FINDING REGULAR SIMPLE PATHS IN GRAPH DATABASES[*]

ALBERTO O. MENDELZON[†] AND PETER T. WOOD[‡]

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 in relational databases can be expressed in this form, and by the implementation of a query language, $\mathbf{G}^+$, 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 graph and the expression, and characterize syntactically the expressions for such languages.

Key words. Labelled directed graphs, NP-completeness, polynomial-time algorithms, regular expressions, simple paths

AMS(MOS) subject classifications. 68P, 68Q, 68R

1. Introduction. Much of the success of the relational model of data can be attributed to its simplicity, which makes it both amenable to mathematical analysis and easy for users to comprehend. In this latter respect, the availability of non-procedural query languages has been a great asset. However, the fact that queries which are especially useful in new application domains are not expressible in traditional query languages has led to proposals for more powerful query languages, such as the logic-based language *Datalog* [23] and our query language $\mathbf{G}^+$ [9, 10].

The original proposal for the relational model included two query languages of equivalent expressive power: the *relational calculus* and the *relational algebra* [7]. These languages have been used as the yardstick by which other query languages are classified; a query language is said to be *relationally complete* if it has (at least) the expressive power of the relational calculus. However, this notion of completeness has been questioned since it was shown that certain reasonable queries, such as finding the transitive closure of a binary relation, cannot be expressed in the calculus [3, 4]. This particular limitation is overcome in the languages $\mathbf{G}^+$ and Datalog through their ability to express recursive queries.

The design of $\mathbf{G}^+$ 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, [1, 12, 19]). In $\mathbf{G}^+$, 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. Useful applications for such a language can be found in systems representing transportation networks, communication networks, hypertext documents, and so on. In our prototype implementation, queries are drawn on a workstation screen and the database and query results are also displayed pictorially.

Example 1. Let G be a graph describing a hypertext document: nodes are chunks of text and edges are links (cross-references). Readers read the document by following links. In this context, one might be interested in a query such as: Is there a way to get from Section 3.1 to Section 5.2 and then to the Conclusion, without reading any node more than once? The corresponding $\mathbf{G}^+$ query is shown in Figure 1. The left-hand box in the figure contains the pattern graph, while the right-hand box contains the summary graph which specifies how the

[*] A preliminary version of sections of this paper appeared in *Proceedings of the 15th International Conference on Very Large Data Bases* (Amsterdam, The Netherlands, August 22–25, 1989), pages 185–193.

[†] Computer Systems Research Institute, University of Toronto, Toronto, Ont. M5S 1A4, Canada.

[‡] Department of Computer Science, University of Cape Town, Rondebosch 7700, South Africa

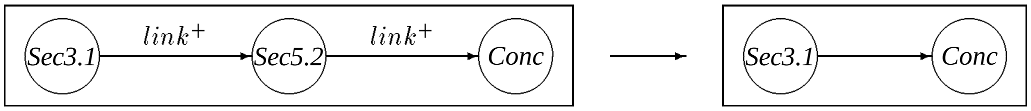


FIG. 1. *Query to test for the existence of a simple path in a hypertext document.*

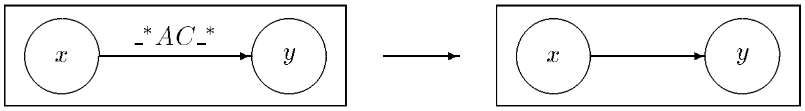


FIG. 2. *Query to find pairs of cities connected by some Air Canada flight.*

output is to be presented to the user. The nodes in this case are labelled with constants to be matched with those in the database. The edges of a pattern graph can be labelled with regular expressions; in this case the desired expression is $link^+$, representing a nonzero sequence of links. This regular expression is used to match the edge labels along *simple* paths in G, thereby satisfying our original request. ∎

Example 2. 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 of Figure 2. The pattern graph in this example comprises only two nodes, this time labelled with variables, while the edge is labelled with the regular expression $\_^*AC\_^*$ (where the underscore matches any edge label in G, and AC is regarded as a single symbol). Once again, the fact that only simple paths are matched during query evaluation ensures that the desired answer is computed. ∎

Although queries in **G**$^+$ can be a lot more general than exemplified in the above two examples, the special case suggested by Example 2 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 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 queries of Examples 1 and 2 are in fact both NP-complete. Using results in [11, 17], we show in §2 that for certain fixed regular expressions (such as R in Example 2), 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 §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. This characterization assumes we have no knowledge concerning the structure of the graph being queried. In §4, we consider extensions where we are given a constraint which the cycles of the input graph are known to satisfy. This knowledge allows us to characterize potentially larger classes of queries which can be solved in polynomial time.

We then designed a general algorithm, presented in §5, 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 §5 into our implementation.

2. Intractability Results.

In this section, we prove some negative results regarding the complexity of finding certain types of simple paths in a particular class of directed graphs. We begin by defining the graph structures as well as the class of queries over these structures in which we are interested.

DEFINITION 1. 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 ψ is an *incidence function* mapping E to $N \times N$. Note that multiple edges between a pair of nodes are permitted in db-graphs. The labels of G are drawn from the finite set of symbols Σ, called the *alphabet*, and λ is an *edge labelling function* mapping E to Σ.

DEFINITION 2. Let Σ be a finite alphabet disjoint from $\{\epsilon, , (,)\}$. A *regular expression* R over Σ is defined as follows.

1. The *empty string* ϵ, the *empty set* $\emptyset$, and each $a \in \Sigma$ are regular expressions.
2. If A and B are regular expressions, then $(A + B)$, AB, and $(A)^*$ are regular expressions.
3. Nothing else is a regular expression.

The expression $(A+B)$ is called the *alternation* of A and B, (AB) is called the *concatenation* of A and B, and $(A)^*$ is called the *closure* of A. We use the underscore (_) to denote the alternation of all elements of Σ. Also, A^+ denotes AA^*, the *positive closure* of A.

The language $L(R)$ *denoted by* R is defined as follows.

1. $L(\epsilon) = \{\epsilon\}$.
2. $L(\emptyset) = \emptyset$.
3. $L(a) = \{a\}$, for $a \in \Sigma$.
4. $L(A + B) = L(A) \cup L(B) = \{w \mid w \in L(A)$ or $w \in L(B)\}$.
5. $L(AB) = L(A)L(B) = \{w_1w_2 \mid w_1 \in L(A)$ and $w_2 \in L(B)\}$.
6. $L(A^*) = \cup_{i=0}^{\infty} L(A)^i$, where $L(A)^0 = \{\epsilon\}$ and $L(A)^i = L(A)^{i-1}L(A)$.

Regular expressions R_1 and R_2 are *equivalent*, written $R_1 \equiv R_2$, if $L(R_1) = L(R_2)$. The *length* of regular expression R, denoted $|R|$, is the number of symbols appearing in the string R.

DEFINITION 3. Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph and $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$, where $v_i \in N$, $1 \leq i \leq n$, and $e_j \in E$, $1 \leq j \leq n - 1$, be a path (not necessarily a simple path) in G. 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 Σ. We say that the path p *satisfies* R if $\lambda(p) \in L(R)$. The *query* Q_R on db-graph 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 $x, y \in N$, and regular expression R over Σ.
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 first prove below that, for certain regular expressions R, FIXED REGULAR PATH(R) is NP-complete. In doing so, we will refer to the following two decision problems.

EVEN PATH
Instance: Directed graph $G = (N, E)$, and nodes $x, y \in N$.
Question: Is there a directed simple path of even length (that is, with an even number of edges) from x to y?

DISJOINT PATHS
Instance: Directed graph $G = (N, E)$, and two pairs of distinct nodes $(w, x), (y, z) \in N \times N$.
Question: Is there a pair of disjoint directed simple paths in G, one from w to x and the other from y to z?

The following theorem uses the above two decision problems to prove the NP-completeness of FIXED REGULAR PATH(R) for two particular regular expressions.

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

Proof. (1) In [17], EVEN PATH is shown to be NP-complete. We can reduce EVEN PATH to FIXED REGULAR PATH(R), where $R = (00)^*$, as follows. Given an instance G, x, y of EVEN PATH, construct a db-graph H isomorphic to G, except that every edge in H is labelled with 0. There is an even simple path from x to y in G if and only if there is a simple path from x to y in H which satisfies R. It is easy to see that FIXED REGULAR PATH(R) is in NP; we conclude that FIXED REGULAR PATH(R), where $R = (00)^*$, is NP-complete.

(2) The fact that DISJOINT PATHS is NP-complete follows immediately from results in [11]. We reduce DISJOINT PATHS to FIXED REGULAR PATH(R), where $R = 0^*10^*$. Given an instance G, w, x, y, z of DISJOINT PATHS, construct a db-graph H isomorphic to G, except that every edge of H is labelled with 0. Now add a new edge (x, y) labelled 1 to H. There is a simple path from w to z satisfying R in H if and only if there are disjoint simple paths from w to x and from y to z in G. We conclude that FIXED REGULAR PATH(R), where $R = 0^*10^*$, is also NP-complete. ∎

COROLLARY 1. *REGULAR SIMPLE PATH is NP-complete.*

Proof. NP-hardness follows from Theorem 1. To show that REGULAR SIMPLE PATH is in NP, we observe that, for an arbitrary regular expression R, given a simple path from x to y in G with path label w, we can check in polynomial time in the lengths of R and w whether or not w is in $L(R)$ [2]. ∎

It is interesting to note that if G is *undirected*, then both EVEN PATH and DISJOINT PATHS can be solved in polynomial time. EVEN PATH can be solved in polynomial time by using matching techniques [17], while a polynomial time algorithm for DISJOINT PATHS is given in [20].

