

Chapter 3: Trees

3.1 General Properties of Trees

Definition 3.1.1: A *tree* is a connected graph without cycles. A *forest* (or an *acyclic graph*) is a graph without cycles.

Definition 3.1.2: A vertex u of a graph is called a *leaf* or *pendant* if $\deg(u) = 1$. A vertex that is not a leaf is called a *node*.

Lemma 3.1.3: Let T be a tree with at least two vertices and let $P = u_0u_1 \cdots u_\ell$ be a longest path in T . Then both u_0 and u_ℓ are leaves.

Theorem 3.1.4: Every tree of order at least two contains at least two leaves.

Theorem 3.1.5: Suppose $S = (d_1, d_2, \dots, d_p)$ is a non-increasing sequence of positive integers. If $d_1 + d_2 + \cdots + d_p = 2(p - 1)$, then there exists a tree whose degree sequence is S .

Definition 3.1.6: Let G be a connected graph. If a tree T is a spanning subgraph of G , then T is called a *spanning tree* of G .

Lemma 3.1.7: Any connected graph G contains a spanning tree.

Corollary 3.1.8: For any graph, there is a spanning subgraph which is a forest. Such a forest is called a *spanning forest*.

Lemma 3.1.9: A graph G of order p with ω components has at least $p - \omega$ edges.

Corollary 3.1.10: Let G be a connected graph of order p . Then G contains at least $p - 1$ edges.

Lemma 3.1.11: Let P and Q be two distinct (u, v) -paths. Then the closed walk PQ^{-1} contains a cycle.

Lemma 3.1.12: If W is a closed walk of odd length, then it contains an odd cycle (cycle with odd order).

Theorem 3.1.13: If T is a simple graph on p vertices, then the following statements are equivalent:

- (1) T is a tree.
- (2) T has $p - 1$ edges and no cycles.
- (3) T has $p - 1$ edges and is connected.
- (4) T is connected and every edge is a bridge.
- (5) There is exactly one path between every pair of distinct vertices in T .
- (6) T has no cycles, but adding an edge to T between a pair of nonadjacent vertices creates exactly one cycle.

Theorem 3.1.14: A graph G is bipartite if and only if G contains no odd cycles.

Corollary 3.1.15: A forest is a bipartite graph.

Theorem 3.1.18: Let T be a tree and let n_i be the number of vertices of degree i . Then the number of leaves of a nontrivial tree is given by

$$n_1 = 2 + n_3 + 2n_4 + 3n_5 + \cdots = 2 + \sum_{i=3}^{\infty} (i-2)n_i.$$

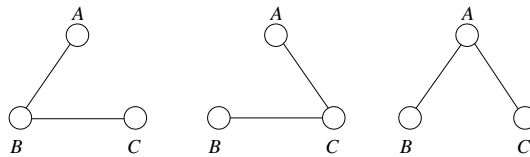
3.3 Number of Spanning Trees

Definition 3.3.1: A (p, q) -graph G is called a *labeled graph* if there is a bijection $f : V(G) \rightarrow S$, where $|S| = p$. The set S is called a *label set*.

Definition 3.3.2: Let G be a graph.

1. Recall that a subtree T of G is called a *spanning tree* of G if $V(T) = V(G)$.
2. Recall that a subforest F of G is called a *spanning forest* if for each component H of G , the subgraph $F \cap H$ is a spanning tree of H .
3. Suppose G is connected. For a fixed labeling of the vertices of G , the number of distinct spanning trees in G is denoted by $\tau(G)$. Hence, $\tau(G - e) = 0$ if e is a cut-edge.

Example 3.3.3: K_3 has three different spanning trees. They are



Proposition 3.3.4: Let G be a graph.

1. If G is a tree, then $\tau(G) = 1$.
2. If G_1, \dots, G_ω are the components of G , then the number of spanning forests of G is $\tau(G_1) \cdots \tau(G_\omega)$.

