A Rete Network Construction Algorithm for Incremental Pattern Matching

Gergely Varró* and Frederik Deckwerth**

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{gergely.varro,frederik.deckwerth}@es.tu-darmstadt.de

Abstract. Incremental graph pattern matching by Rete networks can be used in many industrial, model-driven development and network analysis scenarios including rule-based model transformation, on-the-fly consistency validation, or motif recognition. The runtime performance of such an incremental pattern matcher depends on the topology of the Rete network, which is built at compile time. In this paper, we propose a new, dynamic programming based algorithm to produce a high quality network topology according to a customizable cost function and a user-defined quantitative optimization target. Additionally, the Rete network construction algorithm is evaluated by using runtime measurements.

Keywords: incremental graph pattern matching, search plan generation algorithm, Rete network construction.

1 Introduction

The model-driven development and the network analysis domains both have industrial scenarios, such as (i) checking the application conditions in rule-based model transformation tools [1], or (ii) recognition of motifs [2,3] (i.e., subgraph structures) in social, financial, transportation or communication networks, which can be described as a general pattern matching problem.

In this context, a pattern consists of constraints, which place restrictions on variables. The pattern matching process determines a mapping of variables to the elements of the underlying model in such a way that the assigned model elements must fulfill all constraints. An assignment, which involves all the variables of a pattern, is collectively called a match.

When motif recognition, which aims at collecting statistics about the appearance of characteristic patterns (i.e., subgraph structures) to analyze and improve (e.g., communication) networks, is carried out by a pattern matching engine, two specialties can be identified which are challenging from an implementation aspect due to their significant impact on performance. On one hand, motifs frequently

---

and considerably *share subpatterns*, whose common handling can spare a substantial amount of memory. On the other hand, the motif searching process is invoked and executed *several times* on network graphs which are only *slightly altered* between two invocations. This observation opens up the possibility of using incremental pattern matchers which store matches in a cache, and update these matches incrementally in a change propagation process triggered by notifications about changes in the model (i.e., network graph).

Many sophisticated incremental pattern matchers [4,5,6] are implemented as Rete networks [7] which are directed acyclic graphs consisting of data processing nodes that are connected to each other by edges. Each node represents a (sub)pattern and stores the corresponding matches, while edges can send events about match set modifications. At compile time, the incremental pattern matcher builds a Rete network by using the pattern specifications. At runtime, each node continuously tracks the actual set of matches. When the network receives notifications about model changes, these modifications are processed by and propagated through the nodes. When the propagation is terminated, the network stores the matches for the patterns according to the altered model.

In the state-of-the-art Rete-based incremental pattern matching engines, the recognition of shared subpatterns, which can strongly influence the runtime memory consumption, is carried out at compile time during the construction of the Rete network by hard-wired algorithm implementations, whose design is based on the qualitative judgement of highly-qualified, experienced professionals. This approach hinders (i) the reengineering of the network builder module, (ii) the introduction of quantitative performance metrics, and (iii) the flexible selection of different optimization targets.

In this paper, we propose a new, dynamic programming based algorithm to construct a Rete network which has a high quality according to a customizable cost function and a user-defined quantitative optimization target. The algorithm automatically recognizes isomorphic subpatterns which can be represented by a single data processing node, and additionally, it favours those network topologies, in which a *large number* of these isomorphic subpatterns are handled *as early as possible*. Finally, the effects of the Rete network construction algorithm are quantitatively evaluated by using runtime measurements.

The remainder of the paper is structured as follows: Section 2 introduces basic modeling and pattern specification concepts. The incremental pattern matching process is described in Sec. 3, while Sec. 4 presents the new Rete network construction algorithm. Section 5 gives a quantitative performance assessment. Related approaches are discussed in Sec. 6, and Sec. 7 concludes our paper.

2   Metamodel, Model and Pattern Specification

2.1   Metamodels and Models

A *metamodel* represents the core concepts of a domain. In this paper, our approach is demonstrated on a real-world running example from the network analysis domain [2] whose metamodel is depicted in Fig. 1(a). *Classes* are the nodes

in the metamodel. Our example domain consists of a single class MotifNode.[1] *References* are the edges between classes which can be uni- or bidirectionally navigable as indicated by the arrows at the end points. A navigable end is labelled with a *role name* and a *multiplicity* which restricts the number of targets that can be reached via the given reference. In our example, a MotifNode can be connected to an arbitrary number of MotifNodes via bidirectional motifEdges.

Figure 1(b) depicts a *model* from the domain, whose nodes and edges are called *objects* and *links*, respectively. The model shows an instance consisting of three objects of type MotifNode connected by two links of type motifEdge.
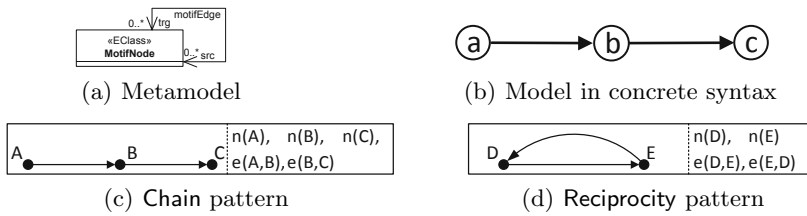

(a) Metamodel   (b) Model in concrete syntax

(c) Chain pattern   (d) Reciprocity pattern

Fig. 1. Metamodel, model and 2 patterns from the motif recognition scenario

2.2   Pattern Specification

A user of the pattern matcher specifies a set of patterns $\mathcal{P}$. As defined in [8,9], a *pattern* $P = (V_P, C_P, t_P, p_P)$ is a set of constraints C_P over a set of variables V_P. A *variable* $v \in V_P$ is a placeholder for an object in a model. A *constraint* $c \in C_P$ specifies a condition (of a *constraint type* $t_P(c)$) on a set of variables (which are also referred to as *parameters* in this context) that must be fulfilled by the objects which are assigned to the parameters. A pattern must be free of undeclared parameters and unused variables.

