LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-85

FINDING MINIMUM CUTSETS IN REDUCIBLE GRAPHS

ADI SHAMIR

JUNE 1977

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-85

FINDING MINIMUM CUTSETS IN REDUCIBLE GRAPHS

Adi Shamir

June 1977

MIT/LCS/TM-85

Finding Minimum Cutsets in Reducible Graphs

Adi Shamir

June 1977

Massachusetts Institute of Technology
Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

Finding Minimum Cutsets in Reducible Graphs

Adi Shamir

Abstract

The analysis of many processes modelled by directed graphs requires the selection of a subset of vertices which cut all the cycles in the graph. Reducing the size of such a cutset usually leads to a simpler and more efficient analysis, but the problem of finding minimum cutsets in general directed graphs is known to be NP-complete. In this paper we show that in reducible graphs (and thus in almost all the "practical" flowcharts of programs), minimum cutsets can be found in linear time. An immediate application of this result is in program verification systems based on Floyd's inductive assertions method.

This research was supported by the Office of Naval Research grant number N00014-76-C-0366.

I. Motivation

A directed graph is often used as a path-generating device, which models the succession of events (in the form of edge traversals) that can take place in some process. Two common examples are the graph representations of finite state machines (with edges labelled by symbols from some alphabet) and of flowcharts of computer programs (with edges labelled by instructions).

Finite directed graphs which do not contain cycles can describe only finitely many paths, each of which contains finitely many edges, and thus the path-analysis of these graphs is usually straightforward. The analysis becomes qualitatively different in the presence of cycles, since the number and length of the paths need not be finite any longer.

However, in many cases the path-analysis of arbitrary graphs can be reduced to that of cycle-free graphs by selecting an appropriate subset of vertices (called cutpoints) such that any cycle in the graph contains at least one cutpoint. These cutpoints dissect the graph in a natural way into cycle-free components, which can be analysed separately. All that remains to do is to relate the overall behaviour of the original graph to that of its components, and this is usually done by some kind of induction.

An important concrete example of such an analysis is Floyd's method for proving the partial correctness of computer programs (Floyd [1967]). Since execution sequences of instructions may be arbitrarily long (or infinite), one uses the selected cutpoints in the flowchart in order to "chop" them into subsequences of bounded size. If the correctness of the specifications attached to the cutpoints is preserved along any such subsequence, one can infer the overall correctness of the program by induction on the number of subsequences.

Graphs may have many sets of cutpoints, all of which are useful in principle (an example of a highly redundant set of cutpoints can be the set of all the vertices). In many cases, the number of cutpoints selected has a strong influence on the complexity of the subsequent analysis. For example, if each cutpoint gives rise to an equation (where the equated quantities may be numbers, logical formulaes, or sets of strings), and the time required in order to solve n such simultaneous equations is a rapidly growing function of n, then minimizing the number n of cutpoints can be very desirable.

The problem of finding the smallest set of vertices which cut all the cycles in a given directed graph is NP-complete (Karp [1972]), and thus the optimization of the set of cutpoints is probably too expensive for big graphs. However, in this paper we show that for reducible graphs, the smallest set of cutpoints can be found in linear time.

Reducible graphs occur naturally in connection with flowcharts of computer programs. All the flowcharts which have a clear loop structure (with uniquely-defined loop entries) are reducible graphs, and as observed empirically, most programs used in practice have this property. Reducible graphs have been extensively analysed in connection with problems of code-optimization (see Aho and Ullman [1973], volume 2).

As a typical application of the suggested algorithm, let us consider once more Floyd's method. An interactive implementation of this method needs a user-supplied set of inductive assertions, one for each cutpoint. Since assertions in non-trivial programs tend to be very long and very detailed, a small number of cutpoints can minimize the user's effort in finding the relations between program variables at each cut-point, in formalizing them as inductive assertions, in "fine tuning" their pairwise power so that they may imply each other, and even in entering them into the computer. Furthermore, the number of verification conditions that the computer must prove is proportional to the number of cutpoints, and thus reducing the number of cutpoints can shorten the verification time considerably.

One possible fallacy in this argument is that some selections of cutpoints may be more natural than others, giving rise to shorter or simpler inductive assertions.

However, a special feature of the suggested algorithm is that the selected cutpoints are always loop entries, and in most practical cases these places tend to be natural locations for inductive assertions.

