CMU-CS-82-155

Deriving Efficient Graph Algorithms

John H. Reif[1]
Aiken Computation Laboratory
Harvard University

William L. Scherlis[2]
Department of Computer Science
Carnegie-Mellon University

December 1982

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 initial definitions. This is the first occasion that these efficient graph algorithms have been systematically derived.

There are several justifications for this work. First, the derivations illustrate several high-level principles of program derivation and suggest methods by which these principles can be realized as sequences of program transformation steps. Second, 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. Third, these examples illustrate how external domain-specific knowledge can enter into the program derivation process. Finally, we believe that future programming tools will be semantically based and are likely to have their foundations in a logic of program derivation. By working through complex examples such as those presented here, we make steps towards a conceptual *and* formal basis for these tools.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms: Algorithms, Design, Languages, Verification

Additional Key Words and Phrases: Program Derivation

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

[2]This research was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and in part by the U.S. Army Communications R&D Command under Contract DAAK80-81-K-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1. Introduction.

Discovery of efficient algorithms is a complex and creative task, requiring sophisticated knowledge both of general-purpose algorithm design techniques and of 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.

Program derivation techniques provide a natural way of explaining and proving 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. Our approach, then, is to explicate algorithms by *justifying their structure* rather than by merely establishing their correctness.

Specifications and algorithms. In this paper we demonstrate how program transformation techniques can be used to derive efficient graph algorithms from intuitive mathematical 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. To illustrate how simple these specifications can be, we give here our specification of the path predicate for directed graphs. Let G be a directed graph with vertices V and adjacency-set function Adj. The predicate $path(u, v)$ is true when there is a path in G from vertex u to vertex v. That is,

$$path(u,v) \;\Leftarrow\; \big(\, u = v \quad \textbf{or} \quad (\exists w \in Adj(u))\, path(w,v)\big)\,.$$

Note that this definition—when interpreted as a sequential algorithm in the usual way—does not always terminate.

With this definition, we obtain a straightforward specification of the strongly-connected components of a directed graph. Two vertices u and v are in the same strongly-connected component if there is a path from u to v and a path from v to u. The strongly-connected components of G are thus the elements of the set $strong$, where

$$strong \;\Leftarrow\; \textstyle\bigcup_{r\in V}\ \{\{s \in V \mid path(r,s) \wedge path(s,r)\}\}\,.$$

Outline of paper. In Section 2 of this paper we derive a series of simple algorithms leading to a family of depth-first search algorithms. These are generalized and utilized in quite different ways in the strong-connectivity algorithms of Section 3 and in the biconnectivity 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 and is similar to the algorithm sketched in [AHU83]. (Similar techniques can be used to derive the almost-linear-time algorithm of [Tarjan73] for flow-graph reducibility.) In the conclusion we discuss further the implications of this work.

The use of combinatorial lemmas. The derivations suggest ways in which programming and algorithm-design techniques separate from domain-specific knowledge. While the depth-first algorithms we derive depend on deep combinatorial properties of depth-first spanning forests, this knowledge can be expressed in the form of a small number of lemmas. These lemmas are used to justify initial specifications of program components and to establish preconditions in later program derivation steps. While we could prove the lemmas entirely in the language of a programming logic, the resulting account of the algorithms would likely be awkward and unnatural. We have thus sought an appropriate balance in our use of facts from graph theory and our use of general-purpose program derivation techniques. (The balance will shift, of course, as our knowledge of programming techniques improves.)

Program transformation techniques. 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. We make use of transformations for realizing complex recursive control structure as explicit data structure that are similar to those described in [Bird80], [Scherlis80], and [Wand80]. These transformations, which are used to "coerce termination" in infinitary definitions such as specification of *path* above, are described in a separate report.

In addition, we make use of the transformations of [Scherlis81] (which are similar in spirit to those described in [Burstall77], but for which there is a guarantee of strong equivalence) in order to specialize function definitions and to effect the merging or "jamming" of loops. Discussion of loop jamming techniques also appears in [Paige81]. No prior knowledge of the details of these basic transformation techniques is required for the purposes of this paper.

Programming language for program derivation. The programming language we use is an ML-like applicative language (see [Gordon79]) supplemented with certain imperative features to allow sequencing and reference to state. Because it is hard to reason about and manipulate programs the are overly committed with respect to order of computation and data representation, we have sought to keep the programming language as unconstraining as possible. In addition, certain features that are difficult to implement but which have clear semantics are included because they often allow derivations to be quite straightforward. The infinitary definition of *path* above provides an example of such a feature; it has straightforward fixed-point semantics.

This approach, in which commitments to sequencing and representation are delayed as long as possible, is also 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 2.3 below) in order to keep the set of transformations simple and yet strong-equivalence preserving. We will explain the unusual features of the language as they are encountered.

Mechanization. We expect that the program derivation techniques such as those refined and applied here and elsewhere will ultimately be of use in practical mechanical programming aids designed to help the programmer in his daily activity.

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. Indeed, it appears that reasoning by analogy will play a very important role in the automation of these techniques.

Other examples and approaches to program derivation are described in [Clark80], [Barstow80], [Green78], [Manna81], and [Bauer81], among others.

2. Depth-First Search.

We start by deriving a family of simple depth-first search algorithms. These derivations and the algorithms that result will be used, either directly or by analogy, in the later derivations.

The development in the first part of this section is identical for directed and undirected graphs. We therefore carry out the development for directed graphs and consider undirected graphs as a special case.

Let $G = (V, E)$ be a finite directed graph with adjacency list representation—for each $v \in V$, $Adj(v)$ is the set of vertices adjacent to v. For undirected graphs $v \in Adj(u)$ if and only if $u \in Adj(v)$.

Paths. We consider first a simple combinatorial definition of a path in a graph. Let u and v range over vertices.

$$path(u, v) \;\Leftarrow\; \big(\, u = v \quad \textbf{or} \quad (\exists w \in Adj(u))\, path(w, v) \,\big) \tag{2.1}$$

While this definition seems to capture the notion of path, it cannot be interpreted in the usual way—either as a nondeterministic sequential algorithm or as a parallel algorithm—since in certain cases it would have no finite execution paths.

The finite closure transformation. We can, however, distinguish two kinds of infinite execution paths—looping paths and divergent paths. Roughly put, a nonterminating path is a *looping* path if 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 consider) Algorithm 2.1 can exhibit looping, but, because u and v are vertices and the set of vertices is finite, it cannot exhibit divergence.

By framing this as a *finite closure* problem, we can apply transformations that eliminate the looping paths, and hence all non-terminating paths. Suppose a function f over a finite domain is defined recursively

$$f(x) \;\Leftarrow\; h(x) \,\oplus\, \bigoplus_{z \in g(x)} f(z), \tag{2.2}$$

where h and g do not call f and, in addition, $\oplus$ is a semilattice with identity in which infinite sums are defined (that is, the partial order induced by $\oplus$ is a complete partial order). This definition has a natural, but unconventional, fixed-point semantics in which f is always defined. The transformations allow this definition to be replaced by an equivalent (with respect to the unconventional semantics) but always terminating (with respect to the conventional semantics) definition, essentially by replacing all redundant recursive calls to f by the identity of $\oplus$. (A full account of this technique is beyond the scope of this paper. The transformations, which are related to the closed-world database techniques described in [Clark78] and [Reiter78], are sketched in [Scherlis80].)

In the case of *path*, $\oplus$ is disjunction and has identity (i.e., minimal element) **false**. The effect of the transformation is thus to replace all redundant calls to *path* with **false**.

In order to carry out the transformation, however, it is necessary to introduce mechanism to keep track of the sequencing of computation, collecting values of x as recursive evaluations of f proceed. This forces us to introduce notions of *state* and *state change* into the definition. State changes can be made either implicitly (by introducing imperative operations) or explicitly (by adding a new "memo" parameter to f). More concretely, we introduce explicit data structure to mark vertex pairs as they are considered; by examining this data structure, the program can foreclose any potentially looping execution paths.

It is difficult to manipulate programs involving state, however, so it is best to delay this transformation whenever possible. We will, therefore, postpone the improvement to *path* until the next transformation is complete.

The specialization transformation. Nearly all of the program derivation steps in our derivations are *specialization* steps. This simple technique, which is presented in detail and proved correct in [Scherlis80], is described here informally by means of an example involving the *path* definition.

Suppose we desire to collect the vertices v reachable from a given vertex u.

$$\{v \mid \mathit{path}(u,v)\}$$

If *path* is computable, then the value of this set can be calculated simply by enumerating all vertices v and testing $\mathit{path}(u,v)$ for each. This method is inefficient, however, since it requires multiple traversals of the same graph. We therefore consider *specializing* the definition of *path* to the computational context of the set abstraction.

The transformation has three steps. First, both sides of the definition of *path*

$$\mathit{path}(u,v) \;\Leftarrow\; \big(\, u = v \quad \textbf{or} \quad (\exists w \in \mathit{Adj}(u))\, \mathit{path}(w,v)\,\big)$$

are *substituted* into the set expression, forming the definition,

$$\{v \mid \mathit{path}(u,v)\} \;\Leftarrow\; \{v \mid \; (u = v) \quad \textbf{or} \quad (\exists w \in \mathit{Adj}(u))\, \mathit{path}(w,v)\}\,. \tag{2.3}$$

This definition, called an *expression procedure*, is easily given meaning within the framework of a nondeterministic text-substitution evaluator model; roughly, it denotes a procedure for computing values of instances of its left-hand side.