No undeclared parameters. The parameters of a constraint c must be variables from the set V_P, formally, $\forall c \in C_P, \forall i \leq ar(t_P(c)) : p_P(c, i) \in V_P$, where $p_P(c, i)$ denotes the ith parameter of constraint c and the inequality $i \leq ar(t_P(c))$ expresses that a constraint c of (constraint) type $t_P(c)$ has an arity $ar(t_P(c))$ number of parameters.

No unused variables. Each variable v must occur in at least one constraint as parameter, formally, $\forall v \in V_P, \exists c \in C_P, \exists i \leq ar(t_P(c)) : p_P(c, i) = v$.

Metamodel-Specific Constraint Types: Constraint type n maintains a reference to class MotifNode in the metamodel. Constraints of type n prescribe that their single parameter must be mapped to objects of type MotifNode. Constraint

---

[1] The intentionally simple metamodel enables a compact data structure representation throughout the paper, which was required due to space limitations. However, this choice yields at the same time to the algorithmically most challenging situation (due to the high complexity of isomorphism checks in „untyped" graphs).

type e refers to association motifEdge. Constraints of type e require a link of type motifEdge that connects the source and the target object assigned to the first and second parameter, respectively.

Example. Figures 1(c) and 1(d) show two sample patterns in visual and textual syntax. The Chain pattern (Fig. 1(c)) has 3 variables (A, B, C), 3 unary constraints of type n, and 2 binary constraints of type e. Constraints of type n and e are depicted by nodes and edges in graphical syntax, respectively. E.g., n(A) prescribes that objects assigned to variable A must be of class MotifNode.

Pattern related concepts. A *morphism* $m = (m_V, m_C)$ is a function on patterns which consists of a pair of functions m_V and m_C on variables and constraints, respectively. A morphism m is *constraint type preserving* if $\forall c \in C_P$: $t_{m(P)}(m_C(c)) = t_P(c)$; and *parameter preserving* if $\forall c \in C_P, \forall i \leq ar(t_P(c))$: $p_{m(P)}(m_C(c), i) = m_V(p_P(c, i))$.

Patterns P and P' are *isomorphic* (denoted by $\triangleq(P) = P'$) if there exists a constraint type and parameter preserving, bijective morphism $\triangleq$ from P to P'. The *join of patterns* P_l *and* P_r *on join variables* $v_{x_1}, \ldots, v_{x_q} \in V_{P_l}$, *and* $v_{y_1}, \ldots, v_{y_q} \in V_{P_r}$ is a pattern with $|V_{P_l}| + |V_{P_r}| - q$ variables and $|C_{P_l}| + |C_{P_r}|$ constraints which is produced by a morphism pair $\bowtie^l$ and $\bowtie^r$ as follows. Each corresponding pair (v_{x_z}, v_{y_z}) of the q join variables is mapped to a (shared) new variable v'_z (i.e., $\bowtie^l_V(v_{x_z}) = \bowtie^r_V(v_{y_z}) = v'_z$). Each non-join variable v_x and v_y of pattern P_l and P_r are mapped to a new variable v'_x and v'_y by $\bowtie^l$ and $\bowtie^r$, respectively. Formally, $\bowtie^l_V(v_x) = v'_x$ and $\bowtie^r_V(v_y) = v'_y$. A new constraint c'_l (c'_r) is assigned to each constraint c_l (c_r) from pattern P_l (P_r) by $\bowtie^l_C$ $(\bowtie^r_C)$ in a constraint type and parameter preserving manner.

A *subpattern* P' *of pattern* P consists of a subset of constraints of pattern P together with the variables occurring in the selected constraints as parameters. Two subpatterns P_1 and P_2 of a pattern P are *unifiable* if they have common variables. These common variables are referred to as *unifiable variables*. Two subpatterns of a pattern are *independent* if they do not share any constraints. The *union* of two independent subpatterns P_1 and P_2 of a pattern (denoted by $P_1 \cup P_2$) is produced by independently computing the union of the variables $(V_{P_1 \cup P_2} := V_{P_1} \cup V_{P_2})$ and the constraints $(C_{P_1 \cup P_2} := C_{P_1} \cup C_{P_2})$ of the two subpatterns and using *identity* morphisms id^l and id^r which map P_1 to $P_1 \cup P_2$ and P_2 to $P_1 \cup P_2$, respectively, in a constraint type and parameter preserving manner. A set of subpatterns of a pattern constitutes a *partition* if they are pairwise independent, and their union produces the pattern itself. In the following, the subpatterns of a pattern constituting a partition are called *components*.

Note that union is performed on components of a given pattern, and results in another component of the same pattern which will replace the operands in the partition. In contrast, a join operates on arbitrary patterns, and yields to a new pattern which is unrelated to the operand patterns. In the context of a join operation, each of the operands and the result pattern has its own variable set.

Example. Figure 2(b) is used to exemplify the concepts of this section. Nodes with s labels in the center (on white background) represent patterns. Each pattern

(a) Initial state stored in $T[8][1]$ (b) State inserted into $T[4][1]$
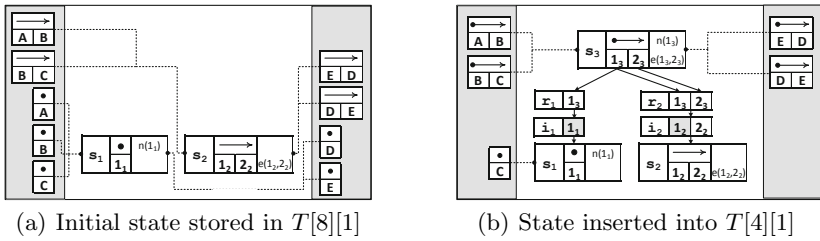
Fig. 2. Illustration of pattern related concepts and the algorithm execution ($k = 1$)

has its own, distinguished set of variables which are marked by indexed integers. The pattern in s_3 is the join of the patterns in s_1 and s_2 on join variables 1_1 and 1_2. In this case, function $\bowtie_V^l$ maps variable 1_1 to 1_3, while $\bowtie_V^r$ assigns variables 1_3 and 2_3 to 1_2 and 2_2, respectively. Constraints $n(1_1)$ and $e(1_2, 2_2)$ are mapped by $\bowtie_C^l$ and $\bowtie_C^r$ to $n(1_3)$ and $e(1_3, 2_3)$, respectively. The patterns on the left side (with grey background) show the components of the Chain (Fig. 1(c)) pattern which share variables labelled by capital letters with the latter pattern. The union of these components can be computed along the (unifiable) variables with the same name resulting in the Chain pattern. The components of the Reciprocity (Fig. 1(d)) pattern are shown on the right side.

