Deriving Efficient Graph Algorithms*

John H. Reif[1]** and William L. Scherlis[2]***

[1] Computer Science Department, Duke University
reif@cs.duke.edu
[2] School of Computer Science, Carnegie-Mellon University
scherlis@cmu.edu

Abstract. Two case studies are presented that demonstrate the systematic derivation of efficient algorithms from simple combinatorial definitions. These case studies contribute to an exploration of evolutionary approaches to the explanation, proof, adaptation, and possibly the design of complex algorithms.

The algorithms derived are the linear-time depth-first-search algorithms developed by Tarjan and Hopcroft for strong connectivity and biconnectivity. These algorithms are generally considered by students to be complex and difficult to understand. The problems they solve, however, have simple combinatorial definitions that can themselves be considered inefficient algorithms.

The derivations employ systematic program manipulation techniques combined with appropriate domain-specific knowledge. The derivation approach offers evolutionary explanations of the algorithms that make explicit the respective roles of programming knowledge (embodied as program manipulation techniques) and domain-specific knowledge (embodied as graph-theoretic lemmas). Because the steps are rigorous and can potentially be formalized, the explanations are also proofs of correctness. We consider the merits of this approach to proof as compared with the usual *a posteriori* proofs. These case studies also illustrate how significant algorithmic derivations can be accomplished with a relatively small set of core program manipulation techniques.

---

* A summary version of the biconnectivity derivation appeared in the first LICS conference, 1983 (Springer LNCS 164).

** This research was supported in part by National Science Foundation Grant NSF-MCS79-21024, ONR N00014-80-C-0647, and DARPA/AFSOR F30602-01-2-0561, NSF ITR EIA-0086015, NSF EIA-0218376, and NSF EIA-0218359.

*** Author for correspondence. This research was supported in part by the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and U.S. Army Communications R & D Command under Contract DAAK80-81-K-0074, and the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. 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 NASA, DARPA, or the U.S. Government.

1 Introduction

The design 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 in general, there is much to be learned—both about algorithms and about programming—from the study of the structure of *derivations* of complex algorithms.

Program manipulation techniques provide a natural way of both explaining and reasoning about 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 that also serves as a specification of the problem to an efficient implementation that may be complex and structurally opaque. Our approach is to explicate algorithms by making design decisions explicit as derivation steps, even when they may have no obvious structural manifestation in the code. A long-term scientific goal is to understand the relative explicatory merits of derivation steps and structural abstraction. Consideration of this goal may lead, for example, to better abstractions with respect to to the derivation steps themselves.

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 are executable. That is, we choose to interpret them as algorithms, even though—as algorithms—they might be inefficient. To illustrate how simple these specifications can be, we give here our specification of the path predicate for directed graphs.

Let $G = \langle V, Adj \rangle$ be a directed graph with vertices V and adjacency-set function $Adj : V \to P(V)$. We define the predicate $\mathit{path}(u, v)$ to hold when there is a path in G from vertex u to vertex v. Leaving the graph G as an implicit parameter, we can write:

$$\mathit{path}(u, v) \quad \Leftarrow_{\mathbf{false}} \quad (u = v) \ \mathbf{or}\ (\exists w \in Adj(u))\mathit{path}(w, v))$$

There is a path from u to v if they are the same or there is a vertex adjacent to u from which there is a path to v.

This is a *closure* definition. Closure definitions are made with respect to an *accumulation* operation, which is a semilattice with identity over a finite domain. In this example, the semilattice is disjunction; the identity is **false**, and the finite set is the two truth values. Intuitively, whenever a "recursive call" to $\mathit{path}(u, v)$ is made for a pair $\langle u, v \rangle$ that has already been calculated *or* is already *being* calculated, then the identity **false** should be the result value. Since the set of

vertices is finite, this assures termination of all evaluations of *path*. (Details on the semantics of closure definitions are in the next section.)

With this definition, we obtain a straightforward specification of the strongly-connected components of a directed graph. Two vertices r and s are in the same strongly-connected component if there is a path from r to s and a path from s to r. The strongly-connected components of G are thus the elements of the set *strong*. Again leaving the graph as an implicit parameter, we have:

$$\mathit{strong} \quad \Leftarrow \quad \{\ \{s \in V \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\} \mid r \in V\ \}\,.$$

Taken naively, these two definitions are a cubic-time algorithm for computing the strongly-connected components of a directed graph.

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 presented in [20] and [1] in conventional fashion. 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 [2]. (Similar techniques can be used to derive the almost-linear-time algorithms of [21] for flow-graph reducibility.) In the conclusion we discuss further the implications of this work. Features of the algorithmic language we use and the various program manipulation techniques are presented along the way.

Combinatorial lemmas and program transformations. 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. It is theoretically possible to prove the lemmas entirely in the language of a programming logic, but 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. We could hypothesize that, in many algorithm derivations, much of the difficulty comes from the diversity of this domain-specific knowledge and its employment, rather than from the diversity of programming-specific knowledge. That hypothesis is supported by these derivations in the sense that a relatively small set of program manipulation rules suffices for most of the steps. But these small steps are generally used in aggregates, and the variety of aggregates can be large.

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 [18], [23], and [5]. These transformations are used to transform closure definitions, such as specification of *path* above, into more conventional definitions.

In addition, we make use of the transformations of [19] (which are similar in spirit to those described in [6], but for which there is a guarantee of strong equivalence—see [17], [22]) in order to specialize function definitions and to effect the merging or "jamming" of loops. Discussion of loop jamming techniques appears in [13]. (This paper is self-contained—no prior knowledge of the details of the above is required to follow the derivations.)

The programming language we use is an ML-like applicative language with imperative features to allow sequencing and explicit management of state. Because it is difficult to reason about and manipulate programs that are overly committed with respect to order of computation and data representation, we have sought to keep the programming language as abstract as possible in these respects. In order to allow derivations to be more perspicuous, certain features are included in the language of programs that may be difficult to implement efficiently, though they have clear semantics. Examples of such features include (1) closure definitions (of *path*, for example), (2) *expression procedures* (see below), and (3) a generous syntax for iteration. Intermediate programs in the derivation process may make extensive use of these features, but the programs that are the end results of the process generally do not. This means that, while precise meaning must be defined, these language features do not necessarily need to be implemented efficiently, or even at all. We explain the unusual features of the language as they are encountered.

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

As in [11] and [12], 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 may play an important role in the automation of these techniques. Other examples and approaches to program derivation are described in [8], [3], [10], and [4], 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 [9], in the later derivations.

The development in the first part of this section is identical for directed and undirected graphs. We 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 $v \in V$, $Adj(v)$ is the set of vertices adjacent to v. Thus $\langle u, v \rangle \in E$ if and only if $v \in Adj(u)$. 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.

$$\mathit{path}(u, v) \quad \Leftarrow_{\mathbf{false}} \quad (u = v \ \mathbf{or}\ (\exists w \in Adj(u))\mathit{path}(w, v)) \tag{1}$$

As noted above, this is a *closure* definition, which is a definition that specifies an accumulation of values. The accumulation operation in this case is disjunction, and the "values" accumulated are pairs $\langle u, v \rangle$ for which *path* is true. A useful operational intuition is that infinite recursions in the computation of $\mathit{path}(u, v)$ are avoided by replacing all redundant calls by the identity of the accumulation operation, which for disjunction is **false**. This replacement is made even when the redundancy is with a call in progress.

More generally, closure definitions are of the form

$$f(x) \quad \Leftarrow_i \quad h(x) \ \oplus \bigoplus_{z \in g(x)} f(z), \tag{2}$$

where x is an element of a finite domain D, computation of h and g does not involve f, and $\oplus$ is a semilattice operator with identity i in which infinite sums are defined (that is, the partial order induced by $\oplus$ is a complete partial order). Intuitively, the identity is the minimal element of an accumulation. In the case of *path*, $\oplus$ is disjunction and the identity is **false**. (The existential quantification is over a finite set, and so is equivalent to a finite disjunction.) Another semilattice operation commonly used in these derivations is set union with the empty set, typically over the finite sets of vertices or edges.

Closure definitions have a fixed-point semantics in which f is always defined as a limit of an ascending chain of sets:

$h(x)$

$h(x) \ \oplus \ \bigoplus_{z_1 \in g(x)} h(z_1)$

$h(x) \ \oplus \ \bigoplus_{z_1 \in g(x)} (h(z_1) \oplus \bigoplus_{z_2 \in g(z_1)} h(z_2))$

$\ldots$

Closure definitions are analogous to memoization, which is a class of techniques (attributed to Donald Michie) for retaining previously computed values

to avoid redundant computation. The key differences between closure definitions and conventional memoization are (1) all redundant invocations return the same identity value, and (2) subsidiary but identical invocations are considered redundant even before the main ancestor invocation completes.

The finite closure transformation. Closure definitions are transformed into conventional definitions by introducing explicit mechanism to track redundant calls (including calls in progress) and committing to order of evaluation. Closure definitions are related to "closed-world" database techniques such as described in [7] and [15].

We carry out these transformations by introducing implicit state change into our language of recursive definitions. In the case of *path*, this involves introducing explicit data structure to record vertex pairs as they are considered. When $\mathit{path}(u, v)$ is to be computed and $\langle u, v\rangle$ is already marked, then **false** is the correct return value. State changes can be made either implicitly (by introducing imperative operations) or explicitly (by adding a new "memo" parameter to f). Regardless, it is difficult to manipulate programs involving state, so it is best to delay the transformation whenever possible. In our derivation of simple depth-first search, we preserve the closure formulation until the next transformation step is complete.

Specialization using expression procedures. Nearly all of the program derivation steps in our derivations are based on the *expression-procedure* technique. This technique is presented in detail and proved correct in [18] with improvements by Sands [17] and Tullsen [22]. We provide sufficient details here for this paper to be self-contained. This technique provides a straightforward way to specialize both recursive program definitions and (we assert without proof) closure definitions. We describe this technique here informally by means of an example involving the *path* definition.

Suppose we want to collect vertices $v \in V$ reachable from a given vertex u.

