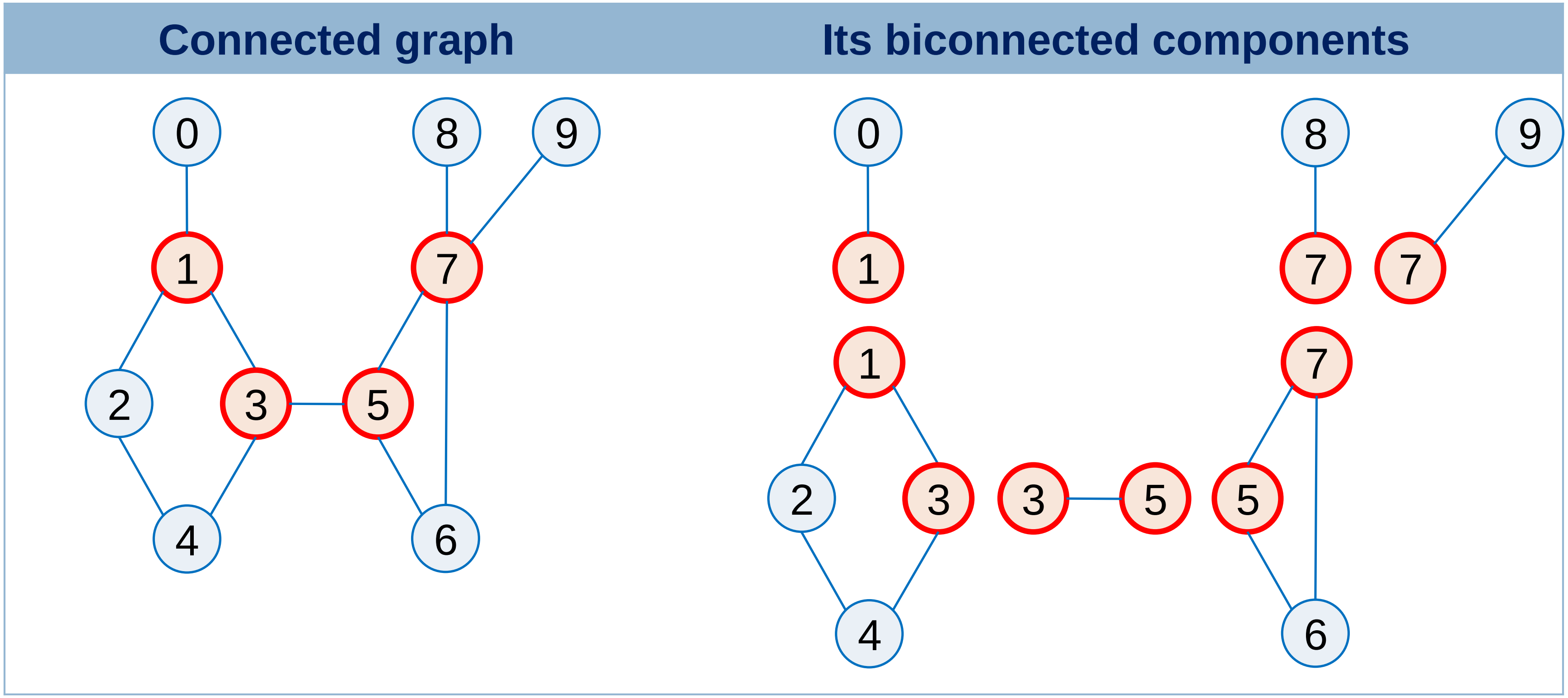
Biconnected graph



Not biconnected graph
Articulation points: 1, 3, 5, 7

# Biconnected Components

- **A biconnected component of a connected graph *G* is a maximal biconnected subgraph *H* of *G***
  - ⇒ A biconnected graph has only one biconnected component
  - ⇒ Finding biconnected components == finding articulation points

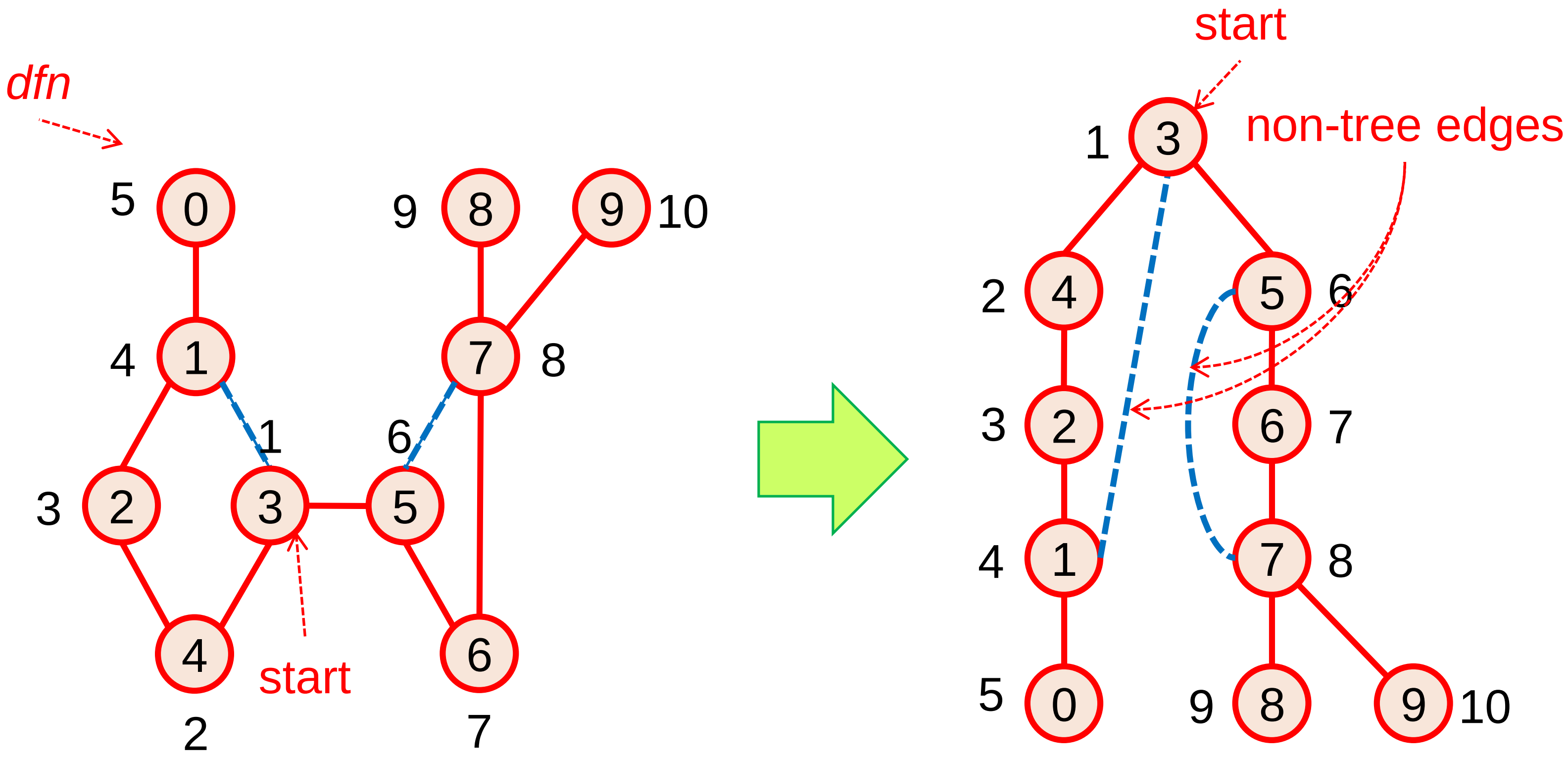


**Connected graph**

**Its biconnected components**

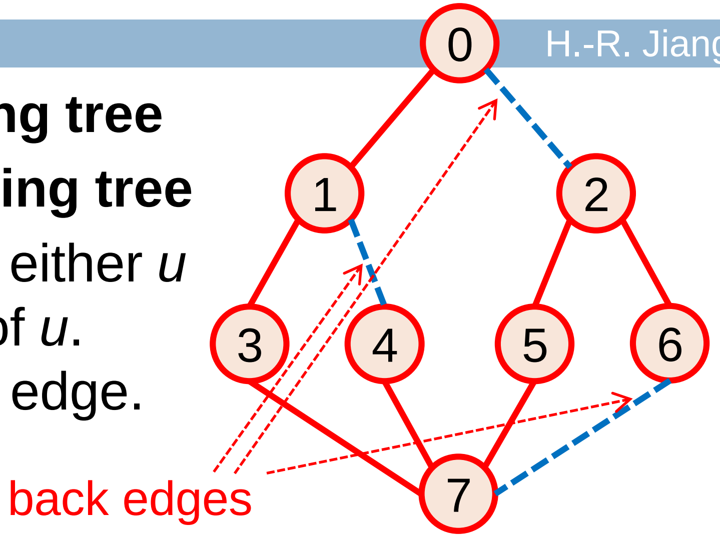
# Depth-First Number

- **Depth-first number**, ***dfn***: **the visit order during DFS**



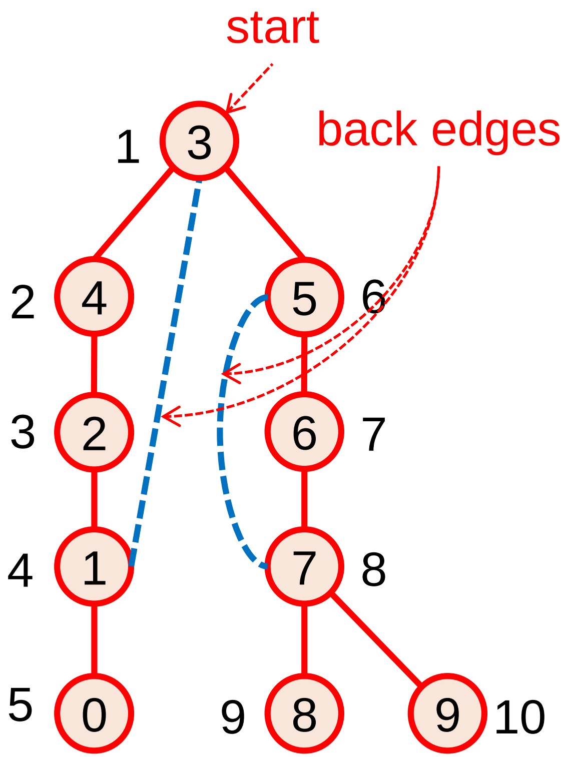
# Back Edges vs. Cross Edges

- No **cross** edges in a depth-first spanning tree
- No **back** edges in a breadth-first spanning tree
  - A nontree edge (*u*, *v*) is a back edge iff either *u* is an ancestor of *v* or *v* is an ancestor of *u*. Otherwise, the nontree edge is a cross edge.

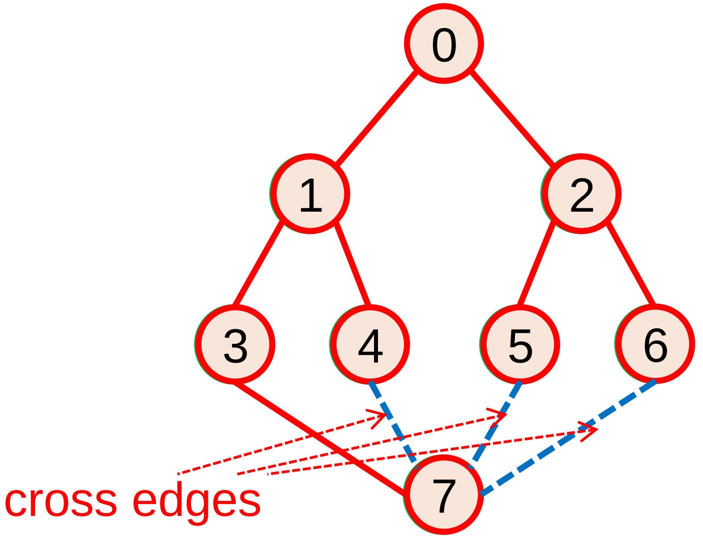


back edges

*DFS*(0) spanning tree


start

back edges

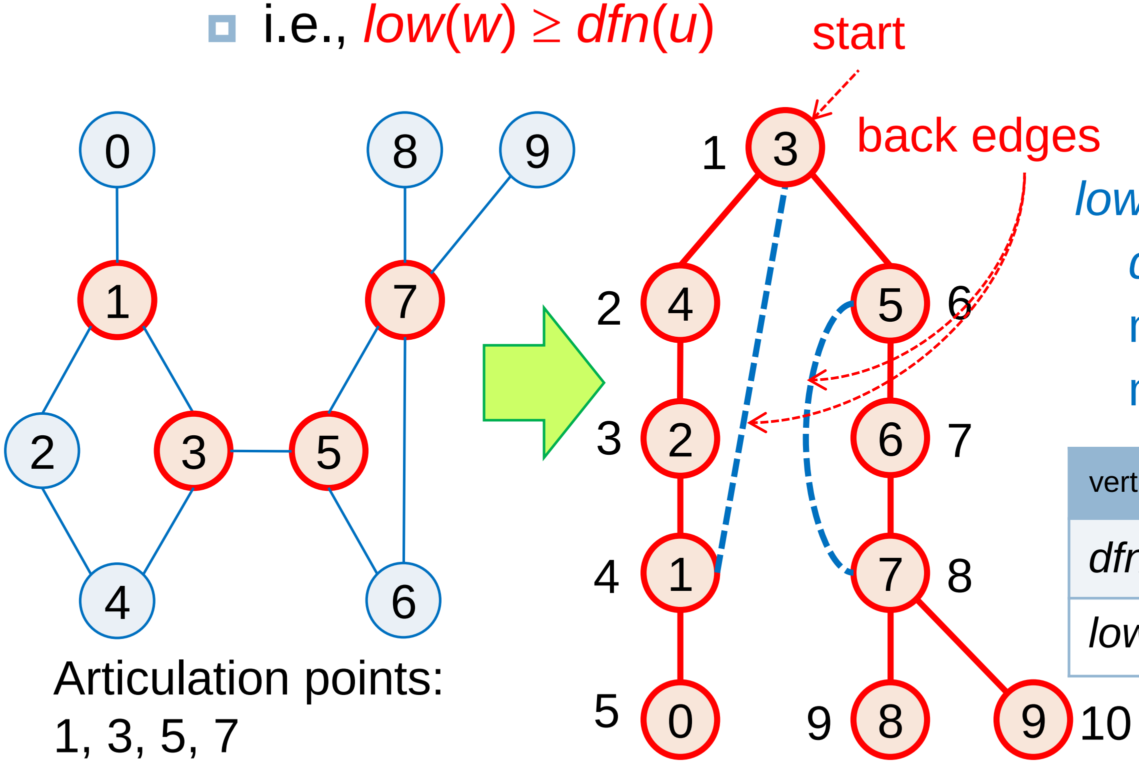


cross edges

*BFS*(0) spanning tree

# Who is an Articulation Point?

- **The root of DFS spanning tree iff it has at least 2 children**
- **Any other vertex *u* iff it has at least one child *w* s.t.**
  - It's impossible to reach an ancestor of *u* using a path composed solely of *w*, descendants of *w* and a single back edge
  - i.e., *low*(*w*) ≥ *dfn*(*u*)



