British Journal of Mathematics & Computer Science
4(5): 612-666, 2014

SCIENCEDOMAIN *international*
www.sciencedomain.org

Applications of Loop Trees

Ward Douglas Maurer[1*]

[1]*Department of Computer Science, George Washington University, Washington, DC 20052, USA.*

Original Research Article

Received: 03 July 2013
Accepted: 22 September 2013
Published: 28 November 2013

Abstract

Aims: In a previous paper we have obtained a result which provides a new way to consider structured programs. Any directed graph whatsoever, according to this result, is a dag overlaid by a structured graph, with loops within loops in which no loops overlap. In such a structured graph, the only backward edges go from somewhere in a loop to the head of that loop. Crucial to this result is the construction of what we call a loop tree. As suggested by R. E. Tarjan, we here apply this result to three well-known situations. We also compare our decomposition method to two earlier such methods, one by Tarjan and one, much older, by Luce.
Methodology: We use the conventions of classical mathematics, in which sets and functions underlie all structures, such as directed graphs and loop trees, and in which all facts obtained in the course of the work are presented as theorems and lemmas, based on definitions and accompanied by valid proofs.
Results: We give a method of solving the single-source path expression problem for a reducible graph by examining its loop tree, which must be unique. We give a necessary and sufficient loop tree condition for a graph to have two edge-disjoint spanning trees, and a necessary and sufficient loop tree condition for a graph to have a feedback vertex.
Conclusion: The study of loop trees can be used to clarify many situations in the theory of directed graphs, in addition to the complete classification of directed graphs mentioned above.

Keywords: Loop trees, path expressions, edge-disjoint spanning trees, feedback vertices, strongly connected components, wheels within wheels.

1 Introduction

This paper is concerned with several applications of the concept of a loop tree, introduced by the author in [1]. In order to make this presentation self-contained, we review basic loop tree notation here. In this paper, by a graph we shall always mean a directed graph. When an algebraic language program P is compiled, producing object code with a flowgraph G, loops in P do not necessarily correspond to strongly connected subsets of G. Nevertheless, we argue that, from a semantic standpoint, an outer loop L of G is a strong component of it which is non-trivial. (A strong component of a graph is called **trivial** if it has just one vertex and no edges.)

**Corresponding author: maurer@gwu.edu;*

An **outer loop** L of a graph G is a non-trivial strong component of it. An **entry point** of L is a vertex $y \in$ L such that there is an edge in G to y from some $x \notin$ L. A **head** of L, in a rooted graph G, is either an entry point or the root of G. Here L always contains a head, of one of these two kinds, which are mutually exclusive; it might contain more than one entry point. Given an outer loop L with a choice of head h, a **loopback** of L is an edge leading to h from somewhere in L, and the **body** B of L is the result of removing all loopbacks from L. The non-trivial strong components of B are then outer loops of B, and, by definition, first-level **inner loops** of G. These may contain further loops, and so on, all of these being higher-level inner loops of G.

A **loop tree** of G is a tree T whose root is G; in which every outer loop of G is a child of G; and in which every outer loop of the body B of any vertex L $\neq$ G in T is a child of L. If G itself is strongly connected, it has one child in T, namely G itself. In all other cases, the children of a node U in T are the non-trivial strong components of the body of U, with respect to some head of U. The fundamental theorem of loop trees [1] states that every graph G whatsoever has at least one loop tree, having several further properties which we use below. There is a linear ordering of the nodes of G as P_1, ..., P_n, in such a way that, for every edge E = ($P_i \rightarrow P_j$) in G, either $i < j$ or else E is a loopback, as defined above. With respect to this ordering, every loop in G is induced by a contiguous subset {P_u, P_{u+1}, ..., P_v} of the nodes, with head P_u, where $u \leq v$. Two loops cannot have the same head, and loops are properly nested, so that, for any two loops, either they are disjoint or one is contained in the other. Since a loop might contain more than one entry point, G might have more than one loop tree. However, it is stated in [1], and proved in [2], that G has a unique loop tree if and only if it is reducible, in the sense of Allen and Cocke. The fundamental theorem of loop trees is here applied in the following five ways.

1.1 Path Expressions

All paths from the start node s of a graph G to an arbitrary node v in G form the language σ(P(s, v)) generated by a specific regular expression P(s, v). Such a regular expression is useful in carrying out global flow analysis and in solving shortest path problems and sparse systems of linear equations. In section 2 below, we present a relation between loop trees and such path expressions. Specifically, given a loop tree of a reducible graph, we show how to generate all P(s, v) for fixed s and all v. We also briefly discuss an extension of this technique to non-reducible graphs.

1.2 Edge-Disjoint Spanning Trees

A **spanning tree** of a graph G is a subgraph of G which is a directed tree, and which contains all vertices of G. Two spanning trees of G are **edge-disjoint** if they have no edges in common. It is known that two edge-disjoint spanning trees of a rooted graph G, both having the same root r, exist if and only if G has no bridges, where a **bridge** is an edge v-w which is included in every path from r to w. In section 3 below, we derive a necessary and sufficient condition for a graph G with a loop tree T to have two such trees, namely that every vertex in G other than r, and every loop in T not containing r, has at least two entry edges. We also show that a graph with a unique loop tree, and having at least two vertices, must have a bridge, and thus cannot have two edge-disjoint spanning trees rooted at r.

1.3 Feedback Vertices

A **feedback vertex** of a strongly connected graph is a vertex contained in every cycle in the graph. Of course, not every graph has such a vertex. A **feedback set** is a set of vertices in a strongly connected graph, such that every cycle in the graph must contain at least one vertex in the set. Feedback vertices and feedback sets have been studied extensively by Smith and Walford [3] and by Garey and Tarjan [4]. In section 4 below, we obtain a necessary and sufficient loop-tree condition for G to have a feedback vertex. The condition is that G have a loop tree T with respect to which G has no parallel loops and no conditional loops, in a sense which we define; and that the innermost loop L of G, which must be unique to T, contains a vertex which covers every other vertex in L. This will then also be true of every other loop tree of G. We also obtain bounds on the size of a minimal feedback set in a graph.

1.4 Clustering Trees

For any strongly connected graph G, we here compare our construction of loop trees with a decomposition tree construction due to Tarjan, to which we refer as a clustering tree. Both of these are tree decompositions in which the root represents G; in which every vertex in the tree represents a strongly connected subgraph of G; and in which, if N_1 is the parent of N_2 in the tree, then G_1 contains G_2, where N_1 represents G_1 and N_2 represents G_2. In section 5 below, we derive a necessary and sufficient condition for a generalization of loop trees to be identical to clustering trees in all cases, and another such condition for these to be identical in at least one case. We then show that every graph has a cost-free extension for which the two types of tree are identical in at least one case.

1.5 Wheels within Wheels

Any strongly connected graph may be decomposed, in at least two ways, into smaller such graphs, which may in turn be decomposed into others, and so on. We have developed one such decomposition; another earlier one was discovered by Luce [5] and later rediscovered by Knuth [6]. In section 6 below, we compare our result with that of Luce, and explicate its advantages over Luce´s decomposition. We use Luce's work to provide a further sharpening of Knuth's decomposition theorem, involving a special kind of graph which we call a chandelier. To make the presentation clearer, we recast Luce's work, using current graph terminology; we also relate this work to further commentary by Chaty and Chein.

1.6 Recent Research

The reader will doubtless note the lack, in this paper, of the usual survey of recent research in the given field. There is a good reason for this. Loop trees were only discovered in 2004, and the fundamental theorem of loop trees was only published, outside of internal technical reports, in 2007. Moreover, loop trees were discovered by accident, in the process of working on improved compiled code for nested recursive procedures. It should make sense, then, that no one has been working on applications of loop trees, other than this author, although we hope that this swiftly changes.

2 Loop Trees and Path Expressions

In this section, by a path, we do not necessarily mean a simple path, except as specified explicitly. Indeed, most of the paths which we consider here may be thought of as resembling execution paths through a flowgraph, each of which may clearly go around a loop more than once, and similarly for several loops. Such execution paths, however, are typically thought of as sequences of statements, which are vertices in the flowgraph; by contrast, paths here, as always in a graph, are sequences of edges, rather than vertices. We are interested mainly in paths of execution which start at one specific node of a graph, called its root or start node. We do not specify a single exit node; rather, we are concerned with all paths from the start node *s* to any other node *v*.

2.1 Introduction to Regular Expressions

Regular expressions here are those built up from variables representing the edges of a graph, using parentheses and the operations $\cup$, $\cdot$, and *, with Λ denoting the null string. They are unrelated to the more general regular expressions found in a programming language such as Perl. It has been known for some time (see, e. g., [7]) that the set of all paths from *s* to *v* is representable as $\sigma(P(s, v))$, where $P(s, v)$ is some regular expression and $\sigma(e)$, in general, is the language generated by the regular expression *e*. We are concerned here with generating the various $P(s, v)$ for an arbitrary graph, and not merely one which is reducible in the sense of Allen and Cocke. Such regular expressions are useful in carrying out global flow analysis and in solving shortest path problems and sparse systems of linear equations.

2.2 Introduction to Loop Trees

Basic loop tree notation was introduced in section 1 above. We will also make use of a specific representation for a loop tree, as described in section 9.1 of [1]. For the purposes of implementing this representation in programming languages of the C family, we now refer to the nodes as P_0, ..., P_{n-1}, rather than as P_1, ..., P_n. If G has *n* nodes, then the representation involves two arrays, called *order* and *loops*. The *order* array gives the new ordering of the graph, with $order[k] = z$ where P_k, as above, is the node whose given index is *z*. The *loops* array specifies where the loops are. If P_k is the head of a loop containing *m* nodes, then $loops[k] = k+m$; if P_k is not the head of any loop, then $loops[k] = k$.

2.3 Generating Path Expressions from a Loop Tree

The problem of finding all of the $P(s, v)$ for fixed *s* and arbitrary *v* is known as the single-source path expression problem. This is by analogy with the single-source shortest path problem, where again *s* is fixed and *v* is arbitrary. We here present a way to solve this problem for a reducible graph G, given its (unique) loop tree as represented above, where *s* is the start node of G. This will be done by a single pass through the nodes of G, given by P_0, ..., P_{n-1} as above, where $s = P_0$. In order to distinguish path expressions in different graphs, and also to avoid two conflicting uses of P, we replace the notation $P(s, v)$ of [7] by $\pi_G(s, v)$, indicating that this expression defines all paths in G, whereas $\pi_K(s, v)$, for a subgraph K of G, defines all those paths which stay within K.

We make use of five rules, denoted by (R1) through (R5) below. We use a stage counter *k*, initialized to zero, and a recursive program C(Y, *h*), where Y is either the entire graph G or the

body B of some loop L in G, having head node with index h. Because of our loop tree representation, L will be of the form $\{P_u, P_{u+1}, ..., P_v\}$, for some $u < v$, where P_u is the head of L; and B, since it is the body of L, contains the same nodes that L does. Here C(B, u) assumes that k is initialized to u; it calculates $\pi_B(P_u, P_w)$ for all w, $u \le w \le v$, and leaves k set to v+1. Similarly, C(G, 0) assumes that k is initialized to 0; it calculates $\pi_G(P_0, P_w)$ for all w, $0 \le w \le n$–1, and leaves k set to n. We describe the general logic of C(Y, h).

At stage k, when we first come to the kth node, we need to calculate $\pi_Y(P_h, P_k)$. We first calculate a temporary value for this; if $h = k$, the temporary value is Λ (the empty string), and we set

$$\pi_Y(P_h, P_h) = \Lambda \tag{R1}$$

If $h \ne k$, then we look at all edges, other than loopbacks, which lead to P_k. Let us refer to these as $X_1 \to P_k$, ..., $X_j \to P_k$. For every $X_i \to P_k$, since it is not a loopback, we must have $X_i = P_a$ for some $a < k$. A path from P_0 to P_k, not ending in a loopback, must therefore be a path from P_h to some $X_i = P_a$, followed by $X_i \to P_k$. However, we already have an expression for all paths from P_h to P_a, since we have already calculated $\pi_Y(P_h, P_m)$ for all m, $h \le m < k$. Our temporary expression for $\pi_Y(P_h, P_k)$ is therefore the union, for $1 \le i \le j$, of all expressions of the form $\pi_Y(P_h, X_i)\cdot(X_i \to P_k)$; that is, it is

$$\pi_Y(P_h, P_k) = \pi_Y(P_h, X_1)\cdot(X_1 \to P_k) \cup ... \cup \pi_Y(P_h, X_j)\cdot(X_j \to P_k) \tag{R2}$$

If there are no loopbacks which lead to P_k, this temporary expression is in fact the permanent expression for $\pi_Y(P_h, P_k)$. If there are loopbacks, it is not, since a path from P_h to P_k might end in a loopback. This will always happen if P_k is the start of a loop L (note that L, being strongly connected, must have at least one loopback); and we can detect this, since we will have $loops[k] = k+z \ne k$. If B is the body of L, then C(Y, h) now proceeds by calling C(B, k). When C(B, k) returns, k has been increased by z; and, for the old value of k, we have calculated:

- permanent values of $\pi_Y(P_h, P_m)$ for $h \le m < k$;
- a temporary value of $\pi_Y(P_h, P_k)$; and
- permanent values of $\pi_B(P_k, P_m)$ for $k \le m < k+z$. From this we must calculate $\pi_Y(P_h, P_m)$

for $h \le m \le k+z$; and the only new calculations here are those for $k \le m \le k+z$. These calculations proceed as follows.

First we calculate $\pi_L(P_k, P_k)$; that is, a expression for all (simple and non-simple) cycles within the loop L which start and end at P_k. To do this, we look at all loopbacks which lead to P_k, again referring to these as $X_1 \to P_k$, ..., $X_j \to P_k$. For every $X_i \to P_k$, since it is a loopback, it leads to P_k from some $X_i = P_a$ in L, and therefore in the body B of L. A simple cycle which starts and ends at P_k must therefore be a path in B from P_k to some $X_i = P_a$, followed by $X_i \to P_k$. However, we already have an expression for all paths from P_k to P_a in B, since we have already calculated $\pi_B(P_k, P_m)$ for all m, $k \le m < k+z$–1. An expression for all simple cycles of this form is therefore the union, for $1 \le i \le j$, of all expressions of the form $\pi_B(P_k, X_i)\cdot(X_i \to P_k)$; that is, it is

$$\pi_B(P_k, X_1)\cdot(X_1 \to P_k) \cup ... \cup \pi_B(P_k, X_j)\cdot(X_j \to P_k)$$

An expression for all cycles, simple and non-simple, of this form, and including the empty cycle Λ, is therefore the * operator of regular expressions applied to this; that is, it is

$$\pi_L(P_k, P_k) = (\pi_B(P_k, X_1)\cdot(X_1 \rightarrow P_k) \cup ... \cup \pi_B(P_k, X_j)\cdot(X_j \rightarrow P_k))^* \quad (R3)$$

From this it is easy to calculate $\pi_L(P_k, P_m)$ for $k \leq m < k+z$. A general path p within L from P_k to P_m will be entirely within B if it contains no loopbacks of L. If it does contain loopbacks of L, then that part of p which follows its last such loopback is within B, and the rest of p is a cycle (simple or non-simple) from P_k to itself. Therefore p must consist of such a cycle, followed by a path within B from P_k to P_m. We therefore have

$$\pi_L(P_k, P_m) = \pi_L(P_k, P_k) \cdot \pi_B(P_k, P_m) \quad (R4)$$

Finally we consider $\pi_Y(P_h, P_m)$ for $k \leq m \leq k+z$. Since P_m is in the loop L, while P_h is not, any path from P_h to P_m must enter L through an entry point. Since G is reducible, L has only one entry point, namely P_k. Therefore any path from P_h to P_m in Y is a path from P_h to P_k in Y followed by a path from P_k to P_m in L, and we have

$$\pi_Y(P_h, P_m) = \pi_Y(P_h, P_k) \cdot \pi_L(P_k, P_m) \quad (R5)$$

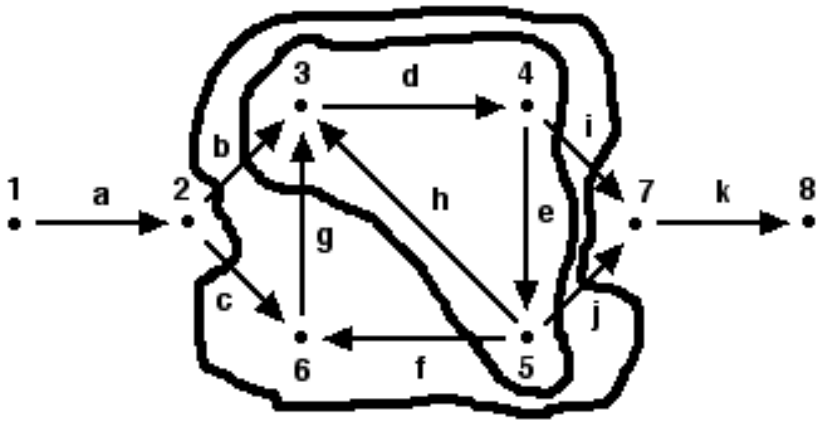


Fig. 2.1 A path expression example

2.4 An Example

The method of the preceding section will now be illustrated, using the graph of Fig. 2.1. This is a reducible graph, with a unique loop tree, since the outer loop (induced by {2, 3, 4, 5, 6, 7, 8}) has exactly one entry point (node 2), while the inner loop (induced by {4, 5, 6}) also has exactly one entry point (node 4). We adopt the following notation:

- G is the entire graph;
- L1 is the outer loop;
- B1 is the body of the loop L1;

- L2 is the inner loop;
- B2 is the body of the loop L2.

The steps in the algorithm are now as follows, keyed to the rules (R1) through (R5):

Y	***h***	***k***	**Rule**	$\pi_Y(P_h, P_k)$
G	1	1	R1	Λ
G	1	2	R2	a (temporary)
B1	2	2	R1	Λ (temporary)
B1	2	3	R2	b
B1	2	4	R2	$b \cdot c$ (temporary)
B2	4	4	R1	Λ (temporary)
B2	4	5	R2	d
B2	4	6	R2	$d \cdot e$
L2	4	4	R3	$(d \cdot e \cdot f)^*$
L2	4	5	R4	$(d \cdot e \cdot f)^* \cdot d$
L2	4	6	R4	$(d \cdot e \cdot f)^* \cdot d \cdot e$
B1	2	4	R5	$b \cdot c \cdot (d \cdot e \cdot f)^*$
B1	2	5	R5	$b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d$
B1	2	6	R5	$b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d \cdot e$
B1	2	7	R5	$b \cdot c \cdot (d \cdot e \cdot f)^* \cdot g$
B1	2	8	R5	$b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d \cdot i \cup b \cdot c \cdot (d \cdot e \cdot f)^* \cdot g \cdot h = b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h)$
L1	2	2	R3	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^*$
L1	2	3	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b$
L1	2	4	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^*$
L1	2	5	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d$
L1	2	6	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d \cdot e$
L1	2	7	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot g$
L1	2	8	R4	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup \cdot g \cdot h)$
G	1	2	R5	$(b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^*$
G	1	3	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b$
G	1	4	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^*$
G	1	5	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d$
G	1	6	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot d \cdot e$
G	1	7	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot g$
G	1	8	R5	$a \cdot (b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h) \cdot i)^* \cdot b \cdot c \cdot (d \cdot e \cdot f)^* \cdot (d \cdot i \cup g \cdot h)$

2.5 Non-Reducible Graphs

In order to extend this method to a graph G which is not reducible, it is necessary to consider all possible loop trees of G. In the worst case, there may be an exponential number of such loop trees. This is therefore not a reasonable method in this case, since Tarjan has given, in [7], a method which runs in polynomial time.

