Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the
11th International Workshop on Graph Transformation and
Visual Modeling Techniques
(GTVMT 2012)

Incremental Pattern Matching for Regular Expressions

Arash Jalali, Amir Hossein Ghamarian, Arend Rensink

12 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Incremental Pattern Matching for Regular Expressions

Arash Jalali[2], Amir Hossein Ghamarian[1], Arend Rensink[3]

[1] a.h.ghamarian@cs.utwente.nl
[3] rensink@cs.utwente.nl
Department of Computer Science,
University of Twente, The Netherlands

[2] arash@netstairs.com
NetStairs.com, Inc.

Abstract: Graph pattern matching lies at the heart of any graph transformation-based system. Incremental pattern matching is one approach proposed for reducing the overall cost of pattern matching over successive transformations by preserving the matches that stay relevant after a rule application. An important issue in any matching scheme, is the ability to properly and consistently deal with various facilities that add to the expressiveness of a GT-tool's rule language. One such feature is the support for regular path expressions, which would let two nodes to be considered as a "match", if a certain path of edges exists between them. In this paper, the incorporation of regular expression support into incremental pattern matching is discussed within the context of the GROOVE tool set. This includes laying down a formal foundation for incremental pattern matching for regular expressions which is then used to justify the extension proposed to add regular expression support to a well-known pattern matching algorithm.

Keywords: Incremental Matching, Path Matching, Regular Expressions, RETE Algorithm, State Space Exploration

1 Introduction

Tool performance is essential to the success of any specification formalism. This is certainly also true for graph transformation, which has an innate disadvantage that it crucially relies on the computationally expensive problem of pattern matching.

Previous work of Bergmann et al [BÖR+08] has shown that the principle of *incremental matching*, in the form of the RETE algorithm [BGT91, For82], can outperform plan-based searching (as advocated in, e.g., [GBG+06, VVF06]). The fundamental observation is that graph transformation is all about the gradual evolution of graphs, and therefore search results found for a given graph mostly continue to be valid for any graph obtained by the application of a single rule. We have ourselves reported that the same is true for the exhaustive state space exploration in GROOVE; see [GJR10].

In this paper we extend the scope of the RETE algorithm to matching regular expressions. These were already included in PROGRES [SWZ99] (where they are called path expressions) and are also offered by GROOVE. A regular expression can be used on a rule edge to indicate that the

host graph is expected to have a path between the image of the rule edge's source to the image of the rule edge's target, such that the path labels, when concatenated, form a word in the language of the regular expression. Typically, one is only interested in the *existence* (or *absence*) of such a path, and not in the precise path itself. A limited version of the problem of dynamic transitive closure, where one is merely interested in answering queries of the form "*is node y reachable from node x?*" in the face of updates to the graph, has already been dealt with in the past (see [DI06] for an exhaustive survey).

The existing RETE algorithm for incremental matching is not geared towards checking regular expressions. The main hurdle is the fact that the match of a regular expression does not have a fixed size; in particular, transitively closed expressions allow paths of arbitrary length. This does not sit well with the basic principle of RETE, which relies on the propagation of a match through a static, non-cyclic network of subgraph checker nodes. For regular expressions this network needs to be cyclic; this could potentially make the propagation non-terminating.

Research question and approach. The questions we set out to answer in this paper are: (i) can RETE be extended to cope with regular expression matching, and (ii) does the resulting algorithm continue to outperform search plan-based matching?

In order to answer the first question, we will extend the concept of a RETE network with path checker nodes; in contrast to the existing subgraph checkers, path checkers may be connected in a cycle. We will then give a cut-off criterion to stop the propagation and guarantee termination. The main proof obligation is to show that the set of paths found at the cut-off moment is large enough to include at least one representative path between every pair of connected nodes.

However, it is not enough to have a terminating algorithm: it should also perform well in practice. This means that the cut-off criterion should be as sharp as possible, so that the propagation is fast enough to beat ordinary regular expression matching. In order to evaluate this, we have added regular expression matching to the RETE implementation in GROOVE and compared the performance with the pre-existing, search plan-based matching algorithm.

The results show that incremental matching continues to pay off in the presence of regular expressions, but that the actual gain depends on the dynamics of the rules. In particular, if the structure that is tested by regular expression matching changes frequently, then incremental matching loses out. This is indeed not surprising, as match propagation for path checker nodes is a potentially complex operation which, when repeated often, does not scale as well as the time-honoured word checking algorithm for regular automata.

2 Definitions

We first define some basic concepts: graphs, paths and match morphisms.

Graphs consist of nodes and labelled edges. We assume the existence of global universes Vertex (ranged over by v), Edge (ranged over by e), and Label of vertex (or node), edge and label entities. Label is further partitioned into the universe Atom of atomic labels (ranged over by a) and RegExp of *regular expressions* (ranged over by R). There are also global functions $src, tgt : \mathsf{Edge} \to \mathsf{Vertex}$ and $lab : \mathsf{Edge} \to \mathsf{Label}$ associating with every edge a source and target node and label.

