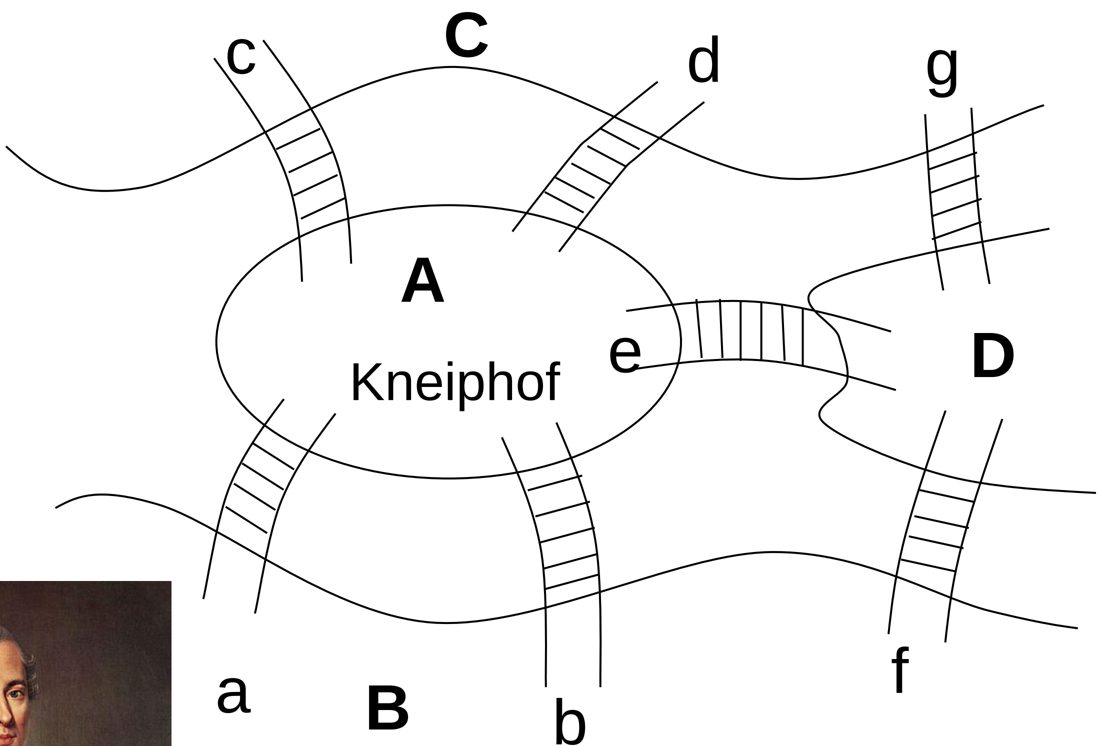
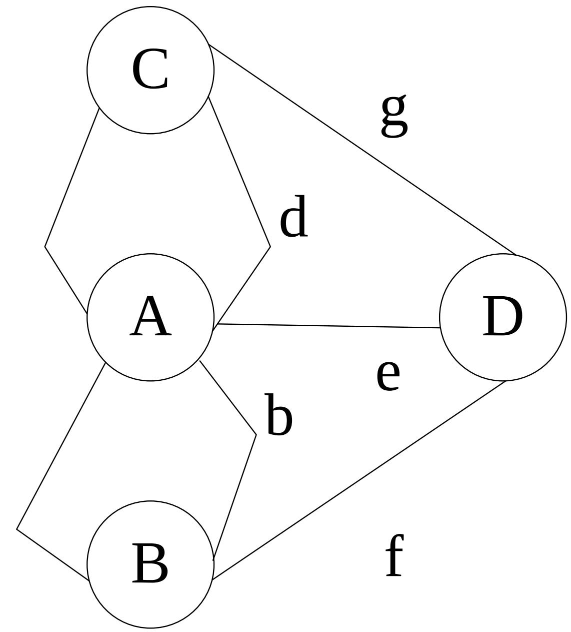
Graphs

Konigsberg Bridge Problem





Euler’s Graph

- Degree of a vertex: The number of edges incident to it
- Euler showed that 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.
  - This walk is called Eulerian.
- No Eulerian walk of the Konigsberg bridge problem since all four vertices are of odd edges.

Application of Graphs

- Analysis of electrical circuits
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Statistical mechanics
- Genetics
- Cybernetics
- Linguistics
- Social Sciences

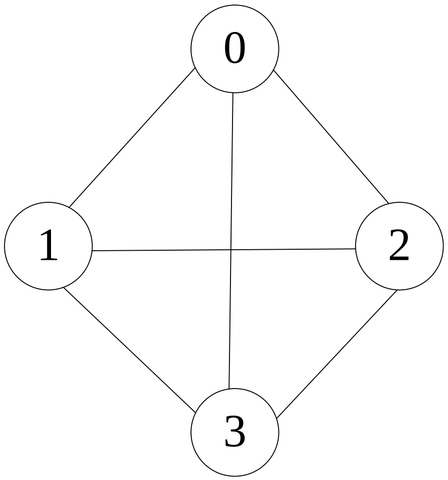
Definition of a Graph

- A graph, $G = (V, E)$, consists of two sets, V and E.
  - V is a finite, nonempty set of vertices.
  - E is set of pairs of vertices called edges.
- The vertices of a graph G can be represented as $V(G)$.
- The edges of a graph, G, can be represented as $E(G)$.
- Graphs can be either **undirected** graphs or **directed** graphs.
  - Undirected graph: A pair of vertices (u, v) or (v, u) represent the same edge.
  - Directed graph: A directed pair $< u, v >$ has u as the **tail** and the v as the **head**.

tail $\longrightarrow$ head

    - $< u, v > \; : \; u \rightarrow v$
    - $< u, v >$ and $< v, u >$ represent different edges. $< u, v > \, != \, < v, u >$

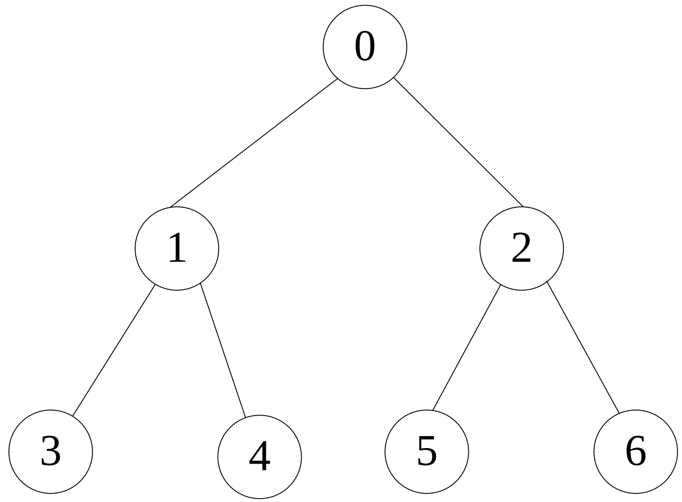
Three Sample Graphs



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

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

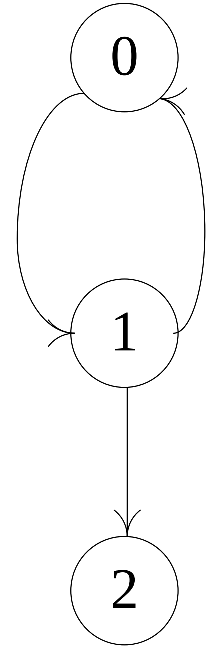
G_1



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

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

G_2



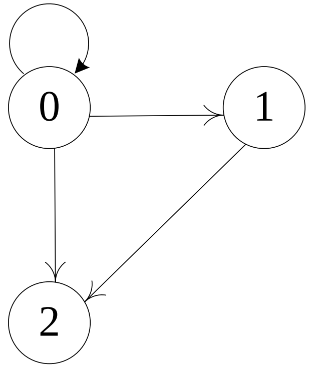
$V(G_3) = \{0, 1, 2\}$

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

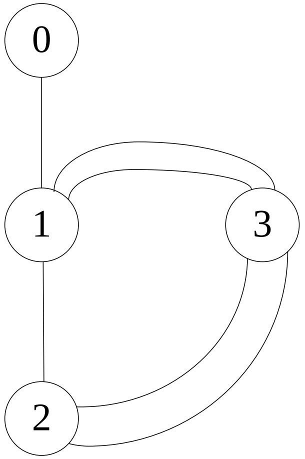
G_3

Graph Restrictions

- A graph may not have an edge from a vertex back to itself
  - (v, v) or $< v, v >$ are called self edge or self loop.
- A graph may not have multiple occurrences of the same edge
  - Without this restriction, it is called a multigraph.



Graph with a self edge



Multigraph Graph

Terminology of Graph

- Graph: $G = (V, E)$
  - V: a set of vertices
  - E: a set of edges
- Edge (arc): A pair (v, w), where $v, w \in V$
- Directed graph (Digraph): A graph with ordered pairs (directed edge)
- Adjacent: w is adjacent to v if $(v, w) \in E$
- Undirected graph: If $(v, w) \in E$, $(v, w) = (w, v)$
- Path: a sequence of vertices $w_1, w_2, \dots, w_N$ where $(w_i, w_{i+1}) \in E$, $\forall\ 1 \le i \le N$.

Terminology of Graph (cont'd)

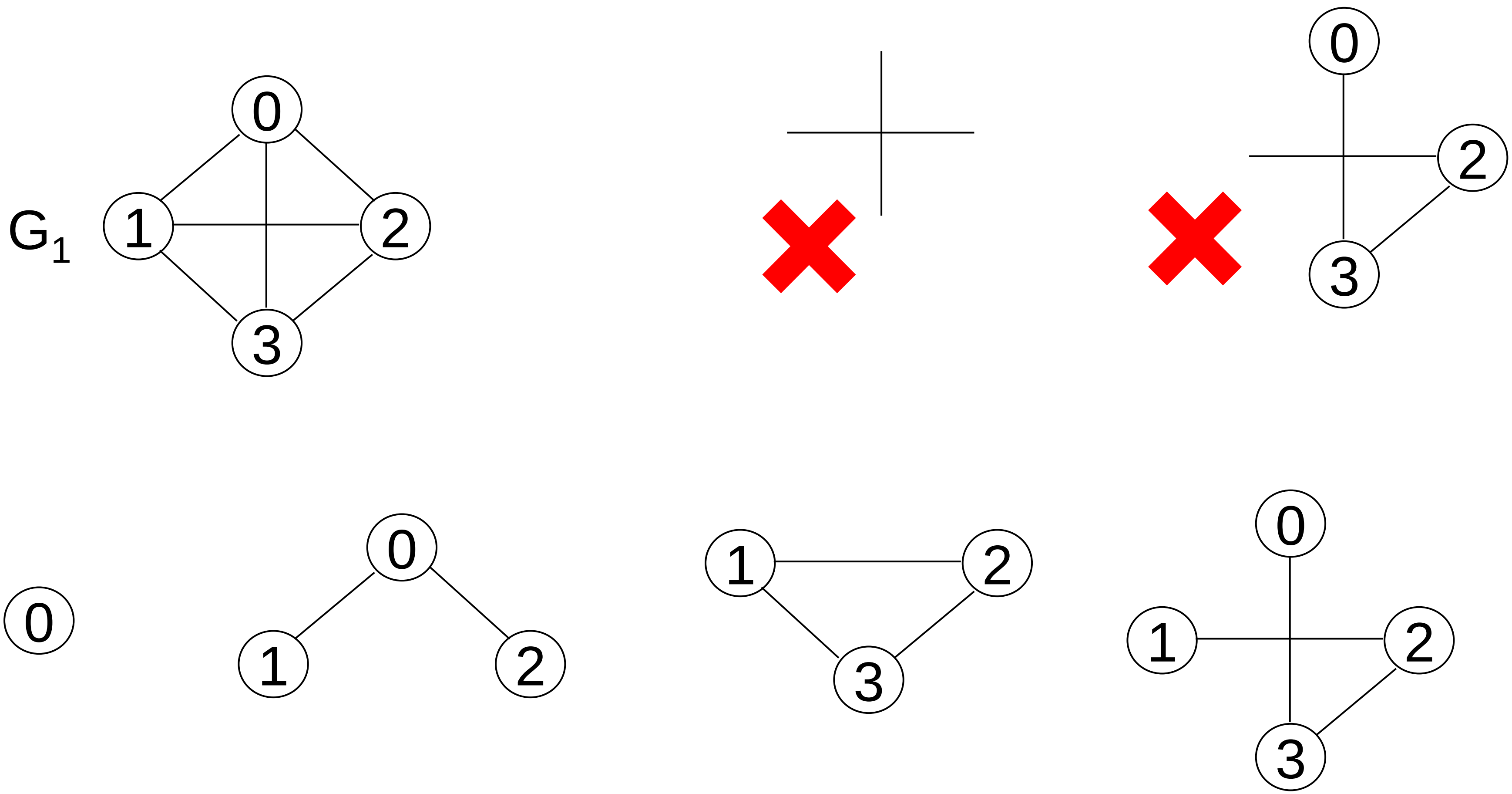
- Length of a path: number of edges on the path.
- Simple path: a path where all vertices are distinct except the first and last.
- Cycle in a directed graph: a path such that $w_1 = \ w_N$
- Acyclic graph (DAG): a directed graph with no cycle.
- Connected: an undirected graph if there is a path from every vertex to every vertex.
- Strongly connected: a directed graph if there is a path from every vertex to every vertex

Complete Graph

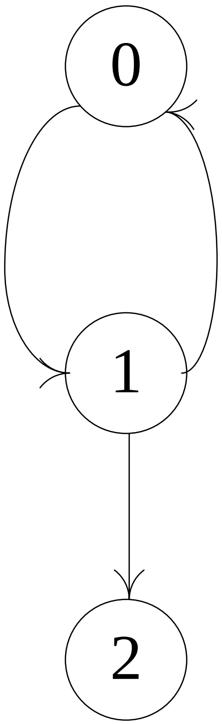
- Complete graph: a graph in which there is an edge between every pair of vertices.
  - The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $n(n-1)/2$.
  - A complete unordered graph is an unordered graph with exactly $n(n-1)/2$ edges.
  - A complete directed graph is a directed graph with exactly $n(n-1)$ edges.

Subgraph

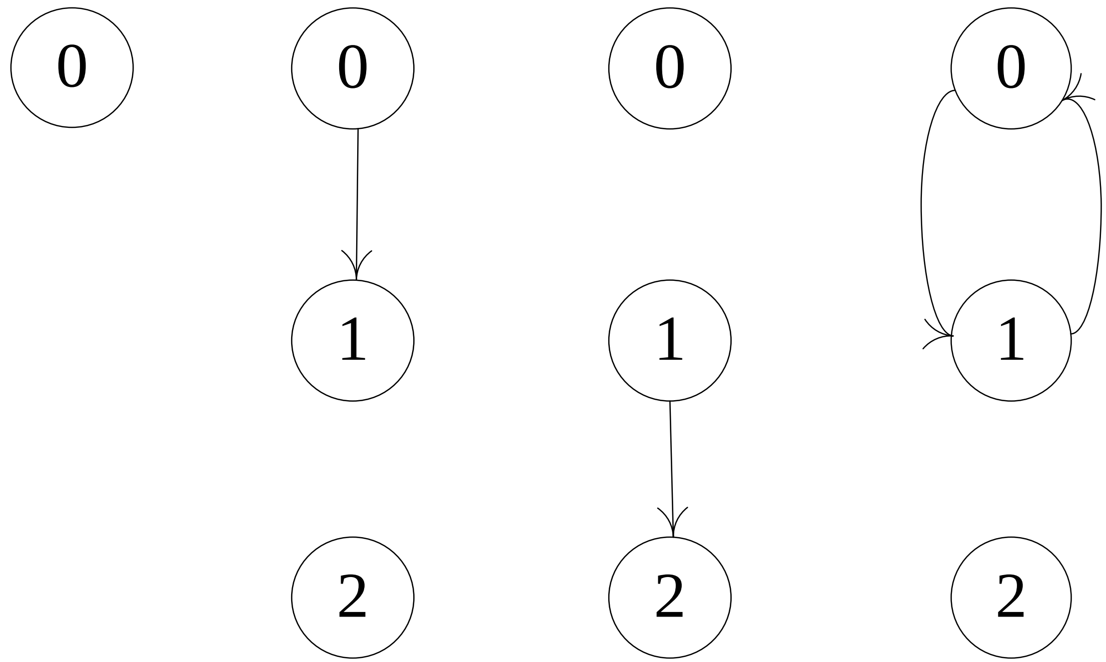
- A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



Subgraph



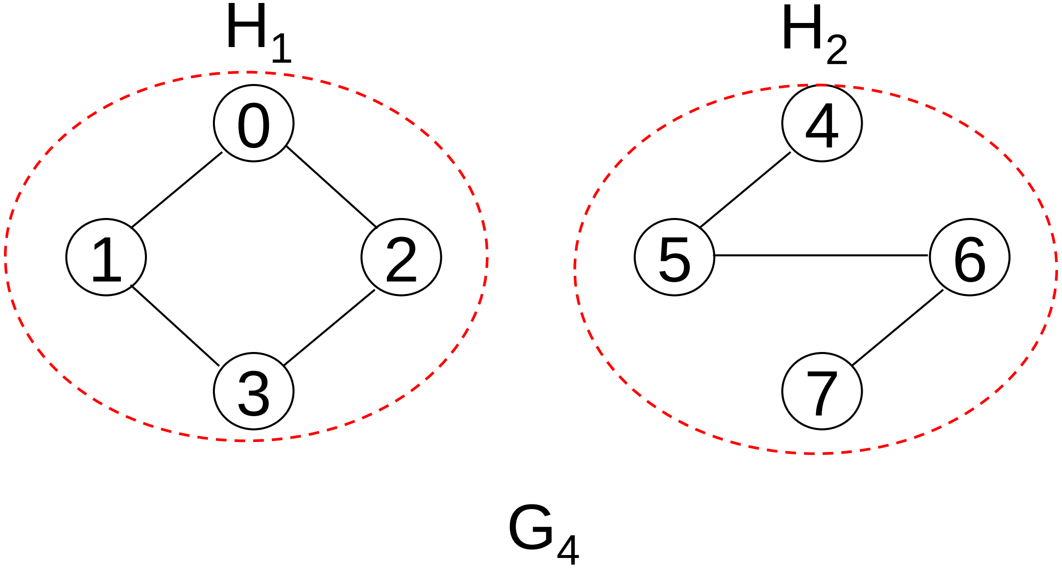
G_3



Subgraphs of G_3

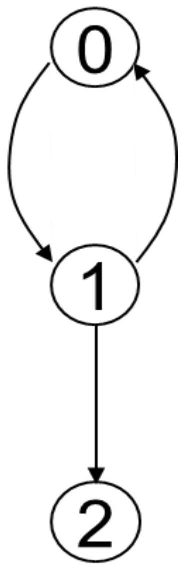
Graphs with Two Connected Components

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

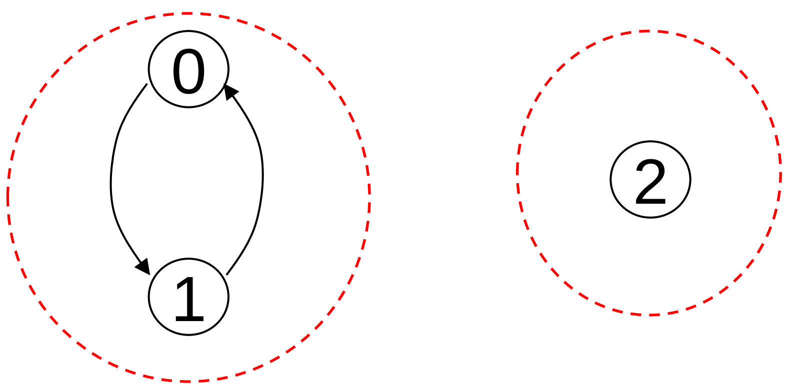


Strongly Connected Component

- A **directed graph** G is said to be strongly connected iff
  - for each 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.
- A strongly connected component is a maximal subgraph that is strongly connected.



G_3



Strongly Connected Components of G_3

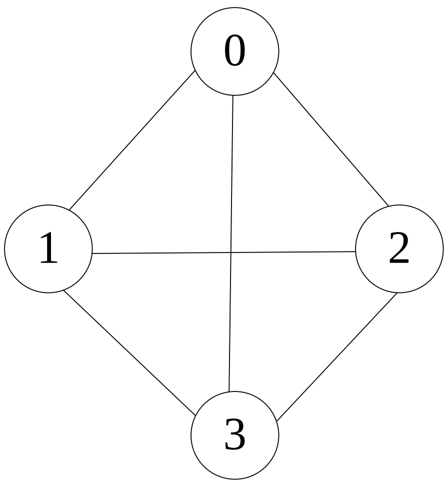
Degree of a Vertex

- Degree: The degree of a vertex is the number of edges incident to that vertex.
- If G is a directed graph, then we define
  - In-degree of a vertex: is the number of edges for which vertex is the head.
  - Out-degree of a vertex: is the number of edges for which the vertex is the tail.
- For a graph G with $\boldsymbol{n}$ vertices and $\boldsymbol{e}$ edges, if $\boldsymbol{d_i}$ is the degree of a vertex $\boldsymbol{i}$ in G, then the number of edges of G is $\mathrm{e} = \sum_{i=0}^{n-1} d_i$.

