Electronic Communications of the EASST
Volume 39 (2011)

Graph Computation Models
Selected Revised Papers from the
Third International Workshop on
Graph Computation Models (GCM 2010)

Minimizing Finite Automata with Graph Programs

Detlef Plump, Robin Suri, and Ambuj Singh

15 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Minimizing Finite Automata with Graph Programs*

Detlef Plump[1], **Robin Suri**[2], **and Ambuj Singh**[3]

[1] The University of York, UK [2] Indian Institute of Technology Roorkee, India [3] Indian Institute of Technology Kanpur, India

Abstract: GP (for Graph Programs) is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from dealing with low-level data structures. In this case study, we present a graph program which minimizes finite automata. The program represents an automaton by its transition diagram, computes the state equivalence relation, and merges equivalent states such that the resulting automaton is minimal and equivalent to the input automaton. We illustrate how the program works by a running example and argue that it correctly implements the minimization algorithm of Hopcroft, Motwani and Ullman. We also prove a quadratic upper bound for the number of rule schema applications used by the program.

Keywords: Graph programs, automata minimization, rule-based programming, correctness proofs

1 Introduction

GP is an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata for graph transformation, freeing programmers from implementing and handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for an abstract machine, and comes with a graphical editor for programs and graphs. We refer to [Plu09] for an overview of the language and to [MP08] for a description of the current implementation.

In this paper, we present a case study about solving a problem with GP that as first sight may not appear to be a graph problem: the minimization of finite automata. It is natural though to represent finite automata by their transition diagrams and to view the minimization process as a sequence of transformation steps on these diagrams. Programmers can visually construct corresponding rule schemata and control the application of these schemata by GP's commands.

We implement the minimization algorithm of Hopcroft, Motwani and Ullman [HMU07] (see also [Sha09]). This algorithm first computes the indistinguishability relation among states, called *state equivalence*, and then merges equivalent states to obtain a minimal automaton that is equivalent to the input automaton. Two states are equivalent if processing strings from either state will have the same result with respect to acceptance. While state equivalence is usually computed by a table-filling algorithm, in our case we directly connect equivalent states with special edges. Once the equivalent states have been determined, we merge them by redirecting edges and removing isolated nodes.

* Work of the second and third author was done while visiting the University of York. Funding by the Department of Computer Science at York is gratefully acknowledged.

In Section 5, we argue that our implementation is correct in that the graph program will transform every input automaton into an equivalent and minimal output automaton. This involves showing that the program terminates, that it correctly computes the state equivalence relation, and that the merging phase produces an automaton in which each equivalence class of states is represented by a unique state. We also show, in Section 6, that the maximal number of rule schema applications used by our program is quadratic in the size of the input automaton.

This paper is a revised and extended version of [PSS10].

2 Graph Programs

We briefly review GP's conditional rule schemata and control constructs. Technical details (including the abstract syntax and operational semantics of GP) can be found in [Plu09], as well as a number of example programs.

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise graph transformation rules in the double-pushout approach with relabelling [HP02], in that labels can contain expressions over parameters of type integer or string. Figure 1 shows a conditional rule schema consisting of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema, the node identifiers 1, 2, 3 specifying which nodes are preserved, and the keyword `where` followed by the condition `not edge(1,3)`.

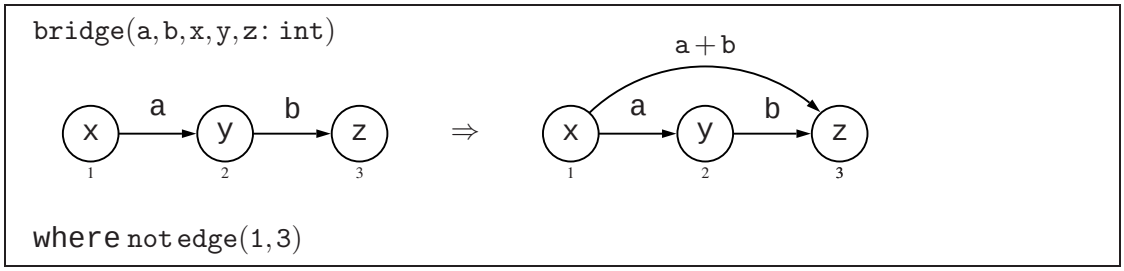


Figure 1: A conditional rule schema

In the GP programming system [MP08], rule schemata are constructed with a graphical editor. Labels in the left graph comprise only variables and constants because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1,3)` in the condition of Figure 1 forbids an edge from node 1 to node 3 when the left graph is matched.

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules, and are applied according to the double-pushout approach with relabelling. A rule schema $L \Rightarrow R$ with condition Γ represents conditional rules $\langle\langle L^\alpha \leftarrow K \rightarrow R^\alpha\rangle, \Gamma^{\alpha,g}\rangle$, where K consists of

the preserved nodes (which are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^{\alpha} \to G$ (see [Plu09]).

GP's commands for controlling rule-schema applications include the non-deterministic one-step application of a rule schema, the non-deterministic one-step application of a set $\{r_1,\ldots,r_n\}$ of rule schemata, the sequential composition $P;Q$ of programs P and Q, the as-long-as-possible iteration $P!$ of a program P, and the branching statement `if` C `then` P `else` Q for programs C, P and Q. The first four of these commands have the expected effects. The branching command first checks if executing C on the current graph G can produce a graph; if this is the case, then P is executed on G, otherwise Q is executed on G.

