Chap 6. Graph (4)

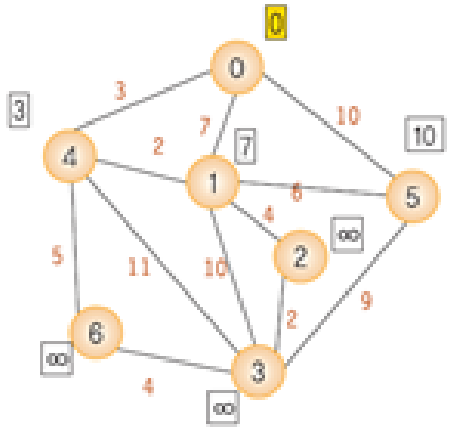
Contents

6.4 Shortest Path

- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

6.4 Shortest Path

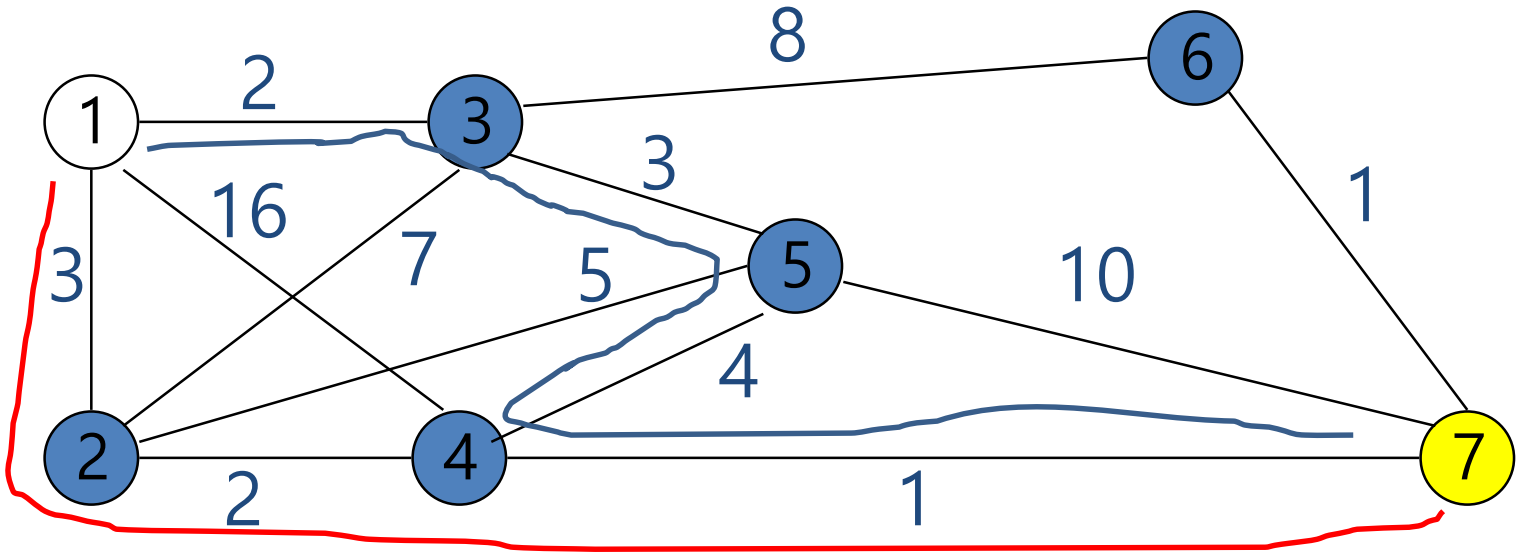
- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v.***
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between 0 and 3
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Single Source Single Destination

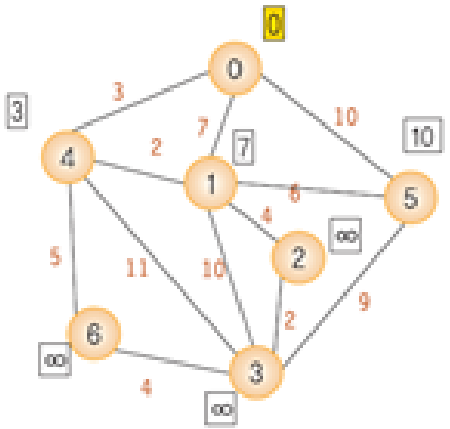
Possible algorithm:

- Leave source vertex using cheapest/shortest edge.
- Leave new vertex using cheapest edge subject to the constraint that a new vertex is reached.
- Continue until destination is reached.



