Graphs

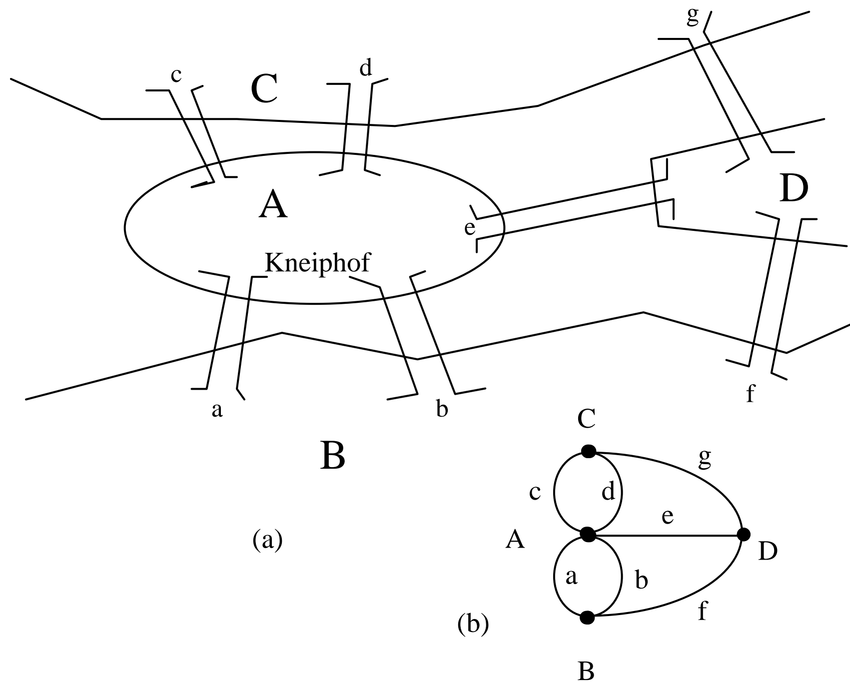
Introduction to Data Structures

Kyuseok Shim

SoEECS, SNU.

6.1 Graph Abstract Data Type

- Konigsberg bridge problem



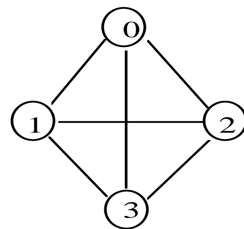
−Eulerian walk
Degree of each vertex is even

Figure 6.1 : (a) Section of the river Pregel in Konigsberg; (b) Euler's graph

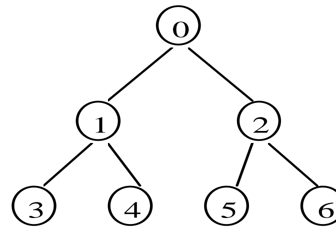
6.1 Graph Abstract Data Type (Cont.)

- Graph G=(V, E)
  - V is a finite, nonempty set of vertices
  - E is a set of edges
  - An edge is a pair of vertices
  - V(G) is the set of vertices of G
  - E(G) is the set of edges of G

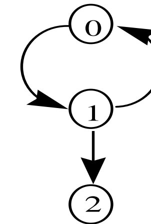
- Undirected (directed) graph
  - The pair of vertices representing an edge is unordered (ordered)



(a) G_1



(b) G_2

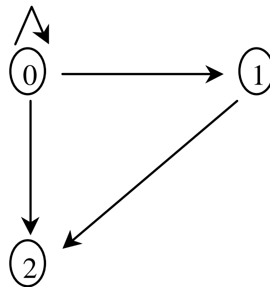


(c) G_3

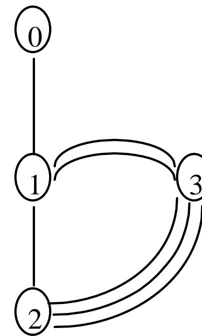
Figure 6.2 : Three sample graphs

6.1 Graph Abstract Data Type (Cont.)

- Restriction
  - A graph may not have an edge from a vertex back to itself
  - A graph may not have multiple occurrences of the same edge



(a) Graph with a self edge



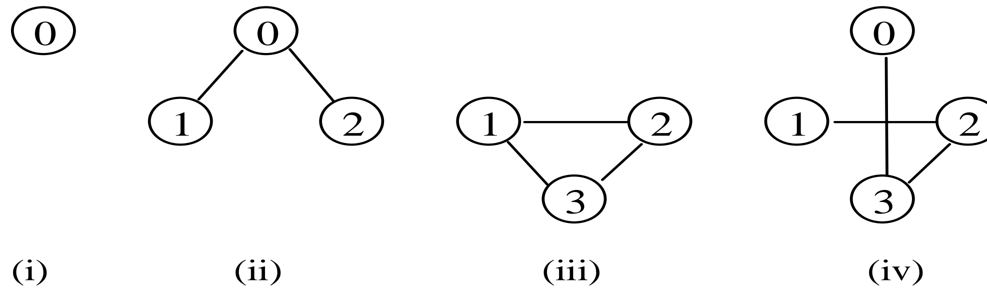
(b) Multigraph

Figure 6.3 : Examles of graphlike structures

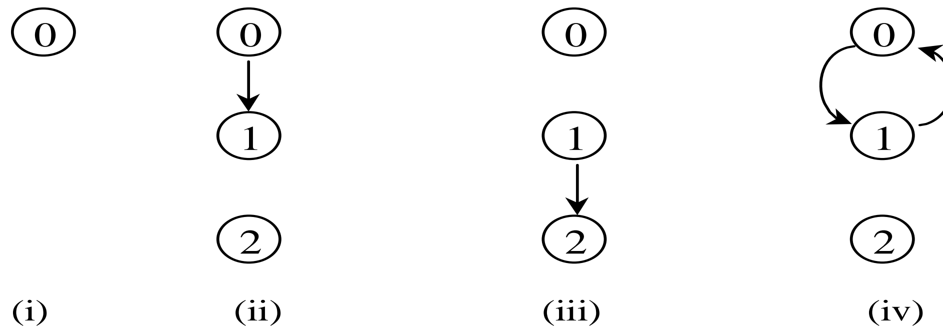
6.1 Graph Abstract Data Type (Cont.)

- Complete graph
  - n-vertex, undirected graph with n(n-1)/2 edges
- (u,v) is an edge in E(G)
  - Vertices u and v are adjacent
  - (u,v) is incident on vertices u and v
  - if (u, v) is a directed edge
    - u is adjacent to v
    - v is adjacent from u
- Subgraph of G
  - Graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

6.1 Graph Abstract Data Type (Cont.)



(a) some of the subgraphs of G_1



(b) some of the subgraphs of G_3

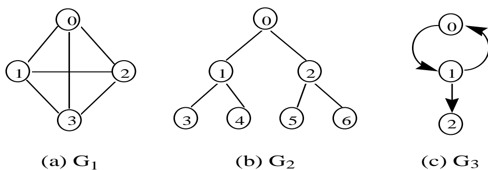


Figure 6.4 : Some subgraphs

6.1 Graph Abstract Data Type (Cont.)

- Path from u to v in G
  - A sequence of vertices u, i_1, i_2, ..., i_k, v such that (u,i_1) (i_1, i_2) ...(i_k, v) are edges in E(G)
  - Length of path is number of edges on it
  - Simple path is path in which all vertices except possibly the first and last are distinct
  - Cycle is simple path in which the first and last vertices are the same

6.1 Graph Abstract Data Type (Cont.)

- Vertices u and v are connected in (undirected) graph G, there is a path in G from u to v
- Connected graph
  - For every pair of distinct vertices u and v in V(G) there is a path from u and v
- (connected) Component
  - maximally connected subgraph
  - maximal: no more vertices or edges can be added while preserving its connectivity

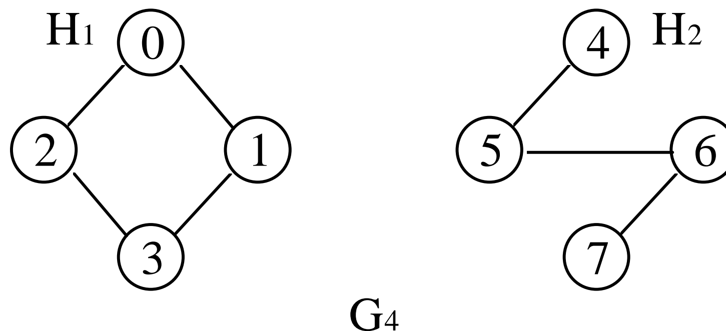


Figure 6.5 : A graph with two connected components

6.1 Graph Abstract Data Type (Cont.)

- Tree
  - Connected acyclic graph
- Degree of vertex
  - Number of edges incident to that vertex
- d_i is degree of vertex i in G with n vertices and e edges

