A UNIFIED APPROACH TO PATH PROBLEMS

by

Robert Endre Tarjan

STAN-CS-79-729
April 1979

C O M P U T E R S C I E N C E D E P A R T M E N T
School of Humanities and Sciences
STANFORD UNIVERSITY

A Unified Approach to Path Problems

Robert Endre Tarjan*/

Computer Science Department
Stanford University
Stanford, California 94305

April, 1979

Abstract. We describe a general method for solving path problems on directed graphs. Such path problems include finding shortest paths, solving sparse systems of linear equations, and carrying out global flow analysis of computer programs. Our method consists of two steps, First, we construct a collection of regular expressions representing sets of paths in the graph. This can be done by using any standard algorithm, such as Gaussian or Gauss-Jordan elimination, Next, we apply a natural mapping from regular expressions into the given problem domain. We exhibit the mappings required to find shortest paths, solve sparse systems of linear equations, and carry out global flow analysis. Our results provide a general-purpose algorithm for solving any path problem, and show that the problem of constructing path expressions is in some sense the most general path problem,

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

Keywords: code optimization, compiling, Gaussian elimination, Gauss-Jordan elimination, global flow analysis, graph algorithm, linear algebra, path problem, regular expression, semi-lattice, 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 contract N00014-76-C-0688, and by a Guggenheim Fellowship, Reproduction in whole or in part is permitted for any purpose of the United States government,

A Unified Approach to Path Problems

1. Introduction.

A fundamental problem in numerical analysis is the solution of a system of linear equations $Ax = b$, where A is an $n \times n$ matrix of coefficients, x is an $n \times 1$ vector of variables, and b is an $n \times 1$ vector of constants. Efficient methods for solving $Ax = b$, such as Gaussian and Gauss - Jordan elimination, have long been known. These methods have been repeatedly rediscovered and applied in other contexts. For example, Floyd's shortest path algorithm [7], which is based on Warshall's transitive closure algorithm [32], is a version of Gauss -Jordan elimination. Kleene's method for converting a finite automaton into a regular expression [20] is a form of Gauss -Jordan elimination; Gaussian elimination also solves this problem [3]. In all these situations the problem of interest can be formulated as the solution of a system of linear equations defined not over the field of real numbers but over some other algebra.

In this paper we provide a unified setting for such problems. Our goal is to show that a solution to one of them can be used to solve them all. One approach to this task is to develop a minimal axiom system for which elimination techniques work (see for instance Aho, Hopcroft, and Ullman [1] and Lehman [21]) and to show that the problems of interest satisfy the axioms. Our approach is somewhat different and resembles that taken by Backhouse and Carré [3]; we believe that the proper setting for such problems is the algebra of regular expressions, which is simple, well-understood, and general enough for our purposes.

We shall use a graph-theoretic approach rather than a matrix' theoretic one because we are interested mainly in sparse problems (problems in which the coefficient matrix A contains mostly zeros), Let G be a directed graph with a distinguished source vertex s , The single-source path expression problem is to find, for each vertex v in G , a regular expression R(s,v) representing the set of all paths from s to v . The all-pairs path expression problem is to find, for each pair of vertices v , w , a regular expression R(v,w) representing the set of all paths from v to w . We shall show that it is possible to use solutions to the single-source and all-pairs path expression problems to find shortest paths in G , to solve systems of linear equations defined on G , and to solve global flow problems defined on G . We solve these problems by providing natural homomorphism that map the regular expressions representing path sets into the algebras in which the given problems are expressed. We define these mappings by reinterpreting the $\cup$, . and $*$ operations used to construct regular expressions. The technical part of our work is in showing that these mappings are indeed homomorphisms.

This paper contains nine sections. Section 2 reviews the properties of regular expressions that we shall use. Section 3 considers shortest path problems. Section 4 examines the solution of systems of linear equations over the real numbers. Sections 5,6,7, and 8 discuss various kinds of global flow analysis problems. Section 9 contains some additional remarks. The appendix contains the graph-theoretic definitions used in the paper.

2. Regular Expressions and Path Expressions.

Let Σ be a finite alphabet containing neither " Λ " 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 defines a set $\sigma(R)$ of strings over Σ as follows:

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

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

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

$\sigma(R_1^*) = \bigcup_{k=0}^{\infty} \sigma(R_1)^k$, where $\sigma(R_1)^0 = \{\Lambda\}$ and $\sigma(R_1)^i = \sigma(R_1)^{i-1} \cdot \sigma(R_1)$.

Two regular expressions R_1 and R_2 are equivalent if $\sigma(R_1) = \sigma(R_2)$. A regular expression R is simple if $R = \emptyset$ or R does not contain $\emptyset$ as a subexpression. We can transform any regular

---

*/ Note that the symbol Λ represents both the regular expression " Λ " and the empty string. Henceforth we shall avoid using quotation marks and allow the context to resolve this ambiguity; similarly for $\emptyset$, $\cup$, $\cdot$, $*$. We shall also freely omit parentheses in regular expressions when the meaning is clear.

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 Λ .

A regular expression R is non-redundant if each string in d(R) is represented uniquely in R . A more precise definition is as follows:

(3a) Λ , $\emptyset$, and a for $a \in \Sigma$ are non-redundant.

(3b) 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 non-empty $w \in R_1^*$ is uniquely decomposable into $w = w_1 w_2 \cdots w_k$ with $w_i \in \sigma(R_1)$ for $1 \leq i \leq k$.

Note that if $\Lambda \in \sigma(R)$, then R^* is redundant.

Let G = (V,E) be a directed graph. We can regard any path in G as a string over E , but not all strings over E are paths in G . A path expression P of type (v,w) is a simple regular expression over E such that every string in c(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.

(4) 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 $P = P_1 \cdot P_2$, 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.

3. Shortest Paths.

Let $G = (V,E)$ be a directed graph with an associated real-valued cost $c(e)$ for each edge e. A shortest path from v to w is a path $p = e_1, e_2, \ldots, e_k$ from v to w such that $\sum_{i=1}^{k} c(e_i)$ is minimum over all paths from v to w. If G contains no cycles of negative total cost, there is a shortest path from v to w if there is any path from v to w. The single-source shortest path problem is to find, for each vertex v, the cost of a shortest path from s to v, where s is a distinguished source vertex. The all-pairs shortest path problem is to find the cost of a shortest path from v to w for all vertex pairs v, w.

We can use path expressions to solve shortest path problems by means of two mappings, cost and shortest path, defined as follows.

(5a) $\underline{\text{cost}}(\Lambda) = 0$, $\underline{\text{shortest path}}(A) = \Lambda$;

$\underline{\text{cost}}(\emptyset) = \infty$, $\underline{\text{shortest path}}(\$) = \text{no path}$;

$\underline{\text{cost}}(e) = c(e)$, $\underline{\text{shortest path}}(e) = e$ for $e \in E$.

