

8.1 Introduction to trees: basic concepts-Reading

Notebook: Discrete Mathematics [CM1020]

Created: 2019-10-07 2:31 PM

Updated: 2019-12-26 1:48 PM

Author: SUKHJIT MANN

Cornell Notes	Topic: 8.1 Introduction to trees: basic concepts-Reading	Course: BSc Computer Science
		Class: Discrete Mathematics- Reading
		Date: December 26, 2019
Essential Question:		
What is a tree?		
Questions/Cues:		
<ul style="list-style-type: none">• What is a tree?• What is a forest?• What is a rooted tree?• What is a m-ary tree?• What is an ordered rooted/binary tree?• What are some properties of trees?• What is a spanning tree?• What is a Depth-first search?• What is a minimum-cost spanning tree?• What are the two greedy algorithms for finding minimum-cost spanning trees?• What is Kruskal's Algorithm?• What is Prim's Algorithm?		
Notes		
<p>A tree is a connected undirected graph with no simple circuits.</p> <p>Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a simple graph.</p> <p>An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.</p> <p>EXAMPLE 1 Which of the graphs shown in Figure 2 are trees?</p> <p>Solution: G_1 and G_2 are trees, because both are connected graphs with no simple circuits. G_3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G_4 is not a tree because it is not connected.</p> <p>Any connected graph that contains no simple circuits is a tree. What about graphs containing no simple circuits that are not necessarily connected? These graphs are called forests and have the property that each of their connected components is a tree. Figure 3 displays a forest.</p> <p>Trees are often defined as undirected graphs with the property that there is a unique simple path between every pair of vertices. Theorem 1 shows that this alternative definition is equivalent to our definition.</p>		

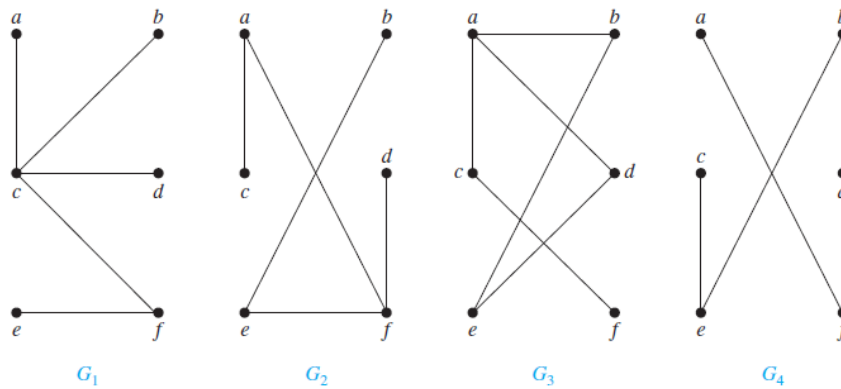


FIGURE 2 Examples of Trees and Graphs That Are Not Trees.

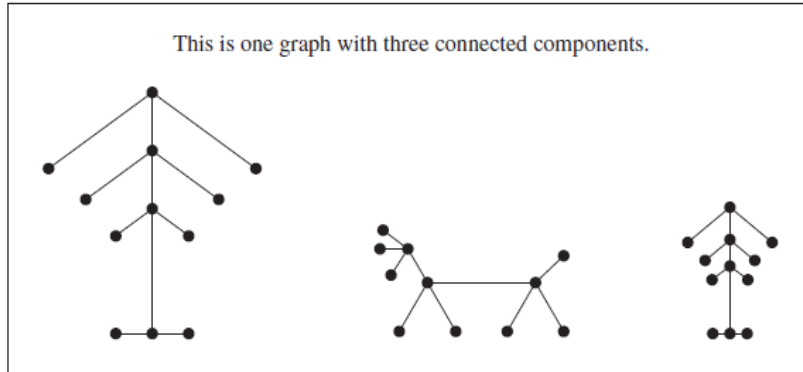


FIGURE 3 Example of a Forest.

A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

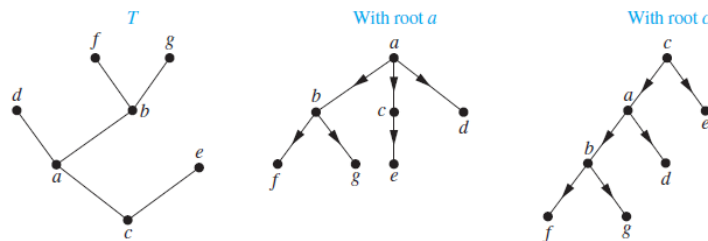


FIGURE 4 A Tree and Rooted Trees Formed by Designating Two Different Roots.