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

This set comprehension suggests that the value of the set can be calculated by enumerating all vertices v and testing $\mathit{path}(u, v)$ for each. This method is inefficient because it requires multiple traversals of the same graph. We therefore consider *specializing* the definition of *path* to the computational context of the set abstraction. That is, we use the definition of path to develop an algorithm for computing $\{v|\mathit{path}(u, v)\}$. This will enable application of finite closure on the reachability sets rather than on individual paths.

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

$$\mathit{path}(u, v) \quad \Leftarrow_{\mathbf{false}} \quad (u = v \ \ \mathbf{or}\ (\exists w \in \mathit{Adj}(u))\mathit{path}(w, v))$$

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

$$\{v|\mathit{path}(u,v)\} \quad \Leftarrow_{\emptyset} \quad \{v|(u = v \textbf{ or } (\exists w \in Adj(u))\mathit{path}(w,v)\}. \tag{3}$$

This definition is called an *expression procedure*. Intuitively, it denotes a procedure for computing values of instances of its left-hand side. It can be given precise meaning within the framework of a substitution-based operational model, for example. (This substitution operation, called *composition*, has certain soundness restrictions, which are described in the above-mentioned references.)

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. This is accomplished by distributing the set abstraction inward and simplifying.

$$\{v|\mathit{path}(u,v)\} \quad \Leftarrow_{\emptyset} \quad \{u\} \quad \cup \quad \bigcup_{w \in Adj(u)} \{v|\mathit{path}(w,v)\} \tag{4}$$

As a result of this simplification, an instance of the left-hand side appears on the right. This means that the definition could be evaluated either by calling *path* or by calling itself (the expression procedure) recursively. (Note that this is still a closure definition over the semilattice of finite set union, with identity $\emptyset = \{v|\textbf{false}\}$.) Unfolding and procedure abstraction can be done on the basis of the expression-procedure call. The operational intuition is that this definition 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 renaming operation is done in two steps. The entire body of the expression procedure first is abstracted into a definition for a new name, *dfs*. The only free variable in the expression is u, so we obtain:

$$\begin{array}{l} \{v|\mathit{path}(u,v)\} \quad \Leftarrow_{\emptyset} \quad \mathit{dfs}(u) \\ \mathit{dfs}(u) \quad \Leftarrow_{\emptyset} \quad \{u\} \cup \bigcup_{w \in Adj(u)} \{v|\mathit{path}(w,v)\} \end{array}$$

We can "unfold" the expression procedure call in the second definition using the first of these definitions. (Observe that with expression procedures multiple execution pathways exist, but all yield identical results when transformation operations are sound.)

$$\begin{array}{l} \{v|\mathit{path}(u,v)\} \quad \Leftarrow_{\emptyset} \quad \mathit{dfs}(u) \\ \mathit{dfs}(u) \quad \Leftarrow_{\emptyset} \quad \{u\} \cup \bigcup_{w \in Adj(u)} \mathit{dfs}(w) \end{array} \tag{5}$$

As an aside, because

$$\mathit{path}(u,v) \;=\; v \in \{v|\mathit{path}(u,v)\}, \tag{6}$$

we can substitute to obtain

$$\begin{array}{l} \mathit{path}(u,v) \;\Leftarrow\; v \in \mathit{dfs}(u) \\ \mathit{dfs}(u) \;\Leftarrow_{\emptyset}\; \{u\} \cup \bigcup_{w \in Adj(u)} \mathit{dfs}(w). \end{array} \tag{7}$$

Observe that this definition accomplishes a frequency reduction for $\mathit{path}(u,v)$ with respect to u.

Finite closure revisited. At this point we can make the finite closure transformation that was postponed earlier. Note that '$\cup$' is the "accumulator" for this semilattice of finite sets and $\emptyset$ is the identity. The transformation entails making explicit the replacement of redundant calls to *dfs* by $\emptyset$. The definition below illustrates one of several possible ways to accomplish this. We first define an explicit array of booleans indexed by the vertices of the graph:

$$\mathit{visit} : V \to \mathbf{bool}$$

The first action before calling *dfs* is to initialize the entire array to **false**.

$$\begin{array}{l} \mathit{path}(u,v) \;\Leftarrow\; v \in \mathit{dfs}(u) \\ \\ \mathit{dfs}(u) \;\Leftarrow\; \mathbf{begin} \\ \qquad \mathit{visit}[V] \leftarrow \mathbf{false}\ ; \\ \qquad \mathit{dfs}'(u) \\ \quad \mathbf{end} \\ \\ \mathit{dfs}'(u) \;\Leftarrow\; \mathbf{begin} \\ \qquad \mathit{visit}[u] \leftarrow \mathbf{true}\ ; \\ \qquad \{u\} \cup \bigcup_{w \in Adj(u)} (\mathbf{if}\ \mathit{visit}[w]\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \mathit{dfs}'(w)) \\ \quad \mathbf{end} \end{array} \tag{8}$$

Our convention is that imperative statements are always enclosed in blocks, and the value of a block is the value of the last expression. In this example, we treat the *visit* array as a variable that is bound in some outer scope that includes the two definitions above.

In this program, 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.

More generally, a definition of the form

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

can be transformed to

$$
\begin{array}{lcl}
f(x) & \Leftarrow & \textbf{begin} \\
 & & \quad \mathit{visit}[D] \leftarrow \textbf{false} \ ; \\
 & & \quad f'(x) \\
 & & \textbf{end} \\
f'(x) & \Leftarrow & \textbf{begin} \\
 & & \quad \mathit{visit}[x] \leftarrow \textbf{true} \ ; \\
 & & \quad h(x) \ \oplus \ \bigoplus_{z \in g(x)} (\textbf{if } \mathit{visit}[z] \textbf{ then } i \textbf{ else } f'(z)) \\
 & & \textbf{end}
\end{array}
$$

where D is the finite universe of $\oplus$.

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 doing the accumulation, and thus is more clear for our purposes.

Distributivity and finite closure. A useful property of finite closure is that the initialization of the *visit* array can be distributed across $\oplus$. That is,

$$
\begin{array}{lcl}
f(x) \oplus f(y) & \Leftarrow & \textbf{begin} \\
 & & \quad \mathit{visit}[D] \leftarrow \textbf{false} \ ; \\
 & & \quad f'(x) \oplus f'(y) \\
 & & \textbf{end}
\end{array}
$$

More generally,

$$
\begin{array}{l}
\bigoplus_{z \in g(x)} (\textbf{begin } \mathit{visit}[D] \leftarrow \textbf{false}; f'(z) \textbf{ end}) \\
\qquad \Longleftrightarrow \\
\textbf{begin } \mathit{visit}[D] \leftarrow \textbf{false}; \bigoplus_{z \in g(x)} f'(z) \textbf{ end}
\end{array}
$$

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

$$
\mathit{comps} \ \Leftarrow \ \{ \ \{v \in V \mid \mathit{path}(r, v)\} \mid r \in V \ \}. \tag{9}
$$

As an aside, we note two other ways to notate this "comprehension of comprehensions." The first is a union of singletons, which is concise but perhaps less perspicuous than the formula above.

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

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

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

In any case, substitution of the improved definition of *path* above into the definition of *comps* and simplification yield

$$\mathit{comps} \Leftarrow \bigcup_{r \in V} \{\textbf{begin } \mathit{visit}[V] \leftarrow \textbf{false}; \ \mathit{dfs}(r) \textbf{ end}\}. \tag{10}$$

This definition performs redundant searches, with worst case running time $O(|V|^2)$. The redundant searches can be avoided by making use of the *visit* array used by *dfs*. We employ the distributivity property of finite closure to initialize the *visit* array once for all calls to *dfs*. This gives the linear-time program:

$$\begin{array}{l} \mathit{comps}(V) \Leftarrow \textbf{begin} \\ \qquad \mathit{visit}[V] \leftarrow \textbf{false} \ ; \\ \qquad \bigcup_{r \in V} (\textbf{if } \mathit{visit}[r] \textbf{ then } \emptyset \textbf{ else } \mathit{dfs}'(r)) \\ \quad \textbf{end} \end{array} \tag{11}$$

$$\begin{array}{lll} \mathit{dfs}'(u) & \Leftarrow & \textbf{begin} \\ & & \quad \mathit{visit}[u] \leftarrow \textbf{true} \ ; \\ & & \quad \{u\} \cup \bigcup_{w \in Adj(u)} (\textbf{if } \mathit{visit}[w] \textbf{ then } \emptyset \textbf{ else } \mathit{dfs}'(w)) \\ & & \textbf{end} \end{array}$$

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. Note that the particular 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.

Tree and tree traversals. The fast depth-first algorithms rely on combinatorial properties of depth-first spanning forests. 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 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 *forest* is a directed graph whose vertices either have indegree one or indegree zero. A vertex with zero indegree is called

a *root*. A vertex with zero outdegree is called a *leaf*. Other vertices are *internal nodes*. We use the notation '$u \xrightarrow{T} v$' as shorthand for '$\langle u, v\rangle \in E$', where E is the set of edges in the tree.

The set of vertices of a tree can be enumerated without repetitions by traversing the edges of the tree and recursively enumerating subtrees. 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. (The tree T is an implicit argument.)

$$\begin{array}{l} \mathit{trav}(u) \Leftarrow \textbf{begin} \\ \qquad \mathit{examine}(u) \;/\!/\; \textbf{forpar}\ w\ \textbf{st}\ u \xrightarrow{T} w\ \textbf{do}\ \mathit{trav}(w) \\ \quad \textbf{end} \end{array} \tag{12}$$

The symbol '//' indicates explicit avoidance of commitment to computation seqeuencing, as does the '**forpar**' notation. This lack of commitment may be interpreted as parallel or nondeterminstic sequential execution of all the specified instances of the loop body. By convention, we use this kind of nondeterminism when all possible executions terminate and yield identical answers, even when there may inefficient execution paths. By this convention, we can infer that the final result of Algorithm (12) above is not influenced by the order of calls to *examine*.)

