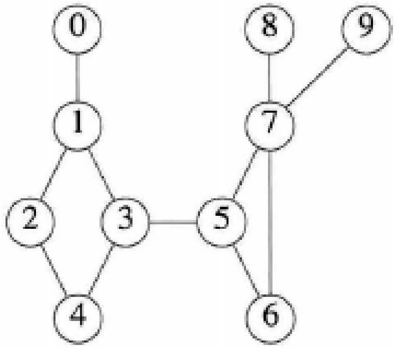
Chap 6. Graph (3)

Contents

6.2.5 Biconnected Components

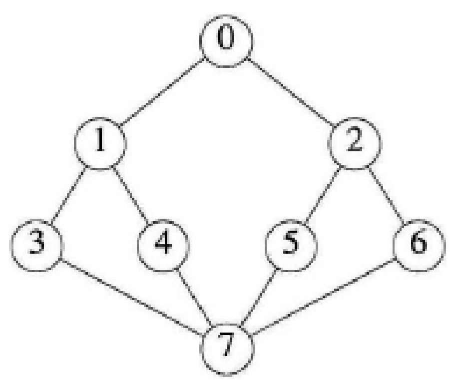
- *articulation point*
  - a vertex *v* of G such that the deletion of *v*, together with all edges incident on *v*, produces a graph, G', that has at least two connected components (*maximal* connected subgraph)
  - Figure (a) has four articulation points,
    - vertices **1,** 3, 5, and 7.



(a) Connected graph

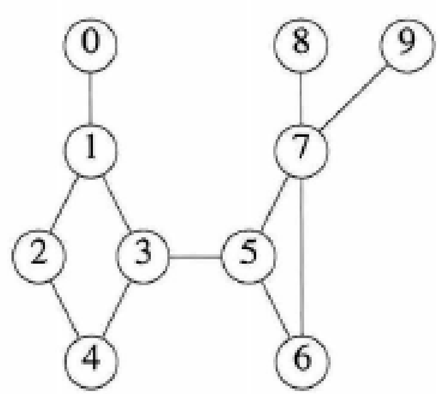
6.2.5 Biconnected Components

- *biconnected graph*
  - a connected graph that has no articulation points.
- following figure is a biconnected graph ?

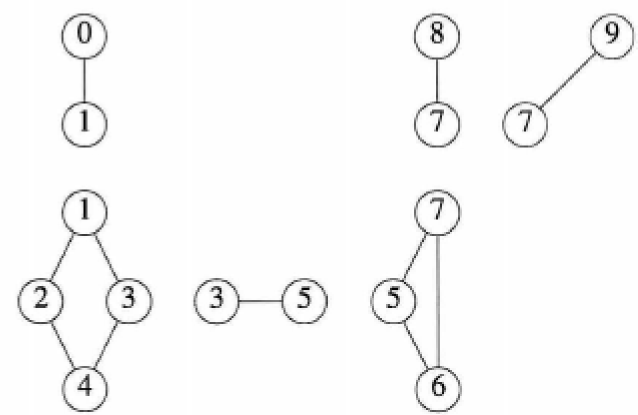


6.2.5 Biconnected Components

- *biconnected component*
  - a connected undirected graph is a *maximal biconnected subgraph*
- the graph of Figure (a) contains the six biconnected components shown in Figure (b).



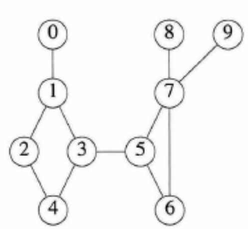
(a) Connected graph

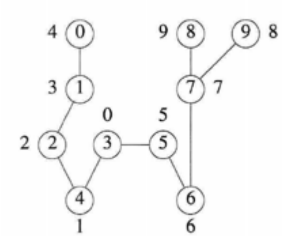


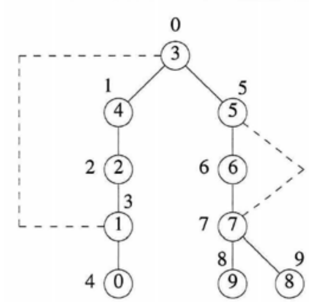
(b) Biconnected components

6.2.5 Biconnected Components

- find the biconnected components of a conected undirected graph, G,
  - by using any depth first spanning tree of G
  - *dfs* (3) applied to the graph of Figure 6.19(a)
    - redrawn the tree in Figure 6.20(b) to better reveal its tree structure.
    - The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search.
    - call this number the *depth first number,* or *dfn,* of the vertex.