3 Edge-Disjoint Spanning Trees

3.1 Introduction

In what follows, all graphs are taken to be directed. A **rooted graph** G contains a vertex *r*, called its **root**, such that for every vertex *v* in G there is a path from *r* to *v*. A **spanning tree** of G is a subgraph of G which is a directed tree, and which contains all vertices of G. Two spanning trees of G are **edge-disjoint** if they have no edges in common. Such trees have many uses, as noted in [8].

The discovery that every graph whatsoever has an inherent structure of loops within loops, representable by a loop tree [1], raises the question as to what such a loop tree must look like, in a graph having two edge-disjoint spanning trees, each with the root *r*. We here provide a necessary and sufficient condition for this, namely that every vertex in G other than *r*, and every loop in T not containing *r*, has at least two entry edges, as these are specified in Definition 3.1 below. We emphasize that our condition does not provide any improvement in the efficiency of finding two edge-disjoint spanning trees with the same root; indeed, it is already possible to do this in linear time [8]. Rather, our condition becomes a way of visualizing graphs which do, and which do not, have two such spanning trees, in order to improve our informal understanding of them.

We also show that a graph with a unique loop tree, and having at least two vertices, cannot have two edge-disjoint spanning trees rooted at *r*.

3.2 Some Lemmas about Loop Trees

Basic loop tree theory was reviewed in section 1 above. Here we will need four more lemmas about loop trees.

LEMMA 3.1. Given a loop tree T for a graph G, and a cycle C in G, there exists a unique loop L in T such that C both contains the head of L and is itself contained entirely in L.

PROOF. Clearly C is contained in some non-trivial strong component L of G, which is an outer loop of G. Suppose first that C does not contain the head *h* of L. Then we argue by induction on the number of vertices in L; C is completely contained within the body B of L, and therefore within some non-trivial strong component L´ of B, where L´ has fewer vertices than L (note that loops within L´ are also loops within L). Now suppose that C contains *h*, and, as before, is contained entirely in L. Here L is unique in this sense because C is contained neither in any other strong component of G (which must be disjoint from L) nor in any loop contained in L (because such a loop cannot contain *h*). This completes the proof.

LEMMA 3.2. Given a loop tree T of a graph G with root *r*, and any vertex *z* in G, there is a path from *r* to *z* which includes no loopbacks in T.

PROOF. We order the vertices of G as $v_1, v_2, ..., v_n$, according to the fundamental theorem of loop trees. We refer to v_i-v_j as a normal edge if $i < j$; that is, if it is not a loopback. We start by showing that, given any vertex *y* in G other than *r*, there is a normal edge to *y* from some *x* in G. Otherwise, the only edges to *y* would be loopbacks, implying that *y* is the head of some loop L in T.

Therefore, since *y* is not *r*, *y* is an entry point of L; so there must be an edge to *y* from some *x* in G which is not in L. This edge cannot be a loopback, since a loopback to the head of L is always from somewhere in L; therefore it is a normal edge. It follows that there is a normal edge to *z* from some vertex z_1 in G, and then to z_1 from some z_2, and so on back to *r*. This produces a path from *r* to *z* containing normal edges only, that is, containing no loopbacks, and completes the proof.

LEMMA 3.3. Given a loop L in a loop tree T of a graph G with root *r*, and an edge *e*-*h*, where *h* is the head of L but *e* is not in L, there is a path from *r* to *e* which contains no vertices in L.

PROOF. As before, we order the vertices of G as $v_1, v_2, ..., v_n$. If the indices of L are $v_i, v_{i+1}, ..., v_j$, then $h = v_i$, as we have seen. Here *e*-*h* cannot be a loopback, since then both *e* and *h* would have to be in L. Therefore, *e*-*h* is a normal edge, and $e = v_k$ for $k < i$. By Lemma 3.2, there is a path π from *r* to *e* involving no loopbacks; and all the vertices of π must then also have indices less than *i*, and thus cannot be in L. This completes the proof.

LEMMA 3.4. Given a loop L in a loop tree T of a graph G with root *r*, there is a path from *r* to the head *h* of L which contains no vertices in L other than *h*.

PROOF. If $h = r$, we are done. Otherwise, *h* is an entry point, and there is an edge *e*-*h*, where *h* is not in L. The lemma follows by concatenating *e*-*h* to the end of the path from *r* to *e* which exists by Lemma 3.3.

3.3 Bridges in Graphs

A **bridge** in a graph G with root *r* is an edge *v*-*w* in G which is included in every path from *r* to *w*. Our work is based on the following result, cited by Tarjan [8], which notes that it follows from more general work of Edmonds [9].

THEOREM 3.1. A graph with root *r* has two edge-disjoint spanning trees, each with root *r*, if and only if it has no bridges.

This is an immediate corollary of Lemma 1 of [8], which is a bit more general; it says that *every* bridge in G must be in *every* spanning tree with root *r*, and that there always exist two spanning trees, having root *r*, whose only common edges are the bridges in G.

Note that the definition of graphs in [8] does not allow them to have self-loops; however, Theorem 3.1 remains true, even in the presence of self-loops. Let G be a graph with root *r*, and let G´ be the result of eliminating all self-loops from G. A self-loop cannot be in a spanning tree, since a tree contains no cycles; hence G has two edge-disjoint spanning trees if and only if G´ does. This holds, by the theorem, if and only if G´ has no bridges. However, a self-loop can never be a bridge, by definition; so G´ has no bridges if and only if G does not.

The definition of graphs in [8] also allows them to have multiple edges. Loop trees were originally developed for graphs without multiple edges, but the theory is easy to generalize; a strong component containing an edge *v*-*w* must contain any further edges from *v* to *w*. Consider now a graph G with root *r*, and with a loop tree T. Let us replace each bridge *v*-*w* in G by two edges, each going from *v* to *w*, producing a graph G´. Clearly a multiple edge cannot be a bridge, since any path from *r* to *w* could always use a different edge from *v* to *w*; thus G´ has no bridges, and

yet G´ has the same loop tree T that G does. Thus any loop tree whatsoever can be a loop tree of a graph with no bridges, and hence with two edge-disjoint spanning trees rooted at *r*. It follows that the existence of such spanning trees is not derivable from the form of T, if multiple edges are allowed. In what follows, therefore, we assume that our graphs have no multiple edges.

We now consider certain conditions under which a graph with the root *r* has a bridge *v*-*w*. The most obvious case is that in which *w* has in degree 1; so in fact all paths whatsoever, going to *w*, must use the edge *v*-*w*. There is, however, another general case, involving loop trees. Let *w* be the unique head of a loop L, somewhere in a loop tree of a graph. Suppose that there is only one edge, *v*-*w*, which goes to *w* from outside L (although there will be at least one other edge to *w* from inside L). Then *v*-*w* is a bridge. Any path from *r* to *w* must enter L at some point, which must be an entry point of L; and, by assumption, this must be *w*, and the only way to get to *w* from outside L is through the edge *v*-*w*.

Both kinds of bridge are illustrated in Fig. 3.1, where the root is taken to be the vertex A. The edges B-C and C-D are bridges, here, since C and D both have indegree 1. The vertex B has in degree 2, but the edge A-B is still a bridge because it is the only way to get to B from the root (A). Note that D-B is not a bridge, since we can get from A to B without going through D-B.

3.4 A Loop Tree Condition for Bridges

We now give a necessary and sufficient condition for an edge in a rooted graph to be a bridge.

THEOREM 3.2. An edge *v*-*w* in a graph G with root *r* is a bridge in G if and only if either:

(**a**) *w* has indegree 1 in G, or
(**b**) *w* is the unique entry point of a loop L, somewhere in a loop tree in G, and *v* is not contained in L, and *v*-*w* is the only edge to *w* from outside L.

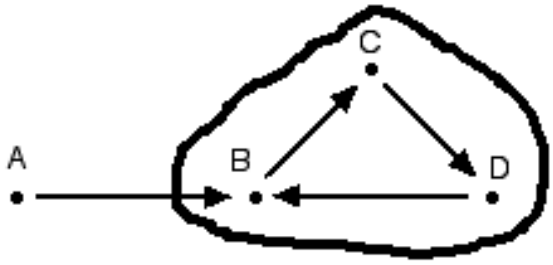


Fig. 3.1. Bridges in a graph

PROOF. We have seen in section 3.3 above that any edge satisfying either of our conditions (**a**) and (**b**) is a bridge. We now show that these are the only two kinds of bridge that G can have. Suppose that *v*-*w* is a bridge. If there is no edge *u*-*w* for $u \neq v$, then we have case (**a**) above. Now suppose that such an edge *u*-*w* exists. Since G is a rooted graph, there is a simple path π from *r* to *u*. Suppose that π did not include *v*; then, by concatenating the edge *u*-*w* to the end of π, we would obtain a path from *r* to *w* which does not include *v*, contradicting our assumption that *v*-*w* is a bridge. Therefore π must include *v*.

In particular, there are simple paths π_1, from r to v, and π_2, from v to u. Here π_1 does not include the edge v-w; otherwise it would have to go back to v, and it would not be a simple path. Suppose now that π_2 does not include the edge v-w. It would then be possible to go from r to v (along π_1), from there to u (along π_2), and then along the edge u-w. This gives a path from r to w that does not include the edge v-w, which also contradicts our assumption that v-w is a bridge. Therefore π_2 includes the edge v-w; and, in particular, there is a path from w to u. This path, followed by the edge from u to w, constitutes a cycle C.

We now show that C cannot include v. Suppose the contrary; then, in particular, there are paths π_3, from w to v, and π_4, from v to u. Here π_4 does not include w, and thus does not include the edge v-w. Thus π_1, from r to v, followed by π_4, from v to u, followed by the edge u-w, provides a path from r to w without including the edge v-w, which is contained in neither π_1 nor π_4. Once more this contradicts our assumption that v-w is a bridge; therefore C does not include v.

Now consider a loop tree T for G. Let L be the loop which exists by Lemma 3.1, so that C contains the head h of L and is itself entirely contained within L. We show that L cannot include v. Suppose the contrary; then, since C does not include v, there is a path π_5 from h to w, along C, that does not include v. By Lemma 3.4, there is always a path π_6 from r to h which includes no vertices in L other than h, and which, therefore, also does not include v (note that $v \neq h$, since C contains h but not v). Concatenating π_6 and π_5, we obtain a path from r to w that does not include v, again contradicting our assumption that v-w is a bridge.

Since w is in L, while v is not, it follows, by definition, that w is an entry point of L. We now show that w is the only entry point of L. Suppose that L has another entry point $q \neq w$. Consider a second loop tree T´, for which q is the head of L. By Lemma 3.4, there is then a path π_7 from r to q which contains no vertices in L other than q. Therefore, π_7 does not contain w and thus does not contain the edge v-w. Since L is strongly connected, there is a path π_8 in L from q to w. Here π_8 also does not contain the edge v-w, since π_8 is entirely within L, while v is outside L. Concatenating π_7 and π_8, we obtain a path from r to w which does not contain the edge v-w; and once more this contradicts our assumption that v-w is a bridge.

Finally we show that v-w is the only edge to w from outside L. Suppose that there is an edge e-w from outside L, where $e \neq v$. By Lemma 3.3, there is a path π_9 to e from r which contains no vertices in L, and therefore does not include the edge v-w. Concatenating the edge e-w to π_9, we obtain a path from r to w that does not include the edge v-w. Again this contradicts our assumption that v-w is a bridge, and completes the proof.

3.5 The Main Result on Edge-Disjoint Spanning Trees

We now provide a necessary and sufficient condition for there to be two edge-disjoint spanning trees in the graph G, with the same root r; or, what is the same thing (by Theorem 3.1), for there to be no bridges in G. This is based on the following definition.

DEFINITION 3.1. Given a graph G with a loop tree T, an **entry edge of a vertex** *w* in G is an edge to *w* from some vertex $v \neq w$. An **entry edge of a loop** L in T is an edge to some vertex *w* in L from some vertex *v* outside L.

THEOREM 3.3. A graph G with root *r* and loop tree T has no bridges (and therefore has two edge-disjoint spanning trees with root *r*) if and only if every vertex of G other than *r*, and every loop of T not containing *r*, has at least two entry edges.

PROOF. It suffices to show that the condition above holds if and only if G has no bridge of either type (**a**) or (**b**), as specified in Theorem 3.2. Let T be a loop tree of G, and let *w* be a vertex in G, with $w \neq r$. Since G has the root *r*, there is a simple path from *r* to *w* in G. Let *v*-*w* be the last edge on this path; here we must have $v \neq w$. If *w* has indegree 1, then *w* has only one entry edge. If *w* is the unique entry point of a loop L in T, and *v* is not contained in L, and *v*-*w* is the only edge to *w* from outside L, then L has *v*-*w* as its only entry edge, and L does not contain *r* (since otherwise it could not contain the entry point *w*). Conversely, suppose that *w* does not have at least two entry edges. Since G has the root *r*, and $w \neq r$, *w* has one entry edge *v*-*w* (with $v \neq w$); hence *v*-*w* is a bridge of type (**a**). Likewise, if L does not contain *r*, and does not have at least two entry edges, it must have one entry edge, say *v*-*w*. In that case *w* is the only entry point of L, and *v* is not contained in L, and *v*-*w* is the only edge to *w* from outside L (otherwise *w* would have two entry edges from vertices unequal to *w*). Therefore, *v*-*w* is a bridge of type (**b**), completing the proof.

3.6 Innermost Loops and Multiple Entry Points

Theorem 3.3 allows two entry edges to the same entry point of a loop. We now show that this is not allowed for innermost loops, where we disregard self-loops in taking a given loop to be innermost. This result will be needed in our treatment, in section 3.7 below, of a graph having a unique loop tree.

LEMMA 3.5. A graph G with root *r*, and with no bridges, and with at least one vertex other than *r*, must have at least one loop not containing *r*, and not a self-loop.

PROOF. Let v_1 be any vertex of G, other than *r*. Here v_1 must have at least two entry edges, by Theorem 3.3, and at least one of these is from a vertex (call it v_2) other than *r*, since G has no multiple edges. Similarly, v_2 must have at least two entry edges; and at least one of these is from a vertex (call it v_3) other than *r*, and so on. This produces a sequence v_k of vertices, with an edge from v_{k+1} to v_k for all $k \geq 1$. Since G is finite, this sequence must eventually repeat, say $v_i = v_j$, for $i < j$, so that v_j-v_{j-1}-...-v_i (= v_j) is a cycle C, not containing *r*.

Let B be the graph obtained from G by removing any edges of G which lead to *r*. Here C is contained in B, and therefore in some strong component L of B; and L does not contain *r*, since it contains no edges which lead to *r*. There are now two cases; L may be a strong component of G, or, if it is not, it is contained in some strong component of G. This must have *r* as its only additional vertex; and thus *r* is its head and L is its body. In either case, L is a loop in G, not containing *r*. Here $v_2 \neq v_1$ in the above, by Definition 3.1, since the edge from v_2 to v_1 is an entry edge of v_1; so L is not a self-loop. This completes the proof.

The condition that G have at least one vertex other than *r* is clearly necessary here; indeed, if G has just the vertex *r*, and no edges, then it has no bridges, and no loops.

THEOREM 3.4. Let G be a graph with root *r*, and with a loop tree T. If G has no bridges, then any loop L in T, other than a self-loop, and containing no inner loops of its own, other than self-loops, either contains *r* or has at least two entry points.

PROOF. Let L be as in the statement of the theorem, except that it has only one entry point *w* and does not contain *r*. Since L is not a self-loop, it contains some vertex $z \neq w$. By Theorem 3.3, L has two entry edges; these are then edges *u*-*w* and *v*-*w*. Since G has no bridges, it has two edge-disjoint spanning trees T_1 and T_2, with root *r*, by Theorem 3.1. Since T_1 is a spanning tree, it includes *z*, so that there is a simple path π_z from *r* to *z* in T_1. Since *z* is in L, and *w* is the only entry point of L, π_z must enter L at *w*. Let π_z' be that part of π_z which leads from *w* to *z*. Then π_z' is also a simple path, and thus does not include *w* except at the start; therefore π_z' does not include any loopbacks of L. Also, π_z' never leaves L, because otherwise it would have to re-enter L at *w*, since *w* is the only entry point of L. Therefore π_z' is completely contained in the body B of L. Since this is true of any *z* in L, the union of all π_z' over all *z* in L defines a spanning tree T_1' of B, with root *w*. Applying all these arguments to T_2, we obtain a spanning tree T_2' of B, also with root *w*. The trees T_1' and T_2' are edge-disjoint, since T_1 and T_2 are edge-disjoint. It follows from Theorem 3.1, applied to B (which has the root *w*), that B has no bridges. Since B contains $z \neq w$, it now follows from Lemma 3.5 that B contains a loop L´, not a self-loop, with *w* not in L´. Here L´ is a loop inner to L, contradicting our hypothesis, and completing the proof.

A fundamental property of loop trees is that a loop containing the root *r* cannot contain any entry points at all; this is why that case is excluded from Theorem 3.4. Indeed, a graph with no bridges (such as, for example, a complete graph on more than two vertices) can easily have a loop containing *r*.

3.7 Unique Loop Trees and Edge-Disjoint Spanning Trees

We now show that a graph with root *r*, and with a unique loop tree — that is, either one with no loop at all, or one in which every loop has a unique head — must have at least one bridge, and thus, by Theorem 3.1, cannot have two edge-disjoint spanning trees rooted at *r*. The only exception to this is a graph with just one vertex.

LEMMA 3.6. Let G be a graph with root *r* and loop tree T; let L be a loop in G, which contains *r*; and let B be the body of L. If L has no bridges, then neither does B.

PROOF. If L has no bridges, then it has two edge-disjoint spanning trees T_1 and T_2, rooted at *r*, by Theorem 3.1. But T_1 and T_2 cannot contain loopbacks to *r*; otherwise they would contain cycles, and a tree cannot contain cycles. Thus T_1 and T_2 are also edge-disjoint spanning trees of B, which shows that B has no bridges, again by Theorem 3.1.

THEOREM 3.5. A graph G, rooted at *r*, with a unique loop tree T, and containing at least two vertices, must have a bridge.