(5b) $\underline{\text{cost}}(P_1 \cup P_2) = \min\{\underline{\text{cost}}(P_1), \underline{\text{cost}}(P_2)\}$,

$\underline{\text{shortest path}}(P_1 \cup P_2) = \underline{\text{if}}\ \underline{\text{cost}}(P_1) \leq \underline{\text{cost}}(P_2)\ \underline{\text{then}}\ \underline{\text{shortest path}}(P_1)$ $\underline{\text{else}}\ \underline{\text{shortest path}}(P_2)$;

$\underline{\text{cost}}(P_1 \cdot P_2) = \underline{\text{cost}}(P_1) + \underline{\text{cost}}(P_2)$,

$\underline{\text{shortest path}}(P_1 \cdot P_2) = \underline{\text{shortest path}}(P_1) \cdot \underline{\text{shortest path}}(P_2)$;

$\underline{\text{cost}}(P_1^*)$ $\underline{\text{if}}\ \underline{\text{cost}}(P_1) < 0\ \underline{\text{then}}\ -\infty\ \underline{\text{else}}\ 0$,

$\underline{\text{shortest path}}(P_1^*)$ $\underline{\text{if}}\ \underline{\text{cost}}(P_1) < 0\ \underline{\text{then}}$ no shortest path $\underline{\text{else}}\ \Lambda$.

Lemma 1. Let P be a path expression of type (v,w). If $\underline{cost}(P) = \infty$, there is no path in $c(P)$. If $\underline{cost}(P) = -\infty$, there are paths of arbitrarily small cost in $\sigma(P)$. Otherwise, $\underline{shortest\ path}(P)$ is a minimum cost path in $\sigma(P)$, and the cost of $\underline{shortest\ path}(P)$ is $\underline{cost}(P)$.

Proof. Straightforward by induction on the number of operation symbols in P. □

Theorem 1. Let $P(v,w)$ be a path expression representing all paths from v to w. If $\underline{cost}(P(v,w)) = \infty$, there is no path from v to w. If $\underline{cost}(P(v,w)) = -\infty$, there are paths of arbitrarily small cost from v to w. Otherwise, $\underline{shortest\ path}(P(v,w))$ is a shortest path from v to w; the cost of this path is $\underline{cost}(P(v,w))$.

Proof. Immediate from Lemma 1. □

Theorem 2. Let $P_1(v,w)$ be a path expression such that $\sigma(P_1(v,w))$ contains at least all the simple paths from v to w. If there is a shortest path from v to w, $\underline{shortest\ path}(P(v,w))$ gives one such path; its cost is $\underline{cost}(P(v,w))$.

Proof. Any shortest path is simple. □

By applying Theorem 1 we can use a solution to the single-source (or all-pairs) path expression problem to solve the single-source (or all-pairs) shortest path problem. By Theorem 2 it is sufficient to use path expressions representing only the simple paths if we have a separate test for negative cycles. The following theorem provides such a test.

Theorem 3. Let s be a distinguished source vertex in G . For' every vertex v , let $P_1(s,v)$ be a path expression such that $\sigma(P_1(s,v))$ contains at least all the simple paths from s to v. Then G contains a negative cycle if and only if there is some edge e such that $\underline{cost}(P_1(s,h(e)) + e(e) < \underline{cost}(P_1(s,t(e)))$.

Proof. Straightforward. See Ford and Fulkerson [10]. □

4. Systems of Linear Equations.

The next problem to which we shall apply our technique is the solution of a system $Ax = b$ of linear equations over the set $\mathbb{R}$ of real numbers [11]. This problem has pitfalls not present in the other problems we examine. The system $Ax = b$ does not always have a solution; even if it does, the solution need not be unique. Furthermore the standard algorithms for finding a solution, such as Gaussian elimination, may not succeed even if a unique solution exists. (To deal with this difficulty, numerical analysts have devised more complicated algorithms, such as Gaussian elimination with pivoting [11].) We shall avoid these issues by proposing a method that almost always gives a solution when one exists.

We begin by rewriting $Ax = b$ as $-b + (A-I)x = x$, where I is the $n \times n$ identity matrix. Let x_0 be a new variable; then the system $-b + (A-I)x = x$ is equivalent to

$$\begin{pmatrix} 1 \\ \bar{0} \end{pmatrix} + A' \begin{pmatrix} x_0 \\ x \end{pmatrix} = \begin{pmatrix} x_0 \\ x \end{pmatrix} \quad , \quad \text{where } A' = \begin{pmatrix} 0 & \bar{0} \\ -b & A-I \end{pmatrix}$$

and $\bar{0}$ denotes a zero matrix of the appropriate size. Let $G = (V,E)$ be the graph having $n+1$ vertices (one for each variable x_i) and m edges (one for each non-zero entry in A') such that there is an edge e with $h(e) = v_j$ and $t(e) = v_i$ if and only if the entry in row i and column j of A' is non-zero; let $a(e)$ be the value of this entry. Then the system of equations takes the form

$$(6) \qquad x(s) = 1 \ ; \ x(v) = \sum \{a(e)x(h(e)) \mid e \in E \text{ and } t(e) = v\} \text{ if } v \neq s \ ,$$

where $s = v_{\square}$.

We solve this system by extending the mapping a to regular expressions over E as follows.

(7a) $a(A) = 1 \; ; \; a(\emptyset) = 0 \; .$

(7b) $a(R_1 \cup R_2) = a(R_1) + a(R_2) \; ;$

$a(R_1 \cdot R_2) = a(R_1)a(R_2) \; ;$

$a(R_1^*) = 1/(1 - a(R_1)) \; .$

Note that $a(R_1^*)$ is defined if and only if $a(R_1) \neq 1$. If R is a regular expression over E , then a(R) is a rational function of $a(e_1), a(e_2), \ldots \boxtimes a(e_m)$ and is defined except on a set of measure zero in $\mathbb{R}^m$. Note also that the operation of addition into which union is mapped is not idempotent. This forces us to deal only with non-redundant regular expressions.

Lemma 2. If R_1 and R_2 are two equivalent non-redundant regular expressions over E , then $a(R_1) = a(R_2)$ whenever both $a(R_1)$ and $a(R_2)$ are defined.

Lemma 2 is the hardest result in this paper, and we shall postpone its proof.

Theorem 4. For each vertex v , let P(s,v) be a non-redundant path expression representing all paths from s to v . If a(P(s,v)) is defined for all v , then the mapping x defined by x(v) = a(P(s,v)) satisfies (6).

Proof. The only path from s to s in G is the empty path; by Lemma 2, $x(s) = a(P(s,s)) = a(A) = 1$. If $v \neq s$, then $\bigcup \{P(s,h(e))\cdot e \mid e\epsilon E \text{ and } t(e) = v\}$ is a non-redundant regular expression representing the set of all paths from s to v. By Lemma 2,

$$x(v) = a(P(s,v)) = a(\bigcup \{P(s,h(e))\cdot e \mid e\epsilon E \text{ and } t(e) = v\}$$

$$= \sum \{a(e)x(h(e)) \mid e\epsilon E \text{ and } t(e) = v\} . \quad \square$$

Thus the mapping a almost always gives a solution to (6). It remains for us to prove Lemma 2. We employ Salomaa's method for showing the completeness of an axiom system for regular expressions [28]. We shall use the notation $Q \equiv R$ to denote that $\sigma(Q) = \sigma(R)$ and $a(Q) = a(R)$ wherever both $a(Q)$ and $a(R)$ are defined. A non-redundant regular expression Q is equationally characterized in terms of non-redundant regular expressions $Q_1, Q_2, \ldots, Q_q$ if $Q = Q_1$ and

$$(8) \qquad Q_i \equiv \left(\bigcup_{j=1}^{m} Q_{ij} \cdot e_j \right) \cup D(Q_i) \quad \text{where } D(Q_i) \in \{\emptyset, \Lambda\} \text{ and}$$

$$Q_{ij} \in \{Q_k \mid 1 \leq k \leq q\} \text{ for all } j .$$

Lemma 3. Every non-redundant regular expression over E is equationally characterized.

