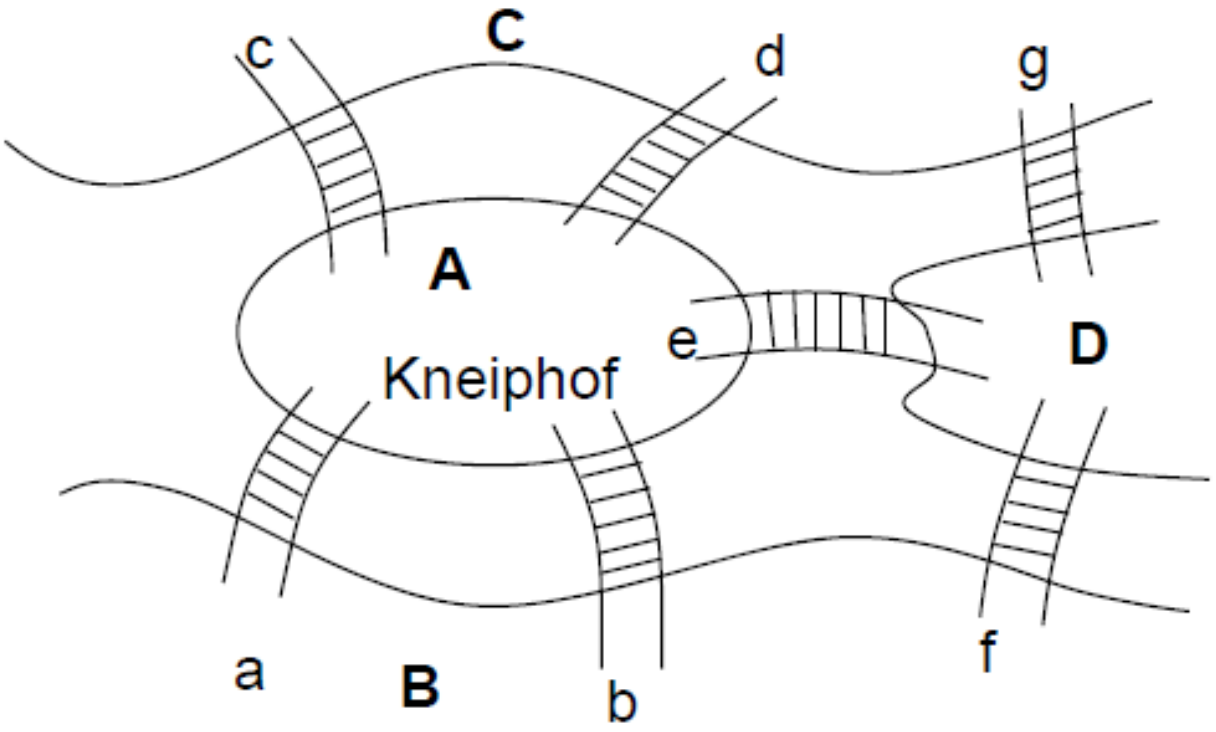
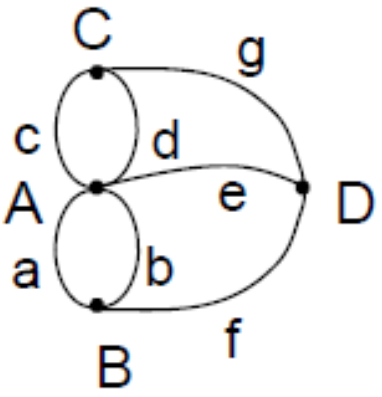
DATA STRUCTURE

Chapter 6: 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.

APPLICATIONS OF GRAPHS

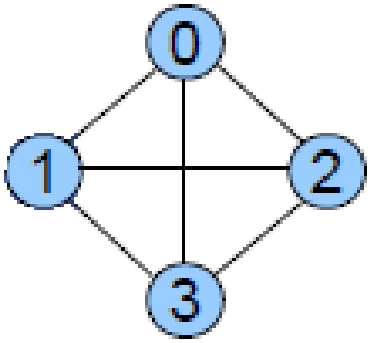
- Analysis of electrical circuits
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Statistical mechanics
- Genetics
- Cybernetics
- Linguistics
- Social Sciences

DEFINITION OF A GRAPH

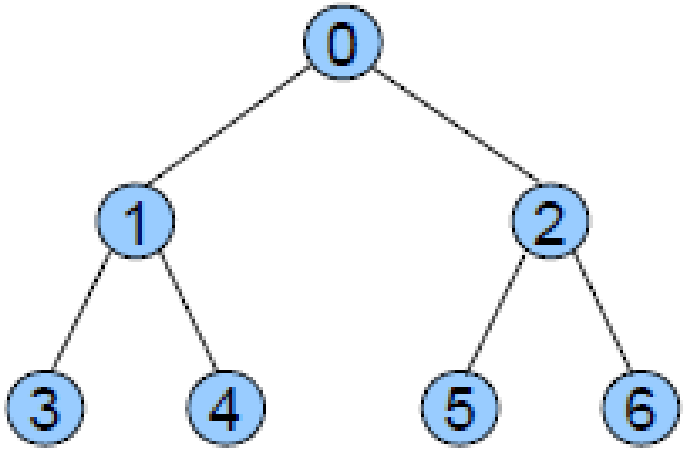
- A graph, G=(V, E), consists of two sets, V and E.
  - V is a finite, nonempty set of vertices.
  - E is set of pairs of vertices called edges.
- The vertices of a graph G can be represented as V(G).
- Likewise, the edges of a graph, G, can be represented as E(G).

- Graphs can be either undirected graphs or directed graphs.
- For a undirected graph, a pair of vertices *(u, v)* or *(v, u)* represent the same edge.
- For a directed graph, a directed pair <*u, v*> has *u* as the tail and the *v* as the head. Therefore, <*u, v*> and <*v, u*> represent different edges.

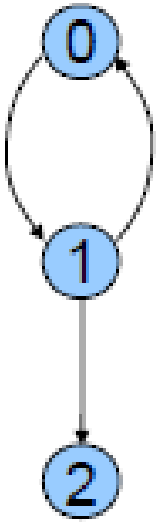
THREE SAMPLE GRAPHS



G_1



G_2

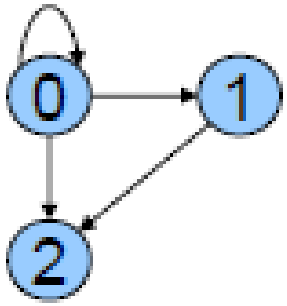


G_3

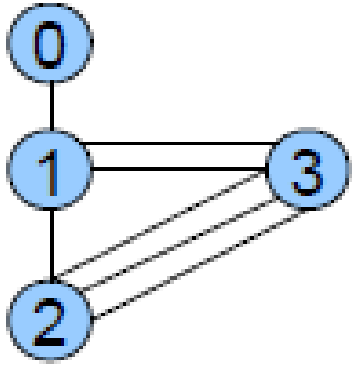
$V(G_1) = \{0, 1, 2, 3\}$
$E(G_1) = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$
$E(G_2) = \{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$V(G_3) = \{0, 1, 2\}$
$E(G_3) = \{<0,1>,<1,0>,<1,2>\}$

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.



(a) Graph with a self edge



(b) Multigraph

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 $(v, w) \in E, (v, w) = (w, v)$

TERMINOLOGY OF GRAPH (CONT'D)

- Path: a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ where $(w_i, w_{i+1}) \in E, \forall 1 \leq i \leq N$.
- 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.

TERMINOLOGY OF GRAPH (CONT'D)

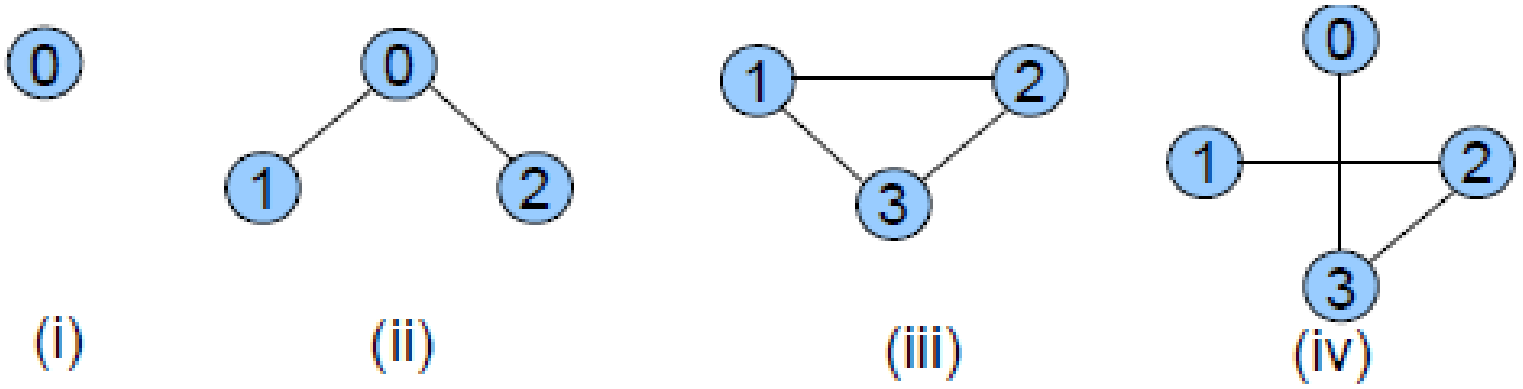
- Connected: an undirected graph if there is a path from every vertex to every vertex.
- Strongly connected: a directed graph if there is a path from every vertex to every vertex.
- Complete graph: a graph in which there is an edge between every pair of vertices.

COMPLETE GRAPH

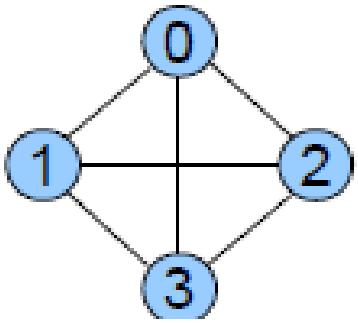
- The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $n(n-1)/2$.
- A complete unordered graph is an unordered graph with exactly $n(n-1)/2$ edges.
- A complete directed graph is a directed graph with exactly $n(n-1)$ edges.

SUBGRAPH

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

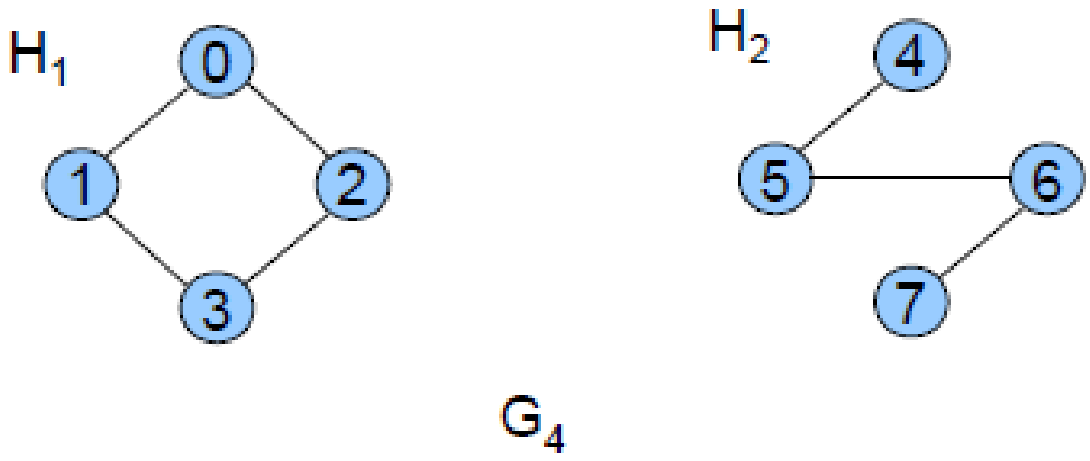


(a) Some subgraphs of G_1



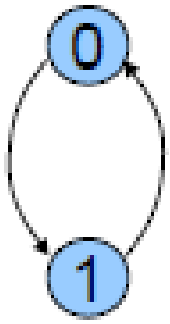
GRAPHS WITH TWO CONNECTED COMPONENTS

- 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.



Strongly Connected Components of G_3

DEGREE OF A VERTEX

- 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 *n* vertices and e edges, if d_i is the degree of a vertex *i* in G, then the number of edges of G is

$$e = (\sum_{i=0}^{n-1} d_i)/2$$

ABSTRACT DATA TYPE OF GRAPHS

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

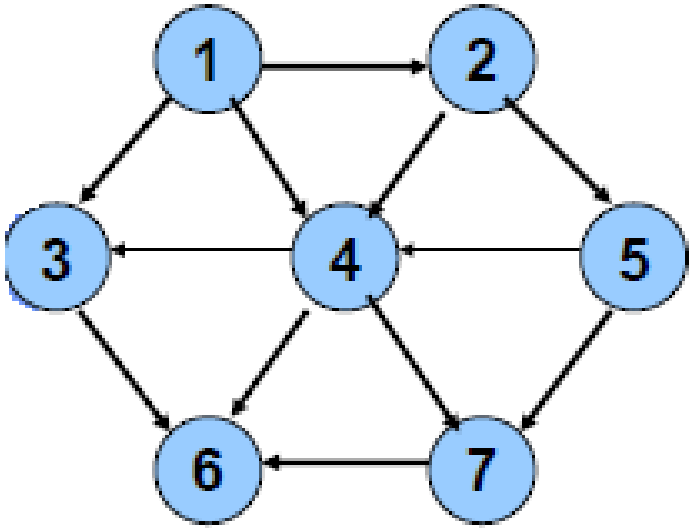
Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

