

Grappa RE

A Tool for Efficient Graph Recognition Based on Finite Automata and Regular Expressions

Mattia De Rosa

Department of Informatics
University of Salerno
Fisciano (SA), Italy
matderosa@unisa.it

Mark Minas

Computer Science Department
Universität der Bundeswehr München
Neubiberg, Germany
mark.minas@unibw.de

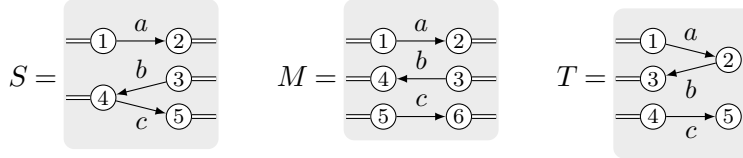
A recent paper by Drewes, Hoffmann, and Minas (GCM 2023 proceedings, to appear) has shown that certain graph languages can be defined and efficiently recognized by finite automata when strings over typed symbols are interpreted as graphs. This approach has been implemented in the tool *Grappa RE*, which is described in this paper. *Grappa RE* allows for the convenient specification of graph languages through regular expressions, converts each of them into a minimized deterministic finite automaton, and checks whether it can recognize graphs without the need for backtracking. Measurements confirm that recognition runs in linear time.

1 Introduction

Engelfriet and Vereijken have described that hypergraphs can be composed of elementary graphs using sequential and parallel composition to define hyperedge replacement grammars [10]. If only sequential composition, i.e., concatenation, is used, then such a composition yields a string over an alphabet (representing elementary graphs). The resulting graphs are therefore interpretations of these strings. Finite automata are a common means of defining regular (string) languages, and by interpreting the generated strings, such automata define graph languages. This idea is not new, finite automata for algebraic structures were already studied by Arbib and Give'on in the 1960s [1, 11], Bozapalidis and Kalampakas analyzed finite automata on Engelfriet's and Vereijken's hypergraphs [4, 16], Brugging, König *et al.* in the context of hypergraphs as cospans [3], and more recently Earnshaw and Sobociński studied regular languages of morphisms in free monoidal categories, which can be interpreted as hypergraphs or string diagrams, with their associated automata [9]. All of these approaches focus on *defining* graph languages through automata.

In contrast, Hoffmann, Drewes, and Minas recently proposed an approach to efficient graph recognition by finite automata in the sense that an automaton is “executed” to decide whether a given graph is a member of its graph language or not [8]. More precisely, given a finite automaton and a graph, a decision procedure can decide whether the graph is a member of the automaton's graph language. This is always possible with an inefficient backtracking algorithm, but for certain automata backtracking can be avoided, allowing a recognition in linear time in the number of graph edges. These automata must satisfy certain conditions, which are presented in [8] together with algorithms for checking them.

This paper describes the tool *Grappa RE*, which implements these checking algorithms, and the recognition procedure for automata satisfying these conditions. Special emphasis is placed on

Figure 1: Three graphs S , M , and T .

the techniques necessary for efficient graph recognition in linear time. *Grappa RE* also supports regular expressions, which can be used to conveniently specify finite automata. Experiments with this tool confirm the statement in [8] that it is possible to recognize graphs with finite automata in linear time. In this way, *Grappa RE* complements the tool landscape of GRAPPA¹. While GRAPPA (Graph Parsers) is a tool environment for generating efficient top-down and bottom-up parsers for hyperedge replacement grammars and contextual hyperedge replacement grammars [5, 14, 6, 7], *Grappa RE* (GRAPPA for Regular Expressions) realizes graph recognizers for regular expressions.

This paper is organized as follows: The next section describes informally the approach of using finite automata for efficient graph recognition, as proposed in [8], and describes in detail the underlying recognition algorithm with and without backtracking. Then Section 3 describes how this approach has been implemented in *Grappa RE*. Special emphasis is placed on the use of regular expressions as a compact way to specify finite automata, and the efficient implementation of constant time edge selection, which is required for graph recognition in linear time. Section 4 then summarizes the results of experiments with some graph languages specified by regular expressions and recognizable by *Grappa RE*. These results show that *Grappa RE* does indeed recognize graphs in linear time (in the number of their edges). Section 5 concludes the paper.

2 Finite Automata Accepting Graphs

We consider edge-labeled hypergraphs (which we will simply call graphs) without node labels. A ranked vocabulary is used to label hyperedges (which we will simply call edges). Each label ℓ has a rank $rank(\ell) \in \mathbb{N}$, and each edge labeled ℓ is then attached to $rank(\ell)$ different nodes. Each graph G also has a front and a rear interface $front(G)$ and $rear(G)$, respectively, which are repetition-free sequences of nodes in G . Front nodes may also occur in the rear interface. We say that a graph G is of type (m, n) if $|front(G)| = m$ and $|rear(G)| = n$; m is said to be the front type of G and n its rear type.

Figure 1 shows three examples of such graphs with the labels a , b , and c with $rank(a) = rank(b) = rank(c) = 2$. Each graph is contained in a rectangular box, with circles symbolizing nodes and arrows symbolizing binary edges. Nodes of the front interface are marked with double lines starting from the left edge of the box; double lines from the right edge mark nodes of the rear interface. The order of the double lines from top to bottom defines the order of the nodes in that interface. For example, the graph S consists of the nodes 1, 2, 3, 4 and 5 and three edges marked with a , b , and c . The front interface of S consists of the nodes 1 and 4, i.e., the node sequence 14, the rear interface of the node sequence 235. The graphs S , M , and T are of types $(2, 3)$, $(3, 3)$, and $(3, 0)$, respectively, because the rear interface of T is empty.

¹GRAPPA and *Grappa RE* are available at <https://www.unibw.de/inf2/grappa>

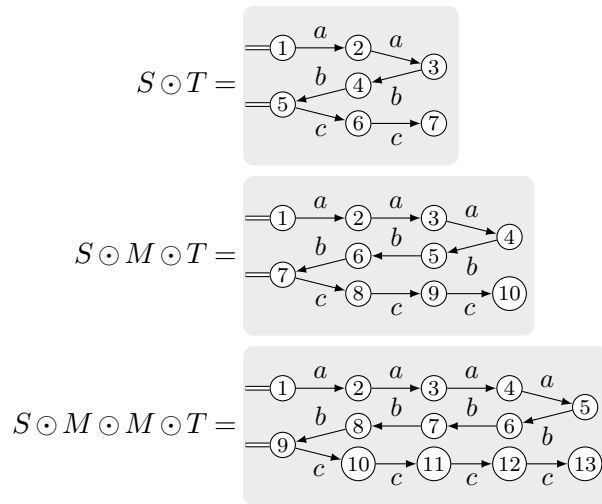


Figure 2: Results of concatenating S , M , and T .