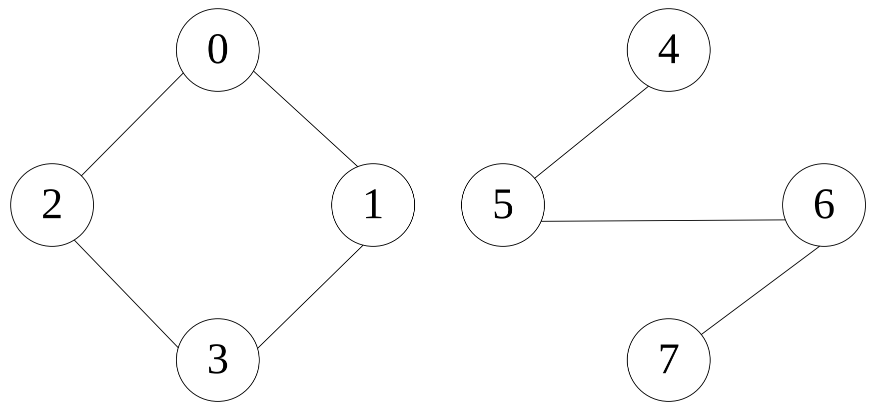
ADT of Graphs

```
class Graph
{// objects: A nonempty set of vertices and a set of undirected edges, where each edge is a
   //edges, where each edge is a pairof vertices.
public:
      virtual ~Graph() {}
            // constructor
      bool IsEmpty() const{return n = = 0};
            // return true iff graph has no vertices
      int NumberOfVertices() const{return n};
            // return no. of vertices in the graph
      int NumberOfEdges() const{return e};
            // return no. of edges in the graph
      virtual int Degree(int u) const = 0;
            // return no. of edges incident to vertex u
      virtual bool ExistsEdge(int u, int v) const = 0;
            // return true iff graph has the edge (u, v)
      virtual void InsertVertex(int v) = 0;
            // insert vertex v into graph; v has no edge
      virtual void InsertEdge(int u, int v) = 0;
            // insert edge (u, v) into graph
      virtual void DeleteVertex(int v) = 0;
            // delete vertex v and all edges incident to it
      virtual void DeleteEdge(int u, int v) = 0;
            // delete edge (u, v)
      private:
            int n;     // no of vertices
            int e;     // no of edges
};
```

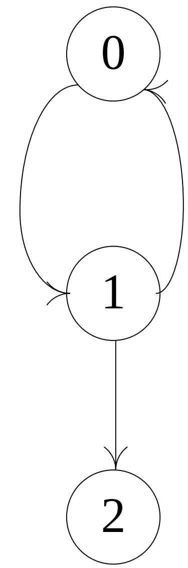
Adjacency Matrix Representation



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



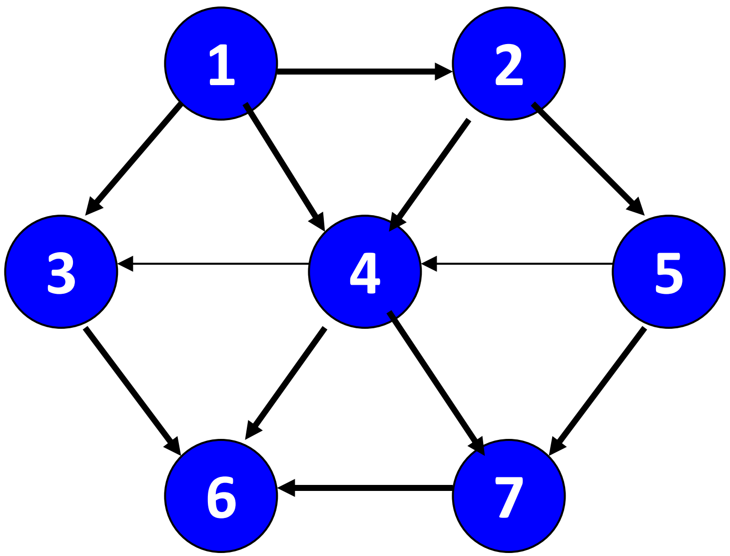
$$
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$



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

Adjacency Matrix Representation

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



good for dense, not for sparse

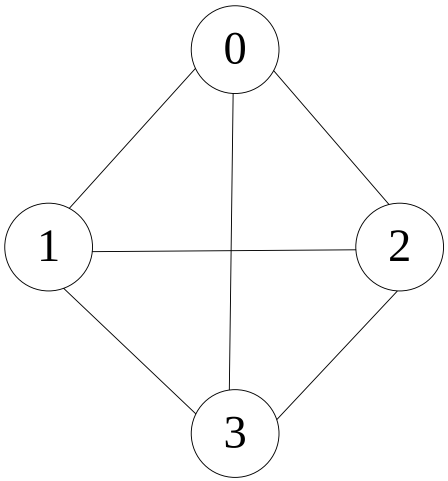
Space: $\Theta(|V|^2)$

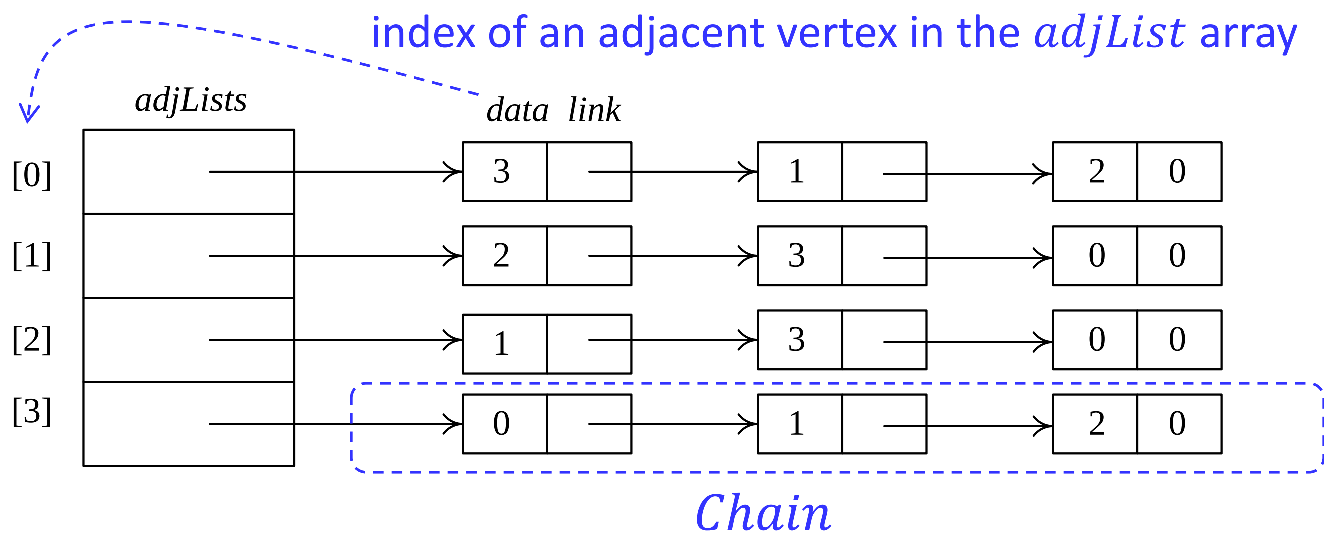
*Undirected graph: symmetric matrix ($|V|^2/2$)

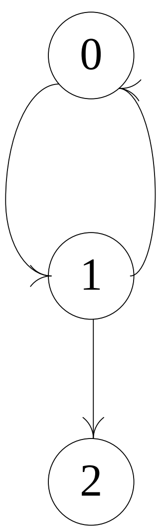
Counting no. of edges: at least $O(|V|^2)$

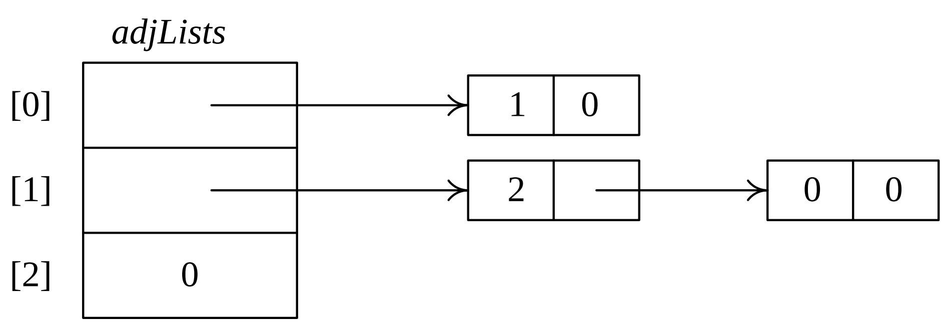
Adjacent Lists

$adjLists$= new $Chain$<int>[n];

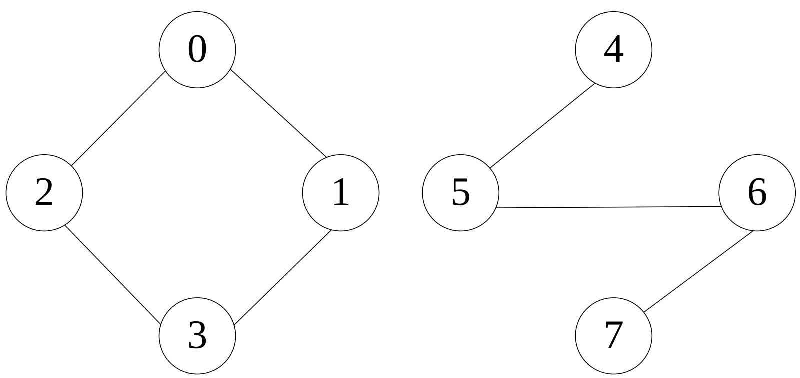


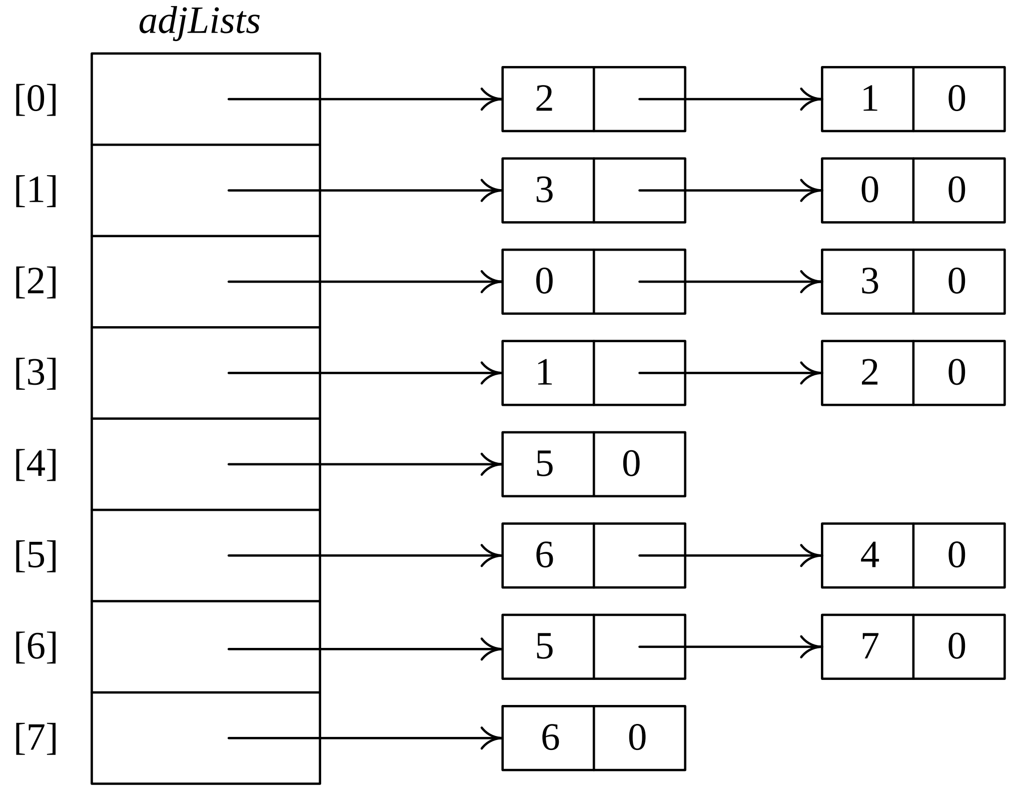






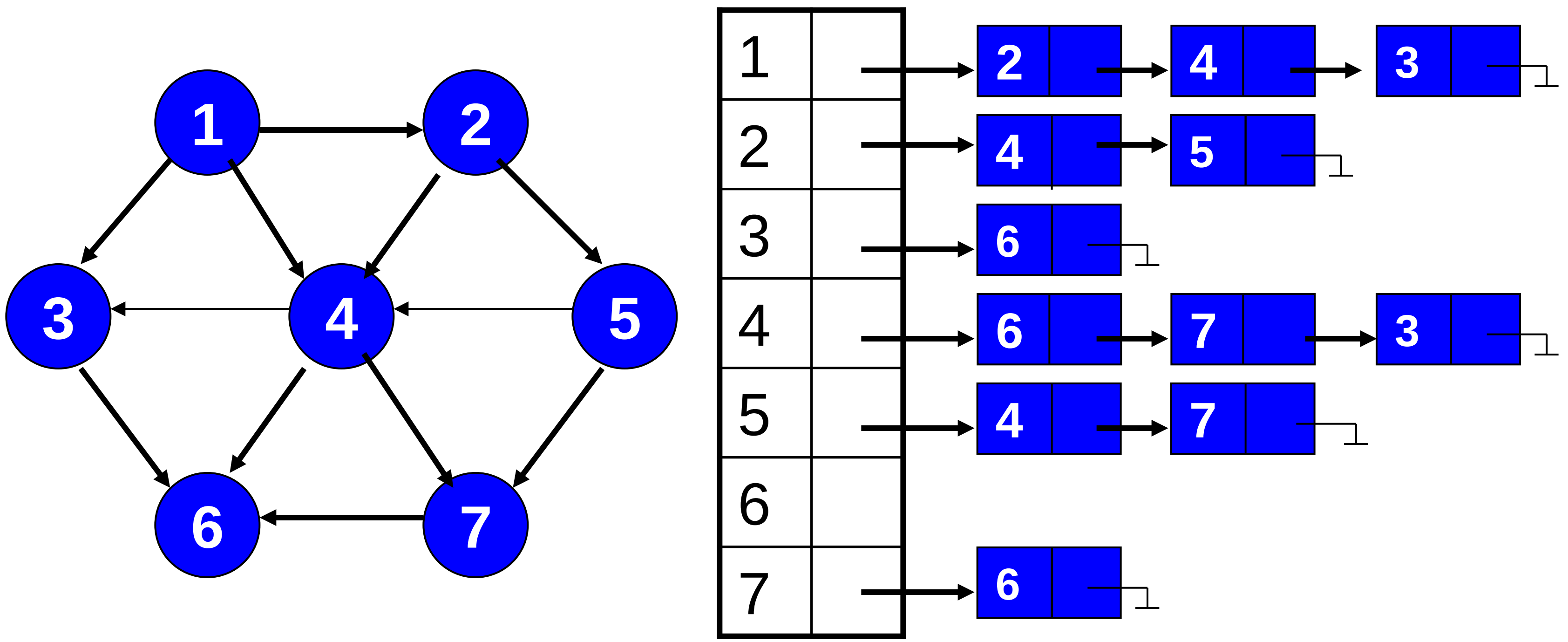
Adjacent Lists





Space: $O(|V| + 2|E|)$ for undirected graph

Adjacency List Representation



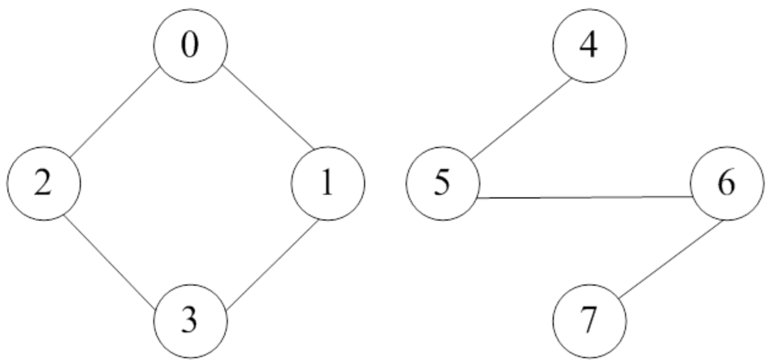
Space: $O(|V| + |E|)$ for directed graph

good for sparse

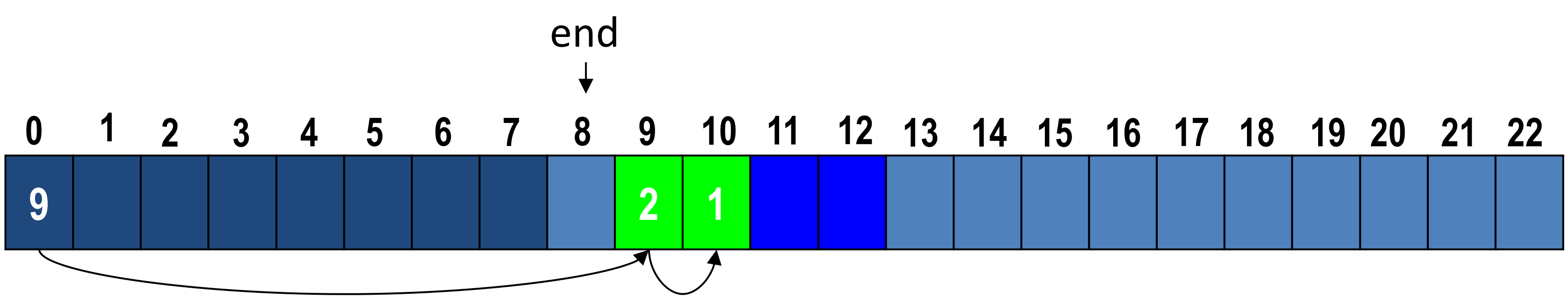
Sequential Representation of Graph G_4

$\textbf{int}\ nodes[\,n\ +\ 2*e\ +\ 1\,];$

- $nodes[\,i\,]$ are starting index for vertex i ;
- $nodes[\,i\,]\sim nodes[\,i+1] - 1$ store edge information for vertex i ;



Edges adjacent to vertex 0: (0, 1), (0, 2)



starting index to its edge information

edge information of all vertex

Sequential Representation of Graph G_4

Edges adjacent to vertex 1: (1, 0), (1, 3)

end

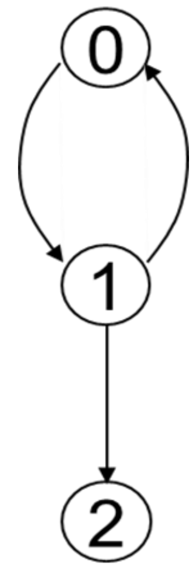
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11								2	1	3	0										

…

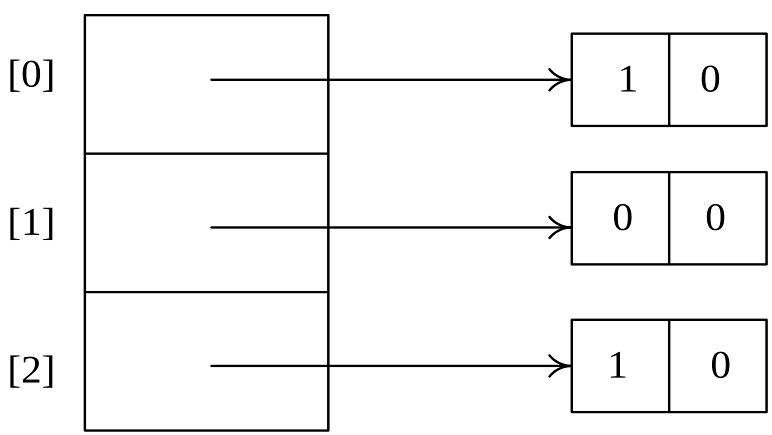
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

Inverse Adjacency Lists for G_3

- Adjacent list
  - Out-degree
- Inverse adjacent list
  - In-degree



G_3

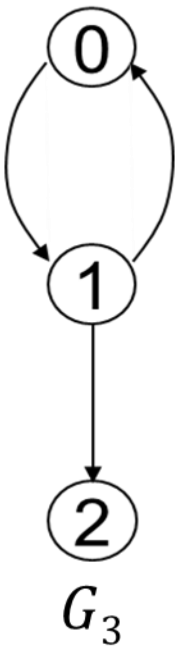


Inverse adjacent list of G_3

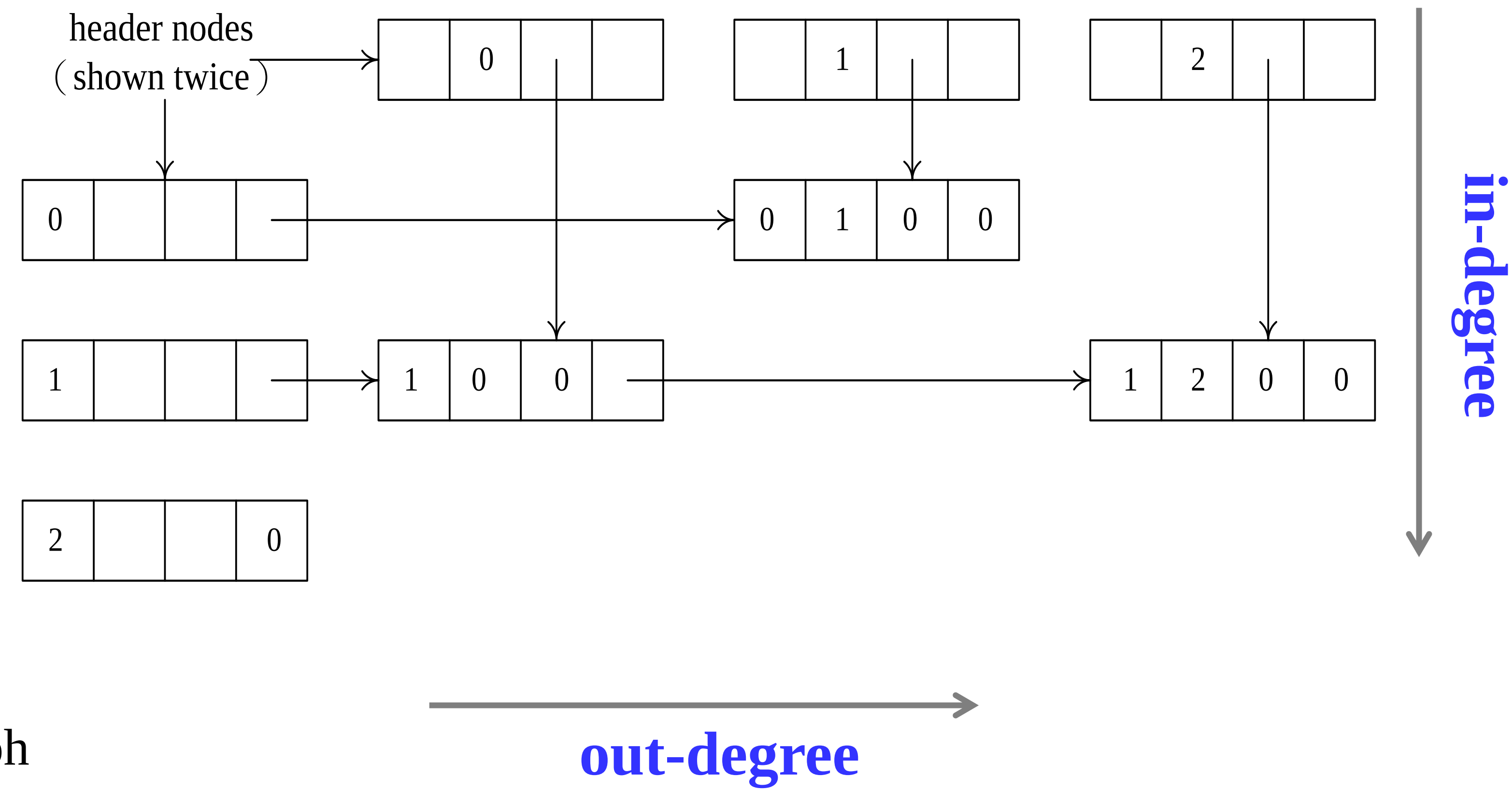
Multilists

- In the adjacency-list representation of an undirected graph, each edge (u, v) is represented by two entries.
- **Multilists**: To be able to determine the second entry for a particular edge and mark that edge as having been examined, we use a structure called multilists.
  - Each edge is represented by one node.
  - Each node will be in two lists (nodes may be shared among server lists)