ADJACENCY MATRIX REPRESENTATION

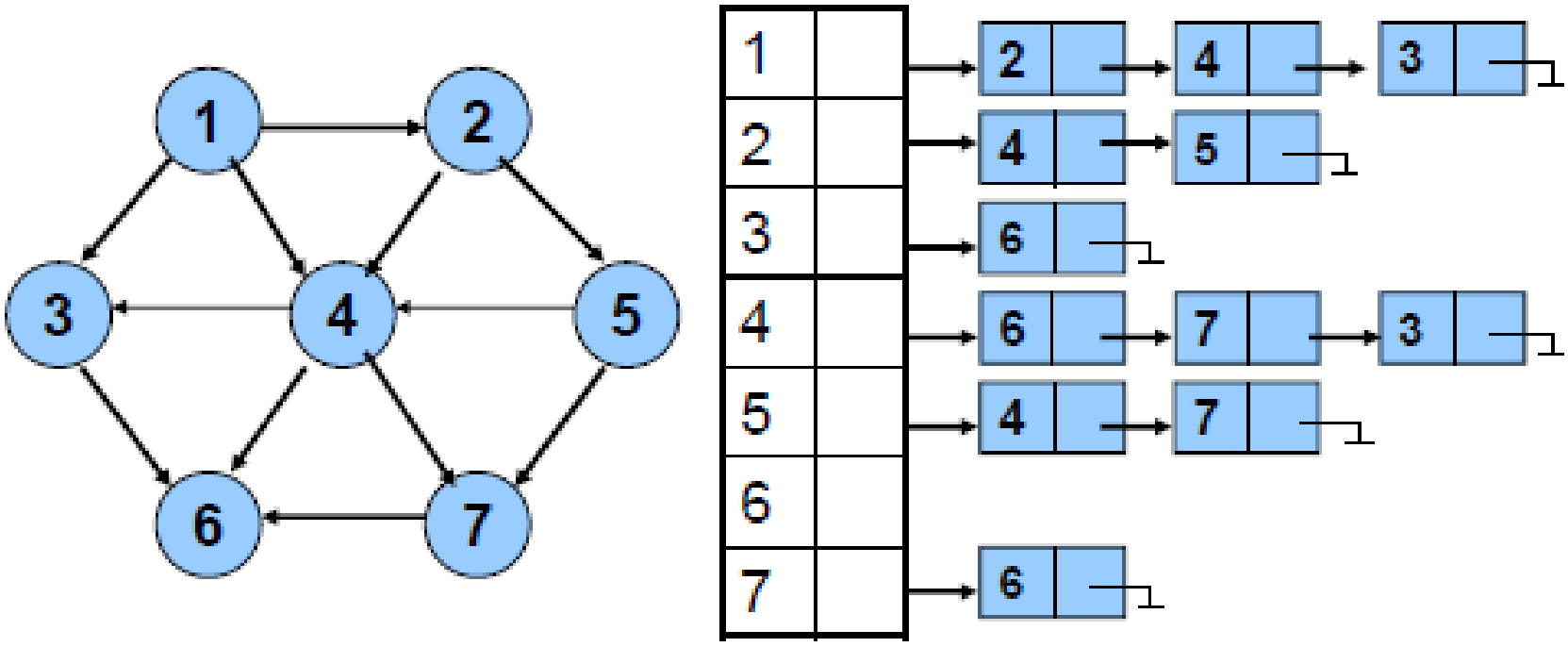


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

Space: $\Theta(|V|^2)$, good for dense, not for sparse

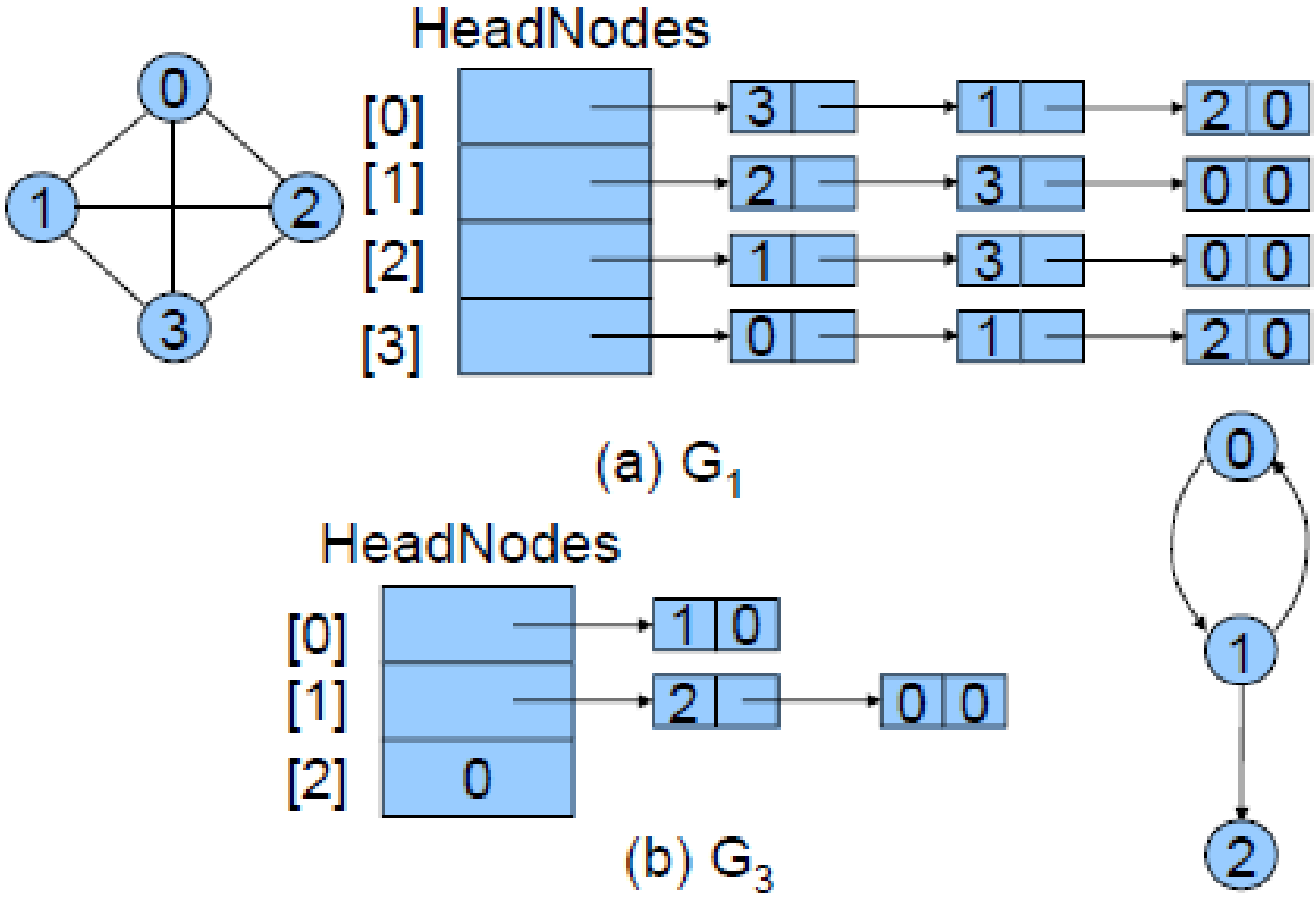
* Undirected graph: symmetric matrix

ADJACENCY LIST REPRESENTATION

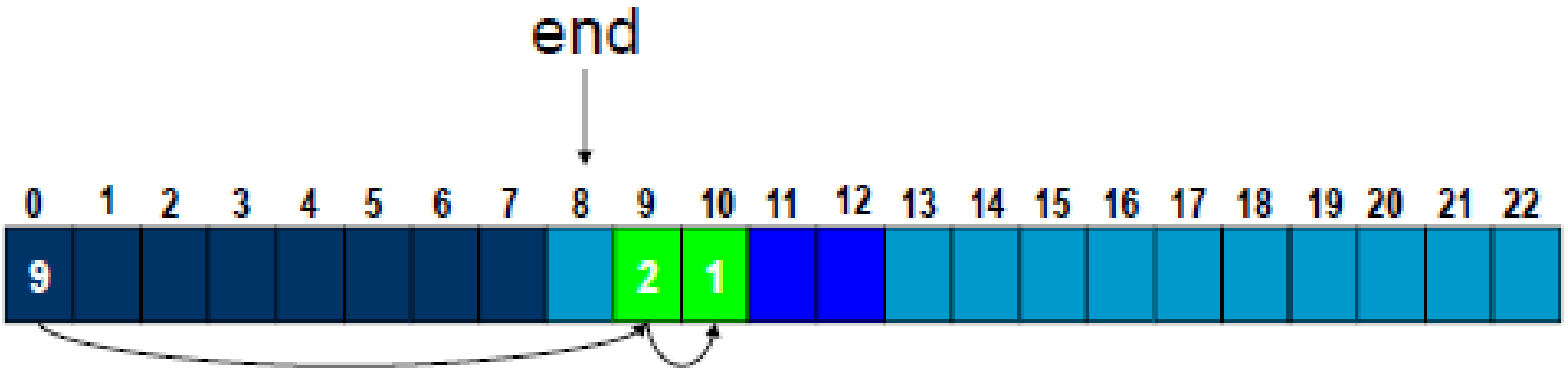


Space: O(|V|+|E|) good for sparse

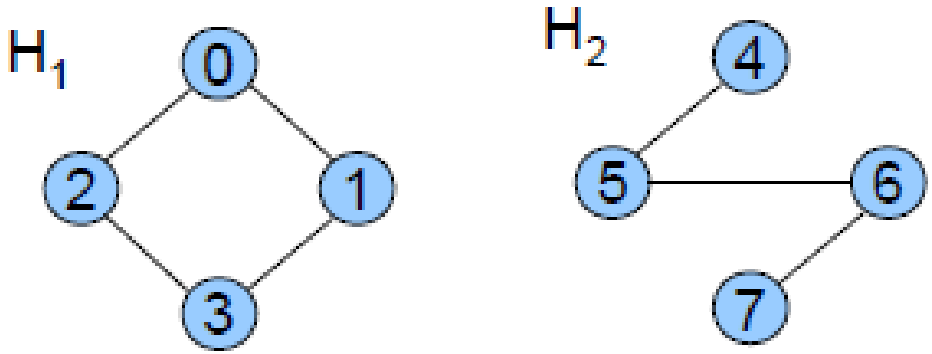
ADJACENT LISTS



SEQUENTIAL REPRESENTATION OF GRAPH G_4

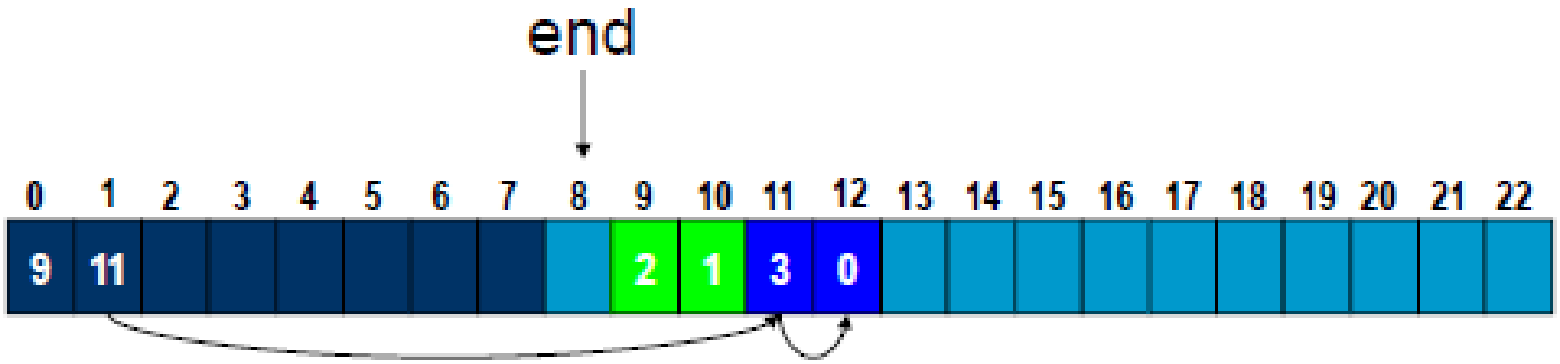


Edges adjacent to vertex 0: (0, 1), (0, 2)

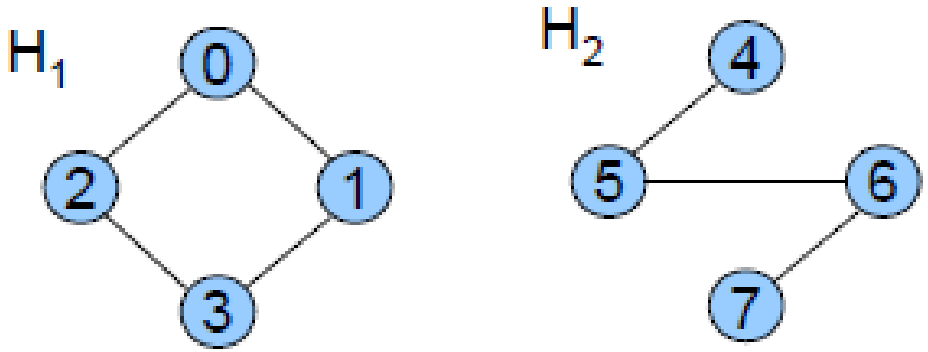


G_4

SEQUENTIAL REPRESENTATION OF GRAPH G_4 (CONT'D)

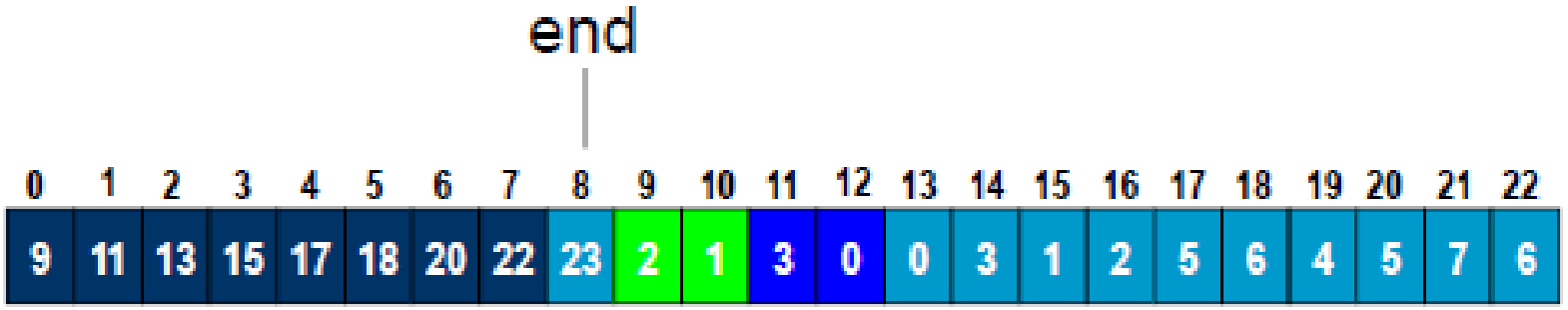


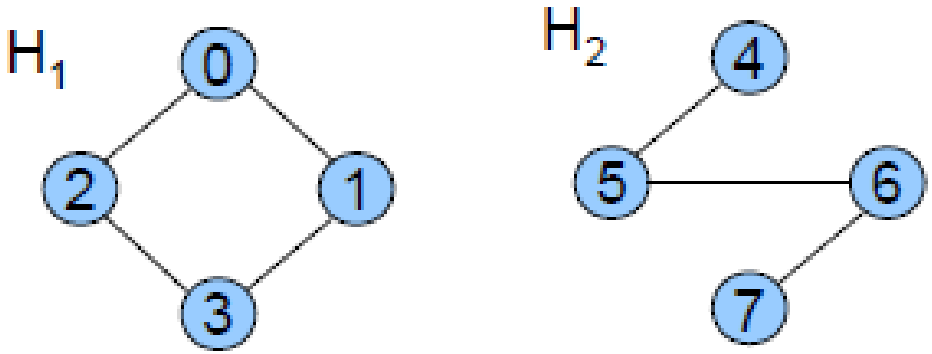
Edges adjacent to vertex 1: (1, 0), (1, 3)