Articulation points:
1, 3, 5, 7

*low*(*w*) = min{
*dfn*(*w*),
min{*low*(*x*) | *x* is *w*'s child},
min{*dfn*(*x*) | (*w*, *x*) is a back edge}}

| vertex | 0 | 1 | 2 | 3 | 4 | **5** | **6** | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| *dfn* | 5 | 4 | 3 | 1 | 2 | **6** | 7 | 8 | 9 | 10 |
| *low* | 5 | 1 | 1 | 1 | 1 | 6 | **6** | 6 | 9 | 10 |

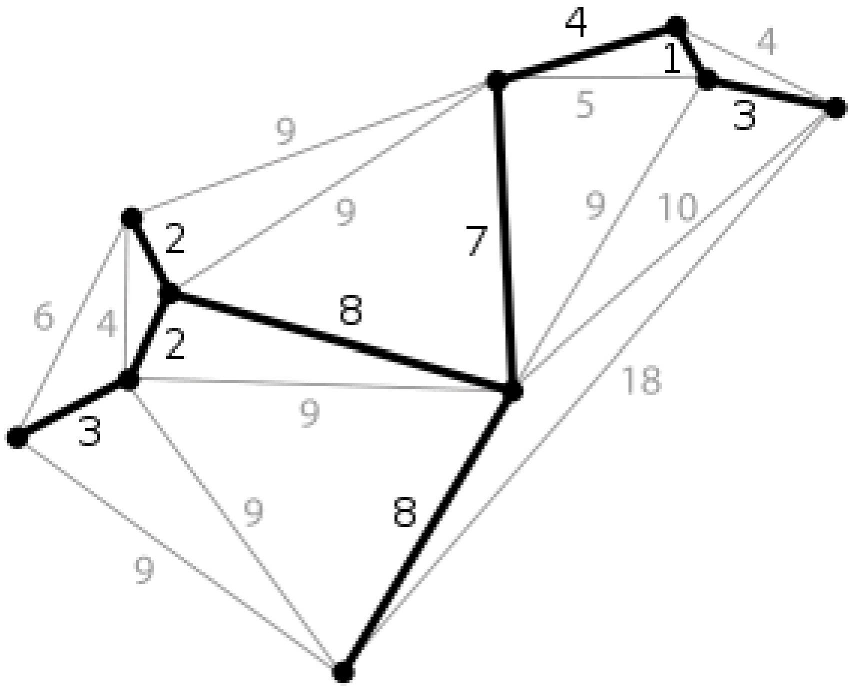
# Computing *dfn* and *low*

```
void Graph::DfnLow(const int x) { // begin DFS at Vertex x
    num = 1;  // num is an int data member of class Graph
    dfn = new int[n]; // dfn is also a data member
    low = new int[n]; // low is also a data member
    for (int i = 0; i < n; i++) dfn[i] = low[i] = 0;
    DfnLow(x, -1); // start at x
    delete[] dfn; delete [] low;
}
void Graph::DfnLow (int u, int v) {
    dfn[u] = low[u] = num++; // visit it!
    for (each vertex w adjacent from u)
        if (dfn[w] == 0) { w is unvisited
            DfnLow(w, u); // recursive
            low[u] = min(low[u], low[w]);
        } else if (w != v)  // v: u's parent, (w, v): back edge
            low[u] = min(low[u], dfn[w]);
}
```

```
void Graph::DFS() {
    visited = new bool[n];
    for (int i = 0; i < n; i++)
        visited[i] = false;
    DFS(x); // begin with vertex 0
    delete [] visited;
}

void Graph::DFS(int v) {
    visited[v] = true;
    for (each vertex w adjacent to v)
        if (!visited[w]) DFS(w);
}
```

# Minimum-Cost Spanning Trees

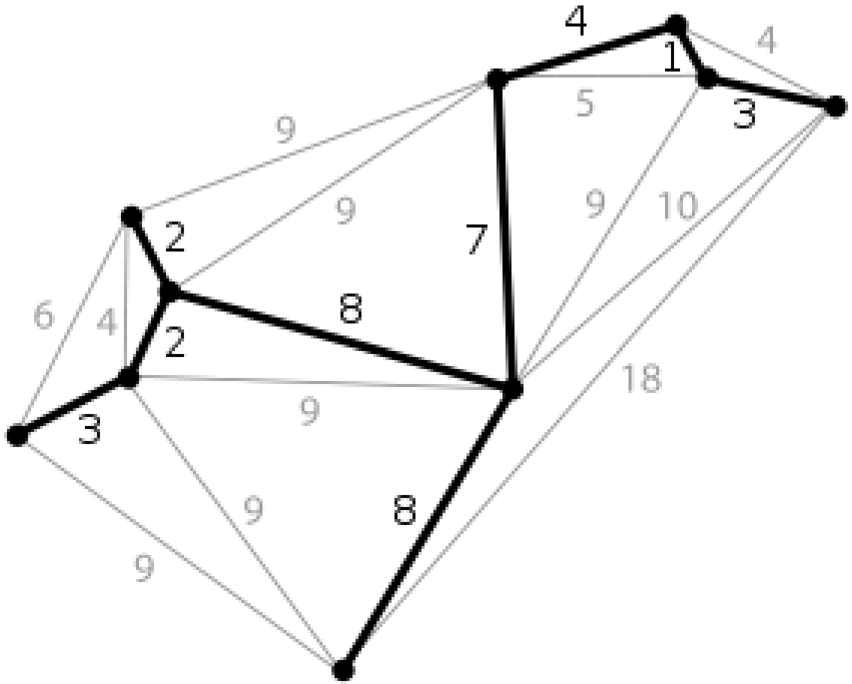
- **Road construction**
- **Water pipes**
- **Routing**



# How to Construct an MST on Your Own?

- **Rule:**
  - Grow a tree
    to connect all vertices
    s.t. the total cost is minimized
- **Q: How?**
- **Hint: Greedy**
- **Basic idea:**
  1. Start from a vertex (or edge)
  2. Expand the tree

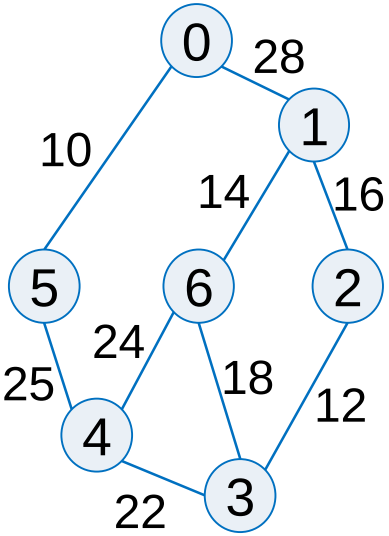
```
MST(G,W)
T = {} // empty ∅
while T does not form a spanning tree do
    find an edge e in E that is safe for T
    T = T union e
return T
```



# Minimum-Cost Spanning Trees

A graph may have many MSTs

- **A.k.a. minimum spanning tree, MST**
- **A minimum-cost spanning tree of a weighted undirected graph:**
  - Tree: has no cycles (selects |*V*|-1 edges)
  - Spanning: spans over all vertices
  - Minimum-cost: has least cost
    - Cost = the sum of the weights of the tree edges
- **Applications: railway, routing, etc.**
  - Build a railway system to connect *n* cities, with the smallest total length of the railroad
- **Known algorithms:**
  - Kruskal's, Prim's, Sollin's
  - Greedy: Make the best decision at each stage

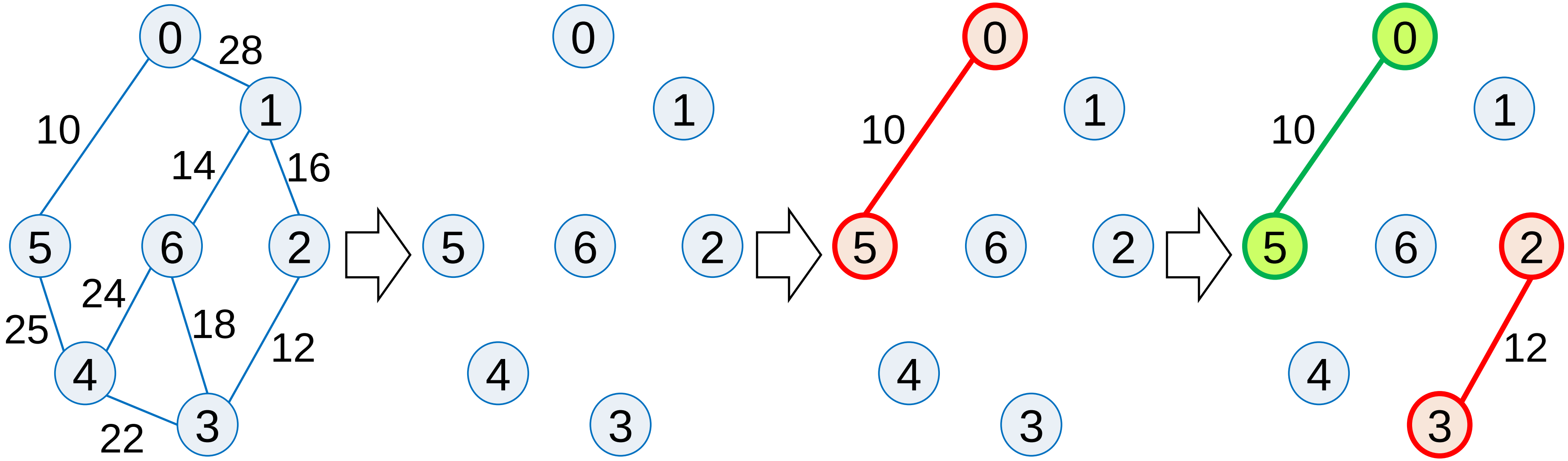


# Kruskal's Algorithm (1/2)

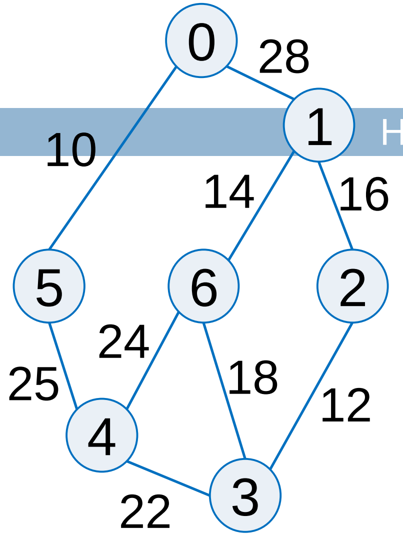
- **Preprocessing: Sort edge weights in nondecreasing order**

1. **Begin with no edges selected**
   - Initially, a forest of |*V*| trees ⇒ Finally, a tree
2. **Add one edge into *T* at a time**
   - Select an edge for inclusion in *T* in nondecreasing order if the resultant *T* does not contain a cycle
   - Only |*V*|-1 edges are included at last