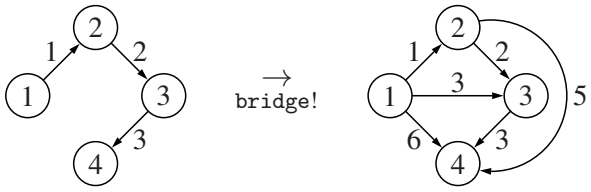


Figure 2: An execution of the program `bridge!`

For example, Figure 2 shows an execution of the program `bridge!`. This program makes an input graph transitive in that for every directed path of the input, the output graph contains an edge from the first node to the last node of the path. Note that the edge with label 6 can be produced by applying `bridge` in two different ways, performing either the addition $3+3$ or $1+5$. In general, a program may produce many different output graphs for the same input. The semantics of GP assigns to every input graph the set of all possible output graphs (see [Plu09, PS10]).

3 Automata Minimization

Our starting point is the abstract minimization algorithm of Hopcroft, Motwani and Ullman [HMU07] (see also [Sha09]). To fix notation, we consider a deterministic finite automaton (DFA) as a system $A = (Q,\Sigma,\delta,q_0,F)$ where Q is the finite set of states, Σ is the input alphabet, $\delta\colon Q\times\Sigma\to Q$ is the transition function, q_0 is the initial state, and F is the set of final (or accepting) states. The extension of δ to strings is denoted by $\delta^*\colon Q\times\Sigma^*\to Q$.

Definition 1 States p and q of an automaton are *equivalent* if for all strings $w\in\Sigma^*$, $\delta^*(p,w)\in F$ if and only if $\delta^*(q,w)\in F$.

Note that this indeed defines an equivalence relation. We say that states p and q are *distinguishable* if they are not equivalent, that is, there must be some string $w\in\Sigma^*$ such that either $\delta^*(p,w)\in F$ and $\delta^*(q,w)\notin F$, or vice-versa.

The following minimization algorithm first marks all unordered pairs of distinguishable states of an automaton A—thus representing state equivalence implicitly by all unmarked pairs of states. In a second phase, equivalent states are merged to form the states of the minimal automaton $\hat{A}$.

Algorithm 1 ([HMU07])

Marking phase

Stage 1:
for each $p \in F$ and $q \in Q - F$ **do** mark the pair $\{p,q\}$

Stage 2:
repeat
  for each non-marked pair $\{p,q\}$ **do**
    for each $a \in \Sigma$ **do**
      if $\{\delta(p,a), \delta(q,a)\}$ is marked **then** mark $\{p,q\}$
until no new pair is marked

{For each state p, the equivalence class of p consists of all states q for which the pair $\{p,q\}$ is not marked.}

Merging phase

Construct $\hat{A} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q_0}, \hat{F})$ as follows:

- $\hat{Q}$ consists of the state equivalence classes.
- $\hat{q_0}$ is the equivalence class containing q_0.
- For each $X \in \hat{Q}$ and $a \in \Sigma$, pick any $p \in X$ and set $\hat{\delta}(X,a) = Y$, where Y is the equivalence class containing $\delta(p,a)$.
- $\hat{F}$ consists of the equivalence classes containing states from F.

By the following lemma, the marking phase of Algorithm 1 correctly computes the state equivalence.

Lemma 1 ([HMU07, Sha09]) *A pair of states is not marked by the marking phase of Algorithm 1 if and only if the states are equivalent.*

Using Lemma 1, the correctness of Algorithm 1 can be established.

Theorem 1 ([HMU07]) *The automaton $\hat{A}$ produced by Algorithm 1 accepts the same language as A and is minimal.*

In the next section, we present an implementation of Algorithm 1 in GP. The correctness of the implementation is proved in Section 5.

4 Implementation in GP

We represent automata by their transition diagrams, that is, graphs in which nodes represent states and edges represent transitions. In the following, the terms 'node' and 'state', respectively 'edge' and 'transition' will often be used synonymously. We make the following assumptions about an input automaton:

1. The states have labels of the form x_i, where x is some integer and $i \in \{0,1\}$. The component i is called a *tag*[1], we require that final states have tag 1 and that non-final states have tag 0. The integer x is arbitrary, except that the initial state, and only this state, has a label of the form 1_i.

2. The transitions are labelled with strings which represent the symbols in Σ.

3. To keep the presentation simple, we assume that all states are reachable from the initial state. (It is straightforward to write a graph program that removes all unreachable states.)

The graph program implementing Algorithm 1 is shown in Figure 3, where `mark`, `merge` and `clean_up` are *macros*. The rule schemata contained in the macros are discussed below.

```
main = mark; merge; clean_up

mark = distinguish!; propagate!; equate!
merge = init; add_tag!; (choose; add_tag!)!; disconnect!; redirect!
clean_up = remove_edge!; remove_node!; untag!
```

Figure 3: GP program for automata minimization

We will explain each stage of the program in Figure 3, using as running example the minimization of the automaton in Figure 4. This automaton accepts all strings over $\{\mathtt{a},\mathtt{b}\}$ that end in two `b`'s.

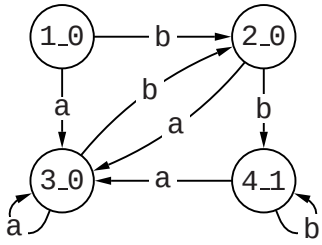


Figure 4: Sample automaton with alphabet $\{\mathtt{a},\mathtt{b}\}$

4.1 Marking Phase

We first need to determine which states are equivalent. For this, we implement the marking phase of Algorithm 1 in the macro `mark`. The macro's rule schemata are shown in Figure 5.

