# Chapter 12

GRAPHS

1. Mathematical Background

2. Computer Representation

3. Graph Traversal

4. Topological Sorting

5. A Greedy Algorithm: Shortest Paths

6. Minimal Spanning Trees
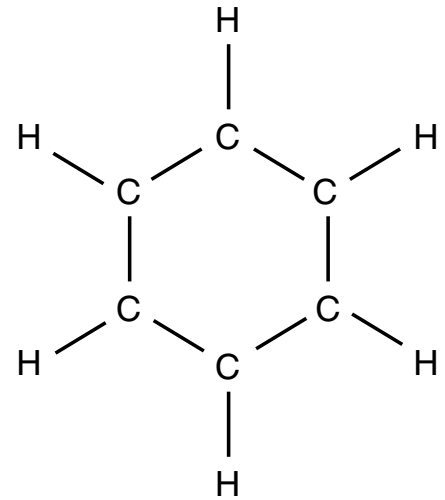
7. Graphs as Data Structures

Graphs: Definitions

1. A ***graph*** G consists of a set V, whose members are called the ***vertices*** of G, together with a set E of pairs of distinct vertices from V.

2. The pairs in E are called the ***edges*** of G.

3. If $e = (v, w)$ is an edge with vertices v and w, then v and w are said to ***lie on*** e, and e is said to be ***incident*** with v and w.

4. If the pairs are unordered, G is called an ***undirected graph***.

5. If the pairs are ordered, G is called a ***directed graph***. The term *directed graph* is often shortened to ***digraph***, and the unqualified term *graph* usually means *undirected graph*.

6. Two vertices in an undirected graph are called ***adjacent*** if there is an edge from the first to the second.

7. A ***path*** is a sequence of distinct vertices, each adjacent to the next.

8. A ***cycle*** is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.

9. A graph is called ***connected*** if there is a path from any vertex to any other vertex.

10. A ***free tree*** is defined as a connected undirected graph with no cycles.

11. In a directed graph a path or a cycle means always moving in the direction indicated by the arrows. Such a path (cycle) is called a ***directed*** path (cycle).

12. A directed graph is called ***strongly connected*** if there is a directed path from any vertex to any other vertex. If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph ***weakly connected***.

13. The ***valence*** of a vertex is the number of edges on which it lies, hence also the number of vertices adjacent to it.
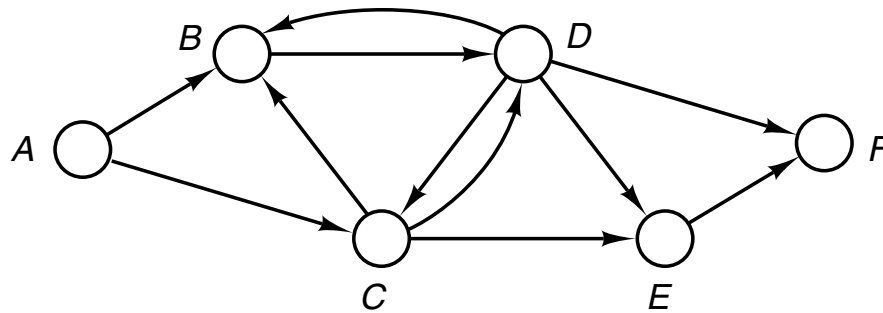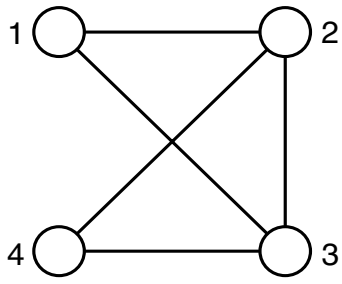
Examples of Graphs

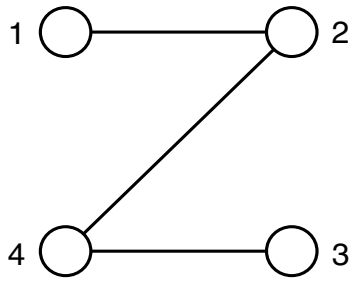Honolulu
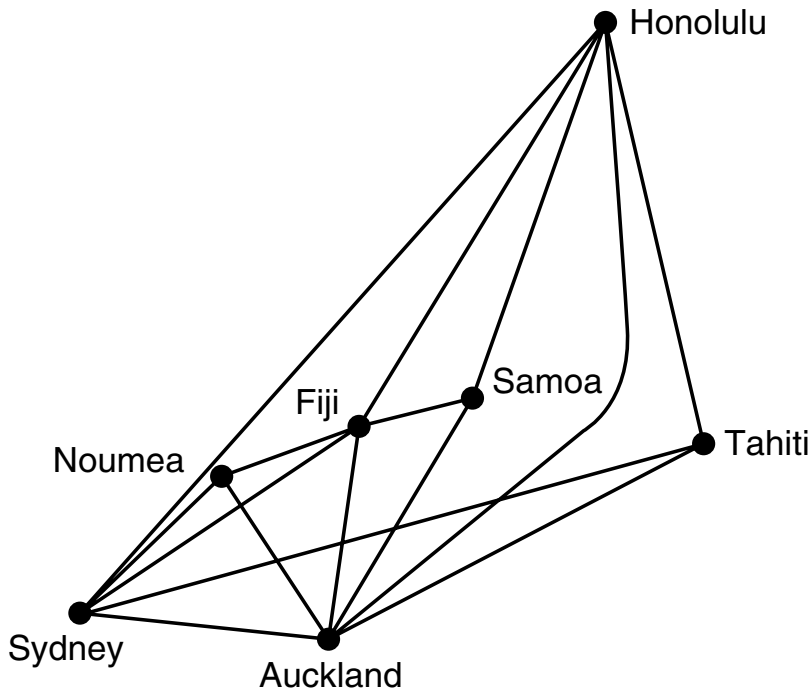
Selected South Pacific air routes