G_4

SEQUENTIAL REPRESENTATION OF GRAPH G_4 (CONT'D)

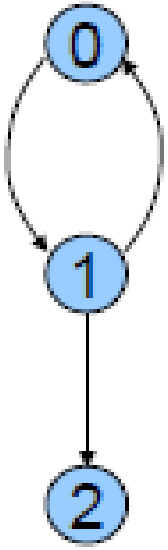


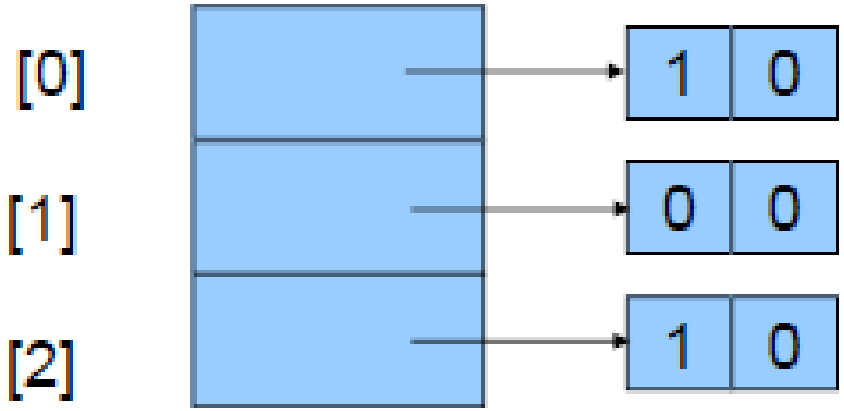


G_4

INVERSE ADJACENCY LISTS FOR G_3

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

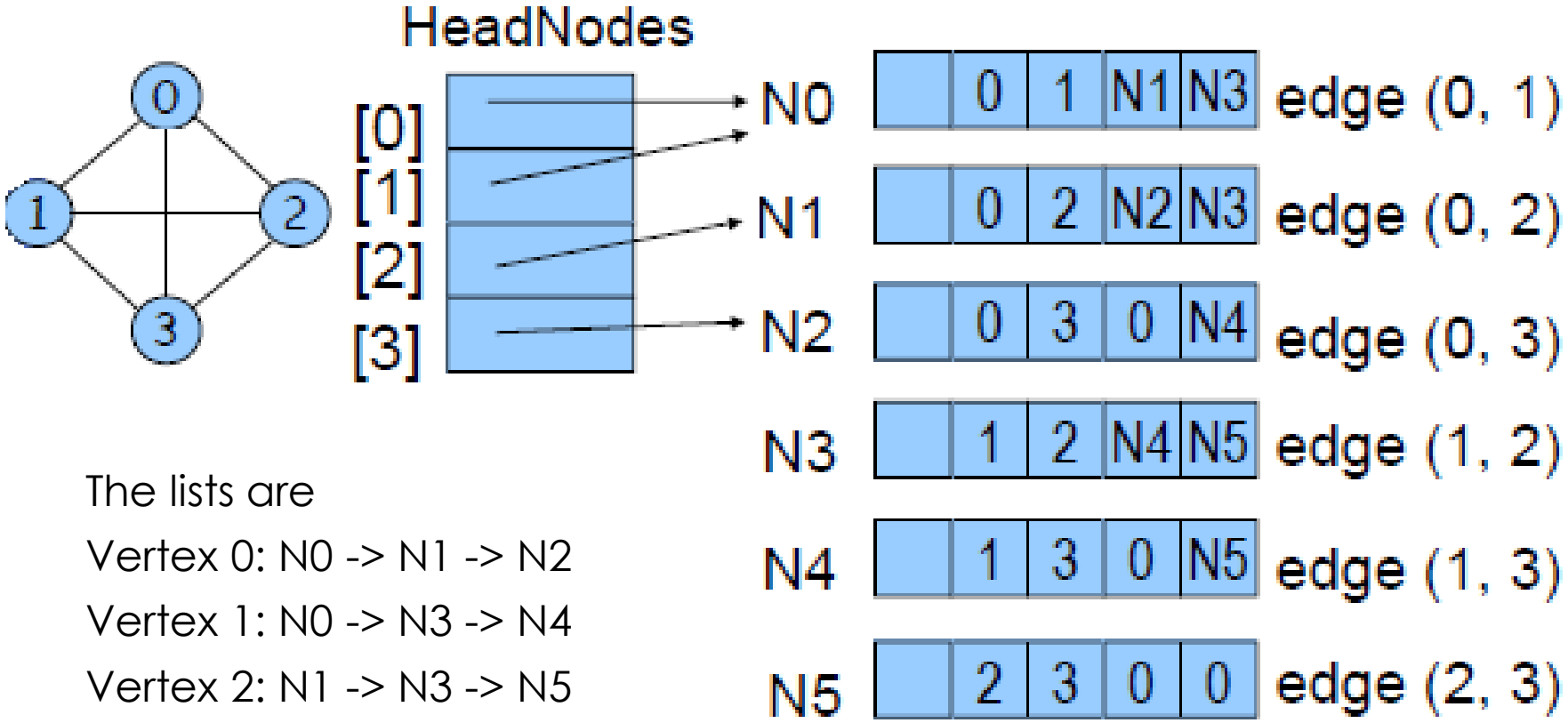




MULTILISTS

- 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.

ADJACENCY MULTILISTS FOR G_1



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*.

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.

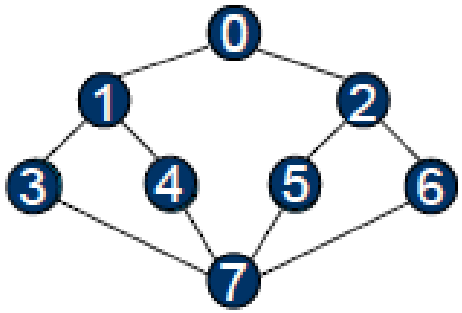
DEPTH-FIRST SEARCH

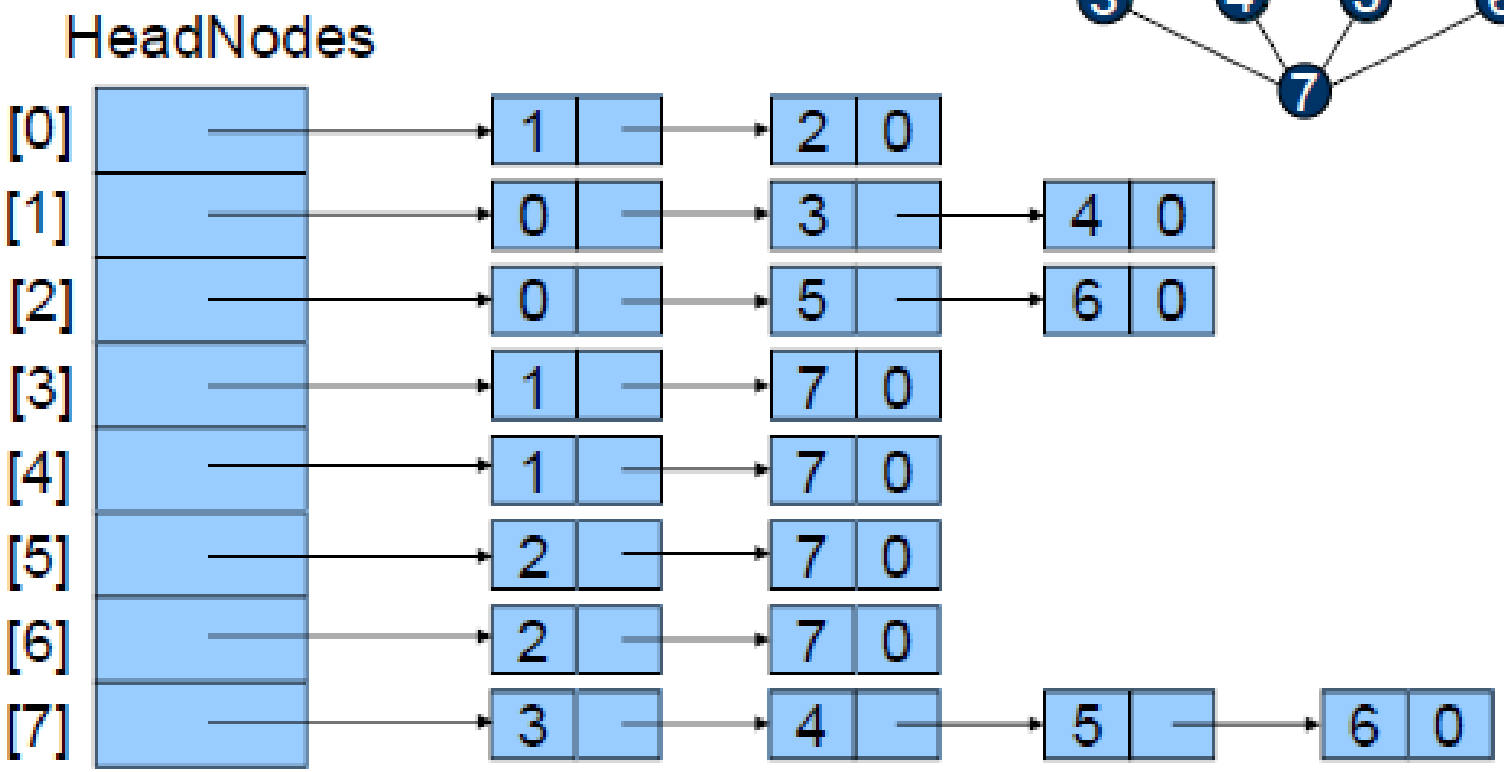
- Depth First Search (DFS): generalization of preorder traversal
- Starting from vertex *v*, process *v* & then recursively traverse all vertices adjacent to *v*.
  - Using stack
- To avoid cycles, mark visited vertices

GRAPH G AND ITS ADJACENCY LISTS

DFS from 0:

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





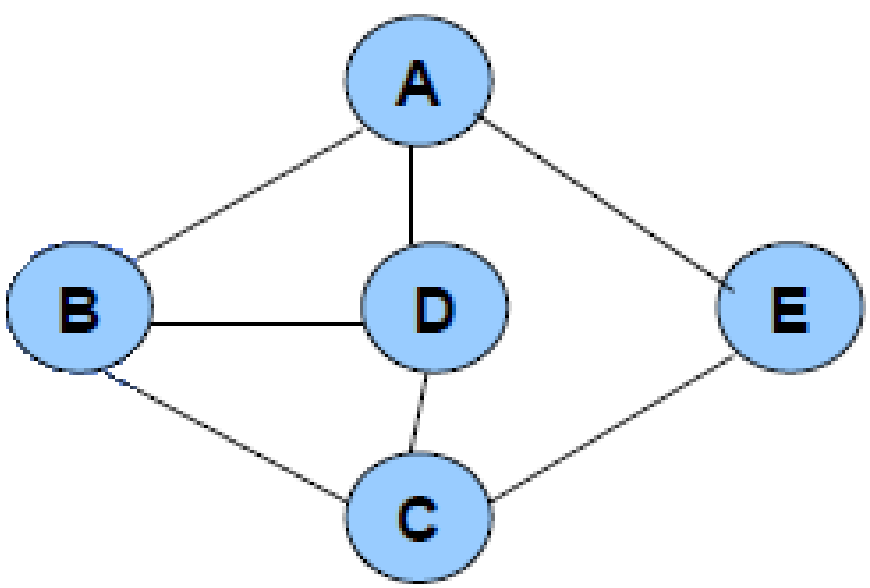
ANALYSIS OF DFS

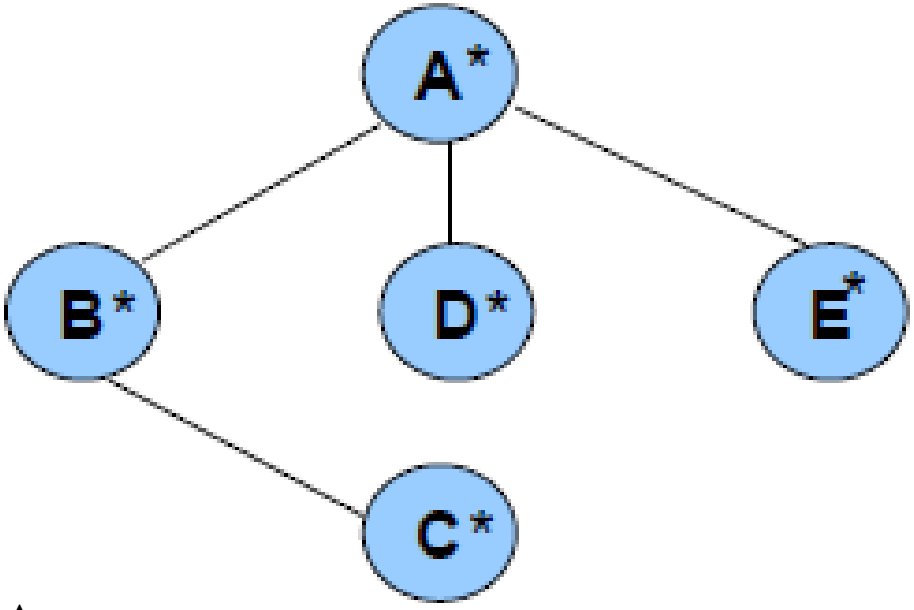
- If G is represented by its adjacency lists, the DFS time complexity is O(e).
  - There are 2e list nodes in the adjacency lists
- If G is represented by its adjacency matrix, then the time complexity to complete DFS is $O(n^2)$.