Remark 3.3.5: By the proposition above, we assume that G is connected. If G' is the graph obtained from G by removing all loops from G , then $\tau(G') = \tau(G)$. Therefore, we always assume that G is loopless when counting the number of spanning trees (or forests) of G .

Let G be a simple graph. A link e of G is said to be *contracted* if it is deleted and its ends are identified. The resulting graph is denoted by $G \cdot e$. The subgraph obtained from $G \cdot e$ by removing all loops is denoted by $G * e$. The underlying simple graph of $G \cdot e$ is called the *simple contraction* of G by e and denoted by G/e . From the remark above, we have $\tau(G \cdot e) = \tau(G * e)$.

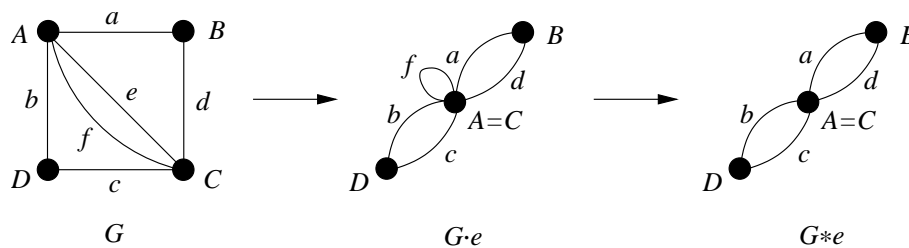


Figure 1: Before and after the contraction of e .

It is easy to see that

$$\begin{aligned} p(G \cdot e) &= p(G) - 1; \\ q(G \cdot e) &= q(G) - 1; \\ \omega(G \cdot e) &= \omega(G). \end{aligned}$$

Proposition 3.3.6: Suppose T is a tree. For any $e \in E(T)$, the contracted graph $T \cdot e$ is still a tree.

Theorem 3.3.7: Let G be a graph and $e \in E(G)$, where e is not a loop. Then $\tau(G) = \tau(G \cdot e) + \tau(G - e)$.

Example 3.3.8: To calculation $\tau(G)$, we often use the graph itself to represent the number of spanning trees. Also, if a graph contains some loops, we may delete the loops from that graph before computing the number of spanning trees (but we did not do this in the following example). In each step, we always choose an edge e that is not a bridge.

$$\begin{aligned} \tau(G) &= \begin{array}{c} \text{[Diagram: Square with diagonal and top edge } e \text{]} \\ = \text{[Diagram: Square with diagonal and left edge } e \text{]} + \text{[Diagram: Square with diagonal and right edge } e \text{]} \\ = \left(\text{[Diagram: Square with diagonal]} + \text{[Diagram: Square with } e \text{]} \right) + \left(\text{[Diagram: Square with diagonal and } e \text{]} + \text{[Diagram: Square with } e \text{]} \right) \\ = 1 + 2 \left(\text{[Diagram: Square with } e \text{]} \right) + \left(\text{[Diagram: Square with diagonal and } e \text{]} + \text{[Diagram: Square with } e \text{]} \right) \\ = 1 + 2 \left(\text{[Diagram: Square with } e \text{]} + \text{[Diagram: Square with } e \text{]} \right) + \left(\text{[Diagram: Square with diagonal and } e \text{]} + \text{[Diagram: Square with } e \text{]} \right) + K_1 \\ = 1 + 2(1 + 1) + (1 + 1) + 1 = 8. \end{array} \end{aligned}$$

To count the number of spanning trees in K_p is the same as computing the number of ways to connect p labeled vertices by a tree. That is, the number of labeled trees of order p . A simple formula for $\tau(K_p)$ was found by the English mathematician A. Cayley in 1889* when he wanted to count the number of chemical molecules with formula C_nH_{2n+2} . He is the first person, in 1857, to use the term ‘tree’ in graph theory. Afterwards, the German mathematician Ernst Paul Heinz Prüfer gave a simple proof on this theorem in 1918†.