PROOF. Suppose the contrary, so that G has no bridges. Since G has at least two vertices, it has at least one loop in T, which is not a self-loop, by Lemma 3.5. Clearly, any self-loop which is a loop in T must be a leaf in T; let us remove from T any such leaves, producing T´. Let L be a leaf in T´. There are now two cases. Suppose first that L does not contain *r*; it then satisfies the conditions of Theorem 3.4, and thus has at least two entry points, say *x* and *y*. Therefore G has at least two loop trees, since there is at least one of these with *x* and with *y* as the head of L. This contradicts our hypothesis.

Now suppose that L contains *r*, so that *r* is the head of L. Let B be the body of L, and consider a bridge *v*-*w* in L, so that any path from *r* to *w* in L contains *v*-*w*. However, any path from *r* to *w* in G must stay entirely in L; otherwise, it would have to re-enter L through an entry point, and, since L contains *r*, it has no entry points. Hence *v*-*w* would in fact be a bridge in G. This contradiction shows that L, in fact, has no bridges; and, by Lemma 3.6, neither does B. Since L is not a self-loop, it contains at least two vertices, as does B. Therefore, by Lemma 3.5 applied to B, it contains a loop L´, not containing *r*, and not a self-loop. Here L´ is a loop inner to L, contradicting our assumption that L is a leaf in T´, and completing the proof.

The condition that G have at least two vertices is necessary, since a graph with just one vertex has a unique loop tree but has no bridges. A self-loop, even in this case, is not a bridge, since it is not on the null path from *r* to itself. Also, a graph with just one vertex *r* does have two edge-disjoint spanning trees, namely T_1, which contains just *r* and no edges, and $T_2 = T_1$. Even though T_1 and T_2 are equal, they are still edge-disjoint; they have no edges in common, since they have no edges at all.

It is possible for a graph, rooted at *r*, and having an outer loop which is a self-loop, to have two edge-disjoint spanning trees, both rooted at *r*, as shown in Fig. 3.2. It is also possible, in a graph having two edge-disjoint spanning trees, for a loop which is not an innermost loop to have a single entry point, as, for example, the loop L2 in Fig. 3.3. In both cases, one spanning tree is shown with bold gray arrows, and the other one with bold black arrows; edges with light black arrows are not in either spanning tree. Note that every vertex in each of these graphs, other than *r*, has at least two entry edges, although *r* has no entry edges at all.

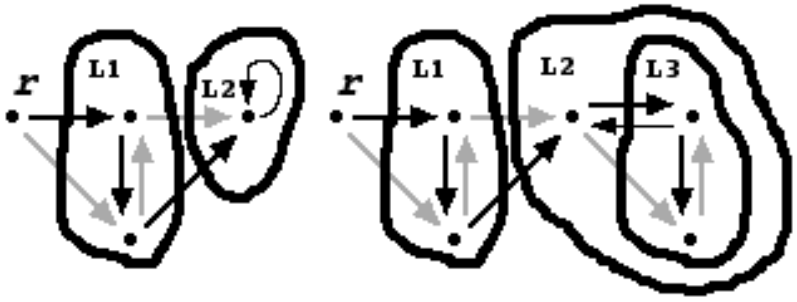


Fig. 3.2. Edge-disjoint spanning tree examples Fig. 3.3. More examples

In the case of a graph G with multiple edges, the most we can say here is that, if G has a unique loop tree and contains at least two vertices, it must have an edge *v*-*w* which is either a bridge, or would be a bridge if all other (multiple) edges from *v* to *w* were removed from G.

4 Feedback Vertices

In this section, vertices are what are often called nodes in a graph. A **feedback set** is a set of vertices in a strongly connected graph, such that every cycle in the graph must contain at least one vertex in the set. If a feedback set contains just one vertex V, then V is known as a **feedback vertex**. The discovery that every graph whatsoever has an inherent structure of loops within loops, representable by a loop tree [1], raises the question as to what such a loop tree must look like, in a graph having a feedback vertex.

We here provide a necessary and sufficient condition for a graph G to have a feedback vertex. The condition is that G have a loop tree T with respect to which G has no parallel loops and no conditional loops, in a sense which we define; and that its innermost loop L, which must be unique to T, contains a vertex which covers every other vertex in L, as this is defined in [3]. We emphasize that our condition does not provide any improvement in the efficiency of finding all feedback vertices in a graph; indeed, it is already possible to do this in linear time [4]. Rather, our condition becomes a way of visualizing graphs which do, and which do not, have feedback vertices, in order to improve our informal understanding of them.

Feedback vertices, and feedback sets in general, have important applications in logic design [3]. They are also useful in the construction of cut points, as part of the proof of correctness of a program [10]. We have done previous work [11] which attempts to develop what would be called, using the terminology above, a "good" feedback set for this purpose.

4.1 Graphs with Feedback Vertices

In order to motivate our theorem, let us first look at a few graphs which have feedback vertices. Clearly, if a graph G contains one and only one cycle, then every vertex in that cycle is a feedback vertex. More generally, if G has only one strong component L, whose body is a dag, then the head of L is a feedback vertex. There are, however, more general such graphs; as, for example, that of the matrix product program of Fig. 4.1. Here there are three nested loops, as indicated by the structured program at the top. This is converted into an unstructured program, whose flowgraph is then given. It should be clear, however, that the vertex containing the statement S = S+A[I,K]*B[K,J] is a feedback vertex. This example may be generalized to allow any number of nested loops, rather than just three.

4.2 Graphs without Feedback Vertices

We now look at some conditions under which a graph cannot have a feedback vertex. The most obvious of these involves what we may call parallel loops; that is, two or more loops in a loop tree that are siblings (that is, they have the same parent). Since such loops are strong components of the body of their common parent, they are disjoint and thus contain disjoint cycles; so a graph having parallel loops cannot have a feedback vertex.

If T contains no parallel loops, then there can be at most one non-trivial strong component L_1 of G. The body B_1 of L_1 can then contain at most one non-trivial strong component L_2, and so on. Every loop tree of such a graph, therefore, is linear in the following sense.

DEFINITION 4.1. A loop tree T of a graph is **linear** if there is some $n \geq 0$ such that T comprises n loops L_1, ..., L_n with each L_i being an inner loop within L_{i-1} for $2 \leq i \leq n$.

Another such condition involves a loop that is not always done when its parent loop is done. We refer to this as a conditional loop, since the most common form of it is an inner loop L´ within an if-statement in its parent loop L. There is thus a cycle C in L which is disjoint from L´. Since L´ also contains at least one cycle, there are two disjoint cycles in the graph, and therefore there cannot be a feedback vertex. It will be necessary, however, to use a slightly more general definition of a conditional loop; instead of requiring that C be in L, we merely require that C be not disjoint from L.

```
FOR I = 1 TO N {
  FOR J = 1 TO N {
    S = 0
    FOR K = 1 TO N
      S = S+A[I,K]*B[K,J]
    C[I,J] = S
  }
}
```

```
(note: since clearly
N > 0 here, the for-
loops above may be
implemented as post-
test loops as below,
rather than the
usual pre-test loop)
```

```
    I = 1
1   J = 1
2   S = 0
    K = 1
3   S = S+A[I,K]*B[K,J]
    K = K+1
    IF (K <= N) GOTO 3
    C[I,J] = S
    J = J+1
    IF (J <= N) GOTO 2
    I = I+1
    IF (I <= N) GOTO 1
    OUTPUT "END"
```

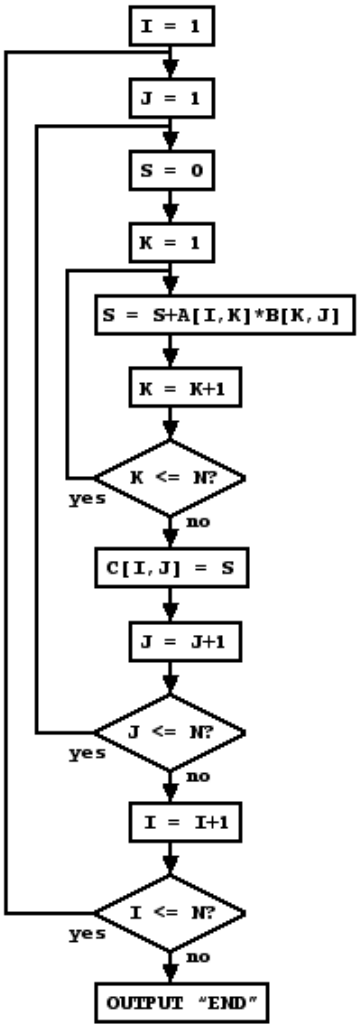


Fig. 4.1. A matrix-product program with a feedback vertex

We may justify this choice informally as follows. Suppose that H is the head of L, and that H is immediately followed by a vertex N which is a conditional exit from L. Thus it is possible to enter L at H, go to N, and take the conditional exit without doing L´. Here, then, L´ is a conditional loop in an informal sense, since L can be entered without doing L´. On the other hand, if L is an outer loop of the program, and there is no cycle in L which is disjoint from L´, there can still be a feedback vertex in L´. However, now suppose that L itself is contained in another loop K. There will then be a path which starts at the head X of K, goes through H and N and back into K, and eventually back to X. This will be a cycle, disjoint from L´, and so G cannot have a feedback vertex.

Accordingly, we make the following definition.

DEFINITION 4.2. A **conditional loop** in a loop tree T of a graph G is a loop L in T, such that there is a cycle C in G, disjoint from L, but not disjoint from the parent of L in T.

The two kinds of conditional loop are illustrated in Figs. 4.2 and 4.3. In Fig. 4.2, the inner loop is conditional in the ordinary sense; it is not done at all, within the outer loop, when the cycle involving **cond1** and **cond2** is done. The structured form of this loop is shown at the top left; a **while** loop inside an **if** statement, which is inside another **while** loop, as here, always leads to this kind of conditional loop. In Fig. 4.3, the loop L3 is not conditional in this sense, but it is still conditional because there is a cycle, A-D-E-A, which is disjoint from L3, but not disjoint from L2, the parent of L3.

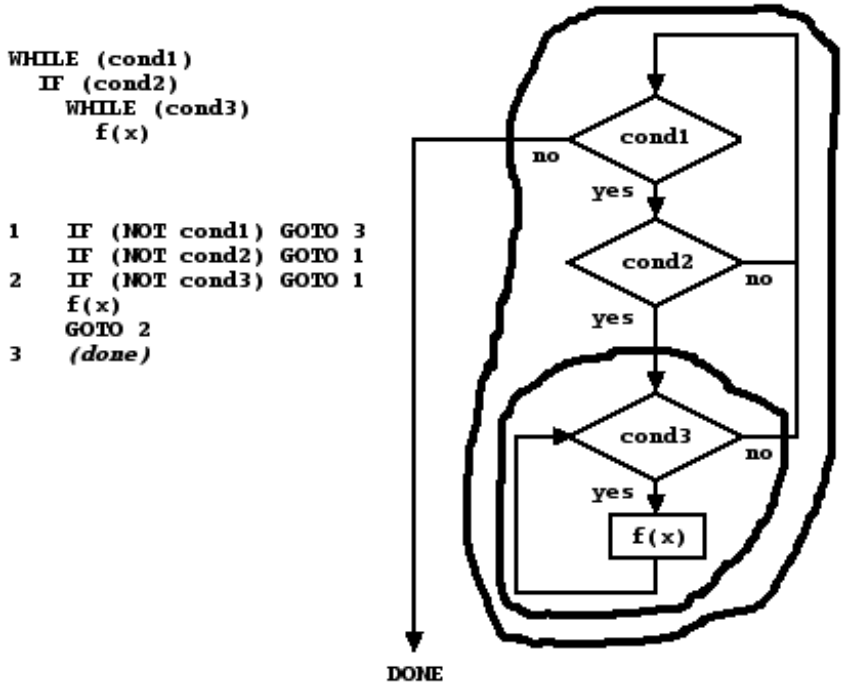


Fig. 4.2. A conditional loop

4.3 Loop Covers

We now extend some terminology used in [3]. A vertex *x* is said to **cover** another vertex *y*, in a graph G, if all cycles in G which contain *y* also contain *x*. A feedback vertex for a graph, then, is one which covers every other vertex in the graph. We here define a local version of this.

DEFINITION 4.3. A vertex *x* is said to **cover a loop** L if it covers every vertex in L.

We will be concerned exclusively with vertices which cover innermost loops. In particular, if such a loop L has exactly one entry point H, then we can easily see that H covers L. In fact, let *y* be a vertex in L, and let C be a cycle containing *y*. If C is not entirely contained in L, it must contain H because H is the only entry point of L, so that C can enter L only through H. If C is entirely contained in L, then again C must contain H, because L is an innermost loop and so the body of L is a dag, which cannot contain the cycle C.

There are, however, other ways that a loop can be covered by one of its vertices, as shown in Fig. 4.4; here that vertex is marked as V, in each case. Note that, even if a loop has only one exit point, that point might not be a loop cover; thus the vertex X in Fig. 4.5, for example, does not cover A, since the cycle A-B-C-A contains A but does not contain X. Another example is shown in Fig.4.6; this is the same graph that was given in Fig.4.3, except that B, rather than D, is now the head of L2. This time, there is no L3, because the body of L2, with B as its head, is a dag. Here there are two disjoint cycles, namely A-D-E-A and B-C-F-B, having non-empty intersections with L2; so no vertex can cover L2.

4.4 The Main Result on Feedback Vertices

THEOREM 4.1. A graph G has a feedback vertex if and only if it has a loop tree T for which all of the following are true:

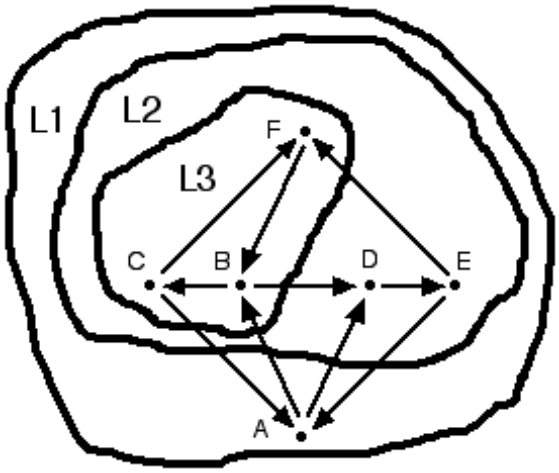


Fig. 4.3. Another kind of conditional loop

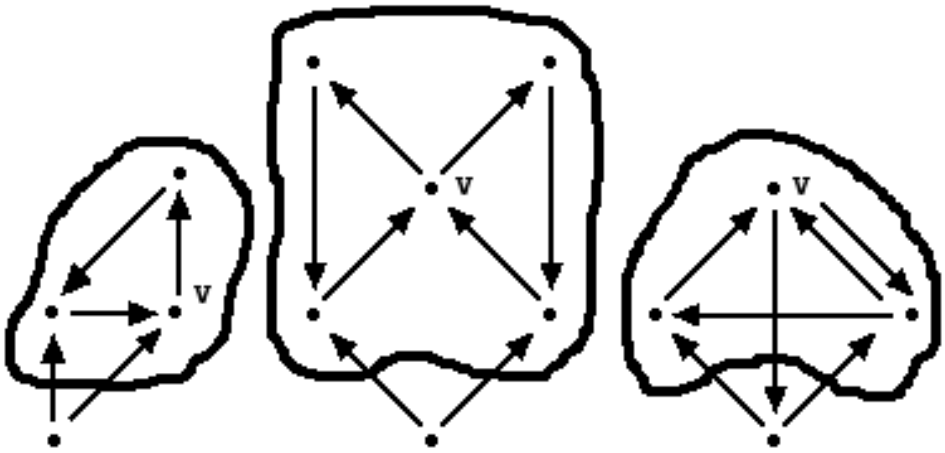


Fig. 4.4. Examples of a loop being covered by one of its vertices

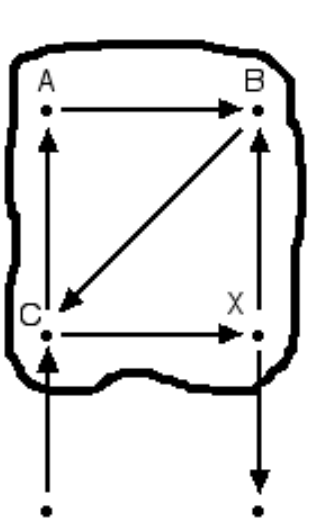


Fig. 4.5. Loops without loop covers

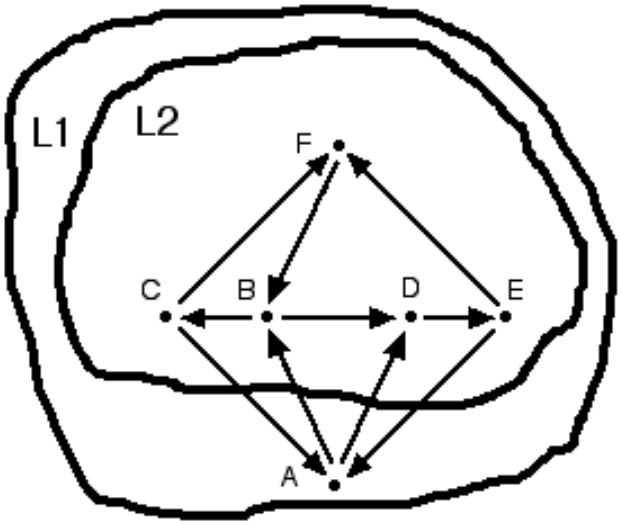


Fig. 4.6. More loops without loop covers

(**a**) G has no parallel loops (and therefore T is linear);
(**b**) G has no conditional loops;
(**c**) The innermost loop of T contains a vertex V which covers it (as in Definition 4.3).

PROOF. We saw in section 4.2 above that, if G has parallel loops, or a conditional loop, it cannot have a feedback vertex. Also, if G has such a vertex V, then V must be in the innermost loop L, in any loop tree. This is because it must, in particular, be contained in every cycle in L; and we know that L contains at least one cycle, because, as a loop, it is non-trivial strongly connected. Here V, being a feedback vertex, must, in particular, cover L. Thus the conditions (**a**)-(**c**) above are necessary; we now show that they are sufficient.

As in Definition 4.1, we denote the loops by L_1 through L_n. Let H_i be the head of L_i, for $1 \leq i \leq n$, and let C be any cycle in G. By Lemma 3.1, there is some L_k such that C contains H_k and is contained entirely in L_k. We now show, by induction on *i*, that C has a non-null intersection with L_i for $k \leq i \leq n$, and thus, in particular, with L_n. By hypothesis, this holds for $i = k$. In general, if C has a non-null intersection with L_i, but not with L_{i+1}, then L_{i+1} is, by Definition 4.2, a conditional loop, contrary to hypothesis. Thus C has a non-null intersection with L_n, and therefore contains V, since V covers L_n. This completes the proof.