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

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
- Program 6.3 (pp.348)

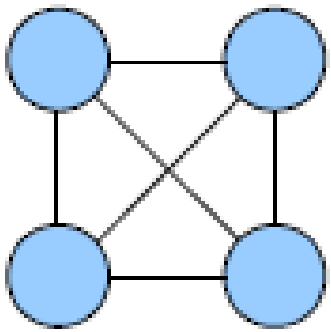
CONNECTED COMPONENTS (CONT'D)

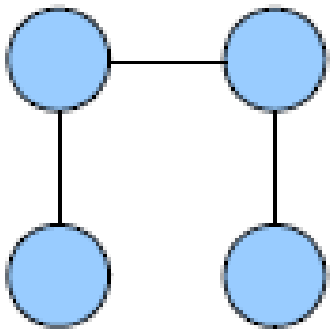
- If G is represented by adjacency lists, the time complexity is O(n+e)
  - O(e) for DFS
  - O(n) for for loops
- If G is represented by adjacency graphs, the time complexity is $O(n^2)$

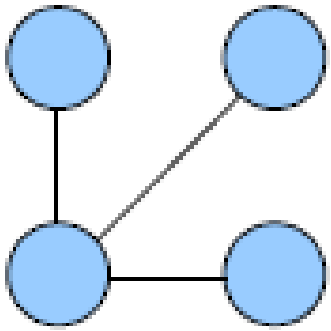
SPANNING TREE

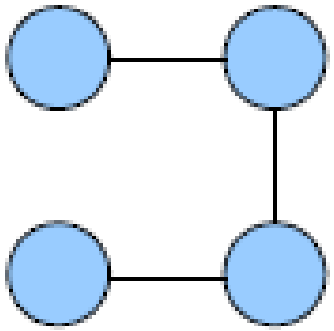
- Any tree consisting solely of edges in G and including all vertices in G is called a *spanning tree*.
  - Can be obtained by using either a depth-first or a breadth-first search.
- A spanning tree is a minimal subgraph, G', of G such that V(G') = V(G), and G' is connected. (Minimal subgraph is defined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

A COMPLETE GRAPH AND ITS SPANNING TREES

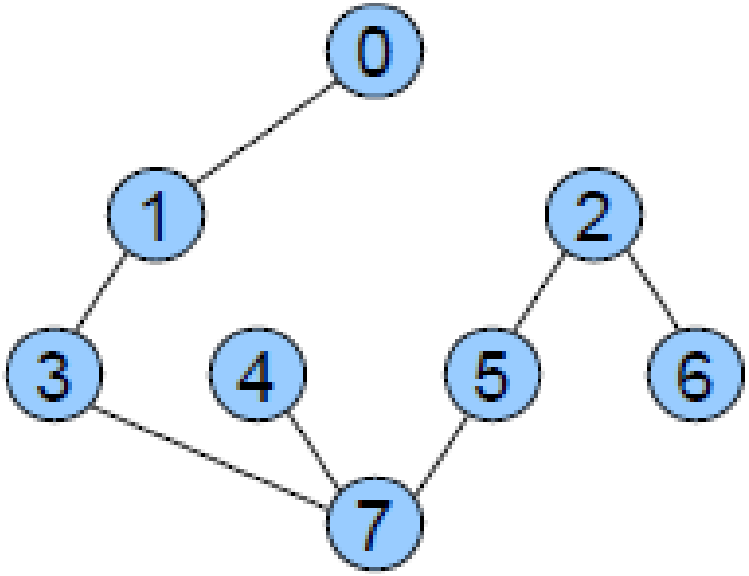




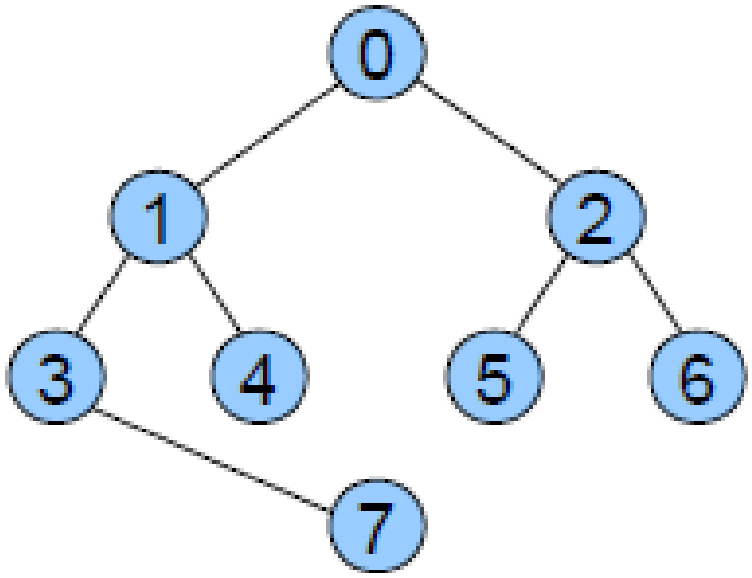




DEPTH-FIRST AND BREADTH-FIRST SPANNING TREES



(a) DFS (0) spanning tree



(b) BFS (0) spanning tree

MINIMAL COST SPANNING TREE

- A *minimum-cost spanning tree* is a spanning tree of least cost
  - Cost: the sum of the costs (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

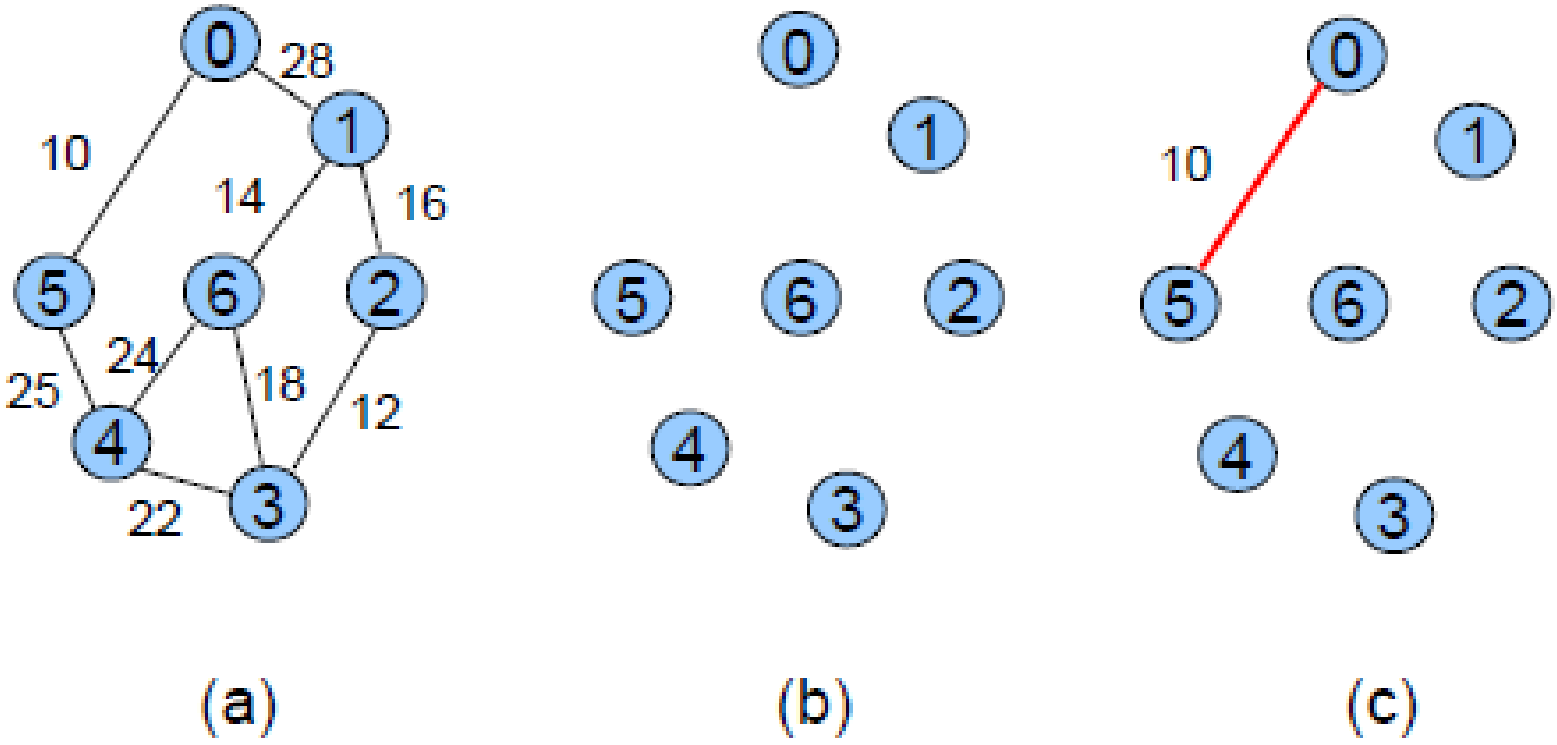
MINIMAL COST SPANNING TREE (CONT'D)

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

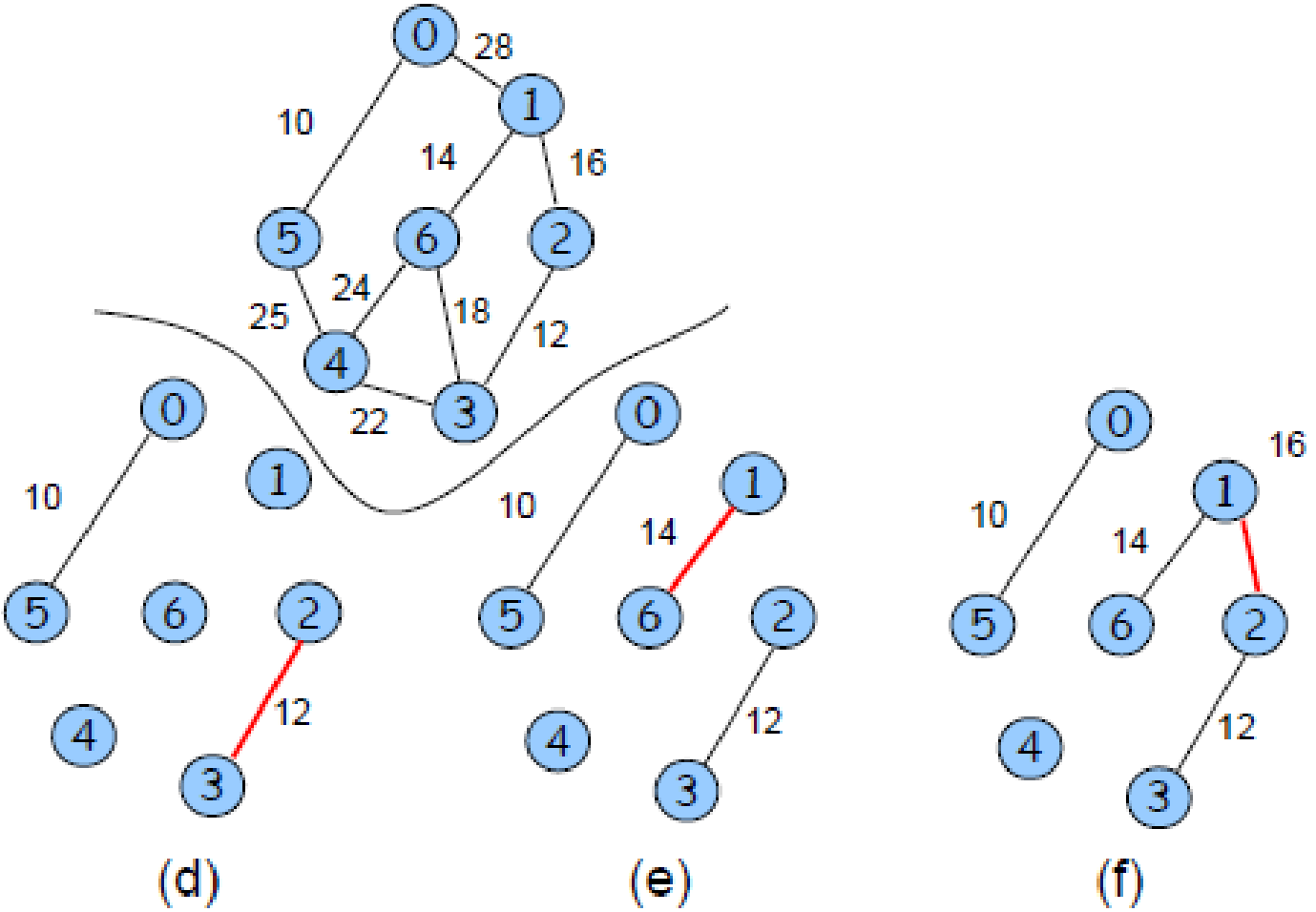
KRUSKAL'S ALGORITHM

- 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
- Time complexity: $O(eloge)$

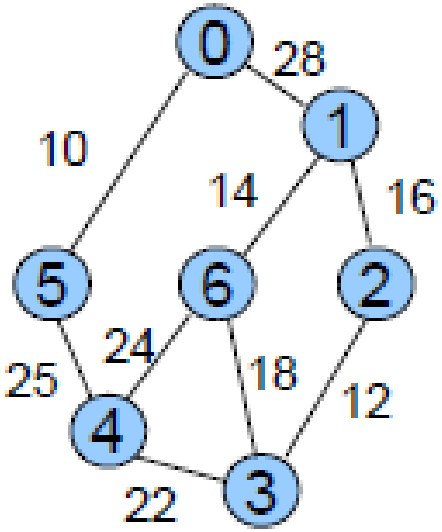
STAGES IN KRUSKAL'S ALGORITHM

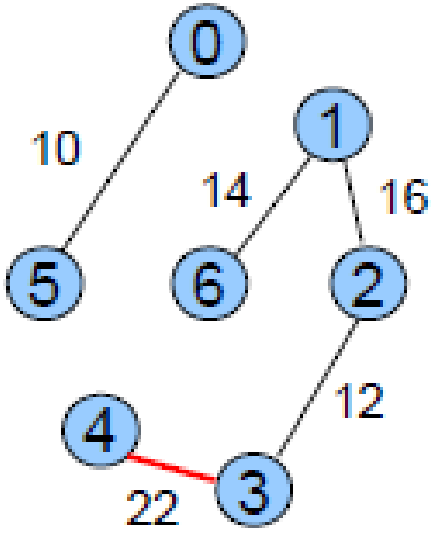


STAGES IN KRUSKAL'S ALGORITHM (CONT'D)

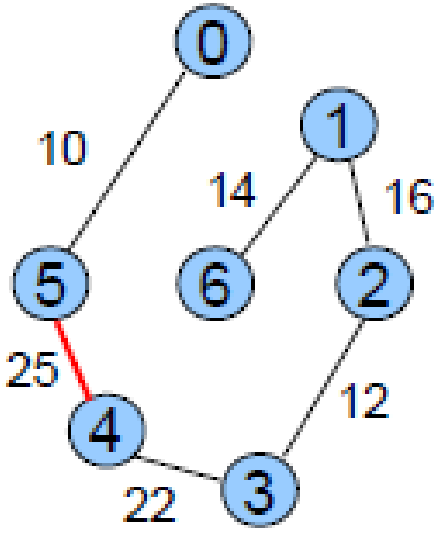


STAGES IN KRUSKAL'S ALGORITHM (CONT'D)





(g)