The second step of the transformation is to *simplify* the right-hand side of the new definition until an instance of the left-hand side appears there. This is accomplished by distributing the set abstraction inward and simplifying.

$$\{v \mid \mathit{path}(u,v)\} \;\Leftarrow\; \{u\} \;\cup\; \textstyle\bigcup_{w \in \mathit{Adj}(u)} \{v \mid \mathit{path}(w,v)\} \tag{2.4}$$

Observe now that this definition is recursive, and that it makes exactly one recursive call for each w, rather than one (to *path*) for each w and v pair, as in the earlier version.

The third and final step of the transformation is to *rename* all instances of the set expression to a new function name with appropriate parameters. (This has the effect of pruning the tree of nondeterministic computation paths.) The only free variable in the expression is u, so we obtain

$$\mathit{dfs}(u) \;\Leftarrow\; \{u\} \;\cup\; \textstyle\bigcup_{w \in \mathit{Adj}(u)} \mathit{dfs}(w)\,. \tag{2.5}$$

(The choice of the name "*dfs*" will be justified shortly.) Now, since

$$path(u,v) = v \in \{v \mid path(u,v)\}\,, \tag{2.6}$$

we obtain

$$\begin{aligned} path(u,v) &\Leftarrow v \in dfs(u) \\ dfs(u) &\Leftarrow \{u\} \cup \textstyle\bigcup_{w \in Adj(u)} dfs(w)\,. \end{aligned} \tag{2.7}$$

This new definition of *path* provides a performance advantage over the original definition of *path* if, in a series of computations, u changes infrequently compared with v and $dfs(u)$ is precomputed.

Finite closure revisited. At this point we can make the finite closure transformation that was postponed earlier. Like disjunction, the accumulator function '$\cup$' has the necessary algebraic properties. We thus replace redundant calls to *dfs* by $\emptyset$, which is the identity of union.

```
path(u, v)  ⇐  begin
                   visit[V] ← false;
                   v ∈ dfs(u)
               end
                                                                    (2.8)
dfs(u)  ⇐  begin
               visit[u] ← true;
               {u} ∪ ⋃_{w∈Adj(u)} (if visit[w] then ∅ else dfs(w))
           end
```

(In general, the value of a block is the value of the last expression unless some other expression is marked by the word **value**. In that case, the value of that expression is saved when it is evaluated, and the saved value is returned after evaluation of the remainder of the block is complete; see, for example, Algorithm 2.19. By convention imperative statements are always enclosed in blocks.)

This program requires some explanation. We have made use of implicit state (i.e., imperative operations on global data structure) to keep the "memo" set, representing it in its characteristic-function form by the array *visit*. The definition of *path* has been modified to initialize the memo set by storing **false** in every element of the array; this indicates that initially no vertices have been visited. (We use the word "memo" to draw analogy with the less powerful—since it has no effect on termination—'memo-function' transformation suggested by Donald Michie.)

In spite of the dependence of intermediate values of *visit* on the choice of computation ordering (which is only partially committed above), it is a property of the finite closure method that the ultimate value of *visit* (and, of course, *dfs*) is independent of the order of evaluation of both the binary union and the quantified union. Sequential evaluation of the outer union results in the natural depth-first search ordering.

As noted above, the same effect could be achieved using a purely applicative program. The imperative program has the advantage, however, of using a notation that avoids commitment to a particular order of computing the unions, and thus is more clear for our purposes.

This transformation step and the prior specialization step commute, but, because of the explicit sequencing of computation, it is more difficult to carry out the specialization once state has been introduced by finite closure.

Connected components of undirected graphs. We can now derive a linear-time program for collecting the connected components of an undirected graph.

$$\mathit{comps} \;\Leftarrow\; \bigcup_{r\in V} \{\{v \mid \mathit{path}(r,v)\}\} \tag{2.9}$$

(This union of singletons can, of course, also be notated using set abstraction, but the result is less perspicuous.

$$\{\{v \mid \mathit{path}(r,v)\} \mid r \in V\}$$

Tarski's "big-E" notation provides a more succinct, but less widely-known notation for the set.

$$\mathbf{E}_{r\in V}\; \{v \mid \mathit{path}(r,v)\} \quad)$$

Substitution of the improved definition of *path* above and simplification yield

$$\mathit{comps} \;\Leftarrow\; \bigcup_{r\in V} \{\,\textbf{begin}\ \mathit{visit}[V] \leftarrow \textbf{false};\ \ \mathit{dfs}(r)\ \textbf{end}\,\}\,. \tag{2.10}$$

Many redundant searches are performed, so this definition is not optimal; indeed, its worst case running time is $O(|V|^2)$.

We observe, however, that redundant searches can be avoided by making use of the *visit* array used by *dfs*. Using the specialization technique, we obtain the linear-time program

```
comps(V)  ⇐  begin
                 visit[V] ← false;                                        (2.11)
                 $\bigcup_{r\in V}$(if visit[r] then $\emptyset$ else {dfs(r)})
             end .
```

The edges traversed by this program form a depth-first search forest whose roots are the values of r for which *dfs* is called in the definition of *comps*. It is easy to see that this algorithm runs in time linear in the number of vertices and edges in the graph. This is shown by associating with each vertex and edge of the graph a constant number of program steps. As before, the sequencing of the union affects intermediate states but not the final result, so we need not commit ourselves to an order of consideration of the elements of V.

Trees and tree traversals. The fast depth-first search algorithms rely on subtle combinatorial properties of the depth-first spanning forests implicit in the prior algorithms. We now derive some simple algorithms for trees that will be useful in the later development.

The depth-first search algorithms we derive make extensive use of "non-local" properties of depth-first search trees they induce. 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. Both of these orderings relate vertices that may be an arbitrary distance apart in the trees. We make derivation steps here that will enable these relations to be computed efficiently.

A *tree* is a directed graph all of whose vertices have indegree one except the *root* vertex, which has indegree zero. A vertex with zero outdegree is called a *leaf*; the others are *internal nodes*.

The set of vertices of a tree can be enumerated without repetitions by traversing the edges of the tree and recursively enumerating subtrees.

```
trav(u)  ⇐  begin
                examine(u)  //  forpar w suchthat $u \to w$ do trav(w)          (2.12)
            end
```

(The symbol '//' indicates parallel execution, which we use (in terminating programs) to indicate explicit avoidance of commitment to computation ordering. Similarly, the notation '**forpar**' indicates parallel (or unordered sequential) execution of all the specified instances of the loop body. In general, explicit sequencing (with ';') will be avoided whenever possible. Finally, the notation '$u \rightarrow w$' is shorthand for '$\langle u, w\rangle \in E$', where E is the set of edges in the tree. Note that this program is executed for the side effect of calling *examine*; it has no value.)

If r is the root of the tree T, then *trav*(r) will cause *examine* to be called exactly once for each vertex of T.

Preorder and postorder enumeration are obtained by making differing commitments to computation sequencing in the definition above. Preorder enumeration results, for example, when the instance of '//' is replaced by ';'. For ordered trees, the loop cases must also be evaluated sequentially.

```
trav(u)  ⇐  begin
                examine(u);                                        (2.13)
                for w suchthat u → w do trav(w)
            end
```

In the case of binary trees (i.e., all vertices have outdegree either zero or two), inorder can also be easily obtained. In this case, we would need to introduce case analysis on u to determine its outdegree.

Relative preorder position can be tested using an instance of Algorithm 2.13, but more efficient programs can be derived. Both preorder and postorder are (finite) linear orderings, and so can be represented by sequences of vertices. With this representation, two vertices can be compared in pre- or postorder simply by examining their relative positions in the appropriate sequence.

A sequence can be represented as an array mapping vertices to integers representing their positions. Let r be the root of a tree.

```
begin p ← 0;  trav(r);  pre[V] end;

trav(u)  ⇐  begin                                                  (2.14)
                pre[u] ← p ← p + 1;
                for w suchthat u → w do trav(w)
            end
```

(The first block is a specification of the computation to be performed.) The result of this program is now an array containing the preorder numbers assigned to the vertices of the tree rooted at r. For brevity, we have omitted the intermediate derivation steps by which this imperative algorithm is obtained.

A similar algorithm can be derived for computing the postorder numbering. By merging Algorithm 2.14 with this new algorithm, we obtain

```
begin p ← 0;  e ← 0;  trav(r);  ⟨pre[V], post[V]⟩ end;

trav(u)  ⇐  begin
                pre[u] ← p ← p + 1;                                (2.15)
                for w suchthat u → w do trav(w);
                post[u] ← e ← e + 1
            end .
```

(Again, we omit transformation steps. A detailed example of the merging technique, which is just a special case of the specialization transformation, is presented in a later section.)

Tree orderings. The *descendent* ordering $\succ$ is the transitive closure of the ordering represented by the edges of a tree. That is,

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

It is undesirable to compute descendency (or ancestry) using a naive implementation of transitive closure, since that would require $O(|V|^3)$ time. We therefore investigate whether we can take advantage of the special properties of trees.

LEMMA 2.1. Let T be a tree with vertices numbered in preorder in array $pre[V]$ and in postorder in array $post[V]$. Then

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

That is, u is a proper descendent of v if and only if both u succeeds v in a preorder traversal and u precedes v in a postorder traversal.

This lemma justifies replacing tests in programs of the form $u \succ v$ by tests of the form

$$pre[u] > pre[v] \qquad \wedge \qquad post[u] < post[v]\,,$$

As shown in the previous section, both numberings can be computed in linear time and in a single tree traversal, so we can now test ancestry in constant time with linear-time precomputation.

Furthermore, u is to the left of v in T if and only if u precedes v in both preorder and postorder. Thus, the relative position of two arbitrary tree vertices can be determined by checking their relative positions in the two orderings.

Depth-first search trees. The depth-first search algorithms on graphs derived earlier impose a natural tree structure on the edges of the graph being searched. That is, the subset of the edges actually traversed forms a forest.

