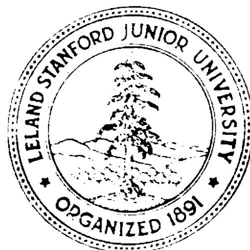
FAST ALGORITHMS FOR SOLVING PATH PROBLEMS

by

Robert Endre Tarjan

STAN-CS-79-734
April 1979

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



Fast Algorithms for Solving Path Problems

Robert Endre Tarjan*/

Computer Science Department
Stanford University
Stanford, California 94305

April, 1979

Abstract.

Let $G = (V,E)$ be a directed graph with a distinguished source vertex s. The single-source path expression problem is to find, for each vertex v, a regular expression $P(s,v)$ which represents the set of all paths in G from s to v. A solution to this problem can be used to solve shortest path problems, solve sparse systems of linear equations, and carry out global flow analysis [30]. We describe a method to compute path expressions by dividing G into components, computing path expressions on the components by Gaussian elimination, and combining the solutions. This method requires $O(m\,\alpha(m,n))$ time on a reducible flow graph, where n is the number of vertices in G, m is the number of edges in G, and α is a functional inverse of Ackermann's function. The method makes use of an algorithm for evaluating functions defined on paths in trees [9,29]. A simplified version of the algorithm, which runs in $O(m \log n)$ time on reducible flow graphs, is quite easy to implement and efficient in practice.

CR Categories: 4.12, 4.34, 5.14, 5.22, 5.25, 5.32.

Keywords: Ackermann's function, code optimization, compiling, dominators, Gaussian elimination, global flow analysis, graph algorithm, linear algebra, path compression, path expression, path problem, path sequence, reducible flow graph, regular expression, shortest path, sparse matrix.

*/ This research was partially supported by the National Science Foundation under grant MCS75-22870-A02, by the Office of Naval Research under contracts NR044-402 and N00014-76-C-0688, by the IBM Corporation, and by a Guggenheim Fellowship. Reproduction in whole or in part is permitted for any purpose of the United States government.

1. Introduction.

The techniques of Gaussian and Gauss-Jordan elimination, originally devised to solve systems of equations over the real numbers, have been repeatedly rediscovered and applied to other problems. These include shortest path problems [6,10,16], path-finding problems [4], global flow analysis [2,12,13,23], and conversion of finite automata to regular expressions [18]. The most fundamental of these problems is the (single source) path expression problem: Given a graph $G = (V,E)$ and a distinguished source vertex s, find a regular expression $P(s,v)$ for each vertex v which represents all paths from s to v in G. By reinterpreting the $\cup$, $\cdot$, and $*$ operations used to construct regular expressions, we can use a solution to the single-source path expression problem to solve other kinds of path problems, including those mentioned above [30]. We thus obtain a general-purpose algorithm for solving any path problem on a given graph.

This paper describes a decomposition method for computing path expressions. The method divides the graph G into components based upon the dominator tree of G, computes a path expression for each component by Gaussian elimination, and combines the solutions using an algorithm for evaluating functions defined on trees [9,29]. The algorithm requires $O(m\,\alpha(m,n))$ time plus time to compute path expressions within the components, where n is the number of vertices in G, m is the number of edges in G, and α is a functional inverse of Ackermann's function. If G is a reducible flow graph, each component of G is a single vertex, and the method requires $O(m\,\alpha(m,n))$ time

total. Although the method is rather complicated, a simplified version, which runs in $O(m \log n)$ time, is quite easy to program and efficient in practice.

The paper contains seven sections. Section 2 reviews the properties of regular expressions used in the following sections. Section 3 reviews standard methods of numerical linear algebra and describes their application to the path expression problem. This section introduces the notion of a _path sequence_ for a graph G and shows how, given a path sequence, one can solve the single-source path expression problem for any source in time proportional to the length of the path sequence. Section 4 presents an $O(m\,\alpha(m,n))$-time algorithm for solving a single-source path problem on a reducible flow graph if the source is the start vertex of the graph. Section 5 extends the algorithm so that it computes path sequences for reducible flow graphs. Section 6 generalizes the method to non-reducible graphs. Section 7 discusses applications and suggests further research topics. The appendix contains the basic graph-theoretic terminology used in the paper, An earlier and much different version of this paper appeared as a Stanford technical report [27].

2. Regular Expressions and Path Expressions.

Let Σ be a finite alphabet containing neither "A" nor "$\emptyset$". A regular expression over Σ is any expression built by applying the following rules.

(1a) "Λ" and "$\emptyset$" are atomic regular expressions; for any $a \in \Sigma$, "a" is an atomic regular expression.

(1b) If R_1 and R_2 are regular expressions, then $(R_1 \cup R_2)$, $(R_1 \cdot R_2)$, and $(R_1)^*$ are compound regular expressions.

In a regular expression, Λ denotes the empty string, $\emptyset$ denotes the empty set, $\cup$ denotes set union, . denotes concatenation, and * denotes reflexive, transitive closure under concatenation.*/ Thus each regular expression R over Σ represents a set c(R) of strings over Σ defined as follows:

(2a) $\sigma(\Lambda) = \{\Lambda\}$; $\sigma(\emptyset) = \emptyset$; $\sigma(a) = (a)$ for $a \in \Sigma$.

(2b) $\sigma(R_1 \cup R_2) = \sigma(R_1) \cup \sigma(R_2) = \{w \mid w \in \sigma(R_1) \text{ or } w \in \sigma(R_2)\}$;

$\sigma(R_1 \cdot R_2) = \sigma(R_1) \cdot \sigma(R_2) = \{w_1 w_2 \mid w_1 \in \sigma(R_1) \text{ and } w_2 \in \sigma(R_2)\}$;

$\sigma(R^*) = \bigcup_{k=0}^{\infty} \sigma(R)^k$, where $\sigma(R)^0 = \{\Lambda\}$ and $\sigma(R)^i = \sigma(R)^{i-1} \cdot \sigma(R)$.

---

*/ Note that each of the symbols A , $\emptyset$, $\cup$, ., * stands in the text both for the symbol itself and for a string, set, or operation. We shall allow the context to resolve this ambiguity. Also, we shall freely omit parentheses from regular expressions when the meaning is clear; we assume the standard operator precedence: * over . over $\cup$.

The <u>reverse</u> R^r of a regular expression R is defined by

(3a) $\Lambda^r = \Lambda$; $\emptyset^r = \emptyset$; $a^r = a$ for $a \in \Sigma$.

(3b) $(R_1 \cup R_2)^r = R_1^r \cup R_2^r$;

$(R_1 \cdot R_2)^r = R_2^r \cdot R_1^r$;

$(R_1^*)^r = (R_1^r)^*$.

Two regular expressions R_1 and R_2 are <u>equivalent</u> if $a(R_1) = \sigma(R_2)$. A regular expression R is <u>simple</u> if $R = \emptyset$ or R does not contain $\emptyset$ as a subexpression. We can transform any regular expression R into an equivalent simple regular expression by repeating the following transformations until none is applicable: (i) replace any subexpression of the form $\emptyset \cdot R_1$ or $R_1 \cdot \emptyset$ by $\emptyset$; (ii) replace any subexpression of the form $\emptyset + R_1$ or $R_1 + \emptyset$ by R_1 ; (iii) replace any subexpression of the form $\emptyset^*$ by A.

A regular expression R is <u>non-redundant</u> if R represents every string in a(R) uniquely. We can make this definition precise as follows:

(4a) Λ , $\emptyset$, and a for each $a \in \Sigma$ are non-redundant.

(4b) Let R_1 and R_2 be non-redundant.

$R_1 \cup R_2$ is non-redundant if $\sigma(R_1) \cap \sigma(R_2) = \emptyset$.

$R_1 \cdot R_2$ is non-redundant if each $w \in \sigma(R_1 \cdot R_2)$ is uniquely decomposable into $w = w_1 w_2$ with $w_1 \in \sigma(R_1)$ and $w_2 \in \sigma(R_2)$.

R_1^* is non-redundant if each $w \in \sigma(R^*)$ is uniquely decomposable into $w = w_1 w_2 \ldots w_k$ with $w_i \in \sigma(R_1)$ for $1 \le i \le k$.

Note that if R^* is non-redundant, $\Lambda \notin \sigma(R)$.

Let $G = (V,E)$ be a directed graph. Any path in G is a sequence of edges, which we can regard as a string over E . A path expression P of (v,w) is a simple regular expression over E such that every string in $\sigma(P)$ is a path from v to w . Every subexpression of a path expression is a path expression, whose type can be determined as follows.

(5) Let P be a path expression of type (v,w) .

If $P = P_1 \cup P_2$, then P_1 and P_2 are path expressions of type (v,w) .

If $P = P_1 \cdot P_2$, then there must be a unique vertex u such that P_1 is a path expression of type (v,u) and P_2 is a path expression of type (u,w) .

If $P = P_1^*$, then v = w and P_1 is a path expression of type (v,w) = (v,v).

It is easy to verify (4) using the fact that P is simple. Note that Λ is a path expression of type (v,v) for any v .