II. Basic Definitions

A graph is a pair (V,E) where V is the set of vertices and $E \subseteq V \times V$ is the set of (directed) edges (an edge from v to its son v′ is denoted by $v \to v'$). A path from v to v′ is a sequence of zero or more edges in E of the form $v = v_1 \to v_2 \to \dots v_k = v'$ (if the intermediate vertices are known, we shorten it into $v \overset{*}{\to} v'$, and say that v′ is a descendant of v). A cycle is a non-empty path from a vertex to itself; it is simple if all the vertices along it (except the first and last) are distinct. A graph which does not contain cycles is called a dag (directed acyclic graph).

A rooted graph is a triple (V,E,r) such that (V,E) is a graph, the root r is in V, and for any $v \in V$ there is a path $r \overset{*}{\to} v$. A depth first search (DFS) of a rooted graph is a way of exploring a rooted graph, which can be implemented in linear time $O(|V|+|E|)$ on a pointer machine, using an auxiliary stack. A detailed description of DFS and its properties can be found in Tarjan [1972]. A DFS defines two possible orders on the vertices:

(i) preorder - the order in which vertices are pushed into the stack during the DFS.

(ii) postorder - the order in which vertices are popped from the stack during the DFS.

A DFS defines a partitioning of the edges into:

(i) Backward edges (or fronds): edges $v \to v'$ such that v' is already in the stack when v is pushed into the stack.

(ii) Dag edges: all the other edges. In the literature these edges are classified further into tree edges, reverse fronds and cross links; we shall not use this finer classification.

The classification of edges may depend both on the graph and on the order of search in the DFS. For a given $G = (V,E,r)$ and DFS α, we define the dag of G defined by α to be $G_d^\alpha = (V,E_d^\alpha,r)$, where E_d^α is the set of dag edges in E. G_d^α is always a rooted dag, and if any edge in $E \setminus E_d^\alpha$ is added to it, a cycle is generated.

III. Cutsets in Graphs

Definition: A set S of vertices in a graph G is a cutset if any cycle in G contains at least one vertex form S. A cutset S is minimum if for any other cutset s', $|s| \leq |s'|$. The vertices in a cutset are called cutpoints.

Note that a minimum cutset of G need not be unique (e.g., when G is a single cycle of length ≥ 2), but all the minimum cutsets have the same size.

<u>Definition</u>: The set of all cycles in a graph G is denoted by C_G. Given a set S of vertices, the set of all the cycles in G which are <u>not</u> cut by vertices in S is denoted by C_G^S.

Clearly, C_G^{ϕ} is the initial set of cycles C_G, and a set S is a cutset iff $C_G^S = \phi$.

A simple but important observation is that the cycle cutting problem is <u>monotonic</u> in the following sense:

<u>Lemma 1</u>: Let G be a graph and let s_1, s_2 be two sets of vertices such that $C_G^{s_1} \subseteq C_G^{s_2}$. Then the minimum number of vertices which should be added to S_2 to get a cutset is equal to or greater than the minimum number of vertices which should be added to S_1 to get a cutset.

<u>Proof</u>: Let S_2' be a minimum set of vertices such that $S_2 \cup S_2'$ is a cutset. Any cycle in $C_G^{s_1}$ is a cycle in $C_G^{s_2}$, and thus must be cut by some vertex in S_2'. Consequently, $S_1 \cup S_2'$ is also a cutset in G, and the minimum number of vertices that should be added to S_1 to obtain a cutset can not exceed $|S_2'|$.

Q.E.D.

This lemma actually asserts that in order to solve a harder problem, more cutpoints are needed. We now use this property of the problem in order to describe an iterative process by which minimum cutsets can be monotonically "grown". The inductive hypothesis used at each stage is that the current set S of vertices is a subset of some minimum cutset. Initially $S = \phi$, which clearly satisfies this hypothesis. When the set S of vertices becomes a cutset, the hypothesis about being a subset of a minimum cutset makes S itself a minimum cutset. The main problem is of course how to select a new vertex which can be added to an intermediate set S without violating the inductive hypothesis and without knowing what the minimum cutsets of G are.