(a) Connected graph


(a) depth first spanning tree



6.2.5 Biconnected Components

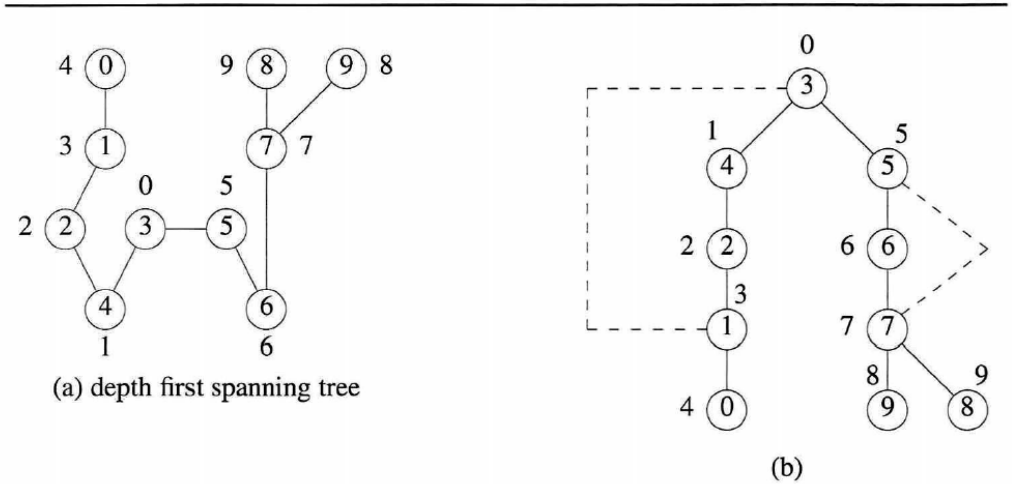
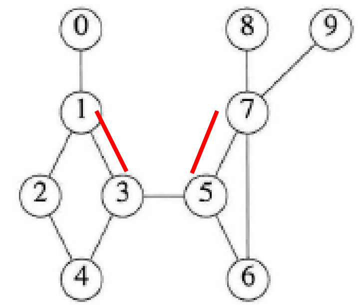


Figure 6.20: Depth first spanning tree of Figure 6.19(a)



(a) Connected graph

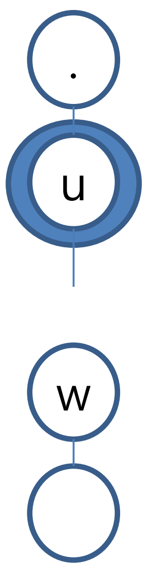
dfn (3) = 0, *dfn* (0) = 4, and *dfn* (9) = 8

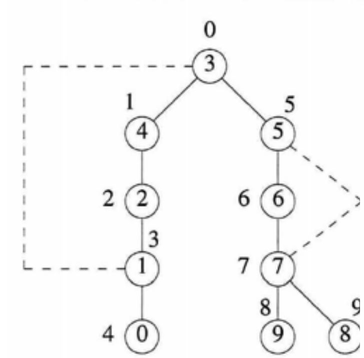
dfn (ancestor) < *dfn* (decedant)

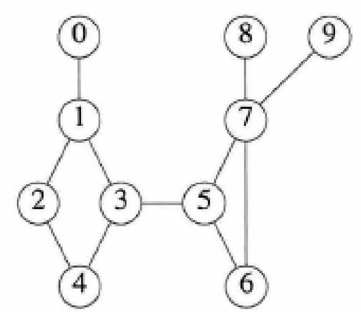
broken lines in Figure 6.20(b) represent nontree edges(*back edge)*

6.2.5 Biconnected Components

- find articulation point
  - root of a depth first spanning tree is an articulation point *iff* it has at least two children
  - any other vertex *u* is an articulation point *iff* it has at least one child *w* such that we cannot reach an ancestor of *u* using a path that consists of only *w*, descendants of w, and a single back edge.





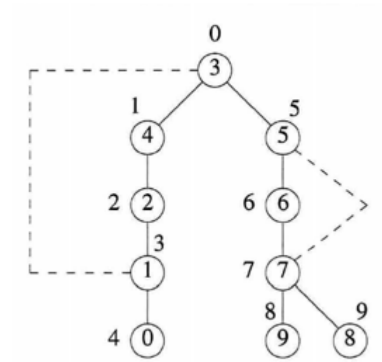


(a) Connected graph

6.2.5 Biconnected Components

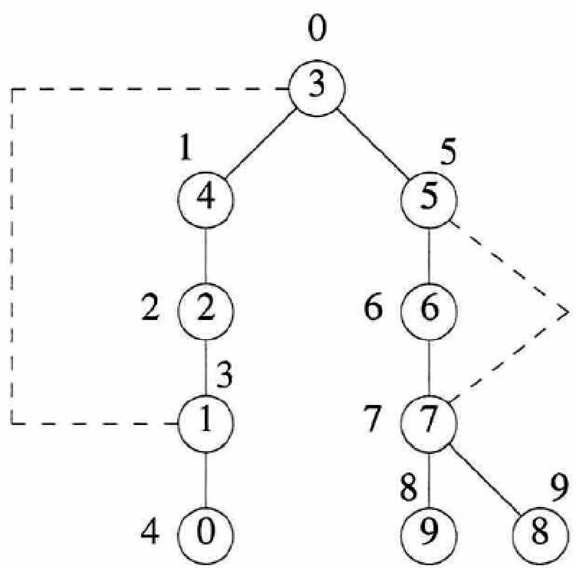
- *low* (u) is the lowest depth first number that we can reach from *u* using a path of descendants followed by at most one back edge

  low (u) = *min{dfn* (u), *min{low* (w) | *w* is a child of *u}*, min *{dfn* (w) | (u, *w)* is a back edge } }



- *u* is an articulation point
  – *low* (w) ≥ *dfn* (u).