In describing algorithms to compute path expressions we shall assume that each $\cup$, $\bullet$, and * operation requires constant time. If we represent the computed path expressions by a directed acyclic graph as described by Aho and Ullman [2, pp. 418-426], this is a reasonable assumption.

3. Path Expression Problems and Path Sequences.

Let $G = (V,E)$ be a directed graph. The single-source path expression problem for source vertex s is the problem of computing, for each vertex $v \in V$, a non-redundant path expression $P(s,v)$ such that $\sigma(P(s,v))$ contains all paths from s to v. The single-sink path expression problem for sink vertex t is the problem of computing, for each vertex $v \in V$, a non-redundant path expression $P(v,t)$ such that $\sigma(P(v,t))$ contains all paths from v to t. The all-pairs path expression problem is the problem of computing, for all pairs $v,w \in V$, a non-redundant path expression $P(v,w)$ such that $\sigma(P(v,w))$ contains all paths from v to w.

In this paper we develop a way to solve path expression problems by using Gaussian elimination in combination with methods for decomposing G into components. In this section we describe how Gaussian elimination applies to such problems. We also describe a well-known decomposition method which uses the strong components of G. In subsequent sections we present a more powerful decomposition method based upon the dominator tree of G.

Gaussian elimination was originally developed to solve a system of linear equations $Ax = b$, where A is an $n \times n$ matrix of real-valued coefficients, x is an $n \times 1$ vector of variables, and b is an $n \times 1$ vector of real-valued constants [11]. The method consists of two steps.

Step 1 (LU decomposition). Decompose A into $A = LU$, where L is unit lower triangular and U is upper triangular.

Step 2 (Frontsolving and backsolving). Solve the triangular systems $Ly = b$ (frontsolving) and $Ux = y$ (backsolving).

The resource requirements of Step 1 dominate those of Step 2 and thus determine the overall requirements of the algorithm [5,28]. The method has several pleasant features, including its amenability to an implementation that takes advantage of the sparsity of A , avoiding arithmetic on numbers known to be zero [8,22]. It is also possible to solve Ax = b for multiple right-hand sides by carrying out Step 1 once and repeating Step 2 for each value of b .

We apply this method to path expression problems by introducing the notion of a path sequence, which generalizes Kennedy's node listing concept [17]. A path sequence for a directed graph G is a sequence $(P_1,v_1,w_1),(P_2,v_2,w_2),\ldots,(P_\ell,v_\ell,w_\ell)$ such that

(6a) For $1 \le i \le \ell$, P_i is a non-redundant path expression of type (v_i,w_i) .

(6b) For $1 \le i \le \ell$, if $v_i = w_i$ then $\Lambda \in \sigma(P_i)$.

(6c) For any non-empty path p in G , there is a unique sequence of indices $1 \le i_1 < i_2 < \ldots < i_k \le \ell$ and a unique partition of p into non-empty paths $p = p_1,p_2,\ldots,p_k$ such that $p_j \in \sigma(P_{i_j})$ for $1 \le j \le k$.

Given a path sequence, we can solve the single-source path expression problem for any source s by using the following propagation algorithm:*/

---

*/ We shall use a syntax resembling Dijkstra's [7] for expressing algorithms.

```
procedure SOLVE;
    begin
initialize:  P(s, s) := Λ; for each v ∈ V-{s} do P(s,v) := ∅ od;
loop:        for i := 1 until ℓ do
                 if v_i = w_i → P(s,v_i) := [P(s,v_i)·P_i]
                 □ v_i ≠ w_i → P(s,w_i) := [P(s,w_i) ∪ [P(s,v_i)·P_i]] fi od
    end SOLVE;
```

In this and subsequent algorithms, the square brackets denote the following simplification procedure. This procedure, when applied recursively, produces regular expressions that are not only simple but also contain no subexpressions of the form $\Lambda \cdot R_1$, $R_1 \cdot \Lambda$, or Λ^* .

```
regular expression procedure [R];
    if R = R_1 ∪ R_2 → if R_1 = ∅ → R_2 □ R_2 = ∅ → R_1 fi
    □ R = R_1·R_2 → if (R_1 = ∅) or (R_2 = ∅) → ∅ □ R_1 = Λ → R_2 □ R_2 = Λ → R_1 fi
    □ R = R_1* → if (R_1 = ∅) or (R_1 = Λ) → Λ fi fi;
```

Lemma 1. Let $(P_1,v_1,w_1),(P_2,v_2,w_2),\ldots,(P_\ell, v_\ell, w_\ell)$ be a path sequence for G and let v be any vertex. After i iterations of the loop in SOLVE, P(s,v) is a non-redundant path expression representing exactly Λ (if s = v) and all non-empty paths p from s to v for which there is a sequence of indices $1 \le i_1 < i_2 < \ldots < i_k < i$ and a partition of P into $P = p_1,p_2,\ldots,p_k$ such that $p_j \in \sigma(P_{i_j})$ for $1 \le j \le k$.

Proof. Straightforward by induction on i . □

Theorem 1. Let $(P_1,v_1,w_1),(P_2,v_2,w_2),\ldots,(P_\ell,v_\ell,w_\ell)$ be a path sequence for G and let v be any vertex. After execution of SOLVE, $P(s,v)$ is a non-redundant path expression representing all paths from s to v.

SOLVE is a generalization of the frontsolving-backsolving step in Gaussian elimination; its running time is $O(n+\ell)$. To solve a single-source path expression problem on a graph G , we construct a path sequence and apply SOLVE once. To solve an all-pairs path expression problem, we construct a path sequence and apply SOLVE n times, once for each possible source. To solve a single-sink path expression problem, we employ the following theorem to construct a path sequence for G^r , and then we solve the corresponding single-source problem on G^r .

Theorem 2. Let $(P_1,v_1,w_1),(P_2,v_2,w_2),\ldots,(P_\ell,v_\ell,w_\ell)$ be a path sequence for a graph G . Then $(P_\ell^r,w_\ell,v_\ell),\ldots,(P_2^r,w_2,v_2),(P_1^r,w_1,v_1)$ is a path sequence for G^r .

Proof. Immediate. □

By Theorem 2 it is no harder to compute a path sequence for G^r than to compute a path sequence for G .

We can construct a path sequence for an arbitrary graph by using a method analogous to Step 1 of Gaussian elimination. The method is similar to Kleene's algorithm for converting a finite automaton into a regular expression [18], except that Kleene uses Gauss-Jordan elimination. Let $G = (V,E)$ be a directed graph whose vertices are numbered from 1 to n and identified by number. The following procedure computes a set of path expressions which when properly ordered gives a path sequence.

```
procedure ELIMINATE;
     begin
initialize:  for v := 1 until n do for w := 1 until n do P(v,w) := ∅ od od;
             for each e ∈ E do P(h(e),t(e)) := [P(h(e),t(e)) ∪ e] od;
loop:        for v := 1 until n do
                  P(v, v) := [P(v,v)*];
                  for each u > v such that P(u,v) ≠ ∅ do
                     P(u,v) := [P(u,v)·P(v,v)];
                     for each w > v such that P(v,w) ≠ ∅ do
                        P(u, w) := [P(u,w) ∪ [P(u,v)·P(v,w)]] od od
     end ELIMINATE;
```

Lemma 2. After the v-th iteration of the loop in ELIMINATE, the following statements are true.

(i) $P(u,w)$ for $u > w$ and $w < v$ is a non-redundant path expression representing exactly the paths from u to w which contain no intermediate vertex larger than w.

(ii) $P(u,w)$ for $u < w$ or $w > v$ is a non-redundant path expression representing exactly the non-empty paths from u to w all of whose intermediate vertices are smaller than $\min\{u, v+1\}$.

Proof. Straightforward by induction on v. □

Theorem 3. After execution of ELIMINATE the following statements are true.

(i) $P(u,w)$ for $u > w$ is a non-redundant path expression representing exactly the paths from u to w which contain no intermediate vertex larger than w.

(ii) $P(u,w)$ for $u < w$ is a non-redundant path expression representing exactly the paths from u to w all of whose intermediate vertices are smaller than u.

Theorem 4. Let $P(u,w)$ for $u,w \in V$ be the path expressions computed by ELIMINATE. Then the following sequence is a path sequence: the elements of $\{(P(u,w),u,w) \mid P(u,w) \notin \{\emptyset,\Lambda\}$ and $u \leq w)$ in increasing order on u, followed by the elements of $\{(P(u,w),u,w) \mid P(u,w) \neq \emptyset$ and $u > w)$ in decreasing order on u.

