CHAPTER 7 TREES

SECTION 7.1

DEFINITION:

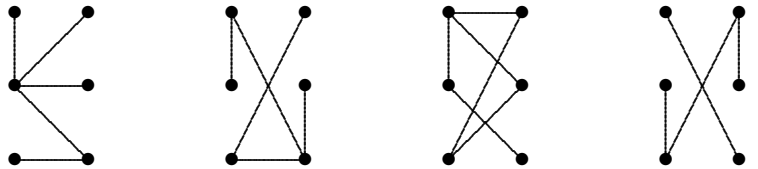
A **TREE** is a connected (undirected) graph with no cycles.

REMARK

A tree cannot contain loops or multiple edges since they are cycles.

EXAMPLE

G_1, G_2 are trees. G_3 is not a tree as it contains a cycle. G_4 is not a tree because it is not connected.



G_1 G_2 G_3 G_4

DEFINITION:

A graph is a **FOREST** if its connected components are trees.

Thus G_1, G_2, G_4 are all forests.

THEOREM:

A graph is a tree iff there is a unique simple path between any two of its vertices

PROOF: Suppose T is a tree. Then every two of its vertices are connected by a path. Since every path can be "reduced" to a simple path (see tutorial), we conclude that every pair of vertices are connected by a simple path.

Suppose there is a pair of vertices a, b with two different simple paths:

$$P_1 : a = u_0 u_1 \dots u_n = b, \quad P_2 : a = v_0 v_1 \dots v_m = b.$$

Let k be the index for which $u_i = v_i$, $i = 0, \dots, k$ and $u_{k+1} \neq v_{k+1}$. (This k exists because P_1 and P_2 are distinct.) Let s and t be indices such that $s, t > k+1$, $u_s = v_t$, and the vertices $u_{k+1}, \dots, u_{s-1}, v_{k+1}, \dots, v_{t-1}$ are distinct. Then $u_k, u_{k+1}, \dots, u_{s-1}, u_s =$

Chapter 10 in text

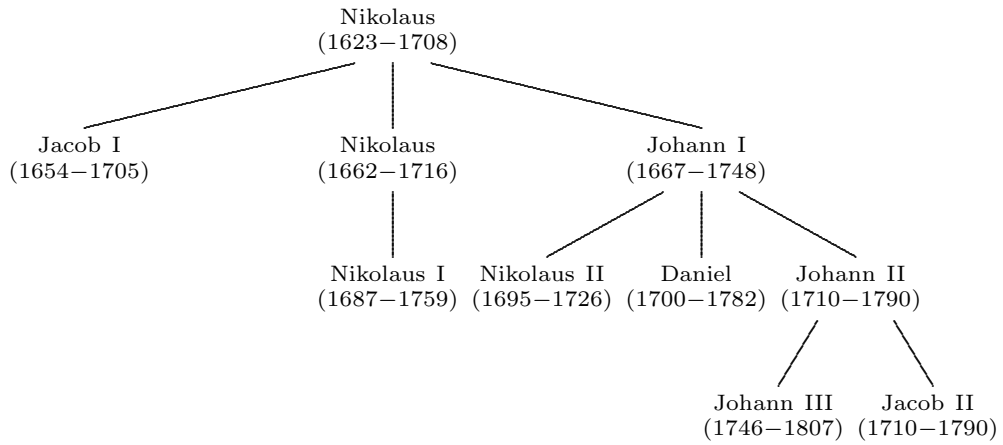
$v_t, v_{t-1}, \ldots, v_k$ is a cycle in T, a contradiction. Thus every two different vertices are connected by a unique simple path.

Next we prove the converse. Suppose that every two different vertices are connected by a unique simple path. Then T is connected. Moreover, T cannot contain any cycle as any two different vertices on the cycle will be connected by two different simple paths. Thus T is a tree.

DEFINITION:

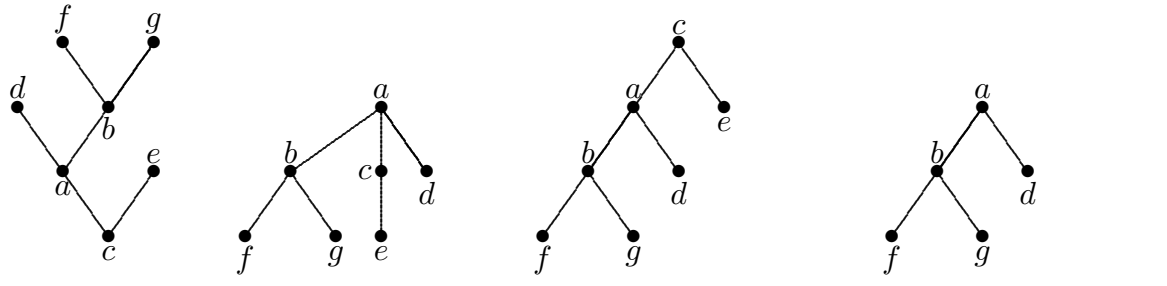
A **ROOTED TREE** is a tree in which one vertex has been designated as the **ROOT** and all edges are directed away from the root.