4.5 Feedback Vertices and Multiple Loop Trees

The above theorem places strong restrictions on the loop tree T; this does not, however, preclude a graph with a feedback vertex from having more than one loop tree. Indeed, it might have two loop trees with different heights, as shown in Figs. 4.7 and 4.8. Here, if H1 is the head of L1, as in Fig. 4.7, there is no inner loop; if H2 is the head of L1, as in Fig. 4.8, then L1 has the inner loop L2. However, in both cases, the loop tree is linear; and the feedback vertex F (or H1, for that matter) is contained in the innermost loop, in each case, and covers that loop.

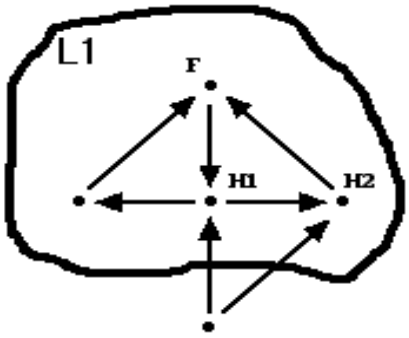


Fig. 4.7. Multiple loop trees

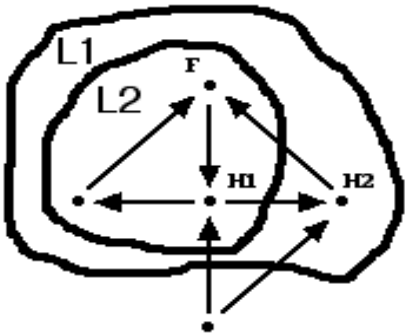


Fig. 4.8. Further multiple loop trees

4.6 Further Investigations

The determination as to whether a feedback set of size *n* exists, within a given graph, is NP-complete [12]. By contrast, a loop tree, as defined in section 1 above, may be found in polynomial time. (The determination of all loop trees remains exponential, although for a low-level reason: the total number of loop trees for a graph may have exponential size; indeed, every permutation of the vertices of a complete graph corresponds to a loop tree for that graph.) In view of these facts, we will start by considering applications of feedback sets.

Two such applications are mentioned in [4]. The first is concerned with the graphs of logic circuits, and is treated in [3]. The second, treated in [10], is directly concerned with flowgraphs, and is also related to some of our own earlier work. We take this up here at some length, and, in particular, we question whether the “best” feedback set, in that context, is the one with minimum size.

Also, in this section, we relate inner loops, in the sense of section 1 above, to Smith and Walford's sets F, G_F, and G_R, introduced in [3]. We show that any graph with a feedback vertex has just one loop in an extended sense, although there might be more than one loop in its loop tree. We obtain a connection, in a special case, between loop trees and the concept of depth first numbering. Finally, we use loop trees to estimate the size of a minimal feedback set.

We note the following terminological differences among these papers:

Garey and Tarjan [4]	Vertex	Arc	Cycle
Smith and Walford [3]	Vertex	Arc	Loop
Alternative notation	Node	Edge	Cycle

4.7 Program Correctness and Feedback Sets

Suppose we wish to prove the correctness of a program having a given flowgraph. We assume that our program is not structured, and may contain arbitrary go-to statements; in loop tree notation, there might be more than one loop tree for the graph. This may be accomplished in eight steps (see [11]):

(1) Identify the terminal vertices of the graph and assign an exit assertion to each one; this is an assertion concerning the variables of the program which will hold at the time that that exit is taken.
(2) Identify the initial vertex of the graph and assign an entry assertion to it; this is an assertion concerning the variables of the program which must hold when the program starts. (If the graph has more than one initial vertex, the entire proof of correctness is redone for each initial vertex.)
(3) Make a statement of correctness of this program: if it starts at the initial vertex with the entry assertion holding, then (**a**) it will not loop endlessly; (**b**) it will not try to execute any statement S at a time when S is not well defined; and (**c**) when it gets to a terminal vertex, the corresponding exit assertion will hold.
(4) Identify intermediate assertion points, or what Manna [10] calls cutpoints; there must be one such point in every cycle of the flowgraph. Associate an intermediate assertion with every such point. Define an assertion point of the program as either an initial vertex, a terminal vertex, or a cutpoint.
(5) Identify the verification paths in the program; these are paths from one assertion point to another, with no assertion points in between. The assertion associated with the first (last) statement in a path is called the initial (final) assertion of that path.
(6) State the verification conditions of the program, one associated with each verification path. Such a condition says that if the path is started with its initial assertion valid, then it will not try to execute any statement S at a time when S is not well defined, and that, when it reaches the end of that path, its final assertion will be valid.
(7) Prove all the verification conditions.
(8) Prove that the program terminates. (The techniques for this are relatively simple extensions of those above; we omit the details.)

Feedback sets arise in step (4) above, since the condition on a set of cutpoints is that every cycle in the flowgraph must contain at least one cutpoint. A simple way to choose cutpoints is to choose them as the destination points of reverse edges in the flowchart, relative to some ordering of the vertices (normally the order in which the corresponding program is written).

4.8 Feedback Set Measures

Let us now consider what makes a "good" set of cutpoints, or feedback set. The treatment given in [3] concerns finding such a set with as few vertices as possible; that of [12] shows that this problem is NP-complete. There are two possible general approaches to this situation. We may try to find a solution to the problem which, although exponential in the general case, "appears to be quite efficient for large graphs arising in practical applications" (see the abstract of [3]). Or we may question whether, for the purposes of program correctness proof, we actually want to minimize the number of vertices in a feedback set.

In [11] we address the problem of finding a "best" set of intermediate assertion points, by which we mean one which minimizes the total length of all verification conditions as described in step (6) above. Since these are what have to be proved, we would like their total size to be as small as possible. Provided that we use internal size only (that is, not counting the lengths of the initial and final assertions in each verification condition, but only the lengths of what arises from the statements in the path), we show that this is minimized by choosing, as cutpoints, exactly the join points; that is, those of indegree greater than 1, in the program. In the present context, this has the further advantage that such points may be determined immediately from the flowgraph.

4.9 Graph Partitions and Loop Trees

In [3], as part of their algorithm for finding minimal feedback sets, Smith and Walford introduce sets F, G_F, and G_R. Here F is a set of vertices of the graph G, and G is then partitioned into G_F and G_R (that is, $G_F \cap G_R = \emptyset$ and $G_F \cup G_R = G$). A vertex is in G_F if and only if it is contained in at least one cycle of G which also contains a vertex in F, and is not contained in any cycle of G which does not contain a vertex of F.

We may note first that F is a feedback set of a strongly connected graph G if and only if the corresponding $G_F = G$ (that is, $G_R = \emptyset$). If F is not a feedback set, then there exists at least one cycle C in G which does not contain any elements of F, and any vertex V in C cannot be in G_F and so must be in G_R. If F is a feedback set and V is a vertex in G, then V is contained in some cycle (since G is strongly connected); any cycle containing V must contain some element of F, by definition of a feedback set, and so V is in G_F, thus implying that $G_F = G$.

Now let the entire graph G be strongly connected, so that it is an outer loop, in our sense. Let F be a one-element set, containing only the head H of G. Then G_R consists precisely of all vertices in inner loops of G. Note that if V is in an inner loop, that loop is in the body B of G and cannot contain H, so V is not in G_F. Conversely, if V is not in G_F, then V is in some cycle of G that does not contain H. That cycle is completely contained in B, and thus in some strong component of B, that is, an inner loop of G.

We may extend this example by including, within F, a single chosen head for each inner loop, in addition to H. The corresponding G_R now consists precisely of all vertices in all second-level inner loops of G. Continuing this process recursively, we arrive at a set F containing the head of every loop in some specific loop tree of G; and this time $G_R = Ø$. This is, therefore, one way of getting a feedback set F (see also section 8 below), although this F is not necessarily of minimum size.

4.10 Feedback Vertices and Single Extended Loops

Given any $n > 0$, we may construct a graph having n nested loops, and therefore with a loop tree of height n, but still having a single feedback vertex, as noted in section 4.9 above. Despite this, however, every graph having a feedback vertex must have one and only one "loop" L in an extended sense, namely that the head of L is not necessarily an entry point of L. As an example, consider the matrix product program of Fig. 4.1, namely:

```
        I = 1                          |       FOR I = 1 TO N {
1       J = 1                          |         FOR J = 1 TO N {
2       S = 0                          |           S = 0
        K = 1                          |           FOR K = 1 TO N
3       S = S+A[I,K]*B[K,J]            |             {S = S+A[I,K]*B[K,J]}
        K = K+1                        |           C[I,J] = S
        IF (K <= N) GOTO 3             |         }
        C[I,J] = S                     |       }
        J = J+1                        |       OUTPUT "END"
        IF (J <= N) GOTO 2             |
        I = I+1                        |
        IF (I <= N) GOTO 1             |
        OUTPUT "END"                   |
```

This program may be rewritten as follows:

```
        I = 1                          |       I = 1
        GOTO 1                         |       GOTO 1
3       S = S+A[I,K]*B[K,J]            |       WHILE (TRUE) {
        K = K+1                        |         S = S+A[I,K]*B[K,J]
        IF (K <= N) GOTO 3             |         K = K+1
        C[I,J] = S                     |         IF (K <= N) CONTINUE
        J = J+1                        |         C[I,J] = S
        IF (J <= N) GOTO 2             |         J = J+1
        I = I+1                        |         IF (J > N) {
        IF (I > N) GOTO 4              |           I = I+1
1       J = 1                          |           IF (I > N) BREAK
2       S = 0                          |       1:  J = 1
        K = 1                          |         }
        GOTO 3                         |         S = 0
4       OUTPUT "END"                   |         K = 1
                                       |       }
                                       |       OUTPUT "END"
```

It is not difficult, although a bit tedious, to check the logic of the two programs above and verify that they are the same. The head of the "loop" L, in the rewritten program, is the statement with label 3, and this is not an entry point of L. It should be clear, from this example, that rewriting a program in this way does not necessarily make it easier to understand, even though here we are replacing three loops by one. Also, although the statement with label 3 is also a feedback vertex for the graph, that fact does not help in the process of proving the program correct. Even though there is now only one intermediate assertion point, the total length of all the verification conditions of the program actually increases when this choice is made, as noted in [11].

In general, however, any graph with a feedback vertex V may be rewritten in this way. Here V becomes the head of L; with respect to V, the body of L is a dag, because otherwise it would have a cycle not containing V, contradicting the definition of a feedback vertex. However, in many cases (as above), V is not an entry point of L, so the graph does not necessarily have a loop tree of height 1.

4.11 Depth First Numbers and Loop Trees

The (preorder) depth first numbers (DFNs) of the vertices of a graph (also sometimes called discovery times) are in the sequential order in which they are encountered in a depth first search (DFS). Thus the DFN of the *k*th vertex to be encountered is *k*. Depth first numbers are treated in [4], and it is natural to ask whether there are connections between them and loop trees. At least in one special case, there is, indeed, a strong connection.

Suppose that the loop L has no exits, that is, edges from a vertex in L to a vertex outside L. (This is uncommon in flowgraphs, although it frequently occurs in other kinds of graphs, such as call graphs.) Then there exist integers i and j, for any DFS, such that the DFNs of the vertices of L are precisely $i, i+1, ..., j$. Here i is the DFN of the first entry point of L (call it H) to be encountered in this DFS. No vertices of L, therefore, are encountered before H, and thus the DFN of such a vertex cannot be less than i. Since there are no exits from L, the only way the DFS can leave L is by popping H from its stack. Before this is done, every vertex in L must be encountered, since L is strongly connected. If j is the largest DFN of a vertex in L, therefore, all vertices of L, and only these vertices, have DFNs in the range from i through j.

4.12 Estimating the Size of a Minimal Feedback Set

We finally introduce lower and upper bounds on the size of a minimal feedback set, as an extension of the process of finding loop trees.

THEOREM 4.2. Let G be a graph having a loop tree T with n vertices, m of which are leaves of T. Let k be the size of a minimal feedback set for G. Then $m \leq k \leq n-1$.

PROOF. We first note that any two leaves of T represent disjoint subgraphs of G. In fact, let U and V be leaves, and, among all common ancestors of U and V, let X be the one farthest from the root. Let X represent a subgraph with body B, and let Y be that child of X which is also an ancestor of U. By the choice of X, we see that Y is not an ancestor of V, and therefore X has another child, Z, which is an ancestor of V. Since Y and Z are children of X, they represent distinct, and therefore disjoint, strong components of B. Since U and V are descendants of X and Y respectively, they also represent disjoint subgraphs.

It follows that the *m* leaves of T represent mutually disjoint strongly connected sets, each of which contains at least one cycle. Any feedback set of G must contain one vertex in every such cycle, and its size must therefore be at least *m*. On the other hand, every vertex in T, other than the root, represents a loop with a head, and any cycle in the graph must pass through one of these heads. The set of all *n*–1 of these heads, therefore, is a feedback set for G, completing the proof.

If G has more than one loop tree, then, by repeatedly applying the logic above, we see that *m* and *n* may be taken here, respectively, as the maximum value of *m* and the minimum value of *n*, over all loop trees of G. Such further tightening of these bounds, however, cannot be expected to go too far, in light of the fact that the problem of determining a minimum feedback set, in general, is NP-complete ([12], as noted in [4] and [3]).

5 Clustering Trees

5.1 Loop Trees and their Generalizations

We shall need two generalizations of the concept of a loop tree, as introduced in Section 1 above. Suppose first that we were to include trivial, as well as non-trivial, strong components in the tree, at every level. This would make the construction simpler and more general, in one sense; but the name "loop tree" would no longer be appropriate, since trivial strong components are not loops. Hence we use the term **generalized loop tree** for this case. Also, any graph has at least one loop tree, but we will here be concerned only with strongly connected graphs. For such a graph G, every loop tree has G as the root, whose only child is again G. This redundancy is unnecessary in our context, and we therefore speak of the **basic loop tree** of a strongly connected graph, namely the subtree whose root is the single child of the actual root. Combining these two constructions, we obtain the **basic generalized loop tree** (or **BGL tree**) of a strongly connected graph.

5.2 Clustering Trees

Our object is to compare these BGL trees with what Tarjan [13] calls decomposition trees; these have applications in cluster analysis. Since loop trees also involve decompositions, we will use a separate term, **clustering trees**, to refer to the trees studied in [13]. It is assumed that we have a strongly connected graph G in which all edges have distinct weights. The leaves of the clustering tree are the vertices of G, and we build this tree from the bottom up, adding edges in increasing order by weight. Whenever the addition of an edge causes two or more components A_1, ..., A_k to coalesce into a single component A, we make A the parent of A_1, ..., A_k in the tree. When the last edge is added, the tree is complete, with G as its root.

Clustering trees and loop trees have very different motivations. Clustering trees are used in cluster analysis of data with an asymmetric similarity measure. The "distance" from A to B is not necessarily the same as that from B to A (as, for example, with flight times between cities, affected by prevailing winds). There is, however, always a "distance" from every vertex to every other, and these distances, or weights, are always distinct. The aim here is to find clusters, that is, groups of vertices, every one of which is close to every other one. Loop trees, by contrast, are used in the analysis of flowgraphs. The only obvious notion of distance, here, is distance of one statement from another in a program. However, here the vertices in the desired groups, or loops, might be far away from each other in the program, and our task is to find the loops.

Nevertheless, these two kinds of decomposition tree, for a graph G, have several similarities. In each of these trees, the root represents G; every vertex in the tree represents a strongly connected subgraph of G; and, if N_1 is the parent of N_2 in the tree, then G_1 contains G_2, where N_1 represents G_1 and N_2 represents G_2. It therefore becomes a matter of mathematical interest to determine relations between these two kinds of tree.

5.3 Top-Down Construction of Clustering Trees

We find it useful to provide an alternative definition of a clustering tree. First we notice that, in the construction of such a tree, the actual values of the weights are never used, but only their order. Since the weights are all distinct, their order is unique. Formally, we may define an **edge-ordered graph** as one with n edges given in the order $e_1, e_2, ..., e_n$. A graph in which the edges have distinct weights may then be identified with an edge-ordered graph, in which the kth smallest edge, in order of weight, is e_k, for $1 \leq k \leq n$. In what follows, therefore, we ignore weights, and assume that we have an edge-ordered graph.

Secondly, our definitions become more precise when our tree is built top-down rather than bottom-up. When building the tree bottom-up, we add e_1 first, then e_2, and so on. When building it top-down, we start with all edges in the graph; then we remove e_n, then e_{n-1}, and so on. At each stage, then, we have left only the edges $e_1, e_2, ..., e_k$ for some k; and the strong components of the graph, with all the original vertices but with only these edges, are the leaves of that part of the tree which has so far been constructed, from the top down.

Suppose now that we remove e_k, leaving only the edges $e_1, e_2, ..., e_{k-1}$. Here e_k was in some strong component, which, at this stage, is a leaf v. Removing e_k can affect only v, and no other strong component at this stage. It might happen that v is still strongly connected, even with e_k removed, in which case the tree does not change at this stage. If the tree does change, it is because v, with e_k removed, now has more than one strong component; each of these, at this stage, becomes a child of v.

5.4 Decomposition Trees of Cycles

Our first question is that of finding conditions on a graph G, with a BGL tree T, for which every clustering tree is the same as T. To simplify the statements of our theorems, we make the following definition.

DEFINITION 5.1. An edge-ordered graph G, with a BGL tree T, is said to have the **loop clustering property** if its clustering tree is the same as T.

We now have a surprisingly simple answer to the question we raised.

THEOREM 5.1. A graph G, with a BGL tree T, has the loop clustering property with any edge ordering whatsoever if and only if G is a single cycle (of any length).

Of course, if G is in fact a single cycle, it has a unique BGL tree; in the general case, the correspondence between the two types of tree is specific to a particular BGL tree.

PROOF. First suppose G is a single cycle, and consider a BGL tree T for G, with G as the root. Regardless of how we choose a head *h* for G, there will be one and only one edge *e* in G which leads to *h*; and here *e* is the only loopback in G. If we remove *e* from G, producing the body B of G, then B is a dag, and thus every child of G in T is trivial. Now consider the clustering tree T′ of G, built from the top down. No matter which edge is removed first, the graph G, with only the remaining edges, is a dag, and thus every child of G in T′ is also trivial. Thus T and T′ are the same in this case, and G, with T, has the loop clustering property.

Now suppose that G, with T, is not a single cycle. Since G is strongly connected, it must contain at least one cycle. Of all cycles contained in G, let C be one of shortest length *j*. Suppose first that $j = 1$, so that C is a self-loop, involving an edge *e* from a vertex *v* to itself. Since G is not a single cycle, there must be another vertex in the graph, say *w*. Since G is strongly connected, there are paths from *v* to *w* and from *w* to *v*, so that *k*, the number of edges in G, is at least 3. We will now show that G has at least two different clustering trees, contradicting our assumption that a clustering tree is always the same as T. Let $e_1, e_2, ..., e_k$ be the edges of G. If $e = e_1$, then *e* is removed last, in the top-down construction. At the second stage from the end, C is a leaf, and then, at the last stage, *v* becomes a child of C. On the other hand, if $e = e_k$, where $k \neq 1$, as we have seen, then *e* is removed first, in the construction. Since the cycle involving *v* and *w* has not yet been removed, *v* can never become a child of C, whose single edge has been removed at the first stage. Thus these two clustering trees are different.