The idea of Prüfer is to construct a bijection between the set of labeled trees with p vertices and the set of all sequences (called *Prüfer code* or *Prüfer sequence*) of the form (a_1, \dots, a_{p-2}) , where $a_i \in \{1, 2, \dots, p\}$.

Theorem 3.3.9 (Cayley): The complete graph K_p has p^{p-2} different spanning trees for $p \geq 1$.

Algorithm: Trees to Prüfer Codes

Given a tree T on p vertices such that $V(T) = \{1, 2, \dots, p\}$. We want to construct the Prüfer code (a_1, \dots, a_{p-2}) of T and a sister code (b_1, \dots, b_{p-2}) , where $a_i \in \{1, \dots, p\}$ and $b_i \in \{1, \dots, p - 1\}$.

Step 1. Let $i = 1$.

Step 2. Choose a leaf from the tree with the smallest label b_i . Let a_i be the unique neighbor of b_i .

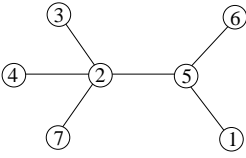
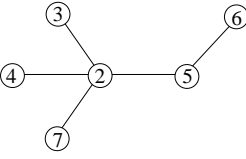
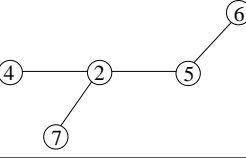
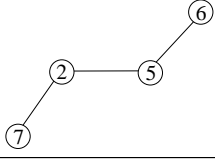
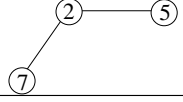

*A. Cayley, A theorem on trees, *Quart. J. Math.*, **23** (1889), 376-378.

†H. Prüfer, Neuer Beweis eines Satzes über Permutation, *Arch. Math. Phys.*, **27** (1918), 742-744.

Step 3. Remove the leaf b_i , leaving a smaller tree.

Step 4. If there are only two vertices left, stop. Otherwise, let $i := i + 1$ and go back to Step 2.

Example 3.3.10: Consider the labeled tree

labeled tree	i	b_i	a_i
	1	1	5
	2	3	2
	3	4	2
	4	6	5
	5	5	2
			

The Prüfer sequence of the tree is $(5, 2, 2, 5, 2)$.

Notice that for any nontrivial tree, the graph K_2 is obtained when the algorithm terminates. At each stage of the algorithm, vertex p is never deleted.

Algorithm: Prüfer Codes to Trees

Given a Prüfer code (a_1, \dots, a_{p-2}) . Firstly, let $a_{p-1} = p$. Thus we have the augmented sequence $(a_1, \dots, a_{p-2}, a_{p-1})$. If we know the sister sequence $(b_1, \dots, b_{p-2}, b_{p-1})$, then we can construct the tree because the $p - 1$ edges $a_1b_1, \dots, a_{p-1}b_{p-1}$ were fixed.

Step 1. Let $i = 1$ and let $B = \{1, 2, \dots, p - 1\}$.

Step 2. Let $b_i \in B$ be the smallest number not in (a_i, \dots, a_{p-1}) .

Step 3. Remove b_i from B .

Step 4. If $i = p - 1$, stop. Otherwise let $i := i + 1$ and go back to Step 2.

Example 3.3.11: Consider the Prüfer sequence $(5, 2, 2, 5, 2)$. Since the length of the sequence is 5, the tree has 7 vertices. The augmented sequence is $(5, 2, 2, 5, 2, 7)$.

i	augmented sequence	b_i	B
1	(5, 2, 2, 5, 2, 7)	1	2, 3, 4, 5, 6
2	(2, 2, 5, 2, 7)	3	2, 4, 5, 6
3	(2, 5, 2, 7)	4	2, 5, 6
4	(5, 2, 7)	6	2, 5
5	(2, 7)	5	2
6	(7)	2	

3.2 Rooted Trees

A *rooted tree* is a tree T in which a special vertex r is singled out, which is called the *root* of the tree. Such a rooted tree is denoted by (T, r) .

