Chapter 2

Trees and Connectivity

2.1 Definitions and Simple Properties

A graph G is called acyclic if it contains no cycles.

Since loops are cycles of length one while a pair of parallel edges produces a cycle of length two, any acyclic graph must be simple.

A graph G is called a tree if it is a connected acyclic graph.

Figure 2.1 shows all trees with at most five vertices.

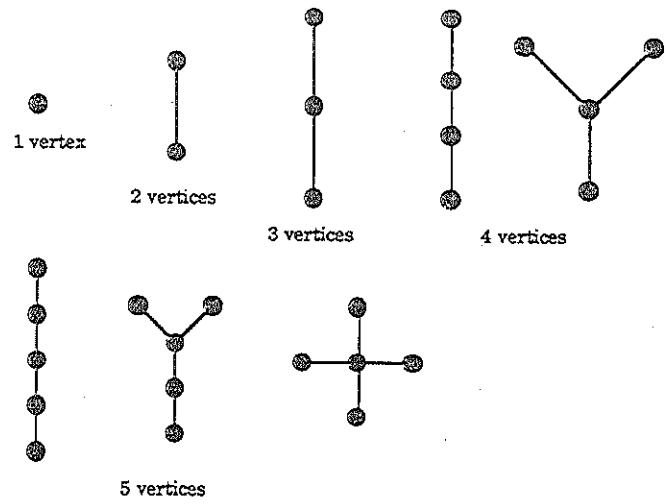

Figure 2.1: The trees with at most five vertices.

Figure 2.2 shows all trees with six vertices while Figure 2.3 shows two more luxuriant trees.

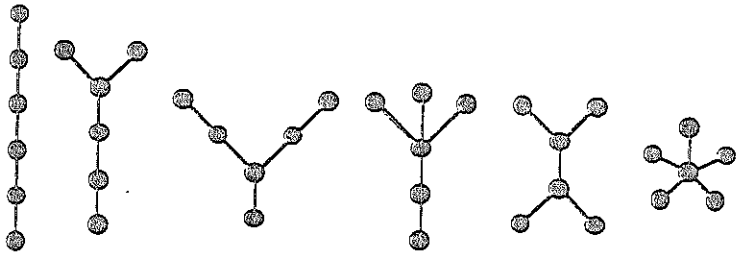

Figure 2.2: The trees with six vertices.

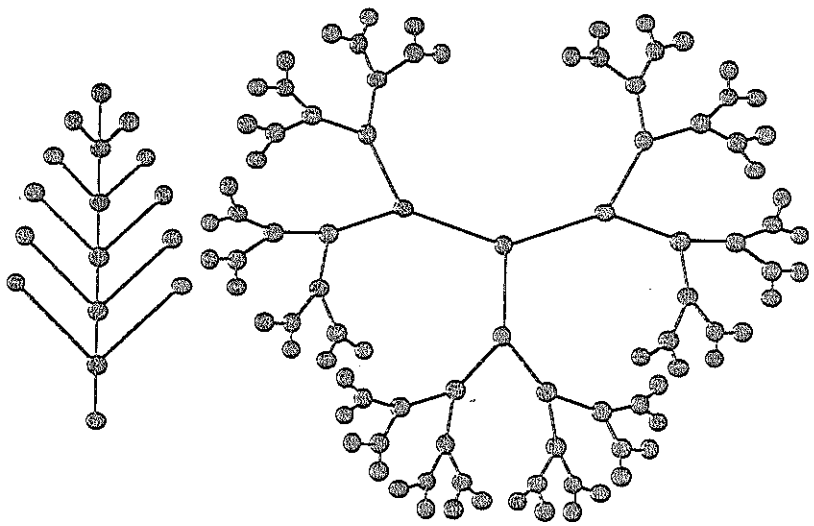

Figure 2.3: Two fancier trees.

Part (a) of the first result in this chapter gives an important property of trees, while part (b) shows that this property almost characterises trees.

Theorem 2.1 *(a) Let u and v be distinct vertices of a tree T. Then there is precisely one path from u to v.*

(b) Let G be a graph without any loops. If for every pair of distinct vertices u and v of G there is precisely one path from u to v, then G is a tree.

Proof (a) Suppose that the result is false. Then there are two different paths from u to v, say $P = u_0u_1u_2 \dots u_mv$ and $P' = v_0v_1v_2 \dots v_nv$ where $u_0 = u = v_0$. There is then a first index k greater than 0 for which $u_k \neq v_k$. Let $x = u_{k-1} = v_{k-1}$ and let w be the first vertex after x which belongs to both P and P'. (The vertex x might well be u and the vertex w might well be v but at least there are such vertices x and w.) Then $w = u_i = v_j$ for some indices i and j both greater than $k-1$. (See Figure 2.4.)

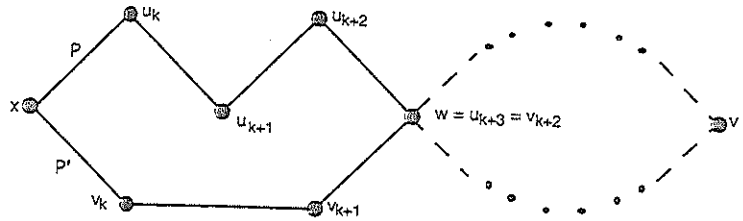

Figure 2.4

This produces a cycle $C = xu_{k+1} \dots u_iv_{j-1} \dots v_{k+1}x$ since no two of the "u" terms are repeated (as P is a path), no two of the "v" terms are repeated (as P' is a path), and, by the definition of w, no "u" term in C can be a "v" term. Since T is a tree, T has no cycles. This contradiction means that our initial assumption (the first sentence of the proof) must be false. Thus there is precisely one path from u to v.

(b) Since there is a path between each pair of its vertices, G must be connected. Thus it remains to show that G has no cycles. Firstly, since G has no loops it has no cycles of length one. Now suppose that G has a cycle of length at least two, say $C = v_1v_2 \dots v_nv_1$ where $n \geq 2$. Since a cycle is a trail, the edge v_nv_1 does not appear in the path $v_1v_2 \dots v_n$. Thus $P = v_1v_n$ and $P' = v_1v_2 \dots v_n$ are two different paths from v_1 to v_n. (See Figure 2.5.) This contradicts our assumptions. Hence G has no cycles and so is a tree. □

The next result shows that, with one exception, all trees have at least two "leaves".

Theorem 2.2 *Let T be a tree with at least two vertices and let $P = u_0u_1 \dots u_n$ be a longest path in T (so that there is no path in T of length greater than n). Then both u_0 and u_n have degree 1, i.e., $d(u_0) = 1 = d(u_n)$.*

Proof Suppose that $d(u_0) > 1$. The edge $f = u_0u_1$ contributes 1 to the degree of u_0 and so there must be another edge e from u_0 to a vertex v of T (which is different

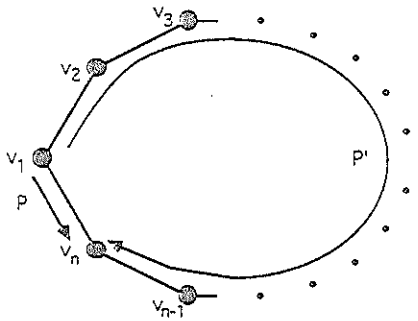

Figure 2.5

from f). If this vertex v is one of the vertices of the path P then we may set $v = u_i$ for some $i = 0, 1, \ldots, n$ and this produces a cycle $C = u_0u_1 \ldots u_iu_0$ (the last edge being e). Since T is a tree it has no cycles and so this is a contradiction. Thus the remaining possibility for v is that it is not one of the vertices of the path P. But then $P_1 = vu_0u_1 \ldots u_n$, where the first edge is e, gives a path of length $n+1$ in T, in contradiction to our assumption that P is a longest path (and of length n). This final contradiction shows that there is no such edge e and so $d(u_0) = 1$, as required. Similarly $d(u_n) = 1$, as required. □

Corollary 2.3 *A tree with at least two vertices has more than one vertex of degree 1.*

Proof In such a tree T there is a longest path P (of length greater than 0) and so the Theorem produces at least two vertices of degree 1. □

Note that in Figure 2.2 all our trees with 6 vertices had 5 edges. We now use Corollary 2.3 to prove a more general result:

Theorem 2.4 *If T is a tree with n vertices then it has precisely $n-1$ edges.*

Proof We use induction on n. When $n = 1$, i.e., T has only 1 vertex, then, since it has no loops, T can not have any edges, i.e., it has $n - 1 = 0$ edges. This establishes that the result is true for $n = 1$.

Now suppose that the result is true for $n = k$ where k is some positive integer. We use this to show that the result is true for $n = k+1$. Let T be a tree with $k+1$ vertices and let u be a vertex of degree 1 in T. (Note that such a vertex exists by Corollary 2.3.) Let $e = uv$ denote the unique edge of T which has u as an end. Then if x and y are vertices in T both different from u, any path P joining x to y does not go through the vertex u since if it did it would involve the edge e twice. Thus the subgraph $T-u$, obtained from T by deleting the vertex u (and the edge e), is connected. Moreover if C is a cycle in $T - u$ then C would be a cycle in T — impossible, since T is a tree. Thus the subgraph $T - u$ is also acyclic. Hence $T - u$ is a tree. However $T - u$ has k vertices (since T has $k + 1$) and so, by our induction assumption, $T - u$ has $k - 1$ edges. Since $T - u$ has exactly 1 edge less than T (the edge e), it follows that T has k edges, as required. In other words, assuming the result is true for k, we have shown that it is true for $k + 1$. Thus, by the principle of mathematical induction, it is true for all positive integers k. □

Let G be an acyclic graph. Then any subgraph of G must also contain no cycles. In particular, the connected components of G are also acyclic and so they are trees. For this reason an acyclic graph is also called a forest. Figure 2.6 gives an example of a forest.

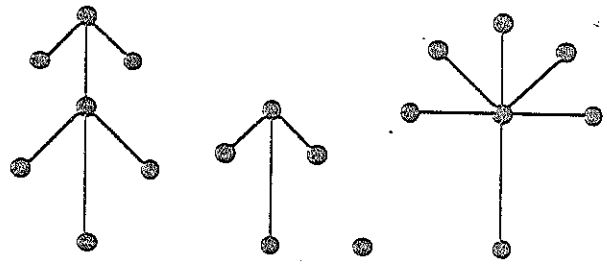

Figure 2.6: A (black) forest.

The next result extends Theorem 2.4, a result on trees, to cover forests.

Theorem 2.5 *Let G be an acyclic graph with n vertices and k connected components, i.e., $\omega(G) = k$. Then G has $n - k$ edges.*

Proof Denote the k components of G by $C_1, \ldots, C_k$ and suppose that for each $i, 1 \leq i \leq k$, the ith component C_i has n_i vertices. Then $n = n_1 + n_2 + \cdots + n_k$. Also, since each C_i is a tree, by Theorem 2.4 it has $n_i - 1$ edges and so since each edge of G belongs to precisely one component of G the total number of edges in G is $(n_1 - 1) + (n_2 - 1) + \cdots + (n_k - 1)$. Thus G has $(n_1 + n_2 + \cdots + n_k) - k$ edges, i.e., $n - k$ edges, as required. □

Exercises for Section 2.1

2.1.1 Give a list of all trees with 7 vertices and all trees with 8 vertices. Your lists should not contain any pair of trees which are isomorphic. (There are 11 non-isomorphic trees on 7 vertices and 23 non-isomorphic trees on 8 vertices.)

2.1.2 The complete bipartite graphs $K_{1,n}$, known as the **star graphs**, are trees. Figure 2.7 shows the star graphs $K_{1,4}$ and $K_{1,6}$. Prove that the star graphs are the only complete bipartite graphs which are trees.