|*V*|: size of *V*

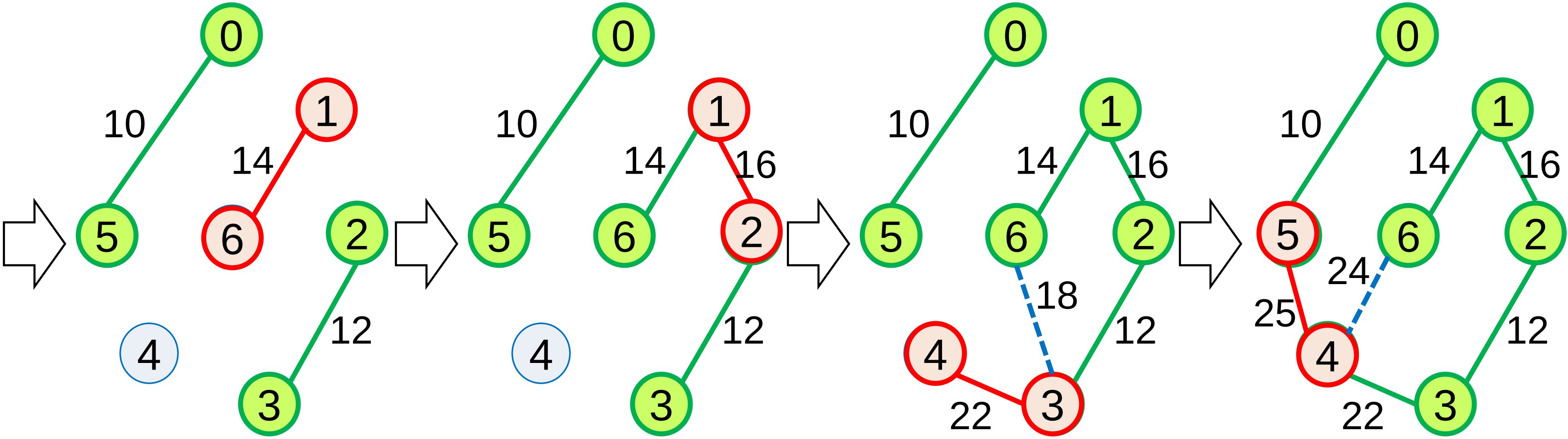


# Kruskal's Algorithm (2/2)



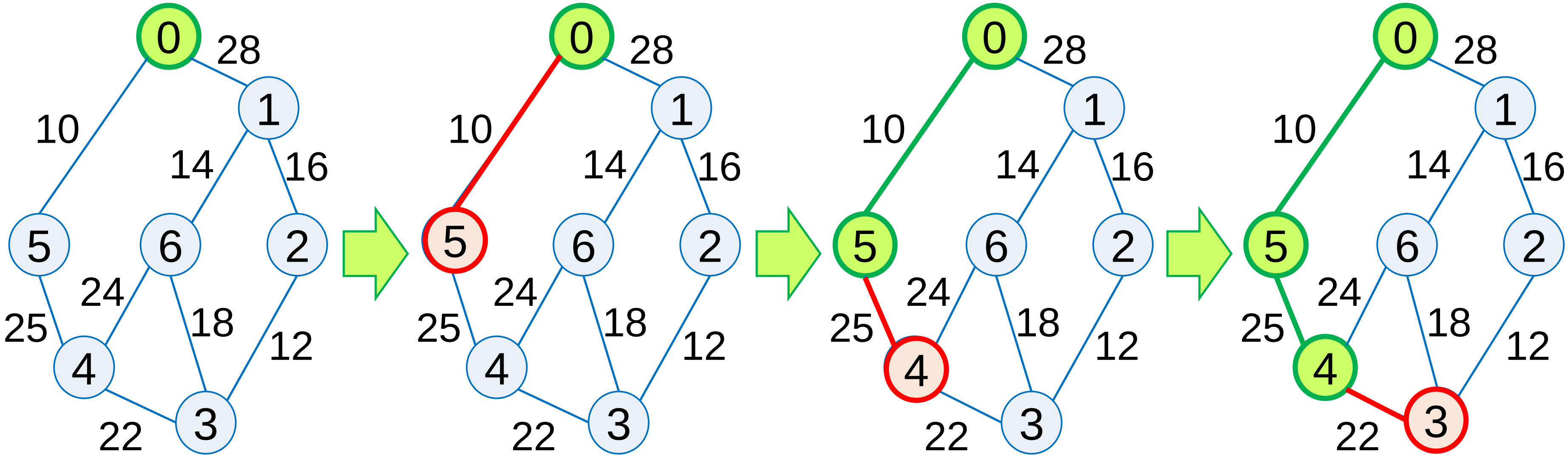
```
Kruskal(G, w)
T = ∅ // record MST edges in T
Sort edge weight w in nondecreasing order
while |T| < |V|–1 and E ≠ ∅ do
    remove min weighted edge (u, v) from E
    if (u, v) is safe for T then add (u, v) to T
```

- **Time complexity: O(*E* lg *E*)**



# Prim’s Algorithm (1/2)

1. **Begin with a tree *T* containing a single vertex**
   - Any vertex in the original graph
   - Initially, a tree with one vertex ⇒ Finally, a tree with all vertices
2. **Add a min weight edge (*u*, *v*) to *T* s.t.**
   - $T \cup \{(u, v)\}$ is still a tree ⇒ safe!
   - *u* is in *T*, *v* is not

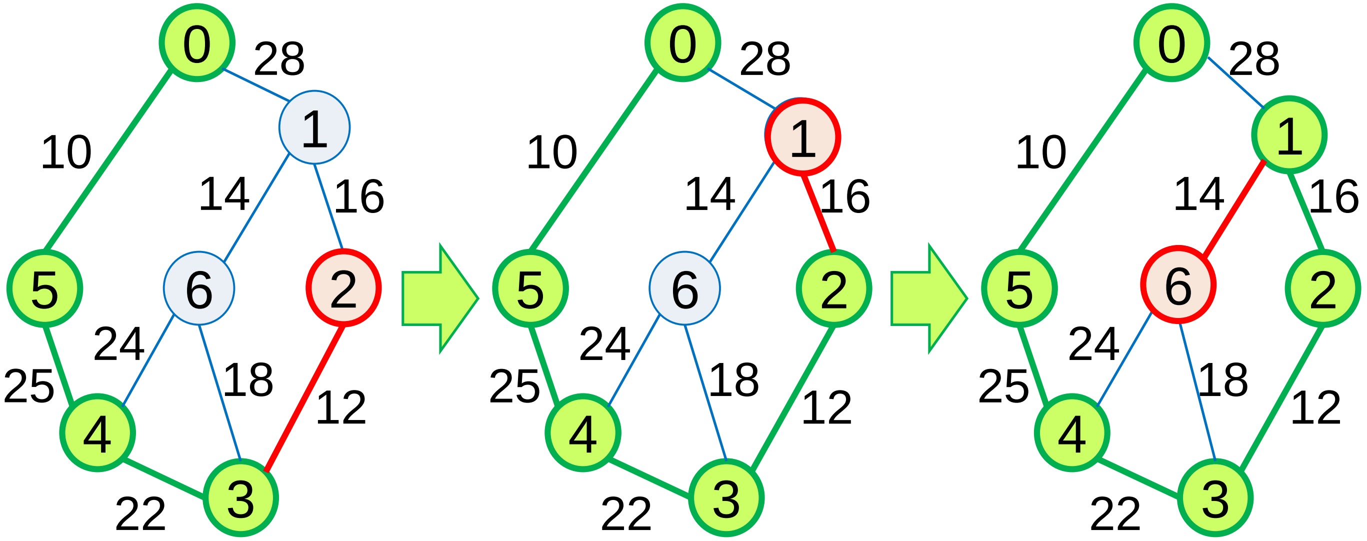


# Prim’s Algorithm (2/2)

```
Prim(G, w, r)
foreach vertex u do
    $key[u] = \infty$; $\pi[u]$ = NIL
$key[r] = 0$ // key: min weight of any edge connecting to a vertex in the tree
$Q = V$ // Q: min-priority queue for vertices not in the tree, based on key[]
while $Q \neq \emptyset$ do
    u = ExtractMin(Q) // also remove u from Q
    foreach vertex v in Adj[u] do
        if v in Q and $w(u, v) < key[v]$ then $\pi[v] = u$; $key[v] = w(u, v)$
        // decrease key for v in Q
```

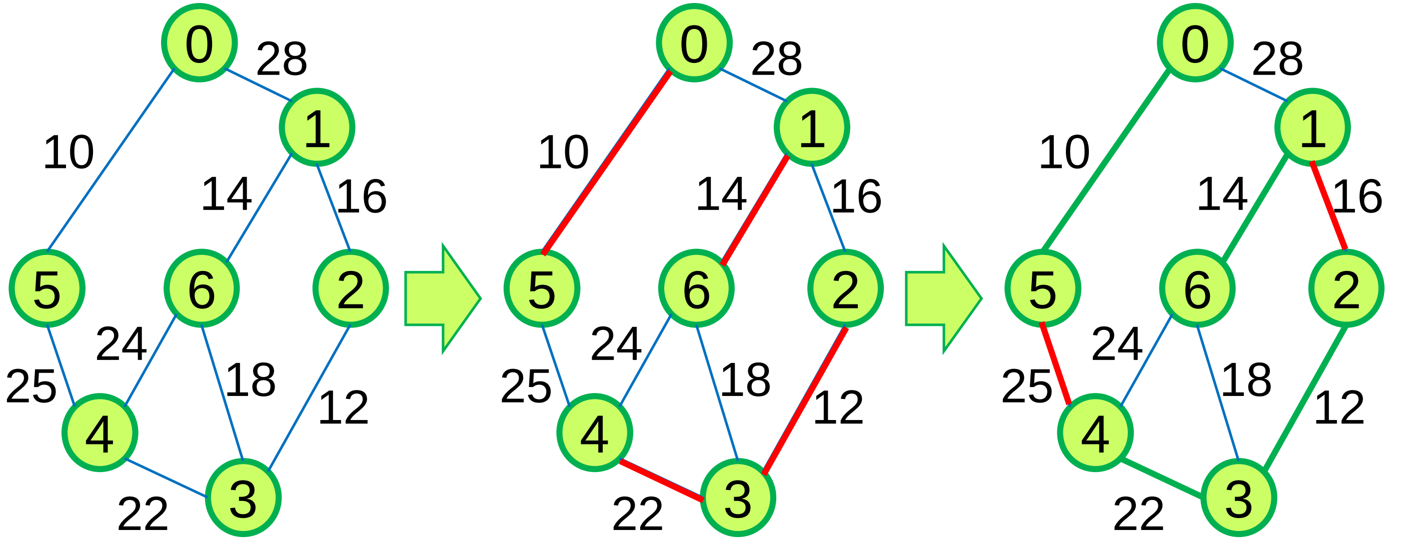
Do not sort *w* directly

- □ **Time complexity: O(*E* lg *V*)**



# Sollin’s Algorithm

1. **Begin with no edges selected**
   - Initially, a forest of *n* trees ⇒ Finally, a tree
2. **Select several edges at a time**
   - Select one min-cost edge for each tree to connect another tree



(0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), (6, 1) are selected

(5, 4), (1, 2), (2, 1) are selected

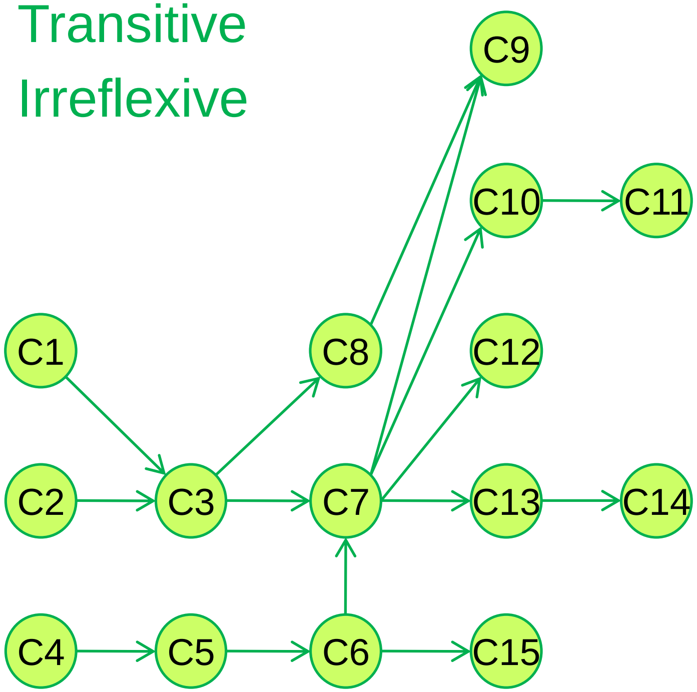
# Topological Sort

## Static timing analysis

## Simulation

# Precedence Relation in Graphs

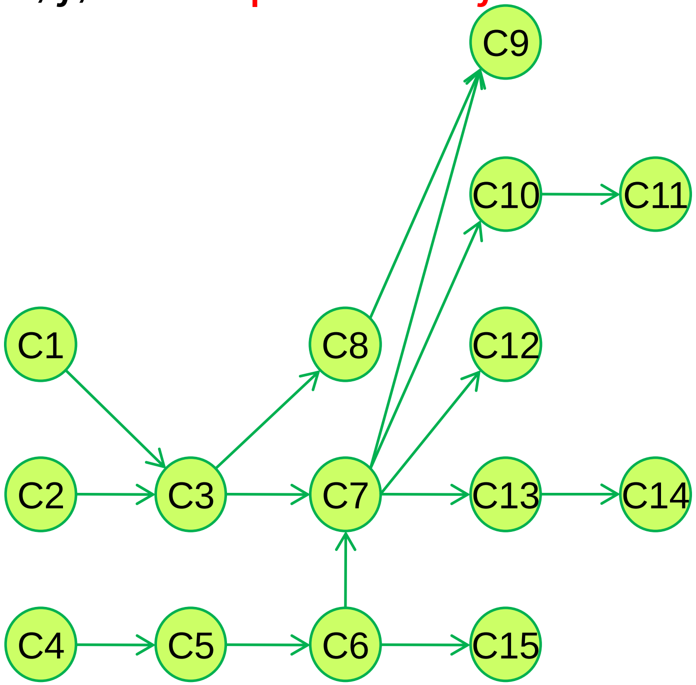
- **DAG: directed acyclic graph**
- **The precedence relation between two vertices**
  - $E(G)$: $<i, j>$: Task $i$ must occur before $j$
  - e.g., course schedule
    - $<i, j> \in E(G)$: $i$ is $j$'s prerequisite
    - Transitive
    - Irreflexive



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

# Topological Order

- **Definition: A topological order is a linear ordering of the vertices of a DAG (directed acyclic graph) *G* s.t.**
  - if $<i, j> \in E(G)$ $\forall i, j$, then *i* precedes *j* in the linear ordering
  - $\Rightarrow$ not unique



C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C13, C12, C14

# Topological Sort

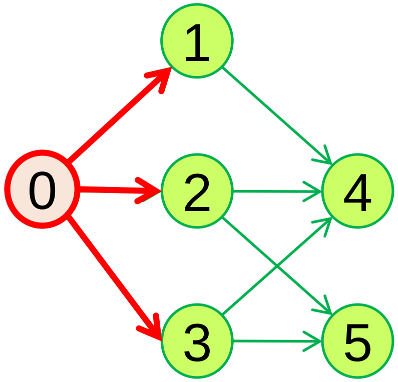
```
1. input an n-vertex digraph;
2. if (every vertex has a positive indegree) return; // is it a DAG?
3. for (int i = 0; i < n; i++) do // output the vertices
4.     pick a vertex v of zero indegree;
5.     cout << v;
6.     delete v and its out-going edges from the graph;
7. }
```

- **Time complexity: O(*n*+*e*)**
  - Adopt adjacency list with an additional "indegree" field
  - Use a queue/stack to keep "free" vertices
  - 1—2: parse the graph once and put free vertices into queue/stack
  - 3—7: pick free vertices from queue/stack one at a time
    - Output it
    - Remove out-going edges
    - Put new free vertices into queue/stack

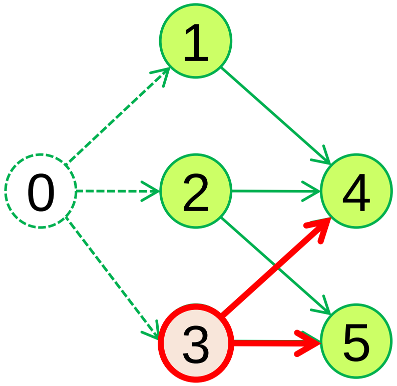
Always can find 0-indegree vertices?