The following theorem shows that under certain (strong) conditions, this can be safely done:

<u>Theorem 1:</u> Let S be a subset of some minimum cutset in a graph G, and let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ be an uncut cycle in C_G^S. Suppose that v_1 has the property that any cycle in C_G^S cut by some v_i $(2 \leqslant i \leqslant k)$ is also cut by v_1. Then there exists a minimum cutset in G which contains $S \cup \{v_1\}$.

<u>Proof</u>: By assumption any cycle in C_G^S left uncut by v_1 is also left uncut by $v_2, v_3, \ldots, v_k$, and thus $C_G^{S\cup\{v_1\}} \subseteq C_G^{S\cup\{v_i\}}$ for all $2 \leqslant i \leqslant k$. By Lemma 1, the size of the minimum cutset containing $S\cup\{v_1\}$ is smaller than or equal to the size of the minimum cutset containing $S\cup\{v_i\}$ However, the cycle $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ is not cut by vertices in S, and thus any cutset containing S must also contain $S\cup\{v_j\}$ for some

$1 \leqslant j \leqslant k$. Consequently, $S \cup \{v_1\}$ is contained in some minimum cutset in G.

Q.E.D.

The problem in applying Theorem 1 is that in general, there need not exist a cycle for which one of the vertices is superior to all the others in cycle-cutting power. However, as we shall show later in the paper, this is exactly the case if the graph G is reducible.

Note that a vertex v_1, which does not satisfy the condition in Theorem 1 with respect to the initial set $S = \phi$, may still satisfy the weakened condition at a later stage (with respect to a bigger S), since C_G^S becomes successively smaller. (For example, when a single uncut cycle remains in C_G^S, any vertex along this cycle can be taken as v_1). Thus even if very few vertices satisfy this condition initially, it is still possible to construct iteratively a full cutset if sufficiently many vertices become available at later stages.

IV. Reducible Graphs

A number of equivalent definitions of reducible graphs are known (see Hecht and Ullman [1974]). One of them is:

Definition: A rooted graph G is reducible if the dag of G defined by α, G_d^{α}, is the same for any DFS α of G.

The simplest example of a non-reducible graph appears in Fig. 1(a). A DFS of this graph can proceed either along $v_1 \rightarrow v_2$

or along $v_1 \to v_3$, giving the two decompositions illustrated in Fig. 1(b) and Fig. 1(c) (the backward edges are denoted by double arrows and the dag edges by single arrows).

On the other hand, the graph in Fig. 2 is reducible, since it is easy to verify that any DFS must recognize $v_3 \to v_1$ $v_4 \to v_2$ $v_5 \to v_3$ and $v_6 \to v_1$ as backward edges, and all the other edges as dag edges.

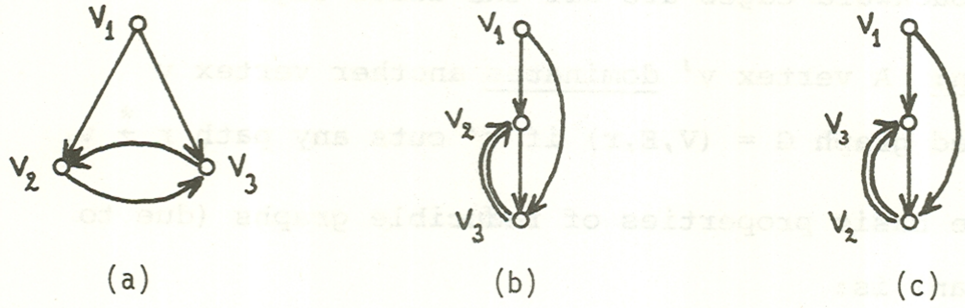

Fig. 1

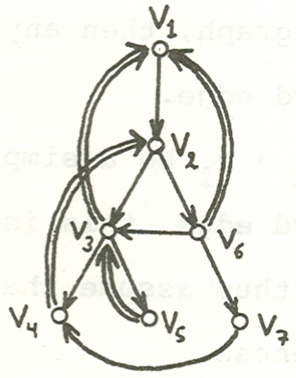

Fig. 2

In order to check whether big graphs are reducible, some less direct methods must be used. The best known algorithm appears in Tarjan [1974], and its time complexity is slightly more than linear in the size of the graph. Some classes of graphs can be shown to contain only reducible graphs, and thus special checks are not needed for them. For example, all the graphs which can be obtained by adding to a rooted tree some edges that always point from vertices to their tree ancestors are reducible. The dag of these graphs is the original rooted tree, and the backward edges are all the added edges.

