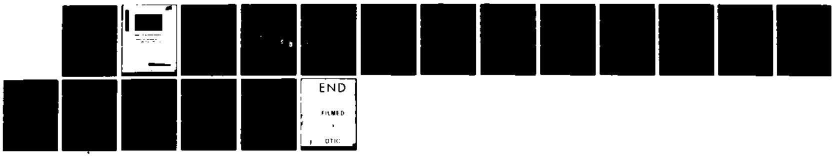
AD-A122 828 DERIVING EFFICIENT GRAPH ALGORITHMS(U) HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB J REIF ET AL. AUG 82 TR-30-82 N00014-80-C-0674 1/1

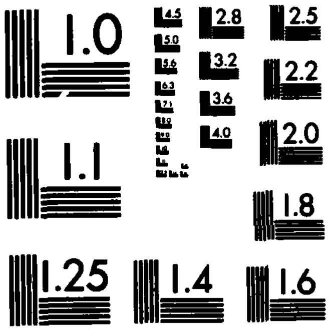
UNCLASSIFIED F/G 12/1 NL



END

FILMED

DTIC



2

ADA 122 828

Harvard University

Center for Research in Computing Technology

7 125

Aiken Computation Laboratory
33 Oxford Street
Cambridge, Massachusetts 02138

DERIVING EFFICIENT GRAPH ALGORITHMS

John H. Reif*

William L. Sherlis

TR-30-82

August, 1982

*This work was supported in part by the National Science Foundation Grant NSF-MCS79-21024 and the Office of Naval Research Contract N00014-80-C-0647.

AD A122828

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Deriving Efficient Graph Algorithms		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Reif, William Scherlis		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0674
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University, Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, 800 North Quincy Street, Arlington, VA 22217		12. REPORT DATE August, 1982
		13. NUMBER OF PAGES 13
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) same as above		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC ELECTE DEC 28 1983 S D H

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

graph algorithms, connectivity, biconnectivity, depth-first-search algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see reverse side

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

Abstract. Ten years ago Hopcroft and Tarjan discovered a class of very fast algorithms for solving graph problems such as biconnectivity and strong connectivity. While these depth-first-search algorithms are complex and can be difficult to understand, the problems they solve have simple combinatorial definitions that can themselves be considered algorithms, though they might be very inefficient or even infinitary. We demonstrate here how the efficient algorithms can be systematically *derived* using program transformation steps from the intuitive but preliminary definitions.

There are several justifications for this work. First, we believe that the evolutionary approach used in this paper offers more natural explanations of the algorithms than the usual *a posteriori* proofs that appear in textbooks. Second, the derivations illustrate several high-level principles of program derivation and suggest methods by which these principles can be realized by sequences of program transformation steps. Third, these examples illustrate how external domain-specific knowledge can enter into the program derivation process. This is the first occasion that such efficient graph algorithms have been systematically derived.

Deriving Efficient Graph Algorithms

John H. Reif
Aiken Computation Laboratory
Harvard University

William L. Scherlis
Department of Computer Science
Carnegie-Mellon University

Abstract. Ten years ago Hopcroft and Tarjan discovered a class of very fast algorithms for solving graph problems such as biconnectivity and strong connectivity. While these depth-first-search algorithms are complex and can be difficult to understand, the problems they solve have simple combinatorial definitions that can themselves be considered algorithms, though they might be very inefficient or even infinitary. We demonstrate here how the efficient algorithms can be systematically *derived* using program transformation steps from the intuitive but preliminary definitions.

There are several justifications for this work. First, we believe that the evolutionary approach used in this paper offers more natural explanations of the algorithms than the usual *a posteriori* proofs that appear in textbooks. Second, the derivations illustrate several high-level principles of program derivation and suggest methods by which these principles can be realized by sequences of program transformation steps. Third, these examples illustrate how external domain-specific knowledge can enter into the program derivation process. This is the first occasion that such efficient graph algorithms have been systematically derived.

1. Introduction.

Discovery of efficient algorithms is a complex and undisciplined task, requiring sophisticated knowledge of both general-purpose algorithm design techniques and special-purpose mathematical facts related to the problems being solved. While the *process* of algorithm discovery is certain to be exceedingly difficult to mechanize, there is much to be learned—both about algorithms and about programming—from the study of the structure of *derivations* of complex algorithms.

In this paper we demonstrate how program transformation techniques can be used to derive efficient graph algorithms from intuitive specifications. These specifications are simple combinatorial definitions that we choose to interpret as algorithms, even though—as algorithms—they might be very inefficient or even infinitary.

The derivations suggest ways in which algorithm-design knowledge separates from domain-specific knowledge. While the depth-first algorithms we derive depend on deep combinatorial properties of depth-first spanning forests, the algorithms can nonetheless be derived using only general-purpose program derivation techniques—supported by the necessary combinatorial lemmas. Indeed, we are optimistic that this sort of separation can be achieved in general, and surveys such as [Tarjan77] appear to support this possibility. If this proves possible, then the program derivation techniques refined and applied here and elsewhere can ultimately be of use in practical mechanical programming aids—aids designed primarily for the programmer, not the algorithm designer.