Proof. By induction on the number of operation symbols in the regular expression.

$$\emptyset \equiv \left(\bigcup_{j=1}^{m} \emptyset \cdot e_j \right) \cup \emptyset \quad ; \quad \Lambda \equiv \left(\bigcup_{j=1}^{m} \emptyset \cdot e_j \right) \cup \Lambda \quad ;$$

$$e_j \equiv \emptyset \cdot e_1 \cup \ldots \cup \Lambda \cdot e_j \cup \ldots \cup \emptyset \cdot e_m \cup \emptyset \qquad \text{for } 1 \leq j \leq m \ .$$

Thus every atomic regular expression is equationally characterized.

Suppose Q and R are equationally characterized. Let $Q_1, \ldots, Q_q$ be non-redundant regular expressions such that $Q = Q_1$ and (8) holds. Let $R_1, \ldots, R_r$ be non-redundant regular expressions such that $R = R_1$ and (9) holds.

$$(9) \qquad R_i \equiv \left(\bigcup_{j=1}^{m} R_{ij} \cdot e_j \right) \cup D(R_i) \text{ where } D(R_i) \in \{\emptyset, \Lambda\} \text{ and } R_{ij} \in \{R_k \mid 1 \leq k \leq r\} \text{ for all } j \ .$$

We shall equationally characterize $Q \cup R$, $Q \cdot R$, and Q^*, assuming they are non-redundant.

Let $1 \leq u \leq q$, $1 \leq v \leq r$, and suppose $Q_u \cup R_v$ is non-redundant. Combining (8) and (9) we obtain

$$(10) \qquad Q_u \cup R_v \equiv \left(\bigcup_{j=1}^{m} (Q_{uj} \cup R_{vj}) \cdot e_j \right) \cup D(Q_u) \cup D(R_v)$$

$$\equiv \left(\bigcup_{j=1}^{m} (Q_{uj} \cup R_{vj}) \cdot e_j \right) \cup D(Q_u \cup R_v) \ ,$$

since if $\sigma(Q_u) \cap \sigma(R_v) = \emptyset$, then $D(Q_u) = \emptyset$ or $D(R_v) = \emptyset$. Furthermore $Q_{uj} \cup R_{vj}$ is non-redundant for $1 \leq j \leq m$. Thus if $Q \cup R$ is non-redundant, the set of equations (10) such that $Q_u \cup R_v$ is non-redundant equationally characterizes $Q \cup R = Q_1 \cup R_1$.

Let $1 \leq v \leq r$, $s > 0$, and $1 \leq u_1 < u_2 < \ldots < u_s \leq q$. Suppose $Q \cdot R_v \cup \left(\bigcup_{i=1}^{s} Q_{u_i} \right)$ is non-redundant. If $D(R_v) = \emptyset$, we obtain from (8) and (9) that

$$(11)\quad Q\cdot R_v \cup \left(\bigcup_{i=1}^{s} Q_{u_i}\right) \equiv \left(\bigcup_{j=1}^{m}\left(Q\cdot R_{vj} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j}\right)\right)\cdot e_j\right) \cup \left(\bigcup_{i=1}^{s} D(Q_{u_i})\right)$$

$$\equiv \left(\bigcup_{j=1}^{m}\left(Q\cdot R_{vj} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j}\right)\right)\cdot e_j\right) \cup D\left(Q\cdot R_v \cup \left(\bigcup_{i=1}^{s} Q_{u_i}\right)\right).$$

Furthermore $Q\cdot R_{vj} \cup \left(\bigcup_{i=1}^{s} R_{u_i j}\right)$ is non-redundant for $1 \le j \le m$. If $D(R) = A$, we obtain from (8) and (9) that

$$(12)\quad Q\cdot R_v \cup \left(\bigcup_{i=1}^{s} Q_{u_i}\right) \equiv \left(\bigcup_{j=1}^{m}\left(Q\cdot R_{vj} \cup Q_{1j} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j}\right)\right)\cdot e_j\right) \cup D(Q_1) \cup \left(\bigcup_{i=1}^{s} D(Q_{u_i})\right)$$

$$\equiv \left(\bigcup_{j=1}^{m}\left(Q\cdot R_{vj} \cup Q_{1j} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j}\right)\right)\cdot e_j\right) \cup D\left(Q\cdot R_v \cup \left(\bigcup_{i=1}^{s} Q_{u_i}\right)\right).$$

Furthermore $Q\cdot R_{vj} \cup Q_{1j} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j}\right)$ is non-redundant for $1 \le j \le m$.

It follows that if $Q\cdot R$ is non-redundant, we can equationally characterize $Q\cdot R = Q\cdot R_1$ in terms of $\left\{ Q\cdot R_v \cup \left(\bigcup_{i=1}^{q} Q_{u_i}\right) \mid 1 \le v \le r,\ s \le 0,\ 1 \le u_1 < u_2 < \ldots < u_s \le q,\right.$ and $\left. Q\cdot R_v \cup \left(\bigcup_{i=1}^{q} Q_{u_i}\right) \text{ is non-redundan} \right\}$.

Finally we must consider Q^*. Suppose Q^* is non-redundant. Then $D(Q) = \emptyset$. From (8) we obtain

(13) $$Q^* \equiv \left(\bigcup_{j=1}^{m} Q_{1j} \cdot e_j \right)^*$$

$$\equiv \left(\bigcup_{j=1}^{m} Q^* \cdot Q_{1j} \cdot e_j \right) \cup \Lambda \quad .$$

Furthermore $Q^* \cdot Q_{1j}$ is non-redundant for $1 < j < m$.

Let $s \geq 1$ and $1 < u_1 < u_2 < \ldots < u_s < q$. Suppose $Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i} \right)$ is non-redundant. If $D(Q_{u_i}) = \emptyset$ for $1 \leq i \leq s$, then

(14) $$Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i} \right) \equiv \left(\bigcup_{j=1}^{m} Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i j} \right) \cdot e_j \right) \cup \emptyset \quad ,$$

where $Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i j} \right)$ is non-redundant for $1 \leq j \leq m$.