2.1.3 Prove that any tree with at least two vertices is a bipartite graph.

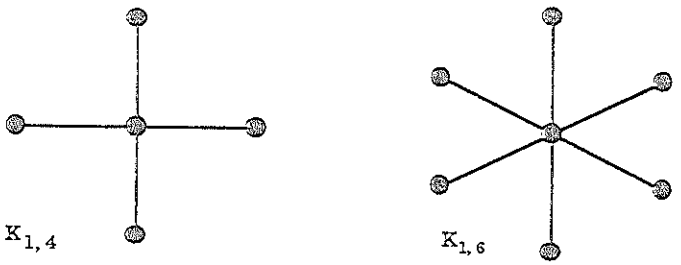

Figure 2.7: The star graphs $K_{1,4}$ and $K_{1,6}$.

2.1.4 Let T be a tree and let u and v be two vertices of T which are not adjacent. Let G be the supergraph of T obtained from T by joining u and v by an edge. Prove that G contains a cycle.

2.1.5 Let F be a forest and let u and v be two nonadjacent vertices of F. Let G be the supergraph of F obtained from F by joining u and v by an edge. Prove that G is a forest if and only if u and v belong to different components of F.

2.1.6 Let T be a tree with n vertices, where $n \geq 4$, and let v be a vertex of maximum degree in T.

(a) Show that T is a path if and only if $d(v) = 2$.

(b) Prove that T is isomorphic to a star graph (see Exercise 2.1.2 above) if and only if $d(v) = n - 1$.

(c) Prove that if $d(v) = n-2$ then any other tree with n vertices and maximum vertex degree $n - 2$ is isomorphic to T.

(d) Prove that if $n \geq 6$ and $d(v) = n-3$ then there are exactly 3 non-isomorphic trees which T can be.

2.1.7 Let T be a tree with n vertices, where $n \geq 3$. Show that there is a vertex v in T with $d(v) \geq 2$ such that every vertex adjacent to v, except possibly for one, has degree 1.

2.1.8 Let T be a tree and let v be a vertex of maximum degree in T, say $d(v) = k$. Prove that T has at least k vertices of degree 1.

2.2 Bridges

The first theorem of this section shows that if we remove an edge from a graph then the number of connected components of the graph either remains unchanged or it increases by exactly 1. Recall that $\omega(G)$ denotes the number of connected components of G.

Theorem 2.6 *Let e be an edge of the graph G and, as usual, let $G-e$ be the subgraph obtained by deleting e. Then $\omega(G) \leq \omega(G-e) \leq \omega(G)+1$.*

Proof Let e have end vertices u and v and let C be the connected component of G to which e (and u and v) belong. First suppose that u and v are distinct. Then the edge e forms, by itself, a path from u to v.

Suppose now that there is another path P from u to v. Then P can not involve the edge e and so must also be a path in the subgraph $G-e$. Thus, in this case, u and v are connected in $G-e$ (as well as in G). Moreover if x and y are any two vertices in the component C then there is still a walk from x to y in $G-e$ (since given a walk from x to y in G, we can simply replace any occurrence of the edge e in this walk by the path P to produce a walk in $G-e$). This shows that in this case the vertices of C still give one connected component in the subgraph $G-e$. Since any other connected component of G is unaffected by the removal of e, it follows that $G-e$ has the same number of components as G, i.e., $\omega(G-e) = \omega(G)$.

We now consider the remaining possibility,namely that the edge e gives the only path from u to v. Let x be any vertex in the component C. Then either

(i) there is a path from x to u which does not involve e or

(ii) every path from x to u must involve e.

If (i) holds then x is in the same component of $G-e$ as u, because the deletion of e does not affect such a path.

If (ii) holds then in any path P from x to u, since it involves e, e must in fact be the last edge in the path and so the second last vertex must be v. This then produces a path from x to v which does not involve e and so x is in the same component of $G-e$ as v. Moreover since e is the only path from u to v in G, u and v are in different components in $G-e$. The above arguments then show that the deletion of e in this case has split up C into two components in $G-e$, namely one where every vertex is connected to u and another in which every vertex is connected to v. Thus we have increased the number of components by 1, i.e., in this case $\omega(G-e) = \omega(G)+1$.

The above argument assumes that u is different from v. If $u = v$ then e is a loop and its removal does not change the number of components, i.e., if e is a loop then $\omega(G-e) = \omega(G)$. □

An edge e of a graph G is called a **bridge** (or a **cut edge** or an **isthmus**) if the subgraph $G-e$ has more connected components than G has.

Thus, if e is a bridge in G, by Theorem 2.6 we have $\omega(G-e) = \omega(G)+1$, i.e., $G-e$ has one more component than G has. The bridges of the graph G of Figure 2.8 are the edges e and f.

Roughly speaking a bridge is an edge which is the only link between two parts of a graph. Its deletion will cut up the graph into more disjoint parts. We illustrate this in Figure 2.8.

The proof of Theorem 2.6 shows that an edge e with end vertices u and v is a bridge in the graph G if and only if e is not a loop and e gives the only path in G from u to v. Another way of saying this is given in the following theorem.

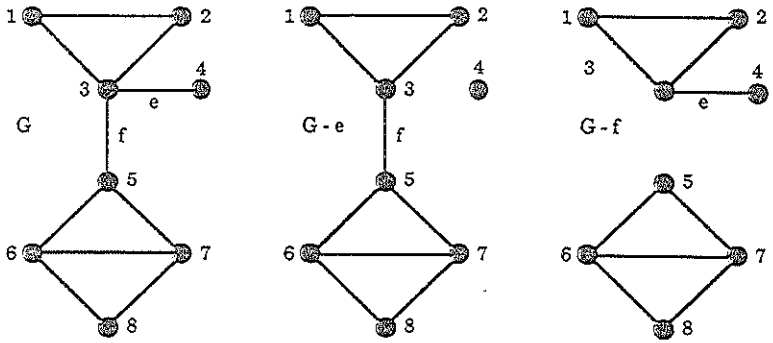

Figure 2.8: A graph and two bridge deletions.

Theorem 2.7 *An edge e of a graph G is a bridge if and only if e is not part of any cycle in G.*

Proof Let e have end vertices u and v. If e is not a bridge then, by the above remarks, it is either a loop or there is a path $P = uu_1 \dots u_n v$ from u to v, different from the edge e. If it is a loop then it forms a cycle (by itself). If there is such a path P then $C = uu_1 \dots u_n vu$, the concatenation of P with e, is a cycle in G. This shows that if e is not a bridge then it is part of a cycle. This is equivalent to saying that if e is not part of any cycle then e must be a bridge.

Conversely, suppose that e is part of some cycle $C = u_0u_1 \dots u_m$ in G. Let $e = u_iu_{i+1}$. In the case where $m = 1$, $C = u_0u_1$ and so C is just the edge e and e is a loop. On the other hand, if $m > 1$ then $P = u_iu_{i-1} \dots u_0u_{m-1} \dots u_{i+1}$ is a path from u to v different from e. (See Figure 2.9.) Thus, by the remarks preceding the proof, e is not a bridge. This shows that if e is a bridge then it is not part of any cycle in G, completing the proof. □

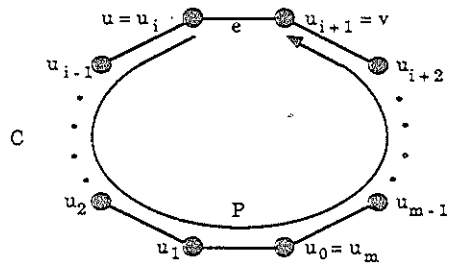

Figure 2.9

We can now describe trees using bridges by the following result.

Theorem 2.8 *Let G be a connected graph. Then G is a tree if and only if every edge of G is a bridge, i.e., if and only if for every edge e of G the subgraph $G - e$ has two components.*

Proof Suppose that G is a tree. Then G is acyclic, i.e., it has no cycles, and so no edge of G belongs to a cycle. In other words, if e is any edge of G then, by Theorem 2.7, it is a bridge, as required.

Conversely suppose that G is connected and that every edge e of G is a bridge. Then G can have no cycles since any edge belonging to a cycle is not a bridge, by Theorem 2.7. Hence G is acyclic and so is a tree as required. □

The corollary of the next result will enable us to give yet another characterisation of trees.

Theorem 2.9 *Let G be a graph with n vertices and q edges and, as usual, let $\omega(G)$ denote the number of connected components of G. Then G has at least $n - \omega(G)$ edges, i.e., $q \geq n - \omega(G)$.*

Proof We use induction on q, starting with $q = 0$. If $q = 0$ then G has no edges and so $\omega(G) = n$, since in this case each vertex constitutes a component. Thus for $q = 0$ we have $q = 0 \geq 0 = n - \omega(G)$, establishing the result in this case.

Although we do not need to prove the result for $q = 1$ at this stage let us do so to give peace of mind to those worried about induction starting at 0 instead of 1. If $q = 1$ then there is only one edge in G, say with end(s) u and v. Then u and v are in the same component and all other vertices each give another component. There being at least $n - 2$ other such vertices ($n - 1$ if $u = v$), we get $\omega(G) \geq n - 1$ and so $n - \omega(G) \leq 1 = q$, as required.

Now suppose the result is true for some value k of q where $k \geq 0$. Using this assumption we will prove that it is true for a graph of $k + 1$ edges. Thus let G be a graph with $k + 1$ edges and select one of these edges, e, say. Then the subgraph $G - e$ has k edges and so by our assumption

$$k \geq n - \omega(G - e). \quad (*)$$

(Note that $G-e$ has still n vertices.) Now, by Theorem 2.6, we have $\omega(G-e) \leq \omega(G)+1$ so, multiplying by -1, $-\omega(G - e) \geq -\omega(G) - 1$. Substituting back into $(*)$ gives $k \geq n - \omega(G) - 1$ and so $k + 1 \geq n - \omega(G)$, i.e., our graph G with $k + 1$ edges satisfies the required inequality. Hence, assuming the result true for k, we have shown it to be true for $k + 1$. Thus, by mathematical induction, the result is true in general. □

Corollary 2.10 *A connected graph G with n vertices has at least $n - 1$ edges. (In other words, a graph with n vertices and less than $n - 1$ edges can not be connected.)*

Proof If G is connected then $\omega(G) = 1$ so, by the Theorem, if q denotes the number of edges in G we have $q \geq n - 1$. □

We now provide the promised characterisation of trees:

Theorem 2.11 *Let G be a graph with n vertices. Then the following three statements are equivalent:*
(i) G is a tree,
(ii) G is an acyclic graph with $n-1$ edges,
(iii) G is a connected graph with $n-1$ edges.

Proof We prove (i) $\Rightarrow$ (ii) $\Rightarrow$ (iii) $\Rightarrow$ (i).

(i) $\Rightarrow$ (ii): Suppose that G is a tree. Then by definition G is an acyclic graph and by Theorem 2.4 it has $n-1$ edges. Thus (ii) holds.

(ii) $\Rightarrow$ (iii): Assume that G is an acyclic graph with $n-1$ edges and, as usual, let $\omega(G)$ denote the number of connected components of G. Then, by Theorem 2.5, G has $n-\omega(G)$ edges. Thus $\omega(G)=1$, in other words, G is connected. This establishes (iii).

(iii) $\Rightarrow$ (i): Assume that G is a connected graph with $n-1$ edges. To prove (i), i.e., that G is a tree, we must show that G is acyclic. We do this by contradiction. Thus, assume that G is not acyclic. Then G contains a cycle and every edge of this cycle can not be a bridge, by Theorem 2.7. Choose such an edge e. Then, since e is not a bridge, $G-e$ is still connected. However $G-e$ has $n-2$ edges and n vertices, which is impossible by the above corollary. This contradiction has arisen from our assumption that G is not acyclic. Hence G is acyclic and so a tree as required. □

Exercises for Section 2.2

2.2.1 Find all bridges in the graph of Figure 2.10.

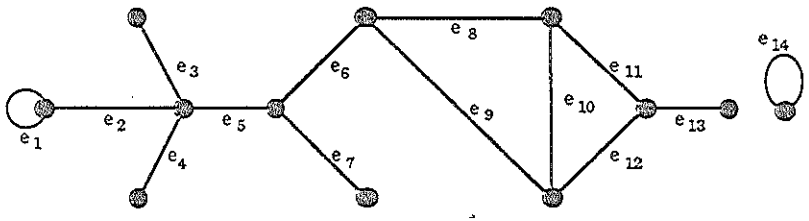

Figure 2.10: Which edges are bridges?