Program derivation techniques also provide a natural way of explaining complicated algorithms. Conventional proofs may succeed in convincing a reader of the correctness of an algorithm without supplying any hint of why the algorithm works or how it came about. A derivation, on the other hand, is analogous to a *constructive* proof; it takes a reader step by step from an initial algorithm he accepts as a specification of the problem to a highly connected and efficient implementation of it. As in [Clark80], we are deriving a *family of related algorithms.* Even though the algorithms we derive here do not all have the same specifications, the strong relations between them become manifest in the explicit structure of their derivations.

In Section 2 of this paper we derive a family of depth-first search algorithms. These are generalized and utilized in quite different ways in the biconnectivity algorithms of Section 3 and in the strong-connectivity algorithms of Section 4. These algorithms were discovered by Hopcroft and Tarjan and are (conventionally) presented in [Tarjan72] and [AHU74]. The variant of Tarjan's strong-connectivity algorithm that we derive in Section 4 is attributed to Kosaraju. (We can also apply similar techniques to derive the almost-linear-time algorithm of [Tarjan73] for flow-graph reducibility.) In the conclusion we discuss further the implications of this work.

Because we seek to demonstrate how derivations, clearly presented, can lead to a better understanding of the algorithms derived, the emphasis in this paper is primarily on the conceptual structure of the derivations and only secondarily on the actual formal transformation techniques. Indeed, most of the transformation techniques we use have appeared elsewhere, though perhaps in other forms. We make use of transformations for realizing complex recursive control structure as explicit data structure similar to those described in [Bird80], [Scherlis80], and [Wand80]. In addition, we make implicit use of transformations such as those described in [Burstall77] or [Scherlis81] to effect the merging or "jamming" of loops and to specialize function definitions. Discussion of loop jamming techniques also appears in [Paige81].

We do not, in this summary, specify precisely the programming language we use, except to say that it is a straightforward LISP-like (or ML-like) applicative language supplemented with assignment to variables and modification of data structures. For the sake of clarity of derivations, it is important that the programming language not be overly constrained. In particular, certain features that are difficult to implement but which have clear semantics often allow derivations to be quite straightforward. This is vividly illustrated in the case of the SETL language in the derivations of [Paige81]. Another example is the language used in [Scherlis81], which was extended (to include expression procedures—used, for example, in Algorithm 3.1 below) in order to keep the set of transformations simple and yet strong-equivalence preserving.

2. Depth-First Search.

We consider first the case of undirected graphs. Let $G = (V, E)$ be an undirected graph with adjacency list representation; for each $v \in V$, $Adj(v)$ is a canonically ordered list of edges connected to v. Observe that $v \in Adj(u)$ if and only if $u \in Adj(v)$.

Paths. As our starting point, we take the combinatorial definition of a path in a graph. Let u and v range over vertices.

$$path(u,v) \Leftarrow \left(u = v \quad \textbf{or} \quad (\exists w \in Adj(u))\, path(w,v) \right) \tag{2.0}$$

This definition, considered as an algorithm, has potentially infinite execution paths. Suitable semantics for the or operator would allow this algorithm to terminate correctly whenever there is a graph path, but for many graphs and vertex pairs this "algorithm" has no finite execution paths.

We can, however, distinguish two kinds of infinite execution paths—looping paths and divergent paths. Roughly, a nonterminating path is a *looping* path when only finitely many *distinct* recursive calls are made along that path; if the number of distinct calls grows without bound, then the path is *divergent*. In the case of finite graphs (the only graphs we will consider) Algorithm 2.0 can exhibit looping, but, because u and v are vertices and the set of vertices is finite, it cannot exhibit divergence.

By a semantic sleight of hand, looping evaluations can be replaced with finite ones. In the case of Algorithm 2.0, it is consistent with our interpretation to replace all looping paths with **false**. We effect this change by introducing explicit data structure to mark nodes as they are visited; by examining this data structure, the program can foreclose any potentially looping execution paths. (Transformations for carrying out this kind of change are sketched in [Scherlis80] and are related to the closed-world database techniques described in [Clark78] and [Reiter78].)

The transformation has two steps. First, we observe that the second parameter of *path* never changes and so can be made free. This reduces the number of possible recursive calls and hence the amount of data structure required.

```
path(u, v)  ⇐  vpath(u)
  where                                                        (2.1)
vpath(u)  ⇐  ( u = v  or  (∃w ∈ Adj(u)) vpath(w) )
```

Next, we introduce a boolean array, *visit*(V), initially **false** for each vertex $v \in V$.

```
path(u, v)  ⇐  begin visit(V) ← false;  vpath(u) end
  where
vpath(u)  ⇐                                                    (2.2)
    if visit(u) then false
      else begin visit(u) ← true;  ( u = v or (∃w ∈ Adj(u)) vpath(w) ) end
```

(The value of a block is the value of the last expression unless some other expression is marked by the word **value**, in which case the value of that expression is saved when it is evaluated and returned after evaluation of the remainder of the block is complete; by convention imperative statements are always enclosed in blocks.) This imperative program terminates for all finite graphs. Note that the same effect could be acheived in a purely applicative framework by adding another parameter (representing a continuation), but the resulting program would be less clear for our purposes.

