Electronic Communications of the EASST
Volume 54 (2012)

Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

Rooted Graph Programs

Christopher Bak and Detlef Plump

12 pages

Guest Editors: Christian Krause, Bernhard Westfechtel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Rooted Graph Programs

Christopher Bak and Detlef Plump

The University of York, UK

Abstract: We present an approach for programming with graph transformation rules in which programs can be as efficient as programs in imperative languages. The basic idea is to equip rules and host graphs with distinguished nodes, so-called roots, and to match roots in rules with roots in host graphs. This enables graph transformation rules to be matched in constant time, provided that host graphs have a bounded node degree (which in practice is often the case). Hence, for example, programs with a linear bound on the number of rule applications run in truly linear time. We demonstrate the feasibility of this approach with a case study in graph colouring.

Keywords: Graph programs, rooted graphs, time complexity, constant-time graph matching, graph colouring

1 Introduction

The bottleneck for using graph transformation rules in programming is the inefficiency of graph matching. In general, to match the left-hand graph L of a rule within a host graph G requires time $\text{size}(G)^{\text{size}(L)}$. As a consequence, linear graph algorithms are slowed down to polynomial complexity when they are recast as programmed graph transformation systems.

One way to speed up graph matching, going back to Dörr [5], is to equip rules and host graphs with distinguished nodes, so-called roots, and to match roots in rules with roots in host graphs. The same idea underlies Fujaba's requirement that each method must have a "this" node at which graph matching starts [8, 12]. A related concept in GrGen are rules that return graph elements to restrict the location of subsequent rule applications [6].

Dodds and Plump [4, 2] have considered rooted graph transformation by using uniquely labelled nodes as roots. They show that graph matching can be achieved in constant time if rules have a connected left-hand graph and host graphs have bounded node degrees. In addition, they use rooted rules in a rule-based extension of C that allows to check the shape safety of pointer manipulations [3]. In this paper, we generalise the approach of [4, 2] from plain rules to programs in the graph programming language GP 2 [10]. We extend GP with rooted rule schemata and present a matching algorithm which deals with the label expressions in these schemata.

Our main contribution is to identify *fast* rule schemata, a large class of rooted conditional rule schemata, and to prove that they can be applied in constant time if host graphs have a bounded node degree. In practice, the latter assumption is often satisfied. For example, traffic networks, digital circuits and social networks usually have an upper bound on the number of edges attached to nodes. In Section 6, we apply fast rule schemata in a case study on graph colouring. We give a GP program which checks whether the input graph is 2-colourable and, if this is the case, colours the graph. We prove that this program runs in time linear in the size

of input graphs, demonstrating that rooted GP programs can achieve the time complexity of programs in imperative languages.

2 Graph Transformation

We first recall the graph transformation approach underlying GP, namely the double-pushout approach with relabelling [7], and then accommodate this framework to rooted graphs.

2.1 Non-rooted graph transformation

A (partially labelled) *graph* G is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ where V_G is a finite set of *nodes*, E_G is a finite set of *edges*, s_G and t_G are functions that assign to each edge a source and a target node respectively, l_G is the partial node-labelling function and m_G is the total edge-labelling function. We write $l_G = \bot$ if $l_G(v)$ is undefined. Both node and edge labels are taken from a fixed label set $\mathscr{L}$. Unlabelled nodes are used in rules to relabel nodes (see below). There is no need to relabel edges because they can be deleted and reinserted with a new label.

A node w is *reachable* from a node v if $v = w$ or there are an edge e and a node v' such that v and v' are incident to e and w is reachable from v'. (Note that this defines undirected reachability.) An edge e is reachable from v if the source and target of e are reachable from v. A graph is *connected* if every node is reachable from every other node.

Given graphs G and H, a *premorphism* $g\colon G \to H$ is a pair of functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources and targets. That is, for all edges e in G, $s_H(g_E(e)) = g_V(s_G(e))$ and $t_H(g_E(e)) = g_V(t_G(e))$. If g also preserves labels, that is $m_H(g_E(e)) = m_G(e)$ for all edges e and $l_H(g_V(v)) = l_G(v)$ for all nodes v with $l_G(v) \neq \bot$, then g is a *morphism*. A morphism whose node and edge functions are both injective and surjective is an *isomorphism*. If g satisfies $g(x) = x$ for all nodes and edges x, then g is an *inclusion*.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ is a pair of inclusions $K \to L$ and $K \to R$ where L and R are totally labelled graphs. We refer to L, R and K as the *left-hand side*, the *right-hand side* and the *interface*, respectively.