2.2.2 Let G be a connected graph.

(a) If G has 17 edges what is the maximum possible number of vertices in G?

(b) If G has 21 vertices what is the minimum possible number of edges in G?

2.2.3 Let G be a graph with 4 connected components and 24 edges. What is the maximum possible number of vertices in G?

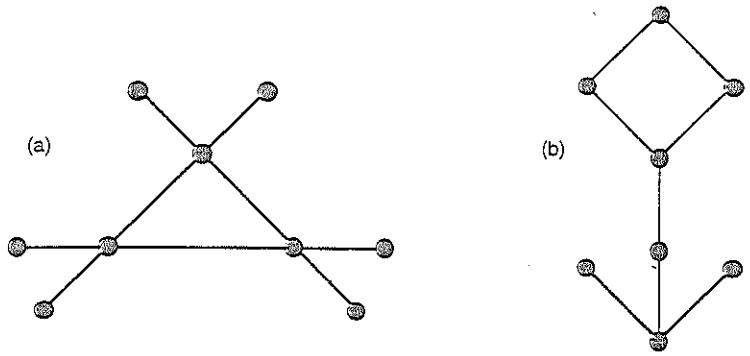

Figure 2.11: Two unicyclic graphs.

2.2.4 A graph G is called **unicyclic** if it is connected and contains precisely one cycle. The graphs of Figure 2.11 are unicyclic.

Prove that a connected graph G with n vertices and e edges is unicyclic if and only if $n=e$.

2.3 Spanning Trees

Let G be a graph. Recall from Section 1.4 that a subgraph H of G is called a spanning subgraph of G if the vertex set of H is the same as the vertex set of G.

A **spanning tree** of a graph G is a spanning subgraph of G that is a tree.

Our first result of this section shows that the graphs which have spanning trees are easily described.

Theorem 2.12 *A graph G is connected if and only if it has a spanning tree.*

Proof Suppose that G is connected with n vertices and q edges. Then, by Corollary 2.10, we have $q \geq n-1$. If $q=n-1$ then, by (iii) $\Rightarrow$(i) of Theorem 2.11, G is a tree and so we can take $T=G$ as a spanning tree of G.

If $q>n-1$ then, by Theorem 2.4 (or by (i) $\Rightarrow$ (iii) of Theorem 2.11), G is not a tree and so G must contain a cycle. Let e_1 be an edge of such a cycle. Then the subgraph $G-e_1$ is connected (since e_1 is not a bridge), has n vertices, and has $q-1$ edges. If $q-1=n-1$ then, repeating the above argument gives $T=G-e_1$ as a spanning tree of G.

If $q-1>n-1$ then $G-e_1$ is not a tree so, as before, there is a cycle in $G-e_1$. Removing an edge e_2 from such a cycle gives a subgraph $G-\{e_1,e_2\}=(G-e_1)-e_2$

which is connected, has n vertices and $q-2$ edges. We keep on repeating this process, deleting $q-n+1$ edges altogether, to eventually produce a subgraph T which is connected, has n vertices and $q-(q-n+1)=n-1$ edges. Thus by Theorem 2.11, T is a tree and since it has the same vertex set as G it is a spanning tree of G.

Conversely, if G has a spanning subtree T, then given any two vertices u and v of G then u and v are also vertices of the connected subgraph T. Thus u and v are connected by a path in T and so by a path in G. This shows that G is connected. □

Figures 2.12 and 2.13 illustrate the Theorem.

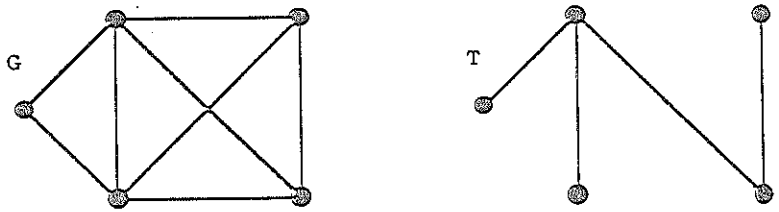

Figure 2.12: A connected graph and a spanning tree.

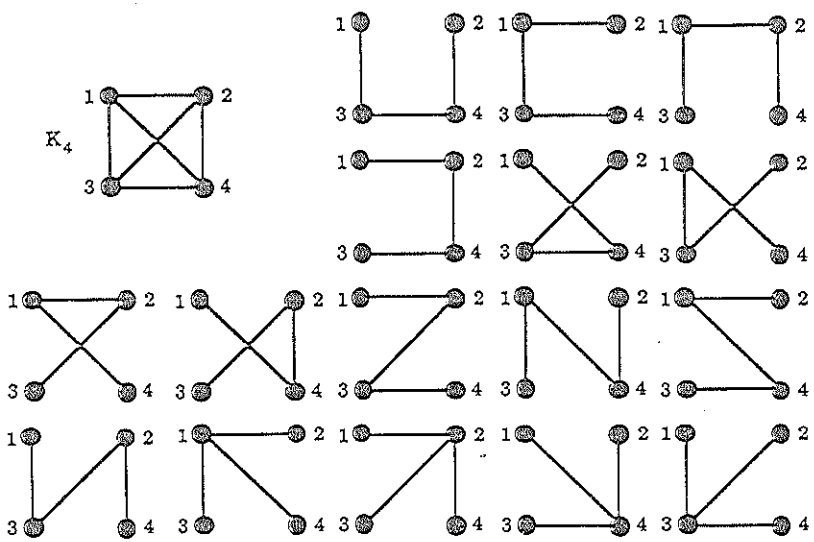

Figure 2.13: K_4 and its 16 different spanning trees.

Note that in the 16 different spanning trees of K_4 shown in Figure 2.13 there are only two non-isomorphic ones — the first 12 shown are isomorphic to each other, while the last four are also isomorphic to each other. K_6, the complete graph on 6 vertices, has 1296 *different* spanning trees, but just 6 *non-isomorphic* ones. Another

way of saying this is, given 6 vertices, then there are 1296 different ways of joining these vertices to form a tree if we label the vertices $1,2,3,4,5,6$, but if we drop these labels then there are only 6 different ways.

The subject of counting how many spanning trees and non-isomorphic spanning trees there are for a given graph was probably initiated by the English mathematician Arthur Cayley, who used trees to try to count the number of saturated hydrocarbons C_nH_{2n+2} containing a given number of carbon atoms. Cayley was the first person to use the term "tree" (in 1857) and in 1889 [12] he proved the following result which tells us that given n vertices, labelled $1,\ldots,n$, then there are n^{n-2} different ways of joining them to form a tree.

Theorem 2.13 (Cayley, 1889) *The complete graph K_n has n^{n-2} different spanning trees.*

Proof We omit the proof but, for those interested, see pages 32–35 of Bondy and Murty [7] or pages 50–52 of Wilson [65]. □

Exercises for Section 2.3

2.3.1 Give a list of all spanning trees, including isomorphic ones, of the connected graphs of Figure 2.14. How many non-isomorphic spanning trees are there in each case?

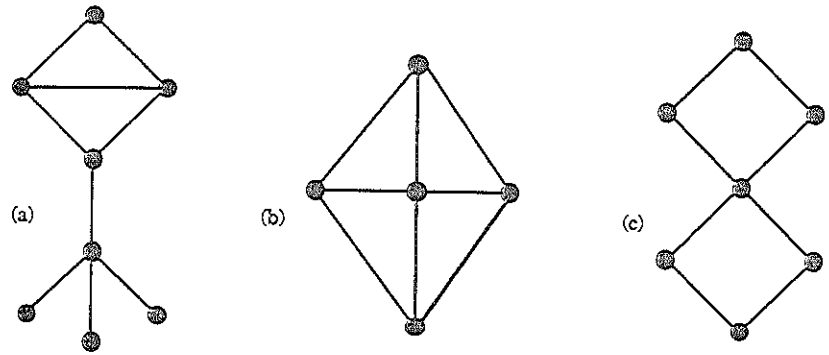

Figure 2.14

2.3.2 Let T be a tree with at least k edges, $k \geq 2$. How many connected components are there in the subgraph of T obtained by deleting k edges of T?

2.3.3 Let G be a connected graph which is not a tree and let C be a cycle in G. Prove that the complement of any spanning tree of G contains at least one edge of C.

2.3.4 Let e be an edge of the connected graph G.

(a) Prove that e is a bridge if and only if it is in every spanning tree of G.

(b) Prove that e is a loop if and only if it is in no spanning tree of G.

2.3.5 Let G be a graph with exactly one spanning tree. Prove that G is a tree.

2.3.6 An edge e (not a loop) of a graph G is said to be **contracted** if it is deleted and then its end vertices are fused (identified). The resulting graph is denoted by $G * e$. Figure 2.15 illustrates this.

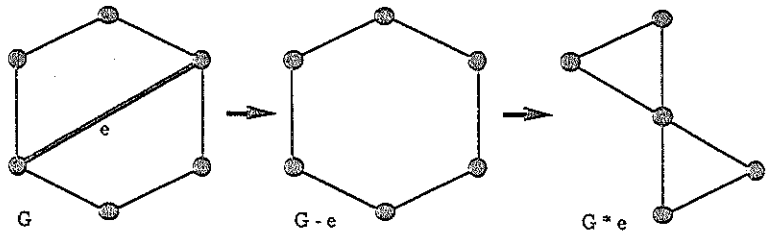

Figure 2.15

(a) Prove that if T is a spanning tree of G which contains e then $T * e$ is a spanning tree of $G * e$.

(b) Prove that if U is a spanning tree of $G * e$ then there is a unique spanning tree T of G which contains e and is such that $U = T * e$.

(c) Let $\tau(G)$ denote the number of different (not necessarily non-isomorphic) spanning trees of the connected graph G. Prove, using (a) and (b), that if e is an edge of G which is not a loop then

$$\tau(G) = \tau(G - e) + \tau(G * e).$$

2.3.7 Part (c) of Exercise 2.3.6 provides a way of calculating $\tau(G)$ for any connected graph G. Following the presentation given in Bondy and Murty [7], we illustrate it with an example in Figure 2.16 where the number of spanning trees for each graph is denoted pictorially by the graph itself. The idea is to break down the initial graph by a series of edge contractions to produce a collection of trees or "trees with loops". The number of graphs in this collection is then $\tau(G)$ for the initial graph G. Part (c) of Exercise 2.3.6 is used in each step of the breakdown. The contracted edges are shown thicker. The final "expression" consists of 11 graphs which are either trees or "trees with loops" and so we can conclude that $\tau(G) = 11$.

Use this method to find $\tau(G)$ for the graph G of Figure 2.15.