Let (T, r) be a rooted tree. For any $u \in V(T)$, let P_u be the unique path between r and u (see Theorem 3.1.13). Then

1. The *parent* of u is the neighbor of u in P_u and denoted by $\text{Par}(u)$.
2. Any neighbor of u other than $\text{Par}(u)$ is called a *child* of u . The set of children of u is denoted by $\text{Chi}(u)$.
3. Two vertices are called *siblings* if they have the same parent.
4. Any vertex on P_u other than u is called an *ancestor* of u . The set of ancestors of a vertex u is denoted by $\text{Anc}(u)$.
5. A vertex v that has u as an ancestor is called a *descendant* of u . The set of descendants of a vertex u is denoted by $\text{Des}(u)$.

Definition 3.2.1: A rooted tree (T, r) in which the left/right order of every set of siblings is specified is called an *ordered rooted tree*.

Definition 3.2.2: A *binary tree* is an ordered rooted tree in which each vertex has at most two children. Each child of a vertex is called either the *left child* or the *right child*. A subtree rooted at the left (right) child of vertex u is known as u 's *left (right) subtree*. By convention, the single vertex tree is considered a "trivial" binary tree.

Definition 3.2.3: A *regular* or *full* binary tree is a binary tree that is trivial or satisfies the following conditions:

1. There is exactly one vertex of degree two, namely the root.
2. All vertices other than the root have degree one or three.

Definition 3.2.4: Let (T, r) be a rooted tree. For $u \in V(T)$, the distance $d_T(u, r)$ is called the *level* of u in T and denoted by $\ell_T(u)$ or $\ell(u)$. The maximum level in (T, r) is called the *height* of (T, r) . A rooted tree of height k is called a $(k + 1)$ -*level tree*.

Definition 3.2.5: A binary tree T of height k is called *complete* if for each level $i \in \{0, 1, \dots, k - 1\}$ of T have precisely 2^i vertices, in which they are called *full levels*.

Proposition 3.2.6: For any regular binary tree T on p vertices, the height $ht(T)$ satisfies

$$\lceil \log_2(p + 1) - 1 \rceil \leq ht(T) \leq \frac{p - 1}{2},$$

where $\lceil x \rceil$ denotes the least integer not less than x .

3.4 Searching Trees

Depth-first Search

The idea of DFS is to start at a designated source vertex (called root) and explores a path in the graph as far as possible. When there is no edge to go, then backtracks one level and tries again to go deeper in another route. The process repeats until all vertices have been visited. Suppose a connected graph is given. Applying the above exploration, it will obtain a spanning tree T of the graph and each edge of T will be traversed exactly twice. Such a tree is called a *depth-first search tree* (or *DFS tree*).

Example 3.4.1: Consider the following tree. We start from a and want to visit all vertices at least once.

Firstly, we imagine that we stand at vertex a . We can see that there are two ways to go, one is vertex b and the other is vertex c . We arbitrary choose one way, say b . Then we have two choices again: go to d or e . Suppose we choose d , then we have three choices: go to i , j or k . Suppose we choose i , then there is no way to go. Here we backtrack one level and back to vertex d . Vertex i has been visited, hence we have two choices: go to j or k . Repeat this process to choose vertex one by one. Finally, the graph in Fig. 2 will be obtained, where the numbers indicated in the figure are the order of this visit. This order is called a *preorder traversal*.

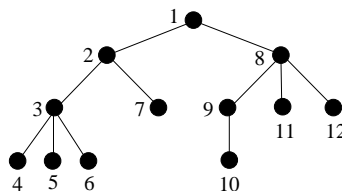
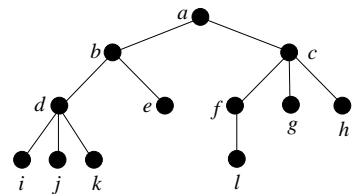
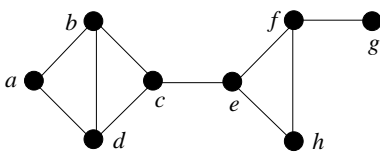


Figure 2: Preorder traversal of DFS.