3   Incremental Pattern Matching Process

As [9] states, *pattern matching* is the process of determining mappings for all variables in a given pattern, such that all constraints in the pattern are fulfilled. The mappings of variables to objects are collectively called a *match* which can be a *complete match* when all the variables are mapped, or a *partial match* in all other cases.[2] The overall process of *incremental pattern matching* is as follows:

Compile time tasks. At compile time, a Rete network [7], whose structure is presented in Sec. 3.1, is built from the pattern specifications by a network construction algorithm which will be discussed in details in Sec. 4.

Runtime behaviour. At runtime, the Rete network continuously tracks (i) the complete matches for all patterns in the underlying model and (ii) those partial matches that are needed for the calculation of the complete matches. These matches are stored in the Rete network and incrementally updated in a change propagation process which is triggered by notifications about model changes as presented in Sec. 3.2.

3.1   Rete Network

A *Rete network* is a directed acyclic graph whose nodes are data processing units which are organized into a parent-child relationship by the edges (considering the

---

[2] A match maps only pattern variables to model objects, while a morphism maps variables *and* constraints of a pattern to their counterparts in another pattern.

traditional source-to-target direction). The nodes are partitioned into skeletons $\mathcal{S}$, indexers $\mathcal{I}$, and remappers $\mathcal{R}$. The connections expressed by the edges are also restricted, because skeletons, remappers, and indexers can only be connected to remappers, indexers, and skeletons, respectively.

A *skeleton* calculates matches for a pattern in the Rete network. A *basic skeleton*, which corresponds to a pattern with a *single* constraint, has no outgoing edges. A *joined skeleton* is connected in the Rete network by edges to its left r_l and right r_r child remappers, and it represents a pattern with *several* constraints which is assembled from 2 smaller patterns, whose (great-grandchild) skeletons can be reached in the Rete network via paths (of length 3) along the left and right child remappers of the joined skeleton, respectively.

A *remapper* maintains an array-based mapping from the variables of its grandchild skeleton to the variables of its parent joined skeleton to support the match computation performed in the latter node.

An *indexer* stores the matches produced by its child skeleton in a table. Each field of this table contains the mapping of a variable (represented by a column) to an object according to the match (symbolized by a row). The matches are sorted according to the values that were assigned to a subsequence of variables (the so-called *indexed variables*) of the child skeleton. The skeleton and its indexed variables uniquely identify the corresponding indexer in the Rete network.

Example. Figure 3 depicts two sample Rete networks, which track the matches of the patterns of Figs. 1(c) and 1(d) on the model of Fig. 1(b). The identifiers of skeletons s, indexers i and remappers r are marked in the (leftmost) rectangles in the node headers. The pattern represented by a skeleton is shown in the header as well. In Fig. 3(b), basic skeleton s_1 corresponds to the pattern which has a single unary constraint of type n on parameter 1_1. This skeleton produces matches for the Rete network which map variable 1_1 to all MotifNodes from the model. These matches are stored sorted according to the values assigned to indexed variable 1_1 (shown by the grey column) in indexer i_1. MotifEdges are entered into the Rete network in skeleton s_2 and stored in indexer i_2. This indexer sorts the motifEdges according to their source objects, as only variable 1_2 is indexed. Joined skeleton s_3 carries out a join of patterns in skeletons s_1 and s_2 on join variables 1_1 and 1_2. To perform this operation, (i) join variables 1_1 and 1_2 have to be indexed in the grandchild indexers i_1 and i_2, respectively, (ii) variable 1_1 of skeleton s_1 has to be remapped by (left child) remapper r_1 according to $\bowtie^l$ to variable 1_3 of skeleton s_3, and similarly (iii) variables 1_2 and 2_2 must be remapped by (right child) remapper r_2 according to $\bowtie^r$ to variables 1_3 and 2_3, respectively. Joined skeleton s_4 joins patterns in skeletons s_1 and s_3 on join variables 1_1 and 2_3. Note that this join operation only involves variable 2_3 from skeleton s_3, consequently, indexer i_3 must only index this variable. Skeletons s_5 and s_6 represent patterns which are isomorphic to the Chain and the Reciprocity pattern, respectively. As a consequence, the matches produced by skeleton s_5 are the complete matches for the Chain pattern (in the left grey framed table), while skeleton s_6 creates no complete matches for the Reciprocity pattern. Note that skeleton s_6 joins the pattern in skeleton s_3 via two distinct paths by using join variables 1_3 and 2_3 in

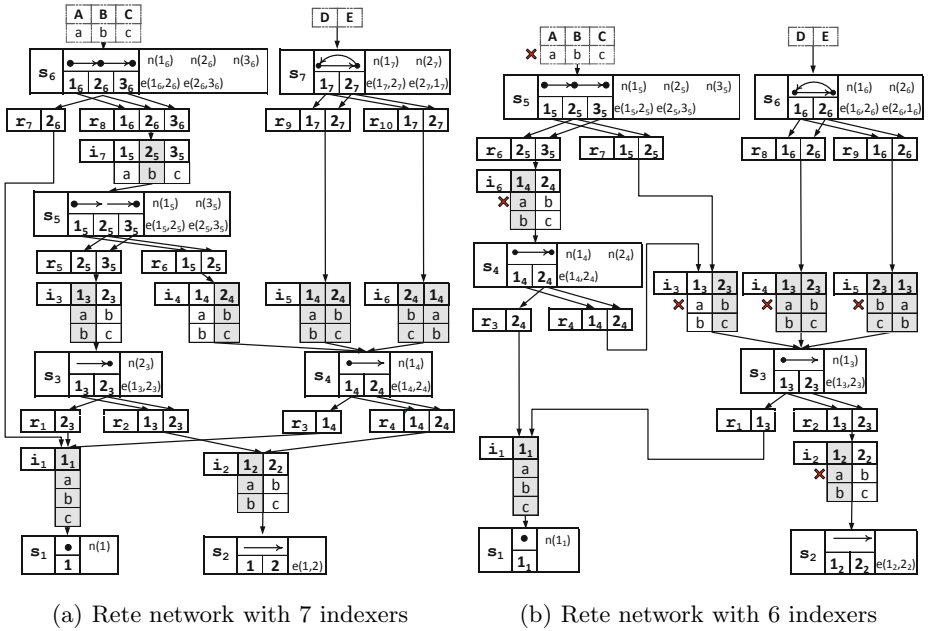(a) Rete network with 7 indexers   (b) Rete network with 6 indexers