An example of a rooted tree is shown below.



The Bernoulli Family of Mathematicians

Another example shows T with different roots, T_1 and T_2.



T T_1 With root at a T_2 With root at c Subtree of T_2 rooted at a

TERMINOLOGY

Suppose T is a rooted tree with root a. If $\overline{uv}$ is an edge, then u is the **PARENT** of v and v is a **CHILD** of u. Vertices with the same parent are called **SIBLINGS**. If $u \neq v$, and

there is a simple path from u to v (with u above v), then u is an **ANCESTOR** of v and v a **DESCENDANT** of u. A vertex if called a **LEAF** if it has no children. A vertex that is not a leaf is called an **INTERNAL VERTEX**. Thus the root a is an internal vertex unless it is the only vertex in the tree in which case it is a leaf.

If u is a vertex of T, the **SUBRTREE** rooted at u is the subgraph consisting of u and all its descendants.

For example, in T_2,

- b and d are children of a and they are siblings and a is their parent.
- f, g, d, e are leaves. The rest are internal vertices.
- a, c are ancestors of d. b, d, f, g are descendants of a.

DEFINITION:

A rooted tree is called an m-ary tree if each of its internal vertices has at most m children.

An m-ary is **FULL** if every internal vertex has exactly m children.

When $m = 2$, such a tree is called a **BINARY TREE**.

For example T_1 is a 3-ary tree while T_2 is binary.

DEFINITION:

An **ORDERED ROOTED TREE** is a rooted tree in which the children of each vertex are ordered. Ordered rooted trees are drawn so that the children are ordered from left to right.

In an ordered binary tree, if a vertex u has 2 children, the first child is called it **LEFT CHILD** and the second child is the **RIGHT CHILD**. The subtree rooted at the left child is called its **LEFT SUBTREE** and the subtree rooted at the right child is called its **RIGHT SUBTREE**.

For example, for tree T_2, the left subtree of c is the subtree rooted at a while the right subtree consists of a single vertex e.

Trees are used as models in many areas. For example, in chemistry, the molecular structure of saturated hydrocarbons are trees. The organizational structures of large organizations and family trees are often presented as rooted trees. The computer file systems with its root directory and subdirectories are rooted trees. (Read more on this in the text book.)

PROPERTIES OF TREES

THEOREM:

A tree with $n \geq 2$ vertices has at least two vertices of degree 1.

PROOF: Take a simple path of maximum length which connects two distinct vertices:

$$v_0 v_1 \ldots, v_m.$$

Suppose $\deg(v_0) > 1$. Then v_0 is adjacent to a vertex $x \neq v_1$. If $x = v_i$ for some $i \in \{2, \ldots, m\}$, we have a cycle

$$v_0 \quad v_1 \quad \ldots \quad v_i \quad v_0$$

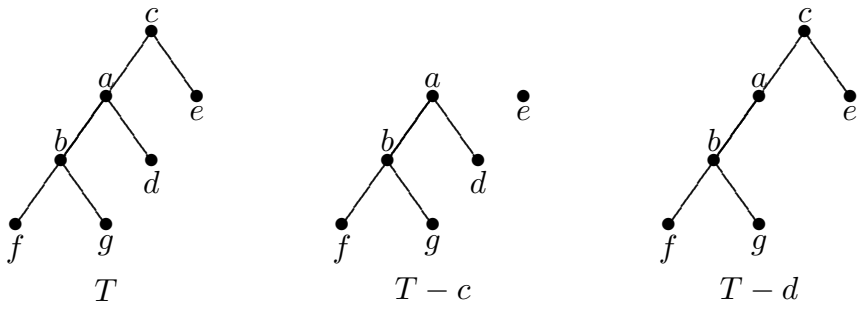
If not, then $xv_0v_1 \ldots v_m$ is a longer path, a contradiction. Thus $\deg(v_0) = 1$. Similarly, $\deg(v_m) = 1$.

THEOREM:

Let v be a vertex of degree 1 in a tree T with $n \geq 2$ vertices. Then $T - v$ is also a tree.