Definition 1 (graph) A graph G is a tuple $\langle V_G, E_G\rangle$ in which $V_G \subseteq \mathsf{Vertex}$ is the set of vertices (or nodes) and $E_G \subseteq \mathsf{Edge}$ is the set of edges, such that $src(E_G) \subseteq V_G$ and $tgt(E_G) \subseteq V_G$.

We distinguish *host graphs*, in which only Atom is used as label universe, and *rule graphs*, in which also RegExp labels may occur. Regular expressions are defined by the following grammar:

$$\mathsf{R} ::= \lambda \mid a \mid -\mathsf{R} \mid \mathsf{R}_1|\mathsf{R}_2 \mid \mathsf{R}_1.\mathsf{R}_2 \mid \mathsf{R}^*$$

where λ stands for the empty word and $a \in \mathsf{Atom}$. The only non-standard operation is $-\mathsf{R}$. In graphs, this will be matched by an inversed path; to capture this formally, we introduce the *complementary actions* $\overline{\mathsf{Atom}}$, disjoint from Label. For arbitrary $a \in \mathsf{Atom} \cup \overline{\mathsf{Atom}}$ we use $\bar{a}$ to denote the complement of a; this operator is its own inverse, hence $\bar{\bar{a}} = a$. This is extended to words $w \in \mathsf{Word}$ where $\mathsf{Word} = (\mathsf{Atom} \cup \overline{\mathsf{Atom}})^*$ by also reversing them: $\bar{\varepsilon} = \varepsilon$ and $\overline{aw} = \bar{w}\,\bar{a}$.

The language of a regular expression R is then inductively defined by $L_\mathsf{R} \subseteq \mathsf{Word}$ such that:

$$\begin{aligned}
L_\lambda &= \{\varepsilon\} \\
L_a &= \{a\} \\
L_{-\mathsf{R}} &= \{\bar{w} \mid w \in L_\mathsf{R}\} \\
L_{\mathsf{R}_1.\mathsf{R}_2} &= \{w_1 w_2 \mid w_1 \in L_{\mathsf{R}_1}, w_2 \in L_{\mathsf{R}_2}\} \\
L_{\mathsf{R}_1|\mathsf{R}_2} &= L_{\mathsf{R}_1} \cup L_{\mathsf{R}_2} \\
L_{\mathsf{R}^*} &= \{w_1 \cdots w_n \mid n \geq 0, w_1, \ldots, w_n \in L_\mathsf{R}\}
\end{aligned}$$

To define the meaning of regular expressions over graphs, we need the concept of *paths* in a host graph. For this purpose, we first introduce *inverse edges*: $\bar{E}_G$ for a graph G will denote the set of inverse edges, such that $src(\bar{e}) = tgt(e)$, $tgt(\bar{e}) = src(e)$ and $lab(\bar{e}) = \overline{lab(e)}$.

Definition 2 (path) A *path* is an alternating sequence $p = v_1\, e_2\, v_2 \cdots e_n\, v_n$ of vertices and edges, starting and ending on a vertex, such that $v_{i-1} = src(e_i)$ and $tgt(e_i) = v_i$ for $1 < i \leq n$.

We use Π to denote the set of all paths and $\Pi(G)$ for the subset of all paths in G, and extend the source and target functions to paths by $src(p) = v_1$ and $tgt(p) = v_n$. Path concatenation of $p = v_1\, e_2\, v_2 \cdots e_n\, v_n$ and $p = v'_1\, e'_2\, v'_2 \cdots e'_{n'}\, v'_{n'}$ is defined by

$$p.p' = v_1\, e_2\, v_2 \cdots e_n\, v_n\, e'_2\, v'_2 \cdots e'_{n'}\, v'_{n'} \qquad \text{if } v_n = v'_1$$

Two further relevant functions on paths are $word : \Pi \to \mathsf{Word}$ that maps each path in G to the corresponding sequence of labels, and path reversal $rev : \Pi \to \Pi$, inductively defined by:

$$\begin{aligned}
word(v) &= \varepsilon & \qquad rev(v) &= v \\
word(p\,e\,v) &= word(p)\, lab(e) & \qquad rev(p\,e\,v) &= v\,\bar{e}\, rev(p)
\end{aligned}$$

We can now formally define the relation between regular expressions and graphs. For an arbitrary regular expression R and graph G, we define $\Pi^\mathsf{R}(G)$ as the set of paths in G satisfying R, thus:

$$\Pi^\mathsf{R}(G) = \{p \in \Pi(G) \mid word(p) \in L_\mathsf{R}\} \ . \qquad (1)$$

Note that $\Pi^{\mathsf{R}}(G)$ is in general infinite, due to the possible presence of loops in G.