Given a graph G and a rule $r = \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $g\colon L \to G$ satisfies the *dangling condition* if no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case G *directly derives* graph H, denoted by $G \Rightarrow_{r,g} H$ or just $G \Rightarrow_r H$, if H can be constructed from G as follows:

1. Obtain a subgraph D by removing all nodes and edges in $g(L) - g(K)$.
2. Add (disjointly) the nodes and edges of $R - K$ to D, keeping all labels. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.
3. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is H.

Note that H is specified only up to isomorphism, that is, every graph isomorphic to H qualifies as a result of the rule application. Abstractly, a direct derivation can be defined by a pair of natural pushouts in the category of partially labelled graphs; we refer to [7] for this characterisation.

2.2 Rooted graph transformation

We extend the above definitions to include distinguished root nodes in both rules and host graphs. Our approach is to treat rooted graphs and root-preserving morphisms as "first-class citizens" instead of encoding roots with labels. Unlike [4, 2], we allow multiple roots in rule schemata and host graphs; this may be useful in applications with disconnected host graphs.

A *rooted graph* is a pair $\langle G, P_G \rangle$ where G is a graph and $P_G \subseteq V_G$ is a set of *roots*. A morphism $g : G \to H$ is *root-preserving* if $g(P_G) \subseteq P_H$. Note that rooted graphs (over some label set) and root-preserving morphisms form a category.

A *rooted rule* $r = \langle \langle L, P_L \rangle \leftarrow \langle K, P_K \rangle \rightarrow \langle R, P_R \rangle \rangle$ is a pair of root-preserving inclusions $\langle K, P_K \rangle \to \langle L, P_L \rangle$ and $\langle K, P_K \rangle \to \langle R, P_R \rangle$ where L and R are totally labelled. Given a rooted graph G and a root-preserving injective morphism $g\colon L \to G$ satisfying the dangling condition, a direct derivation $G \Rightarrow_{r,g} H$ is constructed as above and by defining $P_H = (P_G - g_V(P_L - P_K)) \cup (P_R - P_K)$. This construction can be characterised by a pair of natural pushouts in the category of rooted graphs and root-preserving morphisms (omitted for lack of space).

3 Rooted Graph Programs in GP

We extend the graph programming language GP with rooted programs. A complete definition of GP and its revised version GP 2 is given in [9, 10]. In this section, we describe GP's most important features informally.

3.1 Conditional Rule Schemata

GP's principal programming constructs are conditional rule schemata. These extend the rules of Subsection 2.1 with expressions in labels and with application conditions. For example, Figure 1 shows the declaration of a conditional rule schema `bridge`, where roots are depicted as nodes with bold borders. Only the left- and right-hand side of the rule schema are declared. By convention, the interface is the unlabelled and rootless graph consisting of the numbered nodes.

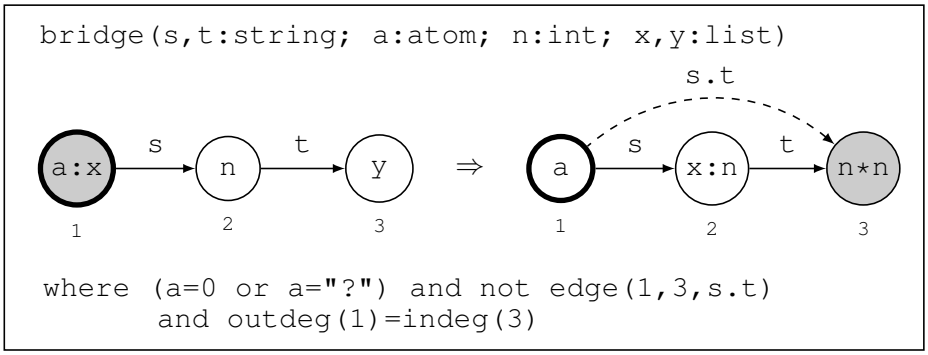

Figure 1: Declaration of a conditional rule schema

The top line of the declaration states the name of the rule schema and declares the variables that are used in the labels and in the condition. All variables occurring in the right-hand side and

in the condition must also occur in the left-hand side because their values at execution time are determined by matching the left-hand side with a subgraph of the host graph.