Note: For any graph G and a vertex $v \in V(G)$, $G - v$ is the graph obtained from G by deleting the vertex v and all the edges incident with v.

EXAMPLE



PROOF: (of the theorem) Since every cycle in $T - v$ would be a cycle of T, we conclude that $T - v$ contains no cycle. Next we have to prove that $T - v$ is connected. For any two distinct vertices x, y in $T - v$, there is a unique simple path $P : u_1u_2 \ldots u_t$, where $u_1 = x$, $u_t = y$. Since $\deg(u_i) \geq 2$, for $i = 2, \ldots, t$, $u_i \neq v$. Thus P is still a path in $T - v$. Thus $T - v$ is connected.

THEOREM:

A tree with n vertices has $n - 1$ edges.

PROOF: We prove by induction on n.

When $n = 1$, the tree has only a single vertex and no edge. Thus the theorem is true.

Now assume any tree with $k(\geq 1)$ vertices has $k - 1$ edges. Let T be a tree with $k + 1$ vertices. Let v be a vertex of degree 1 of T. Then $T - v$ is a tree with k vertices. By the

induction hypothesis, it has $k-1$ edges. Hence T has k edges. The proof is now complete by induction.

THEOREM:

A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices

PROOF: Each vertex, except the root, is the child of exactly one internal vertex. Since each internal vertex has m children, there are altogether mi children. Hence the total number of vertices is $mi + 1$.

THEOREM:

Suppose a full m-ary tree has n vertices, i internal vertices and ℓ leaves. Then

$$n = mi + 1 = \frac{m\ell - 1}{m - 1}$$
$$i = \frac{n-1}{m} = \frac{\ell - 1}{m-1}$$
$$\ell = \frac{(m-1)n + 1}{m} = (m-1)i + 1$$

PROOF: Let i and ℓ be the number of internal vertices and leaves of the tree and n be the total number of vertices. Then $n = i + \ell$ and $n = mi + 1$. Eliminating i from these equations, we get $mn - n = m\ell - 1$. From this, we get the first identity. Others can be obtained in the same way.

EXAMPLE

Suppose someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other persons. Some people do this but others do not. Suppose that no one receives more than 1 letter and that there are 100 people who have received the letter did not send it on. How many people have seen the letter, including the first person?

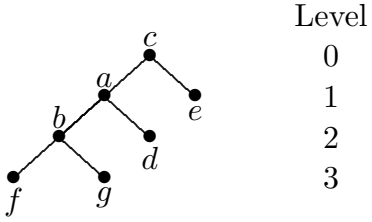
Soln: The situation can be represented by a full 4-ary tree and $\ell = 100$ leaves. The number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$.

DEFINITION:

Let T be a tree rooted at a. The **LEVEL** of a vertex v is the length of the simple path from a to v. The level of a is defined to be 0. The **HEIGHT** of T is the maximum of the levels of vertices. In other words, the height is the length of the longest path from the root to its vertices.

A rooted m-ary tree of height h is **BALANCED** if all leaves are at level h or $h-1$.

For the binary tree T_2, the level of a, e is 1, the level of b, d is 2 while the level of f, g is 3. The height is 3.



T_2

THEOREM:

In an m-ary tree there are at most m^k vertices at level k. If the height is h, there are at most m^h leaves.

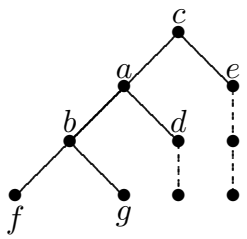
PROOF: We shall prove the first part by induction on k.

Basis step: At level 0, there is only $1 = m^0$ vertex.

Inductive step: Suppose that there are at most m^k vertices at level k, where $k \geq 0$. Since each vertex has at most m children, the number of vertices at level $k+1$ is at most $m \cdot m^k = m^{k+1}$ and the proof of the induction step is complete.

For the second part, if all the leaves at at level h, then obviously there are at most m^h leaves.

Suppose there is a leaf u at level $k < h$. Then we can add new vertices v_{k+1}, v_{k+2}, …, v_h and new edges uv_{k+1}, $v_{k+1}v_{k+2}$, …, $v_{h-1}v_h$ to obtain a new tree T'. In T', v_h replaces u as a leaf. Repeat this process for all the leaves that are not level h, we can "transfer" all the leaves to level h. (See example below). Thus the number of leaves is $\leq m^h$.



d, e are "transferred to level 3

The following is an easy corollary.