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

6.1 Graph Abstract Data Type (Cont.)

```
1.  class Graph
2.  {// objects: A nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.
3.  public:
4.          virtual ~Graph() {}
5.                  // virtual destructor
6.          bool IsEmpty() const { return n == 0 };
7.                  // return true iff graph has no vertices
8.          int NumberOfVertices() const { return n };
9.                  // return number of vertices in the graph
10.         int NumberOfEdges() const { return e };
11.                 // return number of edges in the graph
12.         virtual int Degree(int u) const = 0;
13.                 // return number of edges incident to vertex u
14.         virtual bool ExistsEdge(int u, int v) const = 0;
15.                 // return true iff graph has the edge (u,v)
16.         virtual void InsertVertex(int v) = 0;
17.                 // insert vertex v into graph; v has no incident edges
18.         virtual void InsertEdge(int u, int v) = 0;
19.                 // insert edge (u,v) into graph
20.         virtual void DeleteVertex(int v) = 0;
21.                 // delete v and all edges incident to it
22.         virtual void DeleteEdge(int u, int v) = 0;
23.                 // deelte edge (u,v) from the graph
24. private:
25.                 int n;          //number of vertices
26.                 int e;          // number of edges
27. };
```

ADT 6.1 : Abstract data type Graph

6.1.3 Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Definition
  - G=(V,B) is a graph with n vertices, n≥1
  - Adjacency matrix A of G
    - two dimensional n x n array
    - A[i][j]=1 iff edge(i, j) is in E(G)

- Properties
  - Space needed is n^2
  - A is symmetric for undirected G
    - Need only upper or lower triangle of A

Adjacency Matrix (Cont.)

- Degree of vertex
  - $i = \sum_{j=0}^{n-1} A[i][j]$
- Complexity of operations :
  - $n^2 - n = O(n^2)$

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

(a) G_1

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

(b) G_3

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

(c) G_4

Figure 6.7 : Adjacency matrices

Adjacency Lists

- Representation
  - one list for each vertex in G
    - Nodes in list i represent vertices that are adjacent from vertex i
    - Each list has a head node
    - Vertices in a list are not ordered
  - Fields of node
    - data : index of vertex adjacent to vertex i
    - link
- Declaration in C++

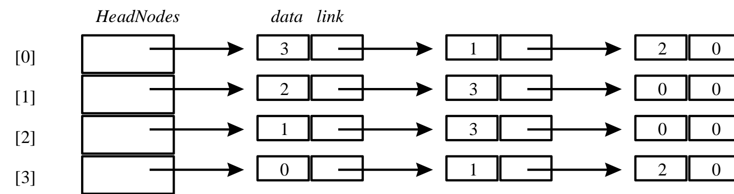
```
class Graph
{
private:
   List<int> *HeadNodes;
   int n;
public:
   Graph(const int vertices = 0) : n(vertices)
   { HeadNodes = new List<int>[n];};
};
```

Adjacency Lists (Cont.)

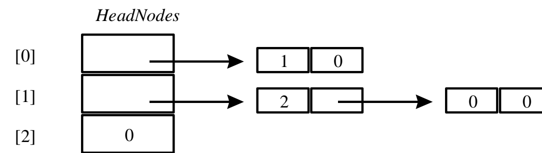
- For n vertices and edges
  - requires n head nodes and 2e list nodes

- Complexity of operations
  - # nodes in adjacency list = O(n+e)

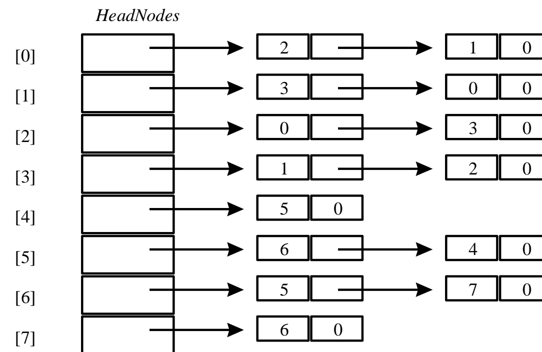
Adjacency Lists (Cont.)



(a) G1



(b) G3



(c) G4

Figure 6.8 : Adjacency lists

Adjacency Lists (Cont.)

- Packing nodes
  - Eliminate pointers
  - node[i] : Starting point of list for vertex i
  - Vertices adjacent from node i :
  - node[node[i]], ..., node[node[i+1]−1]

int $nodes[n + 2*e + 1]$;

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

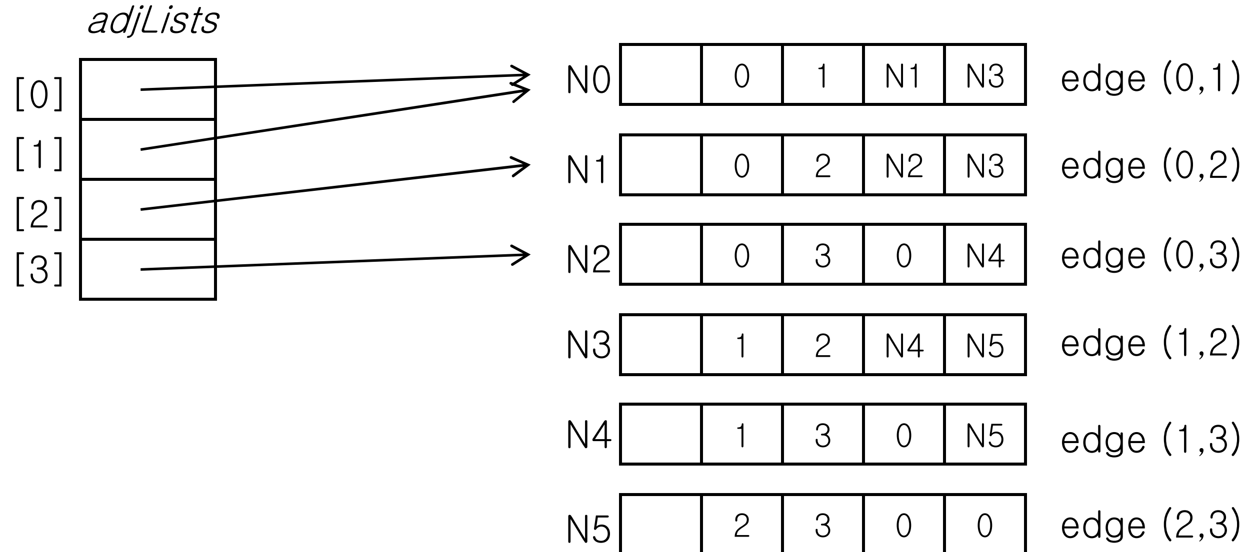
Figure 6.9 : Sequential representation of graph G4

Adjacency Multilists

- Property
  - For each edge, there will be exactly one node
  - But this node will be in two lists
- boolean mark field *m*
  - Indicate whether or not the edge has been examined
- storage requirement
  - Same as for normal adjacency lists except for the addition of mark bit
- node structure

m	*vertex1*	*vertex2*	*link1*	*link2*

Adjacency Multilists (Cont.)



The lists are

vertex 0: N0 -> N1 -> N2
vertex 1: N0 -> N3 -> N4
vertex 2: N1 -> N3 -> N5
vertex 3: N2 -> N4 -> N5

Figure 6.12:Adjacency multilists for G_1 of Figure 6.2(a)

Weighted Edges

- Network
  - Graph with weighted edges

- Representation
  - Adjacency matrix
    - A[i][j] keeps weight
  - Adjacency list
    - Additional field in list node keeps weight

6.2 Elementary Graph Operations

- Graph traversal
  - Given G=(V, E) and a vertex v in V(G)
  - Visit all vertices reachable from v

Depth-First-Search

- Procedure
  1. Visit start vertex v
  2. An unvisited vertex w adjacent to v is selected, and initiate DFS from w
  3. When u is reached such that all its adjacent vertices have been visited
     - Back up to the last vertex visited that has an unvisited vertex w adjacent to it
     - Initiate DFS from w
  4. Search terminates when no unvisited vertex can be reached from visited vertices

Depth-First-Search (Cont.)

```
1.  virtual void Graph::DFS() // Driver
2.  {
3.          visited = new bool[n];
4.                   // visited is declared as a bool* data member of graph
5.          fill(visited, visited + n, false);
6.          for(int v=0; v<n; v++)
7.               if(visited[v] == false)
8.                     DFS(v); // start search at vertex 0
9.          delete [] visited;
10. }

11. virtual void Graph::DFS(const int v) // Workhorse
12. { // Visit all previously unvisited vertices that are reachable from vertex v.
13.         visited[v] = true;
14.         for(each vertex w adjacent to v) // actual code uses an iterator
15.               if(!visited[w]) DFS(w);
16. }
```

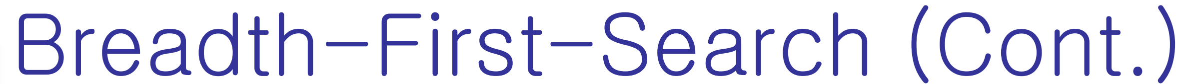
Program 6.1 : Depth-first search

Depth-First-Search (Cont.)