Fig. 3. Sample Rete networks

the left branch, and 2_3 and 1_3 in the right branch. As the left and right paths both involve 2 join variables, indexers i_4 and i_5 must use both join variables 1_3 and 2_3 for indexing (however, in a different order).

3.2   Incremental Pattern Matching at Runtime with Rete Network

To demonstrate the runtime behaviour of a Rete network in an incremental setting, let us suppose that the Rete network is already filled with matches computed from the initial content of the underlying model. More specifically, (i) indexers store the (partial or complete) matches calculated by their child skeleton, (ii) basic skeletons provide access for the Rete network to the model, and (iii) the top-most joined skeletons (i.e., without skeleton ancestors) already produced the complete matches for the corresponding patterns.

When the underlying model is altered, the Rete network is notified about this model change. This notification triggers a bottom-up change propagation process, which passes *match deltas* (i.e., representing match additions or deletions) from basic skeletons towards the top-most joined skeletons. As a common behaviour in this process, each node carries out 3 steps, namely, it (i) receives a match delta from one of its child nodes as input, (ii) performs data processing which might result in new match deltas as output, and (iii) optionally propagates all the output match deltas to all of its parent nodes.

Example. If the link between objects a and b is removed from the model of Fig. 1(b), then the matches marked by (red) crosses in Fig. 3(b) are deleted from

the indexers of the Rete network in a bottom-up change propagation process starting at basic skeleton s_2 and terminating at joined skeletons s_5 and s_6.

4   Dynamic Programming Based Network Construction

As demonstrated in Fig. 3, the number of indexers has an obvious and significant influence on the runtime memory usage of the Rete network. As a consequence, our network construction algorithm uses this parameter as an optimization target to quantitatively characterize Rete network topologies.

A Rete network with few indexers is built by a dynamic programming based algorithm which iteratively fills states into an initially empty table T with $n+1$ columns and k rows, where n is a value derived from the initial state and $k \geq 1$ is a user-defined parameter that influences the trade-off between efficiency and optimality of the algorithm. A *state* represents a partially constructed Rete network, whose quality is defined by an *arbitrary* cost function. A state is additionally characterized by a *unification point (UP) indicator* which is the "distance" of the partial Rete network from a final topology that must symbolize all patterns in the specification. In table T, the column $T[col]$ stores the best k states (in an increasing cost order), whose UP indicator is *col*, while $T[col][row]$ is the *row*th best from these states.

The main distinguishing feature of the algorithm is that the table *only stores a constant number* of states *in each column*, immediately discarding costly network topologies, which are not among the best k solutions, and implicitly all their possible continuations. The algorithm itself shares its core idea (and its two outermost loops) with the technique presented in [10] which was used for generating search plans for batch pattern matchers, but the current approach uses *completely different data structures* in the optimization process.

Algorithm data structures. A *state* S contains a Rete network RN_S, sets of components Comp_S and skeleton patterns Skel_S, and an isomorphism function iso_S. Each pattern P in the specification will be represented in the component set Comp_S of state S by a partition of its subpatterns which are called *components of pattern P in state S* (denoted by Comp_S^P) in the following. The component set Comp_S is the collection of all components of all patterns in state S. A *skeleton pattern P_s* corresponds to skeleton s in the Rete network RN_S, and it represents a set of isomorphic components which are mapped to skeleton pattern P_s by the *isomorphism function* iso_S. The skeleton patterns that have a corresponding skeleton in network RN_S are contained in set Skel_S. The cost c_S of a state S can be *arbitrarily* defined. In this paper, the number of indexers $|\mathcal{I}_{RN_S}|$ in the Rete network RN_S is used as a cost function.

Unification points. A *unification point (UP) on variable v* is a situation, when variable v is unifiable by a pair of components of a pattern P in a state S. To compactly characterize the number of UPs on variable v, a *unification point indicator upi_S^v for variable v* is introduced as the number of those components of pattern P in state S which contain variable v. The *unification point indicator*

upi_S *of a state* S is calculated as $\sum_{P \in \mathcal{P}} \sum_{v \in V_P}(upi_S^v - 1)$. The subtraction is only required to be able to evaluate the term $\sum_{v \in V_P}(upi_S^v - 1)$ to 0, if and only if each variable of pattern P appears in a single component from the set Comp_S.

Example. Figure 2 depicts two states from the Rete network construction process. The tables on the left and right sides of each state (on the area with grey background) represent the components, whose union always results in the Chain and Reciprocity patterns, respectively. These components are mapped by the isomorphism function iso_S (denoted by the dashed lines) to the (jointly depicted) skeleton patterns and Rete network in the middle. Note that a skeleton pattern always unambiguously corresponds to a skeleton. The two states have 0 and 2 indexers, respectively, which are used as costs of the states. In Fig. 2(a), the UP indicators for variables B, D, E are 3, as each of these variables appears in 3 components, while the UP indicators for variables A and C are 2. The UP indicator of the state itself is $3 \cdot (3 - 1) + 2 \cdot (2 - 1) = 8$.