The finite depth-first search algorithm is obtained by rotating the initial *visit* test from callee to caller and by writing *vpath* as a function, *dfs*, defined such that

$$v \in dfs(u) \Leftrightarrow path(u,v)\,, \qquad \text{i.e.,} \qquad dfs(u) = \{v \mid path(u,v)\}\,.$$

The function *dfs* will thus precompute the set of possible paths from a given vertex—the *connected component* associated with that vertex. (Again, we assume *visit*(V) is initially **false**.)

```
dfs(u)  ⇐
    begin
      visit(u) ← true;                                         (2.3)
      {u} ∪ $\bigcup_{w \in Adj(u)}$(if visit(w) then ∅ else dfs(w))
    end
```

This function can easily be derived by specializing the definition of *path* to the set computation specified above using the techniques of [Scherlis81]. (A more detailed example of specialization appears in Section 3 below.)

Connected Components. The set of connected components

$$comps(V) \Leftarrow \bigcup_{r \in V} \{\, \mathbf{begin}\ visit(V) \leftarrow \mathbf{false};\ dfs(r)\ \mathbf{end}\,\}$$

can be quickly computed by making use of the *visit* array.

$$comps(V) \Leftarrow \mathbf{begin}\ visit(V) \leftarrow \mathbf{false};\ \bigcup_{r \in V}(\mathbf{if}\ visit(r)\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \{dfs(r)\})\ \ \mathbf{end} \qquad (2.4)$$

(This is derived by using the applicative representation for the *visit* array and noting redundant components.) This program requires $O(|V| + |E|)$ time.

Depth-First Spanning Forests. The fast depth-first search algorithms are based on subtle combinatorial properties of the depth-first spanning forests implicit in the prior algorithms. In the case of undirected graphs, the depth-first search divides the edges of a graph into two sets, *tree edges*, the edges actually traversed during search, and the other edges, which are called *fronds*. We use the notation $u \rightarrow v$ to indicate tree edges and $u \nrightarrow v$ to indicate fronds.

We will occasionally need to distinguish the fronds explicitly during search. With respect to Algorithm 2.3, we observe that the fronds are exactly those edges (u, w) for which the *visit*(w) test is true but (since the graph is undirected) such that w is not the father of u in the search tree.

```
dfs(u) ⇐
    begin
      visit(u) ← true;
      {u} ∪ ⋃_{w∈Adj(u)}(if visit(w)                                          (2.5)
                            then (if w ≠ father(u) then assert[u ↛ w]); ∅
                            else assert[u → w ∧ father(w) = u]; dfs(w))
    end
```

Here we have decorated Algorithm 2.3 with assertions distinguishing the two sets of edges. The *father* function, which is defined implicitly by the assertion, can be realized explicitly using data structure, in effect transforming the assertion into an assignment. This requires a very simple proof by induction that the instance of *father*(u) in the test will have been previously assigned a value. (In the case of a root, *father* can return a special value, say Λ, that will cause the test to fail.)

Using the specialization techniques mentioned above, we can eliminate all references to the *father* function/array by introducing a new parameter to *dfs* that will be the father of u in the depth-first search tree. This requires a slight modification of the definition of *comps*. (For æsthetic reasons we also reorient the nested conditionals.)

```
comps(V) ⇐ ⋃_{r∈V}(if visit(r) then ∅ else {dfs(r, Λ)})
dfs(u, v) ⇐
    begin
      visit(u) ← true;                                                        (2.6)
      {u} ∪ ⋃_{w∈Adj(u)}(if ¬visit(w) then assert[u → w]; dfs(w, u)
                            elseif w ≠ v then assert[u ↛ w]; ∅
                            else ∅)
    end
```

Similar derivations can be carried out in the case of directed graphs; the resulting algorithms are simpler since the father tests (and associated parameters) are not needed.

Tree Orderings. All of the lemmas on which the depth-first-search algorithms are based make use of "non-local" properties of depth-first search trees; that is, they require testing relations between vertices that may be an arbitrary distance apart in the trees. In particular, both the biconnectivity and strong connectivity algorithms are based on lemmas that make use of *ancestor* or *descendent* orderings in the search forest. We make derivation steps here that will enable these relations to be precomputed entirely in the course of a single depth-first-search pass. (We now expand our discussion to include directed graphs as well as undirected graphs.)