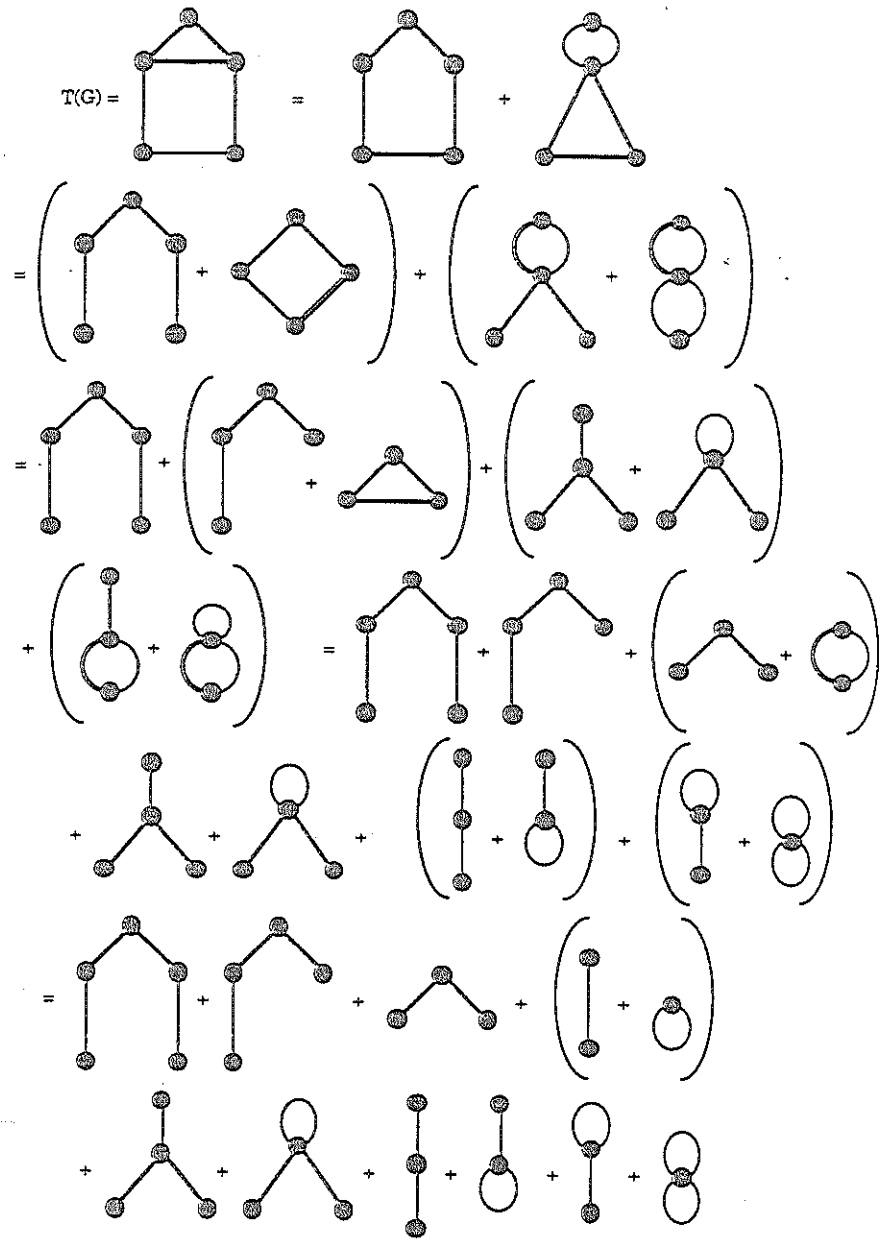

Figure 2.16: Calculation of $\tau(G)$ using contractions.

2.4 Connector Problems

Suppose certain villages in an area are to be joined to a water supply situated in one of the villages. The system of pipes is to consist of pipelines connecting the water towers of two villages. For any two villages we know how much it would cost to build a pipeline connecting them, provided such a pipeline can be built at all. How can we find an economical system of pipes? This is an example of a connector problem and we solve it using spanning trees and the concept of a weighted graph.

A **weighted graph** is a graph G in which each edge e has been assigned a real number $w(e)$, called the **weight** (or length) of e. If H is a subgraph of a weighted graph, the **weight** $w(H)$ of H is the sum of the weights $w(e_1) + \cdots + w(e_k)$ where $\{e_1, \ldots, e_k\}$ is the set of edges of H.

Many optimisation problems amount to finding, in a suitable weighted graph, a certain type of subgraph with minimum (or maximum) weight. In our introductory problem we let G be the graph whose vertex set is the set of villages and in which xy is an edge if (and only if) it is possible to build a pipeline joining the villages x and y. We can then make G into a weighted graph by assigning to each edge the cost of constructing the corresponding pipeline. For example suppose that there are 6 villages, A, B, C, D, E, F, and we get the weighted graph G of Figure 2.17.

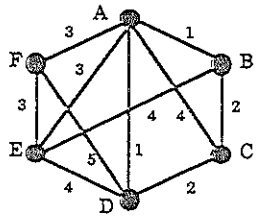

Figure 2.17: A weighted graph.

The lack of an edge from B to D (for example) indicates that it is not possible to build a pipeline from B to D. The number (weight) 4 assigned to the edge from A to C indicates the cost of building a pipeline from A to C, for example.

Since part of the problem is to ensure that every village is supplied with water from the source village we are looking for a connected spanning subgraph of G. Moreover since we want to do this in the most economical way, such a spanning subgraph should have no cycles, because the deletion of an edge (a pipeline) from a cycle in a connected spanning subgraph still leaves us with a connected spanning subgraph. Thus we are looking for a spanning tree of G. Moreover the economical factor means that we want the cheapest such spanning tree, i.e., a spanning tree of minimum weight.

Such a tree is called a **minimal spanning tree** or an **optimal tree** for G.

We now present two ways, due to Kruskal [38] and Prim [51], of finding a minimal spanning tree for a connected weighted graph where no weight is negative.

(1) *Kruskal's algorithm.*

In this algorithm we first choose an edge of G which has smallest weight among the edges of G which are not loops. Next, again avoiding loops, we choose from the remaining edges one of smallest weight possible which does not form a cycle with the edge already chosen. We repeat this process taking edges of smallest weight among those not already chosen, provided no cycle is formed with those that have been chosen. If the graph has n vertices we stop after having chosen $n-1$ edges. These edges form an acyclic subgraph T of G and we will prove below that T is a minimal spanning tree of G. This gives the following three step procedure:

Kruskal's Algorithm

Step 1. Choose e_1, an edge of G, such that $w(e_1)$ is as small as possible and e_1 is not a loop.

Step 2. If edges $e_1, e_2, \ldots, e_i$ have been chosen, then choose an edge e_{i+1}, not already chosen, such that

(i) the induced subgraph $G[\{e_1, \ldots, e_{i+1}\}]$ is acyclic and

(ii) $w(e_{i+1})$ is as small as possible (subject to condition (i)).

Step 3. If G has n vertices, stop after $n-1$ edges have been chosen. Otherwise repeat Step 2.

In Figure 2.18 we perform the algorithm on our example G of Figure 2.17, indicating at each stage the edges chosen by shaded lines. The weight, $w(T)$, of T is 1+1+2+3+3, i.e., $w(T) = 10$.

We still have to prove that Kruskal's algorithm does produce a minimal spanning tree. We do this now:

Theorem 2.14 *Let G be a weighted connected graph in which the weights of the edges are all non-negative numbers. Let T be a subgraph of G obtained by Kruskal's algorithm. Then T is a minimal spanning tree of G.*

Proof As noted in the description of the algorithm, T is an acyclic subgraph of G with $n-1$ edges. If T has m vertices and k connected components then, by Theorem 2.5, it has $m-k$ edges, i.e., $n-1 = m-k$. Since $m \leq n$ and $k \geq 1$ this can only happen when $n = m$ and $k = 1$ (because $n - m = 1 - k$). Thus T is connected and a spanning subgraph, i.e., T is indeed a spanning tree of G.

It remains then to show that the weight of T is at a minimum. In order to do this, we suppose that S is a spanning tree of G with less weight than T, i.e., $w(S) < w(T)$, and obtain a contradiction. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges of T in the order that they were produced by Kruskal's algorithm. Since S is different from T there will be a first edge, e_k say, in this sequence which does not lie in S. Then the subgraph H of G obtained by adding the edge e_k to S has n edges and so is no longer a tree (by Theorem 2.4). Thus this subgraph H must contain a cycle C, say. C must contain the edge e_k, since otherwise C would be in the acyclic S. Moreover C must contain

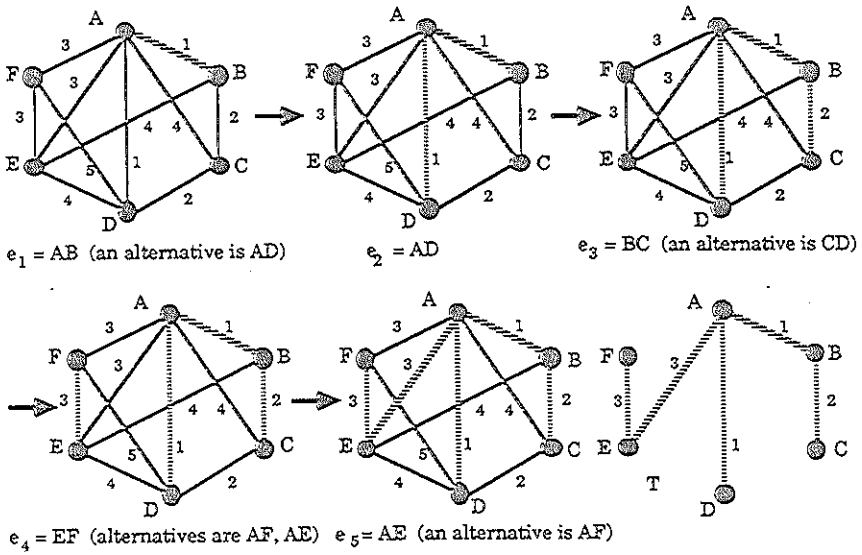

Figure 2.18: A minimal spanning tree construction using Kruskal's algorithm.

an edge e of S that is not in T, since otherwise C would be in the acyclic T. The subgraph $H - e$ is then still connected (since e belongs to a cycle in H) and so, since it has $n - 1$ edges, is also a spanning tree of G. However, since e is not an edge in T and e_k is the first edge in T which is not in S, by the algorithm it follows that $w(e_k) \leq w(e)$. (If $w(e) < w(e_k)$ we would have chosen e instead of e_k as the kth edge.) Since $H - e$ has been formed by replacing e with e_k and $w(e_k) \leq w(e)$ we have that $w(H - e) \leq w(S)$. Moreover $H - e$ has one more edge in common with T than S has because e_k is in T while e is not..

The procedure that we just performed on S to produce $H - e$ is now performed repeatedly, on $H - e$ and its successors, one step at a time, to gradually change S into T, each stage giving a spanning tree with weight at most $w(S)$. Thus at the final stage we get $w(T) \leq w(S)$, a contradiction to our earlier assumption. Hence $w(T)$ is after all at a minimum and so T is a minimum spanning tree, as required. □

(2) *Prim's algorithm.*

In this algorithm for finding a minimal spanning tree we first choose a vertex v_1 of the connected graph G — any vertex v_1 will do. We next choose one of the edges of smallest weight in G which is not a loop and which is incident with v_1, say $e_1 = v_1v_2$. Next we choose an edge of smallest weight in G which is incident with either v_1 or v_2 but with the other end point neither of these, i.e., we choose $e_2 = v_iv_3$ where $i \in \{1, 2\}$ but $v_3 \neq v_1, v_2$. We repeat this process of taking edges of smallest weight one of whose ends is a vertex previously chosen and the other end becoming involved for the first time, until we have chosen $n - 1$ edges (assuming the graph has n vertices). Then at

this stage we have involved each of the n vertices of G and by the construction the resulting subgraph is connected. Hence it is a tree by Theorem 2.11 and so a spanning tree of G. We shall prove below that it is a minimal spanning tree. (Note: as in Kruskal we do assume all weights are non-negative.) The algorithm then involves four steps:

Prim's Algorithm

Step 1. Choose any vertex v_1 of G.

Step 2. Choose an edge $e_1 = v_1v_2$ of G such that $v_2 \neq v_1$ and e_1 has smallest weight among the edges of G incident with v_1.

Step 3. If edges $e_1, e_2, \ldots, e_i$ have been chosen involving end points $v_1, v_2, \ldots, v_{i+1}$ choose an edge $e_{i+1} = v_jv_k$ with $v_j \in \{v_1, \ldots, v_{i+1}\}$ and $v_k \notin \{v_1, \ldots, v_{i+1}\}$ such that e_{i+1} has smallest weight among the edges of G with precisely one end in $\{v_1, \ldots, v_{i+1}\}$.

Step 4. Stop after $n - 1$ edges have been chosen. Otherwise repeat Step 3.

In Figure 2.19 we perform the algorithm on our example G of Figure 2.17 (the same example used for Kruskal's algorithm), indicating at each stage the edges chosen by shaded lines. Figure 2.20 shows the resulting spanning tree.