Finally, a *match morphism* is a function from a rule graph L to a host graph G that provides images in G for the atom-labelled edges of L and guarantees the existence of paths in G for the regular expression-labelled edges of L:

Definition 3 (match morphism) Given a rule graph L and a host graph G, a match morphism of L into G is a partial function $f : (V_L \cup E_L) \to (V_G \cup E_G)$ that is total on V_L such that $f(V_L) \subseteq V_G$ and $f(E_L) \subseteq E_G$, satisfying for all $e \in E_L$:

- if $lab(e) = a \in \mathsf{Atom}$, then $f(e) = e' \in E_G$ such that $src(e') = f(src(e))$, $tgt(e') = f(tgt(e))$ and $lab(e') = a$;
- if $lab(e) = \mathsf{R} \in \mathsf{RegExp}$, then there is a $p \in \Pi^{\mathsf{R}}(G)$ such that $src(p) = f(src(e))$ and $tgt(p) = f(tgt(e))$.

For the purpose of our discussions, we shall only be concerned with the second condition in the above definition. We use F^L to denote the set of all match morphisms of L, and $F^L(G)$ for the subset of match morphisms into G. It is important to note that the paths themselves are not part of the match morphism images. Among other things, this implies that $F^L(G)$ is always finite.

It should also be noted that, in this paper, we deal with positivie conditions only. Negative application conditions can be supported using the same techniques as described in [GJR10].

3 Incremental Matching

The RETE algorithm for incremental matching uses the notion of *subgraph checkers* that keep a record of their match morphisms into a host graph, and are *updated* upon every graph change. Graph changes are additions and deletions of nodes and edges, using the following (partially defined) operations:

$$
\begin{array}{ll}
G+v = \langle V \cup \{v\}, E\rangle & \text{for all } v \notin V_G \\
G+e = \langle V, E \cup \{e\}\rangle & \text{for all } e \notin E_G \text{ with } src(e), tgt(e) \in V_G \\
G-v = \langle V \setminus \{v\}, E\rangle & \text{for all } v \in V_G \text{ such that } \nexists e \in E_G : v \in \{src(e), tgt(e)\} \\
G-e = \langle V, E \setminus \{e\}\rangle & \text{for all } e \in E_G
\end{array}
$$

The essence of incremental matching is the existence of "cheap" updates $\delta^L_{+x}(G), \delta^L_{-x}(G) \subseteq F^L$ for every rule graph L host graph G and graph element $x \in \mathsf{Vertex} \cup \mathsf{Edge}$, which capture the *change* to $F^L(G)$ when a vertex or edge is added or removed:

$$
\begin{array}{ll}
F^L(G+v) = F^L(G) \cup \delta^L_{+v}(G) & F^L(G-v) = F^L(G) \setminus \delta^L_{-v}(G) \\
F^L(G+e) = F^L(G) \cup \delta_{+e}(G) & F^L(G-e) = F^L(G) \setminus \delta^L_{-e}(G)
\end{array}
$$

whenever the graph operations are defined. In RETE, this is achieved by creating a directed acyclic graph $\langle N, \sqsubset\rangle$ (called a network) of *checker nodes* $n \in N$, each of which is associated with a subgraph $S_n \subseteq L$ such that $n_1 \sqsubset n_2$ implies $S_{n_1} \subset S_{n_2}$. For the $\sqsubset$-minimal nodes, the S_n are single nodes and edges of L, whereas $S_n = L$ when n is the $\sqsubset$-maximum.

Each checker node n continually maintains $F^{S_n}(G)$, and computes its own δ's upon every graph update, propagating from small to large in the $\sqsubset$-ordering. For instance, for a rule graph $L = \langle \{v'_0, v'_1\}, \{e'\} \rangle$ consisting only of a single edge e' with $src(e') = v'_0$ and $tgt(e') = v'_1$, the δ's are given by

$$\delta^L_{+v}(G) = \delta^L_{-v}(G) = \emptyset$$

$$\delta^L_{+e}(G) = \delta^L_{-e}(G) = \begin{cases} \{(v'_0 \mapsto v_0, v'_1 \mapsto v_1, e' \mapsto e)\} & \text{if } lab(e) = lab(e') \\ \emptyset & \text{otherwise} \end{cases}$$

Only non-empty δ are propagated to the $\sqsubset$-successors in the network, reducing the amount of work required to update the entire structure.

3.1 Path matches

Since this paper deals with the extension of RETE to regular expressions, we will not go further into the details of the standard algorithm. The extension involves the definition of new checker nodes for paths and extending the RETE network with those nodes. There are two important novelties:

- The path checker nodes do not have to maintain all paths, but only have to test for the existence of a path (see Def. 3).
- The $\sqsubset$-ordering over path checkers is no longer acyclic; instead, we will have $n \sqsubset n$ whenever R_n (the regular expression associated with n) is of the form R^*.