The descendent ordering is the transitive closure of the ordering represented by the edges of the depth-first search forest. That is,

$$v \succ u \qquad \text{if and only if} \qquad \text{there is a path of tree edges from } u \text{ to } v.$$

We can introduce this ordering into the *dfs* program simply by adding an appropriate assertion, as we did in the case of *father*. After the entire graph has been traversed by *dfs*, the descendent ordering will be the transitive closure of the asserted relation, $\succ$. An easy induction proof can establish that no contradictory relations are asserted. (We have temporarily eliminated the father parameter in order to extend the applicability of this algorithm to directed graphs.)

```
$dfs(u)$ $\Leftarrow$
    begin
      $visit(u) \leftarrow$ true;                                                        (2.7)
      $\{u\} \ \cup \ \bigcup_{w \in Adj(u)}$(if $\neg visit(w)$ then assert$[w \succ u]$; $dfs(w, u)$ else $\emptyset$)
    end
```

We seek linear-time algorithms, so we will not be able to accept a naive implementation of this algorithm—computation of the transitive closure alone would typically require $O(V^3)$ time. We must therefore continue the derivation process and make use of further combinatorial properties. Since we are concerned with descendent orderings in *trees*, it is natural to consider introducing explicit pre- and endorder relations represented by numberings. Both numberings can be computed in linear time in a single tree traversal and, in combination, determine the descendent ordering.

LEMMA 2.1. Let T be a tree with vertices numbered in preorder and endorder (in arrays $pre(V)$ and $end(V)$). Then

$$u \succ v \qquad \text{if and only if} \qquad \begin{array}{l} pre(u) > pre(v) \quad \text{and} \\ end(u) < end(v)\,. \end{array}$$

That is, u is a proper descendent of v if and only if both u succeeds v in the preorder numbering and u precedes v in the endorder numbering.

Preorder and endorder numbers in the depth-first search forest can be assigned through the use of assignable free variables (that we will call p and e) in the *dfs* procedure. For brevity, we omit intermediate derivation steps that lead to the following imperative algorithm (on either directed or undirected graphs) for simultaneously calculating the two numberings.

```
$p \leftarrow 0$;   $e \leftarrow 0$;
$dfs(u)$ $\Leftarrow$
    begin
      $visit(u) \leftarrow$ true;                                                        (2.8)
      $pre(u) \leftarrow p \leftarrow p + 1$;
      value ( $\{u\} \ \cup \ \bigcup_{w \in Adj(u)}$(if $\neg visit(w)$ then $dfs(w, u)$ else $\emptyset$) );
      $end(u) \leftarrow e \leftarrow e + 1$
    end
```

(The value notation is explained in the remark following Algorithm 2.2.) If $pre(V)$ is initially zero then the *visit* array can be eliminated by replacing the **if** test with the test $pre(w) = 0$; we do this below.

Certain ancestry tests do not require use of both of the numberings. In particular, if it is known that two vertices are related by the descendent relation, but it is not known in which direction they are related, then (by a simple corollary of the Lemma above) either preorder or endorder suffices. Since the preorder numbering is also useful as a replacement for the *visit* array, we choose it. (The resulting algorithm will be applied in the next section to undirected graphs, so we reintroduce the father parameter and corresponding assertions.)

```
$p \leftarrow 0$;  $pre(V) \leftarrow 0$
$dfs(u, v) \Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p + 1$;                                                    (2.9)
      $\{u\} \cup \bigcup_{w \in Adj(u)}$(if $pre(w) = 0$ then assert[$u \rightarrow w$]; $dfs(w, u)$
                             elseif $w \neq v$ then assert[$u \twoheadrightarrow w$]; $\emptyset$
                             else $\emptyset$)
    end
```

3. Computing Biconnected Components.

Let $G = (V, E)$ be an undirected connected graph. An *articulation point* is a vertex whose removal disconnects G. A graph is *biconnected* if it has no articulation point. A *biconnected component* C is a maximal set of edges that contains no vertex whose removal disconnects the vertices contained in the edges of C.

Our derivation of the linear-time algorithm for detecting biconnected components is based on a technical lemma characterizing articulation points. Once the articulation points are found then the biconnected components can be collected in a single depth-first traversal. Assume that the depth-first search algorithm has been performed on a graph to obtain a depth-first search tree.

LEMMA 3.1. (Hopcroft–Tarjan) A vertex a of a depth-first search tree is an articulation point if (1) it is the root and has more than one son, or if (2) there is a son u of a none of whose descendents have a frond to a proper ancestor of a. That is, if a is not the root, then it is an articulation point exactly if it has a son u such that $Low(u) \succeq a$, where

$$Low(u) = \min_{\succ}(Lowset(u))$$

and $Lowset(u) = \{u\} \cup \{x \mid s \succeq u \wedge s \twoheadrightarrow x\}$.

Note that the elements of $Lowset(u)$ are always comparable under $\succ$, so we may use the Lemma 2.1 to replace our explicit use of the '$\succ$' ordering by tests on *pre* values, as in Algorithm 2.9.

We first derive an algorithm for detecting articulation points and then modify it to enable the biconnected components to be collected on the fly. Articulation point detection is achieved by specializing the *dfs* algorithm in three stages—computation of *Lowset*, computation of *Low*, and assertion of articulation points.

