# Finding Regular Simple Paths in Graph Databases

*Alberto O. Mendelzon*
*Peter T. Wood*[†]

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

## ABSTRACT

We consider the following problem: given a labelled directed graph $G$ and a regular expression $R$, find all pairs of nodes connected by a simple path such that the concatenation of the labels along the path satisfies $R$. The problem is motivated by the observation that many recursive queries can be expressed in this form, and by the implementation of a query language, G+, based on this observation. We show that the problem is in general intractable, but present an algorithm than runs in polynomial time in the size of the graph when the regular expression and the graph are free of *conflicts*. We also present a class of languages whose expressions can always be evaluated in time polynomial in the size of both the database and the expression, and characterize syntactically the expressions for such languages.

## 1. INTRODUCTION

The design of our query language G+ [CMW87, CMW88] is based on the observation that many of the recursive queries that arise in practice—and in the literature—amount to graph traversals. For example, see [Agra87, GSS87, R*86]. In G+, we view the database as a directed, labelled graph, and pose queries which are graph patterns; the answer to a query is the set of subgraphs of the database that match the given pattern. In our prototype implementation, queries are drawn on a workstation screen and the database and query results are also displayed pictorially.

EXAMPLE 1.1: Let $G$ be a graph representing airline flights: the nodes of $G$ denote cities, and an edge labelled $a$ from city $b$ to city $c$ means that there is a flight from $b$ to $c$ with airline $a$. Assume that we want to find all pairs of cities that are connected by a sequence of flights such that (a) at least one flight is with Air Canada (AC), and (b) no city is visited more than once.

This query can be expressed by the graph pattern in Figure 1.1.

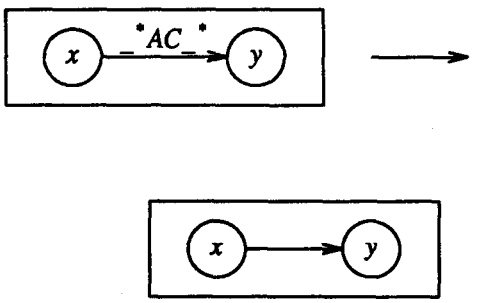


**Figure 1.1.** A graphical query.

The first box in the figure contains the pattern graph, while the second box contains the summary graph which specifies how the output is to be presented to the user. The edges of a pattern graph can be labelled with regular expressions; in this case the desired expression is $\_^*AC\_^*$ (where the underscore matches any edge label in $G$). This regular expression is used to match the edge labels along *simple* paths in $G$, thereby satisfying our original request. □

Figure 1.2 shows two screens from the $G$ prototype. In the first screen, one window contains a database graph where the nodes are cities and the edges are flights between cities, labelled with airline names. The other window contains a query graph asking for all pairs of nodes connected by a simple path of a certain form. The second screen shows one of the answers, displayed by shading nodes of the database graph. Instead of using a summary graph, the user has chosen from a menu to display only the endpoints of each path.

Although queries in G+ can be a lot more general than exemplified above, this special case is challenging enough from an algorithmic point of view if we want to process queries efficiently. The problem addressed in this paper is: given a regular expression $R$ and a graph $G$, find

[†] Current address: Department of Computer Science, University of Cape Town, South Africa.





Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

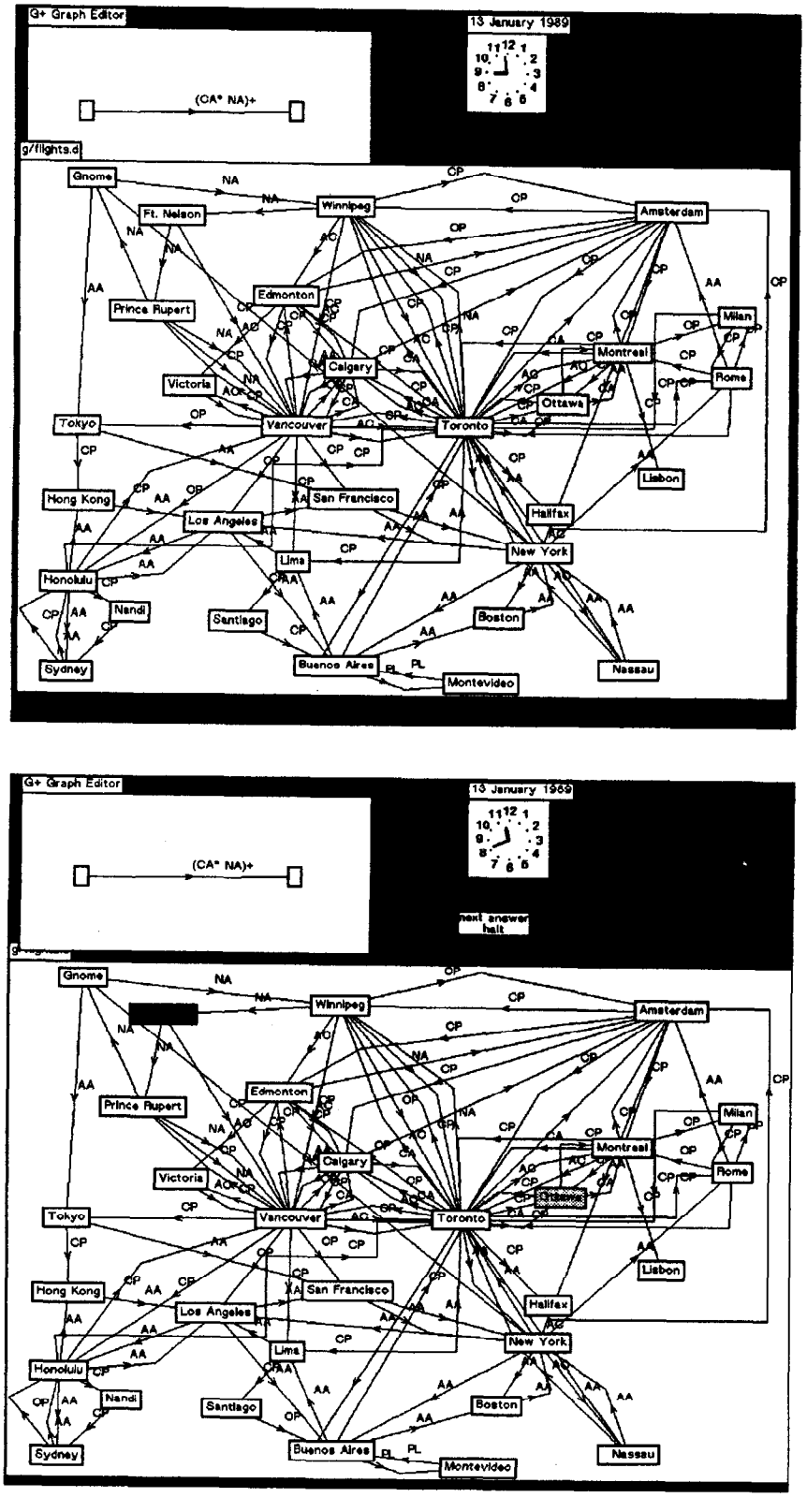


**Figure 1.2. The G+ user interface**

all pairs of nodes in $G$ which are connected by a simple path $p$, where the concatenation of edge labels comprising $p$ is in the language denoted by $R$.

When trying to find an efficient solution for this problem to incorporate in our implementation of $G^+$, we were somewhat surprised to discover that the query of Example 1.1 is in fact NP-complete. Using results in [FHW80, LaPa84], we show in Section 2 that for certain fixed regular expressions (such as $R$ in the above example), the problem of deciding whether a pair of nodes is in the answer of a query is NP-complete, making the general problem NP-hard. We first attacked this problem by determining what it is in the language of $R$ that makes the problem hard. In Section 3, we present a class of languages for which query evaluation is solvable in time polynomial in both the length of the regular expression and the size of the graph. We characterize these languages syntactically in terms of the regular expressions that denote them and the finite automata that recognize them. We then designed a general algorithm, presented in Section 4, which is correct for arbitrary graphs and queries, and is guaranteed to run in polynomial time in the size of the graph if the regular expression and graph are free of "conflicts", in a sense to be defined precisely in that section. As special cases, any query is free of conflicts with any acyclic database graph and any restricted expression query is free of conflicts with any arbitrary graph. Since we cannot restrict our prototype to work only on conflict-free queries and graphs, and it is expensive to test for conflict-freedom beforehand, it is quite convenient to have a single algorithm that works in all cases, and we have in fact incorporated the algorithm of Section 4 into our implementation.

## 2. INTRACTABILITY RESULTS

We begin by defining the graph structures as well as the class of queries over these structures in which we are interested.

DEFINITION: A *database graph* (*db-graph*, for short) $G=(N, E, \psi, \Sigma, \lambda)$ is a directed, labelled graph, where $N$ is a set of *nodes*, $E$ is a set of *edges*, and $\psi$ is an *incidence function* mapping $E$ to $N\times N$. So multiple edges between a pair of nodes are permitted in db-graphs. The labels of $G$ are drawn from the finite set of symbols $\Sigma$, called the *alphabet*, and $\lambda$ is an *edge labelling function* mapping $E$ to $\Sigma$.

DEFINITION: Let $\Sigma$ be a finite alphabet disjoint from $\{\varepsilon,\varnothing,(,)\}$. A *regular expression* $R$ over $\Sigma$ and the language $L(R)$ *denoted* by $R$ are defined in the usual way. Let $G=(N, E, \psi, \Sigma, \lambda)$ be a db-graph and $p=(v_1,e_1,\ldots,e_{n-1},v_n)$, where $v_i\in N$, $1\le i\le n$, and $e_j\in E$, $1\le j\le n-1$, be a path in $G$. We say $p$ is a *simple path* if all the $v_i$'s are distinct for $1<i<n$. We call the string $\lambda(e_1)\cdots\lambda(e_{n-1})$ the *path label* of $p$, denoted by $\lambda(p)\in\Sigma^*$. Let $R$ be a regular expression over $\Sigma$. We say that the path $p$ *satisfies* $R$ if $\lambda(p)\in L(R)$. The *query* $Q_R$ on db-graph $G$, denoted by $Q_R(G)$, is defined as the set of pairs $(x,y)$ such that there is a simple path from $x$ to $y$ in $G$ which satisfies $R$.
If $(x,y)\in Q_R(G)$, then $(x,y)$ *satisfies* $Q_R$.

A naive method for evaluating a query $Q_R$ on a db-graph $G$ is to traverse every simple path satisfying $R$ in $G$ exactly once. The penalty for this is that such an algorithm takes exponential time when $G$ has an exponential number of simple paths. Nevertheless, we will see below that in general we cannot expect an algorithm to perform much better, since we prove that, for particular regular expressions, the problem of deciding whether a pair of nodes is in the answer of a query is NP-complete. On the other hand, refinements can lead to guaranteed polynomial time evaluation under conditions studied in the following two sections.