**Implementation:**
In-degree + out-going edges

# Example: Topological Sorting

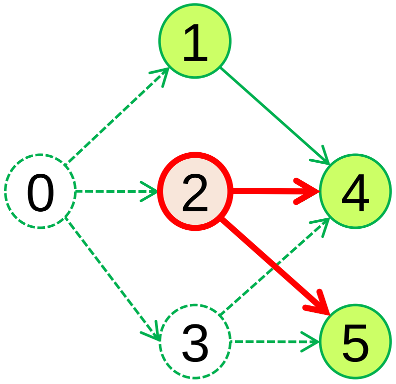


Output: 0



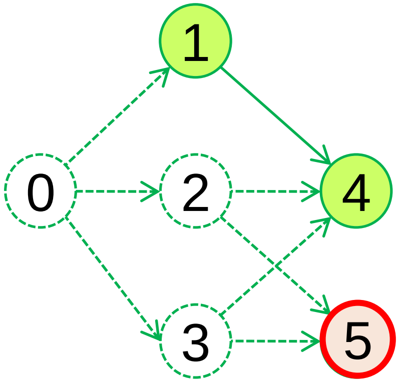
0-indegree: 1, 2, 3

Output: 0, 3



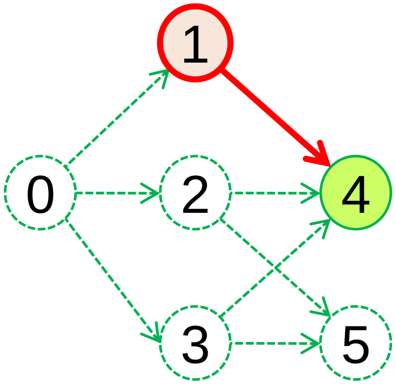
0-indegree: 1, 2

Output: 0, 3, 2



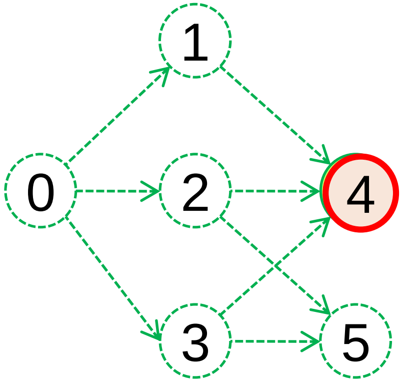
0-indegree: 1, 5

Output: 0, 3, 2, 5



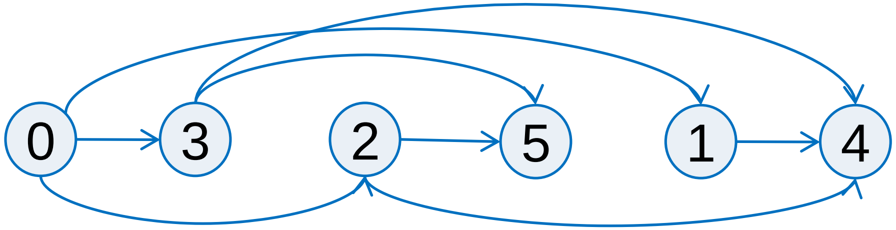
0-indegree: 1

Output: 0, 3, 2, 5, 1



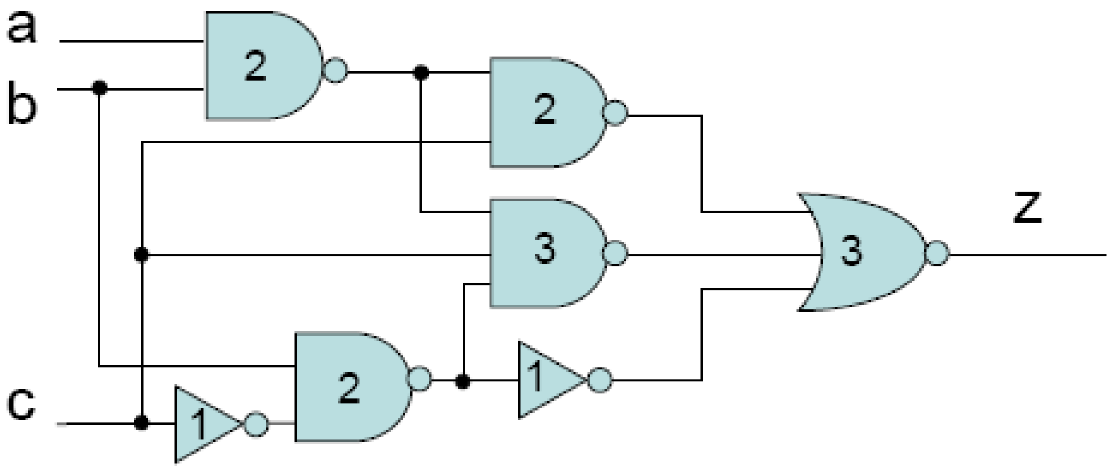
0-indegree: 4

Output: 0, 3, 2, 5, 1, 4



# Static Timing Analysis

- **A combinational logic network:**
  - DAG (directed acyclic graph)
  - Vertex: performs a primitive logic function and has delay
  - Edge: passes a signal and has delay
- **How to compute the critical paths?**
  - A critical path is a path of longest length
  - e.g., for simplicity, edges have no delay



Longest delay? Critical path?

# Definitions on Timing

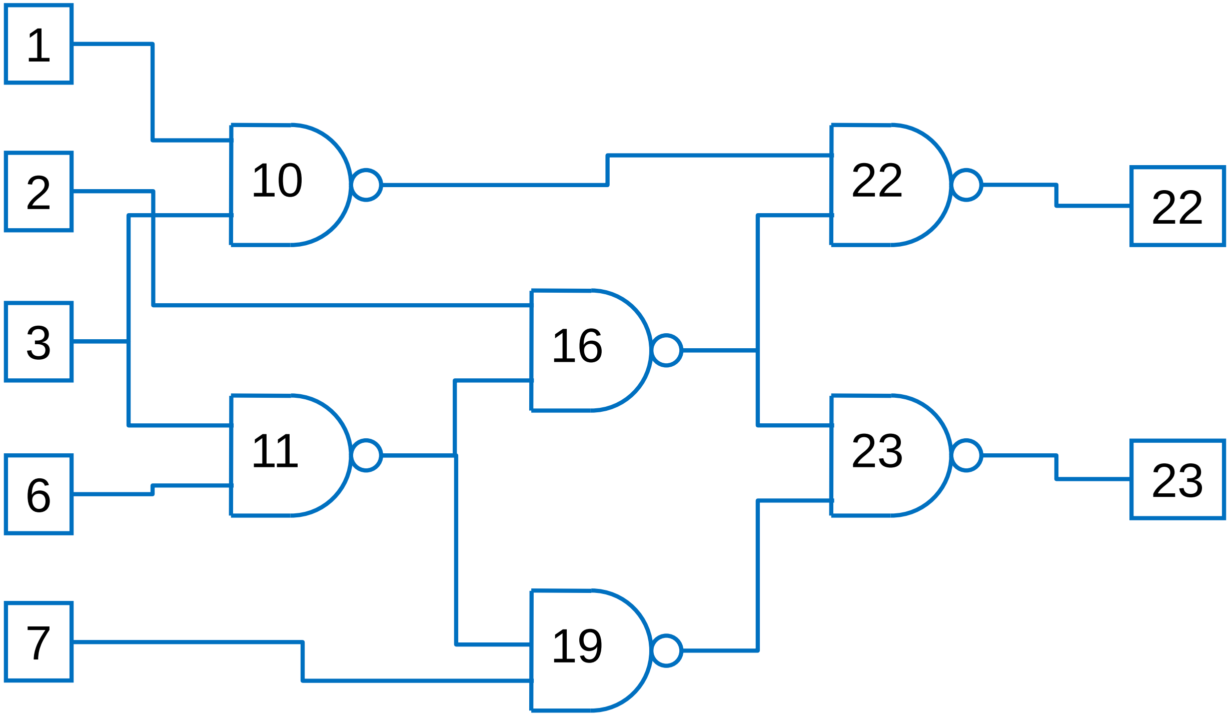
- **Def: A critical path is a path of longest length**
  - Not unique
- **Def: The arrival time of a gate:**
  - Its earliest allowed time to start computing its output
- **Def: The required time of a gate:**
  - Its latest allowed time to generate its output
- **Def: slack = criticality = required time – arrival time**
  - WNS (worst negative slack)
  - TNS (total negative slack)

# Example: C17 Input

- □ **Primary inputs/primary outputs**
- □ **Connection**

```
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

10 is the output of a nand gate whose inputs are 1 and 3

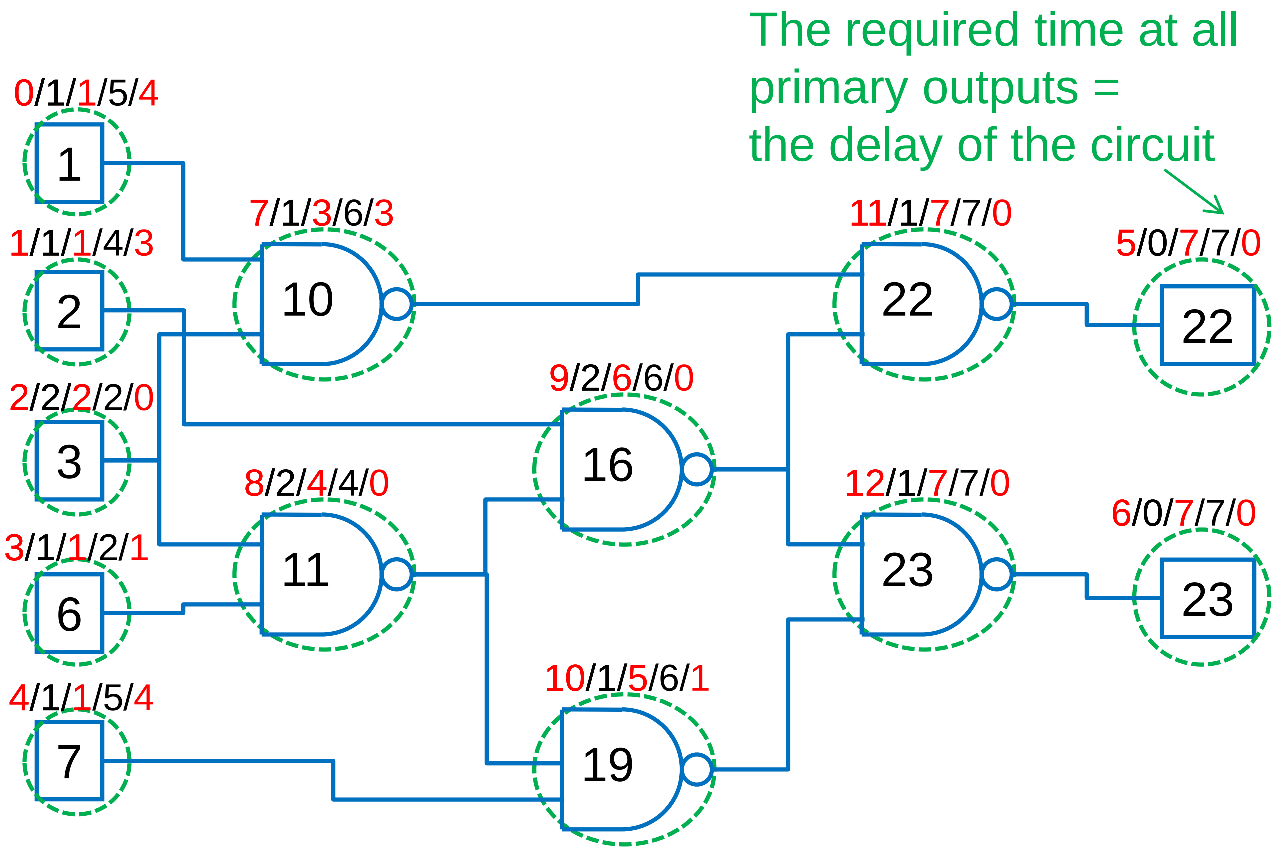


# Example: C17 Graph

- □ **Vertex: gate/input/output: Assume delay = # of fanout gates**
- □ **Edge: wire**

| | |
|---|---|
| 0 | INPUT(1) |
| 1 | INPUT(2) |
| 2 | INPUT(3) |
| 3 | INPUT(6) |
| 4 | INPUT(7) |
| 5 | OUTPUT(22) |
| 6 | OUTPUT(23) |
| 7 | 10 = NAND(1, 3) |
| 8 | 11 = NAND(3, 6) |
| 9 | 16 = NAND(2, 11) |
| 10 | 19 = NAND(11, 7) |
| 11 | 22 = NAND(10, 16) |
| 12 | 23 = NAND(16, 19) |

VertexID



The required time at all primary outputs = the delay of the circuit

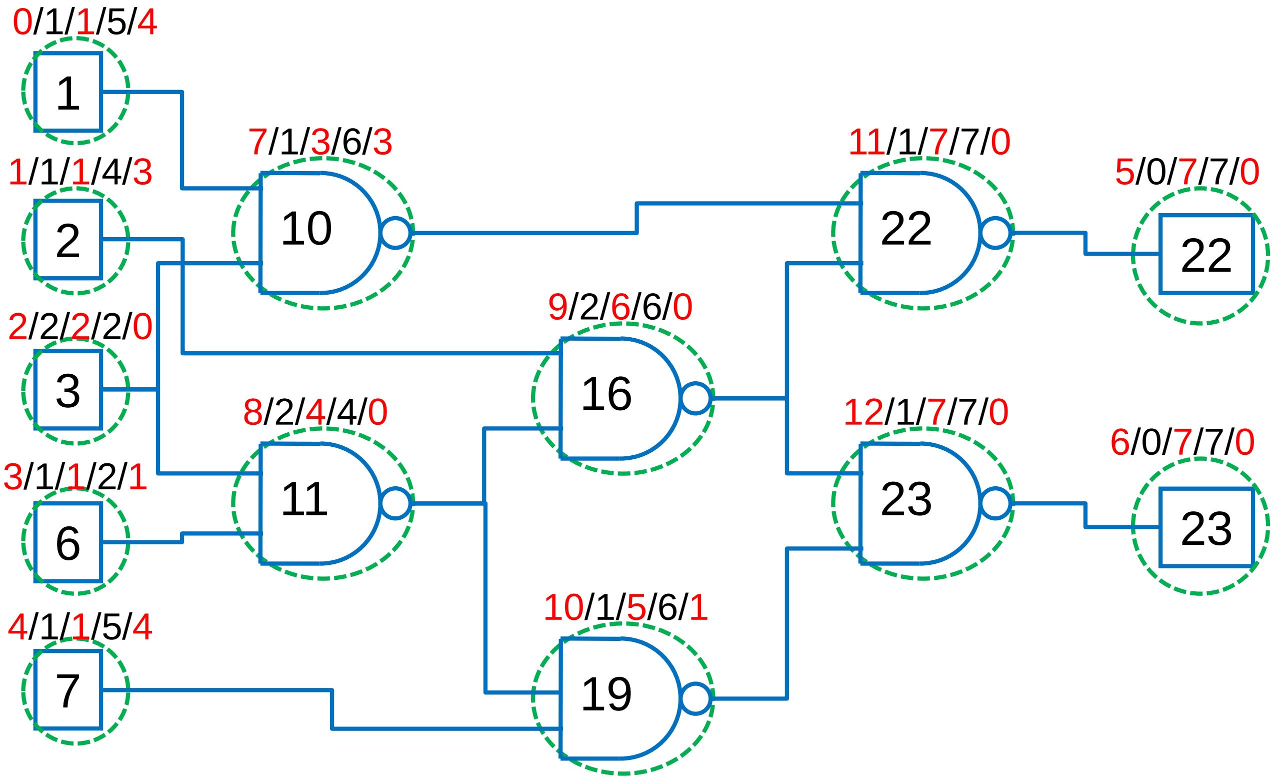
VertexID/delay/arrival/required/slack

# Example: C17 Output

- □ **Critical path delay, input pins, output pins**
- □ **Delay info of each vertex**