<u>Definition</u>: A vertex v′ <u>dominates</u> another vertex v in a rooted graph G = (V,E,r) if v′ cuts any path $r \overset{*}{\rightarrow} v$.

One of the basic properties of reducible graphs (due to Hecht and Ullman) is:

<u>Lemma 2</u>: If v → v′ is a backward edge in a reducible graph G, then v′ dominates v.

Using this lemma, it is easy to prove:

<u>Lemma 3</u>: If G is a reducible graph, then any simple cycle in G contains exactly one backward edge.

<u>Proof</u>: Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ be a simple graph in G. If none of the edges is a backward edge, this is also a cycle in the dag G_d - a contradiction. We thus assume that $v_k \rightarrow v_1$ is a backward edge, and show its uniqueness.

If $v_i \rightarrow v_{i+1}$ is any other backward edge, consider a simple path P from the root r to v_{i+1}. If v_1 occurs along this path, then the path P' which follows P from r to v_1 and then proceeds to v_i through $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_i$ does not contain v_{i+1}. On the other hand, if v_1 does not occur along P, then the path P'' which follows P from r to v_{i+1} and then proceeds to v_k through $v_{i+1} \rightarrow v_{i+2} \rightarrow \dots \rightarrow v_k$ does not contain v_1. Both possibilities contradict the dominance condition.

Q.E.D.

Any cycle in a graph contains at least one simple cycle (just consider the first repeated occurrence of a vertex along the cycle, with respect to an arbitrary starting point). Lemma 3 provides a useful partitioning of the set of simple cycles in G in terms of their backward edges. Thus in order to check whether a given set S is a cutset, it suffices to check for any backward edge $v \rightarrow v'$ in G that all the forward dag paths from v' to v contain vertices from S.

We end this section by analysing the possible interactions between simple cycles in a reducible graph:

Theorem 2: Let C_1 and C_2 be two simple cycles in a reducible graph G, which have a common vertex w. If $u \rightarrow u'$ and $v \rightarrow v'$ are the backward edges in C_1 and C_2, respectively, then either u' or v' is contained in both C_1 and C_2 (see Fig. 3).

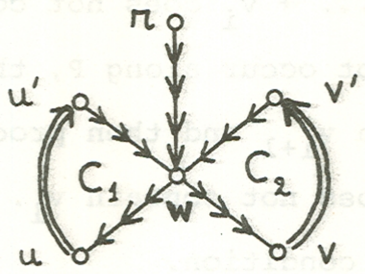

Fig. 3

Proof: Consider an arbitrary path P in the dag G_d from the root r to w. Since this path can be extended to u and v, both u' and v' must occur along P. Without loss of generality, we can assume that u' preceeds v' along P. Let P' be the path constructed in the following way:

(i) follow P from r to u';

(ii) follow C_1 from u' to w;

(iii) follow C_2 from w to v.

Since v' dominates v, it must be contained in one of these three segments. By assumption, it does not occur along segment (i) (unless u = v'), and by the properties of dags it cannot occur along segment (iii) (unless w = v'). The result that v' occurs along both C_1 and C_2 immediately follows.

Q.E.D.

V. Minimum Cutsets in Reducible Graphs

Definition: If $v \rightarrow v'$ is a backward edge in a reducible graph G, then v' is called a head and v is called a tail in G (we also say that v' and v are corresponding head and tail).

Definition: Let G be a reducible graph which is partially cut by a set S of vertices. Then a head v is active if there is some dag path from v to a corresponding tail, which does not contain vertices from S. An active head is maximal if none of its dag descendents in G is an active head.

Example: The heads in the graph in Fig. 2 are the vertices v_1, v_2 and v_3 (note that v_1 has two corresponding tails, and that v_2 is both a head and a tail in G). When $S = \{v_3\}$, the head v_1 is active (there is an uncut path $v_1 \rightarrow v_2 \rightarrow v_6$ to one of the two corresponding tails), the head v_2 is active (the paths $v_2 \rightarrow v_3 \rightarrow v_4$ and $v_2 \rightarrow v_6 \rightarrow v_2 \rightarrow v_4$ are cut by S, but the path $v_2 \rightarrow v_6 \rightarrow v_7 \rightarrow v_4$ is still open), and the head v_3 is not active. Consequently, the only maximal active head in G is v_2. $\square$

Note that if a head v is active, then there is at least one cycle in C_G^S which contains v, but not necessarily vice versa. However, unless S is a cutset, the graph G contains at least one active head, and thus also at least one maximal active head.