Each variable is declared with a type which is either `int`, `string`, `atom` or `list`. Types form a subtype hierarchy in which integers and character strings are basic types, both of which are atoms, which in turn are considered as lists of length one. In general, a label in GP is a list of atoms each of which is either an integer or a character string. Labels in host graphs do not contain expressions; they are fixed values in $(\mathbb{Z} \cup \mathrm{Char}^*)^*$, where Char is the set of available characters.

Lists are constructed by the colon operator which represents list concatenation. For example, the label of node 1 on the left of Figure 1 stands for a list whose first element `a` is an integer or a character string, followed by a (possibly empty) rest list `x`. String concatenation is signified by the dot operator, as in the edge label `s.t` on the right of Figure 1. Labels in the right-hand side of a rule schema may contain arithmetic expressions such as `n*n` in node 3 on the right of Figure 1.

Besides having labels, both nodes and edges can be *marked*. Graphically, a marked node is shaded, and a marked edge is dashed. Marked items in rule schemata can only match marked items in host graphs. Vice versa, marked items in host graphs can only be matched by marked items in rule schemata. Marks are used as boolean flags and should not be confused with roots.

Rule schemata have an optional condition, declared with the keyword `where`. The condition is a boolean expression containing built-in predicates and functions, label expressions, and node identifiers. For example, the subexpression `not edge(1,3,s.t)` in Figure 1 demands that there must not be an edge in the host graph from node 1 to node 3 that is labelled with the string `s.t` (where `s` and `t` stand for the host graph labels of the edges in the left-hand side). To apply the rule schema according to a match of the left-hand side, the condition must evaluate to true for that match and its induced assignment of values to variables.

3.2 Programs

GP programs consist of a finite number of rule schemata and a command sequence which controls their application to a host graph (see Figure 3 for an example program). The main control constructs are: application of a set of conditional rule schemata $\{r_1, \ldots, r_n\}$, where one of the applicable schemata in the set is nondeterministically chosen or otherwise the command fails; sequential composition $P;Q$ of programs P and Q; as-long-as-possible iteration $P!$ of a program P; and conditional branching statements `if` C `then` P `else` Q and `try` C `then` P `else` Q, where C, P and Q are arbitrary command sequences.

To execute `if` C `then` P `else` Q on a state G (the current graph), first the condition C is executed on G. If this produces a graph, then this result is disposed and P is executed on state G. Alternatively, if C fails on G, then Q is executed on G. This behaviour makes it possible to encode complex tests in the condition C which do not alter the current state.

The command `try` C `then` P `else` Q also first executes C on G and, if this fails, executes Q on G. However, if C produces a graph H, then P is executed on H rather than on G.

GP also allows to define (non-recursive) macros, which are command sequences represented by identifiers. For example, the program in Figure 3 contains the macro `colouring`. Macros are used for better readability but have no semantic significance.

4 A Matching Algorithm for Rooted Rule Schemata

We present a matching algorithm for rooted rule schemata which extends the corresponding algorithms in [4, 2]. The main difference is that instead of matching plain graph transformation rules, we have to deal with the syntax of GP 2-rule schemata. In particular, the algorithm must compare label expressions of the left-hand side with values in host graph labels and, besides finding matches of the graph structure, compute assignments of values to variables. These assignments are used both in evaluating the application condition of the rule schema and in calculating the labels of new and relabelled items when the rule schema is applied. Another extension to the previous algorithms is that we allow multiple roots in rule schemata and host graphs, while the cited papers assume a single root. Moreover, it is now possible to designate arbitrary nodes as roots whereas before a root had to be identified by a uniquely occurring label.