```
7
5 0 1 2 3 4
2 5 6
0 1 4
1 1 3
2 2 0
3 1 1
4 1 4
5 7 0
6 7 0
7 3 3
8 4 0
9 6 0
10 5 1
11 7 0
12 7 0
```

VertexID in ascending order

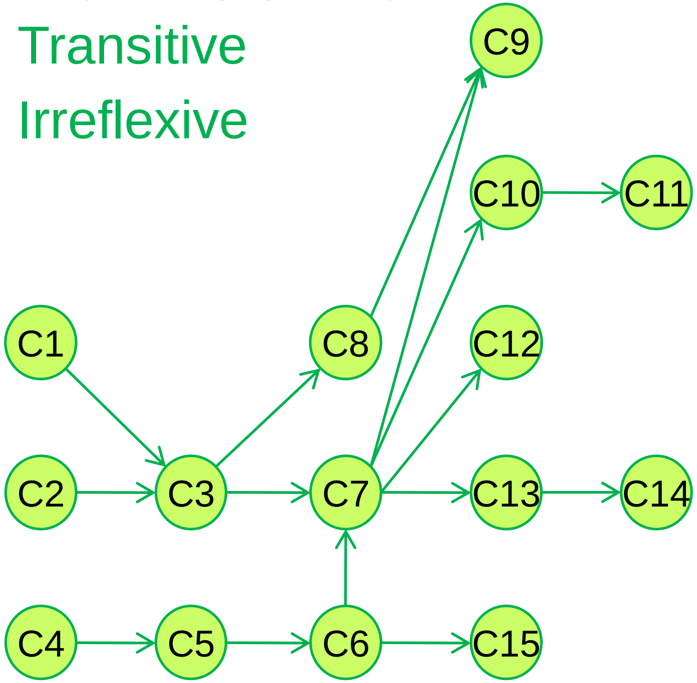


VertexID/delay/arrival/required/slack

# Activity Networks

# Activity-on-Vertex Networks

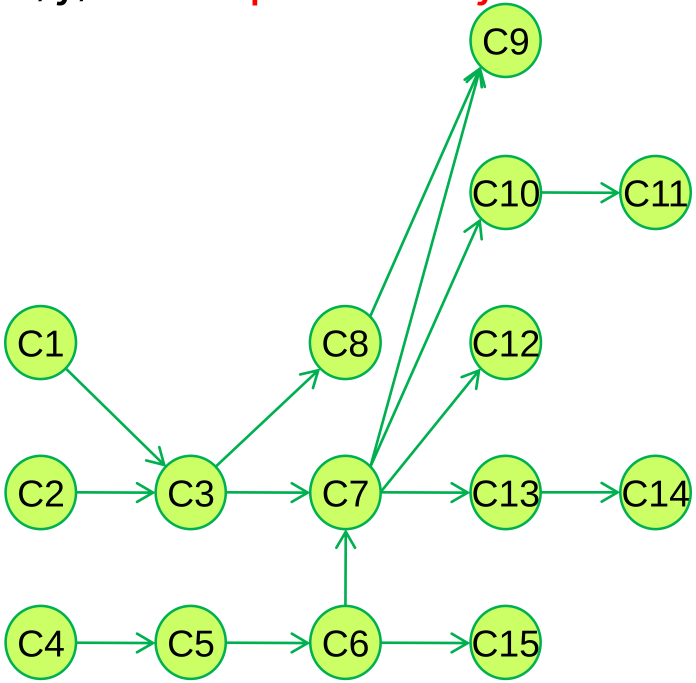
- **Definition: An activity-on-vertex (AOV) network is a digraph *G***
  - *V*(*G*): activities or tasks
  - *E*(*G*): <*i*, *j*>: precedence relation
    - Task *i* must occur before *j*
  - E.g., AOV of courses
    - <*i*, *j*>∈*E*(*G*): *i* is *j*'s prerequisite
    - Transitive
    - Irreflexive



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

# Topological Order

- **Definition: A topological order is a linear ordering of the vertices of a DAG (directed acyclic graph) *G* s.t.**
  - if *<i, j>*∈*E*(*G*) ∀*i*, *j*, then *i* precedes *j* in the linear ordering
  - ⇒ not unique



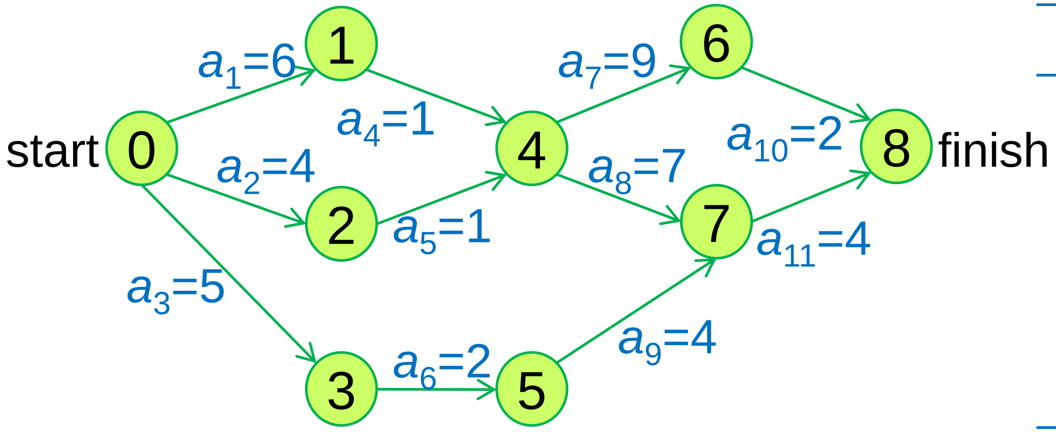
C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C13, C12, C14

# Activity-on-Edge Networks

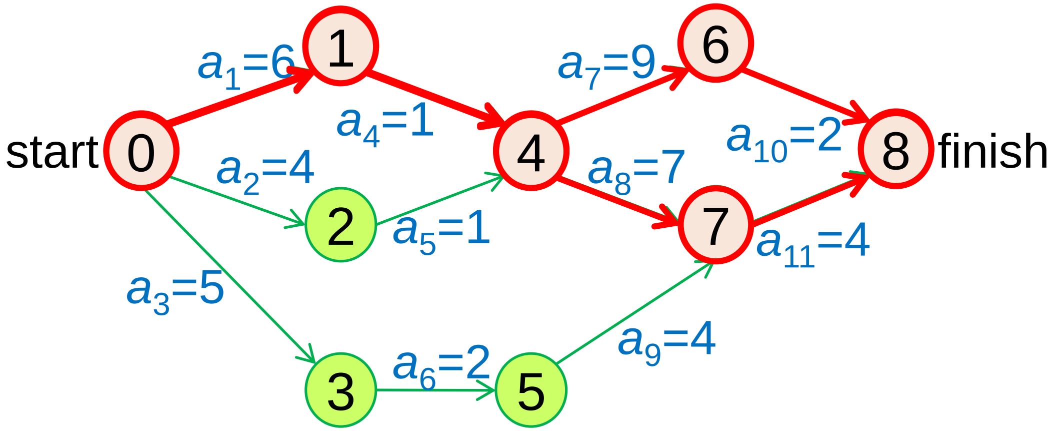
- **Definition: An activity-on-edge (AOE) network is a digraph *G***
  - *V*(*G*): events: signal the completion of certain activities
  - *E*(*G*): activities: launch if the event on “from” vertex has occurred
    - Edge weight: the time needed to perform the activity
  - E.g., AOE of projects
    - How fast can the project be done?
    - What are the bottlenecks?
    - How to speedup the project?



| event | interpretation |
|---|---|
| 0 | Start of project |
| 1 | Completion of activity $a_1$ |
| 4 | Completion of activities $a_4$ and $a_5$ |
| 7 | Completion of activities $a_8$ and $a_9$ |
| 8 | Completion of project |

# Critical Paths

- **A critical path is a path of longest length**
  - Not unique, e.g, *length*(0, 1, 4, 6, 8)=18=*length*(0, 1, 4, 7, 8)=18



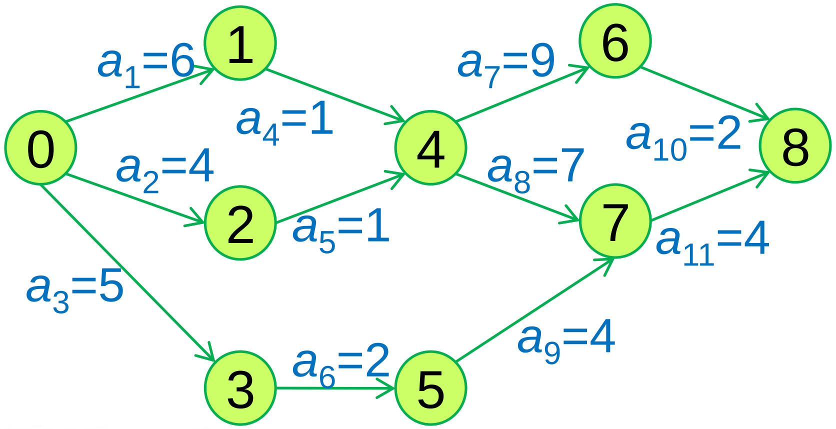
- **Earliest time *e*(*i*) of activity $a_i$:**
  - its earliest allowed start time
- **Latest time *l*(*i*) of activity $a_i$:**
  - its latest start time without increasing the project duration
- **Criticality = *l*(*i*)-*e*(*i*) = slack**
- **Critical activity: *e*(*i*) = *l*(*i*)**

# Earliest Time

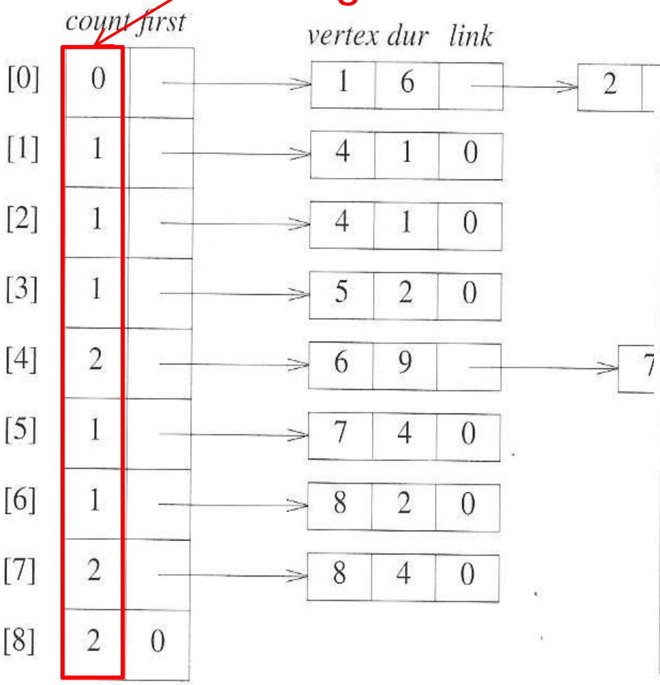
- **For an activity $a_i$ on $<k, l>$**
  - $e(i) = ee[k] = \max_{<j,\, k> \in E} \{ee[j] + \text{duration\_on\_}<j, k>\}$

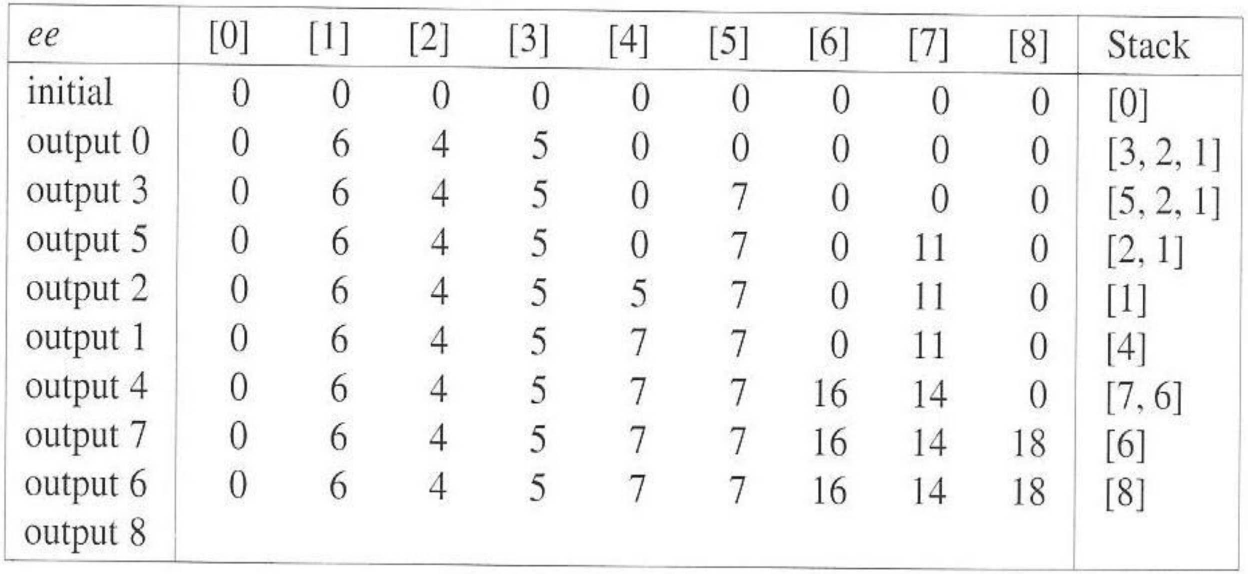
earliest event time

- **How?**
  - In topological order
  - Adjacency list with in-degree



In-degree





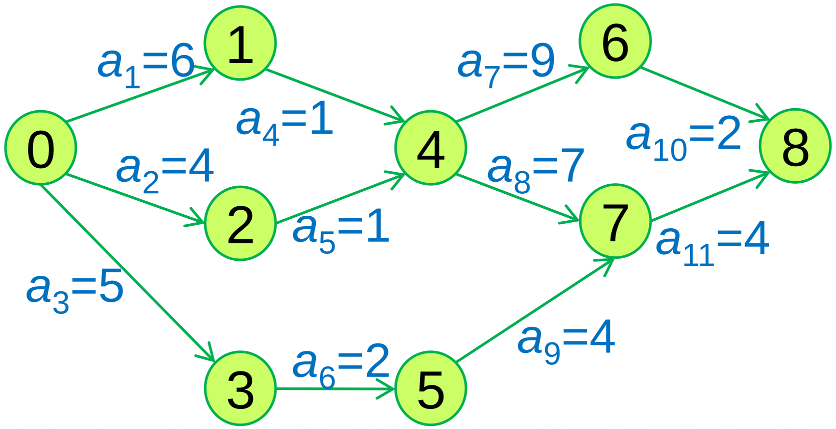
| ee | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | Stack |
|---|---|---|---|---|---|---|---|---|---|---|
| initial | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [0] |
| output 0 | 0 | 6 | 4 | 5 | 0 | 0 | 0 | 0 | 0 | [3, 2, 1] |
| output 3 | 0 | 6 | 4 | 5 | 0 | 7 | 0 | 0 | 0 | [5, 2, 1] |
| output 5 | 0 | 6 | 4 | 5 | 0 | 7 | 0 | 11 | 0 | [2, 1] |
| output 2 | 0 | 6 | 4 | 5 | 5 | 7 | 0 | 11 | 0 | [1] |
| output 1 | 0 | 6 | 4 | 5 | 7 | 7 | 0 | 11 | 0 | [4] |
| output 4 | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 0 | [7, 6] |
| output 7 | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 18 | [6] |
| output 6 | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 18 | [8] |
| output 8 | | | | | | | | | | |

# Latest Time

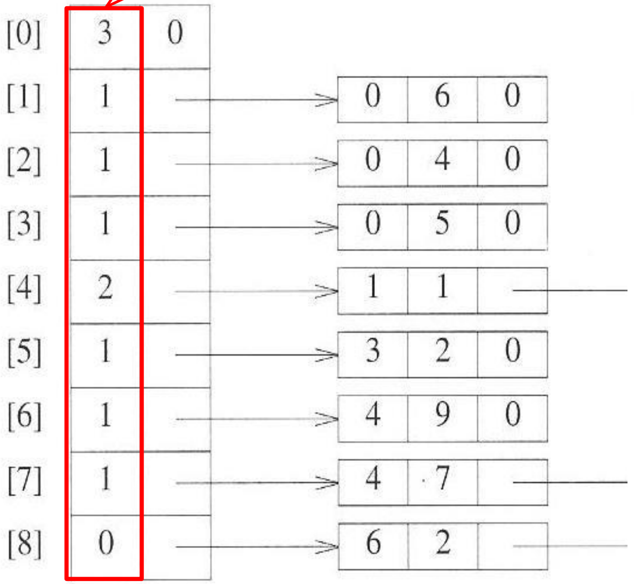
- **For an activity $a_i$ on $<k, l>$**
  - $l(i) = le[l] - \text{duration\_of\_}a_i = \min_{<l,\, j> \in E} \{le[j] - \text{duration\_on\_}<l, j>\}$
    $- \text{duration\_of\_}a_i$

latest event time

- **How?**
  - In reverse topological order
  - Inverse adjacency list
    with out-degree



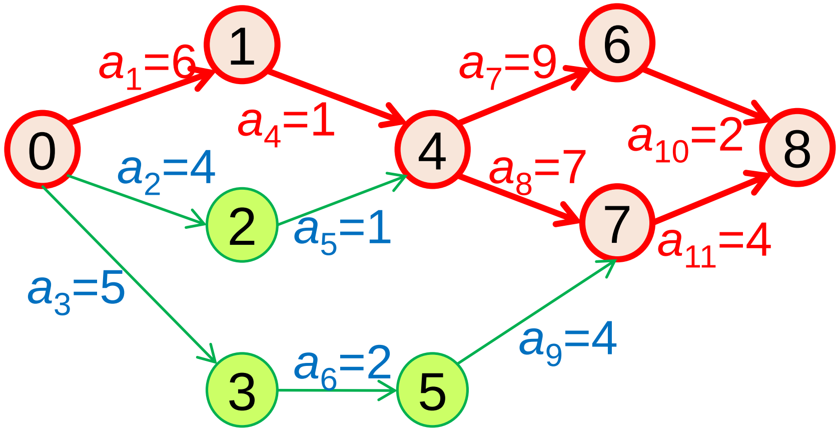
out-degree

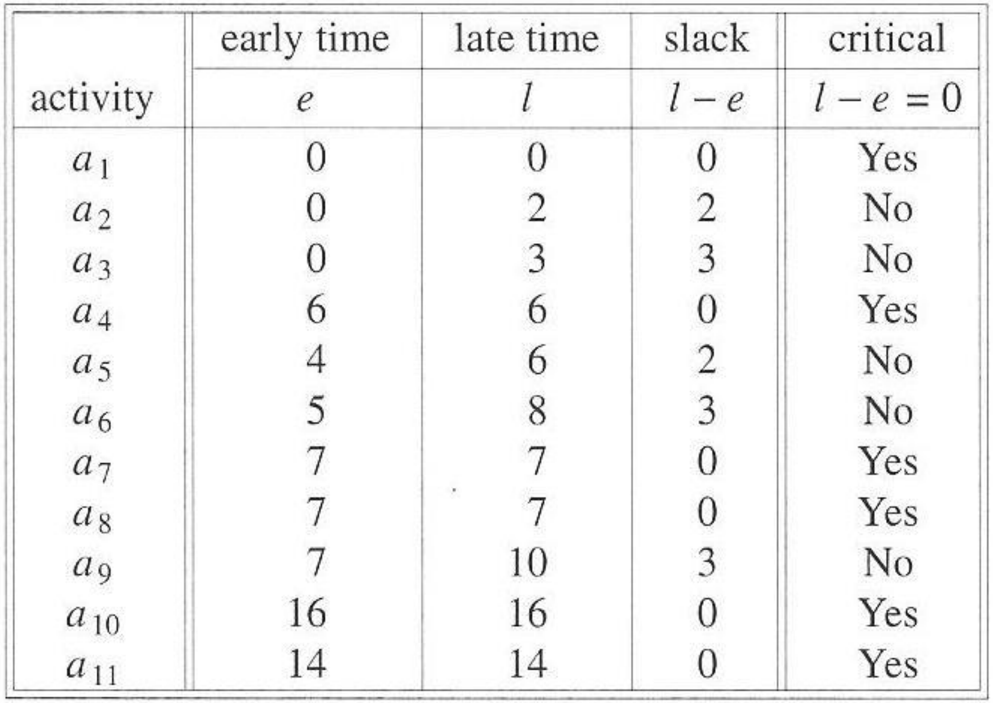


| *le* | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | Stack |
|---|---|---|---|---|---|---|---|---|---|---|
| initial | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | [8] |
| output 8 | 18 | 18 | 18 | 18 | 18 | 18 | 16 | 14 | 18 | [7,6] |
| output 7 | 18 | 18 | 18 | 18 | 7 | 10 | 16 | 14 | 18 | [5,6] |
| output 5 | 18 | 18 | 18 | 8 | 7 | 10 | 16 | 14 | 18 | [3,6] |
| output 3 | 3 | 18 | 18 | 8 | 7 | 10 | 16 | 14 | 18 | [6] |