Consider the following decision problem.

REGULAR SIMPLE PATH
*Instance:* Db-graph $G=(N, E, \psi, \Sigma, \lambda)$, nodes $x,y\in N$, regular expression $R$ over $\Sigma$.
*Question:* Does $G$ contain a directed simple path $p=(e_1,\ldots,e_k)$ from $x$ to $y$ such that $p$ satisfies $R$, that is, $\lambda(e_1)\lambda(e_2)\cdots\lambda(e_k)\in L(R)$?

This is equivalent to asking "Is $(x,y)\in Q_R(G)$?". When the instance comprises only the db-graph, we refer to the problem as FIXED REGULAR PATH(R). That is, for FIXED REGULAR PATH(R) we measure the complexity only in terms of the size of the db-graph. We prove below that, for certain regular expressions $R$, FIXED REGULAR PATH(R) is NP-complete (which implies that REGULAR SIMPLE PATH is NP-hard).

THEOREM 2.1: *Let* 0 *and* 1 *be distinct symbols in* $\Sigma$. *FIXED REGULAR PATH(R), in which R is either (1)* $(00)^*$, *or (2)* $0^*10^*$, *is NP-complete.*

PROOF: Consider the problem EVEN PATH, which is "Given a directed graph $G=(N,E)$ and nodes $x,y\in N$, is there a directed simple path of even length from $x$ to $y$?", and DISJOINT PATHS, which is "Given a directed graph $G=(N,E)$ and pairs of distinct nodes $(w,x),(y,z)\in N\times N$, is there a pair of disjoint directed simple paths in $G$, one from $w$ to $x$ and the other from $y$ to $z$?". For (1), the reduction is from EVEN PATH, which is shown to be NP-complete in [LaPa84]. For (2), the reduction is from DISJOINT PATHS, whose NP-completeness follows immediately from results in [FHW80]. □

Theorem 2.1 is a rather negative result, since it implies that queries might require time which is exponential in the size of the db-graph, not only the regular expression, for their evaluation. Thus, for regular expressions such as those in Theorem 2.1, we certainly would not expect an evaluation algorithm to run in polynomial time. This result, however, is not a function of the complexity of the particular regular expression but rather of the nature of the language denoted by the regular expression. A class of languages for which REGULAR SIMPLE PATH is in P is the subject of the next section.

## 3. RESTRICTED REGULAR EXPRESSIONS

We first introduce some terminology and definitions.

DEFINITION: A *nondeterministic finite automaton* (NDFA) $M$ is a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where $S$ is a finite set of *states*, $\Sigma$ is the *input alphabet*, $\delta$ is the *state transition function*, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is the set of *final states*. The concepts of the *extended* transition function $\delta^*$, the language $L(M)$ *accepted* by $M$, and the *transition graph* associated with $M$ are defined in the standard way, as is the notion of a *deterministic finite automaton* (DFA) [HoUl79].

DEFINITION: Given an NDFA $M=(S, \Sigma, \delta, s_0, F)$, for each pair of states $s,t \in S$, we define the *language from s to t*, denoted by $L_{st}$, as the set of strings that take $M$ from state $s$ to state $t$. In particular, for a state $s \in S$, the *suffix language* of $s$, denoted by $L_{sF}$ (or $[s]$, for short), is the set of strings that take $M$ from $s$ to some final state. Clearly, $[s_0]=L(M)$. Similar definitions apply for a DFA.

Given a regular expression $R$ over $\Sigma$, an $\varepsilon$-free NDFA $M=(S, \Sigma, \delta, s_0, F)$ that accepts $L(R)$ can be constructed in polynomial time [AHU74]. From now on, we will assume that all NDFAs are $\varepsilon$-free.

EXAMPLE 3.1: Figure 3.2 shows the transition graph $T$ of a DFA $M$. State 0 is the initial state of $M$, while all states are final (denoted by a double circle). (We do not show (reject) states that are not on some path from the inital state to a final state.) $L(M)$ is denoted by the regular expression $0^*+1^*+0^*1$. The suffix language of state 1 is $[1]=0^*+0^*1$, while $[2]=\varepsilon$. □

Let $R_1$ and $R_2$ be regular expressions. In the subsequent analysis, it will be useful to refer to an NDFA which accepts the language $L(R_1 \cap R_2)$. The construction of such an NDFA is defined as follows.

DEFINITION: Let $M_1=(S_1,\Sigma,\delta_1,p_0,F_1)$ and $M_2=(S_2,\Sigma,\delta_2,q_0,F_2)$ be NDFAs. The NDFA for $M_1 \cap M_2$ is $I=(S_1 \times S_2, \Sigma, \delta, (p_0,q_0), F_1 \times F_2)$, where, for $a \in \Sigma$, $(p_2,q_2) \in \delta((p_1,q_1),a)$ if and only if $p_2 \in \delta_1(p_1,a)$ and $q_2 \in \delta_2(q_1,a)$. We call the transition graph of $I$ the *intersection graph* of $M_1$ and $M_2$.

We saw in the previous section that, for certain regular expressions $R$, it is very unlikely that we will find an algorithm for evaluating $Q_R$ on an arbitrary graph $G$ that will always run in time polynomial in the size of $G$. One such regular expression is $0^*10^*$. However, it turns out that if the regular expression $R=0^*10^*+0^*$ is specified instead, then $Q_R$ is evaluable in polynomial time on any db-graph $G$. The reason is that if there is an arbitrary path from node $x$ to node $y$ in $G$ that satisfies $R$, then there is a simple path from $x$ to $y$ satisfying $R$. In such a case, we need not restrict ourselves to looking only for simple paths in $G$, but can instead look for *any* path satisfying $R$. We define the corresponding decision problem below.