We indicate such facts in our programs by writing *assertions*, which are expressions enclosed in the special brackets '$[\![\ ]\!]$' and located at points in the program where the facts are true. (This notation is also used to denote preconditions. See, for example, the derivation of Algorithm 2.21.) For example, we can annotate Algorithm 2.8 to obtain

```
path(u, v)  ⇐  begin visit[V] ← false;  v ∈ dfs(u) end

dfs(u)  ⇐  begin
              visit[u] ← true;                                                (2.16)
              {u}  ∪  ⋃_{w∈Adj(u)}(if visit[w] then ∅ else [[u → w]] dfs(w))
           end
```

In the **else** clause we have asserted that $\langle u, w\rangle \in E$ is a tree edge. This set of tree edges forms the depth-first search forest.

We are now ready to develop an algorithm for carrying out a preorder traversal of a depth-first search tree of a graph. This will be accomplished by deriving a program that simultaneously computes *dfs* and *trav*. We indicate the simultaneous computation by writing a block,

begin *trav*(u); $\langle dfs(u), pre[V]\rangle$ **end**.

Recall that *trav* (see Algorithm 2.14) returns a value—the *pre* array—only implicitly. Thus, the effect of the computation of this block will be to store values into the *pre* array and to return the set of nodes reachable from u. For simplicity, we assume for the moment that the graph is connected.

Let r be a vertex of G. By substituting both definitions into the block and making obvious simplifications, we obtain

```
begin  $p \leftarrow 0$ // $visit[V] \leftarrow$ false;  (begin $trav(r)$;  $\langle dfs(r), pre[V]\rangle$ end)  end

(begin $trav(u)$; $dfs(u)$ end)  $\Leftarrow$
     begin
       $visit[u] \leftarrow$ true  //  $pre[u] \leftarrow p \leftarrow p+1$;
       $\{u\}$  $\cup$  (begin
                         for $w$ suchthat $u \to w$ do $trav(w)$;                    (2.17)
                         $\left(\bigcup_{w \in Adj(u),\ \neg visit[w]} dfs(w)\right)$
                     end
     end
```

(We have carried out some trivial transformations on the initial specification in order to bring it into the form of the definition.) Now $u \to w$ if and only if $w \in Adj(u) \wedge \neg visit(w)$, so the two loops range over the same set. Since they do not interact, this implies that they can be *merged*—the pair of iterations can be combined into a single iteration.

At this point, we can make two further simplifications. First, we *rename* the block being defined to the simple name *dfs* (superseding the previous use of this name), and, second, we observe that if $pre[V]$ is initialized to 0, then

$$visit[u] = \textbf{false} \quad \text{if and only if} \quad pre[u] = 0\,,$$

and we can eliminate the *visit* array and use *pre* instead. The following much shorter program results.

```
begin $p \leftarrow 0$ // $pre[V] \leftarrow 0$;  $S \leftarrow dfs(r)$;  $\langle S, pre[V]\rangle$ end

$dfs(u)$  $\Leftarrow$
     begin
       $pre[u] \leftarrow p \leftarrow p+1$;                                          (2.18)
       $\{u\}$  $\cup$  $\bigcup_{w \in Adj(u),\ pre[w]=0} dfs(w)$
     end
```

The ordering represented by *pre* is called a ***depth-first-search ordering*** of the vertices of the graph.

By a development similar to that for *pre* and a merge step similar to the one just completed, the *post* ordering can be computed as well.

```
begin $p \leftarrow 0$ // $e \leftarrow 0$ // $pre[V] \leftarrow 0$;  $S \leftarrow dfs(r)$;  $\langle S, pre[V], post[V]\rangle$ end

$dfs(u)$  $\Leftarrow$
     begin
       $pre[u] \leftarrow p \leftarrow p+1$;                                          (2.19)
       value $\{u\}$  $\cup$  $\left(\bigcup_{w \in Adj(u),\ pre[w]=0} dfs(w)\right)$;
       $post[u] \leftarrow e \leftarrow e+1$
     end
```

Depth-first search in undirected graphs. We now consider the special case of depth-first search in undirected graphs. In this case, 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***. While the tree edges are directed edges, we leave the fronds undirected (for the moment). We use the notation $u \leftrightarrow v$ to indicate fronds and, as before, $u \rightarrow v$ to indicate tree edges. Thus, every edge $\langle u, v\rangle$ is either a tree edge, a reverse tree edge, or a frond.

We will occasionally need to distinguish the fronds explicitly during search. With respect to Algorithm 2.8, we observe that the fronds are exactly those edges $\langle u, w\rangle$ 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]                                              (2.20)
                            then (if w ≠ father(u) then ⟦u ↔ w⟧ ) ∅
                            else ⟦u → w ∧ u = father(w)⟧  dfs(w))
    end
```

Here we have decorated Algorithm 2.8 with assertions distinguishing the two sets of edges. The *father* function can be considered to be defined implicitly by the assertion. (In the case of a root, *father* can return a special value, say Λ, that will cause the test to fail.)

Observe now that the father of u is known whenever *dfs* is called recursively. Using the specialization technique, 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 being generated. We do this by forming an expression procedure for

$$[\![v = \mathit{father}(u)]\!]\ \mathit{dfs}(u)\,.$$

(In an expression procedure name an assertion denotes a precondition.) We obtain

```
⟦v = father(u)⟧ dfs(u)  ⇐
    begin
      visit[u] ← true;
      {u} ∪ ⋃_{w∈Adj(u)}(if ¬visit[w] then ⟦u → w ∧ u = father(w)⟧ dfs(w)       (2.21)
                            elseif w ≠ father(u) then ⟦u ↔ w⟧ ∅
                            else ⟦w → u⟧ ∅)
    end.
```

(For aesthetic reasons we have also reoriented the nested conditionals.) This is simplified by replacing *father*(u) in the test by v. After renaming (again superseding the name *dfs*), we have the definition

```
dfs(u, v)  ⇐
    begin
      visit[u] ← true;
      {u} ∪ ⋃_{w∈Adj(u)}(if ¬visit[w] then ⟦u → w⟧ dfs(w, u)                    (2.22)
                            elseif w ≠ v then ⟦u ↔ w⟧ ∅
                            else ⟦w → u⟧ ∅)
    end.
```

Finally, we carry through the transformation steps described earlier to obtain an algorithm similar to Algorithm 2.19.

```
begin $p \leftarrow 0$ // $e \leftarrow 0$ // $pre[V] \leftarrow 0$; $S \leftarrow dfs(r, \Lambda)$;  $\langle S, pre[V], post[V] \rangle$ end

$dfs(u, v) \Leftarrow$
    begin
      $pre[u] \leftarrow p \leftarrow p + 1$;
      value $\{u\} \cup \bigcup_{w \in Adj(u)}$(if $pre[w] = 0$ then $[\![u \rightarrow w]\!]$ $dfs(w, u)$                (2.23)
                                  elseif $w \neq v$ then $[\![u \nleftrightarrow w]\!]$ $\emptyset$
                                  else $[\![w \rightarrow u]\!]$ $\emptyset$);
      $post[u] \leftarrow e \leftarrow e + 1$
    end
```

(We use Λ to stand for a value not equal to any vertex.)

A further specialization. In the biconnectivity algorithm derivation, we will need to classify fronds into *forward fronds* and *reverse fronds*. Observe that if $u \nleftrightarrow w$ then either u is a descendent of w or vice-versa. If $u \succ w$ then $\langle u, w \rangle$ is a reverse frond, notated $u \dashrightarrow w$; otherwise the edge is a forward frond, and we write $w \nrightarrow u$.

Lemma 2.1 provides a fast method for distinguishing forward and reverse fronds. It is an immediate consequence of the lemma that

$$pre[u] < pre[w] \qquad \text{implies} \qquad u \not\succ w\,,$$

and similarly for *post*. Therefore, if it is known that two vertices are related by the descendency relation, but it is not known in which direction, then it suffices to check *either* the preorder *or* the postorder numberings.

On the basis of this fact, we obtain the following depth-first search algorithm.

```
begin $p \leftarrow 0$ // $pre[V] \leftarrow 0$; $S \leftarrow dfs(r, \Lambda)$;  $\langle S, pre[V] \rangle$ end