First we introduce some notation used in the algorithm. A *partial premorphism* $g\colon G \to H$ is a pair of partial functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ such that for each edge e in G, if $g_E(e)$ is defined then $g_V(s_G(e))$ and $g_V(t_G(e))$ are also defined and satisfy $s_H(g_E(e)) = g_V(s_G(e))$ and $t_H(g_E(e)) = g_V(t_G(e))$. We write $\mathrm{Dom}(g_V)$ and $\mathrm{Dom}(g_E)$ for the sets of nodes and edges on which g is defined. Given partial premorphisms $f,g\colon G \to H$, *f extends g by a node v* if $\mathrm{Dom}(f_V) = \mathrm{Dom}(g_V) \cup \{v\}$ and $\mathrm{Dom}(f_E) = \mathrm{Dom}(g_E)$. Also, *$f$ extends g by an edge e* if $\mathrm{Dom}(f_E) = \mathrm{Dom}(g_E) \cup \{e\}$ and $\mathrm{Dom}(f_V) = \mathrm{Dom}(g_V) \cup \{s_G(e), t_G(e)\}$. Given a rooted graph $\langle L, P_L\rangle$ and $p \in P_L$, an *edge enumeration* for p is a list of edges $e_1, \ldots, e_n$ such that $\{e_1, \ldots, e_n\}$ is the set of all edges reachable from p, e_1 is incident to p, and for $i = 2, \ldots, n$, e_i is incident to the source or target of some edge in $\{e_1, \ldots, e_{i-1}\}$.

Algorithm Rooted Graph Matching
$A \leftarrow \{\langle h\colon L \xrightarrow{par} G, \emptyset\rangle \mid \mathrm{Dom}(h) = \emptyset\}$
while there exists an untagged root $p \in P_L$ **do**
  $A_0 \leftarrow \{\langle h\colon L \xrightarrow{par} G, \alpha_{h'}\rangle \mid h$ is injective and root-preserving, and
    there exists $\langle h', \alpha_{h'}\rangle$ in A such that h extends h' by $p\}$
  tag p
  AssignmentUpdate(A_0)
  for $i = 1$ to n **do**
    $A_i \leftarrow \{\langle h\colon L \xrightarrow{par} G, \alpha_{h'}\rangle \mid h$ is injective and root-preserving, and
      there exists $\langle h', \alpha_{h'}\rangle$ in A_{i-1} such that h extends h' by $e_i\}$
    if $s(e_i) \in P_L$ **then** tag $s(e_i)$
    if $t(e_i) \in P_L$ **then** tag $t(e_i)$
    AssignmentUpdate(A_i)
  end for
  $A \leftarrow A_{p_n}$
end while
return A

Figure 2: Algorithm Rooted Graph Matching

Given a rooted host graph $\langle G, P_G\rangle$, the left-hand side $\langle L, P_L\rangle$ of a fixed rooted rule schema, and an edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$, the algorithm in Figure 2 computes all

matches of $\langle L, P_L \rangle$ in $\langle G, P_G \rangle$. The algorithm assumes that each node in L is reachable from some root. It incrementally constructs a set A of pairs of partial premorphisms $h\colon L \xrightarrow{par} G$ and partial assignments α_h. By a *partial assignment* we mean a partial function $\mathrm{Var}(L) \to (\mathbb{Z} \cup \mathrm{Char}^*)^*$, where $\mathrm{Var}(L)$ is the set of variables occurring in L. The roots in L are tagged whenever they are matched; initially they are all untagged.

The algorithm calls the procedure AssignmentUpdate, which exploits that expressions in the left-hand side of a GP rule schema are constrained to prevent ambiguous variable assignments. Expressions must not contain arithmetic operators, more than one occurrence of a list variable, or more than one occurrence of a string variable in a single string expression.

Lists and strings are stored internally as doubly-linked lists with pointers *first* and *last* pointing to the first and last element. Hence the first and last element of a list or string, as well as the predecessor and successor of the current element, can be accessed in unit time. With this data structure, only the pointers *first* and *last* are needed when assigning a value to a list, atom or string variable, as the rest of the list or string can be accessed through the *next* and *prev* operators.