Finally suppose that $j > 1$, and let $C = u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_j = u_1$. Since G is not C, there must be at least one edge e' in G which is not in C. We argue that there must also be at least one vertex *w* which is not in C. Otherwise, e' is $u_a \rightarrow u_b$ for some *a* and *b*. By combining e' with the path within C from u_b to u_a, we obtain a new cycle C′ whose length, by the construction, is strictly smaller than *j*, contradicting the choice of C. Since G is strongly connected, there is a path π in G from u_1 to *w*; let u_i be the last vertex in π which is also on C. Since $u_j = u_1$, we can always assume $i \neq j$ here. Since *w* is not on C, so that *w* is not u_i, there is a vertex w' (possibly $w' = w$) which immediately follows u_i in π. Thus there are edges $e'_1 = u_i \rightarrow u_{i+1}$ (where u_{i+1} exists since $i \neq j$) and $e'_2 = u_i \rightarrow w'$ in G. Here *e* is part of a cycle C′, since G is strongly connected; and $C' \neq C$ since C′ contains w', which is not on C by the choice of u_i. Also, e'_1 is in C but not in C′, while e'_2 is in C′ but not in C. Therefore, in the top-down process, if e'_1 is removed first, then C is broken but C′ remains, while if e'_2 is removed first, then C′ is broken but C remains. However, u_i is on both C and C′; so, as before, we have a contradiction because G, with T, has at least two different clustering trees. This completes the proof.

5.5 Bottlenecks

Theorem 5.1 shows us that most edge-ordered graphs do not have the loop clustering property. We can ask, however, whether a given graph has any edge ordering at all, for which it has that property. That this is not true for every graph may be seen from Fig. 5.1. If A is the start node, then C→A and E→A are the loopbacks, and the remainder of the graph is a dag; and thus this graph has a unique BGL tree T, which has every individual vertex as a child of G, which is at the top. Now consider the top-down order in which edges are removed. If we remove A→B, B→C, or C→A first, the strong component {A, D, E} will remain in the clustering tree T′. If we remove A→D, D→E, or E→A first, the strong component {A, B, C} will remain in T′. Neither of these

components are in T, as we have seen; so no edge ordering of this graph has the loop clustering property.

Consider now Fig. 5.2, which is much like Fig. 5.1 except for the vertex F. This graph has a unique BGL tree T, very much like that of the graph of Fig. 5.1. Suppose here that, in forming the clustering tree T′, we eliminate the edge A→F first. The resulting graph is a dag (it has no *directed* cycles, although that entire graph is an undirected cycle). Therefore we can eliminate the remaining edges in any order, and T′ will be the same as T. The important difference between the two graphs is that, in Fig. 5.2, we have an edge (namely A→F) which is contained in *every* cycle of the graph; in Fig. 5.1, we have no such edge. This leads us to the following definition.

DEFINITION 5.2. An edge in a rooted graph G, with root *r*, is a **bottleneck** if it is contained in every cycle within G that contains *r*.

The justification for the name should be clear; if you start at *r*, you cannot get back to *r* except through the bottleneck. Bottlenecks are related to vertex covers [3]; a vertex *x* is said to cover the root *r* of a rooted graph G, if all cycles in G which contain *r* also contain *x*. Here we are requiring that all cycles in G which contain *r* also contain an edge (that is, the bottleneck), rather than a vertex.

Bottlenecks are also related to the concept of a bridge as defined by Tarjan [13]; that is, an edge $e = u \rightarrow v$ such that every path from *r* to *v* must go through *e*. In particular, a bottleneck in a strongly connected graph G is always a bridge in this sense; for suppose the contrary. Then there is a path π from *r* to *v* that does not contain *e*. Since G is strongly connected, there is then a simple path π' in G from *v* to *r* which does not contain *e* (otherwise it would go back to *v*, and it would not be a simple path). Combining π and π', we would obtain a cycle in G, containing *r*, but not containing *e*, contrary to hypothesis. On the other hand, not every bridge is a bottleneck; thus there are no bottlenecks in the graph of Fig. 5.1, and yet A→B, B→C, A→D, and D→E are all bridges.

Often a graph G has a bottleneck for low-level reasons. For example, if there is only one edge leading from *r*, then that edge is clearly a bottleneck. Also, if G has one and only one loopback, then that loopback is a bottleneck, for a similar reason. However, these are not the only kinds of bottleneck, as may be seen from Fig. 5.3. By following either R→A→C, R→B→C, or R→A→B→C by either C→D→E→F→R or C→D→G→H→R, we obtain a cycle. These are the only simple cycles here which include the root R, and C→D is the only edge which is on all of these, and therefore the only bottleneck.

5.6 The General Loop Clustering Property Criterion

If G is strongly connected, with *r* being its head, and having a BGL tree T, we refer to a bottleneck as an **outer bottleneck** if it is not contained in any inner loops that G might have, in T. Not all bottlenecks are outer, as we may see from Fig. 5.4, which is the same as Fig. 5.3 with the loops L1 and L2 shown. Here C→D is contained in the inner loop L2, and is therefore not an outer bottleneck. (This graph actually has a further inner loop, which is either {B, C, D, G} or {A, C, D, E}, depending on whether A or B is taken as the head of L2.)

LEMMA 5.1. Let G be a strongly connected rooted graph with root *r*, and having an outer bottleneck *e*. Let B be the body of G, and let G´ be the graph obtained from G by removing *e*. Then the strong components of G´ are the same as those of B.

PROOF. The root *r* is a trivial strong component of B. It is also a trivial strong component of G´, because otherwise *r* would be contained in a non-trivial strong component of G, and hence in a cycle of G containing *r*; and such a cycle must contain *e*, by Definition 5.2, and is therefore not in G´. Any strong component L of B, other than {*r*}, is a maximal strongly connected subset of B, and cannot contain *r*. Because *e* is an outer bottleneck, it is not contained in L; therefore L is contained in G´. It remains strongly connected, and it is also maximal strongly connected in G´ (and therefore a strong component of G´). This is because any strongly connected graph L´, contained in G´ and containing L, could not contain *r*, because *r* is a trivial strong component of G´; and it would remain strongly connected in G, and thus in B, contradicting our assumption that L is maximal strongly connected in B. The trivial strong components of G´ consist of all vertices which are not in non-trivial strong components of G´. These are, therefore, exactly those vertices which are not in non-trivial strong components of B; so these components are also the same in G´ as they are in B. This completes the proof.

It should also be clear that the concepts of a bottleneck and an outer bottleneck are immediately extensible to loops within a BGL tree of G, in which the root of every such loop is taken to be its head. This extension is necessary in stating the main result of this section, as follows.

THEOREM 5.2. Let G be a strongly connected graph with a BGL tree T. Then there exists an ordering of the edges of G, with respect to which G has the loop clustering property, if and only if every loop in T has an outer bottleneck.

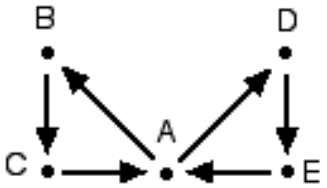


Fig. 5.1. Loop clustering property example

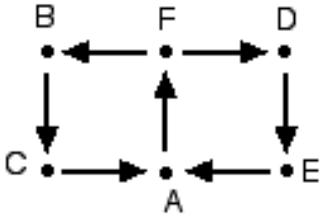


Fig. 5.2. Another example

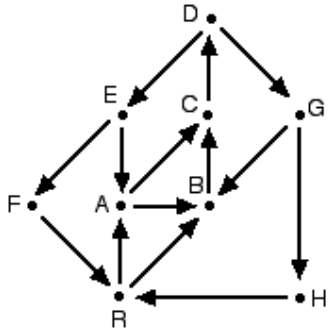


Fig. 5.3. Bottlenecks and outer bottlenecks

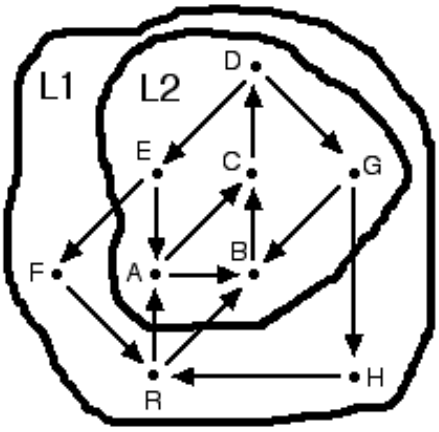


Fig. 5.4. Another example

PROOF. Let T be a BGL tree for G in which every loop has an outer bottleneck. As usual for trees, we define the level of a vertex in T as its distance from G (the root); and a level order of the vertices of T is an order $v_1, v_2, ..., v_j$ in which v_1 is the root, followed by all vertices at level 2 (in arbitrary order), then all vertices at level 3, and so on. For $1 \le i \le j$, let x_i be an outer bottleneck of the loop represented by v_i; and now choose the ordering $e_1, e_2, ..., e_k$ of the edges of G in such a way that $e_k = x_1$, $e_{k-1} = x_2$, and so on, so that in general $e_{k+1-i} = x_i$. Usually $j < k$ here, and the remaining edges $e_1, e_2, ..., e_{k-j}$ may be chosen in any order.

Consider now the top-down construction of the clustering tree T´ for G, in this order. We first eliminate $e_k = x_1$, and, since this is an outer bottleneck for G as a whole, we are left with the graph G´ as in Lemma 5.1. By this lemma, G´ has the same strong components as does the body B of G; and all these become children of G in T´, exactly as in T. Now we eliminate $e_{k-1} = x_2$, which is an outer bottleneck for some non-trivial strong component L of B. Here L is an inner loop, and, since x_2 is an outer bottleneck for L, we are left with a graph G" which is like G´ as in Lemma 5.1, but applied this time to L. By this lemma, G" has the same strong components as does the body B´ of L; and all these become children of L in T´, exactly as in T. We now proceed to eliminate $e_{k-2} = x_3$, and so on. Because of the level ordering of the vertices of T, the parent L´ of any such vertex L is added to T´ before L is. Let v be an outer bottleneck of L, and let v' be an outer bottleneck of L´. When v' is eliminated, the strong components of the body of L´, including L, become part of T´, so that the same can happen to L when v is later eliminated. After eliminating x_1 through x_j, the only remaining vertices in either T or T´ are trivial, and no further building is done, so that in fact T = T´.

Now suppose that some loop L in T does not have an outer bottleneck; we show that no ordering $e_1, e_2, ..., e_k$ of the edges of G leads to a graph having the loop clustering property. Assume the contrary, so that T = T´ for this ordering. Since L is in T, it is thus also in T´. Let i be the smallest integer such that, in the bottom-up process, when $e_1, e_2, ..., e_i$ have been added, L is included among the strong components produced so far. Therefore, when $e_1, e_2, ..., e_{i-1}$ have been added,

L has not been so included, but adding e_i causes L to be included. Consider now the remainder of the process of adding e_{i+1}, e_{i+2}, ..., e_k. This process only combines existing strong components into larger ones; it cannot affect those parts of T´ which have already been constructed. When the process is finished, it produces T´, which, by assumption, is the same as T. Within T, however, the children of L are the (trivial and non-trivial) strong components of the body of L. Therefore, when L is first included by adding e_i, these same strong components must already have been in that part of T´ that had been constructed so far.

We now note that e_i cannot be in any non-trivial strong component of the body of L, nor can it be outside L, since adding e_i, in either of those cases, could not produce L by combining its strong components. However, e_i, by assumption, is not an outer bottleneck of L, and therefore there is some cycle C in L, containing the head of L, that does not contain e_i. Thus L cannot, in fact, be produced when e_i is added, since C would have to be included in what is produced. This contradiction completes the proof.

5.7 Multiple Null Node Expansions

Using the terminology of Theorem 5.2, we see that there are strongly connected graphs for which T´ is never the same as T. We might ask, then, whether such a graph G might be expanded to form a graph G´ for which T´ = T for some choice of ordering of the edges of G´. Ideally, we would like G´ to be just as efficient as G, both in space and time, under some reasonable interpretation of these. In fact, this can always be done, using a generalization of the idea of a null node expansion. Given an edge $v \rightarrow w$ in a graph G, a **null node expansion** of G is a graph G´ obtained from G by adding a new vertex z, removing the edge $v \rightarrow w$, and introducing two new edges $v \rightarrow z$ and $z \rightarrow w$. If G is a flowgraph, any flow of execution along $v \rightarrow w$ is replaced by the use of $v \rightarrow z$ followed by $z \rightarrow w$. Here z is assumed to take no additional space or time, so that the total space or time taken by any execution of G´ is the same as that of the corresponding execution of G.

We generalize this notion by considering several edges $v_1 \rightarrow w$, $v_2 \rightarrow w$, ..., $v_j \rightarrow w$, all leading to the same vertex w. A **multiple null node expansion** G´ of G is now obtained by adding a new vertex z, as before; removing all the above edges; introducing the new edge $z \rightarrow w$ as before; and then introducing the further new edges $v_1 \rightarrow z$, $v_2 \rightarrow z$, ..., $v_j \rightarrow z$. Any flow of execution along any $v_i \rightarrow w$ is replaced by the use of $v_i \rightarrow z$ followed by $z \rightarrow w$. As before, since z takes no additional space or time, the executions of G´ are as efficient as the corresponding executions of G. As with ordinary null node expansions, multiple null node expansions can be iterated, expanding several vertices like w in the process. We now show the utility of such expansions in this context.

DEFINITION 5.3. A **unique loopback BGL tree** is a BGL tree in which every loop has one and only one loopback.

THEOREM 5.3. Any strongly connected graph, not having a unique loopback BGL tree, has an iterated multiple null node expansion which has a unique loopback BGL tree.

PROOF. Let T be a BGL tree for G, and suppose that T is not a unique loopback tree. Given any loop L in T having the head w, and having j loopbacks $v_1 \rightarrow w$, $v_2 \rightarrow w$, ..., $v_j \rightarrow w$, for $j > 1$, we form the multiple null node expansion of G as above, in which L now has the single loopback $z \rightarrow w$. By

repeating this process for every such loop in T, we obtain an iterated multiple null node expansion in which every loop has a single loopback. This completes the proof.

COROLLARY. For any strongly connected graph G, with a BGL tree T, there exists an ordering of the edges, either of G or of some iterated multiple null node expansion of G, with respect to which it has the loop clustering property.

PROOF. We saw, at the end of section 5.5 above, that a unique loopback in a loop L is always a bottleneck. It is also an outer bottleneck; it cannot be contained in any loop L´ inner to L, since L´ cannot contain the head of L. The corollary now follows immediately from Theorems 5.2 and 5.3.

5.8 Loop Trees and Communication Networks

The theorems above might appear unsatisfactory, in that a closer connection between clustering trees and BGL trees might have been expected. There is a larger question, however, as to whether loop trees, when applied to communication networks, are useful at all. Loop trees arose originally out of our analysis of call graphs, in which a self-loop corresponds to ordinary recursion, and a more general loop corresponds to mutual recursion. They were quickly applied, also, to flowgraphs; and here a loop corresponds roughly, although not quite always syntactically, to a loop in the ordinary programming sense. Consider now a communication network, in which, as usual, we expect communication to proceed in both directions between any two vertices. What is the interpretation, in this context, of a loop in a loop tree? Fig. 5.5 indicates one of the difficulties here. This graph G gives simple two-way communication along a line, in either direction (note that it would be highly unlikely for a flowgraph to have this form). Suppose that the start node of G is u_1; then G has a unique loop tree consisting of n–1 nested loops, each containing the vertices u_k through u_n for some k, $1 \leq k \leq n$–1. Thus loops are not necessarily obvious divisions of a communication network.

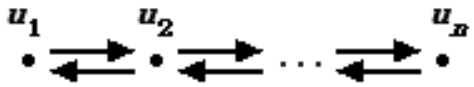


Fig. 5.5. Why loop trees make little sense for communication networks

6 Wheels within Wheels

6.1 Introduction

In recent work [1] we introduce a decomposition of an arbitrary strongly connected graph G into subgraphs. Such a decomposition has an associated tree, which we call a loop tree, in which every node N, other than the root, represents a strongly connected subgraph H of G. Our work was motivated by the consideration of flowgraphs; if G is a flowgraph and N represents H as above, then H is either a single node, with no edges, or else it is, in a semantic sense, a loop in G.

There is also a completely different way of decomposing a strongly connected graph. This was described by Luce [5] in a form which applies to some, but not all, such graphs. Luce's work was later rediscovered by Knuth [6], who used still another technique to provide an extension of Luce's decompositions to any strongly connected graph. Luce's work was motivated by considering communication networks, and many of his concepts do not have obvious informal interpretations when applied to flowgraphs. However, from a purely formal point of view, both Luce's decomposition and Knuth's extension of it are very similar to ours.

Using Knuth's terminology, we may define a wheel, which is a strongly connected graph decomposed into strongly connected parts, that are then decomposed into others, and so on. Extending this terminology, we define a *wheel tree* to represent such a decomposition. In both wheel trees and loop trees, every node represents a strongly connected subgraph; if N and N´ represent respective subgraphs H and H´, and if N is the parent of N´, then H contains H´; while if N and N´, as above, have the same parent, then H and H´ are disjoint.

The purpose of this section is threefold. We start by describing Luce's work, using current terminology for graphs. We define wheel trees and associated concepts, and show the similarities and differences between wheel trees and loop trees. We then use Luce's work to provide a sharpened form of Knuth's decomposition. Finally, we relate this work to the further commentary on both [5] and [6] provided by Chaty and Chein [14].

Specifically, Knuth's theorem implies that, in our terminology, every strongly connected graph has a wheel tree; but no information is provided as to its form. In section 6.13 below, however, we show, using one of Luce's decomposition theorems, that such a tree may always be taken to have a particular form, which we call a chandelier. This has a special node X, such that every node in the chandelier has exactly one child if and only if it is a proper ancestor of X.

6.2 Loop Trees and Wheel Trees

A brief description of our theory of loop trees was given in Section 1 above. We now compare this with the theorem presented in [6] according to which any strongly connected graph D that has any edges at all has a wheel decomposition. This means that D is composed of strongly connected graphs D_1, D_2, ..., D_n, for $n \geq 1$, together with edges $x_1 \rightarrow y_2$, $x_2 \rightarrow y_3$, ..., $x_{n-1} \rightarrow y_n$, $x_n \rightarrow y_1$, where each x_i and each y_i is a node of D_i. Suppose now that we set up a tree whose root is associated with D, and having n children, associated respectively with D_1, D_2, ..., D_n. Since the D_i are all strongly connected, they may be decomposed further, and a tree, which we call a wheel tree (Definition 6.1), may be built up in this way.