The main theorem justifying the cycle cutting algorithm can now be formulated as follows:

Theorem 3: Let G be a reducible graph, and let S be a subset of a minimum cutset in G. If v_1 is a maximal active head in G, then $S \cup \{v_1\}$ is also a subset of a minimum cutset in G.

Proof: Since v_1 is an active had, there is a simple cycle C_1: $v_1 \to v_2 \to \ldots \to v_k \to v_1$ in C_G^S in which $v_k \to v_1$ is the (unique) backward edge. If C_2 is any other cycle in C_G^S which is cut by some v_i $(2 \leqslant i \leqslant k)$, then C_1 and C_2 have a common vertex v_i . By Theorem 2, either v_1 or the head in C_2 (call it v') is contained in both cycles. If the active head v' is contained in C_2, then there is a dag path (along C_1) from v_1 to v' , which contradicts the maximality of the active head v_1 (unless $v_1 = v'$). Thus v_1 must cut C_2, and we can apply Theorem 1 in order to deduce that $S \cup \{v_1\}$ is a subset of a minimum cutset in G.

Q.E.D.

The basic algorithm is now a straightforward consequence of Theorem 3:

Algorithm A

1) Start with $S = \phi$.

2) Select a maximal active head v in G with respect to the current set S. If there is none, stop, otherwise set $S \leftarrow S \cup \{v\}$ and go on to step 2.

When implementing this algorithm, it is convenient to enumerate the heads in G by a DFS, and to consider them in postorder. By the properties of DFS, all the dag descendents of a vertex v (and in particular the active heads among them) occur before v in the postorder. Any head which is found to be active at some intermediate stage in the algorithm is immediately added to S, thus ceasing to be active with respect to the new S. Furthermore, the set S can only expand, and thus a non-active head cannot become active again at a later stage. Consequently, if heads are considered in postorder, then any head which is still active when its turn comes is a maximal active head.

Algorithm A can thus be implemented on a pointer machine in the following way:

<u>Algorithm B</u>

<u>Note</u>: "top" represents the vertex which is currently at the top of the stack.

```
1)  Set $S \leftarrow \phi$, push r into the (empty) stack.
2)  If there is an unmarked edge top $\rightarrow$ v, mark it and go to step 3,
    otherwise go to step 6.
3)  If v has not been visited so far, push v into the stack and
    go to step 2.
4)  If top $\rightarrow$ v is a backward edge, mark v as a head.
5)  Go to step 2.
6)  If v is marked as a head and is active with respect to the
    current set S, set  $S \leftarrow S \cup \{v\}$.
7)  Pop the top of the stack; if the stack is empty, halt,
    otherwise go to step 2.
```

A straightforward implementation of step 6, based directly on the definition of an active head, can be quite inefficient, but at least it shows that minimum cutsets in reducible graphs can be found in polynomial time. In the following section we optimize this "pedagogical" algorithm into a linear time algorithm.

VI. The Linear Algorithm

The simplest way of checking whether a given head v is active is to search for uncut dag paths between v and its corresponding tails. This can be done in linear time by propagating labels from v through the dag edges, but the labeling process has to be repeated for any maximal head, thus giving a $|V| \cdot |E|$ algorithm.

In order to develop a more efficient algorithm, we structure the search in such a way that each edge is used only once, even though it may belong to dag paths between many head-tail pairs in G. Since the search must convey sufficient information in order to determine which of these paths are cut and which are still open, it seems that we need labels that are sets of heads. Unfortunately, this method leads to non-linear algorithms even when the best known set manipulating techniques (Tarjan [1975]) are used.

As it turns out, the special structure of reducible graphs allows us to use much more economical labels. To each vertex v we attach a single number $\ell_s(v)$, which may change when new cutpoints are added to the current set S. Denoting by number(v) (an integer between 1 and $|V|$) the position of v in the preorder (rather than postorder) sequence of vertices in G, we define:

<u>Definition</u>: $\ell_S(v) = \max \{\text{number}(v') \mid$ there exists a backward edge $v'' \to v'$, and a dag path $v \overset{*}{\to} v''$ which is not cut by $S\}$.