The terminology for trees has botanical and genealogical origins. Suppose that T is a rooted tree. If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v (the reader should show that such a vertex is unique). When u is the parent of v , v is called a **child** of u . Vertices with the same parent are called **siblings**. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as an ancestor. A vertex of a rooted tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

If a is a vertex in a tree, the **subtree** with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

EXAMPLE 2 In the rooted tree T (with root a) shown in Figure 5, find the parent of c , the children of g , the siblings of h , all ancestors of e , all descendants of b , all internal vertices, and all leaves. What is the subtree rooted at g ?



Solution: The parent of c is b . The children of g are h , i , and j . The siblings of h are i and j . The ancestors of e are c , b , and a . The descendants of b are c , d , and e . The internal vertices are a , b , c , g , h , and j . The leaves are d , e , f , i , k , l , and m . The subtree rooted at g is shown in Figure 6.

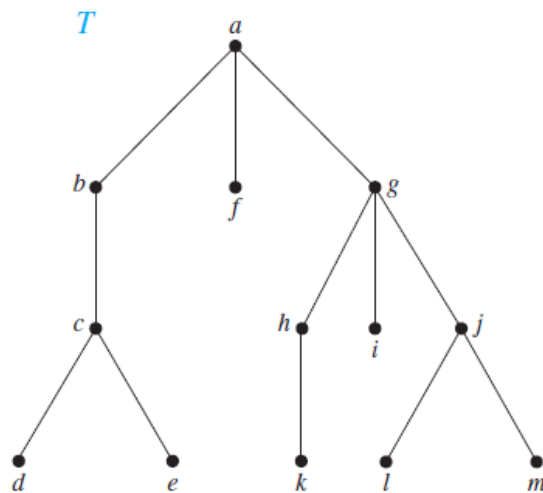


FIGURE 5 A Rooted Tree T .

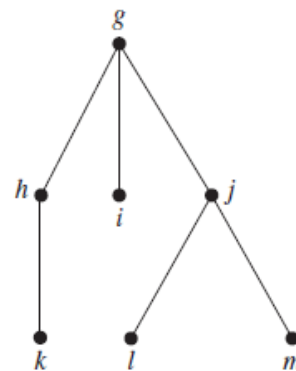


FIGURE 6 The Subtree Rooted at g .

A rooted tree is called an m -ary tree if every internal vertex has no more than m children. The tree is called a *full m -ary tree* if every internal vertex has exactly m children. An m -ary tree with $m = 2$ is called a *binary tree*.

EXAMPLE 3 Are the rooted trees in Figure 7 full m -ary trees for some positive integer m ?

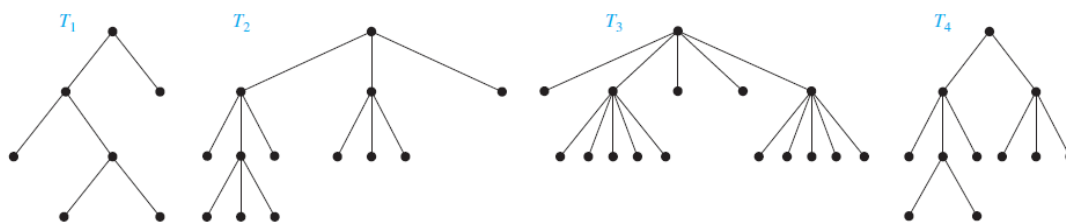


FIGURE 7 Four Rooted Trees.

Solution: T_1 is a full binary tree because each of its internal vertices has two children. T_2 is a full 3-ary tree because each of its internal vertices has three children. In T_3 each internal vertex has five children, so T_3 is a full 5-ary tree. T_4 is not a full m -ary tree for any m because some of its internal vertices have two children and others have three children. ◀

ORDERED ROOTED TREES An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges. We will use such orderings of edges in drawings without explicitly mentioning that we are considering a rooted tree to be ordered.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex. The reader should note that for some applications every vertex of a binary tree, other than the root, is designated as a right or a left child of its parent. This is done even when some vertices have only one child.