Dijkstra's Algorithm(Single Source All Destinations)

- The distance of a vertex ***v*** from a vertex ***s*** is the length of a shortest path between ***s*** and ***v***
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex ***s***
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

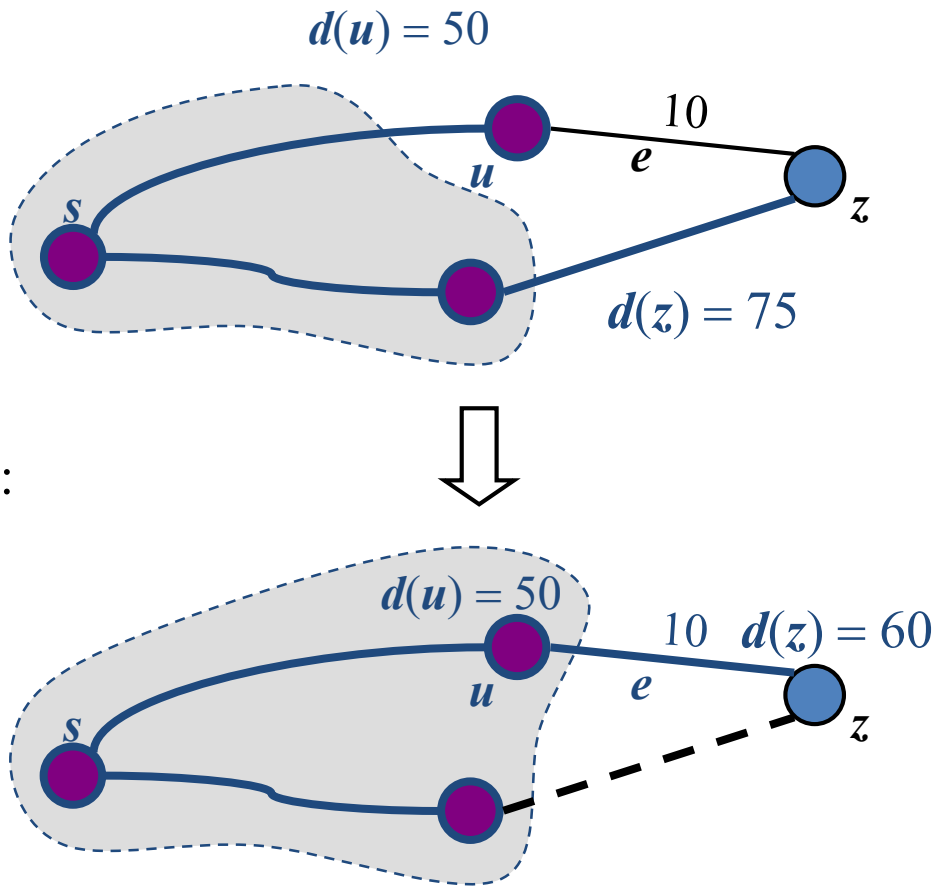


Edge Relaxation

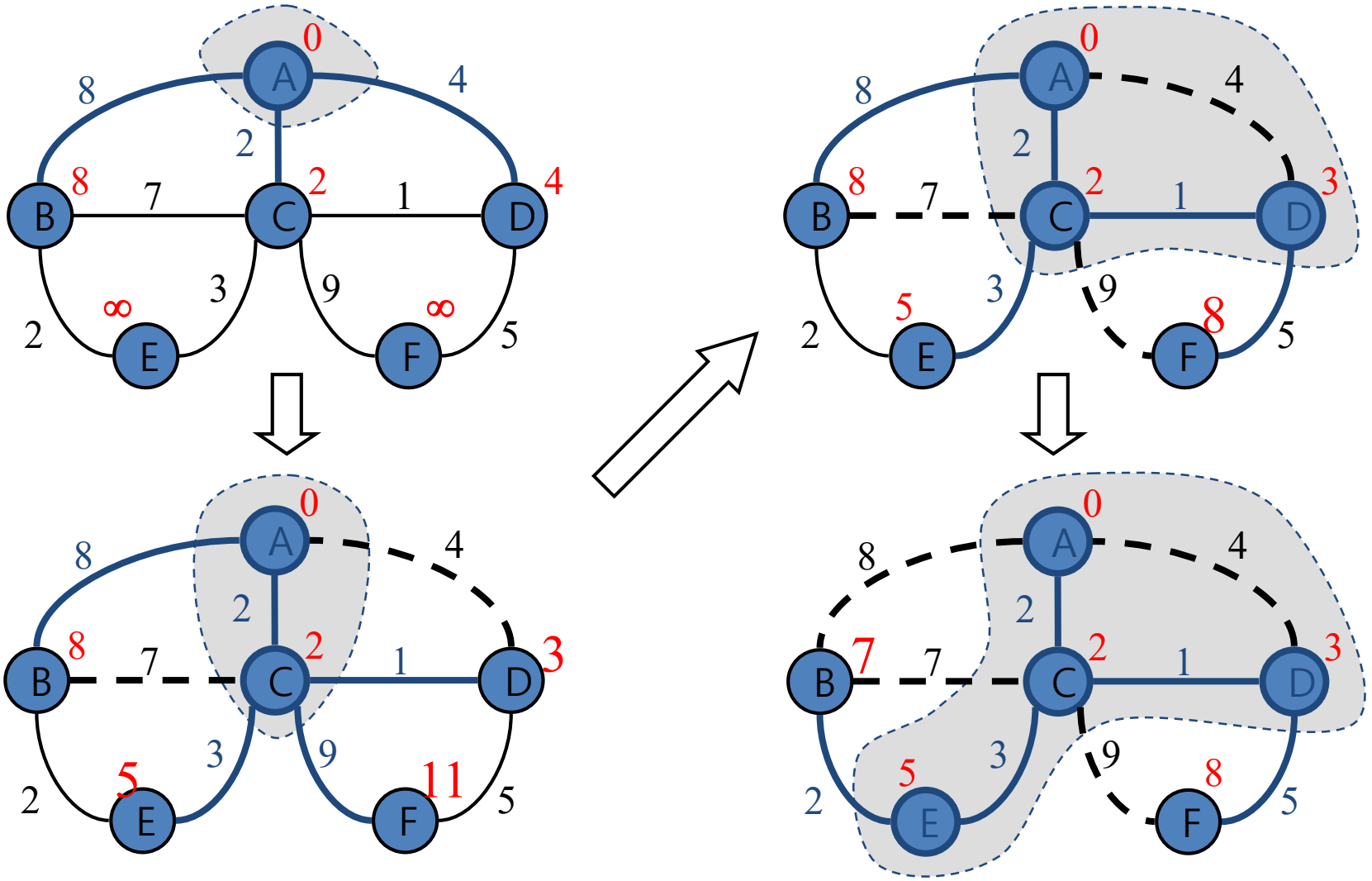
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud

- The relaxation of edge e updates distance $d(z)$ as follows:

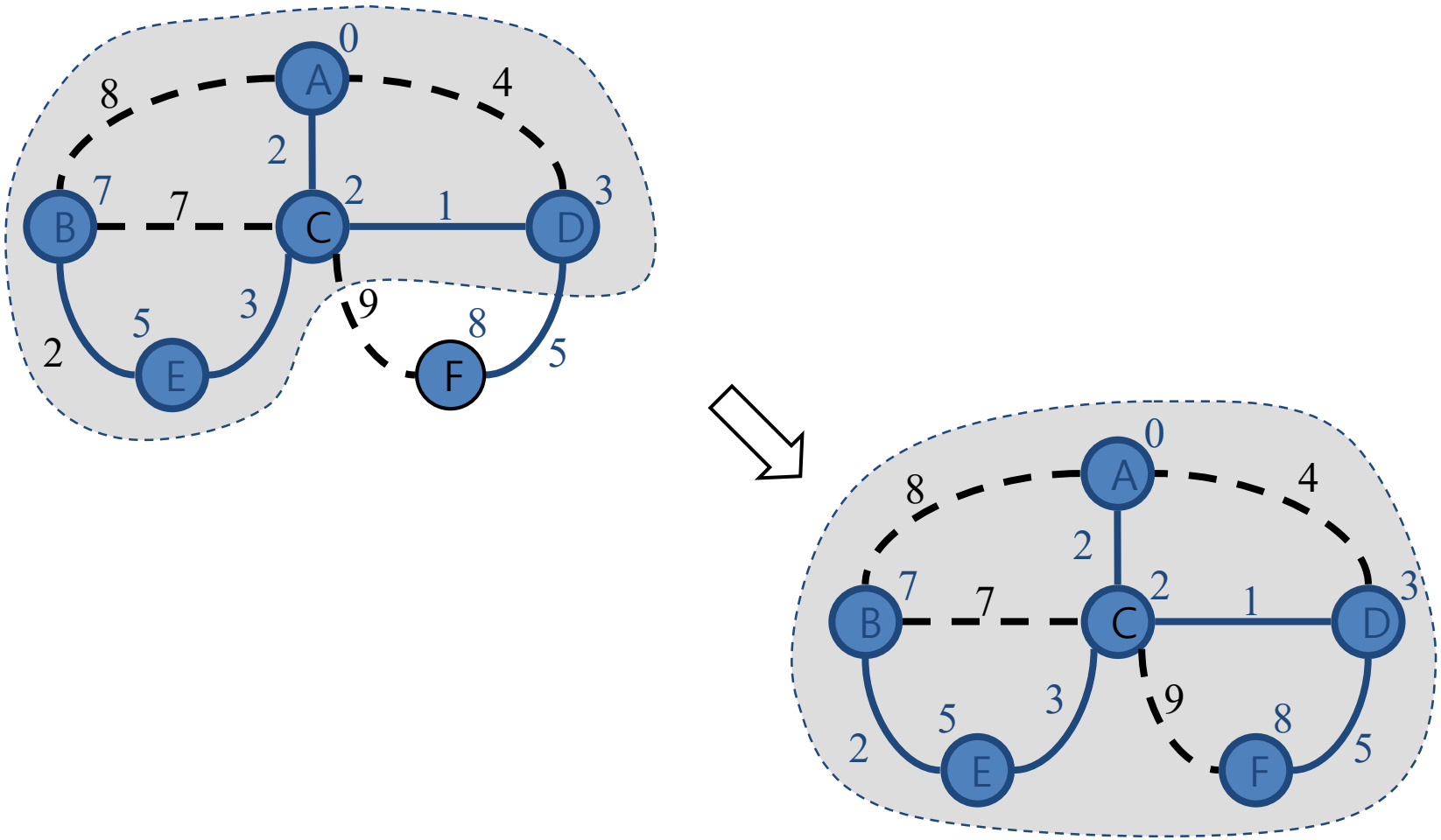
  $d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