AssignmentUpdate is omitted for lack of space; we give a brief outline of its operation. The procedure iterates over its input, a set of pairs of partial premorphisms and partial assignments. For each pair $\langle h, \alpha \rangle$, it iterates over all untested labels l in the domain of h. Each l and corresponding host graph value $h(l)$ are evaluated by a local procedure which will also update the partial assignment if l contains a variable and the two expressions can be matched. In particular, expressions containing a list variable or a string variable are tested by comparing the individual components (atoms or characters) that occur before and after the single variable. If all components match, then the variable has a unique mapping. This mapping is specified by assigning locations to the *first* and *last* pointers of the string or list variable. This is sufficient; the rest of the string or list can be accessed through the list operators as the value is stored as a doubly linked list in the graph data structure.

Proposition 1 (Correctness of Rooted Graph Matching) *The algorithm Rooted Graph Matching returns the set of all pairs $\langle g, \alpha \rangle$ where $g\colon L \to G$ is an injective root-preserving premorphism and $\alpha\colon \mathrm{Var}(L) \to (\mathbb{Z} \cup \mathrm{Char}^*)^*$ is a total assignment such that $g\colon L^\alpha \to G$ is label-preserving.*

Here L^α is the graph obtained from L by replacing each variable x with the value $\alpha(x)$. According to the semantics of GP 2 [10], g must be label-preserving after this replacement, that is, it must be a graph morphism $L^\alpha \to G$. We omit the proof of Proposition 1 for lack of space.

5 Complexity of Rooted Rule Schemata

In this section, we analyse the complexity of the rooted graph matching algorithm and of applying a conditional rule schema with a given match. We make the following general assumption.

Assumption 1 (Complexity model)
When analysing the complexity of rule schemata and programs, we assume that

1. rule schemata and programs are fixed, and
2. integer operations and character comparisons are computed in unit time.

The first assumption is customary in algorithm analysis where programs are fixed and running time is measured in terms of input size. In our setting, programs consist of fixed rule schemata and the input size is the size of a host graph and its labels.[1] The second assumption is consistent with the uniform cost criterion for random access machines, the standard complexity model in algorithm analysis [1, 11].

Our matching algorithm assumes that each node in a left-hand graph is reachable from some root. This alone does not guarantee that rule schemata can be applied in time independent of the host graph. To achieve this, we need to impose some more restrictions on the form of rooted rule schemata. We will show that, under mild assumptions on host graphs, rule schemata of the following form can be applied in constant time.

Definition 1 (Fast rule schema)
A rule schema $L \Rightarrow R$ with application condition c is *fast* if

1. each node in L is reachable from some root,
2. L and R do not contain repeated list, string or atom variables, and
3. c does neither contain the `edge` predicate nor a test $e_1{=}e_2$ or $e_1{!}{=}e_2$ where both e_1 and e_2 contain a list, string or atom variable.

The first condition ensures that matches can only occur in the neighbourhood of roots. The second condition makes it unnecessary to check the equality of lists or strings, or to copy lists or strings. The third condition rules out tests that require more than constant time.

Applying a conditional rule schema $L \Rightarrow R$ to a host graph G requires several phases: finding a root-preserving match of L in G and constructing the induced variable assignment; checking the dangling condition and the application condition; removing items from $L - K$; adding items from $R - K$; and relabelling nodes. In the following we focus on the complexity of the matching phase because, in the worst case, it is far slower than the other phases.

We give the following lemma without proof due to lack of space.

Lemma 1 *Given a fast rule schema $L \Rightarrow R$ and a host graph G, the procedure AssignmentUpdate compares each label in L in constant time with the corresponding label in G.*

We can now show that fast rule schemata can be matched in constant time, provided that both node degrees and the number of roots in host graphs are bounded. The *degree* of a node v is the sum of the number of edges with source v and the number of edges with target v.

Theorem 1 *The algorithm Rooted Graph Matching runs in constant time for fast rule schemata if there are upper bounds on the maximal node degree and the number of roots in host graphs.*

Proof. Consider a fast rule schema $L \Rightarrow R$ and a host graph G. Let l be the number of roots in L. Let b and r be upper bounds on the maximal node degree and the number of roots in host graphs respectively.

First we count the number of times the set of partial premorphisms $L \xrightarrow{par} G$ is updated. We assume a data structure where adding either a node, an edge, or a node and an edge to an existing

[1] We need to consider the size of labels because they can contain arbitrarily large lists and strings.

morphism takes unit time. There are at most l iterations of the while loop and, within each iteration, at most $m = |E_L|$ iterations of the for loop. Note that, by Assumption 1, both l and m are constants.