Orthogonal List Representation



Directed graph



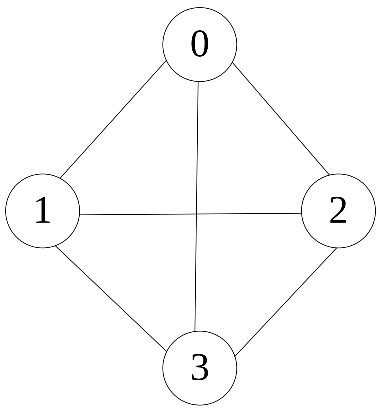
Adjacency Multilists for G_1

edge node structure in the list

m	vertex1	vertex2	link1	link2

Mark field used for indication

Undirected graph



N0		0	1		edge (0,1)
N1		0	2		edge (0,2)
N2		0	3		edge (0,3)
N3		1	2		edge (1,2)
N4		1	3		edge (1,3)
N5		2	3		edge(2,3)

The lists are

vertex 0 : N0 → N1 → N2

vertex 1 : N0 → N3 → N4

vertex 2 : N1 → N3 → N5

vertex 3 : N2 → N4 → N5

Weighted Edges

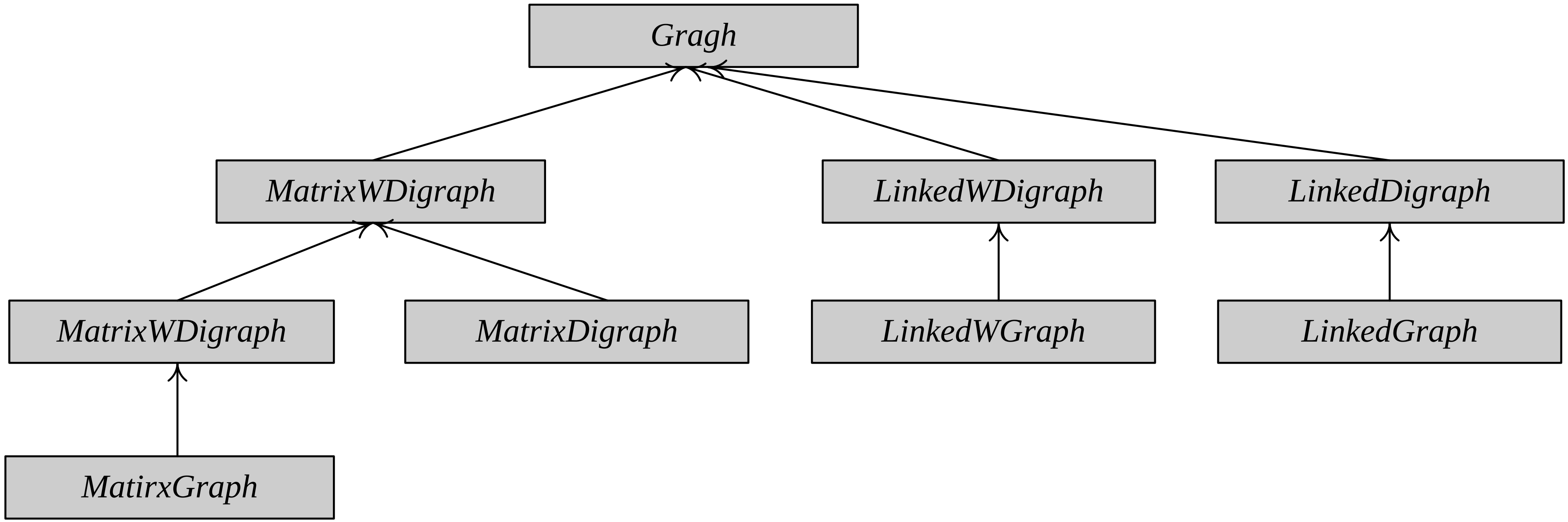
- Very often the edges of a graph have weights associated with them.
  - Distance from one vertex to another
  - Cost of going from one vertex to an adjacent vertex
- To represent weight, we need additional field, $weight$, in each entry.
- A graph with weighted edges is called a *network*.

Possible Graph Derivation Hierarchy

- Matrix
- Linked adjacency lists
- Sequential adjacency list
- Adjacency multilists

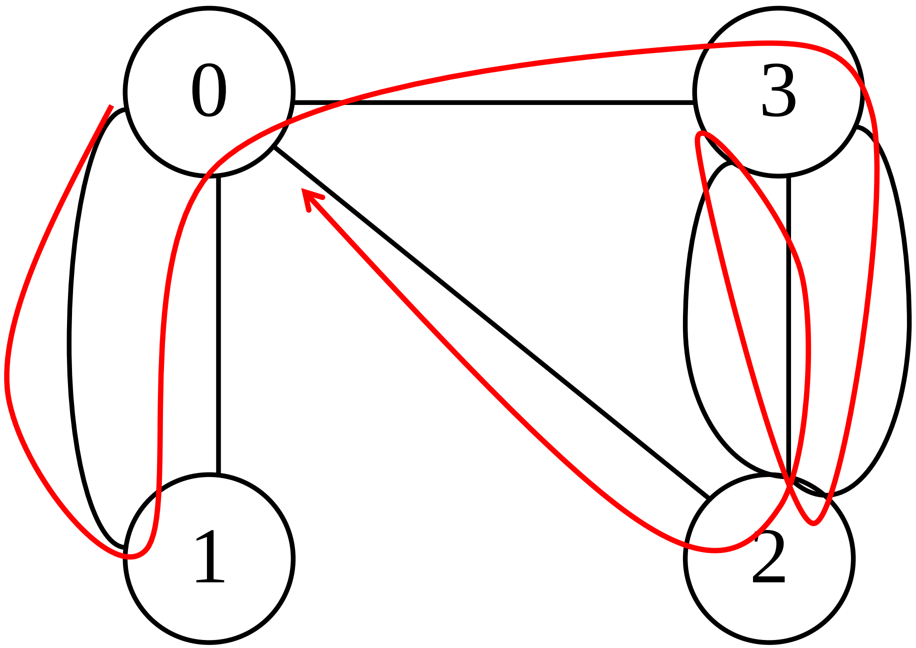
- Directed
- Undirected

- Weighted
- Unweighted



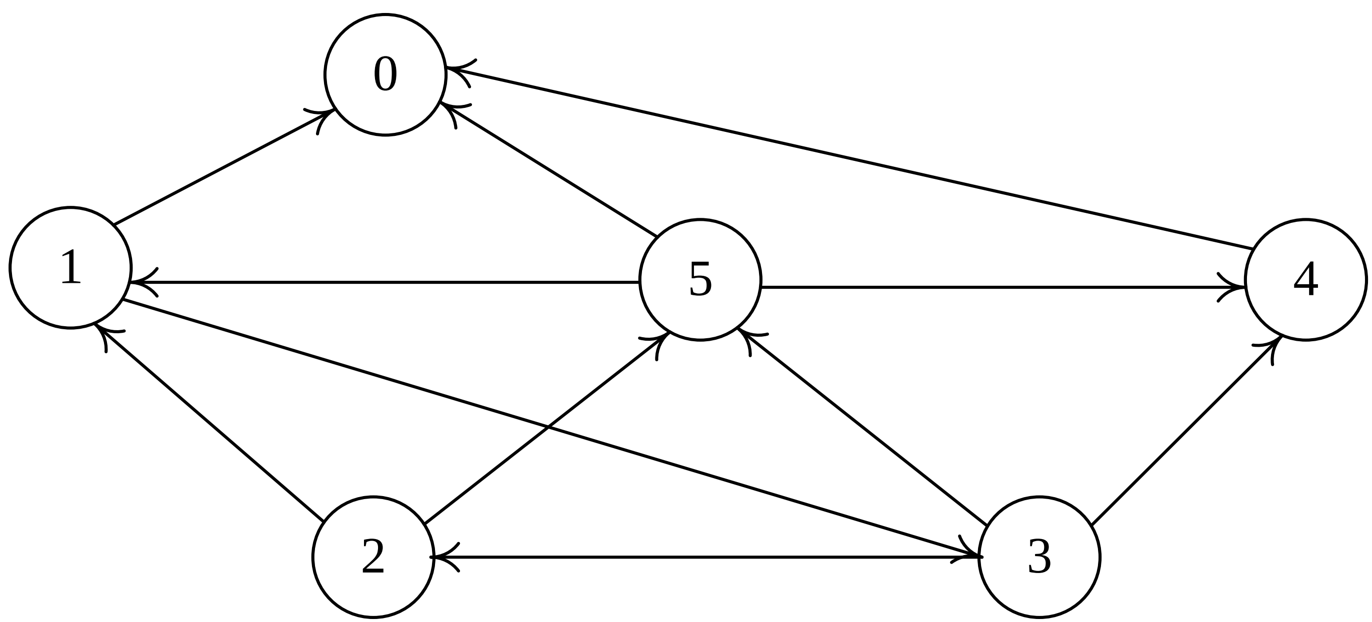
Eulerian Walk

- Does the graph have an Eulerian walk?
  - A walk starting at any vertex, going through each edge exactly once and terminating at the start vertex



Digraph

- What are the In-degree and out-degree of 5?
- What are the strongly connected components?
  – For each u and v, existing a directed path from u to v and also from v to u
  – Maximal



Graph Operations

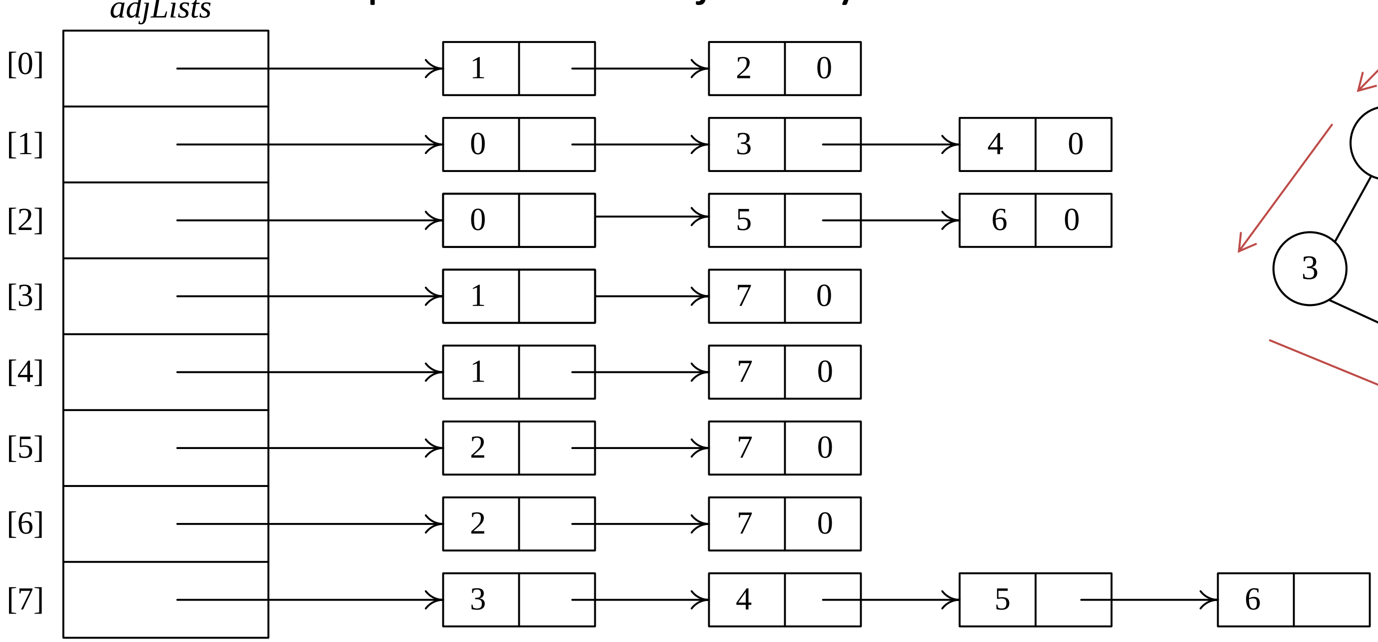
- A general operation on a graph G is to visit all vertices in G that are reachable from a vertex v.
  - Depth-first search
  - Breadth-first search
- Both search methods work on directed and undirected graphs.

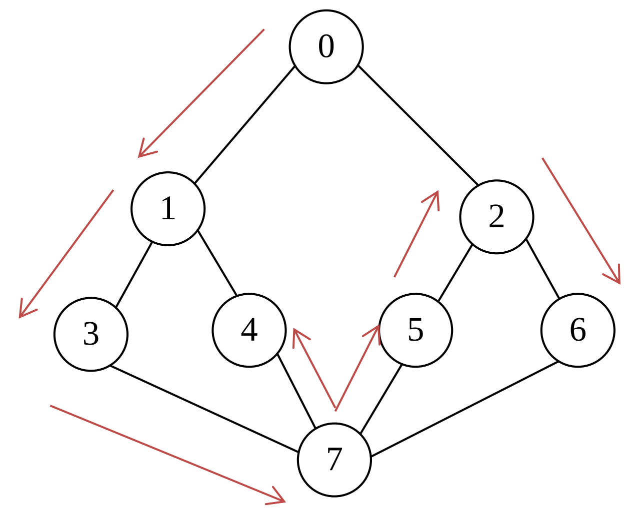
Depth-First Search

- Depth First Search: generalization of preorder traversal
- Starting from vertex v, process v & then recursively traverse all vertices adjacent to v.
  - Using stack
- To avoid cycles, mark visited vertices

Depth-First Search

Graph G and Its Adjacency Lists





DFS starts from 0: 0, 1, 3, 7, 4, 5, 2, 6

Depth-First Search (cont'd)

```
virtual void Graph::DFS() // driver
{
      visited = new bool[n];
            // visited is declared as a bool* data member of Graph
      fill (visited, visited + n, false);
      DFS(0); // start search at vertex 0
      delete [] visited;
}
virtual void Graph::DFS(const int v)    // workhorse
{ // visit all previously unvisited vertices that are reachable from v
      visited[v] = true;
       cout<< v;
      for (each vertex w adjacent to v) // actual code uses an iterator
            if (!visited[w]) DFS(w);
}
```

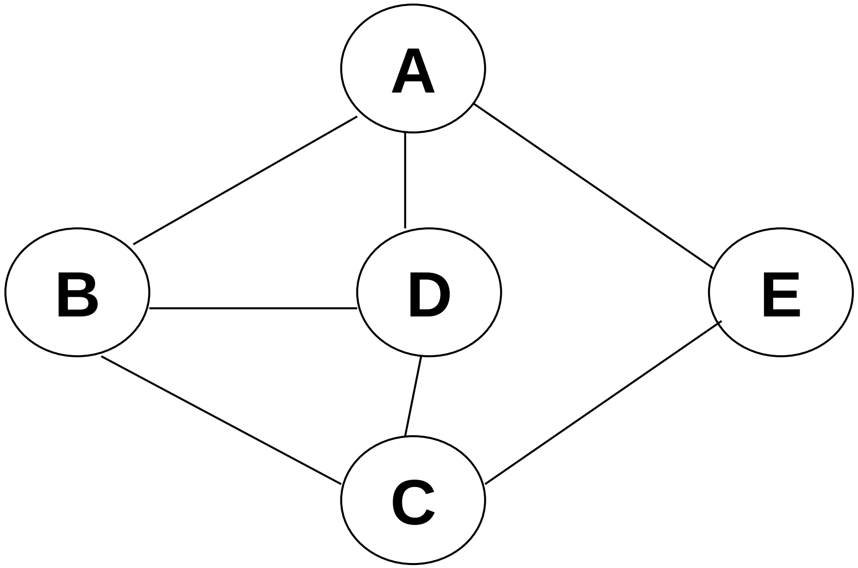
Analysis of DFS

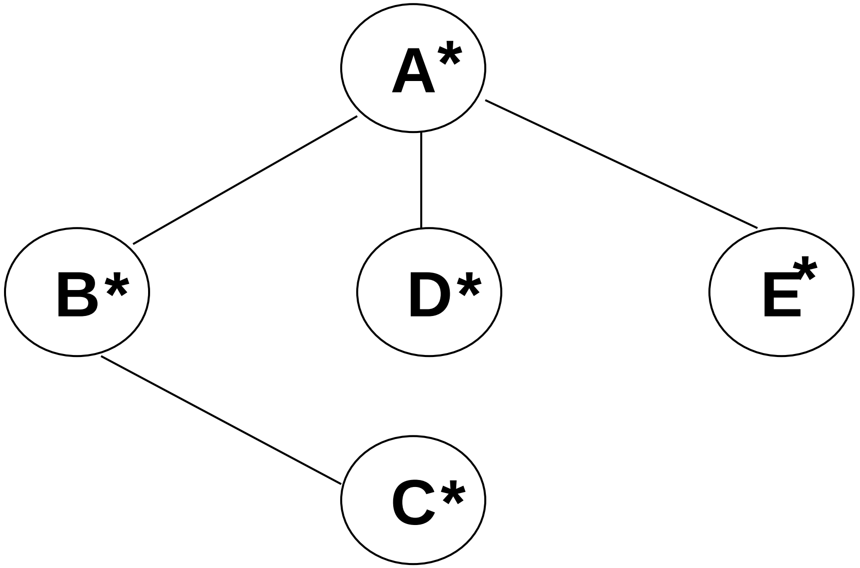
- If G is represented by its adjacency lists, the DFS time complexity is $O(e)$.
  - There are $2e$ list nodes in the adjacency lists
- If G is represented by its adjacency matrix, then the time complexity to complete DFS is $O(n^2)$.

Breadth-First Search

- Breadth-First search (BFS): level order tree traversal
  - BFS algorithm: using queue
- To avoid cycles, mark visited vertices
- If G is represented by its adjacency lists, the BFS time complexity is $O(e)$.
- If G is represented by its adjacency matrix, then the time complexity to complete BFS is $O(n^2)$.

Breadth First Search (cont'd)





BFS from A: A, B, D, E, C

Breadth First Search (cont'd)

```
virtual void Graph::BFS(int v)
{// a breadth first search of the graph is carried out beginning at vertex v
 // visited[i] is set to true when v is visited. The function uses a queue.
    visited = new bool [n];
    fill (visited, visited + n, false);
    visited[v] = true;
    Queue<int> q;
    q.Push (v);
    while (!q.IsEmpty ()) {
        v = q.Front ();
        cout << v;
        q.Pop (); //delete
        for (all vertices w adjacent to v)    // actual code uses an iterator
            if (!visited [w]) {
                q.Push (w);
                visited[w] = true;
            }
    }   //end of while
    delete [] visited;
}
```

Connected Components

- If G is an undirected graph, its connected components can be determined by calling DFS or BFS
- Check if there is any unvisited vertex
- If G is represented by adjacency lists, the time complexity is $O(n + e)$
  - $O(e)$ for DFS
  - $O(n)$ for for loops
- If G is represented by adjacency graphs, the time complexity is $O(n^2)$

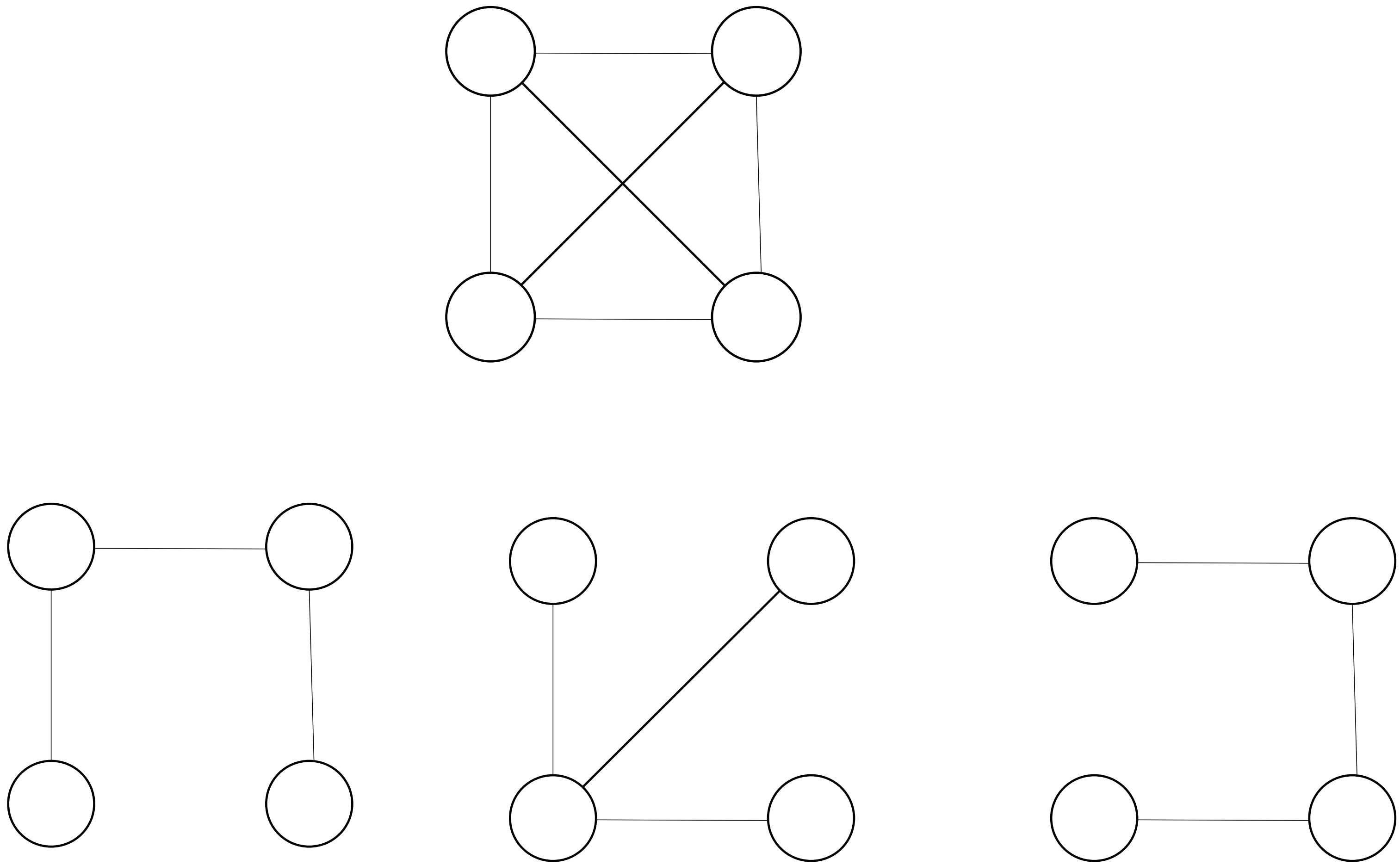
Find Connected Components

```
virtual void Graph::Components()
{// determine the connected components of the graph
 // visitedis assumed to be declard as a bool* data member of Graph
      visited = new bool [n];
      fill (visited, visited + n, false);
      for (i = 0 ; i < n ; i++)
            if (!visited [i]) {
                  DFS(i); //find a component
                  OutputNewComponent ();
            }
      delete [] visited;
}
```

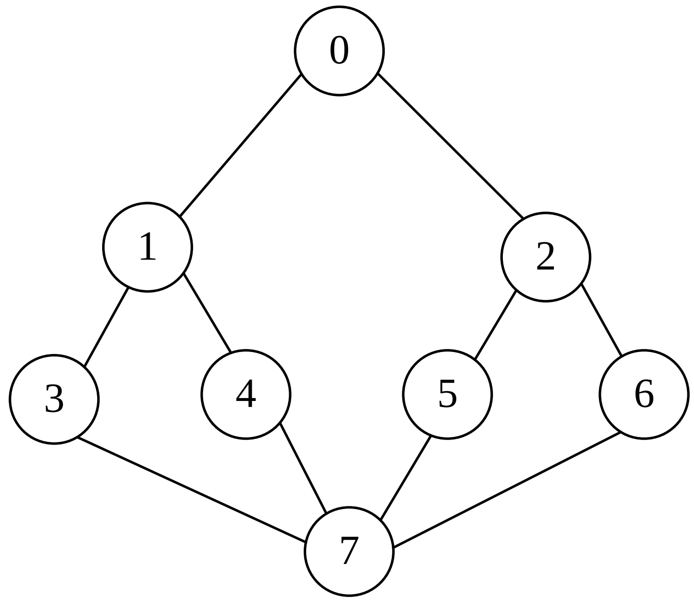
Spanning Tree

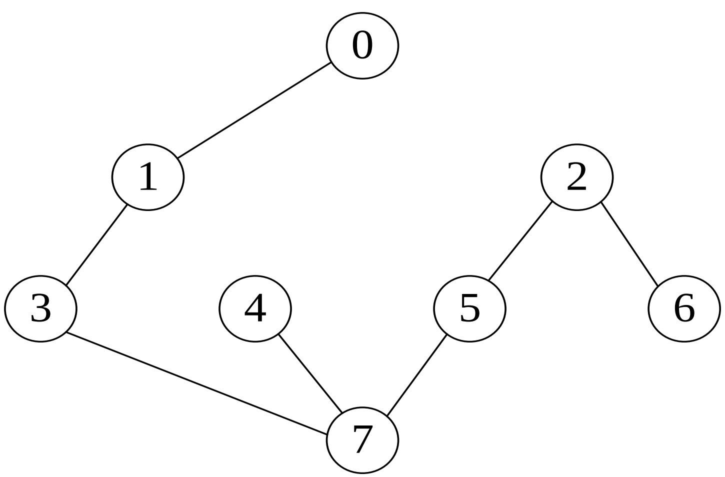
- Any tree consisting solely of edges in G and including all vertices in G is called a *spanning tree*.
  - Can be obtained by using either a DFS or a BFS.
- A spanning tree is a minimal subgraph, G', of G such that $V(G') = V(G)$, and G' is connected.
  - Minimal subgraph is defined as one with the fewest number of edges.
- A spanning tree has $n-1$ edges
  - Any connected graph with n vertices must have at least $n-1$ edges
  - all connected graphs with $n-1$ edges are trees.