- Analysis
  - If adjacency list is used
    - Examines each node in the adjacency lists at most once
    - There are 2e list nodes
    - O(e)
  - If adjacency matrix is used
    - time to determine all adjacent vertices to v : O(n)
    - total time : $O(n^2)$

- Procedure
  1. visit start vertex v
  2. visit all unvisited vertices adjacent to v
  3. visit unvisited vertices adjacent to the newly visited vertices

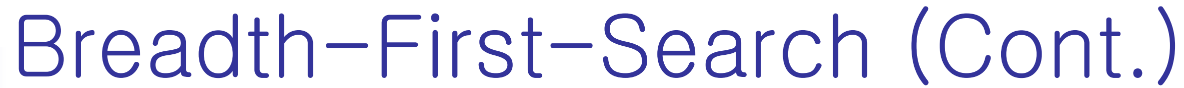
Breadth-First-Search (Cont.)



```
1.  virtual void Graph::BFS() // Driver
2.  {
3.          visited = new bool[n];
4.                  // visited is declared as a bool* data member of graph
5.          fill(visited, visited + n, false);
6.          for(int v=0; v<n; v++)
7.               if(visited[v] == false)
8.                    BFS(v); // start search at vertex 0
9.          delete [] visited;
10. }

11. virtual void Graph::BFS(int v)
12. { // A breadth first search of the graph is carried out beginning at vertex v.
13.    // visited[i] is set to true when v is visited. The function uses a queue.
14.         visited[v] = true;
15.         Queue<int> q;
16.         q.Push(v);
17.         while(!q.IsEmpty()) {
18.                 v = q.Front();
19.                 q.Pop();
20.                 for(all vertices w adjacent to v) // actual code uses an iterator
21.                         if(!visited[w]) {
22.                                 q.Push(w);
23.                                 visited[w] = true;
24.                         }
25.         } // end of while loop
26. }
```

Program 6.2 : Breadth-first search

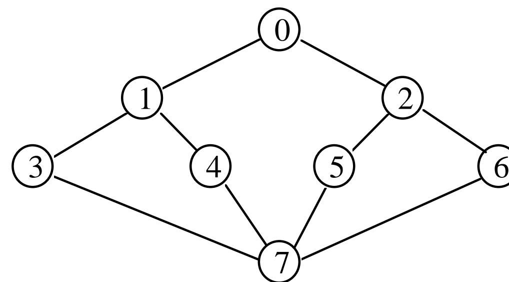


Breadth-First-Search (Cont.)

- Analysis
  - adjacency matrix : $O(n^2)$
  - adjacency list : $O(e)$

Example (DFS and BFS)

- DFS
  - 0 -> 1 -> 3 -> 7 -> 4 -> 5 -> 2 -> 6
- BFS
  - 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7



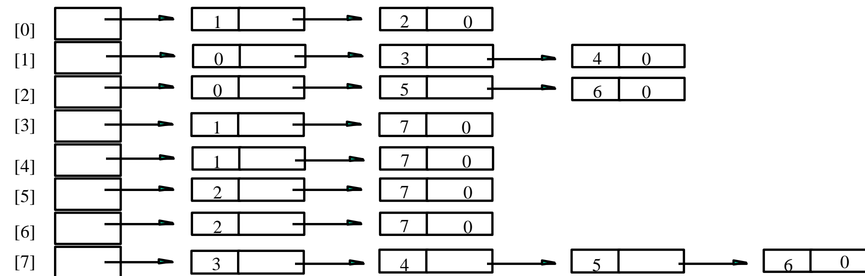
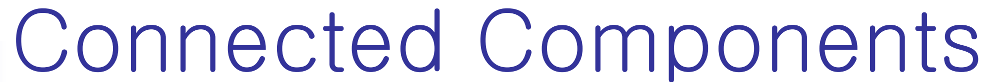


Figure 6.17 : Graph G and its adjacency lists



Connected Components

- If G is an undirected Graph, one can know its connectivity by simply making a call to either DFS or BFS
  - Making a call to either DFS or BFS and then determining if there is any unvisited vertex
- Determining connected components
  - Obtained by making repeated calls to either DFS or BFS
  - Start with a vertex that has not yet been visited

Connected Components (Cont.)

```
1.  virtual void Graph::Components()
2.  { // Determine the connected components of the graph.
3.          // visited is assumed to be declared as a bool* data member of Graph
4.          visited = new bool[n];
5.          fill(visited, visited + n, false);
6.          for(i =0; i < n; i++)
7.                  if(!visited[i]) {
8.                          DFS(i); // find a componenet
9.                          OutputNewComponents();
10.                 }
11.         delete [] visited;
12. }
```

Program 6.3 : Determining connected components

Connected Components (Cont.)

- Analysis
  - adjacency matrix : $O(n^2)$
  - adjacency list : $O(n+e)$

Spanning Trees

- Spanning tree
  - Tree consisting of edges in G and including all vertices
  - Depth-First-Spanning tree
  - Breadth-First-Spanning tree

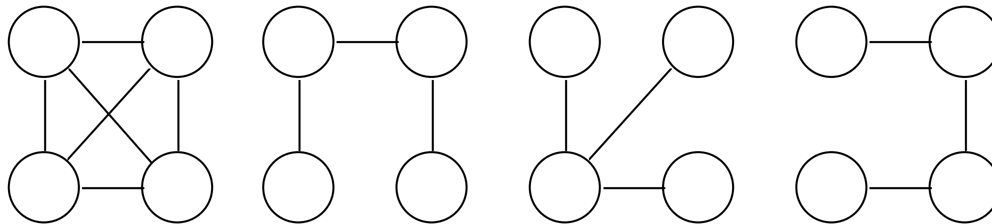


Figure 6.18 : A complete graph and three of its spanning trees

Spanning Trees (Cont.)

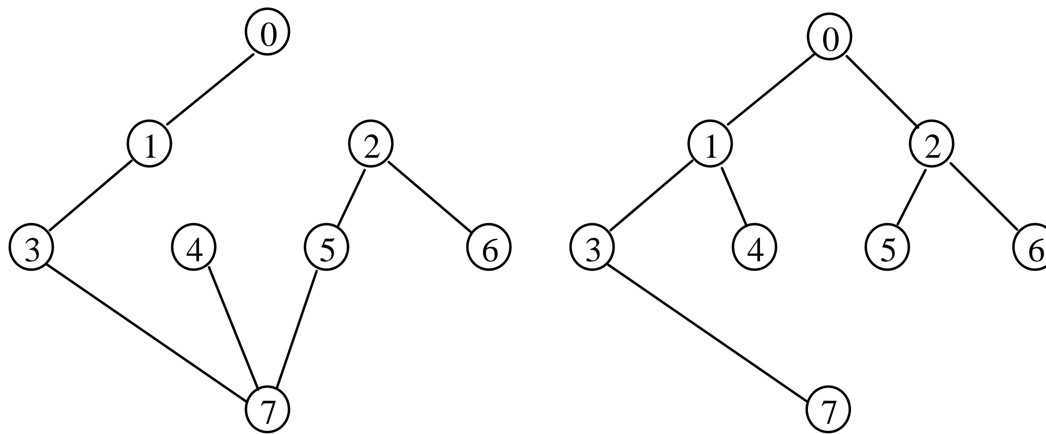


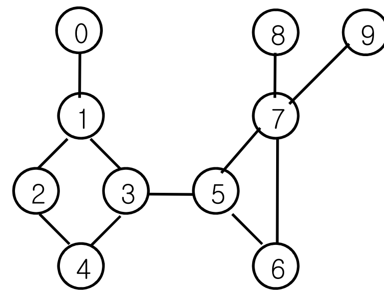
Figure 6.19 : Depth-first and breadth-first spanning trees for graph of Figure 6.17

- Properties
  - If a non tree edge is introduced into any spanning tree, then a cycle is formed
    - Ex) If (7,6) edge is added to Fig 6.19(a), then the resulting cycle is 7,6,2,5,7
    - Used to obtain an independent set of circuit equations for an electrical network
  - A Spanning tree is a minimal subgraph G' of G such that
    - V(G') = V(G)
    - G' is connected
  - Spanning tree has n-1 edges

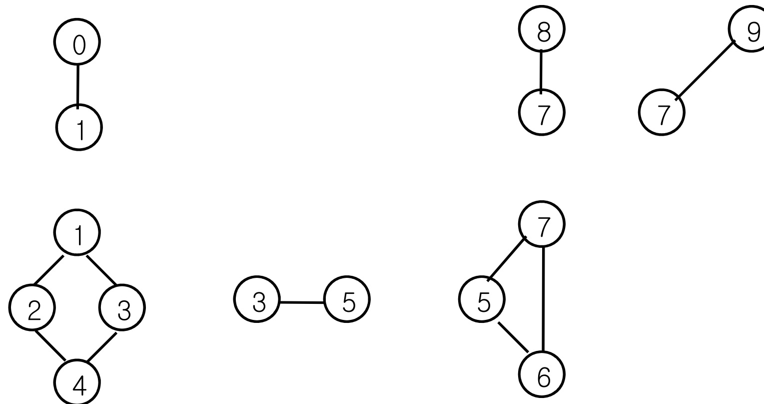
Biconnected Components

- Articulation point
  - A vertex v whose deletion operation leaves behind a graph that has at least two connected components
- Biconnected graph
  - A connected graph that has no articulation points
- Biconnected component
  - Maximal biconnected subgraph
  - The original graph contains no other biconnected subgraph

Biconnected Components (Cont.)