*REGULAR PATH*
*Instance:* Db-graph $G=(N, E, \psi, \Sigma, \lambda)$, nodes $x,y \in N$, regular expression $R$ over $\Sigma$.
*Question:* Does $G$ contain a directed path (not necessarily simple) $p=(e_1, \ldots, e_k)$ from $x$ to $y$ such that $p$ satisfies $R$, that is, $\lambda(e_1)\lambda(e_2)\cdots\lambda(e_k) \in L(R)$?

LEMMA 3.1: *REGULAR PATH can be decided in polynomial time.*

PROOF: Given db-graph $G$ along with nodes $x$ and $y$ in $G$, we can view $G$ as an NDFA with initial state $x$ and final state $y$. Construct the intersection graph $I$ of $G$ and $M=(S, \Sigma, \delta, s_0, F)$, an NDFA accepting $L(R)$. There is a path from $x$ to $y$ satisfying $R$ if and only if there is a path in $I$ from $(x,s_0)$ to $(y,s_f)$, for some $s_f \in F$. All this can be done in polynomial time [HRS76]. □

We are interested in conditions under which REGULAR SIMPLE PATH (which is appropriate because of our semantics) can be reduced to REGULAR PATH. The following lemma states one such condition.

LEMMA 3.2: *REGULAR SIMPLE PATH can be decided in polynomial time on acyclic db-graphs.*

PROOF: Follows immediately from Lemma 3.1 and the fact that every path in an acyclic graph is simple. □

Suppose that we want to characterize a class of regular expressions for which we can guarantee that REGULAR SIMPLE PATH reduces to REGULAR PATH and hence is solvable in polynomial time. If we assume that we know nothing about the structure of the db-graphs, we have to ensure that, for such a regular expression $R$, whenever string $w$ is in $L(R)$, every string obtainable from $w$ by removing one or more symbols must also be in $L(R)$. Otherwise, if $w=xay$ is in $L(R)$ but $xy$ is not in $L(R)$ (where $a \in \Sigma$ and $x,y \in \Sigma^*$), we can construct a graph $G$ comprising a single simple path from $u$ to $v$ and passing through $z$, in which

there is a loop at $z$ labelled $a$, the path from $u$ to $z$ is labelled $x$, and the path from $z$ to $v$ is labelled $y$ (see Figure 3.1). There is a non-simple path from $u$ to $v$ in $G$ that satisfies $R$ but no simple path from $u$ to $v$ satisfying $R$.

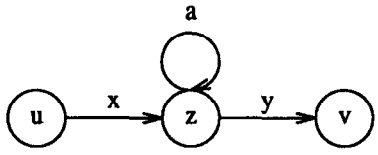


**Figure 3.1.**

DEFINITION: ( [Carr79] ) An *abbreviation* of a string $w$ is any string that can be obtained from $w$ by removing one or more symbols of $w$.

So we are looking for a class of regular expressions which denote languages that are closed under abbreviation. Now consider the following definition for the class of restricted regular expressions.

DEFINITION: For $a \in \Sigma$, denote the regular expression $(a+\varepsilon)$ by $(a?)$ (as is done in the *grep* utility of Unix[†], for example). Given a regular expression $R$, let $R'$ be a regular expression obtained by replacing some occurrence of a symbol $a \in \Sigma$ in $R$ by $(a?)$. $R$ is *restricted* if and only if $R \equiv R'$, for any $R'$ obtained from $R$ as defined above.

EXAMPLE 3.2: The regular expression $R_1 = 0^*+1^*+0^*1$ is restricted, since $R_1$ can be rewritten as $0^*(1?)+1^*$, which is equivalent to $(0?)^*(1?)+(1?)^*$. Recall, from Theorem 3.1, that FIXED REGULAR PATH(R) is NP-complete for $R = 0^*10^*$. $R$ is not restricted, but $R' = 0^*10^*+0^*$ is restricted, since $R'$ can be written as $0^*(1+\varepsilon)0^*$, which is equivalent to $(0?)^*(1?)(0?)^*$. □

DEFINITION: A DFA $M = (S, \Sigma, \delta, s_0, F)$ exhibits the *Suffix Language Containment Property* (the *Containment Property*, for short) if, for each pair $s,t \in S$ such that $s$ and $t$ are on a path from $s_0$ to some final state and $t$ is a successor of $s$, $[s] \supseteq [t]$ (that is, $L_{sF} \supseteq L_{tF}$).

EXAMPLE 3.3: Consider the regular expression $R_1 = 0^*+1^*+0^*1$ from the previous example, and a DFA $M$ accepting $L(R_1)$ whose transition graph $T$ is given in Figure 3.2. We can verify that $M$ exhibits the Containment Property by noting that [3] is denoted by $1^*$, [2] by $\varepsilon$, [1] by $0^*+0^*1$, and [0] by $0^*+1^*+0^*1$. Obviously, $[1] \supseteq [1]$ and $[3] \supseteq [3]$. It is easy to check that $[1] \supseteq [2]$, $[0] \supseteq [1]$ and $[0] \supseteq [3]$. The fact that $M$ exhibits the Containment

[†] Unix is a trademark of AT&T.