THEOREM:

If an m-ary tree of height h has ℓ leaves, then $h \geq \lceil \log_m \ell \rceil$. If the tree is full and balanced, then $h = \lceil \log_m \ell \rceil$.

PROOF: Since $\ell \leq m^h$, we have $h \geq \lceil \log_m \ell \rceil$.

When the tree is full and balanced, the number of vertices at level k, $0 \leq k \leq h-1$, is m^k. Thus there are m^{h-1} vertices at level $h-1$. Each of these vertices is either a leaf or has m children at level h which are leaves. Since there is at least one vertex at level h, we have $m^{h-1} < \ell \leq m^h$. From this we get $h = \lceil \log_m \ell \rceil$.

SECTION 7.2 APPLICATIONS OF TREES

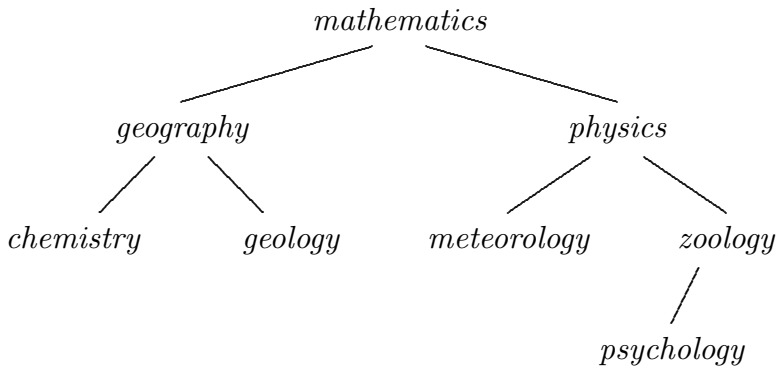
BINARY SEARCH TREES

We use a recursive procedure to construct the binary search tree of a list of items. The first item is assigned to the root. To add a new item, compare it to all the items already in the tree, starting with the root. If it is larger, move it to the right and if this vertex has no right child, this new item becomes the right child of this vertex. Move to the left if it is smaller.

EXAMPLE

Construct the binary search tree of the following items using lexicographic order:

mathematics, physics, geography, zoology, meteorology, geology, psychology, chemistry



The Binary Search Tree

REMARK

The maximum number of comparisons needed to search for an item is the height of the tree. Thus the maximum is minimized if the tree is balanced. There are ways to construct a balanced search tree.

UNIVERSAL ADDRESS SYSTEM

Procedures for traversing all vertices of an ordered rooted tree rely on the ordering of the children. We'll describe one way to order all the vertices of an ordered rooted tree. To produce this order, we must first label all the vertices. We do this recursively:

1. Label the root 0.

2. Label its children $1, 2, \ldots$ from left to right.

3. For each vertex v at level n with label A, label its children from left to right $A.1, A.2, A.3, \ldots$.

Following this procedure, a vertex at level n has label that looks like $x_1.x_2.x_3 \cdots .x_n$.

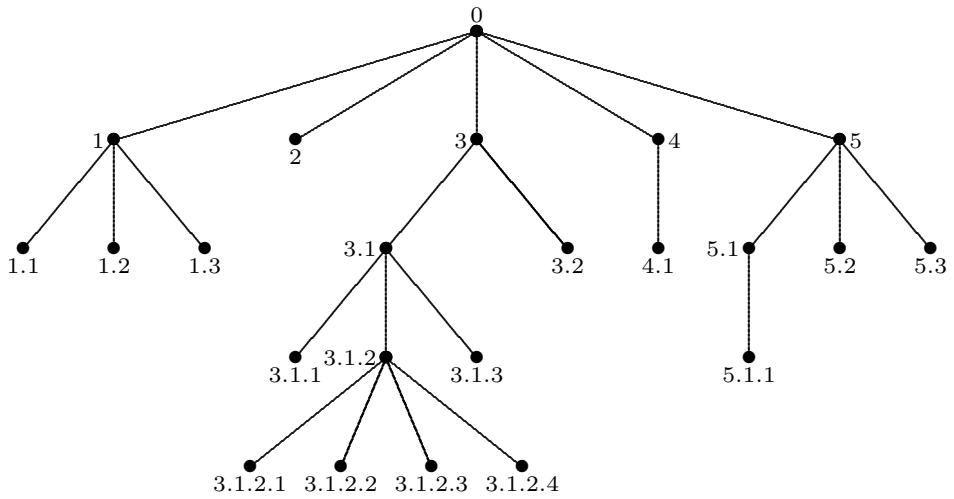
This labelling is called the **UNIVERSAL ADDRESS SYSTEM** of the rooted ordered tree.