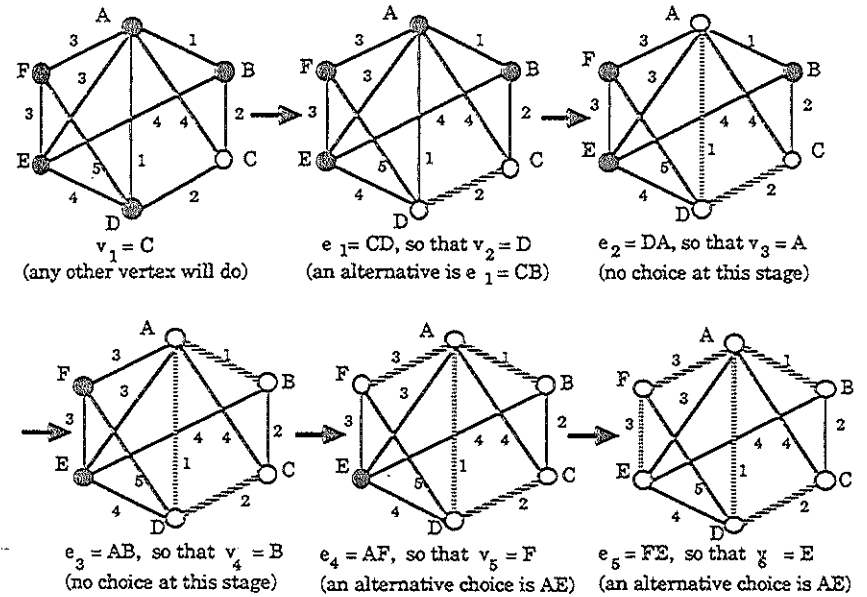

Figure 2.19: A minimal spanning tree construction using Prim's algorithm.

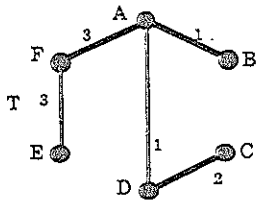

Figure 2.20: The minimal spanning tree.

The weight, $w(T)$, of T is $w(e_1)+w(e_2)+w(e_3)+w(e_4)+w(e_5) = 2+1+1+3+3 = 10$ (which is, as expected, the weight of the minimal spanning tree given by Kruskal's algorithm).

The main differences between Kruskal's algorithm and Prim's algorithm are that

(i) Kruskal's may lead to several subtrees being grown "simultaneously" and then being joined together whereas Prim's leads to one subtree growing steadily, from one initial vertex and

(ii) Kruskal's depends on being able to detect cycles whereas Prim's depends on not choosing a previously chosen vertex.

Difference (ii) means that in computer implementation Prim's algorithm is usually faster than Kruskal's.

We now prove that Prim's algorithm does indeed produce a minimal spanning tree.

Theorem 2.15 *Let G be a weighted connected graph in which the weights of the edges are all non-negative numbers. Let T be a subgraph of G obtained by Prim's algorithm. Then T is a minimal spanning tree of G.*

Proof As noted in the description of the algorithm, T is indeed a spanning tree of G. Thus it remains to show that the weight of T is at a minimum. In order to do this we suppose that S is a minimal spanning tree of G chosen to have as many edges in common with T as possible. We shall prove that $S = T$, so showing that T is minimal. We do this by contradiction.

Hence suppose that $S \neq T$. Then T has at least one edge which is not in S. Let e_k be the first edge chosen by Prim's algorithm which is in T but not in S, i.e., e_k is the kth edge chosen by the algorithm, e_k is not in S, but previous edges (if any) chosen by the algorithm are all in S (as well as T). Suppose that e_k has end vertices u and v. Then, since u and v are in the tree S there is a unique path P in S connecting u to v, and P does not involve e_k.

Now if T_i denotes the subtree created in G after the addition of the ith edge e_i, for $1 \leq i \leq n-1$, then, by the description of Prim's algorithm, one end of e_k is in T_{k-1} and the other is not. Let us suppose that u is in T_{k-1} but v is not. Then, since P is a path from u to v it must involve at least one edge having one end in T_{k-1} and the other not. Let e^* denote such an edge in the path P. Then $w(e^*) \geq w(e)$ since otherwise e^* has less weight than e_k and Prim's algorithm would then have incorporated e^* and not e_k as the kth edge.

Now the path P in S together with the edge e_k gives a cycle in G so if we replace the edge e^* in S with the edge e_k we still have a connected subgraph with n vertices and $n-1$ edges. In other words, replacing e^* in S with e_k gives a new spanning tree R. Since $w(e^*) \geq w(e_k)$, the weight of R is not greater than that of S and so R must also be a minimal spanning tree. However R has one more edge in common with T than S has, namely the edge e_k. This contradicts the assumption that S was chosen to be a minimal spanning tree with as many edges in common with T as possible. This contradiction has arisen from the supposition that $S \neq T$. Hence $S = T$ and T is a minimal spanning tree, as required. □

For computer implementation a weighted graph is usually presented in matrix form: if G has n vertices and w_{ij} denotes the weight of an edge from vertex v_i to v_j, then, assuming that G has no parallel edges, we may present G as the $n \times n$ matrix with (i,j)th entry w_{ij}, taking w_{ij} as ∞ if there is no edge from v_i to v_j.

Thus, for example, taking $v_1 = A, v_2 = B$, etc. in our example above, our graph G is represented by the 6×6 matrix

$$\begin{bmatrix} \infty & 1 & 4 & 1 & 3 & 3 \\ 1 & \infty & 2 & \infty & 4 & \infty \\ 4 & 2 & \infty & 2 & \infty & \infty \\ 1 & \infty & 2 & \infty & 4 & 5 \\ 3 & 4 & \infty & 4 & \infty & 3 \\ 3 & \infty & \infty & 5 & 3 & \infty \end{bmatrix}.$$

The two algorithms have generated minimal spanning trees. They may also be used to generate *maximal* spanning trees, i.e., spanning trees which have *greatest* possible weight. To do this we create a new weighted graph with the same vertices and edges as the one given but where we replace each weight $w(e)$ by $M - w(e)$ where M is any number greater than the weight $w(e)$ of every edge e of the graph. Then any minimal spanning tree of this new weighted graph has the sum of its weights $M - w(e)$ at a minimum, i.e., the sum of the weights $w(e)$ are at a *maximum*, and so the corresponding spanning tree in the original weighted graph is a *maximal* spanning tree.

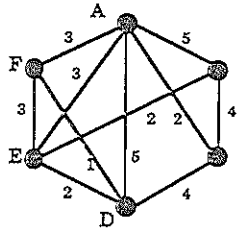

Figure 2.21: The weighted graph G'.

For example, in our graph G of Figure 2.17 take $M = 6$, since 6 is greater than each $w(e)$, to give the new weighted graph G' shown in Figure 2.21. By performing one of

the two algorithms on G' we may find a minimal spanning tree for G', as for instance shown in Figure 2.22. We then convert this into a maximal spanning tree for G. (Again see Figure 2.22.) The maximal spanning tree here has weight $5+4+4+3+4=20$. Of course, every other maximal spanning tree also has this weight.

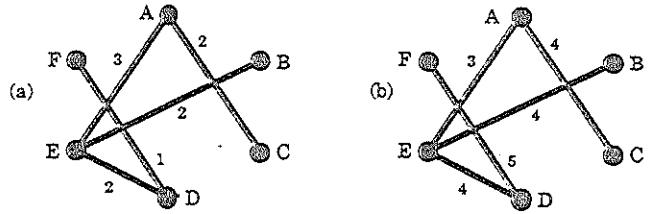

Figure 2.22: (a) A *minimal* spanning tree for the weighted graph G' of Figure 2.21 which gives (b) a *maximal* spanning tree for the weighted graph G of Figure 2.17.

Exercises for Section 2.4

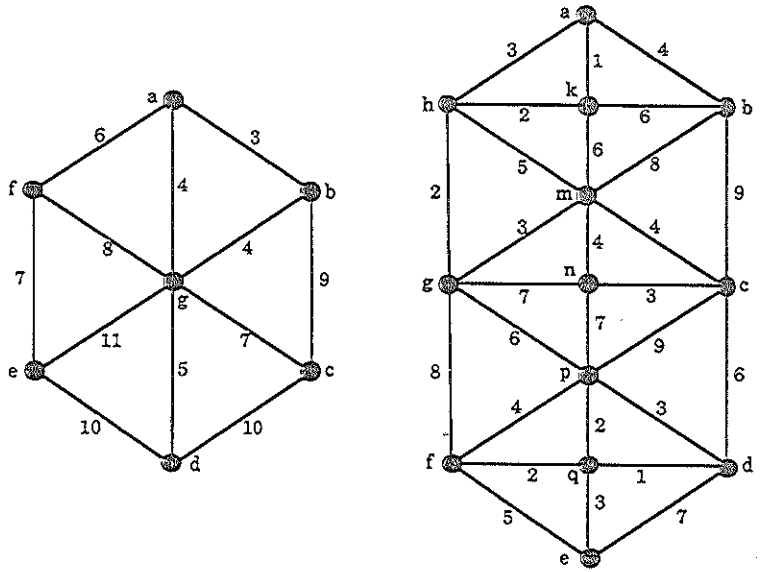

Figure 2.23

2.4.1 Find a minimal spanning tree for each of the connected weighted graphs of Figure 2.23 using both Kruskal's algorithm and Prim's algorithm.

2.4.2 Find a *maximal* spanning tree for each of graphs of Figure 2.23 using either Kruskal's algorithm or Prim's algorithm.

2.4.3 The following is a third algorithm for finding a minimal spanning tree of a connected weighted graph G with n vertices where each weight is non-negative. Simply delete one by one those edges of G with largest weight, provided each such deletion does not result in a disconnected graph, until there are just $n-1$ edges left. Then the resulting subgraph is a minimal spanning tree of G. Perform this algorithm on the graphs of Figure 2.23.

2.4.4 Prove that if G is a connected weighted graph in which no two edges have the same weight then G has a unique minimum spanning tree.

2.5 Shortest Path Problems

(1) *The Breadth First Search (BFS) technique.* Let G be a graph and let s, t be two specified vertices of G. We will now describe a method of finding a path from s to t, if there is any, which uses the least number of edges. Such a path, if it exists, is called a shortest path from s to t. The method assigns labels $0, 1, 2, \ldots$ to vertices of G and is called the Breadth First Search (BFS for short) technique. It is given by the following algorithm.

The Breadth First Search Algorithm

Step 1. Label vertex s with 0. Set $i = 0$.

Step 2. Find all unlabelled vertices in G which are adjacent to vertices labelled i. If there are no such vertices then t is not connected to s (by a path). If there are such vertices, label them $i+1$.

Step 3. If t is labelled, go to Step 4. If not, increase i to $i+1$ and go to Step 2.

Step 4. The length of a shortest path from s to t is $i+1$. Stop.

For example, for the graph of Figure 2.24, first s is labelled 0. Then a and f are labelled 1. Then b, d and e are labelled 2. Then c and t are labelled 3. Since t is labelled 3, the length of a shortest path from s to t is 3.

For the graph of Figure 2.25, first s is labelled 0. Then a, b and c are labelled 1. Then d is labelled 2. But now there are no unlabelled vertices adjacent to d, the only vertex labelled 2. Hence, by Step 2, we can conclude that there is no path from s to t.

We now show that Step 4's statement is justified by proving that the Breadth First Search works.

Theorem 2.16 *A vertex v of the graph G is labelled the number $\lambda(v)$ by the BFS algorithm above if and only if the length of a shortest path from s to v is $\lambda(v)$. (In particular if t is labelled n then the shortest path from s to t has n edges.)*

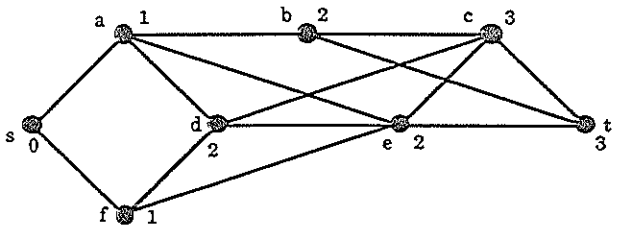

Figure 2.24

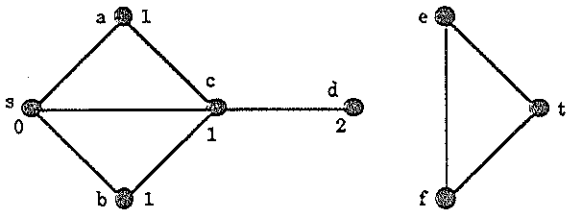

Figure 2.25

Proof The proof is by induction on the value i of $\lambda(v)$. First if $i = 0$ then v has label $\lambda(v) = 0$ and by the algorithm this occurs precisely when $v = s$. However the length of a shortest path from s to s is 0 and so in this case the label $\lambda(v)$ does indeed give the length of the shortest path from s to v.