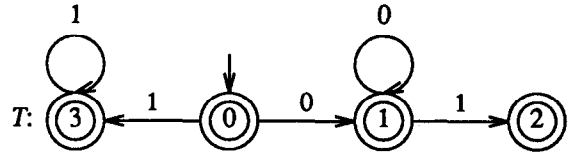


**Figure 3.2.** DFA for $0^*+1^*+0^*1$.

Property and $R_1$ is restricted is no coincidence, as the following theorem states. □

THEOREM 3.1: *Let $R$ be a regular expression over $\Sigma$, and $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$. The following three statements are equivalent:*

*(1) $R$ is a restricted regular expression,*
*(2) $L(R)$ is closed under abbreviations, and*
*(3) $M$ exhibits the Containment Property.*

PROOF: (Sketch)

(1) => (2) Assume that $R$ is restricted but that $L(R)$ is not closed under abbreviations. Then there is a symbol $a \in \Sigma$ and strings $x, y \in \Sigma^*$ such that $xay \in L(R)$ but $xy \notin L(R)$. Let $M_R = (T, \Sigma, \mu, t_0, E)$ be an NDFA that accepts the language $L(R)$. Let $T' = \mu^*(t_0, x)$, that is, the set of states $M_R$ can be in after reading $x$. Since $L(M_R) = L(R)$ and $xy \notin L(R)$, for no $r \in T'$ can $y$ be in $[r]$. On the other hand, $xay \in L(M_R)$, so there is a state $p \in T'$ such that $q \in \mu(p,a)$ and $y \in [q]$. Since $R$ is restricted, adding an $\varepsilon$-transition from $p$ to $q$ leaves $L(M_R)$ unchanged. But if we do so, then $y \in [p]$, $xy \in L(M_R)$, and $L(M_R)$ is no longer equal to $L(R)$, which is a contradiction. We conclude that $L(R)$ is closed under abbreviations.

(2) => (3) Assume that $[s] \not\supseteq [t]$ for some pair $s,t$ of reachable states in $M$ such that $\delta(s,a) = t$, for some $a \in \Sigma$. That is, there is a string $y \in \Sigma^*$ for which $y \in [t]$ but $y \notin [s]$. Let $x \in \Sigma^*$ be a string for which $\delta^*(s_0, x) = s$. It follows that $xay \in L(M)$, but that $xy \notin L(M)$. Since $L(M) = L(R)$, we conclude that $L(R)$ is not closed under abbreviations.

(3) => (1) Assume that $R$ is not restricted, and let $M_R$ be an NDFA accepting $L(R)$ as above. Then there is an $a$-transition in $M_R$ from $s$ to $t$ for which adding an $\varepsilon$-transition from $s$ to $t$ alters $L(M_R)$. Let $x \in \Sigma^*$ be a string for which $s \in \mu^*(t_0, x)$. That is, there is a string $y \in [t]$ such that $y \notin [r]$ for any $r \in \mu^*(t_0, x)$. Now consider the DFA $M$. Assume that $\delta^*(s_0, x) = p$. Since $L(M_R) = L(M)$ and $xay \in L(M_R)$, there must be a state $q$ in $M$ such that $\delta(p,a) = q$ and $y \in [q]$. However, $y \notin [p]$, for otherwise $xy \in L(M)$ which would mean that $L(M) \neq L(M_R)$. Hence, $[p] \not\supseteq [q]$, so $M$ does not exhibit the Containment Property. □

COROLLARY 3.1: *REGULAR SIMPLE PATH can be decided in polynomial time for restricted regular expressions.*

PROOF: By Theorem 3.1, if $R$ is a restricted regular expression, then $L(R)$ is closed under abbreviations, which implies that REGULAR SIMPLE PATH reduces to REGULAR PATH. By Lemma 3.1, REGULAR PATH can be decided in polynomial time. □

Note that we our results show that $R$ being restricted is a sufficient condition for REGULAR SIMPLE PATH being solvable in polynomial time, but it is not necessary. For example, the expression $01^*$ is not restricted, but the corresponding query can be evaluated in polynomial time on arbitrary graphs using the algorithm we present in Section 5. It would be interesting to find other naturally defined classes of regular expressions that lead to a polynomial time solution for REGULAR SIMPLE PATH.

By adapting an algorithm to minimize the number of states of a DFA [HoUl79], we can compute the suffix language containment relation for all pairs of states in a DFA $M$. This relation will be used in the next section; it also provides an obvious method for testing whether or not a regular expression $R$ is restricted (using Theorem 3.1). Since the construction of a DFA $M$ accepting $L(R)$ may take exponential time (in the size of $R$), this test is not efficient. However, it is important to stress that we are trying to avoid the possibility of spending exponential time in the size of the db-graph in answering a query. Also, we have the following.

THEOREM 3.2: *Let $R$ be a regular expression over alphabet $\{0\}$. Deciding whether $R$ is not restricted is NP-complete.*

PROOF: By reduction from the problem of deciding whether a regular expression does not denote $0^*$, which is shown to be NP-complete in [StMe73]. □

## 4. AN EVALUATION ALGORITHM

In this section, we describe an algorithm for evaluating a query $Q_R$ on a db-graph $G$. As is to be expected from the results of Section 2, the algorithm does not run in polynomial time in general. It does, however, run in polynomial time under the sufficient conditions identified in Section 4, namely, when $G$ is acyclic or $R$ is restricted. In fact, we show that the algorithm runs in polynomial time if $G$ and $R$ are conflict-free, a condition implied whenever $G$ is acyclic or $R$ is restricted.