6.2.5 Biconnected Components



(b)

an articulation point

Vertex 1의 자식

vertex 3 : root node

vertex 1 : *low* (0) = 4 ≥ *dfn* (1) = 3.

vertex 5 : *low* (6) = 5 ≥ *dfn* (5) = 5

vertex 7 : *low* (8) = 9 ≥ *dfn* (7) = 7

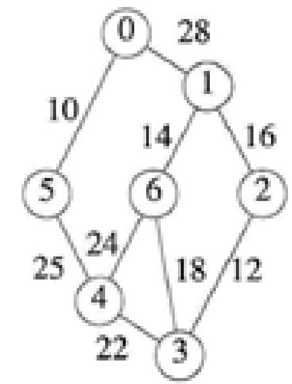
low (9) = 8 ≥ *dfn* (7) = 7

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

vertex 4 : *low* (2) = 0 < *dfn* (4) = 1

6.3 Minimum Cost Spanning Trees

- *Cost* of a spanning tree
  - sum of the costs (weights) of the edges in the spanning tree

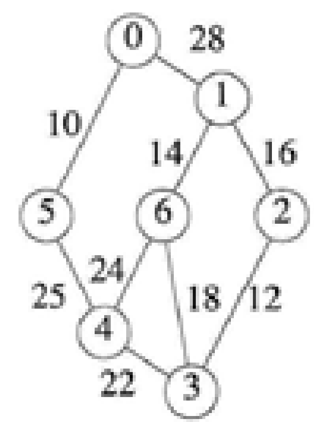


- *Minimum cost spanning tree*
  - a spanning tree of least cost

- Kruskal's, Prim's and Sollin's algorithms
  - three algorithms to build minimum cost spanning tree of a connected undirected graph
  - greedy method

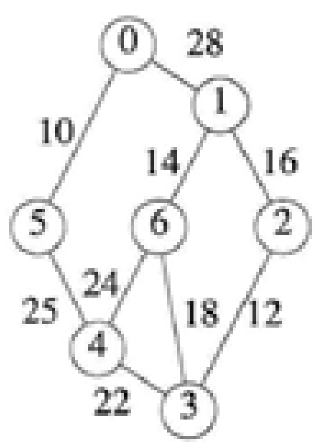
- *Greedy method*
  - at each stage, make the best decision possible at the time
    - based on either *a least cost* or *a highest profit* criterion
  - make sure the decision will result in a feasible solution
    - satisfying the constraints of the problem