If no such head v' exists, then $\ell_S(v)$ is defined to be 0.

<u>Example</u>: Consider the graph in Fig. 2, in which number$(v_i) = i$ for all the vertices. When $S = \{v_5\}$, there are only two heads (v_1, v_2) whose corresponding tails are accessible from v_3 through a path which is not cut by S. Since number$(v_2) >$ number(v_1), $\ell_S(v_3) = 2$. When $S = \phi$, on the other hand, there is also an uncut dag path $v_3 \to v_5$ to the tail of the backward edge $v_5 \to v_3$, and thus $\ell_S(v_3) = 3$. □

Let us now assume that these labels are initialized and updated (whenever S changes) by some external process, so that Algorithm B can take advantage of this extra piece of information. The following theorem shows that when successive active heads are considered and deactivated in postorder, step 6 in Algorithm B can be implemented as a simple check which requires only constant time:

<u>Theorem 4</u>: Let G be a reducible graph and let S be an arbitrary set of vertices in it. If v is a vertex such that none of its dag descendents is an active head, then v itself is an active head iff $\ell_S(v) = \text{number}(v)$.

Proof: We first show that if $\ell_S(v) = \text{number}(v)$, then v is an active head (the additional assumption about v is needed only for the other direction).

Since number(v) is always non-zero, $\ell_S(v) \neq 0$ and thus there exists a backward edge $v'' \to v'$ and an uncut dag path $v \overset{*}{\to} v''$ such that $\ell_S(v) = \text{number}(v')$. Since vertices are uniquely numbered, $v' = v$, and thus v is a head and v'' is one of its corresponding tails. Since $v \overset{*}{\to} v''$ is not cut by S, v is an active head in G.

On the other hand, if v is an active head then there is an uncut path from v to a corresponding tail v'', and by definition $\ell_S(v)$ is at least number(v). Suppose $\ell_S(v) = \text{number}(v') > \text{number}(v)$. Then there is a backward edge $v'' \to v'$ and an uncut dag path $v \overset{*}{\to} v''$. By the properties of preorder numbers, v is visited by the DFS before v'. If v' is not a dag descendent of v, then the dag-descendent v'' of v is also visited before v', contradicting the assumption that $v'' \to v'$ is a backward edge. Consequently, there is some dag path $v \overset{*}{\to} v'$ in G.

By the properties of reducible graphs, the head v' dominates the tail v'', and thus v must be contained in any path $r \overset{*}{\to} v''$. Consider in particular the dag path $r \overset{*}{\to} v \overset{*}{\to} v''$ in which $v \overset{*}{\to} v''$ is the given uncut path. The vertex v' cannot occur along the $r \overset{*}{\to} v$ subpath, since it is a dag descendent of v. Consequently, v' occurs along the uncut dag path $v \overset{*}{\to} v''$, and thus there is also an uncut dag path $v' \overset{*}{\to} v''$. This implies that v' is an active head, thus contradicting the assumption about v (note that the degenerate case $v = v'$ is impossible, since number(v') > number(v))

Q.E.D.

What remains to be done is to develop an efficient procedure for generating the labels $\ell_s(v)$. If we first calculate all these labels with respect to the initial set $S = \phi$, we may have to spend too much time updating them as S changes. Instead, we mix together the two operations of label calculation and cutpoint selection. At any intermediate stage in the process, only some of the vertices in G have labels (say, the subset L), and S is a subset of these labelled vertices. In order to make the process efficient, we must add new vertices to L and S according to the following rules:

(i) Vertices are added to L in postorder, i.e., a vertex is labelled only after the labels of all its dag sons are known. This order minimizes the effort involved in calculating the labels.

(ii) A vertex can be added to S only immediately after it is added to L. This order minimizes the effort involved in updating the labels, as S expands.

It is a convenient coincidence that the natural processing order in Algorithm B (which was defined for entirely different purposes) exactly satisfies these two conditions.

We now develop two procedures, one for adding a vertex to L (keeping s fixed) and one for adding a vertex to S (keeping L fixed). These "atomic" procedures preserve the correctness of the labels in the set L with respect to the cutpoints in the set S, and the overall correctness of the labelling process then follows by induction on $|L| + |S|$.