| output 6 | 3 | 18 | 18 | 8 | 7 | 10 | 16 | 14 | 18 | [4] |
| output 4 | 3 | 6 | 6 | 8 | 7 | 10 | 16 | 14 | 18 | [2,1] |
| output 2 | 2 | 6 | 6 | 8 | 7 | 10 | 16 | 14 | 18 | [1] |
| output 1 | 0 | 6 | 6 | 8 | 7 | 10 | 16 | 14 | 18 | [0] |

# Summary on Example AOE

- **How fast can the project be done?**
  - 18
- **What are the bottlenecks?**
  - $a_1$, $a_4$, $a_7$, $a_8$, $a_{10}$, $a_{11}$
- **How to speedup the project?**
  - Reduce durations for $a_1$, $a_4$, $a_7$, $a_8$, $a_{10}$, $a_{11}$





| activity | early time | late time | slack | critical |
|---|---|---|---|---|
| | $e$ | $l$ | $l - e$ | $l - e = 0$ |
| $a_1$ | 0 | 0 | 0 | Yes |
| $a_2$ | 0 | 2 | 2 | No |
| $a_3$ | 0 | 3 | 3 | No |
| $a_4$ | 6 | 6 | 0 | Yes |
| $a_5$ | 4 | 6 | 2 | No |
| $a_6$ | 5 | 8 | 3 | No |
| $a_7$ | 7 | 7 | 0 | Yes |
| $a_8$ | 7 | 7 | 0 | Yes |
| $a_9$ | 7 | 10 | 3 | No |
| $a_{10}$ | 16 | 16 | 0 | Yes |
| $a_{11}$ | 14 | 14 | 0 | Yes |

# Shortest Paths and Transitive Closure

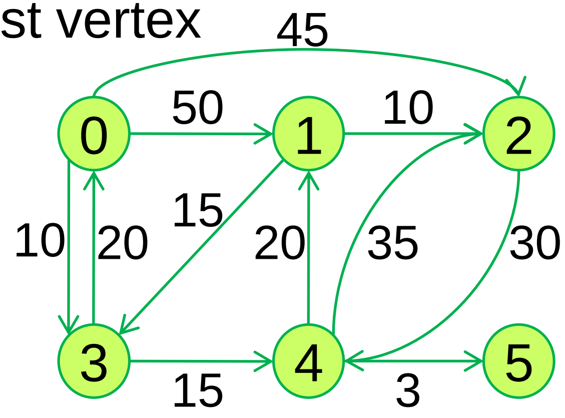
**Single source/all destinations**

**All pairs shortest paths**

**Transitive closure**

# Shortest Paths

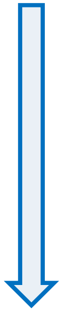
- **Given a digraph *G* and vertices *A* and *B* in *G***
  - Is there a path from *A* to *B*?
  - If there is more than one path from *A* to *B*, which is shortest?
- **The length of a path ≡ the sum of the lengths of the edges on it**
  - Length: cost, weight
  - What if unweighted? The # of edges, i.e., weight = 1
    - Q: How can you find the shortest path for an nonweighted graph?
- **Variants**
  - Source: starting vertex; destination: last vertex
  - Single source single destination
  - Single source all destinations
  - All sources single destination
  - All-pairs



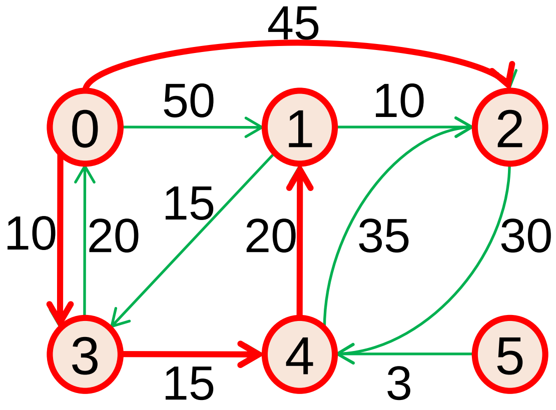
# Single Source/All Destinations
## -- Nonnegative edge weights

- Start from a vertex, say 0
- Choose one vertex at a time
- Calculate its path length through only chosen vertices

| Path | Length |
|---|---|
| 0, 3 | 10 |
| 0, 3, 4 | 25 |
| 0, 3, 4, 1 | 45 |
| 0, 2 | 45 |
| 0, 5 | - |



Nondecreasing

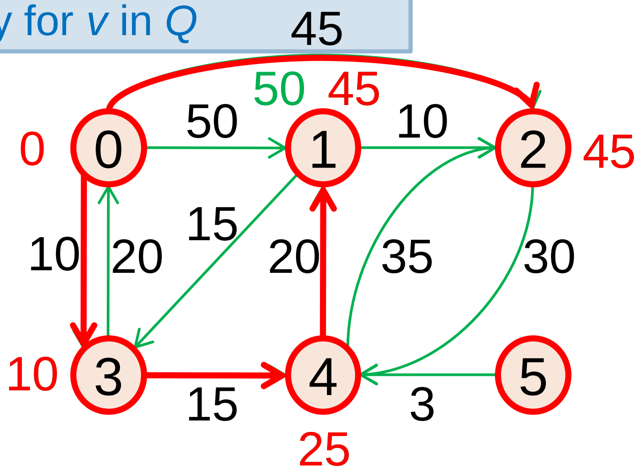


Output: Shortest path tree

# Dijkstra's Algorithm

- **Allow only nonnegative edge weights**
- **Greedy! Cf. Prim's algorithm and BFS**
- **Time complexity: O(*E* lg *V*) using min-heap**

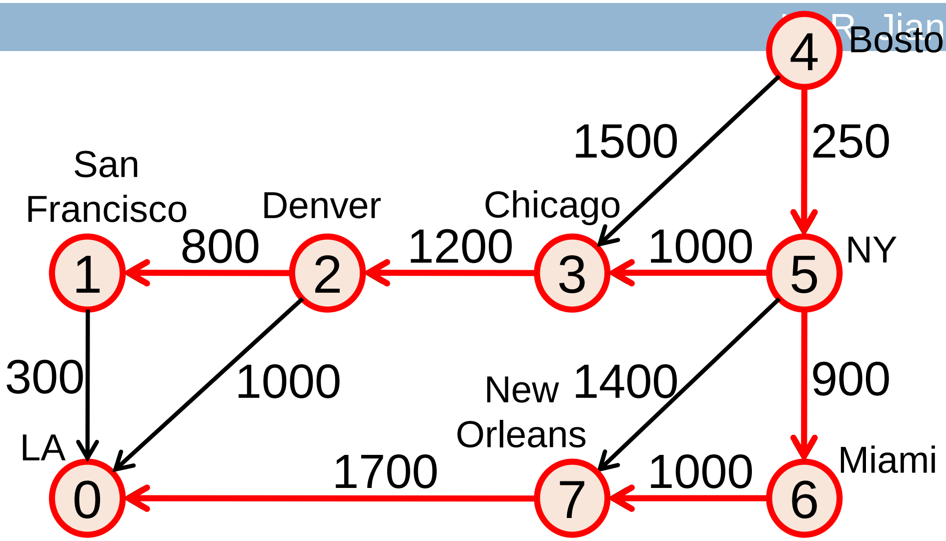
```
Dijkstra(G, w, v) // source: v
foreach vertex u do
    $\delta[u] = \infty$; $\pi[u]$ = NIL
$\delta[v] = 0$ // $\delta$: shortest path length estimate so far
$Q = V$ // Q: min-priority queue for vertices not in the tree, based on $\delta[]$
while $Q \neq \emptyset$ do
    u = ExtractMin(Q)  // also remove u from Q
    foreach vertex v in Adj[u] do
        if v in Q and $\delta[u]$ + length[u][v] < $\delta[v]$ then // shorter path found
            $\pi[v] = u$; $\delta[v] = \delta[u]$ + length[u][v] // decrease key for v in Q
```



# Example: Dijkstra’s Algorithm

## -- Nonnegative edge weights

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & \infty & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & \infty & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$



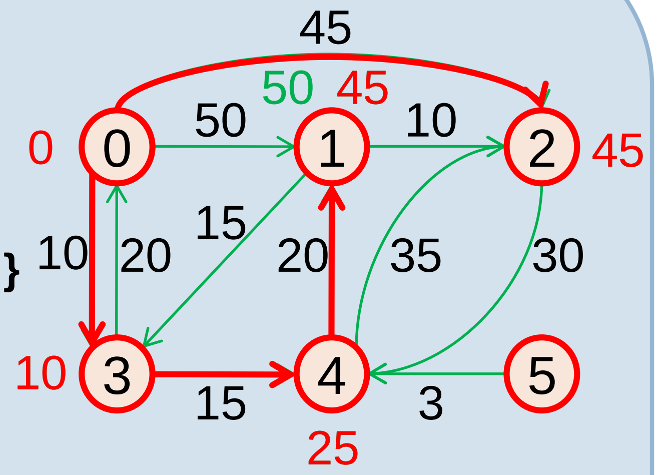
| Iteration | *u* | δ[] | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial | 4 | +∞ | +∞ | +∞ | 1500 | 0 | 250 | +∞ | +∞ |
| 1 | 5 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 2 | 6 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 3 | 3 | +∞ | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 4 | 7 | 3350 | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 5 | 2 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 6 | 1 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| end | 0 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |

# Dijkstra’s Algorithm

## -- Textbook version

- **Greedy! Cf. Prim’s algorithm and BFS**
- **Time complexity: O($n^2$) using Adjacency matrix**

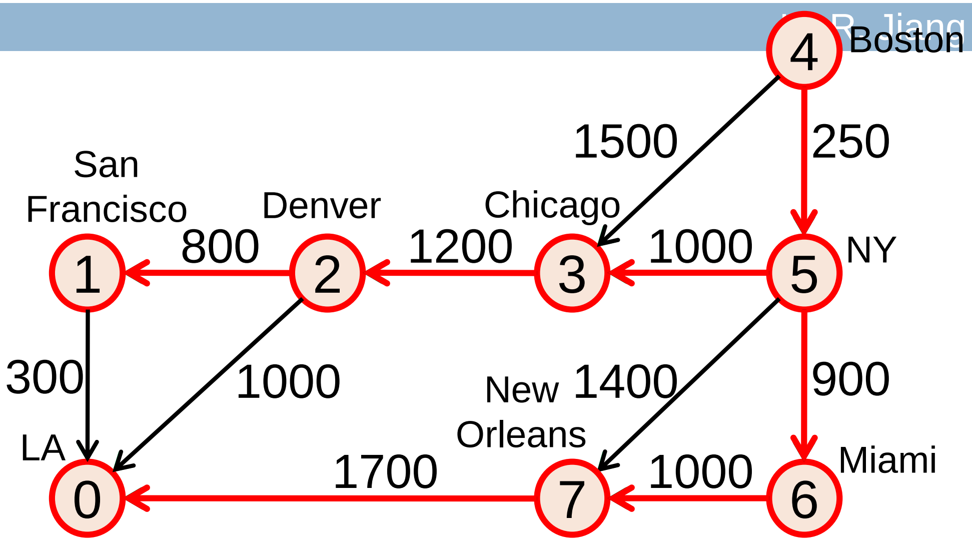
```
void Graph::ShortestPath(const int n, const int v) { // vertices: 0..n-1; source: v
// length[i][j]: edge length from i to j; +∞ for non-edge
// dist[j]: shortest path length from v to j found so far
// s[i]: shortest path length determined yet?

