Solving Regular Path Queries[⋆]

Yanhong A. Liu and Fuxiang Yu

Computer Science Dept., State University of New York, Stony Brook, NY 11794
{liu,fuxiang}@cs.sunysb.edu

Abstract. Regular path queries are a way of declaratively specifying program analyses as a kind of regular expressions that are matched against paths in graph representations of programs. These and similar queries are useful for other path analysis problems as well. This paper describes the precise specification, derivation, and analysis of a complete algorithm and data structures for solving regular path queries. We first show two ways of specifying the problem and deriving a high-level algorithmic solution, using predicate logic and language inclusion, respectively. Both lead to a set-based fixed-point specification. We then derive a complete implementation from this specification using Paige's methods that consist of dominated convergence, finite differencing, and real-time simulation. This formal derivation allows us to analyze the time and space complexity of the implementation precisely in terms of size parameters of the graph and the deterministic finite automaton that corresponds to the regular expression. In particular, the time and space complexity is linear in the size of the graph. We also note that the problem is PSPACE-complete in terms of the size of the regular expression. In applications such as program analysis, the size of the graph may be very large, but the size of the regular expression is small and can be considered a constant.

1 Introduction

Regular path queries are a way of declaratively specifying program analyses as a kind of regular expressions that are matched against paths in graph representations of programs [5]. Related queries are also used in model checking [9]. Program analysis and model checking are important for many applications. For example, program analysis is critical for program optimization, and model checking is important for formal verification. In fact, regular expressions provide a general framework for capturing many path problems [18, 17]. Program analysis and model checking are just two of many applications.

This paper describes the precise specification, derivation, and analysis of a complete algorithm and data structures for solving regular path queries. The specification and derivation consist of two parts. First, specify the problem and derive a high-level algorithm that can be expressed using a set-based language

[⋆] This work was supported in part by ONR under grants N00014-01-1-0109 and N00014-99-1-0132.

with fixed-point operations. Then, start with a set-based fixed-point specification and derive a complete implementation with precise data structures and operations on them. This formal derivation allows us to analyze the time and space complexity of the implementation precisely in terms of size parameters of the graph and the deterministic finite automaton that corresponds to the regular expression. In particular, the time and space complexity is linear in the size of the graph. We also note that the problem is PSPACE-complete in terms of the size of the regular expression. In applications such as program analysis and model checking, the size of the graph may be very large, but the size of the regular expression is small and can be considered a constant [5].

The derivation from a set-based fixed-point specification to a complete implementation uses Paige's methods that are centered around finite differencing [10, 14, 11], i.e., computing expensive set expressions incrementally. We first use dominated convergence [3] at the higher level to transform fixed-point operations into loops. We then apply finite differencing [14, 11] to transform expensive set expressions in loops into incremental operations. Finally, we use real-time simulation [12, 2] at the lower level to implement sets and set operations using efficient data structures. The derivation is completely systematic, and the resulting algorithm and data structures can be analyzed precisely and map to physical implementations directly.

In contrast, starting at some initial specification of the problem and arriving at a set-based fixed-point specification have not been as systematic. A fixed-point specification often corresponds to some high-level algorithm already. How should one obtain it? In a most recent work by de Moor et al. [5], specification and derivation that arrive at such a high-level algorithm are given, using calculus of relations and universal algebra. This paper shows two other ways of specifying the problem and deriving a high-level algorithmic solution, using predicate logic and language inclusion, respectively. Both derivations are extremely succinct, and both results lead easily to a set-based fixed-point specification. The initial specification using predicate logic corresponds rather directly to the given English description of the problem.

The rest of the paper is organized as follows. Section 2 describes the problem specification and two ways of arriving at a high-level algorithmic solution. Section 3 expresses the high-level solution using a set-based language with fixed-point operations, and introduces Paige's approach for computing fixed points iteratively. Sections 4 and 5 derive the precise incremental computation steps and data structures, respectively. Section 6 discusses related issues. Section 7 compares with related work and concludes.

2 Problem specification and high-level algorithmic solution

We describe how the problem can be specified and how a high-level algorithm for solving it can be derived succinctly in two ways, using predicate logic and language inclusion, respectively.

The regular path query problem. Consider an edge-labeled directed graph G and a regular-expression pattern P. We say that a path in G matches P if the sequence of edge labels on the path is in the regular language generated by P. The regular path query problem is:

> Given an edge-labeled directed graph G with a special vertex $v0$, and a regular expression P, compute all vertices v in G such that all paths from $v0$ to v match P.