Initialization. Each pattern P in the specification is split into components $\mathsf{C}_1^P, \ldots, \mathsf{C}_{|C_P|}^P$ with *single* constraints which trivially constitute a partition of pattern P. Components $\mathsf{C}_1^P, \ldots, \mathsf{C}_{|C_P|}^P$ of each pattern P are added to the set Comp_{S_0}. For each constraint type t appearing in any of the patterns, a skeleton s_t and a corresponding skeleton pattern P_{s_t} are added to the Rete network RN_{S_0} and skeleton pattern set Skel_{S_0}, respectively. The skeleton pattern P_{s_t} has $ar(t)$ new variables and one constraint of type t with the newly created variables as parameters. In this way, all components C consisting of a single constraint of type t, which are obviously isomorphic, can be represented by skeleton pattern P_{s_t} which is registered into the isomorphism function as $\mathsf{iso}_{S_0}(\mathsf{C}) = P_{s_t}$.

Algorithm. Algorithm 1 determines the UP indicator upi_{S_0} of the initial state S_0 (line 1), and stores this state S_0 in $T[n][1]$ (line 2). Then, the table is traversed by processing columns in a decreasing order (lines 3–11). In contrast, the inner loop (lines 4–10) proceeds in an increasing state cost order starting from the best state $T[col][1]$ in each column $T[col]$. For each stored state S, the possible extensions Δ_{Skel} of the skeleton pattern set Skel_S are determined by `calculateDeltas` (line 6) which are used by `calculateNextStates` (line 7) to produce all continuations of state S. Each next state S' (lines 7–9) is conditionally inserted into the column $T[upi_{S'}]$ identified by the corresponding UP indicator $upi_{S'}$ in the procedure `conditionalInsert` (line 8) if the next state S' is among the k best states in the column $T[upi_{S'}]$. When the three loops terminate, the algorithm returns the Rete network $RN_{T[0][1]}$ (line 12).

The basic idea when producing all continuations of a state S (lines 6–7) is that unifiable components are aimed to be replaced by their union. As (i) isomorphic components are represented by a single skeleton pattern in state S (and a corresponding skeleton in the Rete network RN_S), and (ii) the union of components can be expressed by a new skeleton pattern, which is the join of the skeleton patterns of the unifiable components, a single join operation can also characterize the unification of numerous component pairs from the set Comp_S.

Algorithm 1. The procedure `calculateReteNetwork`(S_0, k)

```
 1: n := upi_{S_0}
 2: T[n][1] := S_0
 3: for (col := n to 1) do
 4:     for (row := 1 to k) do
 5:         S := T[col][row] // current state S
 6:         Δ_Skel := calculateDeltas(S)
 7:         for each (S' ∈ calculateNextStates(S, Δ_Skel)) do
 8:             conditionalInsert(T[upi_{S'}], S')
 9:         end for
10:     end for
11: end for
12: return  RN_{T[0][1]}
```

In order to support effective subpattern sharing in the Rete network, *a single join should represent as many unifications as possible.* This can only be achieved if the complete set of applicable joins and their corresponding unifications are determined in advance, and the actual computation of next states is delayed.

Section 4.1. The procedure `calculateDeltas`(S) iterates through all unifiable components of all patterns in state S, and for each unification, a corresponding join is determined in such a manner that the union of the components is isomorphic to the result of the join. In other words, the set of applicable joins (i.e., the skeleton deltas in Sec. 4.1) is calculated together with a grouping of unifications (i.e., the component deltas in Sec. 4.1), in which each group contains those unifications that can be characterized by a single join.

Section 4.2. The procedure `calculateNextStates`$(S, \Delta_{\mathsf{Skel}})$ iterates through all applicable joins, and for each corresponding group, all those independent subsets are calculated which do not share any unifications. The unifications in these subsets can be used for preparing the next states.

The procedure `conditionalInsert`$(T[upi_{S'}], S')$ calculates index $\mathbf{c}$ which marks the position at which state S' should be inserted based on its cost. Index $\mathbf{c}$ is set to $k + 1$ if state S' is not among the best k states. Formally, $\mathbf{c}$ is the smallest index for which $c_{S'} < c_{T[upi_{S'}][\mathbf{c}]}$ holds (or $T[upi_{S'}][\mathbf{c}] = \texttt{null}$). If $\mathbf{c} < k + 1$, then state $T[upi_{S'}][k]$ is removed, elements between $T[upi_{S'}][\mathbf{c}]$ and $T[upi_{S'}][k - 1]$ are shifted downward, and state S' is inserted at position $\mathbf{c}$.

Example. Due to space limitations, Fig. 2 can only exemplify an incomplete, single iteration of the algorithm execution. The initial state (Fig. 2(a)) has a UP indicator 8. Consequently, table T (not shown in Fig. 2) has 8 columns, and the initial state is inserted into $T[8][1]$. When this state is processed by the procedure `calculateDeltas`(S), all unifiable component pairs are evaluated. During this evaluation, it is determined that e.g., (J1) if skeletons s_1 and s_2 are joined on variables 1_1 and 1_2 (see s_3 in Fig. 2(b)), then this join alone represents the unification of the component pairs (i) $\mathsf{n}(\mathsf{A}), \mathsf{e}(\mathsf{A}, \mathsf{B})$; (ii) $\mathsf{n}(\mathsf{B}), \mathsf{e}(\mathsf{B}, \mathsf{C})$;

(iii) $n(D), e(D, E)$; and (iv) $n(E), e(E, D)$. Three additional join possibilities (not shown in Fig. 2) are identified in the same stage, namely, (J2) skeletons s_1 and s_2 can be joined on variables 1_1 and 2_2 as well (resulting in a node with an incoming edge). Skeleton s_2 can be joined to itself (J3) either on variable sequences $1_2, 2_2$ and $2_2, 1_2$ (forming a cycle from the two edges), (J4) or on variables 2_2 and 1_2 (providing a chain from the two edges). The procedure `calculateDeltas`(S) computes the information exemplified on case (J1) for all the 4 joins, which is passed as Δ_{Skel} to the procedure `calculateNextStates`$(S, \Delta_{\mathsf{Skel}})$ in line 7 for further processing. The 4 unifiable component pairs of case (J1) have no constraints in common, consequently, these four unifications and the corresponding join can be directly used to build a next state (Fig. 2(b)), in which skeleton s_3 alone represents the 4 (isomorphic) components on the sides. Three additional next states are constructed for cases (J2)–(J4) as well. The next states prepared for cases (J1), (J3), and (J4) are inserted into empty slots $T[4][1]$, $T[6][1]$, and $T[7][1]$, respectively, according to their UP indicators, while the state created for case (J2) (again with UP indicator 4) is discarded (in line 8), as the state of Fig. 2(b) stored already in slot $T[4][1]$ has less indexers. When the three loops terminate, Alg. 1 returns the Rete network of Fig. 3(b) from the field $T[0][1]$.