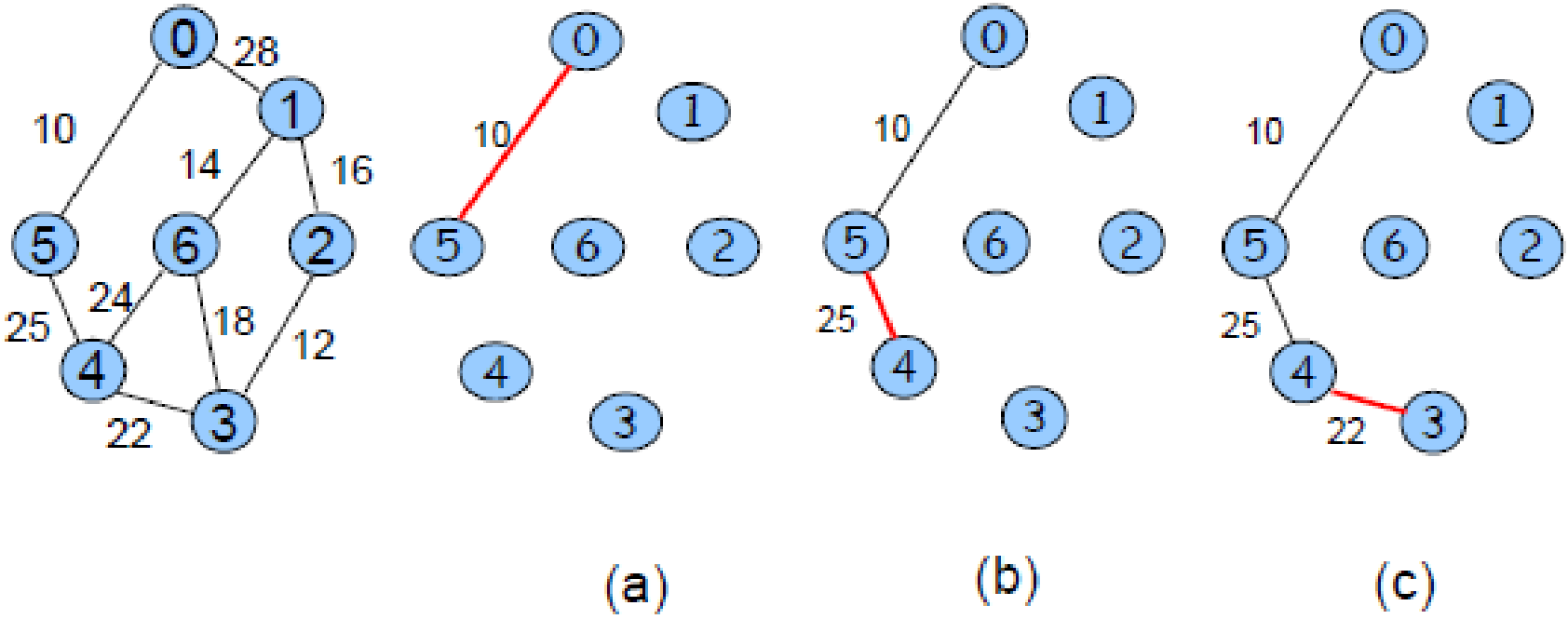


(h)

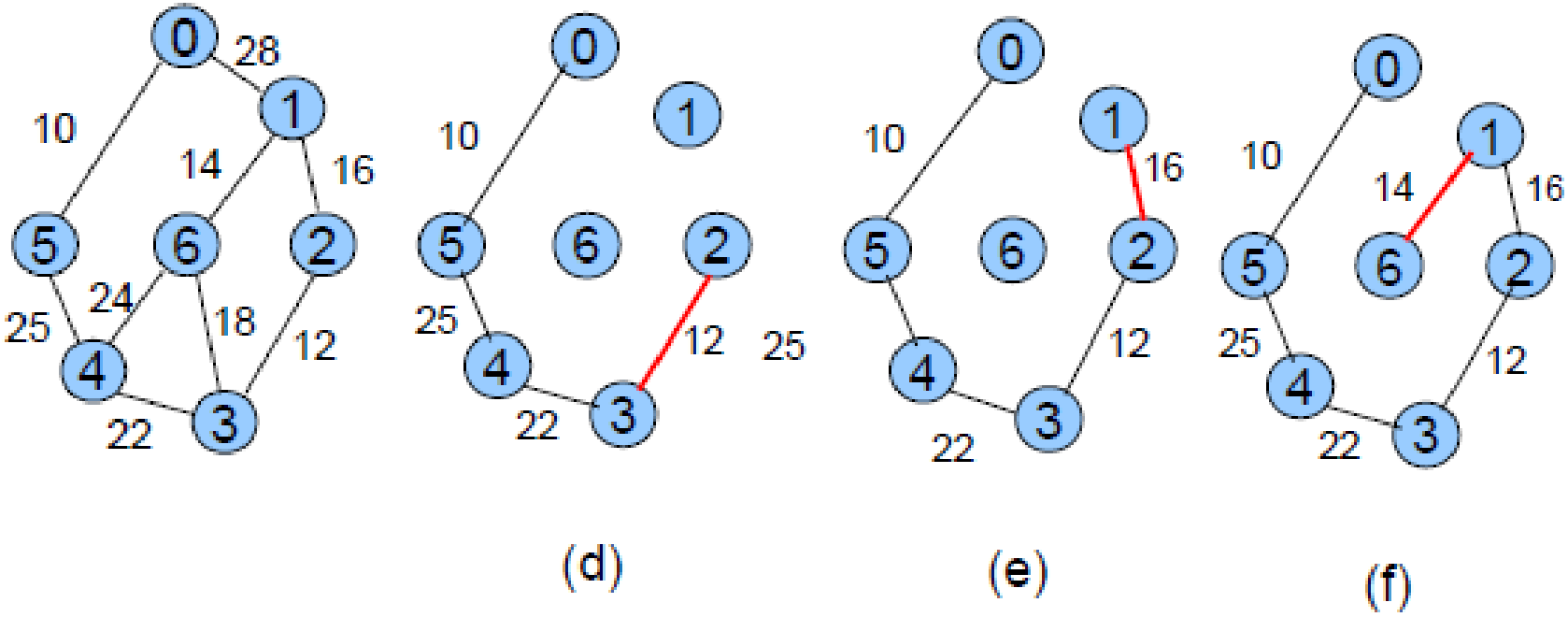
PRIM'S ALGORITHM

- 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 ∪ $\{(u, v)\}$ 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



STAGES IN PRIM'S ALGORITHM (CONT'D)



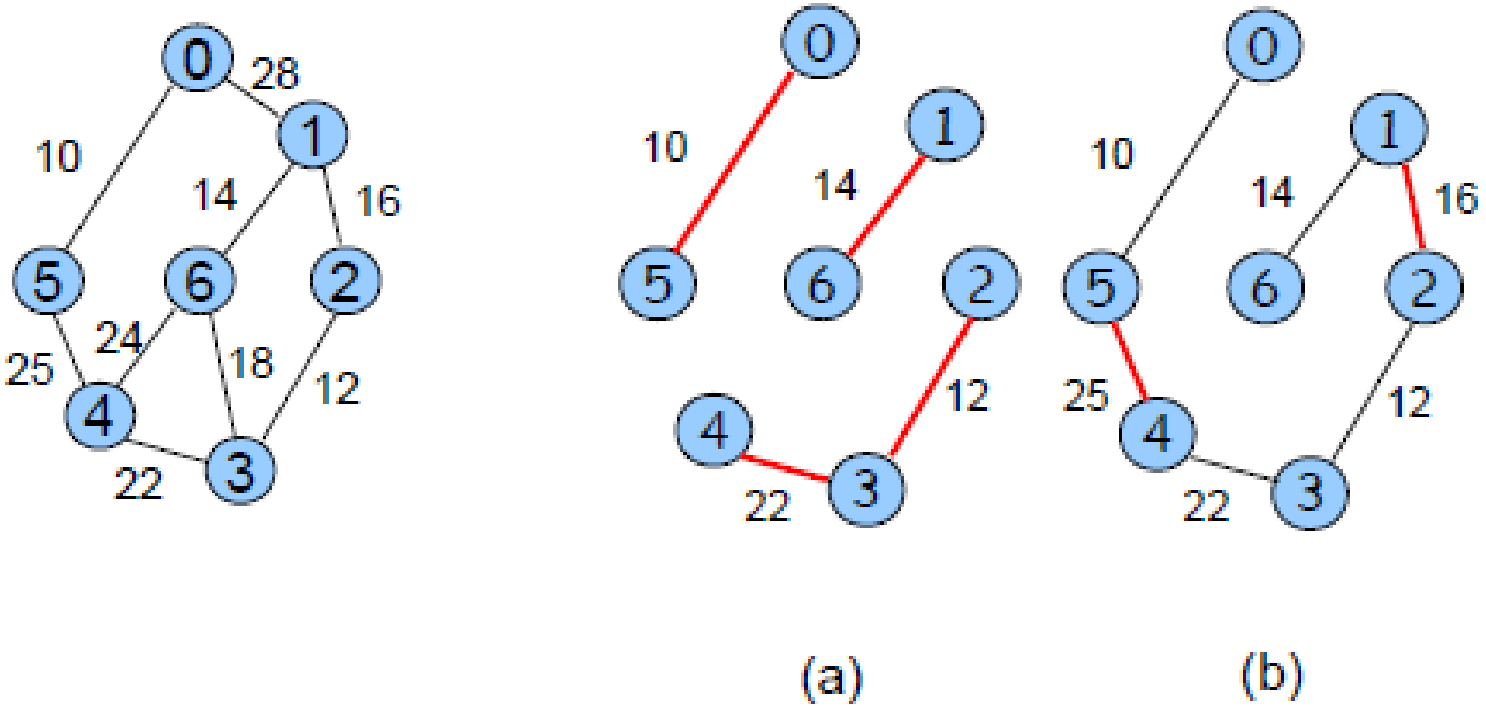
SOLLIN'S ALGORITHM

- 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, a 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)

SOLLIN'S ALGORITHM (CONT'D)

- It's possible that two trees in the forest to select the same edge. Only one should be used.
- Also, 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.

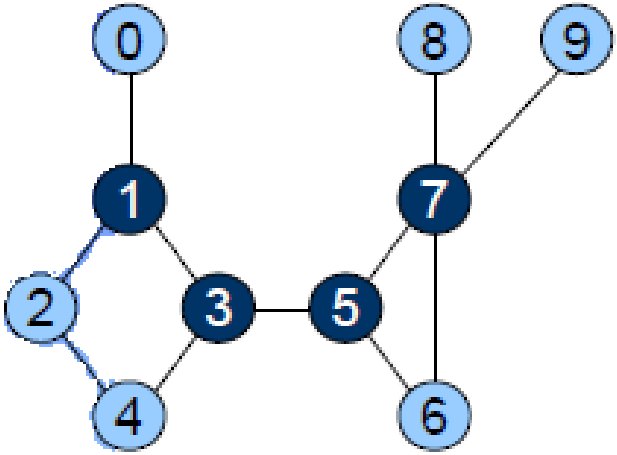
STAGES IN SOLLIN'S ALGORITHM



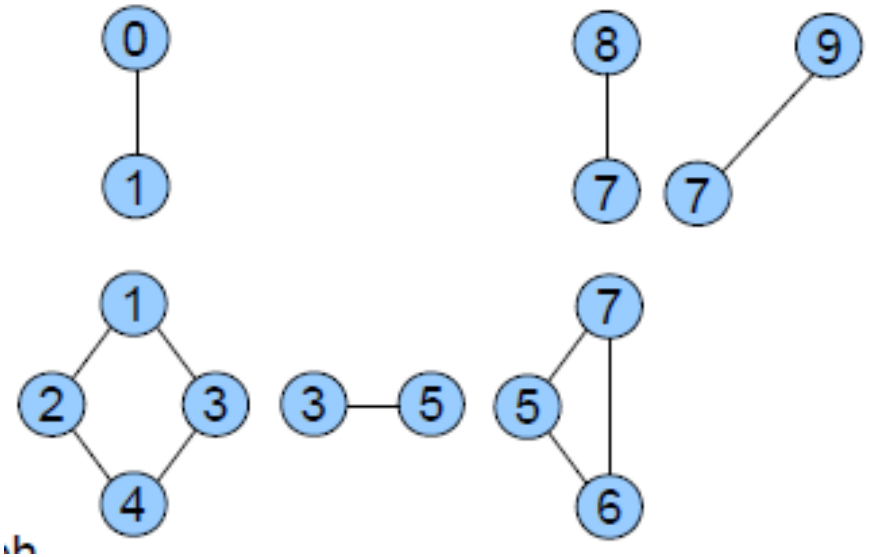
BICONNECTED COMPONENTS

- 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 two connected components
- A biconnected graph is a connected graph that has no articulation points
  - e.g., Figure 6.16a
- A biconnected component of a connected graph G is a maximal biconnected subgraph H of G
  - G contains no other subgraph that is both biconnected and properly contains H.

A CONNECTED GRAPH AND ITS BICONNECTED COMPONENTS



A connected graph



Its biconnected components

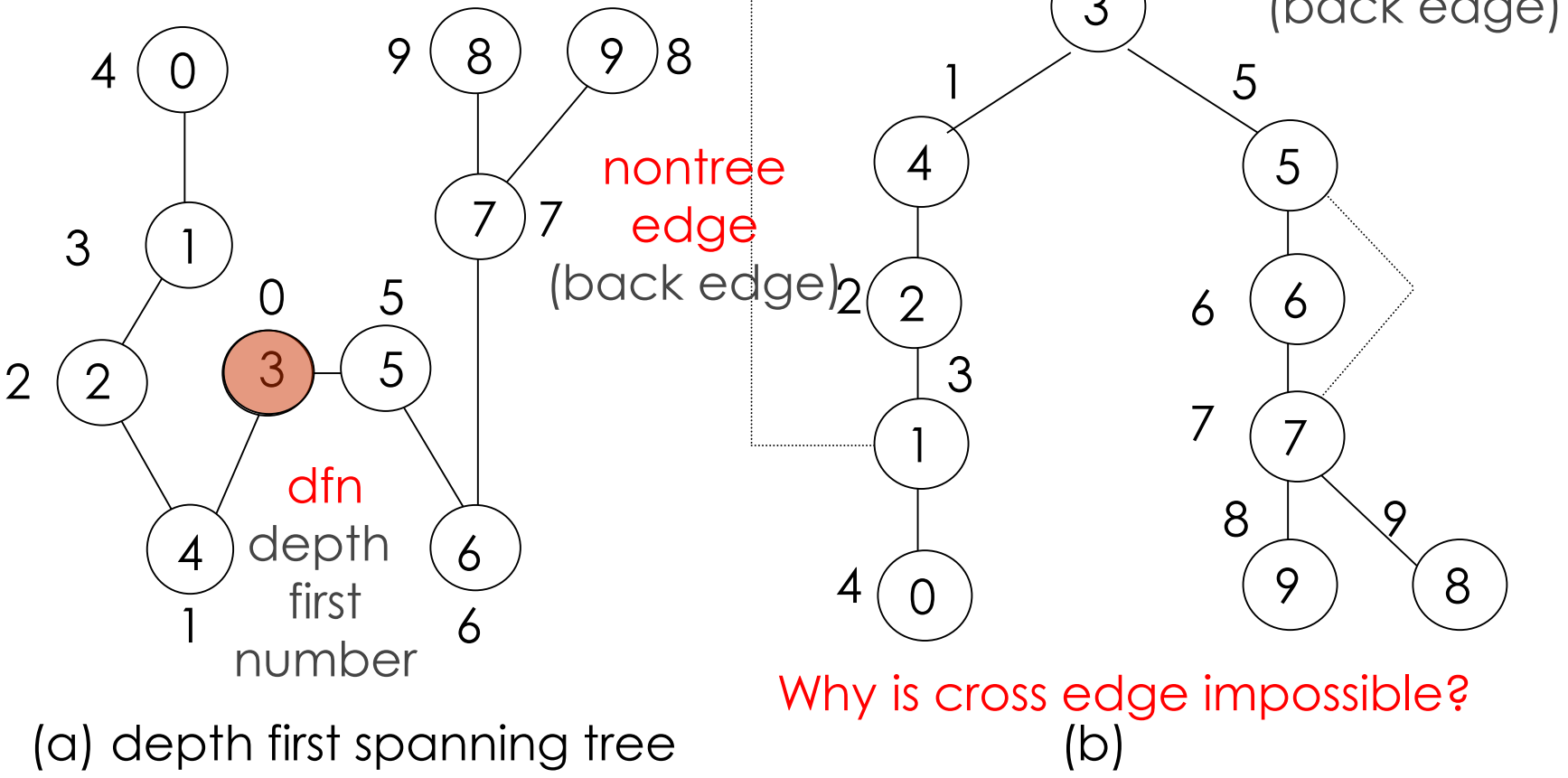
BICONNECTED COMPONENTS (CONT'D)

- 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
  - found by using any depth-first spanning tree of G

BICONNECTED COMPONENTS (CONT'D)

- 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
- In a DFS of an undirected graph G, every edge of G is either tree edge or a back edge
  - Forward edges and cross edges only appear in directed graphs

DEPTH-FIRST SPANNING TREE



If u is an ancestor of v then dfn(u) < dfn(v).