$dfs(u, v) \Leftarrow$
    begin
      $pre[u] \leftarrow p \leftarrow p + 1$;
      $\{u\} \cup \bigcup_{w \in Adj(u)}$(if $pre[w] = 0$ then $[\![u \dashrightarrow w]\!]$ $dfs(w, u)$                (2.24)
                            elseif $w = v$ then $[\![w \rightarrow u]\!]$ $\emptyset$
                            elseif $pre[u] > pre[w]$ then $[\![u \dashrightarrow w]\!]$ $\emptyset$
                            else $[\![w \nrightarrow u]\!]$ $\emptyset$ )
    end
```

Observe that the four cases can be distinguished in constant time (given the prior linear-time computation of the *pre* array).

Although this classification of cases clearly does not help solve the immediate problem of collecting reachable vertices, it will be very useful when we use specialization to merge this algorithm with other algorithms obtained in the derivations for biconnectivity in Section 4, below.

3. Strongly-Connected Components.

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

$$\mathit{path}(u,v) \Leftarrow \big(u = v \quad \textbf{or} \quad (\exists w \in \mathit{Adj}(u))\, \mathit{path}(w,v) \big) . \tag{3.1}$$

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 makes sense to consider ***reverse paths*** as well.

$$\mathit{revpath}(u,v) \Leftarrow \big(u = v \quad \textbf{or} \quad (\exists w \in \mathit{Adj}^{-1}(u))\, \mathit{revpath}(w,v) \big) \tag{3.2}$$

(Here, Adj^{-1} denotes the inverse adjacency function defined such that $v \in \mathit{Adj}(u)$ if and only if $u \in \mathit{Adj}^{-1}(v)$.) 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, it suffices to collect all vertices u such that $\mathit{path}(r,u)$ and $\mathit{path}(u,r)$. 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 the strongly-connected components partition the vertices of a directed graph. We therefore take the following definition as our starting specification of the strongly-connected components.

$$\mathit{strong} \Leftarrow \textstyle\bigcup_{r \in V} \{\{s \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\}\} \tag{3.3}$$

If *path* requires time linear in the number of vertices, then this definition, evaluated naively, requires $O(|V|^3)$ time. To simplify the derivation steps, however, we start with the infinitary version of *path* given above.

First steps. Our strategy for improving this definition is to focus on the inner set and develop a method for calculating its value efficiently. A natural first step is to specialize the definition of *path* to the context

$$\{s \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\} ,$$

but the specialization transformation fails because after substitution and simplification the parameters in the pair of recursive calls do not match. A natural response is to generalize, separating either the r or s pairs of parameters into distinct variables. (Generalization is a common heuristic for obtaining inductive proofs and has been incorporated into several automatic systems; [Boyer75] and [Manna79] describe examples.)

$$\begin{aligned} &\{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \\ &\qquad \Leftarrow \{s \mid \big((u = s) \textbf{ or } (\exists w \in \mathit{Adj}(u))\, \mathit{path}(w,s)\big) \wedge \mathit{path}(s,r)\} \end{aligned} \tag{3.4}$$

We have substituted the definition of *path* in the first instance, but not the second. As in the very first specialization example, we simplify by distributing the set abstraction inward.

$$\begin{aligned} &\{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \\ &\qquad \Leftarrow \{s \mid u = s \wedge \mathit{path}(s,r)\} \cup \textstyle\bigcup_{w \in \mathit{Adj}(u)} \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \end{aligned} \tag{3.5}$$

This expression procedure is recursive, so our simplification is partly successful.

Unfortunately, we are forced to test $\mathit{path}(u, r)$ on every iteration. This predicate is certainly true, for example, on the initial call,

$$\mathit{strong} \Leftarrow \bigcup_{r \in V} (\{ [\![\mathit{path}(r,r)]\!] \{s \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\}\}), \tag{3.6}$$

and appears to be true on the others. To test this latter conjecture, we specialize the already specialized definition a bit further, to a context in which $\mathit{path}(u, r)$ is assumed to be true on entry.

$$\begin{aligned} &[\![\mathit{path}(u,r)]\!] \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \Leftarrow \\ &\quad \{s \mid u = s \wedge \mathit{path}(s,r)\} \cup \bigcup_{w \in Adj(u)} \\ &\qquad\qquad \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \end{aligned} \tag{3.7}$$

The assertion can be established, or not, for the recursive case by introducing an obvious case analysis.

$$\begin{aligned} &[\![\mathit{path}(u,r)]\!] \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \Leftarrow \\ &\quad \{u\} \cup \bigcup_{w \in Adj(u)} \\ &\qquad (\textbf{if } \mathit{path}(w,r) \textbf{ then } \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \\ &\qquad\quad \textbf{else } \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\}) \end{aligned} \tag{3.8}$$

The assertion is clearly true in the **then** clause. In the **else** clause, it follows from transitivity of *path* that the set must be empty.

$$\begin{aligned} &[\![\mathit{path}(u,r)]\!] \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \Leftarrow \\ &\quad \{u\} \cup \bigcup_{w \in Adj(u)} \\ &\qquad (\textbf{if } \mathit{path}(w,r) \textbf{ then } [\![\mathit{path}(w,r)]\!] \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \\ &\qquad\quad \textbf{else } \emptyset) \end{aligned} \tag{3.9}$$

The effect of this transformation sequence is now clear; the $\mathit{path}(u, r)$ test has simply been shifted to the caller, and so the conjecture is *not* established. Although this is not a very significant improvement, these definitions will prove easier to manipulate than Algorithm 3.5. Also, note that the definition is still infinitary, but it does have the prerequisite structure for the finite closure transformation. As before, however, we will postpone making that transformation as long as possible.

The final transformation step in the specialization sequence is to rename the expression procedure for

$$[\![\mathit{path}(u,r)]\!] \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\}$$

to $\mathit{sc}(u, r)$.

$$\begin{aligned} &\mathit{strong} \Leftarrow \bigcup_{r \in V} \{\mathit{sc}(r,r)\} \\ &\mathit{sc}(u,r) \Leftarrow \{u\} \cup \bigcup_{w \in Adj(u)} (\textbf{if } \mathit{path}(w,r) \textbf{ then } \mathit{sc}(w,r) \textbf{ else } \emptyset) \end{aligned} \tag{3.10}$$

The reversed algorithm. The key insight in this derivation can now be revealed: We observe 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, if possible, use the orderings defined in Section 2 to test ancestry.

There are two ways we could 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* both 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 3.10 and applying the finite closure transformation (as we did in Algorithms 2.8 and 2.11), we obtain

```
strong  ⇐  begin
               $visit2[V] \leftarrow$ false;
               $\bigcup_{r \in V}$(if $visit2[r]$ then $\emptyset$ else $\{scr(r,r)\}$)
           end
$scr(u,r)$  ⇐                                                          (3.11)
    begin
        $visit2[u] \leftarrow$ true;
        $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$ (if $\neg visit2[w] \wedge path(r,w)$ then $scr(w,r)$)
    end
```

(We have reserved the name *visit* for use in the depth-first search tree precomputation required for testing $path(r,w)$.)

A blind alley. It may appear that we could obtain an acceptable implementation of Algorithm 3.11 by replacing $path(r,w)$ with the test $w \in dfs(r)$ and using specialization to factor the *dfs* calculation out of *scr* into *strong*.

```
strong  ⇐  begin
               $visit2[V] \leftarrow$ false;
               $\bigcup_{r \in V}$(if $visit2[r]$ then $\emptyset$
                          else begin
                                   $visit[V] \leftarrow$ false;
                                   $\{scr'(r,r,dfs(r))\}$
                               end)                                    (3.12)
           end
$scr'(u,r,D)$  ⇐
    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. This algorithm runs in time $O(|V|^2)$.

Strongly-connected component roots. Our ability to use the ancestry test techniques of Section 2 depends on a crucial lemma. This lemma captures most of the non-trivial graph-theoretic knowledge required in the derivation of the strongly-connected components algorithm.

LEMMA 3.1. Let G be a directed graph with a depth-first search forest F that has ancestry ordering $\succ$. For each strongly-connected component S of G there is a unique vertex r called the *root* of S such that $r = \min_{\succ}(S)$.

This lemma has several important consequences.

(1) The roots of the forest F are roots of strongly-connected components.

(2) For each strongly-connected component S and for each $v \in S$ and $w \notin S$ such that $v \to w$, w is the root of a strongly-connected component.

(3) Items (1) and (2) above yield all the strongly-connected component roots.

(4) If r is the root of a strongly-connected component and $\mathit{path}(w, r)$ is true in G, then

$$\mathit{path}(r, w) \qquad \text{if and only if} \qquad w \succeq r\,.$$

Operationally, the lemma suggests that we try to arrange that $\mathit{scr}(r, r)$ be called (in the definition of *strong*) for the strongly-connected component roots and no other vertices. All the strongly-connected components will still be found, and, by the fourth consequence above, if we guarantee that r is a root, then the $\mathit{path}(r, w)$ test can be replaced by the constant-time test $w \succeq r$. (We do this below.)

To find the roots, we must first collect the depth-first-search forest roots, and then, as we find components, locate the remaining roots.

Our first order of business, then, is to construct the depth-first-search forest F. (Recall that this is distinct from the reverse forest constructed by *scr*.)

```
forest ⇐ begin
             $pre[V] \leftarrow 0 \,//\, p \leftarrow 0 \,//\, e \leftarrow 0$;
             for $r \in V$ do (if $pre[r] = 0$ then $dfs(r)$)
          end
$dfs(u)$ ⇐ begin
             $pre[u] \leftarrow p \leftarrow p + 1$;
             for $w \in Adj(u)$ do
                if $pre[w] = 0$ then $dfs(w)$;
             $post[u] \leftarrow e \leftarrow e + 1$
          end                                                    (3.13)
```

The ancestry test. The fourth consequence of Lemma 2.1, as noted above, allows replacement of the *path* test in *scr* by a test of the ancestry relation, under the condition tht r is always the root of a strongly-connected component. The ancestry test, as shown in Section 2, can be accomplished in constant-time by an examination of the *pre* and *post* numberings obtained in

Algorithm 3.13.

```
$strong(R)$ $\Leftarrow$ begin
                    $visit2[V] \leftarrow$ false;
                    $\bigcup_{r \in R}\{scr(r, r)\}$
                end

$scr(u, r)$ $\Leftarrow$                                                          (3.14)
    begin
        $visit2[u] \leftarrow$ true;
        $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$ ( if $\neg visit2[w] \wedge pre[r] > pre[w] \wedge post[r] < post[w]$
                                  then $scr(w, r)$ )
    end
```

We must now focus on the problem of finding the roots required by *strong*.

Finding some roots. The first consequence of the lemma suggests that we collect the roots of the forest F as they are found.

```
$forest$ $\Leftarrow$ begin
                $pre[V] \leftarrow 0 \mathbin{//} p \leftarrow 0 \mathbin{//} e \leftarrow 0$;                              (3.15)
                $\bigcup_{r \in V}$(if $pre[r] = 0$ then begin $dfs(r)$; $\{r\}$ end)
            end
```

Thus, *forest* yields a nonempty subset of the final set of roots.