The main issue is to decide what kind of information every path checker node n should maintain, given that it is only required to know whether a satisfying path exists between given (arbitrary) $v_1, v_2 \in V_G$. Important criteria for this decision are: (i) the amount of information should be kept as small as possible and (ii) incremental updates should be cheap. To meet these criteria, we introduce an *algebra of path matches*:

$$m ::= \langle v_0 \rangle \mid \langle e \rangle \mid \mathsf{inv}(m) \mid \mathsf{dot}(m_1, m_2) \mid \mathsf{close}(m_1, m_2)$$

Here, $v_0 \in \mathsf{Vertex}$ and $e \in \mathsf{Edge}$; the other operators are syntactic constructors that give rise to a tree-like structure of path matches. The close constructor, for instance, represents a match for a closure regular expression. The universe of all path matches will be denoted M. The following table defines functions $start, end : \mathsf{M} \to \mathsf{Vertex}$ that retrieve the start and end nodes of a path match, and a function $int : \mathsf{M} \to 2^{\mathsf{Vertex}}$ that retrieves the set of internal nodes:

m	$start(m)$	$end(m)$	$int(m)$
$\langle v_0 \rangle$	v_0	v_0	$\emptyset$
$\langle e \rangle$	$src(e)$	$tgt(e)$	$\emptyset$
$\mathsf{inv}(m_1)$	$end(m_1)$	$start(m_1)$	$\emptyset$
$\mathsf{dot}(m_1, m_2)$	$start(m_1)$	$end(m_2)$	$\emptyset$
$\mathsf{close}(m_1, m_2)$	$start(m_1)$	$end(m_2)$	$\{start(m_1)\} \cup int(m_2)$

m is said to be a path match *in* G if it only uses nodes and edges from G. We call a path match m *consistent* if

- $m = \mathsf{inv}(m_1)$ implies m_1 is consistent;
- $m = \mathsf{dot}(m_1, m_2)$ implies (i) m_1 and m_2 are consistent and (ii) $end(m_1) = start(m_2)$;
- $m = \mathsf{close}(m_1, m_2)$ implies (i) m_1 and m_2 are consistent, (ii) $end(m_1) = start(m_2)$, (iii) $start(m_1) \notin int(m_2)$ and (iv) $m_2 = \langle v_0 \rangle$ or $m_2 = \mathsf{close}(m_{21}, m_{22})$.

Note in particular condition (iv) of close, which states that the second operand is always either an empty path match or itself of the form close.

Obviously, every path match is essentially a path with additional structure. Indeed, we can easily retrieve the path from the path match:

$$\begin{aligned}
path(\langle v_0 \rangle) &= v_0 \\
path(\langle e \rangle) &= src(e)\, e\, tgt(e) \\
path(\mathsf{inv}(m)) &= rev(path(m)) \\
path(\mathsf{dot}(m_1, m_2)) &= path(m_1).path(m_2) \\
path(\mathsf{close}(m_1, m_2)) &= path(m_1).path(m_2)
\end{aligned}$$

We now recursively define the path matches of arbitrary regular expressions in a given graph G, as the smallest sets satisfying the following equations:

$$\begin{aligned}
M^{\lambda}(G) &= \{\langle v \rangle \mid v \in V_G\} \\
M^{a}(G) &= \{\langle e \rangle \mid e \in E_G, lab(e) = a\} \\
M^{-\mathsf{R}}(G) &= \{\mathsf{inv}(m) \mid m \in M^{\mathsf{R}}(G)\} \\
M^{\mathsf{R}_1|\mathsf{R}_2}(G) &= M^{\mathsf{R}_1}(G) \cup M^{\mathsf{R}_2}(G) \\
M^{\mathsf{R}_1.\mathsf{R}_2}(G) &= \{m = \mathsf{dot}(m_1, m_2) \mid m_1 \in M^{\mathsf{R}_1}(G), m_2 \in M^{\mathsf{R}_2}(G), m \text{ is consistent}\} \\
M^{\mathsf{R}^*}(G) &= M^{\lambda}(G) \cup \{m = \mathsf{close}(m_1, m_2) \mid m_1 \in M^{\mathsf{R}}(G), m_2 \in M^{\mathsf{R}^*}(G), m \text{ is consistent}\}
\end{aligned}$$

In particular, the path matches of the Kleene star are defined recursively. The following property is the core of the correctness proof for our incremental match algorithm.