The subprogram `distinguish!` implements Stage 1 of Algorithm 1. Given two states such that one is a final state and the other is not, by assumption, the states carry tags `1` and `0` respectively. In this case we mark the states as distinguishable by connecting them with two `1`-labelled edges of opposite direction (drawn as a single edge with two arrowheads). The condition `not edge(1,2,1)` in `distinguish` forbids a `1`-labelled edge between nodes `1` and `2` to make sure that `distinguish!` terminates. The ternary `edge` predicate refines the binary predicate

[1] In general, a label in GP has the form $x_1\_x_2\_\ldots\_x_n$ where each x_i is either an integer or a character string.

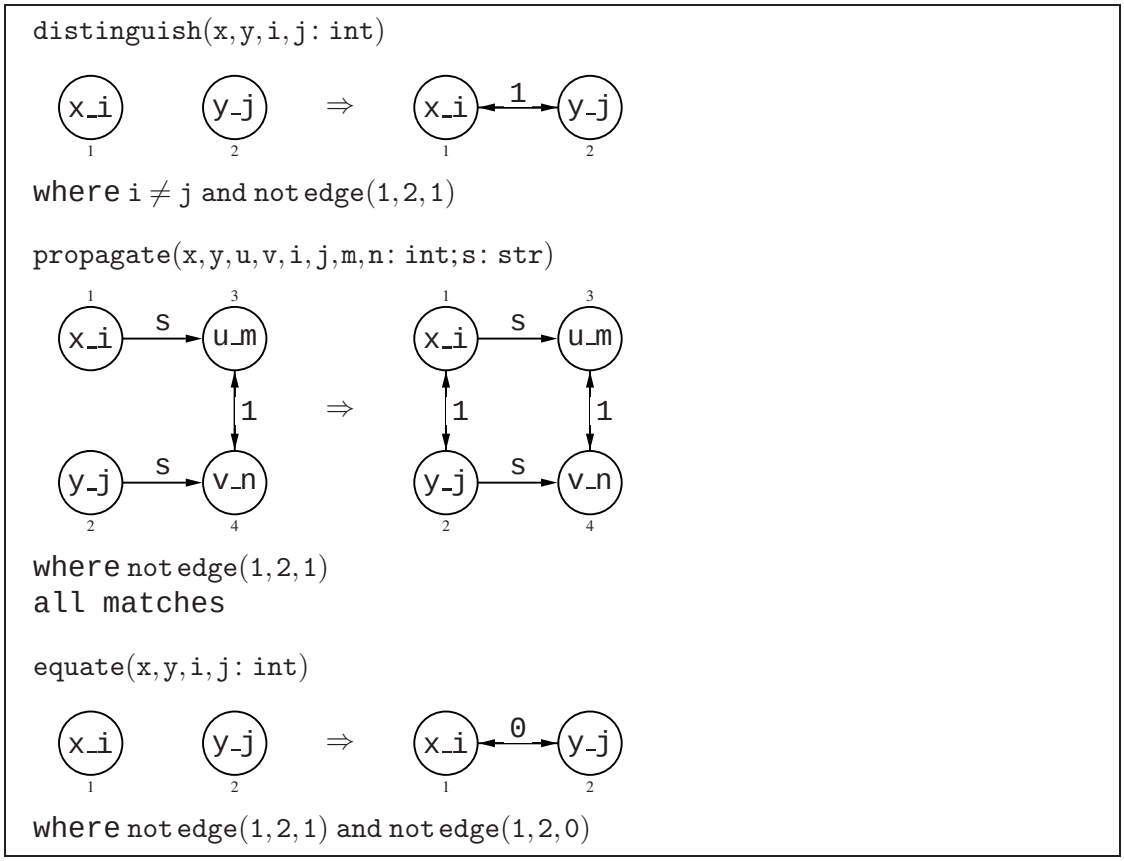


Figure 5: Rule schemata of the macro `mark`

discussed in Section 2 in that it allows to specify the label of the forbidden edge.[2] See Figure 6 for the effect of `distinguish!` on the sample automaton, where we typeset new labels in italics.

Next, the rule schema `propagate` looks for pairs of states that have not yet been discovered as distinguishable (and so are not linked by a `1`-edge). The states must have outgoing transitions with the same symbol, leading to states that have already been discovered as distinguishable. Again, a newly discovered pair of distinguishable states is marked by `1`-labelled edges with opposite directions. The subprogram `propagate!` thus implements the repeat-loop of Algorithm 1.

Rule schema `propagate` has the '`all matches`' attribute, meaning that nodes of the schema can be merged before the schema is applied. An alternative view is that `propagate` can be applied using non-injective graph morphisms. (See [HMP01] for details and the equivalence of both views.) For the benefit of the reader, Figure 7 lists the standard rule schemata represented by `propagate` that are possibly applicable to an automaton. Other schemata obtained by node merging can be ruled out because our automata do not contain `1`-labelled loops and do not have

[2] This predicate is not yet implemented in GP but will be included in the next release.

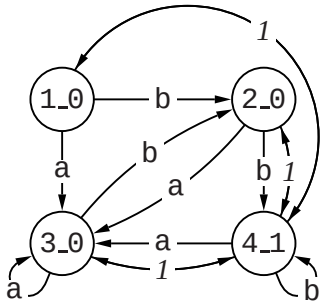


Figure 6: Sample automaton after `distinguish!`

states with multiple outgoing transitions labelled with the same symbol.