Lowset Computation. Since $s \succeq u$ if and only if $s \in dfs(u)$, we see that $Lowset(u)$ is equivalent to $\{x \mid s \in dfs(u) \wedge s \nrightarrow x\}$. We derive a program to compute the value of this expression by expanding the definition of dfs given by Algorithm 2.9. This will require introduction of the father parameter to dfs in the set specification. Direct substitution for dfs and preliminary simplification yield

```
$\{x \mid s \in dfs(u,v) \wedge s \nrightarrow x\}$  $\Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p+1;$
      $\{x \mid s \in \{u\} \wedge s \nrightarrow x\}$
          $\cup$ $\{x \mid s \in \bigcup_{w \in Adj(u)}($if $pre(w) = 0$ then assert$[u \rightarrow w]$; $dfs(w,u)$        (3.1)
                                          elseif $w \neq v$ then assert$[u \nrightarrow w]$; $\emptyset$
                                          else $\emptyset) \wedge s \nrightarrow x\}$
    end .
```

(For conciseness, we omit the initialization of p and pre.) Distribution of the set abstraction into the union and conditional and simplification give

```
$\{x \mid s \in dfs(u,v) \wedge s \nrightarrow x\}$  $\Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p+1;$
      $\{x \mid u \nrightarrow x\}$
          $\cup$ $\bigcup_{w \in Adj(u)}($if $pre(w) = 0$ then assert$[u \rightarrow w]$;                    (3.2)
                                               $\{x \mid s \in dfs(w,u) \wedge s \nrightarrow x\}$
                                   elseif $w \neq v$ then assert$[u \nrightarrow w]$; $\emptyset$
                                   else $\emptyset)$
    end .
```

We can now *form a recursion;* this is done by replacing both instances of the dfs set abstraction with a name, $Lowset(u,v)$ (equivalent to the old $Lowset$ with a father parameter added). Further, since all the fronds for u are computed inside the loop we decide to commute the outer union, postponing calculation of $\{x \mid u \nrightarrow x\}$ until the fronds have been detected.

```
$Lowset(u,v)$  $\Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p+1;$
      $\bigcup_{w \in Adj(u)}($if $pre(w) = 0$ then assert$[u \rightarrow w]$; $Lowset(w,u)$
                      elseif $w \neq v$ then assert$[u \nrightarrow w]$; $\emptyset$          (3.3)
                      else $\emptyset)$
          $\cup$ $\{x \mid u \nrightarrow x\}$
    end
```

Now since $\{x \mid u \nrightarrow x\}$ is equivalent to

$$\bigcup_{w \in Adj(u)}(\textbf{if } (u \nrightarrow w) \textbf{ then } \{w\} \textbf{ else } \emptyset),$$

we substitute this expression into Algorithm 3.3, merge the unions, and simplify on the basis of the assertions to get the final $Lowset$ program.

```
$Lowset(u,v)$  $\Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p+1;$
      $\bigcup_{w \in Adj(u)}($if $pre(w) = 0$ then $Lowset(w,u)$                  (3.4)
                      elseif $w \neq v$ then $\{w\}$
                      else $\emptyset)$
    end
```

(The assertions have been dropped to avoid clutter.)

Low Computation. A similar specialization sequence is now used to transform this algorithm into a program for *Low*(u, v), defined as $\min_{\succ}(\{u\} \cup \mathit{Lowset}(u, v))$, where v is the father of u.

```
Low(u, v)  ⇐
    begin
      $pre(u) \leftarrow p \leftarrow p + 1$;
      $\min(u, \min_{w \in Adj(u)}($if $pre(w) = 0$ then $Low(w, u))$                    (3.5)
                                   elseif $w \neq v$ then $w$
                                   else $\infty)$
    end
```

(Here ∞ denotes a maximal vertex value; note that u would do.) We obtained this program by expanding the definition of *Lowset* in the definition of *Low*, simplifying, and renaming.

Articulation Point Detection. We are now ready to locate articulation points. Excepting the root, the articulation points can be located simply by inspecting *Low* values in the depth-first-search tree. That is, u is an articulation point if $u \rightarrow w$ and $\mathit{Low}(w, u) \succeq u$. We specialize as before, but this time the effect is simply to introduce the appropriate assertion into the search algorithm.

```
Low(u, v)  ⇐
    begin
      $pre(u) \leftarrow p \leftarrow p + 1$;
      $\min(u, \min_{w \in Adj(u)}($if $pre(w) = 0$
                                    then begin if $\ell \succeq u$ then assert[$Art(u)$]; $\ell$ end     (3.6)
                                         where $\ell = Low(w, u)$
                                   elseif $w \neq v$ then $w$
                                   else $\infty)$
    end
```

(Just as with '$\succ$', if *Art* is not asserted for a particular vertex then that vertex is not an articulation point.)