Theorem 1 *Let G be a host graph and R a regular expression.*

1. *$M^{\mathsf{R}}(G)$ is finite;*
2. *For every $m \in M^{\mathsf{R}}(G)$, $path(m) \in \Pi^{\mathsf{R}}(G)$;*
3. *For every $p \in \Pi^{\mathsf{R}}(G)$, there is an $m \in M^{\mathsf{R}}(G)$ such that $start(m) = src(p)$ and $end(m) = tgt(p)$.*

Clause 1 ensures that $M^{\mathsf{R}}(G)$ can be effectively computed. It follows from the fact that, due to the consistency conditions on path matches, $int(\mathsf{close}(m_1, m_2)) \supset int(m_2)$, combined with the fact that $int(m) \subseteq V_G$ for all $m \in M^{\mathsf{R}}(G)$. Clause 2 guarantees that the path matches are sound: they only generate paths that satisfy the regular expression. Finally, Clause 3, called the *sufficient coverage* clause, states that the path matches are complete: they contain at least one representative element for every pair of host nodes between which there exists a path.

This provides us with an answer to the question above: the RETE path checker node for R will maintain the set $M^{\mathsf{R}}(G)$. In the remainder of this section we will show that this can be updated incrementally.

3.2 Incremental Computation of Path Matches

Given the definition of the path matches, defining the δ-functions, which now range over the terms of our path match algebra, is relatively straightforward. For the base regular expressions they are given by

$$\delta^{\lambda}_{+v}(G) = \delta^{\lambda}_{-v}(G) = \{\langle v\rangle\} \qquad \delta^{\lambda}_{+e}(G) = \delta^{\lambda}_{-e}(G) = \emptyset$$

$$\delta^{a}_{+v}(G) = \delta^{a}_{-v}(G) = \emptyset \qquad \delta^{a}_{+e}(G) = \delta^{a}_{-e}(G) = \begin{cases} \{\langle e\rangle\} & \text{if } lab(e) = a \\ \emptyset & \text{otherwise} \end{cases}$$

For the composed regular expressions we can ignore the distinction between node and edge updates: for $x \in \mathsf{Vertex} \cup \mathsf{Edge}$

$$\begin{aligned}
\delta^{-\mathsf{R}}_{+x}(G) &= \{\mathsf{inv}(m) \mid m \in \delta^{\mathsf{R}}_{+x}(G)\} \\
\delta^{\mathsf{R}_1|\mathsf{R}_2}_{+x}(G) &= \delta^{\mathsf{R}_1}_{+x}(G) \cup \delta^{\mathsf{R}_2}_{+x}(G) \\
\delta^{\mathsf{R}_1.\mathsf{R}_2}_{+x}(G) &= \{m = \mathsf{dot}(m_1,m_2) \mid m_1 \in \delta^{\mathsf{R}_1}_{+x}(G), m_2 \in M^{\mathsf{R}_2}(G+x), m \text{ is consistent}\} \\
&\quad \cup \{m = \mathsf{dot}(m_1,m_2) \mid m_1 \in M^{\mathsf{R}_1}(G+x), m_2 \in \delta^{\mathsf{R}_2}_{+x}(G), m \text{ is consistent}\} \\
\delta^{\mathsf{R}^*,0}_{+x}(G) &= \delta^{\lambda}_{+x}(G) \cup \{m = \mathsf{close}(m_1,m_2) \mid m_1 \in \delta^{\mathsf{R}}_{+x}(G), m_2 \in M^{\mathsf{R}^*}(G), m \text{ is consistent}\} \\
\delta^{\mathsf{R}^*,i+1}_{+x}(G) &= \{m = \mathsf{close}(m_1,m_2) \mid m_1 \in M^{\mathsf{R}}(G+x), m_2 \in \delta^{\mathsf{R}^*,i}_{+x}(G), m \text{ is consistent}\} \\
\delta^{\mathsf{R}^*}_{+x}(G) &= \textstyle\bigcup_{i=0}^{\omega} \delta^{\mathsf{R}^*,i}_{+x}(G)
\end{aligned}$$

The most interesting is of course the case for R^*. This is approximated by a sequence of steps, which is defined as unbounded but in practice will always result in $\delta^{\mathsf{R}^*,i}_{+x}(G) = \emptyset$ from a certain value of i onward — the argument is the same as for the finiteness of $M^{\mathsf{R}^*}(G)$.

The removal updates for the composed operators only have to look at existing matches; in other words, we do not have to compute them:

$$\begin{aligned}
\delta^{-\mathsf{R}}_{-x}(G) &= \{\mathsf{inv}(m) \in M^{\mathsf{R}}(G) \mid m \in \delta^{\mathsf{R}}_{-x}(G)\} \\
\delta^{\mathsf{R}_1|\mathsf{R}_2}_{-x}(G) &= \delta^{\mathsf{R}_1}_{-x}(G) \cup \delta^{\mathsf{R}_2}_{-x}(G) \\
\delta^{\mathsf{R}_1.\mathsf{R}_2}_{-x}(G) &= \{\mathsf{dot}(m_1,m_2) \in M^{\mathsf{R}}(G) \mid m_1 \in \delta^{\mathsf{R}_1}_{-x}(G) \text{ or } m_2 \in \delta^{\mathsf{R}_2}_{-x}(G)\} \\
\delta^{\mathsf{R}^*,0}_{-x}(G) &= \delta^{\lambda}_{-x}(G) \cup \{\mathsf{close}(m_1,m_2) \in M^{\mathsf{R}}(G) \mid m_1 \in \delta^{\mathsf{R}}_{-x}(G)\} \\
\delta^{\mathsf{R}^*,i+1}_{+x}(G) &= \{m = \mathsf{close}(m_1,m_2) \mid m_2 \in \delta^{\mathsf{R}^*,i}_{-x}(G)\} \\
\delta^{\mathsf{R}^*}_{-x}(G) &= \textstyle\bigcup_{i=0}^{\omega} \delta^{\mathsf{R}^*,i}_{-x}(G)
\end{aligned}$$