Lemma 1 guarantees that after termination of `propagate!`, all pairs of distinguishable states have been discovered. Thus we can mark the remaining pairs as equivalent, linking their states with `0`-labelled edges in the subprogram `equate!`. The effect of `propagate!` and `equate!` on the sample automaton is shown in Figure 8a and Figure 8b. We remark that `0`-edges create a structure similar to the "equivalent states layer" in the FIRE Station tool for regular language visualisation [FCW05].

4.2 Merging Phase

After termination of the macro `mark`, the states of the input automaton are partitoned into equivalence classes: these are the subsets of states that are pairwise linked by `0`-labelled edges. Next we have to merge all the states in each partition into one state representing the partition. We need to ensure that all transitions to states that are not representing partitions are redirected to the unique states representing the partitions. Transitions outgoing from non-representative states can be removed, as can these states themselves. The merging process is implemented by the macro `merge`, whose rule schemata are shown in Figure 9.

We first consider the partition containing the initial state. The rule schema `init` marks this state as the unique representative of its partition by adding an extra `0`-tag to the state's label. Then the loop `add_tag!` marks all other states in the initial partition with an extra `1`-tag. This marking procedure is repeated for all other partitions, by the nested loop `(choose; add_tag!)!`. In each iteration of the outer loop, some unmarked state is chosen as the unique representative of its partition and subsequently all other states in the partition are marked as non-representative states.

After all states have been marked as representatives or non-representatives, the rule schemata `disconnect` and `redirect` take care of the transitions leaving and reaching non-representative states. The loop `disconnect!` removes all outgoing transitions (including loops), as these are no longer needed, while `redirect!` redirects each transition reaching a non-representative state to the unique representative of that state's partition. Note that by the '`all matches`' attribute of `redirect`, transitions between equivalent states become loops at the representatives. The effect of `init; add_tag!` and the whole macro `merge` on the sample automaton is shown in Figure 8c and Figure 8d.

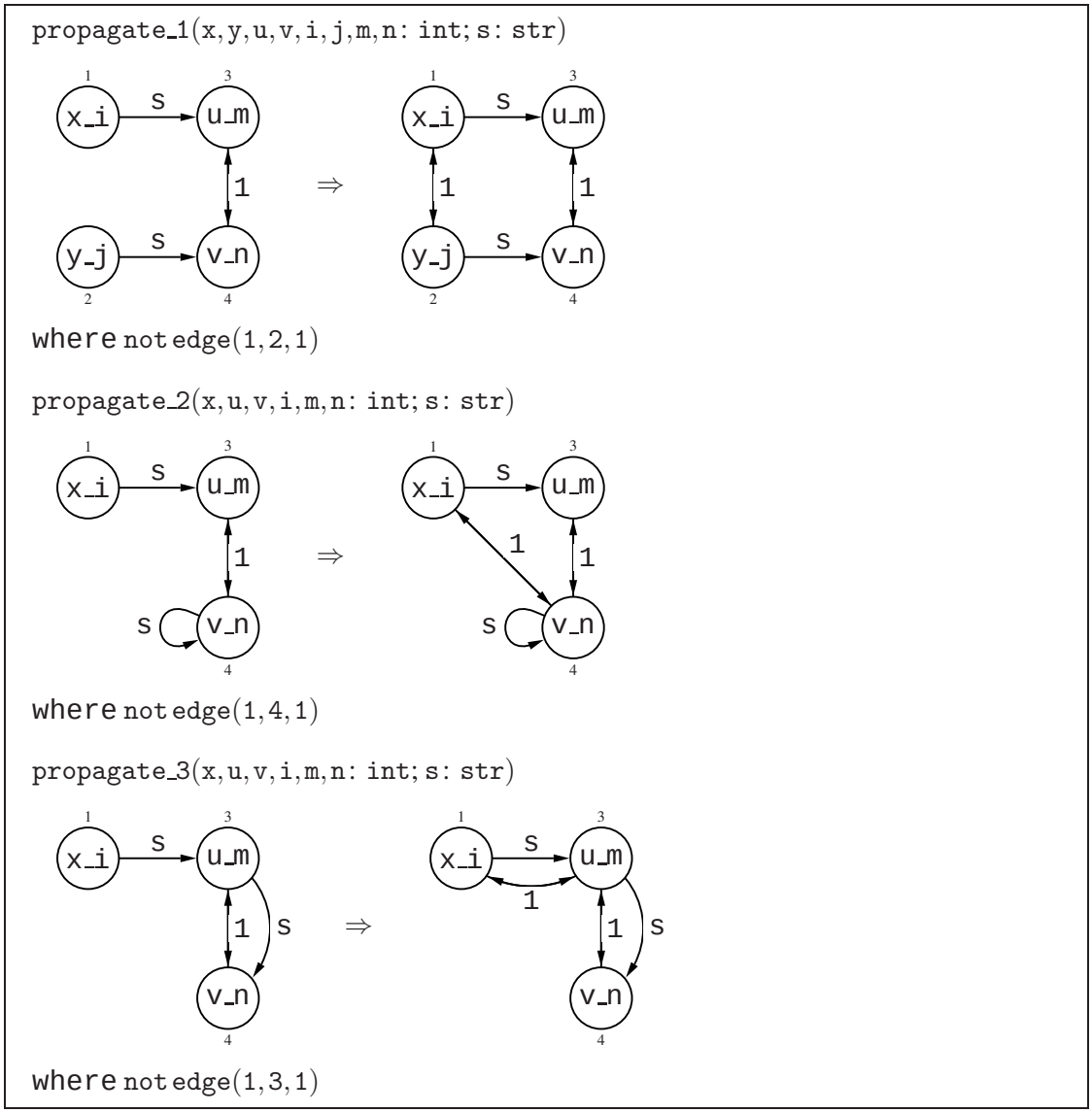
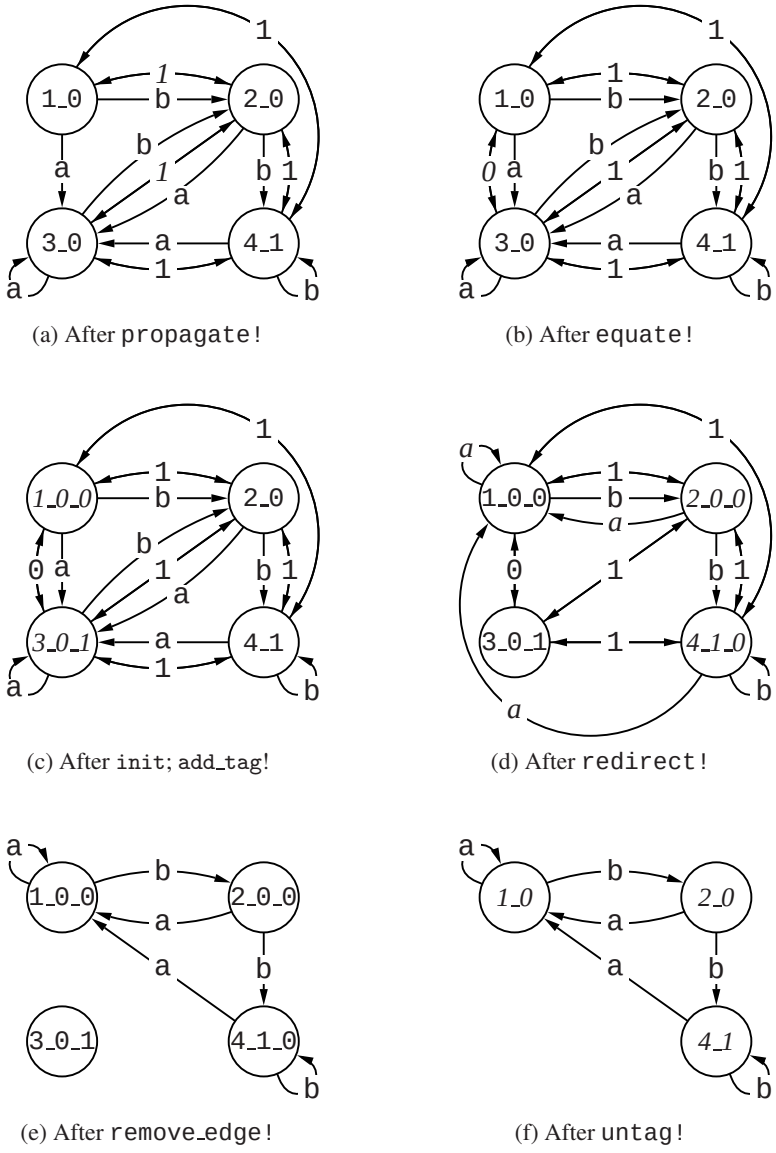