As a final step, we make the min calculation explicit by introducing a new local variable in *Low*. At the same time, we replace instances of '$\succ$' with comparisons of *pre* values (following our earlier remark). The values of *Low* are thus now integers rather than vertices; our main interest, however, has become the assertions regarding articulation points.

```
$p \leftarrow 0$;  $pre(V) \leftarrow 0$;  $Low(r, \Lambda)$;
Low(u, v)  ⇐
    begin var $m$;
      $m \leftarrow pre(u) \leftarrow p \leftarrow p + 1$;
      for $w \in Adj(u)$ do
        if $pre(w) = 0$                                                                  (3.7)
            then begin if $\ell \geq u$ then assert[$Art(u)$]; $\ell$ end
                   where $\ell = Low(w, u)$
          elseif $w \neq v$ then $m \leftarrow \min(m, w)$;
      $m$
    end
```

Biconnected Components. The following lemma will enable collection of biconnected components to be performed simultaneously with the detection of articulation points. As before, assume that depth-first search has been performed on a given graph. Let C be a biconnected component and let V_C be the set of vertices contained in the edges of C.

LEMMA 3.2. (Hopcroft and Tarjan) For every biconnected component C, there is a unique vertex $a \in V_C$ such that $v \succeq a$ for all $v \in V_C$ and a is the root of the search tree or an articulation point.

[[For brevity, the remainder of this section has been omitted. It will appear in the final paper.]]

4. Strongly-Connected Components.

In this section we derive an interesting linear time algorithm for strong connectivity attributed to Kosaraju. Let us return to the original definition of *path*. Let $G = (V, E)$ be a directed graph and let u and v range over the vertices V.

$$\mathit{path}(u,v) \Leftarrow \left(u = v \quad \textbf{or} \quad (\exists w \in \mathit{Adj}(u))\, \mathit{path}(w,v) \right) \tag{4.1}$$

The *path* relation holds between u and v just when there is a directed path in G from u to v. Since we are dealing with directed graphs, it will be helpful to test *reverse paths* as well.

$$\mathit{revpath}(u,v) \Leftarrow \left(u = v \quad \textbf{or} \quad (\exists w \in \mathit{Adj}^{-1}(u))\, \mathit{revpath}(w,v) \right) \tag{4.2}$$

Clearly, $\mathit{path}(u,v)$ if and only if $\mathit{revpath}(v,u)$.

Two vertices u and v in a graph are *strongly connected* if $\mathit{path}(u,v)$ and $\mathit{revpath}(u,v)$ both hold. A maximal set of strongly connected vertices is called a *strongly connected component.* To find the strongly connected component associated with a particular vertex r, if suffices to collect all vertices u reachable from r such that $\mathit{path}(u,r)$. For, if both v and w have this property with respect to r, then by transitivity of the *path* relation, they are themselves strongly connected. This implies that strongly connected components are disjoint.

$$\begin{aligned} \mathit{strong}(V) &\Leftarrow \textstyle\bigcup_{r \in V} \{\mathit{sc}(r,r)\} \\ \mathit{sc}(u,r) &\Leftarrow \{u\} \cup \textstyle\bigcup_{w \in \mathit{Adj}(u)} (\textbf{if } \mathit{path}(w,r) \textbf{ then } \mathit{sc}(w,r)) \end{aligned} \tag{4.3}$$

(Note that *sc* is not yet a terminating program.)

The trick in this derivation comes from the observation that the second parameter of the *path* relation remains constant on all recursive calls of *sc* for a particular root. This suggests that we should be able to do a single depth-first traversal from r and use the orderings defined in Section 2 to test ancestry.

There are two ways we can obtain this advantage. First, we could use *revpath* instead of *path*, and compute ancestry using *its* depth-first search tree (since the *dfs* realizations of *path* and *revpath* do recursion on the first parameter). Alternatively, we could reverse the direction of the search in *sc* above (using Adj^{-1} instead of Adj), causing the path test parameters to be reversed, and thus use the *path* search tree. In either case, we will need to traverse the graph in both the forward and backward directions.

The situation is symmetrical, and we arbitrarily choose the latter alternative. By reversing Algorithm 4.3 and introducing a boolean array (as we did in Algorithm 2.2), we obtain

```
strong(V) ⇐ begin visit2(V) ← false; $\bigcup_{r \in V}$(if visit2(r) then $\emptyset$ else {scr(r,r)}) end
scr(u,r) ⇐
    begin
        visit2(u) ← true;                                                    (4.4)
        {u} ∪ $\bigcup_{w \in Adj^{-1}(u)}$(if ¬visit2(w) ∧ path(r,w) then scr(w,r))
    end
```

(We call the boolean array *visit2* to avoid name conflict with the boolean array used in the forward depth-first search.)

A Blind Alley. It may appear that we could obtain an acceptable implementation of Algorithm 4.4 by replacing $path(r, w)$ with the test $w \in dfs(r)$ and factoring the *dfs* calculation out of *scr* into *strong*.

```
$strong(V)$  $\Leftarrow$  begin
                 $visit2(V) \leftarrow$ false;
                 $\bigcup_{r \in V}$(if $visit2(r)$ then $\emptyset$ else $\{scr'(r, r, dfs(r))\}$)
             end
$scr'(u, r, D)$  $\Leftarrow$                                                        (4.5)
    begin
      $visit2(u) \leftarrow$ true;
      $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$(if $\neg visit2(w) \wedge w \in D$ then $scr'(w, r, D)$)
    end
```

Unfortunately, the set of roots required for the reverse-search forest is not necessarily the same as that required for forward search, and so the $dfs(r)$ calculation in *strong* could do redundant traversals.