Each of the two NP-completeness results of Theorem 1 can be generalized. We first generalize from the regular expression $(00)^*$ to expressions of the form w^*, for any $w \in \Sigma^*$ such that $|w| \geq 2$. For this we use the following NP-complete problem from [17], which was used there to show the NP-completeness of EVEN PATH.

PATH VIA A NODE
Instance: Directed graph $G = (N, E)$, and nodes $x, y, m \in N$.
Question: Is there a directed simple path from x to y via m?

THEOREM 2. FIXED REGULAR PATH(R), *in which $R = w^*$, for any $w \in \Sigma^*$ such that $|w| \geq 2$, is NP-complete.*

Proof. Once again, membership in NP is easy to demonstrate. We reduce PATH VIA A NODE to FIXED REGULAR PATH(R) using a variation of the construction from [17]. Given an instance G, x, y, m of PATH VIA A NODE, construct a db-graph $H = (N', E')$ as follows:

$$
\begin{aligned}
N' &= \big((N - \{m\}) \times \{1, 2\}\big) \cup \{m\}, \\
E' &= \{\big((u,1),(u,2)\big) \mid u \in N - \{m\}\} \cup && (1)\\
&\quad \{\big((u,2),(v,1)\big) \mid (u,v) \in E\} \cup && (2)\\
&\quad \{\big((u,2),m\big) \mid (u,m) \in E\} \cup && (3)\\
&\quad \{\big(m,(u,1)\big) \mid (m,u) \in E\} && (4)
\end{aligned}
$$

The proof now divides into two parts, depending on whether w is of even or odd length. Rather than introducing additional nodes into the above structure, which we believe would obscure the proof, below we allow edges to be labelled with strings of symbols. The length of a path is the length of its concatenated edge labels.

Assume that $w = w_1 w_2$, where $|w_1| = n$ and $|w_2| = n$, $n \geq 1$. There are two copies of each edge of types 1 and 2 above, one copy labelled with w_1, the other with w_2. Edges of type 3 are labelled with w_2, while edges of type 4 are labelled with w_1. We claim that there is a simple path from x to y through m in G if and only if there is a simple path from $(x, 1)$ to $(y, 2)$ satisfying R in H.

If there is a path p from x to y through m in G, then let p_1 be the subpath of p from x to m, and p_2 be the subpath of p from m to y. Let u be the predecessor of m on p_1 and v be the successor of m on p_2. Then in H we can traverse a simple path from $(x, 1)$ to $(u, 2)$ which satisfies $(w_1 w_2)^* w_1$, followed by the edges labelled w_2 and w_1 from $(u, 2)$ to m and from m to $(v, 1)$, respectively, followed by a simple path from $(v, 1)$ to $(y, 2)$ which satisfies $(w_2 w_1)^* w_2$. The overall path thus satisfies $(w_1 w_2)^*$ and is guaranteed to be simple.

Now assume there is a simple path p from $(x, 1)$ to $(y, 2)$ in H which satisfies R. All strings in $L(R)$ are of length mn, where m is even. Any path from $(x, 1)$ to $(y, 2)$ which does not pass through m must be of length kn, where k is odd. We conclude that p must pass through m in H; hence, there is a simple path from x to y via m in G.

We now consider the case in which $|w| = 2n + 1$, $n \geq 1$. Let $w = a_0 w_1 = w_2 a_{2n}$, where $|w_1| = |w_2| = 2n$, $n \geq 1$. One copy of each edge of type 1 in H is labelled with a_0, the other with a_{2n}. One copy of each edge of type 2 is labelled with w_1, the other with w_2. Type 3 edges are labelled with w_1, while type 4 edges are labelled with w_2.

It is easy to see that if there is a simple path from x to y via m in G, there must be a simple path satisfying R in H. For the other direction, it suffices to note that simple paths in H from $(x, 1)$ to $(y, 2)$ which do not pass through m have length $m(2n + 1) + 1$, $m \geq 0$, while those which do pass through m have length $k(2n + 1)$, $k \geq 2$, which are also the lengths of strings in $L(R)$. These two can never be equal for $n \geq 0$. We conclude that if there is a simple path from $(x, 1)$ to $(y, 2)$ in H satisfying R, there must be a simple path from x to y via m in G. ∎

We now generalize the NP-completeness result for FIXED REGULAR PATH(R) where $R = 0^*10^*$. If $S \subseteq \Sigma$, let S also denote the alternation of its elements.

THEOREM 3. *Let R be a regular expression of the form S^*wT^*, where S and T are subsets of Σ and $w \in \Sigma^+$. In addition, assume that either (1) some a in w appears in neither S nor T, or (2) there are symbols $b \in S$ and $c \in T$ such that neither appears in w. Then* FIXED REGULAR PATH(R) *is NP-complete.*

Proof. Once again, FIXED REGULAR PATH(R) is obviously in NP. We use essentially the same reduction from DISJOINT PATHS to FIXED REGULAR PATH(R) as in Theorem 1 for this more general case.

Given an instance G, w, x, y, z of DISJOINT PATHS, construct a db-graph H isomorphic to G, except that two copies of each edge of H are made, one labelled with $b \in S$, and one labelled with $c \in T$. For case (2), b and c are those symbols mentioned in the statement of the theorem; for case (1), we choose $b \neq a$ and $c \neq a$. Assume that $w = a_1a_2 \cdots a_n$. Now add $n-1$ nodes $v_1, v_2, \ldots, v_{n-1}$ to H, along with the path $p_w = (x, e_1, v_1, \ldots, e_{n-1}, v_{n-1}, e_n, y)$, where e_i is labelled with a_i, $1 \leq i \leq n$.

If there are disjoint simple paths from w to x and from y to z in G, it is easy to see that there must be a simple path from w to z satisfying R in H. Assume now that there is a simple path p from w to z satisfying R in H. Then p must be of the form $p_1p_wp_2$, since, in both cases (1) and (2), p_w contains an edge label which appears nowhere else in H and has to appear on any path in H satisfying R. We conclude that there must be disjoint simple paths from w to x and from y to z in H, and hence in G. ∎

Theorems 2 and 3 are rather negative results, since they imply 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 Theorems 2 and 3, we certainly would not expect an evaluation algorithm to run in polynomial time. One such example is the "Air Canada" query used in Example 2 (as long as the alphabet Σ contains at least two symbols). These results, however, are not a function 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. In this section, we characterize a class of queries about regular simple paths which can be evaluated in polynomial time. We first introduce some terminology and definitions.

DEFINITION 4. A *nondeterministic finite automaton* (NDFA) M is a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is the *input alphabet*, δ is the *state transition function* which maps $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of S, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is the set of *final states*. The *extended* transition function δ^* is defined as follows. For $s, t \in S$, $a \in \Sigma$, and $w \in \Sigma^*$

$$\delta^*(s, \epsilon) = \{s\}, \text{ and}$$
$$\delta^*(s, wa) = \bigcup_{t \in \delta^*(s,w)} \delta(t, a)$$

The NDFA M *accepts* $w \in \Sigma^*$ if $\delta^*(s_0, w) \cap F \neq \emptyset$. The language $L(M)$ *accepted* by M is the set of all strings accepted by M. A *deterministic finite automaton* (DFA) is an NDFA in which the state transition function is a mapping from $S \times \Sigma$ to S.

DEFINITION 5. Let $M = (S, \Sigma, \delta, s_0, F)$ be an NDFA. The *transition graph* associated with M is a directed, labelled graph $(S, E_M, \psi_M, \Sigma, \lambda_M)$. If $t \in \delta(s, a)$ for $s, t \in S$ and $a \in \Sigma$, then there is an edge e in E_M with $\psi_M(e) = (s, t)$ and $\lambda_M(e) = a$. By confusing representations, we will sometimes say there is a *transition* from state s to state t in M (or t is a *successor* of S) if $t \in \delta(s, a)$, and there is a *path* from s to t if $t \in \delta^*(s, w)$ for some

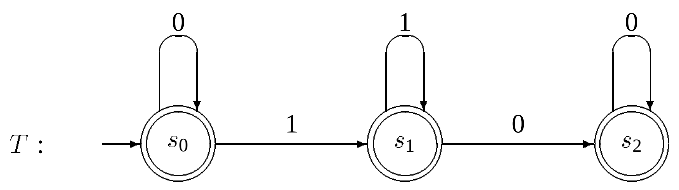


FIG. 3. *Transition graph T of a DFA.*

$w \in \Sigma^*$. Again, similar definitions apply for a DFA.

DEFINITION 6. 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. Then, for state s and set of states T, we can define the *language from s to T*, denoted by L_{sT}, as $\bigcup_{t \in T} L_{st}$. 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 Σ, an ϵ-free NDFA $M = (S, \Sigma, \delta, s_0, F)$ which accepts $L(R)$ can be constructed in polynomial time [2]. From now on, we will assume that all NDFAs are ϵ-free.