Figure 7: Rule schemata represented by `propagate` using '`all matches`'

Finally, the rule schema `clean_up` exhaustively applies the rule schemata shown in Figure 10. The loop `remove_edge!` deletes all integer-labelled edges, as these auxiliary structures are no longer needed. Then `remove_node!` deletes all non-representative states—these states have become isolated. The remaining states are the unique representatives of their equivalence classes. Last but not least, `untag!` removes the auxiliary second tag of each state so that the remaining tag indicates, as before, whether a state is final or not. The resulting automaton is the unique minimal automaton equivalent to the input automaton (see next section). The automata resulting from `remove_edge!` and the overall program in our running example are shown in Figure 8e and Figure 8f.



(a) After `propagate!`

(b) After `equate!`

(c) After `init`; `add_tag`!

(d) After `redirect!`

(e) After `remove_edge!`

(f) After `untag!`

Figure 8: Snapshots of the sample automaton

5 Correctness of the Implementation

In this section we argue that the graph program of Figure 3 correctly implements Algorithm 1.

Lemma 2 *The program of Figure 3 terminates for every input automaton.*

Proof. By the conditions of the rule schemata `distinguish` and `propagate`, each application of these schemata reduces the number of state pairs that are not linked by 1-labelled edges

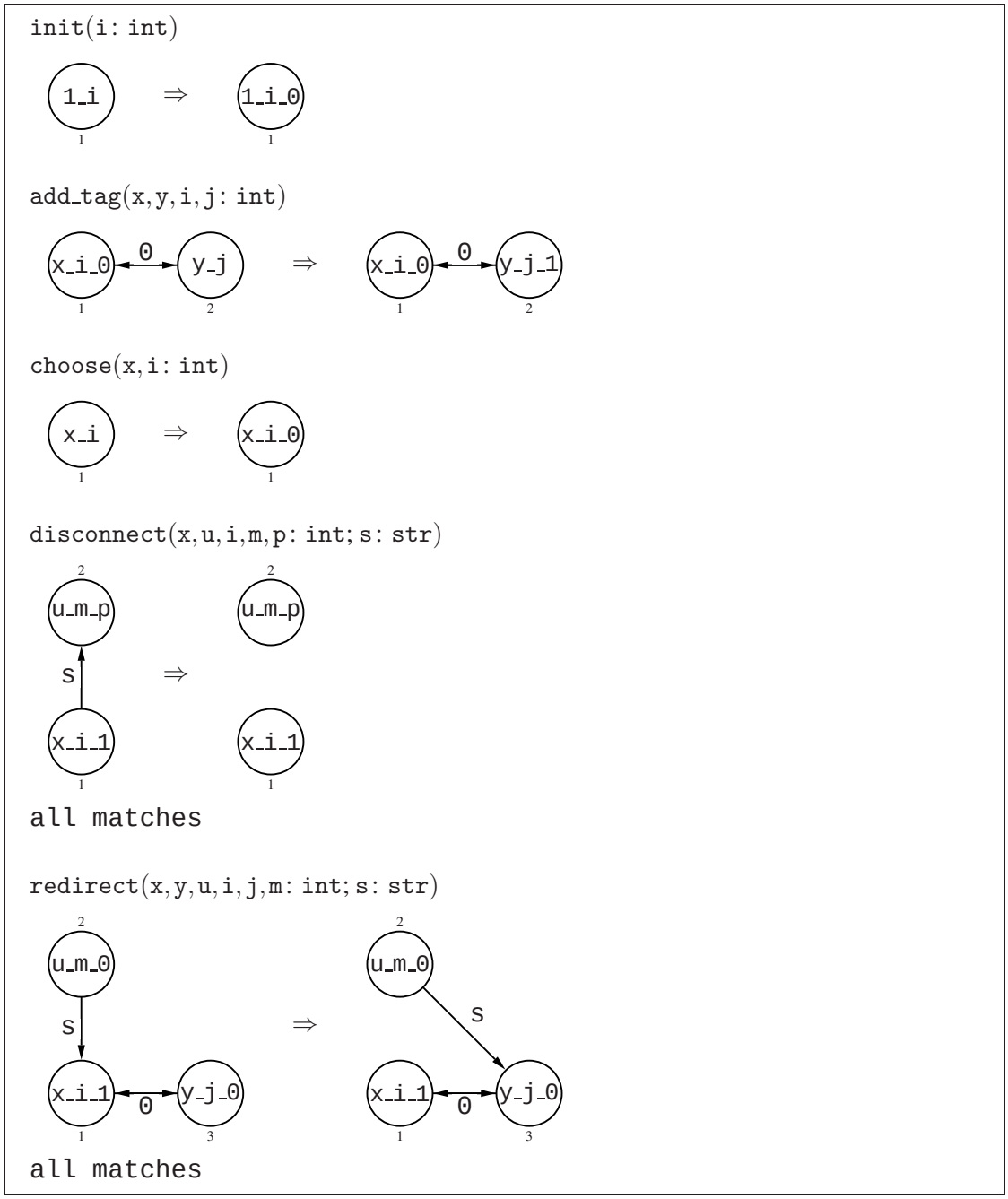


Figure 9: Rule schemata of the macro `merge`

of opposite direction. Similarly, each application of `equate` reduces the number of state pairs that are not linked by 0-labelled edges of opposite direction. Thus the macro `mark` terminates.

Each application of the rule schema `add_tag` reduces the number of states that do not have

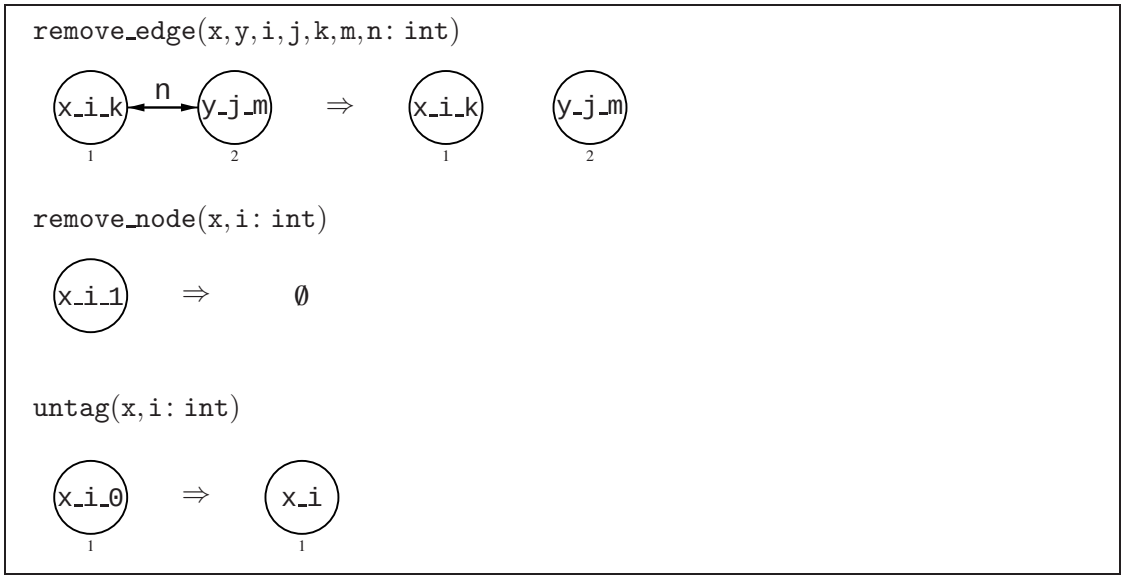


Figure 10: Rule schemata of the macro `clean_up`

a label of the form *x_i_*1, where *x* and *i* are integers. Hence both the first loop `add_tag!` and the nested loop `(choose; add_tag!)!` terminate (note that `choose` does not affect labels of the form *x_i_*1). The loop `disconnect!` is trivially terminating as each application of `disconnect` reduces the number of edges in a graph. The loop `redirect!` terminates because each application of `redirect` reduces the sum of the degrees of nodes with a label of the form *x_i_*1. Thus the macro `merge` terminates, too.