Precisely, we regard the given graph as a set G of labeled edges of the form $\langle v1, a, v2\rangle$, with source and target vertices $v1$ and $v2$ respectively and edge label a, where $v0$ is a special vertex in the graph. We consider a deterministic finite automaton (DFA) corresponding to the given regular expression, where P is a set of labeled transitions of the form $\langle s1, a, s2\rangle$, with source and target states (vertices) $s1$ and $s2$ respectively and transition (edge) label a, and where $s0$ is the start state, and F is the set of final states. We assume that for each state there is an outgoing edge for each label; a trap state can be added to achieve this if needed. The initial specifications (1) and (4) below are correct even if P is nondeterministic. We discuss why we use a DFA instead of the regular expression or its corresponding nondeterministic finite automaton (NFA) in Section 6.

Using predicate logic. The problem as described above can be written directly as: Given G, $v0$, P, $s0$, and F, compute the set

$$\{v \in \mathit{vertices}(G) \mid \forall p\ (p \in \mathit{path}(v0, v, G) \Rightarrow \exists s\ (p \in \mathit{path}(s0, s, P) \land s \in F))\} \quad (1)$$

where $\mathit{vertices}(G)$ is the set of all vertices in G, $\mathit{path}(v0, v, G)$ is the set of all sequences of edge labels on paths from $v0$ to v in G, and $\mathit{path}(s0, s, P)$ is similar.

Since P is a DFA, i.e., it is deterministic, the right side of $\Rightarrow$ in (1) equals

$$\neg\exists s\ (p \in \mathit{path}(s0, s, P) \land s \notin F)$$

which equals

$$\forall s\ (\neg p \in \mathit{path}(s0, s, P) \lor s \in F).$$

Now, move $\forall s$ above out of $\Rightarrow$, and move negated left operand of $\lor$ to left of $\Rightarrow$. We have that (1) equals

$$\{v \in \mathit{vertices}(G) \mid \forall p\ \forall s\ (p \in \mathit{path}(v0, v, G) \land p \in \mathit{path}(s0, s, P) \Rightarrow s \in F)\}$$

which, letting $G \times P = \{\langle\langle v1, s1\rangle, a, \langle v2, s2\rangle\rangle \mid \langle v1, a, v2\rangle \in G \land \langle s1, a, s2\rangle \in P\}$ be the product of P and Q, equals

$$\{v \in \mathit{vertices}(G) \mid \forall p\ \forall s\ (p \in \mathit{path}(\langle v0, s0\rangle, \langle v, s\rangle, G \times P) \Rightarrow s \in F)\} \quad (2)$$

We can see that (2) computes the set of vertices v in G such that, if there is a path from $\langle v0, s0\rangle$ to $\langle v, s\rangle$ in $G \times P$, i.e., if $\langle v, s\rangle$ is reachable from $\langle v0, s0\rangle$, then s is in F. Precisely, moving $\forall p$ inside yields $\exists p\ (p \in \mathit{path}(\langle v0, s0\rangle, \langle v, s\rangle, G \times P))$ on the left side of $\Rightarrow$. If we let $\mathit{reach}(G \times P, \langle v0, s0\rangle)$ be the set of nodes reachable from $\langle v0, s0\rangle$ in $G \times P$, then $\exists p\ (p \in \mathit{path}(\langle v0, s0\rangle, \langle v, s\rangle, G \times P))$ equals $\langle v, s\rangle \in \mathit{reach}(G \times P, \langle v0, s0\rangle)$, and (2) equals

$$\{v \in \mathit{vertices}(G) \mid \forall s\ (\langle v, s\rangle \in \mathit{reach}(G \times P, \langle v0, s0\rangle) \Rightarrow s \in F)\} \quad (3)$$

Either (2) or (3) expresses the same high-level algorithm derived earlier [5].

Using language inclusion. We can regard the given graph G as a labeled transition system, or an NFA, with start state $v0$ and the set of final states not yet defined. Then the problem is to compute a subset U of the vertices in G, such that if $v0$ is the start state and U is the set of final states, then the language $L_{(G,v0,U)}$ generated by the NFA $(G, v0, U)$ is a subset of the language $L_{(P,s0,F)}$ generated by the DFA $(P, s0, F)$, and U is the largest such set. That is, given G, $v0$, P, $s0$, and F, the problem is to compute

$$\max\{U \subseteq \mathit{vertices}(G) \mid L_{(G,v0,U)} \subseteq L_{(P,s0,F)}\} \tag{4}$$

where max follows the partial order of set inclusion $\subseteq$.