Two graphs G and H can be concatenated to form a graph $G \odot H$ if their types match, which means that the rear type of G is the same as the front type of H . $G \odot H$ is obtained by taking the disjoint union of G and H and then merging the corresponding nodes from $rear(G)$ and $front(H)$. **Figure 2** shows the result of the concatenations $S \odot T$, $S \odot M \odot T$, and $S \odot M \odot M \odot T$. If we define $S \odot M^k \odot T$ as the graph resulting from the concatenation of S , k copies of M and finally T , then $S \odot M^k \odot T$ is a graph representation of the string $a^{k+2}b^{k+2}c^{k+2}$.

The approach in [8] is based on the idea of defining a vocabulary of graph symbols and assigning a specific (elementary) graph $\llbracket s \rrbracket$ to each symbol s as an interpretation. Strings over this vocabulary can then be interpreted as graphs obtained by concatenating the elementary graphs of the symbols. Of course, this assumes that the string is valid in the sense that it consists only of concatenations of symbols such that the corresponding elementary graphs match in type, so that their concatenation is defined. Otherwise, the string has no valid interpretation.

So-called *blank* and *atom symbols* are defined as vocabulary, with *blanks* and *atoms* as their graph interpretations. Blanks are discrete graphs whose front interface contains all nodes of the blank. In contrast, each atom contains exactly one edge, and no atom has a node that is neither in its front interface nor attached to its edge. Therefore, all nodes of the rear interface must also occur in the front interface, be connected to the edge, or both. These restrictions are necessary for efficient graph recognition, because in this way every node during the recognition process is either already contained in the rear interface of the already recognized subgraph, or is attached to the edge that is read next.

We write blank symbols as $\varepsilon_{\varrho}^{(n)}$ and atom symbols as ℓ_{ϱ}^{φ} for any $n \in \mathbb{N}$, sequences φ and ϱ over $\{1, \dots, n\}$ without repetitions, and edge label ℓ with $rank(\ell) \leq n$. The corresponding blank and atom, $\llbracket \varepsilon_{\varrho}^{(n)} \rrbracket$ and $\llbracket \ell_{\varrho}^{\varphi} \rrbracket$, respectively, have nodes $1, \dots, n$ and ϱ as their rear interfaces. $\llbracket \varepsilon_{\varrho}^{(n)} \rrbracket$ has $1 \cdots n$, $\llbracket \ell_{\varrho}^{\varphi} \rrbracket$ has φ as its front interface. The unique edge of $\llbracket \ell_{\varrho}^{\varphi} \rrbracket$ is labeled ℓ and attached to the nodes $1, \dots, rank(\ell)$ (in that order).

For example, **Figure 3** shows that the graph S in **Figure 1** is isomorphic to a concatenation of the three atoms $\llbracket a_{23}^{13} \rrbracket$, $\llbracket b_{312}^{32} \rrbracket$, and $\llbracket c_{342}^{41} \rrbracket$, i.e., S is an interpretation of $a_{23}^{13} b_{312}^{32} c_{342}^{41}$. Similarly,

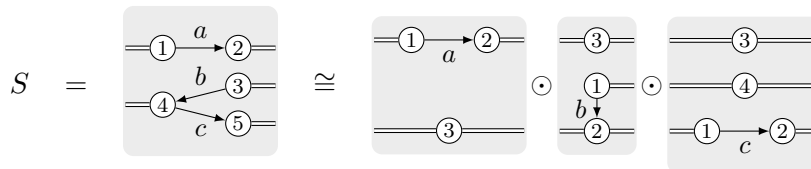


Figure 3: S as in Figure 1 as an interpretation of $a_{23}^{13} b_{312}^{32} c_{342}^{341}$.

M and T are interpretations of $a_{234}^{134} b_{314}^{324} c_{342}^{341}$ and $a_{234}^{134} b_3^{123} c_\varepsilon^1$, respectively.

A finite automaton over such symbols defines a language of strings accepted by the automaton, and thus defines a graph language, as long as all these strings have valid graph interpretations, which is easy to ensure: We define the type of each symbol to be the type of its elementary graph ($\varepsilon_\varrho^{(n)}$ and ℓ_ϱ^φ then have type $(n, |\varrho|)$ and $(|\varphi|, |\varrho|)$, respectively), and consider only those automata that allow assigning a rank $rank(q) \in \mathbb{N}$ to each state q , such that each incoming transition of q has a symbol of rear type $rank(q)$, and each outgoing transition of q has a symbol of front type $rank(q)$.

Let $a^n b^n c^n$ denote the language of all graphs representing strings in $\{a^k b^k c^k \mid k \in \mathbb{N}\}$. Continuing the example above, Figure 4 shows a finite automaton accepting all strings that can be interpreted as graphs in $a^n b^n c^n$. q_0 is its initial state, q_5 its only final state. The states have the ranks $rank(q_0) = rank(q_1) = 2$, $rank(q_2) = rank(q_4) = rank(q_6) = 3$, $rank(q_3) = 1$, and $rank(q_5) = 0$.

2.1 Recognition with Backtracking

While interpreting a string accepted by an automaton is straightforward, recognizing a given graph, i.e., finding a string representation of the graph such that the string is accepted by the automaton, involves a search process, usually with backtracking due to nondeterministic decisions. Algorithm 1 outlines this process as a classic depth-first search with backtracking using the recursive *depthFirst* procedure. It tries to find a walk through the finite automaton from the initial state to some final state that accepts a string with the input graph G as its interpretation. It either fails in Line 6 or terminates successfully in Line 8.

The algorithm does not call the *depthFirst* procedure but fails immediately if G has a discrete node that is not in its front interface (Line 5 and Line 6). Recall that no atom or blank can create a discrete node, since all nodes in the rear interface also occur in the front interface or are attached to the (unique) edge of the atom.

The *depthFirst* procedure is called with three parameters q, F, U : q is a state of the automaton, F (for “front”) contains a sequence of nodes that is the front interface of the next elementary

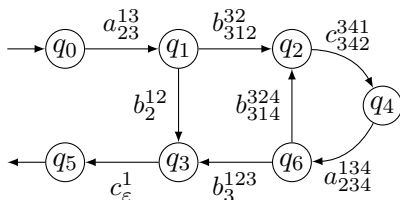


Figure 4: Finite automaton accepting graphs in $a^n b^n c^n$.

Algorithm 1: Recognizing a graph with a finite automaton using backtracking.

Input : input graph G and a finite automaton \mathfrak{A} .