We can order the vertices of the tree using the *lexicographic ordering.* Two vertices

$$x_1.x_2.x_3 \cdots .x_n < y_1.y_2.y_3 \cdots .y_m$$

if there exists $1 \leq i \leq n$, with $x_1 = y_1$, $x_2 = y_2$, ..., $x_{i-1} = y_{i-1}$ and $x_i < y_i$ or if $x_j = y_j$ for $j = 1, \ldots, n$ and $n < m$. (Think of the ordering used in the dictionary.)

EXAMPLE



Vertices are ordered:

$$0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2$$
$$< 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3$$

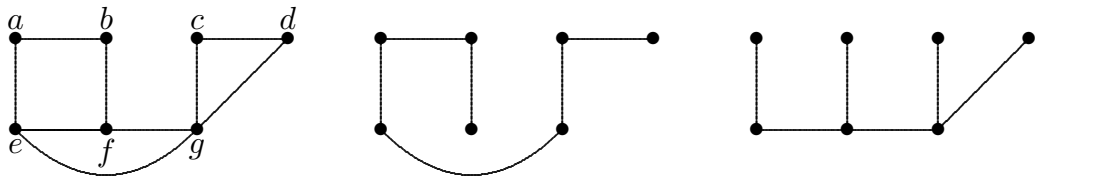
Many information can be obtained. For example, the unique path from 0 to 3.1.2.3 is, tracing backwards,

$$3.1.2.3, \quad 3.1.2, \quad 3.1, \quad 3, \quad 0$$

SECTION 7.3 SPANNING TREES

DEFINITION:

Let G be a simple graph. A **SPANNING TREE** of G is a subgraph of G that is a tree containing every vertex of G.



G A Spanning Tree Another Spanning Tree

THEOREM:

A simple graph is connected iff it contains a spanning tree.

PROOF: First suppose that a simple graph G contains a spanning tree T. Since T is connected, G is certainly connected.

Conversely, suppose that G is connected. If G does not contain any cycle, G itself is a spanning tree.

Now suppose that G contains a cycle C. The deleting an edge from C leaves a connected graph. We can repeat this process until there is no cycles left, i,.e., we are left with a tree. This is a spanning tree of G.

DEPTH FIRST SEARCH

This is an algorithm to construct spanning tree in a connected graph.

Step 0. Initially, T is empty.

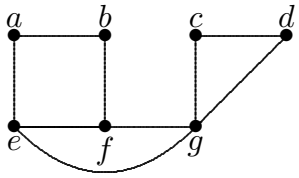
Step 1. Choose an arbitrary vertex and label it 1. Add 1 to T.

Step 2. After a vertex has just been labelled j, choose an unlabelled vertex that is adjacent to j. Label it as $j+1$. Add vertex $j+1$ and the edge joining it to j to T.

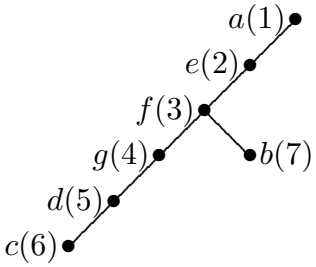
If all vertices adjacent to j have been labelled, backtrack to vertex $j-1$ and repeat the process. Stop when all vertices are labelled. The resulting subgraph T is a spanning tree rooted at 1, called a **DEPTH FIRST SEARCH TREE** of G. The labelling obtained is called a **DEPTH FIRST SEARCH LABELLING**.

EXAMPLE

Find a DFS tree in the following graph.



G



DFS tree

1. Choose a and label it 1.

2. Choose a neighbour of 1, e, and label it 2. Add edge 12.

3. Choose a neighbour of 2, f, and label it 3. Add edge 23.

4. Choose a neighbour of 3, g, and label it 4. Add edge 34.

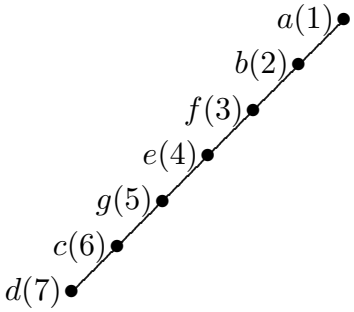
5. Choose a neighbour of 4, d, and label it 5. Add edge 45.