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 ';' and when '**forpar**' is replaced by '**for**' which, for ordered trees, implies that the loop cases are evaluated in order of the edges. (Hereafter, this order is assumed implicitly when '**for**' is notated.)

$$\begin{array}{l} \mathit{trav}(u) \Leftarrow \textbf{begin} \\ \qquad \mathit{examine}(u); \\ \qquad \textbf{for}\ w\ \textbf{st}\ u \xrightarrow{T} w\ \textbf{do}\ \mathit{trav}(w) \\ \quad \textbf{end} \end{array} \tag{13}$$

Relative preorder position can be tested using an instance of Algorithm (13), but not very efficiently. 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. In the algorithm below, the array $pre[V]$ below maps vertices to their preorder numbers. Let r be the root of a tree.

begin $p \leftarrow 0$; *trav*(r); $pre[V]$ **end**;

$$
\begin{array}{l}
\mathit{trav}(u) \Leftarrow \textbf{begin} \\
\qquad pre[u] \leftarrow p \leftarrow p+1; \\
\qquad \textbf{for } w \textbf{ st } u \xrightarrow{\mathrm{T}} w \textbf{ do } \mathit{trav}(w) \\
\quad \textbf{end}
\end{array}
\tag{14}
$$

The result of this program is the array *pre* containing the preorder numbers assigned to the vertices of the tree rooted at r.

A similar algorithm can be derived for computing the postorder numbering. By merging Algorithm (14) with this new algorithm, we obtain

$$\textbf{begin } p \leftarrow 0;\ e \leftarrow 0;\ \mathit{trav}(r);\ \langle pre[V], post[V] \rangle \textbf{ end};$$

$$
\begin{array}{l}
\mathit{trav}(u) \Leftarrow \textbf{begin} \\
\qquad pre[u] \leftarrow p \leftarrow p+1; \\
\qquad \textbf{for } w \textbf{ st } u \rightarrow w \textbf{ do } \mathit{trav}(w); \\
\qquad post[u] \leftarrow e \leftarrow e+1 \\
\quad \textbf{end } .
\end{array}
\tag{15}
$$

(Transformation steps are omitted. A more interesting example of loop merging using 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. The notation $u \succ v$ can be read as "u is a descendent of v."

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

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

Lemma 1. *Let T be an ordered tree with vertices numbered in preorder (visiting nodes before their subtrees in order) in array $pre[V]$ and in postorder (visiting subtrees in order prior to nodes) in array $post[V]$. Then*

$$u \succ v \qquad \text{if and only if } pre[u] > pre[v] \text{ and } post[u] < post[v].$$

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. Note that it is crucial that we traverse tree in left to right order, as specified in Algorithm (13).

Furthermore, u is to the left of v in T if and only if u precedes v in both preorder and postorder. Thus, which of the four possible relative positions held by 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 forest structure on the edges of the graph being searched. That is, the subset of the *graph* edges that are 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 denotes preconditions when it appears on a left hand side. See, for example, the derivation of Algorithm (21).) We can annotate the graph traversal Algorithm (8) to obtain

$$\begin{array}{l} \mathit{path}(u,v) \Leftarrow \textbf{begin } \mathit{visit}[V] \leftarrow \textbf{false};\ v \in \mathit{dfs}'(u) \textbf{ end} \\ \\ \mathit{dfs}'(u) \Leftarrow \textbf{begin} \\ \qquad\qquad \mathit{visit} \leftarrow \textbf{true}; \\ \qquad\qquad \{u\} \cup \bigcup_{w \in Adj(u)} (\textbf{if } \mathit{visit}[w] \textbf{ then } \emptyset \textbf{ else } [\![u \xrightarrow{\mathrm{T}} w]\!]\ \mathit{dfs}'(w)) \\ \qquad\quad \textbf{end} \end{array} \tag{16}$$

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 doing a preorder numbering of a depth-first search forest of a graph. This is accomplished by deriving a program that simultaneously computes dfs' and trav. That is, we first compute dfs' to identify tree edges, and then $\mathit{trav}$to follow those tree edges and assign preorder numbers. For simplicity, we assume the directed graph is connected, which means the forest will have a single root r. We accomplish this by writing an expression procedure for the block:

$$\textbf{begin } \mathit{dfs}'(u);\ \mathit{trav}(u) \textbf{ end}$$

Unfolding the initialization, reordering, and adding some additional structure yields the following main program. Here r is an arbitrary vertex of G. Note the slight jerrymander to create an instance of the expression procedure block. Note also that we ignore the result of dfs' and return only the *pre* array containing those numbers.

$$\textbf{begin } \mathit{visit}[V] \leftarrow \textbf{false};\ p \leftarrow 0;\ \langle(\textbf{begin } \mathit{dfs}'(r);\ \mathit{trav}(r) \textbf{ end}), \mathit{pre}[V]\rangle \textbf{ end}$$

We define the expression procedure by substituting the definitions for the preorder version of *trav* (Algorithm (14)) and *dfs'* (Algorithm (16)) into the block and doing some initial restructuring:

$$
\begin{array}{l}
(\textbf{begin } \mathit{dfs}'(u);\ \mathit{trav}(u) \textbf{ end}) \Leftarrow \\
\quad \textbf{begin} \\
\qquad \mathit{visit}[u] \leftarrow \textbf{true}; \\
\qquad \mathit{pre}[u] \leftarrow p \leftarrow p+1; \\
\qquad \{u\} \cup (\ \textbf{begin} \\
\qquad\qquad \left(\bigcup_{\substack{w \in Adj(u) \\ \neg \mathit{visit}[w]}} [\![u \xrightarrow{\mathrm{T}} w]\!]\ \mathit{dfs}'(w) \right) \\
\qquad\qquad\quad \textbf{for } w \textbf{ st } u \xrightarrow{\mathrm{T}} w \textbf{ do } \mathit{trav}(w); \\
\qquad\qquad \textbf{end}\ ) \\
\quad \textbf{end}
\end{array}
\tag{17}
$$

We assert $u \rightarrow w$ because it is true just when $w \in Adj(u) \wedge \neg \mathit{visit}(w)$. This implies that the two loops range over the same set. Because they do not interact, they can be *merged*—combined into a single iteration.

At this point, we make two further simplifications. First, we *rename* the block being defined to the simple name *dfs*(superseding the previous use of this name—this bad habit will continue throughout the derivation below). Second, we observe that *if* $pre[V]$ is initialized to 0, then

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

and we can eliminate the *visit* array and use *pre* instead.

Because the *dfs'* results are unused, simplifications eliminate calculation of the set union. The following shorter program results after unfolding, renaming, and eliminating the expression procedure:

$$
\begin{array}{l}
\textbf{begin } p \leftarrow 0;\ \mathit{pre}[V] \leftarrow 0;\ \mathit{dfs}(r);\ \mathit{pre}[V] \textbf{ end} \\
\\
\mathit{dfs}(u)\ \Leftarrow \\
\quad \textbf{begin} \\
\qquad \mathit{pre}[u] \leftarrow p \leftarrow p+1\ ; \\
\qquad \textbf{for } w \textbf{ st } w \in Adj(u)\ \wedge\ \mathit{pre}[w] = 0 \textbf{ do } \mathit{dfs}(w) \\
\quad \textbf{end}
\end{array}
\tag{18}
$$

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 complete, the *post* ordering can be computed as well.

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

dfs(u) ⇐
  begin
    pre[u] ← p ← p + 1 ;                                          (19)
    for w st (w ∈ Adj(u) ∧ pre[w] = 0) do dfs(w)
    post[u] ← e ← 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, $u \stackrel{\mathrm{T}}{\rightarrow} v$ to indicate tree edges, where u is the parent node of v. 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 (8), we observe that the fronds are exactly those edges $\langle u, v\rangle$ for which the *visit*$[w]$ test is true but (since the graph is undirected) such that w is not the parent of u in the search tree. As before, we recycle the name *dfs*.

```
dfs'(u) ⇐
  begin
    visit[u] ← true;
    {u} ∪ ⋃_{w∈Adj(u)}(if visit[w]                                 (20)
                        then [[if w ↛↠ u then u ⇔ w]] ∅
                        else [[u → w]] dfs'(w))
  end
```

Here we have decorated Algorithm (8) with assertions distinguishing the two sets of edges. The assertion in the **then**clause states that if the already-seen adjacent node is not the immediate parent node of u, then the edge being traversed is a frond. In order to classify all graph edges, the test within the assertion needs to be computable. Observe, however, that the parent of u is known whenever *dfs*$'$ is called. We can use the specialization technique to introduce a new parameter to *dfs* that will be the parent of u in the depth-first-search tree being generated. We do this by forming an expression procedure for

$$[\![v \stackrel{\mathrm{T}}{\rightarrow} u]\!]\ \mathit{dfs}(u).$$

Recall that in an expression procedure name an assertion denotes a precondition. The additional information provided by the assertion allows us to introduce a case analysis on the parentage of u into the **then** clause, transforming

$$[\![\textbf{if } w \not\mapsto u \textbf{ then } u \Leftrightarrow w]\!]\ \emptyset$$

into

$$\begin{array}{l}\textbf{if } w \neq v \textbf{ then}[\![u \Leftrightarrow w]\!]\ \emptyset\\ \quad\textbf{else } [\![w \xrightarrow{\mathrm{T}} u]\!]\ \emptyset\end{array}$$

After renaming (v is the parent of u) and reorienting the nested conditionals, we obtain the definition:

$$\begin{array}{l}\mathit{dfs}(u,v) \Leftarrow\\ \quad\textbf{begin}\\ \quad\quad \mathit{visit}[u] \leftarrow \textbf{true};\\ \quad\quad \{u\} \cup \bigcup_{w\in Adj(u)}(\textbf{if } \neg \mathit{visit}[w] \textbf{ then } [\![u \rightarrow w]\!]\ \mathit{dfs}(w,u)\\ \qquad\qquad\qquad\qquad\qquad \textbf{else if } w \neq v \textbf{ then } [\![u \Leftrightarrow w]\!]\ \emptyset\\ \qquad\qquad\qquad\qquad\qquad \textbf{else } [\![w \rightarrow u]\!]\ \emptyset)\\ \quad\textbf{end}.\end{array} \tag{21}$$

In order to use this definition, one exceptional case must be considered, which is the initial call when u is a root. To handle this case we can either unroll the recursion one step, introducing an additional auxiliary dfs'' definition, or we can add a special non-vertex value Λ to denote a phantom parent node for the root case. From a derivational point of view, these are equivalent, and, in the interests of simplifying the presentation, we select the latter case.