The evaluation algorithm traverses simple paths in $G$, using a DFA $M$ accepting $L(R)$ to control the search by marking nodes as they are visited. We must record with which state of $M$ a node is visited, since we must allow a node to be visited with different states (which correspond to distinct nodes in the intersection graph of $G$ and $M$). In order to avoid visiting a node twice in the same state, we would like to retain the state markings on nodes as long as possible. Unfortunately, the following example shows that, in general, traversing only simple paths in $G$ and retaining state markings can lead to incompleteness in query evaluation.

EXAMPLE 4.1: Consider the query $Q_R$, where $R = aaa$. An automaton $M$ accepting $L(R)$ and a db-graph $G$ are shown in Figure 4.1.

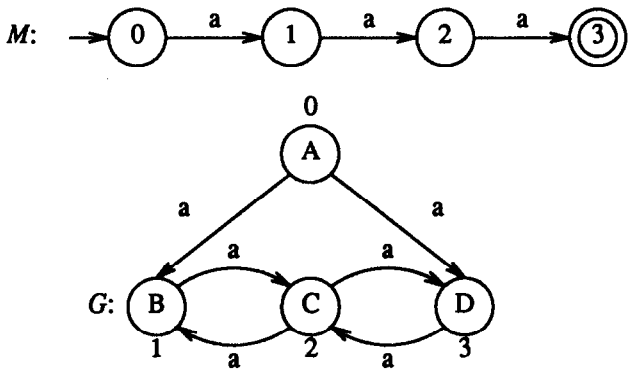


**Figure 4.1.** A DFA $M$ and db-graph $G$.

Assume that we start traversal from node $A$ in $G$, and follow the path to $B$, $C$ and $D$. Nodes $A$, $B$, $C$ and $D$ are marked with states 0, 1, 2 and 3, respectively (as shown in Figure 4.1), and the answer $(A,D)$ is found, since 3 is a final state. If we now backtrack to node $C$, we cannot mark $B$ with state 3 because $(A,B,C,B)$ is a non-simple path. So we backtrack to $A$, and visit $D$ in state 1. If we have retained markings, however, we cannot visit node $C$ as it is already marked with state 2. Consequently, the answer $(A,B)$ is not found. □

As we will see below, it is safe to retain markings when $G$ is acyclic or $R$ is restricted . However, we can do better; because of the structure of a particular db-graph $G$, it may be the case that we can retain markings and evaluate $Q_R$ in polynomial time even if $G$ is not acyclic and $R$ is not restricted.

DEFINITION: Let $I$ be the intersection graph of a db-graph $G$ and a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$. Assume that for nodes $u$ and $v$ in $G$ and states $s$ and $t$ in $M$, there are paths $p$ from $(u,s_0)$ to $(v,s)$ and $q$ from $(v,s)$ to $(v,t)$ in $I$ (that is, there is a cycle at $v$ in $G$ that satisfies $L_{st}$), such that no first component of a node on $p$ or $q$ repeats except for the endpoints of $q$. In other words, $p$ and $q$ correspond to a simple path and a simple cycle, respec-

tively, in $G$. If $[s] \not\supseteq [t]$, then we say there is a *conflict* between $s$ and $t$ at $v$. If there are no conflicts in $I$, then $I$ is said to be *conflict-free*, as are $G$ and $R$.

It is obvious that if $G$ is acyclic, then $I$ is conflict-free no matter what regular expression $R$ appears in $Q_R$. Also, if $R$ is restricted, then, by Theorem 3.1, $M$ exhibits the Containment Property; hence, $I$ is conflict-free irrespective of the structure of $G$. We will show that $Q_R$ can be evaluated in polynomial time if $I$ is conflict-free. Hence, conflict-freedom is a third (weaker) sufficient condition for $Q_R$ to be polynomial time evaluable. For example, if $R = (00)^*$ and $G$ is a bipartite (cyclic) graph, then $G$ and $R$ are conflict-free and so $Q_R$ can be evaluated in polynomial time on $G$ (in contrast to Theorem 2.1).

The concept of conflict detection is embodied in the following query evaluation algorithm, Algorithm C. If no conflicts are detected, the algorithm retains markings, while whenever a conflict arises, it unmarks nodes so that no answers are lost.

EXAMPLE 4.2: Consider again the DFA $M$ and the db-graph $G$ of Example 4.1. Recall that, if markings were retained, the answer $(A,B)$ would not be found. However, there is a conflict in the intersection graph of $G$ and $M$. This is because node $B$ in $G$ can be marked with state 1 and there is a cycle at $B$ which satisfies $L_{13}$, but $[1] \not\supseteq [3]$. Algorithm C detects such conflicts and unmarks node $C$ on backtracking, enabling the answer $(A,B)$ to be found. □

**Algorithm C**: Evaluation of a query on a db-graph.

*Input:* Db-graph $G = (N, E, \psi, \Sigma, \lambda)$, query $Q_R$.
*Output:* $Q_R(G)$, the value of $Q_R$ on $G$.
*Method:*

1. Construct an DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$.
2. Initialize $Q_R(G)$ to $\varnothing$.
3. For each node $v \in N$, set $CM[v]$ to *null* and $PM[v]$ to $\varnothing$.
4. Test $[s] \supseteq [t]$ for each pair of states $s$ and $t$.
5. For each node $v \in N$,
   (a) call SEARCH-C($v$, $v$, $s_0$, *conflict*)
   (see Figure 4.2)
   (b) reset $PM[w]$ to $\varnothing$ for any marked node $w \in N$.