6. Choose a neighbour of 5, c, and label it 6. Add edge 56.

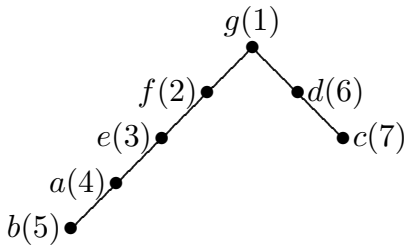
7. Neighbours of 6 are now all labelled as are 4 a,d 5. Backtrack to 3. Choose a neighbour of 3, b, and label it 7. Add edge 37.

8. End since all vertices are now labelled. The DFS tree is shown on the left. The numbers in the parentheses are the DFS labelling of the vertices.

Very often, we order the vertices and make choices according to the ordering. Thus in step 1, we would choose the first vertex to be labelled as 1. In step 2, we would choose the first available vertex. For example, the following shows the resulting DFS trees if we use the alphabetical ordering (left) and the reverse alphabetical ordering (right).



Alphabetical Ordering



Reverse Alphabetical Ordering

BREADTH FIRST SEARCH

This is an algorithm to construct a spanning tree of a connected simple graph.

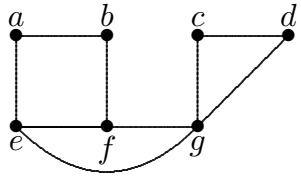
Step 1. Arbitrarily choose a vertex as the root and label it 1. Add it to T.

Step 2. Visit the labelled vertices in ascending order. Suppose vertex j is being visited and that $k(\geq j)$ vertices have been labelled. Labelled all the unlabelled vertices that are adjacent to j as $k+1, k+2, \ldots$ and add the corresponding edges to T.

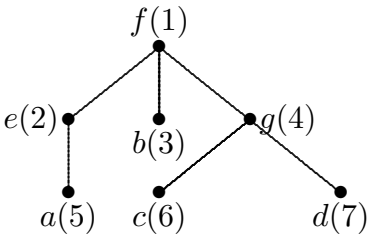
Step 3. Repeat 2 until all vertices are labelled. T is a spanning tree rooted at 1. It is called a **BREADTH FIRST SEARCH TREE**.

EXAMPLE

Find a BFS tree in the following graph.



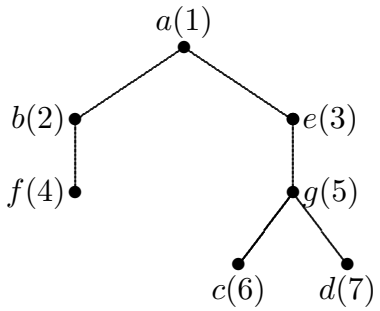
G



BFS Tree

1. Choose f add label 1. Label its neighbours e, b, g as 2, 3, 4. Add edges 12, 13, 14.
2. At vertex 2, label its neighbour a 5 add edge 25.
3. All neighbours of 3 are already labelled.
4. At vertex 4, label the neighbours c, d as 6, 7. Add edges 46, 47.
4. Step as all vertices are labelled. TheBFS tree obtained is shown on the right.

As in the DFS, we can also order the verices and choose according to the ordering. The BFS tree obtained using the alphabetical ordering is shown below.



Alphabetical Ordering

SECTION 7.4 MINIMUM SPANNING TREES

A company plans to build a communications network connecting its 5 computer centres. Any pair of these can be linked with a leased telephone line with differing costs. Which

links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized?

To ensure connectivity, the network have to be a spanning tree. Thus we need a spanning tree in which the total cost is minimized.

DEFINITION:

A **WEIGHTED GRAPH** is a graph G for which each edge e has a real-valued weight $w(e)$.

The **TOTAL WEIGHT** $w(G)$ of a weighted graph is the sum of the weights of all the edges.

A **MINIMAL SPANNING TREE** of a weighted graph is a spanning tree that has the least weight among all spanning trees of the graph.

We shall describe two algorithms to construct a minimum spanning tree. Both are greedy algorithms.

PRIM'S ALGORITHM

G is a weighted connected graph with n vertices.

Step 1. Choose any edge of minimum weight and add it to T.

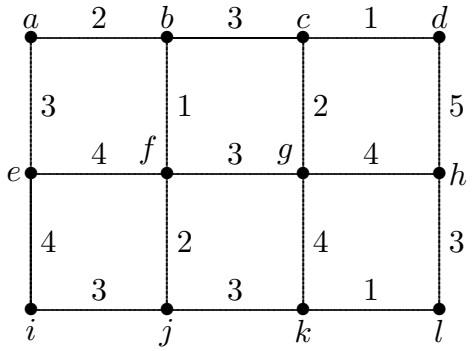
Step 2. Among the edges that join a vertex of T to a vertex not in T, choose one of minimum weight and add it to T.