Proof. The sequence specified in the theorem certainly satisfies (6a) and (6b). To prove (6c), let p be any non-empty path in G. Let v_0 be the maximum vertex on p. Let p_0 be the part of p from the first occurrence of v_0 to the last occurrence of v_0 (if v_0 only occurs once, $p_0 = \Lambda$). For $i \geq 1$, let v_i be the largest vertex occurring on p after the last occurrence of v_{i-1}, and let p_i be the part of p from the last occurrence of v_{i-1} to the last occurrence of v_i. Let v_ℓ be the last such v_i defined $(v_\ell = t(p))$. For $i > 1$, let v_{-i} be the largest vertex occurring on p before the first occurence of v_{-i+1}. Let p_{-2i+1} be the part of p from the last occurrence of v_{-i} before p_{-2i+2} to the beginning of p_{-2i+2}, and let p_{-2i} be the part of p from the first occurrence of v_{-i}

to the beginning of p_{-2i+1} . Let v_{-k} be the last such v_{-i} defined ($v_{-k} = h(p)$) . Then $p = p_{-2k}, p_{-2k+1}, \ldots, p_{-1}, p_0, p_1, \ldots, p_\ell$ with $p_{-2i} \in \sigma(P(v_{-i}, v_{-i}))$ for $0 \le i \le k$, $p_{-2i+1} \in \sigma(P(v_{-i}, v_{-i+1}))$ for $1 \le i \le k$, and $p_i \in \sigma(P(v_{i-1}, v_i))$ for $1 \le i \le \ell$. Ignoring empty paths p_i , we get a partition of p which satisfies (6b). It is straightforward but tedious to show that this partition is unique. □

ELIMINATE thus gives us a way to construct path sequences. The resource requirements of the method depend in a complicated way upon the sparsity of G. By rearranging the computation in the loop of ELIMINATE and using appropriate data structures we can implement ELIMINATE to run in

$$O\left(\ell + \sum_{v=1}^{n} |\{P(u,v) \neq \emptyset \mid u > v\}| \cdot |\{P(v,w) \neq \emptyset \mid w > v\}| \right)$$ time and $O(\ell)$

storage space, where ℓ is the length of the computed path sequence [5,28]. (By only storing $P(u,w)$ for pairs u, w such that eventually $P(u,w) \neq \emptyset$, we can avoid spending $O(n^2)$ time in initialization.)

For dense graphs the time bound is $O(n^3 + m)$ and the space bound is $O(n^2)$. For sparse graphs, the resource requirements depend upon the vertex numbering chosen. Numerical analysts have devoted much effort to finding good numbering schemes, both for arbitrary sparse graphs and for graphs with special. structure [5,8,22,28]. All their techniques <u>except</u> off-diagonal pivoting [11] apply to the computation of path sequences.

In order to improve the efficiency of this method, we shall combine it with two decomposition techniques. The idea is to break the problem

graph into subgraphs, apply ELIMINATE to construct a path sequence for each subgraph, and combine these path sequences into a path sequence for the original graph. Our first decomposition technique is well-known to numerical analysts and uses the strong components of G .

Theorem 5. Suppose $G = (V,E)$ is acyclic (i.e., each strong component is a single vertex) and that the vertices of G are numbered in topological order. Then the elements of $\{(e,h(e),t(e)) \mid e \in E\}$ in increasing order on $h(e)$ comprise a path sequence.

Proof. Immediate. □

By Theorem 5, any acyclic graph has a path sequence of length m , which can be found in $O(n+m)$ time using a linear-time topological sorting procedure [19,25].

Theorem 6. Suppose $G = (V,E)$ is a directed graph with strong components $G_1,G_2,\ldots,G_k$, ordered so that no edge leads from a component G_i to a component G_j with $j < i$. For $1 \leq i \leq k$, let X_i be a path sequence for G_i , and let Y_i be a sequence consisting of the elements of $\{(e,h(e),t(e)) \mid h(e) \in G_i \text{ and } t(e) \notin G_i\}$ ordered arbitrarily. (Note that Y_k is empty.) Then $X_1,Y_1,X_2,Y_2,\ldots,X_{k-1},Y_{k-1},X_k$ is a path sequence for G .

Proof. Immediate. □

Theorem 6 generalizes the method of Theorem 5 to arbitrary directed graphs. We can find the strong components of a directed graph in $O(n+m)$ time using the algorithm of Tarjan [24]. Thus Theorem 6 gives a method

for finding a path sequence in $O(n+m)$ time plus the time to find path sequences for the strong components. The length of the sequence is $O(m)$ plus the total length of the strong components' sequences.

4. Computing Path Expressions for Reducible Flow Graphs.

Although decomposition using strong components is efficient and useful in practice, many problem graphs have one or only a few strong components. In the remaining sections of this paper we develop a more powerful decomposition technique based upon dominators. We begin by considering reducible flow graphs. A flow graph $G = (V,E,r)$ is a directed graph with a distinguished start vertex r such that every vertex in G is reachable from r. By Theorem 6 we need only consider strongly connected graphs, so this reachability condition is no restriction.

A reducible flow graph $G = (V,E,r)$ is a flow graph that can be reduced to the graph consisting of the single vertex r and no edges by means of the following transformations:

T_1 (remove a loop): If e is an edge such that $h(e) = t(e)$, delete edge e.

T_2 (remove a vertex): If $w \neq r$ is a vertex such that all edges e with $t(e) = w$ have $h(e) = v$ for some vertex v, contract w into v by deleting w and all edges entering w, and converting any edge e with $h(e) = w$ into an edge e' with $h(e') = v$ and $t(e') = t(e)$.

This definition is due to Hecht and Ullman [14]; there are many other equivalent definitions of reducible flow graphs [12,14,15,26]. Intuitively a flow graph is reducible if every cycle has a single entry from the start vertex. These graphs play an important role in global flow analysis, because the control flow of a reasonably well-structured program can be modelled by a reducible flow graph [3,20].

As the reduction by T1 and T2 takes place, each vertex in the reduced graph represents a subgraph of the original graph, called a region, and each edge in the reduced graph represents an edge in the original graph. We define this notion formally as follows.

(7a) Each vertex and edge in the original graph represents itself.

(7b) If T_1 is applied to delete an edge e, then vertex $h(e) = t(e)$ in the reduced graph represents the union of what $h(e)$ and e represent.

(7c) If T_2 is applied to contract vertex w into vertex v, then V in the reduced graph represents the union of what v, w, and all the deleted edges e with $h(e) = v$, $t(e) = w$ represent. Any new edge e' represents what the corresponding old edge e represents.

It is not hard to show that each region is indeed a subgraph of G and that the regions corresponding to the vertices of any reduced graph are vertex-disjoint [31]. Furthermore every region I has a unique header vertex v such that any edge e with $h(e) \notin I$, $t(e) \in I$ has $t(e) = v$ [31]. The header is the unique vertex in the region which has not yet been contracted into another vertex. When the reduction is complete, r represents a region comprising the entire graph G.

If a flow graph is reducible, there is a reduction order $v_1, v_2, \ldots, v_{n-1}, v_n = r$ of the vertices such that the graph can be reduced to r in the following way [26]: For i from 1 to n-1, we apply T_1 to delete all loops at v_i; then we apply T_2 to contract v_i into another vertex v_j with

$j > i$. After deleting all vertices except $v_n = r$, we apply T_1 to delete all loops at r. This way of carrying out the reduction has the following property. If we regard the repeated application of T_1 at a vertex v_i followed by the application of- T_2 to delete v_i as a single step, then between any two steps the entry vertex of any region has no edges entering it from within the region.

We shall assume henceforth that the vertices of G are numbered from 1 to n in a reduction order and identified by number. We shall also assume that *header*(v) for $v \neq r$ is the vertex into which v is eventually contracted, that *cycle*(v) for any vertex v is the set of edges in G represented by edges deleted when applying T_1 to delete loops at v, and that *noncycle*(v) for $v \neq r$ is the set of edges in G represented by edges deleted when applying T_2 to delete v. The following lemma states some basic properties of *header*, *cycle*, and *noncycle*.

Lemma 3. Suppose G is a reducible flow graph whose vertices are numbered in a reduction order. Let v be any vertex and let e be any edge. Then

(i) if $v \neq r$, $\underline{\text{header}}(v) > v$;

(ii) either $h(e) = \underline{\text{header}}(t(e))$ or $h(e) \leq t(e)$;

(iii) if $e \in \underline{\text{cycle}}(t(e))$ then $\underline{\text{header}}^i(h(e)) = t(e)$ for some $i > 0$; and

(iv). if $e \in \underline{\text{noncycle}}(t(e))$ then $\underline{\text{header}}^i(h(e)) \neq t(e)$ for all $i > 0$ but $\underline{\text{header}}^i(h(e)) =$ *header*@(e)) for some $i > 0$.

Proof. Straightforward. □

The algorithm of Tarjan [26] computes a reduction order and associated arrays <u>header</u>, <u>cycle</u>, and <u>noncycle</u> in $O(m\ \alpha(m,n))$ time. Using this information we can solve the single-source path expression problem whose source vertex is r. The algorithm resembles the methods of Ullman [31] and Graham and Wegman [12] for solving "forward" data flow problems; we discuss this resemblance at the end of the section.

The algorithm computes path expressions as the reduction proceeds, using a data structure representing the current regions. The data structure consists of a forest whose vertices are the vertices of G and whose edges are the pairs (<u>header</u>(v),v) such that v has been contracted into <u>header</u>(v). Thus this <u>header forest</u> consists of one tree per region; the tree representing a region contains exactly the vertices in the region and has the header of the region as its root. With every vertex v in the forest is associated a non-redundant path expression R(v). The algorithm manipulates the forest by means of four operations:

INITIALIZE(v): Form a tree with one vertex v and associated path expression $R(v) := \Lambda$.