The algorithm marks nodes in $G$ with states of the DFA $M$ when they are visited. Two types of marking are used for each node $v$: (1) a current marking ($CM[v]$), which (if non-null) tells us that $v$ is already on the stack (Lines 7 and 14), and (2) a previous marking ($PM[v]$), which is a set of states and tells us earlier markings of $v$, excluding the current path (Line 15). Current markings are used to avoid following paths in $G$ that are not simple, while previous markings are used where possible to

```
procedure SEARCH-C (u, v, s, var conflict)
6.  conflict ← false
7.  CM[v] ← s
8.  if s ∈ F then Q_R(G) ← Q_R(G) ∪ {(u,v)} fi
9.  for each edge in G from v to w with label a do
10.   if δ(s,a) = t and t ∉ PM[w] then
11.     if CM[w] = q then conflict ← ([q] ⋣ [t]) fi
      else /* CM[w] is null */
12.     SEARCH-C (u, w, t, new-conflict)
13.     conflict ← conflict or new-conflict
      fi
    od
14. CM[v] ← null
15. if not conflict then PM[v] ← PM[v] ∪ {s} fi
  end SEARCH-C
```

**Figure 4.2.** Evaluation of a query on a db-graph.

prevent a node in $G$ from being visited more than once in the same state during a single execution of Line 5(a). We do not describe the algorithm further here, but rely on a detailed example presented below to demonstrate how query evaluation, and in particular the detection of conflicts, is performed. □

The following theorem is proved in [Wood88].

THEOREM 4.1: *Algorithm C is correct; that is, given db-graph $G$ and query $Q_R$, Algorithm C adds $(x,y)$ to $Q_R(G)$ if and only if $(x,y)$ satisfies $Q_R$.*

THEOREM 4.2: *In the absence of conflicts, Algorithm C runs in an amount of time which is bounded by a polynomial in the size of the db-graph.*

PROOF: Let $G$ be a db-graph with $n$ nodes and $e$ edges. In the absence of conflicts, Algorithm C essentially performs $n$ depth-first searches of the graph, and hence runs in $O(ne)$ time. □

EXAMPLE 4.3: Let $R = a((bc + \varepsilon)d + ec)$ be the regular expression for query $Q_R$. A DFA $M$ accepting $L(R)$ and a db-graph $G$ are shown in Figure 4.3. We demonstrate the execution of Algorithm C in evaluating $Q_R$ on $G$. Assume that we start by marking node $A$ of $G$ with state 0 of $M$, after which we proceed to mark $B$ with 1, $C$ with 2, and $D$ with 3. Since no edge labelled $d$ leaves $D$, we backtrack to $C$ and attempt to visit $B$ in state 3. Although $B$ already has a current marking ($CM[B] = 1$), this is not a conflict since $[1] \supseteq [3]$. The algorithm now backtracks to node $B$ in state 1 and marks $E$ with state 4. After backtracking again to $B$ in state 1, the markings are given as in Figure 4.3. Next, the algorithm marks $C$ with state 5 and $D$ with 4. On backtracking to $C$ and attempting to mark $B$ with 4, a conflict is detected since $[1] \not\supseteq [4]$. So on back-

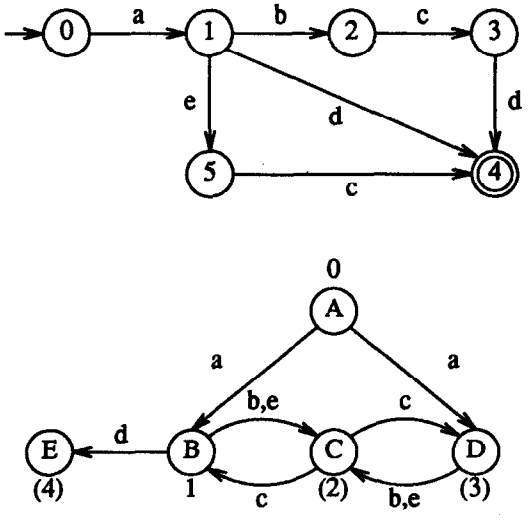


**Figure 4.3.** A DFA $M$ and marked db-graph $G$ ($PM$s are in parentheses).

tracking to $A$, the marking 5 is removed from $C$, 1 is removed from $B$, and neither is made a previous marking. Now $D$ is marked with 1, but since $C$ has a previous marking of 2, that marking is not repeated. So $C$ is marked with 5 (which was previously removed), after which $B$ can be marked with 4. When the algorithm backtracks to $C$ and attempts to visit $D$, it discovers that $D$ was previously marked with 4, so no conflict is registered. The markings are now given as in Figure 4.4. No more markings are generated by Algorithm C for the traversal rooted at $A$, so the set of answers with first component $A$ is $\{(A,B),(A,D),(A,E)\}$. □

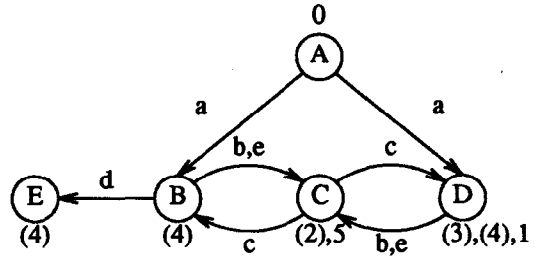


**Figure 4.4.** The final set of markings for Example 4.3.

If $R$ is restricted or $G$ is acyclic, no conflicts occur so Algorithm C runs in polynomial time. For the same reason, the algorithm also runs in polynomial time on the example of finding even paths in a bipartite graph. However, even in the presence of conflicts, Algorithm C can run in polynomial time in the size of $G$. This is the case, for example, if $R$ is a (*)-free regular expression. In this case, $L(R)$ is finite; say the longest string in the language is of length $q$. Then there can be at most $O(n^q)$ simple paths in $G$, and in the worst case the algorithm traverses each one exactly once.