Since P is a DFA, if S^c denotes the complement of set S, we have

$$L_{(P,s0,F)} = (L_{(P,s0,F^c)})^c$$

Thus, the language inclusion in (4), with the new right side $(L_{(P,s0,F^c)})^c$, equals

$$L_{(G,v0,U)} \cap L_{(P,s0,F^c)} = \emptyset$$

which equals

$$L_{(G\times P,\langle v0,s0\rangle,U\times F^c)} = \emptyset \tag{5}$$

where the second $\times$ is a simple Cartesian product.

Using the *reach* notation in (3), (5) equals

$$\mathit{reach}(G \times P, \langle v0, s0\rangle) \cap (U \times F^c) = \emptyset$$

which equals

$$\forall\langle v, s\rangle\ (\langle v, s\rangle \in \mathit{reach}(G \times P, \langle v0, s0\rangle) \wedge s \in F^c \Rightarrow v \notin U)$$

Thus, (4) equals

$$\max\{U \subseteq \mathit{vertices}(G) \mid \forall\langle v, s\rangle\ (\langle v, s\rangle \in \mathit{reach}(G\times P, \langle v0, s0\rangle)\wedge s \in F^c \Rightarrow v \notin U)\}$$

which equals

$$\mathit{vertices}(G) - \{v \mid \exists s\ (\langle v, s\rangle \in \mathit{reach}(G \times P, \langle v0, s0\rangle) \wedge s \in F^c)\} \tag{6}$$

Note that (6) equals (3) except for the double negation based on the rule

$$\{x \in S \mid \mathit{predicate}(x)\} = S - \{x \in S \mid \neg\ \mathit{predicate}(x)\}.$$

Starting with either (6) or (3) will lead to not only the same complete algorithm but also the same derivation except for the initial appearance of set difference. We will use (6) for no particular reason.

3 Approach for deriving a complete implementation

Notation. We use a set-based language for deriving a complete implementation. The language is based on SETL [15, 16] extended with a fixed-point operation [3]. Primitive data types are sets, pairs, and maps, i.e., binary relations represented as sets of pairs. Their syntax and operations on them are summarized below:

$\{X_1, ..., X_n\}$	a set with elements X_1,...,X_n
$[X_1, X_2]$	a pair with elements X_1 and X_2
$\{[X_1, Y_1], ..., [X_n, Y_n]\}$	a map that maps X_1 to Y_1, ..., X_n to Y_n
$\{\}$	empty set
$S + T,\ S - T$	union and difference, respectively, of sets S and T
S **with** X, S **less** X	$S + \{X\}$ and $S - \{X\}$, respectively
$S \subseteq T$	whether S is a subset of T
$X \in S,\ X \notin S$	whether or not, respectively, X is an element of S
dom(M)	domain of map M, i.e., $\{X : [X, Y] \in M\}$
ran(M)	range of map M, i.e., $\{Y : [X, Y] \in M\}$
$M\{Z\}$	image set of Z under map M, i.e., $\{Y : [X, Y] \in M \mid X = Z\}$
$M[S]$	image set union of S under M, i.e., $\{Y : [X, Y] \in M \mid X \in S\}$

We use the notation below for set comprehension. Y_i's enumerate elements of all S_i's; for each combination of $Y_1, ..., Y_n$, if the Boolean value of expression Z is true, then the value of expression X forms an element of the resulting set.

$$\{X : Y_1 \in S_1, ..., Y_n \in S_n \mid Z\} \quad \text{set former}$$

$\mathbf{LFP}_{\subseteq, S_0}(F(S), S)$ denotes the minimum element S, with respect to partial ordering $\subseteq$, that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$. We use standard control constructs **while**, **for**, and **if**, and we use indentation to indicate scoping. We abbreviate $X := X$ **op** Y as X **op** $:= Y$.

A set-based fixed-point specification. We represent a set of labeled edges of the form $\langle x1, a, x2\rangle$ using a set of pairs of the form $[x1, [a, x2]]$, which can be built straightforwardly in the same loop that reads in the 3-tuple form. Thus, if $[x1, [a, x2]]$ is a labeled edge in G, then $x1$ is in **dom**(G), a is in **dom**(**ran**(G)), and $x2$ is in **ran**(**ran**(G)). So $vertices(G)$ = **dom**(G) + **ran**(**ran**(G)), and (6) can be expressed directly using set and fixed-point notation as

$$\mathbf{dom}(G) + \mathbf{ran}(\mathbf{ran}(G)) - \{v : [v, s] \in \mathbf{LFP}_{\subseteq, \{[v0, s0]\}}((G \times P)[R], R) \mid s \notin F\} \quad (7)$$

where $G \times P = \{[[v1, s1], [a, [v2, s2]]] : [v1, [a, v2]] \in G \wedge [s1, [b, s2]] \in P \mid a = b\}$.