Consider the execution of the first iteration of the while loop. First, a single root from L is matched with all unmatched roots in G. Since no roots have been matched yet, r partial morphisms are created. Then, in each iteration, either a single edge or an edge and a node is added to the domain of one of more morphisms in the current set. As node degrees in G are bounded by b, no more than b additions can take place. This gives a worst-case running time of $r + b|A_0| + b|A_1| + \ldots + b|A_{m-1}|$. The set A_0 contains at most r morphisms, A_1 contains at most br morphisms, etc. It follows that the running time is $r + br + b^2r + \ldots + b^m r = r\sum_{i=0}^{m} b^i$.

Next, the second root of L is matched. Since one root in G has already been matched, the maximum size of the new morphism set is $b^m r(r-1)$. Hence, by the same argument as before, the maximal running time after the second iteration of the while loop is

$$r\sum_{i=0}^{m} b^i + r(r-1)\sum_{i=m}^{2m} b^i.$$

After the l-th and final iteration of the while loop, the total running time is bounded by

$$r\sum_{i=0}^{m} b^i + r(r-1)\sum_{i=m}^{2m} b^i + \ldots + r(r-1)\ldots(r-l+1)\sum_{i=(l-1)m}^{lm} b^i.$$

The procedure AssignmentUpdate is called after each update of the set of premorphisms. Each execution checks at most two labels for every premorphism in the set since on each update, at most two new items are added to the domain of the premorphism. Thus, by Lemma 1, it follows that the total execution time of Rooted Graph Matching is bounded by a constant factor of the above expression. □

Given a match of the left-hand side of a fast rule schema, checking the application condition and the dangling condition, and deleting, adding and relabelling items can be done in constant time. Hence we obtain the following corollary of Theorem 1.

Corollary 1 *Fast rule schemata can be applied in constant time if there are upper bounds on the maximal node degree and the number of roots in host graphs.*

Proof sketch. Consider again a fast rule schema $L \Rightarrow R$ with condition c and a host graph G. By Theorem 1, constructing a premorphism $g\colon L \to G$ and induced variable assignment α (or determining there is no such pair) requires only constant time. We need to prove that the remaining phases of rule schema application can be executed in constant time, too.

By Definition 1, the condition c is a boolean combination of subexpressions each of which is either (1) a relational operator applied to integer expressions, (2) a test $e_1{=}e_2$ or $e_1{!}{=}e_2$ where e_1 and e_2 do not both contain list, string or atom variables, or (3) a type check $\texttt{int}(e)$, $\texttt{string}(e)$ or $\texttt{atom}(e)$. Subexpressions of the first kind can be evaluated in constant time by (note that all expressions in c are of constant size). By the same assumption, tests according to (2) take only constant time because no comparisons are made between atom, string or list variables. Type

checks according to (3) can be done in unit time if the data structure for labels records type information suitably.

The dangling condition for an injective premorphism $g\colon L \to G$ can be checked by comparing the degree of each node v in $L-K$ with the degree of its image $g(v)$. We assume a graph representation where nodes are stored together with their indegree and outdegree. This operation then takes time of order $|V_L|$, a constant.

Given a match satisfying the dangling condition, removing the items in $g(L-K)$ can be executed in time proportional to $|L|-|K|$. Similarly, the addition of nodes and edges takes time proportional to $|R|-|K|$.

Finally, relabelling a string or list only requires redirecting the pointers *first* and *last* to a particular label in G. For string concatenation, two more pointer redirections are required to combine the two strings. There are at most $|V_K|$ relabellings, so the time needed is proportional to $|V_K|$. □

The overall time complexity of a fast rule schema is largely determined by the number of roots in both the rule schema and the host graph. This is to be expected since the number of roots available for matching will increase the number of matches. Indeed, if all nodes were roots, then rooted matching would be identical to conventional graph matching. In practice, we aim to limit the number of roots. For example, in our case study in the next section, we use only one root in both rule schemata and host graphs.