A Complete Graph and Its Spanning Trees

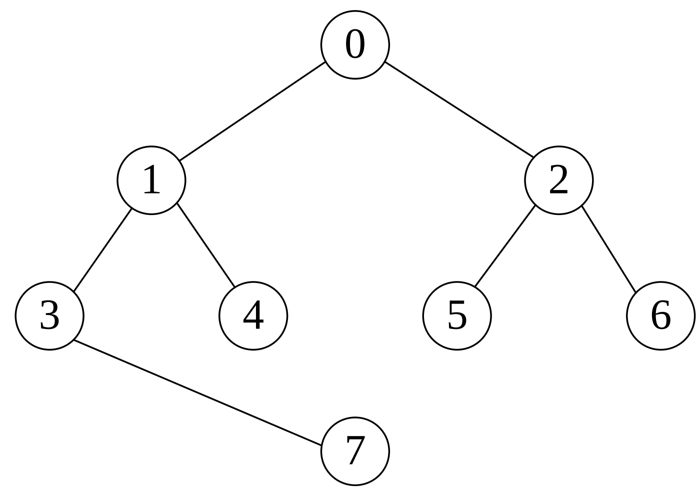


Depth-First and Breadth-First Spanning Trees





DFS (0) spanning tree



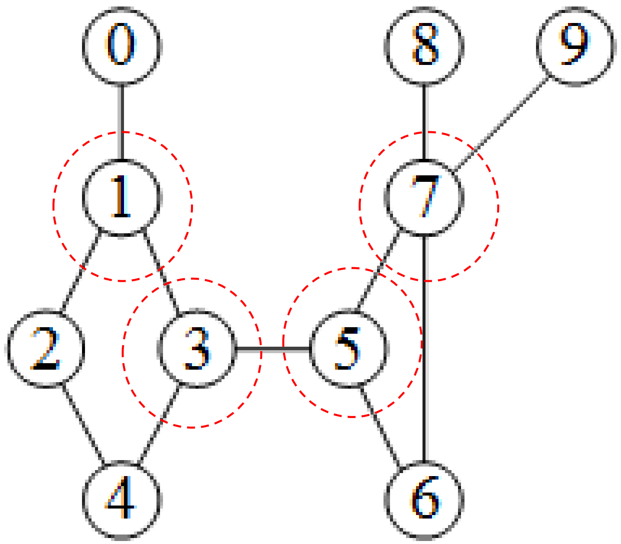
BFS (0) spanning tree

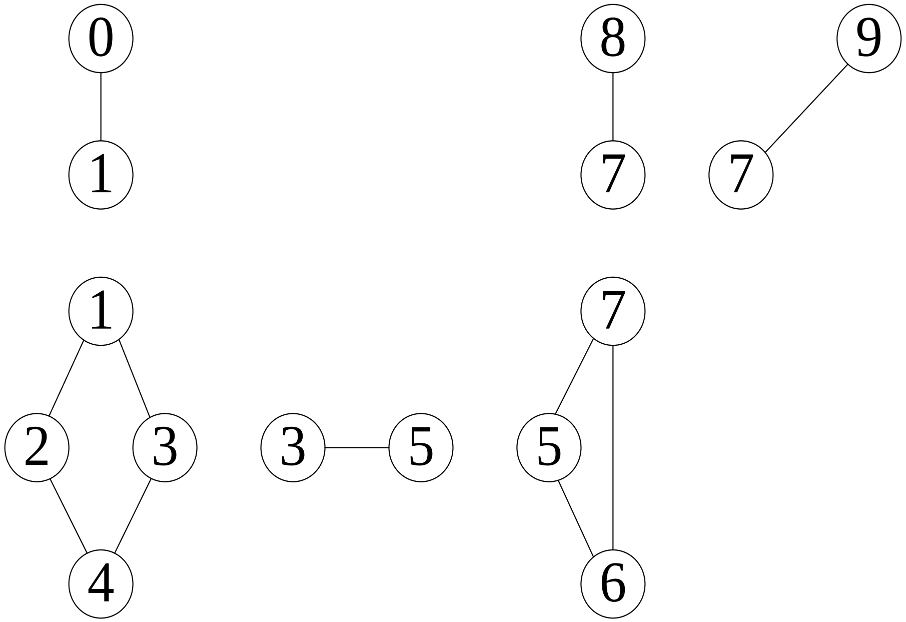
Biconnected Components

- **Articulation Point**: A vertex v of G is an articulation point iff **the deletion of v, together with the deletion of all edges incident to v**, leaves behind a graph that has at least 2 connected components.
- A biconnected graph is a connected graph that has no articulation points
- A biconnected component of a connected graph G is a **maximal biconnected subgraph** H of G
  - Maximal: G contains no other subgraph that is both biconnected and properly contains H.

A Connected Graph and Its Biconnected Components

Any articulation point?





biconnected components

- maximal
- biconnected
- subgraph

How to find articulation point?

Biconnected Components (contd.)

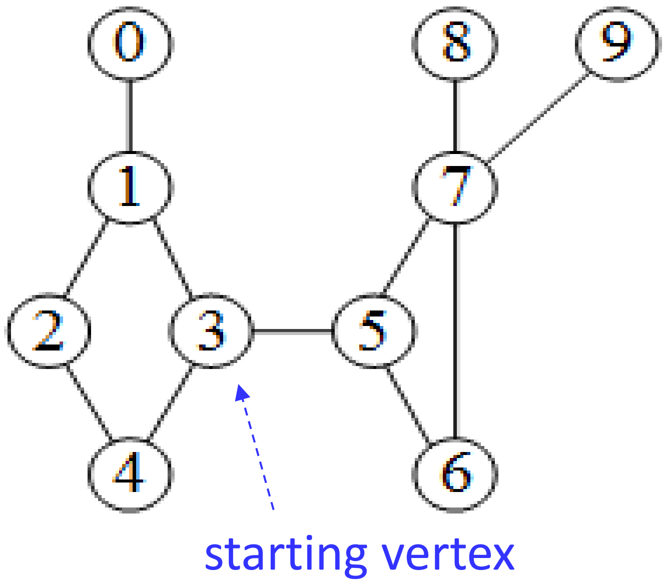
- Properties of BCC:
  - Two biconnected components of the same graph can have **at most one vertex** in common
  - The biconnected components of G partition the edges of G
  - **No edge** can be in two or more biconnected components
- The biconnected components of a connected, undirected graph G can be found by using any **DFS tree** of G

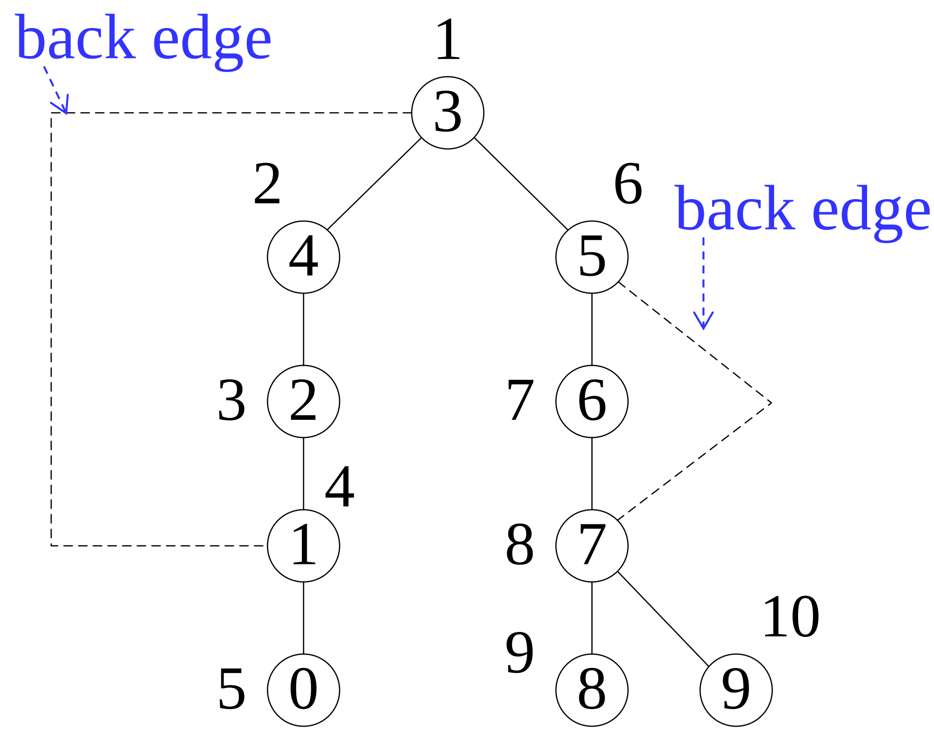
Biconnected Components (contd.)

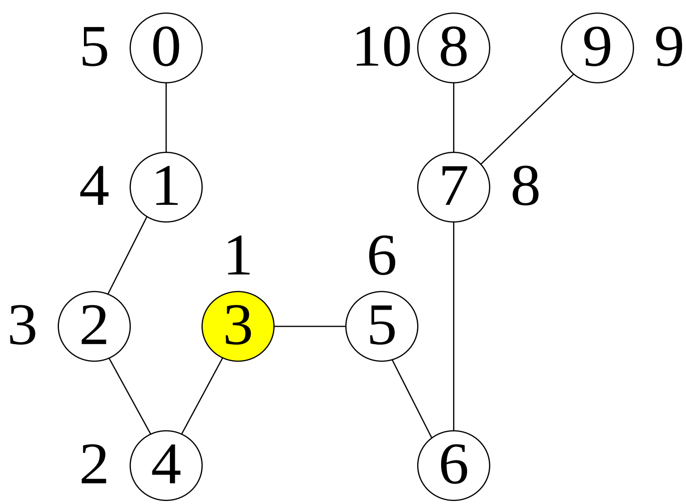
- In a DFS of an undirected graph G, every edge of G is either tree edge or a back edge
  - Edge (u, v) is a **tree edge** if v was first discovered by exploring edge (u, v)
  - A nontree edge (u, v) is a **back edge** with respect to a spanning tree T *iff* either u is an ancestor of v or v is an ancestor of u

✓ Forward edges and cross edges only appear in directed graphs

Depth-First Spanning Tree





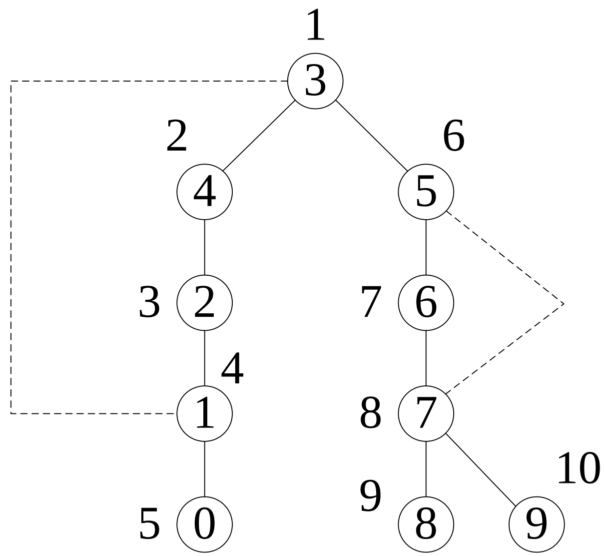


Biconnected Components (contd.)

- The root of the DFS tree is an articulation point *iff* it has at least two children.
- Depth-first number, $\boldsymbol{dfn(w)}$, is defined as the order that w is discovered by DFS
- $\boldsymbol{low(w)}$ is the lowest $dfn(w)$ that can be reached from *w* using a path of descendants followed by, at most, one back edge.

$$low(w) = min\{\, dfn(w),\ min\{\, low(x) | x \textit{ is a child of } w\}, \\ min\{\, dfn(x) | (w, x) \textit{ is a back edge}\}\}$$

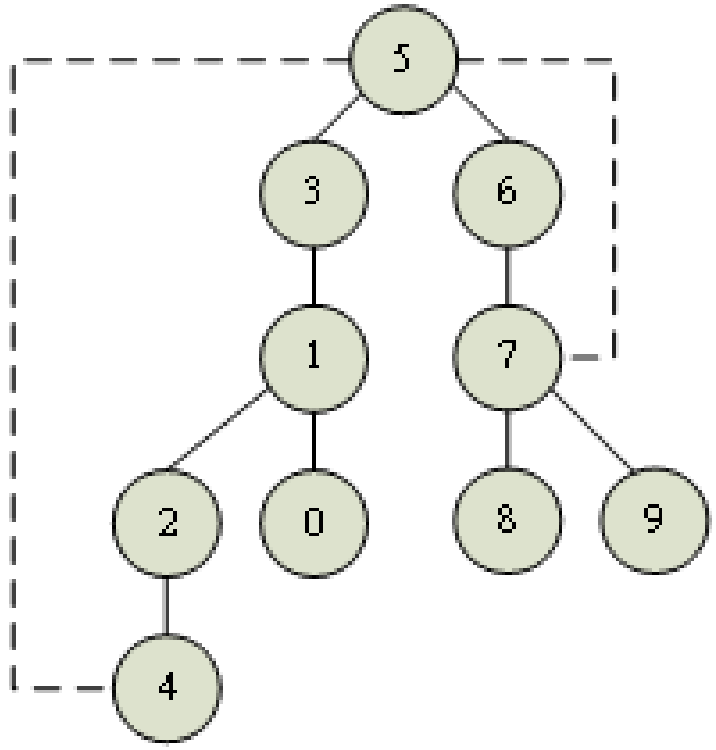
Biconnected Components (contd.)



$$low(w) = min\{\, dfn(w),\ min\{\, low(x) | x \ \textit{is a child of} w\},$$
$$min\{\, dfn(x) | (w, x) \ \textit{is a back edge}\}\}$$

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

Depth-First Spanning Tree



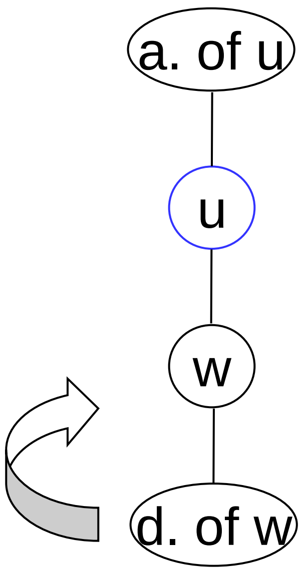
$$low(w) = min\{\, dfn(w),\ min\{\, low(x) | x \ \textit{is a child of} \ w\},\ min\{\, dfn(x) | (w, x) \ \textit{is a back edge}\}\}$$

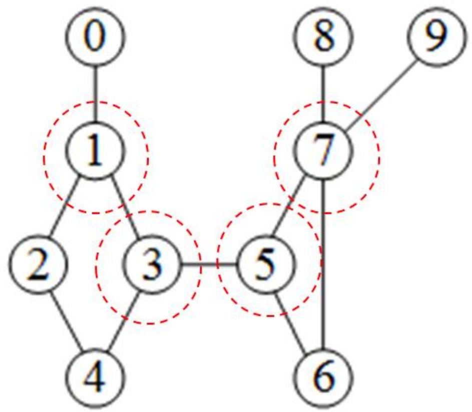
vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	5	2	6	1	7	8	9	10
low	5	1	1	1	1	6	6	6	10	9

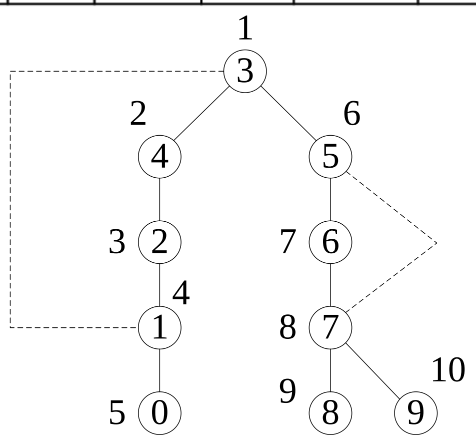
Biconnected Components (contd.)

- A vertex $\boldsymbol{u}$ is an **articulation point** *iff*
  - u is either the root of the spanning tree and has two or more children
  - u is not the root and u has a child w such that $low(w) \geq dfn(u)$.

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

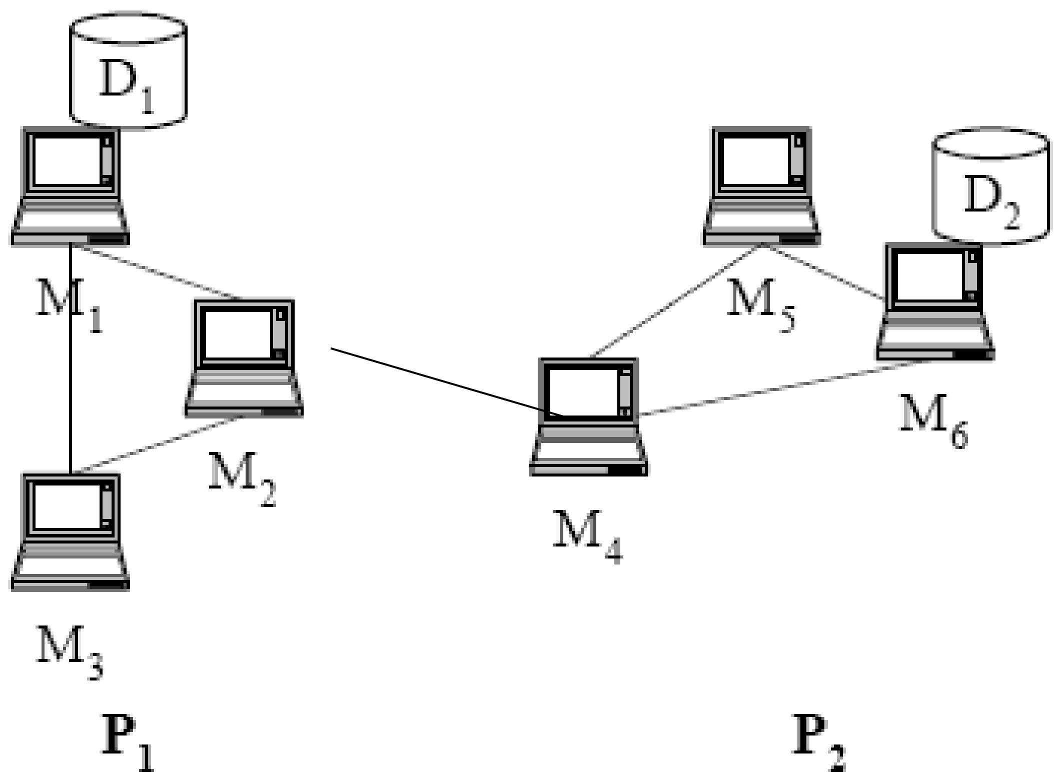






One Usage of Biconnected Components

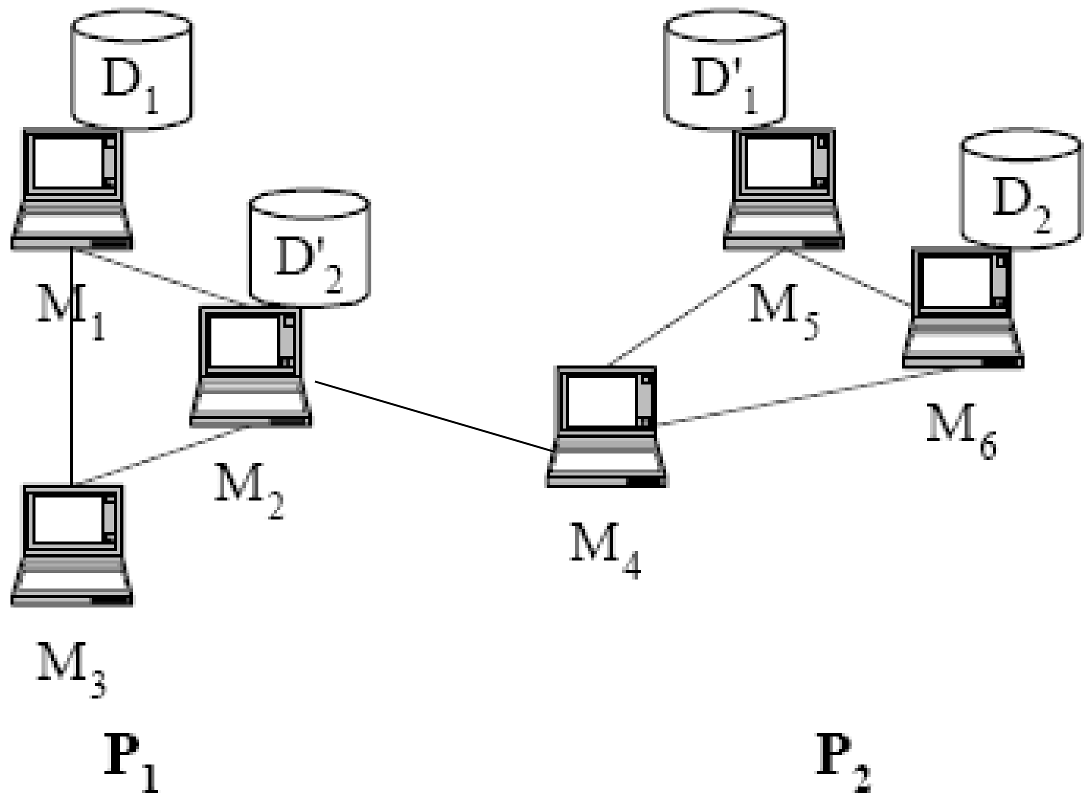
- Ad-hoc network



(a) Without replication

One Usage of Biconnected Components (contd.)