Output: *success* if \mathfrak{A} accepts G , and *failure* otherwise.

```

1 begin
2    $q \leftarrow$  initial state of  $\mathfrak{A}$ ;
3    $F \leftarrow$  front interface of  $G$ ;
4    $U \leftarrow$  all edges of  $G$ ;
5   if all discrete nodes of  $G$  occur in  $F$  then call  $\text{depthFirst}(q, F, U)$ ;
6   stop with failure

7 procedure  $\text{depthFirst}(q$ : State,  $F$ : Sequence of nodes,  $U$ : Set of edges)
8   if  $q$  is final,  $F$  is the rear interface of  $G$ , and  $U = \emptyset$  then stop with success;
9   foreach outgoing transition  $t$  of  $q$  do
10    let  $q'$  be the state reached by  $t$ ;
11    if  $t$  is an atom transition then
12      let  $\alpha$  be the atom symbol of  $t$ ;
13      foreach edge  $e \in U$  as specified by  $\alpha$  and attached to nodes in  $F$  do
14        let  $F'$  be  $F$  after modifying it according to  $\alpha$  and  $e$ ;
15        call  $\text{depthFirst}(q', F', U \setminus \{e\})$ 
16    else
17      let  $F'$  be  $F$  after modifying it according to the blank symbol of  $t$ ;
18      call  $\text{depthFirst}(q', F', U)$ 

```

graph to be read, and U (for “unread”) contains all edges of G that have not yet been read. The procedure finally terminates successfully if it can (recursively) find a walk from q to some final state such that it reads all edges in U , starting with F as the front interface and ending with the rear interface of G . This means that if depthFirst is called with q being a final state, F being the rear interface of G , and $U = \emptyset$, [Algorithm 1](#) will terminate successfully ([Line 8](#)). Otherwise depthFirst tries each of the outgoing transitions of q in the usual depth-first manner in [Line 9](#), i.e., it selects an arbitrary transition first and continues the search with recursive procedure calls to depthFirst (see below). These calls return only if they cannot find a successful walk. The next transition is then selected in [Line 9](#), and so on. The current depthFirst call is terminated, i.e., [Algorithm 1](#) backtracks, if all outgoing transitions of q lead to a dead end.

If the selected transition in [Line 9](#) is an atom transition, i.e., if it is labeled with an atom symbol α , it tries to find a matching edge e not yet read ([Line 13](#)); α and F determine which edge may be selected: α determines the label of e and how it must be connected to nodes in F . In particular, the front nodes of $\llbracket \alpha \rrbracket$ must correspond to the nodes in F . Nodes that are not in F but are attached to e must be “new” nodes, i.e., they must not have been encountered before. The latter follows from the fact that nodes of $\llbracket \alpha \rrbracket$ that do not appear in its front interface are created by this step. If the recognizer has several candidates to choose from in [Line 13](#), it will select them one by one by recursively calling depthFirst . If this call fails, it will try the next one, and so on.

However, if the selected transition in [Line 9](#) is labeled with a blank symbol, no further

decision is required; a blank can always be processed, it just modifies the rear interface of the subgraph of G read so far (Line 17) and continues the depth-first search by recursively calling *depthFirst* in Line 18.²

2.2 Recognition without Backtracking

This search can be expensive because it uses backtracking to try out possible solutions instead of systematically selecting suitable transitions and edges, which can lead to exponential runtimes. To address this problem, [8] has discussed conditions under which backtracking can be avoided completely in this search process. In fact, the automaton must satisfy additional conditions to allow a deterministic recognition process without backtracking: It must have the *transition selection* (TS) property and the *free edge choice* (FEC) property, which will be explained below. Since backtracking is then not necessary, Algorithm 1 is transformed into a simplified version (Algorithm 2), which will be discussed next.

It is clear that a deterministic choice of a suitable transition is impossible if the automaton \mathfrak{A} is nondeterministic. That is, if it contains a state with two outgoing transitions labeled with the same graph symbol. Therefore, the finite automaton must first be transformed into a deterministic finite automaton (DFA), as in the case of transforming a nondeterministic finite automaton for strings, before it can be used by Algorithm 2. The approach proposed in [8] uses a modified version of the well-known powerset construction. Surprisingly, however, the powerset construction does not always produce a deterministic automaton in this context. This can happen if \mathfrak{A} uses atom symbols which differ only in their rear interfaces (see [8, Example 4.1]). In this case, however, one can always transform \mathfrak{A} into an equivalent automaton \mathfrak{A}' , which can then be transformed into an equivalent DFA \mathfrak{A}_d using the modified powerset construction. The transformation of \mathfrak{A} into the DFA \mathfrak{A}_d is thus a two-step process, consisting of the preprocessing step (called *disambiguation*) and the following (modified) powerset construction.

If \mathfrak{A}_d produced by the modified powerset construction contains blank transitions, all of them reach a final state that has no outgoing transitions. Consequently, a blank transition can only be selected if all edges of G are have been read. All previously selected transitions must be atom transitions. Each loop cycle of the while loop (Lines 5–14 of Algorithm 2) selects an atom transition and then reads a corresponding edge of G . The TS property of \mathfrak{A}_d implies that the only possible transition can be selected as shown in Algorithm 2: The foreach loop (Lines 6–13) tries all outgoing atom transitions of the current state in a predefined order until it finds the first one such that G contains a matching edge that has not yet been read before. This order is determined by the algorithm that checks the TS property of \mathfrak{A}_d described in [8]. In fact, \mathfrak{A}_d has the TS property if such an order exists for each of its states.

Note that Algorithm 2 in Line 9 selects any unread edge e that matches the atom symbol α of the selected transition. In fact, every edge that matches α is equally suitable if \mathfrak{A}_d has the FEC property: If one of these candidates is the right choice for \mathfrak{A}_d to accept G , then all of them are. [8] has proposed a sufficient criterion for checking the FEC property, i.e., \mathfrak{A}_d has the FEC property if the criterion is met. But if the criterion is not met, \mathfrak{A}_d may or may not have the FEC property. The criterion consists of identifying all transitions in \mathfrak{A}_d that are *deferrable*. These are transitions where Algorithm 2 can choose between several possible candidates in Line 9. They are called deferrable because one of the candidates is selected now, while the others must be

²Note that a cycle of blank transitions can lead to an infinite recursion, which could easily be prevented by keeping track of the transitions visited so far. However, this has been omitted here to simplify the presentation.

Algorithm 2: Recognizing a graph with a DFA that has the FEC and TS property.

Input : input graph G and a DFA \mathfrak{A}_d that has the FEC and TS property.
Output: *success* if \mathfrak{A}_d accepts G , and *failure* otherwise.

- 1 **if** G contains discrete nodes that are not in its front interface **then stop** with *failure*;
- 2 $q \leftarrow$ initial state of \mathfrak{A}_d ;
- 3 $F \leftarrow$ front interface of G ;
- 4 $U \leftarrow$ all edges of G ;
- 5 **while** $U \neq \emptyset$ **do**
- 6 **foreach** outgoing atom transition t of q **do**
- 7 let α be the atom symbol of t ;
- 8 **if** there is an edge $e \in U$ as specified by α and attached to nodes in F **then**
- 9 let e be such an edge;
- 10 modify F according to α and e ;
- 11 set q to the state reached by t ;
- 12 remove e from U ;
- 13 **continue** while loop at **Line 5**
- 14 **stop** with *failure*
- 15 **if** q is a final state and F is the rear interface of G **then stop** with *success*;
- 16 **if** there is an outgoing blank transition of q such that its blank turns F into the rear interface of G **then stop** with *success*;
- 17 **stop** with *failure*

selected in later cycles of the while loop, possibly using other transitions. [8] has outlined how to identify deferrable transitions. The criterion now requires that the rear interface of α does not contain any nodes that are not also contained in its front interface, i.e., the selection of e in **Line 9** cannot affect the selection of F in **Line 10**.