Two decomposition theorems are presented in [5], of which the second is relevant here. There are strong conditions on the graph to be decomposed, but the conclusion is also stronger, in that we have $n \geq 2$ rather than $n \geq 1$ in the above decomposition. Formal treatments of both Luce's and Knuth's work are given in section 6.11 below.

6.3 Basic Definitions

Any application of Luce's work [5] must start with its terminology. Luce was concerned with oriented graphs, which, in modern terminology, are multigraphs; that is, there can be more than one edge between two given nodes. He considered a special case of multigraphs, which he calls networks, and which are, in fact, the same as (directed) graphs in the modern sense. Further notation used by Luce is also often not what is used today.

Luce begins by defining oriented graphs, networks, subnetworks, complete subnetworks, *q*-chains and their products, and connected and disconnected networks. These definitions are compared with today's terminology in Table 1. Most important, here, is Luce's use of the term "connected," which means, in current terminology, *strongly* connected.

6.4 The Degree of a Graph

We next pass to some of Luce's further definitions. The first is that of the **degree** of a graph ([5], p. 703). A graph has degree 0 if it is not (strongly) connected. It has degree 1 if, first, of all, it does not have degree 0 (so that it is strongly connected), but it has an edge whose removal leaves a graph which is not strongly connected.

Luce generalizes this to a graph of degree *k*, in which you have to remove *k* edges to get a graph which is not strongly connected. His approach now involves two decomposition theorems for graphs. The first of these reduces the study of all graphs to the study of graphs of degree 1; this is done through the concept of the sum of several graphs, which we take up in section 6.8 below. The second is a decomposition only for graphs of degree 1; we take this up in sections 6.11 and 6.14 below.

It will be necessary, in these later sections, to present examples of graphs of degrees 2 and 3. Fig. 6.1 shows the double ring, a graph of degree 2; if we remove the two edges which lead outward from any node, the result is no longer strongly connected, since there are no paths leading from

that node. However, if we remove any single edge, the result is still strongly connected. Fig. 6.2 shows the double cube, which, by similar logic, may be seen to be a graph of degree 3.

6.5 Minimal and 1-Minimal Graphs

A **minimal** network (i. e., graph), in [5], is strongly connected and also **1-minimal**, meaning that the removal of *any* edge would result in a graph which is not strongly connected. This definition appears to be motivated by an analysis of communication networks. Suppose that, in such a network G, there are three nodes, A, B, and C, with edges A→B, B→C, and A→C; then G is not minimal, in an informal sense. You can always eliminate the edge A→C from G, and obtain a smaller network which achieves the same effect by replacing any communication along A→C by the use of A→B and B→C.

Table 1. Comparison of Luce's terminology with current terminology

Luce's terminology	**Current terminology**	**Luce's definition (pp. 701-702 of [5])**
Network	(Directed) graph	"A *network* N...is a system composed of...a finite non-empty set of...*nodes*...and...a...subset of the set of all ordered pairs of nodes" (p. 701)
Links	Edges	"The members of P...are called the *links* of N" (p. 701)
Oriented graph	Multigraph	*See first 6 lines, p. 701*
(*ab*)	*ab*	"...bracketed ordered pairs *(ab), (ca),* ... [will be used] to denote links" (p. 701)
Initial node of *(ab)*	*a*, in $a \rightarrow b$	"If *(ab)* is a link, the first node, *a*, will be called the *initial node*..." (p. 701)
End node of *(ab)*	*b*, in $a \rightarrow b$	"...and the second, *b*, the *end node* of the link" (p. 701)
Subnetwork	Subgraph (not necessarily induced)	"A *subnetwork N´* of a network *N* is a subset *M´* of the nodes, *M*, of *N*, with *P´* taken to be some subset (not necessarily proper) of those links of *N* which are definable on *M´*" (p. 702)
Complete subnet work	Subgraph of *N* containing all nodes of *N*	"If *M´ = M*, we shall say that the subnetwork is *complete*" (p. 702)
Non-reflexive network	Graph with no self-loops	"We shall call a network *non-reflexive* if there are no links of the form *(aa)*" (p. 702)
Arc	Two-node or one-node cycle (*not* a single edge)	"In case both...*(ab)* and *(ba)* are present in a network,...an *arc ab* exists between *a* and *b*, the arc consisting of this pair of links...A link of the form *(aa)* is always the arc *aa*" (p. 702)
q-chain from *a* to *b*	Simple path, of length *q*, from *a* to *b*	"A...*q-chain* from *a* to *b* is a set of *q* links of the form (ac_1), (c_1c_2), ..., $(c_{q-2}c_{q-1})$, $(c_{q-1}b)$, such that no node appears more than once, except in the case $a = b$ where *a* appears twice" (p. 702)
(ab, q)	$a \rightarrow c_1 \rightarrow c_2 \rightarrow ... \rightarrow c_{q-1} \rightarrow b$	"Any *q*-chain from *a* to *b* will be denoted by *(ab, q)*" (p. 702)
Product of two chains	Concatenation of two paths	"If *c* is a node included in a *q*-chain from *a* to *b*, then we may subdivide the chain into the 'product' of two chains, one from *a* to *c*, and the other from *c* to *b*..." (p. 702)
Circuit	Cycle	"An (oriented) *circuit* is a chain of the form *(aa, q)*" (p. 702)
Connected	Strongly connected	"A network is *connected* if there exists a chain from each node to every other node" (p. 702)
Disconnected	Not strongly connected	"A network which is not connected is *disconnected*" (p. 702)

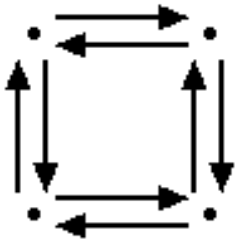


Fig. 6.1. The double ring

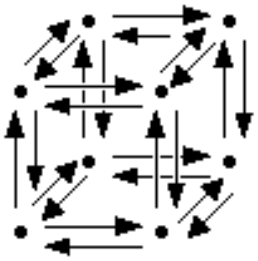


Fig. 6.2. The double cube

6.6 Uniform and 1-Uniform Graphs

Luce's motivation for defining uniform networks (that is, graphs) is to develop "a condition implying that there is an even distribution of connectedness throughout the network; roughly, that the degree of any connected subnetwork is not greater than that of the network itself" ([5], p. 703). A **uniform** graph is one which is strongly connected and **1-uniform**, meaning that every strongly connected subgraph is of degree 1.

This is in contrast to the situation illustrated by the graph of Fig. 6.3. Here we have a strongly connected graph; however, if we delete the edge from A to B (or from B to A), the result is not strongly connected. Therefore, this graph is of degree 1, by the definition in section 4 above. However, if we eliminate the node A entirely, together with both of its edges, the result has degree 2, as we also saw in section 4. Thus this graph is not 1-uniform, and therefore not uniform. It is also not minimal; indeed, you can remove four edges (B→E, E→D, D→C, and C→B) from the graph, producing a strongly connected graph.

This example may be generalized. Whenever a graph G is not uniform, it has a subgraph G´ of degree at least 2. At least one edge X→Y may always be eliminated from G´, producing a strongly connected subgraph H´. This implies that there is a path π in H´ from X to Y. If you eliminate X→Y from G, the remaining graph, H, will still be strongly connected. In fact, for any nodes U and V in H, we may replace X→Y by π in any path from U to V in G that involves X→Y, producing a path from U to V in H. This implies that a graph which is not uniform is not minimal; or, more simply, a minimal graph is always uniform ([5], p. 704).

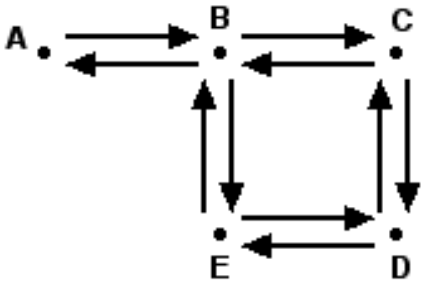


Fig. 6.3. Removing edges to produce a minimal graph

6.7 Descendants of a Graph

Given a graph G which is strongly connected but not minimal, you can always eliminate at least one edge from it, and the result will still be strongly connected. Suppose now that you continue to eliminate edges, one by one, preserving strong connectivity as you go, until you reach a graph D for which you cannot do this any more. That is, you cannot eliminate any edge from D and still preserve strong connectivity, so that D is minimal, by the definition in section 6.5 above.

Sometimes there is more than one way to do this. For example, in the graph of Fig. 6.3, we could, as noted in section 6 above, remove the edges B→E, E→D, D→C, and C→B to produce a strongly connected subgraph. We could also, however, have removed the edges B→C, C→D, D→E, and E→B, and the result would still be strongly connected. Note that we are removing the same number of edges in each case (four, here).

There are other cases, however, in which you can remove different numbers of edges, in producing a minimal graph. Thus, in Fig. 6.4, we can eliminate one edge from G_1 to produce G_2; or we can eliminate two edges from G_1 to produce G_3. Note that both G_2 and G_3 are minimal; they are both strongly connected, but you cannot get a strongly connected graph by eliminating more edges from either one.

We now consider minimal graphs obtained as above, by eliminating *as many edges as possible*. These are what Luce calls **descendants** of the original graph. In the first case above, we can produce a descendant by eliminating four edges, in either of two possible ways. In the second case above, G_3 is a descendant of G_1, while G_2 is not, because it is produced from G_1 by eliminating only one edge instead of two.

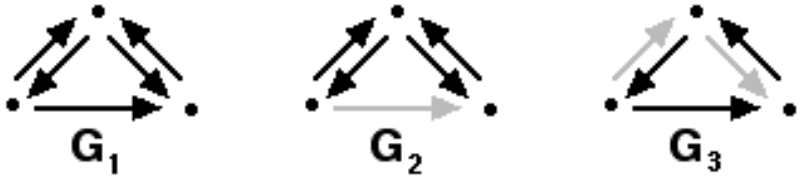


Fig. 6.4. Another example

6.8 Sums and Decompositions

Looking again at the graph of Fig. 6.1, we can see that it is made up of two cycles, one going clockwise around the ring and the other going counterclockwise. This is a special case of what Luce refers to as the sum of two subgraphs. In general, a graph G is the **sum** of subgraphs G_1, G_2, ..., G_k if every node of G is contained in all of the G_i but every edge of G is contained in exactly one of the G_i.

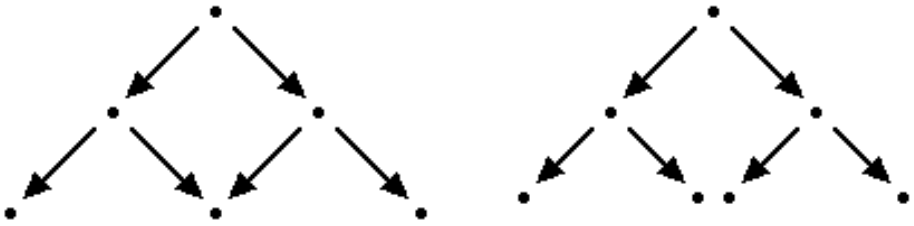
Suppose now that the degree of G is 1, which implies that G is strongly connected. Let G_1 be a descendant of G, as in the preceding section (here G_1 = G if G is already minimal). Form a graph G_2 out of all the nodes of G, together with exactly those nodes that were removed from G to form

G_1. Then G is the sum, in the above sense, of G_1 and G_2. Here G_1 is minimal (because it is a descendant of G), while G_2 is not strongly connected, since otherwise (as Luce proves) G would have degree 2, not 1.

Luce's first decomposition theorem, which we mentioned in section 6.4 above, generalizes this to a graph of degree $k > 1$, which is expressed as the sum of k+1 subgraphs, all of which are 1-minimal (see section 5 above). The first of these, G_1, is always connected (and therefore minimal), while the last, G_{k+1}, is always disconnected. The remaining G_2 through G_k may be either connected or disconnected, but their connected components are minimal. As we will see in section 6.14 below, our approach avoids this first decomposition theorem entirely, using only the second one, taken up in section 6.11 below.

6.9 Trees, Arcs, and Undirected Graphs

We now pass to trees, and what they mean for Luce. In the modern sense, Luce's networks are directed graphs, but Luce's trees are not directed trees. A directed tree, today, is a directed graph containing no cycles, not even undirected ones, as in Fig. 6.5. This is clearly not what Luce means by a tree, since he states (p. 707, lines 13-14) that "...a network which is a tree is minimal." A minimal graph, however, is (strongly) connected, implying that it has at least one cycle, so it cannot be a directed tree in the above sense.



A general dag (containing no directed cycles, although it does contain an undirected cycle)

A directed tree (not even any undirected cycles)

Fig. 6.5. Directed and undirected cycles

To understand Luce's notion of a tree, one has to consider another of his definitions (Table 1). An **arc**, for Luce, is not a single edge (as in [14], for example), but rather a pair of nodes, say U and V, with both edges U→V and V→U present in the graph. Suppose now that we have a (directed) graph in which every edge is part of an arc; then we can produce a corresponding undirected graph in which each arc is replaced by a single undirected edge. Luce calls this a graph, and says that, in this case, a network is a graph. When [5] was written, the term "graph," in general, meant what we now call an undirected graph.

Later, [5] refers to "the concept of a tree in graph theory," meaning, therefore, the theory of undirected graphs. A tree, then, for Luce, is what is obtained by replacing every edge *a*-*b*, in an undirected tree, by the two edges $a \rightarrow b$ and $b \rightarrow a$, as in Fig. 6.6. Now it makes sense that a network which is a tree is minimal. A tree, in this sense, is strongly connected, but the removal of any of its (single) links leaves a graph which is not strongly connected. If $a \rightarrow b$ is removed, there is no

longer a path from *a* to *b*, because such a path, followed by $b \rightarrow a$, would have formed a cycle; and a tree has no cycles.

6.10 L-Trees and Proper Descendants

We will refer to trees, in the sense of section 6.9 above, as **L-trees** (L for Luce), in order to distinguish them from directed trees in the modern sense, such as decomposition trees. Note that a single node with no edges is acyclic, and therefore a tree, as an undirected graph. It is therefore also an L-tree, as a directed graph. We refer to such an L-tree as a **trivial** L-tree.

We define, in the obvious way, a **proper descendant** of G; that is, a descendant H of G such that $H \neq G$. Luce introduces a lemma ([5], Lemma 3.1, p. 707), which says, restated in modern terminology, that a proper descendant of a graph cannot be an L-tree; this will be used in section 6.13 below. We here do not repeat the proof of this lemma, although we note that an example has already been introduced, in Fig. 6.4 above. Here the graph G_2 is an L-tree, as illustrated in Fig. 6.7. As we noted in section 6.7 above, G_2 is not a proper descendant of G_1.

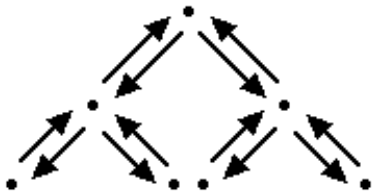


Fig. 6.6. A directed tree, for Luce

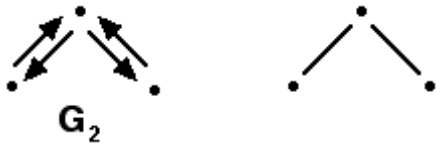


Fig. 6.7. An L-tree, obtained from an undirected tree

6.11 Wheels

We now take up the study of Luce's second decomposition theorem. This theorem is seemingly restrictive, as it applies only to a graph which (*a*) contains no self-loops, (*b*) is minimal, and (*c*) is not an L-tree. More recently, Knuth [6], using a completely different method, proved that any strongly connected graph has a decomposition similar to, but more general than, that of Luce. In section 6.13 below, we will use Luce's decomposition in order to prove a stronger version of Knuth's decomposition. The following definitions are meant to be applicable to both of these decompositions.

DEFINITION 6.1. A (Knuth) **wheel** W is (defined recursively as) either a single node with no edges (in which case it is called **trivial** and its **wheel tree** is a single node), or a graph consisting

of n wheels W_1, W_2, ..., W_n, where $n \geq 1$, together with n edges $x_1 \rightarrow y_2$, $x_2 \rightarrow y_3$, ..., $x_{n-1} \rightarrow y_n$, $x_n \rightarrow y_1$, where each x_i and each y_i is a node of W_i, for $1 \leq i \leq n$, in which case:

a) the **size** of W is n,
b) the **thickness** of W is the minimum of n and the thicknesses of all nontrivial W_i, $1 \leq i \leq n$, and
c) the **wheel tree** of W consists of a root node which, if W is non-trivial, has, as children, the root nodes of the wheel trees of W_1, W_2, ..., W_n.

Every node in the wheel tree of W corresponds to a wheel somewhere in the recursively expressed definition of W. It should be clear that a wheel of thickness 2, for example, is one for which, not only is its size at least 2, but the sizes of all wheels corresponding, in this way, to all nodes in its wheel tree are also at least 2, with one of them being equal to 2. We now need three lemmas.

LEMMA 6.1. Every wheel is strongly connected.

PROOF. Using the notation of Definition 6.1, let W be a non-trivial wheel, where by induction we may assume that W_1, W_2, ..., W_n are all strongly connected. Then, if X is in W_i and Y is in W_j, there is a path from X to x_i (because W_i is strongly connected) to y_{i+1} (or y_1 if $i = n$) to x_{i+1} (or x_1, because W_{i+1}, or W_1, is strongly connected), and so on, and finally to Y. Knuth's theorem now says that, conversely, every strongly connected graph is a wheel [6].

LEMMA 6.2. Every wheel containing a self-loop has thickness 1.

PROOF. Informally, we can see this because, at some point in the decomposition, the self-loop must be an edge like $x_1 \rightarrow y_2$ above; but $x_1 = y_2$ here, so that $n = 1$ and the thickness is 1. Formally, if W contains a self-loop, then W is non-trivial; if any of W_1, W_2, ..., W_n has thickness 1, we are done. By inductive hypothesis, there are no self-loops in W_1, W_2, ..., W_n, so the self-loop must be an edge like $x_1 \rightarrow y_2$ above. In that case, $n = 1$ and the thickness is 1, just as before.

LEMMA 6.3. A non-trivial L-tree has thickness 2.

PROOF. Let T be an L-tree and let S be its associated undirected tree. Clearly S has a node U of degree 1 (otherwise it would contain a cycle); let V be the node adjacent to U in S, so that T contains both U→V and V→U. Let T´ be obtained by removing U, U→V, and V→U from T; then T´ is also an L-tree. We form T into a wheel of size 2 by taking W_1 to be U by itself; W_2 to be T´; $x_1 \rightarrow y_2$ to be U→V; and $x_2 \rightarrow y_1$ to be V→U. By induction, W_2 is either trivial, or it has thickness 2, while W_1 is trivial. Since T has size 2, it therefore has thickness 2.

6.12 Compound Circuits

The inclusion of the notion of thickness in Definition 6.1 is intended to allow it also to apply to another of Luce's definitions. A **compound circuit** is a wheel of thickness greater than 1. Luce explicitly specifies that a compound circuit has no self-loops; however, this follows anyway from Lemma 6.2. In order to state Luce's theorem here more succinctly, we introduce another definition.