EXAMPLE 4 What are the left and right children of d in the binary tree T shown in Figure 8(a) (where the order is that implied by the drawing)? What are the left and right subtrees of c ?

Solution: The left child of d is f and the right child is g . We show the left and right subtrees of c in Figures 8(b) and 8(c), respectively. ◀

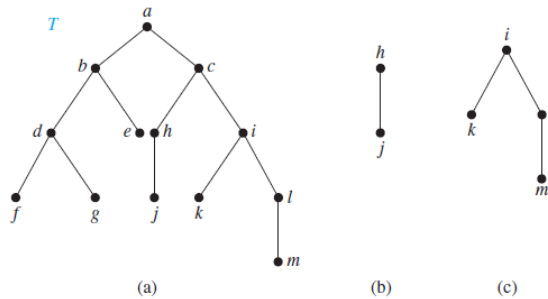


FIGURE 8 A Binary Tree T and Left and Right Subtrees of the Vertex c .

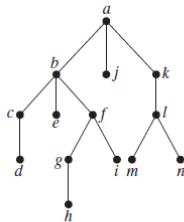
THEOREM 2 A tree with n vertices has $n - 1$ edges.

THEOREM 3 A full m -ary tree with i internal vertices contains $n = mi + 1$ vertices.

THEOREM 4 A full m -ary tree with

- (i) n vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,
- (ii) i internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,
- (iii) l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.

EXAMPLE 9 Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives more than one letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?



Solution: The chain letter can be represented using a 4-ary tree. The internal vertices correspond to people who sent out the letter, and the leaves correspond to people who did not send it out. Because 100 people did not send out the letter, the number of leaves in this rooted tree is $l = 100$. Hence, part (iii) of Theorem 4 shows that the number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$. Also, the number of internal vertices is $133 - 100 = 33$, so 33 people sent out the letter. ◀

FIGURE 13 A Rooted Tree.

BALANCED m -ARY TREES It is often desirable to use rooted trees that are “balanced” so that the subtrees at each vertex contain paths of approximately the same length. Some definitions will make this concept clear. The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices. In other words, the height of a rooted tree is the length of the longest path from the root to any vertex.

EXAMPLE 10 Find the level of each vertex in the rooted tree shown in Figure 13. What is the height of this tree?

Solution: The root a is at level 0. Vertices b , j , and k are at level 1. Vertices c , e , f , and l are at level 2. Vertices d , g , i , m , and n are at level 3. Finally, vertex h is at level 4. Because the largest level of any vertex is 4, this tree has height 4. ◀

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

EXAMPLE 11 Which of the rooted trees shown in Figure 14 are balanced?

Solution: T_1 is balanced, because all its leaves are at levels 3 and 4. However, T_2 is not balanced, because it has leaves at levels 2, 3, and 4. Finally, T_3 is balanced, because all its leaves are at level 3. ◀

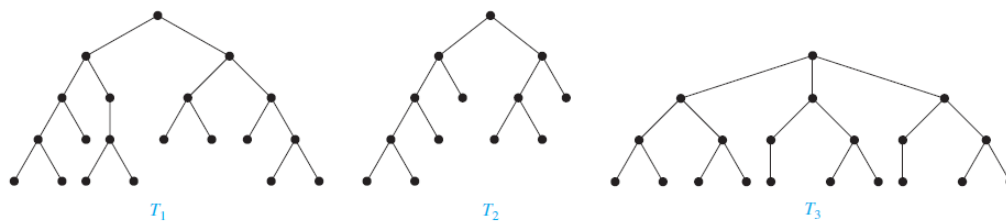


FIGURE 14 Some Rooted Trees.

A BOUND FOR THE NUMBER OF LEAVES IN AN m -ARY TREE It is often useful to have an upper bound for the number of leaves in an m -ary tree. Theorem 5 provides such a bound in terms of the height of the m -ary tree.

THEOREM 5
COROLLARY 1

There are at most m^h leaves in an m -ary tree of height h .

If an m -ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$. If the m -ary tree is full and balanced, then $h = \lceil \log_m l \rceil$. (We are using the ceiling function here. Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x .)

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 .

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree. We will give an example before proving this result.

EXAMPLE 1 Find a spanning tree of the simple graph G shown in Figure 2.

Solution: The graph G is connected, but it is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of G . Next remove the edge $\{e, f\}$ to eliminate a second simple circuit. Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of G . The sequence of edge removals used to produce the spanning tree is illustrated in Figure 3.