Now assume that the statement is true for all values of i from 0 up to and including a fixed value k, i.e., if the vertex v of G has been labelled $\lambda(v)$ where $0 \leq \lambda(v) \leq k$ then the length of a shortest path from s to v is $\lambda(v)$ and conversely any vertex v having shortest path to s of length m where $0 \leq m \leq k$ has label $\lambda(v) = m$. Let u be a vertex of G which has label $\lambda(u) = k+1$. Then by the algorithm there is an edge e from a vertex v with $\lambda(v) = k$ to the vertex u. By our assumption there is a shortest path of length k from s to v and concatenating this with e gives a path of length $k+1$ from s to u. Moreover this path is the shortest possible since if there were one shorter then by our induction assumption u would have been given a label $\lambda(u)$ with $0 \leq \lambda(u) \leq k$. Thus if u is a vertex with $\lambda(u) = k+1$ then there is a shortest path of length $k+1$ from s to u.

Conversely, if there is a shortest path from s to a vertex w of length $k+1$ then the second last vertex v, say, of this path has a shortest path to s of length k and so, by our assumption, $\lambda(v) = k$. Since w is adjacent to v, the algorithm gives $\lambda(w) = k+1$. This completes the verification of the statement for $k+1$. Hence by induction the result is true for all values of $\lambda(v)$, as required. □

Once the algorithm has been performed successfully, i.e., Step 4 has been accomplished, we can use the following trace-back (or back-tracking) algorithm to find an

actual shortest path from s to t. This algorithm uses the labels $\lambda(v)$ which were produced in the BFS algorithm. It produces a path $v_0, v_1, \ldots, v_{\lambda(t)}$ such that $v_0 = s$ and $v_{\lambda(t)} = t$.

The Back-tracking Algorithm for a Shortest Path

Step 1. Step 1. Set $i = \lambda(t)$ and assign $v_i = t$.

Step 2. Step 2. Find a vertex u adjacent to v_i and with $\lambda(u) = i-1$. Assign $v_{i-1} = u$.

Step 3. Step 3. If $i = 1$, stop. If not, decrease i to $i-1$ and go to Step 2.

For example for the graph of Figure 2.24 we have $\lambda(t) = 3$ so we start with $i = 3$ and $v_3 = t$ (Step 1). We can then choose e adjacent to $v_3 = t$ with $\lambda(e) = 2$ and assign $v_2 = e$. Next we can choose f adjacent to $v_2 = e$ with $\lambda(f) = 1$ and assign $v_1 = f$. Finally we take s adjacent to f with $\lambda(s) = 0$ and assign $v_0 = s$. This gives the shortest path $v_0v_1v_2v_3 = s\ f\ e\ t$ from s to t. In general there may of course be many shortest paths from s to t and the previous algorithm finds just one of them. A simple extra labelling, assigning a label $\mu(v)$ to each vertex v which has been labelled in the BFS algorithm in a back-tracking manner, actually produces $\mu(s)$ which is the number of shortest paths from s to t. This is done by the following algorithm.

The Back-tracking Algorithm for the Number of Shortest Paths

Step 1. Set $i = \lambda(t)$ and $\mu(t) = 1$. All other vertices v for which $\lambda(v) = \lambda(t)$ are assigned $\mu(v) = 0$.

Step 2. For each vertex v which satisfies $\lambda(v) = i-1$ compute the sum

$$\sum \mu(u)$$

over all u's which satisfy the following condition: $\lambda(u) = i$ and v is adjacent to u; if there are parallel edges, $\mu(u)$ is repeated in this summation as many times as there are parallel edges. For each such v, set $\mu(v)$ equal to this sum.

Step 3. If $i = 1$, stop. If not, decrease i to $i-1$ and go to Step 2.

We illustrate the algorithm using the graph of Figure 2.24. We get:

Step 1. $i = 3$, $\mu(t) = 1$, $\mu(c) = 0$ (since $\lambda(c) = 3 = \lambda(t)$).

Step 2. The vertices v with $\lambda(v) = 2$ are b, d, e. Then $\mu(b) = \mu(t) + \mu(c)$ (since t, c are the vertices $u = 1 + 0$ with $\lambda(u) = 3$ and adjacent to b) $= 1$. Similarly $\mu(d) = \mu(c) = 0$ and $\mu(e) = \mu(t) + \mu(c) = 1$. In summary we have produced three new labels : $\mu(b) = 1$, $\mu(d) = 0$, $\mu(e) = 1$.

Step 3. Decrease $i = 3$ to 2 and go to Step 2.

Step 2. The vertices v with $\lambda(v) = 1$ are a, f. Then $\mu(a) = \mu(b) + \mu(d) + \mu(e) = 1 + 0 + 1 = 2$ and $\mu(f) = \mu(d) + \mu(e) = 0 + 1 = 1$.

Step 3. Decrease $i = 2$ to 1 and go to Step 2.

Step 2. The only vertex v with $\lambda(v) = i - 1 = 0$ is s. Then $\mu(s) = \mu(a) + \mu(f) = 2 + 1 = 3$.

Step 3. Since $i = 1$ we stop.

Our calculations give $\mu(s) = 3$ so there are 3 shortest paths from s to t. (In fact they are $s\,a\,e\,t$, $s\,a\,b\,t$ and $s\,f\,e\,t$.)

We omit the proof of the algorithm's validity. (The method is by induction, showing that for all vertices v for which $\lambda(v) < \lambda(t), \mu(v)$ is the number of paths of length $\lambda(t) - \mu(v)$ from v to t.)

(2) *Dijkstra's algorithm*

We now consider shortest path problems in weighted graphs. Given a path P from vertex s to vertex t in a weighted graph G we define the length of P to be the sum of the weights of its edges. (In fact this corresponds to the usual length of a path in an unweighted graph if we assign the weight 1 to each edge.) We wish to consider the problem of finding a shortest path from s to t, i.e., a path of least length. Since in general the weight of an edge can be negative it is possible to have paths of negative length. However the following algorithm due to Dijkstra [16] will be restricted to weighted graphs where the weight $w(e)$ of each edge e is non-negative, i.e., $w(e) \geq 0$.

As with BFS we use a labelling technique. Initially we set $\lambda(s) = 0$ and, for $v \neq s, \lambda(v)$ is as yet undetermined. We next label all neighbours v of s by $\lambda(v)$ where $\lambda(v)$ is the weight of the edge from s to v. Let u be the vertex among those v for which $\lambda(u)$ is minimum. Now find those neighbours w of u and, for those w not already labelled assign the label $\lambda(w) = \lambda(u) + w(e)$, $w(e)$ being the weight of the edge from u to w, while for those w already labelled $\lambda(w)$ change the label to $\lambda(u) + w(e)$ if this is smaller. Now find the w among these for which $\lambda(w)$ is a minimum ... this is getting too complicated! We now present the algorithm in stepwise form, using the convention that $\infty + x = \infty$ for any real number x, and $\infty + \infty = \infty$.

Dijkstra's Algorithm

Step 1. Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G. (We will think of T as the set of vertices "uncoloured".)

Step 2. Let u be a vertex in T for which $\lambda(u)$ is minimum.

Step 3. If $u = t$, stop.

Step 4. For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + w(e)$ change the value of $\lambda(v)$ to $\lambda(u) + w(e)$ (i.e., given an edge e from an "uncoloured" vertex v to u, change $\lambda(v)$ to $min(\lambda(v), \lambda(u) + w(e))$).

Step 5. Change T to $T - \{u\}$ and go back to Step 2, (i.e., "colour" u and then go back to Step 2 to find an "uncoloured" vertex with minimum label).

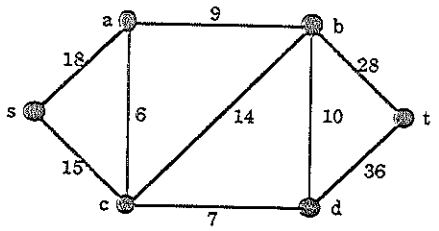

Figure 2.26

We illustrate this with the weighted graph of Figure 2.26.

Step 1. The initial labelling is given by:

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	∞	∞	∞	∞	∞
T	$\{s,$	$a,$	$b,$	$c,$	$d,$	$t\}$

Step 2. $u = s$ has $\lambda(u)$ a minimum (with value 0).

Step 4. There are two edges incident with u, namely sa and sc. Both a and c are in T, i.e., they are not yet coloured. $\lambda(a) = \infty > 18 = 0 + 18 = \lambda(s) + w(sa)$ so $\lambda(a)$ becomes 18. Similarly $\lambda(c)$ becomes 15.

Step 5. T becomes $T - \{s\}$, i.e., we colour s. Thus we have

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	18	∞	15	∞	∞
T	$\{$	$a,$	$b,$	$c,$	$d,$	$t\}$

Step 2. $u = c$ has $\lambda(u)$ a minimum and u in T (with $\lambda(u) = 15$).

Step 4. There are 3 edges cv incident with $u = c$ having v in T, namely ca, cb, cd. $\lambda(a) = 18 < 21 = 15 + 6 = \lambda(c) + w(ca)$ so $\lambda(a)$ remains as 18. $\lambda(b) = \infty > 29 = 15 + 14 = \lambda(c) + w(cb)$ so $\lambda(b)$ becomes 29. Similarly $\lambda(d)$ becomes $15 + 7 = 22$.

Step 5. T becomes $T - \{c\}$, i.e., we colour c. Thus we have

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	18	29	15	22	∞
T	$\{$	$a,$	$b,$		$d,$	$t\}$

Step 2. $u = a$ has $\lambda(u)$ a minimum for u in T (with $\lambda(u) = 18$).

Step 4. There is only one edge av incident with $u = a$ having v in T, namely ab. $\lambda(b) = 29 > 27 = 18 + 9 = \lambda(a) + w(ab)$ so $\lambda(b)$ becomes 27.

Step 5. T becomes $T - \{a\}$, i.e., we colour a. Thus we have

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	18	27	15	22	∞
T	{		b,		d,	t}

Step 2. $u = d$ has $\lambda(u)$ minimum for u in T (with $\lambda(u) = 22$).

Step 4. There are two edges dv incident with $u = d$ having v in T, namely db and dt. $\lambda(b) = 27 < 32 = 22 + 10 = \lambda(d) + w(db)$ so $\lambda(b)$ remains as 27. $\lambda(t) = \infty > 58 = 22 + 36 = \lambda(d) + w(dt)$ so $\lambda(t)$ becomes 58.

Step 5. T becomes $T - \{d\}$, i.e., we colour d. Thus we have

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	18	27	15	22	58
T	{		b,			t}

Step 2. $u = b$ has $\lambda(u)$ minimum for u in T (with $\lambda(u) = 27$).

Step 4. There is only one edge bv for v in T, namely bt. $\lambda(t) = 58 > 55 = 27 + 28 = \lambda(b) + w(bt)$ so $\lambda(t)$ becomes 55.

Step 5. T becomes $T - \{b\}$, i.e., we colour b. Thus we have

vertex v	s	a	b	c	d	t
$\lambda(v)$	0	18	27	15	22	55
T	{					t}

Step 2. $u = t$, the only choice.

Step 3. Stop.

When the algorithm stops the values $\lambda(v)$ give the lengths of the shortest paths from the vertex s to each vertex v. Thus the lengths of these paths from s to a, b, c, d, t are 18, 27, 15, 22 and 55 respectively.