UPDATE(v,R): If v is a root, assign $R(v) := R$,

LINK(v,w): If v and w are roots, combine the trees with roots v and w by making v the parent of w.

EVAL(v): If $r = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = v$ is the tree path from the root r of the tree containing v to v, return a non-redundant path expression equivalent to $R(v_0) \cdot R(v_1) \cdot \ldots \cdot R(v_k)$.

The algorithm maintains the following invariant: If I is a region and v is a vertex in I , then EVAL(v) represents exactly all paths in I from the header of I to v .

```
     procedure REDUCE;
          begin
initialize:     for each v∈V do INITIALIZE(v) od;
loop:           for v := 1 until n-1 do
                     P:= ∅; Q := ∅;
                     for each e ∈ noncycle(v) do P := [P ∪ [EVAL(h(e))·e]] od;
                     for each e ∈ cycle(v) do Q := [Q ∪ [EVAL(h(e))·e]] od;
                     UPDATE(v,[P·[Q*]]);
                     LINK(header(v),v) od;
finalize:       P(r,r) := ∅;
                for each e ∈ cycle(r) do P(r,r) := [P(r,r) ∪ [EVAL(h(e))·e]] od;
                P(r, r) := [P(r,r)*];
                for v := 1 until n-1 do P(r,v) := [P(r,r)·EVAL(v)] od
          end REDUCE;
```

Lemma 4. After the v-th iteration of the loop in REDUCE, EVAL(u) for any vertex u represents exactly all paths in the current region I containing u from the header of I to u .

Proof. By induction on v . The lemma is certainly true before the first iteration of the loop. Suppose the lemma is true before the v-th iteration of the loop. Let I_1 be the current region containing v and

let I_3 be the current region containing header(v) . Let I_2 be the region containing v after T_1 is applied to eliminate all loops at v . Let I_4 be the region containing v after T_2 is applied to contract v into header(v) ; i.e., after the v-th iteration of the loop.

I_2 consists of I_1 and the edges in cycle(v) . I_4 consists of I_2 , I_3 , and the edges in noncycle(v) ; the header of I_4 is the header of I_3 .

I_1 contains no edges entering v . It follows from the induction hypothesis that the value of Q after the v-th iteration is a non-redundant path expression representing all paths from v to v in I_2 which do not contain v as an intermediate vertex. Thus Q^* represents all paths in I_2 from v to v . It also follows from the induction hypothesis that the value of P after the v-th iteration is a non-redundant path expression representing all paths in I_4 from the header of I_4 to v which do not contain v as an intermediate vertex.

If u is a vertex in I_3, then the paths in I_4 from the header of I_4 to u are exactly the paths in I_3 from the header of I_3 to u . If u is a vertex in I_2 , the paths in I_4 from the header of I_4 to u are exactly the paths p partitionable into $P = p_1, p_2, p_3$, where $p_1 \in \sigma(P)$, $p_2 \in \sigma(Q^*)$, and p_3 is a path in I_1 from the header of I_1 to u . Thus adding edge (header(v),v) to the forest and replacing the old value (Λ) of P(v) by $[P \cdot [Q^*]]$ guarantees that the lemma holds after the v-th iteration of the loop. □

Corollary 1. After execution of REDUCE, $R(v)$ for any vertex $v \neq r$ is a non-redundant path expression representing exactly the set of paths from header(v) to v all of whose intermediate vertices are smaller than header(v) .

Proof. For any vertex $v \neq r$, let I_4 be the region containing v after the v-th iteration of the loop in REDUCE. Let $R(v)$ be the path expression computed for v during this iteration. By Lemma 4, $R(v)$ is a non-redundant path expression representing all paths in I_4 from header(v) to v . Any path in G from header(v) to v which leaves I_4 must contain header(v) twice, since the only way to enter I_4 is through header(v) . □

Theorem 7. Let v any vertex. After execution of REDUCE, $P(r, v)$ is a non-redundant path expression representing all paths from r to v.

Proof. Lemma 4 holds after the last iteration of the loop in REDUCE. A proof similar to that of Lemma 4 shows that $P(r,r)$ as computed in the final part of REDUCE is a non-redundant path expression representing all paths from r to r in G. It follows from Lemma 4 that the computed value of $P(r,r)$ for $v \neq r$ is a non-redundant path expression representing all paths from r to v in G . □

Procedure REDUCE requires $O(n+m)$ time plus time for n calls on INITIALIZE, n-1 calls on UPDATE, n-1 calls on LINK, and m+n-1 calls on EVAL; thus the forest manipulation operations dominate the running time of the algorithm. Tarjan [29] describes two ways to implement the forest operations. The first is a simple method

called path compression which requires $O(m \log n)$ time. The second is a sophisticated off-line method which by preprocessing the entire sequence of EVAL and LINK operations is able to perform all the forest manipulation in $O(m\,\alpha(m,n))$ time... (It is easy to precompute the sequence of EVAL and LINK operations performed by REDUCE.) Farrow [9] presents another $O(m\,\alpha(m,n))$ -time method called stratified path compression. This method has the advantage of being on-line, although the proof of its time bound is very complicated.

By using either of the $O(m\,\alpha(m,n))$ -time algorithms for forest manipulation we obtain a moderately complicated $O(m\,\alpha(m,n))$ -time implementation of REDUCE. By using path compression we obtain an $O(m \log n)$ -time implementation of REDUCE which is remarkably simple and efficient. We favor the latter implementation for practical applications.

Ullman's algorithm for forward data flow analysis [31] is essentially identical to REDUCE except that it uses 2-3 trees to carry out the forest operations. Its time bound is $O(m \log n)$ but it is more complicated than our method using path compression. Graham and Wegman's algorithm [12] is a version of REDUCE which uses no auxiliary data structure but carries out a form of path compression on the original graph. Its time bound is $O(m \log n)$ but it also is more complicated than our method using path compression. Experimental comparisons between these methods would be valuable.

5. Computing Path Sequences for Reducible Flow Graphs,

Some kinds of data flow analysis, such as the computation of live variables [17], require that information be propagated backward rather than forward through the control flow graph of the program. We can carry out such backward data flow analysis by solving a single-source path problem on the reverse of the control flow graph. Since reducibility is not preserved by graph reversal, the algorithm of Section 5 is inadequate for this purpose. In this section, we shall modify REDUCE so that it computes a path sequence for any reducible flow graph. By using such a path sequence and applying Theorem 6 if necessary, we can solve single- and multi-source path problems on any flow graph which is reducible or whose reverse is reducible, This provides an efficient way to do backward data flow analysis.

In order to develop this algorithm, we need to examine the implementation of the header forest operations. We shall describe a generic implementation of which path compression [29] and stratified path compression [9] are special cases. We shall use this generic implementation in an extension of REDUCE which computes path sequences.

The generic implementation uses a compressed forest to represent the header forest. With each vertex v_j of the compressed forest is associated a path expression $S(v)$. The method maintains the following invariants.

(8a) For each tree T in the header forest, there is a corresponding tree T^c of the compressed forest which contains the same vertices as T .

(8b) If $v \to w$ in a tree T^c of the compressed forest, then $v \overset{*}{\to} w$ in T. In particular, corresponding trees T and T^c have the same root.

(8c) For any vertex v, let $r = v_0 \to v_1 \to \cdots \to v_k = v$ be the path in the header forest from a root to v, and let $r = w_0 \to w_1 \to \cdots \to w_\ell = v$ be the path in the compressed forest from a root to v. Then $R(v_0) \cdot R(v_1) \cdot \ldots \cdot R(v_k)$ and $S(w_0) \cdot S(w_1) \cdot \ldots \cdot S(w_\ell)$ are equivalent non-redundant path expressions.

The compressed forest is represented by an array ancestor such that ancestor(v) is the parent of v in the compressed forest; if ancestor(v) = 0 then v is a root. The following procedures implement the forest operations.

```
procedure INITIALIZE(v);
    begin ancestor(v) := 0; S(v) := Λ end;

procedure UPDATE(v,R);
    S(v) := R;

procedure LINK(v,w);
    ancestor(w) := v;
```

```
regular expression procedure EVAL(v);
    begin
        non-deterministically execute COMPRESS(u) for an
           arbitrary sequence of vertices u;
        let v_0,v_1, . . .,v_k be such that v = v_k, ancestor(v_i) = v_{i-1} for
           for 1 ≤ i ≤ k, and ancestor(v_0) = 0;
        EVAL := if k = 0 → Λ
                 [] k ≠ 0 → S(v_1) . S(v_2) . . . . . S(v_k) fi
    end EVAL;

procedure COMPRESS(u);
    if ancestor(ancestor(u)) ≠ 0 →
        S(u) := S(ancestor(u)) . S(u);
        ancestor(u) := ancestor(ancestor(u)) fi;
```

It is evident that COMPRESS preserves (8a)-(8c); thus the procedures above are a valid implementation of the header forest operations. The following lemma is easy to prove using the results in Section 4.