(a) A connected graph



(b) Its biconnected components

Figure 6.20: A connected graph and its biconnected components

Determining Biconnected components

- Depth-First Number
  - The sequence in which the vertices are visited during the DFS
- Back edge (u, v)
  - Nontree edge
  - Either u is an ancestor of v or v is an ancestor of u
- low(w)
  - **min{** dfn(w), min{low(x)| x is a child of w},
    min{dfn(x)|(w, x) is a back edge} **}**
- Articulation point (2 cases)
  - vertex u is an articulation point iff
  1. If u is the root of the spanning tree and has two or more children
  2. If u has a child w such that low(w) $\geq$ dfn(u)

Determining Biconnected components (Cont.)

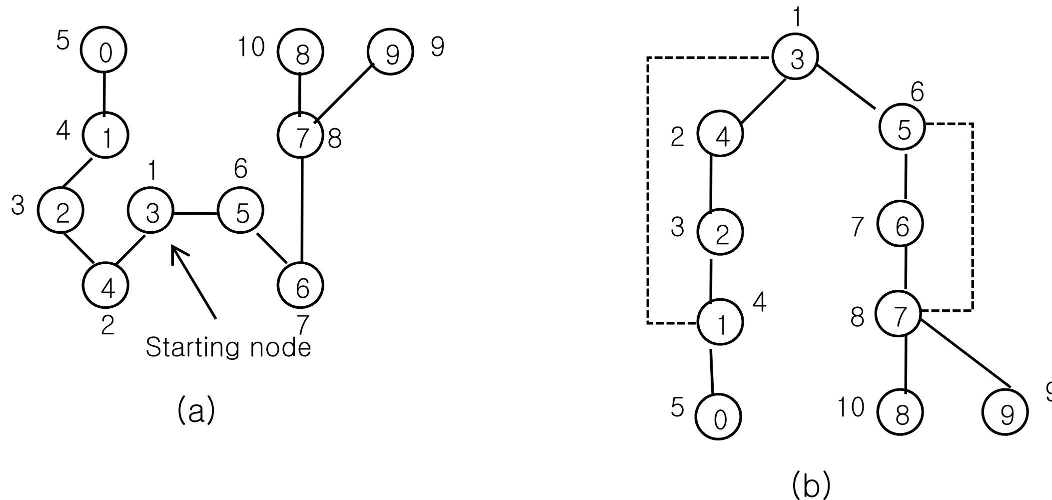


Figure 6.21: Depth-first spanning tree of Figure 6.20(a)

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

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

```
1.  virtual void Graph::DfnLow(const int x) // begin DFS at vertex x
2.  {
3.         num = 1;              // num is an int data member of Graph
4.         dfs = new int[n];     // dfn is declared as int* in Graph
5.         low = new int[n];     // low is declared as int* in Graph
6.         fill(dfn, dfn + n, 0);
7.         fill(low, low + n, 0);
8.         DfnLow(x, -1); // start at vertex x
9.         delete [] dfn;
10.        delete [] low;
11. }

12. void Graph::DfnLow(const int u, const int v)
13. { // Compute dfn and low while performing a depth first search beginning at vertex u.
14.   // v is the parent (if any) of u in the resulting spanning tree.
15.        dfn[u] = low[u] = num++;
16.        for(each vertex w adjacent from u) // actual code uses an iterator
17.               if(dfn[w] == 0) { // w is an unvisited vertex
18.                      DfnLow(w, u);
19.                      low[u] = min(low[u], low[w]);
20.               }
21.               else if(w != v) low[u] = min(low[u], dfn[w]); // back edge
22. }
```

Program 6.4: Computing dfn and low

Determining Biconnected components (Cont.)

```
1.  virtual void Graph::Biconnected()
2.  {
3.          num = 1;                              // num is an int datamember of Graph
4.          dfn = new int[n];   // dfn is declared as int* in Graph
5.          low = new int[n];   // low is declared as int* in Graph
6.          fill(dfn, dfn + n, 0);
7.          fill(low, low + n, 0);
8.          Biconnected(0,-1); // start at vertex 0
9.          delete [] dfn;
10.         delete [] low;
11. }
12. virtual void Graph::Biconnected(const int u, const int v)
13. { // Compute dfn and low, and output the edges of G by their biconnected components.
14.   // v is the parent (if any) of u in the resulting spanning tree.
15.   // s is an initially empty stack declared as a data member of Graph.
16.         dfn[u] = low[u] = num++;
17.         for(each vertex w adjacent from u) { // actual code uses an iterator
18.                 if((v != w)&&(dfn[w]<dfn[u])) add (u,w) to stack s;
19.                 if(dfn[w] == 0) { // w is an unvisited vertex
20.                         Biconnected(w, u);
21.                         low[u] = min(low[u], low[w]);
22.                         if(low[w] >= dfn[u]) {
23.                                 cout << "New Biconnected Component: " << end;
24.                                 do {
25.                                         delete an edge from the stack s;
26.                                         let this edge be (x, y);
27.                                         cout <<  x << "," << y << endl;
28.                                 } while( (x,y) and (u,w) are not the same edge)
29.                         }
30.                 }
31.                 else if (w != v) low[u] = min(low[u], dfn[w]); // back edge
32. }
```

Program 6.5: Outputting biconected components when n > 1

6.3 Minimum-Cost Spanning Trees

- Cost of a spanning tree
  - Sum of the costs (weights) of the edges in the spanning tree
- Min-cost spanning tree
  - A spanning tree of least cost
- Greedy method
  - At each stage, make the best decision possible at the time
    - Based on either a least cost or a highest profit criterion
  - Make sure the decision will result in a feasible solution
    - Satisfy the constraints of the problem
- To construct min-cost spanning trees
  - Best decision : least-cost
  - Constraints
    - Use only edges within the graph
    - Use exactly n-1 edges
    - May not use edges that produce a cycle

Kruskal's Algorithm

- Theorem
  - Let G be any undirected, connected graph. Kruskal's algorithm generates a minimum-cost spanning tree.
- Proof
  - Kruskal’s method results in a spanning tree whenever a spanning tree exist
    - The only edges that get discared are thos that result in a cyle
    - The deletion of a single edge that is on a cycle of a connected graph results in a graph that is also connected
    - Hence, if G initially is connected, the algorithm form a connected graph
  - The constructed spanning tree, T, is minimum cost
    - Since G has a finite number of spanning trees, it must have at least one of minimum cost. Let U be a minimum cost spanning tree.
    - We assume T is not U. Let k, k>0, be the number of edges in T not in U
    - We shall show that T and U have the same cost by transforming U into T. At each step, k will be reduced by 1. Further , the cost of U will not change as a result of the transformation
    - As a result, U will have the same cost as the initial U and will consist of exactly those edges that are in T. This implies that T is of minimum cost

Kruskal's Algorithm (Cont.)

- Each step involves adding to U one edge, e, from T and removing one edge, f, from U
  - Let e be the least-cost edge in T not in U. Such an edge must exist as k>0
  - When e is added to U, a unique cycle is created. Let f be any edge on this cycle that is not in T. Node that at least one of the edges on this cycle is not in T, as T contains no cycles
  - V = U + {e} – {f}
- We need to show that the cost of V is the same as that of U
- The cost of V is the cost of U plus the cost of e minus the cost of f
- The cost of e cannot be less than the cost of f, as otherwise the spanning tree V has a smaller cost than U, which is impossible
- If e has a higher cost than f, then f is considered before e by Kruskal's algorithm. Since f is not in T, Kruskal's algorithm must have discarded this edge at this time
- Hence, f, together with edges in T having a cost less than or equal to the cost of f must form a cycle. By the choice of e, all these edges are also in U. Hence, U must also contain a cycle. But it does not, as it is a spanning tree
- So, the assumption that e is of higher cost than f leads to a contradiction. The only possibility that remains is that e and f have the same cost. Hence, V has the same cost as U

Kruskal's Algorithm (Cont.)

- Procedure
  - Build a min-cost spanning tree T by adding edges to T one at a time
  - Select edges for inclusion in T in nondecreasing order of their cost
  - Edge is added to T if it does not form a cycle

Kruskal's Algorithm (Cont.)