Finding the remaining roots. By the lemma, the remaining roots can be found by examining strongly-connected components as they are found. If r is a root then $scr(r, r)$ returns returns the vertices of its corresponding strongly-connected component. Let S be a strongly-connected component. The following definition is a direct realization of consequence (3) of the lemma.

```
$update(S)$ $\Leftarrow$                                                          (3.16)
    $\bigcup_{v \in S}$ ($\bigcup_{w \in Adj(v)}$ (if $\neg visit2(w) \wedge v \rightarrow w$ then $\{w\}$ else $\emptyset$) )
```

Given a strongly-connected component S, *update*(S) returns the set of strongly-connected component roots the are direct descendents of vertices in S. This definition can be improved in several ways, for example by replacing the $v \rightarrow w$ test by a comparison of pre- and postorder indices. (See Algorithm 3.20.)

An alternative approach to finding roots (which we do not follow here but which is used in [AHU83]) is to rely on a further consequence of the lemma. It is always the case, after zero or more strongly-connected components have been found, that the unvisited (in the reverse search) vertex with smallest preorder number (in the original forward search) will be a strongly-connected component root. Initially, this is the first vertex visited by *dfs*. A final algorithm could be obtained by deriving a simple program and associated data structure that quickly yields the vertex with minimum preorder number. This would be done by merging a simple minimum-finding program with *scr* to keep track of the minimum preorder index as vertices are visited.

In this presentation, however, we retain the *update* procedure above.

Final improvements. The final algorithm is obtained by merging two programs. The first, Algorithm 3.14 derived earlier, finds strongly-connected components, given a set of roots.

```
$strong(R)$  $\Leftarrow$  begin
                    $visit2[V] \leftarrow$ false;
                    $\bigcup_{r \in R}\{scr(r, r)\}$
               end

$scr(u, r)$  $\Leftarrow$                                                                  (3.17)
    begin
        $visit2[u] \leftarrow$ true;
        $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$ ( if $\neg visit2[w] \wedge pre[r] > pre[w] \wedge post[r] < post[w]$
                                   then $scr(w, r)$ )
    end
```

The second, which follows directly from the lemma, is an infinitary definition of the set of roots.

$$R \Leftarrow forest \cup \bigcup_{r \in R} update(scr(r, r)) \qquad (3.18)$$

(To avoid trouble with the *visit2* array, we assume for the moment that the infinitary *scr* is used.)

We omit here this straightforward merge of *strong* with the definition of R. Its result is the following new definition of *strong*.

```
$strong(forest)$

$strong(R)$  $\Leftarrow$
    if $R = \emptyset$ then $\emptyset$                                              (3.19)
      else let $r$ = choose $R$ in
              let $S = scr(r, r)$ in
                $\{S\} \cup strong(update(S) \cup R - \{r\})$
```

(The **choose** operation picks an arbitrary element of its argument, which must be a set. The **let** construct is used to bind local names.)

The strong connectivity algorithm. We have resolved the strong connectivity algorithm,

$$\mathit{strong}(\mathit{forest}),$$

into two basic phases. First, *forest* is used to collect the depth-first search forest roots and to precompute the pre- and postorder numberings used for testing ancestry. Second, *strong* is used to do reverse depth-first searches from these roots, collecting strongly-connected components and new roots along the way.

```
var $p, e, pre[V], post[V]$;
$strong(forest)$

$forest \Leftarrow$
    begin
        $pre[V] \leftarrow 0 \; // \; p \leftarrow 0 \; // \; e \leftarrow 0$;
        $\bigcup_{r \in V}$ (if $pre[r] = 0$ then begin $dfs(r)$; $\{r\}$ end)
    end

$dfs(u) \Leftarrow$
    begin
        $pre[u] \leftarrow p \leftarrow p + 1$;
        for $w \in Adj(u)$ do
            if $pre[w] = 0$ then $dfs(w)$;
        $post[u] \leftarrow e \leftarrow e + 1$
    end

$strong(R) \Leftarrow$                                                        (3.20)
    if $R = \emptyset$ then $\emptyset$
    else let $r =$ choose $R$ in
            let $S = scr(r, r)$ in   $[\![S$ is a strong component$]\!]$
                $\{S\} \cup strong(update(S) \cup R - \{r\})$

$scr(u, r) \Leftarrow$
    begin
        $visit2[u] \leftarrow$ true;
        $\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$ ( if $\neg visit2[w] \wedge pre[r] > pre[w] \wedge post[r] < post[w]$
                                  then $scr(w, r)$
                                  else $\emptyset$ )
    end

$update(S) \Leftarrow$
    $\bigcup_{v \in S}$ ($\bigcup_{w \in Adj(v)}$ (if $\neg visit2(w) \wedge pre[v] > pre[w] \wedge post[v] < post[w]$
                          then $\{w\}$
                          else $\emptyset$) )
```

This algorithm runs in time linear in the number of vertices and edges in the graph. As before, we demonstrate this by associating with each vertex and edge of the graph a constant number of program steps.

The procedure *forest* is a simple iteration in which each vertex r in V is considered exactly once. Because of the *pre* array, *dfs* is called (by *forest* and recursively) at most once for each vertex in the graph. On each call to *dfs*, the loop body is executed once for each edge leaving the node u. Thus, overall, *dfs* visits each vertex once and each edge twice.

A similar argument applies to *strong* and *scr*. This leaves *update*, which is called exactly once for each strongly-connected component. In a given call to *update*, each vertex in the component is examined once in the outer union, and each edge connected to that vertex is examined exactly twice (overall) in the inner union. Thus, overall, *update* examines each vertex once and each edge twice.

Further steps. 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.

4. 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 specification for the biconnected components of a graph makes use of a modified version of the original *path* definition. Let u, v, and a be vertices in an undirected graph.

$$\begin{aligned} &\mathit{path}_a(u,v) \Leftarrow \\ &\quad u = v \ \ \textbf{or} \ \ (\exists w \in \mathit{Adj}(u))\,(w = v \ \ \textbf{or} \ \ (w \neq a \ \textbf{and} \ \mathit{path}_a(w,v))) \end{aligned} \tag{4.1}$$

There is a path from u to v that ***avoids*** a if u and v are equal or if there is path avoiding a from a vertex w adjacent to u to v. (The subscripting of the parameter a is for syntactic convenience only.) Observe that

$$\mathit{path}_a(u,v) \qquad \text{if and only if} \qquad \mathit{path}_a(v,u)\,.$$

There is a natural special case of this definition, obtained by an obvious specialization step.

$$\begin{aligned} &[\![u \neq a \wedge v \neq a]\!]\, \mathit{path}_a(u,v) \Leftarrow \\ &\quad u = v \ \ \textbf{or} \ \ (\exists w \in \mathit{Adj}(u))\,((w \neq a \ \textbf{and} \ [\![w \neq a \wedge v \neq a]\!]\, \mathit{path}_a(w,v))) \end{aligned} \tag{4.2}$$

Two adjacent edges $\langle u,v\rangle$ and $\langle v,w\rangle$ are ***biconnected*** if $\mathit{path}_v(u,w)$. Thus, the biconnected component associated with a graph edge $\langle u,v\rangle$ is a set of edges,

$$\mathit{bc}(\langle u,v\rangle) \Leftarrow \{\langle u,v\rangle\} \cup \left(\bigcup_{\substack{\langle v,w\rangle \in E \\ \mathit{path}_v(u,w)}} \mathit{bc}(\langle v,w\rangle) \right). \tag{4.3}$$

(The specialized definition of *path* will suffice in this context.)

Finally, Let $G = (V, E)$ be an undirected graph with no self-loops (edges of the form $\langle u,u\rangle$). Then the set *bcomps* contains the biconnected components of G.

$$\mathit{bcomps} \Leftarrow \bigcup_{\langle u,v\rangle \in E} \{\mathit{bc}(\langle u,v\rangle)\} \tag{4.4}$$

Specialization to depth-first search. The essence of the biconnectivity algorithm is in the definition of *bc*. Our initial goal will be to obtain a finitary—and efficient—version of this definition. We will not apply the finite closure transformation directly, as this would cause us to have to mark *edges* as being visited. Rather, we will assume that a depth-first search forest *already exists*, and *specialize* the definition of *bc* to traverse tree edges in depth-first search order. That is, we will merge the *bc* definition with a simple depth-first search traversal of the graph.

Note that this merge must actually incorporate a finite closure transformation, as we will be changing the termination properties of *bc*. Rather than carrying this out formally (which would involve going into the technical details of the finite closure transformation method), we will make informal arguments concerning the order of depth-first search traversal. The specialization process will be more difficult than in previous examples because we will need to make use of auxiliary lemmas concerning graphs and trees.

We start by assuming the edge given to *bc* is a tree edge and that previous tree edges have already been traversed in depth-first search order. Our approach will be to consider a variety of cases for the body of the union, depending on the type of the edge $\langle v, w\rangle$. Recall from Section 2 that an undirected graph edge is either a tree edge, a reverse tree edge, a forward frond, or a reverse frond. The first step is to introduce a conditional into the body of the union to distinguish the four cases. Our goal will be to simplify this definition in such a way that *bc* is called recursively for tree edges only *and* that the edges are traversed in a depth-first search order.

$$\begin{array}{l} [\![u \rightarrow v]\!]\ bc(\langle u,v\rangle)\ \Leftarrow \\ \quad \{\langle u,v\rangle\}\ \cup\ \bigcup_{\langle v,w\rangle\in E}\big(\,\textbf{if}\ v \rightarrow w\ \textbf{then}\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(\langle v,w\rangle)\ \textbf{else}\ \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif}\ u = w\ \textbf{then}\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(\langle v,w\rangle)\ \textbf{else}\ \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif}\ v \dashrightarrow w\ \textbf{then}\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(\langle v,w\rangle)\ \textbf{else}\ \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{else}\ [\![w \dashrightarrow v]\!]\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(\langle v,w\rangle)\ \textbf{else}\ \emptyset)\,\big) \end{array} \tag{4.5}$$

We have distributed the *path* test into the four cases. We now consider each of the cases individually.