6 Case Study: 2-Colouring

Vertex colouring has many applications [11] and is among the most frequently considered graph problems. We focus on 2-colourability: a graph is *2-colourable*, or *bipartite*, if we can assign one of two colours to each node such that the source and target of each edge have different colours.

The GP program `2colouring` in Figure 3 expects a connected and unmarked input graph G with atomic node labels and a single root. The program will either produce a 2-colouring for G by appending the integer 0 or 1 to each node label, or return G unmodified if no 2-colouring exists. For the rest of this section, by a *rooted graph* we mean a connected graph with a single root.

In Figure 3, the roots in rule schemata are depicted with a thick border. For notational convenience, the rule schemata `colour`, `illegal` and `back` contain bidirectional edges. Each of these rule schemata actually represents a set of two distinct rule schemata with normal edges such that the edge direction is the same in the left- and right-hand side. For example, `colour` stands for the set {`colour1`, `colour2`} where `colour1` and `colour2` differ only by the edge direction. When `colour` is called by the main program, `colour1` or `colour2` is selected nondeterministically and applied. If it is not applicable, then the other rule schema is attempted.

The program traverses a host graph in depth-first order, starting at the unique root which is coloured with 0. Whenever an edge is encountered whose source or target has a colour i and whose other node is uncoloured, then the other node is coloured with $1-i$. If `colour` is no longer applicable, then the rule schema `back` moves the root one position back on the path of coloured nodes and the colouring process starts anew.

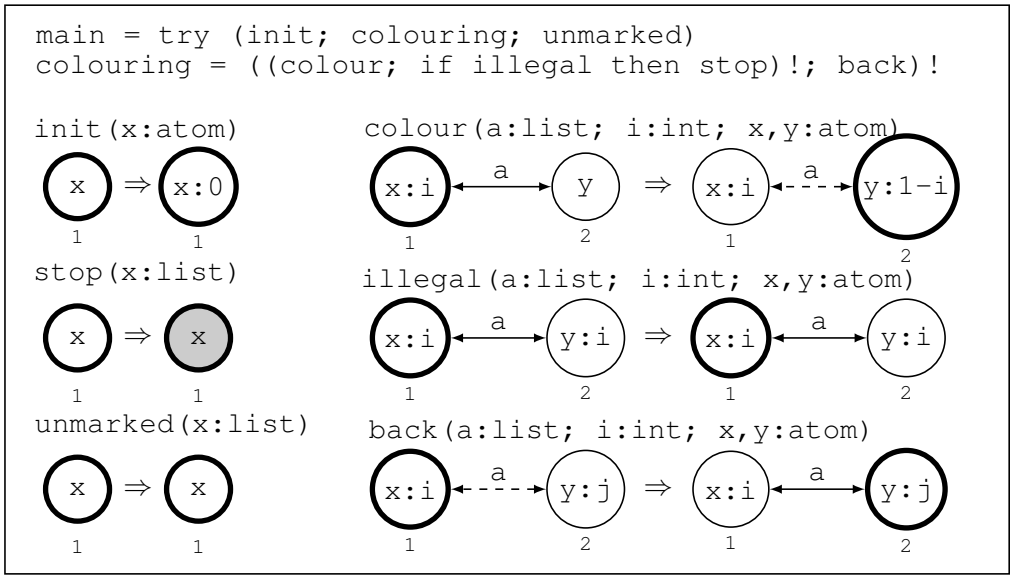

Figure 3: GP program `2colouring`

Upon termination of the macro `colouring`, the rule schema `unmarked` checks whether the root is unmarked or not. If not, then the rule schema `illegal` has detected an edge whose ends have the same colour and `stop` has marked the root. In this case the input graph is not bipartite and by the semantics of the `try` command, the input graph is returned as the application of `unmarked` failed. On the other hand, if the root is unmarked, then the whole graph has been correctly coloured.

Proposition 2 (Correctness of `2colouring`) *Given a rooted input graph G with atomic node labels, the program `2colouring` returns a 2-coloured version of G if G is bipartite, otherwise it returns G unchanged.*

We omit the proof for lack of space. Note that each of the nine rule schemata can only be applied at the unique root of the current graph. Therefore the root needs to be moved around, which happens with both `colour` and `back`. However, care must be taken to prevent the root being moved back and forth between the same nodes forever. The program avoids this kind of looping by marking an edge only when an incident node gets a colour and unmarking this edge when `back` is applied to it (without altering the colours).