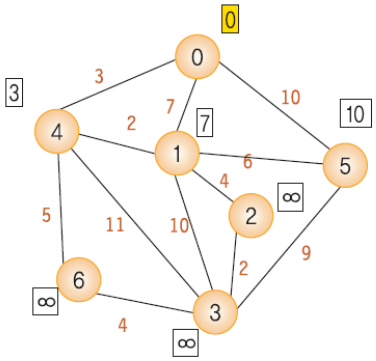
Example


A
B
C
D
E
F
0
8
4
2
8
7
2
1
4
2
∞
3
9
∞
5
A
B
C
D
E
F
0
8
4
2
8
7
2
1
3
2
5
3
9
8
5
A
B
C
D
E
F
0
8
4
2
8
7
2
1
3
2
5
3
9
11
5
A
B
C
D
E
F
0
8
4
2
7
7
2
1
3
2
5
3
9
8
5

Example (cont.)



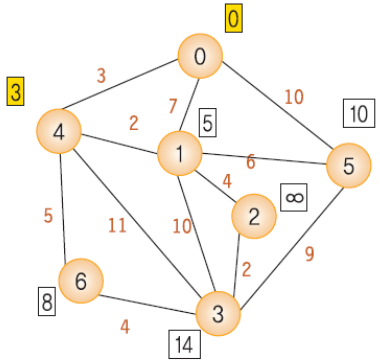
Dijkstra's Algorithm



S={0}

distance[] =

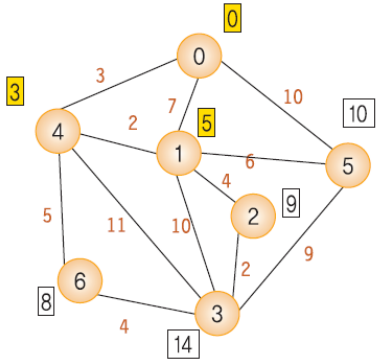
0	1	2	3	4	5	6
0	7	∞	∞	3	10	∞



S={0, 4}

distance[] =

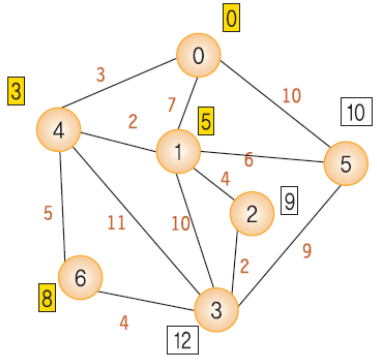
0	1	2	3	4	5	6
0	5	∞	14	3	10	8



S={0, 4, 1}

distance[] =

0	1	2	3	4	5	6
0	5	9	14	3	10	8

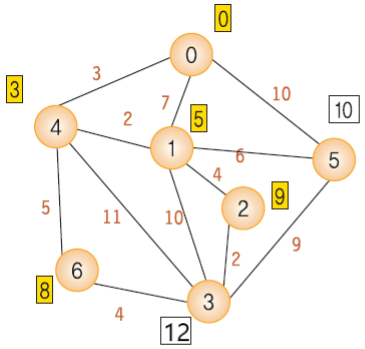


S={0, 4, 1, 6}

distance[] =

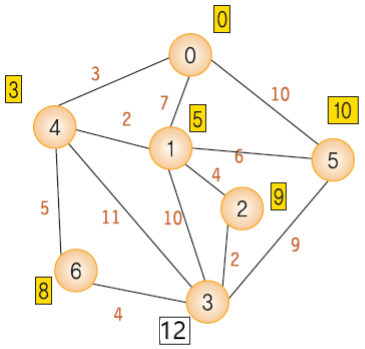
0	1	2	3	4	5	6
0	5	9	12	3	10	8

Dijkstra's Algorithm



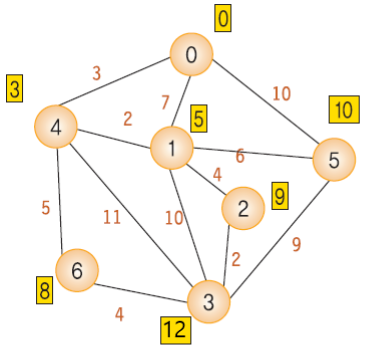
S={0, 4, 1, 6, 2}

	0	1	2	3	4	5	6
distance[]=	0	5	9	12	3	10	8



S={0, 4, 1, 6, 2, 5}

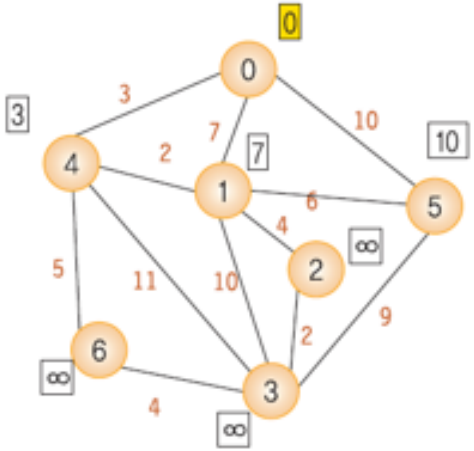
	0	1	2	3	4	5	6
distance[]=	0	5	9	12	3	10	8



S={0, 4, 1, 6, 2, 5, 3}

	0	1	2	3	4	5	6
distance[]=	0	5	9	12	3	10	8

Dijkstra's Algorithm




```
#include <stdio.h>
#include <limits.h>
#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 7          // number of vertex
#define INF 1000                // no connection
int weight[MAX_VERTICES][MAX_VERTICES]={ // adjacency matrix
{ 0, 7, INF, INF, 3, 10, INF },
{ 7, 0, 4, 10, 2, 6, INF },
{ INF, 4, 0, 2, INF, INF, INF },
{ INF, 10, 2, 0, 11, 9, 4 },
{ 3, 2, INF, 11, 0, INF, 5 },
{ 10, 6, INF, 9, INF, 0, INF },
{ INF, INF, INF, 4, 5, INF, 0 }};