The termination of the three loops in the macro `clean_up` is similarly easy to see. The rule schemata of the first two loops reduce the number of edges respectively the number of nodes, and each iteration of the loop `untag!` reduces the number of nodes with three tags. □

Lemma 3 *The macro* `mark` *links two distinct states by a* `0`*-labelled edge if and only if the states are equivalent.*

Proof. The loop `distinguish!` implements stage 1 of the marking phase of Algorithm 1 in that it links final states with non-final states by a `1`-labelled edge, marking such pairs as non-equivalent. Also, `propagate!` implements stage 2 of the marking phase: the three standard rule schemata represented by `propagate` (see Figure 7) cover the possible relations between the state pairs $\{p,q\}$ and $\{\delta(p,a),\delta(q,a)\}$ in the repeat-loop of Algorithm 1. In particular, they cover the special cases $p=\delta(p,a)$, $q=\delta(q,a)$, $p=\delta(q,a)$ and $q=\delta(p,a)$. Hence Lemma 1 implies that after termination of `propagate!`, two states are linked by a `1`-labelled edge if and only if they are not equivalent. The loop `equate!` then links two distinct states by a `0`-labelled edge if and only if they are not linked by a `1`-labelled edge, implying the proposition. □

Lemma 4 *After termination of the macro* `clean_up`*, two states are equivalent if and only if they are equal.*

Proof. Consider an equivalence class of states of the input automaton. Exactly one state in this class is selected either by the rule schema `init` (in the case of the initial state's class) or by the rule schema `choose` (in all other cases), and a `0`-tag is appended to the state's label. Then the loop `add_tag!` marks all other states in the equivalence class with an extra `1`-tag. Subsequently, `disconnect!` removes all transitions outgoing from `1`-tagged states and `redirect!` redirects away all transitions leading to `1`-tagged states. Hence, after termination of the macro `merge`, `1`-tagged states can be incident only to edges labelled with `0` or `1`. All these edges are deleted by the loop `remove_edge!`, so the `1`-tagged states become isolated and are eventually removed by `remove_node!`. Thus, upon termination of the macro `clean_up`, from each equivalence class exactly one state remains in the resulting automaton. □

Theorem 2 *For every input automaton A, the automaton $\hat{A}$ produced by the program of Figure 3 is equivalent to A and minimal.*

Proof. By Theorem 1, Lemma 2 and Lemma 3, it suffices to show that the subprogram `merge; clean_up` correctly implements the merging phase of Algorithm 1. This can be seen as follows:

- By Lemma 4, each equivalence class of A is represented by its unique representative element in $\hat{A}$.
- The rule schema `init` selects the initial state of A as the representative of its class and `untag` makes this state the initial state of $\hat{A}$.
- Consider any equivalence class of states X, its representative $p \in X$ and any $a \in \Sigma$. If $\delta(p,a)$ is the representative of its equivalence class, then both states are marked with a `0`-tag in `merge` and the transition from p to $\delta(p,a)$ is preserved by the subprogram `disconnect!; redirect!`. Otherwise, if $\delta(p,a)$ does not represent its class, then it is marked with a `1`-tag in `merge`. In this case `redirect!` redirects the transition $p \to \delta(p,a)$ to the unique representative of the class of $\delta(p,a)$. Hence $\hat{\delta}(X,a)$, the equivalence class of $\delta(p,a)$, does not depend on the choice of p and thus is well-defined.
- In an equivalence class containing a final state, all states are final as otherwise the loop `distinguish!` would have linked the non-final states with the final state by `1`-labelled edges. Hence the representative of such a class is a final state.

□

6 Time Complexity

In this section we establish an upper bound for the number of rule schema applications of the minimization program, in terms of the size of the input automaton. This provides a worst-case estimate for the running time of our program, where we abstract from the cost of rule schema matching.[3]

[3] The complexity of rule schema matching is beyond the scope of this paper.

As before, let Σ be the alphabet of an input automaton and Q its set of states. We show that each loop in the program of Figure 3 terminates after at most $|Q|^2$ or $|\Sigma| \cdot |Q|$ rule schema applications. In the following lemmata, n always refers to the number of states (nodes) in an input automaton. Our proofs tacitly rely on the fact that none of the rule schemata of the minimization program increases the number of nodes in a graph.