If $D(Q_{u_i}) = \Lambda$ for some (unique) i such that $1 \leq i \leq s$, then

(15) $$Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i} \right) \equiv \left(\bigcup_{j=1}^{m} Q^* \cdot \left(Q_{1j} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j} \right) \right) \cdot e_j \right) \cup \Lambda \quad ,$$

where $Q^* \cdot \left(Q_{1j} \cup \left(\bigcup_{i=1}^{s} Q_{u_i j} \right) \right)$ is non-redundant for $1 \leq j \leq m$. It follows that we can equationally characterize Q* in terms of

$$\{Q^*\} \cup \left\{ Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i} \right) \mid s > 1,\ 1 \leq u_1 < u_2 < \ldots < u_s \leq q, \text{ and } Q^* \cdot \left(\bigcup_{i=1}^{s} Q_{u_i} \right) \text{ is non-redundant} \right\} \quad . \quad \square$$

We are now ready to prove Lemma 2, We extend $\cup$, ., $\equiv$ to ordered pairs of regular expressions by defining $(Q_1, R_1) \cup (Q_2, R_2) = (Q_1 \cup Q_2, R_1 \cup R_2)$, $(Q_1, R_1) \cdot (Q_2, R_2) = (Q_1 \cdot Q_2, R_1 \cdot R_2)$, $(Q_1, R_1) \equiv (Q_2, R_2)$ if and only if $Q_1 \equiv Q_2$ and $R_1 \equiv R_2$.

Proof of Lemma 2. Suppose Q and R are non-redundant regular expressions such that $\sigma(Q) = \sigma(R)$. Let Q, R be characterized in terms of $\{Q_i \mid 1 \leq i \leq q\}$, $\{R_i \mid 1 \leq i \leq r\}$ by (8), (9), respectively. We construct a set X of pairs (Q_u, R_v) such that $\sigma(Q_u) = \sigma(R_v)$. We begin with $X = \{(Q,R)\}$. We process pairs in X and add new elements to X until all pairs in X are processed. We process a pair (Q_u, R_v) as follows. By (15) and (16) we have

$$(Q_u, R_v) \equiv \left(\bigcup_{j=1}^{m} (Q_{uj}, R_{vj}) \cdot (e_j, e_j) \right) \cup (D(Q_u), D(R_v)) \quad .$$

Since $\sigma(Q_u) = \sigma(R_v)$, we have $D(Q_u) = D(R_v)$ and $a(Q_{uj}) = \sigma(R_{vj})$ for $1 \leq j \leq m$. We add each pair (Q_{uj}, R_{vj}) for $1 \leq j \leq m$ to X if it is not already present.

We obtain a set of pairs $X = \{(Q^{(1)}, R^{(1)}), \ldots, (Q^{(s)}, R^{(s)})\}$ such that $s \leq qr$, $Q^{(i)} \equiv R^{(i)}$ for $1 \leq i \leq s$, and $(Q^{(i)}, R^{(i)}) \equiv \bigcup_{j=1}^{m} (Q_j^{(i)}, R_j^{(i)}) \cdot (e_j, e_j) \cup (D_i, D_i)$, where each pair $(Q_j^{(i)}, R_j^{(i)})$ appears in X.

Consider the system of equations $x_i = \sum_{j=1}^{m} a(e_j) x_{ij} + a(D_i)$, where $x_{ij} = x_k$ if $Q_j^{(i)} = Q^{(k)}$. This system is satisfied by $x_1 = a(Q^{(i)})$ if $a(Q^{(i)})$ is defined for $1 \leq i \leq s$ and by $x_i = a(R^{(i)})$ if $a(R^{(i)})$ is defined for $1 < i < s$. We can rewrite this system as $x = Ax + b$, where each entry in A is a linear combination of $a(e_1), a(e_2), \ldots, a(e_m)$, or equivalently as $(A-I)x = -b$. This system has a unique solution when the determinant of A-I is non-zero, which is true except for values of $a(e_1), a(e_2), \ldots, a(e_m)$ forming a set of

measure zero in $\mathbb{R}^m$. Thus $a(Q^{(i)}) = a(R^{(i)})$ for $1 \leq i < s$ except on a set of measure zero. In particular $a(Q) = a(R)$ except on a set of measure zero. Since $a(Q)$ and $a(R)$ are rational functions of the $a(e_j)$'s, $a(Q) = a(R)$ when both are defined. □

5. Continuous Data Flow Problems.

Many problems in global code optimization can be formulated as path problems of the kind we are considering. The general setting is as follows. We represent a computer program by a flow graph $G = (V,E,s)$. Each vertex represents a basic block of the program (a block of consecutive statements having a single entry and a single exit). Each edge represents a possible transfer of control between basic blocks. The start vertex s represents the start of the program. We are interested in determining, for each basic block, facts which must be true on entry to the block regardless of the actual path of program execution. Such facts can be used for various kinds of code optimization. See Aho and Ullman [2], Hecht [14], and Shaefer [25].

To represent the universe of possible program facts, we use a set L having a commutative, associative, idempotent meet operation $\wedge$; such an algebraic structure is called a lower semi-lattice. If x and y are two possible program facts, $x \wedge y$ represents the information common to both. We can define a relation $\leq$ on L by $x < y$ if and only if $x \wedge y = x$. The properties of $\wedge$ imply that $<$ is a partial order on L [27]; we interpret $x \leq y$ to mean that fact y contains more information than fact x. We shall assume that L is complete, by which we mean that every subset $X \subset L$ has a greatest lower bound with respect to $\leq$; we denote this greatest lower bound by $\wedge X$. If $X = \{x_1, x_2, \ldots, x_n\}$, then $\wedge X = x_1 \wedge x_2 \wedge \ldots \wedge x_n$. We use $\perp$ to denote $\wedge L$, i.e., the minimum element in L. For any functions f and g having common domain and range L, we define $f < g$ if and only if $f(x) \leq g(x)$ for all elements x in the domain of f and g,

To represent the effect of the program on the universe of facts, we associate with each edge e a function f_e such that, if fact x is true on entry to $h(e)$ and control passes through edge e, then $f_e(x)$ will be true on entry to $t(e)$. We can extend these functions to paths by defining $f_p(x) = x$ if p is the empty path, $f_p(x) = (f_{e_k} \text{ of } e_{k-1} \circ \cdots \circ f_{e_1})(x)$ if $p = e_1, e_2, \ldots, e_k$. What we want to compute is $\wedge \{f_p(\perp) \mid p \text{ is a path from } s \text{ to } v\}$ for each vertex v. (We assume the minimum fact $\perp$ is true on entry to the program.)

This discussion motivates the following definitions. A <u>continuous data flow framework</u> (L,F) is a complete lower semi-lattice L with meet operation $\wedge$ and a set of functions $F: L \to L$ satisfying the following axioms:

(16a) (identity) F contains the identity function ι.

(16b) (closure) F is closed under meet, function composition, and $*$, where $(f \wedge g)(x) = f(x) \wedge g(x)$ and $f^*(x) = \wedge \{f^i(x) \mid i \geq 0\}$.

(16c) (continuity) For every $f \in F$ and $X \subseteq L$, $f(\wedge X) = \wedge \{f(x) \mid x \in X\}$.

A <u>continuous data flow problem</u> consists of a flow graph $G = (V,E,s)$, a continuous data flow framework (L,F), and a mapping from E to F; we use f_e to denote the function associated with edge e. The <u>meet over all paths</u> (MOP) <u>solution</u> to this problem is the mapping <u>mop</u> from V to L given by $\text{mop}(v) = \wedge \{f_p(\perp) \mid p \text{ is a path from } s \text{ to } v\}$.