4.1   Skeleton Pattern Delta Calculation

The procedure `calculateDeltas`(S) uses skeleton deltas and component deltas as new data structures to represent applicable, but delayed joins and unions, respectively. A *skeleton delta* consists of a set of component deltas $\Delta_{s'}$, a skeleton pattern $P_{s'}$ and a Rete network $RN_{s'}$. A *component delta* in the set $\Delta_{s'}$ contains two components C_l and C_r, and an isomorphism $\triangleq$ which maps the union $C_l \cup C_r$ of the components to the skeleton pattern $P_{s'}$.

The procedure `calculateDeltas`(S) (Algorithm 2) iterates through each pair C_l^P, C_r^P of unifiable components of pattern P in state S (lines 2–3). For each such pair, the method `createSkeletonPattern` (line 5) prepares a skeleton pattern $P_{s'}$ and an isomorphism $\triangleq$, such that $\triangleq$ maps the union of the components C_l^P and C_r^P to the skeleton pattern $P_{s'}$ (i.e., $\triangleq(C_l^P \cup C_r^P) = P_{s'}$). If the skeleton pattern $P_{s'}$ is already represented in the set Δ_{Skel} by another skeleton pattern P_{s*}, which is isomorphic to $P_{s'}$ according to an other morphism $\triangleq^*$ (line 6), then the component delta $(C_l^P, C_r^P, \triangleq \circ \triangleq^*)$ is simply added to the already stored set Δ_{s*} (line 7), as $C_l^P \cup C_r^P$ is isomorphic to skeleton pattern P_{s*} as well. Otherwise, a new Rete network $RN_{s'}$ is created by `createReteNetwork` (line 9), a new singleton set $\Delta_{s'}$ is prepared with the component delta $(C_l^P, C_r^P, \triangleq)$ (line 10), and the skeleton delta $(\Delta_{s'}, P_{s'}, RN_{s'})$ is added to the set Δ_{Skel} (line 11).

To describe the procedure `createSkeletonPattern`, let us suppose that components C_l^P and C_r^P are mapped by function isos_S to skeleton patterns P_{s_l} and P_{s_r}, respectively. Consequently, there exists an isomorphism $\triangleq^l$ ($\triangleq^r$) from component C_l^P (C_r^P) to skeleton pattern P_{s_l} (P_{s_r}). The new skeleton pattern $P_{s'}$ is the join of skeleton patterns P_{s_l} and P_{s_r} (by using $\bowtie^l$ and $\bowtie^r$), where the join variables in skeleton pattern P_{s_l} (P_{s_r}) are the images of the unifiable variables of components C_l^P and C_r^P according to isomorphism $\triangleq^l$ ($\triangleq^r$). The new

---

Algorithm 2. The procedure `calculateDeltas(S)`

1: $\Delta_{\mathsf{Skel}} := \emptyset$
2: **for each** $(P \in \mathcal{P})$ **do**
3:   **for each** $(\mathsf{C}_l^P, \mathsf{C}_r^P \in \mathsf{Comp}_S^P)$ **do**
4:     **if** $(\mathsf{C}_l^P \neq \mathsf{C}_r^P \wedge \mathtt{areUnifiable}(\mathsf{C}_l^P, \mathsf{C}_r^P))$ **then**
5:       $(P_{s'}, \triangleq) := \mathtt{createSkeletonPattern}(\mathsf{iso}_S, \mathsf{C}_l^P, \mathsf{C}_r^P)$
6:       **if** $(\exists(\Delta_{s^*}, P_{s^*}, RN_{s^*}) \in \Delta_{\mathsf{Skel}}, \exists \triangleq^* : \triangleq^*(P_{s'}) = P_{s^*})$ **then**
7:         $\Delta_{s^*} := \Delta_{s^*} \cup \left\{ (\mathsf{C}_l^P, \mathsf{C}_r^P, \triangleq \circ \triangleq^*) \right\}$
8:       **else**
9:         $RN_{s'} := \mathtt{createReteNetwork}(RN_S, \mathsf{iso}_S, \mathsf{C}_l^P, \mathsf{C}_r^P)$
10:        $\Delta_{s'} := \left\{ (\mathsf{C}_l^P, \mathsf{C}_r^P, \triangleq) \right\}$
11:        $\Delta_{\mathsf{Skel}} := \Delta_{\mathsf{Skel}} \cup \{ (\Delta_{s'}, P_{s'}, RN_{s'}) \}$
12:      **end if**
13:    **end if**
14:   **end for**
15: **end for**
16: **return** Δ_{Skel}

---

isomorphism $\triangleq$ can be defined as a composition of morphisms $\bowtie^{l,r}$ and $\triangleq^{l,r}$, namely, $\forall v \in V_{\mathsf{C}_l^P} : \triangleq_V(v) := \bowtie_V^l(\triangleq_V^l(v))$, $\forall c \in C_{\mathsf{C}_l^P} : \triangleq_C(c) := \bowtie_C^l(\triangleq_C^l(c))$, $\forall v \in V_{\mathsf{C}_r^P} : \triangleq_V(v) := \bowtie_V^r(\triangleq_V^r(v))$, and $\forall c \in C_{\mathsf{C}_r^P} : \triangleq_C(c) := \bowtie_C^r(\triangleq_C^r(c))$.

The procedure `createReteNetwork` creates a new Rete network $RN_{s'}$ by adding a new skeleton s' and its left r_l and right r_r remappers (plus the corresponding edges) to the old network RN_S. Indexer i_l (i_r) is either reused from RN_S if RN_S already contained it as a parent of skeleton s_l (s_r), or newly created. The edges between these indexers and skeletons are handled analogously. As the exact internal parameterization of network nodes is easily derivable from morphisms $\bowtie^l$, $\bowtie^r$, $\triangleq^l$, and $\triangleq^r$, it is not discussed here due to space limitations.