int distance[MAX_VERTICES];   // shortest distance from source node
int found[MAX_VERTICES];      // visited vertex information
int path[MAX_VERTICES];       // passed vertex information
```

Dijkstra's Algorithm

```
void shortestPath(int v, int cost[][MAX_VERTICES],
                  int distance[], int n, short int found[])
{/* distance[i] represents the shortest path from vertex v
    to i, found[i] is 0 if the shortest path from i
    has not been found and a 1 if it has, cost is the
    adjacency matrix */ // v: start vertax, n : number of vertices
   int i,u,w;
   for (i = 0; i < n; i++) {    // initialize
      found[i] = FALSE;
      distance[i] = cost[v][i];
   }
   found[v] = TRUE;      // start vertax
   distance[v] = 0;
   for (i = 0; i < n-2; i++) {
      u = choose(distance,n,found);
      found[u] = TRUE;
      for (w = 0; w < n; w++) // number of vertax
         if (!found[w])
            if (distance[u] + cost[u][w] < distance[w])
               distance[w] = distance[u] + cost[u][w];
   }
}
```

Program 6.9: Single source shortest paths

how to store path?
path[w]=0; //start
path[w]=u;

Dijkstra's Algorithm

```
int choose(int distance[], int n, short int found[])
{/* find the smallest distance not yet checked */
   int i, min, minpos;
   min = INT_MAX;
   minpos = -1;
   for (i = 0; i < n; i++)
      if (distance[i] < min && !found[i]) {
         min = distance[i];
         minpos = i;
      }
   return minpos;
}
```