Benzene molecule

Message transmission in a network
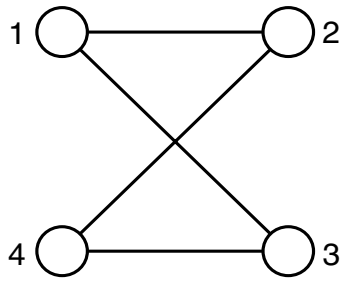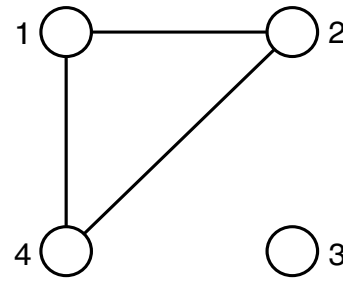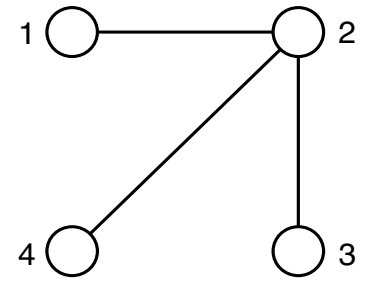
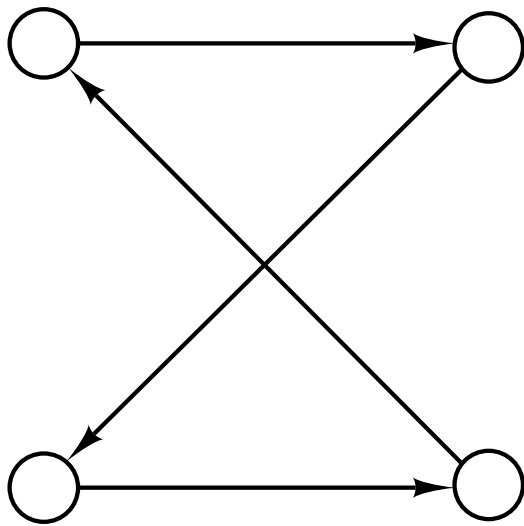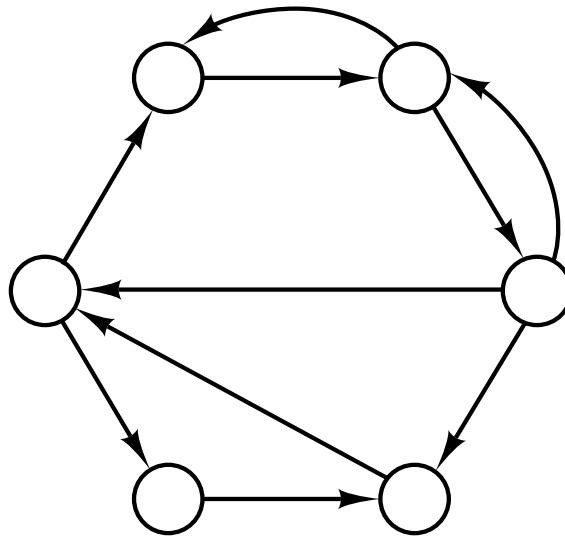| 1 ○――○ 2 | 1 ○――○ 2 | 1 ○――○ 2 | 1 ○――○ 2 | 1 ○――○ 2 |
| Connected | Path | Cycle | Disconnected | Tree |
| (a) | (b) | (c) | (d) | (e) |

Directed cycle

(a)

Strongly connected

(b)
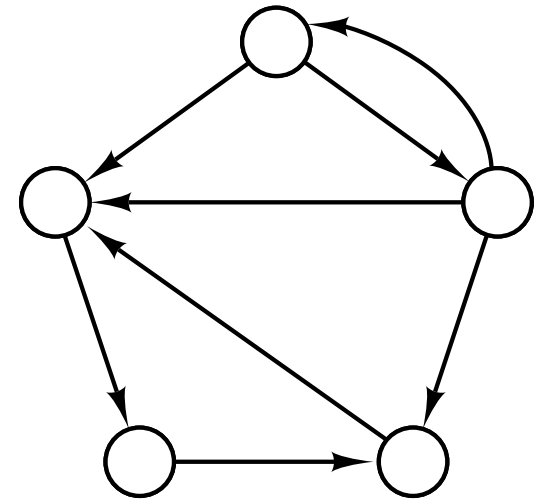
Weakly connected

(c)

Set Implementations of Digraphs

> DEFINITION A ***digraph*** G consists of a set V, called the ***vertices*** of G, and, for all $v \in V$, a subset A_v of V, called the set of vertices ***adjacent*** to v.

Set as a bit string:

```
template <int max_set>
struct Set {
  bool is_element[max_set];
};
```

Digraph as a bit-string set:

```
template <int max_size>
class Digraph {
  int count;               //   number of vertices, at most max_size
  Set<max_size> neighbors[max_size];
};
```

Digraph as an adjacency table:

```
template <int max_size>
class Digraph {
  int count;               //   number of vertices, at most max_size
  bool adjacency[max_size][max_size];
};
```

List Implementation of Digraphs
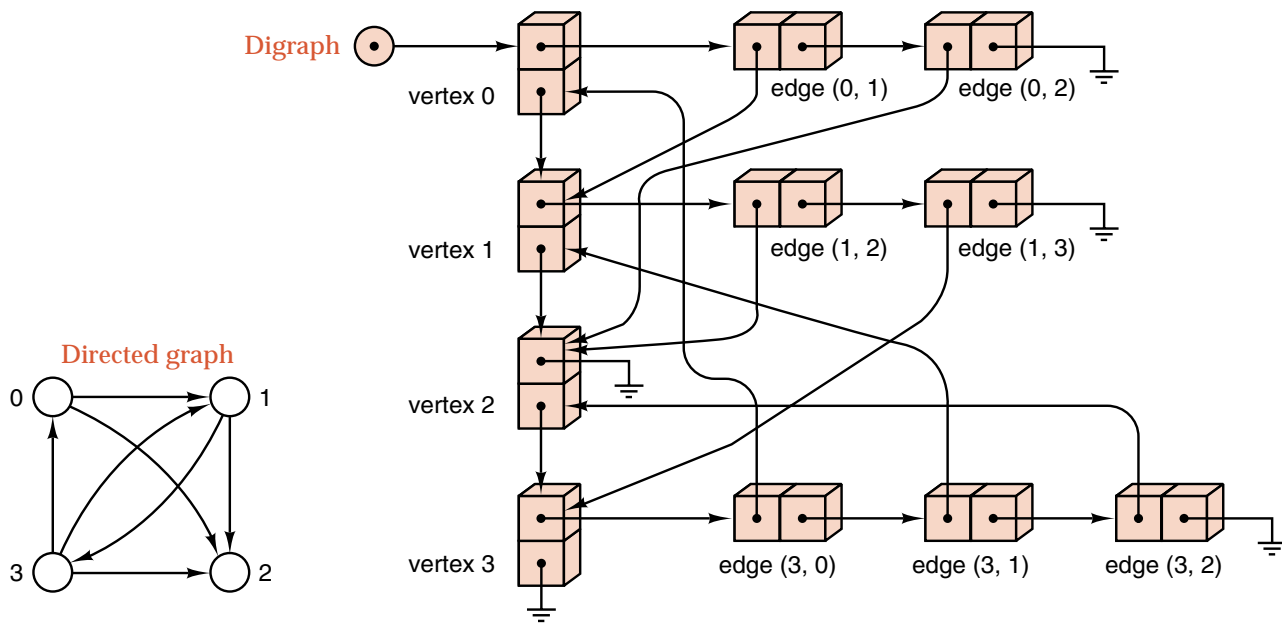
```
typedef int Vertex;
template <int max_size>
class Digraph {
   int count;                          //   number of vertices, at most max_size
   List<Vertex> neighbors[max_size];
};



class Edge;                            //   forward declaration
class Vertex {
   Edge *first_edge;                   //   start of the adjacency list
   Vertex *next_vertex;                //   next vertex on the linked list
};
class Edge {
   Vertex *end_point;                  //   vertex to which the edge points
   Edge *next_edge;                    //   next edge on the adjacency list
};
class Digraph {
   Vertex *first_vertex;               //   header for the list of vertices
};
```

Directed graph

Adjacency sets
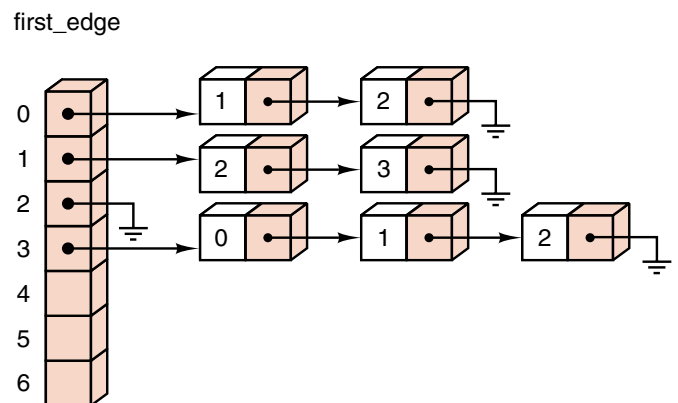
vertex	Set
0	{ 1, 2 }
1	{ 2, 3 }
2	Ø
3	{ 0, 1, 2 }