In the following table we summarise the above process as carried out on our example, recording the choices of u (in the Steps 2), the changes in the $\lambda(v)$ values (in the Steps 4), and the changes to T, i.e., the colouring of the vertices, (in the Steps 5). The values in bold type indicate the choices of u and the corresponding $\lambda(u)$ given by Step 2 of the algorithm.

	vertices v						T					
	s	a	b	c	d	t						
	0	∞	∞	∞	∞	∞	{s,	a,	b,	c,	d,	t}
stepwise		18	∞	**15**	∞	∞	{	a,	b,	c,	d,	t}
$\lambda(v)$		**18**	29		22	∞	{	a,	b,		d,	t}
values			27		**22**	∞	{		b,		d,	t}
			27			58	{		b,			t}
						55	{					t}

We prove the validity of Dijkstra's algorithm in two stages. First

Theorem 2.17 *In Dijkstra's algorithm, if at some stage $\lambda(v)$ is finite (i.e., not ∞) for the vertex v then there is a path from s to v whose length is $\lambda(v)$.*

Proof If $v = s$ then $\lambda(v)$ is always 0 and the trivial path from s to itself has length 0. Thus the result is true if $v = s$. Now suppose $v \neq s$. Then, from the algorithm, since at Step 1 $\lambda(v) = \infty$ but now $\lambda(v)$ is finite, there must be some vertex u_1 given by Step 2 of the algorithm such that u_1 is adjacent to v, joined by the edge e_1 say, and the value $\lambda(v)$ is given by $\lambda(u_1) + w(e_1)$. Immediately after $\lambda(v)$ was given this value, Step 5 was applied to colour u_1. Also Steps 2 and 4 imply that once a vertex is coloured, i.e., no longer in T, then its label remains the same thereafter. Since $\lambda(u_1)$ is finite we can repeat the process to find the vertex u_2 adjacent to u_1, joined by the edge e_2 say, and with label $\lambda(u_1) = \lambda(u_2) + w(e_1)$. We keep on repeating this backward search until we get back to s. The vertices thus found form a path of length $\lambda(v)$ from v back to s since

$$\begin{aligned}
\lambda(v) &= \lambda(u_1) + w(e_1) \\
&= \lambda(u_2) + w(e_2) + w(e_1) \\
&= \lambda(u_3) + w(e_3) + w(e_2) + w(e_1) \\
&= \cdots \\
&= \lambda(s) + w(e_n) + \cdots + w(e_1) \\
&= w(e_n) + \cdots + w(e_1),
\end{aligned}$$

the length of the path consisting of the n weighted edges $e_1, \ldots, e_n$ produced in the process. We do have a path, i.e., no vertex u_i is repeated, since by the method of their choice, u_1 is coloured after u_2, u_2 is coloured after u_3, etc. □

For the second stage of the verification we let, for any vertex v, $\delta(v)$ denote the length of a shortest path from s to v , i.e., $\delta(v)$ is the distance from s to v , taking $\delta(v) = \infty$ if there is no path from s to v.

Theorem 2.18 *In Dijkstra's algorithm when a vertex u is chosen in Step 2 its label $\lambda(u)$ has value $\delta(u)$. (In particular, upon termination of the algorithm, $\lambda(t) = \delta(t)$.)*

Proof We use induction on the order in which the vertices are coloured, i.e., in which they leave T. The first vertex to be coloured is $v = s$ and since $\lambda(s) = 0 = \delta(s)$ the result is true in this case. Now suppose that u is a vertex different from s and that the result holds for all vertices coloured before u. If $\lambda(u) = \infty$ when u is chosen at Step 2 then, since $\lambda(s)$ is finite, there must be a vertex u' which is the first to be coloured having label $\lambda(u') = \infty$, and $u' \neq s$. Because of the minimum requirement in Step 2 when u' is chosen in Step 2 all remaining uncoloured vertices v have $\lambda(v) = \infty$. Moreover since u' is the first, all vertices v' coloured before u' have $\lambda(v')$ finite. This implies that there is no edge e from such a v' to either u' or any uncoloured v, because otherwise $\lambda(u')$ (or $\lambda(v)$) would be changed to the finite $\lambda(v') + w(e)$. It follows that

there is no path from s to u since $\lambda(s)$ is finite while $\lambda(u) = \infty$. Thus $\delta(u) = \infty$, as required.

It remains to consider the case where $\lambda(u)$ is finite. By the previous theorem, $\lambda(u)$ is the length of some path from s to u. Thus $\lambda(u) \geq \delta(u)$. We have to show that $\lambda(u) > \delta(u)$ is impossible. Let a shortest path from s to u be given by $s = v_0 v_1 \dots v_k = u$. Then, denoting the edge from v_{i-1} to v_i by e_i (for $1 \leq i \leq k$), its length is

$$\sum_{i=1}^{k} w(e_i) = \delta(u).$$

Let v_j denote the last vertex on this path to be coloured before u. Then, by the induction hypothesis,

$$\lambda(v_j) = \delta(v_j) = \sum_{i=1}^{j} w(e_i).$$

If $v_{j+1} \neq u$ $(= v_k)$, then $\lambda(v_{j+1}) \leq \lambda(v_j) + w(e_{j+1})$ after v_j has been coloured (because of Step 4). Since labels only decrease if they change at all in step 4, when u is coloured $\lambda(v_{j+1})$ still satisfies this inequality. We have [1] $\lambda(v_{j+1}) \leq \lambda(v_j) + w(e_{j+1}) = \delta(v_j) + w(e_{j+1}) = \delta(v_{j+1}) \leq \delta(u)$. Thus if $\lambda(u) > \delta(u)$ we would get $\lambda(v_{j+1}) < \lambda(u)$ which, since v_{j+1} is not coloured before u, contradicts the minimality assumption for u in Step 2. Thus if $v_{j+1} \neq u$ we must have $\lambda(u) = \delta(u)$, as required. Finally if $v_{j+1} = u$ then as before $\lambda(u) = \lambda(v_{j+1}) \leq \lambda(v_j) + w(e_{j+1}) = \delta(v_j) + w(e_{j+1}) = \delta(u)$, giving again $\lambda(u) \leq \delta(u)$ and so $\lambda(u) = \delta(u)$ as required. □

The method of proof used in Theorem 2.17 actually produces a shortest path. It is a backtracking technique: starting at the final label assigned to t we go back along the temporary (previously assigned) labels for t until we get a change; we then move to the vertex which has caused this change and do a similar backtrack on its labels until we find a change, and so on. The vertices found in this way then give us a shortest path. We illustrate this using the table on page 74. Here the vertices of the shortest path are boxed.

	vertices v						T
	s	a	b	c	d	t	
	0	∞	∞	∞	∞	∞	$\{s,\ a,\ b,\ c,\ d,\ t\}$
stepwise		18	∞	15	∞	∞	$\{\ \ a,\ b,\ c,\ d,\ t\}$
$\lambda(v)$		**18**	29		22	∞	$\{\ \ a,\ b,\ \ \ d,\ t\}$
values			27		22	∞	$\{\ \ \ \ b,\ \ \ d,\ t\}$
			27			58	$\{\ \ \ \ b,\ \ \ \ \ t\}$
						55	$\{\ \ \ \ \ \ \ \ \ \ t\}$

[1] It is at this stage in the proof we use our assumption that the weights of the edges are non-negative:

$$\delta(u) = \sum_{i=1}^{k} w(e_i) = w(e_1) + \dots + w(e_{j+1}) + w(e_{j+2}) + \dots + w(e_k) \geq w(e_1) + \dots + w(e_{j+1})$$

since $w(e_{j+2}), \dots, w(e_k)$ are all non-negative.

In detail, $\lambda(t)$'s change from 58 to 55 was caused by b, $\lambda(b)$'s change from 29 to 27 was caused by a, $\lambda(a)$'s change from ∞ to 18 was caused by s, so the shortest path is $s\ a\ b\ t$. (Check that it has length 55 from Figure 2.26.)

Exercises for Section 2.5

2.5.1 Carry out the BFS algorithms on the graphs of Figure 2.27 to find the length of a shortest path from vertex a to vertex z, an example of such a shortest path, and the number of shortest paths from a to z.

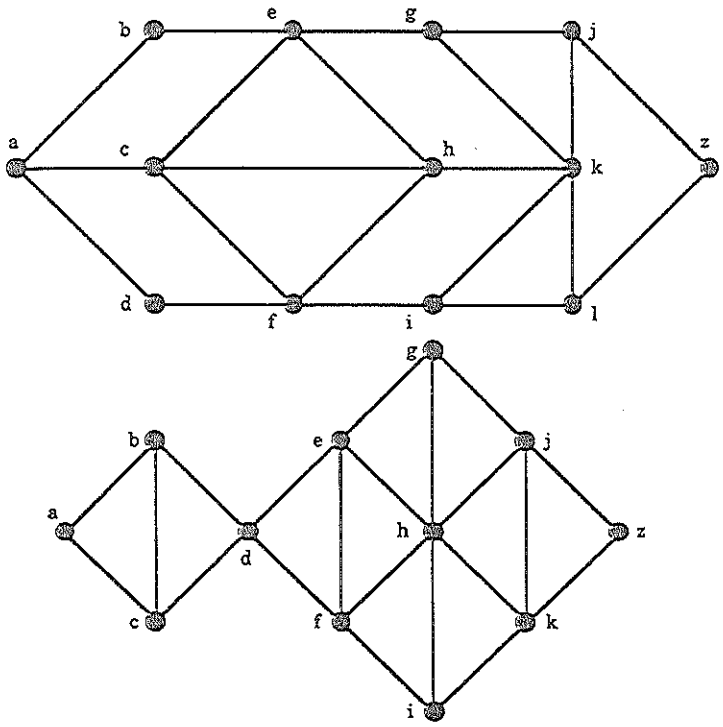

Figure 2.27

2.5.2 Use Dijkstra's algorithm on the connected weighted graphs of Figure 2.28 to find the length of shortest paths from the vertex a to each of the other vertices and to give examples of such paths. Set out your answer in tabular form as explained in the text.

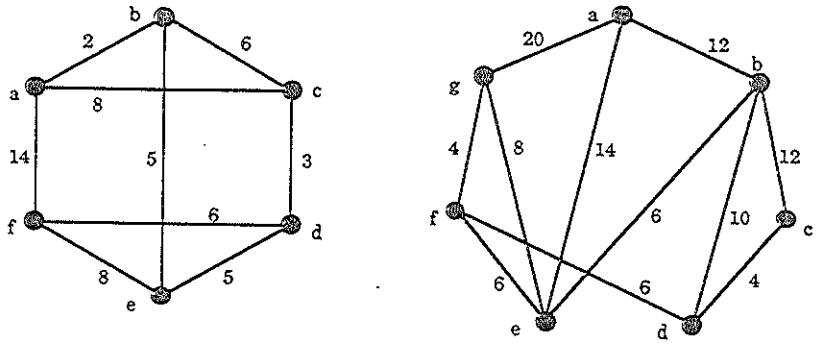

Figure 2.28

2.6 Cut Vertices and Connectivity

We begin this section by studying the vertex analogue of a bridge.

A vertex v of a graph G is called a **cut vertex** (or **articulation point**) of G if $\omega(G-v) > \omega(G)$.

In other words, a vertex v is a cut vertex of G if its deletion disconnects some connected component of G, thereby producing a subgraph having more connected components than G has. For example, in Figure 2.29, v is a cut vertex of G_1 since $\omega(G_1) = 1$ while $\omega(G_1 - v) = 3$. On the other hand, the graph G_2 has no cut vertices.

Cut vertices can be characterised using paths, as we now see.

Theorem 2.19 *Let v be a vertex of the connected graph G. Then v is a cut vertex of G if and only if there are two vertices u and w of G, both different from v, such that v is on every u–w path in G.*

Proof First let v be a cut vertex of G. Then $G - v$ is disconnected and so there are vertices u and w of G which lie in different components of $G - v$. Thus, although there is a path in G from u to w, there is no such path in $G - v$. This implies that every path in G from u to w contains the vertex v, as required.