Finally, we carry through the transformation steps described earlier to obtain an algorithm similar to Algorithm (19).

$$\textbf{begin } p \leftarrow 0;\ e \leftarrow 0;\ \mathit{pre}[V] \leftarrow 0;\ \mathit{dfs}(r,\Lambda);\ \langle \mathit{pre}[V], \mathit{post}[V]\rangle \textbf{ end}$$

$$\begin{array}{l}\mathit{dfs}(u,v)\ \Leftarrow\\ \quad\textbf{begin}\\ \quad\quad \mathit{pre}[u] \leftarrow p \leftarrow p+1\ ;\\ \quad\quad \textbf{for } w \textbf{ st } (w \in Adj(u)\ \wedge\ \mathit{pre}[w] = 0)\\ \quad\quad\quad \textbf{do } (\textbf{if } \mathit{pre}[w] = 0 \textbf{ then } [\![u \rightarrow w]\!]\ \mathit{dfs}(w,u)\\ \qquad\qquad\qquad\quad \textbf{else if } w \neq v \textbf{ then } [\![u \Leftrightarrow w]\!]\ \emptyset\\ \qquad\qquad\qquad\quad \textbf{else } [\![w \rightarrow u]\!]\ \emptyset);\\ \quad\quad \mathit{post}[u] \leftarrow e \leftarrow e+1;\\ \quad\textbf{end}\end{array} \tag{22}$$

A further specialization. In the biconnectivity algorithm derivation, we will need to classify fronds into *forward fronds* and *reverse fronds*. Observe that

when $u \Leftrightarrow w$ in a depth-first search tree, 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 $w \Leftarrow u$.

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

$$pre[u] < pre[w] \quad \text{implies} \quad u \not\succ w.$$

and analogously 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 slightly reorganize the Algorithm (21) above to obtain the following undirected depth-first search algorithm. This algorithm fully classifies the undirected graph edges into directed tree edges and fronds (though it does not make use of this information).

$$\begin{array}{l}
dfs(u, v) \Leftarrow \\
\quad \textbf{begin} \\
\quad pre[u] \leftarrow p \leftarrow p + 1 \ ; \\
\quad \textbf{for } w \textbf{ st } (w \in Adj(u) \ \wedge \ pre[w] = 0) \\
\qquad \textbf{do } (\textbf{if } pre[w] = 0 \textbf{ then } [\![u \rightarrow w]\!] \ dfs(w, u) \\
\qquad\qquad \textbf{else if } w = v \textbf{ then } [\![w \rightarrow u]\!] \ \emptyset \\
\qquad\qquad \textbf{else if } pre[u] > pre[w] \textbf{ then } [\![u \dashrightarrow w]\!] \ \emptyset \\
\qquad\qquad \textbf{else } [\![w \dashrightarrow u]\!] \ \emptyset) \\
\quad post[u] \leftarrow e \leftarrow e + 1; \\
\textbf{end}
\end{array} \tag{23}$$

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

This classification of cases will be useful when we 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 and based on the work of Tarjan and Hopcroft. Let $G = (V, E)$ be a directed graph and let u and v range over the vertices V. Recall the original definition of *path*,

$$path(u, v) \quad \Leftarrow_{\textbf{false}} \quad (u = v \textbf{ or } (\exists w \in Adj(u)) path(w, v)). \tag{24}$$

The *path* relation holds between u and v just when there is a directed path in G from u to v.

Two vertices u and v in a graph are *strongly connected* if $\mathit{path}(u,v)$ and $\mathit{path}(v,u)$ 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 the transitivity of the path relation implies that v and w are themselves also strongly connected. The strongly-connected components thus partition the vertices of a directed graph. This leads us to take the following definition as our starting specification of the strongly-connected components.

$$\mathit{strong} \quad \Leftarrow \quad \{\ \{s \in V \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\} \mid r \in V\ \}\,. \tag{25}$$

If path requires time linear in the number of vertices, then this definition, evaluated naively, requires $O(|V|^3)$ time.

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 does not produce useful results for this formulation of the definition because the recursions do not naturally merge. A natural response to this failure is to generalize, separating either the r or s pairs of parameters into distinct variables. We separate the two instances of r into two distinct variables, u and r. (Generalization is a common heuristic for obtaining inductive proofs and has been incorporated into several automatic systems; [11] describes examples.)

$$\begin{array}{l}\{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \quad \Leftarrow_{\emptyset} \\ \quad \{s \mid ((u=s)\ \mathbf{or}\ (\exists w \in \mathit{Adj}(u))\mathit{path}(w,s)) \ \wedge\ \mathit{path}(s,r)\}\end{array} \tag{26}$$

We have substituted the definition of path in the first instance, but not the second. We simplify by distributing the conjunction and set abstraction inward.

$$\begin{array}{l}\{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \quad \Leftarrow_{\emptyset} \\ \quad \{s \mid u = s \wedge \mathit{path}(s,r)\} \ \cup\ \bigcup_{w \in \mathit{Adj}(u)} \{s \mid \mathit{path}(w,s) \wedge \mathit{path}(s,r)\}\end{array} \tag{27}$$

This expression procedure is recursive. Unfortunately, we are forced to test $\mathit{path}(u,r)$ (on the left side of the union, where $u = s$) on every iteration. This predicate is certainly true, for example, on the initial call,

$$\mathit{strong} \Leftarrow \textstyle\bigcup_{r \in V} (\{[\![\mathit{path}(r,r)]\!]\ \{s \mid \mathit{path}(r,s) \wedge \mathit{path}(s,r)\}\})\,, \tag{28}$$

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 with $\mathit{path}(u,r)$ is assumed to be true on entry.

$$[\![\mathit{path}(u,r)]\!]\ \{s|\mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \quad \Leftarrow_{\emptyset} \quad \{s|u = s \wedge \mathit{path}(s,r)\} \ \cup\ \bigcup_{w \in Adj(u)} \{s|\mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \tag{29}$$

The assumption allows simplification of the left-hand side of the union to the singleton $\{u\}$. The assertion can be established, or not, for the recursive case by introducing an obvious case analysis.

$$\begin{array}{l} [\![\mathit{path}(u,r)]\!]\ \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \quad \Leftarrow_{\emptyset} \\ \quad \{u\} \ \cup\ \bigcup_{w \in Adj(u)} \\ \qquad (\ \textbf{if}\ \mathit{path}(w,r)\ \textbf{then}\ \{s|\mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \\ \qquad\quad \textbf{else}\ \{s|\mathit{path}(w,s) \wedge \mathit{path}(s,r)\}\ ) \end{array} \tag{30}$$

The assertion is clearly true in the **then** clause because it is the test. In the **else** clause, where there is no path from w to r, it must follow from the transitivity of *path* that there can be no vertices s that satisfy the condition.

$$\begin{array}{l} [\![\mathit{path}(u,r)]\!]\ \{s \mid \mathit{path}(u,s) \wedge \mathit{path}(s,r)\} \quad \Leftarrow_{\emptyset} \\ \quad \{u\} \ \cup\ \bigcup_{w \in Adj(u)} \\ \qquad (\ \textbf{if}\ \mathit{path}(w,r)\ \textbf{then}\ [\![\mathit{path}(w,r)]\!]\ \{s|\mathit{path}(w,s) \wedge \mathit{path}(s,r)\} \\ \qquad\quad \textbf{else}\ \emptyset\ ) \end{array} \tag{31}$$

The effect of this transformation sequence is now clear—responsibility for the $\mathit{path}(u,r)$ test has been shifted to the caller, and the conjecture is *not* established. These small improvements will facilitate later steps.

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

$$\begin{array}{l} \mathit{strong} \Leftarrow \bigcup_{r \in V} \{\mathit{sc}(r,r)\} \\ \\ \mathit{sc}(u,r) \quad \Leftarrow_{\emptyset} \quad \{u\} \ \cup\ \bigcup_{w \in Adj(u)} (\textbf{if}\ \mathit{path}(w,r)\ \textbf{then}\ \mathit{sc}(w,r)\ \textbf{else}\ \emptyset) \\ \\ \mathit{path}(u,v) \quad \Leftarrow_{\textbf{false}} \quad (u = v\ \textbf{or}\ (\exists w \in Adj(u))\mathit{path}(w,v)) \end{array} \tag{32}$$

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 r. 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

$$\begin{array}{l} \mathit{revpath}(u,v) \quad \Leftarrow_{\mathbf{false}} \\ \quad (u = v \ \mathbf{or}\ (\exists w \in Adj^{-1}(u))\mathit{revpath}(w,v)) \end{array} \tag{33}$$

instead of *path*, and compute ancestry using *its* depth-first search tree (since the *dfs* realizations of *path* and *revpath* both do recursions on the first parameter). (In the above, Adj^{-1} denotes the inverse adjacency function such that $v \in Adj(u)$ if and only if $u \in Adj^{-1}(v)$.) Clearly, $\mathit{path}(u,v)$ if and only if $\mathit{revpath}(v,u)$.)