Example 3. Figure 3 shows the transition graph T of a DFA M. State s_0 is the initial state of M, while all states are final (denoted by a double circle). (We do not show (reject) states which are not on some path from the initial state to a final state.) $L(M)$ is denoted by the regular expression $0^*1^*0^*$. The suffix language of state s_1 is $[s_1] = 1^*0^*$, while $[s_2] = 0^*$. ∎

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 7. 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 which 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 which 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 $x, y \in N$, and regular expression R over Σ.
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 1. *REGULAR PATH can be decided in polynomial time.*

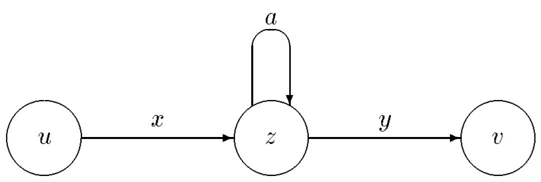


FIG. 4. *A graph containing a non-simple path.*

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 [14]. ∎

In [22], Tarjan provides a polynomial-time algorithm for constructing a regular expression which represents the set of all paths between two nodes of a given graph. As an alternative to the above procedure, one could decide in polynomial time whether there was a path between x and y in G satisfying R by first using Tarjan's algorithm to construct a regular expression R_{xy} representing all paths between x and y in G, and then determining whether the intersection of $L(R)$ and $L(R_{xy})$ was nonempty using NDFAs. The results of the previous section show that it is unlikely that a polynomial-time analogue of Tarjan's algorithm exists for describing the set of all *simple* paths between two nodes.

DEFINITION 8. Let G be a db-graph, $M = (S, \Sigma, \delta, s_0, F)$ a DFA or NDFA, and I the intersection graph of G and M. We call a node (x, s_0) in I an *initial* node, and a node (y, s_f), $s_f \in F$, a *final* node.

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

Proof. Follows immediately from Lemma 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 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 4). There is a non-simple path from u to v in G which satisfies R but no simple path from u to v satisfying R.

DEFINITION 9. An *abbreviation* of a string w is any string which can be obtained from w by removing one or more symbols of w [6].

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 10. For $a \in \Sigma$, denote the regular expression $(a + \epsilon)$ by $(a?)$ (as is done in the *grep* utility of Unix[1], for example). Given a regular expression R, let R' be the 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.

Note that the above definition of restricted regular expressions is semantic rather than syntactic. This has two significant consequences: on the one hand, we are able to prove an equivalence theorem below (Theorem 4) relating restricted regular expressions to languages and automata; on the other, the recognition problem for restricted regular expressions becomes difficult (Corollary 3).

Example 4. The regular expression $0^*1^*0^*$ is restricted: it is equivalent to $(0?)^*(1?)^*(0?)^*$. Recall, from Theorem 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 $0^*(1+\epsilon)0^*$, which is equivalent to $(0?)^*(1?)(0?)^*$. ∎

DEFINITION 11. 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, $L_{sF} \supseteq L_{tF}$).

The following result, although not used elsewhere, provides some interesting restrictions on the structure of DFAs that exhibit the Containment Property.

PROPOSITION 1. *Let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA. If M exhibits the Containment Property, then*

1. *every state in M, which is on a path from s_0 to a state in F, is final,*
2. *the minimum DFA for M exhibits the Containment Property, and*
3. *if M is minimum, then every cycle in M is a loop.*

Proof. (1) Every final state in M accepts ϵ. By the transitivity of "$\supseteq$", every state which is on a path from s_0 to a state in F must also accept ϵ, and hence must be final.

(2) Let $M' = (S', \Sigma, \delta', s_0, F')$ be the minimum DFA equivalent to M. Each state in M' represents a set of equivalent states in M. Assume that $s \in S'$ represents $\{s_1, \ldots, s_k\}$, where $s_i \in S$, $1 \le i \le k$, and that $t \in S'$ represents $\{t_1, \ldots, t_m\}$, where $t_j \in S$, $1 \le j \le m$. There is a transition $\delta'(s, a) = t$ in M' only if, for each s_i, $1 \le i \le k$, in M, there is a transition $\delta(s_i, a) = t_{j_i}$, for some $1 \le j_i \le m$. In M, $[s_i] \supseteq [t_{j_i}]$, $1 \le j_i \le m$, $1 \le i \le k$. Since M' is equivalent to M, $[s] = [s_i]$, $1 \le i \le k$, and $[t] = [t_{j_i}]$, $1 \le j_i \le m$, $1 \le i \le k$. We conclude that $[s] \supseteq [t]$.

(3) Consider a cycle in M which is not a loop, and let s and t be two states on the cycle. Since $[u] \supseteq [v]$ for every pair of consecutive states on the cycle, we conclude from the transitivity of "$\supseteq$" that $[s] \supseteq [t]$ and that $[t] \supseteq [s]$. But then $s \equiv t$, and so M is not minimum, a contradiction. ∎

Example 5. Consider the regular expression $R = 0^*1^*0^*$, and the DFA M accepting $L(R)$ whose transition graph T is given in Figure 3. We can verify that M exhibits the Containment Property by noting that $[s_2]$ is denoted by 0^*, $[s_1]$ by 1^*0^*, and $[s_0]$ by $0^*1^*0^*$. Obviously, $[s_0] \supseteq [s_0]$, $[s_1] \supseteq [s_1]$, and $[s_2] \supseteq [s_2]$. It is easy to check that $[s_1] \supseteq [s_2]$ and $[s_0] \supseteq [s_1]$. Note also that, by Proposition 1, each state is final and, since M is minimal, every cycle in M is a loop. The fact that M exhibits the Containment Property and R is restricted is no coincidence, as we demonstrate below. ∎

THEOREM 4. *Let R be a regular expression over Σ, 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.*

[1] Unix is a trademark of AT&T.

Proof. In our proof, we will use the NDFA $M_R = (T, \Sigma, \mu, t_0, E)$ constructed from regular expression R (such that $L(M_R) = L(R)$) as detailed in [2], and in which ϵ-transitions are usually present. There is a one-to-one correspondence between non-ϵ-transitions in M_R and occurrences of symbols in R, so that it makes sense to refer to *the* transition in M_R corresponding to an occurrence of symbol a in R, and vice versa. Furthermore, replacing an occurrence of a in R by $(a?)$ is equivalent to including an ϵ-transition from the source state to the target state of the transition in M_R corresponding to the occurrence of a.

(1) $\Rightarrow$ (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 $xy \notin L(R)$. Now consider M_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 ϵ-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) $\Rightarrow$ (3) We prove the contrapositive. 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) $\Rightarrow$ (1) Once again we prove the contrapositive. Assume that R is not restricted. Then there is an a-transition in M_R from s to t for which adding an ϵ-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 $r \in \mu^*(t_0, x)$; hence, $xy \notin L(M_R)$. 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 $L(M) = L(M_R)$. Hence, $[p] \not\supseteq [q]$, so M does not exhibit the Containment Property. ∎

THEOREM 5. *REGULAR SIMPLE PATH can be decided in polynomial time for restricted regular expressions.*

Proof. Let the db-graph G and the regular expression R, where R is restricted, constitute an instance of REGULAR SIMPLE PATH. By Lemma 1, it is sufficient to show that whenever there is a path from x to y in G which satisfies R, there is a simple path from x to y satisfying R. Assume that $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$ is a non-simple path from $x = v_1$ to $y = v_n$ in G. Since p is non-simple, $v_i = v_j$, for some $1 \leq i, j \leq n$. Assume that $i < j$, that is, $p = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, e_j, \ldots, v_n)$, and let $p' = (v_1, \ldots, e_{i-1}, v_i, e_j, \ldots, v_n)$. Since p satisfies R, $\lambda(p) \in L(R)$. The path label $\lambda(p')$ is an abbreviation of $\lambda(p)$. By Theorem 4, $L(R)$ is closed under abbreviations; hence, $\lambda(p') \in L(R)$ and p' satisfies R. Removing all such cycles from p will leave a simple path from x to y which satisfies R. ∎

Thus the class of restricted regular expressions is one for which query evaluation can be performed efficiently. We now show that, even though the classes of restricted regular expressions and regular languages closed under abbreviations are subclasses of their regular counterparts, at least they are closed under the regular operators.

THEOREM 6. *Let Σ be an alphabet. The class of regular languages over Σ which is closed under abbreviations is also closed under alternation, concatenation and closure.*

Proof. Let L_1 and L_2 be regular languages closed under abbreviations. It is immediate that $L_1 + L_2$ is closed under abbreviations too. Now let $L = L_1 L_2$ and consider $w = w_1 w_2 \in L$ such that $w_1 \in L_1$ and $w_2 \in L_2$. Let $w' = w_1' w_2'$ be an abbreviation of w. Clearly, string w_i' is an abbreviation of w_i, $i = 1, 2$, and since L_1 and L_2 are closed under abbreviations,

Algorithm S: Compute the suffix language containment relation for a DFA.

```
INPUT:
    DFA $M = (S, \Sigma, \delta, s_0, F)$

OUTPUT:
    For each pair $s, t \in S$, whether $[s] \supseteq [t]$ or not.

METHOD:
1.      for $s \in S - F$ and $t \in F$ do mark $(s, t)$ od
2.      for each ordered pair of distinct states $(s, t) \in ((S \times S) - ((S - F) \times F))$ do
3.          if for some $a \in \Sigma$ $(\delta(s, a), \delta(t, a))$ is marked then
4.              mark $(s, t)$
5.              recursively mark all unmarked pairs on the list for $(s, t)$ and
                on the lists of other pairs that are marked at this step
            else /* no pair $(\delta(s, a), \delta(t, a))$ is marked */
6.              for all $a \in \Sigma$ do
7.                  put $(s, t)$ on the list for $(\delta(s, a), \delta(t, a))$ unless $\delta(s, a) = \delta(t, a)$
                od
            fi
        od
```

FIG. 5. *Computing the suffix language containment relation for DFA $M = (S, \Sigma, \delta, s_0, F)$.*

$w_1' \in L_1$ and $w_2' \in L_2$. Hence, $w_1' w_2' = w'$ is in L, allowing us to conclude that L is closed under abbreviations.