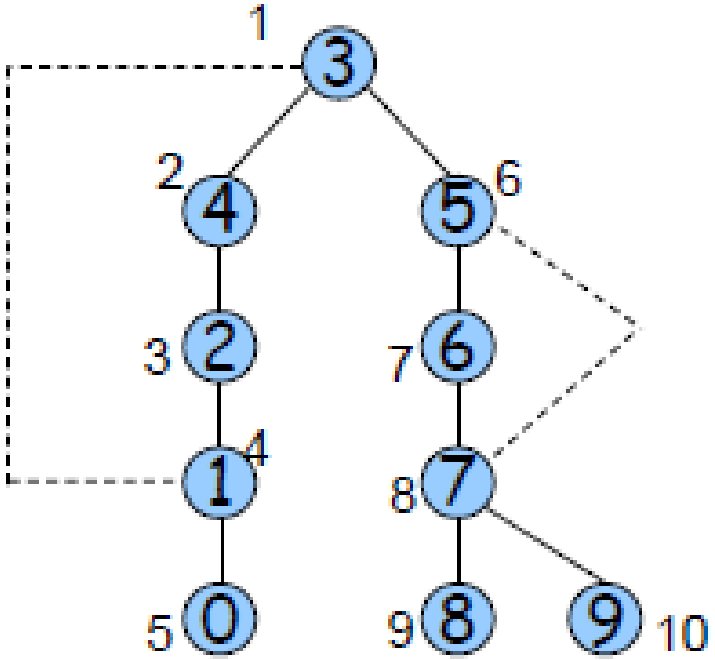
BICONNECTED COMPONENTS (CONT'D)

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

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

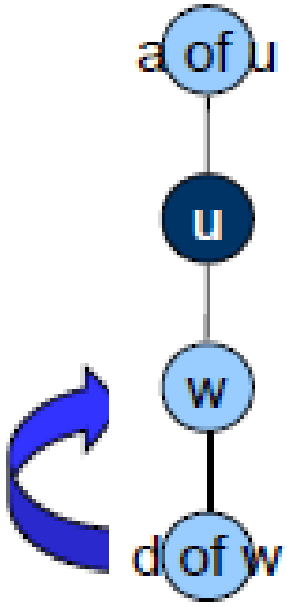
DFN AND LOW VALUES FOR THE SPANNING TREE

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

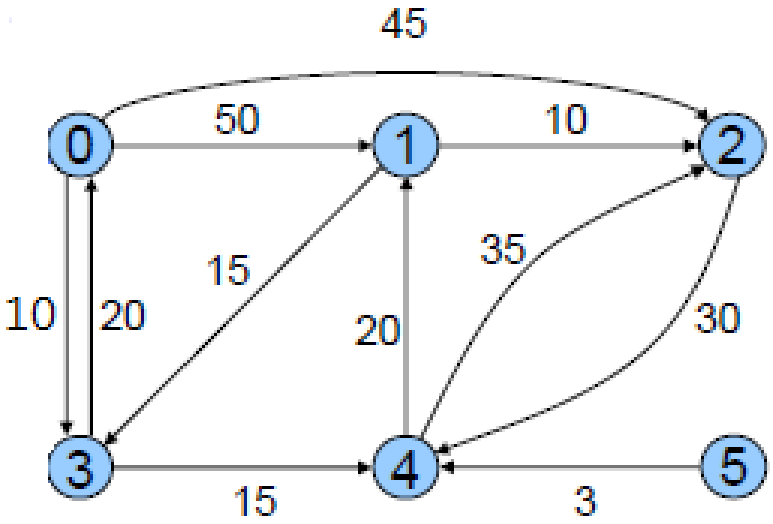


BICONNECTED COMPONENTS (CONT'D)

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



GRAPH AND SHORTEST PATHS FROM VERTEX 0 TO ALL DESTINATIONS



(a) Graph

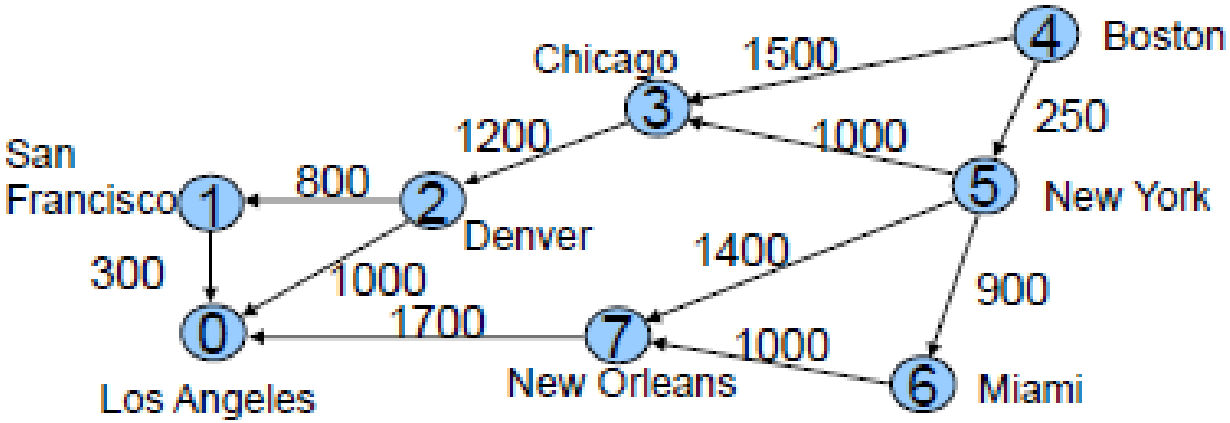
Path	Length
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

(b) Shortest paths from 0

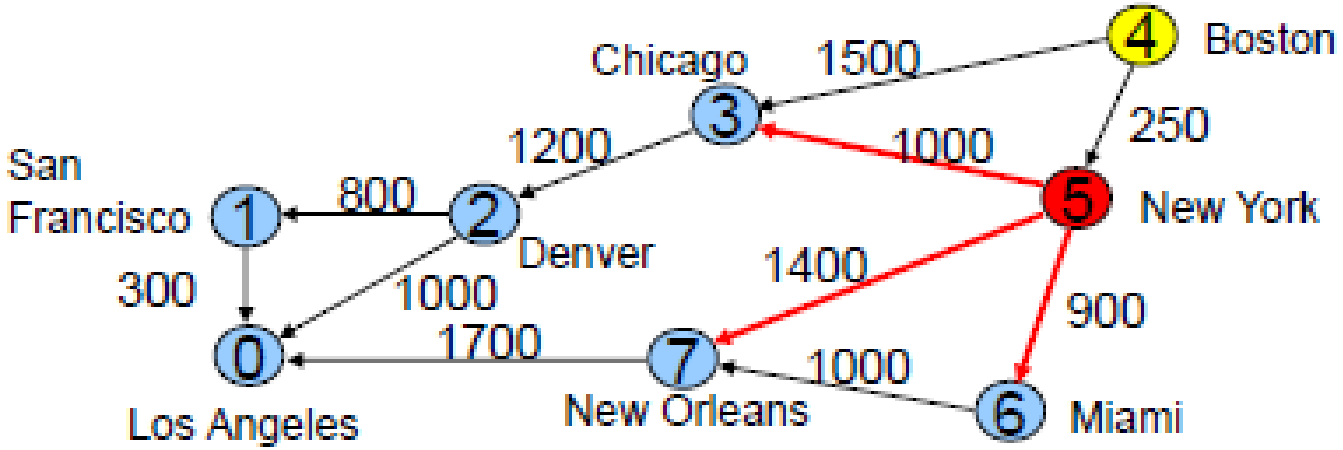
SINGLE SOURCE/ALL DESTINATIONS: NONNEGATIVE EDGE COSTS

- Let S denote the set of vertices to which the shortest paths have already been found.
  - 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.**
  - The destination of the next path generated must be the vertex u that has the minimum distance **among all vertices not in S.**
  - The vertex u selected in 2) becomes a member of S.

DIAGRAM FOR EXAMPLE 6.5



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



$$\begin{aligned}\text{Distance}[3] &= \min\{\text{Distance}[3], \text{Distance}[5]+\text{length}[5][3]\}\\ &= \min\{1500, 250+1000\}\\ &= 1250\end{aligned}$$

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**				

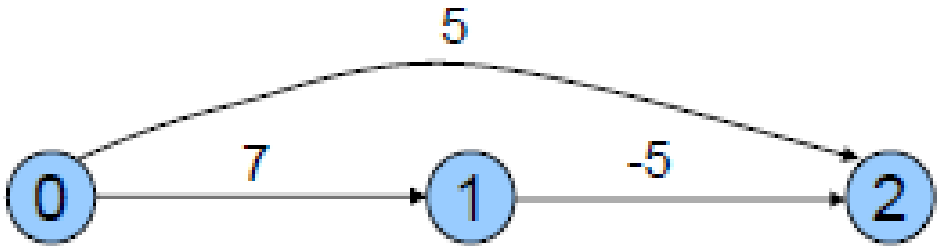
ACTION OF SHORTEST PATH

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

SINGLE SOURCE/ALL DESTINATIONS: NONNEGATIVE EDGE COSTS (CONT'D)

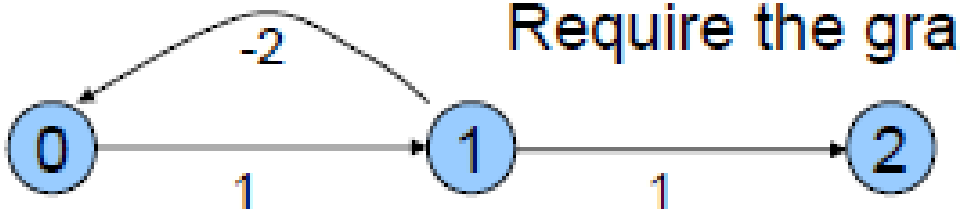
- The algorithm is first given by Edsger Dijkstra. Therefore, it's sometimes called Dijkstra Algorithm.
- Time complexity
  - Adjacency matrix, adjacency list: $O(n^2)$
  - Using Fibonacci heap: $O(nlogn + e)$

DIRECTED GRAPHS



(a) Directed graph with a negative-length edge

Require the graph has no cycles



(b) Directed graph with a cycle of negative length

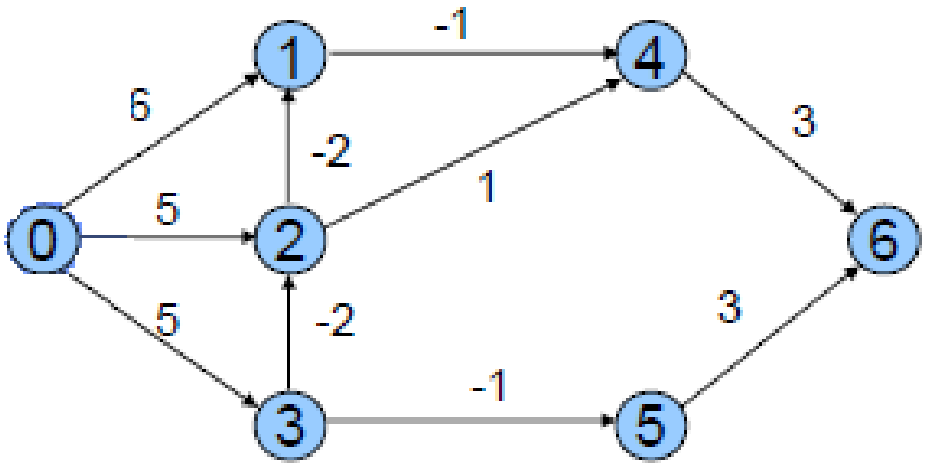
SINGLE SOURCE/ALL DESTINATIONS: GENERAL WEIGHTS

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

SINGLE SOURCE/ALL DESTINATIONS: GENERAL WEIGHTS (CONT'D)

- The distance can be computed in recurrence by the following:
  $dist^k[u] = \min\{dist^{k-1}[u], \min\{dist^{k-1}[i] + length[i][u]\}\}$
- The algorithm is also referred to as the Bellman-Ford Bellman Algorithm.
- Time complexity:
  - Adjacency matrix: $O(n^3)$
  - Adjacency list: $O(ne)$