Alternatively, we could reverse the directions 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 adapting Algorithm (32) to reversed edge direction and applying the finite closure transformation (as we did in Algorithms (8) and (11)), we obtain

$$\begin{array}{l} \mathit{strong} \Leftarrow \mathbf{begin} \\ \qquad\qquad \mathit{visit2}[V] \leftarrow \mathbf{false}\ ; \\ \qquad\qquad \bigcup_{r \in V}(\mathbf{if}\ \mathit{visit2}[r]\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \{\mathit{scr}(r,r)\}) \\ \qquad\ \mathbf{end} \\ \\ \mathit{scr}(u,r) \Leftarrow \\ \quad \mathbf{begin} \\ \quad\ \mathit{visit2}[u] \leftarrow \mathbf{true}\ ; \\ \quad\ \{u\} \cup \bigcup_{w \in Adj^{-1}(u)}(\mathbf{if}\ \neg \mathit{visit2}[w] \wedge \mathit{path}(r,w)\ \mathbf{then}\ \mathit{scr}(w,r)\ \mathbf{else}\ \emptyset) \\ \quad \mathbf{end} \end{array} \tag{34}$$

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

A blind alley. It may appear that we could obtain an acceptable implementation of Algorithm (34) by replacing $\mathit{path}(r,w)$ with the test $w \in \mathit{dfs}(r)$ and using specialization to factor the *dfs* calculation out of *scr* into *strong*.

$$\begin{array}{l} \mathit{strong} \Leftarrow \mathbf{begin} \\ \qquad\qquad \mathit{visit2}[V] \leftarrow \mathbf{false}\ ; \\ \qquad\qquad \bigcup_{r \in V}\ (\mathbf{if}\ \mathit{visit2}[r]\ \mathbf{then}\ \emptyset \\ \qquad\qquad\qquad\qquad \mathbf{else\ begin} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{visit}[V] \leftarrow \mathbf{false}\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{scr}'(r,r,\mathit{dfs}(r))\} \\ \qquad\qquad\qquad\qquad\qquad\ \mathbf{end}) \\ \qquad\ \mathbf{end} \end{array} \tag{35}$$

$\mathit{scr}'(u, r, D) \Leftarrow$
begin
$\mathit{visit2}[u] \leftarrow$ **true**;
$\{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$(**if** $\neg \mathit{visit2}[w] \wedge w \in D$ **then** $\mathit{scr}'(w, r, D - \{w\})$)
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 $\mathit{dfs}(r)$ 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 algorithms for strongly-connected components.

Lemma 2. *Let G be a directed graph with a (forward) 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 $v \rightarrow w$, 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
$$\mathit{path}(r, w) \quad \text{if and only if} \quad w \succeq r.$$

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. On the basis of Section 2, we can accomplish this by checking *pre*. 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 $w \succeq r$.

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. This is distinct from the reverse forest constructed by *scr*.

forest $\Leftarrow$ **begin**
$pre[V] \leftarrow 0;\ p \leftarrow 0;\ e \leftarrow 0;$
for $r \in V$ **do** (**if** $pre[r] = 0$ **then** $[\![r \in Roots]\!]$ $\mathit{dfs}(r)$)
end

(36)

$\mathit{dfs}(u) \Leftarrow$ **begin**
$\quad pre[u] \leftarrow (p \leftarrow p+1);$
$\quad$ **for** $w \in Adj(u)$ **do**
$\quad\quad$ **if** $pre[w] = 0$ **then** $[\![u \xrightarrow{\mathrm{T}} w]\!]$ $\mathit{dfs}(w)$;
$\quad post[u] \leftarrow (e \leftarrow e+1)$
end

We note tree edges as they are found. In the final algorithm, an array *parent* is used to track the tree edges in order to refine Algorithm (39).

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 that 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 (36).

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

(37)

$\mathit{scr}(u,r) \Leftarrow$
begin
$\quad \mathit{visit2}[u] \leftarrow$ **true** ;
$\quad \{u\} \cup \bigcup_{w \in Adj^{-1}(u)}$(**if** $\neg \mathit{visit2}[w] \wedge (pre[r] < pre[w]) \wedge (post[r] > post[w])$
$\quad\quad$ **then** $\mathit{scr}(w,r)$)
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. We can use *forest* to collect a nonempty subset of the final set of roots, namely those vertices where each *dfs* is initiated until all of V is preorder numbered.

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

(38)

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 $\mathit{scr}(r, r)$ 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.

$$\begin{array}{l} \mathit{update}(S) \Leftarrow \\ \quad \bigcup_{v \in S} \left(\bigcup_{w \in Adj(v)} (\textbf{if } (\neg \mathit{visit2}(w) \wedge v \rightarrow w) \textbf{ then } \{w\} \textbf{ else } \emptyset) \right) \end{array} \tag{39}$$

Given a strongly-connected component S, $\mathit{update}(S)$ returns the set of strongly-connected component roots then are the direct descendants of vertices in S. The test $v \rightarrow w$ in this definition needs to be computed. This is accomplished by making tree edge relationships explicit in a new array parent. The array values are initialized in dfs. (See Algorithm (42) for details.)

An alternative approach to finding roots (used in [2] but not followed here) is to rely on a further consequence of the lemma: After zero or more strongly-connected components have been found, 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 near-final algorithm is obtained by merging two programs. The first, Algorithm (37) derived earlier, finds strongly-connected components, given a set of roots. The second, which follows directly from the lemma, is a closure definition of the set of roots.

$$R \;\Leftarrow_{\emptyset}\; \mathit{forest} \cup \bigcup_{r \in R} \mathit{update}(\mathit{scr}(r, r)) \tag{40}$$

A merge of strong with the definition of R gives:

$$\begin{array}{l} \mathit{strong}(\mathit{forest}) \\ \\ \mathit{visit2}[V] \leftarrow \textbf{false} \\ \\ \mathit{strong}(R) \Leftarrow \\ \quad \textbf{if } R = \emptyset \textbf{ then } \emptyset \\ \quad\quad \textbf{else let } r = \textbf{choose } R \textbf{ in} \\ \quad\quad\quad \textbf{let } S = \mathit{scr}(r, r) \textbf{ in} \\ \quad\quad\quad\quad \{S\} \cup \mathit{strong}(\mathit{update}(S) \cup R - \{r\}) \end{array} \tag{41}$$

(The **choose** operation picks an arbitrary element of 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.

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

var $p, e, pre[V], post[V], \mathit{visit2}[V], \mathit{parent}[V]$;

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

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

$\mathit{visit2}[V] \leftarrow$ **false** ;

$\mathit{strong}(R) \Leftarrow$ (42)
$\quad$ **if** $R = \emptyset$ **then** $\emptyset$
$\quad\quad$ **else let** $r =$ **choose** R **in**
$\quad\quad\quad$ **let** $S = \mathit{scr}(r, r)$ **in** $\quad [\![S$ is a strong component$]\!]$
$\quad\quad\quad\quad \{S\} \cup \mathit{strong}(\mathit{update}(S) \cup R - \{r\})$

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

$\mathit{update}(S) \Leftarrow$
$\quad \bigcup_{v \in S}(\bigcup_{w \in Adj(v)}$(**if** $\neg \mathit{visit2}(w)\ \wedge\ \mathit{parent}[w] = v$
$\quad\quad\quad\quad\quad\quad\quad\quad$ **then** $\{w\}$
$\quad\quad\quad\quad\quad\quad\quad\quad$ **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 of 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 complement 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.

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 other ways. At this point, however, the structure of the linear-time algorithm is most clearly apparent, so we conclude the derivation here.

4 Biconnected Components

Let $G = (V, E)$ be an undirected connected graph with no self-loops (i.e., edges of the form $\langle u, u \rangle$). By convention, sets and relations involving undirected graph edges are assumed to be symmetrically closed. 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.

$$\begin{array}{l} path_a(u,v) \quad \Leftarrow_{\mathbf{false}} \\ \quad u = v \ \ \mathbf{or} \ \ (\exists w \in Adj(u))(w = v \ \mathbf{or} \ (w \neq a \ \mathbf{and} \ path_a(w,v))) \end{array} \tag{43}$$

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

$$path_a(u,v) \qquad \text{if and only if} \qquad path_a(v,u).$$

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

$$[\![u \neq a \wedge v \neq a]\!]\ \mathit{path}_a(u,v) \quad \Leftarrow_{\mathbf{false}} \quad (44)$$
$$u = v \ \mathbf{or}\ (\exists w \in \mathit{Adj}(u))(w \neq a \ \mathbf{and}\ [\![w \neq a \wedge v \neq a]\!]\ \mathit{path}_a(w,v))$$

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,

$$\mathit{bc}(\langle u,v\rangle) \quad \Leftarrow_{\emptyset} \quad \{\langle u,v\rangle\} \cup \left(\bigcup_{\substack{\langle v,w\rangle \in E \\ \mathit{path}_v(u,w)}} \mathit{bc}(\langle v,w\rangle) \right). \quad (45)$$

(The specialized definition of *path* will suffice in this context.) Observe that by our various symmetry assumptions, $\mathit{bc}(\langle u,v\rangle) = \mathit{bc}(\langle ,uv\rangle)$.

The set *bcomps* contains the biconnected components of G.

$$\mathit{bcomps} \Leftarrow \bigcup_{\langle u,v\rangle \in E} \{\mathit{bc}(\langle u,v\rangle)\} \quad (46)$$

Specialization to depth-first search. Our initial goal is to obtain an efficient version of *bc* that is not a closure definition.

Application of the finite closure transformation induces a depth-first search forest, as shown in Section 2. Our initial step is to specialize the definition of *bc* in order to enable it to be merged with the depth-first search algorithm (i.e., the recursions of *bc* and *dfs* will be made simultaneous). We start with the assumption that a depth-first-search forest *already* exists (and we can cheaply test edge type). To enable the merge with *dfs*, we specialize the definition of *bc* to traverse tree edges, and in the depth-first search order. By assuming existence of the forest, we can exploit information about edge type.