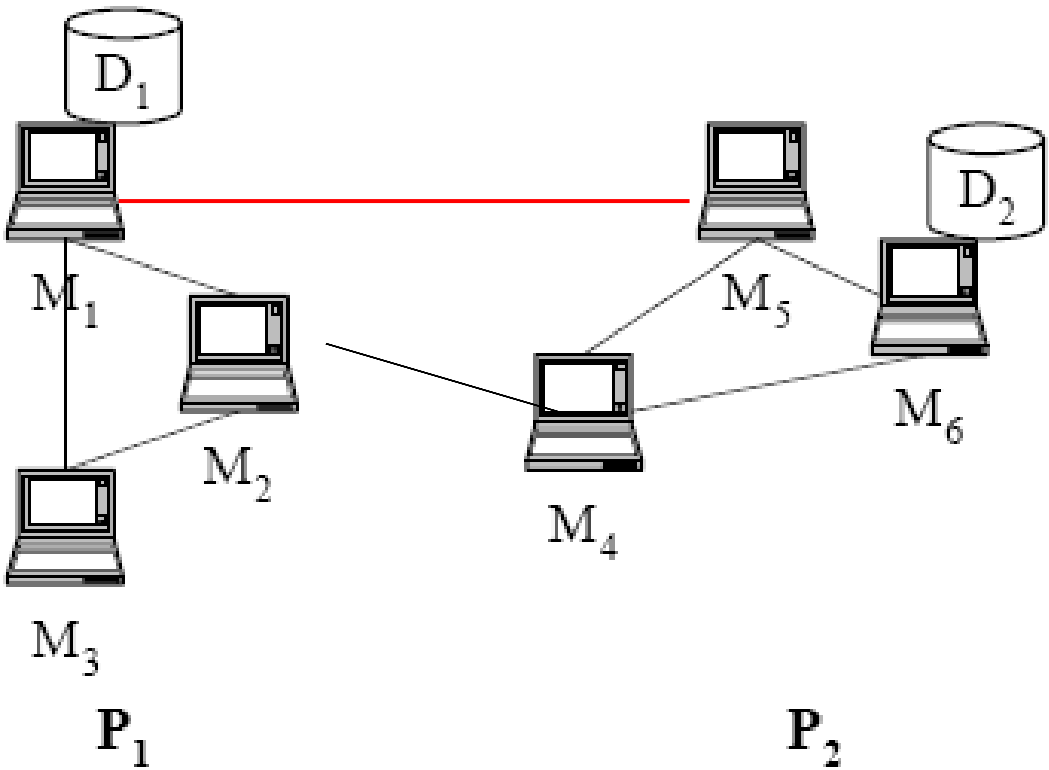
- Ad-hoc network



(b) With replication

One Usage of Biconnected Components

- Ad-hoc network



(c) With replication

Minimal Cost Spanning Tree

- A ***minimum-cost spanning tree*** is a spanning tree of least cost
  - **Cost**: The sum of the weights of the edges in the spanning tree
- Three greedy-method algorithms available to obtain a minimum-cost spanning tree
  - Kruskal's algorithm
  - Prim's algorithm
  - Sollin's algorithm

Select $n - 1$ edges from a weighted graph of n vertices with minimum cost.

Minimal Cost Spanning Tree (contd.)

- Constraints
  - Must use only edges with the graph.
  - Must use exactly $n - 1$ edges.
  - May not use edges that produce a cycle.

Kruskal’s Algorithm

- Kruskal’s algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time.
- The algorithm selects the edges for inclusion in T in non-decreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
- **Theorem 6.1**: Let G be any undirected, connected graph. Kruskal’s algorithm results in a minimum-cost spanning tree.
- Time complexity: $O(e \log e)$

Stages in Kruskal's Algorithm

0 — 10 — 5

2 — 12 — 3

1 — 14 — 6

1 — 16 — 2

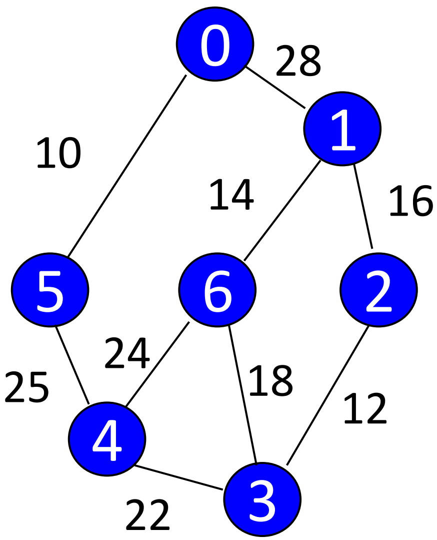
3 — 18 — 6

3 — 22 — 4

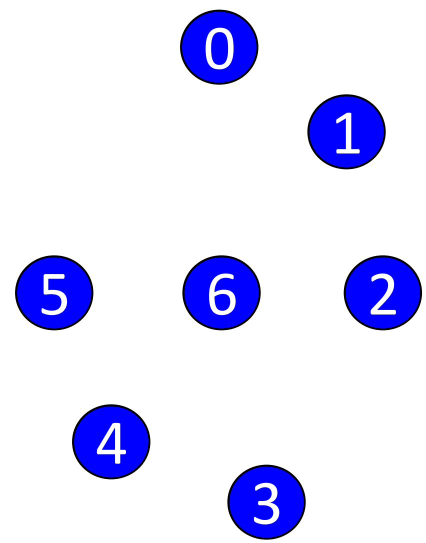
4 — 24 — 6

4 — 25 — 5

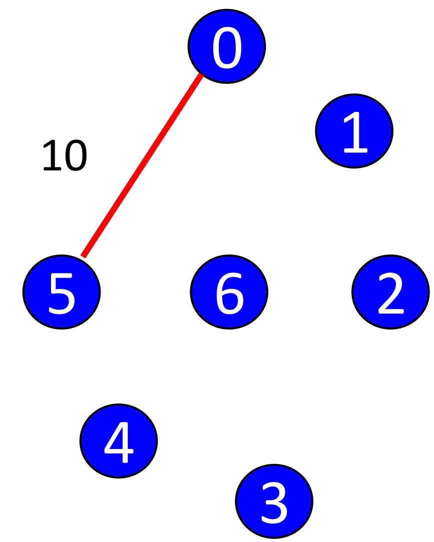
0 — 28 — 1



(a)



(b)



(c)

Stages in Kruskal's Algorithm (Cont.)

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

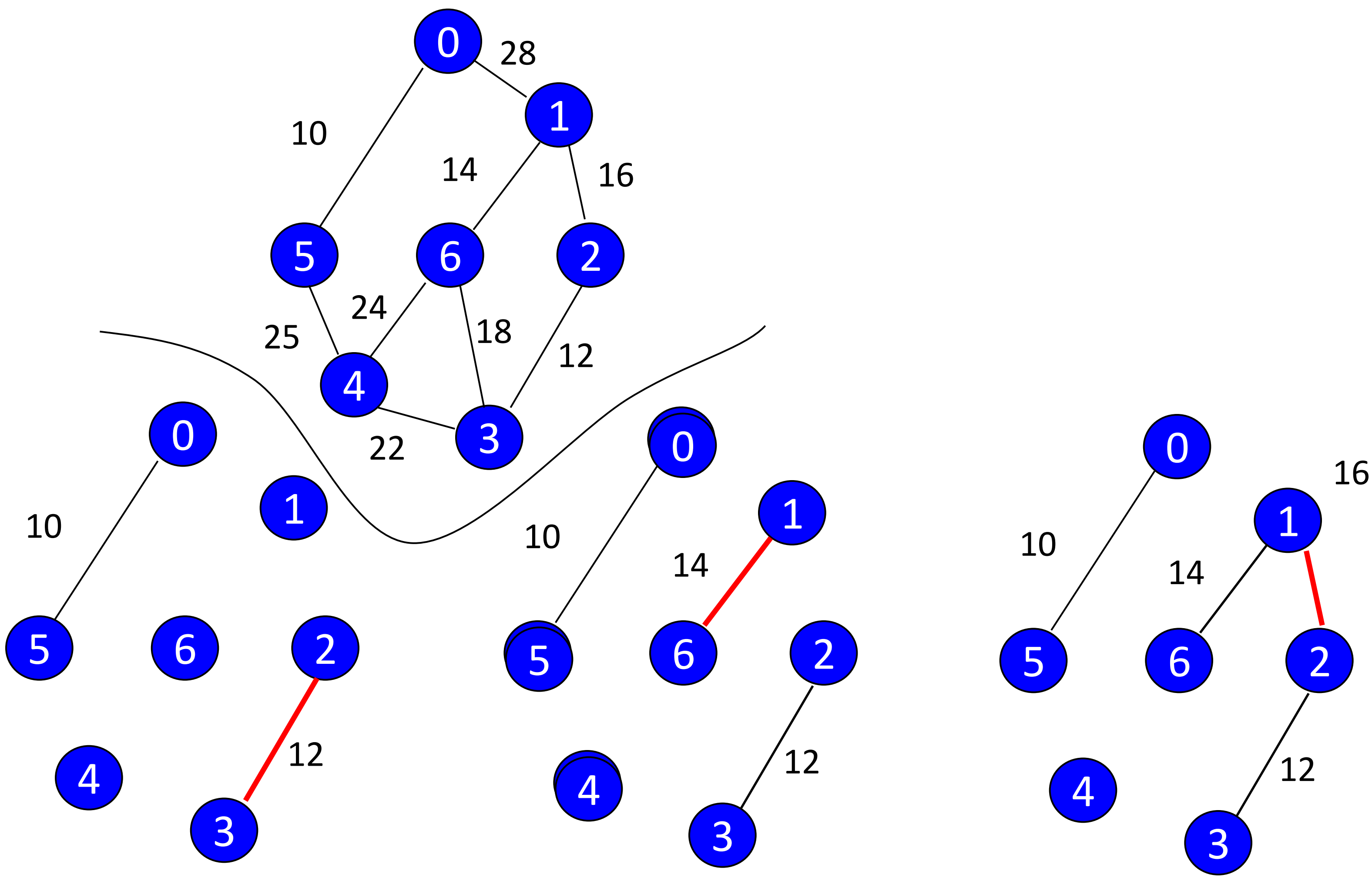
3 —18— 6

3 —22— 4

4 —24— 6

4 —25— 5

0 —28— 1



(d) (e) (f)

Stages in Kruskal's Algorithm (Cont.)

0 10 5

2 12 3

1 14 6

1 16 2

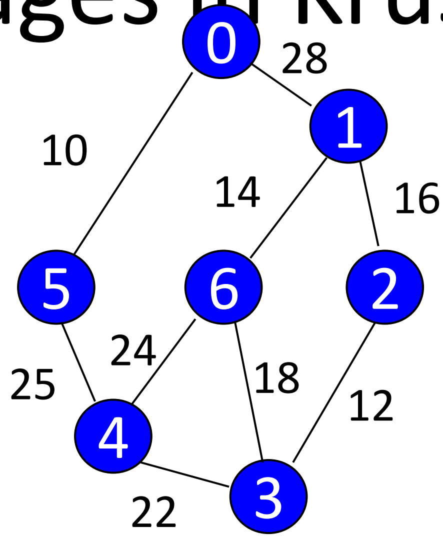
3 18 6

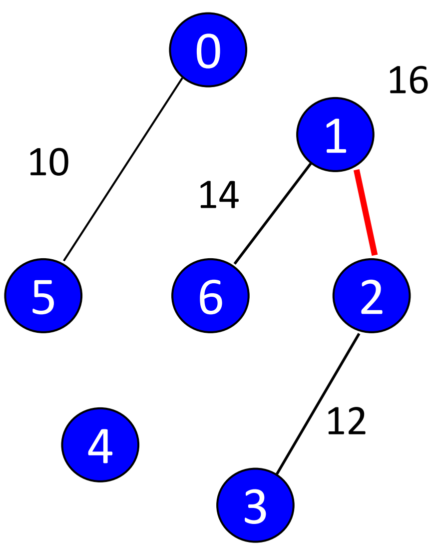
3 22 4

4 24 6

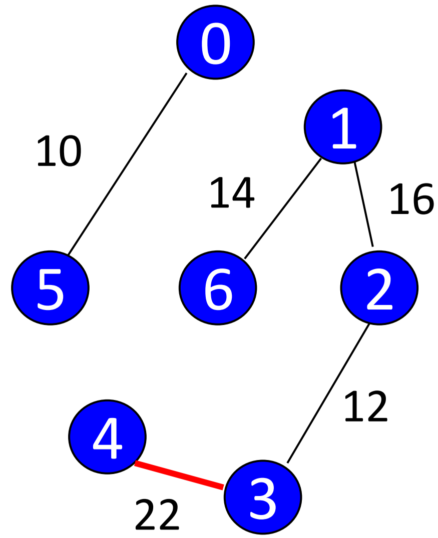
4 25 5

0 28 1

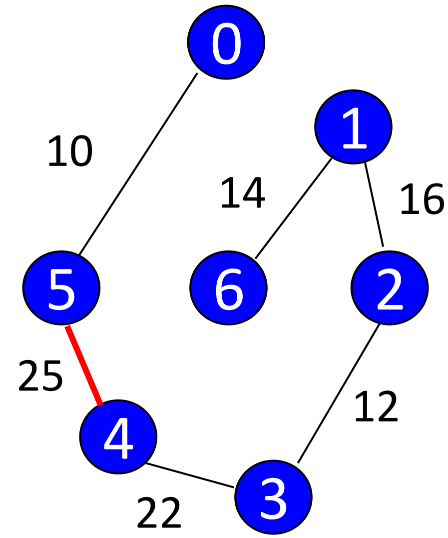




(f)



(g)



(h)

Kruskal's Algorithm

```
T = Φ;
while ( ( T contains less than n – 1 edges) && (E is not empty) ) {
    Choose an edge (v, w) from E of lowest cost;
    Delete (v, w) from E;
    if ( (v, w) does not create a cycle in T) add (v, w) to T;
    else discard (v, w);
}
if ( T contains fewer than n – 1edges ) cout << “no spanning tree” << endl;
```

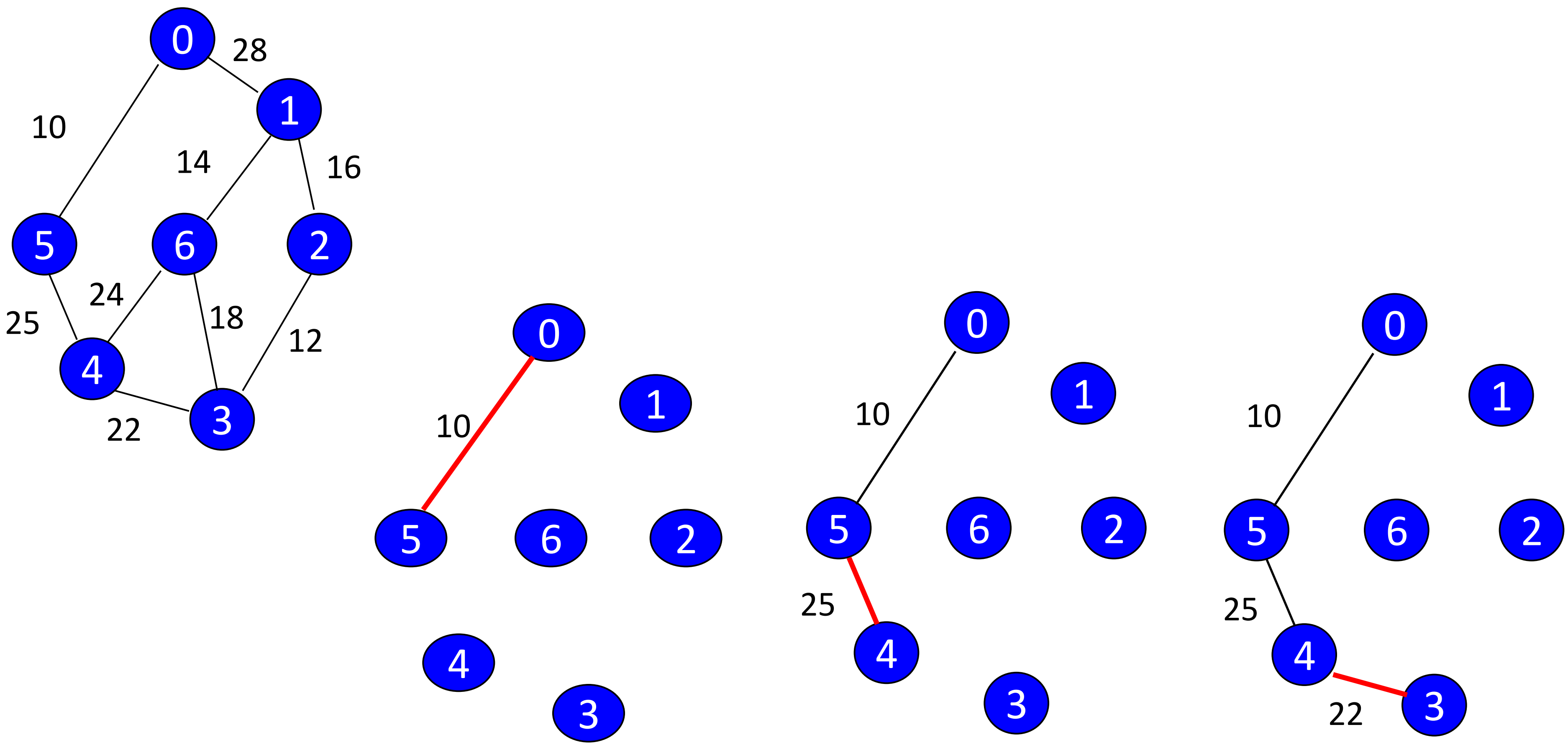
min heap construction time O(e)
choose and delete O(log e)

find find & union O(log e)

Prim's Algorithm

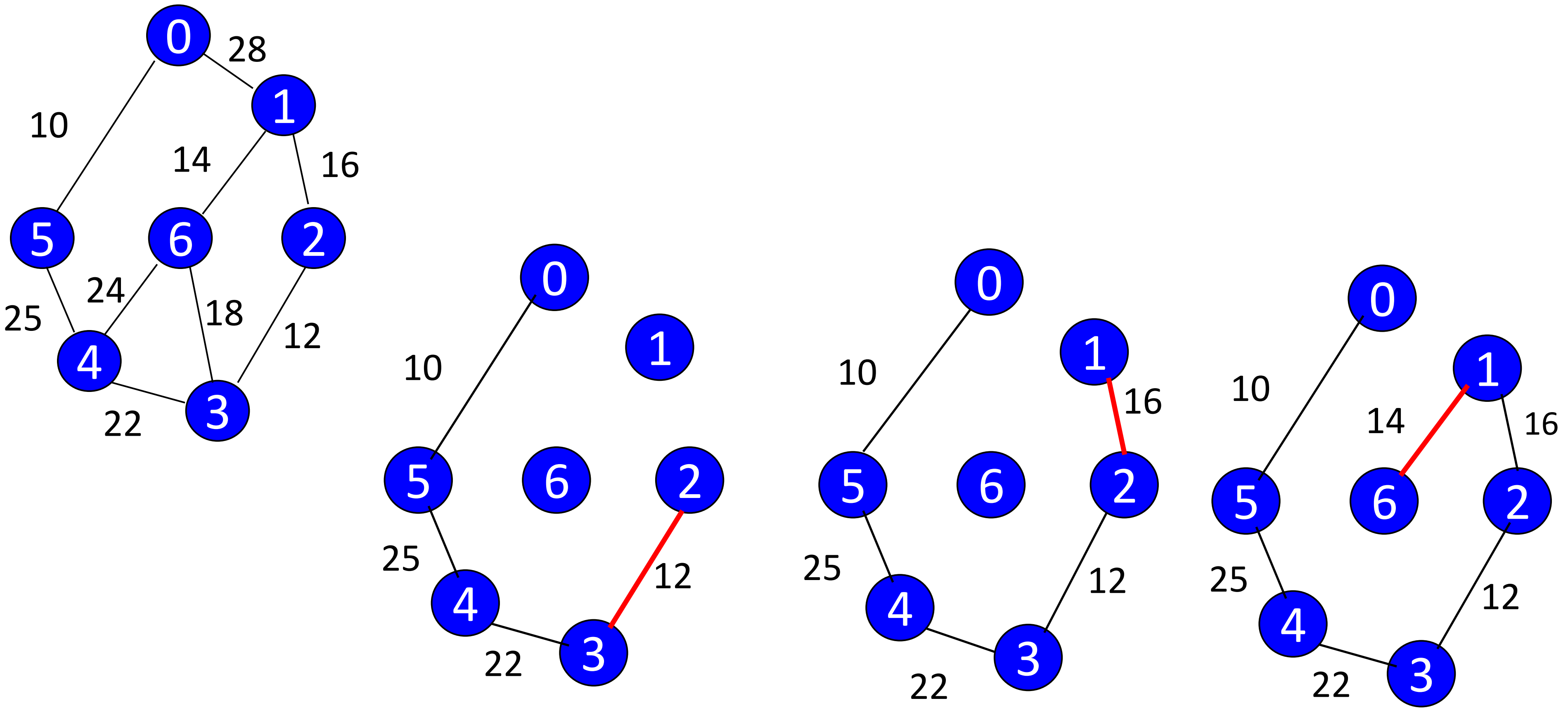
- The set of selected edges forms a tree at all times when using Prim's algorithm
  - In Prim's algorithm, a least-cost edge (u, v) is added to T such that $T \cup \{(u, v)\}$ is also a tree. This repeats until T contains $n - 1$ edges.
- Time complexity
  - $O(n^2)$
  - A faster implementation is possible when **Fibonacci heap** is used

Stages in Prim's Algorithm



(a) (b) (c)

Stages in Prim's Algorithm (Cont.)