We can use path expressions to solve continuous data flow problems by means of the mapping f defined as follows.

(17a) $f(\Lambda) = \iota$;

$f(e) = f_e$.

(17b) $f(P_1 \cup P_2) = f(P_1) \wedge f(P_2)$;

$f(P_1 \cdot P_2) = f(P_2) \circ f(P_1)$;

$f(P_1^*) = f(P_1)^*$.

Lemma 4. Let $P \neq \emptyset$ be a path expression of type (v,w) . Then for all $x \in L$, $f(P)(x) = \wedge \{f_p(x) \mid p \in \sigma(P)\}$.

Proof. By induction on the number of operation symbols in P . The lemma is immediate if P is atomic. Suppose the lemma is true for path expressions containing fewer than k operation symbols, and let P contain k operation symbols. We have three cases.

Suppose $P = P_1 \cup P_2$. Then

$$f(P)(x) = f(P_1)(x) \wedge f(P_2)(x) = (\wedge \{f_p(x) \mid p \in \sigma(P_1)\}) \wedge (\wedge \{f_p(x) \mid p \in \sigma(P_2)\})$$

$$= \wedge \{f_p(x) \mid x \in \sigma(P_1) \cup \sigma(P_2)\} = \wedge \{f_p(x) \mid p \in \sigma(P)\} .$$

Suppose $P = P_1 \cdot P_2$. Then

$$f(P)(X) = f(P_2)(f(P_1)(x)) = f(P_2)(\wedge \{f_{p_1}(x) \mid p_1 \in \sigma(P_1)\})$$

$$= \wedge \{f(P_2)(f_{p_1}(x)) \mid p_1 \in \sigma(P_1)\} \qquad \text{by continuity}$$

$$= \wedge \{\wedge \{f_{p_1 p_2}(x) \mid p_2 \in \sigma(P_2)\} \mid p_1 \in \sigma(P_1)\}$$

$$= \wedge \{f_{p_1 p_2}(x) \mid p_1 \in \sigma(P_1) \text{ and } p_2 \in \sigma(P_2)\} = \wedge \{f_p(x) \mid p \in \sigma(P)\} .$$

Similarly we can show that if P_1 has fewer than k operation symbols then $f(P_1)^i(x) = \wedge \{f_p(x) \mid p \in \sigma(P_1)^i\}$ for any $i \geq 0$.

Suppose $P = P_1^*$. Then

$$f(P)(x) = f(P_1)^*(x) = {}_A\{f(P_1)^i(x) \mid i \geq 0\}$$

$$= {}_A\{\wedge \{f_p(x) \mid p \in \sigma(P_1)^i\} \mid i \geq 0\} = \wedge \{f_p(x) \mid p \in \sigma(P_i^*)\} \ . \quad \square$$

Theorem 5. For any vertex v , let P(s,v) be a path expression representing all paths from s to v. Then mop(v) = f(P(s,v))(⊥) .

Thus we can use a solution to the single-source path expression problem to solve continuous data flow problems. For examples and extensive discussions of such problems see Cousot and Cousot [5], Fong, Kam, and Ullman [9], Graham and Wegman [13], Kam and Ullman [16,17], Kildall [19], and Rosen [23].

6. Monotone Data Flow Problems.

Many important global flow problems are not continuous [17]. For such problems there is in general no algorithm to compute the meet over all paths solution [17], and we must be satisfied with less information than the MOP solution provides. In such situations the following approach is appropriate.

A monotone data flow framework (L,F) is a complete lower semi-lattice L with meet operation $\wedge$ and a set of functions $F: L \to L$ satisfying the following axioms:

(18a) (identity) F contains the identity function ι .

(18b) (closure) F is closed under meet and function composition.

(18c) (monotonicity) For every $f \in F$ and $x,y \in L$ $x \leq y$ implies $f(x) \leq f(y)$.

(18d) (approximation to f^*) For every function $f \in F$, there is a function $f^{@} \in F$ such that

(i) $f^{@}(x) \leq f^{i}(x)$ for all $x \in L$, $i > 0$; and

(ii) if $x,y \in L$ satisfy $f(x) \wedge y > x$, then $f^{@}(y) > x$.

Monotone frameworks generalize continuous frameworks by requiring only monotonicity (18c) in place of continuity (16c) and by requiring only a pseudo transitive closure function. Note that f^* is the maximum function satisfying (18d).

A monotone data flow problem consists of a flow graph $G = (V,E,s)$, a monotone data flow framework (L,F) , and a mapping from E to F whose values we denote by f_e for $e \in E$. A fixed point for this problem is a mapping $z: V \to L$ such that

(19) $z(s) = \perp$ and $f_e(z(h(e))) \geq z(t(p))$ for any $e \in E$.

A safe solution to the data flow problem is a mapping $x: V \to L$ such that

(20a) $x(v) \leq f_p(\perp)$ for any vertex v and any path p from s to v ; and

(20b) $x(v) \geq z(v)$ for any fixed point z and any vertex v .

Thus a safe solution is a conservative approximation to the MOP solution which is at least as informative as any fixed point. It is easy to prove that any fixed point satisfies (20a); if the data flow problem is continuous, the MOP solution is the maximum fixed point [19].

We can use a slight variant of the mapping defined in Section 4 to compute a safe solution to a monotone data flow problem. Let f be defined as in (17), except $f(P_1^*) = f(P_1)^@$.

Lemma 5. Let $P \neq \emptyset$ be a path expression of type (v,w) . Then $f(P)(x) \leq f_p(x)$ for all $p \in S(P)$ and $x \in L$.

Proof. By induction on the number of operation symbols in P , The lemma is immediate if P is atomic. Suppose the lemma is true for path expressions containing fewer than k operation symbols, and let p contains k operation symbols. We have three cases.

Suppose $P = P_1 \cup P_2$, and $p \in P$. If $p \in P_1$ then

$f(P)(x) = f(P_1)(x) \wedge f(P_2)(x) \leq f(P_1)(x) \leq f_p(x)$ by the induction hypothesis;

similarly if $p \in P_2$,

Suppose $P = P_1 \cdot P_2$ and $p = p_1 p_2$ with $p_1 \in P_1$, $p_2 \in P$. Then

$$f(P)(x) = f(P_2)(f(P_1)(x)) \leq f(P_2)(f_{p_1}(x)) \leq (f_{p_2} \circ f_{p_1})(x) = f_p(x)$$

by monotonicity and the induction hypothesis.

Suppose $P = P_1^*$ and p [illegible] $p_1 p_2$ [illegible] with $p_i \in P_1$ for $1 \leq i \leq k$. Then

$$f(P)(x) = f(P_1)^{@}(x) \leq f(P_1)^k(x) \quad \text{by (8d)(i)}$$

$$\leq f_p(x) \quad \text{by monotonicity and the induction hypothesis, as above.} \square$$

Lemma 6. Let $P \neq \emptyset$ be a path expression of type (v,w). If z is any fixed point, then $f(P)(z(v)) > z(w)$.

Proof. By induction. The lemma is immediate if P is atomic. Suppose the lemma is true for path expressions containing fewer than k operation symbols, and let P contain k operation symbols. We have the usual three cases.

