Finding Regular Simple Paths in Graph Databases

- Basic definitions
- Regular paths
- Regular simple paths
- An query evaluation algorithm

Example.

Let *G* be a graph describing a hypertext document:
Nodes – chucks of text
Edges – links (cross-references).
Readers read the document by following links.

Query: is there a way to get from Section 3.1 to Section 5.2 and then to the conclusion?

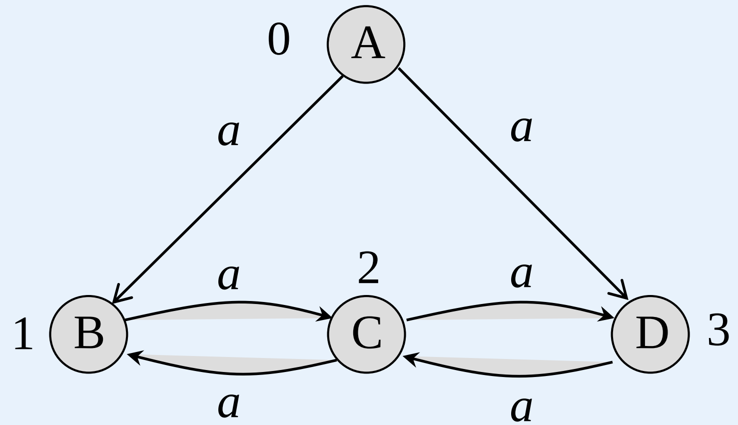
Basic definitions

We model a graph database as a labeled directed graph

$G = (V, E, \Sigma, \theta)$,

where V is a set of nodes,
E is a set of edges,
Σ is a set of symbols, called the *alphabet*, and
θ is an *edge labeling* function mapping E to Σ.

G:



Finding Regular Simple Paths

A regular path expression (or regular expression) is defined by the following grammar:

$$\alpha := ø \mid \varepsilon \mid a \mid - \mid (\beta_1 + \beta_2) \mid (\beta_1\beta_2) \mid \beta^*,$$

where α, β, β_1, and β_2 denote regular path expressions,
a denotes a constant in Σ,
"-" denotes a wildcard matching any constant in Σ,
ø denotes the empty set, and
ε denotes the empty string.

Example: (00)*, 0*10*, 1*01*, 0*10* + 1*01*.

Basic definitions

The language $L(\alpha)$ (a set of strings) created from α is defined as follows.

$L(\varepsilon) = \{\varepsilon\}$.

$L(\emptyset) = \emptyset$.

$L(a) = \{a\}$, for $a \in \Sigma$.

$L(\beta_1 + \beta_2) = L(\beta_1) \cup L(\beta_2) = \{w \mid w \in L(\beta_1)$ or $w \in L(\beta_2)$.

$L(\beta_1\beta_2) = L(\beta_1)L(\beta_2) = \{w_1w_2 \mid w_1 \in L(\beta_1)$ and $w_2 \in L(\beta_2)\}$.

$L(\beta^*) = \cup_{i=0}L^i(\beta)$, where $L^0(\beta) = \{\varepsilon\}$ and $L^i(\beta) = L^{i-1}(\beta)L(\beta)$.

Regular expressions α_1 and α_2 are equivalent, written $\alpha_1 \equiv \alpha_2$, if $L(\alpha_1) = L(\alpha_2)$. The length of regular expression α, denoted $|\alpha|$, is the number of symbols appearing in α.

Basic definitions

A *nondeterministic finite automaton* (NDFA) M is a 5-tuple

$(S, \Sigma, \delta, s_0, F)$,

where 1. S is the finite set of states of the control.
2. Σ is the alphabet from which input symbols are chosen.
3. δ is the state transition function which maps $S \times (\Sigma \cup \{\varepsilon\})$ to the set of subsets of S.
4. s_0 in S is the initial state of the finite control.

5. $F \subseteq S$ is the set of finite (or accepting) states.

Associated with an NDFA is a directed graph, in which
each node stands for a state in the NDFA, and
each edge (s, s') labeled with a symbol a in Σ for a state transit
$\delta(s, a)$ which contains s'.

The extended transition function δ^* is defined as follows.