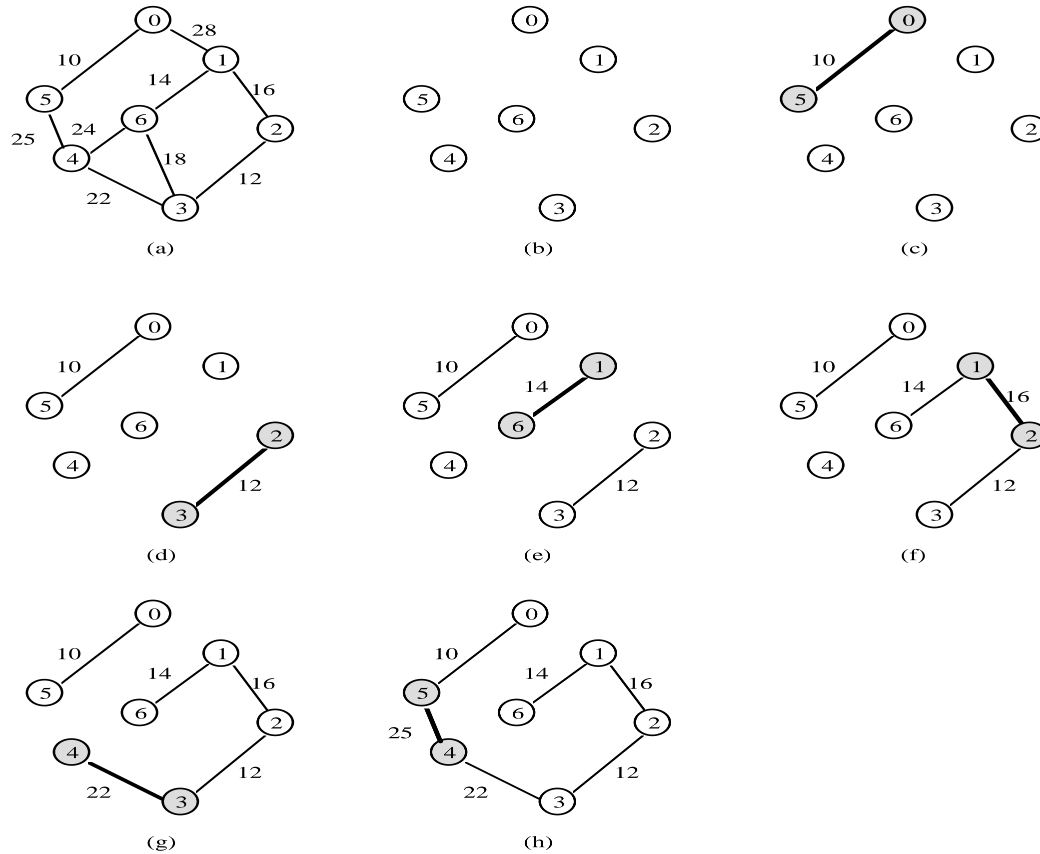


Figure 6.23 : Stages in Kruskal's algorithm

```
1.  T = Φ;
2.  while( (T contains less than n-1 edges) && (E not empty) ) {
3.        choose an edge (v,w) from E of lowest cost;
4.        delete (v,w) from E;
5.        if( (v,w) does not create a cycle in T ) add (v,w) to T;
6.        else discard (v,w);
7.  }
8.  if (T contains fewer than n-1 edge) cout << "no spanning tree" << endl;
```

Program 6.6: Kruskal's algorithm

- Time Complexity
  - When we use a min heap to determine the lowest cost edge , O(eloge)

Prim's Algorithm

- Property
  - At all times during the algorithm the set of selected edges forms a tree
- Procedure
  - Begin with a tree T that contains a single vertex
  - Add a least-cost edge (u,v) to T such that $T \cup \{(u,v)\}$ is also a tree
  - Repeat until T contains n-1 edges

Prim's Algorithm (Cont.)

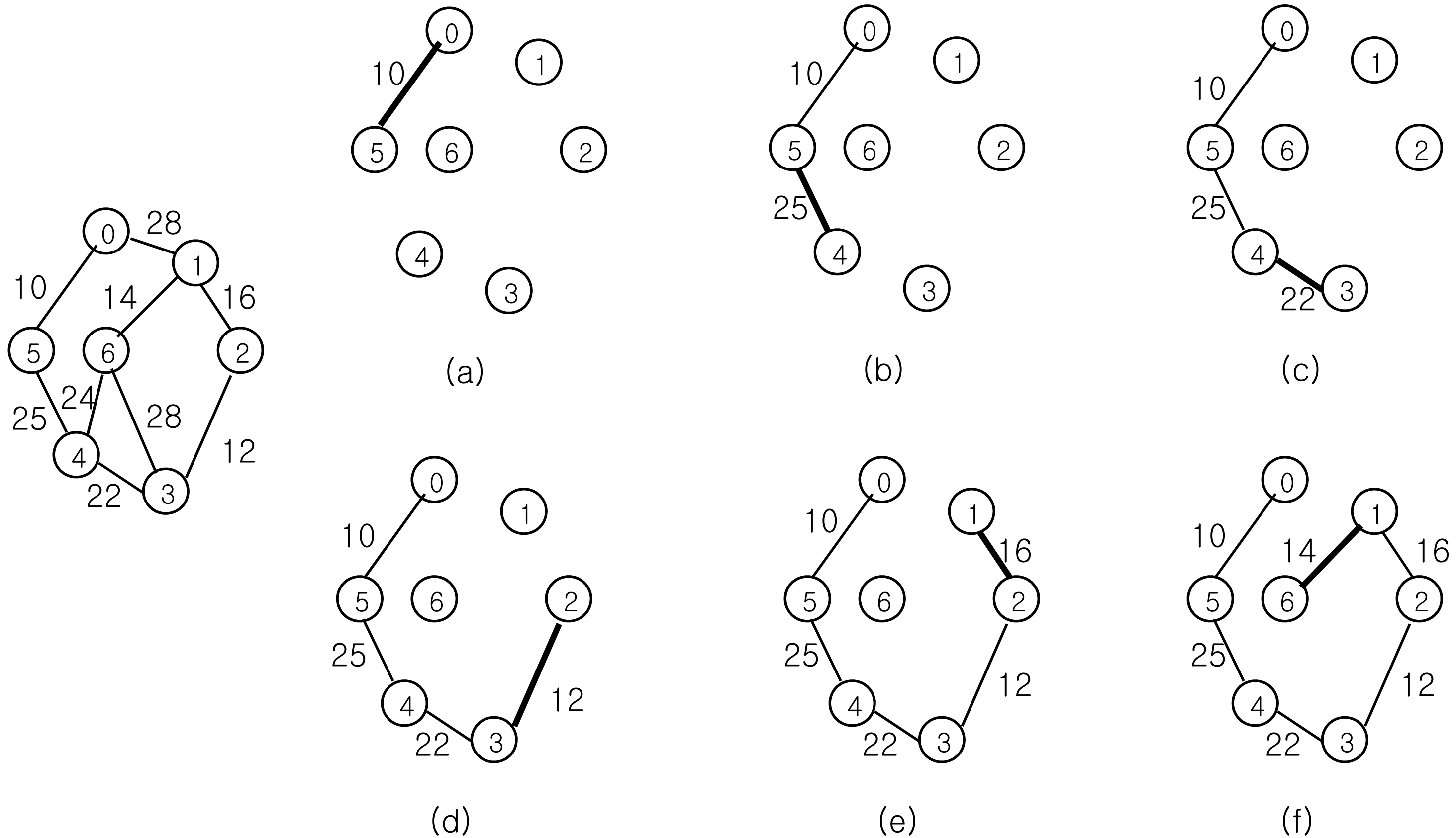


Figure 6.24: Stages in Prim's algorithm

Prim's Algorithm (Cont.)

```
1.  // Assume that G has at least one vertex.
2.  TV = { 0 };  // start with vertex 0 and no edges
3.  for(T = Φ; T contains fewer than n-1 edges; add (u,v) to T)
4.  {
5.      Let (u,v) be a least-cost edge such that u ∈ TV and v !∈ TV;
6.      if(there is no such edge) break;
7.      add v to TV;
8.  }
9.  if(T contains fewer than n-1 edges) cout << "no spanning tree" << endl;
```

Program 6.7: Prim's algorithm

- Time Complexity
  - $O(n^2)$
  - Asymptotically faster implementations are also possible
    - Fibonacci heaps, …
    - O(eloge)

- Form a spanning forest at each stage
- During a stage select one edge for each tree in this forest
- Procedure
  - Begin with single vertices
  - Select minimum cost edge at each vertex
  - Add distinct edges
  - Repeat until there is only one tree or no edges remain to be selected

Sollin's Algorithm (Cont.)