Suppose $u = w$; that is, w is the father of v. In this case $path_v(u,w)$ is trivially true, so we must compute $bc(\langle v,w\rangle)$. But $\langle v,w\rangle = \langle u,v\rangle$, and we are already computing $bc(\langle u,v\rangle)$, so (by our finite closure argument) we replace the new *bc* call by $\emptyset$. To simplify notation, we also apply transformations so *bc* is passed two adjacent vertices, rather than the edge between them.

$$\begin{array}{l} [\![u \rightarrow v]\!]\ bc(u,v)\ \Leftarrow \\ \quad \{\langle u,v\rangle\}\ \cup\ \bigcup_{\langle v,w\rangle\in E}\big(\textbf{if}\ v \rightarrow w\ \textbf{then}\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(v,w)\ \textbf{else}\ \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif}\ u = w\ \textbf{then}\ \emptyset \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif}\ v \dashrightarrow w\ \textbf{then}\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(v,w)\ \textbf{else}\ \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{else}\ [\![w \dashrightarrow v]\!]\ (\textbf{if}\ path_v(u,w)\ \textbf{then}\ bc(v,w)\ \textbf{else}\ \emptyset)\big) \end{array} \tag{4.6}$$

We next consider the case of a reverse frond $v \dashrightarrow w$. In this case, $path_v(u,w)$ is always true since v is a direct descendent of u and w is an ancestor of u. We must therefore include $bc(v,w)$. Now $\langle v,w\rangle$ is not a tree edge, so this recursive call will not be in the specialized form. We therefore expand the definition of *bc* in this context and simplify based on the assumptions. Since w is an ancestor of u and there is an edge adjacent to w already known to be in the same component as $\langle u,v\rangle$, we can (by the finite closure argument and by the assumption of depth-first order of traversal) replace *all* the recursive *bc* calls from w by $\emptyset$ and retain only the single edge $\langle v,w\rangle$. Observe that this implies all reverse fronds from a vertex are collected at that vertex.

The third case, $w \not\twoheadrightarrow v$, reduces to $\emptyset$. In this case $\langle v, w\rangle$ is a forward frond, and there must be a vertex t such that t is an ancestor of w and such that $v \to t$ has already been traversed. Now, if $\langle v, w\rangle$ is in the same component as $\langle u, v\rangle$, then it will have been found already (by the immediately preceeding case and by the assumption of depth-first order of traversal). If not, then the *path* test would fail and the result would be $\emptyset$. Thus, the result is $\emptyset$ for both possible eventualities.

The final case turns out to very easy. If $\langle v, w\rangle$ is a tree edge, then the recursive call to *bc* is already in the specialized form.

We thus obtain the following finitary definition.

$$\begin{array}{l} [\![u \to v]\!]\ bc(u,v) \ \Leftarrow \\ \quad \{\langle u,v\rangle\} \ \cup \ \bigcup_{\langle v,w\rangle \in E} (\textbf{if } v \to w \textbf{ then } (\textbf{if } path_v(u,w) \textbf{ then } [\![v \to w]\!]\ bc(v,w) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \emptyset) \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif } u = w \textbf{ then } \emptyset \\ \qquad\qquad\qquad\qquad\qquad \textbf{elseif } v \not\twoheadrightarrow w \textbf{ then } \{\langle v,w\rangle\} \\ \qquad\qquad\qquad\qquad\qquad \textbf{else } [\![w \not\twoheadrightarrow v]\!]\ \emptyset) \end{array} \tag{4.7}$$

There are two ways in which the biconnected components algorithm can now be improved. First, the *path* test in *bc* could be made more efficient, and, second, the definition of *bcomps* could be improved to avoid collecting redundant components.

Articulation edges. The improvement of the biconnectivity algorithm *bcomps* depends on the following lemma.

LEMMA 4.1. Let G be an undirected graph with depth-first search forest F. Every biconnected component B contains a unique tree edge $u \to v$, called the *articulation edge*, such that u is an ancestor of every vertex in the edges of B.

This lemma has two useful consequences.

(1) Every tree edge leaving the roots of the trees in a depth-first search forest is an articulation edge.

(2) If $u \to v$ and $v \to w$, then

$$path_v(u,w) \qquad \text{if and only if} \qquad \langle v,w\rangle \text{ is not an articulation edge.}$$

An immediate application of the lemma is to the original definition of *bcomps*. Since every biconnected component has a unique articulation edge associated with it, *bcomps* can be modified to call *bc* for articulation edges only. Let *aedges* be the set of articulation edges.

$$bcomps \ \Leftarrow \ \bigcup_{\langle u,v\rangle \in aedges} [\![u \to v]\!]\ \{bc(u,v)\} \tag{4.8}$$

Note that since every articulation edge is a tree edge and since the set of biconnected components is a partition of the set of edges, the specialized version of *bc* can be applied here.

Collecting biconnected components. Since articulation edges are tree edges, we will attempt to collect them in a single depth-first search. We assume, again, that the tree edges are already so classified and, in addition, we assume that *root*(r) is true if r is a root in the depth-first search forest. The algorithm below reduces the problem to testing individual tree edges using a predicate *aedge*.

$$\begin{aligned} \mathit{aedges} \ &\Leftarrow\ \bigcup_{\substack{root(r)\\ r\rightarrow s}} \big(\{\langle r,s\rangle\} \cup \mathit{ae}(r,s) \big) \\ [\![u \rightarrow v]\!]\ \mathit{ae}(u,v) \ &\Leftarrow\ \bigcup_{\substack{w\in Adj(v)\\ v\rightarrow w}} \big(\textbf{if } \mathit{aedge}(v,w) \textbf{ then } \{\langle v,w\rangle\} \cup \mathit{ae}(v,w) \\ &\qquad\qquad\qquad \textbf{else } \mathit{ae}(v,w) \big) \end{aligned} \tag{4.9}$$

The second consequence of the lemma enables replacement of the test '$\mathit{aedge}(v,w)$' by the test '$\neg path_v(u,w)$.'

It is now a natural step to merge this search for articulation edges with the algorithm *bcomps* for collecting the edges of individual components. The following algorithm results after an obvious specialization step.

$$\begin{aligned} \mathit{bcomps} \ &\Leftarrow\ \bigcup_{\substack{root(r)\\ r\rightarrow s}} \big(\{\mathit{bc}(r,s)\} \cup \mathit{ae}(r,s) \big) \\ [\![u \rightarrow v]\!]\ \mathit{ae}(u,v) \ &\Leftarrow\ \bigcup_{\substack{w\in Adj(v)\\ v\rightarrow w}} \big(\textbf{if } \mathit{aedge}(v,w) \textbf{ then } \{\mathit{bc}(v,w)\} \cup \mathit{ae}(v,w) \\ &\qquad\qquad\qquad \textbf{else } \mathit{ae}(v,w) \big) \end{aligned} \tag{4.10}$$

The function *ae* now returns a set of biconnected components.

It is clear from the structure of this algorithm that it would be advantageous to merge the computations of *ae* and *bc*. We do this by developing an expression procedure for the pair

$$[\![u \rightarrow v]\!]\ \langle \mathit{bc}(u,v),\ \mathit{ae}(u,v)\rangle\ .$$

The result of this program will be a pair of sets. The first is the set of edges of the current component accumulated thus far; the second is the set of components accumulated thus far. After substitution and simplification, we obtain

```
$bcomps \Leftarrow \bigcup_{root(r),\ r\rightarrow s} \big( \{B\} \cup A$  where $\langle B,A\rangle = \langle bc(r,s), ae(r,s)\rangle \big)$

$[\![u \rightarrow v]\!]\ \langle bc(u,v), ae(u,v)\rangle \Leftarrow$
    $\langle \{\langle u,v\rangle\} \cup B, A\rangle$
      where
        $\langle B,A\rangle = \langle \bigcup, \bigcup\rangle_{w\in Adj(v)}$
                (if $v \rightarrow w$ then ( if $\neg path_v(u,w)$
                                          then $[\![aedge(v,w)]\!]\ \langle \emptyset, \{B'\} \cup A'\rangle$
                                          else $\langle B',A'\rangle$ )
                                        where $\langle B',A'\rangle = [\![v \rightarrow w]\!]\ \langle bc(v,w), ae(v,w)\rangle$
                  elseif $u = w$ then $\langle \emptyset,\emptyset\rangle$
                  elseif $v \not\rightarrow w$ then $\langle \{\langle v,w\rangle\},\emptyset\rangle$
                  else $\langle \emptyset,\emptyset\rangle$) .
```
(4.11)

(We have, in this example, introduced a new notation for the simultaneous accumulation of sets. Suppose the function f returns a pair of sets. Then the notation

$$\langle \bigcup, \bigcup\rangle_{w\in S}\ (f(w))$$

describes a pair of sets and yields the same result as

$$\big\langle \bigcup_{w\in S} (\mathit{first}[f(w)]),\ \bigcup_{w\in S} (\mathit{second}[f(w)]) \big\rangle\ ,$$

where *first* and *second* select the corresponding elements of a pair.)

We complete the specialization step by renaming the pair to a simple name, *ba*.

```
$bcomps \;\Leftarrow\; \bigcup_{\substack{root(r)\\ r\rightarrow s}} (\{B\} \cup A$   where $\langle B, A\rangle = ba(r,s))$

$ba(u,v) \;\Leftarrow$
    $\langle \{\langle u,v\rangle\} \cup B, A\rangle$
        where $\langle B, A\rangle = \langle \bigcup, \bigcup\rangle_{w\in Adj(v)}$
                    ( if $v \rightarrow w$
                          then ((if $\neg path_v(u,w)$
                                     then $[\![aedge(v,w)]\!]\ \langle \emptyset, \{B'\} \cup A'\rangle$
                                     else $\langle B', A'\rangle$ )
                                 where $\langle B', A'\rangle = ba(v,w))$                (4.12)
                      elseif $u = w$ then $\langle \emptyset, \emptyset\rangle$
                      elseif $v \dashrightarrow w$ then $\langle \{\langle v,w\rangle\}, \emptyset\rangle$
                      else $\langle \emptyset, \emptyset\rangle$ ) .
```

It now remains to derive a method for efficiently testing $\neg path_v(u,w)$.