Let L be a regular language closed under abbreviations. Since $\epsilon \in L$ and regular languages closed under abbreviations are also closed under concatenation, L^* must be closed under abbreviations. ∎

COROLLARY 2. *The class of restricted regular expressions over Σ is closed under alternation, concatenation and closure.*

Example 6. One of the simplest restricted regular expressions is 0^*. Since the class of restricted regular expressions is closed under alternation, concatenation and closure, $0^* + 1^*$ and $0^*1^*0^*$ (which we have already seen) are restricted. On the other hand, restricted expressions can also sometimes be built from expressions which are not restricted; examples include $0^* + 1^*$, $0^*10^* + 0^*$ (which we have already seen), $(00)^* + 0^*$, and $((0^*1)^* + 0^*)^*$. ∎

Given a query Q_R, we would like to test whether R is restricted in order to know that it is safe to use a polynomial time evaluation algorithm. By adapting an algorithm to minimize the number of states of a DFA [13], we can compute the suffix language containment relation for all pairs of states in a DFA M. The suffix language containment relation will be used in subsequent sections; it also provides an obvious method for testing whether or not a regular expression R is restricted (using Theorem 4). The algorithm for computing the suffix language containment relation, Algorithm S, is shown in Figure 5. Lines 3 to 7 of Algorithm S are taken directly from the algorithm in [13]. That algorithm marks pairs of *inequivalent* states, so it considers unordered pairs of states. Lines 1 and 2 of our algorithm are altered appropriately in order to consider ordered pairs of states. If (s, t) is marked by Algorithm S, then $[s] \not\supseteq [t]$.

If M has n states, then Algorithm S runs in $O(n^2)$ time (assuming a constant alphabet) [13]. (An alternative, almost linear-time algorithm is given in [2].) Since the construction of a DFA M accepting $L(R)$ may take exponential time (in the size of R), using Algorithm S to test whether a regular expression is restricted 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, it turns out that determining whether or not R is restricted is a hard problem. Consider the following result.

PROPOSITION 2 ([21]). *Determining whether a regular expression over alphabet* $\{0\}$ *does not denote* 0^* *is NP-complete.*

We will use this result to show that the problem of deciding whether a regular expression over alphabet Σ is not restricted is NP-hard. To do so, we first prove the following.

THEOREM 7. *Let R be a starred regular expression over alphabet* $\{0\}$. *Deciding whether R is not restricted is NP-complete.*

Proof. We first show that the problem is in NP. If R is not restricted, then $L(R)$ is not closed under abbreviations (Theorem 4). Thus, there is a string in 0^* that is not in $L(R)$. If $L(R) \neq 0^*$, then, by considering a DFA accepting $L(R)$, it can be seen that there must be an $n \leq 2^{|R|}$ such that $0^n \notin L(R)$. A nondeterministic polynomial time algorithm can verify that R is not restricted by first guessing the binary representation of n, and then testing whether there is a path in the transition graph of an NDFA accepting $L(R)$ of length n to a final state. The latter step can be done deterministically in time polynomial in the length of R [21].

We reduce the problem of Proposition 2 to the present problem by showing that R is not restricted if and only if R does not denote 0^*. We have already shown that if R is not restricted, then $L(R) \neq 0^*$. Conversely, assume that R does not denote 0^*. Let x be the shortest string in 0^* that is not in $L(R)$. Since R is starred, $L(R)$ is infinite, so there is a string $xy \in L(R)$ for which $y \neq \epsilon$. But x is an abbreviation of xy; hence, by Theorem 4, R is not restricted. ∎

COROLLARY 3. *Deciding whether a regular expression over alphabet* Σ *is not restricted is NP-hard.*

4. Constrained Cycles in Db-Graphs.

In some instances, knowledge about the cyclic structure of a db-graph G allows us to determine (without consulting G itself) that a particular query Q_R can be evaluated in polynomial time on G. We have already shown that, in the extreme case when G is acyclic, Q_R is always evaluable in polynomial time. Let us assume that we know that the cyclic structure of G is constrained by a regular expression C; that is, every cycle label in G is in $L(C)$.

DEFINITION 12. Let C be a regular expression over Σ, and $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph. Let Y be the set of cycle labels in G, namely

$$Y = \{\lambda(c) \mid c \text{ is a cycle in } G\}.$$

We say that G *complies with* C if $Y \subset L(C)$. The regular expression C is called a *cycle constraint*.

Each cycle constraint C defines a class of db-graphs whose cyclic structure satisfies C. For example, in this way we can define the classes of bipartite graphs, loop-free graphs, and acyclic graphs by specifying the regular expressions $(\_\ \_)^+$, $\_\ \_(\_^*)$, and , respectively[2]. The class of db-graphs with unconstrained cycles is defined by the expression $\_^+$, which denotes Σ^+.

Before continuing, we need to introduce some terminology regarding properties of the intersection graph of a db-graph and a transition graph.

DEFINITION 13. Let I be the intersection graph of db-graph $G = (N, E, \psi, \Sigma, \lambda)$ and transition graph T of NDFA $M = (S, \Sigma, \delta, s_0, F)$. We say that a path $p = ((v_1, s_1), \ldots, (v_n, s_n))$,

[2] Recall that if $\Sigma = \{a_1, \ldots, a_n\}$, then _ (underscore) is shorthand for $a_1 + \cdots + a_n$.

where $v_i \in N$ and $s_i \in S$, in I is *db-simple* if $v_i \neq v_j$, $1 \leq i, j \leq n$. In other words, p is db-simple if and only if $(v_1, \ldots, v_n)$ is a simple path in G. In addition, we call I *simplicial* if whenever there is a path $p = ((v_1, s_1), \ldots, (v_n, s_n))$, where $v_1 \neq v_n$ and $s_n \in F$, there is a db-simple path from (v_1, s_1) to (v_n, s_n'), $s_n' \in F$, in which the first components of nodes form a subset of the first components of nodes on p.

From the above definition and Lemma 1, it is clear that if the intersection graph I of a db-graph and the transition graph corresponding to a regular expression R is simplicial, then Q_R can be evaluated in polynomial time in the size of I. The following theorem characterizes simplicial intersection graphs in the presence of cycle constraints.

THEOREM 8. *Let C be a cycle constraint. For query Q_R, let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$ and T be the transition graph of M. For every db-graph G complying with C, the intersection graph I of G and T is simplicial if and only if whenever there is a path from a reachable state s to t in T satisfying C, $[s] \supseteq [t]$.*

Proof. (If) Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph complying with C and

$$p = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, \ldots, v_n)$$

be a non-simple path satisfying R in G. Hence, there is a path q from (v_1, s_0) to (v_n, s_f), $s_f \in F$, in I. For notational simplicity, let $w_1 = \lambda(e_1) \cdots \lambda(e_{i-1})$, $w_2 = \lambda(e_i) \cdots \lambda(e_{j-1})$, and $w_3 = \lambda(e_j) \cdots \lambda(e_{n-1})$. So $w_1 w_2 w_3 \in L(R)$. Since G complies with C, $w_2 \in L(C)$. Assume that $\delta^*(s_0, w_1) = s$ and $\delta^*(s, w_2) = t$. So there is a path in T from s to t satisfying C and a path from (v_i, s) to (v_i, t) in I; hence, by assumption, $[s] \supseteq [t]$. The string w_3 is in $[t]$ because p satisfies R, so $w_3 \in [s]$ as well. It follows that $w_1 w_3 \in L(R)$ and therefore that

$$p' = (v_1, \ldots, e_{i-1}, v_i, e_{j+1}, \ldots, v_n)$$

satisfies R. This process can be repeated to obtain a db-simple path q' from (v_1, s_0) to (v_n, s_f'), $s_f' \in F$, such that the first components of q' form a subset of the first components of q. We conclude that I is simplicial.