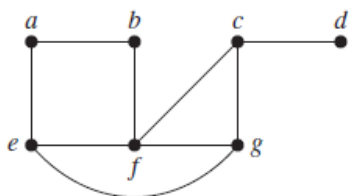


FIGURE 2 The Simple Graph G .

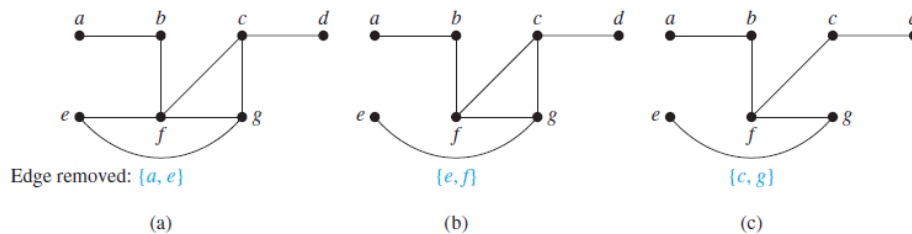


FIGURE 3 Producing a Spanning Tree for G by Removing Edges That Form Simple Circuits.

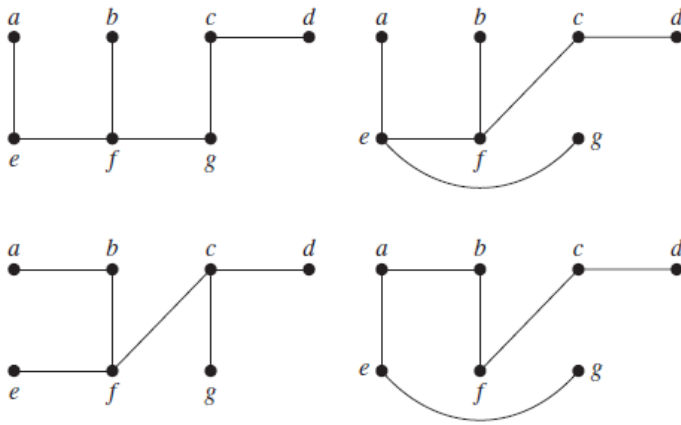


FIGURE 4 Spanning Trees of G .

A simple graph is connected if and only if it has a spanning tree.

We can build a spanning tree for a connected simple graph using **depth-first search**. We will form a rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree. Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex.

The reader should note the recursive nature of this procedure. Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available. However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called **backtracking**, because the algorithm returns to vertices previously visited to add paths. Example 3 illustrates backtracking.

EXAMPLE 3 Use depth-first search to find a spanning tree for the graph G shown in Figure 6.



Solution: The steps used by depth-first search to produce a spanning tree of G are shown in Figure 7. We arbitrarily start with the vertex f . A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible. This produces a path f, g, h, k, j (note that other paths could have been built). Next, backtrack to k . There is no path beginning at k containing vertices not already visited. So we backtrack to h . Form the path h, i . Then backtrack to h , and then to f . From f build the path f, d, e, c, a . Then backtrack to c and form the path c, b . This produces the spanning tree. ▶

The edges selected by depth-first search of a graph are called **tree edges**. All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges**.

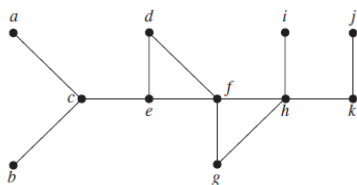


FIGURE 6 The Graph G .

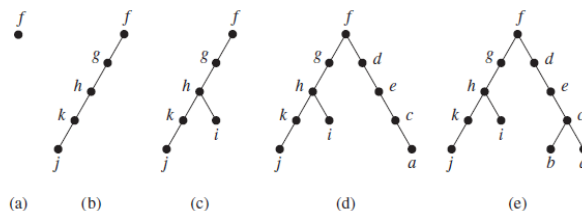

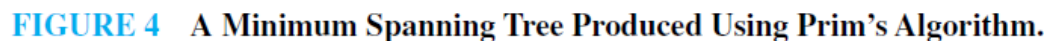
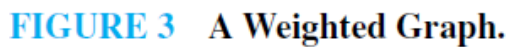


FIGURE 7 Depth-First Search of G .