Adjacency table

	0	1	2	3
0	F	T	T	F
1	F	F	T	T
2	F	F	F	F
3	T	T	T	F

Digraph

vertex 0   edge (0, 1)   edge (0, 2)

vertex 1   edge (1, 2)   edge (1, 3)

vertex 2

vertex 3   edge (3, 0)   edge (3, 1)   edge (3, 2)

Directed graph

(a) Linked lists
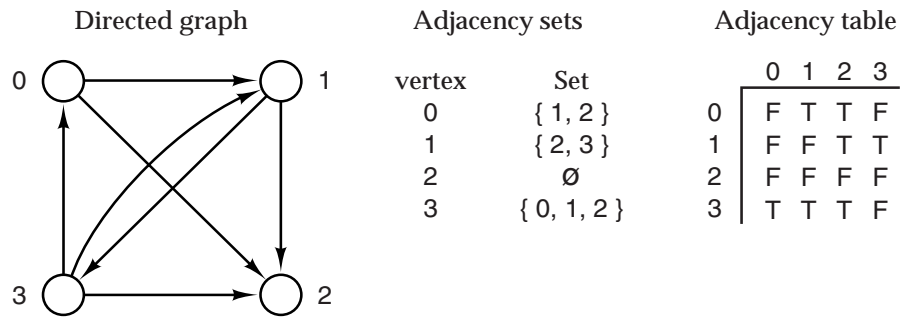
count = 4

vertex	adjacency list						
0	1	2	–	–	–	–	–
1	2	3	–	–	–	–	–
2	–	–	–	–	–	–	–
3	0	1	2	–	–	–	–
4	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–

(b) Contiguous lists

count = 4

first_edge

0 → 1 → 2

1 → 2 → 3

2

3 → 0 → 1 → 2

4

5

6

(c) Mixed

Graph Traversal

■ **Depth-first traversal** of a graph is roughly analogous to pre-order traversal of an ordered tree. Suppose that the traversal has just visited a vertex v, and let $w_1, w_2, \ldots, w_k$ be the vertices adjacent to v. Then we shall next visit w_1 and keep $w_2, \ldots, w_k$ waiting. After visiting w_1, we traverse all the vertices to which it is adjacent before returning to traverse $w_2, \ldots, w_k$.

■ **Breadth-first traversal** of a graph is roughly analogous to level-by-level traversal of an ordered tree. If the traversal has just visited a vertex v, then it next visits *all* the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to v have been visited.

Depth-first traversal

Breadth-first traversal
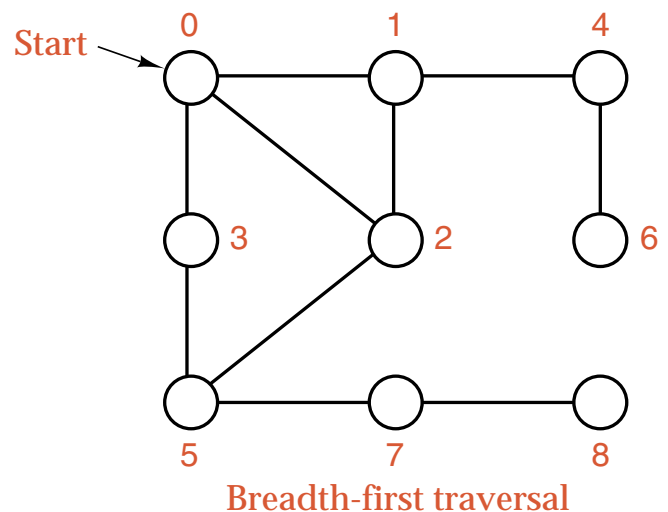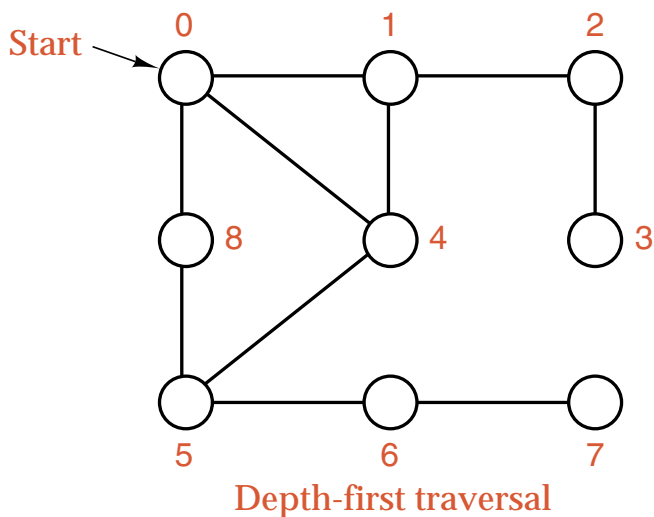
Depth-First Algorithm

```
template <int max_size>
void Digraph<max_size>::depth_first(void (*visit)(Vertex &)) const
/* Post:  The function *visit has been performed at each vertex of the Digraph in
          depth-first order.
   Uses:  Method traverse to produce the recursive depth-first order.  */
{
   bool visited[max_size];
   Vertex v;
   for (all v in G) visited[v] = false;
   for (all v in G) if (!visited[v])
      traverse(v, visited, visit);
}
```

The recursion is performed in an auxiliary function traverse.

```
template <int max_size>
void Digraph<max_size>::traverse(Vertex &v, bool visited[],
                                          void (*visit)(Vertex &)) const
/* Pre:   v is a vertex of the Digraph.
   Post:  The depth-first traversal, using function *visit, has been completed for v and
          for all vertices that can be reached from v.
   Uses: traverse recursively.  */
{
   Vertex w;
   visited[v] = true;
   (*visit)(v);
   for (all w adjacent to v)
     if (!visited[w])
        traverse(w, visited, visit);
}
```