(d) (e) (f)

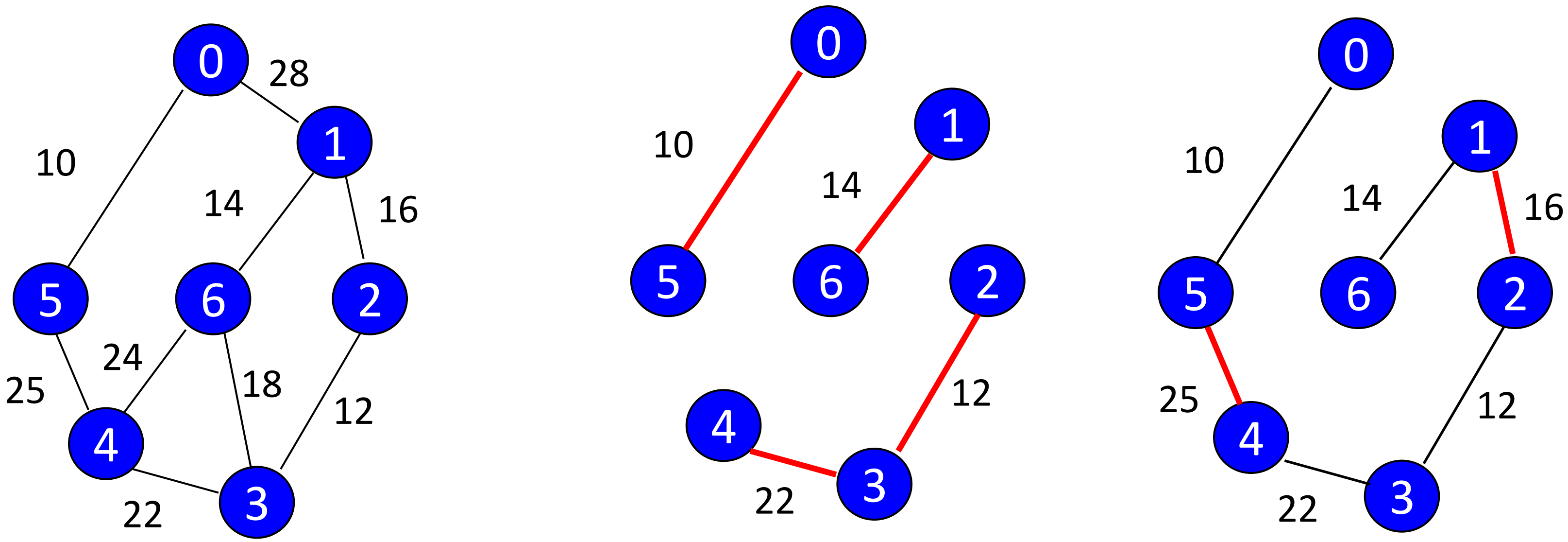
Prim’s Algorithm

```
// assume G has at least one vertex
TV = {0}; // start with vertex 0 and no edges
for (T = Φ ; T contains fewer than n – 1 edges ; add  (u, v) to T)
{
      Let (u, v) be a least-cost edge such that u ∈ TV and v ∉ TV;
      if (there is no such edge) break;
      add v to TV;
}
if ( T contains fewer than n – 1 edges) cout << “no spanning tree” << endl;
```

Sollin’s Algorithm

- Contrast to Kruskal’s and Prim’s algorithms, Sollin’s algorithm selects multiple edges at each stage
- At the beginning, all the n vertices form a spanning forest
- During each stage, an **minimum-cost** edge is selected for each tree in the forest.
  - The edges selected by vertices 0, 1, …, 6 are, respectively, $(0,5)$, $(1,6)$, $(2,3)$, $(3,2)$, $(4,3)$, $(5,0)$
- It’s possible that two trees in the forest to select the same edge. Only one should be used.
- It’s possible that the graph has multiple edges with the same cost. So, two trees may select two different edges that connect them together. Again, only one should be retained.

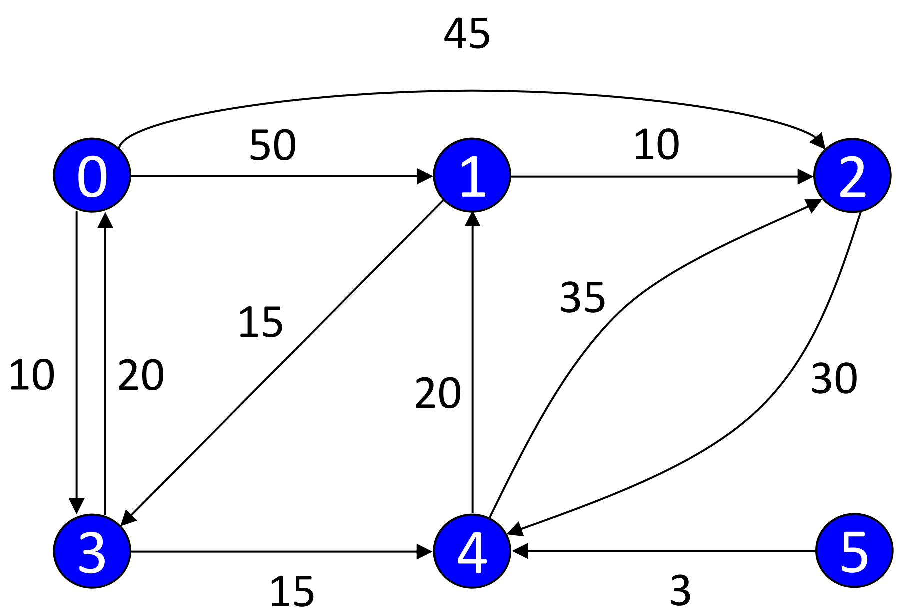
Stages in Sollin's Algorithm



(a) (b)

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

Graph and Shortest Paths From Vertex 0



shortest paths from 0
to all destinations

Path	Length
1) **0**, 3	10
2) **0**, 3, 4	25
3) **0**, 3, 4, 1	45
4) **0**, 2	45

Shortest Paths

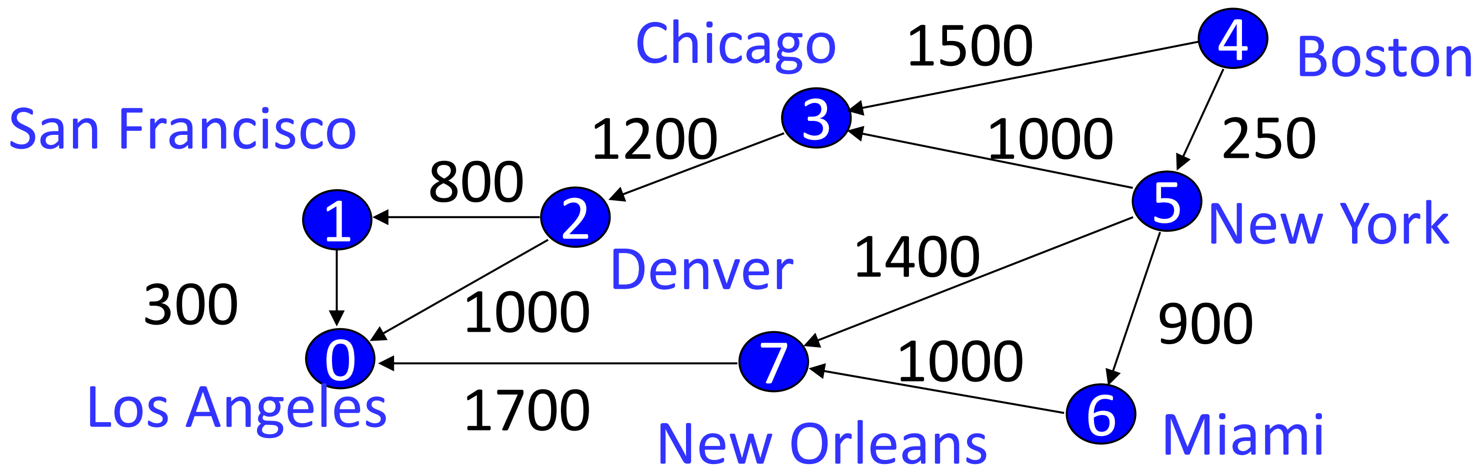
- Single source/all destinations: Nonnegative edges costs.
- Single Source/all destinations: General Weights.
- All-Pairs Shortest Paths

Single Source/All Destinations: Nonnegative Edge Costs

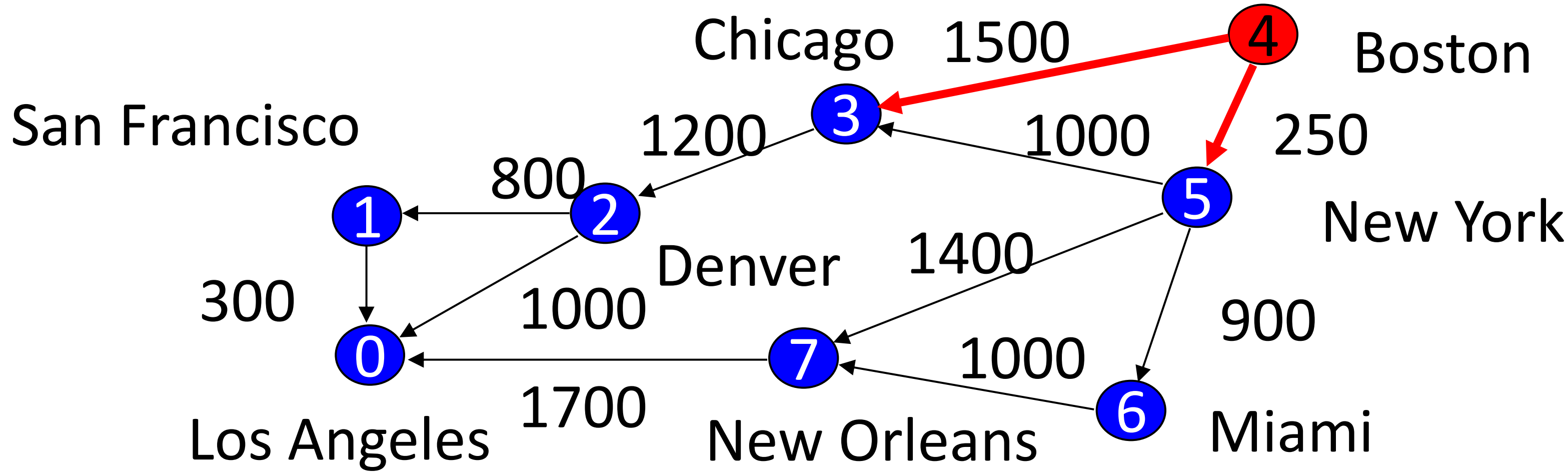
- Let S denote the set of vertices to which the shortest paths have already been found.

1. If the next shortest path is to vertex u, then the path begins at v, ends at u, and goes through only vertices that are in S.
2. The destination of the next path generated must be the vertex u that has the <u>minimum distance</u> among all vertices not in S.
3. The vertex u selected in 2. becomes a member of S.

Example

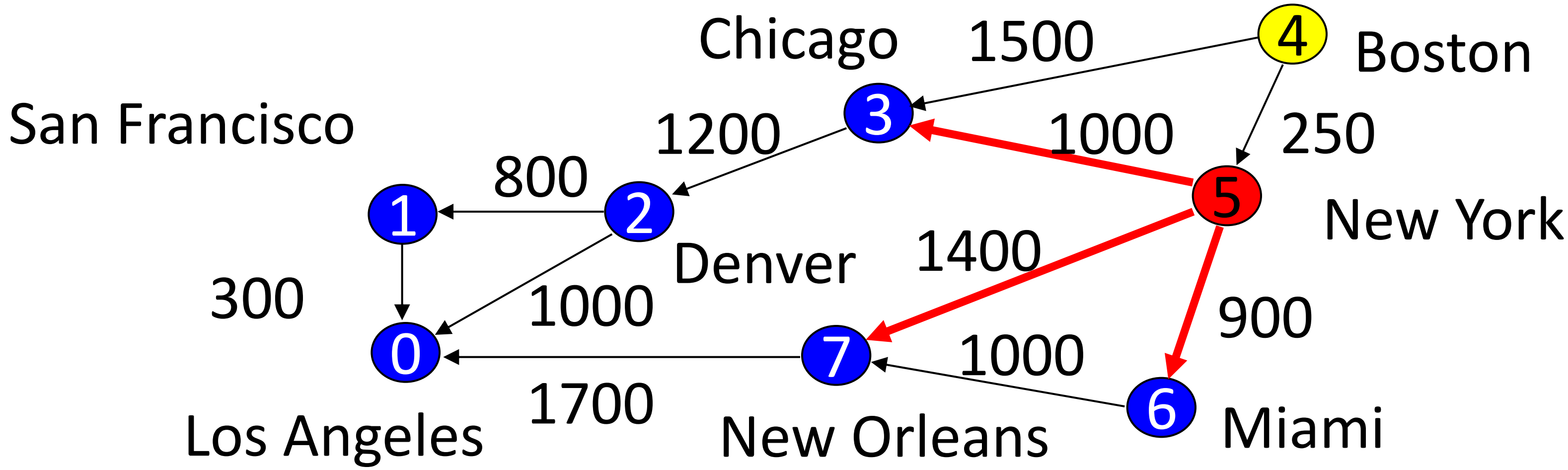


$$
\begin{array}{c}
\\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{bmatrix}
0 & & & & & & & \\
300 & 0 & & & & & & \\
1000 & 800 & 0 & & & & & \\
 & & 1200 & 0 & & & & \\
 &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



$$Distance[3] = \min\{Distance[3], Distance[4] + length[4][3]\}$$
$$= \min\{\infty, 0 + 1500\}$$
$$= 1500$$

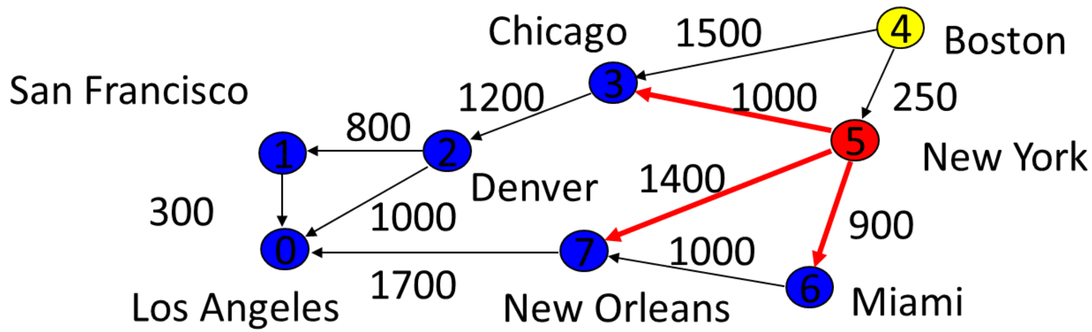
Iteration	Vertex selected	S	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial		{}	+∞	+∞	+∞	+∞	**0**	+∞	+∞	+∞
1	4	{4}	+∞	+∞	+∞	**1500**	0	**250**	+∞	+∞



$$Distance[3] = \min\{Distance[3], Distance[5] + length[5][3]\}$$
$$= \min\{1500, 250 + 1000\}$$
$$= 1250$$

Iteration	Vertex selected	S	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial		{}	+∞	+∞	+∞	+∞	**0**	+∞	+∞	+∞
1	4	{4}	+∞	+∞	+∞	1500	0	**250**	+∞	+∞
2	5	{4,5}	+∞	+∞	+∞	**1250**	0	250	**1150**	**1250**

Action of Shortest Path



Iteration	Vertex selected	S	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	4	{4}	+∞	+∞	+∞	1500	0	**250**	+∞	+∞
1	5	{4,5}	+∞	+∞	+∞	1250	0	250	**1150**	1650
2	6	{4,5,6}	+∞	+∞	+∞	**1250**	0	250	1150	1650
3	3	{4,5,6,3}	+∞	+∞	2450	1250	0	250	1150	**1650**
4	7	{4,5,6,3,7}	3350	+∞	**2450**	1250	0	250	1150	1650
5	2	{4,5,6,3,7,2}	3350	**3250**	2450	1250	0	250	1150	1650
6	1	{4,5,6,3,7,2,1}	**3350**	3250	2450	1250	0	250	1150	1650

ShortestPath()

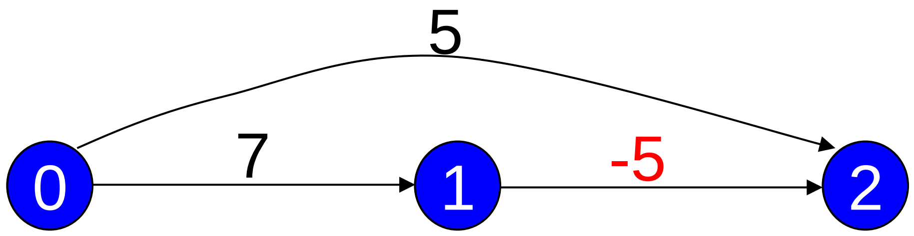
```
void MatrixWDigraph::ShortestPath(const int n, const int v)
{ // dist[j], 0 ≤ j < n, is set to be the length of the shosrtest apth from v to j
  // in a directed greaph G contains n vertices and edge lenths given by length[i][j]
    for (int i = 0; i < n ; i++) { s[i] = false; dist[i] = length[v][i];} // initialize
    s[v] = true;
    dist[v] = ∞;

    for (i = 0; i < n−2 ; i++) { // determine n – 1paths from vertex v
        int u = Choose (n);    // Choose return a value u such that:
                               // dist[u] = minimum dist[w], where s[w] = false
        s[u] = true;
        for (int w = 0; w < n ; w++)
            if (! s[w] && dist[u] + length[u][w] < dist[w])
              dist[w] = dist[u] + length[u][w];
    } //end of for (i = 0; ...)
}
```

Single Source/All Destinations: Nonnegative Edge Costs (contd.)

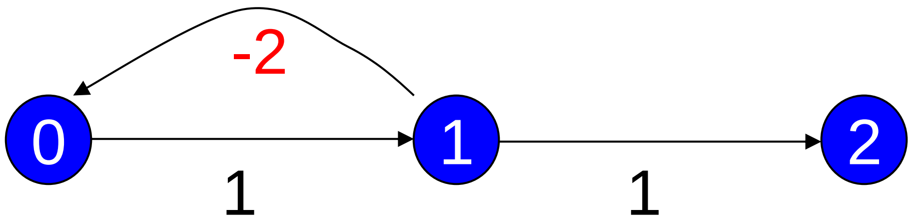
- The algorithm is first given by Edsger Dijkstra. Therefore, it's sometimes called Dijkstra Algorithm.
- Time complexity
  - Adjacency matrix, adjacency list: $O(n^2)$
  - Using Fibonacci heap: $O(n \log n + e)$

Directed Graphs /w Negative Length

- When negative edge lengths are permitted, we require that the graph have no cycles of negative length



(a) Directed graph with a negative-length edge



(b) Directed graph with a cycle of negative length

Single Source/All Destinations: General Weights

- When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n-1$ edges on it.
  - If the shortest path from v to u with at most $k, k > 1$, edges has no more than k– 1 edges, then $disk^{k}[u] = disk^{-1}[u]$.
  - If the shortest path from v to u with at most $k, k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex i followed by the edge $< i, u >$. The path from v to i has k– 1 edges, and its length is $disk^{k-1}[i]$.

Single Source/All Destinations: General Weights (contd.)