Conversely, suppose that u and w are two vertices of G, different from v, such that every path in G from u to w contains v. Then there can be no path from u to w in $G - v$. Thus $G - v$ is disconnected (with u and w lying in different components). Hence v is a cut vertex, as required. □

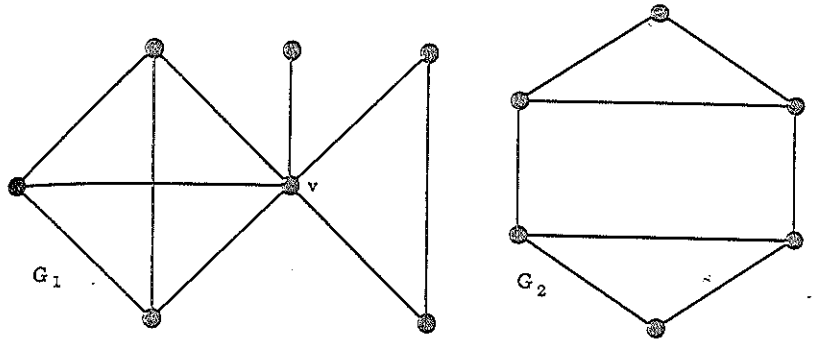

Figure 2.29

Clearly no vertex of a complete graph is a cut vertex. On the other hand, if P_n is the path of length n, where $n \geq 3$, then taking u and w to be the end vertices of P_n in Theorem 2.19, we see that every vertex v, apart from u and w, is a cut vertex. In other words, all but two vertices of P_n are cut vertices.

There is no graph in which every vertex is a cut vertex, as can be seen from our next result.

Theorem 2.20 *Let G be a graph with n vertices, where $n \geq 2$. Then G has at least two vertices which are not cut vertices.*

Proof Clearly we may suppose that G is a connected graph. We proceed by assuming the result is false for our G and so the proof will be complete if we derive a contradiction from this.

Thus we are assuming that there is at most one vertex in G which is not a cut vertex. Now let u, v be vertices in G such that the distance $d(u,v)$ between them is the greatest of distances between pairs of vertices in G, i.e., $d(u,v) = \text{diam}(G)$. (See Exercises 1.6.8, 1.6.9.) Since G is connected and has at least two vertices, $u \neq v$. Thus, by our assumption, one of these two vertices must be a cut vertex, say v. Then $G - v$ is disconnected and so there is a vertex w in G which does not belong to the same component as u does in $G - v$. This implies that every uw path in G contains the vertex v.

It follows from this that the shortest path in G from u to w contains the shortest path from u to v and this contradiction completes our proof. □

If a connected graph G has a cut vertex v then the connectedness of G is vulnerable at v. If G has no cut vertex then its connectedness is not vulnerable to a deletion of one of its vertices. We now put a measure on this vulnerability.

Let G be a simple graph. The **(vertex) connectivity** of G, denoted by $\kappa(G)$, is the smallest number of vertices in G whose deletion from G leaves either a disconnected graph or K_1.

For example, since the graph G_2 of Figure 2.29 has no cut vertex but a disconnected subgraph is produced when the vertices u and v are deleted, it follows that $\kappa(G_2) = 2$. We leave it to the reader to see that $\kappa(G_3) = 3$ and $\kappa(G_4) = 4$ for the graphs G_3 and G_4 of Figure 2.30.

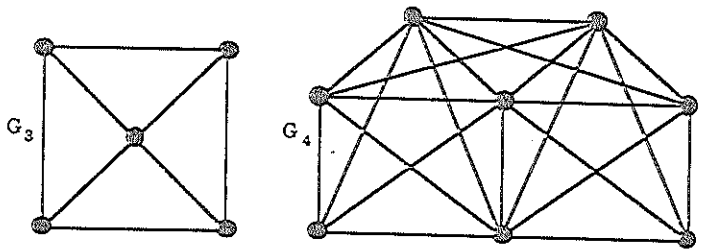

Figure 2.30: $\kappa(G_3) = 3$ and $\kappa(G_4) = 4$.

For $n \geq 2$, the deletion of any vertex from K_n results in K_{n-1} and in general the deletion of t vertices (where $t < n$) results in K_{n-t}. This shows that $\kappa(K_n) = n - 1$.

It is also easy to see that a connected graph G has $\kappa(G) = 1$ if and only if either $G = K_2$ or G has a cut vertex. Moreover $\kappa(G) = 0$ if and only if either $G = K_1$ or G is disconnected.

A simple graph G is called **n-connected** (where $n \geq 1$) if $\kappa(G) \geq n$.

It follows that G is 1-connected if and only if G is connected and has at least two vertices. Moreover G is 2-connected if and only if G is connected with at least three vertices but no cut vertices. We finish this chapter with a characterisation of 2-connected graphs due to Whitney [64]. First we need a definition.

Let u and v be two vertices of a graph G. A collection $\{P_{(1)}, \ldots, P_{(n)}\}$ of $u - v$ paths is said to be **internally disjoint** if, given any distinct pair $P_{(i)}$ and $P_{(j)}$ in the collection, u and v are the only vertices $P_{(i)}$ and $P_{(j)}$ have in common.

Theorem 2.21 (Whitney, 1932) *Let G be a simple graph with at least three vertices. Then G is 2-connected if and only if for each pair of distinct vertices u and v of G there are two internally disjoint $u - v$ paths in G.*

Proof Suppose first that any pair of distinct vertices is connected by a pair of internally disjoint paths. Then clearly G is connected so it remains to prove that G has no cut vertices. Assume, to the contrary, that v is a cut vertex of G. Then, by Theorem 2.19, there are two vertices u and w of G, both different from v, such that v is on every $u - w$ path in G. However, by the hypothesis there are two internally disjoint $u - w$ paths in G and at most one of these can then pass through v. Thus v is *not* on every $u - w$ path, a contradiction. Hence G has no cut vertices.

Now conversely suppose that G is 2-connected. Let u and v be a pair of distinct vertices of G. We use induction on $d(u, v)$, the distance between u and v (see Exercise 1.6.8), to show that there is a pair of internally disjoint $u - v$ paths.

First, if $d(u, v) = 1$ then u and v are joined by an edge, say e. It follows from Exercise 2.6.2 that e is not a bridge, since G has no cut vertices. Hence, there is a $u - v$ path P different and so internally disjoint from the $u - v$ path Q given by the single edge e.

We now assume that $d(u, v) = k \geq 2$ and that if x and y are any pair of vertices with $d(x, y) < k$ then there are two internally disjoint $x - y$ paths. Let P be a path of length k from u to v and let w be the second last vertex of P. Then $d(u, w) = k - 1$ and there are two internally disjoint $u - w$ paths, say Q_1 and Q_2.

Since G is 2-connected, w is not a cut vertex, i.e., $G - w$ is connected and so there is a $u - v$ path P' which does not pass through w. Let x be the last vertex of P' which is also a vertex of either Q_1 or Q_2 (such a vertex exists since u is common to all three paths). See Figure 2.31 for an illustration of this.

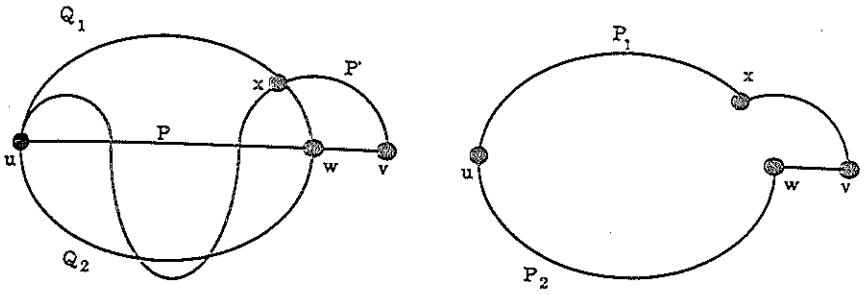

Figure 2.31

Suppose that $x \in Q_1$. Let P_1 be the $u - v$ path given by the $u - x$ section of Q_1 followed by the $x - v$ section of P'. Let P_2 be the $u - v$ path given by the path Q_2 followed by the edge wv. Then P_1 and P_2 are internally disjoint, by the definition of Q_1, Q_2, x and P'. (See Figure 2.31.) The proof now follows by induction. □

Corollary 2.22 *Let u and v be two vertices of the 2-connected graph G. Then there is a cycle C of G passing through both u and v.*

Proof Let P_1 and P_2 be two $u - v$ internally disjoint $u - v$ paths, as guaranteed by Theorem 2.21. Then $P_1 \cup P_2$ gives a cycle C containing both u and v, as required.

Exercises for Section 2.6

2.6.1 Prove that a vertex v of a tree T is a cut vertex if and only if $d(v) > 1$.

2.6.2 Let G be a connected graph with at least three vertices. Prove that if G has a bridge then G has a cut vertex.

2.6.3 Find $\kappa(G)$ for the graphs G of Figure 2.32. If $\kappa(G) = 1$, identify the cut vertices of G.

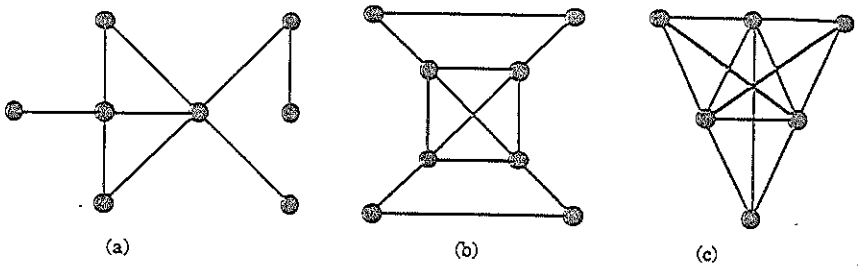

Figure 2.32

2.6.4 Give an example of a simple connected graph G with n vertices having a cut vertex v such that $\omega(G - v) = n - 1$ and each connected component of $G - v$ consists of an isolated vertex.

2.6.5 Let T be a tree with at least three vertices. Prove that there is a cut vertex v of T such that every vertex adjacent to v, except for possibly one, has degree 1.

2.6.6 Let v be a cut vertex of the simple connected graph G. Prove that v is *not* a cut vertex of its complement $\overline{G}$. (Hint: see Exercise 1.6.12.)

2.6.7 Let G be a simple connected graph with at least two vertices and let v be a vertex in G of smallest possible degree, say k.

(a) Prove that $\kappa(G) \leq k$.

(b) Prove that $\kappa(G) \leq 2e/n$, where e is the number of edges and n is the number of vertices in G.

2.6.8 Let $v_1, v_2, \ldots, v_n$ be n distinct vertices of the n-connected graph G and form the supergraph H of G by introduction of a new vertex v, not in G, which is adjacent to each of $v_1, v_2, \ldots, v_n$. Prove that H is n-connected.

2.6.9 Let G be an n-connected graph and let H be the join $G + K_1$. Prove that H is $(n + 1)$-connected.

Chapter 3

Euler Tours and Hamiltonian Cycles

3.1 Euler Tours

Recall, from Section 1.6, that a trail in a graph G is a walk in G in which the edges are distinct, i.e., no edge of G appears in the trail more than once.

A trail in G is called an **Euler trail** if it includes every edge of G.

Thus a trail is Euler if each edge of G is in the trail exactly once.

A **tour** of G is a closed walk of G which includes every edge of G at least once.
An **Euler tour** of G is a tour which includes each edge of G *exactly* once.

Thus an Euler tour is just a closed Euler trail.

A graph G is called **Eulerian** or **Euler** if it has an Euler tour.

For example, the graphs G_1 and G_2 of Figure 3.1 have an Euler trail and an Euler tour respectively.

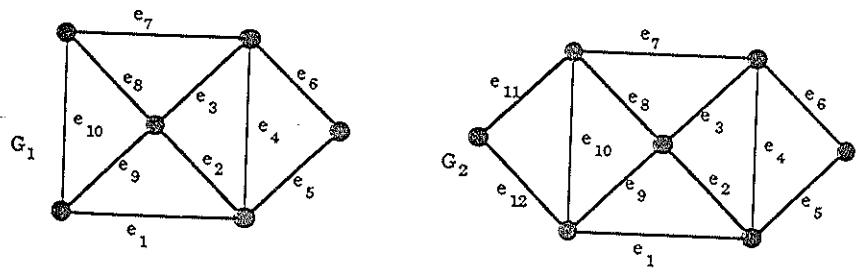

Figure 3.1: G_1 has an Euler trail. G_2 is an Euler graph.