Approach. The method has three steps: (1) dominated convergence, (2) finite differencing, and (3) real-time simulation.

Dominated convergence [3] transforms a set-based fixed-point specification into a **while**-loop. The idea is to perform a small update operation in each iteration. The fixed-point expression $\mathbf{LFP}_{\subseteq, \{[v0, s0]\}}((G \times P)[R], R)$ in (7) is transformed into the following **while**-loop, making use of $\lambda R.F(R) \cup R$ being monotone and inflationary at $[v0, s0]$:

```
$R := \{[v0, s0]\};$
$\mathbf{while\ exists}\ [v, s] \in (G \times P)[R] - R$                    (8)
    $R\ \mathbf{with} := [v, s];$
```

This code is followed by

$$O := \mathbf{dom}(G) + \mathbf{ran}(\mathbf{ran}(G)) - \{v : [v, s] \in R \mid s \notin F\};$$

When the loop in (8) terminates, R is the set of nodes in $G \times P$ reachable from $[v0, s0]$. O is the output set.

To simplify the initialization code, we can move initialization of R into the loop body, yielding:

```
$R := \{\};$
$\mathbf{while\ exists}\ [v, s] \in \{[v0, s0]\} + (G \times P)[R] - R$
    $R\ \mathbf{with} := [v, s];$                                        (9)
$O := \mathbf{dom}(G) + \mathbf{ran}(\mathbf{ran}(G)) - \{v : [v, s] \in R \mid s \notin F\};$
```

Finite differencing [14, 11] transforms expensive set operations in a loop into incremental operations. The idea is to replace expensive expressions exp_1, ..., exp_n in a loop $LOOP$ with fresh variables E_1, ..., E_n, respectively, and maintain the invariants $E_1 = exp_1$, ..., $E_n = exp_n$ by inserting appropriate initializations or updates to E_1, ..., E_n at each assignment in $LOOP$. We denote the transformed loop as

$$\Delta\ E_1, ..., E_n \langle LOOP \rangle$$

For our program (9), expensive expressions, i.e., non-constant-time expressions here, are the one that computes O and others that are needed for computing $\{[v0, s0]\} + (G \times P)[R] - R$. We use fresh variables to hold their values. These variables are initialized together with the assignment $R := \{\}$ and are updated incrementally as R is augmented by $[v, s]$ in each iteration. Liu [7] gives references to much work that exploits similar ideas.

Real-time simulation [12, 2] selects appropriate data structures for representing sets so that operations on them can be implemented efficiently. The idea is to design sophisticated linked structures based on how sets and set elements are accessed, so that each operation can be performed in worst-case constant time and with at most a constant (a small fraction) factor of overall space overhead.

4 Finite differencing

This section transforms (9) to compute expensive set operations incrementally.

Identifying expensive subexpressions. The expensive subcomputations in (9) are named as follows,

```
$I\ = G \times P$                                              // product graph
$S\ = I[R]$                                                    // successors in product graph
$W = \{[v0, s0]\} + S - R$                                     // workset                        (10)
$O\ = \mathbf{dom}(G) + \mathbf{ran}(\mathbf{ran}(G)) - \{v : [v, s] \in R \mid s \notin F\}$  // output set
```

We want to compute these computations incrementally using finite differencing. That is, we want to update the sets I, S, W, and O whenever we update R. Thus, the overall computation in (9) becomes

```
$\Delta\ I, S, W, O \ \langle R := \{\};$
                  $\textbf{while exists } [v, s] \in W$                (11)
                     $R \textbf{ with} := [v, s]; \rangle$
```

Initialization and incremental maintenance. First, sets I, S, W, and O need to be initialized together with $R := \{\}$ before the loop in (11). That is, they need to be set to values that correspond to $R = \{\}$ based on their definitions in (10). This yields

```
$I\ := G \times P;$
$S\ := \{\};$
$W := \{[v0, s0]\};$
$O\ := \textbf{dom}(G) + \textbf{ran}(\textbf{ran}(G));$
```