DEFINITION 6.2. A **generalized compound circuit** is either a compound circuit or a single node with no edges. A **generalized L-tree** is the result of replacing every node in an L-tree by a generalized compound circuit, and every edge between nodes in the L-tree by an edge between nodes in the corresponding generalized compound circuits.

LEMMA 6.4. A generalized L-tree has thickness greater than 1, unless it is a single node with no edges.

PROOF. The logic here is almost identical to that for ordinary L-trees. Let T be a generalized L-tree, with underlying tree S. If S is a single node, then T is either a single node with no edges, or a single compound circuit, of thickness greater than 1. Otherwise, let U and V be as in Lemma 6.3, and let U′ and V′ be the generalized compound circuits associated, in T, with U and V respectively. There now exist edges $u_1 \rightarrow v_2$ and $u_2 \rightarrow v_1$, where u_1 and u_2 are in U′, while v_1 and v_2 are in V′. Let T′ be obtained by removing U′, $u_1 \rightarrow v_2$, and $u_2 \rightarrow v_1$ from T; then T′ is also a generalized L-tree. We form T into a wheel of size 2 by taking W_1 to be U′; W_2 to be T′; $x_1 \rightarrow y_2$ to be $u_1 \rightarrow v_2$; and $x_2 \rightarrow y_1$ to be $u_2 \rightarrow v_1$. By induction, W_2 has thickness greater than 1, unless it is trivial; and so does W_1, because it is a generalized compound circuit. Since T has size 2, it therefore has thickness 2.

Luce's second decomposition theorem now says that a minimal graph with no self-loops, which is not an L-tree, is a generalized L-tree. This result is deeper than Knuth's theorem, but it is also seemingly more restrictive. However, Luce's theorem may be used in a proof of a generalization of Knuth's theorem. Note that this theorem puts no restriction on the form of the wheel tree, whereas ours will put a strong restriction on it, as indicated by the following definition.

DEFINITION 6.3. A **chandelier** is a rooted tree T containing a node X such that any node in T has exactly one child if and only if it is a proper ancestor of X. All proper ancestors of X constitute the **upper part** of T, the remainder of T (including X) being its **lower part**.

Fig. 6.8 shows a chandelier and the visual justification of its name. Note that X and every node below it has either no children, or at least two children, whereas every node above X has exactly one child. Mathematically, a chandelier could consist of just a lower part, or of just X and an upper part, or even of X by itself. The thickness of the lower part of a chandelier is always greater than 1, unless the lower part consists of X by itself.

6.13 A Generalization of Knuth's Theorem

We are now ready for our main result on wheel trees.

THEOREM 6.1. Any strongly connected graph G is a wheel whose wheel tree is a chandelier.

PROOF. First we ask whether G has a self-loop. If so, we express G as a wheel with size 1, in which $x1 = y1$, and W1 is G with the self-loop removed. We denote W1, here, by G1, and continue removing all self-loops in this fashion, producing G2, G3, etc., until we arrive at a graph $H = Gj$ with no self-loops. Every node in the wheel tree of G, from G down to H, has exactly one child, except for H.

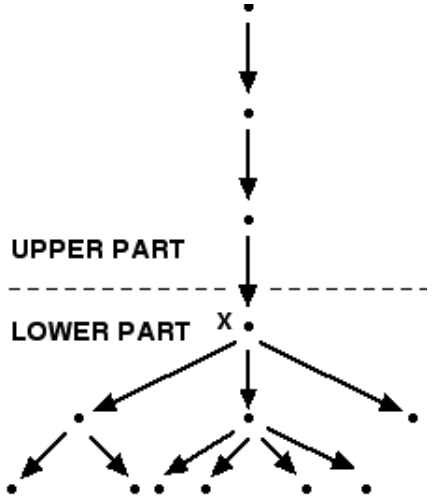


Fig. 6.8. A chandelier

At this point we ask if H is an L-tree. Suppose it is; then we apply Lemma 6.3, which shows that the wheel tree of H has thickness 2. Let X be the root of this wheel tree; then the wheel tree of G, with this X, satisfies the requirements of Definition 6.3, and we are done in this case.

Now suppose that H is not an L-tree. If so, then, if it is minimal, we set K = H and proceed to the next paragraph. If H is not minimal, then let K be a descendant of H; note that, by Luce's lemma of section 6.10 above, K is not an L-tree, since it is a proper descendant of H. Let z_1, z_2, ..., z_k be the edges which are removed from H to produce K, where each z_i is $a_i \rightarrow b_i$ for $1 \leq i \leq k$. Let H_i be H with z_1, z_2, ..., z_i removed, for $0 \leq i \leq k$, so that H_0 = H and H_k = K. For $1 \leq i \leq k$, then, H_i is H_{i-1} with z_i (= $a_i \rightarrow b_i$) removed; and we may express H_{i-1} as a wheel of size 1, in which $x_1 = a_i$, $y_1 = b_i$, and W_1 is H_i (that is, H_{i-1} with z_i removed). Every node in the wheel tree of G, from G down to K, has exactly one child, except for K.

At this point K is a minimal graph, not an L-tree, and containing no self-loops, so that Luce's second decomposition theorem applies. Accordingly, K is a generalized L-tree, and, by Lemma 6.4, it has thickness greater than 1, unless it is a single node with no edges. Let X be the root of the wheel tree of K; then the wheel tree of G, with this X, satisfies the requirements of Definition 6.3, as before, and we are done.

6.14 Chandeliers and Luce's Two Decompositions

Luce presents two decompositions of graphs; but Theorem 6.1 uses only the second of these theorems. We will now explain why. Luce's first decomposition is a way of reducing the study of graphs to the study of minimal graphs; any graph which is not minimal, and which might, indeed, have high degree, may be expressed as the sum of simpler graphs.

The double cube of Fig. 6.2 is a good example of this. If we were to apply Luce's first decomposition theorem to this, we would find this graph G decomposed into G_1 and G_2, where G_1 is further decomposed into H_1 and H_2. Here G, G_1, and H_1 are shown in Fig. 6.9. We omit the rest of the details, except to note that there is a decomposition tree for G, constructible in this way. However, this tree is not like our other decomposition trees because G_2, an internal node in the tree, is not strongly connected, although it has children in the tree which do correspond to strongly connected subgraphs.

In constructing a chandelier for the double cube G, however, a much simpler decomposition is used. The special node X, as in Definition 6.3, may now be taken to represent the graph H_1, as in Fig. 6.9. Note that G has 24 edges, while H_1 has only eight. Each of the 16 remaining edges, in turn, is taken as the single edge in a wheel of size 1, resulting in sixteen nodes at the upper part of the chandelier. The decomposition of H_1 is then into a wheel of size 8, and each child of H_1 in the chandelier is a single node. All this is illustrated in Fig. 6.10.

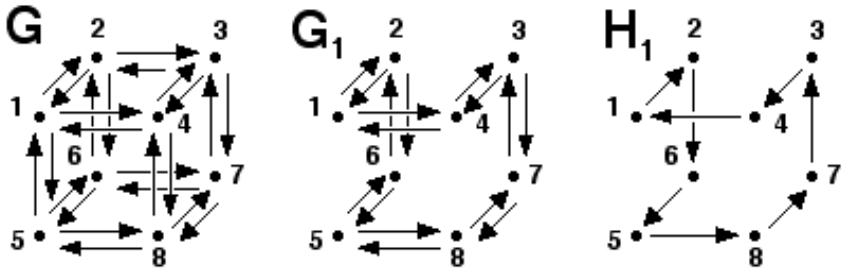
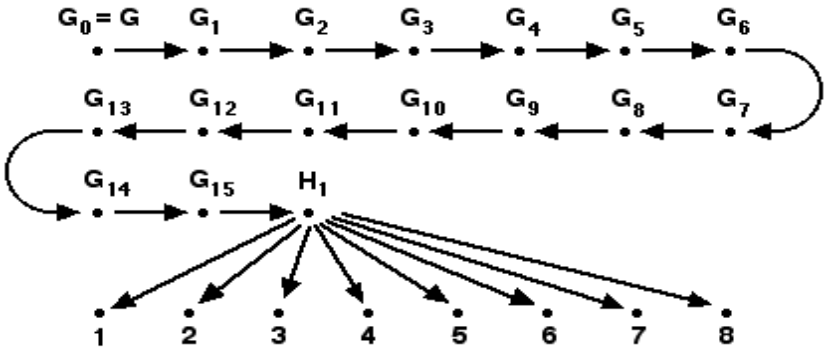


Fig. 6.9. Constructing a chandelier for the double cube



$G_1 = G \sim \{1 \to 4\}$ $G_2 = G_1 \sim \{2 \to 1\}$ $G_3 = G_2 \sim \{3 \to 2\}$ $G_4 = G_3 \sim \{4 \to 3\}$

$G_5 = G_4 \sim \{5 \to 1\}$ $G_6 = G_5 \sim \{6 \to 2\}$ $G_7 = G_6 \sim \{7 \to 6\}$ $G_8 = G_7 \sim \{8 \to 4\}$

$G_9 = G_8 \sim \{1 \to 5\}$ $G_{10} = G_9 \sim \{2 \to 3\}$ $G_{11} = G_{10} \sim \{3 \to 7\}$ $G_{12} = G_{11} \sim \{4 \to 8\}$

$G_{13} = G_{12} \sim \{5 \to 6\}$ $G_{14} = G_{13} \sim \{6 \to 7\}$ $G_{15} = G_{14} \sim \{7 \to 8\}$ $H_1 = G_{15} \sim \{8 \to 5\}$

Fig. 6.10. The constructed chandelier

6.15 K-Rooted Trees and Related Concepts

For wheels with thickness greater than 1, our wheel trees have been studied by Chaty and Chein [14], which calls them **K-rooted trees** (K for Knuth). Using the terminology of this paper:

- a **reducible** graph is a wheel with size greater than 1;
- a **K-decomposition** of a reducible graph is the specification of W_1, W_2, ..., W_n and the x_i and the y_i in the wheel;
- a **totally reducible** graph is a wheel with thickness greater than 1;
- a **K-rooted tree** is what we are calling a wheel tree (but only for totally reducible graphs);
- the **depth** of a K-rooted tree is the maximum height of a wheel tree (when there is more than one wheel tree) for a totally reducible graph;
- an **optimum K-rooted tree** is a wheel tree with maximum height;
- the **order** of a compound circuit is the number of nodes in its wheel tree;
- a **circuitic** (or **circuitus**) extension of a node is the result of replacing the node by a circuit (that is, a cycle) and edges to the node by edges to somewhere in the circuit.

Chaty and Chein now proceed to define a **contractible elementary cycle** C as one from which there is exactly one edge proceeding outward, and to which there is exactly one edge proceeding inward. Formally, the definition in [14] involves the quotient graph in which C is contracted to a point. (If the cycle is not contractible, there will be parallel edges in the quotient graph.) Chaty and Chein now define a sequence of graphs, each one obtained from the previous one by a circuitus extension, and show (using our notation) that a wheel has a wheel tree of thickness greater than 1 if and only if it can be derived from a single point using a sequence of circuitus extensions as above.

6.16 Generalizations to Graphs of Higher Degree

The notions of 1-minimal graphs, 1-uniform graphs, and descendants of a non-minimal graph are generalized by Luce [5] to graphs of degree greater than 1. In such a case, Luce speaks of *k*-minimal graphs, *k*-uniform graphs, and *k*-descendants of a graph.

A ***k*-uniform** graph is one in which every strongly connected subgraph has degree not greater than *k*. A graph of degree *k* which is not *k*-uniform can easily be produced by starting with some graph of degree *k* and attaching to it a graph of degree higher than *k*.

A ***k*-minimal** graph is one in which removal of any edge results in a subgraph of degree *k*–1. For example, a double ring, like that of Fig. 6.1 (but having any number of nodes), is 2-minimal. Removal of any edge U→V leaves a graph of degree 1; it is strongly connected, but removing the remaining edge that starts at U leaves a disconnected graph (with no paths starting at U). In the same way, the double cube of Fig. 6.2 is 3-minimal. Here removal of any edge U→V leaves a graph of degree 2, since removing the remaining two edges that start at U leaves a disconnected graph as before.

For $k \geq 2$, Luce proves that every *k*-minimal graph is *k*-uniform, and of degree *k*. This generalizes the fact that a minimal graph is uniform and of degree 1, although the precise statement of Luce's lemma is not always true for $k = 1$. The problem is that a 1-minimal graph does not have to be

strongly connected, and indeed any graph which is not strongly connected is 1-minimal, although possibly of degree 0, not 1.

The removal of edges from a graph G of degree 1 is now generalized in [5] to the removal of edges from a graph of degree k, producing a graph which is q-minimal for some $q \leq k$. This is called a q-descendant of G; and Luce states (without providing counterexamples) that q-descendants do not always exist for $q > 1$, although they do always exist for $q = 1$.

6.17 Decompositions of Graphs of Higher Degree

In [5], Luce also generalizes his first decomposition theorem to graphs of degree higher than 1. Let us consider a graph G of degree 2, since that case may be easily visualized. Here the decomposition is into three graphs, G_1, G_2, and G_3, each of which contains all the nodes of G. The graph G_1 is minimal, and therefore strongly connected; G_2 is 1-minimal, but not necessarily strongly connected, and therefore not necessarily minimal. (In the generalization to degree k, it is still only G_1 that must be strongly connected.) Also, G_3 must be not strongly connected, and is therefore always 1-minimal. However, any strongly connected subgraph of G_2 is minimal. Finally, G_1 is a descendant of the sum of G_1 and G_2, which is in turn a 2-descendant of G.

To illustrate this, consider the graph G of Fig. 6.11, which has degree 2; if we remove B→A and B→C, the result is not strongly connected, because there are no longer paths to anywhere from B. Here G is the sum of G_1, G_2, and G_3, as in Fig. 6.12. Note that G_3, here, is not strongly connected, because of the isolated nodes B and D. Also, G_2, here, happens to be strongly connected, so that both G_1 and G_2, here, are minimal (and, in fact, interchangeable).

For a general graph of degree $k > 2$, the decomposition is always into the sum of k+1 subgraphs, all of which are 1-minimal. The first of these, G_1, is always connected (and therefore minimal), while the last, G_{k+1}, is always disconnected. The remaining G_2 through G_k may be either connected or disconnected, but their connected components are minimal. Each sum $G_1+...+G_j$ is always a j-descendant (as at the end of section 6.16 above) of the sum $G_1+...+G_{j+1}$.

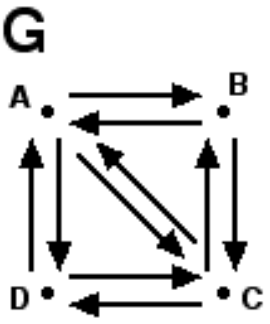


Fig. 6.11. Decompositions of graphs of higher degree

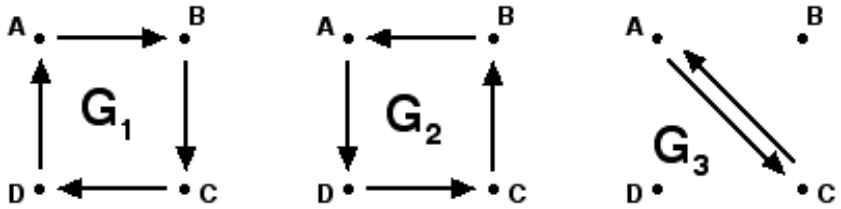


Fig. 6.12. The decomposed graph

6.18 Decomposition Trees and Isolated Paths

We now ask whether a graph, having been decomposed as above, may be decomposed further, producing a tree bearing certain similarities to a loop tree. This question is taken up in [5] as follows: "...the study of an arbitrary network has been reduced to the study of a collection of 1-minimal networks. These...are either connected, and so minimal, or disconnected. But a disconnected network consists of isolated nodes, isolated chains, and connected pieces...the connected pieces are minimal. If the theorem is applied repeatedly to the connected pieces...it may, in the same sense, be reduced to isolated nodes, isolated chains, and minimal subnetworks."

Luce's use of the term "isolated chains" (meaning, in modern terminology, isolated paths) is different from what is used today. An isolated node is one which is separated completely from all other nodes; in other words, there are no edges to or from an isolated node. Two paths today, therefore, would be considered isolated if they were separated completely from each other, meaning, in this case, that they had no nodes in common. However, a disconnected network (that is, a graph which is not strongly connected) does not necessarily consist of isolated nodes, connected pieces, and (in the above sense) isolated chains, as may be seen from the graph of Fig. 6.13. Here the connected pieces are (induced by) {1, 2, 3} and {6, 7, 8}; but the remainder of the graph is (induced by) {2, 3, 4, 5, 6}, and this is not made up of paths having no node in common. Isolated chains, for Luce, then, are chains having no *links* (i. e., edges) in common, such as $2\rightarrow4\rightarrow5\rightarrow6$ and $3\rightarrow4$ here.

6.19 A Decomposition Tree Example

We now give an example of a decomposition tree for the double cube G of Fig. 6.2. This is 3-minimal, and therefore 3-uniform and of degree 3, as we saw in section 6.16 above. Here G has 24 edges, of which we remove eight, producing a double ring on eight nodes, which we call G_1; and we denote the remaining edges by G_2. All this is shown in Fig. 6.14.

We now use Luce's Lemma 2.1 (p. 703 of [5]) which says that a graph of degree *k*, having *m* nodes, must have at least *km* edges. Here *k* is 2 and *m* is 8, so any graph of degree 2 on these nodes must have at least 16 edges. We use this to show that G_1 is a 2-descendant of G. On one hand, we have eliminated eight edges from G to produce the double ring G_1, which is 2-minimal (and therefore 2-uniform and of degree 2, as we saw in section 6.16 above). On the other hand, if we had eliminated more than eight edges, the resulting graph would have fewer than 16 edges and therefore, as we saw above, could not have degree 2. This shows that 8 is the maximum number of edges we can remove, and still leave a graph of degree 2.

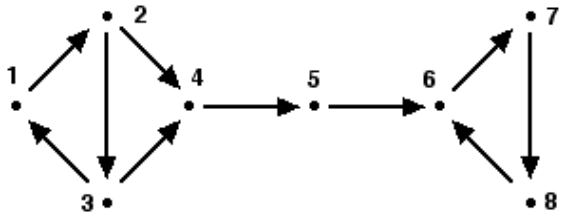


Fig. 6.13. Decompositions of disconnected networks

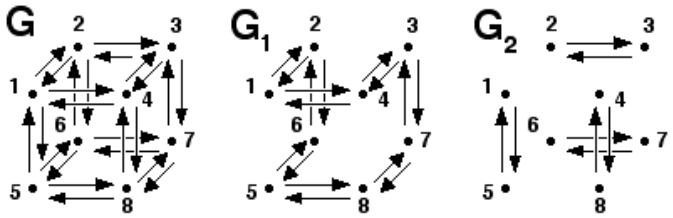


Fig. 6.14. Decomposition of a 3-minimal graph

The graph G_2 is not strongly connected, but it is made up of four strongly connected pieces, which we denote by K_1, K_2, K_3, and K_4, as in Fig. 6.15. We now obtain a 1-descendant of the graph G_1, which is a single ring that we denote by H_1. The eight edges which we remove from G_1 to get H_1 will be referred to as H_2, as in Fig. 6.16. By an argument similar to the one above, we show that H_1 is a 1-descendant of G_1 here. By eliminating eight edges from G_1, we obtain H_1, which is 1-minimal; eliminating any edge from H_1 produces a graph which is not strongly connected. On the other hand, if we were to eliminate more than eight edges from G_1, we would obtain a graph with fewer than eight edges, but still having eight nodes, so that it could not be strongly connected.