Finding articulation edges. In order to implement the *path* test efficiently, we need a second technical lemma.

LEMMA 4.2. Let G be an undirected graph with depth-first search forest F and let $u \rightarrow v$ and $v \rightarrow w$ be edges in F. Then

$$\begin{aligned} path_v(u,w) &\equiv (\exists s,t)\,(u \succeq t \wedge t \dashleftarrow\!\!\!\dashrightarrow s \wedge s \succeq w) \\ &\equiv (\exists s,t)\,(v \succ t \wedge t \dashleftarrow\!\!\!\dashrightarrow s \wedge s \succeq w)\,. \end{aligned}$$

That is, there is a path from u to w avoiding v exactly when there is a frond extending from a descendent s of w to a proper ancestor t of v.

Our goal is to compute this test efficiently in the course of a single depth-first search. The key insight at this point is to represent the *set* of possible values of t such that $t \dashleftarrow\!\!\!\dashrightarrow s$ and $s \succeq w$ by a single value—the most remote ancestor found thus far. If this ancestor turns out to be a proper ancestor of v, then there is indeed a path avoiding v from u (the father of v) to w (a son of v).

In other words, we seek to compute something like

$$low(w) \;\Leftarrow\; \min_{\succ}(\{t \mid (\exists s)\, s \dashrightarrow t \;\wedge\; s \succeq w\})\ .$$

Unfortunately, because the elements of the set are not always pairwise comparable, this minimum is not well defined. It is the case, however, that each element of the set is either an ancestor or a descendent of w. Furthermore, all ancestors of w are themselves pairwise comparable. Since v is an ancestor of w and since we are only interested in t that are proper ancestors of v, descendents of w can be ignored during search. We implement this improvement by means of a simple modification to the above specification. This modification is easily seen to follow from the lemma.

$$low(w) \;\Leftarrow\; \min_{\succ}(\,\{w\} \;\cup\; \{t \mid (\exists s)\, s \dashrightarrow t \;\wedge\; s \succeq w\}\,) \tag{4.13}$$

Now, $v \succ \mathit{low}(w)$ if and only if $\mathit{path}_v(u, w)$. In other words, $v \rightarrow w$ is an articulation edge if and only if $\mathit{low}(w) \succeq v$.

In order to develop a depth-first search algorithm for computing *low*, we separate the computation into two stages.

$$\begin{aligned} \mathit{low}(w) &\Leftarrow \min_{\succ}(\{w\} \cup \mathit{lowset}(w)) \\ \mathit{lowset}(w) &\Leftarrow \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \succeq w\} \end{aligned} \tag{4.14}$$

Lowset computation. We observe first that $\{s \mid s \succeq w\}$ is exactly $\mathit{dfs}(w)$. We recall the definition of *dfs* from Section 2.

```
begin $p \leftarrow 0$ // $\mathit{pre}[V] \leftarrow 0$; $S \leftarrow \mathit{dfs}(r, \Lambda)$; $\langle S, \mathit{pre}[V]\rangle$ end

$[\![u \rightarrow v]\!]\ \mathit{dfs}(u, v) \Leftarrow$
    begin
      $\mathit{pre}[v] \leftarrow p \leftarrow p + 1$;
      $\{v\} \cup \bigcup_{w \in \mathit{Adj}(v)}($if $\mathit{pre}[w] = 0$ then $[\![v \rightarrow w]\!]\ \mathit{dfs}(v, w)$
                                  elseif $w = u$ then $[\![w \rightarrow v]\!]\ \emptyset$
                                  elseif $\mathit{pre}[v] > \mathit{pre}[w]$ then $[\![v \nrightarrow w]\!]\ \emptyset$
                                  else $[\![w \nrightarrow v]\!]\ \emptyset\,)$
    end                                                                    (4.15)
```

(In order to maintain consistent notation in this section, we are using a slightly different vertex labeling convention that of Section 2.)

Since *dfs* requires a father parameter, we revise slightly our definition of *lowset*.

$$[\![u \rightarrow v]\!]\ \mathit{lowset}(u, v) \Leftarrow \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \mathit{dfs}(u, v)\} \tag{4.16}$$

As before, we assume $u \rightarrow v$. We also assume that a special value Λ is passed for u when v is a root. (We are renaming parameters to be consistent with their subsequent usage.)

Direct substitution for *dfs* in the definition of *lowset* and preliminary simplification yield the expression procedure,

```
$[\![u \rightarrow v]\!]\ \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \mathit{dfs}(u, v)\} \Leftarrow$
    begin
      $\mathit{pre}[v] \leftarrow p \leftarrow p + 1$;
      $\{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \{v\}\}$
          $\cup\ \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \bigcup_{w \in \mathit{Adj}(v)}$         (4.17)
                              $($if $\mathit{pre}[w] = 0$ then $[\![v \rightarrow w]\!]\ \mathit{dfs}(v, w)$
                                  elseif $w = u$ then $[\![w \rightarrow v]\!]\ \emptyset$
                                  elseif $\mathit{pre}[v] > \mathit{pre}[w]$ then $[\![v \nrightarrow w]\!]\ \emptyset$
                                  else $[\![w \nrightarrow v]\!]\ \emptyset\,)\}$
    end .
```

We now distribute the set abstraction into the union and conditional and simplify.

```
$[\![u \rightarrow v]\!]\ \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \mathit{dfs}(u, v)\} \Leftarrow$
    begin
      $\mathit{pre}[v] \leftarrow p \leftarrow p + 1$;
      $\{t \mid v \nrightarrow t\}$
          $\cup\ \bigcup_{w \in \mathit{Adj}(v)}($if $\mathit{pre}[w] = 0$                                   (4.18)
                    then $[\![v \rightarrow w]\!]\ \{t \mid (\exists s)\ s \nrightarrow t \ \wedge\ s \in \mathit{dfs}(v, w)\}$
                  elseif $w = u$ then $[\![w \rightarrow v]\!]\ \emptyset$
                  elseif $\mathit{pre}[v] > \mathit{pre}[w]$ then $[\![v \nrightarrow w]\!]\ \emptyset$
                  else $[\![w \nrightarrow v]\!]\ \emptyset\,)\}$
    end .
```

Finally, we can form a recursion. As before, we do this by renaming all instances of the set abstraction to a simple name.

```
lowset(u, v)  ⇐
    begin
      $pre[v] \leftarrow p \leftarrow p + 1$;
      $\{t \mid v \dashrightarrow t\}$
        $\cup\ \bigcup_{w \in Adj(v)}($if $pre[w] = 0$ then $[\![v \rightarrow w]\!]$ lowset$(v, w)$                    (4.19)
                                  elseif $w = u$ then $[\![w \rightarrow v]\!]\ \emptyset$
                                  elseif $pre[v] > pre[w]$ then $[\![v \dashrightarrow w]\!]\ \emptyset$
                                  else $[\![w \dashrightarrow v]\!]\ \emptyset\,)\}$
    end
```

Now since $\{t \mid v \dashrightarrow t\}$ is equivalent to

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

we substitute this into Algorithm 4.19, merge the unions, and simplify on the basis of the assertions to obtain the final *lowset* program.

```
lowset(u, v)  ⇐
    begin
      $pre[v] \leftarrow p \leftarrow p + 1$;
      $\bigcup_{w \in Adj(v)}($if $pre[w] = 0$ then $[\![v \rightarrow w]\!]$ lowset$(v, w)$
                        elseif $w = u$ then $[\![w \rightarrow v]\!]\ \emptyset$                              (4.20)
                        elseif $pre[v] > pre[w]$ then $[\![v \dashrightarrow w]\!]\ \{w\}$
                        else $[\![w \dashrightarrow v]\!]\ \emptyset\,)\}$
    end
```

Low computation. A similar specialization sequence is now used to transform this algorithm into a program for $low(u, v)$, defined

$$[\![u \rightarrow v]\!]\ low(u,v) \ \Leftarrow\ \min_{\succ}(\{v\} \cup lowset(u,v)).$$

We obtain

```
$low(u, v)$ $\Leftarrow$
    begin
      $pre[v] \leftarrow p \leftarrow p + 1;$
      $\min_{w \in Adj(v)}($ $\min(v,$ (if $pre[w] = 0$ then $[\![v \rightarrow w]\!]$ $low(v, w)$                    (4.21)
                                   elseif $w = u$ then $[\![w \rightarrow v]\!]$ $\infty$
                                   elseif $pre[v] > pre[w]$ then $[\![v \dashrightarrow w]\!]$ $w$
                                   else $[\![w \dashrightarrow v]\!]$ $\infty$ $)))$
    end
```

(Here ∞ denotes a maximal vertex value; note that v would do.) An immediate simplification is to distribute the inner 'min' into the conditional.

```
$low(u, v)$ $\Leftarrow$
    begin
      $pre[v] \leftarrow p \leftarrow p + 1;$
      $\min_{w \in Adj(v)}($ if $pre[w] = 0$ then $[\![v \rightarrow w]\!]$ $\min(v, low(v, w))$                    (4.22)
                       elseif $w = u$ then $[\![w \rightarrow v]\!]$ $v$
                       elseif $pre[v] > pre[w]$ then $[\![v \dashrightarrow w]\!]$ $\min(v, w)$
                       else $[\![w \dashrightarrow v]\!]$ $v$ $)$
    end
```

Using preorder numbers. Recall that according to the lemma, if $low(v, w)$ is a descendent of v, then $v \rightarrow w$ is an articulation edge. Furthermore, it is always the case that the result of low is an ancestor or a descendent of v, so we can test the relation using the preorder numbering. This prompts us to specialize the definition of low to return preorder numbers rather than vertices. After several straightforward transformations, we obtain

```
$low(u, v)$ $\Leftarrow$
    begin
      $m \leftarrow pre[v] \leftarrow p \leftarrow p + 1;$
      for $w \in Adj(v)$ do
          if $pre[w] = 0$ then begin $[\![v \rightarrow w]\!]$
                                  $m \leftarrow \min(m, \ell)$ //
                                  (if $\ell \geq pre[v]$ then $[\![aedge(v, w)]\!]$ )                    (4.23)
                                end
                           where $\ell = low(v, w)$
          elseif $w = u$ then $[\![w \rightarrow v]\!]$
          elseif $pre[v] > pre[w]$ then $[\![v \dashrightarrow w]\!]$ $m \leftarrow \min(m, pre[w])$
          else $[\![w \dashrightarrow v]\!]$ )))
    end .
```

(We have aded an assertion noting when articulation edges are found.) Note that there is no action for two branches of the conditional.