Then, sets I, S, W, and O also need to be maintained incrementally together with R **with** $:= [v, s]$ in the loop body in (11). That is, we need to update their values corresponding to the update R **with** $:= [v, s]$ based on their definitions in (10). Clearly, set I remain unchanged, and the other sets can be updated as follows, where the two updates to W are with respects to the update to S and R, respectively.

```
$S\ + := \{[v2, s2] : [v, [a, v2]] \in G, [s, [b, s2]] \in P \mid a = b \wedge [v2, s2] \notin S\};$
$W + := \{[v2, s2] : [v, [a, v2]] \in G, [s, [b, s2]] \in P \mid a = b \wedge [v2, s2] \notin W \wedge [v2, s2] \notin R\};$
$W \textbf{ less} := [v, s];$
$\textbf{if } s \notin F \wedge v \in O \textbf{ then}$
    $O \textbf{ less} := v;$
```

Adding these initialization and incremental updates to the initialization and the body of the loop in (11), we obtain the following complete program:

```
$I\ := G \times P;$
$S\ := \{\};$
$W := \{[v0, s0]\};$
$O\ := \textbf{dom}(G) + \textbf{ran}(\textbf{ran}(G));$
$R\ := \{\};$
$\textbf{while exists } [v, s] \in W$
    $S\ + := \{[v2, s2] : [v, [a, v2]] \in G, [s, [b, s2]] \in P \mid a = b \wedge [v2, s2] \notin S\};$
    $W + := \{[v2, s2] : [v, [a, v2]] \in G, [s, [b, s2]] \in P \mid a = b \wedge [v2, s2] \notin W \wedge [v2, s2] \notin R\};$
    $W \textbf{ less} := [v, s];$
    $\textbf{if } s \notin F \wedge v \in O \textbf{ then}$
        $O \textbf{ less} := v;$
    $R \textbf{ with} := [v, s];$
                                                                      (12)
```

Eliminating dead code. It is easy to see that sets I and S can be eliminated, and (12) becomes

```
$W := \{[v0, s0]\}$;                                                    // initialize $W$
$O\ := \mathbf{dom}(G) + \mathbf{ran}(\mathbf{ran}(G))$;                 // initialize $O$
$R\ := \{\}$;                                                           // initialize $R$
while exists $[v, s] \in W$
    $W+ := \{[v2, s2] : [v, [a, v2]] \in G, [s, [b, s2]] \in P \mid a = b \land [v2, s2] \notin W \land [v2, s2] \notin R\}$;
                                                                        // add to $W$
    $W$ less $:= [v, s]$;                                               // delete from $W$
    if $s \notin F \land v \in O$ then
        $O$ less $:= v$;                                                // update $O$
    $R$ with $:= [v, s]$;                                               // update $R$
```
(13)

Finally, transform all aggregate set operations into explicit loops that process set elements one at a time. We obtain the following complete algorithm after finite differencing; for completeness, we also show the input and output explicitly.

```
input$(G, P, v0, s0, F)$;
$W := \{[v0, s0]\}$;                          // initialize $W$
$O\ := \{\}$;                                 // initialize $O$
for $v1 \in \mathbf{dom}(G)$                  //     using for-loops
    if $v1 \notin O$ then
        $O$ with $:= v1$;
    for $a \in \mathbf{dom}(G\{v1\})$
        for $v2 \in (G\{v1\})\{a\}$
            if $v2 \notin O$ then
                $O$ with $:= v2$;
$R\ := \{\}$;                                 // initialize $R$

while exists $v \in \mathbf{dom}(W)$          // iterate through $W$
    while exists $s \in W\{v\}$               //     using while-loops
        for $a \in \mathbf{dom}(G\{v\})$      // iterate through $(G \times P)\{[v, s]\}$
            for $v2 \in (G\{v\})\{a\}$        //     using for-loops
                for $s2 \in (P\{s\})\{a\}$    //     to add to $W$
                    if $s2 \notin W\{v2\} \land s2 \notin R\{v2\}$ then
                        $W\{v2\}$ with $:= s2$;    // add to $W$
        $W\{v\}$ less $:= s$;                 // delete from $W$
        if $s \notin F \land v \in O$ then
            $O$ less $:= v$;                  // update $O$
        $R\{v\}$ with $:= s$;                 // update $R$
output$(O)$;
```
(14)

Analysis of time complexity. Assume each primitive operation in the above algorithm takes $\mathcal{O}(1)$ time, which will be achieved using data structures in the next section. The algorithm considers each edge in $G \times P$ at most twice, once for adding to and once for deleting from W. Therefore, the time complexity of the above algorithm is $\mathcal{O}(|G| * |P|)$.

5 Data structure selection

We describe how to guarantee that each set operation in algorithm (14) takes worst-case $\mathcal{O}(1)$ time.