Example 3.4.2: We want to find a spanning tree for the given graph by applying depth-first search. In order to find the spanning tree, we have to choose some edges to keep the subgraph connected and acyclic.



Suppose we start from a . We may choose b or d to visit. If we choose b , then we can choose d or c from b . Suppose we choose to visit c and then choose to visit d . At this moment, we cannot visit a or b because it will create a cycle and so there is no way to go further. We have to

backtracks to c and choose e and so on. Finally, we obtain Fig. 3, where the numbers in the figure represent the preorder traversal of the DFS spanning tree. The edges indicated by arrows are edges of the DFS spanning tree.

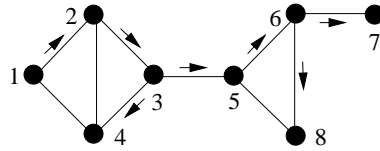


Figure 3: A DFS spanning tree.

We may use a tabular form to present the search.

DFS Spanning Tree Algorithm:

- Step 1. Choose v as the root and $R = \emptyset$. Assign $l(v) = (1, *)$; $U = V(G) \setminus \{v\}$ (keeps track unlabeled vertices); $b^* = v$ (current vertex) and $i = 2$.
- Step 2. If $N(b^*) \cap U = \emptyset$, then go to Step 3. If not, then choose $w \in N(b^*) \cap U$ (you may pre-order the vertices first). Put b^*w in R ; let $l(w) = (i, b^*)$ and remove w from U . Let $b^* = w$ and add one to i . Return to Step 2.
- Step 3. Let b^* (a new one) be the second coordinate of $l(b^*)$ (backtrack).
- Step 4. If $b^* = v$ and $N(b^*) \cap U = \emptyset$, stop. A spanning tree of the component that contains the root v has been found. Otherwise, repeat Step 2. If $U = \emptyset$, then the tree found is a spanning tree of G . If $U \neq \emptyset$, then G is disconnected.

Example 3.4.3: Given a graph G and its adjacency lists:

Vertex v	$N(v)$
a	b, e, f, g
b	a
c	d, g
d	c, e, g
e	a, d, f
f	a, e, g
g	a, c, d, f

Use d, g, c, b, a, f, e as the pre-ordering and we take d as the root.

Pass	Step	$N(b^*) \cap U$	w	R \uparrow b^*w	$l(w)$	U	b^*	i
0	1			\emptyset	$(1, *)$	g, c, b, a, f, e	d	2
1	2	g, c, e	g	dg	$(2, d)$	c, b, a, f, e	g	3
2	2	c, a, f	c	gc	$(3, g)$	b, a, f, e	c	4
3	2#, 3	\emptyset					g	
4	2	a, f	a	ga	$(4, g)$	b, f, e	a	5
5	2	b, f, e	b	ab	$(5, a)$	f, e	b	6
6	2#, 3	\emptyset					a	
7	2	f, e	f	af	$(6, a)$	e	f	7
8	2	e	e	fe	$(7, f)$	\emptyset		

Recall the Robbins Theorem:

Theorem 2.5.10: A graph G has a strong orientation if and only if it is connected and has no bridges. By using the Hopcroft-Tarjan Algorithm on an orientable graph, we can find a strong orientation on it.

Hopcroft-Tarjan Algorithm: Given a connected graph G without bridge.

Step 1. Obtain a DFS spanning tree T of G by using DFS Spanning Tree Algorithm.

Step 2. Orient each edge of T towards the vertex with higher number.

Step 3. Orient each of the remaining edges of G towards the vertex with lower number.