4.2   Next State Calculation

The procedure `calculateNextStates(S, \Delta_{\mathsf{Skel}})` (Algorithm 3) iterates through all skeleton deltas $(\Delta_{s'}, P_{s'}, RN_{s'})$ in the set Δ_{Skel} (line 2). In order to clarify the role of the inner loop (lines 3–8), let us examine its body (lines 4–7) first. The new Rete network $RN_{S'}$ simply uses the network $RN_{s'}$ from the skeleton delta (line 4). The skeleton pattern $P_{s'}$ is added to the skeleton pattern set Skel_S of state S to produce the new one (line 5). The procedure `calculateComponents` (line 6) creates a new component set $\mathsf{Comp}_{S'}$ from the old one Comp_S by replacing the components C_l and C_r of each component delta $(\mathsf{C}_l, \mathsf{C}_r, \triangleq)$ from the set $\Delta_{s'}^I$ with their union $\mathsf{C}_l \cup \mathsf{C}_r$. The new isomorphism function $\mathsf{iso}_{S'}$ retains the mappings of those components from the old one iso_S that do not appear in any component deltas from the set $\Delta_{s'}^I$, while the union $\mathsf{C}_l \cup \mathsf{C}_r$ of component pairs mentioned in a component delta $(\mathsf{C}_l, \mathsf{C}_r, \triangleq)$ is mapped to skeleton pattern $P_{s'}$ (i.e., $\mathsf{iso}_{S'}(\mathsf{C}_l \cup \mathsf{C}_r) = P_{s'}$). A new state $S' = (RN_{S'}, \mathsf{Skel}_{S'}, \mathsf{Comp}_{S'}, \mathsf{iso}_{S'})$ is added to the set Δ_S representing the possible continuations of state S (line 7).

Algorithm 3. The procedure `calculateNextStates`$(S, \Delta_{\mathsf{Skel}})$

1: $\Delta_S := \emptyset$
2: **for each** $((\Delta_{s'}, P_{s'}, RN_{s'}) \in \Delta_{\mathsf{Skel}})$ **do**
3: **for each** $\left(\Delta_{s'}^I \in \texttt{allMaximalIndependentSets}\,(\Delta_{s'})\right)$ **do**
4:    $RN_{S'} := RN_{s'}$
5:    $\mathsf{Skel}_{S'} := \mathsf{Skel}_S \cup \{\, P_{s'} \,\}$
6:    $(\mathsf{Comp}_{S'}, \mathsf{iso}_{S'}) := \texttt{calculateComponents}(S, \Delta_{s'}^I)$
7:    $\Delta_S := \Delta_S \cup \{\, (RN_{S'}, \mathsf{Skel}_{S'}, \mathsf{Comp}_{S'}, \mathsf{iso}_{S'}) \,\}$
8: **end for**
9: **end for**
10: **return** Δ_S

As the set $\mathsf{Comp}_{S'}$ must also contain *independent* components, the replacement in line 6 is only allowed if all component delta pairs $(\mathsf{C}_l^{P_\alpha}, \mathsf{C}_r^{P_\alpha}, \triangleq^{P_\alpha})$ and $(\mathsf{C}_l^{P_\beta}, \mathsf{C}_r^{P_\beta}, \triangleq^{P_\beta})$ from the set $\Delta_{s'}^I$ are independent, which means that they either originate from different patterns (i.e., $P_\alpha \neq P_\beta$), or they do not share any components (i.e., $\mathsf{C}_{l,r}^{P_\alpha} \neq \mathsf{C}_{l,r}^{P_\beta}$). As pairwise independence does not necessarily hold for the component deltas in set $\Delta_{s'}$, the method `allMaximalIndependentSets` carries out the Bron-Kerbosch algorithm [11] (line 3), and calculates all such subsets of $\Delta_{s'}$, whose (component delta) elements are pairwise independent.

5   Measurement Results

In this section, we quantitatively assess the effect of subpattern sharing on the number of indexers by comparing the case when our algorithm builds a *separate* Rete network for each pattern with the situation when isomorphic subpatterns are represented by shared skeletons (i.e., *combined* approach). For the evaluation, we used the patterns from [2], and the algorithm parameter k was set to 1.

The measurement results are presented in Table 1. A column header has to be interpreted in a *cumulative* manner including all patterns which appear in the headers of the *current and all the preceding* columns. A value in the first row shows the *sum* of the number of indexers[3] in those Rete networks that have been *separately* built for the patterns in the (current and its preceding) column headers. In contrast, a value in the second row presents the number of indexers[3] in the *single* Rete network that has been constructed by the *combined* approach which used the patterns in the (current and its preceding) column headers as input. The values in the third row express the memory reduction as the ratio of the values in the first two rows. Rows four and five denote the Rete network construction runtimes[4] for the separate and combined approach, respectively, while the sixth row depicts the ratio of the values from the previous two rows.

---

[3] The parent indexers of the basic skeletons were not included in either case, as their functionality (e.g., navigation on edges) is provided by the underlying modeling layer.

[4] The runtime values are averages of 10 user time measurements performed on a 1.57 GHz Intel Core2 Duo CPU with Windows XP Professional SP 3 and Java 1.7.

Table 1. Measurement results

	Pattern	FeedForward	FeedBack	Caro	DoubleCross	InStar	OutStar	Reciprocity
Indexers [#]	Separate	4	8	13	17	21	25	27
	Combined	4	5	7	11	14	20	21
Ratio	Combined / Separate	1.00	0.63	0.54	0.65	0.67	0.80	0.78
Runtime [ms]	Separate	12.500	14.063	23.438	28.126	79.689	134.377	134.377
	Combined	10.938	21.875	56.250	106.250	428.125	770.313	843.750
Ratio	Combined / Separate	0.88	1.56	2.40	3.78	5.37	5.73	6.28

The most important conclusion from Table 1 is that the combined approach uses 20–46% less indexers than the separate approach for the price of an increase in the algorithm runtime by a factor of 1–6 which is not surprising as the combined approach has to operate on tables that are wider by approximately the same factor. For a correct interpretation, it should be noted that the number of indexers influences the memory consumption at runtime, while the algorithm is executed only once at compile time.