Step 3. Stop when T has $n-1$ edges.

T is a minimum spanning tree.

EXAMPLE

Find a minimal spanning of the following graph.

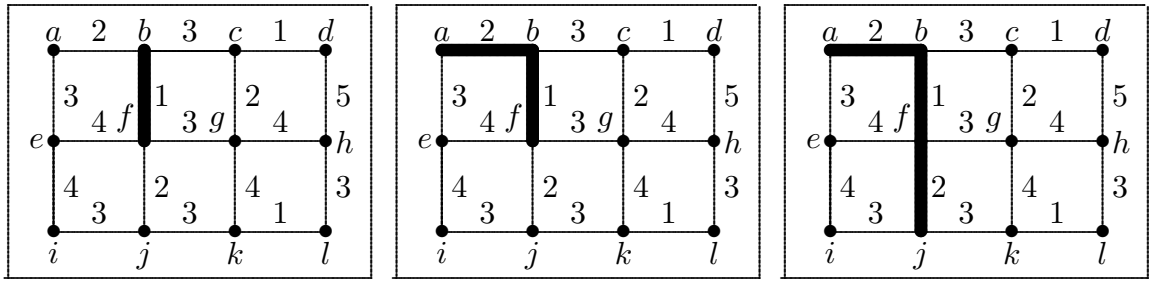


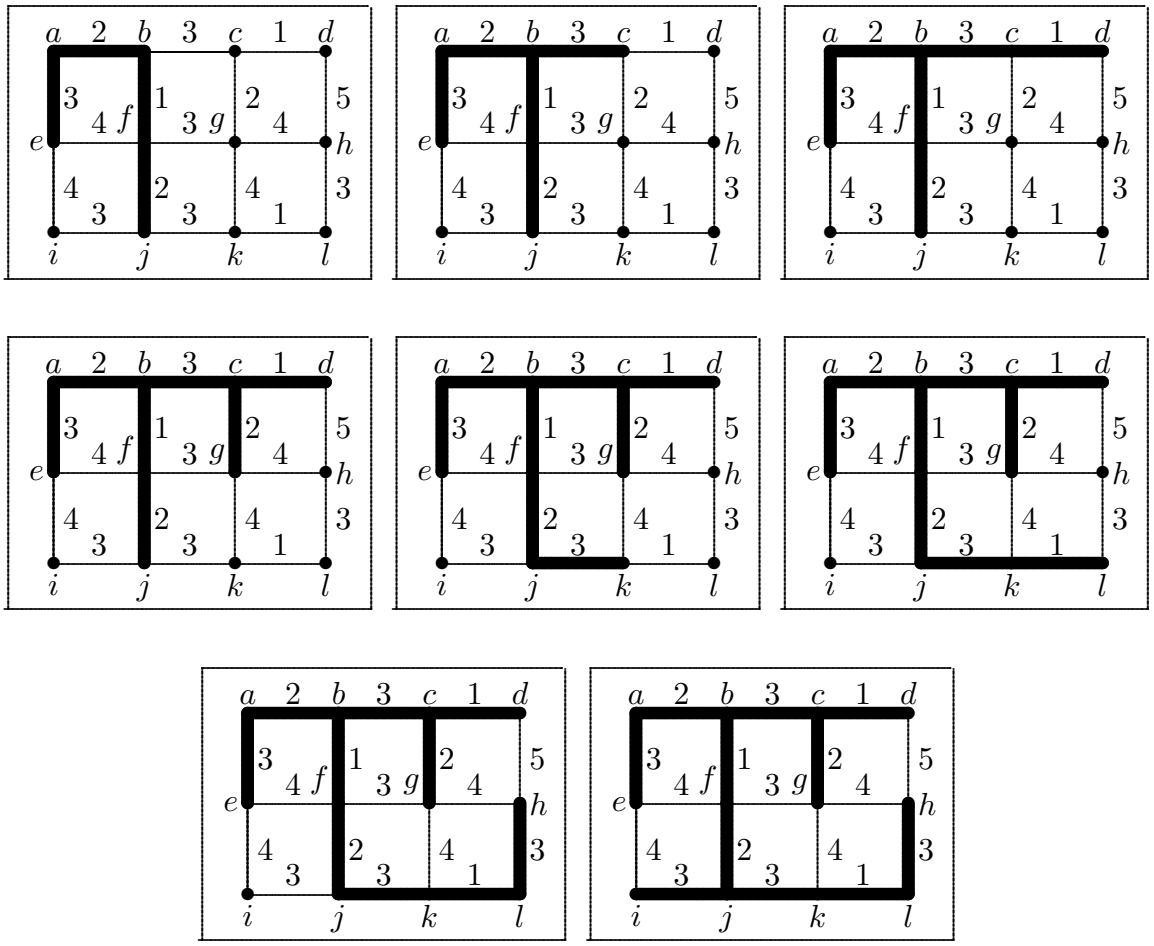
The edges are picked in order:

$$bf, \quad ab, \quad jf, \quad ae, \quad bc, \quad cd, \quad cg, \quad jk, \quad kl, \quad lh, \quad ij$$

Weight: 24

The following sequence shows that construction of the tree.