// initialize
    for(int i = 0; i < n; i++) {s[i] = false; dist[i] = length[v][i];}
    s[v] = true; dist[v] = 0;

// determine n-1 paths from v
    for(int i = 0; i < n-2; i++) {
        int u = Choose(n); // Choose finds u with min dist[] and false s[] (use min-heap)
        s[u] = true;
        for(int w = 0; w < n; w++) // update u’s neighbor w  where s[w] is false
            if ((! s[w]) && (dist[u] + length[u][w] < dist[w]))
                dist[w] = dist[u] + length[u][w]; // shorter path found
    }
}
```



# Example: Dijkstra’s Algorithm

## -- Textbook version

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & \infty & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & \infty & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$



| Iteration | s[]=true | *u* | ***dist*[]** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial | - | - | +∞ | +∞ | +∞ | 1500 | 0 | 250 | +∞ | +∞ |
| 1 | {4} | 5 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 2 | {4, 5} | 6 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 3 | {4, 5, 6} | 3 | +∞ | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 4 | {4, 5, 6, 3} | 7 | 3350 | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 5 | {4, 5, 6, 3, 7} | 2 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 6 | {4, 5, 6, 3, 7, 2} | 1 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| end | {4, 5, 6, 3, 7, 2, 1} | | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |

# Prim’s vs. Dijkstra’s Algorithms

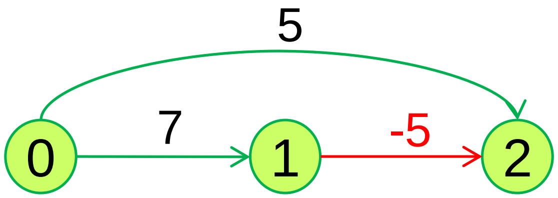
```
Prim(G, w, r)
foreach vertex u do
    key[u] = ∞; π[u] = NIL
key[r] = 0 // key: min weight of any edge connecting to a vertex in the tree
Q = V // Q: min-priority queue for vertices not in the tree, based on key[]
while Q ≠ ∅ do
    u = ExtractMin(Q)
    foreach vertex v in Adj[u] do
        if v in Q and w(u, v) < key[v] then // update key
            π[v] = u; key[v] = w(u, v)
```

```
Dijkstra(G, w, v) // source: v
foreach vertex u do
    δ[u] = ∞; π[u] = NIL
δ[v] = 0 // δ: shortest path length estimate so far
Q = V // Q: min-priority queue for vertices not in the tree, based on δ[]
while Q ≠ ∅ do
    u = ExtractMin(Q)
    foreach vertex v in Adj[u] do
        if v in Q and δ[u] + length[u][v] < δ[v] then // shorter path found
            π[v] = u; δ[v] = δ[u] + length[u][v]
```

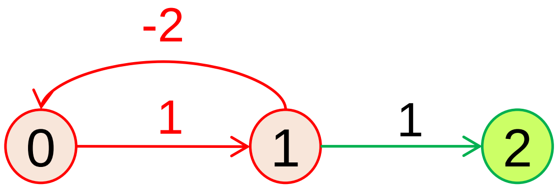
# Single Source/All Destinations

## -- General weights

- **Allow negative edges**



- **Disallow negative cycles**



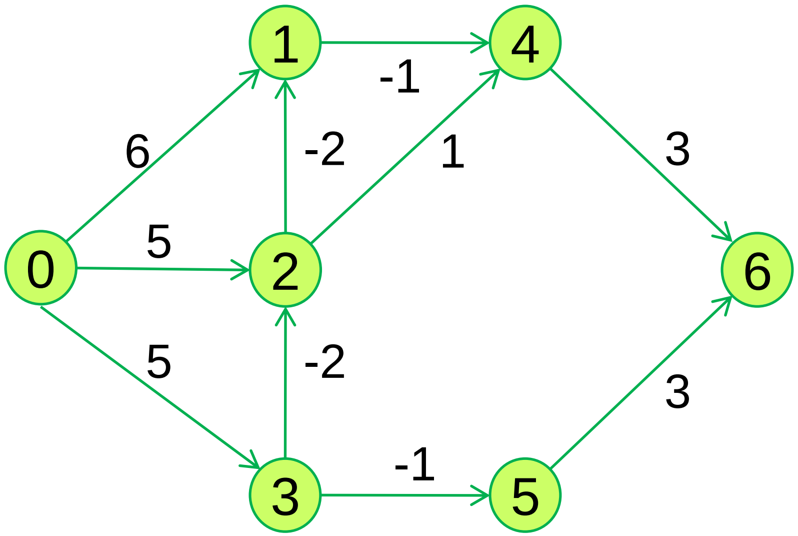
# Bellman and Ford Algorithm

- **If no negative cycles, a shortest path has at most *n*-1 edges**
  - Idea: induction on *k* (# of edges)
- ***dist*$^k$[*u*]: the shortest path length from source *v* to *u* using at most *k* edges**
  - $dist^k[u] = \min_i \{dist^{k-1}[u], \min\{dist^{k-1}[i] + length[i][u]\}\}$
- **Time complexity: O($n^3$)**

```
void Graph::BellmanFord(const int n, const int v) { // vertices: 0..n-1; source: v
// length[i][j]: edge length from i to j; +∞ for non-edge
// dist[i]: shortest path length from v to i using k edges

// initialize $dist^1$
    for(int i = 0; i < n; i++) dist[i] = length[v][i];
// compute $dist^k$
    for(int k = 2; k <= n-1; k++)
        for(each u s.t. u != v and u has at least one incoming edge)
            for(each incoming edge <i, u>)
                if(dist[u] > (dist[i] + length[i][u])) //update dist
                    dist[u] = dist[i] + length[i][u];
}
```

# Example: Bellman and Ford Algorithm

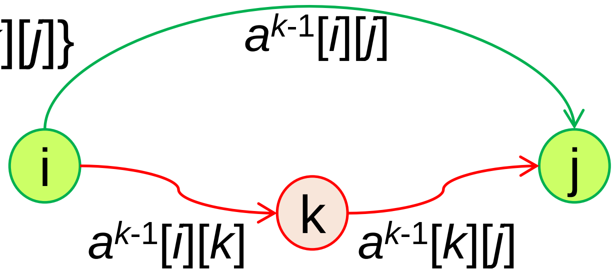
## -- General weights



| *k* | $dist^k$[0..6] | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | 6 | 5 | 5 | +∞ | +∞ | +∞ |
| 2 | 0 | 3 | 3 | 5 | 5 | 4 | +∞ |
| 3 | 0 | 1 | 3 | 5 | 2 | 4 | 7 |
| 4 | 0 | 1 | 3 | 5 | 0 | 4 | 5 |
| 5 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |
| 6 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |

# All-Pairs Shortest Paths

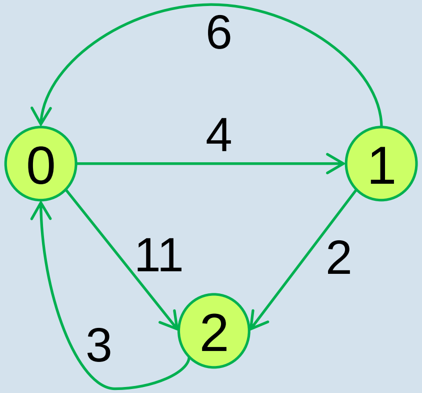
- **Run *n* independent single-source/all-destinations problems?**
  - Each iteration, choose a different vertex as source
- **Better idea?**
- **Define $a^k[i][j]$ as the shortest path length from *i* to *j* going through only vertices of index <= *k***
  - $\Rightarrow$ Goal: find $a^{n-1}[i][j]$ (vertices: $0..n-1$)
  - $a^{-1}[i][j] = length[i][j]$
  - $a^k[i][j] = \min\{a^{k-1}[i][j], a^{k-1}[i][k] + a^{k-1}[k][j]\}$



# All-Pairs Shortest Path Algorithm