6   Related Work

Motif recognition algorithms. The state-of-the-art motif recognition algorithms are excellently surveyed in [12]. These are batch techniques which match all non-isomorphic (graph) patterns up to a certain size, in contrast to our incremental approach, which builds a Rete network only for the (more general, constraint-based) patterns in the specification (and for a small part of their subpatterns). In the rest of this section, which is still knowingly incomplete, only Rete network based incremental approaches are mentioned.

Rete network construction in rule-based systems. As Rete networks were used first in rule-based systems, different network topologies have been analyzed in many papers from the artificial intelligence domain including [13], which recognized that linear structures can be replaced by (balanced) tree-based ones. However, this report provided neither cost functions to characterize the quality of a Rete network, nor algorithms to find good topologies.

A graph based Rete network description was proposed in [14] together with cost functions that could be used as optimization targets in a network construction process. Furthermore, the author gives conditions for network optimality according to the different cost metrics, in contrast to our dynamic programming based approach, which could only produce provenly optimal solution if the number of rows was not limited by the constant parameter k. On the other hand, no network construction algorithm is discussed in [14].

Rete network construction in incremental pattern matchers. Incremental graph pattern matching with Rete networks [7] was examined decades ago in [4] which already described an advanced network compilation algorithm (beyond the presentation of the runtime behaviour of the Rete network). This approach processed pattern specifications one-by-one, and it was able to reuse network nodes in a restricted manner, namely, if a subpattern was isomorphic to another one from a previous pattern, for which a network node had *actually* been

generated earlier in the construction procedure. In this sense, the recognition of isomorphic parts in two patterns depends on the order, in which the subpatterns of the first pattern had been processed. However, [4] gives no hint how such an order can be found.

Another sophisticated, Rete network based incremental graph pattern matching engine [6] has recently been used for state space exploration purposes in graph transformation systems. In this setup, the standard Rete approach was extended by graph transformation related concepts such as quantifiers, nested conditions, and negative application conditions. Additionally, disconnected graph patterns could also be handled. Regarding the Rete network construction, [6] uses the same technique as [4] with all its strengths and flaws.

IncQuery [5,15] is also a high quality pattern matcher that uses Rete networks for incremental query evaluation. Queries can be defined by graph patterns which can be reused and composed in a highly flexible manner. If isomorphic subpatterns are identified as standalone patterns, then they can be handled by a single node which can be reused by different compositions leading to the original patterns, but the *automated identification of isomorphic subpatterns is not yet supported* in contrast to our approach. As another difference, the *constructed Rete network has always a linear topology* in IncQuery, while our algorithm can produce a balanced net structure as well. Considering the Chain and the Reciprocity patterns, the Rete network of Fig. 3(b) can only be constructed in IncQuery if the user *manually* specifies skeletons s_3 and s_4 as patterns and the complete network structure by pattern compositions.

7   Conclusion

In this paper, we proposed a novel algorithm based on dynamic programming to construct Rete networks for incremental graph pattern matching purposes. The cost function and the optimization target used by the algorithm can be easily replaced and customized. As the basic idea of the proposed algorithm is similar to the technique presented in [10] for batch pattern matching, our *fully implemented* network building approach can be easily integrated into the search plan generation module of the Democles tool which will be able to handle batch and incremental scenarios in an integrated manner.

As an evaluation from the aspect of applicability, the proposed algorithm can (i) use model-sensitive costs (originating from model statistics), (ii) handle n-ary constraints in pattern specifications, and (iii) be further customized by setting parameter k which influences the trade-off between efficiency and optimality.

The most important future task is to assess the effects of network topologies on the runtime performance characteristics of the pattern matcher in industrial application scenarios by using different cost functions and optimization targets in the proposed network construction algorithm.

References

1. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
2. von Landesberger, T., Görner, M., Rehner, R., Schreck, T.: A system for interactive visual analysis of large graphs using motifs in graph editing and aggregation. In: Magnor, M.A., Rosenhahn, B., Theisel, H. (eds.) Proceedings of the Vision, Modeling, and Visualization Workshop, DNB, pp. 331–339 (2009)
3. Krumov, L., Schweizer, I., Bradler, D., Strufe, T.: Leveraging network motifs for the adaptation of structured peer-to-peer-networks. In: IEEE Proceedings of the Global Communications Conference, pp. 1–5 (2010)
4. Bunke, H., Glauser, T., Tran, T.-H.: An efficient implementation of graph grammar based on the RETE-matching algorithm. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 174–189. Springer, Heidelberg (1991)
5. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Proc. of the 3rd Int. Workshop on Graph and Model Transformation, pp. 25–32. ACM (2008)
6. Ghamarian, A.H., Jalali, A., Rensink, A.: Incremental pattern matching in graph-based state space exploration. In: de Lara, J., Varró, D. (eds.) Proc. of the 4th International Workshop on Graph-Based Tools. ECEASST, vol. 32 (2010)
7. Forgy, C.L.: RETE: A fast algorithm for the many pattern/many object match problem. Artificial Intelligence 19, 17–37 (1982)
8. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Workshop on Graph Transformation and Visual Modeling Techniques, vol. 6, ECEASST (2007)
9. Varró, G., Anjorin, A., Schürr, A.: Unification of compiled and interpreter-based pattern matching techniques. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 368–383. Springer, Heidelberg (2012)
10. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An algorithm for generating model-sensitive search plans for EMF models. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 224–239. Springer, Heidelberg (2012)
11. Bron, C., Kerbosch, J.: Algorithm 457: Finding all cliques of an undirected graph. Communications of the ACM 16(9), 575–577 (1973)
12. Wong, E., Baur, B., Quader, S., Huang, C.-H.: Biological network motif detection: Principles and practice. Briefings in Bioinformatics 13(2), 202–215 (2012)
13. Perlin, M.W.: Transforming conjunctive match into RETE: A call-graph caching approach. Technical Report 2054, Carnegie Mellon University (1991)
14. Tan, J.S.E., Srivastava, J., Shekhar, S.: On the construction of efficient match networks. Technical Report 91, University of Houston (1991)
15. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)