Suppose $P = P_1 \cup P_2$. Then $f(P)(z(v)) = f(P_1)(z(v)) \wedge f(P_2)(z(v)) \geq z(w)$ by the induction hypothesis.

Suppose $P = P_1 \cdot P_2$. Let u be the vertex such that P_1 is of type (v,u) and P_2 is of type (u,w). Then $f(P)(z(v)) = f(P_2)(f(P_1)(z(v))) \geq f(P_2)(z(u)) \geq z(w)$ by the induction hypothesis.

Suppose $P = P_1^*$. By the induction hypothesis, $f(P_1)(z(v)) \wedge z(v) \geq z(v)$. [illegible] (8d)(ii), $f(P)(z(v)) = f(P_1)^{@}(z(v)) \geq z(v)$ [illegible] $\square$

Theorem 6. For each vertex v, let $P(s,v)$ be a path expression representing all paths from s to v. Then the function $x: V \to L$ definedby $x(v) = f(P(s,v))(\perp)$ is a safe solution.

Proof. By Lemma 5, $x(v) = f(P(s,v))(\perp) \leq f_p(\perp)$ for all $p \in S(P(s,v))$; thus x satisfies (20a). Let z be any fixed point. By Lemma 6, $x(v) = f(P(s,v))(\perp) = f(P(s,v))(z(s)) \geq z(v)$; thus x satisfies (20b). □

7. Bounded Data Flow Problems.

Most interesting data flow problems satisfy a stronger condition on L than completeness, called the descending chain condition; every descending chain $x_1 > x_2 > x_3 > \ldots$ in L is finite. For semi-lattices satisfying the descending chain condition, continuity is equivalent to distributivity: $f(x \wedge y) = f(x) \wedge f(y)$ for all $f \in F$ and $x, y \in L$. Our continuous data flow problems are thus a generalization of the distributive data flow problems considered by Kildall [19]. Although most global flow problems satisfy the descending chain condition, some, such as type checking [33], do not.

If the set of functions F in a data flow framework satisfies a boundedness condition, then we can compute an approximation $f^{@}$ to f^{*} for any function $f \in F$ using only function meet and composition. If the framework is continuous as well, it is possible to compute the MOP solution from a set of path expressions representing only some of the paths from the start vertex. We shall consider a hierarchy of boundedness axioms. For $k > 1$, a k-bounded data flow framework (L,F) is a complete lower semi-lattice L with meet operation A and a set of functions $F: L \to L$ satisfying identity (18a), closure (18b), monotonicity (18c), and

(21) (k-boundedness) $f^k(x) \geq \wedge\{f^i(x) \mid 0 \leq i \leq k-1\}$ for all $f \in F$ and $x \in L$.

For $k > 1$, a k-semi-bounded data flow framework (L,F) is a complete lower semi-lattice L with meet operation A and a set of functions $F: L \to L$ satisfying (18a), (18b), (18c), and

(22) (k-semi-boundedness) $f^k(x) \geq (A\{f^i(x) \mid 0 \leq i \leq k-1\}) \; A \; f^k(y)$ for all $f \in F$ and $x, y \in L$.

We define k-bounded and k-semi-bounded data flow problems in the obvious way. It is easy to show that k-boundedness implies k-semi-boundedness and k-semi-boundedness implies $(k+1)$-boundedness. Boundedness, being a property of F and not of L, is neither stronger nor weaker than the descending chain condition. The k-bounded and k-semi-bounded data flow problems include some, but not all, of the global flow problems mentioned in the literature. Problems that use bit vectors, such as finding available expressions [31] and finding live variables [18] are 1-semi-bounded but not 1-bounded, Problems that use "structured partition lattices", such as common subexpression detection [9,16,19], are 2-bounded but not 1-semi-bounded. Type checking [33] is not k-bounded unless some bound is artificially imposed,

Lemma 7. In a k-bounded data flow framework (L,F), $f^* = A\{f^i \mid 0 \leq i \leq k-1\}$ for all $f \in F$.

Proof. We prove by induction on j that if $j > k$, $fj(x) \geq A\{f^i(x) \mid 0 \leq i \leq k-1\}$ for all $f \in F$ and $x \in L$. The claim is true for $j = k$ by k-boundedness. Suppose $j > k$ and the claim is true for $j-1$. Then

$$fj(x) = f^{j-1}(f(x)) \geq A\{f^i(x) \mid 1 \leq i \leq k\}$$ by the induction hypothesis

$$\geq A\{f^i(x) \mid 0 \leq i \leq k-1\}$$ by k-boundedness.

The lemma follows from the claim. □

Lemma 8. In a k-bounded data flow framework (L,F), the function $f^@$ defined by $f^@ = (f \wedge \iota)^{k-1}$ for $f \in F$ satisfies (18d).

Proof. By repeated use of monotonicity, we obtain $f^{@}(x) = (f \wedge \iota)^{k-1}(x) \leq \wedge \{f^{i}(x) \mid 0 \leq i < k-1\}$, which implies (18d)(i) by Lemma 7. We prove by induction on j that if $f(x) \wedge y \geq x$, then $(f \wedge \iota)^{j}(y) \geq x$. The result is immediate for $j = 0$. Suppose $(f \wedge \iota)^{j-1}(y) \geq x$. Then $(f \wedge \iota)^{j}(y) \geq f(x) \wedge x \geq x$. Thus $f(x) \wedge y \geq x$ implies $f^{@}(x) = (f \wedge \iota)^{k-1}(x) \geq x$, and (18d)(ii) holds. □

If (L,F) is a k-bounded data flow framework and $f \in F$, we can compute f^{*} using $O(k)$ function meets and compositions by Lemma 7. We can compute an approximation $f^{@}$ to f^{*} in $O(\log k)$ function meets and compositions by Lemma 8. (We trade accuracy for time if we compute $f^{@}$ instead of f^{*}.) Theorem 6 thus gives a method to solve bounded data flow problems using only function meet, composition, and application.

Suppose (L,F,G,f_e) is a data flow problem which is not only bounded but continuous. In this case $f^{@} = f^{*}$, and we can compute the MOP solution using only function meet, camposition, and application, with $O(\log k)$ such operations replacing each $*$. We can also use path expressions representing only some of the paths from s, as demonstrated by the next results.

Lemma 9. Let (L,F,G,f_e) be a k-bounded continuous data flow problem. Let v be a vertex in G and let p be a path from s to v that is not k-simple. Then there is a set S of paths from s to v such that each path in S is shorter than p and $f_p \geq \wedge \{f_q \mid q \in S\}$.

Proof. If p is not k-simple, then p contains some vertex u at least k+1 times. Let $p = p_0 p_1 p_2 \ldots p_k p_{k+1}$ where each p_i for $1 \leq i \leq k$ is a cycle from u to u. (Both p_0 and p_{k+1} may be the empty path.) Then

$$f_p > f_{p_{k+1}} \circ (\wedge \{f_{p_i} \mid 1 \leq i \leq k\})^k \circ f_{p_0} \quad \text{by continuity}$$

$$> f_{p_{k+1}} \circ \wedge \{(\wedge \{f_{p_i} \mid 1 \leq i \leq k\})^j \mid 0 \leq j \leq k-1\} \circ f_{p_0}$$

by k-boundedness

$$\geq \wedge \{f_q \mid q = p_0 q_1 q_2 \ldots q_\ell p_{k+1} \quad \text{where } 0 \leq \ell \leq k-1$$

and $q_j \in \{p_i \mid 1 \leq i \leq k\}$ for $1 \leq j \leq \ell\}$. □

Corollary 1. Let (L,F,G,f_e) be a k-bounded continuous data flow problem. Let v be a vertex in G and let p be a path from s to v . Then $f_p \geq \wedge \{f_q \mid q \text{ is a k-simple path from s to v}\}$.