The decomposition tree for G is therefore that of Fig. 6.17. The main difference between this tree and a loop tree is concerned with G_2, which, as we saw above, is not strongly connected. In a loop tree, every node, other than the root, represents a loop, which must be a strongly connected subgraph.

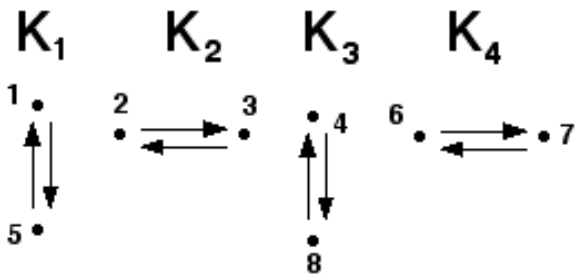


Fig. 6.15. Decomposition of a 3-minimal graph (continued)

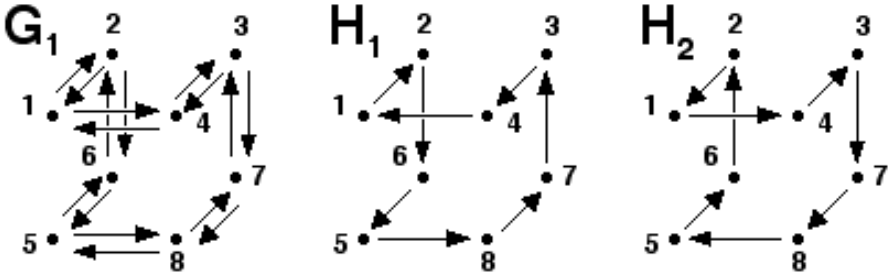


Fig. 6.16. Decomposition of a 3-minimal graph (further continued)

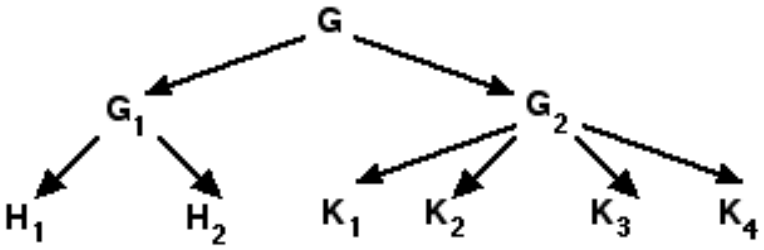


Fig. 6.17. Decomposition of a 3-minimal graph (concluded)

6.20 L-Trees and Proper Descendants

We define, in the obvious way, a **proper descendant** of G; that is, a descendant H of G such that H ≠ G. The following was stated, but not proved, in section 6.19 above. It was proved in [5]; we prove it again here, using modern terminology.

LEMMA (Lemma 3.1, p. 707 of [5]). A proper descendant of a strongly connected graph G cannot be an L-tree.

PROOF. Suppose that H is an L-tree which is a proper descendant of G, obtained by removing edges e_1, ..., e_k from G, in that order. Let H´ be H with the additional edge e_k = U→V, so that H´ is not minimal. We show that H is a proper descendant of H´, as well as of G. Suppose the contrary; since H is minimal, there must be another minimal graph H", obtained by removing at least two edges from H´. Then H" would be obtained from G by removing e_1, ..., e_{k-1} (to produce H´), followed by at least two more edges; that is, at least k+1 edges in all. This would contradict our assumption that H is a proper descendant of G, obtained by removing as many edges of G as possible, with what remains still being strongly connected.

We obtain a contradiction by showing that such an H" exists after all. Since H is strongly connected, there is a path π in H from V to U. We first show that the length of π is at least 2. Otherwise, it would have length 1, so it would be a single edge V→U. However, since H is an L-

tree and it contains V→U, it also contains U→V. That would contradict the assumption that U→V is a new edge, added to H to produce H´.

We now obtain H" by removing π from H´. Since π has length at least 2, we are removing at least two edges from H´, as noted above. It remains to show that H" is strongly connected. Let X and Y be in H"; we need to construct a path in H" from X to Y. Since H´ is strongly connected, and also contains X and Y, there is a path π´ in H´ from X to Y. If this path does not involve any edge in π, we are done.

Now let π be $V_1 \rightarrow V_2 \rightarrow ... \rightarrow V_k$, where V_1 = V and V_k = U, and suppose that π´ involves some edge $V_i \rightarrow V_{i+1}$, where $1 \leq i < k$. Since H is an L-tree, every edge in H is part of what Luce calls an arc (see section 6.9); therefore H also contains the path $V_k \rightarrow V_{k-1} \rightarrow ... \rightarrow V_1$. We now replace $V_i \rightarrow V_{i+1}$ in π´ by the path $V_i \rightarrow V_{i-1} \rightarrow ... \rightarrow V_1 \rightarrow V_k \rightarrow V_{k-1} \rightarrow ... \rightarrow V_{i+1}$, which is in H" (note that $V_1 \rightarrow V_k$ is in H", since V_1 = V and V_k = U). This completes the proof.

6.21 The Degree of a Flowgraph

In this and the next three sections, we take up the question of whether Luce's work in [5] motivated by a study of communication networks, has much relevance to flowgraphs, which motivated our work described in [1]. The answer appears to be no, but the details may be of some mathematical interest.

Luce states in [5] that his first decomposition theorem reduces the study of all graphs to that of graphs of degree 1. However, for flowgraphs this would not appear to be of great importance, because almost all flowgraphs have degree either 0 or 1 anyway. In fact, we have the following:
Any flowgraph containing an assignment statement is of degree either 0 or 1.

To prove this, first assume that the flowgraph G is not of degree 0, so that it is strongly connected. Consider an assignment statement S in G, and let T be the next statement after S; note that T is always executed immediately following S. Now remove the link in G from S to T, producing a graph G´. There are no paths in G´ from S to any other node, because all such paths would have to go through the link that was just removed. Hence G´ is not strongly connected, and G is thus of degree 1.

It is possible, although of doubtful utility, to construct a flowgraph of degree 2. For example, we could do this for the double ring of Fig. 6.1; every statement is now an if-statement, passing control by one position either one or the other way around the ring, depending on the result of a condition. One could include assignment statements as side effects of the if-statements, in order to do useful work.

Mathematically, we could even construct a flowgraph of any degree higher than 2, by using a **case** or **switch** statement at every node. For degree 3, each node is now one vertex of a cube, which has three adjoining vertices, to any of which control may pass depending on which case is applicable, as in the double cube of Fig. 6.2. For degree $n > 3$, each node is one vertex of a hypercube in n-dimensional space, having n adjoining vertices.

6.22 Minimal Flowgraphs

In section 6.21 above we mentioned that a flowgraph might not be minimal for a good reason. Having said this, it is conceivably still of interest to determine which flowgraphs G are minimal and which are not. The answer appears a bit strange, even for flowgraphs of structured programs without the go-to. Let us assume that G is strongly connected, and that we build up G by replacing parts of it by more complex parts. If G is minimal, and we replace L1: S; L2: within it by some other construction Z, then the result *is* minimal in each of the cases (1) through (6) below:

1) If Z is L1: S1; L3: S2; L2:. Here we had an old edge, L1→L2, which does not remain; and there are now two new edges, namely L1→L3 and L3→L2, and the removal of either of these leaves the graph not strongly connected.
2) If Z is an **if**-statement with **else**, of the form L1: **if** C **then** L3: S1 **else** L4: S2; L2:, where S1 and S2 are single nodes. There are four new edges, namely L1→L3, L3→L2, L1→L4, and L4→L2, and the removal of any of these leaves the graph not strongly connected.
3) If Z is L1: **while** (C) {L3: S1;} L2:, where S1 is a single node. We had an old edge, L1→L2, which remains, and there are now two new edges, namely L1→L3 and L3→L1; and the removal of any of these leaves the graph not strongly connected.
4) If Z is a **while** (**true**) construction containing a single break. Removing the break leaves the graph not strongly connected, since there is no way to get to the statement following the **while** loop.
5) If Z is L1: **for** (C1; C2; C3) {S1;} L2:, where S1 is a single node. This is equivalent to an assignment followed by a while statement, and rules (1) and (3) above apply.
6) If Z is a **case** or **switch** statement in which every case is non-null, including the default case. Removing the link from the start of the statement to any case, or from any case to the end of Z, leaves the graph not strongly connected.

However, the result is *not* minimal in each of the cases (7) through (12) below:

7) If Z is an **if**-statement without **else**, of the form L1: **if** C **then** L3: S1; L2:, even if S1 is not a single node. We had an old edge, L1→L2, which remains; and there are two new edges, namely L1→L3 and L3→L2. The removal of L1→L2 now leaves a graph which is strongly connected.
8) If Z is L1: **do** S; L3: **while** (C); L2:, even if S is not a single node. We had an old edge, L1→L2, which does not remain, and there are now three new edges, namely L1→L3, L3→L1, and L3→L2; and the removal of L3→L1 leaves a graph which is still strongly connected.
9) If Z is a **while** (**true**) construction containing more than one **break**. Removing any **break** now leaves the graph strongly connected, since we can always get to the statement following the **while** loop by means of another **break**.
10) If Z is a **while** (C) construction containing **break**, one or more times, where C is anything other than **true**. Removing any **break** now leaves the graph strongly connected, since we can still get to the statement following the **while** loop through the normal **while** logic when C is **false**.
11) If Z is a conditional **continue**. Removing the edge which does the continue leaves the graph strongly connected.
12) If Z is a **case** or **switch** statement in which some case is null, and is represented by an edge from the start of the **case** to the statement following Z. Removal of that link leaves the graph strongly connected.

6.23 Null Node Expansions and Minimal Graphs

In considering whether a flowgraph G may be replaced by a flowgraph H which is equivalent to G, there is always the question of what equivalence means. Clearly H might provide exactly the same computations that G does, and, at the same time, H might be less efficient than G, with respect to time and/or space requirements. It is therefore useful to define a class of equivalent graphs which do not present this problem.

Null node expansions were introduced in section 5.7 above. There are many possible reasons to introduce null nodes. Here we will merely be concerned with one of these, related to [5]: *a non-minimal graph always has a null node expansion which is minimal.* Suppose we have an edge, L0→L1, the removal of which leaves the graph still strongly connected. We can always replace L0→L1 by two edges, L0→L2 and L2→L1, where L2 is a null node, as in section 5.7. Removing either of these edges leaves the graph not strongly connected; and if we do this for every such edge, we will have a minimal graph. (In example (12) of the preceding section, this would be minimal if the null case had its own node.).

6.24 Wheel Trees and Loop Trees

An individual loop, within a loop tree, may be derived from a general acyclic graph with loopbacks, by replacing some of its elements by inner loops. An individual wheel, within a wheel tree, may be derived from a simple cycle by replacing some of its elements by inner wheels. This greater simplicity of wheel trees, however, comes at a cost; the height of a wheel tree is often greater than the height of the corresponding loop tree. We will now illustrate this with an example. Consider the following program:

```
(A)     s0;
(B)     do {
          switch (expr) {
(C1)        case 1: while (cond1) s1; break;
(C2)        case 2: while (cond2) s2; break;
              . . .
(Ck)        case k: while (condk) sk;
          }
(E)     } while (cond)
(F)     f;
```

The flowgraph of this program is given in Fig. 6.18. Consider the outer loop here, that is, the entire graph except for the nodes A and F. A loop tree for that graph consists of the **do** loop, containing k **while** loops, all at the same level. The height of this loop tree, therefore, is 2. We now prove that any wheel tree for this graph has height at least k+1. If k = 1, the wheel tree also has height 2 (= k+1); B, C_1, and E form the outer wheel, while C_1 and D_1 form the inner wheel.

If k > 1, then let W be a representation of this graph as a wheel, as in section 6.11, containing inner wheels W_1, W_2, ..., W_n. Let W_1 be the inner wheel that contains D_k. There are four cases, only three of which are actually possible:

1) W_1 consists of D_k by itself. In that case, as we can see from Fig. 6.18, $x_1 \rightarrow y_2$ must be $D_k \rightarrow C_k$, while $x_n \rightarrow y_1$ must be $C_k \rightarrow D_k$. Therefore $y_2 = x_n$ and so $n = 2$, with W_2 being the remainder of the graph. Otherwise, W_1 must contain C_k, since C_k is the only node with an edge either to or from D_k.
2) W_1 consists of D_k and C_k by themselves. In that case, again from Fig. 6.18, $x_1 \rightarrow y_2$ must be $C_k \rightarrow E$, while $x_n \rightarrow y_1$ must be $B \rightarrow C_k$. Let W_2 be the inner wheel that contains E. Since W_2 is strongly connected, it also contains B; but $B = x_n$, so that again $n = 2$ and W_2 is the remainder of the graph. In either of these cases, W_2 is strongly connected; also, any wheel tree of W_2 must have height at least $(n–1)+1 = n$, by induction, so any wheel tree for W must have height at least $n+1$.

In the remaining two cases, W_1 contains D_k, C_k, and either B or E, since these are the only nodes with edges to or from C_k. If W_1 contains E, then, since W_1 is strongly connected, it also contains B; so, in any case, W_1 contains B.

3) W_1 contains all D_i except for some D_j, where $j \neq k$. Here W_2 contains D_j, and W_2 does not contain B, so that W_2 must be either D_j by itself, or D_j and C_j by themselves. This then reduces to case (1) or (2) above, with j taking the place of k.

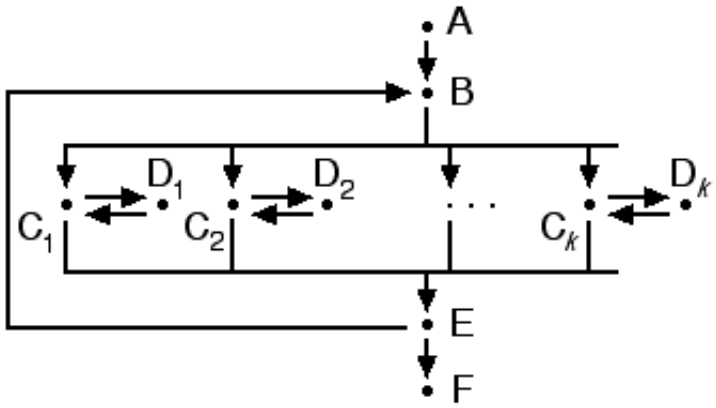


Fig. 6.18. The height of a wheel tree compared to that of a loop tree

4) There exist two D_i, say D_a and D_b, which are not contained in W_1; but W_1 still contains D_k, C_k, and B. We now show that this case is impossible. Clearly D_a and D_b cannot be in the same inner wheel, since that wheel would have to contain B; but B is actually in W_1. Therefore, D_a and D_b are contained in distinct inner wheels, and there must now be a path around W, from D_a to D_b (or from D_b to D_a), which does not go through W_1. Since W_1 contains B, this path cannot go through B. This is impossible, however, because every path from D_a to D_b, or from D_b to D_a, in this graph must in fact go through B. This completes the proof.

A consequence of this is that wheel trees do not, in general, represent the looping depth of a graph, even though, like loop trees, they constitute a decomposition of strongly connected graphs into a tree structure, with a strongly connected subgraph at every level.

7 Conclusions

We have compared three methods of repeated decomposition of a directed graph, using strong components, namely wheel trees, clustering trees, and loop trees. We have shown that loop trees are a better source of understanding of the properties of flowgraphs than are clustering trees or wheel trees. We have also shown that there are well-known concepts in the theory of directed graphs that can be better understood by considering their loop trees. These include path expressions, edge-disjoint spanning trees, and feedback vertices. Our contention is that loop trees are the first important 21st-century development in the theory of directed graphs, particularly flowgraphs, and that anyone interested in this theory should learn about them.

Acknowledgment

The author is grateful to R. E. Tarjan for calling to his attention the work of the following authors, as work related to this author's own: Edmonds [9], on edge-disjoint spanning trees; Karp [12] on feedback sets and NP-completeness; Smith and Walford [3] and Garey and Tarjan [4], on feedback vertices, feedback sets, and vertex covers; Luce [5], Knuth [6], and Chaty and Chein [14], on an alternative method of decomposition involving strong components; as well as some of Tarjan's own work on clustering [13], edge-disjoint spanning trees [8], and path expressions [7].

Competing Interests

Author has declared that no competing interests exist.

References

[1] Maurer WD. Generalized structured programs and loop trees. Sci Comp Prog. 2007;67:223-246.

[2] Maurer WD. Loop trees for directed graphs and their applications. Technical Report TR-GWU-CS-05-004. Washington: Computer Science Dept, George Washington University; 2005.

[3] Smith GW Jr, Walford RB. The identification of a minimal feedback vertex set of a directed graph. IEEE Trans Circuits Syst. 1975;CAS-22(1):9-14.

[4] Garey MR., Tarjan RE. A linear-time algorithm for finding all feedback vertices. Info Proc Letters. 1978;7(6):274-276.

[5] Luce D. Two decomposition theorems for a class of finite oriented graphs. Amer J of Math. 1952;74(3):701-722.

[6] Knuth DE. Wheels within wheels. J Combinatorial Theory (B). 1974;16:42-46.

[7] Tarjan RE. Fast algorithms for solving path problems. J Assoc Comp Mach. 1981;28(3):594-614.

[8] Tarjan RE. Edge-disjoint spanning trees and depth-first search. Acta Informatica. 1976;6:171-185.
[9] Edmonds J. Edge-disjoint branchings. In: Rustin R, editor. Combinatorial algorithms. New York: Algorithmics Press; 1972.
[10] Manna Z. Mathematical theory of computation. New York: McGraw-Hill; 1974.
[11] Maurer WD. A minimization theorem for verification conditions. Waterloo, Ontario, Canada: Proc. 8th International Conf on Computing and Automation (ICCI '96, CD-based proceedings); 1996.
[12] Karp RM. Reducibility among combinatorial problems. In: Miller RE, Thatcher JW, editors. Complexity of computer computations. New York: Plenum Press; 1972.
[13] Tarjan RE. An improved algorithm for hierarchical clustering using strong components. Info Proc Letters. 1983;17:37-41.
[14] Chaty G, Chein M. A note on top down and bottom up analysis of strongly connected digraphs. Discrete Math. 1976;16:309-311.

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
www.sciencedomain.org/review-history.php?iid=360&id=6&aid=2624