All set operations in (14) are of the following primitive forms: set initialization $S := \{\}$, computing domain set $\mathbf{dom}(M)$, computing image set $M\{x\}$, element retrieval **for** $X \in S$ and **while exists** $X \in S$, membership test $X \in S$ and $X \notin S$, and element addition S **with** X and deletion S **less** X.

We use *associative access* to refer to membership test ($X \in S$ and $X \notin S$) and computing image set ($M\{X\}$). Such an operation requires one to be able to locate an element (X) in a set (S or $\mathbf{dom}(M)$).

Based representations. Consider using a singly linked list for a set and for each of the domain and image sets of a map, and letting each element in a domain linked list contain a pointer to its image linked list. That is, represent a set as a linked list, and represent a map as a linked list of linked lists. It is easy to see that, if associative access can be done in worst-case $\mathcal{O}(1)$ time, so can all other primitive operations. To see this, note that computing a domain set or an image set needs to return only a pointer to the set; retrieving an element from a set needs to locate only any element in the set; and adding or deleting an element from a set can be done in constant time after doing an associate access. An associative access would take linear time if a linked list is naively traversed to locate an element. A classical approach to solve this problem is to use hash tables [1] instead of linked lists. However, this gives average, rather than worst-case, $\mathcal{O}(1)$ time for each operation, and has an overhead of computing hashing related functions for each operation.

Paige et al. [12, 2] describe a technique for designing linked structures that support associative access in worst-case $\mathcal{O}(1)$ time with little space overhead for a general class of set-based programs. Consider

```
for $X \in W$ or while exists $X \in W$
   ...$X \in S$... or ...$X \notin S$... or ...$M\{X\}$... where the domain of $M$ is $S$
```

We want to locate value X in S after it has been located in W. The idea is to use a finite universal set B, called a base, to store values for both W and S, so that retrieval from W also locates the value in S. B is represented as a set (this set is only conceptual) of records, with a K field storing the key (i.e., value). Set S is represented using a S field of B: records of B whose keys belong to S are connected by a linked list where the links are stored in the S field; records of B whose keys are not in S store a special value for undefined in the S field. Set W is represented as a separate linked list of pointers to records of B whose keys belong to W. Thus, an element of S is represented as *a field in* the record, and S is said to be *strongly based* on B; and element of W is represented as *a pointer to* the record, and W is said to be *weakly based* on B. This representation allows an arbitrary number of weakly based sets but only a constant number of strongly based sets. Essentially, base B provides a kind of indexing.

It is easy to see the S field can be a single bit if S is not traversed; otherwise, if there is no deletion from S, then elements in S only need to be linked using a singly linked list. Only when S is traversed and there is any deletion operation from S do we need a doubly linked list.

Data structures. Consider the **while**-loops in (14). The outer **while**-loop retrieves elements (named v) from the domain of W and locates it in the domain of G (by $G\{v\}$), in set O (by $v \in O$), and in the domain of R (by $R\{v\}$). The inner **while**-loop retrieves elements (named s) from a image set of W and locates it in the domain of P (by $P\{s\}$) and in set F (by $s \notin F$). There are also associative accesses into the domain of the range of G (by $(G\{v\})\{a\}$) and the domain of the range of P (by $(P\{v\})\{a\}$). Finally, there are associative accesses into both the domains and ranges of both W (by $s2 \notin W\{v2\}$) and R (by $s2 \notin R\{v2\}$). Note that the kinds of accesses in the **while**-loops include also those needed for initialization before the **while**-loops.

We use a base $B1$ for the set of vertices in G and a base $B2$ for the set of states in P. The domain of G and set O are strongly based on $B1$, and the domain of P and set F are strongly based on $B2$. The domains of W and R are also strongly based on $B1$. However, due to associative accesses into the image sets of G, P, W, and R, there are more than a constant number of sets that also need to be strongly based. Therefore, the based-representation method does not completely apply. Nevertheless, we may use arrays for the image sets of G, P, W, and R. This still guarantees the worst-case constant running time for each primitive operation in (14).

The precise data structures are as follows. First, we have a set of records for vertices in G, and each record has the following fields:

1. a key representing the vertex;
2. an array for the image set of this vertex under G, indexed by labels (for outgoing edges), and where each element includes the head of a singly linked list of vertices (for target vertices following an edge with the indexing label) in G and includes a link that links elements in the image set in a singly linked list; plus a header for the linked list of elements in the image set;
3. a bit for whether the vertex is in O;
4. an array for the image set of this vertex under W, indexed by the states in P, and where each element includes a bit for whether the state is in the image set and includes a link that links elements in the image set in a singly linked list; plus a header for the linked list of elements in the image set;
5. an array for the image set of this vertex under R, indexed by the states in P, and where each element includes a bit for whether the state is in the image set.