Proof. By induction on the length of p using Lemma 9. □

Theorem 7. Let (L,F,G,f_e) be a k-bounded continuous data flow problem. For each vertex v , let $P_k(s,v)$ be a path expression such that $S(P_k(s,v))$ contains at least all the k-simple paths from s to v . Then $\underline{mop}(v) = f(P_k(s,v))(\perp)$, where f is defined as in Section 5.

Proof. Immediate from Lemma 4 and Corollary 1. □

Lemma 10. Let (L,F,G,f_e) be a k-semi-bounded continuous data flow problem. Let v be a vertex in G and let p be a path from s to v which is not k-semi-simple. Then there is a set S of paths from s to v such that each path in S is shorter than p and $f_p > \wedge \{f_q \mid q \in S\}$.

Proof. If p is not k-semi-simple, then p can be partitioned into $p = p_0 p_1 p_2 p_3 \cdots p_{k+2} p_{k+3}$, where p_1 and p_i for $3 \leq i \leq k+2$ are cycles, and p_0, p_2, p_{k+3} are possibly empty. Then

$$f_p \geq f_{p_{k+3}} \circ (\wedge \{f_{p_i} \mid 3 \leq i \leq k+2\})^k \circ f_{p_0 p_1 p_2} \quad \text{by continuity}$$

$$\geq f_{p_{k+3}} \circ \wedge \{(\wedge \{f_{p_i} \mid 3 \leq i \leq k+2\})^j \mid 0 \leq j \leq k-1\} \circ f_{p_0 p_1 p_2}$$

$$\wedge f_{p_{k+3}} \circ (\wedge \{f_{p_i} \mid 3 \leq i \leq k+2\})^k \circ f_{p_0 p_2} \quad \text{by k-semi-boundedness and continuity}$$

$$\geq (\wedge \{f_q \mid q = p_0 p_1 p_2 q_1 q_2 \cdots q_\ell p_{k+3} \text{ where } 0 \leq \ell \leq k-1$$

$$\text{and } q_j \in \{p_i \mid 3 \leq i \leq k+2\} \text{ for } 1 \leq j \leq \ell\})$$

$$\wedge (\wedge \{f_q \mid q = p_0 p_2 q_1 q_2 \cdots q_k p_{k+3} \text{ where } q_j \in \{p_i \mid 3 \leq i \leq k+2\}$$

$$\text{for } 1 \leq j \leq k\}) \ . \ \square$$

Corollary 2. Let (L, F, G, f_e) be a k-semi-bounded continuous data flow problem. Let v be a vertex in G and let p be a path from s to v. Then $f_p \geq \wedge \{f_q \mid q \text{ is a k-semi-simple path from s to v}\}$.

Proof. By induction on the length of p using Lemma 10. $\square$

Theorem 8. Let (L, F, G, f_e) be a k-semi-bounded continuous data flow problem. For each vertex v, let $P'_k(s,v)$ be a path expression such that $S(P'_k(s,v))$ contains at least all the k-semi-simple paths from s to v. Then $\underline{mop}(v) = f(P'_k(s,v))(\perp)$, where f is defined as in Section 5.

Proof. Immediate from Lemma 4 and Corollary 2. □

Corollaries 1 and 2 require continuity; in fact, the MOP solution is not effectively computable in a general 2-bounded monotone data flow problem [17]. See Kam and Ullman [16] and Tarjan [29] for further discussion of the effect of boundedness on global flow analysis.

8. An Idiosyncratic Data Flow Problem.

As a final application of our technique, we shall consider a data flow problem that does not fit naturally into the semi-lattice framework, but that can still be solved easily using a mapping from path expressions. The problem arises in the optimization of very-high-level languages and has been studied by Fong [8].

Let $G = (V, E, s)$ be the flow graph of a program which contains occurrences of an expression $\mathcal{E}$. With each edge e of the program is associated an effect, which has one of four values depending upon what flow of control through edge e does to the value of $\mathcal{E}$.

$$\underline{\text{effect}}(e) = \left\{ \begin{array}{l} \underline{\text{gen}} \\ \underline{\text{kill}} \\ \underline{\text{injure}} \\ \underline{\text{trans}} \end{array} \right\} \text{ if } \left\{ \begin{array}{l} \text{*the program recomputes } \mathcal{E} \\ \text{the program makes a large change in the value of } \mathcal{E} \\ \text{the program makes a small change in the value of } \mathcal{E} \\ \text{the program does not affect the current value of } \mathcal{E} \end{array} \right\}$$

For any vertex v, we say $\mathcal{E}$ is implicitly available on entry to v if there is a positive bound b such that, for every path $p = e_1, e_2, \ldots, e_k$ from s to v, there is an i such that (i) $\underline{\text{effect}}(e_i) = \underline{\text{gen}}$, (ii $\underline{\text{effect}}(e_j) \neq \underline{\text{kill}}$ for $1 < j < k$, and (iii) the number of values j such that $i < j < k$ and $\underline{\text{effect}}(e_j) = \underline{\text{injure}}$ is bounded by b. Note that the bound b can depend upon the vertex v but not upon the path p,

The problem we wish to solve is to determine from $(\underline{\text{effect}}(e) \mid e \in \mathcal{E})$ the vertices at which $\mathcal{E}$ is implicitly available. The idea is that if the most-recently-computed value of $\mathcal{E}$ can be injured only a bounded number of times before entering v, we can compute the value on entry

to v from the most-recently-computed value by performing a bounded number of updates. Otherwise, we must completely recompute ε to obtain its value on entry to v .

Fong [8] claims that this problem cannot be formulated within the semi-lattice framework, "at least in the only natural choice of semi-lattice." However, Fong observes that the problem can still be solved efficiently. We shall define a mapping from path expressions for this purpose.

Let $D = \{g, t_0, t_+, \omega\}$ be a set having operations $\wedge, \circ, @$ defined by the following tables.

$\wedge$	g	t_0	t_+	ω
g	g	t_0	t_+	ω
t_0	t_0	t_0	t_+	ω
t_+	t_+	t_+	t_+	ω
ω	ω	ω	ω	ω

$\circ$	g	t_0	t_+	ω
g	g	g	g	ω
t_0	g	t_0	t_+	ω
t_+	g	t_+	t_+	ω
ω	g	ω	ω	ω

	@
g	t_0
t_0	t_0
t_+	ω
ω	ω

Let the mapping f from path expressions to D be defined as follows.