Example 3.4.4: Given a graph G and its adjacency lists:

Vertex v	$N(v)$
a	b, e, f, g
b	a, c
c	d, g
d	c, e, g
e	a, d, f
f	a, e, g
g	a, c, d, f

Clearly the graph (see Fig. 4A) contains no bridges. If we use d, g, c, b, a, f, e as the pre-ordering and take d as the root, then applying the DFS Spanning Tree Algorithm, we have a DFS spanning tree (see Fig. 4B). Moreover, apply the Hopcroft-Tarjan Algorithm will obtain an orientation of the graph (see Fig. 4C).

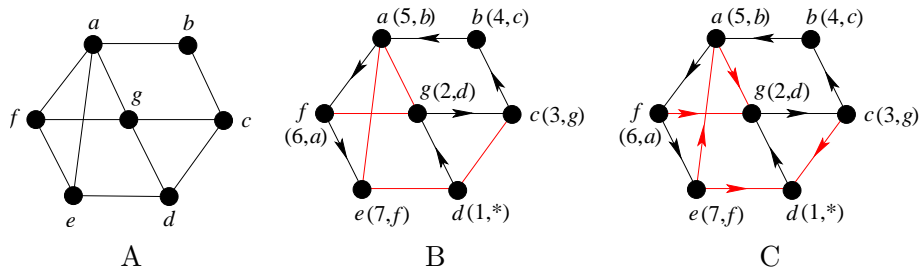


Figure 4: An orientation of a graph.

If we use d, g, a, b, f, c, e as the pre-ordering and take d as the root, then we obtain

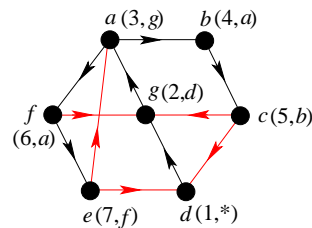


Figure 5: An other orientation of a graph.

Breadth-first Search

The idea of BFS is to start at the root and then explores the neighbors of the original vertex. For each of those neighbors, we investigate them one by one until all vertices have been visited. Suppose a connected graph is given. Apply the above exploration, we will obtain a spanning tree of the graph. Such a tree is called a *breadth-first search tree* (or *BFS tree*).

Example 3.4.5: We apply breadth-first search on the tree in Example 3.4.1. Starting from a , we can go to b or c . Suppose we choose to go to b and then c . Afterwards, jump back to b and we can go to d or e . Suppose we go to d first and then e . After that, we go back to c , and then visit f, g and h . Repeating this process we have Fig. 6, where the numbers are the order of the search.

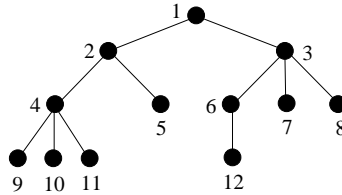


Figure 6: Preorder traversal of BFS.

Example 3.4.6: We find a spanning tree for the graph in Example 3.4.2 by using BFS. Starting from a we go to b and then d . That is, we choose the edges ab and ad . Now we jump back to b and we can only choose the vertex c . That is, we choose the edge bc . We cannot choose d because it will create a cycle. Due to the search order, we jump to d but there is no way to go. Hence we jump to c . Repeat this process we have Fig. 7, which is a BFS spanning tree.

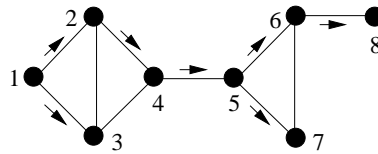


Figure 7: A BFS spanning tree.

3.5 Minimum Spanning Trees

Definition 3.5.1: Let G be a graph and let $W : E(G) \rightarrow \mathbb{R}$.

1. The ordered tuple (G, W) is called a *weighted graph* and the function W is called a *weight function* of G .
2. If $H \subseteq G$, then $W(H) = \sum_{e \in E(H)} W(e)$ is called the *weight* of H in G .