Recall from Section 2 that an undirected graph edge can be either a tree edge or a non-tree edge, and, when it is given direction (in a search), it can be one of four kinds of edges: a tree edge, a reverse tree edge, a forward frond, or a reverse frond. The first step is to introduce a case analysis into the body of the union to distinguish the four cases for an edge $\langle v,w\rangle$. Initially all branches of the case analysis are identical; our intent is to specialize each according to its particular condition.

$$[\![u \xrightarrow{\mathrm{T}} v]\!]\mathit{bc}(\langle u,v\rangle) \quad \Leftarrow_{\emptyset} \quad (47)$$
$$\{\langle u,v\rangle\} \cup$$
$$\bigcup_{\langle v,w\rangle \in E} (\mathbf{if}\ v \xrightarrow{\mathrm{T}} w\ \mathbf{then}\ (\mathbf{if}\ \mathit{path}_v(u,w)\ \mathbf{then}\ \mathit{bc}(\langle v,w\rangle)\ \mathbf{else}\ \emptyset)$$
$$\mathbf{else\ if}\ u = w\ \mathbf{then}(\mathbf{if}\ \mathit{path}_v(u,w)\ \mathbf{then}\ \mathit{bc}(\langle v,w\rangle)\ \mathbf{else}\ \emptyset)$$
$$\mathbf{else\ if}\ v \dashrightarrow w\ \mathbf{then}(\mathbf{if}\ \mathit{path}_v(u,w)\ \mathbf{then}\ \mathit{bc}(\langle v,w\rangle)\ \mathbf{else}\ \emptyset)$$
$$\mathbf{else}\ [\![w \dashrightarrow v]\!]\ \mathbf{then}(\mathbf{if}\ \mathit{path}_v(u,w)\ \mathbf{then}\ \mathit{bc}(\langle v,w\rangle)\ \mathbf{else}\ \emptyset)\ )$$

We have distributed the path test into the four cases of this closure definition, each of which we now consider individually, with the goal of making this a tightly recursive expression procedure (i.e., meeting the precondition that the edge $\langle v, w\rangle$ is a tree edge). The first case is easy. If $\langle v, w\rangle$ is a tree edge, then the recursive call to bc is already in the specialized form.

In the second case $u = w$. In this case w is the parent of v and so $\mathit{path}_v(u, w)$ is trivially true. We must compute $\mathit{bc}(\langle v, w\rangle)$ or, replacing w by u, $\mathit{bc}(\langle v, u\rangle)$. This is a reverse tree edge, and by symmetry it necessarily leads to redundant visits, enabling us to replace the call by $\emptyset$. To simplify notation, we do a trivial specialization so bc is passed two adjacent vertices, rather than the edge between them.

$$
\begin{array}{l}
[\![u \xrightarrow{\mathrm{T}} v]\!]\mathit{bc}(u, v) \quad \Leftarrow_{\emptyset} \\
\quad \{\langle u, v\rangle\} \cup \\
\quad\quad \bigcup_{\langle v,w\rangle \in E}(\textbf{if } v \xrightarrow{\mathrm{T}} w \textbf{ then } (\textbf{if } \mathit{path}_v(u, w) \textbf{ then } \mathit{bc}(v, w) \textbf{ else } \emptyset) \\
\qquad\qquad\qquad \textbf{else if } u = w \textbf{ then } \emptyset \\
\qquad\qquad\qquad \textbf{else if } v \dashrightarrow w \textbf{ then } (\textbf{if } \mathit{path}_v(u, w) \textbf{ then } \mathit{bc}(v, w) \textbf{ else } \emptyset) \\
\qquad\qquad\qquad \textbf{else } [\![w \dashrightarrow v]\!] \textbf{ then } (\textbf{if } \mathit{path}_v(u, w) \textbf{ then } \mathit{bc}(v, w) \textbf{ else } \emptyset)\ )
\end{array}
\tag{48}
$$

We next consider the third case, of a reverse frond $v \dashrightarrow w$. In this case, v is a direct descendent of u and w is an ancestor of u, and so $\mathit{path}_v(u, w)$ is always true. We must therefore include $\mathit{bc}(v, w)$. Now $\langle v, w\rangle$ is not a tree edge, so this recursive call will not be in the specialized form. We therefore unfold 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 assumption of depth-first traversal order) replace *all* the recursive bc calls from w by $\emptyset$ and retain only the single reverse frond $\langle v, w\rangle$. Observe that this implies all reverse fronds from a vertex are collected at that vertex.

The final case, $w \dashrightarrow v$, 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 $v \xrightarrow{\mathrm{T}} t$ 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 preceding 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.

$$
\begin{array}{l}
[\![u \xrightarrow{\mathrm{T}} v]\!]\mathit{bc}(u, v) \Leftarrow \\
\quad \{\langle u, v\rangle\} \cup \\
\quad\quad \bigcup_{\langle v,w\rangle \in E}(\textbf{if } v \rightarrow w \textbf{ then } (\textbf{if } \mathit{path}_v(u, w) \textbf{ then } [\![u \xrightarrow{\mathrm{T}} w]\!]\ \mathit{bc}(v, w) \textbf{ else } \emptyset) \\
\qquad\qquad\qquad \textbf{else if } u = w \textbf{ then } \emptyset \\
\qquad\qquad\qquad \textbf{else if } v \dashrightarrow w \textbf{ then } \{\langle v, w\rangle\} \\
\qquad\qquad\qquad \textbf{else } [\![w \dashrightarrow v]\!]\ \emptyset\ )
\end{array}
\tag{49}
$$

Because the remaining call to bc is directly recursive, and the call always follows *tree* edges, this does not need to be a closure definition.

At this point there are two principal additional steps in improving the algorithm for biconnected components. First, the definition of *bcomps* is improved to avoid collecting redundant components—this is done by introducing the abstraction of articulation edges (in the next section). Second the *path* test in *bc* is made more efficient through an ancestry test (in a later section).

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

Lemma 3. *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 $u \xrightarrow{\mathrm{T}} v$ and $v \xrightarrow{\mathrm{T}} w$, then

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

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 us inductively assume we can *a priori* compute the set of articulation edges, which we call *aedges*.

$$\mathit{bcomps} \Leftarrow \bigcup_{\langle u,v\rangle \in \mathit{aedges}} [\![u \xrightarrow{\mathrm{T}} v]\!]\ \{\mathit{bc}(u,v)\} \tag{50}$$

Every articulation edge is a tree edge, and the set of biconnected components is a partition of the set of edges. Therefore, the specialized version of *bc* can be applied.

Collecting biconnected components. Because articulation edges are tree edges, we attempt to collect them in a single depth-first search. Again we assume that the tree edges are already so classified and, in addition, that the predicate $\mathit{root}(r)$ holds when 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*.

$$\mathit{aedges} \Leftarrow \bigcup_{\substack{\mathit{root}(r)\\ r \xrightarrow{\mathrm{T}} s}} (\{\langle r,s\rangle\} \cup \mathit{ae}(r,s)) \tag{51}$$

$$[\![u \xrightarrow{\mathrm{T}} v]\!]\mathit{ae}(u,v) \Leftarrow \bigcup_{\substack{w \in \mathit{Adj}(v)\\ v \xrightarrow{\mathrm{T}} w}} (\textbf{if}\ \ \mathit{aedge}(v,w)\ \textbf{then}\ \{\langle v,w\rangle\} \cup \mathit{ae}(v,w)\ \ \textbf{else}\ \mathit{ae}(v,w))$$

The second consequence of the lemma enables replacement of the condition '$\mathit{aedge}(v,w)$' by the condition '$\neg \mathit{path}_v(u,w)$' because we know $u \xrightarrow{\mathrm{T}} v$ and $v \xrightarrow{\mathrm{T}} w$.

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. Note that the function *ae* now returns a set of biconnected components.

$$\mathit{bcomps} \Leftarrow \bigcup_{\substack{\mathit{root}(r) \\ r \xrightarrow{\mathrm{T}} s}} (\{\langle r,s\rangle\} \cup \mathit{ae}(r,s)) \tag{52}$$

$$[\![u \xrightarrow{\mathrm{T}} v]\!]\mathit{ae}(u,v) \Leftarrow \bigcup_{\substack{w \in \mathit{Adj}(v) \\ v \xrightarrow{\mathrm{T}} w}} (\textbf{if } \mathit{aedge}(v,w) \textbf{ then } \{\mathit{bc}(v,w)\} \cup \mathit{ae}(v,w) \textbf{ else } \mathit{ae}(v,w))$$

Merging. It is clear from the structure of the *bcomps* 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

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

The result of this program is 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.

In the example below, we introduce notation for the simultaneous accumulation of set. 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 denotes the same result as

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

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

After substitution and simplification of Algorithms (49) and (52), we obtain

$$\mathit{bcomps} \Leftarrow \bigcup_{\substack{\mathit{root}(r) \\ r \xrightarrow{\mathrm{T}} s}} (\{B\} \cup A \ \textbf{ where } \langle B, A\rangle = [\![r \xrightarrow{\mathrm{T}} s]\!]\langle \mathit{bc}(r,s), \mathit{ae}(r,s)\rangle)$$

$$
\begin{array}{l}
[\![u \xrightarrow{\mathrm{T}} v]\!]\ \langle \mathit{bc}(u,v), \mathit{ae}(u,v)\rangle \Leftarrow \\
\quad \langle \{\langle u,v\rangle\} \cup B, A\rangle \\
\qquad \textbf{where} \\
\qquad \langle B, A\rangle = \langle \bigcup, \bigcup\rangle_{w \in Adj(v)} \\
\qquad\qquad\qquad (\textbf{if } v \xrightarrow{\mathrm{T}} w \textbf{ then} \\
\qquad\qquad\qquad\quad (\ \textbf{if } \neg \mathit{path}_v(u,w) \\
\qquad\qquad\qquad\qquad\quad \textbf{then } [\![\mathit{aedge}(v,w)]\!]\ \langle \emptyset, \{B'\} \cup A'\rangle \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \langle B', A'\rangle\ ) \\
\qquad\qquad\qquad\qquad \textbf{where } \langle B', A'\rangle = [\![v \xrightarrow{\mathrm{T}} w]\!]\ \langle \mathit{bc}(v,w), \mathit{ae}(v,w)\rangle \\
\qquad\qquad\quad \textbf{else if } u = w \textbf{ then } \langle \emptyset, \emptyset\rangle \\
\qquad\qquad\quad \textbf{else if } v \dashrightarrow w \textbf{ then } \langle \{\langle v,w\rangle\}, \emptyset\rangle \\
\qquad\qquad\quad \textbf{else } \langle \emptyset, \emptyset\rangle\ ) .
\end{array}
\tag{53}
$$