KRUSKAL'S ALGORITHM

G is a weighted connected graph with n vertices.

Step 1. Sort the edges in order of increasing weight. T consists of isolated vertices of G.

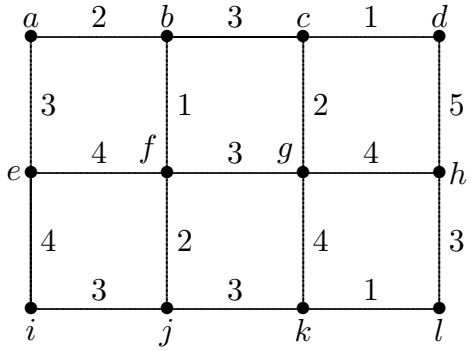
Step 2. Examine the edges in order. Add it to T if it joins two distinct connected components of T.

Step 3. Stop when T has $n-1$ edges.

T is a minimum spanning tree.

EXAMPLE

Find a minimal spanning of the following graph.



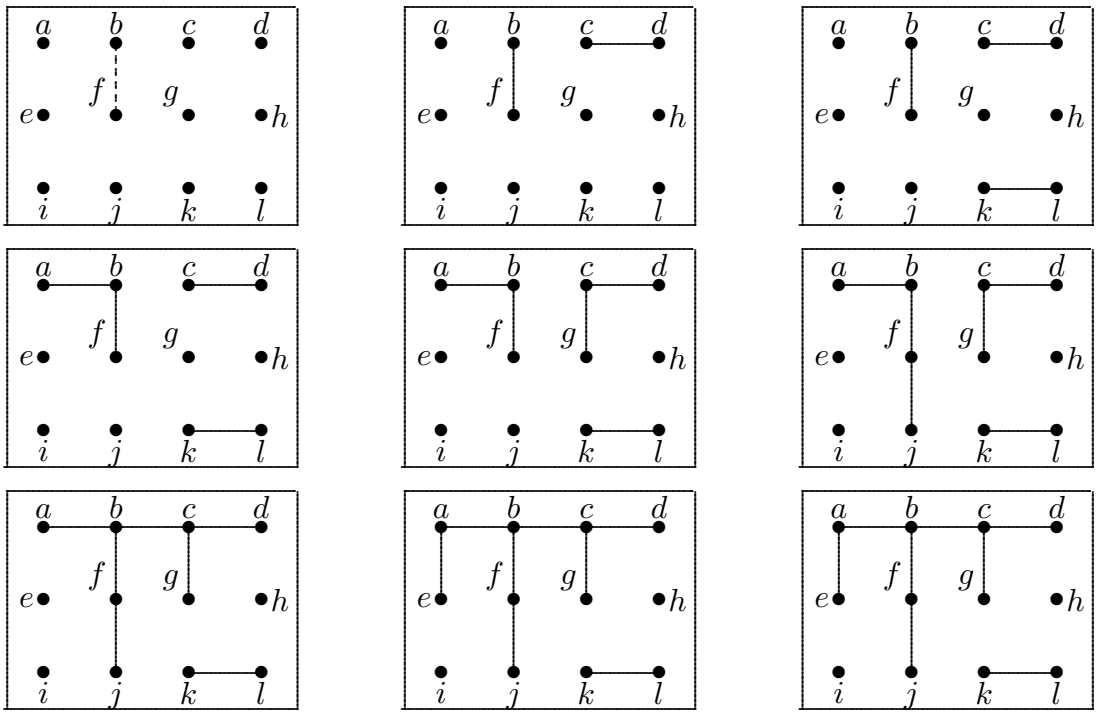
Kruskal's algorithm:

1. Order the edges:

bf, cd, kl, ab, cg, fj, bc, ae , fg, hl, ij, jk, ef, gh, ei, gk, dh

Edges picked:bf, cd, kl, ab, cg, fj, bc, ae, hl, ij, jk. Weight: 24

The following show the way the tree is constructed.



fg not added