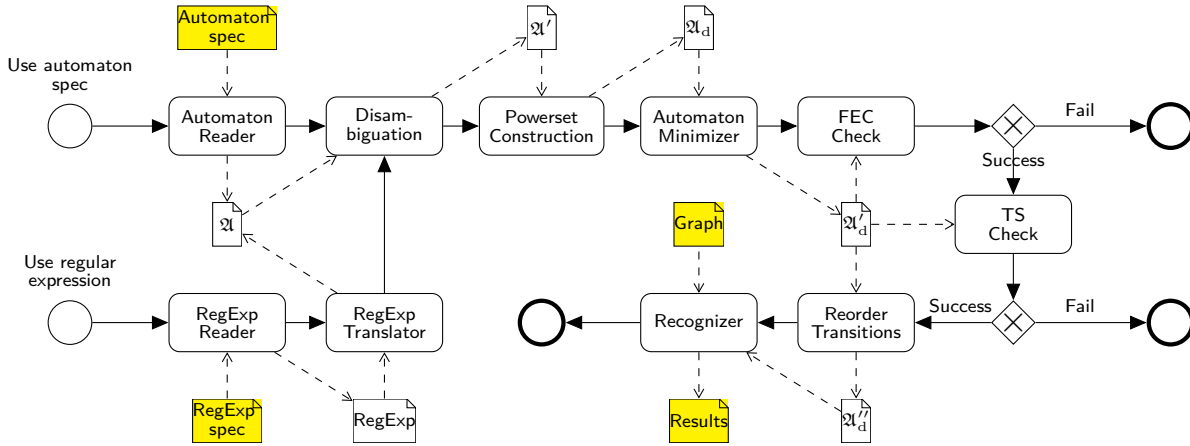
Algorithm 2 terminates successfully after it has read all edges of G in its while loop and has either reached a final state where F and the rear interface of G are identical (**Line 15**), or it can reach a final state by taking a blank transition that modifies F in the rear interface of G (**Line 16**). Otherwise, \mathfrak{A} does not accept G and **Algorithm 2** terminates with a failure.

As noted above, a single edge is read in each cycle of the while loop in **Lines 5–14**. [8] argued that a loop cycle takes only constant time if edge selection is implemented properly. Consequently, **Algorithm 2** takes only linear time (in the number of edges) to recognize an input graph. This is demonstrated in **Section 4**.

3 Grappa RE

The approach described in [8] and summarized in the previous section has been realized in the tool *Grappa RE*. It is implemented in JAVA and uses several external packages, in particular `dk.brics.automaton` [17] with its standard algorithms for finite automata and regular expressions for strings, and `GRAPHSTREAM` [2] for graph visualization. The architecture of *Grappa RE* is represented as a BPMN³ diagram in **Figure 5**. As usual, rectangles represent

³Business Process Management Notation

Figure 5: Architecture of *Grappa RE*

activities, document icons represent data, either external files (e.g., *Automaton spec*) read or written by *Grappa RE*, which have a yellow background, or internal data (e.g., \mathfrak{A}), drawn with a white background. The gateways represent decisions where only one of the outgoing paths is taken.

Grappa RE can be used in two modes, represented by the two different start events, i.e., either with the specification of a finite automaton (“*Use automaton spec*”), or with a regular expression (“*Use regular expression*”). We will describe the first mode first, then [Section 3.1](#) will describe the second mode. Both modes can be used interactively with a GUI or in a headless variant. The GUI is described in [Section 3.3](#), and the headless variant was used in the experiments to check the speed of the recognition procedure described in [Section 4](#).

Grappa RE reads finite automata and input graphs from text files. In the first mode (“*Use automaton spec*”), the *Automaton Reader* reads the specification of a finite automaton from a text file. [Figure 6](#) shows such a textual specification of the automaton in [Figure 4](#). The specification must contain the ranked alphabet of edge labels after the `symbol` keyword; the rank of each symbol is given in parentheses. The states are listed after the `state` keyword. State ranks (see [Section 2](#)) are again given in parentheses. Final states are marked with `*`, the start state with the `start` keyword, and transitions in the obvious way. For example, a^{13}_{23} represents the atom symbol a_{23}^{13} and $c^1_{<>}$ represents the atom symbol $c_{<>}^1$. Blank symbols such as $\varepsilon_{12}^{(2)}$ are written as $<>^2_{12}$, but do not appear in [Figure 6](#).

The *Automaton Reader* produces an internal representation \mathfrak{A} of the automaton and then

```

auto abc {
  symbol a(2), b(2), c(2);
  state q0(2), q1(2), q2(3), q3(1), q4(3), q5(0)*, q6(3);
  start q0;
  q0 -- a^13_23 --> q1; q1 -- b^32_312 --> q2; q1 -- b^12_2 --> q3;
  q2 -- c^341_342 --> q4; q3 -- c^1_<> --> q5; q4 -- a^134_234 --> q6;
  q6 -- b^324_314 --> q2; q6 -- b^123_3 --> q3;
}

```

Figure 6: *Grappa RE* specification of the automaton in [Figure 4](#).

transforms it into a DFA in the two-step process outlined in [Section 2](#): *Disambiguation* is the preprocessing step that ensures that the *Powerset Construction* really produces a deterministic finite automaton, denoted here by \mathfrak{A}_d .

Similar to the string case, the DFA \mathfrak{A}_d can contain nondistinguishable states. These are states that accept the same languages over graph symbols when starting in these states. Since sequences of graph symbols are special cases of strings, one can use any standard algorithm to minimize \mathfrak{A}_d by merging all non-distinguishable states. *Grappa RE* uses Hopcroft’s algorithm [15] with an implementation provided by `dk.brics.automaton` [17], which yields an equivalent DFA \mathfrak{A}'_d .

The following steps in [Figure 5](#) check if backtracking can be avoided when using \mathfrak{A}'_d to recognize graphs. *FEC Check* checks if it has the FEC property (see [Section 2](#)), and *TS Check* checks if it has the TS property using the algorithms described in [8]. If one of them fails, efficient graph recognition cannot be guaranteed, and *Grappa RE* refuses to use \mathfrak{A}'_d for graph recognition. However, if both succeed, *Reorder Transitions* then reorders the transitions of each state as required by [Algorithm 2](#) and described in [8]. In fact, this is done by topologically sorting the (atom) transitions of each state according to a partial order obtained by the TS check. The output of this step is a DFA \mathfrak{A}''_d .

\mathfrak{A}''_d is finally used in the *Recognizer* step that implements [Algorithm 2](#). The input graph G is either read from an external file in JSON format, or it can be generated programmatically in headless mode. This step checks if \mathfrak{A}''_d (and therefore \mathfrak{A}) accepts G by finding a sequence s of graph symbols such that $G \cong \llbracket s \rrbracket$. The result of this step, which can be visually inspected in GUI mode, is described in [Section 3.3](#).

