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 $G = (V, \text{Adj})$ be a directed graph with vertices $V$ and adjacency-set function $\text{Adj} : V \rightarrow \mathcal{P}(V)$. We define the predicate $\text{path}(u, v)$ 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:

$$\text{path}(u, v) \equiv \text{false} \quad (u = v) \text{ or } (\exists w \in \text{Adj}(u))\text{path}(w, v)$$

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 $\text{false}$, and the finite set is the two truth values. Intuitively, whenever a “recursive call” to $\text{path}(u, v)$ is made for a pair $(u, v)$ that has already been calculated or is already being calculated, then the identity $\text{false}$ should be the result value. Since the set of vertices is finite, this assures termination of all evaluations of $\text{path}$. (Details on the semantics of closure definitions are in the next section.)