Breadth-First Algorithm

```
template <int max_size>
void Digraph<max_size> :: breadth_first(void (*visit)(Vertex &)) const
/* Post:  The function *visit has been performed at each vertex of the Digraph in
          breadth-first order.
    Uses: Methods of class Queue.  */
{
  Queue q;
  bool visited[max_size];
  Vertex v, w, x;
  for (all v in G) visited[v] = false;
  for (all v in G)
    if (!visited[v]) {
      q.append(v);
      while (!q.empty()){
        q.retrieve(w);
        if (!visited[w]) {
          visited[w] = true;
          (*visit)(w);
          for (all x adjacent to w)
            q.append(x);
        }
        q.serve();
      }
    }
}
```
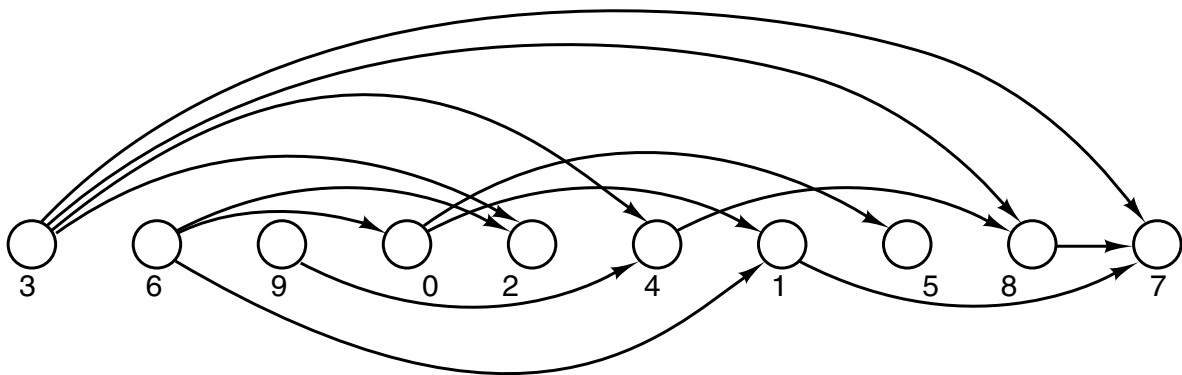
Let G be a directed graph with no cycles. A ***topological order*** for G is a sequential listing of all the vertices in G such that, for all vertices $v, w \in G$, if there is an edge from v to w, then v precedes w in the sequential listing.
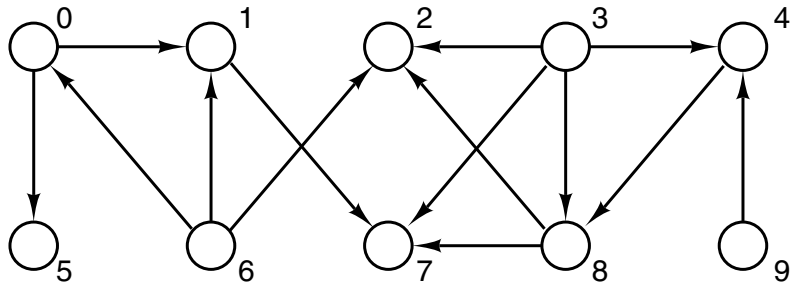
Directed graph with no directed cycles
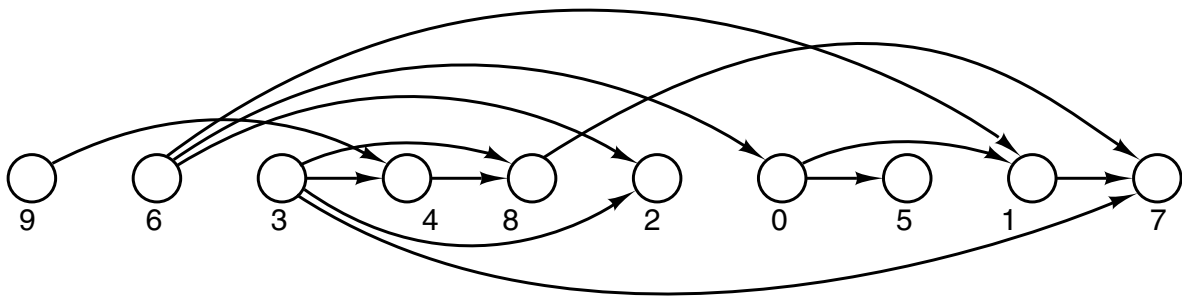
Depth-first ordering

Breadth-first ordering

Digraph Class, Topological Sort

typedef int Vertex;

template <**int** graph_size>
class Digraph {
public:
  Digraph( );
  void read( );
  void write( );

//   methods to do a topological sort
  void depth_sort(List<Vertex> &topological_order);
  void breadth_sort(List<Vertex> &topological_order);

private:
  int count;
  List <Vertex> neighbors[graph_size];
  void recursive_depth_sort(Vertex v, **bool** visited[ ],
                       List<Vertex> &topological_order);
};

Depth-First Algorithm

```
template <int graph_size> void Digraph<graph_size>::
    depth_sort(List<Vertex> &topological_order)
```
/* **Post:** *The vertices of the* Digraph *are placed into* List topological_order *with a depth-first traversal of those vertices that do not belong to a cycle.*

  Uses: *Methods of* **class** List*, and function* recursive_depth_sort *to perform depth-first traversal.* */
```
{ bool visited[graph_size];
  Vertex v;
  for (v = 0;  v < count;  v++) visited[v] = false;
  topological_order.clear();
  for (v = 0;  v < count;  v++)
    if (!visited[v])           //    Add v and its successors into topological order.
      recursive_depth_sort(v, visited, topological_order);
}
```

```
template <int graph_size> void Digraph<graph_size>::
recursive_depth_sort(Vertex v, bool *visited, List<Vertex> &topological_order)
```
/* **Pre:** *Vertex* v *of the* Digraph *does not belong to the partially completed* List topo-logical_order.

  Post: *All the successors of* v *and finally* v *itself are added to* topological_order *with a depth-first search.*

  Uses: *Methods of* **class** List *and the function* recursive_depth_sort. */
```
{ visited[v] = true;
  int degree = neighbors[v].size();
  for (int i = 0;  i < degree;  i++) {
    Vertex w;                          //    A (neighboring) successor of v
    neighbors[v].retrieve(i, w);
    if (!visited[w])            //    Order the successors of w.
      recursive_depth_sort(w, visited, topological_order);
  }
  topological_order.insert(0, v);   //    Put v into topological_order.
}
```

Breadth-First Algorithm

template <**int** graph_size> **void** Digraph<graph_size>::
   breadth_sort(List<Vertex> &topological_order)
/* **Post:** *The vertices of the* Digraph *are arranged into the* List topological_order
       which is found with a breadth-first traversal of those vertices that do not belong
       to a cycle.
  Uses: *Methods of classes* Queue *and* List. */

```
{ topological_order.clear( );
  Vertex v, w;
  int predecessor_count[graph_size];
  for (v = 0;  v < count;  v++) predecessor_count[v] = 0;
  for (v = 0;  v < count;  v++)
    for (int i = 0;  i < neighbors[v].size( );  i++) {
      neighbors[v].retrieve(i, w);   //    Loop over all edges v — w.
      predecessor_count[w]++;
    }
  Queue ready_to_process;
  for (v = 0;  v < count;  v++)
    if (predecessor_count[v] ==  0)
      ready_to_process.append(v);
  while (!ready_to_process.empty( )) {
    ready_to_process.retrieve(v);
    topological_order.insert(topological_order.size( ), v);
    for (int j = 0;  j < neighbors[v].size( );  j++) {
      neighbors[v].retrieve(j, w);   //    Traverse successors of v.
      predecessor_count[w]−−;
      if (predecessor_count[w] ==  0)
        ready_to_process.append(w);
    }
    ready_to_process.serve( );
  }
}
```

A Greedy Algorithm: Shortest Paths

The problem of shortest paths:

Given a directed graph in which each edge has a nonnegative **weight** or cost, find a path of least total weight from a given vertex, called the **source**, to every other vertex in the graph.
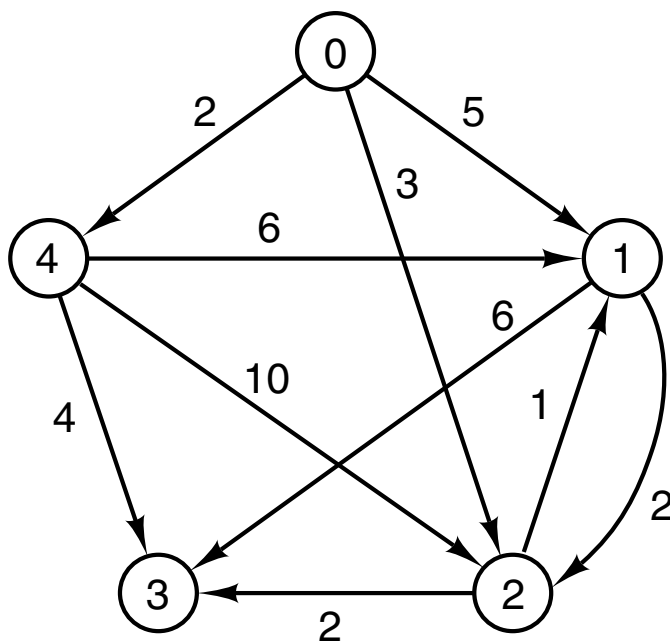

Method:

We keep a set S of vertices whose closest distances to the source, vertex 0, are known and add one vertex to S at each stage. We maintain a table distance that gives, for each vertex v, the distance from 0 to v along a path all of whose vertices are in S, except possibly the last one. To determine what vertex to add to S at each step, we apply the **greedy** criterion of choosing the vertex v with the smallest distance recorded in the table distance, such that v is not already in distance.

Finding a Shortest Path

- Choose the vertex v with the smallest distance recorded in the table distance, such that v is not already in distance.

- Prove that the distance recorded in distance really is the length of the shortest path from source to v. For suppose that there were a shorter path from source to v, such as shown below. This path first leaves S to go to some vertex x, then goes on to v (possibly even reentering S along the way). But if this path is shorter than the colored path to v, then its initial segment from source to x is also shorter, so that the greedy criterion would have chosen x rather than v as the next vertex to add to S, since we would have had distance[x] < distance[v].
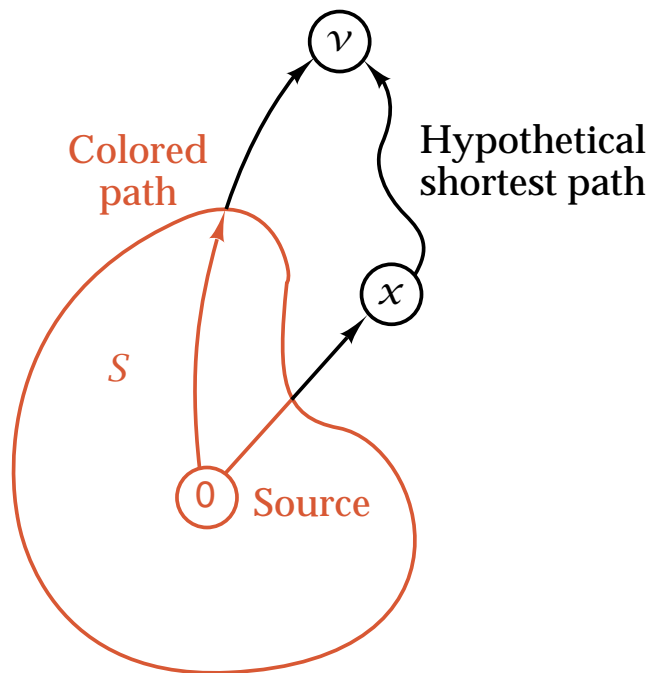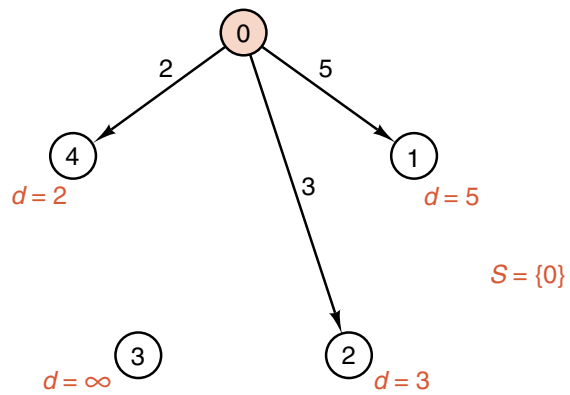