The *Recognizer* step tries to recognize the input graph as described by [Algorithm 2](#), i.e., by selecting suitable edges of G step by step. *Grappa RE* provides two different implementations of this edge selection: a simple implementation searches for a suitable edge iteratively and thus takes linear time (in the number of edges of G). Graph recognition then takes quadratic time, as shown in [Section 4](#). However, *Grappa RE* also provides a more efficient selection implementation, described in [Section 3.2](#), which takes only constant time. Thus, *Grappa RE* can recognize graphs in linear time, as claimed in [8] and demonstrated in [Section 4](#).

3.1 Regular Expressions

Grappa RE also provides a second mode where the finite automaton is not specified directly as in [Figure 6](#), but by a regular expression. These regular expressions use graph symbols as elementary symbols, but are otherwise structured like normal regular expressions. In particular, they use concatenation and alternatives, as well as the Kleene star, with the same semantics as in the string case. Each such expression corresponds in the usual way to a finite automaton with the same language. However, not every regular expression over graph symbols is a valid expression. Rather, the finite automaton corresponding to the expression may only accept valid graph interpretations as sequences of graph symbols, i.e., the types of successive graph symbols must match. Just as this can be ensured for automata (see [Section 2](#)), it is also possible for regular expressions, as long as it is not just ε , because it does not have any type information, i.e., how many front and rear nodes a corresponding graph has. Consequently, ε is not a valid regular expression.

For example, the automaton shown in [Figure 4](#) and [Figure 6](#) can be more compactly described

```

regexp abc {
  symbol a(2), b(2), c(2);

  a^13_23 b^12_2 c^1_<>
  |
  a^13_23 b^32_312 c^341_342
  (a^134_234 b^324_314 c^341_342)*
  a^134_234 b^123_3 c^1_<>
}

```

Figure 7: Regular expression for the automaton in Figure 4.

by the regular expression

$$a_{23}^{13} b_2^{12} c_\varepsilon^1 \mid a_{23}^{13} b_{312}^{32} c_{342}^{341} (a_{234}^{134} b_{314}^{324} c_{342}^{341})^* a_{234}^{134} b_3^{123} c_\varepsilon^1.$$

Step *RegExp Reader* (see Figure 5) can read such a regular expression in textual format as in Figure 7. This specification must also include a declaration of the ranked vocabulary of edge labels. Graph symbols are then specified with the same syntax as in automata specifications (e.g., Figure 6).

RegExp Reader also checks if the regular expression is valid, and *RegExp Translator* then transforms it into an equivalent finite automaton, yielding \mathcal{A} as an internal representation of the resulting finite automaton. To implement this step, it again uses `dk.brics.automaton` [17], which provides an implementation for the string case that can be used directly here. The automaton \mathcal{A} is then used in the same way as in the mode “*Use automaton spec*”.

3.2 Efficient Edge Selection

As described above, *Grappa RE* implements edge selection as used in Line 9 of Algorithm 2 in a straightforward simple way and in an efficient proper way. We will refer to them as the *simple* and *efficient* implementations, respectively.

To speed up edge selection, both implementations build a data structure before starting the recognition process. The simple implementation simply collects lists of edges, where each list contains all edges with the same label. When Line 9 needs to select an edge as specified by an atom symbol ℓ_φ , it inspects the list for ℓ and looks for the first edge attached to the current nodes in F as specified by φ . Once a matching edge is found, it can be efficiently removed from the list because it is implemented as a doubly linked list. However, the sequential search for this edge takes linear time in the number of edges with the label ℓ . Section 4 shows that graph recognition with this simple implementation then takes quadratic time and is rather slow.

The efficient implementation builds more sophisticated data structures before starting the recognition process and uses techniques similar to those in GRAPPA for efficient graph parsing [14]. It uses a two-step process:

1. In the first step, the automaton \mathcal{A}''_d (see Figure 5) is analyzed and all atom symbols in \mathcal{A}''_d are collected in a set. This set contains all atom symbols that can specify an edge selection (as in Line 9 in Algorithm 2). Each of the atom symbols ℓ_φ specifies how the edge to be selected must be connected to the current nodes in F . The other nodes of the edge, i.e., those that are not referred to by φ , must be nodes that have not been encountered before.

This information is then used in the second step to collect suitable lists of edges, which speeds up edge selection.

2. In the second step, lists of edges of the input graph G are collected, one list for each situation that can occur during edge selection (as in **Line 9** in **Algorithm 2**). Each of these lists will contain all edges that are suitable in the corresponding situation.

As an example, consider the graph language $a^n b^n c^n$ and its DFA in **Figure 4**. It contains the atom symbols a_{23}^{13} , a_{234}^{134} , b_2^{12} , b_3^{123} , b_{312}^{32} , b_{314}^{324} , c_ε^1 , and c_{342}^{341} . Now consider the situation of selecting an edge that matches $\ell_\varphi^\varphi = a_{23}^{13}$, F in **Algorithm 2** must consist of two nodes, say $F = (m_1, m_2)$. 1 at position 1 of $\varphi = 13$ indicates that one must choose an a -edge attached to two nodes (n_1, n_2) such that $n_1 = m_1$, and n_2 is not yet read since 2 does not occur in φ . Finding such an edge is easy and takes only constant time if each node has a list of all a -edges connected to that node as its first node. This is the usual association list approach. Note, however, that one cannot select an a -edge whose second node has already been read. Consequently, one must remove all a -edges from the corresponding association lists as soon as their second attached node has been encountered. This can also be done efficiently if each edge keeps the information in which association lists it is stored. Their number has an upper bound, which can be read from the set of atom symbols of \mathfrak{A}_d'' . And removing edges from all these lists can also be done efficiently if these lists are doubly linked lists and one keeps references to the corresponding list buckets. Therefore, removing an edge from all corresponding lists also takes constant time.

However, keeping only simple association lists is not always sufficient to ensure constant time edge selection. To see this, consider the *Spikes* graph language represented by

$$s_{134}^{124} (s_{124}^{134})^* s_\varepsilon^{123} \mid s_{421}^{423} (s_{423}^{421})^* s_\varepsilon^{321} \mid s_{243}^{143} (s_{143}^{243})^* s_\varepsilon^{123}$$

where label s has rank 3. We call these graphs *Spikes* because of their shape; **Figure 8** shows three of them. Nodes are drawn as circles, s -edges as rectangles connected to their attached nodes. Numbers indicate the order of the attachments. *Spikes* graphs are of type $(3,0)$. Nodes of their front interfaces are indicated by lines to the left border, ordered from top to bottom. Graph recognition must start with a unique edge; for the graphs in **Figure 8** this is always the edge attached to n_1, n_2, n_3 . The central node n_1 , which is connected to all edges of the graph, can be any node of its front interface, as shown in **Figure 8**. Consequently, any front node can