- The distance can be computed in recurrence by the following:

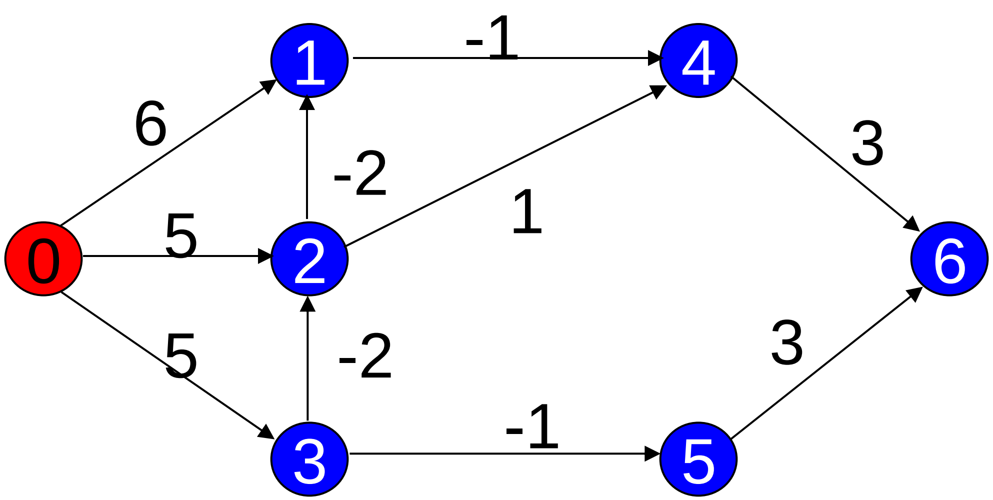
$$disk^k[u] = \min\{\, disk^{k-1}[u], \min_i disk^{k-1}[i] + length[i][j]\}$$

- The algorithm is also referred to as the Bellman-Ford Algorithm.
- Time complexity:
  - Adjacency matrix: $O(n^3)$
  - Adjacency list: $O(ne)$

Bellman-Ford Algorithm

```
void MatrixWDigraph::BellmanFord(const int n, const int v)
{ // single source all destination shortest paths with negative edge lenths.
    for (int i = 0; i < n ; i++) dist[i] = length[v][i]; // initialize dist
    for (int k = 2; i <= n−1 ; k++)
        for (each u such that u != v and u has at least one incomeing edge)
            for (each    <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u]) dist[u] = dist[i] + length[i][u];
}
```

Shortest Paths with Negative Edge Lengths



(a) A directed graph

k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2							
3							
4							
5							
6							

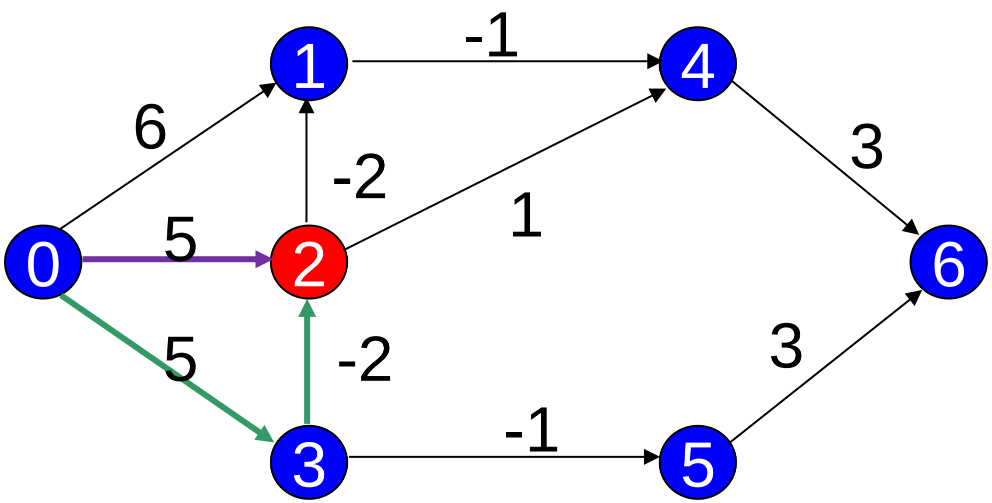
(b) $dist^k$

$\underline{dist^2[2]}$

1. $dist^{k-1}[u]$
   - $dist^1[2] = 5$

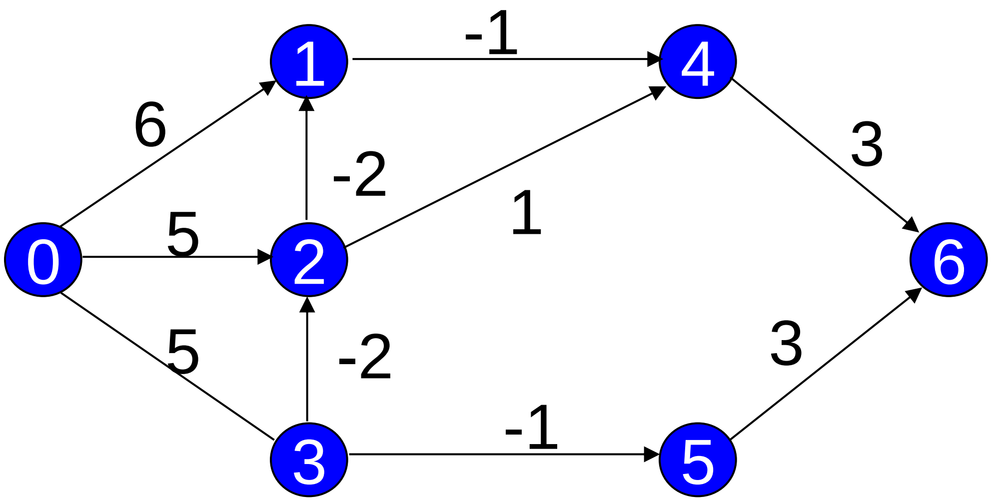
2. $dist^{k-1}[i] + length[i][u]$
   - $dist^1[0]{+}length[0][2] = 5$
   - $dist^1[3]{+}length[3][2] = 5 - 2 = 3$

$\min(5,5,3) = 3$



k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2			3				
3							
4							
5							
6							

Shortest Paths with Negative Edge Lengths (contd.)

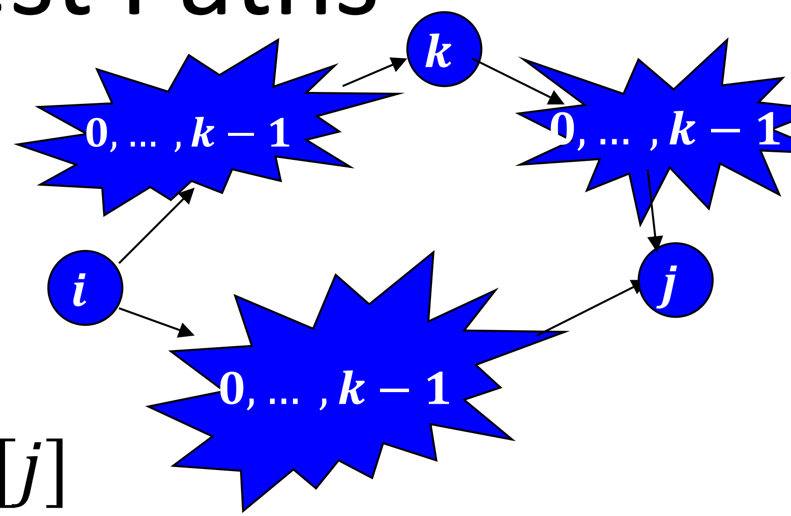


k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	**3**	**3**	5	5	4	∞
3	0	**1**	3	5	**2**	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

All Pairs Shortest Paths

- Find the shortest paths between all pairs of vertices.
  - Solution 1: Apply shortest path n times with each vertex as source. $O(n^3)$
  - Solution 2
    - Represent the graph G by its cost adjacency matrix with $length[i][j]$
    - If the edge $< i, j >$ is not in G, the $length[i][j]$ is set to some sufficiently large number
    - $A[i][j]$ is the cost of the shortest path form i to j, using only those intermediate vertices with an index $\leq\ k$

All-Pairs Shortest Paths



- Floyd-Warshall algorithm
- Notations
  - $A^{-1}[i][j]$: is just the $length[i][j]$
  - $A^{n-1}[i][j]$: the length of the shortest i-to-j path in G
  - $A^{k}[i][j]$: the length of the shortest path from i to j going through no intermediate vertex of index greater than k.

All-Pairs Shortest Paths (contd.)

- How to determine the value of $A^k[i][j]$?

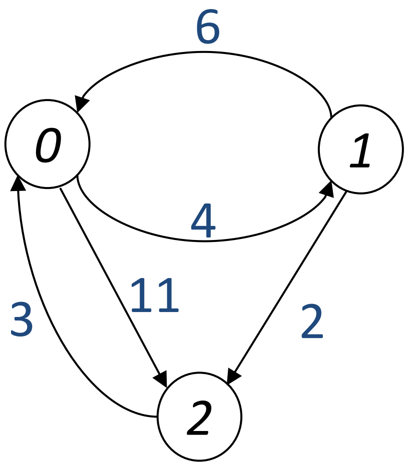
$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$$

- Calculate the $A^0, A^1, A^2, \ldots, A^{n-1}$ from A^{-1} iterativelly
- Time complexity
  - $O(n^3)$

All-Pairs Shortest Paths (contd.)

```
void MatrixWDigraph::AllLengths(const int n)
{// length[n][n] is the adjacency matrix of a graph with n-vertices
// a[i][j] is the shortest path between i and j
    for (int i = 0; i<n; i++)
      for (int j = 0; j<n; j++)
          a[i][j]= length[i][j];      // copy length into a
    for (int k= 0; k<n; k++)    // for a path with highest vertex index k
      for (i= 0; i<n; i++)          // for all possible pairs of vertices
        for (int j= 0; j<n; j++)
          if((a[i][k]+a[k][j])<a[i][j])a[i][j] = a[i][k] + a[k][j];
 }
```

Example for All-Pairs Shortest-Paths Problem



A^{-1}

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

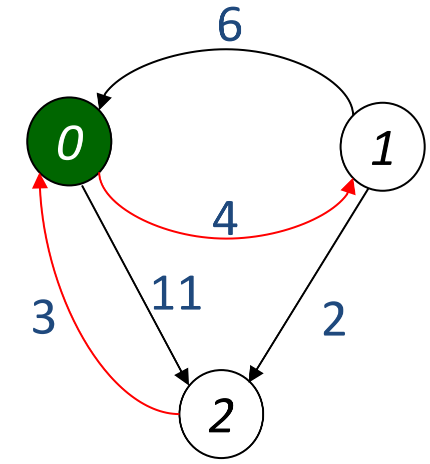
A^{0}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

Compute $A^0[2][1]$:

$$A^{-1}[2][1] = \infty$$
$$A^{-1}[2][0] + A^{-1}[0][1] = 3 + 4 = 7$$

$$\min\{\infty, 7\} = 7$$



Example for All-Pairs Shortest-Paths Problem

A^0

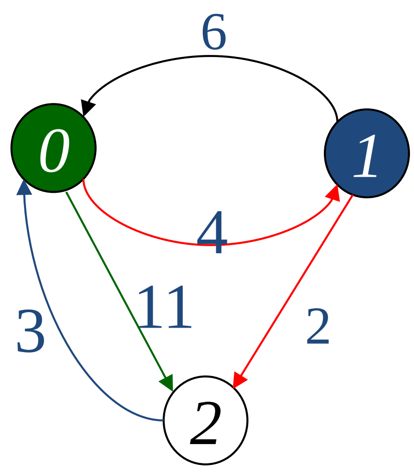
A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

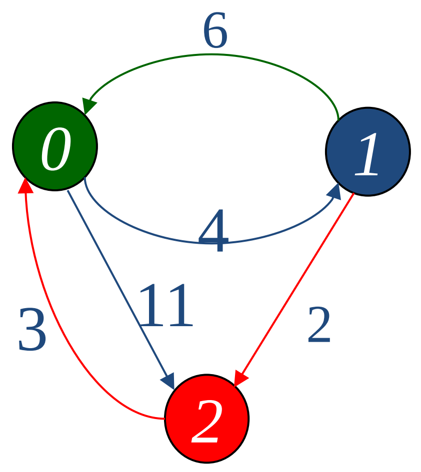
A^1

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^2

A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

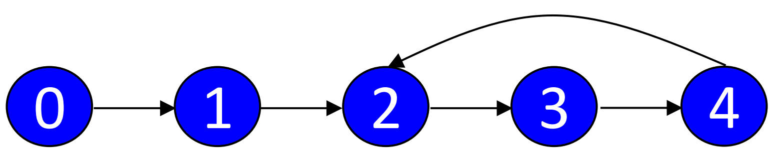




Transitive Closure

- **Definition**: The **transitive closure matrix**, denoted A^+, of a graph G, is a matrix such that $A^+[i][j] = 1$ if there is a path of length > 0 from i to j; otherwise, $A^+[i][j] = 0$.
- **Definition**: The **reflexive transitive closure matrix**, denoted A^*, of a graph G, is a matrix such that $A^*[i][j] = 1$ if there is a path of length ≥ 0 from i to j; otherwise, $A^*[i][j] = 0$.

Graph G and Its Adjacency Matrix A, A^+, A^*



(a) Digraph G

$$
\begin{array}{c c}
 & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}
\end{array}
$$

(b) Adjacency matrix A

Graph G and Its Adjacency Matrix A, A^+, A^*

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

(c) A^+

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

(d) A^*

Activity-on-Vertex (AOV) Networks

- **Definition:** Activity-On-Vertex network (AOV network)
  - A directed graph G
  - the vertices represent tasks or activities
  - the edges represent precedence relations between tasks.
- **Definition:** Vertex i in an AOV network G is a predecessor of vertex j iff there is a directed path from vertex i to vertex j.
  - i is an immediate predecessor of j iff $< i, j >$ is an edge in G.
  - If i is a predecessor of j, then j is an successor of i.
  - If i is an immediate predecessor of j, then j is an immediate successor of i.

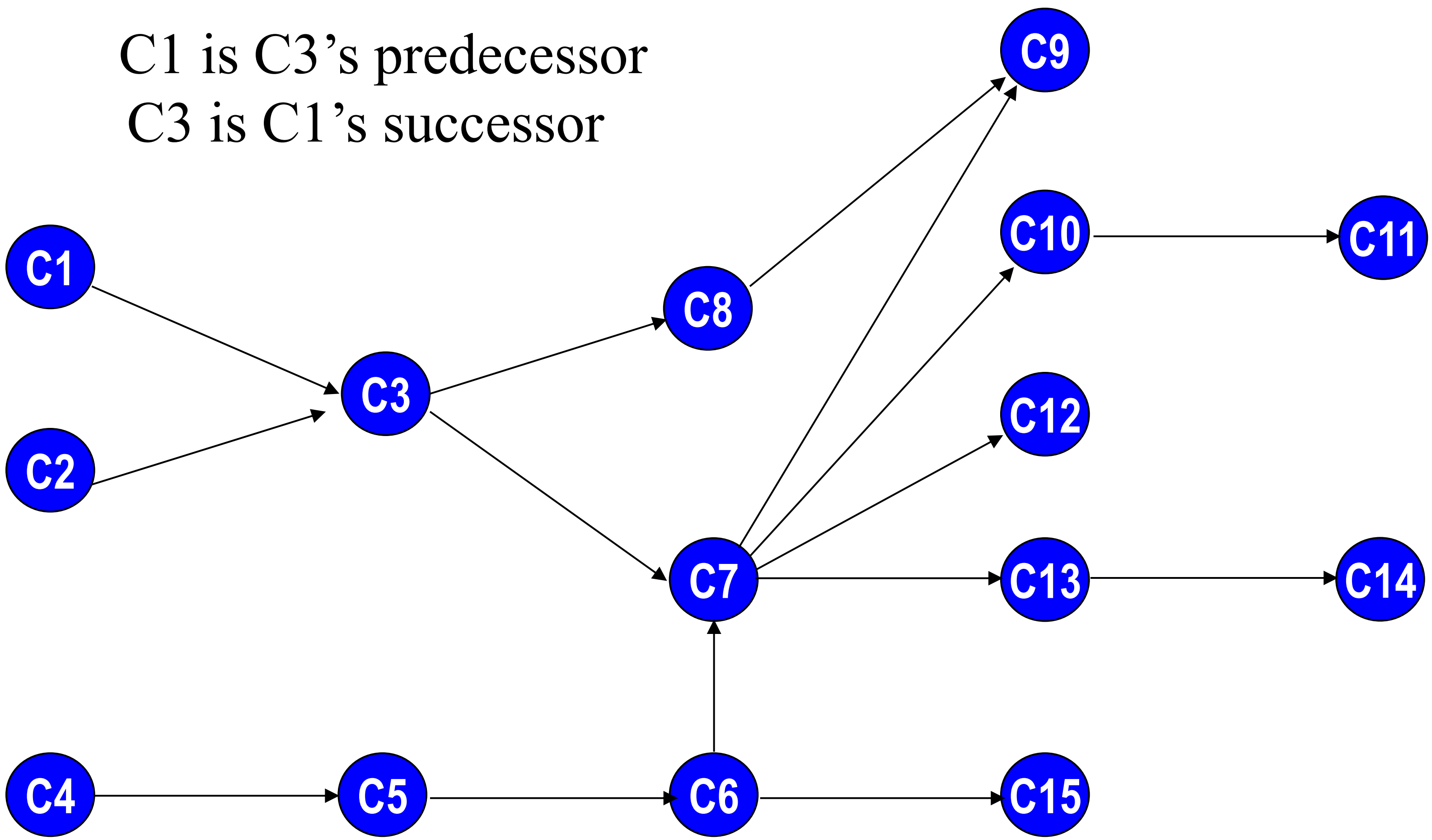
Activity-on-Vertex (AOV) Networks (contd.)

- **Definition**: A *topological order* is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering.

An Activity-on-Vertex (AOV) Network

Course number	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	C5

An Activity-on-Vertex (AOV) Network (Cont.)



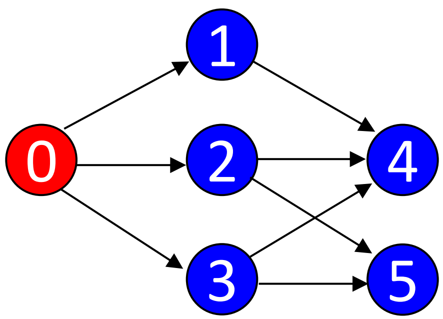
Topological Sorting Algorithm

```
input the AOV network. Let n be the no. of vertices.
for (int i = 0; i<n; i++) // output the vertices
{
  if (every vertex has a predecessor) return;
    // twork hass a cycle and is infeasible
  pick a vertex v that has no predecessor
  count << v;
  delete v and all edges leading out of v from the network
}
```

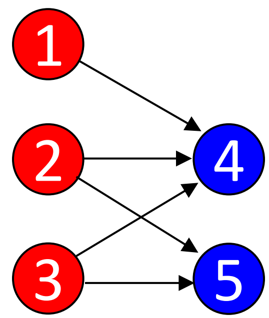
Finding topological order on an AOV network

0, 3, 2, 5,1, 4

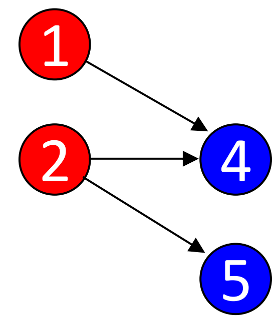
- Pick a vertex v that has no predecessors (i.e., $in-degree = 0$)



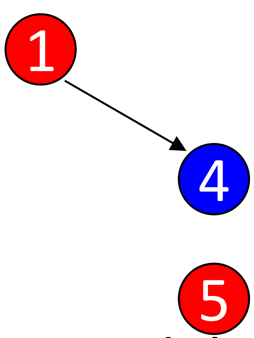
(a) Initial



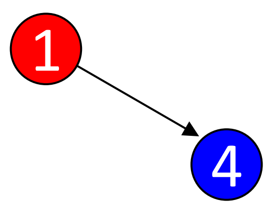
(b) Vertex 0 deleted



(c) Vertex 3 deleted



(d) Vertex 2 deleted

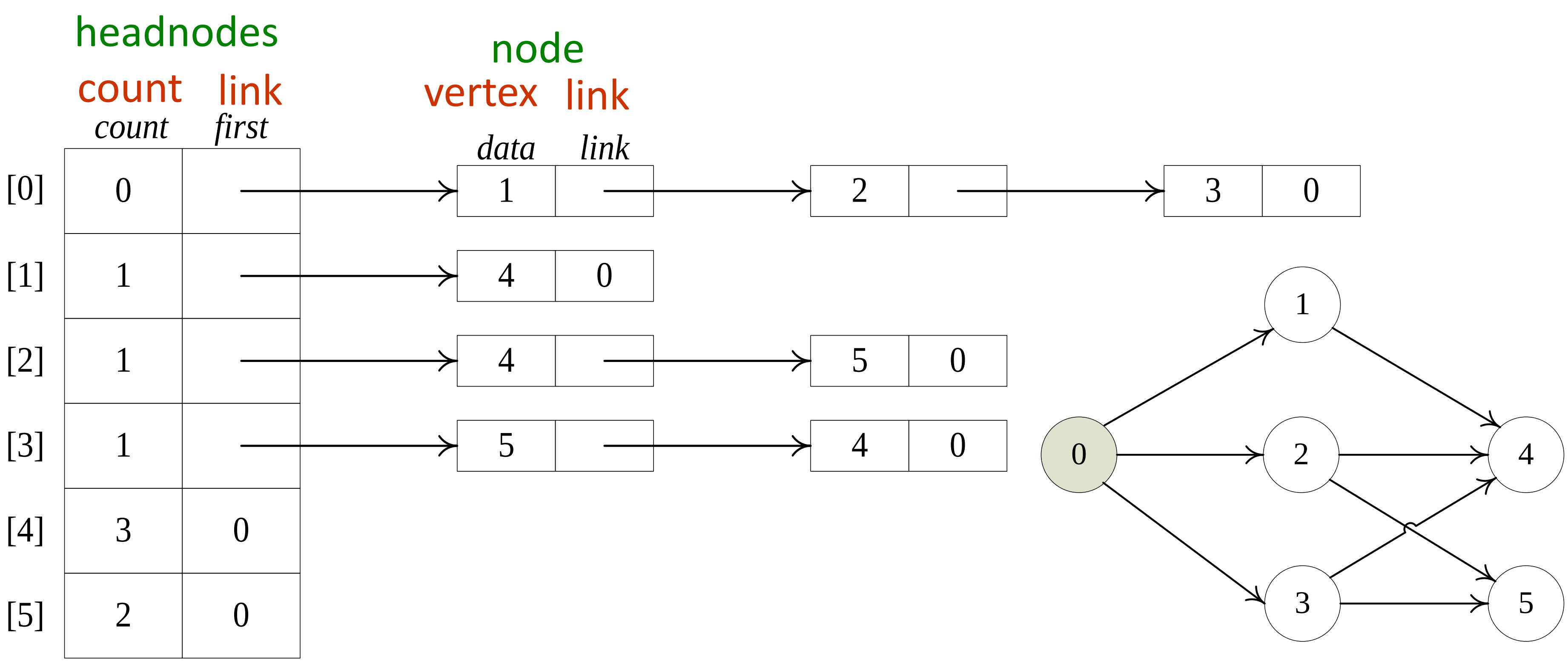


(e) Vertex 5 deleted

(f) Vertex 1 deleted

Issues in Data Structure Consideration

- Decide whether a vertex has any predecessors.
  - Each vertex has a count.
- Decide a vertex together with all its incident edges.
  - Adjacency list



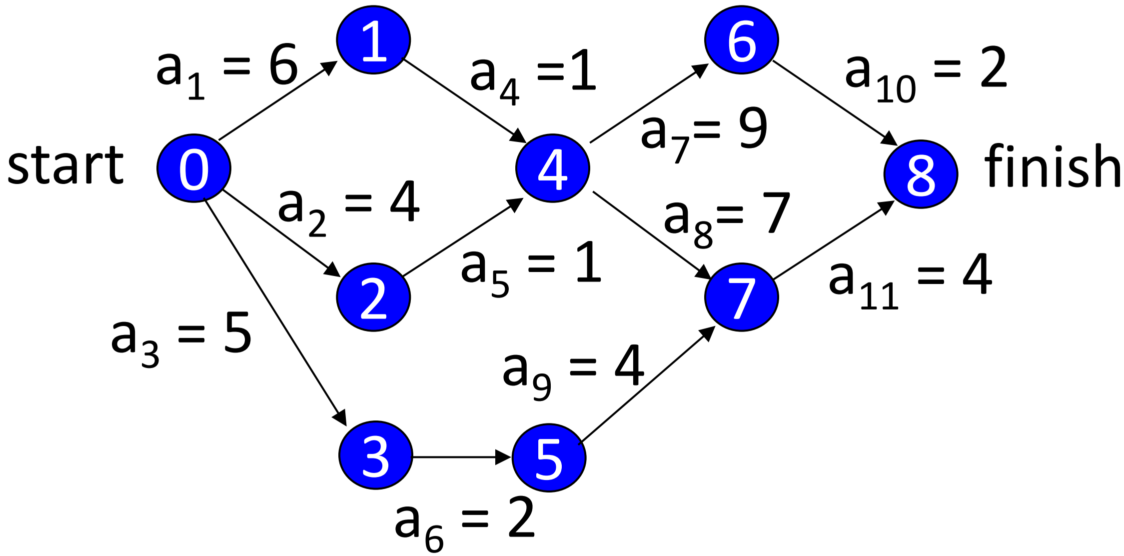
TopologicalOrder()

```
void LinkDigraph::TopologicalOrder()
{// the n vertices of a network are listed in topological order
    int top = -1;
    for (int i = 0; i<n; i++) //generate a linked stack of vertices with no predecessors
        if (count[i] == 0) { count[i] = top; top = i;}
    for (i = 0; i< n; i++)
        if (top = =-1) throw "Network has a cycle.";
        int j = top; top = count[top];    //unstack a vertex
        count << j <<endl;
        Chain<int>::ChainIterator ji = adjLists[j].begin();
        while (ji) { // decrease the count of the successor vertices of j
            count[*ji]--;
            if (count[*ji] == 0) { count[*ji] = top; top = *ji;}    // add *ji to stack
            ji++;
        }
}
```

Activity on Edge (AOE) Networks

- Activity on edge, or AOE, network is an activity network closely related to the AOV network.
- The directed edges in the graph represent tasks or activities to be performed on a project.
  - directed edge
    - tasks or activities to be performed
  - vertex
    - events which signal the completion of certain activities
  - number
    - time required to perform the activity

An AOE Network



Edge: activity
Vertex: event

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

An AOE Network (contd.)