- When we add v to S, we think of v as now colored and also color the shortest path from source to v. Next, we update the entries of distance by checking, for each vertex w not in S, whether a path through v and then directly to w is shorter than the previously recorded distance to w.
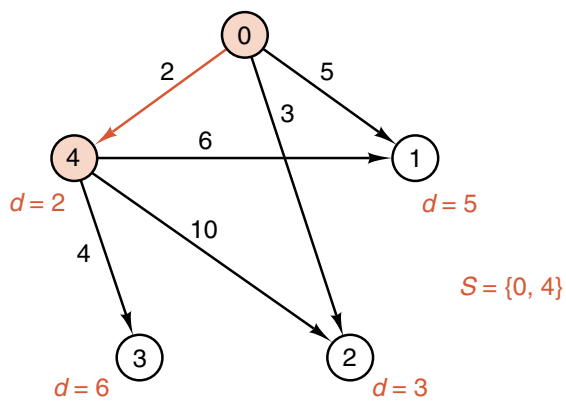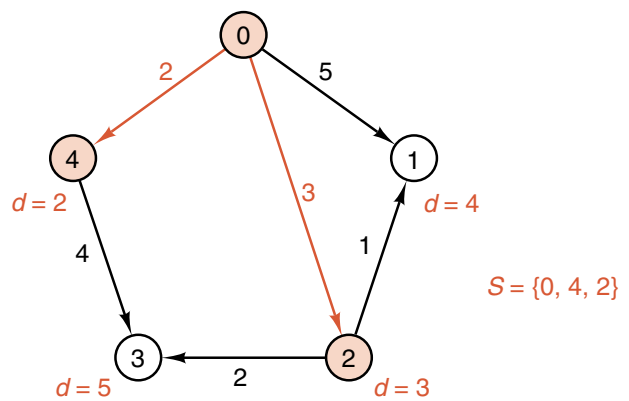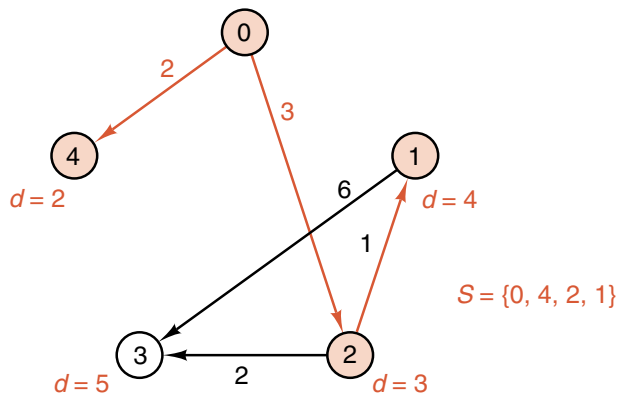
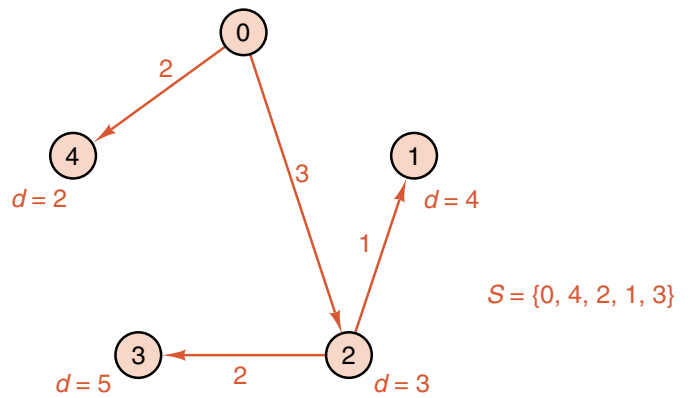# Example of Shortest Path
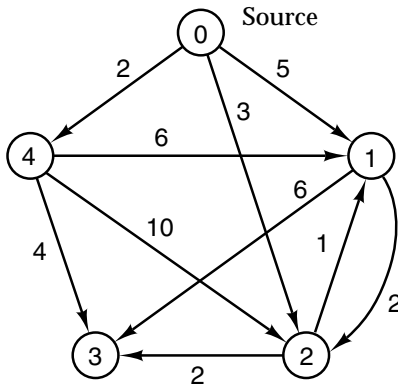
```cpp
template <class Weight, int graph_size>
class Digraph {
public:                    //    Add a constructor and methods for Digraph input and output.
  void set_distances(Vertex source, Weight distance [ ]) const;
protected:
  int count;
  Weight adjacency [graph_size] [graph_size];
};

template <class Weight, int graph_size>
void Digraph<Weight, graph_size> :: set_distances(Vertex source,
                                                  Weight distance [ ]) const
/* Post: The array distance gives the minimal path weight from vertex source to each
         vertex of the Digraph. */
{ Vertex v, w; bool found [graph_size];   //    Vertices found in S
  for (v = 0;  v < count;  v++) {
    found [v] = false;
    distance [v] = adjacency [source] [v];
  }
  found [source] = true;        //    Initialize with vertex source alone in the set S.
  distance [source] = 0;
  for (int i = 0;  i < count;  i++) {  //    Add one vertex v to S on each pass.
    Weight min = infinity;
    for (w = 0;  w < count;  w++) if (! found [w])
      if (distance [w] < min) {
        v = w;
        min = distance [w];
      }
    found [v] = true;
    for (w = 0;  w < count;  w++) if (! found [w])
      if (min + adjacency [v] [w] < distance [w])
        distance [w] = min + adjacency [v] [w];
  }
}
```
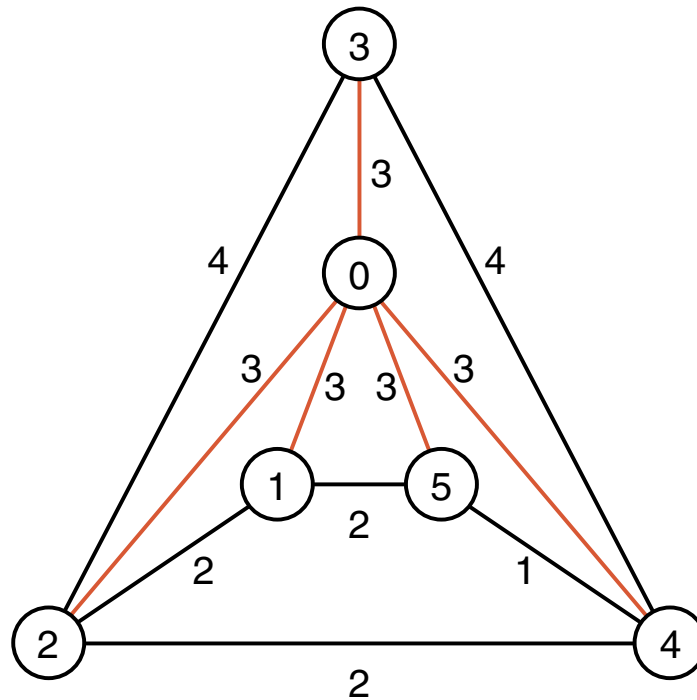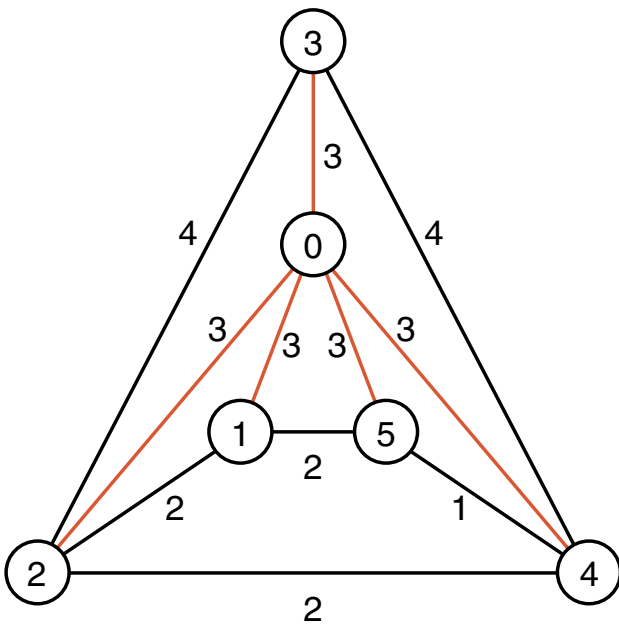
Minimal Spanning Trees

■ Shortest paths from source 0 to all vertices in a network:

■ If the original network is based on a connected graph G, then the shortest paths from a particular source vertex to all other vertices in G form a tree that links up all the vertices of G.

■ A (connected) tree that is build up out of all the vertices and some of the edges of G is called a ***spanning tree*** of G.

> DEFINITION A ***minimal spanning tree*** of a connected network is a spanning tree such that the sum of the weights of its edges is as small as possible.

Two Spanning Trees

Weight sum of tree = 15
(a)
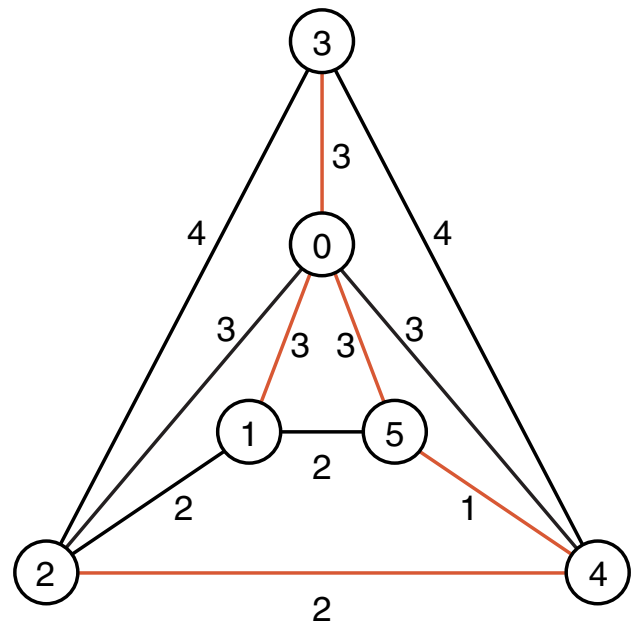
Weight sum of tree = 12
(b)

Prim's Algorithm for Minimal Spanning Trees

- Start with a source vertex. Keep a set X of those vertices whose paths to source in the minimal spanning tree that we are building have been found. Keep the set Y of edges that link the vertices in X in the tree under construction. The vertices in X and edges in Y make up a small tree that grows to become our final spanning tree.