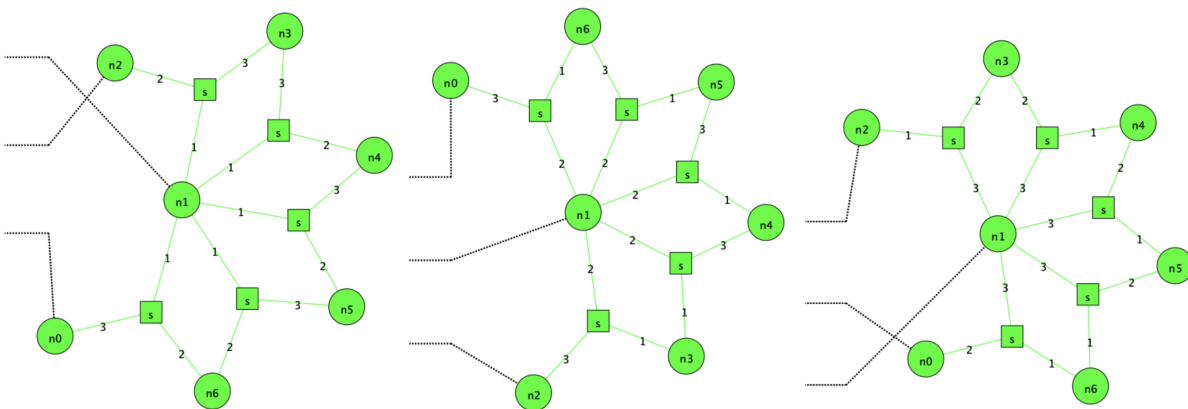


Figure 8: Three *Spikes* graphs.

be attached to any number of edges. Thus, it is impossible to select the unique starting edge in constant time by simply accessing an association list of a front node. Instead, one must be able to look up edges by pairs of their nodes, as can be seen in the following example: The first alternative of the regular expression starts with s_{134}^{124} , i.e., we are looking for an s -edge whose first and second attached nodes are the first and second nodes in its front interface, i.e., n_1 and n_2 for the left *Spikes* graph. This is only the case for the edge attached to n_1, n_2, n_3 .

Grappa RE uses hash tables to manage association lists for tuples of nodes and to allow efficient edge lookup. The association lists are created and filled with edges before recognition starts. The hash table lookup, and thus the association list lookup, takes constant time on average because each hash table is fixed after preprocessing; only the contents of the association lists are modified in the same way as described above. Since the number of atom symbols occurring in a DFA is fixed, all these data structures require linear space in the size of the input graph if each hash table size is chosen to be proportional to the number of all edges. In addition, on average, setting up these data structures takes linear time in the size of the input graph.

3.3 Graphical User Interface

Grappa RE also provides a GUI that allows the user to interactively visualize the recognition process. Specifically, as shown in [Figure 9](#), the *Grappa RE* window is divided into three parts: on the top left is a visualization of the input graph, on the top right is the visualization of the minimized DFA and a bar to control execution, and at the bottom is the sequence of atoms recognized during execution. Since the input graph does not provide any layout information, a force-directed layout algorithm [2] is used to display it, which also allows the user to manually move nodes by dragging them with the mouse. The same type of interface was used for the DFA, where the initial state is highlighted in yellow and accepting states have a dashed border instead.

The user can advance the execution of the DFA manually step by step by pressing the *Next* button, or automatically at a selected time interval by pressing the *Run* button. During this operation, the current state of the DFA is highlighted in red, as well as the edges representing the transitions when they are followed. “Consumed” input graph elements are also highlighted (in green) during execution, and it is also possible to click on them later to get information about which transition marked them (as shown in [Figure 9](#) below).

Finally, at the bottom, the sequence of atoms is shown as it was detected during recognition. For ease of understanding, the user can also click the *Merge* button to replace the first two atoms by their concatenation. This merging process is even animated. It is possible to keep clicking this button to iteratively merge the two leftmost graphs until the bottom contains only one graph, i.e., the concatenation of all the atoms at the beginning. The user can also save this visualization as a graphic file in SVG format.

4 Examples and Experiments

To confirm that graph recognition with finite automata can indeed run in linear time as claimed in [Section 2](#) and [8], the recognition time was measured for graphs of four different graph languages. The graph languages are $a^n b^n c^n$ and *Spikes*, as introduced in [Sections 2](#) and [3](#), respectively, and *Palindromes* and *Wheels*. *Palindromes* contains all graphs representing palindromes over $\{a, b\}$, and *Wheels* contains all wheel graphs as defined in [12, p. 92] and shown in