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) $dist^k$

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

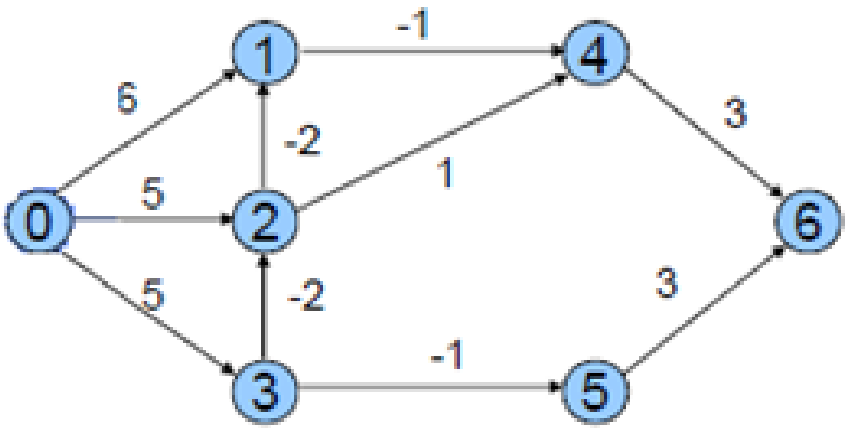
$\underline{dist^{k-1}[u]}$

$dist^{1}[2]=5$

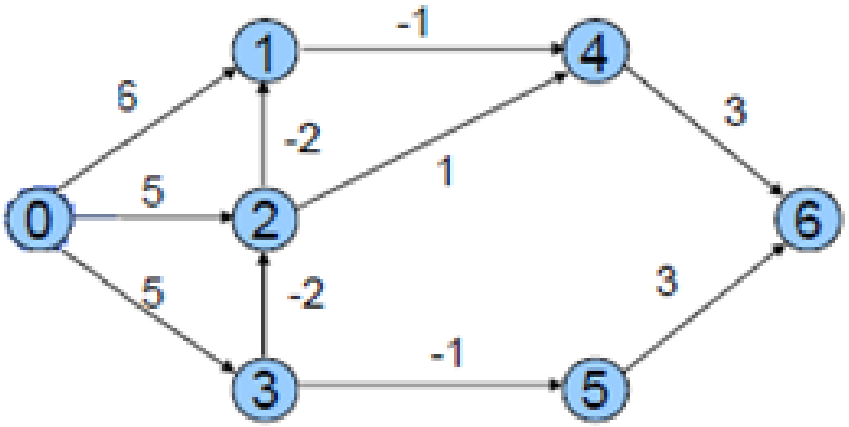
$\underline{dist^{k-1}[i]+length[i][u]}$

$dist^{1}[0]+length[0][2]=5$

$dist^{1}[3]+length[3][2]=5-2$



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



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

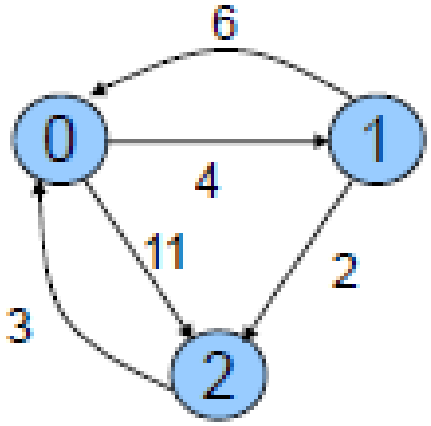
ALL-PAIRS SHORTEST PATHS

- 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 (CONT'D)

- How to determine the value of $A^k[i][j]$
  - $A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$
- Time complexity
  - $O(n^3)$

EXAMPLE FOR ALL-PAIRS SHORTEST-PATHS PROBLEM



$A^0[2][1]$

$A^{-1}[2][1] = \infty$

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

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

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(a) A^{-1}

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

(b) A^0

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

(c) A^1

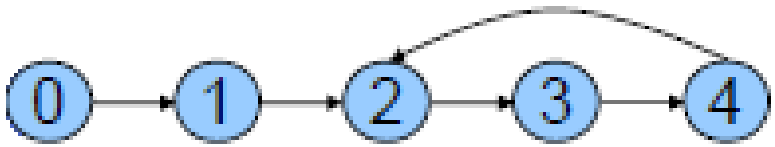
A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

(d) A^2

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 $length \geq 0$ from i to j; otherwise, $A^*[i][j] = 0$.

GRAPH G AND ITS ADJACENCY MATRIX A, A^+, A^*



(a) Digraph G

	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	0	0	0

(b) Adjacency matrix A

	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

(c) A^+

	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

(d) A^*

ACTIVITY-ON-VERTEX (AOV) NETWORKS

- Definition: Activity-On-Vertex network or 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.

AOV NETWORKS (CONT'D)

- 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 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 AOV NETWORK (CONT'D)

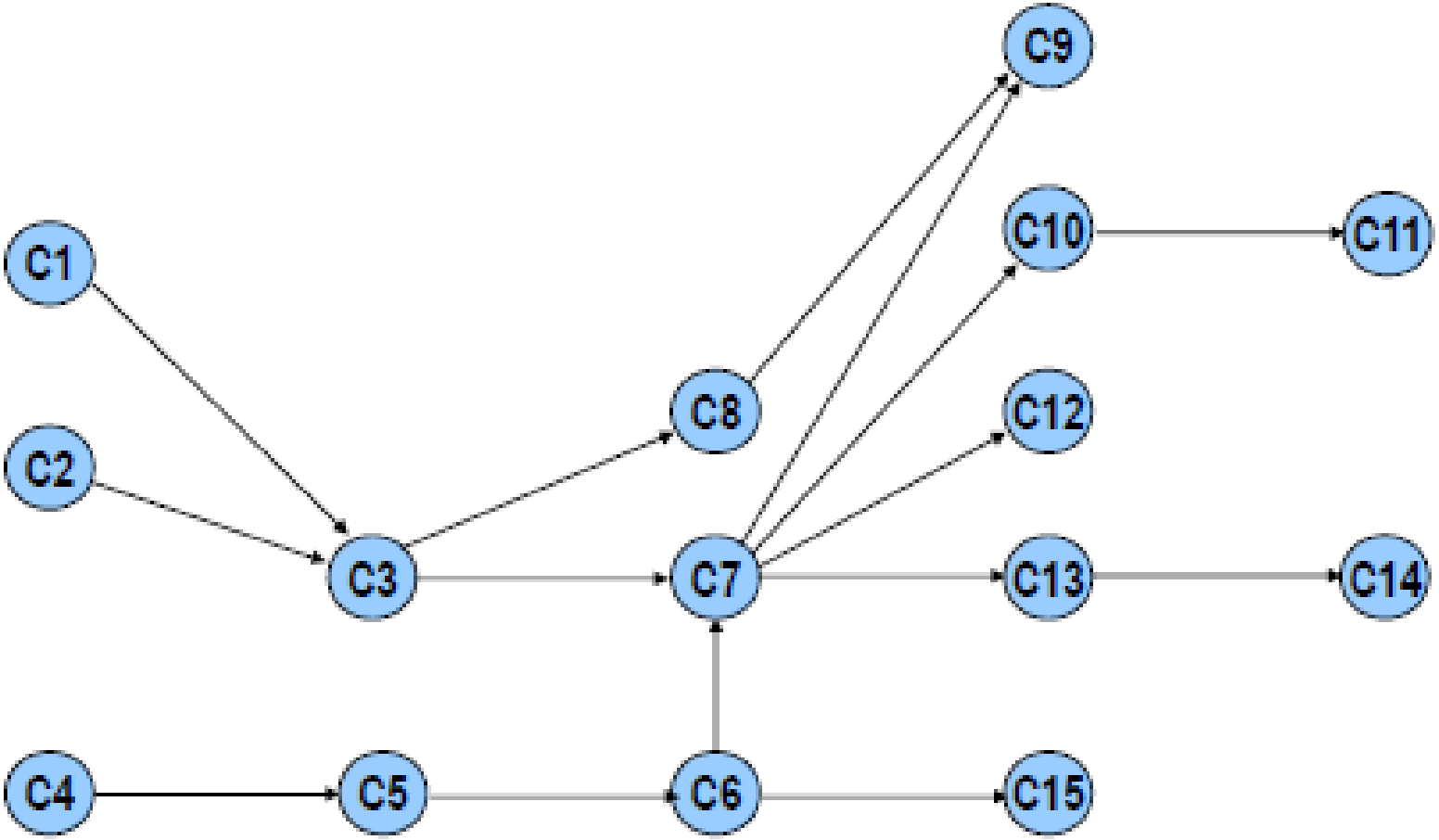
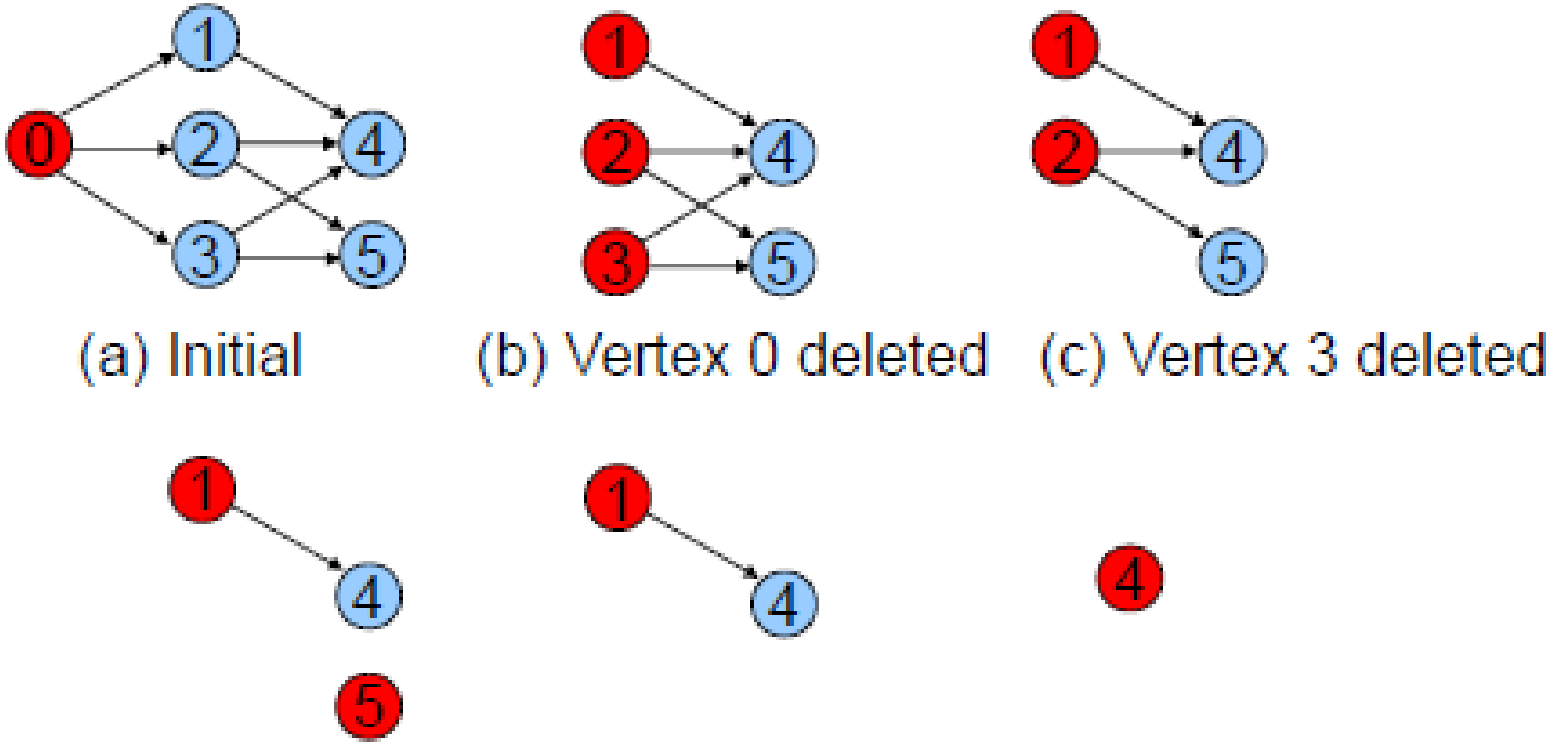
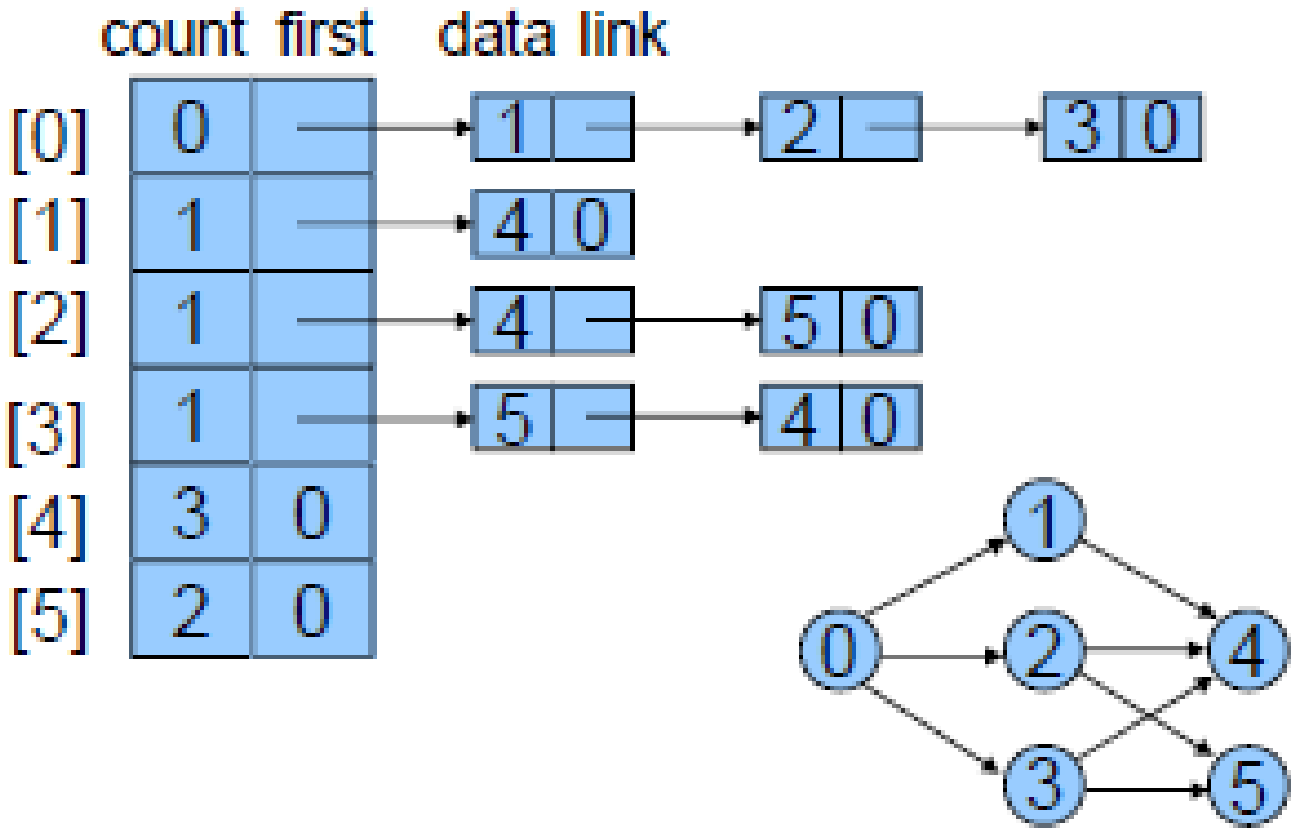