- Let s and t be two states in S.
- For $a \in \Sigma$, and $w \in \Sigma^*$, $\delta^*(s, \varepsilon) = \{s\}$, and
$$\delta^*(s, wa) = \cup_{t \in \delta^*(s,w)} \delta(t, a).$$

Basic definitions

An NDFA $M = (S, \Sigma, \delta, s_0, F)$ 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 a nondeterministic finite automaton (S, I, δ, s_0, F) with the following conditions satisfied:

1. $\delta(s, \varepsilon) = \phi$ for all $s \in S$, and
2. Each state has 1 or 0 successor.

Simple paths

- Let Σ be a finite alphabet disjoint from $\{\varepsilon, \phi, (,)\}$.
- A regular expression R over Σ and the language $L(R)$ denoted by R are defined in the usual way.
- Let $G = (V, E, \Sigma, \theta)$ 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 $e_j \in E$, $1 \le j \le n$, be a path in G.
- We say p is a simple path if all the v_i's are distinct for $1 \le i \le n$. We call the string

  $$\theta(e_1) \ldots \theta(e_{n-1})$$

  the path label of p, denoted by $\theta(p) \in \Sigma^*$.

Finding Regular Simple Paths

Let R be a regular expression over Σ.

We say that the path p satisfies R if $\theta(p) \in L(R)$. The query Q_R on db-graph G, denoted by $Q_R(R)$, 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 $(x, y) \in Q_R(R)$, then (x, y) satisfies Q_R.

Regular simple path Problem

Instance: db-graph $G = (V, E, \Sigma, \theta)$,
nodes $x, y \in N$, regular expression
R over Σ.

Question: Does G contain a directed simple path

$$p = (v_1, e_1, \ldots, e_{n-1}, v_n)$$

from x to y such that p satisfies R, that is,

$$\theta(e_1) \ldots \theta(e_{n-1}) = \theta(p) \in L(R)?$$

Naïve method

A naïve 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.

Intersection graph

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 $a \in \Sigma$, $(p_1, q_1) \in \delta((p_2, q_2), a)$ 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.

Regular path Problem

Instance: db-graph $G = (V, E, \Sigma, \theta)$, nodes $x, y \in V$, regular expression R over Σ.

Question: Does G contain a directed path (not necessarily simple) $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$ from x to y such that p satisfies R, that is,

$$\theta(e_1) \ldots \theta(e_{n-1}) = \theta(p) \in L(R)?$$

Regular path Problem can be decided in polynomial time

- We view the db-graph $G = (V, E, \Sigma, \theta)$ 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 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.

Algorithm *A*

- A db-graph $G = (V, E, \Sigma, \theta)$ with nodes $x, y \in V$. (We view G as an NDFA with initial state x and final state y.)
- regular expression R over Σ.

Question: Does G contain a directed path (not necessarily simple)

$$p = (v_1, e_1, \ldots, e_{n-1}, v_n)$$

from x to y such that p satisfies R, that is,

$$(e_1) \ldots \theta(e_{n-1}) = \theta(p) \in L(R)?$$

Algorithm *A*

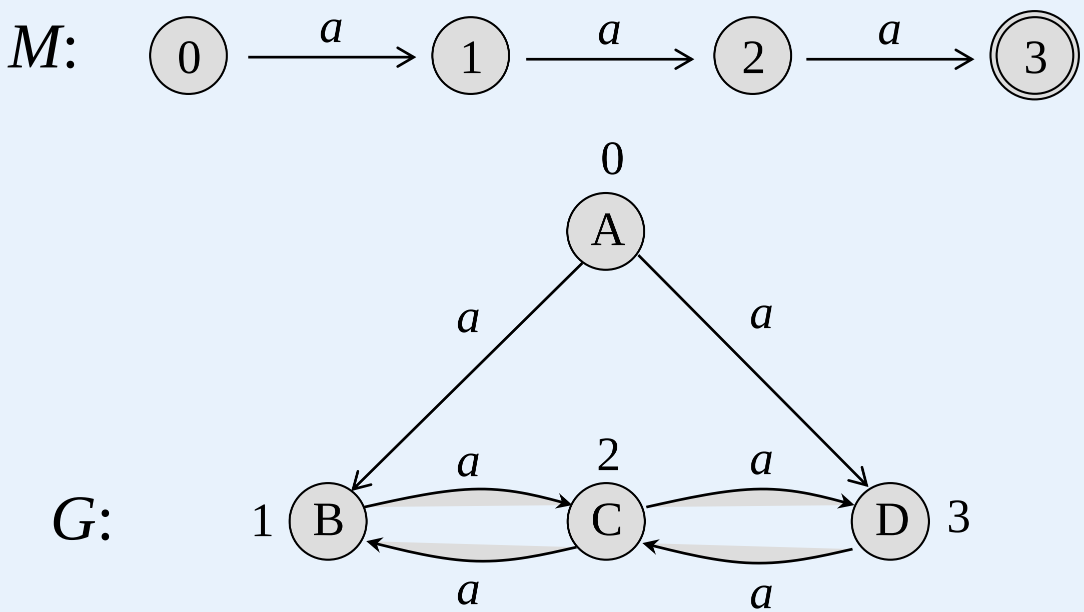
$$(e_1) \ldots \theta(e_{n-1}) = \theta(p) \in L(R)?$$