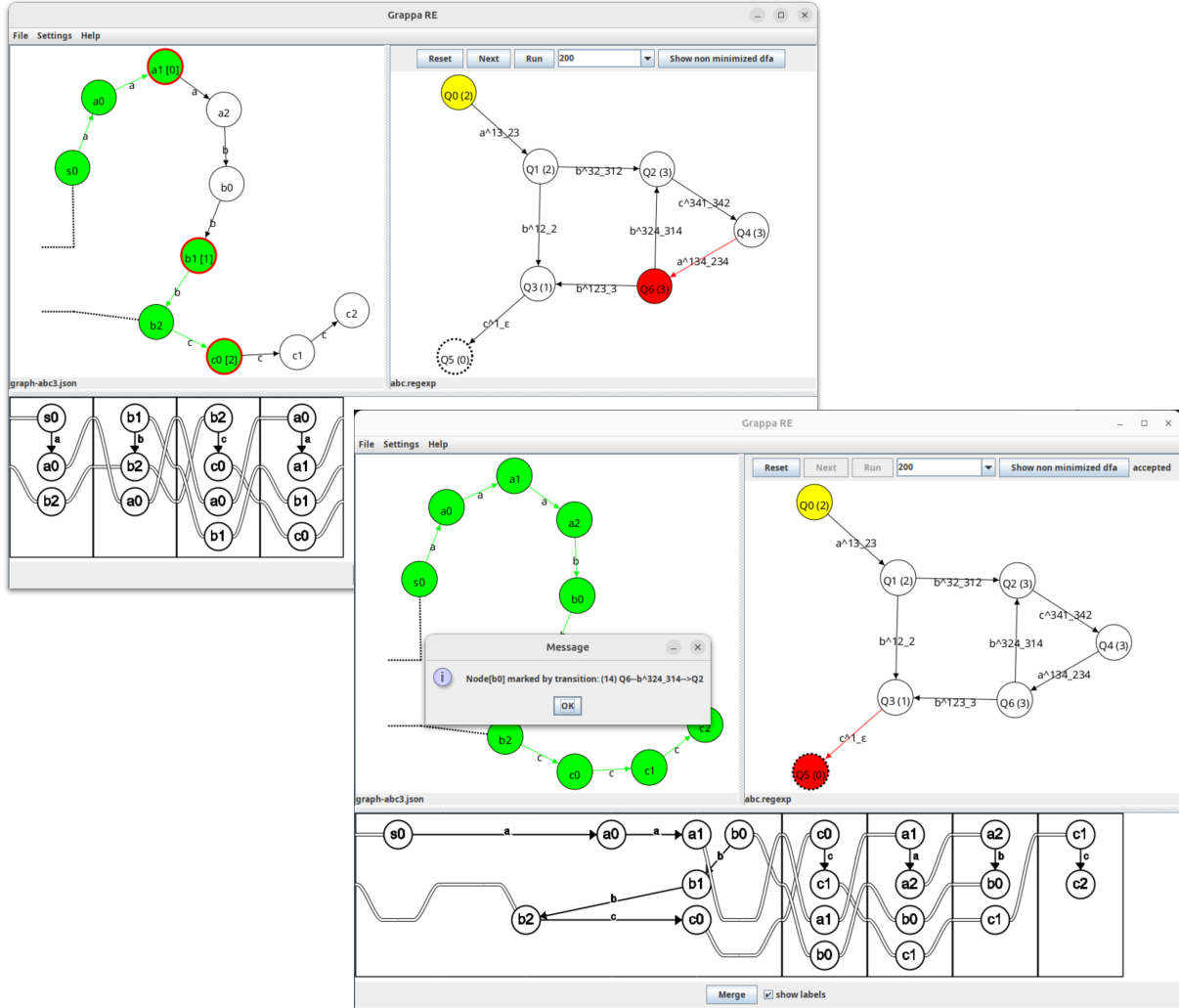


Figure 9: Two screenshots of *Grappa RE* with a graph representing $aaabbbccc$ and the automaton recognizing $a^n b^n c^n$ (see also Figure 4). Top: The current state sequence: Q_0, Q_1, Q_2, Q_4 and Q_6 . Lower: Completed execution (accepted graph). A dialog tells you which transition has read and marked a node by clicking on it, here b_0 . At the bottom you can see the result after the user merged the first four atoms by clicking the *Merge* button four times.

Figure 10. The latter two are defined by

$$\begin{aligned}
 & (a_{23}^{13} a_{31}^{32} \mid b_{23}^{13} b_{31}^{32})^* (a_{23}^{13} a_{\epsilon}^{12} \mid b_{23}^{13} b_{\epsilon}^{12} \mid a_{\epsilon}^{12} \mid b_{\epsilon}^{12}) && \text{Palindromes} \\
 & t_{12}^{\epsilon} s_{123}^{32} (t_{324}^{314} s_{123}^{123})^* t_{32}^{312} s_{\epsilon}^{12} && \text{Wheels}
 \end{aligned}$$

The experiments consisted of running graph recognition using both the simple and the efficient edge selection algorithms (see Section 3.2) on input graphs of increasing size and measuring the execution time of the recognition algorithm. They were run on a 2022 Mac Studio with an Apple M1 Max processor, 64 GiB of RAM, and the Temurin 21 JAVA virtual machine. To get more accurate measurements, each recognition run was repeated 40 times on the same graph,

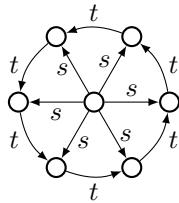


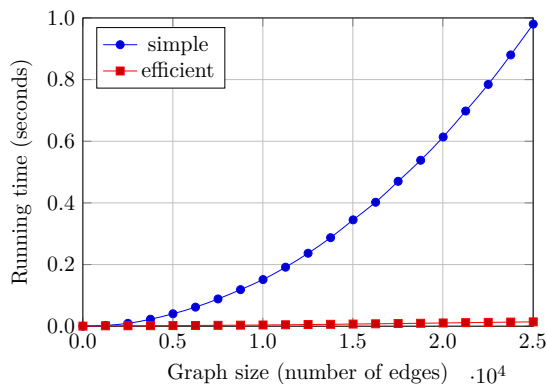
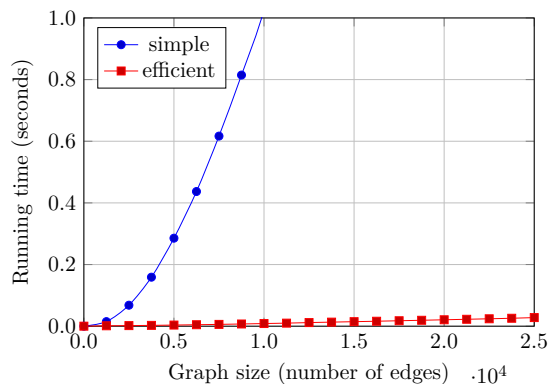
Figure 10: A wheel graph with six spokes.

shuffling the edges randomly each time to avoid any bias due to favorable or unfavorable ordering of the edges. The average execution time was then calculated by excluding the four worst times to exclude cases that were slowed down by garbage collection. The results of such executions are shown in Figures 11 and 12 for the languages $a^n b^n c^n$ and *Spikes*, respectively. In the plots, the x-axis shows the number of edges of the input graph and the y-axis shows the execution time in seconds. They clearly show that the runtime for the simple implementation of edge selection is quadratic and that the efficient implementation is indeed much faster.

In addition, Figure 13 compares the time taken to recognize graphs using only the efficient edge selection algorithm for the four languages on larger input graphs; in this case, we reduced the number of runs of each test from 40 to 6 in order to reduce the time required given the larger size of the graphs. It can be seen that the increase in execution time remains linear even when scaling up to large graphs with a million edges.

In addition to the recognition execution time, we also measured the time taken for the various operations required to create the DFA for the four languages. The operations of generating the automaton from a regular expression, checking the automaton for ambiguity, performing the powerset construction, minimizing the DFA, and checking the DFA for the FEC property took less than 1ms in total. Checking the TS property (and reordering the transitions) took on average 7.5ms (between 6.8ms and 8.3ms). Since it is also possible to store the DFA resulting from these operations and reuse it for recognition operations, the time taken by these operations can be considered negligible.

Regarding the times shown in the graphs, in the case of the efficient edge selection imple-

Figure 11: Recognizing $a^n b^n c^n$ graphs using the simple and the efficient implementation.Figure 12: Recognizing *Spikes* graphs using the simple and the efficient implementation.

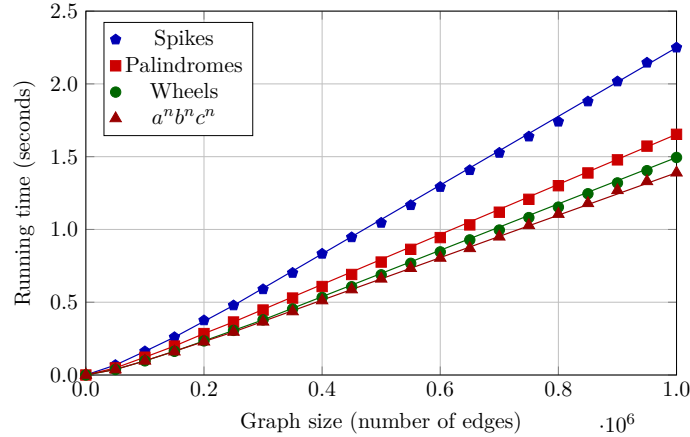


Figure 13: Recognizing different kinds of graphs using the efficient implementation.

mentation, they include both the time to generate the optimized data structures of the input graph and the time to execute the DFA. In particular, the time to create the optimized data structures was found to be about two-thirds of the total time, while the time to execute the DFA was about one-third of the total time.