The procedure for extending L is motivated by the following theorem:

Theorem 5: Let G be a reducible graph and let S be an arbitrary set of vertices. Then for any vertex v in G, the following equation holds:

$$\ell_s(v) = \begin{cases} 0 & \text{if } v \varepsilon S \text{ or } v \text{ does not have descendents in G} \\ \max[\ell_s(v_1), \ldots, \ell_s(v_i), \text{number}(v_{i+1}), \ldots, \text{number}(v_k)] & \text{otherwise} \end{cases}$$

where $v \rightarrow v_1, \ldots, v \rightarrow v_i$ are all dag edges emanating from v, and $v \rightarrow v_{i+1}, \ldots, v \rightarrow v_k$ are all the backward edges emanating from v.

Proof: If $v \varepsilon S$ or v does not have sons, then $\ell_s(v)$ must be 0 since there cannot be any uncut dag path from v to a tail of a backward edge.

If v is not in S, then clearly $\ell_s(v) \geqslant \text{number}(v_j)$ for any vertex v_j such that $v \rightarrow v_j$ is a backward edge, since there is a trivial path from v to the tail v whose corresponding head is v_j. Similarly, if v is not in S then $\ell_s(v) \geqslant \ell_s(v_j)$ for any v_j such that $v \rightarrow v_j$ is a dag edge, since any uncut dag path from v_j to a tail can be extended to an uncut dag path from v to that tail. Consequently, $\ell_S(v)$ must be at least the maximum specified in the theorem.

On the other hand, $\ell_S(v)$ cannot exceed this maximum, since any dag path from v to a tail is either trivial, or else uses some dag edge $v \rightarrow v_j$ emanating from v. The corresponding head's number is thus bounded from above by at least one entry in the maximum. Q.E.D.

The equation in Theorem 5 shows how a new label can be computed in postorder from the known labels of its dag sons, and from the numbers of its corresponding heads. Note that while the labels of these backward sons are still unknown, they already have preorder numbers, and thus no preliminary graph traversal is needed in order to prepare these numbers. Note further that when the label of v is first computed, $v \notin S$, and thus the check $v \varepsilon S$ in the equation in Theorem 5 is superficial.

The procedure for computing these labels makes use of the fact that max is an associative operation, and thus it can be computed incrementally, taking new son values into account as they become available. The "temporary labels" thus obtained become valid labels as soon as all the sons of v are visited and we are ready to backtrack from v. The number of operations involved in adding a new vertex to L is equal to its out-degree, and thus all the $|V|$ vertices can be added to L by at most $|E|$ max operations.

We now consider the problem of updating the labels of vertices in L when a new vertex is added to S. As proved in the following theorem, at most one label can be affected if the rules of the game are observed.

Theorem 6: Let G be a reducible graph, let L be a set of vertices for which labels have been computed in postorder, and let S be a subset of L. If v is the next vertex added to L, then adding v to S leaves all the labels in L correct with respect to the new set SU{v}, and changes $\ell_{SU\{v\}}(v)$ to 0.

Proof: The fact that $\ell_{SU\{v\}}(v) = 0$ follows from the definition, since any dag path from v to a tail is cut by SU{v} .

If v′ is any other vertex for which $\ell_S(v')$ is already known, then all the dag descendents of v′ are in L, and thus none of them is v. Since the addition of v to S cannot affect the uncut dag paths from v′ to tails, $\ell_{SU\{v\}}(v') = \ell_S v')$.

Q.E.D.

The final algorithm uses a single DFS of G in order to number the vertices of G in preorder, to label them in postorder, to consider successive heads in postorder, and to add new vertices to S (changing their labels to 0) when their postorder label and preorder number coincide. It uses the same skeleton as Algorithm B, and its time and space complexities are clearly linear in the size |V|+|E| of G. Even though this algorithm is concise and easy to implement, its formal proof of correctness (say, using Floyd's inductive assertions method) is surprisingly subtle.