## 5. CONCLUSION

We have addressed the problem of finding nodes in a labelled, directed graph which are connected by a simple path satisfying a given regular expression. This study was motivated by the observation that many recursive queries can be expressed in this form, and by the implementation of a query language based on this observation.

We began by describing how a naive algorithm might evaluate such queries. Although this algorithm runs in exponential time in the worst case, we showed that we cannot expect to do better since the evaluation problem is in general NP-hard. Using the fact that the associated problem for paths in general (as opposed to simple paths) is solvable in polynomial time, we characterized the class of restricted regular expressions, whose associated queries can be evaluated in polynomial time. Finally, we presented an algorithm for evaluating arbitrary expressions on arbitrary graphs. This algorithm runs in polynomial time if either (a) the regular expression is restricted or closure-free, (b) the graph is acyclic, or (c) the regular expression and graph are conflict-free.

We should point out that the analysis in this paper, and the implementation itself, assume the graph can be entirely stored in main memory. This is a reasonable assumption in many cases, especially because in the intended applications of $G^+$ the graph is often only the fraction of the database that can be presented visually in a natural way. Relaxing this assumption provides an interesting area for further study. Other researchers, investigating similar algorithms for transitive closure, have claimed that they are amenable to efficient secondary storage implementation [IoRa88]. Our next version of the implementation will use the *HAM* hypertext package developed at Tektronix [DeSc86] as a graph storage server; this package maintains a graph on secondary storage, providing an efficient implementation of the graph-based operations that are the primitive steps of our algorithm.

Finally, we note that the problem of solving general $G$ queries is more general than REGULAR SIMPLE PATH. For example, the language contains also aggregate operators that allow the expression of queries such as "find the length of the shortest path between two nodes." Queries with aggregates can be intractable independently of the properties of the regular expression; for example, one can also express the query "find the length of the *longest* sim-

ple path between two nodes, '' which is NP-complete. Further discussion of aggregate operators can be found in [CrMe88]. A new language, that does not insist on simple path semantics and captures exactly the queries computable in logarithmic space, is presented in [Cons89].

## ACKNOWLEDGEMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada. We thank Mariano Consens and the anonymous reviewers for useful comments.

## References

Agra87.
R. AGRAWAL, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," *Proc. 3rd Int. Conf. on Data Engineering*, pp. 580-590, 1987.

AHU74.
A.V. AHO, J.E. HOPCROFT, AND J.D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

Carr79.
B. CARRE, *Graphs and Networks*, Oxford University Press, Oxford, England, 1979.

Cons89.
M.P. CONSENS, "Graphlog: 'Real Life' Recursive Queries Using Graphs," M.Sc. thesis, Department of Computer Science, University of Toronto, 1989.

CrMe88.
I.F. CRUZ AND A.O. MENDELZON, "Summary Queries in the G+ Query Language," in *Office and Data Base Systems Research '88*, ed. F.H. Lochovsky, pp. 160-188, Tech. Report CSRI-212, Univ. of Toronto, 1988.

CMW87.
I.F. CRUZ, A.O. MENDELZON, AND P.T. WOOD, "A Graphical Query Language Supporting Recursion," *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 323-330, 1987.

CMW88.
I.F. CRUZ, A.O. MENDELZON, AND P.T. WOOD, "G+: Recursive Queries Without Recursion," *Proc. 2nd Int. Conf. on Expert Database Systems*, pp. 355-368, 1988.

DeSc86.
N. DELISLE AND M. SCHWARTZ, "Neptune: a Hypertext System for CAD Applications," *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 132-143, 1986.

FHW80.
S. FORTUNE, J. HOPCROFT, AND J. WYLLIE, "The Directed Subgraph Homeomorphism Problem," *Theor. Comput. Sci.*, vol. 10, pp. 111-121, 1980.

GSS87.
G. GRAHNE, S. SIPPU, AND E. SOISALON-SOININEN, "Efficient Evaluation for a Subset of Recursive Queries," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 284-293, 1987.

HoUl79.
J.E. HOPCROFT AND J.D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

HRS76.
H.B. HUNT, D.J. ROSENKRANTZ, AND T.G. SZYMANSKI, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-free Languages," *J. Comput. Syst. Sci.*, vol. 12, pp. 222-268, 1976.

IoRa88.
Y.E. IOANNIDIS AND R. RAMAKRISHNAN, "Efficient Transitive Closure Algorithms," Computer Sciences Tech. Report #765, Univ. of Wisconsin-Madison, 1988.

LaPa84.
A.S. LAPAUGH AND C.H. PAPADIMITRIOU, "The Even-Path Problem for Graphs and Digraphs," *Networks*, vol. 14, pp. 507-513, 1984.

R*86. A. ROSENTHAL, S. HEILER, U. DAYAL, AND F. MANOLA, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 166-176, 1986.

StMe73.
L.J. STOCKMEYER AND A.R. MEYER, "Word Problems Requiring Exponential Time," *Proc. 5th Ann. ACM Symp. on Theory of Computing*, pp. 1-9, 1973.

Wood88.
P.T. WOOD, "Queries on Graphs," Ph.D. thesis, Department of Computer Science, University of Toronto, Tech. Report CSRI-223, 1988.