Lemma 5. If v is any vertex such that ancestor$(v) \neq 0$, then $S(v)$ is a non-redundant path expression representing exactly the set of paths from ancestor(v) to v all of whose intermediate vertices are smaller than ancestor(v).

EVAL is a non-deterministic procedure which is free to choose an arbitrary sequence of vertices u on which to execute COMPRESS(u). We obtain a specific implementation by including a mechanism for making this choice. Path compression uses the following version of EVAL.

```
regular expression procedure EVAL(v);
    if ancestor(v) = 0 → EVAL := A
    [] ancestor(v) ≠ 0 → PATH_COMPRESS(v); EVAL := S(v) fi;

procedure PATH-COMPRESS(v);
    if ancestor(ancestor(v)) ≠ 0 →
        PATH-COMPRESS(ancestor(v));
        S(v) := S(ancestor(v)) . S(v);
        ancestor(v) := ancestor(ancestor(v)) fi;
```

Stratified path compression uses a more complicated compression mechanism which requires the maintenance of additional data structures [9].

The following version of REDUCE uses the generic implementation of the header forest operations to compute a path sequence. Procedures EVAL and COMPRESS are modified so that they add elements to the path sequence as a side effect.

```
procedure REDUCE AND-SEQUENCE;
      begin
initialize:  for each v∈V do INITIALIZE(v) od;
             sequence := the empty sequence;
loop:        for v := 1 until n-1 do
                 P := ∅; Q := ∅;
                 for each e ∈ noncycle(v) do P := [PUEVAL_AND_SEQUENCE(e)] od;
                 for each e ∈ cycle(v) do Q := [QUEVAL_AND_SEQUENCE(e)] od;
add1:            if [Q*] ≠ Λ → add ([Q*],v,v) to sequence fi;
                 UPDATE(v, [P·[Q*]]);
                 LINK(header(v),v) od;
finalize:    Q := ∅; --
             for each e ∈ cycle(r) do Q :=[QUEVAL AND SEQUENCE(e)] od;
add2:        if [Q*] ≠ Λ → add ([Q*],r,r) to sequence fi;
             for v := n-1 by -1 until1 do add (S(v),ancestor(v),v) to sequence od
      end REDUCE_AND_SEQUENCE;

regular expression procedure EVAL_AND_SEQUENCE(e);
      begin
          non-deterministically execute COMPRESS AND SEQUENCE(u) for
            an arbitrary sequence of vertices u;
          let v_0, v_1, . . .,v_k be such that h(e) = v_k, ancestor(v_i) = v_i 1 for
            1 ≤ i ≤ k, and ancestor(v_0) = 0;
          if k = 0 → EVAL AND SEQUENCE := e
          [] k ≠ 0 → EVAL_AND_SEQUENCE := S(v_k)·e;
               for i := k-1 by -1 until 1 do
                    add (EVAL_AND_SEQUENCE,v_i,t(e)) to sequence;
                    EVAL_AND_SEQUENCE := S(v_i) • EVAL_AND SEQUENCE od fi
      end EVAL_AND_SEQUENCE;
```

```
procedure COMPRESS/D-SEQUENCE(u);
    if ancestor(ancestor(u)) ≠ 0 →
        add (S(u),ancestor(u),u) to sequence;
        S(u) := S(ancestor(u))·S(u);
        ancestor(u) := ancestor(ancestor(u)) fi;
```

Theorem 8. The sequence computed by REDUCE-AND SEQUENCE is a path sequence for G .

Proof. The proof is similar to the proof of Theorem 4 but a little more complicated. We shall assume for purposes of the proof that statement add 1 always adds $([Q^*],v,v)$ to sequence , whether or not $[Q^*] = \Lambda$; similarly for statement add 2. This modification does not affect the properties of sequence in which we are interested.

Lemma 5 and an inspection of REDUCE_AND_SEQUENCE show that the computed sequence satisfies (6a) and (6b). To prove (6c), let p be an arbitrary path in G . Let $v_0 = h(p)$. For $i \geq 1$, let v_i be the first vertex on p such that $v_i > v_{i-1}$. Let v_k be the last vertex so defined (v_k is the largest vertex on p). Let $v_{k+1} = t(p)$. Let p_{2k} be the part of p from the first occurrence of v_k to the last occurrence of v_k . Let p_{2k+1} be the part of p following p_{2k} . For $0 < i < k-1$, let p_{2i+1} be the part of p from the last occurrence of v_i before p_{2i+2} to the beginning of p_{2i+2}. Let p_{2i} be the part of p from the first occurrence of v_i to the beginning of p_{2i+1} . Then $p = p_0, p_1, \ldots, p_{2k+1}$, where p_{2i} for $0 \leq i \leq k$ is a path from v_i to v_1 containing no vertex greater than v_i , and p_{2i+1} for $0 \leq i \leq k$ is a path from v_1 to v_{i+1} all of whose intermediate vertices are less than v_i .

For $0 \le i \le k$, $p_{2i} \in \sigma(Q^*(v_i))$, where $Q(v_i)$ for $v_i \neq r$ is the value of Q computed during the v_i -th iteration of the loop in REDUCE_AND_SEQUENCE, and Q(r) is the value of Q computed during the final part of REDUCE-AND-SEQUENCE. In order to represent p as in (6c), it remains for us to (i) partition each path p_{2i+1} for $0 \le i \le k-1$ into a sequence of paths represented by triples appearing in <u>sequence</u> between $([Q(v_i)^*], v_i, v_i)$ and $([Q(v_{i+1})^*], v_{i+1}, v_{i+1})$, and (ii) partition p_{2k+1} into a sequence of paths represented by triples appearing in <u>sequence</u> after $([Q(v_k)^*], v_k, v_k)$.

Consider any path p_{2i+1} for $0 \le i \le k-1$. Let e_i be the last edge on this path-. Then $t(e_i) = v_{i+1}$, and $h(e_i)$ is a descendant of v_i in the compressed tree just after the v_i -th iteration of the loop in REDUCE-ND-SEQUENCE. We partition p_{2i+1} into $p_{2i+1} = p_{2i+1,0}, p_{2i+1,1}, \ldots, p_{2i+1,\ell}$ as follows. Let $j = 0$ and $p_{2i+1}^{(0)} = p_{2i+1}$. Repeat the following step until it no longer applies.

<u>General step</u>. Suppose $h(e_i)$ is not a descendant of $h(p_{2i+1}^{(j)})$ in the compressed tree when edge e_i is processed by REDUCE. Consider the moment when $h(e_i)$ becomes a non-descendant of $h(p_{2i+1}^{(j)})$. This event must be caused by an execution of COMPRESS(u) such that <u>ancestor</u>(u) $= h(p_{2i+1}^{(j)})$. Let $p_{2i+1,j}$ be the part of $p_{2i+1}^{(j)}$ from the beginning to $p_{2i+1}^{(j)}$ to the last occurrence of u. Partition $p_{2i+1}^{(j)}$ into $p_{2i+1}^{(j)} = p_{2i+1,j}, p_{2i+1}^{(j+1)}$ and replace j by j+1.

Consider a single execution of the general step, Path $p_{2i+1}^{(j)}$ must contain u since $h(p_{2i+1}^{(j)}) \overset{*}{\rightarrow} u \overset{*}{\rightarrow} h(e_i)$ in the header tree. Thus $p_{2i+1}^{(j)}$ can be partitioned as stated. Execution of COMPRESS(u) causes $(S(u), h(p_{2i+1}^{(j)}), u)$ to be added to <u>sequence</u> ; $p_{2i+1,j} \in \sigma(S(u))$. After execution of COMPRESS, $h(e_i)$ is a descendant of $u = h(p_{2i+1}^{(j+1)})$ in the compressed tree.

Suppose the general step is executed ℓ times. Let $p_{2i+1,\ell} = p_{2i+1}^{(\ell)}$. By the discussion above, there is a subsequence of triples (P_0,u_0,w_0) , (P_1,u_1,w_1) , ..., $(P_{\ell-1},u_{\ell-1},w_{\ell-1})$ appearing in <u>sequence</u> after $([Q(v_i)^*],v_i,v_i)$ and before triples of the form (P,u,v_{i+1}) , and such that $p_{2i+1,j} \in P_j$ for $0 \leq j \leq \ell-1$. Furthermore $h(e_i)$ is a descendant of $h(p_{2i+1,\ell})$ in the compressed tree just after all compression is finished during the execution of EVAL_AND_SEQUENCE(e_i) . The operation of EVAL_AND_SEQUENCE(e_i) adds a triple $(P_\ell, h(p_{2i+1,\ell}), v_{i+1})$ such that $p_{2i+1,\ell} \in \sigma(P_\ell)$ to <u>sequence</u> . Thus we obtain a satisfactory partition of p_{2i+1} .

The partitioning of p_{2k+1} is the same as the partitioning of p_{2i+1} for $1 \leq i \leq k-1$ except that the path $p_{2i+1,\ell}$ must be further partitioned into paths represented by triples (S(v),<u>ancestor</u>(v),v) added to <u>sequence</u> during the final part of REDUCE_AND_SEQUENCE. The details are straightforward.