- To construct minimum 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



6.3.1 Kruskal's Algorithm

- 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



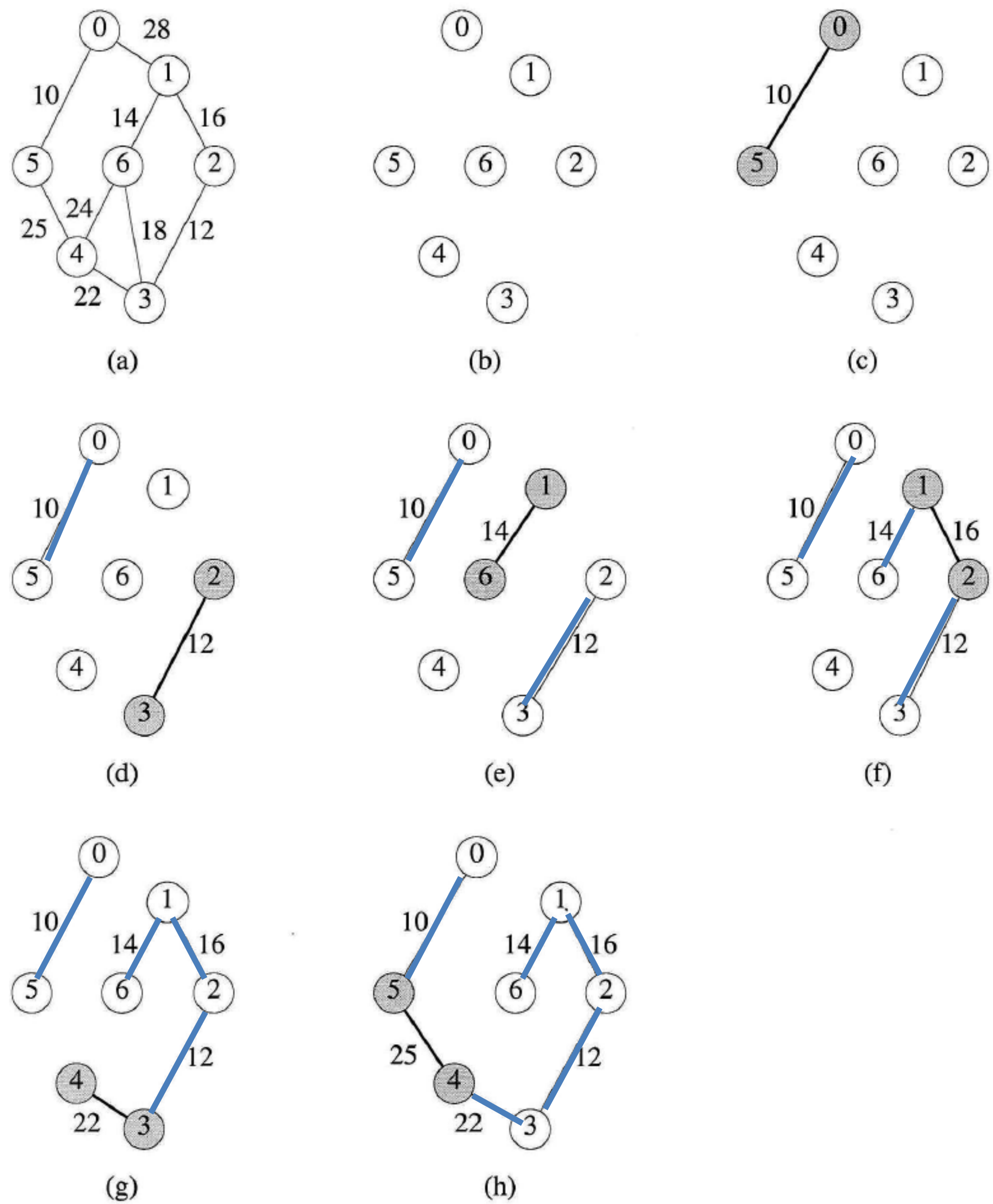


Figure 6.22: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
----	---	initial	Figure 6.22(b)
(0,5)	10	added to tree	Figure 6.22(c)
(2,3)	12	added	Figure 6.22(d)
(1,6)	14	added	Figure 6.22(e)
(1,2)	16	added	Figure 6.22(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.22(h)
(0,1)	28	not considered	