(Only if) Assume that there is a path p from s to t in T which satisfies C but for which $[s] \not\supseteq [t]$. The constraint C cannot be , for otherwise p would not satisfy C. Since s is reachable in T, there is a string w_1 such that $\delta^*(s_0, w_1) = s$. Furthermore, $[t]$ cannot be , for otherwise $[s] \supseteq [t]$. So let w_3 be a string in $[t]$ but not in $[s]$, and w_2 be the path label of p. The string w_2 cannot be ϵ since p must be of length greater than zero. We can construct a db-graph $G = (N, E, \psi, \Sigma, \lambda)$ comprising a single non-simple path

$$q = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, \ldots, v_n)$$

such that $\lambda(e_1) \cdots \lambda(e_{i-1}) = w_1$, $\lambda(e_i) \cdots \lambda(e_{j-1}) = w_2$, and $\lambda(e_j) \cdots \lambda(e_{n-1}) = w_3$. G complies with C since the only cycle in G is labelled with w_2 which is in $L(C)$. The path q satisfies R because $w_1 w_2 w_3 \in L(R)$. Hence, there is a path from (v_1, s_0) to (v_n, s_f), $s_f \in F$, in I. However, the path

$$q' = (v_1, \ldots, e_{i-1}, v_i, e_{j+1}, \ldots, v_n)$$

does not satisfy R since $w_1 w_3 \notin L(R)$ (otherwise w_3 would be in $[s]$). Consequently, there is no db-simple path from (v_1, s_0) to (v_n, s_f'), $s_f' \in F$, in I, and we conclude that I is not simplicial. ∎

The above result does not depend on the particular DFA accepting $L(R)$. Consider two DFAs, $M_1 = (S_1, \Sigma, \delta_1, s_0, F_1)$ and $M_2 = (S_2, \Sigma, \delta_2, t_0, F_2)$, accepting $L(R)$, and let $s \in S_1$ and $t \in S_2$ be a pair of states such that there is a string x for which $\delta_1^*(s_0, x) = s$ and

$$\begin{tikzcd}
R \text{ restricted} \arrow[d, Rightarrow] & & G \text{ acyclic} \arrow[d, Rightarrow] \\
\text{ } & G \text{ complies with a cycle constraint compatible with } R & \text{ }
\end{tikzcd}$$

FIG. 6. *Relationship between regular expression R and db-graph G for query Q_R.*

$\delta_2^*(t_0, x) = t$. Because $L(M_1) = L(M_2)$, it must be the case that $[s] = [t]$ (that is, $s \equiv t$). In other words, the fact that Theorem 8 is true independent of the particular DFA chosen is a consequence of the Myhill-Nerode theorem, which states that a language is accepted by a DFA if and only if it is the union of some of the equivalence classes of a right-invariant equivalence relation of finite index [13]. This leads us to the following definition.

DEFINITION 14. Let R be a regular expression and T be the transition graph for a DFA accepting $L(R)$. We say that R is *compatible with* cycle constraint C if whenever there is a path from (a reachable state) s to t in T satisfying C, $[s] \supseteq [t]$.

Theorem 8 generalizes our previous results. For the case when G is acyclic, $C =$, and no path in T satisfies C so the result holds vacuously. In other words, every regular expression is compatible with . When the cyclic structure of G is unconstrained, C denotes Σ^+, and every path in T satisfies C, so $[s]$ must contain $[t]$ for all pairs of reachable states in T. This corresponds to the case of restricted regular expressions; that is, a regular expression R is compatible with C (where C denotes Σ^+) if and only if R is restricted. The relationship among these properties is shown in Figure 6.

By appealing once again to the result of Lemma 1, we obtain the following corollary to Theorem 8.

COROLLARY 4. *Let C be a cycle constraint and G be a db-graph that complies with C. A query Q_R on G can be evaluated in polynomial time in the size of both R and G if R is compatible with C.*

A simple algorithm for testing whether a regular expression is compatible with a cycle constraint is given in Figure 7. Because it constructs DFAs from regular expressions R and C, the algorithm can take exponential time in the length of R and C. However, deciding whether R and C are compatible is NP-hard, since deciding whether R is restricted is a special case of testing compatibility.

THEOREM 9. *Given a regular expression R and a cycle constraint C, deciding whether R and C are compatible is NP-hard.*

Example 7. Let $R = (00)^*$. A DFA M_R accepting $L(R)$ is shown in Figure 8(a). Because $[a] \not\supseteq [b]$, we know that R is not restricted. In fact, we saw in Theorem 1 that deciding if $(x, y) \in Q_R(G)$ is NP-complete for db-graphs in general. However, Q_R can be evaluated in polynomial time on bipartite graphs. As we have already seen, the regular expression $C = (\_\ \_)^+$ defines the class of bipartite graphs. A DFA M_C accepting $L(C)$ is shown in Figure 8(b), while the intersection graph I of M_R and M_C is given in Figure 9. The only paths in I satisfying C which start from a node containing the initial state of M_C and end at a node containing a final state of M_C are from (a, A) to (a, D) and from (b, A) to (b, D). Since $[a] \supseteq [a]$ and $[b] \supseteq [b]$, Corollary 4 tells us that Q_R can be evaluated in polynomial time on any bipartite graph. ∎

Given a query Q_R and a db-graph G, if we know that G complies with cycle constraint C, we can test whether R is compatible with C using the above algorithm. If so, we can use a polynomial time algorithm to evaluate Q_R on G. On the other hand, if we do not know about the cyclic structure of G, it seems that we might have to resort to an exponential

Algorithm: Testing whether a regular expression is compatible with a cycle constraint.

INPUT:
Regular expression R and cycle constraint C.

OUTPUT:
Whether or not R is compatible with C.

METHOD:

1. Construct DFAs $M_R = (S_1, \Sigma, \delta_1, i_1, F_1)$ accepting $L(R)$ and $M_C = (S_2, \Sigma, \delta_2, i_2, F_2)$ accepting $L(C)$.
2. Compute the suffix containment relation for M_R (Algorithm S in §3).
3. Construct the intersection graph I of $M_R \times M_C$.
4. Compute the transitive closure I^+ of I.
5. If $[s] \supseteq [t]$ for each edge $((s, i_2), (t, f))$ in I^+, where $f \in F_2$, answer "yes"; otherwise answer "no".

FIG. 7. *Testing whether a regular expression is compatible with a cycle constraint.*

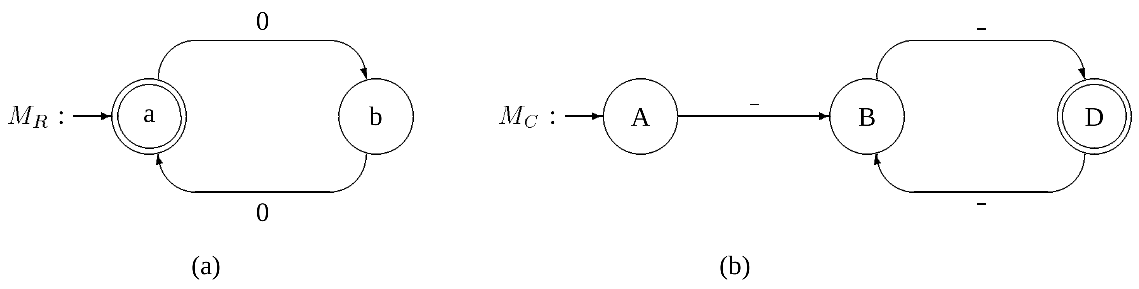


FIG. 8. *DFAs (a) M_R for $R = (00)^*$ and (b) M_C for $C = (\_\,\_)^+$.*

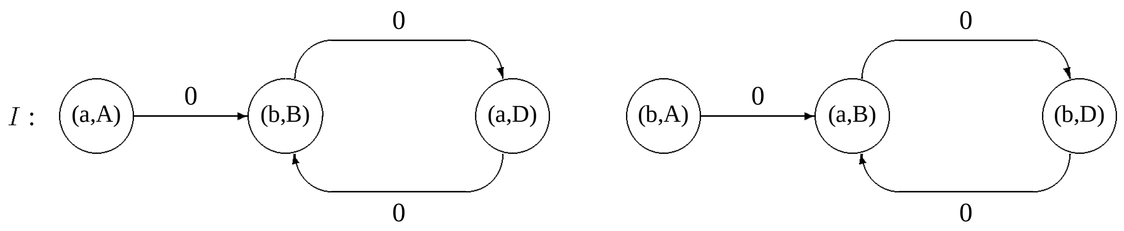


FIG. 9. *The intersection graph I of M_R and M_C (Figure 8).*

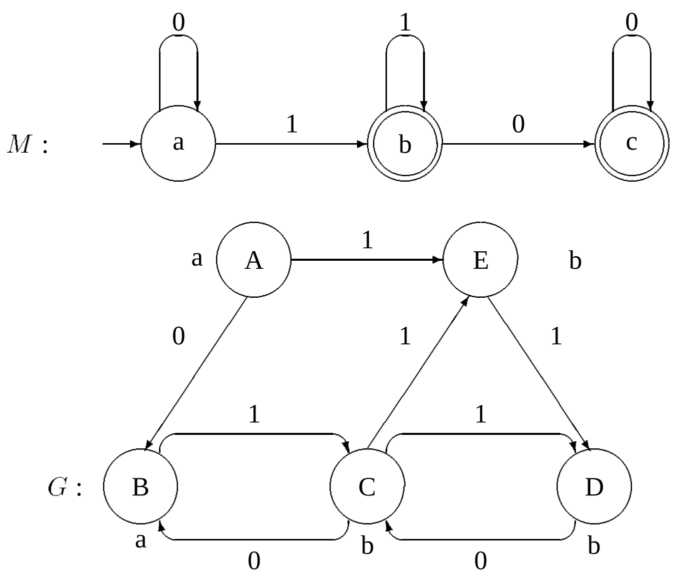


FIG. 10. *A DFA M and db-graph G.*

time algorithm if R is not restricted. In the next section, however, we describe an evaluation algorithm which runs in polynomial time in the size of G if G happens to comply with a cyclic constraint with which R is compatible.