We obtain by the method above a partition of an arbitrary path p which satisfies (6c) if we ignore empty paths in the partition. Showing that the partition is unique is tedious but not difficult. The crucial point is that for any pair u > v , only one triple of

the form (P,u,v) appears in sequence . We leave the details to the reader. Cl

REDUCE_AND_SEQUENCE requires $O(m \log n)$ time to construct a path sequence if path compression is used to implement the forest operations and $O(m\,\alpha(m,n))$ if stratified path compression is used. The length of the path sequence constructed is proportional to the running time. It is interesting to note that the version of the algorithm which carries out no compression generates essentially the same path sequence as ELIMINATE.

6. Decomposition Using Dominators.

In this section we generalize the algorithm of Section 5 so that it becomes a decomposition method applicable to all graphs, The reducible graphs play a role in this method analogous to the role of acyclic graphs in decomposition by strong components, Just as a graph is acyclic if and only if all its strong components are single vertices, a graph is reducible if and only if all its components in the new decomposition are single vertices.

The concept we use is that of a single-entry region, which we make precise as follows. For an arbitrary flow graph $G = (V,E,r)$, we say a vertex v <u>dominates</u> another vertex w if $v \neq w$ and v lies on every path from r to w.

<u>Lemma 6</u> [1]. There is a tree T , called the <u>dominator tree</u> of G , such that v is a proper ancestor of w in T if and only if v dominates w . Vertex r is the root of T and D contains every vertex in G .

For any vertex $v \neq r$, we denote by <u>idom</u>(v) the parent of v in T . Vertex <u>idom</u>(v) is called the <u>immediate dominator</u> of v and is the unique vertex which dominates v and is dominated by every other dominator of v . The dominator tree defines the single-entry regions of G; the following lemma is a technical statement of this fact. (Note the similarity between this lemma and Lemma 3.)

<u>Lemma7.</u> For any edge e , <u>idom</u>(t(e)) is an ancestor of h(e) in T .

<u>Proof,</u> Every path from r to t(e) contains <u>idom</u>(t(e)) . By adding edge e to any path from r to h(e) , we get a path from r to t(e) .

Thus any path from r to $h(e)$ contains $\underline{idom}(t(e))$, and by Lemma 6 $\underline{idom}(t(e)) \overset{*}{\rightarrow} h(e)$ in T . □

For any edge e , let $\tilde{e}$ be an edge such that $t(\tilde{e}) = t(e)$ and $h(\tilde{e}) = h(e)$ if $h(e) = \underline{idom}(t(e))$, $h(\tilde{e}) = u$ where $\underline{idom}(t(e)) \rightarrow u \overset{*}{\rightarrow} h(e)$ in T if $t(e) \neq \underline{idom}(h(e))$. Let $\tilde{G} = (V, \tilde{E}, r)$, where $\tilde{E} = \{\tilde{e} \mid e \in E\}$. We call $\tilde{G}$ the derived graph of G. Figures 1-3 illustrate a graph, its dominator tree, and its derived graph. Note that there are three kinds of edges in the derived graph. If $t(e) = \underline{idom}(h(e))$, then $\tilde{e} = e$ is an edge in T , If $t(e) \overset{*}{\rightarrow} h(e)$ in T then $\tilde{e}$ is a loop. Otherwise $\tilde{e}$ leads from one sibling to another in T .

[Figure 1]

[Figure 2]

[Figure 3]

We call the strong components of $\tilde{G}$ the dominator strong components of G. It is not hard to prove that a graph is reducible if and only if all its dominator strong components are single vertices. The idea of our algorithm is to use Gaussian elimination (or some other method) to compute a path sequence for each dominator strong component of $\tilde{G}$, and to combine these path sequences to form a path sequence for G by using a combination of the methods in Sections 3 and 5. The algorithm manipulates the dominator tree in the same way that REDUCE_AND_SEQUENCE manipulates the tree defined by the header pointers. Henceforth when we refer to descendants and ancestors we mean with respect to the dominator tree T .

The algorithm assumes that the dominator tree of G is known and that the vertices are numbered from 1 to n so that $\underline{idom}(v) > v$ for each vertex $v \neq r$. The algorithm requires the following information: for each vertex u the set $\underline{children}(u)$ of vertices v such that $\underline{idom}(v) = u$, the set $\underline{tree}(u)$ of edges e such that $t(e) = u$ and $h(e) = \underline{idom}(u)$, and the set $\underline{nontree}(u)$ of edges e such that $t(e) = u$ and $h(e) \neq \underline{idom}(u)$; for each edge e the corresponding edge e in $\tilde{G}$. This information and the vertex numbering can be computed in $O(m\,\alpha(m,n))$ time using the dominators algorithm of Lengauer and Tarjan [21].

The algorithm groups together vertices with a common parent and processes these sibling sets in increasing order by parent. The algorithm processes the set of siblings $\underline{children}(u)$ for each vertex u as follows. For each edge e such that h(e) is a child of u, the algorithm uses EVAL_AND_SEQUENCE to compute a path expression $p(\tilde{e})$ representing all paths in G from $h(\tilde{e})$ to $t(\tilde{e})$ which end with edge e and contain only proper descendants of $h(\tilde{e})$ as intermediate vertices. Then the algorithm computes a path sequence X_u for the subgraph $\tilde{G}_u$ of $\tilde{G}$ induced by $\underline{children}(u)$. Substituting $P(\tilde{e})$ for for each edge $\tilde{e}$ appearing in this path sequence produces a sequence Y_u that represents every path in G starting and ending at a child of u and containing only proper descendants of u as intermediate vertices.

The algorithm concatenates Y_u onto the end of the path sequence, By applying SOLVE to Y_u, the algorithm computes for each child v of u a path expression R(v) which represents all paths in G from

u to v containing only proper descendants of u as intermediate vertices. The algorithm completes the processing of the sibling set by executing UPDATE(v,R(v)) ; LINK(u,v) for each child v of u .

The algorithm finishes by computing a path expression Q representing all paths from r to r and adding additional triples to the path sequence just REDUCE_AND_SEQUENCE does. The algorithm appears in more detail below.

```
     procedure DECOMPOSE_AND_SEQUENCE;
          begin
initialize:   for each v∈V do INITIALIZE(v) od;
              sequence = the empty sequence;
loop:         for u := 1 until n do
derive:           for each v ∈ children(u) do
                      for each e ∈ non-tree(v) do
                          P(ẽ) := EVAL_AND_SEQUENCE(e) od od;
eliminate:        compute a path sequence X_u for G_u;
substitute:       form Y_u from X_u by replacing each occurrence of an
                    edge ẽ in a path expression by P(ẽ);
                  sequence := sequence concatenated with Y_u;
solve :           for each v ∈ children(u) do R(v) := ∅;
                      for each e ∈ tree(v) do R(v) := [R(v) ∪ e] od od;
                  for each (P,w,x) ∈ Y_u in order do
                      if w = x → R(w) := [R(w)·P]
                      [] w ≠ x → R(x) := [R(x) ∪ [R(w)·P]] fi od;
update:           for each v ∈ children(u) do
                      UPDATE(v,R(v)); LINK(u,v) od od;
```

```
finalize:     Q := ∅;
              for each e ∈ nontree(r) do Q := [Q ∪ EVAL_AND_SEQUENCE(e)] od;
              if [Q*] ≠ Λ add ([Q*],r,r) to sequence fi;
              for v := n-1 by -1 until 1 do add (S(v), ancestor(v), v)
                to sequence od
          end DECOMPOSE AND-SEQUENCE;
```

This method combines the techniques of Section 3 with the method of Section 5. The parts of the program labelled initialize , derive , update , and finalize are adapted from REDUCE_AND_SEQUENCE and serve to combine the path sequences computed for the dominator strong components (in eliminate-- and substitute) into a path sequence for the entire graph. The two loops labelled solve comprise a version of SOLVE,

We can implement step eliminate using ELIMINATE on the strong components of $\tilde{G}_u$ and combining the results as described in Theorem 6. Step substitute can be performed either after or during the computation of x_U ; the latter is preferable.

The next lemma expresses the properties of the values computed by DECOMPOSE_AND_ELIMINATE; its proof combines the ideas in Theorem 1 and Corollary 1.

Lemma 8. (i) For each edge e in G such that $e \in$ nontree(t(e)) , $P(\tilde{e})$ as computed by DECOMPOSE_AND_SEQUENCE is a non-redundant path expression representing exactly the paths in G from $h(\tilde{e})$ to $t(\tilde{e})$ which end with edge e and contain only proper descendants of $h(\tilde{e})$ as intermediate vertices.

(ii) For each vertex v in G , R(v) as computed by DECOMPOSE_AND_SEQUENCE is a non-redundant path expression representing exactly the paths in G

from *idom*(v) to v which contain only proper descendants of *idom*(v) as intermediate vertices.

(iii) For each vertex u in G , Y_u as computed by DECOMPOSE_AND_SEQUENCE is a sequence $Y_u = (P_1,v_1,w_1),(P_2,v_2,w_2),\ldots,(P_\ell,v_\ell,w_\ell)$ satisfying (6a), (6b), and

(9) For any non-empty path p in G which starts and ends at a child of u and contains only proper descendants of u as intermediate vertices, there is a unique sequence of indices $1 \le i_1 < i_2 < \ldots < i_k \le \ell$ and a unique partition of p into non-empty paths $p = p_1,p_2,\ldots,p_k$ such that $p_j \in \sigma(P_{i_j})$ for $1 \le i \le k$.