Testing Ancestry. Rather than solving this problem directly, we choose instead to consider more subtle methods for realizing the *path* test. In particular, we follow the specialization methodology and make use of facts about the context in which the *path* test occurs in order to produce a specialized program for that test.

As in the case of biconnected component detection, we will need to state several structural lemmas. These lemmas will suggest methods by which the specialization can be achieved and hence by which strongly connected components can be detected quickly. Again as in the case of biconnectivity, the lemmas refer to depth-first search forests and the descendent relations they induce over vertices.

```
$forest(V)$  $\Leftarrow$  begin $visit(V) \leftarrow$ false;  for $r \in V$(if $\neg visit(r)$ then $dfs(r)$) end
$dfs(u)$  $\Leftarrow$
    begin
      $visit(u) \leftarrow$ true;                                                  (4.6)
      for $w \in Adj(u)$ do
        if $\neg visit(w)$ then assert[$w \succ u \wedge u \rightarrow w$]$dfs(w)$
    end
```

(As before, we will let '$\succ$' stand for its own transitive closure; thus, if u and v are vertices, then $u \succ v$ if u is a proper descendent of v.)

Assume that a depth-first search has been performed on a directed graph to obtain a depth-first search forest.

LEMMA 4.1. For each strongly connected component S there is a unique vertex r, called the *root* of S, such that $r = \min_{\succ}(S)$.

It follows from this lemma that if r is the root of a strongly connected component and there is a path from w to r then

$$path(r, w) \quad \text{if and only if} \quad w \succeq r$$

This fact will enable us to replace the *path* test in Algorithm 4.4 with a test of the descendent relation. Recall from Section 2 that the descendent relation can be computed efficiently using pre- and endorder numberings. (We do this in Algorithm 4.10 below.)

We will, however, have to modify the *strong* program so r ranges only over the strongly-connected-component roots, otherwise our replacement would be invalid. A further lemma below will enable us to obtain the roots easily.

```
$strong(R) \Leftarrow$ begin $visit2(V) \leftarrow$ false; $\bigcup_{r \in R} \{scr(r,r)\}$ end
$scr(u,r) \Leftarrow$
    begin
      $visit2(u) \leftarrow$ true;                                                        (4.7)
      $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)} ($if $\neg visit2(w) \wedge w \succeq r$ then $scr(w,r))$
    end
```

We assume here that R is exactly the set of strongly-connected-component roots. Since the components are disjoint and since by Lemma 4.1 each has exactly one root, *strong*(R) will yield all the strongly connected components in the graph.

Finding Roots. We must now consider the problem of efficiently locating the roots. Observe that it follows from the lemma above that every root in the depth-first search forest is the root of a strongly connected component. This suggests that we should modify *forest* to collect the roots of the depth-first search trees as they are explored.

$$forest(V) \Leftarrow \textbf{begin } visit(V) \leftarrow \textbf{false}; \ \bigcup_{r \in V} (\textbf{if } \neg visit(r) \textbf{ then } dfs(r); \ \{r\} \textbf{ else } \emptyset) \textbf{ end} \quad (4.8)$$

Finding More Roots. Algorithm 4.8 collects only a subset of the component roots. The following lemma suggests an approach for locating the remainder of the roots.

LEMMA 4.2. Let S be a strongly connected component. For each $v \in S$ and each $w \notin S$ such that $v \rightarrow w$, w is the root of a strongly connected component.

Since all vertices are included in the depth-first forest, the roots specified by this lemma together with the roots collected by *forest* comprise the set of all component roots. The following program returns a set of the new strongly connected component roots at the outgoing edges of a given component. Let r be the root of component S.

```
$update(r,S) \Leftarrow$
    $\bigcup_{v \in S} (\bigcup_{w \in Adj(v)} ($if $\neg visit2(w) \wedge v \rightarrow w$ then $\{w\}$ else $\emptyset))$      (4.9)
```

This program essentially implements the lemma, with the additional optimization of examining *visit2* to avoid collecting redundant roots. (We omit the derivation of this optimization.) Also, note that it is possible that the representation of S might make the outer union difficult to compute; though we do not do it here, it would then be advantageous to use the observation that the descendents of a strongly connected component can be found by depth-first search and do the search instead.