Collecting components, revisited. Armed with this efficient method of locating articulation edges, we recall the *bcomps* algorithm derived earlier. That algorithm simultaneously collects the set of biconnected components and the set of edges in the current component. We now show how this algorithm can be merged with *low* to obtain an algorithm that simultaneously collects edges in the current component, collects biconnected components, and keeps track of the current *low* value. The resulting algorithm, while somewhat complicated, is very efficient, requiring time linear in the number of vertices and edges.

We start by substituting to obtain an expression procedure for the expression

$$\langle\, ba(u,v), low(u,v)\,\rangle\,.$$

After simplifying and renaming, we obtain

```
bcomps  $\Leftarrow$
    begin
        $pre[V] \leftarrow 0$ // $p \leftarrow 0$;
        $\bigcup_{\substack{root(r)\\ r \to s}} (\{B\} \cup A$  where $\langle B, A, \ell\rangle = balow(r, s))$
    end

balow(u, v)  $\Leftarrow$
    begin var $m$;
        $m \leftarrow pre[v] \leftarrow p \leftarrow p + 1$;
        $\langle \{\langle u, v\rangle\} \cup B,\ A,\ m\rangle$
           where $\langle B, A\rangle = \langle \bigcup, \bigcup\rangle_{w \in Adj(v)}$
                          ( if $pre[w] = 0$
                              then ( let $\langle B', A', \ell\rangle = balow(v, w)$ in
                                         if $\ell \geq pre[v]$
                                             then begin
                                                      $m \leftarrow \min(m, \ell)$;
                                                      $\langle \emptyset, \{B'\} \cup A'\rangle$
                                                  end
                                         else $\langle B', A'\rangle$ )                    (4.24)
                          elseif $pre[w] < pre[v]$  $\wedge$  $w \neq u$
                            then begin
                                     $m \leftarrow \min(m, pre[w])$;
                                     $\langle \{\langle v, w\rangle\}, \emptyset\rangle$
                                 end
                          else $\langle \emptyset, \emptyset\rangle$)
    end .
```

(This algorithm returns a triple instead of two nested pairs.) We now have a linear-time algorithm for computing the set of biconnected components in an undirected graph.

The biconnectivity algorithm. It is traditional in presentations of the biconnected component algorithm that components be emitted as they are found, rather than collected explicitly (as they are in the second component of the result of *balow*). The traditional presentation can be derived easily using transformations that introduce operations on global state and eliminate corresponding operations on explicit results. (The finite closure transformation makes implicit use of such transformations. Again, we do not go into details of the transformation method here; rather, we present this and the next transformation step in an informal manner.)

To carry out the transformation, we distinguish all operations that directly *change* the accumulated value of the second result. There is (essentially) only one place where this happens, which is when B' is added to A' in the innermost conditional. The effect of the transformation is to assert that B' is a biconnected component at that point.

```
bcomps ⇐
    begin
        $pre[V] \leftarrow 0$ // $p \leftarrow 0$;
        for $r \in V$ do (if $pre[r] = 0$ then balow$(\Lambda, r)$)
    end

balow$(u, v)$ ⇐
    begin var $m$;
        $m \leftarrow pre[v] \leftarrow p \leftarrow p + 1$;
        $\langle B, m\rangle$
          where $B = \bigcup_{w \in Adj(v)}$
                          (if $pre[w] = 0$
                              then ( let $\langle B', \ell\rangle$ = balow$(v, w)$ in
                                        let $B'' = B' \cup \{\langle u, v\rangle\}$ in
                                          if $\ell \geq pre[v]$
                                              then begin  [[$B''$ is a component]]                (4.25)
                                                      $m \leftarrow \min(m, \ell)$;
                                                      $\emptyset$
                                                    end
                                            else $B''$ )
                            elseif $pre[w] < pre[v] \;\wedge\; w \neq u$
                              then begin
                                        $m \leftarrow \min(m, pre[w])$;
                                        $\{\langle v, w\rangle\}$
                                    end
                            else $\emptyset$)
    end
```

(We have, in addition, "rotated" the outermost union to the caller; this allows most of the top-level loop of *bcomps* to be incorporated into *balow*.) In this program, *bcomps* is executed only for its side-effect of emitting components; its value can be is ignored.

A final transformation. Although it is not a necessary part of our development, a similar transformation can be carried out to eliminate the first result. In the prior example, the net effect on state of accumulating the set of biconnected components proved to be very simple; biconnected components were simply added to the set as they were found. In this case, however, the net changes to state corresponding to the way the edge set (viewed globally) is accumulated have a stack-like discipline; this is a result of our transformation of merging *bc* and *ae*. The difficulty here is that it is not known whether the edges found by the innermost call to *balow* are part of the current component until the *low* value is tested. The transformation method provides a means for introducing mechanism (in the form of data structure) to keep track of these changes in values.

There are three places where the accumulated set of edges is modified or used. At two of these, an edge is added to the current set. The third, in the innermost conditional, results in the possible removal of a number of edges from the accumulated set (depending on the *low* value). These edges are those that have been most recently accumulated, however, and they are all distinct. The data structure that results is thus a stack, and the following algorithm is obtained.

```
bcomps  ⇐
    begin
        $pre[V] \leftarrow 0$ // $p \leftarrow 0$ // stack←empty;
        for $r \in V$ do (if $pre[r] = 0$ then $balow(\Lambda, r)$)
    end
balow$(u, v)$  ⇐
    begin var $m$;
      $m \leftarrow pre[v] \leftarrow p \leftarrow p + 1$;
      for $w \in Adj(v)$ do
        if $pre[w] = 0$
            then begin
                    Push $\langle v, w \rangle$;
                    let $\ell = balow(v, w)$ in
                      if $\ell \geq pre[v]$
                          then begin
                                    $m \leftarrow \min(m, \ell)$;
                                    Pop to $\langle v, w \rangle$
                                end
                 end
          elseif $pre[w] < pre[v] \;\wedge\; w \neq u$
            then begin
                    $m \leftarrow \min(m, pre[w])$;
                    Push $\langle v, w \rangle$
                 end;
      $m$
    end
```
(4.26)

The stack-pop operation, 'Pop to $\langle v, w \rangle$,' pops all edges on the stack up to and including the edge $\langle v, w \rangle$ and emits this set of edges as a biconnected component.

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 mechanical 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 be 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; on the other hand, if only the final code is available, the conceptual history of the program must, in effect, be rediscovered.

Acknowledgements. We thank Margaret Beard and Greg Harris for careful readings of the manuscript and for insightful comments.

Bibliography

[AHU74]	Aho, A. V., J. E. Hopcroft, and J. D. Ullman, **The Design and Analysis of Computer Algorithms.** Addison-Wesley, 1974.
[AHU83]	Aho, A. V., J. E. Hopcroft, and J. D. Ullman, **Data Structures and Algorithms.** Addison-Wesley, 1983.
[Barstow80]	Barstow, D. R., *The roles of knowledge and deduction in algorithm design.* Yale Research Report178, April 1980.
[Bauer81]	Bauer, F. L., et al., *Programming in a wide spectrum language: a collection of examples.* Science of Computer Programming, Vol. 1, pp. 73–114, 1981.
[Bird80]	Bird, R. S., *Tabulation techniques for recursive programs.* Computing Surveys, Vol. 12, No. 4, pp. 403–417, 1980.
[Boyer75]	Boyer, R. S. and J. S. Moore, *Proving theorems about LISP functions.* Journal of the ACM, Vol. 22, No. 1, 1975.
[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.
[Gordon79]	Gordon, M. J., Milner, A. J., and C. P. Wadsworth, **Edinburgh LCF.** Springer-Verlag Lecture Notes in Computer Science, 1979.
[Green78]	Green C. C. and D. R. Barstow, *On program synthesis knowledge.* Artificial Intelligence, Vol. 10, p. 241, 1978.
[Knuth74]	Knuth D. E., *Structured programming with goto statements.* Computing Surveys, Vol. 6, No. 4, pp. 261–301, 1974.
[Manna79]	Manna Z. and R. Waldinger, *Synthesis: dreams $\Rightarrow$ programs.* IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979.
[Manna81]	Manna Z. and R. Waldinger, *Deductive synthesis of the unification algorithm.* Science of Computer Programming, Vol. 1, pp. 5–48, 1981.
[Paige81]	Paige, R. and S. Koenig, *Finite differencing of computable expressions.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 402–454, 1982.
[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.