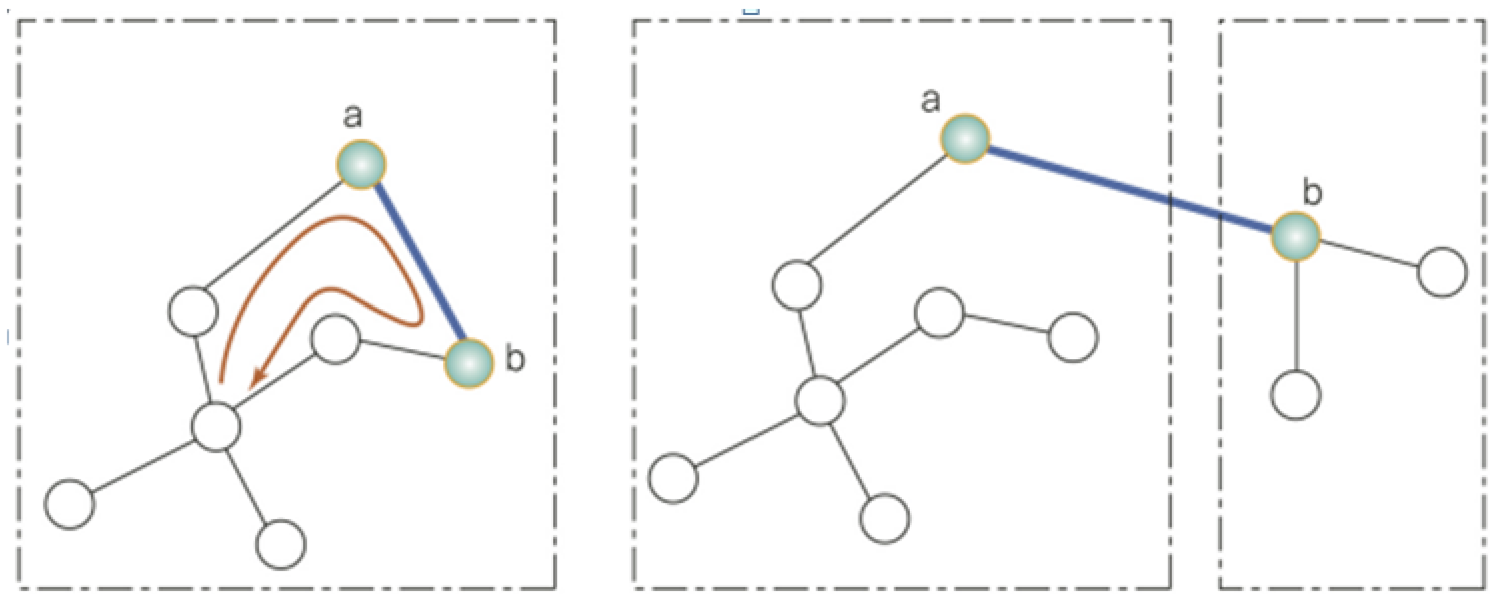
Summary of Kruskal's algorithm applied to Figure 6.22(a)

```
T = {};
while (T contains less than n-1 edges && E is not empty) {
   choose a least cost edge (v,w) from E;
   delete (v,w) from E;
   if ((v,w) does not create a cycle in T)
      add (v,w) to T;
   else
      discard (v,w);
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.7: Kruskal's algorithm

Union And Find Operations

- check cycle using Union And Find Operations
- find(a), find(b)
  - if the result is the same set, this operation makes a cycle.



S1 S1 S2

Union And Find Operations

- union operation
  - obtain the union of S1 and S2
  - simply make one of the trees a subtree of the other

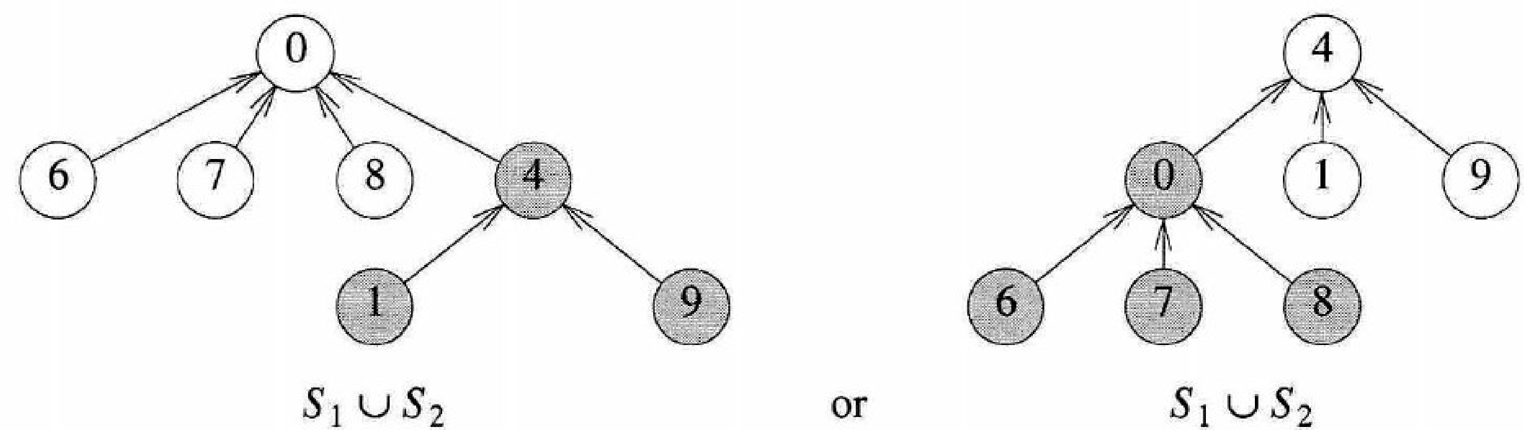


Figure 5.38: Possible representation of $S_1 \cup S_2$

Union And Find Operations

- To implement the set union operation
  - we simply set the parent field of one of the roots to the other root.
- If *i* is an element in a tree with *root j,* and *j* has a pointer to entry *k* in the set name table, then the set name is just *name[k]*.

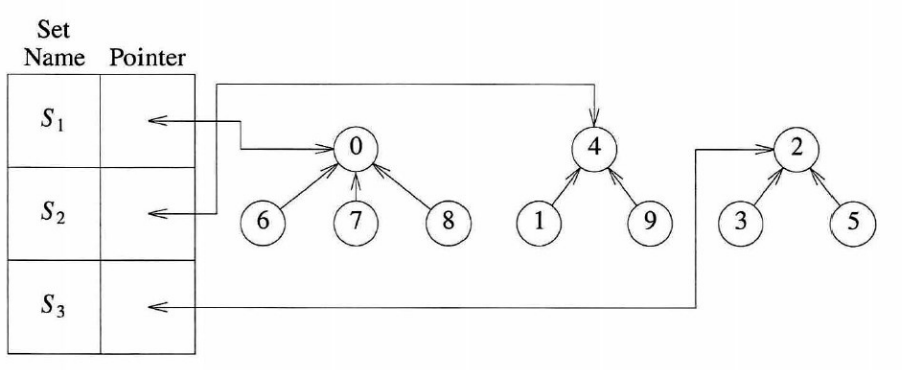


Figure 5.39: Data representation of S_1, S_2, and S_3

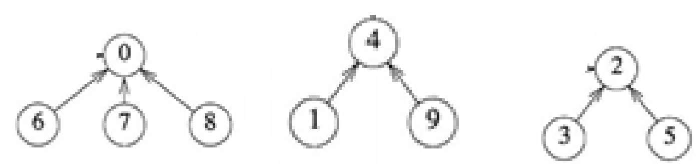
Union And Find Operations

- the nodes in the trees are numbered 0 through n - 1 we can use the node's number as an index.
- representation of the sets, S 1 , S 2 , and S 3
- root nodes have a parent of -1.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	−1	4	−1	2	−1	2	0	0	0	4

Figure 5.40: Array representation of S_1, S_2, and S_3

to find 5, we start at 5, and then move to 5's parent, 2.



S 1 , S 2 , and S 3

Union And Find Operations