- Initially, source is the only vertex in X, and Y is empty. At each step, we add an additional vertex to X: This vertex is chosen so that an edge back to X has as small as possible a weight. This minimal edge back to X is added to Y.

- For implementation, we shall keep the vertices in X as the entries of a Boolean array component. We keep the edges in Y as the edges of a graph that will grow to give the output tree from our program.

- We maintain an auxiliary table neighbor that gives, for each vertex v, the vertex of X whose edge to v has minimal cost. We also maintain a second table distance that records these minimal costs. If a vertex v is not joined by an edge to X we shall record its distance as the value infinity. The table neighbor is initialized by setting neighbor[v] to source for all vertices v, and distance is initialized by setting distance[v] to the weight of the edge from source to v if it exists and to infinity if not.

- To determine what vertex to add to X at each step, we choose the vertex v with the smallest value recorded in the table distance, such that v is not already in X. After this we must update our tables to reflect the change that we have made to X. We do this by checking, for each vertex w not in X, whether there is an edge linking v and w, and if so, whether this edge has a weight less than distance[w]. In case there is an edge (v, w) with this property, we reset neighbor[w] to v and distance[w] to the weight of the edge.
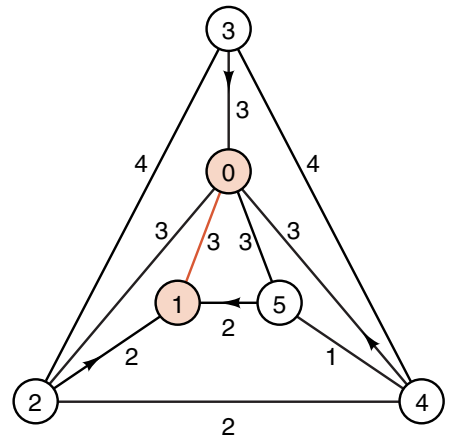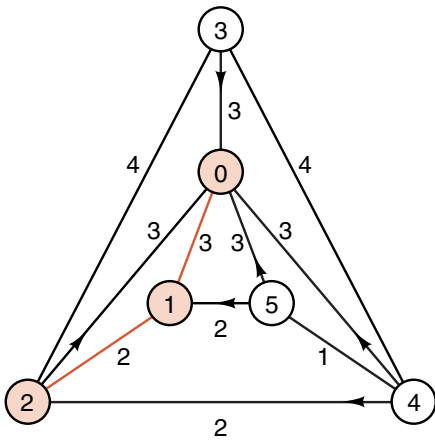
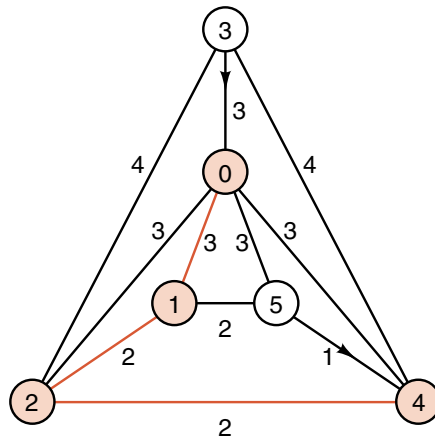# Example of Prim's Algorithm
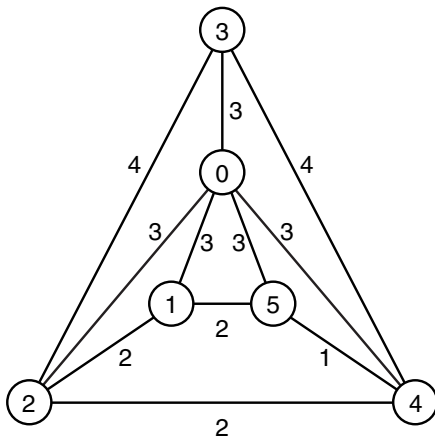
(a)
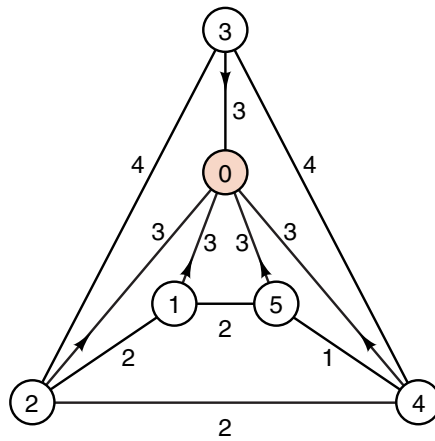
(b)

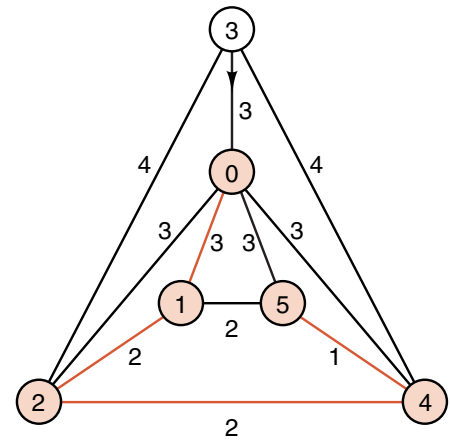(c)

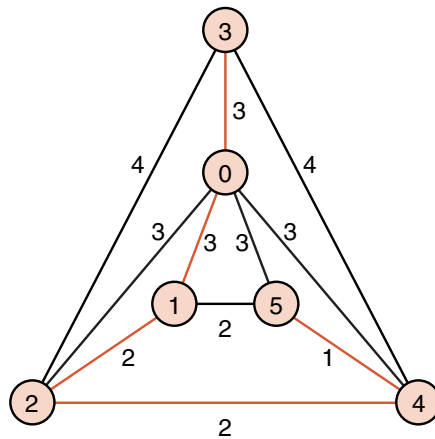(d)

(e)

(f)

Minimal spanning tree, weight sum = 11
(g)

Implementation of Prim's Algorithm

```
template <class Weight, int graph_size>
class Network:  public Digraph<Weight, graph_size> {
public:
  Network( );
  void read( );                    //    overridden method to enter a Network
  void make_empty(int size = 0);
  void add_edge(Vertex v, Vertex w, Weight x);
  void minimal_spanning(Vertex source,
                        Network<Weight, graph_size> &tree) const;
};
```

- read is overridden to make sure that the weight of any edge (v, w) matches that of the edge (w, v): In this way, we preserve our data structure from the potential corruption of undirected edges.

- make_empty(**int** size) creates a Network with size vertices and no edges.

- add_edge adds an edge with a specified weight to a Network.

- As for the shortest-path algorithm, we assume that the **class** Weight has comparison operators.