The Algorithm. We have now resolved the strongly connected component algorithm into two phases. First, *forest* is used to collect depth-first search forest roots and to precompute the pre- and endorder numbering used for testing ancestry. Second, *strong* is used to do reverse depth-first searches from these roots, collecting strongly connected components along the way. The *update* procedure is used to collect new roots as strongly connected components are found.

```
var $p, e, pre(V), end(V), visit2$;
$forest(V) \Leftarrow$
    begin
      $pre(V) \leftarrow 0$;
      $p \leftarrow e \leftarrow 0$;
      $\bigcup_{r \in V}$(if $pre(r) = 0$ then $dfs(r)$; $\{r\}$ else $\emptyset$)        end
$dfs(u) \Leftarrow$
    begin
      $pre(u) \leftarrow p \leftarrow p + 1$;
      value ( $\{u\} \cup \bigcup_{w \in Adj(u)}$(if $pre(w) = 0$ then assert[$u \rightarrow w$]; $dfs(w, u)$    else $\emptyset$) );
      $end(u) \leftarrow e \leftarrow e + 1$
    end
$strong(R) \Leftarrow$
    begin var $r, S$;                                                          (4.10)
      choose $r \in R$;
      $S \leftarrow scr(r, r)$;
      assert[$Component(S)$];
      $strong(update(r, S) \cup R - \{r\})$
    end
$scr(u, r) \Leftarrow$
    begin
      $visit2(u) \leftarrow$ true;
      $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$( if $\neg visit2(w) \wedge pre(w) > pre(r) \wedge end(w) < end(r)$
                                     then $scr(w, r)$ )
    end
$update(r, S) \Leftarrow$
    $\bigcup_{v \in S}$ ( $\bigcup_{w \in Adj(v)}$ (if $\neg visit2(w) \wedge v \rightarrow w$ then $\{w\}$ else $\emptyset$) )
```

Of course, the program derivation process has no definite termination criteria. We could continue improving this algorithm by realizing the various implicit loops, by frequency reduction (e.g., for *pre*(*r*) calculation), by eliminating set operations (e.g., in *strong*), and in many other ways. We conclude at this point, however, since the structure of the linear-time algorithm is now most clearly apparent and since the next set of derivation steps fall within the range of established techniques.

5. Conclusions.

This work is a step towards developing a new paradigm for the presentation and explication of complex algorithms and programs. It seems to us insufficient to simply provide a program or algorithm in final form only. Even with "adequate" documentation and proof, the final code cannot be as revealing to the intuition as a derivation of that code from initial specifications.

Ideally, a programming environment should support the programmer in the process of building derivations.

In a specific problem domain, such as graph algorithms, certain facts and fundamental algorithms should be available for access. The value of this store of facts should not be underestimated. In our derivations, for example, certain algorithms were repeatedly used as paradigms for the development of other algorithms. This kind of analogical development is similar in heuristic content to the goal-directed transformation of algorithms required to carry out the loop merging optimization or in order to create recursive calls during specialization.

We are still very far from automating the heuristic side of the derivation process. In fact, we argue that at this point our efforts are better directed at discovering and exercising useful transformations, developing foundations for proving their correctness, and developing tools for *interactive* program development that can make appropriate use of outside domain-specific knowledge. For example, it appears that once the necessary outside lemmas are stated and proved, only a modest deduction capability would required in such a programming environment; it would be used mainly to establish preconditions for transformations and application of lemmas.

Finally, by storing program derivations as data structures in a program development system, *program modifications* can be carried out simply by making changes at the appropriate places in the derivation structure; if only the final code is available, the conceptual history of the program must, in effect, be rediscovered.

Acknowledgements. We wish to thank Margaret Beard for her help in the initial stages of this work.

Bibliography

[AHU74] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[Bird80] Bird, R. S., *Tabulation techniques for recursive programs*. Computing Surveys(1980), Vol. 12, No. 4, pp. 403–417.

[Burstall77] Burstall, R. M. and J. Darlington, *A transformation system for developing recursive programs*. Journal of the ACM, Vol. 24, No. 1, pp. 44–67, 1977.

[Clark78] Clark, K., *Negation as failure*. In: Logic and Databases. Gallaire, H., and J. Minker, eds., Plenum, 1978.

[Clark80] Clark, K. and J. Darlington, *Algorithm classification through synthesis*. Computer Journal, Vol. 23, No. 1, 1980.

[Knuth74] Knuth D. E., *Structured programming with goto statements*. Computing Surveys, Vol. 6, No. 4, pp. 261–301, 1974.

[Paige81] Paige, R. and S. Koenig, *Finite differencing of computable expressions*. Rutgers University Technical Report LCSR-TR-8, 1981.

[Reiter78] Reiter, R., *On closed world data bases*. In: Logic and Databases. Gallaire, H., and J. Minker, eds., Plenum, 1978.

[Scherlis80] Scherlis, W. L., *Expression procedures and program derivation*. Ph. D. thesis, Stanford University, 1980.

[Scherlis81] Scherlis, W. L., *Program improvement by internal specialization*. Eighth Symposium on Principles of Programming Languages, pp. 41–49, 1981.

[Tarjan72] Tarjan, R. E., *Depth first search and linear graph algorithms*. SIAM Journal of Computing, Vol. 1, No. 2, pp. 146–160, 1972.

[Tarjan73] Tarjan, R. E., *Testing flow graph reducibility*. Fifth ACM Symposium on the Theory of Computing, pp. 96–107, 1973.

[Tarjan77] Tarjan, R. E., *Complexity of combinatorial algorithms*. Stanford Computer Science Report, 1977.

[Wand80] Wand M., *Continuation-based program transformation strategies*. Journal of the ACM, Vol. 27, No. 1, pp. 164–180, 1980.

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)