(21a) $\quad f(\Lambda) = t_0$;

$$f(e) = \left\{ \begin{array}{l} g \\ \omega \\ t_+ \\ t_0 \end{array} \right\} \text{ if } \underline{\text{effect}}(e) = \left\{ \begin{array}{l} \underline{\text{gen}} \\ \underline{\text{kill}} \\ \underline{\text{injure}} \\ \underline{\text{trans}} \end{array} \right\} \text{ for } e \in E \text{ .}$$

(21b) $f(P_1 \cup P_2) = f(P_1) \wedge f(P_2)$;

$f(P_1 \circ P_2) = f(P_1) \circ f(P_2)$;

$f(P_1^*) = f(P_1)^{@}$.

We call a path $p = e_1, e_2, \ldots, e_k$ in G a t_i -path if $\underline{\text{effect}}(e_j) \in \{\underline{\text{injure}}, \underline{\text{trans}}\}$ for $1 \leq j \leq k$ and the number of edges e_j such that $\underline{\text{effect}}(e_j) = \underline{\text{injure}}$ is i . We call a path p a g_i -path if it can be partitioned into $p = p_1, e, p_2$, where $\underline{\text{effect}}(e) = \underline{\text{gen}}$ and p_2 is a t_i -path. We call a path p an ω-path if it can be partitioned into $p = p_1, e, p_2$, where $\underline{\text{effect}}(e) = \underline{\text{kill}}$ and p is a t_i -path for some i .

Lemma 11. Let P be a path expression. Then

(i) $f(P) = g$ if there is a bound b such that every path in $\sigma(P)$ is a g_i -path with $i \leq b$;

(ii) $f(P) = t_0$ if there is a bound b such that every path in $a(P)$ is either a g_i -path with $i < b$ or a t_0 -path, and $\sigma(P)$ contains at least one t_0 path.

(iii) $f(P) = t_0$ if there is a bound b such that every path in $\sigma(P)$ is either a g_i -path with $i < b$ or a t_i -path with $i \leq b$, and $a(P)$ contains at least one t_i -path with $i > 0$.

(iv) $f(P) = \omega$ in all other cases. (For any bound b , $\sigma(P)$ contains either a g_i -path with $i > b$, a t_i -path with $i > 0$, or an ω-path.)

Proof. Straightforward but tedious, by induction on the number of operation symbols in P . □

Theorem 9. For each vertex v in G , let P(s,v) be a path expression representing all paths from s to v in G . Then ξ is implicitly available at v if and only if f(P(s,v)) = g .

Proof. Immediate from Lemma 11.

Actual occurrences of the implicit availability problem usually involve a number of expressions. We can perform the computation associated with Theorem 9 in parallel for all the expressions by using bit vector operations. Since D contains four elements, we need two bit vectors for each value computed (rather than the three proposed by Fong [8]). By adding an additional element to D we can compute the explicitly available expressions (those available with no injuries) in addition to the implicitly available ones.

9. Remarks.

We have shown how to use path expressions to solve three kinds of path problems on directed graphs. Our results allow us to build a general algorithm for solving path problems on directed graphs; to solve a particular path problem, we merely interpret $\cup$, ., and * appropriately. We can base such an algorithm on Gaussian or Gauss -Jordan elimination [21]. Tarjan[30] discusses another algorithm, which is especially efficient on reducible and almost-reducible graphs [15,28].

Our results serve to formally justify the empirical observation that the same algorithms work on many different path problems, There are of course algorithms that solve only a particular kind of path problem, such as Dijkstra's[6] and Fredman's [12] shortest path algorithms and Pan's improvement to Strassen's algorithm for solving linear equations [4,22,26]. However, any algorithm able to compute path expressions also solves all the path problems we have considered here.

Our ideas extend easily to matrix multiplication problems and to problems requiring the transitive closure of a matrix. See Aho, Hopcroft, and Ullman [1] and Lehman [21] for discussions of such problems.

Appendix: Graph-Theoretic Definitions

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 $t(e)$. 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 $h(e_1)$ to $t(e_k)$. 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.

If there is a path from a vertex v to a vertex w, then w is reachable from v. A flow graph $G = (V,E,s)$ is a graph containing a distinguished start vertex s such that every vertex is reachable from s.

A simple path p is a path containing no vertex twice. For $k \geq 1$, a k-simple path is a path containing no vertex $k+1$ times. Thus a 1-simple path is simple. A k-semi-simple path is a path p that can be partitioned as $p = p_1, e, p_2$, where p_1 is simple, e is an edge, and p_2 is k-simple.

Acknowledgement.

My thanks to Fran Berman for extensive and illuminating discussions of these ideas.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974, 195-206.

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

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

[4] J. R. Bunch and J. E. Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," Math. Comp. 28 (1974), 231-236.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," Conf. Record of' the Fourth ACM Symp. on Principles of Prog. Lang. (1977), 238-252.

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," Num. Math. 1 (1959), 269-271.

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

[8] A. C. Fong, "Generalized common subexpressions in very high level languages," Conf. Record of the Fourth ACM Symp. on Principles of Prog. Lang. (1977), 48-57.

[9] A. C. Fong, J. B. Kam, and J. D. Ullman, "Applications of lattice algebra to loop optimization," Conf. Record of the Second ACM Symp. on Principles of Prog. Lang. (1975), 1-9.

[10] L. R. Ford, Jr. and D. R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, N.J., 1962, 130-134.

[11] G. E. Forsythe and C. B. Moler, Computer Solution of Linear Algebraic Equations, Prentice-Hall, Englewood Cliffs, N.J., 1967, 27-36.

[12] M. L. Fredman, "New bounds on the complexity of the shortest path problem," SIAM J. Comput. 5 (1976), 83-89.

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

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

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

[16] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," Journal ACM 23 (1976), 158-171.

[17] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," Acta Info. 7 (1977), 305-317.

[18] 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.

[19] G. A. Kildall, "A unified approach to program optimization," Conf. Record of the ACM Symp. on Principles of Prog, Lang. (1973), 10-21.

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

[21] D. J. Lehman "Algebraic structures for transitive closure," Theoretical Comp. Sci. 4 (1977), 59-76.

[22] V. Y. Pan, "Strassen's algorithm is not optimal: trilinear technique of aggregating uniting and canceling for constructing fast algorithms for matrix operations," Proc. 19th Annual Symp. on Foundations of Computer Science (1978), 166-176.

[23] B. K. Rosen, "Monoids for rapid data flow analysis," Research Report RC 7032 IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (1978).

[24] A. Salomaa, "Two complete axiom systems for the algebra of regular events," Journal ACM (1966), 158-169.

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

[26] V. Strassen, "Gaussian elimination is not optimal," Num. Math. 13 (1969), 354-356.

[27] G. Szász, Introduction to Lattice Theory, B. Balkay and G. Tóth, trans., Academic Press, New York, 1963, 38-42, 59-60.

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

[29] R. E. Tarjan, "Iterative algorithms for global flow analysis," Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, 91-102.

[30] R. E. Tarjan, "Fast algorithms for solving path problems," Journal ACM, submitted.

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

[32] S. Warshall, "A theorem on Boolean matrices," Journal ACM 9 (1962), 11-12.

[33] B. Wegbreit, "Property extraction in well-founded property sets," IEEE Trans. on Software Engineering 1 (1975), 270-285.