```
int simpleFind(int i)
{
   for(; parent[i] >= 0; i = parent[i])
      ;
   return i;
}
void simpleUnion(int i, int j)  // i, j : root
{
   parent[i] = j;
}
```

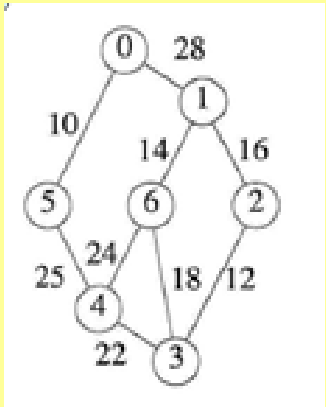
Program 5.19: Initial attempt at union-find functions

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	−1	4	−1	2	−1	2	0	0	0	4

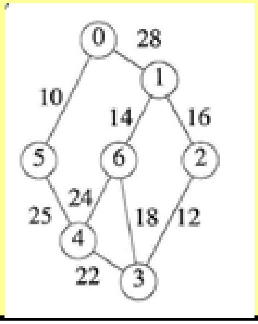
Figure 5.40: Array representation of S_1, S_2, and S_3

Kruskal's Algorithm

```
#include <stdio.h>
#define MAX_VERTICES 100
#define INF 1000
...
// heap element
typedef struct {
		int key;	//  weighted value
		int u;		// vertex 1
		int v;		// vertex 2
} element;
#define MAX_ELEMENT 100
typedef struct {
		element heap[MAX_ELEMENT];
		int heap_size;
} HeapType;
// ...
// insert edge in heap
void insert_heap_edge(HeapType *h, int u, int v, int weight) {
		element e;
		e.u = u;
		e.v = v;
		e.key = weight;
		insert_min_heap(h, e);
}
// make heap
void insert_all_edges(HeapType *h) {
		insert_heap_edge(h, 0, 1, 28);
		insert_heap_edge(h, 1, 2, 16);
		insert_heap_edge(h, 2, 3, 12);
		insert_heap_edge(h, 3, 4, 22);
		insert_heap_edge(h, 4, 5, 25);
		insert_heap_edge(h, 5, 0, 10);
		insert_heap_edge(h, 6, 1, 14);
		insert_heap_edge(h, 6, 3, 18);
		insert_heap_edge(h, 6, 4, 24);
}
```



```
void kruskal(int n)
{
int edge_accepted=0;        // number of selected edge
HeapType h;                 // min heap
int uset, vset;             // root of vertex
element e;                  // heap elemnet
init(&h);                   // init heap
insert_all_edges(&h);       // make heap
set_init(n);                // initialize union-find funtion.
while( edge_accepted < (n-1) ) // number of  edge < (n-1)
        {
        e = delete_min_heap(&h);            // get a vertex from heap
        uset = find(e.u);                   // find() root
        vset = find(e.v);                   // find() root
        if( uset != vset ){                 // different root
                printf("(%d,%d) %d ₩n",e.u, e.v, e.key);
                edge_accepted++;
                union(uset, vset);          // union two vertices.
                }
        }
}
//
main()
{
kruskal(7);  //number of node
}
```

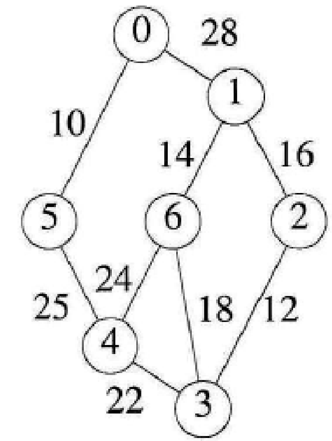


Time complexity of the Kruskal

- The Kruskal algorithm depends mostly on the time to align the edges.
- Operations such as cycle testing are performed very quickly compared to alignment.
- The time complexity of the Kruskal algorithm is $O(e*log(e))$ if we arrange e edges of the network with efficient algorithms such as quick sort.

6.3.2 Prim's Algorithm

- Algorithm
  - Build a minimum cost spanning tree *T* by adding edges to *T* one at a time.
  - At each stage, add a least cost edge to *T* such that the set of selected edges is also a tree.
  - Repeat the edge addition step until *T* contains n-1 edges.