Program 6.10: Choosing the least cost edge

Time complexity of the Dijkstra's Algorithm

- *Repeat number of vertices for each vertex*
- $O(n^2)$

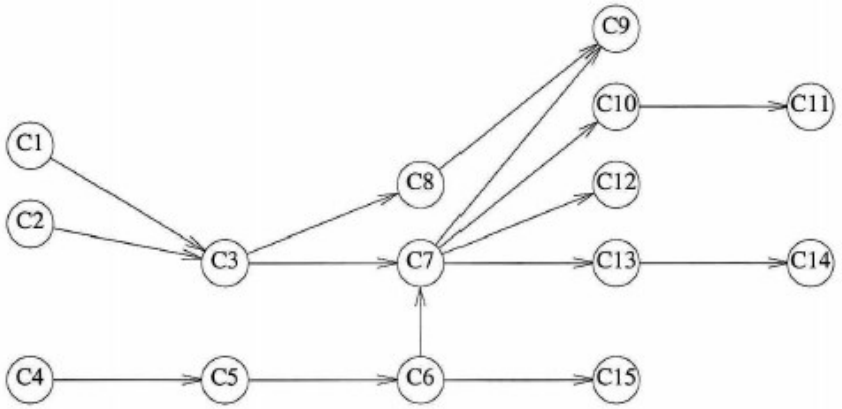
6.5 ACTIVITY NETWORKS

- **6.5.1 Activity-on-Vertex (AOV) Networks**
  - **Definition:** A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks
    - <i , j> edge in AOV network G
      - i is a *immediate predecessor* of vertex *j*
      - *j* is an *immediate successor* of *i*.

6.5 ACTIVITY NETWORKS

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

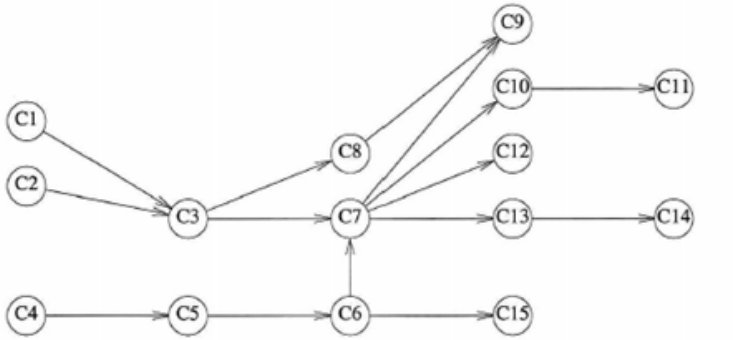
Figure 6.37: An activity-on-vertex (AOV) network

immediate predecessors of C7 :
C3 and C6
immediate successors of C7 :
C9, Cl0, Cl2, and Cl3
Cl4 is a successor of C7
Cl4 is not an immediate successor, of C3.

6.5 ACTIVITY NETWORKS

- 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

- There are several possible topological orders for the network of Figure 6.37(b).
  - Cl, C2, C4, C5, C3, C6, C8, C7, ClO, Cl3, Cl2, C14, C15, Cll, C9
  - C4, C5, C2, Cl, C6, C3, C8, C15, C7, C9, ClO, Cll, C12, C13, Cl4



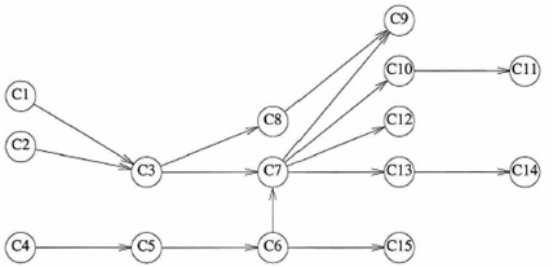
(b) AOV network representing courses as vertices and prerequisites as edges

Topological sorting

- The algorithm to sort the tasks into topological order
  - by listing a vertex in the network that has no predecessor

```
1  Input the AOV network.  Let n be the number of vertices.
2  for (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    output v;
8    delete v and all edges leading out of v;
9  }
```