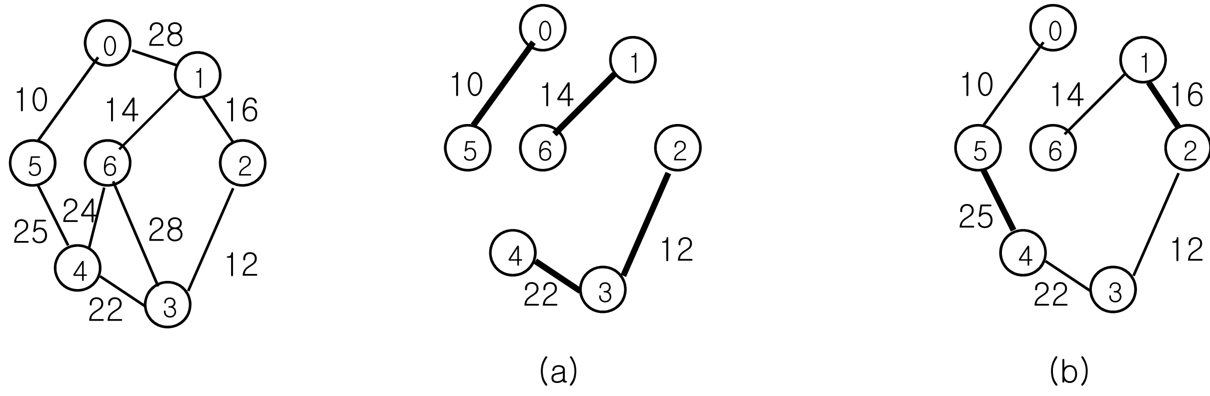


Figure 6.25: Stages in Sollin's algorithm

- Single Source/All Destinations: Nonnegative Edge Costs
- Single Source/All Destinations: General Weights
- All-Pairs Shortest Paths
- Transitive Closure

Single Source/All Destinations: Nonnegative Edge Costs

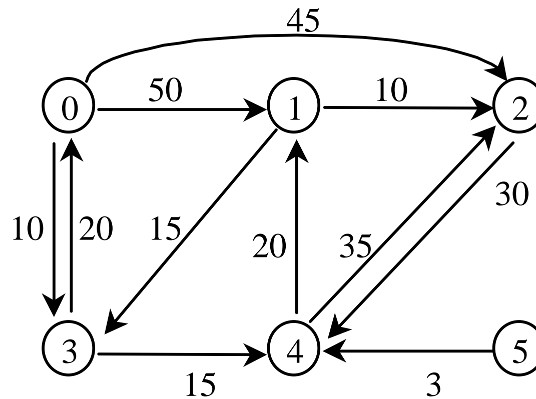
- Given
  - A digraph G=(V, E)
  - A length function length(i, j), length(i, j) ≥ 0, for edges of G
  - A source vertex v
- Problem
  - Determine a shortest path from v to each of the remaining vertices of G

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

- Definition of class Graph

```
class Graph
{
private:
  int length[nmax][nmax];
  int dist[nmax];
  Boolean s[nmax];
public:
  void ShortestPath(const int, const int);
  int choose(const int);
};
```

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



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

(a) graph

(b) shortest paths from 0

Figure 6.26 : Graph and shortest paths from vertex 0

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

- S: set of vertices to which the shortest paths have already been found
- dist[w], for w not in S
  - The length of the shortest path starting from v
  - Go through only the vertices that are in S
  - End at w
- A greedy algorithm will generate the shortest paths in nondecreasing order of path length

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

- We observe that when paths are generated in nondecreasing order of length
  1. If the next shortest path is to vertex u, then the path goes through only vertices that are in S
     - All of the intermediate vertices on the shortest path must be in S
     - Proof) Assume there is a vertex w on this path that is not in S. Then the v-to-u path also contains a path from v to w that is less than that of the v-to-u path. But, by the observation, paths are generated in nondecreasing order of length. so, the shorter path from v to w has been generated already. Hence, there is no intermediate vertex that is not in S
  2. The destination of the next path generated must be the vertex u that has the minimum distance, dist[u], among all vertices not in S
     - This follows from the definition of dist and observation ①
     - If there are several vertices not in S with the same dist, then any of these may be selected
  3. The vertex u selected in ② becomes a member of S. At this point, the length of the shortest paths starting at v, going through vertices only in S, and ending at a vertex w not in S may decrease. Therefore, if dist[w] decreases, then the change is due to the path from v to u to w. The length of this path is dist[u]+length(<u,w>)

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

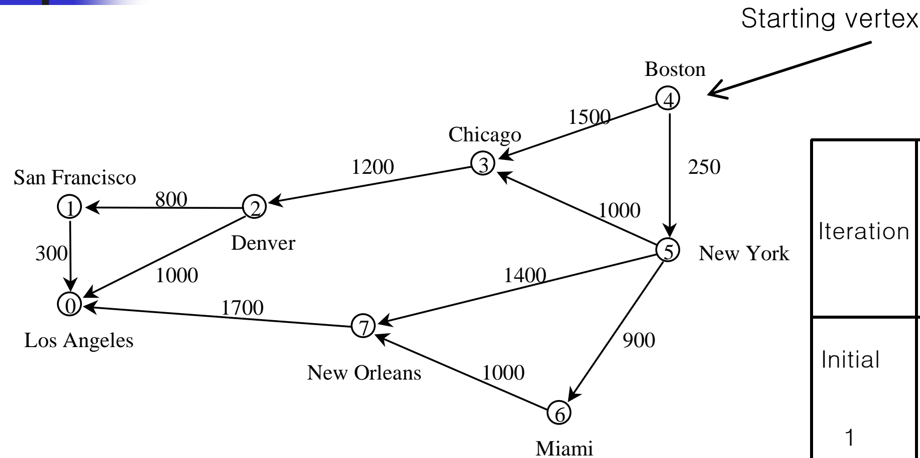
```
 void MatrixWDigraph::ShortestPath(const int n, const int v)
{ // dist[j], 0 ≤ j < n, is set to the length of the shortest path from v to j
  // in a digraph G with n vertices and edge lengths given by length[i][j].
      for(int i = 0; i < n; i++) { s[i] = false; dist[i] = length[v][i]; } // initialize
      s[v] = true;
      dist[v] = 0;

      for(i = 0; i < n-2; i++) { // determine n-1 paths from vertex v
            int u = Choose(n); // .returns 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; ···)
}
```

Program 6.8 : Determining the shortest paths

- Time Complexity
  - $O(n^2)$, n: number of vertices

Example 6.5



(a) Digraph

	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

(b) Length-adjacency matrix

Figure 6.27 : Digraph for Example 6.5

Iteration	S (shortest paths have already been found)	Vertex selected	Distance							
			LA [0]	SF [1]	DEN [2]	CHI [3]	BOST [4]	NY [5]	MIA [6]	NO [7]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

Figure 6.28 : Action of ShortestPath on digraph of Figure 6.27

Single Source/All Destinations: General Weights

- Some or all of the edges of the directed graph G may have negative length

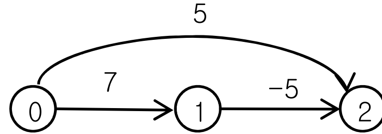


Figure 6.29: Directed graph with a negative-length edge

- Assume there are no cycles of negative length
  - 0 -> 1 -> 0 -> 1 -> … -> 2 => $-\infty$

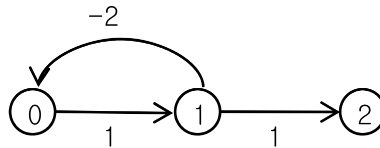


Figure 6.30: Directed graph with a cycle of negative length

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

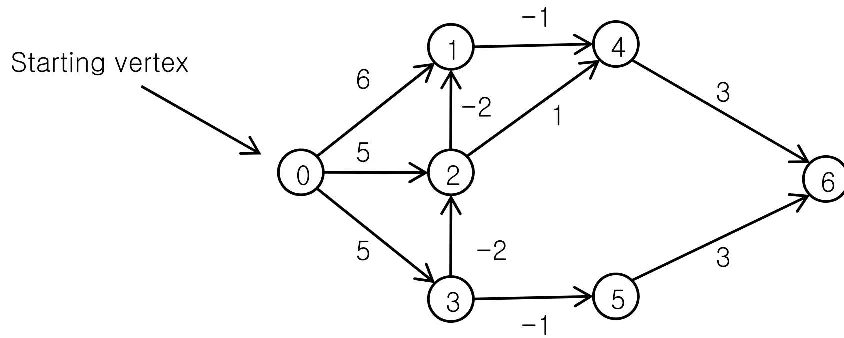
- $dist^k[u]$
  - The length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most k edges

- When there are no cycles of negative length, we can find exact shortest path which has at most n-1 edges

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

- Goal
  - Compute $dist^{n-1}[u]$ for all u
  - This can be done using dynamic programming methodology
- Observation
  - If the shortest path from v to u with at most k, k>1, edges has no more than k−1 edges
    - $dist^{k}[u] = dist^{k-1}[u]$
  - If the shortest path from v to u with at most k, k>1, edges has exactly k edges
    - It is comprised of a shortest path from v to some vertex j followed by the edge <j,u>. The path from v to j has k−1 edges, and its length is $dist^{k-1}[j]$
    - All vertices i such that the edge <i,u> is in the graph are candidates for j
    - $dist^{k}[u] = \min\{dist^{k-1}[u], \min_{i}\{dist^{k-1}[i]+length[i][u]\}\}$