```
T = {};
TV= {0};  /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
      let (u, v) be a least cost edge such that u ∈ TV and
                                                  v ∉ TV;
      if (there is no such edge)
            break;
      add v to TV;
      add (u, v) to T;
}
if (T contains fewer than n-1 edges)
     printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm

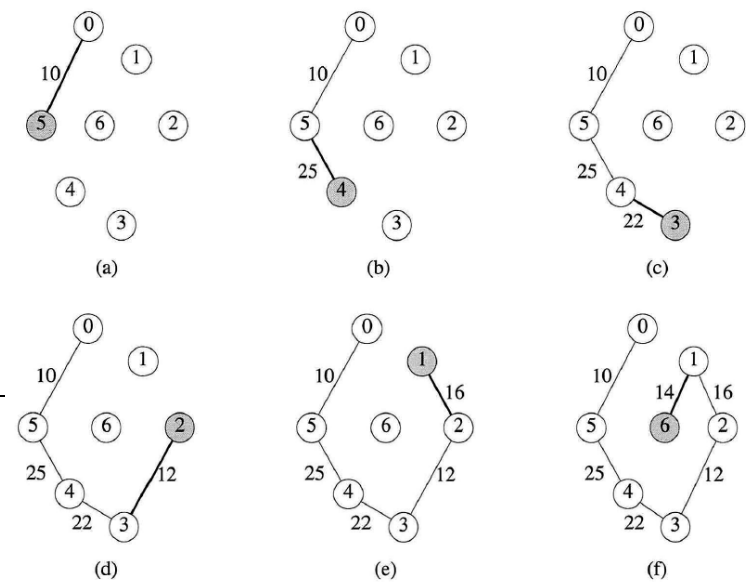


Figure 6.24: Stages in Prim's algorithm

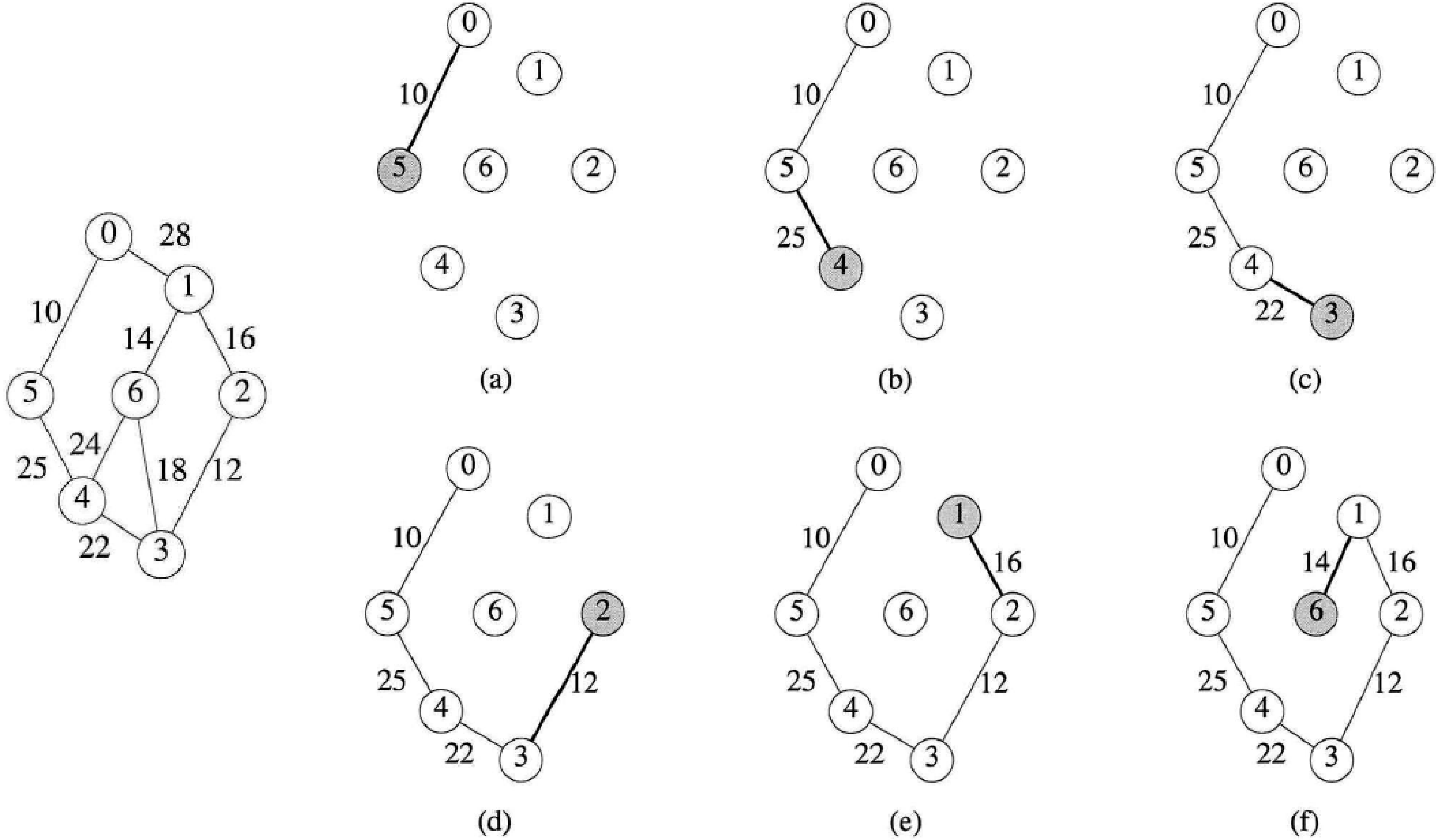
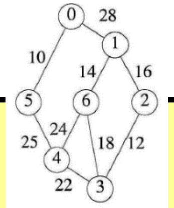