Algorithm C

(Note: labels are denoted by $\ell(v)$, without an explicit set subscript; the algorithm keeps only one system of such labels, which correspond at any stage to the current set S.)

```
1) Set $S \leftarrow \phi$; set vertex counter $c \leftarrow 1$; clear all flags;
   push r into the (empty) stack.
2) Set number(top) $\leftarrow c$, $c \leftarrow c+1$, $\ell$(top) $\leftarrow 0$ .
3) If there is some unmarked edge top $\rightarrow$ v, mark it and
   proceed, otherwise go to step 7.
4) If v has not been visited so far, push v into the
   stack and go to step 2.
5) If top $\rightarrow$ v is a backward edge, set
   $\ell$(top) $\leftarrow$ max($\ell$(top), number(v)) and to to step 3.
6) Set $\ell$(top) $\leftarrow$ max($\ell$(top), $\ell$(v)) and go to step 3.
7) If $\ell$(top) = number(top), set $S \leftarrow S \cup \{$top$\}$ and $\ell$(top) $\leftarrow 0$.
8) Save the current top of the stack in $v'$ ; pop the stack;
   if the stack is empty, halt, otherwise, set
   $\ell$(top) $\leftarrow$ max($\ell$(top), $\ell(v')$); go to step 3.
```

Example: We demonstrate the operation of Algorithm C on the graph G in Fig. 2. Note that the DFS in which left sons are considered before right sons gives rise to preorder numbers satisfying number$(v_i) = i$ for all i.

□

We start by setting $S \leftarrow \phi$ and pushing the root v_1 into the stack. We then proceed along the dag path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$, pushing v_2 v_3 and v_4 into the stack. The edge $v_4 \rightarrow v_2$ is then found to be a backward edge, and thus $\ell(v_4) \leftarrow \max(0,\text{number}(v_2)) = 2$. Since v_4 does not have other sons, we check that $\ell(v_4) = 2 \neq 4 = \text{number}(v_4)$, pop it from the stack, and set $\ell(v_3) \leftarrow \max(0,\ell(v_4)) = 2$. The vertex v_5 is then pushed into the stack, and the traversal of the backward edge $v_5 \rightarrow v_3$ sets $\ell(v_5) \leftarrow \max(0,\text{number}(v_3)) = 3$. Popping v_5 from the stack, we update $\ell(v_3) \leftarrow \min(2,\ell(v_5)) = 3$. We then consider the backward edge $v_3 \rightarrow v_1$, which sets $\ell(v_3) \leftarrow \max(3, \text{number}(v_1)) = 3$. Before popping v_3 from the stack, we discover that $\ell(v_3) = 3 = \text{number}(v_3)$, and we thus add v_3 to S and change its label to 0.

During the rest of the process, vertex v_7 gets the label 2 calculated earlier for v_4, vertex v_6 gets the label 2 (as $\max(0,\ell(v_3), \ell(v_7), \text{number}(v_1))$, and vertex v_2 gets the label 2 (as $\max(0,\ell(v_3), \ell(v_6))$. Before backtracking from v_2 to v_1 we again discover that $\ell(v_2) = 2 = \text{number}(v_2)$, and thus add v_2 to S, changing its label to 0. This leads to $\ell(v_1) \leftarrow \max(0,\ell(v_2)) = 0$, and the algorithm halts after v_1 is popped from the stack.

The minimum cutset found by the algorithm is $S = \{v_3,v_2\}$. It is not uniquely minimum since $\{v_5,v_2\}$ is also a cutset. All the other cutsets contain three or more vertices. □

Acknowledgements: Comments by M. S. Paterson and R. E. Tarjan helped simplify an earlier version of the algorithm.

Bibliography

1) A. V. Aho and J. D. Ullman [1973]: "The Theory of Parsing, Translation and Compiling", Prentice-Hall, 1973.

2) R. W. Floyd [1967]: "Assigning Meanings to Programs", Proc. Symp. Appl. Math. 19, 19-32.

3) M. S. Hecht and J. D. Ullman [1974]: "Characterizations of Reducible Flow Graphs, J. ACM 21, 367-375.

4) R. M. Karp [1972]: "Reducibility among Combinatorial Problems", in "Complexity of Computer Computations" (ed. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 85-104.

5) R. E. Tarjan [1972]: "Depth First Search and Linear Graph Algorithms", SIAM J. Computing 1, 146-160.

6) R. E. Tarjan [1974]: "Testing Flow Graph Reducibility", J. Computer and System Science 9, 355-365.

7) R. E. Tarjan [1975]: "Efficiency of a Good but Not Linear Set Union Algorithm", J. ACM 22, 215-225.