5. 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 §2, the algorithm does not run in polynomial time in general. It does, however, run in polynomial time under the sufficient conditions identified in §3 and §4, namely, when G is acyclic, R is restricted, or G complies with a cycle constraint compatible with R. In fact, we show that the algorithm runs in polynomial time if G and R are conflict-free, a condition implied by those above.

The evaluation algorithm traverses 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, requiring answer nodes to be connected by simple paths in G and retaining state markings can lead to incompleteness in query evaluation.

Example 8. Consider the query Q_R, where $R = 0^*1^+0^*$. An automaton M accepting $L(R)$ and a db-graph G are shown in Figure 10. Note the similarity between M and the automaton of Figure 3 in §3. 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 a, a, b and b, respectively, and the answers (A, C) and (A, D) are found, since b is a final state. We cannot mark C with state c because (A, B, C, D, C) is a non-simple path. If we now backtrack to node C, we can mark E with b, resulting in the answer (A, E) being found. Node D is still marked with b (as shown in Figure 10), so we backtrack to C. However, once again we cannot mark B with state c because (A, B, C, B) is a non-simple path. So we backtrack to A, and find that E is

$$\begin{tikzcd}
R \text{ restricted} \arrow[dr, Rightarrow] & & G \text{ acyclic} \arrow[dl, Rightarrow] \\
& G \text{ complies with a cycle constraint compatible with } R \arrow[d, Rightarrow] & \\
& G \text{ and } R \text{ are conflict-free} &
\end{tikzcd}$$

FIG. 11. *Relationship between regular expression R and db-graph G for query Q_R.*

already marked with state b. Consequently, the search terminates without the answer (A, B) being found. ∎

It turns out that it is safe to retain markings when G is acyclic or R is restricted. However, because of the structure of a particular db-graph G, it might 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 15. Let I be the intersection graph of a db-graph G and a DFA $M = (S, \Sigma, \delta, s_0, F)$. An *initial path* in I is any path of the form $((v_0, s_0), \ldots, (v_n, s_n))$. The initial path p is *conflict-free* if (1) p is db-simple, or (2) p is $q \cdot (v, s)$, where q is conflict-free and if v appears in q, then for some (v, t) in q, $[t] \supseteq [s]$. If for no (v, t) in q is it the case that $[t] \supseteq [s]$, then there is a *conflict* at v.

If every simple initial path in I is conflict-free, then I is said to be *conflict-free*[3], 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 4, M exhibits the Containment Property; hence, I is conflict-free irrespective of the structure of G. Finally, if G complies with a cycle constraint compatible with R, then, by Theorem 8, G and R are conflict-free. We will show that Q_R can be evaluated in polynomial time if I is conflict-free. Hence, conflict-freedom is another (weaker) sufficient condition for Q_R to be polynomial time evaluable (see Figure 11).

The result of the following lemma is used in our evaluation algorithm.

LEMMA 3. *Let I be the intersection graph of a db-graph G and a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$. An initial path p in I is conflict-free if and only if (1) p is db-simple or (2) p is $q \cdot (v, s)$, where q is conflict-free and if v appears in q, then for the* first *(v, t) in q, $[t] \supseteq [s]$.*

Proof. The "if" direction is trivial. Assume that p is conflict-free but not db-simple. Furthermore, assume that p is $q \cdot (v, s)$, where q is conflict-free and v appears in q. We prove, by induction on the number of occurrences of v in q, that $[t] \supseteq [s]$ where (v, t) is the first occurrence of v in q.

The basis in which v occurs only once in q is trivial. Assume that the inductive hypothesis is true for fewer than n occurrences of v in q, and let p be $q \cdot (v, s)$. Since p is conflict-free, we know from the definition that for some (v, r) in q, $[r] \supseteq [s]$. By the inductive hypothesis, $[t] \supseteq [r]$; hence, $[t] \supseteq [s]$, as required. ∎

Example 9. Consider again the DFA M and the db-graph G of Example 8 shown in Figure 10. The intersection graph I of G and M is shown in Figure 12. Recall that, if markings were retained, the answer (A, B) would not be found. However, there is a conflict in I. This is because there is an initial path in I from (A, a) via (B, a) to (B, c), but $[a] \not\supseteq [c]$.

[3] This is a strictly weaker definition of conflict-freedom than that given in [18].

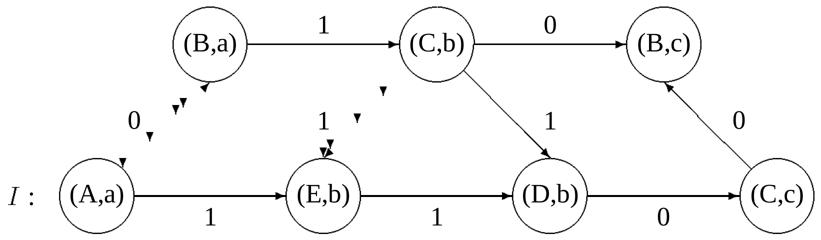


FIG. 12. *The intersection graph I of db-graph G and DFA M of Figure 10.*

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 $\emptyset$.
3. For each node $v \in N$, set $CM[v]$ and $PM[v]$ to $\emptyset$.
4. Test $[s] \supseteq [t]$ for each pair of states $s$ and $t$ in $M$.
5. For each node $v \in N$,
      (a) call SEARCH($v, v, s_0$,conflict) (see Figure 14)
      (b) reset $PM[w]$ to $\emptyset$ for any marked node $w \in N$.
```

FIG. 13. *Evaluation of a query on a db-graph.*

Algorithm C detects such conflicts and unmarks nodes on backtracking, enabling the answer (A, B) to be found. ∎

We now proceed with a description of Algorithm C, shown in Figure 13. The algorithm uses a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$ to control a depth-first search of the db-graph G (Line 1). There are two reasons why a DFA rather than an NDFA is used. The first is to ensure that no conflicts are encountered when R is restricted. The second reason is to avoid detecting unnecessary conflicts in I. In an NDFA, if $[s] \not\supseteq [t]$, it might be the case that there is a state q such that both s and q are in $\delta^*(s_0, w)$, for some $w \in \Sigma^*$, and $[q] \supseteq [t]$. If node v in G is first marked with s, following which a cycle at v satisfying L_{st} is traversed, a conflict would be registered. This is unnecessary since v would subsequently be marked with q, and any simple path from v satisfying $[t]$ would be found because $[q] \supseteq [t]$.

Algorithm C traverses the transition graph of M and the db-graph G simultaneously, in effect performing a depth-first search of the intersection graph I of G and M. We will often refer to trees of the depth-first search forest generated by Algorithm C. Because of Line 5(a), each tree T in the forest is rooted at an initial node of I. When a final node of I is reached, Line 8 adds the appropriate pair of nodes from G to $Q_R(G)$. Lines 9 and 10 force the algorithm to consider only paths in G which satisfy R, that is, paths in I.

While the traversal of I is restricted to simple paths, it is not necessarily restricted to db-simple paths; we will prove below that it is safe to traverse non-db-simple paths in the