Note that the only appearance on the right-hand side of ae and bc are in a context where the precondition holds. We rename the pair and its precondition to a simple name, ba.

$$
\mathit{bcomps} \Leftarrow \bigcup_{\substack{\mathit{root}(r) \\ r \xrightarrow{\mathrm{T}} s}} (\{B\} \cup A \ \ \textbf{where } \langle B, A\rangle = \mathit{ba}(r,s))
$$

$$
\begin{array}{l}
\mathit{ba}(u,v) \Leftarrow \\
\quad \langle \{\langle u,v\rangle\} \cup B, A, \rangle \\
\qquad \textbf{where } \langle B, A\rangle = \langle \bigcup, \bigcup\rangle_{w \in Adj(v)} \\
\qquad\qquad\qquad (\textbf{if } v \xrightarrow{\mathrm{T}} w \textbf{ then} \\
\qquad\qquad\qquad\quad (\ \textbf{if } \neg \mathit{path}_v(u,w) \\
\qquad\qquad\qquad\qquad\quad \textbf{then } [\![\mathit{aedge}(v,w)]\!]\ \langle \emptyset, \{B'\} \cup A'\rangle \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \langle B', A'\rangle\ ) \\
\qquad\qquad\qquad\qquad \textbf{where } \langle B', A'\rangle = \mathit{ba}(v,w)\rangle \\
\qquad\qquad\quad \textbf{else if } u = w \textbf{ then } \langle \emptyset, \emptyset\rangle \\
\qquad\qquad\quad \textbf{else if } v \dashrightarrow w \textbf{ then } \langle \{\langle v,w\rangle\}, \emptyset\rangle \\
\qquad\qquad\quad \textbf{else } \langle \emptyset, \emptyset\rangle\ ) .
\end{array}
\tag{54}
$$

It now remains to derive a method for efficiently testing $\neg \mathit{path}_v(u,w)$.

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

Lemma 4. *Let G be an undirected graph with depth-first search forest F and let $u \xrightarrow{T} v$ and $v \xrightarrow{T} w$ be edges in F. Then*

$$\begin{aligned} \mathit{path}_v(u,w) &\equiv (\exists s,t)(u \succeq t \wedge t \Leftrightarrow s \wedge s \succeq w) \\ &\equiv (\exists s,t)(v \succ t \wedge t \Leftrightarrow s \wedge s \succeq w). \end{aligned}$$

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 $t \Leftrightarrow s$ 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

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

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 t of the set is either an ancestor or a descendent of w. Furthermore, all ancestors of w are themselves pairwise comparable since they are on the tree path from the root to w. Since v is an ancestor of w and since we are only interested in t that are proper ancestors of v, descendants of w can be ignored during search. We implement this improvement by means of a simple modification to the above specification.

$$\mathit{low}(w) \Leftarrow \min_{\succ}(\{w\} \cup \{t \mid (\exists s)\ s \dashrightarrow t \wedge s \succeq w\})\,. \tag{55}$$

The lemma (4.2) immediately implies: $v \succ \mathit{low}(w)$ if and only if $\mathit{path}_v(u,w)$ (because $u \xrightarrow{T} v$). In other words, $v \xrightarrow{T} w$ 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.

$$\begin{aligned} &\mathit{low}(w) \Leftarrow \min_{\succ}(\{w\} \cup \mathit{lowset}(w)) \\ &\mathit{lowset}(w) \Leftarrow \{t \mid (\exists s) s \dashrightarrow t \wedge s \succeq w\} \end{aligned} \tag{56}$$

Lowset computation. We observe first that $\{s | s \succeq w\}$ is exactly $\mathit{dfs}(w)$. Modifying Algorithm (23) slightly,

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

$$
\begin{array}{l}
\mathit{dfs}(u,v) \Leftarrow \\
\quad \textbf{begin} \\
\qquad \mathit{pre}[v] \leftarrow p \leftarrow p+1 \ ; \\
\qquad \{v\} \cup \bigcup_{w \in Adj(u)} (\textbf{if } \mathit{pre}[w] = 0 \textbf{ then } [\![v \xrightarrow{\mathrm{T}} w]\!] \ \mathit{dfs}(w,v) \\
\qquad\qquad\qquad\qquad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad \textbf{else if } \mathit{pre}[v] > \mathit{pre}[w] \textbf{ then } [\![v \dashrightarrow w]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad \textbf{else } [\![w \dashrightarrow v]\!] \ \emptyset) \\
\quad \textbf{end}
\end{array}
\tag{57}
$$

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

$$
[\![u \xrightarrow{\mathrm{T}} v]\!] \ \mathit{lowset}(w) \Leftarrow \{t \mid (\exists s) s \dashrightarrow t \wedge s \in \mathit{dfs}(u,v)\}
\tag{58}
$$

As before, we assume $u \xrightarrow{\mathrm{T}} v$. We also assume that a special value Λ is passed for u when v is a root.

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

$$
\begin{array}{l}
[\![u \xrightarrow{\mathrm{T}} v]\!] \{t \mid (\exists s) s \dashrightarrow t \wedge s \in \mathit{dfs}(u,v)\} \Leftarrow \\
\quad \textbf{begin} \\
\qquad \mathit{pre}[v] \leftarrow p \leftarrow p+1 \ ; \\
\qquad \{t \mid (\exists s) s \dashrightarrow t \wedge s \in \{v\}\} \\
\qquad\quad \cup \{t \mid (\exists s) s \dashrightarrow t \ \wedge \ s \in \bigcup_{w \in Adj(u)} \\
\qquad\qquad\qquad\qquad (\textbf{if } \mathit{pre}[w] = 0 \textbf{ then } [\![v \xrightarrow{\mathrm{T}} w]\!] \ \mathit{dfs}(v,w) \\
\qquad\qquad\qquad\qquad\quad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad\quad \textbf{else if } \mathit{pre}[v] > \mathit{pre}[w] \textbf{ then } [\![v \dashrightarrow w]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad\quad \textbf{else } [\![w \dashrightarrow v]\!] \ \emptyset)\} \\
\quad \textbf{end}.
\end{array}
\tag{59}
$$

We distribute the set comprehension into the union and conditional and simplify. In particular, three instances of $\{t \mid (\exists s) s \dashrightarrow t \wedge s \in \emptyset\}$ simplify to $\emptyset$.

$$
\begin{array}{l}
[\![u \xrightarrow{\mathrm{T}} v]\!] \{t \mid (\exists s) s \dashrightarrow t \wedge s \in \mathit{dfs}(u,v)\} \Leftarrow \\
\quad \textbf{begin} \\
\qquad \mathit{pre}[v] \leftarrow p \leftarrow p+1 \ ; \\
\qquad \{t \mid v \dashrightarrow t\} \\
\qquad\quad \cup \bigcup_{w \in Adj(u)} (\textbf{if } \mathit{pre}[w] = 0 \\
\qquad\qquad\qquad\qquad \textbf{then } [\![v \xrightarrow{\mathrm{T}} w]\!] \ \{t \mid (\exists s) s \dashrightarrow t \wedge s \in \mathit{dfs}(v,w)\} \\
\qquad\qquad\qquad\qquad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad \textbf{else if } \mathit{pre}[v] > \mathit{pre}[w] \textbf{ then } [\![v \dashrightarrow w]\!] \ \emptyset \\
\qquad\qquad\qquad\qquad \textbf{else } [\![w \dashrightarrow v]\!] \ \emptyset) \\
\quad \textbf{end}.
\end{array}
\tag{60}
$$

The one instance of *dfs* is contained in a recursive instance of the expression procedure, so we can rename.

$$
\begin{array}{l}
\mathit{lowset}(u,v)\} \Leftarrow \\
\quad \textbf{begin} \\
\quad\quad pre[v] \leftarrow p \leftarrow p+1\ ; \\
\quad\quad \{t \mid v \dot{\to} t\} \\
\quad\quad\quad \cup \bigcup_{w \in Adj(u)} (\textbf{if } pre[w] = 0 \textbf{ then } [\![v \xrightarrow{\mathrm{T}} w]\!]\ \mathit{lowset}(v,w) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!]\ \emptyset \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } pre[v] > pre[w] \textbf{ then } [\![v \dot{\to} w]\!]\ \emptyset \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } [\![w \dot{\to} v]\!]\ \emptyset) \\
\quad \textbf{end}.
\end{array}
\tag{61}
$$

We can eliminate the outermost union because of this fact:

$$\{t | v \dot{\to} t\} \quad = \quad \bigcup_{w \in Adj(v)} (\textbf{if } v \dot{\to} w \textbf{ then } \{w\} \textbf{ else } \emptyset),$$

We combine this into the third arm of the condition in Algorithm (52).

$$
\begin{array}{l}
\mathit{lowset}(u,v)\} \Leftarrow \\
\quad \textbf{begin} \\
\quad\quad pre[v] \leftarrow p \leftarrow p+1\ ; \\
\quad\quad \bigcup_{w \in Adj(u)} (\textbf{if } pre[w] = 0 \textbf{ then } [\![v \xrightarrow{\mathrm{T}} w]\!]\ \mathit{lowset}(v,w) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!]\ \emptyset \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } pre[v] > pre[w] \textbf{ then } [\![v \dot{\to} w]\!]\ \{w\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } [\![w \dot{\to} v]\!]\ \emptyset) \\
\quad \textbf{end}
\end{array}
\tag{62}
$$

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

$$[\![u \xrightarrow{\mathrm{T}} v]\!]\ \mathit{low}(u,v) \ \Leftarrow\ \min_{\succ}(\{v\} \cup \mathit{lowset}(u,v)).$$