In an actual implementation, removal can be implemented efficiently by keeping a mapping from each match to the set of matches derived from it; i.e., from m to all existing matches of the form $\mathsf{inv}(m)$, $\mathsf{dot}(m,m')$, $\mathsf{dot}(m',m)$, $\mathsf{close}(m',m)$ and $\mathsf{close}(m,m')$. This obviates all further lookups.

The correctness criterion of incremental updating is captured by the following theorem:

Theorem 2 *For every graph G, regular expression* R *and element* $x \in \mathsf{Vertex} \cup \mathsf{Edge}$,

$$M^{\mathsf{R}}(G+x) = M^{\mathsf{R}}(G) \cup \delta^{\mathsf{R}}_{+x}(G)$$
$$M^{\mathsf{R}}(G-x) = M^{\mathsf{R}}(G) \setminus \delta^{\mathsf{R}}_{-x}(G) \quad .$$

In order to show this theorem for the choice operator, we need an alternative characterisation of the set δ^{R}_{-x}, namely that it always removes precisely those match objects that contain x. For every $x \in \mathsf{Vertex} \cup \mathsf{Edge}$ define

$$M_x = \{m \in M \mid x \text{ occurs in } m\} \quad .$$

Now we can easily show the following property, by induction on the structure of R:

Proposition 1 *For every graph G, regular expression* R *and element* $x \in \mathsf{Vertex} \cup \mathsf{Edge}$,

$$\delta^{\mathsf{R}}_{-x}(G) = M^{\mathsf{R}}(G) \cap M_x \quad .$$

With the use of this property we can prove the main theorem above.

Proof of Theorem 2. By induction on the structure of R. The only really interesting cases are those of $M^{\mathsf{R}_1|\mathsf{R}_2}(G-x)$ and $M^{\mathsf{R}^*}(G+x)$.

Case $M^{\mathsf{R}_1|\mathsf{R}_2}(G-x) = M^{\mathsf{R}_1|\mathsf{R}_2}(G) \setminus \delta^{\mathsf{R}_1|\mathsf{R}_2}_{-x}(G)$**.** This follows from

$$\begin{aligned}
M^{\mathsf{R}_1|\mathsf{R}_2}(G-x) &= M^{\mathsf{R}_1}(G-x) \cup M^{\mathsf{R}_2}(G-x) && \text{(by definition)}\\
&= (M^{\mathsf{R}_1}(G) \setminus \delta^{\mathsf{R}_1}_{-x}) \cup (M^{\mathsf{R}_2}(G) \setminus \delta^{\mathsf{R}_2}_{-x}) && \text{(by the induction hypothesis)}\\
&= (M^{\mathsf{R}_1}(G) \setminus M_x) \cup (M^{\mathsf{R}_2}(G) \setminus M_x) && \text{(by Prop. 1)}\\
&= (M^{\mathsf{R}_1}(G) \cup M^{\mathsf{R}_2}(G)) \setminus M_x && \text{(by set algebra)}\\
&= M^{\mathsf{R}_1|\mathsf{R}_2}(G) \setminus \delta^{\mathsf{R}_1|\mathsf{R}_2}_{-x}(G) && \text{(by definition and Prop. 1).}
\end{aligned}$$

Case $M^{\mathsf{R}^*}(G+x) = M^{\mathsf{R}^*}(G) \cup \delta^{\mathsf{R}^*}_{+x}(G)$**.** The inclusion from right to left is immediate. Now assume $m \in M^{\mathsf{R}^*}(G+x)$. We proceed by induction on $|m|$, where $|m| = 0$ if m is not of the form close, and $|m| = |m_2| + 1$ if $m = \mathsf{close}(m_1, m_2)$. The hypothesis is that either $m \in M^{\mathsf{R}^*}(G)$ or $m \in \delta^{\mathsf{R}^*,i}_{+x}(G)$ for some $i \leq |m|$.

Base case: $|m| = 0$ It follows that $m \in M^{\lambda}(G+x)$. Then $m \in M^{\lambda}(G)$ or $m \in \delta^{\lambda}_{+x}(G)$ by the outer induction hypothesis; in the first case $m \in M^{\mathsf{R}^*}(G)$ and in the second $m \in \delta^{\mathsf{R}^*,0}_{+x}(G)$. Since $0 \leq |m|$ we are done.