Program 6.13: Design of an algorithm for topological sorting



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

Topological sorting

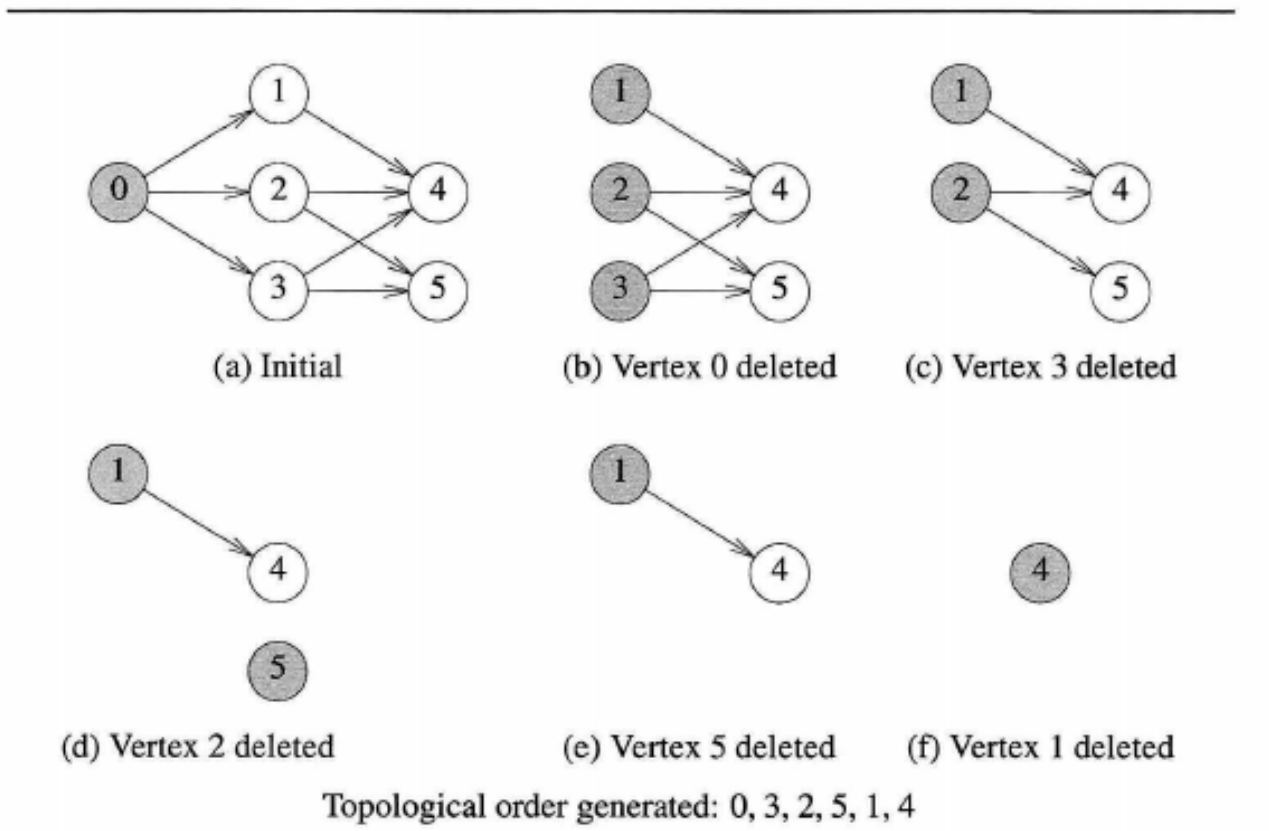
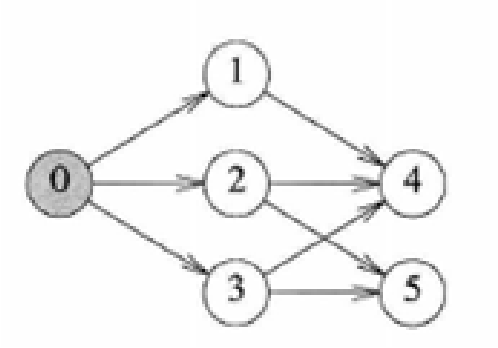


Figure 6.38: Action of Program 6.13 on an AOV network (shaded vertices represent candidates for deletion)

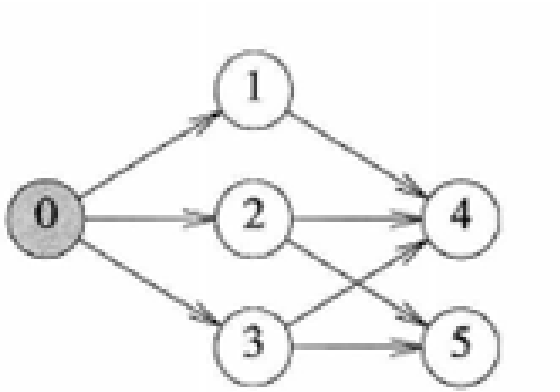
Topological sorting

- Consider the data representation for the AOV network.
  - decide whether a vertex has any predecessors
    - maintain a count of the number of immediate predecessors each vertex has.
  - delete a vertex together with all its incident edges
  - represente its adjacency lists