We now analyse the time complexity of `2colouring`. First note that all rule schemata are fast in the sense of Definition 1. Hence, by Corollary 1, we know that each rule schema takes only constant time on rooted graphs of bounded degree. Moreover, none of the rule schemata increases any node degree or the number of roots. Hence repeated rule schema applications in program runs preserve the assumptions of Corollary 1.

Then, to show that the running time of `2colouring` is linear in the size of the input graph, it suffices to show that the maximal number of rule schema applications is linear. This argument takes into account the linear overhead of the `try` command, which can be implemented by

copying the input graph and returning the copy in case the command sequence fails.

As to the number of rule schema applications, observe first that the rule schemata `init`, `unmarked`, `illegal` and `stop` can be ignored because each of them is applied at most once in a program run. Next, we notice that `colour` reduces the number of uncoloured nodes and `back` does not increase this number. Hence `colour` is applied at most n times, where n is the number of nodes in the input graph. Moreover, `back` cannot be applied more often than `colour` because the input graph is unmarked, only `colour` creates an edge mark, and `back` removes one edge mark. Thus, there are at most $2n$ applications of `colour` and `back`. Altogether, we have shown the following.

Proposition 3 (Time complexity of `2colouring`) *On rooted input graphs with atomic node labels and bounded node degree, the running time of `2colouring` is linear in the size of graphs.*

This is significantly better than what can be achieved with unrooted programs. For, in the worst case of rule schema matching, even a clever algorithm requires at least linear time as it needs to search the complete host graph. Since each node of a bipartite input graph gets coloured, it follows that such a program has at least quadratic running time.

7 Conclusion

We have presented an approach for programming with graph transformation rules in which the bottleneck of graph transformation—the inefficiency of graph matching—is circumvented by using rooted rules which only match in the neighbourhood of host graph roots. Rooted graph transformation has been cleanly embedded in the framework of the double-pushout approach and has been extended to rule schemata in the graph programming language GP.

We have shown an algorithm which matches a large class of rooted conditional rule schemata in constant time, provided that host graphs have bounded node degrees. Our case study demonstrates that algorithms of practical importance, such as graph colouring, can be implemented with rooted GP programs whose time complexity is as good as that of programs in imperative languages. Moreover, we have demonstrated that due to the simplicity of GP and its semantics, the correctness and complexity of rooted graph programs is amenable to formal analysis. Essentially, because fast rule schemata can be applied in constant-time, to show that a program with fast rule schemata has a certain time complexity T, it suffices to prove that the maximal number of rule schema applications is of order T.

In future work, we will consider alternative sufficient conditions that make rooted programs fast. For example, in [4] it is shown that graph transformation rules can be applied in constant time if the outdegree of nodes in host graphs is bounded and the left-hand sides of rules contain a directed path from the root to each node. A corresponding result should hold for fast rule schemata if each node in a left-hand side is reachable from some root by a directed path.

Another topic for future work is the complexity of rooted graph matching and rooted graph programs on host graphs with unbounded node degrees. The 2-colouring problem, for example, may be simple enough to construct a graph program that runs in linear time on arbitrary (single-rooted) host graphs.

Finally, we will aim at confirming our theoretical results by implementing a rooted version of GP and comparing the performance of graph programs with that of programs in traditional programming languages.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] M. Dodds. *Graph Transformation and Pointer Structures*. PhD thesis, The University of York, 2008.

[3] M. Dodds and D. Plump. Extending C for checking shape safety. In *Proc. Graph Transformation for Verification and Concurrency (GT-VC 2005)*, volume 154(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2006.

[4] M. Dodds and D. Plump. Graph transformation in constant time. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 367–382. Springer-Verlag, 2006.

[5] H. Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[6] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer-Verlag, 2006.

[7] A. Habel and D. Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.

[8] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

[9] D. Plump. The graph programming language GP. In *Proc. International Conference on Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.

[10] D. Plump. The design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012.

[11] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, second edition, 2008.

[12] M. von Detten, C. Heinzemann, M. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story diagrams — syntax and semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2012.