```
    procedure SEARCH ($u, v, s$, var conflict)
    /*
        $u$ and $v$ are nodes in the db-graph
        $s$ is a state in the DFA
        db-cycle is a Boolean flag
    */
6.      conflict $\leftarrow$ false
7.      $CM[v] \leftarrow CM[v] \cup \{s\}$
8.      if $s \in F$ then $Q_R(G) \leftarrow Q_R(G) \cup \{(u, v)\}$ fi
9.      for each edge in $G$ from $v$ to $w$ with label $a$ do
10.         if $\delta(s, a) = t$ and $t \notin CM[w]$ and $t \notin PM[w]$ then
11.             if FIRST($CM[w]$) $= q$ and $([q] \not\supseteq [t])$ then
12.                 conflict $\leftarrow$ true
                else /* $CM[w] = \emptyset$ or $[q] \supseteq [t]$ */
13.                 SEARCH ($u, w, t$, new-conflict)
14.                 conflict $\leftarrow$ conflict or new-conflict
                fi
            fi
        od
15.     $CM[v] \leftarrow CM[v] - \{s\}$
16.     if not conflict then $PM[v] \leftarrow PM[v] \cup \{s\}$ fi
    end SEARCH
```

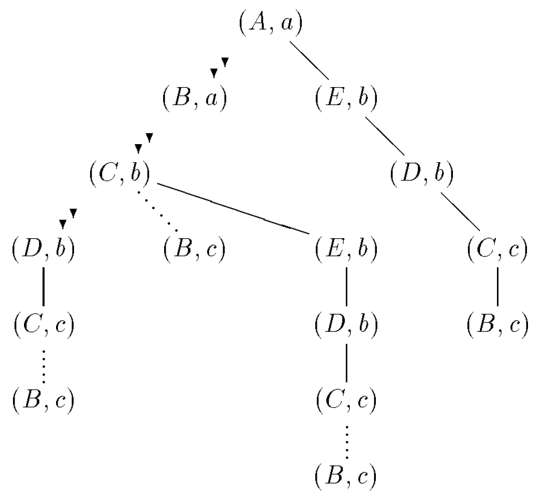
FIG. 14. *Search procedure for query evaluation.*

absence of conflicts. Nodes in G are marked with states of M when they are visited. Two sets of markings are used for each node v: (1) a set of current markings ($CM[v]$) which indicates the states with which v is associated on the current path on the stack of procedure SEARCH (Lines 7 and 15), and (2) a set of previous markings ($PM[v]$) which represents earlier markings of v, excluding the current path (Line 16). Current markings are used to avoid cycles in I and to detect conflicts, while previous markings are used where possible to prevent a node in G from being visited more than once in the same state during a single execution of Line 5(a). The function FIRST applied to marking set $CM[v]$ returns the first state marking for v on the current path, or false if there is no marking.

A node w is visited in state t only if t is not in the previous markings of w and either w is currently unmarked ($CM[w]$ is empty) or the first state marking q for v on the current path is such that $[q] \supseteq [t]$, that is, there is no conflict between q and t at v (Lines 10 to 13). Note that there may in fact be a conflict between t and some later marking of v on the current path, but this does not affect the correctness of the algorithm, as we will demonstrate below.

Lines 6, 11 and 12 implement the conflict detection; that is, *conflict* is true if there is a conflict between states q and t at node w. If *conflict* is set to true at Line 12, then Lines 14, 15 and 16 ensure that the marking of any node which was on the stack at the time the conflict was detected is removed once that node is unstacked. If no conflict occurs on any path rooted at (v, s), then s is added to the previous markings of v in Line 16.

In the proofs that follow, we will often say that (v, s), for example, is on the stack of procedure SEARCH. The variables v and s refer to the middle two parameters of SEARCH and correspond to the node (v, s) in the corresponding intersection graph. The reason for excluding the other two parameters of SEARCH is that u (the first) remains unchanged during an execution of Line 5(a), while we are not always concerned about the value of *conflict*. We



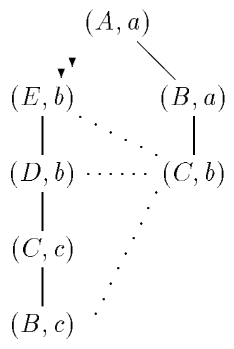


FIG. 15. *Two possible depth-first search trees.*

will also sometimes exclude *conflict* when referring to a particular invocation of SEARCH, for example, SEARCH(u, v, s). Before proving the correctness of Algorithm C, we demonstrate its behaviour by means of an example.

Example 10. Consider again the intersection graph I of Figure 12. Two possible depth-first search trees (DFSTs) traversed by Algorithm C are shown in Figure 15. Note that nodes in a DFST can be repeated because of unmarking; for example, node (D, b) appears three times in Figure 15(a). Dotted edges in the figure lead to nodes for which SEARCH is not called, either because of a conflict (those in (a)), or because the node is already marked via either CM or PM (as in (b)). These latter edges correspond to forward, back and cross edges in a conventional DFST [2].

Assume that Algorithm C starts traversal from node (A, a), that is, SEARCH(A, A, a)) is called at Line 5(a), and that the order of traversal is according to the DFST in Figure 15(a). Since initially nodes B, C and D have no current marking, Line 11 evaluates to false and SEARCH is called successively with (B, a), (C, b) and (D, b). Because b is a final state, (A, C) and (A, D) are added to $Q_R(G)$ by Line 8. Although C already has a current marking (namely b), the fact that $[b] \supseteq [c]$ means that Line 11 again evaluates to false and SEARCH is called with (C, c). Now because the first marking for B is a and $[a] \not\supseteq [c]$, a conflict is registered at Line 12. The algorithm now backtracks, removing current markings (Line 15) and not assigning previous markings (Line 16).

Considering (B, c) from (C, b) again gives rise to a conflict, so the algorithm tries the path via (E, b). Note that (D, b) and (C, c) are no longer marked so they are revisited, once again giving rise to a conflict. By the time the algorithm backtracks to (A, a) all nodes (other than A) are unmarked, so that the db-simple path to (B, c) can finally be found and (A, B) added to $Q_R(G)$.

If the path to (B, c) via (E, b) had been chosen first by Algorithm C (as in Figure 15(b)), then no conflicts would have been detected, resulting in previous markings being kept for B, D and E. On traversing the path to (C, b), Line 10 would ensure that (B, c), (D, b) and (E, b) are not revisited and no conflicts are registered. ∎

LEMMA 4. *If* conflict *is false at Line 16 of SEARCH(u, v, s), then Algorithm C has performed an entire depth-first search of I from node (v, s).*

Proof. The proof proceeds by induction on the length of the longest simple path p from (v, s) in I. If p is of length zero, the result follows trivially. Assume the result holds for nodes in I from which the longest simple path is of length $n-1$, and consider node (v, s) for which the longest simple path in I is of length n.

For *conflict* to be false at Line 16 of SEARCH(u, v, s), it must be that, for each successor (w, t) of (v, s) in I, either (1) $t \in PM[w]$ at Line 10, or (2) *new-conflict* must have been false at Line 14. In case (1), *conflict* must have been false at Line 16 of SEARCH(u, w, t) in order for t to be added to $PM[w]$. In case (2), *conflict* must have been false at Line 16 of SEARCH(u, w, t) so that *new-conflict* is false at Line 14. Since the longest simple path from (w, t) in I must be of length less than or equal to $n-1$, we conclude from the inductive hypothesis that an entire depth-first search from (w, t) has been performed by Algorithm C. Clearly, Lines 9 and 10 consider every successor of (v, s) in I, so the result follows. ∎

DEFINITION 16. A node (v, s) in depth-first search tree T is called a *conflict predecessor* if, for some successor (w, t) of (v, s) in I, w appears in an ancestor of (v, s) in T and, for the first such occurrence (from the root), say (w, q), it is the case that $[q] \not\supseteq [t]$. In other words, there is a conflict between q and t at w.

LEMMA 5. *Consider the execution of SEARCH(u, v, s) in DFS tree T. State s is added to $PM[v]$ in Line 16 if and only if no descendant of (v, s) in T is a conflict predecessor.*

Proof. If s is added to $PM[v]$ in Line 16, then *conflict* must be false. Hence, by Lemma 4, an entire depth-first search of I from (v, s) must have been performed. But a conflict predecessor is a node (w, t) in T which has a successor in I that does not appear as a successor of (w, t) in T. Thus, no conflict predecessor can appear as a descendant of (v, s) in T.

If no descendant of (v, s) in T is a conflict predecessor, then *conflict* is false for all such descendants and hence for (v, s) itself. Thus, s is added to $PM[v]$ in Line 16. ∎

THEOREM 10. *Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph, and R be a regular expression over Σ. Let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$, and I be the intersection graph of G and M. Algorithm C is correct; that is, Algorithm C adds (u, z) to $Q_R(G)$ if and only if there is a db-simple path from (u, s_0) to (z, s_f), $s_f \in F$, in I (that is, there is a simple path from u to z in G satisfying R).*

Proof. Algorithm C clearly terminates, since Line 10 ensures that only simple paths in I are considered, and no simple path from an initial node is considered more than once.

(*Only if*) If the algorithm adds (u, z) to $Q_R(G)$, then it must traverse a depth-first search tree T rooted at (u, s_0) in which there is a simple path p from (u, s_0) to (z, r), $r \in F$.

Assume that p is not db-simple, and that the db-node v appears more than once on p. Let the first occurrence of v on p be in I-node (v, s) and the last such occurrence be in (v, t). Thus s was the first state added to $CM[v]$, and in order for SEARCH(u, v, t) to have been called in Line 13, Line 11 must have ensured that $[s] \supseteq [t]$. Hence, there is a path p' from (v, s) to (z, q), $q \in F$, in I such that the sequence of db-nodes on p' is identical to that on the path from (v, t) to (z, r) on p. Since (v, s) and (v, t) are the first and last occurrences, respectively, of v on p, there is a path from (u, s_0) to (z, q), $q \in F$, in I which is db-simple with respect to v.

A simple induction on the number of repeated db-nodes on p shows that there is a db-simple path from (u, s_0) to (z, s_f), $s_f \in F$.

(*If*) Assume there is a db-simple path p from (u, s_0) to (z, s_f), $s_f \in F$, in I. Obviously, if the algorithm traverses p we are done. Assume that it does not. Let (v, s) be the last node on p that is traversed, and (w, t) be the successor of (v, s) on p. The reason (w, t) is not visited cannot be because of a conflict, since p is db-simple. So it must have been the case that $t \in PM[w]$ at Line 10. By Lemmas 4 and 5, an entire depth-first search of I from (v, s) must have been performed. Since there is a path from (v, s) to (z, s_f) in I, SEARCH(u, z, s_f) must have been called in which case (u, z) would have been added to $Q_R(G)$ in Line 8. ∎

THEOREM 11. *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. The essential point is that, in the absence of conflicts, Algorithm C performs a normal depth-first search of the intersection graph which is polynomial in the size of the db-graph. A detailed analysis of the time complexity of the algorithm follows.