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



(a) A directed graph

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

(b) $dist^k$

Figure 6.31: Shortest paths with negative edge lengths

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

```
1.  void MatrixWDigraph::BellmanFord(const int n, const int v)
2.  { // Single source all destination shortest paths with negative edge lengths
3.        for(int i=0; i<n; i++) dist[i] = length[v][i]; // initialize dist

4.        for(int k=2; k<=n-1; k++)
5.              for(each u such that u != v and u has at least one incoming edge)
6.                    for(each <i,u> in the graph)
7.                          if(dist[u] > dist[i] + length[i][u]) dist[u] = dist[i] + length[i][u];
8.  }
```

Program 6.9: Bellman and Ford algorithm to compute shortest paths

- Time Complexity
  - lines 5 to 7
    - $O(n^2)$ when adjacency matrices are used
    - O(e) when adjacency lists are used
  - Overall
    - $O(n^3)$ when adjacency matrices are used
    - O(ne) when adjacency lists are used

All-Pairs Shortest Paths

- Find the shortest paths between all pairs of vertices u and v
- Assume no cycles of negative length

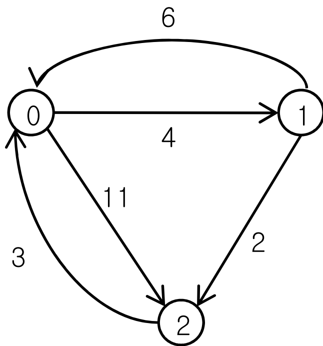
All-Pairs Shortest Paths (Cont.)

- The Graph G is represented by its length-adjacency matrix
- $A^k[i][j]$
  - The length of the shortest path from i to j
  - No intermediate vertex of index greater than k
  - $A^{n-1}[i][j]$ will be the length of the shortest i-to-j path
  - $A^{-1}[i][j]$ = length[i][j]

All-Pairs Shortest Paths (Cont.)

- Observation
  - The shortest path from i to j going through no vertex with index greater than k
    - The path does not go through the vertex with index k, so its length is $A^{k-1}[i][j]$
  - The shortest path goes through vertex k
    - The path consists of a subpath from i to k and another one from k to j
    - $A^{k}[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}, k \geq 0$

All-Pairs Shortest Paths (Cont.)



(a) Example digraph

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b) A^{-1}

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

(c) A^{0}

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

(d) A^{1}

A^{2}	0	1	2
0	0	4	11
1	5	0	2
2	3	7	0

(e) A^{2}

Program 6.32: Example for all-pairs shortest-paths problem

All-Pairs Shortest Paths (Cont.)

```
1.  void MatrixWDigraph::AllLengths(const int n)
2.  { // length [n][n] is the adjacency matrix of a graph with n vertices.
3.  // a[i][j] is the length of the shortest path between i and j.
4.        for(int i=0; i<n; i++)
5.              for(int j=0; j<n; j++)
6.                    a[i][j] = length[i][j];       // copy length into a
7.        for(k=0; k<n; k++)                         // for a path with highest vertex index k
8.              for(int i=0; i<n; i++)               // for all possible pairs of vertices
9.                    for(int j=0; j<n; j++)
10.                           if( (a[i][k]+a[k][j]) < a[i][j] ) a[i][j] = a[i][k] + a[k][j];
11. }
```

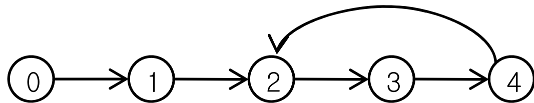
Program 6.10: All-pairs shortest paths

- Time Complexity
  - $O(n^3)$

Transitive Closure

- Transitive closure matrix A^+
  - a matrix such that $A^+[i][j] = 1$ if there is a path of length > 0 from i to j, otherwise $A^+[i][j] = 0$
- Reflexive transitive closure matrix A^*
  - a matrix such that $A^*[i][j] = 1$ if there is a path of length ≥ 0 from i to j, otherwise $A^*[i][j] = 0$
- We can use *AllLengths*(Program 6.10) to compute A^+
  - Begin with
    - $length[i][j] = 1$ if $<i,j>$ is an edge in G
    - $length[i][j] = LARGE$ if $<i,j>$ is not in G
  - When *AllLengths* terminates
    - $A^+[i][j] = 1$ iff $a[i][j] < LARGE$
    - otherwise $A^+[i][j] = 0$
  - We can get A^* by setting all diagonal elements equal to 1
  - Total time: $O(n^3)$

Transitive Closure (Cont.)



(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	1	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^*

Transitive Closure (Cont.)

- Undirected Graph Case
  - Can be found more easily from its connected components
  - Time complexity for determining the connected components: $O(n^2)$ for a adjacent matrix, $O(ne)$ for adjacent lists
  - For every pair of distinct vertices in the same component, $A^+[i][j] = 1$
  - On the diagonal, $A^+[i][i] = 1$ iff the component containing i has at least two vertices

6.5 Activity Networks

- Activity-on-Vertex(AOV) Networks

- Activity-on-Edge(AOE) Networks

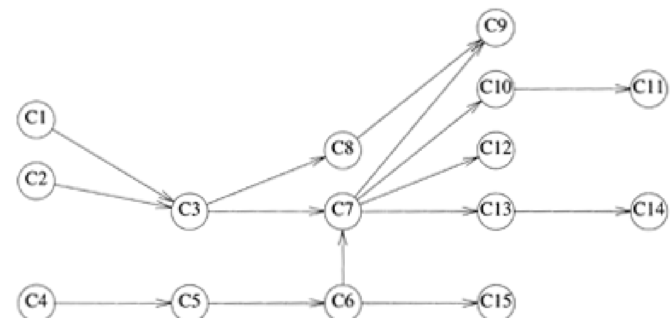
6.5.1 Activity-on-Vertex(AOV) Networks

- A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks

- Relationship of vertices
  - Predecessor
    immediate predecessor
    Successor
    immediate successor

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

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

Figure 6.36: An activity-on-vertex (AOV) network

Topological Order

- Linear ordering of the vertices of a graph
  - 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

- Example(topological order)
  - C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9
  - C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

Topological Order (Cont.)

```
1.  Input the AOV network. Let n be the number of vertices.
2.  for(int i=0; i<n; i++) // output the vertices
3.  {
4.       if(every vertex has a predecessor) return;
5.            // network has a cycle and is infeasible
6.       pick a vertex v that has no predecessors;
7.       cout << v;
8.       delete v and all edges leading out of v from the network;
9.  }
```

Program 6.11: Design of a topological sorting algorithm

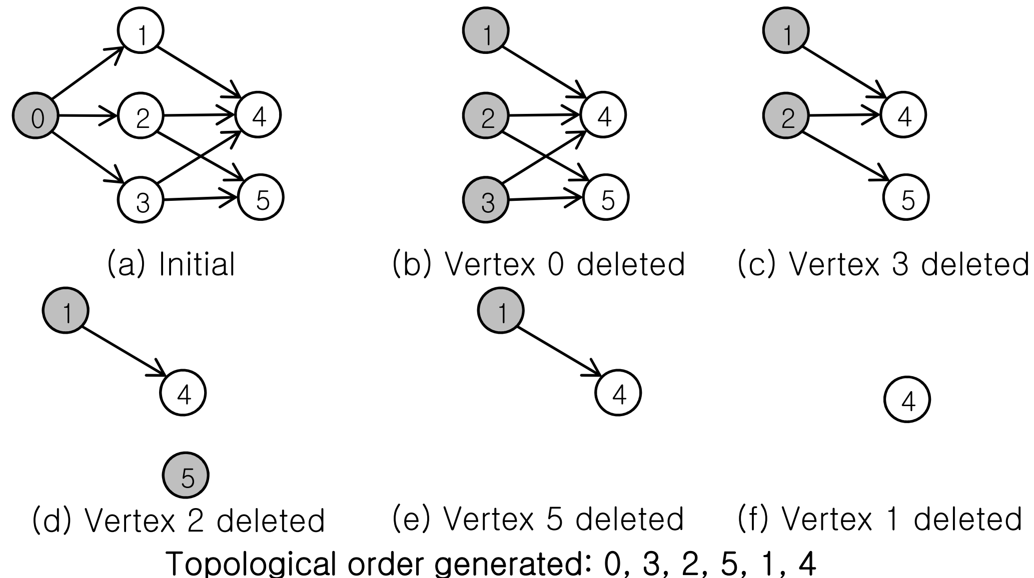


Figure 6.36 : Action of Program 6.11 on an AOV network (shaded vertices rep resent candidates for deletion)