1. Traverse simple paths in *G*, using a DFA *M* accepting *L*(*R*) to control the search by marking nodes as they are visited.
2. Record with which state of *M* a node is visited. (We allow a node to be visited with different states.)
3. A node with the same state cannot be visited more than once.

Incompleteness

Using the above algorithm, we may fail to find all the answers.

Example Consider a query Q_R, where $R = aaa$.



Finding Regular Simple Paths

- Assume that we start traversal from node *A* in *G*, and follow the path to *B*, *C* and *D*. Node *A*, *B*, *C* and *D* are marked with 0, 1, 2 and 3, respectively, and the answer (*A*, *D*) is found, since 3 is a final state.
- If we 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. However, if we have retained markings, we cannot visit node *C* as it is already marked with state 2. Consequently, the answer (*A*, *B*) is not found.

Suffix language

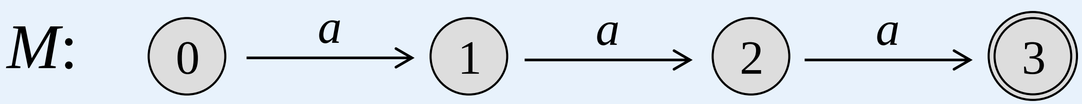
Definition Given an NDFA $M = (S, \Sigma, \delta, s_0, F)$, for each pair of states $s, t \in S$, we define the language from s to t, denoted by L_{st}, as the set of strings that take M from state s to state t. In particular, for a state $s \in S$, the suffix language of s, denoted by L_{sF} (or $[s]$), 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.

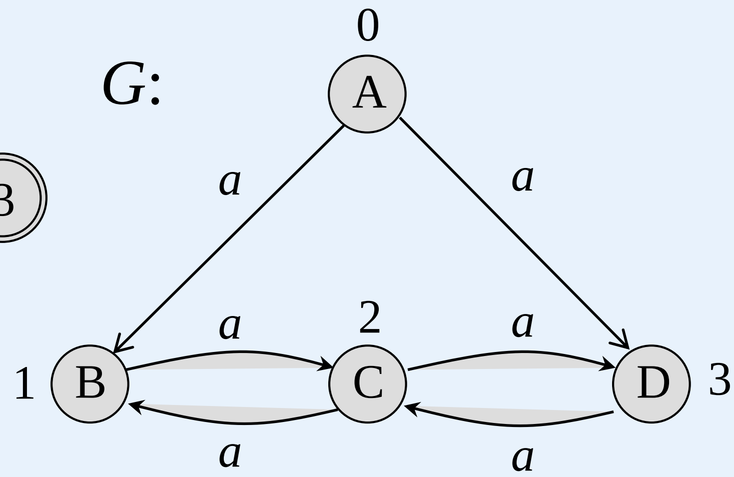
Suffix language

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 $s, t \in S$, 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 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, respectively, in G. If $[t] \not\subset [s]$, 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.

Example

Consider the following *M* and *G*.





Recall that, if markings were retained, the answer (*A*, *B*) would not be found. However, there is a conflict. 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 [3] $\not\subset$ [1].

Algorithm *B*

$$(e_1)\ \ldots\ \theta(e_{n-1}) = \theta(p) \in L(R)?$$

1. Traverse simple paths in G, using a DFA M accepting $L(R)$ to control the search by marking nodes as they are visited.
2. Record with which state of M a node is visited. (We allow a node to be visited with different states.)
3. If no conflicts are detected, the algorithm retains markings, while whenever a conflict arises, it unmarks nodes so that no answers are lost.
4. A node with the same state cannot be visited more than once.

Algorithm *B*

Input: db-graph $G = (V, E, \Sigma, \theta)$, query Q_R.

Output: $Q_R(G)$, the value of Q_R on G.