Induction step: $|m| = n+1$**.** It follows that $m = \mathsf{close}(m_1, m_2)$ with $|m_2| = n$; hence $m_1 \in M^{\mathsf{R}}(G+x)$ and $m_2 \in M^{\mathsf{R}^*}(G+x)$. By the outer induction hypothesis, either (i) $m_1 \in M^{\mathsf{R}}(G)$ or (ii) $m_1 \in \delta^{\mathsf{R}}_{+x}(G)$, and by the inner induction hypothesis, either (iii) $m_2 \in M^{\mathsf{R}^*}(G)$ or (iv) $m_2 \in \delta^{\mathsf{R}^*,i}_{+x}(G)$ for $i \leq |m_2|$. This gives rise to the following combinations of cases:

- (i) and (iii): Then $m \in M^{\mathsf{R}^*}(G)$, hence we are done.
- (ii) and (iii): Then $m \in \delta_{+x}^{\mathsf{R}^*,0}(G)$; since $0 \leq |m|$ we are done.
- (iv): Then $m \in \delta_{+x}^{\mathsf{R}^*,i+1}(G)$; since $i+1 \leq |m|$ we are done.

□

4 Implementation

In this section we briefly discuss how the extended RETE algorithm has been implemented in GROOVE. Our discussion will be exclusive to regular expressions. For a more general explanation of RETE and its implementation in GROOVE, the reader is referred to [GJR10]. Our explanation will mainly focus on the static build of the RETE network, as the dynamic behavior of the network follows directly from the formal semantics of the δ operations.

4.1 Path-checkers

The RETE algorithm relies heavily on the network of checker nodes that are responsible for finding and passing along partial matches for one or more rules. The network in the classic RETE algorithm comes equipped with several types of checker nodes, including *edge checkers*, *node checkers*, *subgraph checkers*, as well as *condition/production rule checkers*. In order to add the functionality of finding matches for paths, as opposed to partial graphs, it is natural to think that one needs a new type of checker node that finds and passes along matches for a specific kind of path - hence the addition of *path checkers*.

A path checker node, in general, receives one or two partial path matches, and then tries to combine and/or transform them based on the semantics of the regular expression operator they represent. For instance, a *sequence path checker*, receives two path matches, and tries to see if they can be combined into a larger path by sequentially concatenating them.

4.2 The RETE Static Build

Given a rule edge with a regular expression label R, we use the syntax of R as a guide to build the RETE network, such that it would be able to incrementally find path matches. Figure 1 shows

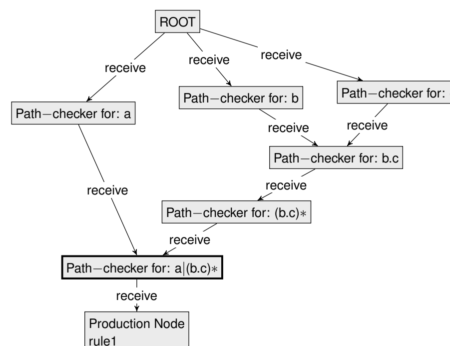

Figure 1: RETE network for a|(b.c)*

(a) ‘add’

(b) ‘del’

(c) ‘move’

(d) ‘mark’

(e) A chain of nodes

Figure 2: The chain manipulation example

the RETE network that finds matches for the expression a|(b.c)*. It is not a coincidence that the way the path checkers are connected to each other represents an upside-down view of the expression’s syntax tree, as syntax trees are top-down representations of an expression, while the RETE network is constructed to build matches in a bottom-up way. The highlighted path checker in Figure 1 is the one that performs the final choice operation. It should also be noted that although the network looks acyclic, the closure path checker for (b.c)+ does in fact have a built-in loop-back mechanism that performs the closure operation repeatedly enough, i.e. until the path matches become inconsistent, to construct a sufficient coverage for that expression.

5 Experimental Results

Figure 2, shows a grammar for manipulating chains of nodes. Four operations (grammar rules) are defined over these chains: *marking* the end of the chain, *adding* nodes to the marked end from a pool of available nodes, *moving* the end marker closer to the beginning, and *deleting* a node from the marked end.

We chose this grammar for our experiments as it allowed us to create the different scenarios under which RETE is expected to perform better or worse than search plan. This was inspired by the benchmarking results in [GJR10], as well as the intuitive rule that RETE is expected to perform better when the extra work it does at finding all complete and partial matches at each state is preserved across several consecutive rule applications.