Topological sorting

```
typedef struct node *nodePointer;
typedef struct node
        int vertex;
        nodePointer link;
} ;
typedef struct {
        int count;
        nodePointer link;
} hdnodes;
hdnodes graph[MAX-VERTICES];
```



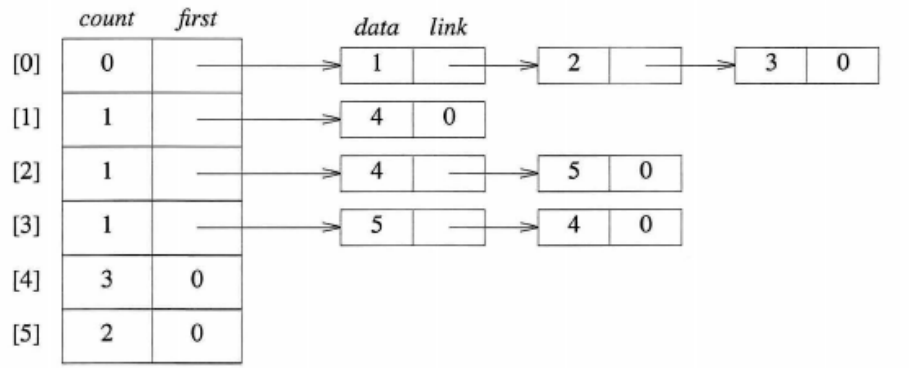


Figure 6.39: Internal representation used by topological sorting algorithm

Topological sorting

```
void topSort(hdnodes graph[], int n)
{
   int i,j,k,top;
   nodePointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         push(i)
      }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,
            "\nNetwork has a cycle. Sort terminated. \n");
         exit(EXIT_FAILURE);
      }
      else {
         j=pop()
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr→link) {
         /* decrease the count of the successor vertices
            of j */
            k = ptr→vertex;
            graph[k].count--;
            if (!graph[k].count) {
            /* add vertex k to the stack */
               push(k)
            }
         }
      }
}
```

Program 6.14: Topological sort

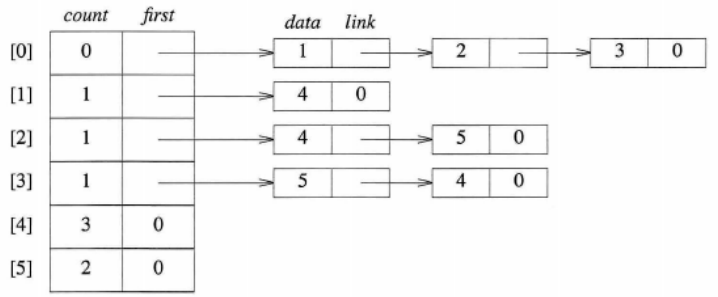
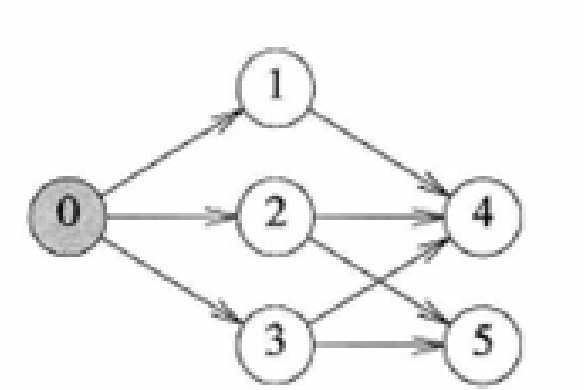


Figure 6.39: Internal representation used by topological sorting algorithm



Topological sorting

```
void topSort(hdnodes graph[], int n)
{
   int i,j,k,top;
   nodePointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         graph[i].count = top;
         top = i;
      }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,
            "\nNetwork has a cycle. Sort terminated. \n");
         exit(EXIT_FAILURE);
      }
      else {
         j = top;    /* unstack a vertex */
         top = graph[top].count;
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr→link) {
         /* decrease the count of the successor vertices
            of j */
            k = ptr→vertex;
            graph[k].count--;
            if (!graph[k].count) {
            /* add vertex k to the stack */
               graph[k].count = top;
               top = k;
            }
         }
      }
}
```

push(i)

j=pop()

push(k)

Program 6.14: Topological sort

	count	first	data	link				
[0]	0		1		→ 2		→ 3	0
[1]	1		4	0				
[2]	1		4		→ 5	0		
[3]	1		5		→ 4	0		
[4]	3	0						
[5]	2	0						

Figure 6.39: Internal representation used by topological sorting algorithm