We also have a set of records for states in P, and each record has the following fields:

1. a key representing the state;

2. an array for the image set of this state under P, indexed by labels (for outgoing transitions), and where each element includes the pointer to a state (target state following an edge with the indexing label) in P;
3. a bit for whether the state is in F.

We also link vertices in G in a singly linked list for traversal in the initialization of W and O. Finally, we use a singly linked list to link elements in $\mathbf{dom}(W)$ for traversal by the outer **while**-loop. Any other variable in the program is just a pointer to one of these two kinds of records.

Items 2 to 5 in the first set of records are for representing G, O, W, and R, respectively, and items 2 to 3 in the second set of records are for representing P and F, respectively. Note that the first three items in the two sets of records are similar, except that item 2 for P is simpler than item 2 for G, because P is deterministic, and because elements in the image sets for P do not need to be traversed. Also, item 5 is similar to item 4 but is simpler, because elements in the image sets for R do not need to be traversed.

It is easy to see that, with the above data structures, each primitive operation can be done in $\mathcal{O}(1)$ time. For example, to remove a vertex v from O, we only need to change the bit field for O in the record for v from 1 to 0.

Analysis of space complexity. Let N be the number of nodes in G; let S be the number of states in P, and let A be the number of distinct labels (i.e., the alphabet) in G and P. For the first set of records, item 2 together takes $\mathcal{O}(|G| + N * A)$ space, for edges in input graph G plus for each vertex the array indexed by labels; items 4 and 5 both take $\mathcal{O}(N * S)$ space. For the second set of records, item 2 together takes $\mathcal{O}(|P| + S * A)$ space, for transitions in the DFA P plus for each state the array indexed by labels. Others take space associated with parts of the above data structures. Therefore, the space complexity is

$$\mathcal{O}(|G| + |P| + N * A + S * A + N * S).$$

Note that the input size is $\Theta(|G| + |P|)$, and $reach(G \times P, [v0, s0])$ is of size $\mathcal{O}(N * S)$.

6 Discussion

This section discusses related issues about special cases in the input graph, about the computational complexity of the problem with respect to the regular-expression pattern, and about variables in regular-expression patterns.

Unreachable vertices and epsilon edges in input graph. There may be vertices in G that are not reachable from vertex $v0$. Since there is no path from $v0$ to these vertices, they satisfy any property about paths. Therefore, according to the specification of the problem, these vertices belong to the output set. Our algorithm computes exactly as specified and includes these vertices in the output. If, in some applications, we are not interested in those unreachable vertices. Starting from a slightly revised specification, we can easily obtain a slightly revised algorithm, using exactly the same derivation method.

There might be epsilon edges in G in general. In this case, one has to compute the product of G and P slightly differently. The idea is to add an epsilon edge from each state to itself in P before computing the product graph. After this addition, our method works as shown. Note that, in either case, our algorithm does not actually build the product of G and P. Building it would incur a substantial space and time overhead which is unacceptable for analyzing large graphs.

PSPACE-completeness with respect to input regular expression. We have used a DFA instead of the regular expression or its corresponding NFA to solve this problem. However, there is yet no polynomial time algorithm to convert a regular expression to a DFA. Is it possible to use the regular expression or its corresponding NFA directly and obtain an efficient overall algorithm?

We have proved that the regular path query problem is PSPACE-complete and thus NP-hard with respect to the regular expression. Basically, we can reduce the regular path query problem to the totality of regular expression problem [6], and vice versa, both in polynomial time. Since the totality of regular expression problem is PSPACE-complete [6], so is the regular path query problem. Therefore we consider using a DFA the best we can do for now.

In practice, typical in program analysis and model checking, the size of the input graph may be very large, but the size of the regular expression is small and is considered a constant [5]. Therefore, we can just convert the regular expression to a DFA and then use our derived algorithm.

We have skipped the step that converts a regular expression to a DFA, since there are standard algorithms. Those who are interested in this may find an algorithm by Chang and Paige [4], also derived using Paige's methods, that improves over previous algorithms for this.

Variables in regular-expression patterns. For applications in program analysis and model checking, it is often desirable to include variables in regular-expression patterns. Such a variable can match any labels in the program graph. To handle such variables, two extensions are needed.

First, the mapping between a pattern variable and the label it is instantiated to must be maintained so that multiple occurrences of the same variable are instantiated to the same label in the query result. Such a mapping is established when a variable is matched against a label for the first time; afterwards, all occurrences of the variable are treated as the instantiated label.