Let Q_R be a query where R is of length m, and G be a db-graph with n nodes and e edges. Although there can be as many as $O(2^m)$ states in a DFA accepting $L(R)$, this is just a constant in terms of the size of G. Nevertheless, we will assume that M has q states and will include q in our analysis of the time complexity of Algorithm C. Since M has at most $O(q^2)$ transitions, the intersection graph I for G and M has $O(qn)$ nodes and $O(q^2e)$ edges.

Line 1 of Algorithm C can be done in $O(q^2)$ time, while Line 2 requires only constant time. Line 3 takes $O(n)$ time and Line 4 $O(q^2)$ time. Line 5 is executed n times, and, in any execution, each node in I is visited at most once if I is conflict-free. This is because when (v, s) is stacked s is added to $CM[v]$ and Line 10 ensures that (v, s) cannot be restacked; when (v, s) is unstacked, s is added to $PM[v]$ (Line 16) and is not removed from $PM[v]$ until the present execution of Line 5(a) has terminated. Once again, Line 10 ensures that (v, s) cannot be revisited during the present execution of 5(a).

Only constant time is needed for Lines 6, 12 and 14. For each db-node v, $CM[v]$ can be implemented as a stack with access to its bottom element through the function FIRST. Hence, Lines 7, 11 and 15 can be performed in constant time, as can Line 16 since $\{s\}$ and $PM[v]$ are disjoint (by Line 10). Line 8 can be implemented to take $O(q)$ time: the pair (u, v) is added to $Q_R(G)$ if and only if there is no other final state in $PM[v]$. Line 10 can also be done in $O(q)$ time. Lines 9 and 10 inspect each edge leaving a node in I, and since no node in I can be revisited, SEARCH can be called $O(q^2e)$ times. Each call takes $O(q)$ time, so a single execution of Line 5(a) takes $O(q^3e)$ time. A single execution of Line 5(b) takes $O(n)$ time, so the total time spent in Line 5 is $O(n(q^3e + n))$. Consequently, Algorithm C runs in $O(n(q^3e + n))$ time. In terms of the size of G, Algorithm C runs in $O(ne)$ time (under the assumption that there are more edges than isolated nodes). ∎

From the relationship depicted in Figure 11, we obtain the following.

COROLLARY 5. *Algorithm C evaluates Q_R on G in time polynomial in the size of G if*

1. *R is restricted,*
2. *G is acyclic, or*
3. *G complies with a cycle constraint compatible with R.*

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. Let q be the length of R. If R is (*)-free, there are only a finite number of strings in $L(R)$ and the length of the longest such string is q. This then is also then an upper bound on the length of the longest db-simple path in I. Hence, there can be at most $O(n^q)$ db-simple paths in I. So even if Algorithm C traverses every db-simple path in I exactly once (the worst case), it still runs in polynomial time in the size of G.

A number of circumstances other than those identified above can lead to polynomial-time solutions. For example, there are certainly queries that can be evaluated in polynomial time on arbitrary db-graphs but whose regular expressions are not restricted. One such class of regular expressions are those of the form wa^*, where w is a string of fixed length. Unfortunately, there are db-graphs on which Algorithm C takes exponential time to evaluate the associated queries.

Clearly, there is much scope for further investigation. Additional classes of queries/db-graphs for which polynomial-time evaluation is possible should be identified and appropriate, more general evaluation algorithms developed. Algorithm C itself could be enhanced so that it reacts in a more sophisticated manner on detecting a conflict. One possibility is to flag the source of the conflict and not to unmark nodes until the algorithm backtracks from the flagged node.

6. Conclusions. 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 on relational databases 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.

Having considered restrictions on the structure of regular expressions, we turned our attention to the cyclic structure of the graphs being queried. We introduced the notion of a cycle constraint, and showed that if a graph G complied with a cycle constraint which was compatible with a regular expression R, then $Q_R(G)$ could be evaluated in polynomial time. Finally, we presented an algorithm for evaluating arbitrary expressions on arbitrary graphs. This algorithm runs in polynomial time if (a) the regular expression is restricted or closure-free, (b) the graph complies with a cycle constraint compatible with the regular expression (a special case being when the graph is acyclic), or (c) the regular expression and graph are conflict-free.

While it is difficult to say how often the above conditions will be encountered in practice, we did show that the class of restricted regular expressions is closed under the regular operators. A good starting point for investigation into larger classes of expressions and graphs with polynomial-time evaluation algorithms would be to attempt to identify the class of expressions and graphs which are not conflict-free, but on which Algorithm C runs in polynomial time.

Our emphasis in this paper has been on identifying circumstances in which the regular simple path problem can be solved in polynomial time, rather than designing the most efficient algorithm for these cases. We believe this is a topic for future research. For example, it would be interesting to see whether techniques used on sparse graphs, such as those in [16], could be employed in our algorithm in order to improve its efficiency on sparse graphs.

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 our query language $\mathbf{G}^+$ 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 [15].

Finally, we note that research has been done on the expressive power of graph-based query languages in which the restriction of simple path semantics is dropped. One such language that captures exactly the queries computable in nondeterministic logarithmic space is presented in [8]. On-line algorithms for regular path finding are given in [5], while a survey of many results can be found in [24].

REFERENCES

[1] R. AGRAWAL, *Alpha: An extension of relational algebra to express a class of recursive queries*, in Proceedings of the 3rd International Conference on Data Engineering, New York, 1987, IEEE, pp. 580–590.

[2] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] A. AHO AND J. ULLMAN, *Universality of data retrieval languages*, in Proceedings of the 6th ACM Symposium on Principles of Programming Languages, New York, 1979, ACM, pp. 110–120.

[4] F. BANCILHON, *On the completeness of query languages for relational databases*, in Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science, vol. 64 of Lecture Notes in Computer Science, New York, 1978, Springer-Verlag, pp. 112–123.

[5] A. BUCHSBAUM, P. KANELLAKIS, AND J. VITTER, *A data structure for arc insertion and regular path finding*, in Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 22–31.

[6] B. CARRÉ, *Graphs and Networks*, Oxford University Press, Oxford, England, 1979.

[7] E. CODD, *Relational completeness of data base sublanguages*, in Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 65–98.

[8] M. CONSENS AND A. MENDELZON, *Graphlog: a visual formalism for real life recursion*, in Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, Tenn., Apr. 2–4, 1990, ACM, New York, pp. 404–416.

[9] I. CRUZ, A. MENDELZON, AND P. WOOD, *A graphical query language supporting recursion*, in Proceedings of the ACM SIGMOD Conference on Management of Data, San Francisco, Calif., May 27–29, 1987, ACM, New York, pp. 323–330.

[10] ———, $\mathbf{G}^+$*: recursive queries without recursion*, in Proceedings of the 2nd International Conference on Expert Database Systems, Tysons Corner, Virginia, Apr. 25–27, 1988, Benjamin/Cummings, Redwood City, pp. 355–368.

[11] S. FORTUNE, J. HOPCROFT, AND J. WYLLIE, *The directed subgraph homeomorphism problem*, Theoretical Comput. Sci., 10 (1980), pp. 111–121.

[12] G. GRAHNE, S. SIPPU, AND E. SOISALON-SOININEN, *Efficient evaluation for a subset of recursive queries*, in Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, Calif., Mar. 23–25, 1987, ACM, New York, pp. 284–293.

[13] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[14] H. HUNT, D. ROSENKRANTZ, AND T. SZYMANSKI, *On the equivalence, containment, and covering problems for the regular and context-free languages*, J. Comput. System Sci., 12 (1976), pp. 222–268.

[15] Y. IOANNIDIS AND R. RAMAKRISHNAN, *Efficient transitive closure algorithms*, in Proceedings of the 14th International Conference on Very Large Data Bases, Los Angeles, Calif., Aug. 29–Sept. 1, 1988, Morgan Kaufmann, Palo Alto, pp. 382–394.

[16] B. JOUMARD AND M. MINOUX, *An efficient algorithm for the transitive closure and a linear worst-case complexity result for a class of sparse graphs*, Inf. Process. Lett., 22 (1986), pp. 163–169.

[17] A. LAPAUGH AND C. PAPADIMITRIOU, *The even-path problem for graphs and digraphs*, Networks, 14 (1984), pp. 507–513.

[18] A. MENDELZON AND P. WOOD, *Finding regular simple paths in graph databases*, in Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, The Netherlands, Aug. 22–25, 1989, Morgan Kaufmann, Palo Alto, pp. 185–193.

[19] A. ROSENTHAL, S. HEILER, U. DAYAL, AND F. MANOLA, *Traversal recursion: a practical approach to supporting recursive applications*, in Proceedings of the ACM SIGMOD Conference on Management of Data, Washington, DC, May 28–30, 1986, ACM, New York, pp. 166–176.

[20] Y. SHILOACH, *A polynomial solution to the undirected two paths problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 445–456.

[21] L. STOCKMEYER AND A. MEYER, *Word problems requiring exponential time*, in Proceedings of the 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, Apr. 30–May 2, 1973, ACM, New York, pp. 1–9.

[22] R. TARJAN, *Fast algorithms for solving path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 594–614.

[23] J. ULLMAN, *Implementation of logical query languages for databases*, ACM Trans. Database Syst., 10 (1985), pp. 289–321.

[24] M. YANNAKAKIS, *Graph-theoretic methods in database theory*, in Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, Tenn., Apr. 2–4, 1990, ACM, New York, pp. 230–242.