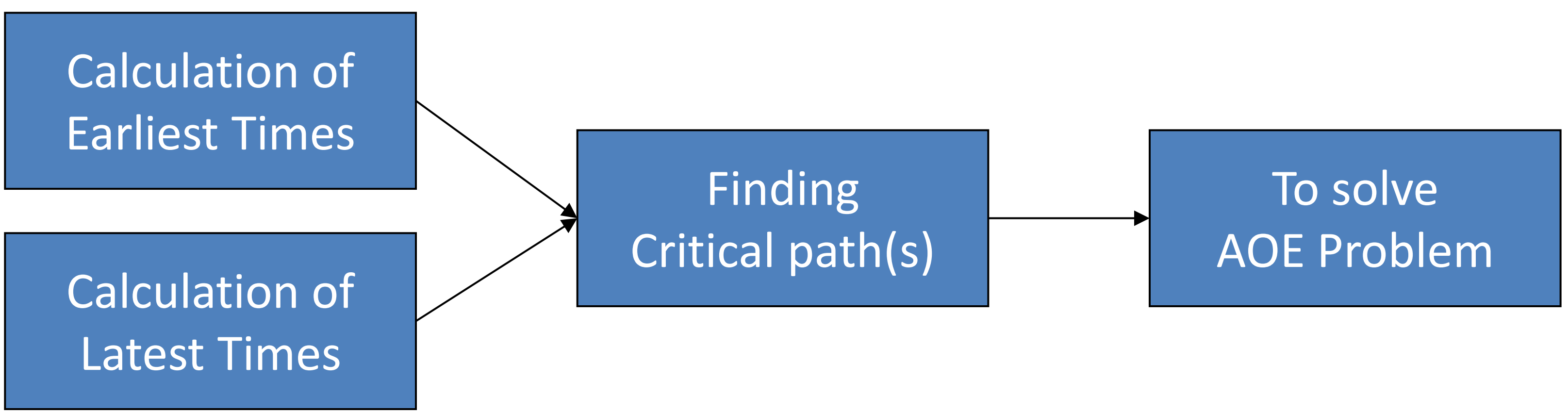
- A path of the longest length is a *critical path*
- The *earliest time* that an event i can occur is the length of the **longest** path from the start vertex 0 to the vertex i
- The earliest time an event can occur determines the ***earliest start time*** for all activities (i.e., $e(i)$) represented by edges leaving that vertex
- For every activity a_i, the *latest time*, $l(i)$, that an activity may start without increasing the project duration

An AOE Network (contd.)

- All activities for which $e(i) = l(i)$ are called *critical activities*
- Earliest event time: $ee[j]$
- Latest event time: $le[j]$
- Activity a_i is represented by edge $< k, l >$
  - $e(i) = ee[k]$
  - $l(i) = le[l] - duration\ of\ activity\ a_i$

Critical Activity

- A critical activity is an activity for which $e(i) = l(i)$.
- The difference between $e(i)$ and $l(i)$ is a measure of how critical an activity is.



An AOE Network (contd.)

- Calculation of $ee[j]$ and $le[j]$
  - $P(j)$ is the set of all vertices adjacent to vertex j
  - $S(j)$ is the set of all vertices adjacent from vertex j

$$ee[0] = 0$$

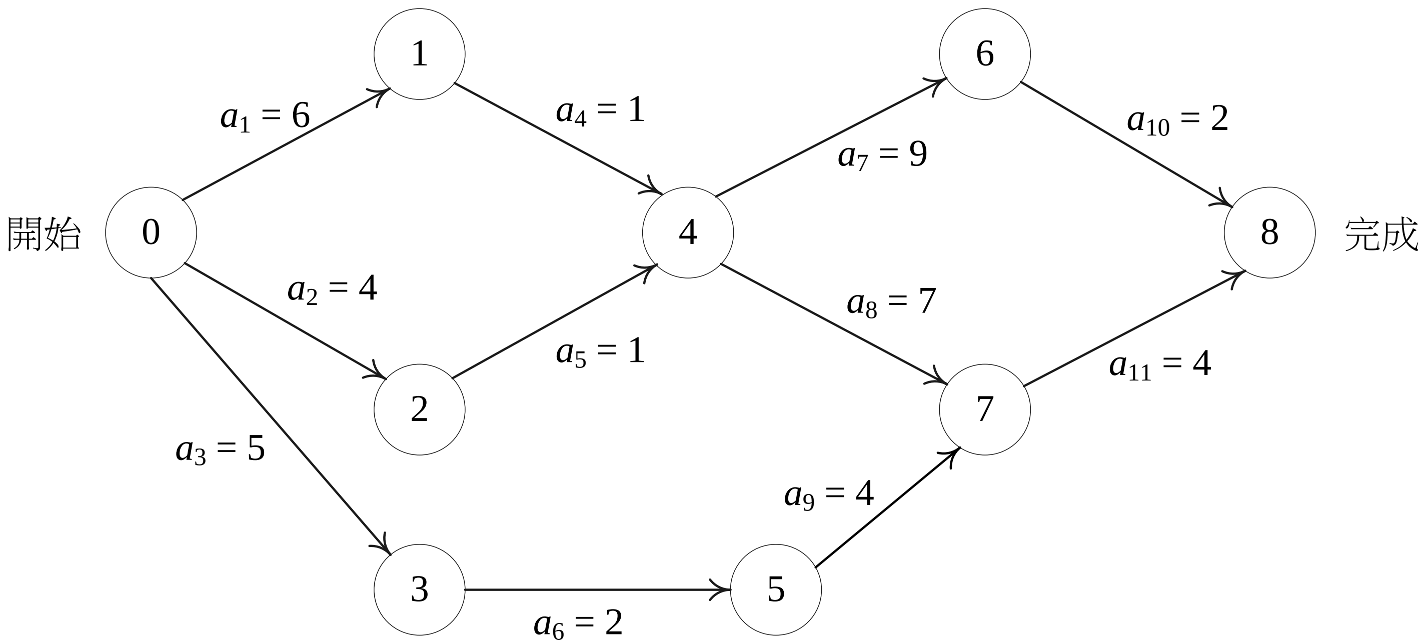
$$ee[j] = \max_{i \in P(j)} \{ee[i] + duration\ of < i, j >\}$$

$$le[n-1] = ee[n-1]$$

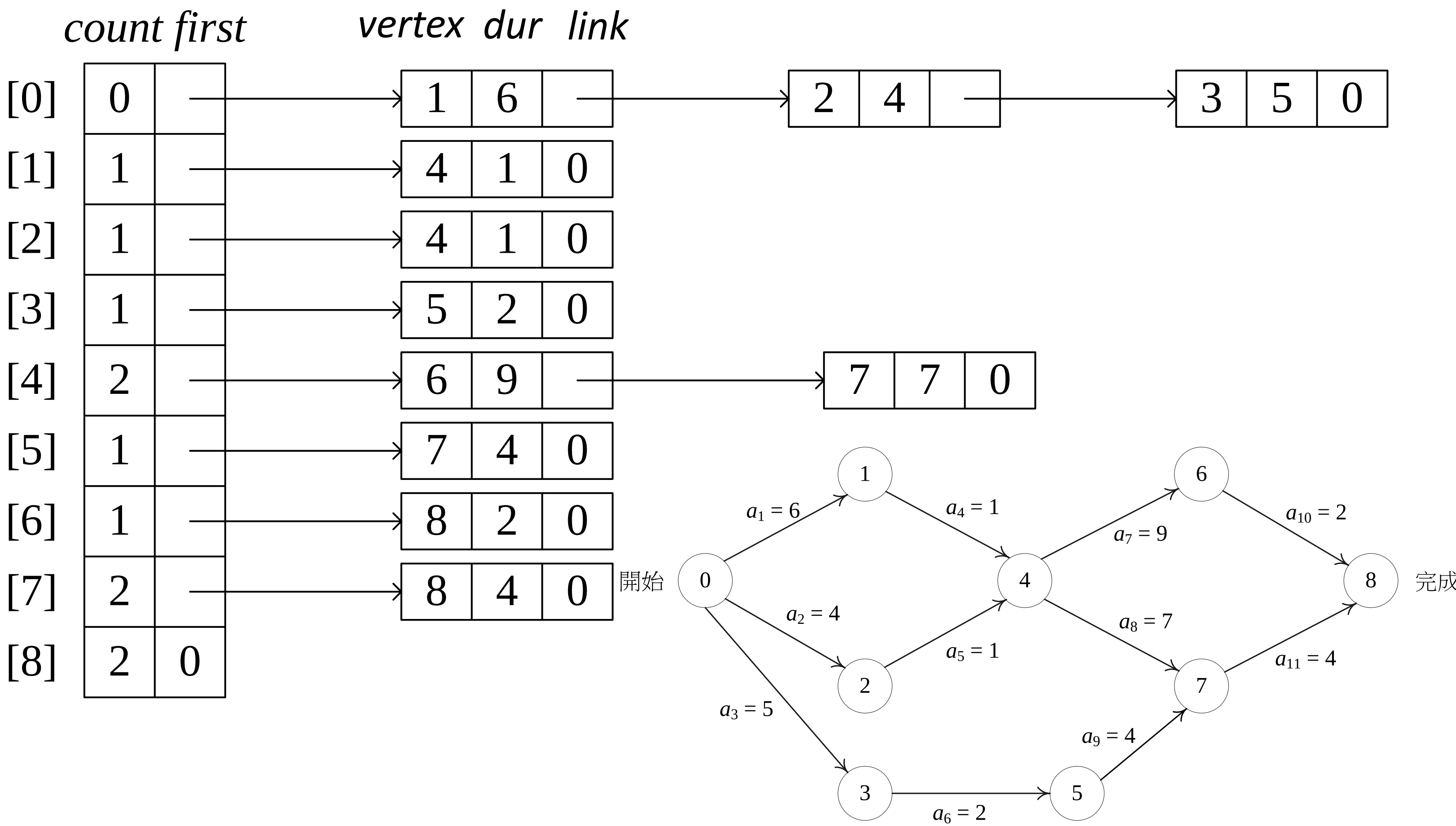
$$le[j] = \min_{i \in S(j)} \{le[i] - duration\ of < j, i >\}$$

- Using topological order

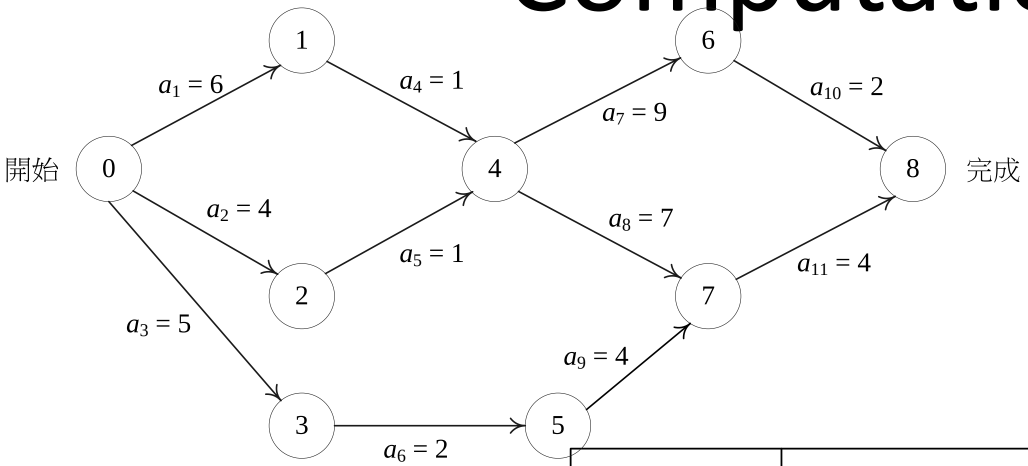
Example: Computing earliest from topological sort



Adjacency Lists for Figure 6.38 (a)



Computation of ee



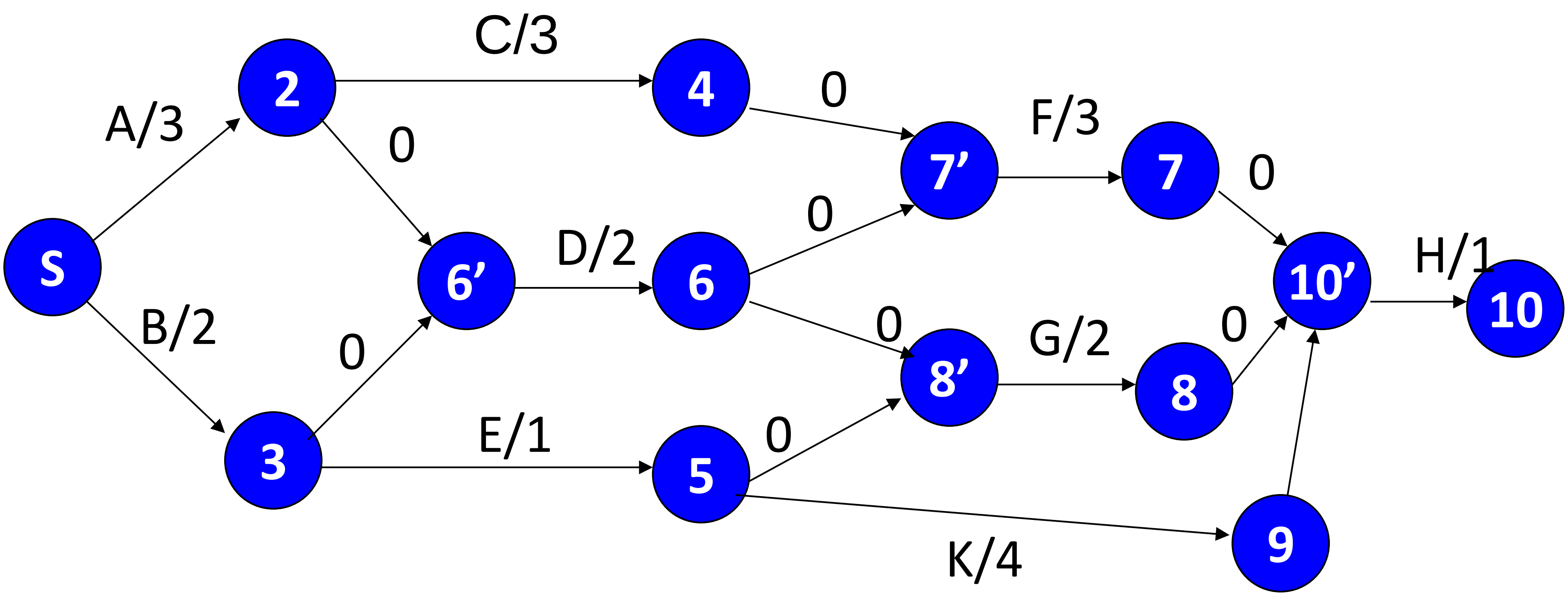
In topological sorting, the vertices wi
$in - degree = 0$ are placed in stack

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	stack
start	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										

Critical Path Analysis (cont.)

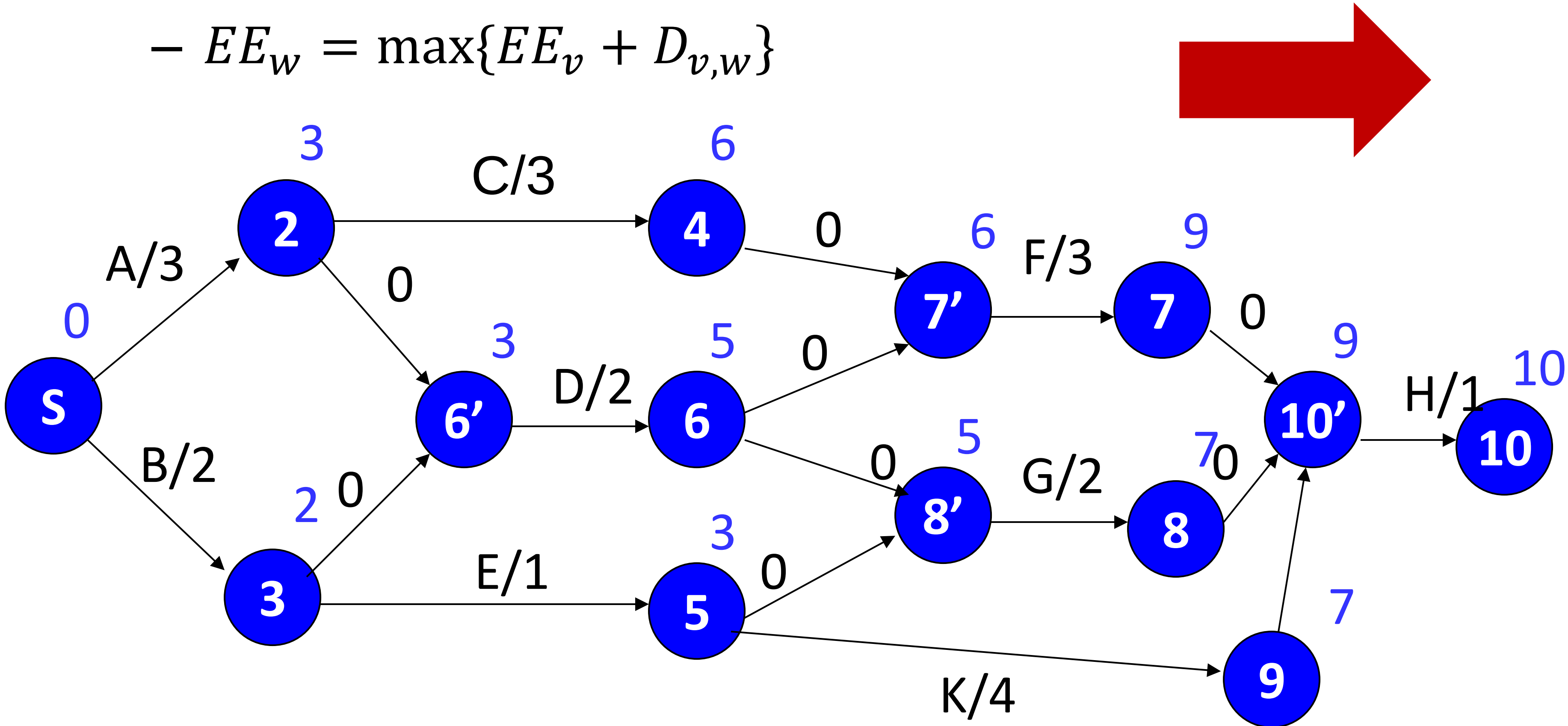
- AOE graph

edge ID/cost



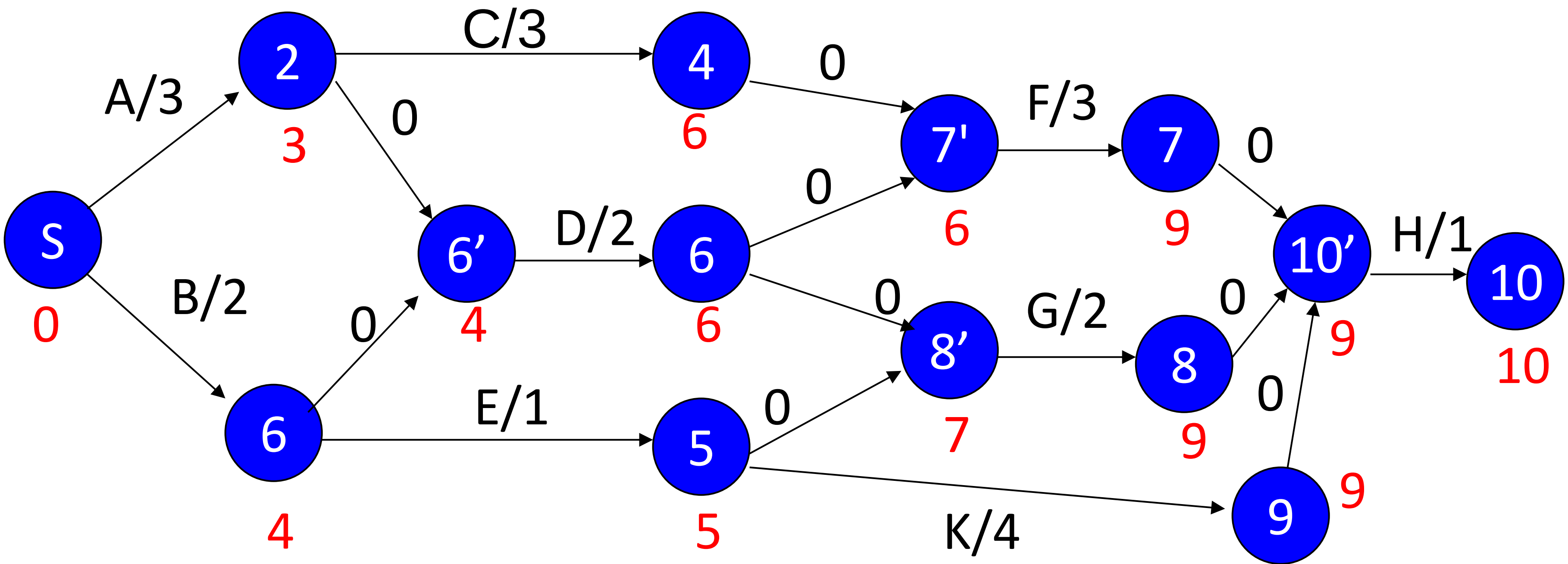
Critical Path Analysis (cont.)

- Earliest completion times: longest path
  - computed by topological order
  - $EE_1 = 0$
  - $EE_w = \max\{EE_v + D_{v,w}\}$



Critical Path Analysis (cont.)

- Latest completion times:
  - latest time without affecting final completion time
  - computed by reverse topological order
  - $LE_{10} = EE_{10}$
  - $LE_V = \min\{LE_W - D_{V,W}\}$



Critical Path Analysis (cont.)

- $Slack\ time(v,w) = LE_w - EEw$
- Critical path = zero slack time