Second, all possible instantiations of all variables must be explored. One way of implementing this is to use backtracking; if coded in Prolog, this is achieved automatically by the Prolog engine. Another way is to maintain all possible instantiations at the same time, by incrementally adding new possible instantiations and removing failed instantiations. This achieves a kind of bottom-up computation.

A complete formal derivation that arrives at precise algorithmic steps and data structures, together with precise time and space complexity analysis, is yet to be studied. Based on our experience with derivation that exploits finite differencing in particular and incrementalization [7] in general, we believe that the derived algorithm would not be a backtracking algorithm; instead it would perform a bottom-up computation that exploits and maintains all instantiations incrementally at the same time.

7 Related work and conclusion

Tarjan showed, over two decades ago, that regular expressions provide a general approach for path analysis problems [18], and he gave efficient algorithms for constructing regular-expression patterns for various path problems [17]. The regular path query problem considered in this paper is a kind of inverse to the single-source path expression problem [17] Tarjan studied.

The regular path query problem was studied by de Moor et al. recently for program analysis and compiler optimization [5], where the problem was specified, and a high-level algorithm for solving it was derived, formally using calculus of relations and universal algebra. They analyze this high-level algorithm and conclude that the time complexity is linear in the size of the input graph. They also describe an implementation using a tabled Prolog system.

In contrast, we show how a high-level algorithm can be derived formally and easily using tools such as predicate logic. The initial specification using predicate logic corresponds rather directly to the given English description of the problem. We further derive a complete algorithm and data structures with precise analysis of both time and space complexity. We also describe related issues explicitly including the PSPACE-completeness of the problem with respect to the input regular expression.

The derivation of the complete algorithm and data structures uses Paige's methods [10, 14, 11, 3, 12, 2] with no invention, except that arrays have to be used since based representations [12, 2] do not apply to this problem completely. Even though Paige's method for data structure selection advocates the use of no arrays but pointers for low-level implementation, we found through our experience that arrays are necessary for many applications not only to achieve better asymptotic complexity but also to reduce significant constant factors. Systematic methods for the use of arrays in place of pointers, with precise analysis of time and space trade offs, is an important subject that needs further study.

Another important issue is how to find an appropriate high-level specification to start a derivation. Paige's methods are completely systematic for derivations starting with set-based fixed-point specifications. While such specifications can be obtained straightforwardly for many program analysis problems, it is not true in general, which is the case for the problem considered in this paper. The fact that obtaining a high-level algorithmic solution for this problem requires substantial exposition in [5] confirms that this is not a trivial issue. We make this step easier for this problem using tools such as predicate logic which captures the English description of the problem more directly.

Much previous work has studied specification and derivation starting with predicate logic, e.g., deductive synthesis [8]. However existing techniques are not systematic or automatable, unlike Paige's methods (when they apply of course) and system [13]. How to make these methods and techniques more systematic is a subject for future research.

Acknowledgments. We would like to thank Ernie Cohen for initially suggesting the use of predicate logic for the high-level specification and derivation. We are grateful to Ker-I Ko for his advice on how to explain and prove PSPACE-complete problems precisely. We thank Eerke Boiten and anonymous referees whose detailed comments helped improve the paper.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.
2. J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*, pages 126–164. North-Holland, Amsterdam, 1991.
3. J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.
4. C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. *Theoret. Comput. Sci.*, 178(1-2):1–36, May 1997.
5. O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queires, Nov. 2001. Revised and being reviewed for *Special Issue of Higher-Order and Symbolic Computation* dedicated to Bob Paige.
6. D.-Z. Du and K.-I. Ko. *Theory of Computational Complexity.* John Wiley & Sons, 2000.
7. Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.
8. Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, Reading, Mass., 1993.
9. M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 331–354. Springer-Verlag, Berlin, Sept. 1999.
10. R. Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 6 of *Computer Science and Artificial Intelligence.* UMI Research Press, Ann Arbor, Michigan, 1981. Revision of Ph.D. dissertation, New York University, 1979.
11. R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.
12. R. Paige. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*, pages 69–73. Canadian Scholars Press, 1989. Proceedings of ICCI '89: The International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

13. R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Proceedings of Joint 6th International Conference on Programming Languages: Implementations, Logics and Programs and 4th International Conference on Algebraic and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer-Verlag, Berlin, Sept. 1994.
14. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.
15. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, Berlin, New York, 1986.
16. W. K. Snyder. The SETL2 Programming Language. Technical report 490, Courant Institute of Mathematical Sciences, New York University, Sept. 1990.
17. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.
18. R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, July 1981.