Figure 6.24: Stages in Prim's algorithm

Prim Algorithm



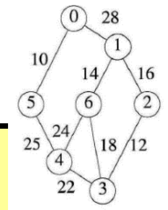

```c
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 7
#define INF 1000L
int weight[MAX_VERTICES][MAX_VERTICES]={
{ 0, 28, INF, INF, INF, 10, INF },
{ 28, 0, 16, INF, INF, INF, 14 },
{ INF, 16, 0, 12, INF, INF, INF },
{ INF, INF, 12, 0, 22, INF, 18 },
{ INF, INF, INF, 22, 0, 25, 24 },
{ 10, INF, INF, INF, 25, 0, INF },
{ INF, 14, INF, 18, 24, INF, 0 }};
int selected[MAX_VERTICES];
int dist[MAX_VERTICES];
int get_min_vertex(int n)
{
int v,i;
for (i = 0; i <n; i++)
            if (!selected[i]) {v = i;  break; }
for (i = 0; i < n; i++)
            if ( !selected[i] && (dist[i] < dist[v])) v = i;
return (v);
}
```

```
{ 0,    28, INF,  INF, INF,  10, INF },
{ 28,   0,   16,  INF, INF, INF,  14 },
{ INF, 16,   0,    12, INF, INF, INF },
{ INF, INF, 12,    0,   22, INF, 18 },
{ INF, INF, INF,  22,    0,  25,  24 },
{ 10,  INF, INF, INF,   25,    0, INF },
{ INF,  14, INF,  18,   24, INF,    0 }};
```

dist[u]

0	0	28	INF	INF	INF	10	INF
5	0	28	INF	INF	25	10	INF

Prim Algorithm(cont.)



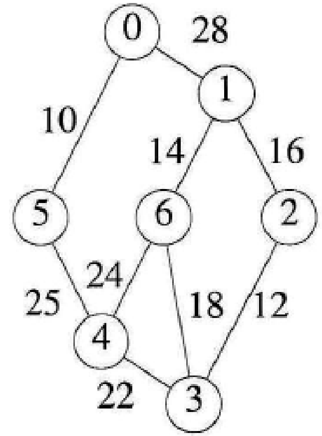

```
void prim(int s, int n)
{int i, u, v;
for(u=0;u<n;u++) // initialize dist and selected
          {dist[u]=INF;
           selected[u] = FALSE;
          }
dist[s]=0;
for(i=0;i<n;i++){
          u = get_min_vertex(n);
          selected[u]=TRUE;
          if( dist[u] == INF ) return;
          printf("%d ", u);
          for( v=0; v<n; v++)
                    if( weight[u][v]!= INF)
                              if( !selected[v] && weight[u][v]< dist[v] )
                                        dist[v] = weight[u][v];
          }
}

main()
{prim(0, MAX_VERTICES);
}
```

```
{ 0,   28, INF,  INF, INF,  10, INF },
{ 28,   0,  16,  INF, INF, INF,  14 },
{ INF, 16,   0,   12, INF, INF, INF },
{ INF, INF, 12,    0,  22, INF, 18 },
{ INF, INF, INF,  22,   0,  25,  24 },
{ 10,  INF, INF, INF,  25,   0, INF },
{ INF,  14, INF,  18,  24, INF,   0 }};
```

dist[u]

0	0	28	INF	INF	INF	10	INF
5	0	28	INF	INF	**25**	10	INF

```
{ 0,    28, INF,  INF, INF,  10, INF },
{ 28,   0,   16,  INF, INF, INF,  14 },
{ INF, 16,   0,    12, INF, INF, INF },
{ INF, INF, 12,    0,   22, INF, 18 },
{ INF, INF, INF,  22,   0,   25,  24 },
{ 10,  INF, INF, INF,  25,    0, INF },
{ INF,  14, INF,  18,  24, INF,    0 }};
```



dist[u]

0	0	28	INF	INF	INF	**10**	INF
5	0	28	INF	INF	**25**	10	INF
4	0	28	INF	**22**	25	10	24
3	0	28	**12**	22	25	10	18
2	0	**16**	12	22	25	10	18
1	0	16	12	22	25	10	**14**
6	0	16	12	22	25	10	14

selected[u]

1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	1	0
1	0	0	1	1	1	0
1	0	1	1	1	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	1

Time complexity of the Prim

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