To better demonstrate the behavior of RETE, some of the runs in our experiment made use of a subset of the rules. The experiment was performed on a machine with a dual core 1.6GHz Intel CPU and 2.5GB of RAM. Table 1 summarizes the results of our experiment. The first column indicates which rules have been used. The ‘Len’ column specifies the length of the chain in the initial graph, and the ‘Pool’ column indicates the number of available nodes in the pool, if that run makes use of the ‘add’ rule. The ‘RETE’ and ‘SP’ columns contain the total exhaustive exploration time for RETE and search plan. In the case of search plan, the reported time is the average time taken using DFS and BFS exploration strategies. For RETE, we used the specially tailored strategy as discussed in [GJR10]. The right-most column in Table 1 indicates RETE’s performance gain or loss (if it is a negative number) compared to SP.

The numbers clearly show that in the absence of any rule that modifies the structure of the

#	Rules	Len	Pool	States / Transitions	RETE(ms)	SP(ms)	Prf. Gain(%)
1	move	27	-	2941 / 8042	1781	1984	10.2
2	move	54	-	19063 / 54439	18734	40437	53.7
3	move,mark	12	-	2048 / 8192	1016	2531	59.9
4	move,mark	15	-	16384 / 77824	7297	21758	66.5
5	move,add,del	2	5	23248 / 42615	6062	5750	-5.2
6	move,add,del	4	5	357605 / 634772	89719	87000	-3.0
7	all	2	5	179480 / 764760	90125	125375	28.1

Table 1: Experimental results for exhaustive state space exploration

chain (rows 1 to 4), the path matches found for next+ by RETE remain valid throughout the exploration, while SP is forced to follow and match the chain repeatedly after each rule application. When the rules 'del' and 'add' are present, RETE starts losing a lot of matches, or ends up creating more matches every time a next edge is deleted or created by those rules. This leads to RETE losing the race to SP (rows 5 and 6). This performance loss is easily compensated when the rule 'mark', which also does not modify the structure of the chain, is used along with the others (row 7), where a 5.2% loss is turned into a 28% performance gain. While the other experiments are meant to reveal the underlying mechanisms affecting performance, row 7 represents the typical and naturally ocurring scenario in which RETE can be used to achieve gains. There is however another characteristic that should be noted about the above scenarios: RETE's loss occurs at a more or less stable rate (rows 5 and 6) even when doubling the length of the chain has resulted in a 11 fold increase in the size of the state space. Rows 1-4 show that this is not the case when SP is losing, as increasing the length of the chain causes further declines in its performance compared to RETE.

6 Conclusion and Future Work

In this paper, we provided an extension to the RETE algorithm to support matching for regular path expressions. Finding paths in a graph that satisfy a given regular expression is particularly problematic as the matches for a regular expression with transitive closures are neither of a fixed size, nor is there always a finite number of them due to the presence of cycles. This gave rise to the question of what is actually meant by finding a match for a regular expression, which we addressed by formally defining paths and their matches. We also provided formal specifications of incremental updates to the finite set of matches found for each regular expression and showed that our formulation of the set of matches for a given regular expression gives rise to a finite and complete set, which makes it suitable for implementation.

Our experimental results showed that RETE can in fact provide performance gains in regular expression matching when the dynamics of the situation are such that RETE's 'find once, use many times' behavior makes it possible to avoid repetitive finding of long, mostly unaltered paths.

As future work, one possible direction for improvement would be to boost RETE's regular expression matching by enhancing its path checkers, especially the closure path checkers, with an *on-demand* capability, such that they would only produce matches when needed and only

those that are anchored at a certain node in the host graph. In our example grammar discussed in Section 5, the closure path checker does in fact find paths that lie in the middle of the chain, which are basically of little use, since all our operations happen at the end of the chain. This could lead to serious performance declines in finding matches for the expression a* in cases where the host graph is dense with *a*-labelled edges and there are frequent updates in the form of addition and removal of such edges.

Bibliography

[BGT91] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*. Pp. 174–189. Springer-Verlag, London, UK, 1991.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental pattern matching in the VIATRA model transformation system. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*. Pp. 25–32. ACM, NY, USA, 2008.

[DI06] C. Demetrescu, G. F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms* 4(3):353–383, 2006.

[For82] C. Forgy. RETE, a fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence* 19:17–37, 1982.

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GRGEN: A Fast SPO-Based Graph Rewriting Tool. In Corradini et al. (eds.), *International Conference on Graph Transformations (ICGT)*. LNCS 4178, pp. 383–397. Springer, 2006.

[GJR10] A. H. Ghamarian, A. Jalali, A. Rensink. Incremental Pattern Matching in Graph-Based State Space Exploration. *ECEASST* 32, 2010.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES approach: language and environment. In Ehrig et al. (eds.), *Handbook of graph grammars and computing by graph transformation: applications, languages, and tools*. Volume 2, pp. 487–550. World Scientific Publishing Co., Inc., 1999.

[VVF06] G. Varró, D. Varró, K. Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai and Taentzer (eds.), *GraMot 2005, International Workshop on Graph and Model Transformations*. ENTCS 152, p. 191–205. Elsevier, 2006.
http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/gramot05_vvf.pdf