1. Construct a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$.
2. Initialize $Q_R(G)$ to $\varnothing$.
3. For each node $v \in V$, 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 $v \in V$,

   (a) call *search*-$G(v, v, s_0, conflict)$

   (b) reset $PM[w]$ to $\varnothing$ for any marked node $w \in V$.

Two types of markings:

$CM[v]$ – used to indicate that v is already on the stack

$PM[v]$ – a set of states, recording earlier markings of v, excluding the current path.

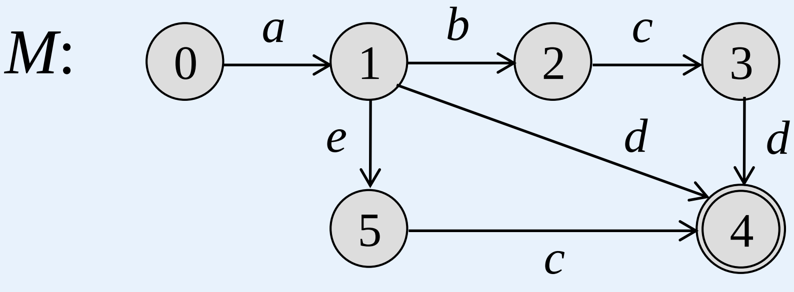
procedure *search-G*(u, v, s, **var** *conflict*)

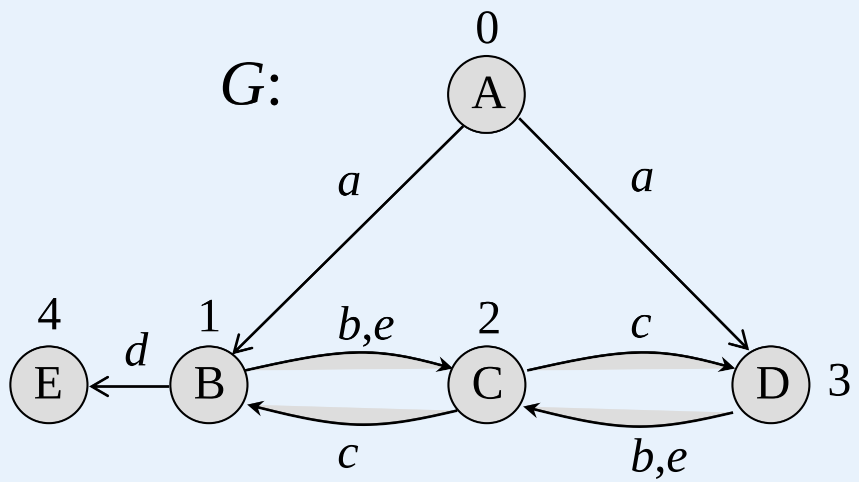
conflict $\leftarrow$ *false*
$CM[v] \leftarrow s$
if $s \in F$ **then** $Q_R(G) \leftarrow Q_R(G) \cup \{(u, v)\}$
for each edge in G from v to w with label a **do**
    if $\delta(s, a) = t$ and $t \notin PM[w]$ **then**
    if $CM[w] = q$ **then** *conflict* $\leftarrow ([t] \not\subset [q])$
    else /* $CM[w]$ is *null**/
        search-G(u, w, t, *new-conflict*)
        conflict $\leftarrow$ *conflict* **or** *new-conflict*
$CM[w] \leftarrow$ *null*
if not *conflict* **then** $PM[w] \leftarrow PM[w] \cup \{s\}$

Example

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 below.





Assume that we start by marking node A with state 0, after which we proceed to mark B with 1, C with 2, and B with 3.

Since no edge labeled d leaves B, we backtrack to C and attempt to visit B in state 3.

Finding Regular Simple Paths

- Although B has already a current marking ($CM[B] = 1$), this is not a conflict since $[1] \supseteq [3]$.
- The algorithm now backtrack to B in state 1 and marks E with state 4.
- After backtracking again to B in state 1, the markings are given as in the above figure.
- 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 $[4] \not\subset [1]$.
- So on backtracking to A, the markings 5 and 1 will be removed from C and B, respectively.

Finding Regular Simple Paths

- Now *D* is marked with 1, but since *C* has a previous marking of 2, that marking will not be repeated. So *C* is marked with 5 (along another transit labeled with *e* in *M*. 5 was previously removed.) After this, *B* can be marked with 4.
- When the algorithm backtracks to *C* and attempt to visit *D*, it discovers that *D* was previously marked with 4, so no conflict is registered. The marking are now given as shown below.