The first algorithm that we will discuss was originally discovered by the Czech mathematician Vojtěch Jarník in 1930, who described it in a paper in an obscure Czech journal. The algorithm became well known when it was rediscovered in 1957 by Robert Prim. Because of this, it is known as **Prim's algorithm** (and sometimes as the **Prim-Jarník algorithm**). Begin by choosing any edge with smallest weight, putting it into the spanning tree. Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree, never forming a simple circuit with those edges already in the tree. Stop when $n - 1$ edges have been added.


Solution: A minimum spanning tree constructed using Prim's algorithm is shown in Figure 4. The successive edges chosen are displayed. 



Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after $n - 1$ edges have been selected.

The reader should note the difference between Prim's and Kruskal's algorithms. In Prim's algorithm edges of minimum weight that are incident to a vertex already in the tree, and not forming a circuit, are chosen; whereas in Kruskal's algorithm edges of minimum weight that are not necessarily incident to a vertex already in the tree, and that do not form a circuit, are chosen. Note that as in Prim's algorithm, if the edges are not ordered, there may be more than one choice for the edge to add at a stage of this procedure. Consequently, the edges need to be ordered for the procedure to be deterministic. Example 3 illustrates how Kruskal's algorithm is used.

EXAMPLE 3 Use Kruskal's algorithm to find a minimum spanning tree in the weighted graph shown in Figure 3.

Extra Examples  *Solution:* A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in Figure 5.

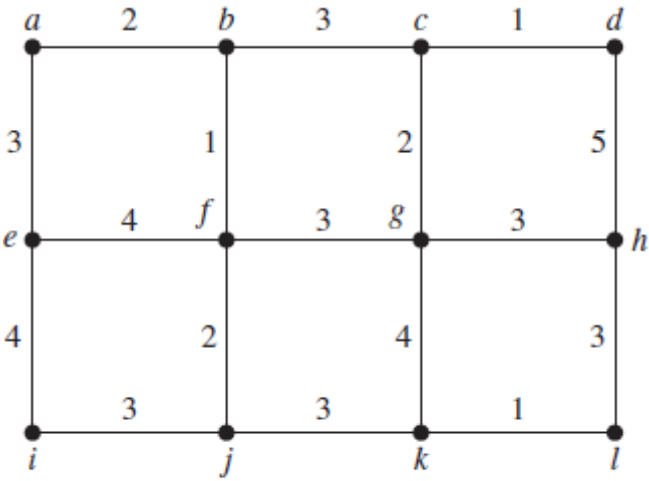


FIGURE 3 A Weighted Graph.

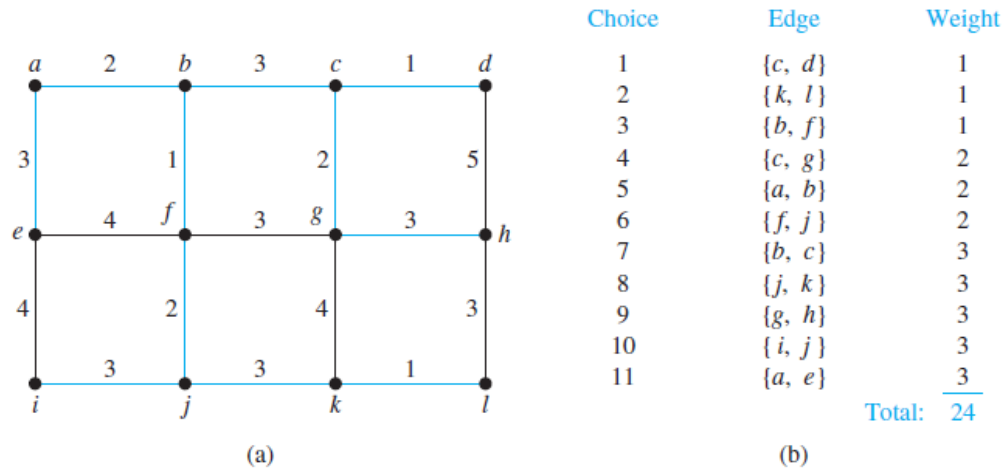


FIGURE 5 A Minimum Spanning Tree Produced by Kruskal's Algorithm.

Summary

In this week, we learned what tree is, what a forest is, what a rooted tree is? Alongside this, we explored some properties of trees, what a spanning tree and minimum-cost spanning trees are and the two greedy algorithms for finding minimum cost spanning trees; Kruskal's and Prim's Algorithm.