- We expect clients to declare a largest possible Weight value called infinity.

```
template <class Weight, int graph_size>
void Network < Weight, graph_size > :: minimal_spanning(Vertex source,
                          Network<Weight, graph_size> &tree) const
/* Post:  The Network tree contains a minimal spanning tree for the connected com-
          ponent of the original Network that contains vertex source.  */
{ tree.make_empty(count);
  bool component[graph_size];    //    Vertices in set X
  Weight distance[graph_size];   //    Distances of vertices adjacent to X
  Vertex neighbor[graph_size];   //    Nearest neighbor in set X
  Vertex w;
  for (w = 0;  w < count;  w++) {
    component[w] = false;
    distance[w] = adjacency[source][w];
    neighbor[w] = source;        }
  component[source] = true;   //    source alone is in the set X.
  for (int i = 1;  i < count;  i++) {
    Vertex v;                        //    Add one vertex v to X on each pass.
    Weight min = infinity;
    for (w = 0;  w < count;  w++) if (!component[w])
      if (distance[w] < min) {
        v = w;
        min = distance[w]; }
    if (min < infinity) {
      component[v] = true;
      tree.add_edge(v, neighbor[v], distance[v]);
      for (w = 0;  w < count;  w++) if (!component[w])
        if (adjacency[v][w] < distance[w]) {
          distance[w] = adjacency[v][w];
          neighbor[w] = v;                        }
    }
    else break;                        //    finished a component in disconnected graph
  }
}
```
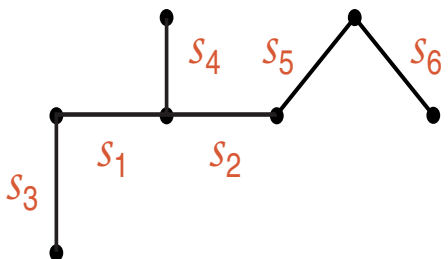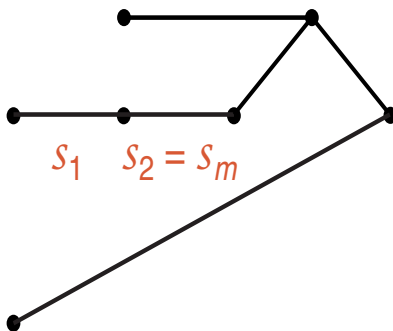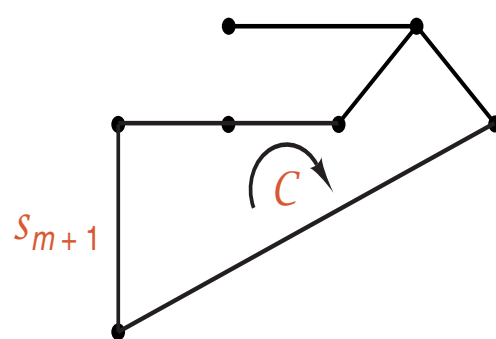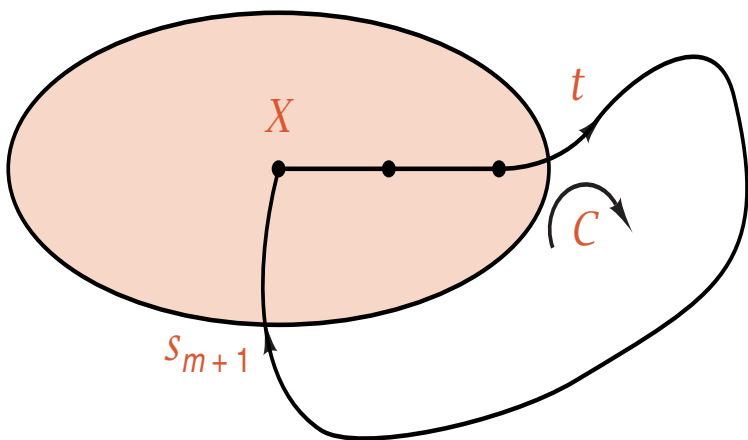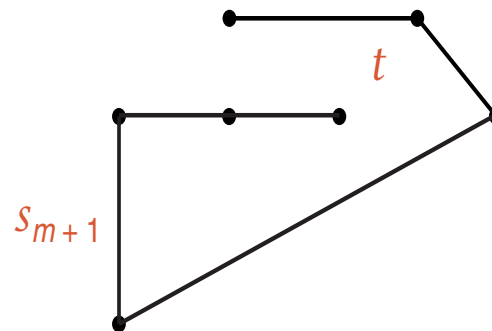
S

(a)

T

(b)

$T + s_{m+1}$

(c)

C leaves and reenters X

(d)

$U = T + s_{m+1} - t$

(e)

Verification of Prim's Algorithm

> For a connected graph G, the spanning tree S produced by Prim's algorithm has a smaller edge-weight sum than any other spanning tree of G.

- Prim's algorithm determines a sequence of edges $s_1, s_2, \ldots, s_n$ that make up the tree S, where s_1 is the first edge added to the set Y in Prim's algorithm, s_2 is the second edge added to Y, and so on.

- We prove that if m is an integer with $0 \leq m \leq n$, then there is a minimal spanning tree that contains the edges s_i with $i \leq m$.

- We use induction on m. The base case, where $m = 0$, is vacuously true.

- The final case, with $m = n$, shows that there is a minimal spanning tree that contains all the edges of S, and therefore agrees with S, so S is a minimal spanning tree.

- For the inductive step, assume that $m < n$ and T is a minimal spanning tree that contains the edges s_i with $i \leq m$. We prove that there is a minimal spanning tree U with these edges and s_{m+1}. If s_{m+1} already belongs to T, we can simply set $U = T$, so we shall also suppose that s_{m+1} is not an edge of T.

- Let X be the set of vertices of S belonging to the edges

$$s_1, s_2, \ldots, s_m$$

  and let R be the set of remaining vertices of S. In Prim's algorithm, the selected edge s_{m+1} links a vertex of X to R, and s_{m+1} is at least as cheap as any other edge between these sets.

- Adding s_{m+1} to T must create a cycle C, since the connected network T certainly contains a multi-edge path linking the endpoints of s_{m+1}. The cycle C must contain an edge $t \neq s_{m+1}$ that links X to R, since as we move once around the closed path C we must enter the set X exactly as many times as we leave it.

- Prim's algorithm guarantees that the weight of s_{m+1} is less than or equal to the weight of t. Therefore, the new spanning tree U obtained from T by deleting t and adding s_{m+1} has a weight sum no greater than that of T. We deduce that U must also be a minimal spanning tree of G, but U contains the sequence of edges $s_1, s_2, \ldots, s_m, s_{m+1}$. This completes the induction.

A Story

I married a widow who had a grown-up daughter. My father, who visited us quite often, fell in love with my step-daughter and married her. Hence, my father became my son-in-law, and my step-daughter became my mother.

Some months later, my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson.

My wife is my grandmother, since she is my mother's mother. Hence, I am my wife's husband and at the same time her step-grandson; in other words,

I am my own grandfather.

Pointers and Pitfalls

1. Graphs provide an excellent way to describe the essential features of many applications, thereby facilitating specification of the underlying problems and formulation of algorithms for their solution. Graphs sometimes appear as data structures but more often as mathematical abstractions useful for problem solving.

2. Graphs may be implemented in many ways by the use of different kinds of data structures. Postpone implementation decisions until the applications of graphs in the problem-solving and algorithm-development phases are well understood.

3. Many applications require graph traversal. Let the application determine the traversal method: depth first, breadth first, or some other order. Depth-first traversal is naturally recursive (or can use a stack). Breadth-first traversal normally uses a queue.

4. Greedy algorithms represent only a sample of the many paradigms useful in developing graph algorithms. For further methods and examples, consult the references.