Topological sorting algorithm

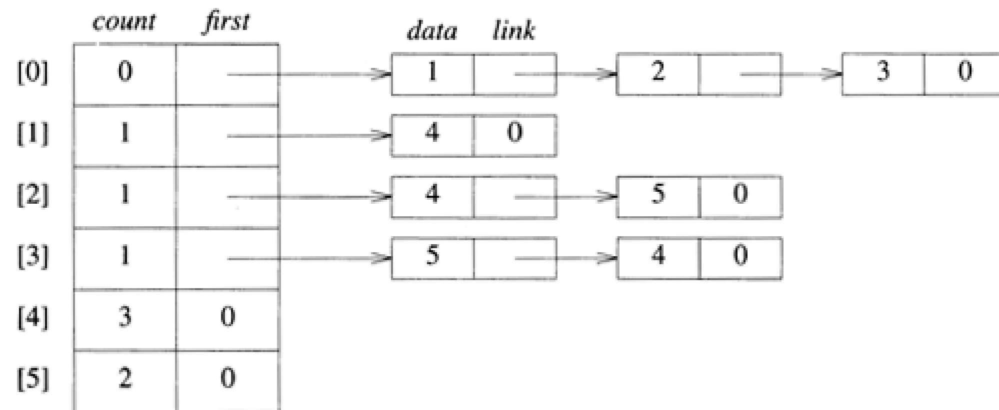


Figure 6.38 : Internal representation used by topological sorting algorithm

Topological sorting algorithm (Cont.)

```
1.  void LinkedDigraph::TopologicalOrder()
2.  { // The n vertices of a network are listed in topological order.
3.        int top = -1;
4.        for(int i=0; i<n; i++) //create a linked stack of vertices with
5.            if(count[i] == 0) { count[i] = top; top = i; }

6.        for(i=0; i<n; i++) {
7.            if(top == -1) throw "Network has a cycle.";
8.            int j=top; top = count[top]; // unstack a vertex
9.            cout << j << endl;
10.           Chain<int>::ChainIterator ji = adjLists[j].begin();
11.           while(ji) { // decrease the count of the successor vertices of j
12.               count[*ji]--;
13.               if(count[*ji] == 0) { count[*ji] = top; top = *ji; } // add *ji to stack
14.               ji++;
15.           }
16.       }
17. }
```

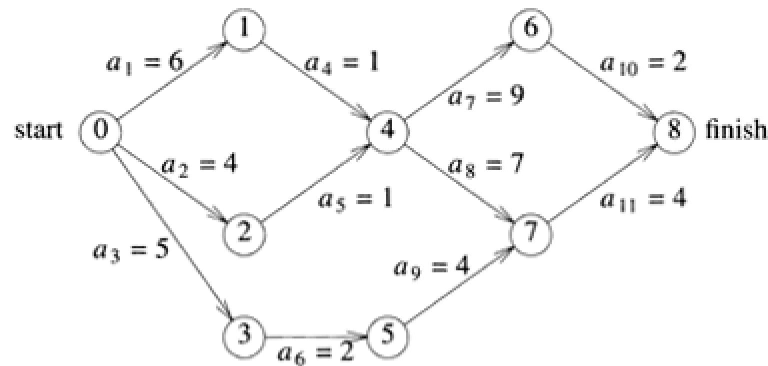
Program 6.11: Design of a topological sorting algorithm

- Time Complexity
  - lines 4 and 5: O(n)
  - lines 6 to 10: O(n) time over the entire algorithm
  - lines 11 to 15: $O(d_i)$, d_i is the out degree of vertex i
  - total time: $O(n + \sum_{i=0}^{n-1} d_i) = O(n + e)$

6.5.2 Activity-on-Edge(AOE) Networks

- The tasks to be performed on a project are represented by directed edges
- Vertices represent events
- An event occurs only when all activities entering it have been completed
- Events signal the completion of certain activities
- The number associated with each activity is the time needed to perform that activity

6.5.2 Activity-on-Edge(AOE) Networks (Cont.)



(a) Activity network of a hypothetical project

event	*interpretation*
0	start of project
1	completion of activity a_1
4	completion of activities a_4 and a_5
7	completion of activities a_8 and a_9
8	completion of project

(b) Interpretation of some of the events in the network of (a)

Figure 6.39 : An AOE network

6.5.2 Activity-on-Edge(AOE) Networks (Cont.)

- Activities can be carried out in parallel
  - The minimum time to complete the project is the length of the longest path

- Critical path
  - A path of longest length
  - A network may have more than one critical path
  - Example (6.39(a))
    - 0, 1, 4, 6, 8
    - 0, 1, 4, 7, 8

- Earliest time
  - The *earliest time* that an event i can occur
  - 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* e(i) for all activities leaving that vertex
  - Example (6.39(a))
    - e(i) represents the earliest start time for activity a_i
    - e(7) = e(8) = 7

6.5.2 Activity-on-Edge(AOE) Networks (Cont.)

- Latest time l(i)
  - The latest time that an activity a_i may start without increasing the project duration
  - Example (6.39(a))
    - e(6) = 5, l(6) = 8
    - e(8) = 7, l(8) = 7
- Criticality of an activity
  - l(i) – e(i)
- Critical activities
  - All activities for which e(i) = l(i)

Calculation of Early/late Activity Times

- Terms
  - ee[j]: earliest event time
  - le[j]: latest event time

- Earliest start time, Latest time
  - a_i: activity represented by edge<k, l>
  - e(i) = ee[k]
  - l(i) = le[l] – duration of activity a_i

6.5.2.1 Calculation of Early Activity Times

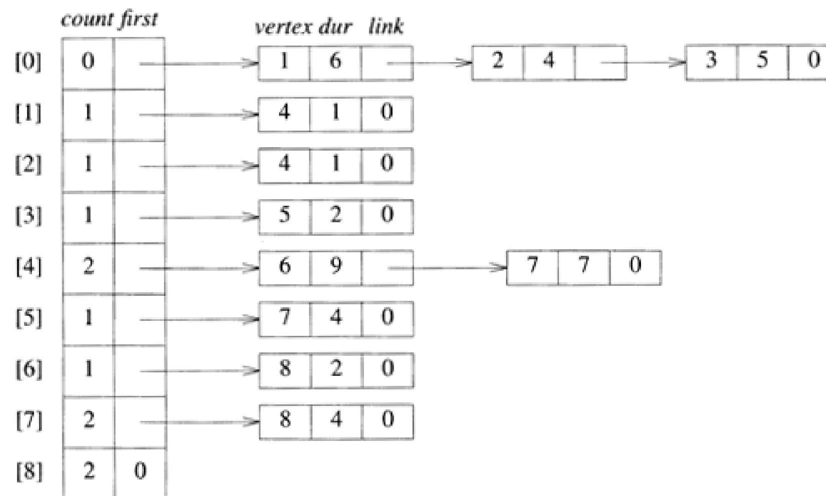
- e(i) = ee[k]

- ee[0] = 0
  $$ee[j] = \max_{i \in P(j)}\{ee[i] + \text{duration of } \langle i, j \rangle\}$$
  where P(j) is the set of all vertices adjacent to vertex j

- We can use *TopologicalOrder* (program 6.12) to compute the early event times
  - Modify so that it returns the vertices in topological order
  - Insert the line below between lines 12 and 13

```
ee[*ji] = max( ee[*ji], ee[j] + duration of <j, *ji>);
```

6.5.2.1 Calculation of Early Activity Times (Cont.)



(a) Adjacency lists for Figure 6.38(a)

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

(b) Computation of *ee*

Figure 6.40: Computing *ee* using modified *TopologicalOrder* (Program 6.12)

6.5.2.2 Calculation of Late Activity Times

- We start with le[n-1] = ee[n-1]
- $le[j] = \min_{i \in S(j)} \{ le[i] - \text{duration of } \langle j,i \rangle \}$
  - S(j) is the set of vertices adjacent from vertex j
- Before le[j] can be computed for some event j, the latest event time for all successor events must be computed
- Example (6.39(a))

le[8] = ee[8] = 18
le[6] = min{le[8]-2} = 16
le[7] = min{le[8]-4} = 14
le[4] = min{le[6]-9, le[7]-7} = 7
le[1] = min{le[4]-1} = 6
le[2] = min{le[4]-1} = 6
le[5] = min{le[7]-4} = 10
le[3] = min{le[5]-2} = 8
le[0] = min{le[1]-6, le[2]-4, le[3]-5} = 0

6.5.2.2 Calculation of Late Activity Times (Cont.)

Earliest start time: e(i) = ee[k]
Latest time l(i) = le[l] – duration of activity a_i

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

Figure 6.41 : Early, late, and criticality values

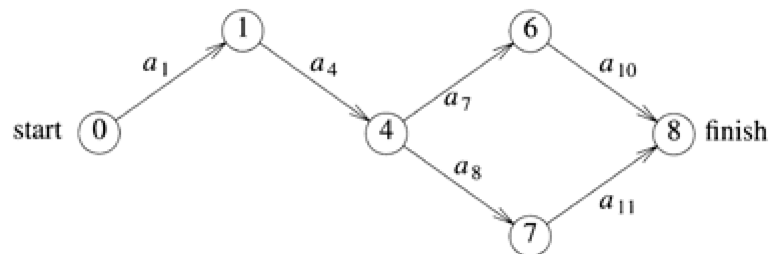


Figure 6.42 : Graph obtained after deleting all noncritical activities