Proof. Straightforward by induction on the number of times the loop in DECOMPOSE_AND_SEQUENCE is executed. □

Theorem 9. Procedure DECOMPOSE_AND_SEQUENCE correctly computes a path sequence for G .

Proof. Analogous to the proof of Theorem 8. □

DECOMPOSE_AND_ELIMINATE thus provides a way to compute path sequences in arbitrary graphs. The running time of the method is $O(m\,\alpha(m,n)+t)$ if stratified path compression is used to implement the forest operations and $O((m \log n)+t)$ if path compression is used, where t is the time to find path sequences for the dominator strong components of G . The length of the path sequence produced is either $O(m\,\alpha(m,n))+\ell$ or $O(m \log n)+\ell$, where ℓ is the total length of the path sequences for the dominator strong components.

7. Remarks.

In this paper we have described fast algorithms for solving path expression problems on reducible or almost-reducible graphs. The fastest method requires $O(m\,\alpha(m,n)+t)$ time to compute a path sequence for an arbitrary directed graph, where t is the amount of time required to compute path sequences for the dominator strong components. A slower but much simpler method requires $O(m \log n + t)$ time and promises to be easy to program and efficient in practice.

By using our algorithms in combination with the mapping technique described by Tarjan[30], we can solve many kinds of path problems, including finding shortest paths, carrying out forward and backward global flow analysis, and solving sparse systems of linear equations. There are two rather different ways of doing this. The first is to use the solution to a path expression problem as a general-purpose straight-line program which solves any particular path problem by properly interpreting $\cup$, $\cdot$, and $*$. The second is to use an algorithm for solving a path expression problem to solve a particular path problem by reinterpreting $\cup$, $\cdot$, and $*$ within the algorithm; this avoids the intermediate step of first constructing a directed acyclic graph representing a set of path expressions. The choice between these two methods depends upon the time and space available and whether we want to solve one or many path problems on the same graph.

For path problems in which the operation corresponding to + is idempotent, the non-redundancy and uniqueness conditions in (6) and Theorem 1 are not necessary and can be dropped [30]. In such cases we can use the sophisticated algorithm of Tarjan [29] to carry out the

forest manipulation operations and achieve an $O(m\,\alpha(m,n)+t)$ time bound [27]. It does not seem possible to adapt this method to satisfy non-redundancy, however. The only interesting path problem known to the author which does not have the idempotent property is the solution of sparse systems of linear equations. For this problem another form of tree manipulation described by Tarjan [29] gives a rather simple $O(m\,\alpha(m,n)+t)$ -time algorithm [28].

The method of decomposition by dominators is a kind of single-element "tearing" [5] in which the clever use of data structures allows us to make the combining step very efficient. The result may be generalizable in various directions. For instance, on problem graphs for which there is no natural start vertex we would like to know how to pick a start vertex which gives the finest decomposition. It may also be possible to extend the technique to regions with two or more entry vertices. We leave these questions to the ambitious reader.

Appendix: Graph Theoretic Terminology.

A directed graph $G = (V,E)$ is a finite set V of vertices and a finite set E of edges such that each edge e has a head $h(e) \in V$ and a tail $t(e) \in V$. We regard the edge e as leading from $h(e)$ to $t(e)$, and we say the edge e leaves $h(e)$ and enters $t(e)$. We usually denote the number of vertices by n and the number of edges by m. A loop is an edge e with $h(e) = t(e)$. A path $p = e_1, e_2, \ldots, e_k$ is a sequence of edges such that $t(e_i) = h(e_{i+1})$ for $1 \leq i \leq k-1$. The path is from $h(p) = h(e_1)$ to $t(p) = t(e_k)$. The path contains edges $e_1, e_2, \ldots, e_k$ and vertices $h(e_1), h(e_2), \ldots, h(e_k), t(e_k)$ and avoids all other edges and vertices. There is a path of no edges from any vertex to itself. A cycle is a non-empty path from a vertex to itself. A graph is acyclic if it contains no cycles.

The reverse G^r of a graph G is the graph formed by replacing each edge e with an edge e^r such that $h(e^r) = t(e)$ and $t(e^r) = h(e)$. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are graphs, G_1 is a subgraph of G_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. G_1 is the subgraph of G_2 induced by V_1 if $V_1 \subseteq V_2$ and $E_1 = \{e \in E_2 \mid h(e), t(e) \in V_1\}$.

A vertex v is reachable from a vertex w in a graph G if there is a path from v to w. G is strongly connected if every vertex is reachable from every other vertex. The strong components of G are its maximal strongly connected subgraphs. These components are uniquely defined and partition the vertices of G.

A flow graph $G = (V,E,r)$ is a graph with a distinguished start vertex r such that every vertex is reachable from r. A (directed, rooted) tree $T = (V,E,r)$ is a flow graph with $|E| = |V|-1$. The start

vertex r is the root of the tree. Any tree is acyclic, and if v is any vertex in T , there is a unique path from r to v . If v and w are vertices in a tree T and there is a path from v to w , V is an ancestor of w and w is a descendant of v . We denote this relationship by $v \overset{*}{\rightarrow} w$. If in addition $v \neq w$, v is a proper ancestor of w and w is a proper descendant of v , denoted by $v \overset{+}{\rightarrow} w$. If there is an edge from v to w , v is the parent of w and w is a child of v , denoted by $v \rightarrow w$. Two vertices with a common parent are siblings. In a tree each vertex has a unique parent (except the root, which has no parent).

References

[1] A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Volume II: Compiling, Prentice-Hall, Englewood Cliffs, N.J. (1972), 915.

[2] A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977, 408-517.

[3] F. E. Allen, "Control flow analysis," SIGPLAN Notices 5, 7 ((1970), 1-19.

[4] R. C. Backhouse and B. A. Carré, "Regular algebra applied to path-finding problems," J. Inst. Maths. Applics. 15 (1975), 161-186.

[5] J. R. Bunch and D. J. Rose, "Partitioning, tearing, and modification of sparse linear systems," J. Math. Analysis and Applics. 48 (1974), 574-593.

[6] B. A. Carré, "An algebra for network routing problems," J. Inst. Math. Applics. 7 (1971), 273-294.

[7] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[8] I. s. Duff, "A survey of sparse matrix research," Proc. IEEE 65 (1977), 500-535.

[9] R. Farrow, "Efficient variants of path compression on unbalanced trees," unpublished manuscript, 1978.

[10] R. Floyd, "Algorithm 97: shortest path," Comm. ACM 5 (1962), 345.

[11] G. E. Forsythe and C. B. Moler, Computer Solution of Linear Algebraic Equations, Prentice-Hall, Englewood Cliffs, N.J.,1967.

[12] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," Journal ACM 23 (1976), 172-202.

[13] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier, New York, [illegible].

[14] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," SIAM J. Comput. 1 (1972), 188-202.

[15] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," Journal ACM 21 (1974), 367-375.

[16] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," Journal ACM 24 (1977), 1-13.

[17] K. W. Kennedy, "Node listings applied to data flow analysis," Conf. Record of the Second ACM Symp. on Principles of Prog. Lang. (1975),10-21.

[18] S. C. Kleene, "Representation of events in nerve nets and finite automata," Automata Studies, C. Shannon and J. McCarthy, eds., Princeton University Press, Princeton, N. J., 1956, 3-40.

[19] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968, 258-265.

[20] D. E. Knuth, "An empirical study of FORTRAN programs," Software Practice and Experience 1 (1971), 105-133.

[21] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in flow graphs," Trans. on Prog. Lang. and Systems 1 (1979), to appear.

[22] D. J. Rose, A. H. Sherman, R. E. Tarjan, and G. F. Whitten, "Algorithms and software for in-core factorization of sparse symmetric positive definite matrices," Computers and Structures 10 (1979), 411-418.

[23] M. Schaefer, A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[24] R. E. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput. 1 (1972), 146-160.

[25] R. Tarjan, "Finding dominators in directed graphs," SIAM J. Comput, 3 (1974), 62-89.

[26] R. E. Tarjan, "Testing flow graph reducibility," J. Comp. and Sys, Sciences 9(1974), 355-365.

[27] R. E. Tarjan, "Solving path problems on directed graphs," Technical Report STAN-CS-75-528, Computer Science Department, Stanford University, 1975.

[28] R. E. Tarjan, "Graph theory and Gaussian elimination," Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, 3-22.

[29] R. E. Tarjan, "Applications of path compression on balanced trees," Journal ACM, to appear.

[30] R. E. Tarjan, "A unified approach to path problems," Technical Report STAN-CS-79-729, Computer Science Department, Stanford University, 1979; also Journal ACM, submitted.

[31] J. D. Ullman, "Fast algorithms for the elimination of common subexpressions," Acta Informatica 2 (1973), 191-213.