5 Conclusions

This paper has described *Grappa RE*, a tool for analyzing graphs in the sense that it decides for a given graph whether it can be generated by a given finite automaton or not. *Grappa RE* implements the approach recently proposed by Hoffmann, Drewes, and Minas [8], where graphs are considered as interpretations of strings and finite automata are used to define a graph language and also as a device for deciding the membership problem. In particular, *Grappa RE* implements the procedures for checking whether a given automaton allows efficient graph recognition. This paper has shown that the recognition is indeed linear in the size of the input graph, and it has described the implementation approach that was necessary to achieve linear runtime complexity. Experiments have shown that even very large graphs can be analyzed very quickly with this implementation, since only about $2\mu\text{s}$ is needed to analyze each edge of the input graph on standard computers.

Future work will consider applications of the approach described in [8], e.g., by using regular expressions to be used in the nested graph conditions of Habel and Pennemann [13, 18] to specify global graph properties and check them with automata.

References

- [1] Michael A. Arbib & Yehoshafat Give'on (1968): *Algebra Automata I: Parallel Programming as a Prolegomena to the Categorical Approach*. *Information and Control* 12(4), pp. 331–345, doi:[10.1016/S0019-9958\(68\)90374-4](https://doi.org/10.1016/S0019-9958(68)90374-4).
- [2] Stefan Balev, Antoine Dutot, Yoann Pigné, Guilhelm Savin et al. (2024): *GraphStream - A Dynamic Graph Library*. Available at <https://graphstream-project.org/>.

- [3] Christoph Blume, H.J. Sander Bruggink, Martin Friedrich & Barbara König (2013): *Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-Accepting Tree Automata*. *Journal of Visual Languages & Computing* 24(3), pp. 192–206, doi:[10.1016/j.jvlc.2012.10.002](https://doi.org/10.1016/j.jvlc.2012.10.002).
- [4] Symeon Bozagalidis & Antonios Kalampakas (2008): *Graph automata*. *Theoretical Computer Science* 393(1-3), pp. 147–165, doi:[10.1016/j.tcs.2007.11.022](https://doi.org/10.1016/j.tcs.2007.11.022).
- [5] Frank Drewes, Berthold Hoffmann & Mark Minas (2015): *Predictive Top-Down Parsing for Hyperedge Replacement Grammars*. In Francesco Parisi-Presicce & Bernhard Westfechtel, editors: *Graph Transformation - 8th International Conf., ICGT 2015. Proceedings*, LNCS 9151, Springer, pp. 19–34, doi:[10.1007/978-3-319-21145-9_2](https://doi.org/10.1007/978-3-319-21145-9_2).
- [6] Frank Drewes, Berthold Hoffmann & Mark Minas (2019): *Formalization and Correctness of Predictive Shift-Reduce Parsers for Graph Grammars based on Hyperedge Replacement*. *Journal on Logical and Algebraic Methods for Programming* 104, pp. 303–341, doi:[10.1016/j.jlamp.2018.12.006](https://doi.org/10.1016/j.jlamp.2018.12.006).
- [7] Frank Drewes, Berthold Hoffmann & Mark Minas (2021): *Rule-Based Top-Down Parsing for Acyclic Contextual Hyperedge Replacement Grammars*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 14th International Conference, ICGT 2021*, LNCS 12741, Springer, pp. 164–184, doi:[10.1007/978-3-030-78946-6_9](https://doi.org/10.1007/978-3-030-78946-6_9).
- [8] Frank Drewes, Berthold Hoffmann & Mark Minas (2024): *Finite Automata for Efficient Graph Recognition*. To appear in Proceedings 14th International Workshop on Graph Computation Models, Leicester, UK, 18th July 2023. Available at <http://arxiv.org/abs/2404.15052>.
- [9] Matthew Earnshaw & Pawel Sobociński (2022): *Regular Monoidal Languages*. In Stefan Szeider, Robert Ganian & Alexandra Silva, editors: *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 241, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 44:1–44:14, doi:[10.4230/LIPIcs.MFCS.2022.44](https://doi.org/10.4230/LIPIcs.MFCS.2022.44).
- [10] Joost Engelfriet & Jan Joris Vereijken (1997): *Context-Free Graph Grammars and Concatenation of Graphs*. *Acta Informatica* 34(10), pp. 773–803, doi:[10.1007/s002360050106](https://doi.org/10.1007/s002360050106).
- [11] Yehoshafat Give'on & Michael A. Arbib (1968): *Algebra Automata II: The Categorical Framework for Dynamic Analysis*. *Information and Control* 12(4), pp. 346–370, doi:[10.1016/S0019-9958\(68\)90381-1](https://doi.org/10.1016/S0019-9958(68)90381-1).
- [12] Annegret Habel (1992): *Hyperedge Replacement: Grammars and Languages*. LNCS 643, Springer, doi:[10.1007/BFb0013875](https://doi.org/10.1007/BFb0013875).
- [13] Annegret Habel & Karl-Heinz Pennemann (2005): *Nested Constraints and Application Conditions for High-Level Structures*. In H.-J. Kreowski et al., editors: *Formal Methods in Software and System Modeling*, LNCS 3393, Springer, pp. 293–308, doi:[10.1007/978-3-540-31847-7_17](https://doi.org/10.1007/978-3-540-31847-7_17).
- [14] Berthold Hoffmann & Mark Minas (2017): *Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars*. In M. Seidl & S. Zschaler, editors: *STAF 2017 Workshops*, LNCS 10748, Springer, pp. 76–91, doi:[10.1007/978-3-319-74730-9_7](https://doi.org/10.1007/978-3-319-74730-9_7).
- [15] John Hopcroft (1971): *An $n \log n$ Algorithm For Minimizing States In A Finite Automaton*. In Zvi Kohavi & Azaria Paz, editors: *Theory of Machines and Computations*, Academic Press, pp. 189–196, doi:[10.1016/B978-0-12-417750-5.50022-1](https://doi.org/10.1016/B978-0-12-417750-5.50022-1).
- [16] Antonios Kalampakas (2011): *Graph Automata: The Algebraic Properties of Abelian Relational Graphoids*. In Werner Kuich & George Rahonis, editors: *Algebraic Foundations in Computer Science - Essays Dedicated to Symeon Bozagalidis on the Occasion of His Retirement*, LNCS 7020, Springer, pp. 168–182, doi:[10.1007/978-3-642-24897-9_8](https://doi.org/10.1007/978-3-642-24897-9_8).
- [17] Anders Møller (2021): *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. Available at <http://www.brics.dk/automaton/>.
- [18] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Dissertation, Carl-von-Ossietzky-Universität Oldenburg. Available at <http://oops.uni-oldenburg.de/884/1/pendev09.pdf>.