```
void Graph::AllLengths(const int n) { // vertices: 0..n-1
// length[i][j]: edge length from i to j; +∞ for non-edge
// a^k[i][j]: shortest path length from i to j using only vertices of index <= k

// initialize a^-1
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            a[i][j] = length[i][j];
// compute a^k[i][j]
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                if(a[i][j] > (a[i][k] + a[k][j])) //update a[i][j] using k
                    a[i][j] = a[i][k] + a[k][j];
}
```

Time complexity: $O(n^3)$



| $a^{-1}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 11 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | +∞ | 0 |

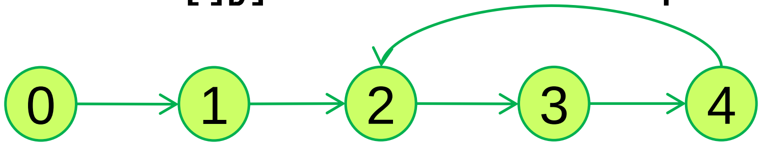
| $a^{0}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 11 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | 7 | 0 |

| $a^{1}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 6 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | 7 | 0 |

| $a^{2}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 6 |
| 1 | 5 | 0 | 2 |
| 2 | 3 | 7 | 0 |

# Transitive Closure

- **Given a digraph *G* with unweighted edges, determine if there is a path from vertex *i* to vertex *j***
  - Positive length: transitive closure (exclude itself)
    - *A*+[*i*][*j*] = 1 iff there is a path of length > 0 from *i* to *j*
  - Non-negative length: reflexive transitive closure (include iteself)
    - *A**[*i*][*j*] = 1 iff there is a path of length ≥ 0 from *i* to *j*



$$
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 & 0
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 & 1 \\
3 & 0 & 0 & 1 & 1 & 1 \\
4 & 0 & 0 & 1 & 1 & 1
\end{array}
\qquad
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 & 1 \\
3 & 0 & 0 & 1 & 1 & 1 \\
4 & 0 & 0 & 1 & 1 & 1
\end{array}
$$

Adjacency matrix

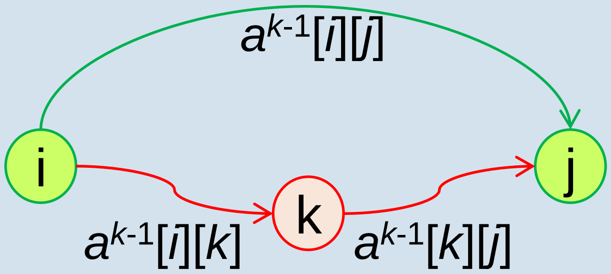
$A^+$

$A^*$

# How?

- **Modify all-pairs shortest paths**
  - *length*[][]: length to adjacency
  - *a*[*i*][*j*]: update like answering yes/no

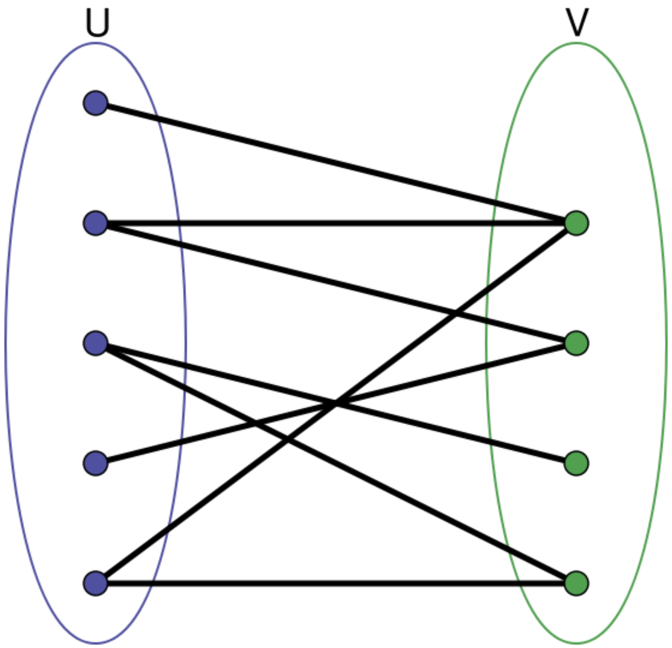
```
void Graph::TransitiveClosure(const int n) { // vertices: 0..n-1
// length[i][j]: adjacency between i and j
// $a^{k}$[i][j]: has path from i to j using only vertices of index <= k

// initialize $a^{-1}$
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            a[i][j] = length[i][j];
// compute $a^{k}$[i][j]
    for(int k = 0; k <= n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                a[i][j] = a[i][j] || (a[i][k] && a[k][j]);
}
```

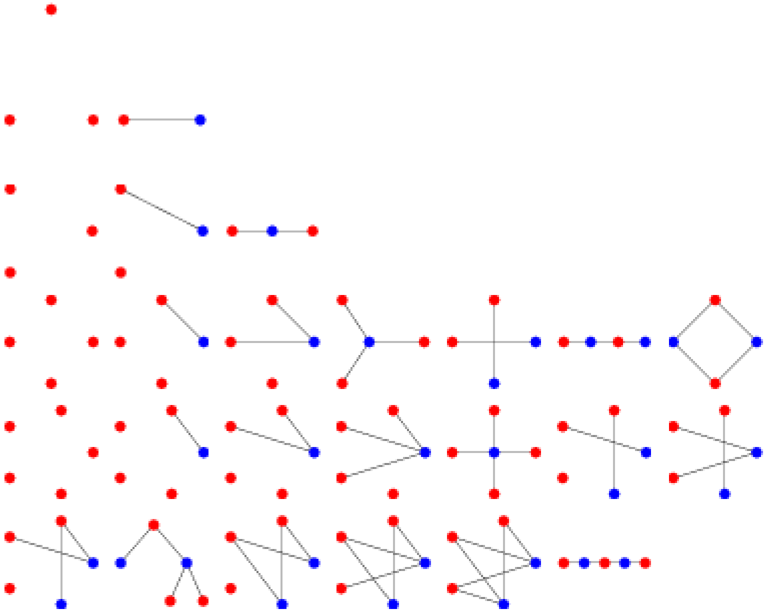


# Appendix: Bipartite Graphs

# Bipartite Graphs

- **A bipartite graph (bigraph) is a graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent**
  - Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles

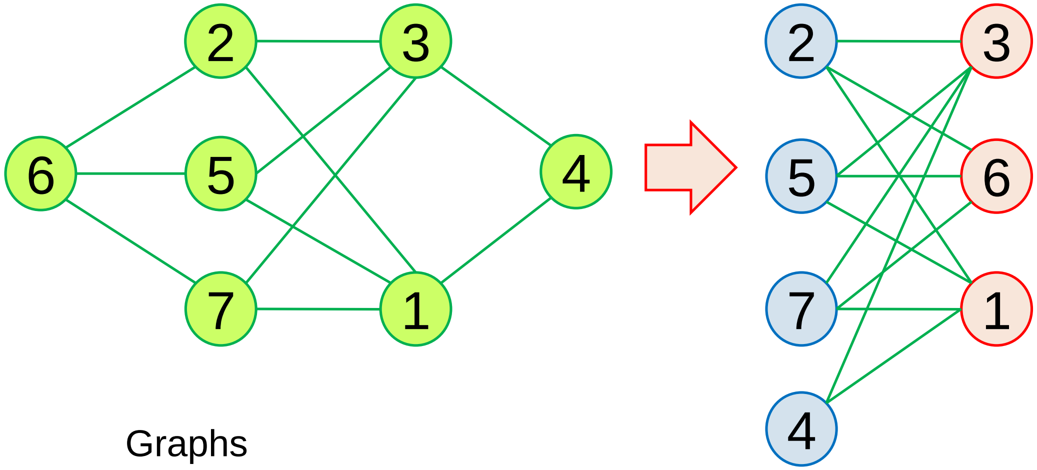




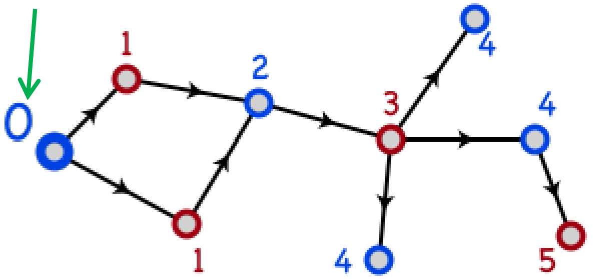
http://mathworld.wolfram.com/BipartiteGraph.html

# Testing Bipartiteness

- **For a connected bigraph, its bipartition can be defined by the parity of the distances from any arbitrarily chosen vertex *v***
  - One subset consists of the vertices at even distance to *v*
  - The other subset consists of the vertices at odd distance to *v*
- **How?**
  1. Encode the distance (parity) to each vertex
  2. Examine each edge to verify if its endpoints located at different subsets
- **Q: How to encode the distance?**

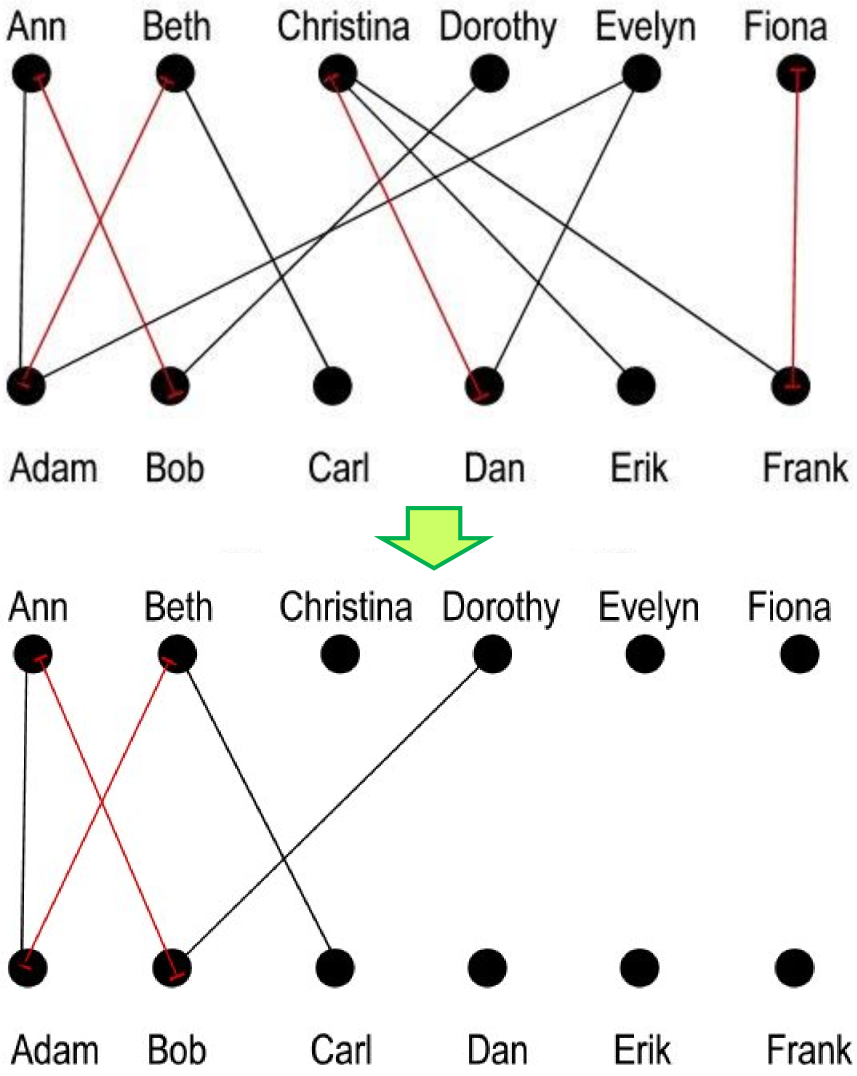


Arbitrarily chosen vertex



# The Marriage Problem and Matching

- **Maximum cardinality matching**
  - Perfect matching
    - Every one gets married
      - Ms./Mr. right? No!



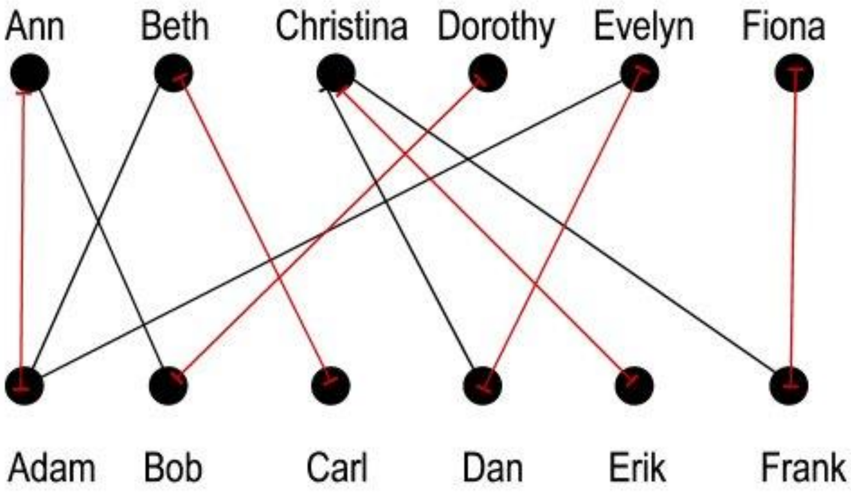


Figure 2 A perfect matching

# The Stable Matching

- **Perfect or stable?**
- **Perfect matching: everyone gets married**

- **Stable matching: perfect matching with no unstable pairs**
  - Unmatched pair m-w is unstable if
    man m and woman w prefer each other to current partners
- **The stable marriage problem:**
  - (Input) Given the preference lists of n men and n women,
  - (Output) find a stable matching if one exists

- **Propose-and-reject algorithm [Gale-Shapley 1962]**
  - Intuitive method that guarantees to find a stable matching
  - Male-optimal