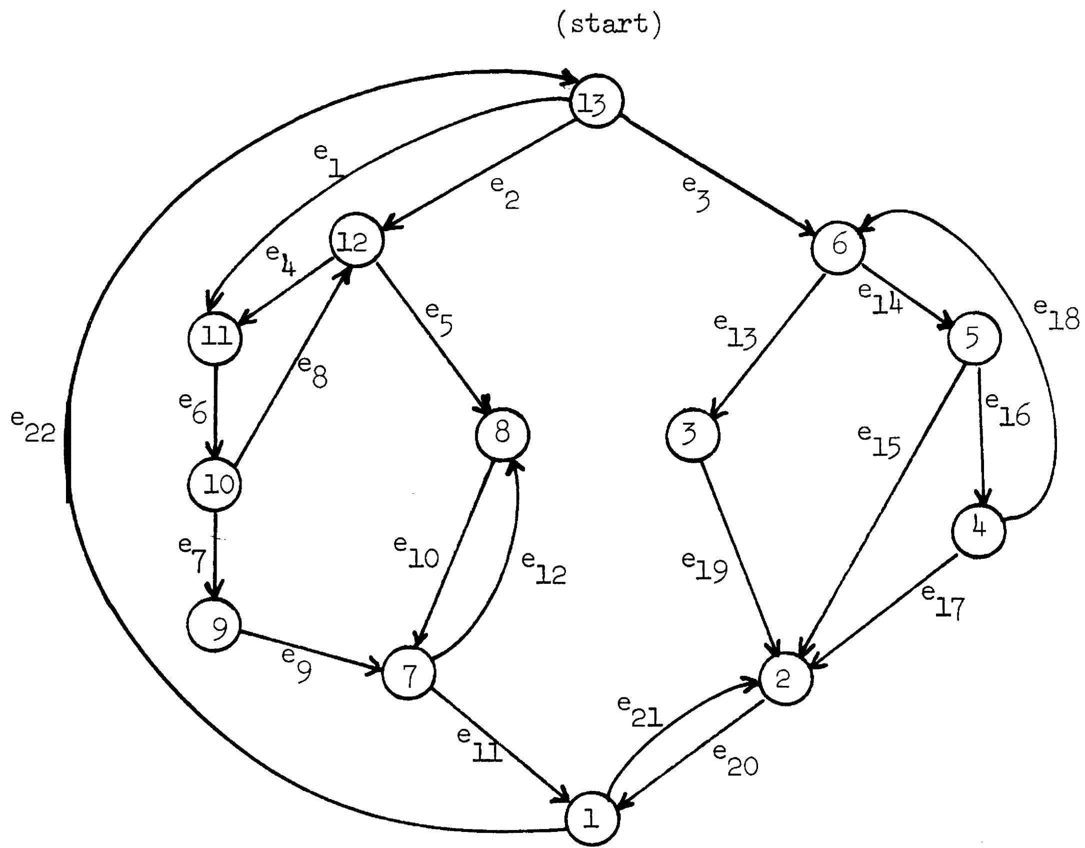


Figure 1. A flow graph G ,

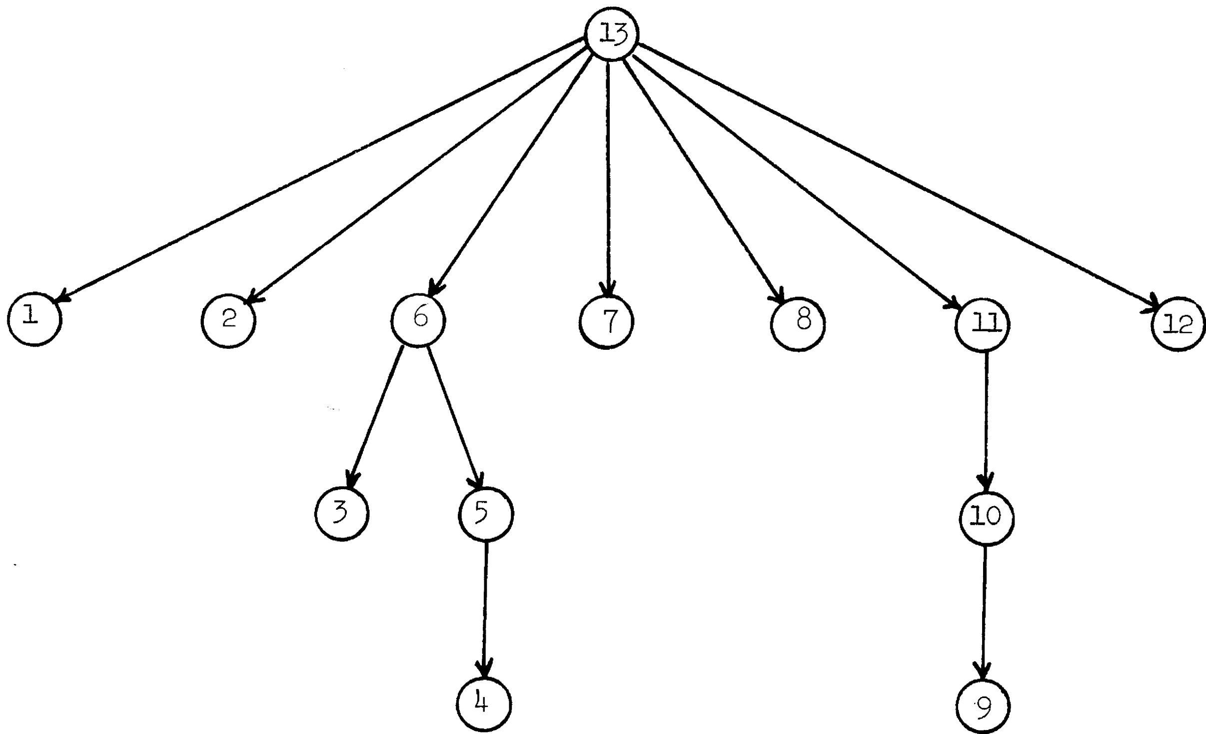


Figure 2. The dominator tree of G .

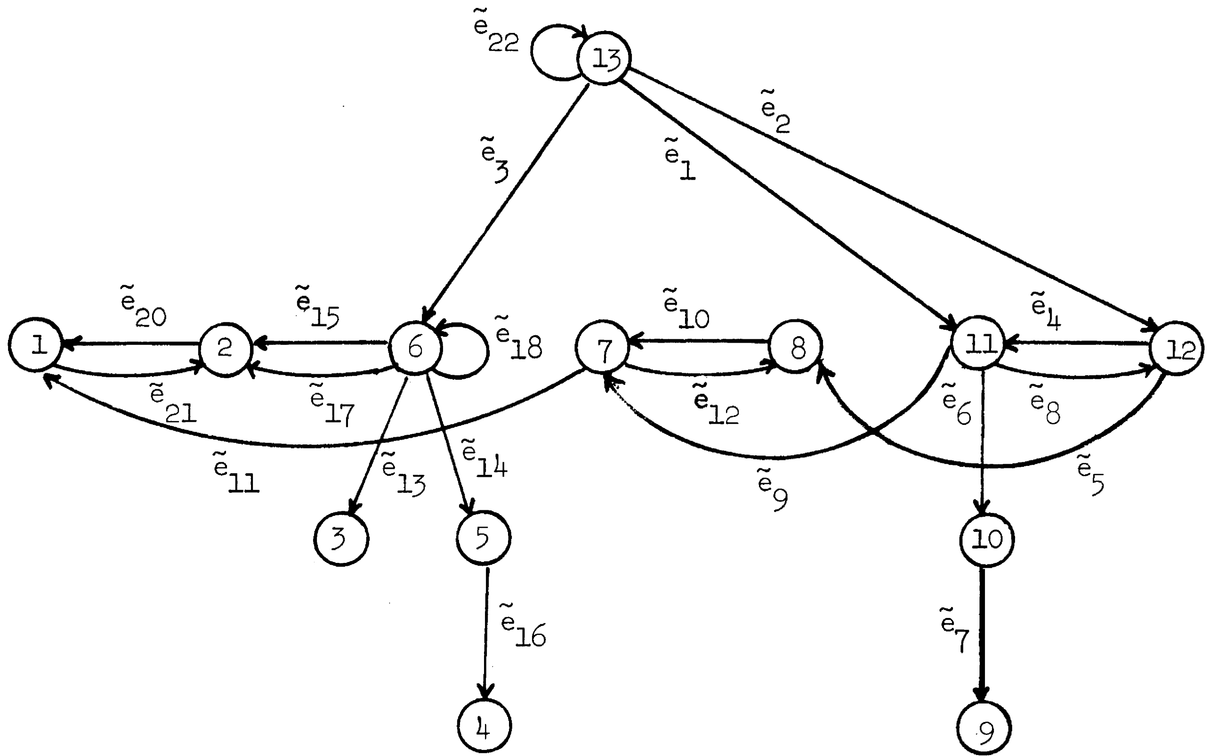


Figure 3. The derived graph of G . The vertex sets of the dominator strong components are {1,2} , {3} , {4} , {5} , {6} , {7,8} , {9} , {10} , {11,12} , {13} .