Lemma 5 *The loops* `distinguish!`, `propagate!` *and* `equate!` *each terminate after at most* n^2 *rule schema applications.*

Proof. Given a graph X, let $\#X$ be the number of pairs $\langle u, v \rangle$ of nodes such that there is no edge with label 1 from u to v. Then $\#X \leq n^2$ and for every step $G \to_{\texttt{distinguish}} H$ and $G \to_{\texttt{propagate}} H$, we have $\#G > \#H$. This implies the claim for `distinguish!` and `propagate!`. The same argument works for `equate!` if we redefine $\#X$ as the number of pairs $\langle u, v \rangle$ such that there is no edge with label 0 from u to v. □

Lemma 6 *The loops* `add_tag!` *and* `(choose;add_tag!)!` *each terminate after at most n rule schema applications.*

Proof. Given a graph X, let $\#X$ be the number of nodes with a label of the form i_j, where i and j are integers. Then $\#X \leq n$ and every step $G \to_{\texttt{add\_tag}} H$ and $G \to_{\texttt{choose}} H$ satisfies $\#G > \#H$. This implies the claim. □

The complexity of the loops for disconnecting nodes and redirecting edges depends not only on the number of nodes (states) but also on the size of the alphabet Σ.

Lemma 7 *The loops* `disconnect!` *and* `redirect!` *each terminate after at most* $|\Sigma| \cdot n$ *rule schema applications.*

Proof. Each node of an input automaton has $|\Sigma|$ outgoing edges labelled with symbols from Σ (represented as strings), and no rule schema removes or creates such edges before `disconnect!` is executed. Hence `disconnect!` terminates after $|\Sigma| \cdot n$ rule schema applications.

Given a graph X, let $\#X$ be the number of Σ-labelled edges whose target nodes have labels of the form i_j_1 for some integers i and j. Then $\#X \leq |\Sigma| \cdot n$ and every step $G \to_{\texttt{redirect}} H$ satisfies $\#G > \#H$. Hence `redirect!` terminates after at most $|\Sigma| \cdot n$ rule schema applications. □

Lemma 8 *The loop* `remove_edge!` *terminates after at most* n^2 *rule schema applications.*

Proof. The following invariant of the minimization program is easy to prove: in each graph of a computation, each pair of distinct nodes is connected by at most one pair of opposite edges labelled with 1 or 0. (Note that an input automaton does not possess such edges.) This invariant clearly implies the claim. □

Lemma 9 *The loops* `remove_node!` *and* `untag!` *each terminate after at most n rule schema applications.*

Proof. The claim is obvious in the case of `remove_node`. For `untag`, it follows from the fact

that every step $G \to_{\texttt{untag}} H$ reduces the number of nodes labelled i_j_k for some integers i, j and k. □

Summarising the above lemmata, we can see that the number of rule schema applications used by the minimization program is quadratic in the size of the input automaton.

Theorem 3 *The program of Figure 3 terminates after at most* $O(|Q|^2 + |\Sigma| \cdot |Q|)$ *rule schema applications.*

7 Conclusion

We have shown how to minimize finite automata with rule-based, visual programming. Programmers need not be concerned with low-level data structures such as state tables but can directly manipulate the transition diagrams of automata. Moreover, GP's rule schemata and control constructs provide a convenient language for reasoning about the correctness and the complexity of the implementation. Last but not least, the `all matches` option for rule schemata has proved to be useful for keeping the number of rule schemata small, and an extended `edge` predicate has been crucial for forbidding particular edges in the conditions of rule schemata.

The macro `merge` merges equivalent states by choosing representatives of equivalence classes, removing and redirecting transitions, and removing isolated states. A simpler implementation would use non-injective rule schemata to merge states directly—but such rule schemata are not available in GP. Non-injective rule schemata are also useful in other applications and may be realised in a future version of GP.

Finally, this case study could be extended by implementing more efficient automata minimization algorithms. We chose the algorithm of Hopcroft, Motwani and Ullman because of its simplicity, but its cubic running time is not optimal. More efficient algorithms include the quadratic algorithm of Hopcroft and Ullman [HU79] and Hopcroft's $n\log n$ algorithm [Hop71].

Acknowledgements: We are grateful for the comments of the anonymous referees which helped to improve the presentation of this paper.

Bibliography

[FCW05] M. Frishert, L. G. Cleophas, B. W. Watson. FIRE Station: An Environment for Manipulating Finite Automata and Regular Expression Views. In *Implementation and Application of Automata (CIAA 2004), Revised Selected Papers*. Lecture Notes in Computer Science 3317, pp. 125–133. Springer-Verlag, 2005.

[HMP01] A. Habel, J. Müller, D. Plump. Double-Pushout Graph Transformation Revisited. *Mathematical Structures in Computer Science* 11(5):637–688, 2001.

[HMU07] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.

[Hop71] J. E. Hopcroft. An *n*log*n* algorithm for minimizing the states in a finite automaton. In Kohavi (ed.), *The Theory of Machines and Computations*. Pp. 189–196. Academic Press, 1971.

[HP02] A. Habel, D. Plump. Relabelling in Graph Transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*. Lecture Notes in Computer Science 2505, pp. 135–147. Springer-Verlag, 2002.

[HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[MP08] G. Manning, D. Plump. The GP Programming System. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*. Electronic Communications of the EASST 10. 2008.

[Plu09] D. Plump. The Graph Programming Language GP. In *Proc. Algebraic Informatics (CAI 2009)*. Lecture Notes in Computer Science 5725, pp. 99–122. Springer-Verlag, 2009.

[PS10] D. Plump, S. Steinert. The Semantics of Graph Programs. In *Proc. Rule-Based Programming (RULE 2009)*. Electronic Proceedings in Theoretical Computer Science 21, pp. 27–38. 2010.

[PSS10] D. Plump, R. Suri, A. Singh. Minimizing Finite Automata with Graph Programs. In *Proc. Graph Computation Models (GCM 2010)*. CTIT Workshop Proceedings WP 2010-05, pp. 97–110. University of Twente, 2010.

[Sha09] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2009.