FIGURE 6.36 ACTION OF PROGRAM 6.11 ON AN AOV NETWORK

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



LIST REPRESENTATION OF AN AOV NETWORK



AN AOE NETWORK

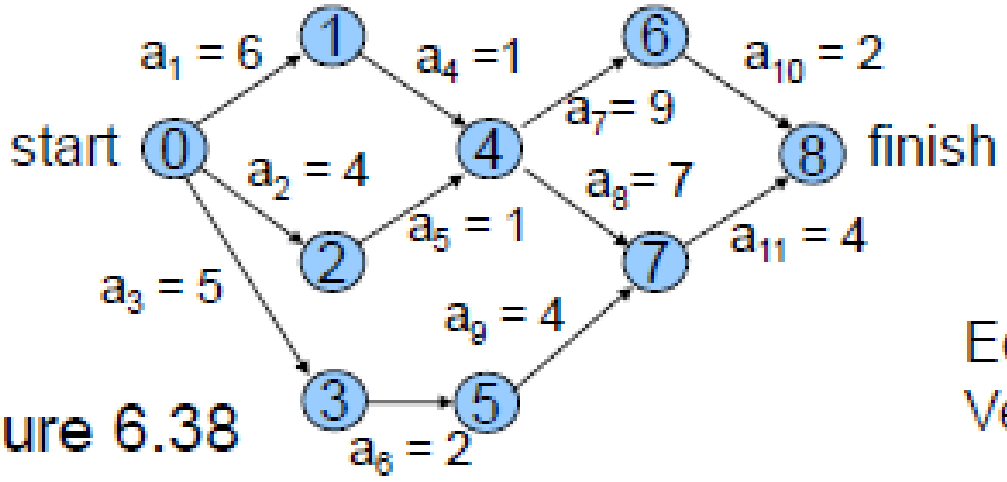


Figure 6.38

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 (CONT'D)

- 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 (CONT'D)

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

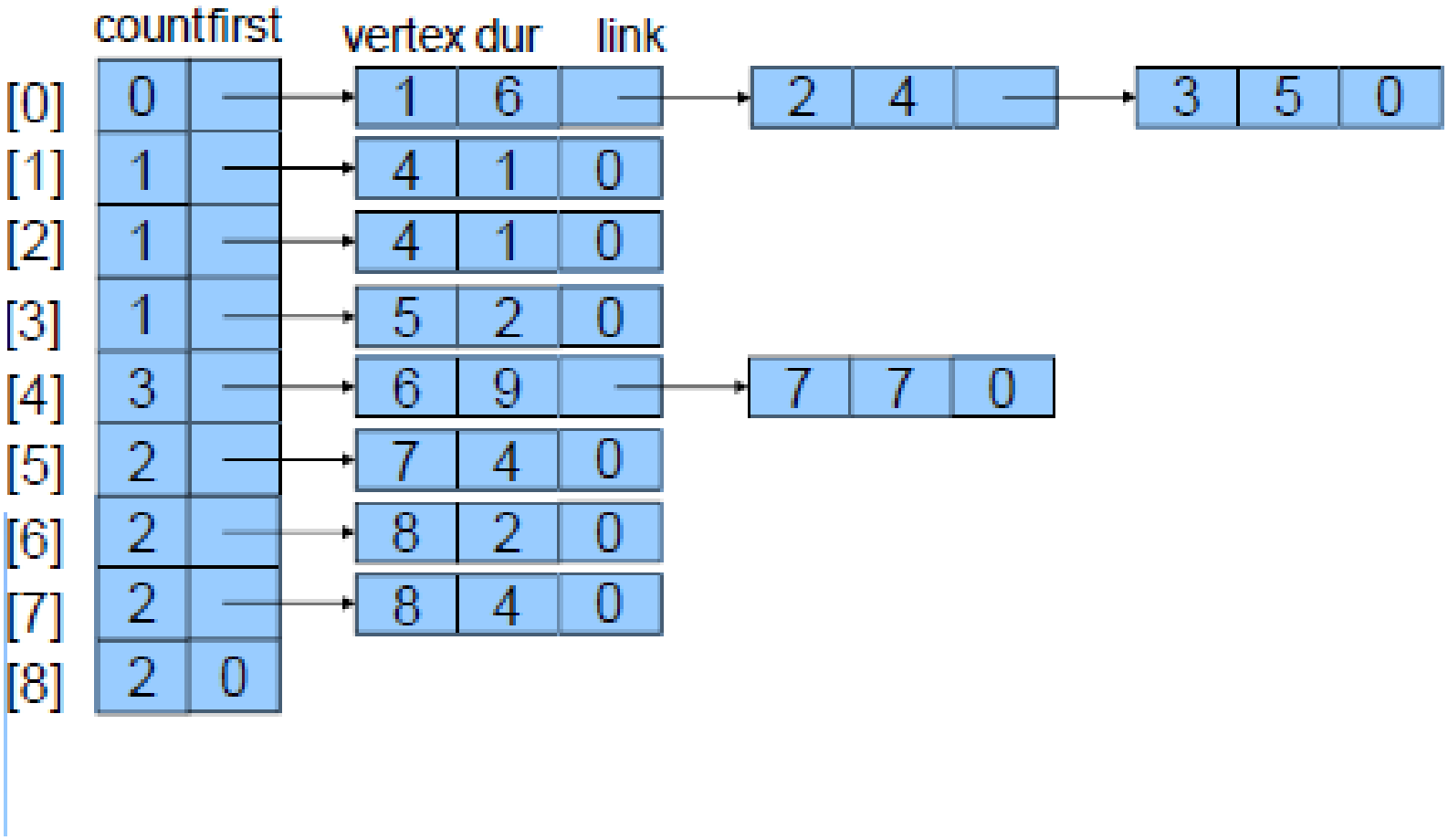
AN AOE NETWORK (CONT'D)

- 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

$$ee[j] = \max_{ie P(j)} \{ee[i] + duration\ of < i, j >\}, where$$
$$ee[0] = 0$$
$$le[j] = \min_{ie S(j)} \{le[i] - duration\ of < j, i >\}, where$$
$$le[n-1] = ee[n-1]$$

- Using topological order

ADJACENCY LISTS FOR FIGURE 6.38 (A)



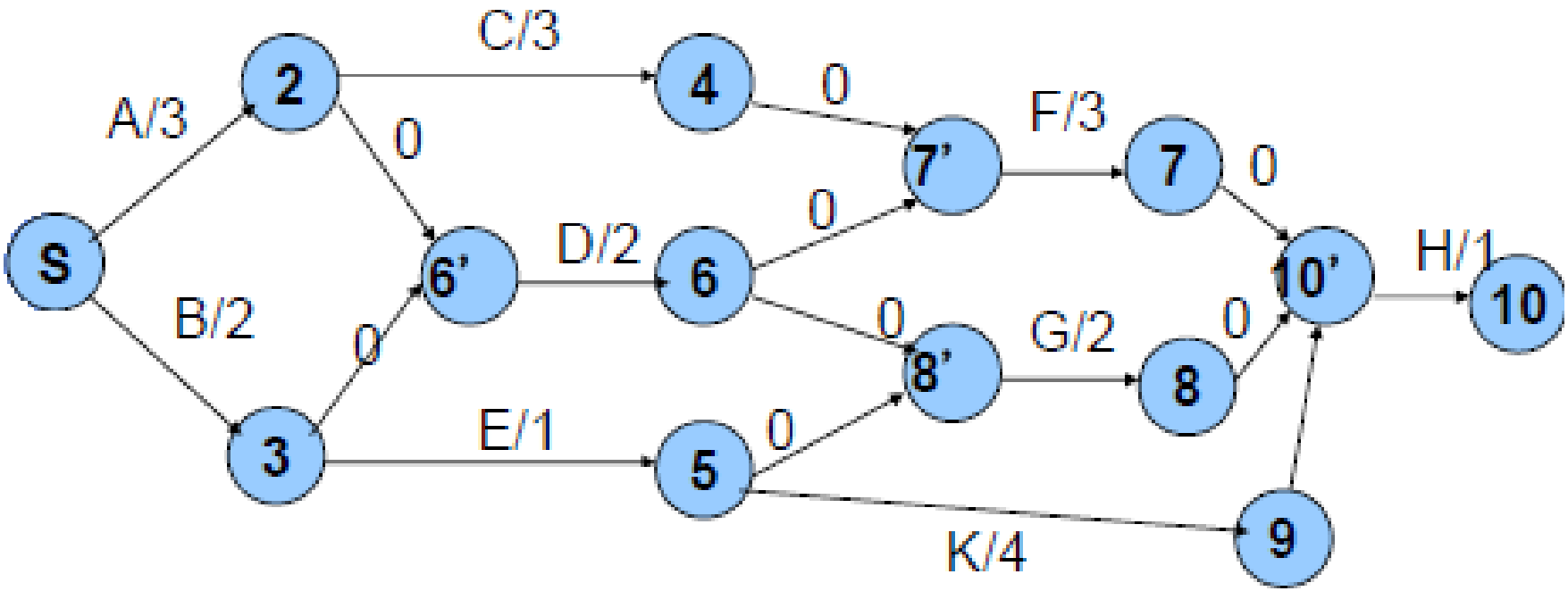
COMPUTATION OF EE

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	0	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										

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

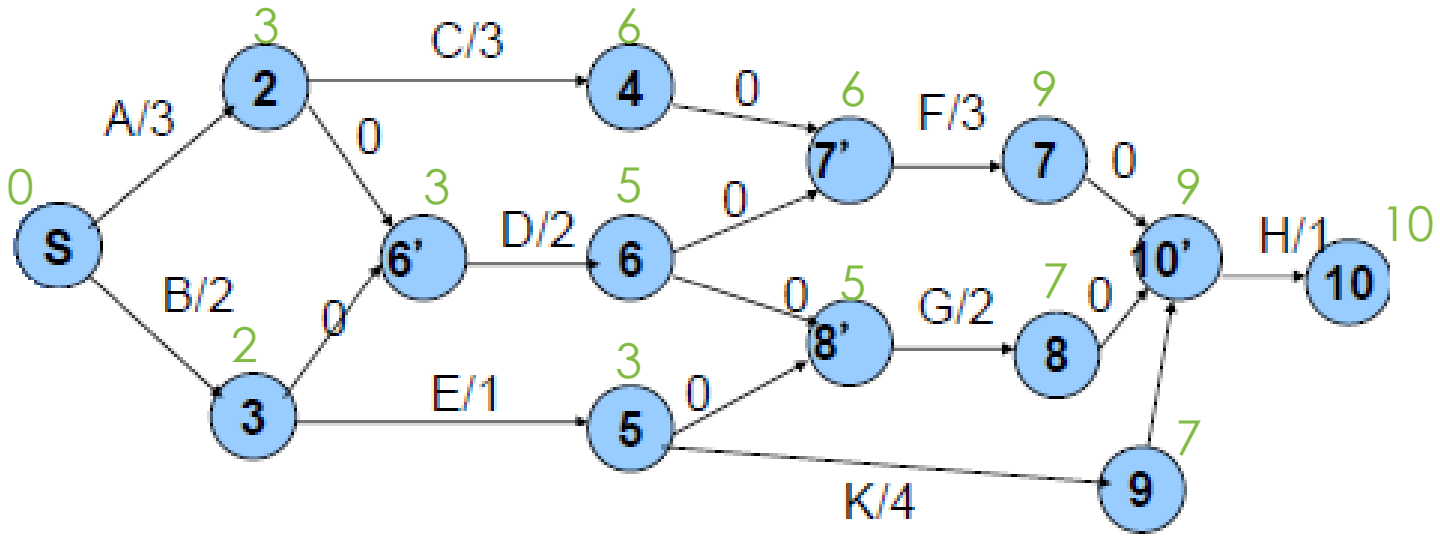
CRITICAL PATH ANALYSIS (CONT'D)

- AOE graph



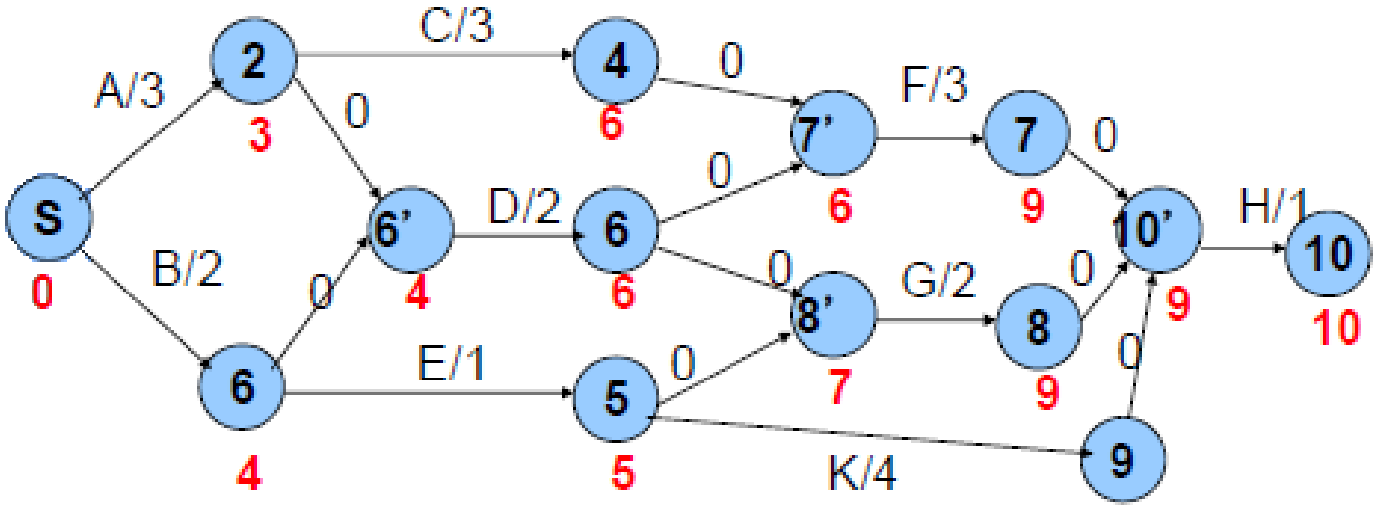
CRITICAL PATH ANALYSIS (CONT'D)

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



CRITICAL PATH ANALYSIS (CONT'D)

- Latest completion times:
  - latest time without affecting final completion time
  - computed by reverse topological order
  - $LE_{10} = EE_{10}$
  - $LE_v = \min(LE_w - Dv_{,w})$



CRITICAL PATH ANALYSIS (CONT'D)

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