Note that, if two vertices u and v in a weighted graph (G, W) are joined by more than one edge, then we may delete the edges with larger weights. Also, we may delete all loops (if any) of the graph, which does not affect the optimal tree. Moreover, if two vertices u and v in a weighted graph (G, W) are not adjacent, then any spanning tree does not contain the edge uv . Hence, if we add the edge uv in G and define $W(uv) = \infty$, then it does not affect the optimal tree too. Therefore, we may assume all the weighted graph being considered is a weighted complete graph and the codomain of the weight function is $\mathbb{R} \cup \{\infty\}$.

Definition 3.5.2: Let (G, W) be a weighted complete graph of order p . We define a $p \times p$ matrix C , whose row and column are named by the vertices of G as follows: For $u, v \in V(G)$, the (u, v) -entry of C is defined by $W(uv)$. Such a matrix is called the *cost matrix* of (G, W) .

Definition 3.5.3: For a connected weighted graph (G, W) , a spanning tree T with the minimum weight $W(T)$ is called a *minimum spanning tree* or *optimal tree*.

Two algorithms for finding minimum spanning tree will be introduced, which are Kruskal's algorithm and Prim's algorithm. The following algorithm, which is based on BFS, was proposed by Joseph B. Jr. Kruskal[‡]. He was a student of Roger C. Lyndon and Paul Erdős at Princeton University.

Kruskal's Algorithm: Given a weighted (simple) graph G of order p . Let L be a list of edges that have been pre-sorted by weight in ascending order.

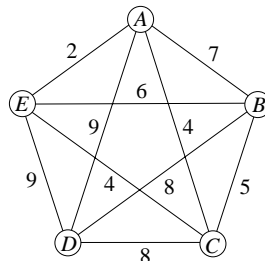
Step 1. $S = \emptyset$.

Step 2. Let e be the next edge on the sorted list L with $e \notin S$ and the edge-induced subgraph $G[S \cup \{e\}]$ is acyclic. Let $S := S \cup \{e\}$.

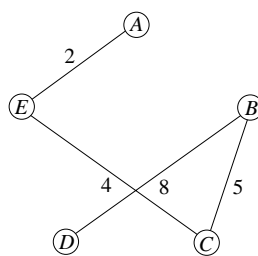
Step 3. If $|S| = p - 1$, stop. Otherwise, return to Step 2.

Theorem 3.5.4: Let G be a weighted connected graph and let T be a subgraph of G obtained by Kruskal's algorithm. Then T is a minimal spanning tree of G .

Example 3.5.5: Consider the graph

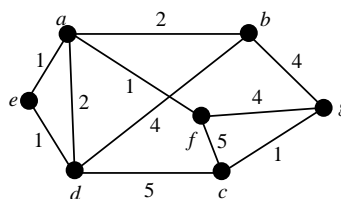


According to Kruskal's algorithm we choose AE first. And then we choose AC or CE . Suppose we choose CE , then we cannot choose AC because it will create a triangle. Hence we choose BC . If we choose BE , then it creates a triangle. The next edge with smallest weight is AB . However, if we choose it, then a 4-cycle is created and thus we can only choose BD or CD . If we choose BD , then we obtain the following graph.



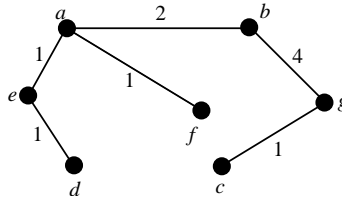
Its weight is $2 + 4 + 5 + 8 = 19$.

Example 3.5.6: Apply Kruskal's Algorithm to the following weighted graph.



[‡]J.B. Jr. Kruskal, On the shortest spanning subtree of a graph and travelling salesman problem, *Proc. Amer. Math. Soc.*, **7** (1956), 48-50.

We will get a minimal spanning tree



R.C. Prim modified Kruskal's algorithm and provided the following algorithm[§].

Prim's Algorithm Given a weighted (simple) graph G of order p .

Step 1. Select a vertex v and let $V = \{v\}$, $E = \emptyset$.