We obtain,

$$
\begin{array}{l}
\mathit{low}(u,v) \ \Leftarrow \\
\quad\quad \textbf{begin} \\
\quad\quad\quad pre[v] \leftarrow p \leftarrow p+1 \\
\quad\quad\quad \min_{w \in Adj(v)} (\min_{\succ}(v, (\textbf{if } pre[w] = 0 \textbf{ then } [\![v \xrightarrow{\mathrm{T}} w]\!]\ \mathit{low}(v,w) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } w = u \textbf{ then } [\![w \xrightarrow{\mathrm{T}} v]\!]\ \infty \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } pre[v] > pre[w] \textbf{ then } [\![v \dot{\to} w]\!]\ w \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } [\![w \dot{\to} v]\!]\ \infty))\ ) \\
\quad\quad \textbf{end}
\end{array}
\tag{63}
$$

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

```
low(u, v) ⇐
    begin
      pre[v] ← p ← p + 1                                                    (64)
      min_{w∈Adj(v)}(if pre[w] = 0 then [[v →^T w]] min_≻(v, low(v, w))
                     else if w = u then [[w →^T v]] v
                     else if pre[v] > pre[w] then [[v ⇸ w]] min(v, w)
                     else [[w ⇸ v]] v)
    end
```

Using preorder numbers. Recall that according to the lemma, if $\mathit{low}(v, w)$ is a descendent of v, then $v \xrightarrow{T} w$ 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 straightforward transformation, we obtain:

```
low(u, v) ⇐
    begin
      m ← pre[v] ← p ← p + 1
      for w ∈ Adj(v) do
          if pre[w] = 0 then let ℓ = low(v, w) in
                                    begin [[v →^T w]] trigger
                                      m ← min(m, ℓ);
                                      (if ℓ ≥ pre[v] then [[aedge(v, w)]] )
                                    end
             else if w = u then [[w →^T v]]
             else if pre[v] > pre[w] then [[v ⇸ w]] m ← min(m, pre[w])
             else [[w ⇸ v]]
    end.
```

We have added 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 revisit the *bcomps* algorithm derived earlier. That algorithm simultaneously collects the set of biconnected components and the set of edges in the current component. We merge algorithm 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, requires time linear in the number of vertices and edges. We start by substituting to obtain and expression procedure for the expression $\langle \mathit{ba}(u, v), \mathit{low}(u, v)\rangle$. We then combine two conditional cases and rename.

```
bcomps ⇐
    begin
        pre[V] ← 0; p ← 0;
        ⋃_{root(r)} ({B} ∪ A where ⟨B, A, ℓ⟩ = balow(r, s))
          r →^T s
    end

balow(u, v) ⇐
    begin var m;
      m ← pre[v] ← p ← p + 1;                                          (65)
      ⟨{⟨u, v⟩} ∪ B, A, m⟩
        where ⟨B, A⟩ = ⟨⋃,⋃⟩_{w∈Adj(v)}
                        (if pre[w] = 0
                            then (let ⟨B', A', ℓ⟩ = balow(v, w) in
                                    if ℓ ≥ pre[v]
                                        then begin
                                                [[B' is component]]
                                                m ← min(m, ℓ);
                                                ⟨∅, {B'} ∪ A'⟩
                                             end
                                        else ⟨B', A⟩)
                          else if pre[w] < pre[v] ∧ w ≠ u
                              then begin
                                      m ← min(m, pre[w]);
                                      ⟨{⟨v, w⟩}, ∅⟩
                                   end
                          else ⟨∅, ∅⟩)
    end.
```

This algorithm returns a triple instead of two nested pairs. Recall that a biconnected component is a maximal set of edges that contains an articulation edge. In the triple $\langle B, A, \ell\rangle$, B is the set of edges thus far in the current biconnected component, A is the set of components collected so far, and ℓ is the current minimum *low* value. Note that at the top level ℓ is not used in the triple $\langle B, A, \ell\rangle$. We now have a linear-time algorithm for computing the set of biconnected components in an undirected graph.

The biconnectivity algorithm. In many presentations of the biconnectivity algorithm components are emitted as they as found, rather than collected explicitly (as in the second component of the result of *balow*, above). This traditional presentation can be derived using transformations that introduce operations on global state and eliminate corresponding operations on explicit results. This corresponds to the implicit/explicit distinction raised in Section 2 above.

This transformation, strictly speaking, is not of Algorithm (23) but rather an alternative choice for state management in doing finite closure for *bcomps*

(e.g., Algorithm (53)). With respect to Algorithm (23), 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.

```
bcomps ⇐
    begin
        pre[V] ← 0  //  p ← 0;
        for r ∈ V do (if pre[r] = 0 then balow(Λ, r))
    end

balow(u, v) ⇐
    begin var m;
        m ← pre[v] ← p ← p + 1;
        ⟨B, m⟩
          where B = ⋃_{w∈Adj(v)}
                        (if pre[w] = 0
                            then (let ⟨B', ℓ⟩ = balow(v, w) in
                                      let B'' = B' ∪ {⟨u, v⟩} in
                                        if ℓ ≥ pre[v]                    (66)
                                          then begin
                                                   [[B'' is a component]]
                                                   m ← min(m, ℓ);
                                                   ∅
                                               end
                                          else B'')
                            else if pre[w] < pre[v]  ∧  w ≠ u
                                then begin
                                         m ← min(m, pre[w]);
                                         {⟨v, w⟩}
                                     end
                            else ∅)
    end
```

(We have in addition, "rotated" the outermost union to the callee; 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.

A final transformation A similar transformation can be carried out to eliminate the first result of *balow*. In the prior example, biconnected components were added to the set as they were found. In this case the changes to state corresponding to accumulation of the edge set have a stack-like discipline. This is a result of our transformation of merging *bc* and *ae*. The difficulty 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.

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] ← 0  //  p ← 0  //  stack ← empty;
        for r ∈ V do (if pre[r] = 0 then balow(Λ, r))
    end

balow(u, v) ⇐
    begin var m;
        m ← pre[v] ← p ← p + 1;
        for w ∈ Adj(v) do
          if pre[w] = 0
            then begin
                    Push ⟨v, w⟩;
                    let ℓ = balow(v, w) in
                      if ℓ ≥ pre[v]                                  (67)
                        then begin
                                m ← min(m, ℓ);
                                Pop to ⟨v, w⟩
                                [[Popped edges are a component ]]
                             end
            else if pre[w] < pre[v]  ∧  w ≠ u
              then begin
                      m ← min(m, pre[w]);
                      Push ⟨v, w⟩
                    end;
        m
    end
```

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

These derivations are intended as a step towards developing an approach to the explication, proof, and possibly the adaptation and design of complex algorithms. The underlying hypothesis is that derivation can often be more revealing of specific design steps and their rationale than the usual textbook presentation of a complete algorithm and proof. This hypothesis has been explored from

time to time over the past two decades, and both in textbooks and in research papers. The derivational approach to algorithms provides a (potentially revisionist) glimpse at the evolutionary process by which algorithms can be created from simple components. This is potentially most useful to an algorithm designer when faced with a new computational problem or architecture, providing a more solid basis for moving forward than structural analogy with existing algorithms. Work such as reported here on sequential algorithms led to later work in the 1980s and 1990s on the derivation of parallel algorithms such as [14] and [16].

There are challenges associated with the transformational approach to presentation. The manipulations are ultimately formal, and it continues to be difficult to suppress detail and make large steps. In a presentation (such as the foregoing), this can inhibit the understanding of the essential structure and components of the solution. For example, the biconnectivity algorithm results from merging three loops. Each is informed by a particular deep fact from graph theory—respectively, depth-first search, articulation edges, and ancestry testing using preorder numbers. But each is structured to enable the mering to be done.

Of course, we are still far from automating the heuristic side of the derivation process. In fact, we argue that at this point our efforts are still best directed at discovering and exercising useful transformation techniques, developing foundations for establishing their soundness, and developing tools for *interactive* program development that can make appropriate use of outside domain-specific knowledge. This capability still lacks in present refactoring tools. In a specific problem domain, such as graph algorithms, certain facts and fundamental algorithms are frequently reused. In our derivations, for example, the depth-first search algorithms were repeatedly used as templates for the development of related algorithms.

By treating these program derivations as structured data objects in a program development system, program *modification*, even when it appears pervasive at code level (as in the last three derivation steps above) might be carried out by making relatively small changes at the appropriate places in the derivation structure. This is because a derivation provides a way to make certain design decisions explicit that are not apparent or even easily deduced when only the final code is available.

Acknowledgments. This paper is in honor of Zohar Manna, his technical contributions, and his inspiration. We appreciate thoughtful inputs (over a period of many years) from Margaret Beard, Greg Harris, Doreen Yen, Jerry Goldin, N.S.Sridharan, David Barstow, and Phil Wadler.

References

1. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.
3. D. Barstow. The roles of knowledge and deduction in algorithm design. Research Report 178, Yale University, April 1980.
4. F. Bauer et al. Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming*, 1:73–114, 1981.
5. R. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, 1980.
6. R. Burtall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
7. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases.* Plenum, 1978.
8. K. Clark and J. Darlington. Algorithm classification through synthesis. *Computer Journal*, 23(1), 1980.
9. S. Dietzen and W. Scherlis. Analogy and program development. In J. Boudreaux, editor, *The Role of Language in Problem Solving II.* North Holland, 1987.
10. C. Green and D. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10:241, 1978.
11. Z. Manna and R. Waldinger. Synthesis: dreams → programs. *IEEE Transactions on Software Engineering*, SE-5(4), July 1979.
12. Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.
13. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.
14. J. H. Reif. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, 1993.
15. R. Reiter. On closed world data bases. In Gallaire and Minker [7].
16. R. W. Robert Paige, John Reif. *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publishers, 1993.
17. D. Sands. Higher-order expression procedures. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 178–189. ACM Press, 1995.
18. W. Scherlis. *Expression procedures and program derivation.* PhD thesis, Stanford University, 1980.
19. W. Scherlis. Program improvement by internal specialization. In *Eighth Symposium on Principles of Programming Languages*, pages 41–49, 1981.
20. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
21. R. Tarjan. Testing flow graph reducibility. *Fifth ACM Symposium on the Theory of Computing*, pages 96–107, 1973.
22. M. Tullsen and P. Hudak. Shifting expression procedures into reverse. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 95–104, 1999.
23. M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.