Step 2. Among all $u \notin V$, let $e = uw$ be a minimum weight edge, where $w \in V$. Let $V := V \cup \{u\}$ and $E := E \cup \{uw\}$.

Step 3. If $|E| = p - 1$, then stop. Otherwise, return to Step 2.

Theorem 3.5.7: *Let G be a weighted connected graph and let T be a subgraph of G obtained by Prim's algorithm. Then T is a minimal spanning tree of G .*

The Prim's algorithm usually apply in tabular form. Consider Example 3.5.5 again.

Example 3.5.8: We start from the cost matrix.

	A	B	C	D	E
A	*	7	4	9	2
B	7	*	5	8	6
C	4	5	*	8	4
D	9	8	8	*	9
E	2	6	4	9	*

Suppose we choose B first. We have to check the vertices that are adjacent to B . Which means we have to look at the numbers in the column B . Since the other end point of the considered edges is not the vertex B , we need not look at the row B . Thus we delete the row B from the matrix.

	A	<u>B</u>	C	D	E
A	*	7	4	9	2
C	4	5	*	8	4
D	9	8	8	*	9
E	2	6	4	9	*

The underlined vertex (now is B) is the end point(s) of the chosen edge. The boxed number indicate the chosen edge in the following step.

Hence we choose the edge BC . Since the next chosen edge must be start at B or C , we only look at columns B and C as well as the other end of such edge is neither B nor C . Therefore, row C may be deleted.

	A	<u>B</u>	<u>C</u>	D	E
A	*	7	4	9	2
D	9	8	8	*	9
E	2	6	4	9	*

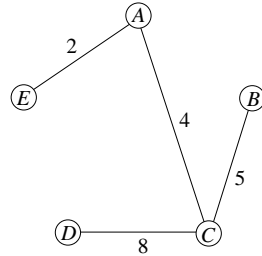
[§]R.C. Prim, Shortest Connection Networks and Some Generalization, *Bell System Tech. J.*, **36** (1957), 1389-1401.

Repeat the above process, we have

	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>
<u>D</u>	9	8	8	*	9
<u>E</u>	2	6	4	9	*

	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>
<u>D</u>	9	8	8	*	9

Finally we have



which is the minimal spanning tree (with weight 19) of G .

Notice that it is different from the spanning tree obtained in Example 3.5.5. It shows that minimal spanning tree is not unique.

Example 3.5.9: Consider the Example 3.5.6 again. Suppose we start from a . We have

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>b</u>	2	*	∞	4	∞	∞	4	
<u>c</u>	∞	∞	*	5	∞	5	1	
<u>d</u>	2	4	5	*	1	∞	∞	<i>ae</i>
<u>e</u>	1	∞	∞	1	*	∞	∞	
<u>f</u>	1	∞	5	∞	∞	*	4	
<u>g</u>	∞	4	1	∞	∞	4	*	

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>b</u>	2	*	∞	4	∞	∞	4	
<u>c</u>	∞	∞	*	5	∞	5	1	
<u>d</u>	2	4	5	*	1	∞	∞	<i>af</i>
<u>f</u>	1	∞	5	∞	∞	*	4	
<u>g</u>	∞	4	1	∞	∞	4	*	

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>b</u>	2	*	∞	4	∞	∞	4	
<u>c</u>	∞	∞	*	5	∞	5	1	
<u>d</u>	2	4	5	*	1	∞	∞	<i>ed</i>
<u>g</u>	∞	4	1	∞	∞	4	*	

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>b</u>	2	*	∞	4	∞	∞	4	
<u>c</u>	∞	∞	*	5	∞	5	1	<i>ab</i>
<u>g</u>	∞	4	1	∞	∞	4	*	

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>c</u>	∞	∞	*	5	∞	5	1	<i>bg</i>
<u>g</u>	∞	4	1	∞	∞	4	*	

	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>f</u>	<u>g</u>	
<u>c</u>	∞	∞	*	5	∞	5	1	<i>gc</i>

We get the same result as Example 3.5.6.